

Classe Pile : Implémentation Liste Chaînée

```
maillonPile = Struct  
    (valeur = object,  
     suivant = SAME)
```

class Pile:

```
def __init__(self):  
    self._sommet = None
```

```
def est_vide(self):  
    b = (self._sommet == None)  
    return b
```

```
def est_pleine (self):  
    # pas de limitation du nb. d'éléments  
    b = False  
    return b
```

```
def empile (self,e):  
    #insertion d'un nouveau noeud en tête de liste  
    tmp = maillonPile(valeur=e,suivant=self._sommet)  
    self._sommet = tmp
```

```
def sommet (self):  
    e = self._sommet.valeur  
    return e
```

```
def dépile (self):  
    self._sommet = self._sommet.suivant
```

Analyse Algorithmique

- Complexité des trois méthodes est $O(1)$
 - Les opérations n'impliquant que le premier maillon, tous les calculs sont réalisés indépendamment de la taille de la liste.

Plan

■ Piles

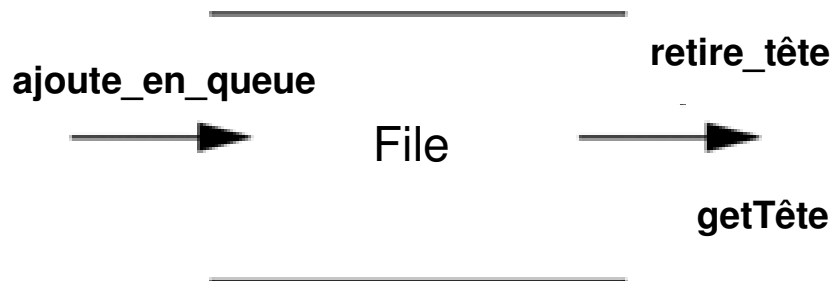
- Introduction
- Implémentation avec tableau
- Implémentation avec liste chaînée

■ Files

- Introduction
- Implémentation avec tableau
- Implémentation avec liste chaînée

■ Applications

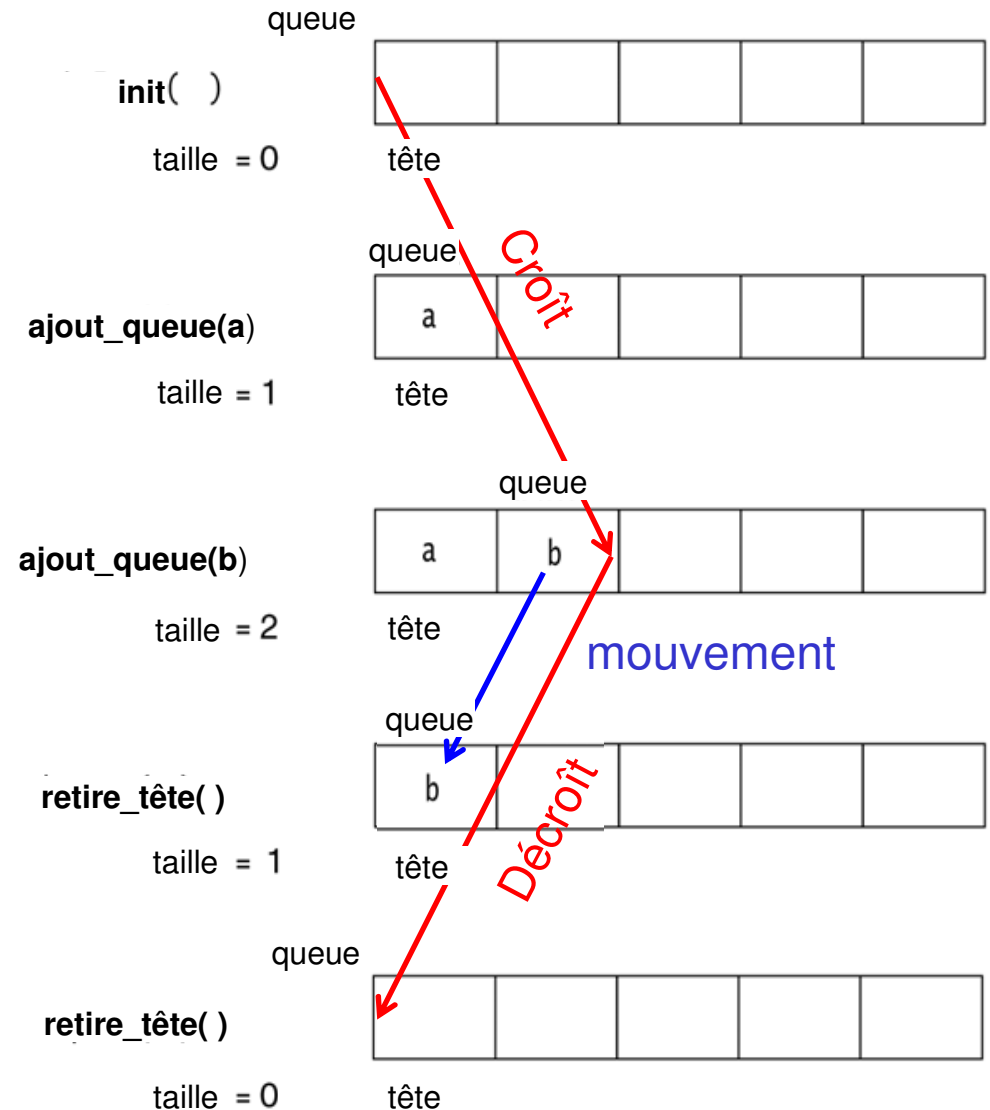
Introduction aux Files



- Une file est une liste pour laquelle l'élément accédé est le plus anciennement inséré
 - FIFO
- Les files jouent un rôle important dans des applications où une opération non achevée possède la priorité par rapport à une autre opération plus récente
 - Equité
 - Afin d'éviter la famine
- 3 opérations:
 - **ajoute_en_queue** – insertion d'un élément à l'arrière
 - **retire_tête** – suppression de l'élément de tête
 - **getTête** – détermine l'élément de tête sans le retirer

Implémentation avec Tableau

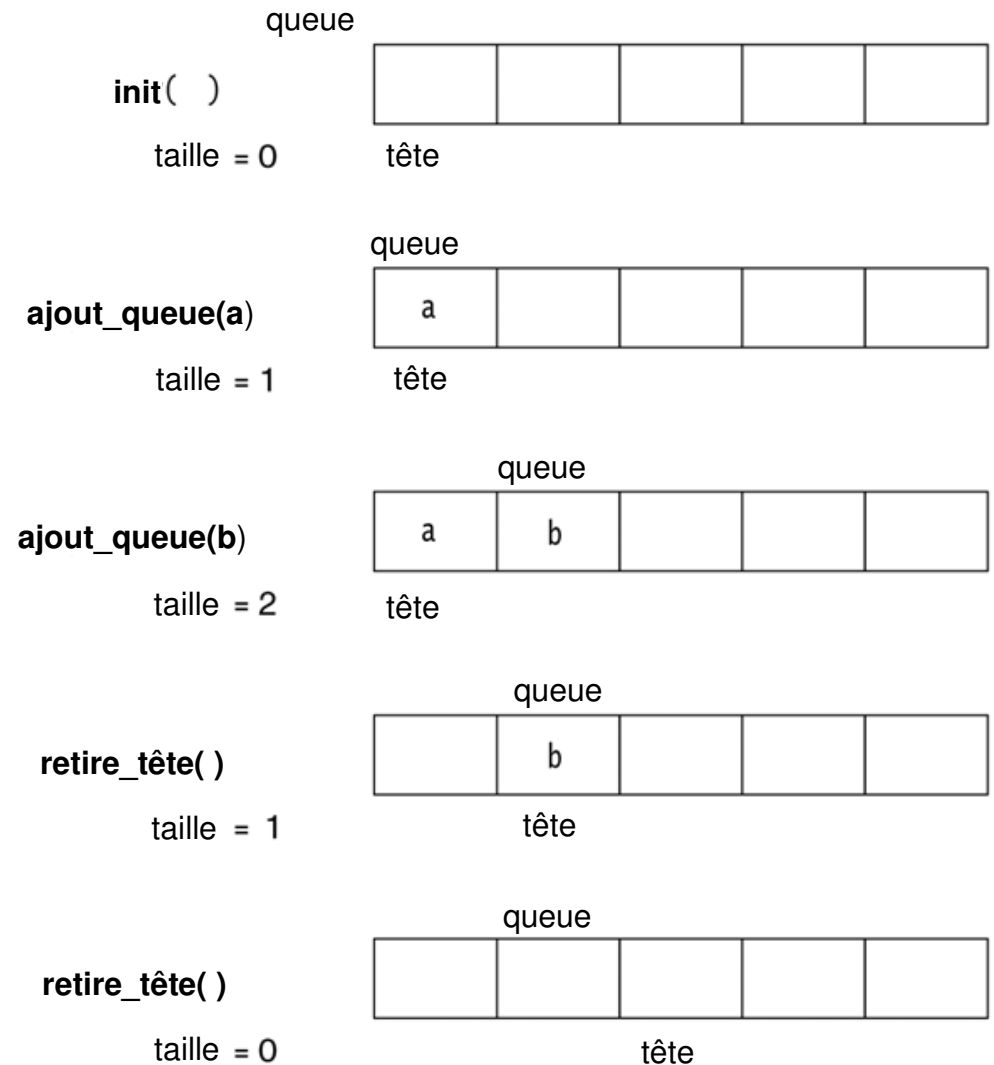
- Utilisation d'un tableau standard **n'est pas** une option naturelle dans la mesure où
 - La taille du tableau croît avec chaque opération **ajout_queue**
 - mais elle ne décroît pas avec chaque opération **retire_tête**
 - ◆ à moins que l'élément le plus récemment accédé occupe la position 0 etc...
 - ce qui nécessite de bouger tous les autres éléments



Implémentation avec Tableau Circulaire (1)

- Problème peut être résolu si

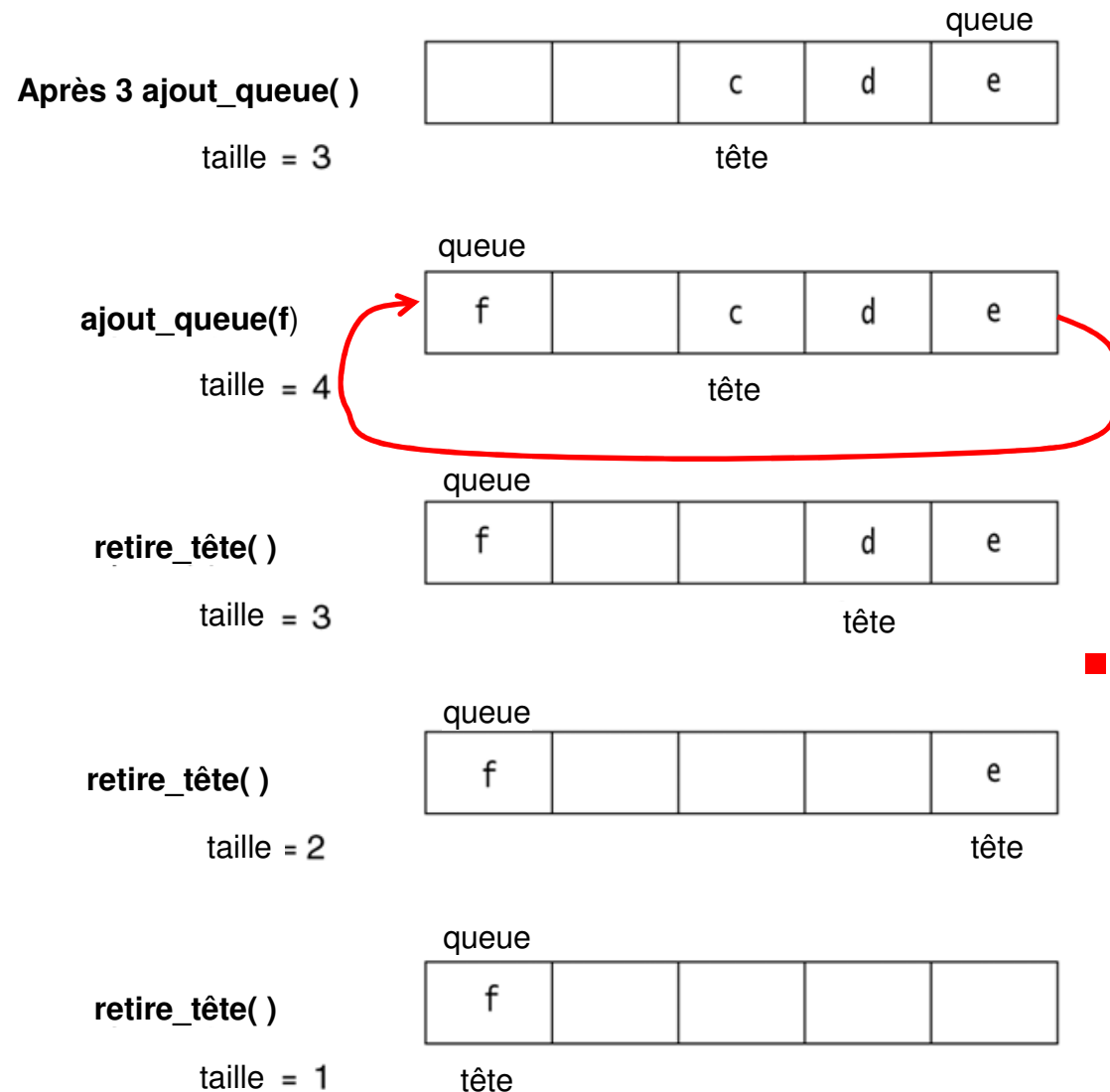
1. mise en oeuvre d'un élément *tête* avançant avec chaque opération *retire_tête*



Implémentation avec Tableau Circulaire (2)

2. le tableau est circulaire,

- ◆ i.e., la dernière position est reliée à la position d'indice 0.



- Dans ce cas, le fait que la file soit pleine ne peut être vérifié par l'élément *queue*;
 - utilisation d'un compteur qui donne le nombre d'éléments nbElements

Classe File : Interface

class File:

def __init__ (self):
"""

:sortie self:
:post-cond: file initialisée et vide
"""

def est_vide (self):
"""

:entrée self:
:sortie b: bool
:post-cond: b est True si la file est vide
"""

def est_pleine (self):
"""

:entrée self:
:sortie b: bool
:post-cond: b est True si la file est pleine
"""

def ajout_queue (self,e):
"""

:entrée e: object
:entrée-sortie self:
:pré-cond: la file n'est pas pleine
:post-cond: e est ajouté en queue de file
"""

def retire_tête (self):
"""

:entrée-sortie self:
:pré-cond: la file n'est pas vide
:post-cond: élément en tête de file retiré
"""

def getTête (self):
"""

:entrée self:
:sortie e: object
:pré-cond: la file n'est pas vide
:post-cond: retourne la valeur de l'élément en tête
"""

Classe File : Implémentation Tableau

MAX = 100

class File:

```
def __init__(self):
    self._tabfile = empty(MAX, object)
    self._tete = 0
    self._queue = -1
    self._nbElements = 0

#### méthode privée
def incremente(x)
    """
    :entrée-sortie x: int
    :pré-cond: x est un indice du tableau
    :post-cond: x est incrémenté ou vaut 0
                s'il est le dernier indice
    """
    if x == MAX-1:
        x = 0
    else:
        x = x+1
```

```
def est_vide(self):
    b = (self._nbElements == 0)
    return b
```

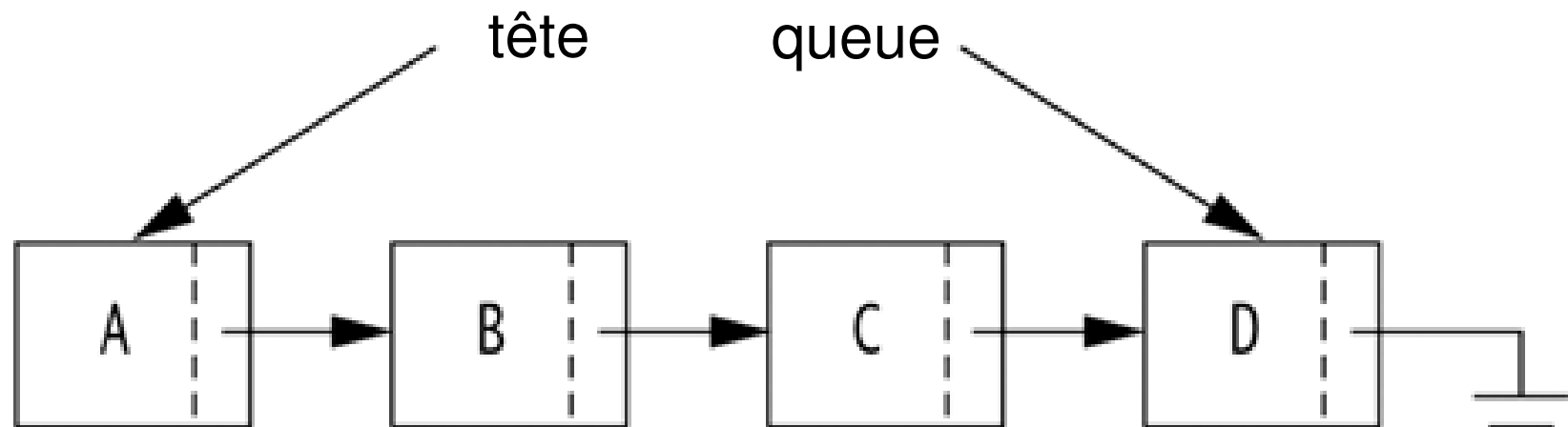
```
def est_pleine(self):
    b = (self._nbElements == MAX)
    return b
```

```
def ajout_queue(self,e)
    self._incrmente(self._queue)
    self._tabfile[self._queue] = e
    self._nbElements = self._nbElements+1
```

```
def getTête(self)
    e = self._tabfile[self._tête]
    return e
```

```
def retire_tête(self)
    self._nbElements = self._nbElements-1
    self._incrmente(self._tête)
```

File - Implémentation avec Liste Chaînée



Classe File : Implémentation Liste Chaînée

```
maillonFile = Struct (  
    valeur = object,  
    suivant = SAME )
```

class File:

```
def __init__(self):  
    self._tete = None  
    self._queue = None
```

```
def est_vide(self):  
    b = (self._tête == None)  
    return b
```

```
def est_pleine (self)  
    #pas de limitation du nb éléments  
    b = False  
    return b
```

```
def ajout_queue (self,e)  
    tmp = maillonFile(valeur=e, suivant=None)  
    if self.est_vide():  
        self._queue = tmp  
        self._tête = self._queue  
    else:  
        self._queue.suivant = tmp  
        self._queue = self._queue.suivant
```

```
def getTete (self)  
    e = self._tete.valeur  
    return e
```

```
def retire_tete (self)  
    self._tete = self._tete.suivant
```

Plan

■ Piles

- Introduction
- Implémentation avec tableau
- Implémentation avec liste chaînée

■ Files

- Introduction
- Implémentation avec tableau
- Implémentation avec liste chaînée

■ Applications

Application 1 : Texte bien parenthésé

Considérons un tableau de caractères représentant un texte. Ce texte contient des parenthèses rondes (« (» et «) ») et carrées (« [» et «] »).

A chaque parenthèse ouvrante doit correspondre une parenthèse fermante du même type, et réciproquement.

Par ailleurs, si une parenthèse ouvrante est ouverte à l'intérieur d'un autre couple de parenthèses, sa parenthèse fermante doit elle aussi se trouver à l'intérieur du même couple.

Le tableau ci-dessous donne des exemples de textes bien et mal parenthésés.

Bien parenthésés	Mal parenthésés
abc	(
(abc)	abc)
ab[cd]ef	ab)c
a[b]c(d)e	a(b)c
a((b)c)d	a(b(c)d
a(b[c()]e)f)g	a(b[c]d]e

Application 1 : Texte bien parenthésé

■ Conditions nécessaires

- Texte contient autant de « (» que de «) », et autant de « [» que de «] »
- Type (rond ou carré) d'une parenthèse fermante doit toujours correspondre au type de la dernière parenthèse ouvrante rencontrée (et non encore fermée)

■ Idée : stocker dans une pile les types des parenthèses ouvrante rencontrées

- Lorsqu'on rencontre une parenthèse fermante, il faut s'assurer qu'il reste dans la pile une parenthèse ouvrante à fermer, et que son type correspond à celui de la parenthèse fermante.
- Si ces conditions sont vérifiées, la parenthèse ouvrante est retirée de la pile. Sinon, on peut interrompre le traitement.
- A la fin du texte, il faut également vérifier qu'aucune parenthèse ne demeure ouverte dans la pile.

Application 1 : Texte bien parenthésé - Spécification

```
def bien_parenthese (txt):
```

```
    """
```

```
    :entrée txt: str
```

```
    :sortie b: bool
```

```
    :post-cond: b est True si le texte est bien parenthésé
```

```
    """
```

Texte bien parenthésé - Implémentation

```
def bien_parenthese (txt):  
    b = True  
    p = Pile()  
    i = 0  
    while b and i < len(txt):  
        if txt[i] == "(" or txt[i] == "[":  
            p.empile(txt[i])  
        elif txt[i] == ")":  
            if not p.est_vide() and p.sommet() == "(":  
                p.depile()  
            else:  
                b = False  
        elif txt[i] == "]":  
            if not p.est_vide() and p.sommet() == "[":  
                p.depile()  
            else:  
                b = False  
        i = i+1  
    if not p.est_vide():  
        b = False  
    return b
```


Application 2 : Inversion de l'ordre des éléments d'une pile - Spécification

def inverse_pile(p):

"""

:entrée/sortie p: Pile

:post-cond: inverse l'ordre des éléments de p

"""

Application 2 : Inversion de l'ordre des éléments d'une pile - Implémentation

■ Idée

- dépiler tous les éléments de la pile, puis les ré-empiler dans l'ordre où ils sont sortis. On utilisera donc une file pour le stockage intermédiaire.

```
def inverse_pile(p):  
    f = File()  
    while not p.est_vide():  
        e = p.sommet()  
        p.depile()  
        f.ajout_queue(e)  
    while not f.est_vide():  
        e = f.getTete()  
        f.retire_tete()  
        p.empile(e)
```

Exercice

Ecrire une méthode pour inverser l'ordre d'apparition des éléments d'une `LCListe` dont l'implémentation est réalisée avec maillons tête et queue factices. Par exemple, la liste "tete→a→b→c→queue" sera transformée en "tete→c→b→a→queue"

Spécification :

```
def inverse_liste (self):
```

```
    """
```

```
    :entrée-sortie self:
```

```
    :post-cond: l'ordre d'apparition des éléments de la liste  
                 est inversé
```

```
    """
```