



COURSEWORK 2 REPORT

Karaoke Application

CST2550 – Software Engineering Management and Development

08/05/2020

Table of Contents

Abstract	3
Karaoke Application.....	3
Introduction	4
Brief description	4
Layout of the report	4
Design.....	5
Pseudo Code	5
Add song(s) to song library	5
Search song library for a song by title	5
Add song to end of playlist.....	6
Play next song in (and remove from) playlist.....	7
Retrieve song in playlist (to view).....	8
Delete song from playlist.....	9
Analysis of time complexity of solution (HashMap)	9
Put and Get operation	9
GUI mock-ups/wireframe diagrams	10
The main page	10
The media player	10
The full screen media player	11
Add a song to the library	11
The song and playlist page	12
The settings.....	12
Testing	13
Description of testing approaches used	13
Evidence of testing (The description of each test)	13
Add song to song library (Appendix A)	13
Search song library for a song by title (Appendix B).....	13
Add a song to the end of a playlist (Appendix C)	13
Play next song in (and remove from) playlist (Appendix D)	13
Retrieve song from playlist to view (Appendix E).....	13
Delete song from playlist (Appendix E).....	14
Evidence of testing (Screenshot from the terminal)	14
Conclusion.....	14
Summary of work done	14
Limitations of approach and critical reflection of work	15
How would I approach a similar project differently in the future?	15

References	16
Appendices	17
Appendix A	17
Appendix B.....	17
Appendix C.....	17
Appendix D	18
Appendix E.....	18
Appendix F	18

Abstract

Karaoke Application

This project seeks to analyse data structures in Java programming language that correlate to the performance of the application in terms of time complexity and storage. A well-designed graphical user interface is build using JavaFX and a fully functional back-end is implemented into the program. This program can be used by anyone, is simple to understand and has a good performance. The end result is a Karaoke Application which is complete and running without bugs and errors.

Introduction

Brief description

This project is mostly about research on data structures. The aim is to be able to properly choose and use the most appropriate data structure to store data in an application. In this application, I need to design and program an application, a Karaoke Application, which shall import data from a file and put the data in a chosen data structure. The user shall be able to choose song(s) from this list and insert it into a playlist of his own. This playlist will be played in order. The program should contain the basic functions of a media player and once a song has been played, it should be deleted from the playlist.

Layout of the report

This report contains important data about the project. The project is about a Karaoke application. The report contains:

1. Pseudo code of the main functionalities
2. Analysis of time complexity
3. GUI Mock-Ups / wireframes
4. Description of testing approaches used
5. Evidence of testing
6. Summary of work done
7. Limitations of my approach and critical reflection of my work
8. How I would approach a similar project differently in the future
9. References
10. Appendices

Design

Pseudo Code

Add song(s) to song library

Read from file () {

 Create a new hash table with key as String and value as Song object

 While (line in file is not empty) {

 String array containing the data of the line split by “tab”

 Create a song object with the data in the string array

 Put the song object in the hash table (song name as key, song object as value) {

 If (number of keys \geq 10 times no. of chains) resize (2 times no. of chains) {

 Create temporary hash table with 2 times the previous size

 For (all nodes which are not empty) {

 Put the old key and value into the temporary hash table

 }

 Replace the old hash table with the temporary hash table

 }

 Generate a hash code with the input key

 For (all nodes which are not empty) {

 If (key of node == generated key) {

 Add the song value to the key

 end

 }

 Increase the size of table by 1

 Populate the hash table with the key and value

 }

 }

 Read next line in file

 }

 Return the hash table

}

Search song library for a song by title

Search for song by title (song hash table, search criteria) (

```

Create an observable list with datatype as the Song object
For (each item in the songs hash table, use the key of the node) {
    If (the key matches the search criteria) {
        Get the song object (key) {
            Generate a key for the key
            For (all nodes which are not empty) {
                If the key of the node matches the generated key {
                    Return the value of this node
                }
            }
            If there is no match, return null
        }
    }
    Add the value to the observable list
}
}
}

```

Add song to end of playlist

```

User selects a row (containing the song) which he wants to add to playlist
Get the selected object from the table's observable list
Get the song name from the selected object
Add to playlist file (selected song name) {
    Choose the playlist.txt file
    Create a buffered writer
    If (the file is empty) write song name on the same line
    Else write song name on new line
}

```

Play next song in (and remove from) playlist

On “next” button press or end of media {

Next media action (songs hash table, playlist linked list) {

If (there is more than 1 song in the playlist) {

Stop the media player

Dispose the media player

Select the first row in the playlist table and get the index

Delete from playlist (index 0) {

Get the playlist into a linked list

Remove index 0 from the playlist

Overwrite the playlist file with the new linked list

}

Refresh the playlist table

Select the first item in the playlist table

Get the matching object in the songs hash table

Get the video file using the song.GetFileName()

Create a new media and media player

Set the media player into media view

Play the media player

If (mute state Boolean is true) set the media player to mute

Set the time slider to zero and set the max value to the video length

Set the media to the value of the volume slider

Add the “next media action” to the new media player

Set the song title and artist name using the song hash table

} else {

Stop the media player

Dispose the media player

Close the player stage

Clear the playlist file

}

}

}

Retrieve song in playlist (to view)

Refresh the playlist () {

 Create a linked list <String>

 Get the linked list from importData.getPlaylist() {

 Create a linked list <String>

 Read file playlist.txt

 Create a buffered reader

 While (line in text file is not empty) {

 Add the line to the playlist linked list (line data) {

 Define a node with the last position

 Create a new node with the last item and the user data

 Assign last to the new node

 If (the last is empty) assign the node to the first position

 Else assign the node after the last existing node

 Increase the size of the linked list playlist by 1

 Increase the number of elements in the linked list by 1

 }

 Read the next line

 }

 Return the playlist linked list

}

Clear the playlist table

For (all items in the playlist linked list) {

 Create a Song object with the song name from the linked list

 Add the song object to the playlist table

}

}

Delete song from playlist

```
Delete from playlist (index to delete) {  
    Remove from playlist (index to delete) {  
        Remove data from the node of the index  
        Decrease all node value by 1 index  
        Delete the last node  
        Return the playlist  
    }  
    Update the playlist.txt file  
}
```

Analysis of time complexity of solution (HashMap)

HashMap is useful for solving problems due to its $O(1)$ time complexity for both get and put operations.

HashMap works on Hashing principle and use hash code as base for storing key-value pair. HashMap distribute objects across its nodes in such a way that the HashMap puts the objects and retrieve them in a constant time of $O(1)$.

Put and Get operation

The put operation starts by computing the hash code of the key, calculate the array index from hash code and then move to the calculated node to see if there is any key-value pair present.

If key-value pair is found, the node's value will be replaced.

If there is no matching key-value, it will go to the end of the list and create a new key-value.

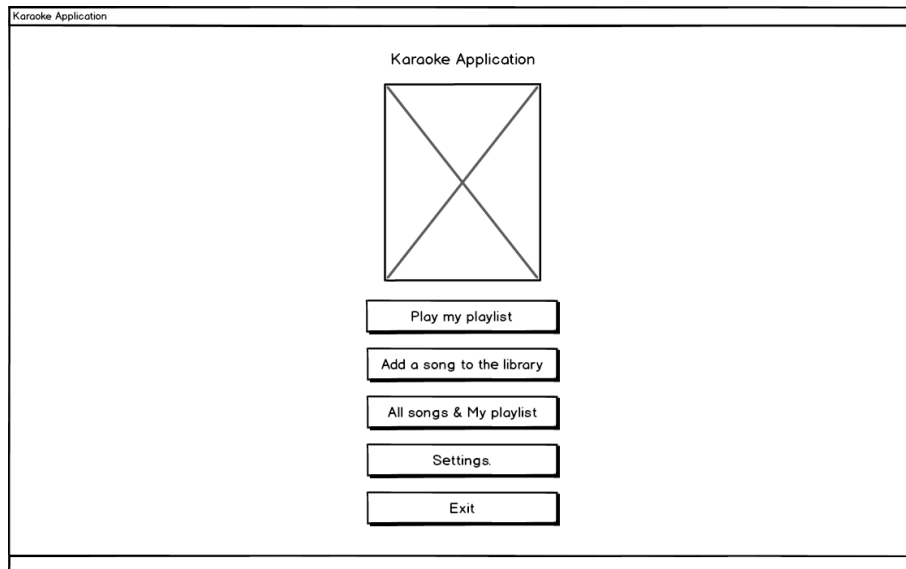
HashMap put and get operation time complexity is $O(1)$ assuming that the key-value pairs are well distributed across the buckets.

HashMap best and average case for Search, Insert and Delete is $O(1)$ and its worst case is $O(n)$.

GUI mock-ups/wireframe diagrams

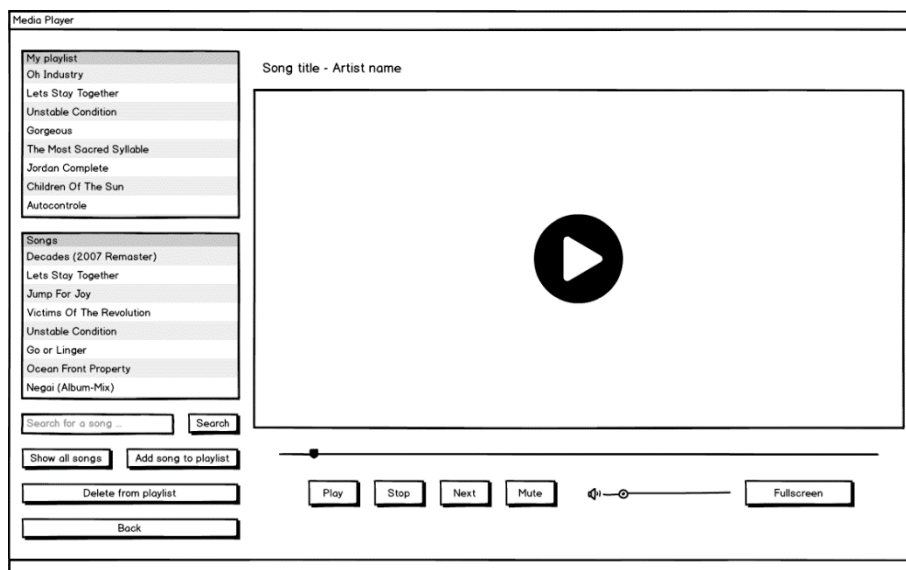
The main page

This is the main page of the Karaoke Application. On this page, there is a label at the top with an image just below it. There are also 5 buttons; to play the playlist, add a song to the library, show all songs and the playlist, settings and exit respectively.



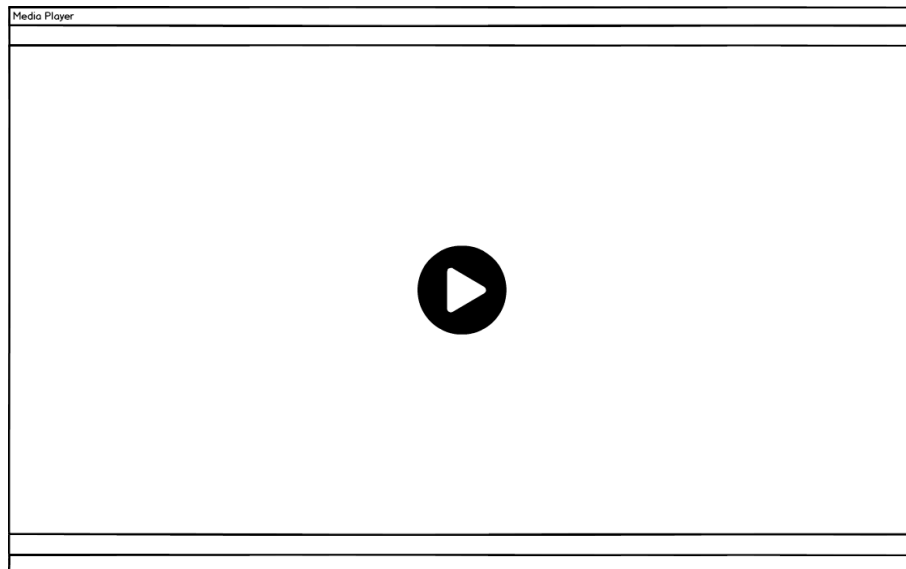
The media player

This player page contains the playlist, the songs list, a search field and button, and means to add and remove songs from the playlist. There is a media view above which there is a text which displays the song name and artist name. Below the media view, there is a slider which controls the seek of the media player. There are the play/pause, stop, next, mute/unmute and Fullscreen button. There is also a volume slider.



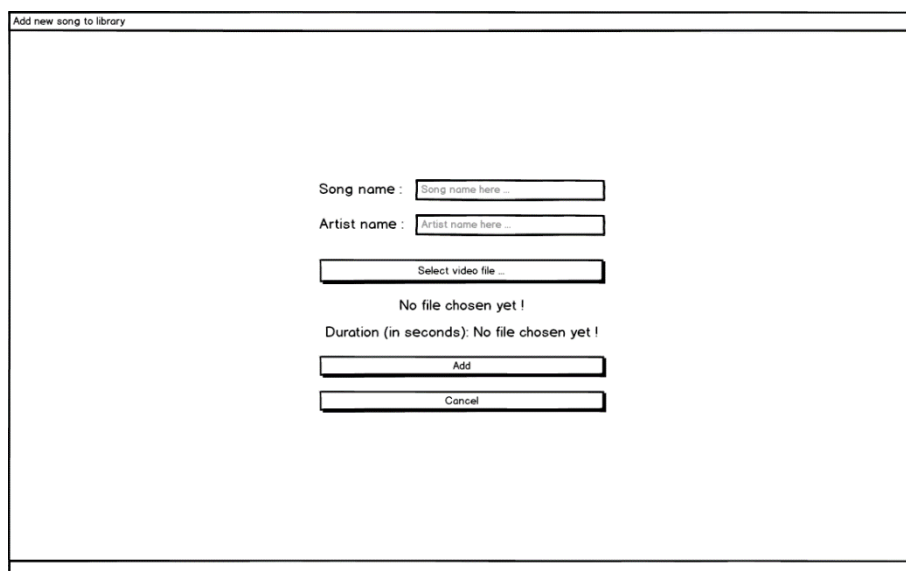
The full screen media player

On clicking on the Fullscreen button on the media player page, the scene will change to allow the media view to take the full size of the screen. On this page, the only way to switch back to the normal player is by pressing the “ESCAPE” key on the keyboard.



Add a song to the library

On this page, the user can add a new song to the library. The user has to enter the song name, the artist name and select the video file. The system will calculate the duration automatically and also copy the video to the software folder.



The song and playlist page

On this page, the user can see the full song library list with their respective artist's name and the song duration. A song can be selected and added to the playlist. The playlist table is found on the left. A search field and buttons are found below the playlist table on the left for the user to use the page.

All songs & My playlist			
My playlist Oh Industry Lets Stay Together Unstable Condition Gorgeous The Most Sacred Syllable Jordan Complete Children Of The Sun Autocontrolle Go Widdit Neighbor Tic Tac Toe No Disturb	Song Name	Artist Name	Song Duration (seconds)
	Slow Stone	Balmorhea	396
	B	Opusculus	333
	Consider This	Filter	258
	Necessary Death	End	173
	G	Opusculus	122
	I'm Sick Y'All (Mono, 2016 Remaster)	Otis Redding	177
	M	Opusculus	180
	O	Shygirl	147
	Jim Henson's Zombie Hand	Mithridate	245
	Children Of The Revolution	T. Rex	150
	This Town	Kacey Musgraves	177
	Experiment (Relax Piano Music)	Deep Sleep Music Delta Binaural 432 Hz	225
	Crack The Bell	Wall of Voodoo	215
	V	Opusculus	107
	W	Opusculus	150
	Gerudo Valley (From the Legend of Zelda: Ocarina of Time)	The 8-Bit Big Band	272
Search for a song ... Search Show all Add to playlist Remove from playlist Empty my playlist Back	X	Opusculus	155
	Artefacts	Molecule	390
	Jordan	Complete	167
	Field Party (feat. JJ Lawhorn)	The Lacs	263
	The of Video Game	Without A Care	263
	Flying Practice	Gui Boratto	284
	I'll Be Your Home	Drop Out Vegas	220
	La famiglia consonanti	Le mele canterine	144
	Sourwood Mountain	Ed Holey	171
	Consider Yourself (Remastered)	Lionel Bart	242
	Other Worlds	Trivium	289
	Téro Lonj	Mayra Andrade	263
	BFG Division (From "Doom")	BillyTheBard11th	260

The settings

On this settings page, the user can choose where he wants the playlist pane to appear on the media player page and the view all songs page (either left or right). The user can also choose which User Interface mode (UI mode) he wants to use (either light or dark). The latter can also restore the default setting (left and light) using the click of a button.

Settings	
Playlist pane position :	<input checked="" type="button" value="Left"/> <input type="button" value="Right"/>
UI Mode :	<input checked="" type="button" value="Light"/> <input type="button" value="Dark"/>
<input type="button" value="Save"/>	
<input type="button" value="Restore default and save"/>	
<input type="button" value="Cancel"/>	

Testing

Description of testing approaches used

A test approach is an implementation of strategy in a project which defines how testing would be.

In this project, the testing approach used is Unit Testing and the framework used is JUnit.

Unit testing is a software-level testing which is used to test individual components of the software. The purpose of this test is to validate the function of each method and ensure that it is performing as expected. It usually has input(s) and output(s). If the output(s) is as expected depending on the input(s), it is said that the Unit test has passed.

Evidence of testing (The description of each test)

Add song to song library (Appendix A)

This test will read the sample song file and will add all the data into a hash table. AssertNotNull is used to make sure that the hash table contains data from the file. If the hash table contains data, the test will pass.

Search song library for a song by title (Appendix B)

This test will get the data from the file into a hash table. It will then search for a predefined song. If the search result returns the Song object which matches with the Song object predefined, the AssertTrue function will return true and the test will pass.

Add a song to the end of a playlist (Appendix C)

This test will add song to the playlist file with a predefined name. Then a linked list created and the data from the file is inserted into it. The AssertTrue function will check the last index of the linked list. If the value of the last index matches the predefined name, the test will pass.

Play next song in (and remove from) playlist (Appendix D)

This test will get the playlist, clear the playlist and add 2 predefined values to the playlist. The first one (index 0) will be removed from the playlist. Then the test will check AssertTrue if the new index one contains the second predefined value. The test will pass if so.

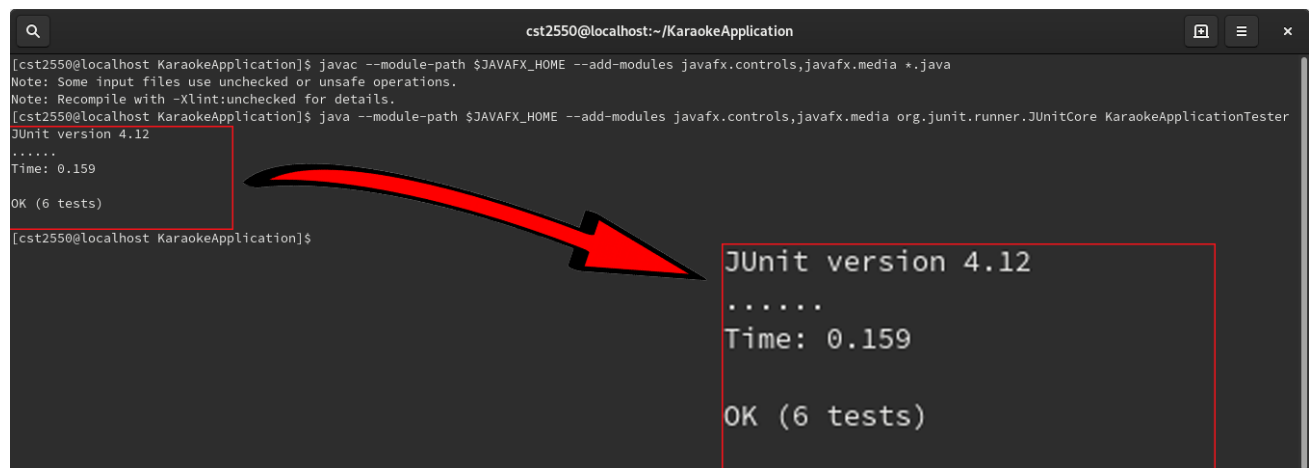
Retrieve song from playlist to view (Appendix E)

This test will read the playlist file and insert all the data into a linked list. The AssertNotNull function will check whether the linked list is empty or not. If the linked list contains data, the test will pass.

Delete song from playlist (Appendix E)

This test will get the playlist into a linked list and clear it. It will then add 2 predefined values to the linked list. It will search for the index of a specified song name. The function will then remove the said index from the playlist linked list. AssertTrue function will be used to check whether the linked list contains the removed song. If the playlist does not contain the remove song, the test will pass.

Evidence of testing (Screenshot from the terminal)



A terminal window titled 'cst2550@localhost: ~/KaraokeApplication' showing the execution of JUnit tests. The terminal output includes the following text:

```
[cst2550@localhost KaraokeApplication]$ javac --module-path $JAVA_HOME --add-modules javafx.controls,javafx.media *.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[cst2550@localhost KaraokeApplication]$ java --module-path $JAVA_HOME --add-modules javafx.controls,javafx.media org.junit.runner.JUnitCore KaraokeApplicationTester
JUnit version 4.12
.....
Time: 0.159
OK (6 tests)
[cst2550@localhost KaraokeApplication]$
```

A red box highlights the test results, and a red arrow points from it to a larger, magnified view of the same text on the right side of the image.

Conclusion

Summary of work done

This project is about developing software which should be able to play songs in order of a playlist created by the user. To achieve this goal, I had to consider that the library could contain a tremendous number of songs. Therefore, I had to choose a proper data structure. I have done some research about data structures and concluded that using Hashing is the most appropriate structure to use, given that it gives the best performance time-wise. After choosing a proper data structure, I proceeded to the design of the user interface. Then, I created the design and built the media player, both in JavaFX. After the front end and the data structure, I did the testing of the software using Unit Testing.

Limitations of approach and critical reflection of work

This Karaoke Application has several limitations despite having all of the required functionalities. For example, the user can only create and use one playlist. In many cases, the user would want to have a second playlist for the second choice of songs. Another limitation of the project is, on the full-screen mode of the media player, there is no media action button. Therefore, to pause, stop and skip the video, the user will have to exit the full-screen first. Choosing a separate song list file from the terminal is considered to be a limitation in my point of view.

As there was no requirement for the user interface, I had the free choice of my own for the design. I analysed various designs and concluded that they all hold identical characteristics in diverse positions. Therefore, spending time in design for this particular project was a waste and I used the simplest design I could think of. The more that I got indulged in the programming part of this coursework, the more I felt energetic about it, and the more I wanted to continue implementing new features. Lastly, I discovered that even simple lines of codes can result in an amazing application if properly thought of and well implemented.

How would I approach a similar project differently in the future?

If I had to do a similar project differently in the future, I would, first of all, include the choice to use multiple playlists. A way of doing this is to allow the user to create a new text file with the name of the playlist that he wants. Then a dropdown which will allow the user to choose into which playlist he wants to add his song. In this case, the user can use multiple playlists. To solve the full-screen media buttons issue, I could implement the use of keyboard shortcuts or even buttons which appear only on moving the cursor on a specific area of the stage. Using a file-chooser to change the song library file would ease the user instead of using an argument in the terminal.

References

- Introc.cs.princeton.edu. 2020. *Algorithms And Data Structures*. [online] Available at: <<https://introc.cs.princeton.edu/java/40algorithms/>> [Accessed 8 May 2020].
- Interview Cake: Programming Interview Questions and Tips. 2020. *Big O Notation / Interview Cake*. [online] Available at: <<https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity>> [Accessed 8 May 2020].
- Bigocheatsheet.com. 2020. *Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @Ericdrowell*. [online] Available at: <<https://www.bigocheatsheet.com/>> [Accessed 8 May 2020].
- Friesen, J., 2020. *Data Structures And Algorithms In Java, Part 1: Overview*. [online] JavaWorld. Available at: <<https://www.javaworld.com/article/3215112/java-101-datastructures-and-algorithms-in-java-part-1.html>> [Accessed 8 May 2020].
- Goodrich, M. and Tamassia, R., 2006. *Data Structures And Algorithms In Java*. Hoboken, N.J: Wiley.
- Docs.oracle.com. 2020. *HashMap (Java Platform SE 8)*. [online] Available at: <<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>> [Accessed 8 May 2020].
- beginnersbook.com. 2020. *HashMap In Java With Example*. [online] Available at: <<https://beginnersbook.com/2013/12/hashmap-in-java-with-example/>> [Accessed 8 May 2020].
- beginnersbook.com. 2020. *LinkedList In Java With Example*. [online] Available at: <<https://beginnersbook.com/2013/12/linkedlist-in-java-with-example/>> [Accessed 8 May 2020].
- Docs.oracle.com. 2020. *MediaPlayer (Javafx 2.2)*. [online] Available at: <<https://docs.oracle.com/javafx/2/api/javafx/scene/media/MediaPlayer.html>> [Accessed 8 May 2020].
- Sedgewick, R. and Wayne, K., 2015. *Algorithms*. Upper Saddle River (NJ) [etc.]: Addison-Wesley.

Appendices

Appendix A

@Test

```
public void addSongToLibrary() {  
    HashST<String, Song> songs = importData.importSongList();  
    assertNotNull("Songs should not be null", songs);  
}
```

Appendix B

@Test

```
public void searchSongInSongLibrary() {  
    HashST<String, Song> songs = importData.importSongList();  
    String searchKey = "Angel";  
    Song shouldMatchSong = songs.get("Angel");  
    Song matchedSong = null;  
    for (String songKey : songs.keys()) {  
        if (searchKey.equalsIgnoreCase(songKey)) {  
            matchedSong = songs.get(songKey);  
        }  
    }  
    assertTrue(shouldMatchSong.equals(matchedSong));  
}
```

Appendix C

@Test

```
public void addSongToPlaylistTester() {  
    exportData.addToPlaylist("Song Test");  
    LinkedList<String> playlist = importData.getPlaylist();  
    assertTrue(playlist.get(playlist.size()-1).equals("Song Test"));  
}
```

Appendix D

@Test

```
public void testNextSong() {
    LinkedList<String> playlist = importData.getPlaylist();
    playlist.clear();
    exportData.updateFile("playlist.txt", playlist);
    exportData.addToPlaylist("Song Test");
    exportData.addToPlaylist("Next Song");
    playlist = importData.getPlaylist();
    String firstSong = playlist.get(0);
    String secondSong = playlist.get(1);
    playlist.remove(0);
    assertTrue(playlist.get(0).equalsIgnoreCase(secondSong));
}
```

Appendix E

@Test

```
public void retrieveSongFromPlaylist() {
    LinkedList<String> playlist = importData.getPlaylist();
    assertNotNull("Playlist should not be null", playlist);
}
```

Appendix F

@Test

```
public void deleteSongFromPlaylist() {
    LinkedList<String> playlist = importData.getPlaylist();
    playlist.clear();
    exportData.updateFile("playlist.txt", playlist);
    exportData.addToPlaylist("Song to delete");
    exportData.addToPlaylist("A Song");
    playlist = importData.getPlaylist();
    int index = -1;
    for (int i = 0; i < playlist.size(); i++) {
        if (playlist.get(i).equalsIgnoreCase("Song to delete")) {
            index = i;
        }
    }
    playlist.remove(index);
    assertTrue(!playlist.contains("Song to delete"));
}
```