# Data structures and Algorithm

## Project document

## *Pharmacy management system*

**Group members:**

AREEB RAZA(23009105073)

SAMR IMRAN(23009105010)

FATIMA AMIR(23009105020)

HASEEB USMAN(23009105018)

**Department:**

BS IT (BATCH 9) (A)

**Semester:**

3rd

**Submitted to:**

MA'AM JANNAT

MA'AM AQSA ZAHID

## Overview:

The Pharmacy Management System is a comprehensive C++ program designed to manage the daily operations of a pharmacy. It encompasses various features such as

- Managing inventory,
- Handling customer requests,
- Processing prescriptions, and
- Maintaining patient records in a binary search tree.

The system is built using dynamic **data structures** like

- Arrays
- Linked lists
- Binary search trees

ensuring efficient storage and retrieval of data.

**Key Features:**

- **Inventory Management**: (Using **Arrays**)

  Manage medicines in stock,

  1. Update quantities of existing medicines,
  2. Restock medicines, and
  3. Add new medicines.

- **Prescription Handling**:

  Manage customer prescriptions, including

  1. Adding,
  2. Deleting,
  3. Updating, and
  4. Displaying prescriptions.

- **Customer Requests**:

  Process customer requests in a FIFO **queue** fashion.

- **Patient Records**:

  Maintain patient records using a **binary search tree** with various operations like

  1. Inserting,
  2. Deleting, and
  3. Searching patients.

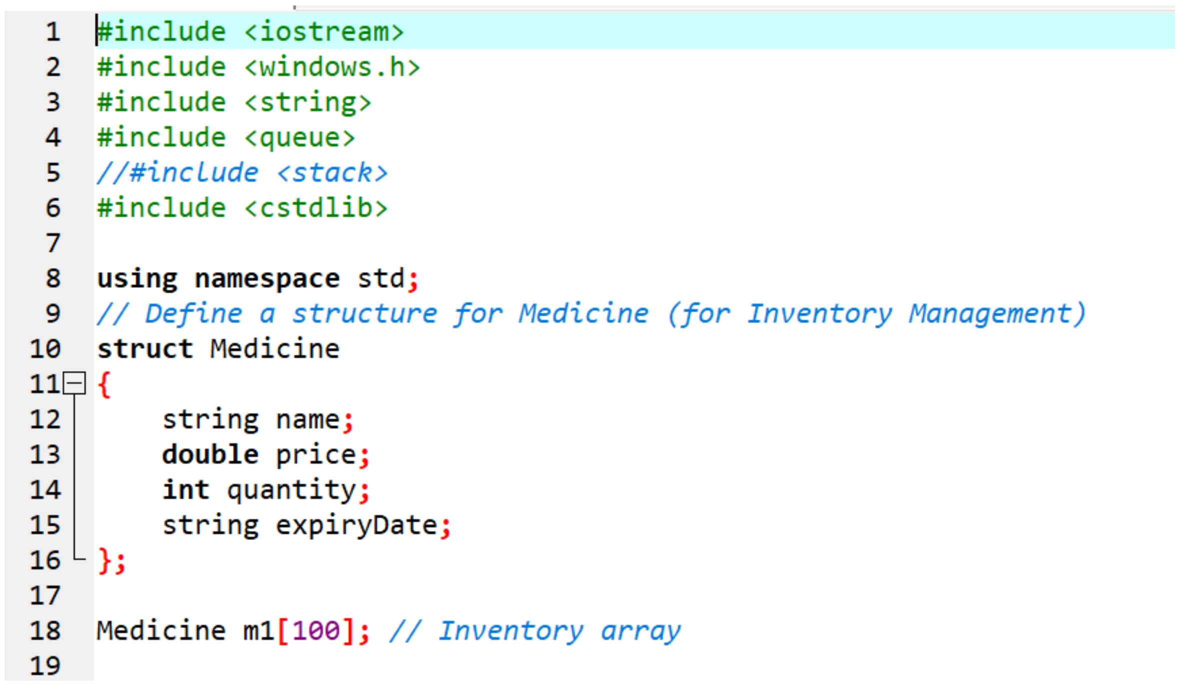
## Data Structures:

### 1. Medicine Structure

This structure represents a single medicine in the pharmacy's inventory. It has the **medicine's**

1. Name,
2. Price,
3. Quantity,
4. ExpiryDate along with
5. A **global object** of the structure **m1** having size **100.**

## Code implementation



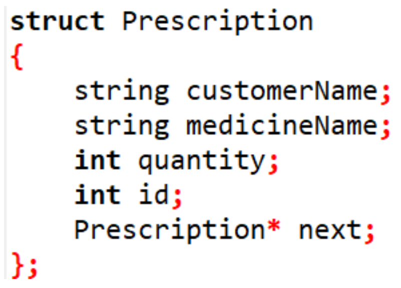
```
#include <iostream>
#include <windows.h>
#include <string>
#include <queue>
//#include <stack>
#include <cstdlib>

using namespace std;
// Define a structure for Medicine (for Inventory Management)
struct Medicine
{
    string name;
    double price;
    int quantity;
    string expiryDate;
};

Medicine m1[100]; // Inventory array

```

## 2. Prescription Structure

This structure represents a prescription issued to a customer and kept in the pharmacy’s DB. It contains

1. CustomerName
2. medicineName
3. quantity
4. id
5. and the next **pointer** of the **prescription type** for moving forward in the **linked list**

## <u>Code implementation</u>



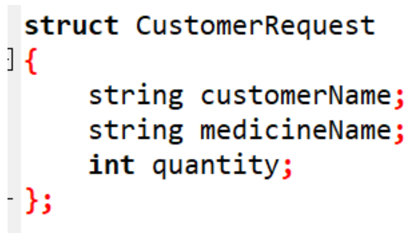
```
struct Prescription
{
    string customerName;
    string medicineName;
    int quantity;
    int id;
    Prescription* next;
};
```

## 3. CustomerRequest Structure

This structure holds information regarding a customer’s medicine request. It contains

1. CustomerName
2. medicineName
3. quantity

## <u>Code implementation</u>



```
struct CustomerRequest
{
    string customerName;
    string medicineName;
    int quantity;
};
```

4. **PatientNode Structure:**

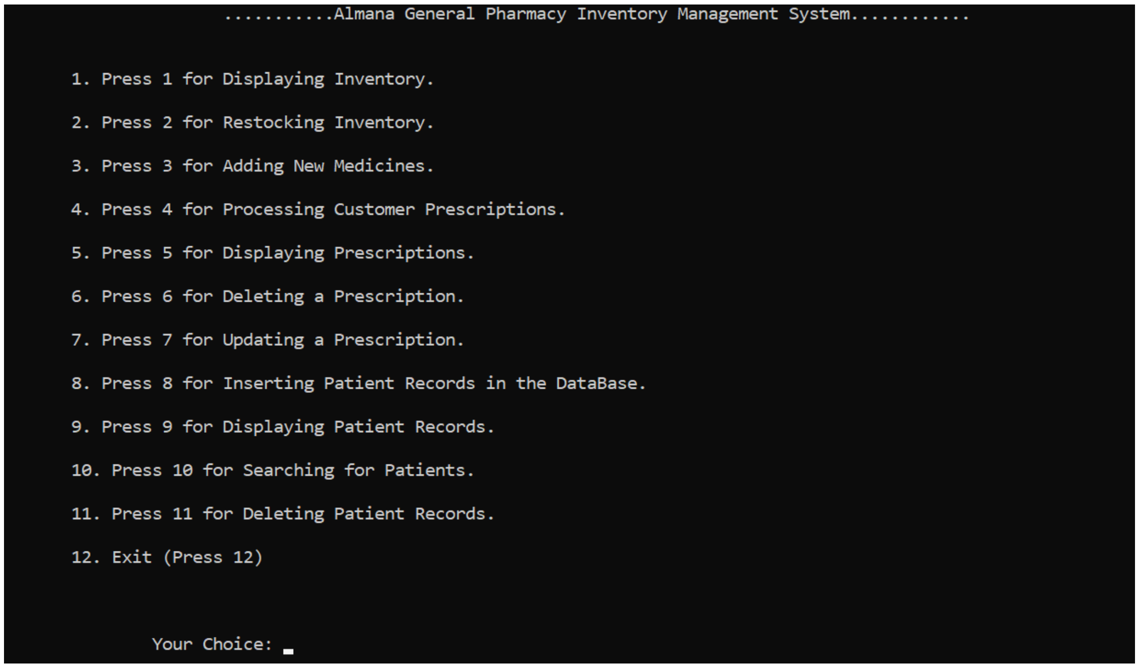
   The patient node's structure in the code represents the node used for storing patients records. It contains the following elements

5. PatientName
6. PrescriptionHistory
7. Id
8. Left and right pointers of the patient node type to move left and right in the tree.

## Code implementation

```
struct PatientNode
{
    string patientName;
    string prescriptionHistory;
    int ID;
    PatientNode* left;
    PatientNode* right;
};
```

## Display page



```
                    ...........Almana General Pharmacy Inventory Management System...........


      1. Press 1 for Displaying Inventory.

      2. Press 2 for Restocking Inventory.

      3. Press 3 for Adding New Medicines.

      4. Press 4 for Processing Customer Prescriptions.

      5. Press 5 for Displaying Prescriptions.

      6. Press 6 for Deleting a Prescription.

      7. Press 7 for Updating a Prescription.

      8. Press 8 for Inserting Patient Records in the DataBase.

      9. Press 9 for Displaying Patient Records.

      10. Press 10 for Searching for Patients.

      11. Press 11 for Deleting Patient Records.

      12. Exit (Press 12)


              Your Choice:
```

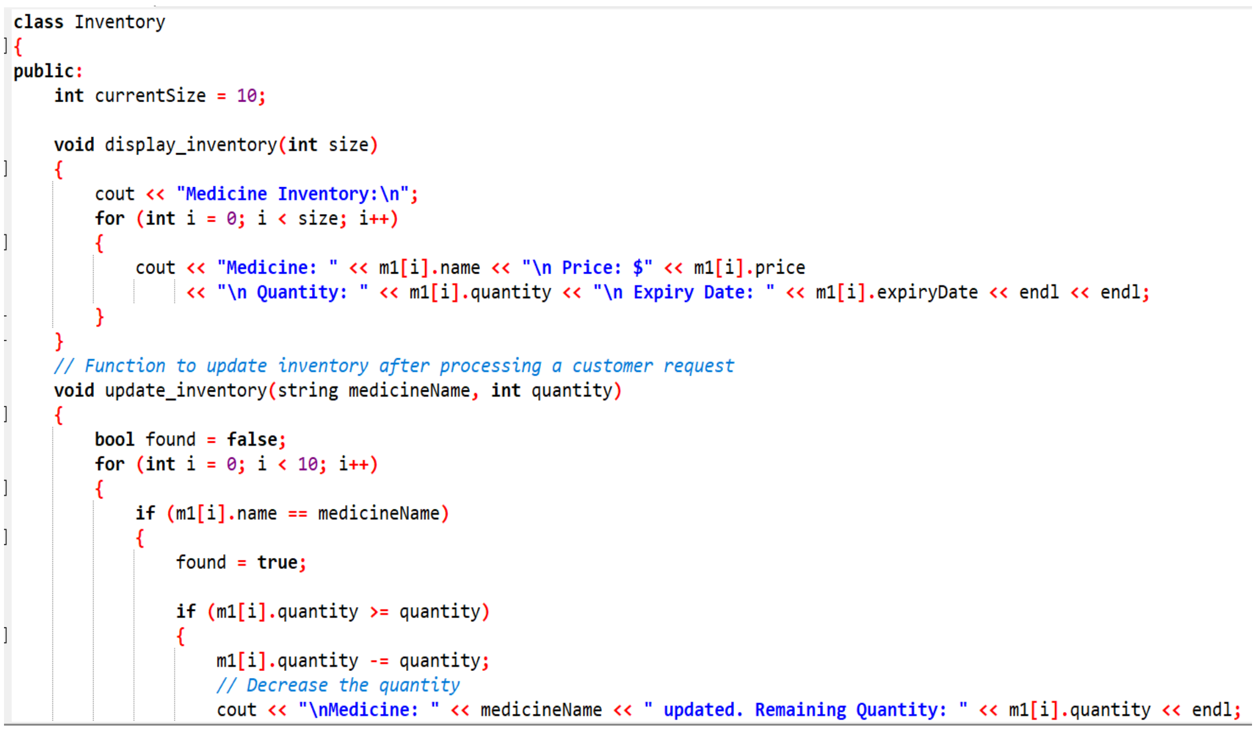
## ***Classes and Their Functions***

### *1.* Inventory Class

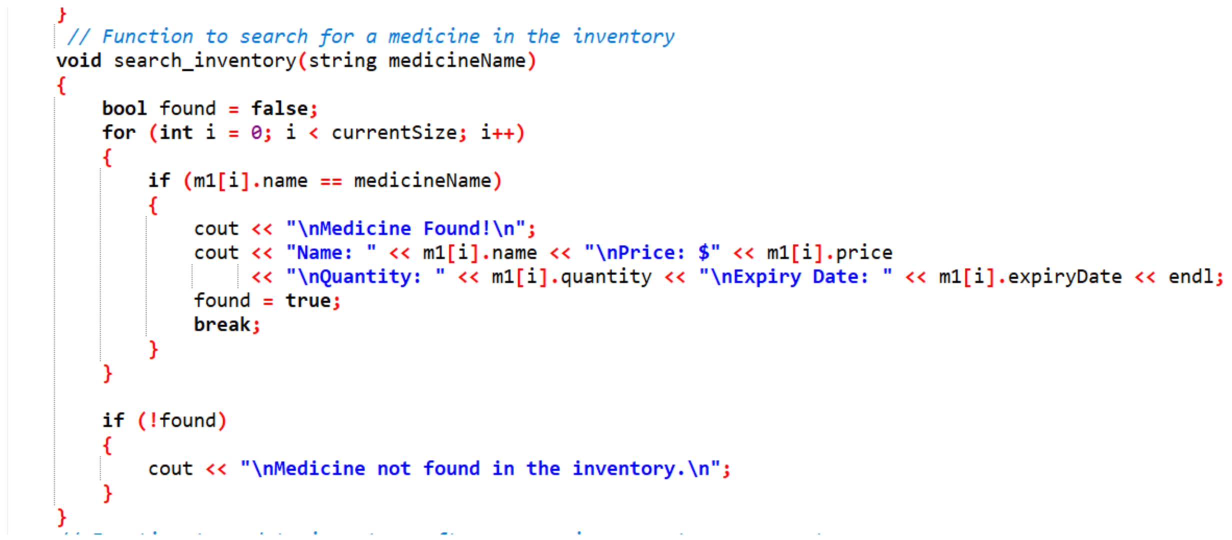
Manages the inventory of medicines. It provides functionalities for **displaying the inventory**, **updating stock**, **restocking medicines**, and **adding new medicines**.

### Code implementation

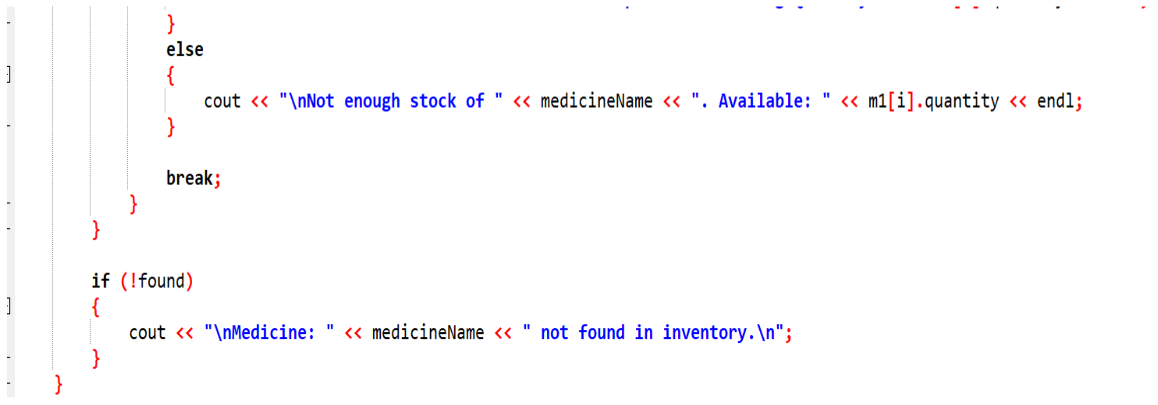
#### ➢ Displaying Inventory



```
class Inventory
{
public:
    int currentSize = 10;

    void display_inventory(int size)
    {
        cout << "Medicine Inventory:\n";
        for (int i = 0; i < size; i++)
        {
            cout << "Medicine: " << m1[i].name << "\n Price: $" << m1[i].price
                 << "\n Quantity: " << m1[i].quantity << "\n Expiry Date: " << m1[i].expiryDate << endl << endl;
        }
    }
    // Function to update inventory after processing a customer request
    void update_inventory(string medicineName, int quantity)
    {
        bool found = false;
        for (int i = 0; i < 10; i++)
        {
            if (m1[i].name == medicineName)
            {
                found = true;

                if (m1[i].quantity >= quantity)
                {
                    m1[i].quantity -= quantity;
                    // Decrease the quantity
                    cout << "\nMedicine: " << medicineName << " updated. Remaining Quantity: " << m1[i].quantity << endl;
```

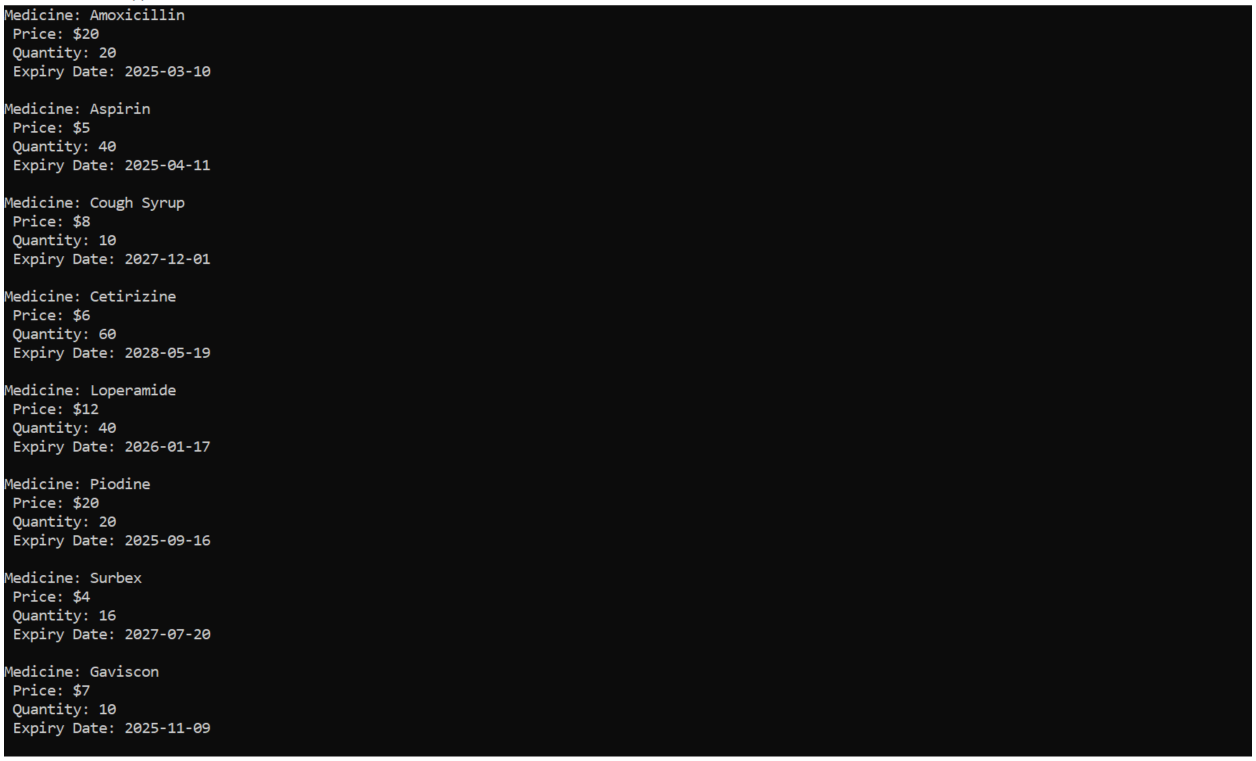


```
}
// Function to search for a medicine in the inventory
void search_inventory(string medicineName)
{
    bool found = false;
    for (int i = 0; i < currentSize; i++)
    {
        if (m1[i].name == medicineName)
        {
            cout << "\nMedicine Found!\n";
            cout << "Name: " << m1[i].name << "\nPrice: $" << m1[i].price
                 << "\nQuantity: " << m1[i].quantity << "\nExpiry Date: " << m1[i].expiryDate << endl;
            found = true;
            break;
        }
    }

    if (!found)
    {
        cout << "\nMedicine not found in the inventory.\n";
    }
}
```



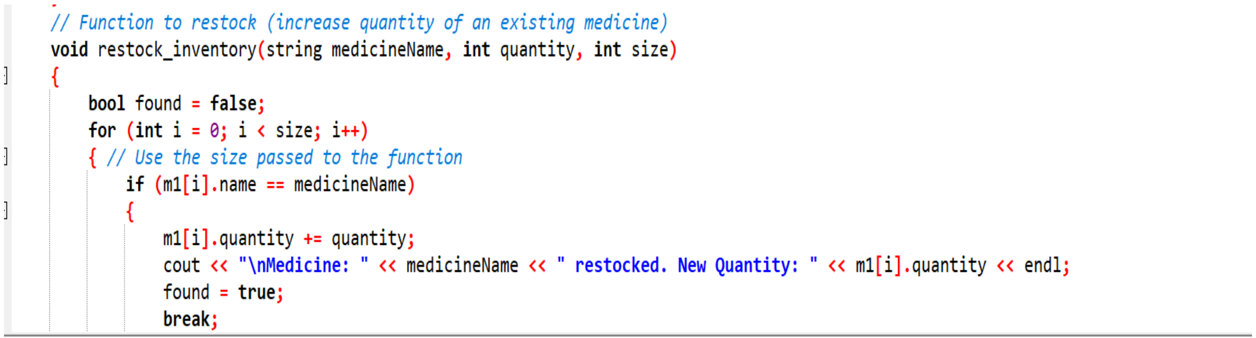
```
            }
            else
            {
                cout << "\nNot enough stock of " << medicineName << ". Available: " << m1[i].quantity << endl;
            }

            break;
        }
    }

    if (!found)
    {
        cout << "\nMedicine: " << medicineName << " not found in inventory.\n";
    }
}
```

## **Output:**

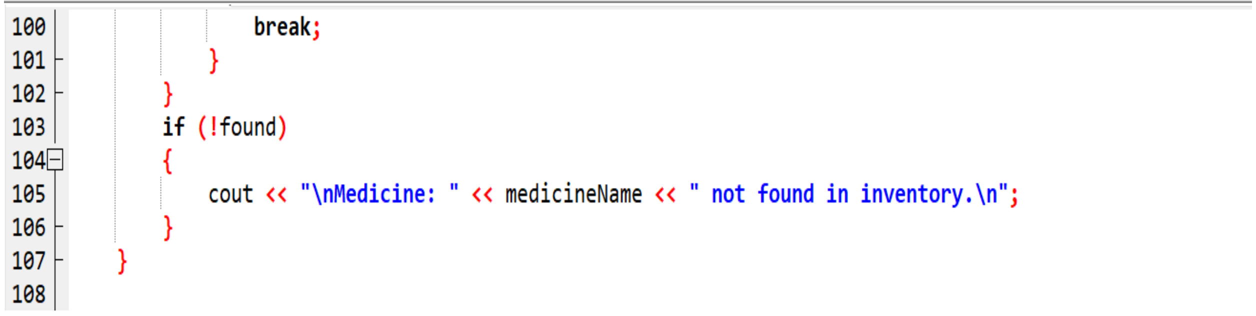


```
Medicine: Amoxicillin
 Price: $20
 Quantity: 20
 Expiry Date: 2025-03-10

Medicine: Aspirin
 Price: $5
 Quantity: 40
 Expiry Date: 2025-04-11

Medicine: Cough Syrup
 Price: $8
 Quantity: 10
 Expiry Date: 2027-12-01

Medicine: Cetirizine
 Price: $6
 Quantity: 60
 Expiry Date: 2028-05-19

Medicine: Loperamide
 Price: $12
 Quantity: 40
 Expiry Date: 2026-01-17

Medicine: Piodine
 Price: $20
 Quantity: 20
 Expiry Date: 2025-09-16

Medicine: Surbex
 Price: $4
 Quantity: 16
 Expiry Date: 2027-07-20

Medicine: Gaviscon
 Price: $7
 Quantity: 10
 Expiry Date: 2025-11-09
```

## ➢ Restock Inventory:

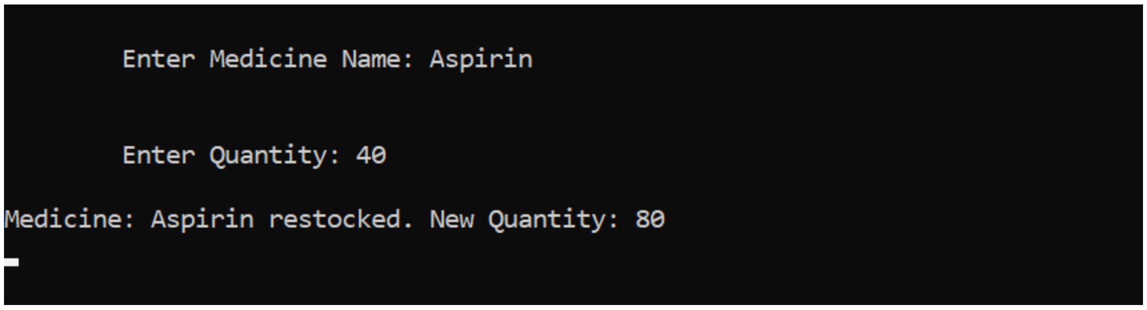


```cpp
// Function to restock (increase quantity of an existing medicine)
void restock_inventory(string medicineName, int quantity, int size)
{
    bool found = false;
    for (int i = 0; i < size; i++)
    { // Use the size passed to the function
        if (m1[i].name == medicineName)
        {
            m1[i].quantity += quantity;
            cout << "\nMedicine: " << medicineName << " restocked. New Quantity: " << m1[i].quantity << endl;
            found = true;
            break;
```



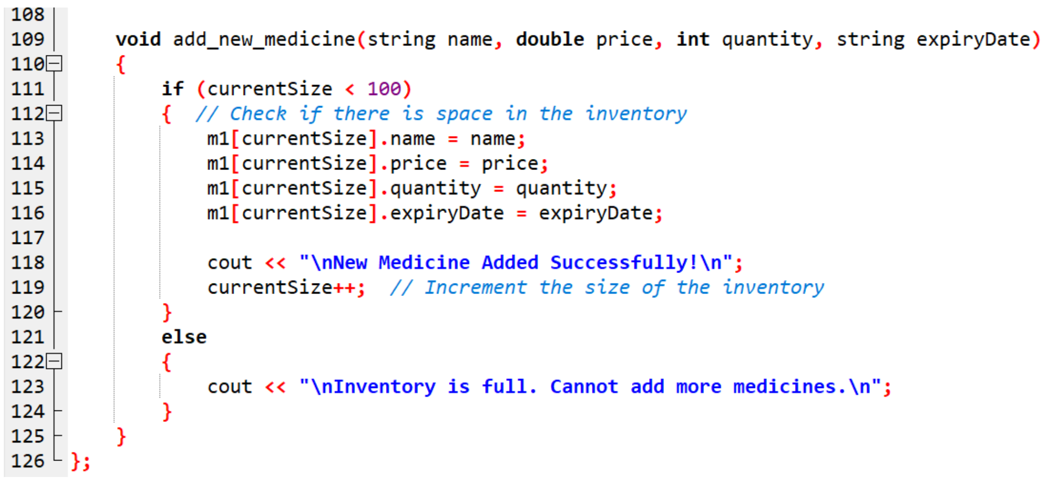
```cpp
            break;
        }
    }
    if (!found)
    {
        cout << "\nMedicine: " << medicineName << " not found in inventory.\n";
    }
}
```

## Output:



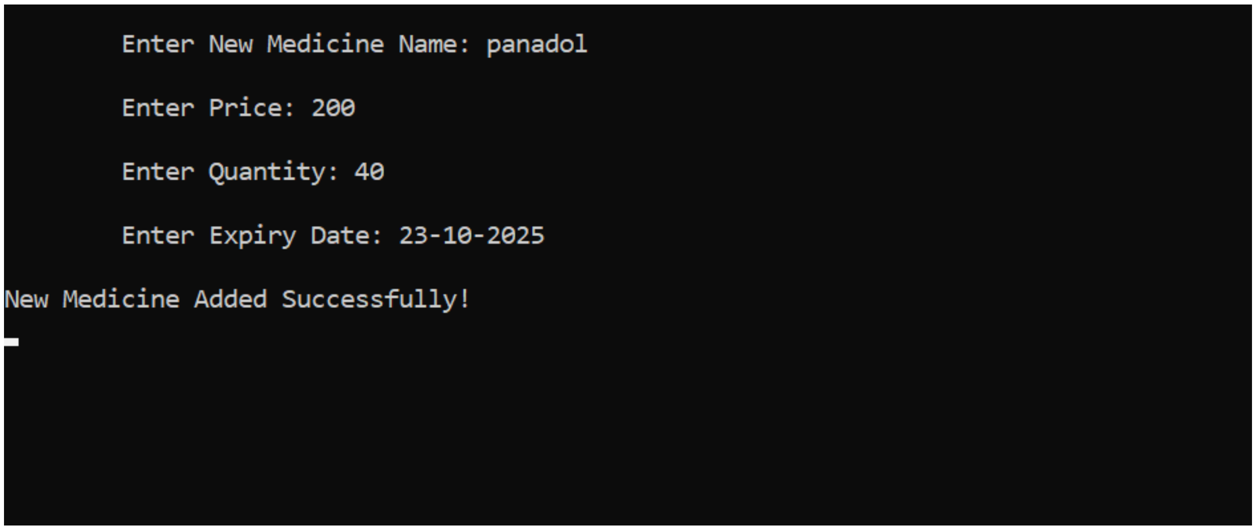
```
        Enter Medicine Name: Aspirin


        Enter Quantity: 40

Medicine: Aspirin restocked. New Quantity: 80
_
```

## ➢ Adding new Medicine:



```cpp

    void add_new_medicine(string name, double price, int quantity, string expiryDate)
    {
        if (currentSize < 100)
        {  // Check if there is space in the inventory
            m1[currentSize].name = name;
            m1[currentSize].price = price;
            m1[currentSize].quantity = quantity;
            m1[currentSize].expiryDate = expiryDate;

            cout << "\nNew Medicine Added Successfully!\n";
            currentSize++;  // Increment the size of the inventory
        }
        else
        {
            cout << "\nInventory is full. Cannot add more medicines.\n";
        }
    }
};
```

## Output:



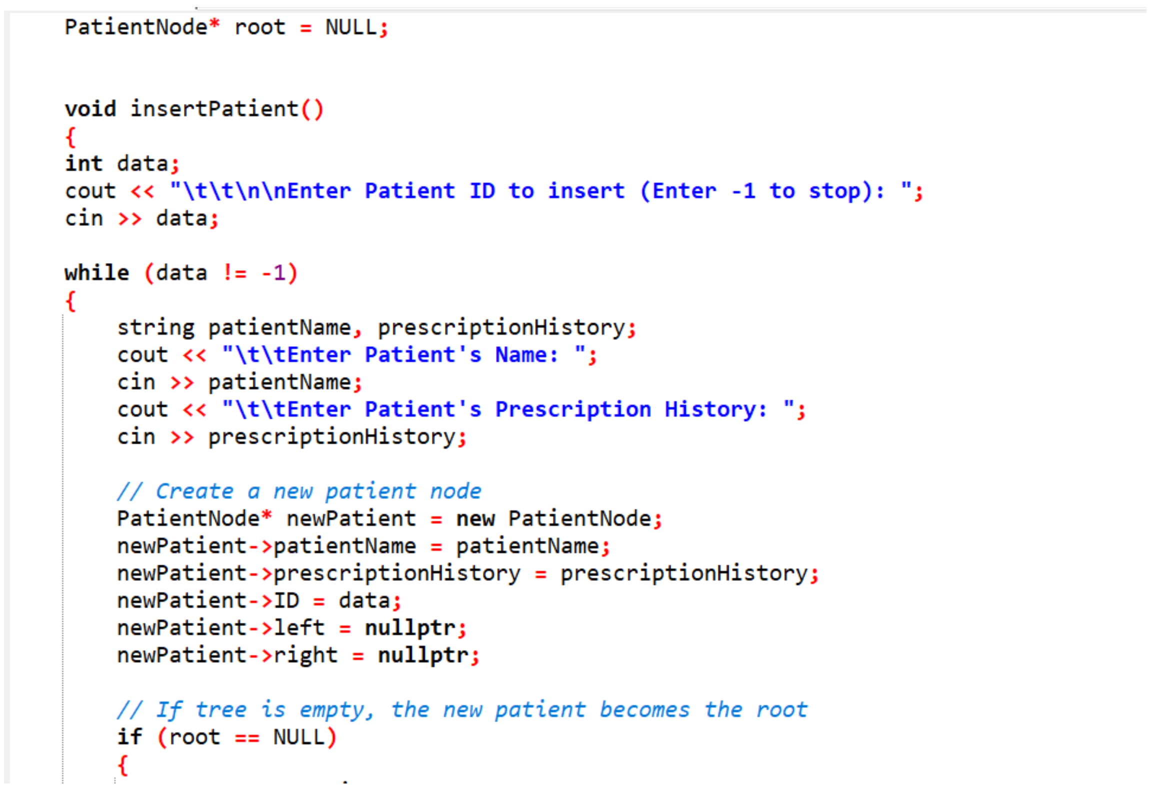
```
        Enter New Medicine Name: panadol

        Enter Price: 200

        Enter Quantity: 40

        Enter Expiry Date: 23-10-2025

New Medicine Added Successfully!
_
```

## 2. *PatientRecord Class*

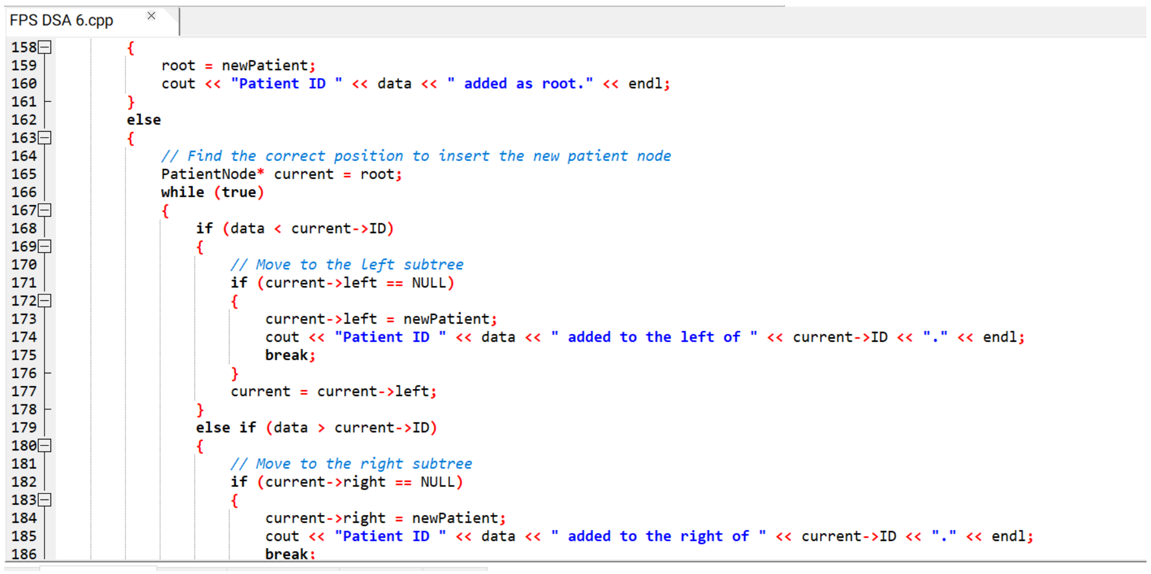
Manages patient records stored in a **binary search tree**.

## Code implementation

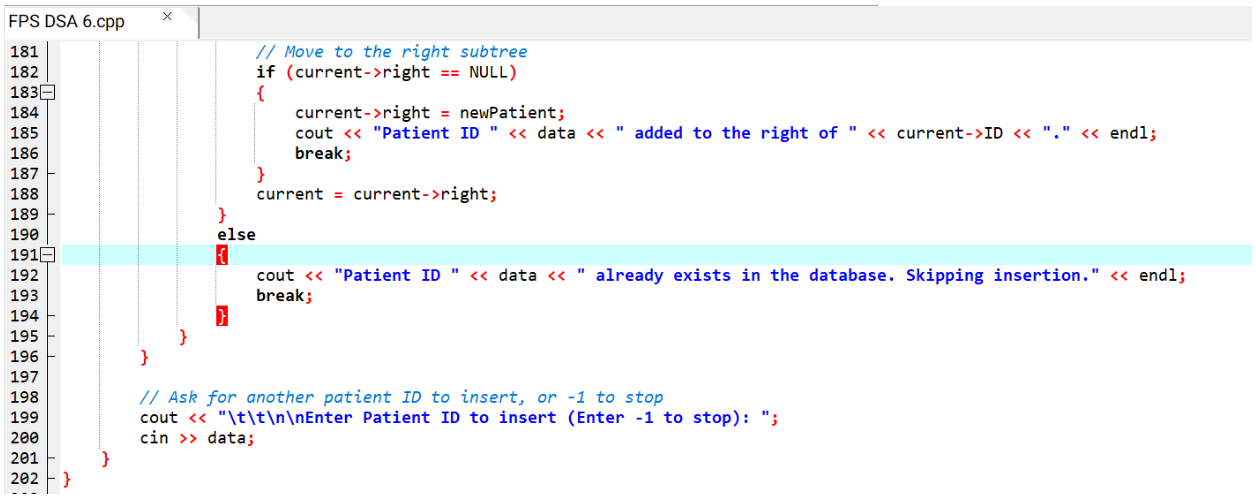
### ➢ InsertPatient:



```cpp
PatientNode* root = NULL;


void insertPatient()
{
int data;
cout << "\t\t\n\nEnter Patient ID to insert (Enter -1 to stop): ";
cin >> data;

while (data != -1)
{
    string patientName, prescriptionHistory;
    cout << "\t\tEnter Patient's Name: ";
    cin >> patientName;
    cout << "\t\tEnter Patient's Prescription History: ";
    cin >> prescriptionHistory;

    // Create a new patient node
    PatientNode* newPatient = new PatientNode;
    newPatient->patientName = patientName;
    newPatient->prescriptionHistory = prescriptionHistory;
    newPatient->ID = data;
    newPatient->left = nullptr;
    newPatient->right = nullptr;

    // If tree is empty, the new patient becomes the root
    if (root == NULL)
    {
```

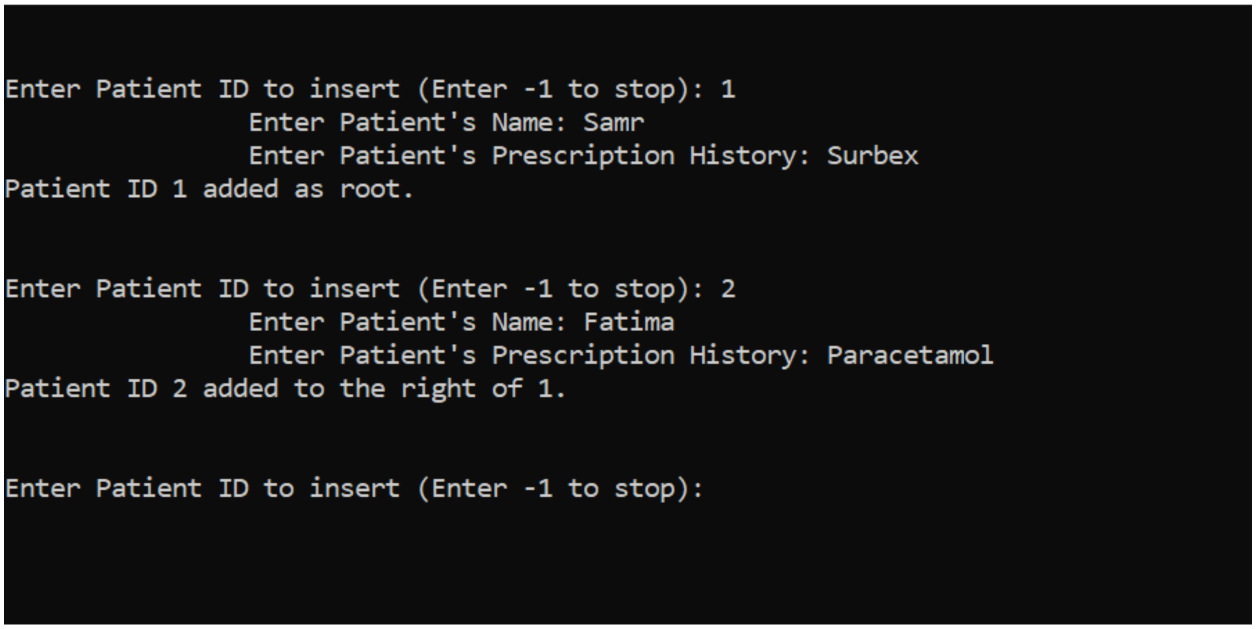


```cpp
    {
        root = newPatient;
        cout << "Patient ID " << data << " added as root." << endl;
    }
    else
    {
        // Find the correct position to insert the new patient node
        PatientNode* current = root;
        while (true)
        {
            if (data < current->ID)
            {
                // Move to the left subtree
                if (current->left == NULL)
                {
                    current->left = newPatient;
                    cout << "Patient ID " << data << " added to the left of " << current->ID << "." << endl;
                    break;
                }
                current = current->left;
            }
            else if (data > current->ID)
            {
                // Move to the right subtree
                if (current->right == NULL)
                {
                    current->right = newPatient;
                    cout << "Patient ID " << data << " added to the right of " << current->ID << "." << endl;
                    break;
```


FPS DSA 6.cpp

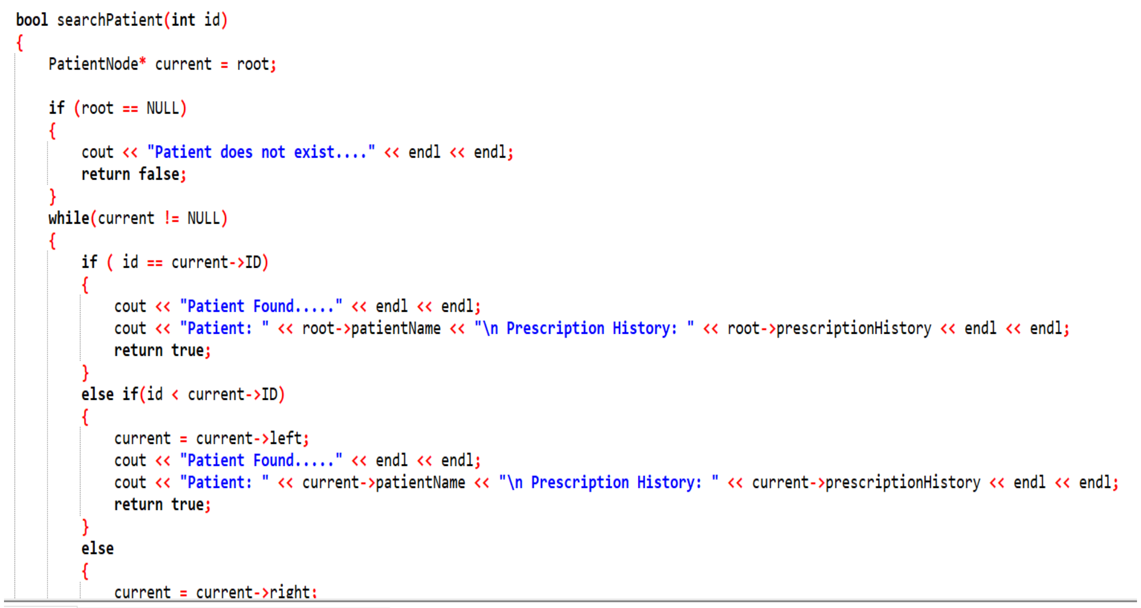
```cpp
                    // Move to the right subtree
                    if (current->right == NULL)
                    {
                        current->right = newPatient;
                        cout << "Patient ID " << data << " added to the right of " << current->ID << "." << endl;
                        break;
                    }
                    current = current->right;
                }
                else
                {
                    cout << "Patient ID " << data << " already exists in the database. Skipping insertion." << endl;
                    break;
                }
            }
        }

        // Ask for another patient ID to insert, or -1 to stop
        cout << "\t\t\n\nEnter Patient ID to insert (Enter -1 to stop): ";
        cin >> data;
    }
}
```

## <u>Output</u>

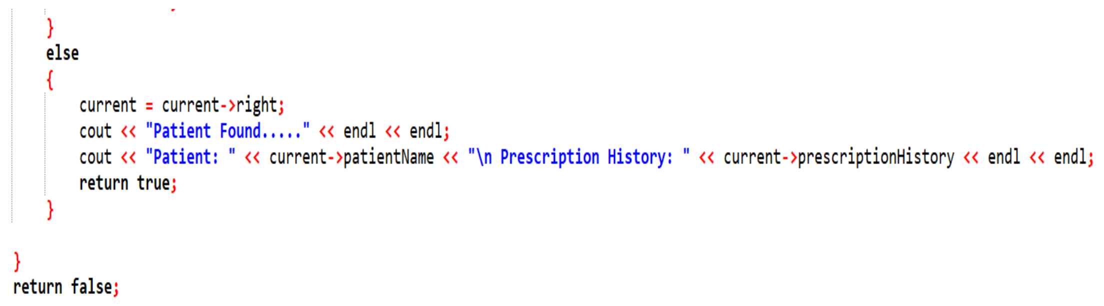


```
Enter Patient ID to insert (Enter -1 to stop): 1
                Enter Patient's Name: Samr
                Enter Patient's Prescription History: Surbex
Patient ID 1 added as root.


Enter Patient ID to insert (Enter -1 to stop): 2
                Enter Patient's Name: Fatima
                Enter Patient's Prescription History: Paracetamol
Patient ID 2 added to the right of 1.


Enter Patient ID to insert (Enter -1 to stop):
```

## ➢ **SearchPatient:**

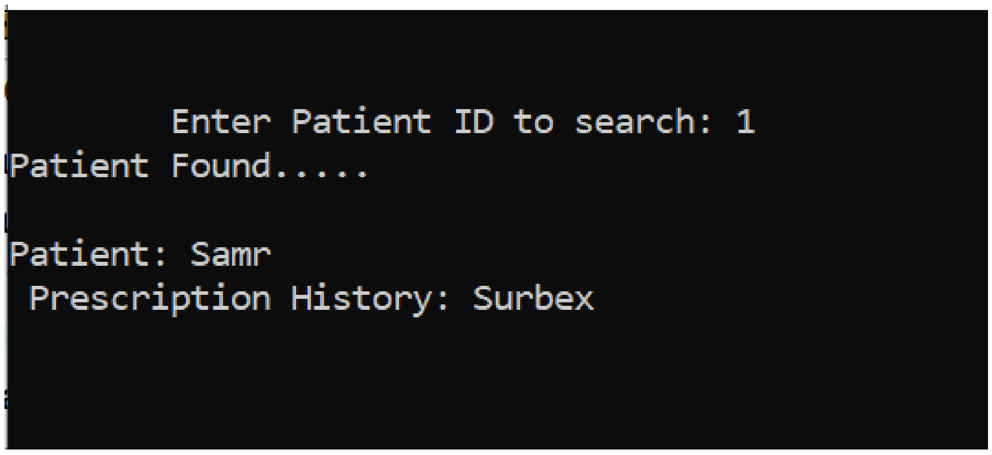


```
bool searchPatient(int id)
{
    PatientNode* current = root;

    if (root == NULL)
    {
        cout << "Patient does not exist...." << endl << endl;
        return false;
    }
    while(current != NULL)
    {
        if ( id == current->ID)
        {
            cout << "Patient Found....." << endl << endl;
            cout << "Patient: " << root->patientName << "\n Prescription History: " << root->prescriptionHistory << endl << endl;
            return true;
        }
        else if(id < current->ID)
        {
            current = current->left;
            cout << "Patient Found....." << endl << endl;
            cout << "Patient: " << current->patientName << "\n Prescription History: " << current->prescriptionHistory << endl << endl;
            return true;
        }
        else
        {
            current = current->right;
```



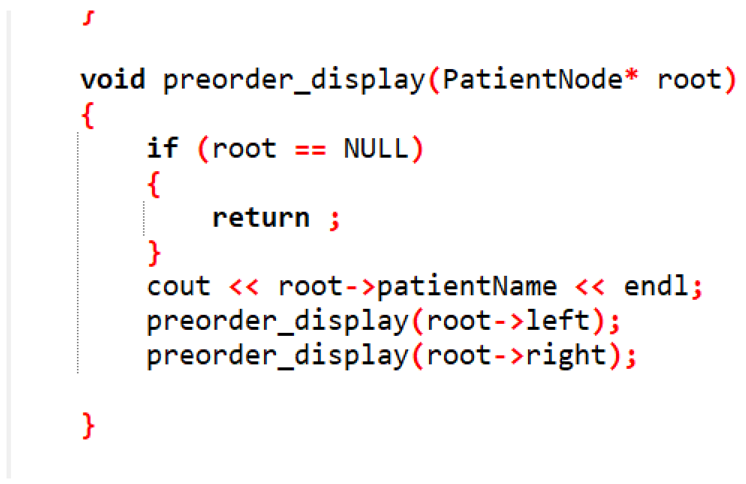
```
        }
        else
        {
            current = current->right;
            cout << "Patient Found....." << endl << endl;
            cout << "Patient: " << current->patientName << "\n Prescription History: " << current->prescriptionHistory << endl << endl;
            return true;
        }

}
return false;
```

## **Output:**



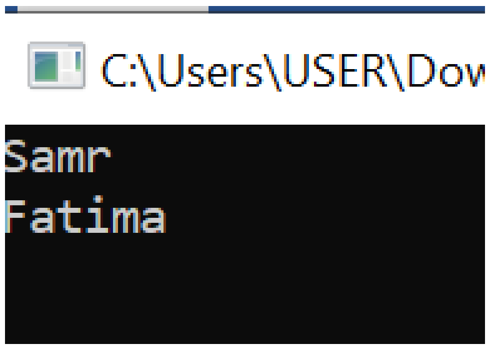
```
        Enter Patient ID to search: 1
Patient Found.....

Patient: Samr
 Prescription History: Surbex
```

## ➢ Display in a Preorder Manner:



```
void preorder_display(PatientNode* root)
{
    if (root == NULL)
    {
        return ;
    }
    cout << root->patientName << endl;
    preorder_display(root->left);
    preorder_display(root->right);

}
```

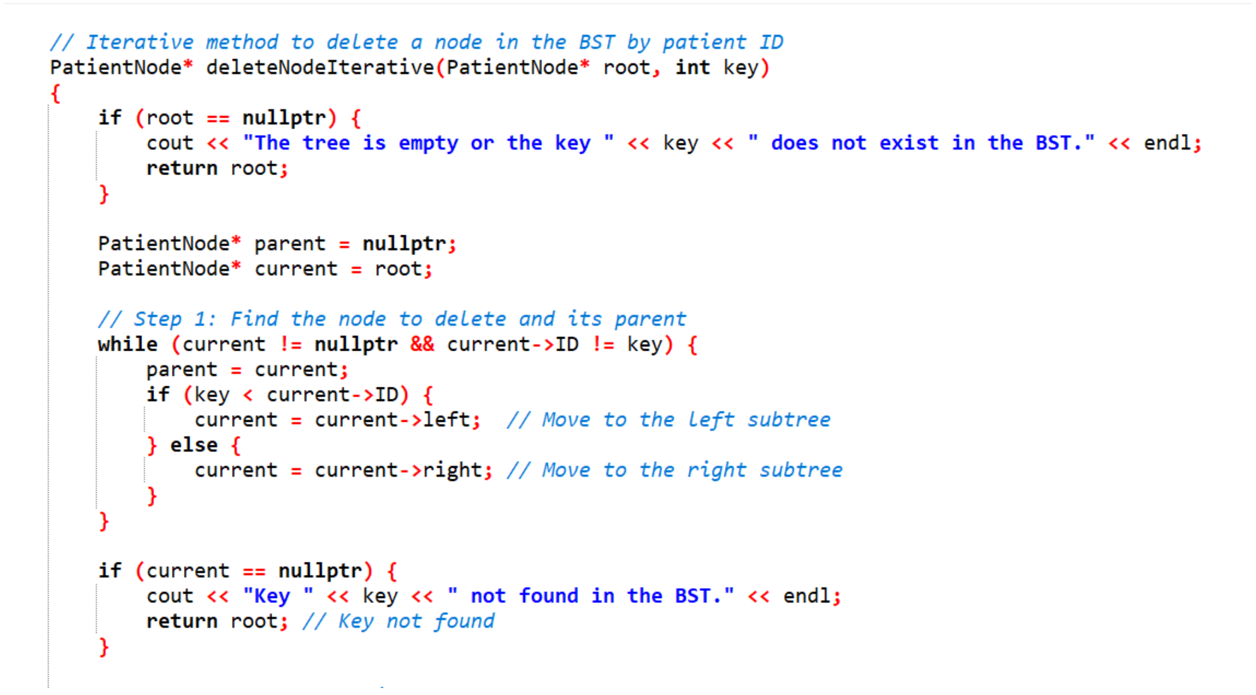
**Output:**



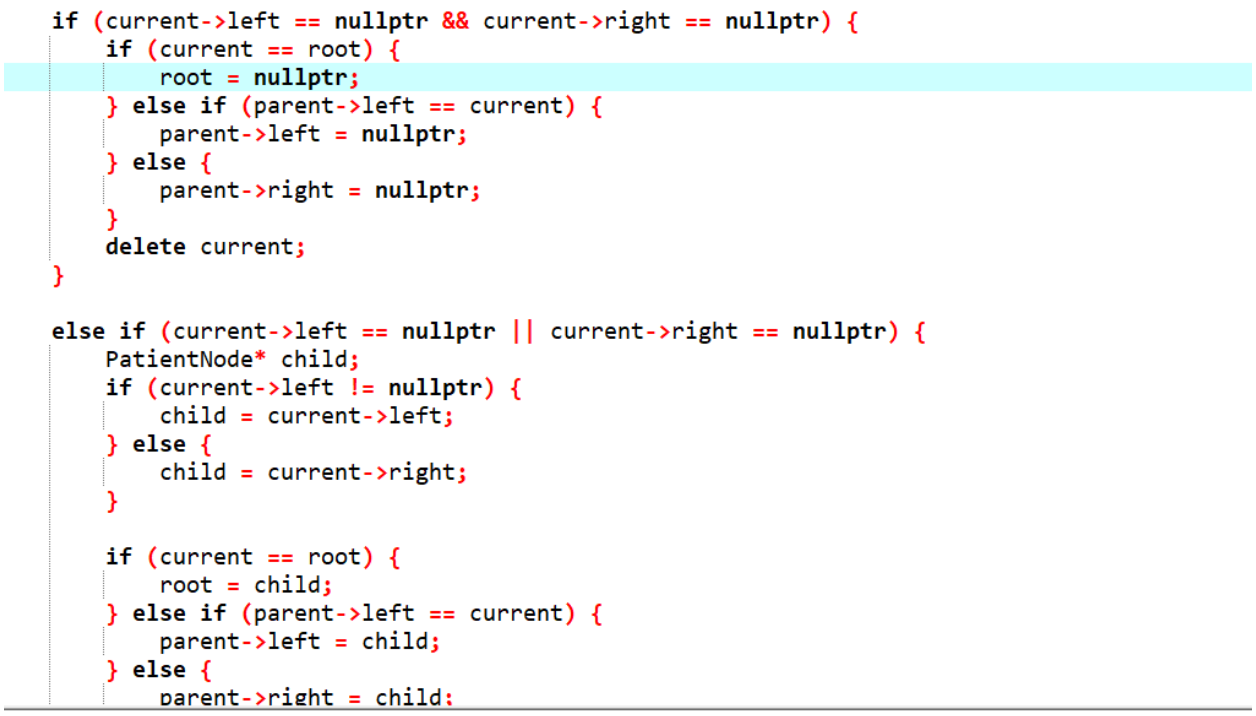
```
C:\Users\USER\Dow
```

```
Samr
Fatima
```

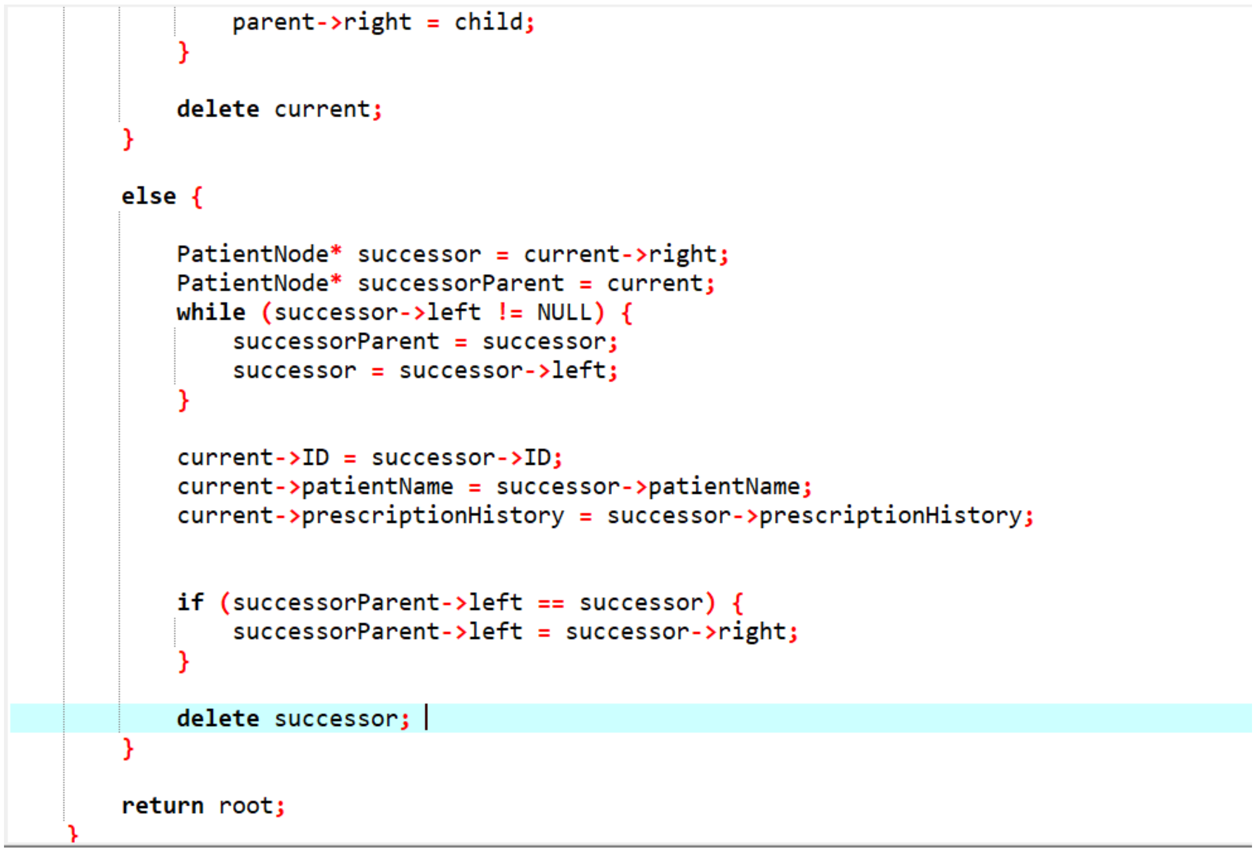
## ➢ DeleteRecord:



```cpp
// Iterative method to delete a node in the BST by patient ID
PatientNode* deleteNodeIterative(PatientNode* root, int key)
{
    if (root == nullptr) {
        cout << "The tree is empty or the key " << key << " does not exist in the BST." << endl;
        return root;
    }

    PatientNode* parent = nullptr;
    PatientNode* current = root;

    // Step 1: Find the node to delete and its parent
    while (current != nullptr && current->ID != key) {
        parent = current;
        if (key < current->ID) {
            current = current->left;  // Move to the left subtree
        } else {
            current = current->right; // Move to the right subtree
        }
    }

    if (current == nullptr) {
        cout << "Key " << key << " not found in the BST." << endl;
        return root; // Key not found
    }
```

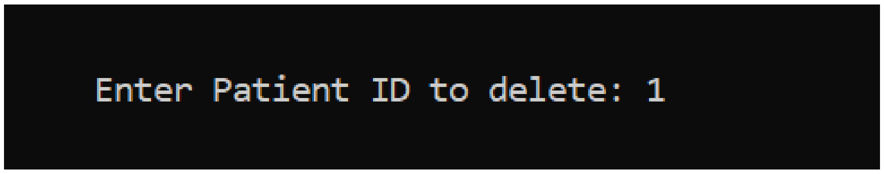


```cpp
if (current->left == nullptr && current->right == nullptr) {
    if (current == root) {
        root = nullptr;
    } else if (parent->left == current) {
        parent->left = nullptr;
    } else {
        parent->right = nullptr;
    }
    delete current;
}

else if (current->left == nullptr || current->right == nullptr) {
    PatientNode* child;
    if (current->left != nullptr) {
        child = current->left;
    } else {
        child = current->right;
    }

    if (current == root) {
        root = child;
    } else if (parent->left == current) {
        parent->left = child;
    } else {
        parent->right = child;
```

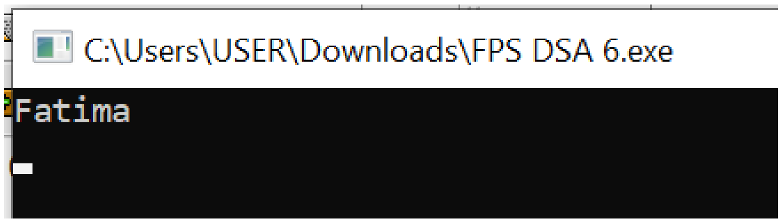


```
            parent->right = child;
        }

        delete current;
    }

    else {

        PatientNode* successor = current->right;
        PatientNode* successorParent = current;
        while (successor->left != NULL) {
            successorParent = successor;
            successor = successor->left;
        }

        current->ID = successor->ID;
        current->patientName = successor->patientName;
        current->prescriptionHistory = successor->prescriptionHistory;


        if (successorParent->left == successor) {
            successorParent->left = successor->right;
        }

        delete successor;
    }

    return root;
}
```

**Output:**



```
Enter Patient ID to delete: 1
```



C:\Users\USER\Downloads\FPS DSA 6.exe

```
Fatima
```

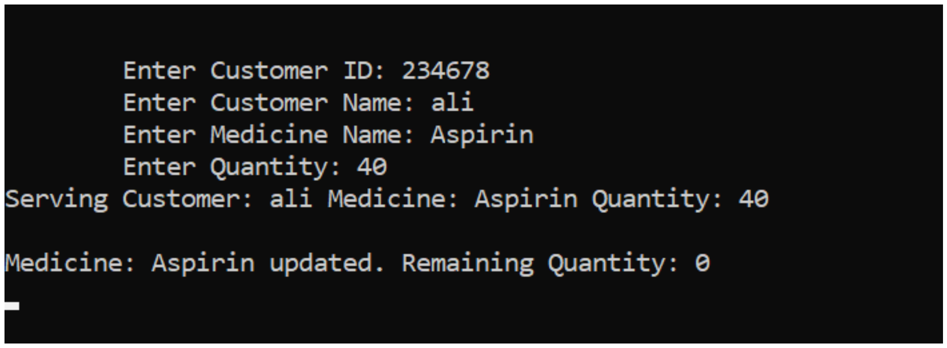
## 3. ***Prescriptions Class***

Manages prescriptions made by customers using a linked list.

## Code implementation

### ➢ AddNewPrescriptions

```cpp
class Prescriptions
{
private:
    Prescription* head;

public:
    Prescriptions() : head(NULL) {}

    void addNewPrescriptions(string customerName, string medicineName, int quantity, int id)
    {
        Prescription* temp = new Prescription;
        temp->customerName = customerName;
        temp->id = id;
        temp->medicineName = medicineName;
        temp->quantity = quantity;
        temp->next = head;
        head = temp;
    }
```

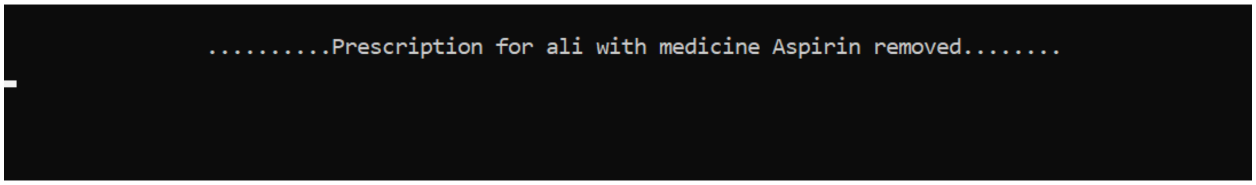


```
        Enter Customer ID: 234678
        Enter Customer Name: ali
        Enter Medicine Name: Aspirin
        Enter Quantity: 40
Serving Customer: ali Medicine: Aspirin Quantity: 40

Medicine: Aspirin updated. Remaining Quantity: 0

```

### ➢ Delete_Prescription

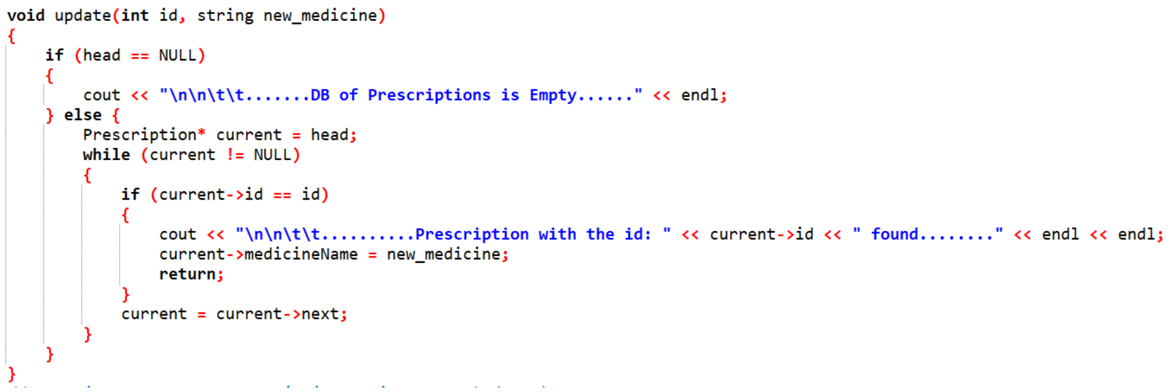
```cpp
void dequeue_prescription()
{
    if (head == NULL)
    {
        cout << "\n\n\t\t.......DB of Prescriptions is Empty......" << endl;
        return; // If there are no prescriptions, return
    }

    Prescription* temp = head;
    head = head->next;

    cout << "\n\n\t\t.........Prescription for " << temp->customerName
        << " with medicine " << temp->medicineName << " removed........" << endl;


    delete temp;
}
```

## Output:



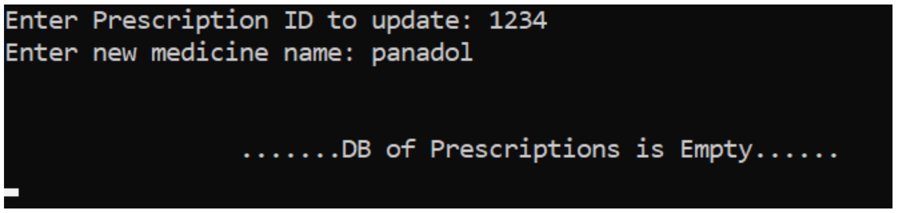
```
..........Prescription for ali with medicine Aspirin removed........
```

## ➢ Update prescription



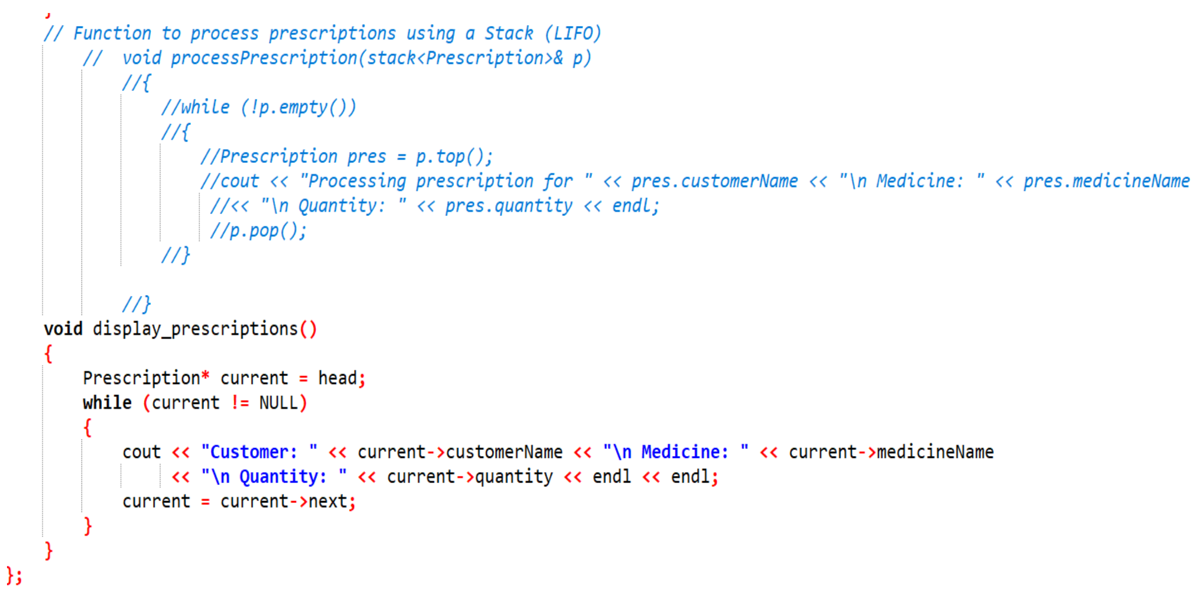
```
void update(int id, string new_medicine)
{
    if (head == NULL)
    {
        cout << "\n\n\t\t.......DB of Prescriptions is Empty......" << endl;
    } else {
        Prescription* current = head;
        while (current != NULL)
        {
            if (current->id == id)
            {
                cout << "\n\n\t\t..........Prescription with the id: " << current->id << " found........" << endl << endl;
                current->medicineName = new_medicine;
                return;
            }
            current = current->next;
        }
    }
}
```

## Output



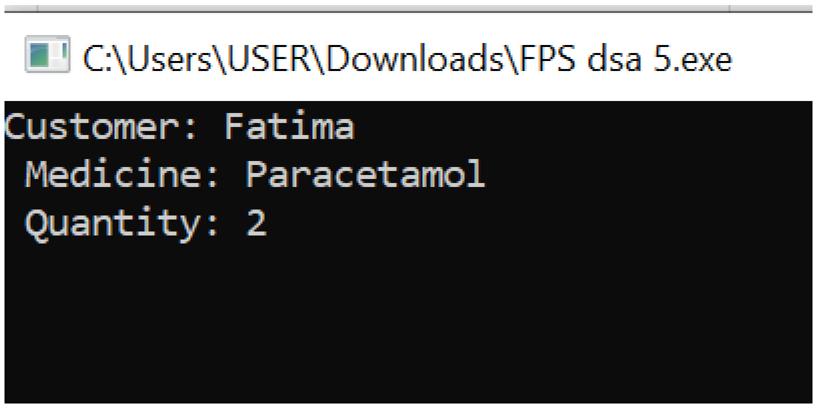
```
Enter Prescription ID to update: 1234
Enter new medicine name: panadol


                .......DB of Prescriptions is Empty......
```

- ➢ **Commented processPrescription function**
- ➢ **Display_prescription**



```
,
// Function to process prescriptions using a Stack (LIFO)
    //  void processPrescription(stack<Prescription>& p)
        //{
            //while (!p.empty())
            //{
                //Prescription pres = p.top();
                //cout << "Processing prescription for " << pres.customerName << "\n Medicine: " << pres.medicineName
                 //<< "\n Quantity: " << pres.quantity << endl;
                 //p.pop();
            //}

        //}
void display_prescriptions()
{
    Prescription* current = head;
    while (current != NULL)
    {
        cout << "Customer: " << current->customerName << "\n Medicine: " << current->medicineName
             << "\n Quantity: " << current->quantity << endl << endl;
        current = current->next;
    }
}
};
```

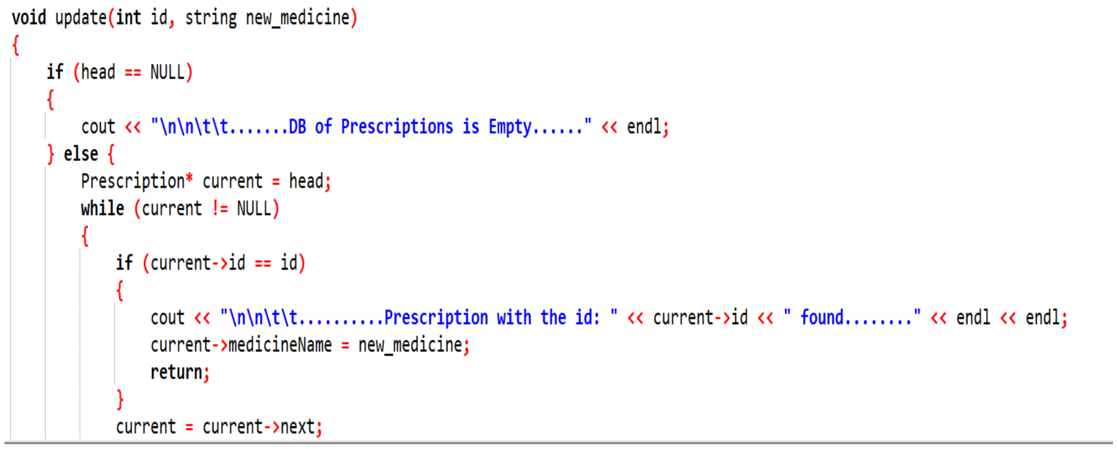
<u>**Output:**</u>



C:\Users\USER\Downloads\FPS dsa 5.exe

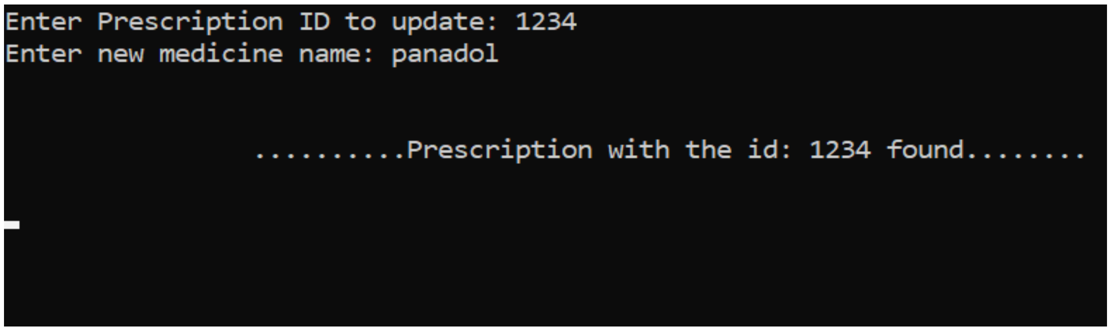
```
Customer: Fatima
 Medicine: Paracetamol
 Quantity: 2
```

## ➢ Update:



```
void update(int id, string new_medicine)
{
    if (head == NULL)
    {
        cout << "\n\n\t\t.......DB of Prescriptions is Empty......" << endl;
    } else {
        Prescription* current = head;
        while (current != NULL)
        {
            if (current->id == id)
            {
                cout << "\n\n\t\t..........Prescription with the id: " << current->id << " found........" << endl << endl;
                current->medicineName = new_medicine;
                return;
            }
            current = current->next;
```

## Output:

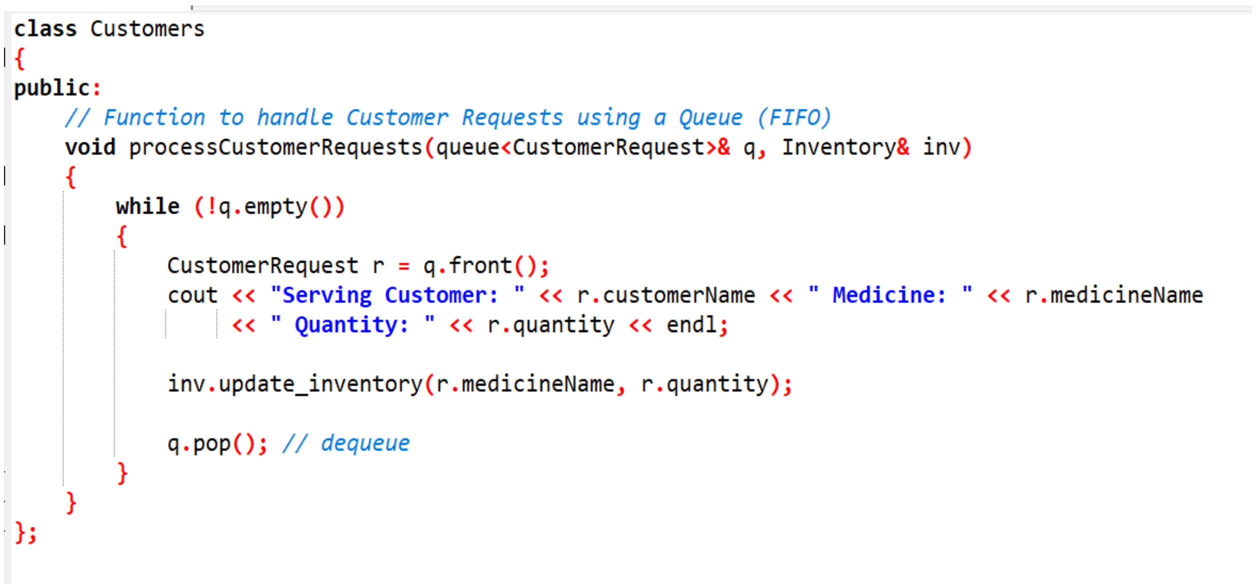


```
Enter Prescription ID to update: 1234
Enter new medicine name: panadol


                ..........Prescription with the id: 1234 found........

_
```

## ***4. Customers Class***

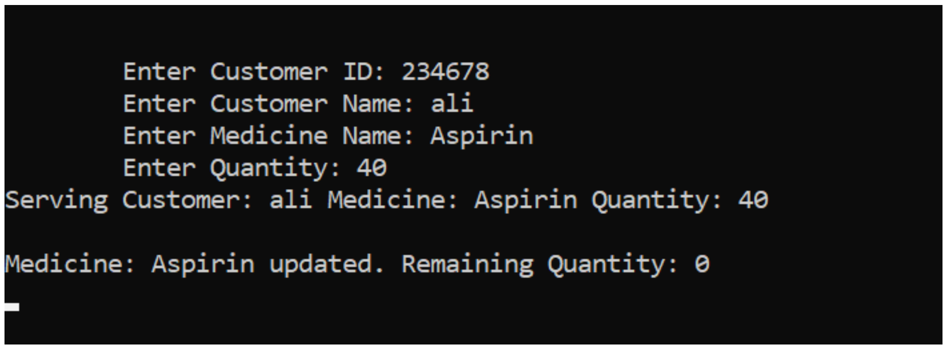
Handles customer requests using a queue (FIFO), ensuring that customers are processed in the order they arrive.

## Code implementation



```
class Customers
{
public:
    // Function to handle Customer Requests using a Queue (FIFO)
    void processCustomerRequests(queue<CustomerRequest>& q, Inventory& inv)
    {
        while (!q.empty())
        {
            CustomerRequest r = q.front();
            cout << "Serving Customer: " << r.customerName << " Medicine: " << r.medicineName
                << " Quantity: " << r.quantity << endl;

            inv.update_inventory(r.medicineName, r.quantity);

            q.pop(); // dequeue
        }
    }
};
```

## Output:



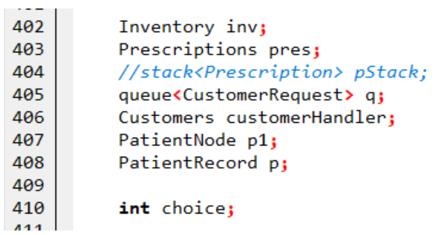
```
         Enter Customer ID: 234678
         Enter Customer Name: ali
         Enter Medicine Name: Aspirin
         Enter Quantity: 40
Serving Customer: ali Medicine: Aspirin Quantity: 40

Medicine: Aspirin updated. Remaining Quantity: 0
```

## *Main Function (main())*

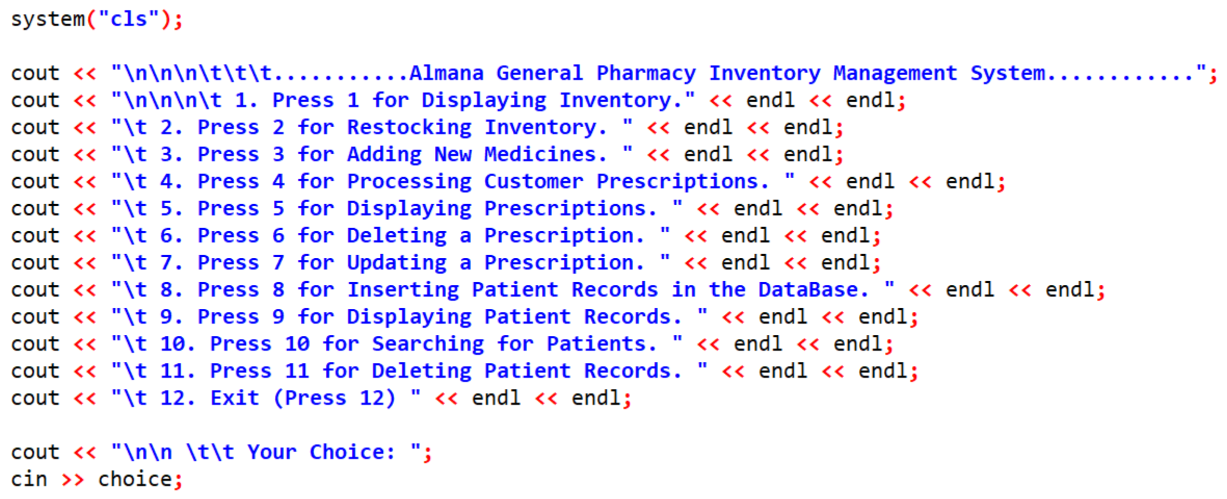
The main function runs a menu-driven interface that allows the user to interact with the pharmacy management system. The user can select from the following options:

## <u>Code implementation</u>

### *Menu and objects:*



```
    Inventory inv;
    Prescriptions pres;
    //stack<Prescription> pStack;
    queue<CustomerRequest> q;
    Customers customerHandler;
    PatientNode p1;
    PatientRecord p;

    int choice;
```



```
system("cls");

cout << "\n\n\n\t\t\t..........Almana General Pharmacy Inventory Management System...........";
cout << "\n\n\n\t 1. Press 1 for Displaying Inventory." << endl << endl;
cout << "\t 2. Press 2 for Restocking Inventory. " << endl << endl;
cout << "\t 3. Press 3 for Adding New Medicines. " << endl << endl;
cout << "\t 4. Press 4 for Processing Customer Prescriptions. " << endl << endl;
cout << "\t 5. Press 5 for Displaying Prescriptions. " << endl << endl;
cout << "\t 6. Press 6 for Deleting a Prescription. " << endl << endl;
cout << "\t 7. Press 7 for Updating a Prescription. " << endl << endl;
cout << "\t 8. Press 8 for Inserting Patient Records in the DataBase. " << endl << endl;
cout << "\t 9. Press 9 for Displaying Patient Records. " << endl << endl;
cout << "\t 10. Press 10 for Searching for Patients. " << endl << endl;
cout << "\t 11. Press 11 for Deleting Patient Records. " << endl << endl;
cout << "\t 12. Exit (Press 12) " << endl << endl;

cout << "\n\n \t\t Your Choice: ";
cin >> choice;
```

1. **Display Inventory**: Displays the current inventory of medicines.
2. **Restock Inventory**: Restocks a specific medicine by increasing its quantity.
3. **Add New Medicines**: Adds a new medicine to the inventory.
4. **Process Customer Prescriptions**: Takes customer details and processes their prescriptions, adding them to the system.
5. **Display Prescriptions**: Displays all prescriptions in the system.
6. **Delete a Prescription**: Deletes a prescription by its ID.
7. **Update a Prescription**: Updates the medicine in a prescription by its ID.
8. **Insert Patient Records**: Inserts new patient records into the binary search tree.
9. **Display Patient Records**: Displays patient records in preorder traversal.
10. **Search for Patients**: Searches for a patient by their unique ID.
11. **Delete Patient Records**: Deletes a patient record by its ID.
12. **Exit**: Exits the system.

The program uses `system("cls")` to clear the screen after each operation, ensuring the interface remains clean.

## Key Observations

- **Dynamic Data Structures**: The system employs dynamic data structures such as linked lists for storing prescriptions and binary search trees for managing patient records. These structures provide efficient operations for insertion, deletion, and searching.
- **Inventory Management**: The inventory of medicines is managed using an array, and operations such as updating stock, restocking, and adding new medicines are carried out on this array.
- **Customer Request Queue**: The customer requests are managed in a FIFO queue, ensuring that customers are served in the order they arrive.
- **Menu-Driven Interface**: The menu-driven system makes it easy for users to interact with the program and perform various operations.
- **Error Handling**: The system includes conditional checks to ensure valid actions, such as checking if a medicine exists before updating the inventory.

## Conclusion

This Pharmacy Management System is a complete solution for managing a pharmacy's inventory, prescriptions, and patient records. By utilizing dynamic data structures like linked lists and binary search trees, the system offers efficient data management and retrieval. The user-friendly interface ensures that pharmacy staff can easily interact with the system and perform necessary tasks. With its robust error handling and flexible features, the system is a valuable tool for any pharmacy operation.This documentation provides an overview of the program's design, data structures, functions, and key features.