**<https://github.com/ArnavG-728/Playwise>**

**Name: Arnav Gupta**

**Reg No.: RA2211028010107**

**Department : Networking and Communications**

**Year: 4th Year**

[**ag7768@srmist.edu.in**](mailto:ag7768@srmist.edu.in)

[**arnavstudies28@gmail.com**](mailto:arnavstudies28@gmail.com)

**Data Structure Trade-offs**

| **Feature** | **Chosen DS** | **Justification** |
| --- | --- | --- |
| Playlist | Doubly Linked List | Fast insertions/deletions; bi-directional traversal for skip/back operations |
| Playback History | Stack (list) | LIFO playback control; enables quick undo of recently played songs |
| Ratings | Binary Search Tree (BST) | Efficient rating-based grouping; supports fast range/rating queries |
| Lookup | HashMap (dict) | O(1) metadata access by song ID or title |
| Sorting | Merge Sort | Stable sort with guaranteed O(n log n) time; good for complex comparison keys |
| Snapshot Export | Dict + Selective Traversal | Lightweight, real-time snapshots with only relevant fields (top 5, last 5, etc.) |
| Performance Analysis | time.time + tracemalloc | Platform-independent tracking of execution time and memory with minimal overhead |
| Artist Blocklist | HashSet (set) | O(1) membership check for blocking/allowing songs during addition/play |
| Duration Visualizer | Linear Scan / Min-Max Tracking | Fast aggregate computation; avoids full sort; memory-efficient |

**Algorithm Justification with Examples**

1. **Doubly Linked List in PlaylistEngine**

**Chosen Over:** Array/List, Singly Linked List  
**Reason:**

* Efficient insert/delete from both ends and middle: O(1) once node is found.
* Bidirectional traversal is needed for operations like moving backward/forward in the playlist.

**Example Scenario:**  
User wants to move song at index 4 to index 1.

* Using a list: O(n) shifting elements
* Using a doubly linked list: detach and reattach nodes → no shifting of elements.

**Trade-off:** Slightly higher memory use per node due to prev pointer.

1. **Stack (List) in PlaybackHistory**

**Chosen Over:** Queue, Deque  
**Reason:**

* LIFO structure perfectly models playback history (most recent played song is undone first).
* Python’s list supports efficient append/pop at the end (O(1)).

**Example Scenario:**  
User accidentally skips to the next song, wants to go back — undo\_last\_play() gives immediate access to the last played song.

**Trade-off:** None significant; simple and optimal.

1. **Binary Search Tree in SongRatingBST**

**Chosen Over:**

* HashMap: Can’t support range queries
* List: Linear search for rating match
* Balanced BST (e.g., AVL, Red-Black Tree): Not necessary unless the tree becomes highly unbalanced

**Why BST:**

* Allows storing multiple songs with same rating in each node.
* Supports fast insertion, lookup, and range queries.

**Example Scenario:**  
Query all songs with a rating of 4 — direct match.  
Want top-rated songs? Traverse rightmost path.

**Trade-off:**

* Worst-case O(n) if unbalanced.
* You can later upgrade this to a self-balancing tree if needed.

1. **HashMap in SongLookup**

**Chosen Over:** List, Set  
**Reason:**

* Provides O(1) lookup for song metadata by either ID or title.
* Extremely efficient and perfect for mapping keys to values.

**Example Scenario:**  
Fetch all metadata about "Shape of You" using either song\_id = s1 or title = "Shape of You" instantly.

**Trade-off:**

* Slightly more memory usage, but acceptable for the performance gain.

1. **Merge Sort in PlaylistSorter**

**Chosen Over:** Bubble Sort, Quick Sort, Heap Sort  
**Why Merge Sort:**

* **Stable Sort:** Keeps the relative order of songs with equal keys (e.g., same duration).
* **Guaranteed O(n log n):** No worst-case slowdowns like Quick Sort.
* **Recursive Divide-and-Conquer:** Elegant for sorting song dictionaries.

**Example Scenario:**  
User wants songs sorted by duration, descending.

*[{"title": "A", "duration": 300}, {"title": "B", "duration": 200}]*

*# Result after sort with reverse=True:*

*[{"title": "A", ...}, {"title": "B", ...}]*

**Trade-off:**  
Uses O(n) additional memory for merging; acceptable for playlists of manageable size.

1. **Time + Memory Analysis in PerformanceAnalyzer**

**Why time.time() + tracemalloc:**

* time.time() gives execution time down to microseconds.
* tracemalloc allows memory tracking without OS dependencies.

**Example Use:**  
Measure how much time and memory is consumed by reversing a playlist of 1000 songs.

1. **Snapshot Export with Selective Traversal in SystemSnapshot**

**Why not full traversal or deep copy?**

* Only essential data (top 5 songs, last 5 played, rating counts) is extracted.
* Keeps snapshot export lightweight and relevant.

**Example:**  
Instead of serializing entire playlist of 1000 songs, you only extract:

*top\_5\_longest = sorted(songs, key=lambda x: x["duration"])[:5]*

**Trade-off:** Not exhaustive, but perfect for real-time dashboards or exports.

1. **Artist Blocklist with HashSet Membership**

**Why use a set() instead of list or database query?**

* set provides **O(1) average-case lookup time**, much faster than list’s O(n).
* Ideal for frequent “is artist blocked?” checks during song addition or playback.
* Avoids expensive database or list scans.

**Example Use:**  
Prevent adding songs by banned artists during dynamic playlist creation.

*if artist in blocked\_artists\_set:*

*skip\_song()*

**Trade-off:** Cannot store metadata about blocked artists (e.g., reason, date blocked) unless you use a dict or class structure. But for fast filtering, set is optimal.

1. **Play Duration Visualizer Using Min/Max Heaps**

**Why Min/Max heaps or linear scan instead of sorting full list?**

* Sorting all songs has O(n log n) complexity.
* Tracking min/max duration during song addition or via heaps gives **O(n)** total time.
* Reduces time and memory usage for large playlists.

**Example Use:**  
Efficiently report summary statistics in a mobile UI:

*total = sum(song["duration"] for song in playlist)*

*shortest = heapq.nsmallest(1, playlist, key=lambda x: x["duration"])[0]*

*longest = heapq.nlargest(1, playlist, key=lambda x: x["duration"])[0]*

**Trade-off:** You get only extremes (min/max), not full order. But ideal for quick summaries in dashboards or exports.

**Pseudocode for Key Algorithms:**

1. **Merge on Playlists**

*function merge\_sort(songs, key, reverse):*

*if len(songs) <= 1:*

*return songs*

*mid = len(songs) // 2*

*left = merge\_sort(songs[:mid], key, reverse)*

*right = merge\_sort(songs[mid:], key, reverse)*

*return merge(left, right, key, reverse)*

*function merge(left, right, key, reverse):*

*result = []*

*while left and right:*

*if (left[0][key] < right[0][key]) XOR reverse:*

*result.append(left.pop(0))*

*else:*

*result.append(right.pop(0))*

*return result + left + right*

1. **BST Insertion**

*function insert\_song(node, rating, song\_id, metadata):*

*if node is None:*

*create new node with rating, insert song*

*return node*

*if rating < node.rating:*

*node.left = insert\_song(node.left, ...)*

*elif rating > node.rating:*

*node.right = insert\_song(node.right, ...)*

*else:*

*append song to node.songs*

*return node*

**Architecture Diagram**

**A diagram of a computer

AI-generated content may be incorrect.**

**Sample Test Cases**

**A screen shot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**