**Experiment no 5**

**NAME:- KARTIK RATHOD**

**Roll no:- 16 Division:-C**

**Aim:** To implement Circular Queue ADT using array

**Objective:**

Circular Queue offer a quick and clean way to store FIFIO data with maximum size

Theory:A Circular queue is a linear data structure where the first index comes right

after the last index assuming indices are attached in a circular manner.

**Algorithm :**1. Create an array 'items' of size 'SIZE' and initialize 'front' and 'rear' to -1.

2. Define a function isFull():

- If ((front == rear + 1) or (front == 0 and rear == SIZE - 1)), return true (1).

- Otherwise, return false (0).

3. Define a function isEmpty():

- If 'front' is -1, return true (1).

- Otherwise, return false (0).

4. Define a function enQueue(element):

- If isFull() returns true:

- Print "Queue is full."

- Else:

- If 'front' is -1, set 'front' to 0.

- Increment 'rear' (rear = (rear + 1) % SIZE).

- Store 'element' in 'items[rear]'.

- Print "Inserted -> element".

5. Define a function deQueue():

- If isEmpty() returns true:

- Print "Queue is empty."

- Return -1 to indicate an error.

- Else:

- Store 'element' as 'items[front]'.

- If 'front' equals 'rear', indicating there's only one element:

- Set 'front' and 'rear' back to -1 to empty the queue.

- Otherwise, increment 'front' (front = (front + 1) % SIZE).

- Print "Deleted element -> element".

- Return 'element'.

6. Define a function display():

- If isEmpty() returns true:

- Print "Empty Queue."

- Else:

- Print "Front -> front".

- Print "Items -> ".

- Loop 'i' from 'front' to 'rear':

- Print "items[i] ".

- Print "Rear -> rear".

7. In the main program:

- Perform operations like enQueue, deQueue, and display to demonstrate the circular queue.

8. End of the program.

**Circular Queue implementation in C:**

#include <stdio.h>

#define SIZE 5

int items[SIZE];

int front = -1, rear = -1;

// Check if the queue is full

int isFull() {

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

return 0;

}

// Check if the queue is empty

int isEmpty() {

if (front == -1) return 1;

return 0;

}

// Adding an element

void enQueue(int element) {

if (isFull())

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

else {

if (front == -1) front = 0;

rear = (rear + 1) % SIZE;

items[rear] = element;

printf("\n Inserted -> %d", element);

}

}

// Removing an element

int deQueue() {

int element;

if (isEmpty()) {

printf("\n Queue is empty !! \n");

return (-1);

} else {

element = items[front];

if (front == rear) {

front = -1;

rear = -1;

}

// Q has only one element, so we reset the

// queue after dequeing it. ?

else {

front = (front + 1) % SIZE;

}

printf("\n Deleted element -> %d \n", element);

return (element);

}

}

// Display the queue

void display() {

int i;

if (isEmpty())

printf(" \n Empty Queue\n");

else {

printf("\n Front -> %d ", front);

printf("\n Items -> ");

for (i = front; i != rear; i = (i + 1) % SIZE) {

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

}

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

printf("\n Rear -> %d \n", rear);

}

}

int main() {

// Fails because front = -1

deQueue();

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

// Fails to enqueue because front == 0 && rear == SIZE - 1

enQueue(6);

display();

deQueue();

display();

enQueue(7);

display();

// Fails to enqueue because front == rear + 1

enQueue(8);

return 0;

}

**Conclusion:**In conclusion, a circular queue is a data structure that offers efficient and effective management of data elements in a circular manner. It combines the advantages of both queues and arrays, making it particularly useful for scenarios where elements need to be processed in a cyclical or circular fashion.

**Output:**

