# TP 4 : Files

À chaque exercice, faire des tests !

## 1 Files

Cette section demande d'implémenter une file à l'aide d'un tableau (classe `ArrayQueue`), puis d'une liste chaînée circulaire (classe `LinkedListQueue`).

Une file doit proposer les méthodes suivantes :

**`head(self)`** Renvoie la tête de la file `self`. Lance une exception ValueError si la file est vide.

**`is_empty(self)`** Teste si la file `self` est vide.

**`add(self, x)`** Ajoute l'élément `x` en fin de la file `self`.

**`remove(self)`** Supprime la tête de la file `self`. Lance une exception ValueError si la file est vide.

Pour rappel, la syntaxe pour déclarer une classe en Python est la suivante :

```
class ArrayQueue:
    def __init__(self, size_max):
        # initialisation
    def head(self):
        # code
    # etc...

# Utilisation
if __name__ == "__main__":
    file = ArrayQueue(4)
    file.add(1)
    # etc...
```

Dans le cas d'une implémentation avec un tableau (ArrayQueue), on pourra passer une taille maximale `size_max` à la fonction d'initialisation. Dans le cas d'implémentation avec une liste chaînée il n'y a pas besoin.

Pour rappel, l'implémentation avec un tableau demande d'utiliser des attributs pour connaître les positions du début et fin de la file dans le tableau. L'implémentation avec une liste chaînée demande de connaître le point d'entrée de la file (qui change lorsqu'on ajoute un élément à la file).

L'implémentation avec une liste chaînée demande de créer une classe `Node` pour représenter les nœuds de la liste. Chaque nœud doit avoir un attribut pour la valeur, et un attribut pointant vers le nœud suivant (qui peut être `None` s'il représente le dernier élément de la liste).

## 2 Liste doublement chaînée

Écrire une structure de pile mutable à l'aide d'une liste doublement chaînée (classe `DoubleLinkedList`) qui fournit les méthodes supplémentaires suivantes :

**`concat(self, other)`** Concatène une autre liste doublement chaînée à `self` en temps constant. Cette fonction pourra être « destructive » et modifier `other`, qui ne doit plus être utilisé après utilisation dans `concat`.

**`size(self)`** Renvoie la taille de la liste.

**`member(self, v)`** Teste si la valeur `v` appartient à la liste `self`.

On utilisera une classe `NodeBidir` pour les nœuds qui fournit à chaque nœud un prédécesseur en plus d'un successeur.

Pour implémenter `concat` en temps constant, la liste pourra disposer d'un attribut pointant vers le dernier nœud de la liste.

## 3 Bonus : file non mutable

Implémenter une file non mutable à l'aide d'une liste chaînée circulaire (classe `LinkedListImmutQueue`). Les méthodes sont mêmes que pour la `LinkedListQueue`, mais `add` et `remove` renvoient une nouvelle file, plutôt que de modifier l'originale. On pourra réutiliser la classe `LinkedListQueue` à laquelle on a ajouté une méthode `deepcopy` qui renvoie une copie profonde de la file.

Tester que la file est bien immutable, donc par exemple :

```
f = LinkedListImmutQueue()
f = f.add(1)
print(f.is_empty()) # False
```

```
g = f.remove()
print(g.is_empty()) # True
print(f.is_empty()) # toujours False
g = g.add(2)
g = g.add(3)
print(f.head()) # 1
print(g.head()) # 2
```