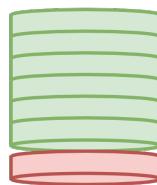


Chap. 6 – Piles et files

lout comme les tableaux, **Pile et File** sont des structures de données qui permettent de (1) stocker des ensembles d'objets et (2) ajouter/retirer des objets un à un.

6.1 – Introduction

Dans une pile (en anglais stack), chaque opération dépile le dernier élément arrivé et ajoute un sommet.



Dernier entré, premier sorti (en anglais **LIFO** pour last in, first out)

Dans une file (en anglais *queue*), chaque opération peut être ajoutée à la fin de la file. Pour imaginer ce qu'il se passe, imaginez que l'on a une liste d'attente dans laquelle (1) les personnes arrivent et (3) sortent services dans leur ordre d'arrivée.



Premier arrivé, premier sorti (en anglais **FIFO** pour *first in, first out*)

6.2 – Interface commune aux piles et aux files

Classiquement, chacune de ces deux structures a une **interface** proposant au minimum les quatre opérations suivantes :

pile	file	opérations
Pile()	File()	créer une structure initialement vide
est_vide()	est_vide()	tester si une structure est vide
empile()	enfile()	ajouter un élément à une structure
depile()	defile()	retirer et obtenir un élément d'une structure

REMARQUE

Comme pour les tableaux et les listes chaînées, on préconisé pour les piles et les files une **structures homogènes**. C'est-à-dire que tous les éléments stockés aient le même type.

REMARQUE

Dans ce cours, nos structures de pile et de file seront considérées **mutables** : chaque opération d'ajout ou de retrait d'un élément **modifie la pile ou la file** à laquelle elle s'applique.

```
>>> f = File()
>>> f.enfiler(1)
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> f.entre.depiler()
3
>>> f.entre.depiler()
2
>>> f.entre.depiler()
1
```

```
>>> f = File()
>>> f.enfiler(1)
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> f.defiler()
1
>>> f.defiler()
2
>>> f.defiler()
3
>>> f.defiler()
Traceback (most recent call last):
IndexError: defiler sur file vide
```

```
[ ]: class File:
    def __init__(self):
        """
        Exemples et tests:
        >>> f = File()
        >>> print(type(f.entre))
        <class '__main__.Pile'>
        >>> print(type(f.sortie))
        <class '__main__.Pile'>
        """
        self.entre = Pile()
```

6.3 – Interface et utilisation d'une pile

Mais il aurait été tout à fait possible d'en décider autrement.

interfaçage explicatifs et commentaires

Pile[i].
le type des piles contenanth des
éléments de type T. Par exemple T

est-vide(p: File[T]) -> bool
prend en paramètre une pile p et
chaînes de caractères

acqui paralame et revue une partie
vide capable de contenir n'importe

```
None empiler(p: Pilote[], e: I) -> empiler(p: Pilote[], e: I) -> None
    if p == null {
        p = new Pilote();
        p.empile(e);
    } else {
        empiler(p.suivant(), e);
    }
    return p;
}

Pilote depiler(Pilote p) {
    if (p == null) {
        throw new RuntimeException("la pile est vide");
    }
    Pilote tete = p.tete();
    p = p.suivant();
    return tete;
}

void afficher(Pilote p) {
    if (p == null) {
        System.out.println("la pile est vide");
    } else {
        afficher(p.suivant());
        System.out.println(p.tete());
    }
}
```

z. Un peut alors maintenir l'élément de premier plan avec une pile de sorte, si il y a eu retournement, l'élément apparaît à la nouvelle pile de sorte d'enlever. A moins que cette pile soit encore vide (ce qui signifierait que les deux piles étaient vides), et donc la file est également vide.

de la file après tous les autres (discipline FIFO de la file due l'on veut réaliser), et il sortira bien le dernier puisqu'il sera ajouté le premier sur la pile de sortie (discipline LIFO de la pile utilisée).

implémenter la classe File en utilisant deux fil.
File

```

Exemples et tests :


<>>> f = File()
<>>> print(type(f))
<class '__main__.File'>
<>>> print(f)
<__main__.File object at 0x0000000002A0>
<>>> print(f.__dict__)
{'__class__': <class '__main__.File'>, '__delattr__': <method 'delattr' of 'File' objects>, '__dict__': <attribute '__dict__' of 'File' objects>, '__dir__': <method 'dir' of 'File' objects>, '__doc__': None, '__eq__': <method 'eq' of 'File' objects>, '__format__': <method 'format' of 'File' objects>, '__ge__': <method 'ge' of 'File' objects>, '__getattribute__': <method 'getattribute' of 'File' objects>, '__gt__': <method 'gt' of 'File' objects>, '__hash__': <method 'hash' of 'File' objects>, '__le__': <method 'le' of 'File' objects>, '__lt__': <method 'lt' of 'File' objects>, '__ne__': <method 'ne' of 'File' objects>, '__reduce__': <method 'reduce' of 'File' objects>, '__reduce_ex__': <method 'reduce_ex' of 'File' objects>, '__repr__': <method 'repr' of 'File' objects>, '__setattr__': <method 'setattr' of 'File' objects>, '__str__': <method 'str' of 'File' objects>, '__subclasshook__': <method 'subclasshook' of 'File' objects>}
<>>> f.est_vide()
True
<>>> f.entreer(1)
<>>> f.est_vide()
False
<>>> f.depileer()
<>>> f.est_vide()
True
<>>> f.empileer(1)
<>>> f.est_vide()
False
<>>> f.entreer(1)
<>>> f.est_vide()
True
<>>> f.empileer(1)
<>>> f.est_vide()
False
<>>> f.depileer()
<>>> f.est_vide()
True
<>>> f.entreer(1)
<>>> f.est_vide()
False

```

ACTIVITÉ 10

Exemple

Exemple d'utilisation des piles : Considérons un navigateur web dans lequel on s'intéresse à deux opérations : **aller** à une nouvelle page et **revenir** à la page précédente. On veut que le bouton de retour en arrière permette de remonter une à une les pages précédentes, et ce jusqu'au début de la navigation.

En plus de l'adresse courante, qui peut être stockée dans une variable à part, il nous faut donc conserver l'ensemble des pages précédentes auxquelles il est possible de revenir. *Puisque le retour en arrière se fait vers la dernière page qui a été quittée, la discipline dernier entré, premier sorti des piles est exactement ce dont nous avons besoin pour cet ensemble.*

6.4 – Interface et utilisation d'une file

Comme pour les piles, on note `File[T]` le type des files contenant des éléments de type `T`.

interface	explications et commentaires
<code>File[T]</code>	le type des files contenant des éléments de type <code>T</code>
<code>creer_file() -> File[T]</code>	créer une file vide
<code>est_vide(f: File[T]) -> bool</code>	renvoie <code>True</code> si <code>f</code> est vide et <code>False</code> sinon
<code>enfiler(f: File[T], e) -> None</code>	ajoute l'élément <code>e</code> à la fin de la file <code>f</code>
<code>defiler(f: File[T]) -> T</code>	retirer et renvoyer l'élément situé au début de la file <code>f</code>

S'ajoute un mécanisme liant les deux paquets : une fois la pioche vide on retourne la défausse pour en faire une nouvelle pioche, laissant à la place une défausse vide.

Cette gestion des cartes correspond à une structure de file : une fois la pioche initiale vidée, les cartes seront piochées précisément dans l'ordre dans lequel elles ont été défaussées.

La première défaussee sera la première piochée (FIFO).

On peut donc définir une nouvelle version de la classe `File` utilisant ce principe. Une file réalisée ainsi est caractérisée par deux attributs `entree` et `sortie` :

- le premier contenant une pile dans laquelle on ajoute les nouveaux éléments et
- le second une pile d'où l'on prend les éléments retirés.

Une file ainsi créée est vide lorsque ces deux piles sont toutes les deux vides.

Ajouter un nouvel élément consiste simplement à empiler cet élément sur la pile d'entrée.

Retirer un élément est l'opération la plus délicate.

1. Pour commencer, deux cas de figures :

- Si la pile de sortie **n'est pas vide** (cas simple), il suffit de dépiler son premier élément.
- Sinon la pile de sortie **est vide** (cas délicat). Il faut alors commencer par retourner la pile d'entrée pour la mettre à la place de la pile de sortie. On peut réaliser cette opération intermédiaire en transférant un à un tous les éléments de la pile d'entrée sur la pile de sortie. En effet, le premier élément prélevé sur la pile d'entrée est le dernier entrant (discipline LIFO de la pile utilisée), c'est-à-dire celui qui devra sortir

implémenter le constructeur de la classe Pilote qui constitue une pile vide en définissant son attribut contenu comme la liste vide None.

ACTIVITÉ 1

La structure de liste chaînée donne une manière élémentaire de réaliser une pile. Empiler un nouvel élément revient à ajouter un nouveau million en tête de liste, tandis que dépiler un élément revient à supprimer le million de tête.

On peut ainsi construire une classe Pile définie par un unique attribut contenu associé à l'ensemble des éléments de la pile, stockés sous la forme d'une liste chaînée.

6.5 – Réalisation d'une pile avec une liste chaînée

Exemple d'utilisation des titres : Considérons le jeu de cartes de la partie. Lorsque joueur possède un paquet de cartes et pose à chaque manche la carte prise **sur le dessus du paquet**. Le vainqueur de la manche récupère alors les cartes possédées au-dessous de son paquet.

En plus des cartes possédées au centre de la table nous avons besoin de conserver en mémoire le paquet de cartes de chaque joueur. Puisque les cartes sont remises dans un paquet à une extrémité et prélevées à l'autre, la discipline **premier entre**, premier sorti des files est exactement ce dont nous avons besoin pour chacun de ces ensembles.

– toute carte remise dans la réserve est ajoutée à l'autre pile (la dé-
plaçche),

- toute carte prise dans la réserve est retirée dans l'une de ces piles

Chacun de ces deux paquets de cartes est une pile, et ces deux paquets forment ensemble la réserve de cartes. On a ensuite la discipline suivante :

On prend pour cela modèle sur un jeu de cartes où l'on dispose d'une pile de cartes au sommet de laquelle on prend des cartes (disposées face cachée), et d'une défausse, au sommet de laquelle on dispose (disposées

Illustrons cette construction par un exemple.

Une réalisat^{ion} radicalemeⁿt différente de cette même structure de filee consiste à utiliser deux piles.

6.8 – Réaliser une file avec deux piles

```

class Maillon:
    """ Maillon d'une liste chaînée """
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class Pile:
    """
    Encapsulation des piles à l'aide des Maillons de listes chaînées.
    """
    def __init__(self):
        """
        Constructeur de Pile.

        Exemple et tests:
        >>> p = Pile()
        >>> print(p.contenu)
        None
        """
        self.contenu = None

# tests de la classe
testmod()

```

ACTIVITÉ 2

Étendre la classe Pile en implémentant la méthode est_vide.

Exemples et tests :

```

>>> p = Pile()
>>> print(p.est_vide())
True
>>> p.contenu = Maillon(1, None)
>>> print(p.est_vide())
False

```

```

[ ]: # attention, lorsqu'on implémente une
# classe en une seule fois, il faut écrire
# `class Pile:`.
# La syntaxe `class Pile(Ma_Classe):` est utilisée
# dans ce notebook (et dans les juges en lignes)
# pour étendre la classe existante Ma_Classe et lui
# ajouter de nouvelles méthodes.
#
# Ici, la classe Pile s'étend elle même !
class Pile(Pile):
    def est_vide(self) -> bool:
        """

```

```

>>> print(v)
1
>>> v = p.depiler()
Traceback (most recent call last):
IndexError: depiler sur une pile vide

```

```

[ ]: class Pile:
    """
    Classe Pile
    (implémentée avec les tableaux Python)
    """
    def __init__(self):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> print(p.contenu)
        []
        """
        self.contenu = []

    def est_vide(self):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> print(p.est_vide())
        True
        """
        return self.contenu == []

    def empiler(self, valeur):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> p.empiler(1)
        >>> print(p.est_vide())
        False
        >>> print(p.contenu)
        [1]
        >>> p.empiler(2)
        >>> p.empiler(3)
        >>> print(p.contenu)
        [1, 2, 3]
        """
        self.contenu.append(valeur)

    def depiler(self):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> p.empiler(1)
        >>> p.empiler(2)
        >>> p.empiler(3)
        >>> v = p.depiler()
        >>> print(v)

```

```

    bool: True si et seulement si la pile est vide
    Returns:
        Est ce que la pile est vide
        est
        true si et seulement si la pile est vide
        false
        p = file()
        p.read()
        p.est_vide()
        p = None
        None
        return self.contenu is None
testmode()

```

ACTIVITÉ 3

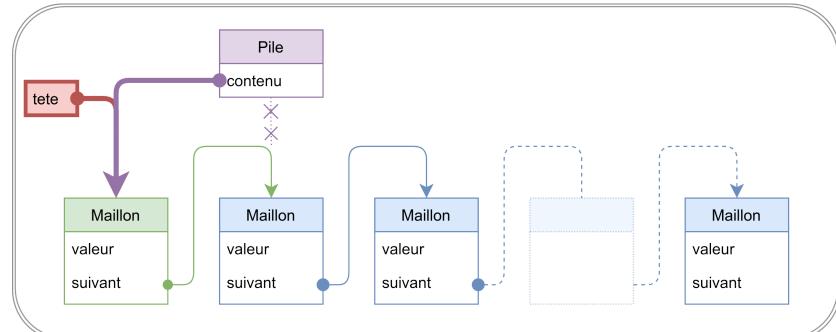
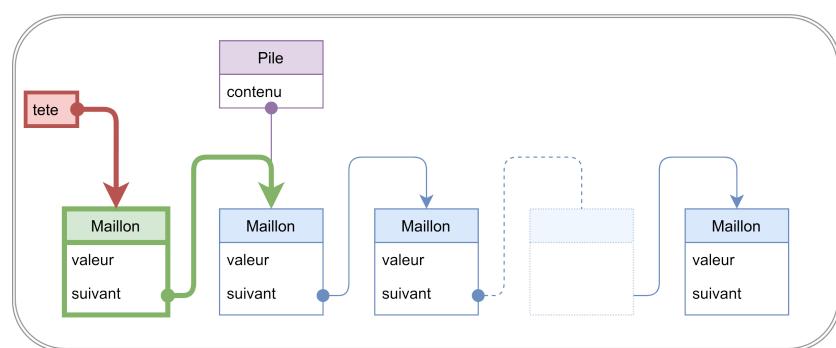
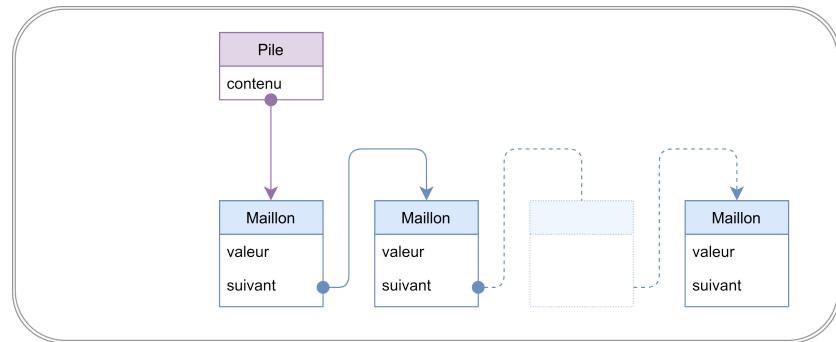
implémenter la méthode `empiler` dans la classe `Pile`. Pour cela construire une nouvelle liste chaînée dont le premier maillon contient :

- valuer : la valeur à empiler
- suivant : le premier million de la liste d'origine de la pile

Example et tests

```
<<< p = File()
```

```
>>> assert not p.est_vide()
>>> print(p.contenu.valeur)
1
>>> p.empiler(2)
>>> print(p.contenu.valleur)
2
```



```
[ ]: # attention, lorsqu'on implémente une
# classe en une seule fois, il faut écrire
# `class Pile:`.
# La syntaxe `class Pile(Ma_Classe):` est utilisée
```

```
[ ]: class File(File):
    def defiler(self):
        """
        Supprime et renvoie le premier élément de la file
        Raises:
            IndexError: si la file est vide
        Returns:
            T: valeur du premier élément de la file
        Exemples et tests:
        >>> f = File()
        >>> f.enfiler(1)
        >>> f.enfiler(2)
        >>> f.enfiler(3)
        >>> print(f.defiler())
        1
        >>> assert f.queue.valeur == 3
        >>> print(f.defiler())
        2
        >>> assert f.queue.valeur == 3
        >>> print(f.defiler())
        3
        >>> assert f.tete == None
        >>> assert f.queue == None
        >>> f.defiler()
        Traceback (most recent call last):
        IndexError: defiler sur une file vide
        """
        if self.est_vide():
            raise IndexError("defiler sur une file vide")

        valeur = self.tete.valeur
        self.tete = self.tete.suivant

        if self.tete is None:
            self.queue = None

        return valeur

testmod()
```

6.7 – Réalisation d'une pile avec les tableaux de Python



ACTIVITÉ 9

Les tableaux de Python réalisent également directement une structure de pile, avec leurs opérations `append` et `pop` qui s'exécutent en moyenne en temps constant. Cette richesse des tableaux redimensionnables propre

Pour finir, implementer `depjLer` afin de récupérer la valeur au sommet de la pile.

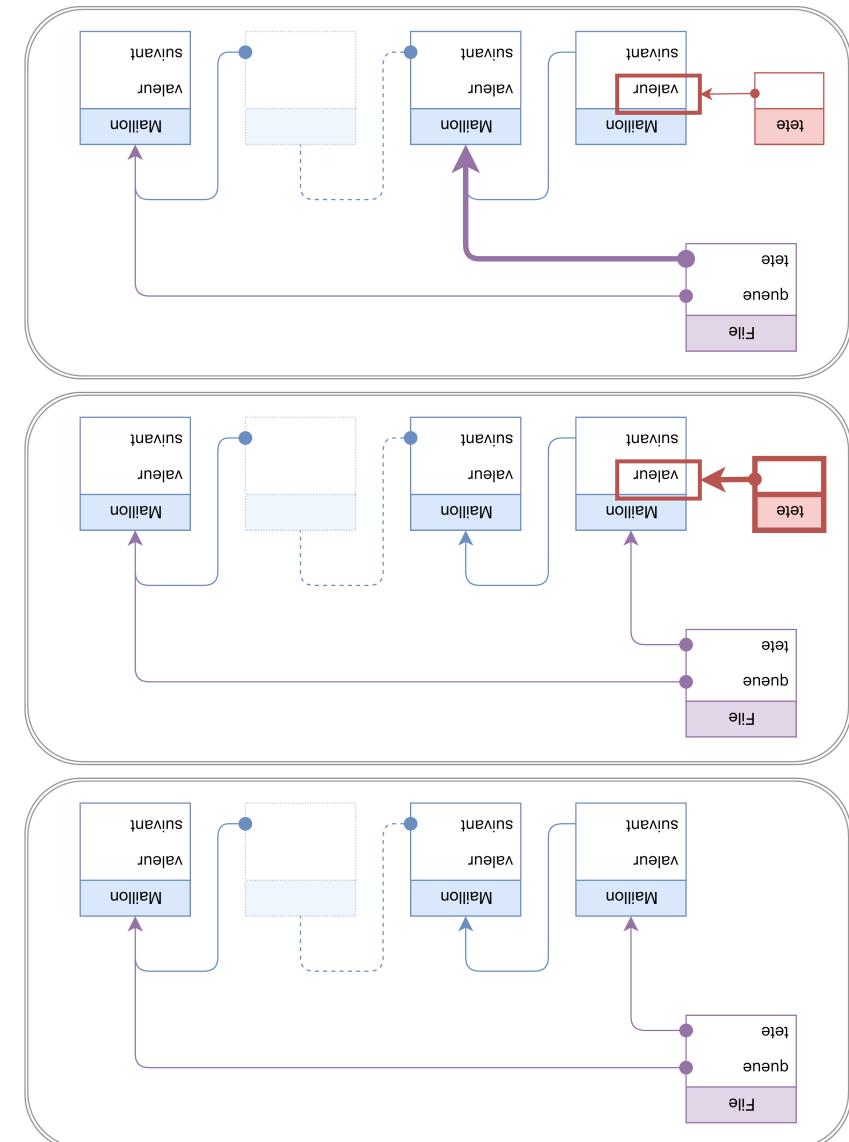
`Si la pile est vide, lever une exception indiquant : "IndexError".`

`Si la pile suivant du mailloin supprimé.`

`de la liste chaînée. Pour cela, le nouveau mailloin de tête doit être le mailloin suivant du mailloin supprimé.`

`Enfin, après la mise à jour de la liste chaînée, il faut renvoyer la valeur qui avait été prélevée dans le mailloin de tête d'origine.`

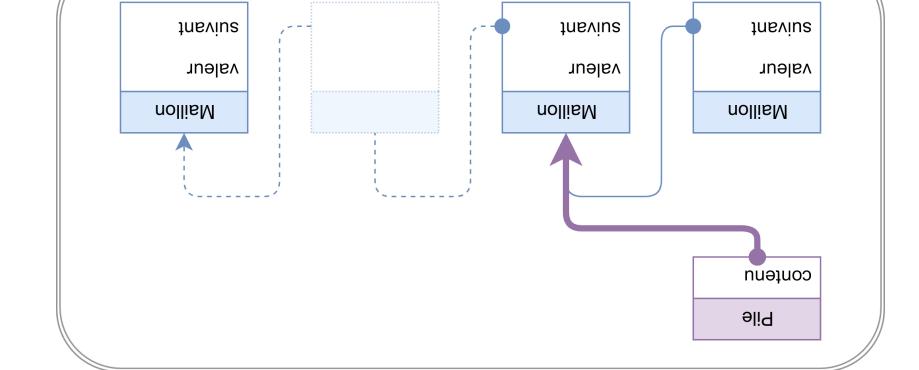
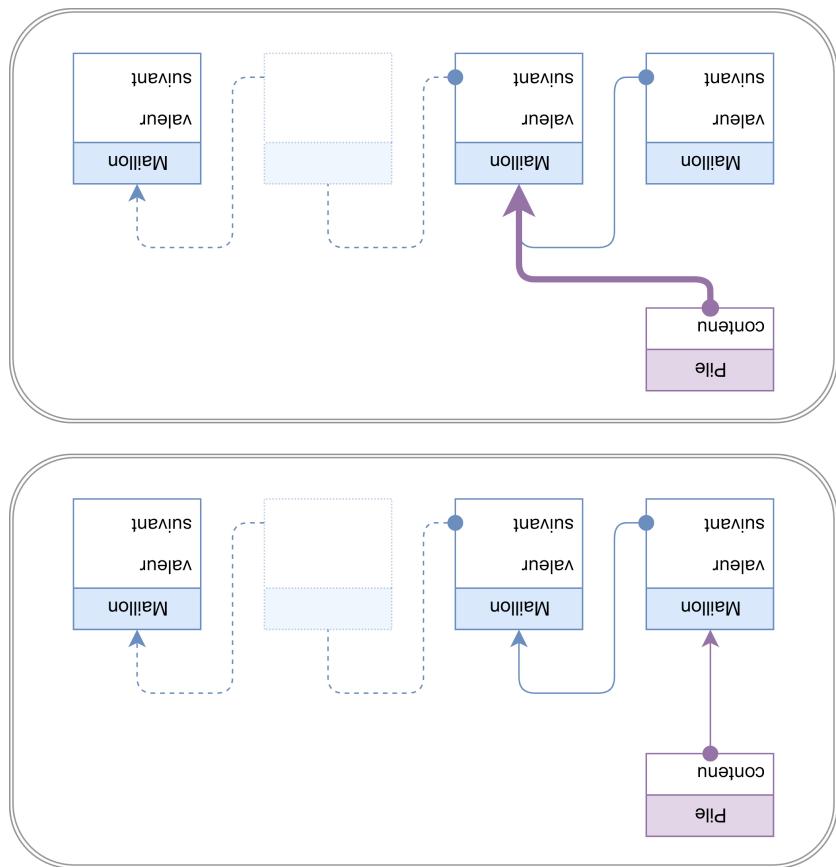
ACTIVITÉ 4



Exemple et tests :

```
>>> p = Pile()
>>> p.empiler(1)
>>> p.empiler(2)
>>> v = p.depiler()
>>> print(v)
2
>>> v = p.depiler()
>>> print(v)
1
>>> v = p.depiler()
Traceback (most recent call last):
IndexError: depiler sur une pile vide
```

```
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> print( f.defiler() )
1
>>> assert f.queue.valeur == 3
>>> print( f.defiler() )
2
>>> assert f.queue.valeur == 3
>>> print( f.defiler() )
3
>>> assert f.tete == None
>>> assert f.queue == None
>>> f.defiler()
Traceback (most recent call last):
IndexError: defiler sur une file vide
```



```

    # Ici, la classe File s'etend elle même !
    # pour étendre la classe estante File, il faut écrire
    # dans ce notebook (et dans les lignes en lignes)
    # pour ajouter des méthodes.

    # La signature classe File(MaClasse) : est utilisée
    # lorsque on implémente une classe File.

    # attention, lorsqu'on implémente une
    # classe en une seule fois, il faut écrire
    # class File:
        # dans ce notebook (et dans les lignes en lignes)
        # pour ajouter des méthodes.

    class File(Pile):
        """Défille la valeur de tête de la pile.

        def dépiler(self):
    
```

implémenter la méthode `dépiler()` qui retire le premier million de la file et renvoie la valeur de ce million.

ACTIVITÉ 8

```

    >>> f = File()
    >>> f.énfiler(1)
    Examples et tests :
    
```

Pour retirer un élément il s'agit de supprimer le premier million de la file, exactement comme il avait été fait lors de l'utilisation d'une chaîne pour réaliser l'attribut `self.queue`. Pour éviter de faire cela à chaque fois qu'une file vide a une pile. Cependant, si le million retiré est le dernier, on veut également redéfinir `File.vider()`. afin de maintenir notre invariant que une file vide a deux attributs qui valent `None`.

```

    self.queue = nouveau
    self.queue.suivant = nouveau
    else:
        self.tete = nouveau
        self.set_vide()
    nouveau = Mailion(valleur, None)
    """
    primitif(f, queue.valleur)
    >>> primitif(f.tete.valleur)
    >>> f.énfiler(3)
    >>> f.énfiler(2)
    >>> f.énfiler(1)
    >>> primitif(f.queue.valleur)
    >>> primitif(f.tete.valleur)
    >>> f.set_vide()
    Examples et tests :
    valleur (1) : valeur à ajouter
    args:
    ajoute la valeur à la fin de la file
    def enfile(self, valleur):
    
```

```

Raises:
    IndexError: si la pile est vide

Returns:
    T: valeur de tête de la pile

Exemple et tests:
>>> p = Pile()
>>> p.empiler(1)
>>> p.empiler(2)
>>> v = p.depiler()
>>> print(v)
2
>>> v = p.depiler()
>>> print(v)
1
>>> v = p.depiler()
Traceback (most recent call last):
IndexError: depiler sur une pile vide
"""
if self.est_vide():
    raise IndexError("depiler sur une pile vide")

tete = self.contenu
valeur_tete = tete.valeur
maillon_suivant = tete.suivant
self.contenu = maillon_suivant

return valeur_tete

testmod()

```

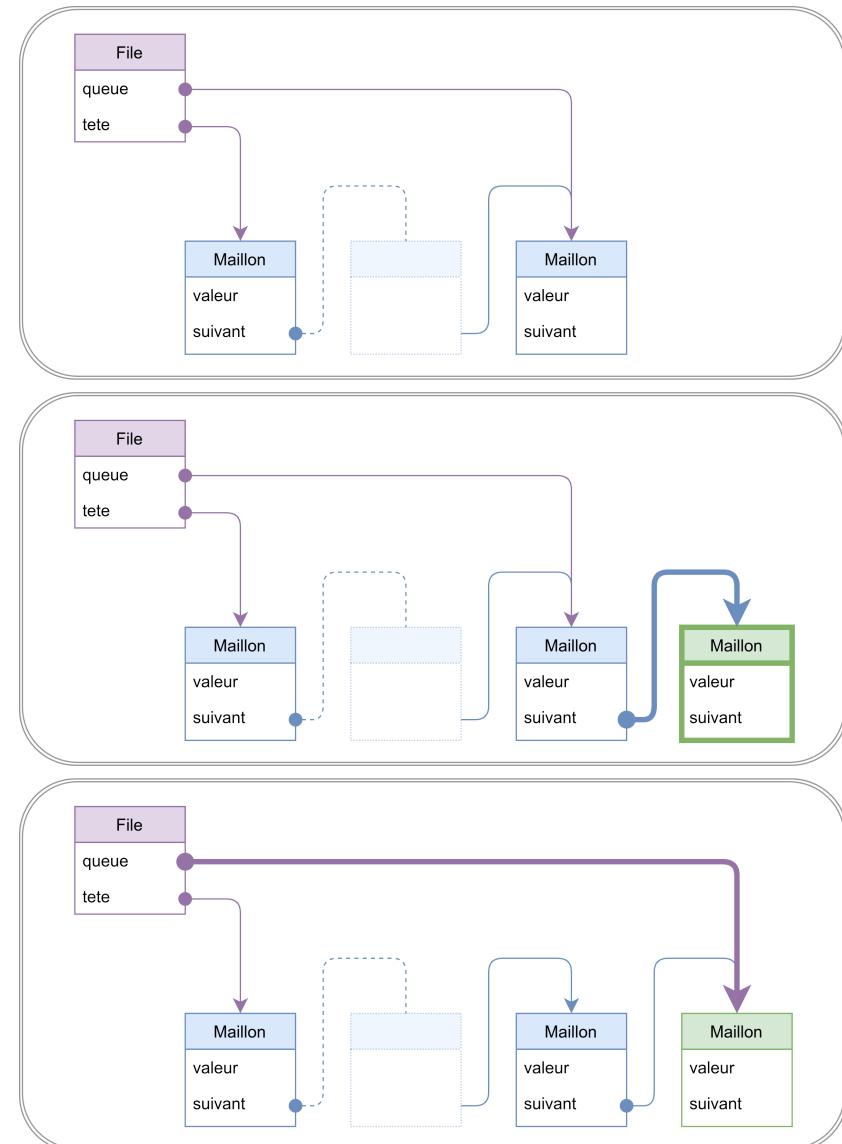
6.6 – Réalisation d'une file avec une liste mutable

La structure de liste chaînée donne également une manière de réaliser une file, à condition de considérer la variante des *listes chaînées mutables*.

En effet, on peut retirer l'élément de tête en retirant le maillon de tête. MAIS, l'ajout d'un nouvel élément à l'arrière de la file revient à ajouter un nouveau maillon en queue de liste. **Une mutation intervient** à cet endroit : alors que le maillon qui était le dernier de la liste chaînée avant l'ajout n'avait pas de suivant défini, il a comme suivant après l'ajout le nouveau maillon créé pour le nouvel élément.

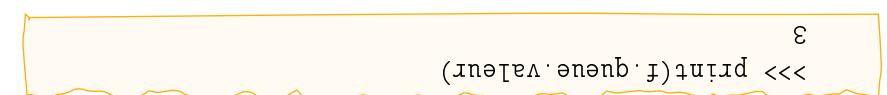
Autre différence avec la structure de pile, il faut accéder efficacement au dernier maillon.

Pour cela, le plus intéressant est de conserver dans notre structure de donnée un





3
 >>> print(f.queue.valeur)



```
testmod()

self.queue = None
self.tete = None
"""
None
>>> print(f.queue)
None
None
>>> print(f.tete)
None
None
>>> f = File()
>>> f = File()
examples et tests:
"""
def __init__(self):
    self.survivant = survivant
    self.valeurs = valeurs
    def __init__(self, valeur, survivant):
        """
        num Mailin d'une liste chaînée num
        [ ] : class Mailin:
class File:
    """
    pass
```

```
None
>>> print(f.queue)
None
None
>>> print(f.tete)
None
>>> f = File()
>>> f = File()
```

Exemples et tests :

On peut ainsi construire une classe File dont le constructeur définit deux attributs, l'un appelé tete et l'autre appelle queue, et désignant respectivement le premier million et le dernier million de la liste chaînée utilisée pour stocker les éléments.

Il suffit de remplir la première tête et la dernière queue, et de définir le constructeur de la classe File qui définit les deux attribut.

implémenter le constructeur de la classe File qui définit les deux attribut.

mais tete et queue sont initialisé à None.

ACTIVITÉ 5

attribut permettant d'accéder directement au dernier million.




ACTIVITÉ 6

La file vide est caractérisée par le fait qu'elle ne contient aucun maillon. En conséquence, sa tête et sa queue sont indéfinies. En outre, l'un comme l'autre ne peut valoir `None` que dans ce cas. Pour tester la vacuité de la file, il suffit donc de consulter l'un des deux attributs.

Implémenter la méthode `est_vide()` qui renvoie `True` si et seulement si l'attribut `tete` vaut `None`.

Exemples et tests :

```
>>> f = File()
>>> print(f.est_vide())
True
>>> f.tete = Maillon(1, None)
>>> print(f.est_vide())
False
```

```
[ ]: # attention, lorsqu'on implémente une
# classe en une seule fois, il faut écrire
# `class File:`.
# La syntaxe `class File(Ma_Classe):` est utilisée
# dans ce notebook (et dans les juges en lignes)
# pour étendre la classe existante Ma_Classe et lui
# ajouter de nouvelles méthodes.
#
# Ici, la classe File s'étend elle même !
class File(File):
    def est_vide(self):
        """
        Est ce que la file est vide?

        Returns:
            bool: True si la file est vide
        Exemples et tests:
        >>> f = File()
        >>> print(f.est_vide())
        True
        >>> f.tete = Maillon(1, None)
        >>> print(f.est_vide())
        False
        """
        return self.tete is None
```


`testmod()`

L'ajout d'un nouvel élément à l'arrière de la file demande de créer un nouveau maillon. Ce maillon prend la dernière place, et n'a donc pas de maillon suivant.

Ce maillon est alors définie comme suivant le maillon de queue actuel.

On a cependant besoin de traiter le cas particulier où il n'existe pas de maillon de queue, qui correspond à une file initialement vide. Dans ce cas le nouveau maillon devient l'unique maillon de la file, et donc son maillon de tête.

Pour finir, dans tous les cas, notre nouveau maillon devient en outre le nouveau maillon de de queue de la file.


ACTIVITÉ 7

En suivant l'algorithme décrit ci-dessus, implémenter la méthode `enfiler` qui admet comme argument une valeur à ajouter en queue de file.

Exemples et tests :

```
>>> f = File()
>>> f.enfiler(1)
>>> print(f.tete.valeur)
1
>>> print(f.queue.valeur)
1
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> print(f.tete.valeur)
1
```