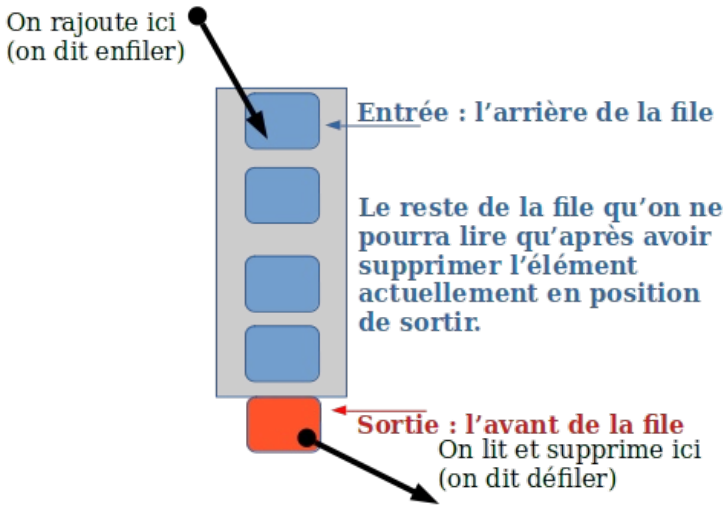
RÉSUMÉ 16 - TYPE ABSTRAIT FILE FIFO

Lien vers l'activité : type abstrait File FIFO
www.infoforall.fr - Dernière modif. : 25 10 2020

1 - TYPE ABSTRAIT FILE / QUEUE

ORGANISATION GÉNÉRALE

C'est une **séquence ordonnée et finie** d'éléments respectant le principe FIFO.
FIFO veut dire **First In First Out**. Le premier élément arrivé sera le premier à repartir.
On peut donc voir ce type abstrait comme une sorte de LISTE où on s'engage à n'agir que sur les extrémités : l'entrée dans la PILE est nommée **ARRIERE** et la sortie de la PILE se nomme **AVANT**.



FONCTIONS D'INTERFACE MINIMALE

Il en existe des versions mutables et non mutables. Ici, c'est non mutable.

- `nouvelleFile() -> File`
- `estVide(file:File) -> bool`
- `enfiler(elt:Elt, file:File) -> File` : on rajoute un élément ARRIERE
- `defiler(file:File) -> File` : on supprime l'élément AVANT.
- `lireAvant(file:File) -> Elt`

2 IMPLÉMENTATIONS

TABLEAU NE CONTENANT QUE LES DONNÉES

On retrouve l'utilisation d'un **TABLEAU** . Mais c'est peu efficace (linéaire au défilement)

DEUX PILES

On peut même implémenter une FILE comme un tableau de deux PILES.

- La pile `pileEntree` stocke les entrées dans la FILE. On la stocke à l'index 0 de la FILE.
- La pile `pileSortie` stocke les sorties de la FILE. On la stocke à l'index 1 de la FILE.
- Si on veut utiliser `defiler` ou `lireAvant` alors que `pileSortie` est vide, on déverse tous les éléments de la PILE d'entrée vers la pile de SORTIE : cela a pour effet d'inverser l'ordre d'arrivée dans la pile de sortie !

L'ensemble des deux piles se comportent donc comme une file. Il suffit de déverser la pile d'entrée dans la pile de sortie à chaque fois que la sortie est vide.
Voir le schéma ci-dessous.

LISTE CHAÎNÉE

Cette implémentation est efficace car on obtient un coût CONSTANT à l'enfilement et au défilement.
On trouve des versions avec des objets et des versions basées sur des tableaux servant de cellules.

TABLEAU CONTENANT INFOS ET TABLEAU DE DONNÉES

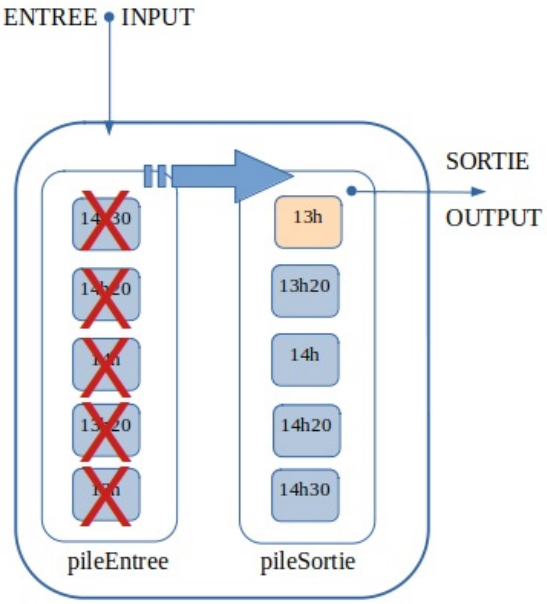
Coût CONSTANT également mais on utilise des tableaux non homogènes.

DICTIONNAIRE CONTENANT INFOS ET TABLEAU DE DONNÉES

Coût CONSTANT à l'enfilement et au défilement.

ET BIEN D'AUTRES ENCORE

On trouve des versions utilisant un simple tableau, deux tableaux... Tout est possible tant que cela respecte le principe FIFO.



IMPLÉMENTATION AVEC LE TYPE NATIF `list` DE PYTHON (PEU EFFICACE)

```python
class Elt :
    '''Juste pour pouvoir noter Elt dans les prototypes'''

class File :
    '''Juste pour pouvoir noter File dans les prototypes'''

'''Implémentation de type abstrait Pile en utilisant des tableaux dynamiques'''

def nouvelleFile() -> File:
    return []

def fileVide(f:File) -> bool:
    return f == []

def enfiler(elt:Elt, f:File) -> None :
    f.append(elt) # rajoute l'élément à la fin du tableau (ARRIERE pour nous)

def defiler(f:File) -> Elt:
    if not fileVide(f) :
        return f.pop(0) # supprime et renvoie le premier élément du tableau (AVANT pour nous)

def lireAvant(f:File) -> Elt:
    if not fileVide(f) :
        return f[0] # renvoie le contenu de l'index 0, AVANT pour nous
```

IMPLÉMENTATION CLASSIQUE À CONNAÎTRE : DEUX PILES

```python
class Elt :
    '''Juste pour pouvoir noter Elt dans les prototypes'''

class Pile :
    '''Juste pour pouvoir noter Pile dans les prototypes'''

class File :
    '''Juste pour pouvoir noter File dans les prototypes'''

'''Implémentation de type abstrait Pile en utilisant des tuples (sommet, autre_pile)'''

def nouvellePile() -> Pile :
    '''Renvoie une liste vide'''
    return ()

def pileVide(p:Pile) -> bool :
    '''Renvoie True si la pile est vide'''
    return p == ()

def empiler(elt:Elt, p:Pile) -> Pile :
    '''Renvoie une nouvelle pile où elt est le sommet et pile la suite de notre nouvelle pile.'''
    return (elt, p)

def lireSommet(p:Pile) -> Elt :
    '''Renvoie la valeur stockée au sommet, sans la supprimer de la pile'''
    if not pileVide(p)  :
        return p[0]

def depiler(p:Pile) -> Pile :
    '''Renvoie une nouvelle pile où on a supprimé l'ancien sommet'''
    if pileVide(p) : # la liste envoyée est ()
        return nouvellePile()
    else :
        return p[1]

# Fonction de l'implémentation FILE hors Interface

def deverser(f:File) -> None :
    while not pileVide(f[0]) :
        element = lireSommet(f[0])
        f[0] = depiler(f[0])
        f[1] = empiler(element, f[1])

# Fonctions d'interface de la FILE

def nouvelleFile() -> File :
    pileEntree = nouvellePile() # inputStack en anglais
    pileSortie = nouvellePile() # outputStack en anglais
    return [pileEntree, pileSortie]

def fileVide(f:File) -> None :
    return pileVide(f[0]) and pileVide(f[1])

def enfiler(elt:Elt, f:File) -> None:
    f[0] = empiler(elt, f[0])

def defiler(f:File) -> Elt:
    if not fileVide(f) :
        if pileVide(f[1]) :
            deverser(f)
        valeur = lireSommet(f[1])
        f[1] = depiler(f[1])
        return valeur

def lireAvant(f:File) -> Elt :
    if not fileVide(f) :
        if pileVide(f[1]) :
            deverser(f)
        return lireSommet(f[1])
```

IMPLÉMENTATION EFFICACE : LA LISTE CHAÎNÉE

```python
'''Implémentation de la file sous forme d'une liste chaînée'''

# Déclaration des Classes Cellule et File
class Elt :
    '''Juste pour pouvoir noter Elt dans les déclarations des fonctions'''

class Cellule :
    '''Classe permettant de créer des cellules-maillons basiques'''

    def __init__(self, valeur, suivant) :
        assert isinstance(suivant, Cellule) or suivant == None
        self.v = valeur
        self.s = suivant

    def renvoyerCelluleFinale(self) :
        if self.s == None :
            return self
        else :
            return self.s.renvoyerCelluleFinale()

class File :
    '''Classe implémentant une Liste sous forme Liste chaînée '''

    def __init__(self) :
        self.avant = None
        self.arriere = None
        self.nombre = 0

# Fonction d'observation qui n'a rien à faire dans l'interface d'une File !
def afficherFile(f) :
    '''Fonction debug, ce n'est pas une fonction d'interface. Affiche Avant -> Arrière'''
    tableau = recupererValeur(f.avant)
    return tuple(tableau)

def recupererValeur(cellule) :
    if cellule.s == None :
        return [cellule.v]
    else :
        return [cellule.v] + recupererValeur(cellule.s)

# Fonctions d'interface de la FILE

def nouvelleFile() -> File :
    return File()

def fileVide(f:File) -> bool :
    return f.avant == None

def enfiler(x:Elt, f:File) -> None :
    '''On insére la nouvelle valeur à l'arrière'''
    if fileVide(f) :                # C'est vide : Avant et Arrière sont la même Cellule
        cellule = Cellule(x, None)
        f.avant = cellule
        f.arriere = cellule
        f.nombre = 1
    else :                          # Il y a déja une cellule Arrière : on lui rajoute une suite
        cellule = Cellule(x, None)
        f.arriere.s = cellule   # on rajoute la cellule à la suite de l'actuelle arrière
        f.arriere = cellule     # on déclare la nouvelle cellule comme l'arrière
        f.nombre = f.nombre + 1

def lireAvant(f:File) -> Elt :
    if not fileVide(f) :
        return f.avant.v

def defiler(f:File) -> Elt :
    if not fileVide(f) :
        reponse = f.avant.v       # on mémorise l'ancienne valeur à l'avant
        f.avant = f.avant.s       # la nouvelle Cellule avant est le successeur de la précédente
        if f.avant == None :      # si l'avant n'existe pas en réalité
            f.arriere = None
        f.nombre = f.nombre - 1
        return reponse            # on renvoie la valeur supprimée

def taille(f:File) -> int :
    return f.nombre
```