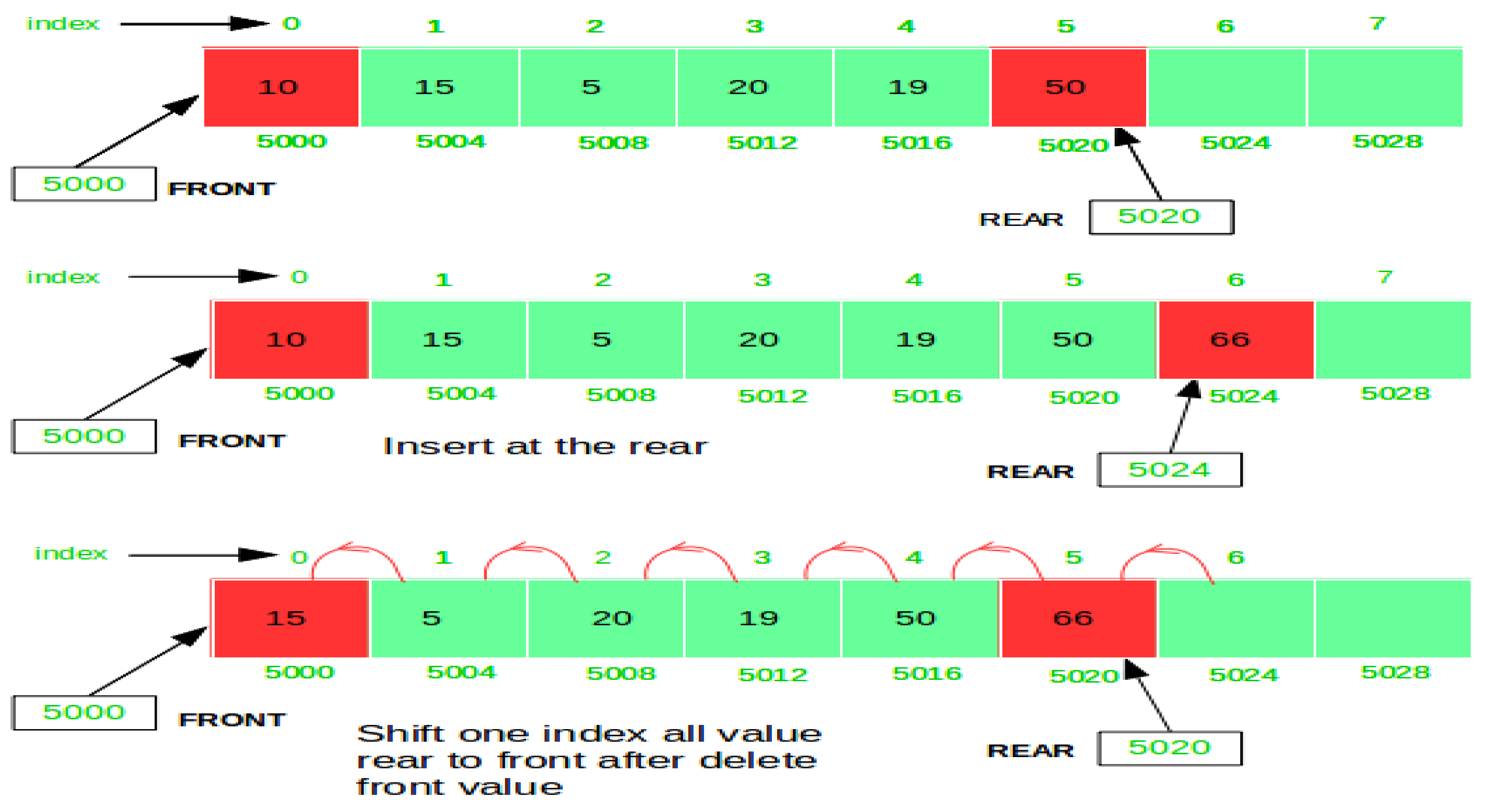
Contents :-

- ❑ Queue implementation with array.
- ❑ Circular Queue implementation
- ❑Queue implementation with Linklist.
- ❑ Double Ended Queue Implementation with double link list.
- ❑ Adapter Pattern
- ❑ Circular Link list

IMPETUS

Queue:-

- A queue is another kind of linear data structure that is used to store elements just like any other data structure but in a particular manner.
- In simple words, we can say that the queue is a type of data structure in the Java programming language that stores elements of the same kind.
- The components in a queue are stored in a FIFO (First In, First Out) behavior.



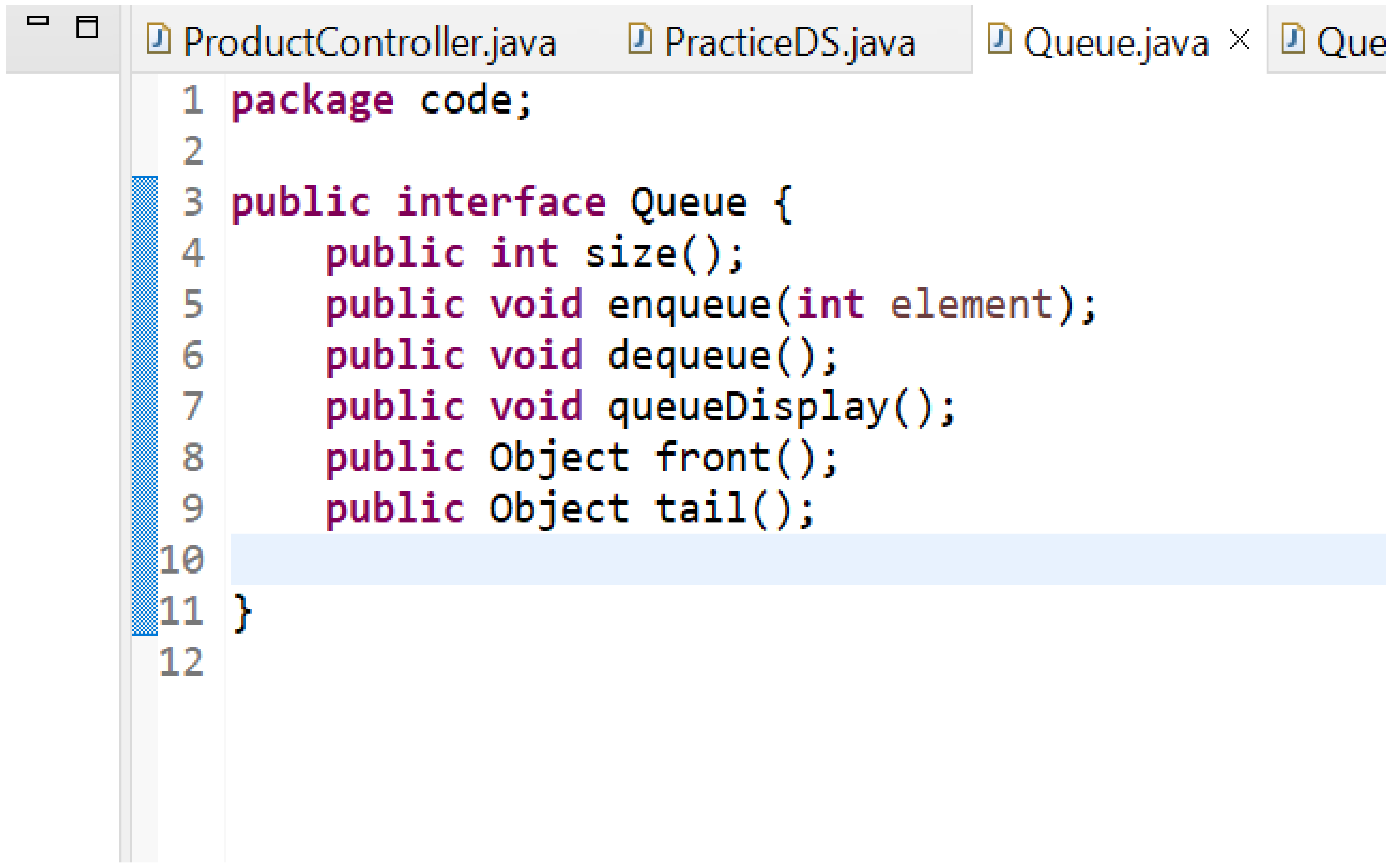
Operations on queue:-

To implement queue using Arrays, we first declare an array that holds n number of elements.
Then we define the following operations to be performed in this queue.

- **Enqueue:** An operation to insert an element in the queue is Enqueue (function queue Enqueue in the program). For inserting an element at the rear end, we need first to check if the queue is full. If it is full, then we cannot insert the element. If rear < n, then we insert the element in the queue.
- **Dequeue:** The operation to delete an element from the queue is Dequeue (function queue Dequeue in the program). First, we check whether the queue is empty. For dequeue operation to work, there has to be at least one element in the queue.
- **Front:** This method returns the front of the queue.
- **Tail**: This method returns the element at rear of the queue.
- **Display:** This method traverses the queue and displays the elements of the queue.

A queue interface in java:-

The queue data structure is “build-in” class in java’s java.util.package. But we can define our own queue interface.



ProductController.java | PracticeDS.java | Queue.java | Que

```java
package code;

public interface Queue {
    public int size();
    public void enqueue(int element);
    public void dequeue();
    public void queueDisplay();
    public Object front();
    public Object tail();

}
```

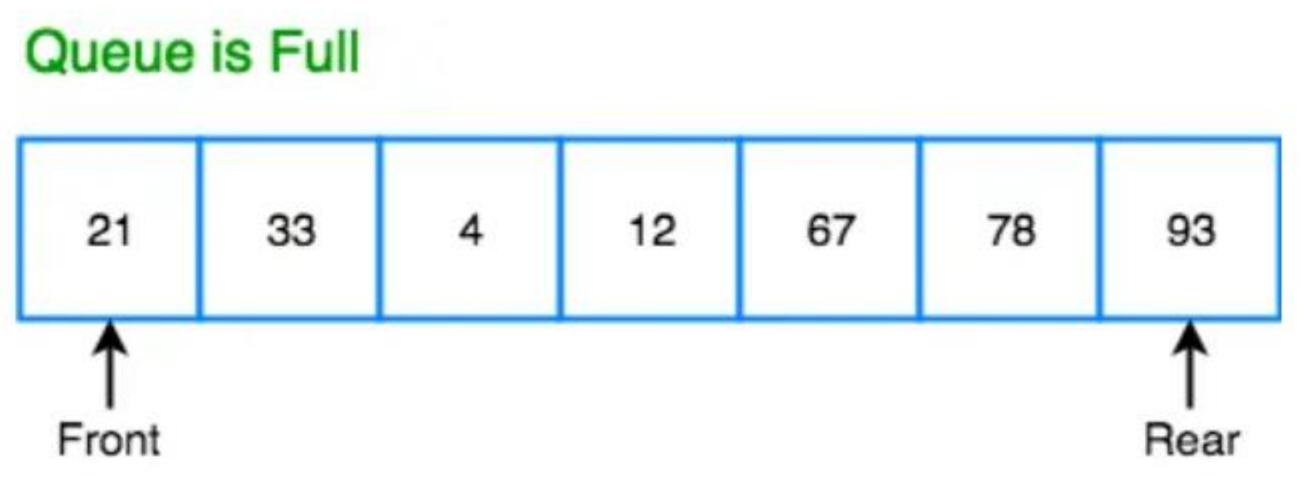
Drawbacks of simple queue with array

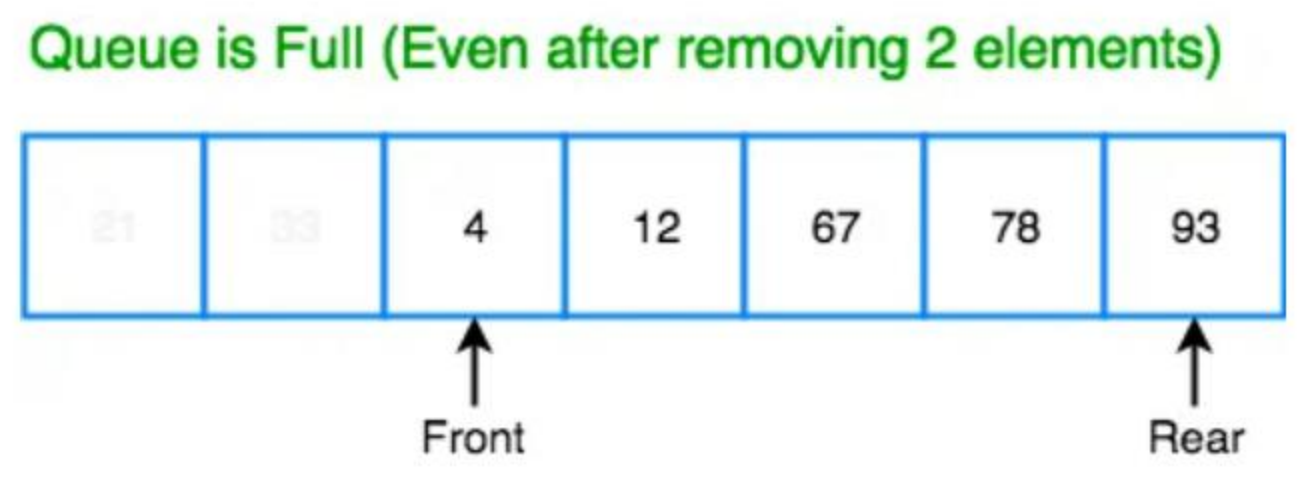
Memory Wastage:

-The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.

Array Size:

-There might be situations in which, we may need to extend the queue to insert more elements if we use an array to implement queue, It will almost be impossible to extend the array size, therefore deciding the correct array size is always a problem in array implementation of queue.

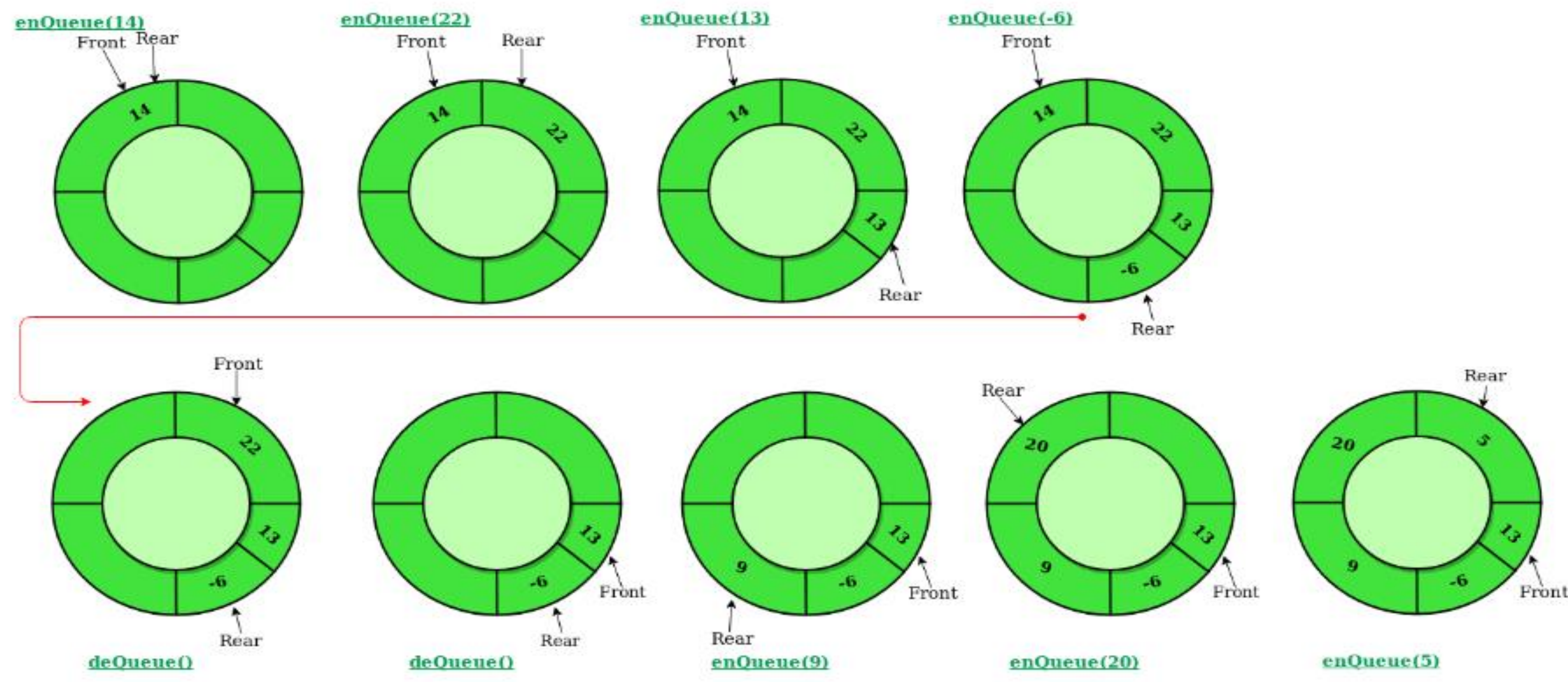




Circular queue:-

One method of overcoming is by using a circular queue, which ensures that the full array is utilized when storing the elements.

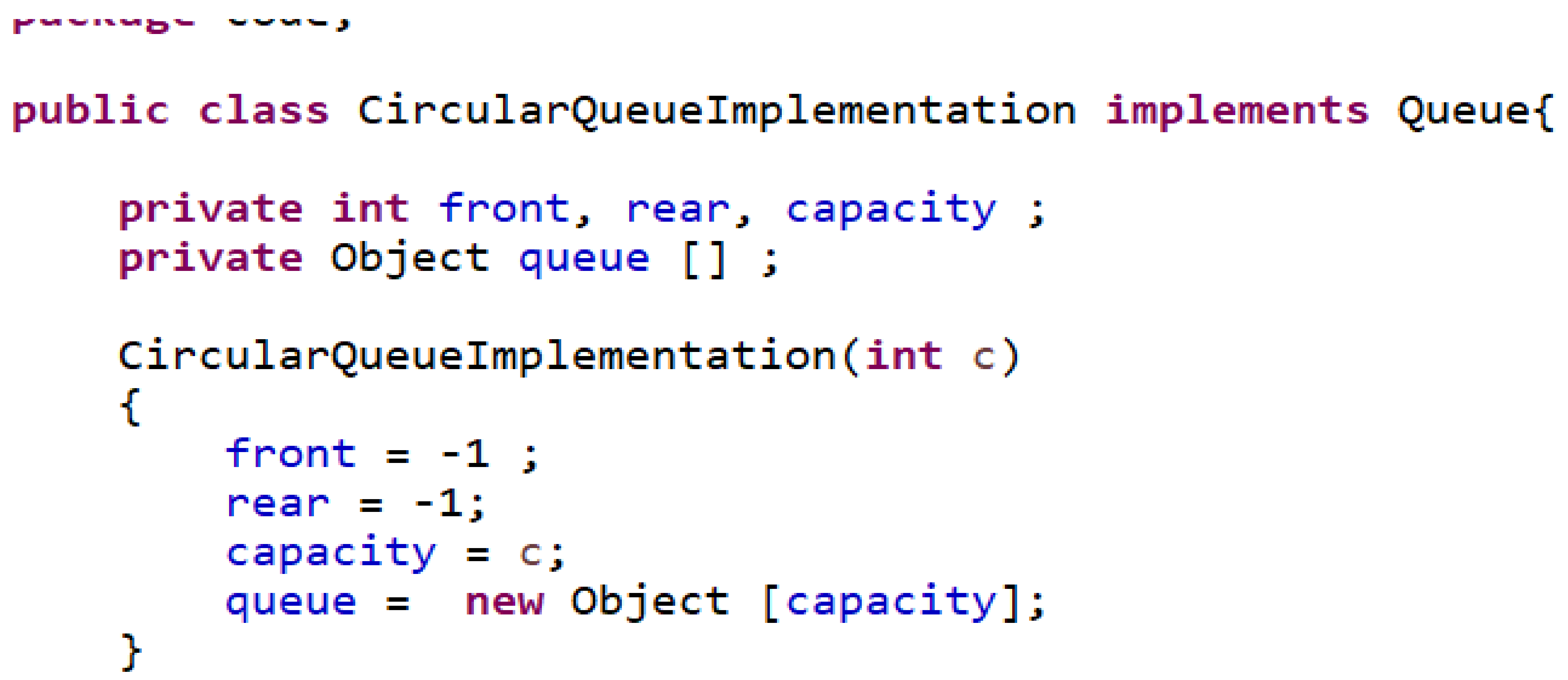
In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



IMPETUS

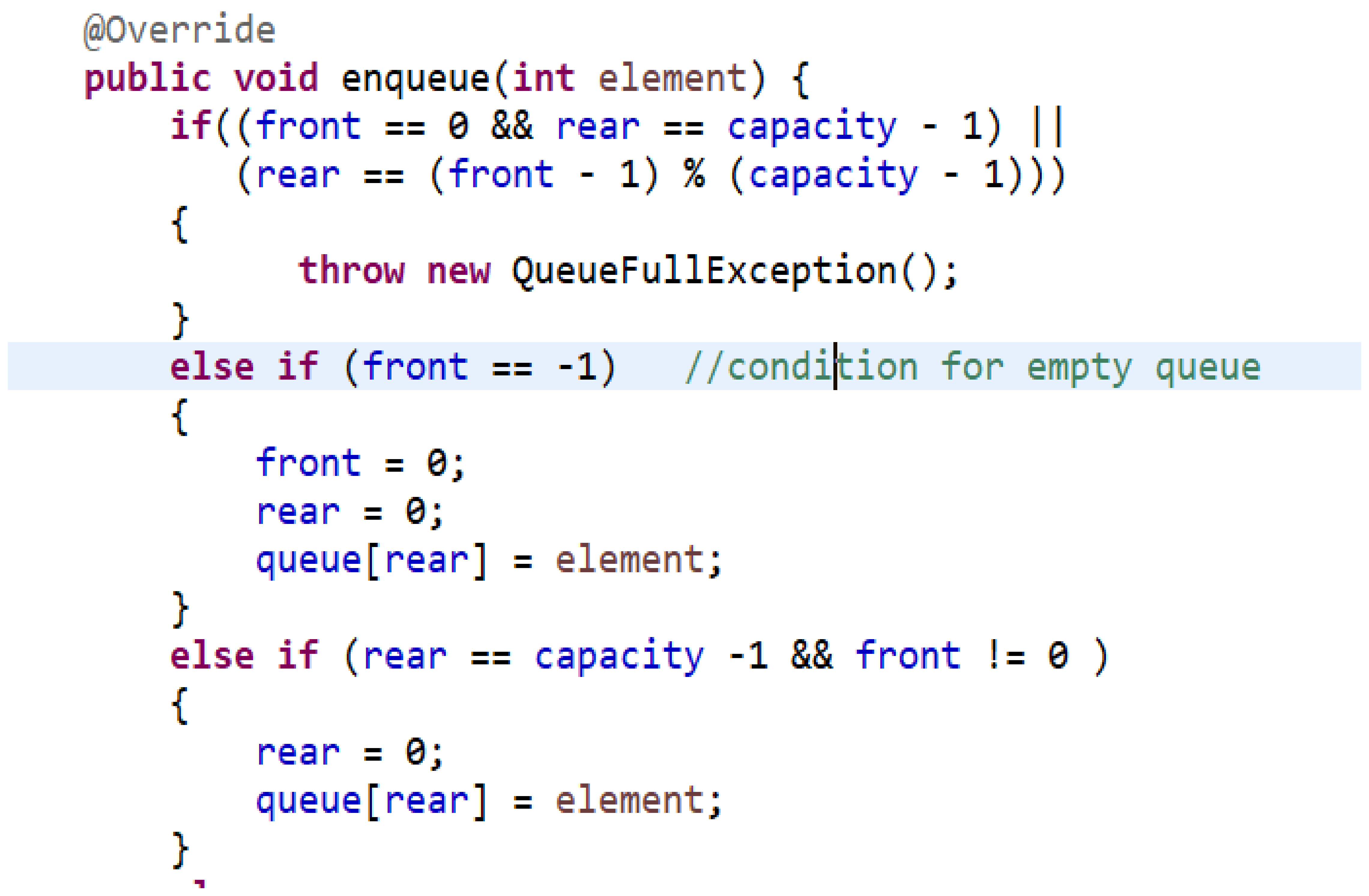
Implementation:-

- This is the Structure of our queue.

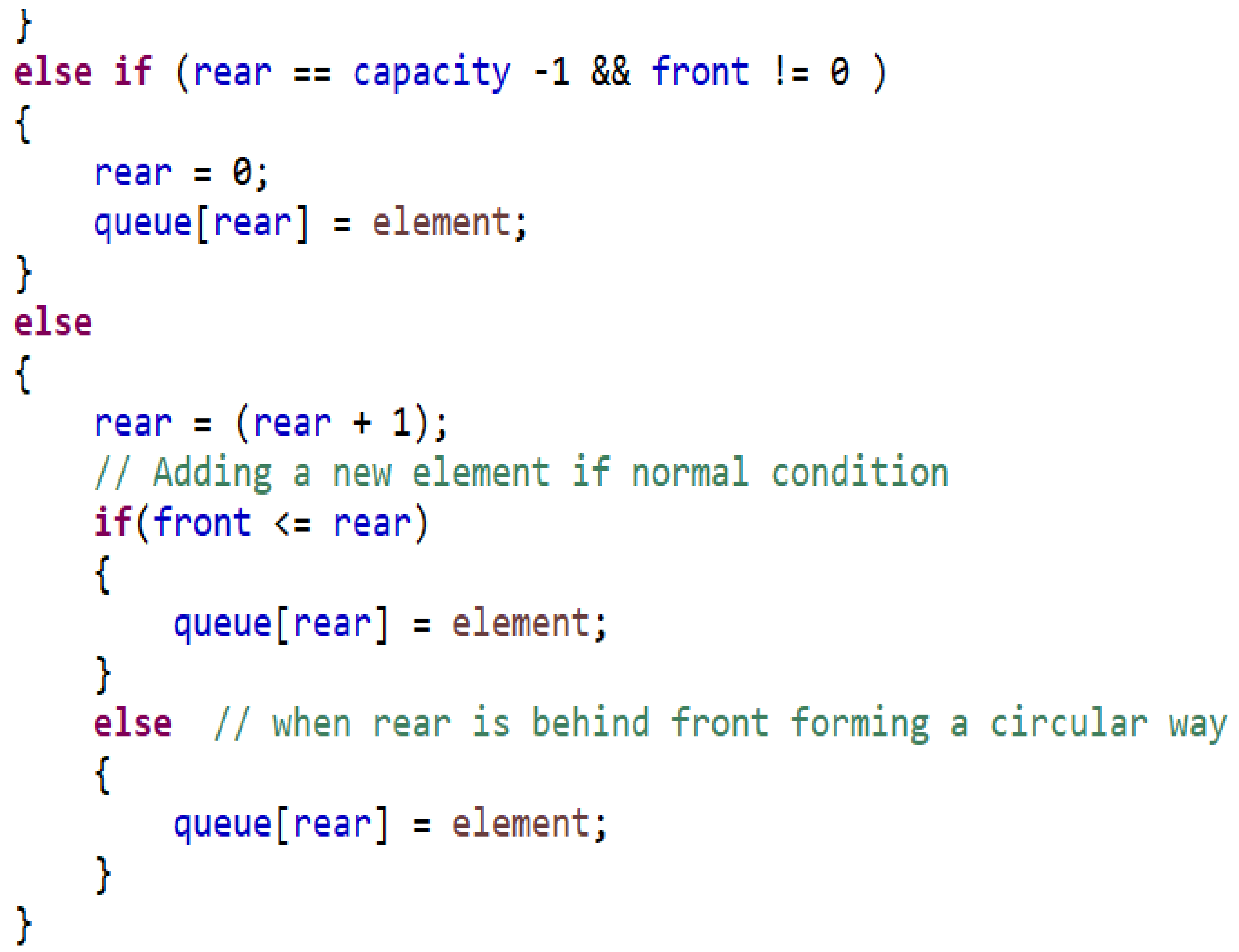


```
package code;

public class CircularQueueImplementation implements Queue{

    private int front, rear, capacity ;
    private Object queue [] ;

    CircularQueueImplementation(int c)
    {
        front = -1 ;
        rear = -1;
        capacity = c;
        queue =  new Object [capacity];
    }
```

Enqueue() :-

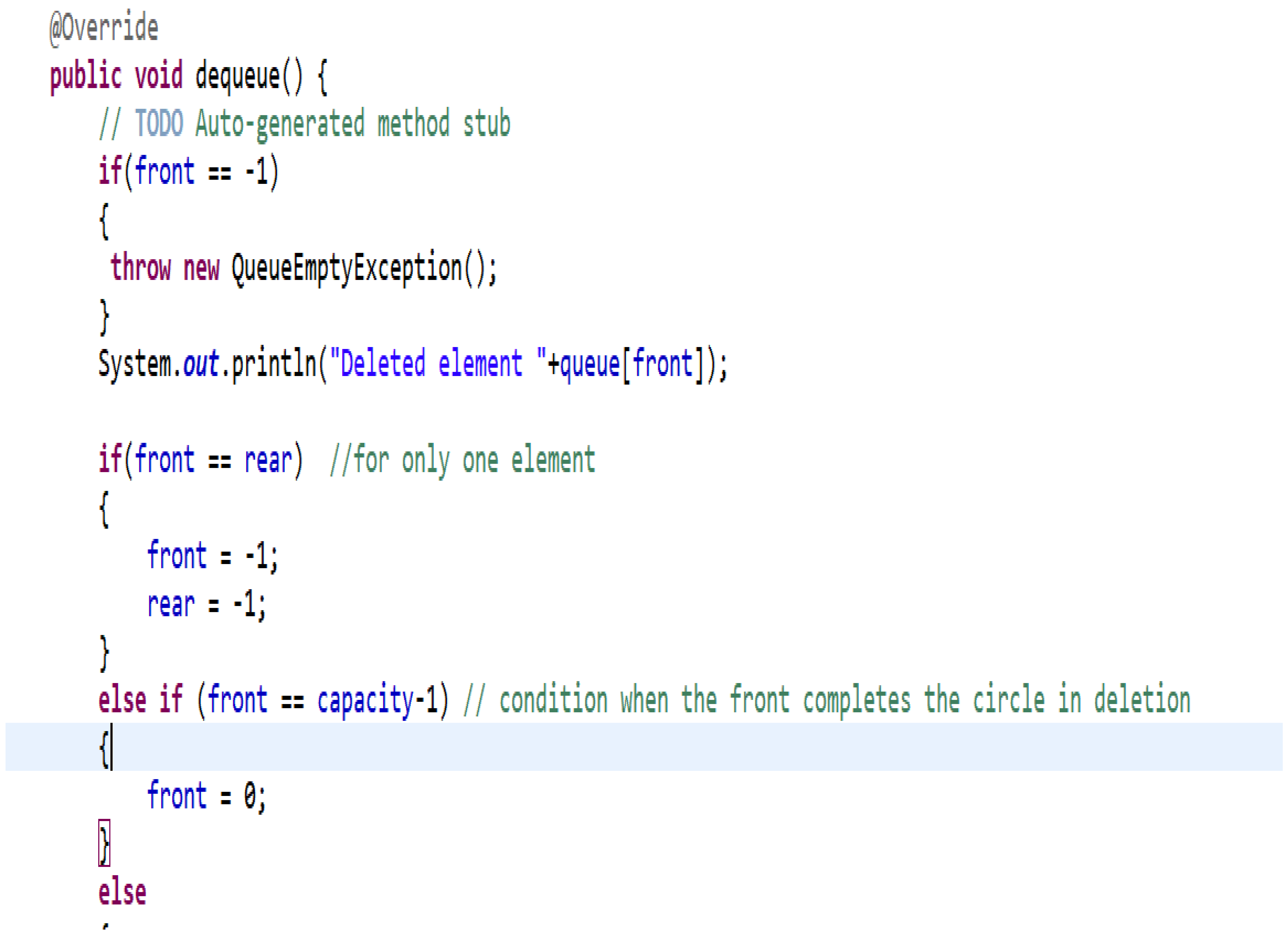


```
@Override
public void enqueue(int element) {
    if((front == 0 && rear == capacity - 1) ||
       (rear == (front - 1) % (capacity - 1)))
    {
          throw new QueueFullException();
    }
    else if (front == -1)   //condition for empty queue
    {
        front = 0;
        rear = 0;
        queue[rear] = element;
    }
    else if (rear == capacity -1 && front != 0 )
    {
        rear = 0;
        queue[rear] = element;
    }
```

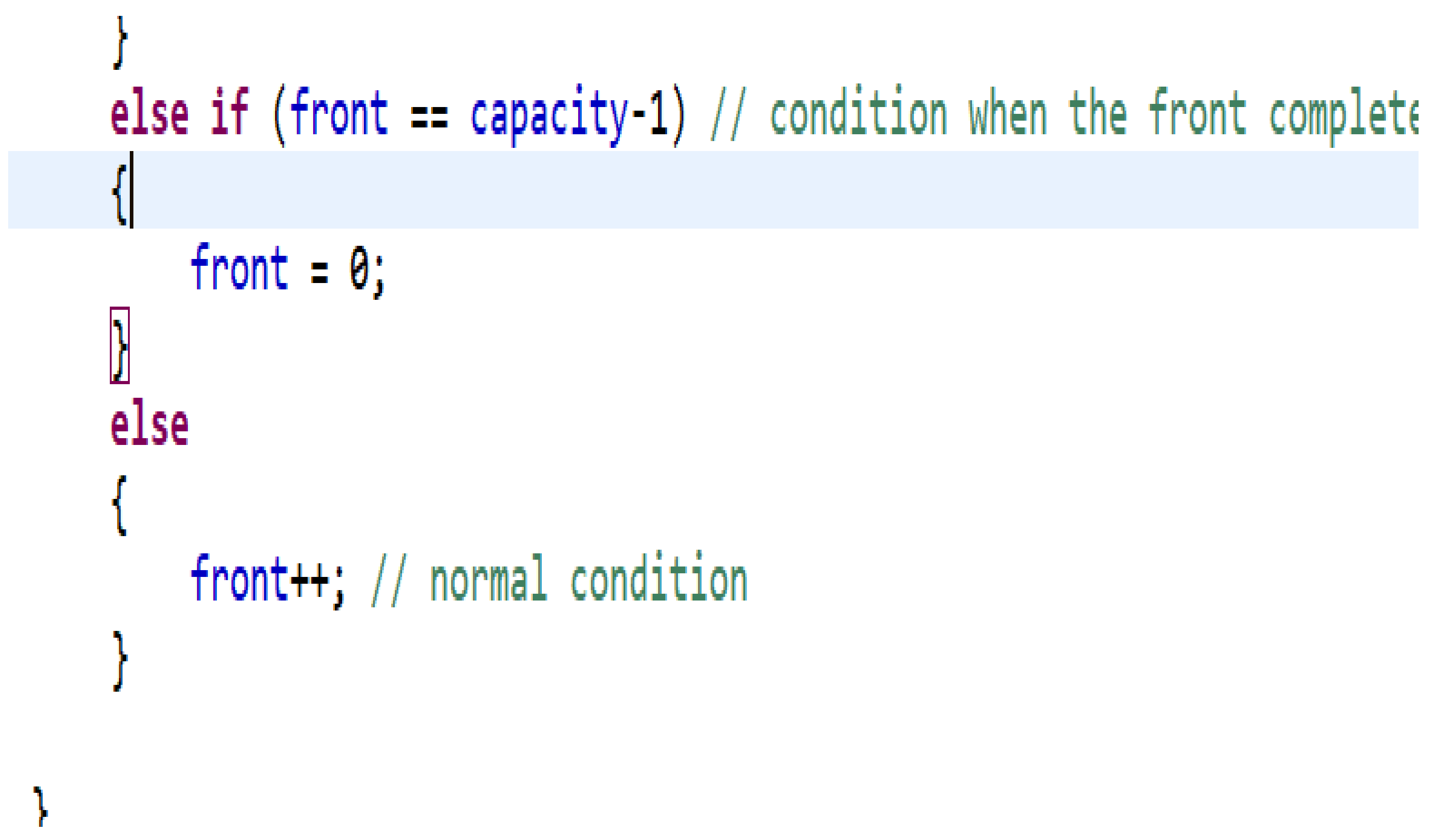


```
}
else if (rear == capacity -1 && front != 0 )
{
    rear = 0;
    queue[rear] = element;
}
else
{
    rear = (rear + 1);
    // Adding a new element if normal condition
    if(front <= rear)
    {
        queue[rear] = element;
    }
    else  // when rear is behind front forming a circular way
    {
        queue[rear] = element;
    }
}
```

Dequeue():-



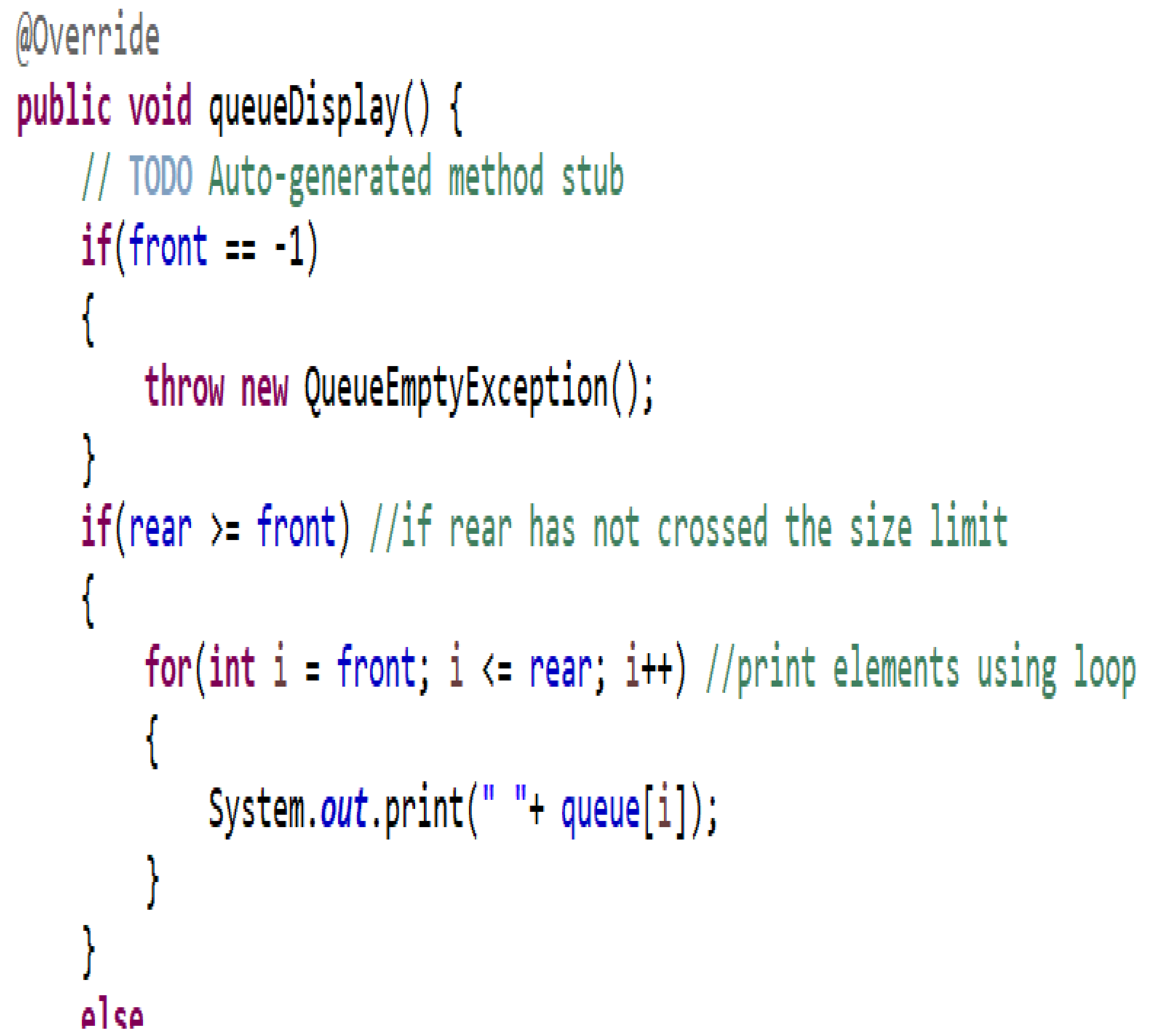
```
@Override
public void dequeue() {
    // TODO Auto-generated method stub
    if(front == -1)
    {
     throw new QueueEmptyException();
    }
    System.out.println("Deleted element "+queue[front]);

    if(front == rear)  //for only one element
    {
        front = -1;
        rear = -1;
    }
    else if (front == capacity-1) // condition when the front completes the circle in deletion
    {
        front = 0;
    }
    else
```



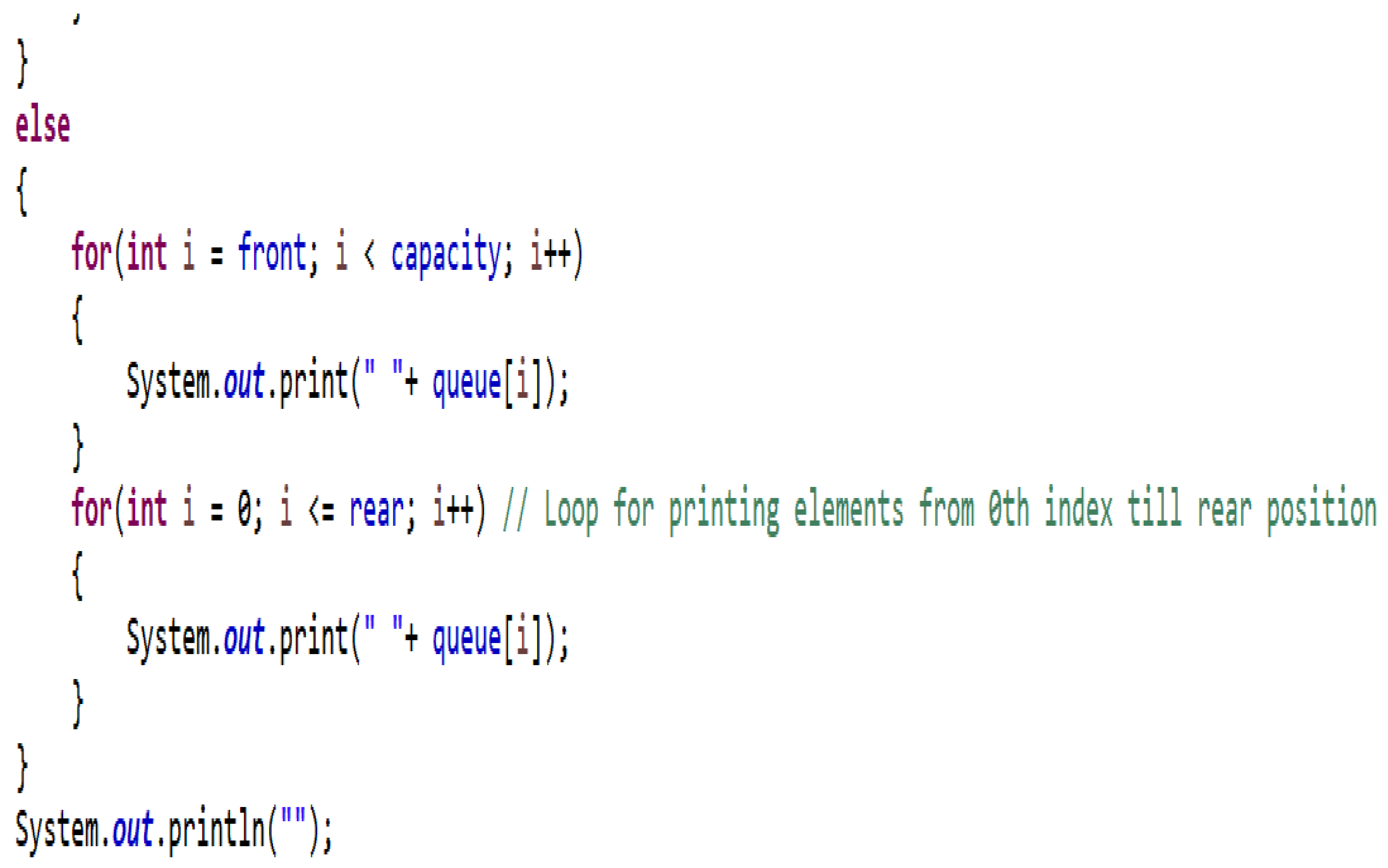
```
    }
    else if (front == capacity-1) // condition when the front complete
    {
        front = 0;
    }
    else
    {
        front++; // normal condition
    }

}
```

IMPETUS

Display():-



```
@Override
public void queueDisplay() {
    // TODO Auto-generated method stub
    if(front == -1)
    {
        throw new QueueEmptyException();
    }
    if(rear >= front) //if rear has not crossed the size limit
    {
        for(int i = front; i <= rear; i++) //print elements using loop
        {
            System.out.print(" "+ queue[i]);
        }
    }
    else
```

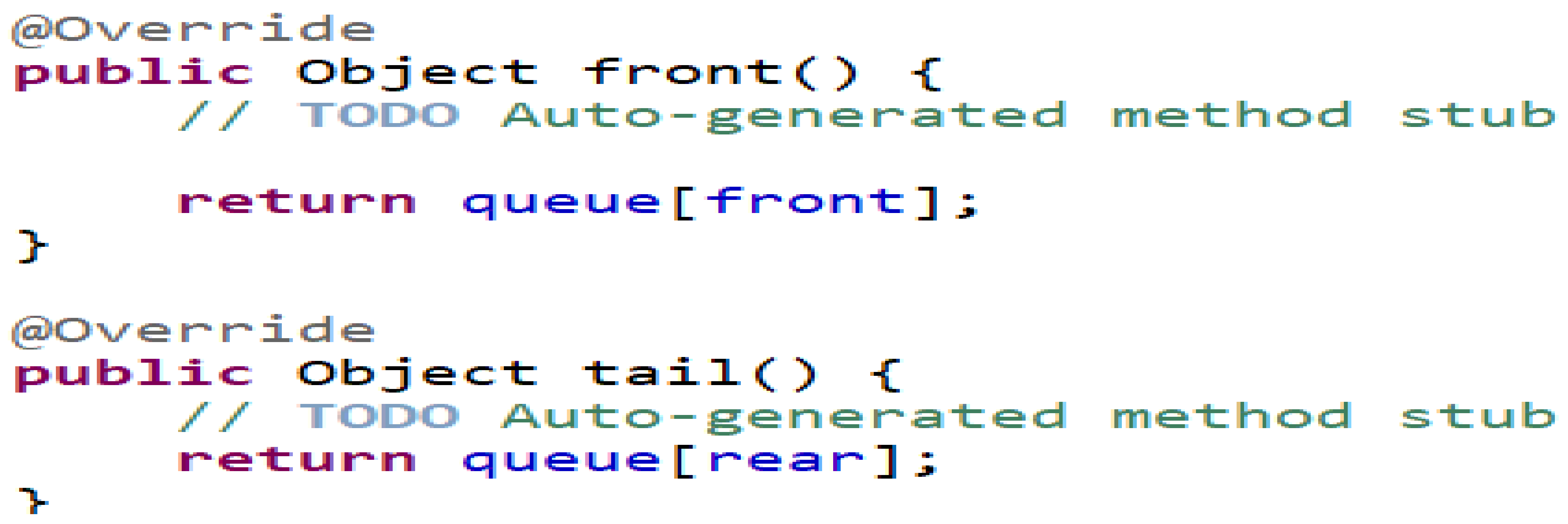


```
    }
    else
    {
        for(int i = front; i < capacity; i++)
        {
            System.out.print(" "+ queue[i]);
        }
        for(int i = 0; i <= rear; i++) // Loop for printing elements from 0th index till rear position
        {
            System.out.print(" "+ queue[i]);
        }
    }
    System.out.println("");
```

IMPETUS

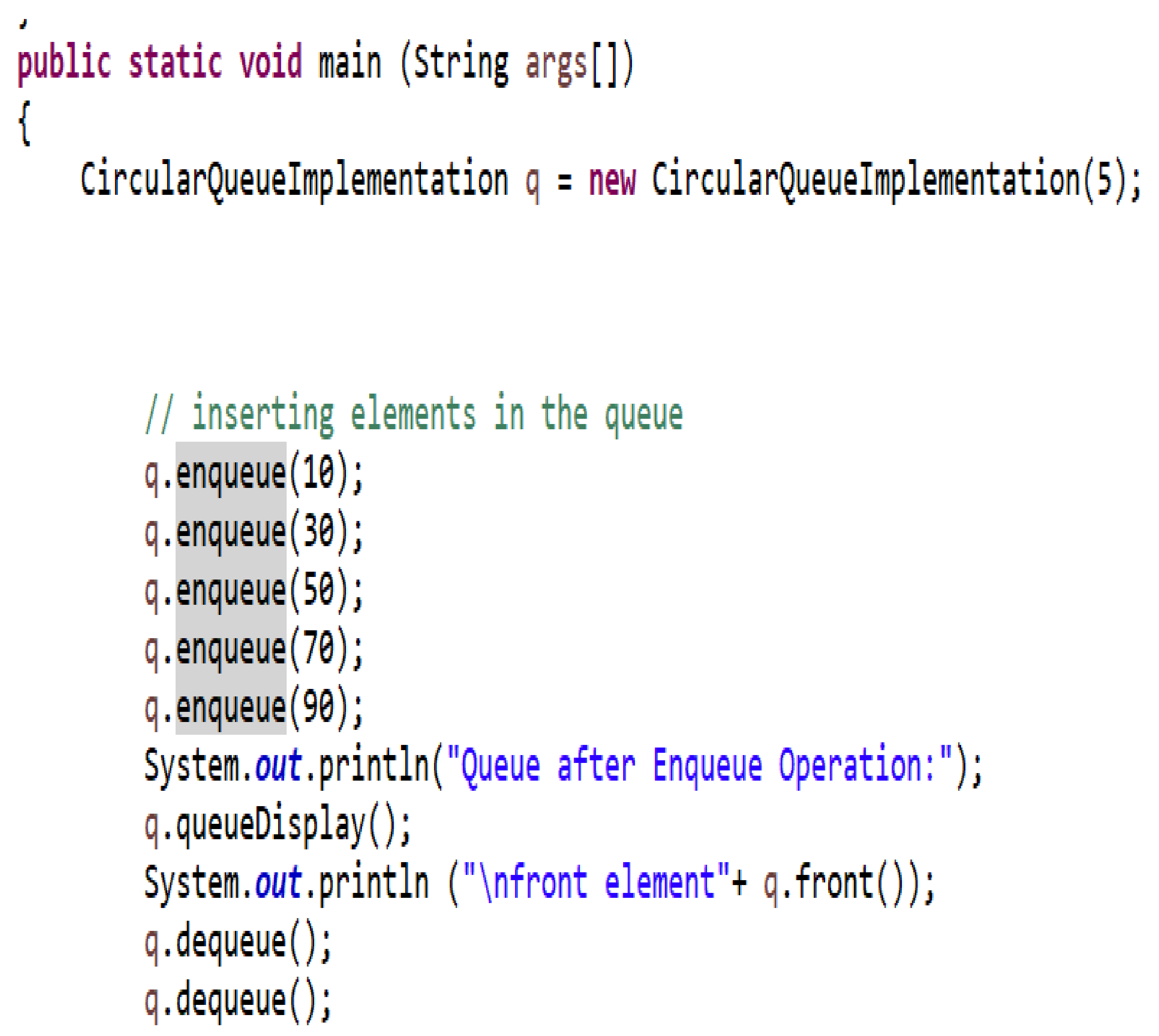
Head and tails methods:-

These are just simple methods to know the location of front and rear elements in the queue.

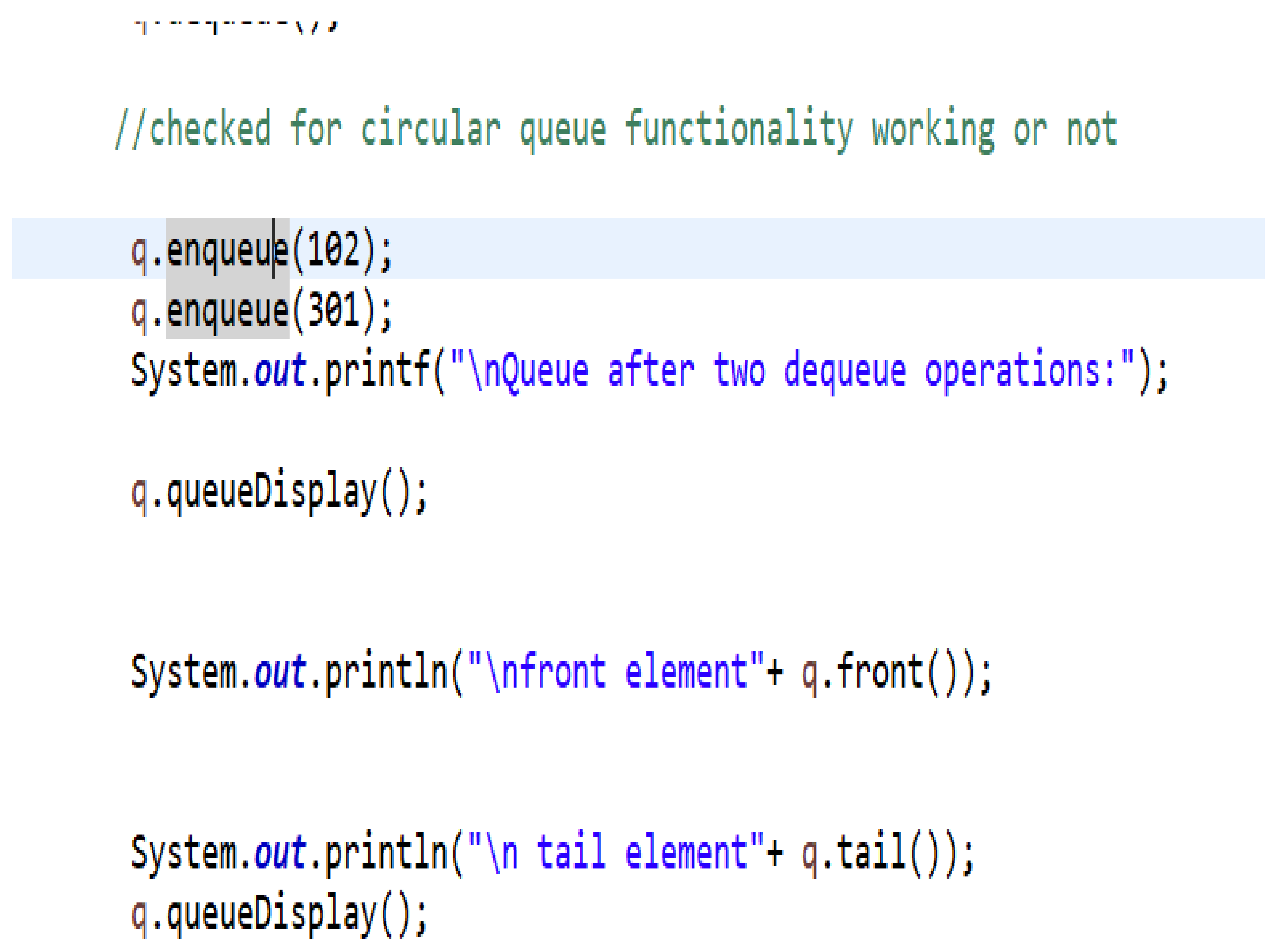


```java
@Override
public Object front() {
    // TODO Auto-generated method stub

    return queue[front];
}

@Override
public Object tail() {
    // TODO Auto-generated method stub
    return queue[rear];
}
```

Operations on queue:-

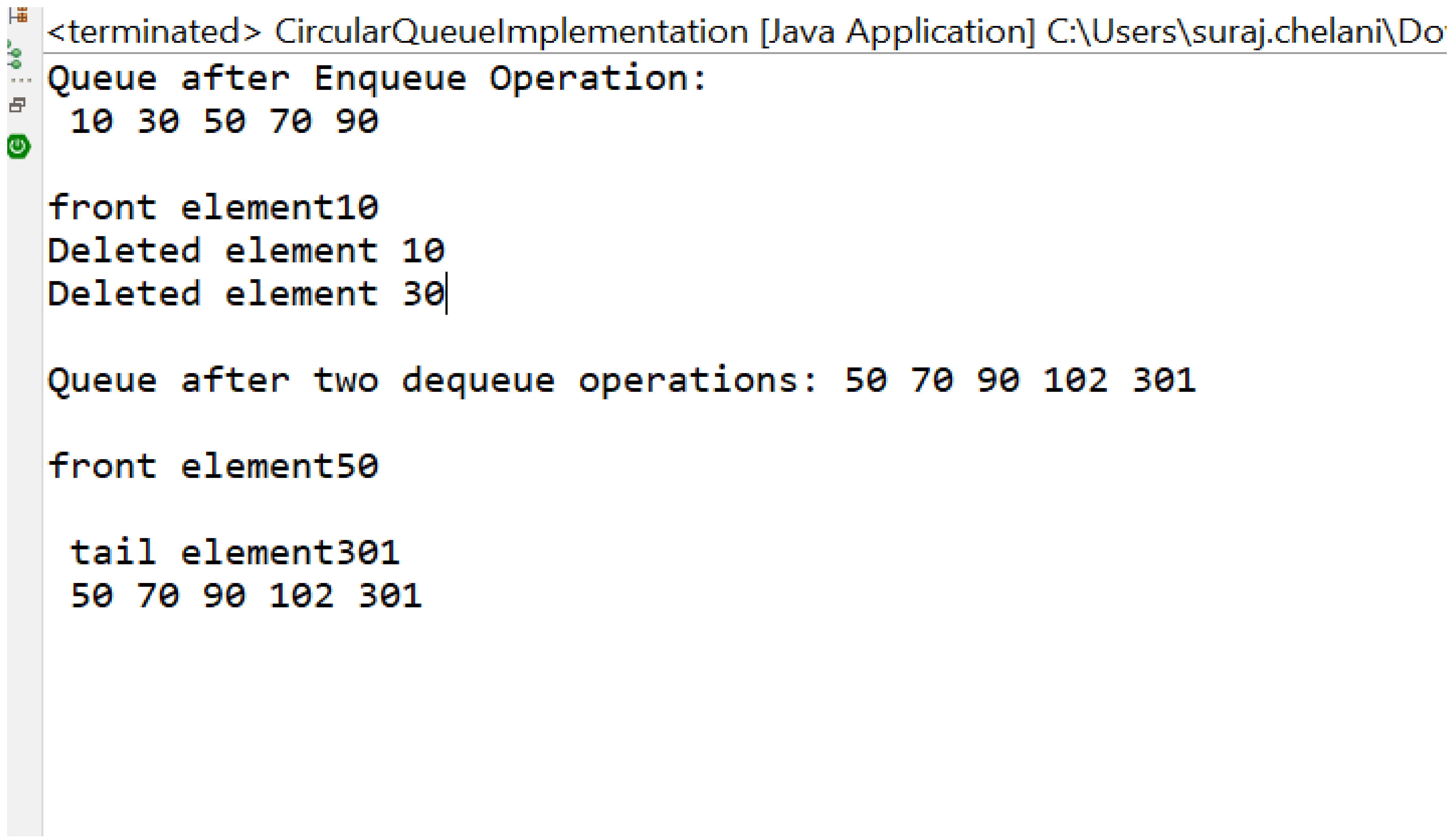


```java
public static void main (String args[])
{
    CircularQueueImplementation q = new CircularQueueImplementation(5);



        // inserting elements in the queue
        q.enqueue(10);
        q.enqueue(30);
        q.enqueue(50);
        q.enqueue(70);
        q.enqueue(90);
        System.out.println("Queue after Enqueue Operation:");
        q.queueDisplay();
        System.out.println ("\nfront element"+ q.front());
        q.dequeue();
        q.dequeue();
```



```java
//checked for circular queue functionality working or not

  q.enqueue(102);
  q.enqueue(301);
  System.out.printf("\nQueue after two dequeue operations:");

  q.queueDisplay();


  System.out.println("\nfront element"+ q.front());


  System.out.println("\n tail element"+ q.tail());
  q.queueDisplay();
```

Output:-

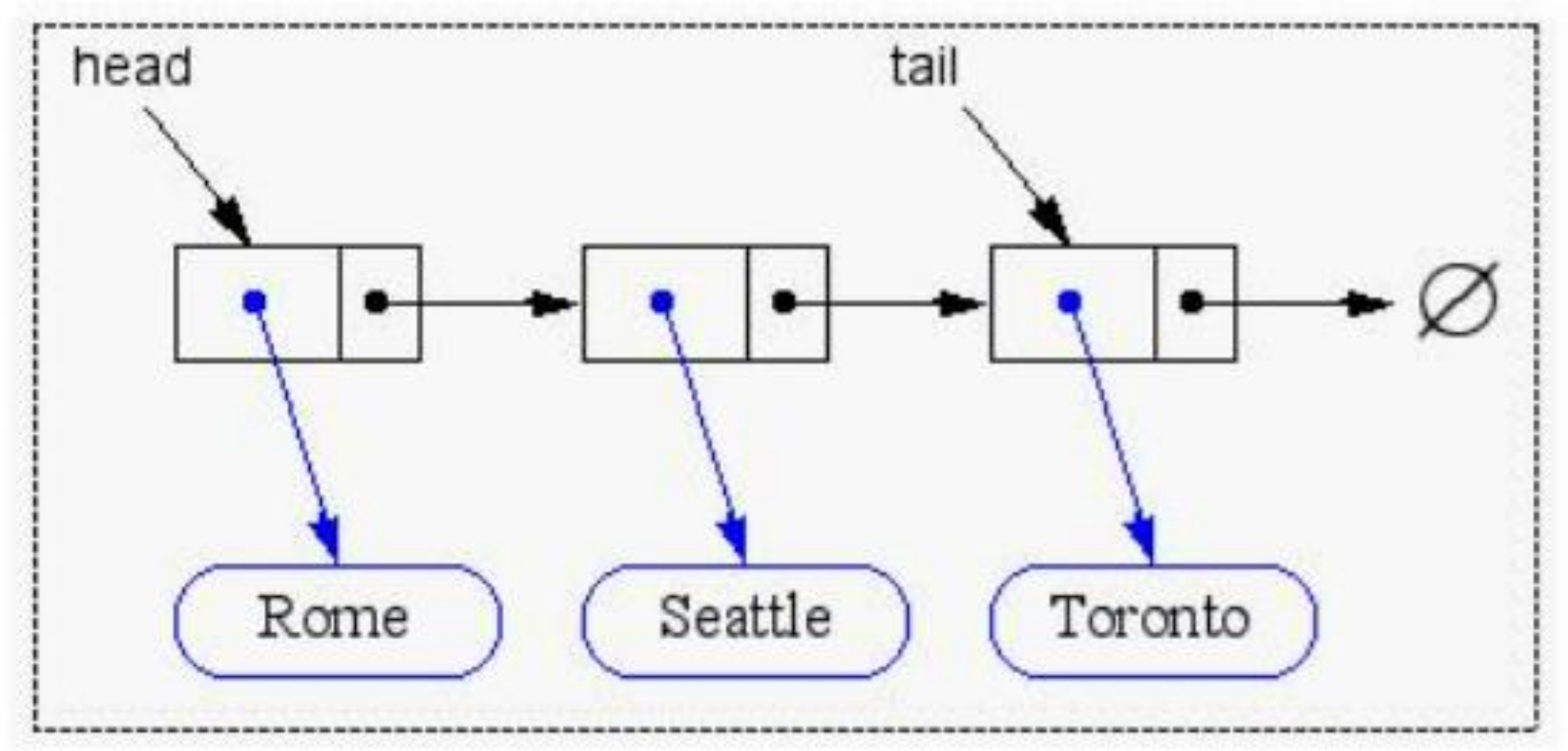


```
<terminated> CircularQueueImplementation [Java Application] C:\Users\suraj.chelani\Do
Queue after Enqueue Operation:
 10 30 50 70 90

front element10
Deleted element 10
Deleted element 30

Queue after two dequeue operations: 50 70 90 102 301

front element50

 tail element301
 50 70 90 102 301
```

Queue with Linklist:-

In a Queue data structure, we maintain two pointers, **front**, and **rear**.
The **front** points to the first item of the queue and **rear** points to the last item.

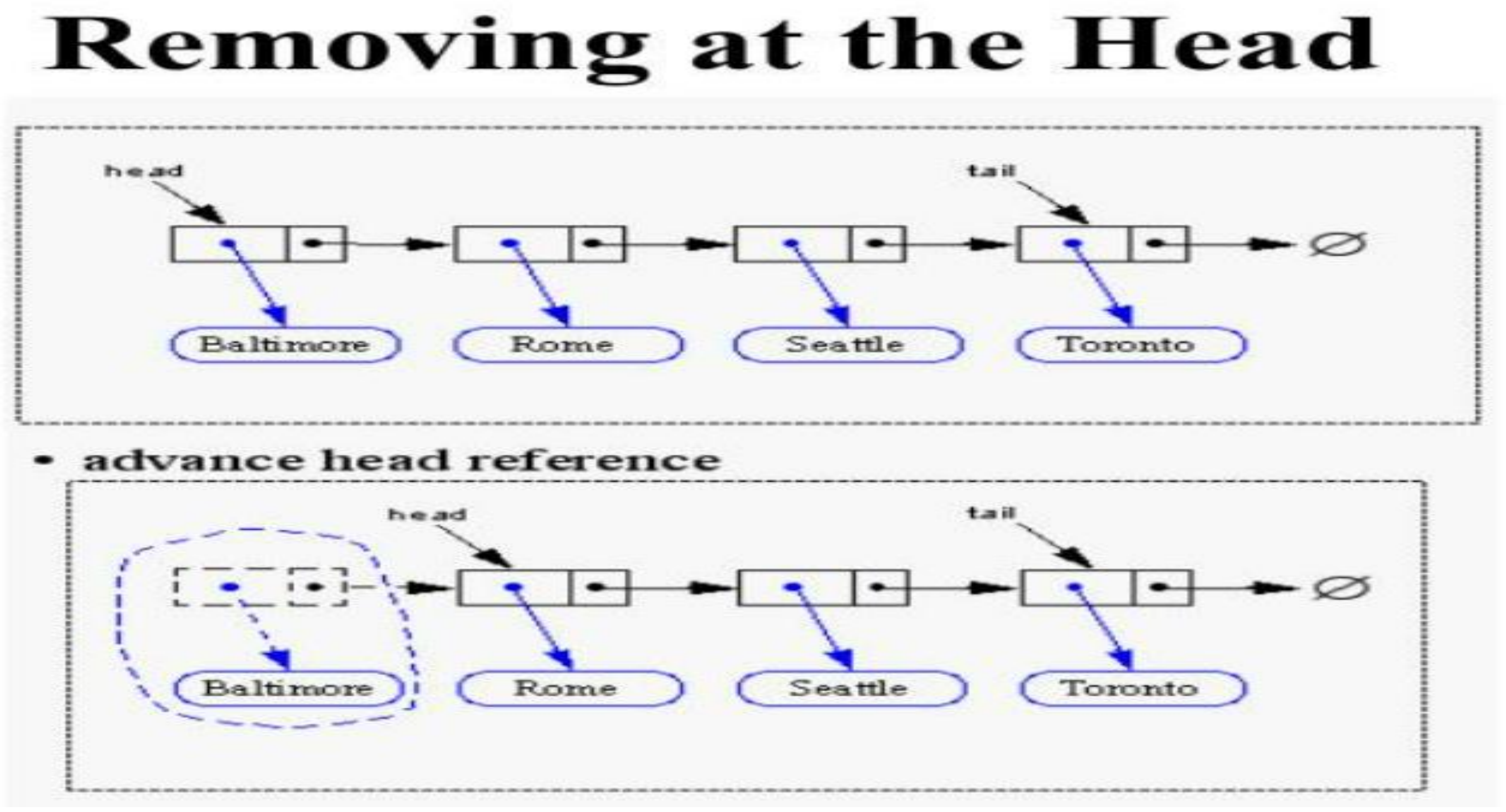
- **enQueue()** This operation adds a new node after **rear** and moves **rear** to the next node.
- **deQueue()** This operation removes the front node and moves **front** to the next node.



Operation on Queue with Linklist:-

Performing Dequeue In Queue with link list.

Time Complexity: O(1)

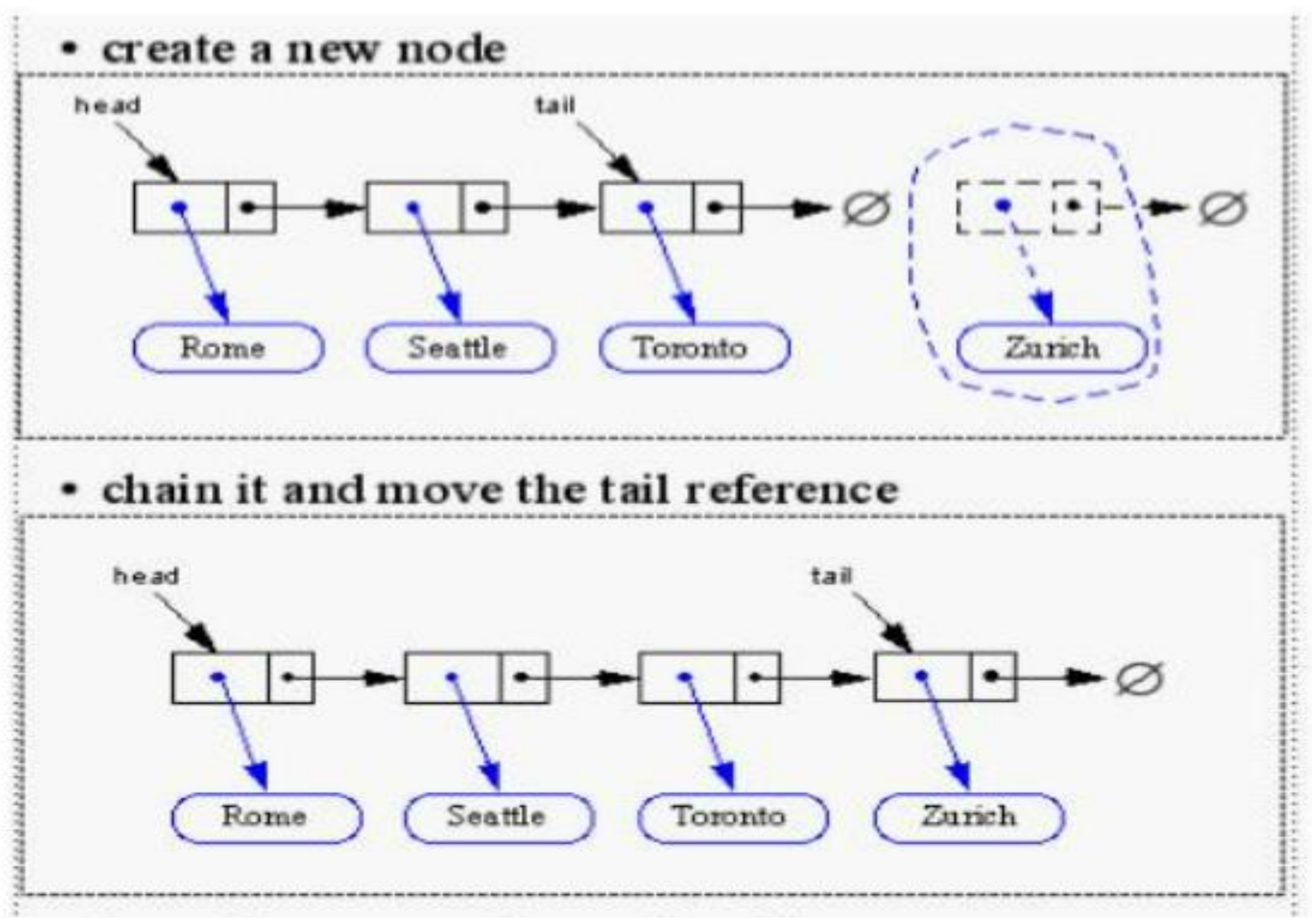


IMPETUS

Enqeue():-

In Enqueue we add an element to the tail of the link list .
Here tail is the rear of queue.
Time Complexity: O(1)

Inserting at the Tail



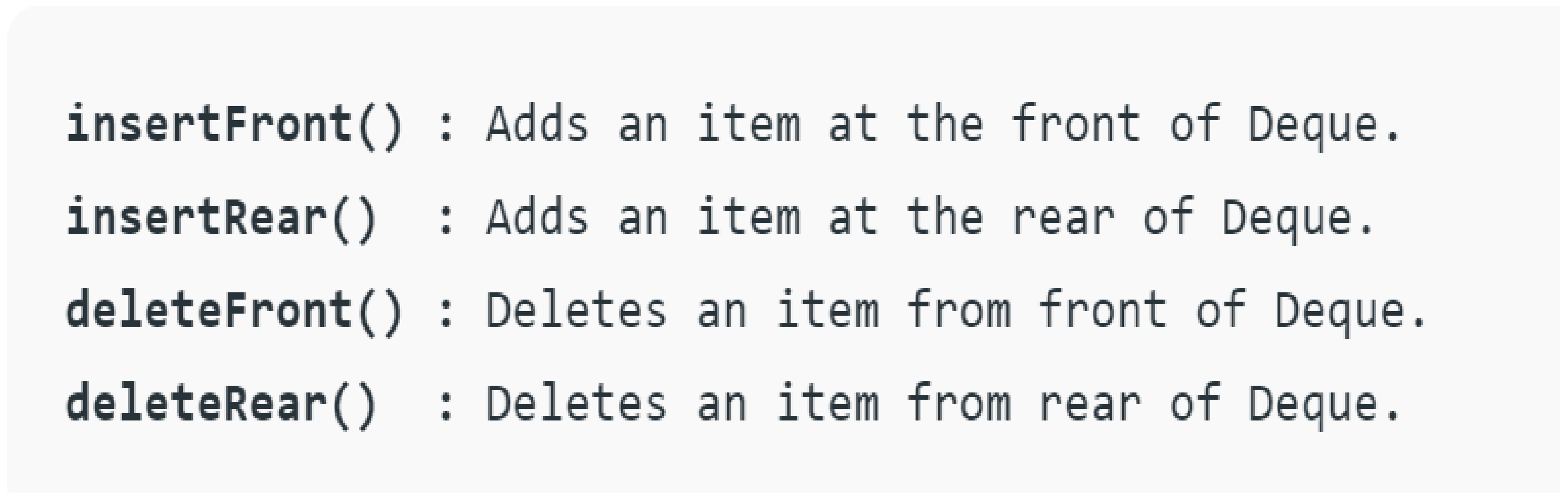
So ,what About deletion at the tail of queue can we perform it?

Double-Ended Queue:-

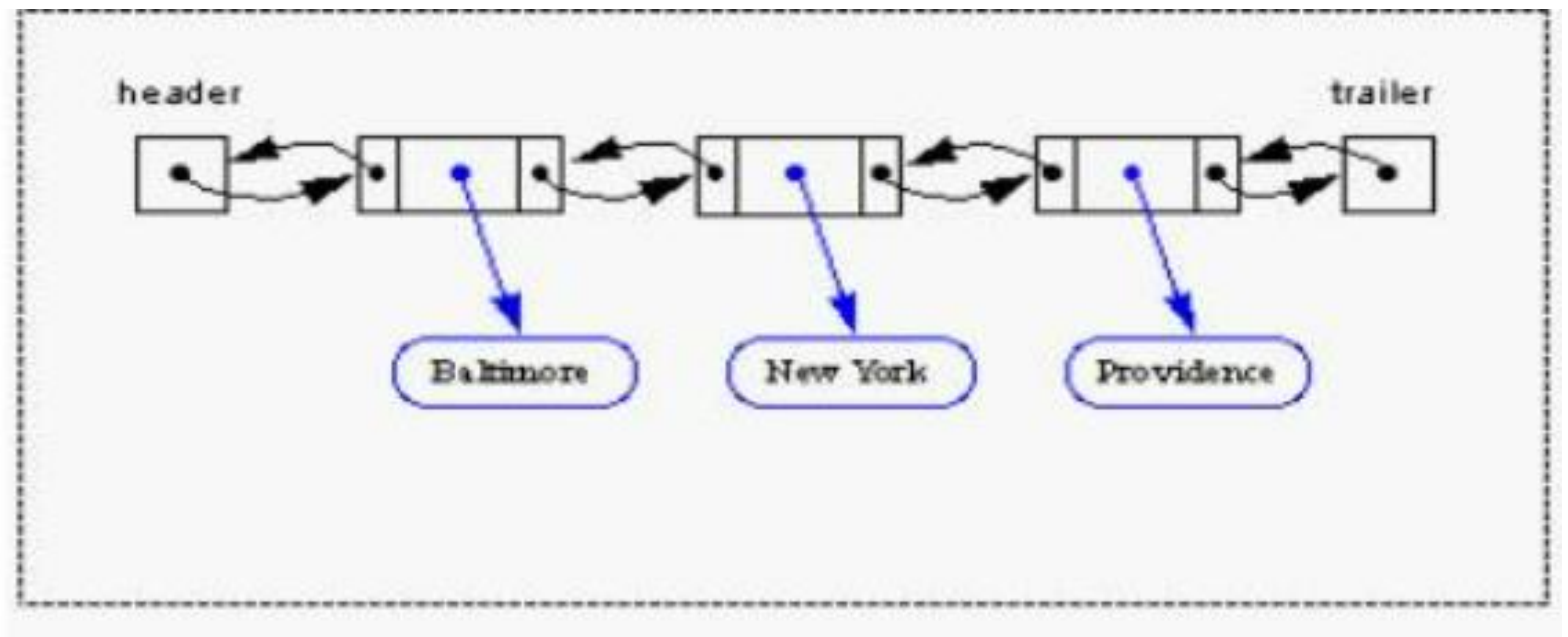
Double-ended queue is an abstract data type. A deque represents a linear collection of elements that support insertion, retrieval and removal of elements at both ends. Dequeue, often abbreviated to deque.

Operations on Deque :

Mainly the following four basic operations are performed on queue :



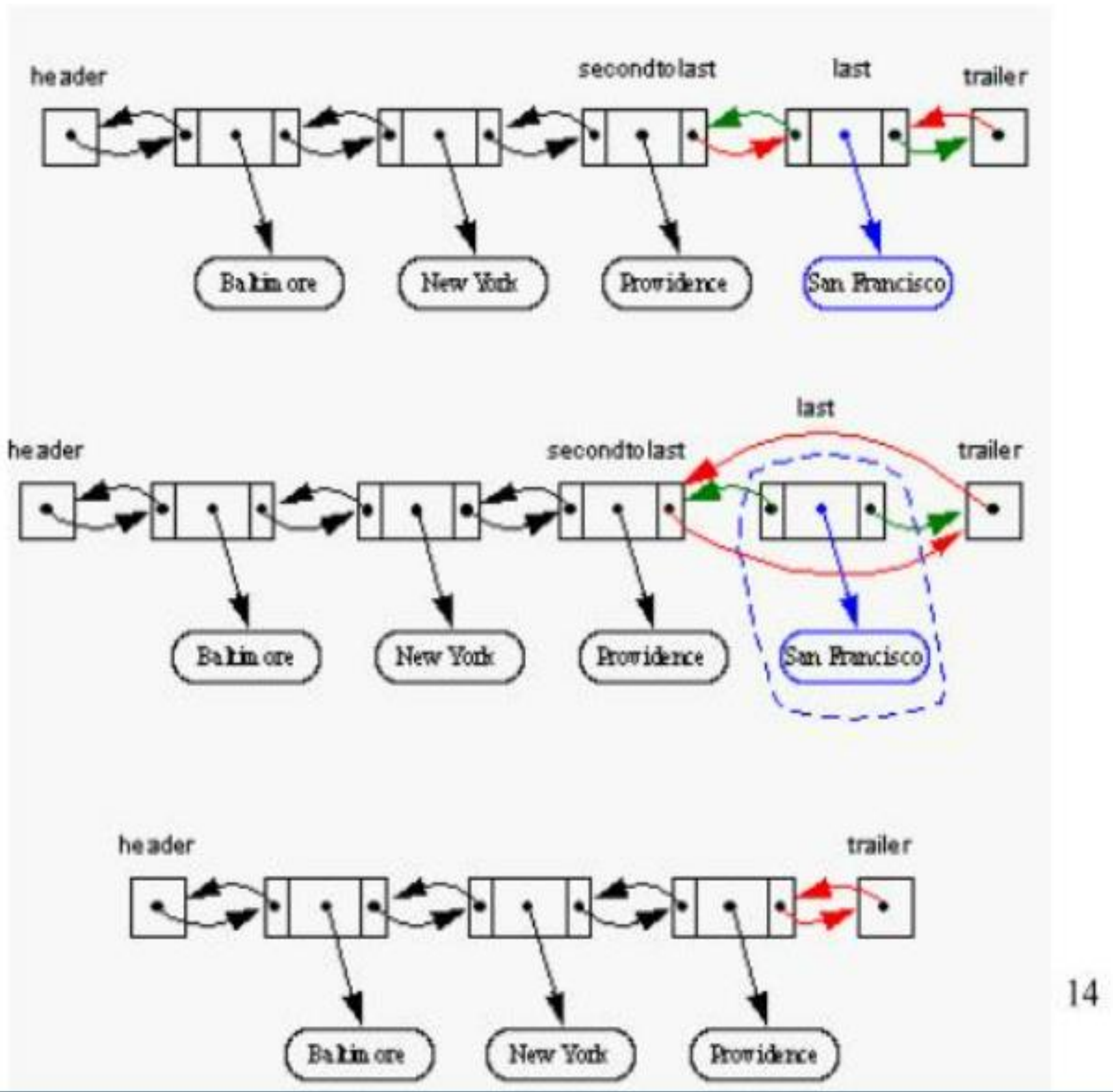
```
insertFront() : Adds an item at the front of Deque.
insertRear()  : Adds an item at the rear of Deque.
deleteFront() : Deletes an item from front of Deque.
deleteRear()  : Deletes an item from rear of Deque.
```



IMPETUS

Operations:-

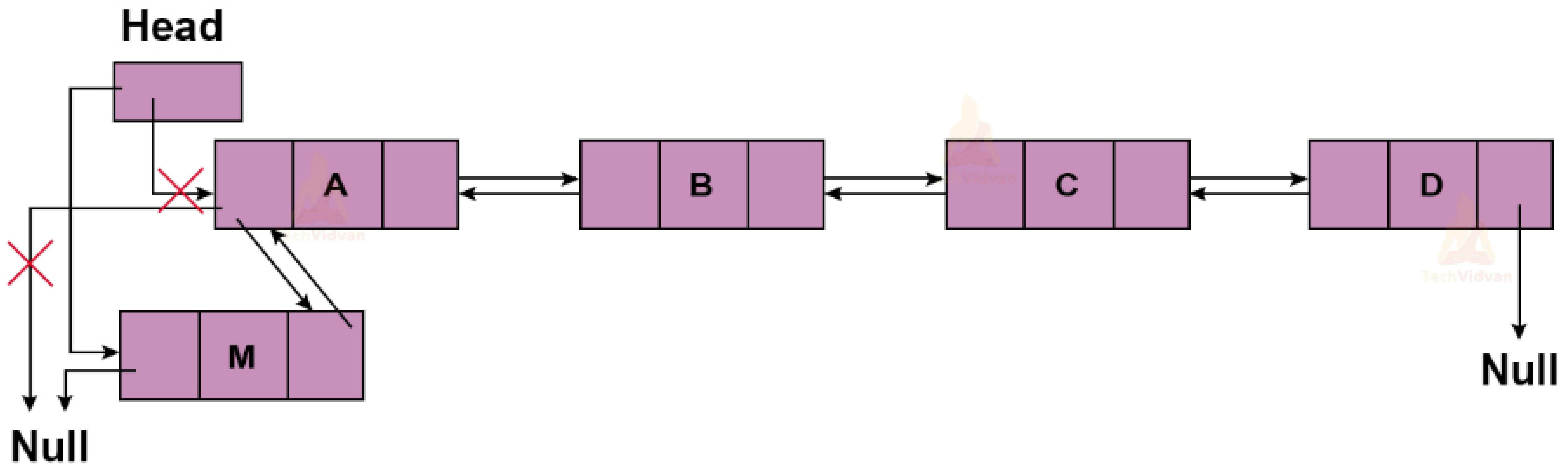
- Deletion of the last element of the queue
- Here is the Explanation of the working of deleteRear() method
- **Time Complexity:** O(1)



InsertFront():-

Below is the diagrammatic explanation of Insertion at first of queue.

Time Complexity: O(1)

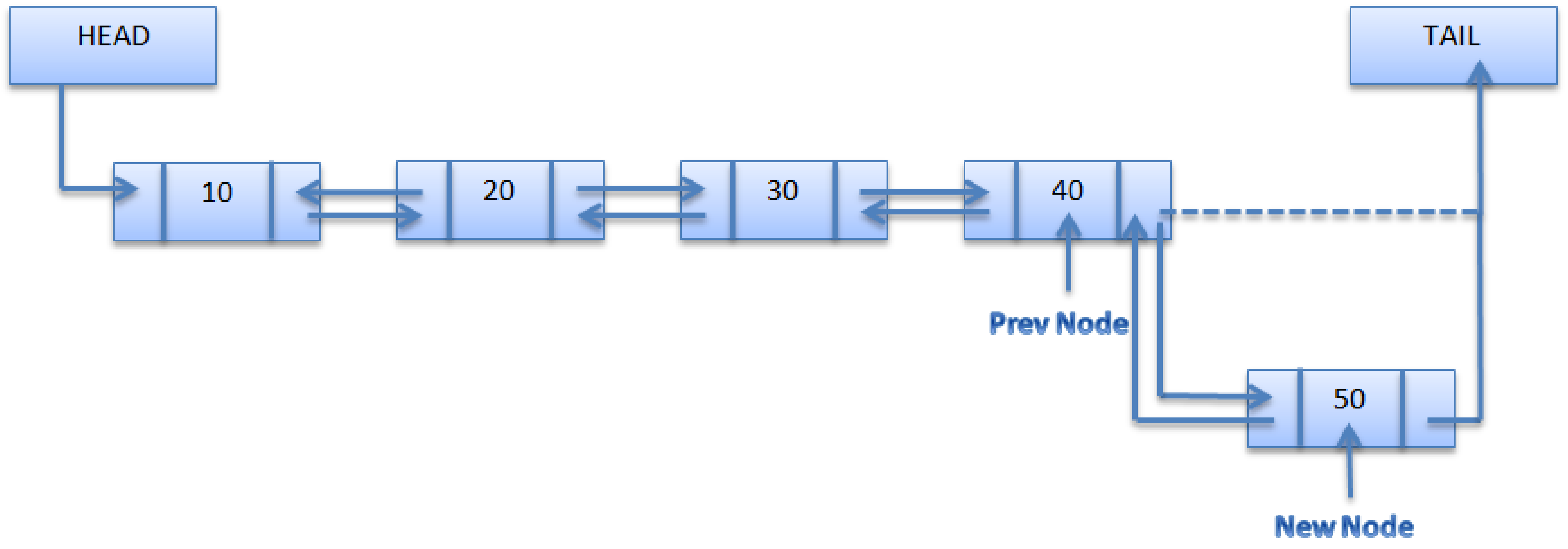


IMPETUS

InsertRear():-

Below is the diagrammatic explanation of Insertion at last of queue.

Time Complexity: O(1)

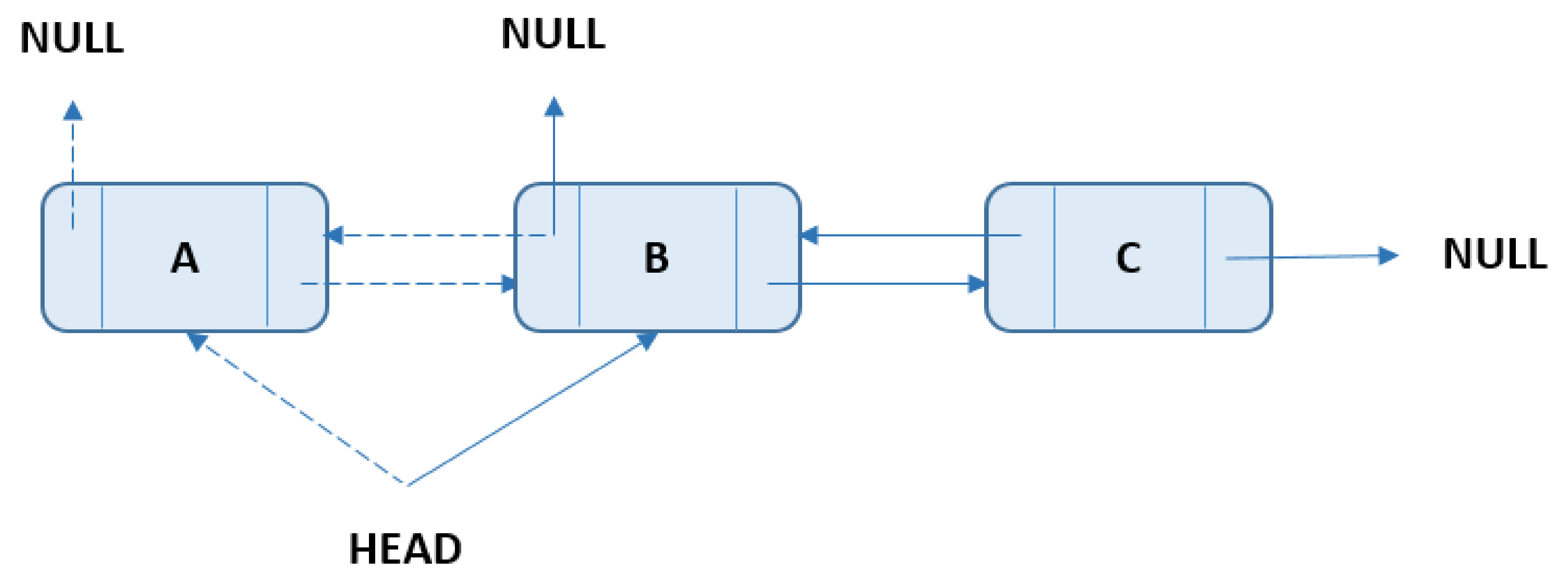


IMPETUS

DeleteFront():-

Below is the diagrammatic explanation of Insertion at last of queue.

Time Complexity: O(1)



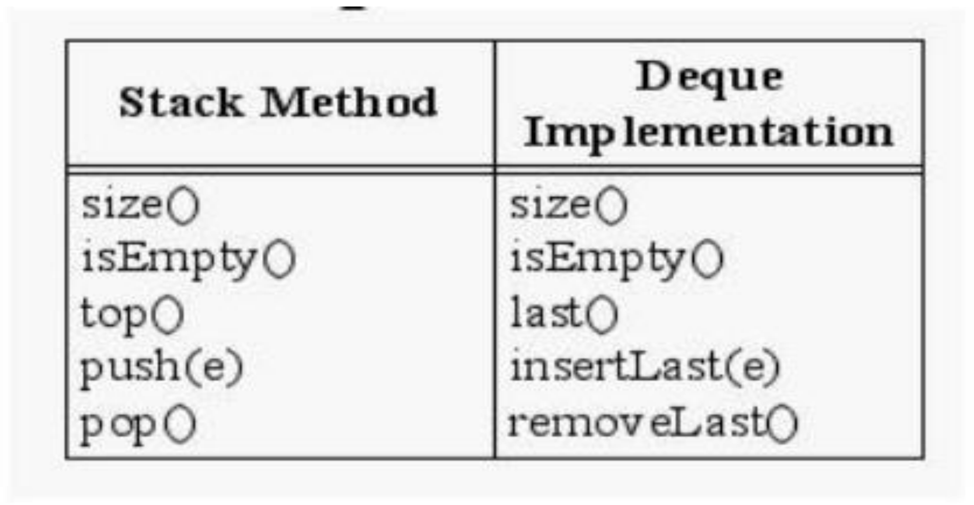
A node of Double linklist has prev and next link .
So , all the methods of the dequeue have a constant time complexity which is O(1)

The Adapter Pattern:-

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

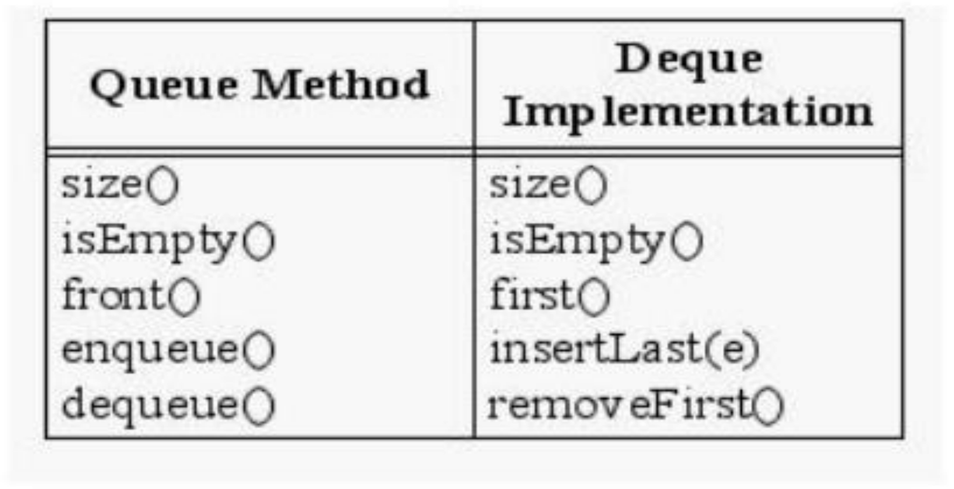
- ➢ Using a Dequeue to implement stack or queue is an example of adapter pattern.
- ➢ Dequeue can act as a stack or queue according to the requirement.

Stacks with Deques:



Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

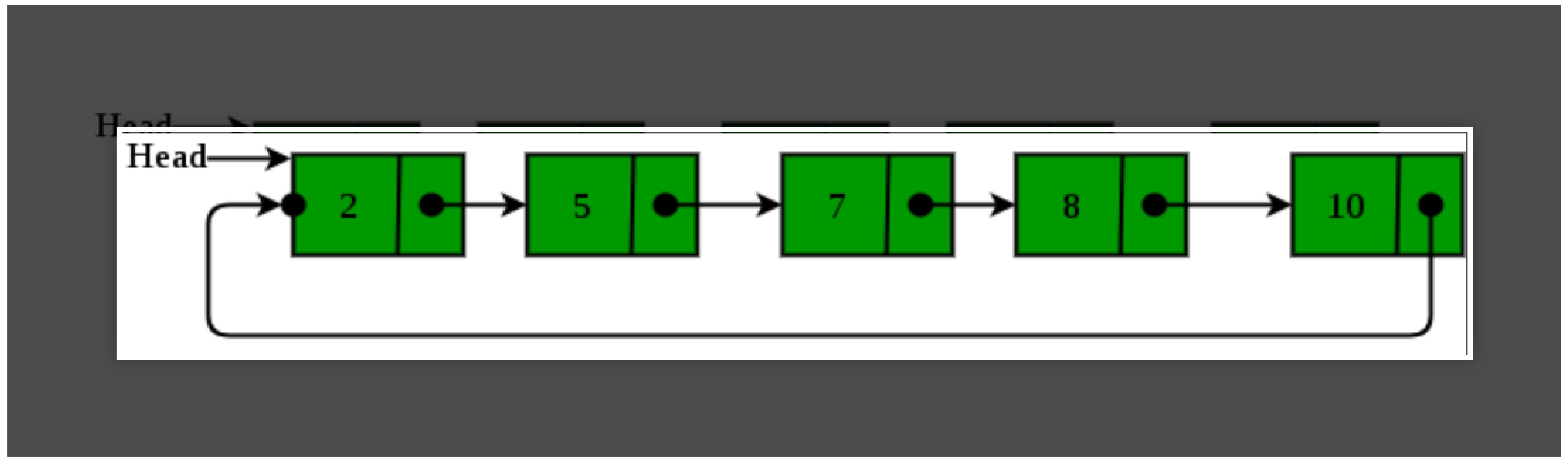
Queues with Deques:



Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

Circular LinkList:-

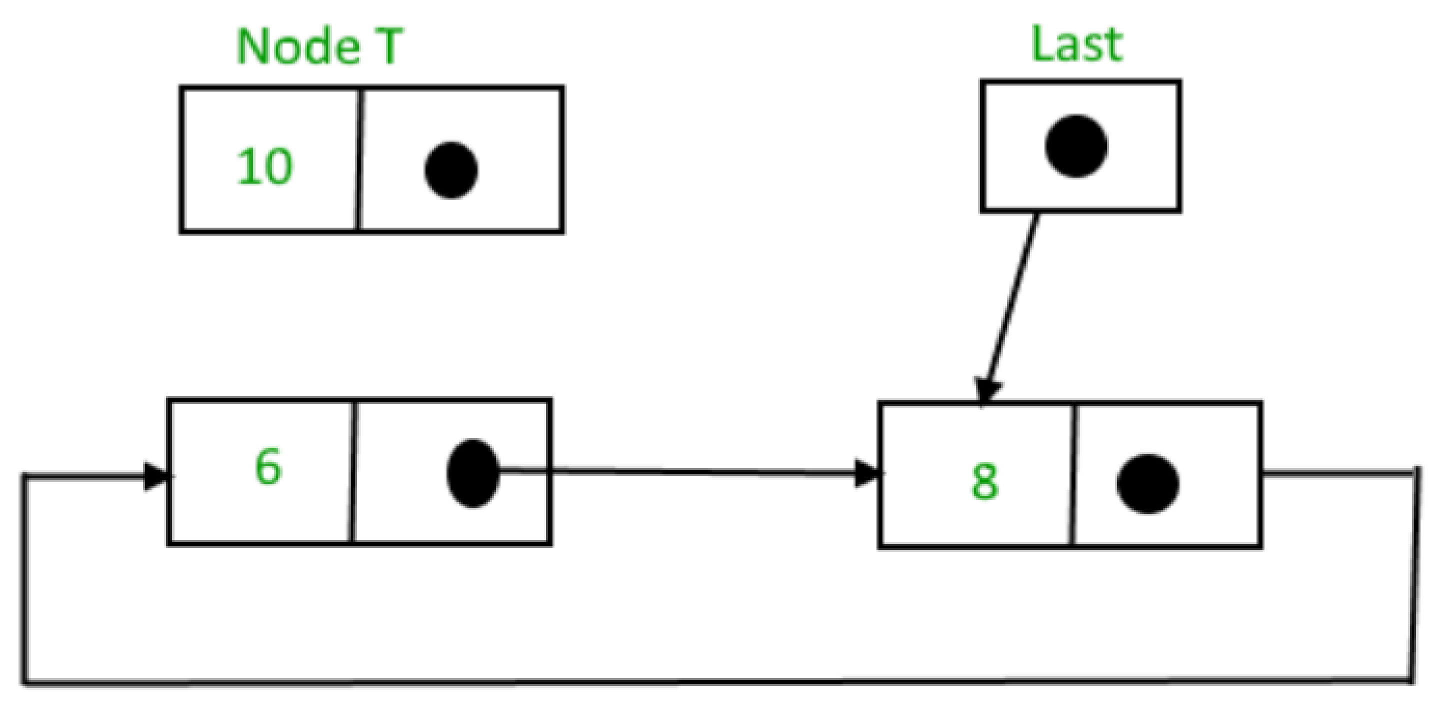
*The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.*



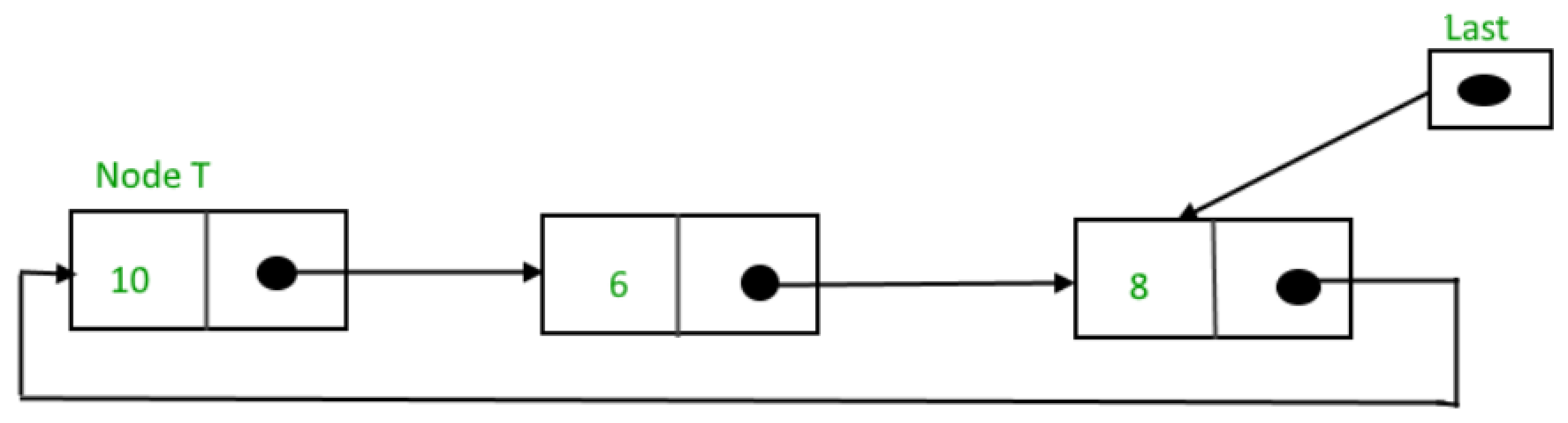
We can use Circular linklist as well to implement Queue.

Insert At Beginning:-

- **Time Complexity:** O(1)



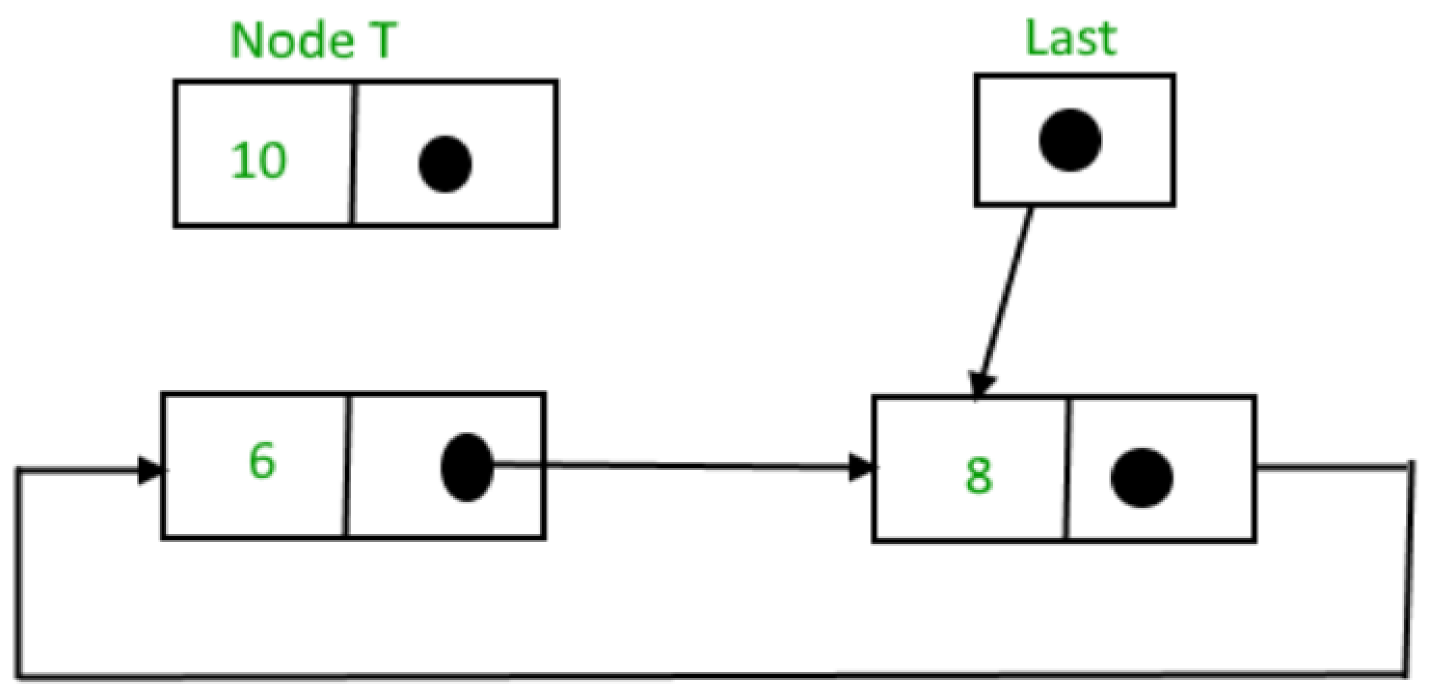
Circular linked list before insertion



Circular linked list after insertion

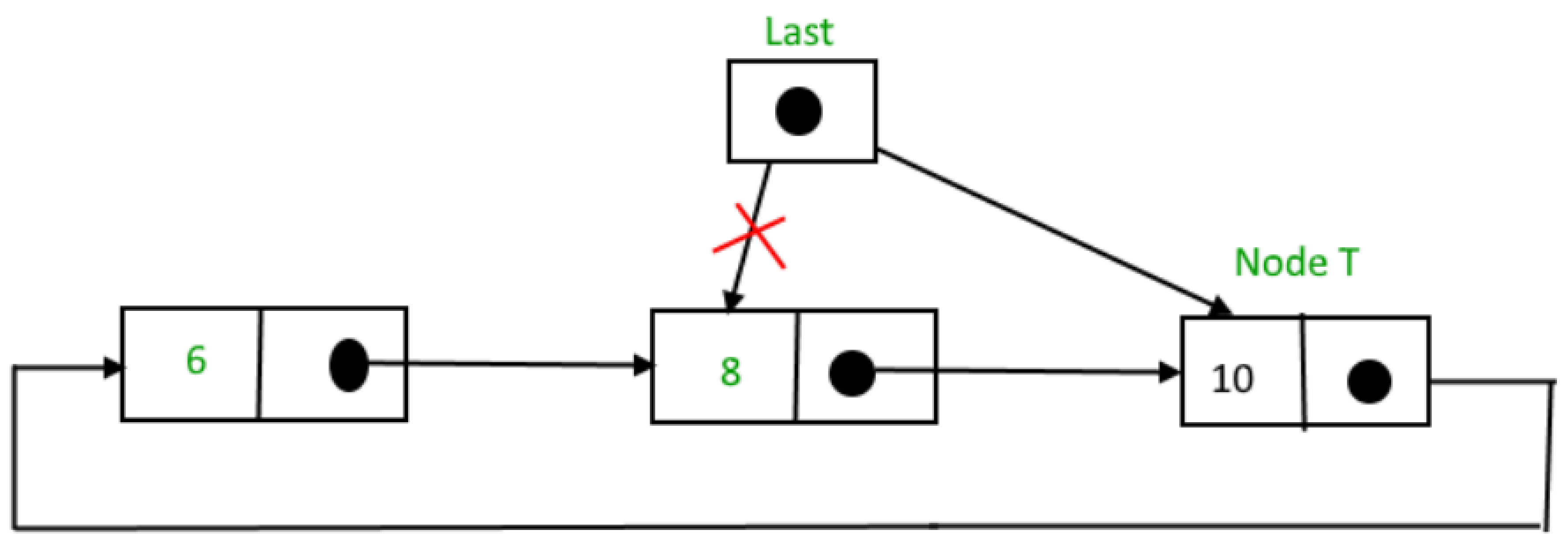
Insert At End:-

- **Time Complexity:** O(1)



Circular linked list before insertion

After insertion,

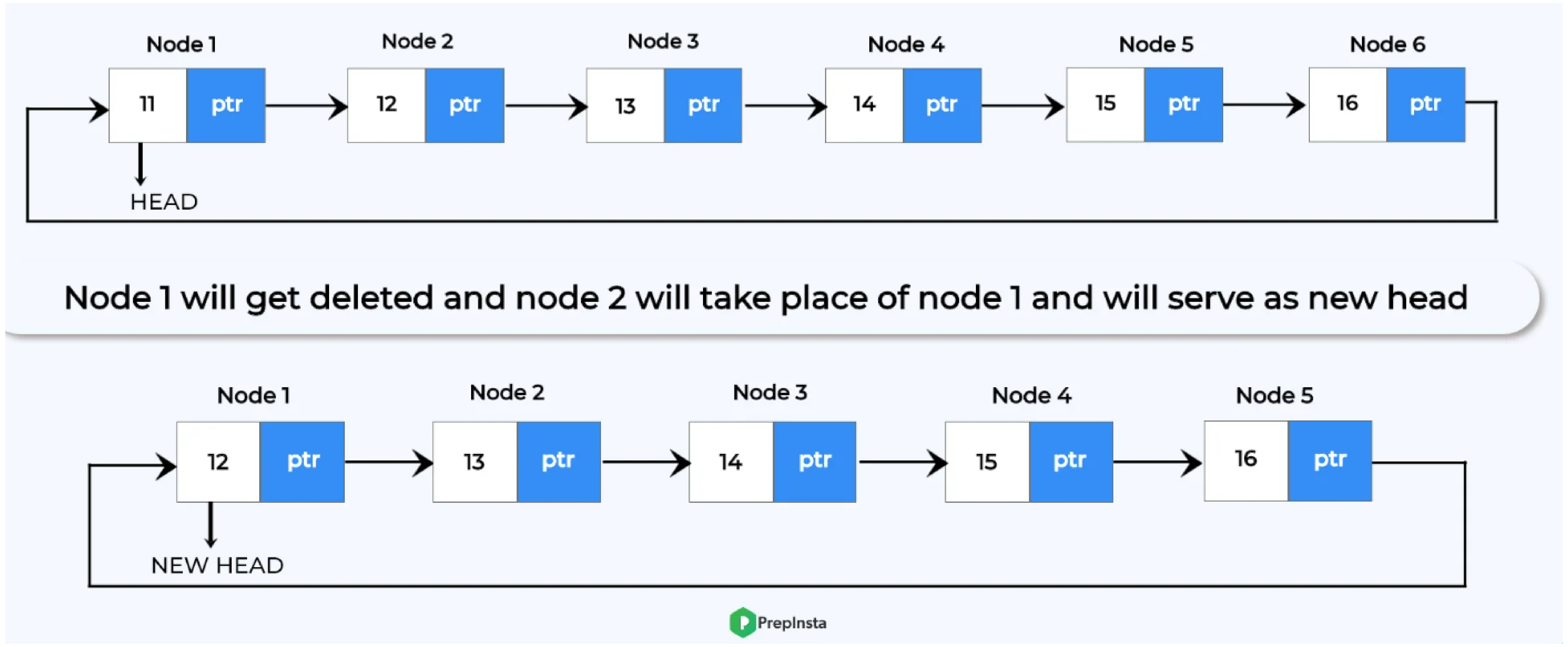


Circular linked list after insertion of node at the end

IMPETUS

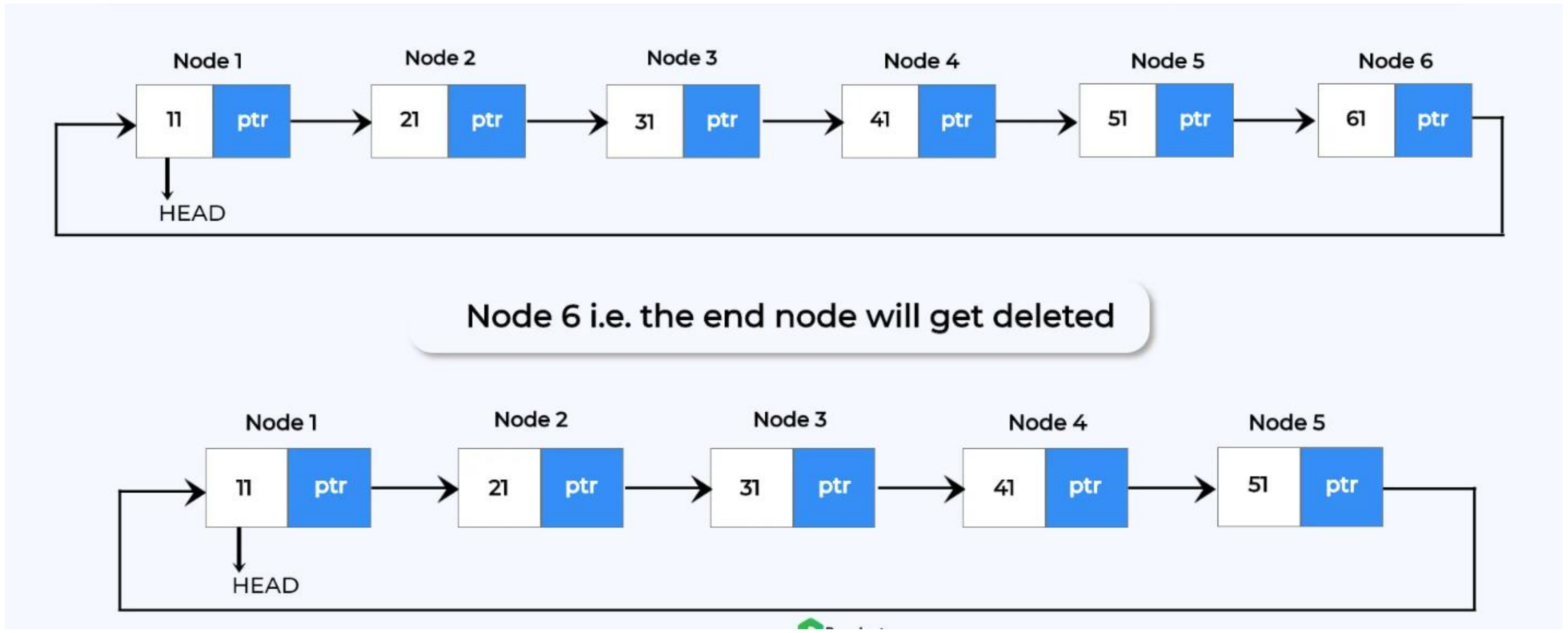
deletion At Beginning:-

- **Time Complexity:** O(1)


Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
11
12
13
14
15
16
ptr
HEAD
Node 1 will get deleted and node 2 will take place of node 1 and will serve as new head
NEW HEAD
PrepInsta

deletion At End:-

- **Time Complexity:** O(n)



Thank You!

IMPETUS