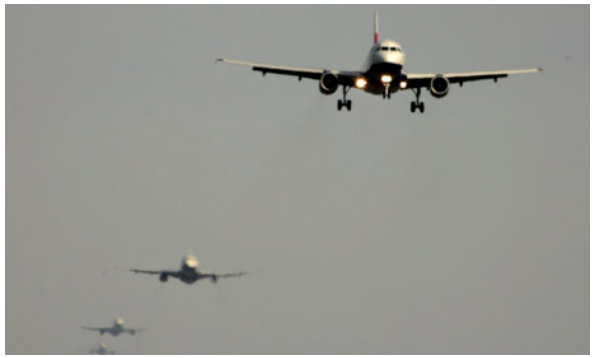
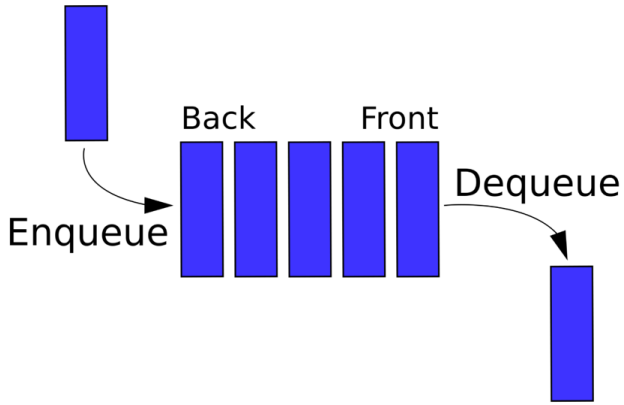
Files
ADT *Queue*
Applications
ListQueue

La File est caractérisé par deux opérations : `enqueue` (enfiler) et `dequeue` (défiler)





Pile de disques
Queue d'écoute
Inventé en 1925 par Eric Waterworth.

La File est caractérisée aussi par sa politique de premier entré premier sorti (first-in-first-out; FIFO). On parle souvent de "buffer", par exemple d'entrées ou de sorties, il s'agit en fait de files "d'attentes".

Opérations : exemple

Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

```
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "23 mars 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Queue
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

#ADT Queue (Classe de base)
class Queue:

    # constructeur
    def __init__( self ):
        pass

    # retourne le nombre d'éléments
    def __len__( self ):
        pass

    # produit une chaîne de caractères:
    # les éléments entre crochets
    # séparés par des virgules
    # taille et capacité de la structure de données
    # indiquées lorsque pertinent
    def __str__( self ):
        pass

    # indique s'il y a des éléments
    # dans la Queue
    def is_empty( self ):
        pass

    # ajoute un élément à la fin de la Queue
    def enqueue( self, element ):
        pass

    # retire le prochain élément de la Queue
    def dequeue( self ):
        pass

    # retourn le premier élément
    # en Queue sans le retirer
    def first( self ):
        pass
```

Applications

Applications directes :

- Listes d'attente, bureaucratie
- Accès aux ressources partagées (par exemple, imprimante)
- Multiprogrammation

Applications indirectes :

- Structure de données auxiliaires pour les algorithmes
- Composant d'autres structures de données

File basée sur une tableau

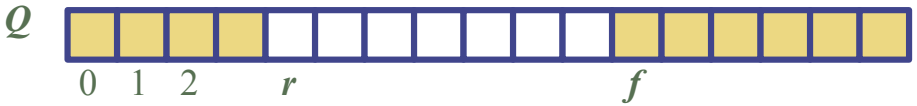
Utiliser un tableau de taille N de manière circulaire.
On utilise deux variables qui suivent l'avant et l'arrière de la file :

- *f* indice de l'élément avant (front)
- *r* index immédiatement après l'élément arrière (rear)

On utilise l'opération *modulo*
L'emplacement du tableau *r* est maintenu vide

configuration normale

configuration circulaire



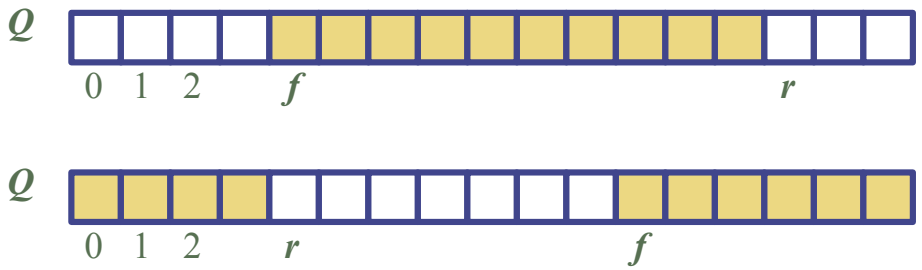
Algorithme ***size***()
  return($N - f + r$) mod N

Algorithme ***isEmpty***()
  return($f = r$)

Opération *enqueue*

```
Algorithme enqueue(o)
  if size() = N then
    throw FullQueueException
  else
    Q[r] = o
    r = (r + 1) mod N
```

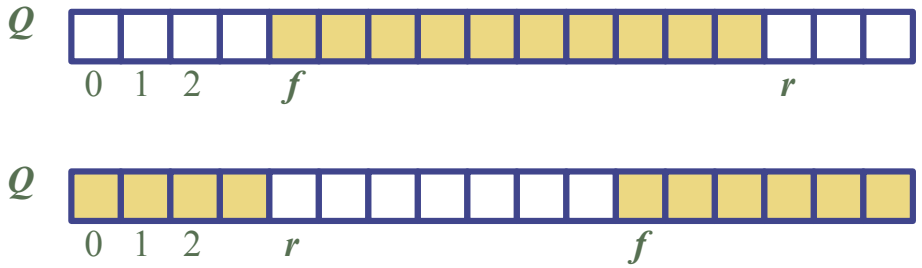
L'opération *enqueue* renvoie une exception si le tableau est plein.



Opération *dequeue*

```
Algorithme dequeue()
  if isEmpty() then
    throw EmptyQueueException
  else
    o = Q[f]
    f = (f + 1) mod N
    return o
```

L'opération *dequeue* renvoie une exception si la file d'attente est vide.



```
from Queue import Queue
class ListQueue( Queue ):

    DEFAULT_CAPACITY = 1

    def __init__( self, capacity = DEFAULT_CAPACITY ):
        self._data = [None] * capacity
        self._capacity = capacity
        self._size = 0
        self._front = 0

    # retourne une chaîne de caractères représentant la Queue
    # avec capacité
    def __str__( self ):
        pp = str( self._data )
        pp += "(size = " + str( len( self ) )
        pp += ")[first = " + str( self._front )
        pp += "; capacity = " + str( self._capacity ) + "]"
        return pp

    # retourne le nombre d'éléments
    def __len__( self ):
        return self._size

    # indique s'il y a des éléments
    # dans la Queue
    def is_empty( self ):
        return self._size == 0
```

```
# retourn le premier élément
# en Queue sans le retirer
def first( self ):
    if self.is_empty():
        return None
    else:
        return self._data[self._front]

# retire le prochain élément de la Queue
def dequeue( self ):
    # si aucun élément, on retourne None
    if self.is_empty():
        return None
    else:
        # on prend le premier élément
        elem = self._data[self._front]
        # on vide l'espace
        self._data[self._front] = None
        # on ajuste l'index du front
        self._front = ( self._front + 1 ) % len( self._data )
        # on décrémente la taille
        self._size -= 1
        # on retourne l'élément
        return elem

# ajoute un élément à la fin de la Queue
def enqueue( self, elem ):
    # on vérifie s'il y a de l'espace dans la liste
    # et sinon, on double sa capacité
    if self._size == len( self._data ):
        self._resize( 2 * len( self._data ) )
    # on calcule l'index du premier élément disponible
    avail = ( self._front + self._size ) % len( self._data )
    # on ajoute le nouvel élément à cet index
    self._data[avail] = elem
    # on incérmente la taille
    self._size += 1

# redimensionne le tableau à capacité c
def _resize( self, newcapacity ):
    old = self._data
    # on crée un nouveau tableau de capacité c
    self._data = [None] * newcapacity
    # on copie les éléments de l'ancien tableau dans le nouveau
    walk = self._front
    for k in range( self._size ):
        self._data[k] = old[walk]
        walk = ( 1 + walk ) % len( old )
    # on remet la tête de file au début de la liste et
    # on ajuste la capacité
    self._front = 0
    self._capacity = newcapacity
```