Lecture

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

Queue

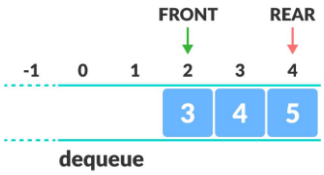
Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end.

Circular Queue

In today's lecture, we will discuss the circular queues. Why we need circular queues if we already have regular queues? As you may remember in last lecture we implemented the ***dequeue*** method with moving the elements one step forward after each ***dequeue*** operation. This lead to the ***dequeue*** cost to increase to ***O(n)***. as we have to move each item in the queue forward and if we have ***n*** number of items in the queue then we have to move ***n*** elements forward with each ***dequeue*** operation hence the ***dequeue*** cost increased to ***O(n)***.

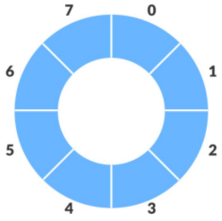
There is a workaround to avoid this situation if we move the ***head*** to next in-line item then we can achieve the ***dequeue*** operation in ***O(1)***. However, this method also has a demerit.

Let's have a look at this demerit:

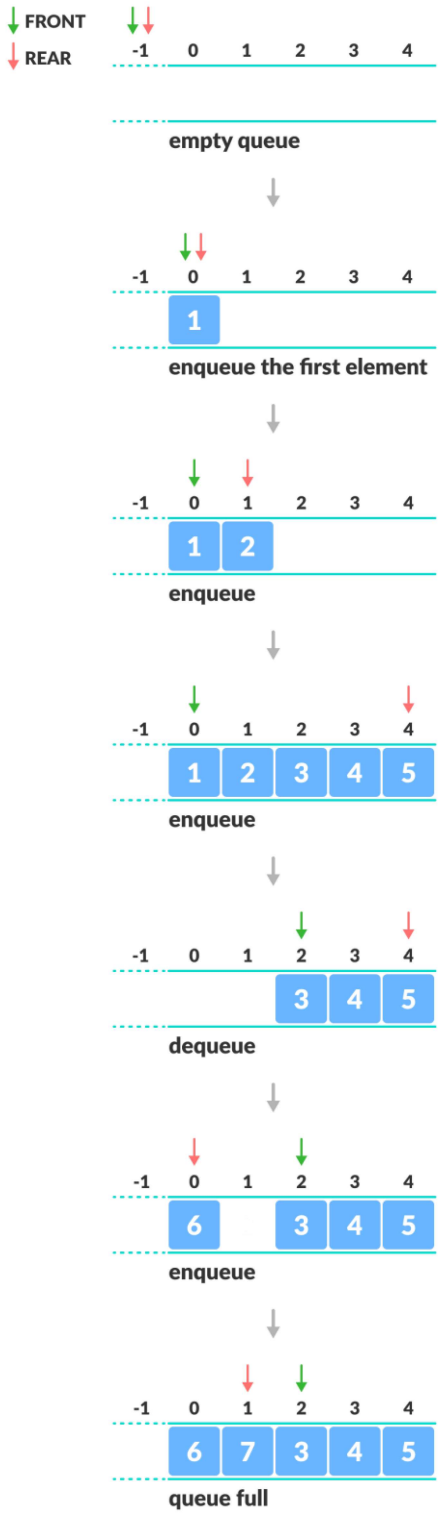


While the ***front*** or ***head*** moves towards the ***rear*** we are wasting space behind the ***front*** that now cannot be used as the ***rear*** reaches the end of array it will stop moving. Even though there are two spaces left in the queue we cannot insert item to it unless ***rear*** can go around again.

We can implement the circular queues by utilizing the movement of ***rear*** or ***tail*** to start again from the beginning of array hence forming a circle. This will yield us the circular queue. The advantage of circular queue is that we can achieve the ***dequeue*** operation in ***O(1)*** without losing any space in our array that we are using as queue.



We can achieve the circular motion however the array remain as is it's the movement of the pointers for ***head*** and ***tail*** that behave like this is a circle as they reach the end they go back to start.



Representation of Circular Queue with Array

The beauty of ADT is that we can implement the Data Structure in any way as long as it's primitive operations perform as intended. You may find multiple implementations online but we will implement the circular queue as we implemented the queue in the first place so you can understand it better.

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

    public CircularQueue()
    {
        arr = new int[5];
        n = 5;
    }

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

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

    public boolean isFull()
    {
        if(tail == (head+n-1))
            return true;
        else
            return false;
    }

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

```
    }

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

    public int front()
    {
        if(isEmpty())
        {
            System.out.println("Queue is Empty");
            return -9999;
        }
        else
            return arr[head%n];
    }

    public int size()
    {
        return (tail-head)+1;
    }
}
```

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

```
public class CQDemo
{
    public static void main(String[] args)
    {
        CircularQueue que = new CircularQueue(5);
        int x = que.dequeue();
        que.enqueue(23);
        que.enqueue(2);
        que.enqueue(73);
        System.out.println(que.front());
        que.enqueue(21);
        que.enqueue(109);
        System.out.println("Queue Size is " + que.size());
        System.out.println("Is Queue Full: " + que.isFull());
```

```
        que.enqueue(13);
        x = que.dequeue();
        System.out.println("Dequeued Value Received is " + x);
        x = que.dequeue();
        x = que.dequeue();
        x = que.dequeue();
        x = que.dequeue();
        System.out.println("Dequeued Value Received is " + x);
        x = que.dequeue();

        System.out.println("\nREPEAT AGAIN\n");
        //lets repteat to check weather its working
        x = que.dequeue();
        que.enqueue(23);
        que.enqueue(2);
        que.enqueue(73);
        System.out.println(que.front());
        que.enqueue(21);
        System.out.println("Queue Size is " + que.size());
        System.out.println("Is Queue Full: " + que.isFull());
        que.enqueue(13);
        que.enqueue(15);
        x = que.dequeue();
        System.out.println("Dequeued Value Received is " + x);
        x = que.dequeue();
        x = que.dequeue();
        x = que.dequeue();
        x = que.dequeue();
        System.out.println("Dequeued Value Received is " + x);
        x = que.dequeue();
    }
}
```

Output:

```
Queue is Empty
23
Queue Size is 5
Is Queue Full: true
Queue is full
Dequeued Value Received is 23
Dequeued Value Received is 109
Queue is Empty

REPEAT AGAIN

Queue is Empty
23
Queue Size is 4
Is Queue Full: false
```

```
Queue is full
Dequeued Value Received is 23
Dequeued Value Received is 13
Queue is Empty
```