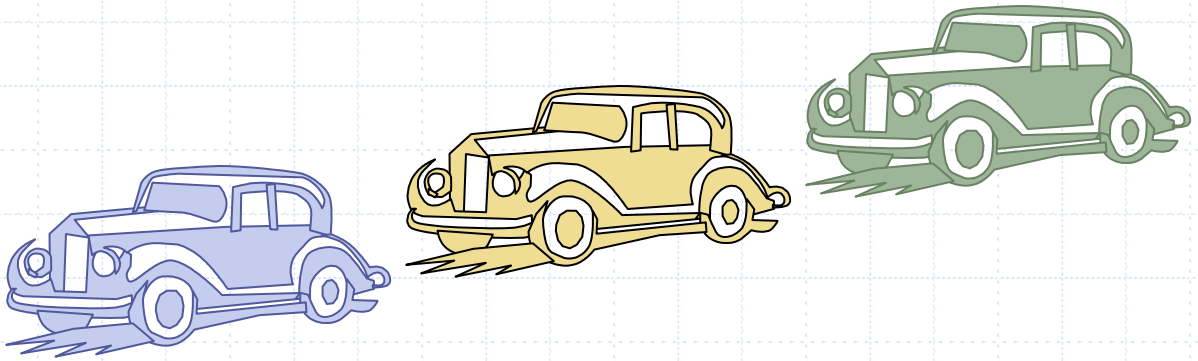
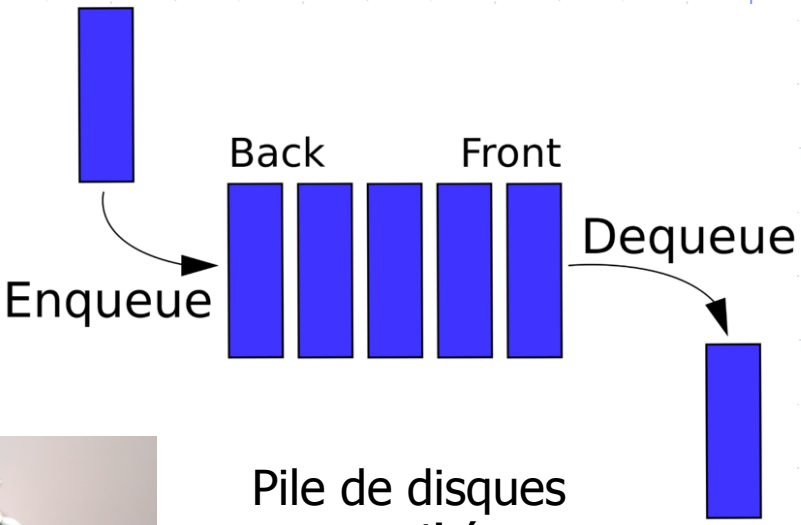
Queues



Le type abstrait `queue` est caractérisé par deux opérations : `enqueue` (enfiler) et `dequeue` (défiler)



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

- La queue 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 d'une queue.

The Queue ADT…

```
#ADT Queue "interface"
class Queue:

    def __init__( self ):
        pass

    #return the number of elements in Queue
    def __len__( self ):
        pass

    #convert a Queue into a string:
    # elements listed between brackets
    # separated by commas
    # element front and rear highlighted
    # size and capacity of the data structure
    # indicated
    def __str__( self ):
       pass
```

The Queue ADT

```
#indicate whether no element are
#stored in the Queue
def is_empty( self ):
    pass

#add element on the Queue
def enqueue( self, element ):
    pass

#remove an element from the Queue
def dequeue( self ):
    pass

#return the first element
#without removing it
def first( self ):
    pass
```

Example

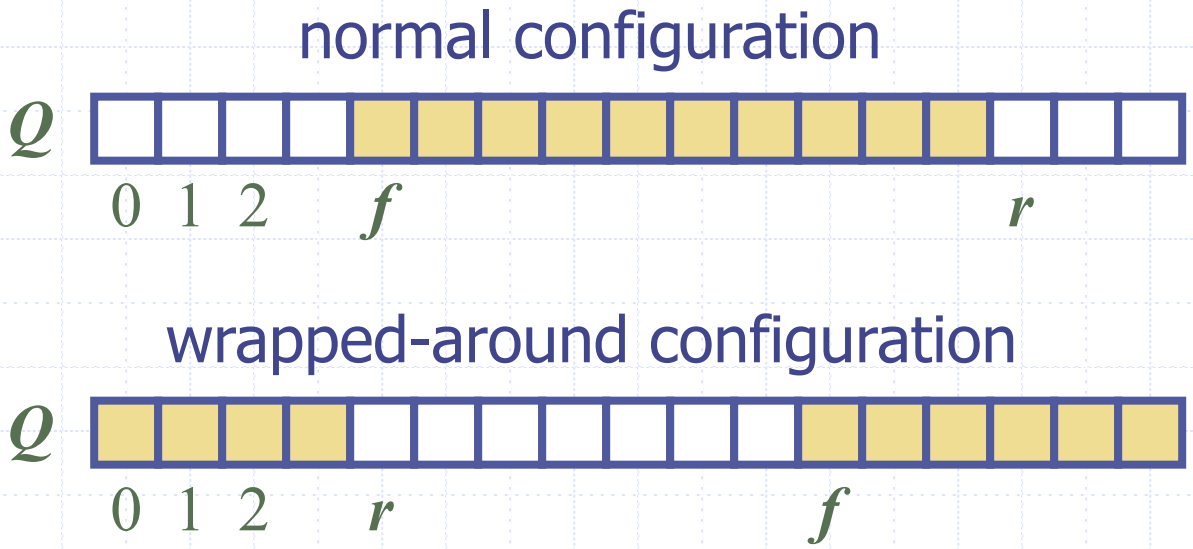
Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)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

Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

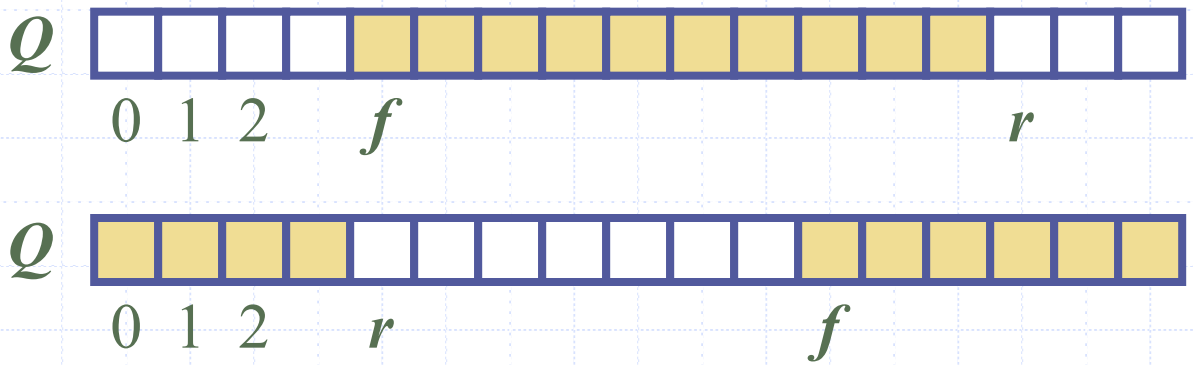


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

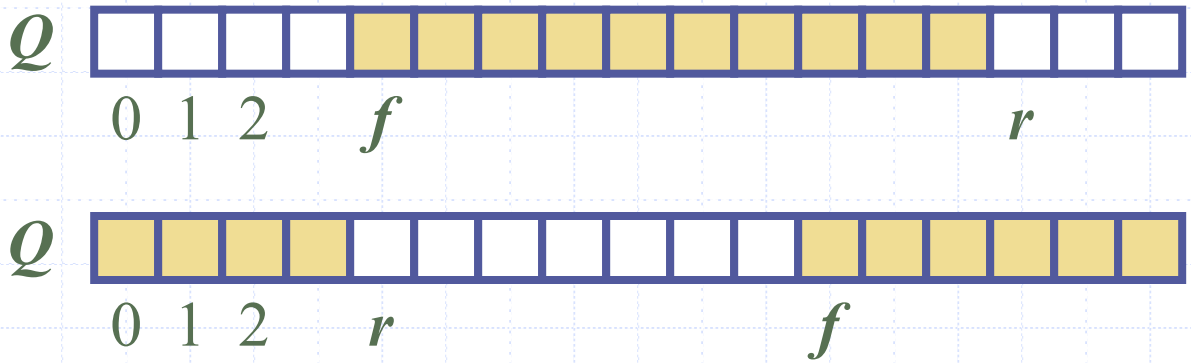
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

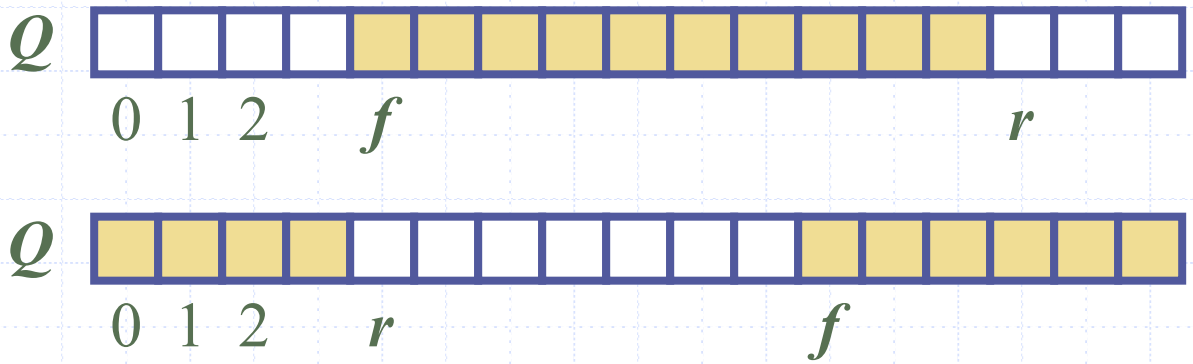
Algorithm *enqueue*(o)
  if *size*() = N **then**
    throw *FullQueueException*
  else
    $Q[r] = o$
    $r = (r + 1) \bmod N$



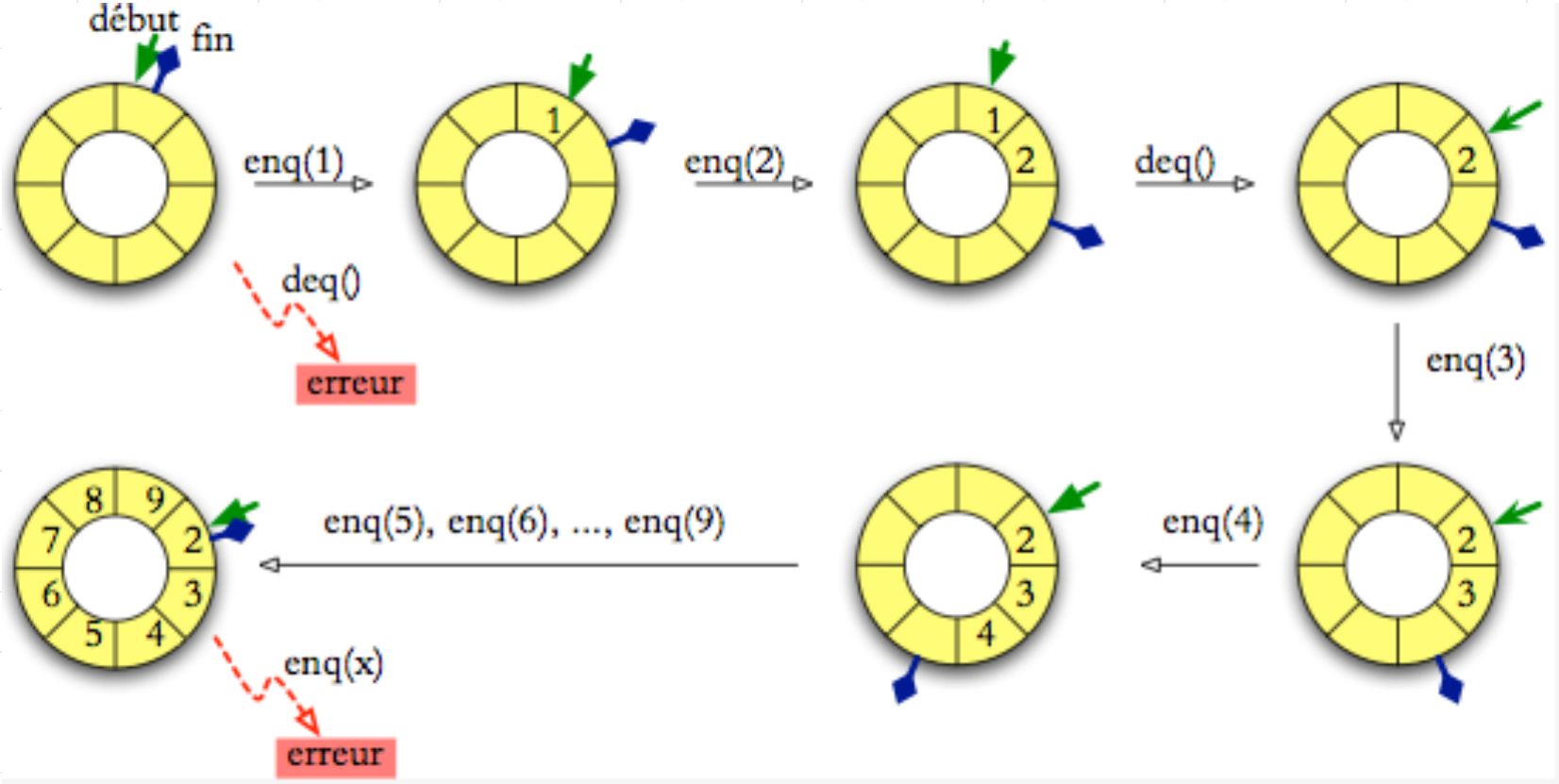
Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

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



Implantation du type abstrait `queue` avec une structure de données en anneau ("buffer" circulaire)



- On peut implanter une queue avec un anneau ("buffer" circulaire) dans un tableau. On utilise 2 indices pour le début et la fin de la queue et l'opération modulo n pour un tableau de taille n.

Queue in Python

- Use the following three instance variables:
  - _data: is a reference to a list instance with a fixed capacity.
  - _size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - _front: is an integer that represents the index within data of the first element of the queue (assuming the queue is not empty).

ArrayQueue...

```
class ArrayQueue:

    #implements the ADT Queue (Queue.py)
    #uses the python default List

    DEFAULT_CAPACITY = 1

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

ArrayQueue...

```
def __str__( self ):
    pp = str( self._data )
    pp += "(size = " + str( len( self ) )
    pp += ")[first = " + str( self._front )
    pp += "; capacity = " + str( self._capacity ) + "]"
    return pp

def __len__( self ):
    return self._size

def is_empty( self ):
    return self._size == 0
```

ArrayQueue...

```
def first( self ):
    if self.is_empty():
        return False
    else:
        return self._data[self._front]

def dequeue( self ):
    if self.is_empty():
        return False
    else:
        elem = self._data[self._front]
        self._data[self._front] = None
        self._front = ( self._front + 1 ) % len( self._data )
        self._size -= 1
        return elem
```

ArrayQueue

```
def enqueue( self, elem ):
    if self._size == len( self._data ):
        self._resize( 2 * len( self._data ) )
    avail = ( self._front + self._size ) % len( self._data )
    self._data[avail] = elem
    self._size += 1

def _resize( self, newcapacity ):
    old = self._data
    self._data = [None] * newcapacity
    walk = self._front
    for k in range( self._size ):
        self._data[k] = old[walk]
        walk = ( 1 + walk ) % len( old )
    self._front = 0
    self._capacity = newcapacity
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)