19CSE212 / Data Structures & Algorithms



B-Tech/II Year CSE/IV Semester

|  |  |
| --- | --- |
| **NAME** | **ROLL NO** |
| **M. JESWANTH MIRACLE BABU** | **CB.EN.U4CSE21228** |

Amrita School of Engineering, Coimbatore

Department of Computer Science and Engineering 2022 -2023 Even Semester

Topic : Playlist Manager

Existing Data Structure:  
Linked List or Arrays  
Code:-

class Song:

    def \_init\_(self, title, artist, duration):

        self.title = title

        self.artist = artist

        self.duration = duration

        self.next = None  # Pointer to the next song in the playlist

class Playlist:

    def \_init\_(self):

        self.head = None  # Pointer to the first song in the playlist

        self.tail = None  # Pointer to the last song in the playlist

    def addSong(self, title, artist, duration):

        newSong = Song(title, artist, duration)

        if not self.head:

            # If the playlist is empty, set the new song as the head and tail

            self.head = newSong

            self.tail = newSong

        else:

            # Otherwise, add the new song to the end of the playlist

            self.tail.next = newSong

            self.tail = newSong

    def removeSong(self, title):

        current = self.head

        prev = None

        while current:

            if current.title == title:

                # If the current song matches the title, remove it from the playlist

                if current == self.head:

                    # If the song is the head of the playlist, update the head pointer

                    self.head = current.next

                elif current == self.tail:

                    # If the song is the tail of the playlist, update the tail pointer

                    self.tail = prev

                    self.tail.next = None

                else:

                    # Otherwise, update the previous song's pointer to skip over the current song

                    prev.next = current.next

                return True  # Return True to indicate that the song was successfully removed

            prev = current

            current = current.next

        return False  # Return False to indicate that the song was not found in the playlist

    def shufflePlaylist(self):

        # Convert the linked list to an array

        songsArray = []

        current = self.head

        while current:

            songsArray.append(current)

            current = current.next

        import random

        # Fisher-Yates shuffle algorithm

        for i in range(len(songsArray) - 1, 0, -1):

            j = random.randint(0, i)

            songsArray[i], songsArray[j] = songsArray[j], songsArray[i]

        # Convert the shuffled array back to a linked list

        self.head = songsArray[0]

        self.tail = songsArray[-1]

        for i in range(len(songsArray) - 1):

            songsArray[i].next = songsArray[i + 1]

        self.tail.next = None

playlist = Playlist()

playlist.addSong("Bohemian Rhapsody", "Queen", "5:55")

playlist.addSong("Stairway to Heaven", "Led Zeppelin", "8:02")

playlist.addSong("Hotel California", "Eagles", "6:30")

print(playlist)

playlist.removeSong("Stairway to Heaven")

print(playlist)

playlist.shufflePlaylist()

print(playlist)

Time Complexity:

Adding a Song:

* Creating a new Song object: O(1)
* Adding the song to the end of the playlist: O(1)

Overall time complexity: O(1)

Removing a Song:

* Searching for the song to remove: O(n)
* Updating the pointers to remove the song from the playlist: O(1)

Overall time complexity: O(n)

Shuffling the Playlist:

* Converting the linked list to an array: O(n)
* Performing the Fisher-Yates shuffle algorithm: O(n)
* Converting the shuffled array back to a linked list: O(n)

Overall time complexity: O(n)

Proposed Hybrid Data Structures:

DOUBLY LINKED LIST AND GRAPH:-

**Code:**

class Song:

    def \_\_init\_\_(*self*, *title*, *artist*, *duration*, *genre*):

*self*.title = *title*

*self*.artist = *artist*

*self*.duration = *duration*

*self*.genre = *genre*

*self*.prev = None

*self*.next = None

class PlaylistManager:

    def \_\_init\_\_(*self*):

*self*.head = None

*self*.tail = None

*self*.graph = {}

    def addSong(*self*, *title*, *artist*, *duration*, *genre*):

        new\_song = Song(*title*, *artist*, *duration*, *genre*)

*if* not *self*.head:

*self*.head = new\_song

*self*.tail = new\_song

*else*:

*self*.tail.next = new\_song

            new\_song.prev = *self*.tail

*self*.tail = new\_song

*self*.graph[new\_song] = []

        current = *self*.head

*while* current != new\_song:

*if* current.genre == new\_song.genre or current.artist == new\_song.artist:

*self*.addConnection(current, new\_song)

            current = current.next

    def removeSong(*self*, *title*):

        current = *self*.head

*while* current:

*if* current.title == *title*:

*if* current.prev:

                    current.prev.next = current.next

*if* current.next:

                    current.next.prev = current.prev

*del* *self*.graph[current]

*return* True

            current = current.next

*return* False

    def addConnection(*self*, *song1*, *song2*):

*if* *song1* in *self*.graph and *song2* in *self*.graph:

*if* *song2* not in *self*.graph[*song1*]:

*self*.graph[*song1*].append(*song2*)

*if* *song1* not in *self*.graph[*song2*]:

*self*.graph[*song2*].append(*song1*)

    def removeConnection(*self*, *song1*, *song2*):

*if* *song1* in *self*.graph and *song2* in *self*.graph:

*self*.graph[*song1*].remove(*song2*)

*self*.graph[*song2*].remove(*song1*)

    def getPlaylist(*self*):

        playlist = []

        current = *self*.head

*while* current:

            playlist.append(current.title)

            current = current.next

*return* playlist

    def getRecommendedSongs(*self*, *song*):

*if* *song* in *self*.graph:

*return* [adj\_song.title *for* adj\_song *in* *self*.graph[*song*]]

*return* []

playlistManager = PlaylistManager()

playlistManager.addSong("Bohemian Rhapsody", "Queen", "5:55","Rock")

playlistManager.addSong("Stairway to Heaven", "Led Zeppelin", "8:02","Rock")

playlistManager.addSong("Hotel California", "Eagles", "6:30","Country")

playlistManager.addSong("Da club", "50 Cent", "6:30","Rock")

playlistManager.addConnection(playlistManager.head, playlistManager.head.next)

playlistManager.addConnection(playlistManager.head.next, playlistManager.tail)

print(playlistManager.getPlaylist())

print(playlistManager.getRecommendedSongs(playlistManager.head))

Output:

['Bohemian Rhapsody', 'Stairway to Heaven', 'Hotel California', 'Da club']

['Stairway to Heaven', 'Da club']

**Time Complexities:**

* **‘addSong’**: The time complexity for adding a song to the playlist is O(1) because it involves updating the tail pointer and adding the song to the graph with no connections initially.
* **‘removeSong’**: The time complexity for removing a song from the playlist is O(n) in the worst case because it involves traversing the playlist to find the song by title.
* **‘addConnection’** and **‘removeConnection’**: The time complexity for adding or removing a connection between two songs in the graph is O(1) because it involves appending or removing an element from a list
* **‘getPlaylist’ and ‘getRecommendedSongs’**: The time complexity for retrieving the playlist as a list of song titles is O(n) because it requires traversing the entire playlist to collect the titles.

**Why Hybrid data can be chosen over existing data structure?**

Hybrid data structures offer advantages over using a single data structure by combining the strengths of multiple structures to optimize various operations and improve overall performance

* **Efficient Playlist Management**: Doubly linked list allows for easy insertion, removal, and traversal of songs in the playlist, ensuring the desired order is maintained.
* **Representation of Relationships**: Graph data structure enables the representation of relationships between songs, such as similarity or artist connections, allowing for features like song recommendations and personalized playlists.
* **Enhanced Recommendation Generation**: The combination of a doubly linked list and a graph enables the playlist manager to utilize both the playlist order and song connections to generate accurate and diverse recommendations for users.

In summary, using a hybrid data structure, such as a combination of a doubly linked list and a graph, in a playlist manager improves playlist management, enables representation of relationships, enhances recommendation generation and provides flexibility and scalability.

**Why was this specific combination of data structures (doubly linked list and graph) chosen over other hybrid data structures for fulfilling the requirements of this playlist manager?**

The choice of a specific hybrid data structure, such as a combination of a doubly linked list and a graph, over other hybrid data structures depend on the specific requirements and goals of the playlist manager.

The requirement for playlist manager to work efficiently:

* The playlist manager should efficiently manage the order of songs in the playlist. It should allow for adding, removing, and rearranging songs in the playlist. Operations like playing the next or previous song, shuffling, and repeating songs should be performed seamlessly.
* The playlist manager should be able to represent relationships between songs. These relationships can be based on factors such as similarity, genre, artist, or user preferences.

Given these requirements and goals, the combination of a doubly linked list and a graph data structure can be a suitable choice.

BINARY SEARCH TREE AND HASH TABLE: -

**CODE:-**

class BSTNode:

    def \_\_init\_\_(self, song):

        self.song = song

        self.left = None

        self.right = None

class BST:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, song):

        if self.root is None:

            self.root = BSTNode(song)

        else:

            current = self.root

            while True:

                if song.title < current.song.title:

                    if current.left is None:

                        current.left = BSTNode(song)

                        break

                    else:

                        current = current.left

                else:

                    if current.right is None:

                        current.right = BSTNode(song)

                        break

                    else:

                        current = current.right

    def search(self, title):

        current = self.root

        while current is not None:

            if title == current.song.title:

                return current.song

            elif title < current.song.title:

                current = current.left

            else:

                current = current.right

        return None

    def inorder\_traversal(self, current):

        if current is not None:

            self.inorder\_traversal(current.left)

            print(current.song)

            self.inorder\_traversal(current.right)

    def remove\_song(self, title):

        self.root = self.\_delete\_recursive(title, self.root)

    def \_delete\_recursive(self, title, current):

        if current is None:

            return current

        if title < current.song.title:

            current.left = self.\_delete\_recursive(title, current.left)

        elif title > current.song.title:

            current.right = self.\_delete\_recursive(title, current.right)

        else:

            if current.left is None:

                return current.right

            elif current.right is None:

                return current.left

            temp = self.\_find\_min(current.right)

            current.song = temp.song

            current.right = self.\_delete\_recursive(temp.song.title, current.right)

        return current

    def \_find\_min(self, current):

        while current.left is not None:

            current = current.left

        return current

class HashTable:

    def \_\_init\_\_(self):

        self.size = 100

        self.table = [None] \* self.size

    def \_hash\_function(self, song\_id):

        return song\_id % self.size

    def insert(self, song):

        key = self.\_hash\_function(song.id)

        if self.table[key] is None:

            self.table[key] = [song]

        else:

            self.table[key].append(song)

    def search(self, song\_id):

        key = self.\_hash\_function(song\_id)

        if self.table[key] is not None:

            for song in self.table[key]:

                if song.id == song\_id:

                    return song

        return None

class Song:

    def \_\_init\_\_(self, id, title, artist, duration, genre):

        self.id = id

        self.title = title

        self.artist = artist

        self.duration = duration

        self.genre = genre

    def \_\_str\_\_(self):

        return f"ID: {self.id}, Title: {self.title}, Artist: {self.artist}, Duration: {self.duration}, Genre: {self.genre}"

class PlaylistManager:

    def \_\_init\_\_(self):

        self.bst = BST()

        self.hash\_table = HashTable()

    def add\_song(self, song):

        self.bst.insert(song)

        self.hash\_table.insert(song)

    def remove\_song(self, title):

        song = self.bst.search(title)

        if song:

            del self.hash\_table.table[self.hash\_table.\_hash\_function(song.id)]

            self.bst.remove\_song(title)

            print("Song removed from the playlist.")

        else:

            print("Song not found in the playlist.")

    def search\_song(self, title):

        return self.bst.search(title)

    def display\_playlist(self):

        self.bst.inorder\_traversal(self.bst.root)

class PM:

    def \_\_init\_\_(self):

        self.playlist\_manager = PlaylistManager()

    def add\_song(self):

        print("Add a Song")

        id = int(input("Enter song ID: "))

        title = input("Enter song title: ")

        artist = input("Enter artist name: ")

        duration = int(input("Enter song duration: "))

        genre = input("Enter song genre: ")

        song = Song(id, title, artist, duration, genre)

        self.playlist\_manager.add\_song(song)

        print("Song added to the playlist.")

    def remove\_song(self):

        print("Remove a Song")

        title = input("Enter song title: ")

        self.playlist\_manager.remove\_song(title)

    def search\_song(self):

        print("Search for a Song")

        title = input("Enter song title: ")

        song = self.playlist\_manager.search\_song(title)

        if song:

            print(song)

        else:

            print("Song not found in the playlist.")

    def display\_playlist(self):

        print("Playlist")

        self.playlist\_manager.display\_playlist()

    def run(self):

        while True:

            print("Menu:")

            print("1. Add a Song")

            print("2. Remove a Song")

            print("3. Search for a Song")

            print("4. Display Playlist")

            print("0. Exit")

            choice = input("Enter your choice: ")

            if choice == "1":

                self.add\_song()

            elif choice == "2":

                self.remove\_song()

            elif choice == "3":

                self.search\_song()

            elif choice == "4":

                self.display\_playlist()

            elif choice == "0":

                break

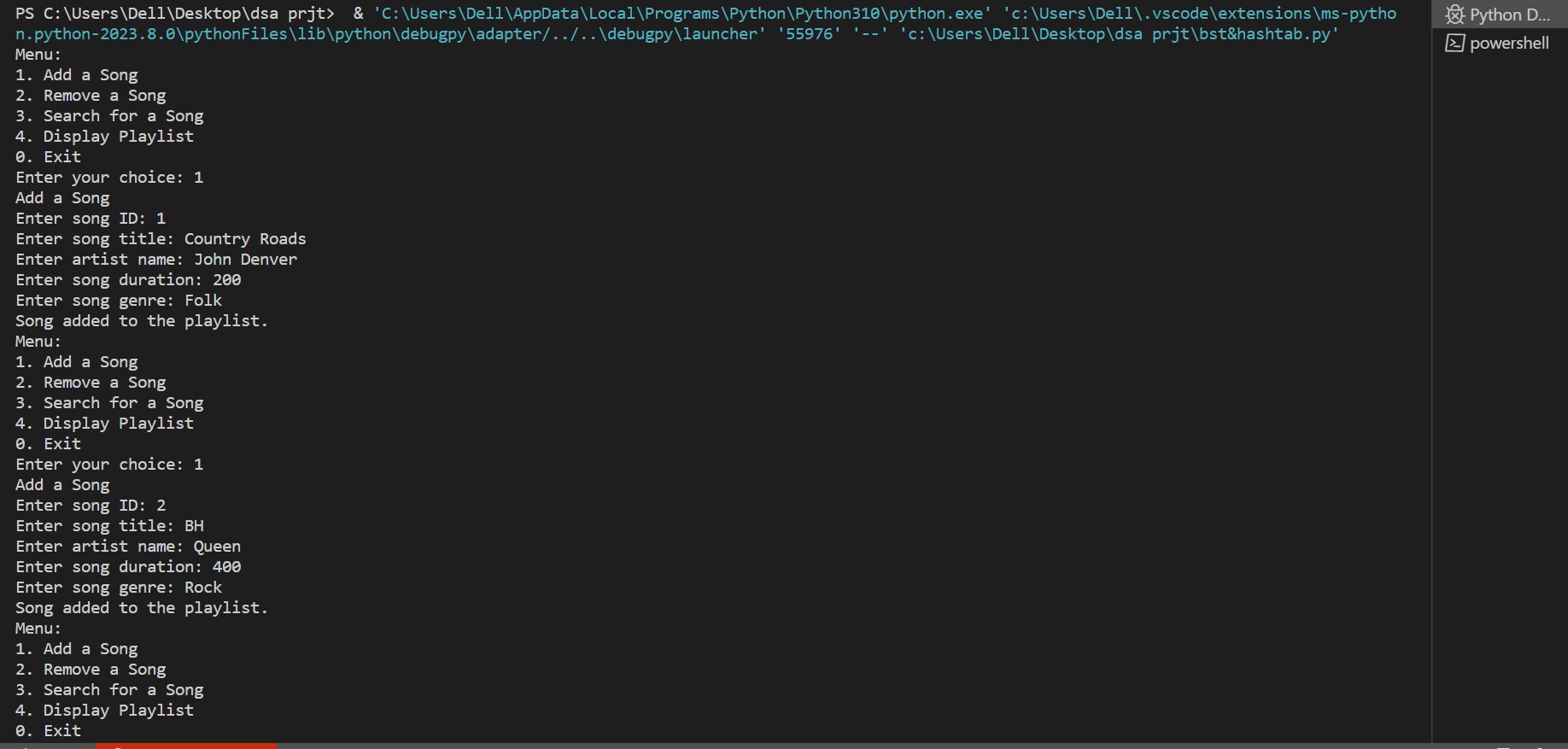
            else:

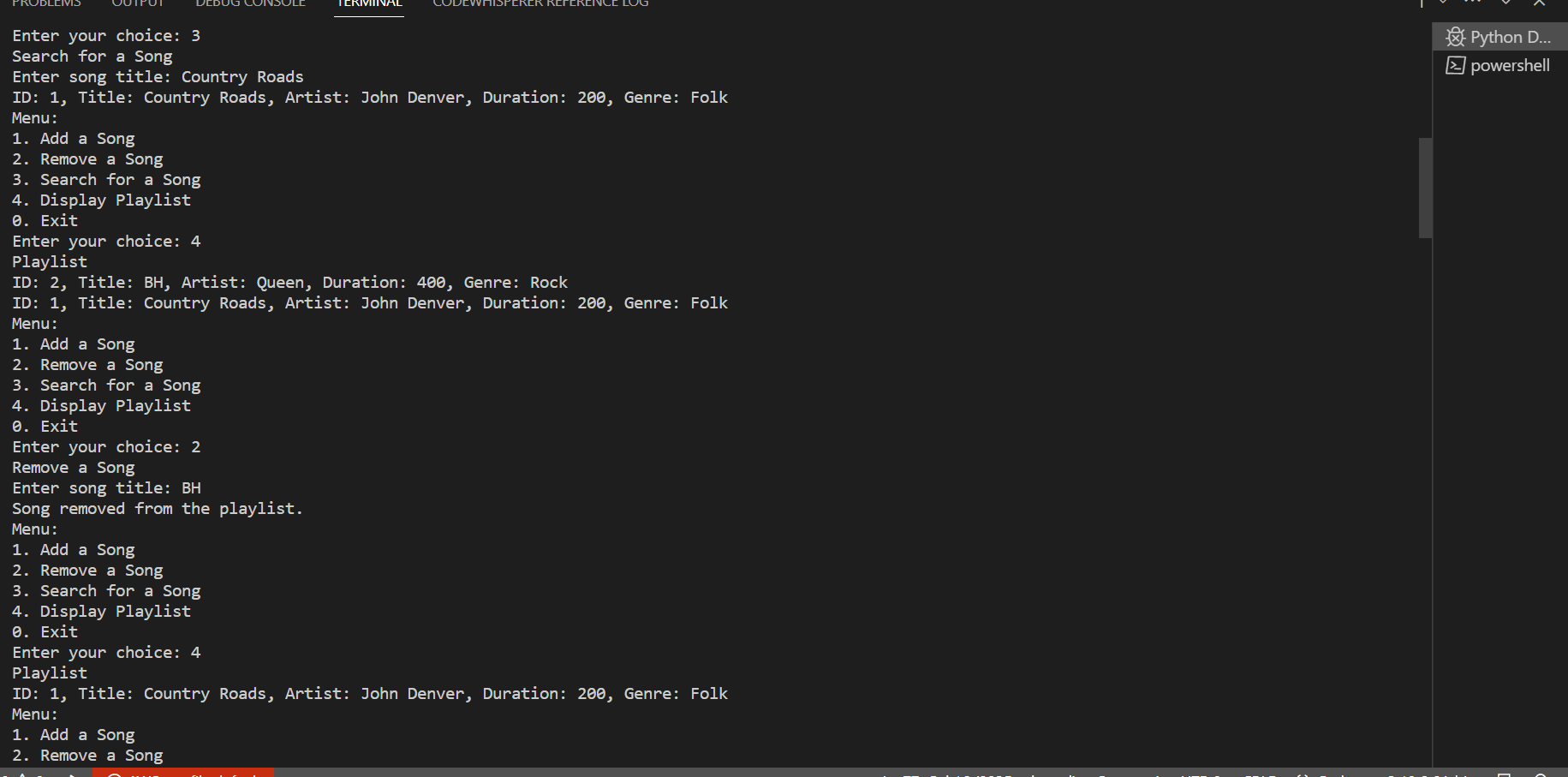
                print("Invalid choice. Please try again.")

ui = PM()

ui.run()

**Output:-**







**Time Complexity:**

**Insertion:**

* In the worst case scenario, where the binary search tree is unbalanced, the time complexity of insertion is O(n).
* In the average case, assuming a balanced binary search tree, the time complexity of insertion is O(log n).

**Search:**

* In the worst case scenario, where the binary search tree is unbalanced, the time complexity of search is O(n).
* In the average case, assuming a balanced binary search tree, the time complexity of search is O(log n).

**Inorder Traversal:**

The time complexity of inorder traversal is O(n).

**Removal:**

* In the worst case scenario, where the binary search tree is unbalanced, the time complexity of removal is O(n).
* In the average case, assuming a balanced binary search tree, the time complexity of removal is O(log n).

**Hash Table Operations:**

* Operations like insertion and search, have an average time complexity of O(1) since they directly access the table using the hashed key.
* However, in the worst case scenario, where collisions occur frequently, the time complexity can be O(n).

**Why Hybrid data can be chosen over existing data structure?**

**Efficient search:**

BST provides efficient search operations with a time complexity of O(log n) on average. This allows quick retrieval of songs based on their titles, artists, or other criteria. In contrast, a linked list requires linear search, resulting in a time complexity of O(n) in the worst case.  
  
**Fast insertions and deletions:**

BST provides efficient insertions and deletions with a time complexity of O(log n) on average. This is advantageous when adding or removing songs from the playlist. In a linked list, insertions and deletions require finding the appropriate position in the list, resulting in a time complexity of O(n) in the worst case.

**Faster updates and modifications:**

Hash Table allows efficient storage and retrieval of additional information about songs (e.g., duration, album, genre) using a unique identifier (e.g., song ID) as the key. This makes it easy to access and update song details.

The hybrid data structure allows for faster updates and modifications to the playlist. When a song's details change (e.g., title or artist), only the BST needs to be updated, while the song's information in the hash table remains intact. This reduces the need for extensive data rearrangement compared to a linked list.

**Why was this specific combination of data structures (Binary Search Tree and Hash Table) chosen over other hybrid data structures for fulfilling the requirements of this playlist manager?**

**Requirements:**

* Efficient search
* Additional information storage for easier modification

Another possible Hybrid Data Structure we can use is **“AVL TREE AND HASH TABLE”.**

But we are not using it for the following reasons:

* Complexity: AVL trees require additional bookkeeping to maintain balance, which can introduce a small overhead in terms of space and time complexity compared to regular BSTs. This can impact the overall performance of operations such as insertion and deletion.
* Implementation complexity: AVL trees are more complex to implement compared to basic BSTs, requiring careful handling of rotations and balance factors.

https://bootcamp.uxdesign.cc/build-a-playlist-with-linked-lists-and-javascript-63472a80ed6f