Huffmankodierung (A404)

Referenten:

Wajih Tarkhani - Michael Eichhorn - Leonhard Obkirchner

Einleitung:

- **Datenkomprimierung ist sehr wichtig**
  - Notwendigkeit effizienter Methoden zur Reduzierung der Dateigröße
  - Anwendung von Kompressionsalgorithmen zur effizienten Codierung großer Datenmengen

- **Zwei grundlegende Arten von Datenkompression: verlustfreie und verlustbehaftete Kompression.**

- **Für uns wichtig ist Huffman-Algorithmus: Ein verlustfreier Algorithmus**

Einleitung

- **Signatur von Encode und Decode:**

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

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

- **Parameter:**
  - len : Länge von dem gegebenen Datum
  - data: Zeiger auf die zu kodierenden Daten

Lösungansatz:

- **Inhalt:**

    1. **Huffman Encode**
    2. **Huffman Decode**

Encode

- **Umsetzung:** *Beispiel : **ABRAKADABRAB***

1. **Häufigkeitsanalyse durchführen**

Characters	A	B	D	K	R
Frequency	5	3	1	1	2

2. **Min-Heap mit allen Zeichen und ihre Häufigkeit erstellen**

K	D	R	B	A
1	1	2	3	5

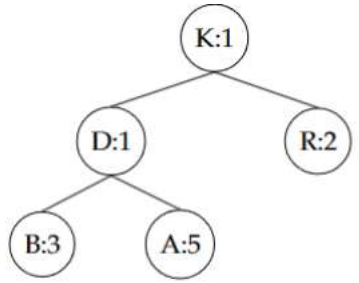
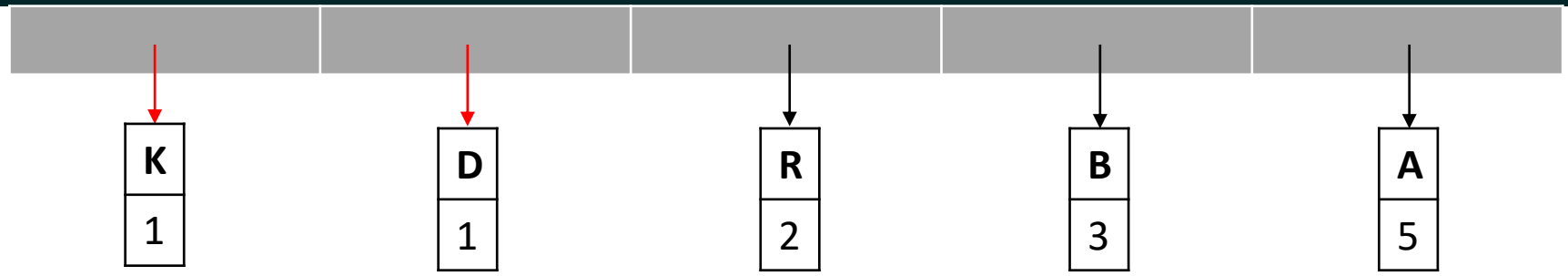


Abbildung 3: MinHeap : Array und Baum Darstellung

3. **Erstellung des Huffman Baums anhand des Min-Heaps**

Encode

Der erstellte Min-Heap als Array:



Huffman Baum erstellen:

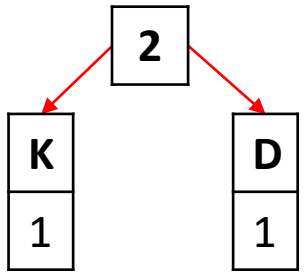
1. Extrahiere **zwei Knoten** mit der **minimalen Häufigkeit** aus dem Min-Heap
2. Erstelle einen **neuen Knoten**
3. Füge diesen Knoten zum Min-Heap hinzu

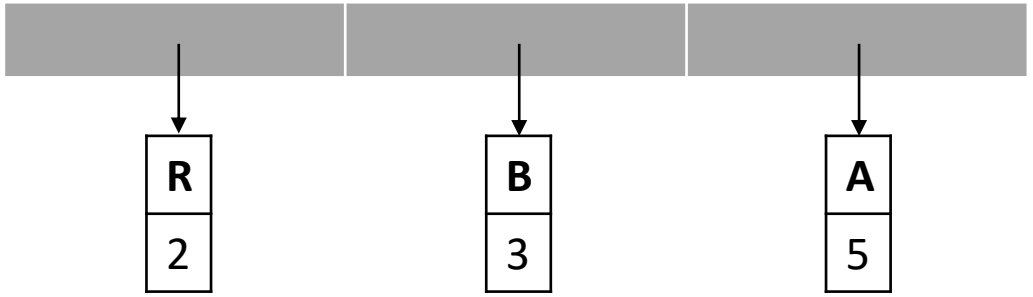
4. Wiederhole Schritte 1 bis 3 bis **der Heap nur noch einen Knoten** enthält.
5. Der **verbleibende Knoten** ist der Wurzelknoten und der **Baum ist fertig**.

Encode

- **pop_Min()**
- **pop_Min()**

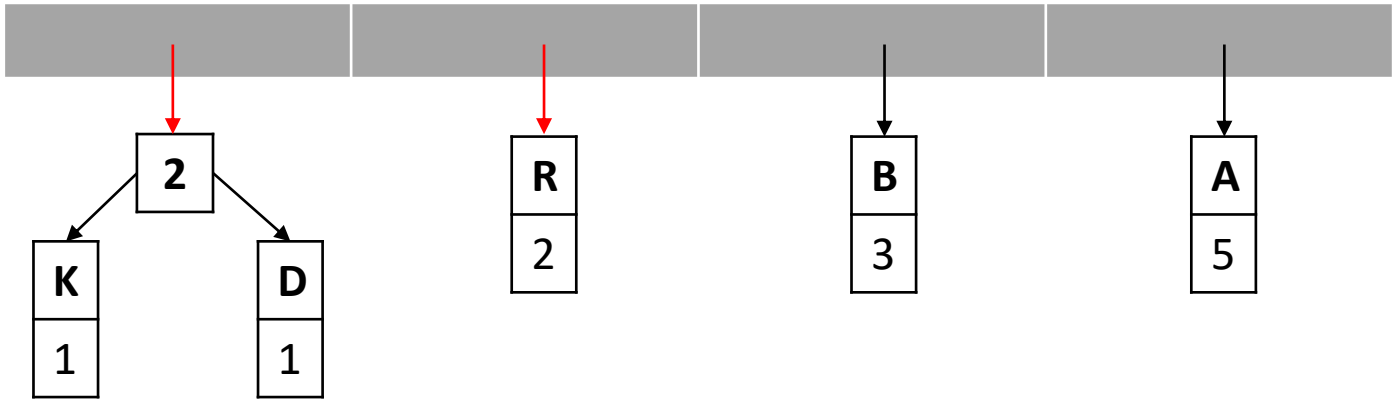
neuen Knoten erstellen





Encode

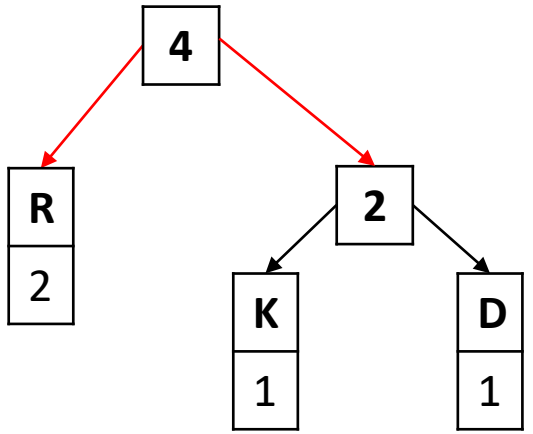
- **insert()**

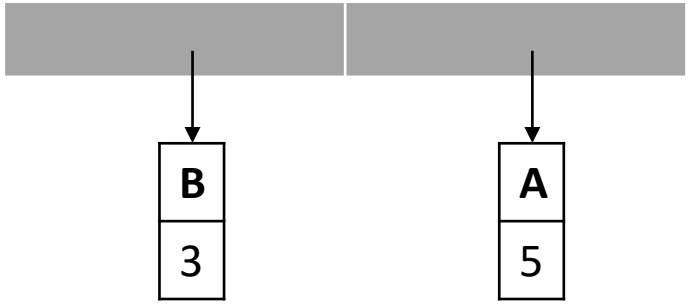


Encode

- **pop_Min()**
- **pop_Min()**

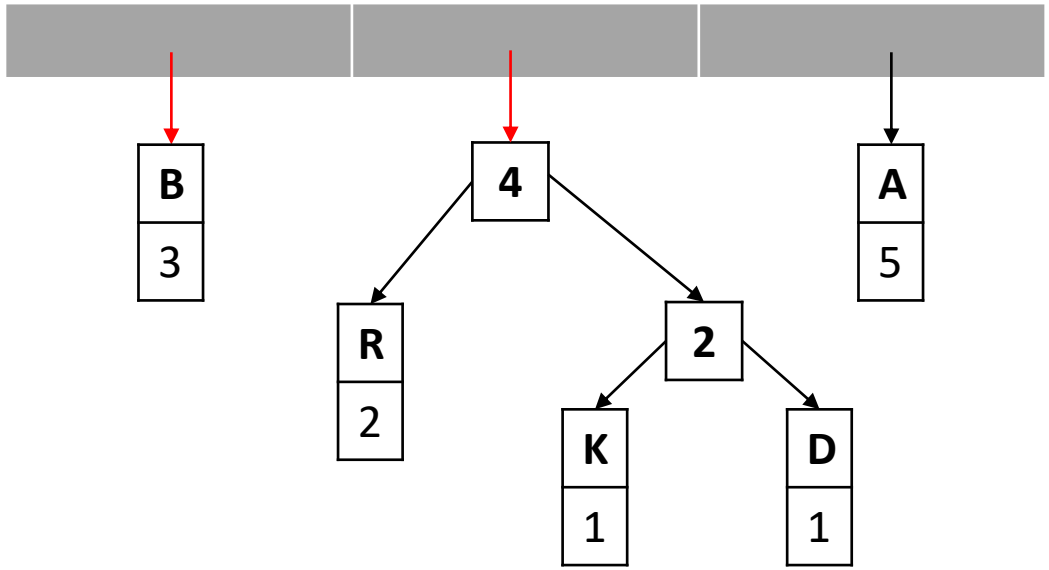
neuen Knoten erstellen





Encode

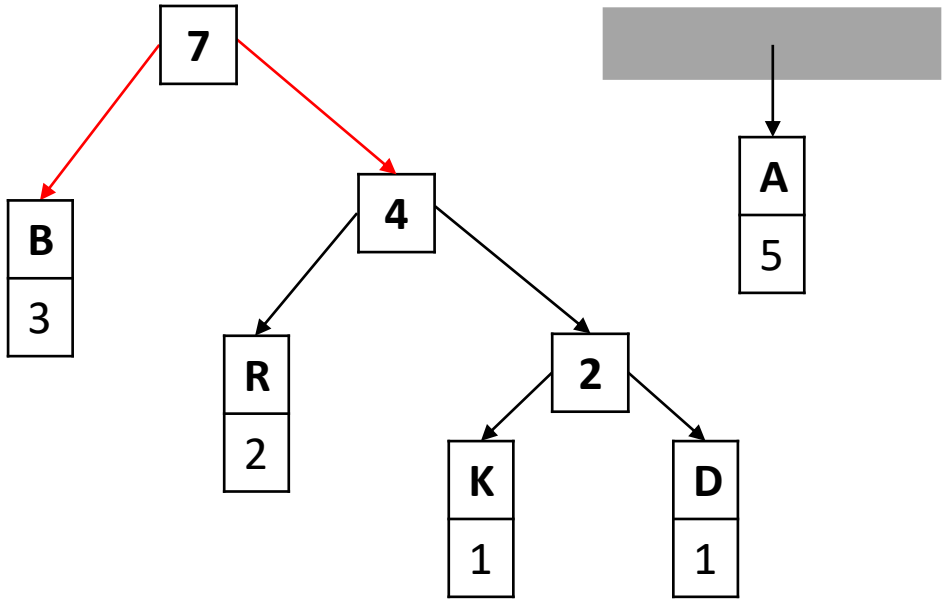
- **insert()**



Encode

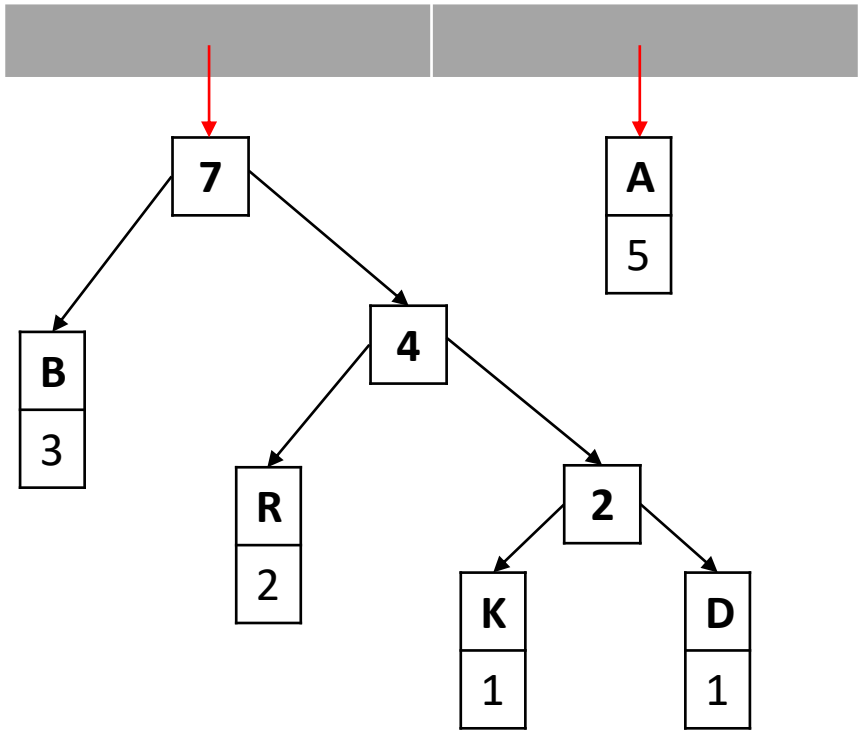
- **pop_Min()**
- **pop_Min()**

neuen Knoten erstellen



Encode

- **insert()**



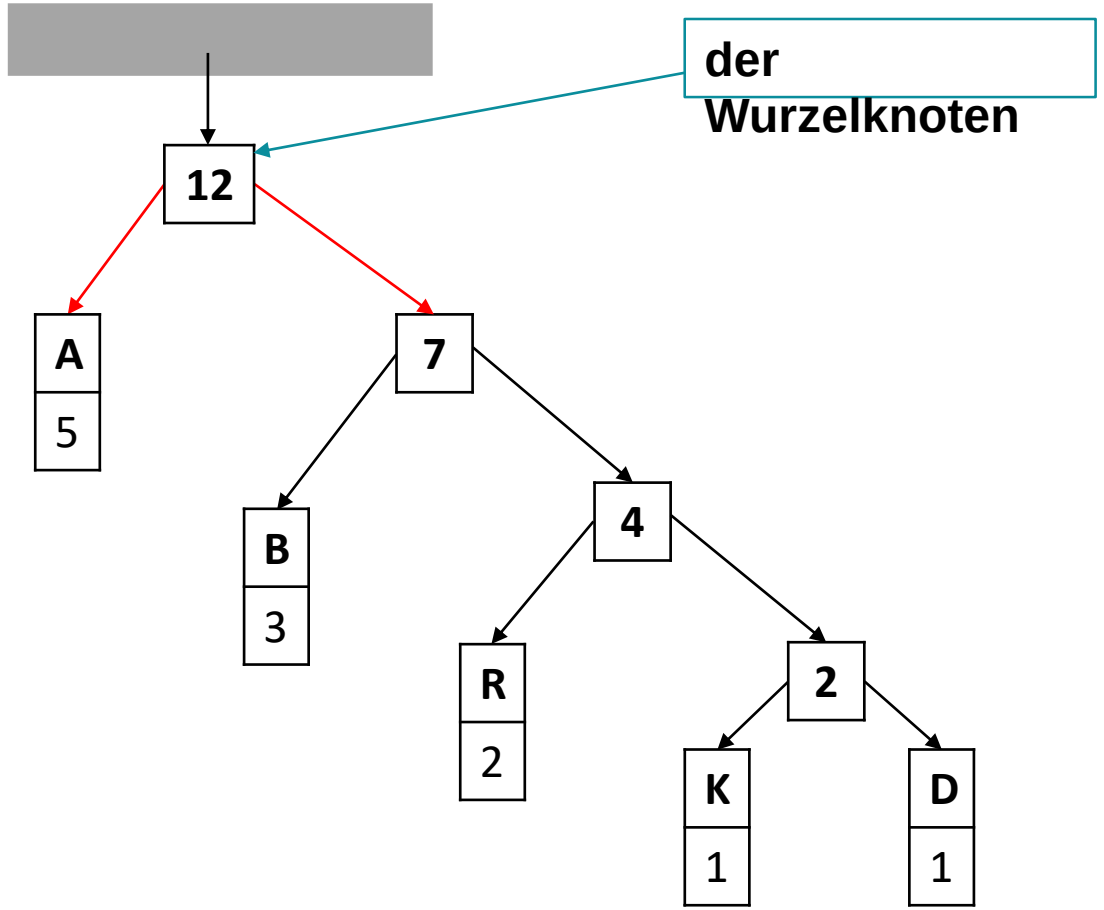
Encode

- **pop_Min()**
- **pop_Min()**

➡ **neuen Knoten erstellen**

- **insert()**

➢ **Huffman Baum ist fertig**



Encode

- Das **Dictionary** mit dem fertigen Baum erstellen
  - Jede **linke Kante** ist **0** zugewiesen
  - Jede **rechte Kante** ist **1** zugewiesen
  - Code generieren, indem man den **Pfad** von der **Wurzel zum Blatt** folgt.

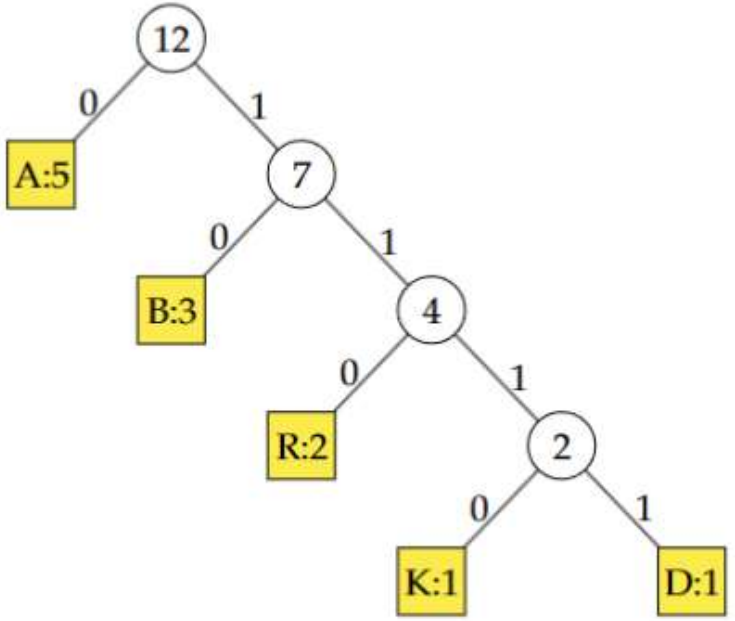


Abbildung 1: Huffman Baum

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

Abbildung 2: Dictionary

❖ **Ergebnis**: *ABRAKADABRAB* = **0 10 110 0 1110 0 1111 0 10 110 0 10**

Decode

- Zwei Ansätze möglich:
  - Mit dem Dictionary
  - **Mit dem Huffman Baum**

Umsetzung:

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
3. Stoßt man auf einen **Blattknoten**, fügt es dem Ergebnis hinzu
4. Wiederhole den Prozess bis alle Bits verarbeitet sind

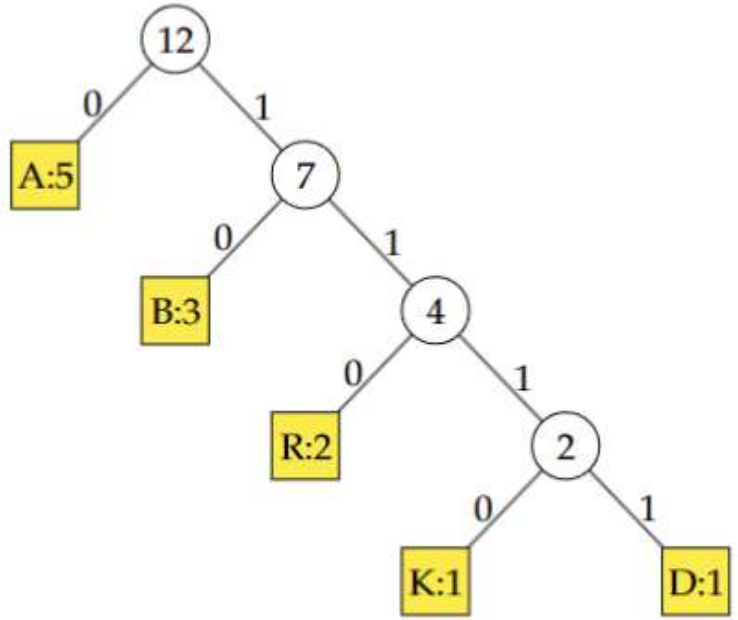


Abbildung 1: Huffman Baum

Korrektheit

- Eingaben für die Hauptfunktionen?

  Encode:
  - Strings mit ASCII-Zeichen
  - Beliebige Länge

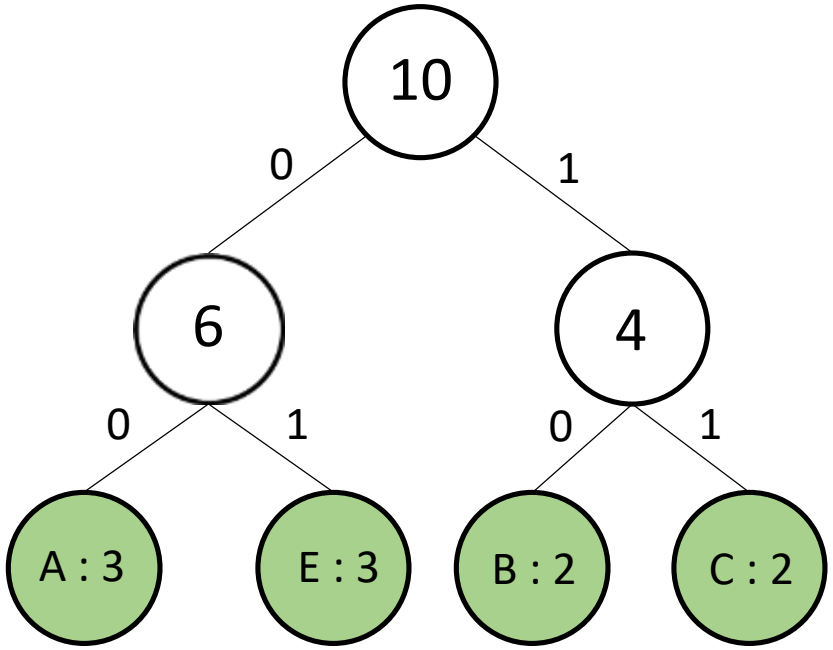
  Decode:
  - String aus 0 und 1
  - Wurde zuvor encodiert

- Ergebnisse der Hauptfunktionen?

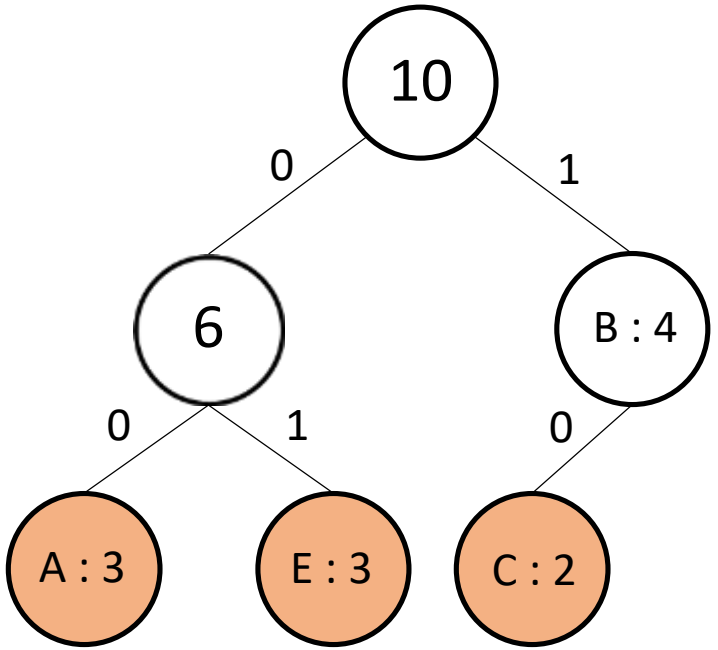
  Encode:
  - String aus 0 und 1

  Decode:
  - Decodierter String

Korrektheit



gültiger Huffman-Baum



ungültiger Huffman-Baum

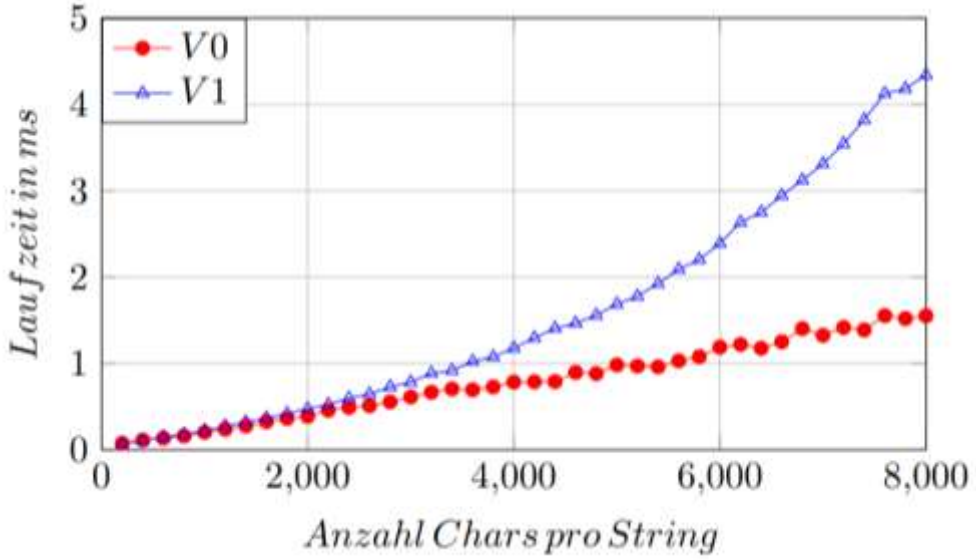
Korrektheit

- Umsetzung des präfix-freien Baumes
  - Min-Heap
    - Entfernen zweier Nodes
    - Einfügen als Kinder einer neuen Node
    - Parent-Node wieder einfügen
  - Aufbau des Baumes von unten
- Restliche Encodierung
  - Erstellen eines Dictionary
  - Iterieren über die Eingabe und Einfügen der Codierung

- Decodierung
  - Erstellen des Huffman-Baums
  - Iterieren über die Eingabe und Zuteilung von Code und Character

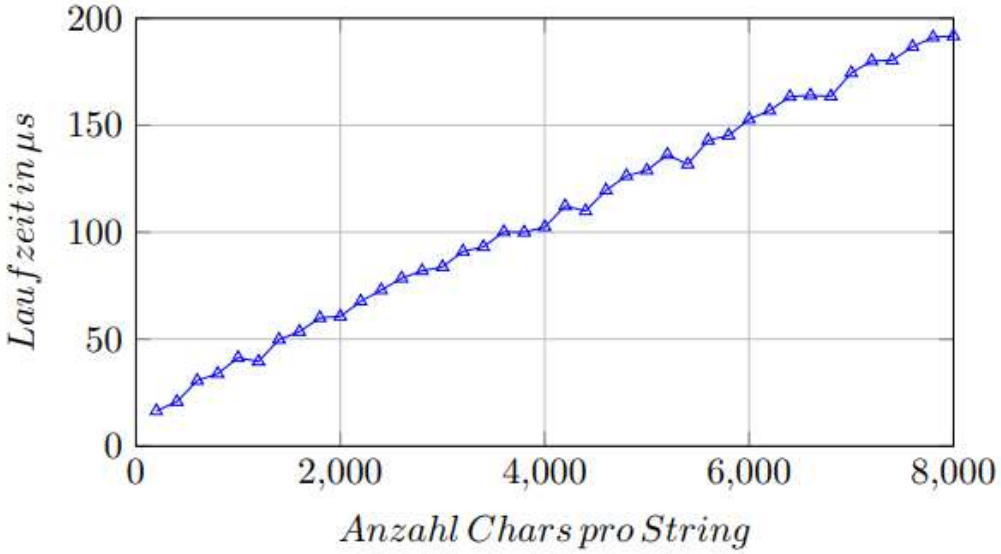
Performanzanalyse

Encode



- Referenzimplementierung V1 fast exponentielles Wachstum
- Hauptimplementierung V0 linear -> besser

Decode



- Für V0 und V1 gleiche Decode-Methode
- Lineares Wachstum

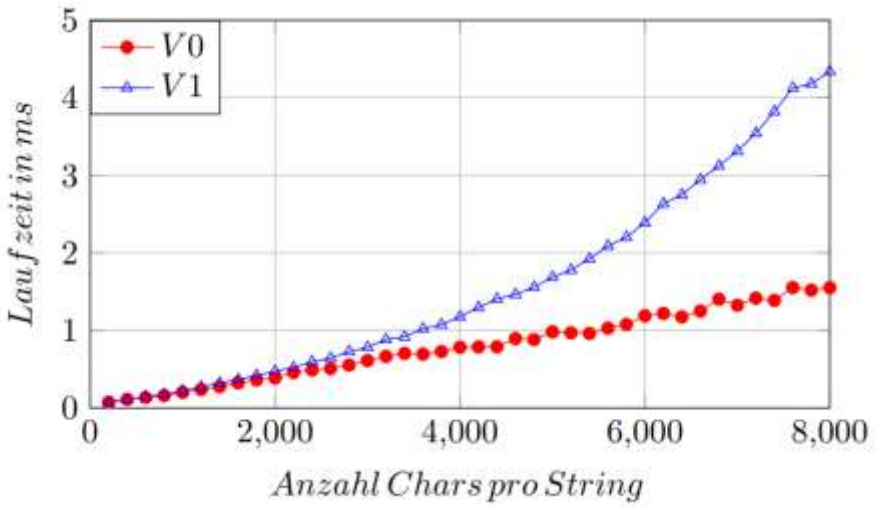
Performanzanalyse

Encode_V1

- Zwischen V0 und V1 nur Unterschiede im Erstellen des Dictionaries
- V1 mit Index als Referenz für Char
- Nachteil: viele überflüssige Einträge

(char)113 = "q" (char)114 = "r"

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

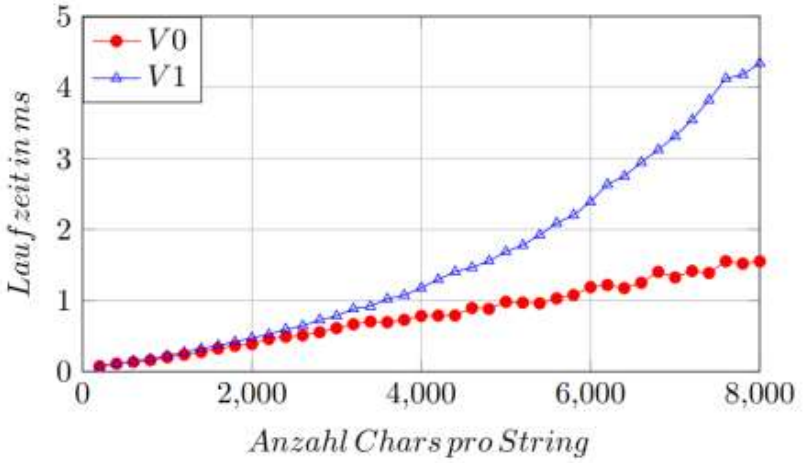


Performanzanalyse

Encode_V0

- Alle Einträge als Struct mit Char und dessen Kodierung
- Jeder Index gefüllt mit Char aus Eingabe-String
  -> keine unnötige Iterierung über leere Einträge wie bei V1

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



Zusammenfassung

- **Huffman-Codierung ist eine effektive Methode zur Datenkompression.**

- **Vorteile:**
  - ✓ Verlustfreier Algorithmus
  - ✓ einfach zu implementieren und zu verstehen
  - ✓ optimale Methode zur Datenkompression

- **Nachteile:**
  - Die Effizienz von Huffman-Codierung hängt von der Größe der Eingabedatei
  - zusätzlichen Speicherplatz für den Huffman-Baum oder das Dictionary
  - nicht geeignet für Daten mit einer gleichmäßigen Häufigkeitsverteilung

Fragen?