

1

Activité

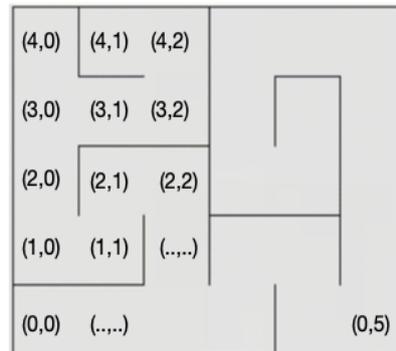


FIGURE 1 – sortir d'un labyrinthe

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 première bifurcation. Représenter la pile des positions parcourues jusqu'à la première impasse. Représenter la pile obtenue lorsque l'on dépile toutes les positions jusqu'au retour en arrière.

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'exécute 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 connaître 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, en un TEMPS CONSTANT.

2.1 Exercice 1 : pile d'instructions

Considérons l'exemple suivant :

```

1 def h(x):
2     return x+1
3
4 def g(x):
5     return h(x)+2
6
7 def f(x)
8     return g(x)+1

```

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

2.2 Exercice 2 : implémentation type *objet* d'une Pile

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

```

1 class Pile:
2     def __init__(self):
3         self.pile = [] # on crée une liste vide lors de la création de l'objet.
4
5     def push(self, element_a_empiler):
6
7         '''
8         Empile un élément sur la pile.
9         :param element_a_empiler: ce qu'on veut
10        '''
11        # à compléter
12
13    def size (self):
14        '''
15        :returns: un entier correspondant au nombre d'éléments sur la pile.
16        '''
17        # à compléter
18
19    def pop(self):
20        '''Renvoie l'élément sur le haut de la pile et l'enlève de la pile.
21        :returns: dernier élément empilé
22        :raises: renvoi une erreur si la pile est vide !
23        '''
24        # à compléter
25
26    def is_empty(self):
27        '''
28        :returns: un booléen True si la pile est vide
29        '''
30        #à compléter
31
32    def head(self):
33        '''retourne l'element du sommet de la pile
34        '''

```

On représente la classe de la manière suivante :

Class Pile
-pile (list)
+push()
+pop()
+size()
+is_empty()

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

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

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

```
1 ma_pile = ma_pile.push(valeur) # empile valeur dans ma_pile
2 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.

2.3 Exercice 3 : expression correctement parenthésée

On dispose d'une expression mathématique :

$$g = \text{"}[(1 + 2) * 3 + 10 * (3 - 1)\text{"}$$

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 chaîne `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 chaîne 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.

2.4 Exercice 4 : 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 : `defiler` (`dequeue`)

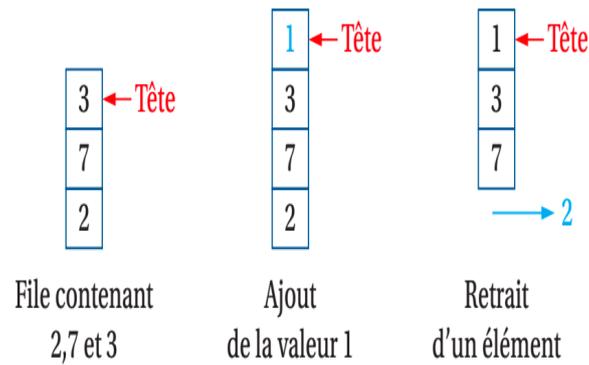


FIGURE 2 – illustration du fonctionnement d'une File

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

- `is_empty`
- `enqueue`
- `dequeue`

1. Ecrire un programme test qui crée 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 copier les valeurs depuis cette File dans `ma_file`.*

Correction des exercices en ligne

3.1 Exercice 1 : implementer la pile

```
1 # exercice 1
2 L = ['a',1,'b',2,'c',3,'d',4]
3
4 def Pile():
5     return []
6
7 def est_vide(pile):
8     return pile == []
9
10 def depile(pile):
11     assert pile != [], 'impossible de depiler : pile vide'
12     return pile.pop()
13
14 def empile(a,pile):
15     pile.append(a)
16
17 def sommet(pile):
18     assert pile != [], 'pile vide'
19     return pile[-1]
20
21
22 p = Pile()
23 for a in L:
24     if isinstance(a,int):
25         empile(a,p)
26 print(p)
27
28 # affiche
29 [1, 2, 3, 4]
```

3.2 Exercice 2 : tests d'assertion

```
1 p2 = Pile()
2 depile(p2)
```

Le traceback affiche :

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

```
1 sommet(p2)
```

Le traceback affiche :

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

3.3 Exercice 3 : deverser une pile

```
1 # exercice 3
2 def deversePile(p1,p2):
3     while not(p1==[]):
4         a = depile(p1)
```

```

5     empile(a, p2)
6     return p2

```

3.4 Exercice 4 : evaluation d'une expression en NPI

```

1 def evalNPI(L):
2     dicoP = {'+' : add,
3             '-' : soust,
4             '*' : multip
5             }
6     p=[]
7     for a in L:
8         if a in dicoP:
9             deuxieme = depile(p)
10            premier = depile(p)
11            r = dicoP[a](premier,deuxieme)
12            empile(r,p)
13        elif isinstance(a,int) :
14            empile(a,p)
15    return p[0]

```

4

Correction de la fiche sd1 piles

4.1 Ex 2 : deverser une pile

Classe Pile (non demandé)

```

1 class Pile:
2     def __init__(self):
3         self.lst = []
4
5     def is_empty(self):
6         return self.lst == []
7
8     def push(self, val):
9         self.lst.append(val)
10
11    def pop(self):
12        return self.lst.pop()
13
14    def head(self):
15        return self.lst[-1]
16
17    def __str__(self):
18        s = ''
19        for c in self.lst:
20            s += str(c) + ', '
21    return s

```

réponse aux questions de l'exercice :

```

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

```

```

6 p1,p2 = Pile(),Pile()
7 p1.push(1)
8 p1.push(2)
9 p1.push(3)
10 deversePile(p1,p2)
11 print(p2)

```

4.2 Exercice 3 : expression correctement parenthésée

```

1 dicoS = {"[": "]", "(" : ")"}
2
3
4
5 def correct_parenth(g):
6     p = Pile()
7     for c in g:
8         if c in dicoS:
9             p.push(c)
10            if c == dicoS[p.head()]:
11                p.pop()
12            return p.is_empty()

```

On essaie :

```

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

```

4.3 Ex 4 : Files

1. Programme test

```

1 ma_file = File()
2 ma_file.enqueue("Premier")
3 ma_file.enqueue("Deuxieme")
4 ma_file.enqueue("Troisieme")
5 ma_file.dequeue()

```

2. Affiche les éléments d'une file

```

1 f2 = Queue()
2 while not ma_file.is_empty():
3     a = ma_file.dequeue()
4     f2.enqueue(a)
5     print(a)
6
7 while not ma_file.is_empty():
8     a = f2.dequeue()
9     ma_file.enqueue(a)

```

Cours : structures de données linéaires

5.1 Def : structure de données

Une **structure de données** est une manière de stocker, d'accéder à, et de manipuler des données.

Structure de données **linéaire** : les données sont rangées sur un ligne ou une colonne.

Un **type abstrait** décrit essentiellement une interface, indépendamment du langage de programmation.

5.2 Pile

5.2.1 Def et interface

Une Pile est une structure de données linéaire. Le dernier élément entré sera aussi le premier à sortir (Last In First Out : LIFO). C'est une structure très utilisée pour le *retour en arrière*, *backtracking*, le parcours en profondeur d'un graphe...

L'interface d'une structure de données décrit de quelle manière on peut la manipuler. Les opérations possibles sont :

- test d'une pile vide (renvoie True si la pile est vide)
- ajout d'un élément (empiler = push), mis au sommet de la pile
- retire le premier élément de la pile, celui au sommet (dépiler = pop) si la pile est non vide et renvoie cet élément.
- lire le sommet de la pile.

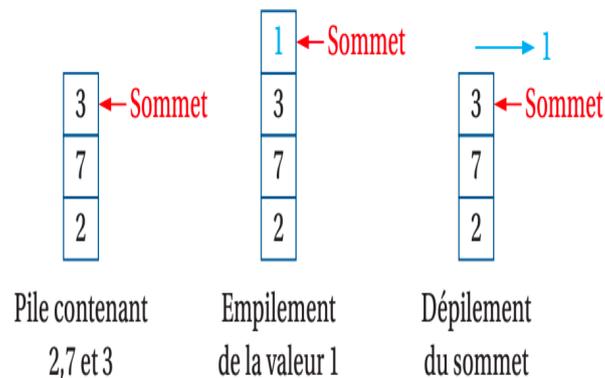


FIGURE 3 – illustration du fonctionnement d'une Pile

5.2.2 Implémentations en python

L'implémentation contient le code de ces méthodes (comment fait Python). Il n'est pas nécessaire de connaître l'implémentation pour manipuler la structure de données. Chaque opération doit se faire en temps constant, indépendant de la taille de la Pile : $O(1)$.

La liste en python est particulièrement adaptée à cette implémentation.

```

1 pile = [] # creation d'une pile
2 pile == [] # tester si la pile est vide
3 pile.append(valeur) # empile valeur dans la pile
4 pile.pop() # depiler l'element au sommet
5 pile[-1] # lire l'element au sommet de la pile

```

5.3 File

5.3.1 Def et interface

Une file (queue ou FIFO pour first in, first out) est une collection d'objets munie des opérations suivantes :

- savoir si la file est vide (`is_empty`);
- ajouter un élément dans la file (enfiler ou enqueue) en temps constant;
- retirer et renvoyer l'élément le plus ancien de la file (defiler ou dequeue).

C'est une structure de données utilisée en informatique pour par exemple gérer la file d'attente lors de la gestion des impressions.

5.3.2 Implémentations en python

L'implémentation d'une File est imparfaite avec une liste en python pour l'opération de *defiler*. On peut utiliser l'instruction `file.pop(0)`. Le problème est que celle-ci ne s'effectue pas en temps constant.

La bibliothèque standard possède toutefois un module nommé `collections` qui contient quelques structures de données supplémentaires, dont le type `deque`, qui est une implémentation de file :

```
1 from collection import deque
2 f = deque()
3 f.append("Premier")
4 f.append("Deuxieme")
5 print(f.popleft())
6 f.append("Troisieme")
7 while f:
8     print(f.popleft())
```

6

Listes

6.1 Principe

Une liste chaînée est une structure linéaire, constituée de cellules. Une cellule (ou *maillon*) contient :

- une valeur (de n'importe que type),
- et un pointeur qui dirige soit vers la cellule suivante, soit vers `None`, auquel cas la cellule est la *dernière* de la séquence.

Il s'agit d'une structure *linéaire*, mais dans laquelle les éléments n'occupent pas *à priori*, des positions contigües dans la mémoire :

(voir cours sd2 : Listes)