```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <stack>
#include <queue>
#include <climits>
using namespace std;

const int NO_PATH = -1; // Use -1 to indicate no path

// Incident structure for the first program
struct Inc
{
    int id;                 // Incident ID
    string patName;        // Patient's name
    string desc;           // Description of the incident
    int priority;          // Priority of the incident
};

// Patient structure for the second program
struct Pat
{
    int id;                // Patient ID
    string name;          // Patient name
    int priority;         // Priority (lower number = higher priority)
    string phone;         // Patient phone number
    string ailment;       // Patient ailment
    int aptTime;          // Time of appointment (24-hour format, e.g., 900 for 9:00 AM)
};

// Inventory Item structure for the third program
struct Item
{
    string name;          // Name of the item
    int quantity;         // Quantity of the item
    int priority;         // Priority (lower = more critical)
};

// Node structure to hold hospital details
struct N
{ // N: Node
    string n; // n: name
    vector<pair<string, int>> r; // r: resources
};

// Class for Union-Find data structure
class UF
{ // UF: UnionFind
public:
    vector<int> p, rk; // p: parent, rk: rank

    UF(int sz)
    { // sz: size
        p.resize(sz);
        rk.resize(sz, 0);
        for (int i = 0; i < sz; ++i)
        {
             p[i] = i;
        }
    }

    int f(int x)
    { // f: find
        if (p[x] != x)
        {
            p[x] = f(p[x]);
        }
        return p[x];
    }

    void u(int x, int y)
    { // u: unite
        int rx = f(x), ry = f(y);
        if (rx != ry)
        {
            if (rk[rx] > rk[ry])
            {
                p[ry] = rx;
            }
            else if (rk[rx] < rk[ry])
            {
                p[rx] = ry;
            }
            else
            {
                p[ry] = rx;
                rk[rx]++;
            }
        }
    }
};

// Function declarations for the first program
vector<int> rabinSearch(const string& text, const string& pattern);
void searchIncByDesc(vector<Inc>& incidents);
void inputIncidents(vector<Inc>& incidents);
int partitionIncidents(vector<Inc>& incidents, int low, int high);
void quickSortIncidents(vector<Inc>& incidents, int low, int high);
void displayAllIncidents(const vector<Inc>& incidents);
void displayIncByPriority(const vector<Inc>& incidents);
void editIncDetails(vector<Inc>& incidents);
void removeIncFromList(vector<Inc>& incidents);
void showIncidentMenu();
void handleIncidentUserInput(vector<Inc>& incidents); // Fixed function name


// Function declarations for the second program
struct PatComp
{
    bool operator()(Pat a, Pat b)
    {
        return a.priority > b.priority; // Min-Heap (higher priority first)
    }
};

void addUrgPat(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue);
void viewUrgPats(const priority_queue<Pat, vector<Pat>, PatComp>& urgQueue);
void addRegPat(queue<Pat>& regQueue);
void viewregPats(queue<Pat>& regQueue);
void serveNextUrgPat(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue);
void serveNextRegPat(queue<Pat>& regQueue);
void manageAvailableSlots(vector<int>& slots);
void sortRegPats(queue<Pat>& regQueue);
void updatePatData(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue);
void removePatAppointment(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue, queue<Pat>& regQueue);
void displayPatientMenu(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue, queue<Pat>& regQueue,
vector<int>& slots);


// Function declarations for the third program
int hashItem(const string& itemSearchName, int hashSize);
void addInvItem(vector<vector<Item>>& invHashTable, int hashSize);
void findInvItem(const vector<vector<Item>>& invHashTable, int hashSize);
void editInvItem(vector<vector<Item>>& invHashTable, int hashSize);
void removeInvItem(vector<vector<Item>>& invHashTable, int hashSize);
void showInvItems(const vector<vector<Item>>& invHashTable);
struct BSTNode
{
    Item item;
    BSTNode* left;
    BSTNode* right;
    BSTNode(Item it) : item(it), left(nullptr), right(nullptr) {}
};
BSTNode* addBSTNode(BSTNode* root, Item item);
void findBSTNode(BSTNode* root, const string& itemSearchName);
void listBST(BSTNode* root);
BSTNode* removeBSTNode(BSTNode* root, const string& itemName);
void clearBST(BSTNode* root);
void showInventoryMenu(const int hashSize, vector<vector<Item>>& invHashTable, BSTNode* bst);


// Function declarations for the fourth program
int vInt(string prompt);
bool vNode(int idx, int n);
void dijkstra(const vector<vector<int>> &g, int s, int n, vector<int> &d, vector<int> &p);
void printPath(const vector<int> &p, int s, int d, const vector<int>& dist);
void inputGraph(int &n, vector<vector<int>> &g, vector<N> &nodes);
void checkResources(int n, const vector<N> &nodes);
void modifyResources(vector<N> &nodes);
void addNewNode(int &n, vector<vector<int>> &g, vector<N> &nodes);
void findHospitalsWithResource(vector<N> &nodes, int n);
void displayAdjacencyMatrix(const vector<vector<int>> &g, int n);
void menufour(vector<vector<int>> &g,vector<N> &nodes, int n);


// Main menu function
void showMainMenu()
{
    cout << "\n--- Main Menu ---\n";
    cout << "1. Emergency Medical Incident Categoriztion\n";
    cout << "2. Patient Appointment Scheduling System\n";
    cout << "3. Medical Supply Inventory Management\n";
    cout << "4. Healthcare Resource Allocation Sysytem\n";
    cout << "5. Exit\n";
    cout << "Enter your choice: ";
}

int main()
{
    vector<Inc> incidents;
    priority_queue<Pat, vector<Pat >, PatComp> urgQueue; // Urgent appointments (Min-Heap)
    queue<Pat> regQueue; // Regular appointments (FIFO Queue)
    vector<int> slots; // Available time slots for appointments
    const int hashSize = 10; // Size of the hash table
    vector<vector<Item>> invHashTable(hashSize); // Hash Table
    BSTNode* bst = nullptr; // Root of Binary Search Tree
    vector<vector<int>> g; // g: graph
    vector<N> nodes; // N: Node
    int n;

    int choice;
    do
    {
        showMainMenu();
        cin >> choice;

        switch (choice)
        {
        case 1:
            handleIncidentUserInput(incidents);
            break;
        case 2:
            displayPatientMenu(urgQueue, regQueue, slots);
            break;
        case 3:
            showInventoryMenu(hashSize, invHashTable, bst);
            break;
        case 4:
            menufour(g,nodes,n);
            break;
        case 5:
            cout << "Exiting program.\n";
            break;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    }
    while (choice != 5);

    return 0;
}

// Implementations for the first program functions
vector<int> rabinSearch(const string& text, const string& pattern)
{
    vector<int> positions;
    int textLength = text.length();
    int patternLength = pattern.length();

    if (patternLength > textLength)
    {
        return positions;
    }

    const int d = 256;
    const int q = 101;

    int h = 1;
    for (int i = 0; i < patternLength - 1; ++i)
    {
        h = (h * d) % q;
    }

    int patternHash = 0;
    int textHash = 0;

    for (int i = 0; i < patternLength; ++i)
    {
        patternHash = (d * patternHash + pattern[i]) % q;
        textHash = (d * textHash + text[i]) % q;
    }

    for (int i = 0; i <= textLength - patternLength; ++i)
    {
        if (patternHash == textHash)
        {
            if (text.substr(i, patternLength) == pattern)
            {
                positions.push_back(i);
            }
        }

        if (i < textLength - patternLength)
        {
            textHash = (d * (textHash - text[i] * h) + text[i + patternLength]) % q;
            if (textHash < 0)
            {
                textHash = (textHash + q);
            }
        }
    }

    return positions;
}

void searchIncByDesc(vector<Inc>& incidents)
{
    string query;
    cout << "Enter part of the description to search for: ";
    cin.ignore();
    getline(cin, query);

    bool found = false;

    for (const auto& inc : incidents)
    {
        vector<int> positions = rabinSearch(inc.desc, query);
        if (!positions.empty())
        {
            cout<< "Found in incident with priority ";
            cout<< inc.priority;
            cout<< ": ";
            cout<< inc.desc << endl;
            found = true;
        }
    }

    if (!found)
    {
        cout << "No incidents found with the description containing \"";
        cout << query;
        cout << "\"\n";
    }
}

void inputIncidents(vector<Inc>& incidents)
{
    int numInc;
    cout << "Enter number of incidents: ";
    cin >> numInc;
    cin.ignore();

    for (int i = 0; i < numInc; ++i)
    {
        Inc newInc;
        newInc.id = i + 1;

        cout << "Enter patient name for incident ";
        cout << i + 1;
        cout << ": ";
        getline(cin, newInc.patName);

        cout << "Enter priority (number) for incident ";
        cout << i + 1;
        cout << ": ";
        cin >> newInc.priority;
        cin.ignore();

        cout << "Enter description for incident ";
        cout << i + 1;
        cout << ": ";
        getline(cin, newInc.desc);

        incidents.push_back(newInc);
    }
    cout << "Incidents added successfully!\n";
}

int partitionIncidents(vector<Inc>& incidents, int low, int high)
{
    int pivot = incidents[high].priority;
    int i = low - 1;

    for (int j = low; j < high; ++j)
    {
        if (incidents[j].priority <= pivot)
        {
            i++;
            swap(incidents[i], incidents[j]);
        }
    }
    swap(incidents[i + 1], incidents[high]);
    return i + 1;
}

void quickSortIncidents(vector<Inc>& incidents, int low, int high)
{
    if (low < high)
    {
        int pi = partitionIncidents(incidents, low, high);
        quickSortIncidents(incidents, low, pi - 1);
        quickSortIncidents(incidents, pi + 1, high);
    }
}

void displayAllIncidents(const vector<Inc>& incidents)
{
    if (incidents.empty())
    {
        cout << "No incidents available.\n";
        return;
    }
    cout << "\n--- All Incidents ---\n";
    cout << "Incident ID | Patient Name | Priority | Description\n";
    for (const auto& inc : incidents)
    {
        cout << inc.id;
        cout << " | ";
        cout << inc.patName;
        cout << " | ";
        cout << inc.priority;
        cout << " | ";
        cout << inc.desc << endl;
    }
}

void displayIncByPriority(const vector<Inc>& incidents)
{
    int priority;
    cout << "Enter the priority of the incident to search: ";
    cin >> priority;

    bool found = false;
    for (const auto& inc : incidents)
    {
        if (inc.priority == priority)
        {
            cout << "Found incident: ";
            cout << inc.id;
            cout << " | ";
            cout << inc.patName;
            cout << " | ";
            cout << inc.desc << endl;
            found = true;
        }
    }
    if (!found)
    {
        cout << "No incidents found with priority ";
        cout << priority << endl;
    }
}

void editIncDetails(vector<Inc>& incidents)
{
    int index;
    cout << "Enter the incident number to edit (1 to ";
    cout << incidents.size();
    cout << "): ";
    cin >> index;
    if (index < 1 || index > incidents.size())
    {
        cout << "Invalid incident number!" << endl;
        return;
    }
    index--;

    int newPriority;
    string newDescription;
    cout << "Enter new priority for incident ";
    cout << index + 1;
    cout << ": ";
    cin >> newPriority;
    cin.ignore();
    cout << "Enter new description for incident ";
    cout << index + 1;
    cout << ": ";
    getline(cin, newDescription);

    incidents[index].priority = newPriority;
    incidents[index].desc = newDescription;
    cout << "Incident ";
    cout << index + 1;
    cout << " updated successfully!\n";
}

void removeIncFromList(vector<Inc>& incidents)
{
    int index;
    cout << "Enter the incident number to remove (1 to ";
    cout << incidents.size();
    cout << "): ";
    cin >> index;
    if (index < 1 || index > incidents.size())
    {
        cout << "Invalid incident number!" << endl;
        return;
    }
    index--;

    incidents.erase(incidents.begin() + index);
    cout << "Incident ";
    cout << index + 1;
    cout << " removed successfully!\n";
}

void showIncidentMenu()
```

```cpp
{
    cout << "\n--- Emergency Medical Incident Categoriztion ---\n";
    cout << "1. Input Incidents\n";
    cout << "2. Prioritize Incidents (Quick Sort)\n";
    cout << "3. Display All Incidents\n";
    cout << "4. Display Specific Incident\n";
    cout << "5. Edit Incident\n";
    cout << "6. Remove Incident\n";
    cout << "7. Search Incidents by Description\n";
    cout << "8. Return to Main Menu\n";
    cout << "Enter your choice: ";
}

void handleIncidentUserInput(vector<Inc>& incidents)
{
    int choice;
    do
    {
        showIncidentMenu();
        cin >> choice;

        switch (choice)
        {
        case 1:
            inputIncidents(incidents);
            break;
        case 2:
            if (incidents.empty())
            {
                cout << "No incidents available to prioritize.\n";
            }
            else
            {
                quickSortIncidents(incidents, 0, incidents.size() - 1);
                cout << "Incidents prioritized:\n";
                displayAllIncidents(incidents);
            }
            break;
        case 3:
            displayAllIncidents(incidents);
            break;
        case 4:
            displayIncByPriority(incidents);
            break;
        case 5:
            editIncDetails(incidents);
            break;
        case 6:
            removeIncFromList(incidents);
            break;
        case 7:
            searchIncByDesc(incidents);
            break;
        case 8:
            cout << "Returning to main menu.\n";
            break;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    }
    while (choice != 8);
}

// Implementations for the second program functions
void addUrgPat(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue)
{
    Pat newPat;
    cout << "Enter patient ID: ";
    cin >> newPat.id;
    cout<<"Name: ";
    cin >> newPat.name;
    cout<<"Priority: ";
    cin >> newPat.priority;
    cout<<"Phone number: ";
    cin >> newPat.phone;
    cout<<"Ailment: ";
    cin >> newPat.ailment;
    cout<< "Appointment time(in 24-hour format, e.g., 900 for 9:00 AM): ";
    cin >> newPat.aptTime;
    urgQueue.push(newPat);
    cout << "Patient added to urgent appointments.\n";
}

void viewUrgPats(const priority_queue<Pat, vector<Pat>, PatComp>& urgQueue)
{
    if (urgQueue.empty())
    {
        cout << "No urgent appointments to display.\n";
        return;
    }

    priority_queue<Pat, vector<Pat>, PatComp> tempQueue = urgQueue; // Create a temporary copy to display all patients
    cout << "\n--- Urgent Appointments ---\n";
    while (!tempQueue.empty())
    {
        Pat p = tempQueue.top();
        tempQueue.pop();
        cout << "ID: ";
        cout << p.id;
        cout << ", Name: ";
        cout << p.name;
        cout << ", Priority: ";
        cout << p.priority;
        cout << ", Phone: ";
        cout << p.phone;
        cout << ", Ailment: ";
        cout << p.ailment;
        cout << ", Time: ";
        cout << p.aptTime << endl;
    }
}
void viewregPats(queue<Pat>& regQueue)
{
    if (regQueue.empty())
    {
        cout << "No regular appointments to display.\n";
        return;
    }

     queue<Pat> tempQueue = regQueue; // Create a temporary copy to display all patients
    cout << "\n--- Regular Appointments ---\n";
    while (!tempQueue.empty())
    {
        Pat p = tempQueue.front();
        tempQueue.pop();
        cout << "ID: ";
        cout << p.id;
        cout << ", Name: ";
        cout << p.name;
        cout << ", Time: ";
        cout << p.aptTime << endl;
    }
}

void addRegPat(queue<Pat>& regQueue)
{
    Pat newPat;
    cout << "Enter patient ID: ";
    cin >> newPat.id;
    cout<< "Name: ";
    cin >> newPat.name;
    cout<<"Appointment time (in 24-hour format, e.g., 900 for 9:00 AM): ";
    cin >> newPat.aptTime;
    newPat.priority = INT_MAX; // Default priority for regular appointments
    regQueue.push(newPat);
    cout << "Regular appointment added.\n";
}

void serveNextUrgPat(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue)
{
    if (urgQueue.empty())
    {
        cout << "No urgent appointments.\n";
        return;
    }
    Pat p = urgQueue.top();
    urgQueue.pop();
    cout << "Serving urgent appointment:\n";
    cout << "Patient ID ";
    cout << p.id;
    cout << ", Name: ";
    cout << p.name;
    cout << ", Priority: ";
    cout << p.priority << endl;
}

void serveNextRegPat(queue<Pat>& regQueue)
{
    if (regQueue.empty())
    {
        cout << "No regular appointments.\n";
        return;
    }
    Pat p = regQueue.front();
    regQueue.pop();
    cout << "Serving regular appointment:\n";
    cout << "Patient ID ";
    cout << p.id;
    cout << ", Name: ";
    cout << p.name;
    cout << ", Time: ";
    cout << p.aptTime << endl;
}

void manageAvailableSlots(vector<int>& slots)
{
    int n;
    cout << "Enter the number of available time slots: ";
    cin >> n;

    slots.resize(n);
    cout << "Enter the time slots (in 24-hour format, e.g., 900 for 9:00 AM): ";
    for (int i = 0; i < n; ++i)
    {
        cin >> slots[i];
    }

    cout << "Available time slots: ";
    for (int s : slots)
    {
        cout << s;
        cout << " ";
    }
    cout << endl;
}

void sortRegPats(queue<Pat>& regQueue)
{
    vector<Pat> tempPats;

    while (!regQueue.empty())
    {
        tempPats.push_back(regQueue.front());
        regQueue.pop();
    }

    sort(tempPats.begin(), tempPats.end(), [](const Pat& a, const Pat& b)
    {
        return a.aptTime < b.aptTime;
    });

    for (const Pat& p : tempPats)
    {
        regQueue.push(p);
    }

    cout << "Regular appointments sorted by time.\n";
}

void updatePatData(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue)
{
    int id;
    cout << "Enter patient ID to change details: ";
    cin >> id;

    priority_queue<Pat, vector<Pat>, PatComp> tempQueue;
    bool found = false;

    while (!urgQueue.empty())
    {
        Pat p = urgQueue.top();
        urgQueue.pop();
        if (p.id == id)
        {
            cout << "Enter new details for patient ID " << id << "\n";
            cout << "New name: ";
            cin >> p.name;
            cout << "New priority: ";
            cin >> p.priority;
            cout << "New phone number: ";
            cin >> p.phone;
            cout << "New ailment: ";
            cin >> p.ailment;
            found = true;
        }
        tempQueue.push(p);
    }

    urgQueue = tempQueue;

    if (found)
    {
        cout << "Patient details updated.\n";
    }
    else
    {
        cout << "Patient ID not found.\n";
    }
}

void removePatAppointment(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue, queue<Pat>& regQueue)
{
    int id;
    char ap;
    cout << "Urgent (U)  or Regular (R): ";
    cin >> ap;
    cout << "Enter patient ID to remove appointment: ";
    cin >> id;

    bool found = false;
    priority_queue<Pat, vector<Pat>, PatComp> tempQueue;
    if(ap=='U' || ap== 'u')
    {
    while (!urgQueue.empty())
    {
        Pat p = urgQueue.top();
        urgQueue.pop();
        if (p.id == id)
        {
            found = true;
            cout << "Appointment removed: ";
            cout << p.name;
            cout << " with ID ";
            cout << p.id << endl;
        }
        else
        {
            tempQueue.push(p);
        }
    }

    urgQueue = tempQueue;
    }
    if (ap=='R' || ap=='r')
    {
        queue<Pat> tempRegQueue;
        while (!regQueue.empty())
        {
            Pat p = regQueue.front();
            regQueue.pop();
            if (p.id == id)
            {
                found = true;
                cout << "Regular appointment removed: ";
                cout << p.name;
                cout << " with ID ";
                cout << p.id << endl;
            }
            else
            {
                tempRegQueue.push(p);
            }
        }
        regQueue = tempRegQueue;
    }

    if (!found)
    {
        cout << "Patient ID not found in any appointment.\n";
    }
}

void displayPatientMenu(priority_queue<Pat, vector<Pat>, PatComp>& urgQueue, queue<Pat>& regQueue,
vector<int>& slots)
{
    int choice;
    do
    {
        cout << "\n--- Patient Appointment Scheduling System ---\n";
        cout << "1. Add Urgent Appointment\n";
        cout << "2. Add Regular Appointment\n";
        cout << "3. Serve Urgent Appointment\n";
        cout << "4. Serve Regular Appointment\n";
        cout << "5. View Urgent Appointments\n";
        cout << "6. View Regular Appointments\n";
        cout << "7. Manage Time Slots\n";
        cout << "8. Sort Regular Appointments by Time\n";
        cout << "9. Change Patient Details\n";
        cout << "10. Remove Appointment\n";
        cout << "11. Return to Main Menu\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice)
        {
            case 1:
                addUrgPat(urgQueue);
                break;
            case 2:
                addRegPat(regQueue);
                break;
            case 3:
                serveNextUrgPat(urgQueue);
                break;
            case 4:
                serveNextRegPat(regQueue);
                break;
            case 5:
                viewUrgPats(urgQueue);
                break;
            case 6:
                viewregPats(regQueue);
                break;
            case 7:
                manageAvailableSlots(slots);
                break;
            case 8:
                sortRegPats(regQueue);
                break;
            case 9:
                updatePatData(urgQueue);
                break;
            case 10:
                removePatAppointment(urgQueue, regQueue);
                break;
            case 11:
                cout << "Returning to main menu.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
            }
        }
    while (choice != 11);
}

// Implementations for the third program functions
int hashItem(const string& itemSearchName, int hashSize)
{
    int hashValue = 0;
    for (char c : itemSearchName)
    {
        hashValue = (hashValue * 31 + c) % hashSize;
    }
    return hashValue;
}

void addInvItem(vector<vector<Item>>& invHashTable, int hashSize)
{
    Item newInvItem;
    cout << "Enter item name: ";
    cin >> newInvItem.name;
    cout<< "Quantity: ";
    cin >> newInvItem.quantity;
    cout<<"Priority: ";
    cin >> newInvItem.priority;

    int hashValue = hashItem(newInvItem.name, hashSize);
    invHashTable[hashValue].push_back(newInvItem);
    cout << "Item '";
    cout << newInvItem.name;
    cout << "' added to inventory.\n";
}

void findInvItem(const vector<vector<Item>>& invHashTable, int hashSize)
{
    string itemSearchName;
    cout << "Enter item name to search: ";
    cin >> itemSearchName;

    int hashValue = hashItem(itemSearchName, hashSize);
    for (const Item& item : invHashTable[hashValue])
    {
        if (item.name == itemSearchName)
        {
            cout << "Item found: Name = " << item.name
                 << ", Quantity = " << item.quantity
                 << ", Priority = " << item.priority << endl;
            return;
        }
    }
    cout << "Item '";
    cout << itemSearchName;
    cout << "' not found in inventory 1.\n";
}

void editInvItem(vector<vector<Item>>& invHashTable, int hashSize)
{
    string itemSearchName;
    cout << "Enter item name to update: ";
    cin >> itemSearchName;

    int hashValue = hashItem(itemSearchName, hashSize);
    for (Item& item : invHashTable[hashValue])
    {
        if (item.name == itemSearchName)
        {
            cout << "Enter new quantity: ";
            cin >> item.quantity;
            cout<<"New priority: ";
            cin >> item.priority;
            cout<< "Item '";
            cout<< item.name;
            cout << "' updated.\n";
            return;
        }
    }
    cout << "Item '";
    cout << itemSearchName;
    cout << "' not found in inventory 1.\n";
}

void removeInvItem(vector<vector<Item>>& invHashTable, int hashSize)
{
    string itemSearchName;
    cout << "Enter item name to delete: ";
    cin >> itemSearchName;

    int hashValue = hashItem(itemSearchName, hashSize);
    auto& bucket = invHashTable[hashValue];
    for (auto it = bucket.begin(); it != bucket.end(); ++it)
    {
        if (it->name == itemSearchName)
        {
            bucket.erase(it);
            cout << "Item '";
            cout << itemSearchName;
            cout << "' deleted from inventory 1.\n";
            return;
        }
    }
    cout << "Item '";
    cout << itemSearchName;
    cout << "' not found in inventory 1.\n";
}

void showInvItems(const vector<vector<Item>>& invHashTable)
{
    vector<Item> collectedItems;

    for (const auto& bucket : invHashTable)
    {
        for (const Item& item : bucket)
        {
            collectedItems.push_back(item);
        }
    }
    if (collectedItems.empty())
    {
        cout << "No items in inventory 1 to display.\n";
        return;
    }
    cout << "Items sorted by restocking priority:\n";
    for (const Item& item : collectedItems)
    {
        cout << "Name: ";
        cout << item.name;
        cout << ", Quantity: ";
        cout << item.quantity;
        cout << ", Priority: ";
        cout << item.priority << endl;
    }
}

BSTNode* addBSTNode(BSTNode* root, Item item)
{
    if (!root)
    {
        return new BSTNode(item);
    }
    if (item.name < root->item.name)
    {
        root->left = addBSTNode(root->left, item);
    }
    else if (item.name > root->item.name)
    {
        root->right = addBSTNode(root->right, item);
    }
    return root;
}

void findBSTNode(BSTNode* root, const string& itemSearchName)
{
    if (!root)
    {
        cout << "Item '";
        cout << itemSearchName;
        cout << "' not found in inventory 2.\n";
        return;
    }

    if (itemSearchName < root->item.name)
    {
        findBSTNode(root->left, itemSearchName);
    }
     else if (itemSearchName > root->item.name)
    {
        findBSTNode(root->right, itemSearchName);
    }
    else
    {
        cout<< "Item found: Name = " << root->item.name
             << ", Quantity = " << root->item.quantity
             << ", Priority = " << root->item.priority << endl;
    }
}

void listBST(BSTNode* root)
{
    if (!root)
    {
        return;
    }
    listBST(root->left);
    cout << "Name: " << root->item.name
         << ", Quantity: " << root->item.quantity
         << ", Priority: " << root->item.priority << endl;
    listBST(root->right);
}

// Delete Item from BST
BSTNode* removeBSTNode(BSTNode* root, const string& itemName)
{
    if (!root)
    {
        cout << "Item '";
        cout << itemName;
        cout << "' not found in inventory 2.\n";
        return root; // Item not found
    }

    // Traverse the tree
    if (itemName < root->item.name)
    {
        root->left = removeBSTNode(root->left, itemName); // Go left
    }
    else if (itemName > root->item.name)
    {
        root->right = removeBSTNode(root->right, itemName); // Go right
    }
    else
    {
        // Node with only one child or no child
        if (!root->left)
        {
            BSTNode* temp = root->right;
            delete root; // Free memory
            return temp; // Return the right child
        }
        else if (!root->right)
        {
            BSTNode* temp = root->left;
            delete root; // Free memory
            return temp; // Return the left child
        }

        // Node with two children: Get the inorder successor (smallest in the right subtree)
        BSTNode* temp = root->right;
        while (temp && temp->left)
        {
            temp = temp->left; // Find the leftmost node
        }

        // Copy the inorder successor's content to this node
```

```cpp
        root->item = temp->item;

        // Delete the inorder successor
        root->right = removeBSTNode(root->right, temp->item.name);
    }
    return root; // Return the (potentially unchanged) node pointer
}

void clearBST(BSTNode* root)
{
    if (!root) return;
    clearBST(root->left);
    clearBST(root->right);
    delete root;
}

void showInventoryMenu(const int hashSize, vector<vector<Item>>& invHashTable, BSTNode* bst)
{
    int menuChoice;
    do
    {
        cout << "\n--- Medical Supply Inventory Management ---\n";
        cout << "1. Add Item to Inventory 1\n";
        cout << "2. Search Item in Inventory 1\n";
        cout << "3. Update Item in Inventory 1\n";
        cout << "4. Delete Item from Inventory 1\n";
        cout << "5. Display Restock Priority \n";
        cout << "6. Add Item to Inventory 2\n";
        cout << "7. Search Item in Inventory 2\n";
        cout << "8. delete Item from Inventory 2\n";
        cout << "9. Display Inventory 2\n";
        cout << "10. Return to Main Menu\n";

        cout << "Enter your choice: ";
        cin >> menuChoice;

        switch (menuChoice) {
            case 1:
                addInvItem(invHashTable, hashSize);
                break;
            case 2:
                findInvItem(invHashTable, hashSize);
                break;
            case 3:
                editInvItem(invHashTable, hashSize);
                break;
            case 4:
                removeInvItem(invHashTable, hashSize);
                break;
            case 5:
                showInvItems(invHashTable);
                break;
            case 6:
            {
                Item newInvItem;
                cout << "Enter item name: ";
                cin >> newInvItem.name;
                cout<<"Quantity: ";
                cin >> newInvItem.quantity;
                cout<<"Priority: ";
                cin >> newInvItem.priority;
                bst = addBSTNode(bst, newInvItem);
                cout << "Item '";
                cout << newInvItem.name;
                cout << "' added to inventory 2.\n";
                break;
            }
            case 7:
            {
                string itemSearchName;
                cout << "Enter item name to search in inventory 2: ";
                cin >> itemSearchName;
                findBSTNode(bst, itemSearchName);
                break;
            }
            case 8:
            {
                string itemSearchName;
```

```cpp
                    cout << "Enter item name to delete from BST: ";
                    cin >> itemSearchName;
                    bst = removeBSTNode(bst, itemSearchName); // Delete from BST
                    break;
                }
                case 9:
                    cout << "Inventory 2(sorted by name):\n";
                    listBST(bst);
                    break;
                case 10:
                    cout << "Returning to main menu.\n";
                    break;
                default:
                    cout << "Invalid choice! Please try again.\n";
            }
        }
        while (menuChoice != 10);
    }

    // Function to handle invalid integer input
    int vInt(string prompt)
    { // vInt: validInt
        int val;
        while (true)
        {
            cout << prompt;
            if (cin >> val)
            {
                cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Clear input buffer
                return val;
            }
            else
            {
                cout << "Invalid input. Please enter a valid integer.\n";
                cin.clear();  // Clear error flag
                cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Clear input buffer
            }
        }
    }

    // Check if node index is valid
    bool vNode(int idx, int n)
    { // vNode: validNode
        return (idx >= 0 && idx < n);
    }

    // Dijkstra's algorithm for shortest path
    void dijkstra(const vector<vector<int>> &g, int s, int n, vector<int> &d, vector<int> &p)
    { // g: graph, s: source, d: dist, p: prev
        d.assign(n, INT_MAX); // Initialize distances to INT_MAX
        p.assign(n, -1);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
        d[s] = 0;
        pq.push({0, s});

        while (!pq.empty())
        {
            int u = pq.top().second;
            int dist = pq.top().first;
            pq.pop();

            if (dist > d[u])
            {
                continue;
            }

            for (int v = 0; v < n; ++v)
            {
                if (g[u][v] > 0 && g[u][v] != INT_MAX)
                { // Check for valid path
                    int newDist = d[u] + g[u][v];
                    if (newDist < d[v])
                    {
                        d[v] = newDist;
                        p[v] = u;
                        pq.push({newDist, v});
                    }
                }
            }
        }
    }

    // Print the shortest path and distance
    void printPath(const vector<int> &p, int s, int d, const vector<int>& dist)
    { // p: prev, s: source, d: destination
        if (d == -1 || d == s)
        {
            cout << s << " ";
            return;
        }
        printPath(p, s, p[d], dist);
        cout << d << " ";
    }

    // Input graph details
    void inputGraph(int &n, vector<vector<int>> &g, vector<N> &nodes)
    { // g: graph, N: Node
        //cout << "Enter number of hospitals: ";
        n = vInt("Enter number of hospitals: ");  // Ensuring valid integer input

        g.resize(n, vector<int>(n, 0));
        nodes.resize(n);

        for (int i = 0; i < n; ++i)
        {
            cout << "Enter name of hospital ";
            cout << i;
            cout << ": ";
            cin >> nodes[i].n; // n: name

            int rCount = vInt("Enter number of resources for hospital " + nodes[i].n + ": "); //
    rCount: resourceCount
            for (int j = 0; j < rCount; ++j)
            {
                string rName; // rName: resourceName
                int avail; // avail: availability
                cout << "Enter resource name : ";
                cin >> rName;
                avail = vInt("Enter availability: ");
                nodes[i].r.push_back({rName, avail}); // r: resources
            }

            for (int j = 0; j < i; ++j)
            {
                cout << "Enter distance between ";
                cout << nodes[i].n;
                cout << " and ";
                cout << nodes[j].n;
                cout << " (enter -1 for no path): \n";
                int distance = vInt("Enter distance: ");
                if(distance==NO_PATH)
                {
                    g[i][j]=INT_MAX;
                }
                else
                {
                    g[i][j]=distance;
                }
                g[j][i] = g[i][j]; // Ensure symmetry
            }
        }
    }

    // Display resources for each hospital
    void checkResources(int n, const vector<N> &nodes)
    { // N: Node
        for (int i = 0; i < n; ++i)
        {
            cout << "Hospital ";
            cout << i;
            cout << " - ";
            cout << nodes[i].n;
            cout << ":\n";
            for (const auto &res : nodes[i].r)
            { // r: resources
                cout << "Resource: ";
```

```cpp
                cout << res.first;
                cout << ", Availability: ";
                cout << res.second << endl;
            }
        }
    }

    // Modify resources for a hospital
    void modifyResources(vector<N> &nodes)
    { // N: Node
        int idx = vInt("Enter hospital index to modify resources: ");
        if (!vNode(idx, nodes.size()))
        { // vNode: validNode
            cout << "Invalid hospital index.\n";
            return;
        }
        cout << "Modifying resources for ";
        cout << nodes[idx].n;
        cout << ":\n";
        int choice;
        do
        {
            cout << "\n1. Add new resource\n";
            cout << "2. Update existing resource\n";
            cout << "3. Remove existing resource\n";
            cout << "4. Go back\n";
            choice = vInt("Enter your choice: ");
            switch (choice)
            {
            case 1:
            {
                string rName; // rName: resourceName
                int avail; // avail: availability
                cout << "Enter resource name: ";
                cin >> rName;
                avail = vInt("Enter availability: ");
                nodes[idx].r.push_back({rName, avail});
                cout << "Resource added successfully.\n";
                break;
            }
            case 2:
            {
                string rName; // rName: resourceName
                cout << "Enter resource name to update: ";
                cin >> rName;
                for (auto &res : nodes[idx].r)
                { // r: resources
                    if (res.first == rName) {
                        cout << "Enter new availability: ";
                        res.second = vInt("Enter new availability: ");
                        cout << "Resource updated successfully.\n";
                        break;
                    }
                }
                break;
            }
            case 3:
            {
                string rName; // rName: resourceName
                cout << "Enter resource name to remove: ";
                cin >> rName;
                nodes[idx].r.erase(remove_if(nodes[idx].r.begin(), nodes[idx].r.end(),
                                                    [rName](const pair<string, int> &res)
                                                    {
                                                        return res.first == rName;
                                                    }),
                                            nodes[idx].r.end());
                cout << "Resource removed successfully.\n";
                break;
            }
            case 4:
                cout << "Going back.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
            }
        }
        while (choice != 4);
```

```cpp
}

// Add a new hospital node
void addNewNode(int &n, vector<vector<int>> &g, vector<N> &nodes)
{
    N newNode; // newNode: Node
    cout << "Enter name of new hospital: ";
    cin >> newNode.n; // n: name

    int rCount = vInt("Enter number of resources for the new hospital: "); // rCount: resourceCount
    for (int i = 0; i < rCount; ++i)
    {
        string rName; // rName: resourceName
        int avail; // avail: availability
        cout << "Enter resource name: ";
        cin >> rName;
        avail = vInt("Enter availability: ");
        newNode.r.push_back({rName, avail}); // r: resources
    }

    // Resize the graph to accommodate the new node
    g.resize(n + 1, vector<int>(n + 1, 0)); // Resize g to (n+1) x (n+1) and initialize with 0

    // Set distances for the new node
    for (int i = 0; i < n; ++i)
    {
        cout << "Enter distance between ";
        cout << newNode.n;
        cout << " and ";
        cout << nodes[i].n;
        cout << ": ";
        cout << "(If no path then put -1)\n";
        int distance = vInt("Enter distance: ");
        if(distance == NO_PATH)
        {
            g[n][i]=INT_MAX;
        }
        else
        {
            g[n][i]=distance;
        }
        g[i][n] = g[n][i]; // Ensure symmetry
    }

    n++; // Increment the number of nodes
    nodes.push_back(newNode); // Add the new node to the list

}

// Find hospitals with a given resource
void findHospitalsWithResource(vector<N> &nodes, int n)
{
    string r; // r: resource
    cout << "Enter resource name: ";
    cin >> r;

    UF uf(n); // UF: Union-Find

    // Group hospitals with the same resource
    for (int i = 0; i < n; ++i)
    {
        for (const auto &res : nodes[i].r)
        {
            if (res.first == r)
            {
                for (int j = 0; j < n; ++j)
                {
                    if (i != j)
                    {
                        for (const auto &otherRes : nodes[j].r)
                        {
                            if (otherRes.first == r)
                            {
                                uf.u(i, j); // unite hospitals
                            }
                        }
                    }
                }
            }
        }
    }

    // Create a vector to store hospitals by their group
    vector<vector<string>> groups(n); // Each index represents a group

    // Populate the groups with hospital names
    for (int i = 0; i < n; ++i)
    {
        int grp = uf.f(i); // Find the group of the hospital
        groups[grp].push_back(nodes[i].n); // n: name
    }

    // Collect hospitals that have the specified resource
    vector<string> hospitalsWithResource;

    // Check each group for hospitals with the specified resource
    for (const auto &group : groups)
    {
        for (const auto &hospitalName : group)
        {
            // Find the index of the hospital in nodes to check its resources
            for (int i = 0; i < n; ++i)
            {
                if (nodes[i].n == hospitalName)
                {
                    // Check if this hospital has the resource
                    for (const auto &res : nodes[i].r)
                    {
                        if (res.first == r)
                        {
                            hospitalsWithResource.push_back(hospitalName);
                            break; // No need to check other resources for this hospital
                        }
                    }
                    break; // Break after finding the hospital
                }
            }
        }
    }

    // Display hospitals with the specified resource
    cout << "\nHospitals with resource ";
    cout << r;
    cout << ":\n";
    for (const auto &hospitalName : hospitalsWithResource)
    {
        cout << hospitalName;
        cout << "\n";
    }
}

void displayAdjacencyMatrix(const vector<vector<int>> &g, int n)
{
    cout << "Adjacency Matrix:\n";

    // Print the header row with indices
    cout << "   "; // Initial space for row header
    for (int i = 0; i < n; ++i)
    {
        cout << i;
        cout << " "; // Print each hospital index
    }
    cout << endl; // Move to the next line after the header

    // Print each row of the adjacency matrix
    for (int i = 0; i < n; ++i)
    {
        cout << i;
        cout << " "; // Print the hospital index for the row
        for (int j = 0; j < n; ++j)
        {
            if (g[i][j] == INT_MAX)
            {
                cout << "INF "; // Use "INF" to represent no path
            }
            else
```

```cpp
            {
                cout << g[i][j];
                cout << " "; // Print the distance
            }
        }
        cout << endl; // Move to the next line after each row
    }
}

void menufour(vector<vector<int>> &g,vector<N> &nodes, int n)
{
    inputGraph(n, g, nodes); // g: graph, N: Node

    int choice;
    do
    {
        cout << "\nHospital System Menu:\n";
        cout << "1. View all hospitals and resources\n";
        cout << "2. Modify resources for a hospital\n";
        cout << "3. Add a new hospital\n";
        cout << "4. Find hospitals with a specific resource\n";
        cout << "5. Perform shortest path search\n";
        cout << "6. Display adjacency matrix\n";
        cout << "7. Exit\n";
        choice = vInt("Enter your choice: "); // vInt: validInt

        switch (choice)
        {
        case 1:
            checkResources(n, nodes); // n: number of hospitals, N: Node
            break;
        case 2:
            modifyResources(nodes); // N: Node
            break;
        case 3:
            addNewNode(n, g, nodes); // g: graph, N: Node
            break;
        case 4:
            findHospitalsWithResource(nodes, n); // N: Node
            break;
        case 5:
        {
            int src = vInt("Enter the source hospital index for the shortest path: "); // src: source
            int dest = vInt("Enter the destination hospital index for the shortest path: "); //
dest: destination

            if (!vNode(src, n) || !vNode(dest, n)) { // vNode: validNode
                cout << "Invalid hospital indices.\n";
                break;
            }

            vector<int> d, p; // d: dist, p: prev
            dijkstra(g, src, n, d, p); // g: graph, src: source, d: dist, p: prev
            cout << "Shortest path from ";
            cout << nodes[src].n;
            cout << " to ";
            cout << nodes[dest].n;
            cout << ":\n"; // n: name
            printPath(p, src, dest, d); // p: prev, src: source, dest: destination, d: dist
            cout << "\nDistance: ";
            cout << d[dest];
            cout << endl; // d: dist
            break;
        }
        case 6:
            displayAdjacencyMatrix(g,n);
            break;

        case 7:
            cout << "Exiting program.\n";
            break;
        default:
            cout << "Invalid choice. Please try again.\n";
        }
    }
    while (choice != 7);
}
```