MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

TYPES ABSTRAITS DE DONNÉES - IMPLANTATIONS ET PROPOSITIONS DE MISE EN ŒUVRE

SOMMAIRE

La ressource « Types abstraits de données - Présentation » donne une description des listes, piles et files. L'objet de ce document est de proposer des implantations successives de ces objets théoriques, notamment dans le cadre de la mise en œuvre d'une progression spiralaire du programme de terminale.

Nous avons suivi dans ces ressources un ordre de présentation habituel : listes, puis piles et enfin files. Cet ordre de présentation peut être modifié lors de la présentation aux élèves ; l'ordre piles puis files et, plus tard dans l'année, listes, permet notamment un nouveau réinvestissement de la notion, à choisir notamment en fonction des besoins dans les projets.

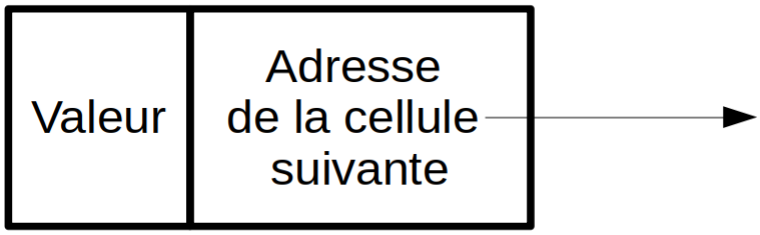
De façon plus spécifique, cette ressource a été conçue de façon à permettre l'illustration de l'utilité de l'interface unique, déconnectée du choix d'implantation du code. Un projet ou exercice, utilisant un type abstrait de données peut évoluer en ne modifiant que l'implantation de la structure et voir ses performances évoluer, sans voir son code propre être modifié.

Introduction de la classe « Cellule »

Cette première partie, indépendante des 3 autres (Structures implémentant les listes, piles et files), n'est nécessaire que lors de l'utilisation des classes pour implanter les différentes structures.

Que cela soit pour les listes, les piles ou les files, ces structures sont définies par un ensemble d'éléments ordonnés et reliés entre eux. Pour faciliter les implantations de ces structures en utilisant **la programmation orientée objet**, nous allons introduire une classe **Cellule** qui permet de définir un objet contenant deux attributs :

- un attribut **valeur** définissant la valeur contenue dans cette "cellule" (nombre, texte, un autre objet de nature quelconque…) ;
- un attribut **suivant** définissant l'adresse d'un **autre objet Cellule** qui "suit" cet objet (qui est l'objet suivant dans l'ordre de la structure).



La structure de base pour les implantations objet : la cellule

Différentes instances de cette classe seront utilisées par les trois structures que nous présentons dans cette ressource. Cela permet d'ordonner les éléments de la structure (des « maillons » pour une liste chaînée, les éléments contenus dans les piles et les files…).

```
"""
Implantation de la classe "Cellule"
Elle sera utilisée pour les listes, les piles et les files.
"""
class Cellule :
    def __init__(self, valeur = None, suivant = None) :
        """
        Paramètres
        ----------
        valeur : type quelconque
            Description : Une valeur stockée dans la cellule
        suivant : Un autre objet de type "Cellule"
            Description : La cellule qui "suit" cette cellule selon l'ordre
            défini par la structure.
        ----------
        Crée une cellule avec une valeur et l'adresse de la cellule qui la suit.
        """
        self.valeur = valeur
        self.suivant = suivant
```

Structures implémentant les Listes

Implantation du type abstrait de données liste avec les listes Python

Une mise en œuvre de la structure réalisant le type abstrait de données liste de la ressource « Types abstraits de données - Présentation » peut être proposée dès le début de la terminale, en n'utilisant que les acquis de première :

```
# Implantation du type Abstrait de données liste avec les listes Python
def listeCree() :
    """ crée une liste vide en s'appuyant sur les listes Python"""
    return []

def listeAjout(liste, element) :
    """ Paramètres
    ----------
    liste : une liste à laquelle on souhaite ajouter un élément
    element : l'élément à ajouter, de type quelconque
    ----------
    ajoute un élément en tête de liste"""
    liste.append(element)

def listeTete(liste) :
    """ Paramètres
    ----------
    liste : une liste
```

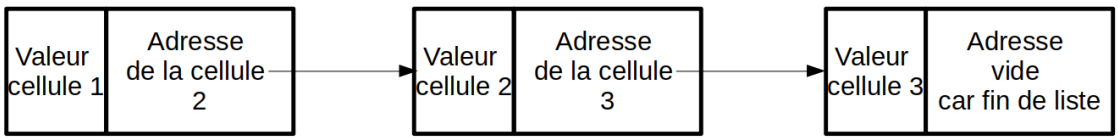
```
    ---------
    renvoie la valeur de l'élément en tête de liste"""
    if not listeEstVide(liste) :
        return liste[-1]
    else :
        return None

def listeQueue(liste) :
    """ Paramètres
    ---------
    liste : une liste
    ---------
    renvoie la queue de la liste """
    return liste[:-1]
    # il peut être pertinent d'utiliser la syntaxe liste.pop() afin d'éviter le slicing

def listeEstVide(liste) :
    """ Paramètres
    ---------
    liste : une liste
    ---------
    renvoie ``True`` si la liste est vide, ``False`` sinon"""
    return len(liste)==0
```

Implantation utilisant la programmation orientée objet

Une fois le cours éponyme réalisé, une implantation utilisant la programmation orientée objet peut être demandée. L'interface proposée se prête mieux à l'encapsulation d'un module "Liste".

Chaque valeur de la liste est encapsulé dans un objet **Cellule** (défini plus haut) qui contient une valeur ainsi que l'adresse de la cellule suivante.

La liste devient alors **liste chaînée**.



Une liste chaînée à trois maillons

Une proposition très minimaliste d'implantation peut être :

```
#On importe la classe "Cellule" que l'on a défini précédemment (et qui doit être dans le même dossier)
from Cellule import Cellule

class Liste :
    def __init__(self) :
        """
        Crée une liste vide.
```

```
        L'attribut "head" est un objet Cellule qui définit la cellule
        en tête de la liste (premier élément de la liste)
        """
        self.head = None

    def estVide(self) :
        """
        Renvoie ``True`` si la liste est vide et ``False`` sinon.
        """
        return self.head == None

    def insererEnTete(self, element) :
        """
        Paramètres
        ----------
        element : N'importe quel type
            Description : L'élément à ajouter en tête de la liste

        -------
        Ajoute un élément en tête de liste.
        """
        nouvelle_cellule = Cellule(element, self.head)
        self.head = nouvelle_cellule

    def tete(self) :
        """
        Renvoie la valeur de l'élément en tête de liste.
        """
        if not(self.estVide()) :
            return self.head.valeur

    def queue(self) :
        """
        Renvoie la liste privée de son premier élément (queue de la liste)
        """
        subList = None
        if not(self.estVide()) :
            subList = Liste()
            subList.head = self.head.suivant
        return subList
```

Celle-ci peut être complétée par des méthodes implantant de nouvelles possibilités décrites succinctement dans la ressource « Types abstraits de données - Présentation ». Par exemple :

- renvoyer la longueur de la liste ;
- accéder au élément d'une liste ;
- ajouter un élément à la fin de la liste ;
- rechercher un élément dans une liste en renvoyant "Vrai" si l'élément est présent, "Faux" sinon.

On peut alors compléter la classe précédente par :

```
def __len__(self) :
    """
    Renvoie le nombre d'éléments de la liste.
    """
    taille  = 0
    celluleCourante = self.head
    while(celluleCourante != None) :
        celluleCourante = celluleCourante.suivant
        taille += 1
    return taille

def __getitem__(self, position) :
    """
    Paramètres
    ----------
    position : Entier positif
        Description : La position de l'élément qu'on veut obtenir dans la liste
    -------
    Renvoie l'élément situé à la position spécifiée en paramètre dans la liste.
    """
    assert position < len(self) , "la position n'existe pas dans la liste"
    celluleCourante = self.head
    for i in range(position) :
        celluleCourante = celluleCourante.suivant
    return celluleCourante.valeur

def ajouterFin(self, element) :
    """
    Paramètres
    ----------
    element : type quelconque
        Description : L'élément à ajouter à la fin de la liste
    -------
    Ajoute un élément à la fin de la liste.
    """
    nouvelle_cellule = Cellule(element, None)
    if self.estVide() :
        self.head = nouvelle_cellule
    else:
        celluleCourante = self.head
        while celluleCourante.suivant != None:
            celluleCourante = celluleCourante.suivant
        celluleCourante.suivant = nouvelle_cellule

def __contains__(self, element) :
    """
```

```
    Paramètres
    ----------
    element : type quelconque
    Description : L'élément qu'on souhaite vérifier
    -------
    Renvoie ``True`` si la liste contient l'élément et ``False`` sinon.
    """
    trouve = False
    celluleCourante = self.head
    while celluleCourante != None :
        if celluleCourante.valeur == element :
            trouve = True
        celluleCourante = celluleCourante.suivant
    return trouve
```

Implantation avec la récursivité et la programmation orientée objet

Dans le déroulement de l'année, une fois la récursivité travaillée, ou en application de celle-ci, une nouvelle implantation peut être demandée. Les structures de données peuvent alors être vues de nombreuses fois, avec une structure différente, permettant l'assimilation de la différence entre type abstrait de données et structure. De même, une évolution de l'implantation précédente peut être créée, sans changer le programme qui l'utilise.

Par exemple, on peut ajouter ces deux méthodes récursives dans la classe **Cellule** :

```
def nombreCellules(self) :
    """
    Retourne le nombre de cellules accessibles à partir de cette cellule.
    On compte la cellule + ses voisins.
    """
    if self.suivant == None :
        return 1 #Cas de base
    else :
        return 1 + self.suivant.nombreCellules() #Cas récursif


def element(self, distance) :
    """
    Paramètres
    ----------
    distance : Nombre entier positif
        Description : Distance de l'élément auquel on veut accéder
    -------
    Retourne l'élément se trouvant à une distance spécifiée de la cellule
    La distance 0 donne donc la valeur de la case courante, 1 celle de son
    voisin immédiat, etc.

    """
```

```
        assert distance < self.nombreCellules(), "la position n'existe pas"
        if distance == 0 :
            return self.valeur #Cas de base
        else :
            return self.suivant.element(distance - 1) #Cas récursif
```

Ce qui permet de considérablement raccourcir le code des méthodes suivantes dans la classe **Liste** :

```
    def __len__(self) :
        """
        Renvoie le nombre d'éléments de la liste.
        """
        if self.estVide() :
            return 0
        else :
            return self.head.nombreCellules()

    def __getitem__(self, position) :
        """
        Paramètres
        ----------
        position : Entier positif
            Description : La position de l'élément qu'on veut obtenir dans la liste
        -------
        Renvoie l'élément situé à la position spécifié en paramètre dans la liste.
        """
        assert position < len(self) , "la position n'existe pas dans la liste"
        return self.head.element(position)
```

Il est tout à fait possible (en ajoutant des méthodes à **Cellule**) d'implanter un fonctionnement récursif d'autres primitives.

Structures implémentant les Piles

Implantation du type abstrait de données Pile avec les listes Python

Une mise en œuvre de la structure réalisant le type abstrait de données **pile** de la ressource « Types abstraits de données - Présentation » peut être proposée dès le début de la terminale, en n'utilisant que les acquis de première :

```
# Implantation du type Abstrait de données pile avec les listes Python
def pileCree() :
    """
    Crée une pile vide en s'appuyant sur les listes Python
    """
    return []

def pileTaille(pile):
```

```
    """
    Paramètres
    ----------
    pile : Une pile, telle que créée dans ce module (utilisant une list Python)
        Description : La pile dont on veut connaître le nombre d'éléments
    -------
    Renvoie le nombre d'éléments contenus dans la pile.
    """
    return len(pile)

def pileEstVide(pile) :
    """
    Paramètres
    ----------
    pile : Une pile, telle que créée dans ce module (utilisant une list Python)
        Description : La pile dont on souhaite déterminer si elle est vide.
    -------
    Renvoie ``True`` si la pile est vide, et ``False`` sinon
    """
    return pileTaille(pile) == 0

def pileEmpiler(pile, element) :
    """
    Paramètres
    ----------
    pile : Une pile, telle que créée dans ce module (utilisant une list Python)
        Description : La pile sur laquelle on souhaite empiler un élément
    element : N'importe quel type de données.
        Description : L'élément à empiler dans la pile.
    -------
    Empile un élément dans la pile passée en paramètres.
    """
    pile.append(element)

def pileDepiler(pile) :
    """
    Paramètres
    ----------
    pile : Une pile, telle que créée dans ce module (utilisant une list Python)
        Description : La pile sur laquelle on souhaite dépiler un élément
    -------
    Dépile (supprime de la pile) l'élément au sommet de la pile et le renvoie.
    """
    if pileEstVide(pile) :
        return None
    else :
        return pile.pop()

def pileSommet(pile) :
    """
```

```
    Paramètres
    ----------
    pile : Une pile, telle que créée dans ce module (utilisant une list Python)
        Description : La pile dont on veut connaître le sommet.
    -------
    Renvoie le sommet de la pile (mais ne le dépile pas).
    """
    if pileEstVide(pile) :
        return None
    else :
        return pile[len(pile) - 1]
```

Implantation des piles utilisant la programmation orientée objet

Une fois le cours éponyme réalisé, une implantation utilisant la programmation orientée objet peut être demandée.

Chaque valeur contenue dans la pile est encapsulée dans un objet **Cellule** (défini plus haut) qui contient une valeur ainsi que l'adresse de la cellule suivante.

Un attribut **top** est attaché à la pile. C'est en fait un objet de type **Cellule** correspondant au sommet.

À partir du sommet, on peut donc accéder à chaque valeur contenue dans la pile.

L'empilement et le dépilement va donc consister à « mettre à jour » l'attribut **top** de cet objet.

Une proposition d'implantation peut être :

```
"""
On importe la classe "Cellule" qu'on a défini précédemment.
Doit être dans le même dossier
"""
from Cellule import Cellule

class Pile :
    def __init__(self) :
        """
        Crée une pile vide.
        L'attribut "top" est un objet Cellule qui définit la cellule
        constituant le "sommet" de la pile.
        """
        self.top = None

    def estVide(self) :
        """
        Renvoie ``True`` si la pile est vide et ``False`` sinon.
        """
        return self.top == None

    def sommet(self) :
```

```
        """
        Renvoie la valeur de l'élément au sommet de la pile.
        """
        if not(self.estVide()) :
            return self.top.valeur
        else :
            return None

    def empiler(self, element) :
        """
        Paramètres
        ----------
        element : est de n'importe quel type
            Description : L'élément à empiler sur la pile.
        -------
        Ajoute un élément au sommet de la pile.
        """
        nouvelleCellule = Cellule(element, self.top)
        self.top = nouvelleCellule

    def depiler(self) :
        """
        Dépile et renvoie l'élément situé au sommet de la pile.
        """
        if not(self.estVide()) :
            valeur = self.top.valeur
            self.top = self.top.suivant
            return valeur
        else :
            return None

    def __len__(self) :
        """
        Renvoie le nombre d'éléments de la pile.
        """
        taille  = 0
        celluleCourante = self.top
        while(celluleCourante != None) :
            celluleCourante = celluleCourante.suivant
            taille += 1
        return taille
```

On remarque que la méthode **len** est **identique** à celle utilisée pour les listes. Cela montre que ces deux structures partagent des similitudes dans leur organisation.

Implantation des piles avec la récursivité et la programmation orientée objet

De la même manière que pour les **listes**, on peut remplacer la méthode suivante par une version exploitant la classe **Cellule** de manière récursive :

```
def __len__(self) :
    """
    Renvoie le nombre d'éléments de la pile.
    """
    if self.estVide() :
        return 0
    else :
        return self.sommet.nombreCellules()
```

Pour cela, il faut déjà avoir implémenté la méthode **nombreCellules** dans la classe **Cellule** (voir la section sur les listes).

Prolongements

Au-delà des implantations des piles présentées dans ce document, il est aussi possible d'utiliser **une bibliothèque Python** permettant de reproduire le comportement d'une structure de **pile**. Cette librairie est nommée **deque** et peut s'utiliser à la fois pour les **piles** et les **files**. Concernant les **piles**, on peut proposer l'exemple d'utilisation :

```
from queue import deque

p = deque() #Création d'une pile vide.
p.append(element) #Ajout d'un élément dans la pile.
p[-1] #Sommet de la pile
p.pop() #Dépilement.
len(p) #Taille de la pile
len(p) == 0 #Vérifier si la pile est vide ou non
```

Attention, les méthodes permettant d'utiliser cette pile sont très semblables à celles d'une **list** (Python).

Comparaison d'efficacité pour les piles

Afin de comparer l'efficacité des différentes structures qui implantent les **piles**, nous allons mesurer le temps d'exécution des deux opérations principales, **l'empilement** et **le dépilement**.

Attention toutefois, le temps présenté ici **est seulement indicatif** pour permettre de **comparer** les structures. En effet, **le temps d'exécution ne sera pas le même selon l'ordinateur**, son état, etc. Néanmoins, peu importe la machine, les différences de temps observées entre l'exécution de ces opérations sur les structures permet d'estimer si une structure est plus efficace qu'une autre.

Pour avoir un temps d'exécution assez représentatif, nous effectuons un **empilement** puis un **dépilement** de **10 millions** d'éléments.

Opération - Empilement

Type de pile	Temps d'exécution
Pile avec des listes Python	1,55 secondes
Pile objet	9,67 secondes
Pile avec bibliothèque deque	0,54 secondes

Opération - Dépilement

Type de pile	Temps d'exécution
Pile avec des listes Python	3,5s secondes
Pile objet	3,6s secondes
Pile avec bibliothèque deque	0,56 secondes

On remarque que, de manière générale, l'implantation objet **est la moins performante** et l'implantation avec la bibliothèque est **la plus performante**. Cela s'explique notamment car l'implantation avec les listes Python et celle de deque repose sur une programmation avancée et optimisée, contrairement à l'implantation objet qui a été construite sans utiliser de structure externe.

Néanmoins, de manière générale, les ordres de grandeur de complexité sont les mêmes pour les deux opérations pour chaque structure :

- pour l'implantation avec les **listes** Python, cela repose sur les fonctions **append** et **pop** qui sont toutes les deux en temps constant 0(1) ;
- pour l'implantation objet, on n'effectue pas de parcours, on manipule seulement l'attribut **top** et on crée de nouveaux objets de type **Cellule**, avec quelques opérations intermédiaires (ce qui explique la différence observée). On est pratiquement en temps constant 0(1) ;
- pour l'implantation avec **deque**, les fonctions **append** et **pop** sont aussi en temps constant 0(1) (on remarque d'ailleurs que le temps d'exécution de empilement / dépilement est quasiment identique).

La librairie **deque** reste **la plus efficace** (car utile pour la manipulation efficace de piles ou de files, contrairement aux **listes** Python de la première version), et il est recommandé de l'utiliser si vous souhaitez avoir un code performant. Néanmoins, l'implantation **objet** a pour avantage de montrer et d'assimiler le fonctionnement interne de cette structure. C'est donc un meilleur outil d'apprentissage des concepts, contrairement aux librairies qui agissent comme des « boîtes noires ».

Usage dans les parcours de graphes et arbres

Les piles sont utiles dans de nombreux problèmes. Elles permettent notamment de réaliser le **parcours en profondeur** d'un **arbre** ou bien d'un **graphe**, qui sont deux autres structures abordées lors de l'année de terminale. Le **parcours en profondeur** consiste à explorer les nœuds d'un arbre (ou les sommets d'un graphe) dans un certain ordre, suivant la logique **d'empilement** et de **dépilement** des éléments d'une **pile**. Il permet

notamment de trouver un chemin d'un sommet (ou d'un nœud) à un autre (si ce chemin existe), de détecter des cycles dans des graphes.

Par exemple, si on considère un problème où un arbre modélise un **labyrinthe**, cet algorithme permet de trouver un chemin vers la sortie. Sur un graphe représentant un réseau de villes, cela permet de déterminer un trajet allant d'une ville de départ jusqu'à une destination souhaitée (et, par exemple, de déterminer tous les trajets possibles…).

Structures implémentant les Files

Implantation du type abstrait de données File avec les listes Python

Une mise en œuvre de la structure réalisant le type abstrait de données **file** de la ressource « Types abstraits de données - Présentation » peut être proposée dès le début de la terminale, en n'utilisant que les acquis de première :

```
# Implantation du type Abstrait de données file avec les listes Python.
def fileCree() :
    """
    Crée une file vide en s'appuyant sur les listes Python.
    """
    return []

def fileTaille(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (utilisant donc une listes Python)
        Description : La file dont on veut connaître le nombre d'éléments
    -------
    Renvoie le nombre d'éléments contenus dans la file
    """
    return len(file)

def fileVide(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (utilisant donc une liste Python)
            Description : La file qu'on veut vérifier.
    -------
    Renvoie ``True`` si la file est vide, et ``False`` sinon
    """
    return fileTaille(file) == 0

def fileAjouterFin(file, element) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (utilisant donc une liste Python)
```

```
    Description : La file à laquelle on souhaite ajouter un élément
  element : N'importe quel type de données.
    Description : L'élément à ajouter au bout de la file.
  -------
  Ajoute un élément au bout de la file passée en paramètres.
  """
  file.append(element)

def fileRetirerTete(file) :
  """
  Paramètres
  ----------
  file : Une file, telle que créée dans ce module (utilisant donc une liste Python)
    Description : La file à laquelle on souhaite retirer un élément
  -------
  Retirer (supprime de la file) et renvoie l'élément en tête (situé au début)
  de la file.
  """
  if fileVide(file) :
    return None
  else :
    return file.pop(0)

def fileTete(file) :
  """
  Paramètres
  ----------
  file : Une file, telle que créée dans ce module (utilisant donc une liste Python)
    Description : La file dont on veut connaître la tête.
  -------
  Renvoie l'élément en tête (situé au début) de la file (mais ne le supprime pas)
  """
  if fileVide(file) :
    return None
  else :
    return file[0]
```

Implantation du type abstrait de données File avec deux piles

Comme le programme le suggère, il est possible d'implanter une **file** en utilisant deux **piles**. Le procédé est le suivant :

- la file est, au départ, composée de deux piles vides ;
- la première pile est une pile dite « d'entrée » et la seconde « de sortie » ;
- quand on ajoute un élément dans la file, on le place dans la pile « d'entrée » ;
- Quand on retire (ou qu'on accède) au premier élément de la file, on a deux cas :
  - soit la pile « de sortie » est vide et on dépile chaque élément de la pile « d'entrée » pour les empiler immédiatement dans la pile « de sortie » ;
  - soit il y a au moins un élément dans la pile « de sortie », auquel cas on ne fait rien de plus. Enfin, on sélectionne le sommet de la pile « de sortie » ;
- comme il y a deux piles, la taille de la file (et le fait qu'elle soit vide ou non) doit se baser sur les éléments contenus dans les deux piles.

Dans notre implantation, on propose de matérialiser la file sous la forme d'un **tuple** contenant deux **piles**, crées (et manipulées) avec les méthodes du module modélisant le type abstrait de données pile, définit plus tôt dans la section sur les **piles**. On introduit également une nouvelle méthode « organiser » qui sert à effectuer le transfert entre les piles (si nécessaire) avant de retirer ou de récupérer le premier élément de la file.

```
#Il faut importer le module contenant la définition du type abstrait de données Pile
# "ModulePileAbstrait" correspond au nom du fichier contenant ce module.
from ModulePileAbstrait import *

# Implantation du type Abstrait de données file avec deux piles
def fileCree() :
    """
    Crée une file vide sous la forme d'un tuple composé de deux piles.
    La première pile correspond à la pile 'd'entrée', la seconde à celle 'de sortie'
    """
    return (pileCree(), pileCree())

def fileTaille(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (utilisant donc un tuple contenant
deux piles)
        Description : La file dont on veut connaitre le nombre d'éléments
    -------
    Renvoie le nombre d'éléments contenus dans la file
    """
    return pileTaille(file[0]) + pileTaille(file[1])

def fileVide(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (utilisant donc un tuple contenant
deux piles)
            Description : La file qu'on veut vérifier.
    -------
    Renvoie ``True`` si la file est vide, et ``False`` sinon
    """
    return pileEstVide(file[0]) and pileEstVide(file[1])

def fileOrganiser(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (utilisant donc un tuple contenant
deux piles)
        Description : La file que l'on souhaite organiser.'
```

```
    -------
    Si la pile de sortie est vide, dépile chaque élément de
    la pile d'entrée pour les empiler dans la pile de sortie.
    Cela permet de "mettre à jour" les éléments qui doivent sortir.
    """
    if pileEstVide(file[1]) :
        while not(pileEstVide(file[0])) :
            pileEmpiler(file[1], pileDepiler(file[0]))

def fileAjouterFin(file, element) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (donc un tuple contenant deux piles)
        Description : La file à laquelle on souhaite ajouter un élément
    element : N'importe quel type de données.
        Description : L'élément à ajouter à la file.
    -------
    Ajoute un élément au bout de la file passée en paramètres.
    """
    pileEmpiler(file[0], element)

def fileRetirerTete(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (donc un tuple contenant deux piles)
        Description : La file à laquelle on souhaite retirer un élément
    -------
    Retirer (supprime de la file) et renvoie l'élément en tête (situé au début)
    de la file.
    """
    fileOrganiser(file) #On organise la file avant de retirer la tête.
    return pileDepiler(file[1])

def fileTete(file) :
    """
    Paramètres
    ----------
    file : Une file, telle que créée dans ce module (donc un tuple contenant deux piles)
        Description : La file dont on veut connaître la tête.
    -------
    Renvoie l'élément en tête (situé au début) de la file (mais ne le supprime pas)
    """
    fileOrganiser(file) #On organise la file avant de récupérer la tête.
    return pileSommet(pile[1])
```

Implantation des files utilisant la programmation orientée objet

Une fois le cours éponyme réalisé, une implantation utilisant la programmation orientée objet peut être demandée.

Chaque valeur contenue la file est encapsulé dans un objet **Cellule** (définit plus haut) qui contient une valeur ainsi que l'adresse de la cellule suivante.

Deux attributs (de type **Cellule**) sont attachés à la file :

- un attribut **head** correspondant à la tête (sortie) de la file. C'est ici qu'on récupère les éléments qu'on retire de la file ;
- un attribut **end** correspondant au bout (entrée) de la file. C'est à partir de cet emplacement qu'on ajoute des éléments à la file.

Ajouter un élément à la file consiste en une opération où l'on modifie l'attribut **end** de la file en créant un nouvel objet **Cellule** et en affectant correctement les attributs de l'objet. Retirer un élément consiste à modifier l'attribut **head** en récupérant les valeurs de la cellule correspondante et en réaffectant cet attribut à la cellule suivante. Le seul cas particulier auquel il faut faire attention est l'ajout du premier élément : il correspond à la fois à l'attribut **head** et **end** (c'est à la fois la tête et le bout de la file, donc même cellule).

Une proposition d'implantation peut être :

```
"""
On importe la classe "Cellule" qu'on a défini précédemment.
Doit être dans le même dossier
"""
from Cellule import Cellule

class File :

    def __init__(self) :
        """
        Créé une file vide.
        L'attribut "head" est un objet Cellule qui définit la cellule
        constituant la tête (sortie) de la file.
        L'attribut "end" est un objet Cellule qui définit la cellule
        constituant le bout (entrée) de la file.
        """
        self.head = None
        self.end = None

    def estVide(self) :
        """
        Renvoie ``True`` si la file est vide et ``False`` sinon.
        """
        return self.head == None

    def tete(self) :
        """
```

```
        Renvoie la valeur de l'élément en tête de la file (premier élément).
        """
        if not(self.estVide()) :
            return self.head.valeur
        else :
            return None

    def ajouter(self, element) :
        """
        Paramètres
        ----------
        element : N'importe quel type
            Description : L'élément à ajouter au bout de la file.
        -------
        Ajoute un élément au bout de la file.
        """
        dernierCellule = Cellule(element, None)
        if(self.estVide()) :
            #Cas particulier si la file ne contient rien. La tête == Le bout de la file.
            self.head = dernierCellule
        else :
            self.end.suivant = dernierCellule
        self.end = dernierCellule

    def retirerTete(self) :
        """
        Retire et renvoie l'élément situé à la tête de la file (premier élément).'
        """
        if not(self.estVide()) :
            valeur = self.head.valeur
            self.head = self.head.suivant
            return valeur
        else :
            return None

    def __len__(self) :
        """
        Renvoie le nombre d'éléments de la file.
        """
        taille  = 0
        celluleCourante = self.head
        while(celluleCourante != None) :
            celluleCourante = celluleCourante.suivant
            taille += 1
        return taille
```

On remarque que la méthode **len** est **identique** à celle utilisée pour les listes (et les piles). Cela montre, encore une fois, que ces structures partagent des similitudes dans leur organisation.

Implantation des files avec la récursivité et la programmation orientée objet

De la même manière que pour les **listes**, on peut remplacer la méthode suivante par une version exploitant la classe **Cellule** de manière récursive :

```
def __len__(self) :
    """
    Renvoie le nombre d'éléments de la file.
    """
    if self.estVide() :
        return 0
    else :
        return self.head.nombreCellules()
```

Pour cela, il faut déjà avoir implémenté la méthode **nombreCellules** dans la classe **Cellule** (voir la section sur les listes). Par ailleurs, comme les listes et les files ont un attribut de même nom « **head** » pour désigner le début de la structure, les méthodes **len** de ces deux structures sont identiques.

Implantation avec deux piles et la programmation orientée objet

L'implantation avec deux piles est aussi possible en utilisant la programmation orientée objet. La logique reste la même, il suffit d'utiliser deux attributs objets **Piles** (à la place du tuple) tel que définis dans la section correspondante, sur l'implantation objet des piles.

```
#Il faut importer la classe contenant la définition de la classe Pile
# "FichierPile" correspond au nom du fichier contenant cette classe.
from FichierPile import Pile

class File :

    def __init__(self) :
        """
        Créé une file vide.
        L'attribut "pile_entree" est un objet Pile qui stocke les nouvelles
        valeurs ajoutées à la file.
        L'attribut "pile_sortie" est un objet Pile qui stocke les valeurs
        qui vont sortir de la file. Le sommet de cette pile est le prochain
        élément qui doit sortir (premier de la file).
        """
        self.pile_entree = Pile()
        self.pile_sortie = Pile()

    def estVide(self) :
        """
        Renvoie ``True`` si la file est vide et ``False`` sinon.
        """
        return self.pile_entree.estVide() and self.pile_sortie.estVide()

    def organiser(self) :
```

```
        """
        Si la pile_sortie est vide, dépile chaque élément de
        la pile_entree pour les empiler dans pile_sortie.
        Cela permet de "mettre à jour" les éléments qui doivent sortir.
        """
        if self.pile_sortie.estVide() :
            while not(self.pile_entree.estVide()) :
                self.pile_sortie.empiler(self.pile_entree.depiler())

    def tete(self) :
        """
        Renvoie la valeur de l'élément en tête de la file (premier élément).
        """
        self.organiser() #On organise la file avant de récupérer la tête.
        return self.pile_sortie.sommet()

    def ajouter(self, valeur) :
        """
        Paramètres
        ----------
        element : N'importe quel type
            Description : L'élément à ajouter au bout de la file.
        -------
        Ajoute un élément au bout de la file.
        """
        self.pile_entree.empiler(valeur)

    def retirerTete(self) :
        """
        Retire et renvoie l'élément situé en tête (premier élément).'
        """
        self.organiser() #On organise la file avant de retirer la tête.
        return self.pile_sortie.depiler()

    def __len__(self) :
        """
        Renvoie le nombre d'éléments de la file.
        """
        return len(self.pile_entree) + len(self.pile_sortie)
```

Prolongements pour les files

Au-delà des implantations des files présentées dans ce document, il est aussi possible d'utiliser **une bibliothèque Python** permettant de reproduire le comportement d'une structure de **file**. Cette librairie est nommée **deque** et peut s'utiliser à la fois pour les **piles** et les **files**. Concernant les **files** un exemple d'utilisation peut être :

```
from queue import deque

f = deque() #Création d'une file vide.
f.append(element) #Ajout d'un élément au bout de la file.
f[0] #Tête, au début de la file
f.popleft() #On retire l'élément en tête, au début de la file.
len(f) #Taille de la file
len(f) == 0 #Vérifier si la file est vide ou non
```

Attention, les méthodes permettant d'utiliser cette file sont très semblables à celles d'une **liste** (Python).

Comparaison d'efficacité des implantations pour les files

Afin de comparer l'efficacité des différentes structures qui implantent les **files**, nous allons mesurer le temps d'exécution des deux opérations principales, **ajouter un élément au bout de la file** et **retirer la tête**.

Attention toutefois, le temps présenté ici **est seulement indicatif** pour permettre de **comparer** les structures. En effet, **le temps d'exécution ne sera pas le même selon l'ordinateur**, son état, etc. Néanmoins, peu importe la machine, les différences de temps observées, entre l'exécution de ces opérations sur les structures permet d'estimer si une structure est plus efficace qu'une autre.

Pour avoir un temps d'exécution assez représentatif, nous effectuons l'ajout en bout de file de **10 millions** d'éléments puis nous retirons un à un chaque élément jusqu'à vider la file.

Opération - Ajout d'éléments au bout de la file

Type de file	Temps d'exécution
File avec des listes Python	1,55 secondes
File avec deux piles (listes)	2,45 secondes
File objet	8,8 secondes
File objet avec deux piles (objets)	12,35 secondes
File avec bibliothèque deque	0,54 secondes

Opération - Retirer la tête de la file (jusqu'à vider la file)

Type de file	Temps d'exécution
File avec des listes Python	Plus de 15 minutes
File avec deux piles (listes)	14 secondes
File objet	3,5 secondes
File objet avec deux piles (objets)	14,88 secondes
File avec bibliothèque deque	0,54 secondes

On remarque que l'implantation objet avec deux piles **est la moins performante** pour l'ajout, et que celle avec les **listes** Python **est la moins performante** pour retirer le premier élément.

L'implantation utilisant la bibliothèque semble être **la plus performante**.

Pour l'ajout, de manière générale, les ordres de grandeur de complexité sont les mêmes pour chaque structure :

- pour l'implantation avec les **listes** Python, cela repose sur la fonction **append** qui est en temps constant 0(1) ;
- pour l'implantation avec les deux **piles** (en utilisant les **listes**), on appelle la fonction d'empilement sur une pile qui utilise **append**, qui est en temps constant 0(1) ;
- pour l'implantation objet, on n'effectue pas de parcours, on manipule seulement les attributs **head** et **end** et on crée de nouveaux objets de type **Cellule**, avec quelques opérations intermédiaires (ce qui explique la différence observée). On est pratiquement en temps constant 0(1) ;
- de même pour l'implantation objet avec deux piles, il s'agit simplement d'empiler un élément, nous avons vu que cette opération était en temps quasi-constant dans la section précédente 0(1) ;
- pour l'implantation avec **deque**, la fonction **append** est en temps constant 0(1).

Néanmoins, on observe des différences concernant les ordres de grandeur de la complexité de l'opération consistant à retirer la tête de la file :

- pour l'implantation avec les **listes** Python, cela repose sur la fonction **pop paramétrée avec 0**. Si l'appel de « pop » sans paramètre est en temps constant, l'appel avec pour paramètre **0** consiste à déplacer tous les éléments de la **listes** (chaque fois qu'on retire un élément). On a donc une complexité en O(n) ;
- pour l'implantation utilisant deux **piles** (avec des **listes**), tout dépend de l'état de la file (si elle doit être ré-organisée ou non). La complexité de cette opération est O(n) dans le pire des cas (quand il faut ré-organiser les piles), mais sinon, en temps constant O(1), quand il y a des éléments dans la file de sortie ;
- pour l'implantation objet, on n'effectue pas de parcours, on manipule seulement l'attribut **head** . On est pratiquement en temps constant O(1) ;
- même raisonnement que précédemment pour la version objet avec deux **piles**, la complexité n'est pas forcément en temps constant à cause de la problématique de ré-organisation. On observe d'ailleurs que les deux temps sont très proches ;
- pour l'implantation avec **deque**, la fonction **popleft** est en temps constant O(1).

La librairie **deque** reste **la plus efficace** (elle est en effet conçue pour la manipulation efficace de piles ou de files, contrairement aux **listes** Python de la première version), et il est recommandé de l'utiliser pour avoir un code performant. Néanmoins, l'implantation **objet** a pour avantage de montrer et d'assimiler le fonctionnement interne de cette structure. C'est donc un meilleur outil d'apprentissage des concepts, contrairement aux librairies qui agissent comme des « boîtes noires ».

Usage des files dans les parcours de graphes et arbres

Les **files** sont utiles dans de nombreux problèmes. Elles permettent notamment de réaliser le **parcours en largeur** d'un **arbre** ou bien d'un **graphe**, qui sont deux autres structures abordées lors de l'année de terminale. Le **parcours en largeur** consiste à explorer les nœuds d'un arbre (ou les sommets d'un graphe) dans un certain ordre, suivant la logique d'ajout et de retrait des éléments dans **une file**. Il permet notamment de donner les éléments d'un **arbre binaire de recherche** dans l'ordre croissant des valeurs, de calculer la distance de tous les nœuds depuis un nœud source sur un graphe.

Ce parcours est souvent utilisé lors des problèmes de cheminement, afin de trouver une solution avec le plus court chemin possible. D'ailleurs, l'algorithme de **Dijkstra** reprend la logique du parcours en largeur. Cet algorithme permet de trouver le plus court chemin entre deux sommets sur un graphe. On peut imaginer que, par exemple, sur un graphe modélisant un réseau de villes, il permettra de trouver le plus court trajet allant d'une ville à une autre.