**Experiment No: VIII**

*Implementation of Circular Queue and Priority Queue.*

**Aim:** Implementation of Circular Queue and Priority Queue.

**Objectives:**

 To understand the concepts of circular queues and priority queues, including their operations.

 To implement a circular queue using an array.

 To implement a priority queue using a linked list or heap.

 To analyze the functionality and use cases for each type of queue.

**Theory:**

A circular queue is a linear data structure that connects the last position back to the first position to form a circle. This helps in efficiently utilizing space by allowing the enqueue and dequeue operations to wrap around, avoiding the need to shift elements when the queue is not full. The main operations include enqueue (adding an item) and dequeue (removing an item), with checks for overflow and underflow.

A priority queue is an abstract data type where each element has a priority assigned to it. In a priority queue, elements are served based on their priority rather than their order in the queue. Higher priority elements are dequeued before lower priority ones. Priority queues can be implemented using arrays, linked lists, or heaps, with heaps being the most efficient for insertion and deletion operations.

**Program:**

1. C Implementation

*#include* <stdio.h>

*#include* <stdlib.h>

*#define* MAX 100

*// Circular Queue structure*

*struct* CircularQueue {

*int* front, rear;

*int* items[MAX];

};

*// Function to create an empty queue*

*struct* CircularQueue*\** createQueue() {

*struct* CircularQueue*\** queue *=* (*struct* CircularQueue*\**)malloc(*sizeof*(struct CircularQueue));

    queue->front *=* queue->rear *=* *-*1;

*return* queue;

}

*// Check if the queue is full*

*int* isFull(*struct* CircularQueue*\** *queue*) {

*return* (queue->rear *+* 1) *%* MAX *==* queue->front;

}

*// Check if the queue is empty*

*int* isEmpty(*struct* CircularQueue*\** *queue*) {

*return* queue->front *==* *-*1;

}

*// Enqueue operation*

*void* enqueue(*struct* CircularQueue*\** *queue*, *int* *item*) {

*if* (isFull(queue)) {

        printf("Queue overflow\n");

    } *else* {

*if* (isEmpty(queue)) {

            queue->front *=* queue->rear *=* 0;

        } *else* {

            queue->rear *=* (queue->rear *+* 1) *%* MAX;

        }

        queue->items[queue->rear] *=* item;

        printf("%d enqueued to circular queue\n", item);

    }

}

*// Dequeue operation*

*int* dequeue(*struct* CircularQueue*\** *queue*) {

*if* (isEmpty(queue)) {

        printf("Queue underflow\n");

*return* *-*1;

    } *else* {

*int* item *=* queue->items[queue->front];

*if* (queue->front *==* queue->rear) {

            queue->front *=* queue->rear *=* *-*1; *// Reset queue*

        } *else* {

            queue->front *=* (queue->front *+* 1) *%* MAX;

        }

*return* item;

    }

}

*// Front operation*

*int* front(*struct* CircularQueue*\** *queue*) {

*if* (isEmpty(queue)) {

        printf("Queue is empty\n");

*return* *-*1;

    } *else* {

*return* queue->items[queue->front];

    }

}

*int* main() {

*struct* CircularQueue*\** queue *=* createQueue();

    enqueue(queue, 10);

    enqueue(queue, 20);

    printf("Front element is %d\n", front(queue));

    printf("%d dequeued from circular queue\n", dequeue(queue));

*return* 0;

}

1. C++ Implementation using Linked Lists

*#include* <iostream>

*using* *namespace* std;

*// Node structure*

*struct* Node {

*int* data;

*int* priority;

    Node*\** next;

};

*// Priority Queue structure*

*class* PriorityQueue {

*private:*

    Node*\** front;

*public:*

    PriorityQueue() {

        front *=* nullptr;

    }

*// Enqueue operation based on priority*

*void* enqueue(*int* *item*, *int* *priority*) {

        Node*\** newNode *=* *new* Node();

        newNode->data *=* item;

        newNode->priority *=* priority;

        newNode->next *=* nullptr;

*if* (front *==* nullptr *||* front->priority *>* priority) {

            newNode->next *=* front;

            front *=* newNode;

        } *else* {

            Node*\** current *=* front;

*while* (current->next *!=* nullptr *&&* current->next->priority *<=* priority) {

                current *=* current->next;

            }

            newNode->next *=* current->next;

            current->next *=* newNode;

        }

        cout *<<* item *<<* " with priority " *<<* priority *<<* " enqueued to priority queue\n";

    }

*// Dequeue operation*

*int* dequeue() {

*if* (front *==* nullptr) {

            cout *<<* "Queue underflow\n";

*return* *-*1;

        } *else* {

*int* item *=* front->data;

            Node*\** temp *=* front;

            front *=* front->next;

*delete* temp;

*return* item;

        }

    }

*// Front operation*

*int* getFront() {

*if* (front *==* nullptr) {

            cout *<<* "Queue is empty\n";

*return* *-*1;

        } *else* {

*return* front->data;

        }

    }

};

*int* main() {

    PriorityQueue pq;

    pq.enqueue(10, 2);

    pq.enqueue(20, 1);

    cout *<<* "Front element is " *<<* pq.getFront() *<<* endl;

    cout *<<* pq.dequeue() *<<* " dequeued from priority queue\n";

*return* 0;

}

**Output:**

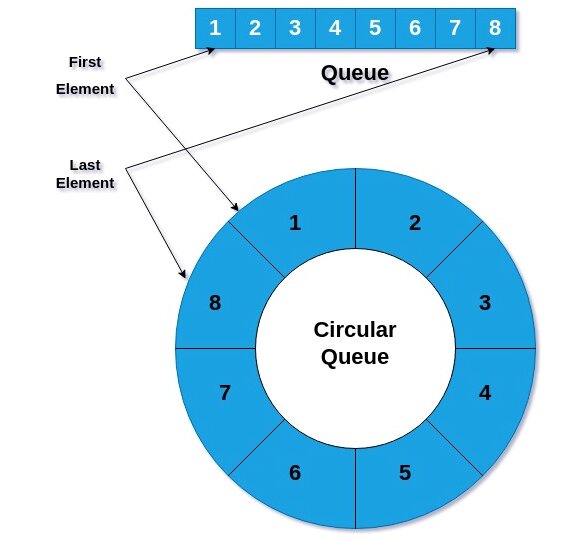
10 enqueued to circular queue

20 enqueued to circular queue

Front element is 10

10 dequeued from circular queue

**Diagram Representation:**



**CO Attained:**

**PO Attained:**

**Conclusion:**