Queues

- Collection of objects that are inserted and removed according to the first-in, first-out (FIFO)
- Elements can be inserted at anytime, but only the element that has been in the queue the longest can be next removed

Examples:

1) printers
2) Web server responding to requests.
3) Scheduling

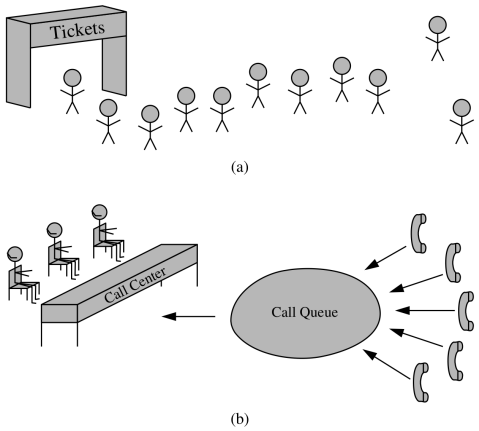
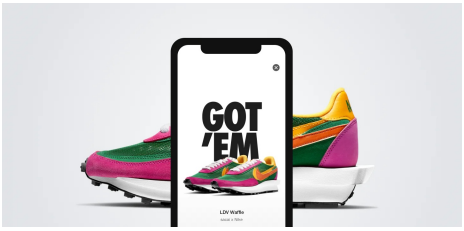


Figure 6.4: Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.



Nike SNKR App

~ for some drops

The Queue Abstract Data Type

~ The Queue abstract data type defines a collection that keeps objects in sequence, where element access and deletion are restricted to the first elements in the queue, and element insertion is restricted to the back of the sequence.

The queue abstract data type (ADT) supports two methods for a queue Q:

Operations:

Q.enqueue(e): Add element e to the back of queue Q.

Q.dequeue(): Remove and return the first element from queue Q; an error occurs if the queue is empty.

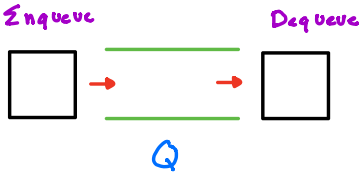
Other supporting Methods:

Q.first(): Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.

Q.is_empty(): Return True if queue Q does not contain any elements.

len(Q): Return the number of elements in queue Q; in Python, we implement this with the special method __len__.

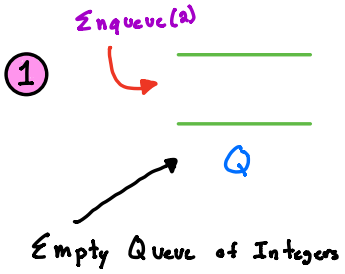
Constant time or O(1)



~ logically, a queue can be shown as a figure or container open from two sides.

~ An element can be inserted or Enqueued from one side and an element can be removed or de-queued from other side.

~ In queue, insertion and removal should happen from different sides.



Logic/Code

Enqueue (2) 1

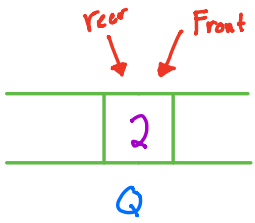
Enqueue (5) 2

Enqueue (3) 3

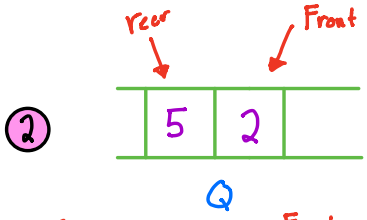
Dequeue() → 2 4

front() → 5

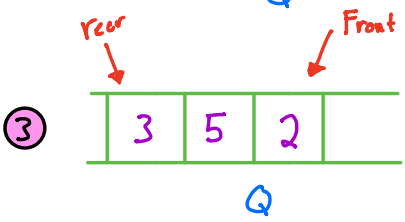
Is Empty → false



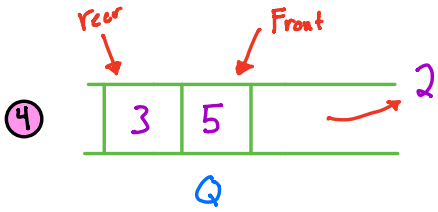
~ After this Enqueue, we have one element in the queue



~ Insert another integer,

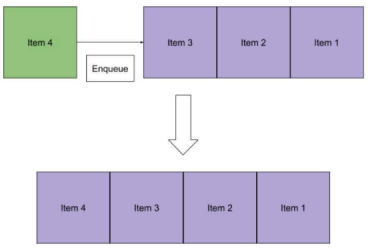


~ Insert another integer,

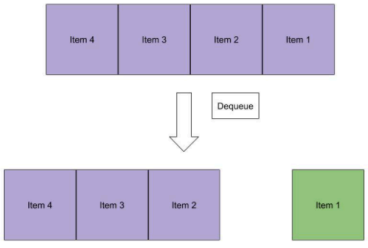


~ 2 will head out of the queue
~ Front moves over to the next item to be queued.

- `enqueue` - adds an element to the end of the queue:



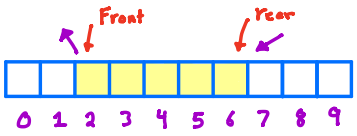
- `dequeue` - removes the element at the beginning of the queue:



How do we implement this in Python?

- Circular Array ✓
- linked lists

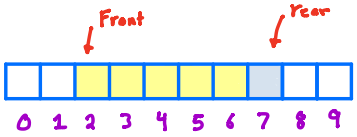
Array Implementation



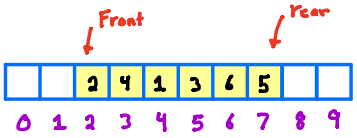
~ any order of front and rear

~ But element must be added from rear and removed from front

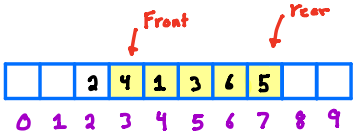
~ At any stage a segment of the array from an index marked as front till an index marked as rear is my Queue and the rest of positions in the array are free space, that can be used to expand the Queue.



~ add a new element towards the rear end, we can write the element to be inserted



~ Inserting the number 5



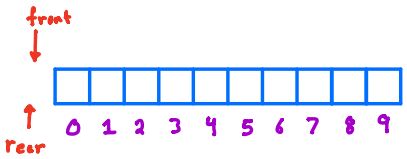
~ Removing an element from the Queue

~ By incrementing front we have discarded index 2 from the Queue

Writing Pseudo Code:

- is the array Empty?

```
Front = A[-1]
Rear = A[-2]
Is Empty():
  if A[Front] == -1 and A[Rear] == -1:
    return True
  else:
    return False
```



Q.enqueue(e): Add element e to the back of queue Q.
Q.dequeue( ): Remove and return the first element from queue Q; an error occurs if the queue is empty.

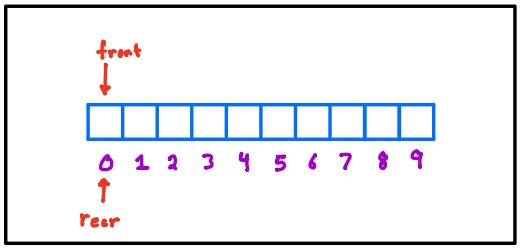
- Adding an Element to an Queue

```
Enqueue (x):

  if Is full():
      return
  else if Is Empty():
      front, rear = A[0]
  else:
      rear = rear +1

  A[rear] = x
```

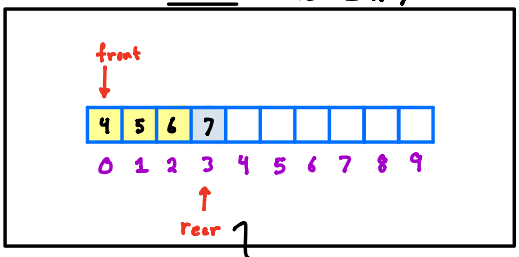
← If array is full return

← If it's not empty

else if:



else ~ not Empty or Full



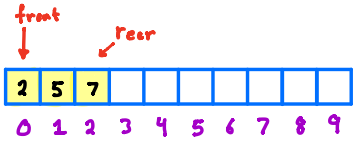
+1 for rear
then becomes the new value
inserted into the Queue

Lets Equeue some elements:

Enqueue (2)

Enqueue (5)

Enqueue (7)



Dequeue Suedo Code:

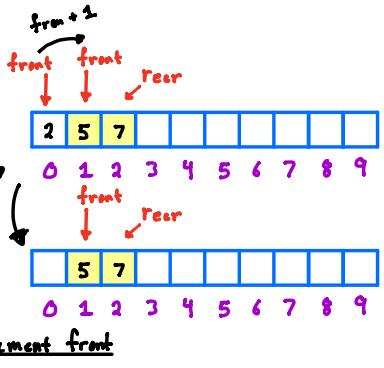
```
Dequeue():
    if IsEmpty():
        return
    else if front == rear:        ~ if Q has one element
        front, rear = A[-1]       ~ Our Dequeue will
                                    make the Queue empty
    else:
        front = front + 1         ~
```

Enqueue (2)

Enqueue (5)

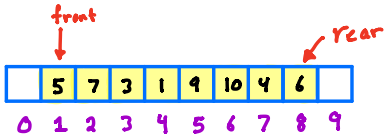
Enqueue (7)

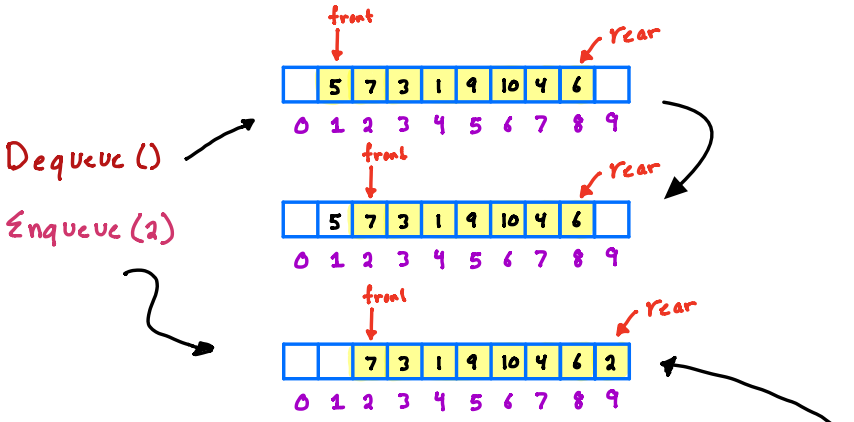
Dequeue () → It will increment front



- Front will be incremented, and our Value will be returned

Enqueue (3)
Enqueue (1)
Enqueue (9)
Enqueue (10)
Enqueue (4)
Enqueue (6)

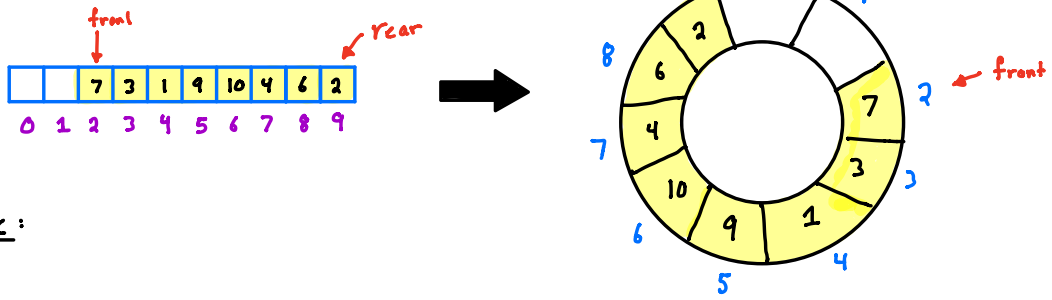




At this point, we cannot Enqueue an element anymore

- What will we do !?

Circular Array



Logic:

Current position = i

Next position = (i+1) % N ← Size of Array
→ Will give remainder

previous position = (i+N-1) % N

For the Example above:

i = N-1

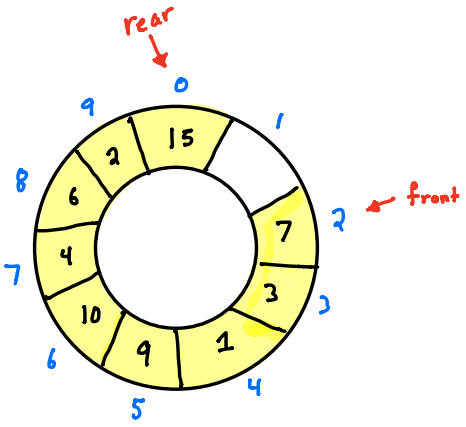
Next = N % N = 0

Modify Sudo Code:

```
IsEmpty():
  if front == -1 and rear == -1:
      return True
  else:
     return False
```

```
Enqueue(x):

   if (rear+1)%N == Front
       return

   else if IsEmpty():

         front, rear = 0
   else:
      rear = (rear+1) % N

   A[rear] = x
```

↖ Modulus gives us the remainder!



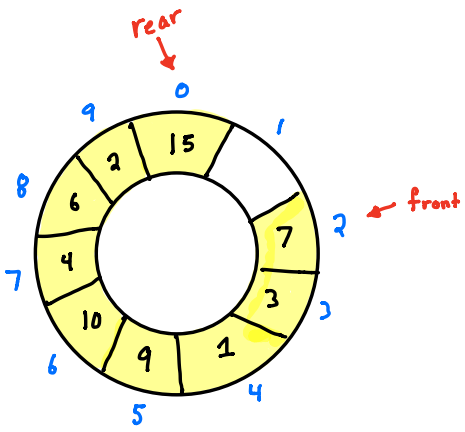
Enqueue(15)

rear = (rear+1) % N
= (9+1) % 10
rear = 0

A[rear] = x
A[0] = 15

```
Dequeue()

  if IsEmpty():
     return

  else if front == rear:
      front, rear = A[-1]        ~ Q is empty

  else:
     front = (front+1) % N
```

```
front()

   return A[front]
```

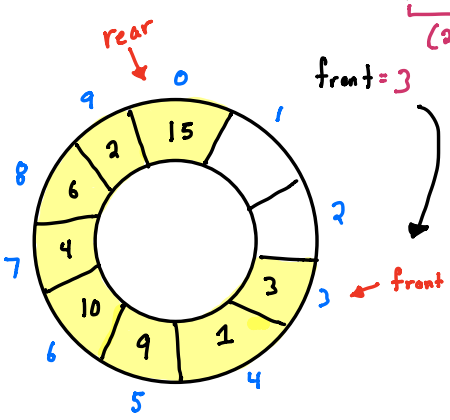


Dequeue()

front = (front + 1) % N

(2+1) % 10

front = 3



Example from book: A Python Implementation of a Queue

```
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
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
```

initial Queue of size 10

Instances:

- **_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).

Call our "Empty Class"

Answer is equal to the "front" index in our Queue list

The index of our recently removed item is now "none"

New front of Queue index

Size of Queue list is reduced by 1.

Should be: self._data

finds avaible spot in Queue, "the rear"

Places the element in the rear

Resizing the Queue

When enqueue is called at a time when the size of the queue equals the size of the underlying list, we rely on a standard technique of doubling the storage capacity of the underlying list. In this way, our approach is similar to the one used when we implemented a DynamicArray in Section 5.3.1.

However, more care is needed in the queue's _resize utility than was needed in the corresponding method of the DynamicArray class. After creating a temporary reference to the old list of values, we allocate a new list that is twice the size and copy references from the old list to the new list. While transferring the contents, we intentionally realign the front of the queue with index 0 in the new array, as shown in Figure 6.7. This realignment is not purely cosmetic. Since the modular arithmetic depends on the size of the array, our state would be flawed had we transferred each element to its same index in the new array.

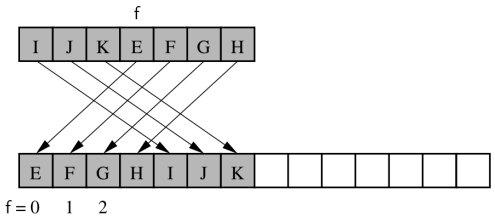


Figure 6.7: Resizing the queue, while realigning the front element with index 0.

How would be reduce the size of the array?

Insert this @ after line 38:

```
if 0 < self._size < len(self._data) // 4:
  self._resize(len(self._data) // 2)
```

Analyzing the Array-Based Queue Implementation

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

Table 6.3: Performance of an array-based implementation of a queue. The bounds for enqueue and dequeue are amortized due to the resizing of the array. The space usage is $O(n)$, where n is the current number of elements in the queue.

Example from Code above: