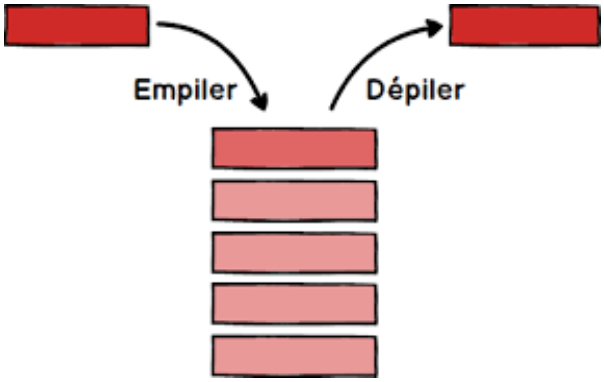
Chapitre 13 - Piles et Files

Dans ce chapitre nous allons décrire des structures de données linéaires appelées **listes**, dont nous verrons deux formes restreintes *très efficaces* : les **piles** et les **files**.

1- Les PILES :

Les **piles** (*stacks* en anglais) correspondent exactement à la notion de pile dans la vie courante :

- Une pile de cartes,
- Une pile d'assiettes…



Pour ajouter un élément on l'empile, il se retrouve donc au-dessus, et pour retirer un élément, on ne peut retirer **que l'élément se trouvant au sommet de la pile**.

En anglais on dit *last in, first out* ou *LIFO* pour dire « dernier arrivé, premier sorti ».

Ce type de structure de données est par exemple utilisé dans :

- les éditeurs avec la fonction Annuler (CTRL+Z)
- les navigateurs pour reculer d'une page.

On étudiera plus tard d'autres cas d'utilisation, par exemple dans la recherche d'éléments stockés dans des **arbres** ou des **graphes** et pour l'exécution des fonctions **récursives**.

a. Implementation en python :

Afin de pouvoir utiliser cette structure de donnée, on peut créer une classe ***Pile*** qui utilisera une liste python. Avec la fonction *len()* et les méthodes natives de python *append()* et *pop()*, on pourra créer d'autres méthodes qui permettront de réaliser les actions suivantes sur un objet de cette classe ***Pile*** :

- « Empiler » : ajoute un élément sur la pile. Le terme anglais correspondant est *push*.
- « Dépiler » : enlève un élément de la pile et **le renvoie**. Le terme anglais correspondant est *pop*.
- « La pile est-elle vide ? » : renvoie vrai si la pile est vide, faux sinon.
- « Nombre d'éléments de la pile » : renvoie le nombre d'éléments dans la pile.
- « Quel est l'élément de tête ? » : renvoie l'élément de tête sans le dépiler. Le terme anglais correspondant est ***peek*** ou ***top***.
- « Vider la pile » : dépiler tous les éléments. Le terme anglais correspondant est ***clear***.

Par exemple, pour obtenir la pile ci-contre et dont le contenu correspond à un historique de navigation, on exécuterait :

```
Etat de la pile Historique de navigation :

|   https://www.google.com/     |
|   https://www.youtube.com/    |
| https://www.lyceebranly.com/  |
|  https://www.nsibranly.fr/    |
---------------------------------
```

```
p = Pile('Historique de navigation')
p.empiler('https://www.nsibranly.fr/')
p.empiler('https://www.lyceebranly.com/')
p.empiler('https://www.youtube.com/')
p.empiler('https://www.google.com/')
print(p)
```

Pour tester la méthode *estVide()*, on pourrait exécuter dans la console :

```
>>> p.estVide()
False
```

Le booléen False est retourné

Pour tester la méthode *depiler()*, on pourrait exécuter dans la console :

L'élément au sommet de la pile est est retourné

Cet élément est retiré de la pile

```
>>> p.depiler()
'https://www.google.com/'

>>> print(p)

Etat de la pile Historique de navigation :

|   https://www.youtube.com/    |
| https://www.lyceebranly.com/ |
|  https://www.nsibranly.fr/    |
--------------------------------
```

Pour tester la méthode *taille()*, on pourrait exécuter dans la console :

```
>>> p.taille()
3
```

Le nombre d'élément de la pile est retourné

Pour tester la méthode *sommet()*, on pourrait exécuter dans la console :

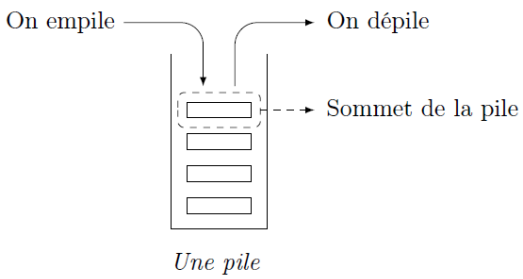
```
>>> p.sommet()
'https://www.youtube.com/'
```

L'élément au sommet de la pile est retourné

Pour tester la méthode *vider()*, on pourrait exécuter dans la console :

```
>>> p.vider()

>>> print(p)

Etat de la pile Historique de navigation :

|  |
----
```

b. EXERCICE TYPE BAC (2019) : On considère une pile nommé P.



Une pile

Question 1 : On suppose dans cette question que le contenu de la pile P est celui donné ci-contre (les éléments étant empilés par le haut). Donner ci-dessous le code python qui permet de créer et de remplir P :

4
2
5
8

```
P = Pile("P")
l = [8,5,2,4]
for val in l :
    P.empiler(val)
```

Question 2 : Quel sera le contenu de la pile Q après exécution de la suite d'instructions suivante ?

```
Q = Pile("Q")
while not P.estVide() :
    val = P.depiler()
    Q.empiler(val)
```

```
Etat de la pile Q :

| 8 |
| 5 |
| 2 |
| 4 |
-----
```

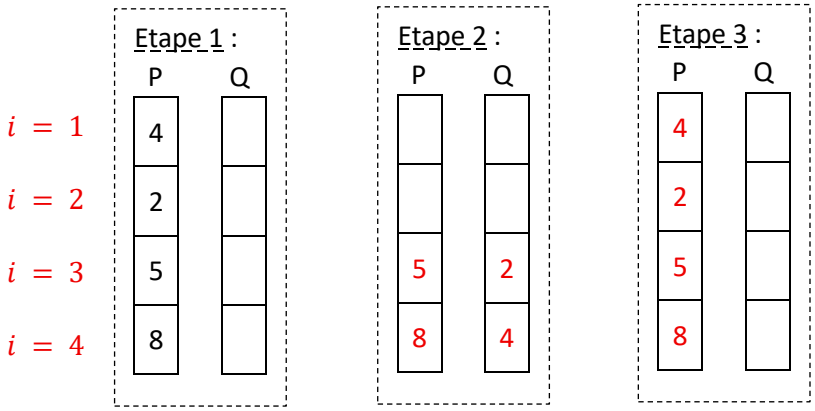
Question 3 : On donne ci-contre le script de la méthode *maxPile()*. Elle a comme paramètre un entier i. Elle renvoie la position j de l'élément maximum parmi les i derniers éléments empilés de la pile P. Après appel de cette fonction, la pile P retrouve son état d'origine. La position du sommet de la pile est 1.

```
def maxPile(self,i) :
    Q = Pile('Q')
    j = 1
    while j <= i:
        x = self.depiler()
        Q.empiler(x)
        if j == 1 :
            max = x
            jMax = j
        elif x > max :
            max = x
            jMax = j
        j = j + 1
    while not Q.estVide():
        x = Q.depiler()
        self.empiler(x)
    return jMax
```

Pour la pile P donnée ci-contre, l'exécution de cette méthode avec comme argument l'entier 2, renvoie la valeur 1 :

```
>>> P.maxPile(2)
1
```

Compléter ci-dessous le contenu des piles P et Q au cours de l'exécution de `P.maxPile(2)` :

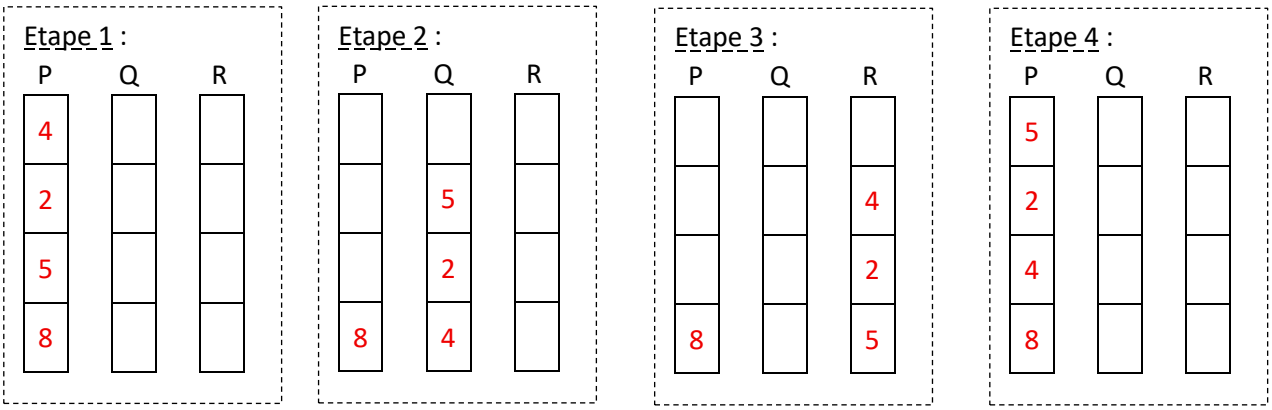


La valeur retournée est $jMax = 1$

Question 4 : Compléter la méthode *retourner()* ayant pour paramètres un entier j. Cette méthode inverse l'ordre des j derniers éléments empilés et ne renvoie rien. On pourra utiliser deux piles auxiliaires. Par exemple, si P est la pile donnée ci-contre à gauche, après l'appel de *P.retourner(3)*, l'état de la pile P sera celui donné ci-contre à droite :


```
def retourner(self,j):
    Q = Pile('Q')
    R = Pile('R')
    n = 1
    while n <= j:
        x = self.depiler()
        Q.empiler(x)
        n = n + 1
    while not Q.estVide():
        x = Q.depiler()
        R.empiler(x)
    while not R.estVide():
        x = R.depiler()
        self.empiler(x)
```

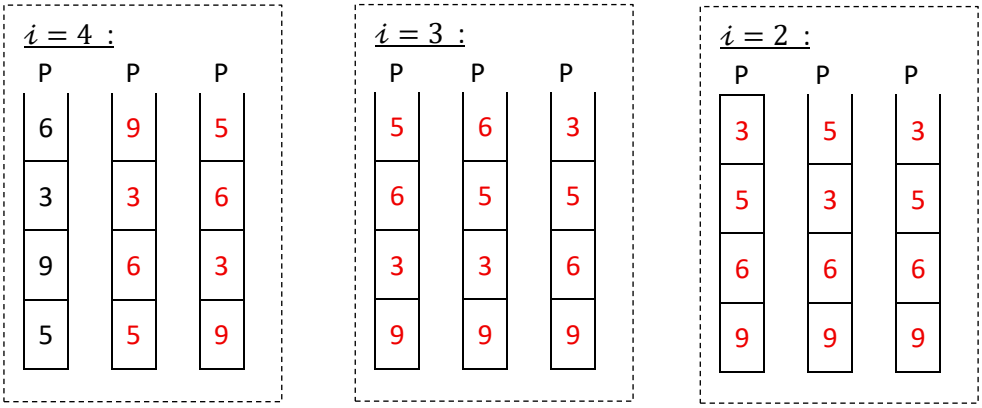
Compléter ci-dessous le contenu des piles P et Q au cours de l'exécution de `>>> P.retourner(3)` :



Question 5 : L'objectif de cette question est de trier une pile de crêpes.
On modélise une pile de crêpes par une pile d'entiers représentant le diamètre de chaque crêpe. On souhaite réordonner les crêpes de la plus grande (placée en bas de la pile) à la plus petite (placée en haut de la pile). On dispose uniquement d'une spatule que l'on peut insérer dans la pile de crêpes de façon à retourner l'ensemble des crêpes qui lui sont au-dessus.
Le principe est le suivant :

- On recherche la plus grande crêpe.
- On retourne la pile à partir de cette crêpe de façon à mettre cette plus grande crêpe tout en haut de la pile.
- On retourne l'ensemble de la pile de façon à ce que cette plus grande crêpe se retrouve tout en bas.
- La plus grande crêpe étant à sa place, on recommence le principe avec le reste de la pile.

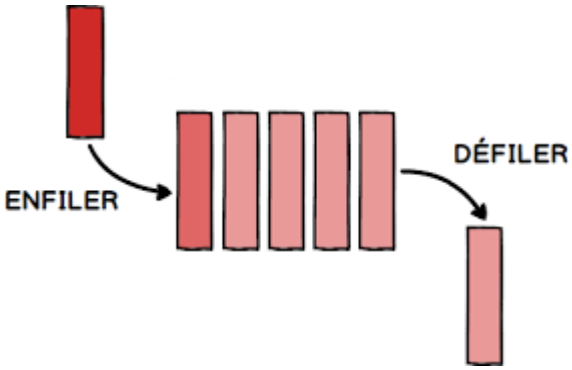
Par exemple :



On donne ci-contre le script de la méthode *triCrepes()*. Elle trie la pile P selon la technique du tri crêpes et ne renvoie rien.
Ecrire sur le schéma ci-dessus, les valeurs des entiers i et j pour chacune des étapes.

```
def triCrepes(self) :
    i = self.taille()
    while i > 1 :
        j = self.maxPile(i)
        self.retourner(j)
        self.retourner(i)
        i = i - 1
```

2- LES FILES :



Les **files**(*queues* en anglais) correspondent également à la notion de file dans la vie courante:

- Une file d'attente à la caisse,
- à un feu rouge…

Lorsqu'on ajoute un élément, celui-ci se retrouve à la fin de la file, et on **retire les éléments dans l'ordre dans lequel ils sont arrivés**. En anglais on dit *first in, first out* ou *FIFO* pour dire: **premier arrivé premier sorti**.

Ce type de structure de données est par exemple utilisé dans :

- Un gestionnaire d'impression pour ordonner l'ordre des impressions.
- Un processeur pour planifier l'ordre des opérations.
- Un serveur web pour ordonner les réponses en fonction de l'ordre des demandes.

a. IMPLEMENTATION EN PYTHON :

Afin de pouvoir utiliser cette structure de donnée, on pourrait créer, à l'image de ce que l'on a fait pour le cas de la **Pile**, une classe ***File*** qui utiliserait une liste python. On choisit ici de faire autrement en utilisant plutôt un paradigme de programmation fonctionnelle. On se propose ainsi de créer ici des fonctions qui permettront de réaliser les actions suivantes :

- « Création d'une file » : **renvoie** une file vide.
- « Enfiler » : ajoute un élément dans la file. Le terme anglais correspondant est *enqueue*.
- « Défiler » : **renvoie** l'élément en tête de la file, et le retire de la file. Le terme anglais correspondant est *dequeue*.
- « La file est-elle vide ? » : renvoie « vrai » si la file est vide, « faux » sinon.
- « Nombre d'éléments dans la file » : renvoie le nombre d'éléments dans la file.
- « Tête de la file » : renvoie l'élément qui se trouve en tête de la file
- « Vider la file » : supprimer les éléments qui sont dans la file. Le terme anglais correspondant est ***clear***.

Par exemple, pour obtenir la file ci-contre et dont le contenu correspond à celui du spooler d'une imprimante qui doit ici encore imprimer 3 fichiers, on exécuterait :

```
Etat de la file Spooler imprimante Epson :

--------------------------------------------
 -> --informatique.docx--memoire.doc--nsi.jpg
--------------------------------------------
```

```
f = creer_file_vide()
enfiler(f,'nsi.jpg')
enfiler(f,'memoire.doc')
enfiler(f,'informatique.docx')
afficher(f,'Spooler imprimante Epson')
```

Pour tester la fonction *est_vide()*, on pourrait exécuter dans la console :

```
>>> est_vide(f)
False
```

Pour tester la fonction *defiler()*, on pourrait exécuter dans la console :

```
>>> defiler(f)
'nsi.jpg'

>>> afficher(f,'Spooler imprimante Epson')

Etat de la file Spooler imprimante Epson :

-----------------------------------
 -> --informatique.docx--memoire.doc
-----------------------------------
```

Pour tester la fonction *taille()*, on pourrait exécuter dans la console :

```
>>> taille(f)
2
```

Pour tester la fonction *tete()*, on pourrait exécuter dans la console :

```
>>> tete(f)
'memoire.doc'

>>> afficher(f,'Spooler imprimante Epson')

Etat de la file Spooler imprimante Epson :

-----------------------------------
 -> --informatique.docx--memoire.doc
-----------------------------------
```

Pour tester la fonction *vider()*, on pourrait exécuter dans la console :

```
>>> vider(f)

>>> afficher(f,'Spooler imprimante Epson')

Etat de la file Spooler imprimante Epson :

----
 -> --
----
```

b. EXERCICE TYPE BAC (2022):

« *Simon* » est un jeu de société électronique de forme circulaire comportant quatre grosses touches de couleurs différentes : rouge, vert, bleu et jaune. Le jeu joue une séquence de couleurs que le joueur doit mémoriser et répéter ensuite. S'il réussit, une nouvelle couleur est ajoutée à la fin de la séquence. La nouvelle séquence est jouée depuis le début et le jeu continue. Dès que le joueur se trompe, la séquence est vidée et réinitialisée avec une couleur et une nouvelle partie commence.

Question 1 : La fonction *ajout()* ci-contre est incomplète. Elle prend en paramètre une file f contenant déjà une séquence de couleurs. Après exécution, elle renvoie cette file avec une couleur en plus, définie aléatoirement. Compléter ce script.

```
def ajout(f) :
    couleurs = ['b','r','j','v']
    i = randint(0,3)
    enfiler(f,couleurs[i])
```

Exemple d'exécution dans la console :

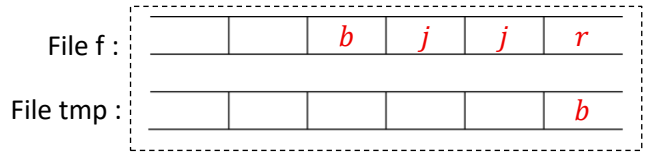
```
>>> f = creer_file_vide()

>>> ajout(f)

>>> ajout(f)

>>> afficher(f,'f')

Etat de la file f :

----------
 -> --j--r
----------
```

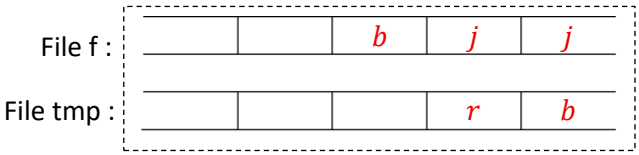
Question 2 : La fonction *afficher_seq()* ci-contre prend en paramètre une file f. Elle permet d'ajouter une nouvelle couleur à la file et ensuite, d'afficher toutes les couleurs de la file, une par une, avec une temporisation de 1 s entre chaque affichage. Exemple d'exécution dans la console :

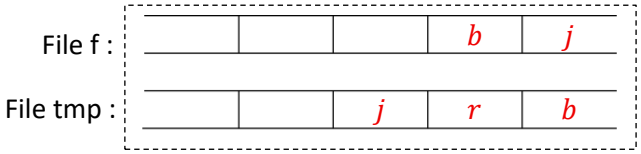
```
def affich_seq(f) :
    tmp = creer_file_vide()
    ajout(f)
    while not est_vide(f):
        c = defiler(f)
        print(c)
        time.sleep(1)
        enfiler(tmp,c)
    while not est_vide(tmp) :
        enfiler(f,defiler(tmp))
```

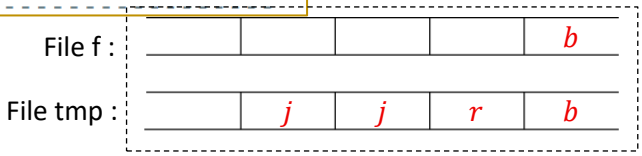
```
>>> afficher(f,'f')

Etat de la file f :

----------------
 -> --j--j--r--b
----------------

>>> affich_seq(f)
b
r
j
j
b

>>> afficher(f,'f')

Etat de la file f :

------------------
 -> --b--j--j--r--b
------------------
```

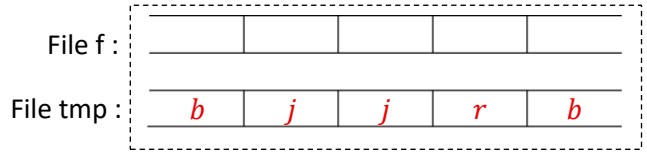
Pour cette exécution, compléter ci-dessous le contenu des files f et tmp :

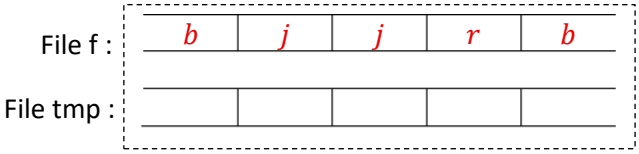












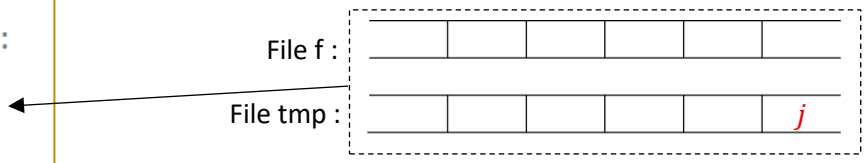
Question 3 : La fonction tour_de_jeu() ci-contre prend en paramètre une file f. Elle permet de gérer le déroulement du jeu.

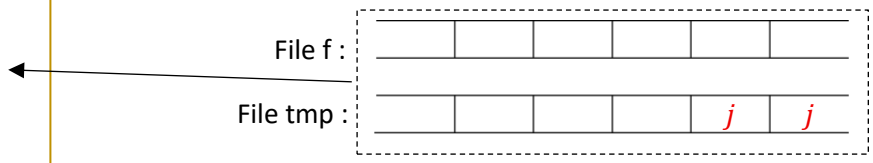
```
def tour_de_jeu(f):
    while True :
        print("\nSéquence : ")
        affich_seq(f)
        tmp = creer_file_vide()
        nb = taille(f)
        print(f"{nb} couleur(s) à saisir : ")
        while not est_vide(f):
            c_joueur = input(f"Couleur {1+taille(tmp)} à saisir : ")
            c_seq = defiler(f)
            if c_joueur == c_seq:
                enfiler(tmp, c_seq)
            else:
                vider(f)
                vider(tmp)
A ->            if c_joueur!= "" : print('\nERREUR, nouvelle partie')
        while not est_vide(tmp):
                enfiler(f, defiler(tmp))
        if c_joueur == "" :
            print('\nFIN')
            return
```

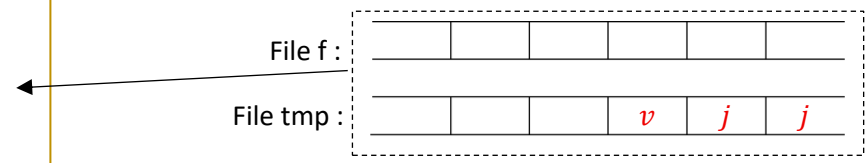
Exemple d'exécution dans la console :

```
>>> tour_de_jeu(f)

Séquence :
j
1 couleur(s) à saisir :
Couleur 1 à saisir : j

Séquence :
j
j
2 couleur(s) à saisir :
Couleur 1 à saisir : j
Couleur 2 à saisir : j

Séquence :
j
j
v
3 couleur(s) à saisir :
Couleur 1 à saisir : j
Couleur 2 à saisir : j
Couleur 3 à saisir : v

Séquence :
j
j
v
r
4 couleur(s) à saisir :
Couleur 1 à saisir : j
Couleur 2 à saisir :

FIN
```

Donner le contenu des files f et tmp sur la ligne repérée A, aux endroits indiqués :







3-CONCLUSION :

Pourrait-on faire du code sans ces concepts de **Piles** ou de **Files** ? Oui bien sûr, mais ces outils permettent de penser différemment son algorithme. Ils peuvent se révéler très efficaces dans différents cas de figures que l'on découvrira encore, dans la suite du programme de Nsi.