> Version pdf

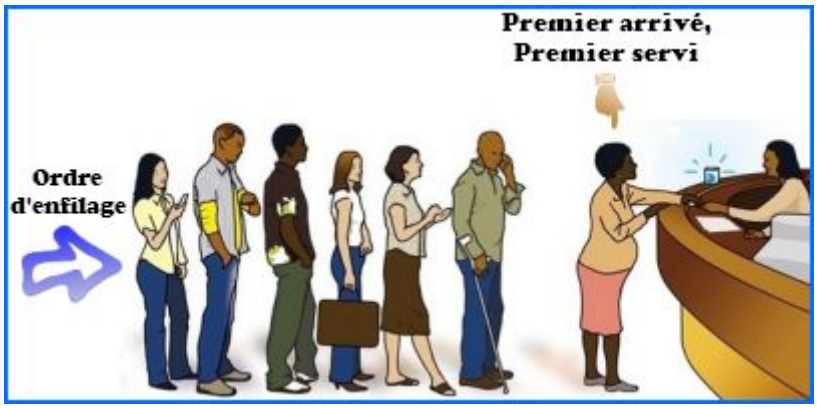
Type abstrait File (Bac 🎯)

Interface

"Point de cours 1"

Une **file** est un ensemble ordonné d'éléments qui se manipule comme une *file d'attente* :

- on peut insérer un élément à la fin de la file, c'est l'opération **enfiler**
- on peut tester si la file est vide, c'est l'opération **file_vide**
- si la file n'est pas vide, on peut retirer l'élément en début de **file**, c'est l'opération **défiler**



L'interface du type abstrait **File** peut se réduire à quatre opérations :

Opération	Signature	Description
creer_file	creer_file()	Renvoie une file vide
file_vide	file_vide(file)	Renvoi un booléen indiquant si la file est vide
enfiler	enfiler(file, elt)	Ajoute `elt` à la fin de la file
defiler	defiler(file)	Retire l'élément au début de la file et le renvoie

Quelques propriétés à retenir :

- Le premier élément qu'on peut retirer d'une file est forcément le premier à y être entré, c'est une structure **First In First Out (FIFO)**.

Acronyme anglais	Signification	Structure
LIFO	Dernier entré premier sorti	Pile
FIFO	Premier entré premier sorti	File

- En particulier si on défile tous les éléments, l'ordre dans lequel on les retire de la file est le même que leur ordre d'insertion.
- La séquence d'éléments dans la **file** peut être représentée par une **liste** mais contrairement à une **pile** on a besoin d'accéder à deux éléments qui sont aux extrémités de la liste.
- Le type File nécessite un accès aux deux extrémités d'une liste donc une implémentation par une *liste chaîné immuable* avec des `tuples` n'est pas possible comme pour le type Pile. On proposera des implémentations de *file mutable* mais on verra une implémentation du type File avec deux piles, qui peut se décliner en *file immuable* si on utilise des *piles immuables* implémentées avec des `tuples` .
- ⏱️ L'accès à deux extrémités peut se traduire par des complexités différentes si on a besoin de parcourir toute la structure pour atteindre l'une des extrémités. Selon l'implémentation on aura :
  - une complexité constante en $O(1)$ et une linéaire en $O(n)$ pour les opérations `defiler` et `enfiler` .
  - une complexité constante en $O(1)$ pour les deux opérations, ce qui peut être réalisé sans trop d'efforts.

"Exemple 1 : applications des files"

La structure de File se retrouve dans de nombreuses situations où on doit gérer un ensemble d'éléments :

- la file d'attente à un guichet de gare, à une caisse de supermarché, pour accéder à une formation sélective sur Parcoursup ...
- le stockage des yaourts dans un rayon de supermarché : le client *défile* le yaourt qui se trouve devant, le cariste *enfile* les nouveaux produits derrière.

On retrouve aussi la structure de File dans de nombreuses situations en informatique :

- les *programmes en cours d'exécution* ou *processus* accédent à tour de rôle à la ressource processeur qui n'exécute qu'un seul programme à la fois, ils sont placés dans une *file de priorité* : le modèle du *tourniquet* équivaut à celui d'une file d'attente : le *processus* qui

achève son temps d'accès processeur vient se placer en fin de file et le *processus* en début de file accède à son tour au processeur.
- les travaux d'impressions lancés sur une imprimante en réseau sont placés dans une *file d'impression*
- lors du parcours d'un graphe en largeur, on maintient une file d'attente des prochains sommets à visiter.

Implémentations

"Avec une liste chaînée"

On peut implémenter le **type abstrait File** par une *liste chaînée mutable* en reprenant l'implémentation objet du type Pile mais au lieu d'un unique attribut `contenu` pointant sur la première cellule de la liste assimilée au sommet de la pile, il faut deux attributs :

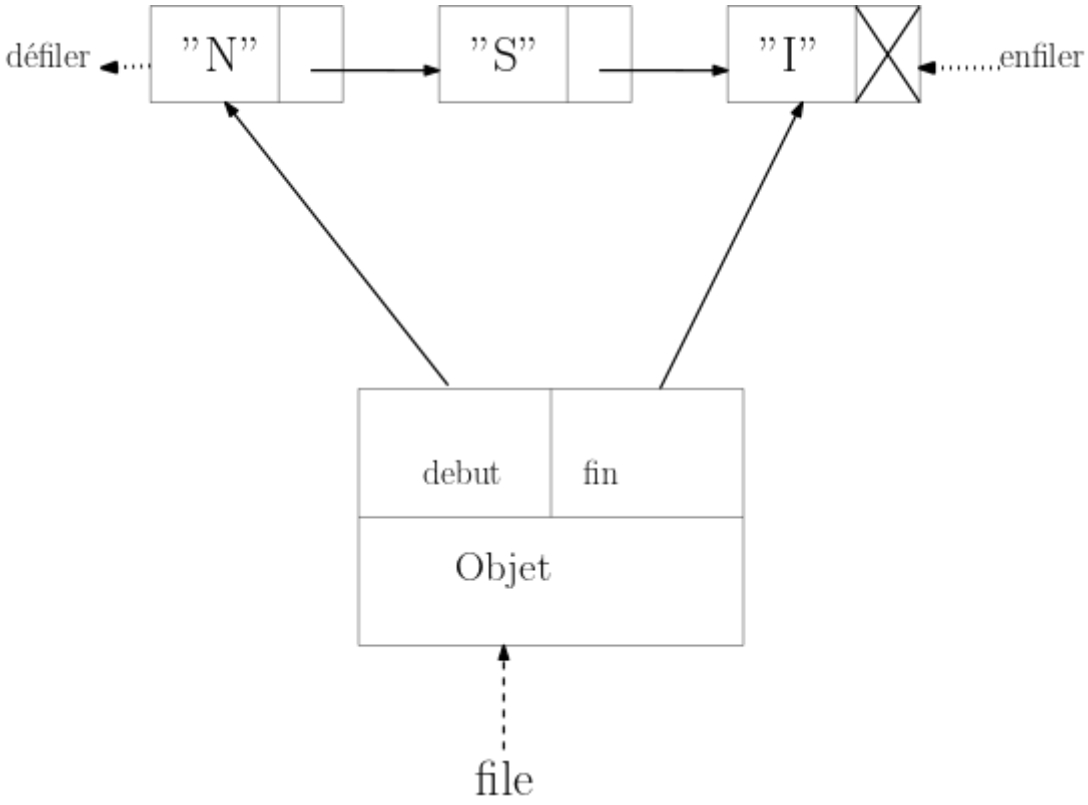
- pour l'opération `defiler` : un attribut `debut` pointant sur la première cellule de la liste
- pour l'opération `enfiler` : un attribut `fin` pointant sur la dernière cellule de la liste

> ⏱️ Ces deux attributs vont permettre de réaliser les opérations `enfiler` et `defiler` en temps constant en $0(1)$, par l'ajout ou la suppression de liens entre cellules, mais attention 👎

"subtilité de la gestion de deux attributs"

Une file est vide si et seulement si les deux attributs `debut` et `fin` ne pointent pas vers une cellule et sont positionnés à `None` .
Il faut donc être vigilant à bien modifier ces deux attributs lorsqu'on passe d'une file vide à une file non vide et vice versa

- si l'opération `defiler` donne une file vide, l'attribut `debut` va prendre la valeur `None` et il faut penser à passer aussi `fin` à `None` sinon il va pointer encore sur la cellule qui a été défilée.
- si l'opération `enfiler` s'applique à une file vide, l'attribut `debut` va pointer vers une nouvelle cellule, il faut penser à faire pointer `fin` vers cette même celulle sinon il pointe encore vers `None` .


défiler
"N"
"S"
"I"
enfiler
debut
fin
Objet
file

```python
class Cellule:

    def __init__(self, elt, suivant):
        self.element = elt
        self.suivant = suivant

class File:

    def __init__(self):
        self.debut = None
        self.fin = None

    def file_vide(self):
        return (self.debut is None) and (self.fin is None)

    def defiler(self):
        assert not self.file_vide(), "File Vide"
        elt = self.debut.element
        self.debut = self.debut.suivant
        if self.debut is None:
            self.fin = None
        return elt

    def enfiler(self, elt):
        if self.file_vide():
            self.fin = Cellule(elt, None)
            self.debut = self.fin
        else:
            self.fin.suivant = Cellule(elt, None)
            self.fin = self.fin.suivant
```

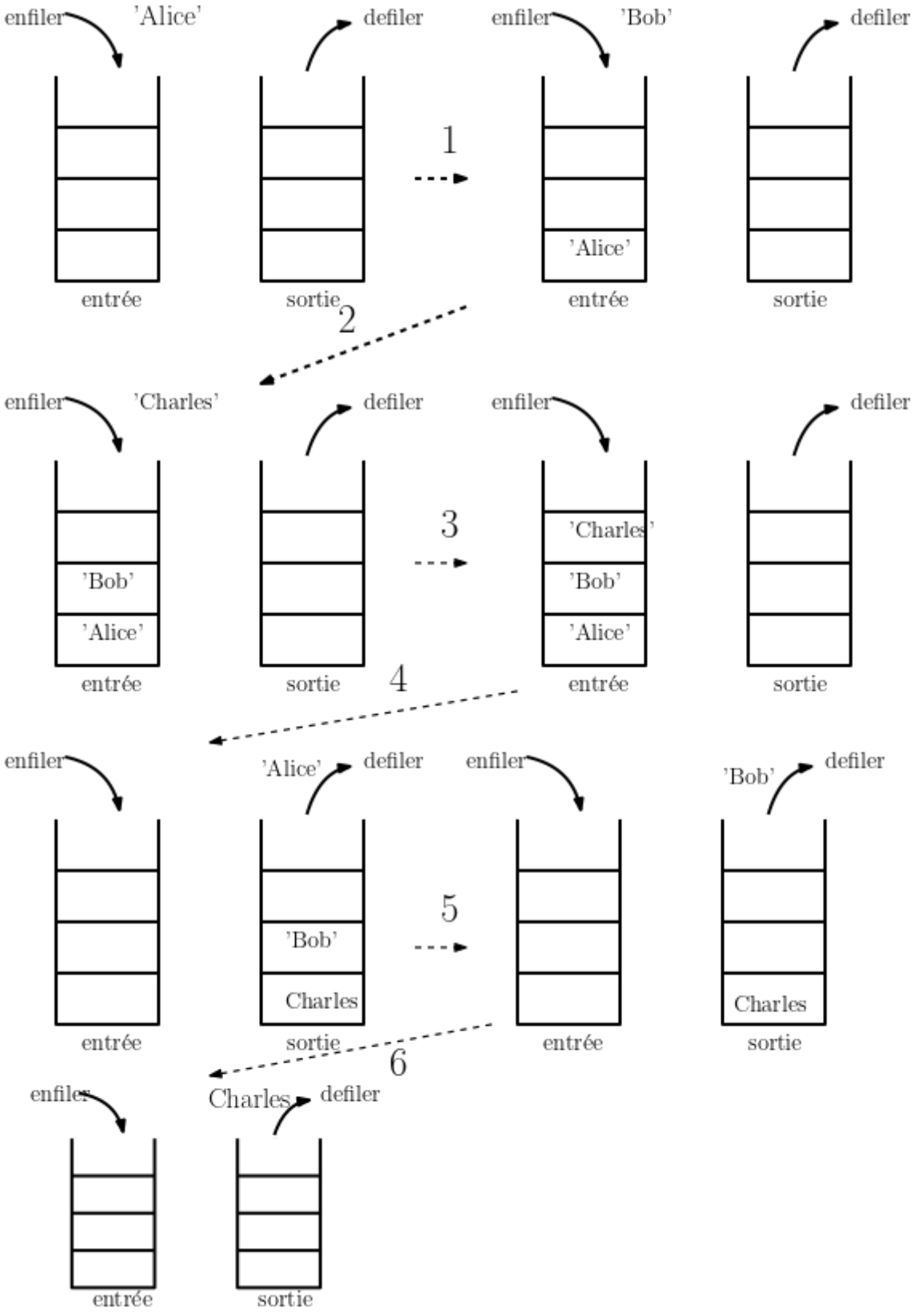
"Avec deux piles"

Une autre implémentation du **type abstrait File** consiste à utiliser deux piles, `entree` et `sortie`. On ajoute les éléments sur la pile `entree` et on les retire de la pile `sortie`. Si la pile `sortie` est vide on y transfère les éléments de la pile `entree`.

Entre son entrée et sa sortie, un élément subit deux opérations de dépilement, chacune inverse l'ordre et donc au final l'ordre est conservé (penser que la composée de deux fonctions décroissantes est une fonction croissante !).

Dans l'exemple ci-dessous :

- on enfile successivement `'Alice', 'Bob', 'Charles'` qui sont empilés dans la pile `entree`
- quand on veut défiler pour la première fois (étape 4), comme la pile `sortie` est vide on transfère d'abord la pile `entree` sur la pile `sortie` par des dépilements successifs (logique LIFO) puis on dépile (logique LIFO) le sommet de `sortie` qui est `'Alice'`. C'est bien le premier élément inséré dans la structure : la composition de deux logiques LIFO équivaut à une logique FIFO


enfiler
'Alice'
defiler
entrée
sortie
1
'Alice'
enfiler
'Bob'
defiler
entrée
sortie
2
enfiler
'Charles'
defiler
'Bob'
'Alice'
entrée
sortie
3
'Charles'
'Bob'
'Alice'
enfiler
defiler
entrée
sortie
4
enfiler
'Alice'
defiler
'Bob'
Charles
entrée
sortie
5
enfiler
'Bob'
defiler
Charles
entrée
sortie
6
enfiler
Charles
defiler
entrée
sortie

```python
class Cellule2:

    def __init__(self, elt, suivant):
        self.element = elt
        self.suivant = suivant

class Pile:

    def __init__(self):
        self.contenu = None

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

    def depiler(self):
        assert not self.pile_vide(), "Pile Vide"
        sommet = self.contenu.element
        self.contenu = self.contenu.suivant
        return sommet

    def empiler(self, elt):
        if self.pile_vide():
            self.contenu = Cellule2(elt, None)
        else:
            self.contenu = Cellule2(elt, self.contenu)

    def queue(self):
        assert not self.pile_vide()
        pile_queue = Pile()
        pile_queue.contenu = self.contenu.suivant
        return pile_queue

    def __str__(self):
        if self.pile_vide():
            return 'None'
        return f"({str(self.contenu.element)},{str(self.queue())})"

class File3:

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

```
    def file_vide(self):
        return self.entree.pile_vide() and self.sortie.pile_vide()

    def defiler(self):
        assert not self.file_vide(), "File Vide"
        if self.sortie.pile_vide():
            # si sortie vide  retourne la pile entree sur la pile sortie
            while not self.entree.pile_vide():
                self.sortie.empiler(self.entree.depiler())
        return self.sortie.depiler()

    def enfiler(self, elt):
        self.entree.empiler(elt)
```