SÉANCE 3

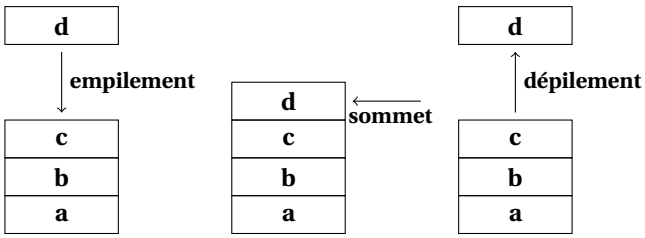
RAPPELS SUR LES PILES, FILES ET DICTIONNAIRES

I - Notion de pile

Une ***pile*** est une structure de données qui reprend l'idée d'une pile d'assiettes : on peut poser une assiette sur cette pile ou reprendre la dernière assiette posée. Les données vont donc être «empilées» et seule la dernière entrée est directement récupérable – l'accès aux éléments inférieurs de la pile n'est pas possible aussi rapidement. On parle de structure LIFO (*last in, first out*).
Les piles sont omniprésentes en informatique. Des exemples simples sont par exemple l'historique d'un navigateur web, le stockage des actions dans un éditeur de texte pour offrir la possibilité à l'utilisateur de revenir en arrière,...

Les fonctions usuelles de manipulation sont :

- la *création* d'une pile vide ;
- le test indiquant si l'on a une *pile vide* ;
- l'*empilement* d'un nouvel élément (en anglais, *push*) ;
- le *dépilement* de l'élément situé au sommet de la pile (en anglais, *pop*).

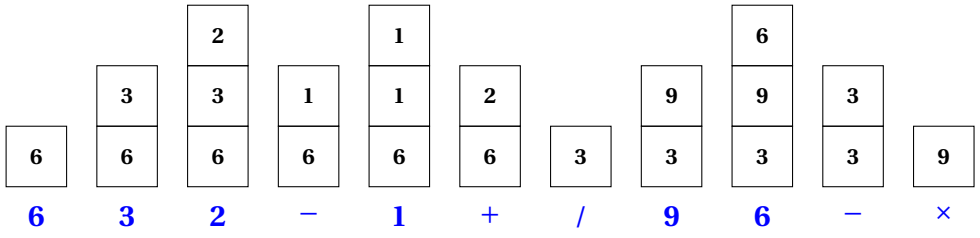


Exemple : évaluation d'une expression arithmétique

De façon usuelle, on note l'opérateur entre les deux opérandes (on écrit $2+3$ pour la somme des entiers 2 et 3) mais il existe d'autres notations. La notation polonaise inverse consiste à écrire l'opérateur après (on écrit 2 3 + pour la somme des entiers 2 et 3). L'intérêt de cette notation est qu'elle fait l'économie des parenthèses ; par exemple, au lieu d'écrire $(1+2)\times(3-4)$, on écrit 1 2 + 3 4 − ×.
Pour évaluer une expression écrite en notation polonaise inverse, on peut utiliser une pile :

- on lit de façon linéaire la succession de symboles ;
- on empile les entiers que l'on rencontre ;
- lorsque l'on rencontre un opérateur, on dépile les deux derniers entiers, on applique l'opérateur et on empile le résultat ;
- à la fin de la lecture de la succession de symboles, la pile contient exactement un entier qui est le résultat de l'évaluation de l'expression.

Illustrons cela avec l'expression 6 3 2 − 1 + / 9 6 − × :



On peut réaliser une pile en Python à l'aide du type `list`. En effet, il suffit de considérer les fonctions données ci-dessous :

```
def creerPile():
    return []

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

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

def depiler(p):
    if estVide(p):
        return "pile vide"
    else:
        return p.pop()
```

Malgré la pertinence des listes pour implémenter la notion de Pile en Python, il convient de s'exercer à n'utiliser que les quatre fonctions de bases sans exploiter la structure sousjacente.

Exemples

1 ► Écrire une fonction `sommet(p)` renvoyant le sommet d'une pile `p` sans la modifier.

2 ► Écrire une fonction qui prend une pile en argument et qui échange les deux éléments au sommet de la pile (sauf s'il y a moins de deux éléments dans la pile auquel cas elle est inchangée).

3 ► Écrire une fonction donnant la «taille» d'une pile sans la détruire.

II - Notion de file

Une structure voisine de celle de pile est la structure de ***file*** qui reprend l'idée d'une file d'attente (sans priorité) : on entre dans la file et on attend que tous ceux arrivés avant passent. On parle de structure FIFO (*first in, first out*).
Les fonctions usuelles de manipulation sont :

- la *création* d'une file vide ;
- le test indiquant si l'on a une *file vide* ;
- l'*enfilement* d'un nouvel élément (le nouvel élément est ajouté en queue de file) ;
- le *défilement* de l'élément situé en tête de la file.

Exemple

On a vu lors du premier TP que l'on appelle *nombre de Hamming*, tout entier naturel n de la forme $2^a3^b5^c$ avec $(a,b,c) \in \mathbb{N}^3$; les premiers sont :

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24.$$

L'une des méthodes abordées reposait sur l'idée du programme suivant :

```
def g(L2, L3, L5) :
    n = min(L2[0], L3[0], L5[0])
    if n == L2[0]:
        L2.pop(0)
    if n == L3[0]:
        L3.pop(0)
    if n == L5[0]:
        L5.pop(0)
    L2.append(2*n)
    L3.append(3*n)
    L5.append(5*n)
    return n

L2, L3, L5 = [1], [1], [1]
LH = []
for no in range(15) :
    LH.append(g(L2, L3, L5))
```

Dans ce qui précède, les listes L2, L3 et L5 sont utilisées comme des files qui «stockent» les doubles, les triples et les quintuples «en attente».

Comme dans l'exemple ci-dessus, les listes Python permettent d'implémenter la structure de file. En effet, il suffit de considérer les fonctions données ci-dessous :

```
def creerFile():
    return []

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

def enfiler(f, x):
    f.append(x)

def defiler(f):
    if estVide(f):
        return "file vide"
    else:
        return f.pop(0)
```

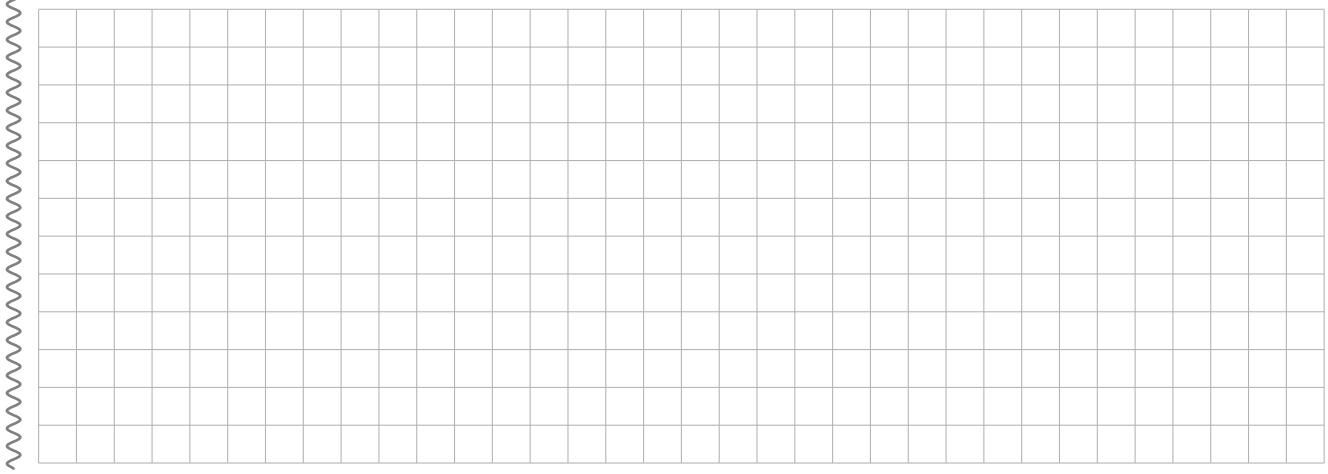
Il convient néanmoins de remarquer que la méthode `pop(0)` n'est pas de complexité anodine. On est préférable d'utiliser l'instruction `deque` du module `collection`. Il suffit par exemple de considérer les fonctions données ci-dessous :

```
from collections import deque

def creerFile():
    return deque()

def estVide(f):
    return f == deque([])

def enfiler(f, x):
    return f.appendleft(x)

def defiler(f):
    return f.pop()
```

Comme pour les files, il convient de s'exercer à n'utiliser que les quatre fonctions de bases sans exploiter la structure sousjacente.

Exemple

Écrire une fonction `consultation(f)` renvoyant l'élément en tête d'une file sans modifier la file.



III - Notion de dictionnaire

Les dictionnaires sont des structures de données formées d'enregistrements repérés chacun par un élément particulier appelé *clé*, différent d'un enregistrement à l'autre, et qui doit être de type non mutable (un entier ou une chaîne de caractères par exemple, les clés peuvent être de types différents). L'enregistrement associé à une clé donnée peut, lui, être de n'importe quel type (entier, liste, tuple...) et peut être modifié (*i.e.* est mutable).
Contrairement à une liste où les données sont repérées par un indice numéroté à partir de 0, il n'y pas d'ordre prédéfini pour les différents enregistrements d'un dictionnaire, la machine organisant en interne les données pour accélérer les accès en fonction des clés. Pour l'utilisateur, un dictionnaire est une collection de données stockées «en vrac» dans laquelle il est facile d'ajouter ou de supprimer un élément.
À ce stade de l'année, on ne s'intéresse pas à la façon dont fonctionne un dictionnaire.

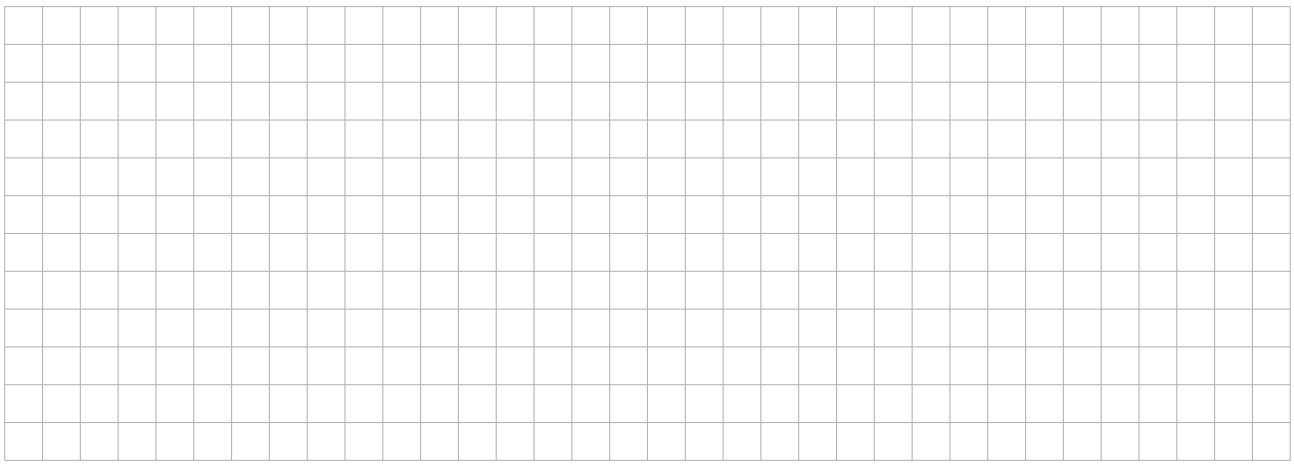
On crée un dictionnaire en mettant entre accolades `{ }` une suite d'enregistrements `clé : valeur` (et le dictionnaire vide s'obtient par `{}`) ou en transformant une liste `L` de listes à deux éléments à l'aide de `dict(L)`.
On accède à l'enregistrement associé à une clé `key` d'un dictionnaire `D` par `D[key]` (ce qui permet aussi éventuellement de créer l'enregistrement). Enfin, il convient de connaître également le moyen de tester si une clé apparaît dans le dictionnaire à l'aide de `in`
Enfin, il est utile de connaître les méthodes `items` et `values`.

```
>>> d1 = {'pcsi1':'DE004', 'pcsi2':'C204', 'mp2i': ['DE002','C205'], 'C011':['ECG11','ECG21']}
>>> d1['pcsi2']
'C204'
>>> 'pcsi1' in d1
False
>>> d1.items()
dict_items([('pcsi1', 'DE004'), ('pcsi2', 'C204'), ('mp2i', ['DE002', 'C205']), ('C011', ['
    ECG11', 'ECG21'])])
>>> d1.values()
dict_values(['DE004', 'C204', ['DE002', 'C205'], ['ECG11', 'ECG21']])
>>> d2 = dict([(i,chr(i)) for i in range(97,100)])
>>> print(d2)
{97: 'a', 98: 'b', 99: 'c'}
>>> d2[100] = 'd'
>>> print(d2)
{97: 'a', 98: 'b', 99: 'c', 100: 'd'}
```

Exemple

1 ► À l'aide d'un dictionnaire, écrire une fonction `compte(ch)` prenant en argument une chaîne de caractères `ch` et comptant le nombre d'occurrences de chaque caractère. Par exemple, on doit avoir :

```
>>> compte('sapristi, voici venir ici des ex-pcsi !')
{'s': 4, 'a': 1, 'p': 2, 'r': 2, 'i': 8, 't': 1, ',': 1, ' ': 6, 'v': 2, 'o': 1, 'c':
    3, 'e': 3, 'n': 1, 'd': 1, 'x': 1, '-': 1, '!': 1}
```



2 ► On considère un dictionnaire `D` représentant un dictionnaire de traduction *anglais vers français* : les clés sont des mots anglais et les valeurs sont la liste des traductions possibles des mots. Écrire le programme qui crée à partir de `D` le dictionnaire *français vers anglais* associé. On pourra tester avec :

```
>>> D = {'make' : ['faire', 'fabriquer', 'réaliser'], 'do' : ['faire'],
         'realize' : ['réaliser', 'prendre conscience', 'parvenir'],
         'produce' : ['réaliser', 'produire', 'fabriquer']}
```