COMP47500 – Advanced Data Structures in Java

Assignment No: 01
Date: 26/02/2025

Title: Playlist Management System using Deque, Stack and List Data Structures

Assignment Type of Submission:			
Group: 16	Yes/No YES	**List all group members' details:** Student Name: Aditi Bhatia Student ID: 24226039 Student Name: Aditi Kailas Student ID: 24213381 Student Name: Shrishti Soni Student ID: 24200374 Student Name: Ishan Bhise Student ID: 24201233	**% Contribution Assignment Workload** 25 25 25 25

1. Problem Domain Description:

The Playlist Management System is designed to efficiently manage and manipulate playlists of songs. It enables users to load songs from a dataset, create playlists, and conduct different operations such as playing the next song, skipping songs, prioritizing songs, shuffling the playlist, and adding new tracks. The system makes use of Java and data structures like Deque (Double-ended Queue), List and Stack to manage the playlist and song history.

The dataset used for this system was sourced from Kaggle (https://www.kaggle.com/datasets/andrewmvd/spotify-playlists) and is based on a subset of users in the #nowplaying dataset who publish their #nowplaying tweets via Spotify. A subset of the dataset was used, focusing on the following columns:
artistname: The name of the artist.
trackname: The title of the track.
playlistname: The name of the playlist that contains this track.

The system is designed to be able to handle large datasets. We have demonstrated this by the stress testing class, which simulates heavy usage scenarios such as adding 10,000 songs, prioritizing 100 songs, shuffling the playlist 100 times, and skipping 500 songs. We optimized this system for performance and made sure to incorporate a user-friendly command line interface for managing playlists.

Key Features:

- **Import Songs from Dataset:** Load songs from a CSV file in which detail information of a song, including artist name, track name, and playlist name, is provided.
- **Select Playlist:** The user selects a playlist from the list of available playlists.
- **Play Next Song:** The next song in the playlist is played.

- Skip Song: Users may skip the present song, which sends it to the end of the playlist.
- **Prioritize Song:** Users would be able to select a particular song and assign it to a position of choice within that playlist.
- **Shuffle Playlist:** The system can randomly shuffle the playlist.
- **Add New Song:** Users would be able to add a new song to the selected playlist.
- **Display Playlist:** The system shows the current playlist with details of each individual song.
- **Display Currently Playing Song:** The system shows which song is being played presently along with the name of the next five songs in the queue.

2. Theoretical Foundations of the Data Structure(s) utilized

2.1 Deque (Double-ended Queue)

A Deque (Double-ended Queue) is a linear data structure that permits insertion and deletion of elements from both ends (front or rear). In other words, it is a combination of a stack (LIFO) and a queue (FIFO). Thus, it allows more versatility. Deque may be considered an abstract data type, generalizing the queue in that the elements may be added or removed from one end or the other..

Operations:

- addFirst(element): Add an element to the front of the Deque.
- addLast(element): Add an element to the rear of the Deque.
- removeFirst(): Remove and return the element from the front of the Deque.
- removeLast(): Remove and return the element from the rear of the Deque.
- peekFirst(): Get the element from the front of the Deque without removing it.
- peekLast(): Get the element from the rear of the Deque without removing it.
- size(): Return the number of elements in the Deque.
- isEmpty(): Check whether the Deque is empty.

Time Complexity:

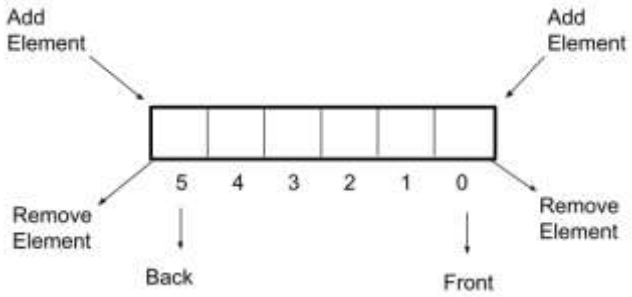
- Insertion and deletion at both ends: O(1)
- Accessing elements: O(1) for front and rear, O(n) for arbitrary positions.

Applications:

- Queue and stack implementations.
- Sliding window problems.
- Management of a playlist, where elements can be added or removed from both ends.

Why is Deque used in our application:

- It supports O(1) time-efficient operations for adding/removing songs to/from either end of the playlist.
- It is flexible for two different ways of processing - queue (FIFO) or stack (LIFO).
- Playlist operations are easy to implement using Deque, resulting in cleaner and maintainable application code.
- Supports dynamic resizing, so it is a good fit for playlists whose sizes can vary.



Source: https://iq.opengenus.org/initialize-deque-in-cpp-stl/

2.2 Stack

Stack is a linear data structure which follows the Last In First Out (LIFO) principles. In the project, history tracking of played songs is the main function of the stack. As the song is played, it is pushed into the stack, thus providing a way for the system to easily pop the last song that was played in case it is ever needed.
A Stack can thus be described as an Abstract Data Type that consists of a collection of elements, chiefly supporting two operations, namely push and pop.

Operations:

- push(element): Adds an element to the top of the Stack
- pop(): Removes and returns the element from the top of the Stack.
- peek(): Retrieves the element at the top of the Stack without removing it.
- size(): Returns the number of elements in the Stack
- isEmpty(): Checks if the Stack is empty.

Time Complexity:

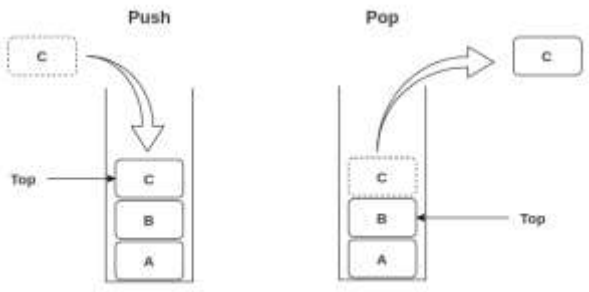
1. Push and Pop are O(1).
2. While accessing any element, O(1) at front and rear, and O(n) for all others.

Applications:

1. Undo in text editors
2. Function call management in programming language
3. Actions history tracking like songs played in a playlist

Why is Stack used in our application:

1. It supports O(1) efficient operations to manage the history of played songs.
2. Easy to implement and understand, suitable for tracking recently played songs.
3. It operates on the last in first out principle that suits the natural behaviour of song history retrieval(the last played song is the first to be called up).



Source: https://www.cgeeksforgeeks.org/stack-meaning-in-dsa/

2.3 List

The List data structure is used to hold and manipulate collections of songs in this program. The List is a dynamic size and allows for very fast access to its elements. Thus, it is suitable for such tasks as prioritizing songs, shuffling the whole playlist, and displaying the playlist. List is just like a collection of elements among which some elements can repeat. It allows access based upon position. It allows insertion/deletion for an element.

Operations:

- add(element): Adds an element to the end of the List
- add(index, element): Adds an element at a specific index
- remove(index): Removes the element at a specific index
- remove(element): Removes the first occurrence of the specified element.
- get(index): Retrieves the element at a specific index.
- size(): Returns the number of elements in the List.
- isEmpty(): Checks if the List is empty.
- shuffle(): Randomly shuffles the elements in the List.

Time Complexity:

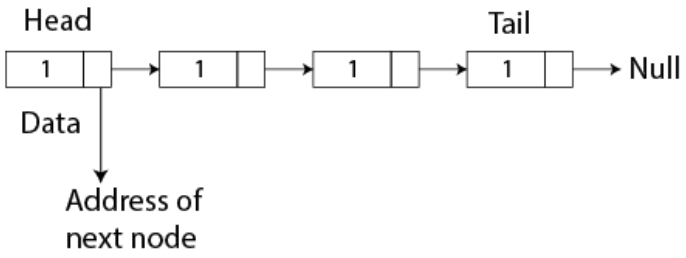
- Insertion and deletion at the end: O(1)
- Insertion and deletion at arbitrary positions: O(n)
- Accessing elements: O(1)

Applications:

- Storing and manipulating collections of data.
- Implementing dynamic arrays.
- Managing playlists where elements need to be accessed or modified at specific positions.

Why is List used in our application:

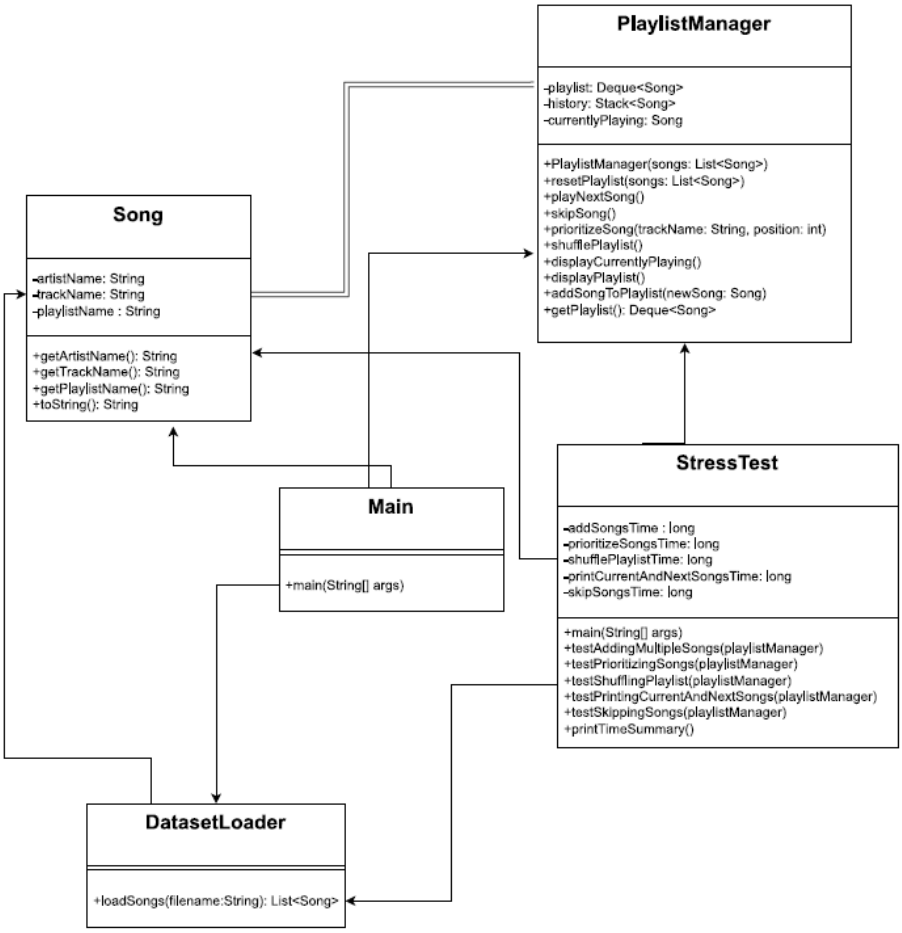
- It is efficient in giving access to elements by index which makes it suitable for operations like prioritizing songs.
- The other nice thing is that it dynamically resizes which makes it easier to work with large datasets.
- It provides utility methods like shuffle(), which are useful for shuffling the playlist.



Source: https://www.prepbytes.com/blog/data-structure/abstract-data-type-adt-in-data-structure/

3. Analysis/Design (UML Diagram(s))

3.1 UML Diagram



Song Class
The Song class represents a song with attributes such as artistName, trackName, and playlistName. It includes a constructor to initialize these attributes and getter methods to retrieve them. The toString() method is overridden to provide a string representation of the song.

DatasetLoader Class
The DatasetLoader will read song records from a CSV file into a line-by-line manner, parse them, and create Song objects, including edge cases such as missing or malformed data.

PlaylistManager Class
The PlaylistManager class is the engine of the system. The entire playlist is maintained by using Deque, while Stack is used to keep track of played songs. It has many methods such as play, skip songs, prioritize songs, shuffle the playlist, and add songs.

Main Class
The Main class serves as the starting point of the application. It loads the songs from the dataset, the available playlists, and allows the user to access the same. It also takes in user input and then calls the necessary method from the PlaylistManager class.

StressTest Class
The StressTest class is used to evaluate the performance of the system under heavy load. It simulates scenarios such as adding 10,000 songs, prioritizing 100 songs, shuffling the playlist 100 times, and skipping 500 songs. The timings of each operation is measured and the results are then summarized for the user.

3.2 Analysis of Algorithms

Different algorithms are implemented in the Playlist Management System to perform functions such as adding songs, putting some songs ahead of others, shuffling the playlist, and skipping the songs. Below is an analysis of the algorithms used and suggestions for tuning their practical efficiency.

3.2.1 Adding Songs

Adding songs into the playlist is implemented using the addLast() method of the Deque. The time complexity for the addition of any single song is O(1).

Analysis: Adding 10,000 songs sequentially takes O(n) time, where n is the number of songs. The Deque's dynamic resizing ensures efficient memory usage, but frequent resizing can lead to occasional performance overhead.

Tuning Practical Efficiency:
- Memory pre-allocation: Pre-allocate memory to the Deque if it is known what the maximum size of the playlist would be so that overheads of rescaling would not be incurred.
- Batch processing: Add songs in batches to minimize resizing operations.

3.2.2 Prioritizing Songs

In prioritizing songs, the user changes song position on the playlist using the add(index, element) method of the List. The time complexity for prioritizing a song is O(n) in the worst case, as elements may need to be shifted.

Analysis: Prioritizing 100 songs in a playlist of 10,000 songs can prove to be computationally expensive due to the O(n) complexity. For small playlists, this operation works efficiently, yet for bigger datasets, this could become a serious bottleneck.

Tuning Practical Efficiency:
- Use a Priority Queue: Work on a priority queue structure, where you may use a Min heap for managing song priorities. This way, the time for prioritization is reduced to O(log n).
- Caching: Cache frequently prioritized songs to avoid reusing the same operations.

3.2.3 Shuffling the Playlist

The playlist is shuffled using the shuffle() method of the List, which randomizes the order of elements. The time complexity for shuffling is O(n), where n is the number of songs.

Analysis: Shuffling a playlist of 10,000 songs 100 times takes O(n * m), where m is the number of shuffles. The operation is efficient on its own, however, multiple runs can severely hamper performance on really large playlists.

Tuning Practical Efficiency:
- Partial Shuffling: Shuffle only a subset of the playlist (e.g., the first 1,000 songs) to reduce computation time.
- Optimized Randomization: Introduce other randomization algorithms with better efficiency than the present one, such as the Fisher-Yates shuffle, which guarantees O(n) time with minimal overhead.
- Caching Shuffled Playlists: Cache shuffled versions of the playlist to avoid re-computation.

3.2.4 Printing Current and Next Songs

Through the get(index) method of the List, the index of the current song and the next 5 songs is retrieved. Accessibility of elements by index takes O(1) time.

Analysis: Printing the current and next 5 songs 100 times has a time complexity of O(m), where m is the number of print operations. This operation is extremely efficient due to constant time access of List.

Tuning Practical Efficiency:
- Batch Retrieval: Retrieve multiple songs in a single operation to reduce the number of method calls.

3.2.5 Skipping Songs

Songs are being skipped by the function removeFirst() from the Deque, which removes the current song and moves it to the end of the playlist. The time complexity for skipping a song is O(1).

Analysis: Skipping 500 songs in a playlist of 10,000 songs takes O(m), where m is the number of skips. The operation is efficient due to the Deque's constant-time removal and insertion.

Tuning Practical Efficiency:
- Batch Skipping: To reduce the amount of method calls, multiple songs can be skipped in one go.
- Lazy Removal: Mark songs as "skipped" and remove them in batches to reduce overhead.

4. Code Implementation

4.1 GitHub (link): https://github.com/aditib26/Group-16-Advanced-DSA-in-Java-Assignment-01

4.2 Set of Experiments and Results

4.2.1 Testing of Individual Operations

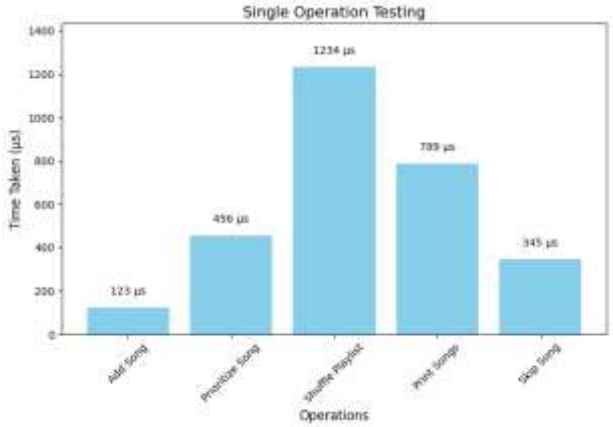
The following experiments were conducted to evaluate the performance of individual operations in the Playlist Management System. Each operation was tested in isolation to measure its execution time in microseconds (µs). The results are summarized in the table below

Experiment	Time Taken (µs)	Observations
Adding a single song	123	Efficient, almost instantaneous
Prioritizing a Single Song	456	Slightly slower due to element shifting
Shuffling the Playlist Once	1234	Longer due to the randomization process
Printing Current and Next 5 Songs	789	Efficient, constant-time access
Skipping a Single Song	345	Highly efficient, constant-time operation

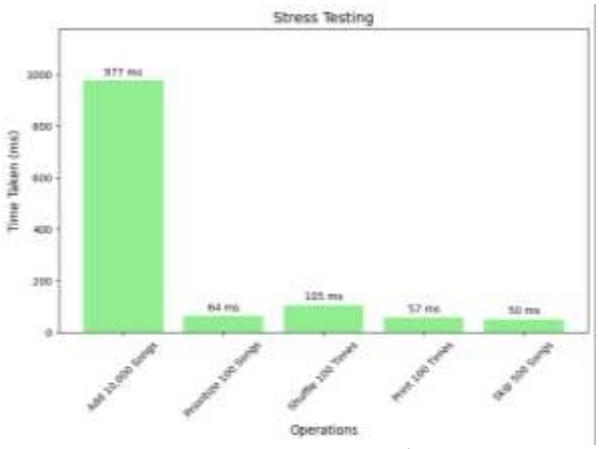
4.2.2 Testing of Bulk Operations (Stress Testing)

Stress Testing is a type of performance testing that evaluates how a system behaves under extreme conditions, such as high load, limited resources, or overwhelming data volumes. The goal is to identify the system's breaking points, ensure stability, and verify that it can handle peak usage without crashing or degrading performance.
These stress tests were designed to aid in assessing the performance of the Playlist Management System under extreme load. The tests simulate real-life circumstances where the system needs to perform large operations, like adding thousands of songs, prioritizing several songs, repeatedly shuffling the playlist, and skipping hundreds of songs. The results are summarized in the table below, along with observations highlighting the system's scalability and performance under stress.

Experiment	Time Taken (ms)	Observations
Adding 10000 songs	977	Efficient for bulk addition, linear scaling
Prioritizing 100 songs	64	Slightly slower due to element shifting
Shuffling the Playlist 100 times	105	Efficient randomization, minimal overhead
Printing Current and Next 5 Songs 100 times	57	Fast and consistent, suitable for real-time use
Skipping 500 songs	50	Highly efficient, constant-time operation



Bar graph showing time taken for a single operation



Bar graph showing the time taken for bulk operations

4.3 Implementation Snippets and Screenshots

The user will first have an option to choose from multiple playlists which are available after choosing he will then be able to manage the playlist by using the following operations:

```
Enter playlist number (or 0 to exit): 11
Selected Playlist: Starred

Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
```

Option 1: Play next song : Plays next song in the playlist and updates the history stack

```
public void playNextSong() {
    if (!this.playlist.isEmpty()) {
        if (this.currentlyPlaying != null) {
            this.history.push(this.currentlyPlaying);
        }

        this.currentlyPlaying = (Song)this.playlist.poll();
        System.out.println("Now Playing: " + this.currentlyPlaying);
    } else {
        System.out.println("No more songs in the playlist.");
    }

}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 1
Now Playing: (Theme From) Red Dead Redemption by Bill Elm & Woody Jackson
```

Option 2: Skip song: Skips the current song and moves it to the end of the playlist.

```
public void skipSong() {
    if (!this.playlist.isEmpty()) {
        Song var1 = (Song)this.playlist.poll();
        this.playlist.offer(var1);
        System.out.println("Skipped: " + var1);
    }

}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 2
Skipped: 22 - Grace Omega (Main theme) by Jin Roh
```

Option 3: Prioritize a song at a specific position: Moves a specific song to a desired position in the playlist.

```
public void prioritizeSong(String var1, int var2) {
    ArrayList var3 = new ArrayList(this.playlist);
    Song var4 = null;
    Iterator var5 = var3.iterator();

    while(var5.hasNext()) {
        Song var6 = (Song)var5.next();
        if (var6.getTrackName().equalsIgnoreCase(var1)) {
            var4 = var6;
            break;
        }
    }

    if (var4 != null) {
        this.playlist.remove(var4);
        var2 = Math.max(0, Math.min(var2, this.playlist.size()));
        ((LinkedList)this.playlist).add(var2, var4);
        System.out.println("Moved " + var4 + " to position " + (var2 + 1));
    } else {
        System.out.println("Song not found.");
    }

}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 3
Enter track name to prioritize: My Way
Enter new position (1-based index): 3
Moved My Way by Nas to position 3
```

The song is moved to the third position

```
Currently Playing: (Theme From) Red Dead Redemption by Bill Elm & Woody Jackson

Up Next (Next 5 songs):
1. 07 - echoes by
2. 22 - Grace Omega (Main theme) by Jin Roh
3. My Way by Nas
4. 5:00 AM by SomethingALaMode
5. 9,000 Miles by Pendulum
```

Option 4: Shuffle playlist: Randomizes the order of songs in the playlist.

```
public void shufflePlaylist() {
    ArrayList var1 = new ArrayList(this.playlist);
    Collections.shuffle(var1);
    this.playlist.clear();
    this.playlist.addAll(var1);
    System.out.println("Playlist shuffled!");
}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 4
Playlist shuffled!
```

Playlist successfully shuffled

```
Currently Playing: (Theme From) Red Dead Redemption by Bill Elm & Woody Jackson

Up Next (Next 5 songs):
1. Partir de Cero by Ana Tijoux
2. Bonus Track by Various Artists
3. Let's Push Things Forward by The Streets
4. Live On - BBC Mark Goodier Session by Pulp
5. The Monster by Eminem
```

Option 5: Show playlist: Displays the current playlist with song details.

```
public void displayPlaylist() {
    if (this.playlist.isEmpty()) {
        System.out.println("Playlist is empty.");
    } else {
        System.out.println("Current Playlist:");
        int var1 = 1;
        Iterator var2 = this.playlist.iterator();

        while(var2.hasNext()) {
            Song var3 = (Song)var2.next();
            int var10001 = var1++;
            System.out.println("" + var10001 + ". " + var3);
        }
    }

}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 5
Current Playlist:
1. (Theme From) Red Dead Redemption by Bill Elm & Woody Jackson
2. 07 - echoes by
3. 22 - Grace Omega (Main theme) by Jin Roh
4. 5:00 AM by SomethingALaMode
5. 9,000 Miles by Pendulum
6. A Proper Story by Darren Korb
7. Abesses - Beat Torrent Remix by Birdy Nam Nam
8. Aerius Light (Kitsun?? Dj-Friendly Edit) by Data
9. Afterburner by Dubmood
10. Again - Live Z??nith Paris by Archive
11. Alice Practice by Crystal Castles
12. Always Waiting by Michael Kiwanuka
13. Angst One by The Toxic Avenger
14. Any Day Will Do Fine by Michael Kiwanuka
15. Bad Things - Soundtrack Version by Jace Everett
16. Bitblaster by Zalza vs. Alk
17. Bones by Michael Kiwanuka
18. Build That Wall (Zia's Theme) by Darren Korb
19. Can't Stop Us by Chipzel
20. Celestica by Crystal Castles
21. Christmas Eve Montage by RJD2
22. Chrono Trigger (Chrono Trigger) by The OneUps
```

Option 6: Show currently playing song and next 5 songs: Displays the currently playing song and the next 5 songs in the queue.

```
public void displayCurrentlyPlaying() {
    if (this.currentlyPlaying != null) {
        System.out.println("Currently Playing: " + this.currentlyPlaying);
    } else {
        System.out.println("No song is currently playing.");
    }

    System.out.println("\nUp Next (Next 5 songs):");
    Iterator var1 = this.playlist.iterator();

    for(int var2 = 0; var1.hasNext() && var2 < 5; ++var2) {
        System.out.println(var2 + 1 + ". " + var1.next());
    }

}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 6
Currently Playing: Let's Push Things Forward by The Streets

Up Next (Next 5 songs):
1. Live On - BBC Mark Goodier Session by Pulp
2. The Monster by Eminem
3. Bookends by Simon & Garfunkel
4. If We Ever Meet Again - Digital Dog Radio Remix by Timbaland
5. The Dreamer by The Tallest Man On Earth
```

Option 7: Add a new song to this playlist: Add a new song to the end of the playlist.

```
public void addSongToPlaylist(Song var1) {
   this.playlist.offer(var1);
   PrintStream var10000 = System.out;
   String var10001 = var1.getTrackName();
   var10000.println("Added new song: " + var10001 + " by " + var1.getArtistName());
}
```

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 7
Enter song name: Rude
Enter artist name: MAGIC!
Added new song: Rude by MAGIC!
```

Song added successfully

```
2137. Sun of a Gun by Oh Land
2138. Wasting My Young Years by London Grammar
2139. It's Growing - Album Version / Stereo by The Temptations
2140. Royals by Lorde
2141. I Don't Feel Like Dancin' - (Linus Loves Vocal Edit) by Scissor Sisters
2142. Build That Wall (Zia's Theme) by Darren Korb
2143. Heard 'Em Say - Live - Abbey Road Studios by Kanye West
2144. Lucy Pearl's Way by Lucy Pearl
2145. Handshake The Gangster by Hey Rosetta!
2146. Rude by MAGIC!
```

Option 8: Return to playlist selection: Return to the playlist selection menu

```
Options:
1. Play next song
2. Skip song
3. Prioritize a song at a specific position
4. Shuffle playlist
5. Show playlist
6. Show currently playing song and next 5 songs
7. Add a new song to this playlist
8. Return to playlist selection
9. Exit
Enter choice: 8
Returning to playlist selection...

Available Playlists:
1. HARD ROCK 2010
2. IOW 2012
3. 2080
4. C418
5. Chill out
6. Classique
7. Daft Punk
8. Electro
9. Ghibli songs
10. Soir??e
```

Video of the Implementation: https://drive.google.com/file/d/1ACj8GnQE-lFAn6M-W4IERLkZ__kSJhoo/view?usp=sharing