Fehlerkorrigierende und komprimierende Codes mit Snap!

Fritz Hasselhorn

20. Dezember 2022

Inhaltsverzeichnis

1 Fehlerkorrigierende und komprimierende Codes

1.1 Fehlerkorrigierende Codes

Eine Codierung ist eine mathematische Vorschrift, die jedem Zeichen einer Urbildmenge ein Zeichen bzw. eine Zeichenfolge einer Bildmenge zuordnet. Weil Computer binär arbeiten, ist für jede Verarbeitung von Zahlen, Buchstaben oder Bildern eine Codierung notwendig.

Bereits behandelte Codes sind die Dualzahlen, der ASCII-Code für Buchstaben bzw. sein Nachfolger, der Unicode und der RGB-Code für Farbwerte. Ausführungen dazu finden sich im Band `Einführung in die Programmierung mit Snap!` in den Kapitel 3 (Grafik), 5 (Dualzahlen) und 6 (Zeichenketten).

Fehlererkennende und fehlerkorrigierende Codes sind Datencodierungen, die zusätzlich zu den codierten Daten noch Informationen enthalten, um Datenfehler zu erkennen oder zu beheben. Beispiele für fehlererkennende Codes im Alltag sind z.B.

- die IBAN (*International Bank Account Number* bzw. Internationale Bankkontonummer), die zwei Prüfzimmer enthält. Banken lehnen eine Online-Überweisung ab, wenn die IBAN fehlerhaft ist,
- die ISBN (Internationale Standardbuchnummer), die Bibliotheken und Buchgeschäfte benutzen, um Bücher eindeutig zu identifizieren,
- die EAN (*European Article Number*, die im Einzelhandel zur Identitifzierung von Waren benutzt wird,
- QR-Codes.

Das **Prinzip der digitalen Datenübertragung** funktioniert wie folgt: Der Sender Person A verfasst eine Nachricht und codiert diese. Es entsteht letztlich eine Folge von Bits, also Nullen und Einsen. Diese Bits werden zum Empfänger übertragen. Dabei treten durch atmosphärische oder technische Störungen Invertierungen auf, d.h. einzelne Bits werden falsch übertragen: Statt einer 0 wird eine 1 empfangen, statt einer 1 eine 0. Der Empfänger erhält die eventuell fehlerhafte Nachricht. Er muss sie decodieren und dabei möglichste die Fehler korrigieren, um die codierte Nachricht zu erhalten, d.h. im Idealfall die ursprüngliche Nachricht.

1.2 Hamming-Abstand

Wir vereinbaren zunächst einige Begriffe:

Bit: Das Wort 'Bit' ist eine Wortkreuzung aus 'binary' und 'digit'. Es ist die kleinste Einheit eines Computers und kann nur die Werte '0' und '1' annehmen.

Bitfolge, Bitsequenz: Eine Bitfolge ist nichts anderes als aneinandergereihte Bits, z.B. 0100.

Code, Codewörter: Ein **Code** entspricht einer Menge von gleichlangen[1] Bitsequenzen. Diejenigen Bitsequenzen, welche im Code vorkommen, nennen wir **Codewörter**. Ganz allgemein kann man sagen: Codewörter sind diejenigen Bitfolgen, die vom Sender verschickt werden.

Codieren: Unter Codieren verstehen wir die Übersetzung einer Nachricht in eine Bitsequenz – sprich in ein Codewort.

Decodieren: Decodieren bedeutet: „Zurückübersetzen“ einer Bitsequenz in eine Nachricht. Bei fehlerkorrigierenden Codes kommt noch ein Schritt hinzu: Umwandlung des erhaltenen Wortes in ein Codewort (nach

[1] In Kapitel 9.8 werden wir ein Beispiel für einen Code mit Codewörtern unterschiedlicher Länge kennen lernen.

festgelegten Regeln), das dann weiter in die Nachricht übersetzt wird.

Invertieren: Dieser Begriff bezieht sich auf Bits. Ein Bit invertieren bedeutet: Seinen Wert ändern – sprich: Aus '0' wird '1' und aus '1' wird '0'.

Paritätsbit: Das Paritätsbit einer Folge von Bits dient als Ergänzungsbit, um die Anzahl der mit 1 belegten Bits (inklusive Paritätsbit) der Folge als gerade (even parity) oder ungerade (odd parity) zu ergänzen. Beispiel: Bei gerader Parität würde die Folge 101 durch eine 0 als Paritätsbit ergänzt, die Folge 100 durch eine 1, weil sowohl 1010 als auch 1001 eine gerade Anzahl von Einsen enthalten.

Wie viele Fehler kann ein gegebener Code erkennen bzw. korrigieren?

Betrachten wir dazu ein Beispiel. C = {00011, 00100, 11010} sei ein Code mit drei Codewörtern, die jeweils eine Bedeutung haben, z.B. grün, gelb, rot. Natürlich könnte man drei Werte auch mit nur 2 Bits kodieren, aber für die sichere Datenübertragung sind längere Codewörter von Vorteil. Dekodiert wird jeweils zu dem Codewort, welches sich von der Nachricht an möglichst wenige Stellen unterscheidet.

C = {00011, 00100, 11010}

Empfangen wird 00111.

00111 unterscheidet sich von 00011 an einer Stelle.

00111 unterscheidet sich von 00100 an zwei Stellen.

00111 unterscheidet sich von 11010 an drei Stellen.

00111 wird dekodiert zu 00011.

Der **Hamming-Abstand** zweier (gleichlanger) Codewörter ist die Anzahl der Stellen, an denen sich die Codewörter unterscheiden.

Der **Hamming-Abstand eines Codes** ist das Minimum aller Hamming-Abstände zwischen Wörtern innerhalb des Codes.

Für den Hamming-Abstand verwenden wir einen Reporter, dem als Parameter die beiden Codewörter übergeben werden. Die Variable für das Ergebnis, `result`, wird auf Null gesetzt. Dann gehen wir die Wörter mit einer Zählschleife buchstabenweise durch. An allen Stellen, wo die Buchstaben nicht übereinstimmen, erhöhen wir `result` um Eins. Abschließend geben wir `result` aus.

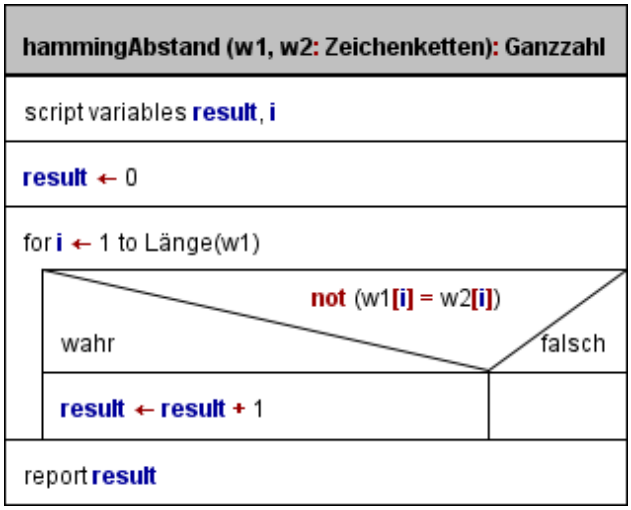

Abbildung 1.1: Berechnung des Hamming-Abstandes zweier Codwörter

Welcher Zusammenhang besteht zwischen dem Hamming-Abstand eines Codes und der Anzahl der Fehler, die bei diesem Code korrigiert werden können? Wenn wir den gesamten Übertragungsvorgang betrachten, dann treten zunächst auf dem Weg vom Sender zum Empfänger k Fehler auf. Wenn der Empfänger das empfangene Wort zum nächsten Codewort korrigiert, dann werden weitere l Bits geändert. Insgesamt werden beim Übertragen und beim Decodieren also maximal $k + l$ Bits verändert.

Wir wissen: $l \leq k$, d.h. die Anzahl der Bits, die bei der Decodierung verändert werden, beträgt höchstens k. Dann man kann durch Invertierung der k „Fehlerbits“ wieder zum ursprünglichen Codewort zurückkehren. Somit ist das nächste Codewort höchstens k Bits entfernt.

Sei m der Hamming-Abstand des Codes. Gilt nun $2k > m$, dann können wir sagen:

$$k + l \leq k + l = 2k < m$$

Somit wird sicher richtig decodiert, denn zn von der ursprünglichen Nachreicht aus zu einem anderen Codewort zu kommen, müsste man mindestens m Bits invertieren, also so viel Bits, wie der Hamming-Abstand des Codes beträgt. Der Abstand zwischen der ursprünglichen Nachricht und der Anzahl der Bits, die von der ursprünglichen Nachricht bis zur Decodierung verändert wurden, ist jedoch kürzer als m.

Also kommt bei der Decodierung wieder das richtige heraus, wenn $2k < m$, also wenn das Doppelte der Anzahl der Bits, die verändert wurden, kleiner ist als der Hamming-Abstand des Codes.

Ist andererseits $2k \geq m$, ist also das Doppelte der Anzahl der veränderten Bits größer oder gleich dem Hamming-Abstand des Codes, dann ist eine falsche Decodierung durchaus möglich.

Zusammengefasst kann man feststellen: Ein Code mit einer Minimaldistanz von m kann k Fehler korrigieren, wobei $k < \frac{m}{2}$.

Bei n Codewörtern müssen $(n-1)+(n-2)+...+2+1$ Berechnungen des Hamming-Abstandes zweier Codewörter durchgeführt werden. Das sind nach der Formel von Gauss $n \cdot \frac{n}{2}$ Berechnungen. Es liegt also eine quadratische Komplexität vor.

1.3 Lineare Codes

Um Codes mit vorgegebenem Abstand einfach konstruieren zu können, führen wir eine Art „Addition“ zweier Bitfolgen ein. Dabei verwenden wir die bekannte XOR-Verknüpfung:

$$0 \otimes 0 = 0 \qquad 0 \otimes 1 = 1$$
$$1 \otimes 0 = 1 \qquad 1 \otimes 1 = 0$$

Beispiel: $1001 \otimes 1101 = 0100$. Achtung! Die hier verwendete XOR-Verknüpfung darf nicht mit der dualen Addition verwechselt werden, die rechnet nämlich mit Übertrag.

Definition: **Ein Code heißt linear, falls die XOR-Verknüpfung von zwei Codewörtern selbst wieder ein Codewort ist**.

Beispiele: Der Code $C = \{000, 011, 101, 110\}$ ist linear, der Code $D = \{000, 001, 011, 100\}$ hingegen nicht (beispielsweise ist $100 \otimes 011 = 111$ kein Codewort.

Wenn wir lineare Codes untersuchen, machen wir eine Reihe von Beobachtungen:

- **Beobachtung 1:** Das Nullwort, d.h. die Bitsequenz 00...0 kommt in jedem Code vor. Das liegt daran, dass die XOR-Verknüpfung jedes beliebigen Codewortes mit sich selbst 00...0 ergibt.
- **Beobachtung 2:** Die Verknüpfung von zwei Codewörtern, die genau den Summen der jeweiligen Bits entspricht, hat genau an den Stellen eine Eins, an denen sich die beiden Summanden unterscheiden.

Das Hamming-Gewicht einer Zeichenkette ist definiert als die Anzahl der vom Nullzeichen des verwendeten Alphabets verschiedenen Zeichen. Hierbei handelt es sich zugleich um den Hamming-Abstand zum Nullvektor, also der Bitsequenz 00...0.

Das Hamming-Minimalgewicht eines Codes ist das kleinstmögliche Gewicht, das von einem seiner Codewörter (ausgenommen 00...0) besessen wird.

Wir beweisen den folgenden Satz:

Der Minimalabstand eines **linearen Codes** ist gleich seinem Minimalgewicht.

Zur Erinnerung: Der Minimalabstand ist der kürzeste Abstand zwischen zwei Codewörter.

Wir werden diese Behauptung in zwei Schritten zeigen: Zuerst beweisen wir, dass der **Minimalabstand** gleichzeitig ein Gewicht eines Codewortes ist. Danach zeigen wir, dass das **Minimalgewicht** gleichzeitig der Abstand zwischen zwei bestimmten Codewörtern ist.

Warum beweist dies, das der Minimalabstand gleich dem Minimalgewicht ist? Bezeichnen wir den Minimalabstand mit d_{min} und das Minimalgewicht mit w_{min}.

Da d_{min} ein Gewicht ist, gilt

$$w_{min} \leq d_{min}$$

(das Minimalgewicht ist kleiner oder gleich d_{min}). Und da w_{min} ein Abstand ist, gilt

$$d_{min} \leq w_{min}$$

(der Minimalabstand ist kleiner oder gleich w_{min}). Also gilt

$$w_{min} = d_{min}$$

.

1. Schritt: Das Gewicht eines Codeworts ist tatsächlich sein Abstand vom Nullwort. Somit entspricht das Minimalgewicht dem Abstand vom Nullwort zum Codewort mit dem kleinsten Gewicht. Also ist das Minimalgewicht ein Abstand zwischen zwei bestimmten Codewörtern.

2. Schritt: Der Minimalabstand bezeichnet ja bekanntlich den kleinstmöglichen Abstand zwischen zwei Codewörtern. Weiter oben haben wir schon gesehen, dass der Abstand von zwei Codewörtern genau dem Gewicht ihrer Summe entspricht. Da wir es mit linearen Codes zu tun haben, ist diese Summe wieder ein Codewort. Jeder Abstand - und damit auch der Minimalabstand - entspricht also dem Gewicht eines bestimmten Codewortes.

Im letzten Kapitel haben wir gesehen:

Ein Code mit Minimaldistanz m kann k Fehler korrigieren, wobei $k < \frac{m}{2}$.

Aus diesem Ergebnis und dem Satz über Minimalabstand und Minimalgewicht folgt:

Ein linearer Code mit Minimalgewicht w kann k Fehler korrigieren, wobei $k < \frac{w}{2}$.

1.4 n-Bit-Repetitionscode

Die einfachste Möglichkeit zur Fehlererkennung und -korrektur besteht darin, jedes Zeichen wiederholt zu senden. So wird im 3-Bit-Repetitionscode jedes Bit drei Mal wiederholt. Bei der Decodierung entscheidet die Mehrzahl der empfangenen Bits, nach welchem Bit decodiert wird.

Beispiel: Die Nachricht „1011“ soll übertragen werden. Dann sendet der Sender die Folge „111 000 111 111“. Durch zwei Fehler wird aber die Folge „110 000 011 111“ empfangen. Der Empfänger erkennt, dass zwei Dreiergruppen Fehler aufweisen und korrigiert diese entsprechend der Mehrheitsregel beide nach 1. Er erhält also „1011“, die ursprüngliche Nachricht.

Wir haben gesehen, dass dieser Code einen Fehler pro 3-Bit-Block korrigieren kann. Zwei Fehler pro 3-Bit-Block werden zwar als „fehlerhaft“ erkannt, werden aber nicht mehr richtig decodiert. Beispiel: Der Sender möchte das Bit „1“ übertragen. Beim Senden passieren jedoch zwei Fehler und aus „111“ wird „001“. Der Empfänger wird dann zu „0“ decodieren.

Wie sieht es aus, wenn jedes Bit 5 Mal wiederholt wird? In einer Fünfergruppe ist die Mehrheit Drei. Wenn also 3 Bit richtig übertragen wurden (und 2 falsch), dann kann die Nachricht richtig decodiert werden. Der 5-Bit-Repetitionscode kann also zwei Fehler korrigieren. Er erkennt bis zu vier Fehlern (d.h. er erkennt, dass Fehler vorliegen, kann aber natürlich nicht genau sagen, wie viele Fehler es sind), weil bei vier Fehlern keine unverfälschte Fünfergruppe entstehen kann.

Betrachten wir noch kurz den allgemeinen Fall: Wir senden jedes Bit n mal (wobei n eine ungerade Zahl ist), oder anders gesagt, für jedes Bit verschicken wir einen Block von n (gleichen) Bits. Der **n-Bit-Repetitionscode** (= der Code, der jedes Bit n mal sendet) kann also pro gesendeten n-Bit-Block $\frac{n-1}{2}$ Fehler korrigieren. Dabei ist zu beachten, dass n eine ungerade Zahl ist. Bei einer geraden Anzahl von Bits ist die Hürde für einen Mehrheitsentscheid höher, weil auch Pattsituationen auftreten können.

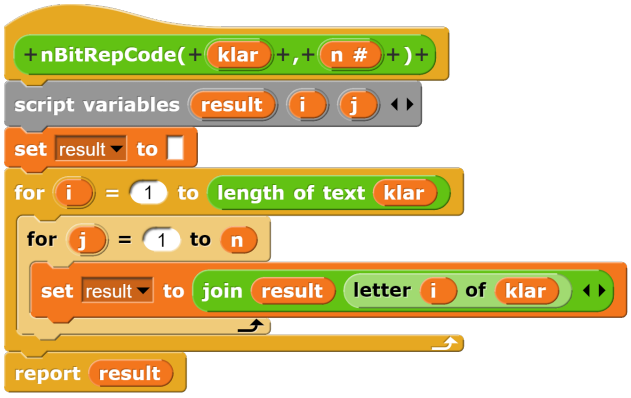

Abbildung 1.2: Codierung n-Bit-Repetitionscode

1.5 (7,4)-Hamming-Code

Der (7,4)-Hamming-Code geht etwas sparsamer mit den Bits um. Statt für jedes Bit 3 oder mehr Sendebits aufzuwenden, fassen wir nun 4 aufeinanderfolgende Bits zu einem Codeblock zusammen und fügen 3 Paritätsbits hinzu. Anschaulich sieht das so aus: Wir „übersetzen“ eine 4-Bit-Sequenz in eine 7-Bit-Sequenz.

Es gibt zwei verschiedene Arten der Darstellung des (7,4)-Hamming-Codes. In Abituraufgabenstellungen werden beide Arten verwendet. Die erste Darstellung ist die 3-Kreis-Figur, wie in der Abbildung 9.1 gezeigt. Dabei stehen die Zahlen 1 bis 4 für die vier Datenbits, die Zahlen 5 bis 7 für die drei Paritätsbits. Die drei

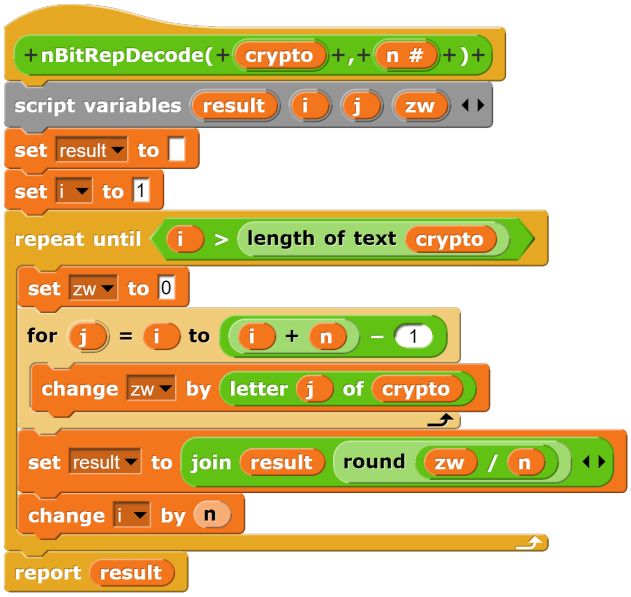

Abbildung 1.3: Decodieren n-Bit-Repetitionscode

Paritätsbits werden so ergänzt, dass sich für jeden der drei Kreise eine gerade Parität ergibt.

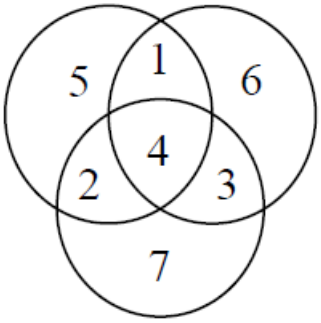

Abbildung 1.4: 3-Kreis-Schema für (7,4)-Hamming-Code

Wir betrachten ein Beispiel. Wir wollen die Bitfolge 1101 mit dem (7,4)-Hamming-Code kodieren. Dazu tragen wir die vier Zahlen der Reihenfolge nach in eine leeres 3-KreisSchema ein. Die erste Eins kommt in das mittlere Feld oben, die zweite schräg links darunter, die Null daneben und die letzte Eins in das Mittelfeld der Figur.

Jetzt werden die drei Paritätsbit so ergänzt, dass in jedem Kreis eine gerade Anzahl von Einsen vorhanden ist. Der Kreis links oben enthält bereits drei Einsen, also wird in das freie Feld eine 1 gesetzt. Der Kreis rechts oben enthält bereits zwei Einsen, also wird in das freie Feld eine 0 gesetzt. Der Kreis unten enthält ebenfalls zwei Einsen, also wird in das freie Feld unten ebenfalls eine 0 gesetzt. Wenn wir die Zahlen in der oben angegebenen Reihenfolge auslesen, erhalten wir als Codierung für „1101“ die Sequenz „1101 100“.

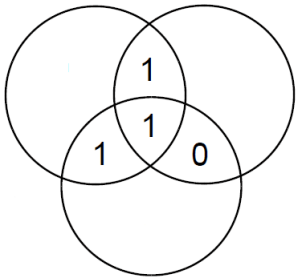

Abbildung 1.5: 3-Kreis-Beispiel

In Klausuren sollte man unbedingt die 3-Kreis-Figur aufzeichnen, die man benutzt. Leider kann es passieren, dass man unter Prüfungsstress die Reihenfolge der Nummerierung durcheinander bringt. Damit sind dann alle Codeworte falsch. Wenn man dann die Reihenfolge angibt, die man tatsächlich benutzt, hat man wenigstens die Chance auf einen Teil der Punkte wegen Folgefehlern.

Das Decodieren funktioniert entsprechend. Nehmen wir an, wir wollen die Nachricht „1001“ senden. Wir tragen die Bitfolge in ein 3-Kreis-Schema ein und ergänzen die Paritätsbits. Im ersten Kreis wird das Paritätsbit 0 ergänzt, im zweiten ebenso und im dritten das Paritätsbit 1. Damit erhalten wir die Bitfolge „1001 001“.

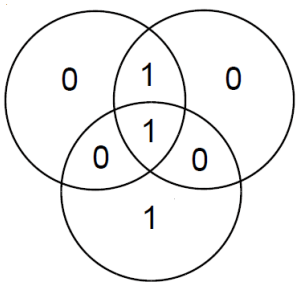

Abbildung 1.6: Nachricht in Beispiel 2

Bei der Übertragung wird das erste Bit invertiert, so dass wir „0001 001“erhalten. Wir tragen die emfangene Nachricht in das 3-Kreis-Schema ein und erhalten:

Wir kontrollieren die Paritäten und stellen fest, dass der obere linke und der obere rechte Kreis eine ungerade Parität aufweisen. Also muss das Bit invertiert werden, das in beiden

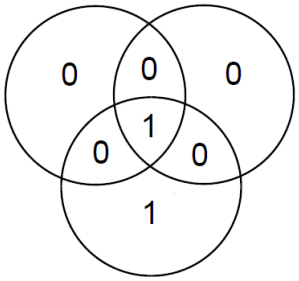

Abbildung 1.7: Emfangene Nachricht in Beispiel 2

Kreisen auftaucht. Damit erhalten wir die ursprüngliche Nachricht „1001“.

Sobald der Empfänger einen Block erhält, schreibt er die 7 Bits in die entsprechenden Felder des Diagramms, wobei das erste Bit ins Feld 1 und das letzte ins Feld 7 kommt. Dann prüft er, ob die Summe aller Zahlen in jedem Kreis gerade ist. Falls ja, nimmt er an, dass kein Fehler passiert ist, und decodiert den Block zu seinen ersten 4 Bits, d.h. er streicht die Prüferbits. Falls nein, nimmt er – optimistischer Weise – an, dass nur ein Fehler passiert ist. Dann sucht er dasjenige Bit, das er ändern muss, um in allen Kreisen ein gerade Summe zu erhalten. Mit andern Worten: Der Empfänger sucht dasjenige „gültige“ Diagramm, welches sich vom erhaltenen Diagramm an genau einer Stelle unterscheidet. Gültig bedeutet dabei, dass die Summe aller Zahlen in jedem Kreis gerade ist.

Das 3-Kreis-Schema ist gut für den (7,4)-Hamming-Code geeignet, es lässt sich aber nicht auf beliebige Hamming-Codes übertragen. Zum Beispiel lässt sich der (15,11)-Hamming-Code mit 15 Datenbits und 11 Nachrichtenbits nicht durch das Schema darstellen. Deshalb haben die Informatiker ein zweites, verallgemeinerungsfähiges Schema entwickelt.

Das zweite Schema geht davon aus, dass die Paritätsbits grundsätzlich an den Stellen der Zweierpotenzen stehen, also an Stelle $2^0 = 1$, $2^1 = 2$ und $2^2 = 4$. Das 4. Paritätsbit, über das nächste Hamming-Code verfügt, stände also an Stelle $2^3 = 8$. Die Datenbits stehen dann in den freien Stellen.

Das erste Paritätsbit steht an Stelle 1. Zu seiner Kontrollgruppe gehören alle Einergruppen mit Abstand 1, also die Bits, die an der 3., 5. und 7. Stelle stehen.

Das zweite Paritätsbit steht an Stelle 2. Zu seiner Kontrollgruppe gehören alle Zweiergruppen mit Abstand 2. Die erste Zweiergruppe besteht aus dem Paritätsbit und seinem rechten Nachbarn, dem 3. Bit. Dann folgen zwei Bits Pause, also das 4. und 5. Bit. Die nächste Zweiergruppe bilden das 6. und das 7. Bit.

Das dritte Paritätsbit steht an Stelle 4. Zu einer Kontrollgruppe gehört die Vierergruppe, beginnend mit dem 4. Element bis zum 7. Element. Die Verteilung der Paritäts- und der Datenbits sowie die Aufteilung der Kontrollgruppen ergeben sich aus der Tabelle.

c_1	c_2	c_3	c_4	c_5	c_6	c_7
p_1	p_2	d_1	p_3	d_2	d_3	d_4
p_1		d_1		d_2		d_4
	p_2	d_1			d_3	d_4
			p_3	d_2	d_3	d_4

Die beiden Darstellungsformen lassen sich ineinander umrechnen. Die vier Datenbits aus dem 3-Kreis-Schema nehmen die Plätze d_1 bis d_4 in der Tabelle ein, also die Plätze 3, 5, 6, 7. Das erste Paritätsbit steht an Stelle 1 der Tabelle, das zweite an Stelle 2 und das dritte an Stelle 4.

Für die Codierung eines Textes mit dem (7,4)-Hamming-Code gibt es zwei verschiedene Methoden. Für die erste sind verschiedene vorbereitende Schritte notwendig. Die Buchstaben müssen in ihren Unicode umgewandelt werden. Das deutsche Alphabet einschließlich der Umlaute liegt beim erweiterten ASCII-Code bzw. beim Unicode im Bereich von 33 (!) bis 252 (ü). Das sind maximal 8 Dualstellen. Jede Unicode-Zahl muss also in zwei Vier-Bit-Sequenzen aufgeteilt werden. Der Dezimalwert der letzten vier Dualstellen ergibt sich als Rest bei der Division durch 16: `set hinten to unicode(letter) mod 16`. Für die ersten vier Dualstellen verwendet man das Ergebnis der Division durch 16, wobei man die Nachkommastellen, d.h. den Rest, löscht: `set vorn to floor of (unicode(letter))/16`. ein Jede Vier-Bit-Sequenz kann dann kodiert werden. Dazu werden die drei Paritätsbits berechnet und angefügt werden. Der dargestellte Block geht davon aus, dass die Dezimalwerte bereits

in eine vierstellige Dualzahl umgerechnet worden sind:

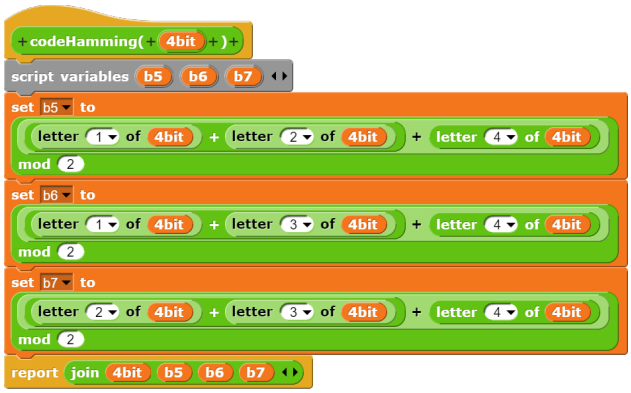

Abbildung 1.8: (7,4)-Hamming-Code Kodierung

Für die zweite Methode betrachtet man zunächst alle möglichen Fälle. Es gibt insgesamt 16 richtige Codeworte bei der (7,4)-Hamming-Codierung. Um diese Codewort zu finden, berechnen wir zu den Dualzahlen von 0000 bis 1111 die jeweiligen Paritätsbits:

4 Bit	1+2+4	1+3+4	2+3+4
0000	0	0	0
0001	1	1	1
0010	0	1	1
0011	1	0	0
0100	1	0	1
0101	0	1	0
0110	1	1	0
0111	0	0	1
1000	1	1	0
1001	0	0	1
1010	1	0	1
1011	0	1	0
1100	0	1	1
1101	1	0	0
1110	0	0	0
1111	1	1	1

Damit haben wir die 16 möglichen Kombinationen. Wir benötigen also nur das dezimale Äquivalent zu den Hexadezimalzahlen, einmal den Rest bei Division durch 16 und einmal den Quotienten aus dem Unicode und 16 (ohne die Nachkommastellen, deshalb `floor of`. Weil die einstelligen Hexedezimalzahlen von 0 bis 15 laufen, wir in der Liste aber die Indexzahlen von 1 bis 16 verwenden, müssen wir 1 hinzuaddieren. Damit vereinfacht sich die Umwandlung in den (7,4)-Hamming-Code:

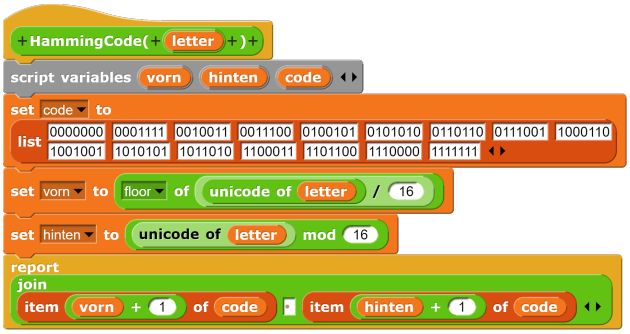

Abbildung 1.9: (7,4)-Hamming-Code Kodierung

Um Fehlerkorrektur und Decodierung vom Computer durchführen zu lassen, muss man alle möglichen Fälle analysieren. Die Tabelle zeigt die acht möglichen Kombinationen der Parität der 3 Kreise. Sind alle drei Paritäten gerade, muss kein Bit invertiert werden. Sind alle drei Paritäten ungerade, dann muss das Bit in der Schnittmenge invertiert werden, also Bit 4.

links o.	rechts o.	unten	invertiere
gerade	gerade	gerade	keins
gerade	gerade	ungerade	7
gerade	ungerade	gerade	6
gerade	ungerade	ungerade	3
ungerade	gerade	gerade	5
ungerade	gerade	ungerade	2
ungerade	ungerade	gerade	1
ungerade	ungerade	ungerade	4

Für die Decodierung benötigen wir einen Hilfsblock, der ein vorgegebenes Bit invertiert:

Damit können wir die Decodierung durchführen: Wir bestimmen die Zeile aus der Tabelle, die wir anwenden müssen, indem wir die Paritäten für die drei Kreise berechnen. Dann multiplizieren wir das Ergebnis des linken oberen Kreises mit 4 und das Ergebnis des rechten oberen Kreises mit 2 und addieren das Ergebnis des unteren Kreises. Damit haben wir die Dualzahl aus den drei Kreisen in eine Dezimalzahl umgewandelt. Aus der

Abbildung 1.10: Invertiere Bit

Tabelle oben ergibt sich dann, welches Bit jeweils invertiert werden muss.

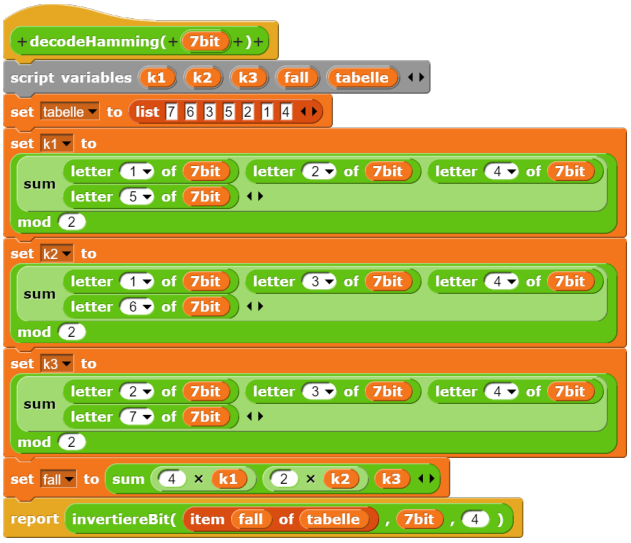

Abbildung 1.11: (7,4)-Hamming-Code Decodierung

Um einen Arbeitsschritt zu sparen, erhält der Block `invertiereBit` einen zusätzlichen Parameter `Anzahl`, der es ermöglicht, die Anzahl der Stellen einzustellen, die der Block ausgibt.

Der (7,4)-Hamming-Code nutzt den Raum der 7-Bit-Sequenzen vollständig aus. Es gibt $2^4 = 16$ verschiedene 4-Bit-Sequenzen und $2^7 = 128$ verschiedene 7-Bit-Sequenzen. Jede der auf 7 Bit vergrößerten 4-Bit-Sequenzen hat 7 „Nachbarn“, d.h. weitere Sequenzen, die sich nur um ein Bit unterscheiden. Es gilt also $16 + 16 \cdot 7 = 128$. Jeder 7-Bit-Sequenz ist ein Codewort oder unmittelbarer Nachbar eines Codeworts.

1.6 Blockparitätsverfahren

Von einem **Paritätsbit** sprechen wir, wenn an eine Folge von Bits, also von Nullen und Einsen, so eine weitere Null oder Eins angefügt wird, dass die Anzahl der Einsen insgesamt gerade (even) oder ungerade (odd) ist. Dazu ein Beispiel: Aus 001 10 wird bei gerader Parität 001 100 und bei ungerader Parität 001 101. Die Formel zu Berechnung des Paritätsbits bei gerader Parität ist `set pbit to (Summe mod 2)`, bei ungerader Parität `set pbit to ((Summe+1) mod 2)`.

Da entweder die Bitfolgen mit einer geraden Anzahl von Einsen zum Code gehören oder die Bitfolgen mit einer ungeraden Anzahl, ist der Abstand zweier Codewörter mindestens Zwei. Mit dem Paritätsbit kann man also einen Fehler erkennen, aber noch nicht korrigieren.

Beim Blockparitätsverfahren werden deshalb nicht zur zeilenweise Paritätsbits angefügt, sondern auch spaltenweise. In der Regel werden 4x4=16 Datenbits in vier Zeilen verwendet, wozu noch 9 Paritätsbits kommen.

d_1	d_2	d_3	d_4	p_1
d_5	d_6	d_7	d_8	p_2
d_9	d_{10}	d_{11}	d_{12}	p_3
d_{13}	d_{14}	d_{15}	d_{16}	p_4
p_5	p_6	p_7	p_8	p_9

Im Folgenden beschränken wir uns auf gerade Parität (even parity).

A.	0	1	0	0	1	B.	0	1	1	0	0
	1	0	0	1	0		0	1	1	0	0
	0	1	1	0	0		0	1	0	0	0
	1	1	1	0	1		1	1	1	1	0
	0	1	0	1	0		1	0	0	1	0
C.	0	1	0	1	1	D.	0	1	0	1	0
	0	0	1	0	1		0	0	0	1	1
	1	1	1	0	0		0	0	1	1	0
	1	1	0	1	1		0	1	0	0	1
	1	1	1	0	1		0	1	0	1	0

Betrachten wir die vier Beispiele oben. In Block A sind alle fünf Zeilenparitäten und alle fünf Spaltenparitäten gerade, also liegt kein (erkennbarer) Fehler vor. In Block B hat die 3. Zeile und die 3. Spalten eine ungerade Parität,

alle anderen Zeilen und Spalten eine gerade Parität. Also wurde das Bit in der 3. Zeile und der 3. Spalte falsch übertragen, es muss eine Eins sein statt einer Null. In Block C haben die 1. Zeile und die 3. Zeile eine ungerade Parität sowie die 1. Spalte und die 3. Spalte. Alle anderen Zeilen und Spalten haben eine gerade Parität. Man erkennt also, dass zwei Fehler vorliegen. Allerdings gibt es zwei Möglichkeiten, die Fehler zu korrigieren: Man könnte die Elemente an 1. Zeile/1. Spalte und 3. Zeile/3. Spalte invertieren oder die Elemente an 1. Zeile/3. Spalte und 3. Zeile/1. Spalte. In diesem Fall werden zwei Fehler erkannt, können aber nicht korrigiert werden. In Block D haben alle Zeilen eine gerade Parität. Von den Spalten weisen die 2. und die 3. Spalte eine ungerade Parität auf. Es liegen also zwei Fehler vor. Zur Korrektur der Fehler gibt es hier vier Möglichkeiten: man könnte die Zahlen in der 2. und 3. Spalte in der 1., 3., 4. oder 5. Zeile miteinander tauschen. Auch in diesem Fall werden die beiden Fehler also erkannt, können aber nicht korrigiert werden.

Ändert sich ein Datenbit, so müssen zwei Paritätsbits angepasst werden. Der Abstand zwischen zwei Codewörtern beträgt also 3. Damit kann das Blockparitätsverfahren einen Fehler pro 16 Datenbits bzw. 25 übertragenen Bits korrigieren.

Im ASCII-Code kann jedes Zeichen durch 8 Bit (eine achtstellige Dualzahl) dargestellt werden. In einem Block von 16 Datenbits können also zwei ASCII-Zeichen übertragen werden.

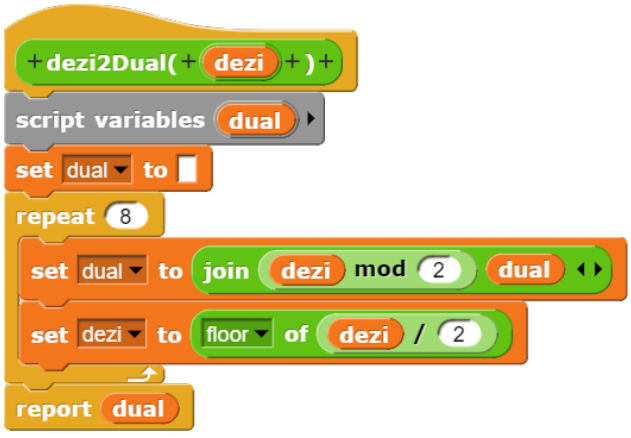

Abbildung 1.12: Dezimal in Dual

Für die Implementierung sind zwei Hilfsblöcke erforderlich, um eine Dezimalzahl in eine achtstellige Dualzahl umzuwandeln und umgekehrt. Dazu passen wir die Blöcke aus Kapitel 6.2 und 6.3 des Skriptes für Jahrgang 11 so an, dass eine achtstellige Dualzahl entsteht. Wir wiederholen deshalb acht Mal die beiden folgenden Schritte:

- Die Dualziffern erhalten wir, indem wir den Rest bei der Division der Dezimalzahl durch 2 vorn an das Ergebnis ansetzen.
- Die Dezimalzahl wird durch 2 geteilt und das Ergebnis abgerundet.

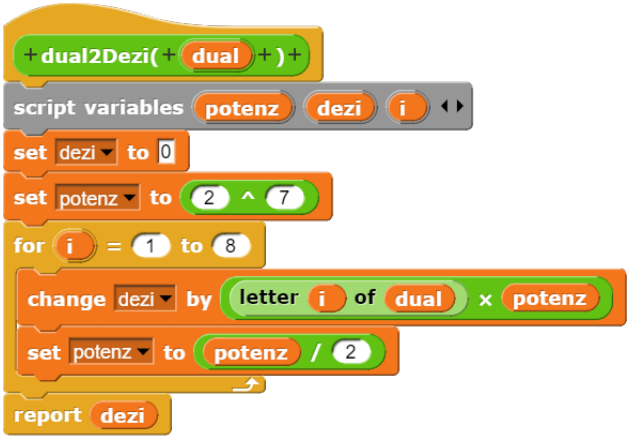

Abbildung 1.13: Dual in Dezimal

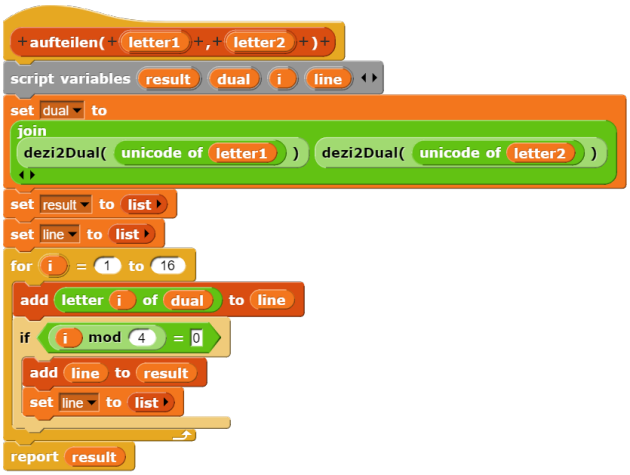

Abbildung 1.14: Aufteilen in 4x4-Block

Zum Umwandeln einer achtstelligen Dualzahl in die zugehörige Dezimalzahl setzen wir nächst `dezi` auf Null und `potenz` auf $2^7 = 128$. Dann wiederholen wir acht Mal:

- Das Produkt aus dem aktuellen Wert von Potenz und der aktuellen Ziffer wird zu `dezi` hinzu addiert.
- Die Potenz wird halbiert.

Abschließend wird `dezi` als Ergebnis zurückgegeben.

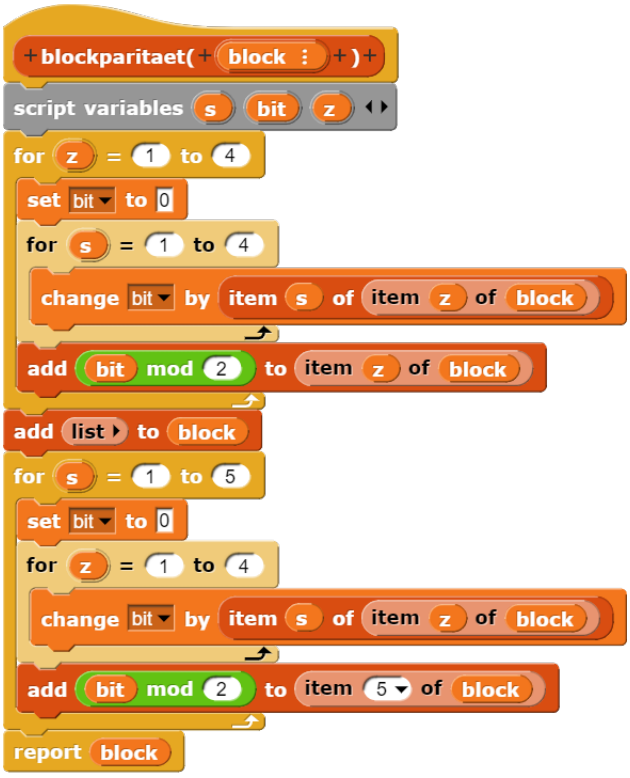

Abbildung 1.15: Blockparität

Der Block `aufteilen` erhält zwei Buchstaben als Parameter. Die beiden Buchstaben werden zunächst in Dezimalzahlen umgewandelt und dann in zwei achtstellige Dualzahlen, die zusammengesetzt werden. Die 16 Ziffern werden dann in je vier Zeilen zu je vier Stellen eingetragen und zu einer Tabelle zusammengesetzt.

Der Block `blockparitaet` erhält eine 4x4-Tabelle mit den Datenbits von zwei Buchstaben. Zunächst werden die Paritätsbits der vier Zeilen berechnet und jeweils als fünftes Element angefügt. Dann wird eine 5. Zeile angefügt und die Paritätsbits der fünf Spalten berechnet und in die 5. Zeile eingetragen. Das Ergebnis ist eine 5x5-Tabelle, bestehend aus 16 Datenbits und 9 Paritätsbits.

Der Block `oddParity` liefert Zeilen- und Spaltennummern von aller Zeilen und Spalten

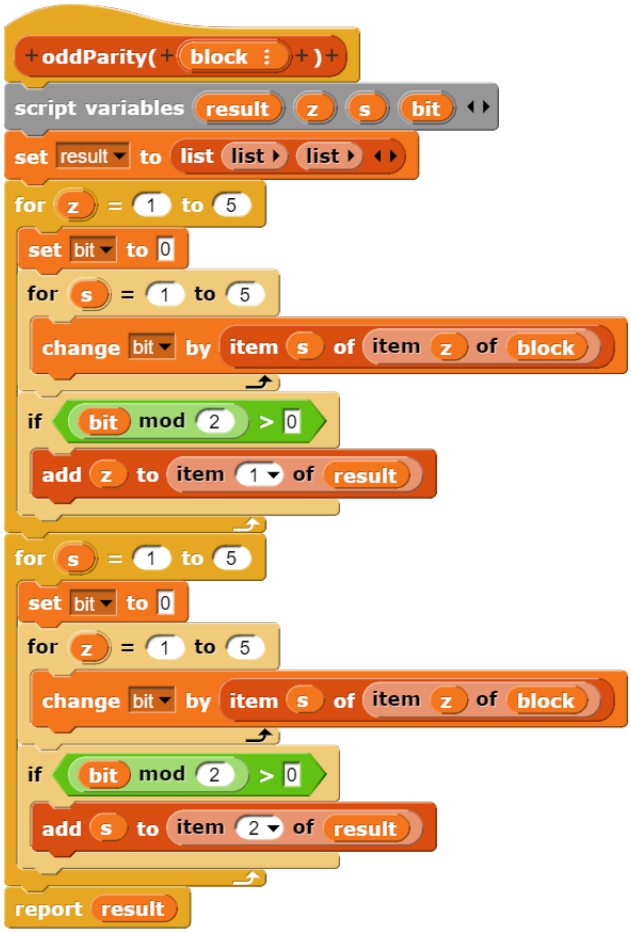

Abbildung 1.16: Bestimmung der Zeilen und Spalten ungerader Parität

mit ungerader Parität. Dazu erstellt der Block eine Liste mit zwei Listen, wobei die erste Liste die Zeilennummern mit ungerader Parität und die zweite Liste die Spaltennummern mit ungerader Parität enthält.

Bei der Bestimmung der Blockparität treten drei Fälle auf:

- Alle Paritäten sind gerade, d.h. die durch `oddParity` erzeugten Listen sind leer.
- Es liegt je eine ungerade Zeile- und eine ungerade Spaltenparität vor, d.h. die durch `oddParity` erzeugten Listen erhalten je ein Element. Das bedeutet, dass das Bit im Schnittpunkt dieser Zeile und Spalte invertiert wurde und korrigiert werden kann.
- In allen anderen Fällen liegt mehr als ein Fehler vor, d.h. die Fehler können nicht mehr korrigiert werden.

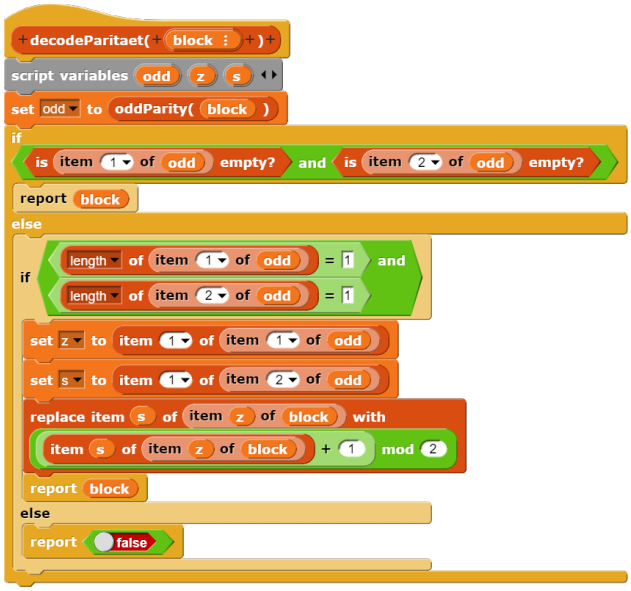

Abbildung 1.17: Blockfehler korrigieren

Der Block `decodeParity` ruft zunächst `oddParity` auf, um die ungeraden Paritäten zu bestimmen. Falls alle Paritäten gerade sind, liegt kein Fehler vor und der als Parameter übergebene Block wird unverändert zurückgegeben. Falls je eine ungerade Spaltenparität und eine ungerade Zeilenparität auftreten, wird das Bit im Schnittpunkt invertiert und der so korrigierte Block zurückgegeben. Falls keiner dieser beiden Fälle auftritt, gibt `decodeParity false` zurück.

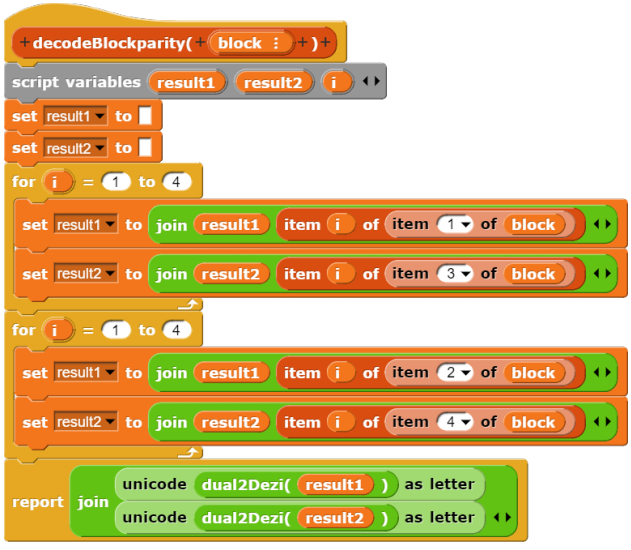

Abbildung 1.18: Block entschlüsseln

Der Block `decodeBlockparity` erhält als Parameter einen 5x5-Block, der bereits korrigiert wurde, sofern dies möglich ist. Das Ergebnis wird hier aufgeteilt in `result1` und `result2`. Zunächst werden in einer `FOR`-Schleife die Datenbits der ersten Zeile zu `result1` und die Datenbits der zweiten Zeile zu `result2` zusammengesetzt. In der zweiten `FOR`-Schleife werden die Datenbits der dritten Zeile an `result1` angefügt und die Datenbits der vierten Zeile an `result2`. Damit sind zwei achtstellige Dualzahlen entstanden, die zunächst in Dezimalzahlen und dann in Buchstaben umgewandelt werden.

1.7 Komprimierende Codes

Als komprimierende Codes bezeichnet man Algorithmen, die den Speicherplatzbedarf von Daten verringern, um diese Daten schneller bzw. kostengünstiger zu übertragen oder zu speichern.

Man unterscheidet zwischen zwei Arten der Komprimierung. Zum einen gibt es die **verlustfreie Kompression**, bei der nach Codierung und Decodierung die anfänglichen Daten wiederhergestellt werden. Eine verlustfreie Komprimierung ist nur möglich, wenn die Originaldaten über Redundanz verfügen. Man bezeichnet diese Verfahren deshalb auch als Redundanzreduktion. Bei Programmcodes und Texten muss man in der Regel verlustfreie Verfahren verwenden.

Zum anderen gibt es jedoch verlustbehaftete Verfahren, bei welchen man von **Irrelevanzreduktion** spricht. Bei der Irrelevanzreduktion werden irrelevante, also überflüssige Informationen weggelassen. Ein Beispiel hierfür sind etwa Audioformate, bei denen für den Menschen nicht hörbare Töne ausgelassen werden, oder Bildformate, die die Auflösung reduzieren. Oft kann mit der Kombination verschiedener Codes ein optimales Ergebnis erzielt werden.

Welche Anforderungen werden an komprimierende Codes gestellt? Wie oben beschrieben ist das Ziel komprimierender Codes die Reduzierung der benötigten Speichermenge. Daraus geht die erste Anforderung hervor, nämlich möglichst viel Speichermenge einzusparen. Eine weitere Anforderung ist, dass die Dateien möglichst fehlerfrei codiert werden, um beim Decodieren keine fehlerhafte Datei zu erhalten.

Auch die Geschwindigkeit, mit der die Daten komprimiert beziehungsweise dekomprimiert werden, spielt bei der Nutzung von komprimierenden Codes im Datenverkehr eine wichtige Rolle, gerade in Zeiten des mobilen Internets.

1.8 Lauflängencodierung

Wir befassen uns zuerst mit der Lauflängencodierung beschäftigen. englisch Runtime Length Encoding (RLE). Sie beruht auf der Eigenschaft, dass viele Daten sogenannte Läufe, also Folgen identischer Zahlen, enthalten. Bei der Lauflängencodierung handelt es sich um eine Redundanzcodierung. Sie ist komplett verlustfrei.

Wie funktioniert die Lauflängencodierung?

Wie eben schon erwähnt, beruht das Prinzip der Lauflängencodierung auf der Tatsache, dass Daten Läufe enthalten. Die Grundidee ist, solche Läufe durch die Anzahl der Zeichen des Laufes und das Zeichen zu codieren. Es muss jedoch sicher gestellt sein, dass der Zähler von Daten unterschieden werden kann. Deswegen nutzt die Lauflängencodierung einen sogenannten Marker, der darauf hinweist, dass jetzt eine Anzahl folgt. Typischerweise werden Läufe also durch (Zeichen Marker Anzahl) codiert. Nun ergibt sich jedoch ein Problem, wenn der Marker selbst als Zeichen in den Daten auftaucht. Ein Beispiel: Wir nehmen als Marker das Zeichen %. Wir wollen die Zeichenfolge „7%44444“ codieren. Wir würden nach unserer bisherigen Erkenntnis die Zeichenfolge „7%4%5“ erhalten. Beim Decodieren würden wir dann jedoch folgende Daten erhalten „777744444“. Diese entsprechen jedoch nicht den zuvor codierten Daten. Der Fehler liegt darin, das in den Daten der Marker als Zeichen vorliegt, vom Code jedoch nicht vom Marker unterschieden werden kann. Deshalb wird der Marker selbst als (Marker Marker) codiert. Nehmen wir nochmal unsere Beispieldaten: „7%44444“. Beim codieren würde sich nun die Zeichenfolge „7%%4%5“ ergeben. Beim Decodieren erhalten wir nun wieder die ursprünglichen Daten „7%44444“. In unserem kleinen Beispiel haben wir so die Daten um ein Zeichen von sieben auf sechs Zeichen reduziert. Zu erwähnen ist noch, dass es ebenfalls Codes gibt, die auch Läufe von Zeichenfolgen codieren.

Lauflängencodierung (RLE Runtime Length Encoding)
20W
20W
1W1S3W1S1W5S1W4S3W
1W1S3W1S3W1S3W1S6W
1W1S3W1S3W1S3W1S6W
1W5S3W1S3W4S3W
1W1S3W1S3W1S3W1S6W
1W1S3W1S3W1S3W1S6W

Im ungünstigsten Fall verdoppelt sich der Speicherbedarf bei der Lauflängencodierung. Das Muster
RGBG
BRGB
GBRG
BGBR
würde folgende Codierung ergeben:
1R1G1B1G
1B1R1G1B
1G1B1R1G
1B1G1B1R

Um die „Aufblähung“ von Dateien zu vermeiden, gibt es eine Variante: Die Lauflängencodierung wird nur dann durchgeführt, wenn mindestens drei gleichfarbige Bits aufeinander folgen.

Aufgabe 1.8.1
- *Zeichne Dein Monogramm in 20x20 Kästchen und führe eine Lauflängencodierung durch. Berechne die Speicherersparnis.*
- *Erstelle ein Struktogramm, das den eben gefundenen Fehler korrigiert und den verbesserten RLE-Algorithmus umsetzt. (Hinweis: „Structorizer“, läuft derzeit leider nicht auf den Schulrechnern (Ausnahme: Netbooks)*

Anwendungsbereiche der Lauflängencodierung: Da für eine möglichst hohe Komprimierungsrate möglichst lange Läufe vorhanden sein müssen, wird die Lauflängencodierung meist zur Codierung von Bildern, Grafiken oder einfachen Clipart Bildern verwendet, bei denen wenige Farben genutzt wurden, wie zum Beispiel bei Schwarz-Weiß-Bildern. Es ist

jedoch nicht üblich die Lauflängencodierung direkt auf die Pixel von Fotos anzuwenden, da häufig nur kurze Läufe vorhanden sind. Ein älteres jedoch bekanntes Dateiformat, welches die Lauflängencodierung nutzt, ist etwa Windows Bitmap. Außerdem kommt auch bei der Fax-Übertragung die Lauflängencodierung zum Einsatz.

1.9 Huffman-Codierung

Die Huffman-Codierung wurde 1951 entwickelt von David A. Huffman (1925-1999) im Rahmen einer Examensarbeit. Seine Grundidee war: häufig vorkommende Codewörter werden mit einem kürzeren Code codiert, selten vorkommende Codewörter mit einem längeren Code. Bei der Huffmann-Codierung handelt es sich wie auch bei der Lauflängencodierung um ein verlustfreies Kompressionsverfahren.

1.9.1 Fano-Bedingung

Die Fano-Bedingung bezeichnet in der Kodierungstheorie die Eigenschaft einer Sprache, **präfix-frei** zu sein. Dies bedeutet, dass kein Wort der Sprache der Anfang eines anderen Wortes der Sprache ist.

Betrachten wir dazu das Beispiel des Morsealphabets. Die Tabelle zeigt einen Ausschnitt des internationalen Morsealphabets:

A	.-	I	..
D	-..	S	...
E	.	U	..-

Wenn man mit Hilfe der Tabelle das Wort „USA“ codiert, so erhält man **..- ... .-**. Das Wort „IDEA“ liefert **.. -.. . .-**. Es handelt sich um die gleiche Zeichenkette, die damit mindestens zwei verschiedene Decodierungsmöglichkeiten besitzt.

Das internationale Morsealphabet ist nicht präfix-frei. Wenn wir Codeworte unterschiedlicher Länge verwenden, dann müssen wir voraussetzen können, das ein Code präfix-frei ist, sonst ist eine einwandfreie Decodierung nicht gewährleistet. Die Leistung von Huffman bestand u.a. darin, dass er ein Verfahren zu Konstruktion eines präfix-freien Codes entwickelt hat.

Die Fano-Bedingung spielt nicht nur in der Huffman-Codierung eine Rolle. Vielmehr liefert sie ein Kriterium, die Eignung eines Codes zu beurteilen. Es gibt deshalb viele Abituraufgaben, in denen Codes untersucht werden müssen. Sie kreisen um die Frage, ob ein Code präfixfrei und damit in der Lage ist, eine einwandfreie Codierung zu liefern, oder welche Veränderungen vorgenommen werden müssen, damit er präfixfrei wird. Der Huffman-Baum kann dabei als Hilfsmittel dienen. Zeichnet man einen Baum mit den Codebuchstaben, dann kann man erkennen, an welchen Stellen Codeworte Präfixe anderer Codeworte sind, weil diese in ihrem Weg zur Wurzel. Eine Voraussetzung ist in jedem Fall, dass Codeworte unterschiedlicher Länge vorkommen. Ein Code, der nur Codeworte einer festen Länge enthält, ist automatisch präfixfrei.

1.9.2 Arbeitsschritte bei der Huffman-Codierung

1. Schritt: Zähle die Häufigkeit der einzelnen Buchstaben/Zeichen und ordne sie der Häufigkeit nach.

D	K	R	B	A
1	1	2	2	5

2. Schritt: Fasse - beginnend mit den seltensten Knoten - die zwei Knoten mit der jeweils kleinsten Häufigkeit zu einem Knoten mit der kombinierten Häufigkeit zusammen, bis nur noch ein Knoten übrig bleibt. Achtung: In diesem Schritt muss bei jedem Zwischenschritt darauf geachtet werden, dass man die beiden Knoten mit der geringsten Häufigkeit zusammenfasst. Ein häufiger Schülerfehler besteht darin, dass nach der ursprünglichen Sortierung einfach nebeneinander liegende Knoten zusammengefasst werden.

   Der Baum wurde nach dem Zeichnen so umgestellt, dass an jedem inneren Blatt der linke Teilbaum einen kleinere Häufigkeit hat als der rechte Teilbaum.

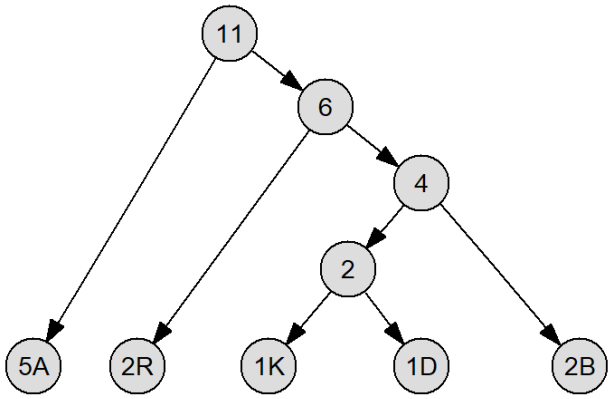

Abbildung 1.19: Huffman-Baum für ABRAKADABRA

3. Schritt: Der Code ergibt sich ausgehend von der Wurzel. Jeder Kante, die nach links abgeht, wird die 0 zugeordnet, jeder Kante nach rechts die 1.

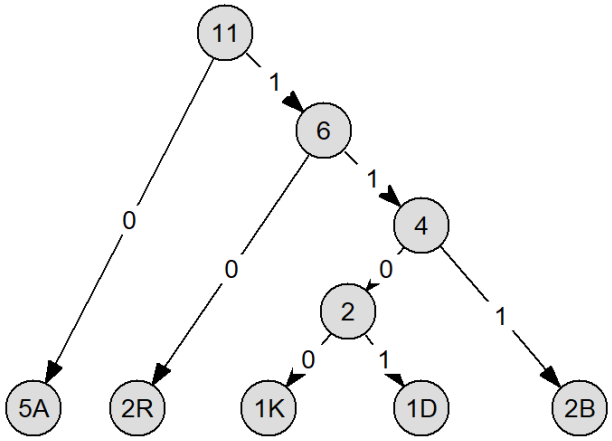

Abbildung 1.20: Code-Baum für ABRAKADABRA

4. Schritt: Aus dem Baum wird die Codetabelle abgeleitet.

A	R	K	D	B
0	10	1100	1101	111

5. Schritt: Mit Hilfe der Codetabelle wird jedem Buchstaben der zugehörige Code zuordnet. Im Beispiel erhält man „b01111001100011010111100“. Die entstehende Dualzahl kann in speichergerechte Stücke zerlegt werden. Bei der Implementierung muss man darauf achten, dass man dem Ergebnis ein „b“ voranstellt. Sonst interpretiert Snap! das Ergebnis als Zahl, lässt führende Nullen weg und rechnet es um in die wissenschaftlichen Schreibweise mit Exponent. Im Beispiel erhalten wir einen codierten Text von 23 Bit. Gegenüber $11 \cdot 8 = 88$ Bit ist das eine Ersparnis von fast 74%. Allerdings muss hiervon noch der Speicherbedarf für die Codetabelle bzw. für den Codebaum abgezogen werden.

Die Decodierung kann mit Hilfe des Huffman-Baums erfolgen oder mit Hilfe der Codetabelle. Bei der Decodierung mit Baum beginnt man mit der Wurzel. Für jede „1“ nimmt man jeweils den rechten, für jede „0“0 jeweils den linken Zweig, bis man an einem Blatt angelangt ist. Man fügt nun den Buchstaben des Blattes hinten an das decodierte Wort an und beginnt am Baum wieder von oben.

Warum ist der so entstandene Code präfixfrei? Die Buchstaben oder Zeichen stehen ausschließlich in den Blättern. Alle inneren Knoten sind durch Addition von Häufigkeiten entstanden und enthalten keine Buchstaben. Ein Präfix läge nur dann vor, wenn man auf dem Weg von der Wurzel zu einem Blatt durch einen inneren Knoten zieht, der auch einen Buchstaben enthält.

1.9.3 Anwendung

Kommen wir nun zu den Anwendungsbereichen. Die Huffman-Codierung ist ein äußerst weit verbreitetes und beliebtes Kompressionsverfahren. So wird die Huffman-Codierung zum Beispiel in bekannten Dateiformaten wie etwa MP3 oder JPEG angewandt. Die Huffman-Codierung ist dort jedoch nur ein Teil einer Reihe von hintereinander angewandten Kompressionsverfahren. Sie ist zwar verlustfrei, jedoch sind es andere angewandte Kompressionsschritte nicht, weshalb die Formate MP3 und JPEG trotzdem verlustbehaftet sind. Auch zum Komprimieren von Texten wird die Huffman-Codierung angewandt, ebenso wie bei der Faxübertragung, nach Anwendung der Lauflängencodierung.

Ein Problem der Huffman-Codierung ist allerdings, dass der Codebaum bzw. die Codetabelle an die jeweiligen Daten angepasst wird. Das führt zwar einerseits zu einem möglichst

knappen Code, erfordert aber andererseits, dass Codebaum und Codetabelle zusammen mit jedem Datensatz abgespeichert werden. Bei kurzen Datensätzen bzw. Texten ist die Ersparnis durch die Huffman-Codierung geringer als der zusätzlich notwendige Speicherplatz für Baum bzw. Tabelle. Dann lohnt sich diese Art der Codierung nicht.

1.9.4 Implementierung bis zur Codetabelle

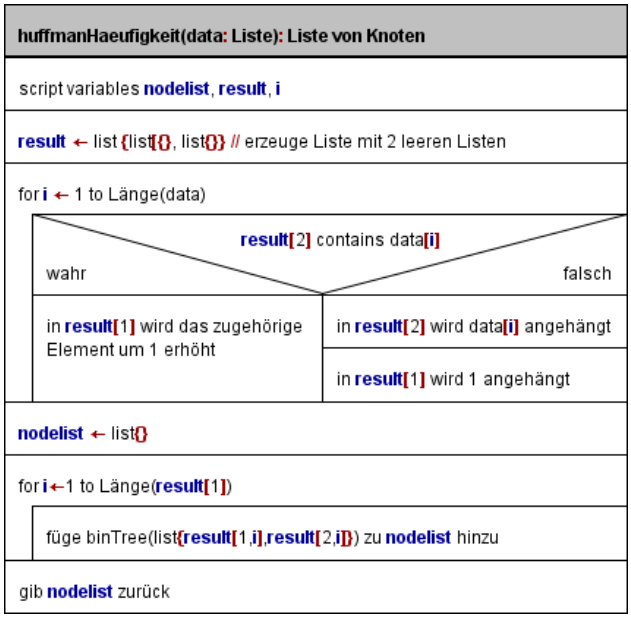

Abbildung 1.21: Struktogramm zur Erstellung der Knotenliste

Wir wollen nicht nur Texte mit dem Algorithmus implementieren, sondern auch Kombinationen wie `3W` oder `2S` aus der Lauflängencodierung. Deshalb verwenden wir als Eingangsparameter keine Zeichenkette, sondern eine Liste. Falls ein Text zur Codierung vorliegt, können wir diesen mit `split by letter` in eine Liste umwandeln.

Zur Implementierung müssen wir zunächst die einzelnen Buchstaben oder Zeichen zählen. Dafür verwenden wir eine geschachtelte Liste, die wir als Tabelle darstellen. Im ersten Element speichern wie die Häufigkeit der einzelnen Buchstaben, im zweiten Element die Buchstaben bzw. Zeichen. In dieser Form der Tabelle können wir zwei Listenbehle ausnutzen, nämlich `list contains element`, der angibt, ob ein Element in einer Liste enthalten ist. Falls das Element enthalten ist, bestimmen wir mit `index of element in liste` die Stelle, an der der Buchstabe in der Liste steht, und addieren an dieser Stelle in der ersten Liste 1. Ist das Element nicht enthalten, hängen wir den neuen Buchstaben an die zweite Liste an und die 1 für das erste Vorkommen an die erste Liste.

Zur Vorbereitung der Baumkonstruktion fassen wir jeweils ein Element der beiden Listen zu einem Knoten zusammen.

Um die Implementierung übersichtlicher zu halten, teilen wir die Operation in zwei Blöcke auf.

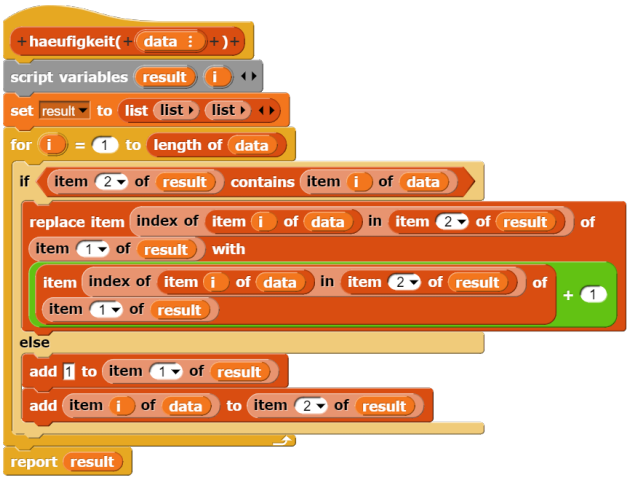

Abbildung 1.22: Auszählen der Häufigkeit

Wir erzeugen eine Liste `result` mit zwei leeren Listen als Elementen. Die erste Liste soll die Häufigkeiten aufnehmen, die zweite die zu codierenden Zeichen. Dann gehen wir mit einer `FOR`-Schleife die als Parameter `data` übergebene Liste durch und prüfe jeweils, ob das aktuelle Element von `data` bereits in der zweiten Liste enthalten ist. Falls ja, wird die Häufigkeit um Eins erhöht. Falls nein, wird das neue Element in die zweite Liste aufgenommen und in der ersten Liste die Häufigkeit Eins notiert.

Die Liste mit den Häufigkeiten und den zu codierenden Zeichen müssen wir in ein Liste mit Knoten umwandeln, um die Knoten zum Huffman-Baum zusammenzufassen. Dazu erzeugen wir eine leere Liste `nodelist`. Dann gehen wir mit einer `FOR`-Schleife die als Parameter übergebene Tabelle durch. Wir fassen

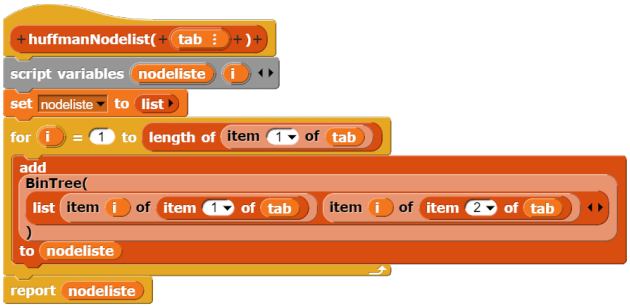

Abbildung 1.23: Erstellen der Knotenliste

jeweils ein Element aus der ersten Teilliste (den Häufigkeiten) und ein Element aus der zweiten Teilliste (den zu codierenden Zeichen) in einer Liste zusammen und erzeugen dann einen Binärbaum mit der zweielementigen Liste als Wurzel.

Abbildung 1.24: Knoten-Operationen

Um aus der Knotenliste den Huffman-Baum zu erstellen, müssen wir die Knoten sortieren. Dazu benötigen wir zunächst einige Operationen, um auf die Merkmale der Knoten zugreifen zu können. Für die Häufigkeit benötigen wir einen `GET`- und einen `SET`-Operator, weil wir die Häufigkeit auslesen müssen, aber auch setzen, wenn wir Knoten zusammenfassen. Für die Buchstaben reichte eine `GET`-Operation zum Auslesen für die Codetabelle.

Welches Sortierverfahren wir verwenden, ist im Grunde unerheblich. Wir haben allerdings nach dem ersten Sortieren immer vorsortierte Listen, die durch Löschen von zwei Elementen und dem Hinzufügen eines neuen Elements aus der alten Liste entstehen. Für vorsortierte Liste eignet sich **BubbleSort** in der optimierten Fassung besonders gut, weil es über eine vorzeitige Abbruchbedingung verfügt. Wenn wir den neuen Knoten immer vorn einfügen, dann steht er nach dem ersten Durchgang auf dem richtigen Platz und die Sortierung ist fertig.

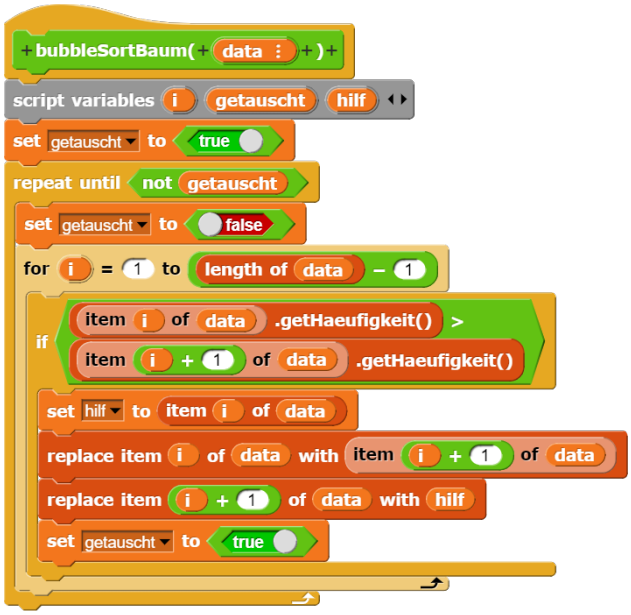

Abbildung 1.25: Bubble-Sort für Bäume mit Häufigkeit

Die einzige Änderung, die wir gegenüber der Grundform vorgenommen haben, ist in der Bedingung der IF-Verzweigung. Statt direkt die beiden Element zu vergleichen (`if item i of data > item i+1 of data`) greifen wir jetzt auf die Häufigkeit zu.

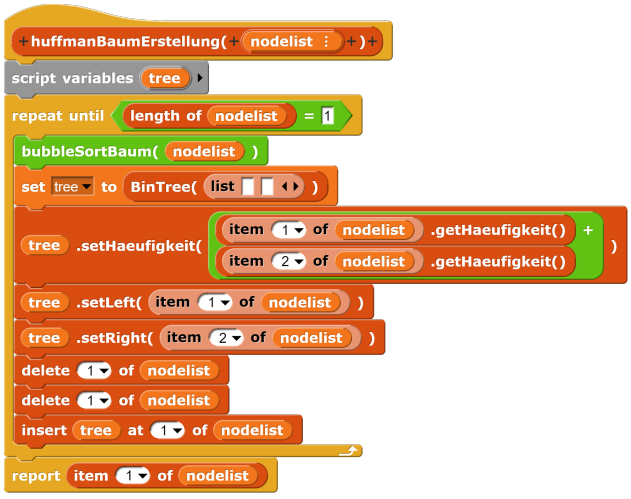

Abbildung 1.26: Baum erstellen

Wir erstellen den Baum aus der Knotenliste, indem wir die Knotenliste nach der Häufigkeit

sortieren. Dann erzeugen wir einen leeren Baum, tragen die kombinierte Häufigkeit aus den beiden seltensten Knoten ein und setzen diese beiden Knoten als linken und rechten Teilbaum in den neuen Baum ein. Anschließend werden zusammengefassten Knoten aus der Knotenliste entfernt und im Baum gespeichert. Dabei löschen wir zwei Mal das erste Element, weil nach dem ersten Löschvorgang das zweite auf den ersten Platz vorrückt. Dieses Verfahren wiederholen wir, bis nur noch ein Element in der Knotenliste vorhanden ist. Das ist der Huffman-Baum.

Das Auslesen der Codes aus dem Huffman-Baum ist der komplizierteste Schritt, weil wir dazu den Baum rekursiv durchlaufen müssen.

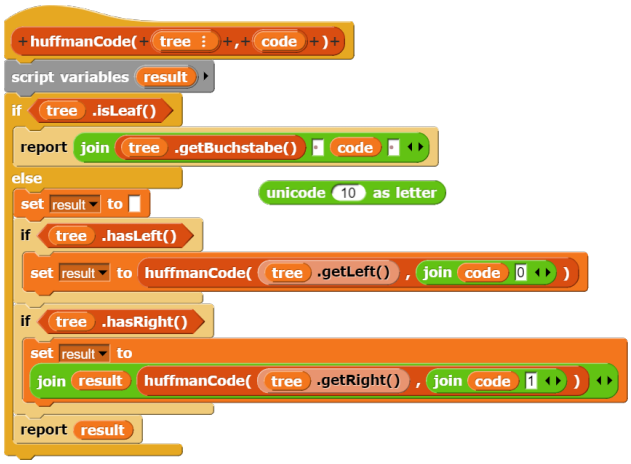

Abbildung 1.27: Auslesen des Codes

Wir verwenden neben dem Baum einen zweiten Parameter Code, der zunächst mit einem Leerstring aufgerufen wird und der dann beim jedem Durchgang um eine Null (falls wir den linken Teilbaum nehmen) oder eine Eins (falls wir den rechten Teilbaum nehmen) ergänzt wird. Die Buchstaben/Zeichen stehen bei Hufmann-Bäumen nur in den Blättern. Wir geben deshalb die Buchstaben und den zugehörigen Code zurück und setzen diese aus den Teilbäumen dann zusammen. Im Ergebnis erhalten wir eine Zeichenkette, die abwechselnd Buchstaben und Code enthält, geordnet nach dem Code.

In der Abbildung sieht einen Block `unicode 10 as letter`, der nicht fest eingebaut ist. Man kann diesen hinten im `JOIN` des ersten Command nach `if tree.isLeaf()` einsetzen, um einen Zeilenumbruch einzufügen, damit man eine Tabelle erhält. Lässt man den Zeilenumbruch weg, dann erhält man eine Zeichenkette, in der die jeweils zu codierenden Zeichen stehen, gefolgt von ihrer Häufigkeit. Als Trennzeichen zwischen Zeichen und Häufigkeit und zwischen den Datensätzen verwenden wir jeweils ein Leerzeichen.

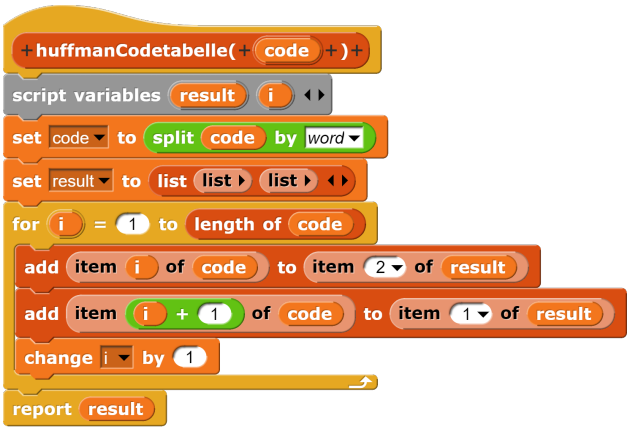

Abbildung 1.28: Erstellen der Codetabelle

Für die Codetabelle verwenden wir das gleiche Format einer Liste mit zwei Teillisten wie für die Häufigkeit. Unsere Zeichenkette mit den Codes wird mit `split by word` in eine Liste aufgeteilt, die dann abwechselnd auf die beiden Teillisten übertragen wird. Wir fügen ein `change i by 1` ein, damit die Zählvariable immer um Zwei weiter gezählt wird, weil erst das übernächste Element wieder ein Zeichen enthält. Im Ergebnis entsteht die Codetabelle.

0	10	1100	1101	111
A	R	K	D	B

Abbildung 1.29: Codetabelle für ABRAKADABRA

Es handelt sich bei den Blöcken in diesem Abschnitt um Reporter, die einzeln getestet, aber auch zusammengesetzt werden können.

1.9.5 Encoding und Decoding

Mit der Codetabelle gestalten sich Kodierung und Decodierung von Huffman einfach. Um deutlich zu machen, dass es sich bei der Ausgabe nicht um eine Zahl, sondern um eine Zeichenkette handelt, setzen wir an den Anfang

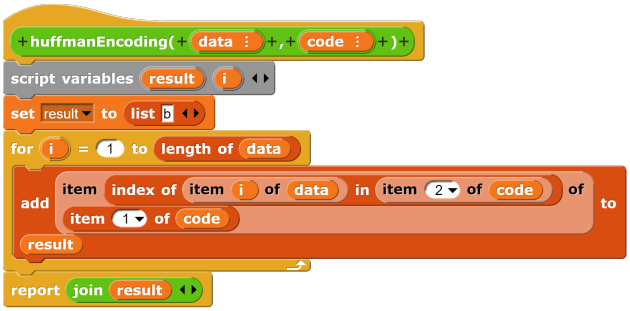

Abbildung 1.30: Huffman-Encoding

von `result` eine kleines `b` für Binärzahl. Eine längere Zahl würde nämlich in Exponentialschreibweise ausgegeben. Beim Codierung gehen wir mit einer `FOR`-Schleife die Liste mit den Zeichen durch, suchen in der unteren Liste die Position des aktuellen Zeichens und hängen dessen Code an das Ergebnis an. Abschließend wandeln wir durch `join(result)` das Ergebnis in eine Zeichenkette um.

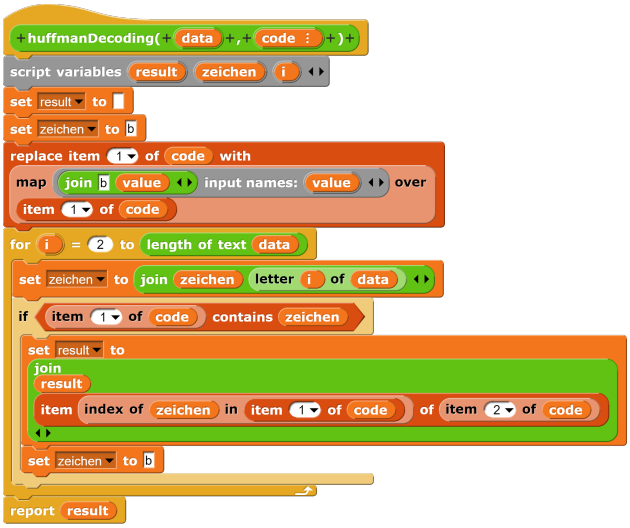

Abbildung 1.31: Huffman-Decoding

Das Decodieren ist etwas aufwendiger, weil nicht jedes Bit des codierten Textes für ein Zeichen steht. Zunächst machen uns die Codezeichen Probleme, weil sie aus eine Folge von Nullen und Einsen bestehen. Eine Folge von Ziffern wird in Snap! als Zahl interpretiert und dass bedeutet, dass führende Nullen weggelassen werden. Diese Verfahrensweise führt zu Falschdecodierungen. Zur Abhilfe wandeln wir die Codetabelle so um, dass wir vor jedes Zeichen ein kleines `b` setzen für Binärzahl. Dafür nutzen wir den Map-Befehl: `replace item 1 of code with map ringify(join(b,value)) over item 1 of code`.

Eine Hilfsvariable `zeichen` wird mit `b` vorinitialisiert. Anschließend setzen wir die gelesenen Bits so lange zusammen, bis wir einen Eintrag in der oberen Liste der Codetabelle, also ein Codewort, finden und fügen dann das entsprechende Zeichen aus der unteren Liste dem Ergebnis hinzu.

1.10 Aufgaben

Aufgabe 1.10.1 *Codiere die Bitfolgen 0011, 1001 und 1011 im (7,4)-Hamming-Code.*

Aufgabe 1.10.2 *Decodiere die Bitfolgen 0001111, 0010110, 1001100 und 1100110 im (7,4)-Hamming-Code.*

Aufgabe 1.10.3 *Berechne den Hamming-Abstand der folgenden Codes:*

- $C_1 = \{00001, 00110, 11000\}$
- $C_2 = \{01100, 10011, 11001, 10101\}$

Aufgabe 1.10.4 *Zeige, dass der folgend Code keinen Fehler korrigieren kann:* $C_3 = \{0111, 0100, 1001\}$

Aufgabe 1.10.5 *Berechne den Hamming-Abstand des (7,4)-Hamming-Codes. Implementiere die dazu notwendigen Operationen.*

Aufgabe 1.10.6 *Untersuche, ob die folgenden Codes linear sind:*

- $C_4 = \{000, 011, 101, 110\}$
- $C_5 = \{0000, 0001, 0011, 0010\}$
- $C_6 = \{00000, 10001, 01101, 11100, 11101\}$
- $C_7 = \{00000, 10111, 01110, 11011\}$

Aufgabe 1.10.7 *In Kapitel 9.2 und 9.3 haben wir das Konzept der „Fehlerkorrektur“ behandelt. Parallel dazu gibt es das Konzept der „Fehlererkennung“: Der Empfänger bemerkt bzw. detektiert einen Fehler, falls er eine Bitsequenz erhält, die nicht als Codewort figuriert. Er ist nicht beauftragt, die empfangene Bitfolge zu decodieren. Geben Sie eine Formel mit Hilfe des Minimalabstandes an, wie viele Fehler ein Code detektieren kann.*

Aufgabe 1.10.8 *Konstruiere einen linearen Code mit 4 Codewörtern der Länge 6, welcher einen Fehler korrigiert.*

Aufgabe 1.10.9 *Gegeben ist die Zeichenkette FISCHERSFRITZFISCHTFRISCHEFISCHE*

Ermitteln Sie für diese Zeichenkette eine Code-Tabelle nach dem Huffman-Verfahren, codieren Sie die Zeichenkette und erläutern Sie das Verfahren.

Vergleichen Sie die Größe der unkomprimierten und der komprimierten Zeichenkette quantitativ unter der Voraussetzung, dass die Zeichen der gegebenen Zeichenkette im ASCII-Format vorliegen.

Aufgabe 1.10.10 *Gegeben ist der folgende Strichcode für Ziffern, wobei W für einen weißen Balken und S für einen schwarzen Balken steht:*

Ziffer	Codewort
0	WSWSS
1	WSW
2	WWWSW
3	WSWS
4	WWWS
5	SWSSS
6	SSSW
7	SWSSSW
8	SWS
9	SSSWS

- *Zeichne den zugehörigen Huffman-Baum für W=0 und S=1 und gib an, welche Codeworte Präfixe anderer Codewörter sind.*
- *Entwickle eine neue Codetabelle für die Ziffern von 0 bis 9 mit minimaler, konstanter Codewortlänge. In jedem Codewort dürfen dabei maximal zwei gleichfarbige Balken nebeneinander stehen. Hinweis: Auch bei diesem Aufgabenteil kann ein modifizierter Huffman-Baum zur Entwicklung einer systematischen Lösung beitragen.*

Aufgabe 1.10.11 *Implementieren Sie ein Programm zur Erstellung eines Huffman-Codes. Erstellen Sie dann einen präfix-freien Morsecode, wobei Sie für die Häufigkeit der einzelnen Buchstaben die Häufigkeitsverteilung der deutschen Sprache zugrunde legen.*