

Stacks, Queues, Deques

CPE111-Programming with Data Structures

Reference

- Rance D. Necaise. Data Structures and algorithms using python. Chapter2. John Wiley&Sons,Inc., 2011
- Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. Data Structures and Algorithms in python. Chapter5. John Wiley&Sons,Inc. 2013

STACK ADT

A *stack* is a data structure that stores a linear collection of items with access limited to a last-in first-out (LIFO) order. Adding and removing items is restricted to one end known as the *top* of the stack. An empty stack is one containing no items.

- **Stack():** Creates a new empty stack.
- **isEmpty():** Returns a Boolean value indicating if the stack is empty.
- **length ():** Returns the number of items in the stack.
- **pop():** Removes and returns the top item of the stack, if the stack is not empty. Items cannot be popped from an empty stack. The next item on the stack becomes the new top item.
- **peek():** Returns a reference to the item on top of a non-empty stack without removing it. Peeking, which cannot be done on an empty stack, does not modify the stack contents.
- **push(item):** Adds the given item to the top of the stack.

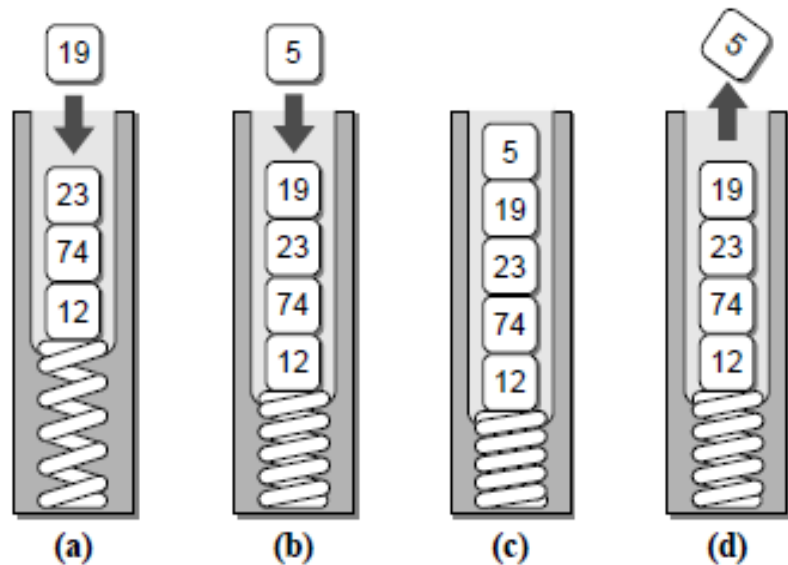
A **stack overflow** is an unwanted condition that occurs when a computer program tries to use more memory space than the call stack has available.



Implementing a Stack with a Python List

Implementing a stack with a python list is the easiest way. The end of the list is used as the top. Other python list's methods are adapted to stack's method as in table.


`S.push(19)` `S.push(5)` `e=S.peek()` `e=S.pop()`



Stack Method	Implement with Python List
<code>S.push(item)</code>	<code>L.append(item)</code>
<code>S.pop()</code>	<code>L.pop()</code>
<code>S.peek()</code>	<code>L.pop(-1)</code>
<code>S.isEmpty()</code>	<code>len(L) == 0</code>
<code>len(S)</code>	<code>len(L)</code>

```
1  # Implementation of the Stack ADT using a Python list.
2  class Stack :
3      # Creates an empty stack.
4      def __init__( self ):
5          self._theItems = list()
6
7      # Returns True if the stack is empty or False otherwise.
8      def isEmpty( self ):
9          return len( self ) == 0
10
11     # Returns the number of items in the stack.
12     def __len__ ( self ):
13         return len( self._theItems )
14
15     # Returns the top item on the stack without removing it.
16     def peek( self ):
17         assert not self.isEmpty(), "Cannot peek at an empty stack"
18         return self._theItems[-1]
```

```
20     # Removes and returns the top item on the stack.
21     def pop( self ):
22         assert not self.isEmpty(), "Cannot pop from an empty stack"
23         return XXXXXXXXXX
24
25     # Push an item onto the top of the stack.
26     def push( self, item ):
27         XXXXXXXXXX
```





you can guess...

QUEUE ADT

A queue is a data structure that a linear collection of items in which access is restricted to a first-in first-out (FIFO) basis. New items are inserted at the back and existing items are removed from the front. The items are maintained in the order in which they are added to the structure.

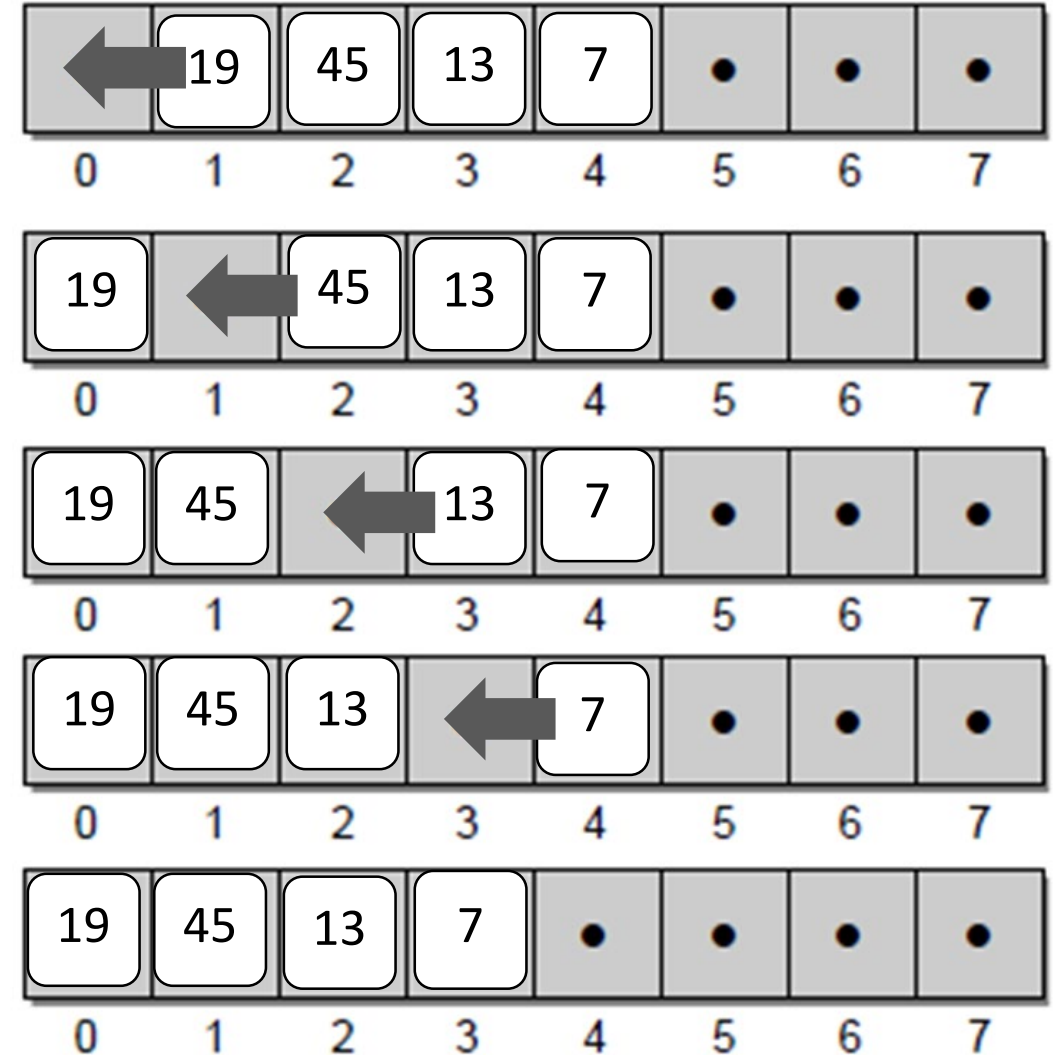
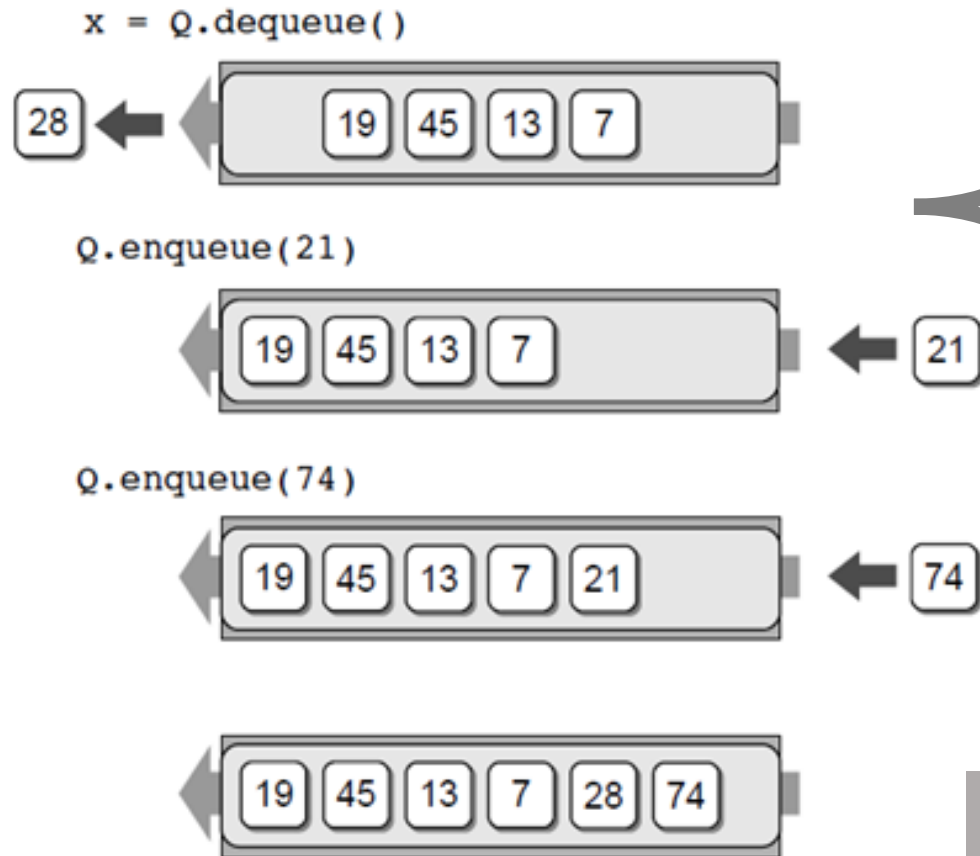
- **Queue():** Creates a new empty queue, which is a queue containing no items.
- **isEmpty():** Returns a Boolean value indicating whether the queue is empty.
- **length ():** Returns the number of items currently in the queue.
- **enqueue(item):** Adds the given item to the back of the queue.
- **dequeue():** Removes and returns the front item from the queue. An item cannot be dequeued from an empty queue.

Implementation of the Queue ADT using a python list

```
1  # Implementation of the Queue ADT using a Python list.
2  class Queue :
3      # Creates an empty queue.
4      def __init__( self ):
5          self._qList = list()
6
7      # Returns True if the queue is empty.
8      def isEmpty( self ):
9          return len( self ) == 0
10
11     # Returns the number of items in the queue.
12     def __len__( self ):
13         return len( self._qList )
14
15     # Adds the given item to the queue.
16     def enqueue( self, item ):
17         
18
19     # Removes and returns the first item in the queue.
20     def dequeue( self ):
21         assert not self.isEmpty(), "Cannot dequeue from an empty queue."
22         return 
```

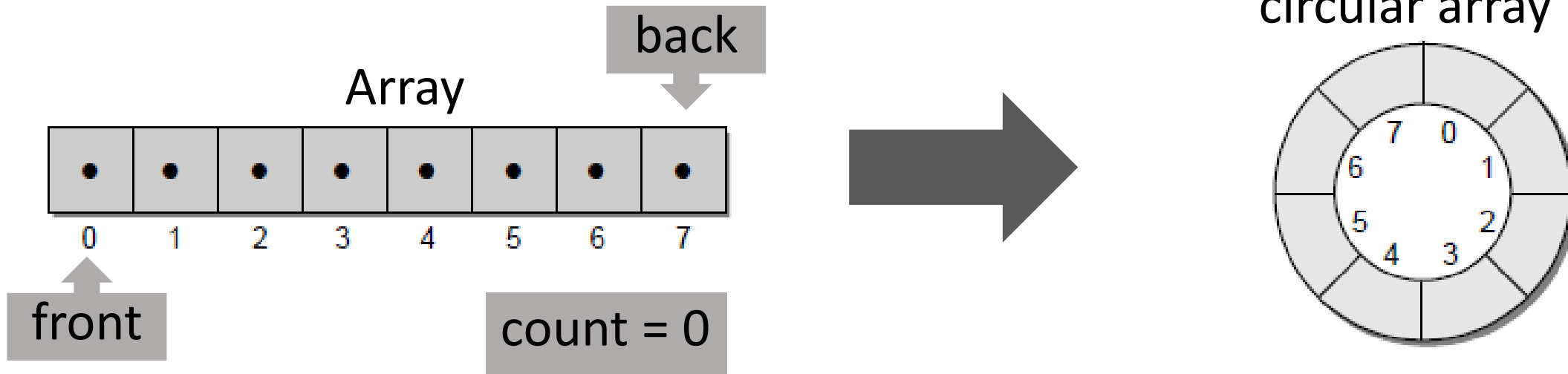


```
Q = Queue()  
Q.enqueue( 28 )  
Q.enqueue( 19 )  
Q.enqueue( 45 )  
Q.enqueue( 13 )  
Q.enqueue( 7 )
```



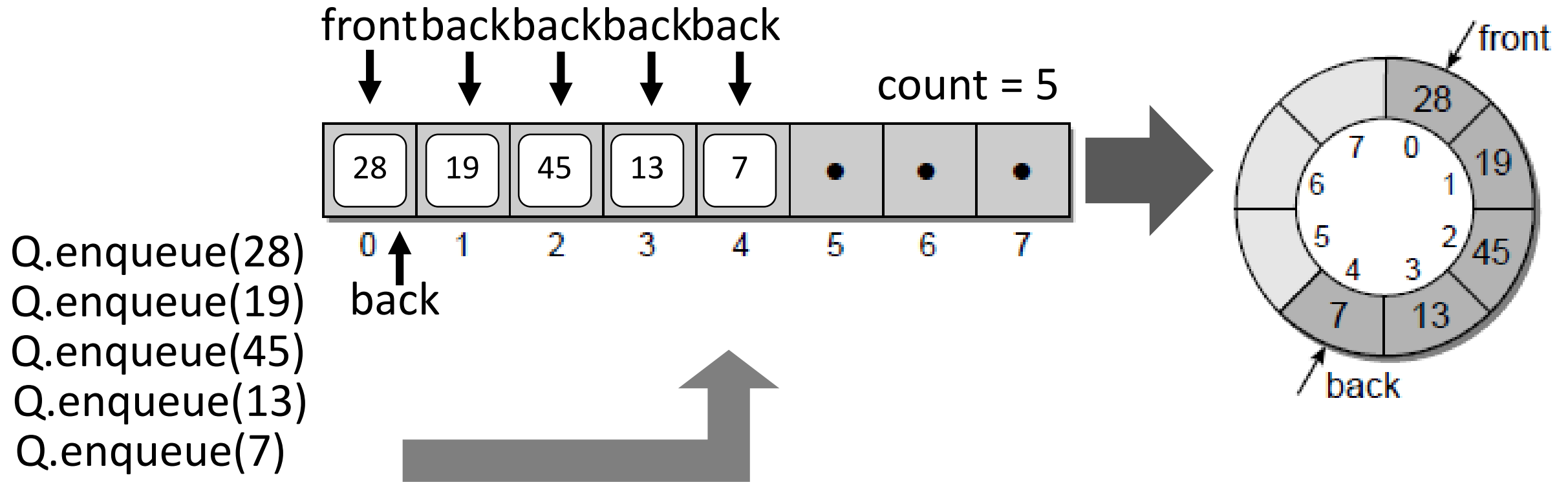
linear time operation penalty for dequeue()

Implementation of the Queue ADT using a circular array



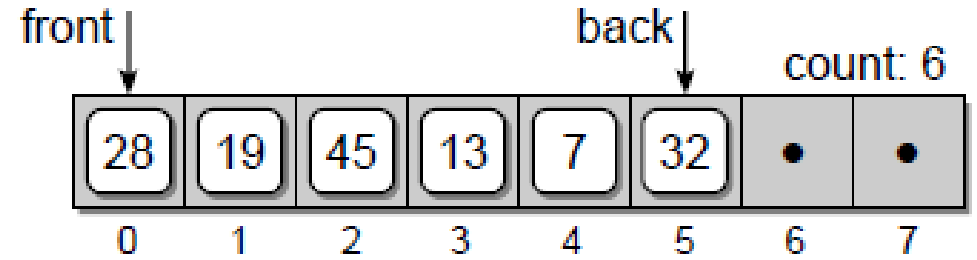
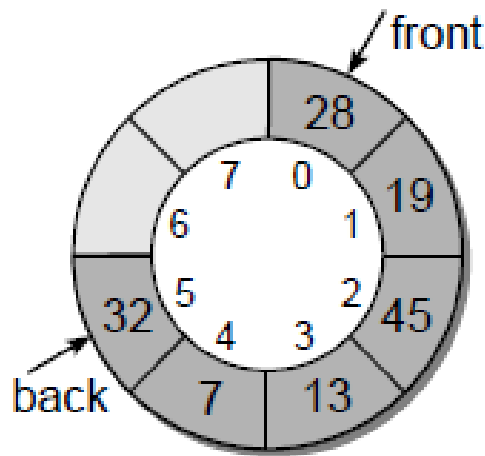
Q = Queue(8)

```
def __init__( self, maxSize ) :  
    self._count = 0  
    self._front = 0  
    self._back = maxSize - 1  
    self._qArray = Array( maxSize )
```

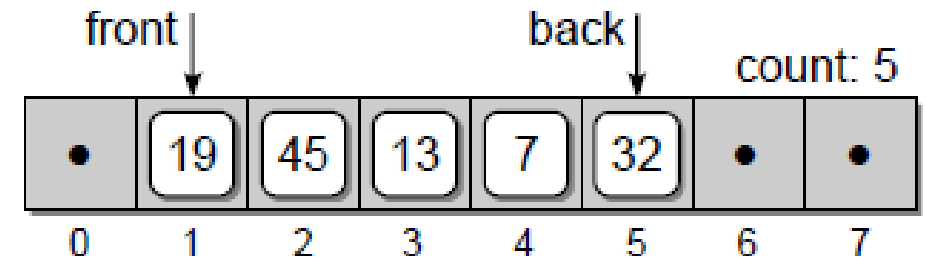
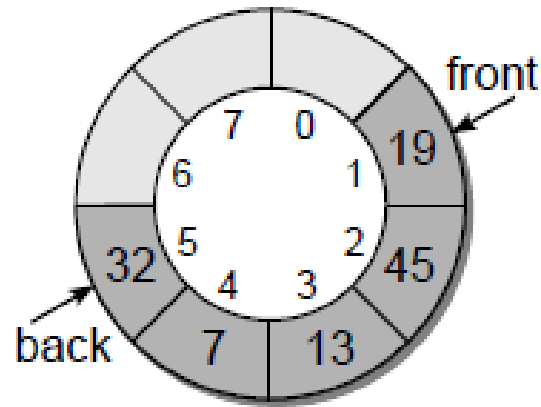


```
def enqueue( self, item ):
    assert not self.isFull(), "Cannot enqueue to a full queue."
    maxSize = len(self._qArray)
    self._back = (self._back + 1) % maxSize
    self._qArray[self._back] = item
    self._count += 1
```

Q.enqueue(32)

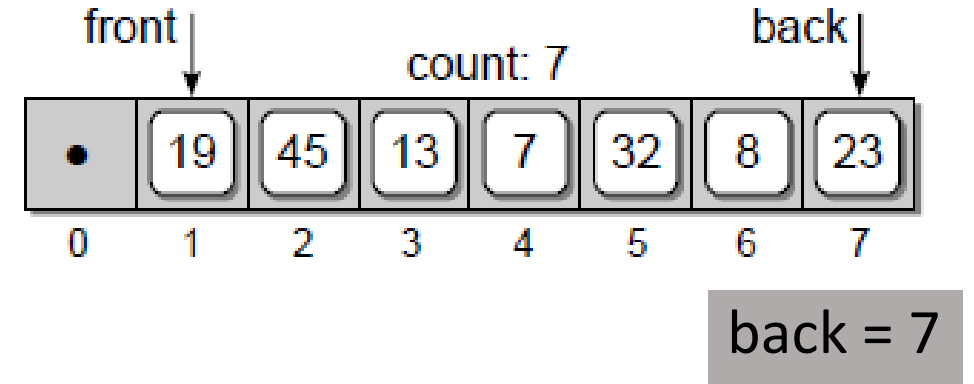
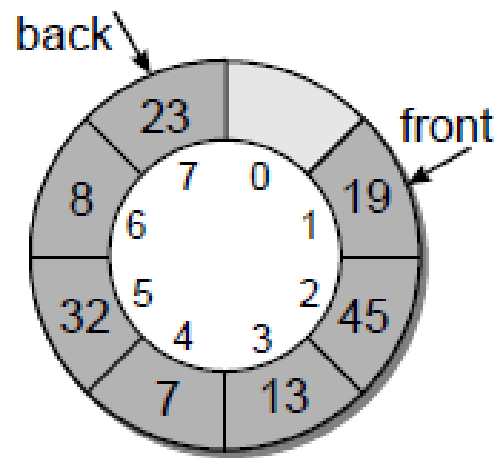


q = Q.dequeue()

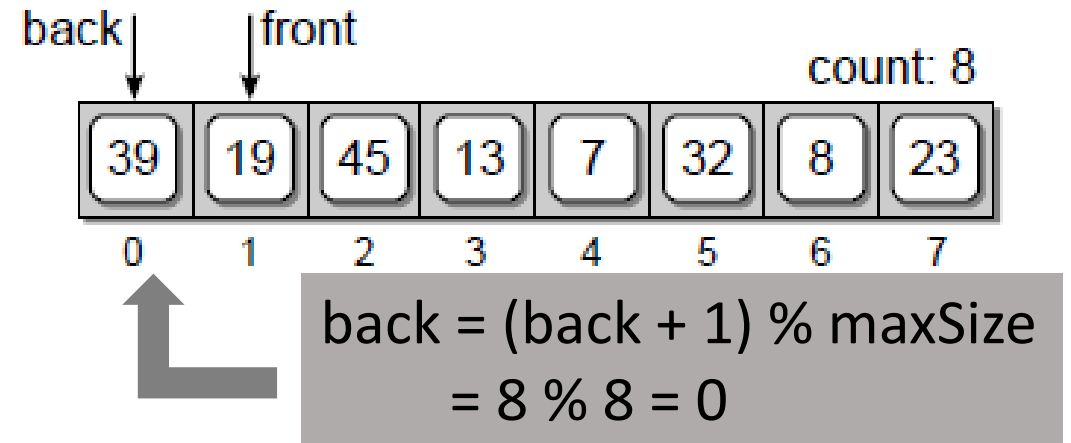
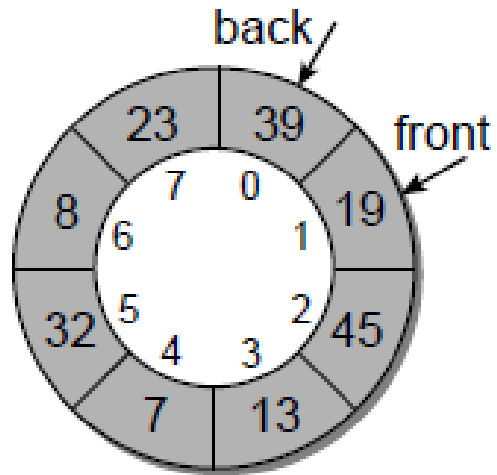


```
def dequeue( self ):  
    assert not self.isEmpty(), "Cannot dequeue from an empty queue."  
    item = self._qArray[ self._front ]  
    maxSize = len(self._qArray)  
    self._front = (self._front + 1) % maxSize  
    self._count -= 1  
    return item
```

Q.enqueue(8)
Q.enqueue(23)



Q.enqueue(39)



```
def enqueue(self, item):  
    assert not self.isFull(), "Cannot enqueue to a full queue."  
    maxSize = len(self._qArray)  
    self._back = (self._back + 1) % maxSize  
    self._qArray[self._back] = item  
    self._count += 1
```

Run Time Analysis

The `circular array` implementation provides a more efficient solution than the `Python list`. The operations all have a worst case **time-complexity of $O(1)$** since the array items never have to be shifted. But the circular array does introduce the drawback of working with a **maximum-capacity queue**. Even with this limitation, it is well suited for some applications.

Priority Queues

A priority queue is simply an extended version of the basic queue with the exception that a *priority* p must be assigned to each item at the time it is enqueued. There are two basic types of priority queues: bounded and unbounded.

- The bounded priority queue assumes a small limited range of p priorities over the interval of integers $[0 \dots p)$.
- The unbounded priority queue places no limit on the range of integer values that can be used as priorities.

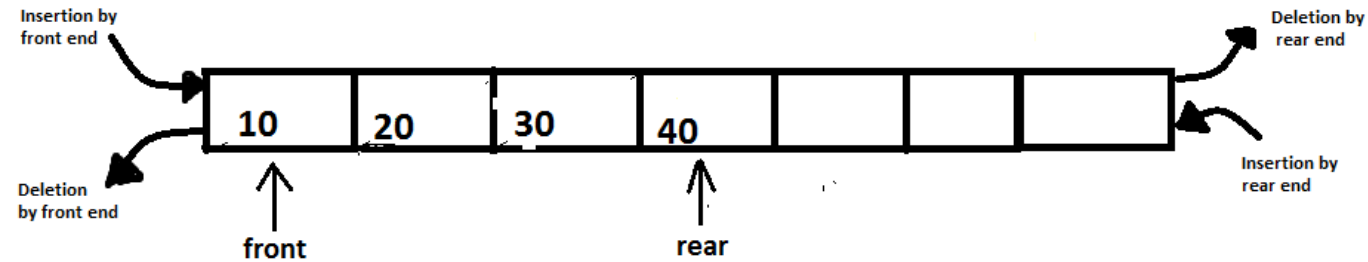
Priority Queue ADT

A **priority queue** is a queue in which each item is assigned a priority and items with a higher priority are removed before those with a lower priority, irrespective of when they were added. Integer values are used for the priorities with **a smaller integer value having a higher priority**.

- ***bounded priority queue***: the priorities to the integer values between 0 and a predefined upper limit p
- ***unbounded priority queue***: places no limits on the range of priorities.

- **PriorityQueue()**: Creates a new empty unbounded priority queue.
- **BPriorityQueue(numLevels)**: Creates a new empty bounded priority queue with priority levels in the range from 0 to $numLevels - 1$.
- **isEmpty()**: Returns a boolean value indicating whether the queue is empty.
- **length ()**: Returns the number of items currently in the queue.
- **enqueue(item, priority)**: Adds the given item to the queue by inserting it in the proper position based on the given priority. The priority value must be within the legal range when using a bounded priority queue.
- **dequeue()**: Removes and returns the front item from the queue, which is the item with *the highest priority*. The associated priority value is discarded. If two items have the same priority, then those items are removed in a FIFO order. An item cannot be dequeued from an empty queue.

Double-Ended Queue ADT



A **doubled-ended queue** is a queue-like data structure that supports insertion and deletion at both the front and the back of the queue.

- **Deque():** Create a new empty deque, which no item with in.
- **AddFirst(item):** Add an item to the front of deque.
- **AddRear(item):** Add an item to the back of deque.
- **DeleteFirst():** Remove and return the first item from deque; an error occurs if the deque is empty.
- **DeleteRear ():** Remove and return the last item from deque; an error occurs if the deque is empty.
- **First():** Return (but do not remove) the first item of deque; an error occurs if the deque is empty.
- **Rear():** Return (but do not remove) the last item of deque; an error occurs if the deque is empty.
- **isEmpty():** Return True if deque does not contain any items.
- **length():** Return the number of items in deque.

Operation	Return Value	Deque
D = Deque()	-	[]
D.AddRear(5)	-	[5]
D.AddFirst(3)	-	[3, 5]
D.AddFirst(7)	-	[7, 3, 5]
d = D.First()	7	[7, 3, 5]
d = D.DeleteRear()	5	[7, 3]
l = length(D)	2	[7, 3]
d = D.DeleteRear()	3	[7]
d = D.DeleteRear()	7	[]
D.AddFirst(6)	-	[6]
d = D.Rear()	6	[6]
D.AddFirst(8)	-	[8, 6]
D.isEmpty()	False	[8, 6]
d = D.Rear()	6	[8, 6]

Deque can be implemented in python using the module “**collections**“. Deque is preferred over **list** in the cases where we need quicker append and pop operations from both the ends of container, as deque provides an **O(1)** time complexity for append and pop operations as compared to list which provides O(n) time complexity.

<https://www.geeksforgeeks.org/deque-in-python/>

```
import collections
```

```
# initializing deque
de = collections.deque([1,2,3])
```