

Musik-App: Verwaltung und Sortierung von Songs

Übersicht

Dieses Projekt implementiert eine Musik-App, die es ermöglicht, Songs zu verwalten, zu sortieren und zu durchsuchen. Die App unterstützt verschiedene Sortier- und Suchalgorithmen und verwendet einen Rot-Schwarz-Baum zur effizienten Verwaltung der Songs.

Installation

1. **Python installieren**: Stelle sicher, dass Python 3.x auf deinem System installiert ist. Du kannst Python von python.org herunterladen und installieren.
2. **Abhängigkeiten installieren**: Installiere die benötigten Python-Bibliotheken mit dem folgenden Befehl:

```
```bash  

pip install -r requirements.txt

```
```

Hauptmenü

Nach dem Starten des Skripts wird das Hauptmenü der Musik-App angezeigt. Du kannst eine der folgenden Optionen auswählen:

- Neuen Song hinzufügen
- Playlist erstellen
- Song zur Playlist hinzufügen
- Song aus Playlist entfernen
- Songs suchen

- Songs sortieren
- Alle Songs anzeigen
- Playlists anzeigen
- Song aus Bibliothek löschen
- Beenden
- Zufällige Songs erstellen

Vorgehensweise

Entwerfen der Architektur

Der erste Schritt bei der Entwicklung der Musik-App bestand darin, eine klare und skalierbare Architektur zu entwerfen. Dies umfasste die Definition der Hauptkomponenten der App, einschließlich der Klassen ``Song``, ``Playlist``, ``RedBlackNode``, ``RedBlackTree`` und ``MusicApp``. Jede Klasse wurde so konzipiert, dass sie spezifische Funktionen und Verantwortlichkeiten hat, um eine modulare und wartbare Codebasis zu gewährleisten.

Implementierung

Nach dem Entwurf der Architektur begann die Implementierung der einzelnen Komponenten. Die ``Song``-Klasse wurde erstellt, um die Attribute und Methoden eines Songs zu definieren. Die ``Playlist``-Klasse verwaltet eine Sammlung von Songs, während die ``RedBlackTree``-Klasse einen effizienten Such- und Sortiermechanismus bietet. Die ``MusicApp``-Klasse integriert alle Komponenten und implementiert die Hauptlogik der App, einschließlich der Benutzerinteraktionen und der Verwaltung der Songs und Playlists.

Validierung

Nach der Implementierung wurde jede Komponente gründlich validiert, um sicherzustellen, dass sie korrekt funktioniert. Dies umfasste das Testen der Methoden

der `Song` - und `Playlist` -Klassen sowie die Überprüfung der Rot-Schwarz-Baum-Operationen in der `RedBlackTree` -Klasse. Die Validierung half dabei, Fehler frühzeitig zu erkennen und zu beheben, bevor die Komponenten in die Hauptanwendung integriert wurden.

Testen und Vergleichen der Algorithmen

Ein wesentlicher Teil der Entwicklung bestand darin, die verschiedenen Sortier- und Suchalgorithmen zu testen und zu vergleichen. Die Algorithmen wurden implementiert und auf ihre Leistung hin überprüft. Dies umfasste die Messung der Ausführungszeit und des Speicherverbrauchs für verschiedene Datensätze. Die Ergebnisse wurden analysiert, um die effizientesten Algorithmen für die spezifischen Anforderungen der Musik-App auszuwählen.

Dokumentation

Abschließend wurde eine umfassende Dokumentation erstellt, die alle Aspekte der Entwicklung der Musik-App abdeckt. Dies umfasst die Beschreibung der Architektur, die Implementierungsdetails, die Validierungsschritte sowie die Testergebnisse der Algorithmen. Die Dokumentation dient als Referenz für zukünftige Entwickler und stellt sicher, dass die App leicht verständlich und erweiterbar ist.

Klassen und Methoden

Dateistruktur der Musik-App

```plaintext

music-app/

├─ main.py

├─ song.py

├─ playlist.py

```
|— red_black_tree.py
|— music_app.py
...

```

- **main.py**: Startet die Musik-App und zeigt das Hauptmenü an.
- **song.py**: Enthält die `Song`-Klasse, die einen Song repräsentiert.
- **playlist.py**: Enthält die `Playlist`-Klasse, die eine Playlist verwaltet.
- **red\_black\_tree.py**: Enthält die `RedBlackNode`- und `RedBlackTree`-Klassen zur Verwaltung von Songs in einem Rot-Schwarz-Baum.
- **music\_app.py**: Implementiert die Hauptlogik der Musik-App und verwaltet die Interaktionen zwischen den verschiedenen Komponenten.

### Klasse `Song` (Datei: `song.py`)

Die `Song`-Klasse repräsentiert einen Song mit den Attributen Titel, Künstler, Album und Genre. Sie enthält folgende Methoden:

**Methoden:**

- `__init__(self, title, artist, album, genre)`: Initialisiert ein Song-Objekt.
- `__str__(self)`: Gibt eine lesbare Darstellung des Songs zurück.
- `__lt__(self, other)`: Vergleichsoperator für weniger als, basierend auf dem Titel.
- `__eq__(self, other)`: Vergleichsoperator für Gleichheit, basierend auf dem Titel.
- `to_dict(self)`: Konvertiert das Song-Objekt in ein Wörterbuch.
- `from_dict(data)`: Erstellt ein Song-Objekt aus einem Wörterbuch.

### Klasse `Playlist` (Datei: `playlist.py`)

Die ``Playlist``-Klasse verwaltet eine Sammlung von Songs und enthält folgende Methoden:

**\*\*Methoden:\*\***

- ``__init__(self, name)`` : Initialisiert eine Playlist mit einem Namen und einer leeren Songliste.
- ``add_song(self, song)`` : Fügt einen Song zur Playlist hinzu.
- ``remove_song(self, title)`` : Entfernt einen Song aus der Playlist basierend auf dem Titel.
- ``__str__(self)`` : Gibt eine lesbare Darstellung der Playlist zurück.
- ``to_dict(self)`` : Konvertiert die Playlist in ein Wörterbuch.
- ``from_dict(data)`` : Erstellt eine Playlist aus einem Wörterbuch.

### Klasse ``RedBlackNode`` (Datei: ``red_black_tree.py``)

**\*\*Beschreibung:\*\*** Repräsentiert einen Knoten in einem Rot-Schwarz-Baum.

**\*\*Methoden:\*\***

- ``__init__(self, song)`` : Initialisiert einen Knoten für den Rot-Schwarz-Baum.

### `RedBlackNode`- und `RedBlackTree`-Klassen (Datei: ``red_black_tree.py``)

Die ``RedBlackNode``-Klasse repräsentiert einen Knoten in einem Rot-Schwarz-Baum. Die ``RedBlackTree``-Klasse implementiert einen Rot-Schwarz-Baum zur effizienten Verwaltung von Songs und enthält folgende Methoden:

- ``__init__(self)`` : Initialisiert einen Rot-Schwarz-Baum mit einem NIL-Knoten.
- ``insert(self, song)`` : Fügt einen neuen Song in den Rot-Schwarz-Baum ein.

- ``fix_insert(self, node)`` : Fixiert den Baum nach dem Einfügen, um die Rot-Schwarz-Eigenschaften zu bewahren.
- ``left_rotate(self, x)`` : Führt eine Linksrotation durch.
- ``right_rotate(self, x)`` : Führt eine Rechtsrotation durch.
- ``get_size(self)`` : Berechnet die Größe des Baums.
- ``bfs_search(self, value, criteria)`` : Führt eine Breitensuche nach einem Song basierend auf einem Kriterium durch.
- ``dfs_search(self, node, value, criteria)`` : Führt eine Tiefensuche nach einem Song basierend auf einem Kriterium durch.
- ``search(self, value, criteria)`` : Führt eine rekursive Suche nach einem Song basierend auf einem Kriterium durch.
- ``delete(self, song)`` : Löscht einen Song aus dem Rot-Schwarz-Baum.
- ``_delete_recursive(self, node, song)`` : Rekursive Methode zum Löschen eines Songs.
- ``_min_value_node(self, node)`` : Findet den Knoten mit dem minimalen Wert.

### Klasse ``MusicApp`` (Datei: ``music_app.py``)

Die ``MusicApp``-Klasse implementiert die Hauptlogik der Musik-App und verwaltet die Interaktionen zwischen den verschiedenen Komponenten. Sie enthält folgende Methoden:

**\*\*Methoden:\*\***

- ``__init__(self)`` : Initialisiert die Musik-App und lädt Songs.
- ``load_songs(self)`` : Lädt Songs aus einer Datei.
- ``save_data(self)`` : Speichert alle Songs in einer Datei.
- ``verify_saved_data(self)`` : Überprüft die gespeicherten Daten.
- ``add_song(self, title, artist, album, genre)`` : Fügt einen neuen Song zur Bibliothek hinzu und speichert die Daten.

- ``delete_song(self, title)`` : Löscht einen Song aus der Bibliothek und speichert die Daten.
- ``display_all_songs(self)`` : Zeigt alle Songs in der Bibliothek an.
- ``measure_memory_and_time(self, method, *args)`` : Misst die Zeit und den Speicherverbrauch einer Methode.
- ``search_song(self)`` : Sucht nach einem Song basierend auf einem Kriterium und einer Suchmethode.
- ``linear_search(self, value, criteria)`` : Sucht linear nach einem Song-Objekt in der Liste basierend auf einem Kriterium.
- ``binary_search(self, value, criteria)`` : Führt eine Binärsuche nach einem Song-Objekt in der sortierten Liste basierend auf einem Kriterium durch.
- ``jump_search(self, value, criteria)`` : Führt eine Jump-Suche nach einem Song-Objekt in der sortierten Liste basierend auf einem Kriterium durch.
- ``interpolation_search(self, value, criteria)`` : Führt eine Interpolationssuche nach einem Song-Objekt in der sortierten Liste basierend auf einem Kriterium durch.
- ``search_all_methods(self, value, criteria)`` : Führt alle Suchmethoden nacheinander aus und gibt die Ergebnisse aus.
- ``measure_memory_and_time_sort(self, sort_method, *args)`` : Misst die Zeit und den Speicherverbrauch eines Sortieralgorithmus.
- ``print_songs(self, message)`` : Gibt eine Nachricht und die Liste der Songs aus.
- ``sort_songs(self)`` : Sortiert die Songs basierend auf einem ausgewählten Algorithmus und Kriterium.
- ``get_sort_order_and_criteria(self, method_name)`` : Fragt die Sortierreihenfolge und das Sortierkriterium ab.
- ``bubble_sort(self, order, criteria)`` : Implementiert den Bubble Sort Algorithmus.
- ``insertion_sort(self, order, criteria)`` : Implementiert den Insertion Sort Algorithmus.
- ``merge_sort(self, order, criteria)`` : Implementiert den Merge Sort Algorithmus.
- ``_merge_sort(self, array, order, criteria)`` : Hilfsmethode für Merge Sort.
- ``merge(self, left, right, order, criteria)`` : Führt das Mergen von zwei Listenhälften durch.
- ``quick_sort(self, low, high, order, criteria)`` : Implementiert den Quick Sort Algorithmus.

- ``partition(self, low, high, order, criteria)`` : Hilfsmethode für Quick Sort.
- ``compare(self, song1, song2, ascending, criteria)`` : Vergleicht zwei Song-Objekte basierend auf einem Kriterium und der Sortierreihenfolge.
- ``create_playlist(self)`` : Erstellt eine neue Playlist.
- ``add_song_to_playlist(self)`` : Fügt einen Song einer Playlist hinzu.
- ``remove_song_from_playlist(self)`` : Entfernt einen Song aus einer Playlist.
- ``display_playlists(self)`` : Zeigt alle Playlists an.
- ``create_random_songs(self, count)`` : Erstellt eine bestimmte Anzahl zufälliger Songs und speichert sie.
- ``main_menu(self)`` : Hauptmenü der Musik-App.

## ## Implementierungsdetails

### ### Datenstrukturen

Die Musik-App verwendet verschiedene Datenstrukturen, um die Songs und Playlists effizient zu verwalten. Listen werden verwendet, um Songs und Playlists zu speichern, während Rot-Schwarz-Bäume für die effiziente Verwaltung und Suche von Songs eingesetzt werden.

### ### Algorithmen

#### #### Sortieralgorithmen

Die Musik-App implementiert mehrere Sortieralgorithmen, darunter Bubble Sort, Insertion Sort, Merge Sort und Quick Sort. Diese Algorithmen werden verwendet, um die Songs basierend auf verschiedenen Kriterien wie Titel, Künstler, Album oder Genre zu sortieren.

#### #### Suchalgorithmen



Die Musik-App verwendet verschiedene Suchalgorithmen, um Songs effizient zu finden. Dazu gehören lineare Suche, Binärsuche, Jump-Suche, Interpolationssuche, BFS und DFS. Diese Algorithmen ermöglichen es, Songs basierend auf verschiedenen Kriterien wie Titel, Künstler, Album oder Genre zu durchsuchen.

### ### Benutzerinteraktionen

Die Benutzeroberfläche der Musik-App ermöglicht es den Benutzern, verschiedene Aktionen durchzuführen, wie das Hinzufügen und Löschen von Songs, das Erstellen und Verwalten von Playlists und das Durchsuchen der Bibliothek. Diese Interaktionen werden durch die Methoden der `MusicApp`-Klasse verwaltet.

### ### Dateiverwaltung

Die Musik-App speichert und lädt Songs und Playlists in und aus Dateien. Die Methoden `load\_songs` und `save\_data` der `MusicApp`-Klasse sind für diese Aufgaben verantwortlich.

### ### Fehlerbehandlung

Die Musik-App enthält Mechanismen zur Fehlerbehandlung und Validierung von Benutzereingaben, um die Robustheit der Anwendung zu gewährleisten. Dies umfasst die Überprüfung der Eingaben und das Abfangen von Ausnahmen.

### ### Leistungsmessung




Die Musik-App misst die Ausführungszeit und den Speicherverbrauch der verschiedenen Algorithmen, um deren Effizienz zu bewerten. Die Methoden `measure\_memory\_and\_time` und `measure\_memory\_and\_time\_sort` der `MusicApp`-Klasse werden für diese Messungen verwendet.

## # Leistungsanalyse des Rot-Schwarz-Baums

### ### Einfügeoperationen

- **Zeitkomplexität**: Die durchschnittliche und schlimmste Zeitkomplexität für das Einfügen eines Songs in den Rot-Schwarz-Baum beträgt  $O(\log n)$ , wobei  $n$  die Anzahl der Songs im Baum ist.
- **Speicherkomplexität**: Der Speicherverbrauch für das Einfügen eines Songs ist konstant  $O(1)$ , da nur ein neuer Knoten hinzugefügt wird.

### ### Beispielmessungen

- **10.000 Songs**:
  -  (Bilder/Red\_Black\_Tree/RBT\_10000/RBT\_10000\_Songs.png)
  - **Benötigte Zeit**: 0.090645 Sekunden
  - **Speicherkapazität**: 960000 Bytes
- **20.000 Songs**:
  -  (Bilder/Red\_Black\_Tree/RBT\_20000/RBT\_20000\_Songs.png)
  - **Benötigte Zeit**: 0.166783 Sekunden
  - **Speicherkapazität**: 480168 Bytes
- **1.000.000 Songs**:
  -  (Bilder/Red\_Black\_Tree/RBT\_1000000/RBT\_1000000\_Songs.png)
  - **Benötigte Zeit**: 21.866240 Sekunden
  - **Speicherkapazität**: 192000000 Bytes

### ### Funktionsweise

- **Einfügen**: Beim Einfügen eines Songs wird der Rot-Schwarz-Baum so angepasst, dass die Rot-Schwarz-Eigenschaften erhalten bleiben. Dies kann Rotationen und Farbänderungen erfordern, um die Balance des Baums zu gewährleisten.

### ### Begründung für Laufzeit und Speicherverbrauch



- **Laufzeit**: Die logarithmische Zeitkomplexität resultiert aus der Höhe des Baums, die im schlimmsten Fall  $O(\log n)$  beträgt. Dies ermöglicht effiziente Einfügeoperationen auch bei großen Datenmengen.
- **Speicherverbrauch**: Der konstante Speicherverbrauch ist auf die Tatsache zurückzuführen, dass nur ein neuer Knoten hinzugefügt wird, ohne dass zusätzliche Datenstrukturen benötigt werden.

## # Sortialgorithmen: Laufzeit- und Speicheranalyse

### ## Übersicht über die Sortialgorithmen

Die folgenden Sortialgorithmen wurden auf Listen von 10.000 und 20.000 Songs angewendet.

#### ### Beispielmessungen

- **10.000 Songs**:
  -  (Bilder/Red\_Black\_Tree/RBT\_10000/RBT\_10000\_Sort.png)
- **20.000 Songs**:
  -  (Bilder/Red\_Black\_Tree/RBT\_20000/RBT\_20000\_Sort.png)

#### ### Bubble Sort

- **10.000 Songs**:
  - **Benötigte Zeit**: 71.401654 Sekunden
  - **Verwendete Speicherkapazität**: 470 Bytes
- **20.000 Songs**:
  - **Benötigte Zeit**: 368.16585 Sekunden
  - **Verwendete Speicherkapazität**: 4740 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**:  $O(n^2)$

- **Speicherkomplexität**:  $O(1)$
- **Funktionsweise**:
  - Bubble Sort ist ein einfacher Vergleichs-basierter Algorithmus. Er wiederholt sich durch die Liste und vergleicht benachbarte Elemente. Wenn zwei benachbarte Elemente in der falschen Reihenfolge sind, werden sie vertauscht. Dieser Vorgang wird wiederholt, bis die Liste vollständig sortiert ist. Der Algorithmus durchläuft die Liste mehrfach, wobei jedes Mal das größte verbleibende Element an seine endgültige Position "aufsteigt".
- **Begründung für Laufzeit und Speicherverbrauch**:
  - **Laufzeit**: Die lange Laufzeit von Bubble Sort resultiert aus der quadratischen Zeitkomplexität  $O(n^2)$ . Bei jeder Iteration werden alle Paare von benachbarten Elementen verglichen und möglicherweise vertauscht, was bei großen Datenmengen sehr ineffizient ist.
  - **Speicherverbrauch**: Der Speicherverbrauch ist sehr gering ( $O(1)$ ), da Bubble Sort in-place arbeitet und keine zusätzlichen Datenstrukturen benötigt.

### ### Insertion Sort

- **10.000 Songs**:
  - **Benötigte Zeit**: 30.481734 Sekunden
  - **Verwendete Speicherkapazität**: 452 Bytes
- **20.000 Songs**:
  - **Benötigte Zeit**: 145.92921 Sekunden
  - **Verwendete Speicherkapazität**: 452 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**:  $O(n^2)$
  - **Speicherkomplexität**:  $O(1)$
- **Funktionsweise**:
  - Insertion Sort sortiert die Liste, indem es ein Element nach dem anderen betrachtet und es an der richtigen Stelle in die bereits sortierte Teilliste einfügt. Der Algorithmus beginnt mit dem zweiten Element und vergleicht es mit den vorherigen Elementen, verschiebt diese gegebenenfalls nach rechts und fügt das aktuelle Element an der richtigen Stelle ein. Dieser Vorgang wird für jedes Element in der Liste wiederholt.

- **Begründung für Laufzeit und Speicherverbrauch**:

- **Laufzeit**: Die Laufzeit von Insertion Sort ist ebenfalls quadratisch ( $O(n^2)$ ), aber in der Praxis oft schneller als Bubble Sort, da weniger Vergleiche und Vertauschungen erforderlich sind, insbesondere wenn die Liste bereits teilweise sortiert ist.

- **Speicherverbrauch**: Der Speicherverbrauch ist sehr gering ( $O(1)$ ), da Insertion Sort in-place arbeitet und keine zusätzlichen Datenstrukturen benötigt.

### ### Merge Sort

- **10.000 Songs**:

- **Benötigte Zeit**: 0.167851 Sekunden

- **Verwendete Speicherkapazität**: 17048 Bytes

- **20.000 Songs**:

- **Benötigte Zeit**: 0.33358 Sekunden

- **Verwendete Speicherkapazität**: 394476 Bytes

- **Big-O-Notation**:

- **Zeitkomplexität**:  $O(n \log n)$

- **Speicherkomplexität**:  $O(n)$

- **Funktionsweise**:

- Merge Sort ist ein Divide-and-Conquer-Algorithmus. Er teilt die Liste wiederholt in zwei Hälften, bis jede Teilliste nur noch ein Element enthält. Dann werden die Teillisten paarweise zusammengeführt (gemergt), wobei die Elemente in der richtigen Reihenfolge angeordnet werden. Dieser Vorgang des Teilens und Mergens wird rekursiv fortgesetzt, bis die gesamte Liste sortiert ist.

- **Begründung für Laufzeit und Speicherverbrauch**:

- **Laufzeit**: Merge Sort hat eine effiziente Zeitkomplexität von  $O(n \log n)$ , da die Liste wiederholt geteilt und die Teillisten effizient zusammengeführt werden. Dies führt zu einer deutlich schnelleren Laufzeit im Vergleich zu Bubble Sort und Insertion Sort.

- **Speicherverbrauch**: Der Speicherverbrauch ist höher ( $O(n)$ ), da Merge Sort zusätzliche Arrays für die Teillisten benötigt, was zu einem höheren Speicherbedarf führt.

### ### Quick Sort

- **10.000 Songs**:
  - **Benötigte Zeit**: 0.285078 Sekunden
  - **Verwendete Speicherkapazität**: 424 Bytes
- **20.000 Songs**:
  - **Benötigte Zeit**: 0.43289 Sekunden
  - **Verwendete Speicherkapazität**: 2088 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**:
    - Durchschnittlich:  $O(n \log n)$
    - Schlimmster Fall:  $O(n^2)$
  - **Speicherkomplexität**:  $O(\log n)$
- **Funktionsweise**:
  - Quick Sort ist ebenfalls ein Divide-and-Conquer-Algorithmus. Er wählt ein "Pivot"-Element aus der Liste aus und partitioniert die Liste so, dass alle Elemente, die kleiner als das Pivot sind, links davon und alle größeren rechts davon stehen. Danach wird der Algorithmus rekursiv auf die Teillisten links und rechts vom Pivot angewendet. Dieser Vorgang wird fortgesetzt, bis die gesamte Liste sortiert ist.
  - **Begründung für Laufzeit und Speicherverbrauch**:
    - **Laufzeit**: Quick Sort hat im Durchschnitt eine sehr effiziente Zeitkomplexität von  $O(n \log n)$ , da die Liste effizient partitioniert wird. Im schlimmsten Fall kann die Laufzeit jedoch quadratisch ( $O(n^2)$ ) sein, wenn das Pivot-Element ungünstig gewählt wird. In der Praxis ist Quick Sort jedoch oft sehr schnell.
    - **Speicherverbrauch**: Der Speicherverbrauch ist gering ( $O(\log n)$ ), da Quick Sort in-place arbeitet und nur zusätzlichen Speicher für die Rekursionsstapel benötigt.

### ## Übersicht über alle Sortieralgorithmen

#### ### 10.000 Songs

- **Bubble Sort**: Benötigte Zeit: 71.401654 Sekunden. Verwendete Speicherkapazität: 470 Bytes.
- **Insertion Sort**: Benötigte Zeit: 30.481734 Sekunden. Verwendete Speicherkapazität: 452 Bytes.
- **Merge Sort**: Benötigte Zeit: 0.167851 Sekunden. Verwendete Speicherkapazität: 17048 Bytes.
- **Quick Sort**: Benötigte Zeit: 0.285078 Sekunden. Verwendete Speicherkapazität: 424 Bytes.

### ### 20.000 Songs

- **Bubble Sort**: Benötigte Zeit: 368.16585 Sekunden. Verwendete Speicherkapazität: 4740 Bytes.
- **Insertion Sort**: Benötigte Zeit: 145.92921 Sekunden. Verwendete Speicherkapazität: 452 Bytes.
- **Merge Sort**: Benötigte Zeit: 0.33358 Sekunden. Verwendete Speicherkapazität: 394476 Bytes.
- **Quick Sort**: Benötigte Zeit: 0.43289 Sekunden. Verwendete Speicherkapazität: 2088 Bytes.

## ## Suchalgorithmen: Laufzeit- und Speicheranalyse

Die folgenden Suchalgorithmen wurden auf eine Liste von 1.000.000 Songs angewendet.

### ### Beispielmessungen

- **10.000 Songs**:
  -  [RBT 1mio\_search](Bilder/Red\_Black\_Tree/RBT\_1000000/RBT\_1000000\_Search.png)

### ### Lineare Suche

- **Benötigte Zeit**: 0.294438 Sekunden

- **Verwendete Speicherkapazität**: 256 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**:  $O(n)$
  - **Speicherkomplexität**:  $O(1)$
- **Funktionsweise**:
  - Die lineare Suche durchsucht die Liste sequenziell von Anfang bis Ende. Jedes Element wird mit dem gesuchten Wert verglichen, bis entweder ein Treffer gefunden wird oder das Ende der Liste erreicht ist.
- **Begründung für Laufzeit und Speicherverbrauch**:
  - **Laufzeit**: Die lineare Suche hat eine Zeitkomplexität von  $O(n)$ , da im schlimmsten Fall jedes Element der Liste durchsucht werden muss.
  - **Speicherverbrauch**: Der Speicherverbrauch ist sehr gering ( $O(1)$ ), da keine zusätzlichen Datenstrukturen benötigt werden.

### ### Binärsuche

- **Benötigte Zeit**: 0.000000 Sekunden
- **Verwendete Speicherkapazität**: 160 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**:  $O(\log n)$
  - **Speicherkomplexität**:  $O(1)$
- **Funktionsweise**:
  - Die Binärsuche funktioniert auf einer sortierten Liste. Sie teilt die Liste wiederholt in zwei Hälften und vergleicht den mittleren Wert mit dem gesuchten Wert. Je nach Ergebnis wird die Suche auf die linke oder rechte Hälfte der Liste beschränkt, bis der Wert gefunden wird oder die Liste erschöpft ist.
- **Begründung für Laufzeit und Speicherverbrauch**:
  - **Laufzeit**: Die Binärsuche hat eine sehr effiziente Zeitkomplexität von  $O(\log n)$ , da die Liste bei jedem Schritt halbiert wird.
  - **Speicherverbrauch**: Der Speicherverbrauch ist sehr gering ( $O(1)$ ), da keine zusätzlichen Datenstrukturen benötigt werden.



### ### Jump-Suche

- **Benötigte Zeit**: 0.000000 Sekunden
- **Verwendete Speicherkapazität**: 228 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**:  $O(\sqrt{n})$
  - **Speicherkomplexität**:  $O(1)$
- **Funktionsweise**:
  - Die Jump-Suche funktioniert auf einer sortierten Liste. Sie springt in festen Schritten durch die Liste, um einen Bereich zu finden, der den gesuchten Wert enthalten könnte. Innerhalb dieses Bereichs wird dann eine lineare Suche durchgeführt.
- **Begründung für Laufzeit und Speicherverbrauch**:
  - **Laufzeit**: Die Jump-Suche hat eine Zeitkomplexität von  $O(\sqrt{n})$ , da die Liste in Schritten der Größe  $\sqrt{n}$  durchsucht wird.
  - **Speicherverbrauch**: Der Speicherverbrauch ist sehr gering ( $O(1)$ ), da keine zusätzlichen Datenstrukturen benötigt werden.

### ### Interpolationssuche

- **Benötigte Zeit**: 0.000000 Sekunden
- **Verwendete Speicherkapazität**: 160 Bytes
- **Big-O-Notation**:
  - **Zeitkomplexität**: Durchschnittlich  $O(\log \log n)$ , schlimmster Fall  $O(n)$
  - **Speicherkomplexität**:  $O(1)$
- **Funktionsweise**:
  - Die Interpolationssuche funktioniert auf einer sortierten Liste. Sie schätzt die Position des gesuchten Wertes basierend auf dem Verhältnis der Differenzen zwischen den Werten und passt die Suchposition entsprechend an. Dieser Vorgang wird wiederholt, bis der Wert gefunden wird oder die Liste erschöpft ist.
- **Begründung für Laufzeit und Speicherverbrauch**:
  - **Laufzeit**: Die Interpolationssuche hat im Durchschnitt eine sehr effiziente Zeitkomplexität von  $O(\log \log n)$ , kann aber im schlimmsten Fall  $O(n)$  erreichen, wenn die Werte ungleichmäßig verteilt sind.

- **Speicherverbrauch**: Der Speicherverbrauch ist sehr gering ( $O(1)$ ), da keine zusätzlichen Datenstrukturen benötigt werden.

### ### Breitensuche (Breadth-first Search)

- **Benötigte Zeit**: 76.208232 Sekunden

- **Verwendete Speicherkapazität**: 2312416 Bytes

- **Big-O-Notation**:

- **Zeitkomplexität**:  $O(V + E)$ , wobei  $V$  die Anzahl der Knoten und  $E$  die Anzahl der Kanten ist

- **Speicherkomplexität**:  $O(V)$

- **Funktionsweise**:

- Die Breitensuche durchsucht einen Graphen Ebene für Ebene. Sie beginnt bei einem Startknoten und besucht alle Nachbarn dieses Knotens, bevor sie zu den Nachbarn der Nachbarn übergeht. Dieser Vorgang wird fortgesetzt, bis der gesuchte Wert gefunden wird oder alle Knoten durchsucht sind.

- **Begründung für Laufzeit und Speicherverbrauch**:

- **Laufzeit**: Die Breitensuche hat eine Zeitkomplexität von  $O(V + E)$ , da jeder Knoten und jede Kante im Graphen einmal besucht wird.

- **Speicherverbrauch**: Der Speicherverbrauch kann hoch sein ( $O(V)$ ), da alle Knoten in einer Warteschlange gespeichert werden müssen.

### ### Tiefensuche (Depth-first Search)

- **Benötigte Zeit**: 0.409650 Sekunden

- **Verwendete Speicherkapazität**: 24 Bytes

- **Big-O-Notation**:

- **Zeitkomplexität**:  $O(V + E)$ , wobei  $V$  die Anzahl der Knoten und  $E$  die Anzahl der Kanten ist

- **Speicherkomplexität**:  $O(V)$

- **Funktionsweise**:

- Die Tiefensuche durchsucht einen Graphen, indem sie so tief wie möglich in einen Zweig des Graphen vordringt, bevor sie zurückkehrt und andere Zweige durchsucht.

Dieser Vorgang wird rekursiv fortgesetzt, bis der gesuchte Wert gefunden wird oder alle Knoten durchsucht sind.

- **Begründung für Laufzeit und Speicherverbrauch**:

- **Laufzeit**: Die Tiefensuche hat eine Zeitkomplexität von  $O(V + E)$ , da jeder Knoten und jede Kante im Graphen einmal besucht wird.

- **Speicherverbrauch**: Der Speicherverbrauch ist in der Regel geringer als bei der Breitensuche, da nur der aktuelle Pfad im Speicher gehalten werden muss.

## ## Übersicht über alle Suchalgorithmen

- **Lineare Suche**: Benötigte Zeit: 0.294438 Sekunden. Verwendete Speicherkapazität: 256 Bytes.

- **Binärsuche**: Benötigte Zeit: 0.000000 Sekunden. Verwendete Speicherkapazität: 160 Bytes.

- **Jump-Suche**: Benötigte Zeit: 0.000000 Sekunden. Verwendete Speicherkapazität: 228 Bytes.

- **Interpolationssuche**: Benötigte Zeit: 0.000000 Sekunden. Verwendete Speicherkapazität: 160 Bytes.

- **Breitensuche**: Benötigte Zeit: 76.208232 Sekunden. Verwendete Speicherkapazität: 2312416 Bytes.

- **Tiefensuche**: Benötigte Zeit: 0.409650 Sekunden. Verwendete Speicherkapazität: 24 Bytes.

## ## Vergleich und Zusammenfassung der Sortier- und Suchalgorithmen

In der Musik-App werden verschiedene Sortier- und Suchalgorithmen verwendet, um die Songs effizient zu verwalten. Bei den Sortieralgorithmen zeigt sich, dass der **Bubble Sort** zwar einfach zu implementieren ist, aber aufgrund seiner Zeitkomplexität von  $O(n^2)$  für große Datensätze ineffizient ist. Im Gegensatz dazu bietet der **Merge Sort** eine deutlich bessere Leistung mit einer Zeitkomplexität von  $O(n \log n)$ , da er die Liste rekursiv teilt und sortiert. **Quick Sort** ist im Durchschnitt ebenfalls sehr effizient mit  $O(n \log n)$ , kann jedoch im schlechtesten Fall auf  $O(n^2)$  abfallen, wenn das Pivot-Element ungünstig gewählt wird.

**Insertion Sort** ist für kleine oder teilweise sortierte Listen geeignet, da er einfacher ist als Merge Sort, aber ebenfalls eine Zeitkomplexität von  $O(n^2)$  im schlechtesten Fall aufweist.

Bei den Suchalgorithmen ist die **lineare Suche** die einfachste Methode, da sie jedes Element der Liste sequenziell durchsucht. Dies macht sie jedoch für große Listen ineffizient, da sie eine Zeitkomplexität von  $O(n)$  hat. Die **Binärsuche** hingegen ist wesentlich schneller mit einer Zeitkomplexität von  $O(\log n)$ , setzt jedoch voraus, dass die Liste sortiert ist. Die **Jump-Suche** kombiniert Elemente der linearen und Binärsuche und bietet eine mittlere Leistung mit einer Zeitkomplexität zwischen  $O(\sqrt{n})$  und  $O(n)$ , abhängig von der Sprunggröße. Die **Interpolationssuche** ist besonders effizient für gleichmäßig verteilte Daten und kann eine Zeitkomplexität von  $O(\log \log n)$  erreichen, ist jedoch im schlechtesten Fall mit  $O(n)$  vergleichbar, wenn die Daten ungleichmäßig verteilt sind.

Zusätzlich gibt es die **Breitensuche (BFS)** und die **Tiefensuche (DFS)**, die beide für die Suche in Baumstrukturen verwendet werden. **BFS** durchsucht die Baumebenen nacheinander und ist besonders nützlich, wenn das gesuchte Element nahe der Wurzel liegt. Sie hat eine Zeitkomplexität von  $O(V + E)$ , wobei  $V$  die Anzahl der Knoten und  $E$  die Anzahl der Kanten ist. **DFS** hingegen durchsucht so tief wie möglich entlang eines Zweigs, bevor sie zurückverfolgt. Dies ist effizient, wenn das gesuchte Element tief im Baum liegt, und hat ebenfalls eine Zeitkomplexität von  $O(V + E)$ .

Insgesamt bieten die verschiedenen Algorithmen unterschiedliche Vor- und Nachteile, abhängig von der Größe und Struktur der Daten sowie den spezifischen Anforderungen der Anwendung. Während Merge Sort und Quick Sort für große Datensätze bevorzugt werden, sind Insertion Sort und Bubble Sort für kleinere oder spezielle Fälle nützlich. Bei den Suchalgorithmen ist die Binärsuche aufgrund ihrer Effizienz oft die beste Wahl, solange die Liste sortiert ist, während die lineare Suche durch ihre Einfachheit besticht. BFS und DFS sind besonders nützlich für die Suche in Baumstrukturen und bieten je nach Anwendungsfall spezifische Vorteile.