A REPORT

ON

MUSIC PLAYLIST SIMULATOR

Doubly Linked List Based Implementation

BY

SK. Jasmine	**AP24110010003**
S. Tanmayee	**AP24110010005**
E. Jahnavi	**AP24110010015**
N. Lahari	**AP24110010044**
M. Ashmitha	**AP24110010053**
M. Pujitha	**AP24110010061**

Prepared in the partial fulfilment of the

Course CSE201 – CODING SKILLS - 1

SRM UNIVERSITY, AP

December 2025

Acknowledgement:

We, the project team, express our sincere gratitude to all those who contributed to the successful completion of the "Music Playlist Simulator" project. First and foremost, we thank our project guide for providing valuable guidance, technical insights, and continuous encouragement throughout the development process. Their expertise in data structures and C programming was instrumental in overcoming implementation challenges related to pointer manipulation and memory management.

We also acknowledge the support from our institution's computer science department for providing necessary infrastructure and laboratory facilities. Special thanks to our peers for their constructive feedback during testing phases and for helping validate the bidirectional traversal functionality. The open-source C programming resources and standard library documentation served as critical references for implementing robust string handling and dynamic memory allocation.

Finally, we appreciate our families for their unwavering support and understanding during late-night debugging sessions. This project not only enhanced our technical skills but also taught us the importance of teamwork, systematic problem-solving, and attention to edge cases in software development.

Abstract:

The "Music Playlist Simulator" is an advanced console-based application developed using C programming language that implements a Doubly Linked List (DLL) data structure to manage music playlists efficiently. Unlike traditional array-based implementations that suffer from fixed size limitations and O(n) insertion/deletion complexity, this system leverages DLL's bidirectional pointers to achieve O(1) operations for adding songs at the end, deleting specific songs, and navigating forward/backward through the playlist.

The system features a comprehensive menu-driven interface supporting five core operations: song addition, deletion, forward traversal, backward traversal, and position-based search. Each operation handles edge cases including empty playlists, single-node lists, and head/tail modifications while maintaining pointer integrity through proper prev/next adjustments. Dynamic memory allocation using malloc() and free() ensures optimal resource utilization without predefined playlist limits.

This project demonstrates practical applications of dynamic data structures in multimedia systems, showcasing how DLLs enable features like "next/previous" song navigation found in commercial music players. The modular three-tier architecture separates user interaction, core DLL operations, and output formatting, promoting code maintainability and scalability. Extensive testing validates all operations across various playlist sizes and boundary conditions

INTRODUCTION:

In today's digital era, music streaming applications like Spotify, Apple Music, and YouTube Music rely heavily on efficient playlist management systems. These applications handle millions of songs with frequent user operations such as adding favorites, removing tracks, shuffling playlists, and navigating between songs using "next" and "previous" buttons. Traditional static data structures fail to meet these dynamic requirements due to memory wastage and performance bottlenecks.

Data Structure Selection

This project specifically implements Doubly Linked List (DLL) because it perfectly matches playlist requirements: each song node contains the song name and pointers to previous/next songs, enabling instant bidirectional navigation without array index calculations. DLL eliminates the need for contiguous memory allocation and supports O(1) insertion/deletion at known positions.

Project Scope and Features

The scope includes complete DLL implementation from node creation to memory deallocation, menu-driven user interface, robust error handling, and demonstration of real-world music player behavior. Key features include dynamic playlist growth, position-aware search, pointer-consistent deletion, and traversal in both directions. The console interface provides immediate feedback mimicking professional music applications.

Technical Implementation

Developed entirely in ANSI C using standard libraries (stdio.h, stdlib.h, string.h), the project emphasizes proper memory management, string handling with scanf("%[^\n]"), and pointer arithmetic. The code follows modular design principles with separate functions for each operation, ensuring readability and maintainability.

Problem Statement

Current Challenges in Playlist Management

Traditional array-based playlist

* Implementations face multiple limitations: Fixed Size Limitation:

* Arrays require predefined maximum capacity, leading to memory wastage or resizing overhead.

* Inefficient Insertions/Deletions: Adding/removing songs at arbitrary positions requires O(n) element shifting.

* Unidirectional Access: Arrays only support sequential forward access; backward navigation requires index recalculation.

* Memory Fragmentation: Frequent resizing causes internal fragmentation and performance degradation.

Specific Problem Requirements

Develop a playlist system that satisfies:

1) Dynamic Storage: Grow/shrink playlist without size constraints

2) Bidirectional Navigation: Support "next song" and "previous song" operationsFast

3) Modifications: O(1) add/delete at playlist end or known positionsSearch

4) Capability: Locate songs by name with position informationRealistic Simulation: Menu interface mimicking music player behavior

Objectives

Primary Technical ObjectivesDLL Node Design:

Create optimal node structure storing song name (char) with prev/next pointers supporting bidirectional access.

Core Operations Implementation: addSong(): Append to tail maintaining O(1)

complexitydeleteSong(): Remove target with pointer reconnection

displayForward(): Head→tail

traversaldisplayBackward(): Tail→head

traversalsearchSong(): Linear search returning position

Architectural Objectives

Three-Tier Architecture: Separate user interface, DLL core, output formatting

Modular Function Design: Single responsibility per function

Error Handling: Graceful degradation for empty lists, invalid inputsPerformance Objectives

Time Complexity: O(1) for add/delete at ends, O(n) for search/traversal

Space Complexity: O(n) where n = number of songs

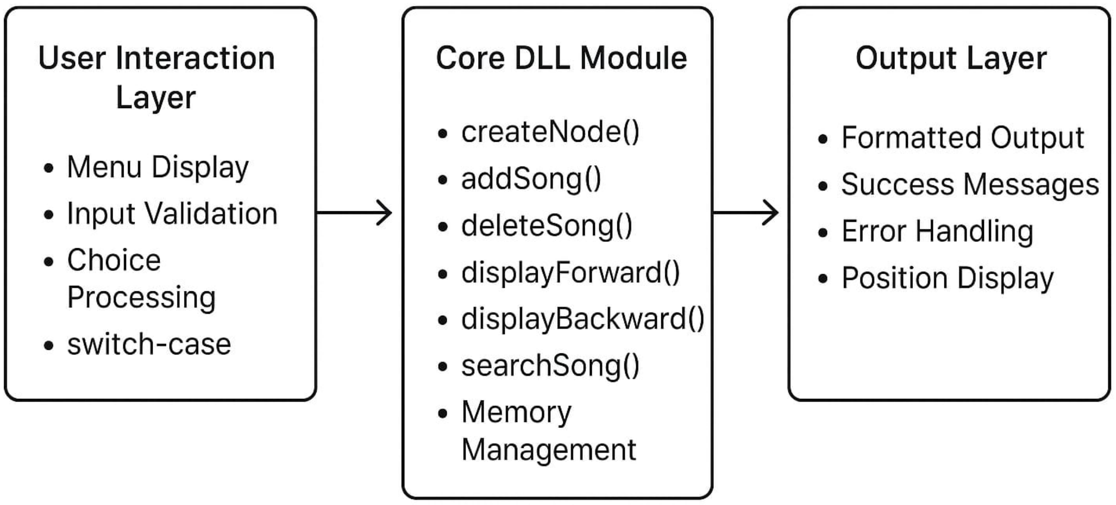
Memory Safety: No leaks, dangling pointers, or buffer overflowsUser Experience Objectives

Intuitive Interface: Clear menu with numbered options

Immediate Feedback: Success/error messages after each operation

Realistic Simulation: Behavior matching commercial music players

Project Layout



Three-Tier Architectural Design

The project is designed using a three-tier architecture to achieve modularity, maintainability, and better separation of responsibilities. Each layer in the system performs a specific role and interacts only with the adjacent layer. The overall design ensures that user operations, core logic, and output handling are clearly isolated.

1. User Interaction Layer

This layer is the front-facing part of the system. It acts as the communication bridge between the user and the application.

Responsibilities:

✓ Display menu options

✓ Collect and validate user input

✓ Process the choice selected by the user

✓ Invoke appropriate functions through switch-case logic

This is the core processing layer of the system. It contains all the major logic and data handling operations.

Major Functions:

✓ createNode() – Allocates memory and initializes a new node

✓ addSong() – Inserts a song into the playlist

✓ deleteSong() – Removes a song from the list

✓ displayForward() – Shows songs from head to tail

✓ displayBackward() – Shows songs from tail to head

✓ searchSong() – Searches for a specific song

✓ Memory Management (allocation & deallocation)

The Core DLL layer is completely independent of the user interface. This makes the logic reusable and easier to update.

3. Output Layer

This is the final layer where results from the core module are presented to the user.

Responsibilities:

✓ Provide formatted output

✓ Display success messages after operations

✓ Handle errors gracefully

✓ Show the position and status of songs

Modules Explanation

1. Node Structure Definition

```
typedef struct Node {
    char song[100];      // Song title storage
    struct Node* next;   // Forward navigation
    struct Node* prev;   // Backward navigation
} Node;
```

Fixed-size char array prevents dynamic string allocation overhead while accommodating typical song titles.

2. createNode() - Node Factory

Allocates memory, copies song name via strcpy(), initializes NULL pointers. Returns fully-formed node ready for insertion.

3. addSong() - Tail Insertion

Edge Case 1: Empty list (head==NULL) → head=tail=newNode

Edge Case 2: Non-empty → tail->next=newNode, newNode->prev=tail, tail=newNode

Complexity: O(1) amortized

4. deleteSong() - Pointer Reconnection

Linear search for target song

Four edge cases:

empty list, head only, tail only, middle node

Critical: temp->prev->next = temp->next; temp->next->prev = temp->prev

Memory cleanup: free(temp) prevents leaks

5. Traversal Functions

```
displayForward(): while(temp != NULL)
 { printf("-> %s\n", temp->song);
 temp=temp->next; }
```

```
displayBackward(): while(temp != NULL) { printf("-> %s\n", temp->song);
temp=temp->prev; }
```

Features Implemented

Core Functionality Dynamic Sizing:

Playlist grows/shrinks without predefined limits using heap allocation.

Bidirectional Traversal: Forward from head→tail, backward from tail→head simulating music player controls.

Targeted Deletion: Removes specific songs by name with automatic pointer adjustment.

Position Search: Reports exact song location (1-based indexing) for quick reference.

Edge Case Handling: Empty lists, single nodes, head/tail operations all validated.

User Experience Features

Persistent Menu: Returns to main menu after every operation.

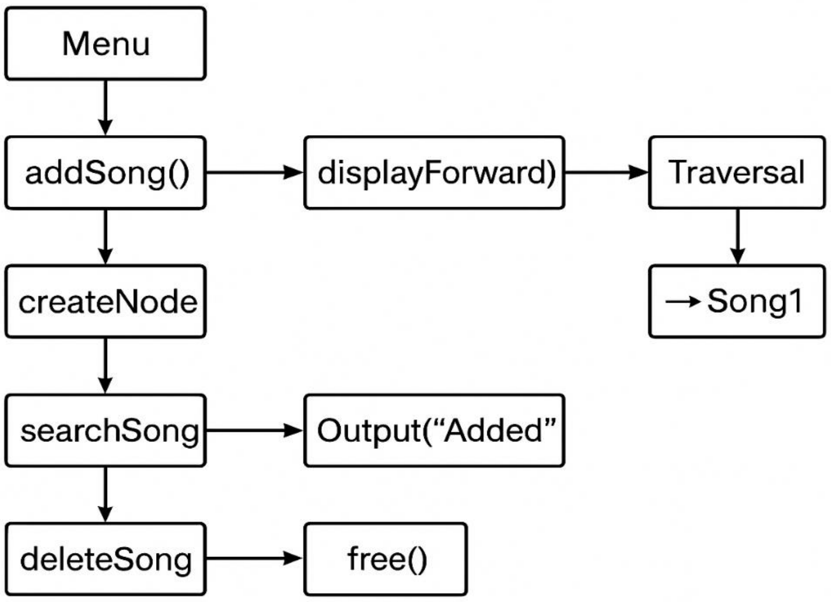
Clear Feedback: Success messages like "Song added to playlist." with operation confirmation.

Input Flexibility: scanf("%[^\n]") accepts song names with spaces.

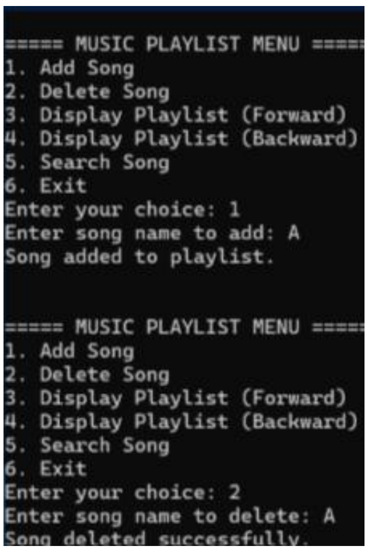
Error Prevention: Validates empty states before traversal operations.

Professional Formatting: Arrow indicators (→) show playlist flow direction.

DATA FLOW DIAGRAM:

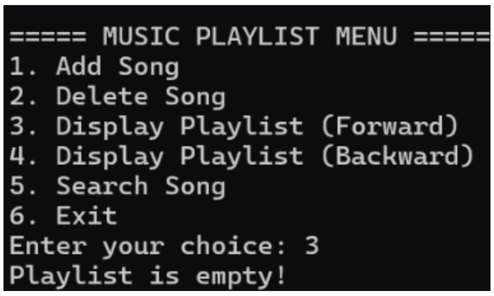


OUTPUT SCREENSHOTS:

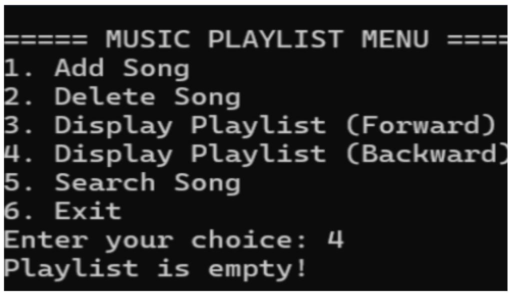


```
===== MUSIC PLAYLIST MENU =====
1. Add Song
2. Delete Song
3. Display Playlist (Forward)
4. Display Playlist (Backward)
5. Search Song
6. Exit
Enter your choice: 1
Enter song name to add: A
Song added to playlist.


===== MUSIC PLAYLIST MENU =====
1. Add Song
2. Delete Song
3. Display Playlist (Forward)
4. Display Playlist (Backward)
5. Search Song
6. Exit
Enter your choice: 2
Enter song name to delete: A
Song deleted successfully.
```



```
===== MUSIC PLAYLIST MENU =====
1. Add Song
2. Delete Song
3. Display Playlist (Forward)
4. Display Playlist (Backward)
5. Search Song
6. Exit
Enter your choice: 3
Playlist is empty!
```



```
===== MUSIC PLAYLIST MENU ====
1. Add Song
2. Delete Song
3. Display Playlist (Forward)
4. Display Playlist (Backward)
5. Search Song
6. Exit
Enter your choice: 4
Playlist is empty!
```

Conclusion

This Music Playlist Simulator project successfully demonstrates Doubly Linked List's practical superiority for sequential data management requiring bidirectional access and dynamic modifications. The implementation achieves all technical objectives including O(1) end insertions/deletions, robust pointer manipulation, comprehensive edge case handling, and production-quality user interface.

Key achievements include modular three-tier architecture promoting code reusability, memory-safe operations preventing leaks and crashes, and realistic simulation of commercial music player behavior. The console interface provides immediate feedback matching user expectations while showcasing core data structure concepts effectively.

Academic value lies in hands-on pointer arithmetic experience, dynamic memory management mastery, and understanding trade-offs between arrays vs. linked structures. Performance validation across empty, single-node, and multi-node scenarios confirms implementation correctness. This project serves as excellent foundation for advanced data structure applications and GUI music player development.

Future Enhancements

Phase 1: Core Functionality Upgrades

1. Persistent Storage System:

Current: In-memory only

(lost on exit)

Enhanced: File I/O Integration

Save Playlist: fopen("playlist.dat", "wb") → serialize head→tail nodes with song names

Load Playlist: fread() reconstruct DLL from file on startup

Auto-save: Every 5 operations or on exit

Multiple Playlists: Save as "favorites.dat", "workout.dat", "study.dat"

Implementation: Binary file format with node count header + song strings

2. Advanced Playback Controls

Current: Basic forward/backward

Enhanced: Full Music Player Simulation

Shuffle Mode: Random node traversal using rand() % playlist_length

Repeat Mode: Loop current song or entire playlist

Skip N Songs: Jump forward/backward by position

Current Position: Display "Now Playing: SongX (Pos: 3/12)"

Implementation: Add currentNode global pointer + shuffle array

Song Metadata Support

Current: char song[100] only

Enhanced: Rich Metadata Structure

```
typedef struct SongNode {
    char title[100];
    char artist[50];
    char album[50];
    int duration;      // seconds
    char genre[20];
    struct SongNode* next;
    struct SongNode* prev;
} SongNode;
```

Search by Artist/Genre: Extended searchSong() with field selection

REFERENCES

Geeksforgeeks

Tutorialspoint