Exercice 1

Labyrinthe

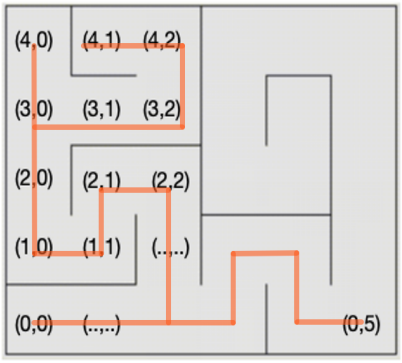


FIGURE 1 – sortir d'un labyrinthe

En adoptant la règle suivante :

- s'il y a une seule direction possible, avancer (couloir)
- s'il y a plusieurs directions possibles : choisir l'une des directions non parcourue
- impasse : revenir en arrière et choisir l'une des directions non parcourues.

> La tortue prend à gauche à la premiere bifurcation [1]. Arrive à une impasse [2]. Puis revient en arrière jusqu'à cette bifurcation [3]. Elle poursuit alors tout droit, ...etc...

1. Comment utiliser une structure de données en *PILE* pour sortir à coup sûr du labyrinthe ?
2. Représenter cette pile aux 3 moments : [1], [2], et [3].

On suppose que la structure de données *PILE* possède l'interface suivante :

fonction	specification	exemple
`Stack`	créé une pile vide	p = Stack()
`is_empty`	test si pile vide, retourne un bool	`is_empty(p)`
`Push`	empile une valeur	`Push(p, elem)`
`Pop`	depile le sommet : Retire et retourne cet element	`Pop(p)`

3. Donner les instructions qui ont permi de créer la pile `p`, et d'ajouter des coordonnées de cases jusqu'à parvenir au moment [1].
4. Donner l'état de la pile `p` au moment [2]
5. Donner les instructions qui ont permi de depiler à partir du moment [2], jusqu'à parvenir au moment [3].

Exercice 2

Connaissance du cours

1. Citer les opérations qui font partie de l'interface d'une *pile*.
2. Vrai ou Faux ? Pour ajouter un élément à une pile, il est aussi facile de l'ajouter au début qu'à la fin de la pile.
3. Vrai ou Faux ? L'opération *dépiler* (ou `pop`) s'execute en un temps qui est proportionnel au nombre de valeurs stockées dans la pile ?
4. Vrai ou Faux ? Pour accéder à un élément d'une pile, de nom `ma_pile` il suffit de connaitre son indice `i` et de faire : `ma_pile[i]`.
5. Vrai ou Faux ? Pour accéder au dernier élément d'une pile, l'opération s'effectue en un temps constant, indépendant de la taille de la pile.
6. Vrai ou Faux ? L'opération sur une liste `liste.pop(0)` permet de supprimer et retourner le premier élément d'une liste
7. Vrai ou Faux ? Cette opération se fait en un TEMPS CONSTANT.

Exercice 3

Pile d'instructions

Considérons l'exemple suivant :

```
def h(x):
  return x+1

def g(x):
  return h(x)+2

def f(x)
  return g(x)+1
```

1. Que se passe-t-il lors de l'appel `f(5)` ? Représenter la pile d'instructions correspondante.
2. Calculer le résultat pour `f(5)`

Exercice 4

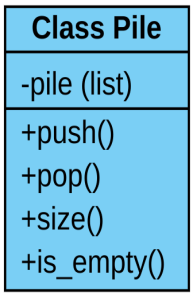
Implémentation type *objet* d'une Pile

On définit la classe `Pile`. Les fonctions sont cette fois des méthodes de classe :

```
class Pile:
  def __init__(self):
    self.pile = [] # on crée une liste vide lors de la création de l'
   objet.

  def push(self, element_a_empiler):

    '''
```

```
    Empile un élément sur la pile.
    :param element_a_empiler: ce qu'on veut
    '''
    # à completer

  def size (self):
    '''
    :returns: un entier correspondant au nombre d'éléments sur la pile.
    '''
    # à completer

  def pop(self):
    '''Renvoie l'élément sur le haut de la pile et l'enlève de la pile.
    :returns: dernier élément empilé
    :raises: renvoi une erreur si la pile est vide !
    '''
    # à completer

  def is_empty(self):
    '''
    :returns: un booléen True si la pile est vide
    '''
    #à completer

  def head(self):
    '''retourne l'element du sommet de la pile
    '''
```

On représente la classe de la manière suivante :



Lorsque l'on veut définir une nouvelle pile, on créé une nouvelle instance de cette classe avec l'instruction :

```
ma_pile = Pile() # on crée une pile
```

Et pour empiler et depiler, on utilise les méthodes associées à cette pile :

```
ma_pile.push(valeur) # empile valeur dans ma_pile
ma_pile.pop() # on depile
```

Comme dans l'exercice 3 en ligne (déverser une pile), écrire une fonction `deversePile` qui déverse une pile `p1` vers une pile `p2`, où les éléments seront mis dans `p2` en sens inverse.

Exercice 5

Expression correctement parenthesée

On dispose d'une expression mathématique :

$$g = [(1 + 2) * 3 + 10 * (3 - 1)$$

On cherche à savoir si cette expression est correctement parenthésée. On utilisera un tableau associatif `dicoS` comprenant les couples de symboles associés, ainsi qu'une pile `p`.

1. Ecrire le contenu du dictionnaire `dicoS`. Les clés seront les symboles ouvrants [, { et (et les valeurs, les caractères fermants], },)
2. Ecrire une boucle bornée qui parcourt les caractères de la chaine `g`
3. Puis le contenu de cette boucle : si le caractère est une parenthèse *ouvrante*, empiler dans `p`. Si le caractère est fermant, dépiler `p` à condition que ce caractère corresponde à celui qui est au sommet de la pile `p`.
4. A quelle condition sur `p` peut-on déduire que l'expression est correctement parenthésée ?
5. Ecrire une fonction qui prend en argument une chaine de caractères représentant l'expression mathématique, et qui renvoie le booléen True si celle-ci est correctement parenthésée.
6. Ajouter les instructions à cette fonction pour que celle-ci lève une exception dans le cas où l'expression n'est PAS correctement parenthésée.

Exercice 6

Implémentation d'une *File*

Une `File` est une structure de données comparable à la `Pile`, mais où la fonction dépile agit sur le **premier** élément de la `File`.

L'interface d'une `File` comprend alors les opérations permettant :
- de tester si une file est vide : *est_vide* (`is_empty`)
- d'ajouter d'un élément à la fin de la série : *enfiler* (`enqueue`)
- de retirer et renvoyer le premier élément de la file : *défiler* (`dequeue`)

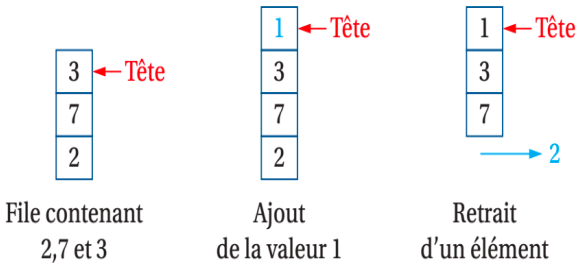


FIGURE 2 – illustration du fonctionnement d'une File

Supposons pour la suite de l'exercice que la *File* soit **implémentée** grâce à une **classe `Queue()`** dont les méthodes sont : `is_empty` `enqueue` (enfile) `dequeue` (defile)

1. Ecrire un programme test qui créé une *File* `ma_file`, et qui enfile les valeurs "Premier", "Deuxieme", "Troisieme", et qui *defile* une fois.
2. Ecrire une boucle non bornée sur `ma_file` qui parcourt la file et affiche ses éléments dans l'ordre, du premier au dernier. La file `ma_file` devra être restaurée, identique à ce qu'elle était avant le parcours par cette boucle. *Astuce : utiliser une autre File pour stocker les valeurs au fur et à mesure que l'on défile. Puis (re)copier les valeurs depuis cette File dans ma_file.*

Exercice 7

Correction des exercices en ligne

7.1 Exercice 1 : implementer la pile

```

# exercice 1
L = ['a',1,'b',2,'c',3,'d',4]

def Pile():
    return []

def est_vide(pile):
    return pile == []

def depile(pile):
    assert pile != [], 'impossible de depiler : pile vide'
    return pile.pop()

def empile(a,pile):
    pile.append(a)

def sommet(pile):
    assert pile != [], 'pile vide'
    return pile[-1]


p = Pile()
for a in L:
    if isinstance(a,int):
        empile(a,p)
print(p)

# affiche
[1, 2, 3, 4]
```

7.2 Exercice 2 : tests d'assertion

```
p2 = Pile()
depile(p2)
```

Le traceback affiche :

```
AssertionError: impossible de depiler : pile vide
```

```
sommet(p2)
```

Le traceback affiche :

```
AssertionError: la pile n_a pas de sommet : pile vide
```

7.3 Exercice 3 : deverser une pile

```
# exercice 3
def deversePile(p1,p2):
    while not(p1==[]):
        a = depile(p1)
        empile(a,p2)
    return p2
```

7.4 Exercice 4 : evaluation d'une expression en NPI

```
def evalNPI(L):
    dicoP = {'+' : add,
             '-' : soust,
             '*' : multip
    }
    p=[]
    for a in L:
        if a in dicoP:
            deuxieme = depile(p)
            premier = depile(p)
            r = dicoP[a](premier,deuxieme)
            empile(r,p)
        elif isinstance(a,int) :
            empile(a,p)
    return p[0]
```

Exercice 8

Correction de la fiche sd1 piles

8.1 Ex 2 : deverser une pile

Classe Pile (non demandé)

```
class Pile:
    def __init__(self):
        self.lst = []

    def is_empty(self):
        return self.lst == []

    def push(self,val):
        self.lst.append(val)

    def pop(self):
        return self.lst.pop()

    def head(self):
        return self.lst[-1]

    def __str__(self):
        s = ''
        for c in self.lst:
```

```
            s += str(c) +','
        return s
```

réponse aux questions de l'exercice :

```
def deversePile(p1,p2):
    while not p1.is_empty():
        a = p1.pop()
        p2.push(a)
# utilisation de la fonction deversePile:
p1,p2 = Pile(),Pile()
p1.push(1)
p1.push(2)
p1.push(3)
deversePile(p1,p2)
print(p2)
```

8.2 Exercice 3 : expression correctement parenthesée

```
dicoS = {"[": "]","(": ")"}



def correct_parenth(g):
    p = Pile()
    for c in g:
        if c in dicoS:
            p.push(c)
        if c == dicoS[p.head()]:
            p.pop()
    return p.is_empty()
```

On essaie :

```
>>> g = '[(1+2)*3+10*(3-1)'
>>> correct_parenth(g)
False
>>> g = '[(1+2)*3+10*(3-1)]'
>>> correct_parenth(g)
True
```

8.3 Ex 4 : Files

1. Programme test

```
ma_file = File()
ma_file.enqueue("Premier")
ma_file.enqueue("Deuxieme")
ma_file.enqueue("Troisieme")
ma_file.dequeue()
```

2. Affiche les éléments d'une file

```
f2 = Queue()
while not ma_file.is_empty():
    a = ma_file.dequeue()
    f2.enqueue(a)
    print(a)

while not ma_file.is_empty():
    a = f2.dequeue()
    ma_file.enqueue(a)
```