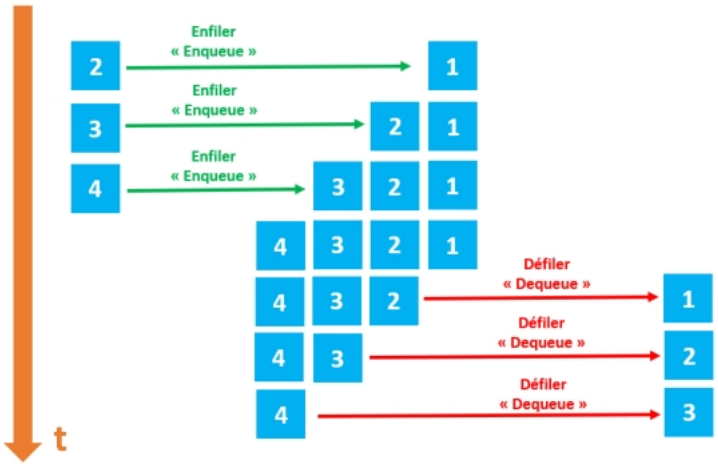
Les files (« queues » en anglais)

Définition d'une file

Les files sont fondées sur le principe du « premier arrivé, premier sorti » : elles sont dites de type **FIFO** (First In, First Out). C'est le principe de la file d'attente devant un guichet.

Voici un schéma :



L'insertion d'un élément dans une file s'appelle une opération de mise en file « Enfiler » et la suppression d'un élément s'appelle une opération de retrait de la file « Défiler ». Dans la file, nous maintenons toujours deux pointeurs, l'un pointant sur l'élément qui a été inséré en premier et qui est toujours présent dans la liste avec le pointeur en avant et l'autre pointant sur l'élément inséré en dernier avec le pointeur arrière.

En général, on utilise une file pour mémoriser temporairement des transactions qui doivent attendre pour être traitées. Voici quelques exemples d'applications :

- les serveurs d'impression, qui doivent traiter des requêtes dans l'ordre dans lequel elles arrivent, et les insère dans une file d'attente (ou une queue) ;
- les requêtes entre machines sur un réseau ;
- certains moteurs multi-tâches, dans un système d'exploitation, qui doivent accorder du temps machine à chaque tâche, sans en privilégier une plus qu'une autre ;
- un algorithme de parcours en largeur utilise une file pour mémoriser les nœuds visités ;
- on utilise des files pour créer toutes sortes de mémoires tampons (buffers en anglais) ;
- ...

Les méthodes associées à une file

Voici les méthodes que l'on doit associer sur un objet de type File :

Action sur la file :	**Méthode de la classe File** :
Créer une file vide appelée maFile	maFile = creer_file_vide()
La file est-elle vide ?	est_vide(maFile)
Enfiler un nouvel élément sur la pile	enfiler(maFile,élément)
Défiler un élément de la pile	defiler(maFile)
Lire la valeur en tête de file	lire_tete(maFile)
Lire la valeur en queue de la file	lire_queue(maPile)

Piles vs files : synthèse

Pile :	**File** :
Les objets sont insérés et supprimés à 1 seule extrémité	Les objets sont insérés et retirés aux 2 extrémités.
Un seul pointeur est utilisé : vers le sommet de la pile.	Deux pointeurs différents : le tête et la fin.
Le dernier objet inséré est le premier à sortir.	L'objet inséré en premier est le premier qui sera supprimé.
Ordre Last In First Out (LIFO)	Ordre First In First Out (FIFO)
Opérations : « Empiler » et « Dépiler ».	Opérations ; « Enfiler » et « Défiler ».
Visualisées sous forme de collections verticales.	Visualisées sous forme de collections horizontales.

TD : Implémenter une file avec une liste Python et des fonctions

Il est possible d'utiliser les tableaux dynamiques Python, class *list*, pour implémenter simplement à la volée des files.

Recopier et compléter le code ci-dessous pour implanter les fonctions élémentaires sur les files.

```
# on utilise les listes Python pour implanter les files
# la tête de la file sera l'élément d'indice 0, le premier enfilé,
# et la tête de file sera l'élément d'indice le plus élevé, celui ajouté en dernier.
# pour enfiler on pourra utiliser la méthode append()
# pour defiler on pourra utiliser la méthode pop(0)

def creer_file_vide():
        return

def est_vide(file):
        return

def enfiler(file, element):


def defiler(file):
        return

def lire_tete(file):
        return

def lire_queue(file):
        return

maFile = creer_file_vide()
print(est_vide(maFile))
enfiler(maFile, 5)
enfiler(maFile, 8)
enfiler(maFile, 10)
print(est_vide(maFile))
x = defiler(maFile)
print(x)
print(lire_tete(maFile))
print(lire_queue(maFile))
```

TP : implémenter une file avec une liste chaînée

On veut écrire une classe pour gérer une file à l'aide d'une liste chaînée. On dispose d'une classe `Maillon` permettant la création d'un maillon de la chaîne, celui-ci étant constitué d'une valeur et d'une référence au maillon suivant de la chaîne :

```
class Maillon :
        def __init__(self,v) :
                self.valeur = v
                self.suivant = None
```

Compléter la classe `File` suivante où l'attribut `dernier_file` contient le maillon correspondant à l'élément arrivé en dernier dans la file (on l'appelle aussi queue :

```
class File :
        def __init__(self) :
                self.dernier_file = None

        def enfile(self,element) :
                nouveau_maillon = Maillon(... , self.dernier_file)
                self.dernier_file = ...

        def est_vide(self) :
                return self.dernier_file == None

        def affiche(self) :
                maillon = self.dernier_file
                while maillon != ... :
                        print(maillon.valeur)
                        maillon = ...

        def defile(self) :
                if not self.est_vide() :
                        if self.dernier_file.suivant == None :
                                resultat = self.dernier_file.valeur
                                self.dernier_file = None
                                return resultat
                        maillon = ...
                        while maillon.suivant.suivant != None :
                                maillon = maillon.suivant
                        resultat = ...
                        maillon.suivant = None
                        return resultat
                return None
```

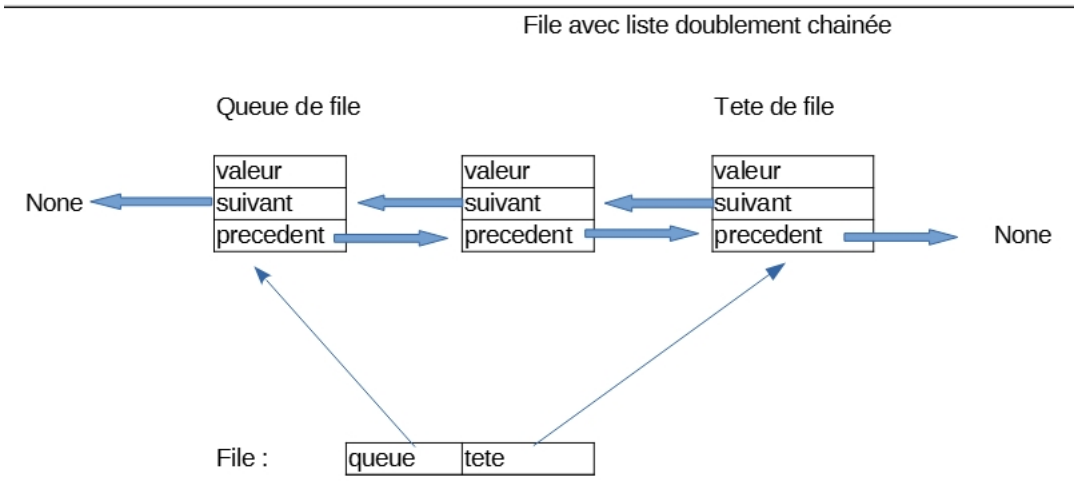
On pourra tester le fonctionnement de la classe en utilisant les commandes suivantes dans la console Python :

```
>>> F = File()
>>> F.est_vide()
 True
>>> F.enfile(2)
>>> F.affiche()
2
>>> F.est_vide()
 False
```

```
>>> F.enfile(5)
>>> F.enfile(7)
>>> F.affiche()
7
5
2
>>> F.defile()
2
>>> F.defile()
5
>>> F.affiche()
7
```

Implémenter une file avec une liste doublement chaînée

La structure de liste doublement chaînée est bien adaptée pour une implémentation de file. Chaque « boîte » ou « cellule » contient une donnée (valeur) , puis deux pointeurs, l'un appelé **suivant** qui indique la cellule du suivant dans la file, et l'autre appelé **precedent** qui indique la cellule qui précède. La cellule en tête de file a pour précédent **None**, et celle en queue de file aussi. Une file est donc un objet qui doit avoir deux attributs : un pointeur vers la **tête**, et un pointeur vers la **queue**.



Recopier le code de départ pour cette implantation.

```
# implémentation d'une File
# en utilisant une liste doublement chainée

class Cellule :

    def __init__(self, valeur, precedent = None, suivant =None):
        self.valeur = valeur
        self.precedent = precedent
        self.suivant = suivant

class File :

    def __init__(self):
        self.longueur = 0
        self.tete = None
        self.queue = None

    def __str__(self):
        ch = "\n Etat de la file : "
        ch += "queue ->"
        pointeur = self.queue
        while pointeur != None :
            ch += str(pointeur.valeur) + " -> "
            pointeur = pointeur.precedent
        ch += " tete"
        return ch

maFile = File()
print(maFile)
```

1. Analyser le constructeur de la classe `Cellule`. Quels sont ses attributs ?
2. Analyser le constructeur de la classe `File`. Quels sont ses attributs ?
3. Analyser la méthode `str` de la classe `File`. A quoi sert-elle ?
4. Exécuter le code et vérifier que la file `maFile` qui est affichée avec la commande `print` est bien vide.
5. Implanter pour la classe `File` la méthode `est_vide(self)`, et vérifier que la commande ci-dessous renvoie bien `True` car pour l'instant `maFile` est vide...

```
maFile.est_vide()
```

6. Implanter pour la classe `File` la méthode `taille(self)` qui renvoie la longueur de la file. Vérifier que le code ci-dessous fonctionne...

```
maFile.taille()
```

7. Passons aux choses sérieuses : implanter pour la classe `File` la méthode `enfiler(self, valeur)` qui ajoutera la valeur passée en argument en queue de file. Attention : il faut que les pointeurs vers les cellules suivantes et précédentes soient bien établis ! Vérifier après implantation de la code suivant fonctionne.

```
maFile.enfiler(5)
maFile.enfiler(8)
print(maFile)
maFile.est_vide()
```

8. Toujours des choses sérieuses : implanter pour la classe `File` la méthode `defiler(self)` qui renverra et supprimera la valeur en tête de file. Attention : il faut que les pointeurs vers les cellules suivantes et précédentes soient bien établis ! Vérifier après implantation de la code suivant fonctionne.

```
maFile.defiler()
maFile.defiler()
print(maFile)
maFile.est_vide()
```

9. Et pour finir : implanter pour la classe `File` la méthode `allerRetour(self)` qui affiche les valeurs de la file de la tête vers la queue puis de la queue vers la tête. Tester votre méthode.

```
maFile = File()
maFile.enfiler(1)
maFile.enfiler(2)
maFile.enfiler(3)
maFile.enfiler(4)
maFile.allerRetour()
```

Vous devez obtenir :

```
1
2
3
4
3
2
1
```