Gymnasium Königsbrunn **Oberstufenjahrgang 17/19**

SEMINARARBEIT

im Fach

Informatik

Thema:

Huffman-Algorithmus mit Schwerpunkt auf der dynamischen Huffman-Codierung

Verfasser: **Marcel Stepanek**

Kursleiterin: **StRin Anna-Gwendolyn Huber**

Inhaltsverzeichnis

1 Einführung

1.1 Historischer Rückblick

Seit jeher ist die Informationsübermittlung im Allgemeinen eine der wichtigsten Dinge im menschlichen Leben. Es könnte sich dabei beispielsweise um einen gesprochenen Satz, ein Verkehrszeichen, die neueste WhatsApp-Nachricht, private Bankdaten oder vielleicht auch um geheime Staatsdokumente handeln. Die Übermittlung selbst ist dabei aber nicht immer einfach. Nachrichten schnell und vor möglichen Mitwissern geschützt zu versenden, birgt große Herausforderungen, damals wie heute.

In Kriegszeiten, wie beispielsweise dem zweiten Weltkrieg hatten die deutschen U-Bootfahrer gegen die immerwährende Bedrohung durch feindliche Abhörstationen und den ständigen Zeitdruck zu kämpfen, um nicht ihre Position beim Senden der Nachricht bekannt zu geben.

Um die gesendeten Informationen geheim zu halten und nicht dem Feind preiszugeben wurde die Verschlüsselungs-/ Chiffriermaschine „Enigma“ benutzt. Sie basierte im Grunde auf festgelegten Ver- und Entschlüsselungstabellen (genannt Walzen), die täglich geändert wurden. Der Absender sowie der Empfänger der Nachricht mussten die gleiche Tabelle besitzen, um miteinander kommunizieren zu können, d.h. die Nachrichten verschlüsseln oder lesen zu können.

Dieses Funktionsprinzip ist bis heute in vielen der einfachen Verschlüsselungsalgorithmen oder Datenkomprimierungsprogrammen enthalten.

1.2 Verbindung der Enigma-Methode mit dem Huffman-Algorithmus

Das Funktionsprinzip des Huffman-Algorithmus, ähnlich dem der Enigma, basiert auf zwei gleichen Entschlüsselungstabellen. Der Algorithmus strebt damit nach den Anforderungen der effizienten (möglichst verlustarmen) und somit schnellen Datenübermittlung.

Es wird dabei versucht, die zu übertragende Datenmenge ohne Datenverlust zu komprimieren, um die in Anspruch genommene Übertragungszeit der Datenmenge möglichst klein zu halten.

Zur Zeit der Entwicklung des Huffman-Algorithmus (1952) durch David A. Huffman war dies entscheidend. Die Übertragungsraten der Verbindungen (anfangs noch Telefonleitungen) waren sehr gering, womit die Übertragungszeiten der gesendeten Datenmenge erheblich lang wurden.

1.3 Begriffserläuterung[1]

Um Verwirrungen zu vermeiden wird kurz der Begriff der Redundanz mithilfe des Begriffs der Information nach Shannon erläutert, da Huffman diese in seiner Publizierung von 1952 als Grundlage nutzt. Claude E. Shannon definiert diese zwei Begriffe in seinem Buch „THE MATHEMATICAL THEORY OF COMMUNICATION - by Claude E. Shannon and Warren Weaver“ von 1949, erstmals für den Bereich der Informationstheorie.

Der grundlegende Begriff der **Information** wird nicht mehr als Ausdruck der „Bedeutung“ gesehen, sondern bezeichnet nun lediglich die Länge des Codes, der für die Übertragung einer bestimmten Nachricht benötigt wird. Die Bedeutung der Nachricht, wie wir sie kennen, spielt keine Rolle mehr.

Ist der Code aber nun länger als seine eigentliche Information, ist er redundant (lat.: „redundare“ - im Überfluss vorhanden sein). Die **Redundanz** beschreibt hierbei das vorhanden Sein von überflüssigem Code, der die gleiche Information besitzt und deshalb eingespart werden könnte.

In der nachfolgenden Arbeit wird der Begriff der „Information“ in seiner ursprünglichen, nicht informationstechnischen Semantik verwendet.

1.4 Zielsetzung

In dieser wissenschaftlichen Arbeit wird das Funktionsprinzip des Huffman-Algorithmus geschildert und auf die verschiedenen Komprimierungsverfahren eingegangen.

Diese Arbeit legt den Schwerpunkt auf die dynamische Huffman-Codierung, die mit Beilspielcode genauer erläutert wird. Die statische und adaptive Huffman-Codierung werden nur kurz thematisiert.

Danach wird ein kurzer Einblick in die derzeitige Verwendung gegeben und Kritik an der Funktionsweise des Algorithmus geäußert.

[1] Vgl. Miszalok, Volkmar (2011): Der Begriff der REDUNDANZ, http://www.miszalok.de/Lectures/L01_Redundancy/Redundancy_d.htm#a2 (Stand: 01.11.2018)

2 Allgemeines Funktionsprinzip[2]

Die allgemeine Funktion des Huffman-Algorithmus ist die Komprimierung eingegebener Datensätze, meistens Textdateien oder heutzutage auch Bild- und Videodateien. Diese werden mithilfe der Huffman-Codierung zuerst in andere Zeichen umgewandelt und dadurch komprimiert. Dem Algorithmus gelingt die Verringerung des benötigten Speicherplatzes dabei durch die Erstellung eines „minimum-redundancy code“ (Huffman 1952, S.1098). Diese Codierung wird als optimaler Code mit kleinster oder minimaler Redundanz bezeichnet und stellt die komprimierte Form der ursprünglichen Datei dar.

2.1 Komprimierungsverfahren

Bei einer Komprimierung im Sinne der Informatik wird generell zwischen zwei Komprimierungsarten unterschieden, der verlustfreien und der verlustbehafteten Komprimierung. Das allgemeine Ziel einer Komprimierung ist die Speicherplatzersparnis gemessen am Komprimierungsfaktor. Er gibt das Verhältnis der Größe der ursprünglichen Datei zur komprimierten Datei an.
Je größer dieser Faktor am Ende ist, desto mehr Speicherplatz wurde eingespart und desto effizienter ist die Komprimierung.

Bei der verlustbehafteten Komprimierung sind die Informationen am Ende nur ähnlich und stimmen nicht exakt mit den ursprünglichen Daten überein.
Bei ihrer Rekonstruktion geht ein kleiner Teil verloren, dieser Vorgang wird auch Datenreduktion genannt.
Im Gegensatz dazu steht die verlustfreie Komprimierung. Hier kann die Information aus den gesendeten Daten vollständig wiederhergestellt werden, genannt Datenkompression.
Vereinfacht vorstellbar ist diese Komprimierung im Vergleich mit moderner Astronautennahrung. Die kleinen Plastiktüten haben im Allgemeinen nur einen begrenzten Stauraum („Speicherplatz“). Dieser Stauraum wird nun mit dem entsprechenden Nahrungsmittel („Daten“) gefüllt. Das Resultat: Die Tüte ist nach dem Füllen voll und würde im Gesamten einen großen Teil des kostbaren Platzes im Frachtraum einnehmen. Saugt man jedoch die überschüssige, unnötige Luft aus der Tüte ab, wird das Volumen der Tüte kleiner und ihr Platzbedarf reduziert.

[2] Vgl. Eiden 1999, S. 4 - 5

Es wurde („Speicher-“) Platz gespart. Der Inhalt der Tüte ist unverändert geblieben und stimmt mit dem Vorherigen überein.

Der Huffman-Algorithmus gehört zu den verlustfrei komprimierenden Algorithmen. Die gesendeten Daten stimmen mit den entschlüsselten Daten am Ende exakt überein und es ist kein Datenverlust festzustellen. Dies ist nur aufgrund der Redundanzen innerhalb der Quelldaten möglich, da nur diese ohne Informationsverlust eingespart werden können.

Um die Daten nun verlustfrei komprimieren zu können, muss zuerst ein Codierungsprozess erfolgen, dieser besteht aus zwei großen Schritten.
Der erste Schritt beinhaltet das Erstellen der Häufigkeitstabellen.
Im zweiten Schritt findet dann mithilfe des Huffman-Baumes die eigentliche Codierung der Eingabedaten statt.

2.2 Häufigkeitstabellen

Diese Tabellen sind der erste Schritt in der Codierung der Daten.
Diese können beim Huffman-Algorithmus **statisch**, **dynamisch** oder **adaptierend** sein. Die drei Tabellen unterscheiden sich dabei nur in ihrer Erstellung und in den daraus resultierenden Eigenschaften.
Sie liefern die Häufigkeiten, die für die Erstellung des Huffman-Baumes (Kapitel 2.4), dem zweiten Schritt der Codierung, notwendig sind.
Die hohe Effizienz der gesamten Codierung hängt dabei maßgeblich von der Häufigkeitsverteilung der einzelnen Zeichen innerhalb der Nachricht ab.
Je höher die Häufigkeit eines Zeichens innerhalb der Nachricht ist, desto kürzer seine codierte Länge (aufgrund des leichteren Verständnisses wird hierfür der Begriff Codierungslänge verwendet).
Die Tabelle hilft nun bei der Zuweisung dieser Codierungslänge.
Das Zeichen, das beispielsweise in der Tabelle die höchste Wahrscheinlichkeit besitzt, bekommt im nächsten Schritt durch den Huffman-Baum die kleinstmögliche Codierungslänge zugewiesen. Dies geschieht, da dieses Zeichen statistisch am häufigsten vorkommt, somit durch die kleinste Codierungslänge der meiste Speicherplatz pro Zeichen eingespart und der größte Komprimierungserfolg erzielt wird. Die Codierung komprimiert die Nachricht also durch die effiziente Umwandlung der einzelnen Zeichen.

2.2.1 Statische Huffman-Codierung/ Static Huffman Coding

Bei der statischen Huffman-Codierung werden bereits festgelegte, unveränderbare (statische) Tabellen als Grundlage der Komprimierung verwendet. Dies hat den Vorteil, eine schon vorher festgelegte Häufigkeitsverteilung nutzen zu können, ohne diese erneut zu berechnen.

Statische Häufigkeitstabellen bestehen beispielsweise aus Tabellen (Abbildung 1[3]) zur Berechnung der Wahrscheinlichkeit des Vorkommens eines Buchstabens in der deutschen Sprache. Je häufiger ein Buchstabe statistisch vorkommt, desto höher ist die Wahrscheinlichkeit, ihn in einem deutschen Text zu finden. Je höher nun die Wahrscheinlichkeit eines Buchstabens ist, desto kleiner wird seine Codierungslänge.

Beim Beispiel der Tabelle des deutschsprachigen Alphabets kommt der Buchstabe „e“ (17,4%) im Durchschnitt viel häufiger vor als der Buchstabe „z“ (1,1%). Der Buchstabe „e“ würde nun eine kleinere Codelänge erhalten als der Buchstabe „z“.

Buchstabe	Deutsch
a	6,5 %
b	1,9 %
c	3,0 %
d	5,1 %
e	17,4 %
f	1,7 %
g	3,0 %
h	4,8 %
i	7,6 %
j	0,3 %
k	1,2 %
l	3,4 %
m	2,5 %
n	9,8 %
o	2,5 %
p	0,8 %
q	0,02 %
r	7,0 %
s	7,3 %
t	6,2 %
u	4,4 %
v	0,7 %
w	1,9 %
x	0,03 %
y	0,04 %
z	1,1 %
œ	-
ß	0,3 %

Abbildung 1- Häufigkeitsverteilung der Buchstaben in der deutschen Sprache

2.2.2 Dynamische Huffman-Codierung/ Dynamic Huffman Coding

Bei dieser Form der Huffman-Codierung werden nicht wie bei der statischen Methode vorgefertigte Tabellen verwendet, sondern es wird für jede einzelne zu codierende Information eine spezielle Tabelle erstellt. Hier besteht der Vorteil darin, mit einer noch genaueren Häufigkeitsverteilung arbeiten zu können und dadurch im Vergleich zur statischen Methode eine optimalere Codierung für die Eingabewerte zu finden.

Die Zuweisung der Häufigkeit und das Erstellen der Häufigkeitstabelle beruht hierbei auf der relativen Häufigkeit der einzelnen Zeichen. Um nun diese für die einzelnen Zeichen der Nachricht zu berechnen, wird die absolute Häufigkeit des Zeichens (P) durch die Gesamtlänge der Nachricht (Ω) geteilt. Dies ergibt nun die relative Häufigkeit des Zeichens in der Nachricht (siehe Formel unten).

$$relative\ Häufigkeit\ des\ Zeichens\ P = \frac{absolute\ Häufigkeit\ des\ Zeichens\ P}{\text{Gesamtlänge der Nachricht } \Omega}$$

[3] Bildquelle zur Abbildung 1: http://www.mathe.tu-freiberg.de/~hebisch/cafe/kryptographie/haeufigkeitstabellen.html

Um dies zu verdeutlichen wird nun eine Häufigkeitstabelle am Beispiel der Nachricht: „Komprimieren[ ]ist[ ]super!“ ([ ] entsprechen Leerzeichen) erstellt.
Die vorkommenden Zeichen sind:
„K“, „O“, „M“, „P“, „R“, „I“, „E“, „N“, „[ ]“, „S“, „T“, „U“, „!“.
Die gesamte Länge der Nachricht beträgt 23 Zeichen.
Es sind jeweils dreimal die Zeichen „R“, „I“ und „E“, jeweils zweimal die Zeichen „M“, „P“, „[]“ und „S“, sowie jeweils einmal die Zeichen „K“, „O“, „N“, „T“, „U“ und „!“ enthalten. Beispielrechnung am Zeichen: „R“: 3/23 = 0,13 = 13 %.
Mithilfe dieser Berechnung wird nun die Häufigkeitstabelle für diese spezielle Nachricht erstellt (Siehe Tabelle 1).

Zeichen	Anzahl	Häufigkeit	in Prozent
K	1	0,043	4%
O	1	0,043	4%
M	2	0,087	9%
P	2	0,087	9%
R	3	0,130	13%
I	3	0,130	13%
E	3	0,130	13%
N	1	0,043	4%
[ ]	2	0,087	9%
S	2	0,087	9%
T	1	0,043	4%
U	1	0,043	4%
!	1	0,043	4%
insg.:	23		~100%

Tabelle 1 - Häufigkeitstabelle der Zeichen zum Beispiel "Komprimieren ist super!"

Hier würde nun beispielsweise das Zeichen „R“ (13%) eine kleinere Codierungslänge als das Zeichen „K“ (4,3%) erhalten, da es häufiger in der zu codierenden Datenmenge vorkommt.

Bei der statischen und der dynamischen Huffman-Codierung wird jeweils der Ausgangsbaum mit den Häufigkeiten zum Dekodieren benötigt und muss deshalb mitgeschickt werden.

2.2.3 Adaptive Huffman-Codierung/ Adaptive Huffman Coding[4]

Die adaptive Huffman-Codierung ist eine Weiterentwicklung der statischen- und dynamischen Huffman-Codierung. Gegenüber den anderen Methoden müssen bei diesem Verfahren die Häufigkeiten beim Übertragen der Daten nicht bekannt sein. Während der Übertragung wird zu jedem Zeitpunkt die Häufigkeit ermittelt und somit die Möglichkeit geschaffen, den Code in Echtzeit anzupassen.
Die adaptive Methode eignet sich deswegen auch für die Echtzeitcodierung, beispielsweise bei Online-Übertragungen, da der Code immer für den jeweiligen Zeitpunkt optimal ist. Der große Vorteil besteht aber darin, dass die Quelldatei am Anfang nur einmal bei der Komprimierung gelesen werden muss und die Statistik beim Dekomprimieren aus den gesendeten Daten erstellt wird.
Die adaptive Huffman-Codierung wird auch nach ihren Entwicklern FGK-Algorithmus (Faller-Gallagher-Knuth-Algorithmus) oder Vitter-Algorithmus genannt.

Um nun die codierten Daten aus einem Fließtext wieder entschlüsseln zu können, muss allerdings eine eindeutige Zuordnung und klare Abgrenzung der einzelnen Codewörter vorliegen.

2.3 Huffman-Codierung als Präfixcode[5]

Um nun Verwirrungen bei der Zuordnung in der Entschlüsselungsphase zu vermeiden, erzeugt die Huffman-Codierung einen sogenannten **Präfixcode**.
Ein Präfixcode hat die Eigenschaft, dass jedes seiner Codewörter einen anderen Anfangsteil besitzt und niemals mehrere Codewörter mit den gleichen Zeichen beginnen. Somit ist jedes Codewort einzigartig und klar in der codierten Zeichenkette erkennbar. Zum Verständnis hier ein kleines Beispiel:
Der Buchstabe „a“ wird mit den Zeichen 01, das Zeichen „b“ mit 11 und das Zeichen „c“ mit 101 codiert. Wird nun die Information „abc“ codiert entsteht die codierte Zeichenkette 0111101. Um nun die Information aus der Zeichenkette auszulesen wird diese zerlegt. Das erste Codewort kann dabei nur das „a“ sein, da dies als einziges mit dem Zeichen „0“ anfängt. Also 01 | 11101 = a | 11101. Dann muss aufgrund der darauffolgenden Zeichen „11“ der Buchstabe „b“ folgen. Also a | 11 | 101 = a | b | 101. Da nun nur noch ein Codewort übrig bleibt, wird dieses dem Buchstaben „c“ zugeordnet.

---

[4] Vgl. Liśkiewicz, Maciej/ Fernau, Henning, oJ., S. 23) & (Vgl.: Möhring 2005, S. 41
[5] Vgl. Möhring 2005, S. 31ff.

Die Information aus der codierten Zeichenkette lautet also a | b | c. Die Codewörter konnten ohne Probleme den eigentlichen Buchstaben zugewiesen werden.

Damit die Huffman-Codierung solch einen Präfixcode mit einer codierten Zeichenkette erstellen kann, muss erst noch der zweite Schritt der Codierung durchlaufen werden. Die Codierung durch den Huffman-Baum.

2.4 Der Huffman-Baum[6]

Der letzte Schritt in der Codierung des Huffman-Algorithmus ist die Erstellung eines Huffman-Baums. Dieser ist ein Binärbaum mit besonderen Eigenschaften. Je nach der Vorliebe des Entwicklers benutzt dieser die absolute oder die relative Häufigkeit der einzelnen Zeichen aus der erstellten Tabelle.
Er wird nicht wie normale Binärbäume von der Wurzel aus nach unten aufgebaut, sondern „Die Konstruktion ist ein sogenanntes bottom-up-Verfahren, das heißt, dass der zur Kodierung gehörende Baum aus den Blättern, die am Anfang gegeben sind, konstruiert wird.“ (Lars Donat 2002, S.1).
Alle zu codierenden Zeichen befinden sich dabei in den Blattknoten des Baumes. Alle anderen Knoten sind Elternknoten und mit den Häufigkeiten der Zeichen gewichtet. Wenn die absolute Häufigkeit genutzt wird, enthält die Wurzel des Baums das Gesamtgewicht aller Zeichen - also die eigentliche Länge der Eingabenachricht. Wenn die relative Häufigkeit als Gewichtung benutzt wird, enthält die Wurzel den Wert 1 oder 100%.
Die Kanten werden für den gesamten Baum durchgehend entweder mit links „0“ und rechts „1“ oder links „1“ und rechts „0“ beschriftet. Dies ist entscheidend, da später durch das Ablaufen der Kanten der Code für den Eingabewert erstellt und dieser durch die Kantenwerte repräsentiert wird. Die Abbildung 2 zeigt eine allgemeine Darstellung des Huffman-Binärbaumes.

[6] Vgl. http://www.ziegenbalg.ph-karlsruhe.de/materialien-homepage-jzbg/cc-interaktiv/huffman/codierung.htm

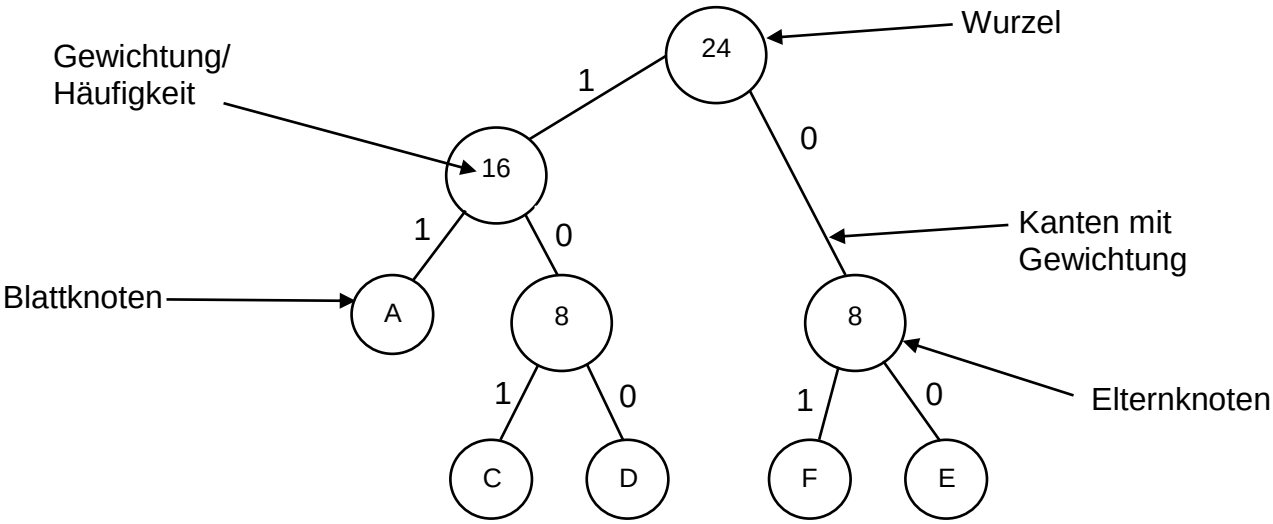

Abbildung 2 - Allgemeine Darstellung eines Huffman-Baumes zu einem beliebigen Beispiel

Der Huffman-Baum selbst wird in drei grundlegenden Phasen aufgebaut.
Die Phasen werden wieder am Beispieltext „Komprimieren[]ist[]super!“ („[ ]“ entsprechen Leerzeichen) erläutert.

Phase eins entspricht dem Auslesen der jeweiligen Häufigkeitstabelle und dem Vorgeben der Anfangswerte.
Jedem Buchstaben wird seine Häufigkeit zugeordnet. Bei dem oben genannten Beispiel ist dies die Häufigkeitstabelle aus Kapitel 2.2.2 (Tabelle 1), es werden zum leichteren Verständnis die absoluten Häufigkeiten verwendet.
Die Eingabenachricht enthält folgende Zeichen:
„K“, „O“, „M“, „P“, „R“, „I“, „E“, „N“, „[ ]“, „S“, „T“, „U“, „!“.
Nun werden allen Zeichen ihre Häufigkeitswerte zugeordnet.
Oben steht das jeweilige Zeichen und darunter die Häufigkeit in „Knotenform“.

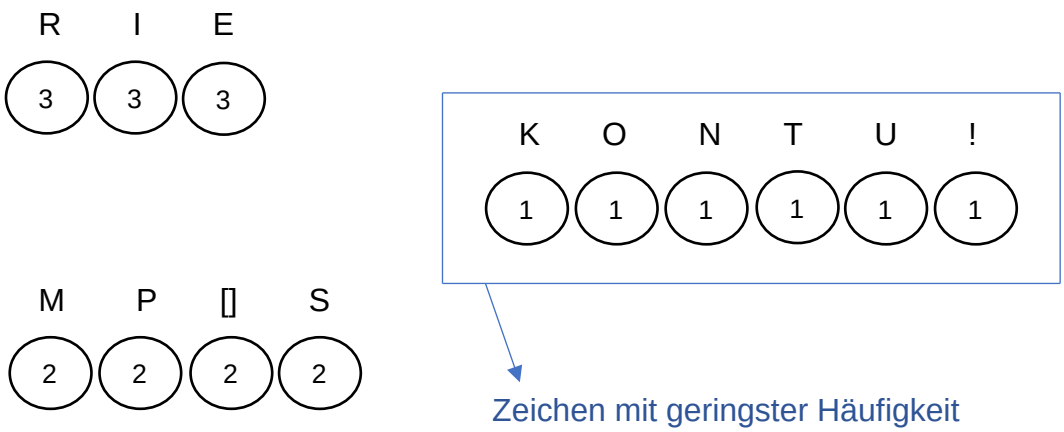

Abbildung 3 - Häufigkeitszuordnung

In **Phase zwei** werden dann aus den Zeichen mit den geringsten Häufigkeiten kleine Teilbäume erstellt und somit die unterschiedlichen Ebenen gebildet. Die Teilbäume werden dabei aus jeweils einem Elternknoten und zwei Kindknoten erzeugt. Die Gewichtung der Kindknoten entspricht dabei dem Häufigkeitswert eines Zeichens. Die Gewichtung des größeren Elternknotens ergibt sich aus den beiden Häufigkeitswerten seiner Kindknoten, die miteinander addiert werden.
Achtung: Kindknoten können gleichzeitig auch Elternknoten sein.
Dies passiert, wenn bereits ein Elternknoten (K1) aus den Häufigkeiten zweier Zeichen gebildet wurde und nun dieser Knoten K1 mit einem anderen Elternknoten (K2) einen neuen Elternknoten (K3) bildet.
Es werden zuerst die Zeichen mit der geringsten Häufigkeit ausgewählt und aus ihnen Teilbäume als Grundlage des gesamten Huffman-Baums erstellt.
Die Kindknoten für die erste Ebene haben im vorher genannten Beispiel jeweils die Werte „1“ und somit haben die Elternknoten die Werte „2“. Jeder neue Kindknoten ist grün umrahmt und jeder neue Elternknoten rot.

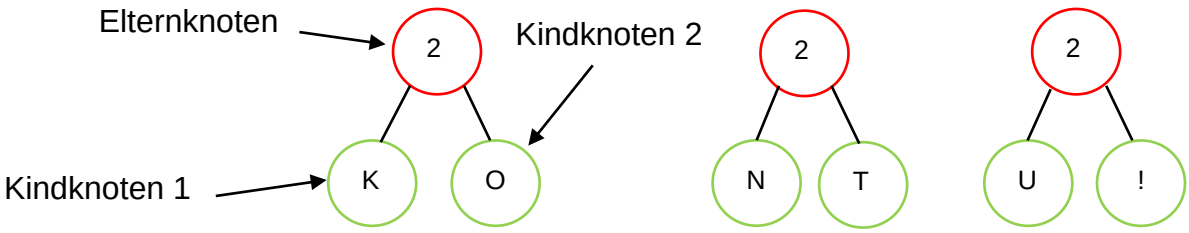

Abbildung 4 - Bildung der Teilbäume der untersten Ebene

In der letzten und damit **dritten Phase** werden nun immer die Teilbäume und Zeichen mit der kleinsten Häufigkeit zu einem neuen Elternknoten zusammengefasst. Es wird mit jedem neuen Bilden eines Teilbaumes Ebene für Ebene erstellt. Dieser Vorgang wird solange wiederholt, bis sich alle Zeichen in einem großen Baum befinden. Somit ergeben alle Ebenen und auch Teilbäume zusammen den gesamten Huffman-Baum.
Zurück zu unserem Beispiel „Komprimieren[ ]ist[ ]super!“:
Die erste Ebene aus Phase 2 wird übernommen.
Nun werden aus den Elternknoten und den übrigen Häufigkeitswerten jeweils die kleinsten Werte zusammengenommen und neue Elternknoten gebildet.
Die kleinsten Werte dabei haben jetzt die Zeichen: „M“, „P“, „S“ und „[ ]“.

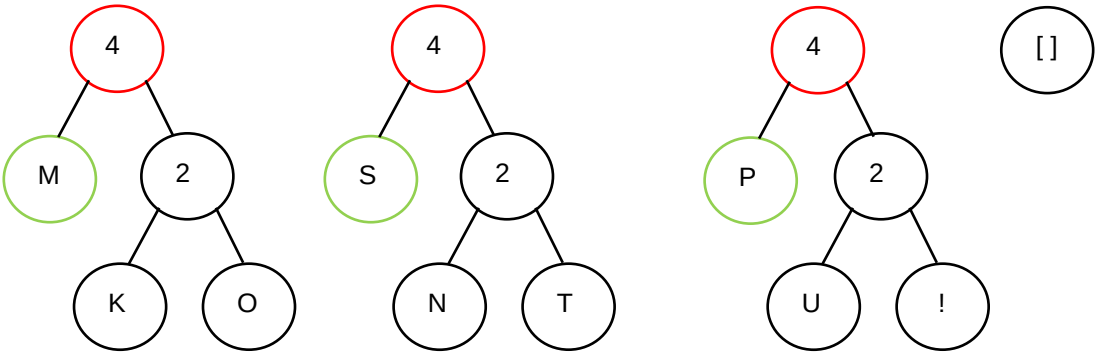

Abbildung 5 - Fortsetzung der Teilbäume von Abbildung 4

Es werden aus allen Zeichen außer „[ ]“ neue Teilbäume gebildet.
In diesem Fall, dass ein Zeichen übrigbleibt, wird dieses mit in die nächste Ebene übernommen und im darauffolgenden Schritt verwendet.
Dieser Ablauf wird nun wie schon erwähnt solange wiederholt, bis alle Zeichen verwendet wurden, dies können in der Theorie unendlich viele sein.
In unserem Beispiel geht es jetzt mit den Zeichen mit der nächst-größeren Häufigkeit weiter. Dies sind die Zeichen: „R“, „I“ und „E“.
Es werden nun wieder nach dem gleichen Prinzip neue Elternknoten gebildet.

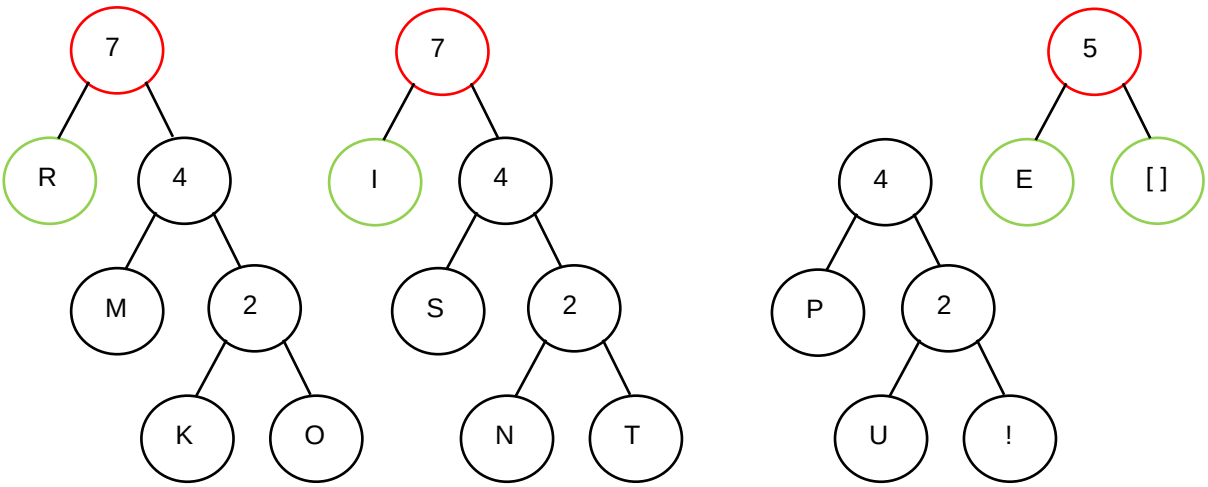

Abbildung 6 - Fortsetzung der Teilbäume von Abbildung 5

In diesem Schritt wurden wieder drei neue Elternknoten gebildet und das vorher übrig gebliebene Zeichen „[ ]“ bildet nun einen Teilbaum mit dem Zeichen „ E“.
Da zu diesem Zeitpunkt keine Zeichen mehr übrig sind, werden nur noch die kleinsten Gewichte der Elternknoten miteinander zu Teilbäumen verbunden.
In unserem Beispiel sind dies auf der linken Seite die zwei Teilbäume mit jeweils dem Gewicht 7 und die Teilbäume auf der rechten Seite mit den Gewichten 4 und 7.

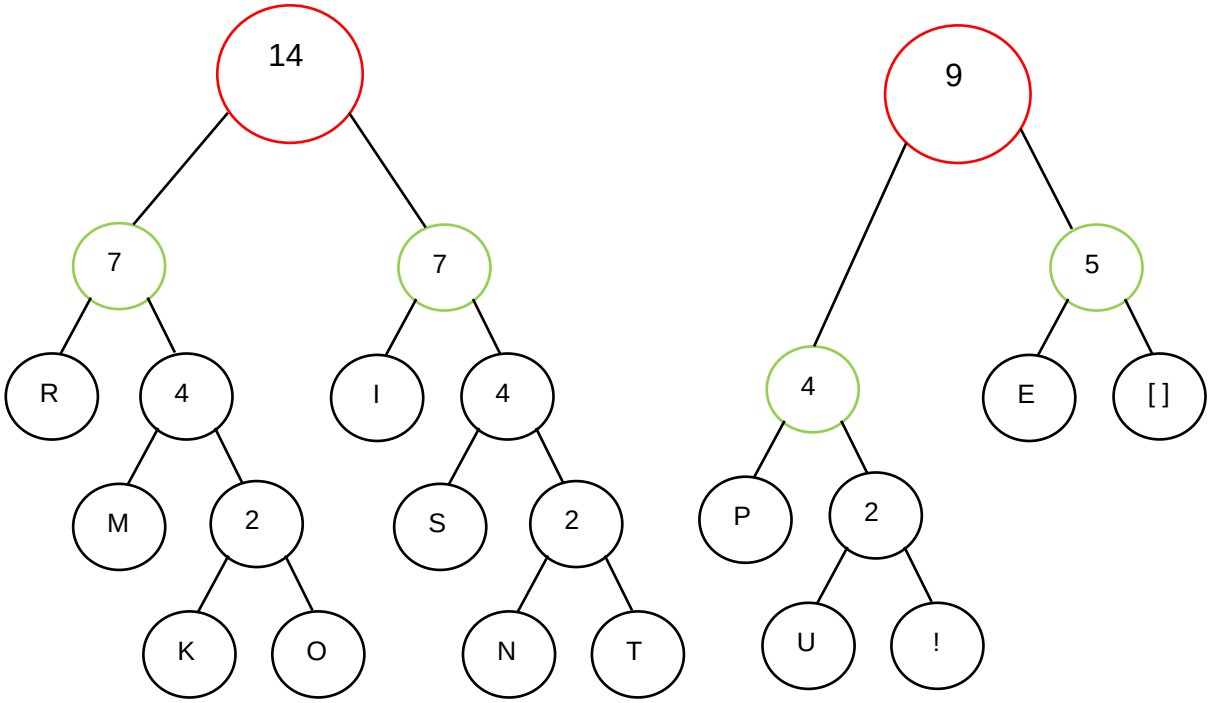

Abbildung 7 - Fortsetzung der Teilbäume von Abbildung 6

Die kleinen Teilbäume wurden nun zu zwei größeren Teilbäumen zusammengesetzt. Der letzte Schritt in diesem Beispiel ist nun die Erstellung des gesamten Huffman-Baumes. Hierbei werden nun die zwei letzten Teilbäume zu einem einzigen Binärbaum verbunden.

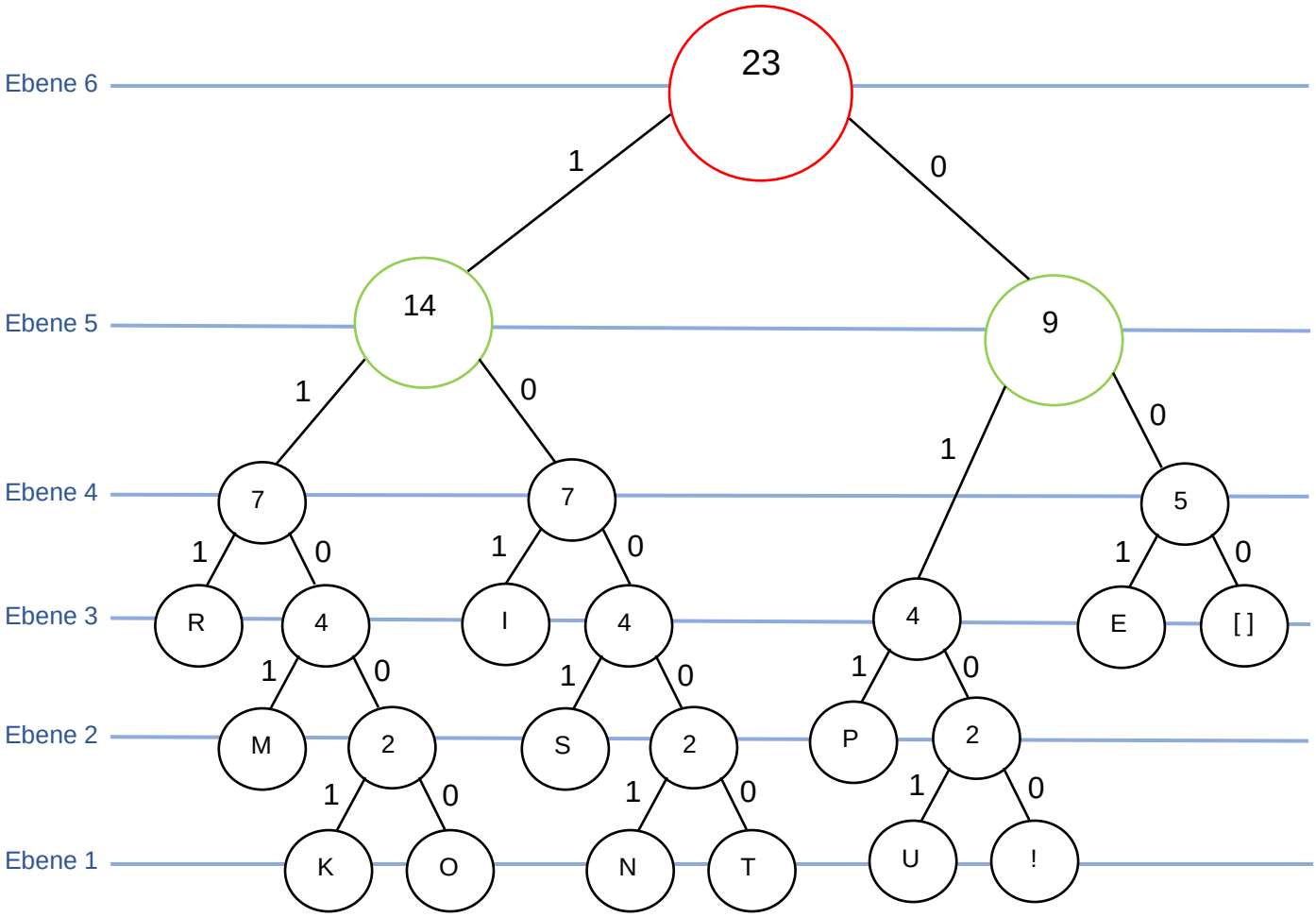

Abbildung 8 - Huffman-Baum für das Beispiel "Komprimieren[ ]ist[ ]super!"

Der Huffman-Baum wurde nun schlussendlich erstellt und die linken Kanten wurden mit „1“, die rechten Kanten mit „0“ beschriftet.

Wenn Sie nun Lust haben, sich selbst einen Huffman-Baum zu erstellen, gibt es hier eine praktische und unkomplizierte Animation für Ihr selber gewähltes Beispiel: https://people.ok.ubc.ca/ylucet/DS/Huffman.html

Der Baum ist nun bereit, um die Ausgangsnachricht zu codieren, die Komprimierung kann jetzt beginnen.

2.5 Codierung

Jedes Zeichen, das mithilfe des Huffman-Baums codiert wird, steht in einem seiner Blattknoten. Um ein bestimmtes Zeichen zu codieren wird zuerst seine Position im Binärbaum bestimmt. Ist das gesuchte Zeichen gefunden, wird von der Wurzel aus über die Kanten der Weg zum gesuchten Blattknoten ermittelt. Dabei wird nach dem „überlaufen“ einer Kante deren Gewicht notiert.
Die aufgeschriebene Reihenfolge von der ersten bis zur letzten besuchten Kante entspricht dann dem Codewort des jeweiligen ermittelten Zeichens.
Die Anzahl der überschrittenen Ebenen stellt die Länge des Codewortes für das jeweilige Zeichen dar. Zur Verdeutlichung wird dieser Vorgang an dem vorherigen Beispiel für das Zeichen „M“ durchgeführt:
Das gesuchte Zeichen „M“ befindet sich in Ebene 2 des linken Teilbaumes.
Es wird bei der Wurzel gestartet. Zuerst wird in den linken Teilbaum über die Kante „1“ abgebogen. Die „1“ wird notiert → 1.
Im nächsten Schritt wird wieder über die „1“ in die vierte Ebene übergeleitet.
Das Codewort lautet jetzt 11.
Danach wird über die Kante „0“ in Ebene 3 verwiesen.
Das Codewort erweitert sich um die Stelle „0“ → 110.
Um nun zu dem gesuchten Zeichen zu kommen, wird durch die letzte Kante „1“ auf den Blattknoten mit dem Wert „M“ verwiesen. Das Codewort für das Zeichen „M “ lautet nun 1101. Es wurden insgesamt 4 Ebenen überschritten, das Codewort hat also eine Länge von 4 Stellen. Die Decodierung der Zeichenkette geschieht rückwärts auf dem gleichen Wege. Es wird bei der Wurzel begonnen und über den vorgegebenen Pfad der Kantengewichte das codierte Zeichen erreicht. Soviel nun zur Theorie des Huffman-Algorithmus. Im Folgenden wird nun die Praxis in Form einer Implementierung in Java behandelt.

3 Erläuterung der exakten Funktionsweise anhand eines Beispiels mit Java-Implementierung[7]

Der implementierte Algorithmus wird in diesem Kapitel durch die Screenshots der selbst programmierten Java-Implementierung in der Entwicklungsumgebung BlueJ erläutert. Es werden die Hauptmethoden vorgestellt und erklärt. Bei der folgenden Erläuterung wird minimales Grundwissen aus der Informatik im Bereich der objektorientierten Programmiersprache Java vorausgesetzt. Diese Implementierung finden Sie selbstverständlich auch im Anhang dieser Arbeit. Um die Implementierung ausführen zu können, muss das Projekt in der BlueJ-Umgebung geöffnet und mithilfe eines Objektes der Klasse Start angestoßen werden. Der Eingabewert muss danach in die enthaltene Methode „Start (String eingabe)“ eingegeben werden, damit der Baum per Konsolenausgabe erstellt wird.

Die Implementierung ist ähnlich wie bei der Theorie in drei große Themenblöcke aufgeteilt. Als erstes werden die Häufigkeiten der Eingabewerte ausgewertet, im darauffolgendem Unterkapitel wird der der Huffman-Baum erstellt und im letzten Unterkapitel wird der erstellte Binärbaum zur Codierung und Komprimierung der Zeichenkette genutzt.

3.1 Auswertung der Häufigkeiten der Eingabezeichenkette

Im ersten Schritt muss nun das eingegebene zu codierende Wort vom Datentyp String in seine einzelnen Zeichen aufgeteilt werden, um danach einzeln als character in einem Blattknoten vorzuliegen. Dies geschieht in der Methode „Eingabe (String wort)“.

```
public void Eingabe(String wort)
{
    for (int i=0;i<wort.length();i++){
        Blattknoten bk = new Blattknoten (wort.charAt(i), 1);
        hinzufügen(bk);
    }
}
```

Abbildung 9 - Methode Eingabe (String wort)

Die Aufteilung des Eingabewerts geschieht hier durch eine for-Schleife mit der Länge des Quelltextes und mit der in ihr enthaltenen Methode „ .charAt()“. Diese liefert den Ausgabewert vom Datentyp character an dem angegebenen Index. Der hier entstandene character wird direkt in einen Blattknoten mit der Häufigkeit „1“ umgewandelt.

[7] Vgl. Jungermann, Florian (2012), o.S.

Dies geschieht, da der entstandene Blattknoten im nächsten Schritt in eine ArrayList eingefügt wird und vorerst noch die gesamten Häufigkeiten der Zeichen unklar sind. Die ArrayList selbst wurde bereits vorher deklariert und initialisiert.
Im zweiten Schritt werden nun die für den Huffman-Algorithmus benötigten Häufigkeiten ermittelt. Jedes Zeichen, egal wie oft es in der eingegebenen Zeichenkette vorkommt, existiert nun als einzelner Blattknoten mit der Häufigkeit „1“. Da nun aber eine gleiche Häufigkeitsverteilung äußerst ungünstig für die folgende Codierung wäre, darf jedes Zeichen maximal einmal als Blattknoten vorliegen.
Dies wird nun in der Methode „hinzufügen (Blattknoten knoten)“ umgesetzt.

```
private void hinzufügen (Blattknoten knoten)
{
    boolean test = false;
    int index= 0;
    for(int i=0; i<baumliste.size(); i++)
    {
        if(((Blattknoten)baumliste.get(i)).zeichenGeben() == knoten.zeichenGeben())
        {

            test = true;
            index= i;
        }

    }

    if (test == true)
    {
        baumliste.get(index).häufigkeitSetzen(baumliste.get(index).häufigkeitGeben()+1);
        System.out.println("Zeichen "+knoten.zeichenGeben()+
        " existiert bereits im Binärbaum. Häufigkeit wurde angepasst. Neue Häufigkeit: "
        +baumliste.get(index).häufigkeitGeben());

    }
    else
    {
        baumliste.add(knoten);
        System.out.println("Blattknoten "+knoten.zeichenGeben()+ " wurde hinzugefügt");
    }

}
```

Abbildung 10 - Methode hinzufügen ()

Durch eine erneute for-Schleife der Länge der Liste, wird nun die ArrayList komplett nach dem übergebenen Blattknoten durchsucht. Wird eine Übereinstimmung des gesuchten Zeichens mit einem Zeichen aus der ArrayList festgestellt, so wird im if-Fall der for-Schleife der Index des Feldes gespeichert und die Hilfsvariable „test“ wird auf „true“ gesetzt.
Über diese Hilfsvariable wird dann die jeweilige Aktion für den Blattknoten ausgelöst, um im Falle „true“ die Häufigkeit des öfter vorkommenden Zeichens zu erhöhen oder im Falle „false“ den Blattknoten mit dem neuen Zeichen in die ArrayList einzufügen.

Die mehrmals vorkommenden Zeichen gehen hierbei nicht verloren, sondern werden in dem Wert der Häufigkeit ihres eigenen Blattknotens durch dessen Erhöhung gespeichert. Nachdem die Zeichen nun alle in Blattknotenform mit unterschiedlicher Häufigkeit vorliegen, kann der Binärbaum erstellt werden.

3.2 Erstellung des Baumes

Der Baum selbst wird in der Methode „public Baum HuffmanBaumErstellen()“ mithilfe zwei sehr umfangreicher if-Fall-Komplexe erstellt, die abhängig von der Größe der ArrayList sind. Der erste dieser drei Fälle wird ausgelöst, sobald die Liste mehr als zwei Elemente oder Zeichen besitzt. Es wird durch zwei integer-Hilfsvariablen „ikleinsthäuf1“ und „ikleinsthäuf2“ der Index der Zeichen mit den kleinsten Häufigkeiten aus der ArrayList gespeichert.

```
if (baumliste.size()>2)
{   int ikleinsthäuf1 = 0;
    int ikleinsthäuf2;

    // Wenn eine kleinere Häufigkeit als die des Zeichens aus der Liste davor gefunden wird,
    // wird diese gespeichert
    for (int i=0; i< baumliste.size();i++)
    {
        if (baumliste.get(i).häufigkeitGeben()< baumliste.get(ikleinsthäuf1).häufigkeitGeben())
        {
            ikleinsthäuf1 = i;
        }
    }

    //spätes initialisieren von ikleinsthäuf2, da diese nie mit ikleinsthäuf1 übereinstimmen darf
    if (ikleinsthäuf1 == 0)
    {
        ikleinsthäuf2 = 1;
    }
    else
    {
        ikleinsthäuf2 = 0;
    }
```

Abbildung 11 - Methode HuffmanBaumErstellen() Teil 1

Dies ist notwendig, da aus den zwei addierten Kindknoten mit den kleinsten Häufigkeiten immer ein neuer Elternknoten entsteht und somit ein neuer Teilbaum gebildet wird. Es wird zuerst nur die Variable ikleinsthäuf1 mit dem Wert „0“ initialisiert. Damit die beiden Hilfsvariablen nicht den gleichen Wert annehmen, wird ikleinsthäuf2 erst später ein Wert zugewiesen. Die ArrayList wird nun wieder mithilfe einer for-Schleife durchsucht. Der if-Fall innerhalb der Wiederholung ist dabei für das Finden der kleinsten Häufigkeit eines Zeichens aus einem Blattknoten zuständig.
Dies geschieht durch einen einfachen booleschen Vergleich der Häufigkeit aus den Blattknoten der Liste und der Häufigkeit des übergebenen Blattknotens.
Ist die Liste einmal ganz durchlaufen, wird der Index des Zeichens mit der kleinsten Häufigkeit in der lokalen Variable ikleinsthäuf1 gespeichert.

Jetzt wird die zweite Variable ikleinsthäuf2 in Abhängigkeit der vorherigen Hilfsvariable ikleinsthäuf1 deklariert. Da beide nicht auf den gleichen Index verweisen dürfen, wird die Variable mithilfe des if-Falls entweder mit dem Wert 1 oder 0 im Gegensatz zu der Variable ikleinsthäuf1 belegt.

```
for(int i =0;i< baumliste.size();i++)
{
    if(baumliste.get(i).häufigkeitGeben()< baumliste.get(ikleinsthäuf2).häufigkeitGeben()
    && i != ikleinsthäuf1 )
     {
         ikleinsthäuf2 = i;
     }
}

System.out.println("\n"+"Die kleinsten Häufigkeiten haben die Zeichen "
    +baumliste.get(ikleinsthäuf1).zeichenGeben()+" und "+baumliste.get(ikleinsthäuf2).zeichenGeben());
```

Abbildung 12 - Methode HuffmanBaumErstellen() Teil 2

Ähnlich wie beim ersten Mal, wird die for-Schleife nochmal für das Durchsuchen der ArrayList genutzt. Um nun sicher zu gehen, dass die Indices beider Variablen am Ende nicht übereinstimmen, wird dies nochmals als Bedingung im if-Fall verhindert. Nach dem erneuten durchlaufen der Liste wird nun der Index des Zeichens mit der zweitkleinsten Häufigkeit in der Hilfsvariable ikleinsthäuf2 gespeichert und beide Variablen werden zur Veranschaulichung ausgegeben. Da jetzt die zwei kleinsten Häufigkeiten bekannt sind, wird mit ihnen ein neuer Teilbaum gebildet.

```
int häufneu = baumliste.get(ikleinsthäuf1).häufigkeitGeben()+baumliste.get(ikleinsthäuf2).häufigkeitGeben();

//Erstellen des neuen Knoten
Knoten knoten = new Knoten (baumliste.get(ikleinsthäuf1),baumliste.get(ikleinsthäuf2),häufneu);
System.out.println("Das Gewicht des neuen Knotens hat den Häufigkeitswert "+häufneu);

//Nun wird der neue Knoten in die Liste an die Stelle von ikleinsthäuf1 eingefügt und die Referenz zu
//ikleinsthäuf2 gelöscht. Dies geschieht, da das Element aus der Liste gelöscht werden muss.
baumliste.set(ikleinsthäuf1, knoten);
baumliste.remove(ikleinsthäuf2);

//rekursiver Aufruf da, mehrere Teilbäume erstellt werden müssen um den gesamten Baum zu erhalten
BaumAusgeben();
return HuffmanBaumErstellen();
```

Abbildung 13 - Methode HuffmanBaumErstellen() Teil 3

Die kleinsten Häufigkeiten werden zuerst einmal addiert und dann direkt als neue Häufigkeit deklariert. Nun wird ein neuer Elternknoten mit dieser neuen Häufigkeit erstellt, der bei seiner Initialisierung auf seinen linken und rechten Kindknoten verweist. Der Häufigkeitswert wird ausgegeben. Die beiden Blattknoten werden, die jetzt Kindknoten sind werden aus der ArrayList entfernt und an die Stelle des Index der Variable kleinsthäuf1 wird der neue Elternknoten gesetzt.
Dies ist notwendig, da nun die Blattknoten bereits durch ihren Elternknoten in der ArrayList vertreten sind und nur einmal vorkommen dürfen. Zum Schluss wird nochmals der Baum ausgegeben und die Methode ruft sich selbst rekursiv auf, da noch weitere Teilbäume erstellt werden müssen.

Dieser große if-Fall wird nun immer wieder ausgelöst, bis die ArrayList nur noch genau zwei Elemente hat. Hier greift dann der zweite große if-Fall-Komplex ein.

```
if(baumliste.size() == 2)
{
    // Die zwei kleinsten Häufigkeiten brauchen nicht mehr ermittelt werden, da es nur zwei Elemente im Baum gibt
    int ikleinsthäuf1 = 0;
    int ikleinsthäuf2 = 1;

    //Der restliche Prozess bleibt der Gleiche
    System.out.println("Die kleinsten Häufigkeiten haben die Zeichen "
        +baumliste.get(ikleinsthäuf1).zeichenGeben()+" und "+baumliste.get(ikleinsthäuf2).zeichenGeben());

    int häufneu = baumliste.get(ikleinsthäuf1).häufigkeitGeben()+baumliste.get(ikleinsthäuf2).häufigkeitGeben();

    Knoten knoten = new Knoten (baumliste.get(ikleinsthäuf1),baumliste.get(ikleinsthäuf2),häufneu);
    System.out.println("Das Gewicht des neuen Knotens hat den Häufigkeitswert "+häufneu);

    baumliste.set(ikleinsthäuf1, knoten);
    baumliste.remove(ikleinsthäuf2);

    BaumAusgeben();
    return HuffmanBaumErstellen();
}
```

Abbildung 14 - Methode HuffmanBaumErstellen() Teil 4

Dieser zweite If-Fall ist im Grunde genau gleich wie sein Vorgänger aufgebaut. Es fehlen nur die for-Schleifen zum Durchsuchen der ArrayList. Diese werden aber gar nicht mehr benötigt, da sowieso nur noch zwei Elemente in der Liste enthalten sind, die nun die zwei kleinsten Häufigkeiten besitzen. Der neue Knoten kann also direkt aus den zwei beiden übrigen Knoten gebildet werden. Die Häufigkeiten werden wieder addiert und ein neuer Knoten wird gebildet.
Es werden beide Kindknoten aus der ArrayList entfernt und der neue Knoten wieder an den Platz der kleinsten Häufigkeit gesetzt. Die Liste enthält nun nur noch den neuen Knoten. Der Baum wird wieder ausgegeben und die Methode ruft sich zum letzten Mal rekursiv auf.
Der Baum wird schlussendlich ein letztes Mal ausgegeben und der Return-Wert ist der letzte Knoten in der ArrayList, die Wurzel des Baums.

```
BaumAusgeben();
System.out.println(" Huffman-Baum erstellt !!!");
return  baumliste.get(0);
```

Abbildung 15 - Methode HuffmanBaumErstellen() Teil 5

Mithilfe dieses Baumes wird nun die codierte Zeichenkette erstellt.

3.3 Erstellung der komprimierten Zeichenkette

Bei der Codierung wird nun eine HashMap zu Hilfe genommen.
Sie besteht aus Schlüsseln (Keys) und einem jeweils zugeordneten Datensatz.

In dieser Implementierung wird die HashMap zum Zuweisen der Zeichen zu ihren Codewörtern verwendet. Die Keys sind hierbei die Zeichen selbst und die Datensätze der verschlüsselte Binärcode aus „1“ und „0“. Die HashMap selbst wird am Anfang der Klasse importiert und initialisiert. Die hier genannte CharMap ist ein erstelltes Objekt der Klasse HashMap und stellt die Zuordnungsliste dar.

```
public void CharMapErstellen(Baum baum, String weg)
{
    //Wenn Teilbaum ein Blattknoten ist wird der Wert des Blattknotens als Schlüssel gespeichert
    if (baum instanceof Blattknoten)
    {
        charMap.put(String.valueOf(((Blattknoten)baum).zeichenGeben()),weg);
    }

    //  Wenn Teilbaum ein Knoten ist wird für jede Linke Abzweigung eine "1" für den Weg hinzugefügt
    // und für jede rechte Abzweigung eine "0"
    // Da es kein Blattknoten ist, wird die Methode wieder rekursiv aufgerufen
    if(baum instanceof Knoten)
    {
        CharMapErstellen(((Knoten)baum).links, weg + "1");
        CharMapErstellen(((Knoten)baum).rechts, weg + "0");
    }
}
```

Abbildung 16 - Methode CharMapErstellen(Baum baum, String weg)

Bei dieser Methode „CharMapErstellen (Baum baum, String weg)“ werden als Eingabewerte zuerst der zu verzeichnende Baum und das Eingabewort als String verlangt, da diese essentiell für den Start dieser HashMap sind. Der Baum selbst als zu durchsuchendes Verzeichnis und der String als Suchvorgabe.
Die HashMap durchsucht nun den Baum von der Wurzel bis zu den Endknoten und findet dabei entweder Eltern- oder Blattknoten. Wird ein Blattknoten gefunden, so wird direkt im ersten if-Fall sein Zeichen und der bereits durchlaufende Weg in der HashMap gespeichert und eingefügt. Ist aber ein Elternknoten gefunden so gibt es zwei Möglichkeiten. Beide stellen einen rekursiven Selbstaufruf dar, aber unterscheiden sich in ihren übergebenen Eingabewerten. Wenn der gefundene Knoten ein linker Teilbaum, so wird der bereits gespeicherte Weg dieses Knotens um „1“ erweitert, ist der gefundene Knoten aber ein rechter Teilbaum, so wird sein Weg um „0“ erweitert. Dies wird nun für den ganzen Baum und somit jedes für jedes enthaltene Zeichen wiederholt. Es baut sich stückweise ein Verzeichnis der einzelnen Zeichen mit ihren Codewörtern auf, die nur noch ausgegeben werden müssen.
Die Anwendungsbereiche des Huffman-Algorithmus sind durch seine einfache Implementierung sehr vielseitig. Er ist deshalb heutzutage sogar fast in allen gängigen Komprimierungsverfahren zu finden.

4 Heutige Anwendungsbereiche[8]

Der Huffman-Algorithmus wird fast ausschließlich als Hilfsalgorithmus in Verbindung mit anderen Komprimierungsalgorithmen verwendet.
Seine Aufgabe besteht meistens darin, Werte stochastisch auszuwerten und dadurch Redundanzen zu beseitigen.
In der JPEG-Komprimierung beispielsweise ist dieser für die stochastische Analyse einer Matrix des Bildes zuständig und in der MP3-Komprimierung werden mithilfe der statischen Huffman-Codierung die Frequenzen der Audiodatei codiert.
Der Huffman-Algorithmus ist also selbst heute noch aktuell und wird gerne wegen seiner hohen Effizienz verwendet.

5 Weiterführende Gedanken

5.1 Vergleich mit ASCII-Codierung[9]

Warum wird nun aber eine Huffman-Codierung beispielsweise der klassischen, einfachen und hoch-kompatiblen ASCII-Codierung vorgezogen?
Die ASCII-Codierung (American Standard Code for Information Interchange) speichert jedes Zeichen eines Textes mit 8 Bit ab.
Das heißt bei unserem Beispiel: „Komprimieren[]ist[]super!“, mit 23 Stellen würde die ASCII-Datei eine Größe von 23*8 Bit = 184 Bit haben.
Dabei beachtet die ASCII-Codierung aber nicht welche Häufigkeit die einzelnen Zeichen innerhalb der Nachricht haben. Genau hier setzt die Huffman-Codierung an, denn wie schon in der vorher in der Arbeit erläutert, baut der Huffman-Algorithmus auf dieser Häufigkeitsverteilung der einzelnen Zeichen auf.
Durch die Codierung mithilfe des Huffman-Baums wird die ursprüngliche Nachricht nun mit 83 Bit codiert (siehe Kapitel 3.3/ 3.4). Das entspricht ca. 45% der Größe der ASCII-codierten Nachricht und somit ist eine Verminderung des Speicherplatzes von 101 Bit, also 55% zu erreichen!! Der Komprimierungsfaktor beträgt hierbei 2,2 (siehe 2.1 Komprimierungsverfahren).
Diese hohe Effizienz bei der Komprimierung ist der große Vorteil des Huffman-Algorithmus. Die Speicherplatzersparnis ist dabei enorm, in unserem Beispiel wird über die Hälfte des ursprünglich benötigten Speicherplatzes eingespart.

---

[8] Vgl. Wickenburg Sebastian/ Rooch Aeneas/ Groß Johannes (o.J.), S.26 &
Vgl. Stowasser, Stefan (o.J.), S. 7
[9] Vgl. Swisseduc.ch Unterrichtsmaterialien für die Sekundarstufe, oJ., S. 1

Der Huffman-Algorithmus selbst könnte somit eigentlich als ideales Komprimierungsverfahren angesehen werden, vor allem wenn man die heutigen Anwendungsbereiche eines Algorithmus sieht, der schon über 50 Jahre alt ist. Seinen großen Durchbruch hatte der Algorithmus aber nie.

5.2 Kritik am Huffman-Algorithmus[10]

Beim genaueren Analysieren entdeckt man deshalb die kleinen aber entscheidenden Nachteile und Schwächen. Diese sind dafür verantwortlich, dass der Huffman-Algorithmus nie als eigenständiges Komprimierungsverfahren genutzt wurde, sondern heutzutage meistens nur in Verbindung mit einem anderen, fortschrittlicheren Komprimierungsalgorithmus.
Vier der ausschlaggebendsten Schwächen werden im Folgenden erläutert.

Angefangen bei der Tatsache, dass die eigentlich einfache Codierung sehr viel Zeit in Anspruch nimmt. Bei der ursprünglichen Huffman-Codierung wurde nur die statische Codierungsmethode verwendet und die Eingabedaten mussten deshalb immer zweimal gelesen werden. Das erste Mal, um die Häufigkeiten der Zeichen zu ermitteln und den Baum aufzubauen, das zweite Mal, um die Nachricht letztendlich zu codieren. „Das schließt Huffmann zur Echtzeitcodierung von Live-Datenströmen […] aus.“ (Lars Donat 2002, S.5), wie man sie im Internet bei jeder Online-Anwendung findet.
Gelöst wurde dieses Problem allerdings durch das Einführen der dynamischen Huffman-Codierung, bei der der Baum für jede Nachricht immer wieder neu erstellt wird. Jedoch taucht hier ein weiteres Problem auf, denn der Baum muss hier erneut angepasst werden. Das verschlechtert allerdings die Kompressionsrate, da sich der Baum auf die bestimmten Eingabedaten erst einstellen und neu aufbauen muss.
Das Verfahren hat seine Schwächen nicht nur in der Implementierung, sondern auch bei der praktischen Anwendung treten Schwierigkeiten auf.
Oft ist nämlich „Der notwendige Zugriff auf die Bitebene, […]“ (Lars Donat 2002, S.5) nicht möglich oder meistens gar nicht vorgesehen, da die meisten Rechnerarchitekturen auf größere Eingaben spezialisiert sind. Dies muss dann durch aufwendige Rechenoperationen gelöst werden.

Ein weiterer großer Kritikpunkt ist der Speicherplatzverbrauch des Baumes.

[10] Vgl.: Donat 27.11.2002, S.5

Dieser muss nämlich jedes Mal mit den eigentlichen, codierten Daten mitgeschickt werden, damit beim Decodieren die Nachricht auch wieder entschlüsselt werden kann. Würde man den Baum aber nicht mitschicken ist die Decodierung der Nachricht nicht möglich. Dieses Problem betrifft besonders kleine Datenmengen, die codiert und versendet werden, da der Speicherplatz des Baumes im Verhältnis zum Speicherplatz der codierten Datei um ein Vielfaches größer ist. Der größte Nachteil der Huffman-Codierung ist meiner Meinung nach allerdings, dass „Das Verfahren […] auf eine günstige Häufigkeitsverteilung der Eingabewerte angewiesen [ist].“ (Lars Donat 2002, S.5). Das heißt, sobald die Eingabedaten eine ungünstige Häufigkeitsverteilung aufweisen, beispielsweise wenn jedes Zeichen in der Nachricht genau dreimal vorkommt, also die gleiche Häufigkeit besitzt, kann die Huffman-Codierung keinen Erfolg in der Komprimierung vorweisen und ist somit in diesem Fall nicht effektiv.

5.3 Schlusswort

Um nun abschließend auf den Huffman-Algorithmus zu blicken und ihn nur als „Hilfsalgorithmus“ oder nur als „nötige Erweiterung“ konventioneller Komprimierungsverfahren zu sehen, ist meiner Meinung nach falsch. Dieser Algorithmus von David A. Huffman, 1952 erstmals publiziert, ist ein Vorreiter des sich damals im Aufschwung befindlichen Zeitalters der Computer und Informationstechnik. 1952 war dieser Algorithmus bahnbrechend.
Er schaffte es erstmals mithilfe eines Binärbaumes einen optimalen Code zu erzeugen, der verlustfrei und gleichzeitig effizient genug war, um die äußerst schlechten Übertragungsraten der damaligen Zeit zu überwinden und schnelle Nachrichtenübertragung zu gewährleisten. Dieser Algorithmus hat es auch nach über 50 Jahren in unserer schnelllebigen und rasant fortschreitenden Zeit der immer neuen Ideen, Erfindungen und Entwicklungen geschafft, sich in gängigen Komprimierungsverfahren zu etablieren und sich nicht verdrängen zu lassen. Wenn man bedenkt, dass jedes Bild (JPEG, PNG) und jeder Song (MP3) auf unseren Computern oder Smartphones einmal durch dieses Verfahren teil-codiert wurde, zeigt das, wie vielseitig einsetzbar, wichtig und unentbehrlich dieser Algorithmus in unserer heutigen Zeit geworden ist.
Wenn Sie das nächste Mal Musik hören oder die Bilder vom letzten Sommerurlaub anschauen, dann stellen Sie sich einfach vor, dass all diese Sachen wahrscheinlich ohne den Algorithmus von David A. Huffman nicht möglich gewesen wären, so wie es heute der Fall ist.

6. Literaturverzeichnis

- Borys, Thomas/ Urff, Christian (2005): Huffman-Codierung interaktiv - Der Huffman-Algorithmus - Ein Beispiel http://www.ziegenbalg.ph-karlsruhe.de/materialien-homepage-jzbg/cc-interaktiv/huffman/codierung.htm (Stand: 02.11.2018)

- Donat, Lars (2002) Proseminar Datenkompression -Thema: Shannon-Fano und Huffman Verfahren https://www.tu-chemnitz.de/informatik/ThIS/downloads/courses/ws02/datkom/Huffman-ShannonFano.pdf (Stand: 26.10.2018) (Huffman-ShannonFano.pdf)

- Eiden, Wolfgang (1999) Komprimierungsverfahren – Eine Unterrichtsplanung https://kluedo.ub.uni-kl.de/frontdoor/deliver/index/docId/246/file/no_series_239.pdf (Stand: 30.10. 2018) (Huffman 4.pdf)

- Huffman, David (1952): A Method for the Construction of Minimum-Redundancy Codes, in *PROCEEDINGS OF THE I.R.E.*, Volume/Band 40, Issue/Heft 9, S. 1098 – 1101 (Huffman-Original.pdf)

- Jungermann, Florian (2012): Der Huffman Algorithmus – Erklärung und Java Implementierung. YouTube, 29.01.2012, Web, 03.11.2018 um 13:32 Uhr, in: https://www.youtube.com/watch?v=z6cCwiP_yjw

- Liśkiewicz, Maciej/ Fernau, Henning (oJ.) Datenkompression https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf (Stand: 01.11.2018) (Komprimieren 2.pdf)

- Möhring, Rolf (2005) Computerorientierte Mathematik II mit Java http://page.math.tu-berlin.de/~moehring/Coma/Skript-II-Java/coma2.pdf (Stand: 02.11.2018) (Huffman 1.pdf)

- Shannon, Claude / Weaver Warren (1949): THE MATHEMATICAL THEORY OF COMMUNICATION - by Claude E. Shannon and Warren Weaver, Illinois, (Shannon-Weaver.pdf)

- Stowasser, Stefan (o.J.): Proseminar Datenkompression - Audiokompression am Beispiel des MPEG-Layer 3 https://www.tu-chemnitz.de/informatik/ThIS/downloads/courses/ss01/datkom/Audiokompression.pdf (Stand: 02.11.2018) (MP3-Komprimierung.pdf)

- Swisseduc.ch Unterrichtsmaterialien für die Sekundarstufe (oJ.): Huffman-Code https://www.swisseduc.ch/informatik/daten/huffmann_kompression/docs/huffman.pdf (Stand: 30.10. 2018) (Huffman 3.pdf)

- Wickenburg Sebastian/ Rooch Aeneas/ Groß Johannes (2002): Die JPEG-Kompression - Eine Arbeit von Sebastian Wickenburg, Aeneas Rooch und Johannes Groß https://www.mathematik.de/spudema/spudema_beitraege/beitraege/rooch/jpgdruck.pdf (Stand: 02.11.2018) (JPEG-Komprimierung.pdf)

7. Abbildungsverzeichnis

8. Eigenständigkeitserklärung

Erklärung

Hiermit erkläre ich, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

…… , den ……..
(Ort) (Datum)

……
(Unterschrift der Schülerin/des Schülers)