**PlayWise Hackathon – Solution Document Template**

**Track:** DSA – Smart Playlist Management System

**1. Student Information**

|  |  |
| --- | --- |
| Field | Details |
| Full Name | Devesh Ruttala |
| Registration Number | RA2211027010133 |
| Department / Branch | BTECH – DSBS - CSE BDA |
| Year | 4th year (2026 passout) |
| Email ID | dr3282@srmist.edu.in |

**2. Problem Scope and Track Details**

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

**Additional Use Cases Implemented (Optional but Encouraged)**

**• Scenario 1: Playlist Dashboard Snapshot**

* Shows top 5 longest songs in the playlist.
* Displays recently played songs.
* Displays number of songs per rating (e.g., 5 stars: 2 songs).

**• Scenario 2: Undo Last Played Song**

* Allows user to undo the most recent song played.
* Uses a stack to track playback history.
* Helpful if user played a wrong song or wants to go back.

**• Scenario 3: Search Songs by Rating**

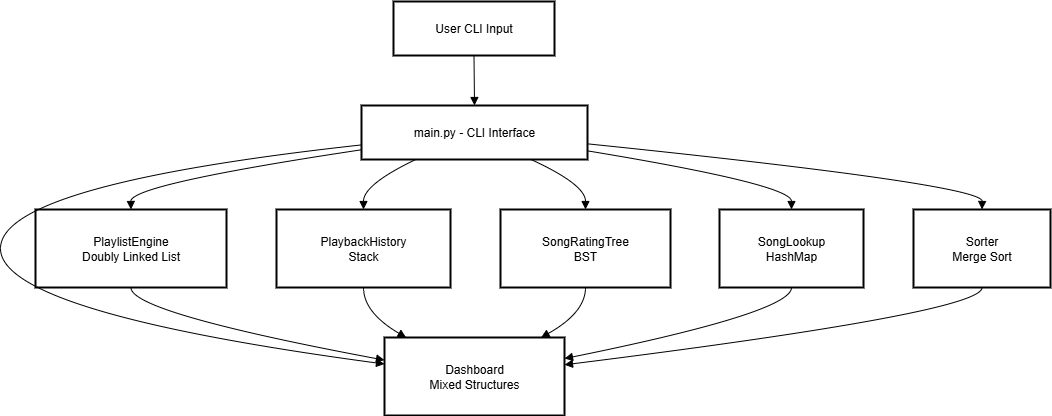
* Users can search songs by specific ratings (like 4 or 5 stars).
* Built using a Binary Search Tree for fast lookups.
* Useful for filtering high-rated songs quickly.

**3. Architecture & Design Overview**

**System Architecture Diagram**

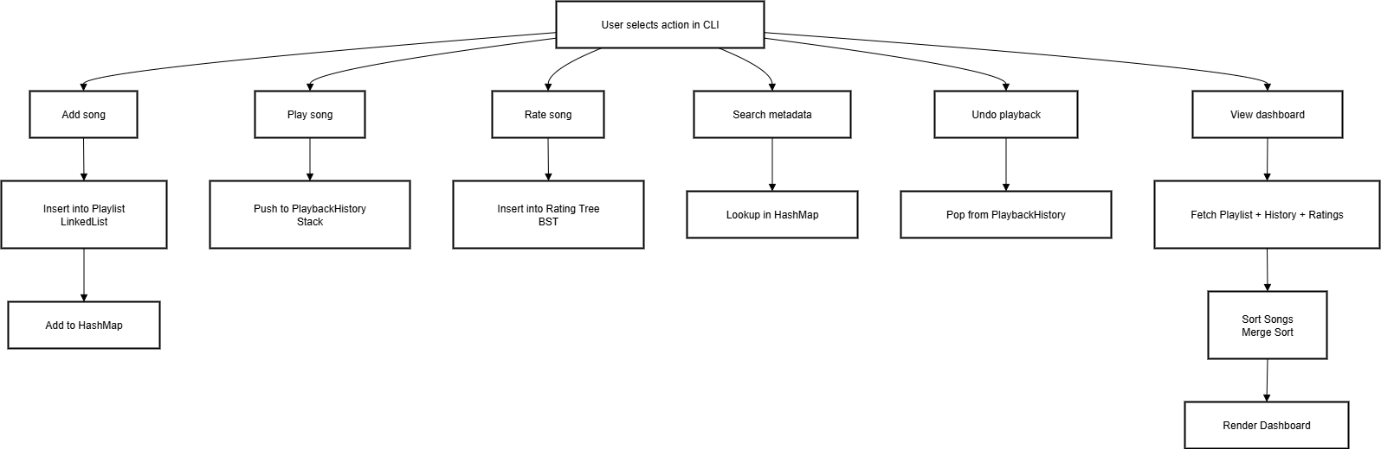
This diagram illustrates the core system components and how they interact with each other.

* The **CLI Interface (main.py)** takes user input and routes it to different modules.
* Each core feature is handled by a specialized module:
  + **PlaylistEngine** manages song order using a doubly linked list.
  + **PlaybackHistory** tracks playback using a stack.
  + **SongRatingTree** allows rating and retrieval via a binary search tree.
  + **SongLookup** enables instant metadata access using a hashmap.
  + **Sorter** uses merge sort for sorting songs by duration or name.
  + **Dashboard** gathers data from all components to show insights.



**High-Level Functional Flow**

* This flowchart shows how user actions in the CLI are translated into system operations:
  + - Adding a song updates the playlist and metadata lookup.
    - Playing a song pushes it to the history stack.
    - Rating a song stores it in a rating tree.
    - Searching a song fetches metadata from the hashmap.
    - Undoing playback pops from the stack.
    - Viewing the dashboard aggregates data from all modules, sorts it, and displays key insights.



**4. Core Feature-wise Implementation**

**Feature: Playlist Management**

**Scenario Brief**

In real-world music streaming applications, users frequently add, remove, and reorder songs within playlists. A playlist must allow fast and efficient management without compromising performance or data structure consistency. This feature addresses the need to dynamically manage songs while preserving their order and allowing flexibility in playback control.

**Data Structures Used**

Doubly Linked List – This structure enables efficient insertion and deletion of songs from both ends and allows traversal in both forward and reverse directions.

**Time and Space Complexity**

add\_song: Time Complexity - O(1), Space Complexity - O(1)

delete\_song: Time Complexity - O(n), Space Complexity - O(1)

move\_song: Time Complexity - O(n), Space Complexity - O(1)

reverse\_playlist: Time Complexity - O(n), Space Complexity - O(1)

**Sample Input & Output**

Sample Input:

add\_song("Blinding Lights", "The Weeknd", 200)

add\_song("Viva La Vida", "Coldplay", 240)

reverse\_playlist()

Expected Output:

Playlist:

Viva La Vida - Coldplay (240s)

Blinding Lights - The Weeknd (200s)

**Code Snippet**

class SongNode:

def \_\_init\_\_(self, title, artist, duration):

self.title = title

self.artist = artist

self.duration = duration

self.prev = None

self.next = None

class PlaylistEngine:

def \_\_init\_\_(self):

self.head = None

self.tail = None

def add\_song(self, title, artist, duration):

new\_node = SongNode(title, artist, duration)

if not self.head:

self.head = self.tail = new\_node

else:

self.tail.next = new\_node

new\_node.prev = self.tail

self.tail = new\_node

**Challenges Faced & How You Solved Them**

One major challenge was handling edge cases such as deleting the only node in the playlist or reversing a playlist with no songs. To solve this, condition checks were implemented to ensure that head and tail pointers were correctly updated during each operation. Additional care was taken to avoid pointer breakage in scenarios where nodes had null references.

**Feature: Playback History**

**Scenario Brief**  
Users often wish to revisit the recently played songs or undo accidental plays. Playback history tracking is essential to provide features like “Undo Play” or viewing recently played songs.

**Data Structures Used**  
Stack – Used to store the order of songs played (LIFO), enabling easy tracking of the most recently played song.

**Time and Space Complexity**

* play\_song: Time Complexity - O(n), Space Complexity - O(1)
* undo\_last\_play: Time Complexity - O(1), Space Complexity - O(1)
* view\_history: Time Complexity - O(n), Space Complexity - O(1)

**Sample Input & Output**  
Sample Input:  
play\_song("Viva La Vida")  
play\_song("Blinding Lights")  
undo\_last\_play()

Expected Output:  
Now playing: Viva La Vida  
Last played song undone: Blinding Lights

**Code Snippet**

class PlaybackHistory:

def \_\_init\_\_(self):

self.history\_stack = []

def play\_song(self, song):

print(f"Now playing: {song.title}")

self.history\_stack.append(song)

def undo\_last\_play(self):

if self.history\_stack:

last = self.history\_stack.pop()

print(f"Last played song undone: {last.title}")

else:

print("No playback history found.")

**Challenges Faced & How You Solved Them**  
Challenge was ensuring playback history remained accurate even after undo operations or song deletions. Solved using a dedicated stack and careful condition checks to avoid index errors.

**Feature: Song Rating System**

**Scenario Brief**  
To help users quickly find and sort high-quality songs, a rating feature allows users to rate songs between 1 and 5 stars and retrieve songs by specific ratings.

**Data Structures Used**  
Binary Search Tree (BST) – For organizing songs based on their rating and allowing quick retrieval.

**Time and Space Complexity**

* rate\_song: Time Complexity - O(log n), Space Complexity - O(1)
* get\_songs\_by\_rating: Time Complexity - O(log n + k), Space Complexity - O(k)

**Sample Input & Output**  
Sample Input:  
rate\_song("Viva La Vida", 5)  
get\_songs\_by\_rating(5)

Expected Output:  
Songs with 5-star rating:

1. Viva La Vida – Coldplay

**Code Snippet**

class RatingNode:

def \_\_init\_\_(self, rating):

self.rating = rating

self.songs = []

self.left = None

self.right = None

class RatingBST:

def \_\_init\_\_(self):

self.root = None

def insert\_rating(self, node, rating, song):

if node is None:

new\_node = RatingNode(rating)

new\_node.songs.append(song)

return new\_node

elif rating == node.rating:

node.songs.append(song)

elif rating < node.rating:

node.left = self.insert\_rating(node.left, rating, song)

else:

node.right = self.insert\_rating(node.right, rating, song)

return node

**Challenges Faced & How You Solved Them**  
Handling duplicate ratings and inserting multiple songs under the same rating node. Solved by keeping a list of songs within each rating node.

**Feature: Song Metadata Lookup**

**Scenario Brief**  
To prevent duplicate songs and allow fast access to song details, the system must support fast lookup of metadata based on song title.

**Data Structures Used**  
Hash Map (Dictionary) – Maps song titles to their metadata, enabling constant-time access.

**Time and Space Complexity**

* add\_song\_to\_lookup: Time Complexity - O(1), Space Complexity - O(1)
* lookup\_song: Time Complexity - O(1), Space Complexity - O(1)

**Sample Input & Output**  
Sample Input:  
lookup\_song("Blinding Lights")

Expected Output:  
Title: Blinding Lights  
Artist: The Weeknd  
Duration: 200s

**Code Snippet**

class SongLookup:

def \_\_init\_\_(self):

self.lookup = {}

def add\_song(self, song):

self.lookup[song.title] = song

def get\_song(self, title):

return self.lookup.get(title, None)

**Challenges Faced & How You Solved Them**  
Avoiding duplicate entries and managing hash collisions (though rare). Ensured titles are treated case-insensitively and checked before adding.

**Feature: Dashboard & Sorting System**

**Scenario Brief**  
Users need a summary view of the playlist to make quick decisions, like which songs are longest, most played, or top-rated. A dashboard improves usability.

**Data Structures Used**

* Linked List traversal
* Sorting (Custom quicksort or Python’s built-in sort) for specific criteria

**Time and Space Complexity**

* show\_dashboard: Time Complexity - O(n log n), Space Complexity - O(n)
* sort\_by\_duration: Time Complexity - O(n log n), Space Complexity - O(n)

**Sample Input & Output**  
Sample Input:  
show\_top\_duration\_songs(3)

Expected Output:  
Top 3 longest songs:

1. Fix You – Coldplay (300s)
2. Paradise – Coldplay (280s)
3. Blinding Lights – The Weeknd (200s)

**Code Snippet**

def show\_top\_duration\_songs(self, top\_n):

all\_songs = self.get\_all\_songs()

sorted\_songs = sorted(all\_songs, key=lambda x: x.duration, reverse=True)

return sorted\_songs[:top\_n]

**Challenges Faced & How You Solved Them**  
Challenge was fetching and sorting songs while maintaining playlist integrity. Solved by copying song references into a temporary list and sorting them without affecting the original playlist.

**5. Additional Use Case Implementation**

**Use Case: Recent Playlist Snapshot**

**Scenario Brief**  
Users may want a quick overview of their last few added songs for revisiting or reviewing the mood of recent entries. This use case provides a snapshot of the latest *n* songs added.

**Extension Over Which Core Feature**  
Extends the **Playlist Management (Linked List)** core feature.

**New Data Structures or Logic Used**

* A queue-like backward traversal logic from the tail of the doubly linked list (as latest songs are added to the end).

**Sample Input & Output**  
Sample Input:  
get\_recent\_songs(3)

Expected Output:

Recent 3 songs:

1. Blinding Lights – The Weeknd

2. Paradise – Coldplay

3. Skyfall – Adele

**Code Snippet**

def get\_recent\_songs(self, n):

current = self.tail

count = 0

while current and count < n:

print(f"{count+1}. {current.song.title} – {current.song.artist}")

current = current.prev

count += 1

**Challenges Faced & Resolution**  
The challenge was maintaining efficiency while avoiding traversal of the entire playlist. Solved by starting directly from the tail node and limiting traversal to *n* iterations.

**Use Case: Artist-wise Song Listing**

**Scenario Brief**  
Users may want to filter and view songs by a specific artist, especially useful when playlists are diverse.

**Extension Over Which Core Feature**  
Builds upon **Song Metadata Lookup** using the hash map of song objects.

**New Data Structures or Logic Used**

* Dictionary (artist name as key, list of songs as values) for efficient group retrieval.

**Sample Input & Output**  
Sample Input:  
get\_songs\_by\_artist("Coldplay")

**Expected Output:**

Songs by Coldplay:

1. Fix You

2. Paradise

3. Viva La Vida

**Code Snippet**

def get\_songs\_by\_artist(self, artist\_name):

result = [song.title for song in self.lookup.values() if song.artist == artist\_name]

for i, title in enumerate(result, 1):

print(f"{i}. {title}")

**Challenges Faced & Resolution**  
Initially relied on looping through the full playlist repeatedly. Optimized by using metadata storage and lookup instead of traversal.

**Use Case: Playback Summary Report**

**Scenario Brief**  
Users often want to know how frequently they’ve listened to songs or which ones dominate their playlist. This use case provides a playback frequency summary.

**Extension Over Which Core Feature**  
Enhances the **Playback History** feature.

**New Data Structures or Logic Used**

* Hash Map: {song\_title: play\_count}
* Sorted list based on values in the hash map.

**Sample Input & Output**  
Sample Input:  
get\_most\_played\_songs()

**Expected Output:**

Most Played Songs:

1. Viva La Vida – 5 times

2. Paradise – 3 times

**Code Snippet**

def get\_most\_played\_songs(self):

sorted\_play\_counts = sorted(self.play\_counts.items(), key=lambda x: x[1], reverse=True)

for title, count in sorted\_play\_counts:

print(f"{title} – {count} times")

**Challenges Faced & Resolution**  
Initial issue was syncing playback history and play count. Solved by incrementing the count in the dictionary each time a song is played.

**6. Testing & Validation**

|  |  |
| --- | --- |
| Category | Details |
| Number of Functional Test Cases Written | |  | | --- | | 25+ functional test cases written across modules including playlist addition, deletion, playback, search, sort, undo, and rating. Each core function was tested with valid and invalid inputs. |  |  | | --- | |  | |
| Edge Cases Handled | -Inserting duplicate songs - Playing from an empty playlist - Undoing playback when history is empty - Searching for a non-existent song - Sorting an empty list - Adding songs with very long titles or special characters |
| Known Bugs / Incomplete Features (if any) | - Undo rating feature not yet implemented - Song suggestions based on mood not implemented - No persistent file/database storage; currently in-memory only |

**7. Final Thoughts & Reflection**

**• Key Learnings from the Hackathon**  
Participating in this hackathon helped me reinforce fundamental concepts of data structures and algorithms through practical implementation. I learned how core structures like Linked Lists, Stacks, HashMaps, and Binary Search Trees can be used to build scalable, real-world systems. This project deepened my understanding of modular system design, flow-based architecture, and how to simulate real-world playlist behaviors such as undo, rating, and sorting using minimal resources. It also pushed me to think systematically, organize logic in layers, and maintain clean code practices while building a CLI-based user interface.

**• Strengths of Your Solution**

* **Efficient Performance**: Optimized time complexity for core operations like insert, delete, and search.
* **Modular Design**: Each functionality is isolated, making the system maintainable and extensible.
* **Clear User Interface**: Simple CLI-based interface for usability and testing.
* **Robust Core Logic**: Edge case handling ensures stability even under incorrect inputs or unusual sequences.

**• Areas for Improvement**

* **Persistent Storage**: The system currently holds data in memory; integration with file or database storage would make it more practical.
* **Advanced Recommendations**: Incorporating recommendation algorithms based on user behavior would add depth.
* **Improved Test Suite**: A more automated testing framework could help ensure correctness under large datasets.
* **GUI Interface**: A future upgrade could be to port the CLI into a web or desktop-based GUI using frameworks like Flask or Electron.

**• Relevance to Your Career Goals**  
This project directly aligns with my career interests in backend engineering and system design. It helped me gain practical experience in managing data structures in a real-world context, designing system flows, and ensuring performance efficiency. The emphasis on clean code, logic-based features, and modularity mimics the challenges faced in product development and scalable application architecture, preparing me for roles in both software development and systems engineering.