Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No:4

AIM: Implementation of Circular Queue Data Structure using array.

THEORY:

Circular Queue:
A circular queue is a type of queue where the last position is connected back to the first position to form a circle. This structure is an improvement over a linear queue as it efficiently uses the available space and avoids the problem of overflow that occurs in a linear queue when elements are dequeued.

Queue Structure:
Front: Points to the position of the first element in the queue.
Rear: Points to the position where the next element will be inserted.
Size: Total number of elements in the queue (not to be confused with capacity).

Circular Nature:
The queue wraps around to the beginning when it reaches the end, allowing efficient use of space.
Operations:

Enqueue: Adds an element to the rear of the queue. If the queue is full, it may overwrite the oldest element if the queue is circularly implemented.
Dequeue: Removes an element from the front of the queue. If the queue is empty, this operation has no effect.
Display: Shows all elements from the front to the rear, handling the wrap-around condition.

Advantages:
Efficient Space Utilization: Unlike a linear queue, a circular queue does not waste space by shifting elements. It efficiently utilizes all available slots.
Fixed Size: The maximum size of the queue is predefined, which helps in managing memory allocation.

Implementation:
Array-Based Circular Queue: Uses an array to implement the circular queue. Two pointers (front and rear) are used to track the beginning and end of the queue, respectively.

Here, are some disadvantages of circular queue:

- It has limited capacity, which can lead to overflow and data loss.
- These are more complex to implement than a simple linear queue.
- It can be difficult to debug when there are errors.
- They require additional logic to handle empty and full states.

CODE :

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 5

int queue[MAX_SIZE];
int front = -1, rear = -1;

int isFull() {
    return (rear + 1) % MAX_SIZE == front;
}

int isEmpty() {
    return front == -1;
}
```

```
void enqueue(int data) {
    if (isFull()) {
        printf("Queue overflow\n");
        return;
    }
    if (isEmpty()) {
        front = 0;
    }
    rear = (rear + 1) % MAX_SIZE;
    queue[rear] = data;
    printf("Element %d inserted\n", data);
}

int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow\n");
        return -1;
    }
    int data = queue[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % MAX_SIZE;
    }
    return data;
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    int i = front;
```

```
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear) break;
        i = (i + 1) % MAX_SIZE;
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();

    printf("Dequeued element: %d\n", dequeue());
    display();

    return 0;
}
```

OUTPUT :

```
Element 10 inserted
Element 20 inserted
Element 30 inserted
Queue elements: 10 20 30
Dequeued element: 10
Queue elements: 20 30


=== Code Execution Successful ===
```

CONCLUSION:

A circular queue optimizes space by reusing the array in a circular fashion. This implementation maintains constant time complexity for insertion and deletion operations, making it efficient for scenarios like buffering and task scheduling. It avoids issues of wasted space and ensures all elements are accessed in a fixed-size buffer.