**Batch: A3 Roll No.: 1911034**

**Experiment / assignment / tutorial No. 6**

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

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title:**  Implementation of Different operations on Queue data structure (Simple Queue, Circular Queue, Dequeue) |

**Objective:** To Understand and Implement Create, Insert, Delete, Search and Destroy operations on Queues.

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **CO** | **Outcome** |
| 1 | Explain the different data structures used in problem solving |

**Books/ Journals/ Websites referred:**

1. *Fundamentals Of Data Structures In C –* Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications –* Jean Paul Tremblay,

Paul G. Sorenson

1. *Data Structures A Pseudo Approach with C –* Richard F. Gilberg & Behrouz A. Forouzan
2. [https://www.geeksforgeeks.org/queue-data-structure/](about:blank)
3. [https://www.thecrazyprogrammer.com/2019/01/types-of-queues-in-data-structure.html](about:blank)

**Abstract**:

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is FIFO, Queues are first in first out type of data structures.

Ordered list of elements in which we add elements only at one end, called Rear end of the queue and delete elements only at the other end, called Front end of the queue.

Queues can be implemented using both Arrays and Linked Lists.

The Types of Queues are-

1. Simple Queue
2. Circular Queue
3. Doubly Ended Queue or Dequeue
4. Priority Queue

**Related Theory: -**

**Simple Queue**

1. A simple queue is a normal queue (FIFO Data structure) where the insertion takes place at one end of the queue (rear end) and is called as enqueue and the deletion tales place at the other end (front end) of the queue and the operation is called as dequeue operation.
2. The elements in a simple queue are arranged in a sequential order and thus the queue is said to be a linear data structure.
3. Its applications are process scheduling, disk scheduling, memory management, IO buffer, pipes, call center phone systems, and interrupt handling

**Circular Queue**

1. In a circular queue , the insertions and deletions are performed according to the FIFO principle, but the last element is connected to the first element to make it in a circular form.
2. It is also called as a ring buffer
3. **Circular Queues** offer a quick and clean way to store FIFO data with a maximum size. Conserves memory as we only store up to our capacity (opposed to a **queue** which could continue to grow if input outpaces output.)

**Doubly Ended Queue or Dequeue**

In a doubly ended queue , two pointers are maintained , one to the front and the other to the end , and the insertions/deletions can take place from both ends.

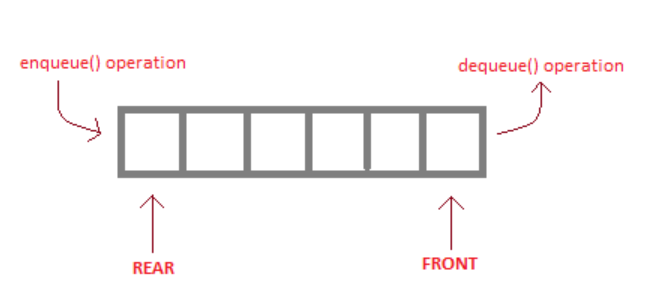
There are two variants of the doubly ended queue:

1. Input restricted dequeue (insertion takes place only at one end and deletions at both ends)
2. Output restricted dequeue (deletion takes place only at one end and insertions at both ends)

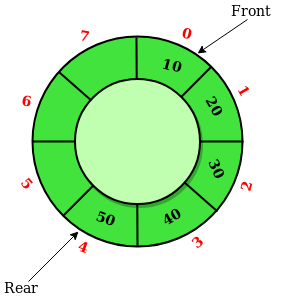
In the computer’s memory, a deque is implemented using either a **circular array** or a **circular doubly linked list**.

**Diagram for :**

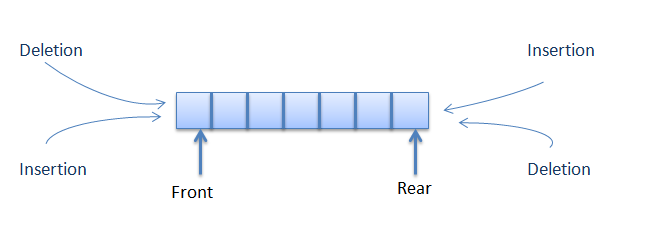
**Simple Queue**



**Circular Queue**



**Doubly Ended Queue or Dequeue**



**Algorithm for Operations performed on the Queue assigned to you:**

**FOR DEQUEUE:**

Let f = pointer to first element of deque r= pointer to last element of deque

MAX = max capacity of dequeue, a[] represents the array that holds the queue

**Algorithm for Inserting at Rear:**

Step1: Input value to be entered (val) from the user .

Step2: If (f=0 AND r=-1) OR (f=r+1) , then the queue is full , so element cannot be inserted. Go to step 6

Step3: else if (f=-1), queue is initially empty , so initialize f=0 and r=0

Step4: else if (r=MAX-1), set r=0 , else set r = r+1;

Step5: set a[r]= val;

Step6: exit

**Algorithm for inserting at the front:**

Step1: Input value to be entered (val) from the user .

Step2: If (f=0 AND r=-1) OR (f=r+1) , then the queue is full , so element cannot be inserted. Go to step 6

Step3: else if (f=-1), queue is initially empty , so initialize f=0 and r=0

Step4: else if (f=0), set f=MAX-1 , else set f= f-1;

Step5: set a[f]= val;

Step6: exit

**Algorithm for deleting from rear:**

Step1: If (f=-1), the queue is empty and hence the element cannot be deleted. Go to step 5

Step2: As the element to be deleted is the rear element , print a[r] before deleting

Step3: if (f=r), queue has only one element , thus set f=-1 and r=-1 to delete that element

Step4: else if (r=0) , set r= MAX-1 , else set r =r-1, to delete the element from the rear (in this case queue has more than one element)

Step5: exit

**Algorithm for deleting from front:**

Step1: If (f=-1), the queue is empty and hence the element cannot be deleted. Go to step 5

Step2: As the element to be deleted is the front element , print a[f] before deleting

Step3: if (f=r), queue has only one element , thus set f=-1 and r=-1 to delete that element

Step4: else if (f=MAX-1) , set f=0 , else set f =f+1, to delete the element from the rear (in this case queue has more than one element)

Step5: exit

**Algorithm for Displaying the Queue**.

Step1: if (f=-1), queue is empty , thus go to step 4

Step 2: if (f<=r), then queue is not empty , initialize a point p = f and iterate it till p=r , thus printing all elements from f to r.

Step3: else if (f>r) initialize a pointer (p=f) and iterate it till p = MAX-1 , thus printing all elements from f to MAX-1, then set p =0 and iterate p till p =r , thus printing elements from 0 to r.

Step4: exit

**Algorithm to find whether an element exists or not (searching):**

Step1: if (f=-1), queue is empty , thus go to step 4

Step 2: if (f<=r), then queue is not empty , initialize a pointer i=f , and initialize flag=0 and repeat step3 till a[i]!=val(to be found) all elements from f to r.

Step3: if a[i] == val, print (‘element found’) ,set flag=1 and end loop and go to step 9. Else , set i=i+1.

Step4: else if (f>r) initialize a pointer (i=f) and repeat step5 till i = MAX-1 , thus searching all elements from f to MAX-1.

Step5: if a[i] == val, print (‘element found’) ,set flag=1 and end loop and go to step9. Else , set i=i+1.

Step 6: set i=0, and repeat step 7 till i!= rear.

Step7: if a[i] == val, print (‘element found’) ,set flag=1 and end loop and go to step9. Else , set i=i+1.

Step8: if flag=0, print (‘element not found , go to step 9’)

Step9 exit.

**Implementation Details:**

1. **Enlist all the Steps followed and various options explored.**

**Various steps followed were as follows :**

1. In this , the queue was implemented using an array
2. The queue has been implemented using two pointers , front and rear , which point to the front and rear elements of the queue , and can be adjusted while inserting or deleting an element
3. The program is menu driven which allows the user to insert , or delete from any end , at any step.
4. Every function of the queue (inserting at front /rear and deleting from front/rear) was implemented using a function that accepts three parameters , an array and the front and rear pointers of the queue, in case of Doubly Ended Queue.
5. In circular queue , the elements are always inserted at the rear end and deleted from the front end

**Options explored:**

1. The queue can be implemented using a linked list or an array. We have also tried implementing it using a linked list , but in this experiment ,we have provided array based implementation.
2. We have also explored the option of adding/deleting elements to the front and rear ends of the queue according to any dynamic option that may be provided by the user at runtime by implementing a switch-case based program. The order of insertion/deletion will be decided by the user.

**List the Queue Assigned to you-**

The queue assigned to me was Dequeue (Doubly Ended Queue). In this queue , insertions and deletions can take place at both ends of the queue.

Also , another queue assigned to us was Circular Queue , where insertions always take place at the rear end , and deletions take place at the front.

**Assumptions made for Input:**

1. As the queue was implemented using an array , it has a fixed size and therefore it cannot grow indefinitely.
2. We have assumed that user will provide an integer value.

**Built-In Functions Used:**

No built-in functions were used.

**Program source code for Operations performed on the Queue assigned to you :**

#include<stdio.h>

#define MAX 10

void insert\_rear(int \*f,int \*r, int a[]);

void insert\_front(int \*f,int \*r,int a[]);

void delete\_rear(int \*f,int \*r, int a[]);

void delete\_front(int \*f,int \*r, int a[]);

void display(int \*f,int \*r, int a[]);

void search(int \*f,int \*r,int a[]);

void destroy(int \*f,int \*r, int a[]);

int main()

{

int ch;

int f =-1; int r=-1;

int a[MAX];

do

{

printf("\n1.Insert at rear");

printf("\n2.Insert at front ");

printf("\n3.Delete from rear ");

printf("\n4.Delete from front ");

printf("\n5.Display ");

printf("\n6.Search for an element");

printf("\n7.Destroy");

printf("\n-1 to exit");

printf("\n\nEnter your choice ");

scanf("%d",&ch);

switch(ch)

{

case 1:

insert\_rear(&f,&r,a);

break;

case 2:

insert\_front(&f,&r,a);

break;

case 3:

delete\_rear(&f,&r,a);

break;

case 4:

delete\_front(&f,&r,a);

break;

case 5:

display(&f,&r,a);

break;

case 6:

search(&f,&r,a);

break;

case 7:

destroy(&f,&r,a);

}

}while(ch!=8);

return 0;

}

void insert\_rear(int \*f, int\*r, int a[])

{

int val;

printf("\nEnter the value to be added ");

scanf("%d",&val);

if( (\*f==0 && \*r==MAX-1) || (\*f==\*r+1) )

{

printf("\nOVERFLOW");

}

if(\*f==-1) //if queue is initially empty

{

\*f=0;

\*r=0;

}

else

{

if(\*r==MAX-1)

\*r=0;

else

\*r=\*r+1;

}

a[\*r]=val;

}

void insert\_front(int\*f, int\*r,int a[])

{

int val;

printf("\nEnter the value to be added ");

scanf("%d",&val);

if( (\*f==0 && \*r==MAX-1) || (\*f==\*r+1) )

{

printf("\nOVERFLOW");

}

if(\*f==-1) //if queue is initially empty

{

\*f=0;

\*r=0;

}

else

{

if(\*f==0)

\*f=MAX-1;

else

\*f=\*f-1;

}

a[\*f]=val;

}

void delete\_rear(int \*f, int \*r,int a[])

{

if(\*f==-1)

{

printf("\nUNDERFLOW");

return;

}

printf("\nThe deleted element is %d\n", a[\*r]);

if(f==r) //Queue has only one element

{

\*f=-1;

\*r=-1;

}

else

{

if(\*r==0)

\*r=MAX-1;

else

\*r=\*r-1;

}

}

void delete\_front(int \*f, int \*r, int a[])

{

if(\*f==-1)

{

printf("\nUNDERFLOW");

return;

}

printf("\nThe deleted element is %d\n", a[\*f]);

if(\*f==\*r) //Queue has only one element

{

\*f=-1;

\*r=-1;

}

else

{

if(\*f==MAX-1)

\*f=0;

else

\*f=\*f+1;

}

}

void display(int \*f, int\*r, int a[])

{

int front=\*f, rear=\*r;

if(front==-1)

{

printf("\nQueue is Empty\n");

return;

}

printf("\nThe elements in the queue are: ");

if(front<=rear)

{

while(front<=rear)

{

printf("%d\t",a[front]);

front++;

}

}

else

{

while(front<=MAX-1)

{

printf("%d\t",a[front]);

front++;

}

front=0;

while(front<=rear)

{

printf("%d\t",a[front]);

front++;

}

}

printf("\n");

}

void search(int \*f,int \*r,int a[])

{

int ele,flag=0,i,count;

int front=\*f, rear=\*r;

int t=front;

printf("enter the element to search\n");

scanf("%d",&ele);

if(front==-1)

{

printf("\nQueue is Empty\n");

return;

}

if(front<=rear)

{

for(i=front;i<=rear;i++)

{count=0;

if(a[i]==ele)

{

printf("element exists\n");

flag=1;

break;

}

count++;

}

}

else

{

for(i=front;i<=MAX-1;i++)

{

count=front;

if(a[i]==ele)

{

printf("element exists\n");

flag=1;break;

}

count++;

}

for(i=0;i<=rear;i++)

{count=0;

if(a[i]==ele)

{

printf("element exists \n");

flag=1;break;

}

count++;

}

}

if(flag==0)

{

printf("element not found\n");

}

}

void destroy(int \*f, int \*r,int a[])

{

int i;

\*f=-1;

\*r=-1;

for(i=0;i<=MAX-1;i++)

{

a[i]=0;

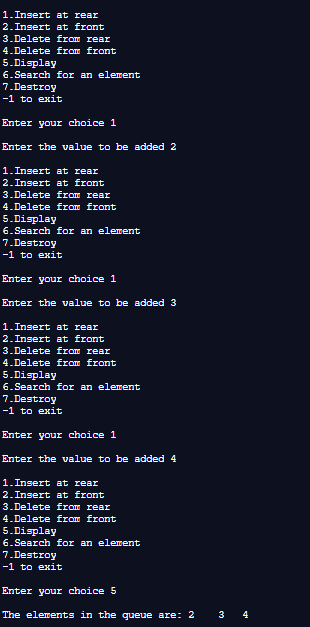
}

printf("queue has been destroyed\n");

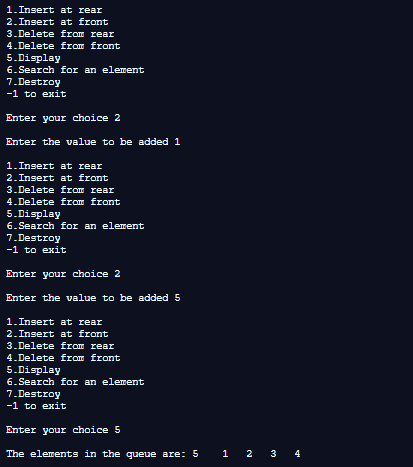
}

**Output Screenshots for Each Operation:**

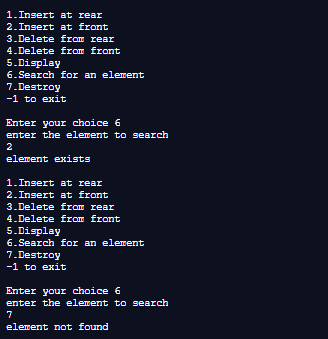
**Inserting at the rear and displaying:**



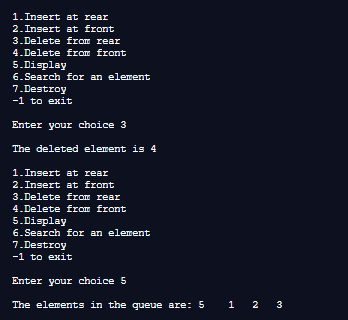
**Inserting at front and displaying :**



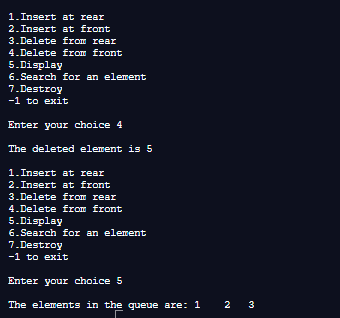
**Finding whether an element exists or not :**



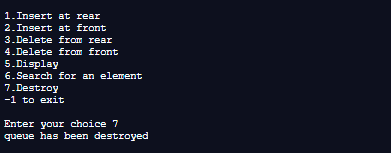
**Deleting from Rear and Displaying :**



**Deleting from front and displaying :**



**Destroying the queue**



**FOR CIRCULAR QUEUE :**

**Algorithms:**

**Let a[] represent the array used to implement the circular queue and let f and r represent the front and rear pointers respectively, MAX represents max capacity of circular queue**

**Algorithm to perform insertion in a circular queue:**

The following cases may arise while implementing insertion in a circular queue using array:

Case 1: if ((f=0) and (r=MAX-1)) or ((r =f-1)), print ‘queue overflow’ and exit.

Case 2: if front=-1, queue is empty , so set front=rear=0.

Case 3: if r = MAX-1, set r=0 , else set r= r+1.

Set a[r] = val.

END

**Algorithm to perform deletion in a circular queue** :

Step1: if (f=-1), print ‘queue underflow’ and exit.

Step2: if (f=r) , the queue contains only one node , so set f=r=-1 and exit.

Step3: if f = MAX-1, set f=0 and exit .(in this case , the front node is the last node , so we set it to 0)

ELSE , set f = f+1.

**Code :**

#include <stdio.h>

//Implementing a Circular Queue - in this , queue is always inserted at the rear end and deleted from the front end

#define MAX 5

void insert\_ele(int \*f, int \*r, int a[])

{

int val;

if((\*f==-1)&&(\*r==-1))//circular Queue is empty

{

printf("Enter the element\n");

scanf("%d",&val);

\*f = 0; \*r=0;

a[\*r]=val;

}

else if (((\*r==MAX-1)&&(\*f==0))||(\*r+1==\*f))

{

printf(" Circular Queue is full\n");

}

else if((\*r==MAX-1)&&(\*f!=0))

{

printf("Enter the element\n");

scanf("%d",&val);

\*r=0;

a[\*r]=val;

}

else

{

printf("Enter the element\n");

scanf("%d",&val);

\*r=\*r+1;

a[\*r]=val;

}

}

void delete\_ele(int \*f,int \*r, int a[])

{

if((\*f==-1)&&(\*r==-1))

{

printf("Queue is empty\n");

}

else if(\*f==\*r)//last element to be deleted

{

\*f=\*r=-1;

}

else

{

if(\*f==MAX-1)

{

\*f=0;

}

else

{

\*f=\*f+1;

}

}

}

void display(int \*f, int \*r, int a[] )

{

int i;

if((\*f==-1)&&(\*r==-1))

{

printf("Queue is empty\n");

}

//case 1: either the queue is full

else if(\*r+1==\*f)

{

for(i=\*f;i<=MAX-1;i++)

{

printf("%d",a[i]);

printf("-->");

}

for(i=0;i<=\*r;i++)

{

printf("%d",a[i]);

if(i!=\*r)

{

printf("-->");

}

}

}

//case 2: queue is again full

else if((\*r==0)&&(\*f==MAX-1))

{

for(i=0;i<=\*r;i++)

{

printf("%d",a[i]);

if(i!=\*r)

{

printf("-->");

}

}

}

//case 3: queue is not full

else {

if(\*r==0)//when the element at index1 of the queue is deleted

{

for(i=\*f;i<=MAX-1;i++)

{

printf("%d",a[i]);

printf("-->");

}

printf("%d",a[\*r]);

}

else {

for(i=\*f;i<=\*r;i++)

{

printf("%d",a[i]);

if(i!=\*r)

{

printf("-->");

}

}

}

}

}

int main(void) {

int a[MAX];

int front =-1; int rear =-1;int ch;

do{

printf("\nEnter 1 to insert an element\n 2 to delete an element\n 3 to display\n-1 to exit\n");

scanf("%d",&ch);

switch(ch)

{

case 1:

{

insert\_ele(&front,&rear,a);

}

break;

case 2:

{

delete\_ele(&front,&rear,a);

}

break;

case 3:

{

display(&front,&rear,a);

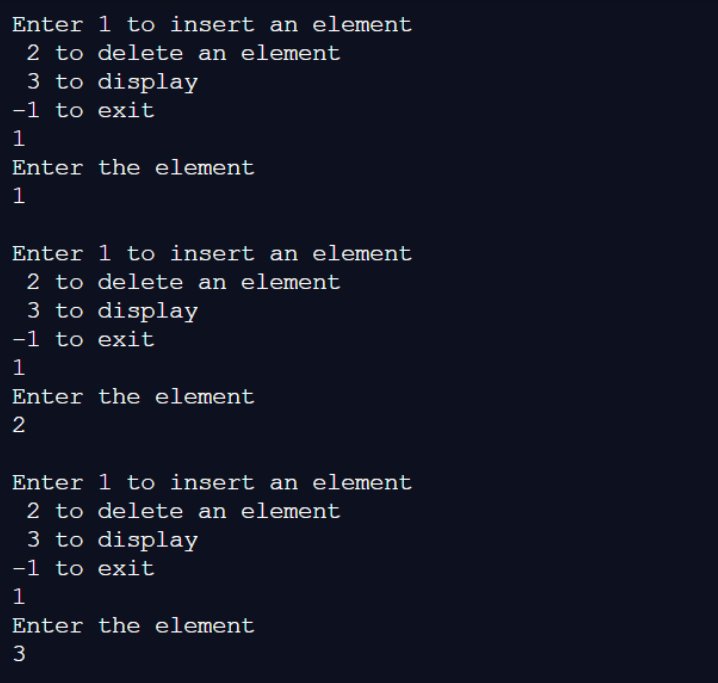
}

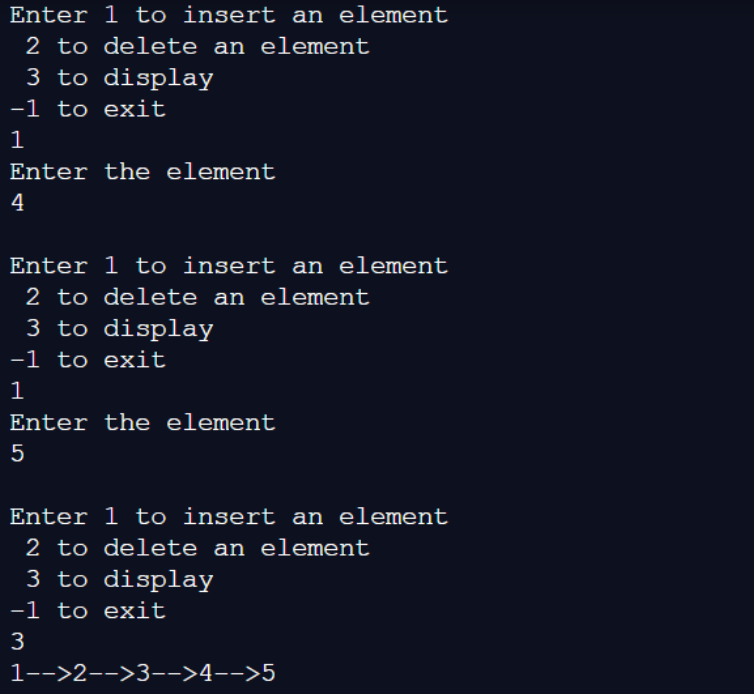
}}while(ch!=-1);

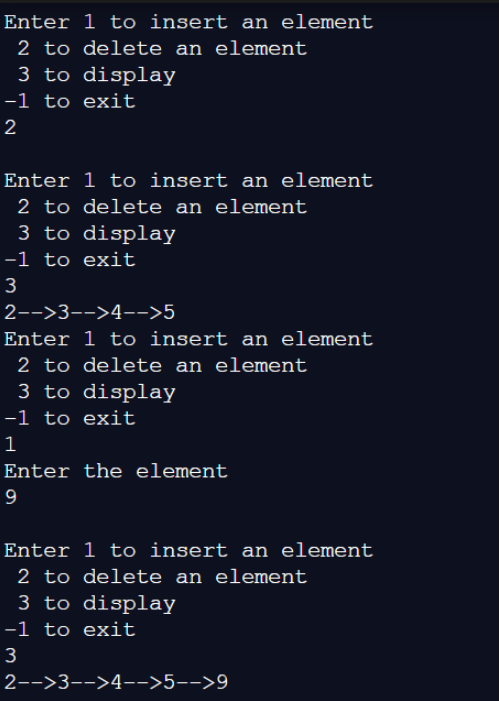
return 0;

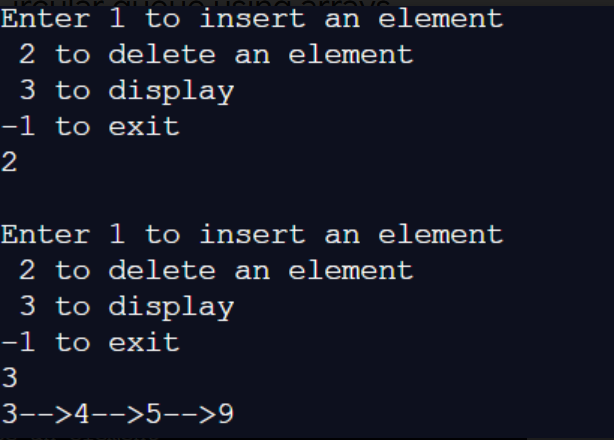
}

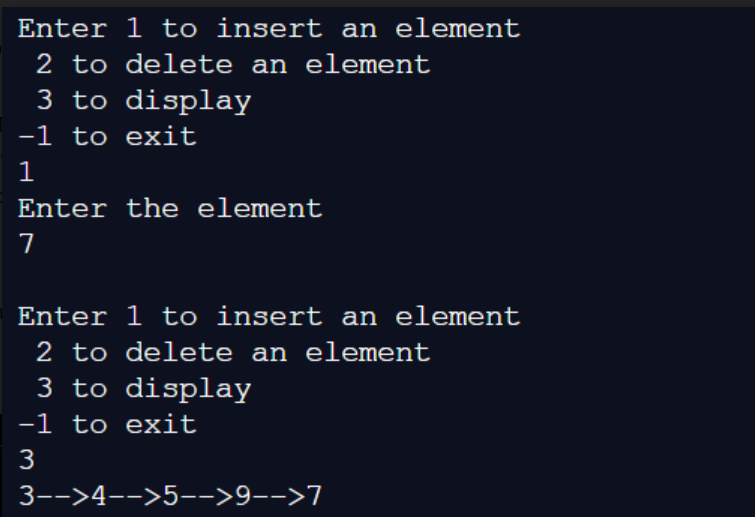
**Screenshots of Output:**











**Explain the Importance of the approach followed by you**

.

In this approach

1. we have considered the edge cases, where the queue may be full, such that insertions cannot be made , or when the queue is empty such that deletions cannot occur.
2. We have also considered adjusted our front and rear pointers appropriately in each function that manipulates the queue for insertions or deletion so that the queue may be useful for accessing in the future operations.
3. We have also considered various possible arrangements of front and rear pointers that may occur when user tries to display the queue.

**Conclusion:-**

In this experiment , we have learnt about the various types of queue data structures , and the unique importance of each of them . We have also learnt how to perform basic operations such as inserting or deleting from front or rear for the Doubly Ended Queue and Circular Queue that was assigned to us.

**PostLab Questions:**

1. **Illustrate 2 Applications of Queues.**

**Application of queues are as follows :**

1. Serving requests on a single shared resource , for example , CPU task-scheduling utilizes the application of queues. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.
2. A special type of queue known as priority queue , which utilizes the applications of queues , but gives priority to certain elements depending on their priority number , finds many applications in Computer Science , such as the Dijkstra’s Shortest Path Algorithm , in which the graph is stored in the form of an adjacency list or a matrix , priority queue can be used to extract the minimum efficiently**.**
3. **How does circular queue overcome the limitation of linear queue?**
4. The essential difference advantage of the linear queue over the circular queue is that the linear queue consumes more space than the circular queue, while the circular queue was devised to limit the memory wastage of the linear queue.
5. When it comes to linear queue the insertion can be performed only from the rear end and deletion from the front end. In a full queue after performing series of successive deletions in the queue arises a certain situation where no new element can be added further even if the space available because the underflow condition (Rear = max – 1) still exists.
6. Circular queue connects the two ends through a pointer where the very first element comes after the last element. It also keeps track of the front and rear by implementing some extra logic so that it could trace the elements that are to be inserted and deleted. With this, the circular queue does not generate the overflow condition until the queue is full in actual.