Izel Rodriguez Diaz

07/18/2025

Music Library App – Project Report

My Music Library App is a terminal-based Java program that allows users to store, play, and organize songs. It supports adding songs, creating a playlist using a linked list, playing songs with queue-based logic, and recommending songs based on genre using a recursive method. The app is entertaining and practical for anyone who wants a lightweight command-line music organizer and player. It mimics a digital jukebox by managing a playback queue and history, and it also serves as an educational tool for understanding data structures.

The program features three major algorithms. The first is a linked list traversal algorithm for adding songs to a playlist. It checks if the head of the list is null; if so, the new song becomes the head. Otherwise, it traverses to the end of the list and appends the new node. The second is a recursive algorithm that recommends songs based on genre. It stops when either the number of recommended songs is met or the list is exhausted, checking genre matches and printing matching songs. The third algorithm uses a queue to play songs and a stack to keep track of history. It polls the next song in the queue and pushes it onto the history stack, ensuring playback order and backtracking support.

These algorithms were developed by breaking the problem into manageable steps and building functions around specific tasks. I used ChatGPT to help guide the structure of these algorithms and understand best practices, but I designed and wrote the recursive logic and list traversal independently to match the learning goals. I also modified the output and error handling to better support user input, such as preventing invalid data and giving clear feedback.

The Big O time complexity of the three algorithms is as follows: the playlist addition (linked list) is O(n) in the worst case due to traversal. The recursive genre recommendation is O(n) because it potentially checks every song once. The playNext algorithm, using a queue, runs in O(1) for polling and pushing to the stack, making it very efficient.

For data structures, I used a stack to track playback history, a queue for the upcoming song queue, a HashMap to organize songs by genre, and a custom singly linked list for the playlist. I chose these structures to align with real-world music players and to

demonstrate a range of CS concepts. For example, the queue allows constant-time next-song retrieval, while the map enables quick genre lookups.

During development, one opportunity I found was enhancing user experience by validating inputs. I added checks to prevent blank titles, artists, or genres, and ensured the system wouldn't accept songs with numbers in names, mimicking real music apps. An error I faced was improperly using a switch statement with a try-catch block inside; I resolved this by placing the try-catch around the whole switch block to ensure it handled input exceptions consistently.

If I were to build a third version of this app, I would add file I/O so the songs and playlist persist between sessions. I'd also consider adding GUI support using JavaFX to make the experience more visual and engaging. Finally, user authentication and genre statistics could make the app feel more personal and robust.

If I were to develop a third version of this app, I'd focus on making it more dynamic and user-friendly. First, I'd integrate real music playback using a library like JavaFX Media so users can listen to songs directly in the app. Next, I'd clean up and optimize the code for better performance, possibly using more efficient data structures. To enhance the experience, I'd add a GUI with JavaFX, featuring interactive visuals like album artwork and playback controls. Additionally, I'd implement file I/O or a simple database to save playlists between sessions and consider adding user accounts for personalized stats, such as genre preferences. These changes would transform the app from a basic playlist manager into a more engaging and functional music player.

Algorithm 1: Adding a Song to a Linked List Playlist

1. Is the head null?
2. If yes, set head = new node
3. If no, traverse to the end of the list
4. Add the new node at the end

```
//adds song to end of playlist
public void addSong(Song song){

    Node newNode = new Node(song);
    if(head == null){
        head = newNode;
    }else{
        Node current = head;
        while(current.next != null){
            current = current.next;
        }
        current.next =newNode;
    }
}
```

Algorithm 2: Genre-Based Recommendation (Recursive Search)

1. Base case: count = 0 or end of list
2. If genre matches, print and decrement count
3. Recurse with next index

```java
//recursive mthod that prints up to count
private static void recommendRecursive(List<Song> songs, String genre, int count, int index){
    if(count == 0 || index >= songs.size())return;
    Song current = songs.get(index);
    if (current.getGenre().equalsIgnoreCase(genre)) {
        System.out.println(current);
        recommendRecursive(songs, genre, count -1, index+1);
    }else{
        recommendRecursive(songs, genre, count, index+1);
    }
}
}
```

Algorithm 3:: Playing the Next Song from the Queue

1. Is the queue empty?
2. If yes, display message
3. If no, remove (poll) song from queue
4. Push to history stack
5. Print playing message

```java
public void playNext(){
    if(!queue.isEmpty()){
        Song next =queue.poll();        //gets next song
        history.push(next);
        System.out.println("Playing: " + next);
    }else{
        System.out.println(x:"Queue is empty.");

    }
}
```