

**Data Structures and Algorithm**

Year 2 (2020/21) Semester 4

**School of InfoComm Technology**

Diploma in Cyber Security and Digital Forensics

**ASSIGNMENT**

Group B – Song / Playlist

|  |  |
| --- | --- |
| **Student Name** | **Student ID** |
| Ezra Ho Jincheng | S10194982A |
| Matthias Gan | S10197146D |
| Hannah Leong | S10195094B |

Contents

[1. Project Type & Details 4](#_Toc63635360)

[1.1. Roles and Contributions 4](#_Toc63635361)

[1.2. List of Data Structures and Algorithms implemented 6](#_Toc63635362)

[2. Reasons to using certain Data Structures and Algorithms 7](#_Toc63635363)

[2.1. Basic Data Structures 7](#_Toc63635364)

[2.1.1. Singly Linked List 7](#_Toc63635365)

[2.1.2. Queue 9](#_Toc63635366)

[2.1.3. Stack 9](#_Toc63635367)

[2.2. Algorithms 10](#_Toc63635368)

[2.2.1. Search 10](#_Toc63635369)

[2.2.2. Delete 10](#_Toc63635370)

[2.2.3. Insert 10](#_Toc63635371)

[3. The Program Explanation and Walkthrough 11](#_Toc63635372)

[3.1. Add Song to library 13](#_Toc63635373)

[3.1.1. Song ID 14](#_Toc63635374)

[3.1.2. Song Name 15](#_Toc63635375)

[3.1.3. Song Artist 16](#_Toc63635376)

[3.1.4. Song Genre 17](#_Toc63635377)

[3.1.5. Song Album 18](#_Toc63635378)

[3.1.6. Song Description 19](#_Toc63635379)

[3.1.7. Song Duration 20](#_Toc63635380)

[3.2. Display all songs in library 22](#_Toc63635381)

[3.3. Remove song from library 24](#_Toc63635382)

[3.4. Display individual song in more detail 29](#_Toc63635383)

[3.5. Add Song to Playlist (Queue ADT) 31](#_Toc63635384)

[3.6. Remove Song from Playlist (Queue ADT) 33](#_Toc63635385)

[3.7. Display Songs in Playlist (Queue ADT) 35](#_Toc63635386)

[3.8. Search History (Stack ADT) 37](#_Toc63635387)

[3.9. Quit (Option 0) 40](#_Toc63635388)

[4. Comparison of Different Data Structures and Algorithms and its Efficacy (Ezra) 41](#_Toc63635389)

[4.1. Singly Linked List vs Array 41](#_Toc63635390)

[4.1.1. 1st Reason: Why was Linked List used instead of Array (Size) 41](#_Toc63635391)

[4.1.2. 2nd Reason: Why was Linked List used instead of Array (Memory) 42](#_Toc63635392)

[4.1.3. 3rd Reason: Why was Linked List used instead of Array (Execution Time) 42](#_Toc63635393)

[4.2. Queue vs Stack 44](#_Toc63635394)

[4.3. Sequential Search vs Binary Search 46](#_Toc63635395)

[5. Algorithm and Data Structures Optimization (Ezra, Matthias) 49](#_Toc63635396)

[5.1. SongID Efficiency 49](#_Toc63635397)

[5.2. Removal of Unnecessary Operations 49](#_Toc63635398)

[5.3. Input Validation 49](#_Toc63635399)

[5.4. Search Algorithm 51](#_Toc63635400)

# Project Type & Details

Our Project is a Song Library and Playlist where the user is able to add Songs to a library, from which he is then able to add songs from the library to a playlist.

# Roles and Contributions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Name* | *Contribution* | *Specifications of contributions* | *Percentage completed* | *Date completed* |
| *Ezra* | *LinkedList.h,*  *LinkedList.cpp,*  *DSA-Final.cpp* | *Fully implemented LinkedList.h and LinkedList.cpp.*  *Contributed to choices for user input and fully contributed on callMenu().* | *100%* | *1st Feb 2021* |
| *Ezra* | *DSA-Final.cpp* | *Input validation for callMenu(), choice 1, 2 and 3, 4.* | *100%* | *3rd Feb 2021* |
| *Ezra* | *DSA-Final.cpp* | *Created unique song ID using running numbers* | *100%* | *4th Feb 2021* |
| *Ezra* | *Song.h, Song.cpp* | *Added song Description and song Album to allow user to add more details when entering a song into the song library* | *100%* | *5th Feb 2021* |
| *Ezra* | *LinkedList.cpp,*  *LinkedList.h,*  *DSA-Final.cpp* | *Implemented get() function (option 4) to get the songs and show more details to the user.* | *100%* | *5th Feb 2021* |
| *Ezra* | *Report* | 1. *Fully completed Advanced Feature on Efficacy of Data Structures and Algorithms* 2. *Contributed Advanced Feature - Algorithm and Data Structure Optimization* 3. *Writeup on how the application is used.* | *100%* | *7th Feb 2021* |
| *Matthias* | *Song.h, Song.cpp, Playlist.h,*  *Playlist.cpp* | *Matthias – implemented Song and Playlist classes* | *100%* | *31st Jan 2021* |
| *Matthias* | *Report* | *Brief intro of queue and why we are implementing it.*  *Brief write up of how the application is used.* | *100%* | *1st Feb 2021* |
| *Matthias* | *Playlist.cpp,*  *Playlist.h* | *Deleted Playlist class as we are only using one playlist and therefore it’s unnecessary* | *100%* | *7th Feb 2021* |
| *Matthias* | *Report* | *Add, remove song from playlist*  *Queue ADT*  *Algorithm & Data Structure Optmisation*  *Updated Project Type and Details* | *100%* | *7th Feb 2021* |
| *Hannah* | *Stack.h,*  *Stack.cpp,*  *DSA-Final.cpp,*  *Report,* | *Fully implemented Stack.h and Stack.cpp.*  *Fully Implemented Search History (option 8)*  *Brief intro to stack and why we are using it for Seach History.*  *Brief explanation of codes for option 8.* | *100%* | *7th Feb 2021* |

# List of Data Structures and Algorithms implemented

|  |  |
| --- | --- |
| Basic Data Structures | Algorithms |
| Linked List | Search |
| Queue | Delete |
| Arrays | Insert |

Ezra Ho:

* Data Structure Implemented: Linked List
* Algorithms Implemented:
  + Insert
  + Delete
  + Search

Matthias Gan:

* Classes Implemented: Song
* Data Structure Implemented: Queue
* Algorithms Implemented:
  + Add Song from Playlist
  + Remove Song from Playlist
  + Display Songs in Playlist

Hannah Leong:

* Data Structure Implemented: Stack
* Algorithms Implemented:
  + Push
  + Pop
  + Display

# Reasons to using certain Data Structures and Algorithms

## Basic Data Structures

### Singly Linked List

Singly Linked List Data Structure is used because the data number of data inserted by the user can’t be predicted and the number of data keeps changing during program execution. Since a song list allows the user to input as many songs as they please, this data structure is very helpful compared to the other data structures.

Since our program does not allows the user to manually input the song ID, it uses running numbers instead to create and maintain the unique ID of each song within the song library. This means that our song library is a sorted linked list as it goes in order or increasing numbers using the unique numbers of the running numbers. Compared to a sorted array, the sorted linked list requires sequential access but can increase and decrease in size accordingly, which is something that the sorted array is unable to perform as the size of the array is fixed. This may in turn, waste the storage space.

Using linked list, the insertion and deletion operations are far faster and easier compared to array. In an array, the insertion and deletion operations are costlier since the memory locations are consecutive and fixed. For linked list, the memory allocation is done during run-time, which means the system does not need to allocate any fixed memory compared to array, where memory is allocated during the compile time.

The time complexity of a sorted Linked List Data Structure is better on the chart when compared to sorted Array Data Structure. Sorted Linked List Data Structure has a time complexity of O(n) for all three Insert, Remove and Retrieve operations while a sorted Array Data Structure has a time complexity of O(n) for both Insert and Remove operation as well as O(logn) for the Retrieve operation. Hence, our decision to choose the Linked List Data Structure over Array Data Structure.



*Fig 2.1.1.1. Table for Time Complexity of Data Structure and Algorithms*

### Queue

Queue is chosen to be implemented to mimic the typical behavior of a music player where we usually add songs to a queue to be played. When we use an iPod, Spotify or Apple Music, we would typically have a playlist containing the songs we prefer to play in a specific order. Using Queue ADT, the user will be able to add Song objects one by one to the back of the queue by keying in the Song ID of the particular song. Similar, to a regular playlist, the user is not required to delete the song from the back of the playlist. Instead, he just needs to key in the Song ID of the song he wants removed from playlist. Given that a song would typically be played one by one in the queue, we are not concerned by the traversal time as it would almost always be simply moving to the next song in queue. When deleting a song, there is a need to traverse one by one through the queue. This is acceptable as one would not be playing the song he wants to delete and therefore the traversal time is not a concern.

### Stack

Stack is implemented to mimic the behavior of a Search history. A Search history is done one by one, as the user would only search 1 song item at a time, it is faster and simpler to just add it to a stack as the song item is being searched. Therefore, it’s a good to use stack as you just add on continuously, by continuously pushing to the stack. We no need to worry about the slow traversal time of a stack as the program will not have to traverse through the stack. When you delete the latest song, the program just has to pop the item off the top of the stack so there is no need to traverse through the stack to search for a specific item since a lot of the time people don’t specifically search for an item to delete from the history. It is also simpler to clear the entire search history by using a destructor to clear all items in the stack and after that a message will be displayed indicating that the search history has been cleared.

## Algorithms

### Search

The search algorithm is designed to check for an element or retrieve an element from any data structure where it is stored. In this project, we will be using sequential search as a link list is used.

In our project, all the Search operations are sequential searches instead of binary search. Sequential Search mainly used within Linked List or Array Data Structures. It traverses sequentially and every element is checked.

### Delete

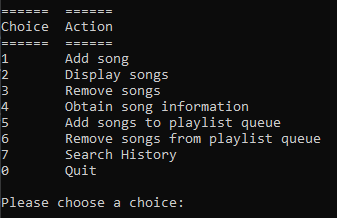
For deletion, in linked list, it means deleting a node from the beginning of the list. This is necessary as users can choose to delete the songs from the song library. Hence, requiring this algorithm.

### Insert

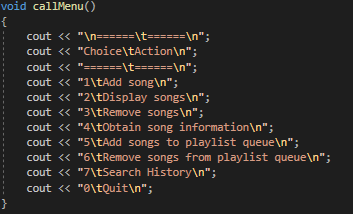
The insert algorithm for linked list is where a value is added at the beginning of the list. This would be helpful as the users are given the option to add their own songs into the song library. Hence, this algorithm is used for linked list.

# The Program Explanation and Walkthrough

When the user first runs the application, the user will see a menu with 6 options as shown in the image below.



*Fig 3.1. Image of DSA-Final.cpp running and displaying the menu – callMenu().*



*Fig 3.2. Image of DSA-Final.cpp callMenu() code.*

This menu will be shown whenever a user has to make use of the main menu to decide on what action to perform next.

***Please enter the number ‘1’ before proceeding to add a song.***

***Please enter the number ‘2’ before proceeding to display song library.***

***Please enter the number ‘3’ before proceeding to delete a song.***

***Please enter the number ‘4’ before proceeding to obtain more song information.***

***Please enter the number ‘5’ before proceeding to add a song to queue.***

***Please enter the number ‘6’ before proceeding to remove a song from queue.***

***Please enter the number ‘7’ before proceeding to user song search history.***

***Please enter the number ‘0’ to exit from program.***

## Add Song to library



*Fig 3.1.1. Image of LinkedList list1*

Linked List is used to add a song into a list, becoming the library as more songs are added. The list for our project is called ‘list1’. Our list does not come seeded with songs. Hence, users have to input their songs before testing the functionalities of the program. This is to ensure that the user tests all functions and does not miss out on certain functions such as ‘add’.

The following options are the parameters for the song, all of which requires user input except for song ID.

### Song ID

The song ID (songID) is crucial when it comes to a song playlist as it is one of the key identifiers / primary key which is constantly being used all the time to make decisions. Once such decision searching for the song ID in order to remove the song from the library. Thus, showing the significance the song ID plays in this project.

Initially, we allowed user to input their own song ID. However, we found difficulty in maintaining the list, hence deciding to use running numbers to allocate each song its unique song ID. Once an ID has been allocated, it cannot be reused, even if the song containing the song ID has been deleted.



*Fig 3.1.1. The variable “i” is used to keep track of the unique song ID.*

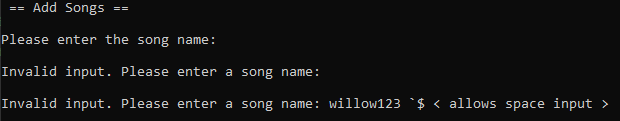


*Fig 3.1.2. How integer is stored and used as song ID.*

The song ID will then be automatically used to set the song ID for the song the user enters. The integer will then increase each time the user enters a new song to be added into the song library.

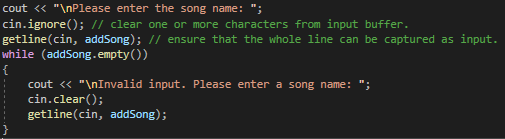
### Song Name

The song name (songName) input is where users enter the name of the song that they would like to save. This is also one of the most important parameters that users have to add in. The input allows for users to enter all alphabets, numbers, and special characters. However, it does not allow user to skip the input by clicking “Enter” without entering any input. This is an input validation method that was used in order to obtain the ideal input from the users.



*Fig 3.1.2.1. Image of DSA-Final.cpp running, display the invalid and valid user input for song name when song is added.*

As seen from the image above, the song name input allows all alphabets, numbers, and special characters, and only rejects an empty input by the user.



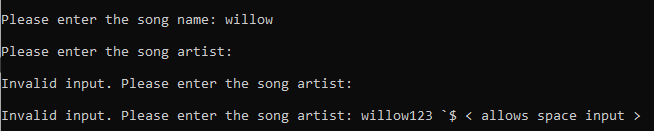
*Fig 3.1.2.2. Image of DSA-Final.cpp adding song name from user input code.*

Figure 3.1.2.2. shows the code within DSA-Final.cpp (where the main program runs), displaying a display text for user to enter the song name. It uses cin.ignore() to clear one or more characters from input buffers and uses getline(cin, addSong) to obtain the whole line of text instead of just a word. If the user does not input any name and presses on “Enter”, it will display an error line and asks the user to input again after clearing the previous attempt using cin.clear().

***Please enter any song name continuing.***

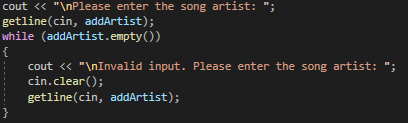
### Song Artist

Song Artist (songArtist) is also an input the user can enter all values such as alphabets, numbers, and special characters except a blank input by the user, similar to the input for song name. This is used to obtain an ideal input from the user.



*Fig 3.1.3.1. Image of DSA-Final.cpp running, display the invalid and valid user input for song artist when song is added.*

As seen from the image above, the song artist input allows all alphabets, numbers, and special characters, and only rejects an empty input by the user.



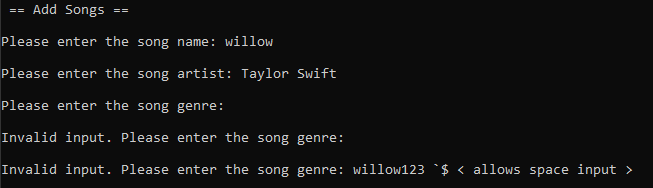
*Fig 3.1.3.2. Image of DSA-Final.cpp adding song artist from user input code.*

Figure 3.1.3.2. shows the code within DSA-Final.cpp (where the main program runs), displaying a display text for user to enter the song name. It uses cin.ignore() to clear one or more characters from input buffers and uses getline(cin, addSong) to obtain the whole line of text instead of just a word. If the user does not input any name and presses on “Enter”, it will display an error line and asks the user to input again after clearing the previous attempt using cin.clear().

***Please enter any valid song artist before continuing.***

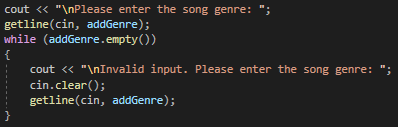
### Song Genre

Song Genre (songGenre) is also an input the user can enter all values such as alphabets, numbers, and special characters except a blank input by the user, similar to the input for song name. This is used to obtain an ideal input from the user.



*Fig 3.1.4.1. Image of DSA-Final.cpp running, display the invalid and valid user input for song artist when song is added.*

As seen from the image above, the song artist input allows all alphabets, numbers, and special characters, and only rejects an empty input by the user.



*Fig 3.1.4.2. Image of DSA-Final.cpp adding song artist from user input code.*

Figure 3.1.4.2. shows the code within DSA-Final.cpp (where the main program runs), displaying a display text for user to enter the song name. It uses cin.ignore() to clear one or more characters from input buffers and uses getline(cin, addSong) to obtain the whole line of text instead of just a word. If the user does not input any name and presses on “Enter”, it will display an error line and asks the user to input again after clearing the previous attempt using cin.clear().

***Please enter any valid song genre before continuing.***

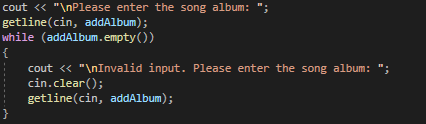
### Song Album

Song Album (songAlbum) is also an input the user can enter all values such as alphabets, numbers, and special characters except a blank input by the user, similar to the input for most of the other inputs here. This is used to obtain an ideal input from the user.



*Fig 3.1.5.1. Image of DSA-Final.cpp running, display the invalid and valid user input for song artist when song is added.*

As seen from the image above, the song artist input allows all alphabets, numbers, and special characters, and only rejects an empty input by the user.



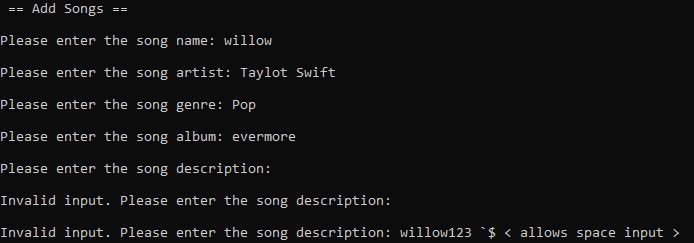
*Fig 3.1.5.2. Image of DSA-Final.cpp adding song artist from user input code.*

Figure 3.1.5.2. shows the code within DSA-Final.cpp (where the main program runs), displaying a display text for user to enter the song name. It uses cin.ignore() to clear one or more characters from input buffers and uses getline(cin, addSong) to obtain the whole line of text instead of just a word. If the user does not input any name and presses on “Enter”, it will display an error line and asks the user to input again after clearing the previous attempt using cin.clear().

***Please enter any valid song album before continuing.***

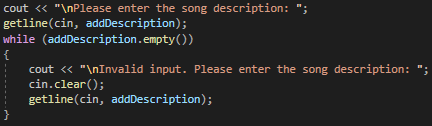
### Song Description

Song Description (songDescription) is also an input the user can enter all values such as alphabets, numbers, and special characters except a blank input by the user, similar to the input for song name. This is used to obtain an ideal input from the user.



*Fig 3.1.6.1. Image of DSA-Final.cpp running, display the invalid and valid user input for song artist when song is added.*

As seen from the image above, the song artist input allows all alphabets, numbers, and special characters, and only rejects an empty input by the user.



*Fig 3.1.6.2. Image of DSA-Final.cpp adding song artist from user input code.*

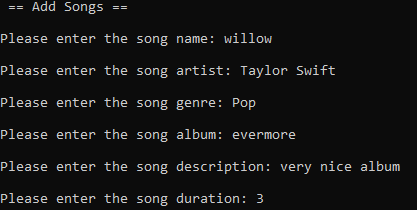
Figure 3.1.6.2. shows the code within DSA-Final.cpp (where the main program runs), displaying a display text for user to enter the song name. It uses cin.ignore() to clear one or more characters from input buffers and uses getline(cin, addSong) to obtain the whole line of text instead of just a word. If the user does not input any name and presses on “Enter”, it will display an error line and asks the user to input again after clearing the previous attempt using cin.clear().

***Please enter any valid song description before continuing.***

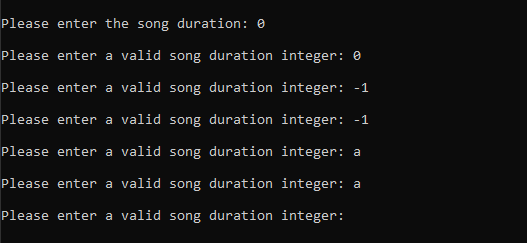
### Song Duration

The song duration (songDuration) is also one important aspect of the song. It allows users to add the duration of the song accordingly. However, we have restricted it to only allow integers to be inputted. This is to prevent having to set up a complicated DateTime format input field and storage data format in order to store the song duration in minutes and seconds. The image below shows how the song duration input will appear.

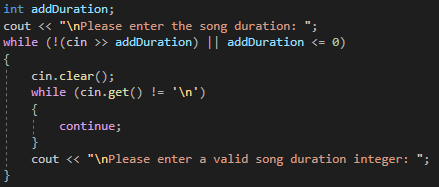
It will not take negative integers or alphabets.



*Fig 3.1.7.1. Image of DSA-Final.cpp running, display the user input for song duration when song is added.*



*Fig 3.1.7.2. Image of DSA-Final.cpp running and displaying the user input for invalid inputs of song duration.*



*Fig 3.1.7.3. Image of DSA-Final.cpp adding song duration from user input code.*

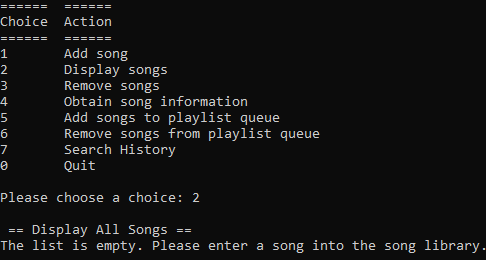
The code shows the question posed to the user. When the user replies, the response is captured and compared against the validation codes available to ensure that the input is not an alphabet and is not less than 0 or 0.

The code (!(cin >> addDuration) || addDuration <= 0) is the one that is used to validate that while the input of the user is not a non-numeric and is less than 0, it will clear the input of the user, create a loop to allow infinite input until the user input does not satisfy the while loop.

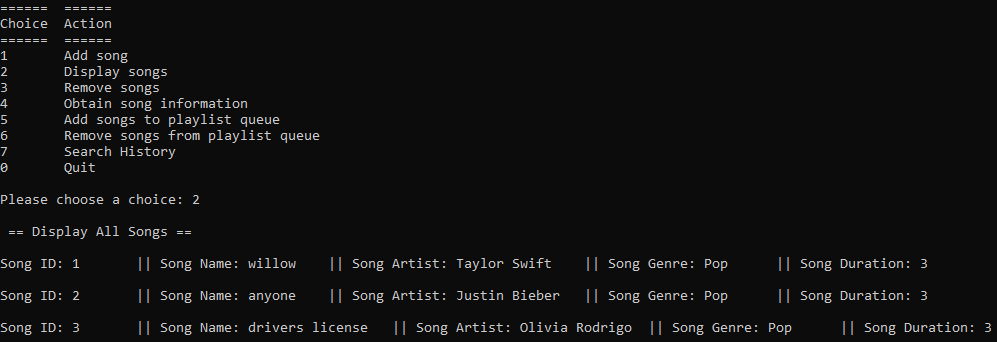
***Please enter any positive integer before continuing.***

## Display all songs in library

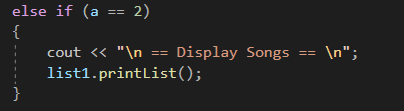
This function prints all the current songs in the library along with their attributes. However, users will need to add their own songs into the Linked List first as there are no seed data as shown in the image below. If the user tries to display the song library before adding any songs, it will result in an error message shown in Fig 3.2.1.



*Fig 3.2.1. Image of DSA-Final.cpp running and attempting to display the number of songs.*

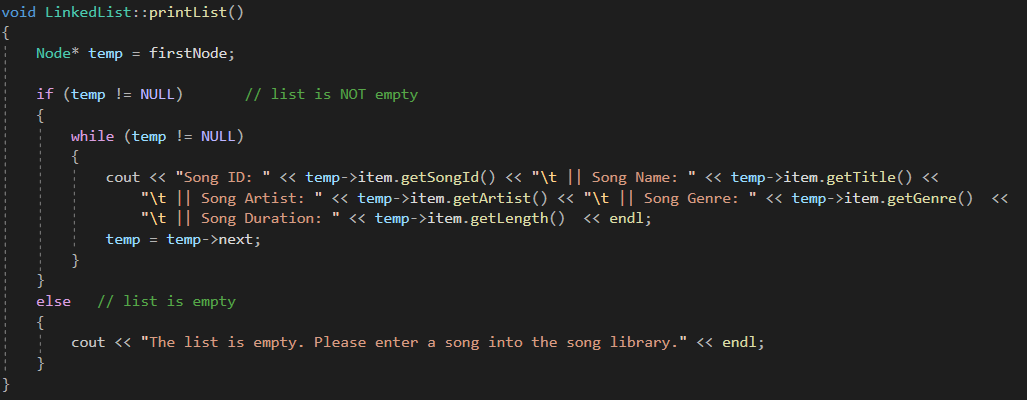


*Fig 3.2.2. Image of DSA-Final.cpp running and display a number of songs after being added in using choice 1.*



*Fig 3.2.3. Image of DSA-Final.cpp adding song duration from user input code.*

I basically used a function to print the song library as it would be simpler, and reusability is also high since it can be called anywhere else needed and the song library would still be the most updated version.



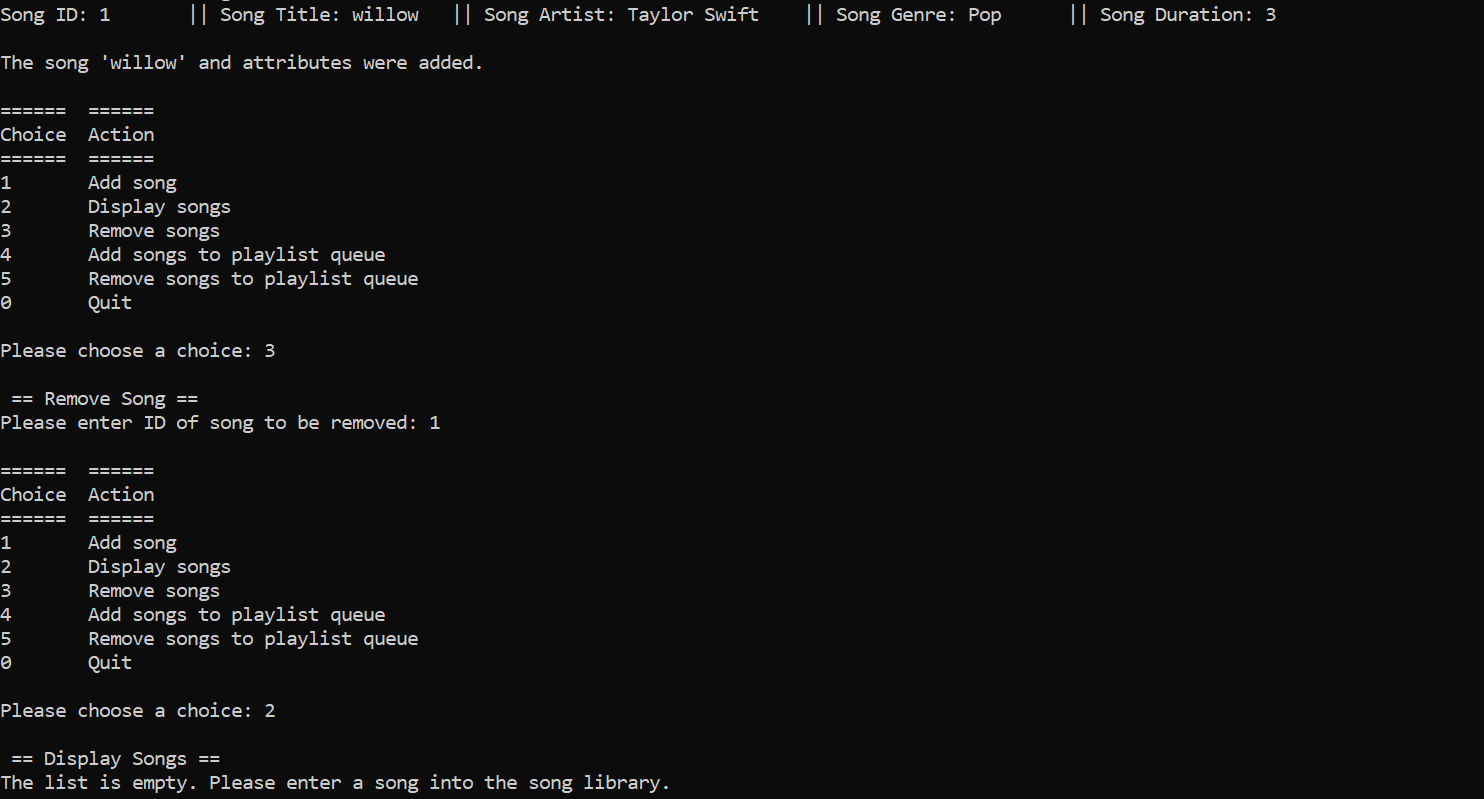
*Fig 3.2.4. Image of LinkedList.cpp printing the song library.*

Firstly, I checked if the song library is empty, if it is then it will return a text notifying the user that the linked list / library is empty.

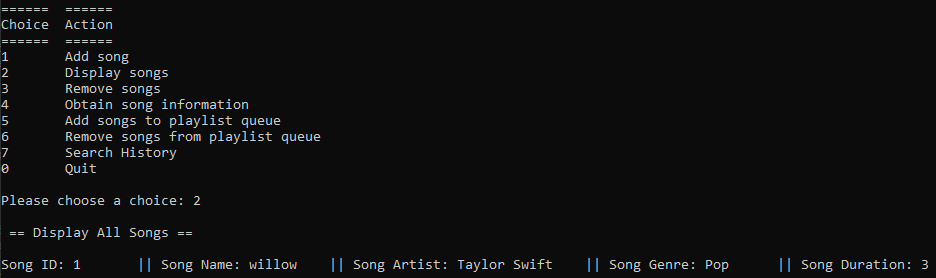
If it is not empty, the linked list will create a temporary node called temp and point to the first node. It will then proceed to print all the attributes of each song such as song ID, song name, song artist and the rest of them. This would produce the output as shown in Fig 3.2.2.

## Remove song from library

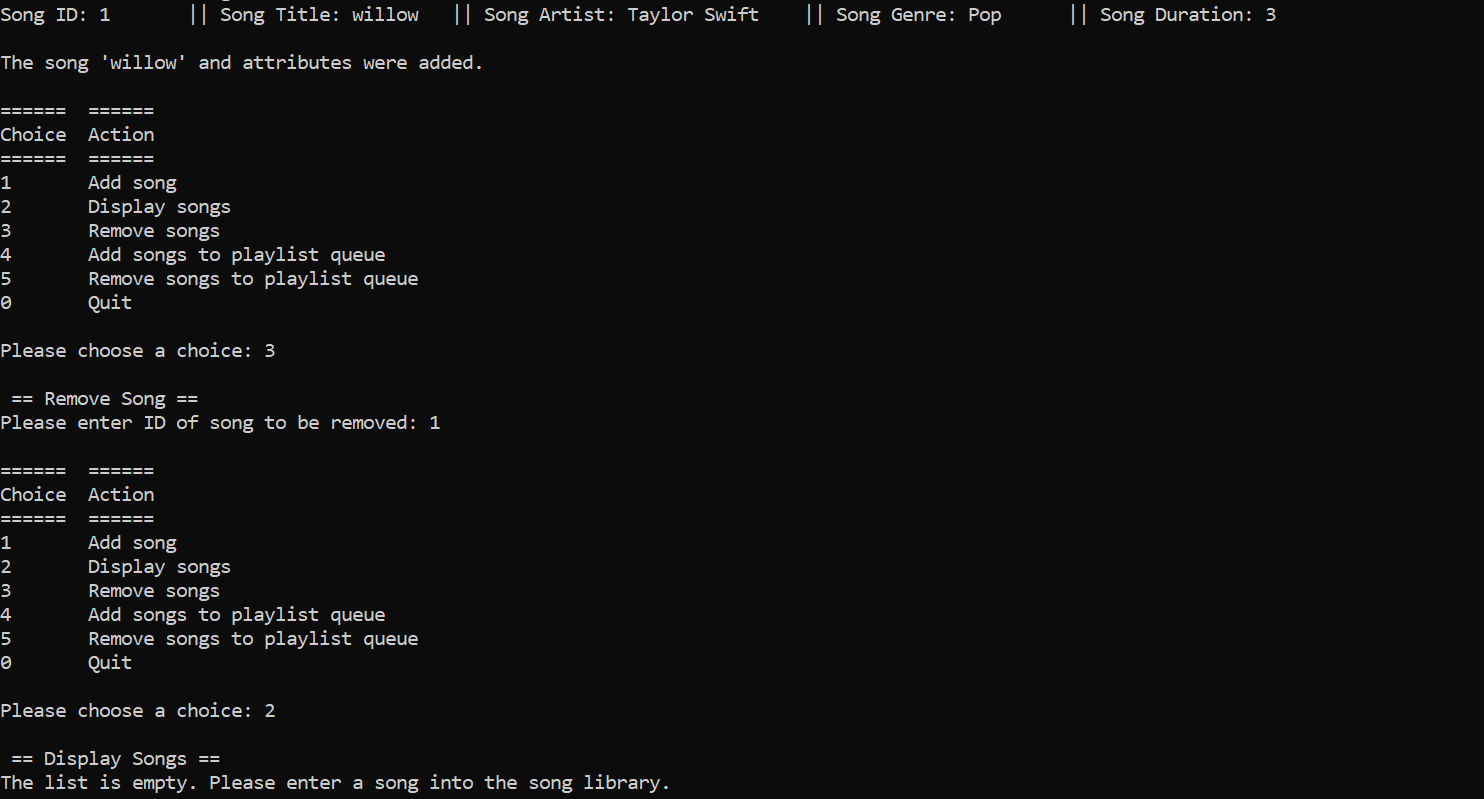
This function asks the user for the ID of the song they wish to delete and deletes it from the library. The user will again, have to add songs into the song library before they can remove a song from the library. This is because there are no seed data within the program to use to instantly remove any songs. If the user tries to remove the song without first adding a song, it the result will be similar to the error message in Fig 3.3.1.



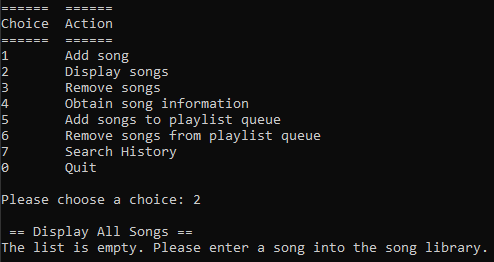
*Fig 3.3.1.1. Image of DSA-Final.cpp running and adding song into the song library.*



*Fig 3.3.1.2. Image of DSA-Final.cpp running and display song within the song library.*

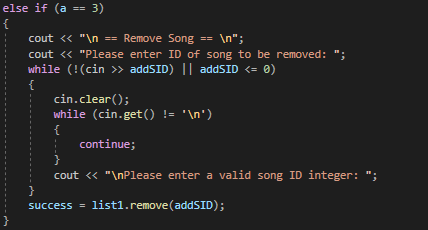


*Fig 3.3.1.3. Image of DSA-Final.cpp running and removal of song from the song library.*



*Fig 3.3.1.4. Image of DSA-Final.cpp running and prove that the song has been removed.*

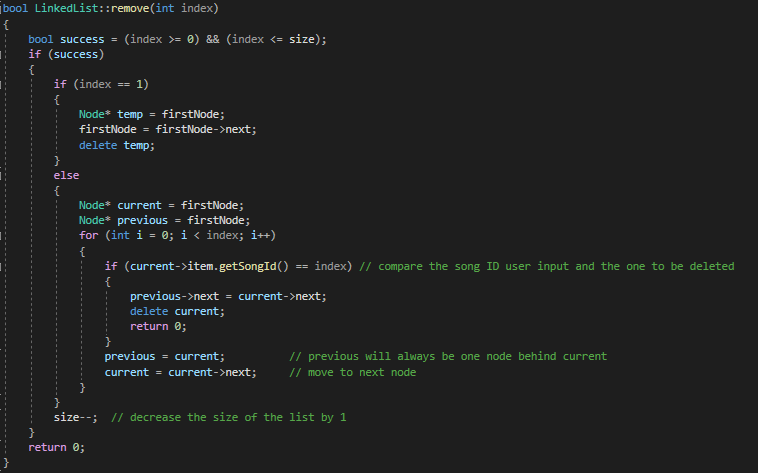
Figure 3.3.1.1 shows a successful insertion of song into the song library. Figure 3.3.1.4 shows the successful removal of song from the song library using the song ID. The user will key in the song ID corresponding to the song added to remove the song. The program will search for the song ID according to the input of the user, then deletes the node entirely after linking the node before it and after together.



*Fig 3.2.3. Image of DSA-Final.cpp removing song from song library using song ID.*

This code is implemented in DSA-Final.cpp, obtaining the input from the user in the form of an integer. The integer is meant to represent the song ID that the user would like to remove.

Once the input has been obtained, it is then compared to (!(cin >> addSID) || addSID <= 0) to check if the input is an alphabet or is less than or equal to 0. If it is, then the code will run an infinite loop after clearing out the input previously keyed in by the user until a satisfied input is obtained.



*Fig 3.2.4. Image of LinkedList.cpp removing song from song library code.*

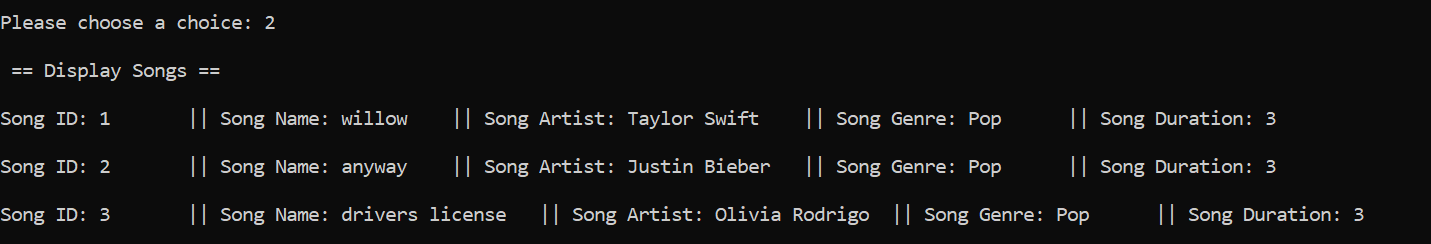
This code is located in LinkedList.cpp. It is the function created to remove the song from the song library using song ID as the input from the user. If there is currently only one song within the song library, the program will create a temporary node called temp and point it to the first node. It will then point the first node to the next node and delete temp the node thereafter.

If there are more than 1 songs located in the song library, the program will create 2 nodes, current and previous, both pointing to the first node. It will then use a for loop to traverse the list using both the nodes, but the current node will always be one node further than the previous node.

current->item.getSongId() == index would help locate the exact song ID to be deleted, then the program will have the current node to be on the node to be deleted. The previous node will then unlink with the node to be deleted and link to the node after the current node. After linking the nodes, it will return 0.

Upon exiting the if-loop, the previous node will replace the current node, only thereafter, will the current node traverse one node further and decrease size by 1 each time a node is removed.

Example: This example below from Figure 3.2.5 to 3.2.8 could possibly help with the visualization of the explanation above.



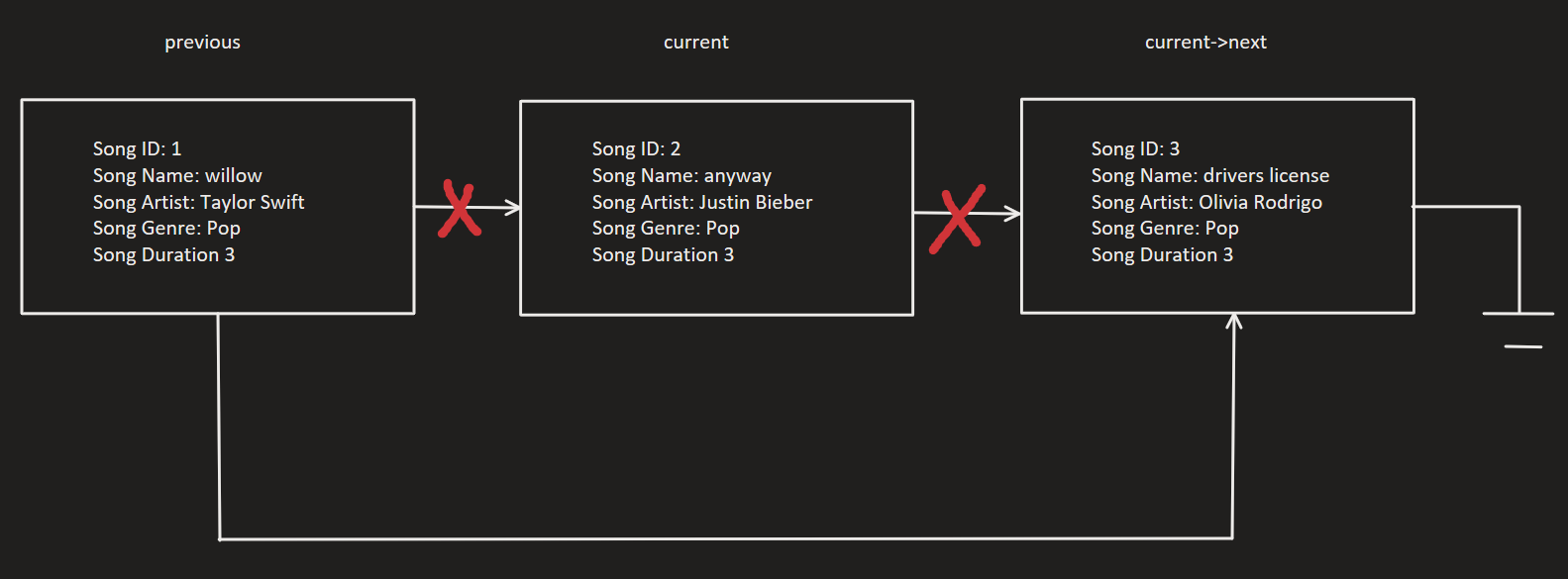
*Fig 3.2.5. Image of LinkedList.cpp displaying song from song library.*



*Fig 3.2.6. Image of linked list song library diagram.*



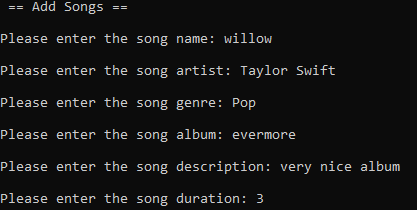
*Fig 3.2.7. Image of LinkedList.cpp linked list displaying songs after deleting song ID 2.*



*Fig 3.2.8. Image of linked list song library diagram once song ID 2 is removed.*

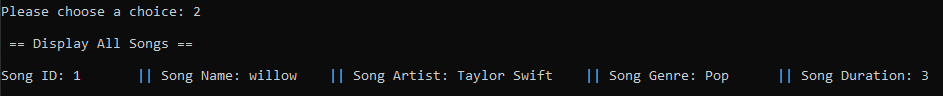
## Display individual song in more detail

This allows user to display the song they requested and provide more details about the song such as the description and album if entered by user. The user will input a description and album when entering a new song as shown by the image below.



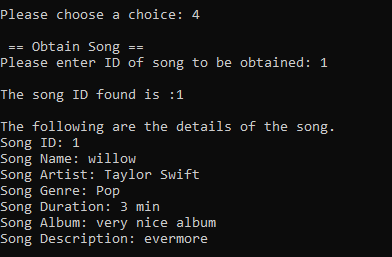
*Fig 3.4.1. Image of DSA-Final.cpp running, display the user input for all fields when song is added.*

The description and album will not show up when the user wants to display all the songs within the song playlist. It will only show up when the user clicks on option 4 – Obtain song information. This will show information of the particular song ID the user has selected as shown in Figure 3.4.3.



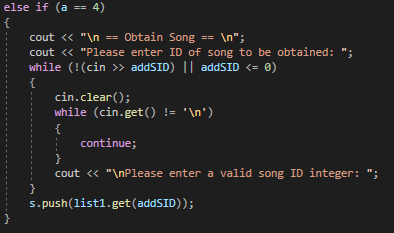
*Figure 3.4.2. Image of DSA-Final.cpp running, displaying the song within song library.*

Figure 3.4.2 shows the song entered by the user has been successfully saved into the song library.



*Figure 3.4.3. Image of DSA-Final.cpp running, displaying the song within song library.*

Figure 3.4.3. shows the user can search for the song using the song ID. It will then display all the details of the song that the user has entered in, such as the description and album to the user through option 4 on the main menu.



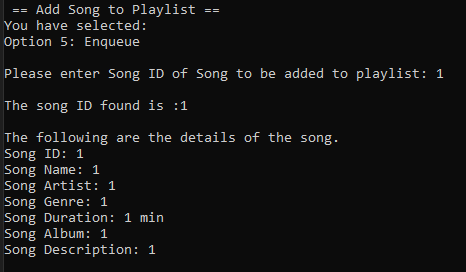
*Figure 3.4.3. Image of DSA-Final.cpp running, displaying the song within song library.*

Figure 3.4.4. shows the DSA-Final.cpp code for option 4 on the user menu. It does input validation by comparing the input of the user, checking if the value is either an alphabet or less than or equals to 0. The reason why 0 is also a restricted number here is because no song ID starts with an integer 0.

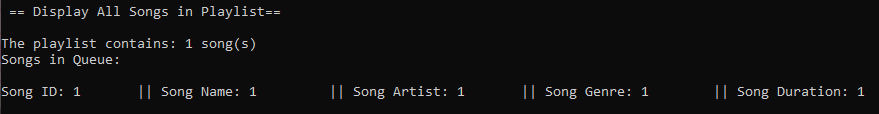
If the user inputs something that is not an alphabet or an integer less than 0, the program will clear the user input and loop a display to ask the user to input the correct type of input we are looking for, which is an integer within the song ID range. It will then get the song object and display it to the user as shown in Figure 3.4.3. The song object will also be pushed into a stack as an input, as the stack here is used as a search history for songs that they require additional information.

* 1. Add Song to Playlist (Queue ADT)

This function allows the user to set up a queue of songs to be played.

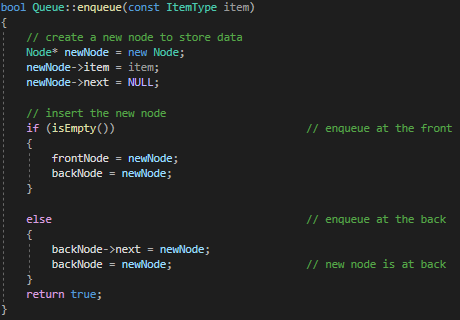


*Fig 3.5.1 Image of DSA-Final.cpp running, displaying a song successfully added to playlist.*



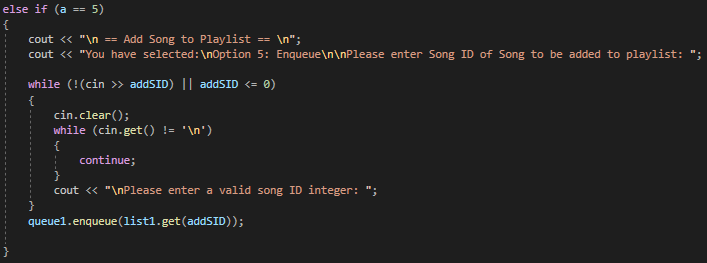
*Fig 3.5.2 Image of DSA-Final.cpp running, displaying the song in playlist.*

Our playlist works by storing song objects in a queue. The user keys in the Song ID of the song from the library he wishes to add to playlist. Input validation is used to ensure the song exists. The get function is then called to retrieve the song object from the library and enqueue it in the playlist. After the song has been enqueued, the details of the song are shown to the user.



*Fig 3.5.3. Image of Enqueue function within Queue.cpp.*

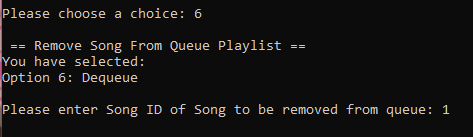
This is the code within Queue.cpp that enables the song object to be obtained by the Queue Data Structure and enqueued into Queue Playlist.

 *Fig 3.5.4. Image of DSA-Final.cpp, enqueuing the song.*

As shown in Figure 3.5.4, when the user enters the number 6 on the main menu, it will ask the user to enter an integer from the song ID that has been allocated to a song. It will then enqueue the song into the Queue Playlist.

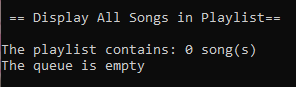
* 1. Remove Song from Playlist (Queue ADT)

This function allows a user to remove a song that is in the queue. If the user has entered an integer into the program, it will display an output as shown in Figure 3.6.1.

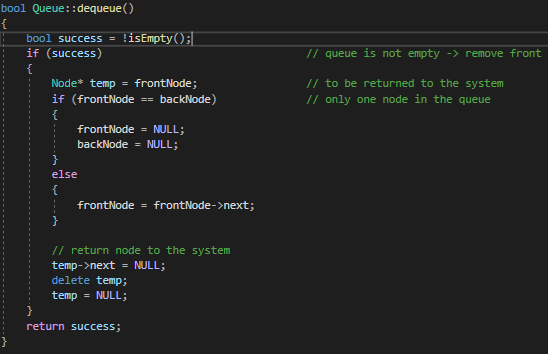


*Fig 3.6.1 Image of DSA-Final.cpp running, displaying the song successfully removed from playlist.*

When there are no songs within the Queue Playlist, the output will be similar to that of Figure 3.6.2.

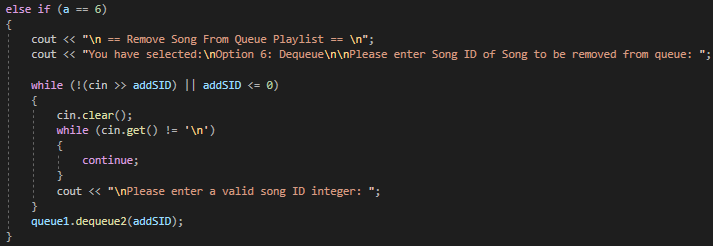


*Fig 3.6.2 Image of DSA-Final.cpp running, displaying empty playlist after song was removed from playlist.*



*Fig 3.6.3. Image of Dequeue function within Queue.cpp.*

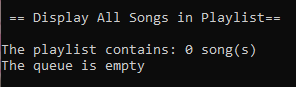
This shows the dequeue operation within Queue.cpp. It will first check that the queue is not empty, then if there is only one node within the queue, will set the front and back nodes to null. Otherwise, it will traverse to the front to dequeue the song.

 *Fig 3.6.4. Image of DSA-Final.cpp, dequeuing the song.*

The user keys in the Song ID of the song from the library he wishes to remove from playlist. Input validation is used to ensure the song exists. The get function is then called to retrieve the song object from the library and dequeue it from the playlist. After the song has been dequeued, the program automatically returns to the main menu.

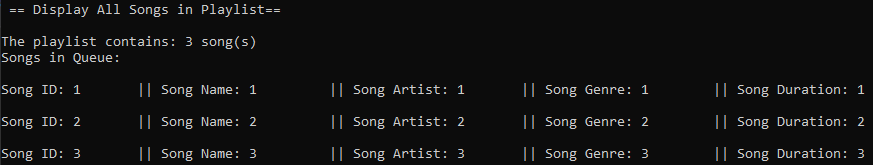
* 1. Display Songs in Playlist (Queue ADT)

This function prints the current number of songs in the playlist queue, songs in the playlist queue along with their attributes. However, users will need to add their own songs into the playlist first as there is no seed data as shown in the image below. If the user tries to display the playlist queue before adding any songs, he will be reminded that it is empty.



*Fig 3.7.1 Image of DSA-Final.cpp running, displaying an empty playlist.*

This will be shown when user try to display songs within the Queue Playlist when there is no song inside.

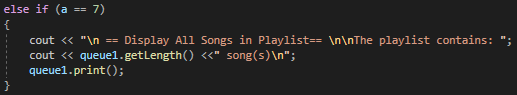


*Fig 3.7.2 Image of DSA-Final.cpp running, displaying the songs in playlist.*

This shows that the queue is able to display the number of songs within the queue, as well as the songs.

 *Fig 3.7.3. Image of Queue.cpp, getLength(), isEmpty() and print() function.*

This is the operation within queue, where the user will find the length of the Queue Playlist as well as print the Queue Playlist.

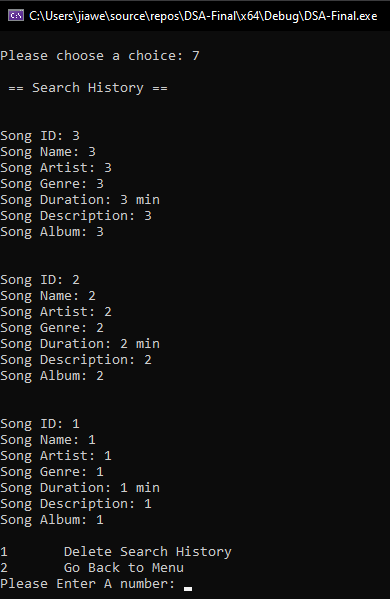


*Fig 3.7.4. Image of DSA-Final option 7.*

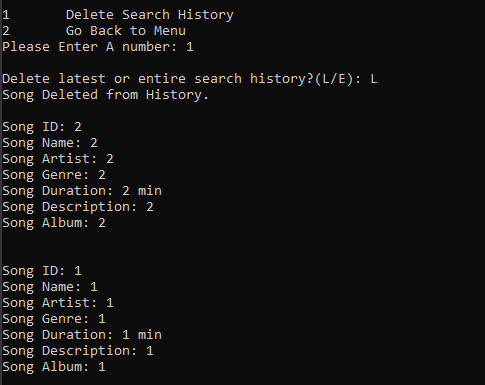
This shows how the user is able to obtain the songs in DSA-Final.cpp.

* 1. Search History (Stack ADT)

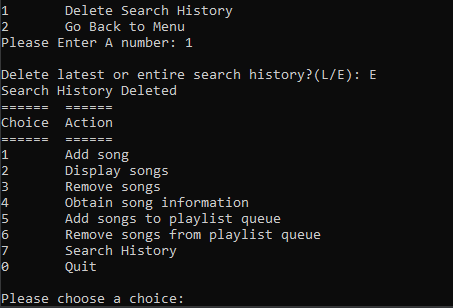
This function allows a user to view their search history when they use option 4. If the user so wishes to Delete their Search History, they can.



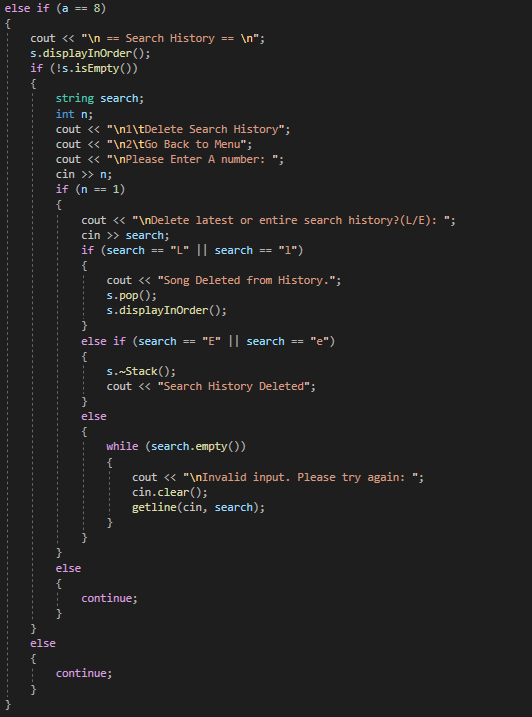
*Figure 3.7.1 Image of DSA-Final.cpp running, displaying the songs within search history.*



*Figure 3.7.2 Image of DSA-Final.cpp running, displaying the history after deleting latest search.*



*Figure 3.7.3 Image of DSA-Final.cpp running, after deleting entire history.*

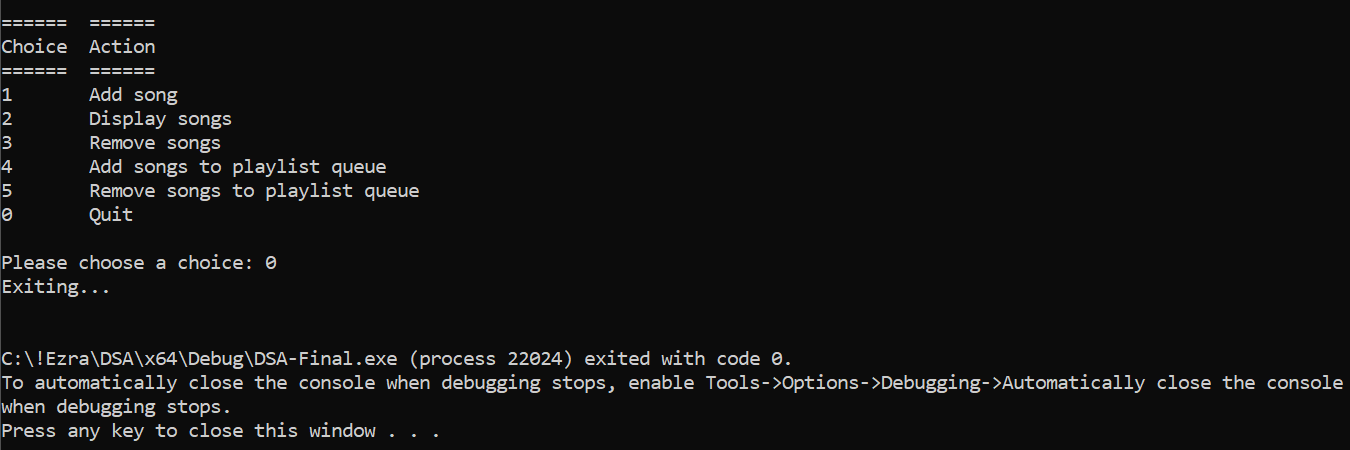
*.* 

*Figure 3.7.4 Image of Search History in DSA-Final.cpp.*

The figure above shows the code in DSA-Final.cpp for option 8 (Search History) on the user menu. When the user selects this option it first displays the Search history, from the latest search to the oldest search. After that it checks if the Stack is empty, if the stack is empty, a message is displayed “Search History is Empty.”. The user then has the option to “Delete Search History” or “Go Back to Menu”. The input is taken in, either 1 or 2, if the user selects option 1, an option will pop up “Delete latest or entire search history?(L/E)”, “L” deletes the item that is at the top of the stack and then displays the remaining items in the stack. “E” deletes all items in the stack, which means the entire Search History is cleared.

* 1. Quit (Option 0)

This function kills the program and stops it from running.



*Fig 3.6.1 Image of user pressing 0, stopping the program from running.*

# Comparison of Different Data Structures and Algorithms and its Efficacy (Ezra)

## Singly Linked List vs Array

In our project, Singly Linked List Data Structure was used enable users to enter and store their songs within the song library, which is the linked list. The first reason why linked list was used instead of array for our project, was because we wanted to ensure that no space was wasted, and the second reason is so that the user can add as many songs as they please without the concern of hitting the maximum limit the array can contain.

Array Data Structure stores their elements in contiguous memory locations, which results in simple calculable addresses for the elements stored, resulting in faster access to the element at a specific index. Although Linked List Data Structure are less rigid in the storage methods and the elements can be stored at random and not in contiguous locations. Due to the latter, Linked List Data Structure requires additional tags referencing the next element (usually called “next”).

### 4.1.1. 1st Reason: Why was Linked List used instead of Array (Size)

The biggest reason why we selected Linked List Data Structure instead of Array Data Structure is because of the required size allocation of the Array Data Structure. We needed the list to be something that the user can edit in size while on runtime. Although we could have predefined a certain size for the array, it would consume more memory since there is no guarantee that the user will fill all of the allocated array memory. Since Linked List Data Structure allows dynamic increase in size of the list during runtime, the user can add as many song as they please and there would be no memory waste, as a deleted element would release the space back to the memory.

The reason why a Linked List Data Structure is able to change dynamically at runtime, is due to each node pointing to the next one, such that data can exist at scattered / non-contiguous addresses. Since Array data can only be stored in contiguous blocks of memory in an array, the size cannot be altered during runtime for the fear of overwriting over other data.

### 4.1.2. 2nd Reason: Why was Linked List used instead of Array (Memory)

For Memory Efficiency, if both Array and Linked List Data Structure have the same number of elements, the linked list uses more memory since it keeps a reference node to point to the next element. However, the size flexibility used in linked list makes it use less memory overall due the aforementioned reason of us allocating too much space and users only using a fraction of it. Linked List Data Structure is extremely helpful in a case where there is uncertainty in size allocation or large variations in the size of data elements.

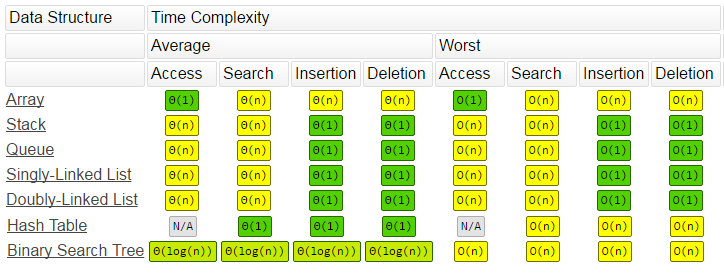
Array Data Structure allocates memory during compile time; hence, memory is already predefined and used although the array itself is empty. But Linked List Data Structure allocates the memory during runtime so it can increase the size proportionately to the amount of data stored.

### 4.1.3. 3rd Reason: Why was Linked List used instead of Array (Execution Time)

In terms of execution time, the elements in an Array Data Structure can be accessed directly with the index, while Linked List Data Structure requires a traversal from the first element to reach any element within the linked list.

Although cache locality is better in arrays due to contiguous memory allocation, which could possibly improve performance. With this advantage, operations such as modifying elements are faster in arrays. However, other operations such as inserting and deleting an element are faster in Linked List Data Structures.

In our project, we mainly use the inserting and deleting operation to add and delete songs from the linked list. Thus, opted for Linked List Data Structure as it is faster when performing these operations.



*Fig 4.1.3.1. Time Complexity chart*

As seen from the image above, the average time complexity of Singly Linked List for Insertion and Deletion is O(1) while the same operation for Array is O(n). This supports my third reason, stating that we mainly use only Insertion and Deletion operations within our project for the Linked List Data Structure.

In conclusion, there are 3 factors which affect our decision to choose between Array and Linked List Data Structure. The 3 factors are the size, the memory allocation and efficiency, and execution time.

## Queue vs Stack

Queue Data Structure is used in our project to ensure that the user can add the song into queue. It helps to ensure the program can function similar to a real song playlist, whereby users can add songs into the queue to be played next once the previous song has ended. The Queue Data Structure is a linear data structure, in which elements can be inserted only from one side (usually called “rear”) and deleted only from the other side (usually called “front”). This data structure follows the First In First Out (FIFO) principle, where the first element inserted into the queue will always be the first to exit.

Meanwhile, the Stack Data Structure is also a linear data structure, but elements can only be inserted and deleted from only one side of the list, called the “top”. Contrary to Queue Data Structure, Stack Data Structure follows the Last In First Out (LIFO) principle, where the element last inserted has to be the first to come out, and the first element to enter the data structure is always the last to exit.

In our program, we used both of these data structures for different reasons. As mentioned above, queue is used to ensure that user can skip to the next track which has been queued. However, stack is used to save the song search user has performed using option 4 - Obtain Song Information, of the main menu. <show picture>

Queue was used as a data structure to ensure that users can queue their songs using the enqueue operation within the Queue Data Structure. This will queue the song from the back using the “Enqueue” operation, meaning that the song added to the queue will be at the back since this data structure uses a First In First Out (FIFO) principle. Users can also remove the song from queue using the “Dequeue” operation, after the song has been played (not being played actually, but just in practice), this process will remove the song at the front from the queue , which was the song that has been played. With these two convenient operations, it is very suitable to use this data structure for this scenario as stack is unable to perform this operation.

Stack is used to contain the search history of users who have tried to obtain more information about a certain song. The song object will be taken by Stack Data Structure and is displayed there when the user clicks on option 7 – Search History.

Since Stack Data Structure uses a Last In First Out (LIFO) principle, a song object searched first will be saved first into the Stack Data Structure using the “Push” operation. This will push the objects into the Stack Data Structure, and as the user searches for more songs, more of those inputs are saved into the Search History option and will be printed in order specified by users.

Despite the differences in operators, the Queue and Stack Data Structures have the same average and worst time complexity when it comes to Insertion and Deletion operations, Enqueue and Dequeue for Queue Data Structure and Push and Pop for Stack Data Structure, respectively. This means that there is objectively no difference in the amount of time that a certain operator takes to perform. Since our program mainly uses Enqueue, Dequeue, Push and Pop, and all are either Insertion or Deletion operators, they all have the time complexity of O(1) as shown by Figure 4.1.3.1 above, stating that there is no difference in time taken to execute the action regardless of the size of data.

Hence, when comparing Queue and Stack Data Structure, there is no clear-cut reason as to which is better or faster. Users will need to know the operations and how these two data structures are different from one another in order to select them and make full use of the operations. The Queue Data Structure is able to better produce results when it comes to programs or problems involving Data Buffers, Traffic Analysis, just to name a few. In the meanwhile, the Stack Data Structure is better applied to problems requiring a back and forward function such as a Back button in a Web Browser, an Undo button in Word processor and more. Thus, the efficiency of each of these two data structures lies in the nature of the program needed.

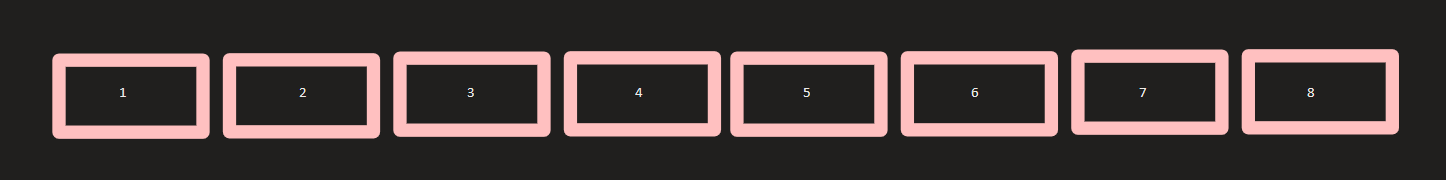
## Sequential Search vs Binary Search

Sequential Search is used in our program instead of Binary Search as our program does not use data structures requiring Binary Search. A sequential search is used for data structures such as Linked List and Array, while binary search is used in algorithms such that are already sorted and only for Array Data Structure.

In our case, since song ID was given by the running number, we can assume that it is a sorted list. We are also using Linked List Data Structure instead of an Array Data Structure, hence unable to compare execution times between sequential and binary search on a linked list.

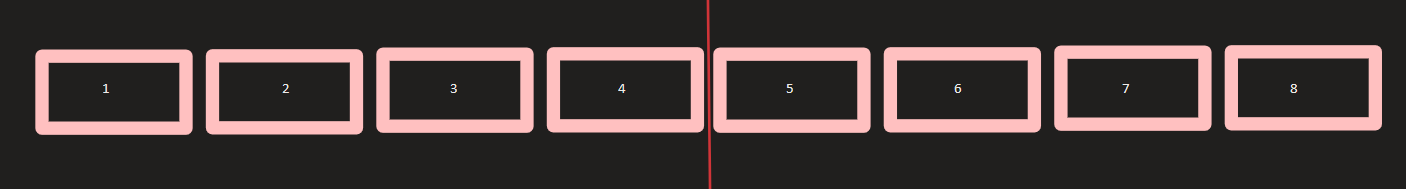
But just for comparison, if we did have an Array Data Structure, due to the nature of the program and this project being small in size, there is almost no difference between the execution timing of sequential and Binary Search Algorithm to the point that it matters that much. However, when a user has thousands or millions of nodes, a Binary Search Algorithm can greatly increase the speed of the search as it cuts the list by half and compare the number the user entered against the middle number of the list. If the number is not found at the middle, the algorithm will then compare and see if the number is smaller or greater than the current number in the middle. If it is smaller, it will cut the list in half and focus searching on the shortened list. This will continue until the output is obtained.

Below is an example to make the explanation clearer.

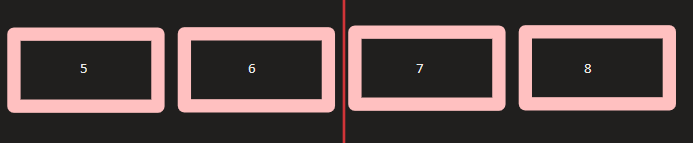
**

*Figure 4.3.1. Image of Sample List consisting 8 elements*

The image above shows 8 elements. If the user wants to find the value ‘6’, located within the 6th element. By using binary search, the algorithm will first find the midpoint of all the elements. In this case, it is between the 4th and 5th element of value ‘4’ and ‘5’ respectively.

*Figure 4.3.2. Image of Sample List consisting 8 elements, being split into 2*

And since the value ‘6’ is greater than the value within the middle element, the binary search will ignore the bottom half of the list (i.e., 1st, 2nd, 3rd, and 4th element). It will only focus on the 5th to 8th element. From here, it will repeat the process of cutting the list into half as shown in the image below.

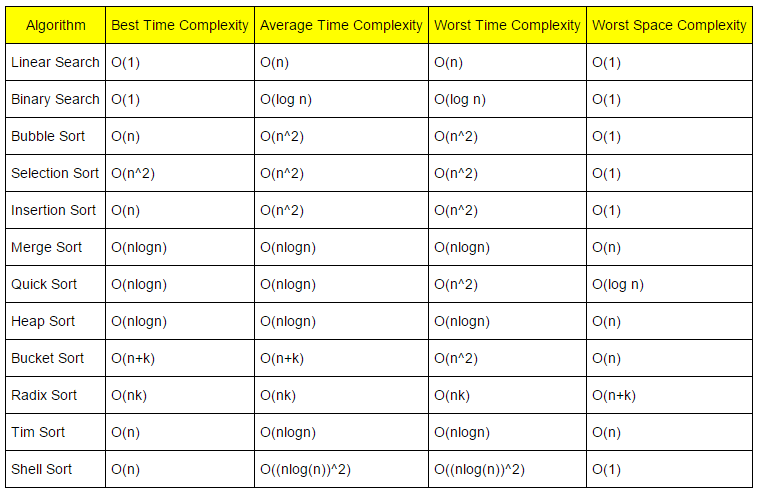
*Figure 4.3.3. Image of Sample List consisting 8 elements, being split into 2 again to be compared*

The program will then compare the user input, 6th element, to the remaining elements and decide which side of the list to remove and which side to focus on. In this case, it will recognize that the value ‘6’ was located in the 6th element.

This is the end of the example for explanation.

Meanwhile, Sequential Search Algorithm traverses through each and every element of the list or array, searching for the element before retrieving it. This will be much slower than binary search in practice.

Therefore, when comparing the time complexity of both Sequential and Binary Search Algorithm, it was safe to say that in average use cases when having a larger program with far more data, Binary Search Algorithm was much faster with an average time complexity of O(log n) and Sequential Search Algorithm with O(n). However, since this program is working with a minimal amount of data, the best time complexity is applied here instead, and in this case, both Sequential and Binary Search Algorithms have the best time complexity of O(1). With this in mind, there is virtually no difference whether we used Binary or Sequential Search Algorithm, as they will both be completed in the same amount of time when using the same dataset. Figure 4.3.4. shows the time complexity for Sequential / Linear Search Algorithm and Binary Search Algorithm.

 *Figure 4.3.4. Image of Time and Space Complexity for Searching and Sorting algorithms*

Although the Linked List Data Structure was not able to perform binary search, we just wanted to compare the time complexity between the two algorithms.

# Algorithm and Data Structures Optimization (Ezra, Matthias)

## SongID Efficiency

Throughout, our program, the user is only asked to key in the details of each Song once. When a Song is added to a playlist (Queue ADT), a new song is not created. Instead, our program traverses the song library (List ADT) where our Songs are stored and adds the song object to the playlist using the Song ID provided by the user. This ensures that are program uses minimal memory when running.

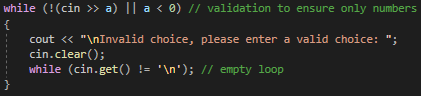
## Removal of Unnecessary Operations

We have also taken steps to ensure that our program is as “light” as possible. Initially, we had implemented a Playlist class to store Playlist objects should we decide to create multiple playlists. However, after deciding we would only have one Playlist, we deleted the entire Playlist class as it was surplus to requirements to run the program. We have also removed other unused methods in our classes and data structures to ensure our code is as lightweight and efficient as possible.

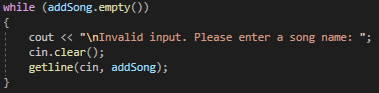
## Input Validation

We have also tried to add as much input validation as we could, but not all inputs are properly validated, which may result in program crashes and other errors to pop up when the user enters an undesirable input. We have enabled input validation for all the user inputs for the main menu, as well as all of the user inputs when a user enters a new song into the song library. This would prompt the user to enter a valid input when attempting to key in a song into the song library. Input validation is not only important within small programs, but in the real world as well. Hackers can easily infiltrate web applications if there is insufficient input validation provided for the input by the programmers. This will in hurt allow hackers to exploit vulnerabilities within the web applications. Within this small program, the damage is minimized as there is no connection to the internet, however, if a real application were to crash when the user enters a wrong input, it would be infuriating for the user experience. Hence, input validation was used to ensure that any wrong inputs would prompt the user to enter the correct format until a desirable input is obtained.

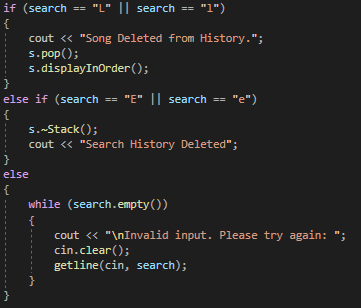
Figure 5.2.1. to 5.2.3. are all screenshots of input validation used in our code.

**

*Figure 5.2.1. Input validation for main menu.*



*Figure 5.2.1. Input validation for song attribute when user adds a new song.*



*Figure 5.2.1. Input validation for search history.*

## Search Algorithm

The search algorithm in our project is optimized as we have opted with the suitable algorithm which is easily to implement and yet speedy. We have decided to use sequential search algorithm as it has the same time complexity of O(n) when a small set of data is used as shown in Figure 5.4.1. below.

By opting for the sequential search algorithm, the program does not have to worry with handling the worst-case scenario as our data set is small. With this, the efficiency of the algorithm is equivalent to that of other search algorithms such as binary search algorithms. Therefore, it has been optimized.