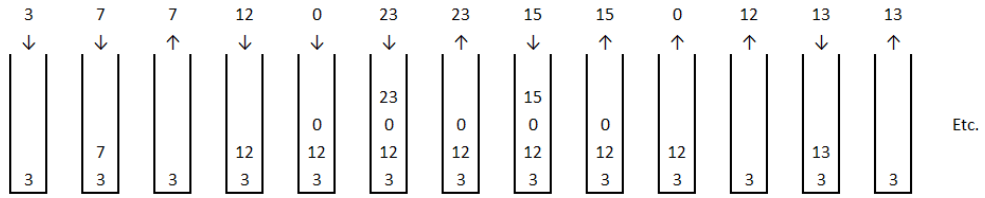
Piles & Files

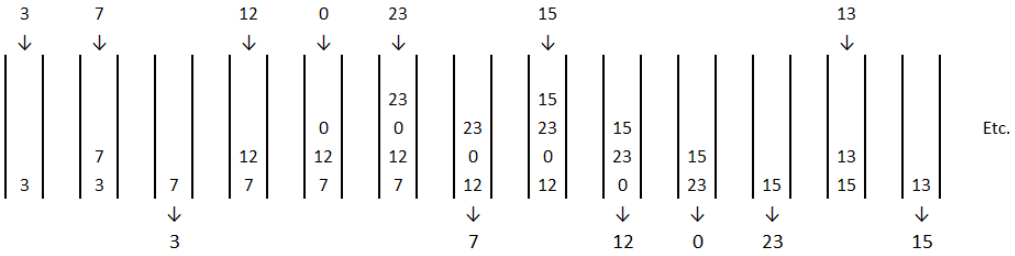
Introduction

La notion de *pile* correspond à celle d'une pile d'assiettes : le dernier élément stocké est le premier élément qui sera déstocké.



Une telle structure de données peut être utile pour :
- gérer les appels de fonctions récursives (rappelez-vous de la pile d'appels) ,
- travailler sur les expressions algébriques (vérifications des parenthèses, évaluation),
- gérer un historique de navigation ou de saisie clavier,
- effectuer certains parcours de graphes ou d'arbres,
- plus généralement, les algorithmes nécessitant de mémoriser des éléments en *mode LIFO (Last In First Out)*.

La notion de *file* correspond à celle d'une file d'attente : le premier élément stocké est le premier élément qui sera déstocké.



Une telle structure de données peut être utile pour :
- gérer l'ordonnancement des processus par un système d'exploitation,
- gérer les buffers utiles aux transmissions de données,
- gérer le spool des travaux d'impression sur une imprimante,
- effectuer certains parcours de graphes ou d'arbres,
- plus généralement, les algorithmes nécessitant de mémoriser des éléments en *mode FIFO (First In First Out)*.

Ces deux structures de données (qui viennent se rajouter aux tableaux, aux dictionnaires, aux matrices et aux tables rencontrées en classe de première) sont très simples puisqu'on peut seulement y ajouter/écrire un élément ou en retirer/lire un élément (on ne peut pas lire un élément qui est encore dans une pile ou dans une file : il faut d'abord l'en retirer).

I : Structure de pile

I . 1 : Implémentation avec un tableau et des fonctions de manipulation

On se base sur un tableau Python (de type `list`) et les méthodes `append` et `pop` des tableaux Python.

```python
def pilevide():
    return []

def empiler(element, pile):
    pile.append(element)

def depiler(pile):
    assert pile != []
    return pile.pop()

def est_vide(pile):
    return pile == []
```

Implémentation

```python
ma_pile = pilevide()          # ma_pile = []
empiler(7, ma_pile)           # ma_pile = [7]
empiler(8, ma_pile)           # ma_pile = [7, 8]
a = depiler(ma_pile)          # ma_pile = [7]        et a = 8
empiler(0, ma_pile)           # ma_pile = [7, 0]
a = depiler(ma_pile)          # ma_pile = [7]        et a = 0
est_vide(ma_pile)             # False
```

Exemple d'utilisation

I . 2 : Implémentation avec une classe `Pile` basée sur un tableau

On fait l'analogue du I.1 en programmation objet.

On utilise donc une classe `Pile` disposant d'un unique attribut `_p` qui est un tableau Python.

On implémente alors les méthodes `empiler`, `depiler` et `est_vide` à l'intérieur de cette classe.

La méthode `pilevide` du I.1 est remplacée par le constructeur `__init__` qui initialise l'attribut `p`.

Enfin on implémente la méthode `__repr__` qui permet d'obtenir une représentation textuelle de la pile.

```python
class Pile:
    '''Classe implémentant une structure de pile'''

    def __init__(self):
        self.p = []

    def empiler(self, element):
        self.p.append(element)

    def depiler(self):
        assert not self.est_vide()
        return self.p.pop()

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

    def __repr__(self):
        return str(self.p)
```

Implémentation

Ici la différence avec le I.1 pour l'utilisateur de la pile est d'abord syntaxique : il utilisera la notation pointée à la place d'une notation fonctionnelle.

```python
ma_pile = Pile()              # ma_pile.p = []
ma_pile.empiler(7)            # ma_pile.p = [7]
ma_pile.empiler(8)            # ma_pile.p = [7, 8]
a = ma_pile.depiler()         # ma_pile.p = [7]        et a = 8
ma_pile.empiler(0)            # ma_pile.p = [7, 0]
a = ma_pile.depiler()         # ma_pile.p = [7]        et a = 0
ma_pile.est_vide()            # False
print(ma_pile)                # affiche `[7]`
```

Exemple d'utilisation

La programmation objet permet aussi de masquer les détails internes d'implémentation (on parle d'*encapsulation*). Nous avons ainsi vu en TD que l'utilisateur de la classe `Pile` n'a pas besoin de savoir que les données sont stockées dans un attribut `p` qui est un tableau. Il serait d'ailleurs de bon goût de remplacer `p` par `_p` pour indiquer que `p` est privé à la classe.

La programmation objet permettrait enfin de profiter de l'*héritage* : on pourrait facilement créer une classe `Superpile` qui hérite de toutes les fonctionnalités de `Pile` en lui rajoutant quelques fonctionnalités spécifiques dont on a besoin à ce moment-là (hors-programme en Terminale NSI).

I . 3 : Interface minimale d'une Pile

L'*implémentation* d'une structure de données désigne la façon dont elle est programmée. L'*interface* quant à elle désigne uniquement ce dont l'utilisateur aura besoin pour pouvoir utiliser cette structure de données. Si on observe les deux exemples d'utilisation donnés ci-dessus, on voit que l'interface minimale d'une structure de pile est très simple :

Interface minimale d'une pile avec l'approche fonctions de manipulation

`pilevide() --> Pile`	: renvoie une pile vide.
`est_vide(p: Pile) --> bool`	: renvoie `True` si `p` est vide et `False` sinon.
`empiler(e: obj, p: Pile) --> None`	: ajoute l'élément `e` au sommet de la pile `p`.
`depiler(p: Pile) --> obj`	: retire et renvoie l'élément au sommet de la pile `p`.

Interface minimale d'une pile avec l'approche programmation objet

`Pile() --> Pile`	: renvoie une instance vide de la classe `Pile`.
`p.est_vide() --> bool`	: renvoie `True` si `p` est vide et `False` sinon.
`p.empiler(e: obj) --> None`	: ajoute l'élément `e` au sommet de la pile `p`.
`p.depiler() --> obj`	: retire et renvoie l'élément au sommet de la pile `p`

Remarque : selon les cas, les mots utilisés dans l'interface peuvent différer (ou pourrait avoir les verbes `ajouter` ou `retirer` au lieu de `empiler` et `depiler`). Ce qui importe c'est que ces quatre fonctionnalités sont *traditionnellement* celles d'une pile. Parfois on trouve des interfaces plus complètes, par exemple une méthode permettant de renvoyer l'élément présent au sommet de la pile sans le retirer.

II : Structure de file

II . 1 : Implémentation avec un tableau et des fonctions de manipulation

On se base sur un tableau Python (de type `list`) et les méthodes `append(elt)` et `pop(0)` des tableaux Python.
On aurait aussi pu se baser sur les méthodes `insert(0, elt)` et `pop()`.

```python
def filevide():
    return []

def enfiler(element, file):
    file.append(element)

def defiler(file):
    assert file != []
    return file.pop(0)

def est_vide(file):
    return file == []
```

Implémentation

```python
ma_file = filevide()          # ma_file = []
enfiler(7, ma_file)           # ma_file = [7]
enfiler(8, ma_file)           # ma_file = [7, 8]
a = defiler(ma_file)          # ma_file = [8]        et a = 7
enfiler(0, ma_file)           # ma_file = [8, 0]
a = defiler(ma_file)          # ma_file = [0]        et a = 8
est_vide(ma_file)             # False
```

Exemple d'utilisation

Cette implémentation présente un défaut dans le cas où la file serait amenée à comporter beaucoup d'éléments et à être sollicitée très souvent. En effet les tableaux Python, qui servent ici à stocker les données de la file, ont une complexité en O(n) pour l'insertion ou la suppression d'éléments ***en début*** de tableau.

Pour rappel, cela est lié au fait que lorsqu'on insère un élément en début de tableau, il faut décaler tous les éléments du tableau "d'une case mémoire vers la droite". Et si on supprime un élément en début de tableau, il faut décaler tous les éléments "d'une case mémoire vers la gauche". Par conséquent, si un tableau comporte 1 million d'éléments, une opération aussi simple que `pop(0)` ou `insert(0, elt)` requiert 1 million de décalages, ce qui n'est pas très performant.

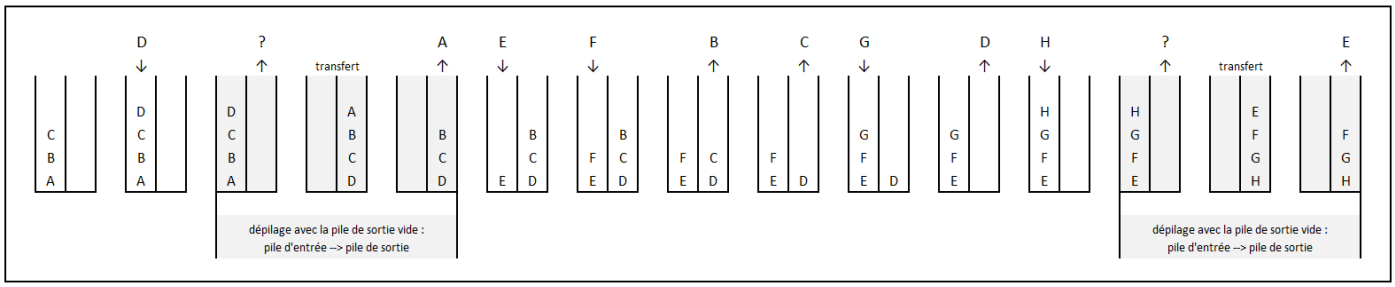
II . 2 : avec une classe File basée sur deux piles

On souhaite implémenter une classe File. Pour éviter le problème mentionné ci-dessus (lenteur) on ne souhaite pas utiliser de tableau. Une façon de procéder est d'utiliser deux piles pour implémenter une file. C'est un mécanisme que vous connaissez sans doute. En effet on peut faire l'analogie avec les jeux de carte pour lesquels tous les joueurs partagent :

- une pile de défausse (où l'on empile les cartes qui ont été jouées, face vers le dessus),
- une pile de pioche (où les joueurs piochent les cartes, face vers le bas, lorsque c'est leur tour de jouer).

Pour un tel jeu, lorsqu'un joueur doit piocher une carte alors que la pioche est vide il suffit de prendre la pile de défausse, de la retourner et cela devient la pile de pioche.

Voici un exemple en images : la pile de gauche est la défausse (ce sera l'attribut `_entree`) et la pile de droite est la pioche (ce sera l'attribut `_sortie`). L'ensemble des deux piles constitue la file que l'on souhaite implémenter.



On constate sans peine, pour ceux qui n'en seraient pas convaincus, que les éléments sont bien défilés dans le même ordre qu'ils ont été enfilés (FIFO).

```python
from module_pile import Pile

class File:
    '''Classe implémentant une structure de file grâce à deux piles'''

    def __init__(self):
        self._entree = Pile()
        self._sortie = Pile()

    def _transfert(self):
        while not self._entree.est_vide():
            self._sortie.empiler(self._entree.depiler())

    def enfiler(self, element):
        self._entree.empiler(element)

    def defiler(self):
        assert not self.est_vide()
        if self._sortie.est_vide():
            self._transfert()
        return self._sortie.depiler()

    def est_vide(self):
        return (self._entree.est_vide() and self._sortie.est_vide())
```

Implémentation

```python
ma_file = File()
ma_file.enfiler(7)
ma_file.enfiler(8)
a = ma_file.defiler()
ma_file.enfiler(0)
a = ma_file.defiler()
ma_file.est_vide()
```

Exemple d'utilisation

II . 3 : Interface minimale d'une File

Pour les files, comme pour les piles, il convient à nouveau d'opérer un distinguo entre l'interface et l'implémentation. Ici l'implémentation se base dans un cas sur un tableau alors que dans l'autre cas elle se base sur deux piles : pourtant cela ne se voit pas du tout dans l'interface. ***Autrement dit l'interface masque les détails de l'implémentation.***

Interface minimale d'une file avec l'approche fonctions de manipulation

`filevide() --> File`	: renvoie une file vide.
`est_vide(f: File) --> bool`	: renvoie `True` si `f` est vide et `False` sinon.
`enfiler(e: obj, f: File) --> None`	: ajoute l'élément `e` à l'arrière de la file `f`.
`defiler(f: File) --> obj`	: retire et renvoie l'élément situé à l'avant de la file `f`.

Interface minimale d'une pile avec l'approche programmation objet

`File() --> File`	: renvoie une instance vide de la classe `File`.
`f.est_vide() --> bool`	: renvoie `True` si `f` est vide et `False` sinon.
`f.enfiler(e: obj) --> None`	: ajoute l'élément `e` à l'arrière de la file `f`.
`f.defiler() --> obj`	: retire et renvoie l'élément situé à l'avant de la file `f`.

III . Mise à disposition dans module & Conclusion

Ce travail d'implémentation d'une file et d'une pile est mis à disposition dans un module `xile.py` qui implémente la version en programmation objet avec une variante néanmoins. Les verbes utilisés pour `empiler/enfiler` ou `dépiler/défiler` des éléments sont `ajouter` et `extraire`.
On retiendra aussi que Python propose une structure de données implémentant des piles et des files : il s'agit du type `deque`.

Les *piles* sont des structures de données linéaires qui travaillent en mode *LIFO (Last In First Out)*.
Les *files* sont des structures de données linéaires qui travaillent en mode *FIFO (First In First Out).*

Pour les piles, l'interface *traditionnelle* minimale est souvent réduite à quatre méthodes : créer une pile vide, ajouter un élément, extraire un élément (en mode LIFO) et tester si la pile est vide.
C'est la même chose pour les files en mode FIFO.

Pour interface donnée, plusieurs *implémentations* sont envisageables, plus ou moins performantes selon les choix effectués. Néanmoins l'utilisateur n'a pas à connaître l'implémentation choisie : il ne doit utiliser que ce qui est spécifié par l'interface. Ce respect de l'interface permet de modifier facilement l'implémentation d'un élément d'un projet : tant que l'interface de cet élément n'est pas modifiée, les autres utilisateurs ne sont pas impactés. L'interface a donc un rôle de cloisonnement qui facilite la conception et la maintenance des programmes.