SYNOPSIS REPORT

TOPIC - PLAYLIST MANGER

Playlist Manager Using Singly Linked List

This document outlines the development of a Playlist Manager using a Singly Linked List data structure. The project aims to create an efficient system for managing music playlists dynamically, allowing users to add, remove, and traverse songs seamlessly. This project serves as a practical exercise in data structures and algorithm design, demonstrating the advantages of linked lists for dynamic data management.

Introduction to Playlist Management

In today's digital landscape, music streaming applications are integral to entertainment. Efficient playlist management ensures a seamless and enjoyable user experience. This project focuses on developing a Playlist Manager using a Singly Linked List, which enables dynamic song addition, removal, and traversal. Unlike array-based storage, a linked list provides dynamic memory management and efficient modifications, making it ideal for playlist management.

A Singly Linked List consists of nodes, where each node contains a song and a pointer to the next node in the sequence. This structure allows for dynamic memory allocation, meaning that memory is allocated only when needed, and deallocated when not, optimizing memory usage. The ability to efficiently add or remove songs without needing to shift elements, as would be required in an array-based approach, makes a linked list a superior choice for this application.

This project serves as a foundational learning experience in data structures, memory management, and real-world application development. Students will gain hands-on experience in implementing linked lists and applying them to solve a practical problem.

Objectives of the Project

- Efficiently manage a playlist using a Singly Linked List.
- Enable users to dynamically add and remove songs.
- Facilitate seamless traversal through the playlist.
- Optimize memory allocation and minimize memory wastage.
- Enhance understanding of linked lists through real-world application development.
- Develop a structured and modular codebase that can be expanded further.

The primary objective is to create a functional playlist manager that demonstrates the benefits of using a Singly Linked List. This includes providing users with the ability to add songs to the beginning or end of the playlist, remove songs by name or position, and easily navigate through the playlist to play the next song. Memory optimization is also a critical objective, ensuring that the playlist manager uses memory efficiently and avoids unnecessary overhead.

Furthermore, the project aims to deepen the understanding of linked lists by providing a real-world application scenario. Students will learn how to implement basic operations such as insertion, deletion, and traversal, and how to apply these operations in a practical context.

Scope and Limitations

Scope:

- Users can add songs to the start or end of the playlist.
- Users can remove a song by specifying its name or position.
- The system will allow traversal through songs to play the next song.
- The entire playlist can be displayed in order.
- The project will provide a basic command-line interface (CLI) for interaction.

Limitations:

- The project will be a console-based application, without a graphical user interface (GUI).

- It will not support database storage; the playlist data will be stored only during runtime.
- The system will not support backward traversal since a singly linked list is being used.
- Advanced features like song searching, shuffling, or sorting are not included.

The project will be confined to a command-line interface (CLI) for user interaction. While this provides a simple and straightforward way to manage the playlist, it lacks the visual appeal and ease of use of a graphical user interface (GUI). The playlist data will only be stored during runtime, meaning that the playlist will be lost when the application is closed. This limitation is due to the absence of database storage, which would allow for persistent storage of playlist data.

Problem Statement and Solution

Problem:

Maintaining and managing playlists dynamically is a crucial feature in modern music applications. A static array-based playlist has limitations in terms of memory allocation and modification efficiency. Deleting or inserting songs at different positions in an array-based approach can lead to inefficient memory usage and time complexity.

Solution:

This project proposes a Singly Linked List-based Playlist Manager, which provides dynamic memory allocation and efficient insertion/deletion operations. Using a linked list eliminates the constraints of fixed storage and allows smooth modifications to the playlist, making it a suitable solution for music applications.

The core problem addressed by this project is the inefficiency of using static array-based playlists. Arrays require a contiguous block of memory to be allocated, which can lead to wasted space if the playlist is not fully utilized. Furthermore, inserting or deleting songs in the middle of an array requires shifting elements, resulting in a time complexity of O(n), where n is the number of songs in the playlist. This can be slow and inefficient, especially for large playlists.

The solution is to use a Singly Linked List, which allows for dynamic memory allocation and efficient insertion/deletion operations. Linked lists do not require

contiguous memory allocation, as each node stores a pointer to the next node in the list. This allows for more efficient memory usage, as memory is allocated only when needed. Inserting or deleting songs in a linked list requires only updating the pointers of the surrounding nodes, resulting in a time complexity of O(1) for insertion and deletion, assuming the position of the node is known.

Proposed System and Methodology

The Playlist Manager system will be implemented using a Singly Linked List, where each node represents a song. The system will provide functionalities to add, remove, display, and traverse songs efficiently. The implementation will follow modular programming practices to ensure code clarity and maintainability.

Key features include:

- Add a Song: Insert a song at the start or end of the playlist.
- Remove a Song: Delete a specified song by its name or position.
- Display Playlist: Show all songs in sequential order.
- Play Next Song: Move to the next song in the playlist.
- Memory Efficiency: Utilize linked lists to allocate memory dynamically and avoid wastage.

Each node in the linked list will store the song name (string) and a pointer to the next song. Operations will include insertion (adding songs dynamically), deletion (removing songs from any position in the list), and traversal (navigating through the playlist to play the next song). Modular programming practices will be followed to ensure code clarity and maintainability. The system will be designed to be easily extensible, allowing for future enhancements and additions.

System Architecture and Technologies Used

Node Structure:

- Song name (string)
- Pointer to the next song

Operations:

- Insertion: Add songs to the list dynamically.
- Deletion: Remove songs from any position in the list.
- Traversal: Navigate through the playlist to play the next song.

Technologies Used:

- Programming Language: C++
- Data Structure: Singly Linked List
- Development Environment: VS Code / CodeBlocks

The system architecture revolves around the Singly Linked List data structure. Each node in the list contains two main components: the song name, which is stored as a string, and a pointer to the next node in the list. The pointer is what allows the nodes to be linked together, forming the playlist. The system will provide functionalities to add, remove, display, and traverse songs efficiently.

The choice of programming language is flexible, with options including C, C++, Java, and Python. Each of these languages provides the necessary tools and libraries for implementing the Singly Linked List and managing memory dynamically. The development environment can also be chosen based on personal preference, with options including VS Code, CodeBlocks.

Use Case Scenarios:

- **Mood-based Listening:** Users can choose from mood-specific playlists like Happy, Sad, Romantic, or Liked to enhance their listening experience.
- **Dynamic Navigation:** Within a selected playlist, users can play, pause, skip to the next, or go to the previous song seamlessly.
- **Playlist Management:** Users can view songs in a playlist before deleting, ensuring accurate deletion.
- **Interactive CLI Flow:** Menu-driven interface for selecting operations (Add, Delete, Play, Navigate, etc.) makes the program user-friendly.

Sample Input-Output or Test Cases:

- Test Case 1: Add Song to 'Happy' Playlist

```
Input:
1. Add Song
2. Select Playlist: Happy
3. Enter Song Name: "Best Day Ever"
4. Enter Artist Name: "SpongeBob"

Output:
Song "Best Day Ever" by SpongeBob added to Happy playlist.
```

- Test Case 2: Play Song from 'Romantic' Playlist

```
Input:
1. Play Song
2. Select Playlist: Romantic
3. Songs in Playlist:
   [1] Perfect by Ed Sheeran
   [2] All of Me by John Legend
4. Enter Choice: 2

Output:
Now Playing: All of Me by John Legend 🎵
```

- Test Case 3: Delete Song

```
Input:
1. Delete Song
2. Select Playlist: Sad
3. Songs in Playlist:
   [1] Let Her Go by Passenger
   [2] Fix You by Coldplay
4. Enter Song to Delete: Let Her Go

Output:
Song "Let Her Go" deleted from Sad playlist.
```

Time and Space Complexity:

Operation	Time Complexity	Space Complexity
Add Song	O(1) (at end) / O(n) (at pos)	O(1)
Delete Song	O(n)	O(1)
Play Next/Previous	O(1)	O(1)
Display Playlist	O(n)	O(1)

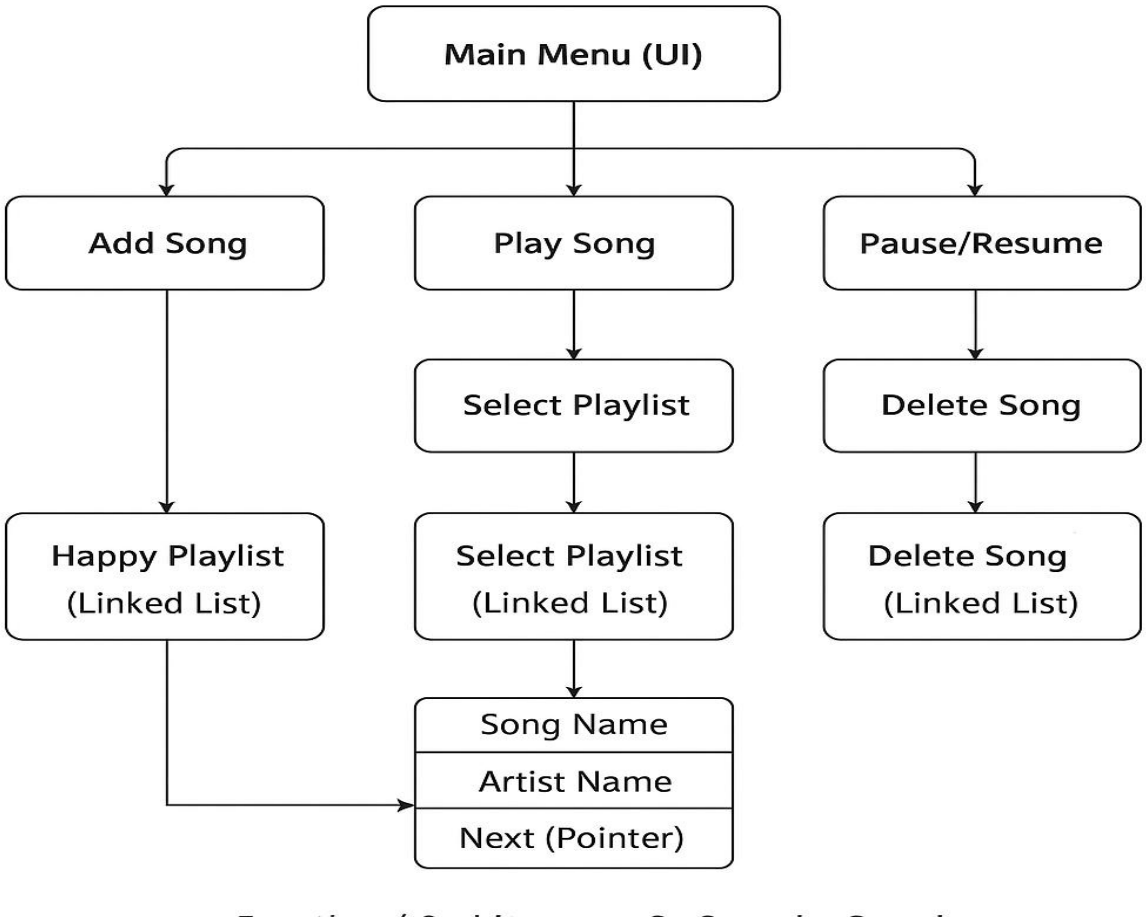
1. n is the number of songs in a playlist.
2. Space complexity is minimal since songs are stored in a dynamic, node-based structure.

Code Design Outline (Modular Functions):

The codebase follows a modular and structured approach for better readability and maintainability.

Function	**Purpose**
addSong()	Add new song to playlist (start or end)
deleteSong()	View playlist then delete song by name
displayPlaylist()	Show all songs in chosen playlist
playSong()	Play any song from selected playlist
playNextSong() / playPreviousSong()	Navigate songs in current playlist
mainMenu()	Drives the full program UI using switch-case
getPlaylistByChoice()	Returns pointer to selected mood-based playlist

Functional Flowchart of Playlist Manager



The functional flowchart illustrates the core structure and behavior of the Playlist Manager system. It begins with a central Main Menu that branches into key operations:

- **Add Song**: Allows users to choose a playlist (Happy, Sad, Romantic, Liked) and add songs along with artist details into the appropriate singly linked list.
- **Play Song**: Displays songs from the selected playlist and enables interactive controls like Play, Pause/Resume, Next, and Previous.
- **Delete Song**: Presents the songs in the chosen playlist and lets users delete by name, ensuring accurate operation.
- **Switch Playlist**: Offers dynamic switching between mood-based playlists during playback.

- **Navigation**: Each function returns to the Main Menu, maintaining a seamless and intuitive flow.

Each playlist is a separate singly linked list, and all operations are driven using modular functions and switch-case logic. This architecture ensures clean memory management, smooth traversal, and extendability.

Conclusion and Future Enhancements

This project presents a robust and interactive solution to playlist management using Singly Linked Lists. It successfully demonstrates the strengths of linked list structures, particularly in efficient memory handling and real-time node manipulation.

The current implementation includes:

- Mood-based playlists (Happy, Sad, Romantic, Liked)
- Song addition and deletion with artist details
- Song playback with next/previous navigation and pause/resume
- Dynamic playlist switching
- Structured, menu-driven CLI for smooth interaction

This project simulates a real-world audio manager while reinforcing core concepts of data structures, algorithms, and modular code design.

Future enhancements may include:

- A graphical user interface (GUI) for improved user experience
- Backward traversal support using Doubly Linked Lists
- Song search and sorting capabilities
- Shuffle functionality
- Persistent storage using files or a database

These additions would transform the Playlist Manager into a more comprehensive, user-friendly music management application, making it scalable for future use cases.