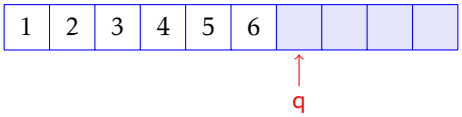
Chapitre 1

Corrigé des exercices

Exercice 1 Lorsqu'on définit une pile à l'aide d'un tableau statique, on maintient un pointeur vers le première case disponible du tableau, qui représente le sommet de la pile :




```
class Pile:
    """ définition d'une pile à l'aide d'un tableau """
    def __init__(self, n):
        self.lst = [None] * n
        self.size = n
        self.q = 0

    def empty(self):
        return self.q == 0

    def full(self):
        return self.q == self.size

    def push(self, x):
        if self.full():
            raise ValueError("pile pleine")
        self.lst[self.q] = x
        self.q += 1

    def pop(self):
        if self.empty():
            raise ValueError("pile vide")
        self.q -= 1
        return self.lst[self.q]
```

Exercice 2 La première pile (la pile a) reçoit les éléments qu'on ajoute à la file. Lorsqu'on veut supprimer un élément de la file, celui-ci est extrait de la pile b à moins que celle-ci ne soit vide, auquel cas les éléments de la pile a sont tout d'abord transférés dans la pile b.

```
class File:
    """ Définition d'une file à l'aide de deux piles """
    def __init__(self):
        self.a = Pile()
        self.b = Pile()

    def empty(self):
        return self.a.empty() and self.b.empty()

    def add(self, x):
        self.a.push(x)

    def take(self):
        if self.empty():
            raise ValueError("file vide")
```

```
        if self.b.empty():
            while not self.a.empty():
                self.b.push(self.a.pop())
        return self.b.pop()
```

Puisque la méthode add revient à appliquer la méthode push à la pile a, sont coût est un O(1). En revanche, le coût de la méthode take dépend de l'état de la pile b : si celle-ci n'est pas vide le coût est un O(1), si elle est vide le coût est un O(n) (coût du transfert de la pile a vers la pile b). Cependant on peut observer que le coût *amorti* de la méthode take est un O(1), car chaque élément de la file ne subit que trois opérations, toutes de coût constant : ajout dans la pile a, transfert vers la pile b, suppression de la pile b.

Exercice 3 On empile les éléments dans celle des deux files qui est non vide (ou n'importe laquelle si les deux sont vides) ; appelons-la a. Lorsqu'on veut dépiler un élément, on transfère les éléments de la file a vers la file b, à l'exception du dernier élément, qui est renvoyé.

```
class Pile:
    """ définition d'une pile à l'aide de deux files """
    def __init__(self):
        self.a = File()
        self.b = File()

    def empty(self):
        return self.a.empty() and self.b.empty()

    def push(self, x):
        if self.b.empty():
            self.a.add(x)
        else:
            self.b.add(x)

    def pop(self):
        if self.empty():
            raise ValueError("liste vide")
        if self.b.empty():
            x = self.a.take()
            while not self.a.empty():
                self.b.add(x)
                x = self.a.take()
        else:
            x = self.b.take()
            while not self.b.empty():
                self.a.add(x)
                x = self.b.take()
        return x
```

Le coût de la méthode push est à l'évidence un O(1) : on se contente d'insérer l'élément dans une des deux files. En revanche, le coût de la méthode pop est systématiquement un $\Theta(n)$, correspondant au transfert d'une file vers l'autre.

Exercice 4 Commençons par définir le notion de cellule :

```
class Cell:
    """ définition d'une cellule avec deux pointeurs """
    def __init__(self, x):
        self.val = x
        self.next = None
        self.pred = None
```

Les opérations d'ajout et de suppression dans une file doivent tenir compte du cas particulier de la file vide (pour l'ajout) et de la file à un seul élément (pour la suppression).

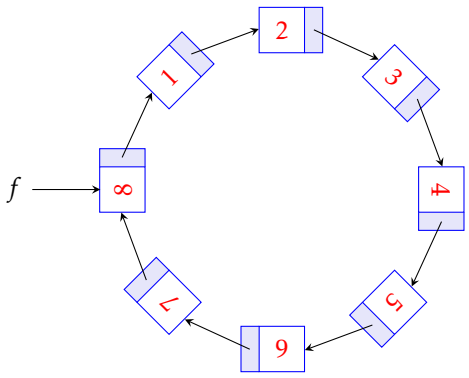
```python
class File:
    """ définition d'une file par liste doublement chaînée """
    def __init__(self):
        self.queue = None
        self.tete = None

    def empty(self):
        return self.tete is None

    def add(self, x):
        c = Cell(x)
        if self.empty():
            self.tete = c
        else:
            c.next = self.queue
            self.queue.pred = c
        self.queue = c

    def take(self):
        if self.empty():
            raise ValueError("file vide")
        c = self.tete
        if c.pred is None:
            self.tete = None
            self.queue = None
        else:
            self.tete = c.pred
        return c.val
```

Pour ajouter un élément, on crée une nouvelle cellule puis on modifie les pointeurs nécessaires pour préserver la structure de liste doublement chaînée.
Pour supprimer un élément, on récupère la cellule en tête de liste puis là encore on modifie les pointeurs nécessaires pour préserver la structure de liste doublement chaînée.

Exercice 5 Nous allons avoir besoin de la classe *Cell* déjà définie en cours :

```python
class Cell:
    """ définition d'une cellule avec pointeur """
    def __init__(self, x):
        self.val = x
        self.next = None
```

Une file est représentée par une liste circulaire dans laquelle chaque élément de la file pointe vers l'élément qui le suit dans la file ; quant à celle-ci, elle pointe vers son dernier élément. Par exemple, la file $f = (1, 2, 3, 4, 5, 6, 7, 8)$ sera représentée en mémoire par le schema suivant :



Pour insérer un élément dans la file, il faut distinguer le cas où celle-ci est vide (il faut alors créer une cellule qui pointe vers elle-même) ou non vide (il faut insérer une nouvelle cellule vers laquelle pointera f et modifier certains pointeurs).

Pour supprimer un élément de la file, il faut distinguer le cas où la file ne contient qu'un élément (elle deviendra vide) et le cas où la file en contient plus d'un, auquel cas la cellule qui suit celle pointée par f sera supprimée et un pointeur modifié.

```
class File:
    """ définition d'une file par liste circulaire """
    def __init__(self):
        self.lst = None

    def empty(self):
        return self.lst is None

    def add(self, x):
        c = Cell(x)
        if self.empty():
            c.next = c
        else:
            c.next = self.lst.next
            self.lst.next = c
        self.lst = c

    def take(self):
        if self.empty():
            raise ValueError("file vide")
        if self.lst.next is self.lst:
            c = self.lst
            self.lst = None
        else:
            c = self.lst.next
            self.lst.next = c.next
        return c.val
```

Exercice 6 La fonction qui suit ne détecte pas d'éventuelles erreurs de syntaxe :

```
def evalue(lst):
    p = Pile()
    for t in lst:
        if t in op_bin:
            y = p.pop()
            x = p.pop()
            p.push(op_bin[t](x, y))
        elif t in op_uni:
            x = p.pop()
            p.push(op_uni[t](x))
        else:
            p.push(t)
    return p.pop()
```

Nous allons maintenant détecter deux types d'erreurs de syntaxe : le cas où on cherche à dépiler une pile vide (s'il manque d'opérande pour un opérateur unaire ou binaire) et le cas où à la fin de l'évaluation la pile contient plus d'un élément. Ne seront pas détectées les erreurs liées à un opérateur de nom inconnu ou les erreurs mathématiques (division par 0 par exemple).

```python
def evalue2(lst):
    p = Pile()
    for t in lst:
        if t in op_bin:
            if p.empty():
                raise ValueError("expression incorrecte")
            y = p.pop()
            if p.empty():
                raise ValueError("expression incorrecte")
            x = p.pop()
            p.push(op_bin[t](x, y))
        elif t in op_uni:
            if p.empty():
                raise ValueError("expression incorrecte")
            x = p.pop()
            p.push(op_uni[t](x))
        else:
            p.push(t)
    s = p.pop()
    if not p.empty():
        raise ValueError("expression incorrecte")
    return s
```

Quelques exemples d'utilisation de ces fonctions :

```
>>> evalue([1, 2, 3, 'sqrt', '*', '+', 4, '/'])
1.1160254037844386

>>> evalue2([1, 2, 'sqrt', '*', '+', 4, '/'])
ValueError: expression incorrecte

>>> evalue2([1, 2, 3, 'sqrt', '*', '+', 4, 5, '/'])
ValueError: expression incorrecte
```