Piles et files

Sommaire

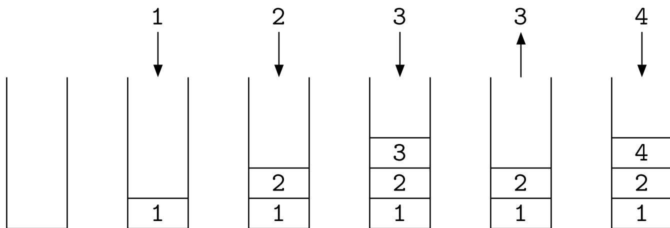
1. Présentation des piles et des files
2. Interfaces minimales des piles et des files
3. Implémentation d'une pile
4. Implémentation d'une file

Sommaire

1. Présentation des piles et des files
2. Interfaces minimales des piles et des files
3. Implémentation d'une pile
4. Implémentation d'une file

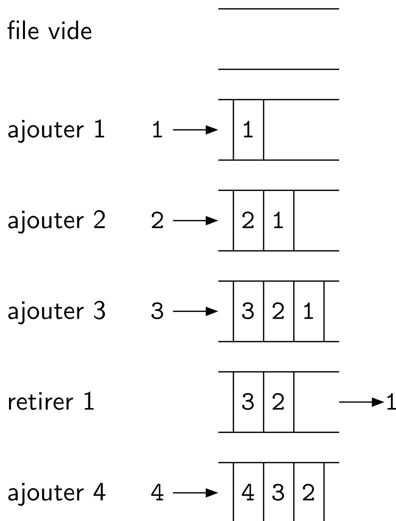
Présentation des piles et des files

- Les piles et les files sont des structures de données caractérisées par
  - une organisation *linéaire* (comme les tableaux et les listes) ;
  - une interface qui leur est propre [1]
- une *pile* (*stack* en anglais)
  - a la structure. . .d'une pile (par exemple d'assiettes) ;
  - est régie par le principe LIFO : Last In First Out (Dernier Entré Premier Sorti).



1. Ce qui milite en faveur d'une approche orientée objet, où les objets sont définis par la relation forte existant entre les données et les méthodes qui les manipulent.

- Une *file* (*queue* en anglais)
  - a la structure. . .d'une file (par exemple de personnes) ;
  - est régie par le principe FIFO : First In First Out (Premier Entré Premier Sorti).



- ces structures de données interviennent notamment (en vrac) :
  - dans l'architecture des microprocesseurs ;
  - dans la gestion des tâches d'impressions dans un réseau ;
  - dans l'exécution des programmes (cf. la récursivité et l'empilement mémoire des contextes des fonctions) ;
  - dans les annulations (Ctrl-Z) des manipulations faites dans un éditeur de texte ;
  - dans la gestion, par l'ordonnanceur d'un système d'exploitation, de l'exécution des processus ;
  - dans la modélisation des attentes à des guichets ;
  - dans la gestion par un navigateur de l'historique des pages visitées.

Ex. 1 Citer, dans les exemples précédents, ceux qui font intervenir une structure de pile ou de file.

Sommaire

1 Présentation des piles et des files

2 Interfaces minimales des piles et des files

3 Implémentation d'une pile

4 Implémentation d'une file

Interfaces minimales des piles et des files

- On définira une pile à l'aide d'une classe `Pile` (réalisant une pile). Si on reporte à un paragraphe ultérieur la question de l'implémentation, l'interface à réaliser doit contenir les fonctions suivantes [2] :

```
def creer_pile() -> Pile[T]:
    """crée une pile vide"""

def est_vide(p: Pile[T]) -> bool:
    """renvoie True si la pile est vide, False sinon"""

def empiler(p: Pile[T], e: T) -> None:
    """ajoute e au sommet de p"""

def depiler(p: Pile[T]) -> T:
    """retire et renvoie l'élément au sommet de p"""
```

2. Où on a utilisé les annotations de type vues précédemment.

- Ci-dessus, il faut comprendre que le type `T` est simplement un type (existant) quelconque[3] qui pourra être fixé lors de la création effective de piles. `Pile[T]` représente alors une pile d'éléments de type `T`.
- Les fonctions à réaliser pour l'interface de la classe `File` (réalisant une file) sont quand à elles :

```
def creer_file() -> File[T]:
    """crée une file vide"""

def est_vide(f: File[T]) -> bool:
    """renvoie True si la file f est vide, False sinon"""

def ajouter(f: File[T], e: T) -> None:
    """ajoute e au sommet de f"""

def retirer(f: File[T]) -> T:
    """retire et renvoie l'élément en tête de f"""
```

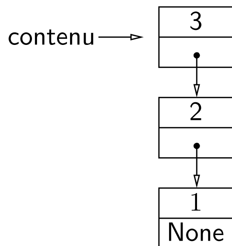
3. En réalité on verra dans un exercice qu'on est en mesure, avec un travail supplémentaire, de réaliser une classe `Pile` pouvant fonctionner avec plusieurs types (`int`, `str`,...) sans avoir à changer le code source. On parle alors de programmation *générique*.

Sommaire

1. Présentation des piles et des files
2. Interfaces minimales des piles et des files
3. Implémentation d'une pile
4. Implémentation d'une file

Implémentation d'une pile

- On choisit d'implémenter une pile à l'aide d'une liste chaînée et en utilisant la classe `Cellule` définie au chapitre précédent, comme illustré sur la figure ci-contre.
- On voit que le sommet de la pile sera la «tête» (première cellule) de la liste chaînée. L'attribut `contenu` de la pile sera simplement la première cellule de la liste, relié aux suivantes.
- Une pile vide sera représentée par une liste chaînée vide, c'est à dire par `None`.



- La réalisation des fonctions `__init__` et `est_vide` est assez facile.

  Détaillons la réalisation de `empiler`.
- Pour empiler un élément `e` sur `p` il faut
  1. créer une cellule dont la valeur est `e` ;
  2. affecter à l'attribut `suivante` de celle-ci l'attribut `contenu` de la pile ;
  3. modifier l'attribut `contenu` de la pile pour qu'il pointe sur la cellule qui vient d'être créée.
- Pour dépiler un élément de la pile il faut
  1. affecter à une variable `v` l'attribut `valeur` de la cellule `contenu` du sommet de la pile ;
  2. modifier le sommet de la pile pour qu'il devienne la deuxième cellule de la liste chaînée (c'est à dire `contenu.suivante`) ;
  3. renvoyer `v`.

  Le code source correspondant à cette implémentation est donné page suivante.

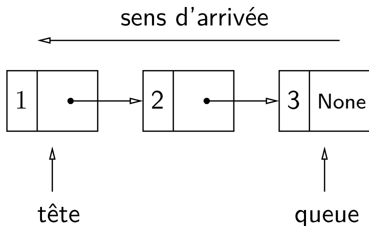
```python
class Cellule:
    """une cellule d'une liste chaînée"""
    def __init__(self, v ,s):
        self.valeur = v
        self.suivante = s

class Pile:
    """structure de pile"""
    def __init__(self):
        self.contenu = None

    def est_vide(self):
        return self.contenu is None

    def empiler(self, v):
        self.contenu = Cellule(v, self.contenu)

    def depiler(self):
        if self.est_vide():
            raise IndexError("dépiler sur une pile vide")
        v = self.contenu.valeur
        self.contenu = self.contenu.suivante
        return v
```

Sommaire

1. Présentation des piles et des files
2. Interfaces minimales des piles et des files
3. Implémentation d'une pile
4. Implémentation d'une file

Implémentation d'une file

- On choisit ici encore d'implémenter la structure de file à l'aide d'une liste chaînée.
- Il faut clairement définir la *tête* et la *queue* de la file. Conventionnellement, la tête de la file est la tête de la liste chaînée, la queue étant la dernière cellule de cette liste, comme illustré sur la figure ci-dessous.



- On aura besoin, pour ajouter un nouvel élément à la file, d'accéder à la dernière cellule de la liste.

Implémentation d'une file

- Plutôt que de parcourir à chaque fois la liste, il est préférable, d'ajouter dans la classe `File` un attribut `queue` permettant de mémoriser l'adresse de la dernière cellule. Le début de la file est alors appelé `tete`.
- On déclare, pour ce faire, la classe `File` et son constructeur comme suit

```
class File:
    """structure de file"""
    def __init__(self):
        self.tete = None
        self.queue = None
```

- La réalisation de la méthode `est_vide` est assez facile. Seule celles de `ajouter` et `retirer` demandent à être détaillées.
- Pour ajouter un élément `e` en queue de la file il faut
  1. créer une cellule qui contient `e` et dont la suivante est `None` ;
  2. «accrocher» cette cellule à la queue de la file (changer `queue.suivante`) ;
  3. mettre à jour la queue de la file, qui devient la cellule créée.
- Pour retirer un élément en tête de la file et le renvoyer il faut
  1. si la file est vide lever une exception ;
  2. sinon, lire la valeur de la tête de la file ;
  3. modifier la tête de la file, qui devient la cellule suivante de la tête. Si la tête est `None`, mettre à jour la queue (qui devient aussi `None`).

Voici le code source de l'implémentation de la classe `File` :

```python
class Cellule:
    """une cellule d'une liste chaînée"""
    def __init__(self,v ,s):
        self.valeur = v
        self.suivante = s

class File:
    """structure de file"""
    def __init__(self):
        self.tete = None
        self.queue = None

    def est_vide(self):
        return self.tete is None
```

```python
    def ajouter(self, e):
        c = Cellule(e, None)
        if self.est_vide():
            self.tete = c
        else:
            self.queue.suivante = c
        self.queue = c

    def retirer(self):
        if self.est_vide():
            raise IndexError("retirer \
            sur une file vide")
        v = self.tete.valeur
        self.tete = self.tete.suivante
        if self.tete is None:
            self.queue = None
        return v
```

Ex. 2 On considère un navigateur web dans lequel on s'intéresse à deux opérations : aller à une nouvelle page et revenir à la précédente. Le bouton de retour doit permettre de revenir aux pages précédemment visitées, et ce jusqu'à la première page.
On remarque, puisque la dernière page stockée dans les pages visitées est la première à sortir, que l'on a affaire à une structure de pile. Une variable `adresse_courante` mémorise la page courante et une pile `adresses_precedentes` mémorise les pages précédentes.
Deux fonctions sont utilisées :

- la première, `aller_a(adresse_cible)`, empile `adresse_courante` dans la pile `adresses_precedentes` (du moment que `adresse_courante` est bien définie) et affecte `adresse_courante` à `adresse_cible` ;
- la seconde, `retour()`, revient à la dernière page visitée, en affectant à `adresse_courante` le résultat du dépilement de la pile `adresses_precedentes` (du moment qu'elle n'est pas vide).

Le code Python de ces deux fonctions, dont on suppose disposer est donné page suivante :

```python
adresse_courante = None
adresses_precedentes = creer_pile()

def aller_a(adresse_cible):
    if adresse_courante is not None:
        adresses_precedentes.empiler(adresse_courante)
    adresse_courante = adresse_cible

def retour():
    if not adresses_precedentes.est_vide():
        adresse_courante = adresses_precedentes.depiler()
```

On souhaite compléter ce programme pour avoir également une fonction `retour_avant` dont le comportement est le suivant :

- chaque appel à la fonction `retour` place la page quittée au sommet d'une deuxième pile `adresses_suivantes` (du moment que la pile `adresses_precedentes` n'est pas vide) ;
- un appel à la fonction `retour_avant` enregistre la page courante dans la pile des adresses précédentes et transforme la page courante en l'adresse enregistrée au sommet de la pile `adresses_suivantes` (du moment que celle-ci n'est pas vide) ;
- toute nouvelle navigation avec `aller_a` annule les adresses suivantes (c'est à dire vide la pile `adresses_suivantes`).

On se propose, avec une approche P.O.O., de créer une classe `Adresses` regroupant trois attributs :

- l'adresse courante, `adresse_courante` ;
- la pile des adresses précédentes, `adresses_precedentes` ;
- la pile des adresses suivantes, `adresses_suivantes`.

Cette façon de procéder permet de ne pas avoir à déclarer des variables globales (mot clef `global` en Python). En effet, si on n'utilise pas ce mot clef au début du code la fonction `aller_a` (simplement par la ligne `global adresse_courante`), la variable globale `adresse_courante` (définie au niveau le plus haut du module) ne peut être modifiée par la fonction `aller_a`.

On considère que l'utilisation du mot clef `global` est une mauvaise pratique en programmation : cela contrevient au principe selon lequel toute fonction ne devrait pas avoir un code dont la validité dépend de son environnement extérieur. Une modification de cet environnement peut entraîner une perte de validité du code de la fonction et obliger à réécrire celui-ci.

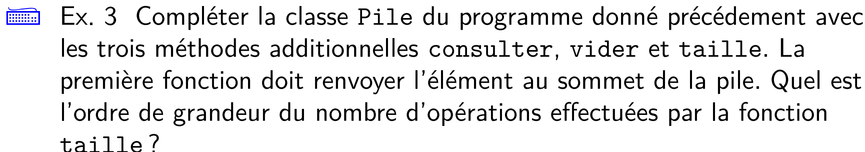
Par ailleurs, passer en argument `adresse_courante` n'est pas non plus une solution lorsque le type de `adresse_courante` est immuable (par exemple le type `int`) (cf. le chapitre sur la portée et le passage des arguments en Python).

La seule façon serait alors de faire que la valeur de retour de la fonction `aller_a` soit la nouvelle valeur de l'adresse courante, ce qui ne correspond pas au cahier des charges qu'on s'est fixé pour cette fonction.

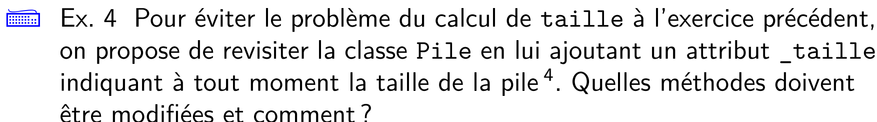
Modifier et compléter le code donné ci-dessus pour définir une classe `Adresses` (utilisant la classe `Pile`, qui elle-même utilise la classe `Cellule`), ayant les trois attributs déjà indiqués et ayant les méthodes :

- `aller_a(self, adresse_cible)`
- `retour(self`
- `retour_avant(self)`

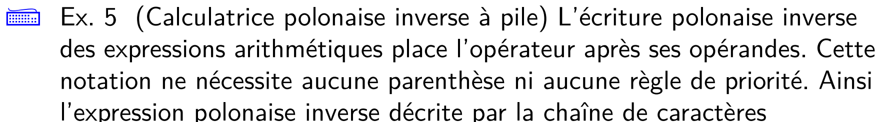
On pourra redéfinir, à la fois dans la classe `Pile` et dans la classe `Adresses` la méthode `__str__` de façon à pouvoir facilement afficher l'état courant des différentes adresses.



Ex. 3 Compléter la classe `Pile` du programme donné précédement avec les trois méthodes additionnelles `consulter`, `vider` et `taille`. La première fonction doit renvoyer l'élément au sommet de la pile. Quel est l'ordre de grandeur du nombre d'opérations effectuées par la fonction `taille` ?



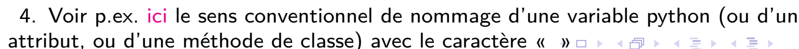
Ex. 4 Pour éviter le problème du calcul de `taille` à l'exercice précédent, on propose de revisiter la classe `Pile` en lui ajoutant un attribut `_taille` indiquant à tout moment la taille de la pile [4]. Quelles méthodes doivent être modifiées et comment ?



Ex. 5 (Calculatrice polonaise inverse à pile) L'écriture polonaise inverse des expressions arithmétiques place l'opérateur après ses opérandes. Cette notation ne nécessite aucune parenthèse ni aucune règle de priorité. Ainsi l'expression polonaise inverse décrite par la chaîne de caractères

```
'1 2 3 * + 4 *'
```

désigne l'expression traditionnellement notée $(1 + 2 \times 3) \times 4$.



4. Voir p.ex. ici le sens conventionnel de nommage d'une variable python (ou d'un attribut, ou d'une méthode de classe) avec le caractère « _ »

La valeur d'une telle expression peut être calculée facilement en utilisant une pile pour stocker les résultats intermédiaires. Pour cela, on observe un à un les éléments de l'expression et on effectue les actions suivantes :

- si on voit un nombre, on le place sur la pile ;
- si on voit un opérateur binaire, on récupère les deux nombres au sommet de la pile, on leur applique l'opérateur, et on replace le résultat sur la pile.

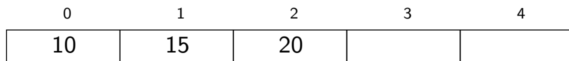
Écrire une fonction `calc_pol_inv(ch)` prenant en argument une chaîne de caractères `ch` représentant une expression en notation polonaise inverse composée d'additions et de multiplications de nombres entiers et renvoyant la valeur de cette expression. On supposera que les éléments de l'expression sont séparés par des espaces. Remarque : on supposera que l'expression `ch` est bien écrite.

Ex. 6 On s'intéresse à une fonction qui prend une file en paramètre, transfère son contenu dans une pile, puis transfère le contenu de la pile à nouveau vers la file. Quel est l'effet de cette fonction sur la file ?

Ex. 7 On se propose dans cet exercice d'implémenter une structure de file à l'aide d'un tableau de taille donnée. La capacité maximale de la file est fixée à sa création et en fait une file de capacité limitée, ou *bornée*. On représente la file par un tuple `(n, taille, tab)` où :

- `n` est la capacité de la file (son nombre maximum d'éléments) ;
- `taille` est son nombre d'éléments ;
- `tab` est le tableau, de taille `n`, contenant ses éléments.

La tête de la file est l'élément d'indice 0 de tab, la queue de la file est l'élément d'indice `taille-1`. La figure ci-dessous représente un exemple de file bornée de capacité 5 contenant 3 éléments.



On souhaite disposer de l'interface suivante pour manipuler une telle file :

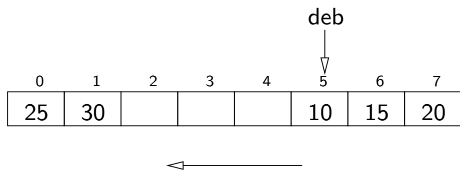
- `creer_file(n)` : renvoie une file bornée vide de capacité `n` (entier)
- `file_vide(file)` : renvoie `True` si la file `file` est vide, `False` sinon.
- `elements(file)` : renvoie une chaîne de caractère contenant, séparés par des blancs, les valeurs des éléments de la file `file` (si elle n'est pas vide), de la tête vers la queue, et la chaîne `'file vide'` sinon.
- `enfiler(file, e)` : ajoute e dans la file `file` et renvoie le tuple contenant la file modifiée.
- `defiler(file)` : retire un élément de la file `file` (du moment qu'elle n'est pas vide) et renvoie un tuple contenant, dans l'ordre, l'élément retiré et le tuple contenant la file modifiée.
- `tete_file(file)` : renvoie l'élément de la tête de la file `file` (du moment qu'elle n'est pas vide).

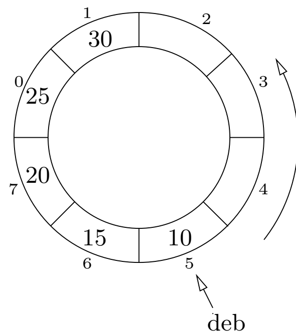
Réaliser l'implémentation en Python de cette structure de file bornée.
Remarques : on pourra gérer le cas d'une file vide à l'aide d'assertions ; on rappelle que les tuples en Python sont immutables, ce qui entraîne qu'il faut utiliser `enfiler(file, e)` à l'aide de l'affectation `f = empiler(f, e)` pour modifier `f`.

Ex. 8 On se propose dans cet exercice de réaliser une structure de file bornée à l'aide d'un tableau circulaire. Une telle file sera appelée *circulaire*. Comme dans l'exercice précédent, la capacité de la file est celle du tableau, fixée à la création de la file.
Une telle file est représentée par un tuple `n, deb, taille, tab` où :

- `n` est la capacité de la file ;
- `deb` est l'indice de la tête de la file ;
- `taille` est son nombre d'éléments ;
- `tab` est le tableau contenant ses éléments.

Pour éviter le coût du décalage des éléments dans le tableau lors du défilement, les éléments de la file sont placés entre l'indice `deb` et l'indice `deb+taille-1` modulo `n`. La figure page suivante montre une file circulaire de capacité 8 pour laquelle `taille` vaut 5 et `deb` vaut 5.


deb
0
1
2
3
4
5
6
7
25
30
10
15
20


1
2
30
0
25
3
20
7
4
15
10
6
5
deb

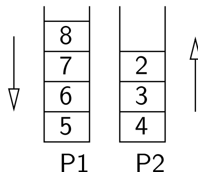
Implémenter en Python l'interface suivante pour cette structure de file circulaire :

- `creer_file(n)` : renvoie une file circulaire vide de capacité `n` (entier)
- `file_vide(file)` : renvoie `True` si la file `file` est vide, `False` sinon.
- `elements(file)` : renvoie une chaîne de caractère contenant, séparés par des blancs, les valeurs des éléments de la file `file` (si elle n'est pas vide) et la chaîne `'file vide'` sinon.
- `enfiler(file, e)` : ajoute `e` dans la file `file` et renvoie le tuple contenant la file modifiée.
- `defiler(file)` : retire un élément de la file `file` (du moment qu'elle n'est pas vide) et renvoie un tuple contenant, dans l'ordre, l'élément retiré et le tuple contenant la file modifiée.
- `tete_file(file)` : renvoie l'élément de la tête de la file `file` (du moment qu'elle n'est pas vide).

Rappel : en Python, le reste de la division euclidienne de `a` par `n` (c'est à dire la valeur de `a` modulo `n`) est l'expression `a % n`.

Ex. 9

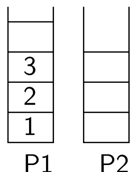
On se propose dans cet exercice d'implémenter une file à l'aide de deux piles. Pour cela on utilise une pile P1 pour enfiler des éléments et une pile P2 pour en défiler, comme représenté sur la figure ci-contre.



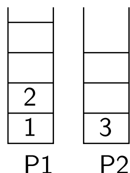
Le fonctionnement de la file est le suivant :

- Enfiler un nouvel élément consiste à l'empiler dans P1 (qui peut être initialement vide) ;
- défiler un élément consiste :
  - si P2 n'est pas vide, à dépiler P2 et récupérer l'élément voulu ;
  - sinon, si P1 n'est pas vide, à réaliser un transfert du contenu de P1 vers P2. On remarque qu'il suffit, pour conserver l'ordre FIFO des éléments, de dépiler tour à tour les éléments de P1 et de les empiler sur P2 ;
  - sinon, Si P1 est vide, on cherche à défiler sur une file vide, ce qui est impossible.

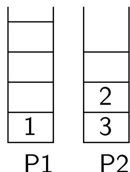
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.



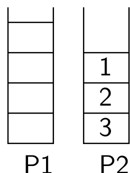
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.



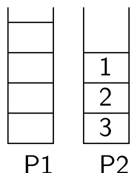
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

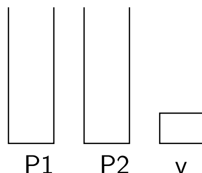


On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

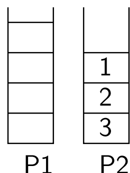


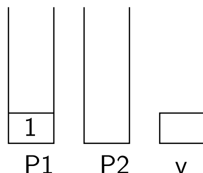
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.



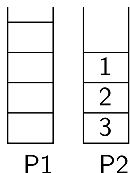


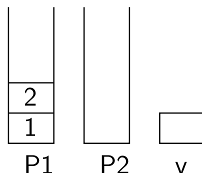
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.





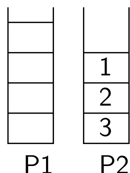
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

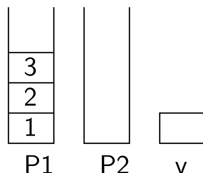




f.enfiler(2)

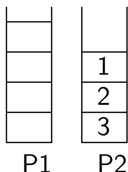
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

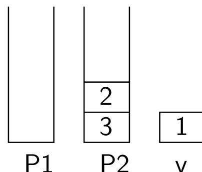




f.enfiler(3)

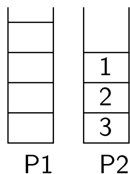
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

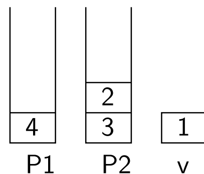




v=f.defiler()

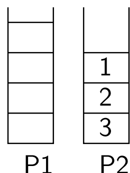
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

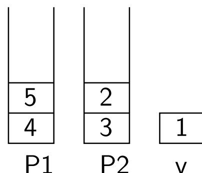




f.enfiler(4)

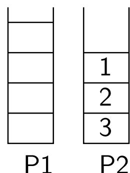
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

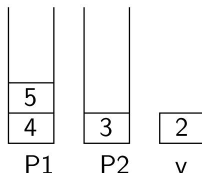




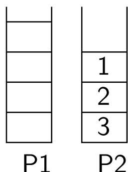
f.enfiler(5)

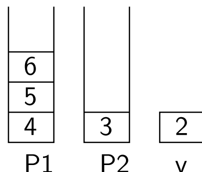
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.





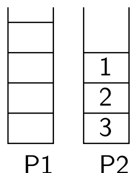
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.

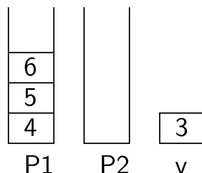




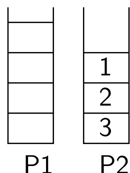
f.enfiler(6)

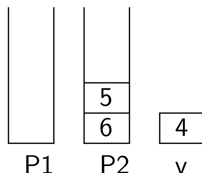
On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.





On montre ci-dessous à gauche, sur un exemple, comment est réalisé le transfert (en vue du défilement) de P1 vers P2 lorsque P2 est vide. On montre également, à droite, comment évoluent les piles lors d'une succession d'opérations sur la file.





En utilisant le code de la classe `Pile`, elle même construite à partir de la classe `Cellule`, et en respectant les idées qui viennent d'être présentées, compléter l'implémentation en Python d'une classe `File` utilisant deux piles et ayant l'interface suivante [5] :

```
class File:
    """structure de file construite à partir de 2 piles"""
    def __init__(self):
        """crée une file vide"""
        self.tete = Pile()
        self.queue = Pile()

    def est_vide(self):
        """renvoie True si la file est vide, False sinon"""
        # code à compléter

    def enfiler(self, e):
        """enfile e dans la file"""
        # code à compléter

    def defiler(self):
        """défile un élément de la file si elle n'est pas vide et le renvoie"""
        # code à compléter
```

5. double-cliquer pour télécharger le code à compléter