Lecture

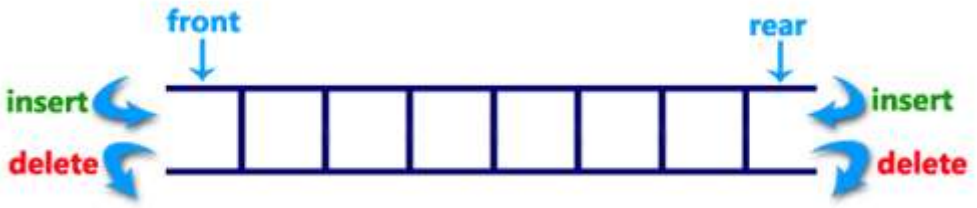
NOTE: FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

Queue

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end.

Double Ended Queue (DEque)

Double Ended Queue (DEQue) is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (***Head*** and ***Tail***). That means, we can ***enqueue*** at both front and rear positions and can ***dequeue*** from both front and rear positions.

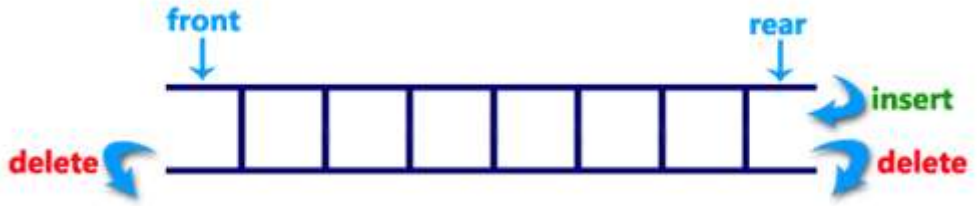


Double Ended Queue can be represented in TWO ways, those are as follows:

1. Input Restricted DEQue
2. Output Restricted DEQue

Input Restricted DEque

In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted DEque

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.

Examples:

Make an example for both the cases. What scenario best describe both the above mentioned ways of DEQue Implementation?

Representation of DEQue with Array

To Understand the process we will implement the DEQue with insertion and deletion at both ends. However, this can be changed by deleting methods for Input Restricted or Output Restricted DEQue. ***You may observe some unusual behavior of the DEQue that we discuss when we see output.***

```
public class DEque
{
    int n = 5;
    int[] arr;
    int head = 0;
    int tail = -1;

    public DEque()
    {
        arr = new int[n];
    }

    public DEque(int x)
    {
        arr = new int[x];
        n = x;
    }

    public boolean isEmpty()
    {
        if(tail < 0)
            return true;
        else
            return false;
    }

    public boolean isFull()
    {
```

```
        if(tail == n-1)
            return true;
        else
            return false;
    }

    public void enqueueAtTail(int value)
    {
        if(isFull())
            System.out.println("Queue is full");
        else
            arr[++tail] = value;
    }

    public void enqueueAtHead(int value)
    {
        if(isFull())
            System.out.println("Queue is full");
        else
        {
            moveBack();
            arr[head] = value;
        }
    }

    public int dequeueAtHead()
    {
        if(isEmpty())
        {
            System.out.println("Queue is Empty");
            return -9999;
        }
        else
        {
            int value = arr[head];
            moveFwd();
            return value;
        }
    }

    public int dequeueAtTail()
    {
        if(isEmpty())
        {
            System.out.println("Queue is Empty");
            return -9999;
        }
        else
        {
            int value = arr[tail];
            tail--;
            return value;
```

```
        }
    }

    private void moveFwd()
    {
        for(int i = 0; i < tail; i++)
        {
            arr[i] = arr[i+1];
        }
        tail--;
    }

    private void moveBack()
    {
        for(int i = tail; i >= 0; i--)
        {
            arr[i+1] = arr[i];
        }
        tail++;
    }
}
```

Code above is the code for DEQue class. However, to test the DEQue class we will use the following code in the main class.

```
public class DEqueDemo
{
    public static void main(String[] args)
    {
        DEque que = new DEque();
        que.dequeueAtTail();
        que.dequeueAtHead();

        // Lets Test Head Side
        System.out.println("\n*** TESTING ONE END ***\n");

        que.enqueueAtHead(23);
        que.enqueueAtHead(2);
        que.enqueueAtHead(73);
        que.enqueueAtHead(21);
        que.enqueueAtHead(109);
        que.enqueueAtHead(13);
        while(!que.isEmpty())
        {
            System.out.print(que.dequeueAtHead() + "   ");
        }
        System.out.println("");
        que.dequeueAtHead();

        // Now Lets check Other End
        System.out.println("\n*** TESTING OTHER END ***\n");
```

```
        que.enqueueAtTail(23);
        que.enqueueAtTail(2);
        que.enqueueAtTail(73);
        que.enqueueAtTail(21);
        que.enqueueAtTail(109);
        que.enqueueAtTail(13);
        while(!que.isEmpty())
        {
            System.out.print(que.dequeueAtTail()+ "    ");
        }
        System.out.println("");
        que.dequeueAtTail();

        // Now Lets Check Normal Operation
        System.out.println("\n*** TESTING NORMAL OPERATION ***\n");

        que.enqueueAtTail(23);
        que.enqueueAtTail(2);
        que.enqueueAtTail(73);
        que.enqueueAtTail(21);
        que.enqueueAtTail(109);
        que.enqueueAtTail(13);
        while(!que.isEmpty())
        {
            System.out.print(que.dequeueAtHead()+ "    ");
        }
        System.out.println("");
        que.dequeueAtHead();
    }
}
```

Output:

```
Queue is Empty
Queue is Empty

*** TESTING ONE END ***

Queue is full
109    21    73    2    23    // Unusual? Is it?
Queue is Empty

*** TESTING OTHER END ***

Queue is full
109    21    73    2    23    // Unusual? Is it?
Queue is Empty

*** TESTING NORMAL OPERATION ***
```

```
Queue is full
23   2   73   21   109   // That's seems ok
Queue is Empty
```

Multiple Queues

Multiple queues are the queues where you will implement more than one queue to implement some idea or some implementation of some specific functionality. For example, if you want to implement priority queues. We will see next how we use multiple queues for priority queues.

Priority Queues

If you have regular queue and data is served on the ***FIFO*** principal but what if a special data arrives that needs to serve first? How you implement the priority for that data in queue?
You got a scenario where you have two priority levels. How can you implement it? What if you got more than two priority levels how you will implement the queues then?
If we have two priority levels then we can also use DEQue. However, it may cause some issues when multiple priority items arrive at once. Each new arrival will push back the previous high priority item. To address the issue lets maintain multiple queues based on priority.

Examples:

What scenario best describe where we require more than one priority level? Do you have Idea how Operating System schedule its jobs?

Representation of Priority Queues using Multiple Queues

We can use the queues that we have already studied and implement the functionality of priority queues.

```
public class PriorityQueue
{
    // We create Multiple Queues
    // Based on levels of Priorities
    CircularQueue q0 = new CircularQueue(5);
    CircularQueue q1 = new CircularQueue(5);
    CircularQueue q2 = new CircularQueue(5);

    public void enqueue(int value, int priority)
    {
        if(priority == 2)
            q2.enqueue(value);
        else if(priority == 1)
            q1.enqueue(value);
        else
            q0.enqueue(value);
    }
```

```
    public int dequeue()
    {
        if(!q2.isEmpty())
            return q2.dequeue();
        else if(!q1.isEmpty())
            return q1.dequeue();
        else
            return q0.dequeue();
    }
}
```

Code above is the code for priority queue class. However, to test the PriorityQueue class we will use the following code in the main class.

```
public class PriorityQueueDemo
{
    public static void main(String[] args)
    {
        PriorityQueue q = new PriorityQueue();
        q.enqueue(10, 0);
        q.enqueue(20, 0);
        q.enqueue(30, 0);
        q.enqueue(40, 1);
        q.enqueue(50, 1);
        q.enqueue(60, 2);
        q.enqueue(70, 0);
        System.out.println("Dequeue Sequence:");
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        q.dequeue();
    }
}
```

Output:

```
Dequeue Sequence:
60
40
50
10
20
30
70
Queue is Empty
```