# Piles et files 10



## 10.1 Pile

### 10.1.1 Présentation

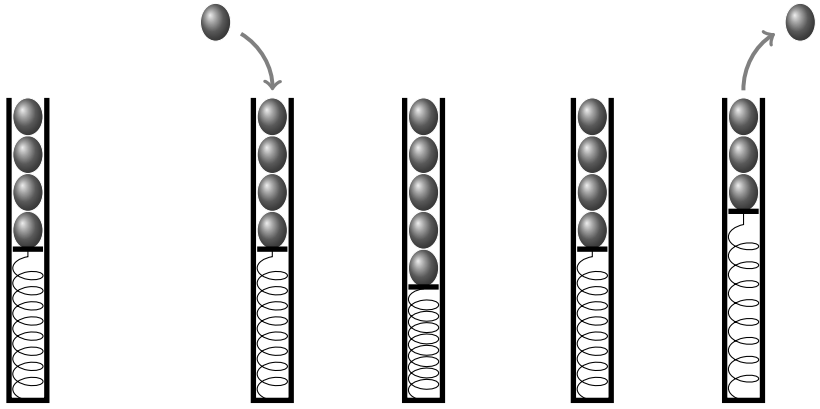
**Définition – Pile**

Une pile est une structure de données dans laquelle le dernier élément stocké est le premier à en sortir. On parle de principe *LIFO* pour *Last In First Out*. Le dernier élément stocké est appelé **sommet**.

Pour gérer une pile, indépendamment de la façon dont elle est implémentée, on suppose exister les opérations élémentaires suivantes :

- `cree_pile()` qui crée une pile vide ;
- `empile(p,x)` qui empile l'élément `x` au sommet de la pile `p` ;
- `depile(p)` qui supprime le sommet de la pile `p` et renvoie sa valeur ;
- `est_vide(p)` qui teste si la pile `p`est vide.

On peut illustrer la structure de pile par l'image suivante.



pile (*stack*) empile (*push*) dépile (*pop*)

Théoriquement, chacune de ces opérations doit se faire à **temps constant** (complexité notée $\mathcal{O}(1)$). Une des possibilités pour implémenter les piles est d'utiliser le module `deque`. Chacun des éléments de la pile peut être un objet de type différent.

```
from collections import deque
# Création d'une pile vide
pile = deque()
# Test si une pile est vide
len(pile) == 0
# Ajout de l'élément Truc au sommet de la pile
pile.append("Truc")
# Suppression (et renvoi) du sommet d'une pile non vide
sommet = pile.pop()
```

### 10.1.2 Applications directes

**Exemple –**

1. *En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la fonction* `copy_pile(p: pile) -> pile`*, permettant de faire une copie de la pile.*
2. Donner la complexité de cette fonction.

```
from collections import deque

pile = deque()
for i in range(10):
    pile.append(i)


def copy_pile(pile):
    pile_tmp = deque()
    pile_copy = deque()

    while pile :
        pile_tmp.append(pile.pop())

    while pile_tmp :
        el= pile_tmp.pop()
        pile.append(el)
        pile_copy.append(el)
    return pile_copy
```

**Exemple –**

1. *En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la fonction* `hauteur(p: pile) -> int` *renvoyant la hauteur de la pile. Attention, la pile initiale ne doit pas être perdue.*
2. *En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la fonction récursive* `hauteur_rec(p: pile) -> int` *renvoyant la hauteur de la pile. La pile initiale peut être perdue, l'utilisateur de la fonction pourra avoir fait une copie préalable.*
3. *Donner la complexité de cette fonction.*

```
def hauteur(pile) -> int :
    pile_temp = deque()
    h = 0
    while pile :
        pile_temp.append(pile.pop())
        h = h+1
    while pile_temp :
        pile.append(pile_temp.pop())
    return h

def hauteur_rec(pile):
    if not pile :
        return 0
    else :
        pile.pop()
        return 1+hauteur_rec(pile)
```

**Exemple –**

*En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la procédure* `reverse(p: pile) -> None`*, procédure pour laquelle les éléments de la pile sont inversés. Donner la complexité de cette fonction.*

```
def reverse(pile):
    pile_tmp = deque()
    pile_tmp2 = deque()

    while pile :
        pile_tmp.append(pile.pop())
    while pile_tmp :
        pile_tmp2.append(pile_tmp.pop())
    while pile_tmp2 :
        pile.append(pile_tmp2.pop())
```

## 10.2 File

### 10.2.1 Présentation

**Définition – File**

Une file est une structure de données dans laquelle le premier élément stocké est le premier à en sortir. On parle de principe *FIFO* pour *First In First Out*.

Pour gérer une file, indépendamment de la façon dont elle est implémentée, on suppose exister les opérations élémentaires suivantes :

- création d'une file vide ;
- test si une file est vide ;
- rajout d'un élément dans la file ;
- suppression (et renvoi) du premier élément inséré dans la file.

Théoriquement, chacune de ces opérations doit se faire à **temps constant**.

Une des possibilités pour implémenter les files est d'utiliser le module `deque`. Chacun des éléments de la file peut être un objet de type différent. Dans cette vision des files, les éléments sont ajoutés « à droite » et sortent de la file « par la gauche ».

```
from collections import deque

# Création d'une file vide
file = deque()

# Teste si une pile est vide
len(file) == 0

# Ajoute l'élément Truc dans la file
file.append("Truc")

# Suppression (et renvoi) du premier élément inséré dans la file
sommet = pile.popleft()
```

### 10.2.2 Applications directes

**Exemple –**

1. *En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la fonction* `copy_file(f: file) -> file`*, permettant de faire une copie de la file.*
2. Donner la complexité de cette fonction.

```
def copy_file(file):
    file_tmp = deque()
    file_copy = deque()
    while file :
        file_tmp.append(file.popleft())

    while file_tmp :
        el= file_tmp.popleft()
        file.append(el)
        file_copy.append(el)
    return file_copy
```

**Exemple –**

1. *En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la fonction* `longueur(f: file) -> int` *renvoyant la longueur de la file. Attention, la file initiale ne doit pas être perdue.*
2. *Donner la complexité de cette fonction.*

```
def longueur(file) -> int :
    file_tmp = deque()
    l = 0
    while file :
        file_tmp.append(file.popleft())
        l = l+1
    while file_tmp :
        file.append(file_tmp.popleft())
    return l
```

**Exemple –**

*En utilisant le module* `deque` *et uniquement les 4 opérations précédemment définies, donner l'implémentation de la procédure* `reverse(f: file) -> None`*, procédure pour laquelle les éléments de la file sont inversés. Donner la complexité de cette fonction.*