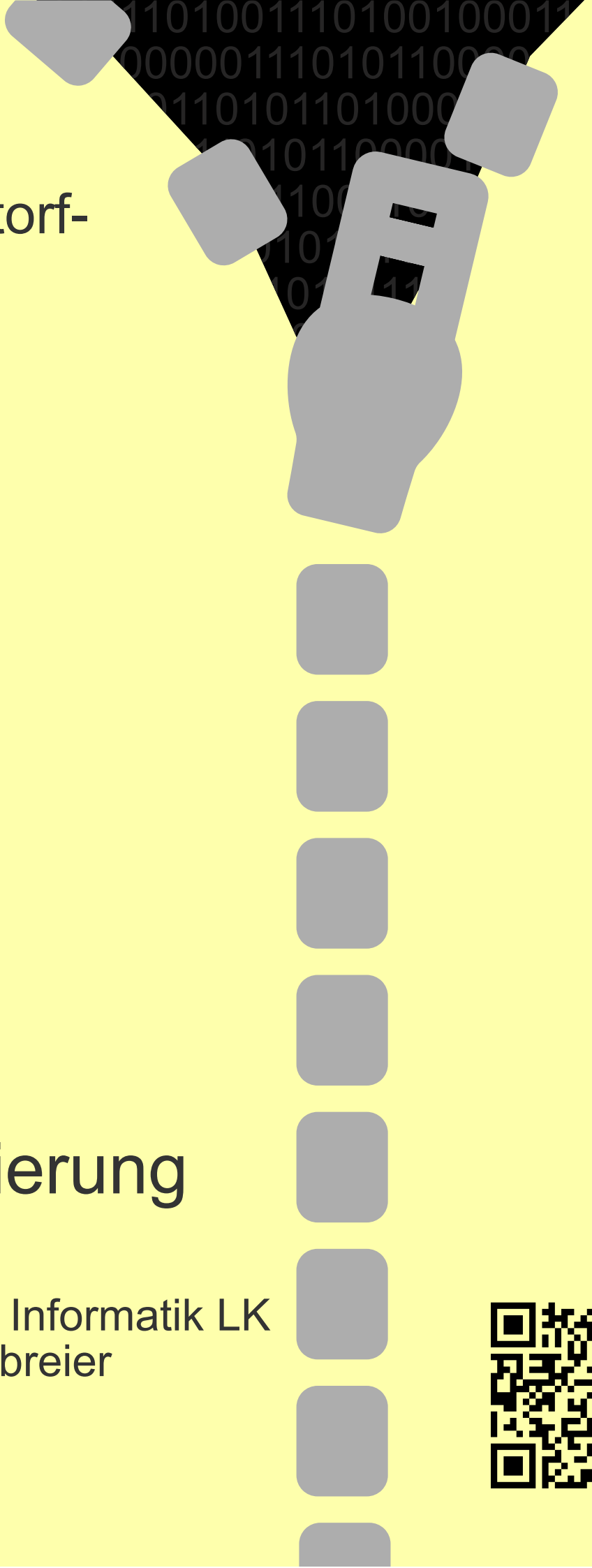


Wilhelm-Hittorf-Gymnasium

Huffman-Komprimierung

Steffen Molitor, Informatik LK
von Herrn Kehlbreier

Q1 2021/2022

Inhaltsverzeichnis

1 Einleitung

In unserer heutigen Welt gibt es eine immense Menge an Daten und sie wird immer mehr. Diese Daten müssen gespeichert, verarbeitet und verschickt werden, weshalb viele Menschen versuchen die Datenmenge so klein, wie möglich zu halten. Dadurch kommt es dazu, dass nahezu jede/r, die/der einen Computer zuhause stehen hat, oder auch einfach ein Smartphone benutzt, schon Kontakt zu einer Form der Datenkomprimierung hatte. Das spiegelt sich beispielsweise in Bildern, oder sogenannten „Zip-Dateien“ wider. Und obwohl, oder gerade weil diese Vorgehensweise etwas so Alltägliches ist, fragt sich niemand, wie genau das denn funktioniert und wieso, trotz der geringeren Speichergröße, die gleichen Daten enthalten sein können. Ich habe mir allerdings genau diese Frage gestellt und möchte somit in dieser Facharbeit etwas Klarheit über die Funktionsweise solcher Komprimierungsalgorithmen schaffen. Genauer gesagt werde ich mich mit der Textkomprimierung anhand des Huffman-Verfahrens beschäftigen und dessen Funktionsweise und Hintergründe erläutern. Gleichwohl man vieles dieses kleinen Teilbereiches auch auf andere Verfahren und Dateiformate anwenden kann, werde ich mich auf die Form des Textes beschränken. Innerhalb der Arbeit war ich dazu in der Lage herauszufinden, warum ASCII und ähnliche Standardformate durch ihren statischen Aufbau eine Menge Speicher verbrauchen und wodurch das bei der Huffman-Komprimierung optimiert wird.

2 Binärbäume

Um die Idee der Huffman-Komprimierung zu verstehen, wird das Prinzip der Binärbäume benötigt. Binärbäume sind eine nicht-lineare Datenstruktur, welche immer leer, oder jeweils einen Binärbaum als linken und rechten Teilbaum haben. Sie können jeweils ein Inhaltsobjekt speichern. Wichtig ist für die Huffman-Komprimierung allerdings, dass so sogenannte „Paths“, oder auf Deutsch Pfade, entstehen, die man binär, also durch verschiedene Abfolgen von „0“ und „1“ darstellen kann. Diese Pfade bilden sich dadurch, dass ein Schritt zu einem linken Teilbaum durch eine „0“ und ein Schritt zu einem rechten Teilbaum durch eine „1“ dargestellt wird. So wird in Abbildung 1 beispielsweise der Pfad „01“ dargestellt.

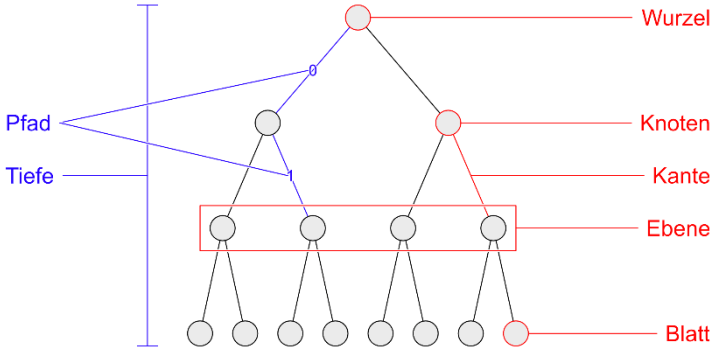


Abbildung 1 - Aufbau eines Binärbaums

3 Grundidee und Geschichte

3.1 Geschichte

Die Huffman Kodierung wurde von David Albert Huffman erfunden. Dies geschah im Zuge seiner Seminararbeit am MIT. Dabei schaffte es Huffman den mittlerweile über 70 Jahre langen effizientesten Komprimierungsalgorithmus zu entdecken und damit seinen eigenen Professor Robert Mario Fano zu übertrumpfen. Jener war zu dem Zeitpunkt Mitbegründer der „Shannon-Fano-Kodierung“ und arbeitete selbst an einer Lösung, um einen noch effizienteren Algorithmus zu entwickeln. Dabei verschwieg dieser auch seinen Studenten, dass die Fragestellung der Arbeit, die optimale und effizienteste Kodierung zu finden, noch offen war. Das könnte mit unter anderem der Grund dafür sein, dass Huffman nah an der Entscheidung war, die Arbeit aufzugeben. Doch glücklicherweise kam dieser auf die Idee einen Binärbaum von unten nach oben aufzubauen, was einer optimierten Version der „Shannon-Fano-Kodierung“ nahekam, welcher von oben nach unten aufgebaut wurde.

3.2 Grundidee

Die Grundidee der Huffman-Komprimierung besteht darin, Daten möglichst platzsparend ohne Datenverluste zu speichern. Dabei bedient sich der Algorithmus des Verfahrens der Häufigkeitsanalyse, womit er sich, anders als beispielsweise ASCII, automatisch an jeden Text spezifisch anpasst. Dabei wird ein Binärbaum aufgebaut, bei dem die häufigsten Zeichen die kürzesten Pfade besitzen, sich also am weitesten oben im Baum befinden. Wichtig ist dabei, dass die inneren Knoten unbesetzt bleiben, Zeichen also nur in den Blättern gespeichert werden. Das garantiert die Präfixfreiheit, welche für die Längenunterschiede der Codes essenziell ist. Sobald der beschriebene Binärbaum aufgebaut wurde, wird auf dessen Basis eine Kodierungstabelle erstellt, in welcher für jedes im Text vorkommende Zeichen ein Code vermerk ist. Diese Tabelle wird dann zuallererst zu dem Kodieren des Textes genutzt und dann in Form eines sogenannten „Headers“, oder auf Deutsch einer Kopfzeile, vor den kodierten Text gehangen. Das Dekodieren geschieht dann über die aus dem Header ausgelesene Tabelle. Dafür muss das Headerprotokoll, in dem der Aufbau des Headers beschrieben wird, im Voraus beiden Seiten klar sein, was allerdings kein Problem ist, da es sich nicht um eine Verschlüsselung handelt.

4 Kodieren und Dekodieren an einem Beispiel

Im Folgenden werde ich anhand des Ausdruckes „Informatikunterricht“ zuerst eine Kodierung nach dem Huffman Prinzip und dann eine Dekodierung des dadurch entstandenen Codes vornehmen.

4.1 Kodieren

Zuallererst wird eine Häufigkeitsanalyse des Ausdruckes erstellt. Dafür wird jedes vorkommende Zeichen gezählt und in eine Tabelle eingefügt. So ergibt sich die nebenstehende Tabelle aus Abbildung 2.

Zeichen	Anzahl
I	1
n	2
f	1
o	1
r	3
m	1
a	1
t	3
i	2
k	1
u	1
e	1
c	1
h	1

Abbildung 2 - Häufigkeitstabelle

Aus dieser Tabelle kann man nun jeweils kleine Teilbäume bilden, wobei immer die beiden Elemente mit der geringsten Wertigkeit zusammengefügt werden. Die entstandenen Teilbäume ersetzen dann mit

der zusammengerechneten Wertigkeit die beiden untergeordneten Elemente in der Liste. Auf die Weise funktioniert auch der Ablauf in Abbildung 3. Dort wurde nun ein recht gleichmäßiger Binärbaum aufgebaut, allerdings kann man an den Zeichen „r“ und „t“ auch gut erkennen, dass sie als häufigste Zeichen auch einen kürzeren Pfad haben. Um den Binärbaum nun zum Kodieren des Ausdruckes zu verwenden, müssen die Pfade anhand von 0 und 1 in einer Tabelle erfasst werden. Diese Tabelle findet sich in Abbildung 4 wieder. Der letzte Schritt zum vollständig kodierten Text besteht darin, die einzelnen Buchstaben durch ihre Codes zu ersetzen. So ist der Ausdruck jetzt allein durch 0 und 1 darstellbar und lautet „00001110000100101010011010011011110101011011101100111101101111100010011110".

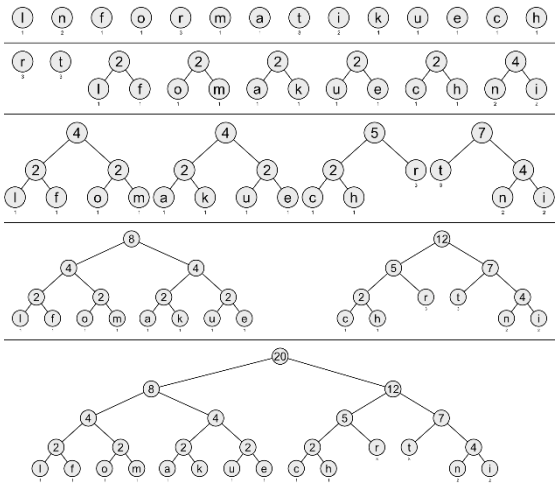


Abbildung 3 - Aufbau des Binärbaums (Vergrößerte Grafik im Anhang bei 9.3)

Zeichen	Code
l	0000
n	1110
f	0001
o	0010
r	101
m	0011
a	0100
t	110
i	1111
k	0101
u	0110
e	0111
c	1000
h	1001

Abbildung 4 - Codetabelle

Das Dekodieren eines mit Huffman komprimierten Ausdruckes geht, sofern man die Codetabelle hat, in einem einzigen Schritt. Dieser Schritt besteht einzig und allein darin den letzten der Komprimierung zu revidieren, also die Zahlen von links nach rechts durchgehen und sobald die Reihenfolge auf einen Code zutrifft, durch dessen dazugehöriges Zeichen zu ersetzen. So werden beispielsweise die ersten vier Stellen des komprimierten Ausdruckes durch „l“ ersetzt, da keine der drei vorherigen Codefolgen in der Codetabelle existiert. So entsteht aus dem Code nun wieder der ursprüngliche Ausdruck „Informatikunterricht“.

4.2 Dekodieren

5 Modellierung und Programmierung

Das ganze Verfahren kann man auch Programmieren, wofür ich im Folgenden ein Modell entwickeln werde. Als Basis der Modellierung wird eine Klasse „Huffman“ genutzt. Diese Klasse benötigt eine Methode zum Komprimieren und eine zum

Dekomprimieren. Während der Programmierung kam dazu noch eine Methode, um die Codes aus dem Binärbaum auszuwerten, da diese rekursiv programmiert werden muss. Die Modellierung der gesamten Klasse ist rechts in Abbildung 5 zu sehen. Um die Codes zusammen mit den dazugehörigen Zeichen zu speichern ist es naheliegend eine Liste zu nutzen. Da die Liste diese beiden Werte zusammenhängend speichern muss, wird auch eine Klasse für Inhaltsobjekte benötigt. Diese Klasse muss nicht mehr können, als einen Code als String und ein Zeichen als char zu speichern. Dazu werden noch passende Getter-Methoden benötigt. Um die spätere Benutzung der Liste zu vereinfachen kann man eine Klasse bauen, welche die Liste nutzt, aber auch an das Wertepaar Code und Zeichen angepasst ist. Also eine Methode, um einen Code in ein Zeichen umzuwandeln und umgekehrt, eine, um ein Zeichen hinzuzufügen und eine, für die manuelle Benutzung, um auf die Liste direkt zuzugreifen. Diese Konstruktion kann man in Abbildung 6 sehen. Aber am wichtigsten ist die Klasse Binärbaum, welcher ein Zeichen und eine Wertigkeit speichert. Die Methoden, die dazu passen sind die getter-Methoden und eine Methode, um den Wert um einen angegebenen Wert zu erhöhen und, für die Einfachheit, einen um den Wert um genau eins zu erhöhen. Auch diese Kombination ist rechts in Abbildung 7 dargestellt. Das gesamte Modell ergibt zusammengefügt die Abbildung 8. Dieses Modell lässt sich nun in Java umsetzen. Angefangen mit der Hauptklasse „Huffman“ in

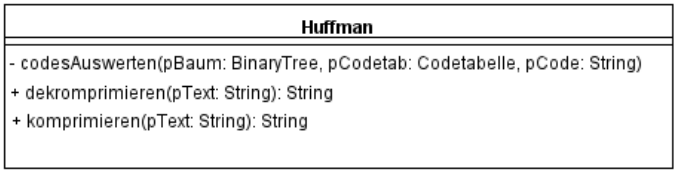


Abbildung 5 - Die Klasse "Huffman"

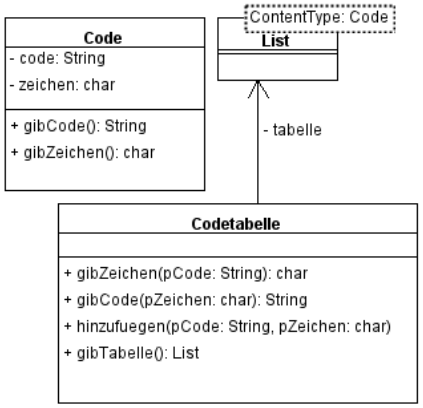


Abbildung 6 - Klassenkonstrukt der Codetabelle

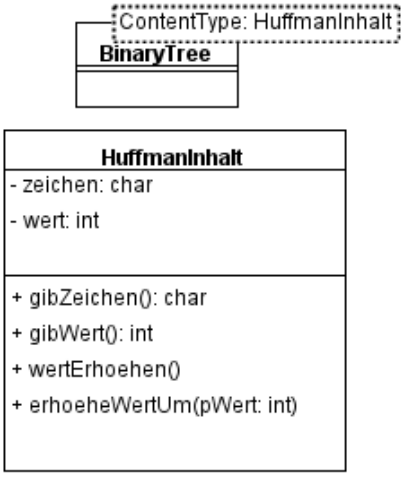


Abbildung 7 - Binärbaum-Konstrukt

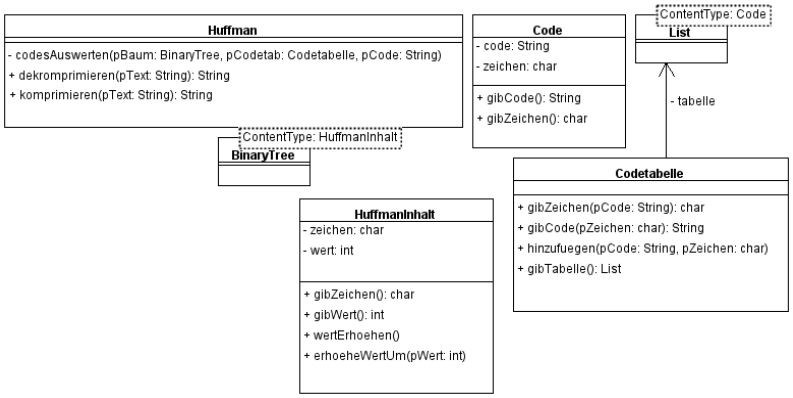


Abbildung 8 - Ganzes Modell

der „komprimieren“-Methode. Dort muss als erstes die Häufigkeitsanalyse vorgenommen werden. Als erstes wird dafür eine Liste, die später auch für die erweiterten Bäume genutzt werden kann, vom Inhaltstyp BinaryTree, welche wiederum den Inhaltstyp HuffmanInhalt haben, benötigt. Nachfolgend wird dann durch den Parameter pText iteriert, wobei jedes Mal nach der Stelle des Buchstabes in der Liste gesucht wird. Falls dabei keine Stelle gefunden wird, wird ein neuer Buchstabe mit der Wertigkeit 1 in die Liste eingefügt, andernfalls wird die Wertigkeit des bestehenden Buchstabens um 1 erhöht. Dieser ganze Abschnitt sieht dann wiefolgt aus:

```
1.  List<BinaryTree<HuffmanInhalt>> liste = new List<BinaryTree<HuffmanInhalt>>();
2.
3.  for(int i = 0; i < pText.length(); i++) {
4.    liste.toFirst();
5.    while(liste.hasAccess() && liste.getContent().getContent().gibZeichen() !=
    pText.charAt(i)) {
6.        liste.next();
7.    }
8.    if(!liste.hasAccess()) {
9.      liste.append(new BinaryTree<HuffmanInhalt>(new HuffmanInhalt(pText.charAt(i), 1)));
10.   } else {
11.     liste.getContent().getContent().wertErhoehen();
12.   }
13. }
```

Ich gehe hierbei nicht mehr auf die Hilfsklassen und -Methoden ein, da diese nur unspannende und für die (De-)Komprimierung irrelevante Algorithmen enthalten. [1]

Der nächste Schritt besteht darin, jeweils zwei Bäume der Liste zusammenzufügen und das so lange, bis nur noch zwei Bäume übrig sind die dann an die Wurzel, welche eine eigene Variabel bekommt, angeschlossen werden können. Dafür wird in einer While-Schleife, welche so lange läuft, wie die Bedingung, dass entweder mehr als zwei Bäume in der Liste sind oder genau zwei Bäume in der Liste sind, aber auch noch ein Baum in einem unfertigen Baum innerhalb der Schleife, der Baum mit der kleinsten Wertigkeit herausgefiltert. Über eine weitere Variabel, die zwischen eins und null wechselt, wird ermittelt, ob dieser Baum als linker oder rechter Teilbaum eingefügt werden soll. Sobald nur noch exakt zwei Bäume in der Liste sind und kein weiterer Baum angefangen wurde, sollen die beiden verbliebenden an die Wurzel angefügt werden. Dieser Abschnitt lässt sich, wie im Anhang unter 9.4 in den Zeilen 24 bis 98 geschehen, programmieren. Danach lässt sich bereits die Codetabelle erstellen. Dafür wird als erstes eine neue Instanz der Klasse „Codetabelle“ erstellt und dann an die Methode „codesAuswerten“ übergeben. Diese Methode füllt die Codetabelle dann über die Daten des Binärbaumes.

[1] Der ganze Quellcode inklusive Hilfsklassen/-Methoden befindet sich im Anhang unter 9.4

Dabei arbeitet sie rekursiv, indem sie durch den Binärbaum traversiert. Für diesen Anwendungsfall wird dabei eine Pre-Order-Traversierung benötigt, da erst geprüft werden muss, ob der aktuelle Baum ein Zeichen enthält, denn das ist auch der Rekursionsanker, weil in diesem Fall keine weiteren Buchstaben weiter unterhalb gefunden werden können. Die ganze Methode sieht dann folgendermaßen aus:

```
1.  static private void codesAuswerten(BinaryTree<HuffmanInhalt> pBaum, Codetabelle pCodetab,
    String pCode) {
2.    if(!pBaum.isEmpty()) {
3.      if(pBaum.getContent().gibZeichen() != Character.MIN_VALUE) {
4.        pCodetab.hinzufuegen(pCode, pBaum.getContent().gibZeichen());
5.      } else {
6.        codesAuswerten(pBaum.getLeftTree(), pCodetab, pCode + "0");
7.        codesAuswerten(pBaum.getRightTree(), pCodetab, pCode + "1");
8.      }
9.    }
10. }
```

Jetzt könnte der angegebene Text schon komprimiert werden, allerdings fehlt für die Übertragung noch der Header. Dafür kann man in Abbildung 9 den Aufbau des verwendeten Headers sehen, wobei sich die inneren drei Elemente für jedes Zeichen wiederholen. Für das erste Element des Headers wird der längste Code ermittelt und die Stellen dessen Länge als Binärzahl vorne als Nullen, bis auf die letzte, welche durch eine Eins dargestellt wird, an den Header gehangen. Danach wird für jedes Zeichen die binäre Darstellung der Codelänge, dann der Code selbst und dann das Zeichen als 8-Bit ASCII-Code angehangen. Sobald sich alle Zeichen im Header befinden, wird dieser über die Codelängenangabe 0 geschlossen. Auch diese Programmierung findet sich im Anhang unter 9.4 innerhalb der Zeilen 104 bis 154. Das Einzige, was jetzt noch für die Komprimierung fehlt, ist das tatsächliche Umschreiben des Textes, welche in eine simplen For-Schleife für jedes Zeichen geschieht. Dafür wird die Methode „giCode(char pZeichen)“, welche den Code für das angegebene Zeichen ausgibt, der Klasse Codetabelle genutzt. Die Umsetzung sieht dann so aus:

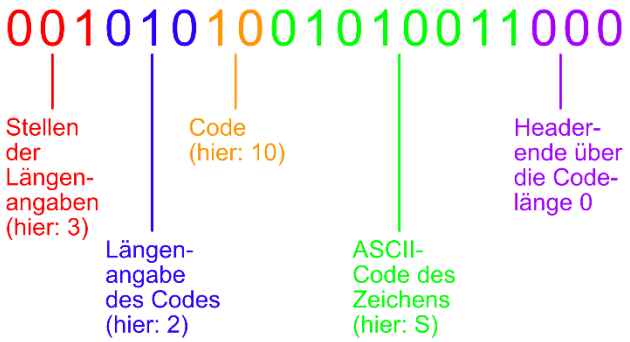


Abbildung 9 - Headeraufbau

```
1.  String finalerText = "";
2.  for (int i = 0; i < pText.length(); i++) {
3.    finalerText += codes.gibCode(pText.charAt(i));
4.  }
```

Als allerletztes wird jetzt der Header mit dem angehängten umgeschriebenen Text ausgegeben.

6 Beurteilung und Abgleich mit anderen Verfahren

6.1 Morsecode

TEXT

7 Schlussfolgerung

TEXT

8 Literatur- und Quellenverzeichnis

Huffman-Kodierung - Wikipedia. (13. Dezember 2021). Von Wikipedia: https://de.wikipedia.org/wiki/Huffman-Kodierung abgerufen

Kempe, T., Löhr, A., Grimm, R., & Scholle, O. (2015). *Informatik 2.* Schöningh.

Pivikina, I. (Juli 2015). *Discovery of Huffman Codes | Mathematical Association of America*. Von Mathematical Association of America: https://www.maa.org/press/periodicals/convergence/discovery-of-huffman-codes abgerufen

Stack Overflow. (kein Datum). Von Stack Overlow: https://stackoverflow.com/ abgerufen

9 Anhang

9.1 Erklärung über die selbstständige Anfertigung der Arbeit

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe.

Darüber hinaus versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken oder aus dem Internet als solche kenntlich gemacht habe.

Ich erkläre mich damit einverstanden, dass die von mir verfasste Facharbeit Dritten zugänglich gemacht werden darf.

_____ , den _____ _____

Ort Datum Unterschrift

9.2 Programmierung als Vorbereitung auf die Thematik

```
1.  #include <stdlib.h>
2.  #include <iostream>
3.  #include <string>
4.  #include <istream>
5.  #include <map>
6.  #include <bitset>
7.
8.  using namespace std;
9.
10.
11. struct Node {
12.     char data;
13.     int weight;
14.     struct Node* left;
15.     struct Node* right;
16.     struct Node* parent;
17.
18.     Node(char val, int wei) {
19.         data = val;
20.         weight = wei;
21.         left = NULL;
22.         right = NULL;
23.         parent = NULL;
24.     }
25.     Node() {
26.         data = NULL;
27.         weight = 0;
28.         left = NULL;
29.         right = NULL;
30.         parent = NULL;
31.     }
32. };
33.
34. string reverse(string pText) {
35.     string result = "";
36.     for (int i = pText.length() - 1; i >= 0; i--) {
37.         result += pText[i];
38.     }
39.     return result;
40. }
41. string cutZeros(string pText) {
42.     string result = "";
43.     int i = 0;
44.     while (pText[i] == '0') {
45.         i++;
46.     }
47.     while (i < pText.length()) {
48.         result += pText[i];
49.         i++;
50.     }
51.     return result;
52. }
53. string cutZeros(string pText, bool keep) {
54.     string result = "";
55.     int i = 0;
56.     while (pText[i] == '0' && i < pText.length() - 1) {
57.         i++;
58.     }
59.     while (i < pText.length()) {
60.         result += pText[i];
61.         i++;
62.     }
63.     return result;
64. }
65. string cutTo(string pText, int pLength) {
66.     string result = "";
67.     int i = 0;
68.     while (i < pText.length() - pLength) {
69.         i++;
70.     }
71.     while (i < pText.length()) {
72.         result += pText[i];
73.         i++;
74.     }
75.     return result;
76. }
77. int binToDec(string bin) {
78.     int result = 0;
79.     for (int i = 0; i < bin.length(); i++) {
80.         if (bin[i] == '1') {
81.             result += pow(2, bin.length() - 1 - i);
```

```
82.         }
83.     }
84.     return result;
85. }
86.
87. int main()
88. {
89.     cout << "Umsetzung der Huffman-Komprimierung als Vorbereitung auf die Facharbeit von Steffen Molitor
    (Angepasste Version fuer die Facharbeit)\n\n";
90.     string kd;
91.     string text;
92.     struct Node* root = new Node();
93.     do {
94.         cout << "Komprimieren, dekomprimieren, oder Fenster schliessen? k/d/e\n";
95.         getline(cin, kd);
96.         if (kd == "k") {
97.             cout << "Zu komprimierender Text: ";
98.             getline(cin, text);
99.             map<int, Node*> charMap;
100.            for (string::size_type i = 0; i < text.size(); i++) {
101.                int found = charMap.size();
102.                for (string::size_type j = 0; j < charMap.size(); j++) {
103.                    if (charMap[j]->data == (char)text.at(i)) {
104.                        found = j;
105.                    }
106.                }
107.                if (found == charMap.size()) {
108.                    charMap[found] = new Node(text.at(i), 1);
109.                }
110.                else {
111.                    charMap[found]->weight = charMap[found]->weight + 1;
112.                }
113.            }
114.            const int chars = charMap.size();
115.            bool flag = true;
116.            int zo = 1;
117.            struct Node* topNode = new Node();;
118.            int lowest;
119.            int parentless;
120.            if (chars > 2) {
121.                while (flag) {
122.                    flag = false;
123.                    lowest = -1;
124.                    parentless = 0;
125.                    for (string::size_type i = 0; i < charMap.size(); i++) {
126.                        if (charMap[i]->parent == NULL) {
127.                            if (lowest == -1 || charMap[i]->weight < charMap[lowest]->weight) {
128.                                lowest = i;
129.                            }
130.                            parentless++;
131.                        }
132.                    }
133.                    if (zo == 1) {
134.                        zo = 0;
135.                        topNode = new Node();
136.                    }
137.                    else {
138.                        zo = 1;
139.                    }
140.                    if (zo == 0) {
141.                        topNode->left = charMap[lowest];
142.                        topNode->weight += charMap[lowest]->weight;
143.                        charMap[lowest]->parent = topNode;
144.                        parentless--;
145.                    }
146.                    else {
147.                        topNode->right = charMap[lowest];
148.                        topNode->weight += charMap[lowest]->weight;
149.                        charMap[lowest]->parent = topNode;
150.                        charMap[charMap.size()] = topNode;
151.                    }
152.                    if (parentless > 2 || parentless == 2 && zo == 0) {
153.                        flag = true;
154.                    }
155.                }
156.            }
157.            zo = 0;
158.            for (string::size_type i = 0; i < charMap.size(); i++) {
159.                if (charMap[i] != NULL && charMap[i]->parent == NULL) {
160.                    if (zo == 0) {
161.                        root->left = charMap[i];
162.                        charMap[i]->parent = root;
163.                        zo = 1;
164.                    }
165.                    else {
166.                        root->right = charMap[i];
```

```
                        charMap[i]->parent = root;
                    }
                }
            }

            map<int, string[2]> codes;
            map<char, string> codetable;
            string code;
            int longest = 0;
            for (int i = 0; i < chars; i++) {
                codes[i][0] = charMap[i]->data;
                code = "";
                struct Node* temp = charMap[i];
                struct Node* parent;
                while (temp->parent != NULL || temp->parent == root) {
                    parent = temp->parent;
                    if (parent->left == temp) {
                        code += "0";
                    }
                    else {
                        code += "1";
                    }
                    temp = parent;
                }
                code = reverse(code);
                codes[i][1] = code;
                codetable[charMap[i]->data] = code;
                if (code.length() > longest) {
                    longest = code.length();
                }
            }
            string header;
            string binlongest = bitset<8>(longest).to_string();
            binlongest = cutZeros(binlongest);
            for (int i = 1; i < binlongest.length(); i++) {
                header += "0";
            }
            header += "1";

            for (string::size_type i = 0; i < codes.size(); i++) {
                header += cutTo(bitset<8>(codes[i][1].length()).to_string(), binlongest.length());
                header += codes[i][1];
                header += bitset<8>((int)codes[i][0][0]).to_string();
            }
            for (int i = 0; i < binlongest.length(); i++) {
                header += "0";
            }
            string finaltext = "";
            for (int i = 0; i < text.length(); i++) {
                finaltext += codetable[text.at(i)];
            }

            cout << "Komprimierter Text: " << header << finaltext << "\n\n" << endl;
        }
        else if (kd == "d") {
            cout << "Zu dekomprimierender Text: ";
            getline(cin, text);
            int lenlen = 0;
            int i = 0;
            while (text[i] == '0') {
                i++;
            }
            lenlen = i + 1;
            string zeroEnd = "";
            for (int j = 0; j < lenlen; j++) {
                zeroEnd += "0";
            }
            bool flag = true;
            string codelen = "";
            string code = "";
            map<string, char> chartable;
            while (flag) {
                code = "";
                codelen = "";
                for (int j = 0; j < lenlen; j++) {
                    i++;
                    codelen += text[i];
                }
                if (codelen != zeroEnd) {
                    for (int j = 0; j < binToDec(codelen); j++) {
                        i++;
                        code += text[i];
                    }
                    string charNum = "";
                    for (int j = 0; j < 8; j++) {
                        i++;
```

```
253.                         charNum += text[i];
254.                     }
255.                     chartable[code] = binToDec(charNum);
256.                 }
257.                 else {
258.                     flag = false;
259.                 }
260.             }
261.             i++;
262.             string dektext = "";
263.             code = "";
264.             for (int j = i; j < text.length(); j++) {
265.                 code += text[j];
266.                 if (chartable[code] != NULL) {
267.                     dektext += chartable[code];
268.                     code = "";
269.                 }
270.             }
271.             cout << "Dekomprimierter Text: " << dektext << "\n\n" << endl;
272.         }
273.         else if (kd != "e") {
274.             cout << "\n\nFalsche Eingabe\n\n";
275.             kd;
276.         }
277.     } while (kd != "e");
278. }
```

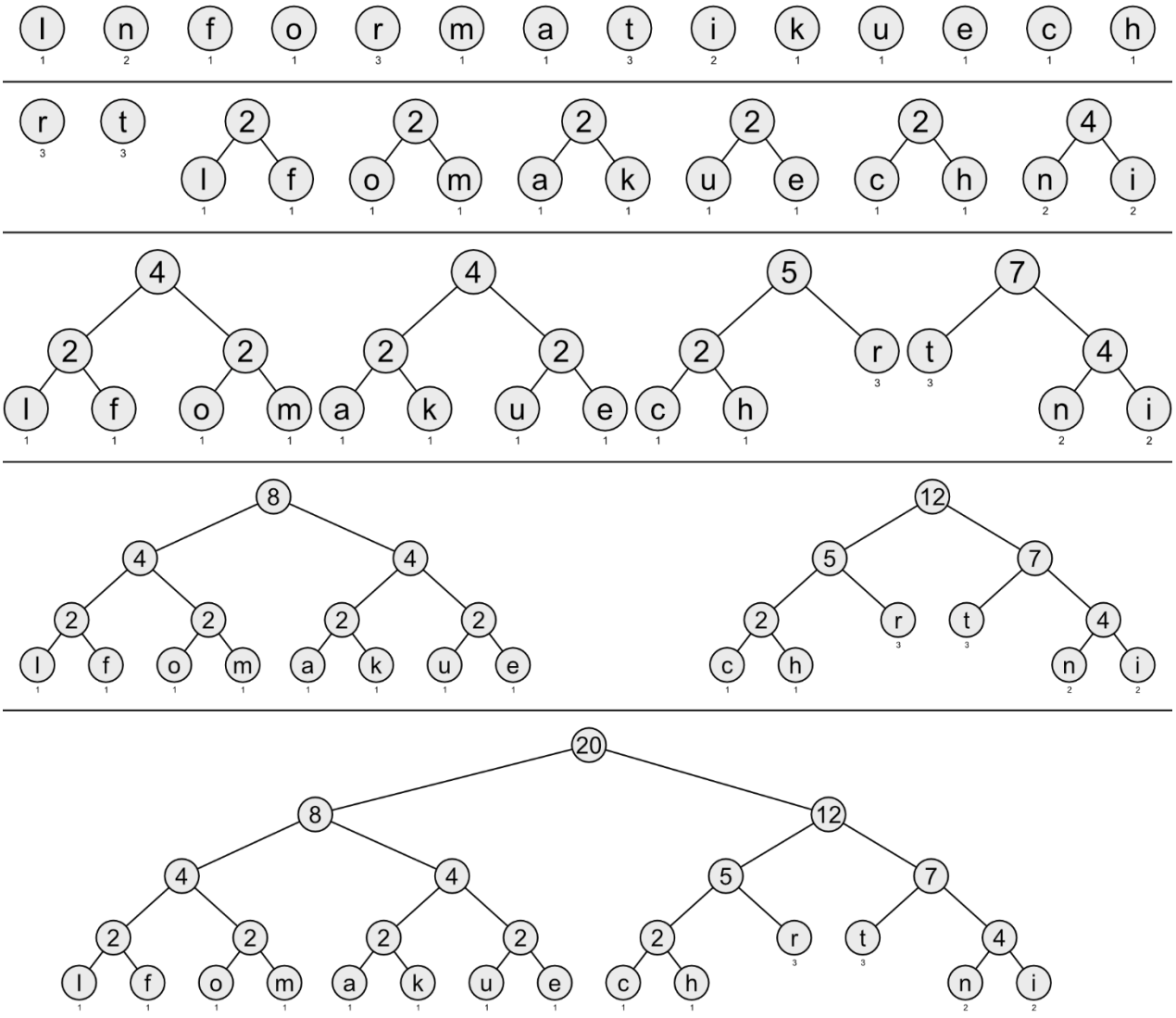
Hinweise:

Der Code wurde in der Programmiersprache C++ in Microsoft Visual Studio verfasst und dort ohne Fehlermeldung, allerdings mit einigen Warnungen, kompiliert.

Ich habe die Vorbereitung bewusst in einer mir wenig bekannten Programmiersprache umgesetzt, um mich intensiver mit dem Thema zu befassen, was allerdings größere Fehler, oder Ungewöhnlichkeiten zur Folge haben kann.

Da es sich um reine Vorbereitungsarbeit handelt, ist der Code nicht dokumentiert und auch nicht grundlegend bereinigt worden.

9.3 Vergrößerte Grafik des Binärbaumaufbaus



9.4 Java-Implementierung

Hauptklasse:

```
1.  public class Huffman
2.  {
3.    //Methode für die Komprimierung eines Textes
4.    static public String komprimieren(String  pText){
5.      BinaryTree<HuffmanInhalt> wurzel = new BinaryTree<HuffmanInhalt>(new HuffmanInhalt()); //Spätere Wurzel
    des Codebaums
6.      List<BinaryTree<HuffmanInhalt>> liste = new List<BinaryTree<HuffmanInhalt>>(); //Liste der Teilbäume
7.
8.      /*
9.       * Teil für die Häufigkeitsanalyse
10.      * (Speichert Buchstaben mit ihrer Wertigkeit in Binärbäumen in der Liste)
11.      */
12.     for(int i = 0; i < pText.length(); i++) {
13.       liste.toFirst();
14.       while(liste.hasAccess() && liste.getContent().getContent().gibZeichen() != pText.charAt(i)) {
15.           liste.next();
16.       }
17.       if(!liste.hasAccess()) {
18.         liste.append(new BinaryTree<HuffmanInhalt>(new HuffmanInhalt(pText.charAt(i), 1)));
19.       } else {
20.         liste.getContent().getContent().wertErhoehen();
21.       }
22.     }
23.
24.     /*
25.      * Teil um die Binärbäume in einem gemeinsamen Baum zusammenzufassen
26.      * (Nimmt immer die beiden kleinsten Binärbäume, um sie in einen Binärbaum zu verlagern -
27.      * beide Teilbäume werden aus der Liste gelöscht und durch den neuen Baum ersetzt)
28.      */
29.     int anzahl = liste.count();  //Anzahl der verschiedenen Buchstaben
30.     boolean flag = true; //Abbruchbedingung
31.     int zo = 1; //Wert der zwischen 1 und 0 wechselt um anzugeben, welcher der beiden Teilbäume gesetzt wird
32.     BinaryTree<HuffmanInhalt> aktuellerBaum = new BinaryTree<HuffmanInhalt>(); //Baum dem untergeordnete
    Bäume zugewiesen werden
33.     int kleinster; //Stelle in der Liste mit dem Binärbaum bin der geringsten Wertigkeit
34.     int kleinsterWert = 6205; //Geringste Wertigkeit
35.     if(anzahl > 2) {
36.       while(flag) {
37.         flag = false;
38.         kleinster = -1;
39.         liste.toFirst();
40.
41.         //Findet den Binärbaum mit der geringsten Wertigkeit
42.         for(int i = 0; i < liste.count(); i++) {
43.           if (liste.getContent() != null && (kleinster == -1 || liste.getContent().getContent().gibWert() <
    kleinsterWert)) {
44.             kleinster = i;
45.             kleinsterWert = liste.getContent().getContent().gibWert();
46.           }
47.           liste.next();
48.         }
49.
50.         //Prüft ob ein neuer übergeordneter Baum erstellt werden muss und schaltet den Indikator (zo) um
51.         if(zo == 1) {
52.           zo = 0;
53.           aktuellerBaum = new BinaryTree<HuffmanInhalt>(new HuffmanInhalt());
54.         } else {
55.           zo = 1;
56.         }
57.
58.         //Fügt in den neuen Baum ein
59.         if(zo == 0) {
60.           //Findet den Teilbaum mit der geringsten Wertigkeit über den Index kleinster wieder
61.           liste.toFirst();
62.           for(int i = 0; i < kleinster; i++) {
63.             liste.next();
64.           }
65.           BinaryTree<HuffmanInhalt> kleinstesZeichen = liste.getContent();
66.
67.           liste.remove(); //Entfernt diesen Binärbaum aus der Liste
68.           aktuellerBaum.setLeftTree(kleinstesZeichen); //Setzt den Baum anstattdessen als linken Teilbaum des
    neuen Baumes
69.           aktuellerBaum.getContent().erhoeheWertUm(kleinstesZeichen.getContent().gibWert()); //Überträgt die
    Wertigkeit des Teilbaumes auf den neuen Baum
70.         } else {
71.           //Findet den Teilbaum mit der geringsten Wertigkeit über den Index kleinster wieder
72.           liste.toFirst();
73.           for(int i = 0; i < kleinster; i++) {
```

```
74.             liste.next();
75.           }
76.           BinaryTree<HuffmanInhalt> kleinstesZeichen = liste.getContent();
77.
78.           liste.remove(); //Entfernt diesen Binärbaum aus der Liste
79.           aktuellerBaum.setRightTree(kleinstesZeichen); //Setzt den Baum anstattdessen als rechten Teilbaum
    des neuen Baumes
80.           aktuellerBaum.getContent().erhoeheWertUm(kleinstesZeichen.getContent().gibWert()); //Überträgt die
    Wertigkeit des Teilbaumes auf den neuen Baum
81.           liste.append(aktuellerBaum); //Fügt den neuen Baum in die Liste ein, da er jetzt vollständig ist
82.         }
83.
84.         /*
85.          * Testet ob ein weiterer Durchlauf nötig ist, also noch mehr als zwei freie Bäume vorhanden
86.          * sind, oder genau zwei, allerdings während schon ein neuer Baum begonnen wurde
87.          */
88.         if (liste.count() > 2 || liste.count() == 2 && zo == 0) {
89.           flag = true;
90.         }
91.       }
92.     }
93.
94.     //Fügt die beiden verbleibenden Teilbäume an die Wurzel an
95.     liste.toFirst();
96.     wurzel.setLeftTree(liste.getContent());
97.     liste.next();
98.     wurzel.setRightTree(liste.getContent());
99.
100.    //Erstellt eine Codetabelle
101.    Codetabelle codes = new Codetabelle();
102.    codesAuswerten(wurzel, codes, "");
103.
104.    //Findet die Länge des längsten Codes heraus
105.    int laenge = 0; //Länge des längsten Codes (Für den Header)
106.    codes.gibTabelle().toFirst();
107.    while(codes.gibTabelle().hasAccess()) {
108.      if(codes.gibTabelle().getContent().gibCode().length() > laenge) {
109.        laenge = codes.gibTabelle().getContent().gibCode().length();
110.      }
111.      codes.gibTabelle().next();
112.    }
113.
114.    //Erstellung des Headers
115.    String header = ""; //Variabel für den Header
116.    String binlongest = Integer.toBinaryString(laenge); //Binäre Darstellung der längsten Codewortlänge
117.
118.    /*
119.     * Fügt für jede Stelle, bis auf die letzte, der binären Länge eine 0 und dann eine 1 in den Header ein
120.     * (Das ermöglicht es die Länge der Codelängen im Header anzugeben ohne sich auf eine bestimmte Länge
121.     * zu beschränken)
122.     */
123.    for(int i = 1; i < binlongest.length(); i++) {
124.      header += "0";
125.    }
126.    header += "1";
127.
128.    //Übertragung der Codetabelle in den Header
129.    codes.gibTabelle().toFirst();
130.    while(codes.gibTabelle().hasAccess()) {
131.        //Angabe der Länge des folgenden Codes
132.        //Füllt die Länge auf um eine standartisierte Länge zu erreichen
133.        for(int j = 0; j < binlongest.length() -
    Integer.toBinaryString(codes.gibTabelle().getContent().gibCode().length()).length(); j++) {
134.          header += "0";
135.        }
136.        header += Integer.toBinaryString(codes.gibTabelle().getContent().gibCode().length()); //Fügt die
    Länge binär an
137.
138.        //Angabe des Codes
139.        header += codes.gibTabelle().getContent().gibCode();
140.
141.        //Angabe des dazugehörigen Zeichens
142.        //Füllt die Länge auf um eine standartisierte Länge zu erreichen
143.        for(int j = 0; j < 8 - Integer.toBinaryString((int)
    codes.gibTabelle().getContent().gibZeichen()).length(); j++) {
144.          header += "0";
145.        }
146.        header += Integer.toBinaryString((int) codes.gibTabelle().getContent().gibZeichen()); //Fügt die
    binäre Darstellung des ASCII-Codes des Zeichens ein
147.
148.        codes.gibTabelle().next();
149.    }
150.
151.    //Gibt das Ende des Headers über die Länge 0 an, was eine Angabe der Länge des gesamten Headers umgeht
152.    for(int i = 0; i < binlongest.length(); i++) {
153.      header += "0";
```

```
154.     }
155.
156.     //Kodiert den Ausgangstext über die Codetabelle
157.     String finalerText = "";
158.     for (int i = 0; i < pText.length(); i++) {
159.       finalerText += codes.gibCode(pText.charAt(i));
160.     }
161.
162.     //Gibt den header mit angefügtem kodierten Text aus
163.     return header + finalerText;
164.   }
165.
166.   //Hilfsmethode um die Codetabelle zu erstellen (rekursiv)
167.   static private void codesAuswerten(BinaryTree<HuffmanInhalt> pBaum, Codetabelle pCodetab, String pCode) {
168.     if(!pBaum.isEmpty()) {
169.       if(pBaum.getContent().gibZeichen() != Character.MIN_VALUE) {
170.         pCodetab.hinzufuegen(pCode, pBaum.getContent().gibZeichen()); //Fügt Zeichen ein, falls gefunden
171.       } else {
172.         codesAuswerten(pBaum.getLeftTree(), pCodetab, pCode + "0"); //Testet den linken Teilbaum mit einem um
    eine 0 erweiterten Code
173.         codesAuswerten(pBaum.getRightTree(), pCodetab, pCode + "1"); //Testet den rechten Teilbaum mit einem
    um eine 1 erweiterten Code
174.       }
175.     }
176.   }
177.
178.   //Methode für die Dekomprimierung eines Textes
179.   static public String dekomprimieren(String  pText){
180.     int laengenl = 0; //Länge der Längenangaben
181.     int i = 0; //Stelle im kodierten Text
182.
183.     //Wertet den ersten Teil des Headers über die Länge der Längenangaben aus
184.     while(pText.charAt(i) == '0') {
185.       i++;
186.     }
187.     laengenl = i + 1; //Speichert den gewonnenen Wert
188.
189.     //Erstellt über die Längenangabe ein Schema für die Markierung des Headerendes
190.     String headerEnde = "";
191.     for(int j = 0; j < laengenl; j++) {
192.       headerEnde += "0";
193.     }
194.
195.     //Wertet die im Header angegebene Codetabelle aus
196.     boolean flag = true; //Abbruchbedingung
197.     String codelaenge = ""; //Länge des aktuellen Codes
198.     String code = ""; //Aktueller Code
199.     Codetabelle zeichen = new Codetabelle(); //Codetabelle um alle Zeichen zu speichern
200.     while(flag) {
201.       code = "";
202.       codelaenge = "";
203.
204.       //Erfasst die Codelänge
205.       for(int j = 0; j < laengenl; j++) {
206.         i++;
207.         if(i < pText.length()) {
208.           codelaenge += pText.charAt(i);
209.         }
210.       }
211.
212.       //Code in Tabelle aufnehmen, sonst Abbruch
213.       if(codelaenge != headerEnde && codelaenge != "" && Integer.parseInt(codelaenge,2) != 0) { //Überprüft
    ob die Codelänge das Headerende markiert
214.
215.         //Erfasst den Code
216.         for(int j = 0; j < Integer.parseInt(codelaenge, 2); j++) {
217.           i++;
218.           code += pText.charAt(i);
219.         }
220.
221.         //Erfasst den binären ASCII-Wert des Zeichens
222.         String charNum = "";
223.         for(int j = 0; j < 8; j++) {
224.           i++;
225.           if(i < pText.length()) {
226.             charNum += pText.charAt(i);
227.           }
228.         }
229.         zeichen.hinzufuegen(code, (char) Integer.parseInt(charNum,2)); //Fügt das Zeichen in die Tabelle ein
230.       } else {
231.         flag = false;
232.       }
233.     }
234.
235.     //Dekodiert den kodierten Text mit der erstellten Codetabelle
236.     i++;
```

```
237.    String dektext = ""; //Dekodierter Text
238.    code = "";
239.    for(int j = i; j < pText.length(); j++) {
240.      code += pText.charAt(j);
241.
242.      //Guckt, ob ein Zeichen mit diesem Code existiert, fügt ansonsten eine weitere Stelle an diesen an
243.      if(zeichen.gibZeichen(code) != Character.MIN_VALUE) {
244.        //Fügt das Zeichen über den Code ein und setzt diesen zurück
245.        dektext += zeichen.gibZeichen(code);
246.        code = "";
247.      }
248.    }
249.    return dektext;
250.  }
251.
252.  //Main-Methode, um die Funktionsweise beim Ausführen zu testen
253.  public static void main(String[] args) {
254.    //Hier sind im Originalquelltext einige Methodenaufrufe, um die Funktionsweise zu testen
255.  }
256. }
```

Codetabellenklasse:

```
1.  public class Codetabelle
2.  {
3.    //Attribute
4.      private List<Code> tabelle = new List<Code>();
5.
6.    //Methoden
7.    public char gibZeichen(String  pCode){
8.      tabelle.toFirst();
9.      while(tabelle.hasAccess() && !tabelle.getContent().gibCode().equals(pCode)) {
10.       tabelle.next();
11.     }
12.     if(tabelle.hasAccess()) {
13.       return tabelle.getContent().gibZeichen();
14.     } else {
15.       return Character.MIN_VALUE;
16.     }
17.   }
18.   public String gibCode(char  pZeichen){
19.     tabelle.toFirst();
20.     while(tabelle.getContent() != null && tabelle.getContent().gibZeichen() != pZeichen &&
    tabelle.hasAccess()) {
21.       tabelle.next();
22.     }
23.     if(tabelle.hasAccess()) {
24.       return tabelle.getContent().gibCode();
25.     } else {
26.       return null;
27.     }
28.   }
29.   public List<Code> gibTabelle() {
30.     return tabelle;
31.   }
32.   public void hinzufuegen(String  pCode, char  pZeichen){
33.     tabelle.append(new Code(pZeichen,pCode));
34.   }
35. }
```

Codeklasse:

```
1.  public class Code
2.  {
3.    //Attribute
4.      private String code;
5.      private char zeichen;
6.
7.    //Konstruktor
8.    public Code(char pZeichen, String pCode) {
9.      zeichen = pZeichen;
10.     code = pCode;
11.   }
12.
13.   //Getter-Methoden
14.   public String gibCode(){
15.     return code;
16.   }
17.   public char gibZeichen(){
```

```
18.      return zeichen;
19.    }
20.  }
```

Zusätzliche „count“-Methode in der Zentralabiturklasse List:

```
1.   //Hinzugefügter Teil - Count-Methode
2.   public int count() {
3.       int i = 0;
4.       toFirst();
5.       while(hasAccess()) {
6.         i++;
7.         next();
8.       }
9.       return i;
10.  }
```

Klasse des Binärbaum-Inhaltsobjektes:

```
1.   public class HuffmanInhalt
2.   {
3.     //Attribute
4.       private char zeichen;
5.       private int wert;
6.
7.     //Konstruktoren
8.     public HuffmanInhalt() {
9.       zeichen = Character.MIN_VALUE;
10.      wert = 0;
11.    }
12.    public HuffmanInhalt(char pZeichen, int pWert) {
13.      zeichen = pZeichen;
14.      wert = pWert;
15.    }
16.
17.    //Getter-Methoden
18.    public char gibZeichen(){
19.      return zeichen;
20.    }
21.    public int gibWert(){
22.      return wert;
23.    }
24.
25.    //Methoden um die Wertigkeit zu erhöhen
26.    public void wertErhoehen() {
27.      wert++;
28.    }
29.    public void erhoeheWertUm(int pWert) {
30.      wert += pWert;
31.    }
32.  }
```