Patient Management System

Struct `Bill`

Unabstracted Code Snippet

```
struct Bill
{
    double amount;
    string description;

    Bill(double amt, string desc) : amount(amt), description(desc) {}
};
```

Explanation of the Code

The `Bill` struct is designed to store information related to billing. It includes an amount and a description, and a constructor to initialize these fields.

Explanation of Variables

- `amount`: A double variable that stores the billing amount.
- `description`: A string that provides a description of the bill.

Struct `patient_details` (`p_d`)

Unabstracted Code Snippet

```
typedef struct patient_details
{
    string name, email, DOB, address, emergency_contact_name,
relation_with_emergency_contact, email_of_emergency_contact,
mobile_emergency_contact, appointment, sex;
    long int mobile_number;
} p_d;
```

Explanation of the Code

This typedef struct represents detailed personal and contact information of a patient. It includes fields for name, email, date of birth, and other relevant information.

Explanation of Variables

- Personal details like `name`, `email`, `DOB`, `address`, `appointment`, `sex`.
- Emergency contact details like `emergency_contact_name`, `relation_with_emergency_contact`, `email_of_emergency_contact`, `mobile_emergency_contact`.
- `mobile_number`: A long integer for the patient's mobile number.

Struct `medications` (`med`)

Unabstracted Code Snippet

```
typedef struct medications
{
    queue<string> meds;
} med;
```

Explanation of the Code

This structure is used to manage a patient's medication information. It uses a queue to store a list of medications as strings.

Explanation of Variables

- `meds`: A queue of strings, each representing a medication.

Struct `patient_history` (`p_h`)

Unabstracted Code Snippet

```
typedef struct patient_history
{
    stack<string> his;
} p_h;
```

Explanation of the Code

This structure keeps track of a patient's medical history using a stack, allowing for easy addition and retrieval of past medical events.

Explanation of Variables

- `his`: A stack of strings, with each string detailing an event in the patient's medical history.

Struct `patient_information` (`info`)

Unabstracted Code Snippet

```
typedef struct patient_information
{
    p_d details;
    med meds;
    p_h his;
    vector<Bill> bills;
    long long int mobile_number;
} info;
```

Explanation of the Code

This structure aggregates all patient information, including personal details, medications, medical history, and billing information.

Explanation of Variables

- `details`: An instance of `p_d` storing patient's personal details.
- `meds`: An instance of `med` for managing medications.
- `his`: An instance of `p_h` for storing patient history.
- `bills`: A vector of `Bill` objects for managing billing information.
- `mobile_number`: A long long integer for the patient's mobile number.

Struct `node`

Unabstracted Code Snippet

```
struct node
{
    info patient;
    int height;
    node *left;
    node *right;

    node(info p) : patient(p), height(1), left(nullptr), right(nullptr) {}
};
```

Explanation of the Code

This struct represents a node in a binary tree, primarily for organizing patient information. It contains patient information, height of the node, and pointers to left and right child nodes.

Explanation of Variables

- `patient`: An instance of `info` containing detailed patient information.
- `height`: An integer representing the height of the node in the tree.
- `left`, `right`: Pointers to the left and right child nodes, respectively.

Function: `Height(node *n)`

Unabstracted Code Snippet

```
int Height(node *n)
{
    if (n == nullptr)
    {
        return 0;
    }
    return n->height;
}
```

Explanation of the Code

The `Height` function calculates the height of a node in the binary tree. It returns `0` if the node is `nullptr`, indicating that the node does not exist. Otherwise, it returns the height of the node.

Explanation of Variables

- `n`: A pointer to a `node`. It is the node whose height is being determined.

Complexity Analysis

- Time Complexity: O(1), as it's a direct access to a property of the node.
- Space Complexity: O(1), no additional space is used.

Function: `new_node(info patient)`

Unabstracted Code Snippet

```
node *new_node(info patient)
{
    return new (nothrow) node(patient);
}
```

Explanation of the Code

This function dynamically allocates memory for a new `node`, initializing it with the given `patient` information. It uses `nothrow` to avoid throwing an exception if memory allocation fails.

Explanation of Variables

- `patient`: An instance of `info` struct containing patient information.

Complexity Analysis

- Time Complexity: O(1), as it involves a single operation of memory allocation.
- Space Complexity: O(1), creating a single node.

Function: `Search(node *root, long long int x)`

Unabstracted Code Snippet

```
node *Search(node *root, long long int x)
{
    if (root == nullptr)
    {
        return nullptr;
    }

    if (x < root->patient.details.mobile_number)
    {
        return Search(root->left, x);
    }
    else if (x > root->patient.details.mobile_number)
    {
        return Search(root->right, x);
    }
    else
    {
        return root;
```

```
    }
}
```

Explanation of the Code

The `Search` function recursively searches for a node with a specific mobile number in a binary tree. It returns `nullptr` if the node is not found or the root is `nullptr`.

Explanation of Variables

- `root` : A pointer to the root node of the binary tree.
- `x` : The mobile number being searched for.

Complexity Analysis

- Time Complexity: O(log n) in the average case, where n is the number of nodes. O(n) in the worst case (skewed tree).
- Space Complexity: O(log n) due to recursion stack space. O(n) in the worst case.

Function: `Insert(node *root, info patient, int *count)`

Unabstracted Code Snippet

```
node *Insert(node *root, info patient, int *count)
{
    if (root == nullptr)
    {
        root = new_node(patient);
        if (root == nullptr)
        {
            cerr << "Memory allocation failed" << endl;
            return nullptr;
        }
        (*count)++;
        return root;
    }

    if (patient.mobile_number < root->patient.mobile_number)
    {
        root->left = Insert(root->left, patient, count);
    }
    else if (patient.mobile_number > root->patient.mobile_number)
```

```
    {
        root->right = Insert(root->right, patient, count);
    }

    root->height = 1 + max(Height(root->left), Height(root->right));

    return root;
}
```

Explanation of the Code

The `Insert` function inserts a new node with patient information into the binary tree. It updates the tree's height after insertion and handles duplicate values by ignoring them (this behavior can be modified as needed).

Explanation of Variables

- `root`: A pointer to the root node of the binary tree.
- `patient`: The patient information to be inserted.
- `count`: A pointer to an integer tracking the number of insertions.

Complexity Analysis

- Time Complexity: O(log n) on average for balanced trees, where n is the number of nodes. O(n) in the worst case (skewed tree).
- Space Complexity: O(log n) due to recursion stack space. O(n) in the worst case.

Function: `Successor(node *root, int *count)`

Unabstracted Code Snippet

```
node *Successor(node *root, int *count)
{
    if (root == nullptr)
    {
        return root;
    }

    if (root->left)
    {
        root = Successor(root->left, count);
    }
```

```
    return root;
}
```

Explanation of the Code

The `Successor` function finds the successor node in a binary tree. It traverses the left subtree to find the minimum value, which is the successor in binary search trees.

Complexity Analysis

- **Time Complexity**: O(h), where h is the height of the tree.
- **Space Complexity**: O(h) due to recursion stack space.

Function: `hashIndex(long long number)`

Unabstracted Code Snippet

```
int hashIndex(long long number)
{
    long long f_n = (3 * number) + 4;
    f_n = f_n % hash_size;
    f_n += hash_size;
    return f_n % hash_size;
}
```

Explanation of the Code

The `hashIndex` function computes a hash index based on the given number. It uses a simple hash function and modulus operation to ensure the index remains within the bounds of `hash_size`.

Complexity Analysis

- **Time Complexity**: O(1), as it's a series of arithmetic operations.
- **Space Complexity**: O(1), no additional space is used.

Function: `new_user_enter_personal_info(info &patient)`

Unabstracted Code Snippet

```
void new_user_enter_personal_info(info &patient)
{
    cout << endl;
    cout <<
"**********************************************************************
**********************" << endl;
    cout << "PATIENT DETAILS" << endl << endl;
    cout << "Name of Patient: ";
    cin.ignore();
    getline(cin, patient.details.name);
    cout << "Email Address of Patient: ";
    getline(cin, patient.details.email);
    cout << "Date of Birth: ";
    getline(cin, patient.details.DOB);
    cout << "Biological Sex of Patient (M/F/O): ";
    getline(cin, patient.details.sex);
    cout << "Address of Patient: ";
    getline(cin, patient.details.address);
    cout << endl;
    cout <<
"**********************************************************************
**********************" << endl;
    cout << "EMERGENCY CONTACT" << endl << endl;
    cout << "Name of Contact: ";
    getline(cin, patient.details.emergency_contact_name);
    cout << "Relation with Contact: ";
    getline(cin, patient.details.relation_with_emergency_contact);
    cout << "Email Address of Contact: ";
    getline(cin, patient.details.email_of_emergency_contact);
    cout << "Phone Number of contact: ";
    getline(cin, patient.details.mobile_emergency_contact);
}
```

Explanation of the Code

This function collects personal and emergency contact information from the user and stores it in the `patient` structure. It prompts the user for each detail and uses `getline` to read the input.

Complexity Analysis

- **Time Complexity**: O(n), where n is the number of characters inputted by the user.
- **Space Complexity**: O(1), as it does not use additional space proportional to the input size.

Function:

`already_registered_user_patient_details(node *patient_node)`

Unabstracted Code Snippet

```
void already_registered_user_patient_details(node *patient_node)
{
    if (patient_node == nullptr)
    {
        cout << "Patient not found in the database." << endl;
        return;
    }

    cout <<
"**********************************************************************
**********************" << endl;
    cout << "PATIENT DETAILS" << endl << endl;

    cout << "User Information:" << endl;
    cout << "-----------------" << endl;
    cout << "Name: " << patient_node->patient.details.name << endl;
    cout << "Mobile Number: " << patient_node->patient.details.mobile_number
<< endl;
    cout << "Email: " << patient_node->patient.details.email << endl;
    cout << "Date of Birth: " << patient_node->patient.details.DOB << endl;
    cout << "Biological Sex of Patient: " << patient_node-
>patient.details.sex << endl;
    cout << "Address of Patient: " << patient_node->patient.details.address
<< endl;

    cout << endl;
    cout <<
"**********************************************************************
**********************" << endl;
    cout << "EMERGENCY CONTACT DETAILS" << endl << endl;

    cout << "Name of Contact: " << patient_node-
>patient.details.emergency_contact_name << endl;
    cout << "Relation with Contact: " << patient_node-
>patient.details.relation_with_emergency_contact << endl;
    cout << "Email Address of Contact: " << patient_node-
>patient.details.email_of_emergency_contact << endl;
}
```

Function: `mobile_number_verification(node *patients_list[], long long number, int &count)`

Unabstracted Code Snippet

```
void mobile_number_verification(node *patients_list[], long long number, int
&count)
{
    int hash_index = hashIndex(number);

    node *patient_node = Search(patients_list[hash_index], number);

    if (patient_node != nullptr)
    {
        cout << "Here!";
        already_registered_user_patient_details(patient_node);
    }
    else
    {
        while (1)
        {
            srand(time(NULL));
            int generated_otp, input_otp;
            generated_otp = (rand() % (899999 + 1)) + 100000;

            cout << "(your OTP is: " << generated_otp << ")" << endl;
            cout << "Kindly Enter the OTP: ";
            cin >> input_otp;
            cout << endl;

            if (generated_otp == input_otp)
            {
                info new_patient_info;
                new_patient_info.details.mobile_number = number;
                new_user_enter_personal_info(new_patient_info);

                patients_list[hash_index] =
Insert(patients_list[hash_index], new_patient_info, &count);
                break;
            }
            else
            {
                cout << "ERROR!! Entered OTP is Wrong!! Please enter the OTP
again." << endl;
            }
```

```
        }
    }
}
```

Explanation of the Code

This function verifies a patient's mobile number. It first computes a hash index and searches for the patient in an array of nodes. If the patient exists, their details are displayed. Otherwise, the function proceeds with OTP verification. Upon successful OTP verification, it collects new patient information and inserts it into the BST tree.

Complexity Analysis

- **Time Complexity**: O(log n) for searching and inserting in the BST tree, where n is the number of nodes. The OTP generation and verification are O(1).
- **Space Complexity**: O(log n) due to recursion in the `Insert` function. O(1) for OTP generation and verification.

Class: `Appointment`

Unabstracted Code Snippet

```
class Appointment
{
public:
    string time;
    string description;

    Appointment(const string &time, const string &description)
        : time(time), description(description) {}
};
```

Explanation of the Code

The `Appointment` class represents an appointment with two main attributes: time and description. It includes a constructor that initializes these attributes.

Explanation of Variables

- `time`: A string that holds the time of the appointment.
- `description`: A string that describes the details of the appointment.

Class: `Calendar`

Unabstracted Code Snippet

```
class Calendar
{
private:
    map<string, vector<Appointment>> appointments;

public:
    void addAppointment(const string &date, const Appointment &appointment)
    {
        appointments[date].push_back(appointment);
    }

    void viewAppointments(const string &date)
    {
        if (appointments.find(date) == appointments.end())
        {
            cout << "No appointments for " << date << endl;
            return;
        }

        cout << "Appointments for " << date << ":" << endl;
        for (const auto &appointment : appointments[date])
        {
            cout << "Time: " << appointment.time << ", Description: " <<
appointment.description << endl;
        }
    }
};
```

Explanation of the Code

The `Calendar` class manages appointments. It stores appointments in a map, where each key is a date and the value is a vector of `Appointment` objects. It provides two methods: `addAppointment` to add a new appointment to the calendar, and `viewAppointments` to view all appointments for a specific date.

Explanation of Variables

- `appointments`: A map where each key is a string representing a date and the value is a vector of `Appointment` objects.

Methods

- `addAppointment`: Adds a new `Appointment` object to the `appointments` map for the specified date.
- `viewAppointments`: Prints all appointments for a given date. If no appointments are found, it notifies the user.

Function: `formatInputDate(string input)`

Unabstracted Code Snippet

```
string formatInputDate(string input)
{
    regex datePattern("(\\d{1,2})[ ]?([a-zA-Z]+)");
    smatch matches;

    if (regex_search(input, matches, datePattern))
    {
        string day = matches[1].str();
        string month = matches[2].str();

        map<string, string> months = {
            {"Jan", "01"}, {"Feb", "02"}, {"Mar", "03"}, {"Apr", "04"},
{"May", "05"}, {"Jun", "06"}, {"Jul", "07"}, {"Aug", "08"}, {"Sep", "09"},
{"Oct", "10"}, {"Nov", "11"}, {"Dec", "12"}};

        if (months.find(month) != months.end())
        {
            if (day.length() == 1)
                day = "0" + day;
            return "2023-" + months[month] + "-" + day;
        }
    }

    return "";
}
```

Explanation of the Code

The `formatInputDate` function transforms a date from a simple format like "5 Dec" into a standardized format "2023-12-05". It uses regular expressions to validate and parse the input

date, and a map to convert month names into their corresponding numerical representation.

Complexity Analysis

- **Time Complexity**: O(1), assuming a constant time for regular expression processing and map lookup.
- **Space Complexity**: O(1), as it uses a fixed amount of extra space.

Struct: `Point`

Unabstracted Code Snippet

```
struct Point
{
    int x, y;
};
```

Explanation of the Code

The `Point` structure represents a point in a 2D space with `x` and `y` coordinates.

Explanation of Variables

- `x`: An integer representing the x-coordinate of the point.
- `y`: An integer representing the y-coordinate of the point.

Arrays: `dx[]` and `dy[]`

Unabstracted Code Snippet

```
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
```

Explanation of the Code

The arrays `dx` and `dy` are used to represent directional movement along the x and y axes, respectively. They are often used in algorithms that require movement in a grid, such as

pathfinding algorithms.

Explanation of Variables

- `dx[]`: An array representing movement in the x-axis. The values correspond to moving left, right, and no movement in the x-axis.
- `dy[]`: An array representing movement in the y-axis. The values correspond to moving up, down, and no movement in the y-axis.

Function: `DFS(vector<vector<char>> maze, int x, int y, int destX, int destY, stack<Point> &path)`

Unabstracted Code Snippet

```
bool DFS(vector<vector<char>> maze, int x, int y, int destX, int destY,
stack<Point> &path)
{
    if (x < 0 || x >= maze.size() || y < 0 || y >= maze[0].size() || maze[x]
[y] == '#' || maze[x][y] == '.')
    {
        return false;
    }

    if (x == destX && y == destY)
    {
        path.push({x, y});
        return true;
    }

    maze[x][y] = '.';

    for (int i = 0; i < 4; i++)
    {
        int adjX = x + dx[i];
        int adjY = y + dy[i];

        if (DFS(maze, adjX, adjY, destX, destY, path))
        {
            path.push({x, y});
            return true;
        }
    }

    maze[x][y] = ' ';
```

```
    return false;
}
```

Function: `addBill(node *patient_node, double amount, const string &description)`

Unabstracted Code Snippet

```
void addBill(node *patient_node, double amount, const string &description)
{
    if (patient_node == nullptr)
    {
        cout << "Patient not found." << endl;
        return;
    }
    patient_node->patient.bills.push_back(Bill(amount, description));
}
```

Explanation of the Code

The `addBill` function adds a new bill to a patient's record. It checks if the patient node exists and then adds the bill with the specified amount and description.

Complexity Analysis

- **Time Complexity**: O(1), as it simply pushes back a new bill into the vector.
- **Space Complexity**: O(1), assuming the vector resizing is negligible.

Function: `viewBills(node *patient_node)`

Unabstracted Code Snippet

```
void viewBills(node *patient_node)
{
    if (patient_node == nullptr)
    {
        cout << "Patient not found." << endl;
        return;
    }
    cout << "Bills for " << patient_node->patient.details.name << ":" <<
```

```
endl;
    for (const auto &bill : patient_node->patient.bills)
    {
        cout << "Amount: $" << bill.amount << ", Description: " <<
bill.description << endl;
    }
}
```

Explanation of the Code

The `viewBills` function displays all the bills associated with a patient. It first checks if the patient exists and then iterates through the bills, printing each bill's amount and description.

Complexity Analysis

- **Time Complexity**: O(k), where k is the number of bills for the patient.
- **Space Complexity**: O(1), as it only prints the existing bills.

Function: `main()`

Unabstracted Code Snippet

```
int main()
{
    Calendar myCalendar;
    string date, time, description;
    node *patients_list[hash_size];
    int count = 0;

    vector<vector<char>> maze;
    FILE *mazeFile;
    mazeFile = fopen("hospitalmaze.txt", "r");
    if (mazeFile == NULL)
    {
        printf("Unable to open the maze file.\n");
        return 0;
    }

    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    vector<char> row;
    map<Point, Point> parentMaze;
    char inputStart, inputEnd;
```

```
while ((read = getline(&line, &len, mazeFile)) != -1)
{
    row.clear();
    for (int i = 0; i < read; ++i)
    {
        if (line[i] != '\n')
        {
            row.push_back(line[i]);
        }
    }
    maze.push_back(row);
}

fclose(mazeFile);
free(line);

map<char, pair<int, int>> pointCoordinates = {
            {'A', {10, 1}},
            {'B', {8, 18}},
            {'C', {16, 5}},
            {'D', {16, 18}},
            {'E', {5, 7}},
            {'F', {11, 2}},
            {'G', {2, 5}},
            {'H', {4, 1}},
            {'I', {17, 7}},
            {'J', {6, 20}},
            {'K', {10, 8}},
            {'L', {17, 12}}};

stack<Point> path;
int startX, startY, destX, destY;

// stack<Point> path;
// Initialize patients_list to nullptr
for (int i = 0; i < hash_size; i++)
{
    patients_list[i] = nullptr;
}

int choice;
long long int mobile;

while (true)
{
```

```
        cout << "\nHospital Management System\n";
        cout << "1. Register New Patient\n";
        cout << "2. Search Old Patient Records\n";
        cout << "3. Book an Appointment\n";
        cout << "4. Check Appointments\n";
        cout << "5. Find Way in Hospital\n";
        cout << "6. Add Bill to Patient\n";
        cout << "7. View Patient Bills\n";
        cout << "8. Exit System\n";
        cout << "Enter your choice: ";
        cin >> choice;


        switch (choice)
        {
            case 1:
                cout <<
"*********************************************************************************
**********************" << endl;
                cout << "PATIENT LOGIN PORTAL" << endl;
                cout << endl;
                cout << "Please Enter Your Mobile Number: ";
                cin >> mobile;
                cout << endl;
                mobile_number_verification(patients_list, mobile, count);
                break;
            case 2:
                cout <<
"*********************************************************************************
**********************" << endl;
                cout << "PATIENT DATA PORTAL" << endl;
                cout << endl;
                cout << "Please Enter Your Mobile Number: ";
                cin >> mobile;
                cout << endl;
                mobile_number_verification(patients_list, mobile, count);
                break;
            case 3:
                cout << "Enter date (e.g., 5 Dec): ";
                cin >> date;
                date = formatInputDate(date);
                if (date.empty())
                {
                    cout << "Invalid date format.\n";
                    break;
                }
```

```
                cout << "Enter time (e.g., 14:30): ";
                cin >> time;
                cout << "Enter description: ";
                cin.ignore();
                getline(cin, description);
                myCalendar.addAppointment(date, Appointment(time,
description));
                break;
            case 4:
                cout << "Enter date to view appointments (e.g., 5 Dec): ";
                cin >> date;
                date = formatInputDate(date);
                if (date.empty())
                {
                    cout << "Invalid date format.\n";
                    break;
                }
                myCalendar.viewAppointments(date);
                break;

            case 5:
                cout << "Enter the letters of the start and destination
points (e.g., A B): ";
                cin >> inputStart >> inputEnd;

                if (pointCoordinates.find(inputStart) !=
pointCoordinates.end() &&
                    pointCoordinates.find(inputEnd) !=
pointCoordinates.end())
                {
                    startX = pointCoordinates[inputStart].first;
                    startY = pointCoordinates[inputStart].second;
                    destX = pointCoordinates[inputEnd].first;
                    destY = pointCoordinates[inputEnd].second;
                }
                else
                {
                    cout << "Invalid input. Please enter letters between A
and L." << endl;
                    break;
                }

                if (DFS(maze, startX, startY, destX, destY, path))
                {
                    cout << "Path found:\n";
                    // Mark the path
```

```
                    while (!path.empty())
                    {
                        Point p = path.top();
                        path.pop();
                        maze[p.x][p.y] = '.';
                    }

                    // Display the maze with the path
                    for (auto &row : maze)
                    {
                        for (char c : row)
                        {
                            cout << c;
                        }
                        cout << endl;
                    }
                }
                else
                {
                    cout << "No path found.\n";
                }

                break;
            case 6:
                cout << "Enter Patient's Mobile Number to Add Bill: ";
                cin >> mobile;
                int hash_index = hashIndex(mobile);
                node *patient_node = Search(patients_list[hash_index],
mobile);
                if (patient_node != nullptr)
                {
                    double billAmount;
                    string billDescription;
                    cout << "Enter Bill Amount: ";
                    cin >> billAmount;
                    cout << "Enter Bill Description: ";
                    cin.ignore();
                    getline(cin, billDescription);
                    addBill(patient_node, billAmount, billDescription);
                }
                else
                {
                    cout << "Patient not found." << endl;
                }
                break;
```

```
            case 7:
                cout << "Enter Patient's Mobile Number to View Bills: ";
                cin >> mobile;
                int hash_index = hashIndex(mobile);
                node *patient_node = Search(patients_list[hash_index],
mobile);
                if (patient_node != nullptr)
                {
                    viewBills(patient_node);
                }
                else
                {
                    cout << "Patient not found." << endl;
                }
                break;
            case 8:
                cout << "Exiting the system...\n";
                return 0;

            default:
                cout << "Invalid choice. Please try again.\n";
        }


    cout << "Exiting program..." << endl;
    return 0;
}
```

Explanation of the Code

The `main` function serves as the entry point for a hospital management system. It handles various functionalities such as patient registration, appointment booking, bill management, and navigating a hospital maze. The function uses a loop to display options and process user input for different tasks.

Major Components of the Code

1. **Initialization**:
   - Initializes `myCalendar`, `patients_list`, and other variables.
   - Reads a maze configuration from a file into a 2D vector, `maze`.
2. **Menu Loop**:
   - Displays options for the user to interact with the system (e.g., register patients, book appointments).

- Uses a `switch` statement to handle different user choices.

3. **Patient Registration and Record Search**:
   - Asks for the patient's mobile number and uses `mobile_number_verification` to either register a new patient or access existing records.
4. **Appointment Booking and Viewing**:
   - Handles appointment booking (`case 3`) and viewing (`case 4`), using the `Calendar` class.
5. **Hospital Maze Navigation**:
   - (`case 5`) Prompts the user for start and end points in the hospital maze and uses the `DFS` function to find a path.
   - Displays the path on the maze layout if found.
6. **Patient Bill Management**:
   - Adds a new bill to a patient's record (`case 6`) and views existing bills (`case 7`) using `addBill` and `viewBills`.
7. **Exit System**:
   - (`case 8`) Exits the program.
8. **Error Handling**:
   - Handles invalid choices and provides feedback to the user.

Complexity Analysis

- **Time Complexity**: Varies depending on the user's choices. For example, maze navigation uses DFS, which has a time complexity of O(n*m).
- **Space Complexity**: Primarily dependent on the size of the maze and the number of patients in the system.