This system is a **Music Playlist Manager** that allows users to manage and organize a playlist of songs. It includes functionalities for adding, removing, sorting, searching, and displaying songs, as well as saving and loading playlists from files. Additionally, the system supports undo and redo actions, as well as a priority queue for managing top-rated songs. Below are the key features and uses of the system:

**Key Features:**

1. **Add Song**: You can add songs to the playlist, and optionally add them to a priority queue as well.
2. **Remove Song**: You can remove a song by its title, and it has undo/redo capabilities.
3. **Display Playlist**: You can display all songs in the playlist, sorted by rating or filtered by a specific rating.
4. **Search Song**: You can search for a song by its title (case-insensitive).
5. **Undo/Redo**: Supports undoing and redoing actions (such as adding or removing a song).
6. **Priority Queue**: Songs can be added to a priority queue, and the top-rated songs can be displayed in the order of their rating.
7. **File Handling**: The program supports loading from and saving to text files for both the playlist and the priority queue.

The system is designed to provide users with full control over their music playlists, with features for managing song information, sorting, searching, and ensuring changes can be easily undone. It also offers file storage, making it easy to save and share playlists. The priority queue provides an efficient way to manage top-rated songs, adding a layer of convenience for music lovers.

**ALGORITHMS USED:**

**1. Linear Search Algorithm** (searchSong)

* **Task**: Search for a song by its title in the playlist.

**Algorithm**:

1. Start from the first song in the playlist.
2. Compare the song's title with the search term.
3. If they match, return the song.
4. If you reach the end of the list and no match is found, return a message saying the song is not found.

**Time Complexity**:

* **O(n)**, where n is the number of songs. You need to check each song until you find a match.

### ****2. Insertion at the End of List**** (addSong)

* **Task**: Add a song to the playlist.

**Algorithm**:

1. Create a new song.
2. Add the song at the end of the playlist.

**Time Complexity**:

* **O(1)** — Inserting at the end of a list is a constant time operation.

### ****3. Deletion from List**** (removeSong)

* **Task**: Remove a song by its title.

**Algorithm**:

1. Start from the first song in the playlist.
2. Compare the title of the song with the one to be removed.
3. If a match is found, remove the song.
4. If the song is not found, return a message saying the song is not in the playlist.

**Time Complexity**:

* **O(n)**, where n is the number of songs. You need to check each song until you find the one to remove.

### ****4. Bubble Sort Algorithm**** (sortByRating)

* **Task**: Sort the songs in the playlist by their ratings.

**Algorithm**:

1. Start with the first song.
2. Compare it with the next song.
3. If the rating of the first song is less than the second, swap them.
4. Move to the next pair of songs and repeat until the playlist is sorted.

**Time Complexity**:

* **O(n²)**, where n is the number of songs. The worst-case scenario occurs when the entire list is unsorted, and we need to perform a swap for each pair.

### ****5. Linear Traversal/Display Algorithm**** (displayPlaylist)

* **Task**: Display all songs in the playlist.

**Algorithm**:

1. Start from the first song.
2. Print the song's details.
3. Move to the next song and repeat until all songs are displayed.

**Time Complexity**:

* **O(n)**, where n is the number of songs. Each song is printed exactly once.

### ****6. Selection Sort Algorithm**** (optional for sorting by ratings)

* **Task**: Sort the songs in the playlist based on their ratings.

**Algorithm**:

1. Find the song with the highest rating in the unsorted part of the playlist.
2. Swap it with the first unsorted song.
3. Repeat the process for the remaining unsorted part.

**Time Complexity**:

* **O(n²)**, where n is the number of songs. Selection sort has a time complexity of O(n²) because it involves nested loops.

### ****7. Simple Stack-Based Undo/Redo Operations****

* **Task**: Implement undo and redo for playlist changes.

**Algorithm**:

* **Undo**: Use a stack to store actions (e.g., adding or removing a song). When undoing, pop the last action from the stack and revert it.
* **Redo**: Use another stack to store undone actions. When redoing, pop an action from the redo stack and apply it again.

**Time Complexity**:

* **O(1)** for both undo and redo operations because they only involve popping from a stack.

### ****8. File I/O (Simple Read and Write Operations)****

* **Task**: Save and load the playlist to/from a file.

**Algorithm**:

* **Saving**: Write each song’s details to a file, one per line.
* **Loading**: Read each line from the file and convert it into a song object.

**Time Complexity**:

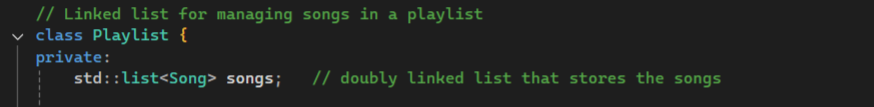
* **O(n)**, where n is the number of songs. Each song needs to be read from or written to the file.

### ****Summary of Algorithms****:

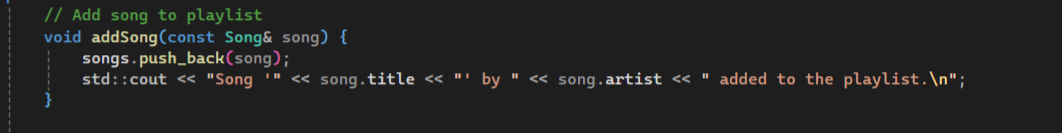
| **Operation** | **Algorithm** | **Time Complexity** |
| --- | --- | --- |
| **Add song to playlist** | **Insertion at the end** | **O(1)** |
| **Remove song from playlist** | **Linear Search + Delete** | **O(n)** |
| **Search for a song** | **Linear Search** | **O(n)** |
| **Sort playlist by rating** | **Bubble Sort / Selection Sort** | **O(n²)** |
| **Display all songs** | **Linear Traversal** | **O(n)** |
| **Undo/Redo action** | **Stack-based undo/redo** | **O(1)** |
| **Save/load playlist to/from file** | **File I/O** | **O(n)** |

### Data Structures Used in the Code:

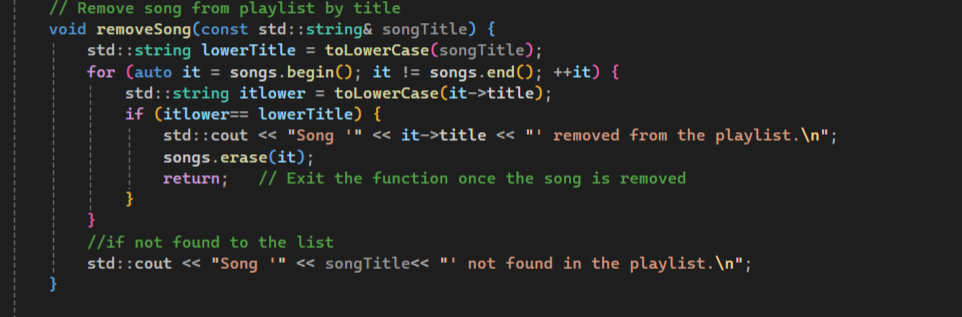
1. **Linked List (std::list)**:
   * **Usage**: The Playlist class uses a doubly linked list to manage a collection of Song objects. Songs can be added, removed, and iterated over efficiently.



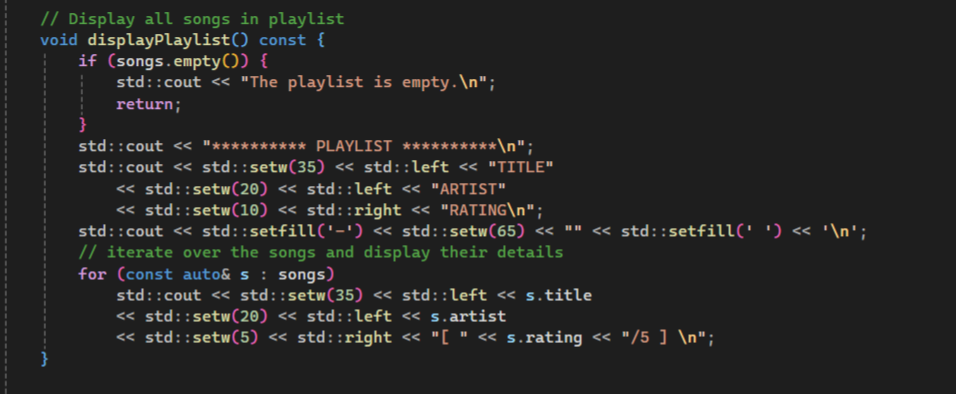
* + **Operations**:
    - **Add song**: O(1) (since std::list::push\_back adds to the end).



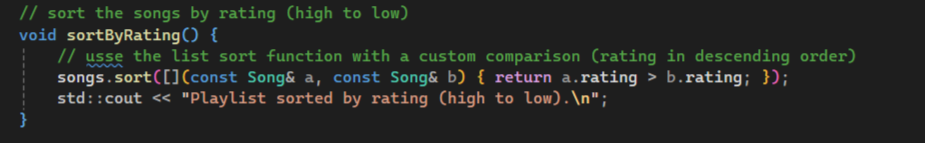
* + - **Remove song**: O(n) (you need to search through the list to find the song by title).



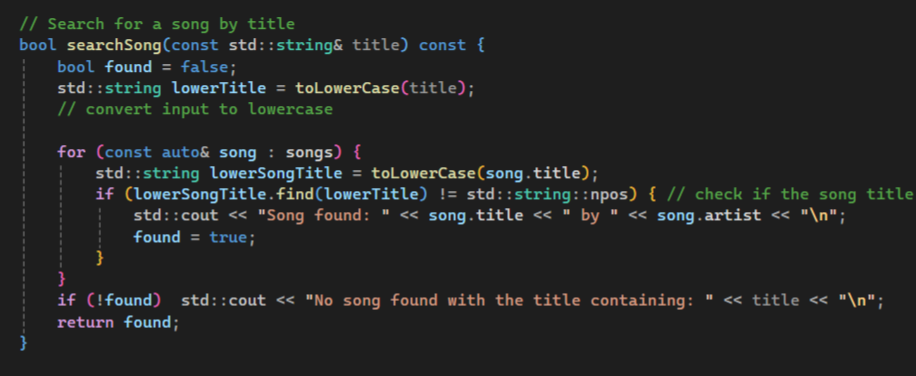
* + - **Display playlist**: O(n) (iterates over all songs to print them).



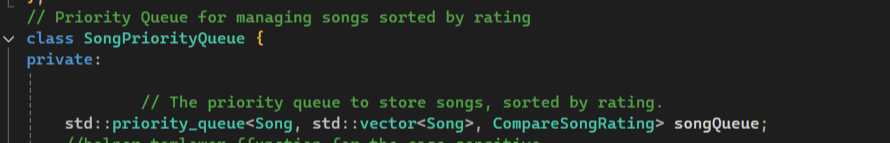
* + - **Sort by rating**: O(n log n) (using std::list::sort with a custom comparator).



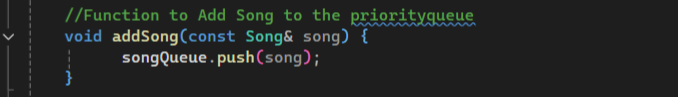
* + - **Search song**: O(n) (linear search through the list).



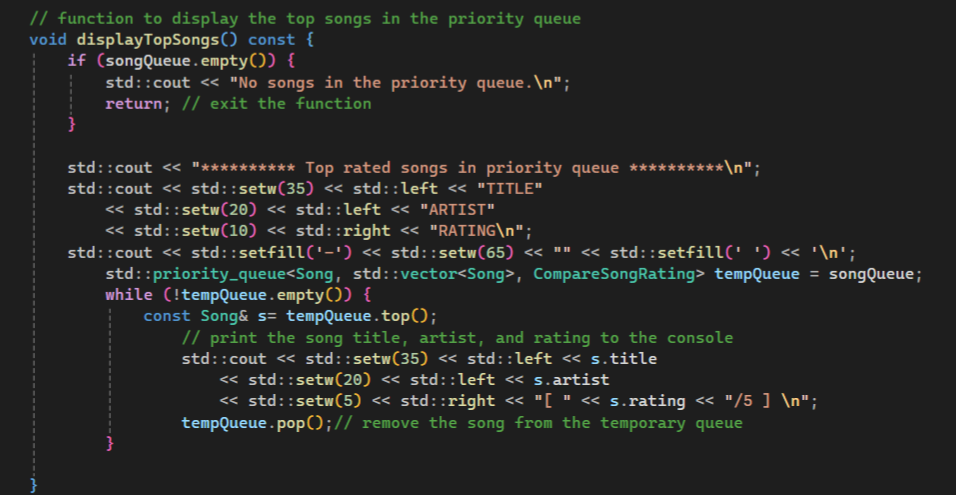
1. **Priority Queue (std::priority\_queue)**:
   * **Usage**: The SongPriorityQueue class uses a priority queue to store songs, which are sorted by their rating (highest first).



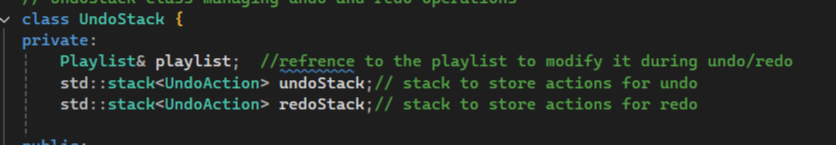
* + **Operations**:
    - **Add song**: O(log n) (inserting into a priority queue requires logarithmic time).



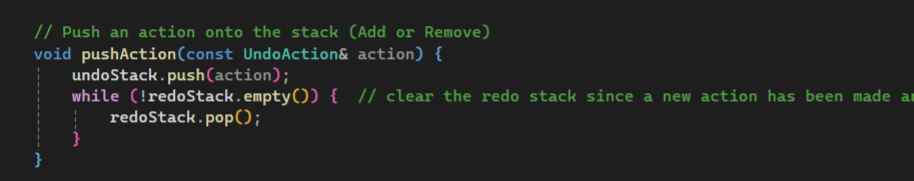
* + - **Display top songs**: O(n log n) (since we need to remove all songs one by one to display them, each removal takes O(log n)).



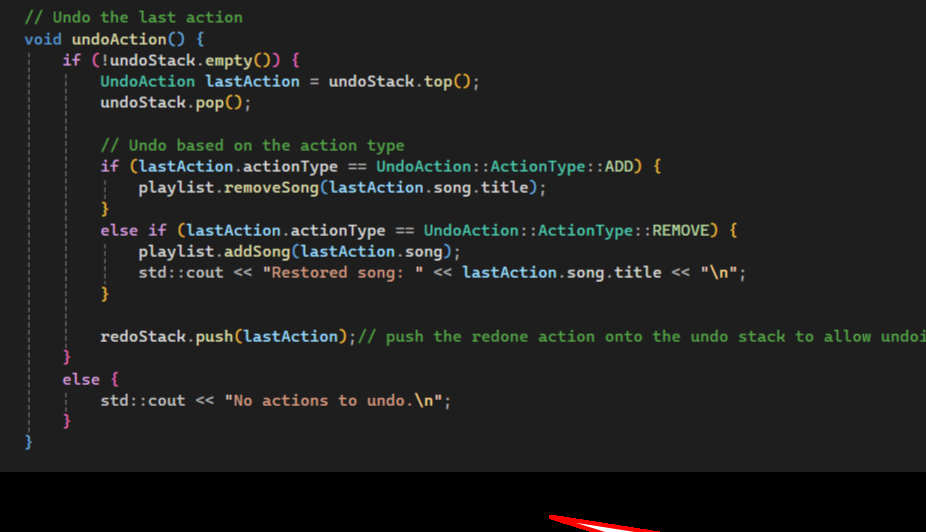
1. **Stack (std::stack)**:
   * **Usage**: The UndoStack class uses two stacks for handling undo and redo operations. Each stack stores an UndoAction object, which contains the action type (add/remove) and the song involved.



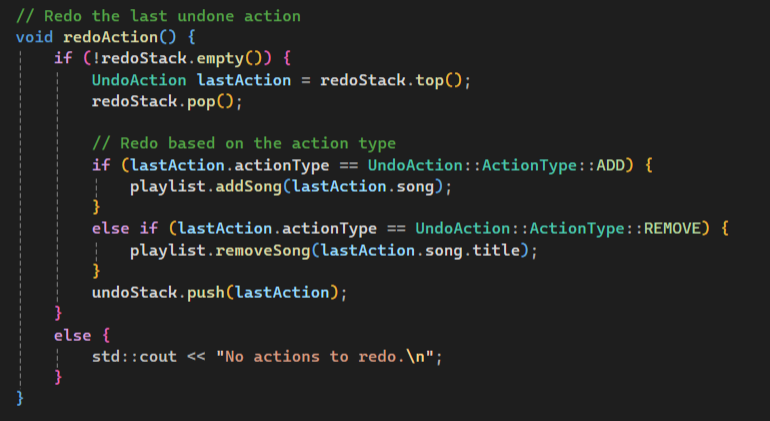
* + **Operations**:
    - **Push action**: O(1) (pushes an action to the stack).



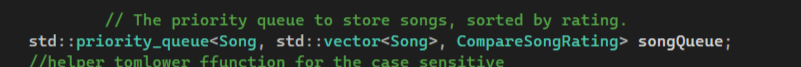
* + - **Undo action**: O(1) (pops the top of the stack and undoes the action).



* + - **Redo action**: O(1) (pops the top of the redo stack and re-applies the action).



1. **Vector (std::vector)**:
   * **Usage**: std::vector is used as the underlying container for the priority queue to hold the songs. The std::priority\_queue uses a vector internally to maintain its heap property.



* + **Operations**:
    - **Insert into the vector**: O(log n) (because the vector is used in conjunction with a heap structure).



1. **String (std::string)**:
   * **Usage**: Used for holding song titles, artist names, and various user inputs throughout the application.
   * **Operations**:
     + **String manipulation** (e.g., lowercase conversion, substring search): O(n) (where n is the length of the string).

