**MUSIC PLAYLIST MANAGER**

**Group Number:** 6

**Name of the Group Members**

**ANKITA.A (231IT007)**

**B K HIMA BINDHU (231IT014)**

|  |
| --- |
| **What is the Application?**  *(150-200 words approx.)* |
| The Music Playlist Manager project is a command-line application that focuses on the underlying data structures and algorithms for managing music playlists. It’s primary application lies in demonstrating how to efficiently handle and manipulate collections of music tracks using programming concepts.  In this project, users can perform various operations such as adding, removing, and searching for songs within a playlist. The implementation leverages data structures like arrays or linked lists to store track information, enabling efficient data manipulation and retrieval. By utilizing algorithms for sorting and searching, the application can quickly organize playlists by different criteria, such as artist, genre, or song title.  Additionally, the project serves as a practical exercise in applying concepts from data structures and algorithms, reinforcing the understanding of how these principles can be used to solve real-world problems. Although it lacks a graphical user interface, the command-line interface allows for straightforward interaction, showcasing the core functionality of a music playlist manager in a simplified form. |
|  |
|  |
| **What is a Possible Solution?**  *(Minimum 150 words or more, include figures and tables if needed)* |
| A possible solution to creating an effective music playlist manager is to conceptualize a system that prioritizes ease of song management, intuitive navigation, and efficient playback control. The primary goal is to enable users to add, delete, and locate songs quickly, as well as navigate seamlessly through their playlist. To achieve this, the solution will focus on creating a logical, organized structure that is user-centric, ensuring that features align with the typical interactions users have with music playlists.  The system would begin by structuring the playlist as a collection that supports dynamic modification, allowing users to add or delete songs as needed. Navigation is central to the user experience, so the solution would integrate an approach that allows users to move both forward and backward through songs efficiently, creating a more interactive and customizable experience.  For song searching, the solution would include a method for quick retrieval based on song names. This approach ensures users can efficiently locate specific songs, even in a large playlist, without scrolling through the entire collection.  Together, these theoretical components provide a blueprint for a responsive, efficient, and user-friendly playlist manager. This design aims to improve the overall user experience by focusing on essential features that align with real-world use cases of playlist management. |
|  |
|  |
| **Which Data Structure to Use for the Solution (Details of the Data Structure)**  *(Minimum 150 words or more, include figures and tables if needed)* |
| For our music playlist manager project, the following data structures have been selected to manage various operations efficiently:  **DATA STRUCTURES**  **1.Doubly Linked List**:  A **Doubly Linked List (DLL)** consists of nodes, where each node contains a data element (in this case, the song) and two pointers: one pointing to the next node and the other pointing to the previous node. This allows for both forward and backward traversal, which is essential for many of the features in our application, such as adding/removing songs and navigating through the playlist.  **Usage**:  **Deleting Songs**: DLL enables efficient insertion and removal of songs at both ends and in the middle, without shifting elements like in arrays. This dynamic operation is crucial when users frequently modify their playlists.  **Playing Next Song**: By traversing the linked list in both directions (forward for next and backward for previous), the DLL allows the user to easily move between songs.  **2. Stacks**:  **Usage:**  **Recently Played List**: **Stack**  manages recently played songs efficiently. Since a stack follows the **Last In, First Out (LIFO)** principle, it allows quick access to the last played song with **O(1)** push and pop operations. This makes it ideal for replaying the most recent songs in reverse order.  **ALGORITHMS**  **2.Linear Search**:  Linear search is an algorithm that searches for an element by checking each item in a list sequentially. It is simple and works well with linked lists where random access is not possible.  **Usage**:  **Searching Songs**: In a linked list, elements are stored non-contiguously, meaning that linear search works well to find a song by traversing each node until the desired song is found. Although this has a time complexity of O(n), it is efficient for smaller playlists.  **3. Merge Sort**:  **Merge Sort** is a divide-and-conquer sorting algorithm that splits the playlist into smaller sublists, sorts them recursively, and then merges them back together.  **Usage**:   * **Sorting Songs**: Merge Sort is used to sort the playlist. It has a time complexity of O(n log n), which is more efficient than other sorting algorithms like bubble sort, especially for larger datasets. It is also a stable sort, ensuring that songs with identical attributes maintain their relative order in the playlist. |
|  |
|  |
| **Justify Why This is the Best Data Structure for this Application.**  *(Minimum 150 words or more; it is better to list the justifications point-wise, including figures and tables if needed)* |
| **1. Doubly Linked List for Adding/Deleting Songs:**   * **Efficient Dynamic Operations**: DLL allows songs to be added or removed without the need to shift elements, making it more efficient than arrays, especially for large playlists where users may frequently update the list. * **Forward and Backward Traversal**: For playing songs and maintaining a recently played list, DLL provides the ability to traverse both directions, which enhances the user experience. This structure ensures that users can easily navigate to the next and previous songs, improving playback control. * **Flexible Modification**: DLL supports operations like adding/removing songs from both ends and the middle, providing flexibility in modifying the playlist.   **2. Linear Search for Searching Songs:**   * **Simplicity and Efficiency for Smaller Playlists**: Linear search is simple and works directly on the linked list. While it has O(n)time complexity, it is efficient for smaller playlists where complex search algorithms (like binary search) may add unnecessary overhead. * **No Need for Sorting**: Since songs are not necessarily in sorted order, linear search is a practical solution as it does not require sorting the entire playlist first, making it ideal for dynamically changing playlists.   **3. Stack for Recently Played List**:  • **Efficient Playback**: Stacks allow quick retrieval of the most recently played song, making it ideal for replaying tracks in reverse order.  • **Simplified Structure**: With a single-directional approach, stacks minimize memory usage and avoid the complexity of managing forward and backward links.  **Merge Sort for Sorting Songs**:   * **Efficient Sorting**: Merge Sort has a time complexity of O(n log n), which is much more efficient than algorithms like bubble sort. This makes it the best choice for sorting larger playlists without sacrificing performance. * **Stability**: Merge Sort is a stable sort, which is essential for ensuring that songs with identical attributes (e.g., songs by the same artist) maintain their relative order in the playlist.   **Overall Applicability**:   * The combination of DLL for dynamic playlist management, Merge Sort for efficient sorting, and Linear Search for simple searching makes this set of data structures ideal for a playlist manager. They balance ease of implementation, efficiency, and functionality, ensuring a smooth user experience while managing and navigating the playlist. |
|  |
|  |
| **Implementation Details (Experimental Setup)**  *(150-200 words approx.)* |
| For this music playlist manager project, **C++** was used as the primary programming language. The main data structure employed is the **Doubly Linked List (DLL)**, which allows efficient addition and deletion of songs, as well as smooth navigation during playback. **Linear Search** is used to search for songs, providing a simple method for traversing the list and finding a song by its name. For sorting the playlist, **Merge Sort** is used due to its O(n log n)time complexity, which is more efficient than simpler sorting algorithms.  The playlist data (song names and relevant metadata) are stored in linked list nodes, and the user interacts with the system through a console interface, adding, deleting, searching, and sorting songs in the playlist. The program was developed and tested on a **Windows** environment using **Visual Studio Code** with the **g++ compiler**. |
|  |
|  |
| **Complexity Analysis**  *(100 words approx.)* |
| The time complexity of each operation in our music playlist manager is as follows:  1. **Adding/Deleting Songs (Doubly Linked List)**:   * **Time Complexity**: O(1) for adding or deleting at the head or tail. For operations at arbitrary positions, the time complexity is O(n)due to the need for traversal.   2.**Playing Next Song (Doubly Linked List)**:   * **Time Complexity**: O(1) since, the **Doubly Linked List (DLL)** allows direct access to both the next and previous songs.   **3. Displaying List of Previously Played Songs (Stack):**   * **Time Complexity**: O(n), where *n* is the number of songs in the stack. Displaying the list of previously played songs involves traversing the stack from the most recent song (top of the stack) to the oldest (bottom of the stack). In the worst case, if all *n* songs need to be displayed, the traversal takes linear time, i.e., O(n). However, if only a subset of songs is displayed, the time complexity would depend on how many songs are accessed.   2. **Searching Songs (Linear Search)**:   * **Time Complexity**: O(n), as the list must be traversed sequentially to find the desired song.   3. **Sorting Songs (Merge Sort)**:   * **Time Complexity**: O (n log n) which is much more efficient than bubble sort for large playlists. Merge sort divides the list into smaller sublists and sorts them recursively, ensuring that even larger datasets can be handled efficiently. |
|  |
|  |
| **Observations and Conclusions**  *(100 words approx.)* |
| The music playlist manager successfully implements all required features, including adding, deleting, searching, and sorting songs. The use of a Doubly Linked List (DLL) provides efficient operations for dynamically managing the playlist, allowing easy song additions and deletions, as well as smooth navigation between songs during playback. Stackswere employed for the recently played feature, enabling quick access to the last played song**.** Linear Search works effectively for smaller playlists, providing simple sequential searching. The decision to use Merge Sort for sorting the playlist ensures better performance with a time complexity of O(nlogn), making the sorting operation efficient even for larger playlists. Overall, the application provides a user-friendly experience with smooth functionality for managing songs. Future improvements could focus on refining the search mechanism and further optimizing performance for large playlists. |