LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 294 – Abgabe zu Aufgabe A404
Sommersemester 2023

Wajih Tarkhani Michael Eichhorn Leonhard Obkirchner

1 Einleitung

Die Datenkomprimierung spielt in verschiedenen Bereichen wie der Datenübertragung, Speicherung und Verarbeitung großer Datenmengen eine entscheidende Rolle. Angesichts steigender Datenmengen und begrenztem Speicherplatz ist es von großer Bedeutung, effiziente Methoden zur Reduzierung der Dateigröße zu entwickeln ohne dabei wesentliche Informationen zu verlieren. Durch die Anwendung von Kompressionsalgorithmen können große Datenmengen effizient codiert werden, um Platz zu sparen und die Übertragungsgeschwindigkeit zu verbessern.
Es gibt zwei grundlegende Arten von Datenkompression: verlustfreie und verlustbehaftete Kompression. Bei der verlustfreien Kompression werden die Daten so komprimiert, dass sie vollständig wiederhergestellt werden können, ohne dass Informationen verloren gehen. Dies ist besonders wichtig für Textdateien, Programmcode, Konfigurationsdateien etc... Auf der anderen Seite ermöglicht die verlustbehaftete Kompression, indem irrelevante oder weniger wichtige Informationen entfernt werden.

In diesem Projekt werden wir uns mit einem *verlustfreien* Algorithmus beschäftigen: *Huffman Algorithmus*. Der Algorithmus basiert auf einem Huffman-Baum, der aus den Zeichen und ihren Häufigkeiten aufgebaut wird. Durch die Zuweisung von *Binärcodes* zu den Zeichen wird die Effizienz der Kompression erreicht. Der Code enthält Funktionen zum **Codieren** und **Dekodieren** von Daten mithilfe des Huffman-Baums.

```
void huffman_encode(size_t len, const char[len] data)
```

```
void huffman_decode(size_t len, const char[len] data)
```

Diese nehmen vom Benutzer spezifizierte Eingabestring bzw. das kodierte Datum `data` der Länge `len`. Bei `huffman_encode` wird die Häufigkeitsanalyse durchgeführt, danach soll den Huffmanbaum konstruiert werden, um mit den einzelnen Bits an den Kanten die Eingabe binär zu kodieren. Bei `huffman_decode` wird der kodierte String mit dem kreierten Baum (oder mit dem erzeugten Dictionary) dekodiert.

Mit diesem Vorwissen mit dem Bezug zu der Aufgabenstellung kann sich nun in den folgenden Kapiteln mit Lösungsansätzen auseinandergesetzt, diese daraufhin auf ihre Korrektheit analysiert und anschließen auf ihre Performanz getestet werden.

2 Lösungsansatz

Bevor wir uns mit der eigentlichen Implementierung auseinandersetzten können, müssen zunächst die Rahmenbedingungen geklärt werden. Dafür ist es zwangsweise notwendig die Frage zu beantworten, welche Eingaben der Nutzer tätigen bzw. nicht tätigen darf. Laut Aufgabestellung ist es erwartet vom Nutzer eine Eingabedatei zur Verfügung zu stellen und mit `-o` eine Ausgabedatei zu wählen oder zu erstellen. Und das gilt für die Kodierung und die Dekodierung, der einzige unterschied ist ,dass man bei der Dekodierung `-d` am Anfang hinzufügt.
Mit diesem definierten Rahmen haben wir die angegebenen Funktionssignaturen geändert, und zwar zwei neue Argumente hinzugefügt: `outputFile` und `to_print`

```
void huffman_encode(size_t len, const char data[len], FILE* outputFile,
    int to_print)

void huffman_decode(size_t len, const char data[len], FILE* outputFile,
    int to_print)
```

2.1 Lösungsansatz von Encode

Zunächst beschäftigen wir uns mit Encode. Wir betrachten dazu ein einfaches Beispiel mithilfe des Wortes *ABRAKADABRAB*.

Ein gängigster Ansatz ist mit der Häufigkeitsanalyse anzufangen, indem wir die Häufigkeit der einzelnen Symbole in den Eingabedaten bestimmen indem wir ein `int array` dafür erstellen.
In diesem Frequenz-Array wird für jeden Buchstaben im Buchstaben-Array seine Häufigkeit im Frequenz-Array erhöht. Dabei wird der *ASCII-Wert* des Buchstabens als Index im Frequenz-Array verwendet. Das bedeutet, dass das Frequenz-Array so groß sein muss, wie die Anzahl der möglichen *ASCII-Werte*. Die Häufigkeit der Buchstaben, die nicht im Buchstaben-Array vorkommen, bleibt bei **0**, da das Frequenz-Array zu Beginn mit Nullen gefüllt wurde.

	...	A	B	...	D	...	K	...	R	...
Index (ASCII)	...	97	98	...	100	...	107	...	114	...
Frequency	0	5	3	0	1	0	1	0	2	0

Abbildung 1: Frequenz Array

Und jetzt von diesem langen array kreieren wir *2 neue Arrays*: ein array für die **Buchstaben** ,die keine Nulle Frequenz haben, und ein array für die **Frequenzen** ,die **größer** Null sind.

1.

Characters	A	B	D	K	R

2.

Frequency	5	3	1	1	2

Im nächsten Schritt bauen wir eine MinHeap-Datenstruktur als ein Array auf, wobei jeder Knoten bzw. jedes Element im Array ein `MinHNode` ist. Ein `MinHNode` enthält einen Buchstaben und seine Häufigkeit sowie zwei Zeiger auf das linke bzw. das rechte Kind.

```
struct MinHNode{
  char character;
  unsigned freq;
  struct MinHNode* left;
  struct MinHNode* right;
};
```

1. Wir fügen alle Elemente in dem MinHeap array ohne die Reihenfolge zu beachten.

A	B	D	K	R
5	3	1	1	2

Abbildung 2: Initialisierung von dem MinHeap-Array

2. Wir führen jetzt `set_up_MinHeap` damit wir die MinHeap Invariante erstellen können.

K	D	R	B	A
1	1	2	3	5

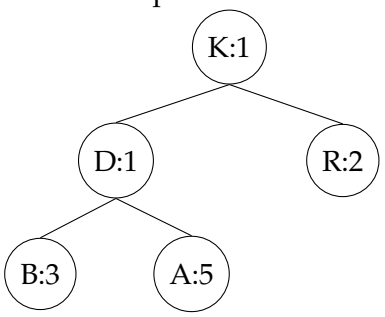
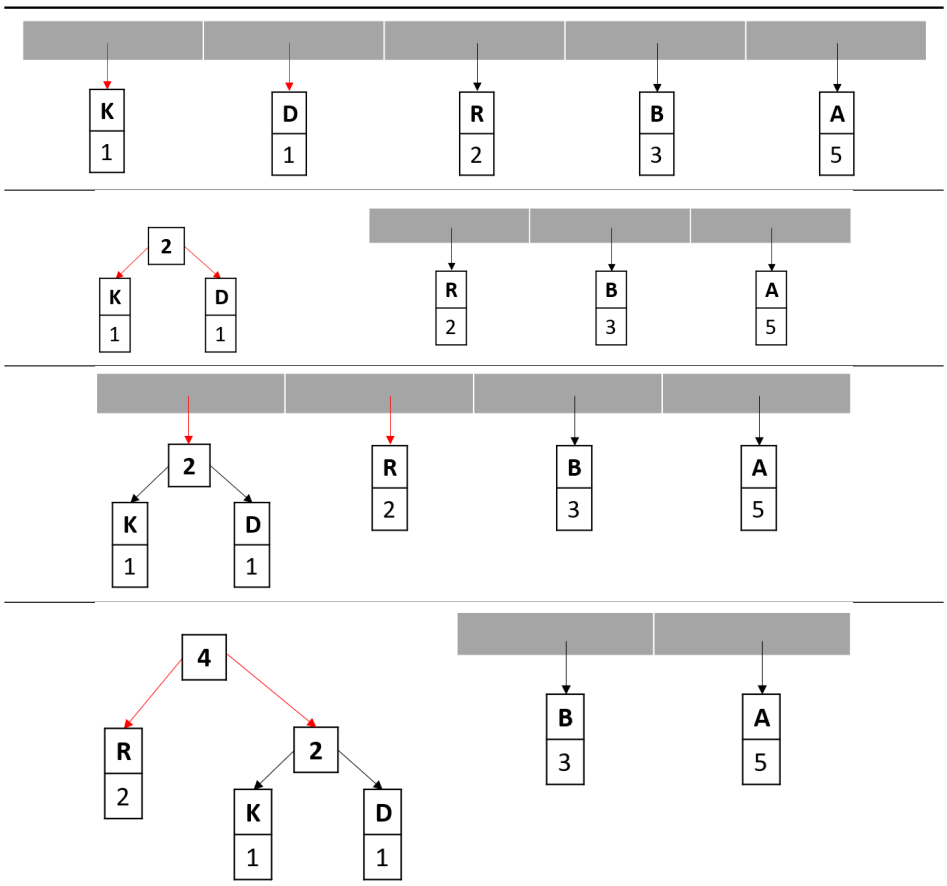


Abbildung 3: MinHeap : Array und Baum Darstellung

3. Unser Ziel ist es einen *präfix-freien** Baum (oder auch Huffman-Baum genannt) zu erstellen, mit dem wir ein sogenanntes *Dictionary* erstellen werden. Um den Huffman-Baum zu erstellen, können haben wir den fertigen MinHeap-Array als eine Prioritätswarteschlange betrachten, wo die Symbole nach ihrer Häufigkeit bereits sortiert sind. Dann können wir immer die zwei Symbole mit der niedrigsten Priorität aus der Warteschlange mit `pop_Min()` entfernen und einen neuen Knoten mit ihrer Summe als Priorität, das `&` als Character und die zwei Symbole als Kinder erstellen. Nach jedem `pop_Min()` rufen wir `sift_down_up()` auf, um die MinHeap Invariante wiederzustellen. Jetzt fügen wir diesen Knoten wieder in die Warteschlange mit `insert()` ein. Wir wiederholen diese Schritte bis nur noch ein Knoten vorhanden ist. Dieser Knoten ist die Wurzel des Baumes.



*Ein präfixfreier Baum wird verwendet, um eine Methode der sogenannten präfixfreien Codierung zu implementieren. Bei einer präfixfreien Codierung werden den einzelnen Symbolen eines Alphabets (zum Beispiel Zeichen oder Wörter) Bitmuster zugeordnet, so dass keine Bitfolge den Anfang einer anderen Bitfolge darstellt.

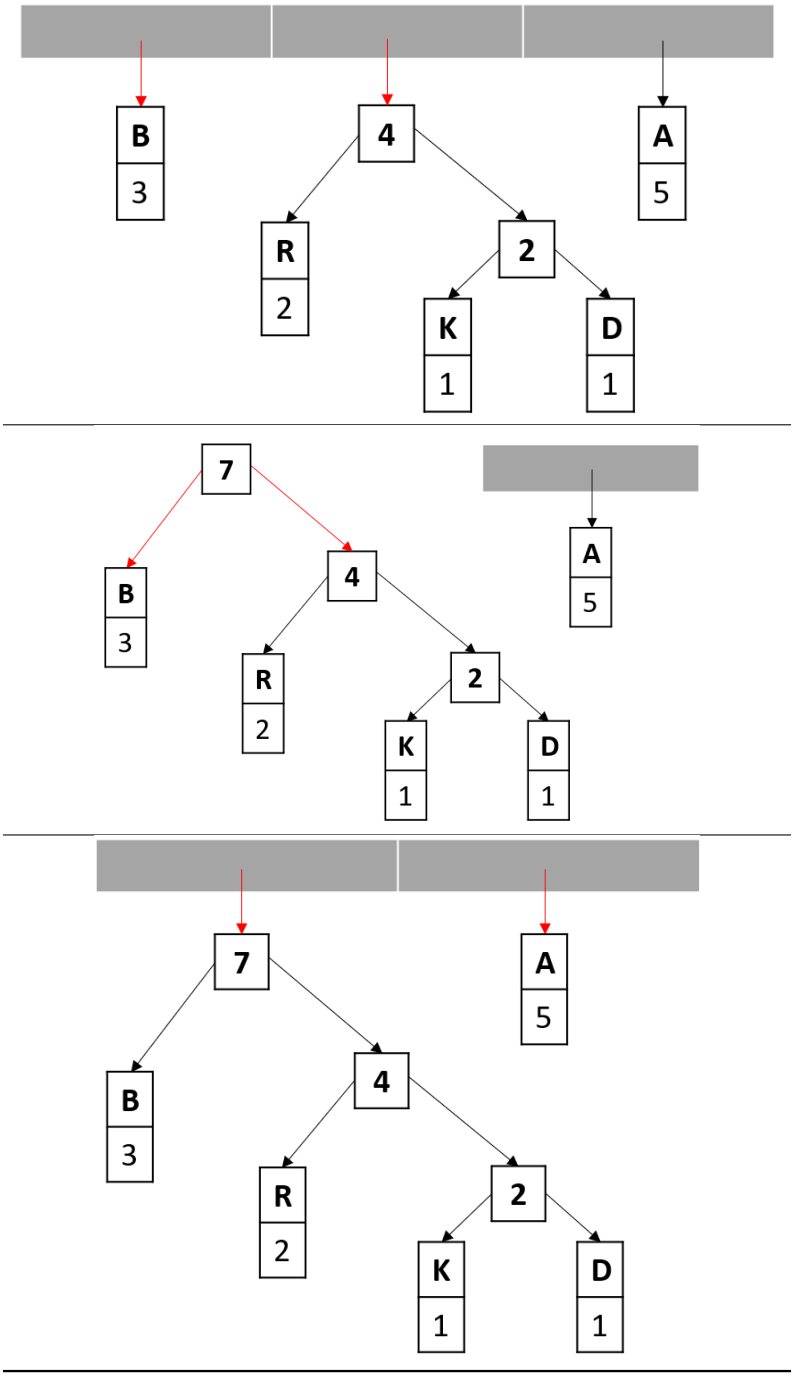
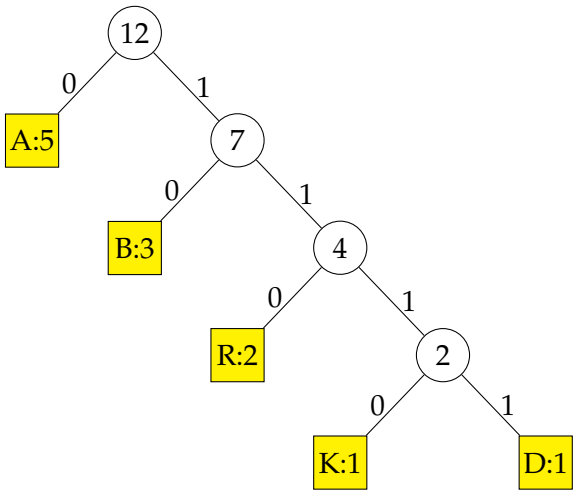


Abbildung 4: Die Folge von `pop_Min()` und `insert()` Schritt für Schritt dargestellt

Und somit bekommen wir den *präfix-freien* Baum, in welchem die häufigsten Symbole die kürzesten Codewörter zugewiesen bekommen.



4. Der fertige Baum kann dann verwendet werden, um mit den einzelnen Bits an den Kanten die Eingabe binär zu kodieren. Dazu erstellt man das *Dictionary*:

Symbol	Kodierung
A	0
B	10
R	110
K	1110
D	1111

5. Das gewählte Beispiel wird dann mithilfe der Tabelle buchstabenweise kodiert:

ABRAKADABRAB = 0 10 110 0 1110 0 1111 0 10 110 0 10

Anstatt $12 \cdot 8 = 96$ Bit zu benötigen, kann das Eingabewort mit $1 + 2 + 3 + 1 + 4 + 1 + 4 + 1 + 2 + 3 + 1 + 2 = 25$ Bit kodiert werden. Zusätzlich zu diesen 25 Bit muss auch der Baum als Datenstruktur gespeichert werden, aber es ist klar, dass die Huffmankodierung bei längeren Texten eine hohe Kompressionsrate erzielen kann.

2.2 Lösungansatz von Decode

Um die kodierten Daten zu dekodieren, gibt es zwei Ansätze: einen mit einem Dictionary und einen mit einem Baum. Der Vorteil des Ansatzes mit einem Dictionary ist, dass er schneller sein kann, da er weniger Schritte benötigt, um ein Zeichen zu finden. Der Nachteil ist, dass er mehr Speicherplatz benötigt, um das Dictionary zu

speichern oder zu übertragen, und dass er möglicherweise nicht funktioniert, wenn die Kodierung nicht `präfix-frei` ist. Der Vorteil des Ansatzes mit einem Baum ist, dass er weniger Speicherplatz benötigt, um den Baum zu speichern oder zu übertragen, und dass es immer funktioniert, egal ob die Kodierung `präfixfrei` ist oder nicht. Dazu ist er noch einfach zu implementieren, im Gegensatz zum ersten Ansatz. Der Nachteil ist, dass er langsamer sein kann, da er mehr Schritte benötigt, um ein Zeichen zu finden. Wir haben uns entschieden, den zweiten Ansatz zu implementieren. Er funktioniert folgendermaßen:

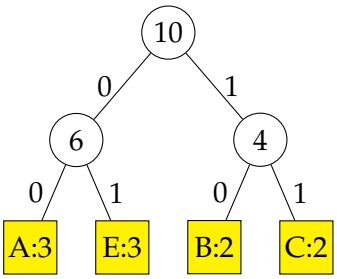
1. Man nimmt die kodierten Daten als eine Bitfolge und beginnt mit einem Zeiger auf die Wurzel des Baums.
2. Man geht durch jedes Bit in der Bitfolge und bewegt den Zeiger entsprechend nach links oder rechts im Baum. Wenn man auf einen Blattknoten stößt, hat man ein Zeichen gefunden, das man dekodieren will, und fügt es dem Ergebnis hinzu. Dann setzt man den Zeiger wieder auf die Wurzel.
3. Man wiederholt den Prozess, bis man alle Bits in der Bitfolge verarbeitet hat.

3 Korrektheit

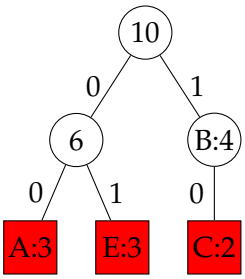
Da es sich bei der Huffman-Codierung um eine verlustfreie Codierung handelt, wird im Weiteren nur die Korrektheit der Implementierung betrachtet. Die Eingabe der Hauptfunktion `huffman_encode` wird nicht weiter beschränkt, solange diese ein String mit ASCII-Zeichen ist. Die Huffman-Codierung arbeitet zwar nicht auf den Werten der ASCII-Codierung, jedoch würde es den Rahmen des Projektes sprengen die Implementierung auf weitere Zeichen zu erweitern. Das Ergebnis der Hauptfunktion wird immer ein String sein, der sich aus dem Alphabet {' ', '0', '1'} zusammensetzt. Die zweite Hauptfunktion `huffman_decode` erwartet einen encodierten String aus Einsen und Nullen und wandelt diesen wieder in einen lesbaren String um. Ein zufälliger Binär-String kann nicht decodiert werden. Die korrekten Eingaben sind Vorraussetzung für eine erfolgreiche En- und Decodierung.

3.1 Wichtigkeit des präfix-freien Baumes

Die zentrale Eigenschaft des Huffman-Baumes ist, dass dieser immer *präfix-frei* ist. Das bedeutet für die Implementierung, dass die verschiedenen ASCII-Zeichen immer in den Blättern gespeichert werden müssen. Ist dies nicht der Fall, ist die Eindeutigkeit der Codierungen nicht mehr gegeben.



Baum 1: gültiger Huffman-Baum



Baum 2: ungültiger Huffman-Baum

Gegeben sind nun die Beispiele aus Baum 1 und 2, betrachtet wird der Teilstring 'ABC' zu den Huffman-Bäumen. Codiert man 'ABC' nach dem gültigen Baum ergibt sich das Dictionary {A: 00, E: 01, B: 10, C: 11} und folgender Code '001011'. Geht man nun zum ungültigen Baum über, fallen zwei Sachen auf. Einerseits steht ein ASCII-Zeichen in einer höheren Tree-Node und diese Tree-Node besitzt nur ein Kind. Es wird angenommen, dass die Node mit 'C' das linke Kind der Node mit 'B' ist. Daraus ergibt sich das Dictionary {A: 00, E: 01, B: 1, C: 10} und der Code '00110'. Würde der Code '00110' wieder decodiert werden, ergibt sich der String 'ABB' und je nach Implementierung kann ein Fehler geworfen werden. In diesem Beispiel erkennt man, dass die Codierung von 'B' ('1') ein Präfix der Codierung von 'C' ('10') ist. Daraus erschließt sich, dass jede Node mit einem gespeicherten ASCII-Zeichen ein Blatt sein muss, damit das Präfix-Problem nicht entsteht. Das impliziert, dass jeder Huffman-Baum ein voller Binär-Baum sein muss. Beim Decodieren wird nur über den encodierten String iteriert und der aktuelle Code-String mit dem Dictionary verglichen.

3.2 Analyse der Implementierung

Hier wird überprüft, ob der generierte Huffman-Baum die Eigenschaften aus dem vorherigen Kapitel erfüllt. Für die Erstellung des Baumes ist die Funktion buildHuffmanTree mit folgender Signatur zuständig:

```
struct MinHNode* buildHuffmanTree(char character[], int freq[], int
    size)
```

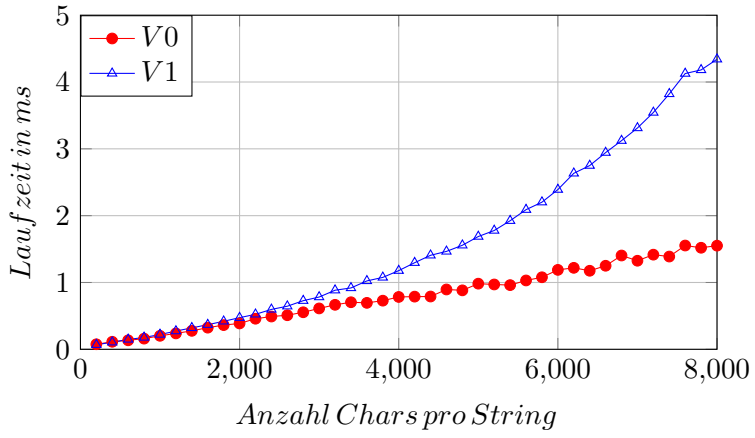
Der Funktion wird ein Array mit den unterschiedlichen ASCII-Zeichen und ein Array mit der jeweiligen Frequenz übergeben. Es wird jedes Zeichen mit zugehöriger Häufigkeit in einer Node gespeichert und einem MinHeap übergeben. Der Heap wird wie eine PriorityQueue benutzt. Über einen Pop-Befehl wird die Node mit der geringsten Frequenz entfernt und zurückgegeben. Entfernt man nun zwei Nodes mittels des Pop-Befehls und weist diese als Kinder einer neuen Node zu, kann man diese neue Eltern-Node wieder dem MinHeap hinzufügen. Hierbei ist es nicht möglich, dass eine

schon vorhandene Node ein Parent einer anderen Node werden kann. Damit ist der Huffman-Baum automatisch ein voller Binärbaum und nur die Blätter enthalten die ASCII-Zeichen.
Um eine erfolgreiche Encodierung sicherzustellen, werden folgende Schritte überprüft. Nach der Erstellung des Huffman-Baumes wird mit einer simplen Rekursion über den Baum iteriert und falls die aktuelle Node ein Blatt ist, wird in einem Dictionary der zugehörige Pfad zum Blatt bzw. der Binär-Code gespeichert. Es wird danach nur noch über die Eingabe iteriert und die entsprechenden Codes der Zeichen in eine Ausgabedatei gespeichert.
Beim Decodieren wird wie beim Encodieren mithilfe der gespeicherten Daten wieder derselbe Huffman-Baum erstellt. Beim Durchlaufen des Codes wird je nach '0' oder '1' zum linken oder zum rechten Kind der aktuellen Node weitergegangen. Ist man bei einem Blatt angekommen, wird das gespeicherte ASCII-Zeichen ausgegeben und der Durchlauf beginnt wieder von vorne. Wenn der Huffman-Baum gültig ist, wird das Decodieren korrekt durchgeführt. Wird ein ungültiger Binär-String übergeben, gibt es dazu kein zugehöriges Dictionary und es wird bzw. kann nicht decodiert werden.

4 Performanzanalyse

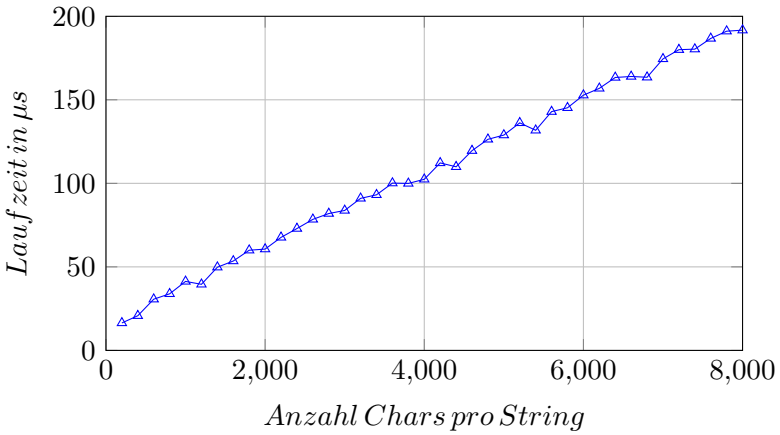
Bevor wir mit der Performanzanalyse beginnen, sollte die Testumgebung spezifiziert werden. Alle Tests wurden auf einem **AMD Ryzen 7 5800H** Prozessor, 3.20 GHz, 16GB Arbeitsspeicher durchgeführt. Kompiliert wurde mit GCC 11.3.0 mit der Compiler-Stufe **O3**. Für jeden Messpunkt wurde die durchschnittliche Laufzeit aus 600 Funktionsaufrufen berechnet. Als Eingabewerte dafür dienten jeweils zwei zufällig generierte Strings aus allen druckbaren ASCII-Zeichen, woraus wiederum der Durchschnitt gebildet wurde. Da hier zwei voneinander getrennte Funktionen Encode und Decode vorliegen, untersuchen wir diese auch in Bezug auf Performanz unabhängig.

4.1 Analyse von Encode



Auf dem oberen Diagramm sind die Laufzeiten der Hauptimplementierung V0 und der Referenzimplementierung V1 bei variierender Länge der Eingabe-Strings verbildlicht. Die Referenzimplementierung verwendet die selbe Funktion zur Erstellung eines Huffmanbaums wie V0, sie unterscheidet sich jedoch im Bilden des Dictionaries grundlegend.
Wie man erkennen kann, nimmt die Ausführungsdauer bei V1 fast exponentiell zu, wohingegen sie bei V0 nur linear ansteigt. Wie weitere Tests ergaben, beruhen ca. 80% der Laufzeit von V1 auf dem schlussendlichen Bilden des enkodierten Strings aus dem Dictionary, aus welchem Grund es Sinn macht diesen Teil zu optimieren, wie in V0 geschehen. Weil mit zufällig generierten Strings jeglicher Länge V0 schneller läuft, sollte diese Funktion bevorzugt werden.

4.2 Analyse von Decode



Da beide Implementierungen die selbe Decode Methode verwenden, kann auch nur eine Funktion getestet werden.
Mit der Länge des Strings nimmt auch die Ausführungszeit von Decode linear zu.

4.3 Interpertation der Performanzanalyse

Da beide Funktion Encode_V0 und Encode_V1 die selbe Funktion beim bilden des Huffman-Baumes verwenden, widmen wir uns in diesem Teil den Unterschideden im Erstellen des Dictionaries.
Der Ansatz der V1 sieht folgendermaßen aus: Das Dictionary wird als Array der Länge 256 erstellt, wobei der Index als das als Zahl dargestellte ASCII-Zeichen zu interpretieren ist und der Wert die binär kodierte Version dieses Zeichens. Folglich bedeutet das, dass bei jedem Schreiben eines enkodierten Chars in den Ausgabe-String durch das Dictionary iteriert werden muss, um den passenden Index zu finden, im schlechtesten Fall sind das 256 Vergleiche.
Auf der unteren Abbildung ist ein Beispiel für ein solches Dictionary gegeben.

Index	0	1	...	113	114	...	254	255
BinCode			...	1111010	1110101	...		

Bei V0 hingegen ist das Dictionary-Array nur so lang, wie viele unterschiedliche Chars in dem Eingabe-String auch tatsächlich existieren. Es gibt also keinen Index, bei welchem das Char eine Häufigkeit von null aufweist, was bei V1 passieren kann. Die Einträge sind ein Struct, das einerseits das Char enthält und andererseits das enkodierte Ebenbild dessen.

Index	0	1	...	47	48	...	88	89
Char		A	...	k	l	...	w	z
BinCode	1001010	1110111	...	011110	111101	...	1001000	100011

Dadurch ist leicht ersichtlich, dass es effizienter sein muss, über ein in den allermeisten Fällen kleineres Array zu iterieren, als über eines, das konstant 256 groß ist.

5 Zusammenfassung und Ausblick

Die Ergebnisse des Projekts zeigen, dass Huffman-Codierung eine effektive Methode zur Datenkompression ist, die den Speicherplatz reduziert und die Übertragungszeit beschleunigt. Die Kompressionsrate hängt von der Häufigkeitsverteilung der Eingabezeichen ab. Je ungleichmäßiger die Verteilung ist, desto höher ist die Kompressionsrate. Rückblickend auf die Vorteile des Algorithmus, kann man sagen ‚dass er eine optimale Methode zur Datenkompression ist. Außerdem ist er einfach zu implementieren und zu verstehen. Und wie jeder Algorithmus hat die Huffmankodierung auch nachteile. Diese sind:

- Die Effizienz von Huffman-Codierung hängt von der Größe der Eingabedatei und der Anzahl der eindeutigen Zeichen ab. Je größer die Datei und je weniger Zeichen, desto effizienter ist die Kodierung.
- Es erfordert einen zusätzlichen Speicherplatz für den Huffman-Baum oder die Tabelle, um die Kodierung zu speichern und wiederherzustellen.
- Es ist nicht geeignet für Daten mit einer gleichmäßigen Häufigkeitsverteilung oder einer geringen Anzahl von Zeichen.

Literatur