Ostfalia
Hochschule für angewandte Wissenschaften
Wolfenbüttel

Fakultät Informatik

Erkan Garan, 70467533
Justin Treulieb, 70468597

Bäume zeichnen im Ebenen-Layout

Betreuer:
J. Eckert

Salzgitter
Suderburg
Wolfsburg

Hiermit versicheren wir, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versicheren, dass wir alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet haben, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Wolfenbüttel, den 17. Mai 2022

Kurzfassung

Bäume sind eine besondere Form von Graphen und stellen eine Datenstruktur dar. Diese Bäume bestehen aus zwei Elementen, den Knoten und Kanten. In dieser Arbeit wird sich mit drei verschiedenen Algorithmen zum Zeichnen von Bäumen beschäftigt und vorgestellt. Diese Algorithmen ermöglichen es, Bäume im Ebenen-Layout zu zeichnen. Zwei dieser Algorithmen kommen von Wetherell und Shannon, der Dritte von Reingold und Tilford. Im Folgenden werden die Abläufe dieser Algorithmen beschrieben und im Späteren in Java implementiert. Zuletzt werden diese Algorithmen an einem konkreten Beispiel verglichen und daran deren zuvor beschriebenen Vor- und Nachteile deutlich gemacht.

Abstract

Trees are a special form of graphes. They represent a data structure and consist of two elements, the nodes and edges. Three different algorithms are discussed and presented here. These algorithms make it possible to draw trees in a planar layout. Two of these algorithms come from Wetherell and Shannon, the third from Reingold and Tilford. In the following, the processes of these algorithms are described and later implemented in Java. Furthermore, these algorithms are compared using a specific example. Finally the advantages and disadvantages, which will be described beforehand, will be shown on that particular example.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

WS Wetherell und Shannon

TR Tilford und Reingold

LL Der am weitesten links stehende Knoten im linken Subtree eines Knotens auf höchster Höhe

LR Der am weitesten rechts stehende Knoten im linken Subtree eines Knotens auf höchster Höhe

RL Der am weitesten links stehende Knoten im rechten Subtree eines Knotens auf höchster Höhe

RR Der am weitesten rechts stehende Knoten im rechten Subtree eines Knotens auf höchster Höhe

1. Einleitung

Schwerpunkt der Arbeit wird die Vorstellung und Erklärung von drei verschiedenen Algorithmen zum Zeichnen von Bäumen im Ebenen-Layout sein. Dabei wird das Hauptaugenmerk auf dem Zeichnen von Binärbäumen liegen. Hierfür wurden drei Algorithmen betrachtet, ein naiver und ein verbesserter Algorithmus von Wetherhell und Shannon sowie einer von Reingold und Tilford.

Zu Beginn wird definiert, was Bäume in der Informatik sind und wo diese Anwendung finden. Hiernach werden die zuvor erwähnten Algorithmen vorgestellt. Dabei wird jeweils auf den Ablauf, die Implementierung in Java und auf die Vor- und Nachteile eingegangen. Zum Schluss wird in einer abschließenden Konklusion, anhand von den durch die Algorithmen gezeichneten Bäume, ein Vergleich gezogen.

1.1. Motivation

Das Verwenden von Bäumen als Datenstruktur bietet eine Möglichkeit zur komprimierten und sortierten Darstellung von Daten, sowohl intern im Programmcode als auch extern als Modell. Wetherell und Shannon erkennen dies in ihrer Arbeit und schrieben „[...] a good drawing of a tree is often a powerful intuitive guide to a modeled problem [...]“ [2, S.514]. In der Praxis können Bäume zum Beispiel zur Sortierung bzw. Speicherung von Daten oder zur Darstellung von unternehmensinternen Hierarchien verwendet werden.

1.2. Zielsetzung

Ziel dieser Arbeit ist, ein Verständnis dafür zu schaffen, wie diese Algorithmen funktionieren und was das Ergebnis für einen bestimmten (binären) Baum ist. Fer-

ner sollen die Algorithmen in Java implementiert werden, um eigene Bäume zeichnen zu können. Dabei werden auch die Vor- und Nachteile der einzelnen Algorithmen betrachtet. Zudem sollen anhand der Implementierungen in Java eine weitere Möglichkeit zur Implementierung der Algorithmen gezeigt werden.

2. Bäume in der Informatik

2.1. Definition von Bäumen

Nach Wetherell und Shannon sind Bäume endliche, gerichtete, zusammenhängende, azyklische Graphen [2, S. 515]. Bäume bestehen lediglich aus zwei Elementen, nämlich den Knoten und den Kanten. Die Kanten sind gerichtet und verbinden die einzelnen Knoten miteinander. Dabei hat jeder Knoten maximal einen Vorgänger, welcher als Vater bezeichnet wird, und null bis n viele Nachfolger, welche Kinder genannt werden. Die Wurzel stellt hierbei einen besonderen Knoten dar, da sie der einzige Knoten ohne Vorgänger ist. Zudem besitzt jeder Knoten eine Höhe. Diese Höhe ist definiert als die Anzahl an Kanten zwischen dem Knoten und der Wurzel [2, S. 515]. Außerdem kann jeder Knoten Daten beinhalten. Eine weitere besondere Form von Knoten sind die Blätter. Diese verfügen über keine Kinder [1, S. 4]. Abbildung 2.1 zeigt einen einfachen Binärbaum.

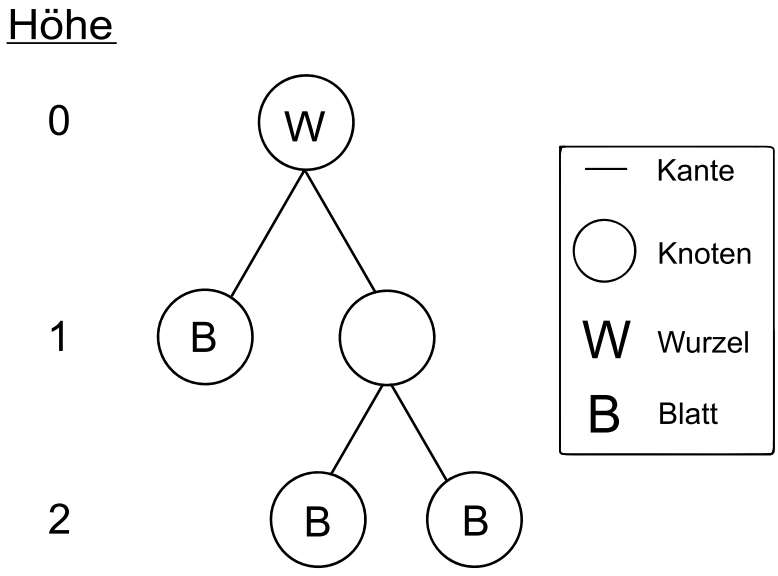

Abbildung 2.1.: Einfacher beschrifteter Binärbaum

Eine spezielle Form von Bäumen stellen die sogenannten Binärbäume dar. Binärbäume sind Bäume, wo jeder Knoten maximal zwei Kinder hat.

Um jeden Knoten eines Baumes abarbeiten zu können, kann über dem Baum traversiert werden. Traversierung bezeichnet das systematische Ablaufen von jedem Knoten eines Baumes beginnend bei der Wurzel. Eine Möglichkeit dazu ist die sogenannte Pre-Order-Traversierung. Dabei wird zunächst der Knoten, dann der linke Teilbaum und zum Schluss der rechte Teilbaum besucht. Hier werden die Väter vor den Kindern durchlaufen. Eine weitere Möglichkeit zur Traversierung ist die Post-Order-Traversierung. Dabei wird als erstes der linke Teilbaum, dann der rechte Teilbaum und dann der Knoten besucht. Bei dieser Art der Traversierung werden die Kinder vor den Väter durchlaufen. Außerdem gibt es noch die In-Order-Traversierung. Bei dieser werden als erstes der linke Teilbaum, dann der Knoten und zum Schluss der rechte Teilbaum durchlaufen [1, S. 22].

2.2. Anwendungsgebiete

Bäume erfahren einen vielfältigen Einsatz in der Informatik, zum Beispiel als Datenstruktur, als Syntaxbäume, als Ausdrucksbäume oder auch als Entscheidungsbäume.

Ein Baum als Datenstruktur kann beispielsweise dazu verwendet werden, um eine Menge von Daten zu sortieren oder in ihnen effizient nach einen bestimmten Datensatz zu suchen. So verwendet der Heap-Sort-Algorithmus einen Baum zum Sortieren von Daten. Ebenso können Bäume, in Form von Syntaxbäumen, dazu verwendet werden, um Syntaxregeln zu überprüfen. Zudem werden Bäume verwendet, um mathematische Ausdrücke auszuwerten. Hierfür wird ein sogenannter Ausdrucksbaum für einen gegebenen mathematischen Ausdruck aufgestellt und ausgewertet. In der Datenanalyse werden Bäume in Form von Entscheidungsbäumen verwendet. Diese werden benutzt um Abhängigkeiten darzustellen. Die Knoten des Baumes stellen in diesem Fall Attribute dar und die Kanten des Knotens die verschiedenen Ausprägungen des Attributs. Die Blätter stellen die vorhergesagten Kategorien dar. So können beispielsweise neue Datensätze, in Abhängigkeit zu ihren Werten, einer bestimmten Kategorie zugeordnet werden.

Wie schon angekündigt, werden Bäume auch außerhalb der Informatik häufig verwendet. Sie können dazu genutzt werden um zum Beispiel eine Hierarchie eines Unternehmens dazustellen oder einen Stammbaum einer Familie wie in der Abbildung 2.2 [1, S. 2].

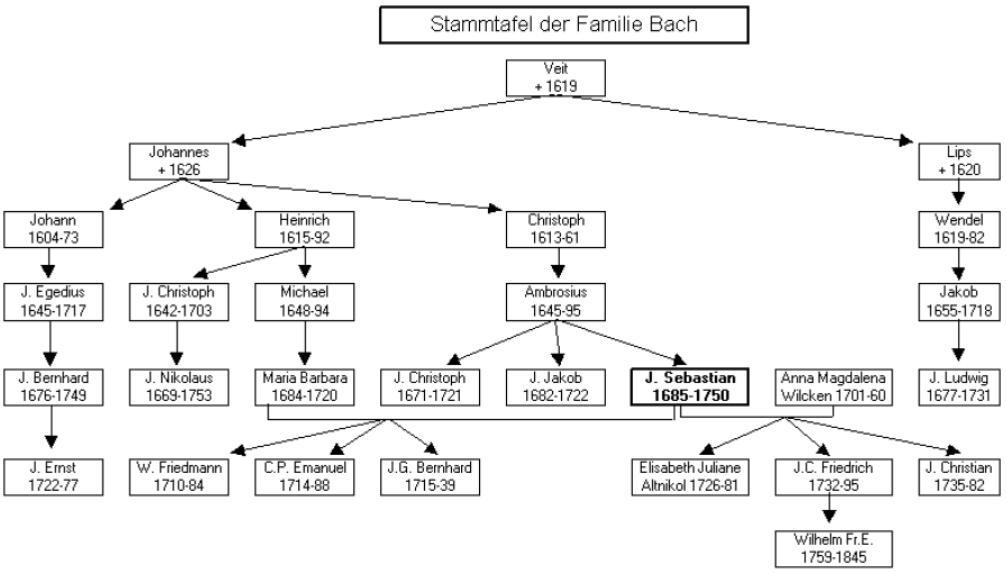

Abbildung 2.2.: Ein Stammbaum, Beispiel für Bäume als Datenstrukturen [1, S. 2]

3. Algorithmen zum Zeichnen von Bäumen

Im Folgenden werden drei verschiedene Algorithmen zum Zeichnen von Bäumen im Ebenen-Layout vorgestellt. Wie in Kapitel 1 bereits angekündigt, wird jeweils auf den Ablauf, einer eigenen Implementierung in Java und auf die Vor- und Nachteile eingegangen. Ziel wird es sein, Verständnis für diese Algorithmen zu schaffen.

3.1. Naiver Algorithmus von Wetherell und Shannon

Das Paper „Tidy Drawings of Trees“ von Charles Wetherell und Alfred Shannon aus dem Jahre 1979, welches im IEEE Trans. Softw. Eng. erschienen ist, handelt von verschiedenen Algorithmen zum Zeichnen von Bäumen [2]. Der erste Algorithmus, der von den beiden Autoren beschrieben und vorgestellt wird, ist ein naiver Algorithmus zum Zeichnen von Bäumen. Dieser Algorithmus soll dabei zwei Anforderungen erfüllen. Die erste Anforderung wird dabei an die Ästhetik des gezeichneten Baumes gestellt.

> *Aesthetic 1:* Nodes of a tree at the same height should lie along a straight line, and the straight lines defining the levels should be parallel [2, S. 515].

Alle Knoten, die dieselbe Höhe haben, sollen sich auf einer horizontalen Linie befinden. Jede Höhe hat dabei eine Linie, auf welcher sich die Knoten befinden sollen und diese Linien sollen alle parallel zueinander sein. Außerdem soll der Algorithmus beim Zeichnen eines Baumes ein physikalisches Limit einhalten:

> *Physical limit:* Tree drawings should occupy as little width as possible (the height of a tree drawing is fixed by the tree itself) [2, S. 515].

Das bedeutet, dass der Algorithmus möglichst schmale Bäume zeichnen soll. Jedoch wird die Höhe des Baumes durch diese Anforderungen nicht eingeschränkt. Stattdessen ist die Höhe durch den Baum vorgegeben [2, S. 515].

Die Naivität des Algorithmus besteht in seiner Simplizität. Dieser Algorithmus soll nur die beiden oberen Anforderungen erfüllen, während die anderen Algorithmen, neben den beiden oberen, noch weitere Anforderungen erfüllen müssen.

3.1.1. Ablauf

Ziel dieses Algorithmus ist es, jedem Knoten eine genaue X- und Y-Koordinate in Abhängigkeit ihrer Beziehung zu ihrem Vor- und Nachgängern zuzuordnen. Dafür muss eine Datenstruktur definiert werden, welche dies ermöglicht. Jeder Knoten muss sowohl seinen Vater als auch eventuelle Kinder kennen. Zudem muss jeder Knoten in der Lage sein, seine X-Koordinate sowie seine Höhe und Y-Koordinate zu speichern. Da jeder Knoten nun seine Beziehungen kennt, kann über dem Baum traversiert werden. Zudem kann jeder Knoten nun so seine Koordinaten speichern.

Dieser Algorithmus besitzt zwei Eingabeparameter: Die Wurzel und die Höhe des Baumes. Die Wurzel muss hierbei vom Typ der zuvor definierten Struktur sein. Zu Beginn wird eine Variable definiert: Ein Array (später Positions-Array genannt), welches die jeweils nächste freie X-Position einer Ebene des Baums beinhaltet. Hiernach wird über die Baumstruktur der Wurzel, in der Pre-Order-Traversierung, traversiert. Nun werden die X- und Y-Attribute der Knoten wie folgt bestimmt und gesetzt:

Der derzeitige Knoten bekommt als X-Position den Wert aus dem Positions-Array, in Abhängigkeit von seiner Höhe im Baum. Danach wird die Zahl im Positions-Array inkrementiert. Die Y-Position des Knoten wird nun in Abhängigkeit zur Höhe des Knoten mit der folgenden Formel berechnet:

$$y := 2 * HoeheDesKnotens + 1$$

Dieses Vorgehen wird nun für alle Knoten in dem Baum wiederholt.

Nach dem Durchlaufen aller Knoten des Baumes sind alle X- und Y-Koordinaten gesetzt und der Baum kann gezeichnet werden. Hierbei bekommen die Knoten eine feste vordefinierte Größe. Die Kanten werden dann als Linien zwischen Vätern und Kindern gezeichnet.

3.1.2. Implementierung in Java

Dieser Algorithmus wurde in Java implemeniert. Hierzu wurde die zuvor vorgestellte Datenstruktur implementiert.

```
1  class Knoten<T> {
2      private int x, y;
3      private T daten;
4      private int hoehe;
5
6      private Knoten<T> vater;
7      protected List<Knoten<T>> kinder;
8
9      public void traversPreOrder(Consumer<? super Knoten<? super T
       >> cons) { /*...*/ }
10
11     setKinder(Knoten<T>... childs) { /*...*/ }
12
13     // Getter, Setter, weitere Hilfsmethoden...
14 }
```

Quellcode 3.1.1: Vereinfachte Implementierung der Knotenklasse

Am Quellcode 3.1.1 lässt sich die implementierte Knoten-Datenstruktur betrachten. Die Klasse Knoten beinhaltet genau einen Vater, eine Liste von Kindern, da jeder Knoten mehr als ein Kind haben kann, sowie die Höhe und die Koordinaten. Zusätzlich beinhaltet diese Klasse auch noch die Möglichkeit jedem Knoten Daten mitzugeben. Für das Verständnis des Algorithmus ist dies aber nicht notwendig und wird im Folgenden nicht mehr berücksichtigt. Ferner wurden Hilfsmethoden, zum Beispiel „traversPreOrder“, implementiert. Nachdem diese Klasse erstellt wurde, kann der Algorithmus als Prozedur implementiert werden. Hierzu wurde eine Prozedur namens „algorithmus1“ erstellt, die zwei Parameter besitzt: Der Wurzel-Knoten und die Höhe des Baums. Die weitere Implementierung wird, wie zuvor beschrieben, durchgeführt. Eine mögliche Implementierung kann wie folgt aussehen:

```java
public static <T> void algorithmus1(Knoten<T> wurzel, int
    maximaleHoehe) {
    // Initialisierung des Positions-Array
    int[] nextX = new int[maximaleHoehe];
    Arrays.fill(nextX, 1);

    // Ueber den Baum traversieren und X/Y-Koordinate setzen
    wurzel.traversPreOrder(knoten -> {
        knoten.setX(nextX[knoten.getHoehe()]);
        knoten.setY(2 * knoten.getHoehe() + 1);

        // Setzen der naechst-freien Position
        nextX[knoten.getHoehe()] += 1;
    });
}
```

Quellcode 3.1.2: Implementierung des naiven Algorithmus

Die Implementierung dieses Algorithmuses folgt dem zuvor beschriebenen Ablauf. So kann am Quellcode 3.1.2 betrachtet werden, dass zu Beginn der Prozedur das Positions-Array, hier „nextX“ benannt, mit Einsen befüllt wird. Folgend wird nun über die Knoten-Datenstruktur in der Pre-Order traversiert. Dafür wurde die Knoten-Instanz-Methode namens „traversPreOrder“, mit einem Consumer als Parameter, aufgerufen. In dem übergebenen Consumer, hier als Lamda-Ausdruck, wird die X- als auch die Y-Koordinate bestimmt. Ferner wird die Ebenenspezifische nächste X-Position im Positions-Array um eins inkrementiert.

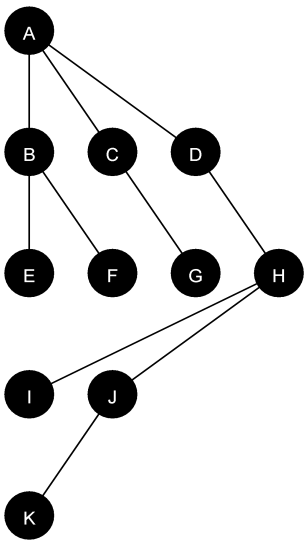

Abbildung 3.1.: Gezeichneter Baum durch den ersten Algorithmus

3.1.3. Vor- und Nachteile

Die Abbildung 3.1 zeigt einen Baum, welcher von unserer Java-Implementierung des naiven Algorithmus von Wetherell und Shannon gezeichnet wurde. Auf dem ersten Blick wird deutlich, dass dieser gezeichnete Baum maximal schmal ist, da jeder Knoten so weit links wie möglich steht. Diese Eigenschaft wird aber nur auf Kosten der Übersichtlichkeit erfüllt. Bei der Beziehung zwischen H als Vater und I und J als Kind wird deutlich, dass der von diesem Algorithmus gezeichnete Baum unübersichtlich ist. Außerdem sagt die Position der Knoten nichts über die Daten aus, welche die Knoten beinhalten könnten. Egal ob die Daten, welche I und J beinhalten, größer oder kleiner als die Daten in H wären, sie würden trotzdem an der selben Position stehen. Deshalb definieren Wetherell und Shannon weitere Anforderungen, die Algorithmen zum Zeichnen von Bäumen erfüllen müssen.

3.2. Verbesserter Algorithmus von Wetherell und Shannon

Wetherell und Shannon stellen in Ihrem Paper einen weiteren, verbesserten Algorithmus zum Zeichnen von Bäumen vor, welcher jedoch ausschließlich Binärbäume zeichnen kann [2]. Dieser Algorithmus weist die Nachteile des naiven Algorithmus nicht mehr auf. Dafür definieren sie zwei weitere Anforderungen, die der Algorithmus erfüllen soll. Jene Anforderungen sind speziell für die von ihm produzierten Darstellungen von Binärbäume.

> *Aesthetic 2:* In a binary tree, each left son should be positioned left of its father and each right son right of its father [2, S. 517].

In einem Binärbaum hat jeder Knoten maximal ein linkes und maximal ein rechtes Kind. Daher soll jedes linke Kind links vom Vater und jedes rechte Kind rechts vom Vater positioniert werden. Zudem soll jeder Vater zentriert über seinen Kindern stehen. Dieses Verhalten legen Wetherell und Shannon in einer weiteren Anforderung fest.

> *Aesthetic 3:* A parent should be centered over its children [2, S. 518].

Im Folgenden wird direkt die modifizierte Version des verbesserten Algorithmus betrachtet. Dafür wird die zweite While-Schleife des Programmcodes mit der Fig. 9 [2, A modification of Algorithm 3, S. 519] ersetzt.

3.2.1. Ablauf

Dieser Algorithmus lässt sich in zwei Phasen unterteilen. In der ersten Phase wird die vorläufige X-Koordinate der einzelnen Knoten bestimmt. In der zweiten Phase werden diese X-Koordinaten bei Bedarf nochmals abgeändert sowie die Y-Koordinate berechnet.

Die Signatur des Algorithmus ist dieselbe wie im naiven Algorithmus. Zu Beginn werden zwei ganzzahlige Arrays, ein Positions-Array (beinhaltet die nächste freie x-Koordinate auf einer bestimmten Höhe) und ein Modifikator-Array (beinhaltet notwendigen Versatz zwischen den Knoten auf einer Höhe), definiert. Diese besitzen

die Länge „Höhe des Baumes“. Hiernach müssen alle Elemente des Positions-Array mit eins und alle Elemente des Modifikator-Array mit null initialisiert werden.

Durch eine Post-Order-Traversierung werden die vorläufigen x-Koordinaten der Knoten bestimmt. Dabei wird zwischen vier verschiedenen Fällen unterschieden:

1. Der Knoten ist ein Blatt
2. Der Knoten besitzt kein linkes Kind
3. Der Knoten besitzt kein rechtes Kind
4. Der Knoten besitzt sowohl linkes als auch rechtes Kind

Im ersten Fall bekommt der Knoten die nächste freie X-Koordinate auf seiner Höhe. Im zweiten Fall wird der Knoten links, mit einem Versatz von eins, von seinem rechten Kind positioniert. Im dritten Fall wird dieser rechts, mit einem Versatz von eins, von seinem linken Kind positioniert. Im letzten Fall wird der Knoten in der Mitte zwischen seinen Kindern positioniert. Hierbei wird die folgende Formel genutzt:

$$x = (left.x + right.x)/2.$$

Bei dem zweiten und vierten Fall kann es passieren, dass ein Knoten links von der eigentlich nächsten freien X-Koordinate platziert wird. Tritt dies ein, so wird das Modifikator-Array angepasst, indem der nötige Versatz auf der Ebene des Knoten auf die Differenz zwischen der vorläufigen X-Koordinate und der eigentlich nächsten freien X-Koordinate gesetzt wird. Außerdem merkt sich jeder Knoten den Modifikator auf seiner Höhe, was im zweiten Teil des Algorithmus benötigt wird. Sollte dieser Knoten zudem kein Blatt sein, dann wird seine vorläufige X-Koordinate nochmal abgeändert. Diese wird dann mit dem Versatz aus dem Modifikator-Array auf seiner Höhe addiert, also nach rechts verschoben. Zum Schluss wird die nächste freie X-Koordinate auf der Ebene bestimmt. Dafür wird die X-Koordinate des Knoten um einen bestimmten Wert addiert (hier zwei) und in das Positions-Array (Index: Höhe des Knotens) geschrieben.

In der zweiten Phase wird eine In-Order-Traversierung über dem Baum ausgeführt. Da diese vorgibt, erst den linken Teilbaum, dann den Knoten und zum Schluss

den rechten Teilbaum zu besuchen, wird zunächst, beginnend von der Wurzel aus, der am weitesten links unten stehende Knoten besucht. Abbildung 3.2 zeigt einen Baum, welcher mit dem verbesserten Algorithmus gezeichnet wurde. In diesem Beispiel wird bei der Wurzel gestartet und nach Knoten D gelaufen. Dort angekommen, wird die X-Koordinate entweder auf die nächste freie X-Koordinate auf der Höhe des Knotens gesetzt oder auf die vorläufige X-Koordinate addiert mit den Modifikatoren der Vorgänger, welche in der ersten Phase für jeden Knoten bestimmt worden. Dabei wird der kleinere der beiden möglichen Werte genutzt. Dies wird gemacht, um sicherzustellen, dass sich der Knoten möglichst weit links befindet und somit die Anforderung an das physikalische Limit erfüllt wird. Sollte ein Knoten nun links von seinem linken Kind bzw. rechts von seinem rechten Kind stehen, dann wird der Knoten so verschoben, dass dieser rechts von seinem linken Kind bzw. links von seinem rechten Kind steht. Dadurch wird garantiert, dass jedes linke Kind links und jedes rechte Kind rechts von seinem Vater steht. Nun ist die X-Koordinate des Knotens final bestimmt worden. Die Y-Koordinate wird genau wie im naiven Algorithmus bestimmt, indem die Höhe des Knotens mit zwei multipliziert und mit einem Versatz (hier eins) addiert wird. Zum Schluss wird die nächste freie X-Position auf der Höhe des Knotens bestimmt, indem die X-Koordinate mit einem Versatz (hier zwei) addiert wird. Dies wird für alle Knoten durchgeführt, bis am Ende jeder Knoten seine finalen Koordianten bekommen hat.

3.2.2. Implementierung in Java

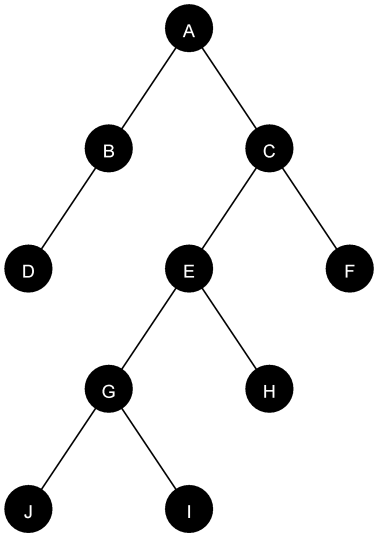

Abbildung 3.2.: Gezeichneter Baum durch den verbesserten Algorithmus

Dieser Algorithmus wurde wie der erste Algorithmus in Java implementiert. Da der gezeigte Algorithmus am Beispiel von Binärbäumen gezeigt wird, wurde eine Binäre-Knoten-Klasse implemeniert, die von der Knoten-Klasse, gezeigt im Quellcode 3.1.1, erbt. In dieser Klasse wurden Hilfsmethoden und zwei weitere Attribute definiert. Die Implementierung sieht vereinfacht wie folgt aus:

```
1  public class BinaryKnoten extends Knoten {
2      private int modifier;
3      private int vistStatus;
4
5      @Override
6      public void traversPostOrder(Consumer<Knoten> cons) {
7          // ...
8      }
9
10     public BinaryKnoten getLeft() { /*...*/ }
11     public BinaryKnoten getRight() { /*...*/ }
12
13     // Getter / Setter ...
14 }
```

Quellcode 3.2.1: Vereinfachte Implementierung der BinaryKnoten-Klasse

Als erstes wurde eine Prozedur mit dem Namen „algorithmus2Verbessert“ definiert, die zwei Eingabeparameter besitzt: der Wurzelknoten des Baums und die

Höhe des Baums. Zu Anfang werden alle Arrays und Variablen, wie im Ablauf bereits beschrieben, initialisiert. Hiernach wird erstmalig in der Post-Order über die Baum-Struktur rekursiv traversiert. Hierfür wird die in der Binären-Knoten-Klasse zuvor erstellte Methode „traversPostOrder“ genutzt. Diese übernimmt als Argument einen Consumer, in dem die einzelnen Knoten in der geforderten Reihenfolge übergeben werden. In dem übergebenem Consumer wird ferner die X-Koordinate und der Modifikator jedes einzelnen Knoten bestimmt.

```
wurzel.traversPostOrder(k -> {
    BinaryKnoten knoten = (BinaryKnoten) k;

    // X-Koordinate bestimmen
    if(istBlatt)
        place = nextPos[h];
    else if(knoten.getLeft() == null)
        place = knoten.getRight().getX() - 1;
    else if(knoten.getRight() == null)
        place = knoten.getLeft().getX() + 1;
    else
        place = (knoten.getLeft().getX() + knoten.getRight().getX
    ()) / 2;

    // setzen alle Variablen
    modifier[h] = Math.max(modifier[h], nextPos[h] - place);
    knoten.setX(place + (istBlatt ? 0 : modifier[h]));
    nextPos[h] = knoten.getX() + 2;
    knoten.setModifier(modifier[h]);
});
```

Quellcode 3.2.2: Vereinfachte Implementierung der Phase 1

Die zweite Phase wurde so implementiert, dass iterativ über den Baum in der Pre-Order traversiert wird. Die Implementierung entspricht dem zuvor gezeigten Ablauf in Kapitel 3.2.

Wird diese Implementierung auf einen beispielhaften Baum angewendet, so kann das Ergebnis in der Abbildung 3.2 betrachtet werden.

3.2.3. Vor- und Nachteile

Durch das Erfüllen der Anforderung, dass jeder Vater über seinen Kindern zentriert werden soll, kann der Algorithmus gegen die Anforderung an das physikalische Limit verstoßen. Abbildung 3.3 zeigt jenes Verhalten. Daraus schließen die beiden Autoren auf folgendes Theorem:

Theorem (Uglification): Minimum width drawings exist which violate Aesthetic 3 by arbitrary amounts [2, S. 519].

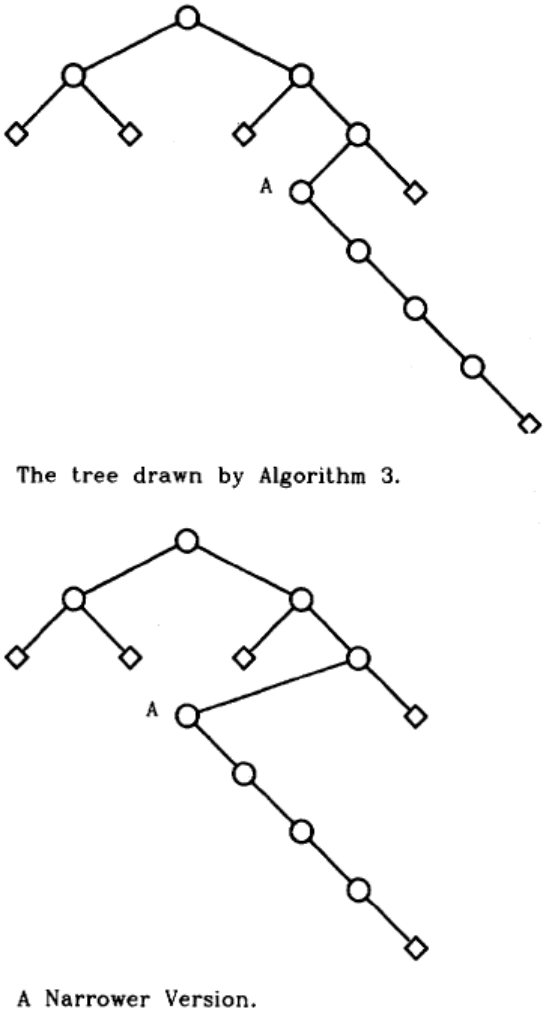

Abbildung 3.3.: Beispiel für das Theorem [2, S. 519]

Bei der schmaleren Variante des Baumes auf Abbildung 3.3 ist der Vater von Knoten A nicht zentriert über beiden Kindern und verstößt somit gegen Aesthetic 3. Die weitere Variante verstößt gegen das physikalische Limit, da der Baum nicht maximal schmal ist. Dies stellt hier einen Trade-off zwischen den beiden Anforderungen dar. Es ist somit notwendig, dass maximal schmale Bäume der Anforderungen Aesthetic 3 widersprechen können.

Im Vergleich zu dem naiven Algorithmus von Wetherell und Shannon zeichnet der verbesserte Algorithmus die Bäume nicht mehr maximal links. Dies sorgt dafür,

dass die gezeichneten Bäume übersichtlicher wirken und ästhetisch ansprechender sind. Falls die Knoten einen Inhalt haben, dann ist die Positionierung der Knoten auch von Relevanz, da das linke Kind kleiner und das rechte Kind größer als der Knoten ist. Beispielsweise Suchbäume sind dadurch auch intuitiver zu verstehen, da die Größe des Inhalts von links nach rechts aufsteigend sortiert ist.

Zudem ist der verbesserte Algorithmus typischerweise nicht in der Lage beliebige Bäume zu zeichnen, sondern ausschließlich Binärbäume.

3.3. Algorithmus von Reingold und Tilford

Das Paper „Tidier Drawings of Trees“ von Edward M. Reingold und John S. Tilford aus dem Jahre 1981, welches im IEEE Transaction on Software Engineering erschienen ist, handelt von einem Algorithmus zum Zeichnen von Bäumen im Ebenen-Layout[3]. Die Motivation der beiden Autoren für das Erstellen dieses Algorithmus beruht darauf, dass sie ein entscheidendes Problem an dem verbesserten Algorithmus von Wetherell und Shannon erkannt haben. Dieser Algorithmus ohne die Modifikation produziert Bäume, welche nicht maximal schmal sind, da das Zentrieren der Väter erzwungen wird. Die modifizierte Variante des Algorithmus produziert zwar maximal schmale Bäume, dafür können diese wesentlich unübersichtlicher sein, was der untere Baum auf Abbildung 3.3 zeigt[1]. Reingold und Tilford erkennen, dass das Problem an dem Algorithmus ist, dass die einzelnen Teilbäume von Knoten außerhalb des Teilbäume beeinflusst werden. Daraus folgern die beiden, dass es mit dem Algorithmus von Wetherell und Shannon dazu kommen kann, dass ein Baum und die Spiegelung desselben Baumes keine Spiegelbilder ergeben. Jedoch wäre es nach Reingold und Tilford wünschenswert, wenn symmetrische Bäume auch symmetrisch gezeichnet werden. Daraus wird eine weitere Anforderung an Algorithmen zum Zeichnen von Bäumen abgeleitet [3, S. 224].

> *Aesthetic 4:* A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree [1, S. 224].

Wenn der Algorithmus eine Spiegelung eines Baumes erhält und der durch den Algorithmus gezeichnete Baum ein exaktes Spiegelbild des eigentlichen Baumes ist, dann ist diese Anforderung erfüllt. Abbildung A.1 zeigt einen Baum und seine Spiegelung der mit unserer Implementierung des Algorithmus von Wetherell und Shannon gezeichnet wurde. Abbildung A.2 zeigt den selben Baum mit seiner Spiegelung, aber dieses mal mit unserer Java-Implementierung von Reingold und Tilfords Algorithmus. Zu erkennen an dieser Abbildung ist, dass der Algorithmus von Reingold und Tilford Aesthetic 4 erfüllt, der Algorithmus von Wetherell und

[1] In dieser Arbeit wurde lediglich die modifizierte Variante vorgestellt, siehe 3.2.

Shannon jedoch nicht. Außerdem sollen Teilbäume immer gleich gezeichnet werden, unabhängig von ihrer Position im Baum.

Um diese Anforderung zu erfüllen, dürfen Knoten außerhalb eines Teilbaums die Knoten innerhalb eines Teilbaums nicht beeinflussen. Damit das erreicht wird, werden bei diesem Algorithmus nicht einzelne Knoten platziert (wie bei Wetherell und Shannon), sondern es werden zwei Teilbäume unabhängig voneinander betrachtet und dann so nah wie möglich aneinander platziert [3, S. 225].

3.3.1. Ablauf

Der Algorithmus zum Zeichnen von Bäumen von Reingold und Tilford lässt sich in zwei Phasen unterteilen. Den ersten Teil stellt die Prozedur "Setup" dar. Diese Prozedur erhält vier Eingabeparameter, nämlich einen Knoten (welche am Anfang die Wurzel ist), die Höhe des Knotens im Gesamtbaum, sowie RMOST und LMOST. RMOST und LMOST sind jedoch nicht vom Datentyp Knoten, sondern von einem neuen Datentyp namens Extreme. Extreme sind Strukturen, die drei Attribute besitzen:

- Verweis auf einen Knoten
- Offset von der X-Koordinate zur Wurzel des Teilbaums
- Höhe des Knotens im Gesamtbaum

Zu Beginn der Prozedur wird geprüft, ob der übergebene Knoten NULL ist. Wenn dies nicht der Fall ist, dann wird die Y-Koordinate des Knotens auf seine Höhe gesetzt. Danach wird die Prozedur rekursiv aufgerufen, um eine Post-Order Traversierung auszuführen. Dieser Aufruf sieht wie folgt aus:

```
1     SETUP (L, LEVEL+1, LR, LL );
2     SETUP (R, LEVEL+1, RR, RL );
```

Quellcode 3.3.1: Rekursiver Aufruf von Setup

Durch die Post-Order Traversierung wird die Setup Prozedur solange aufgerufen, bis der aktuelle Knoten das Blatt unten links ist. Da ein Blatt automatisch der am weitesten links und am weitesten rechts stehende Knoten ist, werden die Adressen von RMOST und LMOST auf diesen Knoten gesetzt. Außerdem wird die Höhe von

RMOST und LMOST auf die Höhe des aktuellen Knotens gesetzt. Zudem werden die Offsets des Knotens und von RMOST sowie LMOST auf null gesetzt.

Wenn der Knoten sowohl ein linkes als auch ein rechtes Kind hat, dann wird geprüft, ob das linke Kind ein rechtes Kind hat. Ist dies der Fall, dann wird der Offset des linken Kindes auf die Summe der linken Offsets addiert und vom aktuellen Abstand abgezogen. Zudem wird dann der Zeiger auf das linke Kind auf das rechte Kind des linken Kindes gesetzt. Andernfalls, wird dann der Offset des linken Kindes von der Summe der linken Offsets subtrahiert und der aktuelle Abstand um den Offset des linken Kindes erhöht. Auch wird danach der Zeiger auf das linke Kind auf das linke Kind des linken Kindes gesetzt. Genau dasselbe wird auch für das rechte Kind geprüft und ausgeführt. Falls danach der aktuelle Abstand kleiner als ein vorher festgelegter Mindestabstand ist, wird der Abstand zwischen den Kindern um den Mindestabstand minus dem aktuellen Abstand erhöht. Außerdem wird der aktuelle Abstand dann auf den Mindestabstand gesetzt. Dies wird solange ausgeführt, bis einer der beiden Zeiger (auf linkes/rechtes Kind) gleich NULL ist. Diese Schleife sorgt dafür, dass entlang der rechten Kontur des linken Teilbaums und entlang der linken Kontur des rechten Teilbaums die Distanz zwischen den beiden bestimmt wird. Gegebenenfalls werden dann die beiden Teilbäume auseinander geschoben, falls sie sich berühren [3, S. 226].

Nach dieser Schleife wird der Offset des aktuellen Knotens auf die Hälfte des Abstandes zwischen den Kindern plus eins gesetzt. Dieser Offset wird dann von der Summe der linken Offsets abgezogen und auf die Summe der rechten Offsets addiert.

Danach werden RMOST und LMOST mit Hilfe von Der am weitesten links stehende Knoten im linken Subtree eines Knotens auf höchster Höhe (LL), Der am weitesten rechts stehende Knoten im linken Subtree eines Knotens auf höchster Höhe (LR), Der am weitesten links stehende Knoten im rechten Subtree eines Knotens auf höchster Höhe (RL) und Der am weitesten rechts stehende Knoten im rechten Subtree eines Knotens auf höchster Höhe (RR) neu festgelegt. Wenn die Höhe von RL größer als die Höhe von LL ist oder der aktuelle Knoten kein linkes Kind hat, dann setze LMOST auf RL und erhöhe den Offset von LMOST um den Offset des Knotens. Ansonsten setze LMOST auf LL und ziehe vom Offset von LMOST den

Offset des Knotens ab. Diese Abfrage wird für RMOST und LR sowie RR wiederholt. Es wird geprüft, ob die Höhe von LR größer ist als die Höhe von RR oder der Knoten kein rechtes Kind hat. Wenn dies der Fall ist, dann wird RMOST auf LR gesetzt und der Offset von RMOST um den Offset des Knotens verringert. Tritt dieser Fall nicht ein, dann wird RMOST auf RR gesetzt und der Offset von RMOST mit dem Offset des Knotens addiert.

RMOST und LMOST korrekt festzulegen ist für den nächsten Schritt wichtig und notwendig. Diese beiden Knoten in einem Subtree sind die einzigen, bei denen die Möglichkeit besteht, dass Threading angewandt wird. Threads sind eine Art „Pseudo-Kante “, welche eingefügt werden, um sicherzustellen, dass sich zwei Teilbäume nicht berühren. Jene Hilfe ist speziell für diesen Algorithmus und findet in den anderen Algorithmen keine Anwendung. Dieser Schritt ist nur dann nötig, wenn die beiden betrachteten Teilbäume eines Knotens unterschiedlich hoch sind und keiner von beiden leer ist. Reingold und Tilford geben dafür ein Beispiel an. Wenn der linke Subtree höher als der rechte Subtree ist, dann muss ein Thread von RR zu dem am weitesten rechts stehenden Knoten des linken Teilbaums auf der nächsten Höhe gelegt werden. Dieser Thread würde in dem Zeiger auf das linke Kind von RR gespeichert werden [3, S. 226].

Die zweite Phase des Algorithmus stellt die Prozedur “petrify” dar. Ziel dieser Prozedur ist es die relativen Offsets, welche in der vorherigen Phase gesetzt worden, in absolute Koordinaten umzuwandeln. Zudem sorgt die Prozedur dafür, dass die Threads gelöscht werden, da diese nicht mehr benötigt werden. Um dies zu erreichen wird eine Pre-Order-Traversierung über dem Baum durchgeführt. Dafür bekommt “petrify” zwei Eingabeparameter, nämlich einen Knoten (zu Beginn wieder die Wurzel) und eine initiale X-Position (ist beliebig). Dann prüft die Prozedur, ob der Knoten ungleich NULL ist. Wenn dies der Fall ist, dann wird geschaut, ob der Knoten einen Thread hat. Da Threads Blätter sein müssen, werden die Verweise des Knotens auf sein linkes und rechtes Kind gelöscht, um keinen Thread fälschlicherweise als Kante darzustellen. Dann setzt die Prozedur die finale X-Koordinate des Knotens auf den übergebenen Parameter. Zum Schluss wird “petrify” wie folgt rekursiv aufgerufen:

```
1     PETRIFY(T.LLINK, XPOS - T.OFFSET);
2     PETRIFY(T.RLINK, XPOS + T.OFFSET);
```

Quellcode 3.3.2: Rekursiver Aufruf von Petrify (Pre-Order)

Bei linken Kindern wird der Offset des Vaters von der X-Koordinate abgezogen. Bei rechten Kindern wird dann die X-Koordinate mit dem Offset des Vaters addiert. Ziel dessen ist, dass linke Kind links vom Vater und rechte Kinder rechts vom Vater stehen.

3.3.2. Implementierung in Java

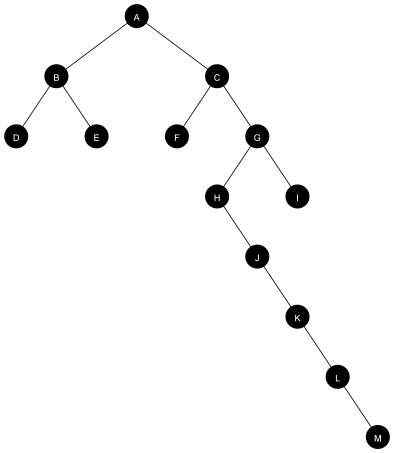

Abbildung 3.4.: Gezeichneter komplexer Baum durch den Tilford Algorithmus

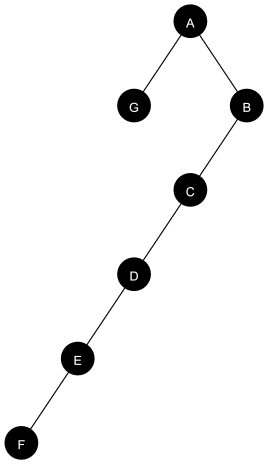

Abbildung 3.5.: Gezeichneter einfacher Baum durch den Tilford Algorithmus

Um diesen Algorithmus implementieren zu können, muss die zuvor erstellte BinaryKnoten-Klasse (siehe Quellcode 3.2.1) um ein Attribut erweitert werden: vom Typ Boolean mit dem Namen „thread“. Zudem wurde eine weitere Klasse namens „Extreme“ definiert, die wie zuvor beschrieben, implementiert wurde. Die Extreme-Klasse sieht wie folgt aus:

```
static class Extreme<T> {
    BinaryKnoten<T> knoten;
    int offset;
    int level;
}
```

Quellcode 3.3.3: Implementierung der Extreme-Klasse

Ferner wurden die zuvor beschriebenen Prozeduren „setup“ und „petrify“ implementiert. Die implementierte Prozedur „setup“ unterscheidet sich zum zuvor beschriebenen Ablauf. Sie wird nun nicht mehr rekursiv aufgerufen und besitzt nur einen Eingabeparameter, den Wurzelknoten. Hiernach wird mithilfe der „traversPostOrder“-Methode aus der BinaryKnoten-Klasse über den Baum traversiert. Diese Änderung kann in dem Codeausschnitt 3.3.4 betrachtet werden.

```
public static <T> void setup(BinaryKnoten<T> wurzel) {
    // Initialisierungen von Variablen
    // <...>
    // Ueber den Baum in der Post-Order traversieren
    wurzel.traversPostOrder(k -> {
        BinaryKnoten<T> knoten = (BinaryKnoten<T>) k;

        // Bestimmen der Y-Koordinate
        knoten.setY(2 * knoten.getHoehe() + 1);

        // Vorlaeufige relative X-Koordinate bestimmen
        // <...>
    }
}
```

Quellcode 3.3.4: Ausschnitt aus der setup-Prozedur

Abweichend zum Ablauf entspricht die Y-Koordinate nicht der Höhe des Knotens. Stattdessen wird diese wie im Ablauf aus dem Kapitel 3.1 berechnet. Dies bietet den Vorteil, dass die Methodik zum Zeichnen der Bäume nicht verändert werden muss. Die weitere Implementierung folgt der Beschreibung aus dem Ablauf.

Die Implementierung der Prozedur „petrify“ entspricht der Beschreibung aus dem Ablauf.

```java
public static <T> void algorithmus3(BinaryKnoten<T> wurzel) {
    // Aufrufen der beiden Algorithmus-Spezifischen Prozeduren
    setup(wurzel);
    petrify(wurzel, 0);

    // Ermitteln des kleinsten x-Wertes...
    // minX := kleinster X-Wert;

    final int offset = Math.abs(minX) + 1;

    // Verschieben aller Knoten nach rechts
    wurzel.traversPreOrder(x -> x.setX(x.getX() + offset));
}
```

Quellcode 3.3.5: Ausschnitt aus der algorithmus3-Prozedur

Hiernach wurde die Prozedur „algorithmus3“ definiert, die im Codeausschnitt 3.3.5 gezeigt wird. Diese ruft zu Beginn die beiden Prozeduren, „setup“ und „petrify“ auf. Danach müssen die X-Koordinaten noch angepasst werden, da diese negativ sein können. Hierfür wird der kleinste X-Wert ermittelt, dessen absoluter Wert addiert mit eins in der Variable „offset“ gespeichert wird. Nun werden alle X-Koordinaten des Baums mit dem Wert aus „offset“ addiert.

Zwei beispielhafte Ergebnisse können in den Abbildungen 3.4 und 3.5 betrachtet werden.

3.3.3. Vor- und Nachteile

Ein Algorithmus, welcher Aesthetic 4 erfüllt, kann Bäume produzieren, welche nicht maximal schmal sind. Für Reingold und Tilford ist das Erfüllen dieser Anforderung jedoch wichtiger als das Einhalten des physikalischen Limits, da Aesthetic 4 dafür sorgt, dass diese Bäume für Menschen übersichtlicher und leichter zu verstehen sind [3, S. 224]. Dafür ist der Algorithmus jedoch, gemessen an den Zeilen im Programmcode, der längste. Selbst bei komplexeren Bäumen, wie in Abbildung 3.5, wird ein übersichtlicher und ästhetisch ansprechender Baum gezeichnet.

Ähnlich wie der verbesserte Algorithmus von Wetherell und Shannon ist der Algorithmus von Reingold und Tilford so wie er ist nur in Lage Binärbäume zu zeichnen. Jedoch beschreiben die beiden Autoren wie der Algorithmus modifiziert werden kann um beliebige Bäume zu zeichnen.

4. Vergleich

Es wurden drei verschiedene Algorithmen zum Zeichnen von Bäumen im Ebenen-Layout vorgestellt, erklärt und in Java implemeniert. Ein Beispiel ist im Anhang durch die Abbildungen A.3, A.4 (Algorithmus von Wetherhell und Shannon ohne Modifizierung), A.5, A.6 gegeben. Diese zeigen allesamt den gleichen Baum, jedoch gezeichnet mit den verschiedenen Algorithmen.

Der erste Algorithmus liefert den schmalsten Baum im Vergleich zu den anderen Algorithmen. Dafür werden die Knoten aber maximal weit nach links platziert, was dafür sorgt, dass der Baum unübersichtlich wirkt und dass die Beziehung zwischen linkem und rechten Kind verloren geht. Wird der Knoten C von der Abbildung A.3 und seine Kinder betrachtet, dann wirkt es so, als hätte C zwei rechte Kinder.

Beim zweiten Algorithmus ohne die Modifikation wird ein ästhetisch ansprechender Baum erzeugt. Jedoch ist der Baum, wie Abbildung A.4 zeigt, nicht maximal schmal. Außerdem produziert dieser Algorithmus nicht in jedem Fall Spiegelbilder, wie in der Abbildung A.1 deutlich wird.

Wird der zweite Algorithmus mit der Modifizierung versehen, wird ein unübersichtlicher Baum gezeichnet. Bei der Abbildung A.5 wird dies deutlich. Die Väter sind nicht über den Kindern zentriert, was bei der Beziehung A-B-C erkannt werden kann. Zudem werden lange Kanten gezeichnet, was bei der Kante zwischen den Knoten J und N deutlich wird.

Im Vergleich zu den anderen Algorithmen produziert der Algorithmus von Reingold und Tilford den ästhetisch ansprechendsten Baum, da dieser alle vier Anforderungen an die Ästhetik erfüllt. Deutlich wird das beim Betrachten der Abbildungen A.6 und A.2. Dies kommt auf Kosten des physikalischen Limits, da der Baum nicht maximal schmal ist.

5. Konklusion

Mit dieser Arbeit sollte ein Verständnis für verschiedene Algorithmen zum Zeichnen von Bäumen geschaffen werden. Dabei wurde begonnen bei einem naiven Algorithmus und hingearbeitet zu jeweils zwei verbesserten Algorithmen. Hierfür wurde besonders auf den Ablauf der einzelnen Algorithmen eingegangen. Zudem wurde für jeden Algorithmus eine mögliche Implementierung in Java dargestellt sowie beispielhaft einige Bäume mit diesen Implementierungen gezeichnet. Außerdem wurde bei jedem Algorithmus auf die spezifischen Vor- und Nachteile des jeweiligen Algorithmus eingegangen sowie ein abschließender Vergleich gezogen.

Nach der Betrachtung des naiven Algorithmus wird deutlich, dass dieser keine ästhetisch ansprechenden Bäume zeichnet. Weiterführend wurde ein verbesserter Algorithmus von WS vorgestellt. Die durch den Algorithmus gezeichneten Bäume halten sich an das physikalische Limit sowie an die ersten beiden Anforderungen an die Ästhetik. Jedoch priorisiert dieser Algorithmus die Anforderung an das physikalische Limit, was zur Folge hat, dass ästhetisch unansprechende Bäume gezeichnet werden. Ferner besitzt dieser Algortihmus nicht die Möglichkeit Spiegelbilder von Bäumen korrekt zu zeichnen. Der Algorithmus von TR greift dies auf und stellt eine weitere Möglichkeit zum Zeichnen von Bäumen dar. Dieser erfüllt alle Anforderungen an die Ästhetik von gezeichneten Bäumen, jedoch teilweise auf Kosten des physikalischen Limits.

Literaturverzeichnis

[1] M.Niklaus, *Bäume in der Informatik*. EducETH, 2007.

[2] C. Wetherell and A. Shannon, "Tidy drawing of trees," *IEEE Trans. Softw. Eng*, vol. SE-5(5), pp. 514–520, 1979.

[3] E. Reingold and J. Tilford, "Tidier drawing of trees," *IEEE Trans. Softw. Eng*, vol. SE-7(2), pp. 223–228, 1981.

A. Anhang A

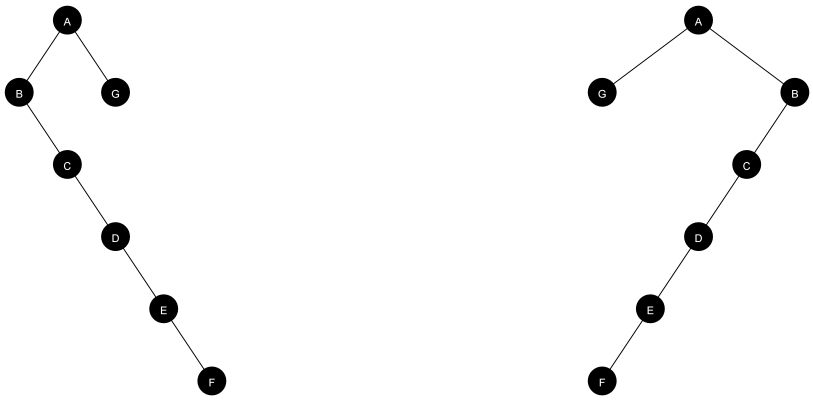

Abbildung A.1.: Baum und Spiegelung gezeichnet nach WS

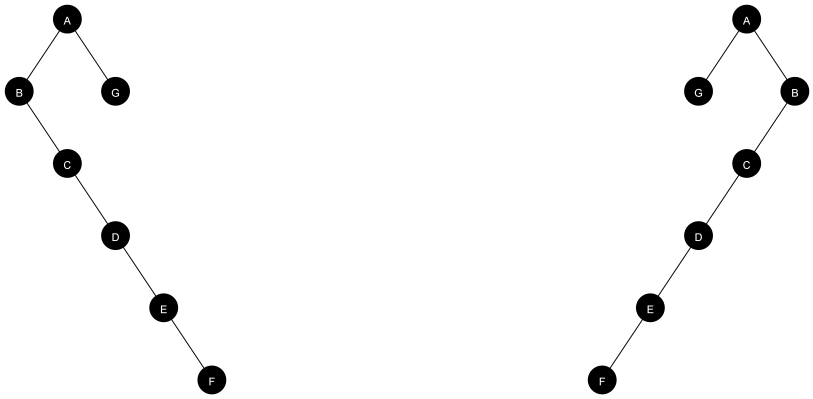

Abbildung A.2.: Baum und Spiegelung gezeichnet nach TR

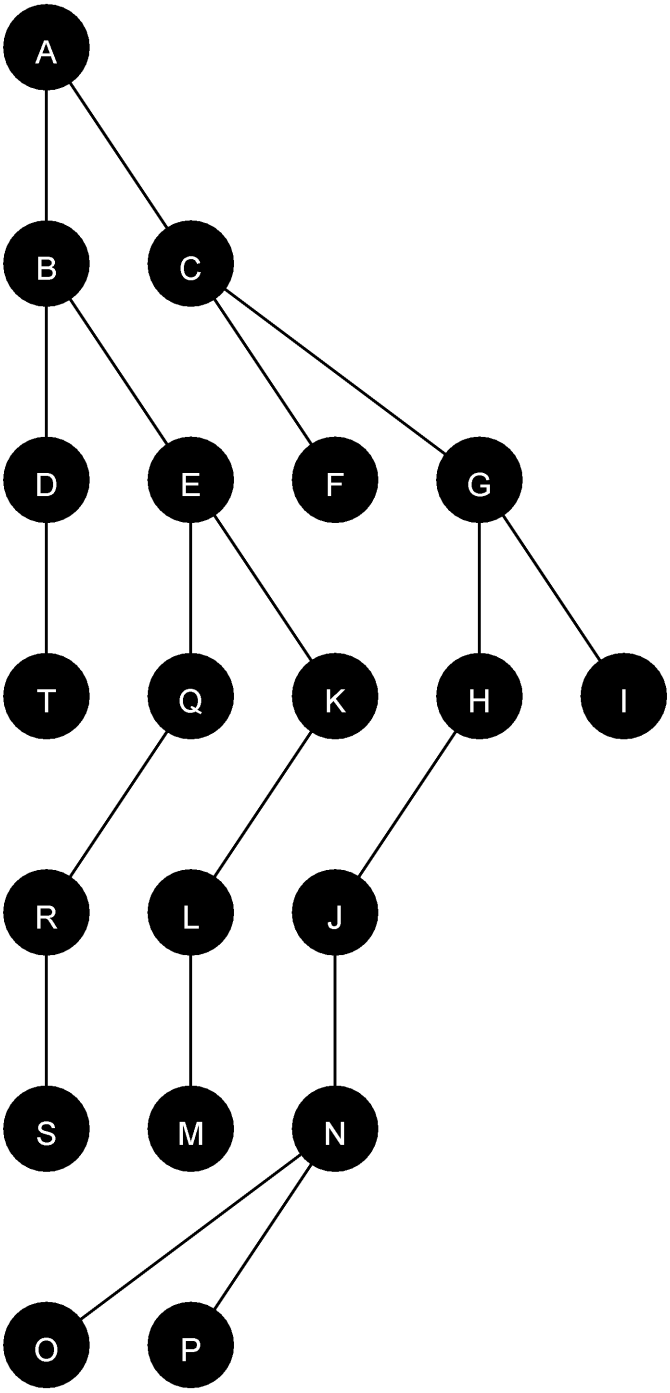

Abbildung A.3.: Komplexer Baum gezeichnet vom naiven WS

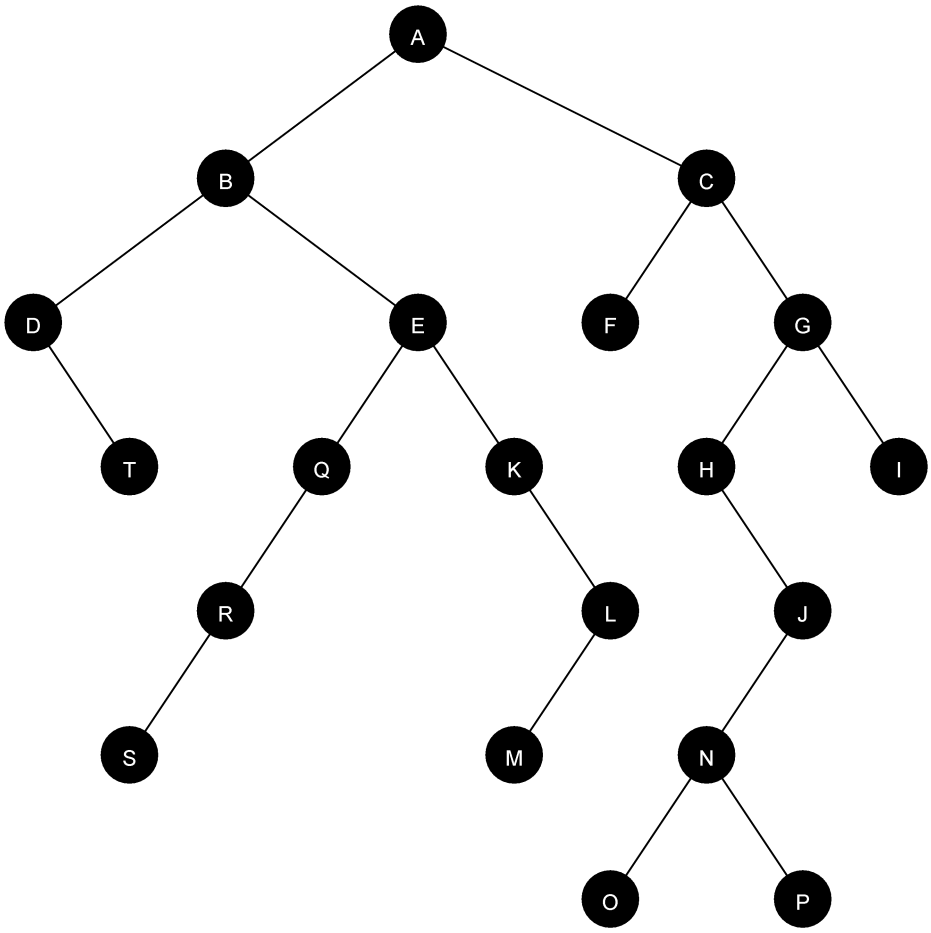

Abbildung A.4.: Komplexer Baum gezeichnet vom WS

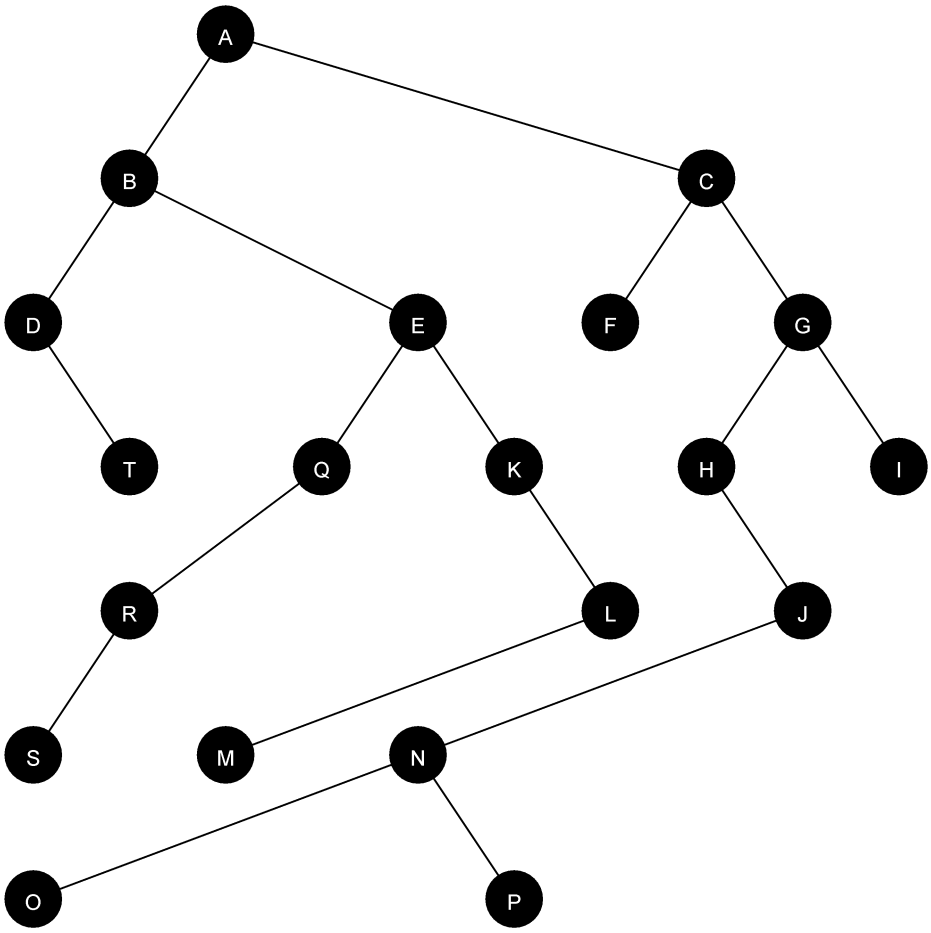

Abbildung A.5.: Komplexer Baum gezeichnet vom modifizierten WS

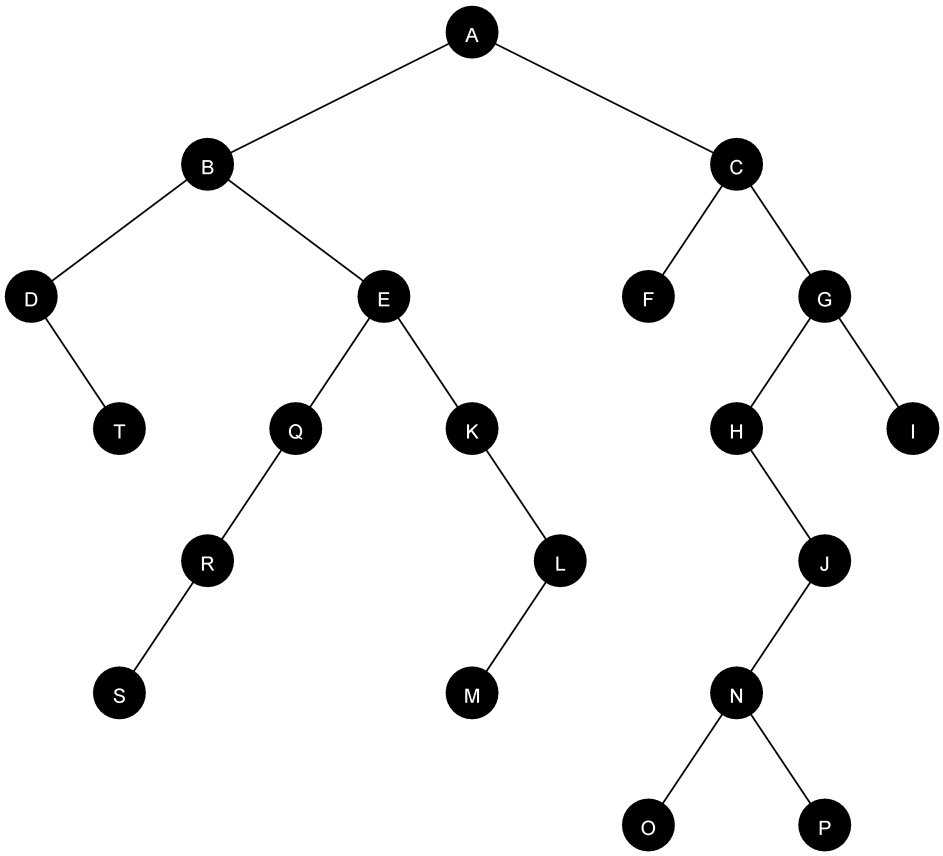

Abbildung A.6.: Komplexer Baum gezeichnet vom TR

B. Anhang B

```java
package algos;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.stream.Stream;

public class Knoten<T> {

    private int x, y;
    private T daten;
    private int hoehe;

    private Knoten<T> vater;
    protected List<Knoten<T>> kinder;

    protected Knoten(T data) {
        this.daten = data;
    }

    public void traversPreOrder(Consumer<? super Knoten<? super T
    >> cons) {
        cons.accept(this);

        if(kinder != null)
            for(Knoten<? super T> a : kinder)
                if(a != null)
                    a.traversPreOrder(cons);
    }

    public void traversPostOrder(Consumer<? super Knoten<? super T
    >> cons) {
        if(kinder != null)
            for(Knoten<? super T> a : kinder)
                if(a != null)
                    a.traversPreOrder(cons);

        cons.accept(this);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public T getData() {
```

```
48          return daten;
49      }
50
51      public Knoten<T> getVater() {
52          return vater;
53      }
54
55      public void setX(int x) {
56          this.x = x;
57      }
58
59      public void setY(int y) {
60          this.y = y;
61      }
62
63      public int getHoehe() {
64          return hoehe;
65      }
66
67      public void setHoehe(int hoehe) {
68          this.hoehe = hoehe;
69      }
70
71      public void setFather(Knoten<T> knoten) {
72          this.vater = knoten;
73      }
74
75      public List<Knoten<T>> getKinder() {
76          return kinder;
77      }
78
79      /**
80      * Setzt die Kinder des Knoten. Hierbei wird der
81      * Vater und die Hoehe im Gesamtbaum fuer die uebergebenen
        Kinder
82      * gesetzt.
83      *
84      * @param childs Kinder des Knoten
85      */
86      public void setKinder(@SuppressWarnings("unchecked") Knoten<T
        >... childs) {
87          this.kinder = Arrays.asList(childs);
88
89          Stream.of(childs).forEach(x -> {
90              if(x == null) return;
91
92              x.setHoehe(this.hoehe + 1);
93              x.setFather(this);
94          });
95      }
96
97      @Override
98      public String toString() {
99          return getData() + " [" + getX() + ", " + getY() + "]";
100     }
101 }
```

Quellcode B.0.1: Knoten-Klasse

```java
package algos;

import java.util.ArrayList;
import java.util.function.Consumer;

public class BinaryKnoten<T> extends Knoten<T> {

    private int modifier;
    private int vistStatus;
    private boolean thread;

    protected BinaryKnoten(T data) {
        super(data);

        this.kinder = new ArrayList<Knoten<T>>(2);
        for(int i = 0; i < 2; i++)
            this.kinder.add(i, null);
    }

    @Override
    public void traversPostOrder(Consumer<? super Knoten<? super T
    >> cons) {
        if(getLeft() != null)
            getLeft().traversPostOrder(cons);
        if(getRight() != null)
            getRight().traversPostOrder(cons);

        cons.accept(this);
    }

    public void traversInOrder(Consumer<? super Knoten<? super T>>
     cons) {
        if(getLeft() != null)
            getLeft().traversInOrder(cons);

        cons.accept(this);

        if(getRight() != null)
            getRight().traversInOrder(cons);
    }

    /**
     * Setzt die Kinder des Knoten. Hierbei wird der
     * Vater und die Hoehe im Gesamtbaum fuer die uebergebenen
    Kinder
     * gesetzt. Werden mehr als zwei Kinder uebergeben, so werden
     * diese ignoriert.
     *
     * Wird nur ein Kind uebergeben, so wird das rechte auf Null
    gesetzt
     *
     * Linkes Kind = childs[0]
     * Rechtes Kind = childs[1]
     *
     * @param childs Kinder des Knoten
     */
    @SuppressWarnings("unchecked")
    @Override
```

```
55      public void setKinder(Knoten<T>... childs) {
56          for(int i = 0; i < Math.min(2, childs.length); i++) {
57              super.kinder.set(i, childs[i]);
58
59              if(childs[i] != null) {
60                  childs[i].setFather(this);
61                  childs[i].setHoehe(getHoehe() + 1);
62              }
63          }
64      }
65
66      public BinaryKnoten<T> getLeft() {
67          if(getKinder() == null)
68              return null;
69
70          return getKinder().size() >= 1 ?  (BinaryKnoten<T>)
    getKinder().get(0) : null;
71      }
72
73      public BinaryKnoten<T> getRight() {
74          if(getKinder() == null)
75              return null;
76
77          return getKinder().size() == 2 ? (BinaryKnoten<T>)
    getKinder().get(1) : null;
78      }
79
80      /**
81       * Die Hoehe des uebergebenen Knoten wird nicht ueberschrieben
82       *
83       * @param left Linkes Kind
84       */
85      public void setLeft(BinaryKnoten<T> left) {
86          super.kinder.set(0, left);
87      }
88
89      /**
90       * Die Hoehe des uebergebenen Knoten wird nicht ueberschrieben
91       *
92       * @param left Rechtes Kind
93       */
94      public void setRight(BinaryKnoten<T> right) {
95          super.kinder.set(1, right);
96      }
97
98      public void setThread(boolean thread) {
99          this.thread = thread;
100     }
101
102     public boolean isThread() {
103         return thread;
104     }
105
106     public void setModifier(int modifier) {
107         this.modifier = modifier;
108     }
109
110     public int getModifier() {
```

```
111         return modifier;
112     }
113
114     public int getVistStatus() {
115         return vistStatus;
116     }
117
118     public void setVistStatus(int vistStatus) {
119         this.vistStatus = vistStatus;
120     }
121 }
```

Quellcode B.0.2: Binary-Klasse

```
1  package algos;
2
3  import java.util.Arrays;
4
5  public class VerbesserterAlgorithmus {
6
7      public static <T> void algorithmus2Verbessert(BinaryKnoten<T>
       wurzel, int hoehe) {
8          // Initialisierung des Position / Modifikator (offset)-
       Array
9          int[] nextPos = new int[hoehe], modifier = new int[hoehe];
10         Arrays.fill(nextPos, 1);
11
12         // ueber den Baum traverieren
13         wurzel.traversPostOrder(k -> {
14             @SuppressWarnings("unchecked")
15             BinaryKnoten<T> knoten = (BinaryKnoten<T>) k;
16
17             // Grundlegende Variablen initialisieren
18             boolean isLeaf = knoten.getLeft() == null && knoten.
       getRight() == null;
19             int place;
20             int h = knoten.getHoehe();
21
22             // Bestimmen der vorruebergehenden X-Koordinate
23             if(isLeaf)
24                 place = nextPos[h];
25             else if(knoten.getLeft() == null)
26                 place = knoten.getRight().getX() - 1;
27             else if(knoten.getRight() == null)
28                 place = knoten.getLeft().getX() + 1;
29             else
30                 // Bei zwei Kindern: zentrieren zwischen denen
31                 place = (knoten.getLeft().getX() + knoten.getRight
       ().getX()) / 2;
32
33             // setzen des Ebenen-Offsets
34             modifier[h] = Math.max(modifier[h], nextPos[h] - place
       );
35
36             // X-Koordinate berechnen
37             knoten.setX(place + (isLeaf ? 0 : modifier[h]));
38
39             // Setzen des naechsten freien Platzes auf der Ebenene
```

```
40              nextPos[h] = knoten.getX() + 2;
41
42              // Setzen des Knoten-Offsets
43              knoten.setModifier(modifier[h]);
44          });
45
46          int modifierSum = 0;
47
48          Arrays.fill(nextPos, 1);
49
50          // 0: first_visit
51          // 1: left_visit
52          // 2: right_visit
53          final int first_visit = 0;
54          final int left_visit = 1;
55          final int right_visit = 2;
56
57          BinaryKnoten<T> current = wurzel;
58          current.setVistStatus(0);
59
60          // ueber den Baum traversieren in der Post-Order
61          while(current != null) {
62              if(current.getVistStatus() == first_visit) { //first
63                  modifierSum += current.getModifier();
64                  current.setVistStatus(left_visit);
65
66                  if(current.getLeft() != null) {
67                      current = current.getLeft();
68                      current.setVistStatus(first_visit);
69                  }
70              }
71              else if(current.getVistStatus() == left_visit) { //
    left
72                  // Berechnen der X-Koordinate
73                  current.setX(Math.min(
74                      nextPos[current.getHoehe()],
75                      current.getX() + modifierSum - current.
    getModifier()
76                  ));
77
78                  if(current.getLeft() != null) {
79                      current.setX(Math.max(current.getX(), current.
    getLeft().getX() + 1));
80                  }
81
82                  if(current.getVater() != null) {
83                      BinaryKnoten<T> father = (BinaryKnoten<T>)
    current.getVater();
84
85                      if(father.getVistStatus() == right_visit) {
86                          current.setX(Math.max(current.getX(),
    father.getX() + 1));
87                      }
88                  }
89
90                  nextPos[current.getHoehe()] = current.getX() + 2;
91                  current.setY(2 * current.getHoehe() + 1);
92                  current.setVistStatus(2);
```

```
93
94                  if(current.getRight() != null) {
95                      current = current.getRight();
96                      current.setVistStatus(first_visit);
97                  }
98              }
99              else { // right
100                 modifierSum -= current.getModifier();
101                 current = (BinaryKnoten<T>) current.getVater();
102             }
103         }
104     }
105
106 }
```

Quellcode B.0.3: WS-Algorithmus-Klasse

```
1  package algos;
2
3  public class TilfordAlgorithmus {
4
5      private static final int MIN_SEPERATION = 2;
6
7      /**
8       * Definition nach dem Beschriebenen Ablauf
9       */
10     private static class Extreme<T> {
11         BinaryKnoten<T> knoten;
12         int offset;
13         int level;
14
15         void set(BinaryKnoten<T> k, int offset) {
16             this.knoten = k;
17             this.level = k.getHoehe();
18             this.offset = offset;
19         }
20     }
21
22     public static <T> void setup(BinaryKnoten<T> wurzel) {
23         // Extreme definieren
24         final Extreme<T>
25             LL = new Extreme<T>(),
26             LR = new Extreme<T>(),
27             RL = new Extreme<T>(),
28             RR = new Extreme<T>();
29
30         // ueber den Baum traversieren
31         wurzel.traversPostOrder(k -> {
32             @SuppressWarnings("unchecked")
33             BinaryKnoten<T> knoten = (BinaryKnoten<T>) k;
34
35             // setzen der Y-Koordinate (Orginal gesetzt durch
    rekursives Aufrufen der Funktion
36             knoten.setY(knoten.getHoehe() * 2 + 1);
37
38             BinaryKnoten<T>
39                     L = knoten.getLeft(),
40                     R = knoten.getRight();
```

```
41
42              // Setzen -> siehe naechste IF
43              Extreme<T>
44                  RMOST = null,
45                  LMOST = null;
46
47              // setzen von RMOST und LMOST
48              if(knoten.getVater() != null) { // gilt fuer alle
    ausser der WURZEL
49                  boolean isLeftSubtree = ((BinaryKnoten<T>) knoten.
    getVater()).getLeft() == knoten;
50                  if(!isLeftSubtree) { // Hier wird LR = RMOST, LL =
     LMOST
51                      RMOST = LR;
52                      LMOST = LL;
53                  }
54                  else { //Hier wird RR = RMOST, RL = LMOST
55                      RMOST = RR;
56                      LMOST = RL;
57                  }
58              }
59
60              // teste ob es sich um ein Blatt handelt
61              if(L == null && R == null) {
62                  // setzen von RMOST / LMOST
63                  RMOST.set(knoten, 0);
64                  LMOST.set(knoten, 0);
65                  knoten.setModifier(0); // modifier = offset
66              }
67              else {
68                  int currentSeperation = MIN_SEPERATION;     //
    CURSEP
69                  int currentNodeSeperation = MIN_SEPERATION; //
    ROOTSEP
70                  int offsetToL = 0;                          //
    LOFFSUM
71                  int offsetToR = 0;                          //
    ROFFSUM
72
73                  // Falls sich die Subtrees ueberschneiden, so
    werden diese ausseinander geschoben
74                  // -> der Offset wird vergroessert (relative
    position zum Vater)
75                  // -> setzen der neuen Offsets aller Knoten in den
     Subtrees
76                  while(L != null && R != null) {
77                      if(currentSeperation < MIN_SEPERATION) {
78                          currentNodeSeperation =
    currentNodeSeperation + (MIN_SEPERATION - currentSeperation);
79                          currentSeperation = MIN_SEPERATION;
80                      }
81
82                      if(L.getRight() != null) {
83                          offsetToL += L.getModifier();
84                          currentSeperation -= L.getModifier();
85                          L = L.getRight();
86                      }
87                      else {
```

```
88                          offsetToL -= L.getModifier();
89                          currentSeperation += L.getModifier();
90                          L = L.getLeft();
91                      }
92
93                      if(R.getLeft() != null) {
94                          offsetToR -= R.getModifier();
95                          currentSeperation -= R.getModifier();
96                          R = R.getLeft();
97                      }
98                      else {
99                          offsetToR += R.getModifier();
100                         currentSeperation += R.getModifier();
101                         R = R.getRight();
102                     }
103                 } // END WHILE
104
105                 // Setzen des Offsets des Knoten -> Er wird
    hierbei zentriert
106                 // zwischen seine Kinder gesetzt
107                 knoten.setModifier((currentNodeSeperation + 1) /
    2);
108                 offsetToL -= knoten.getModifier();
109                 offsetToR += knoten.getModifier();
110
111                 // Da sich nun auch LL, LR, RL, RR verschoben
    haben, muessen die
112                 // Information aktualisiert werden
113                 if(RL.level > LL.level || knoten.getLeft() == null
    ) {
114                     LMOST = RL;
115                     LMOST.offset += knoten.getModifier();
116                 }
117                 else {
118                     LMOST = LL;
119                     LMOST.offset -= knoten.getModifier();
120                 }
121
122                 if(LR.level > RR.level || knoten.getRight() ==
    null) {
123                     RMOST = LR;
124                     RMOST.offset -= knoten.getModifier();
125                 }
126                 else {
127                     RMOST = RR;
128                     RMOST.offset += knoten.getModifier();
129                 }
130
131                 // Setzen der Verweise
132                 if(L != null && L != knoten.getLeft()) {
133                     RR.knoten.setThread(true);
134                     RR.knoten.setModifier(Math.abs(RR.offset +
    knoten.getModifier() - offsetToL));
135                     if(offsetToL - knoten.getModifier() <= RR.
    offset)
136                         RR.knoten.setLeft(L);
137                     else
138                         RR.knoten.setRight(R);
```

```
139                 }
140                 else if(R != null && R != knoten.getRight()) {
141                     LL.knoten.setThread(true);
142                     LL.knoten.setModifier(Math.abs(LL.offset -
    knoten.getModifier() - offsetToR));
143                     if(offsetToR + knoten.getModifier() >= LL.
    offset)
144                         LL.knoten.setRight(R);
145                     else
146                         LL.knoten.setLeft(R);
147                 }
148             }
149         });
150     }
151
152     /**
153      * Traversiert in der Pre-Order ueber den Gegeben Baum (Knoten
    ).
154      * Hierbei wird diese Funktion Rekursiv aufgerufen.
155      *
156      * Hierbei werden die durch setup erstellten Verzweise
157      * geloescht.
158      *
159      * @param knoten Knoten (initial der Wurzel-Knoten)
160      * @param xpos X-Position des Knoten (Initial = 0)
161      */
162     public static <T> void petrify(BinaryKnoten<T> knoten, int
    xpos) {
163         if(knoten != null) {
164             // Setzen der X-Position
165             knoten.setX(xpos);
166
167             // loeschen der Verweise
168             if(knoten.isThread()) {
169                 knoten.setThread(false);
170                 knoten.setRight(null);
171                 knoten.setLeft(null);
172             }
173
174             // Rekursiver Aufruf der Prozedur mit dem Linken
175             // bzw. rechtem Kind
176             petrify(knoten.getLeft(), xpos - knoten.getModifier())
    ;
177             petrify(knoten.getRight(), xpos + knoten.getModifier()
    );
178         }
179     }
180
181     public static <T> void algorithmus3(BinaryKnoten<T> wurzel) {
182         setup(wurzel);
183         petrify(wurzel, 0);
184
185         int[] minX = { Integer.MAX_VALUE };
186         wurzel.traversPostOrder(k -> {
187             if(k.getX() < minX[0])
188                 minX[0] = k.getX();
189         });
190
```

```
191             final int offset = Math.abs(minX[0]) + 1;
192             wurzel.traversPreOrder(x -> x.setX(x.getX()+ offset));
193         }
194
195 }
```

Quellcode B.0.4: RT-Algorithmus-Klasse