C/C++

Training Assignment

Document Code	25e-BM/HR/HDCV/FSOFT
Version	1.1
Effective Date	18/08/2025

Danang, 08/2025

RECORD OF CHANGES

No	Effective Date	Change Description	Reason	Reviewer	Approver

Contents

CODE:	<CPP.MiniProject.Opt1>
TYPE:	<>
LOC:	<Lines of Code>
DURATION:	<240 minutes>

Major Project Assignment: Building the Core Engine for an Automotive Music Player Application

Overall Goal

You are tasked with building the critical backend components for a music player application on a car's infotainment system. The objective is to design an efficient, flexible, and scalable system by selecting and applying the appropriate data structures for each feature.

Context

The system needs to manage a large music library (tens of thousands of songs), allow users to perform fast searches, create and control playlists (playback queues), review their listening history, and use smart features like a non-repeating shuffle.

Part 1: Music Library and Playback Queue Management

Requirements: Use **`std::vector`** and **`std::list`**.

In this part, you must differentiate and select the correct data structure for two distinct tasks: storing the main music library (infrequently modified, requires fast access) and managing the user's playback queue (highly dynamic).

1. **`MusicLibrary` Class:**
    - Create a `MusicLibrary` class that uses a `std::vector<Song>` to store the entire list of songs loaded at startup. You will define the `Song` `struct` yourself (it should have an id, title, artist, album, duration).
    - In your design document, **justify** why `std::vector` is a suitable choice for storing the music library. (Hint: fast random access by index, efficient memory usage, infrequent additions/deletions).
2. **`PlaybackQueue` Class:**
    - Create a `PlaybackQueue` class that uses a `std::list<Song>` to manage the list of songs the user is currently listening to.
    - **Justify** why `std::list` is more suitable than `std::vector` for the playback queue. (Hint: efficient insertion/deletion in the middle of the list, pointers/iterators are not invalidated when the list is modified).
    - Implement the methods: `void addSong(const Song& song)`, `void removeSong(int songID)`, `Song getCurrentSong()`, `void playNext()`.
3. **Integration Algorithm:**

- Write a function `void addAlbumToQueue(const std::string& albumName, const MusicLibrary& library, PlaybackQueue& queue)` that finds all songs from a specific album in the `MusicLibrary` and adds them to the end of the `PlaybackQueue`.

Part 2: Accelerating Searches and Indexing Metadata

Requirements: Use **`std::map`** and **`std::unordered_map`**.

With tens of thousands of songs, a linear search through the `std::vector` (from Part 1) is too slow. You need to build indexing mechanisms to speed up queries.

1. **Search by ID (Performance-Critical):**
    - In your `MusicLibrary` class, add a data member: `std::unordered_map<int, Song*> songIndexByID;`. The `key` is the song's unique ID, and the `value` is a pointer to the corresponding `Song` object in the main `std::vector`.
    - **Justify** why `std::unordered_map` (a hash map) is the optimal choice for searching by ID. (Hint: O(1) average-case complexity).
    - Write a method `Song* findSongByID(int id)` that uses this map.
2. **Search by Title and Sorting:**
    - In `MusicLibrary`, add another data member: `std::map<std::string, Song*> songIndexByTitle;`.
    - **Justify** why `std::map` (a self-balancing binary search tree) is a good choice in this case. (Hint: O(log n) complexity, and the map automatically maintains the keys in alphabetical order, which is useful for displaying a sorted tracklist).
    - Write a method `Song* findSongByTitle(const std::string& title)`.
3. **Advanced Challenge:**
    - A user wants to find all songs by a specific artist. A simple map won't work because one artist has many songs.
    - Design and implement a new index, `std::unordered_map<std::string, std::vector<Song*>> artistIndex`, to efficiently solve this requirement.

Part 3: Implementing History and Shuffle Features

Requirements: Use **`std::stack`**, **`std::queue`**, and **`std::set`**.

1. **Playback History ("Back" Button):**
    - Create a `PlaybackHistory` class that uses a `std::stack<Song>` internally.
    - Whenever a song finishes playing, `push` it onto this stack.
    - Write a `Song playPreviousSong()` method that `pop`s the song from the stack and returns it to the user.
    - **Justify** why the LIFO (Last-In, First-Out) structure of a **Stack** is perfectly suited for a "Back" button's functionality.

2. **"Play Next" Queue:**
    - Describe how you would use a `std::queue` to manage a list of songs that the user has marked to "Play Next."
    - **Justify** why the FIFO (First-In, First-Out) structure of a **Queue** is appropriate for this logic.
3. **Smart Shuffle (No Immediate Repeats):**
    - Create a `ShuffleManager` class. When a user enables shuffle for a playlist, this class must:
        - Create a temporary `std::vector` of the songs and shuffle it using `std::random_shuffle` (or `std::shuffle` in C++11).
        - Use a `std::set<int>` to store the IDs of songs that have **already been played** in the current shuffle cycle.
        - When selecting the next song, if its ID is already in the `set`, skip it and pick the next one in the shuffled vector.
        - When all songs have been played (the `set`'s size equals the playlist's size), clear the `set` to begin a new cycle.
    - **Justify** why using `std::set` to check if a song has already been played is very efficient. (Hint: O(log n) complexity).

Part 4: Integration and Advanced Algorithm

Requirements: Integrate the components you've built and implement a more complex algorithm.

1. **System Integration:**
    - Create a main `MusicPlayer` class that holds instances of `MusicLibrary`, `PlaybackQueue`, and `PlaybackHistory`.
    - Write a method `void selectAndPlaySong(int songID)` that follows this flow:
        1. Uses `findSongByID` from the `MusicLibrary` to find the song.
        2. If another song is currently playing, `push` it to the `PlaybackHistory`.
        3. Set the newly found song as the current track and add it to the `PlaybackQueue`.
2. **"Smart Playlist" Algorithm:**
    - Write a function `PlaybackQueue generateSmartPlaylist(const Song& startSong, const MusicLibrary& library, int maxSize)`.
    - This function's task is to create a "smart" playlist based on the starting song. "Smart" can be simply defined as **songs by the same artist or from the same album**.
    - **Algorithm Hint:**
        1. Use the structure of a Breadth-First Search (BFS) algorithm.
        2. Start with `startSong`. Use a `std::queue` to hold songs to "explore."
        3. Use a `std::set<int>` to mark song IDs that have already been added to the playlist to avoid duplicates.

4. From a given song, its "neighbors" are all other songs by the same artist or on the same album (use the indexes created in Part 2 for fast lookups).
5. Continue the BFS process until the playlist reaches `maxSize`.

3. Final Report:

- Write a short report (approx. 1-2 pages) explaining your data structure choices for each feature.
- For each choice, analyze its algorithmic complexity (Big O Notation) for the main operations (add, remove, search) and prove that your choice is optimal for that feature's requirements.