



Fakultät Informatik

Erkan Garan, 70467533
Justin Treulieb, 70468597

Bäume zeichnen

im Ebenen-Layout

Betreuer:
J. Eckert

Salzgitter

Suderburg

Wolfsburg

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

Wolfenbüttel, den 28. April 2022

Kurzfassung

Bäume sind eine besondere Form von Graphen. Sie stellen eine Datenstruktur dar und bestehen aus zwei Elementen, den Knoten und Kanten. Hier wird sich mit drei verschiedenen Algorithmen beschäftigt und diese 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 Späterem 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 with a specific example. Finally the advantages and disadvantages, which will be described beforehand, will be shown on that particular example.

Inhaltsverzeichnis

Abkürzungsverzeichnis	VI
1. Einleitung	1
1.1. Motivation	1
1.2. Zielsetzung	1
2. Bäume in der Informatik	2
2.1. Definition von Bäumen	2
2.2. Anwendungsgebiete	3
3. Algorithmen zum Zeichnen von Bäumen	5
3.1. Naiver Algorithmus von Wetherell und Shannon	5
3.1.1. Ablauf	6
3.1.2. Implementierung in Java	6
3.1.3. Vor- und Nachteile	8
3.2. Verbesselter Algorithmus von Wetherell und Shannon	9
3.2.1. Ablauf	9
3.2.2. Implementierung in Java	12
3.2.3. Vor- und Nachteile	13
3.3. Algorithmus von Reingold und Tilford	16
3.3.1. Ablauf	17
3.3.2. Implementierung in Java	21
3.3.3. Vor- und Nachteile	23
4. Konklusion	24
Literaturverzeichnis	25
A. Anhang A	26
B. Anhang B	31

Abbildungsverzeichnis

2.1. Einfacher beschrifteter Binärbaum	2
2.2. Ein Stammbaum, Beispiel für Bäume als Datenstrukturen [1]	4
3.1. Gezeichneter Baum durch den ersten Algorithmus	8
3.2. Gezeichneter Baum durch den verbesserten Algorithmus	12
3.3. Beispiel für das Theorem [2]	14
3.4. Beispielhafte Bestimmung für LL, LR, RL, RR	17
3.5. Gezeichneter komplexer Baum durch den Tilford Algorithmus	21
3.6. Gezeichneter einfacher Baum durch den Tilford Algorithmus	21
A.3. Komplexer Baum gezeichnet von naiven Wetherell und Shannon (WS)	27
A.4. Komplexer Baum gezeichnet von WS	28
A.5. Komplexer Baum gezeichnet von verbessertem WS	29
A.6. Komplexer Baum gezeichnet von Tilford und Reingold (TR)	30

Quellcodeverzeichnis

3.1.1. Vereinfachte Implementierung der Knotenklasse	6
3.1.2. Implementierung des naiven Algorithmus	7
3.2.1. Vereinfachte Implementierung der BinaryKnoten-Klasse	12
3.2.2. Vereinfachte Implementierung der Phase 1	13
3.3.1. Rekursiver Aufruf von Setup	18
3.3.2. Rekursiver Aufruf von Petrify (Pre-Order)	20
3.3.3. Implementierung der Extreme-Klasse	21
3.3.4. Ausschnitt aus der setup-Prozedur	22
 B.0.1 Knoten-Klasse	31
B.0.2. Binary-Klasse	32
B.0.3. WS-Naiver-Algorithmus-Klasse	35
B.0.4. WS-Algorithmus-Klasse	35
B.0.5. RT-Algorithmus-Klasse	38
B.0.6. Zusatzklasse: Trees	42
B.0.7. Zusatzklasse: Drawer	44
B.0.8. Zusatzklasse: Mains	46

Abkürzungsverzeichnis

WS Wetherell und Shannon

TR Tilford und Reingold

 **NULL** Zustand für das Fehlen eines Wertes

1. Einleitung

Garan/Treulieb

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.

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]. 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. Ferner 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

Garan

Nach Wetherell und Shannon sind Bäume endliche, gerichtete, zusammenhängende, azyklische Graphen [2]. Die Besonderheit an einem azyklischen Graph ist, dass dieser keine Zyklen enthält. Bäume bestehen im Grunde genommen 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. Eine weitere besondere Form von Knoten sind die Blätter. Diese verfügen nämlich über keine Kinder [1]

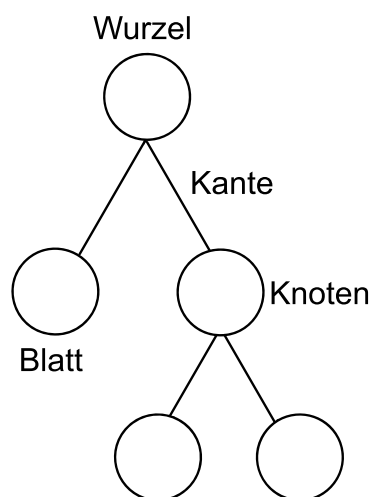


Abbildung 2.1.: Einfacher schrifteter Binärbaum


Eine spezielle Form von Bäumen stellen die sogenannten Binärbäume dar. Binärbäume sind Bäume, wo jeder Knoten maximal zwei Kinder hat. Abbildung 2.1 zeigt einen einfachen Binärbaum, wo jedes Element genau beschriftet ist.

Um jeden Knoten eines Baumes abarbeiten zu können, kann über dem Baum traversiert werden. Traversierung bezeichnet das systematische Ablaufen von jedem Knoten eines Baumes. Der naive Algorithmus von Wetherell und Shannon verwendet 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 also vor den Kindern durchlaufen. Sowohl der verbesserte Algorithmus von Wetherell und Shannon als auch der Algorithmus von Reingold und Tilford verwenden hingegen die Post-Order-Traversierung. Dabei wird als erstes der linke Teilbaum, dann der rechte Teilbaum und dann der Knoten besucht [1]. Bei dieser Art der Traversierung werden die Kinder also vor den Väter durchlaufen. Neben diesen Arten der Traversierung gibt es noch weitere Möglichkeiten, wie über einem Baum traversiert werden kann. Diese sind für das Verständnis der hier vorgestellten Algorithmen aber nicht notwendig.

2.2. Anwendungsgebiete

Treulieb

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 dazu verwendet werden, die Syntax von Quellcode zu überprüfen. Ferner werden diese Bäume als Syntaxbäume bezeichnet. 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 Blätter stellen Kategorien  und die Knoten Bedingungen. So

können beispielsweise neue Datensätze, in Abhängigkeit zu seinen Werten, einer bestimmten Kategorie zugeordnet werden.

Auch außerhalb der Informatik werden Bäume häufig verwendet. Sie können dazu genutzt werden um zum Beispiel eine Hierarchie eines Unternehmens darzustellen oder einen Stammbaum einer Familie wie in der Abbildung 2.2 [1].

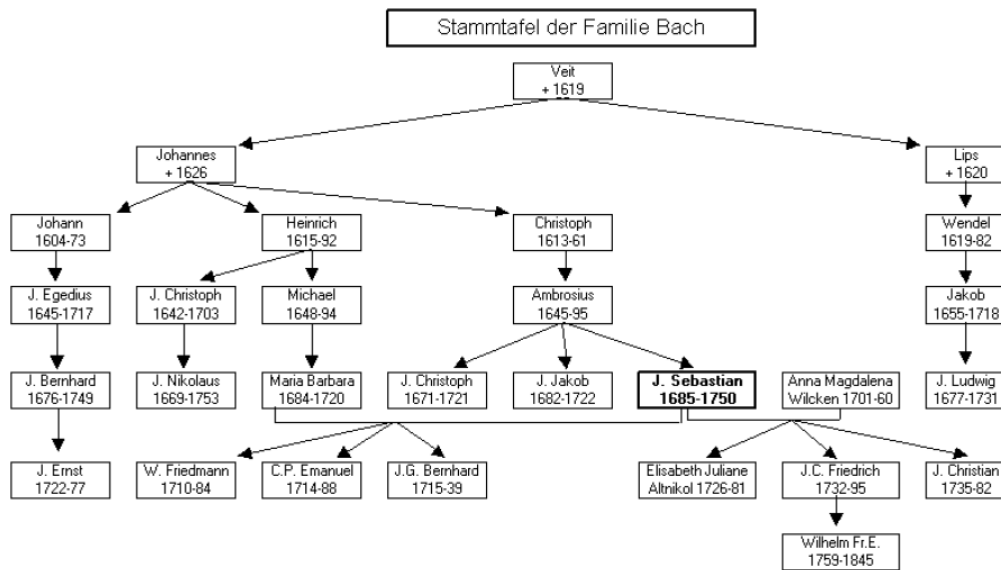


Abbildung 2.2.: Ein Stammbaum, Beispiel für Bäume als Datenstrukturen [1]

3. Algorithmen zum Zeichnen von Bäumen





3.1. Naiver Algorithmus von Wetherell und Shannon




Garan

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. 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].

 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].

Das bedeutet, dass der Algorithmus möglichst schmale Bäume zeichnen soll. Jedoch wird die Höhe des Baumes durch diese Anforderungen nicht eingeschränkt. Stattdessen beimmt der Baum selbst seine Höhe [2].

3.1.1. Ablauf

Treulieb


Bevor die Funktionsweise des Algorithmus beschrieben werden kann, muss die Baumstruktur wie folgt definiert sein: Sie benötigt eine Struktur die symbolisch für ein Knoten des Baums steht. Diese Knoten-Struktur muss hierbei ihren Vater kennen, auf ihre Kinder zugreifen können sowie ihre Position und Höhe im Baum speichern können.

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:


$$y := 2 * \text{HoeheDesKnotens} + 1$$

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. 

3.1.2. Implementierung in Java

Treulieb

Dieser Algorithmus wurde in Java implementiert. Hierzu wurde die zuvor vorgestellte Datenstruktur implementiert. Zusätzlich wurden Methoden hinzugefügt, die beispielsweise ein einfaches Traversieren über den Baum ermöglichen oder zum Setzen von Kindern. 

Diese Klasse sieht wie folgt aus:

```

1 class Knoten {
2     private Knoten father;
3     private Knoten[] childs;
4     private int hoehe;

```

```
5 private int x, y;  
6 private char data;  
7  
8 public void traversPreOrder(Consumer<Knoten> cons) { /*...*/ }  
9  
10 public void setChilds(Knoten... childs) { /*...*/ }  
11  
12 // Getter, Setter, weitere Hilfsmethoden...  
13 }
```

Quellcode 3.1.1: Vereinfachte Implementierung der Knotenklasse

Nachdem diese Klasse implementiert 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:

```
1 public static void algorithmus1(Knoten wurzel, int maximaleHoehe)  
2 {  
3     int[] nextX = new int[maximaleHoehe];  
4     for(int i = 0; i < nextX.length; i++)  
5         nextX[i] = 1;  
6  
7     wurzel.traversPreOrder(knoten -> {  
8         knoten.setX(nextX[knoten.getHoehe()]);  
9         knoten.setY(2 * knoten.getHoehe() + 1);  
10  
11         nextX[knoten.getHoehe()] += 1;  
12     });  
13 }
```

Quellcode 3.1.2: Implementierung des naiven Algorithmus

Wird diese Prozedur mit einer Baumstruktur aufgerufen, so werden alle X- und Y-Koordinaten gesetzt. Ein Beispiel für einen gezeichneten Baum kann in der Abbildung 3.1 betrachtet werden.

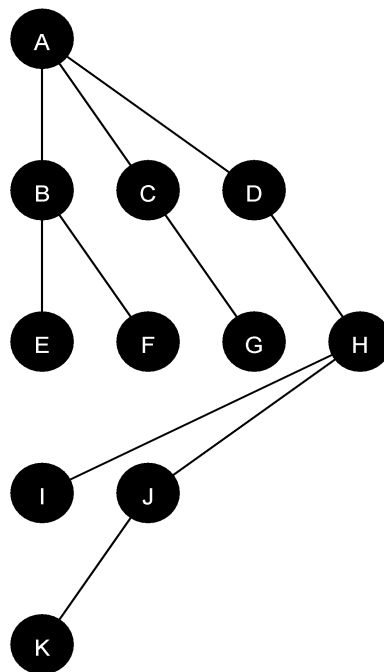


Abbildung 3.1.: Gezeichneter Baum durch den ersten Algorithmus

3.1.3. Vor- und Nachteile

Garan

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 Sohn 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 I und J größer oder kleiner als 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

Treulieb

Wetherell und Shannon stellen in Ihrem Paper einen weiteren, verbesserten Algorithmus zum Zeichnen von Bäumen vor, welcher jedoch ausschließlich Binärbäume zeichnen kann. 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 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].

In einem Binärbaum hat jeder Knoten maximal ein linkes und maximal ein rechtes Kind. Daher ist es auch logisch, dass jedes linke Kind links vom Vater und jedes rechte Kind rechts vom Vater positioniert werden soll. 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].

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] ersetzt.

3.2.1. Ablauf

Garan

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.

Dieser Algorithmus besitzt zwei Eingabeparameter, nämlich die Wurzel des Baumes und die Höhe des Baumes. Zu Beginn werden zwei ganzzahlige Arrays, ein Positions-Array und ein Modifikator-Array, 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. Zudem wird eine Variable namens "ModifikatorSumme" deklariert und mit null initialisiert.

Nun wird über die Baumstruktur in Post-Order traversiert. Dabei werden die vorläufigen X-Koordinaten der Knoten wie folgt bestimmt: Bei der Bestimmung der X-Koordinate wird zwischen vier verschiedenen Fällen unterschieden:

1. Der Knoten ist ein Blatt (besitzt weder ein linkes noch rechtes Kind)
2. Der Knoten besitzt kein linkes Kind
3. Der Knoten besitzt kein rechtes Kind
4. Der Knoten besitzt zwei Kinder (links und rechts)

Im ersten Fall entspricht die X-Koordinate dem Eintrag im Positions-Array in Abhängigkeit zu seiner Höhe. Im zweiten Fall wird der Knoten links, mit einem Offset von eins, von seinem rechten Kind positioniert. Im dritten Fall wird dieser rechts, mit einem Offset von eins, von seinem linken Sohn positioniert. Im letzten Fall wird der Knoten in der Mitte zwischen seinen Kindern positioniert. Hierbei kann die folgende Formel genutzt werden:

$$x = (\text{left}.x + \text{right}.x) / 2.$$

Hiernach wird der Modifikator bestimmt. Dafür wird der größere Wert des entweder bereits existierenden Modifikator der Ebene genommen oder wie folgt berechnet: nächste freie Position der Ebene subtrahiert der zuvor berechneten X-Koordinate. Nun wird der Wert im Position-Array am Index (Höhe des Knotens im Baum) wie folgt neu berechnet: X-Koordinate des Knoten plus eins. Zusätzlich wird der spezifische Modifikator des Knotens gesetzt (Wert aus dem Modifikator-Array am Index: Höhe des Knotens im Baum).

In der zweiten Phase wird zu Beginn eine Variable namens "ModifikatorSumme" deklariert und mit null initialisiert. Hiernach wird über die Baumstruktur in einer modifizierten Pre-Order traversiert. Die ModifikatorSumme entspricht der Summe aus allen Modifikatoren der Väter eines Knotens. Um diese Summe zu berechnen wird bei dem Besuch eines Kind-Knotens der spezifische Modifikator des Vaters auf die Summe addiert. Beim Übergang eines Kindes zurück auf den Vater wird

entsprechend der knotenspezifische Modifikator von der globalen ModifikatorSumme abgezogen. 



Da die Pre-Order Traversierung 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. Auf dem Weg dorthin werden die Modifikatoren von A und B auf die globale ModifikatorSumme addiert. Danach wird die X-Koordinate des Knotens entweder auf die nächste freie X-Koordinate auf der Höhe des Knotens gesetzt oder auf den Wert der vorläufigen X-Koordinate addiert mit der ModifikatorSumme. Dabei wird der kleinere der beiden Werte genutzt, um sicherzustellen, dass der Knoten möglichst weit links ist. Wenn der Knoten einen linken Sohn hat und die X-Koordinate des Sohnes größer ist als die X-Koordinate des Vaters, dann wird die X-Koordinate des Vaters auf die X-Koordinate des Sohnes plus eins gesetzt. Damit wird sichergestellt, dass der Knoten nicht direkt über seinem linken Sohn steht, sondern eine Position weiter rechts. Falls der Knoten nicht die Wurzel ist, der Vater des Knotens bereits besucht und abgearbeitet wurde und die X-Koordinate des Knotens kleiner als die X-Koordinate des Vaters plus eins ist, dann wird die X-Koordinate des Knotens auf die X-Koordinate des Vaters plus eins gesetzt. Ähnlich wie im Fall vorher dient diese Überprüfung dazu, den Knoten richtig zu positionieren. Hier wird dadurch verhindert, dass der Knoten genau unter seinem Vater steht. Stattdessen wird sichergestellt, dass der Knoten rechts von seinem Vater steht, welcher der rechte Sohn sein muss, da der Vater bereits besucht und positioniert wurde. 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 Offset (hier eins) addiert wird. Zum Schluss wird die nächste freie X-Position auf der Höhe des Knotens bestimmt, indem die X-Koordinate mit einem Offset (hier zwei) addiert wird.

3.2.2. Implementierung in Java

Treulieb

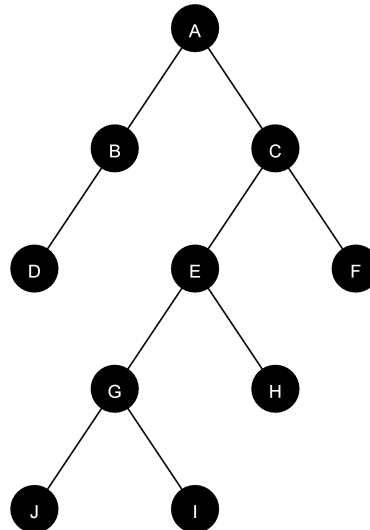


Abbildung 3.2.: Gezeichneter Baum durch den verbesserten Algorithmus

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 implementiert, die von der Knoten-Klasse, gezeigt in der Abbildung 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.

```
1 wurzel.traversPostOrder(k -> {  
2     BinaryKnoten knoten = (BinaryKnoten) k;  
3  
4     // Position / Modifikator des Knoten bestimmen ...  
5 });
```

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 3.2.

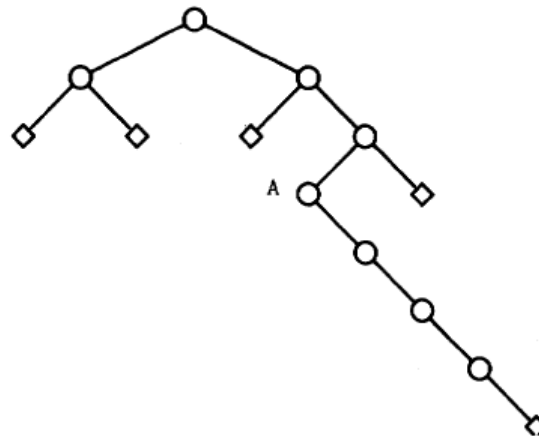
Wird diese Implementierung auf einen beispielhaften Baum angewendet, so kann das Ergebnis in der Abbildung 3.2 betrachtet werden.

3.2.3. Vor- und Nachteile

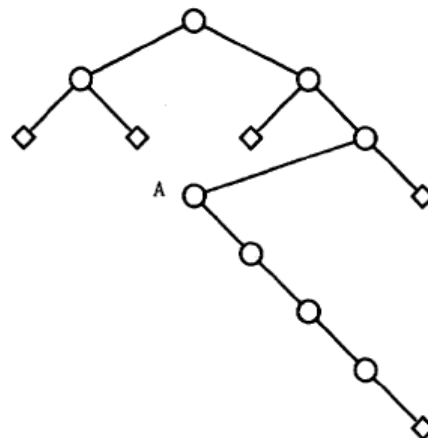
Garan

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]. 




The tree drawn by Algorithm 3.




A Narrower Version.

Abbildung 3.3.: Beispiel für das Theorem [2]

Bei der schmalen 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 möglich, 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 ein  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 nicht in der Lage beliebige Bäume zu zeichnen, sondern ausschließlich Binärbäume. Jedoch kann er durch einige geeignete Erweiterungen so modifiziert werden, dass er auch in der Lage wäre beliebige Bäume zu zeichnen. 

3.3. Algorithmus von Reingold und Tilford

Garan

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. 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. Reingold und Tilford erkennen, dass das Problem an dem Algorithmus ist, dass die einzelnen Subtrees von Knoten außerhalb des Subtrees 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].

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].



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

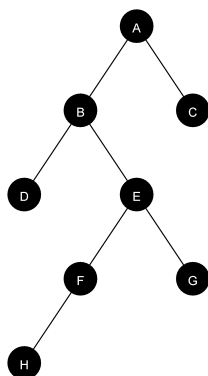
Um diese Anforderung zu erfüllen, dürfen Knoten außerhalb eines Subtrees die Knoten innerhalb eines Subtrees nicht beeinflussen. Damit das erreicht wird, werden bei diesem Algorithmus nicht einzelne Knoten platziert (wie bei Wetherell und Shannon), sondern es werden zwei Subtrees unabhängig voneinander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


3.3.1. Ablauf

Garan/Treulieb

Zum Verständnis des Algorithmus müssen einige Begriffe und Abkürzungen verbindlich erklärt werden, da diese essentiell für diesen Algorithmus sind.

- : Der am weitesten links stehende Knoten im linken Subtree eines Knotens 
- LR: Der am weitesten rechts stehende Knoten im linken Subtree eines Knotens
- RL: Der am weitesten links stehende Knoten im rechten Subtree eines Knotens
- RR: Der am weitesten rechts stehende Knoten im rechten Subtree eines Knotens



	A	B	C	D	E	F	G	H
LL	H	D		D	H	H	G	H
LR	G	D	C	D	F	H	G	H
RL	C	H	C	D	G	H	G	H
RR	C	G	C	D	G	H	G	H

 **Abbildung 3.4.:** Beispielhafte Bestimmung für LL, LR, RL, RR

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, die Höhe des Knotens im Gesamtbaum, sowie RMOST und LMOST. RMOST und LMOST sind jedoch nicht vom Datentyp Node, sondern von einem neuen Datentyp namens Extreme. Extreme sind Strukturen, die drei Attribute besitzen:

- Verweis auf einen Knoten
- Offset zur X-Koordinate zur Wurzel des Subtrees
- Höhe des Knotens im Gesamtbaum

Zu Beginn der Prozedur wird geprüft, ob der übergebene Knoten Zustand für das Fehlen eines Wertes (**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 Söhnen 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 Subtrees und entlang der linken Kontur des rechten Subtrees die Distanz zwischen den beiden bestimmt wird. Gegebenenfalls werden dann die beiden Subtrees auseinander geschoben, falls sie sich berühren [3].

Nach dieser Schleife wird der Offset des aktuellen Knotens auf die Hälfte des Abstandes zwischen den Söhnen 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 LL, LR, RL und 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  angewandt wird. Dieser Schritt ist nur dann nötig, wenn die beiden betrachteten Subtrees 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 Subtrees auf der nächsten Höhe gelegt werden. Dieser Thread würde in dem Zeiger auf das linke Kind von RR gespeichert werden. Benötigt werden diese Threads als eine Art "Pseudo-Kante" um sicherzustellen, dass diese Teilbäume sich nicht berühren und auseinander geschoben werden.

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, damit keine falschen Kanten später beim Zeichnen entstehen. Um dies zu erreichen wird eine Pre-Order-Traversierung über dem Baum durchgeführt. Dafür bekommt “petrify” zwei Eingabeparameter, nämlich einen Knoten (zu Beginn 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

Treulieb

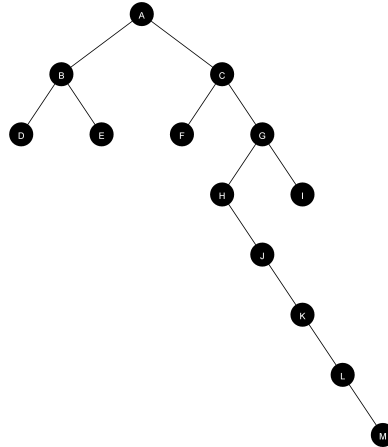


Abbildung 3.5.: Gezeichneter komplexer Baum durch den Tilford Algorithmus

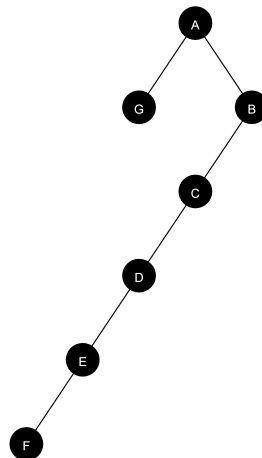


Abbildung 3.6.: Gezeichneter einfacher Baum durch den Tilford Algorithmus

Um diesen Algorithmus implementieren zu können, muss die zuvor erstellte BinaryKnoten-Klasse 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:

```

1 private static class Extreme {
2     BinaryKnoten knoten;
3     int offset;

```

```

4      int level;
5
6      void set(BinaryKnoten k, int offset) {
7          this.knoten = k;
8          this.level = k.getHoehe();
9          this.offset = offset;
10     }
11 }

```

Quellcode 3.3.3: Implementierung der Extreme-Klasse

Ferner wurden die zuvor beschriebenen Prozeduren `betup` und `"petrify"` implementiert. Die implementierte Prozedur `betup` 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.

```

1 public static void setup(BinaryKnoten wurzel) {
2     // Initialisierungen von Variablen
3     // <...>
4     // Ueber den Baum in der Post-Order traversieren
5     wurzel.traversPostOrder(k -> {
6         BinaryKnoten knoten = (BinaryKnoten) k;
7
8         // Bestimmen der Y-Koordinate
9         knoten.setY(2 * knoten.getHoehe() + 1);
10
11         // Vorläufige relative X-Koordinate bestimmen
12         // <...>
13     })
14 }

```

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.

Hiernach wurde die Prozedur `algorithmus3` definiert. Diese ruft zu Beginn die beiden Prozeduren `betup` 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 3.5 und 3.6 betrachtet werden.

3.3.3. Vor- und Nachteile

Garan

Ein Algorithmus, welcher Aesthetic 4 erfüllt, kann Bäume produzieren, welche nicht maximal schmal sind. Für Reingold und Tilford st 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]. Dafür ist der Algorithmus jedoch auch der am komplexesten zu verstehen und implementieren ist. Als Ausgleich produziert dieser Algorithmus aber auch das beste Ergebnis im Vergleich zu den beiden Algorithmen von Wetherell und Shannon. Selbst bei komplexeren Bäumen, wie in Abbildung 3.6, 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. Konklusion

Garan/Treulieb



Es wurden drei verschiedene Algorithmen zum Zeichnen von Bäumen im Ebenen-Layout vorgestellt, erklärt und in Java implementiert. Die Abbildungen A.3, A.4 (Algorithmus von Wetherhell und Shannon ohne Modifizierung), A.5, A.6 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 Reinhold 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. Jedoch ist gerade in der heutigen Zeit das Einhalten eines physikalischen Limits nicht mehr so wichtig wie in den 1980er Jahren.

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

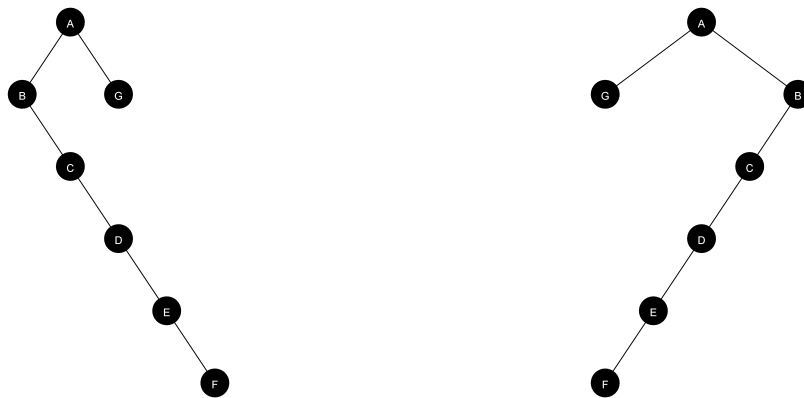


Abbildung A.1.: Baum und Spiegelung gezeichnet nach **WS**

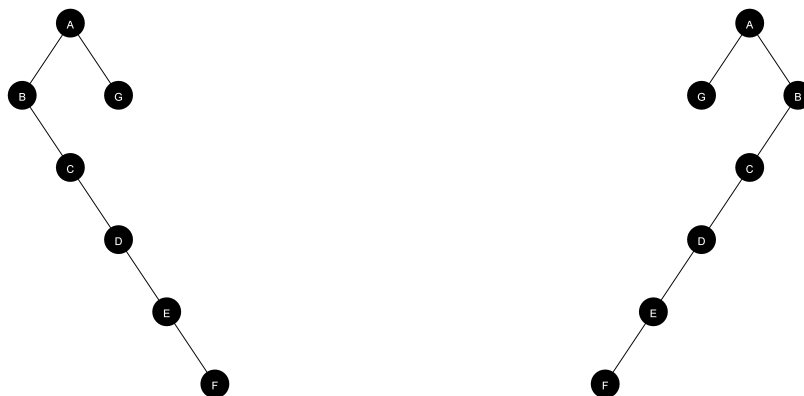


Abbildung A.2.: Baum und Spiegelung gezeichnet nach **TR**

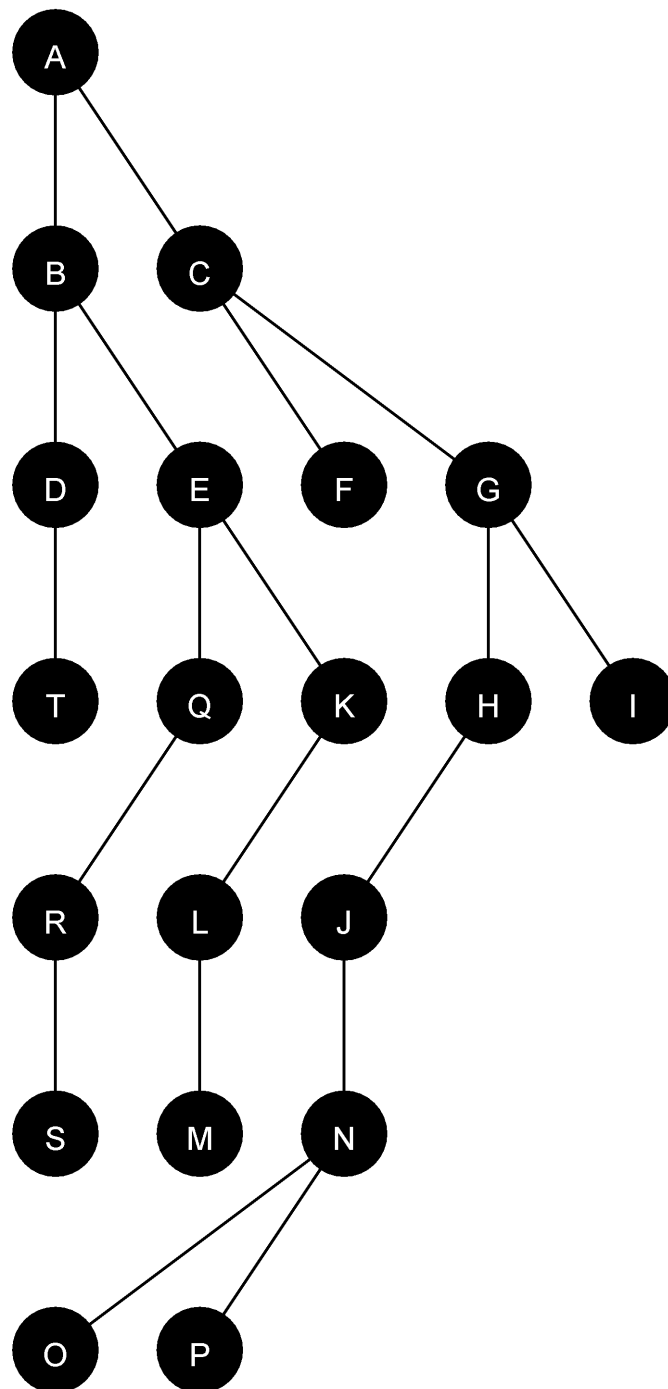


Abbildung A.3.: Komplexer Baum gezeichnet von naiven WS

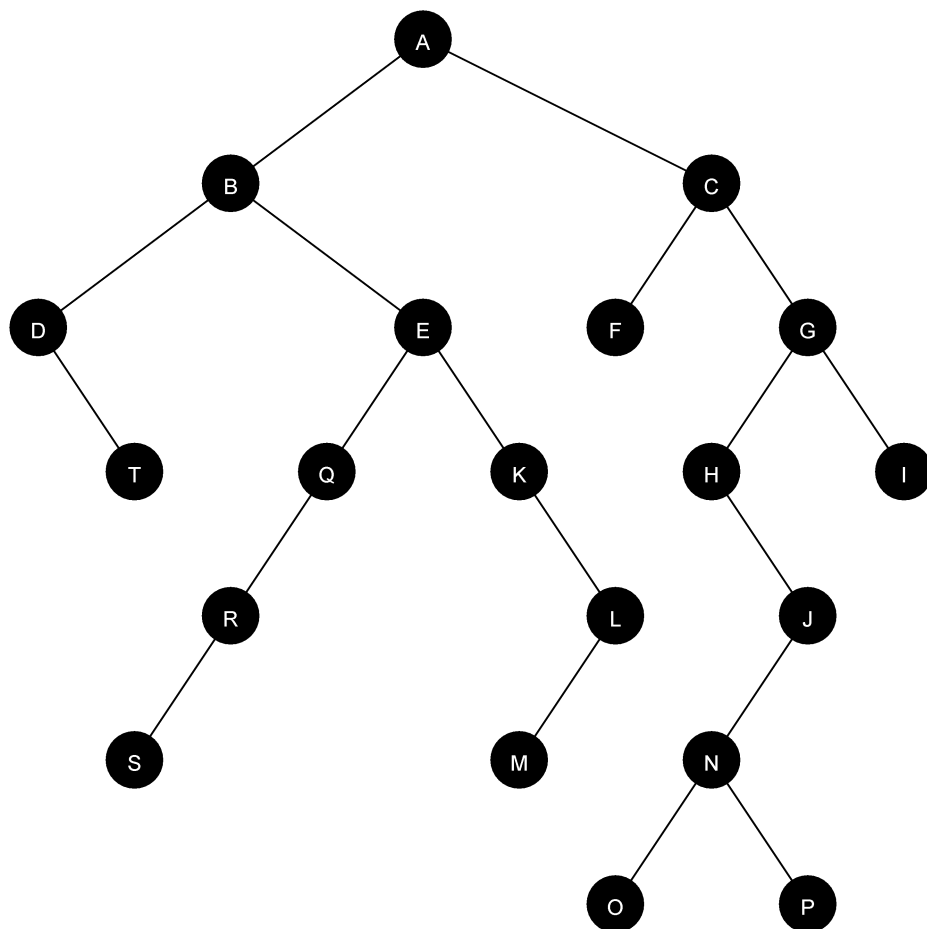


Abbildung A.4.: Komplexer Baum gezeichnet von **WS**

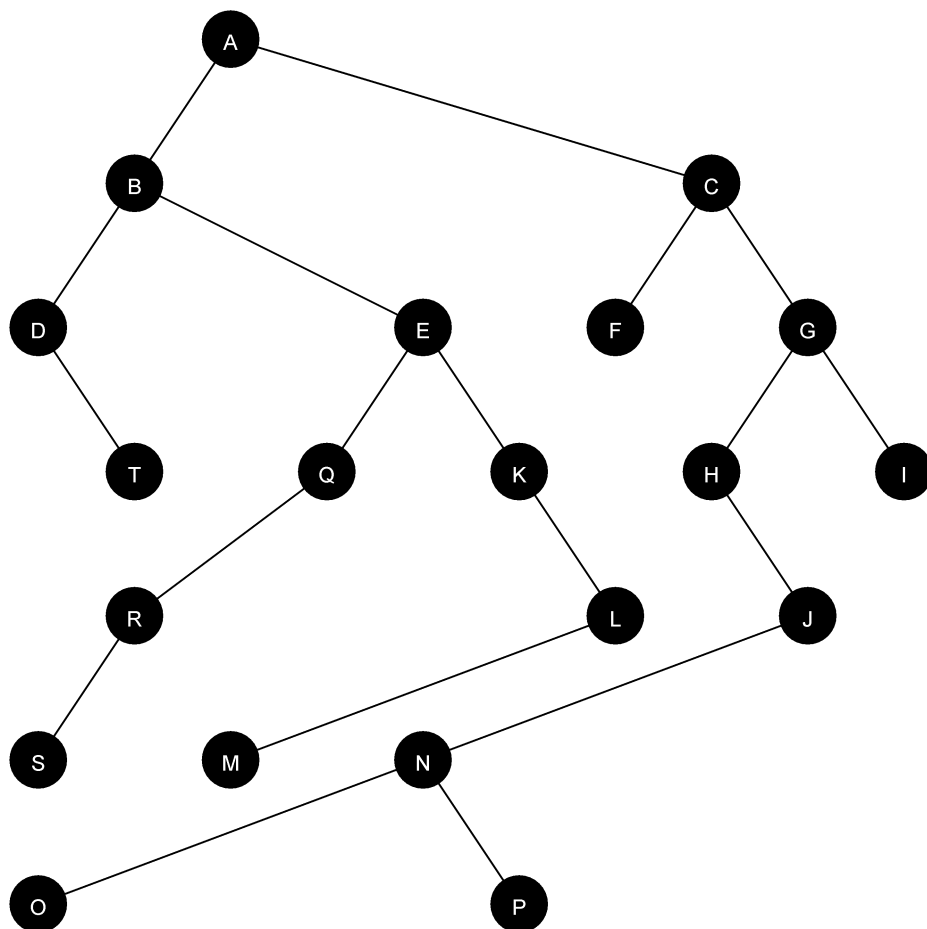


Abbildung A.5.: Komplexer Baum gezeichnet von verbessertem WS

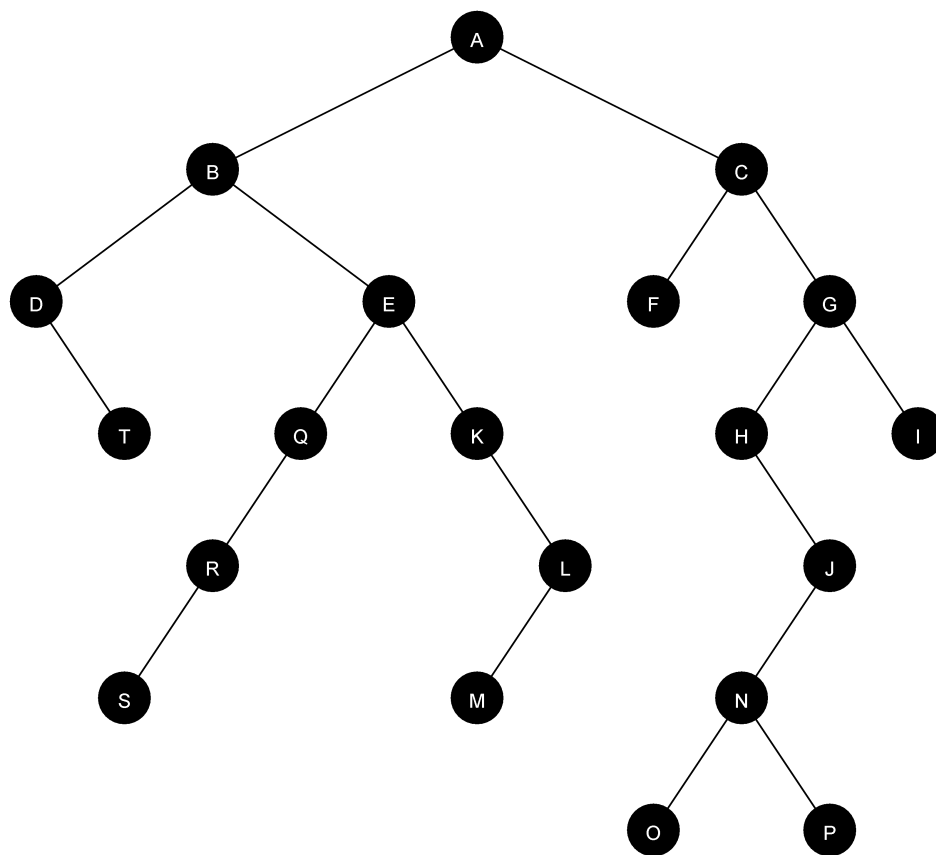


Abbildung A.6.: Komplexer Baum gezeichnet von **TR**



B. Anhang B

```
1 package algos;
2
3 import java.util.function.Consumer;
4 import java.util.stream.Stream;
5
6 public class Knoten {
7
8     private int x, y;
9     private char data;
10    private int hoehe;
11
12    private Knoten father;
13    protected Knoten[] childs;
14
15    protected Knoten(char data) {
16        this.data = data;
17    }
18
19    public void traversPreOrder(Consumer<Knoten> cons) {
20        cons.accept(this);
21
22        if(childs != null)
23            for(Knoten a : childs)
24                if(a != null)
25                    a.traversPreOrder(cons);
26    }
27
28    public void traversPostOrder(Consumer<Knoten> cons) {
29        if(childs != null)
30            for(Knoten a : childs)
31                if(a != null)
32                    a.traversPreOrder(cons);
33
34        cons.accept(this);
35    }
36
37    public int getX() {
38        return x;
39    }
40
41    public int getY() {
42        return y;
43    }
44
45    public char getData() {
46        return data;
47    }
48
49    public Knoten getFather() {
```

```

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

```

Quellcode B.0.1: Knoten-Klasse

```

1 package algos;
2
3 import java.util.function.Consumer;
4

```

```

5 public class BinaryKnoten extends Knoten {
6
7     private int modifier;
8     private int vistStatus;
9     private boolean thread;
10
11     protected BinaryKnoten(char data) {
12         super(data);
13
14         this.childs = new BinaryKnoten[2];
15     }
16
17     @Override
18     public void traversPostOrder(Consumer<Knoten> cons) {
19         if(getLeft() != null)
20             getLeft().traversPostOrder(cons);
21         if(getRight() != null)
22             getRight().traversPostOrder(cons);
23
24         cons.accept(this);
25     }
26
27     public void traversInOrder(Consumer<Knoten> cons) {
28         if(getLeft() != null)
29             getLeft().traversInOrder(cons);
30
31         cons.accept(this);
32
33         if(getRight() != null)
34             getRight().traversInOrder(cons);
35     }
36
37     /**
38      * Setzt die Kinder des Knoten. Hierbei wird der
39      * Vater und die Hoehe im Gesamtbaum fuer die uebergebenen
40      * Kinder
41      * gesetzt. Werden mehr als zwei Kinder uebergeben, so werden
42      * diese ignoriert.
43      *
44      * Wird nur ein Kind uebergeben, so wird das rechte auf Null
45      * gesetzt
46      *
47      * Linkes Kind = childs[0]
48      * Rechtes Kind = childs[1]
49      *
50      * @param childs Kinder des Knoten
51      */
52     @Override
53     public void setChilds(Knoten... childs) {
54         for(int i = 0; i < Math.min(2, childs.length); i++) {
55             super.childs[i] = childs[i];
56
57             if(childs[i] != null) {
58                 childs[i].setFather(this);
59                 childs[i].setHoehe(getHoehe() + 1);
60             }
61         }
62     }
63 }

```



```
61
62 public BinaryKnoten getLeft() {
63     if(getChilds() == null)
64         return null;
65
66     return getChilds().length >= 1 ? (BinaryKnoten) getChilds
67 () [0] : null;
68 }
69
70 public BinaryKnoten getRight() {
71     if(getChilds() == null)
72         return null;
73
74     return getChilds().length == 2 ? (BinaryKnoten) getChilds
75 () [1] : null;
76 }
77
78 /**
79  * Die Hoehe des uebergebenen Knoten wird nicht ueberschrieben
80  *
81  * @param left Linkes Kind
82  */
83 public void setLeft(BinaryKnoten left) {
84     super.childs[0] = left;
85 }
86
87 /**
88  * Die Hoehe des uebergebenen Knoten wird nicht ueberschrieben
89  *
90  * @param left Rechtes Kind
91  */
92 public void setRight(BinaryKnoten right) {
93     super.childs[1] = right;
94 }
95
96 public void setThread(boolean thread) {
97     this.thread = thread;
98 }
99
100 public boolean isThread() {
101     return thread;
102 }
103
104 public void setModifier(int modifier) {
105     this.modifier = modifier;
106 }
107
108 public int getModifier() {
109     return modifier;
110 }
111
112 public int getVistStatus() {
113     return vistStatus;
114 }
115
116 public void setVistStatus(int vistStatus) {
117     this.vistStatus = vistStatus;
118 }
```



```

27         else
28             // Bei zwei Kindern: zentrieren zwischen denen
29             place = (knoten.getLeft().getX() + knoten.getRight
().getX()) / 2;
30
31             // setzen des Ebenen-Offsets
32             modifier[h] = Math.max(modifier[h], nextPos[h] - place
);
33
34             // X-Koordinate berechnen
35             knoten.setX(place + (isLeaf ? 0 : modifier[h]));
36
37             // Setzen des naechsten freien Platzes auf der Ebenene
38             nextPos[h] = knoten.getX() + 2;
39
40             // Setzen des Knoten-Offsets
41             knoten.setModifier(modifier[h]);
42         });
43
44         BinaryKnoten current = wurzel;
45         current.setVistStatus(0);
46
47         // Summe aller Offsets (Modifikatoren) der Vaeter
48         // eines Knoten (bis hin zur Wurzel)
49         int modifierSum = 0;
50
51         // Hier wird ueber den Baum in der Pre-Order
52         // traversiert.
53         while(current != null) {
54             if(current.getVistStatus() == 0) { // erster Besuch
55                 // Berechnen der X-Koordiante des Knoten
56                 // und der Offset-Summe aller Vaeter
57                 current.setX(current.getX() + modifierSum);
58                 modifierSum += current.getModifier();
59                 current.setY(2 * current.getHoehe() + 1);
60                 current.setVistStatus(1);
61
62                 if(current.getLeft() != null) {
63                     current = current.getLeft();
64                     current.setVistStatus(0);
65                 }
66             }
67             else if(current.getVistStatus() == 1) {
68                 current.setVistStatus(2);
69                 if(current.getRight() != null) {
70                     current = current.getRight();
71                     current.setVistStatus(0);
72                 }
73             }
74             else {
75                 modifierSum -= current.getModifier();
76                 current = (BinaryKnoten) current.getFather();
77             }
78         }
79     }
80
81     public static void algorithmus2Verbessert(BinaryKnoten wurzel,
int hoehe) {

```

```

82     // Initialisierung des Position / Modifikator (offset)-
Array
83     int[] nextPos = new int[hoehe], modifier = new int[hoehe];
84     for(int i = 0; i < nextPos.length; i++)
85         nextPos[i] = 1;
86
87     // ueber den Baum traverieren
88     wurzel.traversPostOrder(k -> {
89         BinaryKnoten knoten = (BinaryKnoten) k;
90
91         // Grundlegende Variablen initialisieren
92         boolean isLeaf = knoten.getLeft() == null && knoten.
getRight() == null;
93         int place;
94         int h = knoten.getHoehe();
95
96         // Bestimmen der vorruebergelassenen X-Koordinate
97         if(isLeaf)
98             place = nextPos[h];
99         else if(knoten.getLeft() == null)
100             place = knoten.getRight().getX() - 1;
101         else if(knoten.getRight() == null)
102             place = knoten.getLeft().getX() + 1;
103         else
104             // Bei zwei Kindern: zentrieren zwischen denen
105             place = (knoten.getLeft().getX() + knoten.getRight
().getX()) / 2;
106
107         // setzen des Ebenen-Offsets
108         modifier[h] = Math.max(modifier[h], nextPos[h] - place
);
109
110         // X-Koordinate berechnen
111         knoten.setX(place + (isLeaf ? 0 : modifier[h]));
112
113         // Setzen des naechsten freien Platzes auf der Ebenene
114         nextPos[h] = knoten.getX() + 2;
115
116         // Setzen des Knoten-Offsets
117         knoten.setModifier(modifier[h]);
118     });
119
120     // Summe aller Offsets (Modifikatoren) der Vaeter
121     // eines Knoten (bis hin zur Wurzel)
122     // Hier
123     int modifierSum = 0;
124
125     for(int i = 0; i < nextPos.length; i++)
126         nextPos[i] = 1;
127
128     // 0: first_visit
129     // 1: left_visit
130     // 2: right_visit
131     final int first_visit = 0;
132     final int left_visit = 1;
133     final int right_visit = 2;
134
135     BinaryKnoten current = wurzel;

```

```

136         current.setVistStatus(0);
137
138         // ueber den Baum traversieren in der Post-Order
139         while(current != null) {
140             if(current.getVistStatus() == first_visit) { //first
141                 modifierSum += current.getModifier();
142                 current.setVistStatus(left_visit);
143
144                 if(current.getLeft() != null) {
145                     current = current.getLeft();
146                     current.setVistStatus(first_visit);
147                 }
148             }
149             else if(current.getVistStatus() == left_visit) { //
left
150                 // Berechnen der X-Koordinate
151                 current.setX(Math.min(
152                     nextPos[current.getHoehe()],
153                     current.getX() + modifierSum - current.
getModifier()
154                 ));
155
156                 if(current.getLeft() != null) {
157                     current.setX(Math.max(current.getX(), current.
getLeft().getX() + 1));
158                 }
159
160                 if(current.getFather() != null) {
161                     BinaryKnoten father = (BinaryKnoten) current.
getFather();
162
163                     if(father.getVistStatus() == right_visit) {
164                         current.setX(Math.max(current.getX(),
father.getX() + 1));
165                     }
166                 }
167
168                 nextPos[current.getHoehe()] = current.getX() + 2;
169                 current.setY(2 * current.getHoehe() + 1);
170                 current.setVistStatus(2);
171
172                 if(current.getRight() != null) {
173                     current = current.getRight();
174                     current.setVistStatus(first_visit);
175                 }
176             }
177             else { // right
178                 modifierSum -= current.getModifier();
179                 current = (BinaryKnoten) current.getFather();
180             }
181         }
182     }
183
184 }

```

Quellcode B.0.4: 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 {
11        BinaryKnoten knoten;
12        int offset;
13        int level;
14
15        void set(BinaryKnoten k, int offset) {
16            this.knoten = k;
17            this.level = k.getHoehe();
18            this.offset = offset;
19        }
20    }
21
22    public static void setup(BinaryKnoten wurzel) {
23        // Initialisieren von Hilfs/finalen Variablen
24        final int LL_I = 0, LR_I = 1, RL_I = 2, RR_I = 3;
25
26        // Extremes in ein Array speichern -> uebermoeglicht den
27        // Zugriff in der inneren Funktion (Consumer von travers-Methode)
28        Extreme[] extremes = new Extreme[] {
29            new Extreme(), // LL
30            new Extreme(), // LR
31            new Extreme(), // RL
32            new Extreme()  // RR
33        };
34
35        // ueber den Baum traversieren
36        wurzel.traversPostOrder(k -> {
37            BinaryKnoten knoten = (BinaryKnoten) k;
38
39            // setzen der Y-Koordinate (Original gesetzt durch
40            // rekursives Aufrufen der Funktion
41            knoten.setY(knoten.getHoehe() * 2 + 1);
42
43            BinaryKnoten
44                L = knoten.getLeft(),
45                R = knoten.getRight();
46
47            // setzen von LL, LR, RL, RR
48            Extreme
49                LL = extremes[LL_I],
50                LR = extremes[LR_I],
51                RL = extremes[RL_I],
52                RR = extremes[RR_I];
53
54            // Setzen -> siehe naechste IF
55            Extreme
56                RMOST = null,
57                LMOST = null;
58
59            // setzen von RMOST und LMOST

```

```

58         if(knoten.getFather() != null) { // gilt fuer alle
ausser der WURZEL
59             boolean isLeftSubtree = ((BinaryKnoten) knoten.
getFather()).getLeft() == knoten;
60             if(isLeftSubtree) { // Hier wird LR = RMOST, LL =
LMOST
61                 RMOST = LR;
62                 LMOST = LL;
63             }
64             else { //Hier wird RR = RMOST, RL = LMOST
65                 RMOST = RR;
66                 LMOST = RL;
67             }
68         }
69
70         // teste ob es sich um ein Blatt handelt
71         if(L == null && R == null) {
72             // setzen von RMOST / LMOST
73             RMOST.set(knoten, 0);
74             LMOST.set(knoten, 0);
75             knoten.setModifier(0); // modifier = offset
76         }
77         else {
78             int currentSeperation = MIN_SEPERATION; //
CURSEP
79             int currentNodeSeperation = MIN_SEPERATION; //
ROOTSEP
80             int offsetToL = 0; //
81             int offsetToR = 0; //
82             ROFFSUM
83
84             // Falls sich die Subtrees ueberschneiden, so
werden diese auseinander geschoben
85             // -> der Offset wird vergroessert (relative
position zum Vater)
86             // -> setzen der neuen Offsets aller Knoten in den
Subtrees
87             while(L != null && R != null) {
88                 if(currentSeperation < MIN_SEPERATION) {
89                     currentNodeSeperation =
currentNodeSeperation + (MIN_SEPERATION - currentSeperation);
90                     currentSeperation = MIN_SEPERATION;
91                 }
92                 if(L.getRight() != null) {
93                     offsetToL += L.getModifier();
94                     currentSeperation -= L.getModifier();
95                     L = L.getRight();
96                 }
97                 else {
98                     offsetToL -= L.getModifier();
99                     currentSeperation += L.getModifier();
100                    L = L.getLeft();
101                }
102
103                if(R.getLeft() != null) {
104                    offsetToR -= R.getModifier();

```

```

105         currentSeperation -= R.getModifier();
106         R = R.getLeft();
107     }
108     else {
109         offsetToR += R.getModifier();
110         currentSeperation += R.getModifier();
111         R = R.getRight();
112     }
113 } // END WHILE
114
115 // Setzen des Offsets des Knoten -> Er wird
hierbei zentriert
116 // zwischen seine Kinder gesetzt
117 knoten.setModifier((currentNodeSeperation + 1) /
2);
118 offsetToL -= knoten.getModifier();
119 offsetToR += knoten.getModifier();
120
121 // Da sich nun auch LL, LR, RL, RR verschoben
haben, muessen die
122 // Information aktualisiert werden
123 if(RL.level > LL.level || knoten.getLeft() == null
) {
124     LMOST = RL;
125     LMOST.offset += knoten.getModifier();
126 }
127 else {
128     LMOST = LL;
129     LMOST.offset -= knoten.getModifier();
130 }
131
132 if(LR.level > RR.level || knoten.getRight() ==
null) {
133     RMOST = LR;
134     RMOST.offset -= knoten.getModifier();
135 }
136 else {
137     RMOST = RR;
138     RMOST.offset += knoten.getModifier();
139 }
140
141 // Setzen der Verweise
142 if(L != null && L != knoten.getLeft()) {
143     RR.knoten.setThread(true);
144     RR.knoten.setModifier(Math.abs(RR.offset +
knoten.getModifier() - offsetToL));
145     if(offsetToL - knoten.getModifier() <= RR.
offset)
146         RR.knoten.setLeft(L);
147     else
148         RR.knoten.setRight(R);
149 }
150 else if(R != null && R != knoten.getRight()) {
151     LL.knoten.setThread(true);
152     LL.knoten.setModifier(Math.abs(LL.offset -
knoten.getModifier() - offsetToR));
153     if(offsetToR + knoten.getModifier() >= LL.
offset)

```



```

154         LL.knoten.setRight(R);
155     else
156         LL.knoten.setLeft(R);
157     }
158 }
159 });
160 }
161
162 /**
163  * Traversiert in der Pre-Order ueber den Gegeben Baum (Knoten
164  * ).
165  * Hierbei wird diese Funktion Rekursiv aufgerufen.
166  * Hierbei werden die durch setup erstellten Verzweisse
167  * geloescht.
168  *
169  * @param knoten Knoten (initial der Wurzel-Knoten)
170  * @param xpos X-Position des Knoten (Initial = 0)
171  */
172 public static void petrify(BinaryKnoten knoten, int xpos) {
173     if(knoten != null) {
174         // Setzen der X-Position
175         knoten.setX(xpos);
176
177         // loeschen der Verweise
178         if(knoten.isThread()) {
179             knoten.setThread(false);
180             knoten.setRight(null);
181             knoten.setLeft(null);
182         }
183
184         // Rekursiver Aufruf der Prozedur mit dem Linken
185         // bzw. rechtem Kind
186         petrify(knoten.getLeft(), xpos - knoten.getModifier())
187
188     ;
189         petrify(knoten.getRight(), xpos + knoten.getModifier())
190     };
191 }
192
193 public static void algorithmus3(BinaryKnoten wurzel) {
194     setup(wurzel);
195     petrify(wurzel, 0);
196
197     int[] minX = { Integer.MAX_VALUE };
198     wurzel.traversPostOrder(k -> {
199         if(k.getX() < minX[0])
200             minX[0] = k.getX();
201     });
202
203     final int offset = Math.abs(minX[0]) + 1;
204     wurzel.traversPreOrder(x -> x.setX(x.getX()+ offset));
205 }

```

Quellcode B.0.5: RT-Algorithmus-Klasse

```
1 package algos;
2
3 /**
4  * Diese Klasse beinhaltet Beispiel-Baeume.
5  */
6 public class Trees {
7
8     public static BinaryKnoten[] createTree(int size) {
9         return createTree(size, 'A');
10    }
11
12    public static BinaryKnoten[] createTree(int size, char
13    startChar) {
14        BinaryKnoten[] all = new BinaryKnoten[size];
15        for(int i = 0; i < all.length; i++)
16            all[i] = new BinaryKnoten((char) (startChar + i));
17
18        return all;
19    }
20
21    public static BinaryKnoten createTree1() {
22        BinaryKnoten[] all = createTree(7);
23        all[0].setChilds(all[1], all[6]);
24
25        for(int i = 1; i < 5; i++)
26            all[i].setChilds(null, all[i + 1]);
27
28        return all[0];
29    }
30
31    public static BinaryKnoten createTree2() {
32        BinaryKnoten[] tree1 = createTree(7);
33        tree1[0].setChilds(tree1[6], tree1[1]);
34
35        for(int i = 1; i < 5; i++)
36            tree1[i].setChilds(tree1[i + 1], null);
37
38        return tree1[0];
39    }
40
41    public static BinaryKnoten beispielZeigenLL_LR_RL_RR_Tree() {
42        BinaryKnoten[] all = createTree(8);
43        all[0].setChilds(all[1], all[2]);
44        all[1].setChilds(all[3], all[4]);
45        all[4].setChilds(all[5], all[6]);
46        all[5].setChilds(all[7], null);
47
48        return all[0];
49    }
50
51    public static BinaryKnoten createKomplexTree() {
52        BinaryKnoten[] all = createTree(20);
53
54        all[0].setChilds(all[1], all[2]);
55        all[1].setChilds(all[3], all[4]);
56        all[2].setChilds(all[5], all[6]);
57        all[3].setChilds(null, all[19]);
58        all[6].setChilds(all[7], all[8]);
```

```

58     all[7].setChilds(null, all[9]);
59     all[9].setChilds(all[13], null);
60
61     all[4].setChilds(all[16], all[10]);
62     all[10].setChilds(null, all[11]);
63     all[11].setChilds(all[12], null);
64
65     // index 9
66     all[13].setChilds(all[14], all[15]);
67
68     //
69     all[16].setChilds(all[17], null);
70     all[17].setChilds(all[18], null);
71
72     return all[0];
73 }
74 }

```

Quellcode B.0.6: Zusatzklasse: Trees

```

1  package algos;
2
3  import java.awt.BasicStroke;
4  import java.awt.Color;
5  import java.awt.Font;
6  import java.awt.FontMetrics;
7  import java.awt.Graphics2D;
8  import java.awt.image.BufferedImage;
9  import java.io.File;
10
11 import javax.imageio.ImageIO;
12
13 public class Drawer {
14
15     static final int scaleFactor = 30;
16     static final float scaleY = 0.75f;
17
18     private static int maxX, maxY;
19
20     /**
21      * @param root Wurzel-Knoten
22      * @param filename Datei-Name, in der das Resultat gespeichert
23      * werden soll
24      * @throws Exception - z.B. FileNotFoundException
25      */
26     public static void drawIntoImage(Knoten root, String filename)
27     throws Exception {
28         maxX = 0;
29         maxY = 0;
30
31         // Hier wird nach der maximalen Hoehe des Baums und Breite
32         geschaut
33         root.traversPreOrder(x -> {
34             if(maxX < x.getX()) maxX = x.getX();
35             if(maxY < x.getY()) maxY = x.getY();
36         });
37     }
38 }

```

```
35 // Muss um 1 addiert werden -> Sonst werden Knoten beim
zeichnen abgeschnitten
36 maxY += 1;
37 maxX += 1;
38
39 // Es wird nun skaliert, damit nicht alles zuweit
beieinander liegt
40 maxX *= 10 * scaleFactor;
41 maxY *= 10 * scaleFactor * scaleY;
42
43 // Erstellen des Bildes mit den Massen maxX und maxY
44 final BufferedImage img = new BufferedImage(maxX, maxY,
BufferedImage.TYPE_INT_ARGB);
45 Graphics2D g = img.createGraphics();
46
47 // Basiseinstellungen fuer das zeichnen angeben
48 g.setColor(Color.BLACK);
49 g.setStroke(new BasicStroke(scaleFactor / 5));
50
51 // Hier werden die Knoten und die Linien zw. den Knoten
gezeichnet
52 // Hier muss besonders auf die gegebene Skalierung
geachtet werden
53 root.traversPreOrder(knoten -> {
54     int x = (knoten.getX() * 10) * scaleFactor;
55     int y = (int) ((knoten.getY() * 10) * scaleFactor *
scaleY);
56
57     if(knoten.getFather() != null)
58         g.drawLine(x, y, (knoten.getFather().getX() * 10)
* scaleFactor, (int) ((knoten.getFather().getY() * 10) *
scaleFactor * scaleY));
59
60     g.fillOval(x - 3 * scaleFactor, y - 3 * scaleFactor, 6
* scaleFactor, 6 * scaleFactor);
61 });
62
63 // Zuletzt wird hier noch zusaetzlich der Knoten-Buchstabe
gedruckt
64 // Hierfuer werden gewisse Voreinstellungen zum zeichnen
geaendert
65 g.setColor(Color.WHITE);
66 g.setFont(new Font("Arial", Font.PLAIN, (int) (3 *
scaleFactor * scaleY)));
67 FontMetrics fm = g.getFontMetrics();
68
69 // Zeichnen aller Buchstaben in die Mitte des jeweiligen
Knotens
70 root.traversPreOrder(knoten -> {
71     int x = (knoten.getX() * 10) * scaleFactor;
72     int y = (int) ((knoten.getY() * 10) * scaleFactor *
scaleY);
73
74     g.drawString(" " + knoten.getData(), x - fm.charWidth(
knoten.getData()) / 2, y + 1.5f * scaleFactor * scaleY);
75 });
76
77 // Grafik speichern
```

```

78         g.dispose();
79
80         // Bild ins Dateisystem speichern
81         ImageIO.write(img, "PNG", new File("out/" + filename + ".
png"));
82     }
83 }

```

Quellcode B.0.7: Zusatzklasse: Drawer

```

1 package algos;
2
3 import java.util.function.Consumer;
4
5 public class Mains {
6
7     /**
8      * Zeichne einen Baum und speichere diesen in eine PNG-Datei.
9      *
10     * @param tree Wurzel-Knoten des Baums
11     * @param algo Algorithmus, der zum Positionieren genutzt
12     * werden soll
13     * @param fileName Dateiname, in der das Resultat gespeichert
14     * werden soll
15     * @throws Exception z.B. FileNotFoundException...
16     */
17     private static void draw(BinaryKnoten tree, Consumer<
18     BinaryKnoten> algo, String fileName) throws Exception {
19         algo.accept(tree);
20         Drawer.drawIntoImage(tree, fileName);
21     }
22
23     /**
24     * @param tree Wurzel-Knoten des Baums
25     * @return Hoehe des Baums
26     */
27     private static int getTreeHeight(BinaryKnoten tree) {
28         int[] maxY = {0};
29         tree.traversPreOrder(x -> {
30             if(maxY[0] < x.getHoehe())
31                 maxY[0] = x.getHoehe();
32         });
33
34         return maxY[0];
35     }
36
37     // Option = Welche Aktion in der Main soll ausgefuehrt werden
38     private static int runOption = 2;
39
40     public static void main(String[] args) throws Exception {
41         if(runOption == 0) {
42             draw(Trees.createKomplexTree(), TilfordAlgorithmus::
43             algorithmus3, "v2/komplex_a3");
44             draw(Trees.createKomplexTree(), x -> {
45                 VerbesserterAlgorithmus.algorithmus2Verbessert(x, getTreeHeight
46                 (x) + 1); }, "v2/komplex_a2_v");
47         }
48     }
49 }

```

```
41         draw(Trees.createKomplexTree(), x -> {
VerbesserterAlgorithmus.algorithmus2(x, getTreeHeight(x) + 1);
}, "v2/komplex_a2");
42         draw(Trees.createKomplexTree(), x -> {
NaiverAlgorithmus.algorithmus1(x, getTreeHeight(x) + 1); }, "v2
/komplex_a1");
43     }
44     else if(runOption == 1) {
45         draw(Trees.createTree1(), TilfordAlgorithmus::
algorithmus3, "v2/tree_spiegel_1_a3");
46         draw(Trees.createTree2(), TilfordAlgorithmus::
algorithmus3, "v2/tree_spiegel_2_a3");
47         draw(Trees.createTree1(), x -> {
VerbesserterAlgorithmus.algorithmus2(x, getTreeHeight(x) + 1);
}, "v2/tree_spiegel_1_a2");
48         draw(Trees.createTree2(), x -> {
VerbesserterAlgorithmus.algorithmus2(x, getTreeHeight(x) + 1);
}, "v2/tree_spiegel_2_a2");
49     }
50     else if(runOption == 2) {
51         draw(Trees.beispielZeigenLL_LR_RL_RR_Tree(),
TilfordAlgorithmus::algorithmus3, "tree_beispiel_LL_LR_RL_RR");
52     }
53 }
54
55 }
```

Quellcode B.0.8: Zusatzklasse: Mains