MusicApp

1 BENUTZERANLEITUNG

Im Folgenden werden die Auswahloptionen und Menüs der *MusicApp* erläutert. Beim Starten der App hat der Nutzer die Auswahl eine neue Playlistenkollektion zu erstellen oder eine zuvor abgespeicherte Playliste zu laden:

Do you want to create (N)ew playlist collection or (L)oad an existing one?

- N: Eine neue Playlist-Kollektion wird erstellt. Dabei muss der Name der Kollektion angegeben werden, da diese bei Schließen der App unter dem angegebenen Namen im Ordner „saved_playlists“ gespeichert wird.
- L: Eine bereits existierende Musikkollektion wird geladen. Dafür muss der Name einer der gespeicherten Playlist-Kollektionen angegeben werden .

Daraufhin erscheint das **Hauptmenü**:

------ *Main Menu* ------
1. *Choose existing playlist:*
2. *Delete existing playlist*
3. *Create new playlist*
4. *Show playlist overview*
5. *Save and Exit*
6. *Exit (without saving)*

Erläuterung der Auswahloptionen im Hauptmenü:

- **Option 1:** Die Namen aller bereits hinzugefügten Playlists werden aufgelistet und es kann eine davon ausgewählt werden. Nach der Auswahl einer Playlist erscheint das Playlist-Menü (siehe unten).
- **Option 2**: Eine bereits hinzugefügte Playlist kann mitsamt der beinhalteten Songs gelöscht werden. Dafür muss der Name der Playlist angegeben werden.
- **Option 3:** Eine neue, leere Playlist mit einem wählbaren Namen kann erstellt werden. Danach erscheint das Playlist-Menü (siehe unten) mit der Möglichkeit Songs hinzuzufügen.
- **Option 4:** Es wird eine Übersicht aller hinzugefügten Playlists ausgegeben. Darin sind die Anzahl Songs pro Playlist und die gesamte Länge aller Musikstücke der Playliste enthalten.
  Beispielausgabe: *Playlist 1: contains 10 songs (total time: 58.45 minutes)*
  Playlist 2: contains 20 songs (total time: 74.77 minutes)
- **Option 5:** Die Playlist-Kollektion wird im Ordner „saved_playlists“ unter dem anfänglich gewählten Namen der Kollektion gespeichert und das Programm wird beendet.
- **Option 6:** Das Programm wird beendet. Änderungen an den Playlists werden nicht gespeichert.

Das **Playlist Menü** erscheint sobald eine Playlist ausgewählt wurde und erlaubt das Hinzufügen und Löschen von Songs sowie Such- und Sortieroperationen.

--- Playlist 'Playlist 1' ---
1. Add Song:
2. Delete Song
3. Display Songs
4. Shuffle Playlist
5. Sort Playlist
6. Search Song
7. Print Memory of Playlist
8. Back to Playlist Menu

Erläuterung der Auswahloptionen im Playlist Menü:

- **Option 1:** Der Nutzer hat die Wahl, entweder selbst einen Song zur Playlist hinzuzufügen oder zufällig generierte Songs einzufügen. Möchte der Nutzer einen Song manuell hinzufügen, werden der Songtitel, Künstler, das Album, Genre und die Länge des Musikstücks nacheinander abgefragt. Entscheidet er sich dafür, mehrere zufällige Songs hinzuzufügen, kann die gewünschte Anzahl angegeben werden, und alle erforderlichen Informationen werden automatisch generiert.
- **Option 2:** Hier kann der Titel eines Songs, der gelöscht werden soll, eingegeben werden.
- **Option 3:** Songname, Künstler, Album, Genre und Länge des Musikstücks aller in der Playlist enthaltenen Songs werden ausgegeben. Es kann gewählt werden, ob die Songs alphabetisch nach Titel sortiert oder in der ursprünglichen Reihenfolge der Playlist angezeigt werden.
- **Option 4:** Mischt die Songs der Playlist in eine zufällige Reihenfolge. Diese Option ist nützlich um verschiedene Sortieralgorithmen zu testen. Eine nach Titel sortierte interne Kopie der Songliste bleibt erhalten um das Suchen nach Songs zu ermöglichen.
- **Option 5:** Es stehen verschiedene Algorithmen zur Auswahl, um die Songs in der Playlist zu sortieren. Nach Auswahl des Algorithmus kann festgelegt werden, ob nach Titel, Artist, Album, Genre oder Länge des Musikstücks sortiert werden soll.
- **Option 6:** Es kann nach einem Song gesucht werden. Dabei kann gewählt werden, ob die Suche über Baumstrukturen oder in der sortierten Liste erfolgen soll.
    - Bei der Suche in Bäumen kann die Art des Baums (binärer Suchbaum, AVL-Baum, Rot-Schwarz-Baum) gewählt werden, um nach einem Songtitel zu suchen. Zusätzlich besteht die Option, den Baum mit einer Breath-First Search mit Hilfe des implementierten Iterators der Baumklasse (nur bei Binärem Suchbaum) oder mit Depth-First Search zu durchlaufen, bis das gewünschte Element gefunden wird.
    - Bei der Suche in Listen stehen verschiedene Algorithmen zur Auswahl, wie Linear Search, Binary Search, Ternary Search, Jump Search, Exponential Search und Fibonacci Search. Es kann gewählt werden, ob nach Titel, Artist, Album, Genre oder Länge des Musikstücks gesucht werden soll. Um die Suche nach anderen Kriterien als dem Titel zu ermöglichen, wird eine lokale Kopie der Songliste vor der Suche nach dem entsprechenden Kriterium vorsortiert. Für den Linearen Suchalgorithmus gibt es die Option, eine angepasste Version zu verwenden, die alle Vorkommen von Songs mit dem gesuchten Genre oder einem anderen Kriterium zurückgibt.
- **Option 7:** Gibt den Speicherverbrauch der Listen- und Baumrepräsentation inklusive aller enthaltenen Objekte der Playlist an.
- **Option 8:** Es kann zum Hauptmenü zurückgekehrt werden, um eine andere Playlist auszuwählen oder das Programm zu beenden.

2 DATENSTRUKTUREN

Die Songs in der MusikApp werden in **Song-Objekten** (class Song) verwaltet. Die Song Objekte beinhalten alle Informationen zu dem entsprechenden Song. Dazu gehören Titel, Künstler, Album, Genre sowie die Länge des Musikstücks in Sekunden. Um Vergleiche zwischen Songs zu ermöglichen wurden die *Magic Methods* für Vergleiche wie greater than (__gt__), lower than (__lt__), und equal (__eq__) implementiert. Default ist hierbei der Vergleich nach dem Songtitel. Allerdings ist es auch möglich die Songs nach den anderen verfügbaren Kriterien (Titel, Künstler, Album, Genre sowie die Länge des Musikstücks) zu vergleichen. Hierzu muss das gewünschte Kriterium als Attribut der Song-Klasse gesetzt werden (mit set_key) was zur Folge hat, dass die Vergleichsoperatoren nun automatisch das gewählte Kriterium berücksichtigen.

Die Songs sind in **Playlist-Objekten** (class Playlist) organisiert. Um in diesem Projekt das Anwenden unterschiedlicher Such- und Sortieralgorithmen zu ermöglichen, sind die Songs redundant in mehreren Listen und Baumstrukturen angeordnet. Diese sind: eine nach Titel vorsortierte Liste zum einfachen Suchen, eine Liste mit der eigentliche Reihenfolge der Songs wie sie in der Playlist abgespielt werden, sowie ein binärer Suchbaum, ein AVL Baum und ein Rot-Schwarz Baum. Es besteht die Möglichkeit die Größe der einzelnen Datenstrukturen inklusive aller enthaltener Objekte ausgeben zu lassen, um deren Speichereffizienz zu vergleichen. Zusätzlich wird zu jeder Playlist der Playlistname sowie die gesamte Laufzeit aller enthaltenen Songs mitaufgeführt. Auch in dieser Klasse sind *Magic Methods* implementiert die es erleichtern über die Songs der Playliste zu iterieren (__iter__) , die Anzahl Songs zu ermitteln (__len__) sowie das Playlist Objekt benutzerfreundlich auszugeben (__str__). Eine mögliche Ausgabe sieht so aus: *Playlist1: contains 1000000 songs (total time: 5082551.47 minutes).*

Die Hauptfunktion der **MusicApp-Klasse** ist es, Playlisten und deren Songs zu verwalten. Die einzelnen Playlisten sind dabei in einem Dictionary gespeichert, um einfach und effizient [O(1)] über deren Namen darauf zugreifen zu können. Es gibt Methoden zum Hinzufügen und Löschen von Playlisten sowie zum Einfügen eines bestimmten oder mehrerer zufällig generierter Songs. Beim Hinzufügen von Songs zur Playlist werden diese automatisch in alle Baumstrukturen und Listen der Playlist aufgenommen und beim Löschen wieder aus allen entfernt. Wenn die Option gewählt wird, zufällige Songs hinzuzufügen, werden Songtitel, Künstler und Album mithilfe verschiedener Bibliotheken aus zufällig generierten echten Wortkombinationen erstellt. Eine Liste von möglichen Genres wird einmalig über die Deezer-API abgerufen, und ein zufälliges Genre daraus gewählt. Es besteht die Funktionalität, die erstellten Playlisten abzuspeichern. Hierfür wird automatisch eine JSON-Datei mit dem gewählten Namen der MusicApp im Ordner saved_*playlist* erstellt. Aus dieser Datei lassen sich die Playlisten bei einem erneuten Start der MusicApp wieder vollständig rekonstruieren.

Eine weitere essentielle Funktion der MusicApp-Klasse ist das zur Verfügung stellen von insgesamt acht Sortieralgorithmen sowie sechs Suchalgorithmen für Listen. Zusätzlich kann in drei Baumtypen gesucht werden. Die enthaltenen Such und Sortieralgorithmen werden im Folgenden genauer beschrieben.

3 SORTIERALGORITHMEN

3.1.1 Divide and Conquer

Merge Sort und Quick Sort gehören zu den Divide-and-Conquer-Algorithmen. Dabei wird ein Problem in kleinere Teilprobleme zerlegt, die leicht gelöst werden können. Anschließend werden die Teillösungen zur Gesamtlösung kombiniert [1]. Für dieses Vorgehen eignet sich eine rekursive Implementierung gut, aber eine iterative Umsetzung von Divide-and-Conquer-Algorithmen ist ebenfalls möglich. Vorteile der Divide-and-Conquer-Strategie sind unter anderem eine gute Zeitkomplexität (siehe Kapitel 3.2) sowie die Eignung für Multiprozessorsysteme.

Entsprechend dem Divide-and-Conquer-Prinzip wird bei Merge Sort die zu sortierende Liste immer weiter in zwei Hälften geteilt, bis die Liste nur noch ein Element enthält [2]. Die einzelnen Teillisten werden danach sortiert und dann schrittweise zusammengefügt, bis die komplett sortierte Liste vorliegt.

Bei Quick Sort, einem weiteren Divide-and-Conquer-Algorithmus, wird zuerst ein beliebiges Element aus der Liste gewählt, das sogenannte Pivotelement. Daraufhin werden alle Listenelemente, die kleiner als das Pivotelement sind, links vom Pivotelement angeordnet, und die verbleibenden Werte stehen rechts davon. Nun wird das Vorgehen rekursiv für die linke und rechte Hälfte wiederholt [3].

3.1.2 Weitere Sortieralgorithmen

Selection Sort beginnt mit der Suche nach dem kleinsten Element in der Liste. Dabei wird die Liste von vorne nach hinten durchsucht. Das kleinste Element wird an die erste Position verschoben, und danach wird im unsortierten Teil erneut nach dem kleinsten Element gesucht, bis die Liste sortiert ist [4].

Der **Insertion Sort**-Algorithmus beginnt beim zweiten Listenelement und vergleicht es mit dem vorherigen. Ist es kleiner, wird es an die richtige Position verschoben. Anschließend wird das dritte Element mit den davorliegenden Elementen verglichen und ebenfalls entsprechend einsortiert. Dieser Prozess wird für jedes weitere Element der Liste wiederholt, bis die gesamte Liste sortiert ist [5]. Genau wie Selection Sort ist Insertion Sort ein In-Place-Algorithmus, der direkt in der Liste sortiert.

Shell Sort wurde 1959 von Donald Shell entwickelt und war der erste Sortieralgorithmus mit einer besseren Zeitkomplexität als $O(n^2)$. Es ist eine Variante von Insertion Sort, bei der die Liste zunächst teilsortiert wird. Dazu werden Elemente mit einem bestimmten Abstand zueinander verglichen und mit Insertion Sort sortiert. Der Abstand wird schrittweise verkleinert, bis er schließlich eins beträgt, und Insertion Sort auf der gesamten Liste angewendet wird. Durch das Vorsortieren verringert sich die Laufzeit erheblich [6].

Beim **Bubble Sort**-Algorithmus steigen die größten Werte wie Blasen in einem Glas mit Mineralwasser nach oben. Es werden jeweils zwei benachbarte Elemente von links nach rechts verglichen. Ist das linke Element größer als das rechte, werden sie vertauscht. Nach dem ersten Durchlauf steht der größte Wert am Ende der Liste. Dieser Vorgang wird wiederholt, bis alle Elemente sortiert sind, wobei bereits korrekt platzierte Elemente in den folgenden Durchläufen nicht mehr berücksichtigt werden müssen. **Optimized Bubble Sort** verbessert dieses Verfahren, indem die Iteration vorzeitig abgebrochen wird, sobald in einem Durchlauf keine Vertauschungen mehr stattfinden – ein Zeichen dafür, dass die Liste bereits vollständig sortiert ist [7].

Eine erweiterte Variante von Bubble Sort ist **Cocktail Sort**. Dabei wird zuerst das größte Element von links nach rechts verschoben. Danach folgt ein weiterer Durchgang in die entgegengesetzte Richtung, von rechts nach links, in dem das kleinste Element an den Anfang der Liste verschoben wird. Der

Wechsel zwischen den beiden Richtungen ermöglicht es, sowohl große als auch kleine Elemente schneller an die richtigen Positionen zu bringen [8].

3.2 UNTERSUCHUNG ZEITKOMPLEXITÄT

Tabelle 1 Gemessene Laufzeiten und Zeitkomplexität von Sortieralgorithmen. Die Sortieralgorithmen wurden auf Listen der Länge 1000, 10000 und 100000 angewandt. Angegeben sind die durchschnittliche Laufzeit von drei Durchläufen mit zufällig gemischten Listen als Eingabe, sowie die benötigte Zeit eine bereits sortierte und eine absteigend sortierte Liste zu sortieren. Die O-Notation der Zeitkomplexitäten wurde von [9] übernommen. Bei manchen Algorithmen wurde die maximale Anzahl an Rekursionen überschritten (Rec. Exc.). Sortierungen die nach 30 Minuten nicht fertig waren wurden abgebrochen (>30 min).

	Gemessene Zeit [s] für Listenlängen			**Zeitkomplexität O**		
Sortier-Algorithmus angewendet auf drei Listen verschiedener Länge	**Durch-schnitt unsortierte Liste**	**Sortierte Liste**	**Rückwärts sortierte Liste**	**Durch-schnitt**	**Best case**	**Worst case**
Merge 1000 Merge 10000 Merge 100000	0.008 0.115 2.027	0.008 0.074 1.667	0.004 0.084 1.928	n log(n)	n log(n)	n log(n)
Shell 1000 Shell 10000 Shell 100000	0.021 0.835 16.943	0.004 0.442 11.363	0.057 0.688 11.447	n^(1.2)	n log(n)	n^(3/2)
Quick 1000 Quick 10000 Quick 100000	0.012 0.203 6.586	0.565 Rec.Exc. Rec.Exc.	0.443 Rec. Exc. Rec. Exc.	n log(n)	n log(n)	n^2
Insertion 1000 Insertion 10000 Insertion 100000	0.340 41.385 >30 min	0.001 0.0 11	0.746 50.195	n^2	n	n^2
Bubble 1000 Bubble 10000 Bubble 100000	1.988 173.018 >30 min	2.669 132.462	3.694 106.327	n^2	n^2	n^2
Optimized Bubble 1000 Optimized Bubble 10000 Optimized Bubble 10000	1.540 175.718 >30 min	0.001 0.016	1.954 160.347	n^2	n	n^2
Cocktail 1000 Cocktail 10000 Cocktail 10000	1.369 148.986 >30 min	0.001 0.010	2.081 162.751	n^2	n	n^2
Selection 1000 Selection 10000 Selection 100000	0.291 34.656 >30 min	0.284 37.978	0.294 37.390	n^2	n^2	n^2

Die Laufzeiten der Sortieralgorithmen wurden anhand von drei Listen mit 1.000, 10.000 und 100.000 Songs gemessen. Am schlechtesten schnitten dabei Insertion Sort, Selection Sort und alle Varianten von Bubble Sort ab. Diese Algorithmen sind in der MusicApp iterativ mit zwei verschachtelten for- bzw. while-Schleifen implementiert, was zu einer sehr schlechten Zeitkomplexität von $O(n^2)$ führt. Bei

Optimized Bubble Sort und Cocktail Sort gibt es einen Test, der überprüft, ob bei einer Iteration noch Vertauschungen vorgenommen werden mussten. Wenn nicht, bricht der Algorithmus ab. Daher beträgt die Best-Case-Zeitkomplexität für diese beiden Algorithmen O(n), wenn eine sortierte Liste als Eingabe verwendet wird.

Merge Sort, Shell Sort und Quick Sort haben gut abgeschnitten. Dabei war Merge Sort am schnellsten und benötigte im Durchschnitt bei unsortierten Listen der Länge 100.000 nur etwa zwei Sekunden. Er hat in jedem Fall eine garantierte Zeitkomplexität von O(n log(n)). Diese Zeitkomplexität ist typisch für Divide-and-Conquer-Algorithmen.

Quick Sort ist ebenfalls ein Divide-and-Conquer-Algorithmus, hat jedoch im schlimmsten Fall eine Zeitkomplexität von $O(n^2)$. Bei Shell Sort hingegen ist die Zeitkomplexität auch abhängig von einer geschickten Wahl der Lückengröße.

In der MusicApp ist Quick Sort rekursiv implementiert, und es wurde die maximale Rekursionstiefe bei großen, sortierten Listen als Eingabe überschritten, was zum Abbruch des Programms führte. Generell hat die rekursive Implementierung von Algorithmen mit O(n) eine höhere Speicherkomplexität als die Implementierung von In-Place-Sortierungen [10]. Diese haben eine Speicherkomplexität von O(1).

4 SUCHALGORITHMEN

In einem **binären Suchbaum** hat jeder Elternknoten höchstens zwei Kindknoten. Der Wert des linken Kindes ist kleiner als der des Elternknotens, und der Wert des rechten Kindes ist größer. Auf diese Weise kann der Baum effizient durchsucht werden, da immer nur der passende Teilbaum weiter durchsucht werden muss [11].

Ein **AVL-Baum** ist ein binärer Suchbaum, der eine maximale Höhendifferenz von einem Knoten zwischen seinen Teilbäumen aufweist. Der Höhenunterschied zwischen dem linken und dem rechten Teilbaum eines Knotens wird durch den Balancefaktor beschrieben. Durch Rotationsoperationen wird sichergestellt, dass beim Verändern des Baumes durch Einfügen neuer Knoten oder Löschen vorhandener Knoten die Balance erhalten bleibt [12].

Ein **Rot-Schwarz-Baum** ist ebenfalls ein selbstbalancierender binärer Suchbaum. Das Gleichgewicht wird durch die Färbung der Knoten (rot oder schwarz) bestimmt, die bestimmten Regeln unterliegt. Anhand dieser Regeln werden die nötigen Ausgleichsrotationen festgelegt [13].

Breadth-First Search (BFS) und **Depth-First Search (DFS)** sind Algorithmen, um alle Knoten eines Baumes zu erkunden. BFS durchläuft einen Baum in einer breiten, schichtenweisen Reihenfolge. Eine Warteschlange wird verwendet, um die noch zu untersuchenden Knoten zu verwalten. DFS hingegen erkundet den Baum astweise in die Tiefe. Ein Stapel wird verwendet, um die Grenzknoten zu speichern [14]. Diese Algorithmen können auch in nicht-binären unsortierten Bäumen verwendet werden.

Linear Search ist eine sequentielle Suchmethode, bei der die Liste beginnend von vorne elementweise durchsucht wird, bis das gewünschte Element gefunden ist. Es spielt dabei keine Rolle, ob die Liste sortiert ist [15]. Für sortierte Listen sollten jedoch effizientere Algorithmen bevorzugt werden.

Der **Binary Search** Algorithmus folgt dem Divide-and-Conquer-Ansatz. Zuerst wird geprüft, ob das mittlere Element der sortierten Liste dem gesuchten Element entspricht. Falls ja, wird der Index dieses mittleren Elements zurückgegeben. Ist das gesuchte Element kleiner als das mittlere, wird die Suche rekursiv auf die linke Hälfte der Liste beschränkt. Andernfalls wird die rechte Hälfte durchsucht. Dieser Vorgang wird fortgesetzt, bis das Element gefunden oder die gesamte Liste durchsucht wurde [16].

Ternary Search funktioniert nach dem gleichen Prinzip wie Binary Search, allerdings wird die Liste durch die Wahl von zwei Mittelpunkten in drei Teile geteilt, was die Suche effizienter macht [17].

Beim **Jump Search** Algorithmus wird zuerst die Sprungweite festgelegt, zum Beispiel die Quadratwurzel der Listenlänge. Anschließend wird in der Liste so lange "gesprungen", bis das aktuelle Element größer (oder gleich) dem gesuchten Element ist. Das Element wird dann im vorherigen Sprungblock mithilfe einer linearen Suche ermittelt [18].

Exponential Search ähnelt Jump Search, aber die Größe der Sprungblöcke steigt exponentiell an. Sobald der geeignete Bereich gefunden ist, wird Binary Search angewendet, um das Element zu finden [19].

Bei **Fibonacci Search** wird die Anzahl der möglichen Positionen, an denen ein Element gefunden werden kann, reduziert, indem basierend auf der Fibonacci-Reihe Werte übersprungen werden. Die Logik ähnelt der des Binary Search. Diese Methode ist besonders nützlich für sehr große Listen, die nicht komplett in den Speicher passen [20].

4.1 UNTERSUCHUNG ZEITKOMPLEXITÄT

Tabelle 2 Gemessene Zeiten verschiedener Algorithmen zur Suche von drei Songtiteln in einer Liste mit einer Länge von 1.000.000 Einträgen. In der Spalte "Zeitkomplexität" steht „n“ für die Anzahl der Songs in der Playlist, und „N“ für die Tiefe der Bäume.

	Gemessene Zeit [s]			**Zeitkomplexität O**		
Suchalgorithmus angewendet auf Liste mit 1.000.000 Elementen	**Suche nach Song** „blazing ant of tact“	**Suche nach Song** **„**lyrical lynx“	**Suche nach Song** **„**yellow rabbit“	**Durch-schnitt**	**Best case**	**Worst case**
Exponential	0.01369	0.00000	0.00000	$\log_2 n$	1	$\log_2 n$
Ternary	0.00226	0.00000	0.00000	$\log_3 n$	1	$\log_3 n$
Fibonnaci	0.00944	0.00000	0.00000	log n	1	log n
Jump	0.01343	0.00400	0.00000	$\sqrt{n}$	1	$\sqrt{n}$
Binary	0.00102	0.00000	0.00000	n log(n)	1	n log(n)
Linear	0.14277	0.48248	0.90685	n	1	n
AVL-Baum	0.04864	0.03332	0.00000	log N	1	log N
Rot-Schwarz-Baum	0.01847	0.00114	0.00000	log N	1	log N
Binärer Suchbaum	0.05523	0.00000	0.00444	log N	1	N
Tiefensuche	0.35708	1.14723	2.13316	n	1	n
Breitensuche	42.29745	0.7892	3.62077	n	1	n

Die Suche in sortierten Listen ist im Vergleich zum Sortieren um ein Vielfaches schneller. Daher wurde zur Messung der Zeitkomplexität der Suchalgorithmen eine große Playlist mit einer Million Songs erstellt. Es wurden drei darin enthaltene Songtitel ausgewählt, nach denen gesucht wurde. Diese Songtitel wurden so gewählt, dass sie in einer alphabetischen Sortierung entweder weit vorne, in der Mitte oder weit hinten stehen.

Alle Suchalgorithmen haben eine Best-Case-Performance von O(1), da im Idealfall das erste betrachtete Element bereits das gesuchte ist.

Von den listenbasierten Suchmethoden war die sequentielle Lineare Suche am langsamsten. Selbst bei dem Songtitel „Blazing Ant of Tact“, der alphabetisch weit vorne in der Liste einsortiert ist, war die Lineare Suche mit einer Zeitkomplexität von O(n) etwa zehnmal langsamer als die anderen listenbasierten Algorithmen. Die anderen listenbasierten Suchalgorithmen hatten zwar unterschiedliche Zeitkomplexitäten, wobei Exponential Search, Ternary Search und Fibonacci Search in absteigender Reihenfolge am Zeiteffizientesten sind, aber die Suchen waren so effizient, dass keine nennenswerten Unterschiede gemessen werden konnten. Zum Beispiel lag die Suchzeit für den Song „Yellow Rabbit“ (mit Ausnahme der Linearen Suche) bei allen anderen Algorithmen bei 0,00000 Sekunden.

Bei den baumbasierten Suchverfahren war das vollständige Durchlaufen aller Knoten mit Tiefen- oder Breitensuche, bis das gewünschte Element gefunden wurde, deutlich ineffizienter als die Suche unter Berücksichtigung der speziellen Eigenschaften der Suchbäume. Hier kann eine Zeitkomplexität von O(log(N)) erreicht werden, wobei N von der Tiefe der Bäume abhängt. Da der binäre Suchbaum im Gegensatz zum AVL-Baum und Rot-Schwarz-Baum nicht ausgeglichen ist, könnte im schlimmsten Fall ein linearer Baum, ähnlich einer Liste, vorliegen, was zu einer Worst-Case-Zeitkomplexität von O(N) führen würde, wobei N in diesem Fall der Anzahl aller Songs entspricht.

4.2 Speicherverbrauch der Songliste und der Suchbäume

Der Speicherbedarf der Songliste inklusive der enthaltenen Songobjekte wurde mit denen der Baumstrukturen für eine Million Songs verglichen. Der binäre Suchbaum hatte mit 432 MB den geringsten Speicherverbrauch, gefolgt von der Songliste mit 491 MB, während der AVL-Baum und der Rot-Schwarz-Baum mehr als 635 MB benötigten.

5 QUELLEN FÜR ALGORITHMEN UND FUNKTIONEN

- MusikAPP Grundgerüst: Vorlesung
- Algorithmen für Suchbäume:
    - Binärer Suchbaum: Code wurde aus der Vorlesung übernommen. Ein Depth-First Iterator wurde basierend auf [21] hinzugefügt.
    - AVL-Baum: Code wurde von [22] übernommen und angepasst
    - Rot-Schwarz Baum: Code wurde aus der Vorlesung übernommen. Die Funktionalität zum Löschen von Knoten wurde selbst hinzugefügt
- Suchalgorithmen für Listen:
    - Merge Sort: Code wurde aus der Vorlesung übernommen und angepasst.
    - Quick Sort: Code von [23] übernommen und angepasst.
    - Selection Sort: Code wurde von [24] übernommen und angepasst.
    - Insertion Sort: Code wurde [25]übernommen und angepasst.
    - Shell Sort: Code wurde von [26] übernommen und angepasst.
    - Bubble Sort: Code wurde aus der Vorlesung übernommen und angepasst.
    - Optimized Bubble Sort: Code wurde aus der Vorlesung übernommen und angepasst.
    - Cocktail Sort: Code wurde von [27] übernommen und angepasst.
- Alle Sortieralgorithmen:
    - Linear Search: Code wurde aus der Vorlesung übernommen und angepasst. Es wurde eine zusätzliche Version Implementiert welche die Indices aller Elemente mit dem gesuchten Wert zurückgibt.
    - Binary Search: Code wurde aus der Vorlesung übernommen und angepasst.
    - Ternary Search: Code wurde von [28] übernommen und angepasst.
    - Jump Search: Code wurde aus der Vorlesung übernommen und angepasst.
    - Exponential Search: Code wurde aus der Vorlesung übernommen und angepasst.
    - Fibbonacci Search: Code von [29]übernommen und angepasst.
- Speichern und Laden der Playlist-Kollektion in json-Dateien: Die Funktionalität wurde mit [30]als Vorlage implementiert.

6 QUELLEN

[1] Parewa Labs Pvt. Ltd. Divide and Conquer Algorithm n.d. https://www.programiz.com/dsa/divide-and-conquer (accessed September 26, 2024).

[2] Studyflix. Mergesort: Erklärung mit Beispiel, Pseudocode, Java · [mit Video] n.d. https://studyflix.de/informatik/mergesort-1324 (accessed September 26, 2024).

[3] Quicksort n.d.

[4] W3Schools. DSA Selection Sort n.d. https://www.w3schools.com/dsa/dsa_algo_selectionsort.php (accessed September 26, 2024).

[5] GeeksforGeeks. Insertion Sort Algorithm - GeeksforGeeks n.d. https://www.geeksforgeeks.org/insertion-sort-algorithm/ (accessed September 26, 2024).

[6] Studyflix. Shellsort: Erklärung mit Beispiel, Laufzeit und Codes · [mit Video] n.d. https://studyflix.de/informatik/shellsort-1411 (accessed September 26, 2024).

[7] Studyflix. Bubblesort: Beispiel, Algorithmus, Laufzeit, Java & C · [mit Video] n.d. https://studyflix.de/informatik/bubblesort-1325 (accessed September 26, 2024).

[8] Baeldung. Cocktail Sort | Baeldung on Computer Science n.d. https://www.baeldung.com/cs/cocktail-sort (accessed September 26, 2024).

[9] Kuvina Saydaki. Every Sorting Algorithm Explained in 120 minutes (full series) - YouTube. n.d.

[10] AlgoDaily. AlgoDaily - Understanding Space Complexity n.d. https://algodaily.com/lessons/understanding-space-complexity (accessed September 26, 2024).

[11] StudySmarter. Binärer Suchbaum: Beispiel & Java | StudySmarter n.d. https://www.studysmarter.de/schule/informatik/algorithmen-und-datenstrukturen/binaerer-suchbaum/ (accessed September 26, 2024).

[12] Geeks for Geeks. AVL Tree Data Structure - GeeksforGeeks n.d. https://www.geeksforgeeks.org/introduction-to-avl-tree/ (accessed September 26, 2024).

[13] How Red and Black Trees Work n.d. https://pages.cs.wisc.edu/~jinc/ (accessed September 26, 2024).

[14] Codecademy Team. Tree Traversal: Breadth-First Search vs Depth-First Search | Codecademy n.d. https://www.codecademy.com/article/tree-traversal (accessed September 26, 2024).

[15] W3Schools. DSA Linear Search n.d. https://www.w3schools.com/dsa/dsa_algo_linearsearch.php (accessed September 26, 2024).

[16] W3Schools. DSA Binary Search n.d. https://www.w3schools.com/dsa/dsa_algo_binarysearch.php (accessed September 26, 2024).

[17] educative. What is the ternary search? n.d. https://www.educative.io/answers/what-is-the-ternary-search (accessed September 26, 2024).

[18] HowToDoInJava. Jump Search Algorithm n.d. https://howtodoinjava.com/algorithm/jump-search-algorithm/ (accessed September 26, 2024).

[19] Tutorials Point. Exponential Search Algorithm n.d. https://www.tutorialspoint.com/data_structures_algorithms/exponential_search.htm (accessed September 26, 2024).

[20] Baeldung. Exponential Search | Baeldung on Computer Science n.d. https://www.baeldung.com/cs/exponential-search (accessed September 26, 2024).

[21] Stack Overflow. algorithm - Implementing a depth-first tree iterator in Python - Stack Overflow n.d. https://stackoverflow.com/questions/26145678/implementing-a-depth-first-tree-iterator-in-python (accessed September 26, 2024).

[22] datacamp. AVL Tree: Complete Guide With Python Implementation | DataCamp n.d. https://www.datacamp.com/tutorial/avl-tree?dc_referrer=https%3A%2F%2Fwww.google.com%2F (accessed September 26, 2024).

[23] GeeksforGeeks. Python Program for QuickSort - GeeksforGeeks n.d. https://www.geeksforgeeks.org/python-program-for-quicksort/ (accessed September 26, 2024).

[24] GeeksforGeeks. Python-Programm zum Sortieren nach Auswahl - GeeksforGeeks n.d. https://www.geeksforgeeks.org/python-program-for-selection-sort/ (accessed September 26, 2024).

[25] GeeksforGeeks. Python Program for Insertion Sort - GeeksforGeeks n.d. https://www.geeksforgeeks.org/python-program-for-insertion-sort/ (accessed September 26, 2024).

[26] GeeksforGeeks. Python Program for ShellSort - GeeksforGeeks n.d. https://www.geeksforgeeks.org/python-program-for-shellsort/ (accessed September 26, 2024).

[27] GeeksforGeeks. Python Program for Cocktail Sort - GeeksforGeeks n.d. https://www.geeksforgeeks.org/python-program-for-cocktail-sort/ (accessed September 26, 2024).

[28] GeeksforGeeks. Ternary Search n.d. https://www.geeksforgeeks.org/ternary-search/ (accessed September 26, 2024).

[29] GeeksforGeeks. Fibonacci Search in Python - GeeksforGeeks n.d. https://www.geeksforgeeks.org/fibonacci-search-in-python/ (accessed September 26, 2024).

[30] Stack Overflow. List of objects to JSON with Python - Stack Overflow n.d. https://stackoverflow.com/questions/26033239/list-of-objects-to-json-with-python (accessed September 26, 2024).