

7.1 Pile

7.1 Pile	1
7.2 File	3

7.1.1 Présentation

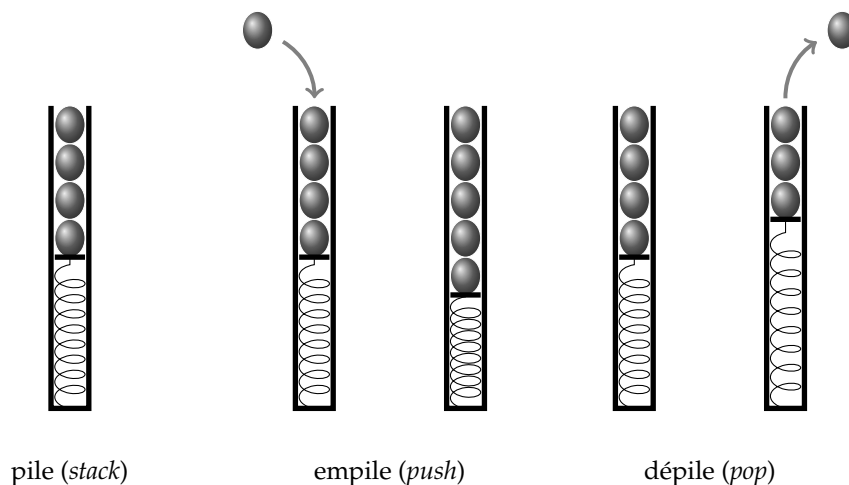
Définition – Pile

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.



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.

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

7.1.2 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.

```

1 from collections import deque
2
3 pile = deque()
4 for i in range(10):
5     pile.append(i)
6
7
8 def copy_pile(pile):
9     pile_tmp = deque()
10    pile_copy = deque()
11
12    while pile :
13        pile_tmp.append(pile.pop())
14
15    while pile_tmp :
16        el= pile_tmp.pop()
17        pile.append(el)
18        pile_copy.append(el)
19    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.

```

1 def hauteur(pile) -> int :
2     pile_temp = deque()
3     h = 0
4     while pile :
5         pile_temp.append(pile.pop())
6         h = h+1
7     while pile_temp :
8         pile.append(pile_temp.pop())
9     return h
10
11 def hauteur_rec(pile):
12     if not pile :
13         return 0
14     else :
15         pile.pop()
16         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.

```

1 def reverse(pile):
2     pile_tmp = deque()
3     pile_tmp2 = deque()
4
5     while pile :
6         pile_tmp.append(pile.pop())
7     while pile_tmp :
8         pile_tmp2.append(pile_tmp.pop())
9     while pile_tmp2 :
10        pile.append(pile_tmp2.pop())

```

7.2 File

7.2.1 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 ».

```

1 from collections import deque
2
3 # Création d'une file vide
4 file = deque()
5
6 # Teste si une pile est vide
7 len(file) == 0
8
9 # Ajoute l'élément Truc dans la file
10 file.append("Truc")
11
12 # Suppression (et renvoi) du premier élément inséré dans la file
13 sommet = file.popleft()

```

7.2.2 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.

```

1 def copy_file(file):
2     file_tmp = deque()
3     file_copy = deque()
4     while file :
5         file_tmp.append(file.popleft())
6
7     while file_tmp :
8         el= file_tmp.popleft()
9         file.append(el)
10        file_copy.append(el)
11    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.

```

1 def longueur(file) -> int :
2     file_tmp = deque()
3     l = 0
4     while file :
5         file_tmp.append(file.popleft())
6         l = l+1
7     while file_tmp :
8         file.append(file_tmp.popleft())
9     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.