

Aim: Implement Circular Queue ADT using an array.

Theory:

A Circular Queue is an extended version of a normal queue where the last Element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called 'Ring Buffer'

Procedure

1. Initialize an array queue of size n , where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.
3. Enqueue: To enqueue an element x into the queue, do the following:
 - Increment rear by 1.
 - If rear is equal to n , set rear to 0.
 - If front is -1, set front to 0.
 - Set `queue[rear]` to x .
4. Dequeue: To dequeue an element from the queue, do the following:
 - Check if the queue is empty by checking if front is -1.
 - If it is, return an error message indicating that the queue is empty.
 - Set x to `queue[front]`.
 - If front is equal to rear, set front and rear to -1.
 - Otherwise, increment front by 1 and if front is equal to n , Set front to 0.
 - Return x .

Code:

```
// C Program to implement the circular queue in c using arrays
#include <stdio.h>

// Define the maximum size of the queue
#define MAX_SIZE 5

// Declare the queue array and front, rear variables
int queue[MAX_SIZE];
int front = -1, rear = -1;

// Function to check if the queue is full
int isFull()
{
    // If the next position is the front, the queue is full
    return (rear + 1) % MAX_SIZE == front;
}

// Function to check if the queue is empty
int isEmpty()
{
    // If the front hasn't been set, the queue is empty
    return front == -1;
}

// Function to enqueue (insert) an element
void enqueue(int data)
{
    // If the queue is full, print an error message and
    // return
    if (isFull()) {
        printf("Queue overflow\n");
        return;
    }
}
```

```

// If the queue is empty, set the front to the first
// position
if (front == -1) {
    front = 0;
}
// Add the data to the queue and move the rear pointer
rear = (rear + 1) % MAX_SIZE;
queue[rear] = data;
printf("Element %d inserted\n", data);
}
// Function to dequeue (remove) an element
int dequeue()
{
    // If the queue is empty, print an error message and
    // return -1
    if (isEmpty()) {
        printf("Queue underflow\n");
        return -1;
    }
    // Get the data from the front of the queue
    int data = queue[front];
    // If the front and rear pointers are at the same
    // position, reset them
    if (front == rear) {
        front = rear = -1;
    }
    else {
        // Otherwise, move the front pointer to the next
        // position
        front = (front + 1) % MAX_SIZE;
    }
}

```

```

// Return the dequeued data
return data;
}

// Function to display the queue elements
void display()
{
// If the queue is empty, print a message and return
if (isEmpty()) {
printf("Queue is empty\n");
return;
}

// Print the elements in the queue
printf("Queue elements: ");
int i = front;
while (i != rear) {
printf("%d ", queue[i]);
i = (i + 1) % MAX_SIZE;
}

// Print the last element
printf("%d\n", queue[rear]);
}

// Main function
int main()
{
// Enqueue some elements
enqueue(10);
enqueue(20);
enqueue(30);

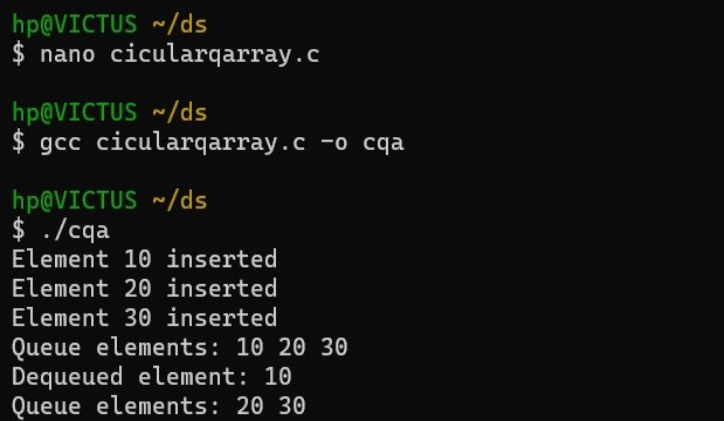
// Display the queue
display();

// Dequeue an element and print it

```

```
printf("Dequeued element: %d\n", dequeue());  
// Display the queue again  
display();  
// End of main function  
return 0;  
}
```

Output:



```
hp@VICTUS ~/ds  
$ nano cicularqarray.c  
  
hp@VICTUS ~/ds  
$ gcc cicularqarray.c -o cqa  
  
hp@VICTUS ~/ds  
$ ./cqa  
Element 10 inserted  
Element 20 inserted  
Element 30 inserted  
Queue elements: 10 20 30  
Dequeued element: 10  
Queue elements: 20 30
```

Conclusion: This experiment explores a more efficient version of the queue, the Circular Queue ADT. It implements this structure using an array, showcasing how to handle circular buffer mechanisms and optimize queue operations by utilizing memory efficiently.