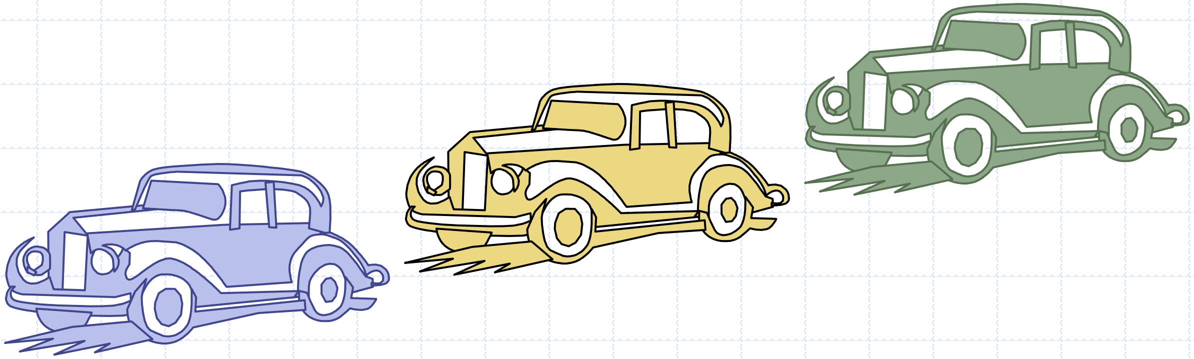
Queues



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

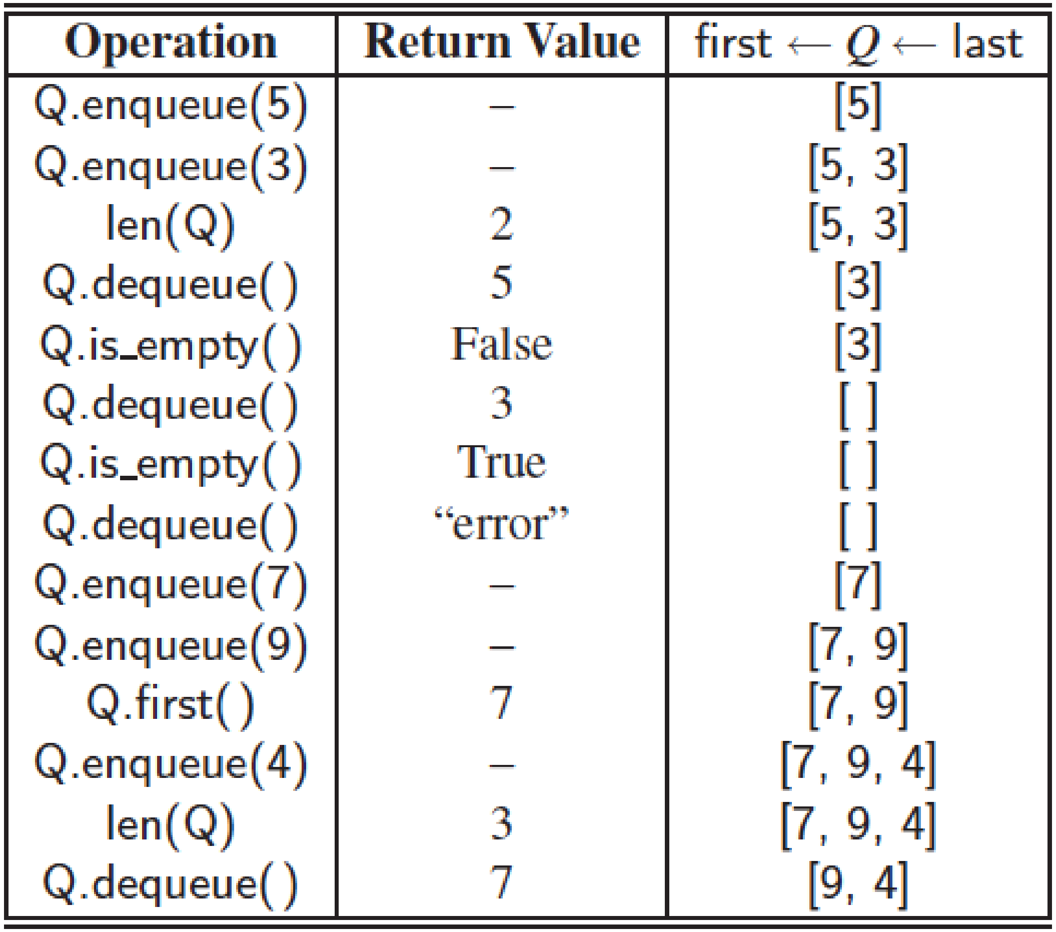
  Analogy: waiting lines at the supermarket

Queue in Action

- *Visualization time*

  http://www.cs.usfca.edu/~galles/visualization/QueueArray.html

Example



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]

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

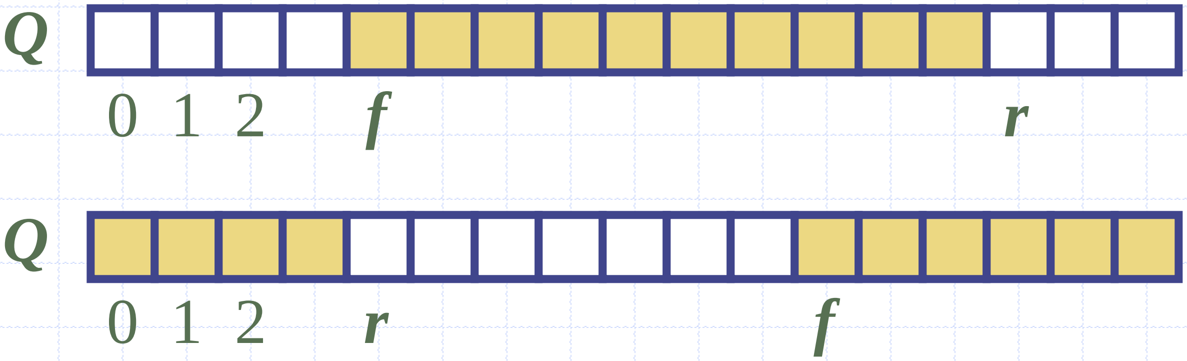
Let’s implement a queue (FIFO)

- 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

Queue Operations (Concept)

- We use the modulo operator (remainder of division)

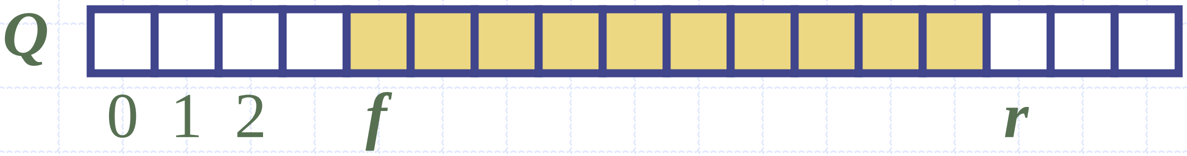
```
Algorithm size()
    return $(N - f + r) \bmod N$

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



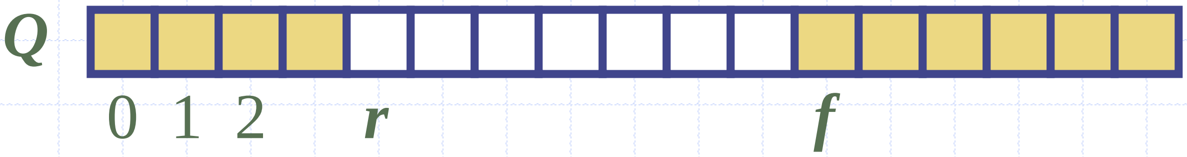
Array-based Queue (Concept)

- 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

normal configuration



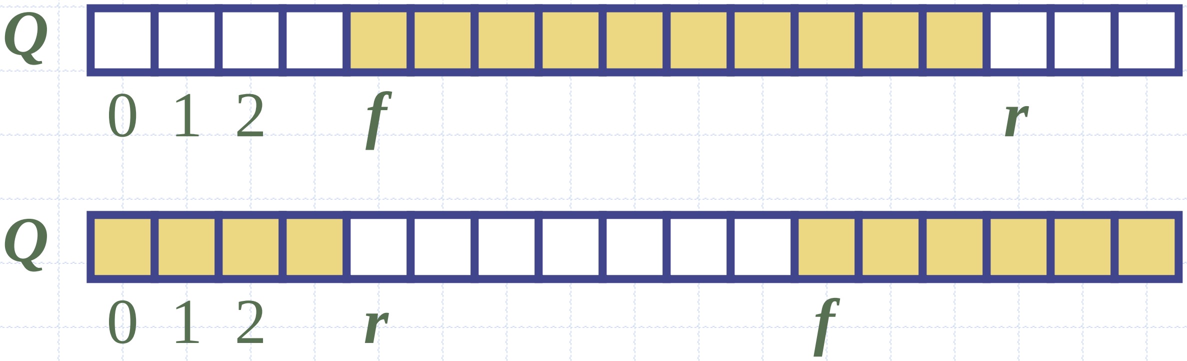
wrapped-around configuration



Queue Operations (Concept) (cont.)

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

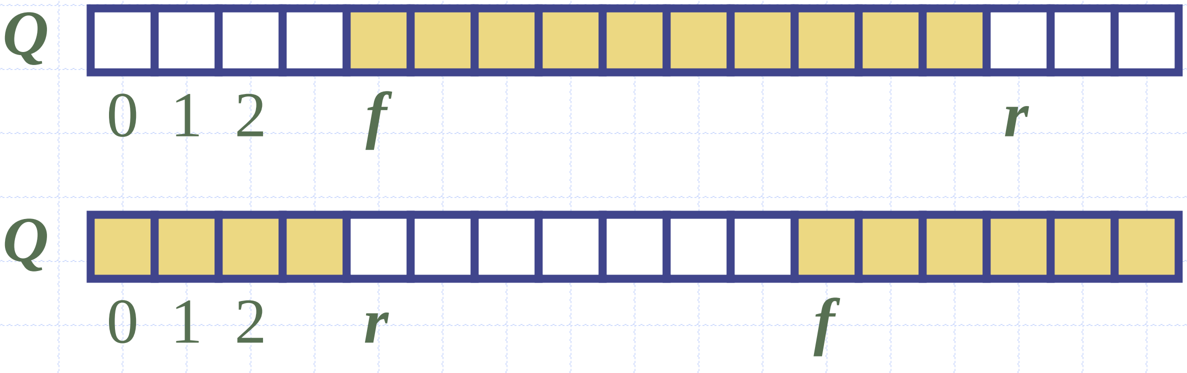
```
Algorithm enqueue(o)
  if size() = N – 1 then
    throw FullQueueException
  else
    Q[r] ← o
    r ← (r + 1) mod N
```



Queue Operations (Concept) (cont.)

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

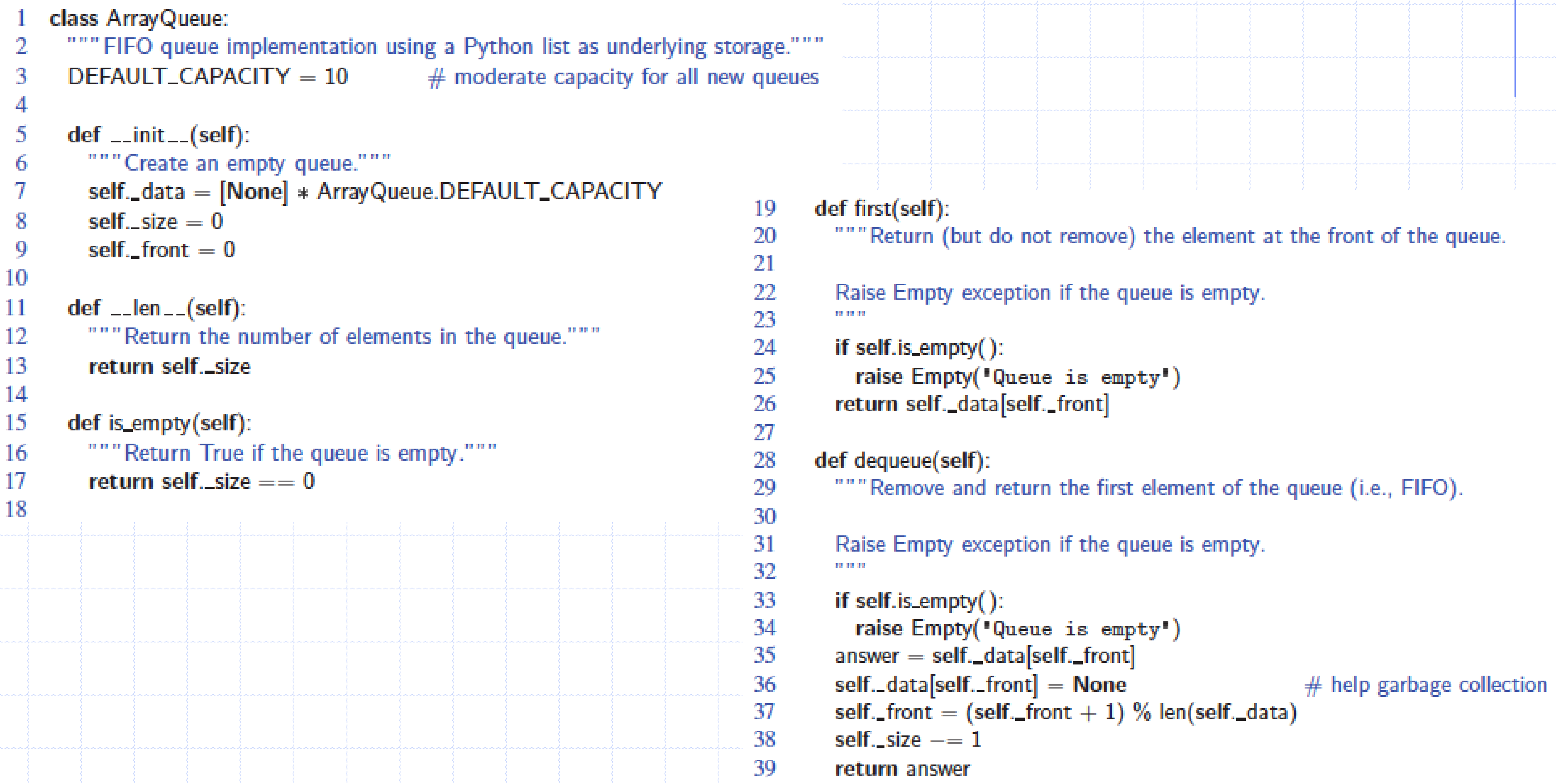
```
Algorithm dequeue()
    if isEmpty() then
        throw EmptyQueueException
    else
        $o \leftarrow Q[f]$
        $f \leftarrow (f + 1) \bmod N$
        return $o$
```



Queue in Python (text book)

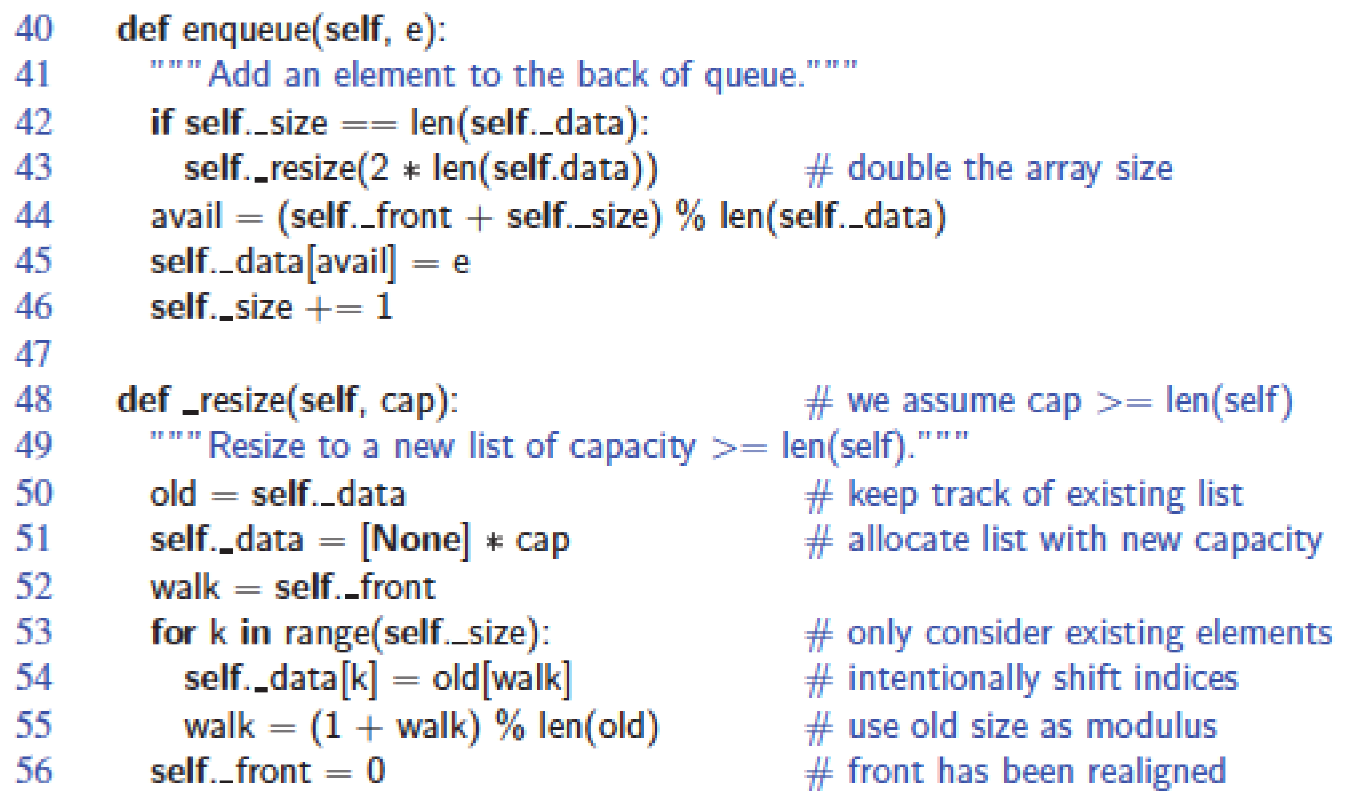
- Use the following three instance variables:
  - _data: is a reference to a list instance with a fixed capacity.
  - _size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - _front: is an integer that represents the index within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

Queue in Python (text book), Beginning



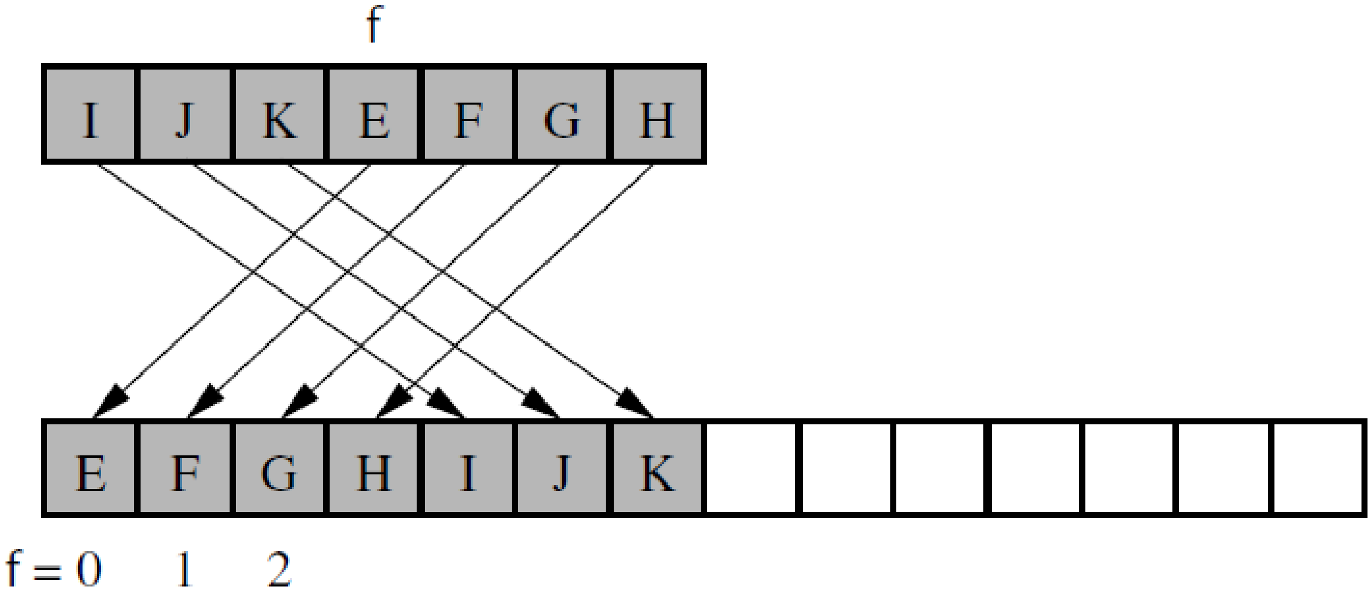
```python
class ArrayQueue:
  """FIFO queue implementation using a Python list as underlying storage."""
  DEFAULT_CAPACITY = 10          # moderate capacity for all new queues

  def __init__(self):
    """Create an empty queue."""
    self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
    self._size = 0
    self._front = 0

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0

  def first(self):
    """Return (but do not remove) the element at the front of the queue.

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._data[self._front]

  def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._data[self._front]
    self._data[self._front] = None                    # help garbage collection
    self._front = (self._front + 1) % len(self._data)
    self._size -= 1
    return answer
```

Queue in Python (text book), Continued



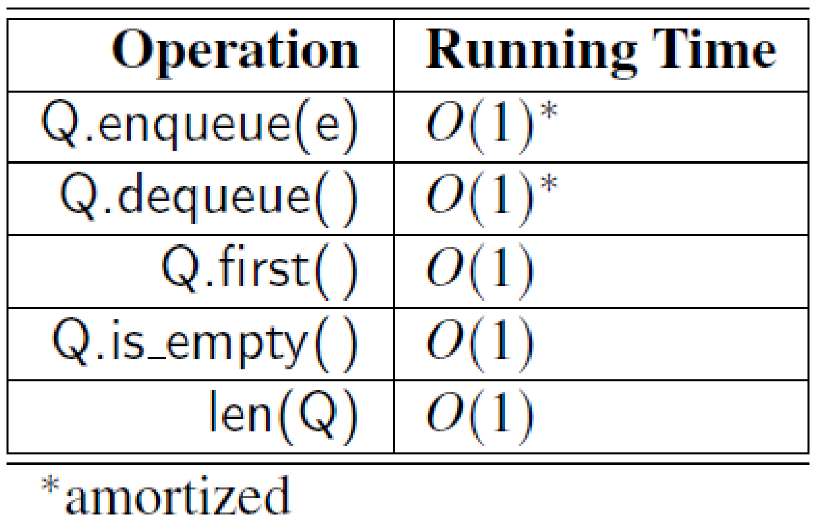
```
  def enqueue(self, e):
    """Add an element to the back of queue."""
    if self._size == len(self._data):
      self._resize(2 * len(self.data))          # double the array size
    avail = (self._front + self._size) % len(self._data)
    self._data[avail] = e
    self._size += 1

  def _resize(self, cap):                       # we assume cap >= len(self)
    """Resize to a new list of capacity >= len(self)."""
    old = self._data                            # keep track of existing list
    self._data = [None] * cap                   # allocate list with new capacity
    walk = self._front
    for k in range(self._size):                 # only consider existing elements
      self._data[k] = old[walk]                 # intentionally shift indices
      walk = (1 + walk) % len(old)              # use old size as modulus
    self._front = 0                             # front has been realigned
```

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



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

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).

Adding and Removing Elements (Queue in Python, Our Approach)

In enqueuer(self, e):

```
avail = (self._front + self._size) % len(self._data)
self._data[avail] = e
self._size += 1
```

In dequeuer(self)

```
answer = self._data[self._front]
self._data[self._front] = None
self._front = (self._front + 1) % len(self._data)
self._size −= 1
return answer
```

Analyzing the Array-Based Queue (Our Approach) What will be runtime complexity?

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

Analyzing the Array-Based Queue (Our Approach)

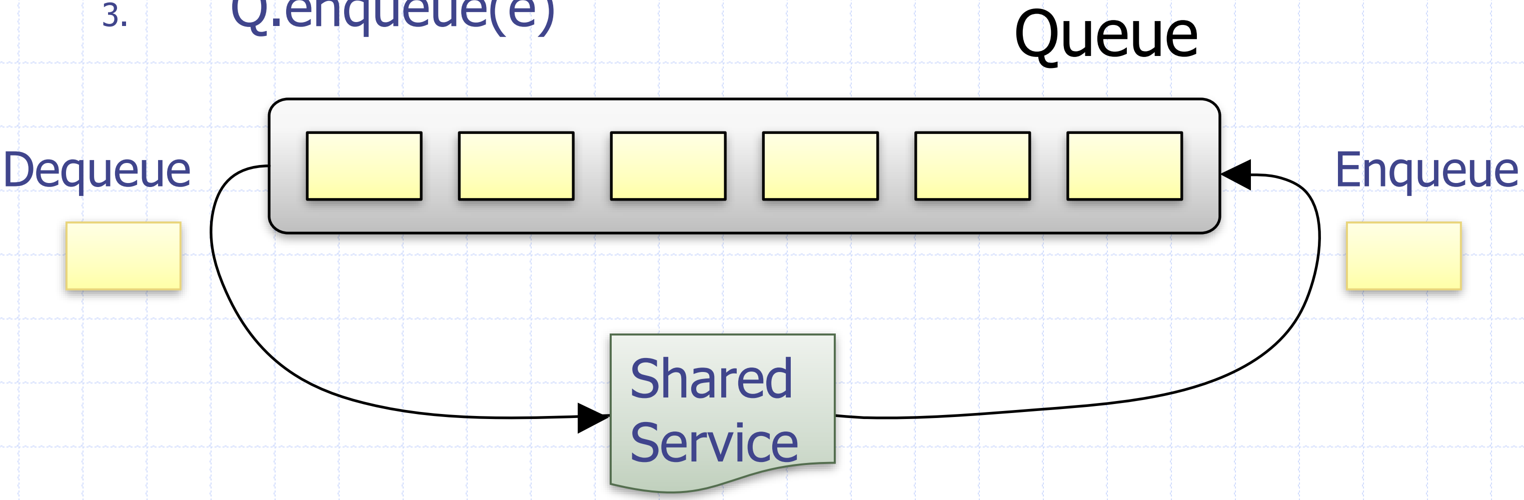
Operation	Running Time
Q.enqueue(e)	O(1)
Q.dequeue()	O(1)
Q.first()	O(1)
Q.is_empty()	O(1)
len(Q)	O(1)

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;

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)


Queue
Dequeue
Enqueue
Shared Service