# Huffman-Kodierung

- Kodierung = Konvertierung der Darstellung von Information in eine andere Darstellung
  - Beispiel: Schriftzeichen sind ein Code, ASCII ein anderer Code
- Eine (von vielen) Aufgaben der Kodierung: platzsparende Speicherung von Information
- Idee der Huffman-Kodierung: Kodiere Eingabewörter mit konstanter Länge (z.B. einzelne Zeichen) durch Code-Wörter mit variabler Länge.
  - Häufig auftretende Eingabewörter werden mit einem kurzen Code dargestellt
  - Seltene Eingabewörter bekommen ein längeres Code-Wort



# Morsecode

- Morsecode kodiert Buchstaben in Abhängigkeit der Häufigkeit

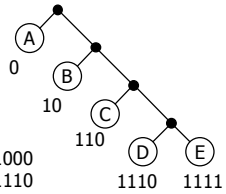
| A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| •– | –••• | –•–• | –•• | • | ••–• | ––• | •••• | •• | •––– | –•– | •–•• | –– |

| N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| –• | ––– | •––• | ––•– | •–• | ••• | – | ••– | •••– | •–– | –••– | –•–– | ––•• |

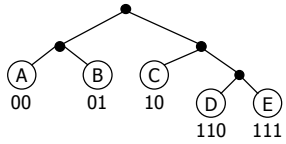
- Alphabet mit drei Zeichen: •, –, ▫ (pause)
- HALLO → •••• ▫ •– ▫ •–•• ▫ •–•• ▫ –––
- Problem beim Weglassen der Pause:
  ••••• –•–••• –•• ––– → EEEEETETEEETEETTT
  ••••• –•–••• –•• ––– → ISTRERAM
  ••••• –•–••• –•• ––– → HRBXM
  ••••• –•–••• –•• ––– → IIAAIAIO … usw.
- Definition Fano-Bedingung:
  kein Code ist Präfix eines anderen Codes

## Erstellung eindeutiger Codes

- Idee: betrachte Eingabewörter als Blätter eines Tries
- Code entspricht Pfad von Wurzel zu Blatt
  - Verzweigung zum linken Sohn → „0"
  - Verzweigung zum rechten Sohn → „1"
- Erfüllt offensichtlich die Fano-Bedingung
- Länge eines Codes = Tiefe des entspr. Blattes
- Beispiel: Σ = {A, B, C, D, E}, Text = `AABAACDAAEABACD`




001000110111000
111101001101110
→ 30 bits




000001000010110000011100010010110
→ 33 bits



- Bemerkung: jeder beliebige Trie mit *M* Blättern kann benutzt werden, um jeden beliebigen String mit *M* verschiedenen Zeichen zu codieren!
  - Die Darstellung als Trie garantiert, daß kein Code für ein Zeichen mit dem Anfang eines anderen übereinstimmt, so daß sich die Zeichenfolge unter Benutzung des Trie auf eindeutige Weise decodieren lässt
  - Bei der Wurzel beginnend bewegt man sich entsprechend den Bits der Zeichenfolge im Trie abwärts; jedesmal, wenn ein Blatt angetroffen wird, gebe man das zu diesem Knoten gehörige Zeichen aus und beginne erneut bei der Wurzel

## Eigenschaften eines optimalen Code-Baumes

- Lemma: Ein optimaler Trie zur Kodierung ist ein voller Baum, d.h., jeder innere Knoten hat genau 2 Kinder.
- Beweis durch Widerspruch (Übungsaufgabe)

→Sei $C$ das Alphabet, dann hat der Trie $|C|$ viele Blätter (klar) und $|C|$-1 innere Knoten (s.o.)

- Gegeben: String $S = s_1 \ldots s_n$, $s_i \in \Sigma$

  sei $h_S(c)$ = absolute Häufigkeit des Zeichens $c$ im String $S$
  sei $d_T(c)$ = Tiefe von $c$ im Trie $T$ = Länge des Codewortes von $c$
- Anzahl Bits zur Kodierung von $S$ mittels Code $T$ gegeben durch

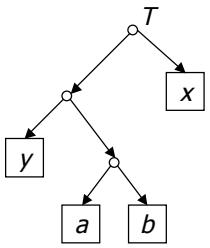
$$B_T(S) = \sum_{c \in C} h_S(c) d_T(c)$$

- Ziel: $T$ bestimmen, so daß $B_T(S)$ = min



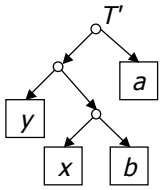
- Lemma: Seien $\Sigma$, $h$ wie oben, seien $x, y \in \Sigma$ diejenigen Zeichen mit minimaler Häufigkeit. Dann ex. ein optimaler Codierungs-Trie für $\Sigma$, in dem $x$, $y$ Brüder sind und auf dem untersten Level.
- Beweis durch Widerspruch:
  - Idee: ausgehend von irgend einem optimalen Trie, konstruiere neuen Trie, der auch optimal ist und die Bedingungen erfüllt
  - Ann.: $T$ ist optimaler Trie, aber $x$, $y$ sind nicht auf max. Level → es ex. 2 Blätter $a$, $b$, die Brüder sind, mit

$$d_T(a) \geq d_T(x) \quad \text{und}$$
$$d_T(b) \geq d_T(y)$$



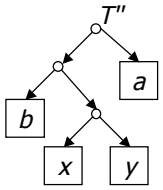
- Vertausche Blätter $x$ und $a$ → Trie $T'$

$$\begin{aligned} B_T(S) - B_{T'}(S) &= \sum_{c \in C} h(c) d_T(c) \\ &\quad - \sum_{c \in C} h(c) d_{T'}(c) \\ &= h(x) d_T(x) + h(a) d_T(a) \\ &\quad - h(x) d_{T'}(x) - h(a) d_{T'}(a) \\ &= h(x) d_T(x) + h(a) d_T(a) \\ &\quad - h(x) d_T(a) - h(a) d_T(x) \\ &= \big(h(a) - h(x)\big)\big(d_T(a) - d_T(x)\big) \\ &\geq 0 \end{aligned}$$



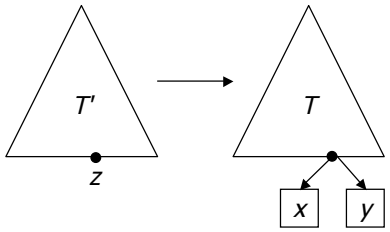

- Analog: $y$ und $b$ vertauschen ergibt $T''$ mit

$$B_{T''}(S) \leq B_{T'}(S)$$

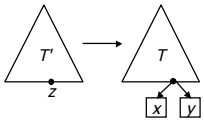





- Lemma:
  Seien $\Sigma$, $h$ wie oben, seien $x, y \in \Sigma$ Zeichen mit minimaler Häufigkeit.
  Setze $\Sigma' := \Sigma \setminus \{x, y\} \cup \{z\}$. Definiere $h$ auf $\Sigma'$ wie auf $\Sigma$ und setze $h(z) := h(x) + h(y)$.
  Sei $T'$ ein optimaler Trie (bzgl. Codelänge) über $\Sigma'$.
  Dann erhält man einen optimalen Trie $T$ für $\Sigma$ aus $T'$, indem man das Blatt $z$ in $T'$ ersetzt durch einen inneren Knoten mit den beiden Kindern $x$ und $y$.



- Beweis:
  - zunächst: $B(T)$ mittels $B(T')$ ausdrücken




  - $\forall c \in \Sigma \setminus \{x, y\} : d_T(c) = d_{T'}(c)$
    $d_T(x) = d_T(y) = d_{T'}(z) + 1$
    $\rightarrow h(x)d_T(x) + h(y)d_T(y) = \big(h(x) + h(y)\big)\big(d_{T'}(z) + 1\big)$
    $= h(z)d_{T'}(z) + h(x) + h(y)$
    $\rightarrow B(T) = B(T') + h(x) + h(y)$
  - Ann.: $T$ ist nicht optimal → ex. $T''$ mit $B(T'') < B(T)$
    - oBdA (gemäß erstem Lemma): $x$, $y$ sind Brüder in $T''$
    - Erzeuge Trie $T'''$ aus $T''$, in dem $x$, $y$ und deren gemeinsamer Vater ersetzt werden durch neues Blatt $z$, mit $h(z) = h(x) + h(y)$
      $\rightarrow B(T''') = B(T'') - h(x) - h(y)$
      $< B(T) - h(x) - h(y) = B(T')$
    - Also wäre schon $T'$ nicht optimal gewesen → Widerspruch!



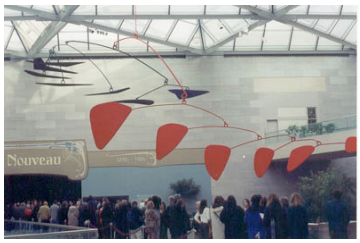
## Huffman-Codes

- Idee: Erzeugung eines optimalen Code-Baums in zwei Schritten
  1. sortiere Wörter nach Häufigkeit
  2. erzeuge Baum „bottom-up" durch wiederholtes Verschmelzen der beiden seltensten Wörter
- Beispiel: Σ = {A, B, C, D, E}, Text = `AABAACDAAEABACD`

1. Ermittlung der Häufigkeiten und Sortierung

| Zeichen | Häufigkeit | $P$(Zeichen) |
|---|---|---|
| A | 8 | 8/15 |
| B | 2 | 2/15 |
| C | 2 | 2/15 |
| D | 2 | 2/15 |
| E | 1 | 1/15 |

## 2. Aufbau des Baums



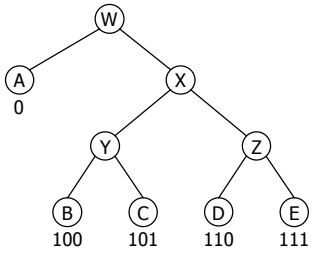

A: 8 B: 2 C: 2 D: 2 E: 1

A: 8 Z: 3 B: 2 C: 2

A: 8 Y: 4 Z: 3

A: 8 X: 7

W: 15



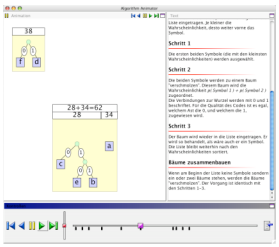

AABAACDAAEABACD
→ 00100001011100011101000101110
= 29 bits



## Implementierung

```
# C = Alphabet, jedes c∈C hat c.freq
# Q = P-Queue, sortiert nach c.freq
füge alle c∈C in Q ein
for i in range( 1, len(C) ):
    x = y = extract_min( q )
    erstelle neuen Trie-Knoten z mit Kindern x und y
    z.freq = x.freq + y.freq
    q.insert( z )
return extract_min(q)    # Wurzel des Baumes
```

(Anmerkung: dies ist ein Greedy-Algorithmus)



Algorithmus-Animation

## Gesamt-Algorithmus zum Kodieren

1. Aufbau des Code-Baums
   - Häufigkeit beim Durchlaufen des Textes ermitteln
   - Heap für die Knoten, geordnet nach Häufigkeit
   - Binärer Trie für den Code-Baum
2. Erzeugen der Code-Tabelle
   - Traversierung des Baums
3. Kodieren des Textes
   - Look-up des Codes pro Zeichen in Code-Tabelle
4. Speichern des Code-Baumes zusammen mit dem kodierten Text



## Dekodierung mit Huffman-Codes

- Aufbau des Code-Baums
  - binärer Trie
- Dekodieren der Bitfolge
  - beginne an der Wurzel des Code-Baums
  - steige gemäß der gelesenen Bits im Baum ab
  - gib bei Erreichen eines Blattes das zugehörige Zeichen aus und beginne von vorne

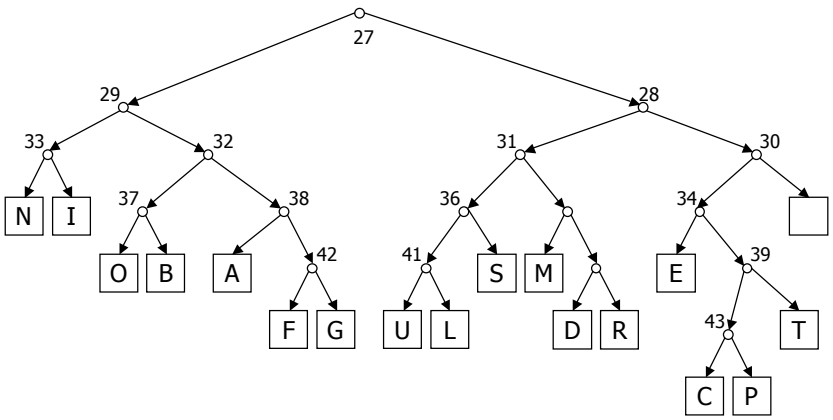
## Weitere Beispiele

- Text: `A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS`
- Dazugehörige Häufigkeits-Tabelle:

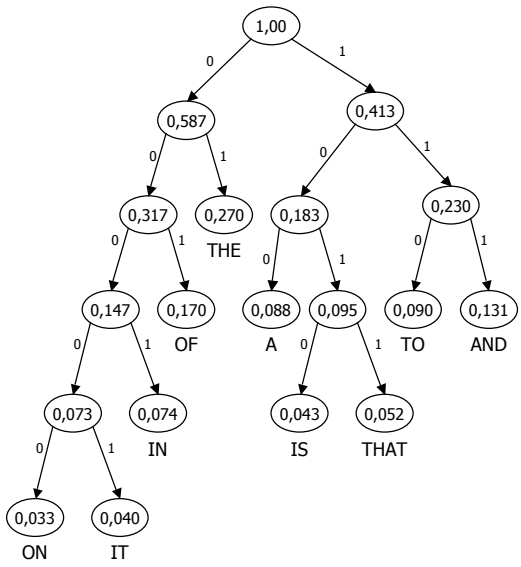
| | | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| k | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| h(k) | 11 | 3 | 3 | 1 | 2 | 5 | 1 | 2 | 0 | 6 | 0 | 0 | 2 | 4 | 5 | 3 | 1 | 0 | 2 | 4 | 3 | 2 | 0 | 0 | 0 | 0 | 0 |

  - es sind 11 Leerzeichen, drei A, drei  B usw.






(Die kleine Zahl oberhalb jedes Knotens in diesem Baum ist der Index für das Array `count`, der angibt, wo die Häufigkeit gespeichert ist. Kann man auch anders nummerieren.)

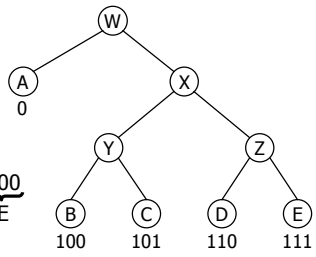


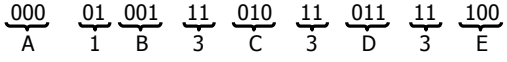

| Codierungseinheit | W'keit des Auftretens | Code-Wert | Code-Länge |
|---|---|---|---|
| the | 0,270 | 01 | 2 |
| of | 0,170 | 001 | 3 |
| and | 0,137 | 111 | 3 |
| to | 0,099 | 110 | 3 |
| a | 0,088 | 100 | 3 |
| in | 0,074 | 0001 | 4 |
| that | 0,052 | 1011 | 4 |
| is | 0,043 | 1010 | 4 |
| it | 0,040 | 00001 | 5 |
| on | 0,033 | 00000 | 5 |



## Kodieren des Wörterbuchs

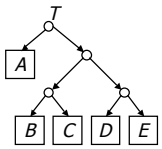
- Ziel: Kompaktes Format des Code-Baums
- Einfachste Variante:
  - Pro Zeichen $k+n$ Bits, $k$ konstant
  - $k$ Bits zur Repräsentation der Zahl $n$, $n$ Bits zur Repr. des Codes
  - Nicht effizient
- Besser:
  - Traversierung des Tries (z.B. in Preorder)
  - Pakete aus Zeichen plus Code-Länge (beides mit konstanter Bit-Anzahl)
- Beispiel: $\Sigma$ = {A, B, C, D, E}



## Noch effizientere Übertragung des Codebaumes

- Beobachtungen:
  - Genaue Lage eines Zeichens im Baum ist egal! Wichtig ist nur die Tiefe!
  - Umarrangieren des Tries (bei fester Zuordnung zwischen Zeichen und Blättern) liefert Trie, in dem Blätter von links nach rechts tiefer werden
  - Codes der Zeichen ändern sich, nicht aber Optimalität
- Darstellung des Tries:
  - Gib Anzahl Zeichen mit bestimmter Länge an, gefolgt von diesen Zeichen, aufsteigend nach Länge
  - Beispiel:
    - 1 Zeichen der Länge 1, 0 der Länge 2, 4 der Länge 3
    - 01, A, 00, , 04, B, C, D, E
  - Dann lässt sich der ursprüngliche Trie wieder genau rekonstruieren






## Eigenschaften

- In gewissem Sinn(!) optimaler, d.h. minimaler, Code
  - Voraussetzung u.a.: Vorkommen der Zeichen ist unabhängig voneinander
- Keine Kompression für zufällige Daten
  - alle Zeichen habe annähernd gleiche Häufigkeit
- Erfordert zweimaliges Durchlaufen des Textes
  - ermöglicht kein „Stream-Processing"
- Wörterbuch muß i.a. zusätzlich gespeichert werden
  - konstante Größe
  - für große Dateien vernachlässigbar

## Huffman-Varianten

- Dynamische Zeichenverteilung
  - Häufigkeitsanalyse vor Beginn der Kodierung
- Statische Zeichenverteilung
  - Anhand von Analysen fest vorgegebener Code-Baum
  - Z.B. für jede Sprache (Englisch,…) ein eigener, fester Code-Baum
  - Wörterbuch muß nicht abgespeichert werden
- Adaptive Zeichenverteilung
  - Anpassung der Verteilung zur Laufzeit
- Anwendung: JPEG (Encoder im Back-End)



## Zusammenfassung Greedy-Algorithmen

- Allgemeine Vorgehensweise:
  - Treffe in jedem Schritt eine Entscheidung, die 1 kleineres Teilproblem übrig lässt
- Korrektheitsbeweis:
  - Zeige, daß eine optimale Lösung des entstehenden Teilproblems zusammen mit der getroffenen Entscheidung zu optimaler Lösung des Gesamtproblems führt
  - Das lässt sich meist durch Widerspruch zeigen:
    - angenommen, es gibt optimale Lösung $S'$, die besser ist als Greedy-Lösung $S$
    - zeige: dann kann $S'$ noch verbessert oder in Greedy-Lösung umgewandelt werden

## Greedy vs. Dynamische Programmierung

- Ähnlich: optimale Unterstruktur
- Verschieden:
  - bei der dynamischen Programmierung erhält man oft mehrere Unterprobleme, bei Greedy-Algorithmen (in der Regel) nur eines
  - bei dynamischer Programmierung hängt die Entscheidung von der Lösung der Unterprobleme ab, bei Greedy-Algorithmen nicht



## Beispiele für Greedy-Algorithmen

- Scheduling
  - Activity-Selection-Problem
  - Deadline-Scheduling
- Graphen-Algorithmen
  - Minimum-Spanning-Tree
  - Dijkstras Kürzester-Weg-Algorithmus
- Greedy-Strategie dient oft als Heuristik für "harte" Probleme:
  - Graphenfärbung (4-Farben-Problem)
  - Traveling-Salesman-Problem (TSP)
  - Set-Covering