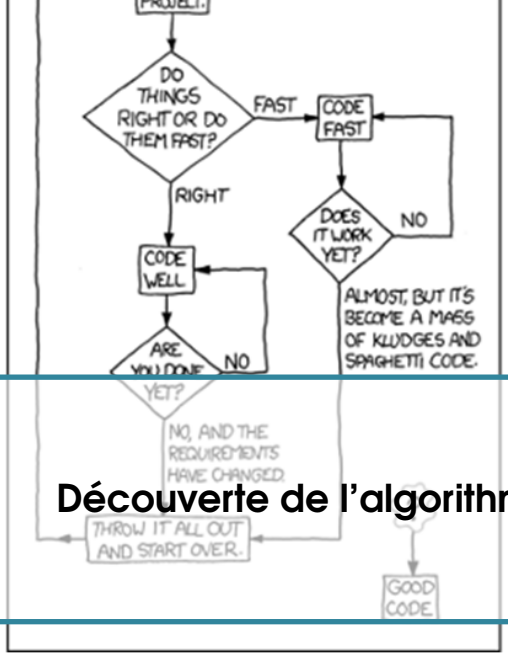
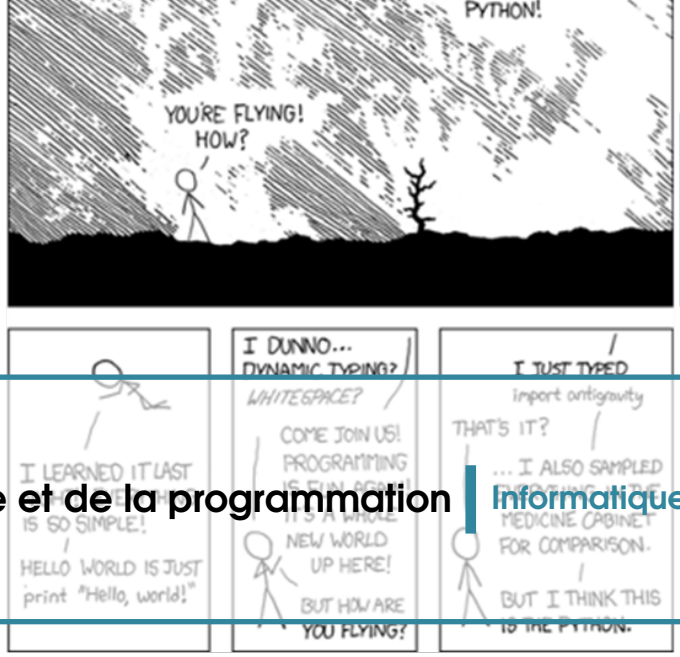




Cours

Chapitre 9

Piles et files

Savoirs et compétences :

- *Piles et files.*

1 Pile

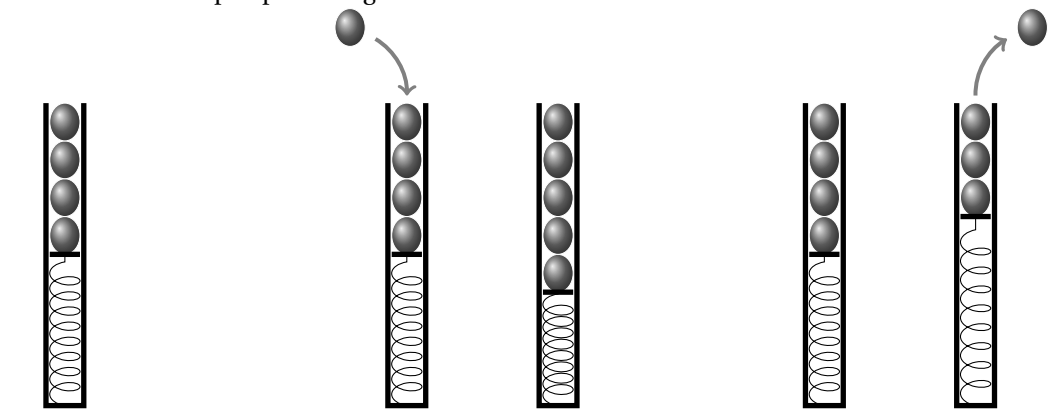
1.1 Présentation

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 pest 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 possiblités 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()
```

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())
```

▪

2 File

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 innséré dans la file.

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

Une des possiblités 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()
```

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.* ▪

Exercices d'application sur les piles

Dans les exercices qui suivent, on utilisera des **piles**. Les piles sont des structures de données basées sur le principe LIFO (Last In First Out : le dernier rentré dans la pile sera le premier à en sortir).

Les opérations élémentaires qu'on peut réaliser sur les piles sont les suivantes :

- savoir si une pile est vide ;
- empiler un nouvel élément sur la pile ;
- récupérer l'élément au sommet de la pile tout en le supprimant. On dit que l'on dépile ;
- accéder à l'élément situé au sommet de la pile sans le supprimer de la pile ;
- on peut connaitre le nombre d'éléments présents dans la pile.

Pour implémenter les piles on utilisera le module `deque`. Chacun des éléments de la pile peut être un objet de type différent.

```
from collections import deque

# Créer une pile vide
pile = deque()

# Tester si une pile est vide
len(pile) == 0

# Ajouer l'élément Truc au sommet de la pile
pile.append("Truc")

# Supprimer (et renvoyer) le sommet d'une pile non vide
sommet = pile.pop()
```

Seules ces fonctions éléméntaires seront utilisées dans les exercices suivants.

Exercice 1 – La parenthèse inattendue

Dans cet exercice, on souhaite savoir si une chaîne de caractères est bien parenthésée ou non. Une chaîne bien parenthésée est une chaîne vide ou la concaténation de chaînes bien parenthésées.

▪ **Exemple** Chaînes bien parenthésées :
- `"()"`,`"()()"`,`"(())"` et `"(()())"`.

Chaînes mal parenthésées :
- `")("`,`"(("`,`"()"` et `"())"`.

▪

Question 1 *Implémenter la fonction* `parentheses(s:str)-> list` *renvoie une liste de tuples correpondants aux indices des parenthèses ouvrantes et fermantes.*

Question 2 *Réaliser un programme permettant de savoir si une chaîne de caractères est bien parenthésée. La structure de pile est-elle nécessaire ?*

Question 3 *Adapter le premier programme pour qu'il puisse traiter des chaînes constituées de parenthèses, de crochets, ou d'accolades. Un mot est alors bien parenthésé si la parenthèse fermante qui correspond à chaque parenthèse ouvrante est du même type.*

Question 4 *Adapter le programme pour qu'il puisse traiter des mots constitués de parenthèses et d'autres caractères, qui n'interfèrent pas avec les parenthèses.*

Question 5 *Écrire une version récursive de la fonction* `parentheses`.

Exercice 2 – Inversion

Question 6 *Écrire une fonction qui intervertit les deux éléments situés au sommet d'une pile de taille au moins égale à 2.*

Exercice 3 – Dépile le n^e

Question 7 *Écrire une fonction qui dépile et renvoie le n^e élément d'une pile de taille au moins égale à n. Le premier élément correspond au sommet de la pile. Les $n-1$ premiers éléments devront conserver leur place.*

Exercice 4 – Lire le n^e

Question 8 *Écrire une fonction qui permet de lire (sans l'extraire) le n-ième élément d'une pile. On prévoira le cas où la pile n'est pas de taille suffisante pour qu'un tel élément existe.*

Exercice 5 – Inversion des extrêmes

Question 9 *Écrire une fonction qui prend une pile non vide en argument et place l'élément situé à son sommet tout au fond de la pile, en conservant l'ordre des autres éléments. Quelle est sa complexité en temps et en espace ?*

Exercice 6 – Inversion de la pile

Question 10 *Écrire une fonction similaire à* `reversed`*, qui prend une pile en argument et renvoie une autre pile constituée des mêmes éléments placés dans l'ordre inverse.*

Question 11 *Si on s'autorise à détruire la pile initiale, quelle est la complexité en temps et en espace de cette fonction ? Et si on ne s'y autorise pas ?*

Exercice 7 – Tu coupes ?

Question 12 *Écrire une fonction couper qui prend une pile et la coupe en enlevant de son sommet un certain nombre d'éléments (tirés au hasard) qui sont renvoyés dans une seconde pile.*

▪ **Exemple** Si la pile initiale est `[1, 2, 3, 4, 5]`, et que le nombre d'éléments retirés vaut 2, alors la pile ne contient plus que `[1, 2, 3]` et la pile renvoyée contient `[5,4]`. ▪