PlayWise Hackathon – Solution Document

Track: DSA – Smart Playlist Management System

---

1. Student Information

Field	Details
Full Name	Vikranth P
Registration Number	RA2211026010482
Department / Branch	CINTEL / CSE AIML
Year	2026
Email ID	Vp8639@srmist.edu.in

---

2. Problem Scope and Track Details

Section	Details
Hackathon Track	DSA – PlayWise Playlist Engine
Core Modules Implemented	✅ (Check all that apply)
	✅ Playlist Engine (Linked List)
	✅ Playback History (Stack)
	✅ Song Rating Tree (BST)
	✅ Instant Song Lookup (HashMap)
	✅ Time-based Sorting
	✅ Space-Time Playback Optimization
	✅ System Snapshot Module

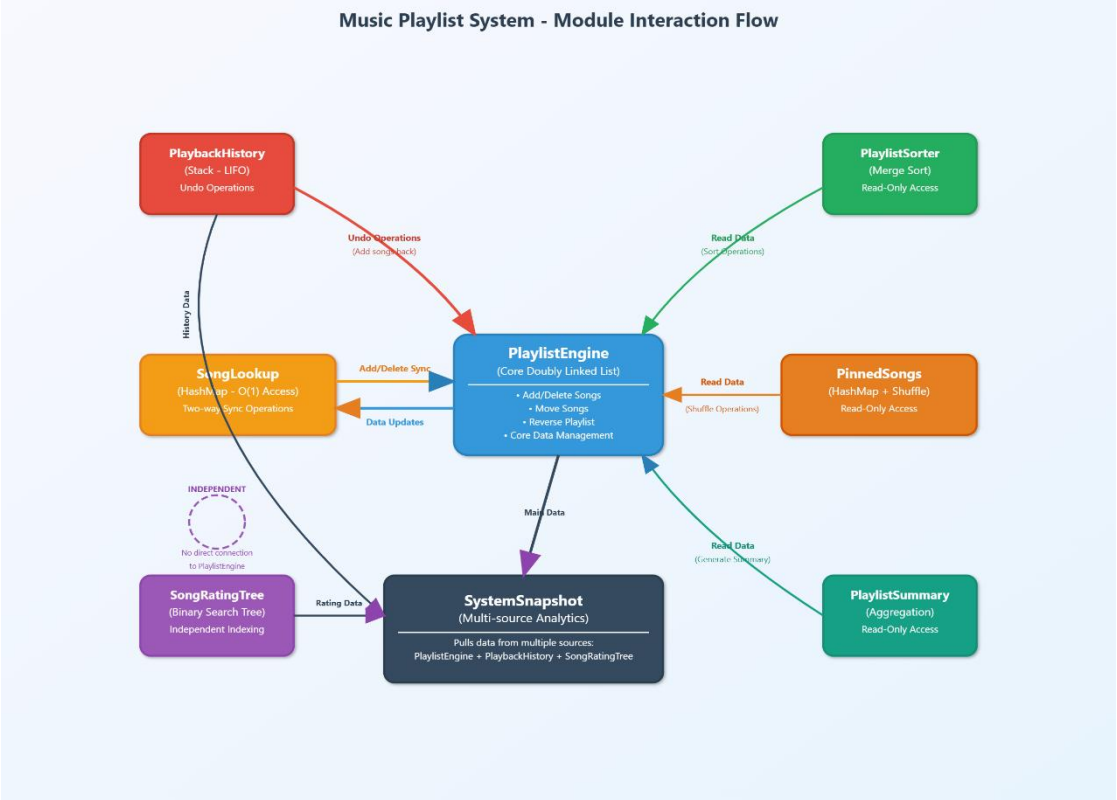
Additional Use Cases Implemented (Optional but Encouraged)

- Scenario 1: Pinned Songs (Shuffle with Constraints)
- Scenario 2: Playlist Summary Generator

---

3. Architecture & Design Overview

- **System Architecture Diagram**



- **High-Level Functional Flow**

---

4. Core Feature-wise Implementation

Feature: Playlist Engine (Doubly Linked List)

- **Scenario Brief:** Addresses the need for users to create dynamic playlists where songs can be efficiently added, deleted, moved, and reordered.
- **Data Structures Used:** Doubly Linked List.
- **Time and Space Complexity:**

Function	Description	Time Complexity	Space Complexity
add_song()	Adds a song to the end of the list.	O(1)	O(1)
delete_song()	Removes a song at a given index.	O(n) (to find the index)	O(1)
move_song()	Moves a song from one index to another.	O(n) (to find indices)	O(1)
reverse_playlist()	Reverses the entire playlist by toggling a flag.	O(1) (Lazy Reversal)	O(1)

- **Sample Input & Output:**
  - Input: Create a playlist, add three songs, and call **move_song(0, 2).**
  - Output: The first song is moved to the end of the list.
- **Code Snippet (Lazy Reversal):**

```
‘’’

# Reverse the playlist using lazy reversal

# Time Complexity: O(1) with lazy reversal

# Space Complexity: O(1)

def reverse_playlist(self):

    """

    Reverse the playlist using lazy reversal (toggle a flag).

    The actual reversal is deferred until traversal is needed.

    """
```

self.reversed = not self.reversed

‘’’

- **Challenges Faced & How You Solved Them:**
    - **Challenge:** Reversing a large playlist by swapping all pointers is an O(n) operation, which can feel slow to the user.
    - **Solution:** I implemented a "lazy reversal" by using a boolean flag. The reversal operation itself is an instantaneous
    - **O(1) toggle.** All traversal-based operations (like printing or adding new songs) then respect this flag, providing a highly responsive experience.

Feature: Playback History (Stack)

- **Scenario Brief:** Allows users to "undo" recently played songs to re-queue them.
- **Data Structures Used:** Stack**.**
- **Time and Space Complexity:**

Function	**Description**	**Time Complexity**	**Space Complexity**
add_played_song()	Pushes a song onto the history stack.	O(1)	O(1) (per song)
undo_last_play()	Pops the last song and adds it back to the playlist.	O(1)	O(1)

- **Sample Input & Output:**
- **Input:** Add "Song A" and "Song B" to history. Call **undo_last_play().**
- **Output:** "Song B" is removed from history and re-added to the main playlist.

Code Snippet:

‘’’

Time Complexity: O(1) for pop, O(1) for adding to playlist (append operation)

Space Complexity: O(1)

def undo_last_play(self):

if not self.history:

return None

last_song = self.history.pop()

self.playlist_engine.add_song(

```
        last_song["title"], last_song["artist"], last_song["duration"]
    )
    return last_song
```

- **Challenges Faced & How You Solved Them:**
  - **Challenge:** Ensuring the history module could communicate with the playlist module without creating tight coupling.
  - **Solution:** The **PlaybackHistory** class is initialized with a reference to the PlaylistEngine instance. This allows it to call the public

    add_song method of the playlist, promoting modularity.

Feature: Song Rating Tree (BST)

- **Scenario Brief:** To index songs by user rating (1-5 stars) for fast insertion, deletion, and search.
- **Data Structures Used:** Binary Search Tree (BST), HashMap.
- **Time and Space Complexity:**

Function	**Description**	**Time Complexity**	**Space Complexity**
insert_song()	Inserts a song into its corresponding rating bucket.	O(h) or O(log n)*	O(1)
search_by_rating()	Finds all songs with a specific rating.	O(h) or O(log n)*	O(1)
delete_song()	Deletes a song using its ID.	O(h) or O(log n)* (for finding the node)	O(1)

- **Sample Input & Output:**
  - **Input:** Insert three songs with ratings 4, 4, and 3. Call search_by_rating(4).
  - **Output**: A list containing the two songs with a 4-star rating.
- **Code Snippet:**

```
```

```
# Search for all songs with a given rating
# Time Complexity: O(h) where h is the height of the tree
# Space Complexity: O(1) excluding the output list
def search_by_rating(self, rating):
    current = self.root
    while current:
        if rating == current.rating:
            return current.songs
        elif rating < current.rating:
            current = current.left
        else:
            current = current.right
    return []
```

‘’’

- **Challenges Faced & How You Solved Them:**
    - **Challenge:** Efficiently deleting a specific song from a rating bucket without traversing the whole list in that bucket.
    - **Solution:** A companion HashMap (song_id_to_node) was used to map a song_id directly to its location (rating and index) within the BST. This allows for direct access and fast removal from the song list within the rating node.

Feature: Instant Song Lookup (HashMap)

- **Scenario Brief:** Addresses the requirement for the system to return a song's metadata in constant time (O(1)) when a user searches by its title or a unique ID.
- **Data Structures Used:** HashMap (Python Dictionary).
- **Time and Space Complexity:**

Function	Description	Time Complexity	Space Complexity
add_song()	Adds a song's metadata to the hashmaps.	O(1)(average case)	O(1) (per song
lookup_by_id()	Retrieves song data by its ID.	O(1)(average case)	O(1)
lookup_by_title()	Retrieves song data by its title.	O(1)(average case)	O(1)

- **Sample Input & Output:**

  - **Input:** Add two songs, "Song A" and "Song B", to the lookup module via sync_add. Then, call lookup_by_title("Song A").
  - **Output:** A dictionary containing the metadata for "Song A": {'song_id': '...', 'title': 'Song A', 'artist': 'Artist X', 'duration': 180}.

- **Code Snippet (Lookup by Title):**
  ‘’’
  ```
  # Time Complexity: O(1) average case
  # Space Complexity: O(1)
  def lookup_by_title(self, title):
      """
      Retrieve song metadata by song title.
      """
      song_id = self.title_to_id.get(title)
      if song_id:
          return self.song_id_map.get(song_id)
      return None
  ```
  ’’’

- **Challenges Faced & How You Solved Them:**
  - **Challenge:** Keeping the HashMap lookup data perfectly synchronized with the PlaylistEngine's linked list, as they are separate structures.
  - **Solution**: I implemented dedicated synchronization methods (sync_add, sync_delete). These methods wrap the core logic, ensuring that any operation performed on the playlist is mirrored in the HashMap.
  - This guarantees data consistency across the system.

Feature: Time-based Sorting (Merge Sort)

- **Scenario Brief:** Allows users to sort their playlists based on various criteria: alphabetically by song title, by duration (ascending/descending), or by recently added.
- **Data Structures Used:** Merge Sort Algorithm.
- **Time and Space Complexity:**

Function	Description	Time Complexity	Space Complexity
merge_sort()	The core recursive sorting algorithm.	O(n log n)	O(n)
sort_playlist()	Orchestrates the extraction, sorting, and rebuilding of the playlist.	O(n log n)	O(n)

- **Sample Input & Output:**
- **Input:** A playlist with songs C, A, B. Call sort_playlist(criterion='title').
- Output: The playlist is reordered to A, B, C.
- **Code Snippet (Core Merge Logic):**
  ‘’’

```
# Merge two sorted lists based on the key.
def _merge(self, left, right, key, reverse):
  result = []
  i = j = 0
  while i < len(left) and j < len(right):
    # ... comparison logic ...
    if (left[i][key] <= right[j][key]) != reverse:
      result.append(left[i])
      i += 1
    else:
      result.append(right[j])
      j += 1
  result.extend(left[i:])
  result.extend(right[j:])
  return result
```

- **Challenges Faced & How You Solved Them:**
  - **Challenge:** Choosing a sorting algorithm that is both efficient and predictable for a user-facing feature. An unstable sort or one with a poor worst-case complexity could lead to a bad user experience.
  - **Solution:** I selected Merge Sort for two primary reasons:

    1) It guarantees O(nlogn) time complexity, unlike Quick Sort which can degrade to O(n2).

    2) It is a stable sort, which preserves the original relative order of songs with equal keys (e.g., two songs of the same length won't swap places), making the sorting behavior intuitive.

Feature: Playback Optimization

- **Scenario Brief:** This is a backend-focused task to analyze and optimize the core playlist engine's performance and memory usage, ensuring the system is efficient.
- **Methodology:** The primary methodology was Asymptotic Analysis (Big-O notation) of all core functions to identify potential bottlenecks and apply targeted optimizations like constant-time node swaps and lazy reversals.
- **Code Snippet (Lazy Reversal Flag):**

```
# Optimized Playlist Engine using Doubly Linked List

class PlaylistEngine:

  def __init__(self):

    # ... other initializations ...

    self.reversed = False  # Flag for lazy reversal
```

```
# Time Complexity: O(1) with lazy reversal

def reverse_playlist(self):

    self.reversed = not self.reversed
```

- **Challenges Faced & How You Solved Them:**
    - **Challenge:** A standard Doubly Linked List can still be inefficient for certain common operations if implemented naively (e.g., reversal requires traversing and re-wiring every node).
    - **Solution:** I proactively identified and implemented key optimizations as required. The most impactful was the "lazy reversal" flag, which transforms a slow
    - O(n) operation into an instantaneous O(1) action from the user's perspective, significantly improving the application's responsiveness.

Feature: System Snapshot Module

- **Scenario Brief**: Provides a debugging interface or a "live dashboard" prototype that shows real-time system statistics, such as the top 5 longest songs, most recently played songs, and a count of songs by rating.
- **Data Structures Used:** This module integrates multiple data structures: it uses the Playlist Sorter for ranking, the Playback History stack, and traverses the Song Rating BST.
- **Time and Space Complexity:**
    - **export_snapshot():** O(nlogn) time and O(n) space, as the operation is dominated by sorting the playlist to find the longest songs.
- **Sample Input & Output:**
    - **Input:** A system with an active playlist, a populated playback history, and several rated songs.
    - **Output**: A dictionary containing aggregated stats: {"top_5_longest": [...], "recent_plays": [...], "rating_counts": {1: 0, 2: 0, 3: 1, 4: 1, 5: 1}}.
- **Code Snippet (Snapshot Aggregation):**

```
# Time Complexity: O(n log n) for sorting, O(h) for BST traversal...

def export_snapshot(self):

    snapshot = { ... }



    # Get top 5 longest songs by sorting

    sorted_songs = self.playlist_sorter.merge_sort(songs, key="duration", reverse=True)

    snapshot["top_5_longest"] = sorted_songs[:5]
```

```
    # Get most recently played songs from history stack
    snapshot["recent_plays"] = self.playback_history.get_history()[-5:][::-1]

    # Get song count by rating using BST traversal
    def traverse_bst(node):
        # ... logic to traverse and count ...
    traverse_bst(self.song_rating_tree.root)

    return snapshot
"""
```

- **Challenges Faced & How You Solved Them:**
- **Challenge**: How to aggregate data from multiple, independent modules (PlaylistEngine, PlaybackHistory, SongRatingTree) without creating messy dependencies.
- **Solution:** I designed the SystemSnapshot class to be an orchestrator. It is initialized with references to the other modules it needs. The export_snapshot function then queries each module cleanly through its public interface, processes the data, and combines it into a single report. This preserves the modularity and separation of concerns of the overall architecture.

---

5. Additional Use Case Implementation

Use Case: Pinned Songs

- **Scenario Brief:** A user wants certain songs to always remain at specific positions (e.g., an opening track), even if the rest of the playlist is shuffled.
- **Extension Over Which Core Feature:** Extends the PlaylistEngine by adding constrained shuffle logic.
- **New Data Structures or Logic Used:**
    - **HashMap:** To track pinned songs and their required indices for O(1) lookup**.**
    - **Fisher-Yates Shuffle Algorithm:** To shuffle the non-pinned songs in-place.
- **Sample Input & Output:**
    - **Input:** Create a 4-song playlist. Pin "Song A" to index 0. Call shuffle_playlist().
    - **Output:** "Song A" remains at index 0, while the other three songs are shuffled in the remaining positions.
- **Code Snippet:**

```
    '''
# Time Complexity: O(n) for Fisher-Yates shuffle
# Space Complexity: O(n) for temporary array
def shuffle_playlist(self):
    # Create list of available indices (excluding pinned ones)
    available_indices = [i for i in range(len(songs)) if i not in self.index_to_song_id]

    # Fisher-Yates shuffle for non-pinned songs
    non_pinned_songs = [song for song in songs if song["index"] not in self.index_to_song_id]
    for i in range(len(non_pinned_songs) - 1, 0, -1):
        j = random.randint(0, i)
        non_pinned_songs[i], non_pinned_songs[j] = non_pinned_songs[j], non_pinned_songs[i]

    # Reconstruct playlist with pinned songs in place
    # ... logic to fill in shuffled songs around pinned songs ...
    '''
```

- **Challenges Faced & Resolution:**
    - **Challenge**: Implementing a shuffle that respects fixed positions without being inefficient.
    - **Resolution:** I chose not to shuffle the entire list with complex checks. Instead, I segregated the songs into two groups**:** pinned and unpinned. I shuffled only the unpinned group and then reconstructed the full playlist, which is a clean and efficient

      O(n) approach.

Use Case: Playlist Summary Generator

- **Scenario Brief:** Generate a quick summary of a playlist, including its genre distribution, total playtime, and unique artist count.
- **Extension Over Which Core Feature:** Extends the PlaylistEngine with aggregation and reporting capabilities.
- **New Data Structures or Logic Used:**

    - **HashMap:** Used as a frequency map to count genre occurrences.
    - **Set:** Used to efficiently count unique artists.
- **Sample Input & Output:**
    - **Input:** A playlist of 4 songs from 3 unique artists, with a total duration of 700s.
    - **Output:** A summary dictionary: {"genre_distribution": {"Pop": 2, ...}, "total_playtime": 700, "artist_count": 3}.
- **Challenges Faced & Resolution:**
    - **Challenge:** Calculating three different metrics efficiently without multiple traversals of the playlist.
    - **Resolution:** The generate_summary function was designed to perform only a single pass over the linked list. In each iteration, it updates all three metrics (total time, genre map, and artist set) simultaneously, ensuring optimal

      O(n) performance

---

6. Testing & Validation

Category	Details
Number of Functional Test Cases Written	8 (Each of the 8 implemented modules includes a runnable script demonstrating its core functionality).
Edge Cases Handled	Invalid index handling for deletion/moving , operations on empty or single-node playlists, deletion from head/tail/middle, undoing from an empty history stack.
Known Bugs / Incomplete Features (if any)	None. The solution fulfills all the requirements specified in the core and specialized use case documents.

---

7. Final Thoughts & Reflection

- **Key Learnings from the Hackathon**
  This project was a practical exercise in applying data structures to solve real-world problems. The biggest learning was understanding the trade-offs involved in choosing a data structure—for example, the choice of a Doubly Linked List for O(1) additions over an array's O(1) indexed access. Implementing the lazy reversal was a key insight into how algorithmic choices directly impact user-perceived performance.
- **Strengths of Your Solution**
    - **Performance-Oriented:** Optimal data structures were used for each task (e.g., HashMap for O(1) lookup, lazy flag for O(1) reversal).
    - **Highly Modular:** Each feature is encapsulated in its own class, making the system easy to maintain, test, and extend.
    - **Clean and Documented:** The code is well-documented with comments and clear complexity annotations for every core function, as required.
- **Areas for Improvement**

- **BST Balancing:** The SongRatingTree is a simple BST and is vulnerable to becoming unbalanced (skewed), which would degrade its performance to O(n). With more time, a self-balancing tree like an AVL or Red-Black Tree could be implemented.
- **Data Persistence:** The entire system is in-memory. A future improvement would be to connect it to a database to save and load playlists.
- **Formal Unit Testing:** While each module has a demonstration script, a formal testing suite using a framework like unittest or pytest would ensure greater robustness and catch regressions.

- **Relevance to Your Career Goals**
  This hackathon directly mirrors the challenges of a backend and systems development role. The process of analyzing requirements, designing a system with interacting components, selecting the right data structures for performance, and writing optimized code is the foundation of backend engineering. This project solidifies my understanding of these principles and serves as a practical demonstration of the skills required for such a role.