Assignment No: 02

Department of Computer Science
Iqra University Islamabad

Data Structures and Algorithms

Maqsood Ahmed
ID: 38186

Problem # 1: [CLO2]

A train system uses different types of linked lists for different functionalities:

Singly linked list: Stores the list of stations a train passes through.

Doubly linked list: Manages train compartments, allowing the addition/removal of compartments in both directions.

Circular linked list: Tracks trains running in a loop between two cities.

Design and implement a program to:

1. Add/remove stations in the route.

2. Add/remove compartments to the train.

3. Manage the circular route of trains.

Source Code:

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

// singly linked list
class Station {
    public:
        string name;
        Station* next;

        Station(string name) {
            this->name = name;
            next = nullptr;
        }
};

// doubly Linked List
class Compartment{
    public:
        string name;
        Compartment* prev;
        Compartment* next;

        Compartment(string name) {
            this->name = name;
            prev = nullptr;
            next = nullptr;
        }
};

// circular linked list
class Train {
    public:
```

```
        string name;
        Train* next;

        Train(string name) {
            this->name = name;
            next = nullptr;
        }
};

// Stationlist class. it stores the list of stations a train passes through
class StationList {
    public:
        // constructor
        StationList(): head_station_list(nullptr) {}
        // function to add the station name in the station list
        void addStation(string station_name) {
            Station* newStation = new Station(station_name);

            // if the stationList is empty just assign the newStation to head_station_list
            if(head_station_list == nullptr) {
                head_station_list = newStation;
                cout << "Station added: " << newStation->name << '\n';
                return;
            }
            // if the stationList is not empty then we add the newStation name at the end of the list
            Station* temp_ptr = head_station_list;

            while(temp_ptr->next != nullptr) {
                temp_ptr = temp_ptr->next;
            }

            temp_ptr->next = newStation;
            cout << "Station added: " << newStation->name << '\n';
        }

        // removes the station in the station list
        void removeStation(string name) {
            // check if the station list is empty or not
            if(head_station_list == nullptr) {
                cout << "Station list is empty!\n";
                return;
            }

            Station* curr = head_station_list;
            Station* prev = nullptr;
            while(curr != nullptr && curr->name != name) {
                prev = curr;
                curr = curr->next;
            }

            if(curr == nullptr) {
                cout << "Station not found!\n";
```

```
            return;
        }

        // removes the station from the list
        prev->next = curr->next;
        delete curr;
        cout << "Station removed Successfully!\n";
    }

    // displays the stations
    void displayStations() {
        Station* temp_ptr = head_station_list;

        while(temp_ptr != nullptr) {
            cout << temp_ptr->name << " -> ";
            temp_ptr = temp_ptr->next;
        }
        cout << "NULL\n";
    }

  private:
    Station* head_station_list;

};

// Manages the train compartments
class CompartmentList {
  public:
  // constructor
  CompartmentList(): head_compartment_list(nullptr) {}

  // adds the compartment
  void addCompartment(string compartment_name) {
    // if the compartment list is empty
    Compartment* newCompartment = new Compartment(compartment_name);
    if(head_compartment_list == nullptr) {
        head_compartment_list = newCompartment;
        cout << "Compartment added: " << newCompartment->name << '\n';
        return;
    }

    Compartment* temp_ptr = head_compartment_list;

    while(temp_ptr->next != nullptr) {
        temp_ptr = temp_ptr->next;
    }

    temp_ptr->next = newCompartment; // added new compartment in the list
    newCompartment->prev = temp_ptr;
    cout << "Compartment added: " << newCompartment->name << '\n';
```

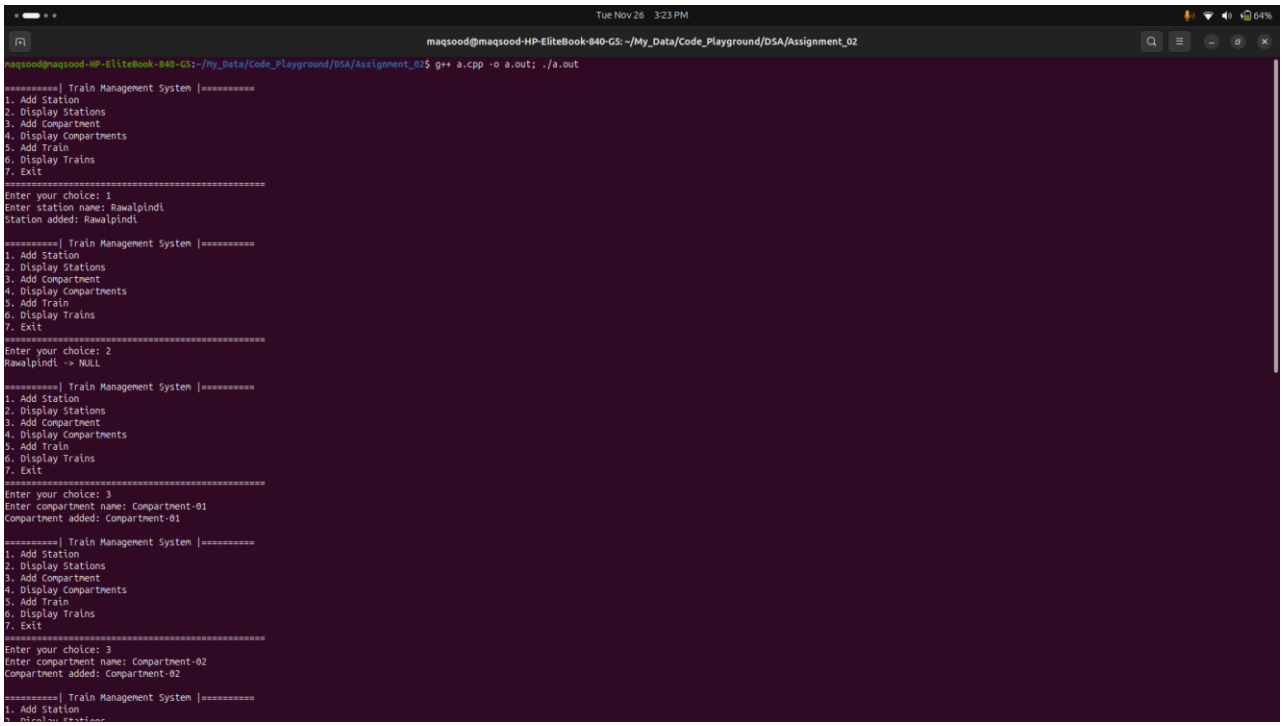
```
}

// removes the compartment
void removeCompartment(string compartment_name) {
    // if the compartment list is empty
    if(head_compartment_list == nullptr) {
        cout << "Compartment added: " << compartment_name << '\n';
        cout << compartment_name << " is not in the Compartment list!\n";
        return;
    }



    // if compartment_name matches with first Compartment
    if(head_compartment_list->name == compartment_name) {
        Compartment* compartmentToDelete = head_compartment_list;
        head_compartment_list = head_compartment_list->next;
        if(head_compartment_list != nullptr) head_compartment_list->prev = nullptr;
        // delete the compartment
        delete compartmentToDelete;
        cout << "Compartment removed: " << compartment_name << '\n';
        return;
    }

    // if the compartment is somewhere in the middle or in the end
    Compartment* temp_ptr = head_compartment_list;


    while(temp_ptr != nullptr && temp_ptr->name != compartment_name) {
        temp_ptr = temp_ptr->next;
    }


    // temp_ptr is nullptr it means compartment is not in the compartment_list
    if(temp_ptr == nullptr) {
        cout << compartment_name << " is not in the Compartment list!\n";
        return;
    }


    // if compartment found in the compartment list
    if(temp_ptr->prev != nullptr)
        temp_ptr->prev->next = temp_ptr->next;
    if(temp_ptr->next != nullptr)
        temp_ptr->next->prev = temp_ptr->prev;
    // delete the compartment
    delete temp_ptr;
    cout << "Compartment removed: " << compartment_name << '\n';


}


// displays the all compartments
void displayCompartments() {
    Compartment* temp_ptr = head_compartment_list;
```

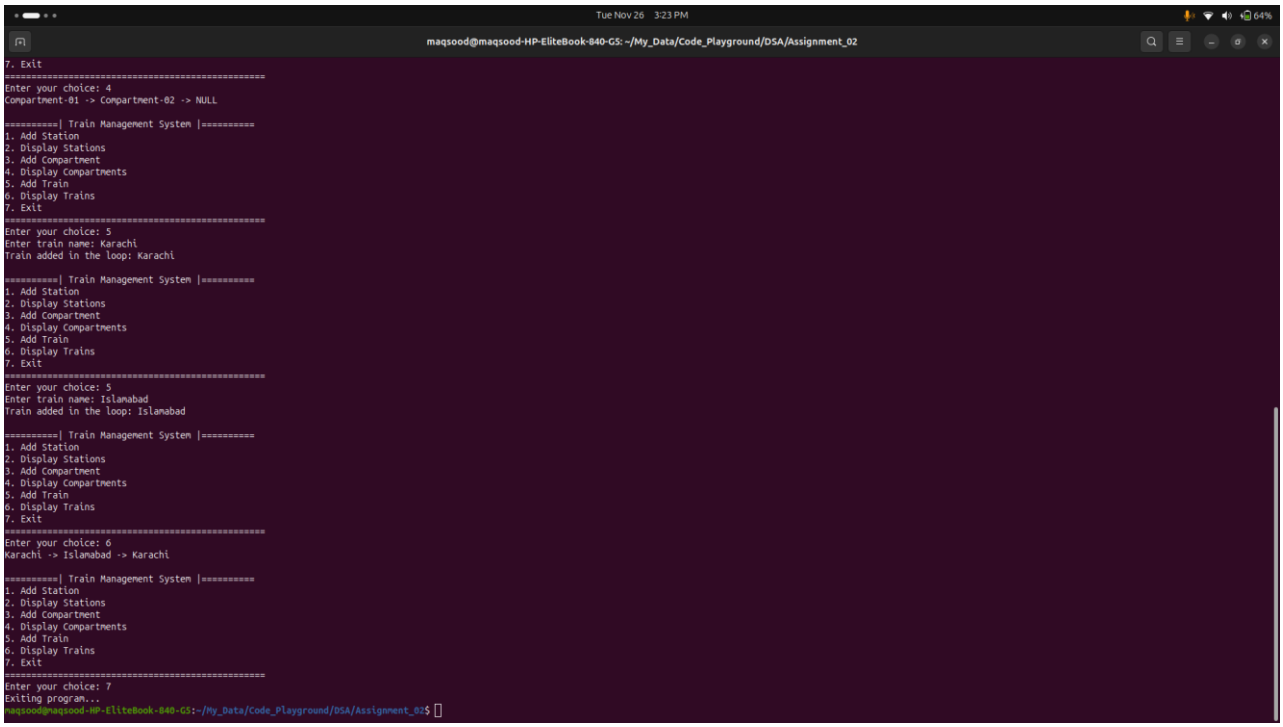
```
        while(temp_ptr != nullptr) {
            cout << temp_ptr->name << " -> ";
            temp_ptr = temp_ptr->next;
        }
        cout << "NULL\n";
    }

    private:
        Compartment* head_compartment_list;
};



// circular linked list for Train loops
class CircularRoute {
    private:
        Train* head_train_list;

    public:

        // adds the Train in the loop
        void addTrain(string train_name) {
            Train* newTrain = new Train(train_name);

            // if there's no train in the cirular loop
            if(head_train_list == nullptr) {
                head_train_list = newTrain;
                head_train_list->next = head_train_list;
                cout << "Train added in the loop: " << train_name << '\n';
                return;
            }

            // if there's already trains in the loop
            Train* temp_ptr = head_train_list;

            while(temp_ptr->next != head_train_list) {
                temp_ptr = temp_ptr->next;
            }

            temp_ptr->next = newTrain;
            newTrain->next = head_train_list;
            cout << "Train added in the loop: " << train_name << '\n';
        }

        // displays the all Trains
        void displayTrains() {
            if(head_train_list == nullptr) {
                cout << "No trains in the loop\n";
                return;
            }

            Train* temp_ptr = head_train_list;
```

```
        do {
            cout << temp_ptr->name << " -> ";
            temp_ptr = temp_ptr->next;
        } while (temp_ptr != head_train_list);
        cout << head_train_list->name << '\n';
    }
};



void displayMenu() {
    cout << "\n==========| Train Management System |==========\n";
    cout << "1. Add Station\n";
    cout << "2. Display Stations\n";
    cout << "3. Add Compartment\n";
    cout << "4. Display Compartments\n";
    cout << "5. Add Train\n";
    cout << "6. Display Trains\n";
    cout << "7. Exit\n";
    cout << "================================================\n";
    cout << "Enter your choice: ";
}

int main() {
    StationList* stationList = new StationList();
    CompartmentList* compartmentList = new CompartmentList();
    CircularRoute* circularRoute = new CircularRoute();


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


        switch (choice) {
        case 1: {
            string station;
            cout << "Enter station name: ";
            cin.ignore();
            getline(cin, station);
            stationList->addStation(station);
            break;
        }
        case 2:
            stationList->displayStations();
            break;
        case 3: {
            string compartment;
            cout << "Enter compartment name: ";
            cin.ignore();
            getline(cin, compartment);
            compartmentList->addCompartment(compartment);
            break;
        }
```

```
        case 4:
            compartmentList->displayCompartments();
            break;
        case 5: {
            string train;
            cout << "Enter train name: ";
            cin.ignore();
            getline(cin, train);
            circularRoute->addTrain(train);
            break;
        }
        case 6:
            circularRoute->displayTrains();
            break;
        case 7:
            cout << "Exiting program...\n";
            break;
        default:
            cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != 7);

    delete stationList;
    delete compartmentList;
    delete circularRoute;

    return EXIT_SUCCESS;
}
```

OUTPUT:





Problem # 2: [CLO2]

In an online gaming leaderboard, user scores are stored in a linked list. Write a function to remove duplicate scores from the list while maintaining the order of the scores.

Source Code:

```
#include <iostream>
using namespace std;
class Node {
    public:
        int score;
        Node* next;

        Node(int new_score) {
            this->score = new_score;
            this->next = nullptr;
        }
};

void removeDuplicatesScores(Node* &leaderboard_score_list) {
```

```
    // If the leaderboard list is empty or has one score, no need to remove duplicates
    if(leaderboard_score_list == nullptr || leaderboard_score_list->next == nullptr) return;

    Node* temp = leaderboard_score_list;
    while(temp != nullptr) {
        Node* curr = temp->next;
        Node* prev = temp;

        while(curr != nullptr) {
            if(curr->score == temp->score) {
                // Duplicate found, remove it
                prev->next = curr->next;
                delete curr;
                curr = prev->next; // move to the next node
            } else {
                prev = curr;
                curr = curr->next;
            }
        }
        temp = temp->next; // Move temp forward to check for the next unique score
    }

    cout << "Duplicate scores removed successfully!\n";
}

void addScore(Node* &leaderboard_score_list, int score) {
    Node* newScore = new Node(score);
    // if the scorelist is empty
    if(leaderboard_score_list == nullptr) {
        leaderboard_score_list = newScore;
        cout << "Score added: " << score << '\n';
        return;
    }
```

```
    Node* temp_ptr = leaderboard_score_list;
    while (temp_ptr->next != nullptr) {
        temp_ptr = temp_ptr->next;
    }

    temp_ptr->next = newScore;
    cout << "Score added: " << score << '\n';
}

void displayLeaderboardScores(Node* leaderboard) {
    cout << "\n";
    while(leaderboard != nullptr) {
        cout << leaderboard->score << " -> ";
        leaderboard = leaderboard->next;
    } cout << "NULL\n";
}

int main(void) {
    Node* leaderboard = nullptr;
    int choice;
    int score;

    do {
        cout << "\n=========| Leaderboard Scores |=========\n\n";
        cout << "1 - Add Score to leaderboard\n";
        cout << "2 - Remove duplicate scores\n";
        cout << "3 - Exit\n";
        cout << "\n=======================================\n";
        cout << "Enter choice:  ";
        cin >> choice;

        switch (choice)
        {
        case 1:
```

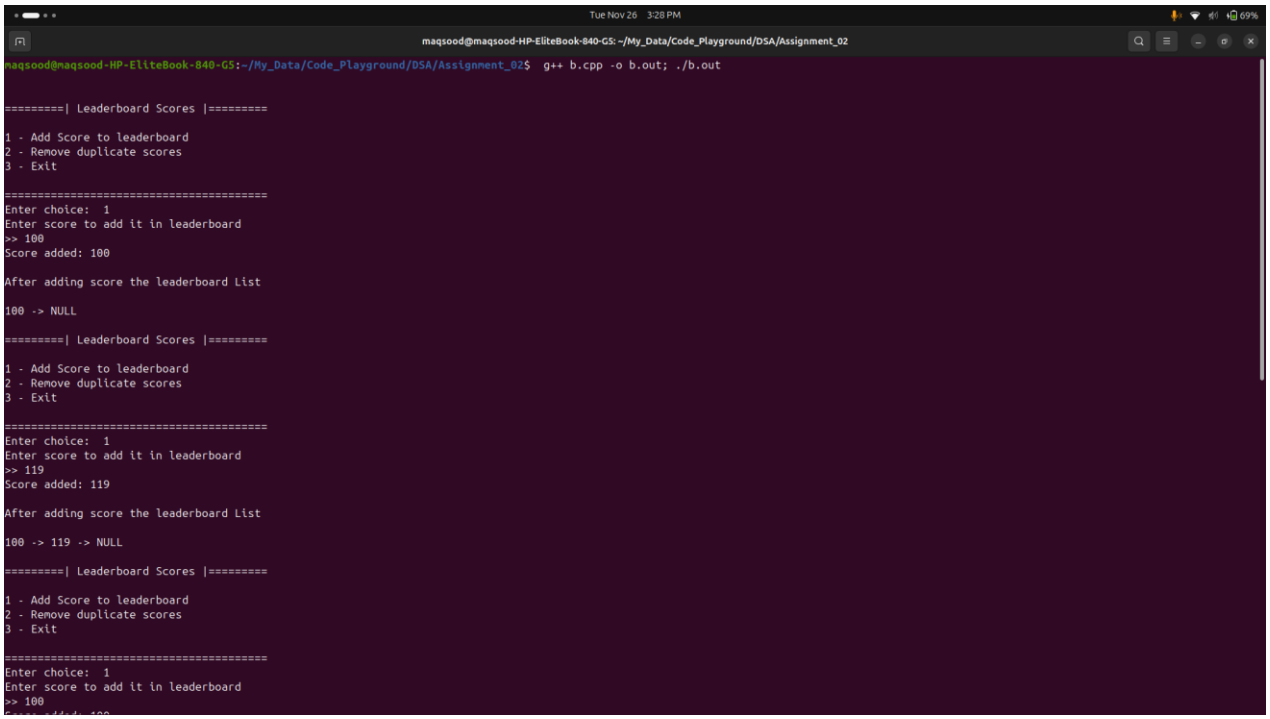
```
            cout << "Enter score to add it in leaderboard\n>> ";
            cin >> score;
            addScore(leaderboard, score);
            cout << "\nAfter adding score the leaderboard List\n";
            displayLeaderboardScores(leaderboard);
            break;
        case 2:
            removeDuplicatesScores(leaderboard);
            cout << "\nAfter removing duplicate scores in the leaderboard List\n";
            displayLeaderboardScores(leaderboard);
            break;
        case 3:
            cout << "Exiting program...\n";
            break;
  #include <iostream>
using namespace std;

// class for linked list
class Node {
    public:
        int score;
        Node* next;

        Node(int new_score) {
            this->score = new_score;
            this->next = nullptr;
        }
};

void removeDuplicatesScores(Node* &leaderboard_score_list) {
    // If the leaderboard list is empty or has one score, no need to remove duplicates
    if(leaderboard_score_list == nullptr || leaderboard_score_list->next == nullptr) return;

    Node* temp = leaderboard_score_list;
```

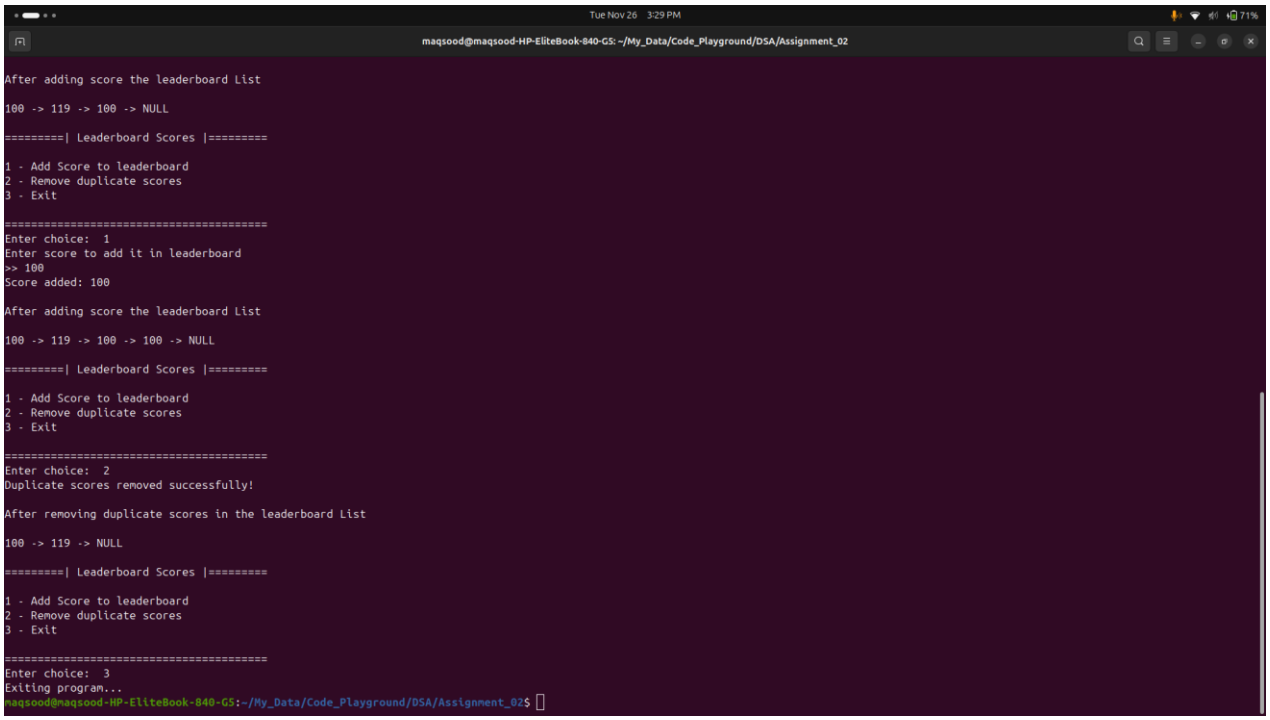
```
    while(temp != nullptr) {
        Node* curr = temp->next;
        Node* prev = temp;

        while(curr != nullptr) {
            if(curr->score == temp->score) {
                // Duplicate found, remove it
                prev->next = curr->next;
                delete curr;
                curr = prev->next; // move to the next node
            } else {
                prev = curr;
                curr = curr->next;
            }
        }
        temp = temp->next; // Move temp forward to check for the next unique score
    }

    cout << "Duplicate scores removed successfully!\n";
}

void addScore(Node* &leaderboard_score_list, int score) {
    Node* newScore = new Node(score);
    // if the scorelist is empty
    if(leaderboard_score_list == nullptr) {
        leaderboard_score_list = newScore;
        cout << "Score added: " << score << '\n';
        return;
    }

    Node* temp_ptr = leaderboard_score_list;
    while (temp_ptr->next != nullptr) {
        temp_ptr = temp_ptr->next;
    }
```

```
    temp_ptr->next = newScore;
    cout << "Score added: " << score << '\n';
}

void displayLeaderboardScores(Node* leaderboard) {
    cout << "\n";
    while(leaderboard != nullptr) {
        cout << leaderboard->score << " -> ";
        leaderboard = leaderboard->next;
    } cout << "NULL\n";
}

int main(void) {
    Node* leaderboard = nullptr;
    int choice;
    int score;

    do {
        cout << "\n=========| Leaderboard Scores |=========\n\n";
        cout << "1 - Add Score to leaderboard\n";
        cout << "2 - Remove duplicate scores\n";
        cout << "3 - Exit\n";
        cout << "\n=========================================\n";
        cout << "Enter choice:  ";
        cin >> choice;

        switch (choice)
        {
        case 1:
            cout << "Enter score to add it in leaderboard\n>> ";
            cin >> score;
            addScore(leaderboard, score);
            cout << "\nAfter adding score the leaderboard List\n";
```

```
            displayLeaderboardScores(leaderboard);
            break;
        case 2:
            removeDuplicatesScores(leaderboard);
            cout << "\nAfter removing duplicate scores in the leaderboard List\n";
            displayLeaderboardScores(leaderboard);
            break;
        case 3:
            cout << "Exiting program...\n";
            break;
        default:
            cout << "Invalid choice!\n";
            break;
        }
    } while(choice != 3);

    delete leaderboard;

    return EXIT_SUCCESS;
}     default:
            cout << "Invalid choice!\n";
            break;
        }
    } while(choice != 3);

    delete leaderboard;

    return EXIT_SUCCESS;
}
```

OUTPUT:


Tue Nov 26 3:28 PM
maqsood@maqsood-HP-EliteBook-840-G5: ~/My_Data/Code_Playground/DSA/Assignment_02
maqsood@maqsood-HP-EliteBook-840-G5:~/My_Data/Code_Playground/DSA/Assignment_02$ g++ b.cpp -o b.out; ./b.out
=========| Leaderboard Scores |=========
1 - Add Score to leaderboard
2 - Remove duplicate scores
3 - Exit
==
Enter choice: 1
Enter score to add it in leaderboard
>> 100
Score added: 100
After adding score the leaderboard List
100 -> NULL
=========| Leaderboard Scores |=========
1 - Add Score to leaderboard
2 - Remove duplicate scores
3 - Exit
==
Enter choice: 1
Enter score to add it in leaderboard
>> 119
Score added: 119
After adding score the leaderboard List
100 -> 119 -> NULL
=========| Leaderboard Scores |=========
1 - Add Score to leaderboard
2 - Remove duplicate scores
3 - Exit
==
Enter choice: 1
Enter score to add it in leaderboard
>> 100


Tue Nov 26 3:29 PM
maqsood@maqsood-HP-EliteBook-840-G5: ~/My_Data/Code_Playground/DSA/Assignment_02
After adding score the leaderboard List
100 -> 119 -> 100 -> NULL
=========| Leaderboard Scores |=========
1 - Add Score to leaderboard
2 - Remove duplicate scores
3 - Exit
==
Enter choice: 1
Enter score to add it in leaderboard
>> 100
Score added: 100
After adding score the leaderboard List
100 -> 119 -> 100 -> 100 -> NULL
=========| Leaderboard Scores |=========
1 - Add Score to leaderboard
2 - Remove duplicate scores
3 - Exit
==
Enter choice: 2
Duplicate scores removed successfully!
After removing duplicate scores in the leaderboard List
100 -> 119 -> NULL
=========| Leaderboard Scores |=========
1 - Add Score to leaderboard
2 - Remove duplicate scores
3 - Exit
==
Enter choice: 3
Exiting program...
maqsood@maqsood-HP-EliteBook-840-G5:~/My_Data/Code_Playground/DSA/Assignment_02$

Problem # 3: [CLO2]

An operating system schedules processes in a round-robin manner using a circular linked list.

Each process is represented as a node. Design and implement:

1. Add a process to the schedule.

2. Execute a process (move to the next one).

3. Remove a completed process from the schedule.

Source Code:

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

class Process {
public:
    int pid;
    Process* next;
    Process(int new_pid) {
        this->pid = new_pid;
        this->next = nullptr;
    }
};

class RoundRobinScheduler {
public:
    RoundRobinScheduler() : head(nullptr), tail(nullptr), current(nullptr) {}

    // Add a process to the schedule
    void addProcess(int new_pid) {
        Process* newProcess = new Process(new_pid);
        if (head == nullptr) {
            head = tail = current = newProcess;
            newProcess->next = newProcess; // Circular link
        } else {
            newProcess->next = head;
```

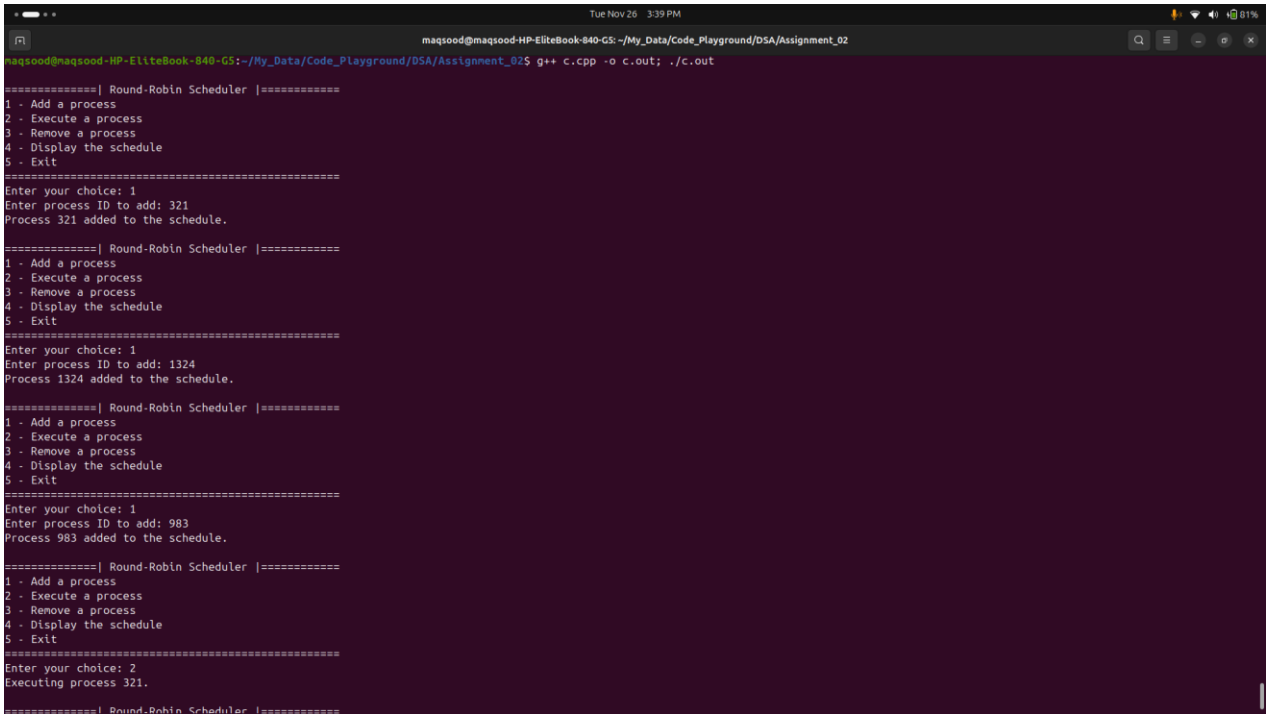
```
        tail->next = newProcess;
        tail = newProcess;
    }
    cout << "Process " << new_pid << " added to the schedule.\n";
}

// Execute the current process
void executeProcess() {
    if (current == nullptr) {
        cout << "No processes in the schedule!\n";
        return;
    }
    cout << "Executing process " << current->pid << ".\n";
    current = current->next; // Move to the next process
}
// Remove a process from the schedule
void removeProcess(int pidToRemove) {
    if (head == nullptr) {
        cout << "No processes in the schedule!\n";
        return;
    }
    Process* temp = head;
    Process* prev = nullptr;
    // If there's only one process
    if (head == tail && head->pid == pidToRemove) {
        delete head;
        head = tail = current = nullptr;
        cout << "Process " << pidToRemove << " removed from the schedule.\n";
        return;
    }

    // Traverse the circular linked list to find the process
    do {
        if (temp->pid == pidToRemove) {
```

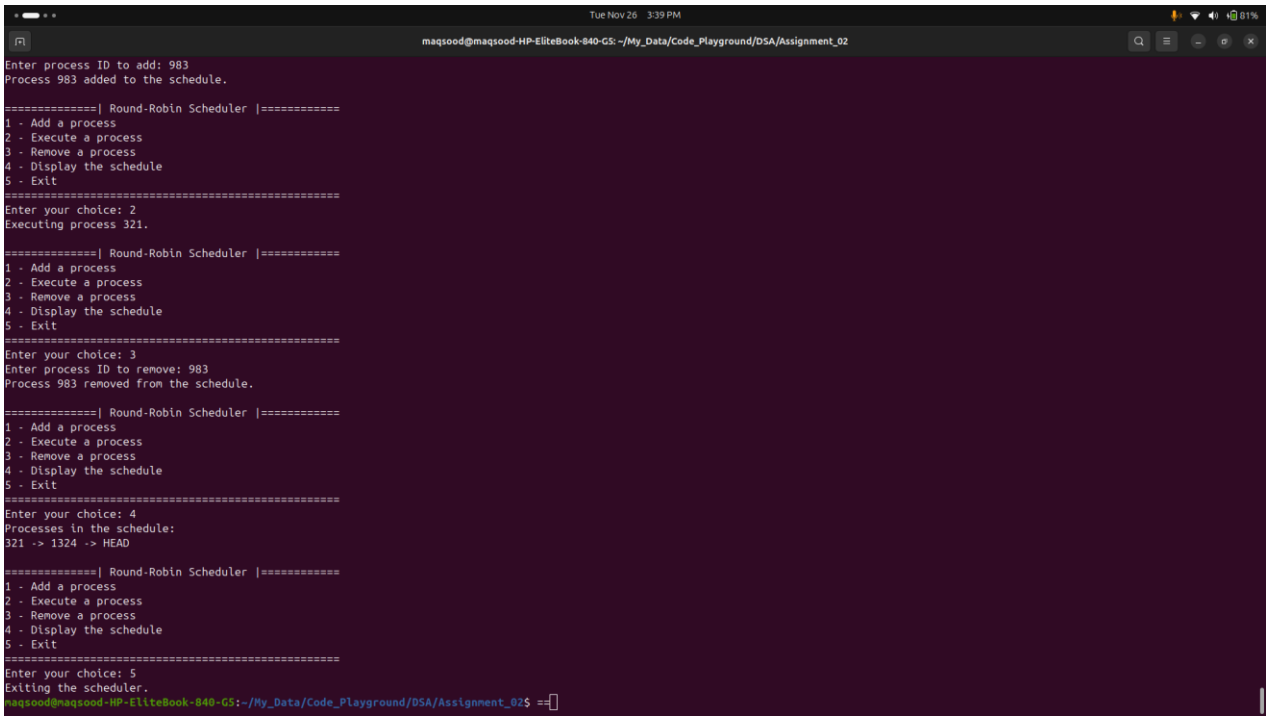
```
            if (temp == head) {
                head = head->next; // Move head to the next process
                tail->next = head; // Maintain circularity
            } else if (temp == tail) {
                tail = prev; // Update tail
                tail->next = head; // Maintain circularity
            } else {
                prev->next = temp->next; // Bypass the current process
            }
            if (current == temp) {
                current = temp->next; // Update current
            }
            delete temp;
            cout << "Process " << pidToRemove << " removed from the schedule.\n";
            return;
        }
        prev = temp;
        temp = temp->next;
    } while (temp != head);

    cout << "Process " << pidToRemove << " not found in the schedule.\n";
}

// Display the schedule
void displaySchedule() {
    if (head == nullptr) {
        cout << "No processes in the schedule!\n";
        return;
    }

    Process* temp = head;
    cout << "Processes in the schedule:\n";
    do {
        cout << temp->pid << " -> ";
```

```
            temp = temp->next;
        } while (temp != head);
        cout << "HEAD\n";
    }
    ~RoundRobinScheduler() {
        if (head == nullptr) return;
        Process* temp = head;
        do {
            Process* toDelete = temp;
            temp = temp->next;
            delete toDelete;
        } while (temp != head);
        head = tail = current = nullptr;
    }

private:
    Process* head;    // Head of the circular linked list
    Process* tail;    // Tail of the circular linked list
    Process* current; // Current scheduled process
};

int main() {
    RoundRobinScheduler scheduler;
    int choice, pid;

    do {
        cout << "\n==============| Round-Robin Scheduler |============\n";
        cout << "1 - Add a process\n";
        cout << "2 - Execute a process\n";
        cout << "3 - Remove a process\n";
        cout << "4 - Display the schedule\n";
        cout << "5 - Exit\n";
        cout << "================================================\n";
        cout << "Enter your choice: ";
```

```
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Enter process ID to add: ";
            cin >> pid;
            scheduler.addProcess(pid);
            break;
        case 2:
            scheduler.executeProcess();
            break;
        case 3:
            cout << "Enter process ID to remove: ";
            cin >> pid;
            scheduler.removeProcess(pid);
            break;
        case 4:
            scheduler.displaySchedule();
            break;
        case 5:
            cout << "Exiting the scheduler.\n";
            break;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    } while (choice != 5);

    return 0;
}
```

OUTPUT:


Tue Nov 26 3:39 PM
maqsood@maqsood-HP-EliteBook-840-G5: ~/My_Data/Code_Playground/DSA/Assignment_02
maqsood@maqsood-HP-EliteBook-840-G5:~/My_Data/Code_Playground/DSA/Assignment_02$ g++ c.cpp -o c.out; ./c.out
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 1
Enter process ID to add: 321
Process 321 added to the schedule.
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 1
Enter process ID to add: 1324
Process 1324 added to the schedule.
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 1
Enter process ID to add: 983
Process 983 added to the schedule.
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 2
Executing process 321.


Tue Nov 26 3:39 PM
maqsood@maqsood-HP-EliteBook-840-G5: ~/My_Data/Code_Playground/DSA/Assignment_02
Enter process ID to add: 983
Process 983 added to the schedule.
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 2
Executing process 321.
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 3
Enter process ID to remove: 983
Process 983 removed from the schedule.
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 4
Processes in the schedule:
321 -> 1324 -> HEAD
==========| Round-Robin Scheduler |==========
1 - Add a process
2 - Execute a process
3 - Remove a process
4 - Display the schedule
5 - Exit
==========
Enter your choice: 5
Exiting the scheduler.
maqsood@maqsood-HP-EliteBook-840-G5:~/My_Data/Code_Playground/DSA/Assignment_02$ ==

Problem # 4: [CLO2]

Design and implement a simple a Music App that maintains a playlist using a doubly linked list.

- **Add functionality to:**
- **Play the next song.**
- **Play the previous song.**
- **Add a song to the playlist.**
- **Remove a song from the playlist.**

Source Code:

```
#include <iostream>
#include <string>
using namespace std;
class Song {
public:
    string title;
    Song* next;
    Song* prev;

    Song(string songTitle) : title(songTitle), next(nullptr), prev(nullptr) {}
};

class Playlist {
public:
    Playlist() : head(nullptr), tail(nullptr), current(nullptr) {}

    // Add a song to the playlist
    void addSong(string songTitle) {
        Song* newSong = new Song(songTitle);
        if (head == nullptr) {
            head = tail = current = newSong;
        } else {
            tail->next = newSong;
```

```
        newSong->prev = tail;
        tail = newSong;
    }
    cout << "Song \"" << songTitle << "\" added to the playlist.\n";
}

// Remove a song from the playlist
void removeSong(string songTitle) {
    if (head == nullptr) {
        cout << "Playlist is empty! Cannot remove any song.\n";
        return;
    }

    Song* temp = head;

    // Traverse the playlist to find the song
    while (temp != nullptr && temp->title != songTitle) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Song \"" << songTitle << "\" not found in the playlist.\n";
        return;
    }

    // Update links to remove the song
    if (temp == head) {
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
    } else if (temp == tail) {
```

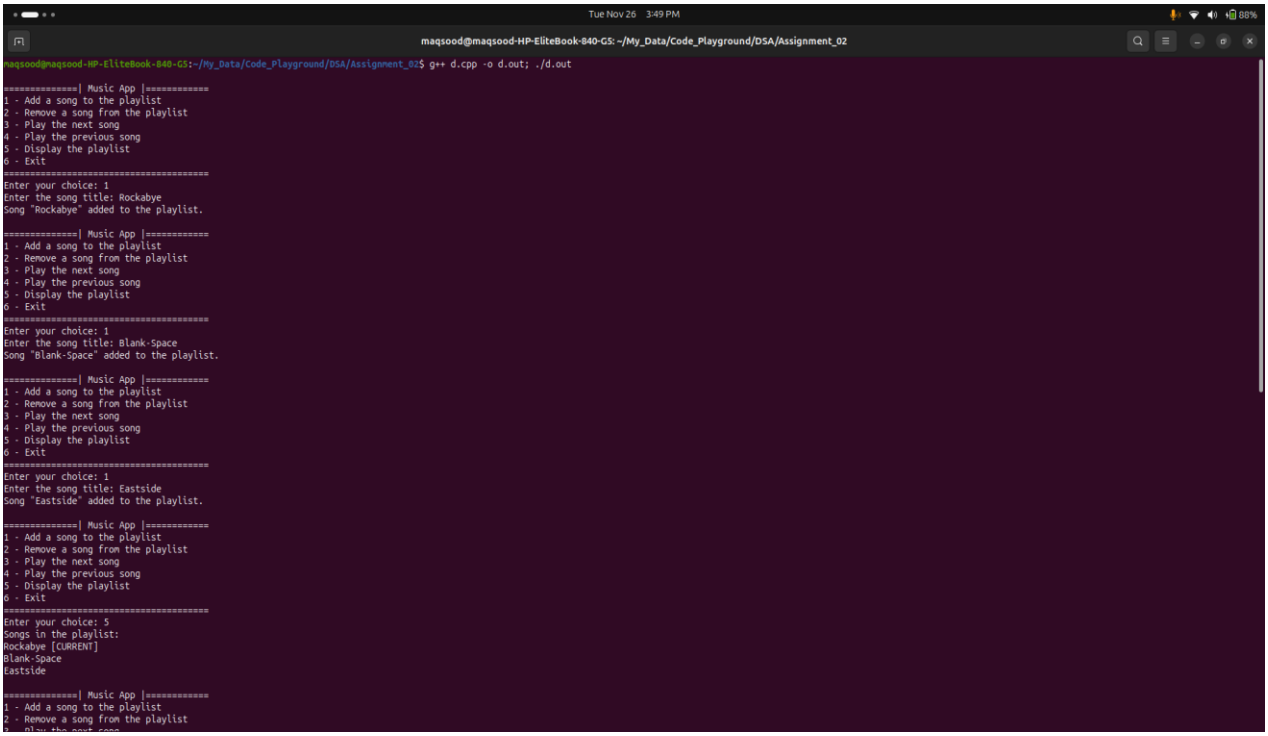
```
        tail = tail->prev;
        if (tail != nullptr) {
            tail->next = nullptr;
        }
    } else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }

    // Update the current pointer if necessary
    if (current == temp) {
        current = temp->next != nullptr ? temp->next : head; // Move to next or reset
    }

    delete temp;
    cout << "Song \"" << songTitle << "\" removed from the playlist.\n";

    if (head == nullptr) {
        tail = current = nullptr; // Reset pointers if the playlist is empty
    }
}

// Play the next song
void playNext() {
    if (current == nullptr) {
        cout << "Playlist is empty! No song to play.\n";
        return;
    }
    current = current->next != nullptr ? current->next : head; // Loop to the start
    cout << "Now playing: \"" << current->title << "\"\n";
}
```

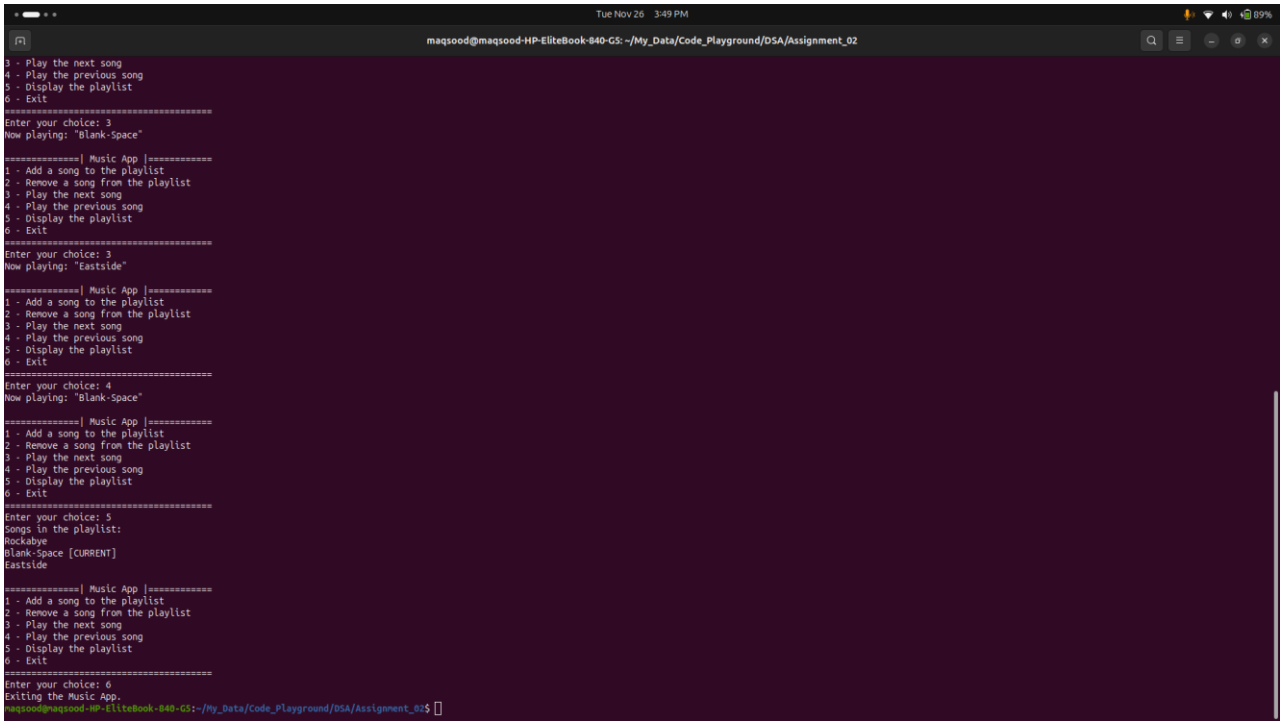
```
// Play the previous song
void playPrevious() {
    if (current == nullptr) {
        cout << "Playlist is empty! No song to play.\n";
        return;
    }
    current = current->prev != nullptr ? current->prev : tail; // Loop to the end
    cout << "Now playing: \"" << current->title << "\"\n";
}

// Display the playlist
void displayPlaylist() {
    if (head == nullptr) {
        cout << "Playlist is empty!\n";
        return;
    }

    cout << "Songs in the playlist:\n";
    Song* temp = head;
    while (temp != nullptr) {
        cout << temp->title << (temp == current ? " [CURRENT]" : "") << "\n";
        temp = temp->next;
    }
}

// Destructor to clean up dynamically allocated memory
~Playlist() {
    while (head != nullptr) {
        Song* temp = head;
        head = head->next;
        delete temp;
    }
```

```
        tail = current = nullptr;
    }

private:
    Song* head;
    Song* tail;
    Song* current;
};

int main() {
    Playlist playlist;
    int choice;
    string songTitle;

    do {
        cout << "\n==============| Music App |============\n";
        cout << "1 - Add a song to the playlist\n";
        cout << "2 - Remove a song from the playlist\n";
        cout << "3 - Play the next song\n";
        cout << "4 - Play the previous song\n";
        cout << "5 - Display the playlist\n";
        cout << "6 - Exit\n";
        cout << "=======================================\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(); // To handle newline character after choice

        switch (choice) {
        case 1:
            cout << "Enter the song title: ";
            getline(cin, songTitle);
            playlist.addSong(songTitle);
```

```
            break;
        case 2:
            cout << "Enter the song title to remove: ";
            getline(cin, songTitle);
            playlist.removeSong(songTitle);
            break;
        case 3:
            playlist.playNext();
            break;
        case 4:
            playlist.playPrevious();
            break;
        case 5:
            playlist.displayPlaylist();
            break;
        case 6:
            cout << "Exiting the Music App.\n";
            break;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    } while (choice != 6);
    return 0;
}
```

OUTPUT:





The End