DATA STRUCTURE AND ALGORITHM

Sir Taha

Muhammad Faheem 42346
Asmar Mehmood 337183

JANUARY 29, 2025

IQRA UNIVERSITY

Hospital Patients Management System

The provided C++ code is an implementation of a hospital management system where we can manage patient data using various data structures like Singly Linked List, Stack, Queue, and Binary Search Tree (BST). It allows the user to interact with these structures to add, display, delete, and traverse patient records.

1. Patient Class

The **Patient** class serves as the primary data structure to store individual patient details. The class contains:

- **Attributes:** Patient ID, Name, Age, Symptoms, Admission Date, and Discharge Date.
- **Methods:**
    - input_patient(): Accepts patient information from the user.
    - display_patient(): Displays the patient's details.
    - get_id(): Retrieves the patient ID.
    - Overloaded == operator to compare patients based on their ID.

2. Singly Linked List Class

The **SinglyLinkedList** class is used to store patient records in a linked list structure. It supports:

- **insertPatient()**: Inserts a new patient at the end of the list.
- **deletePatient()**: Deletes a patient based on the provided ID.
- **isEmpty()**: Checks whether the list is empty.
- **display()**: Displays the entire list of patients.
- **foundDuplicate()**: Checks if there are any duplicate patients in the list by comparing patient IDs.

3. Stack Class

The **Stack** class implements a stack using a linked list. It offers:

- **pushPatient()**: Pushes a patient to the top of the stack.
- **popPatient()**: Pops the top patient from the stack.
- **top()**: Retrieves the patient at the top of the stack.
- **is_empty()**: Checks if the stack is empty.
- **display()**: Displays all patients in the stack.

- **foundDuplicate()**: Identifies duplicate patients in the stack.

4. Queue Class

The **Queue** class implements a queue using a linked list. It provides:

- **enqueuePatient()**: Enqueues a patient at the rear of the queue.
- **dequeuePatient()**: Dequeues a patient from the front of the queue.
- **front()**: Retrieves the patient at the front of the queue.
- **is_empty()**: Checks if the queue is empty.
- **display()**: Displays all patients in the queue.
- **foundDuplicate()**: Checks for duplicate patients in the queue.

5. Binary Search Tree (BST) Class

The **BinarySearchTree** class maintains patient records in a binary search tree. It supports:

- **insertPatient()**: Inserts a patient into the tree based on their patient ID.
- **deletePatient()**: Deletes a patient from the tree by their ID.
- **foundDuplicate()**: Checks for duplicate patients in the tree.
- **inorder_traversal()**, **preorder_traversal()**, **postorder_traversal()**: These methods traverse the tree in different orders and display patient details.
- **display()**: Displays all patients in the BST by performing an inorder traversal.

6. Main Function

The **main()** function implements a menu-driven interface that allows the user to perform various operations. It includes the following functionalities:

- **Add Patient**: Accepts patient information and adds it to the Singly Linked List, Stack, Queue, and BST.
- **Display Patients**: Displays patients in the linked list.
- **Delete from List**: Deletes a patient from the linked list by ID.
- **Push/Pop Stack**: Adds or removes a patient from the stack.
- **Enqueue/Dequeue Queue**: Adds or removes a patient from the queue.
- **Insert to BST**: Inserts a patient into the BST.
- **Traversal (Inorder, Preorder, Postorder)**: Performs tree traversal to display patients in different orders.
- **Exit**: Terminates the program.

Key Features:

- **Patient Management**: The system allows easy management of patient data using multiple data structures, providing flexibility in how data is stored and retrieved.
- **Duplicate Check**: Each data structure (Singly Linked List, Stack, Queue, BST) has a method to check for duplicate patient entries based on patient ID.
- **Traversal**: The BST provides various traversal techniques to view patient data in different orders, which can be useful for search and reporting purposes.

CODE:

```
#include <iostream>
#include <string>
using namespace std;


class Patient {
private:
    int patient_id;
    string name;
    int age;
    string symptoms;
    string admission_date;
    string discharge_date;

public:
    Patient() : patient_id(0), name(""), age(0), symptoms(""), admission_date(""), discharge_date("") {}

    void input_patient() {
        cout << "Enter Patient ID: ";
        cin >> patient_id;
        cin.ignore();
        cout << "Enter Name: ";
        getline(cin, name);
        cout << "Enter Age: ";
```

```
        cin >> age;
        cin.ignore();
        cout << "Enter Symptoms: ";
        getline(cin, symptoms);
        cout << "Enter Admission Date: ";
        getline(cin, admission_date);
        cout << "Enter Discharge Date: ";
        getline(cin, discharge_date);
    }

    void display_patient() {
        cout << "ID: " << patient_id << ", Name: " << name << ", Age: " << age
             << ", Symptoms: " << symptoms << ", Admission Date: " << admission_date
             << ", Discharge Date: " << discharge_date << endl;
    }

    int get_id() const {
        return patient_id;
    }

    bool operator==(const Patient& other) const {
        return patient_id == other.patient_id;
    }
};


class SinglyLinkedList {
private:
    struct Node {
        Patient patient;
        Node* next;

        Node(const Patient& pat) : patient(pat), next(nullptr) {}
```

```
    };

    Node* head;

public:
    SinglyLinkedList() : head(nullptr) {}

    void insertPatient(const Patient& patient) {
        Node* new_node = new Node(patient);
        if (!head) {
            head = new_node;
        }
        else {
            Node* temp = head;
            while (temp->next) {
                temp = temp->next;
            }
            temp->next = new_node;
        }
    }

    void deletePatient(int patientID) {
        if (head == nullptr) {
            std::cout << "List, Stack, Queue is empty." << std::endl;
            return;
        }

        if (head->patient.get_id() == patientID) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }
```

```
    Node* current = head;
    while (current->next && current->next->patient.get_id() != patientID) {
        current = current->next;
    }

    if (current->next == nullptr) {
        std::cout << "Patient not found." << std::endl;
    }
    else {
        Node* temp = current->next;
        current->next = current->next->next;
        delete temp;
    }
}


bool isEmpty() const {
    return head == nullptr;
}

void display() {
    if (isEmpty()) {
        std::cout << "List is empty." << std::endl;
        return;
    }

    Node* temp = head;
    while (temp) {
        temp->patient.display_patient();
        temp = temp->next;
    }
}
```

```
    bool foundDuplicate() {
        Node* outer = head;
        while (outer) {
            Node* inner = outer->next;
            while (inner) {
                if (outer->patient == inner->patient) {
                    std::cout << "Duplicate patient found!" << std::endl;
                    return true;
                }
                inner = inner->next;
            }
            outer = outer->next;
        }
        return false;
    }
};



class Stack {
private:
    struct Node {
        Patient patient;
        Node* next;

        Node(const Patient& pat) : patient(pat), next(nullptr) {}
    };

    Node* top_node;

public:
    Stack() : top_node(nullptr) {}
```

```
void pushPatient(const Patient& patient) {
    Node* new_node = new Node(patient);
    new_node->next = top_node;
    top_node = new_node;
}

void popPatient() {
    if (top_node) {
        Node* temp = top_node;
        top_node = top_node->next;
        delete temp;
    }
    else {
        std::cout << "List, Stack, Queue is empty." << std::endl;
    }
}

Patient top() {
    if (top_node) {
        return top_node->patient;
    }
    return Patient();
}

bool is_empty() const {
    return top_node == nullptr;
}

void display() {
    if (is_empty()) {
        std::cout << "Stack is empty." << std::endl;
        return;
    }
```

```
        Node* temp = top_node;
        while (temp) {
            temp->patient.display_patient();
            temp = temp->next;
        }
    }

    bool foundDuplicate() {
        Node* outer = top_node;
        while (outer) {
            Node* inner = outer->next;
            while (inner) {
                if (outer->patient == inner->patient) {
                    std::cout << "Duplicate patient found!" << std::endl;
                    return true;
                }
                inner = inner->next;
            }
            outer = outer->next;
        }
        return false;
    }
};

class Queue {
private:
    struct Node {
        Patient patient;
        Node* next;

        Node(const Patient& pat) : patient(pat), next(nullptr) {}
    };
```

```
    Node* front_node;
    Node* rear_node;

public:
    Queue() : front_node(nullptr), rear_node(nullptr) {}

    void enqueuePatient(const Patient& patient) {
        Node* new_node = new Node(patient);
        if (rear_node) {
            rear_node->next = new_node;
        }
        rear_node = new_node;
        if (!front_node) {
            front_node = rear_node;
        }
    }

    void dequeuePatient() {
        if (front_node) {
            Node* temp = front_node;
            front_node = front_node->next;
            if (!front_node) {
                rear_node = nullptr;
            }
            delete temp;
        }
        else {
            std::cout << "List, Stack, Queue is empty." << std::endl;
        }
    }

    Patient front() {
```

```
    if (front_node) {
        return front_node->patient;
    }
    return Patient();
}


bool is_empty() const {
    return front_node == nullptr;
}


void display() {
    if (is_empty()) {
        std::cout << "Queue is empty." << std::endl;
        return;
    }

    Node* temp = front_node;
    while (temp) {
        temp->patient.display_patient();
        temp = temp->next;
    }
}

bool foundDuplicate() {
    Node* outer = front_node;
    while (outer) {
        Node* inner = outer->next;
        while (inner) {
            if (outer->patient == inner->patient) {
                std::cout << "Duplicate patient found!" << std::endl;
                return true;
```

```
            }
            inner = inner->next;
        }
        outer = outer->next;
    }
    return false;
  }
};

class BinarySearchTree {
private:
  struct Node {
    Patient patient;
    Node* left;
    Node* right;

    Node(const Patient& pat) : patient(pat), left(nullptr), right(nullptr) {}
  };

  Node* root;

  Node* insert(Node* node, const Patient& patient) {
    if (!node) return new Node(patient);
    if (patient.get_id() < node->patient.get_id()) {
      node->left = insert(node->left, patient);
    }
    else if (patient.get_id() > node->patient.get_id()) {
      node->right = insert(node->right, patient);
    }
    return node;
  }

  Node* deleteNode(Node* node, int patientID) {
```

```
    if (!node) return node;

    if (patientID < node->patient.get_id()) {
        node->left = deleteNode(node->left, patientID);
    }
    else if (patientID > node->patient.get_id()) {
        node->right = deleteNode(node->right, patientID);
    }
    else {
        if (!node->left) {
            Node* temp = node->right;
            delete node;
            return temp;
        }
        else if (!node->right) {
            Node* temp = node->left;
            delete node;
            return temp;
        }

        Node* temp = minValueNode(node->right);
        node->patient = temp->patient;
        node->right = deleteNode(node->right, temp->patient.get_id());
    }
    return node;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left) {
        current = current->left;
    }
    return current;
```

```
    }

    bool foundDuplicate(Node* node, const Patient& patient) {
        if (!node) return false;
        if (node->patient == patient) return true;
        return foundDuplicate(node->left, patient) || foundDuplicate(node->right, patient);
    }

    void inorder(Node* node) {
        if (node) {
            inorder(node->left);
            node->patient.display_patient();
            inorder(node->right);
        }
    }

    void preorder(Node* node) {
        if (node) {
            node->patient.display_patient();
            preorder(node->left);
            preorder(node->right);
        }
    }

    void postorder(Node* node) {
        if (node) {
            postorder(node->left);
            postorder(node->right);
            node->patient.display_patient();
        }
    }

public:
```

```
BinarySearchTree() : root(nullptr) {}


void insertPatient(const Patient& patient) {
    root = insert(root, patient);
}

void deletePatient(int patientID) {
    root = deleteNode(root, patientID);
}

bool foundDuplicate(const Patient& patient) {
    return foundDuplicate(root, patient);
}

void inorder_traversal() {
    if (root) {
        inorder(root);
    }
    else {
        std::cout << "Tree is empty." << std::endl;
    }
}

void preorder_traversal() {
    if (root) {
        preorder(root);
    }
    else {
        std::cout << "Tree is empty." << std::endl;
    }
}
```

```
    void postorder_traversal() {
        if (root) {
            postorder(root);
        }
        else {
            std::cout << "Tree is empty." << std::endl;
        }
    }

    void display() {
        if (root) {
            inorder_traversal();
        }
        else {
            std::cout << "Tree is empty." << std::endl;
        }
    }

    bool isEmpty() {
        return root == nullptr;
    }
};

int main() {
    SinglyLinkedList list;
    Stack stack;
    Queue queue;
    BinarySearchTree bst;

    int choice;
    do {
        cout << "1. Add Patient\n"
            << "2. Display Patients (Linked List)\n"
```

```
    << "3. Delete from List\n"
    << "4. Push to Stack\n"
    << "5. Display Stack\n"
    << "6. Pop from Stack\n"
    << "7. Enqueue to Queue\n"
    << "8. Display Queue\n"
    << "9. Dequeue from Queue\n"
    << "10. Insert to BST\n"
    << "11. Inorder Traversal\n"
    << "12. Preorder Traversal\n"
    << "13. Postorder Traversal\n"
    << "0. Exit\n"
    << "Choose an option: ";
cin >> choice;
cin.ignore();

switch (choice) {
case 1: {
    Patient patient;
    patient.input_patient();
    list.insertPatient(patient);
    stack.pushPatient(patient);
    queue.enqueuePatient(patient);
    bst.insertPatient(patient);
    break;
}
case 2:
    list.display();
    break;
case 3: {
    int patientID;
    cout << "Enter Patient ID to delete: ";
    cin >> patientID;
```

```
    list.deletePatient(patientID);
    break;
}
case 4: {
    Patient patient;
    patient.input_patient();
    stack.pushPatient(patient);
    break;
}
case 5:
    stack.display();
    break;
case 6:
    if (!stack.is_empty()) {
        stack.popPatient();
        cout << "Top patient removed from stack.\n";
    }
    else {
        cout << "Stack is empty.\n";
    }
    break;
case 7: {
    Patient patient;
    patient.input_patient();
    queue.enqueuePatient(patient);
    break;
}
case 8:
    queue.display();
    break;
case 9:
    if (!queue.is_empty()) {
        queue.dequeuePatient();
```

```
            cout << "Front patient removed from queue.\n";
        }
        else {
            cout << "Queue is empty.\n";
        }
        break;
    case 10: {
        Patient patient;
        patient.input_patient();
        bst.insertPatient(patient);
        break;
    }
    case 11:
        cout << "Inorder Traversal:\n";
        bst.inorder_traversal();
        break;
    case 12:
        cout << "Preorder Traversal:\n";
        bst.preorder_traversal();
        break;
    case 13:
        cout << "Postorder Traversal:\n";
        bst.postorder_traversal();
        break;
    case 0:
        cout << "Exiting...\n";
        break;
    default:
        cout << "Invalid choice.\n";
    }
} while (choice != 0);

return 0;
```

}

OUTPUT:

Add Patient

```
1. Add Patient
2. Display Patients (Linked List)
3. Delete from List
4. Push to Stack
5. Display Stack
6. Pop from Stack
7. Enqueue to Queue
8. Display Queue
9. Dequeue from Queue
10. Insert to BST
11. Inorder Traversal
12. Preorder Traversal
13. Postorder Traversal
0. Exit
Choose an option: 1
Enter Patient ID: 42346
Enter Name: fAHEEM
Enter Age: 23
Enter Symptoms: Fever
Enter Admission Date: 11
Enter Discharge Date: 15
```

Display List:

```
Choose an option: 2
ID: 42346, Name: fAHEEM, Age: 23, Symptoms: Fever, Admission Date: 11, Discharge
    Date: 15
```

Delete From List:

```
Choose an option: 2
List is empty.
```

Push To Stack:

```
Choose an option: 4
Enter Patient ID: 42346
Enter Name: Faheem
Enter Age: 23
Enter Symptoms: Fever
Enter Admission Date: 12
Enter Discharge Date: 11
```

Display Stack:

```
Choose an option: 5
ID: 42346, Name: Faheem, Age: 23, Symptoms: Fever, Admission Date: 12, Discharge
    Date: 11
```

POP From Stack:

```
Choose an option: 6
Top patient removed from stack.
```