Huffman Codes und Datenkompression

Ziel: Speichere Daten unter Verwendung von möglichst wenig Speicherplatz

Originalformat → komprimiertes Format

Kompression

- verlustfreie Datenkompression, d.h. Originaldaten können vollständig aus komprimierten Daten zurückgewonnen werden.
- verlustbehaftete Datenkompression:
  z.B Audiodaten (z.B. mp3 - Format)
  Videodaten

zum mp3-Format

Original → gefiltertes Original → mp3-Datei

↑ Filter, verlustbehaftet

↑ verlustfreie Kompression mit Huffman-Codes!

↗ Ersparnis c.a. 20%!

Im Folgenden: verlustfreie Kompression

intuitive Vorstellung: Komprimierung von Textdateien

Annahme: Datei besteht aus einzelnen

Zeichen aus Alphabet (Zeichensatz) C,

z.B: C = ASCII-Zeichensatz

Unicode- -

Java - -

(reservierte Worte
≙ 1 Zeichen)

Gängige Zeichensätze verwenden sogen.
Blockcodes (d.h. Repräsentation von
Zeichen als 0-1-Strings fester Länge)

z.B. ASCII: Länge 8

Unicode: Länge 16

weiteres Beispiel (vereinfacht):

a	⟶	000
b	⟶	001
c	⟶	010
d	⟶	011
e	⟶	100
f	⟶	101
g	⟶	110
␣	⟶	111

Blockcodes erlauben sehr einfache
Codierung und Decodierung

Bsp: f a ␣ g a g a ␣ ...

101 000 111 ...

kodierte Datei:

f	a	␣	g	a	g	a	␣	g	a	f	f	␣	f	e	g	e
101	000	111	110	000	110	000	111	110	000	101	101	111	101	100	110	100

Dekodieren: je 3 Bits ≙ Zeichen ∈ C

101 000 111 110 000 110 000 111 110 000 101 101 111 101 100 110 100

Grundidee der Kompression:

- Häufigkeitsanalyse der Zeichen c in den Originaldate:
- Konstruiere Code variabler Länge, in dem häufige Zeichen kurze Codeworte bekommen und seltene Zeichen lange Codeworte

Bsp: ein Code variabler Länge

a	⟶	0
b	⟶	1
c	⟶	10
d	⟶	11
e	⟶	100
f	⟶	101
g	⟶	110
␣	⟶	111

} Codeworte: jedem Zeichen aus C ist ein Codewort zugeordnet.

Kodierung:

f	a	␣	g	a	g	a	␣	g	a	f	f	␣	f	e	g	e
101	0	111	110	0	110	0	111	110	0	101	101	111	101	100	110	100

b d ?

weiteres Beispiel:

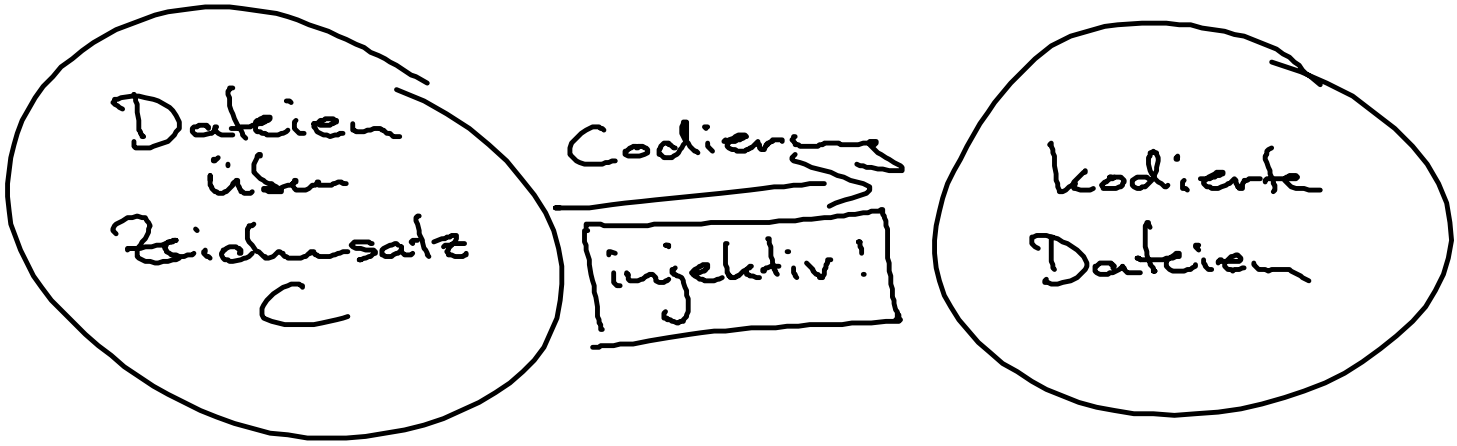
abba ≠ ag

0110 = 0110

Problem: nicht eindeutig dekodierbar!

Codes variabler Länge sind u.U. nicht eindeutig dekodierbar.

Def.: Ein Code heißt **eindeutig dekodierbar** :⇔ verschiedene Originaldateien führen zu verschiedenen kodierten Dateien



Eine wichtige Klasse von eindeutig dekodierbaren Codes sind die sogen. **Präfixcodes:**

Def: Ein Code heißt **Präfixcode**, wenn kein Codewort als Präfix eines anderen Codeworts auftaucht.

Bsp:

a)

a	⟶	0
b	⟶	1
c	⟶	10
d	⟶	11
e	⟶	100
f	⟶	101
g	⟶	110
␣	⟶	111

kein Präfixcode, da 1 (Codewort für b) Präfix ist von 10 (Codewort für c)

6) Jeder Blockcode ist ein Präfixcode

<u>Lemma:</u> Präfixcodes sind eindeutig dekodierbar.

<u>Aber:</u> Präfixcodes sind nicht die einzigen eindeutig dekodierbaren Codes

<u>Bsp:</u>

a	1
b	10
c	00

10000110000010010001
a c c a b c c a c b c a

Im folgenden betrachten wir jedoch nur Präfixcodes, da diese eine besonders schöne Struktur haben und sehr einfach dekodiert werden können.

<u>Lemma:</u> Präfixcodes lassen sich mit binären Bäumen identifizieren:

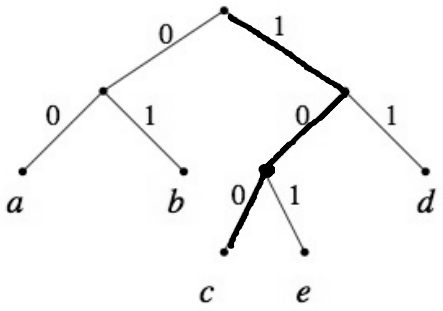
Zeichen $c \in C$ $\hookrightarrow$ Weg im Baum von Wurzel bis Blatt ($\hat{=}$ Zeichen)

Kanten auf dem Weg:
nach links $\hat{=} 0$
" rechts $\hat{=} 1$

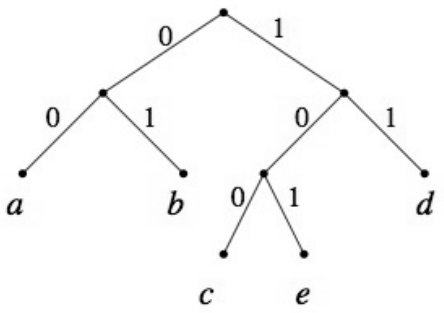
Bsp:

a	⟶	00
b	⟶	01
c	⟶	100
d	⟶	11
e	⟶	101

Baum:



Präfix garantiert, dass die Zeichen bei Konstruktion des binären Baumes nur in Blättern auftauchen.



⟶

a : 00
b : 01
c : 100
d : 11
e : 101

Speicherplatz (#bits) der codierten Datei, die mit Präfixcode T (binärer Baum) codiert wurde:

$$B(T) = \sum_{c \in C} f(c) \cdot h_T(c)$$

$f(c)$: Häufigkeit von c im Originaltext

$h_T(c)$: #bits für Codierung von c

Suche nach „besten" Präfixcode führt zu dem folgenden Optimierungsproblem:

Um eine Datei mit Zeichensatz C und Häufigkeiten $f(c)$, $c \in C$, optimal mit einem Präfixcode zu codieren,

suchen wir einen binären Baum T, dessen Blätter den Zeichensatz C entsprechen, der die Größe

$$B(T) = \sum_{c \in C} f(c) \cdot l_T(c)$$

minimiert, d.h. $B(T) \leq B(T')$ für alle Präfixcodes T'.

- Anwendungsproblem
- mathematische Formulierung des Problems

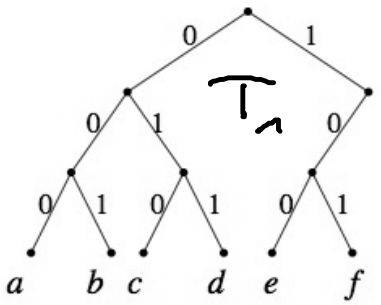
Dieses T heißt dann optimaler Präfixcode.

Beispiel für verschiedene Präfixcodes:

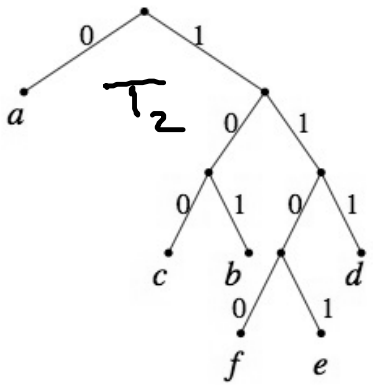
Zeichen c	$f(c)$ in 1000	Code 1	Code 2
a	45	000	0
b	13	001	101
c	12	010	100
d	16	011	111
e	9	100	1101
f	5	101	1100

Σ 100

Code 1



Code 2



$B(T_1) = 300.000$ bits

$B(T_2) = 1000 \cdot (45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4)$ bits
$= 224.000$ bits

Huffman Algorithmus

Input: Zeichensatz C, Häufigkeiten $f(c)$

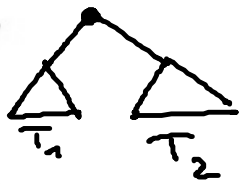
Output: Optimaler Präfixcode T

Algorithmus 3.1 (Huffman Algorithmus)

1. *Fasse jedes Zeichen $c \in C$ als einelementigen Baum auf und füge es in eine Priority Queue Q ein, wobei die Häufigkeit $f(c)$ als Schlüsselwert dient.*

2. *Solange Q mehr als einen Baum enthält:*

   - *Wähle die beiden Bäume T_1 und T_2 mit den kleinsten Häufigkeiten (muss nicht eindeutig sein).*
   - *Entferne sie aus Q.*
   - *Konstruiere einen neuen Baum aus den T_1, T_2 als Teilbäume unter einer neuen Wurzel und gebe ihm die Häufigkeit $f(T_1) + f(T_2)$.*
   - *Füge diesen Baum in Q ein.*

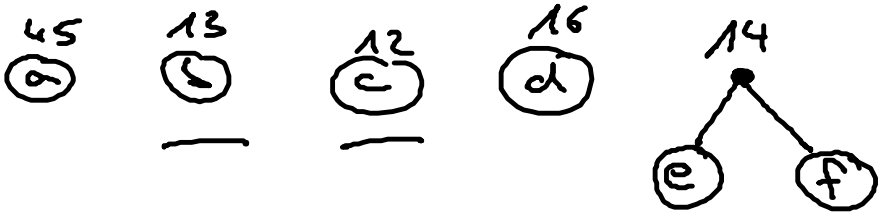


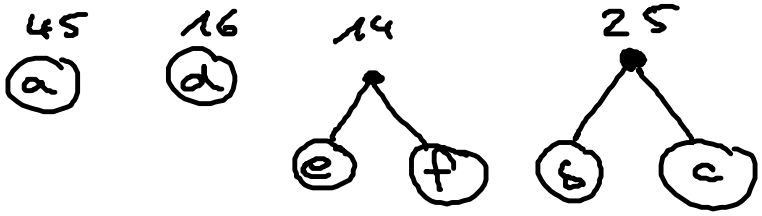
3. *Gebe (den einzig übrig gebliebenen Baum) T in Q zurück. Dieser Baum (der so genannte* Huffman Baum *oder* Huffman Code*) ist ein optimaler Präfixcode.*

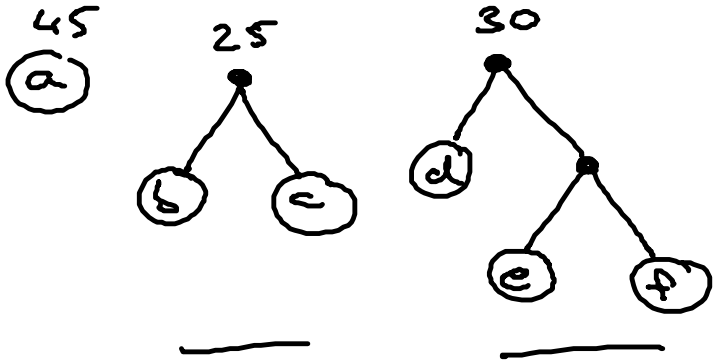
Beispiel:

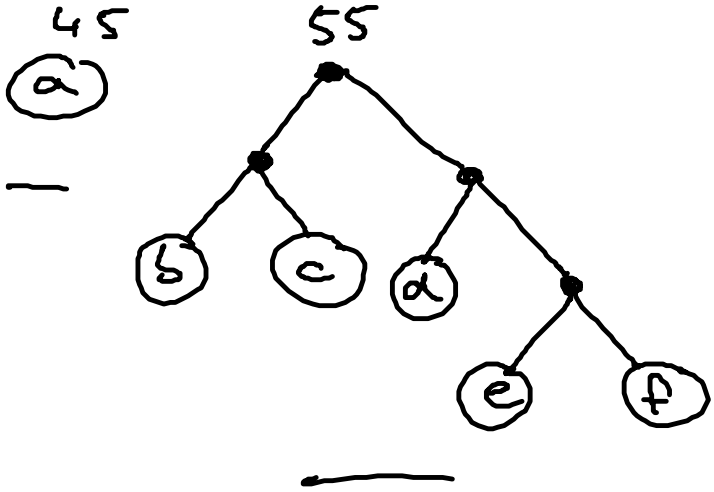
Zeichen	a	b	c	d	e	f
f	45	13	12	16	9	5

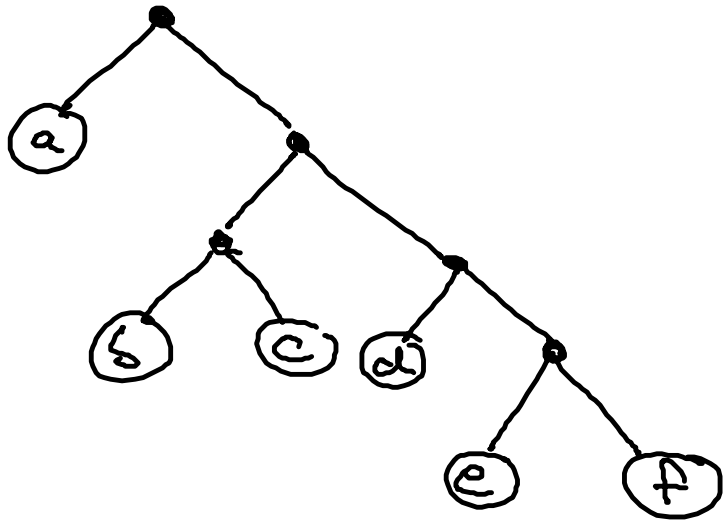
Q:










a
b
c
d
e
f