Assignment 02:

Singly Linked List & Priority Queue

Employee and Patient Management System Using Linked List and Priority Queue

---

Time Management

Average estimated time to complete this assignment is **12 hours**.

• **[1 HOUR] Software Requirement Understanding**
- Reading the assignment document. - Understanding the problem statement for both Employee and Patient Management Systems. - Identifying inputs and outputs for linked list, queue, and priority queue operations.

• **[2 HOURS] Software Design**
- Planning user interactions with both systems: - For Employee Management: add, remove, and display employee data using linked list. - For Patient Management: enqueue, dequeue, and priority updates in patient queue. - Identifying required variables and data structures: - Employee and Patient structures, along with queue and linked list nodes. - Modularizing code into functions: - Function names, tasks, parameters, and return types. - Defining flow sequence of main operations for both parts of the system.

• **[6 HOURS] Software Coding**
- Implementing linked list functions for employee data management. - Coding basic queue operations for the patient queue. - Implementing priority queue logic for patients based on severity and wait time. - Integrating both systems into a cohesive main function.

• **[2 HOURS] Software Testing**
- Testing both systems with various input cases. - Identifying and fixing errors, especially for edge cases in priority queue handling. - Re-running tests to confirm correct queue prioritization and linked list updates.

• **[1 HOUR] Software Documentation**
- Organizing code with proper indentation and naming conventions. - Adding comments to explain function logic and code flow. - Documenting the approach for handling priority and edge cases.

Objectives

- Develop a linked list for employee management and a priority queue for patient management.
- Understand and apply ADT concepts across linked lists, queues, and priority queues.
- Design modular, reusable functions for handling queue prioritization and linked list operations.

Overview

Linked List & Queue ADTs
Used for employee and patient data management, respectively. The Employee Management System relies on linked lists for efficient data handling, while the Patient Management System uses queues with a priority structure to simulate a hospital triage system.

Priority Queue
Implemented in the patient queue to ensure patients with higher severity or longer wait times are seen first.

Problem 1

Implement a system to manage the storage and organization of employee data using a combination of doubly and singly linked lists. First, you will need to implement a doubly linked list to contain the data about the employees. Each node of this linked list will have the following structure:

Structure

```
class EmployeeNode {
public:
    EmployeeData data; // Rest of the employee information, see below
    EmployeeNode* prev; // Link to previous node
    EmployeeNode* next; // Link to the next node
};
```

The employee data (EmployeeData) is represented as a structure containing two fields (name and a linked list of assigned projects with their completion status)

```
struct EmployeeData {
    string name;        // Name of the employee
    string employeeID;  // Unique employee ID
    ProjectNode* head;  // Pointer to the head of the projects/status list
};
```

In the declaration above, head is a pointer that will point to the first node in a list of projects. This list will contain project numbers (nodes) along with their completion status for a particular employee.

```
struct ProjectNode {
    string projectName;     // Name of the project
    string projectCode;     // Unique project code, e.g., PRJ213
    bool isCompleted;       // Project completion status: true if completed, 
false otherwise
    ProjectNode* next;      // Link to the next project node
};
```

The list of employees (EmployeeList) will be implemented as a standard (non-circular) doubly linked list and should be sorted in increasing order of employee IDs. Implement each member function in the EmployeeList class, and feel free to add additional methods if needed.

```
class EmployeeList {
private:
    EmployeeNode* head;
public:
    EmployeeList(); // Constructor to create an empty EmployeeList
    ~EmployeeList(); // Destructor to deallocate EmployeeList

    bool addEmployee(const string& employeeID, const char* name);
    // Add an employee record to the EmployeeList. Employee ID must be unique.
    // Return false if the employee ID already exists.

    bool addProject(const string& employeeID, const string& projectName, const 
string& projectCode, bool isCompleted);
    // Add a project to an employee’s record. Employee ID must exist in the list,
    // and this project should not already be assigned to this employee.
    // Return false if the employee ID doesn’t exist or if the project already 
exists.

    bool removeEmployee(const string& employeeID);
    // Remove an employee's record from the EmployeeList.
    // Return true if the employee is successfully removed.
    void updateEmployee(const string& employeeID, const string& newProjectName,
bool isCompleted);
    // Update an employee's record by adding a new project (along with its
status) to their list of projects.
    // Insert the new project at the head of their project list.

    void displayEmployee(const string& employeeID);
    // Display the record of a particular employee on the console.

    void displayAllEmployees(int order);
    // Display records of all employees. If `order=0`, display the employees
    // in ascending order of their employee IDs. If `order=1`, display them in
descending order.
};
```

Note:

You can add helper functions according to your needs, also write a menu driven program, to show off your system.

Problem 2

Priority Queue Explanation

A **priority queue** is a type of queue where each element is associated with a **priority**. In a standard queue, elements are processed in the **first-in, first-out (FIFO)** order, but in a priority queue, elements are dequeued based on their priority, regardless of their order of arrival.

How Priority Queues Work:

1. **Insertion**: Elements are added to the queue along with their priority.
2. **Dequeuing**: Elements with the **highest priority** are dequeued first. If multiple elements have the same priority, they can be processed based on their order of arrival (FIFO within the same priority level).

Priority queues are commonly implemented using:

- **Arrays** (sorted or unsorted).
- **Heaps** (often binary heaps), which enable efficient access to the highest priority element.
- **Linked lists** with additional logic for maintaining order based on priority.

Priority queues are especially useful in scenarios where tasks, processes, or jobs must be handled according to urgency or importance, such as in operating systems, task scheduling, and network packet management.

Assignment Task: Hospital Patient Management System

In this assignment, students will create a **hospital patient management system** that uses a **priority queue** to manage the order in which patients are seen by doctors based on the severity of their condition.

Each patient has:

1. **Patient ID**
2. **Name**
3. **Severity level**: An integer value from 1 to 10, where 10 represents the highest severity.
4. **Arrival time**: The time when the patient joined the queue.

Requirements:

1. **Enqueue Patient**: When a new patient arrives, add them to the priority queue based on the severity level. If two patients have the same severity, the patient who arrived first should be prioritized.
2. **Serve Patient**: Implement a `servePatient` function that dequeues the patient with the highest severity level. Print the patient's details and update the queue. If no patients are in the queue, print a message indicating the queue is empty.
3. **View Patients in Queue**: Display all patients in the queue ordered by priority, showing their **ID, name, and severity level**. Use descending order based on severity level.

Todo:

- ➔ Remember that higher severity should have a higher priority in the queue.
- ➔ You will Modify your previous **Node-based Queue Implementation** to behave like a **Priority Queue**. Update the code to ensure elements are ordered by priority.