**Experiment no 5 :- Circular Queue.**

**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 :-**

The Circular Queue is similar to a Linear Queue in the sense that it follows the FIFO (First In First Out) principle but differs in the fact that the last position is connected to the first position, replicating a circle.

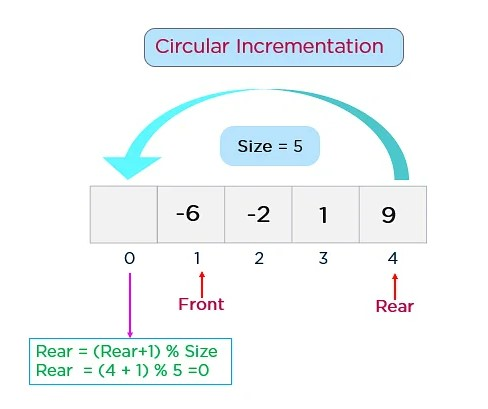
Operations

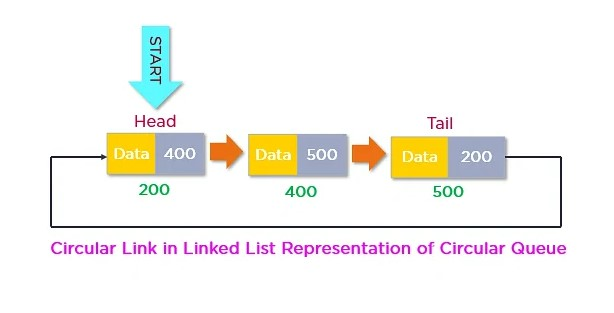
* Front - Used to get the starting element of the Circular Queue.
* Rear - Used to get the end element of the Circular Queue.
* enQueue(value) - Used to insert a new value in the Circular Queue. This operation takes place from the end of the Queue.
* deQueue() - Used to delete a value from the Circular Queue. This operation takes place from the front of the Queue.

## Representation of Circular Queue using Arrays and a Linked List

You can implement the circular queue using both the 1-D [array](https://www.simplilearn.com/tutorials/data-structure-tutorial/arrays-in-data-structure) and the [Linked list](https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure). However, implementing a circular link is a new addition that you need to execute. Additionally, this queue works by the process of circular incrementation. That is, when you reach the end of a queue, you start from the beginning of a queue. The circular incrementation is achievable with the help of the modulo division.

Now you will understand how you can achieve circular incrementation, with the help of an example. Let’s say the MaxSize of your queue is 5, and the rear pointer has already reached the end of a queue. There is one empty space at the beginning of a queue, which means that the front pointer is pointing to location 1.





**ALGORITHM :-**

### 1. Enqueue(x) Operation

* 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

### 2. Dequeue() Operation

* 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

**CODE :-**

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;

}

**OUTPUT:-**



**CONCLUSION :-** The circular queue is a linear data structure whose end is connected to the start and is used in the traffic system, memory management, and CPU scheduling.