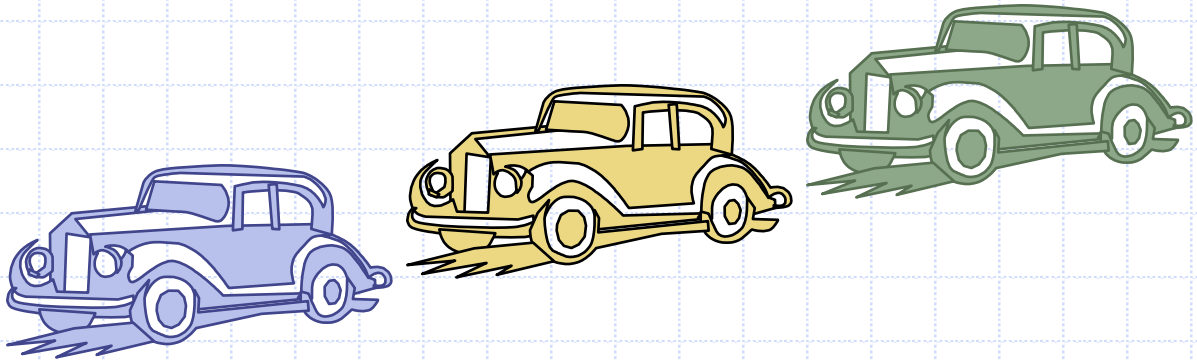
Queues



The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

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

```
# a queue using an array
class Queue:
    # To initialize the object.
    def __init__(self, c):
        self.data = []
        self.front = 0
        self.rear = 0
        self.capacity = c
```

Array-based Queue

- **enQueue:**
  - Addition of an element to the queue.
  - Adding an element will be performed after checking whether the queue is full or not.
  - If *rear* < *n* which indicates that the array is not full then store the element at *arr[rear]* and increment *rear* by *1* but if *rear* == *n* then it is said to be an Overflow condition as the array is full.

Array-based Queue

- **deQueue:**
  - Removal of an element from the queue.
  - An element can only be deleted when there is at least an element to delete i.e. *rear > 0*.
  - Now, element at *data[front]* can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.

Array-based Queue

- **first:**
  - Get the front element from the queue i.e. *data[front]* if queue is not empty.

Array-based Queue

- **display:**
  - Print all element of the queue.
  - If the queue is non-empty, traverse and print all the elements from index *front* to *rear*.
- len:
  - Return size of queue
- isEmpty:
  - Return true if queue is empty

Array-based Queue

```
# a queue using an array
class Queue:
    # To initialize the object.
    def __init__(self, c):
        self.data = []
        self.front = 0
        self.rear = 0
        self.capacity = c
```

Array-based Queue

```
# Function to insert an element
# at the rear of the queue
def enQueue(self, value):
    # Check queue is full or not
    if(self.capacity == self.rear):
        print("\nQueue is full")
    # Insert element at the rear
    else:
        self.data.append(value)
        self.rear += 1
```

Array-based Queue

```
# Function to delete an element
# from the front of the queue
def deQueue(self):
    # If queue is empty
    if(self.front == self.rear):
        print("Queue is empty")
    # Pop the front element from list
    else:
        x = self.data.pop(0)
        self.rear -= 1
```

Array-based Queue

```
# Function to print queue elements
def displayQueue(self):
    if(self.front == self.rear):
        print("\nQueue is Empty")
    # Traverse front to rear to
    # print elements
    for i in self.data:
        print(i, "<--", end = '')
```

Array-based Queue

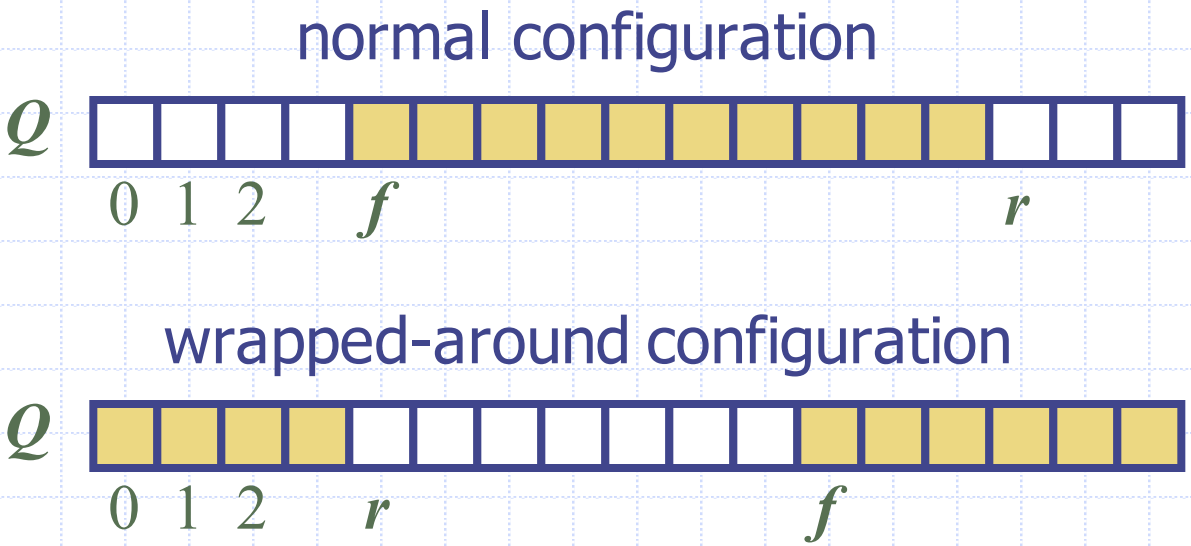
```
# Print front of queue, not delete
def first(self):
    if(self.front == self.rear):
        print("\nQueue is Empty")
    print("\nFront Element is:",
        self.data[self.front])
```

Array-based Queue

```
# return length of queue

def len(self):

  return len(self.data)

# return true if queue is empty

def isEmpty(self):

  return len(self.data)==0
```

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

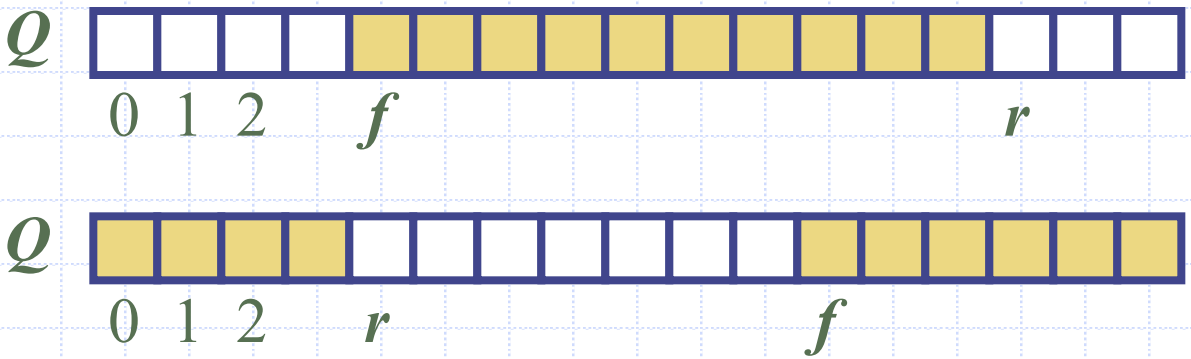


Queue Operations

- We use the modulo operator (remainder of division)

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

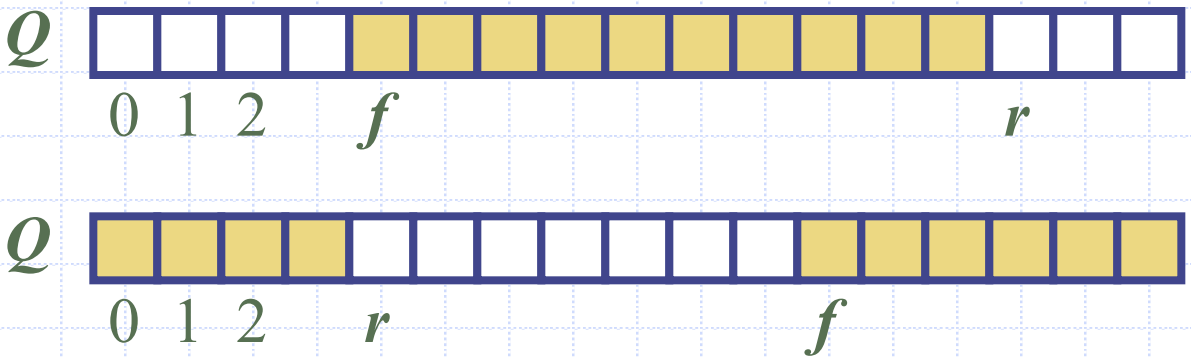
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

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

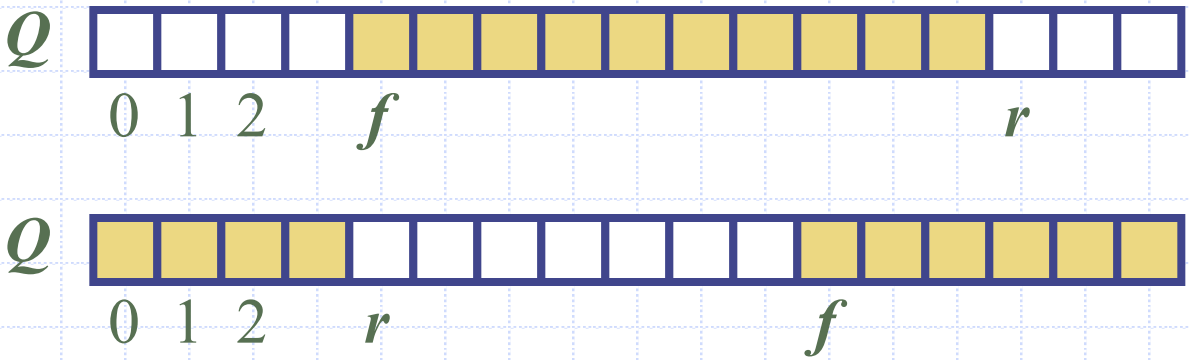
Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *FullQueueException*
  else
    Q[*r*] ← *o*
    r ← (*r* + 1) mod *N*



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
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return o



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).

Queue in Python, Beginning

```
1  class ArrayQueue:
2    """FIFO queue implementation using a Python list as underlying storage."""
3    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10
11   def __len__(self):
12     """Return the number of elements in the queue."""
13     return self._size
14
15   def is_empty(self):
16     """Return True if the queue is empty."""
17     return self._size == 0
18
```

```
19   def first(self):
20     """Return (but do not remove) the element at the front of the queue.
21
22     Raise Empty exception if the queue is empty.
23     """
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._data[self._front]
27
28   def dequeue(self):
29     """Remove and return the first element of the queue (i.e., FIFO).
30
31     Raise Empty exception if the queue is empty.
32     """
33     if self.is_empty():
34       raise Empty('Queue is empty')
35     answer = self._data[self._front]
36     self._data[self._front] = None          # help garbage collection
37     self._front = (self._front + 1) % len(self._data)
38     self._size -= 1
39     return answer
```

Queue in Python, Continued

```
40    def enqueue(self, e):
41      """Add an element to the back of queue."""
42      if self._size == len(self._data):
43        self._resize(2 * len(self.data))          # double the array size
44      avail = (self._front + self._size) % len(self._data)
45      self._data[avail] = e
46      self._size += 1
47
48    def _resize(self, cap):                       # we assume cap >= len(self)
49      """Resize to a new list of capacity >= len(self)."""
50      old = self._data                            # keep track of existing list
51      self._data = [None] * cap                   # allocate list with new capacity
52      walk = self._front
53      for k in range(self._size):                 # only consider existing elements
54        self._data[k] = old[walk]                 # intentionally shift indices
55        walk = (1 + walk) % len(old)              # use old size as modulus
56      self._front = 0                             # front has been realigned
```

Queue in Python- Linked list

- Do your self

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)