Detailed Explanation of the Code

The program implements a **patient management system** using several key concepts in C programming, including data structures (arrays, queues, and hashing), file handling for offline saving, and basic input/output operations. The system manages patient data, doctor assignments, and a patient queue. Let's break down the code into sections for better understanding:

1. Structs (Data Structures)

a) Patient Struct:

```
struct Patient {
    int id;
    char name[MAX_NAME_LENGTH];
    int age;
    char disease[MAX_NAME_LENGTH];
    char visitHistory[100];
    int isOccupied;  // To check if a slot is occupied in the hash map
};
```

- **Purpose:** This structure is used to store details about each patient, such as their ID, name, age, disease, and visit history.
- **isOccupied:** This flag indicates whether the current slot in the hash map for a patient is occupied.

b) Doctor Struct:

```
struct Doctor {
    int id;
    char name[MAX_NAME_LENGTH];
    char specialty[MAX_NAME_LENGTH];
};
```

- **Purpose:** This structure is used to store details about each doctor, including their ID, name, and specialty.

c) Queue Struct:

```
struct Queue {
    int front, rear;
    int items[MAX_PATIENTS];
};
```

- **Purpose:** This structure is used to represent a waiting queue for patients. It stores the indices of patients in the queue, with `front` and `rear` pointing to the start and end of the queue, respectively.

2. Queue Functions

- **initQueue()**: Initializes the queue by setting `front` and `rear` to -1.
- **isQueueEmpty()**: Checks if the queue is empty by checking if `front` is -1.

- **enqueue()**: Adds a patient to the queue by updating the `rear` and storing the patient's ID.
- **dequeue()**: Removes a patient from the queue by updating the `front` and returning the patient's ID.

3. Hashing and Linear Probing for Patient Management

The program uses a **hash map** to store patient records. This allows for efficient retrieval and management of data based on patient ID. The hash function used is simple and based on the modulo operator:

```
int hashFunction(int id) {
    return id % MAX_PATIENTS;
}
```

This function maps the patient's ID to an index in the array. When a patient is added, the program checks if the index is occupied. If it is, it uses **linear probing** to find the next available slot.

- **addPatient()**: This function adds a new patient to the system, prompting the user for details. It computes the hash index and handles collisions using linear probing.
- **removePatient()**: This function removes a patient by their ID. It searches for the patient in the hash map and marks their slot as empty when found.

4. Offline Data Saving and File Handling

To ensure data persistence, the program uses **file operations** to save patient and doctor data to disk:

```
void saveToFile(struct Patient patients[], int patientCount) {
    FILE *file = fopen("patients.txt", "w");
    for (int i = 0; i < MAX_PATIENTS; i++) {
        if (patients[i].isOccupied) {
            fprintf(file, "%d,%s,%d,%s,%s\n", patients[i].id,
patients[i].name, patients[i].age, patients[i].disease,
patients[i].visitHistory);
        }
    }
    fclose(file);
}

void loadFromFile(struct Patient patients[]) {
    FILE *file = fopen("patients.txt", "r");
    if (file) {
        while (fscanf(file, "%d,%49[^,],%d,%49[^,],%99[^\n]",
&patients[i].id, patients[i].name, &patients[i].age, patients[i].disease,
patients[i].visitHistory) == 5) {
            patients[i].isOccupied = 1;
        }
        fclose(file);
    }
}
```

- **saveToFile()**: This function saves the current patient records to a file. Each patient's details are written as a comma-separated string.

- **loadFromFile()**: This function loads the patient records from the file. If the file exists, the program reads the data and populates the `patients[]` array accordingly.

5. Doctor Assignment

```
void displayDoctorAssignments(struct Doctor doctors[], int doctorCount,
struct Patient patients[], int patientCount) {
    printf("\nDoctor Assignments:\n");
    for (int i = 0; i < doctorCount; i++) {
        printf("Doctor: %s, Specialty: %s\n", doctors[i].name,
doctors[i].specialty);
        for (int j = 0; j < patientCount; j++) {
            if (patients[j].id == (i + 1)) {  // Assigning patients based
on ID for simplicity
                printf("  Patient: %s, Disease: %s\n", patients[j].name,
patients[j].disease);
            }
        }
    }
}
```

- **Purpose:** This function displays the doctor assignments by iterating through all doctors and patients. For simplicity, patients are assigned to doctors based on their IDs.

6. Main Menu and Operations

The program operates via a simple menu-driven interface. The user can:

1. **Manage Patient Records**: Add, remove, or display patient details.
2. **Manage Doctor Assignments**: Add, remove, or display doctor assignments.
3. **Manage Waiting Queue**: Add, remove, or display the waiting queue of patients.
4. **Exit**: Close the program.

Each menu option triggers the corresponding function to perform the necessary operation.

Conclusion

This program demonstrates how to use **arrays of structures**, **queues**, and **hash maps** to build a robust patient management system. The use of file operations ensures data persistence even after the program is closed. This makes the system efficient, reliable, and capable of managing large amounts of patient data.