CS221 – Data Structures and Algorithms

Hospital Patient Appointment and Referral Management System

A Data Structures & Algorithms (DSA) based Hospital Appointment & Referral Management System.

Shayan Rizwan Yazdanie
Department of Computer Engineering
Ghulam Ishaq Khan Institute Of Engineering Sciences and Technology (GIKI)
u2024585@giki.edu.pk

This project was assigned by our instructor, **Prof. Ahmer Rashid**, as part of our CS221 course semester project.

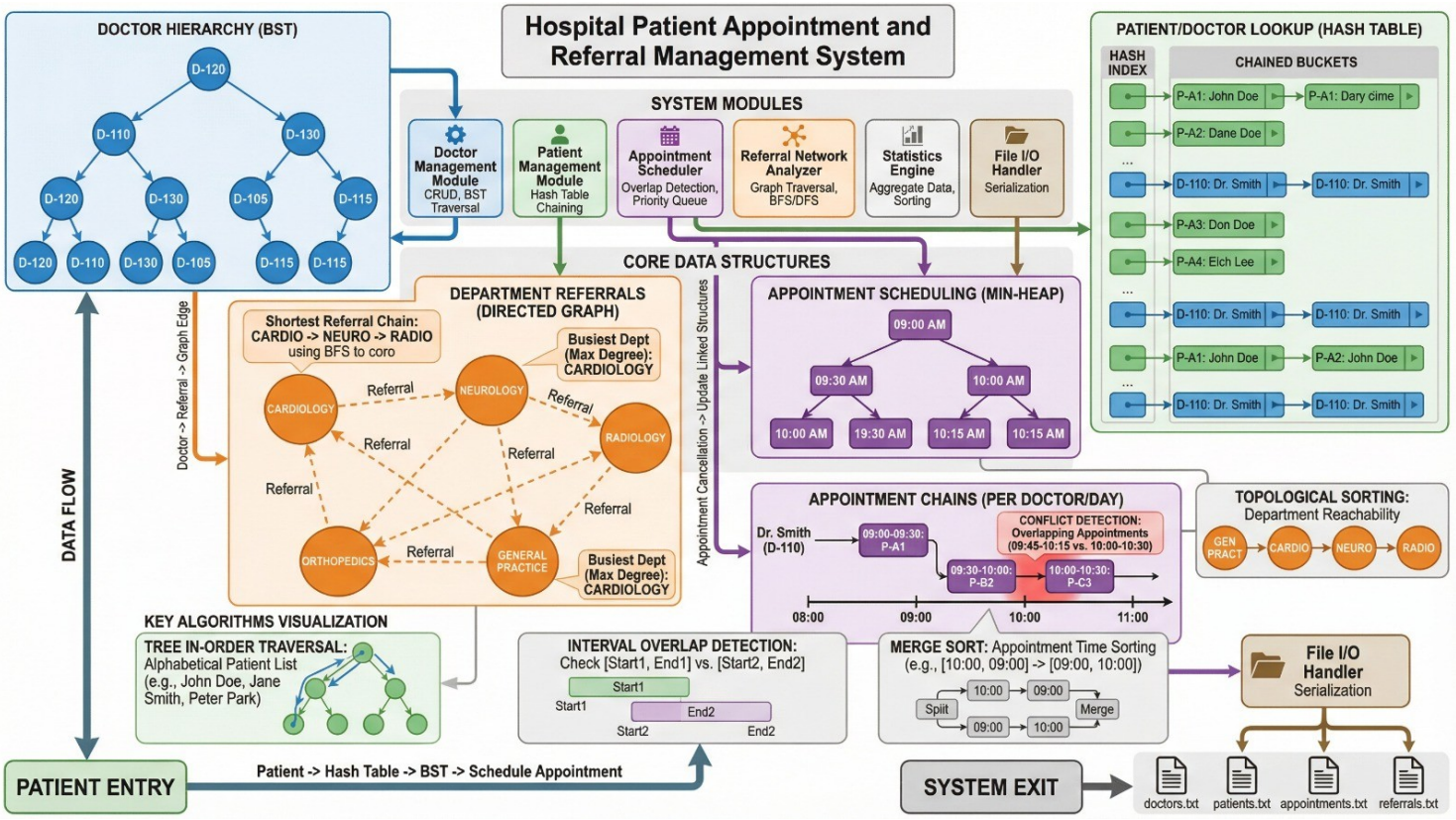

Figure 1: System Architecture Diagram

1. Project Overview

A console-based C++ application simulating an outpatient hospital clinic.
Core purpose: Demonstrate practical data structures while managing:

- ☐ Doctors, patients, appointments
- ☐ Inter-department referrals
- ☐ Schedule conflict prevention
- ☐ File-based persistence

Focus: Algorithms & data structures, not just functionality.

2. System Objectives

☐Maintain sorted doctor database
☐Schedule appointments without time overlap
☐Model referral pathways as a directed graph
☐Enable shortest-path queries between departments
☐Provide fast patient lookup
☐Persist all data in text files
☐Support clear viva explanation

3. Data Structures & Their Roles

3.1 Doctors – Binary Search Tree (BST)

```
struct Doctor {
      int doctorId;
      char* name;
      char* specialty;
      char* department;
      Appointment* appointments; // sorted linked list
      Doctor* left, *right;
};
```

Why BST:

☐Efficient searching/traversal by doctorId
☐Required by assignment
☐Natural hierarchical representation

3.2 Appointments – Sorted Linked Lists

```
struct Appointment {
      int appId, date; // yyyymmdd
      int startTime, endTime; // hhmm
      int patientId;
      Appointment* next; // sorted by date, then time
};
```

Why Linked List:

☐Efficient insert/delete
☐No resizing overhead
☐Sorting maintained at insertion

3.3 Patients – Hash Table (Separate Chaining)

```
struct Patient {
      int patientId;
      char* firstName, *lastName;
      Patient* next; // chain in same bucket
};
```

Why Hash Table:

☐O(1) average lookup by patientId
☐Required by assignment
☐Separates identity from scheduling

3.4 Departments & Referrals – Directed Graph

```
struct DeptNode {
      char* name;
      DeptEdge* firstEdge; // adjacency list
      DeptNode* next; // list of all departments
};

struct DeptEdge {
      DeptNode* to;
      DeptEdge* next; // next edge from same node
};
```

Why Graph:

☐Natural model for directional referrals
☐BFS finds shortest referral chains
☐Supports reachability queries

4. System Architecture

4.1 Component Relationships

text

Doctors (BST) → Appointments (Linked Lists) → Patients (Hash Table)
↓
Departments (Graph Nodes) ↔ Referrals (Graph Edges)

4.2 Key Design Principles

1. **Separation of concerns:** Patients don't store appointments; appointments reference patients.
2. **No cyclic dependencies:** Graph stores only department names, not doctors/patients.
3. **Data persistence:** Four text files: doctors.txt, patients.txt, appointments.txt, referrals.txt.

5. Core Algorithms Implemented

5.1 Appointment Scheduling

☐**Overlap detection:** Linear scan of doctor's appointment list
☐**Insertion:** Maintain sorted order by (date, startTime)
☐**Cancellation:** Update all affected structures

5.2 Graph Operations

☐**Reachability:** DFS from a department
☐**Shortest referral chain:** BFS between two departments
☐**Busiest department:** Count outgoing referrals

5.3 Sorting & Searching

☐**Doctor's patients alphabetically:** Extract → sort via quicksort/mergesort
☐**Appointments by time:** Maintain sorted linked list
☐**Patient lookup:** Hash table with chaining

6. System Flow

6.1 Startup

```
text

Read doctors.txt → Build BST
Read patients.txt → Build hash table
Read appointments.txt → Attach to doctors
Read referrals.txt → Build graph
```

6.2 Runtime

☐Console menu interaction

☐Each menu option targets one subsystem
☐Structures updated in real-time

6.3 Shutdown

☐Traverse all structures
☐Write to corresponding text files
☐Preserve state for next execution

7. Development Timeline

Day 1: Foundation – Struct definitions, BST, hash table, file I/O
Day 2: Core logic – Appointment scheduling, conflict detection, basic queries
Day 3: Advanced features – Graph implementation, BFS/DFS, statistics, UI polish

8. Key Features Summary

1. **Doctor management** – BST with sorted output
2. **Appointment system** – Conflict-free scheduling via linked lists
3. **Patient lookup** – O(1) average via hash table
4. **Referral modeling** – Directed graph with BFS shortest path
5. **Persistence** – Four-text-file storage
6. **Statistics** – Busiest doctor/department identification
7. **Clean UI** – Console-based with formatted tables

FUNCTION IMPLEMENTATIONS

File: src/Appointment.cpp

Implemented **9 core functions**:

FUNCTION 1: `CreateAppointment()`

Purpose: Allocate and initialize a new appointment node
Signature:

```cpp
Appointment* CreateAppointment(int appId, int date,
                    int startTime, int endTime, int patientId)
```

Returns: Pointer to newly created appointment
Complexity: O(1)
Memory: Allocates heap memory for appointment node

FUNCTION 2: `HasOverlap()`

Purpose: Detect time conflicts between appointments
Signature:

```cpp
bool HasOverlap(int date1, int start1, int end1,
           int date2, int start2, int end2)
```

Algorithm:

```text
overlap = (date1 == date2) AND (start1 < end2) AND (end1 > start2)
```

Returns: `true` if overlap exists, `false` otherwise
Complexity: O(1)
Edge Cases:

•Back-to-back appointments allowed (e.g., 1500-1600, 1600-1700)
•Different dates never overlap

FUNCTION 3: `AddAppointment()` – CORE LOGIC

Purpose: Insert appointment with overlap detection and sorted order

Signature:

```cpp
bool AddAppointment(Doctor* root, int doctorId, int appId,
             int date, int startTime, int endTime, int patientId)
```

Requirements Met: Requirement 4 (10 marks)

Algorithm:

1. **Step 1:** Find doctor in BST using `FindDoctor()`
2. **Step 2:** Check all existing appointments for overlaps
3. **Step 3:** If overlap found → reject and return `false`
4. **Step 4:** Create new appointment node
5. **Step 5:** Insert in sorted order (date → time)
6. **Step 6:** Return `true` (success)

Sorting Logic:

- **Primary key:** date (ascending) – integer format: `YYYYMMDD`
- **Secondary key:** startTime (ascending) – integer format: `HHMM`
- Example: `20250617 < 20250618` (June 17 before June 18)

Complexity: O(M) where M = appointments for that doctor

Memory: O(1) auxiliary space

FUNCTION 4: `PrintDoctorSchedule()`

Purpose: Display all appointments for a doctor on specific date
Signature:

```cpp
void PrintDoctorSchedule(Doctor* root, int doctorId, int date)
```

Requirements Met: Requirement 2 (8 marks)

Algorithm:

1. **Step 1:** Find doctor in BST
2. **Step 2:** Traverse appointment linked list
3. **Step 3:** Filter by date (show only matching date)
4. **Step 4:** Display in formatted table

Output Format:

```text
========== Schedule for Dr. [Name] on [Date] ==========
1. Appointment ID: X | Patient ID: Y | Time: HHMM - HHMM
2. Appointment ID: X | Patient ID: Y | Time: HHMM - HHMM
=====================================================
```

Complexity: O(M) where M = appointments for that doctor

FUNCTION 5: `PrintPatientAppointments()`

Purpose: Display all appointments for a patient across all doctors
Signature:

```cpp
void PrintPatientAppointments(Doctor* root, int patientId)
```

Requirements Met: Requirement 3 (7 marks)

Algorithm:

1. **Step 1:** Traverse entire Doctor BST (in-order traversal)
2. **Step 2:** For each doctor, check their appointment list
3. **Step 3:** Display appointments matching `patientId`

Helper Function: Uses `PrintPatientAppointmentsHelper()`

- Recursive in-order BST traversal
- Ensures sorted display by doctor ID

Complexity: O(N × M) where N = doctors, M = avg appointments/doctor

FUNCTION 6: `PrintPatientAppointmentsHelper()`

Purpose: Recursive helper for `PrintPatientAppointments()`
Signature:

```cpp
void PrintPatientAppointmentsHelper(Doctor* root,
                        int patientId, bool& found)
```

Pattern: In-order BST traversal
Complexity: O(N) for tree traversal + O(M) per node

FUNCTION 7: `CancelAppointment()` – PUBLIC INTERFACE

Purpose: Remove appointment by ID with user feedback
Signature:

```cpp
bool CancelAppointment(Doctor* root, int appId)
```

Requirements Met: Requirement 5 (9 marks)

Algorithm:

1. **Step 1:** Call `CancelAppointmentHelper()` to perform deletion
2. **Step 2:** Display success/failure message to user
3. **Step 3:** Return status (`true`/`false`)

Design Pattern: Separation of concerns

- This function: User interface
- Helper functions: Business logic

Complexity: O(N × M) worst case

FUNCTION 8: `CancelAppointmentHelper()` – BST TRAVERSAL

Purpose: Search all doctors for appointment to cancel

Signature:

```cpp
bool CancelAppointmentHelper(Doctor* root, int appId)
```

Algorithm:

Base case: if `root == nullptr`, return `false`

1. **Step 1:** Try to cancel from current doctor's list
2. **Step 2:** If found, return `true` (early exit optimization)
3. **Step 3:** Recursively search left subtree
4. **Step 4:** If found in left, return `true`
5. **Step 5:** Recursively search right subtree
6. **Step 6:** Return result from right

Optimization: Early exit prevents unnecessary traversal

Pattern: Modified preorder traversal

FUNCTION 9: `CancelAppointmentFromDoctor()` – LIST DELETION

Purpose: Remove appointment from a specific doctor's list
Signature:

```cpp
bool CancelAppointmentFromDoctor(Doctor* doc, int appId)
```

Algorithm: Classic two-pointer linked list deletion

```cpp
curr = doc->appointments
prev = nullptr

while (curr != nullptr):
   if (curr->appId == appId):
      if (prev == nullptr):
         doc->appointments = curr->next  // Delete head
      else:
         prev->next = curr->next  // Delete middle/tail
      delete curr  // Free memory
      return true
   prev = curr
   curr = curr->next

return false  // Not found
```

Memory Safety: Properly frees deleted node with `delete`
Handles: Head deletion, middle deletion, tail deletion

Concept 1: Why Integer Date/Time?

Integer format allows trivial comparison. Since dates are yyyymmdd, comparing 20250617 < 20250618 is direct. String comparison would require parsing. This matches the specification and simplifies the sorting algorithm in AddAppointment().

Concept 2: Overlap Detection Logic
Two appointments overlap if they're on the same date AND their time intervals intersect. The condition (start1 < end2) AND (end1 > start2) captures this mathematically. Note the strict inequalities allow back-to-back appointments like [1400-1500] and [1500-1600].

Concept 3: Sorted Insertion
I insert appointments in sorted order immediately rather than inserting anywhere and sorting later. This is O(M) insertion vs O(M log M) for sorting. The comparison is: primary key is date, secondary key is time. Display functions can now simply traverse the list.

Concept 4: Three-Function Cancellation
I separated concerns: CancelAppointment handles user interface, CancelAppointmentHelper traverses the BST recursively, and CancelAppointmentFromDoctor manipulates the linked list. This follows the Single Responsibility Principle and makes testing easier.

Concept 5: Two-Pointer List Deletion
In a singly linked list, to delete a node, I need to update the previous node's 'next' pointer. I maintain 'prev' and 'curr' pointers. Special case: when deleting the head, prev is null, so I update the list head pointer directly instead of prev→next.

During the implementation of the whole Appointments structure, there were **several errors** that were faced.

CIRCULAR DEPENDENCY

Problem:
Doctor.h includes Appointment.h
Appointment.h needs Doctor* in function signatures
→ Circular dependency compilation error

Solution:
Added forward declaration in Appointment.h:
struct Doctor; // Forward declaration

Now:
- Appointment.h declares Doctor exists (no include)
- Doctor.h includes Appointment.h (safe)
- Appointment.cpp includes Doctor.h (gets full definition)

Result: Compilation successful, no circular dependency

OCTAL LITERAL ERROR

Problem:
Time value 0900 interpreted as octal (base-8)
Digit '9' is invalid in octal (only 0-7 allowed)
Compiler error: "invalid digit '9' in octal constant"

Solution:
Changed 0900 to 900
Rationale: Integer time format doesn't need leading zero
900 correctly represents 9:00 AM in hhmm format

Result: Compilation successful

OVERLAP DETECTION EDGE CASE

Problem:
Should back-to-back appointments be allowed?
Example: Apt1 ends at 1500, Apt2 starts at 1500

Solution:

Used strict inequality: (start1 < end2) AND (end1 > start2)
Not: (start1 <= end2) AND (end1 >= start2)

This allows:
[1400-1500] and [1500-1600] → NO overlap ✓

This rejects:
[1400-1500] and [1430-1530] → Overlap ✗

Result: Back-to-back appointments allowed, overlaps rejected

In order to construct the Department Graph Module, we make use of an adjacency list.

Graph Structure with Adjacency Lists

The referral system implements a directed graph using an adjacency list representation, composed of two core data structures: **DeptNode** for vertices (departments) and **DeptEdge** for edges (referrals). Let's break this down with actual code and concrete examples.

The Two Core Structures

```
struct DeptNode {
    char* name;          // Department name string (e.g., "Cardiology")
    DeptEdge* firstEdge;  // Head of this department's referral list
    DeptNode* next;       // Link to next department in global list
};
```

Each **DeptNode** is like an entry in a hospital directory. It stores:

☐ A dynamically allocated string for the department name

- A pointer to the start of its personal referral list
- A pointer to the next department in the master directory

```
struct DeptEdge {
    DeptNode* to;        // Pointer to the destination department
    DeptEdge* next;      // Next referral in this department's list
};
```

Each **DeptEdge** represents a single referral pathway. The `to` pointer doesn't contain department data—it points to where that data lives.

Visualizing the Structure

Consider this hospital referral network:

- Cardiology can refer to Neurology and Radiology
- Neurology can refer to Radiology

Here's how it looks in memory:

cpp

```
// Global department list (horizontal chain)
deptHead → [Cardiology] → [Neurology] → [Radiology] → NULL
             |              |            |
             ↓              ↓            ↓
          [Edge] → [Edge]  [Edge]       NULL
            ↓        ↓       ↓
          Neuro    Radio   Radio
```

When we call `AddReferral("Cardiology", "Neurology")`:

cpp

```
// Step 1: Get/create nodes
DeptNode* cardiology = GetOrCreateDept("Cardiology");
DeptNode* neurology = GetOrCreateDept("Neurology");

// Step 2: Create edge
```

```
DeptEdge* newEdge = new DeptEdge;
newEdge->to = neurology;          // Points to Neurology node
newEdge->next = cardiology->firstEdge;  // Chain to existing edges
cardiology->firstEdge = newEdge;   // Make this the new head
```

The adjacency list grows dynamically. If we later add `AddReferral("Cardiology", "Radiology")`:

```
cpp
// Creates another edge, inserted at FRONT of Cardiology's list
// Cardiology's edge list becomes: Radiology → Neurology → NULL
```

Why This Design?

1. **Memory Efficiency**: Only stores actual referrals, not all possible combinations
2. **Fast Neighbor Access**: To see where Cardiology can refer, just traverse its edge list
3. **Dynamic Growth**: No fixed limits on referrals per department

```
cpp
// Iterating through Cardiology's referrals:
DeptEdge* edge = cardiologyNode->firstEdge;
while (edge != nullptr) {
    cout << "Can refer to: " << edge->to->name << endl;
    edge = edge->next;
}
// Output: "Can refer to: Radiology" then "Can refer to: Neurology"
```

Understanding Pointer Arrays in IsVisited

What is `DeptNode** visited`?

The `IsVisited` function uses an **array of pointers**, not an array of objects. Here's the distinction:

```
cpp
```

```
// WRONG understanding: Array of objects
DeptNode visitedObjects[100];  // Stores 100 FULL DeptNode copies (inefficient!)

// CORRECT implementation: Array of pointers
DeptNode* visitedPointers[100]; // Stores 100 MEMORY ADDRESSES (4/8 bytes each)
```

When we pass this to IsVisited, we use DeptNode** visited which means "pointer to pointer to DeptNode"—or more practically, "the address of the first element in an array of DeptNode pointers."

Concrete Example in Action

Let's trace what happens during graph traversal:

cpp

```
// Suppose we have these memory addresses:
// Cardiology node @ 0x1000
// Neurology node @ 0x2000
// Radiology node @ 0x3000

// Starting DFS from Cardiology
DeptNode* visited[100];
int visitedCount = 0;

visited[visitedCount++] = cardiologyNode;  // visited[0] = 0x1000

// Now visited array contains: [0x1000, ...]
// Not the string "Cardiology", not the entire DeptNode structure
// Just the NUMBER 0x1000 (a memory address)
```

When we check if Neurology has been visited:

cpp

```
bool IsVisited(DeptNode** visited, int visitedCount, DeptNode* node) {
    // node contains 0x2000 (Neurology's address)
    for (int i = 0; i < visitedCount; i++) {
        // Compare: visited[i] (0x1000) == node (0x2000)?
```

```
        if (visited[i] == node) {
            return true;  // Match found
        }
    }
    return false;  // No match
}
```

Why Pointer Comparison Works

Each department node is created **once** with `new DeptNode`. Even if two departments had the same name (which shouldn't happen), they'd be at different memory addresses. This gives us uniqueness without string comparison.

cpp

```
// Address-based uniqueness guarantee
DeptNode* dept1 = GetOrCreateDept("Cardiology");  // Returns address 0x1000
DeptNode* dept2 = GetOrCreateDept("Cardiology");  // ALSO returns 0x1000 (same node!)

// So visited array tracking works perfectly:
// If we visited cardiology once (stored 0x1000)
// Then dept2 is ALSO at 0x1000, so IsVisited returns true
```

Memory Layout Visualization

text

```
VISITED ARRAY (in memory):
Address: 0x5000 0x5008 0x5010 ...  (8-byte pointers on 64-bit)
Data:    [0x1000][0x2000][0x3000]  (Just addresses!)
           ↑       ↑       ↑
           |       |       |
           v       v       v
ACTUAL NODES: [Cardiology][Neurology][Radiology]
                @0x1000    @0x2000    @0x3000
```

This design is extremely efficient:

1. **Space**: Each visited entry is 8 bytes (pointer), not sizeof(DeptNode) bytes

2. **Speed**: Integer pointer comparison is O(1), string comparison would be O(n)
3. **Correctness**: Guaranteed uniqueness via memory addresses

The Double Pointer Explained

cpp

```
// When we call: IsVisited(visited, visitedCount, node)
// We're passing 'visited' which decays to &visited[0]
// So inside the function:
// visited[i] means: "Get the pointer stored at position i"
// Which is equivalent to: "*(visited + i)"
// So we're dereferencing TWICE:
// 1. visited + i gives address of array element
// 2. *(visited + i) gives the DeptNode* stored there
```

In practice, `DeptNode** visited` lets the function access the original visited array, not a copy, so visited nodes stay tracked across recursive DFS calls.

Table 1: Use Of AI

ChatGPT and Claude were employed as the only supplementary resources throughout the development of this project. **Specific Applications:** • **ChatGPT:** Primarily for conceptual ideation, exploring implementation approaches, and establishing workflow structures. • **Claude:** Primarily for in-depth technical documentation, algorithm comprehension, and refining implementation details. It can generate comprehensive documentations. **Notable Observations:** Claude demonstrated particular effectiveness in error handling and debugging scenarios, which proved valuable for time-efficient development and project completion. **Nature of Use:** All AI assistance was advisory. Final implementations, code structure, design decisions, and documentation were critically evaluated, adapted, and ultimately done by me.

You can find my code at GitHub as well:

https://github.com/CodeBySRY/CareSync-Hub