

**Hochschule Karlsruhe  
Technik und Wirtschaft**  
UNIVERSITY OF APPLIED SCIENCES

**Projektarbeit**

**Kanonische Huffman Codes**  
(Canonical Huffman Codes)

**SS 2008**

**Studentin: Michaela Kieneke**

**Dozent: Dr. Heiko Körner**

```
000101011011011001000111011000111110000110100011011110010111110001111000  
1010101110101001011100100101011111110000011001011010100101000101010001010  
1100000101011000111111011101100100001000000000001100011000000111000010010  
01010010101110001110010011000110001111001111101011101100111
```

# Kanonische Huffman Codes

## Vorwort

Sicher kann sich fast jeder von uns daran erinnern, wie er vor vielen Jahren mit Freunden verschlüsselte Nachrichten ausgetauscht hat. Dazu wurde mühsam jedes einzelne Zeichen durch ein anderes ersetzt. Im einfachsten Fall verschob man die Buchstaben des Alphabets um ein oder mehrere Stellen vor oder zurück. Erste Unstimmigkeiten kamen auf, sobald Zahlen bzw. Ziffern mit verschlüsselt werden sollten. Wurden sie einfach ans Alphabet hinten angehängt? Oder sollten die zehn Ziffern für sich selbst verschoben werden?<sup>1</sup> Es mussten also eindeutige Regeln für die Verschlüsselung aufgestellt werden.

Eine Erweiterung waren selbst erfundene oder z.B. vielleicht den Micky Maus Heften entnommene Symbole, die für jedes Zeichen des Alphabets abgemalt werden mussten. Das Verschlüsseln fiel uns noch recht leicht – aber wenn dann eine verschlüsselte Nachricht bei uns ankam, dauerte es meist sehr lange, bis wir alle Zeichen wieder zurück übersetzt hatten. Klar, beim Verschlüsseln konnte aus einer sortierten Liste gewählt werden, beim Entschlüsseln musste man Zeichen für Zeichen vergleichen, um auf das richtige Symbol zu schließen.

Häufig auftretende Buchstaben, wie beispielsweise das „E“ im deutschen Wortschatz, konnte man bald auswendig, seltene Buchstaben wie z.B. das „Q“ konnte man sich eher nicht merken. So kam man auf die Idee, für häufiger auftretende Zeichen einfachere Symbole zur Verschlüsselung zu wählen, die Symbole für seltenerer Zeichen konnten ruhig komplizierter zu malen sein.

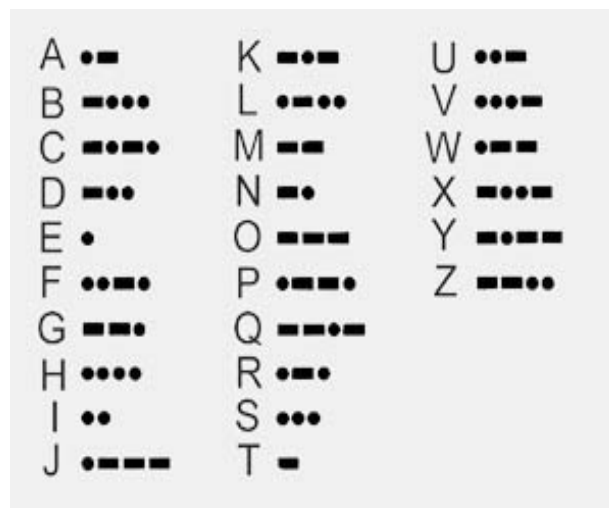


Abbildung 1: Morsealphabet

Etwas Ähnliches lässt sich beim Morsealphabet erkennen, welches auf Samuel Morse zurück geht<sup>2</sup>. Bei diesem wird jedes Zeichen des normalen Alphabets durch eine festgelegte

<sup>1</sup> Claudio Eckert: „Seite 42“, URL: [http://www.seite42.de/81\\_16.htm](http://www.seite42.de/81_16.htm) [Stand 09.04.09]

<sup>2</sup> Wikimedia Foundation Inc.: „Morsecode“, URL: <http://de.wikipedia.org/wiki/Morsecode> [Stand 09.04.09]  
 Dennis Pauler: „VISUM Informationssysteme GmbH“, URL: <http://www.visum-informationssysteme.de>

## Kanonische Huffman Codes

Kombination von kurzen und langen Signalen ersetzt, deren Vorteil darin besteht, dass sie akustisch oder visuell übertragen werden können. Für häufig auftretende Zeichen wie das „E“ wird nur ein kurzes Signal, für das viel seltenere „Q“ werden zwei lange, ein kurzes und wiederum ein langes Signal verwendet.

Diese Idee wird ebenfalls vom Huffman Code berücksichtigt. Allerdings wird dieser nicht dazu verwendet, Texte zu verschlüsseln. Er soll in erster Linie die Daten komprimieren. Dazu wird der vorliegende Text untersucht und den am häufigsten auftretenden Zeichen bzw. Worten die kürzesten Codewörter zugewiesen, welche ausschließlich aus Bits, also 0 und 1, bestehen. Komprimierte Texte lassen sich relativ einfach wieder decodieren, da eine Besonderheit des Huffman Codes darin besteht, dass kein Codewort der Anfang eines anderen Codewortes ist.

Der Nachteil beim normalen Huffman Code besteht darin, dass der Aufbau dieser Codewörter mit unterschiedlichen Ergebnissen durchgeführt werden kann und somit nicht eindeutig ist. Hierzu müssten umfangreiche Regeln aufgestellt werden. Der kanonische Huffman Code dagegen verwendet eine eindeutige Codierung und belegt den Schwerpunkt dieser Projektarbeit.

# Kanonische Huffman Codes

## Inhaltsverzeichnis

<b>Vorwort</b> .....	<b>ii</b>
<b>1 Kompression</b> .....	<b>1</b>
1.1 Arithmetische Kompression.....	1
1.2 Huffman Code.....	1
1.3 Geschwindigkeit und Grenzen der Kompression.....	1
1.3.1 Informationsgehalt.....	2
1.3.2 Entropie.....	2
1.3.3 Mittlere Codewortlänge.....	3
1.3.4 Redundanz.....	4
<b>2 Huffman Code</b> .....	<b>6</b>
2.1 Einfacher Huffman Code (Huffman 1952).....	6
2.1.1 Codierung.....	6
2.1.2 Decodierung.....	10
2.1.2.1 Baumtraversierung.....	10
2.1.2.2 Decodiertabelle.....	10
2.2 Canonical Huffman Code (Hirschberg und Lelewer 1990).....	11
2.2.1 Codierung.....	11
2.2.2 Decodierung.....	12
2.3 Bewertung der Huffman Codierung.....	13
<b>3 Die Implementierung</b> .....	<b>14</b>
3.1 Codierung.....	14
3.1.1 Codewortlänge.....	14
3.1.1.1 Was ist eine Heap Datenstruktur?.....	14
3.1.1.1.1 Wie wird ein Heap aufgebaut?.....	15
3.1.1.1.2 Wie funktioniert die Aktualisierung eines Heaps?.....	18
3.1.1.1.3 Der Heap für die Implementierung des Huffman Codes.....	23
3.1.2 Zuordnung des Codeworts.....	29
3.2 Decodierung.....	32
3.3 Speicherverbrauch und Geschwindigkeit.....	34
3.3.1 Der Aufbau im Heap und erneutes Versickern.....	34
3.3.2 Die Codewortlänge: Baum und Heap im Vergleich.....	39
3.3.3 Zeit zum Codieren unter Verwendung eines Heaps.....	40
3.3.4 Speicher beim Decodieren mit einem Baum.....	40
3.3.5 Speicher beim Decodieren ohne Baum.....	40
3.3.5.1 Zweidimensionales Array.....	40
3.3.5.2 Eindimensionales Array – erste Variante.....	41
3.3.5.3 Eindimensionales Array – zweite (verwendete) Variante.....	42
3.3.6 Benötigte Daten.....	43
3.4 Die grafische Benutzeroberfläche.....	45
3.4.1 Style der GUI.....	46



## Kanonische Huffman Codes

<a href="#">3.4.2 Eingabe des Textes</a>	46
<a href="#">3.4.3 Zeichen- oder Wortcodierung</a>	47
<a href="#">3.4.4 Ausgabe des codierten Textes</a>	47
<a href="#">3.4.5 Decodieren</a>	47
<a href="#">3.4.6 Wörterbuch</a>	47
<a href="#">3.4.7 Löschen</a>	48
<a href="#">3.4.8 Debug-Ausgabe</a>	48
<a href="#">3.4.9 Beenden</a>	48
<a href="#">3.4.10 Fehlerausgabe</a>	48
<b><a href="#">4 Schlusswort</a></b>	<b>49</b>
<b><a href="#">5 Quellenverzeichnis</a></b>	<b>50</b>





# 1 Kompression

## 1 Kompression

Wenn Daten übermittelt werden sollen, ist es von Vorteil, diese möglichst klein zu halten. Hierzu können verschiedene Kompressionsverfahren verwendet werden. Im Folgenden sollen die beiden wesentlichen verlustfreien Kompressionsverfahren, die überwiegend bei Texten und Anwendungsdateien zum Einsatz kommen, kurz mit ihren jeweiligen Vor- und Nachteilen vorgestellt werden.

Bei beiden Verfahren muss die Nachricht im Vorhinein bekannt sein, um eine optimale Kompression zu erreichen.

### 1.1 *Arithmetische Kompression*

Die arithmetische Kompression ist das ergiebigeres Verfahren beim Einsatz zur Codierung „mit Gedächtnis“ (z.B. lässt sich beobachten, dass im deutschen Wortschatz nach einem „Q“ meist ein „U“ folgt). Die Geschwindigkeit ist fast mit der des Huffman Codes vergleichbar. Sie verbraucht weniger Speicher und bietet je nach Code eine bessere Kompression: ein Codewort beim Huffman Code besteht stets aus einer ganzzahligen Anzahl Bits, womit die Länge des Codewortes logarithmisch zur Auftrittswahrscheinlichkeit ist. Dieses Problem löst die arithmetische Codierung, indem ein Wort durch eine Zahl  $z \in [0; 1)$  dargestellt wird.

### 1.2 *Huffman Code*

Der Huffman Code ist bei „nicht lernender“ Codierung schneller als das arithmetische Kompressionsverfahren und arbeitet darüber hinaus bei der Kompression von Texten, bei denen nicht die Zeichen- sondern die Wortwahrscheinlichkeiten zur Berechnung verwendet werden, äußerst schnell. Für häufig auftretende Zeichen bzw. Worte berechnet er kurze Codewörter und längere für seltener auftretende Symbole bzw. Worte.

Die Huffman Codierung ist unter den längenvariablen Codes optimal, weil keine andere Codierung, die jedes auftretende Zeichen mit einer festen Anzahl Bits codiert, die mittlere Codewortlänge des Huffman Codes unterbieten kann.

Zum Einsatz bei „lernender“ Codierung eignet sich das Huffman Verfahren weniger, da Bäume „mit Gedächtnis“ extrem langsam sind und zudem viel Speicher benötigt wird, weil mehrere Bäume während der Codierung entstehen.

### 1.3 *Geschwindigkeit und Grenzen der Kompression*

Eine gute Kompression geht auf Kosten der Geschwindigkeit. Beispielsweise lassen sich 256 Zeichen codieren, indem für die 15 häufigsten ein 4-Bit-Codewort verwendet wird, für die restlichen 12-Bit-Codewörter. Schneller ginge es natürlich, würde einfach für jedes der 256 Zeichen ein 8-Bit-Codewort verwendet. So haben die Verschlüsselungen gearbeitet, die wir als Kinder benutzt haben, indem Zeichen 1:1 ersetzt wurden, eine Kompression

# 1 Kompression

wird so jedoch nicht erreicht.

Darüber hinaus sind bei der Kompression Grenzen gesetzt, egal welches Kompressionsverfahren verwendet wird. Im Folgenden werden Kennzeichen erläutert, die zeigen, welche Kompression theoretisch optimal wäre und welche tatsächlich realisiert wird. Als konkretes Beispiel wird dies an der Huffman Codierung des Wortes „ABRAXAS“ demonstriert, welches auch in den folgenden Kapiteln als Berechnungsbeispiel dient.

## 1.3.1 Informationsgehalt

Der Informationsgehalt eines einzelnen Zeichens  $a_i \in A$  beträgt:

$$I(a_i) = -\log_2 p(a_i) [\text{bit}] = \log_2 \left( \frac{1}{p(a_i)} \right) [\text{bit}]$$

*Text 1: Formel Informationsgehalt*

wobei  $p(a_i)$  die Auftrittswahrscheinlichkeit des Zeichens darstellt. Der Informationsgehalt besagt, welche Bitlänge für jedes Codewort erwartet wird. Da hier nicht immer ganzzahlige Werte herauskommen, muss zur Realisierung bis zum nächsten Bit gerundet werden.

Beim Wort „ABRAXAS“ beträgt Auftrittshäufigkeit von „A“  $3/7$ , die von „B“, „R“, „S“ und „X“  $1/7$ . Daraus ergibt sich folgender Informationsgehalt:

$$I('A') = \log_2 \left( \frac{7}{3} \right) \approx 1,2224 [\text{bit}]$$

$$I('B') = I('R') = I('S') = I('X') = \log_2 7 \approx 2,8074 [\text{bit}]$$

*Text 2: „ABRAXAS“ - Informationsgehalt*

Die erwartete Codewortlänge von „A“ beträgt etwa 1,2224 Bit, gerundet also 1 Bit, die von „B“, „R“, „S“ und „X“ etwa 2,8074, gerundet also 3 Bit.

## 1.3.2 Entropie

Bei der Entropie handelt es sich um ein Maß für den mittleren Informationsgehalt bzw. die mittlere Informationsdichte einer diskreten gedächtnislosen Datenquelle (A). Zur Berechnung werden die Informationsgehalte aller Zeichen mit ihrer Auftrittswahrscheinlichkeit multipliziert und aufaddiert.

# 1 Kompression

$$E(A) = \sum p(a_i) * I(a_i) [\text{bit}] \text{ für alle } a_i \in A$$

*Text 3: Formel Entropie*

Nach Claude Elwood Shannon (1916 – 2001) benötigt die Codierung einer Nachricht der Länge  $n$  mindestens  $n * E(A)$  Bits, unabhängig von der Wahl der Codierung. Somit definiert die Entropie die untere Schranke für die maximal erreichbare Kompressionsrate, also die kleinstmögliche durchschnittliche Länge pro Zeichen.

Die Entropie am Beispiel „ABRAXAS“:

$$E(A) = \frac{3}{7} * 1,2224 + 4 * \frac{1}{7} * 2,8074 \approx 2,1288 [\text{bit}]$$

*Text 4: „ABRAXAS“ - Entropie*

Die Informationsdichte beträgt etwa 2,1288 Bits. Nach C. E. Shannon bedeutet dies, dass die Nachricht „ABRAXAS“ mindestens  $7 * 2,1288$  Bits = 14,9016 Bits zur Übertragung benötigt, aufgerundet also 15 Bits, da weniger als 14,9016 Bits nicht möglich sind.

## 1.3.3 Mittlere Codewortlänge

Die mittlere Codewortlänge jedes Zeichens errechnet sich folgendermaßen:

$$L(A) = \sum p(a_i) * l_i [\text{bit}] \text{ für alle } a_i \in A$$

*Text 5: Formel Mittlere Codewortlänge*

wobei  $l_i$  die tatsächliche Codewortlänge in ganzen Bits des Zeichens  $a_i$  ist. Es handelt sich also um alle Zeichen-Auftrittswahrscheinlichkeiten multipliziert mit ihrer verwendeten Codewortlänge aufsummiert zu einem Gesamtwert. Hier handelt es sich um die tatsächlich realisierbare durchschnittliche Codewortlänge, wobei die Entropie im Gegensatz dazu den exakten Wert der möglichen Kompression beschreibt aber nicht berücksichtigt, ob ein Codewort eine ganzzahlige Länge von Bits erhält. Somit gilt:

# 1 Kompression

$$E(A) \leq L(A)$$

*Text 6: Entropie und mittlere Codewortlänge*

Da die Entropie die untere Schranke der maximalen Kompression darstellt, ist sie kleiner oder gleich der tatsächlich realisierten durchschnittlichen Codewortlänge.

$$L(A) = \frac{3}{7} * 1 + 4 * \frac{1}{7} * 3 \approx 2,1429 [bit] > 2,1288 [bit] \text{ (aus 1.3.2)}$$

*Text 7: „ABRAXAS“ - Mittlere Codewortlänge*

Bei der Codierung des Wortes „ABRAXAS“ wird also eine tatsächliche durchschnittliche Codewortlänge von etwa 2,1429 Bits benötigt.

## 1.3.4 Redundanz

Die Redundanz drückt die Differenz zwischen der Entropie und der tatsächlichen mittleren Codewortlänge aus. Somit beschreibt sie die durchschnittlichen „überflüssigen“ verwendeten Bits pro Codewort.

$$R = L - E [bit]$$

*Text 8: Formel Redundanz*

Idealerweise geht die Differenz gegen Null, was sich jedoch natürlich nicht immer bei einer Kompression realisieren lässt, da Codewörter beispielsweise bei der Huffman Codierung nur ganzzahlige Bitlängen aufweisen können.

Eine optimale Kompression lässt sich mit der Huffman Codierung erreichen, wenn die Auftrittswahrscheinlichkeiten der Zeichen  $1/2^n$  betragen. Als Codewortlänge ergibt sich für jedes Zeichen  $-\log_2 p(a_i) = -\log_2 (1/2^n) = n$ :

# 1 Kompression

$$L = E \text{ mit } p(a_i) = \frac{1}{2^n}$$

$$\rightarrow \sum p(a_i) * l_i = \sum p(a_i) * I(a_i)$$

$$\rightarrow \sum \frac{1}{2^n} * l_i = \sum \frac{1}{2^n} * -\log_2\left(\frac{1}{2^n}\right)$$

$$\rightarrow l_i = -\log_2\left(\frac{1}{2^n}\right) = \log_2 2^n = n$$

*Text 9: Redundanz bei Auftrittswahrscheinlichkeiten  $1/2^n$*

Somit kommt eine ganzzahlige Codewortlänge heraus und es werden bei der Kompression keine Bitanteile „verschenkt“.

$$R = L - E = 2,1429 [\text{bit}] - 2,1288 [\text{bit}] = 0,0141 [\text{bit}]$$

*Text 10: „ABRAXAS“ - Redundanz*

Die Differenz zwischen der durchschnittlichen Codewortlänge und der minimal möglichen Codewortlänge am Beispiel „ABRAXAS“ beträgt 0,0141 Bits. Somit werden bei der Codierung des gesamten Worts insgesamt  $7 * 0,0141$  Bits = 0,0987 Bits „verschenkt“.

$$\begin{aligned} \text{Tatsächliche Kompression} &= 7 * 2,1419 [\text{bit}] = 15,0003 [\text{bit}] \\ \text{Maximal mögliche Kompression} &= 7 * 2,1288 [\text{bit}] = 14,9016 [\text{bit}] \end{aligned}$$

*Text 11: Tatsächliche Kompression und maximal mögliche Kompression*

Tatsächlich benötigt die Darstellung 15 Bits (minimaler Fehler durch Rundung). Laut Berechnungen müsste bei einer maximalen Kompression 14,9016 Bits verwendet werden, welche auf ebenfalls 15 Bits aufgerundet werden. Somit ist im Beispiel „ABRAXAS“ die maximal mögliche Kompression erreicht.

## 2 Huffman Code

### 2 Huffman Code

Die Huffman Codierung wurde 1952 von David Albert Huffman (1925 – 1999) entwickelt. Da dieses Verfahren keine eindeutigen Ergebnisse liefert, erweiterten Hirschberg und Lelewer 1990 das Verfahren unter dem Namen „Canonical Huffman Code“.

#### 2.1 Einfacher Huffman Code (Huffman 1952)

Wie schon erwähnt, ist der Huffman Code unter den längenvariablen Kompressionsverfahren optimal und die codierten Daten können verlustfrei wiederhergestellt werden. Da es sich um einen präfixfreien Code handelt (kein Codewort ist der Anfang eines anderen Codeworts), wird bei der Codeübermittlung auf Trennzeichen verzichtet.

Der Huffman Code ist äußerst schnell sowohl bei der Codierung als auch bei der Decodierung.

Neben dem Nachteil, dass die gesamte Nachricht bekannt sein muss (bzw. die Auftretswahrscheinlichkeiten der einzelnen Symbole), um eine optimale Codierung zu berechnen, ist er außerdem recht fehleranfällig: ein falsch übertragenes Bit kann die gesamte Nachricht verändern und evtl. sogar ab dieser Stelle zerstören.

##### 2.1.1 Codierung

Im Folgenden soll der Aufbau der Huffman Zeichencodierung am schon bekannten Beispiel „ABRAXAS“ veranschaulicht werden.

Zu Beginn werden die Auftretswahrscheinlichkeiten der einzelnen Zeichen im zu codierenden Text ermittelt. Aus diesen werden die späteren Codewortlängen konstruiert.

Zeichen	Absolute Häufigkeit	Relative Häufigkeit
B	1	1/7
R	1	1/7
X	1	1/7
S	1	1/7
A	3	3/7
<b>Summe</b>	<b>7</b>	<b>7/7 = 1</b>

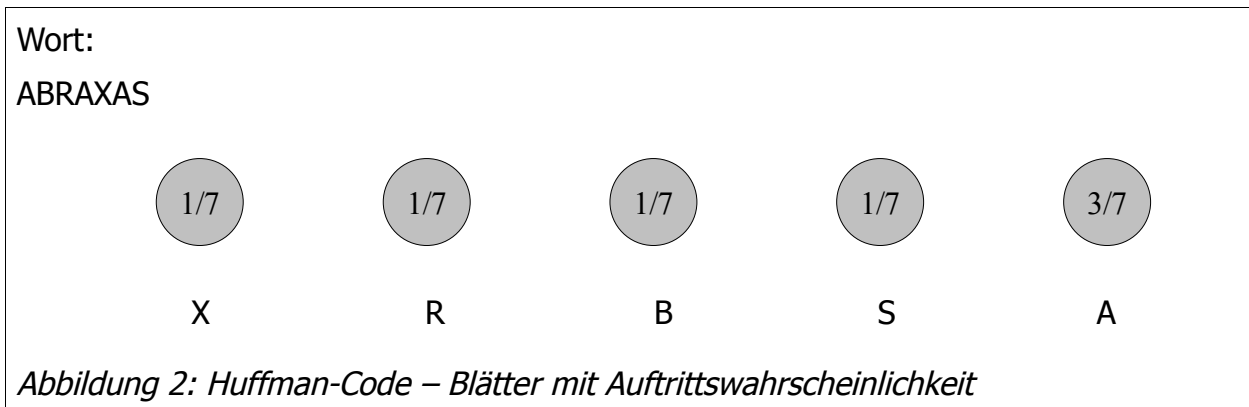
*Tabelle 1: Auftretshäufigkeit*

Die Codewörter werden über einen „binären“ Baum aufgebaut. Die Besonderheit eines binären Baumes liegt darin, dass jeder Knoten maximal zwei Kindelemente besitzt.

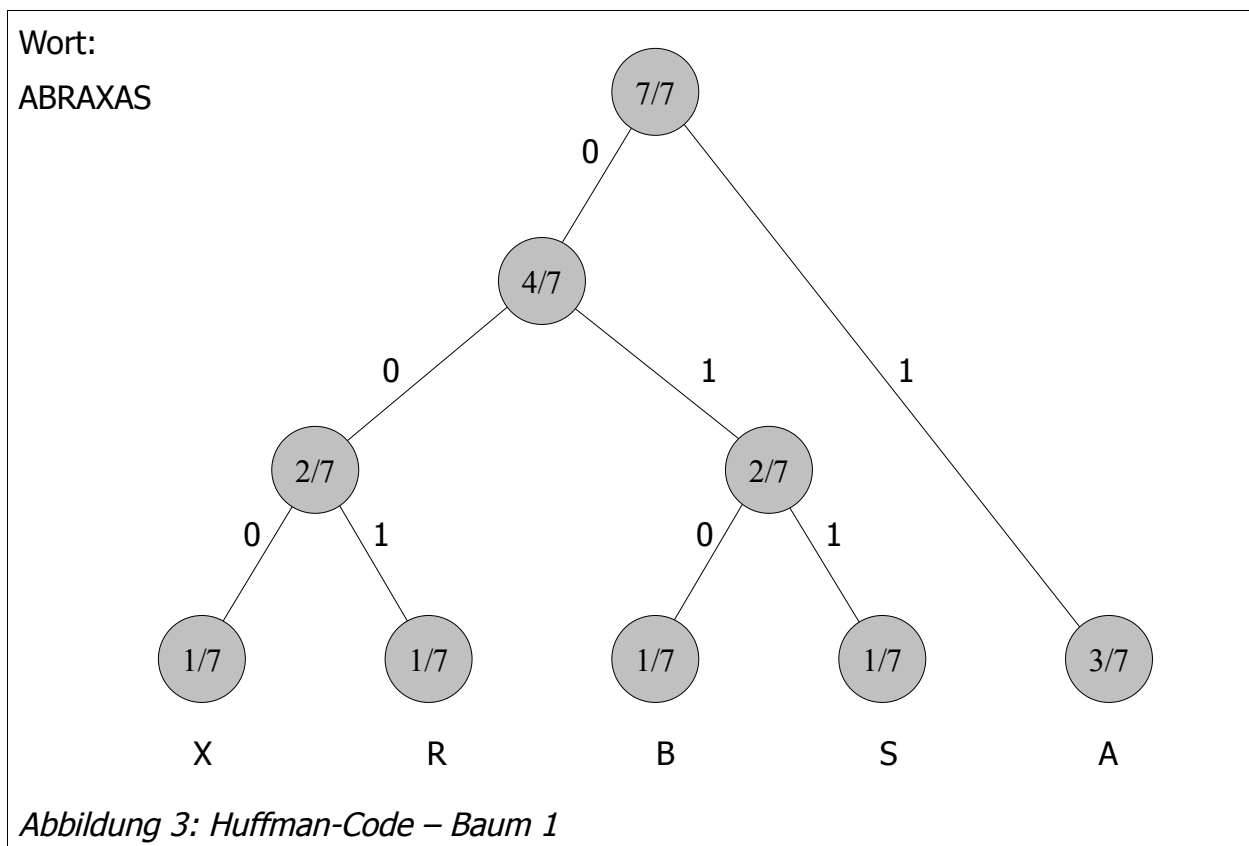
Entwickelt wird dieser Baum von „unten nach oben“, es wird also bei den Blättern gestar-

## 2 Huffman Code

tet. Diese Blätter beinhalten die Wahrscheinlichkeiten der zu codierenden Zeichen:



Jeweils zwei Knoten mit niedrigsten Werten ohne Elternknoten werden verbunden, wobei der so entstehende neue Knoten den Wert der Summe der mit ihm verbundenen Knoten bzw. Blätter erhält. Kommen mehrmals die gleichen „niedrigsten“ Werte vor, ist es nicht relevant, welche beiden ausgewählt werden. Wichtig ist nur, dass stets die niedrigsten Werte berücksichtigt werden, die noch keinen Elternknoten besitzen.



Im Beispiel haben „X“, „R“, „B“ und „S“ alle denselben niedrigsten Wert 1/7. Somit können

## 2 Huffman Code

„X“ und „R“ verbunden werden, da sie idealerweise im Beispiel nebeneinander liegen, wobei der Verbindungsknoten den Wert  $2/7$  erhält. Auf die gleiche Weise werden „B“ und „S“ verbunden. Die beiden Verbindungsknoten haben noch immer einen kleineren Wert als das „A“. Somit werden die beiden verbunden und erhalten den Wert  $4/7$ . Als letztes muss dieser neue Verbindungsknoten nur noch mit dem „A“ verbunden werden. So entsteht die Wurzel mit dem Knoten  $7/7 = 1$ , woran zu erkennen ist, dass die Berechnungen sehr wahrscheinlich korrekt verlaufen sind.

Zur Erstellung des Codes werden die Äste von jedem Knoten mit 0 bzw. 1 benannt, wobei es nicht relevant ist, welches Bit auf welcher Seite steht. Wird durchgehend am linken Ast die 0 und am rechten die 1 verwendet, ergibt sich aus obigem Beispiel folgende Codetabelle:

Symbol $a_i$	$p(a_i)$	Code 1
B	$1/7$	<b>010</b>
R	$1/7$	<b>001</b>
X	$1/7$	<b>000</b>
S	$1/7$	<b>011</b>
A	$3/7$	<b>1</b>

Tabelle 2: Huffman-Code 1

Das „A“ ist das am häufigsten vorkommende Zeichen und erhält somit ein möglichst kurzes Codewort. Am Beispiel lässt sich ebenfalls erkennen, dass es sich bei der Huffman-Codierung um einen präfixfreien Code handelt: keines der Codewörter ist der Beginn eines anderen Codeworts. Somit kann das Codewort ohne Leerzeichen zwischen den einzelnen Zeichen codiert werden:

101000110001011

*Text 12: Huffman Codierung für "ABRAXAS"*

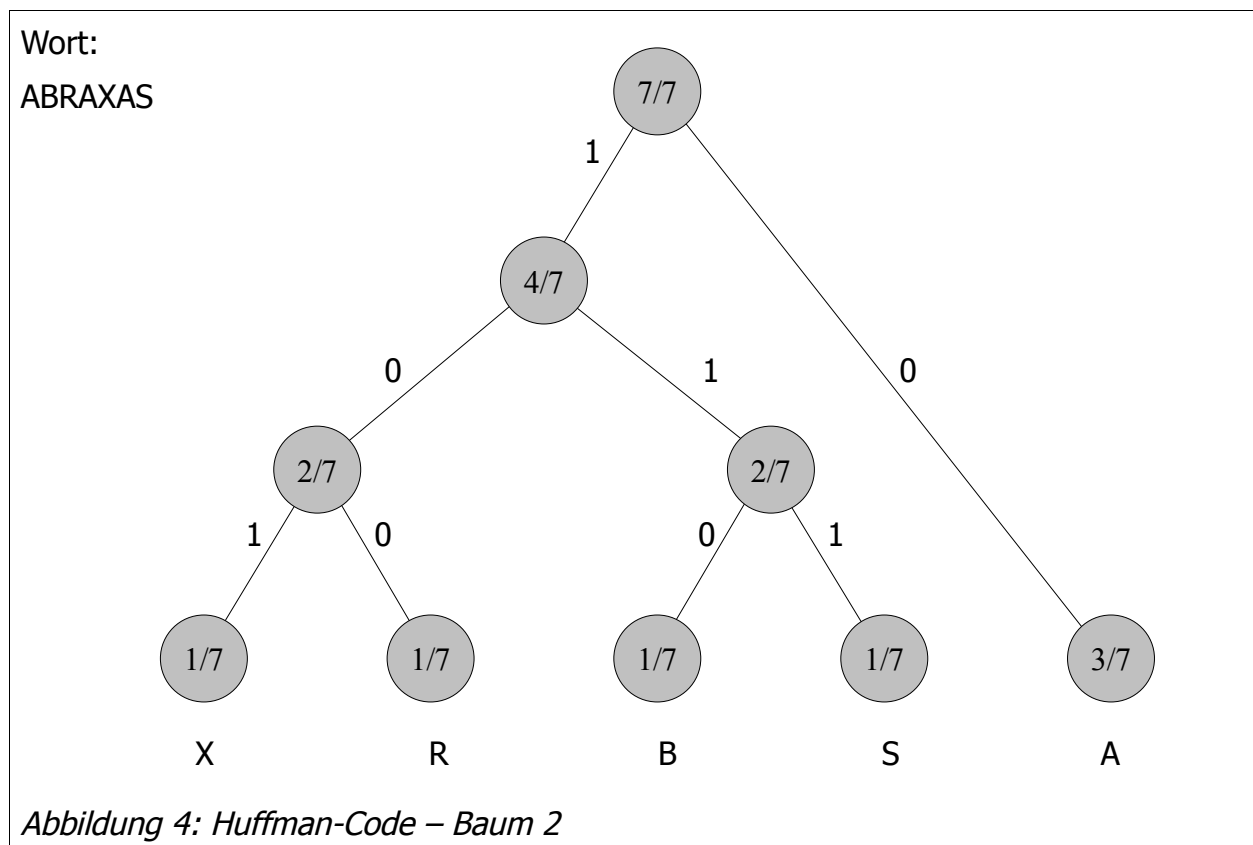
Was aber geschieht, wenn zu Beginn die Buchstaben an den Blättern des Baumes anders sortiert wurden? Oder die Äste anders benannt werden? Richtig: es entsteht ein anderer Code, der mit dem zuvor erstellten lediglich die Codewortlänge gemein hat:

## 2 Huffman Code

Symbol $a_i$	$p(a_i)$	Code 1	Code 2
B	1/7	010	<b>110</b>
R	1/7	001	<b>100</b>
X	1/7	000	<b>101</b>
S	1/7	011	<b>111</b>
A	3/7	1	<b>0</b>

Tabelle 3: Huffman-Code 2

Auf diese Weise ist Code 2 in der obigen Tabelle entstanden. Der dazugehörige Baum sieht folgendermaßen aus:



Wird ein Baum für  $n$  Zeichen entwickelt, erhält dieser genau  $n-1$  innere Knoten. Zur Benennung dieser so entstandenen Äste gibt es genau  $2^{n-1}$  verschiedene Möglichkeiten, die Äste mit 0 und 1 zu versehen. Im vorliegenden Fall würde dies bedeuten, dass es  $2^4 = 16$  Möglichkeiten gibt, gleich optimale Huffman Codes zu erstellen. Aufgrund dieses Nichtdeterminismus (gleicher Input liefert nicht immer den gleichen Output) ist der normale Huffman Code ungeeignet, wenn zwei miteinander kommunizierende Parteien unabhängig von einander die Codewörter erstellen wollen. Der Baum bzw. die Decodiertabelle müssen

## 2 Huffman Code

stets mit übermittelt werden.

### 2.1.2 Decodierung

Um den codierten Text wieder zu entschlüsseln, kommen die Baumtraversierung oder eine Decodiertabelle infrage.

#### 2.1.2.1 Baumtraversierung

Bei der Baumtraversierung wird an der Wurzel des Baum gestartet. Anhand des ersten Zeichens wird der erste Knoten ausgewählt, danach der zweite usw. bis das entsprechende Zeichen decodiert wurde:

101000110001011

*Text 13: Huffman Codierung für "ABRAXAS"*

Unter Verwendung des Code 1 (siehe Abbildung des entsprechenden Baums) liefert „1“ das „A“, „010“ ein „B“, „001“ ein „R“, „1“ wieder ein „A“, „000“ ein „X“, „1“ noch ein „A“ und „011“ ein „S“. Somit ist das Ursprungswort decodiert und heißt natürlich „ABRAXAS“.

#### 2.1.2.2 Decodiertabelle

Als Decodiertabelle wird eine Tabelle mit  $2^L$  Einträgen verwendet, wobei L die maximale Codewortlänge ist. Jeder Eintrag enthält das entsprechende Ausgabezeichen und den Bitrest.

Zum Erstellen dieser Tabelle wurde der Code 1 aus dem obigen Beispiel verwendet.

	<b>Zeichen</b>	<b>Bitrest</b>
000	X	-
001	R	-
010	B	-
011	S	-
100	A	00
101	A	01
110	A	10
111	A	11

*Tabelle 4: Decodiertabelle*

Es werden  $L = 3$  Bits eingelesen und als Tabellenindex verwendet. Das Zeichen kann aus-

## 2 Huffman Code

gegeben werden, der eventuelle Bitrest wird in den Indexpuffer gefüllt. Zum nächsten Auslesen des Zeichens wird die Anzahl der Bits im Indexpuffer wieder bis auf  $L = 3$  „aufgefüllt“, das nächste Zeichen kann ausgegeben werden.

Am Beispiel würde das so aussehen:

Indexpuffer	Ausgelesenes Zeichen	Bitrest
101	A	01 - Übertrag
010	B	-
001	R	-
100	A	00 - Übertrag
000	X	-
101	A	01 - Übertrag
011	S	-

Tabelle 5: Auslesen der Decodiertabelle

### 2.2 Canonical Huffman Code (Hirschberg und Lelewer 1990)

Der kanonische Huffman Code verwendet dieselbe Codewortlänge wie der normale Huffman Code, vermeidet jedoch den Nichtdeterminismus: die Codewörter des kanonischen Huffman Codes sind immer eindeutig, weil die Bits jedes Codeworts nach einem ganz bestimmten Verfahren zugewiesen werden.

#### 2.2.1 Codierung

Zu Beginn der Codierung müssen die entsprechenden Codewortlängen ermittelt werden, wozu beispielsweise der Baum der normalen Huffman Codierung verwendet werden könnte. Danach werden die Zeichen bzw. Worte absteigend nach der Codewortlänge bzw. aufsteigend nach der Auftrittshäufigkeit sortiert. Kommen mehrere Zeichen bzw. Wörter mit gleicher Codewortlänge vor, werden diese nochmal alphabetisch sortiert, um eine eindeutige Codierung zu erhalten:

Symbol $a_i$	$p(a_i)$	Code 1	Code 2	Code 3
B	1/7	010	110	000
R	1/7	001	100	001
S	1/7	011	111	010
X	1/7	000	101	011
A	3/7	1	0	1

Tabelle 6: Kanonischer Huffman Code 3

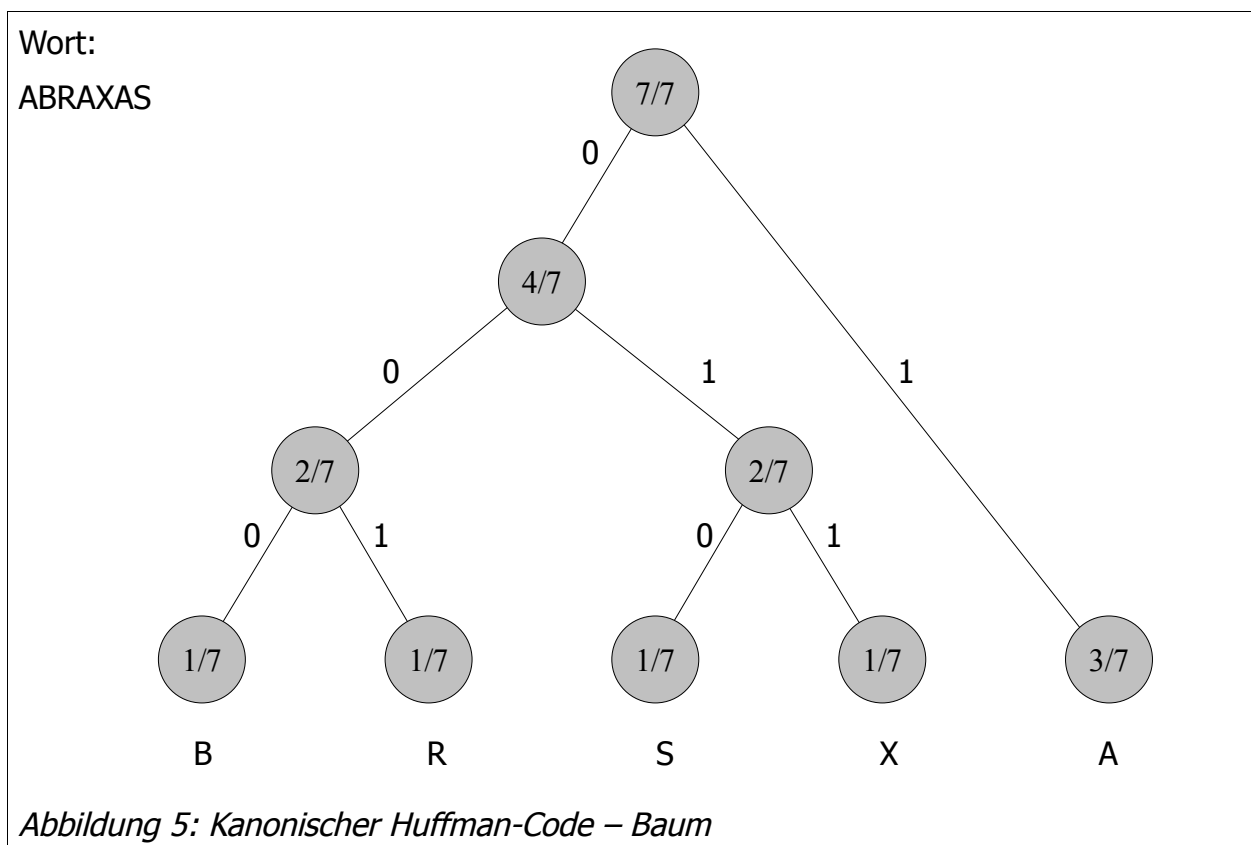
## 2 Huffman Code

Das erste Codewort jeder Länge wird festgelegt, nachfolgende Codewörter derselben Länge erhöhen sich jeweils um 1 (binär). Durch die vorherige Sortierung der Zeichen bzw. Wörter führt dieses Prinzip immer zum selben Ergebnis, da beim ersten Codewort der maximalen Codewortlänge stets die „0“ vergeben wird.

Somit wird aus dem „ABRAXAS“ mithilfe der kanonischen Huffman Codierung:

100000110111010  
Text 14: Kanonische Huffman Codierung für "ABRAXAS"

Diese Codewörter hätten ebenfalls über einen Baum gefunden werden können. Dies wäre jedoch eher zufällig geschehen, da es in der Regel keine Vorschriften zur Anordnung der Blätter und zur Benennung der Äste bei der normalen Huffman Codierung gibt und diese Beschriftung eine von den bereits erwähnten 16 Möglichkeiten darstellt.



### 2.2.2 Decodierung

Zum Decodieren wird kein Baum verwendet wie beim „kleinen Bruder“ der kanonischen

## 2 Huffman Code

Huffman Codierung. Stattdessen liegt eine Liste der Zeichen vor, sortiert nach der lexikalischen Ordnung der Codewörter. Außerdem benötigt man ein Array, indem sich das erste Codewort jeder Länge befindet (*firstcode[]*). Im vorliegenden Beispiel wäre das erste 3-Bit-Codewort „000“ also als Dezimalwert ebenfalls „0“.

...0111010

*Text 15: Codefragment zur Decodierung*

Aufgrund der durchdachten Zuordnung der Codewörter lässt sich aus dem Codewort selber erkennen, welche Codewortlänge es haben muss, während man die ersten Bits betrachtet. Im recht einfach gewählten Beispiel kann es sich schon auf den ersten Blick nicht um ein 1-Bit-Codewort handeln, da dies die 1 wäre. Ein 2-Bit-Codewort kann es ebenfalls nicht sein, weil diese im gewählten Beispiel nicht vorkommen. Bei einem umfangreicheren Beispiel könnte man erkennen, dass die ersten  $n$  gewählten Bits immer kleiner eines bestimmten gespeicherten Werts sind, dem ersten Codewort der Länge  $n+1$  (binär betrachtet, siehe Kapitel 3).

Vom 3-Bit-Codewort „011“ wird das erste 3-Bit-Codewort abgezogen, als Ergebnis bekommen man, um das wievielte 3-Bit-Codewort es sich handelt:

$$\begin{array}{r} 011 \\ - 000 \\ \hline 11 \rightarrow 3 \end{array}$$

*Text 16: Binäre Subtraktion*

Diese Differenz von „3“ besagt also, dass vom ersten Zeichen mit 3-Bit-Codewort noch drei Zeichen weiter gezählt werden muss, um das Originalzeichen zu erhalten. Das erste entsprechende Zeichen ist das „B“, drei Zeichen weiter erhält man das „X“. Diese Art der Decodierung ist wesentlich schneller als die Suche in einem Baum, der zudem noch viel Speicher benötigt und im normalen Huffman Code darüber hinaus zufällig generiert wird.

### 2.3 Bewertung der Huffman Codierung

Obwohl es sich nur um ein kleines Beispiel handelt, ist die Kompression optimal. Die Huffman Codierung erfüllt alle errechneten Werte aus Kapitel 1.

## 3 Die Implementierung

### 3 Die Implementierung

Zur Implementierung wurde sich an dem Buch „Managing Gigabytes: Compressing and Indexing Documents and Images“ von Ian H. Witten, Alistar Moffat und Timothy C. Bell, Kapitel 2.3 „Huffman Coding“, S. 30 – 51, orientiert. Als Programmiersprache wurde Java verwendet.

#### 3.1 Codierung

Zuerst muss für jedes Zeichen  $l$  die Codewortlänge  $l_i$  ermittelt werden. Danach wird von  $l_i = 1$  bis  $maxlength$  (= maximale Codewortlänge) gezählt, wie oft jede Codewortlänge vorkommt und diese in  $numl[l]$  gespeichert. Dies wird benötigt, um die Codewörter jedes Zeichens ( $codeword[i]$ ) entsprechender Länge fortlaufend hoch zu zählen ( $firstcode[l]$  und  $nextcode[l]$ ). Zusätzlich wird  $firstcode$  später zur Decodierung benötigt.

Über das Array  $symbol[]$  in Verbindung mit dem Array  $start[]$  wird gespeichert, welches Symbol welche Codewortlänge und das wievielte Codewort dieser Länge es bekommen hat. Diese Arrays ermöglichen später eine äußerst schnelle Decodierung.

##### 3.1.1 Codewortlänge

Die Berechnung der Codewortlänge beeinflusst nur die Zeit des Codierens, nicht die des Decodierens, trotzdem sollte dies möglichst schnell erfolgen. Die Verwendung eines Baums ist unnötig, da nur die Länge des Codeworts benötigt wird, nicht aber die Bitverteilung wie im „normalen“ Huffman Code. Außerdem verschwendet ein Baum eine große Menge Speicher, was bei umfangreichen Texten zum Problem werden kann.

Zuerst muss das Auftreten aller Zeichen erfasst werden. Um danach die Codewortlänge zu berechnen, wird eine Heap Datenstruktur mit einem Heap Sort verwendet. Dieses Vorgehen arbeitet wesentlich effizienter als eine Baumstruktur.

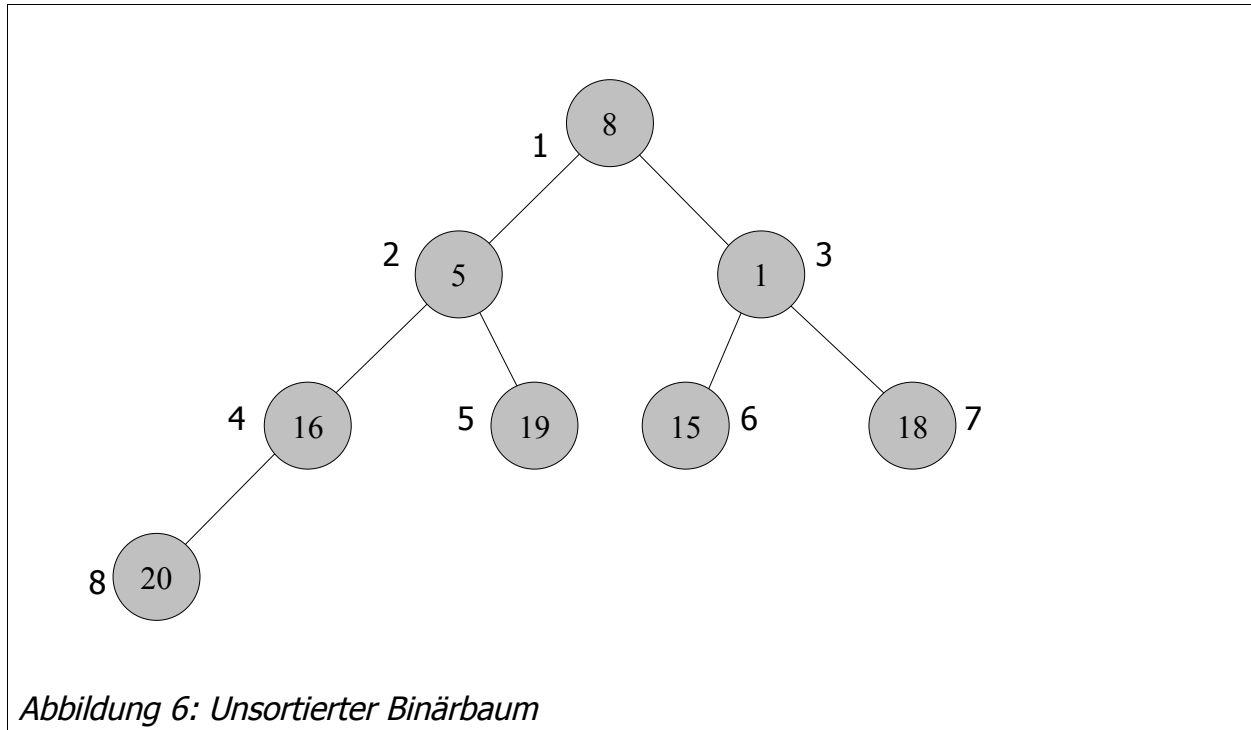
###### 3.1.1.1 Was ist eine Heap Datenstruktur?

Bei einem Heap handelt es sich im Prinzip um einen sortierten binären Baum verpackt in ein Array. Die Werte in Blättern und Knoten werden dazu an bestimmten Stellen im Array gespeichert. Der Heap kann ab- bzw. aufsteigend sortiert sein: „Max-Heap“ bzw. „Min-Heap“. Für einen Max-Heap heißt dies, dass der Vaterknoten stets größer ist als seine beiden Kindknoten. Auf diese Weise lässt sich sehr schnell und effizient wiederholt das größte bzw. kleinste Element eines Heaps ermitteln, welches sich an der „Wurzel“ des Heaps befindet. Dieses Element wird dann entfernt und durch das letzte Element des Heaps ersetzt, welcher sich somit um ein Element verkürzt. Um den Heap wieder aufzubauen, muss dieses Element an erster Stelle wieder „versickert“ werden, das heißt also, soweit mit Kindknoten verglichen und vertauscht werden, bis alle Werte wieder richtig sortiert sind.

## 3 Die Implementierung

### 3.1.1.1.1 Wie wird ein Heap aufgebaut?

Gegeben sei folgender Binär-Baum:



Werden die Knoten von 1 bis 8 durchnummeriert, lässt sich leicht erkennen, dass auf diese Weise Kindknoten von  $i$  die Indizes  $2*i$  bzw.  $2*i + 1$  haben. Diese Tatsache wird im Array verwendet, um auf Kind- bzw. Elternknoten zurückgreifen zu können. Zur Vereinfachung wird ein Array mit der Länge  $n + 1$  angelegt und das erste Element `array[0]` leer gelassen bzw. ist der Inhalt uninteressant. Dies ermöglicht den Zugriff auf der „erste“ Element mit `array[1]`, auf das zweite mit `array[2]` etc.:

Das erste Element mit  $i = 1$  hat die Kindknoten  $2*(i + 1) = 2$  und  $2*(i + 1) + 1 = 3$

```
int array[] = {-, 8, 5, 1, 16, 19, 15, 18, 20}
```



Abbildung 7: Heap-Array – Kindelemente von `array[1]`

Das zweite Element mit  $i = 2$  hat die Kindknoten  $2*(i + 1) = 4$  und  $2*(i + 1) + 1 = 5$

```
int array[] = {-, 8, 5, 1, 16, 19, 15, 18, 20}
```



Abbildung 8: Heap-Array – Kindelemente von `array[2]`

### 3 Die Implementierung

Das dritte Element mit  $i = 3$  hat die Kindknoten  $2*(i + 1) = 6$  und  $2*(i + 1) + 1 = 7$

```
int array[] = {-, 8, 5, 1, 16, 19, 15, 18, 20}
```




Abbildung 9: Heap-Array – Kindelemente von `array[3]`

Das vierte Element mit  $i = 4$  hat den Kindknoten  $2*(i + 1) = 8$

```
int array[] = {-, 8, 5, 1, 16, 19, 15, 18, 20}
```




Abbildung 10: Heap-Array – Kindelemente von `array[4]`

Um den Aufbau eines Heaps der Länge  $n$  durchzuführen, werden nacheinander immer wieder alle noch nicht sortierten Werte der Elternelemente mit denen ihrer Kinder verglichen und eventuell vertauscht. Bei einem Max-Heap muss das Elternelement den höchsten Wert erhalten, bei einem Min-Heap den kleinsten. Gestartet wird dazu bei  $n/2$ , da nur von 1 bis zu dieser Stelle Kinder vorhanden sein können. Im folgenden Beispiel wird ein Max-Heap aufgebaut.

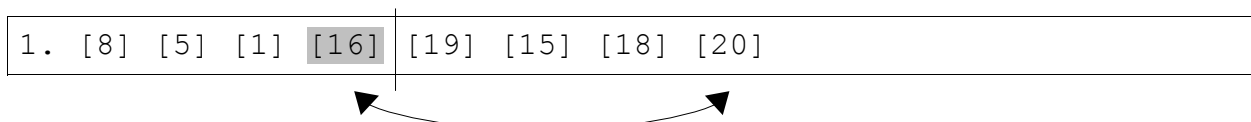
Beim Vergleich werden zuerst die Kinder des entsprechenden Elements untereinander verglichen und der größere Wert zum Vergleich mit dem Elternelement heran gezogen. Ist der Wert des Kindelements größer, wird er mit dem des Elternelements vertauscht.

Kommt es zum Tausch der Werte von Eltern- und Kindelement, muss getestet werden, ob das Kindelement seinerseits ebenfalls Kindelemente besitzt. Falls ja, muss hier ebenfalls ein entsprechender Wertevergleich durchgeführt werden.

Wurden die Vergleiche des  $n/2$ -ten Elements durchgeführt, wird das nächst linksliegende Element auf die gleiche Weise betrachtet.

Diese Betrachtung des nächsten links liegenden Elements wird solange wiederholt, bis der Vergleich beim ersten Element endet (zur Erinnerung: `array[1]`). Durch den Tausch aller Werte befindet sich nun das größte Element des Arrays an erster Stelle und der Max-Heap ist aufgebaut.

Konkret müssen für diesen Aufbau folgende Schritte durchlaufen werden:



Gestartet wird mit dem Vergleich immer links von der Mitte, da die Elemente rechts von der Mitte wie bereits erwähnt keine Kindelemente mehr besitzen. In diesem Fall wird also

### 3 Die Implementierung

mit [16] begonnen. Das einzige Kindelement ist [20]. Da er größer ist, werden die beiden vertauscht:

2. [8] [5] [1] [20] [19] [15] [18] [16]

Als nächstes Element wird das links neben dem zuvor betrachteten zum Vergleich heran gezogen: [1]. Die entsprechenden Kindelemente sind [15] und [18].

3. [8] [5] [1] [20] [19] [15] [18] [16]



Der größere Wert der beiden wird zum Vergleich heran gezogen und da er größer als der seines Elternelements ist, werden beide vertauscht:

4. [8] [5] [18] [20] [19] [15] [1] [16]

Danach wird das nächste links liegende Element betrachtet.

5. [8] [5] [18] [20] [19] [15] [1] [16]



Die Kindelemente [20] und [19] sind beide größer als [5], weswegen mit dem größeren der beiden Kindelemente vertauscht wird:

6. [8] [20] [18] [5] [19] [15] [1] [16]

Da das Element an vierter Stelle ebenfalls ein Kindelement besitzt, muss hier ebenfalls verglichen werden.

7. [8] [20] [18] [5] [19] [15] [1] [16]



Da [16] größer [5] ist, wird auch hier getauscht.

8. [8] [20] [18] [16] [19] [15] [1] [5]

Danach wird links von der Stelle fortgefahren, an der zuvor aufgehört wurde. Es fehlt noch der Vergleich des ersten Elements mit seinen Kindelementen.

9. [8] [20] [18] [16] [19] [15] [1] [5]



Hier müssen [8] und [20] vertauscht werden, da [20] der höhere Wert der Kindelemente ist:

### 3 Die Implementierung

10. [20] [8] [18] [16] [19] [15] [1] [5]

Die [8] besitzt wiederum Kindelemente, die zum Vergleich herangezogen werden:

11. [20] [8] [18] [16] [19] [15] [1] [5]

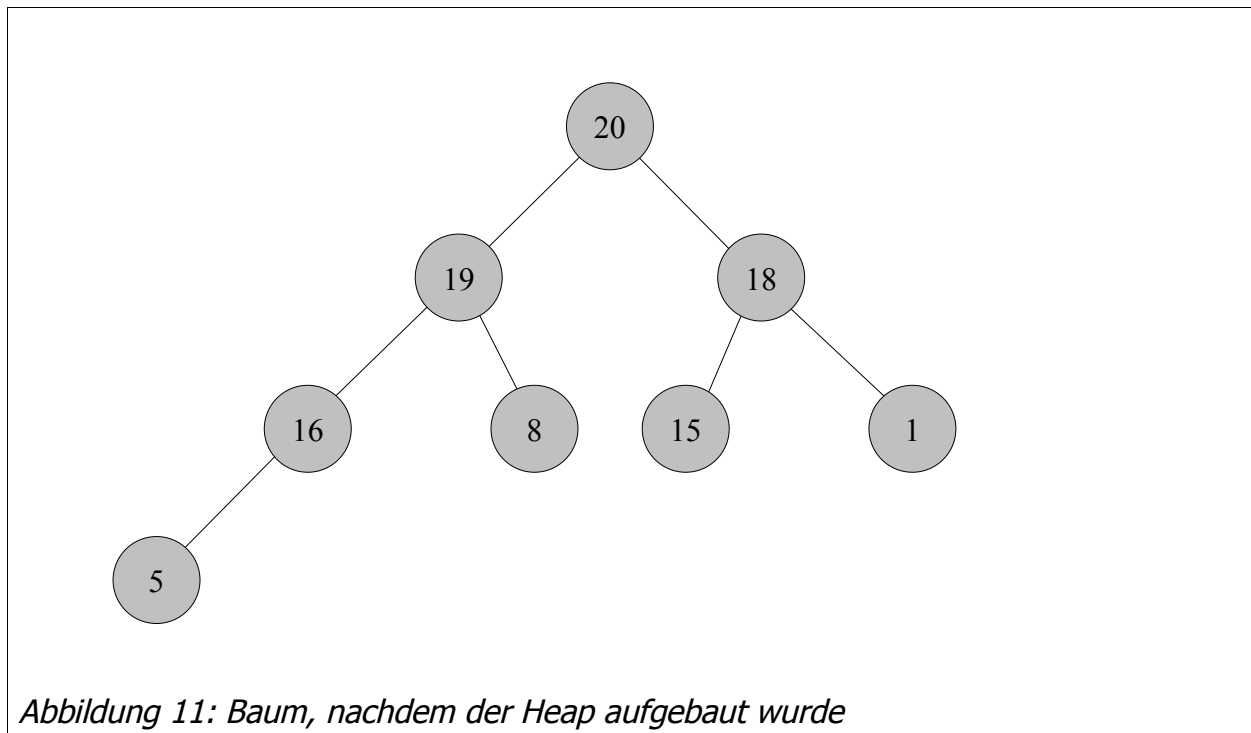


Von [16] und [19] wird der größere der beiden zum Tausch heran gezogen:

12. [20] [19] [18] [16] [8] [15] [1] [5]

Der Wert [8] besitzt keine Kindelemente mehr, somit ist hier der Aufbau des Heaps beendet.

Der Max-Heap würde sich als Binärbaum so darstellen:



Deutlich sieht man, dass jedes Elternelement einen größeren Wert als ein Kindelement hat.

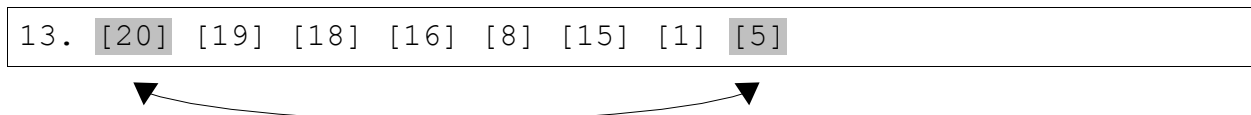
#### 3.1.1.1.2 Wie funktioniert die Aktualisierung eines Heaps?

Nach dem Aufbau des Max-Heaps befindet sich der größte Wert an erster Stelle. Wird dieser entfernt, tritt das letzte Element an seine Stelle. Das Array wird somit um eins verkürzt. Das nun an erster Stelle stehende Element muss solange „versickert“ werden, bis

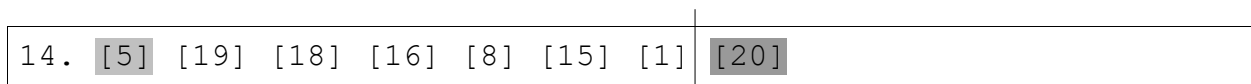
### 3 Die Implementierung

der Heap wieder aufgebaut ist. Hierbei durchläuft es solange den Heap „nach unten“, bis es seine entsprechende Position erreicht hat. Alle anderen Elternelemente sind bereits aus dem ersten Aufbau größer als ihre Kindelemente und müssen nicht mehr sortiert werden.

Dieses „Herausnehmen“ des größten Wertes wird solange wiederholt, bis alle Elemente des Arrays nacheinander der Größe nach sortiert ausgegeben wurden. Am Beispiel setzt sich das wie folgt fort:

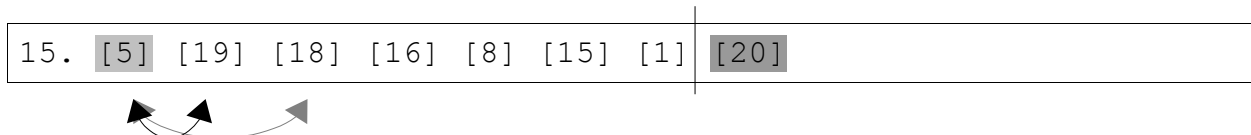


Das erste Element wird mit dem letzten vertauscht und die Anzahl der zu betrachtenden Elemente um eins herabgesetzt:

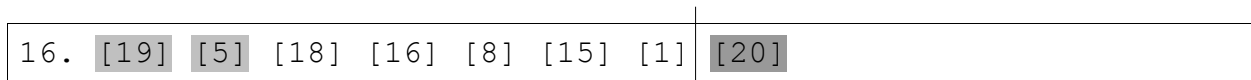


An letzter Stelle befindet sich der größte Wert, der jetzt nicht mehr betrachtet wird.

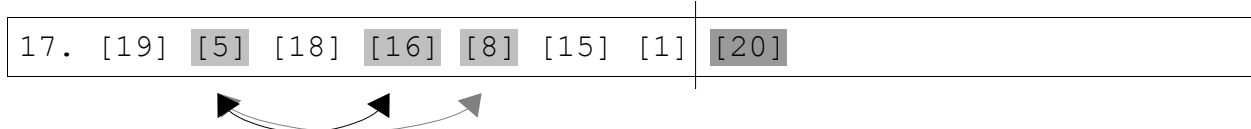
Das erste Element [5] wird mit seinen Kindelementen [19] und [18] verglichen:



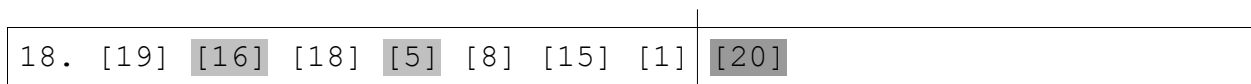
Das Element an zweiter Stelle ist größer und wird somit mit dem ersten Element vertauscht. Der „Versicker-Vorgang“ des Elements [5] hat begonnen.



Als Nächstes müssen die neuen Kindelemente von [5] betrachtet werden.



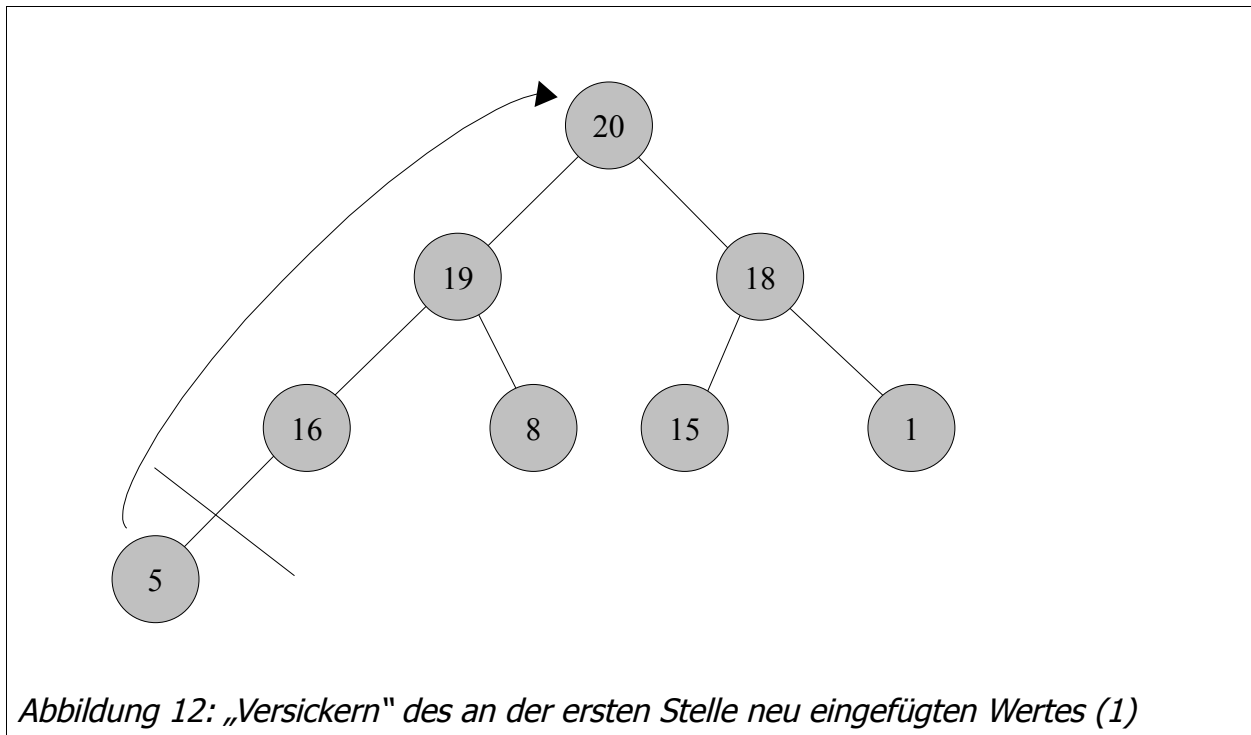
Dies führt zum Tausch mit [16]:



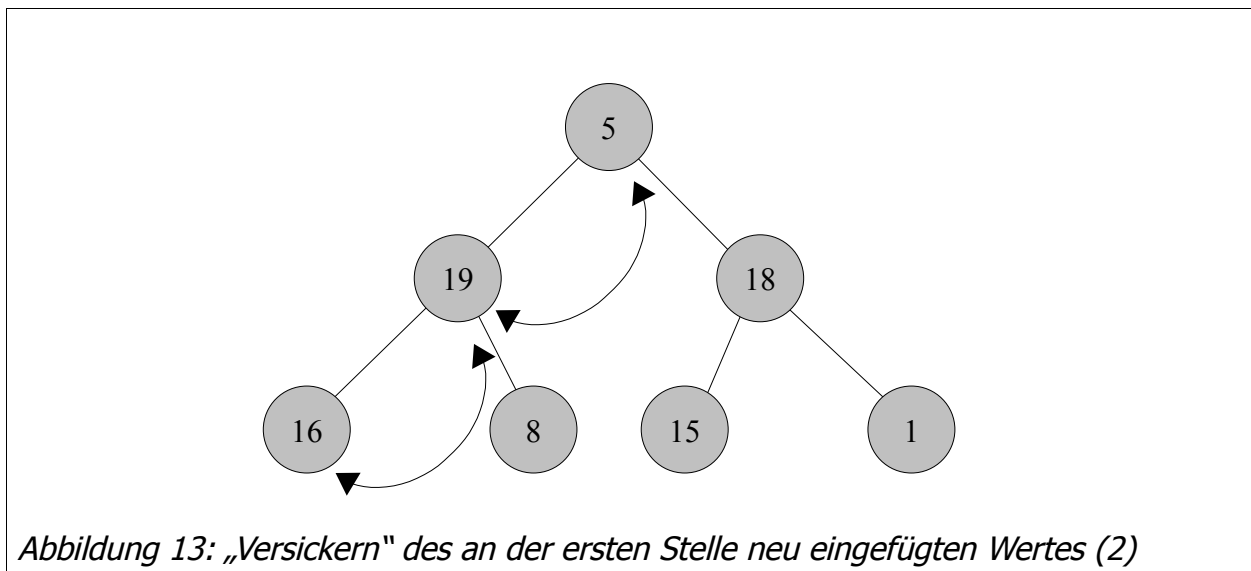
Es gibt keine weiteren Kindelemente zu betrachten. Der Heap ist somit wieder aufgebaut, und der zweitgrößte Wert des Heaps befindet sich an der ersten Stelle.

Am Binärbaum lässt sich dieses „Versickern“ gut veranschaulichen. Zuerst gelangt der Wert „5“ ins Wurzelement:

### 3 Die Implementierung



Dieser Wert wird solange mit den Kindelementen verglichen und vertauscht, bis der Heap wiederhergestellt ist:



Die „19“ und die „16“ haben sich jeweils um eine Ebene nach oben verschoben, die „5“ befindet sich nun in der untersten Ebene:

### 3 Die Implementierung

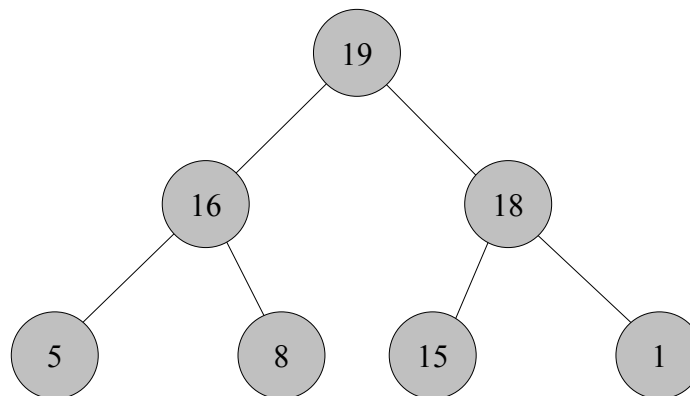


Abbildung 14: „Versickern“ des an der ersten Stelle neu eingefügten Wertes (2)

Der Heap ist wieder aufgebaut, der zweitgrößte Wert des Arrays befindet sich an der ersten Stelle und kann entnommen bzw. mit dem letzten betrachteten Element vertauscht werden. Danach wird der Heap wieder aufgebaut. Das Element [1] muss „versickert“ werden:

19.	[1]	[16]	[18]	[5]	[8]	[15]	[19]	[20]
-----	-----	------	------	-----	-----	------	------	------

Zu Beginn wird Element [1] wird mit seinem größeren Kindelement [18] vertauscht:

20.	[18]	[16]	[1]	[5]	[8]	[15]	[19]	[20]
-----	------	------	-----	-----	-----	------	------	------

Der Wert [1] ist kleiner als das des nächsten durch den Tausch entstandenen Kindelements [15], somit werden diese beiden ebenfalls vertauscht:

21.	[18]	[16]	[15]	[5]	[8]	[1]	[19]	[20]
-----	------	------	------	-----	-----	-----	------	------

Der Heap ist wiederum aufgebaut, der Wert [1] „versickert“, das drittgrößte Element befindet sich an erster Stelle und kann nun mit dem drittletzten Element vertauscht werden:

22.	[1]	[16]	[15]	[5]	[8]	[18]	[19]	[20]
-----	-----	------	------	-----	-----	------	------	------

Zum erneuten Aufbau muss wieder das Element [1] „versickert“ werden. Es kommt zum Tausch mit seinem Kindelement [16]:

23.	[16]	[1]	[15]	[5]	[8]	[18]	[19]	[20]
-----	------	-----	------	-----	-----	------	------	------

An zweiter Stelle muss es wiederum mit seinen Kindelementen verglichen werden, was zum Tausch mit [8] führt:

### 3 Die Implementierung

24. [16] [8] [15] [5] [1] [18] [19] [20]

An dieser Stelle ist sichergestellt, dass an erster Stelle das viertgrößte Element steht, da [1] „versickert“ ist. Erstes Element und viertletzte werden vertauscht:

25. [1] [8] [15] [5] [16] [18] [19] [20]

Durch den Vergleich mit den Kindknoten muss das erste Element (wieder [1], was in diesem Beispiel rein zufällig so oft in Folge passiert) mit seinem Kindelement [15] vertauscht werden:

26. [15] [8] [1] [5] [16] [18] [19] [20]

Bereits jetzt befindet sich das fünftgrößte Element an erster Stelle des Heaps, da [1] bereits bis an seine richtige Stelle „versickert“ ist. Das erste Element kann mit dem fünftletzte vertauscht werden:

27. [5] [8] [1] [15] [16] [18] [19] [20]

Das erste Element wird mit seinen Kindelementen verglichen, folglich müssen [5] und [8] vertauscht werden:

28. [8] [5] [1] [15] [16] [18] [19] [20]

Schon ist das sechstgrößte Element an erster Stelle und wird mit dem sechstletzte Element vertauscht:

29. [1] [5] [8] [15] [16] [18] [19] [20]

Das erste Element wird mit seinem Kindknoten verglichen und muss folglich vertauscht werden:

30. [5] [1] [8] [15] [16] [18] [19] [20]

So befindet sich das zweitletzte größte Element an erster Stelle und kann mit seinem vorbestimmten Platz vertauscht werden:

31. [1] [5] [8] [15] [16] [18] [19] [20]

Übrig bleibt nur noch ein Element, welches automatisch das letzte größte Element der zu betrachtenden Elemente darstellt.

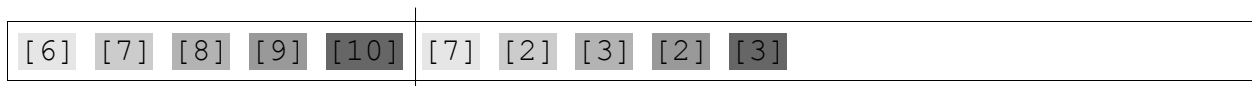
Somit wurden nacheinander die jeweils größten Werte ermittelt und das resultierende Array enthält diese aufsteigend sortiert.

## 3 Die Implementierung

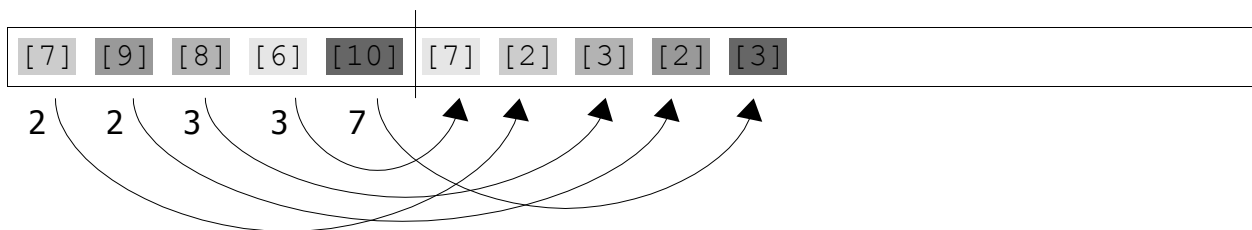
### 3.1.1.1.3 Der Heap für die Implementierung des Huffman Codes

Bei der Implementierung des Huffman Algorithmus wird kein normaler Heap verwendet. Es wird ein Array mit doppelter Länge angelegt, in dem in der zweiten Hälfte die zu sortierenden Werte gespeichert werden, die erste Hälfte enthält „Zeiger“ auf die Werte in der zweiten Hälfte. Sortiert werden nur die Zeiger, nicht die Werte selber. Diese auf den ersten Blick etwas umständliche Art wird benötigt, um später auf recht einfache Weise die Codewortlänge zu berechnen.

Zu Beginn könnte ein solches Array folgendermaßen aussehen:

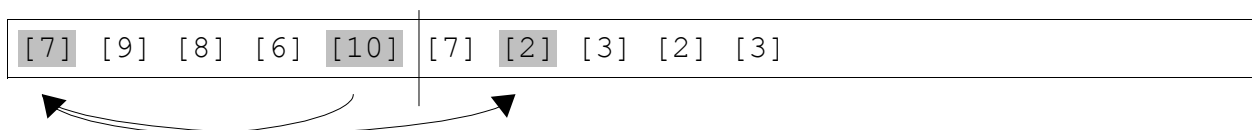


Die Elemente 1 bis 5 zeigen auf die Elemente 6 bis 10. Nach dem Aufbau eines in diesem Fall jetzt Min-Heaps hat sich die zweite Hälfte nicht verändert, nur die Zeiger sind entsprechend umsortiert:

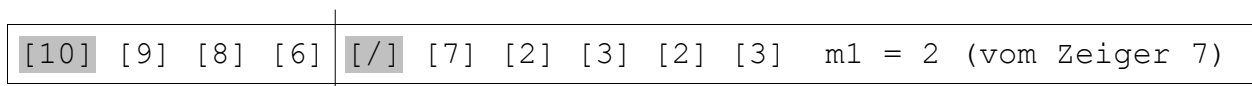


Dieses Array kann weiter zu einem sortierten Baum entwickelt werden, aus dem sich am Ende durch die Tiefe zu jedem Blatt die entsprechende Codewortlänge eines jeden Zeichens ablesen lässt. Dazu werden die beiden kleinsten Werte in der zweiten Hälfte gesucht und addiert im letzten Element der ersten Hälfte gespeichert. Zusätzlich werden in ihren ursprünglichen Speicherorten Zeiger auf den neuen Speicherort der Summe gespeichert. Außerdem wird im ersten Element ein Zeiger auf den Speicherort der Summe gesetzt, damit dieser als mit zu berücksichtigender Wert bei der Berechnung des Baums beachtet und somit quasi aus der Gruppe der Zeiger (zu Beginn die erste Hälfte) ausgeschlossen wird.

Beim ersten Durchlauf ist der Zeiger [7] an erster Stelle, welcher auf den Wert „2“ zeigt.

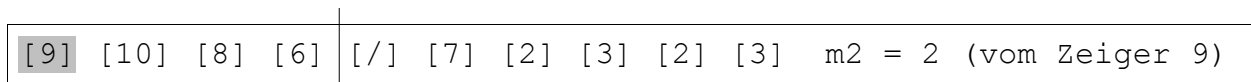


Der Wert der ersten Stelle wird zwischengespeichert (m1) und die letzte Stelle der ersten Hälfte des Arrays wird an die erste Stelle geholt:

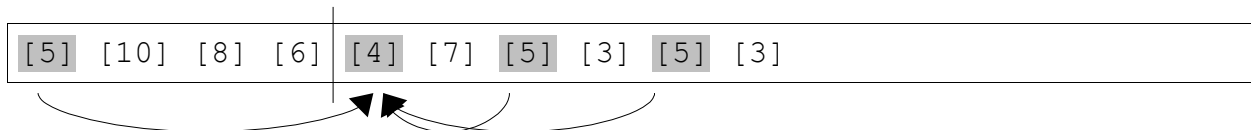


Die [7] verschwindet. Sie wird als Zeiger nicht mehr benötigt, da dieses Blatt nun nicht mehr betrachtet werden muss. Der Zeiger [10] an erster Stelle wird wieder „versickert“, wobei das letzte Element der ersten Hälfte im Array nicht betrachtet wird (der Wert hier ist momentan uninteressant und wird im Folgenden überschrieben):

### 3 Die Implementierung

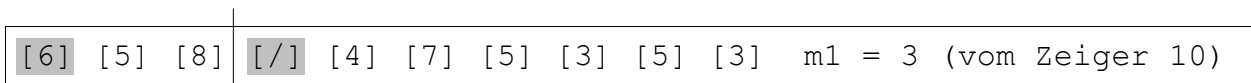


An der ersten Stelle befindet sich nun der zweitkleinste Wert des Heaps, in diesem Fall der Zeiger [9], dessen Wert „2“ ebenfalls zwischengespeichert wird ( $m_2$ ). Diese gespeicherten Werte  $m_1$  und  $m_2$  werden addiert und an der letzten Stelle der ersten Hälfte gespeichert, die beim zweiten Aufbau des Heaps unbeachtet geblieben ist. Danach werden Zeiger von den ehemaligen Speicherorten und vom ersten Element auf diese Stelle im Array gesetzt:



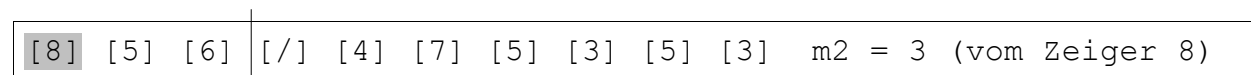
[9] als Zeiger auf ein Blatt ist verschwunden, da dieses ebenfalls nicht mehr betrachtet werden muss.

Der Zeiger [5] an der ersten Stelle wird nun erneut solange im Heap „versickert“, bis der Heap wieder aktualisiert ist. Dabei wird der neue Wert, die Summe aus  $m_1$  und  $m_2$  an der Stelle 5, als zu sortierender Wert mit betrachtet (der Zeiger an der ersten Stelle zeigt auf ihn, er ist ein neu entstandener „Knoten“ aus zwei Blättern). Der Zeiger [10] gelangt an die erste Stelle, dessen Wert „3“ wird zwischengespeichert und der letzte Zeiger des zu sortierenden Teils an die erste Stelle gesetzt:

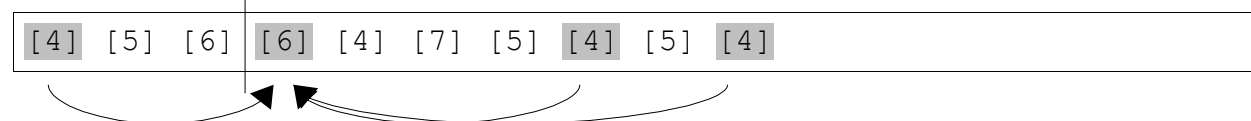


Der Zeiger [10] verschwindet, auch dieses Blatt muss nun nicht mehr betrachtet werden.

Der nächste Aufbau des Heaps wird wie oben über ein Element weniger durchgeführt, an erster Stelle befindet sich der Zeiger [8] auf das zweitkleinste Element „3“:

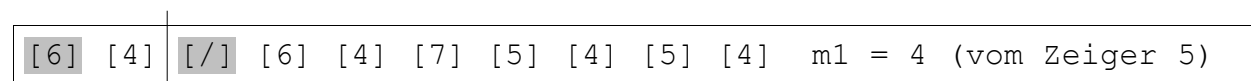


Die Summe beider Elemente wird im „leeren“ Element gespeichert, auf welches wie zuvor wieder drei Zeiger gesetzt werden:



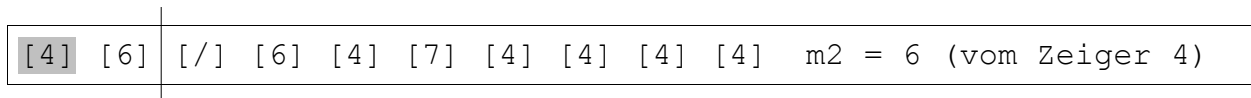
Auch der Zeiger [8] ist nun verschwunden, das Blatt wurde in einem Knoten verarbeitet.

Beim Aufbau des Heaps über die ersten drei Elemente wird der nächst kleinere Wert „4“ gespeichert, der entsprechende Zeiger [5] gelöscht und der Zeiger von der dritten Stelle an seine gesetzt:

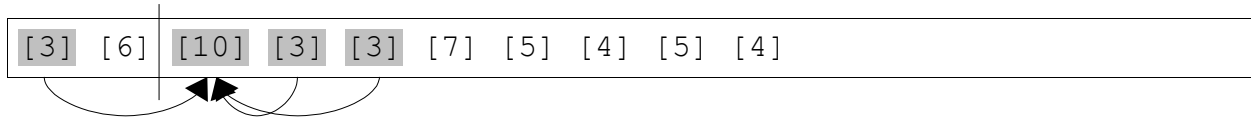


Erneutes „Versickern“ des ersten Elements:

### 3 Die Implementierung

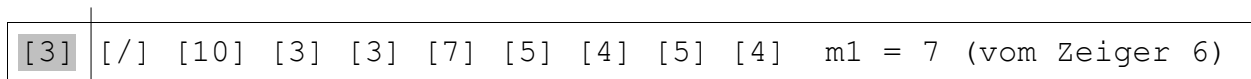


Die Summe der beiden Werte wird an dritter Stelle gespeichert, wiederum werden drei entsprechende Zeiger gesetzt:

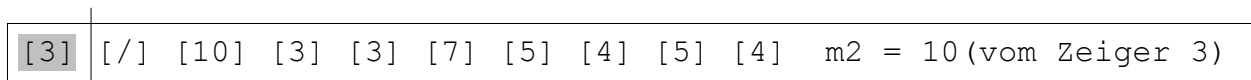


Fast alle Zeiger auf zu verarbeitende Werte sind gelöscht, übrig sind nur noch [3] mit dem Wert „10“ und [6] mit dem Wert „7“.

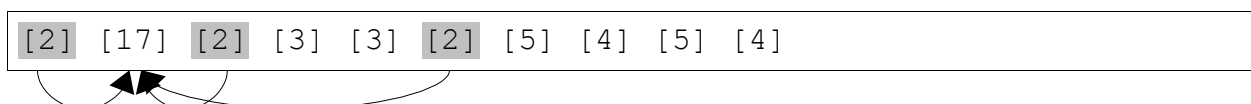
Ein „Versickern“ des ersten Elements ergibt als nächst kleineren Wert „7“ zuerst den Zeiger auf Element [6], welches in  $m_1$  gespeichert wird, um danach mit dem letzten Element zu vertauschen:



Übrig bleibt nun der Zeiger [3], dessen Wert „10“ in  $m_2$  gespeichert wird:



Die Summe der beiden letzten Werte wird an zweiter Stelle im Heap gespeichert, Zeiger im ersten, dritten und sechsten Element werden gesetzt:

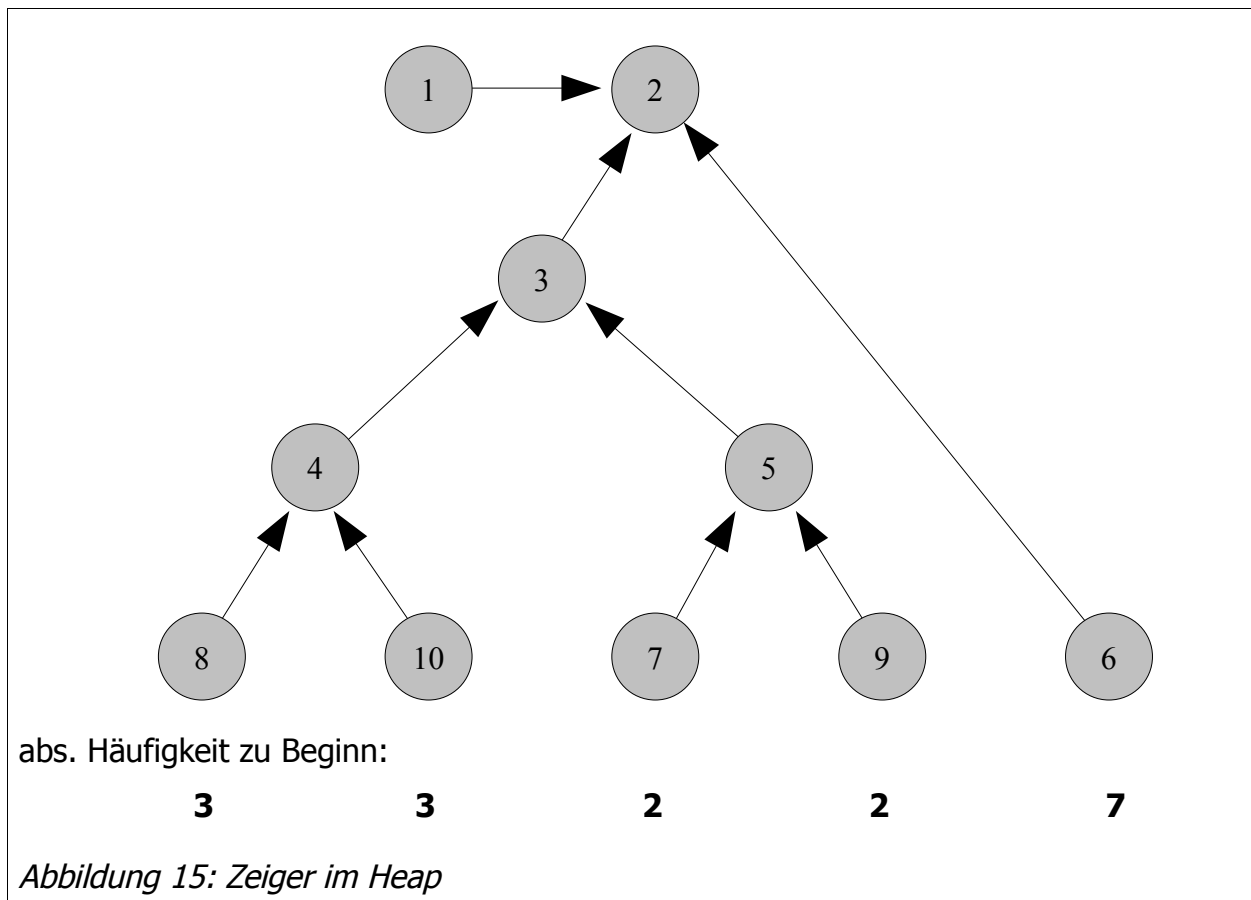


An dieser Stelle ist die Erstellung des „Baums“ zur Berechnung der Tiefe beendet. Das Array besteht nur noch aus einem Zeiger auf ein „Wurzelement“ und aus Zeigern von Kinder- auf Elternelemente (im letzten Fall zeigen zwei Kindelemente auf das Wurzelement), welche durch die ständige Wiederherstellung des Heaps und des Wegnehmens des kleinsten Elements erstellt wurden.

Häufig vorkommende Zeichen sollen kurze Codeworte bekommen, seltener vorkommende längere. Genau dies wird durch diese „Zeigersortierung“ erreicht. Die selten vorkommenden Zeichen haben eine kleine absolute Häufigkeit, durchlaufen also öfter die o.g. Berechnung, weil sie zu anderen kleinen Häufigkeiten addiert werden und wiederum eine verhältnismäßig kleine Summe ergeben.

Im Prinzip macht diese „Zeigersortierung“ genau das, was im zweiten Kapitel zur Huffman Codierung beschrieben wurde: sie erstellt den Baum, indem immer die beiden kleinsten Werte addiert werden und einen neuen Knoten mit dem Wert der Summe ergeben:

### 3 Die Implementierung



An den Stellen 6 bis 10 waren zu Beginn die absoluten Häufigkeiten der verschiedenen Zeichen. Bei diesem kleinen Beispiel lässt sich nun gleich auf den ersten Blick erkennen, dass das Zeichen, welches siebenmal vorkommt, mit einem 1-Bit-Codewort codiert wird und die vier anderen Zeichen ein 3-Bit-Codewort erhalten. Um die Tiefe aus den Zeigern zurück zu rechnen gibt es zwei verschiedene Möglichkeiten.

Bei der ersten Möglichkeit werden einfach die Zeiger von jedem Blatt aus bis zum Wurzelement in einer Schleife hochgezählt und an entsprechender Stelle im Array gespeichert. Je nach Auftrittshäufigkeit der verschiedenen Zeichen kann dies relativ viel Zeit in Anspruch nehmen.

Eleganter und schneller aber vielleicht etwas komplizierter zu verstehen ist die zweite Möglichkeit. Sie basiert darauf, dass die Zeiger von den Kindern ausgehen und auf ihre Elternknoten zeigen, also eigentlich von rechts nach links zu lesen sind. Bei der Betrachtung von links nach rechts wird zuerst der Elternknoten betrachtet, bevor eins der Kinder betrachtet wird. In diesem wird die aktuelle Tiefe gespeichert. Im Array befindet sich an zweiter Stelle das Wurzelement. Somit muss der Wert an dieser Stelle auf 0 gesetzt werden, da sich das Wurzelement stets auf der 0ten Ebene befindet:

### 3 Die Implementierung

[2] [0] [2] [3] [3] [2] [5] [4] [5] [4]

Ab der dritten Stelle wird der Wert des Elternknotens aufgerufen, auf den gezeigt wird. Zu diesem wird 1 addiert (die Tiefe nimmt um 1 zu) und die Summe an der aktuellen Stelle gespeichert.

Im ersten Fall heißt dies, dass im Array an dritter Stelle auf die zweite Stelle verwiesen wird. Der Wert hier beträgt 0. Wird eine 1 addiert, besagt nun der aktuelle Wert, dass sich der Knoten eine Ebene unter dem Wurzelement befindet:

[2] [0] [1] [3] [3] [2] [5] [4] [5] [4] (= 0 + 1)



An der vierten Stelle befindet sich wie der der fünften Stelle ein Zeiger auf das dritte Element. Für beide gilt, dass sie sich eine Ebene tiefer befinden als 1, womit beide den Wert 2 erhalten:

[2] [0] [1] [2] [2] [2] [5] [4] [5] [4] (= 1 + 1)

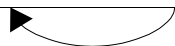


Dies wird bis zum Ende des Heaps wiederholt:

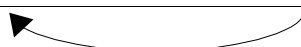
[2] [0] [1] [2] [2] [1] [5] [4] [5] [4] (= 0 + 1)



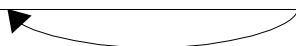
[2] [0] [1] [2] [2] [1] [3] [4] [5] [4] (= 2 + 1)



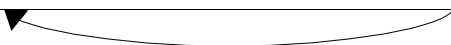
[2] [0] [1] [2] [2] [1] [3] [3] [5] [4] (= 2 + 1)



[2] [0] [1] [2] [2] [1] [3] [3] [3] [4] (= 2 + 1)



[2] [0] [1] [2] [2] [1] [3] [3] [3] [3] (= 2 + 1)



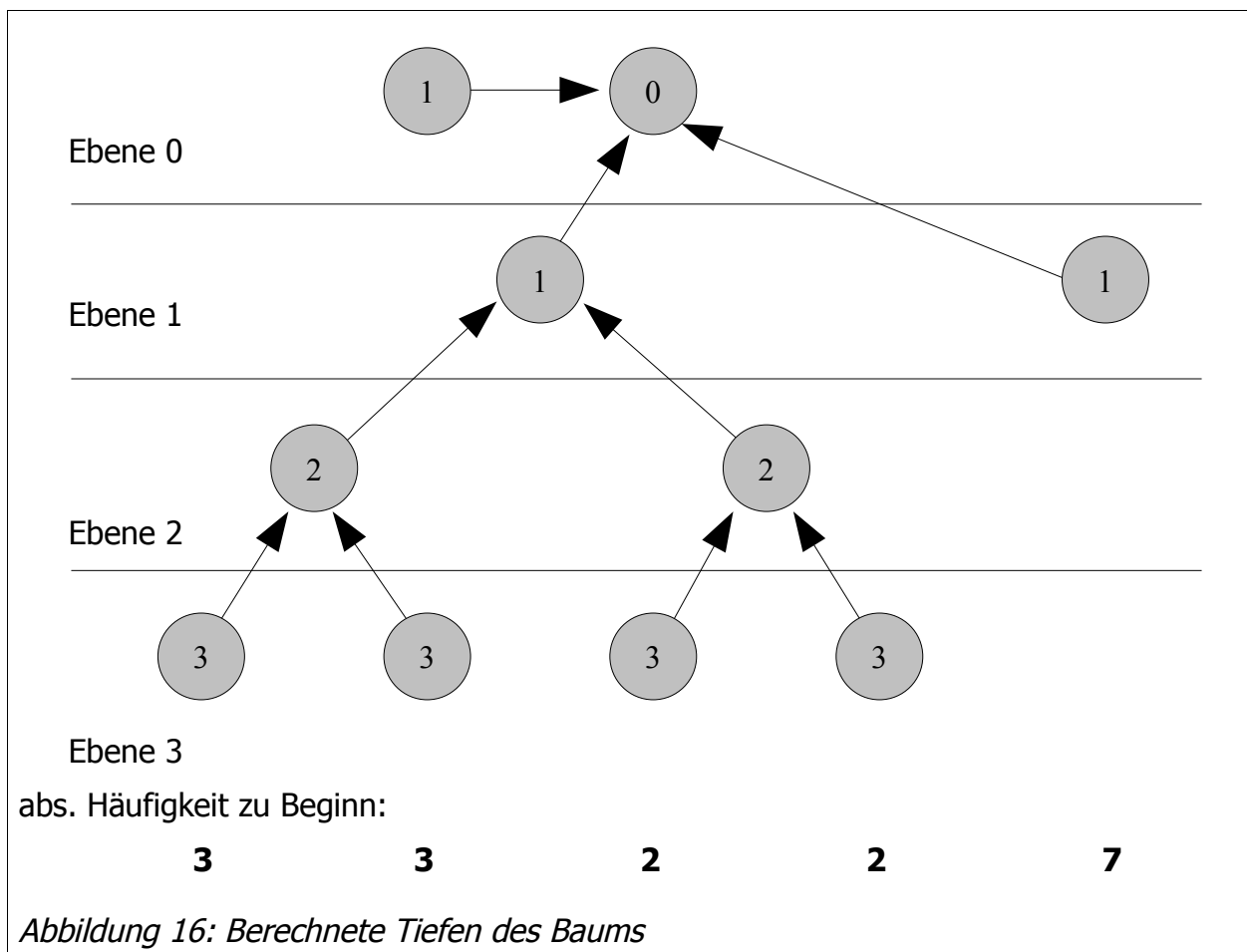
In der zweiten Hälfte des Arrays befinden sich nun die entsprechenden Codewortlängen für jedes Zeichen:

### 3 Die Implementierung

Zeichen Auftrittshäufigkeit	Codewortlänge
7	1
2	3
3	3
2	3
3	3

Tabelle 7: Berechnete Codewortlängen mithilfe des Heaps

Eine einfache Lösung, deren Durchlaufzeit etwa proportional zur Anzahl der Zeichen ist und völlig korrekt arbeitet, wie die folgende Abbildung zeigt:



## 3 Die Implementierung

### 3.1.2 Zuordnung des Codeworts

Nachdem die Codewortlänge für jedes Zeichen berechnet wurde, muss ermittelt werden, wie oft jede Codewortlänge  $l_i$  vorkommt. Diese Anzahl wird in  $numl[l_i]$  gespeichert.

Danach wird der erste Code jeder Codewortlänge berechnet. Im Array  $firstcode[l]$  werden diese gespeichert. Dazu wird der erste Code der längsten Codewortlänge auf 0 gesetzt ( $firstcode[maxlength]$ ). Um das erste Codewort der nächst kürzeren Länge zu berechnen, wird zum ersten Codewort der längsten Codewortlänge die Anzahl der Codewörter dieser Länge addiert und die Summe durch 2 geteilt. Dadurch wird quasi das letzte Bit bei der Berechnung wieder „abgeschnitten“, weil das gesuchte Codewort um ein Bit kürzer ist als das, aus dem der Wert berechnet wird.

```
numl[] = [ ][1][0][4]
firstcode[] = [ ][ ][ ][0]
count[] = [ ][3][1][1][1][1]
```

Im Beispiel „ABRAXAS“ bedeutet dies, dass es mit der Codewortlänge 3 vier verschiedene Zeichen gibt. Aus der Addition von 0 und 4 ergibt sich der Wert 4, die Hälfte beträgt 2, was das erste Codewort der Länge 2 ist. Dies besagt, dass also als Codewörter mit der Länge 3 „000“, „001“, „010“ und „011“ zur Verfügung stehen. Zur Vergabe des nächsten Codewortes wird das letzte 3-Bit-Codewort betrachtet, in diesem Fall das Codewort mit dem Dezimalwert 3, binär „011“. Von diesem wird die letzte Bitstelle entfernt, übrig bleibt „01“, was dem Dezimalwert 1 entspricht. Um die Präfixfreiheit des Codes zu garantieren, müssen alle Codewörter mit 2 Bits einen Dezimalwert größer als 1 haben. Somit kann als erstes Codewort der Dezimalwert 2 vergeben werden.

```
numl[] = [ ][1][0][4]
firstcode[] = [ ][ ][2][0]
count[] = [ ][3][1][1][1][1]
```

2-Bit-Codewörter kommen in dem Beispiel nicht vor, wie man am Array  $numl[]$  sieht. Folglich kann mit dem Wert 2 gleich der Dezimalwert des ersten 1-Bit-Codewortes auf die gleiche Weise gebildet werden. Von der „10“ (Dezimalwert 2) wird die letzte Stelle entfernt = „1“ (bzw. 2 durch 2 dividiert, was das selbe Ergebnis liefert). Da kein 2-Bit-Codewort existiert, bleibt es bei der 1, und es muss nicht mehr addiert werden.

```
numl[] = [ ][1][0][4]
firstcode[] = [ ][1][2][0]
count[] = [ ][3][1][1][1][1]
```

### 3 Die Implementierung

Jedem Zeichen muss nun noch das entsprechende Codewort zugewiesen werden. Dazu wird das „Hilfsarray“ `nextcode[l]` verwendet, welches zu Beginn eine 1:1-Kopie von `firstcode[l]` darstellt, also zu Beginn der erste Code jeder Codewortlänge gespeichert ist. Wurde ein Code bereits verwendet, wird der Wert im Array an entsprechender Stelle um 1 hochgezählt, damit kein Wert doppelt vergeben wird.

```
firstcode[] = [/][1][2][0]
nextcode[] = [/][1][2][0]
```

Für unserem Beispiel „ABRAXAS“ sieht das folgendermaßen aus:

Das „A“ bekommt ein 1-Bit-Codewort, der aktuelle Codewortwert ist lt. `nextcode['A']` 1, somit wird das „A“ mit „1“ codiert. Danach muss der Wert im `nextcode`-Array um 1 hochgezählt werden, damit das nächste Zeichen mit gleicher Codewortlänge ein neues Codewort bekommt. In diesem Fall könnte dies theoretisch vernachlässigt werden, da „A“ das einzige Zeichen ist, welches mit einem 1-Bit-Codewort codiert wird.

```
nextcode[] = [/][2][2][0]
```

Bei der Vergabe der 3-Bit-Codewörter werden die einzelnen Zeichen alphabetisch sortiert: „B“, „R“, „S“ und „X“. Daher bekommt „B“ den ersten Wert im `nextcode`-Array für 3-Bit-Codewörter: „000“. Danach wird der Wert im Array an der dritten Stelle um 1 herauf gezählt:

```
nextcode[] = [/][2][2][1]
```

Die gleiche Prozedur wird für die Codierung des Zeichens „R“ durchlaufen: der aktuelle Wert an dritter Stelle im Array beträgt 1, somit wird das Zeichen mit „001“ codiert, der Wert um eins herauf gezählt.

```
nextcode[] = [/][2][2][2]
```

Das Zeichen „S“ wird mit „010“ codiert, das „X“ entsprechend mit „011“. Danach sieht das `nextcode`-Array folgendermaßen aus:

```
nextcode[] = [/][2][2][4]
```

Es hat seinen Dienst erfüllt und wird nicht mehr benötigt.

### 3 Die Implementierung

Anhand der Berechnungen in Kapitel 2 wurde folgende Codierung erwartet:

Symbol $a_i$	$p(a_i)$	Code 1	Code 2	Code 3
B	1/7	010	110	<b>000</b>
R	1/7	001	100	<b>001</b>
S	1/7	011	111	<b>010</b>
X	1/7	000	101	<b>011</b>
A	3/7	1	0	<b>1</b>

Auf den ersten Blick lässt sich erkennen, dass die Codierung korrekt verlaufen ist. Das komplette Codewort für „ABRAXAS“:

```
100000110111010
```

Die zur Decodierung benötigten Arrays  $start[]$  und  $symbol[]$  werden folgendermaßen erstellt:

Beim Array  $start[]$  handelt es sich um die jeweiligen Positionen des ersten Codeworts der entsprechenden Codewortlänge. Somit lässt es sich einfach aus dem Array  $numl[]$  berechnen, welches die Anzahl jeder Codewortlänge beinhaltet.

```
start[] = [1][0][1][1]
```

Das Array  $start[]$  besagt nun, dass das erste Codewort der Länge 1 sich bei  $symbol[0]$  befinden wird, das erste Codewort der Länge 3 bei  $symbol[1]$ . Die anderen Werte sind beim Beispiel „ABRAXAS“ uninteressant.

Danach wird das Array  $symbol[]$  mit Werten gefüllt, welche später zur Decodierung benötigt werden. Auf das Zeichen „A“ wird später mit  $symbol[start[1] + 0]$  zugegriffen: es bekommt ein Codewort der Länge 1 und ist das erste dieser Länge. Über  $start[1]$  wird die entsprechende Startposition im Array  $symbol[]$  abgerufen. Da es sich beim „A“ um das erste Codewort dieser Länge handelt, wird nichts (also „0“) addiert.

```
symbol[start[1] + 0] = symbol[0] = A
```

Das Zeichen „B“ bekommt ein 3-Bit-Codewort und ebenfalls das erste. Daher wird über  $symbol[start[3] + 0]$  zugegriffen:

### 3 Die Implementierung

```
symbol[start[3] + 0] = symbol[1] = B
```

Das nächste 3-Bit-Codewort ist das „R“, auf welches über  $symbol[start[3] + 1]$  zugegriffen wird. Zum ersten Codewort der Länge 3 muss also zur Positionsbestimmung zur Position des ersten Codewortes der Länge 3 eine „1“ addiert werden.

```
symbol[start[3] + 1] = symbol[2] = R
```

Die Zeichen „S“ und „X“ werden auf die gleiche Weise gespeichert.

```
symbol[start[3] + 2] = symbol[3] = S
symbol[start[3] + 3] = symbol[4] = X
```

Somit sehen  $symbol[]$  und  $start[]$  folgendermaßen aus:

```
Codewortlänge: [1][3][3][3][3]
symbol[] = [A][B][R][S][X]
start[] = [0][3][6][9][12]
```

Im Array  $symbol[]$  sind die verwendeten Zeichen jetzt erst nach der Codewortlänge und dann alphabetisch sortiert. „A“ beginnt mit der Codewortlänge 1, „B“, „R“, „S“ und „X“ besitzen die Codewortlänge 3. Dass über Codewortlänge und Differenz des Codeworts zum ersten dieser Länge auf das Zeichen geschlossen werden kann, ermöglicht die sehr schnelle Decodierung.

#### 3.2 Decodierung

Zur Decodierung wird zuerst das Array  $firstcode[]$  benötigt.

Der zu decodierende Text wird bitweise ausgelesen und die aktuelle Länge dieser Bits zwischengespeichert. Solange dieser Wert kleiner als der erste Code der entsprechenden Länge ist, wird das nächste Bit eingelesen. Ist er größer oder gleich, wird das Array  $symbol[]$  verwendet, um das entsprechende Zeichen zu decodieren. Hierzu wird die Position des ersten Codeworts dieser Länge ermittelt und die Differenz zwischen erstem Codewort und dem eingelesenen Codewort addiert. Handelt es sich um das erste Codewort, ist die Differenz beispielsweise „0“ und die Position des ersten Codeworts liefert gleich das gesuchte Zeichen.

In der Implementierung wird nach jedem Einlesen eines Bits des zu decodierenden Textes der Dezimalwert dieser Bits mit dem entsprechenden  $firstcode$ -Wert verglichen. Solange er kleiner ist, muss ein weiteres Bit eingelesen werden. Ist er nicht mehr kleiner, ist zu-

### 3 Die Implementierung

nächst die Länge des Codeworts bekannt. Wird das erste Codewort dieser Länge vom eingelesenen Codewort abgezogen, erhält man den Wert, um das wievielte Zeichen der entsprechenden Länge es sich handelt.

Zur Erinnerung, am Beispiel „ABRAXAS“ sehen die entsprechenden Arrays folgendermaßen aus:

```
firstcode[] = [1][2][0]
symbol[] = [A][B][R][S][X]
start[] = [0][1][2][3]
```

Im Array *numl[]* befinden sich die Angaben, dass es ein 1-Bit-Codewort gibt und vier 3-Bit-Codewörter. *firstcode[]* besagt, dass das erste Bit alle Codewörter länger als 1 Bit kleiner 1 ist und dass die ersten beiden Bits aller Codewörter länger als zwei Bits kleiner 2 ist. Die 0 an dritter und letzter Stelle besagt, dass es keine Codewörter länger als drei Bit gibt. Das zu decodierende Codewort:

```
100000110111010
```

Betrachtet man nun das erste Bit: „1“ hat als Dezimalwert ebenfalls „1“ und ist nicht kleiner als *firstcode[0] = „1“*, somit handelt es sich um ein 1-Bit-Codewort und ist in diesem Fall außerdem eindeutig, da es nur ein einziges 1-Bit-Codewort gibt.

Um das entsprechende Zeichen aus dem Array *symbol[]* auszulesen, muss der Wert bei  $symbol[start[1] + \text{Integerwert der eingelesenen Zeichen} - firstcode[1]] = symbol[0 + 1 - 1] = symbol[0]$  zurückgegeben werden:

```
symbol[0] = A
```

Somit wurde das erste Zeichen „1“ als „A“ decodiert.

Als nächstes wird eine „0“ eingelesen. Sie ist kleiner als *firstcode[0] = „1“*, somit wird ein weiteres Bit eingelesen, wieder eine „0“. Der Dezimalwert von „00“ beträgt ebenfalls „0“ und ist kleiner als *firstcode[1] = „2“*, ein weiteres Bit wird eingelesen: noch eine „0“. „000“ ist nicht mehr kleiner als *firstcode[2] = „0“*, somit handelt es sich um ein 3-Bit-Codewort. Das erste 3-Bit-Codewort hat den Dezimalwert „0“, somit handelt es sich binär um „000“. Beide Wörter voneinander abgezogen, das eingelesene Bitwort und das erste 3-Bit-Codewort, ergeben „0“. Somit handelt es sich um das erste Zeichen, für das ein 3-Bit-Codewort verwendet wurde, das „B“. Dies verrät  $symbol[start[3] + 0 - 0] = symbol[1] = „B“$ .

## 3 Die Implementierung

```
symbol[1] = B
```

Die nächsten drei Bit werden auf die gleiche Weise eingelesen, allerdings ergibt sich als Differenz die „1“. Damit steht das gesuchte Zeichen in  $symbol[start[3] + 1 - 0] = symbol[1 + 1] = symbol[2] = „R“$ .

```
symbol[2] = R
```

Auf diese Weise wird bis zum Ende decodiert:

```
symbol[0] = A  
symbol[4] = X  
symbol[0] = A  
symbol[3] = S
```

Natürlich kommt das vorher codierte Wort „ABRAXAS“ dabei heraus.

Diese Decodierung ist sehr schnell. Weil nur wenige Operationen durchgeführt werden müssen, kann der Prozessor den Cache-Speicher nutzen, da keine Daten während des Schleifendurchlaufs verloren gehen. Zusätzlich kann die Menge der zuerst eingelesenen Bits angepasst werden, indem überprüft wird, welche Codewortlänge mindestens verwendet wurde, damit vorher keine Schleifendurchläufe gemacht werden müssen.

Wenn die Codewortlänge und der Abstand des Codeworts zum ersten Code der Codewortlänge ermittelt wurde, kann allerdings beim Abfragen des Zeichens ein Cacheverlust auftreten, da immer nur eine begrenzte Anzahl an Zeichen im Cache vorgehalten werden können und diese Grenze bei sehr großen Arrays überschritten wird.

### 3.3 Speicherverbrauch und Geschwindigkeit

Im Folgenden soll kurz erläutert werden, wie groß der Speicheraufwand ist und wie sich die Geschwindigkeiten beim Codieren bzw. Decodieren verhalten.

#### 3.3.1 Der Aufbau im Heap und erneutes Versickern

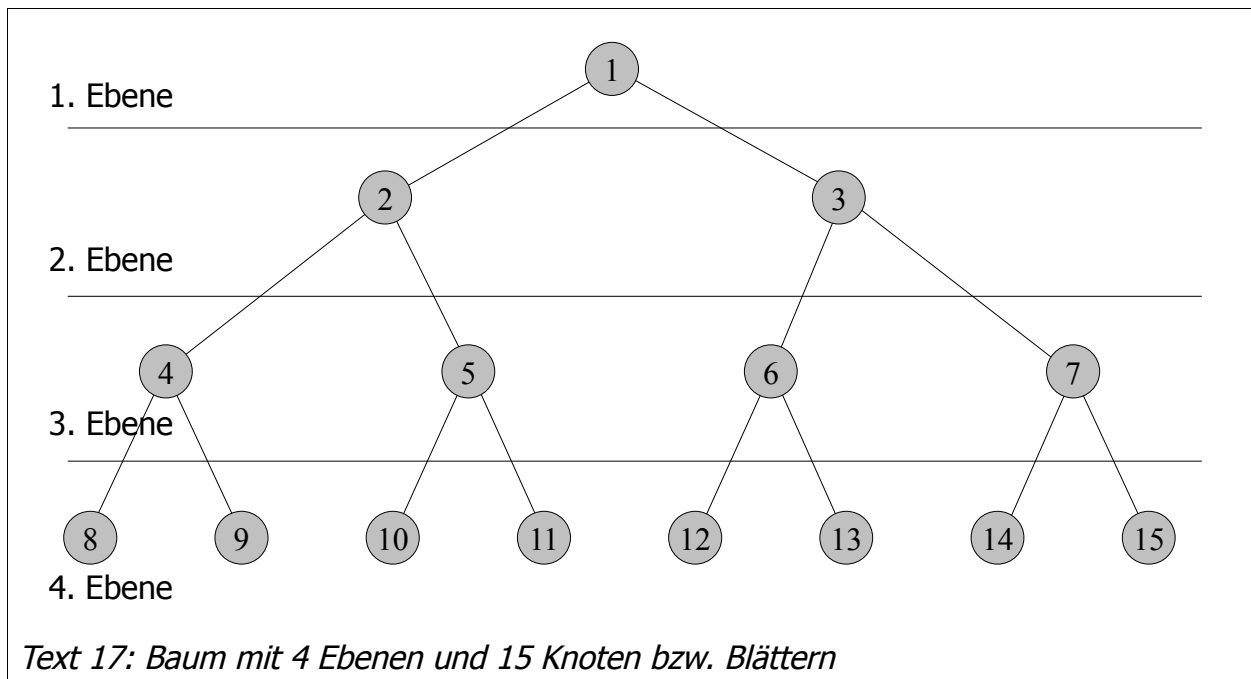
Ein Baum mit  $n$  Einträgen hat die Tiefe  $\log_2 n$  (aufgerundet). Für die Vorsortierung im Heap werden annähernd  $2 * n$  Vergleiche bzw. Vertauschungen benötigt. Dies soll im Folgenden bewiesen werden.

Zu Beginn wird angenommen, dass ein Vorgang daraus besteht, dass die beiden Kindele-

### 3 Die Implementierung

mente eines Elements verglichen und eventuell mit dem Elternelement vertauscht werden: es können also bis zu zwei Aktionen in einem einzigen Vorgang durchgeführt werden.

Ein Baum mit  $n = 15$  Einträgen hat die Tiefe  $\log_2 n = \log_2 15 \approx 4$ , er besitzt also 4 Ebenen. Äquivalent sei  $n = 2^k - 1$ , also  $k = 4$  Ebenen:

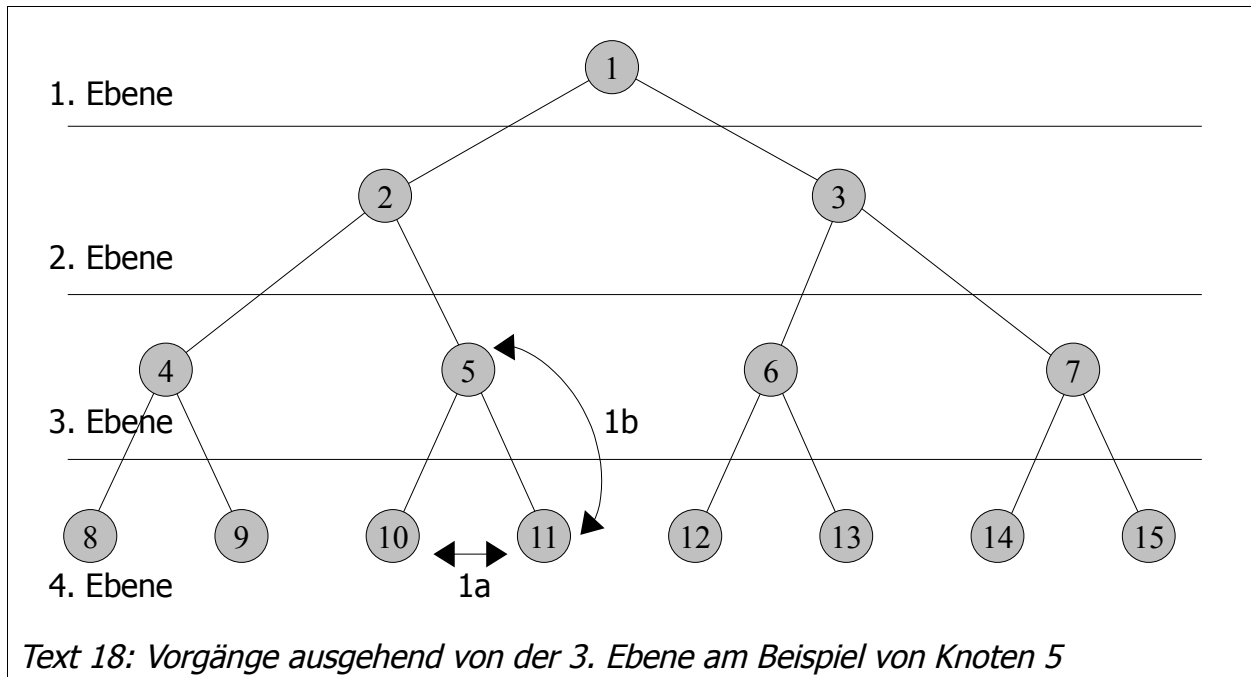


Hierbei handelt es sich um einen „vollen“ Baum, da alle Elemente der dritten Ebene zwei Kindelemente besitzen.

In der letzten Ebene befinden sich  $2^{k-1}$  Blätter, im Beispiel also  $2^3 = 8$  Blätter. Hier müssen noch keine Vorgänge durchgeführt werden. Mit den Vergleichen begonnen wird beim Knoten  $n/2 = 7$  auf der vorletzten Ebene. Hier befinden sich  $2^{k-2}$  Knoten, also  $2^2 = 4$  Knoten.

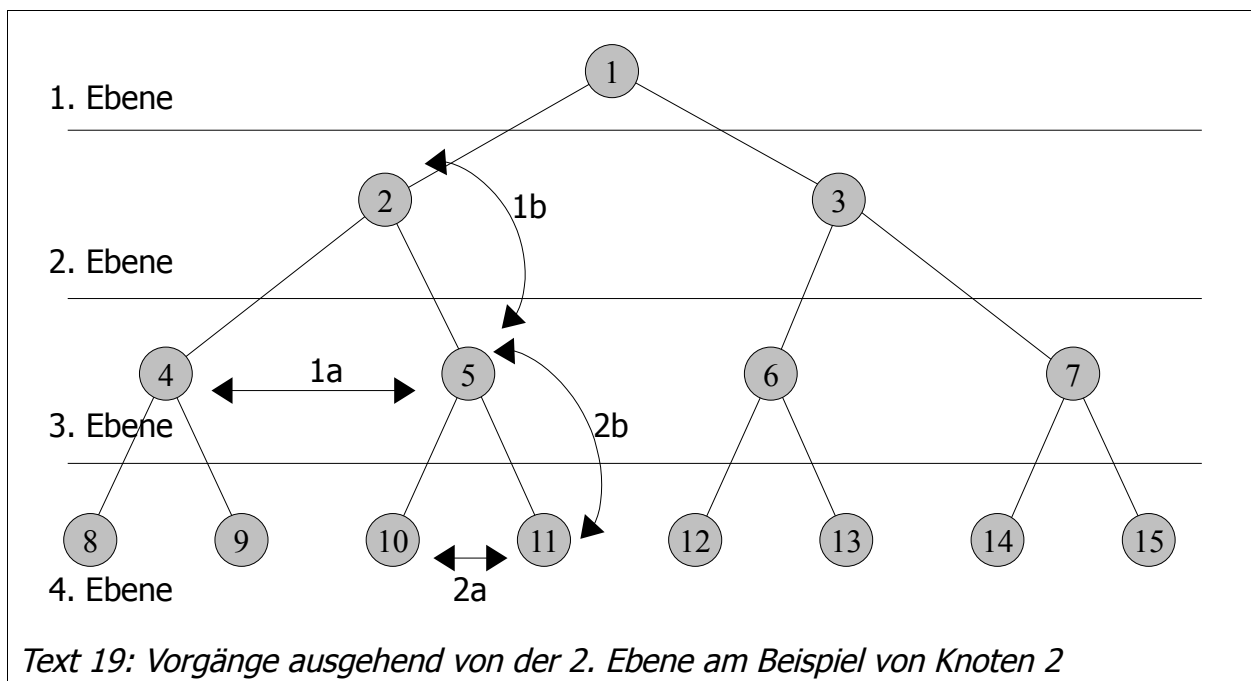
Am Beispiel des 5. Knotens werden seine beiden Kindelemente 10 und 11 verglichen und eventuell ein Tausch durchgeführt:

### 3 Die Implementierung



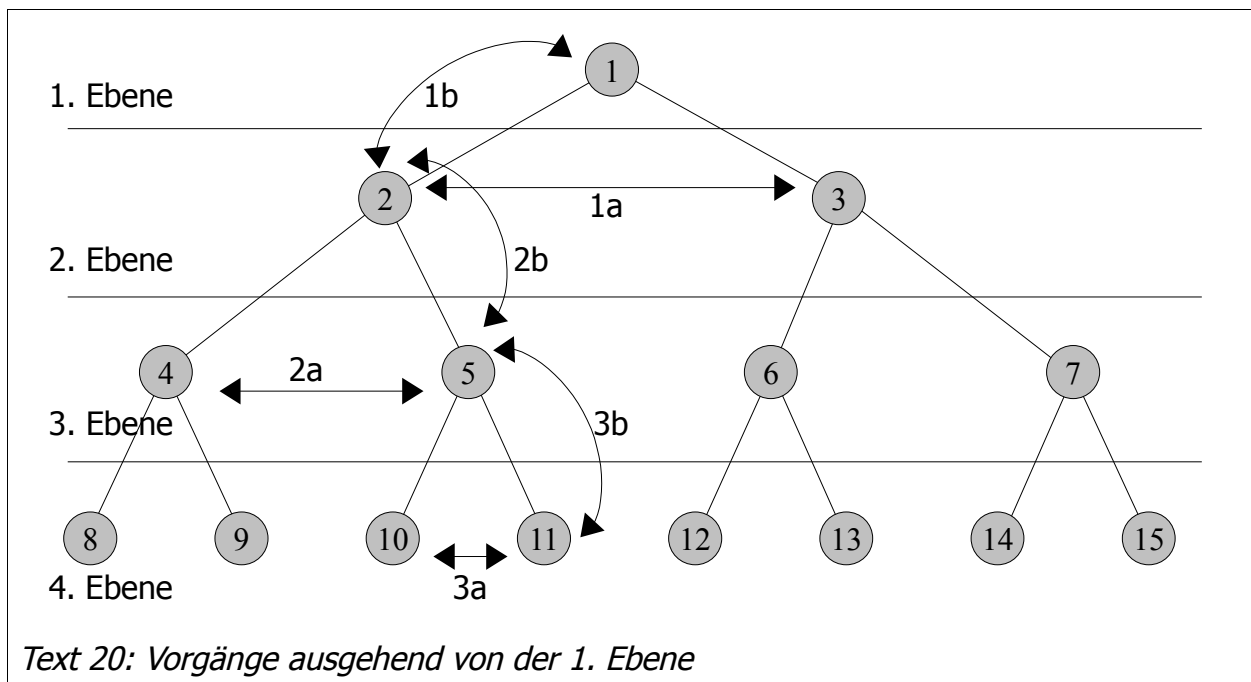
Pro Element wird hier jeweils höchstens ein Vorgang durchgeführt =  $2^{k-2} * 1 = 2^2 * 1 = 4$  Vorgänge auf der vorletzten Ebene.

Auf der vorvorletzten Ebene befinden sich  $2^{k-3}$  Knoten, somit also  $2^1 = 2$  Knoten. Zum Vergleich müssen hier jeweils maximal zwei Vorgänge bis zur untersten Ebene durchgeführt werden:



### 3 Die Implementierung

Beispielsweise werden die Kindelemente des zweiten Knotens verglichen und mit einem von ihnen ein Tausch durchgeführt, im Beispiel mit dem Knoten Nr. 5. Danach wird beim getauschten Element eine Ebene tiefer der gleiche Vorgang durchgeführt: insgesamt sind also  $2^{k-3} * 2 = 2^1 * 2 = 4$  Vorgänge möglich. Dies führt sich entsprechend bis zum Wurzelement fort.



Im Beispiel sind im letzten Durchgang  $2^{k-4} * 3 = 2^0 * 3 = 3$  Vorgänge möglich. Insgesamt ergibt dies  $4 + 4 + 3 = 11$  Vorgänge. Anhand von allgemeinen Formeln soll die Berechnung überprüft werden. Die gesamten Vorgänge addieren sich folgendermaßen:

$$2^{k-2} * 1 + 2^{k-3} * 2 + 2^{k-4} * 3 + \dots + 2^0 * (k-1) = \sum_{i=0}^{k-2} 2^{(k-2)-i} * (i+1)$$

umformen:

$$2^{k-2} \sum_{i=0}^{k-2} 2^{-i} * (i+1) = 2^{k-2} \sum_{i=0}^{k-2} \left(\frac{1}{2}\right)^i * (i+1) = 2^{k-2} * f\left(\frac{1}{2}\right)$$

$$\text{mit } f(x) = \sum_{i=0}^{k-2} x^i * (i+1)$$

### 3 Die Implementierung

Um  $f(\frac{1}{2})$  zu berechnen, wird die Funktion zuerst integriert. Aus dem Integral gewinnt man eine geschlossene Formel, die dann wieder abgeleitet wird. Der „Inhalt“ wird auf diese Weise nicht verändert, jedoch erhält man so eine ebenfalls geschlossene Darstellung:

$$f(x) = \int f(x) dx = \int \sum_{i=0}^{k-2} x^i * (i+1) dx = \sum_{i=0}^{k-2} \int (i+1) x^i dx$$

$$\Rightarrow \sum_{i=0}^{k-2} x^{i+1} = x \sum_{i=0}^{k-2} x^i$$

$$\text{mit } f(x) = (i+1)x^i \rightarrow F(x) = x^{i+1}$$

$$\text{Es gilt: } x^0 + x^1 + x^2 + \dots + x^k = \frac{x^{k+1} - 1}{x - 1}$$

Somit ergibt sich:

$$\int f(x) dx = x \sum_{i=0}^{k-2} x^i = x \frac{x^{k-1} - 1}{x - 1} = \frac{x^k - x}{x - 1}$$

Um zur geschlossenen Darstellung zurückzukommen, muss wieder abgeleitet werden:

$$f(x) = \left( \frac{x^k - x}{x - 1} \right)' = \frac{(kx^{k-1} - 1)(x - 1) - (x^k - x) * 1}{(x - 1)^2}$$

weil gilt:

$$\left( \frac{f}{g} \right)' = \frac{f' * g - f * g'}{g^2}$$

Wird  $x = \frac{1}{2}$  wieder eingesetzt, erhält man folgende Berechnung:

### 3 Die Implementierung

$$\Rightarrow f\left(\frac{1}{2}\right) = \frac{\left(\frac{1}{2}k^{k-1} - 1\right)\left(\frac{1}{2} - 1\right) - \left(\frac{1}{2} - \frac{1}{2}\right) * 1}{\left(\frac{1}{2}\right)^2}$$

$$\Rightarrow 4\left(-\frac{k}{2^k} + \frac{1}{2} - \frac{1}{2^k} + \frac{1}{2}\right) = 4\left(1 - \frac{k+1}{2^k}\right)$$

Wieder in die Formel von Seite 38 eingefügt ergibt sich Folgendes:

$$2^{k-2} * f\left(\frac{1}{2}\right) = 2^{k-2} * 4\left(1 - \frac{k+1}{2^k}\right)$$

$$\Rightarrow 2^{k-2} * 2^2 \left(\frac{2^k}{2^k} - \frac{k+1}{2^k}\right) = 2^k - k - 1 = n - k$$

Im Beispiel ist  $n = 15$  und entsprechend  $k = 4$ :  $n - k = 15 - 4 = 11$ , genau wie am praktischen Beispiel durchgezählt wurde. Pro Vorgang werden maximal zwei Vergleiche durchgeführt, somit ist bewiesen, dass die Anzahl der Vergleiche  $< 2 * n$  ist, da  $n - k < n$  gilt.

Jeden weitere Heap-Aufbau zur Suche nach dem nächsten kleinsten/größten Element benötigt danach höchstens  $2 * \log_2 n$  Vergleiche, da maximal über die gesamte Höhe mit jeweils zwei Kindelementen vergleichen und vertauscht werden muss.

#### 3.3.2 Die Codewortlänge: Baum und Heap im Vergleich

Weil erst alle Zeichen in den Speicher gelesen werden müssen und keine Länge festgelegt werden kann, bevor sie berechnet wurde, würde zur Berechnung der Codewortlänge bei Verwendung eines Baumes mit  $n$  Blättern und  $n - 1$  internen Knoten einiges an Speicher benötigt werden: jedes der  $n$  Blätter speichert das Zeichen, einen Wert und die Information, dass es ein Blatt ist =  $3 * 4$  Bytes. Jeder der  $n - 1$  internen Knoten speichert zwei Kinder und die Summe der Werte dieser Kinder =  $3 * 4$  Bytes. Somit ergeben  $24 * n$  Bytes für Knoteninformationen, Berechnungen und Eltern-Kind-Verhältnisse. Somit brauchte ein Alphabet von 1 Mio. Zeichen schnell 24 MByte Speicher. Zusätzlich würde das Durchlaufen des Baumes einen großen Speicheraufwand bedeuten, da nicht alle Daten im Cache vorgehalten werden können, sobald die Tiefe des Baums zu groß wird. So kann es passieren, dass ständig Daten gebraucht werden, die nicht (mehr) im Speicher sind und somit neu

## 3 Die Implementierung

geladen werden müssen.

Besser ist der Einsatz einer Heap-Daten-Struktur, welche lediglich 8 MB Speicher für dasselbe Alphabet benötigt, da nur  $2 * n * 4$  Byte (=  $8 * n$  Byte für  $n$  „Blätter“ und  $n$  „Zeiger“ auf diese Blätter) gespeichert werden. Zudem ist der Algorithmus äußerst effizient, was die Berechnungszeit betrifft.

### 3.3.3 Zeit zum Codieren unter Verwendung eines Heaps

In der 1. Phase wird das Array (Länge  $2 * n$ ) gefüllt und der Heap aufgebaut (über die ersten  $n$  Einträge). Die Zeit für den Aufbau ist linear, es müssen weniger als  $2 * n$  Schritte durchgeführt werden.

Während der 2. Phase wird zweimal ein Element im Heap „versickert“. Dieses „Versickern“ wird somit  $(2 * n)$ -mal durchgeführt. Die Länge des zu betrachtenden Arrays beträgt hierbei jeweils  $h$ , wobei zu Beginn  $h = n$  ist. Folglich werden für das „Versickern“ maximal  $2 * \log_2 h$  bzw.  $2 * \log_2 n$  Vergleiche benötigt. Über alle Aufrufe betrachtet benötigt diese Iteration  $k * n * \log_2 n$  Schritte ( $k \triangleq$  kleiner konstanter Wert).

In der dritten Phase werden die Elternzeiger gezählt. Im schnellsten Fall ist die Verteilung der Zeichen gleichmäßig und alle Codes sind annähernd  $\log_2 n$  Bits lang (= Höhe des dazugehörigen Baums). Daraus ergibt sich eine Durchlaufzeit von  $n * \log_2 n$  Schritten. Im „worst case“ hat das  $i$ -te Zeichen einen  $i$ -Bit-Code, somit wäre die Durchlaufzeit proportional zur Summe aller  $i \approx n^2/2$ , was die Zeit in Phase 1 und 2 überbieten würde. Die verbesserte Variante dagegen benötigt lediglich eine Durchlaufzeit proportional zur Anzahl der Zeichen.

### 3.3.4 Speicher beim Decodieren mit einem Baum

Jeder Decodier-Baum für  $n$  Zeichen benötigt  $n$  Blätter und besitzt genau  $n-1$  interne Knoten. Jedes Blatt speichert das entsprechende Zeichen (= 4 Bytes) und die Information, dass es ein Blatt ist (= 4 Bytes). Jeder Knoten speichert bis zu zwei Zeiger auf Knoten bzw. Blätter (=  $2 * 4$  Bytes). Somit werden etwa  $4 * 4$  Byte gespeichert, was bei einem Alphabet von 1 Mio. Zeichen 16 MByte Speicher alleine für den Baum ausmacht. Wesentlich weniger benötigt die verwendete Variante, die im Folgenden erläutert wird.

### 3.3.5 Speicher beim Decodieren ohne Baum

Als maximale Länge (*maxlength*) für ein Codewort kann 32 angenommen werden, da auf diese Weise eine recht große und meist ausreichende Anzahl von Codewörtern zur Verfügung steht. Zur Speicherung der Zeichen werden drei verschiedene Varianten kurz beschrieben:

#### 3.3.5.1 Zweidimensionales Array

Als Arrays werden in erster Linie *firstcode[]* und *symbol[][]* benötigt.

### 3 Die Implementierung

*firstcode[]* hat nach vorangegangener Annahme maximal die Länge  $maxlength + 1$  (weil *firstcode[0]* nicht verwendet wird). Wird ein zweidimensionales Array *symbol[][]* verwendet, kann dies die maximale Größe von  $(maxlength + 1) * \max(numl[] + 1)$  besitzen (somit maximal die Größe  $n + 1$  erreichen kann, falls alle Codewörter die gleiche Länge bekommen).

Für „ABRAXAS“ sähe dieses Array so aus:

```
symbol[ ][ ] = new String[4][4]
symbol[1][0] = A
symbol[3][0] = B
symbol[3][1] = R
symbol[3][2] = S
symbol[3][3] = X
```

Somit wäre der maximale Speicheraufwand bei  $(32 + 1) + (32 + 1) * (n + 1) \approx 33 + n * 33$  Wörter, er hängt also von  $n$  ab und vergrößert sich proportional.

#### 3.3.5.2 Eindimensionales Array – erste Variante

Im verwendeten Text<sup>3</sup> wird auf S. 41 von „drei kleinen Arrays“ gesprochen, welche jeweils die maximale Länge von  $maxlength (+ 1)$  erreichen können. Hierbei müssen *numl[]*, *firstcode[]* und *nextcode[]* gemeint sein, die jedoch gar nicht alle zur Decodierung benötigt werden. Lediglich die Daten aus *firstcode[]* fließen in die Berechnungen ein. Darüber hinaus wird das Array *symbol* eindimensional angelegt, wobei das  $l$ -te Element genau *numl[l]* (= Anzahl der Codewörter der Länge  $l$ ) Einträge besitzen soll. Theoretisch besäße so ein Array wiederum maximal die Länge der maximalen Codewortlänge, wobei jedoch jeder Eintrag aus mehreren bestehen könnte. Zugriffen würde beispielsweise auf das erste Codewort der Länge 1 mit *symbol[1,0]* und auf das dritte Codewort der Länge 3 mit *symbol[3,2]*.

Dieses Array sähe für „ABRAXAS“ in etwa so aus:

```
symbol[] = [/] [[A]] [/] [[B][R][S][X]]
```

Praktisch lässt sich ein solches Array mit Java nicht ohne weiteres umsetzen. Zum Einen ist es nur vorgesehen, mit Integerwerten auf ein Array zuzugreifen. Hier bestünde evtl. die Möglichkeit, die Zugriffsmethode zu überschreiben. Zum Anderen ergibt sich das Problem von mehreren Einträgen an einer Position im Array, also quasi ein Array im Array. Eine solche Umsetzung benötigte theoretisch  $(32 + 1) + (32 + 1) = 66$  Wörter, wobei jedoch un-

<sup>3</sup> [Ian H. Witten et al.: „Managing Gigabytes: Compressing and Indexing Documents and Images“]

## 3 Die Implementierung

berücksichtigt ist, dass im Array `symbol[]` ein Eintrag aus mehreren bestehen kann, es sich also quasi um Arrays im Array handelt.

### 3.3.5.3 Eindimensionales Array – zweite (verwendete) Variante

Eine weitere Möglichkeit wäre, die Daten der Reihe nach ins Array zu schreiben und in einem anderen Array die erste Position jeder Codewortlänge zu speichern. Diese Variante wurde bei der Implementierung umgesetzt.

Am Beispiel von „ABRAXAS“ sieht dies wie bereits erwähnt folgendermaßen aus:

```
symbol[] = [A] [B] [R] [S] [X]
start[] = [/] [0] [/] [1]
```

Das Array `symbol[]` besitzt die Länge  $n$ , da jedes Zeichen seinen eigenen Eintrag beansprucht. Das Array `start[]` benötigt die maximale Länge von `maxlength + 1`. Somit beansprucht diese Variante  $n + (32 + 1)$  Wörter.

Im Folgenden die genaue Übersicht der zur Implementierung benötigten Variablen und Arrays:

## 3 Die Implementierung

### 3.3.6 Benötigte Daten

Variable	Vergl. Zur Vorlage	Bedeutung	Länge
<code>int n</code>	$n$	Anzahl der Zeichen/Wörter	
<code>String[] word</code>	$i$	einzelnes Zeichen/Wort	$n$
<code>int[] count</code>	$c_i$	Auftrittshäufigkeit von $i$ (dem Zeichen/Wort)	$n$
<code>int[] huffmanArray</code>	Array $A$ of $2*n$ words	Array für den HeapSort	$2*n$
<code>int[] codelength</code>	$l_i$	Länge des Zeichens/Wortes $i$	$n$
<code>int maxlength</code>	$maxlength$	maximale Codewortlänge	$\leq 32$
<code>int minlength</code>		Minimale Codewortlänge	$\geq 1$
<code>int[] numl</code>	$numl[l]$	Anzahl der Codewörter der Länge $l$	$maxlength + 1$
<code>int[] firstcode</code>	$firstcode[l]$	Dezimalwert des ersten Codeworts der Länge $l$ bzw. Präfixe kleiner als bei längeren Codewörtern	$maxlength + 1$
<code>int[] nextcode</code>	$nextcode[l]$	Speicherung des nächsten Codeworts der Länge $l$ als Dezimalwert	$maxlength + 1$
<code>int[] codeword</code>	$codeword[i]$	Codewort für Symbol $i$	$n$
<code>int[] start</code>		speichert Startpunkt jeder Codewortlänge in $symbol[]$	$maxlength + 1$
<code>String[] symbol</code>	$symbol[]$	1-dimensionales Array $symbol[start[l] + position]$	$n$

### 3 Die Implementierung

Variable	Vergl. Zur Vorlage	Bedeutung
<code>String[] textFiles</code>	-	Wörter des Textes (ohne Satzzeichen) in einem Array
<code>TreeMap&lt;String, Integer&gt; wordCount</code>	-	wird zu Beginn zum Sortieren der Wörter benötigt - nach dem Auslesen wird die TreeMap nicht mehr benötigt
<code>StyledDocument outputDocument</code>	-	formatierte Ausgabe des codierten bzw. decodierten Textes
Zusätzlich einige Variablen zur Debug-Ausgabe bzw. zur Erstellung eines Wörterbuchs		

*Tabelle 8: Verwendete Variablen*

## 3 Die Implementierung

### 3.4 Die grafische Benutzeroberfläche

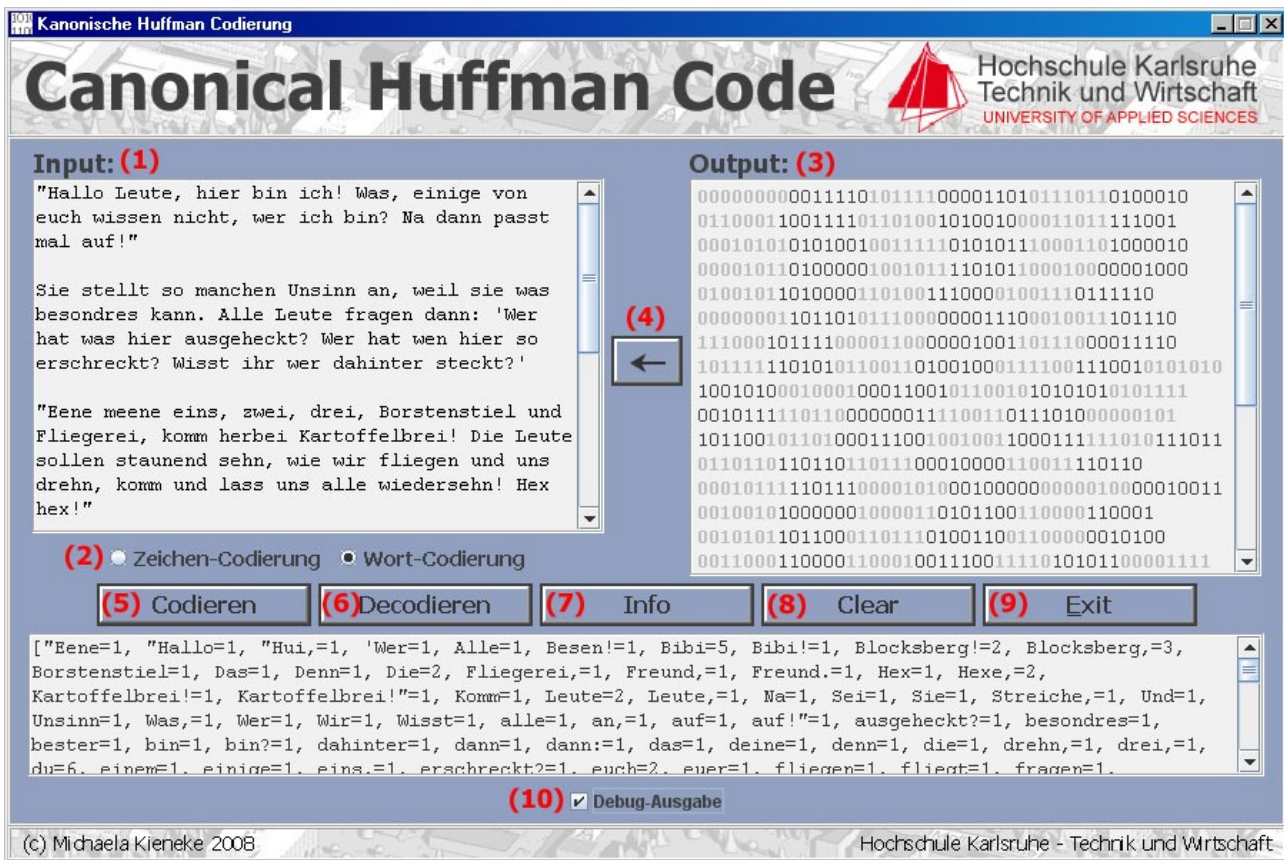


Abbildung 17: Die grafische Benutzeroberfläche

Die Benutzeroberfläche wurde mit Swing-Elementen erstellt. Die Programmierung der einzelnen Controls erfolgte „von Hand“, da entsprechende Tools bzw. Plugins (beispielsweise Visual Editor Project<sup>4</sup>) keine zufriedenstellenden Code erzeugten.

- (1) Texteingabe (3.4.2)
- (2) Zeichen-/Wortcodierung (3.4.3)
- (3) Textausgabe (3.4.4)
- (4) Kopieren der Textausgabe in die Texteingabe (3.4.2)
- (5) Codieren
- (6) Decodieren (3.4.5)
- (7) Wörterbuch (3.4.6)
- (8) Löschen (3.4.7)

4 THE ECLIPSE FOUNDATION: „Visual Editor Project“,  
URL: <http://www.eclipse.org/vep/WebContent/main.php> [Stand 15.04.08]

## 3 Die Implementierung

(9) Beenden (3.4.9)

(10) Debug-Ausgabe (3.4.8)

### 3.4.1 Style der GUI

Um das einheitliche Aussehen der Steuerelemente zu vereinfachen, wurden eigene Elemente mit dem Präfix „My“ entwickelt.

Der Hintergrund der Bedienelemente ist überwiegend auf „durchsichtig“ gesetzt, damit überall eine einheitliche Hintergrundfarbe erscheint, die des „MainContainers“. Der Hintergrund der Textflächen (*TextArea* und *TextPane*) ist auf hellblau gesetzt. Als Schriftart der GUI wird „Tahoma“ verwendet, die Texte in den Textflächen werden mit „Courier New“ dargestellt.

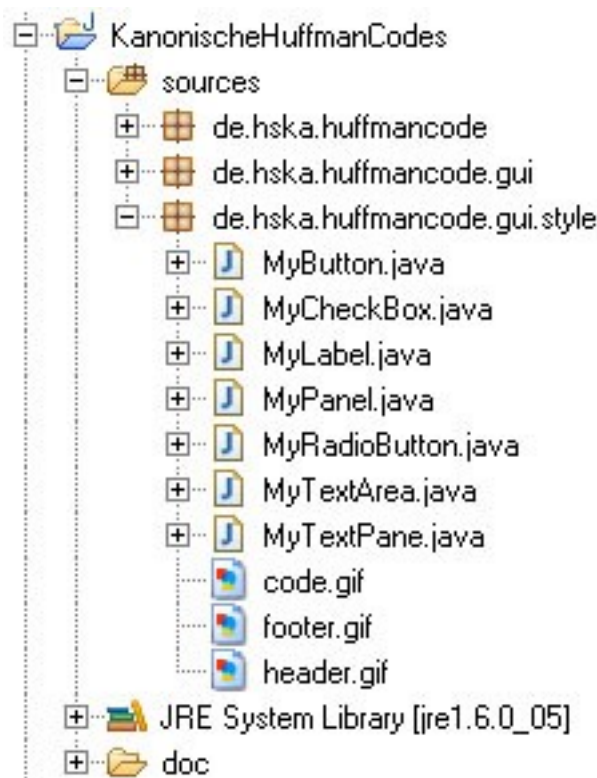


Abbildung 18: Eigene Styles für die GUI

### 3.4.2 Eingabe des Textes

Im linken Textfeld wird der zu codierende bzw. decodierende Text eingegeben. Zu Beginn geschieht dies über die Tastatur. Soll bereits codierter Text aus dem Output-Textfeld wieder decodiert werden, kann dieser auch einfach über den Pfeil-Button in das Input-Textfeld kopiert werden.

## 3 Die Implementierung

### 3.4.3 Zeichen- oder Wortcodierung

Zur Auswahl bei der Codierung stehen Zeichen- oder Wortcodierung.

Bei der Zeichencodierung wird jedes Zeichen berücksichtigt, auch Zahlen, Satz- und Sonderzeichen.

Bei der Wortcodierung werden komplette Wörter codiert. Satzzeichen werden in Kombination mit dem entsprechenden Wort codiert, an dem sie stehen. Somit wird ebenfalls der gesamte Text inklusive Satzzeichen wieder decodiert.

### 3.4.4 Ausgabe des codierten Textes

Im rechten Textfeld wird der codierte bzw. wieder decodierte Text ausgegeben. Tritt ein Fehler bei der Codierung oder Decodierung auf, wird diese Meldung ebenfalls hier ausgegeben.

Beim codierten Text werden zur besseren Übersicht einzelne Zeichen durch abwechselnde Farbgebung unterschiedlich hervorgehoben.

### 3.4.5 Decodieren

Zu decodierender Text kann über die Tastatur eingegeben oder über den Pfeil aus dem Output-Textfeld kopiert werden.

Bei der Decodierung wird das zuvor gespeicherte „Wörterbuch“ verwendet. Passt der zu codierende Text nicht zum Wörterbuch, wird eine entsprechende Fehlermeldung ausgegeben. Die geschieht ebenso, wenn noch gar kein „Wörterbuch“ vorliegt, also noch keine Codierung oder eine Löschung der gesamten Daten durchgeführt wurde.

### 3.4.6 Wörterbuch

Über den „Info“-Button gelangt man zum „Wörterbuch“. Zu Beginn ist dies leer, nach der ersten Codierung beinhaltet es die verwendeten Zeichen und ihre entsprechenden Codewörter.

## 3 Die Implementierung

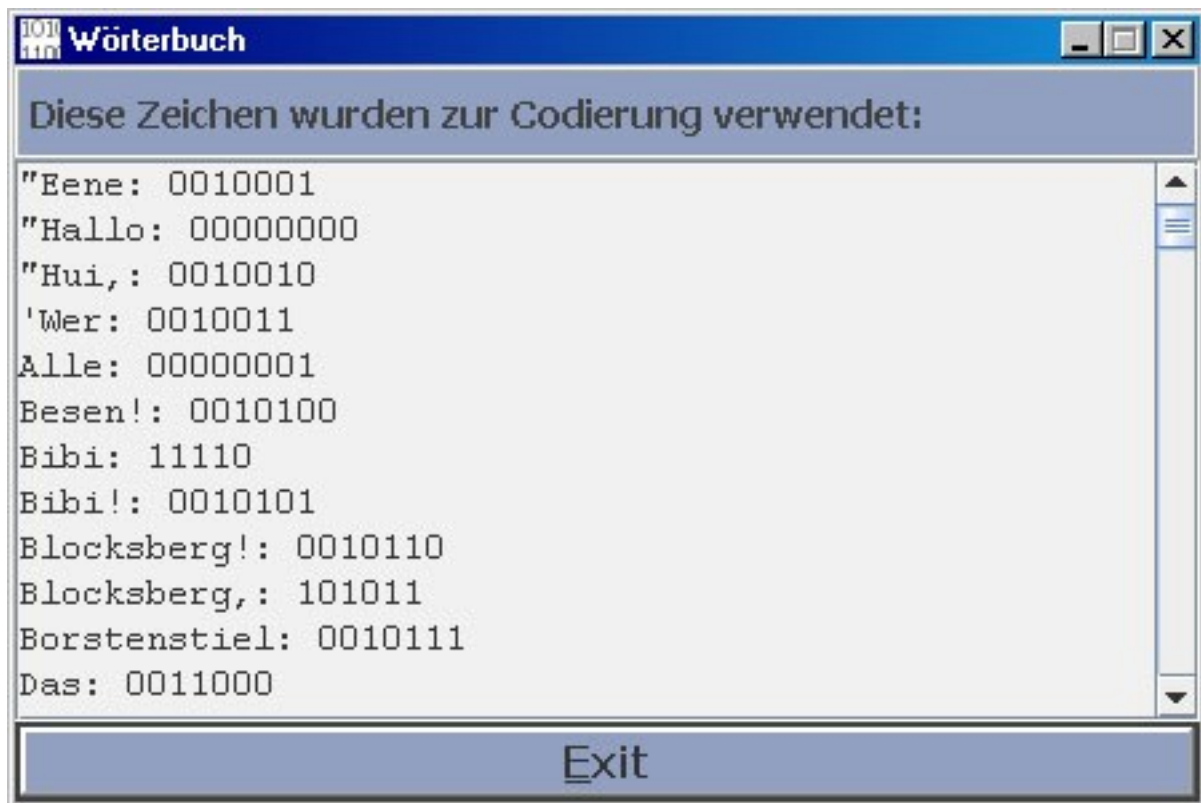


Abbildung 19: Das Wörterbuch

### 3.4.7 Löschen

Mit dem „Löschen“-Button werden alle bisher gespeicherten Ergebnisse gelöscht. Vorsicht: auch das „Wörterbuch“ ist danach leer!

### 3.4.8 Debug-Ausgabe

Bei der Implementierung wurde der Programmierfortschritt in der Konsole kontinuierlich überprüft. Bei ausgewählter CheckBox vor der „Debug-Ausgabe“ werden die entsprechenden Berechnungen des Codes in der unteren Textausgabe gezeigt.

### 3.4.9 Beenden

Die Anwendung kann durch Doppelklick auf die Oberfläche oder über den „Exit“-Button beendet werden.

### 3.4.10 Fehlerausgabe

Es wurde versucht, möglicherweise auftretende Fehler abzufangen. Die entsprechenden Ausgaben erscheinen wie schon erwähnt im Output-Textfeld.



## 4 Schlusswort

### 4 Schlusswort

Natürlich ist bekannt, dass Arrays stets mit `array[0]` beginnen. Bei der Implementierung wurde sich lediglich an den Aufbau von `array[1]` bis `array[n]` gehalten, da dies in der verwendeten Vorlage ebenfalls durchgehend so betrachtet wurde.

Herzlichen Dank an Prof. Dr. Körner für dieses sehr interessante Thema und seine nette Betreuung. Ich habe sehr viel gelernt und werde darauf weiterhin aufbauen.

## 5 Quellenverzeichnis

### 5 Quellenverzeichnis

**[Ian H. Witten et al.: „Managing Gigabytes: Compressing and Indexing Documents and Images“]**

Ian H. Witten et al., „Managing Gigabytes: Compressing and Indexing Documents and Images“, Morgan Kaufmann Publishers, Kapitel 2.3 „Huffman coding“, S. 30 – 51

**[Prof. Dr. rer. nat. Dirk Hoffmann: „Technische Grundlagen Multimedia“]**

Prof. Dr. rer. nat. Dirk Hoffman, „Technische Grundlagen Multimedia“, Foliensatz „MM\_Kompression“,

URL: [http://www.dirkwhoffmann.de/Lehre/slides/MM\\_Kompression.pdf](http://www.dirkwhoffmann.de/Lehre/slides/MM_Kompression.pdf) [Stand 09.04.08]

**[Wikimedia Foundation Inc.: „Heap Sort“]**

Wikimedia Foundation Inc., „Heap Sort“,

URL: [http://de.wikipedia.org/wiki/Heap\\_Sort](http://de.wikipedia.org/wiki/Heap_Sort) [Stand 23.03.08]

**[Wikimedia Foundation Inc.: „Morsecode“]**

Wikimedia Foundation Inc.: „Morsecode“,

URL: <http://de.wikipedia.org/wiki/Morsecode> [Stand 09.04.08]

**[Entwicklungsumgebung]**

Eclipse SDK

Version: 3.3.0

Buid id: I20070625-1500

Download: <http://www.eclipse.org/>

Erste Seite:

„Projektarbeit Kanonische Huffman Codes SS2008 Michaela Kieneke“