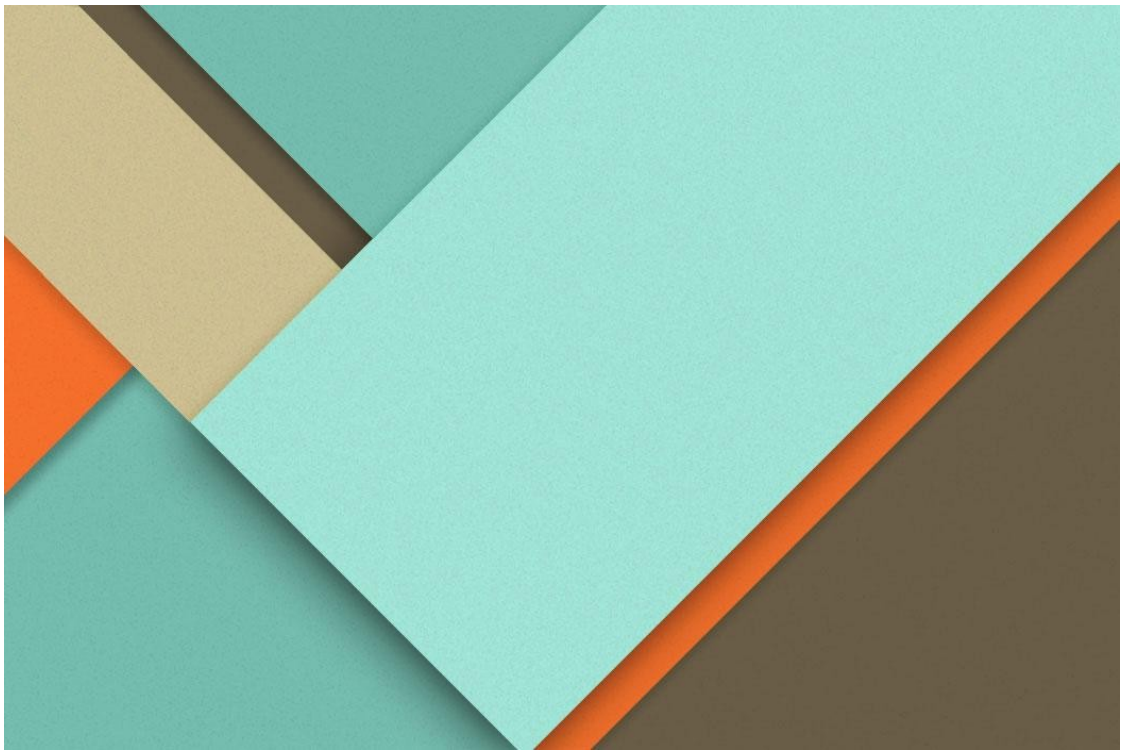

Hospital Management System

02-01-2026

Ilyan khan 24CSE24
Complex Engineering Activity
Data Structures and Algorithms
Computer Systems Engineering
Quaid-Awam University of Engineering, Science and Technology Nawabshah

System Architecture and Data Structures

1. Introduction

The Hospital management system is a C++ application designed to streamline emergency room operations. It integrates three core modules: record keeping, medical triage and resource tracking. By utilizing specific data structures like hash tables and sorted arrays, the system balances the need for fast data retrieval with the necessity of medical prioritisation.

2. Core Data Structures

The efficiency of the system is rooted in the following structural choices:

- **Hash Table (Patient Registration):** To provide O(1) average-time search for patient records. It uses an array of 10 buckets with Linked List Chaining to handle Collisions.

- **Sorted Array (Triage Queue):** A 50-slot array that acts as a priority Queue. It ensures that the patients with the highest medical urgency (Priority 1) are treated before those with lower urgency (Priority 3).

- **Resource Counters:** Integer-based tracking for physical assets like ventilators and beds.

3. Main Functions added:

- **Add Patient:** This function takes the data of a patient such as patient ID, Patient Name and Priority. It then adds the patient to the hospital.

- **Treat Patient:** This function uses the sorted array Queue to treat patients with the priority value, (1-first, 2-after first and 3-last).

- **Search Patient:** This function is used to search any patient with a patient's ID and display that patient's details.

Functional logic and Algorithm Analysis

1. The Workflow Pipeline:

When a user adds a patient, the system performs a dual-action operation:

1. **Storage:** The patient object is hashed into the PatientRegistry.

2. **Sort:** The patient is placed in the ManualTriage queue, Where a Bubble Sort algorithm reorders the queue based on the priority integer.

2. Algorithmic Efficiency:

The system's performance can be mathematically evaluated as:

Operation	Algorithm	Complexity	Why?
Search	Hash Map Lookup	0(1)	Direct Index via ID (mod of 10)
Insertion	Bubble Sort	$0(n^2)$	Nested loops compare every patient pair after addition.
Triage	Array Pop	0(1)	Removing the last element is a constant time operation.

3.Resource allocation Logic

The system uses a conditional decision tree for treatment:

- **Priority 1:** Attempts to reserve a Ventilator first, if unavailable, it falls back to a Bed.

- **Priority 2/3 :** These patients are strictly assigned to Beds.

- **Failure State:** If no resources are available, the patient remains in a "waiting" status, preventing over-allocation.

Critical Evaluation and Future Roadmap

1. System Strengths

- **Data Integrity:** By separating the PatientRegistry from ManualTriage queue, the system maintains a permanent searchable record even after the patient is treated and removed from the active queue.

- **Urgency Based Logic:** Unlike a standard First-In-first-Out (FIFO) queue, this system ensures that life threatening cases (Priority 1) are always moved to the front, simulating real-world medical ethics.

2. Identified Weaknesses

- **Lag problem:** The use of a bubble sort inside a pus function is inefficient ($0(n^2)$). As the patient's volume increases, the system will experience a significant “lag” when adding new records.

3. Proposed Technical Enhancements

- **Dynamic Resizing:** Transition from fixed-size array (queueArray[50]) to dynamic structure like “vector” to prevent system crashes during high-capability events (Overflow).

Appendix: Full Source code

```
#CEP Hospital Management System By Ilyan Khan 24CSE24
#include <iostream>
#include <string>

using namespace std;
// Basic class for patient details
class Patient {
public:
    int id;
    string name;
    int priority;

    Patient() {
        id = 0;
        name = "";
        priority = 3;
    }

    Patient(int i, string n, int p) {
        id = i;
        name = n;
        priority = p;
    }
};

// Linked list node for hash table
class Node {
public:
    Patient data;
    Node* next;
    Node(Patient p) {
        data = p;
        next = NULL;
    }
};
```

```
// Patient registry using Hash Table (Array of Nodes)
class PatientRegistry {
public:
    Node* table[10];

    PatientRegistry() {
        for (int i = 0; i < 10; i++) {
            table[i] = NULL;
        }
    }

    void addPatient(Patient p) {
        int index = p.id % 10;
        Node* newNode = new Node(p);
        newNode->next = table[index];
        table[index] = newNode;
    }

    void findPatient(int searchId) {
        int index = searchId % 10;
        Node* temp = table[index];
        bool found = false;

        while (temp != NULL) {
            if (temp->data.id == searchId) {
                cout << "Record Found: " << temp->data.name << " (Priority: " << temp->data.priority << ")" << endl;
                found = true;
                break;
            }
            temp = temp->next;
        }
        if (!found) cout << "ID not found." << endl;
    }
};


class ManualTriage {
public:
```

```
    Patient queueArray[50];
    int count;

    ManualTriage() {
        count = 0;
    }

    // Adds patient and keeps array sorted by priority
    void push(Patient p) {
        if (count < 50) {
            queueArray[count] = p;
            count++;

            // Basic bubble sort to keep highest priority at the end (index count-1)

            for (int i = 0; i < count - 1; i++) {
                for (int j = 0; j < count - i - 1; j++) {
                    if (queueArray[j].priority < queueArray[j+1].priority) {
                        Patient temp = queueArray[j];
                        queueArray[j] = queueArray[j+1];
                        queueArray[j+1] = temp;
                    }
                }
            }
        }
    }

    Patient pop() {
        Patient p = queueArray[count - 1];
        count--;
        return p;
    }

    bool isEmpty() {
        return count == 0;
    }
};

class ResourceManager {
```

```
public:
    int totalBeds;
    int totalVents;

    ResourceManager() {
        totalBeds = 5;
        totalVents = 2;
    }
};

int main() {
    ManualTriage triage;
    PatientRegistry reg;
    ResourceManager res;

    int choice, p_id, p_pri;
    string p_name;

    cout << "--- SMART HOSPITAL MANAGEMENT---" << endl;

    while (true) {
        cout << "\n1. Add Patient\n2. Treat Next\n3. Search\n4. Exit\nChoice: ";
        cin >> choice;

        switch (choice) {
            case 1:{

                cout << "ID: "; cin >> p_id;
                cout << "Name: "; cin >> p_name;
                cout << "Priority (1-High, 3-Low): "; cin >> p_pri;
                {
                    Patient newP(p_id, p_name, p_pri);
                    triage.push(newP);
                    reg.addPatient(newP);
                }}
                break;

            case 2:
                if (triage.isEmpty()) {
```

```
            cout << "No patients!" << endl;
        } else {
            Patient current = triage.pop();
            cout << "Treating: " << current.name << endl;

            if (current.priority == 1 && res.totalVents > 0) {
                res.totalVents--;
                cout << "Ventilator assigned. Vents left: " << res.totalVents << endl;
                cout<<"Patient treated.\n";
            } else if (res.totalBeds > 0) {
                res.totalBeds--;
                cout << "Bed assigned. Beds left: " << res.totalBeds << endl;
                 cout<<"Patient treated.\n";
            } else {
                cout << "No resources available!" << endl;
            }
        }
        break;

    case 3:
        int s_id;
        cout << "Enter ID: "; cin >> s_id;
        reg.findPatient(s_id);
        break;

    case 4:
        return 0;

    default:
        cout << "Invalid choice." << endl;
    }
  }
  return 0;
}
```

Thank You