Practical 6

Queue Implementation Using Linked List

Objective:

1. To understand the concept and functionality of **queue data structures**.
2. To **implement different types of queues** (Linear, Circular, and Priority) using **linked lists** in C++.
3. To learn **enqueue and dequeue operations** and how they differ in each queue type.

C++ Program: Linear Queue using Linked List

```
#include <iostream>
using namespace std;
// Node structure for linked list
struct Node {
   int data;
   Node* next;
};
// Queue class using linked list
class Queue {
private:
   Node* front;  // Points to the front (head) of the queue
   Node* rear;   // Points to the rear (tail) of the queue
public:
   // Constructor
   Queue() {
      front = rear = NULL;
   }
   // Function to add an element to the queue
   void enqueue(int value) {
      Node* temp = new Node();
      temp->data = value;
      temp->next = NULL;
      // If queue is empty
      if (front == NULL) {
         front = rear = temp;
      } else {
         rear->next = temp;
         rear = temp; }
      cout << "Enqueued: " << value << endl;}
   // Function to remove an element from the queue
   void dequeue() {
      if (front == NULL) {
         cout << "Queue Underflow! (Queue is empty)"
<< endl;
         return;}
      Node* temp = front;
      cout << "Dequeued: " << temp->data << endl;
      front = front->next;
      // If queue becomes empty after deletion
      if (front == NULL)
         rear = NULL;
      delete temp; }
   // Function to display all elements in the queue
   void display() {
      if (front == NULL) {
```

```
        cout << "Queue is empty." << endl;
        return; }
    Node* temp = front;
    cout << "Queue elements: ";
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next; }
    cout << endl;}
  // Function to view the front element
  void peek() {
    if (front == NULL)
        cout << "Queue is empty." << endl;
    else
        cout << "Front element: " << front->data <<
endl;}};
// Main function
int main() {
  Queue q;
  int choice, value;
  cout << "==== Linear Queue using Linked List
====" << endl;
  while (true) {
    cout << "\n1. Enqueue\n2. Dequeue\n3. Peek\n4.
Display\n5. Exit" << endl;
    cout << "Enter your choice: ";
    cin >> choice;
    switch (choice) {
      case 1:
        cout << "Enter value to enqueue: ";
        cin >> value;
        q.enqueue(value);
        break;
      case 2:
        q.dequeue();
        break;
      case 3:
        q.peek();
        break;
      case 4:
        q.display();
        break;
      case 5:
        cout << "Exiting program..." << endl;
        return 0;
      default:
        cout << "Invalid choice! Please try again." <<
endl;   }}
  return 0;}
```

Scenario: Customer Service Ticket System

Problem Context

Imagine you are developing a **Customer Service Ticket Management System** for a company. When customers contact support, each request (ticket) is added to a queue — the first customer who submits a ticket should be the first one served.

However, the number of tickets can vary throughout the day, so the queue size needs to be **dynamic**. To handle this efficiently, you decide to implement the queue using a **linked list**, since it can grow and shrink at runtime without wasting memory.

System Description

- Each **customer request** is represented as a **node** in the linked list.
- The **front** of the queue represents the **customer being served**.
- The **rear** of the queue represents the **latest customer who joined the line**.
- As soon as a ticket is resolved, it is **dequeued** (removed from the front).
- New customer requests are **enqueued** at the rear end.

C++ Program: Circular Queue using Linked List

```
#include <iostream>
using namespace std;
// Node structure for circular linked list
struct Node {
    int data;
    Node* next;};
// Queue class implementing Circular Queue using Linked List
class CircularQueue {
private:
    Node* front;  // Points to the first element
    Node* rear;   // Points to the last element
public:
    // Constructor
    CircularQueue() {
        front = rear = NULL; }
    // Function to check if queue is empty
    bool isEmpty() {
        return (front == NULL);}
    // Function to add element to the queue
    void enqueue(int value) {
        Node* temp = new Node();
        temp->data = value;
        temp->next = NULL;
        // If queue is empty
        if (front == NULL) {
            front = rear = temp;
            rear->next = front;  // Link last node to first node}else {
            rear->next = temp;   // Link old rear to new node
            rear = temp;         // Update rear pointer
            rear->next = front;  // Make it circular    }
        cout << "Enqueued: " << value << endl;}
    // Function to remove element from the queue
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow! (Queue is empty)" << endl;
return;      }
        // Single element case
        if (front == rear) {
            cout << "Dequeued: " << front->data << endl;
            delete front;
            front = rear = NULL;} else {
            Node* temp = front;
            cout << "Dequeued: " << temp->data << endl;
            front = front->next;     // Move front forward
            rear->next = front;      // Maintain circular link
            delete temp;}}
    // Function to display all elements
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty." << endl;
            return;}
        Node* temp = front;
        cout << "Circular Queue elements: ";
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != front);
        cout << endl;}
    // Function to peek front element
    void peek() {
        if (isEmpty()) {
            cout << "Queue is empty." << endl;
        }  else {
            cout << "Front element: " << front->data << endl;}}};
```

// Main function

Scenario: Circular Queue using Linked List

Problem Context

In many computer systems, resources such as CPU time, network bandwidth, or printer access must be shared among multiple users or processes fairly and efficiently.
A Circular Queue provides an ideal structure for such scheduling systems because it allows continuous cycling through tasks without restarting or wasting memory.

To simulate this behavior dynamically, we use a linked list-based circular queue, where the last node points back to the first node — forming a logical loop.

Real-World Example: Round Robin CPU Scheduling

- You are developing a Round Robin CPU Scheduler for an operating system.
- Multiple processes (P1, P2, P3, …) request CPU time.
- Each process is given a fixed time slice (quantum) to execute.
- After using its time, the process is moved to the end of the queue.
- The CPU then assigns time to the next process in line, and the cycle continues.

This scheduling continues in a circular manner until all processes complete — ensuring fairness and equal opportunity for all active tasks.

Priority Queue — Linked List Implementation

```
#include <iostream>

using namespace std;

struct Node {

   int data;

   int priority;

   Node* next;

   Node(int d, int p) : data(d), priority(p), next(nullptr) {}};

class PriorityQueue {

private:

   Node* front;  // points to the highest-priority node (lowest priority value)

public:

   PriorityQueue() : front(nullptr) {}

   ~PriorityQueue() {

      // Free all nodes

      while (front != nullptr) {

         Node* temp = front;

         front = front->next;

         delete temp;   }}

   bool isEmpty() const {

      return front == nullptr;}

   // Insert new node maintaining ascending order of priority (lower value => higher priority)

   // For equal priority values, the new node is placed after existing nodes of same priority (FIFO).

   void enqueue(int data, int priority) {

      Node* temp = new Node(data, priority);

      // If queue is empty or new node has higher priority than front

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

         temp->next = front;

         front = temp;

         cout << "Enqueued: (" << data << ", p=" << priority << ")\n";

         return;}

      // Find the insertion point: move while next exists and next->priority <= priority

      // This places the new node after existing nodes with equal priority (FIFO for equal priority).

      Node* curr = front;

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

         curr = curr->next;}
```

```
        temp->next = curr->next;
        curr->next = temp;
        cout << "Enqueued: (" << data << ", p=" << priority <<
")\n";}
    // Remove and return the front node
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow! (Priority Queue is empty)\n";
            return;}
        Node* temp = front;
        cout << "Dequeued: (" << temp->data << ", p=" << temp-
>priority << ")\n";
        front = front->next;
        delete temp;}
    // View the front element
    void peek() const {
        if (isEmpty()) {
            cout << "Priority Queue is empty.\n";
        } else {
            cout << "Front element: (" << front->data << ", p=" <<
front->priority << ")\n";}
    }

    // Display full queue from front to rear
    void display() const {
        if (isEmpty()) {
            cout << "Priority Queue is empty.\n";
            return;}
        cout << "Priority Queue: ";
        Node* curr = front;
        while (curr != nullptr) {
            cout << "(" << curr->data << ",p=" << curr->priority <<
")";
           if (curr->next) cout << " -> ";
            curr = curr->next;       }
        cout << "\n";}};
```

Scenario: Priority Queue using Linked List

Problem Context

In many real-life systems, tasks or requests do **not all have the same level of importance**. Some must be handled immediately, while others can wait.
For instance, in an **emergency room**, a patient with a critical condition is treated before others, even if they arrive later.
This type of system can be effectively modeled using a **Priority Queue**.

A **Priority Queue** ensures that elements are **served based on their priority** rather than just the order in which they arrive.

Real-World Example: Hospital Emergency Room System

You are designing a program for an **Emergency Room (ER) management system** in a hospital.

- Each patient is assigned a **priority level** based on their condition.
- A **lower number** indicates a **higher priority** (e.g., Priority 1 = Critical).
- Patients are inserted into the queue according to their priority rather than arrival order.
- The **doctor always treats the patient with the highest priority first**, regardless of when they enter.

This system ensures that **urgent cases receive immediate attention**, improving efficiency and fairness in medical emergencies.