Vortrag am 25. Januar 2006
Werner von Siemens Gymnasium Magdeburg
Zeitansatz: 5h (inklusive Programmieraufgaben)


# Datenkompression

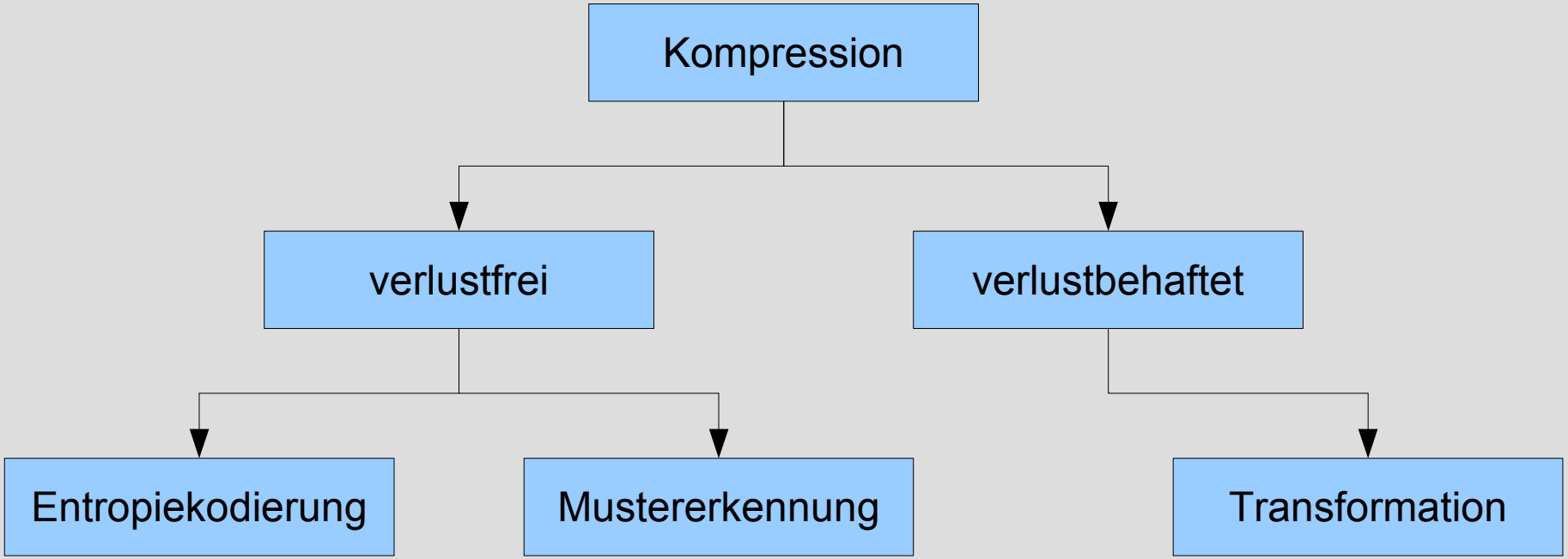
Vortrag von Markus Durzinsky
Student der Otto-von-Guericke-Universität Magdeburg

http://www-e.uni-magdeburg.de/durzinsk
Mardur@gmx.net

# Gliederung

- Überblick über Kompressionsverfahren
- Verlustbehaftete Verfahren
  - Motivation von Transformationen
  - Kompression von Musik (MP3)
  - Kompression von Bildern (JPEG)
- Muster-basierte Verfahren
  - Lauflängenkodierung (RLE) — Programm 1
  - Sukzessive Mustersuche (LZW) — Programm 2
- Entropiekodierung
  - Shannon-Fano-Verfahren — Programm 3
  - Huffman-Verfahren

# Überblick über Kompressionsverfahren



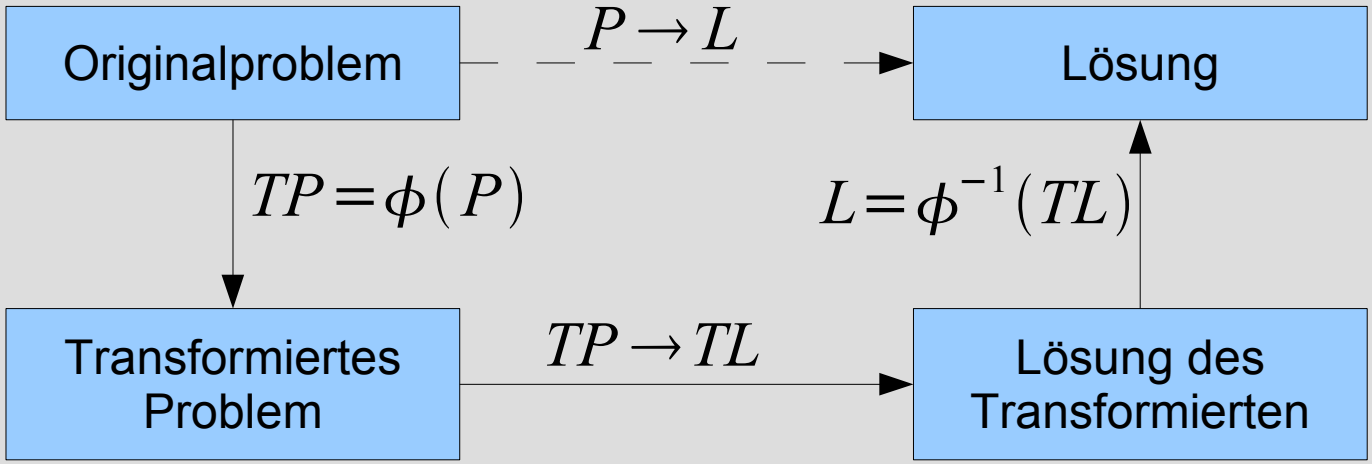

Huffman
Shannon-Fano

Lauflängenkodierung
Lempel-Ziv-Welsh

JPEG, MP3

# Motivation des Transformationsansatzes

- Transformiere Problem von einem *Raum* in einen anderen
- Im transformierten Raum lässt sich das Problem besser / schneller lösen



# Beispiel Diskrete Fouriertransformation (DFT)

- Eindeutige Darstellung von Polynomen durch Koeffizienten

$$(a_n, a_{n-1}, \ldots, a_{1,} a_0) \rightarrow f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

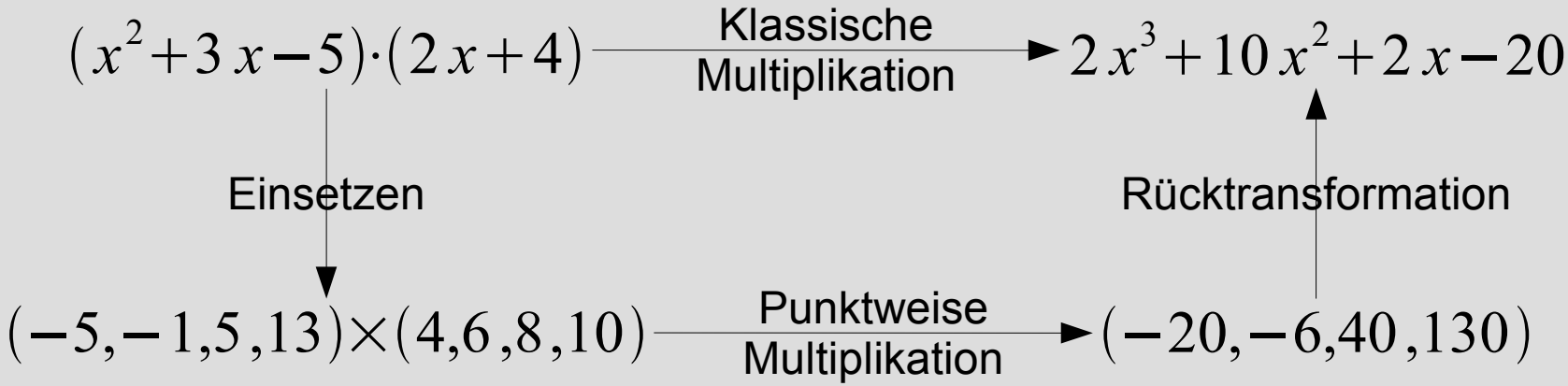
- Eindeutige Darstellung durch Funktionswerte

$$(y_n, y_{n-1}, \ldots, y_{1,} y_0) \rightarrow f \; mit \; f(x_n) = y_n, \ldots, f(x_0) = y_0$$

- Transformation durch Einsetzen

# Aufwand der Polynommultiplikation

- Berechnung an Stellen $x_0=0, x_1=1, x_2=2, x_3=3$




- Klassische Multiplikation mit quadratisch vielen Operationen
- Punktweise Multiplikation mit linear vielen Operationen

# Diskrete Cosinustransformation (DCT)

- Gegeben Funktionswerte (Ortsraum)

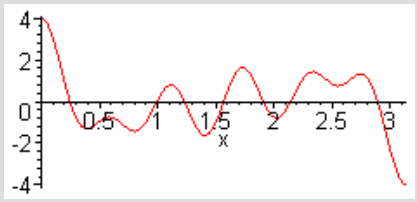
$$f(0), f(1), f(2), \ldots$$

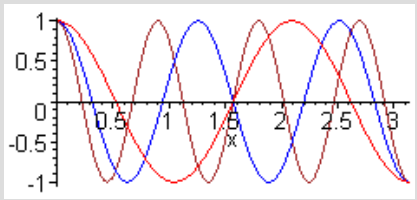
DCT

$$f(x) = a_0 g_0(x) + a_1 g_1(x) + \cdots$$

$$g_k(x) = \cos(k x)$$

(Vereinfachte Darstellung ohne Konstanten)



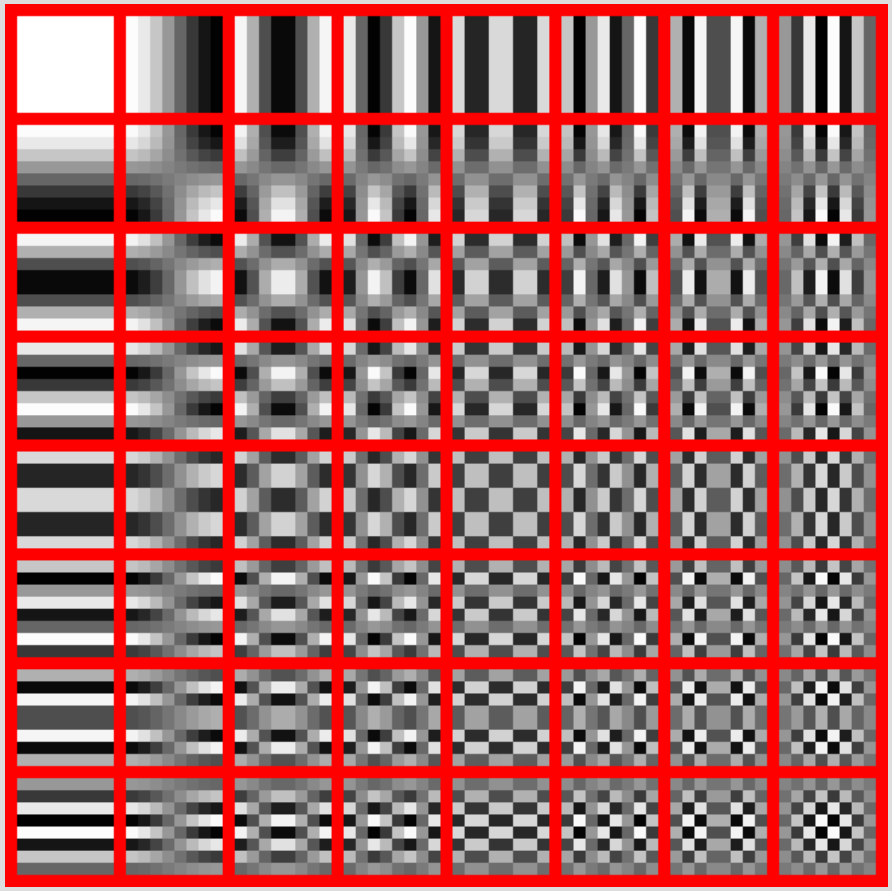




- Summe von Einzelwellen (Frequenzraum)

# DCT für MP3

- DCT arbeitet exakt und verlustfrei
- Datenreduktion erfolgt durch
  - Entfernen nicht hörbarer Frequenzen (außerhalb von 20Hz..18kHz)
  - Entfernen oder Runden nicht hörbarer Unterschiede und überlagerter Signale
- Entstehende Daten werden mit Huffman-Verfahren weiter reduziert

# DCT für JPEG



- Aufteilung in 8x8 Blöcke

$$f(x,y) \qquad \text{für } x,y = 0 \ldots 7$$

- Suche Darstellung

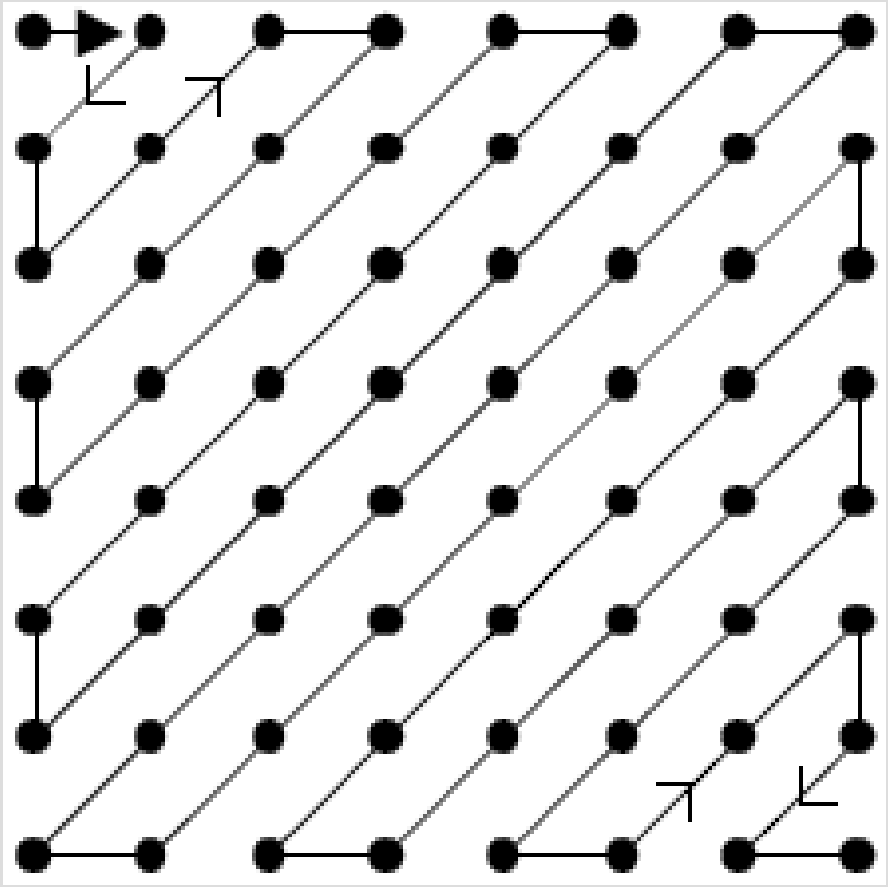
$$f(x,y) = \sum_i \sum_j s_{ij} g_{ij}(x,y)$$

$$g_{ij}(x,y) = \cos(ix)\cos(jy)$$

(Vereinfachte Darstellung ohne Konstanten)

- verlustfrei

# DCT für JPEG

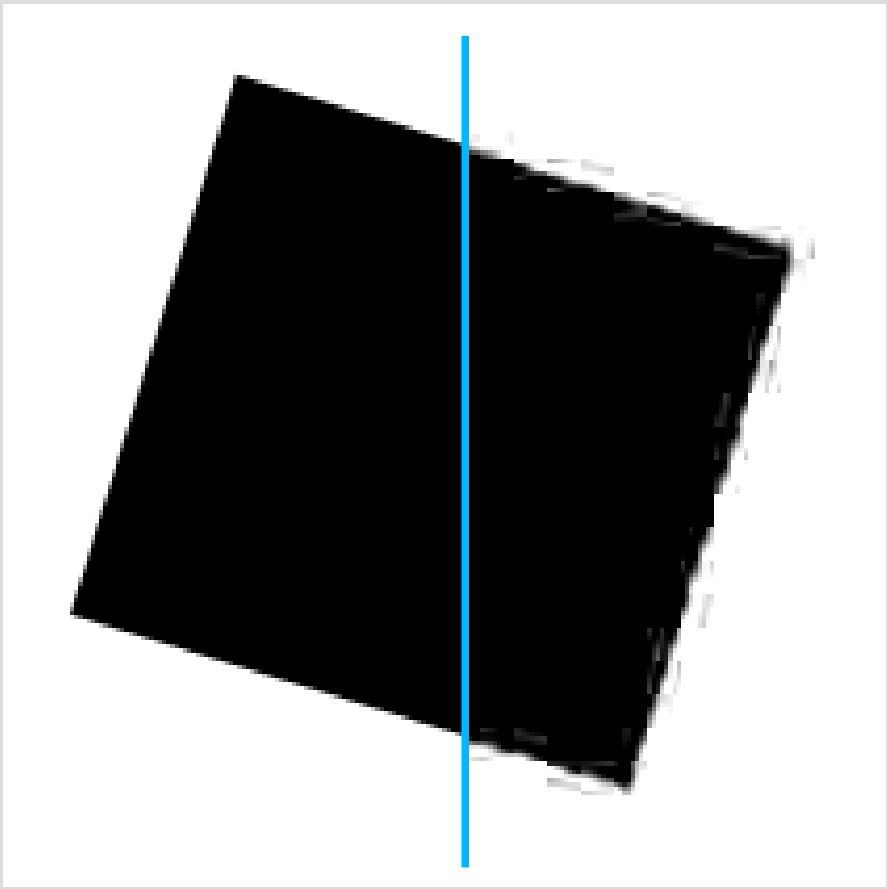


- Rundung der Koeffizienten (Quantisierung)

$$S_{ij} = round(s_{ij} / q_{ij})$$

- Speichere in zig-zag Reihenfolge mit Huffman-Verfahren

# DCT für JPEG



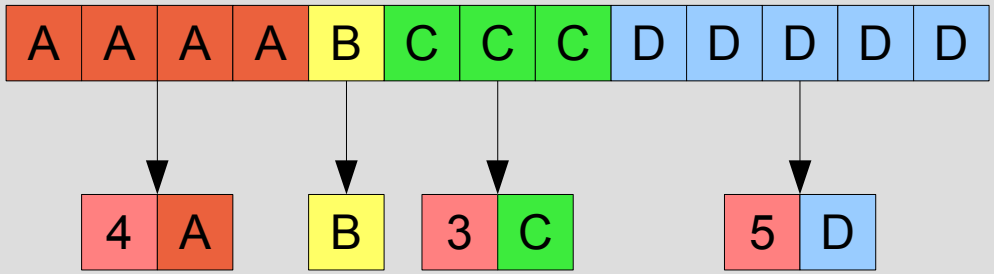
- Verfälschungen durch Datenverlust (Blockartefakte)
- Kompression mit vorgegebener Datengröße oder Qualität möglich
- Progressiv (erst alle $S_{00}, S_{10}, S_{01}$ dann alle anderen speichern)

# Mustererkennung

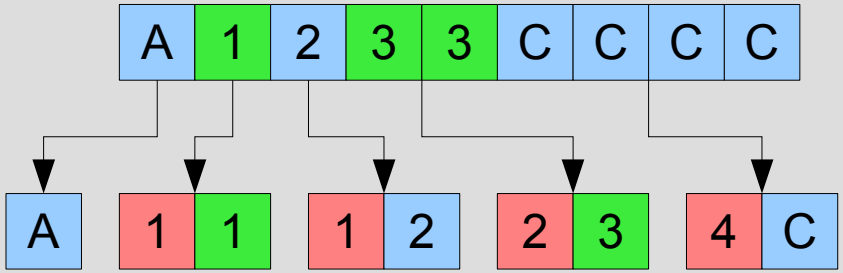
- Speichere Wiederkehrende Muster nur einmal
- Beim nächsten Auftreten, Speichere Referenz auf das Muster
- Beispiel
  - AUCH EIN KLEINER BEITRAG IST EIN BEITRAG
- Referenzen auf frühere Stellen
  - AUCH EIN KLEINER BEITRAG IST {-4} {-3}
- Oder mit vollständiger Mustertabelle
  - AUCH, EIN, KLEINER, BEITRAG, IST: 1 2 3 4 5 2 4

# Lauflängenkodierung (RLE)

- Direkte Wiederholung von Zeichen




- Wie unterscheidet man Zeichen und Anzahl?




- Steuercodes kapseln

# Programmieraufgabe 1
# RLE

- Implementieren Sie die Lauflängenkodierung (und Dekodierung) in Delphi
- Als Eingabe dient eine beliebige Bytefolge
- Zur Unterscheidung von Zeichen und Anzahlen Speichern Sie eine Anzahl $n = 1 \ldots 63$ als $n + 192$
- Alle Zeichen mit ASCII-Code größer als $192$ müssen gekapselt werden, als Folge $[1, code]$ (außer der Code tritt mehrfach auf)

# Programmieraufgabe 1
# RLE – Musterlösung

- Algorithmus RLEEncoding
- code := 0,  count := 0
- WHILE CanReadMore DO
  - next := ReadByte
  - IF next = code AND count < 63 THEN
    - count := count + 1;
  - ELSE
    - WriteCode(code, count);
    - code := next;  count := 1;
- WriteCode(code, count);

- Algorithmus WriteCode(code, count)
- IF count > 1 OR code > 192 THEN
  - WriteByte(count + 192);
- WriteByte(code);

# Programmieraufgabe 1
# RLE – Musterlösung

- Algorithmus RLEDecoding
- WHILE CanReadMore DO
  - code := ReadByte;
  - IF code <= 192 THEN
    - WriteByte(code);
  - ELSE IF CanReadMore THEN
    - count := code – 192;
    - code := ReadByte;
    - FOR i := 1 TO count DO
      - WriteByte(code);

# Kodierung mit Mustertabelle nach Lempel-Ziv-Welsh

- Verfahren benutzt bei Bildformaten GIF, TIFF
- ZIP-Dateien benutzen Variante von LZW

- Ausgabe besteht nur aus Tabellenindizes
- Tabelle wird mit Mustern der Länge eins initialisiert
- Neue Muster entstehen aus alten Mustern plus ein weiteres Zeichen, damit vereinfacht sich die Mustersuche und die Datenstruktur

# LZW-Mustertabelle

- Speichere Muster als Präfix-Suffix-Paar
- Index Muster Präfix Suffix

| Index | Muster | Präfix | Suffix | |
|---|---|---|---|---|
| 65 | A | - | A | |
| 66 | B | - | B | |
| 67 | C | - | C | |
| 256 | AB | 65 | B | |
| 257 | BA | 66 | A | |
| 258 | ABA | 256 | B | |
| 259 | ABAB | 258 | B | (Problemfall) |

- Entsteht bei Eingabe ABABABAB

# Programmieraufgabe 2 LZW-Komprimierung

- Algorithmus LZWEncoding
- initialisiere Mustertabelle mit (<leeres Muster>,zeichen)
- muster ← <leeres Muster>
- solange noch Zeichen verfügbar
  - zeichen ← lese Zeichen
  - wenn (muster,zeichen) in Mustertabelle dann
    - muster ← (muster,zeichen)
  - sonst
    - füge (muster,zeichen) zur Mustertabelle hinzu
    - Ausgabe Index von muster
    - muster ← (<leeres Muster>,zeichen)  (= zeichen)
- wenn muster ≠ <leeres Muster> dann
  - Ausgabe Index von Muster

# Programmieraufgabe 2 LZW-Dekomprimierung

- Algorithmus LZWDecoding
- initialisiere Mustertabelle mit (<leeres Muster>,zeichen)
- last ← <leeres Muster>
- solange noch Codes verfügbar
  - next ← lese Code
  - wenn next in Mustertabelle dann
    - wenn last ≠ <leeres Muster> dann
      - füge (last, erstes Zeichen von Muster next) zur Mustertabelle hinzu
  - sonst
    - füge (last, erstes Zeichen von Muster last) zur Mustertabelle hinzu
  - Ausgabe Muster von next
  - last ← next

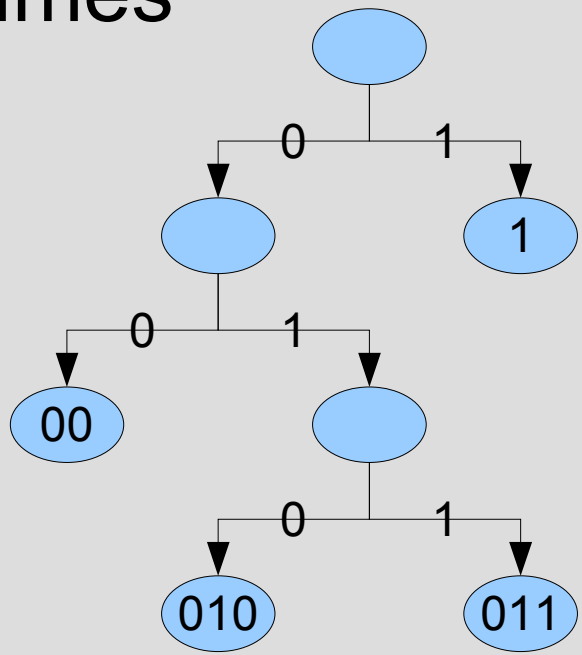
# Entropiekodierung

- Entropie ist ein Maß für den Informationsgehalt einer Menge
- Die Entropie gibt an, wieviele Bits bei optimaler Kompression benötigt werden (falls man jedes Eingabesymbol durch genau ein Codewort kodiert, ohne Mustererkennung)
- Tritt ein Zeichen häufiger auf, besitzt es eine geringere Entropie
- Idee: kodiere häufigere Zeichen mit weniger Bits

# Entropiekodierung

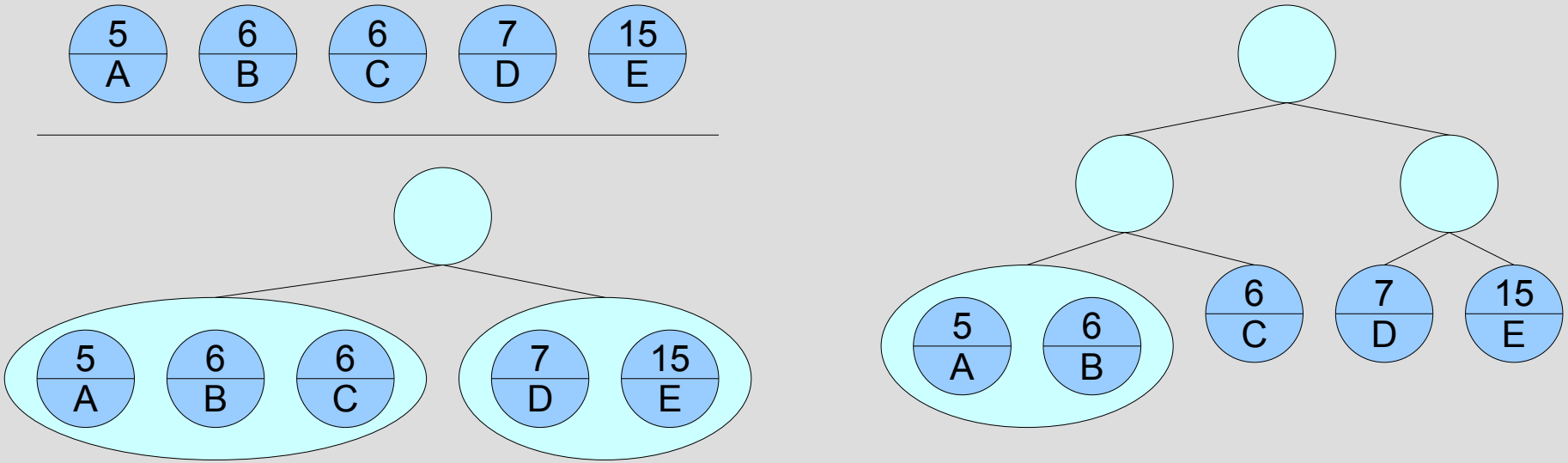
- Wie unterscheidet man eingelesene Codewörter unterschiedlicher Länge?
- Benutze Präfixcode: kein Codewort ist der Anfang eines anderen Wortes (Fano-Kriterium)

- Repräsentiere Präfixcode durch Blätter eines binären Baumes



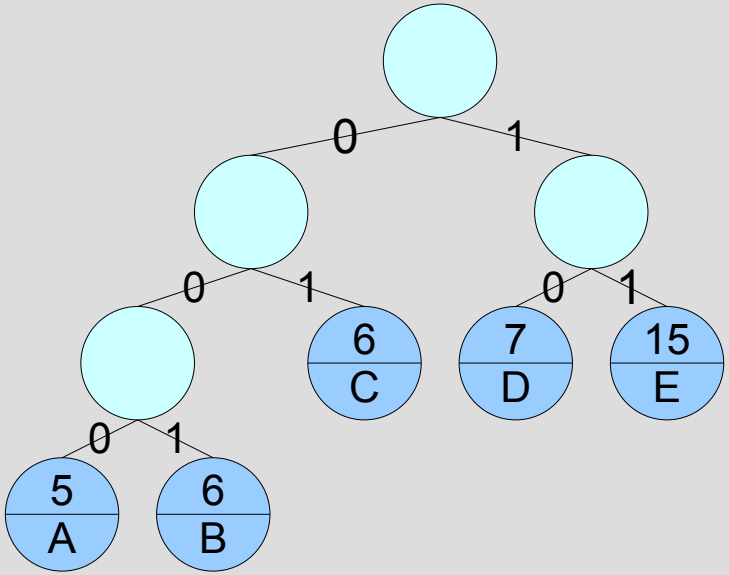
# Shannon-Fano-Verfahren

- Sortiere Knoten nach Häufigkeit
- Zerlege in möglichst gleich große Teile
- Damit befinden sich viele Knoten mit ähnlicher Entropie in einem Teilbaum, und bekommen damit ähnlich lange Codewörter



# Shannon-Fano-Verfahren

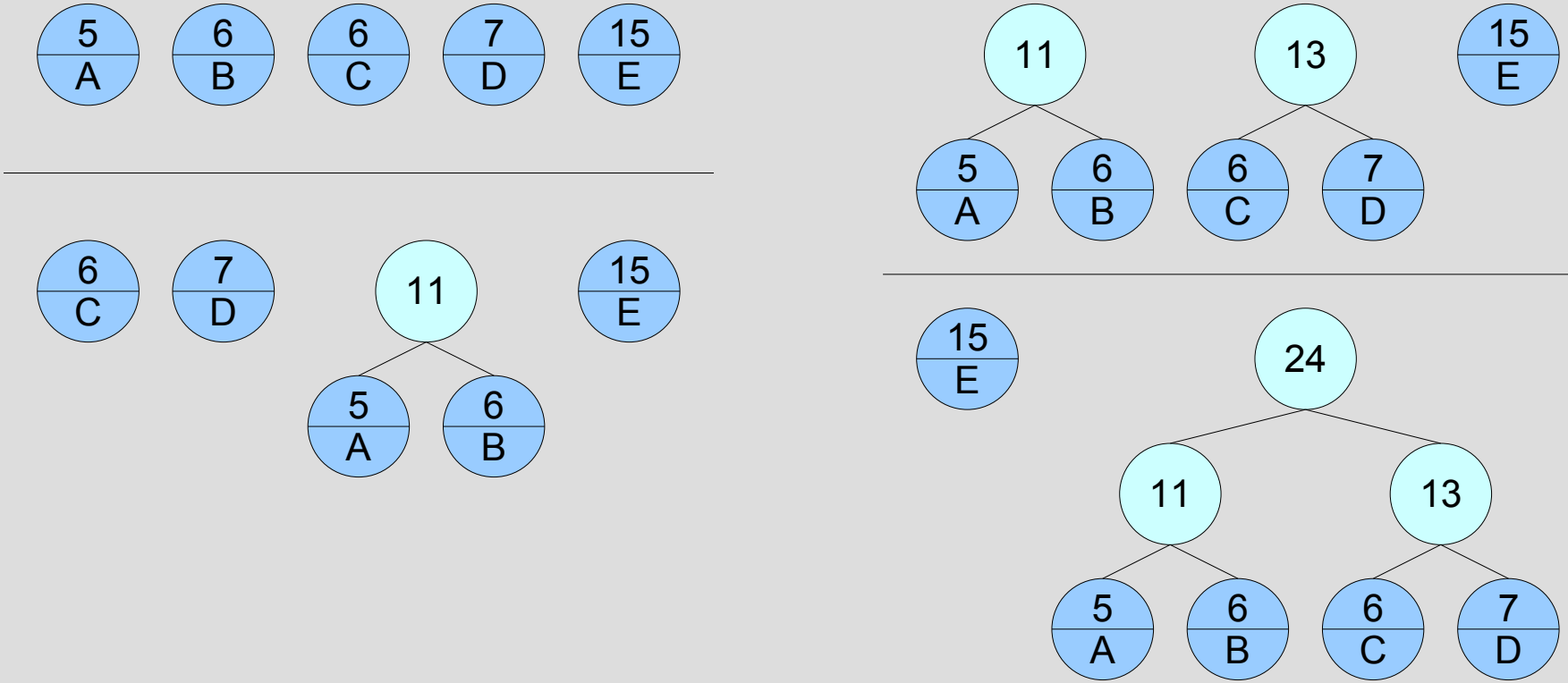
- Ergebnis-Baum




- Komprimierte Länge $2 \cdot (6+7+15) + 3 \cdot (5+6) = 89\, bits$

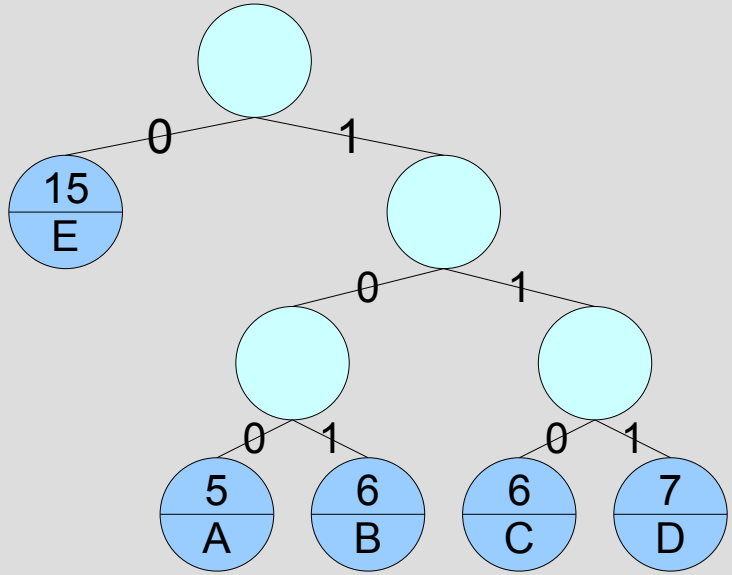
# Huffman-Verfahren

- Füge Knoten mit kleinsten Häufigkeiten zusammen



# Huffman-Verfahren

- Ergebnis als Huffman-Baum




- Komprimierte Länge $1 \cdot 15 + 3 \cdot (5 + 6 + 6 + 7) = 87\, bits$
- Huffman-Baum liefert optimale Codelänge

# Entropiekodierung

- Dynamische Verwaltung des Baumes während des Kodierens
- Nach jedem Zeichen ändern sich Häufigkeiten im restlichen Teil, manche Zeichen treten nicht mehr auf
- Damit werden Codewörter zum Ende hin immer kürzer

# Programmieraufgabe 3 Entropiekodierung

- Programmieren Sie die Entropie-Kodierung und Dekodierung mit Hilfe des Shannon-Fano-Verfahrens
- Hinweise:
  - Als Datenstruktur muss kein Baum verwendet werden, eine Array genügt
  - Die Partitionierung (Konstruktion der Teilbäume) erfolgt auf dem Array mit divide-and-conquer
  - Beachten Sie die maximale Tiefe des Baumes von 30 Knoten (30 Bits Codewörter), aufgrund der Programmierumgebung

# Zusammenfassung

- Verlustbehaftete Kompressionsverfahren reduzieren Daten durch Entfernung von nicht wahrnehmbaren Informationen (*Irrelevanzreduktion*)
- Verlustfreie Verfahren versuchen wiederkehrende Muster zu finden oder die Daten an deren Entropie anzupassen (*Redundanzreduktion*)

# Quellenangabe

- Wikipedia Seite zur Datenkompression:
  http://de.wikipedia.org/wiki/Datenkompression
- Vorlesung Kodierungstheorie und Kryptographie im Sommersemester 2005 bei Prof. Dassow
  http://theo.cs.uni-magdeburg.de/lehre05s/kodierung
- Zur Huffman-Kodierung und Entropie:
  Kyle Loudon, *Mastering Algorithms with C*, 1999 O'Reilly & Associates Inc.
- Zur JPEG-Kompression:
  CCITT Rec. T.81 (1992 E) - digital compression and coding of continuous-tone still images - requirements and guidelines