PlayWise: Building a Smart Music Playlist Management Engine

Background Story

With the explosion of personalized music platforms, a new player—**PlayWise**—is emerging with a bold vision: to redefine how users interact with playlists. The platform must support real-time playlist manipulation, personalized recommendations, memory-efficient search, and smart sorting based on user preferences.

To power this vision, the product team needs a highly optimized backend engine—one that intelligently leverages data structures like **Linked Lists, Stacks, Trees, HashMaps, and Sorting algorithms**. You have been tasked with building this core engine. Your implementation will serve as the backbone of PlayWise's personalization layer.

Common Core Implementation: Modules & Scenarios

Each student/team must implement the following components as part of the **common mandatory work**.
All code must be **clean, modular, and well-documented**, with clear **time and space complexity annotations**.

1. Playlist Engine using Linked Lists

- **Scenario:** Users can create playlists where songs can be added, deleted, reordered, or reversed.
- **Requirements:**
  - Use a **doubly linked list** to implement each playlist.
  - Support operations: add_song(title, artist, duration), delete_song(index), move_song(from_index, to_index), reverse_playlist().
- **Why:**
  - Reinforces ordered data manipulation and pointer logic.
  - Models real-world playlist editing behavior.

2. Playback History using Stack

- **Scenario:** Users can “undo” recently played songs to re-queue them.
- **Requirements:**
  - Maintain a **stack** of recently played songs.
  - Allow undo_last_play() to re-add the last played song to the current playlist.
- **Why:**
  - Explores LIFO behavior.
  - Introduces simple control flow manipulation using stack operations.

3. Song Rating Tree using Binary Search Tree (BST)

- **Scenario:** Songs are indexed by user rating (1–5 stars). The system should allow:
  - Fast insertion, deletion, and search by rating.
- **Requirements:**
  - Use a **BST** where each node holds a rating bucket (1 to 5).
  - Each bucket stores multiple songs with that rating.
  - Allow insert_song(song, rating), search_by_rating(rating), delete_song(song_id).
- **Why:**
  - Applies tree structures to personalized recommendation/search.
  - Encourages reuse of standard BST functions.

4. Instant Song Lookup using HashMap

- **Scenario:** When a user searches by song title or ID, the system must return song metadata in O(1) time.
- **Requirements:**
  - Use a **hash map** to map song_id or title to song metadata.
  - Sync this map with updates in the playlist engine.
- **Why:**
  - Reinforces use of hashing for constant-time lookup.
  - Emphasizes synchronization between structures.

5. Time-based Sorting using Merge/Quick Sort

- **Scenario:** Users can sort playlists based on:
  - Song title (alphabetical),
  - Duration (ascending/descending),
  - Recently added
- **Requirements:**
  - Implement **at least one sorting algorithm** (Merge, Quick, or Heap sort).
  - Allow toggle between sorting criteria.
  - Compare performance with built-in sort (if used).
- **Why:**
  - Deepens understanding of time-space trade-offs.
  - Shows real use of sorting algorithms in dynamic systems.

6. Playback Optimization using Space-Time Analysis

- **Scenario:** You must analyze the memory and performance behavior of your playlist engine.
- **Requirements:**
  - Annotate all core methods with **time and space complexity**.
  - Optimize where possible (e.g., constant-time node swaps, lazy reversals).
- **Why:**
  - Reinforces the importance of asymptotic analysis.
  - Builds habits of thinking like an engineer, not just a coder.

7. System Snapshot Module (Live Dashboard Prototype)

- **Scenario:** A debugging interface shows:
  - Top 5 longest songs,
  - Most recently played songs,
  - Song count by rating.
- **Requirements:**
  - Use combination of sorting, hash maps, and tree traversal to output live stats.
  - Write an export_snapshot() function that returns all dashboard data.
- **Why:**
  - Integrates multiple DSA skills in a mini real-time system.
  - Tests student ability to reason about state and aggregation.

Engineering & Documentation Expectations

All students must:

- Annotate **time and space complexity** for every core function.
- Document **trade-offs** in data structure selection.
- Justify **algorithm choice** with examples.
- Submit a single .md or .pdf **Technical Design Doc** that includes:
- High-level architecture
- Diagrams and flowcharts
- Pseudocode for major algorithms
- Brief benchmarks or test case results (where applicable)

talenciaglobal
Great talent. Global opportunities

Evaluation Focus

Criteria Weight (%

Correctness & Functionality 25%
Engineering & Optimization 25%
System Design & Architecture 20%
Code Quality & Documentation 15%
Creativity & Insight 15%