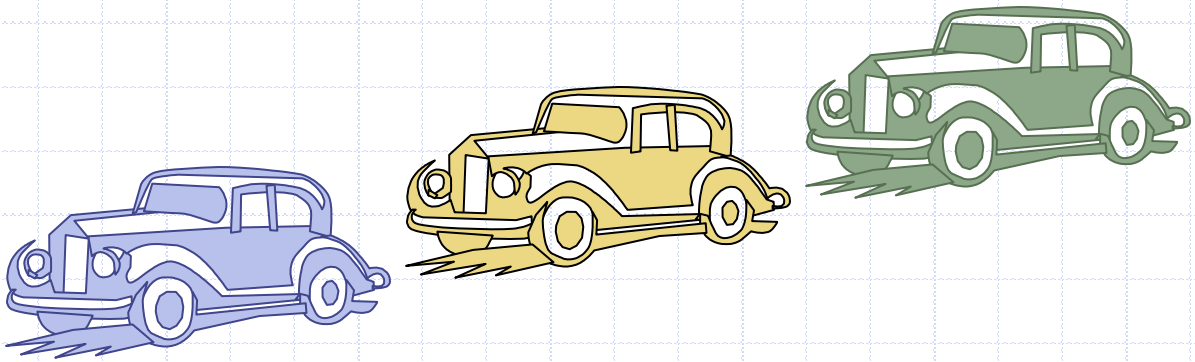
Queues



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

Queues

- First In First Out (FIFO)

  Remove the ***least*** recently added item

  A queue has a front and a rear

  Analogy: waiting lines at the supermarket

Example (conceptual view)

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

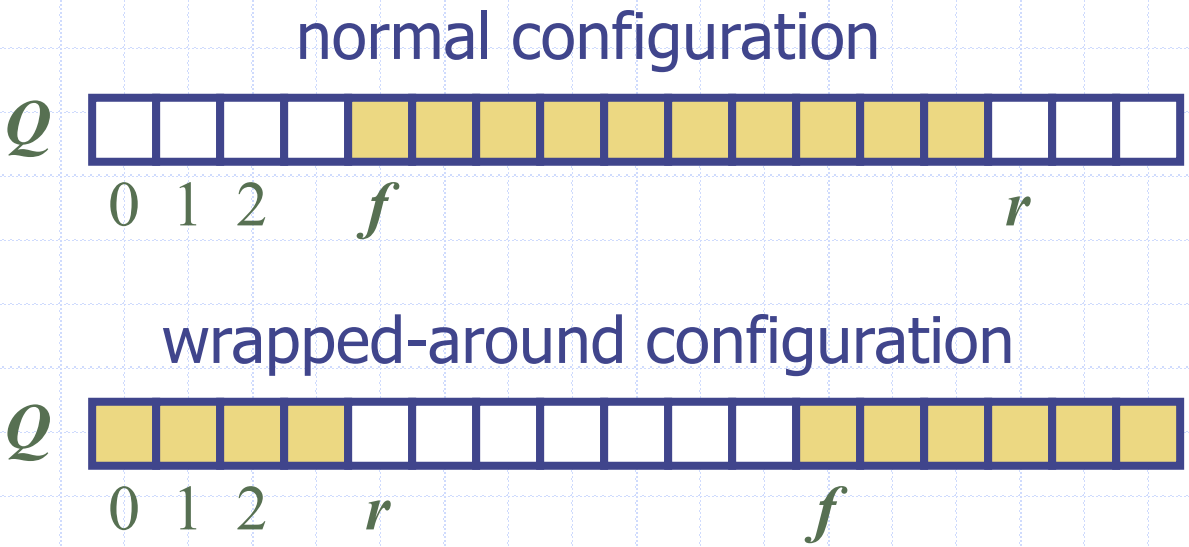
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Let's implement a queue (FIFO)

- Define a Queue class having following methods:
  - __init__(self): #Initialize a queue
  - __len__(self): #Return length of queue
  - is_empty(self): #Return True if queue is empty
  - enqueue(self,e): #Enqueue element **e** in the queue.
  - dequeue(self): return an element from the queue and delete that element.
  - front(self): returns the element at the front without removing it

Array-based Queue (Concept)

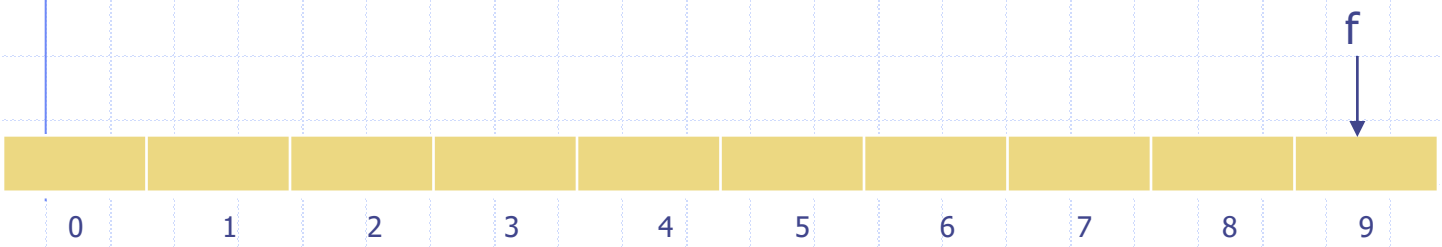
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - *size # of elements in Queue*
- r can be computed based on modulo: (f+size) % N



Example

By default: f = 0. But it is NOT a hard requirement!

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]



Queue status:

len(data) = 10
size = 0
f = 9

Enqueue(e): Put e into the available slot, update size

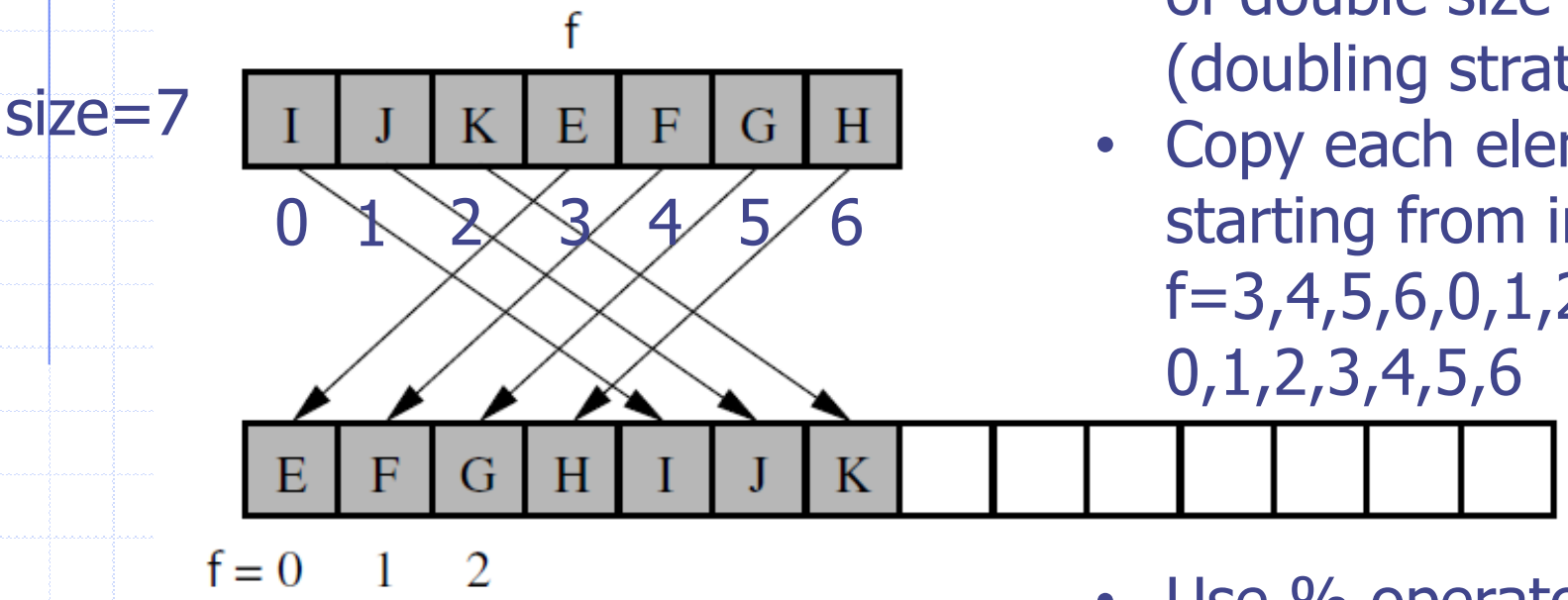
```
avail = (self._front + self._size) % len(self._data)
```

Dequeue(): remove the first element and return it; Then update f & size:

```
self._front = (self._front + 1) % len(self._data)
```

Resizing the Queue
(Queue in Python (text book), continued)

- Create a new array of double size (doubling strategy)
- Copy each element starting from index f=3,4,5,6,0,1,2 to 0,1,2,3,4,5,6



- Use % operator!

Resizing the queue, while realigning the front element with index 0

Queue in Python (text book)

- 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 (text book), 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 (text book), 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
```

Analyzing the Array-Based Queue

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Queue in Python (Our Approach)

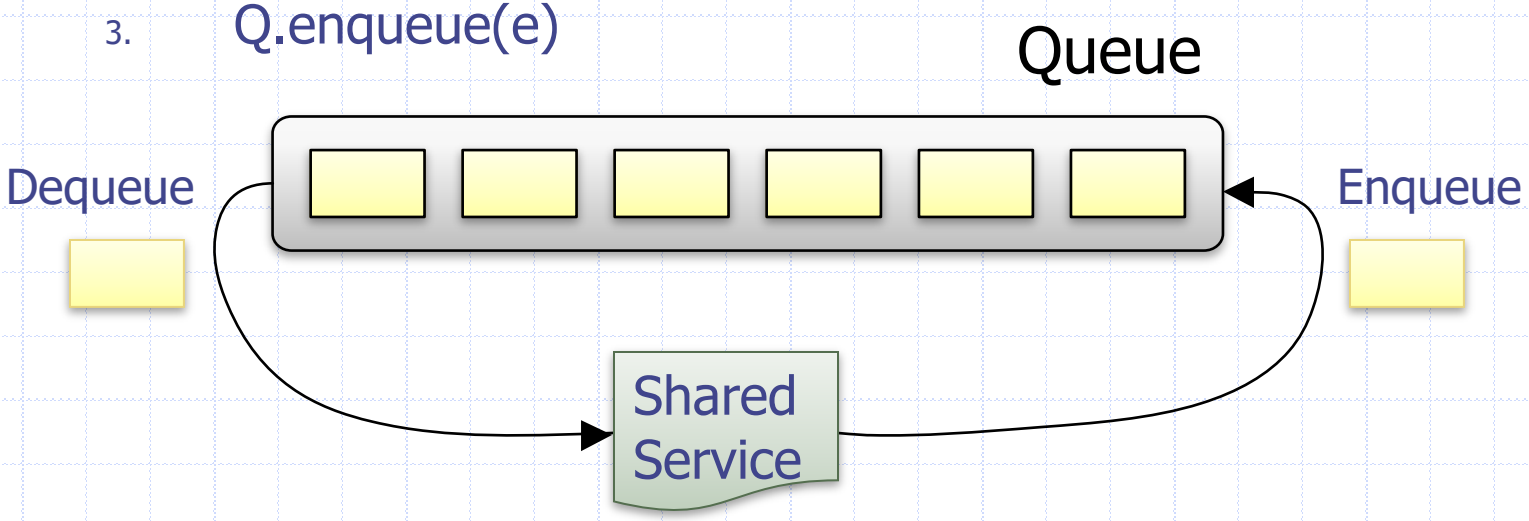
- Our Queue will be Fixed Size. It won't be automatically increase or decrease. If it's full, we will throw an FullQueueException.
- Use the following three instance variables:
  - _data: is a reference to a list instance with a fixed capacity.
  - _size: is an integer representing the current number 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).

Analyzing the Array-Based Queue (Our Approach)

Operation	Running Time
Q.enqueue(e)	O(1)
Q.dequeue()	O(1)
Q.first()	O(1)
Q.is_empty()	O(1)
len(Q)	O(1)

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)



Double Ended Queue (Deck) ADT

- D.add_first(e): Add element e to the front of deque D.
- D.add_last(e): Add element e to the back of deque D.
- D.delete_first( ): Remove and return the first element from deque D; an error occurs if the deque is empty.
- D.delete_last( ): Remove and return the last element from deque D; an error occurs if the deque is empty.
- D.first(): Return (but do not remove) the first element of deque D; an error occurs if the deque is empty.
- D.last(): Return (but do not remove) the last element of deque D; an error occurs if the deque is empty.
- D.is_empty( ): Return True if deque D does not contain any elements.
- len(D): Return the number of elements in deque D;

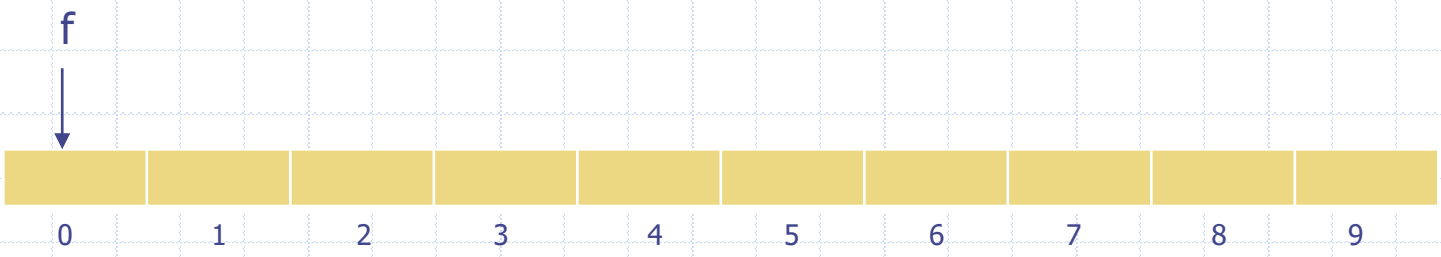
```
def add_first(self, e):
    loc = (self._front - 1) % len(self._data)
    self._data[loc] = e
    self._front = (self._front - 1) % len(self._data)
    self._size += 1

def add_last(self, e):
    loc = (self._front + self._size) % len(self._data)
    self._data[loc] = e
    self._size += 1

def delete_first(self):
    ans = self._data[self._front]
    self._data[self._front] = None
    self._front = (self._front + 1) % len(self._data)
    self._size -= 1
    return ans

def delete_last(self):
    loc = (self._front + self._size - 1) % len(self._data)
    ans = self._data[loc]
    self._data[loc] = None
    self._size -= 1
    return ans
```

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]



Deque status:

len(data) = 10
size = 0
f = 0