fiche 2

Piles et files

La résolution des exercices se fera en utilisant la définition des piles et des files vu en cours. Seules les méthodes liées à celle-ci seront utilisées.

Exercice 1 : On considère une pile p initialement vide de type LIFO. Donner l'état de la pile à chaque étape :

- $p =$ `creer_pile`()
- p.`empile`(5)
- p.`empile`(3)
- p.`empile`(7)
- p.`depile`()
- p.`depile`()
- p.`empile`(4)
- p.`empile`(2)
- p.`depile`()
- p.`depile`()

Exercice 2 :

1. Écrire la fonction `affiche` qui prend en argument une pile, et retourne la chaine de caractère permettant un affichage de la pile.
   Exemple :

```
>>> p = Pile()
>>> p.empile(8)
>>> p.empile(15)
>>> p.empile(12)
>>> print(affiche(p))
|12|
|15|
|8|
```

2. Écrire la fonction `creation_pile` qui prend en argument une liste et retourne la pile dont les éléments sont stockés dans l'ordre de leur lecture dans la pile. Exemple :

```
>>> p = creation_pile([8,1,5,9])
>>> print(affiche(p))
|9|
|5|
|1|
|8|
```

3. Écrire la fonction `taille` qui prend en argument une pile, et renvoie le nombre d'élément qu'elle contient. (la pile ne doit pas finir vide à l'issue d'appel de la fonction.)

4. Écrire la fonction `separation` qui prend en argument une pile d'entier et qui sépare dans deux piles les nombres paires et impaires.

```
>>> p = creation_pile([8,1,2,4,6,4,9])
>>> (p_0,p_1) =separation_pile(p)
>>> print(affiche(p_0))
|8|
|2|
|4|
|6|
|4|

>>> print(affiche(p_1))
|1|
|9|
```

Exercice 3 : **Bon parenthésage :**
Écrire une fonction `bon_parenthesage` qui prend en argument une chaine de caractère représentant une expression algébrique, et vérifie à l'aide d'une pile le bon parenthésage de l'expression.

Exemples 1 : :

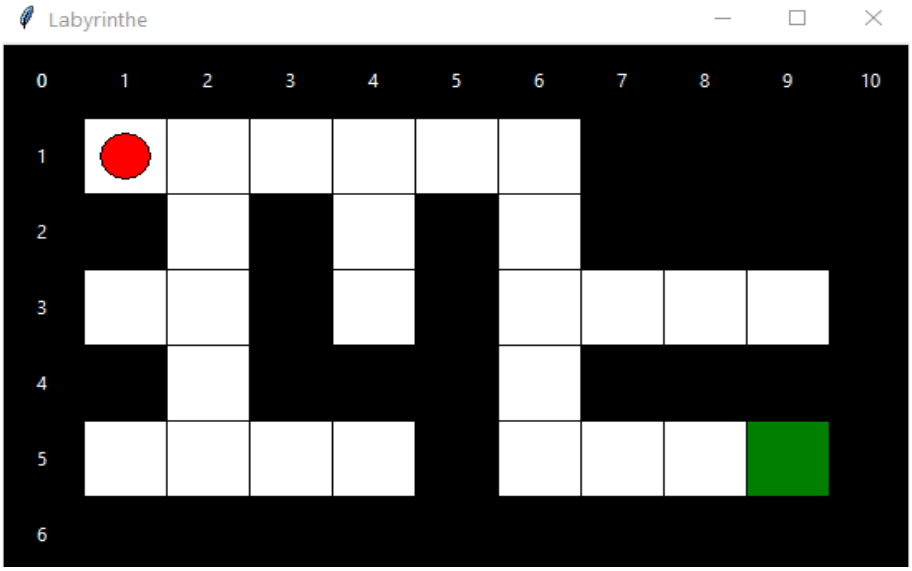
```
>>> print(bon_parenthesage ( "(8(94+5)))-8)*(5(75+(2-56)))"))
False
>>> print(bon_parenthesage ( "(8(94+5)-8)*(5(75+(2-56)))"))
True
```

Exercice 4 : Pour la résolution du labyrinthe en profondeur, nous allons considérer une pile P contenant initialement la position de départ (ici $(1, 1)$).
L'algorithme est le suivant :

- On dépile P.
- On note alors comme visitée la case obtenue.
- On empile les cases voisines accessibles non déjà visitées.

On réitère tant que la case n'est pas l'arrivée (ici la case $(5, 9)$). Si la pile est vide, alors il n'y a pas d'issue à notre labyrinthe.

On considère le labyrinthe suivante :



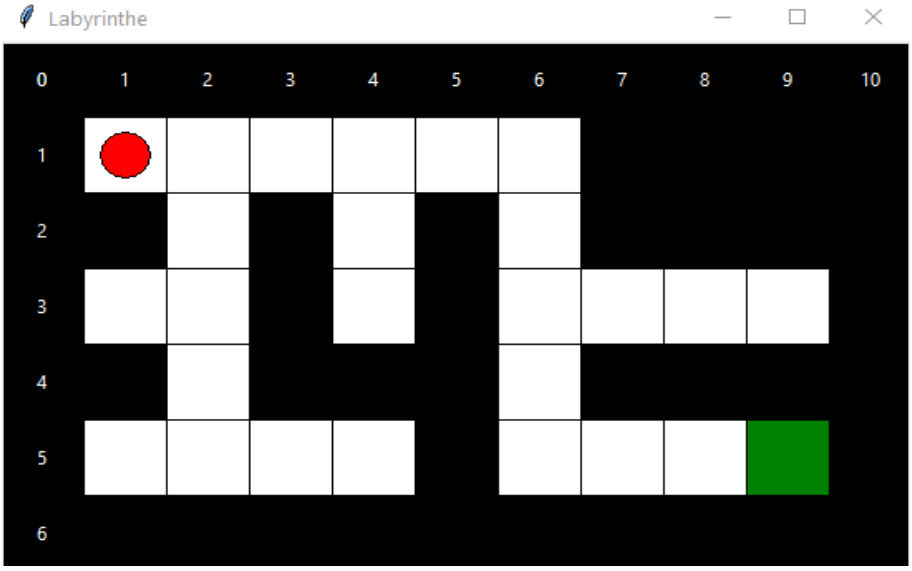
A la main, procédez à la résolution du labyrinthe en détaillant l'état de la pile.

Exercice 5 : Pour la résolution du labyrinthe en largeur, nous allons considérer une file F contenant initialement la position de départ (ici $(1, 1)$).
L'algorithme est le suivant :

- On défile P.
- On note alors comme visitée la case obtenue.
- On enfile les cases voisines accessibles non déjà visitées.

On réitère tant que la case n'est pas l'arrivée (ici la case $(5, 9)$). Si la file est vide, alors il n'y a pas d'issue à notre labyrinthe.

On considère le labyrinthe suivante :



A la main, procédez à la résolution du labyrinthe en détaillant l'état de la file.

Exercice 6 :

1. Écrire la fonction `affiche_liste` qui prend en argument une file, et retourne la chaine de caractère permettant un affichage de la file.
   Exemple :

```
>>> f = File()
>>> f.ajout(5)
>>> f.ajout(12)
>>> f.ajout(10)
>>> print(affichage(f))
10->12->5
```

2. Écrire la fonction `creation_file` qui prend en argument une liste et retourne la file dont les éléments sont stockés dans l'ordre de leur lecture dans la pile. Exemple :

```
>>> f = creation_file([8,1,6,4,5])
>>> print(affichage(f))
5->4->6->1->8
```

3. Écrire la fonction `taille` qui prend en argument une file, et renvoie le nombre d'élément qu'elle contient. (la file ne doit pas finir vide à l'issue d'appel de la fonction.)

4. Écrire la fonction `separation` qui prend en argument une file d'entier et qui sépare dans deux files les nombres paires et impaires.

Exercice 7 :

1. Écrire une fonction `retourne` qui prend en argument deux piles et "retourne" la deuxième pile sur la première. Si la deuxième pile est vide, la fonction ne fait aucune action.

2. Dans cette partie, nous allons implémenter une file à l'aide de deux piles. L'idée est la suivante : le sommet de la première pile (la pile tête) correspond à l'avant de la file, tandis que le sommet de la seconde pile (la pile queue) correspond à l'arrière de la file. Les éléments entrant dans la file sont empilés dans la pile queue. Pour défiler, on dépile la pile tête. Si la pile tête est vide, la seconde pile peut être "retournée" sur la première. Et on est alors en mesure de défiler. La file est vide lorsque les deux piles le sont.

   (a) Nous disposons de la file `f1` dons la tête est la pile `tete` $= [1, 2]$ et la pile `queue` $= [3, 4, 5]$.

   Établir les valeurs de deux piles composant la file `f1` à chacune des actions suivantes :

```
defile f1 ;;
enfile f1 2 ;;
defile f1 ;;
enfile f1 3 ;;
defile f1 ;;
defile f1 ;;
defile f1 ;;
```

(b) Les méthodes ne feront appel qu'aux méthodes définies pour les piles.
On définie la classe `File` :

Classe file :

```
from pile import *
class File :
    def __init__(self):
        self.tete = Pile()
        self.queue=Pile()
```

Écrire les méthodes `file_vide`, `enfile`, `defile` et `tete_file` qui renvoie l'élément en tête de file, et **sans** le faire disparaitre de la file. Si la file est vide, la fonction renverra un message d'erreur.