LEHRSTUHL FÜR RECHNERTECHNIK UND RECHNERORGANISATION

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Projektaufgabe - 0404: Huffmankodierung

Tim Kaiser
Mehmet Gül
Nicolas Witter

1 Einleitung

1956: IBM stellt das erste magnetische Festplattenlaufwerk her. Mit einer Kapazität von 5MB war es damals noch unbezahlbar [1]. Heute ist es nicht unüblich Datenträger mit 1TB, also mehr als das 200,000-fache, oder mehr, in seinem Computer verbaut zu haben. Und die Kapazität von Massenspeicher wird auch in den kommenden Jahren noch weiter steigen. Und doch scheint der vorhandene Speicherplatz immer zu wenig zu sein, da sich die Komplexität der Programme und Dateien im gleichen Maße ausdehnt. Auch die Menge der zu speichernden Daten wächst rasant. Firmen wie Facebook oder Google betreiben riesige Serverfarmen, um mit den gewaltigen Mengen an Urlaubsfotos, Chatnachrichten und Geburtstagsglückwünschen mitzuhalten, die Millionen von Nutzern jeden Tag auf ihre Webseiten hochladen. Für diese Firmen bedeute jedes verschwendete Bit zusätzliche Kosten. So muss auch in einer Zeit schier endlos wachsenden Speichers noch darauf geachtet werden, den existierenden Platz so effizient wie nur irgendwie möglich zu nutzen. Eine häufig verwendete Möglichkeit dies zu erreichen ist die Verwendung von Kompressionsalgorithmen, die den Platz, den eine Datei braucht, verringern. Einer dieser Algorithmen, die sogenannte Huffman-Kodierung, die 1952 von David A. Huffman entwickelt wurde [2]. Mit diesem werden wir uns im Folgenden näher beschäftigen

2 Problemstellung und Spezifikation

2.1 Aufgabenstellung

Die Aufgabenstellung, die diesem Projekt zu Grunde liegt, lässt sich in einen theoretischen und einen praktischen Abschnitt aufteilen. Im Theoretischen beschäftigen wir uns zuerst mit der grundlegenden Funktionsweise des Huffmancodes, und warum er keine Trennelemente zwischen seinen kodierten Zeichen benötigt. Anschließend erarbeiten wir ein Speicherformat für die Baumstruktur und ein geeignetes Ausgabeformat. Zum Abschluss untersuchen wir unsere Implementierung, die wir im praktischen Teil vornehmen, auf Performance. Der Praktische Teil beschäftigt sich sowohl mit der Implementierung der Huffman Kodierung, als auch mit der dazugehörigen Dekodierung.

[1] http://www.pcworld.com/article/127105/article.html (15.1.17)
[2] https://en.wikipedia.org/wiki/Huffman_coding (31.1.17)

Außerdem wird hier noch eine Methode zur Ausgabe des Baumes implementiert. Die Dokumentation hierzu findet sich im Abschnitt "Dokumentation der Implementierung".

2.2 Funktionsweise der Huffman-Codierung

Die Huffman-Kodierung ist ein Algorithmus zur Komprimierung von Textdateien, der die einzelnen Zeichen des Textes mit jeweils eindeutigen Bitfolgen unterschiedlicher Länge kodiert. Die Länge der Folge, und damit der Speicherplatz den sie benötigt, ist hierbei Abhängig von der Häufigkeit des entsprechenden Zeichens im Text. Dem häufigsten Zeichen wird die kürzeste Bitfolge zugeordnet. Dadurch verkürzt sich der zu kodierende Text entsprechend. Die Zuweisung der Binärcodes zu den einzelnen Zeichen geschieht über einen Binärbaum, der aus einer vorher erstellten Häufigkeitstabelle generiert wird. Häufiger vorkommende Zeichen stehen hier auf einer höheren Ebene im Baum als Seltenere. Die genaue Bitfolge, die einem Zeichen zugeordnet wird, hängt von seiner Position im Baum ab. Anschließend wird aus diesem Baum ein sogenanntes „Dictionary“, eine Tabelle in der Zeichen und dazugehörige Binärcodes verzeichnet sind, erstellt. Mithilfe dieser Tabelle wird nun der Text kodiert und damit komprimiert. Das folgende Beispiel soll das Ganze veranschaulichen.

Zu kodierende Zeichenfolge: Huffman
Häufigkeit der einzelnen Zeichen: H:1 u:1 f:2 m:1 a:1 n:1

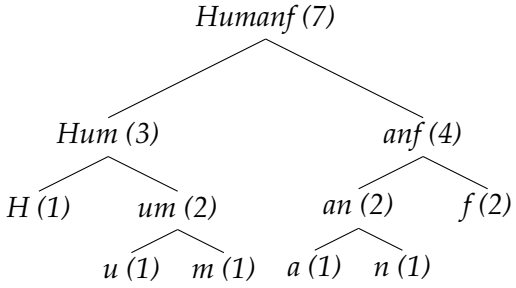


Tabelle 1: Häufigkeit

Zeichen	H	u	f	m	a	n
Anzahl	1	1	2	1	1	1

Nun weist man den Blättern Bitcodes zu. Hierfür geht man den Baum von der Wurzel bis zum entsprechenden Blatt ab. Wann immer man hierbei in einen linken Teilbaum geht, fügt man dem Code eine 0 an, wenn man in einen rechten Teilbaum geht eine 1. Die dadurch entstehenden Codes sind eindeutig, da Knoten keine Informationen tragen. Man spricht hierbei auch von einem präfixfreien Baum, da hier kein Bitcode der Anfang eines anderen sein kann. Deshalb benötigt man auch keine Trennzeichen im kodierten Text [3].

[3] http://www.zotteljedi.de/huffman/ (31.1.17)

3 Lösungsfindung

Zu Beginn des Projekts war es erst einmal notwendig, das Projekt in mehrere Abschnitte, die alle als eigene Funktionen umgesetzt werden können, zu unterteilen. Jeder Abschnitt hat ein klar definiertes In- und Output. Dies hilft um klar abgesteckte Aufgabe unter den einzelnen Teammitgliedern gleichmäßig zu verteilen, einzelne Abschnitte unabhängig von anderen auf Funktion und Performance zu testen und Meilenstein für das Projekt zu setzen. Die Entscheidung fiel auf folgende Unterteilung:

i) Laden von Dateien (Input: .txt Datei; Output: Text in char-Array)

ii) Zählung der Buchstaben (Input: Text; Output: Häufigkeitstabelle)

iii) Erstellen eines Baumes (Input: Häufigkeitstabelle; Output: Baum)

iv) Zuordnung der Bitcodes (Input: Baum; Output: Dictionary)

v) Kodierung (Input: Text, Dictionary; Output: Kodierter Text)

vi) Speichern (Input: Kodierter Text; Output: .txt Datei)

vii) Dekodieren (Input: Kodierter Text; Output: Text in Reinform)

viii) Bonus: Baum ausgeben (Input: Baum)

Nun galt es, Das Genaue Aussehen des jeweiligen In- beziehungsweise Outputs festzulegen. Im Falle der Häufigkeitstabelle und des Dictionary war dies relativ einfach, da hier nur abwechselnd Zeichen und Häufigkeit/Gespeichert werden müssen. Lediglich die Länge der Bitcodes im Dictionary musste festgelegt werden. Die Entscheidung fiel hierbei letztendlich darauf 32Bit, beziehungsweise 4 Byte, zu verwenden, da dies Platz für ausreichend lange Bitcodes bietet, und da es die Standartgröße eines unserer Register ist. Deutlich schwieriger gestaltete sich jedoch die Festlegung des Speicherformats des Baumes und des kodierten Textes, da die nicht nur ein notweniger und nicht trivialer Schritt ist, sondern auch Teil der Aufgabenstellung. Deshalb wird im Folgenden näher auf die jeweiligen Möglichkeiten eingegangen.

3.1 Speicherformat der Baumstrucktur

Eine der Möglichkeiten zur Strukturierung ist eine verlinkte Repräsentation der einzelnen Knoten und ihrer Nachfolger. Es ist also Speicherplatz für 2n-1 Knoten erforderlich, von denen jeder zusätzlich zu den Daten zwei Pointer zu den Nachfolgern enthält. Modifizierungen im Baum lassen sich durch verändern der Pointer effizient lösen, die Traversierung ist aufgrund der Pointer aber langsamer.
Bei der sequenzielle Repräsentation werden die Knoten der Tiefe nach aufgelistet. Nach der Wurzel werden die Knoten der nächsten Tiefe, angefangen mit dem äußersten linken Knoten, gespeichert. Dabei werden alle Positionen einer Tiefe im Baum vollständig im Speicher repräsentiert, der erforderliche Speicherplatz entspricht demnach 2n-1, n

entspricht der Tiefe des Baumes. Der größte Vorteil ist die direkte Adressierbarkeit jedes einzelnen Knoten im Baum und Speichereffizienz für gut ausbalancierte Bäume, erfordert bei Unbalancierten hingegen sehr viel Speicherplatz und nutzt diesen ineffizient.
Die Preorder Repräsentation speichert die Daten in bekannter Reihenfolge ab, zuerst der Knoten selbst, anschließend den linken Teilbaum und zuletzt den Rechten. Der dazu verwendete Speicherplatz entspricht 2n-1 mit der Anzahl der Blätter n, dadurch ist die Repräsentation sehr speichereffizient. Die Traversierung jedoch ist langsam, da um den rechten Teilbaum zu erreichen der komplette linke Teilbaum durchlaufen werden muss.
Der im Algorithmus entstehende Huffman Code ist ein Binärbaum mit n Blättern, wobei n der Anzahl der unterschiedlichen auftretenden Zeichen entspricht. Die maximale Tiefe beträgt ebenfalls n und die Knoten im Baum enthalten keine Daten. Es muss nur das entsprechende AscII-Zeichen im Baum repräsentiert sein, da die absoluten Häufigkeiten für ver- und entschlüsseln nicht mehr relevant sind.
Da die Form und Tiefe des Huffmanbaums einzig von den Häufigkeiten abhängt und damit nicht vorhersehbar ist, könnte eine sequenzielle Repräsentation zu extremer und ineffizienter Speichernutzung führen. Während der Verschlüsselung sind keine Veränderungen am Baum nötig, daher ist die Preorder Repräsentation besser zur Speicherung geeignet als eine verlinkte Repräsentation, da ist sehr speichereffizienter ist und die Traversierungdauer für die Methode akzeptabel ist. Auch ist die schwierige Adressierbarkeit hier kein Thema, da nicht einzelne Knoten betrachtet werden müssen, sondern alle Knoten der Reihe nach. Deshalb fiel die Entscheidung auf dieses Speicherformt des Baumes.

A.A.Puntambekar, Data Structures with C. Technical Publications, 2008.
A.K.Sharma, Data Structure Using C. Pearson Education India, 2011.

3.2 Ausarbeitung eines Speicherformats

Nun galt es nur noch, sich für ein Speicherformat zu entscheiden. Wir haben uns, auch wenn die Aufgabe es erlaubt, gegen eine Speicherung des kodierten Textes in einem für Menschen lesbaren Format entschieden, auch wenn dies einfacher gewesen wäre, da uns eine bitweise Speicherung realitätsnäher erschien. Andernfalls könnte man es kaum als Komprimierung bezeichnen. Auch war kurz angedacht, dass man das Dictionary in einer eigenen Datei speichern könnte. Dieser Gedanke wurde jedoch schnell wieder verworfen, da dies das Projekt nicht wirklich vereinfachen würde, und lediglich den Umgang mit den gespeicherten Dateien verkomplizieren würde. Ein Problem das aufgetreten ist, ist die Tatsache, dass wir den kodierten Text nicht bitweise, sondern nur in „Paketen“ von 8, 16 oder 32 Bit abspeichern können. Dies bedeutet, dass unter Umständen am Ende einige Nullen mehr mit abgespeichert werden könnten, die der Decoder als Zeichen interpretieren könnte. Die Lösung hierfür liegt darin, dass wir die Anzahl der Bits auch mit Abspeichern, damit der Decoder die überschüssigen Bits einfach ignorieren kann. Daraus ergibt sich letztendlich, dass wir zuerst die Länge des Dictionarys, dann die Anzahl an Bits die der kodierte Text umfasst, dann das Dictionary,

wobei jedes Zeichen und jeder Code 32 Bit benötigt, und abschließend noch den kodierten Text, auch in Blöcken von 32 Bit, speichern.

Hier eine kurze Darstellung wie das Speicherformat nun assieht:
<Länge Dictionary: 8n > | <Anzahl Bits: k> | 2*n*(<Zeichen> <Code>) | ⌈k/32⌉*<Codeblöcke>

4 Dokumentation der Implementierung

Die Dokumentation wird genauso wie die Programmierung selbst in die in Punkt 3 (Lösungsfindung) erwähnten Unterpunkte aufgeteilt.

4.1 Punkt i) und vi): Laden und Speichern

4.2 Punkt ii): Zählung der Buchstaben

Diese Assemblerfunktion hat die Aufgabe eine Tabelle anzufertigen, welche alle in einem Characterfeld vorkommenden Character, und ihre häufigkeit, in dem Characterfeld abspeichert.

4.2.1 InputOutput

Diese Assemblerfunktion nimmt als Parameter:

- Eine Adresse die zu einem Characterfeld zeigt in R0.

Diese Assemblerfunktion gibt zurück:

- Eine Adresse die zu der Tabelle zeigt in R0.
- Die Anzahl an Zeilen der Tabelle in R1.

4.2.2 Besonderheiten

Die Tabelle welche diese Funktion erstellt wird, im gegensatz zum C standard, nicht im Stack von niedriger Adressnummer zu höhere Adressnummer angelegt, sondern genau andersrum. Die Tabelle beginnt bei einer höheren Adressnummer, sprich üntenïm Stack und baut sich nach öben"hin auf.Im gegensatz dazu wird ein Feld in C von öben"nach üntenäufgebaut. Das hat damit zu tun dass die Informationen der Tabelle auf den Stack gepusht werden, anstatt wie normalerweise den Stackpointer zu verschieben und den dadurch bereitgestellten Raum zu nutzen. Dadurch wird relativ viel rechenarbeit gespart, da man den Stack als ein dynamisches Feld verwendet.
!ACHTUNG! Nach der verwendung dieser funktion wird der Stackpointer wieder ünter"die Tabelle verschoben. Das versteht der Computer so dass der bereich im Arbeitsspeicher wieder freigegeben wird. Um die Daten in der Tabelle nicht zu verlieren muss man danach den stackpointer auf Adresse der Tabelle - (Anzahl der zeilen in der Tabelle*8) setzen.

4.2.3 Funktionsweise

Diese Assemblerfunktion erstellt die Tabelle indem sie die Informationen in der Tabelle auf den Stack pusht.
Da das abspeichern immer in einer neuen Unterfunktion passiert, und natürlich dafür register gepusht werden wodurch sich der Stackpointer verschiebt, muss man den stackpointer dazu passend verschieben und dann als Startpunkt für die Tabelle abspeichern,damit man keine Daten überschreibt.
Da LDR 2 Bytes lädt, werden zwei Bitmasken erstellt. Einmal 0b0000000011111111 um auf den ersten Byte zuzugreifen und 0b1111111100000000 um auf den zweiten Byte zuzugreifen.
Abgespeichert werden die character indem man die ebreits gespeicherten character durchgeht und nachsieht ob dieser Character bereits einmal gespeichert wurde.
Falls das der Fall ist, dann wird bei dem dazugehörigen Eintrag die Häufigkeit nur um eins erhöht.
Falls das noch nicht der Fall war, dann wird der Character ans Ende der Tabelle gesteckt und bekommt 8 Byte Platz zugewiesen. Vier Byte für den Character und vier Byte für die häufigkeit.

4.3 Punkt iii): Erstellen eines Baumes

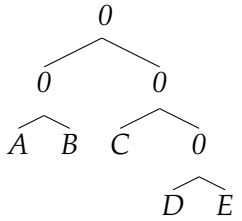
Die Methode benötigt eine Häufigkeitstabelle, deren Länge und einen Pointer zu allozierten Speicherplatz für die Rückgabe des Baumes als Parameter. Sie erstellt auf Basis der Häufigkeitsverteilung einen Huffman Baum und gibt die im Baum enthaltenen Character PreOrder über den mitgegebenen Speicher zurück. Die Funktionalität wird in 3 Untermethoden aufgeteilt, die Register r0-r2 enthalten dabei immer die mitgegebenen Parameter.
Zuerst werden die Einträge in der Häufigkeitstabelle per Insertionsort sortiert. R3 enthält die Anzahl der schon sortierten Elemente, r9 einen Pointer auf das erste unsortierte Element. R5 und r6 speichert den kleinsten Wert und einen Pointer zum zugehörigen Element der aktuellen Iteration, r7 und r8 den Pointer zum nächsten zu vergleichenden Element und die Position dessen in der Tabelle.
Für der Erstellung des Hoffmanbaums werden in _createTmpTree die Knoten und Blätter zu einem Baum vereint. Dazu werden bei n Blättern n-1 Knoten benötigt. Die Knoten werden temporär gespeichert, der writePointer in r3 zeigt auf die nächste freie Position. Pro Knoten werden fünf Integer reserviert, die ersten beiden für linken und rechten Nachfolger, den gesamtwert der Unterbäume und zuletzt ob der linke bzw. rechte Nachfolger ein Blatt ist. Im Fall eines Blattes wird statt eines Pointers der Character gespeichert. Am Anfang jeder Iteration wird das nächste Blatt, welches durch das Sortieren immer den niedrigsten Wert hat, Pointer in r6, geladen, sofern die Anzahl schon integrierten Blätter in r10 kleiner ist als die Gesamtzahl der Blätter. Es wird mit dem ersten Knoten verglichen, ein solcher existiert, wenn der Pointer in r8 zum ersten nicht integrierten Knoten ungleich dem writePointer ist. Das niedrigste Blatt/Knoten wird in den neuen Knoten eingefügt und der Wert dem Wert des Knotens hinzugefügt.

r7 entscheidet ob als linker oder rechter Nachfolger und wird anschließend invertiert, wurde ein Blatt eingefügt wird die Position markiert. Es wird entweder die Zahl der integrierten Blätter oder der Pointer zum ersten nicht integrierten Knoten angepasst. Sofern das Element als rechter Nachfolger eingefügt wurde, wird die Gesamtzahl der erstellten Knoten in r9 erhöht und die Schleife beendet sofern diese n-1 erreicht, zudem wird der writePointer um ein Element weiter gesetzt. Am Ende wird der writePointer in r4 verschoben um diesen als Pointer auf die Wurzel weiterzugeben.
In _createFinalTree wird der temporäre Baum PreOrder durchlaufen, um das Resultat zu erstellen. R3 beinhaltet den Pointer auf den nächsten freien Speicherposition im Resultat und wird nach jedem Eintrag entsprechend angepasst. Angefangen mit der Wurzel wird der aktuelle Knoten als Pointer in r4 gespeichert. Ein unbearbeiteter Knoten wird zunächst durch 0 im finalen Baum repräsentiert, bevor der linke Unterbaum betrachtet wird. Sofern der linke Nachfolger kein Blatt ist wird dieser zum aktiven Knoten und der aktuelle auf den Stack gelegt. Ist der linke Nachfolger ein Blatt wird der Character des Blattes im finalen Baum gespeichert, die Anzahl der übrigen Elemente in r5 um eins verringert. Fortgefahren wird mit dem rechten Nachfolger und die restlichen Knoten um eins verringert. Sofer dieser ein Knoten ist wird er zum aktiven Knoten und als unbearbeiteter Knoten betrachtet. Ist der rechte Nachfolger ein Blatt, wird entsprechend wie bei einem linken verfahren, allerdings wird die Methode beendet, sofern alle Knoten abgearbeitet sind oder zum obersten auf dem Stack liegenden Knoten zurückgegangen und dessen rechter Teilbaum betrachtet.

4.4 Punkt iv): Zweisen von Bitcodes

Input: Pointer zum Baum in r0, Pointer zur reservierten Speicherplatz für die Tabelle in r1
Output: -1 im Fehlerfall

Nun, nachdem der Baum erstellt wurde, müssen den Blättern des Baumes, also den zu kodierenden Zeichen, Binärcodes zugewiesen werde. Die Schwierigkeit besteht hierbei nicht in der Menge des zu schreibenden Quellcodes, sondern eher darin, einen passenden Algorithmus zu finden. Erst unter Betrachtung eines Beispiels wird klar, wie die Methode auszusehen hat.
Gespeichert wird dieser Baum wie folgt:



Wie hier dargestellt traversieren wir den Baum in der Reihenfolge, wie er gespeichert ist, also in Preorder-Reihenfolge. Dabei erstellen wir zu jedem Knoten und Blatt den dazugehörigen Binärcode. Wir beginnen bei der Wurzel mit einer 1. Diese 1 steht am Anfang jedes Binärcode. Sie markiert den Beginn des Codes und erscheint nicht in der

Tabelle 2:

Baum:	0	0	A	B	0	C	0	D	E	
Codes:	1	10	100	101	11	110	111	1110	1111	1
Schritte	lsl >	lsl >	+1 >	+1; lsr >	lsl >	+1 >	lsl >	+1 >	+1; 4*lsr >	

kodierten Form des Textes. Nun gehen wir die anderen Baumelemente nach und nach durch. Wenn wir an einen Knoten kommen, dargestellt durch die 0, wird eine 0 an das Ende unseres Binärcodes angefügt, sprich, es wird nach links geshiftet. Das ist durchaus logisch, weil wir nach jedem Knoten eine Ebene im Baum weiter nach unten gehen. Wenn wir zu einem Buchstaben kommen, speichern wir zuerst den Buchstaben und Code im Dictionary ab und erhöhen dann den Code um eins erhöht. Wenn wir an einem linken Blatt sind, endet der Code nun auf 1, und wir sind damit fertig. Sind wir an einem rechten Blatt, endet der Code nun auf 0. Das bedeutet wir haben diesen Teil des Baumes abgeschlossen und müssen nun wieder ein paar Ebenen nach oben. Deshalb streichen wir einfach alle am Ende stehenden 0 weg, sprich, wir shiften nach rechts. Der Algorithmus endet, wenn wir wieder an der Wurzel sind, der Binärcode also wieder 1 ist.

4.5 Punkt v): Kodieren

4.5.1 InputOutput

Diese Assemblerfunktion nimmt als Parameter:

- Die Adresse des Textes als R0
- Die Adresse des Zielarrays in R1
- Die Größe des Zielarrays in R2
- Die Adresse des Wörterbuchs in R3
- Die Anzahl der Zeichen im Wörterbuch in R4

Diese Assemblerfunktion gibt zurück:

- Bei Erfolg: Die Anzahl der verbrauchten Bytes des Zielarrays in R0
- Falls Das Zielarray nicht ausreicht -1 in R0

4.5.2 Besonderheiten

Das Zielarray beinhaltet am Ende:

- Die Größe des Wörterbuchs (in Byte)

- Die Menge an Bits im codierten Text
- Das Wörterbuch
- Den codierten Text

in genau dieser Reihenfolge.
Die Größe des Wörterbuchs ist im ersten Byte des Characterarrays gespeichert.
Die Menge an Bits im codierten Text ist im 5 Byte des Characterarrays gespeichert.
Das Wörterbuch ist im 9 Byte des Characterarrays gespeichert.
Der codierte Text ist im 9+Größe des Wörterbuchs Byte gespeichert.

4.5.3 Funktionsweise

Am Anfang wird die Länge des Wörterbuchs abgespeichert und der Speicherplatz für die Menge an Bits, welche der codierte Text am Ende haben wird, wird reserviert.
Daraufhin wird das Dictionary im Zielarray abgespeichert. Das passiert indem man einfach die Daten aus dem einen Feld in das andere kopiert. Falls das Wörterbuch nicht ganz reingepasst hat, wird -1 zurückgegeben und das Programm terminiert.
Als nächstes erstellen wir eine Bitmaske die an der 8ten stelle von rechts eine 1 hat. Dies wird unsere Bitmaske zum Speichern. Als nächstes wird der Bitcode bestimmt welcher auschließlich aus 0 besteht, außer an der ersten Position nach der ersten 1 im Bitcode des dazugehörigen Characters welcher gerade komprimiert wird. Diese Bitmaske wird verwendet um zu überprüfen ob der Bitcode an der stelle ein 0 oder eine 1 hat. Als nächstes shiften wir beide Bitmasken immer einmal nach rechts und überprüfen ob der Bitcode eine 1 oder eine 0 hat. Falls er eine 1 hat wird die Bitmaske zum speichern auf ein Register welches später zum Speichern verwendet wird, drauf addiert. Falls an der stelle aber eine 0 ist, shiften wir ganz normal weiter. Jedesmal nachdem wir unsere Bitmasken nach rechts shiften müssen wir überprüfen ob eine von ihnen 0 geworden ist, also ob die 1 welche eine Position angibt verschwunden ist. Falls unsere Speicherbitmaske 0 geworden ist, müssen wir unser Speicherregister im Zielarray abspeichern. Danach setzen wir sie wieder auf 0b10000000. Falls unsere Bitcodebitmaske 0 wird, sind wir mit dem jetzigen Character fertig und können nun ein neues Zeichen abspeichern. Jedesmal wenn wir unsere Bitmasken um eins nach rechts shiften, wird ein Zähler um 1 erhöht der überprüft wie viele Bits in unserem Zielarray abgespeichert werden. Dies ist nachher für das Dekodieren sehr wichtig.
Sollte es vorkommen dass unser Zielarray kein Speicherplatz mehr übrig hat, dann wird -1 zurückgegeben und das Programm terminiert.

4.6 Punkt vii): Dekodieren

Input: Pointer zum Text in r0, Pointer zum reservierten Speicherplatz für den Output in r1; Geschätzte länge in r2
Ouput: Länge des kodieren Textes im Erfolgsfall, sonst -1

Das Dekodieren gestaltet sich deutlich leichter als das Kodieren, da hier kein Dictionary erstellt werden muss. Dieses befindet sich relativ am Anfang des kodierten Textes. Deshalbt beginnt diese Methode damit erst die länge des Dictionarys, dann die Anzahl kodierter Bits, dann die Anfangs- und Endposition des Dictionary und schließlich den Anfang des eigentlichen kodierten Textes in eigene Register zu laden. Nun kann das eigentliche kodieren beginnen. Hierfür wird in jedem Durchlauf der folgenden Schleife mit Hilfe einer Bitmaske ein Bit aus dem gespeicherten Text geladen und dem Code hinzugefügt. Dann wird dieser Code mit einer Hilfsmethode namens _find im Dictionary gesucht. Diese geht einfach das Dictionary eintrag für eintrag durch und schaut ob der gesuchte Code enthalten ist. Wenn der Code gefunden wird wird der entsprechende Buchstabe im Outputarray gespeichert und der Code wird zurückgesetzt. Wird der Code nicht gefunden wird noch ein weiteres Bit hinten angefügt. Die Schleife endet wenn sie durch alle Bits durchgelaufen ist.

4.7 Punkt viii): Ausgeben des Baumes (Bonus)

Input: Baum

Das Ausgeben des Baumes auf der Konsole gestaltet sich auf Grund der Preorderspeicherung der Baumstruktur als deutlich schwieriger als anfangs gedacht. Deshalb muss der Baum zuerst in ein zweidimensionalen Array sortiert werden. Dafür wird der Baum von der Wurzel bis zum letzten Blatt traversiert und dabei jeweils die Entsprechende Ebene im Blick behalten. Dafür Wird sowohl eine Variable Level als auch der Zugehörige Bitcode benötigt. Der Code bestimmt, ähnlich wie beim Erstellen des Dictionarys, wie viele Ebenen zurückgesprungen werden müssen, wenn wir ein Blatt erreichen. Gleichzeitig werden Abstände und Linien zwischen den Knoten mitgespeichert. Die Breite der Abstände und die Länge der Linien sind jeweils von 2$\hat{}$(Tiefe-Level) abhängig. Dieser Faktor wird mit den Hilfsmethoden getDepth und powerOf bestimmt. Abschließend wird der Baum ausgegeben.

4.8 Zusammenfassen aller Methoden

Diese Funktion ist das Bindeglied zwischen all den anderen Unterfunktionen. Sie verwaltet die jeweiligen In- und Outputs und koordiniert den Platz im Arbeitsspeicher.
Sie nimmt als Parameter ein Characterarray zum Text, ein Characterarray für den Output und die länge des Characterarrays für den Output.
Als Output gibt sie entweder die Anzahl der verbrauchten Bytes zurück oder, für den Fall dass der Speicher nicht ausgereicht hat, -1.

5 Ergebnisse

6 Schluss

Abschließend kann nur nocheinmal auf die notwendigkeit von effinzienten Speicheralgorithmen in der heutigen Zeit hingewiesen werden. Da Speicherplatz noch immer kostspielig ist, können Algorithmen wie der von uns umgesetzt Huffmancode, mit einer bis zu 50%igen Koprmierungrate, eine deutliche Ersparnis bieten, auch wenn unsere Implementierung noch Luft zur Verbesserung offen lässt. So könnte beispielsweise ein Dictionary, das variable Längen der Bitfolgen erlaubt, ein deutlich besseres Ergebnis, besonders bei kurzen Texten, bieten. Auch könnte ein effizienterer Suchalgorithmus für das Suchen der Codes im Dictionary die Performanz verbessern. Doch insgesamt ist unsere Umsetzung schon relativ schnell und effizient gelungen.