CHAPTER

5

QUEUES

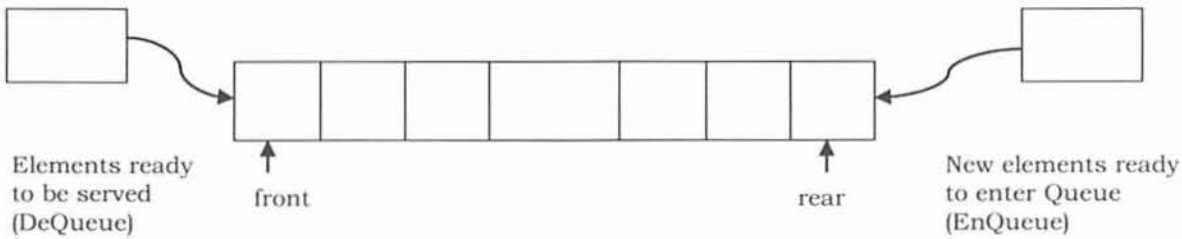
5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Definition: A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called *EnQueue*, and when an element is removed from the queue, the concept is called *DeQueue*.

DeQueueing an empty queue is called *underflow* and *EnQueuing* an element in a full queue is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.



5.2 How are Queues Used

The concept of a queue can be explained by observing a line at a reservation counter. When we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

As this happens, the next person will come at the head of the line, will exit the queue and will be served. As each person at the head of the line keeps exiting the queue, we move towards the head of the line. Finally we will reach the head of the line and we will exit the queue and be served. This behavior is very useful in cases where there is a need to maintain the order of arrival.

5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in the queue must follow the FIFO scheme. For simplicity we assume the elements are integers.

Main Queue Operations

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary Queue Operations

- int Front(): Returns the element at the front without removing it
- int QueueSize(): Returns the number of elements stored in the queue
- int IsEmptyQueue(): Indicates whether no elements are stored in the queue or not

5.4 Exceptions

Similar to other ADTs, executing *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and executing *EnQueue* on a full queue throws a "*Full Queue Exception*".

5.5 Applications

Following are the some of the applications that use queues.

Direct Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

5.6 Implementation

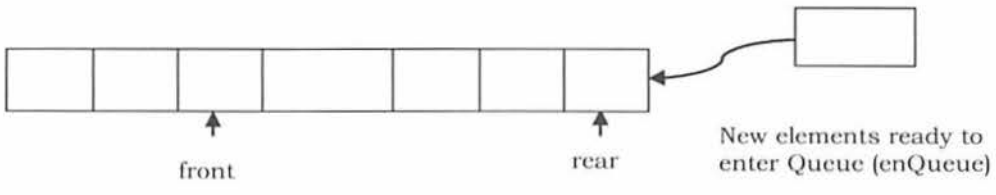
There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.

- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked list implementation

Why Circular Arrays?

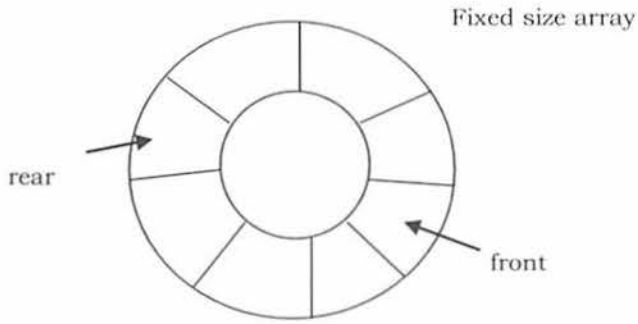
First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at the other end. After performing some insertions and deletions the process becomes easy to understand.

In the example shown below, it can be seen clearly that the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat the last element and the first array elements as contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.



Note: The simple circular array and dynamic circular array implementations are very similar to stack array implementations. Refer to *Stacks* chapter for analysis of these implementations.

Simple Circular Array Implementation



This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of the start element and end element. Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue.

The array storing the queue elements may become full. An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from an empty queue it will throw *empty queue exception*.

Note: Initially, both front and rear points to -1 which indicates that the queue is empty.

```
class Queue(object):
 def __init__(self, limit = 5):
        self.que = []
        self.limit = limit
        self.front = None
        self.rear = None
        self.size = 0
 def isEmpty(self):
        return self.size <= 0
 def enQueue(self, item):
        if self.size >= self.limit:
                print 'Queue Overflow!'
                return
        else:
                self.que.append(item)

        if self.front is None:
                self.front = self.rear = 0
        else:
                self.rear = self.size
        self.size += 1
        print 'Queue after enQueue',self.que
 def deQueue(self):
        if self.size <= 0:
                print 'Queue Underflow!'
                return 0
        else:
                self.que.pop(0)
                self.size -= 1
                if self.size == 0:
                        self.front = self.rear = None
                else:
                        self.rear = self.size-1
                print 'Queue after deQueue',self.que
 def queueRear(self):
        if self.rear is None:
                print "Sorry, the queue is empty!"
```

```
                raise IndexError
            return self.que[self.rear]
    def queueFront(self):
            if self.front is None:
                print "Sorry, the queue is empty!"
                raise IndexError
            return self.que[self.front]
    def size(self):
            return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```

Performance and Limitations

Performance: Let n be the number of elements in the queue:

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Limitations: The maximum size of the queue must be defined as prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

Dynamic Circular Array Implementation

```
class Queue(object):
  def __init__(self, limit = 5):
            self.que = []
            self.limit = limit
            self.front = None
            self.rear = None
            self.size = 0
    def isEmpty(self):
            return self.size <= 0
    def enQueue(self, item):
            if self.size >= self.limit:
                    self.resize()
            self.que.append(item)
            if self.front is None:
                    self.front = self.rear = 0
            else:
                    self.rear = self.size
            self.size += 1
```

```
            print 'Queue after enQueue',self.que
    def deQueue(self):
            if self.size <= 0:
                    print 'Queue Underflow!'
                    return 0
            else:
                    self.que.pop(0)
                    self.size -= 1
                    if self.size == 0:
                            self.front = self.rear = None
                    else:
                            self.rear = self.size-1
                    print 'Queue after deQueue',self.que
    def queueRear(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.que[self.rear]
    def queueFront(self):
            if self.front is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.que[self.front]
    def size(self):
            return self.size
    def resize(self):
            newQue = list(self.que)
            self.limit = 2*self.limit
            self.que = newQue
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("four")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("five")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("six")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```

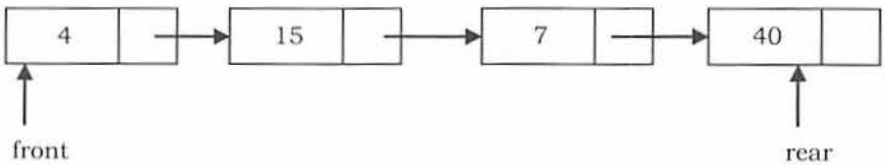
Performance

Let n be the number of elements in the queue.

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Linked List Implementation

Another way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting an element at the end of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.




```
#Node of a Singly Linked List
class Node:
 #constructor
 def __init__(self, data=None, next=None):
            self.data = data
            self.last = None
            self.next = next
 #method for setting the data field of the node
 def setData(self,data):
            self.data = data
 #method for getting the data field of the node
 def getData(self):
            return self.data
 #method for setting the next field of the node
 def setNext(self,next):
            self.next = next
 #method for getting the next field of the node
 def getNext(self):
            return self.next
 #method for setting the last field of the node
 def setLast(self,last):
            self.last = last
 #method for getting the last field of the node
 def getLast(self):
            return self.last
 #returns true if the node points to another node
 def hasNext(self):
            return self.next != None
class Queue(object):
 def __init__(self, data=None):
            self.front = None
            self.rear = None
            self.size = 0
 def enQueue(self, data):
            self.lastNode = self.front
            self.front = Node(data, self.front)
            if self.lastNode:
                        self.lastNode.setLast(self.front)
            if self.rear is None:
                        self.rear = self.front
```

```
            self.size += 1
    def queueRear(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.rear.getData()
    def queueFront(self):
            if self.front is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.front.getData()
    def deQueue(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            result = self.rear.getData()
            self.rear = self.rear.last
            self.size -= 1
            return result
    def size(self):
            return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
print "Dequeuing: "+que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```

Performance

Let n be the number of elements in the queue, then

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of DeleteQueue()	O(1)

Comparison of Implementations

Note: Comparison is very similar to stack implementations and *Stacks* chapter.

5.7 Queues: Problems & Solutions

Problem-1 Give an algorithm for reversing a queue Q. To access the queue, we are only allowed to use the methods of queue ADT.

Solution:

```
class Stack(object):
    def __init__(self, limit = 10):
            self.stk = []
            self.limit = limit
    def isEmpty(self):
```

```
            return len(self.stk) <= 0
    def push(self, item):
            if len(self.stk) >= self.limit:
                    print 'Stack Overflow!'
            else:
                    self.stk.append(item)
            print 'Stack after Push',self.stk
    def pop(self):
            if len(self.stk) <= 0:
                    print 'Stack Underflow!'
                    return 0
            else:
                    return self.stk.pop()
    def peek(self):
            if len(self.stk) <= 0:
                    print 'Stack Underflow!'
                    return 0
            else:
                    return self.stk[-1]
    def size(self):
            return len(self.stk)
#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self, data=None, next=None):
            self.data = data
            self.last = None
            self.next = next
    #method for setting the data field of the node
    def setData(self,data):
            self.data = data
    #method for getting the data field of the node
    def getData(self):
            return self.data
    #method for setting the next field of the node
    def setNext(self,next):
            self.next = next
    #method for getting the next field of the node
    def getNext(self):
            return self.next
    #method for setting the last field of the node
    def setLast(self,last):
            self.last = last
    #method for getting the last field of the node
    def getLast(self):
            return self.last
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
class Queue(object):
    def __init__(self, data=None):
            self.front = None
            self.rear = None
            self.size = 0
    def enQueue(self, data):
            self.lastNode = self.front
            self.front = Node(data, self.front)
            if self.lastNode:
                    self.lastNode.setLast(self.front)
            if self.rear is None:
                    self.rear = self.front
```

```
            self.size += 1
    def queueRear(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.rear.getData()
    def queueFront(self):
            if self.front is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.front.getData()
    def deQueue(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            result = self.rear.getData()
            self.rear = self.rear.last
            self.size -= 1
            return result
    def size(self):
            return self.size
    def isEmpty(self):
            return self.size == 0
que = Queue()
for i in xrange(5):
  que.enQueue(i)
# suppose your have a Queue my_queue
aux_stack = Stack()
while not que.isEmpty():
  aux_stack.push(que.deQueue())
while not aux_stack.isEmpty():
  que.enQueue(aux_stack.pop())
for i in xrange(5):
  print que.deQueue()
```

Time Complexity: O(n).

Problem-2 How can you implement a queue using two stacks?

Solution: The key insight is that a stack reverses order (while a queue doesn't). A sequence of elements pushed on a stack comes back in reversed order when popped. Consequently, two stacks chained together will return elements in the same order, since reversed order reversed again is original order.

Let S1 and S2 be the two stacks to be used in the implementation of queue. All we have to do is to define the EnQueue and DeQueue operations for the queue.

EnQueue Algorithm

- Just push on to stack S1

Time Complexity: O(1).

DeQueue Algorithm

- If stack S2 is not empty then pop from S2 and return that element.
- If stack is empty, then transfer all elements from S1 to S2 and pop the top element from S2 and return that popped element [we can optimize the code a little by transferring only $n-1$ elements from S1 to S2 and pop the n^{th} element from S1 and return that popped element].
- If stack S1 is also empty then throw error.

Time Complexity: From the algorithm, if the stack S2 is not empty then the complexity is O(1). If the stack S2 is empty, then we need to transfer the elements from S1 to S2. But if we carefully observe, the number of

transferred elements and the number of popped elements from S2 are equal. Due to this the average complexity of pop operation in this case is O(1). The amortized complexity of pop operation is O(1).

```
class Queue(object):
    def __init__(self):
        self.S1 = []
        self.S2 = []
    def enqueue(self,element):
        self.S1.append(element)
    def dequeue(self):
        if not self.S2:
            while self.S1:
                self.S2.append(self.S1.pop())
        return self.S2.pop()
q = Queue()
for i in xrange(5):
    q.enqueue(i)
for i in xrange(5):
    print q.dequeue()
```

Problem-3 Show how you can efficiently implement one stack using two queues. Analyze the running time of the stack operations.

Solution: Let Q1 and Q2 be the two queues to be used in the implementation of stack. All we have to do is to define the push and pop operations for the stack.

In the algorithms below, we make sure that one queue is always empty.

Push Operation Algorithm: Insert the element in whichever queue is not empty.

- Check whether queue Q1 is empty or not. If Q1 is empty then Enqueue the element into Q2.
- Otherwise EnQueue the element into Q1.

Time Complexity: O(1).

Pop Operation Algorithm: Transfer $n-1$ elements to the other queue and delete last from queue for performing pop operation.

- If queue Q1 is not empty then transfer $n-1$ elements from Q1 to Q2 and then, DeQueue the last element of Q1 and return it.
- If queue Q2 is not empty then transfer $n-1$ elements from Q2 to Q1 and then, DeQueue the last element of Q2 and return it.

Time Complexity: Running time of pop operation is $O(n)$ as each time pop is called, we are transferring all the elements from one queue to the other.

```
class Queue(object):
    def __init__(self):
            self.queue=[]
    def isEmpty(self):
            return self.queue==[]
    def enqueue(self,x):
            self.queue.append(x)
    def dequeue(self):
            if self.queue:
                    a=self.queue[0]
                    self.queue.remove(a)
                    return a
            else:
                    raise IndexError,'queue is empty'
    def size(self):
            return len(self.queue)
class Stack(object):
    def __init__(self):
            self.Q1=Queue()
```

```
        self.Q2=Queue()
    def isEmpty(self):
        return self.Q1.isEmpty() and self.Q2.isEmpty()
    def push(self,item):
        if self.Q2.isEmpty():
            self.Q1.enqueue(item)
        else:
            self.Q2.enqueue(item)
    def pop(self):
        if self.isEmpty():
            raise IndexError,'stack is empty'
        elif self.Q2.isEmpty():
            while not self.Q1.isEmpty():
                cur=self.Q1.dequeue()
                if self.Q1.isEmpty():
                    return cur
                self.Q2.enqueue(cur)
        else:
            while not self.Q2.isEmpty():
                cur=self.Q2.dequeue()
                if self.Q2.isEmpty():
                    return cur
                self.Q1.enqueue(cur)
stk = Stack()
for i in xrange(5):
  stk.push(i)
for i in xrange(5):
  print stk.pop()
```

Problem-4 **Maximum sum in sliding window:** Given array A[] with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width w. **Output**: An array B[], B[i] is the maximum value from A[i] to A[i+w-1]. **Requirement**: Find a good optimal way to get B[i]

Solution: This problem can be solved with doubly ended queue (which supports insertion and deletion at both ends). Refer *Priority Queues* chapter for algorithms.

Problem-5 Given a queue Q containing n elements, transfer these items on to a stack S (initially empty) so that front element of Q appears at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue, and push and pop operations for the stack, outline an efficient O(n) algorithm to accomplish the above task, using only a constant amount of additional storage.

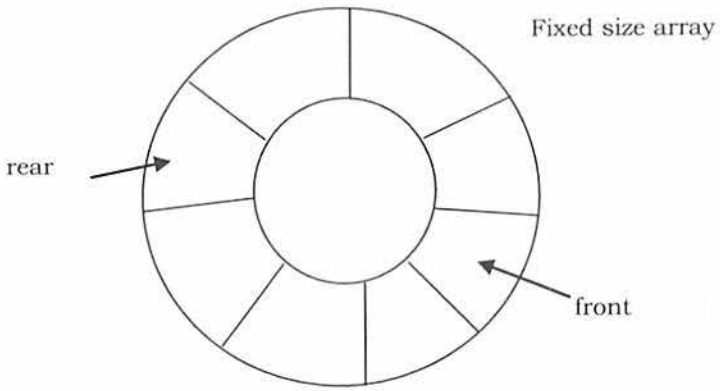
Solution: Assume the elements of queue Q are $a_1, a_2 \ldots a_n$. Dequeuing all elements and pushing them onto the stack will result in a stack with a_n at the top and a_1 at the bottom. This is done in O(n) time as dequeue and each push require constant time per operation. The queue is now empty. By popping all elements and pushing them on the queue we will get a_1 at the top of the stack. This is done again in O(n) time.

As in big-oh arithmetic we can ignore constant factors. The process is carried out in O(n) time. The amount of additional storage needed here has to be big enough to temporarily hold one item.

Problem-6 A queue is set up in a circular array A[0..n - 1] with front and rear defined as usual. Assume that $n - 1$ locations in the array are available for storing the elements (with the other element being used to

detect full/empty condition). Give a formula for the number of elements in the queue in terms of *rear*, *front*, and n.

Solution: Consider the following figure to get a clear idea of the queue.



- Rear of the queue is somewhere clockwise from the front.
- To enqueue an element, we move *rear* one position clockwise and write the element in that position.
- To dequeue, we simply move *front* one position clockwise.
- Queue migrates in a clockwise direction as we enqueue and dequeue.
- Emptiness and fullness to be checked carefully.
- Analyze the possible situations (make some drawings to see where *front* and *rear* are when the queue is empty, and partially and totally filled). We will get this:

$$Number\ Of\ Elements = \begin{cases} rear - front + 1 & \text{if rear} == \text{front} \\ rear - front + n & \text{otherwise} \end{cases}$$

Problem-7 What is the most appropriate data structure to print elements of queue in reverse order?

Solution: Stack.

Problem-8 Implement doubly ended queues. A double-ended queue is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Solution: We will create a new class for the implementation of the abstract data type deque. In removeFront we use the pop method to remove the last element from the list. However, in removeRear, the pop(0) method must remove the first element of the list. Likewise, we need to use the insert method in addRear since the append method assumes the addition of a new element to the end of the list.

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
        self.items.insert(0,item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

Problem-9 Given a stack of integers, how do you check whether each successive pair of numbers in the stack is consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. For example, if the stack of elements are [4, 5, -2, -3, 11,

10, 5, 6, 20], then the output should be true because each of the pairs (4, 5), (-2, -3), (11, 10), and (5, 6) consists of consecutive numbers.

Solution:

```
import math
def checkStackPairwiseOrder(stk):
    que = Queue()
    pairwiseOrdered = 1
    #Reverse Stack elements
    while not stk.isEmpty():
            que.enQueue(stk.pop())
    while not que.isEmpty():
            stk.push(que.deQueue())
    while not stk.isEmpty():
            n = stk.pop()
            que.enQueue(n)
            if not stk.isEmpty():
                    m = stk.pop()
                    que.enQueue(m)
                    if (abs(n - m) != 1):
                            pairwiseOrdered = 0
                            break
    while not que.isEmpty():
            stk.push(que.deQueue())
    return pairwiseOrdered
stk = Stack()
stk.push(-2)
stk.push(-3)
stk.push(11)
stk.push(10)
stk.push(5)
stk.push(6)
stk.push(20)
stk.push(21)
print checkStackPairwiseOrder(stk)
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-10 Given a queue of integers, rearrange the elements by interleaving the first half of the list with the second half of the list. For example, suppose a queue stores the following sequence of values: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Consider the two halves of this list: first half: [11, 12, 13, 14, 15] second half: [16, 17, 18, 19, 20]. These are combined in an alternating fashion to form a sequence of interleave pairs: the first values from each half (11 and 16), then the second values from each half (12 and 17), then the third values from each half (13 and 18), and so on. In each pair, the value from the first half appears before the value from the second half. Thus, after the call, the queue stores the following values: [11, 16, 12, 17, 13, 18, 14, 19, 15, 20].

Solution:

```
def interLeavingQueue(que):
    stk = Stack()
    halfSize = que.size//2
    for i in range(0,halfSize):
        stk.push(que.deQueue())
    while not stk.isEmpty():
        que.enQueue(stk.pop())
    for i in range(0,halfSize):
        que.enQueue(que.deQueue())
    for i in range(0,halfSize):
        stk.push(que.deQueue())
    while not stk.isEmpty():
        que.enQueue(stk.pop())
        que.enQueue(que.deQueue())
que = Queue()
```

```
que.enQueue(11)
que.enQueue(12)
que.enQueue(13)
que.enQueue(14)
que.enQueue(15)
que.enQueue(16)
que.enQueue(17)
que.enQueue(18)
que.enQueue(19)
que.enQueue(20)
interLeavingQueue(que)
while not que.isEmpty():
   print que.deQueue()
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-11 Given an integer k and a queue of integers, how do you reverse the order of the first k elements of the queue, leaving the other elements in the same relative order? For example, if k=4 and queue has the elements [10, 20, 30, 40, 50, 60, 70, 80, 90]; the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90].

Solution:

```
def reverseQueueFirstKElements(que, k):
    stk = Stack()
    if que == None or k > que.size:
        return
    for i in range(0,k):
        stk.push(que.deQueue())
    while not stk.isEmpty():
        que.enQueue(stk.pop())
    for i in range(0,que.size-k):
        que.enQueue(que.deQueue())
que = Queue()
que.enQueue(11)
que.enQueue(12)
    que.enQueue(13)
que.enQueue(14)
que.enQueue(15)
que.enQueue(16)
que.enQueue(17)
que.enQueue(18)
que.enQueue(19)
que.enQueue(20)
que.enQueue(21)
que.enQueue(22)
reverseQueueFirstKElements(que, 4)
while not que.isEmpty():
   print que.deQueue()
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-12 Implement producer consumer problem with python threads and queues.

Solution:

```
#!/usr/bin/env python
from random import randint
from time import sleep
from Queue import Queue
from myThread import MyThread
def writeQ(queue):
    print 'producing object for Q...',
    queue.put('MONK', 1)
    print "size now", queue.qsize()
```

```
def readQ(queue):
    val = queue.get(1)
    print 'consumed object from Q... size now', queue.qsize()
def producer(queue, loops):
    for i in range(loops):
      writeQ(queue)
      sleep(randint(1, 3))
def consumer(queue, loops):
    for i in range(loops):
      readQ(queue)
      sleep(randint(2, 5))
funcs = [producer, consumer]
nfuncs = range(len(funcs))
nloops = randint(2, 5)
q = Queue(32)
threads = []
for i in nfuncs:
  t = MyThread(funcs[i], (q, nloops),
     funcs[i].__name__)
  threads.append(t)
for i in nfuncs:
  threads[i].start()
for i in nfuncs:
  threads[i].join()
print 'all DONE'
```

As you can see, the producer and consumer do not necessarily alternate in execution. In this solution, we use the Queue. We use random.randint() to make production and consumption somewhat varied.

The writeQ() and readQ() functions each have a specific purpose: to place an object in the queue—we are using the string 'MONK', for example—and to consume a queued object, respectively. Notice that we are producing one object and reading one object each time.

The producer() is going to run as a single thread whose sole purpose is to produce an item for the queue, wait for a bit, and then do it again, up to the specified number of times, chosen randomly per script execution. The consumer() will do likewise, with the exception of consuming an item, of course.

You will notice that the random number of seconds that the producer sleeps is in general shorter than the amount of time the consumer sleeps. This is to discourage the consumer from trying to take items from an empty queue. By giving the producer a shorter time period of waiting, it is more likely that there will already be an object for the consumer to consume by the time their turn rolls around again.

These are just setup lines to set the total number of threads that are to be spawned and executed.

Finally, we have our main() function, which should look quite similar to the main() in all of the other scripts in this chapter. We create the appropriate threads and send them on their way, finishing up when both threads have concluded execution.

We infer from this example that a program that has multiple tasks to perform can be organized to use separate threads for each of the tasks. This can result in a much cleaner program design than a single-threaded program that attempts to do all of the tasks.

We illustrated how a single-threaded process can limit an application's performance. In particular, programs with independent, non-deterministic, and non-causal tasks that execute sequentially can be improved by division into separate tasks executed by individual threads. Not all applications will benefit from multithreading due to overhead and the fact that the Python interpreter is a single-threaded application, but now you are more cognizant of Python's threading capabilities and can use this tool to your advantage when appropriate.

Problem-13 Given a string , write a Python method to check whether it is a palindrome or nor using doubly ended queue.

Solution:

```
class Deque:
  def __init__(self):
    self.items = []
```

```
    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)

def palchecker(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False

    return stillEqual

print(palchecker("lsdkjfskf"))
print(palchecker("madam"))
```

Time Complexity: O(n). Space Complexity: O(n).