Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Grafische Anwendung zur Visualisierung von B-Baum-Operationen

Graphical application for the visualization of B-tree operations

Mohamad Sahyouni

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Andreas Lux

Trier, 20.06.2024

Kurzfassung

Diese Bachelorarbeit beschäftigt sich mit der Visualisierung binärer Suchbäume (BST) und B-Bäume mittels JavaFX. Das vorrangige Ziel dieses Projekts besteht darin, ein umfassendes Verständnis ihrer Funktionsweise zu erlangen.
Das Hauptziel dieser Arbeit ist es, die Arbeitsweise von binären Suchbäumen und B-Bäumen mithilfe von interaktiven Visualisierungen verständlich zu erklären. Diese Darstellungen sollen dazu beitragen, die Funktionsweise dieser Datenstrukturen besser nachvollziehbar zu machen und die theoretischen Konzepte verständlicher darzustellen. Die interaktive Natur der Visualisierung ermöglicht es Benutzern, die Struktur aktiv zu erkunden und die Auswirkungen von Prozessen wie Einfügen und Löschen zu beobachten.
Zum Schluss wird ein Ausblick auf potenzielle zukünftige Entwicklungen gegeben.

Abstract

This bachelor thesis focuses on visualizing binary search trees (BSTs) and B-trees using JavaFX. The goal is to gain a complete understanding of how they work.
The main goal of this work is to explain the operating principles of binary search trees and B-trees through interactive visualizations. The purpose of these representations is to make the functionality of these data structures easier to understand and to clarify theoretical concepts. The interactive nature of the visualization allows users to actively explore the structure and observe the effects of operations such as insertion and deletion.
Finally, possible future developments are prospected

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1

Einleitung und Problemstellung

Datenstrukturen sind in den Bereichen Informatik und Softwaretechnik von großer Bedeutung für die effiziente Speicherung, Abfrage und Manipulation von Daten. Bäume haben aufgrund ihrer hierarchischen Struktur und ihrer vielfältigen Anwendungsmöglichkeiten eine wichtige Bedeutung in diesen Datenstrukturen. Im Mittelpunkt dieses Projekts stehen die Darstellung von zwei elementaren Baumdatenstrukturen: dem binären Suchbaum (BST) und dem B-Baum.
Der binäre Suchbaum organisiert Daten hierarchisch und ermöglicht effiziente Such-, Einfüge- und Löschoperationen. Allerdings kann seine Effizienz im schlimmsten Fall abnehmen. Im Gegensatz dazu ist der B-Baum eine Baumstruktur, die sich selbst ausbalanciert. Er verwaltet sortierte Daten und erlaubt effiziente Operationen auch bei umfangreichen Datensätzen.

Der Schwerpunkt liegt auf JavaFX, einem flexiblen Framework zur Erstellung grafischer Benutzeroberflächen in Java. Dieses Framework ermöglicht die Erstellung einer benutzerfreundlichen Oberfläche, mit der Baumstrukturen in Echtzeit manipuliert und beobachtet werden können.

2

Zielsetzung

Die Zielsetzung dieser Arbeit umfasst die folgenden Punkte:

- Entwicklung eines umfassenden Verständnisses von binären Suchbäumen (BSTs) und B-Bäumen, einschließlich ihrer Eigenschaften, Operationen und Vor- sowie Nachteile.
- Implementierung einer benutzerfreundlichen JavaFX-Anwendung mit ansprechender grafischer Oberfläche.
- Visualisierung von BSTs und B-Bäumen in 2D-Baumansicht.
- Ermöglichung von interaktiven Operationen auf den Bäumen (Einfügen, Löschen, Suchen) mit Echtzeit-Visualisierung der Änderungen.
- Integration von visuellen Rückmeldungen und Animationen zur Veranschaulichung der Baumoperationen.
- Unterstützung von Studierenden und Entwicklern beim Erlernen und Verstehen von BSTs und B-Bäumen.
- Bereitstellung eines interaktiven Werkzeugs zur Erforschung und Analyse der Eigenschaften und Verhaltensweisen von Bäumen.

Durch praktische Anwendung und interaktive Visualisierung sollen diese Zielsetzungen dazu beitragen, die theoretischen Konzepte der Baumdatenstrukturen verständlicher zu machen.

3

Grundlagen

Datenstrukturen spielen eine wichtige Rolle bei der effizienten Datenverarbeitung in der Informatik. Um den Zugriff und die Verarbeitung zu optimieren, werden Daten in verschiedenen Arten gespeichert, wie z.B. Arrays, Listen, Warteschlangen und Bäume. Diese Arten sind einige der elementaren Datenstrukturtypen. Jede Struktur hat ihre eigenen Eigenschaften und Anwendungsbereiche, die sich auf ihre Leistung und Effizienz auswirken.[Kot18]

3.1 Bäume

In der Natur stellen Bäume eine der wichtigsten Pflanzenformen dar, und in der Informatik sind sie eine der bedeutendsten Datenstrukturen zur effizienten Organisation, Speicherung und Verwaltung von Daten. Wie in der Natur gibt es auch in der Informatik verschiedene Arten von Bäumen. Die Beziehungen in einem Baum sind hierarchisch: Einige Objekte stehen „über“ anderen, und einige „unter“ anderen.Die Hauptterminologie für Baumdatenstrukturen stammt eigentlich aus Stammbäumen, wo Begriffe wie `Eltern`, `Kind`, `Vorfahr` und `Nachkomme` am häufigsten zur Beschreibung von Beziehungen verwendet werden.[GTG13]

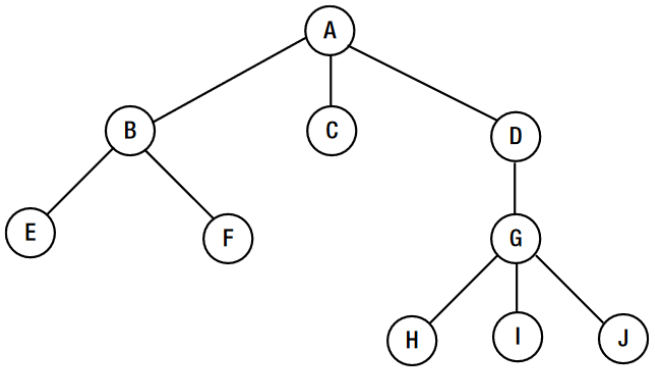

Abbildung 3.1: Ein Baum [Kal14]

Ein Baum ist eine Sammlung von Knoten, die bestimmte Eigenschaften haben. Einer dieser Knoten wird als Wurzel bezeichnet, welche normalerweise ganz oben gezeichnet wird. Die direkten Nachfolger der Wurzel sind die Kinder der Wurzel, während der Vater dieser Knoten die Wurzel ist. Der Vorfahre eines Knotens ist somit der Vater dieses Knotens. Knoten können wie Elemente in einer Liste jeden gewünschten Typ haben.[HUA83]

3.1.1 Definition

Ein Baum `T` ist eine Menge von Knoten, die Elemente speichern, wobei die Knoten eine Eltern-Kind-Beziehung haben, die folgende Eigenschaften erfüllen:

- Falls `T` nicht leer ist, dann hat `T` eine Wurzel, die ein einzigartiger Knoten ist und keinen Vater hat.
- Jeder Knoten `v` aus `T` außer der Wurzel hat einen einzigartigen Vaterknoten `w`.
- Jeder Knoten mit einem Vater `w` ist ein Kind des Knotens `w`.
- Ein Baum kann auch leer sein.

Ein Baum kann auch rekursiv wie folgt definiert werden:
Sei `T` ein Baum. `T` kann entweder leer sein oder einen Knoten `r` haben. `r` ist der Wurzel von `T`, und `a` ist eine Menge von Teilbäumen (wobei `a` möglicherweise leer ist), deren Wurzeln Kinder von `r` sind.[GTG13]

3.1.2 Grundlagen der Baumstruktur

In diesem Abschnitt werden die grundlegenden Konzepte von Komponenten, Tiefe, Grad und Pfaden in Bäumen erläutert.

Die Komponente eines Baumes

Ein Baum besteht grundsätzlich aus Kanten und Knoten.
Knoten können in verschiedene Arten unterteilt werden, wie z.B. `Wurzel`, `Vater`, `Kind`, `innere Knoten` und `Blatt`. `Wurzel`, `Vater` und `Kind` werden oben bereits definiert. Ein `innerer Knoten` ist ein Knoten mit einem oder mehreren Kindern. Falls ein Knoten keine Kinder hat, dann ist er ein `Blatt`. `Geschwister` sind Kinder desselben Knotens [GTG13].

Kanten und Pfade in Bäumen

In einem Baum `T` ist eine `Kante` eine Verbindung zwischen zwei Knoten und wird als ein Paar von Knoten `(u, v)` repräsentiert, wobei `u` der Elternknoten von `v` ist oder umgekehrt (v der Elternknoten von u). Ein `Pfad` in `T` ist eine Folge von Knoten, so dass je zwei aufeinanderfolgende Knoten in der Folge eine Kante bilden [GTG13]. Die Länge eines Pfades in einem Baum wird durch die Anzahl seiner Knoten minus eins bestimmt. Jeder Knoten ist mit einem Pfad der Länge null mit sich selbst verbunden [HUA83].

Höhe, Tiefe und Grad

Die `Höhe` eines Knotens in einem Baum ist die Länge des längsten Pfades von diesem Knoten zu einem Blatt. Im Gegensatz dazu bezeichnet die `Tiefe` (`Ebene`) eines Knotens die Länge des Pfades von diesem Knoten zur Wurzel. Die `Höhe` eines Baumes entspricht der Höhe seiner Wurzel [HUA83].
Der `Grad` eines Knotens ist die Anzahl der Teilbäume des Knotens, anders ausgedrückt, der Grad ist die Anzahl der Kinder [Kal14].

3.1.3 Beispiel: Eigenschaften, Kanten und Pfade des Baumes in Abbildung 3.1:

- `Knoten: A, B, C, D, E ,F, G, H, I, J`
- `Kanten: (A, B), (A, D), (A, C), (B, E), (B, F), (D, G), (G, H), (G, I), (G, J)`

`Knoten`	Typ	Grad	Tiefe	Höhe	Geschwister
`A`	Wurzel	3	0	3	keine
`B`	Vater	2	1	1	D und C
`C`	Blatt	0	1	0	B und D
`D`	Vater	1	1	2	C und B
`E`	Blatt	0	2	0	F und G
`F`	Blatt	0	2	0	E und G
`G`	Vater	3	2	1	E und F
`H`	Blatt	0	3	0	I und J
`I`	Blatt	0	3	0	H und J
`J`	Blatt	0	3	0	H und I

Tabelle 3.1: Zusammenfassung der Knoteneigenschaften

- `Wurzel`: Der Knoten `A` ist die Wurzel des Baumes.
- `Väter`: Die Knoten `A`, `B`, `D`, und `G` fungieren als Väter, da sie jeweils Kindknoten haben.
- `innere Knoten`: Die Knoten `A`, `B`, `D` und `G` sind innere Knoten, da sie mindestens einen Kindknoten besitzen..
- `Kinder`: Die Knoten `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, und `J` sind die Kindknoten des Baumes.
- `Blätter`: Die Knoten `E`, `F`, `H`, `I`, und `J` sind die Blätter des Baumes, da sie keine Kindknoten haben.

- Pfade
  - `Wurzel-Blatt-Pfad`: (A, B, E), (A, B, F), (A, C), (A, D,G, H), (A, D, G, I), (A, D, G, J)
  - `Längster Pfad`: (A, D, G, H) Länge 3

- `Höhe des Baumes`
  Es gibt verschiedene Definitionen für die Höhe eines Baumes. Eine ist oben in 3.1.2 beschrieben, und eine andere Definition laut [Kal14] definiert die Höhe wie folgt: "Die Höhe eines Baumes ist die Anzahl der Ebenen im Baum. Der Baum in Abbildung 3.1 hat eine Höhe von 4. Die Höhe eines Baumes ist um eins größer als seine tiefste Ebene.Ïn dieser Arbeit wird die Definition aus 3.1.2 verwendet.
  Daher ist die Höhe des Baumes gleich der Höhe der Wurzel A und beträgt 3.

Die meisten Informationen, die in diesem Beispiel 3.1.3 verwendet werden, stammen aus [Kal14].

3.1.4 Reihenfolge (Order) der Knoten

In der Baumstruktur werden die Kinder eines Knotens typischerweise von links nach rechts angeordnet. Diese Anordnung impliziert eine gewisse Hierarchie und ermöglicht eine effiziente Navigation durch den Baum.

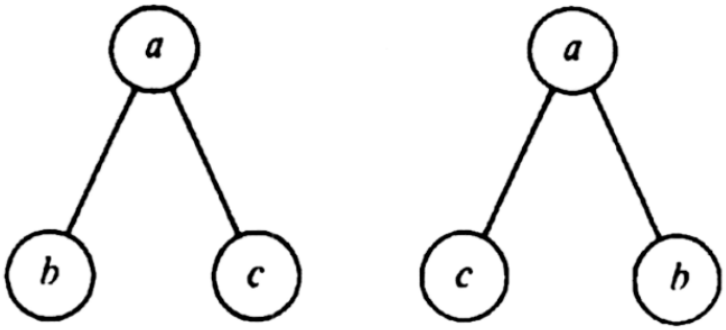

Abbildung 3.2: Zwei Bäume mit unterschiedlicher Reihenfolge [HUA83].

Wenn wir die Reihenfolge der Kinder vernachlässigen und die Bäume in Abbildung 3.2 als `ungeordnete` Bäume betrachten, sind sie gleich. In ungeordneten Bäumen spielt die Reihenfolge der Kinder keine Rolle.

Handelt es sich jedoch um `geordnete` Bäume mit der Reihenfolge "links nach rechts", sind die beiden Bäume in Abbildung 3.2 nicht gleich. Die unterschiedliche Anordnung der Kinder von Knoten `a` führt zu verschiedenen Baumstrukturen [HUA83].

3.2 Binär Suchbäume

Eine besonders beliebte Art von Bäumen sind binäre Suchbäume. Sie sind weit verbreitet und vergleichsweise einfach zu implementieren. Doch Binärbäume dienen nicht nur der Suche. Einige Beispiele für Anwendungen, die durch Bäume beschleunigt werden können, umfassen die Priorisierung von Aufgaben, die Darstellung mathematischer Ausdrücke und syntaktischer Elemente von Computerprogrammen sowie die Organisation der Informationen, die für Datenkomprimierungsalgorithmen benötigt werden [Sha22].

Ein Binärbaum ist ein Baum, bei dem jeder Knoten höchstens zwei Kinder haben kann. Das bedeutet, dass Binärbäume im Grunde Bäume sind, bei denen die Anzahl der Kinder auf zwei begrenzt ist und die alle anderen Eigenschaften normaler Bäume besitzen [McM07].

3.2.1 Rekursive Definition eines Binärbaums:

Ein Binärbaum

* ist entweder `leer`
  oder
* besteht aus einer Wurzel und zwei Teilbäumen, einem linken und einem rechten, wobei jeder Teilbaum selbst ein Binärbaum ist [Kal14].

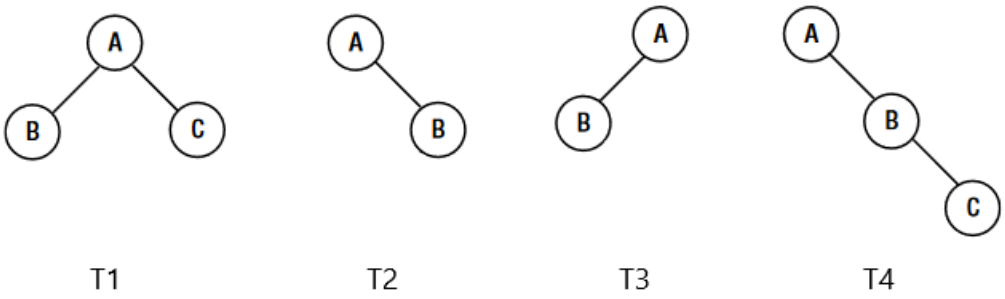

Abbildung 3.3: Vier verschiedene Binärbäume [Kal14].

Die Bäume T1 und T4 aus Abbildung 3.3 sind Binärbäume mit jeweils 3 Knoten, während die Bäume T2 und T3 Beispiele für Binärbäume mit jeweils 2 Knoten sind [Kal14].

Ein Binärbaum heißt vollständig, wenn jeder innere Knoten genau zwei Teilbäume (Kinder) hat [Kal14].
ein spezifische art der Binärbäume ist der Binär Suchbäume

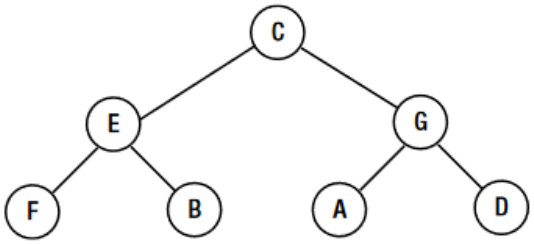

Abbildung 3.4: Ein vollständiger Binärbaum [Kal14].

3.2.2 Reihenfolge in Binärbäumen

Beim Durchlaufen von Binärbäumen, also der Untersuchung aller Knoten in einem Baum, spielen drei spezielle Reihenfolgen eine wichtige Rolle: Preorder, Inorder und Postorder. Jede dieser Reihenfolgen definiert eine bestimmte Abfolge, in der die Knoten des Baumes besucht werden.

Preorder-Reihenfolge (Hauptreihenfolge)

Bei der Preorder-Reihenfolge wird zuerst die Wurzel des Baumes besucht, gefolgt von einem rekursiven Preorder-Durchlauf des linken Teilbaums und anschließend einem rekursiven Preorder-Durchlauf des rechten Teilbaums [Kal14].

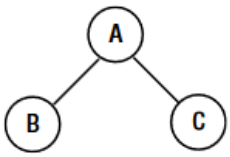

Abbildung 3.5: Die Hauptreihenfolge dieses Binärbaumes lautet: A, B, C [Kal14].

Inorder-Reihenfolge (Symmetrische Reihenfolge)

Im Gegensatz zum Preorder-Durchlauf besucht die Inorder-Reihenfolge zuerst den linken Teilbaum rekursiv, dann die Wurzel und abschließend den rechten Teilbaum rekursiv [Kal14].

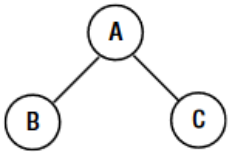

Abbildung 3.6: Die Symmetrische Reihenfolge des Baumes lautet: B, A, C [Kal14].

Postorder-Reihenfolge (Nebenreihenfolge)

In der Postorder-Reihenfolge werden zuerst die beiden Kindknoten eines jeden Knotens rekursiv besucht, bevor der Knoten selbst besucht wird. Der Durchlauf beginnt beim linken Kindknoten. [Kal14].

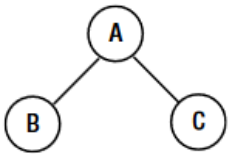

Abbildung 3.7: Die Nebenreihenfolge dieses Binärbaumes lautet: B, C, A [Kal14].

3.2.3 Definition von Binäre Suchbäume

Ein Binärsuchbaum ist entweder leer oder ein Binärbaum mit der folgenden zusätzlichen Eigenschaft: Der Wert jedes Knotens ist größer als alle Werte in seinem linken Teilbaum und kleiner als alle Werte in seinem rechten Teilbaum [McM07].

3.2.4 Suchen in Binär Suchbaum

Die effiziente Suche nach Elementen ist die zentrale Eigenschaft von binären Suchbäumen. Der Suchalgorithmus durchläuft den Baum ausgehend von der Wurzel und vergleicht den Suchwert mit den Werten der Knoten.
Ist der Suchwert kleiner als der Wert des aktuellen Knotens, wird der linke Teilbaum durchsucht. Ist der Suchwert größer, wird der rechte Teilbaum durchsucht. Dieser Prozess setzt sich so lange fort, bis der Suchwert gefunden oder festgestellt wird, dass er nicht im Baum vorhanden ist [Kot18].

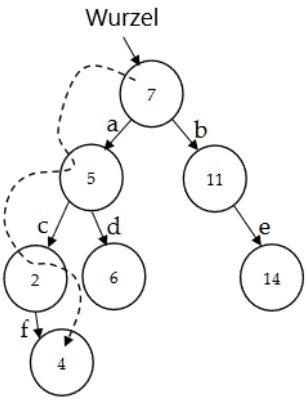

Abbildung 3.8: Suchverfahren nach der Zahl 4 im Binärsuchbaum [Kot18].

Beschreibung der Suchverfahren nach der Zahl 4 im Binärsuchbaum aus Abbildung 3.8:

Beginnend an der Wurzel des Baumes wird der Wert der Wurzel mit der gesuchten Zahl, in diesem Fall 4, verglichen.

Da die gesuchte Zahl kleiner ist als der Wert der Wurzel (7), wird darauf geschlossen, dass sie sich auf der linken Seite der Wurzel befinden muss.

Der linke Kindknoten der Wurzel wird nun besucht, und der Wert des Knotens mit der gesuchten Zahl verglichen.

Dieser Schritt wird wiederholt, bis entweder die gesuchte Zahl gefunden wird oder ein Blattknoten erreicht wird, der die gesuchte Zahl nicht enthält.

Im vorliegenden Beispiel wird dem Pfad von der Wurzel (7) zum linken Kind (5) gefolgt, dann zum linken Kind (2) und schließlich zum rechten Kind (4). Dort wird die gesuchte Zahl 4 gefunden [Kot18].

3.2.5 Einfügen eines neuen Knotens in einen Binären Suchbaum

Erstens wird überprüft, ob der Baum leer ist. Wenn ja, handelt es sich um einen neuen BST, und der einzufügende Knoten wird zum Wurzelknoten. In diesem Fall ist der Algorithmus abgeschlossen. Wenn der Baum jedoch nicht leer ist, muss der BST durchlaufen werden, um die richtige Einfügeposition zu finden.

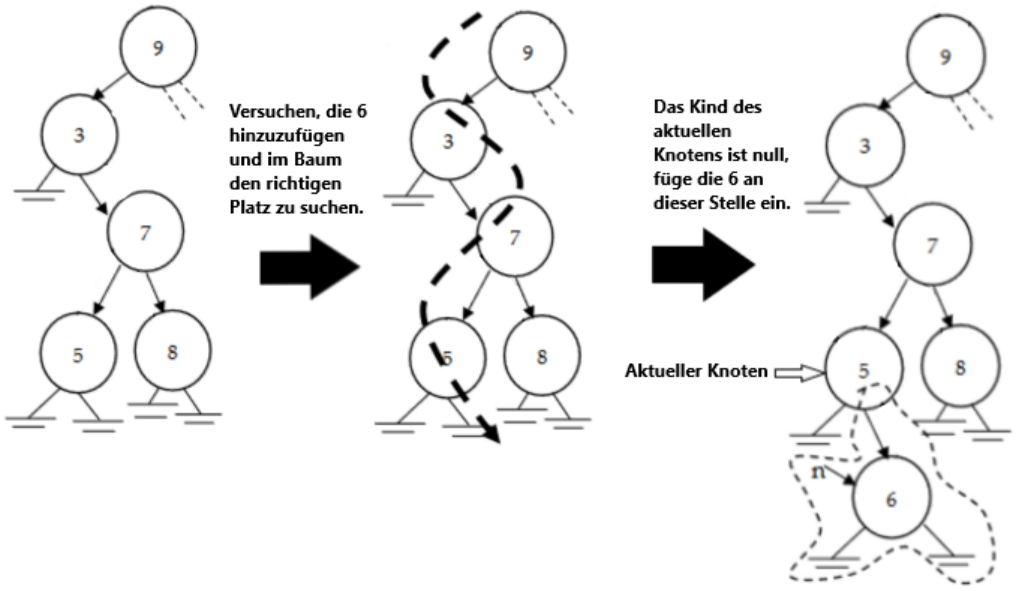

Abbildung 3.9: Den neuen Knoten 6 im Binärsuchbaum hinzufügen[Kot18].

Der Algorithmus:

◦ Startpunkt: Der Algorithmus beginnt am Wurzelknoten des Baumes.

∘Vergleich und Navigation: Der Wert des neuen Knotens wird mit dem Wert des aktuellen Knotens verglichen.

◇ Kleiner als aktuell: Wenn der Wert des neuen Knotens kleiner als der Wert des aktuellen Knotens ist: Ist das linke Kind des aktuellen Knotens null, wird der neue Knoten an dieser Stelle eingefügt und der Algorithmus abgeschlossen. Andernfalls wird der linke Teilbaum des aktuellen Knotens durchlaufen.

◇ Größer als aktuell: Wenn der Wert des neuen Knotens größer als der Wert des aktuellen Knotens ist: Ist das rechte Kind des aktuellen Knotens null, wird der neue Knoten an dieser Stelle eingefügt und der Algorithmus abgeschlossen. Andernfalls wird der rechte Teilbaum des aktuellen Knotens durchlaufen [McM07].

3.2.6 Entfernen eines Knotens von einem Binärsuchbaum

Knoten in einem Binärsuchbaum können entweder ein Kind, zwei Kinder oder keine Kinder (Blatt) haben. In diesem Abschnitt werden die Vorgehensweisen zur Entfernung eines Knotens in allen drei Fällen betrachtet

Entfernung eines Blattes

Um ein Blatt aus einem BST zu entfernen, wird der entsprechende Blattknoten auf null gesetzt [Kal14].

Entfernung eines Knotens mit einem Kind

Wenn ein Knoten genau ein Kind hat, wird der Knoten gelöscht und durch den Teilbaum des Kindes ersetzt [Kal14].

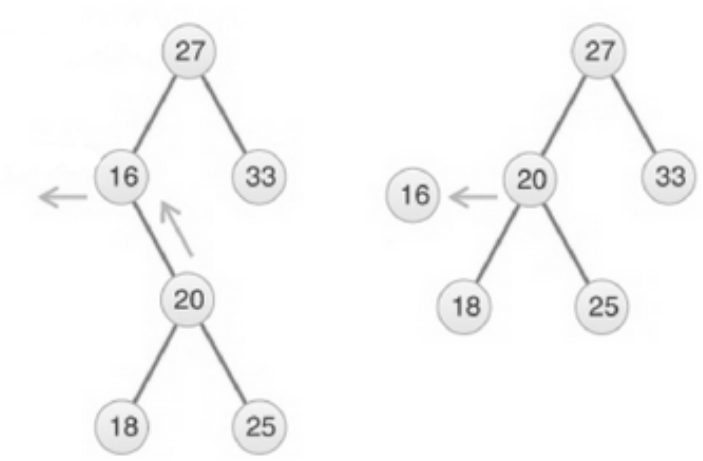

Abbildung 3.10: Beispiel für die Entfernung eines Knotens (16) mit einem Kind [LBC22].

Entfernung eines Knotens mit zwei Kindern

Wenn ein Knoten in einem Binärsuchbaum zwei Kinder hat, ist der Löschvorgang etwas komplexer. Der zu löschende Knoten muss durch einen geeigneten Knoten ersetzt werden, um die Eigenschaften des Binärsuchbaums zu erhalten. Der Ersatzknoten kann entweder der größte Knoten im linken Teilbaum (Vorgänger) oder der kleinste Knoten im rechten Teilbaum (Nachfolger) sein.

Um den Nachfolger zu finden, wird das rechte Kind des zu löschenden Knotens untersucht. Dieser Knoten ist definitionsgemäß größer als der ursprüngliche Knoten. Dann wird den Pfaden der linken Kinder gefolgt, bis keine Knoten mehr vorhanden sind. Da der kleinste Wert in einem Teilbaum am Ende des Pfades der linken Kindknoten liegen muss, führt das Folgen dieses Pfades bis zum Ende zum kleinsten Knoten, der größer als der ursprüngliche Knoten ist [McM07]

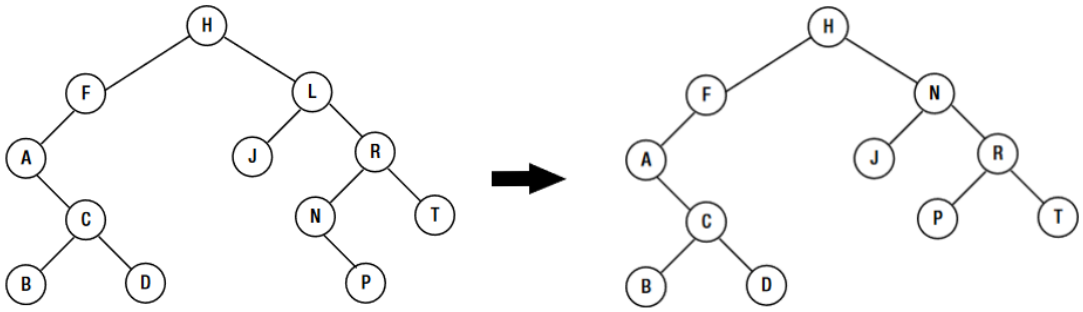

Abbildung 3.11: Beispiel für die Entfernung eines Knotens (L) mit zwei Kinder [LBC22].

3.3 B-Bäume

In der Welt der Datenstrukturen, wo effiziente Speicherung und schneller Zugriff auf Informationen wichtig sind, sind B-Bäume (B-Trees) besonders herausragend. Im Vergleich zu Binary Search Trees (BSTs), die bei der Verarbeitung großer Datenmengen an ihre Grenzen stoßen, stellen B-Trees eine skalierbare Lösung dar, die große Datenmengen problemlos verwalten kann, da sie mehrere Schlüssel pro Knoten speichern können [CLRS22].

3.3.1 Definition von B-Bäume

In diesem Abschnitt werden B-Bäume gemäß [CLRS22] definiert. Ein B-Baum ist eine spezielle Art von balancierendem Suchbaum mit den folgenden Eigenschaften:

1. Jeder Knoten im B-Baum kann mehrere Schlüssel speichern, wodurch der Baum eine hohe Verzweigungsrate hat.
2. Grad: Der Grad (`t`) eines B-Baums bestimmt die maximale Anzahl von Kindknoten pro Knoten.

3. Jeder Knoten in einem B-Baum enthält:
   - Schlüssel: Eine sortierte Menge von Schlüsseln. Die Schlüssel eines Knotens sind in nicht abnehmender Reihenfolge gespeichert.
   - Zeiger: Ein Array von Zeigern, die auf die Kindknoten verweisen. Die Anzahl der Verweise ist um eins größer als die Anzahl der Schlüssel.
4. Die Schlüssel in den internen Knoten trennen die Werte in den Unterbäumen, ähnlich wie bei einem binären Suchbaum, aber mit mehr als zwei Unterbäumen pro Knoten.
5. Alle Blätter sind auf der gleichen Ebene: Der Baum ist immer balanciert, sodass alle Pfade von der Wurzel zu den Blättern die gleiche Länge haben.
6. Die Anzahl der Schlüssel in einem Knoten hängt vom Grad des B-Baumes ab. Ein Knoten darf maximal `2t-1` Schlüssel und `2t` Kinder haben, während alle Knoten außer der Wurzel mindestens `t-1` Schlüssel und `t` Kinder haben müssen.

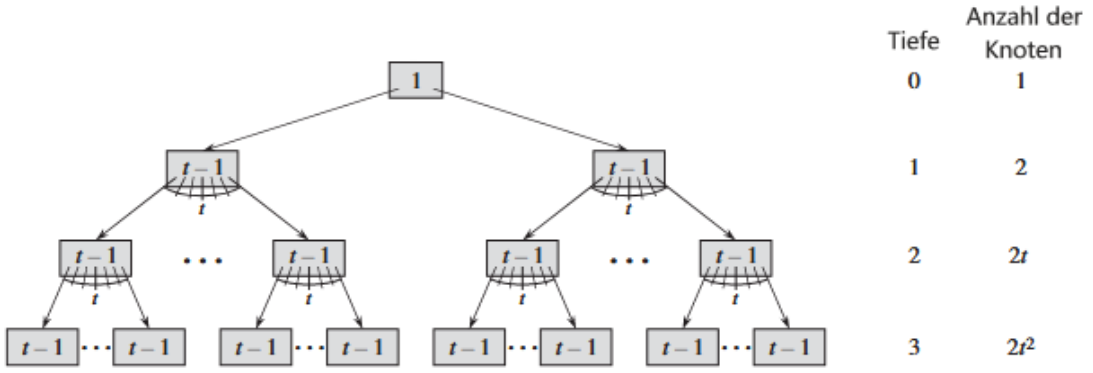

Abbildung 3.12: Ein B-Baum Mit Grad t und Höhe 3 [CLRS22].

Zunächst werden die drei wichtigsten Operatoren eines B-Baumes behandelt. Diese lauten SUCHEN, EINFÜUGEN und LÖSCHEN.

3.3.2 Suchen in B-Bäume

Die Suche in einem Binären Suchbaum (BST) ist relativ einfach: Der Algorithmus vergleicht den gesuchten Wert mit dem Schlüssel des aktuellen Knotens und entscheidet sich dann für einen der beiden Wege, je nachdem, ob der Wert kleiner oder größer ist als der Schlüssel des aktuellen Knotens. In einem B-Baum ist die Suche ähnlich, jedoch gibt es einen entscheidenden Unterschied: Der Algorithmus muss eine "MehrwegeEntscheidung treffen, die der Anzahl der Kindknoten des jeweiligen Knotens entspricht. Das bedeutet, dass anstatt einer binären Entscheidung, die in einem BST getroffen wird, bei einem B-Baum eine Entscheidung basierend auf der Anzahl der Kindknoten getroffen werden muss [CLRS22].

Ablauf der Suche nach dem Schlüssel k

- Die Suche beginnt am Wurzelknoten T des Baums.
- In jedem Knoten x werden alle Schlüssel $x.key_n$ mit dem Suchschlüssel k verglichen.
- Entscheidung anhand des Vergleichs:
  - Falls ein Schlüssel $x.key_i$ mit k übereinstimmt, wird die Suche beendet.
  - Der Algorithmus durchsucht den aktuellen Knoten nach dem ersten Schlüssel, der größer als k ist, und wechselt zum linken Kindknoten dieses Schlüssels.
  - Falls k größer als alle Schlüssel im Knoten ist, folgt der Algorithmus dem rechten Kindknoten des letzten Schlüssels.
- Der Algorithmus durchläuft diese Schritte rekursiv, bis der Zielschlüssel gefunden wird oder der leere Knoten erreicht wird, was bedeutet, dass der Schlüssel nicht im Baum vorhanden ist [CLRS22].

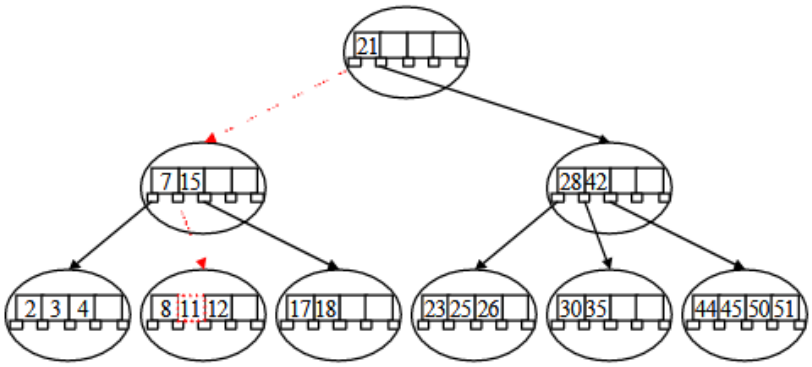

Abbildung 3.13: Der Suchpfad für die Suche nach dem Schlüssel 11 [Mor].

3.3.3 Einfügen in einem B-Baum

In einem BST wird ein neuer Schlüssel einfach hinzugefügt, indem der richtige Platz gesucht und ein neuer Blattknoten mit dem Schlüssel dort eingefügt wird. Dies kann jedoch die Eigenschaften von B-Bäumen verletzen, daher wird zuerst versucht, den Schlüssel in einem vorhandenen Blattknoten hinzuzufügen.
Wenn der Blattknoten bereits voll ist, wird eine Operation zum Teilen eines vollen Knotens y (mit 2t - 1 Schlüsseln) durchgeführt. Diese Operation teilt den Knoten entlang seines mittleren Schlüssels $x.key_t$ in zwei neue Knoten auf, die jeweils nur t - 1 Schlüssel enthalten. Der mittlere Schlüssel wird dabei in den Elternknoten von y verschoben, um die Trennlinie zwischen den beiden neuen Teilbäumen zu markieren [CLRS22].

Wenn auch der Elternknoten von y voll ist, muss er ebenfalls geteilt werden, bevor der neue Schlüssel eingefügt werden kann. Auf diese Weise kann es erforderlich sein, volle Knoten auf dem gesamten Weg nach oben im Baum zu teilen.

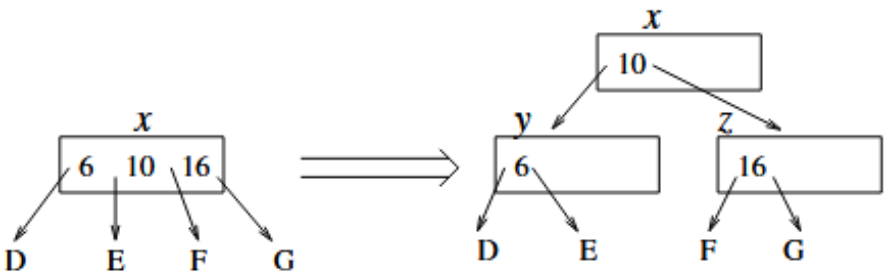

Abbildung 3.14: Teilen eines Knotens [MS04].

Um sicherzustellen, dass der Algorithmus den Baum nur einmal durchläuft, werden alle Knoten auf dem Weg von der Wurzel zum Blatt geteilt.

Das Einfügen eines Schlüssels in einen leeren B-Baum ist relativ einfach. Zuerst wird ein neuer Wurzelknoten mit dem Grad tt des Baumes erstellt. Da der Knoten leer ist, kann der Schlüssel direkt eingefügt werden [CLRS22].

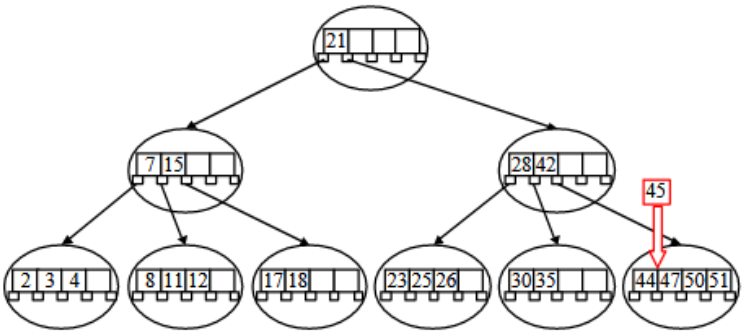

Abbildung 3.15: Ein B-Baum T vor dem Einfügen von 45 [Mor].

Ablauf des Einfügens eines Schlüssels

Um einen Schlüssel in einen B-Baum einzufügen, durchläuft der Algorithmus die folgenden Schritte:

- Suche nach der Einfügeposition.
- Wenn ein voller Knoten erreicht wird, wird auf ihn eine Teilen-Operation durchgeführt.
  - Wenn ein voller Knoten erreicht wird, wird eine Teilen-Operation durchgeführt. Dieser Vorgang wird angewendet, um den vollen Knoten aufzuteilen und Platz für den neuen Schlüssel zu schaffen. Dabei werden zwei neue Blätter erstellt, jedes mit $t-1$ Schlüsseln. Der mittlere Schlüssel des Knotens wird ermittelt und die Schlüssel werden entsprechend auf die neuen Blätter verteilt.
  - Nachdem die Teilen-Operation abgeschlossen ist, wird der neue Schlüssel in das entsprechende Blatt eingefügt.

  - Der Prozess wird fortgesetzt, bis alle vollen Knoten auf dem Weg nach unten geteilt wurden und der neue Schlüssel erfolgreich eingefügt wurde.
- Einfügen des Schlüssels: Nachdem alle vollen Knoten auf dem Weg nach unten geteilt wurden [CLRS22].

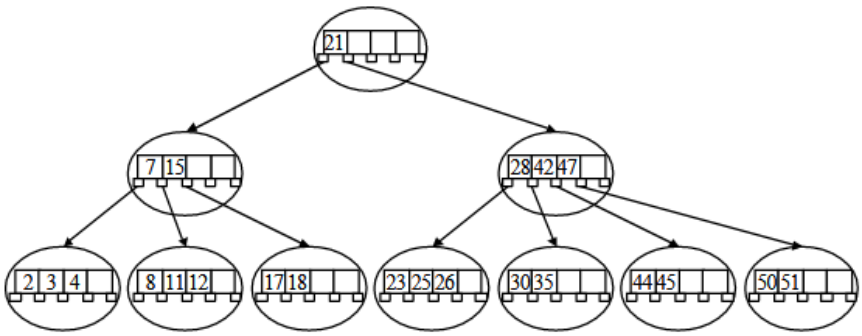

Abbildung 3.16: Der B-Baum T nach dem Einfügen von 45 [Mor].

3.3.4 Löschen eines Schlüssels aus einem B-Baum

Die minimale Knotenfüllung und die Balancierung sind grundlegende Eigenschaften eines B-Baumes, die bei der Entfernung eines Schlüssels beibehalten werden müssen. Daher ist das Löschen eines Elements aus einem B-Baum eine komplexere Operation als das Einfügen oder Suchen.

Die Vorgehensweise beim Löschen wird durch verschiedene Szenarien bestimmt, abhängig davon, ob der zu löschende Schlüssel in einem Blatt oder in einem inneren Knoten liegt. Die minimale Knotenfüllung in einem B-Baum mit Grad t beträgt $t - 1$. Um den Schlüssel in einem Durchlauf zu löschen, muss der Algorithmus sicherstellen, dass alle Knoten auf dem Weg zum zu löschenden Schlüssel mindestens t Schlüssel haben. Dies ist notwendig, da der Knoten, der den zu löschenden Schlüssel enthält, nur $t - 1$ Schlüssel haben könnte, und in diesem Fall könnte ein Schlüssel vom Vaterknoten genommen werden müssen. Wenn ein Knoten auf dem Weg nur $t - 1$ Schlüssel hat, müssen je nach Situation eine der Operationen Verschieben oder Verschmelzen durchgeführt werden, um sicherzustellen, dass dieser Knoten mindestens t Schlüssel besitzt. [CLRS22].

Verschieben:

Beim Verschieben wird ein Schlüssel von einem Knoten zu einem benachbarten Knoten verschoben. Dies geschieht, wenn ein Knoten nur $t - 1$ Schlüssel hat und ein benachbarter Knoten mindestens t Schlüssel hat, um den unterfüllten Knoten zu füllen. Die Verschiebung kann entweder nach links oder rechts erfolgen, je nachdem, ob der Nachbarknoten links oder rechts vom unterfüllten Knoten liegt. Die Verschiebung erfolgt durch eine Rotation über den Vaterknoten [gfU].

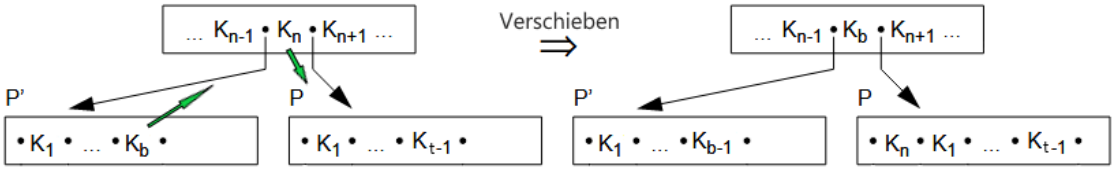

Abbildung 3.17: Beispiel einer Verschiebung zwischen dem Knoten P' mit t Schlüsseln und P mit $t-1$ Schlüsseln [gfU].

Verschmelzen:

Beim Verschmelzen werden zwei benachbarte Knoten zu einem einzigen Knoten verschmolzen, wenn beide Knoten jeweils nur $t-1$ Schlüssel enthalten. Dies geschieht, um die Struktur des Baumes zu erhalten und sicherzustellen, dass kein Knoten unterfüllt ist. Beim Verschmelzen werden die Schlüssel und Zeiger des einen Knotens in den anderen verschoben, und der betreffende Schlüssel im Elternknoten wird entfernt [gfU].

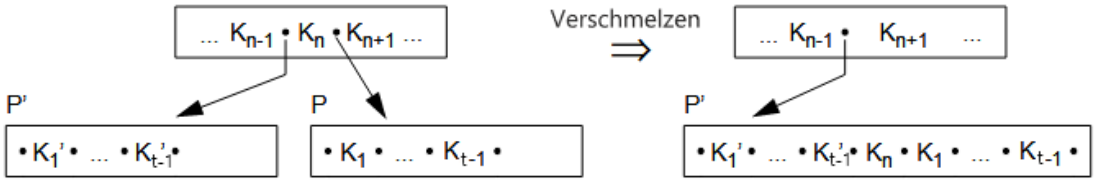

Abbildung 3.18: Beispiel einer Verschmelzung zwischen dem Knoten P' und P, beide mit jeweils $t-1$ Schlüsseln [gfU].

Diese Schritte gewährleisten, dass der B-Baum nach dem Löschen eines Elements seine Struktur und seine Eigenschaften beibehält [CLRS22].

Löschen eines Schlüssels aus einem Blatt

Beim Löschen eines Elements in einem Blatt eines B-Baums gibt es verschiedene Szenarien, die auftreten können:

1. `Löschen aus einem nicht unterfüllten Blatt`: Wenn das Blatt mehr als die minimale Anzahl von Schlüsseln enthält (mindestens t Schlüssel), kann der Schlüssel direkt aus dem Blatt entfernt werden. Dabei verschiebt der Algorithmus gegebenenfalls die verbleibenden Schlüssel, um die Lücke zu füllen [Dro01].

2. `Löschen aus einem unterfüllten Blatt`: Wenn das Blatt nur $t-1$ Schlüssel enthält, muss der Algorithmus Schlüssel von einem benachbarten Blatt übernehmen, um sicherzustellen, dass dieses Blatt mindestens t Schlüssel hat. Dieser Prozess kann je nach Situation als Verschiebung oder Verschmelzung durchgeführt werden. Eine Verschiebung wird vorgenommen, falls das benachbarte Blatt t oder

mehr Schlüssel hat. Eine Verschmelzung erfolgt, falls das benachbarte Blatt ebenfalls nur $t - 1$ Schlüssel hat [Dro01].

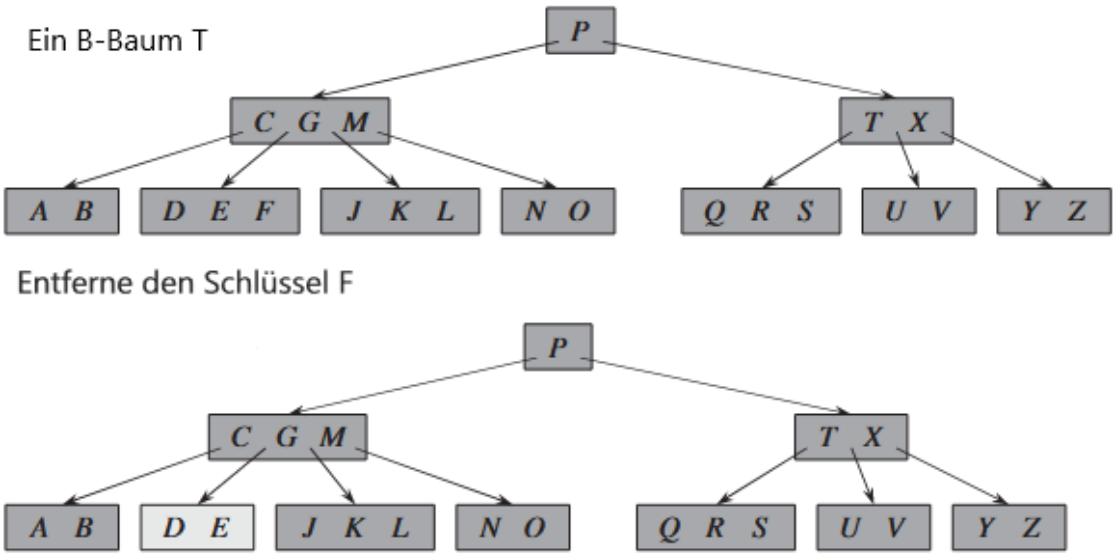

Abbildung 3.19: Das Bild illustriert das Löschen eines Schlüssels aus einem Blatt des B-Baumes T [CLRS22].

Löschen eines Schlüssels in einem inneren Knoten

Wenn der zu löschende Schlüssel in einem inneren Knoten x gefunden wird, sucht der Algorithmus entweder den größten Schlüssel im linken Teilbaum (Vorgänger) oder den kleinsten Schlüssel im rechten Teilbaum (Nachfolger). Dieser Schlüssel wird dann an die Stelle des zu löschenden Schlüssels gesetzt [CLRS22].

`Ersetzen durch Vorgänger oder Nachfolger`:

1. Wenn das linke Kind y, das k vorausgeht, mindestens t Schlüssel hat, ermittelt der Algorithmus den Vorgänger k_0 von k im Teilbaum mit Wurzel y. Der Algorithmus löscht k_0 rekursiv und ersetzt k durch k_0 in x.
2. Wenn y weniger als t Schlüssel hat, untersucht der Algorithmus das rechte Kind z, das k folgt. Wenn z mindestens t Schlüssel hat, ermittelt der Algorithmus den Nachfolger k_0 von k im Teilbaum mit Wurzel z. Der Algorithmus löscht k_0 rekursiv und ersetzt k durch k_0 in x.
3. Wenn sowohl y als auch z nur $t - 1$ Schlüssel haben, verschmilzt der Algorithmus k und alle Schlüssel von z in y. Der Algorithmus entfernt k und den Zeiger auf z aus x und löscht k rekursiv aus y [CLRS22].

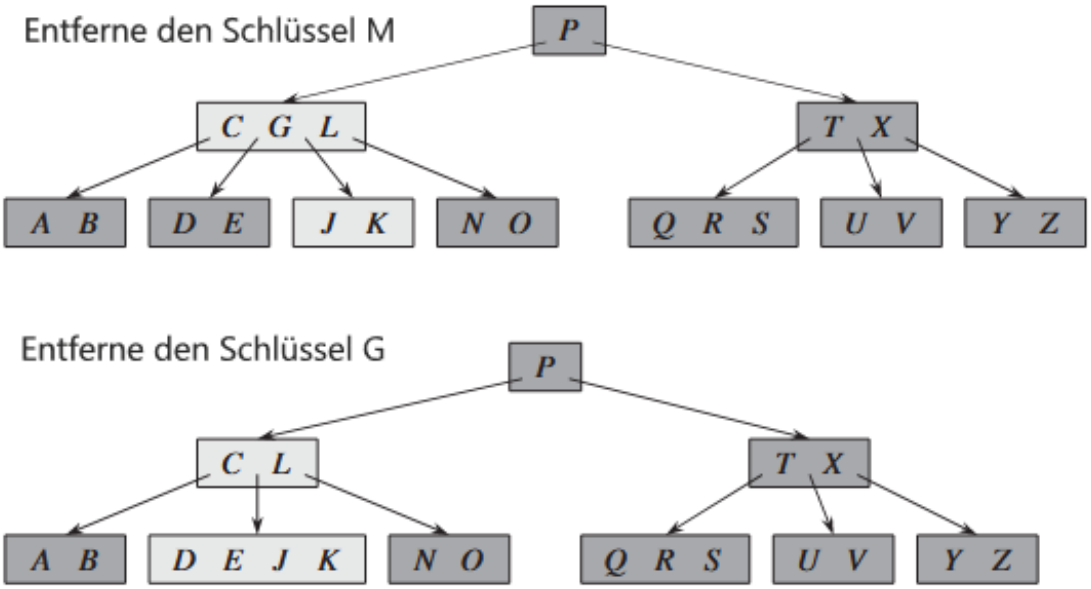

Abbildung 3.20: Das Bild zeigt zwei Beispiele für das Löschen eines Schlüssels aus inneren Knoten des Baumes T aus Abbildung 3.19 [CLRS22].

Wenn der Schlüssel k nicht im inneren Knoten x vorhanden ist, wird der Algorithmus den passenden Unterbaum bestimmen, der k enthalten könnte. Falls der Kindknoten $x : c_i$ nur $t - 1$ Schlüssel hat, wird eine der folgenden Maßnahmen ergriffen:

- Hat $x : c_i$ nur $t - 1$ Schlüssel, aber ein unmittelbares Geschwisterkind mit mindestens t Schlüsseln, wird eine Verschiebung durchgeführt.
- Haben $x : c_i$ und seine unmittelbaren Geschwister nur $t - 1$ Schlüssel, werden $x : c_i$ und ein Geschwisterknoten verschmolzen.

Der Algorithmus fährt dann rekursiv mit dem passenden Kind von x fort [CLRS22].

3.4 JavaFX

JavaFX ist ein fortschrittliches GUI-Toolkit für die Java-Plattform, das Entwicklern die Möglichkeit bietet, plattformübergreifende Desktop-Anwendungen mit visuellen Benutzeroberflächen zu erstellen. Es stellt viele Werkzeuge und Komponenten zur Verfügung, um interaktive Benutzeroberflächen zu entwickeln.

Die Programmiersprache, die ursprünglich F3 (Form Follows Function) genannt wurde, wurde hauptsächlich von Chris Oliver entwickelt. Im Jahr 2007 wurde der Name in JavaFX geändert, um die Integration mit der Java-Plattform zu betonen und die breitere Entwicklergemeinschaft anzusprechen [AA09].

JavaFX hat sich seit seiner ersten Veröffentlichung im Jahr 2008 erheblich weiterentwickelt und zu einem leistungsstarken und vielseitigen Toolkit für die Entwicklung von Rich-Client-Anwendungen entwickelt. Version 1.0 konzentrierte sich auf Desktop-Anwendungen und Web-Applets. Mit Version 1.1, die im März 2009 erschien, wurde die Unterstützung für mobile Geräte (Java ME) hinzugefügt. Version 1.2, veröffentlicht im Juni 2009, führte ein modernes UI-Toolkit ein. Zukünftige Versionen versprachen eine noch größere Reichweite der Plattform, einschließlich TV-Geräten, Blu-ray-Playern und möglicherweise auch persönlichen Videorekordern, sowie eine verbesserte Desktop-Unterstützung mit fortschrittlicheren UI-Steuerelementen [Mor09].

3.4.1 JavaFX-Werkzeuge in diesem Projekt

JavaFX verfügt über viele Werkzeuge, die zum erstellen eines interaktive Benutzeroberfläche verwendet werden können. Dieser Abschnitt beschreibt einige der Werkzeuge, die in diesem Projekt verwendet wurden.

Formen (Shapes)

Die Shape-Klasse fungiert in JavaFX als Basisklasse für alle Formen. Es bietet elementare Merkmale wie Füllung (fill), Umrandung (stroke), Strichbreite (stroke width) und Glätte (smoothness). Das visuelle Aussehen der Form wird durch diese Merkmale bestimmt, darunter ihre Farbe und Umriss. Die Formen wie Kreis, Rechteck, Linie, Polygon usw. sind Unterklassen der Klasse Shape [Ora15b].

verwendete Formen:

- Circle (Kreis): Repräsentiert eine kreisförmige Form, die durch ihre Mittelpunktkoordinaten (centerX und centerY) und ihren Radius definiert ist. [Ora24].
- Rectangle (Rechteck): Erstellt eine rechteckige Form, die durch ihre Breite, Höhe und Position (x- und y-Koordinaten) definiert ist. Die x- und y-Koordinaten beziehen sich auf die Position der oberen linken Ecke des Rechtecks [Ora15c].
- Line (Linie): Repräsentiert ein gerades Liniensegment, das durch seine Anfangs- und Endpunkte (startX, startY, endX, endY) definiert ist [Orand].
- Path (Pfad): Ermöglicht die Angabe einer Pfaddefinition mit den Methoden moveTo, lineTo, curveTo und arcTo [Ora15a].

```
1 import javafx.scene.shape.*;

3 Circle circle = new Circle();
4 circle.setCenterX(100.0f);
5 circle.setCenterY(100.0f);
6 circle.setRadius(50.0f);
```

Listing 3.1: Beispiel zur Erstellung eines Kreises mit einem Radius von 50 Pixeln und einem Mittelpunkt bei den Koordinaten (100, 100) [Ora24].

Übergänge (Transitions)

Übergänge in JavaFX sind grundlegend für die Erstellung von dynamischen Benutzeroberflächen. Sie ermöglichen es, Animationen und Bewegungen zwischen verschiedenen UI-Zuständen zu implementieren. Die folgenden Klassen erben direkt von der Klasse Transition: FadeTransition, FillTransition, ParallelTransition, PathTransition, PauseTransition, RotateTransition, ScaleTransition, SequentialTransition, StrokeTransition und TranslateTransition [Ora13b].

verwendete Übergänge:

- Fade-Transition: Diese Transition erzeugt einen Ein- oder Ausblendeeffekt für visuelle Elemente, indem die Opazität des Nodes kontinuierlich aktualisiert wird, bis der gewünschte Wert erreicht wird [Ora13a].
- Path-Transition: Mit dieser Transition kann die Bewegung eines Elements entlang eines bestimmten Pfades animiert werden. Die Dauer der Animation lässt sich mithilfe der duration-Eigenschaft bestimmen [Pat].
- Parallel-Transition: Diese Transition biete die möglichkeit, mehrere Übergänge gleichzeitig auszuführen [Par].
- Sequential-Transition: Im Gegensatz zur Parallel-Transition spielt diese Transition die Übergänge nacheinander ab. Dies erzeugt eine sequenzielle Animation, die sich gut für geordnete Abläufe eignet [Seq].

```
1  import javafx.scene.shape.*;
2  import javafx.animation.transition.*;

4  ...

6  Rectangle rect = new Rectangle (100, 40, 100, 100);
7  rect.setArcHeight(50);
8  rect.setArcWidth(50);
9  rect.setFill(Color.VIOLET);


12 Path path = new Path();
13 path.getElements().add (new MoveTo (0f, 50f));
14 path.getElements().add (new CubicCurveTo (40f, 10f, 390f, 240f, 1904, 50f)
       );

16 pathTransition.setDuration(Duration.millis(10000));
17 pathTransition.setNode(rect);
18 pathTransition.setPath(path);
19 pathTransition.setOrientation(OrientationType.ORTHOGONAL_TO_TANGENT);
20 pathTransition.setCycleCount(4f);
21 pathTransition.setAutoReverse(true);

23 pathTransition.play();
```

Listing 3.2: Beispiel für eine PathTransition: Eine PathTransition wird verwendet, um ein Rechteck entlang eines definierten Pfades zu animieren. Der Pfad besteht aus einer Bewegung und einer kubischen Kurve. Die Animation dauert zehn Sekunden, kehrt sich automatisch um und wiederholt sich viermal [Pat].

Steuerelemente (Controls)

Die Steuerelemente spielen eine entscheidende Rolle für eine interaktive und benutzerfreundliche Anwendung in JavaFX. Mit ihnen können Nutzer mit der grafischen Oberfläche interagieren, Eingaben tätigen und Aktionen durchführen. Die Unterklasseen der ‘Control‘-Klasse umfassen: Accordion, ButtonBar, ChoiceBox, ComboBoxBase, ScrollBar, ScrollPane, Separator, Slider und weitere [Con].

verwendete Steuerelemente:

- Schaltfläche (Button): Die Klasse Button ist eine Unterklasse von Control. Die Vererbungshierarchie sieht wie folgt aus: Button erbt von ButtonBase, das wiederum von Labeled erbt, und Labeled erbt von Control. Eine Schaltfläche ist ein interaktives Element zur Auslösung von Aktionen durch Klicken [But].
- Schieberegler (Slider): Der Schieberegler wird verwendet, um eine kontinuierliche oder diskrete Auswahl gültiger numerischer Werte anzuzeigen und ermöglicht dem Benutzer die Interaktion mit der Steuerung. Beispielsweise kann er für die Konfiguration von Einstellungen verwendet werden [Sli].
- Textfeld (TextField): Die Klasse TextField ist eine Unterklasse der Klasse TextInputControl, die von Control abgeleitet ist. Textfelder werden verwendet, um Benutzereingaben zu erfassen [Tex].

4

Implementierung

In diesem Kapitel wird die Entwicklung einer JavaFX-Anwendung zur Visualisierung von Binären Suchbäumen (BST) und B-Bäumen beschrieben. Diese Anwendung erlaubt es den Nutzern, Bäume zu erstellen, Knoten hinzuzufügen und zu löschen. Zunächst wird die Struktur des Codes und die wichtigsten Komponenten der Implementierung betrachtet.

4.1 Code-Struktur

Die JavaFX-Anwendung setzt sich aus mehreren Bestandteilen zusammen. Hierzu gehören die Mainscene.fxml als FXML-Datei für die Benutzeroberfläche, der MainsceneController zur Logiksteuerung und die application.css für das Styling. Außerdem umfasst die Anwendung verschiedene Klassen, die sich mit den Algorithmen der Bäume und deren Visualisierung beschäftigen.

4.1.1 klassen und ihrer Funktionalität

- **BinSTree und BSNode**:
  Diese Klassen implementieren einen binären Suchbaum. Die Klasse `BinSTree` stellt die Baumstruktur dar und enthält Methoden zum Hinzufügen, Löschen und Durchsuchen von Knoten. Die Klasse `BSNode` repräsentiert einen einzelnen Knoten im Baum und enthält die Schlüsselwerte sowie Zeiger auf seine linken und rechten Kinder.

- **BTree und BTreeNode**:
  Diese Klassen sind für die Implementierung eines B-Baums verantwortlich. Die `BTree`-Klasse verwaltet die Baumstruktur und bietet Methoden zum Einfügen, Löschen und Durchsuchen von Schlüsseln. Die `BTreeNode`-Klasse repräsentiert einen Knoten im B-Baum und enthält die Schlüsselwerte sowie Zeiger auf seine Kinder.

- **drawBinSearchTree und DrawBTree**:
  Diese Klassen DrawBTree sind für die Visualisierung der Bauminhalte verantwortlich. Mithilfe von JavaFX-Grafikelementen stellen sie die Baumstruktur

grafisch dar und ermöglichen es dem Benutzer, mit den Bäumen zu interagieren, indem sie das Löschen oder Einfügen von Schlüsseln visualisieren.

- **Main**:
  Die Main-Klasse ist das Einstiegspunkt-Programm für die Anwendung. Hier wird die Start-methode definiert, die beim Starten der Anwendung aufgerufen wird. Ihre Hauptaufgabe besteht darin, die Benutzeroberfläche zu initialisieren und die erforderlichen Ressourcen zu laden.

- **MainsceneController**:
  Der MainsceneController ist der Controller für die FXML-Datei der Benutzeroberfläche. Der Controller enthält Methoden und Ereignishandler, die auf Benutzerinteraktionen reagieren und die entsprechende Logik ausführen.

- **launcher**:
  Diese Klasse startet die Anwendung, indem sie die main-methode aus der Main-Klasse aufruft.*(Die Launcher-Klasse wurde implementiert, um ein Problem bei der Erstellung der ausführbaren JAR-Datei zu beheben. Dieser Fehler trat auf, weil die Main-Klasse nicht gefunden werden konnte, wenn sie direkt als Hauptklasse für das Maven-Projekt festgelegt wurde.)*

4.1.2 Benutzeroberfläche

- **Mainscene.fxml**:
  Die Mainscene ist eine FXML-Datei, die das Layout und die Steuerelemente der Benutzeroberfläche definiert. Sie verwendet XML-Tags, um die Struktur der Benutzeroberfläche festzulegen und die Interaktionselemente wie Schaltflächen, Textfelder und andere Komponenten zu platzieren. Diese Datei dient als visuelle Darstellung der Benutzeroberfläche für die Anwendung.

- **application.css**:
  Diese CSS-Datei wird verwendet, um das Styling der Benutzeroberfläche anzupassen und ihr ein ansprechendes Erscheinungsbild zu verleihen.

4.1.3 Klassenstruktur und Beziehungen

Das folgende Klassendiagramm veranschaulicht die Beziehungen zwischen den Klassen in dieser Anwendung. Es bietet einen visuellen Überblick über die Struktur der Anwendung und hilft beim Verständnis der Interaktionen zwischen den verschiedenen Komponenten.

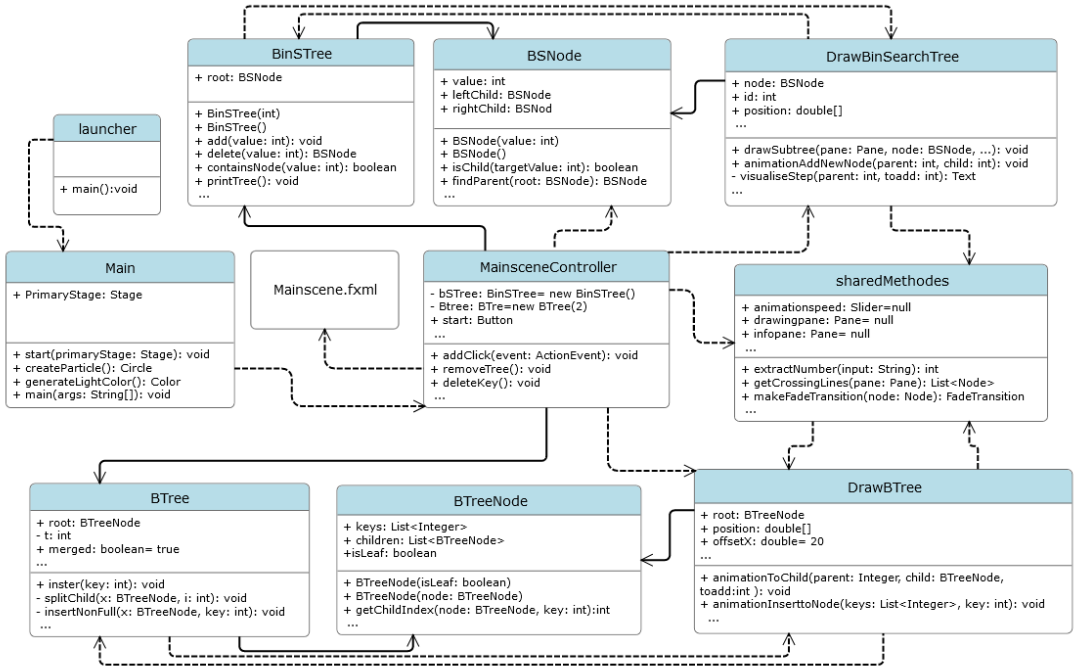

Abbildung 4.1: Klassendiagramm der JavaFX-Anwendung zur Visualisierung von BST und B-Bäumen.

4.2 Implementierung der Animation

In diesem Abschnitt wird die Implementierung der Animation für die Visualisierung von Operationen in einem binären Suchbaum oder B-Baum beschrieben. Durch die visuelle Darstellung können Benutzer die Baumstruktur besser erfassen und die Auswirkungen von Operationen leichter nachvollziehen.

die Implementierung einer Binären Suchbaum besteht aus zwei klassen BSNode die das Knoten eine BST repräsentiert und BinSTree die ein BST repräsetiert.

```
1  public class BSNode {
2      int value;
3      BSNode leftChild;
4      BSNode rightChild;

6      BSNode(int value)
7      {
8          this.value = value;
9          rightChild = null;
10         leftChild = null;
11     }
12     public BSNode() {
13     }
14     //more methodes
15 }
```

Listing 4.1: BSNode Klasse

Die Klasse **BSNode** aus dem Listing 4.1 hat mehrere Attribute und Methoden. Zu den Attributen gehören **int value**, das den Schlüssel (Wert) des Knotens speichert, **BSNode leftChild**, eine Referenz auf den linken Kindknoten, und **BSNode rightChild**, eine Referenz auf den rechten Kindknoten.

Die Klasse verfügt auch über zwei Methoden. Die Methode **boolean isChild(int targetValue)** überprüft rekursiv, ob ein Knoten mit dem *targetValue* als Kind dieses Knotens vorhanden ist. Die Methode **BSNode findParent(BSNode root)** findet und gibt den Elternknoten des aktuellen Knotens ausgehend vom gegebenen Wurzelknoten zurück. Wenn der aktuelle Knoten der Wurzelknoten ist oder nicht im Baum gefunden wird, wird *null* zurückgegeben.

```
1  class BinSTree  {
2      BSNode root;
3      BinSTree(int wert)
4      {
5          root= new BSNode(wert);
6      }
7      public void add( int value) throws InterruptedException {
8          root = addNode(root, null, value);
9      }
10     public void add( int value) throws InterruptedException {
11         root = addNode(root, null, value);
12     }
13     private BSNode addNode(BSNode current, BSNode parent, int value)
           throws InterruptedException
14     {
15         if(current == null) {
16             if(parent!=null)
17             DrawBinSearchTree.animationAddNewNode(parent.value, value);
18             return new BSNode(value);
19         }
20         if (value < current.value) {
21             current.leftChild = addNode( current.leftChild, current, value
                   );
22             if(current.leftChild != null)
23             DrawBinSearchTree.animationToChild(current.value, current.
                   leftChild.value , value);
24         }
25         else if (value > current.value) {
26             current.rightChild = addNode( current.rightChild, current,
                   value);
27             if(current.rightChild != null)
28             DrawBinSearchTree.animationToChild(current.value, current.
                   rightChild.value , value);
29         }
30         else {
31             sharedMethodes.KeyalreadyInTree(value);
32             return current;
33         }
34         return current;
35     }
36     //more methodes
37 }
```

Listing 4.2: BinSTree Klasse

Die Klasse **BinSTree** repräsentiert einen binären Suchbaum. Sie hat ein Attribut, das die Wurzel des binären Suchbaums speichert, und verfügt über viele Methoden zur Verwaltung der Baumstruktur. Zunächst werden die Methoden zum Einfügen

eines Schlüssels betrachtet, um zu zeigen wie die Schritten Animiert werden.

Die öffentliche Methode **add(int value)** ruft die private Methode **addNode** auf, um einen neuen Knoten mit dem angegebenen Wert `value` zum binären Suchbaum hinzuzufügen.

Die private Methode **addNode(BSNode current, BSNode parent, int value)** arbeitet wie in Abschnitt 3.2.5 beschrieben. Sie wird rekursiv aufgerufen, um den richtigen Platz für den neuen Knoten im Baum zu finden. Wenn der aktuelle Knoten `current` null ist, wird ein neuer Knoten mit dem Wert `value` erstellt und zurückgegeben. Dabei wird auch eine Animation hinzugefügt, indem die Methode **animationAddNewNode** aus der Klasse `DrawBinSearchTree` aufgerufen wird. Andernfalls wird überprüft, ob der Wert des neuen Knotens kleiner oder größer als der Wert des aktuellen Knotens ist. Danach wird die Methode rekursiv aufgerufen, um im linken oder rechten Teilbaum des aktuellen Knotens weiterzusuchen. Es wird auch die Methode **animationToChild** aufgerufen, um den Wechsel zum Kindknoten zu visualisieren. Wenn der Wert bereits im Baum vorhanden ist, wird die Methode **KeyalreadyInTree(value)** aus der Klasse `sharedMethodes` aufgerufen. Diese Methode zeigt dem Nutzer einen Text, der darüber informiert, dass der hinzuzufügende Wert bereits vorhanden ist.

```
1  public static void animationToChild(Integer parent, BTreeNode child, int
       toadd) {
2      int ChildMidK = child.keys.get(child.keys.size()/2);
3      Rectangle pRec =  (Rectangle) drawingpane.lookup("#rec"+parent);
4      Rectangle cRec =  (Rectangle) drawingpane.lookup("#rec"+ChildMidK);

6      if(pRec != null && cRec != null)
7      {
8          double[] pRecpos = {pRec.getBoundsInParent().getCenterX(), pRec.
               getBoundsInParent().getCenterY()};
9          double[] cRecpos = {cRec.getBoundsInParent().getCenterX(), cRec.
               getBoundsInParent().getCenterY()};
10         if(moved.containsKey(parent))
11         {
12             pRecpos = Arrays.copyOf(moved.get(parent), moved.get(parent).
                   length);
13         }
14         if(moved.containsKey(ChildMidK))
15         {
16             cRecpos = Arrays.copyOf(moved.get(ChildMidK), moved.get(
                   ChildMidK).length);
17         }
18         if(parent>ChildMidK) pRecpos[0]-=16; else pRecpos[0]+=16;
19         DrawBTree.animationNodeToNode(pRecpos, cRecpos,drawingpane,parent,
               toadd);
20     }

22 }
```

Listing 4.3: Die Methode animationToChild

Die Animation der Pfad von Wurzel bis zum der VaterKnoten des Neuenknotens wird anhand der Methode **animationToChild** animiert. Das Zeichen der

Neuen Knoten und die Anmation des Einfügen wird von der Methode **animationAddNewNode** Dargestellt.

```
1  public static void animationAddNewNode( int parent, int child) throws
       InterruptedException
2  {
3      DrawBinSearchTree parentNode = DrawBinSearchTree.findNodeById("node"+
           parent);
4      double[] position1 = parentNode.position;
5      double[] position2 = null;
6      double offset = DrawBinSearchTree.offsetX;
7      if (parent < child) {
8          position2 = new double[] {position1[0] + offset, position1[1] +
               40};
9      } else {
10         position2 = new double[] {position1[0] - offset, position1[1] +
               40};
11     }
12     Circle circle = new Circle(position1[0] , position1[1] ,15, Color.
           WHITE);
13     circle.setStroke(Color.BLACK);
14     Text text = new Text(position1[0] - 5, position1[1] + 5, String.
           valueOf(child));
15     circle.setVisible(false);
16     text.setVisible(false);

18     drawingpane.getChildren().addAll(circle,text);

20     Path path = new Path();
21     path.getElements().add(new MoveTo(position1[0], position1[1]));
22     path.getElements().add(new LineTo(position2[0], position2[1]));

24     Text infoText = DrawBinSearchTree.visualiseStep(parent, child);
25     PathTransition pathTransition = DrawBinSearchTree.makePathTransition(
           path, circle);
26     PathTransition pathTransition2 = DrawBinSearchTree.makePathTransition(
           path, text);
27     pathTransition.statusProperty().addListener((observable, oldValue,
           newValue) -> {
28         if (newValue == Animation.Status.RUNNING) {
29             circle.setVisible(true);
30             text.setVisible(true);
31             infoText.setVisible(true);
32         }
33         if (newValue == Animation.Status.STOPPED) {
34             drawingpane.getChildren().remove(circle);
35             drawingpane.getChildren().remove(text);
36             infopane.getChildren().remove(infoText);
37         }
38     });

40     ParallelTransition parallelTransition = new ParallelTransition(
           pathTransition,pathTransition2);
41     addAnimations.add(parallelTransition);
42 }
```

Listing 4.4: Die Methode **animationAddNewNode** aus der Klasse `sharedMethodes`.

Die Methode **animationAddNewNode** nimmt zwei Integer-Argumente `parent` und `child` entgegen. Zuerst sucht sie den Elternknoten mithilfe der Methode **findNodeById**, um auf dessen Position zuzugreifen. Anschließend wird die Position des neuen Kindknotens berechnet, um die Bewegungslinie zu erstellen. Daraufhin

werden ein Kreis für den neuen Knoten und ein Text für seinen Wert erstellt, jedoch vorübergehend ausgeblendet. Die Bewegungslinie (`Path`) wird definiert und in eine Pfadübergangsanimation umgewandelt, um den Kreis zu bewegen. Gleichzeitig wird auch der Text entlang der gleichen Linie bewegt. Die gleichzeitige Ausführung beider Pfadübergänge wird durch die Anwendung eines Parallelübergangs implementiert. Während der Animation werden der Kreis, der Text und eine Informationsanzeige eingeblendet. Nach Abschluss der Animation werden die visuellen Elemente aus der Benutzeroberfläche entfernt. Dies geschieht, da sie erneut von der Methode **drawSubtree** mit IDs gezeichnet und in einem Objekt vom Typ `DrawBinSearchTree` gespeichert werden.

Die Implementierung aller Algorithmen für binäre Suchbäume und B-Bäume erfolgt ähnlich wie oben beschrieben. Sie basieren auf den theoretischen Grundlagen, die im Kapitel 3 detailliert erläutert werden. Während der Ausführung dieser Algorithmen wird jeder Schritt dieser Algorithmen mithilfe einer geeigneten Animationsmethode dargestellt, um die Dynamik und Struktur der Baumoperationen zu veranschaulichen.

Diese kontinuierliche visuelle Rückmeldung erleichtert das Verständnis der Baumoperationen und hilft dabei, komplexe Konzepte intuitiv zu erfassen. Die Animationen bieten auch wertvolle Einblicke in die Funktionsweise und die Dynamik von Baumstrukturen.

4.3 Weitere Methoden zur Baumstruktur

Nachdem die grundlegende Implementierung der Baumstruktur und die Animation der Baumoperationen erklärt wurden, werden zusätzliche Methoden vorgestellt. Diese Methoden zielen darauf ab, spezifische Probleme zu lösen und die Effizienz und Stabilität der Baumstruktur zu verbessern.

Vermeidung der Überlappungen von Knoten

Eine wichtige Herausforderung bei der Visualisierung von Bäumen besteht darin, die Überlappung von Knoten zu vermeiden. Wenn die Knoten eines Baumes in der grafischen Darstellung zu nahe beieinander liegen oder sich sogar überschneiden, kann dies die Lesbarkeit und das Verständnis der Baumstruktur verschlechtern. Um dies zu verhindern, werden spezielle Methoden implementiert, wie z.B. die Methode **checkALLPosition()** aus der Klasse **DrawBTree**.

Die Umsetzung eines B-Baums ähnelt der Implementierung eines binären Suchbaums (BST), weist jedoch einen bedeutenden Unterschied auf: In einem B-Baum können Knoten mehrere Schlüssel enthalten. Daher wird für jeden Schlüssel eines Knotens ein Rechteck gezeichnet, und die Rechtecke der Schlüssel eines Knotens werden nebeneinander positioniert.

```
1  public static int[] checkALLPosition() {
2      for(javafx.scene.Node rec1 : drawingpane.getChildren())
3      {
4          if(rec1 instanceof Rectangle)
5          {
6              int val1 = sharedMethodes.extractNumber(rec1.getId());
7              List<Integer> sameLevel = BTree.getKeysAtSameLevel(root, val1)
                   ;
8              for (javafx.scene.Node rec2 : drawingpane.getChildren()) {
9                  if(rec2 instanceof Rectangle)
10                 {   int val2 = sharedMethodes.extractNumber(rec2.getId());
11                     if(!sameLevel.contains(val2)) continue;
12                     if(val1 == val2 || BTreeNode.keysInSameNode(DrawBTree.
                           root,val1, val2))
13                     continue;
14                     double[] position1 = {rec1.getBoundsInParent().
                           getCenterX(), rec1.getBoundsInParent().getCenterY
                           ()};
15                     double[] position2 = {rec2.getBoundsInParent().
                           getCenterX(), rec2.getBoundsInParent().getCenterY
                           ()};
16                     if(moved.containsKey(val1)) position1 = Arrays.copyOf(
                           moved.get(val1), moved.get(val1).length);
17                     if(moved.containsKey(val2)) position2 = Arrays.copyOf(
                           moved.get(val2), moved.get(val2).length);
18                     double deltaX = position1[0] - position2[0];
19                     double deltaY = position1[1] - position2[1];
20                     // Ensure that the keys are located at the same level
                           by comparing their y-coordinates.
21                     // Check if the necessary adjustments have already
                           been made for this pair of eys.
22                     if (Math.abs(deltaX)<= minSpaceBetweenNodes && Math.
                           abs(deltaY)<=5&&
23                     !positionAdjusted.contains(new Pair<>(val1, val2))) {
24                         positionAdjusted.add(new Pair<>(val1, val2));
25                         return new int[] {val1 ,val2};
26                     }
27                 }
28             }
29         }

31     }
32     return null;
33 }
```

Listing 4.5: Die Methode **checkALLPosition** aus der Klasse `DrawBTree`.

Die Methode **checkALLPosition()** beginnt mit einer Schleife über alle Rechtecke im `drawingpane`. Jeder Schlüssel eines Knotens wird als `rec1` bezeichnet. Eine zweite Schleife durchläuft erneut alle Rechtecke, um `rec1` mit jedem anderen Schlüssel (`rec2`) zu vergleichen. Dann werden die Schlüssel für beide Rechtecke extrahiert und miteinander verglichen, um sicherzustellen, dass die beiden Schlüssel auf derselben Ebene liegen und zu zwei verschiedenen Knoten gehören. Wenn dies der Fall ist, wird die Distanz zwischen den beiden Rechtecken überprüft, um die minimale horizontale Distanz zwischen den Knoten beizubehalten. Diese Distanz ist durch das Attribut `minSpaceBetweenNodes` festgelegt. Falls die Rechtecke eine kleinere Distanz haben, werden die beiden Schlüssel zurückgegeben, und die erforderlichen Bewegungen werden durch die Methode **adjustPositions** vorgenommen.

eine Andere methode die zur Vermeidung von überlabbung auch gehört ist die Methode **getCrossingLines**.

```
1  public static List<javafx.scene.Node> getCrossingLines(Pane pane) {
2      List<javafx.scene.Node> lines = pane.getChildren();
3      List<javafx.scene.Node> crossingLines = FXCollections.
           observableArrayList();

5      for (int i = 0; i < lines.size(); i++) {
6          if(lines.get(i ) instanceof Line)
7          {
8              Line line1 = (Line) lines.get(i);
9              for (int j = i + 1; j < lines.size(); j++) {
10                 if(!(lines.get(j) instanceof Line)) continue;
11                 Line line2 = (Line) lines.get(j);
12                 if (doIntersect(line1, line2)) {
13                     crossingLines.add(line1);
14                     crossingLines.add(line2);
15                     return crossingLines;
16                 }
17             }
18         }
19     }
20     return null;
21 }
```

Listing 4.6: Die Methode **getCrossingLines** aus der Klasse `sharedMethodes`.

Diese Methode untersucht jeweils zwei Linien und prüft mithilfe der Methode **doIntersect**, ob sich diese Linien schneiden. Sie verwendet mathematische Berechnungen, um den Schnittpunkt zu finden und zu überprüfen, ob dieser innerhalb der Segmente liegt. Wenn sich zwei Linien schneiden, werden sie zur Liste crossingLines hinzugefügt und zurückgegeben. Andernfalls wird `null` zurückgegeben.

5

Demonstration der Baum-Visualisierung

5.1 Ausführung und Bedienung

Es ist sehr einfach, diese Anwendung zu starten. Es gibt jedoch zwei Möglichkeiten, dies zu tun:

1. **Durch Doppelklicken auf die ausführbare JAR-Datei:** Dies ist die einfachste Methode, um die Anwendung zu starten.
2. **Durch Öffnen der Launcher- oder Main-Klasse mit einem Interpreter wie Eclipse:** Hierbei ist es wichtig zu erwähnen, dass JavaFX bereits installiert sein muss und die JavaFX-Konfiguration im Interpreter durchgeführt wurde.

Bei Java-Versionen älter als Java 21 muss die JAR-Datei über die CMD ausgeführt werden. Dabei wird folgender Befehl verwendet:

```
java -jar <Name der JAR-Datei>
```

Die Benutzeroberfläche der Anwendung bietet mehrere Funktionen, die durch die Interaktion mit bestimmten Steuerelementen aktiviert werden können. Diese Steuerelemente ermöglichen es den Benutzern, verschiedene Aktionen auszuführen und die Anwendung entsprechend ihren Bedürfnissen zu nutzen.

Abbildung 5.1: Steuerelemente der Benutzeroberfläche.

Die Steuerelemente der Anwendung sind wie folgt definiert:

- **Add** Schaltfläche: Fügt einen neuen Schlüssel zum Baum hinzu.
- **Delete** Schaltfläche: Entfernt einen Schlüssel aus dem Baum.
- **Remove Tree** Schaltfläche: Löscht den gesamten Baum.
- **Pause/Play** Schaltfläche: Stoppt/Fortsetzt die Animation.
- **B-Baum/Binärer Suchbaum Schaltfläche**: Wechselt zwischen B-Baum und Binärbaum.

- **Animation Speed** Schieberegler: Ermöglicht das Einstellen der Animationsgeschwindigkeit (nach rechts für schneller).
- **Positions Slider** Schieberegler: Horizontale Bewegung des Baums anpassen.

5.2 Beispiel

Die Anwendung kann entweder einen B-Baum des Grades 2 oder einen binären Suchbaum (BST) darstellen. Hier ist eine kurze Demonstration für einen B-Baum:

Nach dem Öffnen der Anwendung wird zunächst ein Schlüssel mit dem Wert 50 hinzugefügt, indem auf die Schaltfläche **Add** geklickt wird. Dadurch wird ein neuer Knoten mit dem Wert 50 erstellt und im Baum visualisiert.

Nachdem weitere Schlüssel hinzugefügt wurden, wie zum Beispiel 30 und 70, wird bei der Hinzufügung des Schlüssels 20 eine Teilen-Operation durchgeführt, da die Wurzel bereits die maximale Anzahl von Schlüsseln enthält.

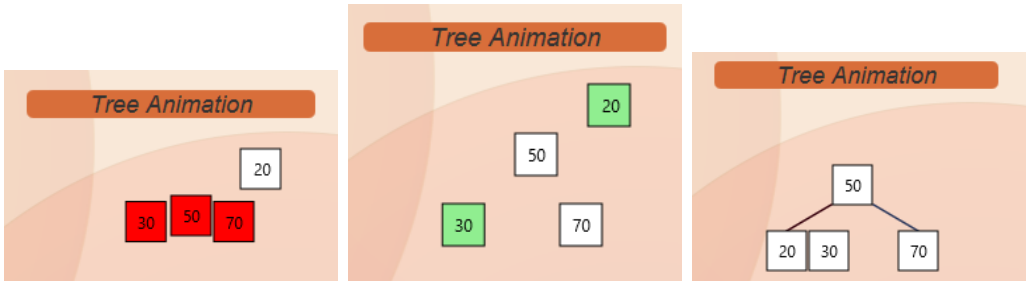

Abbildung 5.2: Schritte der Animation für die Teilen-Operation.

Danach werden die Schlüssel 40 und 45 hinzugefügt. Für die Hinzufügung des Schlüssels 45 muss erneut eine Teilen-Operation durchgeführt werden.

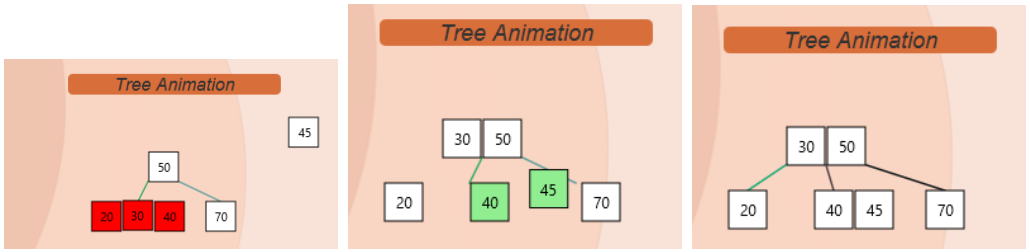

Abbildung 5.3: Schritte der Animation für die Hinzufügung des Schlüssels 45.

Als nächstes wird der Schlüssel 70 gelöscht. Da der Knoten bereits t-1 (1) Schlüssel hat, muss entweder eine Verschiebung oder eine Verschmelzung durchgeführt werden. In diesem Fall hat der benachbarte Knoten t (2) Schlüssel, daher ist es möglich, eine Verschiebung durchzuführen.

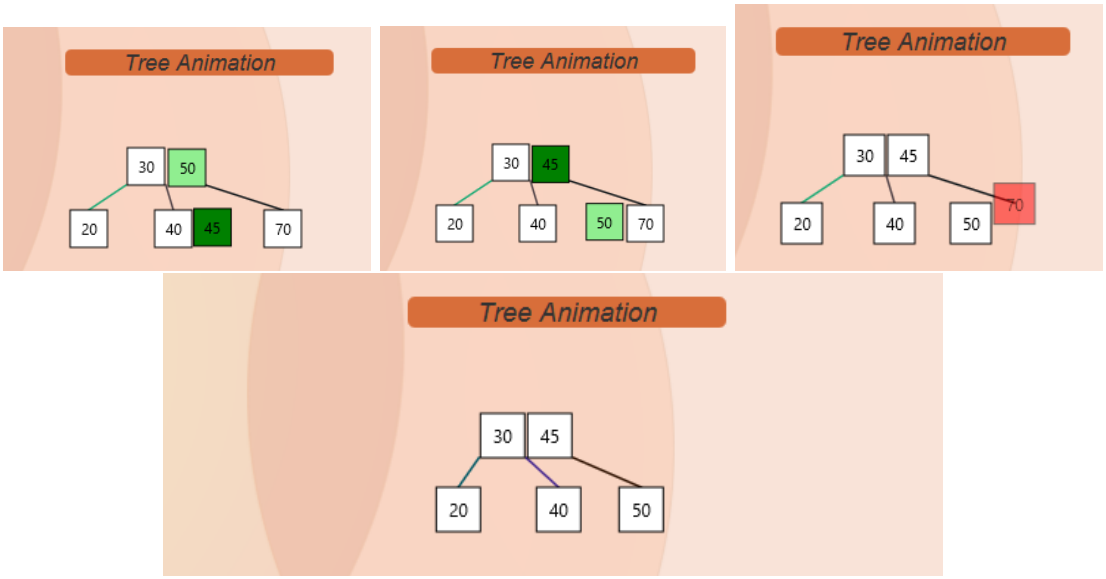

Abbildung 5.4: Schritte der Animation für die Löschung des Schlüssels 70.

5.3 Mögliche Weiterentwicklungen

Die vorgestellte Anwendung stellt bereits eine stabile Basis für die Visualisierung von B-Bäumen und Binärsuchbäumen dar. Trotzdem gibt es viele Optionen zur weiteren Verbesserung und Erweiterung der Anwendung, um sie an verschiedene Anforderungen anzupassen und zusätzliche Funktionalitäten bereitzustellen. Dieser Abschnitt behandelt mögliche Fortschritte und Erweiterungen, die helfen können, die Anwendung noch leistungsfähiger zu gestalten.

5.3.1 Erweiterung der B-Baum-Animation

Zusätzlich zu den genannten Punkten sollte beachtet werden, dass diese Anwendung speziell für B-Bäume vom Grad 2 geeignet ist. Die Visualisierung und die implementierten Methoden in der Klasse **DrawBTree** sind speziell auf diese Art von B-Bäumen abgestimmt.

Um die Anwendung weiter zu entwickeln und für B-Bäume beliebigen Grades geeignet zu machen, könnte es erforderlich sein, die Methoden in der Klasse **DrawBTree** zu überarbeiten. Durch die Implementierung von Schleifen könnten die Operationen und Visualisierungen angepasst werden, um mit einer variablen Anzahl von Schlüsseln und Kindern umgehen zu können.
Die Methoden in der Klasse **BTree** sind bereits mit Schleifen implementiert und eignen sich daher gut für beliebige Grade von B-Bäumen.

Ein Beispiel dafür ist die Methode **AnimationSplitRootMoveChildren** in der Klasse **DrawBTree**. Hier könnte eine Schleife verwendet werden, um höhere Grade zu unterstützen:

```
1  public static void AnimationSplitRootMoveChildren(List<Integer> rootKeys,
       int oldrootChildrensize, BTreeNode newroot, int key)
2  {
3      ...
4      if(newroot.children.get(0).keys.contains(val))
5      {
6          double x = pc.getBoundsInParent().getCenterX()-offsetX;
7          int indexVal = newroot.children.get(0).keys.indexOf(val);
8          newposition[0] =(x - ((newroot.children.get(0).keys.size()-1 )*15)
               %2) +(32 *indexVal+1);
9      }else if(newroot.children.get(1).keys.contains(val))
10     {
11         double x = pc.getBoundsInParent().getCenterX();
12         int indexVal = newroot.children.get(1).keys.indexOf(val);
13         newposition[0] = (x - ((newroot.children.get(1).keys.size()-1 )
               *15)%2) -(32 *indexVal);
14     }else if(newroot.children.get(2).keys.contains(val))
15     {
16         double x = pc.getBoundsInParent().getCenterX()+offsetX;
17         int indexVal = newroot.children.get(2).keys.indexOf(val);
18         newposition[0] = (x - ((newroot.children.get(2).keys.size()-1 )
               *15)%2) -(32 *indexVal+1);
19     }else if(newroot.children.get(3).keys.contains(val))
20     {
21         double x = pc.getBoundsInParent().getCenterX()+offsetX*2;
22         int indexVal = newroot.children.get(3).keys.indexOf(val);
23         newposition[0] = (x - ((newroot.children.get(3).keys.size()-1 )
               *15)%2) -(32 *indexVal);
24     }
25     ...
26 }
```

Listing 5.1: Die Methode **AnimationSplitRootMoveChildren** aus der Klasse **DrawBTree**.

Durch die Verwendung von Schleifen können Operationen wie die Aufteilung von Knoten effizienter und flexibler gestaltet werden, sodass die Anwendung besser für B-Bäume verschiedener Grade geeignet ist.

5.3.2 Neue Baumtypen hinzufügen

Eine weitere sinnvolle Erweiterung wäre die Implementierung und Visualisierung weiterer Baumtypen, wie z.B. AVL-Bäume oder Rot-Schwarz-Bäume. Diese Baumtypen haben unterschiedliche Eigenschaften und Anwendungsfälle. Durch die Unterstützung verschiedener Baumtypen kann die Anwendung für ein breiteres Spektrum an Datenstrukturen genutzt werden und bietet somit mehr Flexibilität und Anwendungsbereiche.

5.3.3 Funktion Einen Schritt zurück

Die Implementierung einer Funktion, die einen Schritt zurück geht, wäre eine sinnvolle Erweiterung der Anwendung. Mit dieser Funktion könnte das Entfernen oder Hinzufügen eines Schlüssels rückgängig gemacht werden. Dies wäre besonders hilfreich für Benutzer, die komplexe Fälle mehrfach animieren und analysieren möchten.

Zur Implementierung dieser Funktion könnten die Baumoperationen in einer Historie gespeichert werden. Diese Historie würde es ermöglichen, den Baumzustand zu einem früheren Zeitpunkt wiederherzustellen. Dabei könnten Methoden zur Speicherung und Wiederherstellung des Baumzustands verwendet werden, um sicherzustellen, dass die Animation korrekt rückgängig gemacht und erneut abgespielt werden kann.

6

Zusammenfassung und Ausblick

In dieser Arbeit wurden die Implementierung und Visualisierung von Bäumen, insbesondere von B-Bäumen und binären Suchbäumen (BST), detailliert erläutert. Es wurden die grundlegenden Methoden zur Manipulation und Darstellung der Baumstrukturen vorgestellt. Besonderes Augenmerk lag auf der Animation der Baumoperationen, um die Veränderungen innerhalb der Baumstruktur verständlich zu machen.

Die Anwendung bietet eine benutzerfreundliche Oberfläche, die es ermöglicht, verschiedene Baumoperationen wie Einfügen, Löschen und Durchlaufen der Knoten visuell nachzuvollziehen. Für die Umsetzung wurden fortgeschrittene Programmiertechniken verwendet, um eine ansprechende und interaktive Visualisierung zu gewährleisten. Insbesondere wurde darauf geachtet, Überlappungen von Knoten zu vermeiden und die Animationen flüssig und nachvollziehbar zu gestalten.

Des Weiteren wurden einige mögliche Weiterentwicklungen beschrieben und erläutert, wie diese umgesetzt werden können, um die Anwendung noch leistungsfähiger und vielseitiger zu gestalten.

Literaturverzeichnis

AA09. Anderson, Gail und Paul Anderson: *Essential JavaFX*. Pearson Education, 2009.

But. *Button (JavaFX 8)*. `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Button.html`. Accessed: 2024-05-30.

CLRS22. Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest und Clifford Stein: *Introduction to algorithms*. MIT press, 2022.

Con. *Control (JavaFX 8)*. `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Control.html`. Accessed: 2024-05-30.

Dro01. Drozdek, Adam: *Data structures and algorithms in Java*. Brooks/Cole, 2001.

gfU. Ungleichungen, Dann gelten folgende: *1 m-Wege-Suchbäume*.

GTG13. Goodrich, Michael T, Roberto Tamassia und Michael H Goldwasser: *Data structures and algorithms in Python*. John Wiley & Sons Ltd, 2013.

HUA83. Hopcroft, John E, Jeffrey D Ullman und Alfred Vaino Aho: *Data structures and algorithms*, Band 175. Addison-wesley Boston, MA, USA:, 1983.

Kal14. Kalicharan, Noel: *Advanced Topics in Java: Core Concepts in Data Structures*. Apress, 2014.

Kot18. Kotrajaras, Vishnu: *First Book for Data Structures & Algorithms in Java*. Department of Computer Engineering, Faculty of Engineering, Chulalongkorn, 2018.

LBC22. Lafore, Robert, Alan Broder und John Canning: *Data Structures & Algorithms in Python*. Addison-Wesley Professional, 2022.

McM07. McMillan, Michael: *Data structures and algorithms using C*. Cambridge University Press, 2007.

Mor. Moritz, Theile: *Theile, Moritz: B-Bäume. Betreuer: Prof. Dr. D. Kossmann, Vorlesungsskript, Ludwig-Maximilians- Universität München, 2001.* `http://wwwbayer.in.tum.de/lehre/WS2001/HSEM-bayer/BTreesAusarbeitung.pdf`.

Mor09. Morris, Simon: *JavaFX in action*. Simon and Schuster, 2009.

MS04. Mehta, Dinesh P und Sartaj Sahni: *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.

Ora13a. Oracle: *FadeTransition (JavaFX 2.0 API) Documentation*. https://docs.oracle.com/javafx/2/api/javafx/animation/FadeTransition.html, 2013. Accessed: 2024-05-30.

Ora13b. Oracle: *Transition (JavaFX 2.0 API) Documentation*. https://docs.oracle.com/javafx/2/api/javafx/animation/Transition.html, 2013. Accessed: 2024-05-30.

Ora15a. Oracle: *JavaFX 8 Path Documentation*. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Path.html, February 2015.

Ora15b. Oracle: *JavaFX 8 Shape Documentation*. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Shape.html, February 2015.

Ora15c. Oracle: *JavaFX Rectangle Documentation*. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Rectangle.html, 2015.

Ora24. Oracle: *JavaFX Circle Documentation*. https://docs.oracle.com/javafx/2/api/javafx/scene/shape/Circle.html, 2024.

Orand. Oracle: *JavaFX Line Documentation*. https://docs.oracle.com/javafx/2/api/javafx/scene/shape/Line.html, n.d.

Par. *ParallelTransition (JavaFX 2)*. https://docs.oracle.com/javafx/2/api/javafx/animation/ParallelTransition.html. Accessed: 2024-05-30.

Pat. *PathTransition (JavaFX 2)*. https://docs.oracle.com/javafx/2/api/javafx/animation/PathTransition.html. Accessed: 2024-05-30.

Seq. *SequentialTransition (JavaFX 2)*. https://docs.oracle.com/javafx/2/api/javafx/animation/SequentialTransition.html. Accessed: 2024-05-30.

Sha22. Shaffer, Clifford A: *Data structures and algorithm analysis*, 2022.

Sli. *Slider (JavaFX 8)*. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Slider.html. Accessed: 2024-05-30.

Tex. *TextField (JavaFX 8)*. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TextField.html. Accessed: 2024-05-30.

A

Glossar

BST	Binär Suchbaum

B

Selbstständigkeitserklärung

☐ Diese Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

☐ Diese Arbeit wurde als Gruppenarbeit angefertigt. Meinen Anteil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser:

Meine eigene Leistung ist:

Datum

Unterschrift der Kandidatin/des Kandidaten