Experiment No. : 4

Title: Implementation of Circular Queue

Batch:A2 **Roll No.:16010421063** **Experiment No.: 4**

Aim: Write a menu driven program to implement a static cicular queue using supporting following operations.

1. Create empty queue,
2. Insert an element on the queue,
3. Delete an element from the queue,
4. Display front element
5. Display all elements of the queue.

Resources Used: Turbo C/ C++/JAVA editor and compiler.

Theory:

What is Linear Queue?

A linear queue is an ordered list in which insertion and deletion happen two different ends. The insertion happens from the *rear* and the deletion takes place at the *front*. It works with the FIFO concept i.e. first in first out. Basic operations of Queue are enqueue, dequeue, isempty, etc.

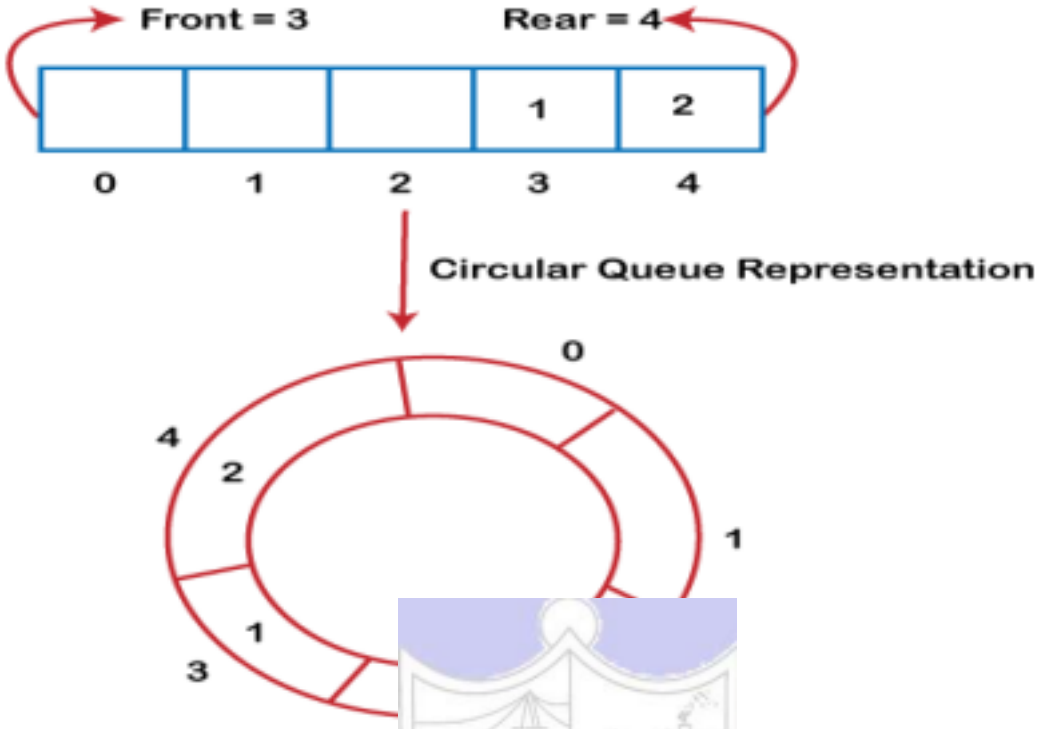
What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of Linear Queue:

If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



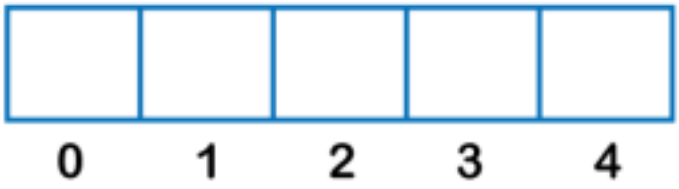
As we can see in the above image, the rear is at the last position of the

Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

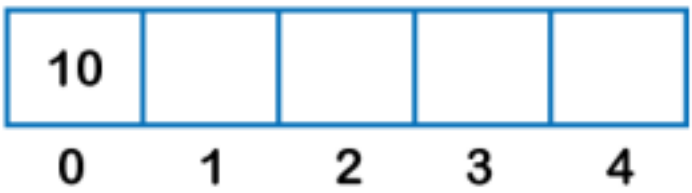
- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **Enqueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **Dequeue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.



Let' ugh the diagrammatic rep

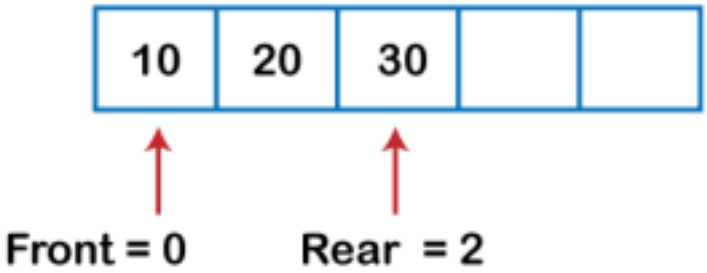
Front = -1
Rear = -1

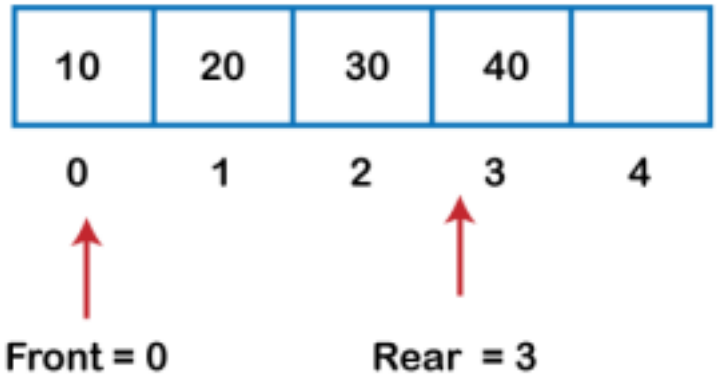
-

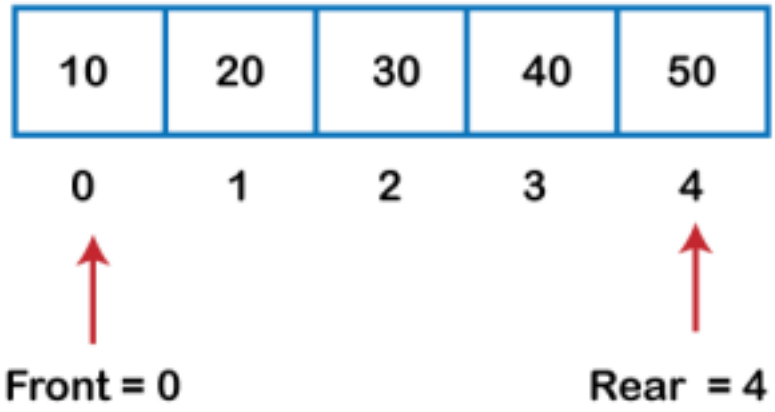


Front = 0

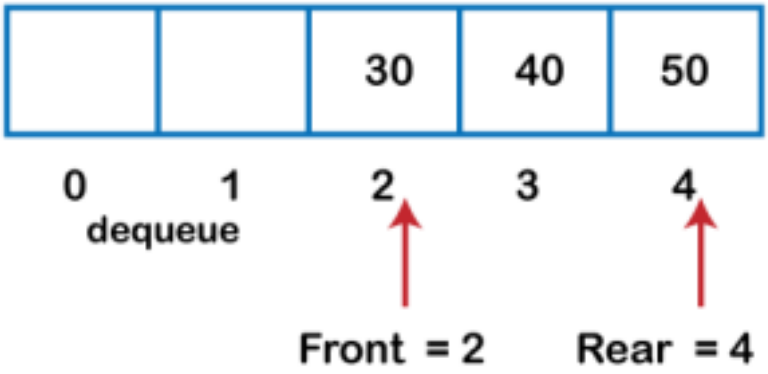
Rear = 0

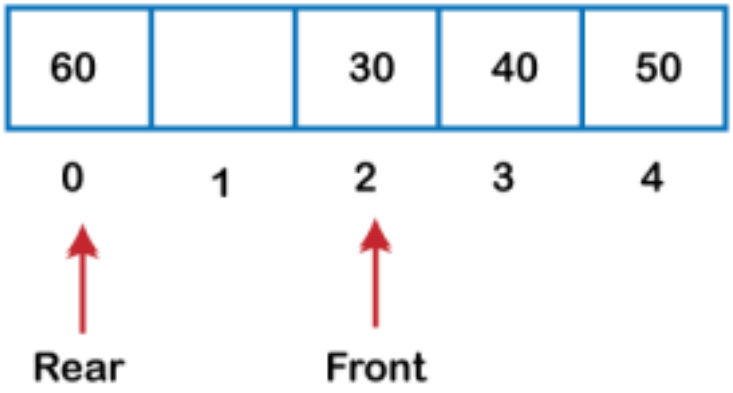


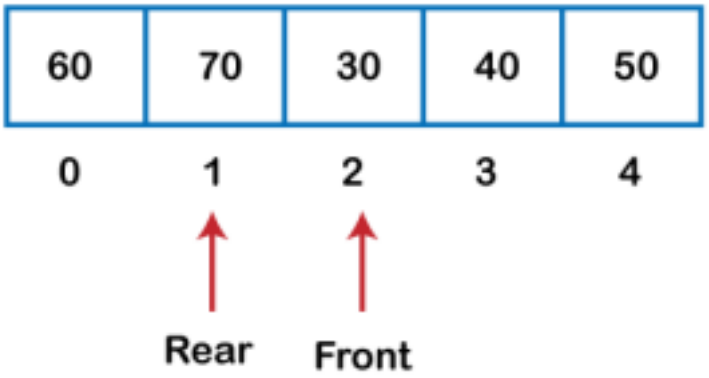



10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4

K. J. SOMAIYA COLLEGE OF ENGG.


30
40
50
0
1
2
3
4
dequeue
Front = 2
Rear = 4





Steps for Implementing Queue Operations

Enqueue(value) and Dequeue() are the primary operations of the queue, which allow you to manipulate the data flow. These functions do not depend on the number of elements inside the queue or its size; that is why these operations take constant execution time, i.e., O(1) [time complexity]. Here, you will deal with steps to implement queue operations:

1. Enqueue(x) Operation

You should follow the following steps to insert (enqueue) a data element into a circular queue -

Step 1: Check if the queue is full (Rear + 1 % Maxsize = Front)

Step 2: If the queue is full, there will be an Overflow error

Step 3: Check if the queue is empty, and set both Front and Rear to 0

Step 4: If Rear = Maxsize - 1 & Front != 0 (rear pointer is at the end of the queue and front is not at 0th index), then set Rear = 0

Step 5: Otherwise, set Rear = (Rear + 1) % Maxsize

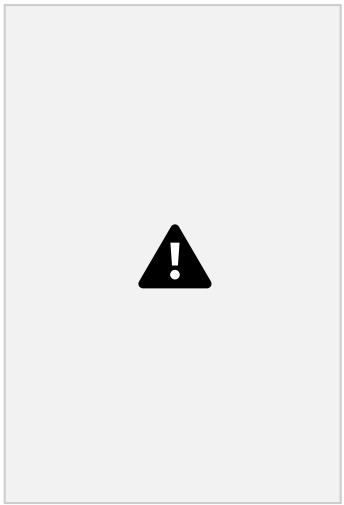
Step 6: Insert the element into the queue (Queue[Rear] = x)

Step 7: Exit

Now, you will explore the Enqueue() operation by analyzing different cases of insertion in the circular queue:

2. Dequeue() Operation

Obtaining data from the queue comprises two subtasks: access the data where the front is pointing and remove the data after access. You should take the following steps to remove data from a circular queue -



Step 1: Check if the queue is empty (Front = -1 & Rear = -1)

Step 2: If the queue is empty, Underflow error

Step 3: Set Element = Queue[Front]

Step 4: If there is only one element in a queue, set both Front and Rear to -1 (IF Front = Rear, set Front = Rear = -1)

Step 5: And if Front = Maxsize -1 set Front = 0

Step 6: Otherwise, set Front = Front + 1

Step 7: Exit

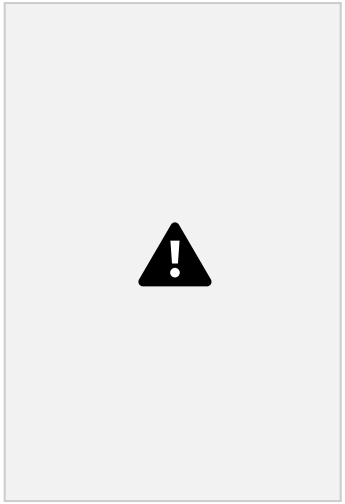
Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case

of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Activity: Students are expected to  implement circular queue using **array.**

NOTE : All functions should be able to handle boundary(exceptional) conditions.

Results: A program implementing solution depicting the correct behaviour of circular queue and capable of handling all possible exceptional conditions and the same is reflecting clearly in the output.

Outcome:

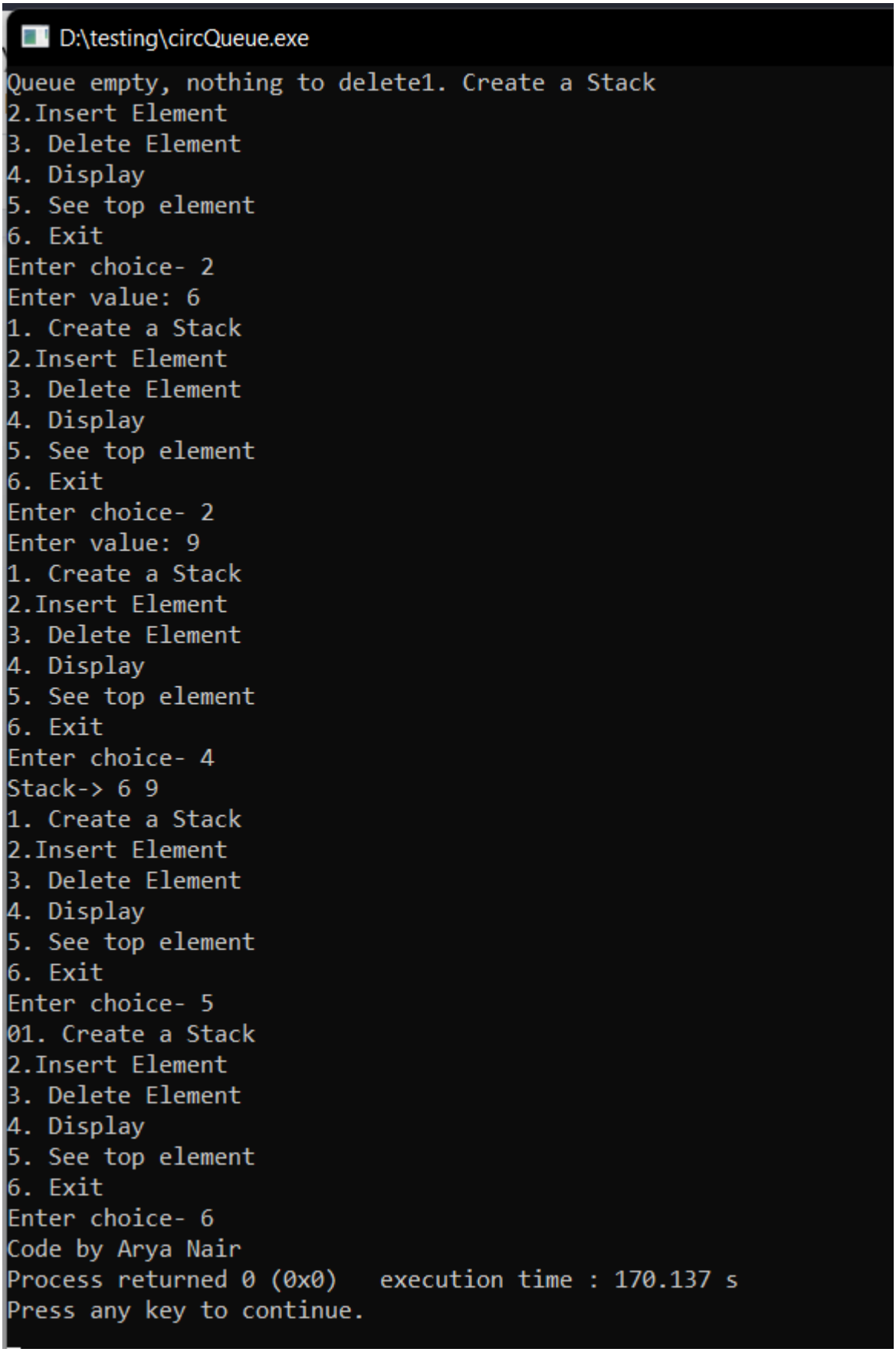
```c
#include<stdio.h>

struct Queue
{
    int front;
    int rear;
    unsigned capacity;
    int* array;
};

int full(struct Queue* q)
{
    if((q->front==q->rear+1)|| (q->front ==0 &&
q->rear==q->capacity-1)){
        return 1;
    }
    return 0;
}
int empty(struct Queue* q)
{
    if(q->front==-1)
    {
        return 1;
    }
    return 0;
}

struct Queue* createQueue(unsigned capacity)
{
    struct Queue* queue=(struct Queue*)malloc(sizeof(struct Queue));
    queue->front=-1;
    queue->rear=-1;
    queue->capacity=capacity;
    queue->array=(int*)malloc(capacity*sizeof(int));
    return queue;
};

void display(struct Queue* q)
{
    if(empty(q))
    {
        printf("queue is empty\n");return;
```

```
    }
    int i;
    printf("Stack-> ");
    for(i=q->front;i!=q->rear;i=(i+1)%q->capacity)
    {
        printf("%d ",q->array[i]);
    }
    printf("%d\n",q->array[i]);

}



void enqueue(int value,struct Queue* q)
{
    if(full(q)){
        printf("Queue full, cant add\n");
        return;
    }
    else{
        if(q->front==-1)
        {
            q->front=0;
        }
        q->rear=(q->rear+1)%q->capacity;
        q->array[q->rear]=value;
    }
}



void dequeue(struct Queue* q)
{
    if(empty(q))
    {
        printf("Queue empty, nothing to delete");
        return;
    }
    int answer=q->array[q->front];
    if(q->front==q->rear)
    {
        q->front=-1;
```

```
        q->rear=-1;
    }
    else{
        q->front=(q->front+1)%q->capacity;
    }
    printf("Deleted Element= %d\n",answer);
    return;
}

void peek(struct Queue* q)
{
    if(empty(q)){
        printf("Queue Empty\n");
        return;
    }
    printf("%d",q->front);
    return;
}


int main()
{
    int choice=0;
    struct Queue* q;

    while(choice!=6){
        printf("1. Create a Stack\n2.Insert Element\n3. Delete
Element\n4. Display\n5. See top element\n6. Exit\nEnter choice- ");

        scanf("%d",&choice);
        switch(choice){
        case 1:{
            printf("Enter the size of array: ");
            int size;
            scanf("%d",&size);
            q=createQueue(size);
            break;
        }
        case 2:{
            printf("Enter value: ");
            int value;
            scanf("%d",&value);
```

```
            enqueue(value,q);
            break;
        }
        case 3:{
            dequeue(q);
            break;
        }
        case 4:{
            display(q);
            break;
        }
        case 5:
            {
                peek(q);
                break;
            }
        case 6:
            {
                printf("Code by Arya Nair");
                break;
            }
        default:{

            printf("Enter valid choice\n");
            break;
        }

        }
    }
}
```

```
Enter choice- 2
Enter value: 4
Queue full, cant add
1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 3
Deleted Element= 5
1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 3
Deleted Element= 8
1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 3
Queue empty, nothing to delete1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice-
```


D:\testing\circQueue.exe
Queue empty, nothing to delete1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 2
Enter value: 6
1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 2
Enter value: 9
1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 4
Stack-> 6 9
1. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 5
01. Create a Stack
2.Insert Element
3. Delete Element
4. Display
5. See top element
6. Exit
Enter choice- 6
Code by Arya Nair
Process returned 0 (0x0)   execution time : 170.137 s
Press any key to continue.

Conclusion: We implemented a Circular queue and while doing so also understood the advantage of memory saving while using a Circular queue instead of the normal queue while implementing a queue with an array. Also tested edge cases such as not being able to insert element when the array is full, not being able to delete when array is empty and not being able to see top element when array is empty

.

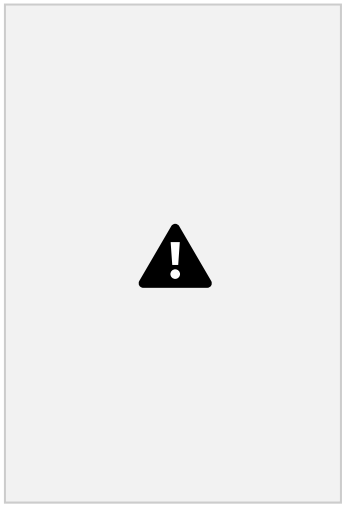
Grade: AA / AB / BB / BC / CC / CD /DD:

Signature of faculty in-charge with date :

References:

Books/ Journals/ Websites:

• Y. Langsam, M. Augenstin and A. Tannenbaum, "Data Structures using C", Pearson Education Asia, 1st Edition, 2002.



(A Constituent College of Somaiya Vidyavihar University)