Circular Queue
Data Structure

- A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size.
- Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.

Abstract Data Type (ADT)

- An abstract data type (ADT) is a data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

Example of ADT

- Integers are an ADT, defined as the values …, −2, −1, 0, 1, 2, …, and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc., which behave according to familiar mathematics

- Abstract stack, which is a last-in-first-out structure, could be defined by three operations: push, that inserts a data item onto the stack; pop, that removes a data item from it; and peek or top, that accesses a data item on top of the stack without removal.

- Abstract queue, which is a first-in-first-out structure, would also have three operations: enqueue, that inserts a data item into the queue; dequeue, that removes the first data item from it; and front, that accesses and serves the first data item in the queue.

Advantages of Circular Queues

- In Circular Queues we utilize memory efficiently. because in queue when we delete any element only front increment by 1, but that position is not used later. so when we perform more add and delete operation, memory wastage increase. But in CQ memory is utilized, if we delete any element that position is used later, because it is circular.

Implementation of Circular Queue

- Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Step 2: Declare all user defined functions used in circular queue implementation.
- Step 3: Create a one dimensional array with above defined SIZE (int cQueue[SIZE])

- Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

- Step 5: Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

Step 1: Check whether queue is FULL.

((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2: If it is FULL, then

display "Queue is FULL!!!

Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then check

rear == SIZE - 1 && front != 0

if it is TRUE, then set rear = -1.

Step 4: Increment rear value by one (rear++), set queue[rear] = value and check

'front == -1' if it is TRUE, then set front = 0.

Deleting a value from Circular Queue

- In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

Step 1: Check whether queue is EMPTY.

(front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

- Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Displays elements of a Circular Queue

Step 1: Check whether queue is EMPTY. (front == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.

Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.

- Step 6: Set i to 0.

- Step 7: Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Program to implement Circular Queue using Array

```
#include<stdio.h>

#include<conio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int cQueue[SIZE], front = -1, rear = -1;
```

```
void enQueue(int value)
{
if((front == 0 && rear == SIZE - 1) || (front == rear+1))
{
printf("Circular Queue is Full. Insertion not possible");
}
```

```
else
{
if(rear == SIZE-1 && front != 0)
rear = -1;
cQueue[++rear] = value;
printf("\nInsertion Success!!!\n");
if(front == -1)
front = 0;
 } }
```

```
void deQueue()
{
if(front == -1 && rear == -1)
printf("\nCircular Queue is Empty! Deletion is not
    possible!!!\n");
else  {
printf("\nDeleted element %d\n",cQueue[front++]);
 if(front == SIZE)
front = 0;
if(front-1 == rear)
 front = rear = -1;
}
}
```

```
void display()
{
if(front == -1)
printf("Circular Queue is Empty.");
Else
{
int i = front;
printf("Circular Queue Elements are");
if(front <= rear)
{
while(i <= rear)
printf("%d",cQueue[i++]);
}
```

```
else
{
while(i <= SIZE - 1)
printf("%d", cQueue[i++]);
i = 0;
while(i <= rear)
printf("%d",cQueue[i++]);
} } }
```