VIETNAM NATIONAL UNIVERSITY – HCM

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE

AND ENGINEERING

ALGORITHMS AND DATA STRUCTURES

Semester 2, 2023 - 2024

Instructor: Vi Chi Thanh, Thai Trung Tin

Topic: Music player

Name: Nguyễn Hồng Sơn

ID: ITDSIU21117

Github repository: DSA-Music_Player

TABLE OF CONTENTS

I. INTRODUCTION

A. Motivation

Developing a music player app can be a deeply rewarding and enriching experience, both personally and professionally. For those with a genuine passion for music, creating a music player app offers the opportunity to combine my technical skills with my love for the art form.

By building a music player, I can improve the user experience for listeners, providing them with a more intuitive, feature-rich, and personalized way to engage with their favorite songs. Beyond the practical application of programming concepts, designing a music player app also allows for innovation and the exploration of new ideas that can differentiate my creation from the competition. Furthermore, a well-crafted music player has the potential to be monetized, turning my project into a viable commercial venture.

Ultimately, the process of developing a music player app can be an invaluable learning experience, enabling me to deepen my understanding of software engineering, multimedia integration, and potentially even areas like audio processing and machine learning.

Whether my goal is to create a tool that enhances people's music listening experience or to build a portfolio piece that showcases my technical abilities, the motivation to develop a music player app is a powerful one that can lead to immensely rewarding outcomes.

B. Technologies

Language:

- Python

Libraries:

- Tkinter
- Customtkinter
- Threading

- Time
- Pygame
- MP3
- Os

C. Launch

Run code file: ".\Final\Final_UI.py"

II. THE PROJECT CHESS GAME

A. Rules

About the rules to run the application:

- Objective:

1. Add your favorite music from the song library box and then just click on the play button "▶", the first music in the playlist box on the left-hand side will be played. The current song played will be illustrated at the bottom of the function button – next, play, previous button – then we also see the process of this song base on the process bar below.
2. After the current song finishes, it will play the next song in the playlist and the application will stop if it does not have another song behind. However, if we add new song after the application stop, it will run the current song again and the next song which is added will be run after it finishes.

- Setup:

3. If you want to add new song into song library, please follow these step:
    1. Add new song file with format "title-artist.mp3" into "Music" folder.

2. Access to the “get_title_artist” method of “MusicManager” class in “Final_Manager.py” file.
3. Add the artist of the song in “artist” parameter base on format “title-artist”: “artist”.
4. Similar with step 3, add title base on format format “title-artist”: “title”.

General:

- “Search” box: write down letters or words of title or artist, and the result will be shown in the song library
- “Add” button: add the song in song library box into playlist box.
- “Delete” button: delete the song in playlist or library base on what you select.
- “▶” button: play or pause the current song of song playlist.
- “<” button: back the current song to the previous song. If reach the first song, it will not response.
- “>” button: go to the next song and it will not response if reach the last song.
- Process bar: will illustrate the current song playing at the moment.
- List box above process bar: show title and artist of the current song.

B. Design

I am employing a basic user interface while developing my project.

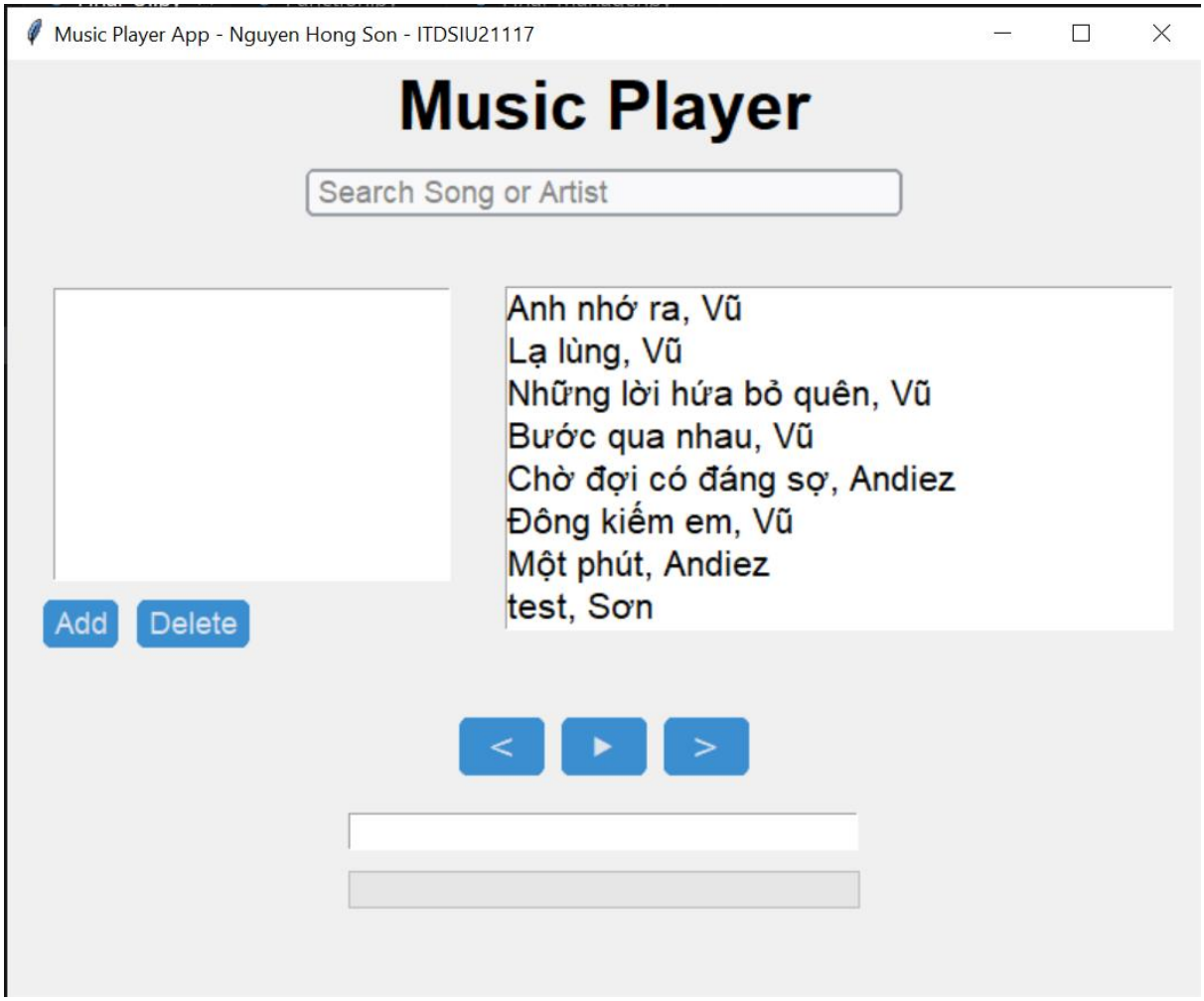


Figure II.B.1. Application UI

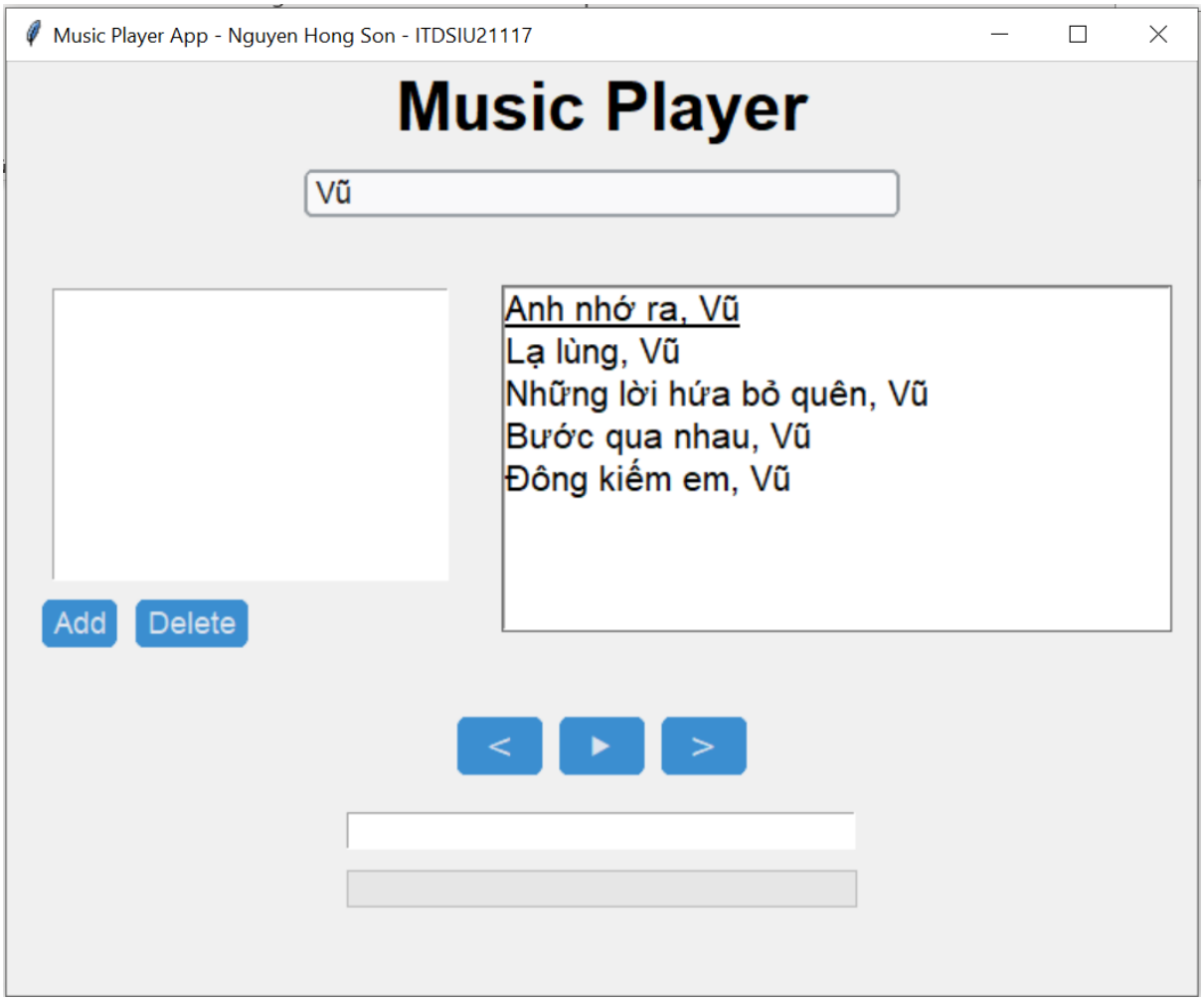


Figure II.B.2. Searching results

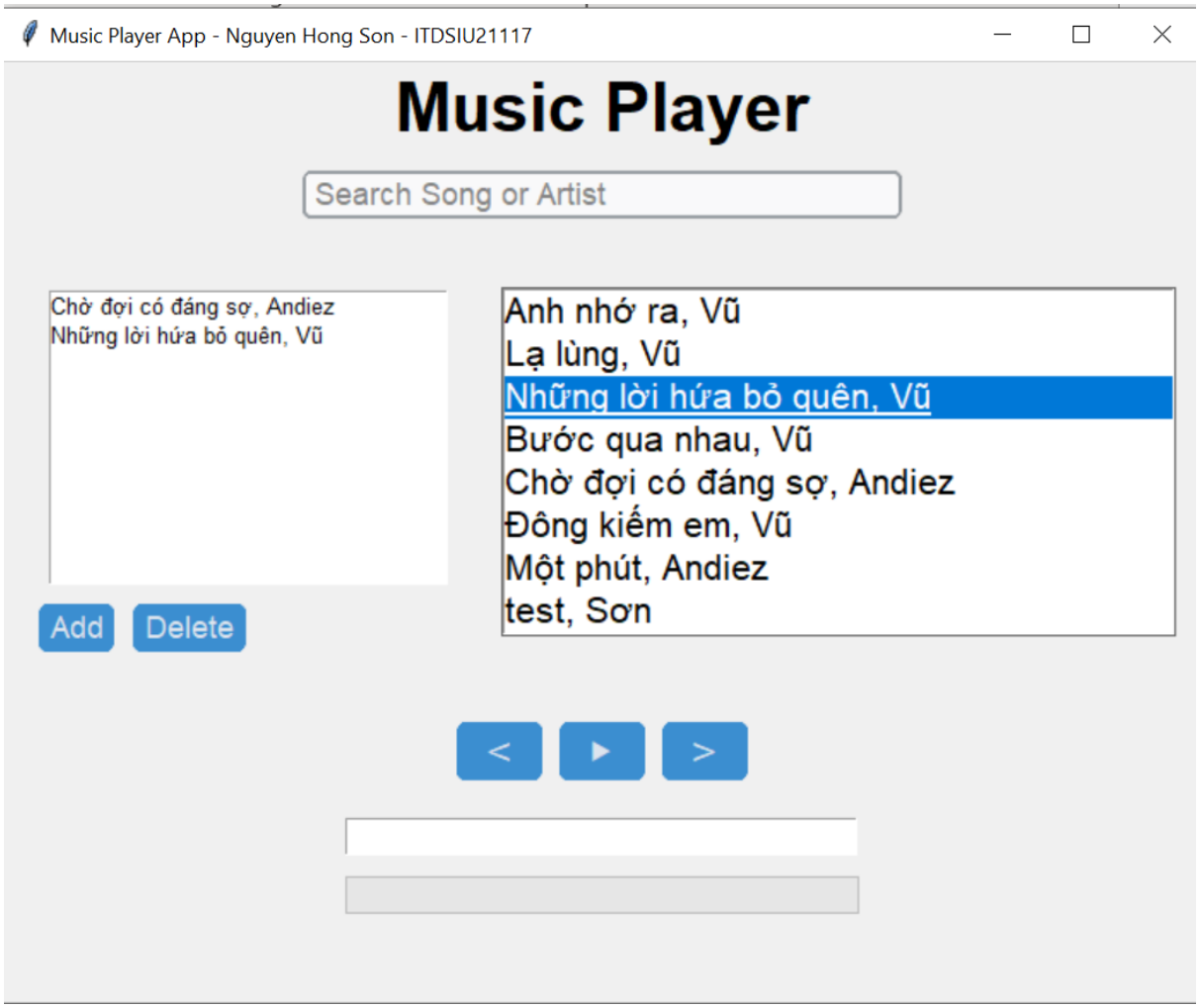


Figure II.B.3. Screen of adding song into playlist

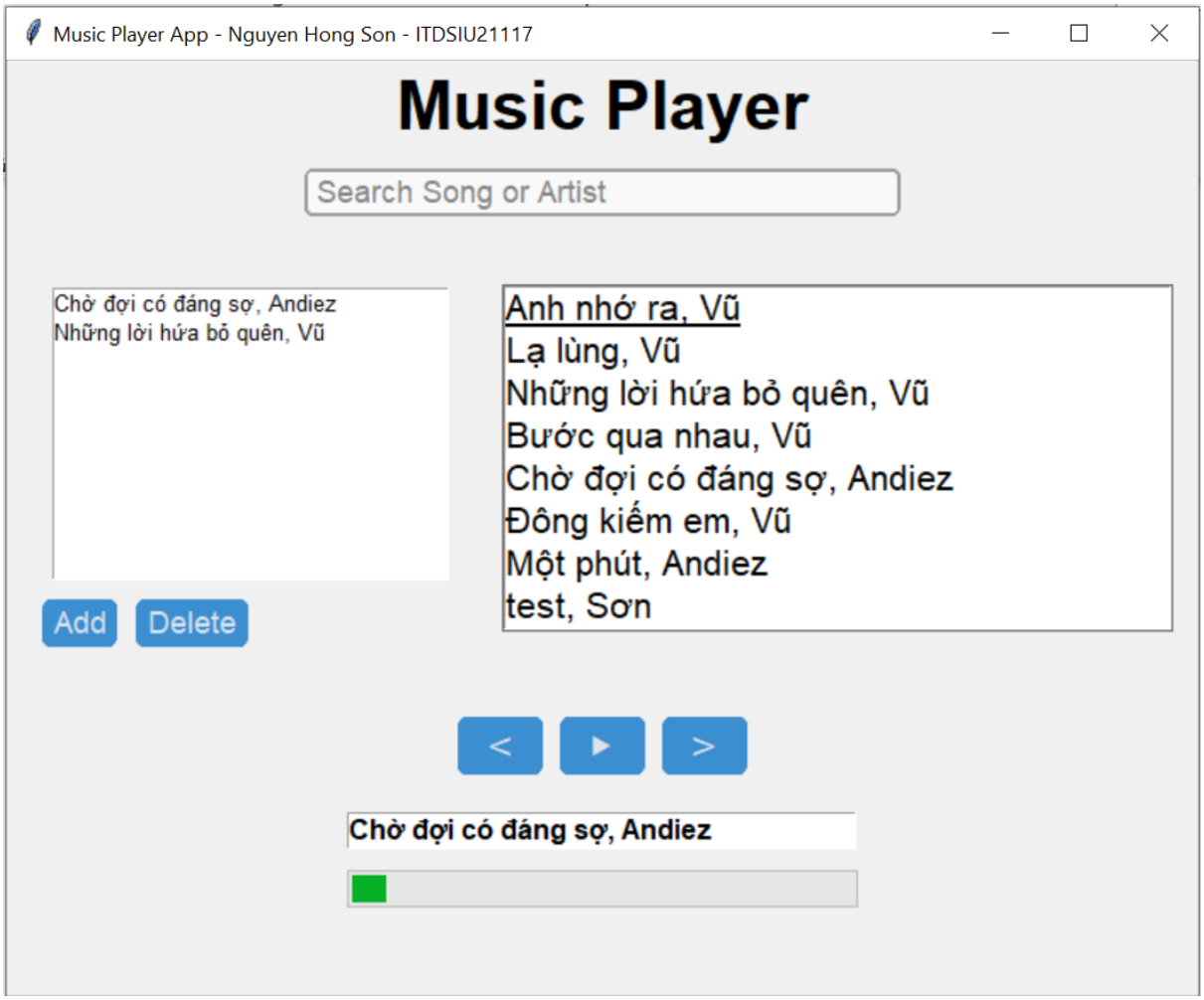


Figure II.B.4. Window of Playing the current song

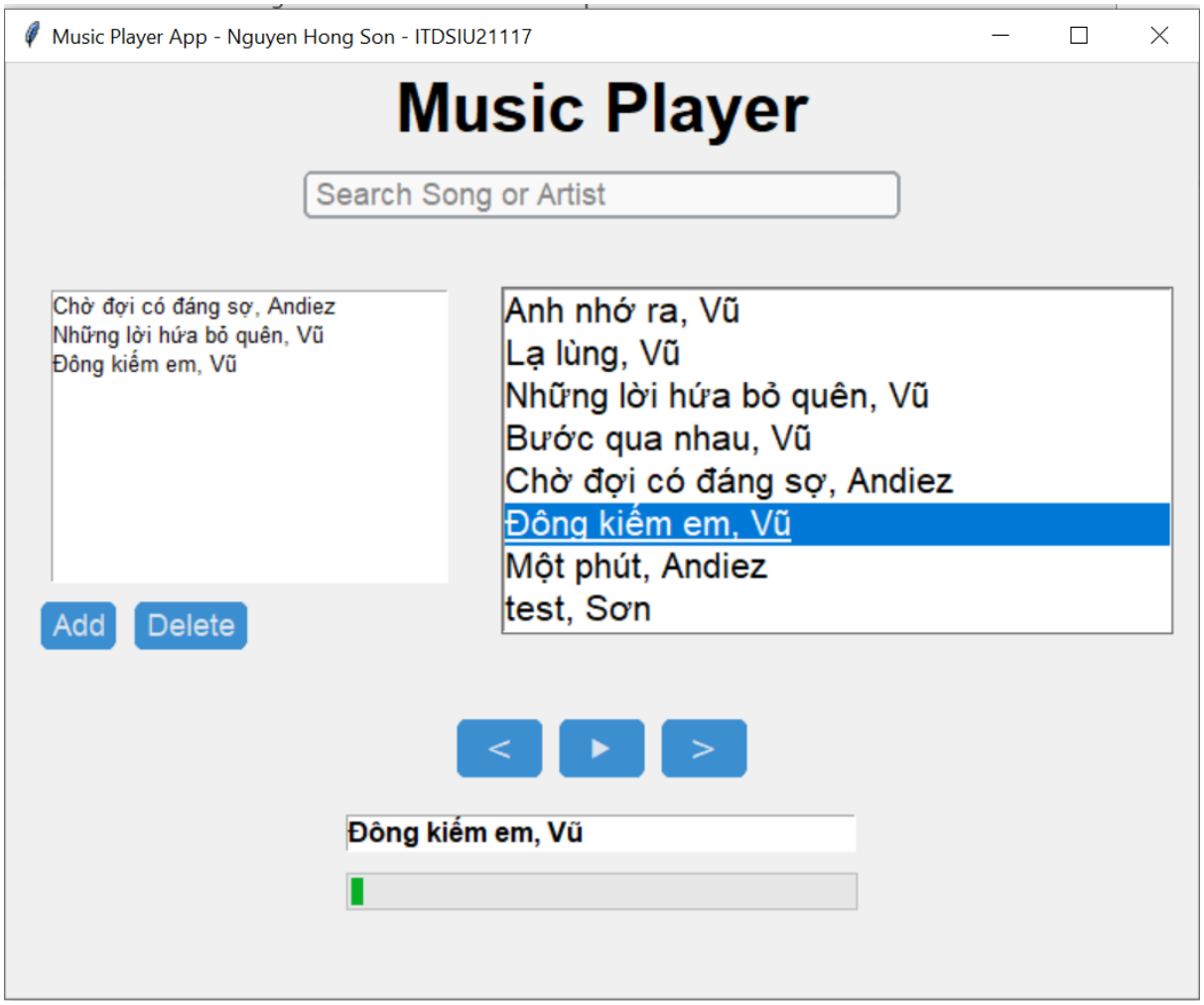


Figure II.B.5. Go through to the last song

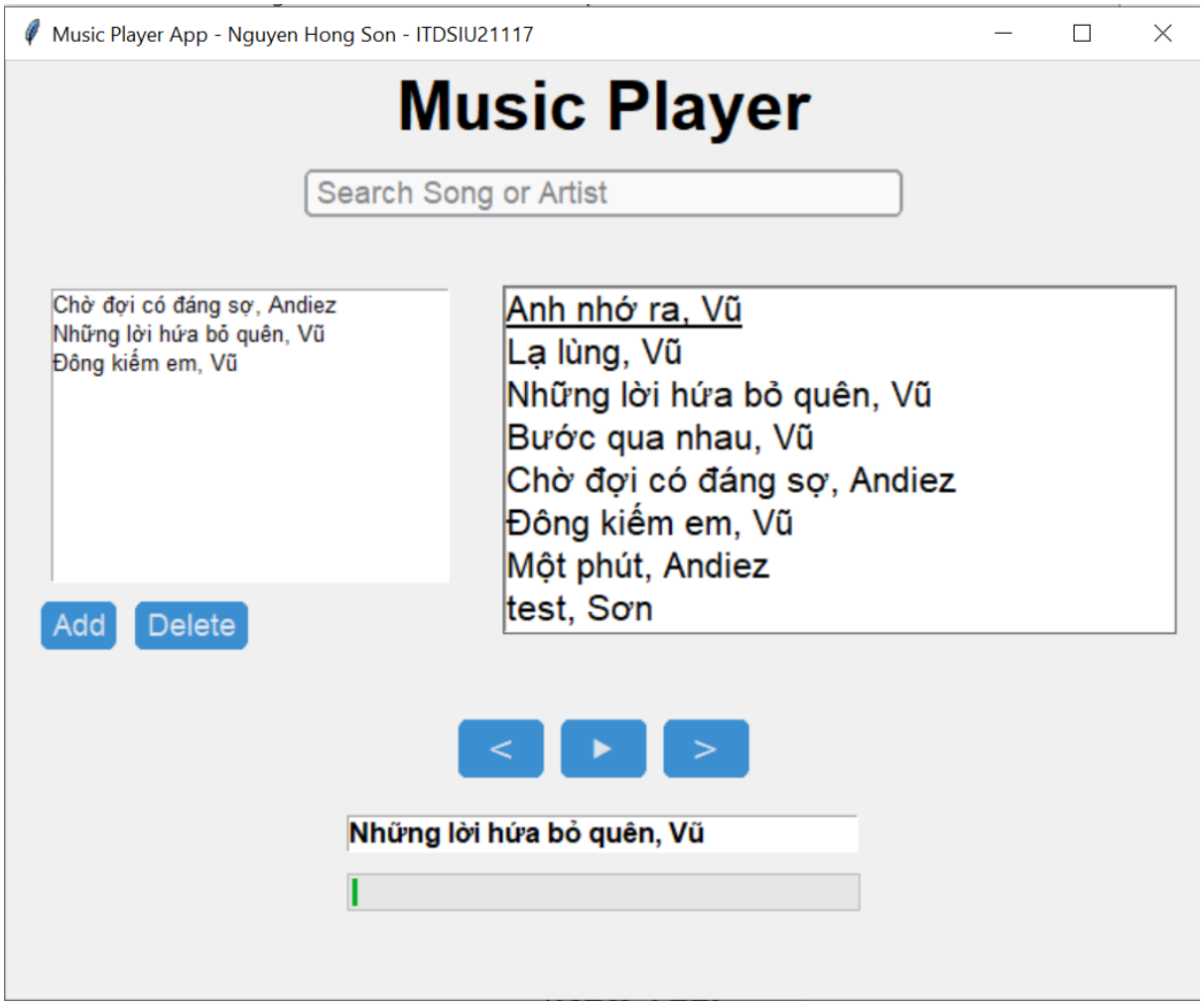


Figure II.B.6. Go back to the previous song

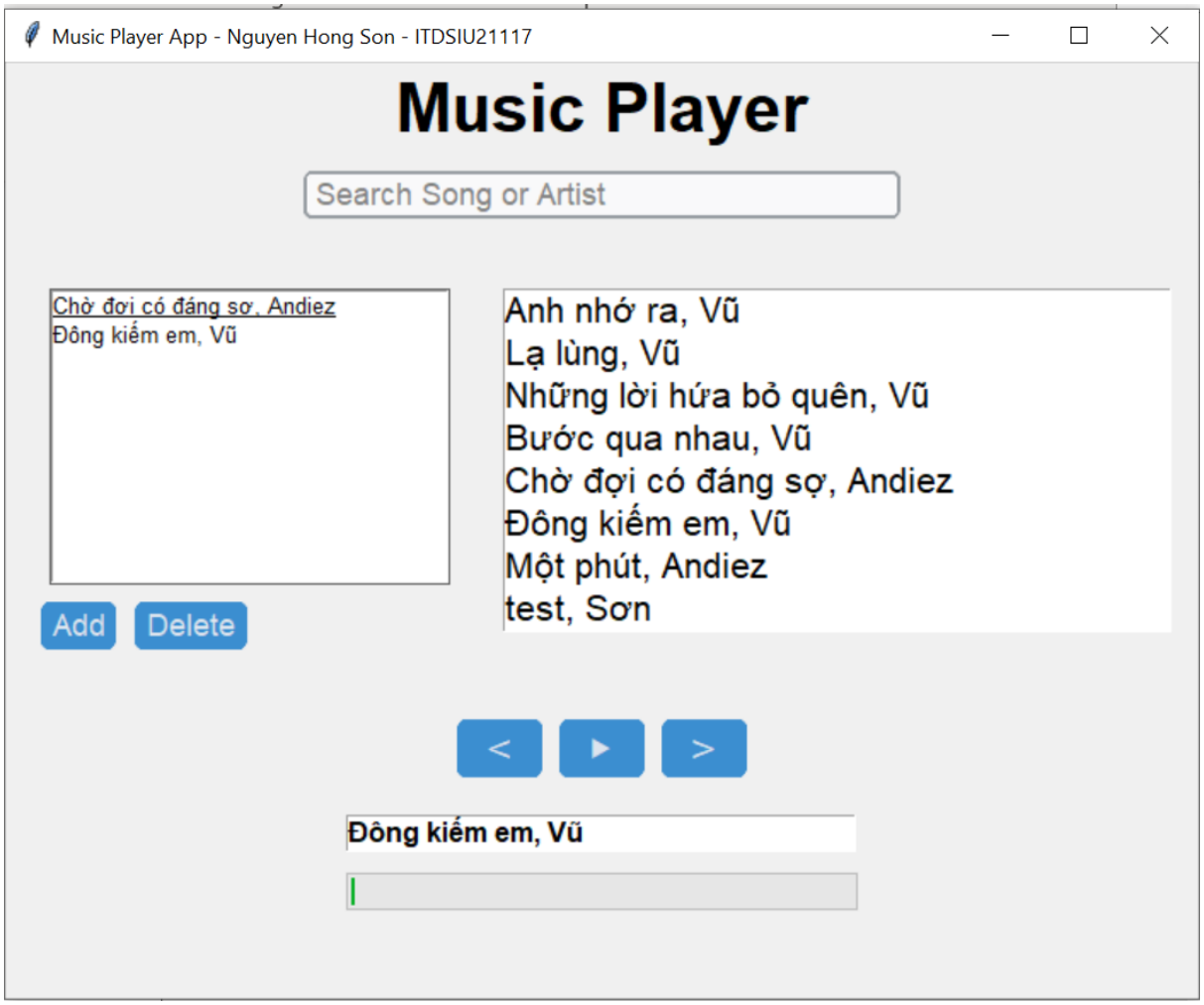


Figure II.B.7. Delete a song in the middle

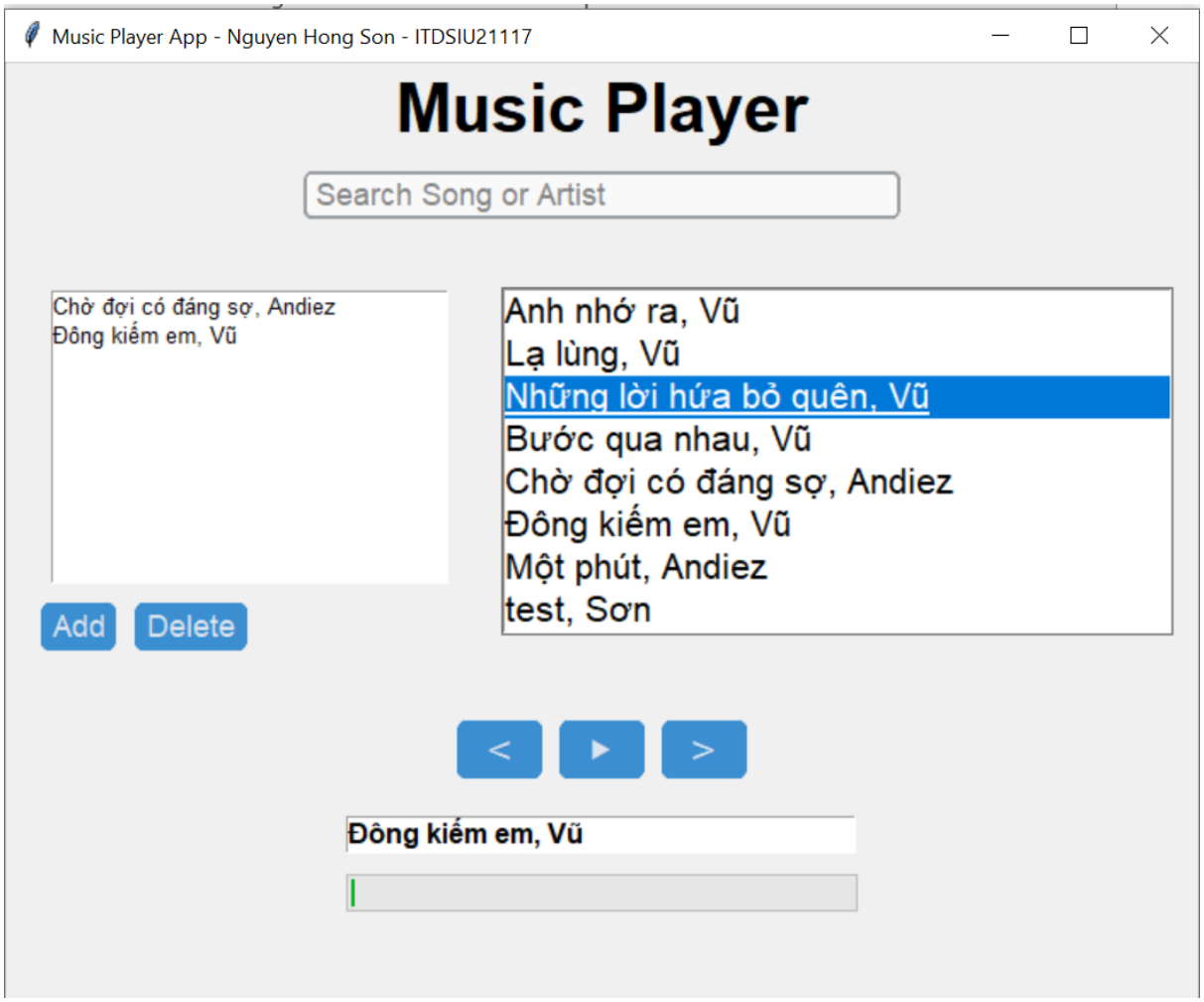


Figure II.B.8. Select the song for deleting

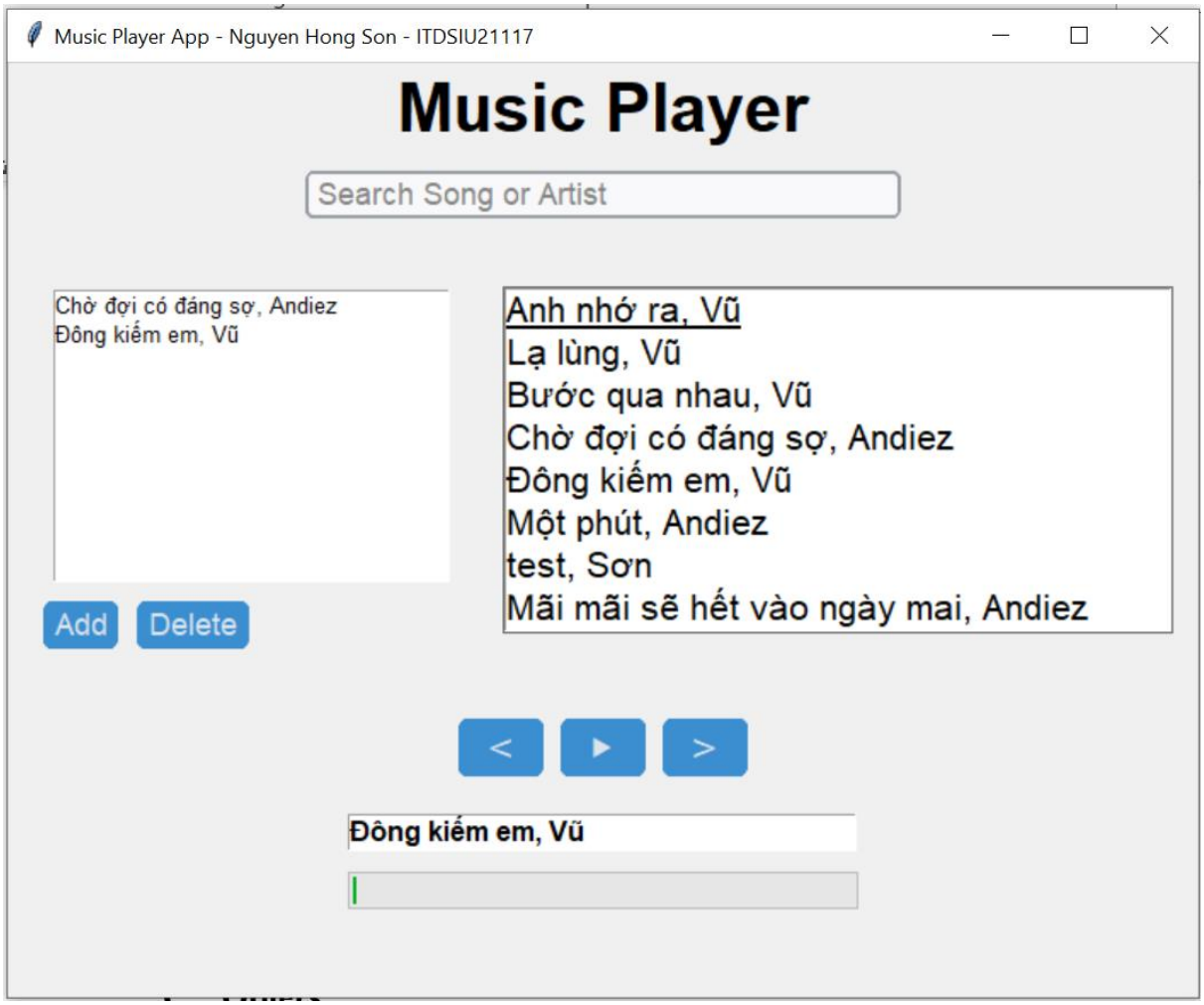
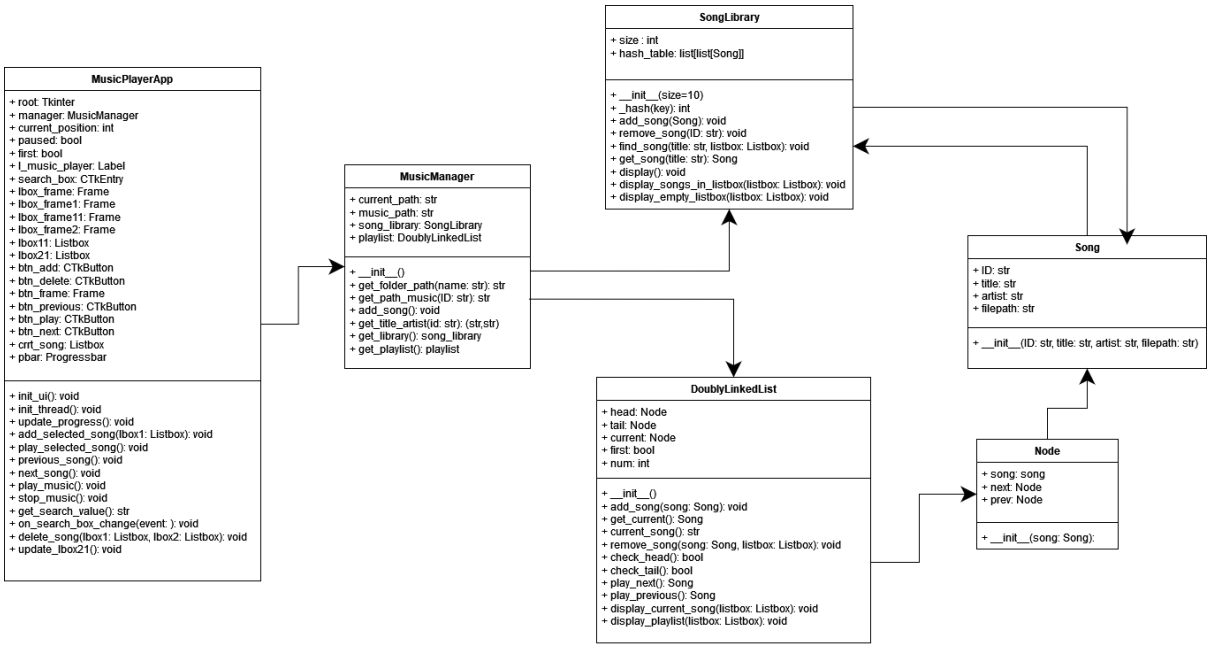


Figure II.B.9. Delete a song in song library

C. Others

- **UML diagram:**



III. DATA STRUCTURE AND ALGORITHMS APLLICATION

A. Hash table

Code:

```
class Song:
    tabnine: test | explain | document | ask
    def __init__(self, ID, title, artist, filepath):
        self.ID = ID
        self.title = title
        self.artist = artist
        self.filepath = filepath

class SongLibrary:
    tabnine: test | explain | document | ask
    def __init__(self, size=10):
        self.size = size
        self.hash_table = [[] for _ in range(self.size)]

    tabnine: test | explain | document | ask
    def _hash(self, key):
        return hash(key) % self.size
```

Purpose: The purpose of using a hash table in the provided SongLibrary class is to quickly add, retrieve, and remove songs from the library, making it an efficient data structure for managing a collection of songs.

Implementation Details:

1. Hash method:

Code:

```
def _hash(self, key):
    return hash(key) % self.size
```

- Generate a hash value for a given key (in this case, the song ID) and mapping it to an index within the hash table.
- The time complexity of the _hash method is O(1), which means it runs in constant time, regardless of the size of the input.
- By combining the constant-time operations of the hash() function and the modulo operation, the overall time complexity of the _hash method is O(1).

2. Add song method:

Code:

```
def add_song(self, song):
    index = self._hash(song.ID)
    for existing_song in self.hash_table[index]:
        if existing_song.ID == song.ID:
            return None  # Song already exists, do not add
    self.hash_table[index].append(song)
```

- The add_song method in the SongLibrary class is responsible for adding a new song to the hash table.
- Complexity:
    1. index = self._hash(song.ID):
        - This step is called the _hash method, which we've already seen has a time complexity of O(1).
    2. for existing_song in self.hash_table[index]:
        - This step iterates through the list of songs stored at the calculated index in the hash table.
        - In the average case, where collisions are well distributed, the number of collisions is expected to be constant (or O(1)).

- However, in the worst case, where all songs hash to the same index, the time complexity of this loop would be O(n), where n is the number of songs at that index.

3. if existing_song.ID == song.ID: return None:
    - This step checks if the song being added already exists in the hash table.
    - The time complexity of this comparison is O(1), as it involves a simple equality check between two IDs.
4. self.hash_table[index].append(song):
    - This step appends the new song to the list at the calculated index in the hash table.
    - The time complexity of the append operation is O(1) on average, as it involves adding the song to the end of the list.

Conclusion:

- **Average case: O(1) + O(1) + O(1) + O(1) = O(1)**
  In the average case, where collisions are well-distributed, the method runs in constant time.
- **Worst case: O(1) + O(n) + O(1) + O(1) = O(n)**
  In the worst case, where all songs hash to the same index, the method's time complexity is linear with respect to the number of songs at that index.

3. Remove song method:

Code:

```
def remove_song(self, ID):
    index = self._hash(ID)
    for i, song in enumerate(self.hash_table[index]):
        if song.ID == ID:
            del self.hash_table[index][i]
```

- The remove_song method in the SongLibrary class is responsible for removing a song from the hash table, given its ID.
- Complexity:
    1. index = self._hash(ID):

- This step is called the _hash method, which has a time complexity of O(1).

2. for i, song in enumerate(self.hash_table[index]):
    - This step iterates through the list of songs stored at the calculated index in the hash table.
    - In the average case, where collisions are well distributed, the number of collisions is expected to be constant (or O(1)).
    - However, in the worst case, where all songs hash to the same index, the time complexity of this loop would be O(n), where n is the number of songs at that index.
3. if song.ID == ID: del self.hash_table[index][i]:

    - This step checks if the current song's ID matches the provided ID, and if so, removes the song from the list at the calculated index.
    - The time complexity of the del operation is O(n), where n is the number of songs remaining in the list after the removal.
    - However, in the average case, where the song to be removed is near the end of the list, the time complexity is closer to O(1).

Conclusion:

- **Average case: O(1) + O(1) + O(1) = O(1)**
  In the average case, where collisions are well-distributed and the song to be removed is near the end of the list, the method runs in constant time.
- **Worst case: O(1) + O(n) + O(n) = O(n)**
  In the worst case, where all songs hash to the same index and the song to be removed is at the beginning of the list, the method's time complexity is linear with respect to the number of songs at that index.

4. Find song method:

Code:

```
def find_song(self, title, listbox):
    for bucket in self.hash_table:
        for song in bucket:
            if title.lower() in song.title.lower() or title.lower() in song.artist.lower():
                listbox.insert(tk.END, f"{song.title}, {song.artist}")
            else:
                listbox.insert(tk.END, None)
```

- The find_song method in the SongLibrary class is responsible for finding songs that match a given title (or artist) and displaying the results in a listbox.
- Complexity:
    1. for bucket in self.hash_table:
        - This step iterates through all the buckets (lists) in the hash table.
        - The time complexity of this loop is O(m), where m is the number of buckets (or the size of the hash table).
    2. for song in bucket:
        - This step iterates through the list of songs stored in each bucket.
        - In the average case, where collisions are well distributed, the number of collisions is expected to be constant (or O(1)).
        - However, in the worst case, where all songs hash to the same index, the time complexity of this loop would be O(n), where n is the number of songs at that index.
    3. if title.lower() in song.title.lower() or title.lower() in song.artist.lower():
        - This step checks if the given title (or artist) matches the title or artist of the current song.
        - The time complexity of this operation is O(k), where k is the length of the title (or artist) being searched.
    4. listbox.insert(tk.END, f"{song.title}, {song.artist}") or listbox.insert(tk.END, None):
        - This step inserts the song information (or None) into the listbox.
        - The time complexity of the insert operation is O(1) on average, as it involves adding the item to the end of the list.

    Conclusion:

    - **Average case: O(m) + O(1) + O(k) + O(1) = O(m + k)**

      In the average case, where collisions are well-distributed, the method's time complexity is linear with respect to the size of the hash table and the length of the search term.
    - **Worst case: O(m) + O(n) + O(k) + O(1) = O(m * n + k)**

      In the worst case, where all songs hash to the same index, the method's time complexity is linear with respect to the size of the hash table, the number of songs at each index, and the length of the search term.

5. Get song method:

Code:

```
def get_song(self, title):
    for bucket in self.hash_table:
        for song in bucket:
            if title.lower() in song.title.lower() or title.lower() in song.artist.lower():
                return song
    return None
```

- The get_song method in the SongLibrary class is responsible for finding and returning a song that matches the given title (or artist)
- Complexity:
    1. for bucket in self.hash_table:
        - This step iterates through all the buckets (lists) in the hash table.
        - The time complexity of this loop is O(m), where m is the number of buckets (or the size of the hash table).
    2. for song in bucket:
        - This step iterates through the list of songs stored in each bucket.
        - In the average case, where collisions are well distributed, the number of collisions is expected to be constant (or O(1)).
        - However, in the worst case, where all songs hash to the same index, the time complexity of this loop would be O(n), where n is the number of songs at that index.
    3. if title.lower() in song.title.lower() or title.lower() in song.artist.lower():
        - This step checks if the given title (or artist) matches the title or artist of the current song.
        - The time complexity of this operation is O(k), where k is the length of the title (or artist) being searched.
    4. return song or return None:
        - This step returns the found song or None if no matching song is found.
        - The time complexity of this operation is O(1).

Conclusion:

- **Average case: O(m) + O(1) + O(k) + O(1) = O(m + k)**
  In the average case, where collisions are well-distributed, the method's time complexity is linear with respect to the size of the hash table and the length of the search term.
- **Worst case: O(m) + O(n) + O(k) + O(1) = O(m * n + k)**

6. Display songs in listbox method:

Code:

```
def display_songs_in_listbox(self, listbox):
    listbox.delete(0, tk.END)
    for bucket in self.hash_table:
        for song in bucket:
            listbox.insert(tk.END, f"{song.title}, {song.artist}")
```

- The display_songs_in_listbox method in the SongLibrary class is responsible for populating a Tkinter Listbox widget with the titles and artists of all the songs in the music library.
- Complexity:
    1. listbox.delete(0, tk.END):
        - This step clears the existing contents of the Listbox widget.
        - The time complexity of this operation is O(n), where n is the number of items in the Listbox.
    2. for bucket in self.hash_table:
        - This step iterates through all the buckets (lists) in the hash table.
        - The time complexity of this loop is O(m), where m is the number of buckets (or the size of the hash table).
    3. for song in bucket:
        - This step iterates through the list of songs stored in each bucket.
        - In the average case, where collisions are well distributed, the number of collisions is expected to be constant (or O(1)).
        - However, in the worst case, where all songs hash to the same index, the time complexity of this loop would be O(n), where n is the number of songs at that index.
    4. listbox.insert(tk.END, f"{song.title}, {song.artist}"):
        - This step inserts the title and artist of the current song into the Listbox widget.
        - The time complexity of this operation is O(log(k)), where k is the number of items in the Listbox.

Conclusion:

- **Average case: O(n) + O(m) + O(1) + O(log(k)) = O(n + m + log(k))**
  In the average case, where collisions are well-distributed, the method's time complexity is linear with respect to the size of the hash table and the length of the song details being printed.
- **Worst case: O(n) + O(m) + O(n) + O(log(k)) = O(n * m + log(k))**

In the worst case, where all songs hash to the same index, the method's time complexity is linear with respect to the size of the hash table, the number of songs at each index, and the length of the song details being printed.

7. Display empty listbox method:

Code:

- The display_empty_listbox method in the SongLibrary class is responsible for displaying an empty Listbox widget, which can be useful in scenarios where the music library is empty or no songs match the current search criteria.
- Complexity:
    1. listbox.delete(0, tk.END):
        - This step clears the existing contents of the Listbox widget.
        - The time complexity of this operation is O(n), where n is the number of items in the Listbox.
    2. listbox.insert(tk.END, None):
        - This step inserts a single None value into the Listbox widget.
        - The time complexity of this operation is O(log(k)), where k is the number of items in the Listbox.

Conclusion:

- **Time complexity: O(n) + O(log(k)) = O(n + log(k))**
  The method's time complexity is linear with respect to the size of the Listbox (for clearing the contents) and logarithmic with respect to the number of items in the Listbox (for inserting the None value).

Summary:

- The SongLibrary class utilizes a hash table to efficiently manage a collection of songs. This allows for quick addition, retrieval, and removal of songs
- The hash table in the SongLibrary class is designed for efficient song management, with most operations having constant time complexity in the average case. The worst-case scenarios are handled gracefully but may degrade to linear complexity depending on hash collisions.

B. Doubly Linked List:

Code:

```
class Node:
    tabnine: test | explain | document | ask
    def __init__(self, song):
        self.song = song
        self.next = None
        self.prev = None

class DoublyLinkedList:
    tabnine: test | explain | document | ask
    def __init__(self):
        self.head = None
        self.tail = None
        self.current = None
        self.first = None
        self.num = 0
```

Purpose: The purpose of using a doubly-linked list to implement a music playlist is primarily to enable efficient navigation and manipulation of the playlist. In a music player application, users often need to navigate back and forth between songs, such as going to the previous or next song in the playlist. With a doubly-linked list, this bi-directional traversal can be accomplished easily by following the next and previous pointers of the nodes. This allows for the efficient implementation of features like "Previous" and "Next" buttons in the music player interface, as the user can quickly move to the desired song without having to traverse the entire list sequentially.

Implementation Details:

1. Add song method:

Code:

```
    def add_song(self, song):
        new_node = Node(song)
        if  self.head == None:
            self.head = self.tail = self.current = new_node
            self.first = True
        else:
            self.first = False
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
        self.num +=1
```

- The add_song method in the SongLibrary class is responsible for adding a new song to the linked list data structure that stores the songs.
- Complexity:

Conclusion:

- **The overall time complexity of the add_song method is O(1), as it involves a constant number of operations, regardless of the size of the linked list.**

2. Get current and current song method:

Code:

```
def get_current(self):
    return self.current.song

tabnine: test | explain | document | ask
def current_song(self):
    return self.current.song.filepath if self.current else None
```

- The get_current method in the SongLibrary class is responsible for retrieving the current song in the library.
- The current_song method in the SongLibrary class is responsible for returning the file path of the current song in the library.
- Complexity:

1. return self.current.song:
    - This step simply returns the song attribute of the Node object pointed to by the self.current pointer.
    - The time complexity of this operation is O(1), as it involves constant-time attribute access.
2. return self.current.song.filepath:
    - This step returns the filepath attribute of the song object stored in the Node object pointed to by the self.current pointer.
    - The time complexity of this operation is O(1), as it involves constant-time attribute access.

Conclusion:

- **The get_current method has a time complexity of O(1), making it a highly efficient and performance-optimized method within the SongLibrary class. This ensures that users can quickly and easily retrieve the current song in the library, contributing to a seamless and responsive user experience.**
- **The overall time complexity of the current_song method is O(1), as it involves a constant number of operations, regardless of the size of the library.**

3. Remove song method:

Code:

```
def remove_song(self, song, listbox):
    if song.ID == self.current.song.ID:
        if self.current == self.head and self.current == self.tail:
            self.head = self.tail = self.current = None
            self.num = 0
            pygame.mixer.music.stop()
            listbox.delete(0, 0)
            return None

        else :
            if self.current.prev:
                self.current.prev.next = self.current.next

            else :
                self.head = self.current.next
                self.current = self.head
                self.num -= 1
                pygame.mixer.music.stop()
                listbox.delete(0, 0)
                return None
            if self.current.next:
                self.current.next.prev = self.current.prev

            else :
                self.tail = self.current.prev
                self.current = self.tail


            if self.current != self.tail:
                self.current = self.current.next
        self.num -= 1
        pygame.mixer.music.stop()
        listbox.delete(0, 0)
    else :
        curr = self.head
        for i in range(self.num):
            if curr.song.ID == song.ID:
                if curr.prev:
                    curr.prev.next = curr.next
                else:
                    self.head = curr.next
                if curr.next:
                    curr.next.prev = curr.prev
                else:
                    self.tail = curr.prev
                self.num -= 1
                break
            curr = curr.next
    return None
```

- The remove_song method in the DoublyLinkedList class is responsible for removing a specific song from the library or the playlist.
- Complexity:
    1. if song.ID == self.current.song.ID:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.
        - The time complexity of this operation is O(1), as it involves a constant-time comparison.
    2. if self.current == self.head and self.current == self.tail:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.
        - The time complexity of this operation is O(1), as it involves a constant-time comparison.
    3. The rest of the operations in the first if block:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.
        - The time complexity of this operation is O(1), as it involves a constant-time comparison.
    4. The operations in the second if block:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.
        - The time complexity of this operation is O(1), as it involves a constant-time comparison.
    5. The operations in the third if block:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.
        - The time complexity of this operation is O(1), as it involves a constant-time comparison.
    6. The operation in the final if block:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.
        - The time complexity of this operation is O(1), as it involves a constant-time comparison.
    7. self.num -= 1:
        - This step checks if the ID of the song argument matches the ID of the current song in the library.

- The time complexity of this operation is O(1), as it involves a constant-time comparison.

8. The else block:
    - This step checks if the ID of the song argument matches the ID of the current song in the library.
    - The time complexity of this operation is O(1), as it involves a constant-time comparison.

Conclusion:

- **The method efficiently handles the case where the current song is being removed, with a time complexity of O(1) for that scenario. However, the linear search through the doubly-linked list for non-current songs contributes to the overall O(n) time complexity of the method.**

4. Check head and tail method:

Code:

```
def check_head(self):
    if self.first :
        return True
    else :
        if self.current == self.head  :
            return True
        else :
            return False

tabnine: test | explain | document | ask
def check_tail(self):
    if self.first :
        return True
    else :
        if self.current == self.tail  :
            return True
        else :
            return False
```

- The check_head and check_tail methods in the DoublyLinkedList class are responsible for checking if the current song pointer (self.current) is pointing to the head or tail of the song playlist
- Complexity:

    Conclusion:

    - **The overall time complexity of both the check_head and check_tail methods is O(1), as they involve a constant number of operations.**

- These methods are useful for determining the position of the current song pointer in the DoublyLinkedList class. By checking if the self.current pointer is pointing to the head or tail of the doubly linked list, these methods can provide valuable information about the current state of the doubly linked list.
- The constant-time complexity of these methods ensures that they can be executed quickly, without introducing any significant performance overhead. This is important for maintaining a responsive and seamless user experience when interacting with the DoublyLinkedList class.

5. Play next and previous method:

Code:

```
    def play_next(self):
        if self.current and self.current.next:
            self.current = self.current.next
            return self.current.song
        return None

    tabnine: test | explain | document | ask
    def play_previous(self):
        if self.current and self.current.prev:
            self.current = self.current.prev
            return self.current.song
        return None
```

- The play_next and play_previous methods in the DoublyLinkedList class are responsible for moving the self.current pointer to the next or previous song in the playlist, and returning the corresponding song object.
- Complexity:

Conclusion:

- **The overall time complexity of both the play_next and play_previous methods is O(1), as they involve a constant number of operations.**
- **Meaning they can be executed in constant time, regardless of the size of the linked list. This is because the methods perform a constant number of pointer updates and do not need to traverse the entire list.**

6. Display current song and playlist method:

Code:

```
def display_current_song(self, listbox):
    listbox.delete(0, 0)
    listbox.insert(tk.END,f"{self.current.song.title}, {self.current.song.artist}")

tabnine: test | explain | document | ask
def display_playlist(self, listbox):
    listbox.delete(0, tk.END)
    crr_song = self.head
    for i in range (self.num):
        listbox.insert(tk.END,f"{crr_song.song.title}, {crr_song.song.artist}")
        crr_song = crr_song.next
```

- The display_current_song and display_playlist methods in the DoublyLinkedList class are responsible for updating the GUI's listbox to display the currently playing song and the full playlist.
- Complexity:
    1. display_current_song(self, listbox):
        - Overall, the time complexity of the display_current_song method is O(1), as it performs a constant number of operations, regardless of the size of the playlist.
    2. display_playlist(self, listbox):

       2.1 listbox.delete(0, tk.END):
        - This step removes all the currently displayed songs from the listbox.
        - The time complexity of this operation is O(n), where n is the number of songs in the listbox, as it involves removing all the items.

       2.2 for i in range (self.num):
        - This loop iterates through all the songs in the playlist (represented by the self.num attribute).
        - The time complexity of this loop is O(n), where n is the number of songs in the playlist.

       2.3 listbox.insert(tk.END,f"{crr_song.song.title}, {crr_song.song.artist}"):
        - This loop iterates through all the songs in the playlist (represented by the self.num attribute).
        - The time complexity of this loop is O(n), where n is the number of songs in the playlist.

Conclusion:

- **The display_current_song method has a time complexity of O(1), as it performs a constant number of operations.**
- **The display_playlist method has a time complexity of O(n), where n is the number of songs in the playlist, as it iterates through the entire playlist to update the listbox**

Summary:

- The implementation of a doubly-linked list for managing a music playlist offers efficient navigation and manipulation capabilities. Key operations such as adding and retrieving songs (add_song, get_current, current_song) are optimized with O(1) time complexity, ensuring quick access regardless of playlist size.
- However, the remove_song method exhibits O(1) complexity only when removing the current song, with a potential worst-case scenario of O(n) for non-current songs due to list traversal.
- While displaying the current song (display_current_song) remains constant in complexity (O(1)), updating the entire playlist (display_playlist) involves iterating through the list, resulting in O(n) complexity.
- In summary, the doubly-linked list structure enhances user interaction with the music playlist by ensuring fast access and manipulation of songs, contributing to a seamless and responsive user experience in a music player application.

C. Application of Design Pattern

Design Pattern: Singleton

Code:

```python
class MusicPlayerApp:
    _instance = None

    tabnine: test | explain | document | ask
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance
    tabnine: test | explain | document | ask
    def __init__(self, root):
        self.root = root
        self.root.title("Music Player App - Nguyen Hong Son - ITDSIU21117")
        self.root.geometry("700x550")
        manager = FM.MusicManager()
        self.lb = manager.get_library()
        self.pl = manager.get_playlist()

        pygame.mixer.init()

        self.current_position = 0
        self.paused = False
        self.first = True

        self.init_ui()
        self.init_thread()
```

- In the __new__ method, the Singleton Pattern is implemented by checking if an instance of the class already exists (cls._instance). If an instance exists, the existing instance is returned. If no instance exists, a new instance is created using the super().__new__(cls) method.
- In the __init__ method, the Singleton Pattern ensures that only one instance of the MusicPlayerApp class is created. The root attribute is initialized with the provided root parameter, which represents the main window of the application.
- By using the Singleton Pattern in the MusicPlayerApp class, I can ensure that only one instance of the music player application is created, which helps maintain consistency and prevents multiple instances from being created.

IV. CONCLUSION

A. Achieved Goals

The development of the music player application in Python successfully met several key objectives:

1. Implement a library feature to store multiple songs in a sequence.
2. Implement a playlist feature to store and play multiple songs in a sequence.
3. Implement a progress bar to indicate the current song's progress in real-time.
4. Implement a feature to handle the end of a song and automatically play the next song in the playlist.
5. Implement a feature to pause, play, and skip songs.
6. Implement a feature to add and delete songs from the playlist or library.
7. Implement a feature to display the current song's information, the playlist, the song library.
8. Implement a search feature to search for songs or artists by name.
9. Implement a feature to display and update the search results real-time.

B. Limitations

The application is well-structured and follows best practices, making it easy to understand and maintain. However, there are some limitations to the current version of the music player:

1. The application does not include any user authentication or authorization mechanisms.
2. The application does not include any support for playing songs from different formats or streaming songs from online sources.
3. The application does not include any support for managing playlists or saving the current playlist to a file.
4. The application does not include any support for managing user preferences or settings.
5. The application does not include any support for managing song metadata, such as album art or song lyrics.
6. The application does not include any support for managing song ratings or reviews.

7. The application does not include any support for managing song genres or categories.
8. The application does not include any support for managing song playlists or creating playlists based on user preferences or song attributes.
9. The application does not include any support for managing song recommendations or suggesting similar songs based on user preferences or song attributes.

C. Future Enhancements

To further improve the music player application, the following enhancements are recommended:

1. Implement a feature to shuffle the playlist
2. Implement a feature to repeat a single song
3. Implement a feature to display song lyrics
4. Implement a feature to display album artwork
5. Implement a feature to display song ratings and reviews
6. Implement a feature to display song recommendations based on the user's listening habits or preferences and user listening history.

V. REFERENCES

1. Custom Tkinter Widgets. (n.d.). Retrieved June 22, 2024, from https://customtkinter.tomschimansky.com/
2. GeeksforGeeks. (n.d.). Doubly Linked List in Python. Retrieved June 22, 2024, from https://www.geeksforgeeks.org/doubly-linked-list-in-python/
3. GeeksforGeeks. (n.d.). Implementation of Hash Table in Python using Separate Chaining. Retrieved June 22, 2024, from https://www.geeksforgeeks.org/implementation-of-hash-table-in-python-using-separate-chaining/
4. Psikka. (n.d.). GitHub - psikka/projects-yt: MusicPlayer. GitHub. Retrieved June 22, 2024, from https://github.com/achudnova/projects-yt/tree/main/MusicPlayer

5. Psikka. (n.d.). YouTube - Psikka: ChatGPT Implementation. Retrieved June 22, 2024, from https://www.youtube.com/watch?v=RqNgCgu5Vw8