C5 Listes et Piles

Programme Terminale

Contenus	Capacités attendues	Commentaires
Structures de données, interface et implémentation	Spécifier une structure de données par son interface. Distinguer interface et implémentation. Ecire plusieurs implémentations d'une même structure de données.	L'abstraction des structures de données est introduite aprés plusieurs implémentations d'une structure simple comme la file (avec un tableau ou avec deuc piles).
Listes, piles, files : structures linéaires. Dictionnaires, index et clé.	Distinguer des structures par le jeu des méthodes qui les caractèrisent. Choisir une structure adaptée à la situation à modéliser. Distinguer la recherche d'une valeur dans une liste et dans un dictionnaire.	On distingue les modes FIFO (First In First Out) et LIFO (Last In First Out) des piles et des files.

1. Préambule : interface ≠ implémentation

⭐
⭐
⭐

A savoir

Les structures que nous allons voir peuvent s'envisager sous deux aspects :

- le côté utilisateur, qui utilisera une **interface** pour manipuler les données.
  - **interface** : Vue "logique" de la structure de données. Elle spécifie la nature des données ainsi que l'ensemble des opérations permises sur la structure.
- le côté concepteur, qui aura choisi une **implémentation** pour construire la structure de données.
  - **Implémentation** : Vue "physique" de la structure de données. Il s'agit de la programmation effective des opérations définies dans l'interface, en utilisant des types de données déja existants.

Nous avons déjà abordé ces deux aspects lors de la découverte de la Programmation Orientée Objet. Le principe d'encapsulation fait que l'utilisateur n'a qu'à connaître l'existence des méthodes disponibles, et non pas le contenu technique de celle-ci. Cela permet notamment de modifier le contenu technique (l'implémentation) sans que les habitudes de l'utilisateur (l'interface) ne soient changées.

2. Structures de données linéaires

2.1. À chaque donnée sa structure

En informatique comme dans la vie courante, il est conseillé d'adapter sa manière de stocker et de traiter des données en fonction de la nature de celles-ci.

En informatique, pour chaque type de données, pour chaque utilisation prévue, une structure particulière de données se revèlera (peut-être) plus adaptée qu'une autre.

2.1.1. Données linéaires

Intéressons nous par exemple aux **données linéaires**. Ce sont des données qui ne comportent pas de *hiérarchie* : toutes les données sont de la même nature et ont le même rôle. Par exemple, un relevé mensuel de températures, la liste des élèves d'une classe, un historique d'opérations bancaires...

Ces données sont «plates», n'ont pas de sous-domaines : la structure de **liste** paraît parfaitement adaptée.

Lorsque les données de cette liste sont en fait des couples (comme dans le cas d'une liste de noms/numéros de téléphone), alors la structure la plus adaptée est sans doute celle du **dictionnaire**.

Les listes et les dictionnaires sont donc des exemples de structures de **données linéaires**.

2.1.2. Données non-linéaires

Même si ce n'est pas l'objet de ce cours, donnons des exemples de structures adaptées aux données non-linéaires :

Si une liste de courses est subdivisée en "rayon frais / bricolage / papeterie" et que le rayon frais est lui-même séparé en "laitages / viandes / fruits & légumes", alors une structure d'**arbre** sera plus adaptée pour la représenter. Les structures arborescentes seront vues plus tard en Terminale.

Enfin, si nos données à étudier sont les relations sur les réseaux sociaux des élèves d'une classe, alors la structure de **graphe** s'imposera d'elle-même. Cette structure sera elle-aussi étudiée plus tard cette année.

2.2. Comment seront traitées ces données linéaires ? Introduction des listes, des piles et des files

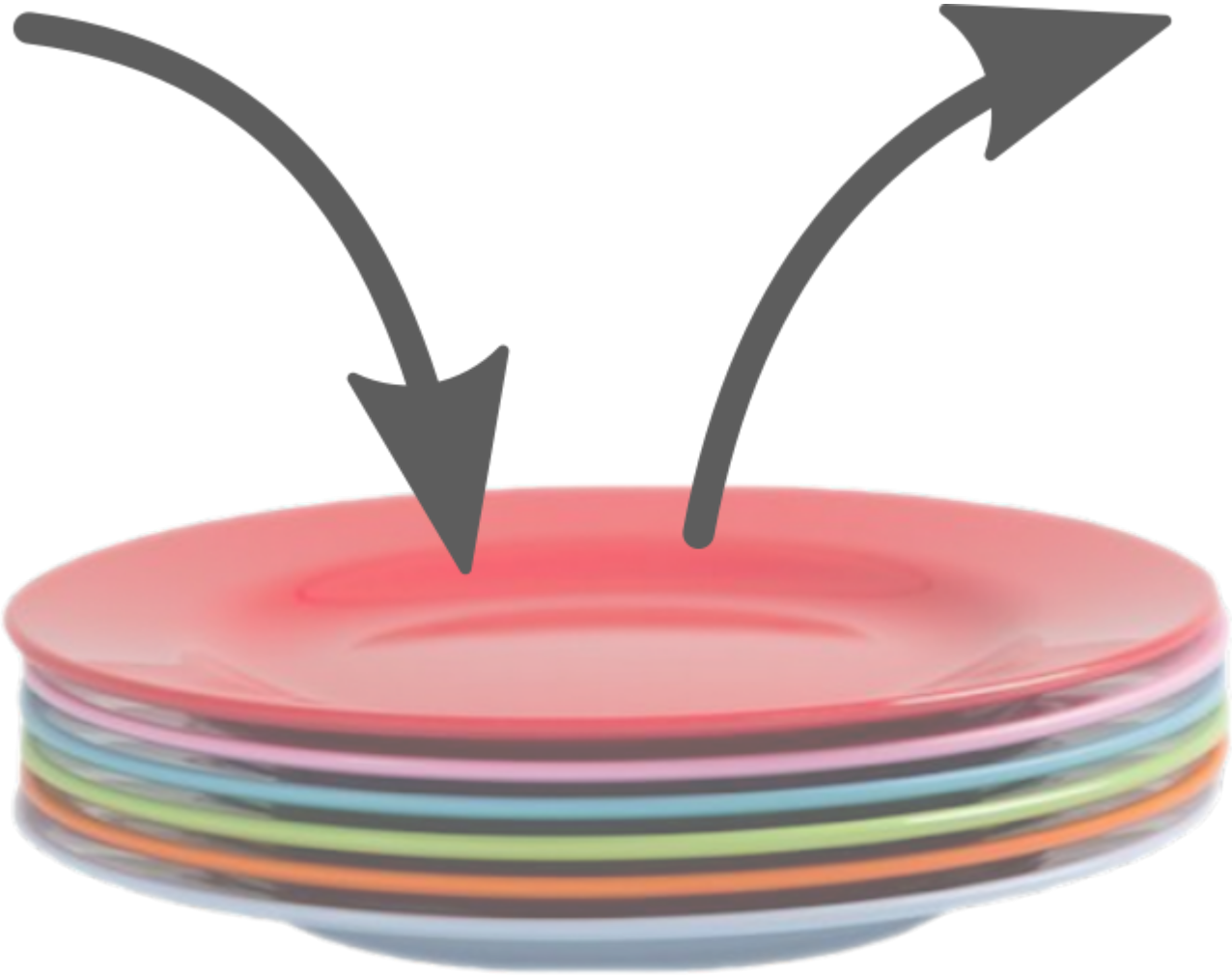
La nature des données ne fait pas tout. Il faut aussi s'intéresser à la manière dont on voudra les traiter :

- À quelle position les faire entrer dans notre structure ?

- À quel moment devront-elles en éventuellement en sortir ?
- Veut-on pouvoir accéder rapidement à n'importe quel élément de la structure, ou simplement au premier ? ou au dernier ?

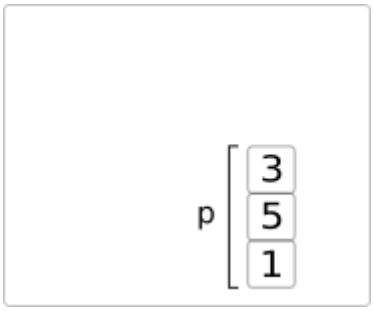
Lorsque ces problématiques d'entrée/sortie n'interviennent pas, la structure «classique» de liste est adaptée. Mais lorsque celle-ci est importante, il convient de différencier la structure de **pile** de celle de **file**.

2.2.1. Les piles (*stack*)



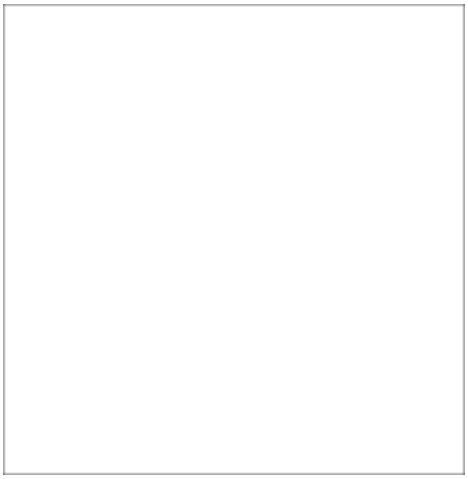
Une structure de **pile** (penser à une pile d'assiette) est associée à la méthode **LIFO** (Last In, First Out) :

- les éléments sont empilés les uns au-dessus des autres,
- et on ne peut toujours dépiler que l'élément du haut de la pile.
- Le dernier élément à être arrivé est donc le premier à être sorti.

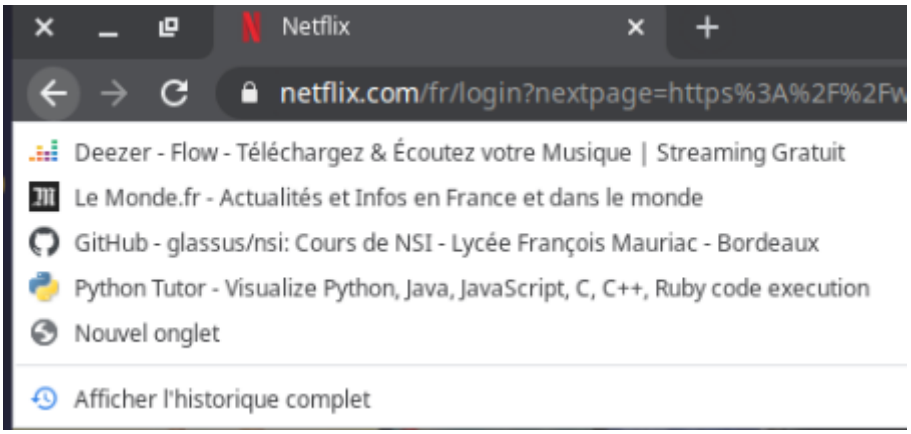


Exemples de données stockées sous forme de pile

- lors de l'exécution d'une fonction récursive, le processeur empile successivement les appels à traiter : seule l'instruction du haut de la pile peut être traitée.



- historiques de navigation sur le Web, la liste des pages parcourues est stockée sous forme de pile : la fonction «Back» permet de «dépiler» peu à peu les pages précédemment parcourues :



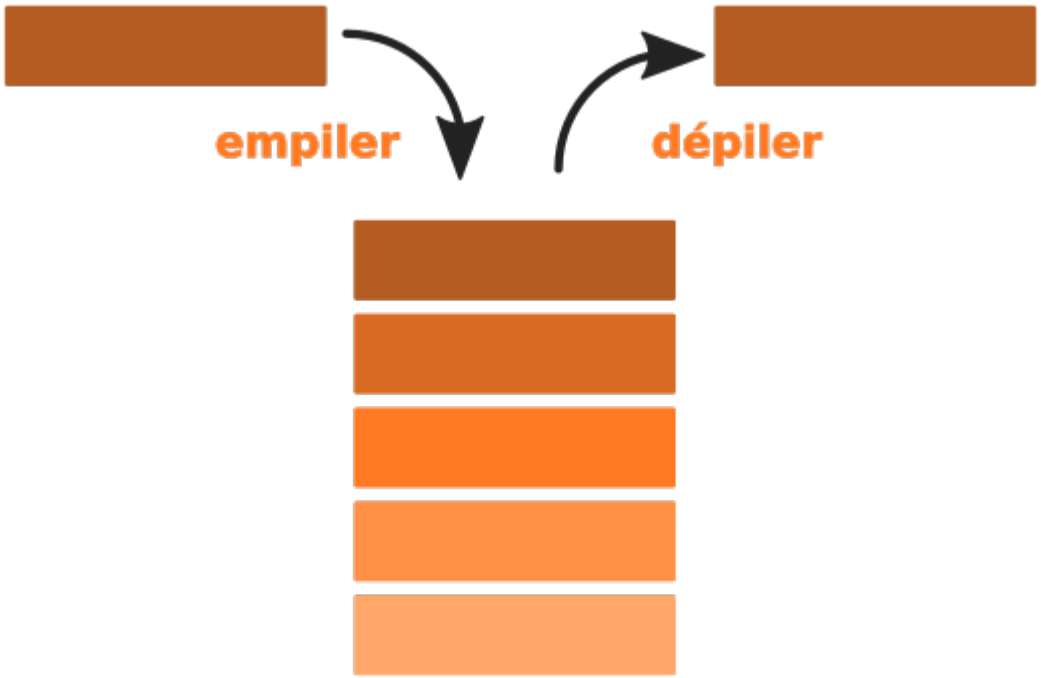
- historiques d'annulation d'instructions (Ctrl+Z)

Interface

On dispose (ou souhaite disposer) sur une pile des méthodes/primitives suivantes:

- déterminer si la pile est vide (`est_vide` , `is_empty`)
- **empiler** un nouvel élément au sommet de la pile (`empiler` , `push`)
- **dépiler** l'élément du sommet de la pile (`depiler` , `pop`) et le renvoyer

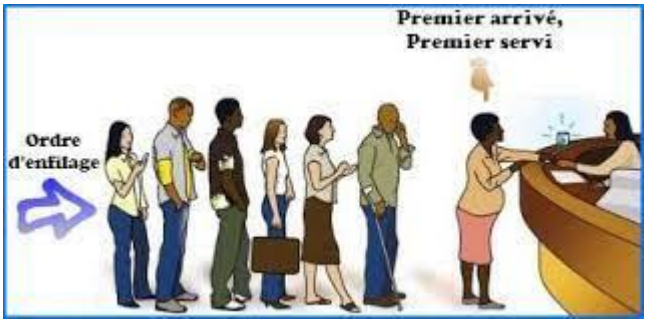
Ces opérations doivent être réalisées **en temps constant**, soit en \(O(1)\).

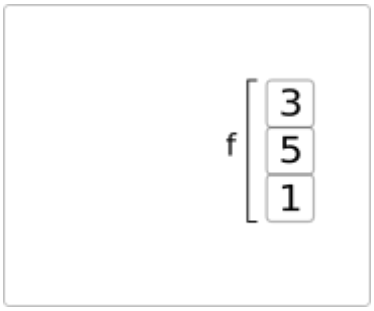


2.3. Les files

A connaitre

Une **file** (queue) est une structure de données *linéaire* contenant des éléments généralement *homogènes* fondée sur le principe «premier arrivé, premier sorti» (en anglais **FIFO** : Fast In, First Out).





Exemples de situations utilisant une file:

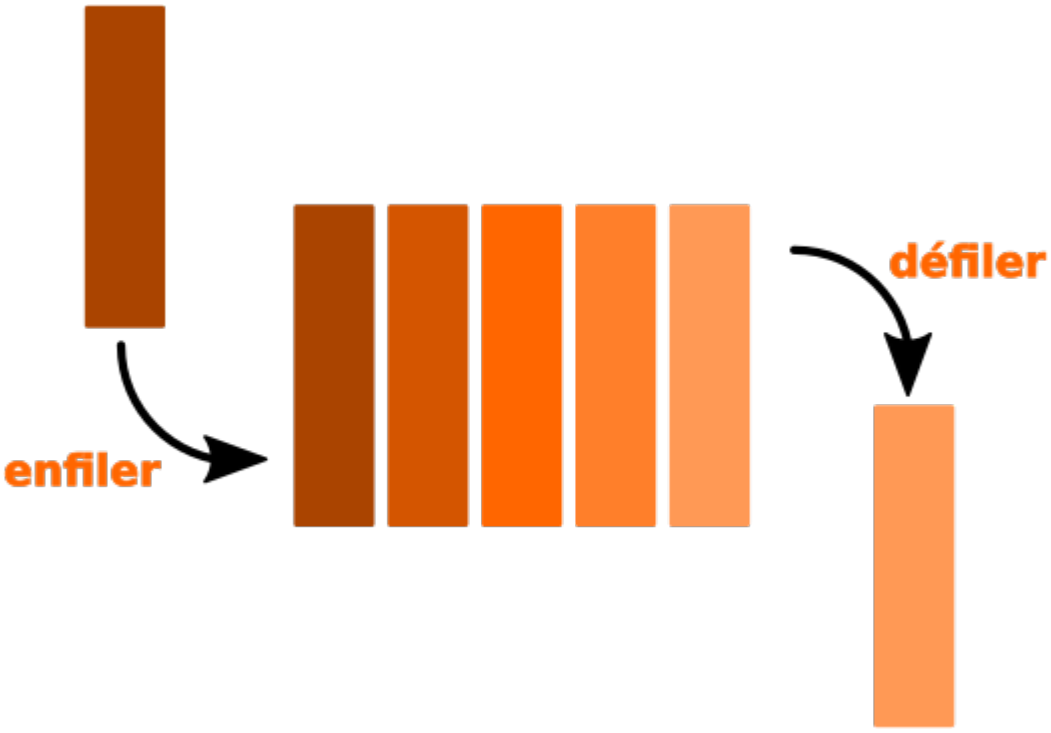
- file d'attente : documents soumis à impression, élèves à la cantine...
- gestion des processus (plus tard...)
- parcours en largeur d'un arbre/graphe (plus tard...)

Interface :

On dispose (ou souhaite disposer) sur une file des méthodes/primitives suivantes:

- déterminer si la file est vide (`is_empty`)
- **enfiler** (ajouter) un nouvel élément dans la file (`enqueue`)
- **défiler** l'élément de tête de la file (`dequeue`) et le renvoyer

Ces opérations doivent être réalisées **en temps constant**, soit en $O(1)$.

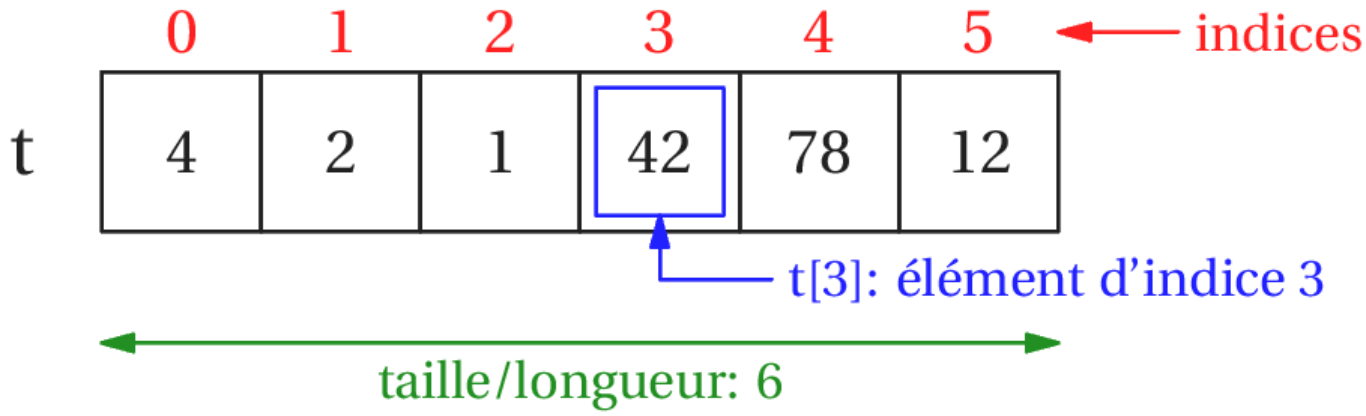


3. Listes chaînées

3.1. Retour sur les tableaux

Dans une structure de **tableau** (**array** en anglais), les données (ou une référence vers les données) sont organisées de manière séquentielle en mémoire, où chaque élément (ou référence) est de même type. On peut donc calculer la position de l'élément (ou de la référence) en mémoire en fonction de son numéro d'ordre dans la séquence.

En règle générale, la taille du tableau est connue à la déclaration. Dans ce cas, on ne peut pas ajouter d'élément au delà de la dernière case prévue (ce qui n'est pas le cas en python).



Quelques propriétés des tableaux :

- création d'un tableau de taille donnée
- accès à un élément à partir de son indice en temps constant (\(O(1)\))

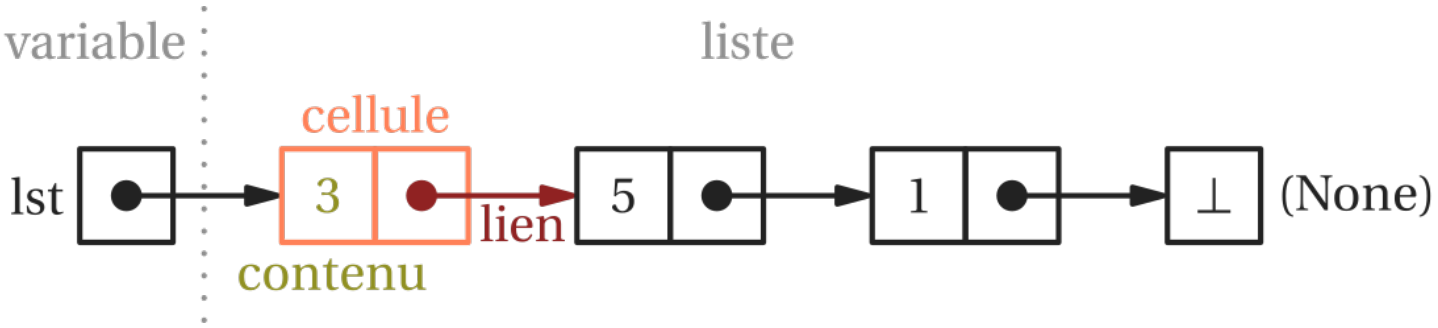
- modification d’un élément à partir de son indice en temps constant (\(O(1)\))

En revanche, l'insertion d'un élément dans le tableau impose de décaler tous les élements d'un indice, elle se fait donc en temps linéaire, soit \(O(n)\)...

3.2. Liste chaînée

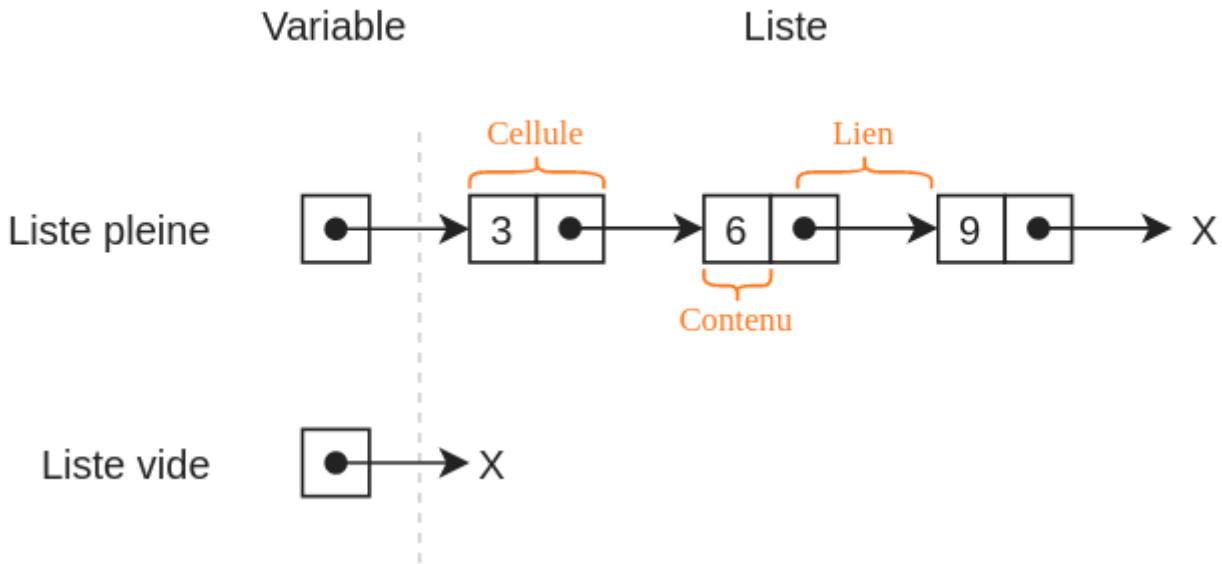
Avec une structure de **liste (chaînée)**, on représente à nouveau une séquence d'éléments (à nouveau le plus souvent homogènes), mais les données ne sont pas nécessairement séquentielles en mémoire. On dispose en revanche d'un moyen permettant de passer d'un élément au suivant, d'où le terme *chaîné*.

Chaque élément est donc stocké dans un bloc mémoire **avec** une deuxième information: l'adresse de l'élément suivant. On parle de **maillon ou cellule** (ou encore **node**) pour désigner ces blocs.



On peut généralement ajouter de nouveaux éléments pour augmenter la taille de la structure dynamiquement.

Et finalement, une liste chaînée est soit vide, soit n'est qu'un lien vers une cellule, qui contient une valeur et un lien vers une cellule, qui est soit vide, soit n'est qu'un lien vers une cellule, qui contient une valeur et un lien vers une cellule, qui...
On parle donc de ***définition récursive*** d'une liste chaînée.

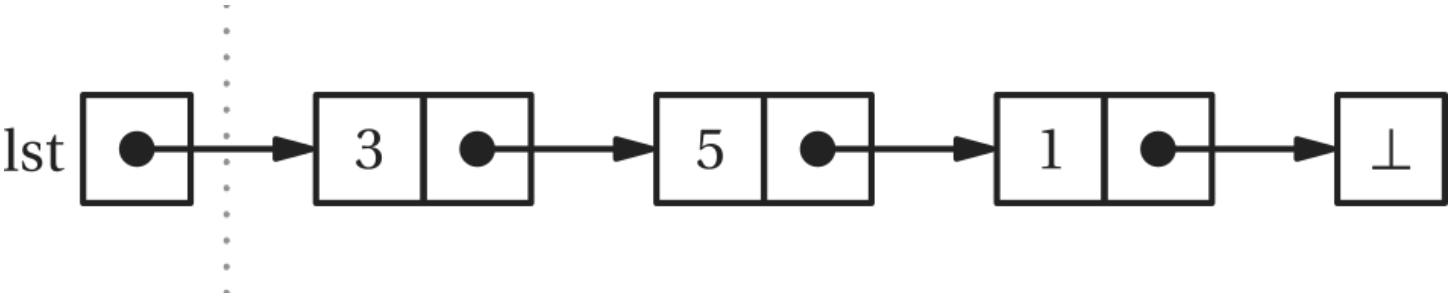


Interface :

On dispose (ou souhaite disposer) sur une liste chaînée des méthodes/primitives suivantes:

- construire une liste vide, souvent nommée `nil`
- déterminer si la liste est vide (`est_vide` , `is_empty`)
- insérer un élément en tête de liste (`insert`)
- récupérer l'élément en tête de liste (`tete` , `head`)
- récupérer la liste privée de son premier élément, appelée la **queue** (`queue` , `tail`)

Ces opérations doivent être réalisées **en temps constant**, soit en \(O(1)\).



Remarque : Accès à un élément

Pour accéder à un élément quelconque, il faut parcourir toute la liste jusqu'à trouver l'élément: le temps d'accès est linéaire, c'est-à-dire proportionnel à la taille de la liste (en \(O(n)\)) et donc non constant.

3.2.1. Implémentation choisie :

- Une liste est caractérisée par un ensemble de cellules.
- Le lien (on dira souvent le «pointeur») de la variable est un lien vers la première cellule, qui renverra elle-même sur la deuxième, etc.
- Chaque cellule contient donc une valeur et un lien vers la cellule suivante.
- Une liste peut être vide (la liste vide est notée `x` ou bien `None` sur les schémas)

Une conséquence de cette implémentation sous forme de liste chaînée est la non-constance du temps d'accès à un élément de liste : pour accéder au 3ème élément, il faut obligatoirement passer par les deux précédents.

3.2.2. Exemple d'implémentation minimale d'une liste chaînée

Exemple : implémentation d'une liste chainée **en POO** 🧡

🐍 **Script Python**

```python
class Cellule :
    def __init__(self, contenu, suivante):
        self.contenu = contenu
        self.suivante = suivante
```

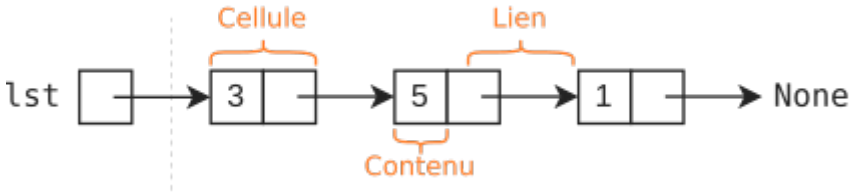
Cette implémentation rudimentaire permet bien la création d'une liste :

🐍 **Script Python**

```python
lst = Cellule(3, Cellule(5, Cellule(1,None)))
```

La liste créée est donc :

Mais plus précisément, on a :



Exercice

Retrouvez comment accéder aux éléments 3, 5 et 1.

On pourra remarquer que l'interface proposée à l'utilisateur n'est pas des plus pratiques...

3.2.3. Un exemple d'interface pour les listes

Imaginons que nous possédons une interface offrant les fonctionnalités suivantes :

- `ListeC()` : crée une liste vide.
- `est_vide()` : indique si la liste est vide.
- `ajoute_tete()` : insère un élément en tête de liste.
- `renvoie_tete()` : renvoie la valeur de l'élément en tête de liste ET le supprime de la liste.

Exercice

On considère l'enchaînement d'opérations ci-dessous. Écrire à chaque étape l'état de la liste `lst` et la valeur éventuellement renvoyée.

Script Python

```
lst = ListeC()
lst.ajoute_tete(3)
lst.ajoute_tete(5)
lst.ajoute_tete(1)
lst.renvoie_tete()
lst.est_vide()
lst.ajoute_tete(2)
lst.renvoie_tete()
lst.renvoie_tete()
lst.renvoie_tete()
lst.est_vide()
```

Implémentation

On ne définit la méthode spéciale `__str__` uniquement pour vérifier et afficher de façon pratique la liste.

Script Python

```
class Cellule:
    def __init__(self, contenu=None, suivante=None):
        self.contenu = contenu
        self.suivante = suivante

    def __str__(self):
        if self.suivante == None :
            return f"{self.contenu}"
        else :
            return f"{self.contenu} -> {str(self.suivante)}"

class ListeC:
    def __init__(self):
        self.tete = None # Liste vide

    def ajoute_tete(self,valeur):
        nouvelle_cellule=Cellule(valeur,self.tete)
        self.tete=nouvelle_cellule

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

    def renvoie_tete(self):
        t=self.tete.contenu
        self.tete=self.queue()
        return t

    def queue(self):
```

```
        return self.tete.suivante




    def __str__(self):
        c=self
        l = []
        while not c.est_vide():
            l.append(str(c.renvoie_tete()))
        l.reverse()
        for v in l:
            self.ajoute_tete(v)
        l.reverse()
        return ' -> '.join(l)
```

Vérification de la question 2

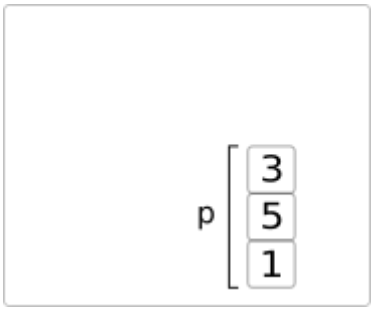
🐍 **Script Python**

```
lst = ListeC()
print(lst)
lst.ajoute_tete(3)
print(lst)
lst.ajoute_tete(5)
print(lst)
lst.ajoute_tete(1)
print(lst)
print("valeur renvoyé : ",lst.renvoie_tete() )

print("Est vide :",lst.est_vide())

lst.ajoute_tete(2)

print(lst)
print("valeur renvoyé : ",lst.renvoie_tete() )
print(lst)
print("valeur renvoyé : ",lst.renvoie_tete() )
print(lst)
print("valeur renvoyé : ",lst.renvoie_tete() )
print(lst)
print("Est vide : ",lst.est_vide())
```

📋 **Texte**

```
3
5 -> 3
1 -> 5 -> 3
valeur renvoyé :  1
Est vide : False
2 -> 5 -> 3
valeur renvoyé :  2
5 -> 3
valeur renvoyé :  5
3
valeur renvoyé :  3

Est vide :  True
```

4. Les Piles



Comme expliqué précédemment, une pile travaille en mode LIFO (Last In First Out). Pour être utilisée, l'interface d'une pile doit permettre a minima :

- la création d'une pile vide
- l'ajout d'un élément dans la pile (qui sera forcément au dessus). On dira qu'on **empile**.
- le retrait d'un élément de la pile (qui sera forcément celui du dessus) et le renvoi de sa valeur. On dira qu'on **dépile**.

4.1. Utilisation d'une interface de pile

Exercice

On considère l'enchaînement d'opérations ci-dessous.
Écrire à chaque étape l'état de la pile `p` et la valeur éventuellement renvoyée.

Bien comprendre que la classe `Pile()` et ses méthodes n'existent pas vraiment. Nous *jouons* avec son interface.

🐍 **Script Python**

```
p = Pile()  #
p.empile(3)   # p=
p.empile(5)  # p=
p.est_vide()  #
p.empile(1)  # p=
p.depile()  # p=    valeur renvoyée :
p.depile()  # p=      valeur renvoyée :
p.empile(9)  # p=
p.depile()  # p=       valeur renvoyée :
p.depile()  # p=      valeur renvoyée :
p.est_vide() #
```

4.2. Implémentation(s) d'une pile

Interface :

L'objectif est de créer une classe `Pile`. L'instruction `Pile()` créera une pile vide. Chaque objet `Pile` disposera des méthodes suivantes :

- `est_vide()` : indique si la pile est vide.
- `empile()` : insère un élément en haut de la pile.
- `depile()` : renvoie la valeur de l'élément en haut de la pile ET le supprime de la pile.
- `__str__()` : permet d'afficher la pile sous forme agréable (par ex : `|3|6|2|5|`) par `print()`

4.2.1. À l'aide du type `list` de Python

Exercice

Créer la classe ci-dessus. Le type `list` de Python est parfaitement adapté. Des renseignement intéressants à son sujet peuvent être trouvés ici.

Dans les deux exercices qui suivent, quelle que soit l'implémentation de la pile, le code suivant :

🐍 **Script Python**

```
p = Pile()
p.empiler(1)
p.empiler(2)
print(p.depiler())
p.empiler(3)
while not p.est_vide():
    print(p.depiler())
```

doit afficher :

🐍 **Script Python**

```
2
3
1
```

🐍 **Script Python**

```
class Pile:
    def __init__(self):
        self.data = ...
```

```
    def est_vide(self):
        pass


    def empile(self,x):
        pass

    def depile(self):
        if self.est_vide() == True :
            raise IndexError("Vous avez essayé de dépiler une pile vide !")
        else :
            pass

    def __str__(self):       # Hors-Programme : pour afficher
        s = "|"              # convenablement la pile avec print(p)
        for k in self.data :
            s = s + str(k) + "|"
        return s

    def __repr__(self):      # Hors-Programme : pour afficher
        s = "|"              # convenablement la pile avec p
        for k in self.data :
            s = s + str(k) + "|"
        return s
```

📋 **Texte**

#A tester

Correction et commentaire(s)

🐍 **Script Python**

```
class Pile:
    def __init__(self):
        self.data = []

    def est_vide(self):
        return len(self.data) == 0

    def empile(self,x):
        self.data.append(x)

    def depile(self):
        if self.est_vide() == True :
            raise IndexError('Vous avez essayé de dépiler une pile vide !')
        else :
            return self.data.pop()

    def __str__(self):       # Hors-Programme : pour afficher
        s = '|'              # convenablement la pile avec print(p)
        for k in self.data :
            s = s + str(k) + '|'
        return s

    def __repr__(self):      # Hors-Programme : pour afficher
        s = '|'              # convenablement la pile avec p
        for k in self.data :
            s = s + str(k) + '|'
        return s
```

A connaître (possibilité d'exercice à compléter lors de l'épreuve pratique)

4.2.2. À l'aide d'une liste chaînée et de la classe `Cellule` créée au 2.3

Au **III.** nous avons créé la classe `Cellule` :

🐍 **Script Python**

```
class Cellule :

    def __init__(self, contenu, suivante):
        self.contenu = contenu
        self.suivante = suivante
```

Exercice

A l'aide cette classe, re-créer une classe `Pile` disposant exactement de la même interface que dans l'exercice précédent.

```python class Pile: def **init**(self): pass

📋 **Texte**

```
def est_vide(self):
    pass

def empile(self, x):
    self.data = Cellule(...,...)

def depile(self):
    v = ... #on récupère la valeur à renvoyer
    self.data = ... # on supprime la 1ère cellule
    return ...

def __str__(self): # Hors-Programme : pour afficher
    s = "|"
    c = self.data
    while c != None :
        s += str(c.contenu)+"|"
        c = c.suivante
    return s
```

```

📋 **Texte**

```
#A tester
```

Correction et commentaire(s)

Script Python

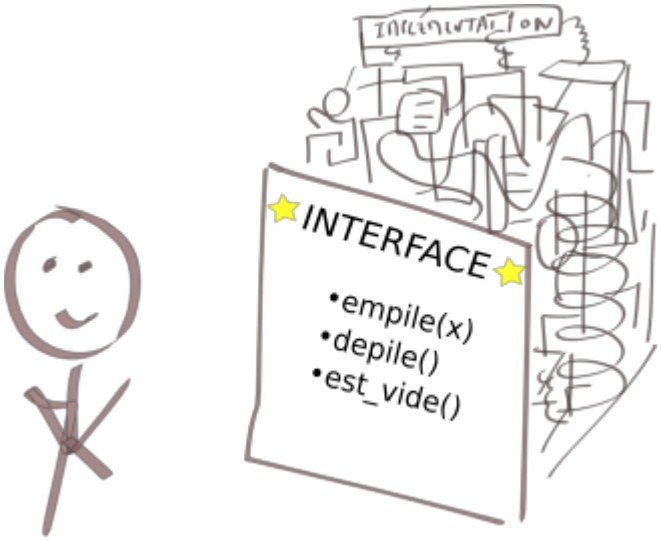
```
class Pile:
    def __init__(self):
        self.data = None

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

    def empile(self, x):
        self.data = Cellule(x,self.data)

    def depile(self):
        v = self.data.contenu #on récupère la valeur à renvoyer
        self.data = self.data.suivante  # on supprime la 1ère cellule
        return v

    def __str__(self):
        s = '|'
        c = self.data
        while c != None :
            s += str(c.contenu) + '|'
            c = c.suivante
        return s
```

Commentaires

A connaître (possibilité d'exercice à compléter lors de l'épreuve pratique)

A retenir :

Pour l'utilisateur, les interfaces du 3.2.1 et 3.2.2 sont strictement identiques. Il ne peut pas savoir, en les utilisant, l'implémentation qui est derrière.



4.3. Application des piles

À l'aide de deux variables `adresses` et `adresse_courante`, et de la classe `Pile` créée plus haut, simulez une gestion de l'historique de navigation internet.
Seules deux fonctions `go_to(nouvelle_adresse)` et `back()` sont à créer.

Exercice

Compléter le code suivant

🐍 **Script Python**

```
adresses = Pile()
adresse_courante = ""

def go_to(nouvelle_adresse) :
    global adresse_courante
    adresses.empile(...)
    adresse_courante = ...

def back():
    global adresse_courante
    ...
    ...
```

Exemple d'utilisation :

🐍 **Script Python**

```
go_to("google.fr")
go_to("lemonde.fr")
go_to("blabla.fr")
```

🐍 **Script Python**

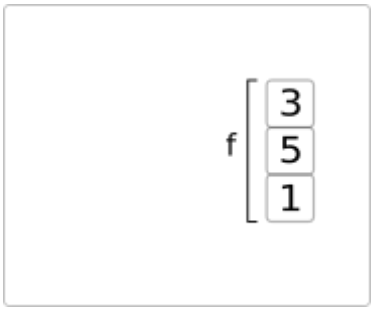
```
back()
adresse_courante
```

Correction et commentaires

🐍 **Script Python**

```
adresses = Pile()
adresse_courante = ""

def go_to(nouvelle_adresse) :
    global adresse_courante
    adresses.empile(nouvelle_adresse)
    adresse_courante = nouvelle_adresse

def back():
    global adresse_courante
    adresses.depile()
    adresse_courante = adresses.depile()
```

5. Les Files



Comme expliqué précédemment, une file travaille en mode FIFO (First In First Out). Pour être utilisée, une file doit permettre a minima :

- la création d'une file vide
- l'ajout d'un élément dans la file (qui sera forcément **au dessous**). On dira qu'on **enfile**.
- le retrait d'un élément de la file (qui sera forcément celui du **dessus**) et le renvoi de sa valeur. On dira qu'on **défile**.

5.1. Utilisation d'une interface de file

Exercice

On considère l'enchaînement d'opérations ci-dessous. Écrire à chaque étape l'état de la file `f` et la valeur éventuellement renvoyée.

🐍 **Script Python**

```
f = File()
f.enfile(3) # f =
f.enfile(5) # f  =
f.est_vide() #
f.enfile(1) # f  =
f.defile() # val renvoyée :  , f =
f.defile() # val renvoyée :  , f =
f.enfile(9) # f =
f.defile() # val renvoyée :  , f =
f.defile()# val renvoyée : , f =
f.est_vide() # True
```

5.2. Implémentation d'une file

Interface :

L'objectif est de créer une classe `File`, disposant des méthodes suivantes :

- `File()` : crée une file vide.
- `est_vide()` : indique si la file est vide.
- `enfile()` : insère un élément en bas de la file.
- `defile()` : renvoie la valeur de l'élément en haut de la file ET le supprime de la file.
- `__str__()` : permet d'afficher la file sous forme agréable (par ex : `|3|6|2|5|`) par `print()`

Test de l'implémentation

Dans les trois exercices qui suivent, quelle que soit l'implémentation de la file, le code suivant:

🐍 **Script Python**

```
f = File()
f.enfiler(1)
f.enfiler(2)
print(f.defiler())
f.enfiler(3)
```

```
while not f.est_vide():
    print(f.defiler())
```

doit afficher :

🐍 **Script Python**

```
1
2
3
```

Exercice

Créer la classe ci-dessus. Là encore, le type «list» de Python est peut être utilisé, voir ici. Néanmoins quelques remarques seront à apporter.

🐍 **Script Python**

```
class File:
    def __init__(self):
        pass

    def est_vide(self):
        pass


    def enfile(self,x):
        pass

    def defile(self):
        if self.est_vide() == True :
            raise IndexError("Vous avez essayé de défiler une file vide !")
        else :


    def __str__(self):       # Hors-Programme : pour afficher
        s = "|"              # convenablement la file avec print(p)
        for k in self.data :
            s = s + str(k) + "|"
        return s
```

Correction et commentaire(s)

🐍 **Script Python**

```
class File:
    def __init__(self):
        self.data = []

    def est_vide(self):
        return len(self.data) == 0

    def enfile(self,x):
        self.data.append(x)

    def defile(self):
        if self.est_vide() == True :
            raise IndexError("Vous avez essayé de défiler une file vide !")
        else :
            return self.data.pop(0)

    def __str__(self):      # Hors-Programme : pour afficher
        s = "|"             # convenablement la file avec print(p)
        for k in self.data :
            s = s + str(k) + "|"
        return s
```

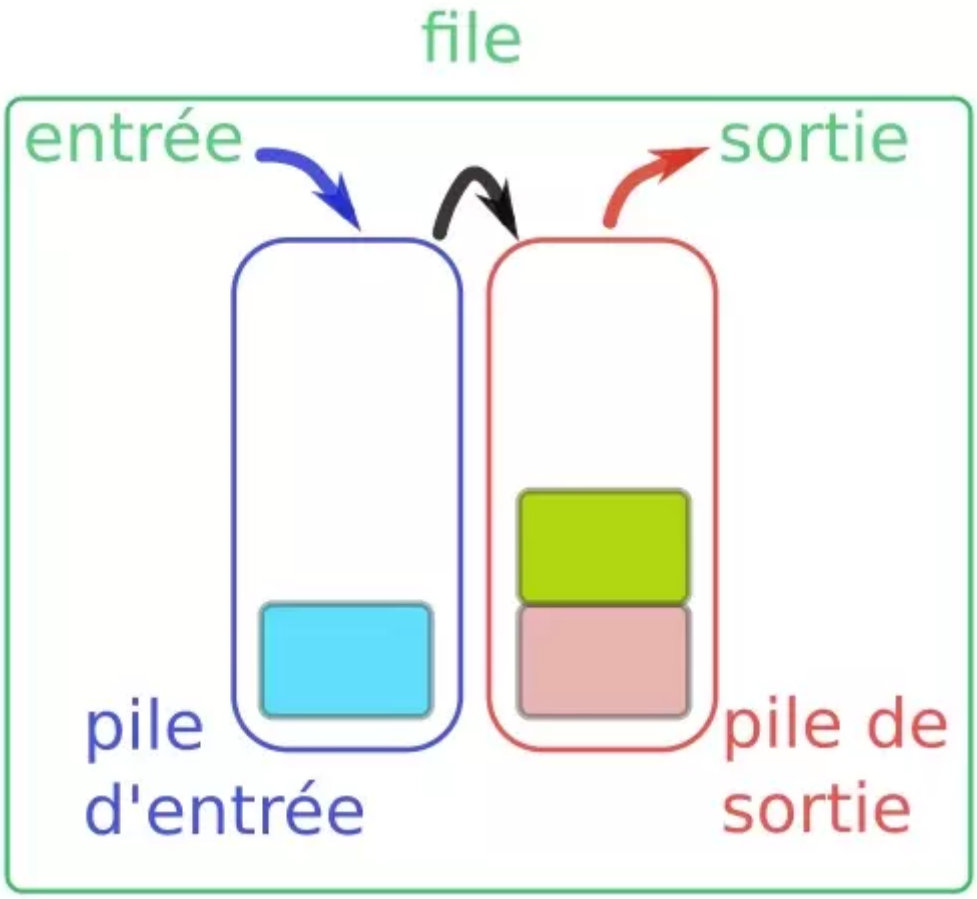
Remarque :
Notre implémentation répond parfaitement à l'interface qui était demandée. Mais si le «cahier des charges» obligeait à ce que les opérations `enfile()` et `defile()` aient lieu en temps constant (en $O(1)$), notre implémentation ne conviendrait pas.

En cause : notre méthode `defile()` agit en temps linéaire ($O(n)$) et non pas en temps constant. L'utilisation de la structure de «liste» de Python (les *tableaux dynamiques*) provoque, lors de l'instruction `self.data.pop(0)` un redimensionnement de la liste, qui voit disparaître son premier élément. Chaque élément doit être recopié dans la case qui précède, avant de supprimer la dernière case. Ceci nous coûte un temps linéaire.

5.3. Implémentation d'une file avec deux piles

Comment créer une file avec 2 piles ?
L'idée est la suivante : on crée une pile d'entrée et une pile de sortie.

- quand on veut enfiler, on empile sur la pile d'entrée.
- quand on veut défiler, on dépile sur la pile de sortie.
- si celle-ci est vide, on dépile entièrement la pile d'entrée dans la pile de sortie.



🐍 **Script Python**

```
# il est impératif de comprendre qu'on peut choisir l'implémentation
# de la classe Pile qu'on préfère parmi les deux traitées plus haut.
# Comme elles ont la MÊME INTERFACE et qu'on ne va se servir que
# de cette interface, leur mécanisme interne n'a aucune influence
# sur le code de la classe File que nous ferons ensuite.

# au hasard, on choisit celle avec la liste chaînée :

class Pile:
    def __init__(self):
        self.data = None

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

    def empile(self, x):
        self.data = Cellule(x,self.data)

    def depile(self):
        v = self.data.contenu #on récupère la valeur à renvoyer
        self.data = self.data.suivante  # on supprime la 1ère cellule
        return v

    def __str__(self):
        s = "|"
        c = self.data
        while c != None :
            s += str(c.contenu)+"|"
```

```
            c = c.suivante
        return s


# il ne faut pas oublier de remettre la classe Cellule qui intervient
# dans notre classe Pile :

class Cellule :

    def __init__(self, contenu, suivante):
        self.contenu = contenu
        self.suivante = suivante
```

🐍 **Script Python**

```
class File:
    def __init__(self):
        self.entree = Pile()
        self.sortie = Pile()

    def est_vide(self):
        return self.entree.est_vide() and self.sortie.est_vide()

    def enfile(self,x):
        self.entree.empile(x)

    def defile(self):
        if self.est_vide():
            raise IndexError("File vide !")

        if self.sortie.est_vide() == True :
            while self.entree.est_vide() == False :
                self.sortie.empile(self.entree.depile())

        return self.sortie.depile()
```

🐍 **Script Python**

```
f = File()
f.enfile(5)
f.enfile(8)

f.defile()
```

5

6. Sujet Epreuve Pratique

Sujet 13 :

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

🐍 **Script Python**

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

Compléter la classe File suivante où l'attribut dernier_file contient le maillon correspondant à l'élément arrivé en dernier dans la file :

🐍 **Script Python**

```
class File:

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

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

def affiche(self):
    maillon = self.tete
    print('<-- sens de la file <--')
    while maillon is not None:
        print(maillon.valeur, end=' ')
        maillon = maillon.suivant
    print()

def enfile(self, element):
    nouveau_maillon = ...
    if self.est_vide():
        self.tete = ...
    else:
        self.queue.suivant = ...
    self... = nouveau_maillon

def defile(self):
    if ...:
        resultat = ...
        self.tete = ...
        return ...
```