Grundlagen der Informatik

Vorlesungsskript

Mitschrift von Falk-Jonatan Strube

Vorlesung von Dr. Boris Hollas

21. Januar 2016

Inhaltsverzeichnis

Mitschrift von
Falk-Jonatan Strube

Allgemeine Informationen

Zugelassene Hilfsmittel Klausur: A-4 Blatt (doppelseitig, handbeschrieben)
Prüfungsvorleistung: alle paar Woche eine Lernabfrage in der Vorlesung (Bestanden wenn insgesamt im Schnitt 50%)
Grundlage der Vorlesung: Grundkurs theoretische Informatik [**hollas2007grundkurs**]
Lernkontrolle ab 23.10.2015 alle zwei Wochen.

1 Aussagenlogik

Mit der Aussagenlogik lassen sich Aussagen formulieren, die entweder wahr oder falsch sind. Aussagen sind atomare Aussagen wie „die Straße ist nass“ oder mit Hilfe von logischen Operatoren zusammengesetzte Aussagen.

1.1 Syntax und Semantik

Definition: Die *Formeln der Aussagenlogik* sind induktiv definiert.

- Jede atomare Aussage ist eine Formel der Aussagenlogik. Diese heißen Atomformeln oder Variablen. Atomformeln bezeichnen wir mit Kleinbuchstaben oder durch Wörter in Kleinbuchstaben.
- Wenn F, G Formeln der Aussagenlogik sind, dann auch $(F \wedge G)$, $(F \vee G)$, $(\neg F)$.

Bsp.: Formeln der Aussagenlogik sind x, y, $x \wedge y$, $(x \wedge (y \wedge z)) \vee (\neg x \wedge (y \wedge \neg z))$, $regnet$, $regnet \wedge nass$, da sie jeweils aus atomaren Aussagen die nach der Definition zusammensetzen lassen bestehen.
Keine Formeln der Aussagenlogik sind $x\wedge$, $\vee x$, $x \wedge \vee y$.
Um Klammern zu sparen, legen wir Prioritäten fest:

Operator	Priorität
$\neg$	höchste
$\wedge$, $\vee$	
$\rightarrow$, $\leftrightarrow$	niedrigste

Definition: Eine *Belegung* einer Formel F der Aussagenlogik ist eine Zuordnung von Wahrheitswerten „wahr“ (1) oder „falsch“ (0) zu den Atomarformeln in F. Daraus ergibt sich der *Wahrheitswert* einer Formel:

- Eine Atomformel ist genau dann wahr, wenn sie mit „wahr“ belegt ist.
- Die Formel $F \wedge G$ ist genau dann wahr, wenn F „wahr“ ist und G „wahr“ ist.
  $F \vee G$ ist wahr, wenn F wahr ist oder G wahr.
  $\neg F$ ist wahr, wenn F falsch ist.

F	G	$F \wedge G$	$F \vee G$	$\neg F$	$F \rightarrow G$	$F \leftrightarrow G$
0	0	0	0	1	1	1
0	1	0	1	1	1	0
1	0	0	1	0	0	0
1	1	1	1	0	1	1

Mitschrift von
Falk-Jonatan Strube

Bsp.: Wenn $regnet$ bedeutet: „Es regnet.“
Wenn $nass$ bedeutet: „Die Straße ist nass.“
Dann bedeutet $regnet \wedge nass$: „Es regnet und die Straße ist nass.“
„Wenn es regnet, dann ist die Straße nass“ ($regnet \rightarrow nass$). Es muss nur der Fall ausgeschlossen werden, der nicht eintreffen kann: $\neg(regnet \wedge \neg nass) \Rightarrow$ Folgendes darf nicht eintreffen: „Es regnet und die Straße ist nicht nass“.

Alles andere („Es regnet nicht und die Straße ist nicht nass“, „Es regnet nicht und die Straße ist nass“ und „Es regnet und die Straße ist nass“) darf eintreffen.

$regnet$	$nass$	$\neg(regnet \wedge \neg nass) = \neg regnet \vee nass$
0	0	1
0	1	1
1	0	0
1	1	1

Definition: Die Operatoren $\rightarrow$ (*Implikation*) und $\leftrightarrow$ (*Äquivalenz*) sind definiert durch:

- $F \rightarrow G = \neg F \vee G$
- $F \leftrightarrow G = (F \rightarrow G) \wedge (G \rightarrow F)$

(Siehe Tabelle oberhalb)

Bsp.: Berechnen des Betrags y einer Zahl x:

```
1 if (x>= 0)
2   y=x;
3 else
4   y=-x;
```

Dargestellt als Formel der Assagenlogik: $((x \geq 0) \rightarrow y = x) \wedge (\neg(x \geq 0) \rightarrow y = -x)$

Definition: Eine Formel F der Aussagenlogik heißt

- *erfüllbar*, wenn es eine Belegung gibt, sodass F wahr ist, sonst *unerfüllbar*. Mit $\perp$ bezeichnen wir eine unerfüllbare Formel (Widerspruch).
- *Tautologie* oder *gültig*, wenn F für jede Belegung wahr ist. Bezeichnung: $\top$

Bsp.:

- $x \wedge y$ ist erfüllbar.
- $((\neg x \wedge y) \vee (x \wedge \neg y)) \wedge \neg(x \vee y)$ ist unerfüllbar (linke Seite: entweder x oder y falsch - rechte Seite: x oder y falsch)
- $x \vee \neg x$ ist eine Tautologie

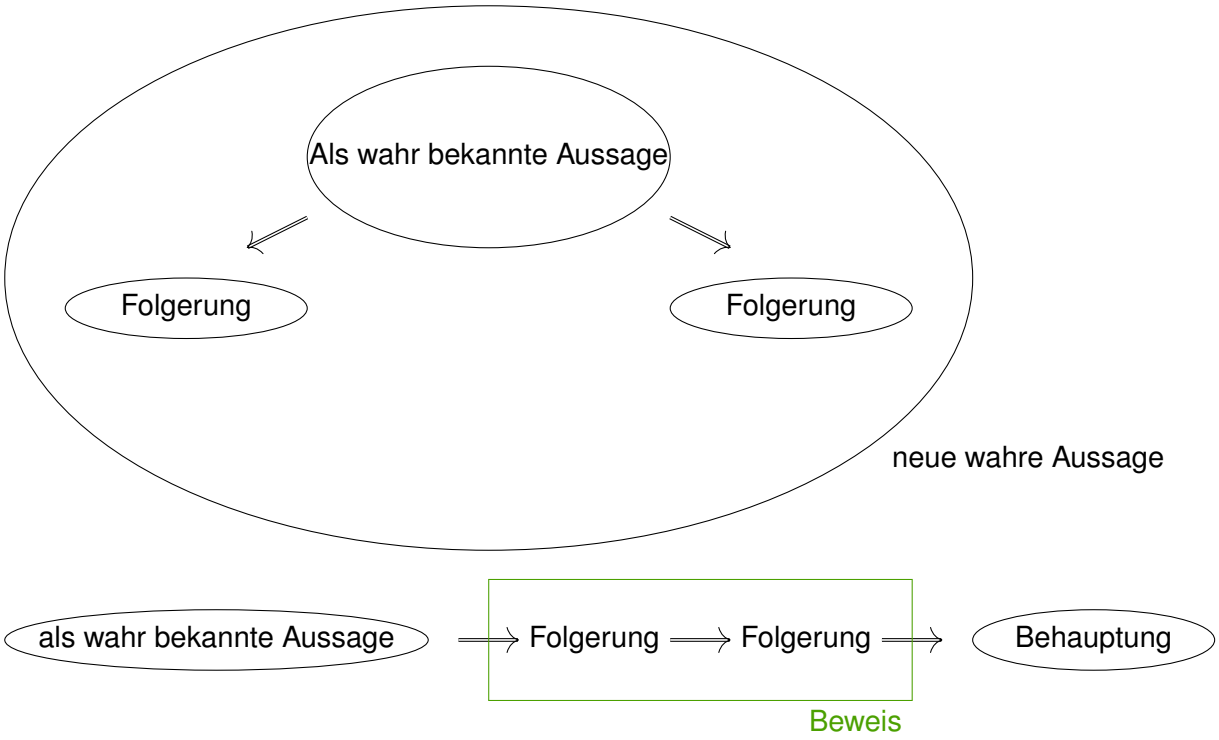
Definition: Wir schreiben $F \equiv G$ („F ist äquivalent zu G“), wenn für jede Belegung gilt: $F \leftrightarrow G$ wahr (d.h., $F \leftrightarrow G$ ist gültig).

1.2 Rechenregeln

siehe Mathematik I

Mitschrift von
Falk-Jonatan Strube

2 Beweistechniken



Direkter Beweis

Bsp.: Wenn $a \in \mathbb{Z}$ gerade ist, dann ist auch a^2 gerade.
$(a \in \mathbb{Z}$ gerade $\Rightarrow a^2$ gerade$)$
Beweis:

- Wenn a gerade ist, gibt es ein n mit $a = 2 \cdot n$.
- Dann gilt $a^2 = 4 \cdot n^2 = 2 \cdot 2n^2$,
- woraus a^2 gerade folgt.

Indirekter Beweis Mit einem indirekten Beweis wird $A \Rightarrow B$ bewiesen, indem die äquivalente Aussage $\neg B \Rightarrow \neg A$ bewiesen wird.

Bsp.: Wenn a^2 gerade ist, dann auch a.
$(a^2$ gerade $\Rightarrow a$ gerade$)$
Beweis: Wir zeigen: Wenn a ungerade ist, dann auch a^2.

- Aus a ungerade folgt $a = 2n - 1$ für ein n.
- Dann ist $a^2 = 4n^2 - 4n + 1 = \underbrace{4(n^2 - n)}_{gerade} + \underbrace{1}_{ungerade}$,
- Aus gerade + ungerade folgt ungerade, woraus a^2 ungerade folgt.

Beweis durch Widerspruch Mit einem Beweis durch Widerspruch wird eine Aussage A bewiesen, indem gezeigt wird, dass die Annahme „A ist falsch“ zu einem Widerspruch führt.
(D.h., es wird $\neg A \rightarrow \perp$ gezeigt)

Mitschrift von
Falk-Jonatan Strube

Bsp.: $\sqrt{2}$ ist irrational. Siehe Mathematik I.

Vollständige Induktion Mit einer vollständigen Induktion lassen sich Aussagen der Art „für alle $n \in \mathbb{N}$ gilt . . . “ beweisen.

Prinzip: Gegeben eine Aussage der Form „für alle $n \in \mathbb{N}$ gilt $A(n)$“

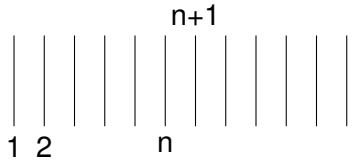
- *Induktionsanfang*: Man zeigt, die Wahrheit der Aussage für $n = 1$ (mit anderen Worten: Man zeigt, dass $A(1)$ wahr ist) [1: die kleinste mögliche Zahl $\Rightarrow$ kann auch 0 oder eine andere sein]
- *Induktionsvorussetzung*: Die Aussage ist für n wahr.
- *Induktionsschritt*: Wenn IV wahr ist, dann ist die Aussage auch für $n + 1$ wahr.

In Formeln: Man zeigt

- IA: $A(1)$
- IV: $A(n)$
- IS: für alle n: $A(n) \Rightarrow A(n + 1)$

Beispiel Dominosteine: Wenn der erste Stein fällt, fällt auch der Zweite. Und wenn der n-te Stein fällt, fällt auch der $n + 1$-te:

- IA: 1. Umstoßen
- IV: Wenn der vorherige Stein umfällt, fällt auch der nächste
- IS: Wenn n-ter umgestoßen wird, dann auch $n + 1$-ter



Bsp.: Für alle $n \geq 1$ gilt $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

Beweis (Induktion):

IA $n = 1$: $1 = \frac{1 \cdot 2}{2}$ ist wahr.

IV Es gelte $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ ist wahr.

IS $n \to n + 1$: Zu zeigen: $\sum_{k=1}^{n+1} k = \frac{(n+1)(n+2)}{2}$ Es gilt:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= (\sum_{k=1}^{n} k) + n + 1 \\ &\overset{IV}{=} \frac{n(n+1)}{2} + n + 1 \\ &= ... = \frac{(n+1)(n+2)}{2} \#\end{aligned}$$

Mitschrift von
Falk-Jonatan Strube

3 Elementare Kombinatorik

Kreuzprodukt:
$A \times B = \{(a,b)|a \in A, b \in B\}$
$A^n = \underbrace{A \times ... \times A}_{n}$
Die *Potenzmenge* einer Menge M ist die Menge aller Teilmengen von M: $\mathcal{P}(M) = \{A|A \subseteq M\}$

Bsp.: $\mathcal{P}(\{1,2\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$

Definition: Die Mächtigkeit einer Menge A ist die Anzahl ihrer Elemente. Notation: $|A|$

Satz: Es gilt $|A^n| = |A|^n$

Beweis: Nach Def. ist $A^n = \{(a_1, ..., a_n)|a_1, ..., a_n \in A\}$. Um das n-Tupel $(a_1, ..., a_n)$ zu erzeugen, gibt es $|A|$ viele Möglichkeiten. Insgesamt gibt es daher $|A|^n$ Möglichkeiten das n-Tupel $(a_1, ..., a_n)$ auszuwählen.

Bsp.: Eine PIN bestehe aus 6 Ziffern. Mit $A = \{0, ..., 9\}$ ist A^6 die Menge aller PINs. Mit obigen Satz folgt: Die Anzahl aller PINs ist $|A^6| = |A|^6 = 10^6$

Bsp.: In dem Programm

```
1 for (i=1 to n)
2   for (j=1 to n)
3     a[i][j]=i+j;
```

werden alle Paare (i,j) erzeugt. Die Anzahl der Paare ist $|\{1, ..., n\}^2| = |\{1, ..., n\}|^2 = n^2$. Es gibt daher n^2 Schleifendurchläufe.

Satz: $|\mathcal{P}(M)| = 2^{|M|}$

Beweis: Für $M = \{m_1, ..., m_n\}$ identifizieren wir eine Teilmenge $A \subseteq M$ durch das n-Tupel $(a_1, ..., a_n)$ mit $a_k \begin{cases} 0 \text{ für } M_k \notin A \\ 1 \text{ für } m_k \in A \end{cases}$. Nach obigen Satz gibt es $|\{0,1\}^n| = 2^n = 2^{|M|}$ derartige Tupel.

Definition: Für eine n-elementige Menge ist $\binom{n}{k}$ die Anzahl ihrer k-elementigen Teilmengen $(n \geq k \geq 0)$.

Bsp.:
$\binom{n}{0} = 1$, da $\emptyset$ die einzige 0-elementige Teilmenge ist.
$\binom{n}{n} = 1$, da es nur eine n-elementige Teilmenge gibt (die Menge selber).

Mitschrift von
Falk-Jonatan Strube

$\binom{n}{1} = n$, da es n 1-elementige Teilmengen gibt.
$\binom{n}{2} = \frac{n(n-1)}{2}$, denn für das 1. Element gibt es n Möglichkeiten, für das 2. Element $n-1$ Möglichkeiten. Da das Element $\{a,b\} = \{b,a\}$ hierbei doppelt gezählt wird, müssen wir durch 2 teilen.

Definition: Eine Permutation der Folge $1, ..., n$ ist eine neue Anordnung dieser Folge.

Bsp.: Alle Permutationen von $1, 2, 3$ sind $1, 2, 3$; $1, 3, 2$; $2, 1, 3$; $2, 3, 1$; $3, 1, 2$; $3, 2, 1$.

Definition: $n! = 1 \cdot ... \cdot n \qquad 0! = 1.$

Satz: Es gibt $n!$ Permutationen von n Zahlen.

Beweis: Für die 1. Stelle gibt es n Möglichkeiten, für die 2. Stelle $n-1$ usw. Für die letzte Stelle nur noch eine Möglichkeit. Insgesamt also $n \cdot ... \cdot 1 = n!$ Möglichkeiten.

Satz: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ [1]

Beweis: Um aus einer n-elementigen Menge k Elemente auszuwählen, gibt es n Möglichkeiten, um das erste Element auszuwählen, für das zweite Element $n-1$ Möglichkeiten, ..., für das k. Element $n-k+1$ Möglichkeiten, insgesamt daher $n \cdot ... \cdot (n-k+1)$ Möglichkeiten. Da die Reihenfolge, in der diese k Elemente ausgewählt werden, keine Rolle spielt, muss dieses Produkt durch $k!$ geteilt werden.
Daher erhalten wir $\binom{n}{k} = \frac{n \cdot ... \cdot (n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$

4 O-Notation

Mit Hilfe der O-Notation lassen sich obere Schranken für die Laufzeit eines Algorithmus angeben (Abschätzung mit $\leq$, die die maximale Laufzeit eines Algorithmus angibt, bspw. $\leq c \cdot n^2$). Um die Laufzeit eines Algorithmus zu messen, bestimmen wir die Anzahl Schritte und geben mit Hilfe der O-Notation deren Größenordnung in Abhängigkeit der Länge der Eingabe an.

Beispiel: lineare Suche

```
1 int lsearch (int a[], int n, int k) {
2   int i;
3   for (i=0; i<n; i++)
4     if ( a[i] == k) return 1; //gefunden
5   return 0; // nicht gefunden
6 }
```

[1] $\sum_{k=0}^{n} \binom{n}{k} = 2^n$

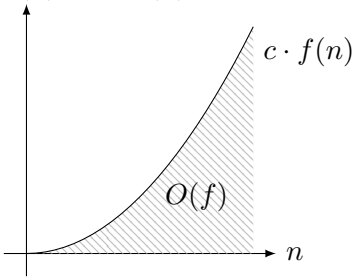
Mitschrift von
Falk-Jonatan Strube

Laufzeit dieser Funktion:

$$\leq \underbrace{c_1 + n \cdot c_2 + c_3}_{g(n)} \overset{\text{Abschätzung}}{\leq} (c_1 + c_2 + c_3) \cdot n = c \cdot n$$

c_1 ... Deklarierung von i
c_2 ... Vergleich der Werte in der Schleife (in n Schleifedurchläufen)
c_3 ... Ausführung return
(Durch den Worst-Case von annähernd unendlich vielen Durchläufen spielen die Konstanten, egal wie groß, keine besonder Rolle mehr und können, wie in der Abschätzung zu sehen, zusammengefasst werden).
Die Laufzeit der linearen Suche liegt in $O(n)$

Definition: Für eine Funktion $f > 0$ ist $O(f)$ die Menge aller Funktionen g, für die gilt: $g(n) \leq c \cdot f(n)$ für ein $c > 0$ für alle großen n.



$2log(n^2 + 1)$
$\leq 2log((1 + 1)n^2)$
$= 2log(2n^2)$
$= 2log(2) + 2log(n^2)$
$\leq (2log2 + 2)log(n^2)$
$= 2(2log2 + 2)log(n)$

Bsp.: $2n^3 - n + 5 \overset{1.)}{\leq} 2n^3 + 5 \leq 7n^3 \in O(n^3)$

1.) $-n$ ist kleiner Null, deswegen ist die rechte Seite ohne $-n$ nachgewiesener Maßen größer (Vorgehensweise Ungleichung aufstellen (siehe auch folgende): weg lassen, was kleiner Null ist; mit n^x o.ä. erweitern, um auszuklammern).

Bsp.:

```
1 for (i=0; i<n-1; i++)
2   for (j=i+1 ; j<n ; j++)
3     if( a[i] == a[j] ) return 1;
4 return 0;
```

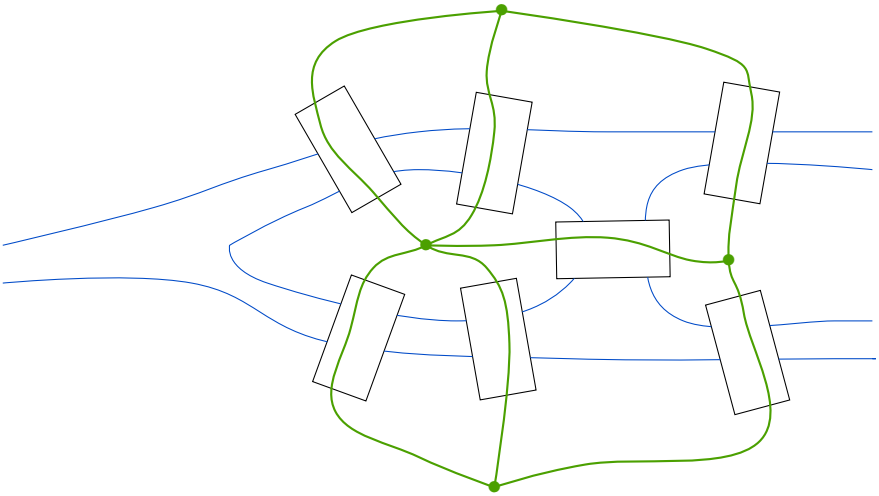
Die if-Anweisung wird höchstens $\binom{n}{2}$ mal durchlaufen. Die Laufzeit ist daher $\leq c_1 \cdot \binom{n}{2} + c_2 \leq (c_1+c_2) \cdot \binom{n}{2} = \frac{c_1 + c_2}{2} \cdot n \cdot (n-1) \leq \frac{c_1 + c_2}{2} \cdot n^2 \in O(n^2)$ (mit c_i ... Zeiteinheiten für Rechenaufwand).

Bsp.: $2 \cdot log(n^2 + 1)$
$\leq 2 \cdot log(n^2(1 + 1))$
$= 2log(2n^2) = 2(log(2) + log(n^2))$

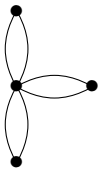
Mitschrift von
Falk-Jonatan Strube

$\leq 2(log(n) + 2log(n))$
$\leq 6log(n) \in O(log(n))$
Schneller mit:
$n^2 + 1 \leq n^3 \Leftrightarrow$
$\frac{1}{n} + \frac{1}{n^3} \leq 1 \Rightarrow$
$0 \leq 1$ für $n \to \infty$
$2 \cdot log(n^2 + 1) \leq 2log(n^3) = 6log(n) \in O(log(n))$

5 Graphen



Gibt es einen Rundweg über alle Brücken?
↷ Abstraktion als Graph:

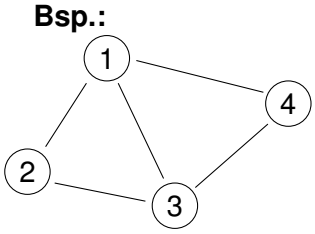


↷ Es gibt keinen Rundweg (⇒ Euler)!

Definition: Ein (ungerichteter) Graph ist ein Paar $G = (V, E)$, wobei

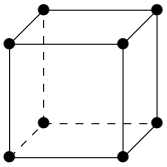
- V die Menge der Knoten und
- E die Menge der Kanten ist, die aus ungeordneten Paaren $\{u, v\}$ von Knoten besteht (also ungerichtet).

Bsp.:

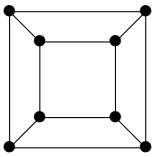


$(\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 4\}, \{3, 4\}\})$
Beispiel:

Mitschrift von
Falk-Jonatan Strube



Die Punkte und Kanten eines 3D-Objektes werden auf einen Graph abgebildet $\Rightarrow$



Definition: Ein Graph heißt *vollständig*, wenn alle Knoten paarweise verbunden sind.

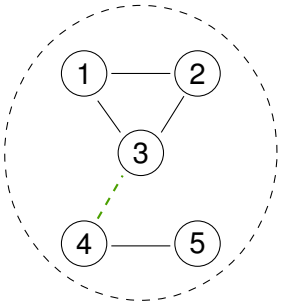
Ein vollständiger Graph mit n Knoten besitzt genau $\binom{n}{2}$ Kanten (jeder Knoten hat den Grad $n-1$).

Definition: Ein Knoten v hat den Grad k, wenn v mit genau k anderen Knoten verbunden ist.
Notation: $deg(v) = k$

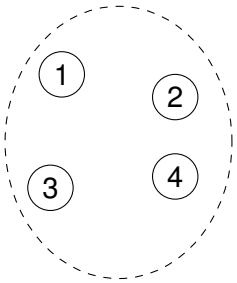
Satz: Für jeden Graphen gilt $\sum_{v \in V} deg(v) = 2|E|$ (Sprich: Die Summe der Grade aller Knoten ist die zweifache Kanten-Anzahl).

Beweis: Wenn wir jede Kante in der Mitte durchschneiden, ist jeder Knoten mit genau $deg(v)$ Hälften verbunden. Die Summe der Knotengrade ist dann die Anzahl der Kantenhälften, und diese ist $2|E|$.

Definition: Ein *Weg* ist eine Folge von Knoten $v_1, ..., v_k$ mit $\{v_l, v_{l+1}\} \in E$ für $l = 1, ..., k-1$. Die Länge dieses Weges ist $k-1$. Ein Weg heißt Kreis, wenn $v_1 = v_k$.
(v_1) —— (v_2) —— ... —— (v_k)
Beispiele:



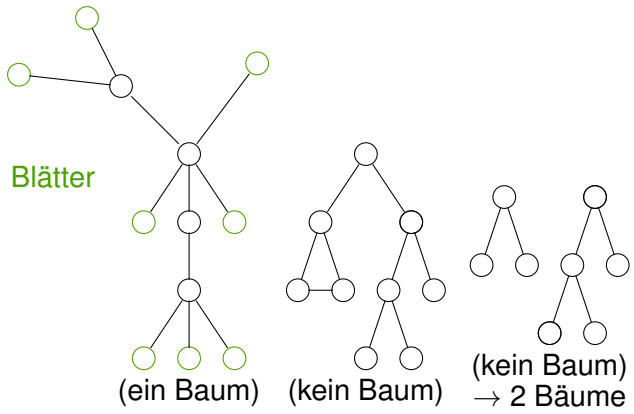
ist ein Graph: $(\{1,2,3,4,5\}, \{\{1,2\},\{1,3\},\{2,3\},\{4,5\}\})$
(der grüne Pfad würde ihn zusammenhängend machen).



ist ein Graph: $(\{1,2,3,4\}, \emptyset\}$

Ein Graph heißt *zusammenhängend*, wenn es für alle Paare von Knoten u, v einen Weg von u nach v gibt.
Ein *Pfad* ist ein Weg, der keinen Knoten mehrfach enthält.

5.1 Bäume



Definition: Ein Baum ist ein zusammenhängender Graph der keine Kreise enthält. Ein Blatt ist ein Knoten v mit $deg(v) \leq 1$ (dem Grad 1, also nur eine Kante hat).
Anmerkung: Auch ein Graph mit nur einem Knoten ist ein Baum - ein Baum der keine Blätter hat.

Satz: Sei $B = (V, E)$ ein Baum. Dann gilt $|E| = |V| - 1$.

Beweis: (Indkution)

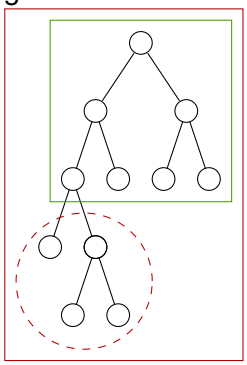
IA: $|V| = 1$: Ein Baum mit nur einem Knoten enthält keine Kanten.

IV: $|E| = |V| - 1$.

IS: $|V| \to |V| + 1$: Sei B ein Baum mit $|V| + 1$ Knoten. B besitzt ein Blatt (siehe Übung). Indem wir dieses Blatt zusammen mit der zugehörogien Kante entfernen, erhalten wir ein Baum B' mit $|V|$ Kanten und nach Induktionsvoraussetzung $|V| - 1$ Kanten. Damit besitzt B $(|V| + 1) - 1$ Kanten.

Definiton: Ein *Wurzelbaum* ist ein Baum mit einem als Wurzel ausgezeichnetem Knoten.

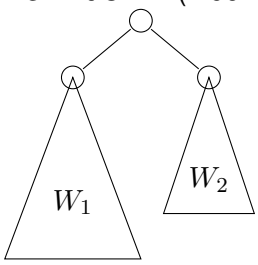
Definition: Ein *binärer Wurzelbaum* ist ein Wurzelbaum, in dem jeder Knoten, der kein Blatt ist, genau zwei Nachfolger besitzt.



perfekt balanciert

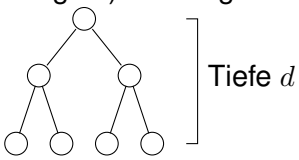
nicht balanciert,
da eine Wurzel länger

Definition: (induktiv)



- Ein einzelner Knoten ist ein binärer Wurzelbaum
- Wenn W_1, W_2 binäre Wurzelbäume sind, dann erhalten wir einen neuen Wurzelbaum, indem die Wurzeln von W_1, W_2 mit einer neuen Wurzel verbunden werden.

Satz: Ein binärer Wurzelbaum mit Tiefe d (d.h. alle Pfade von Wurzel zu einem Blatt haben die Länge d) besitzt genau 2^d Blätter.

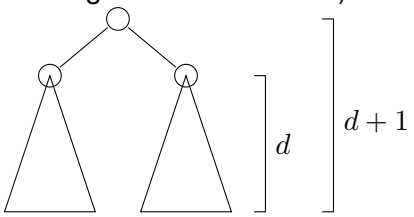


Beweis: (Indkution)

IA: $d = 0$: Ein binärer Wurzelbaum, der nur aus der Wurzel besteht, enthält $2^0 = 1$ Blätter.

IV: $|V| = 2^d$

IS: $d \to d + 1$: Ein binärer Wurzelbaum der Tief $d + 1$ enthält zwei binäre Wurzelbäume (laut vorhergehender Definition) der Tiefe d.



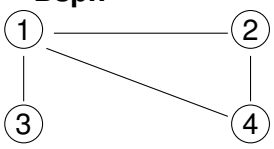
Diese enthalten nach Induktionsvoraussetzung jeweils 2^d Blätter. Folglich besitzt der binäre Wurzelbaum der Tiefe $d + 1$ genau $2 \cdot 2^d = 2^{d+1}$ Blätter.

5.2 Datenstrukturen zur Repräsentation

Es gibt zwei Möglichkeiten, um Graphen darzustellen:

Adjazenzmatrix Für einen Graphen $G = (V, E)$ ist die Adjazenzmatrix eine $|V| \times |V|$-Matrix (a_{uv}) mit $a_{uv} = \begin{cases} 1 & \text{für } \{u, v\} \in E \\ 0 & \text{sonst} \end{cases}$

Bsp.:

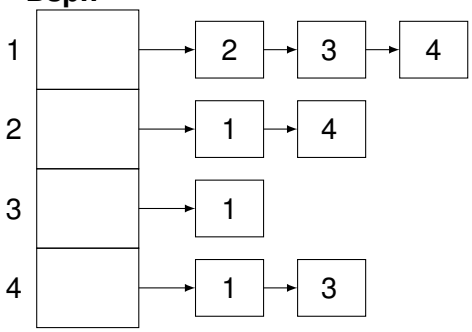


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Adjazenzliste Ein Array hat den Nachteil, dass es nicht in der Länge geändert werden kann. Der Vorteil ist allerdings, dass auf Elemente des Arrays in kurzer Zeit zugegriffen werden kann.
Eine Liste kann wachsen und schrumpfen. Jedes Glied einer Liste verweist auf das nächste. Der Nachteil ist, dass der Zugriff auf Elemente aus der Liste nicht so schnell und einfach ist.

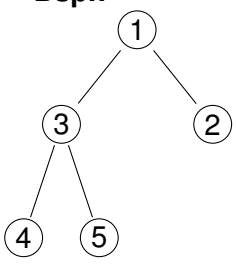
Die Adjazenzliste ist ein Array, das an jeder Position v eine Liste der mit v verbundenen Knoten enthält.

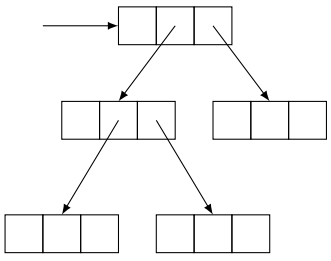
Bsp.:



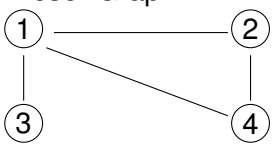
Bäume, insbesondere Binärbäume, lassen sich noch einfacher darstellen: Jeder Knoten wird dargestellt durch eine Datenstruktur, die einen Verweis auf die Nachfolger enthält.

Bsp.:





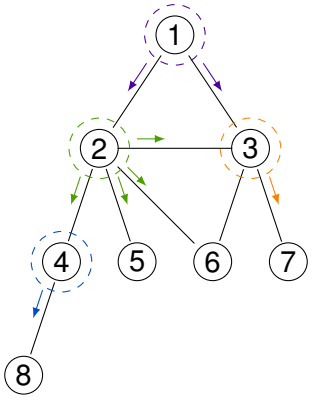
Dieser Graph:



würde dann als $Array(List(2,3,4), List(1,4), List(1), List(1,2))$ dargestellt werden.

5.3 Grundlegende Graphalgorithmen

5.3.1 Breitensuche

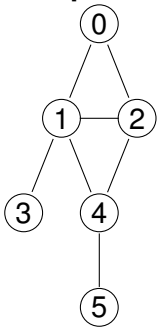


1. Schritt
2. Schritt
3. Schritt
4. Schritt
↷ 8 gefunden

Mit der Breitensuche kann ein Graph systematisch durchsucht werden. Von einem Startknoten ausgehend, besucht die Breitensuche zuerst die dem Startknoten benachbarten Knoten. Anschließend werden die noch nicht besuchten Nachbarn dieser Knoten besucht, usw., bis das Ziel gefunden wurde oder alle Knoten besucht wurden.

```
boolean bfs (node start, node goal){
  for (v ∈ V){
    discovered[v] = false
  }
  queue.enqueue(start)
  discovered[start] = true
  while (¬ queue.isEmpty){
    u = queue.dequeue()
    if (u = goal){
      return true
    }
    else{
      for (v ∈ adj[u]){
        if (¬discovered[v]){
          queue.enqueue(v)
          discovered[v] = true
        }
      }
    }
  }
  return false
}
```

Bsp.:



discovered:

0	1	2	3	4	5
t	✗ t	✗ t	✗ t	✗ t	✗ t

queue:

✗	✗	✗	✗	✗

1.
2.
3.
4.
5.
6. ✓

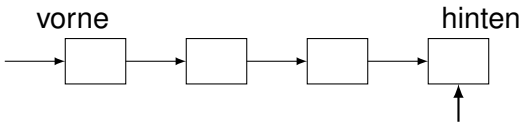
5.3.2 Tiefensuche

Die Tiefensuche lässt sich implementieren...

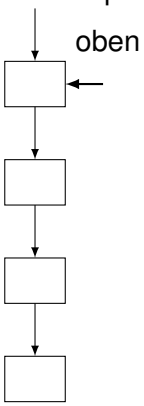
1.) wie die Breitensuche, aber mit einem Stack anstelle einer Warteschlange

2.) rekursiv.

Eine Warteschlange ist eine FIFO (first in, first out) Datenstruktur, die sich implementieren lässt mit einer verketteten Liste, die einen Zeiger auf das letzte Element besitzt.

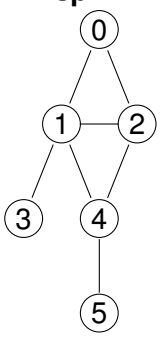


Ein Stack (auch „Keller“) ist eine LIFO (last in, first out) Datenstruktur, die sich durch eine verkettete Liste implementieren lässt.

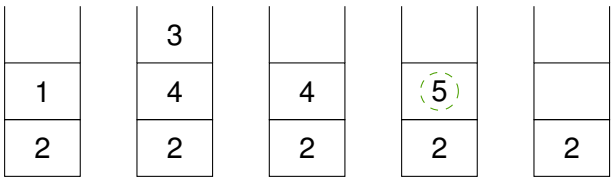



```
boolean tfs (node start, node goal){
  for (v ∈ V){
    discovered[v] = false
  }
  stack.push(start)
  discovered[start] = true
  while (¬stack.isEmpty){
    u = stack.pop()
    if (u = goal){
      return true
    }
    else{
      for (v ∈ adj[u]){
        if (¬discovered[v]){
          stack.push(v)
          discovered[v] = true
        }
      }
    }
  }
  return false
}
```

Bsp.:

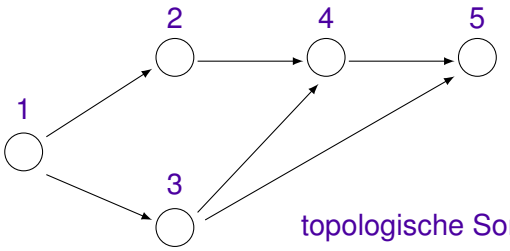


Stack:



Problem bei Breiten und Tiefensuche: Man braucht das Feld „discovered“. Das kann bei großer Anzahl von Knoten ein Problem sein → Speicheraufwändig

5.3.3 Topologisches Sortieren



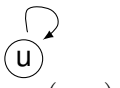
topologische Sortierung

Def.: Ein gerichteter Graph ist ein Paar (V, E) mit $V \neq 0$ und $E \subseteq V \times V$. Die Begriffe Weg, Pfad, Kreis lassen sich entsprechend definieren.

ungerichtet: gerichtet:

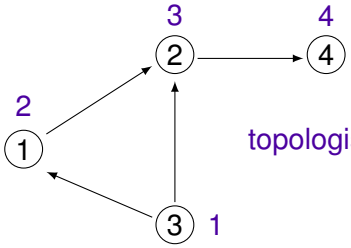
$\{u, v\}$ $(u, v) \in E$

auch möglich:



(u, u)

Def.: Sei $G = (V, E)$ ein gerichteter Graph. Eine *topologische Sortierung* von G ist eine Abbildung $t : V \to \mathbb{N}$ mit $(u, v) \in E \Rightarrow t(u) < t(v)$



topologische Sortierung

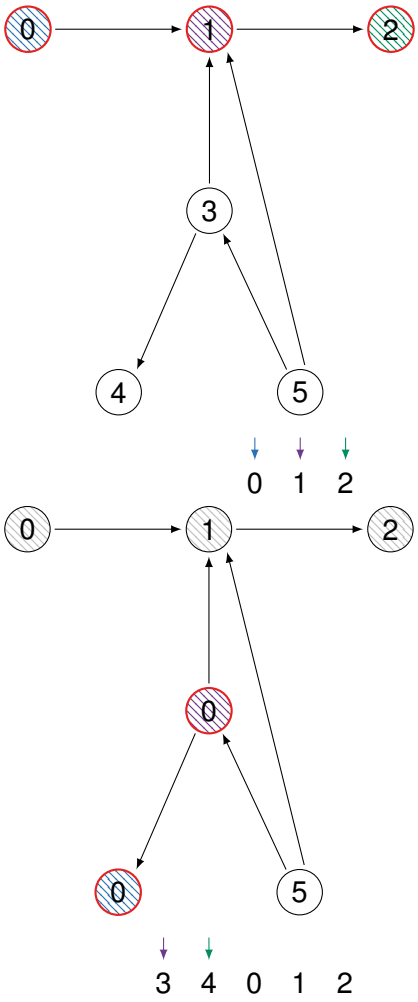
$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (2, 4), (3, 1), (3, 2)\}$

v	1	2	3	4
t	2	3	1	4

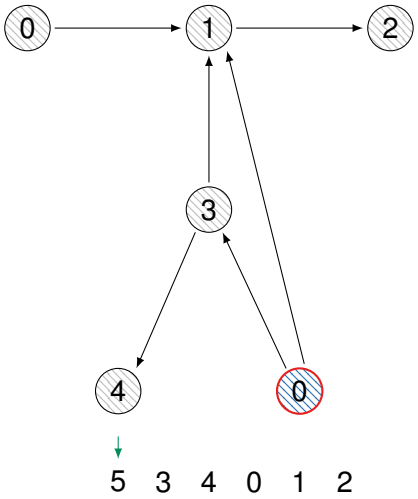
Für einen Kreis oder einen Graph mit einer Schlinge existieren keine topologische Sortierungen.

Eine topologische Sortierung kann durch eine Tiefensuche bestimmt werden.
Alogrithmus TopSort:

```
for v ∈ V
  markiere v mit weiß
for v ∈ V
  tiefensuche(v)

tiefensuche(v){
  v grau: Fehler, Kreis vorhanden
  v weiß: markieren v mit grau
    for( u | (v,u) ∈ E )
      tiefensuche (v)
    markiere v mit schwarz und füge v an den Kopf einer Liste an
}
```



1. grau
2. schwarz
3. schwarz (durch rekursion)
4. schwarz (durch rekursion)



	5	3	4	0	1	2
Endgültige topologische Sortierung:	↓	↓	↓	↓	↓	↓
	0	1	2	3	4	5

Laufzeit dieser topologischen Suche: $O(|V| + |E|)$
Sowohl die Tiefensuche als auch die Breitesuche besitzen eine Laufzeit in $O(|V| + |E|)$.

5.3.4 Suche

Lineare Suche Laufzeit: $O(n)$

Binäre Suche Voraussetzung: Sortiertes Array

0	1	2	3	4	5
1	3	5	6	10	17

Suche nach 4:
in der Mitte starten (abrunden)

$= 4?$
$< 4?$
$> 4?$

> 4, also weiter links gucken

0	1
1	3

$= 4?$
$< 4?$
$> 4?$

> 4, also weiter rechts gucken

1
3

$= 4?$
$< 4?$
$> 4?$

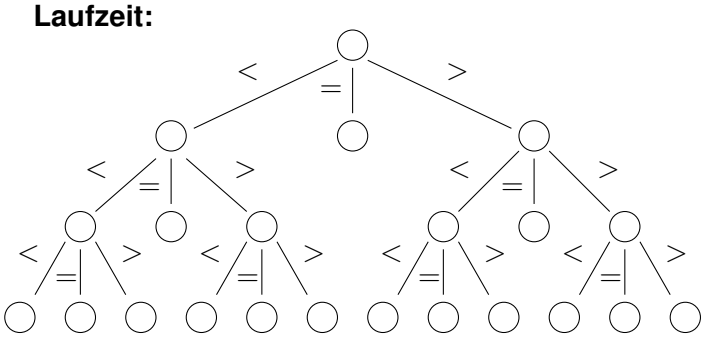
nicht gefunden!

Vorgehen: Es wird die Mitte des Arrays bestimmt (Länge/2 [abgerundet]) und der gesuchten Wert mit dem Wert an dieser Stelle verglichen. Dabei gibt es drei Möglichkeiten:

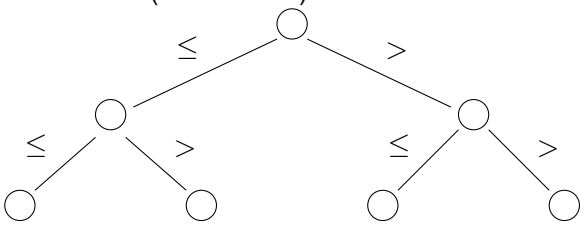
- Gleichheit: Wert gefunden.
- Gesuchter Wert kleiner als Wert an der Stelle im Array: Auf gleiche Weise weitersuchen in der linken Hälfte (ausschließlich des bereits betrachteten Elements).
- Gesuchter Wert größer: Auf gleiche Weise weitersuchen in der rechten Hälfte

Algorithmus wird beendet, wenn der Wert gefunden wurde oder die zu durchsuchende Arrayhälfte keine Elemente mehr enthält.

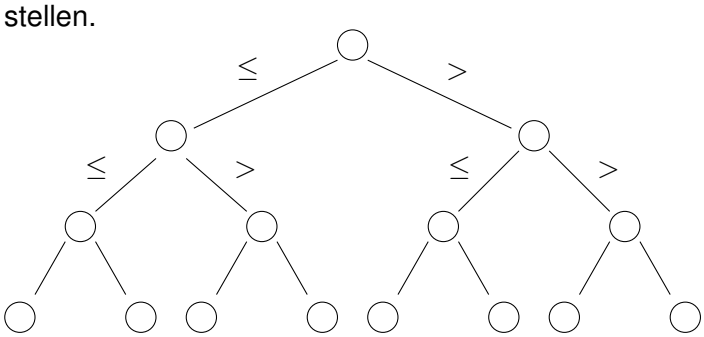
Laufzeit:



Zur Analyse der Laufzeit ändern wir den Algorithmus so, dass nur Vergleiche $\leq$ und $>$ vorgenommen werden. Ferner sei die Länge des Arrays eine Zweierpotenz und das gesuchte Element nicht vorhanden (worst-case).

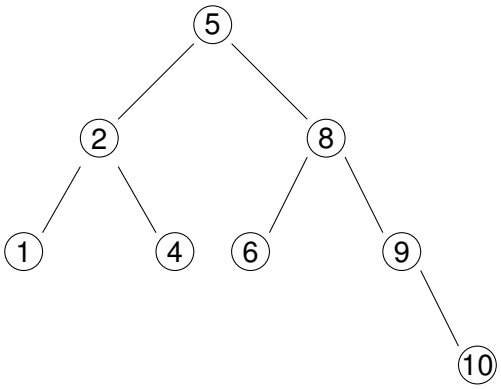


In diesem Fall lässt sich das Verhalten des Algorithmus als vollständiger binärer Wurzelbaum darstellen.

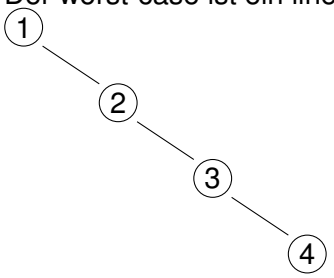


Wenn $n = 2^k$ die Länge des Arrays ist, dann besitzt dieser Wurzelbaum genau 2^k Blätter (die Vergleichen in einen 1-elementigen Array entsprechen). Dieser Binärbaum besitzt daher die Tiefe $k = log_2(n)$. Die Laufzeit der binären Suche liegt daher in $O(log\ n)$ (gilt auch, wenn n keine Zweierpotenz ist).
Um auch dynamische Datenstrukturen effizient durchsuchen zu können, lassen sich binäre Suchbäume nutzen.

Suchbaum Ein *Suchbaum* ist ein binärer Wurzelbaum, in dem jeder linke Teilbaum eines Knotens kleinere Wert und jeder rechte Teilbaum größere Wert als der Vorgängerknoten besitzt.
Beispiel:



(muss nicht zwangsläufig balanciert sein oder 2 Nachfolger haben)
Ein Suchbaum lässt sich ähnlich der binären Suche rekursiv durchsuchen.
Der worst-case ist ein linear entarter Suchbaum:



Die Laufzeit der Suche ist $O(log(n))$, wenn der Baum vollständig balanciert ist und $O(n)$, wenn er linear entartet ist.

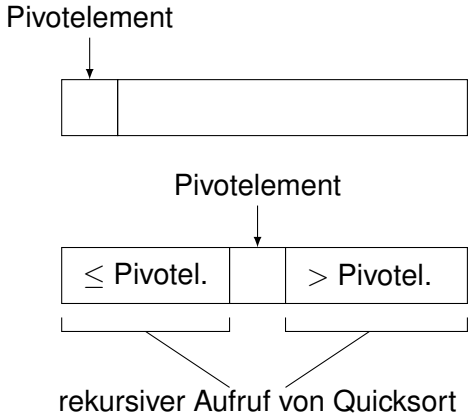
Hashing Prinzip: Mit Hilfe einer Hashfunktion h werden Schlüssel auf eine Position in einem Array (Hashtabelle) abgebildet.
Beispiel für eine Hashfunktion:

```
h(s) = s mod m
```

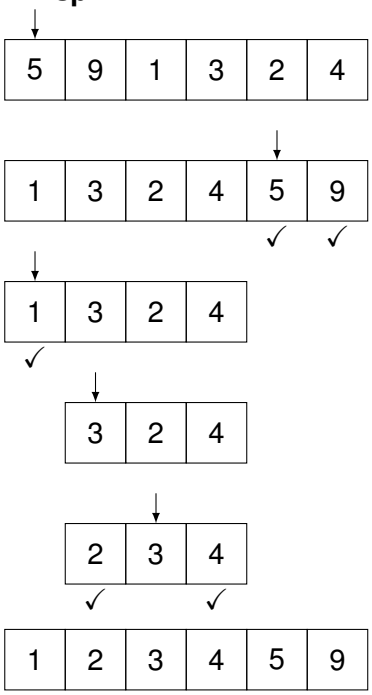
wobei m die Größe der Hashtabelle ist.
Problem: Es können Kollisionen auftreten, d.h. Schlüssel s_1, s_2 mit $h(s_1) = h(s_2)$.
Lösung: Überlauflisten:
An Position $h(s)$ wird eine Liste aller Elemente gespeichert, die diesen Hashwert besitzen.
Unter geeigneten Voraussetzungen besitzt Hashing eine Laufzeit von $O(1)$.

5.3.5 Sortierverfahren

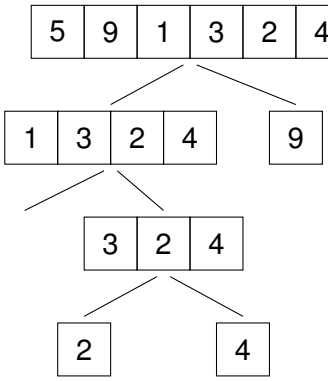
Quicksort partitioniert das zu sortierende Feld anhand eines Pivot-Elements und sortiert rekursiv die dadurch entstandenen Teilfelder.



Bsp.:



Struktur der rekursiven Aufrufe:

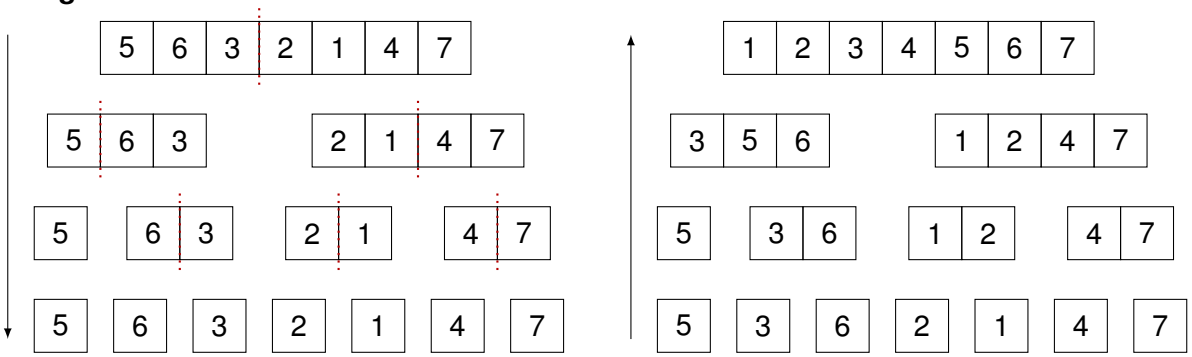


in Scala:

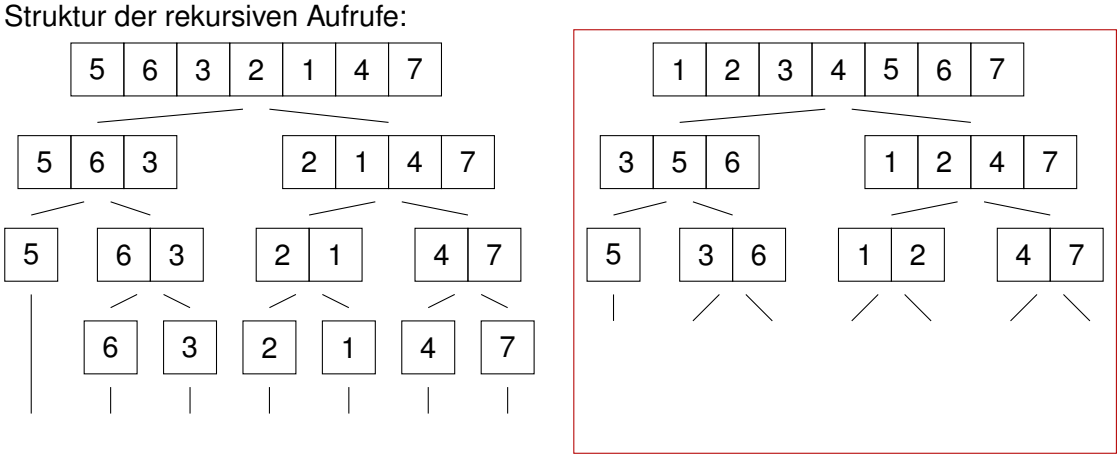
```scala
package alg

object Quicksort{
  def qs(l: List[Int]): List[Int] = l match {
    case Nil => Nil
    case h::t =>
      val (li, re) = t.partition(x=> x <= h)
      qs(li) ::: h :: qs(re)
  }

  def main(args: Array[String]) {
    // println(List(423,2,3,4,67,8,7,12,3,4)))
    // wird folgende Zeile verwendet (zufällige Folge von 10000 Zahlen),
    // ist der Ablauf relativ schnell
    // val l = (for(i <- 0 until 100000) yield scala.util.Random.nextInt).toList
    // sortiert man mit folgender Zeile eine sortierte Liste,
    // dauert es sehr lange oder führt zu einem Fehler (aufgrund zu vieler Rekursionen)
    val l = (for(i <- 0 until 100000) yield i).toList
    val x = qs(l)
  }
}
```

Die Laufzeitanalyse ist schwierig, weil die Länge der Teillisten vom Pivotelement abhängt. Wir betrachten stattdessen ein ähnlichens Verfahren: Mergesort.
Hierbei werden die Listen halbiert, rekursiv sortiert und dann zusammengefügt.

Mergesort

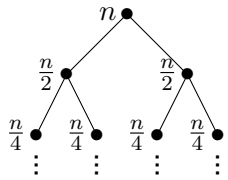


Struktur der rekursiven Aufrufe:



passiert eigentlich in linkem

Laufzeitanalyse: Wir stellen das Verhalten von Mergesort für $n = 2^k$ durch einen Binärbaum dar.



Zum erzeugen der Hälften und dem Zusammenfügen fällt der Aufwand O(|linke Liste| + |rechte Liste|). Dies ist $O(n)$ auf jeder Ebene des Baums. Der Baum hat $n = 2^k$ Blätter und daher die Tiefe k.
Die Laufzeit von Mergesort ist daher $O(n\ log(n))$ (Anzahl der Ebenen n mit LZ auf jeder Ebene).
Mergesort besitzt immer die LZ $O(n\ log(n))$. Dies ist nahe am Optimum. Quicksort besitzt eine durchschnittliche LZ in $O(n\ log(n))$ und eine Worst-Case-LZ in $O(n^2)$.
Da Quicksort mit vorsortierten Listen nicht so gut umgehen kann (wird linear entartet), ist es kein sehr gutes Suchverfahren.

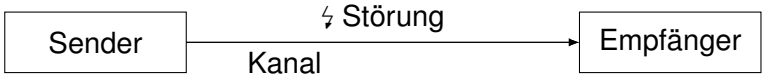
6 Codierungstheorie

Gruppen Erläuternde Beispiele:
$(\mathbb{Z}_2, +)$ (Elemente dieser Menge werden im folgenden *Bits* genannt)
$a + b := a + b\ mod\ 2$
$\Rightarrow 1 + 1 \equiv 0 (mod\ 2)$

inverses Element: $a + a^{-1} = 0 \rightarrow$ hier also: zu 1 ist 1 das inverse Element.

$(\mathbb{Z}_P, +)$ ($\mathbb{Z}_P$: alle Primzahlen)
$7 + 4 \equiv 0(mod\ 11)$
hier bspw. 4 das inverse Element zu 7.

Ausgangslage Codierungstheorie
Problem: Daten können bei der Übertragung verändert werden. Wie können diese Fehler erkannt und ggf. korrigiert werden?



Häufige Lösung zur Fehlererkennung: Prüfsummen.

6.1 Paritätsprüfung

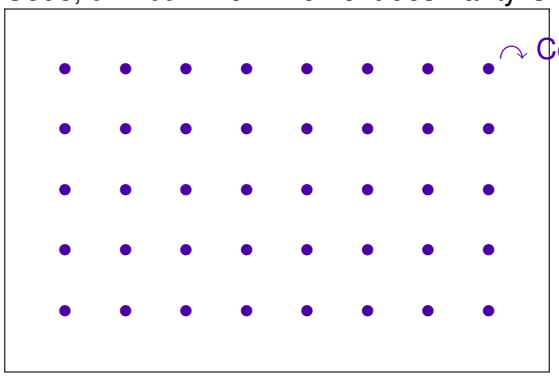
Für eine Folge von Bits $b_1, ..., b_{n-1} \in \{0, 1\}$ wird das *Paritätsbit* $b_n = \left(\sum_{i=1}^{n-1} b_i\right) mod\ 2$ berechnet. Das enthaltene Codewort ist $b_1 ... b_n$

Bsp.: Für die Folge von Bits $0, 1, 1, 0, 1, 0$ ist 1 (Anzahl aller 1er-Bits $mod\ 2$) das Paritätsbit. Das Codewort ist also $0, 1, 1, 0, 1, 0, 1$.

Aus $\sum_{i=1}^{n-1} b_i \equiv b_n (mod\ 2)$ folgt daher (indem auf beiden Seiten $+b_n$ rechnet) $\sum_{i=1}^{n} b_i \equiv 0 (mod\ 2)$.

Die Menge der Wörter des *Parity-Check-Code* lässt sich damit durch $\{(b_1, ..., b_n) \in \{0, 1\}^n \| \sum_{i=1}^{n} b_i \equiv 0 (mod\ 2)\}$ darstellen. $|P_n| = 2^{n-1}$ bezeichnet die Mächtigkeit dieser Menge (alle Bits, bis auf das letzte, können frei gewählt werden).

Bsp.: Für $n = 8$ ist 00110011 (4 Einsen $\Rightarrow 4 mod\ 2 = 0 \Rightarrow$OK) ein Element des Parity-Check-Code, 01110011 kein Element des Parity-Check-Code.



alle Wörter in $\{0, 1\}^n$

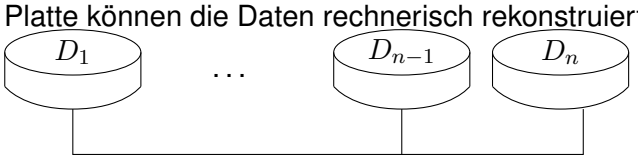
Satz: Der Parity-Check-Code ist 1-fehlererkennend.

Beweis: Seien $b_1...b_n$ ein Codewort und $b'_1...b'_n$ ein Wort, das sich an Stelle k von $b_1...b_n$ unterscheidet. Angenommen, $b'_1...b'_n$ ist ein Codewort. Dann gilt, $0 \equiv \sum_{i=1}^{n} b'_i \equiv \underbrace{\sum_{i=1}^{n} b_i}_{=0} + \underbrace{(b'_k - b_k)}_{=1 \text{(da ungleich)}} \equiv 1$,
Widerspruch. Also ist $b'_1...b'_n$ kein Codewort. Dies kann der Empfänger erkennen.
Bemerkung: Der Parity-Check-Code ist nicht 2-fehlerkorrigierend.
Anwendungen:

- Speicherkontroller, Festplattenkontroller
- Netzwerkprotokolle
- 7-bit-ASCII (Bit 8 als Parität)

Ferner kann das Verfahren zur Rekonstruktion eines verloren gegangenen Bits verwendet werden, wenn die restlichen Bits fehlerfrei sind. Denn aus $0 \equiv \sum_{i=1}^{n} b_i (mod\ 2)$ folgt für $1 \leq k \leq n$:

$b_k \equiv \sum_{i=1, i \neq k}^{n} b_i (mod\ 2)$. Damit kann Bit b_k aus den anderen Bits rekonstruiert werden.
Anwendung: RAID4, RAID5 – Daten und Parität werden auf n Festplatten verteilt. Beim Ausfall einer Platte können die Daten rechnerisch rekonstruiert werden.



Bsp.: $0100x110$ wird korrigiert zu 01001110 (da Parity-Check-Code $mod\ 2 = 0$ ergeben muss).

6.1.1 ISBN-Code

Der ISBN-Code enthält eine Prüfsumme.

Bsp.: $\underbrace{382741826}_{\text{9-stellige Buchnummer}}\ \underbrace{7}_{\text{Prüfziffer}}$
Für die Prüfziffer gilt:
$z_{10} = \left(\sum_{i=1}^{9} i \cdot z_i \right) mod\ 11$
Dabei wird X für den Wert 10 verwendet. Wegen $10 + 1 \equiv 0 (mod\ 11)$ ist 10 das inverse Element zu 1 bezüglich $+$ (d.h. 10 entspricht -1). Aus obiger Gleichung folgt damit:
$0 \equiv \sum_{i=1}^{10} i \cdot z_i (mod\ 11)$
Die Menge der Codewörter ist daher:
$\left\{ z_1, ..., z_{10}) | z_1, ..., z_9 \in \{0, ..., 9\}, z_{10} \in \{0, ..., X\}, \sum_{i=1}^{10} i \cdot z_i \equiv 0 (mod\ 11) \right\}$
$(\mathbb{Z}_{11}, +, \cdot)$ ist ein Köper.
Beispiele:
$1 + 10 \equiv 0$
$2 \cdot 6 \equiv 1 \qquad (6 \mathrel{\widehat{=}} 2^{-1})$
$6 \cdot 2 \equiv 1 \qquad (2 \mathrel{\widehat{=}} 6^{-1})$
damit: $6 \equiv x \underbrace{\Leftrightarrow}_{\text{auf beiden Seiten mit 6 dividieren}} 1 \equiv 2x$

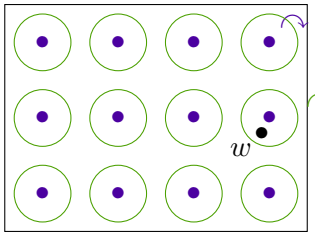
Mitschrift von
Falk-Jonatan Strube

Satz: Der ISBN-Code ist 1-fehler-erkennend.
Beweis: Seien $z_1...z_{10}$ ein Codewort und $z'_1...z'_{10}$ ein Wort, das sich an Stelle k von $z_1...z_{10}$ unterscheidet. Angenommen $z'_1...z'_{10}$ ist ein Codewort. Dann gilt $0(mod\ 11) \equiv \sum_{i=1}^{10} i \cdot z'_i$. Da sich $z'_1...z'_{10}$ an Stelle k von $z_1...z_{10}$ unterscheidet, folgt:
$0 \equiv \sum_{i=1}^{10} i \cdot z_i + k(z'_k - z_k) \Leftrightarrow$
$0 \equiv k(z'_k - z_k)$
Da $k \neq 0$, besitzt k ein bezüglich $\cdot$ inverses Element k^{-1}, womit folgt:
$0 \equiv z'_k - z_k$
und damit
$z_k \equiv z'_k$. Widerspruch!

Satz: Der ISBN-Code erkennt Vertauschungen von Ziffern (Zahlendreher).
Beweis: Seien $z_1...z_{10}$ ein Codewort und $z'_1...z'_{10}$ ein daraus erhaltenes Wort, in dem die Stellen $k, l (k < l)$ vertauscht wurden. Angenommen $z'_1...z'_{10}$ ist ein Codewort. Dann gilt:
$0 \equiv \sum_{i=1}^{10} i \cdot z'_i$
$\equiv 1 \cdot z'_1 + ... + k \cdot z'_k + ... + l \cdot z'_l + ... + 10 \cdot z'_{10}$
$\equiv 1 \cdot z'_1 + ... + k \cdot z_l + ... + l \cdot z_k + ... + 10 \cdot z'_{10}$
$\equiv 1 \cdot z_1 + ... + k \cdot z_l + ... + l \cdot z_k + ... + 10 \cdot z_{10}$
$\equiv \sum_{i=1}^{10} i \cdot z_i + k(z_l - z_k) + l(z_k - z_l)$
$\equiv k(z_l - z_k) + l(z_k - z_l)$
$\equiv k(z_l - z_k) - l(z_k - z_k) \Leftrightarrow$
$k(z_l - z_k) \equiv l(z_k - z_k)$
Da für $z_l \neq z_k$ das Element $(z_l - z_k) \neq 0$ und daher $(z_l - z_k)^{-1}$ existiert, folgt $k \equiv l$. Widerpsruch zu der Annahme $k < l$!

6.2 Fehlerkorrigierende Codes

Idee: Für ein empfangenes Wort w suchen wir ein Codewort v, so dass der Abstand $d(v, w)$ minimal ist (nearest neighbor decoding).



alle Wörter in $\{0,1\}^n$

Def.: Für Wörter $v, v' \in \{0,1\}^n$ ist der *Hamming*-Abstand $d(v, v')$ die Anzahl Stellen, in denen sich v, v' unterscheiden. Der *Minimalabstand* eines Codes C ist $min\{d(v, v') | v, v' \in C,\ v \neq v'\}$.

Bsp.: Parity-Check-Code
$d(0101, 1010) = 4$ (Codewörter unterscheiden sich in allen 4 Stellen)
$d(0101, 0110) = 2$ (Codewörter unterscheiden sich an 2 Stellen)

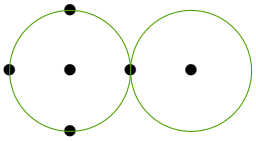
Der Minimalabstand beim Parity-Check-Code ist 2, da beides laut Def. ungleiche Codewörter sind. Der Abstand zweier gültiger Parity-Check-Codewörter ist mindestens 2, somit ist auch der Minimalabstand 2.

Satz:

1.) Ein Code ist k-fehlererkennend gdw. sein Minimalabstand mindestens $k+1$ ist.

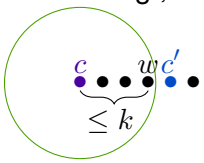
2.) Ein Code ist k-fehlerkorrigierend gdw. sein Minimalabstand mindestens $2k+1$ ist.

Bsp.: Der Parity-Check-Code besitzt den Minimalabstand 2 und ist daher 1-fehlererkennend und 0-fehlerkorrigierend[1].



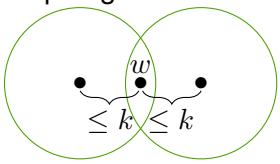
(Kreise überlappen sich. Deswegen können Fehler nicht eindeutig einem Codewort zugewiesen werden)
Beweis:

1.) Fehlererkennung:
($\Rightarrow$) Sei C k-fehlererkennend. Seien ferner $c \in C$ und $w \in \{0,1\}^n$ mit $d(c,w) \leq k$. Da C k-fehlererkennund ist, muss für jedes $c' \in C, c' \neq c$ gelten: $w \neq c'$. Daraus folgt $d(c,c') \geq k+1$, woraus folgt, dass der Minimalabstand von C mindestens $k+1 (\geq k+1)$ ist.

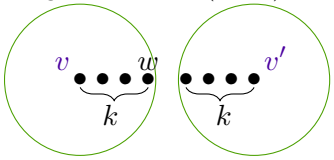


($\Leftarrow$) Sei der Minimalabstand von C mindestens $k+1$. Seien ferner $c \in C$ und $w \in \{0,1\}^n$ mit $d(c,w) \leq k$. Da C den Minimalabstand $\geq k+1$ besitzt, kann w kein Codewort $\neq c$ sein (w liegt innerhalb des Radius k). Daher ist C k-fehlererkennend.

2.) Fehlerkorrektur:
($\Rightarrow$) Wenn der Code k-fehlerkorrigierend ist, darf es nur ein Codewort v mit $d(v,w) \leq k$ für ein empfangenes Wort w geben.



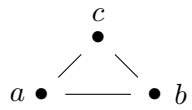
(Die Kugeln dürfen sich nicht überschneiden. Es muss eindeutig bleiben.)
Folglich muss $d(v',w) \geq k+1$ für alle Codewörter $v' \neq v$ gelten, woraus $d(v,v') \geq 2k+1$ folgt.



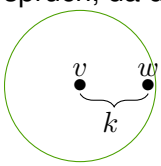
($\Leftarrow$) Für ein Codewort v sei $S(v) = \{w | d(v,w) \leq k\}$. Zu zeigen: aus v, v' Codewörter mit $v \neq v'$ folgt $S(v) \cap S(v') = \emptyset$. Angenommen, es gibt ein $w \in S(v) \cap S(v')$. Dann gilt $d(v,w) \leq k$ und

[1] ein Fehler kann nur korrigiert werden, wenn die Position des fehlerhaften Bits bekannt ist. Deswegen im Allgemein nur 0-fehlerkorrigierend

$d(w, v') \leq k$.



$d(a, b) \leq d(a, c) + d(c, b)$
Daraus folgt mit der Dreiecksungleichung $d(v, v') \leq d(v, w) + d(w, v') = 2k$. Dies ist ein Widerspruch, da der Minimalabstand des Codes mindestens $2k + 1$ ist.



(Wort w kann eindeutig zum Codewort v dekodiert werden.)

Naiver Ansatz zur Fehlerkorrektur: Nachricht mehrfach senden.

Bsp.: $0 \to 000, 1 \to 111$
Der Code $\{000, 111\}$ hat Minimalabstand 3 (Nachricht wurde 3 mal gesendet) und ist daher 1-fehlerkorrigierend.
Nachteil: Platzverschwendung.

Effizienter sind lineare Codes.

6.2.1 Lineare Codes

Die Decodierung durch eine Nearest-Neighbor-Suche im Coderaum ist ineffizient. Um ein effizienteres Verfahren zu erhalten, beschreiben wir Codes durch Matrix-Vektor-Operationen.

Bsp.: Mit $A = (\underbrace{1...1}_{n})$ lässt sich der Parity-Check-Code beschreiben durch $\{w \in \{0,1\}^n | A \cdot w^T = 0\}$ (w, A: Zeilenvektoren) [Veranschaulichung: $(1111) \cdot (0101)^T = 0$]

Def.: Ein Code C heißt *linear*, wenn es eine Matrix A gibt, sodass $C = \{w | A \cdot w^T = 0\}$.
Die Matrix A heißt *Parity-Check-Matrix*.

Satz: Ein binärer linearer Code C ist ein Vektorraum über $\mathbb{Z}_2(+, \cdot)$.
Beweis(Skizze):

1.) Abgeschlossenheit der Vektoraddition:
Zu zeigen: $w_1, w_2 \in C \Rightarrow w_1 + w_2 \in C$ ($+$: Kompomentenweise Addition im Vektorraum)
Aus $w_1, w_2 \in C$ folgt $Aw_1^T = 0, Aw_2^T = 0$ (0 ist der Nullvektor) und daraus $0 = Aw_1^T + Aw_2^T = A(w_1^T + w_2^T) = A(w_1 + w_2)^T$ und daraus $w_1, w_2 \in C$.

2.) Abgeschlossenheit der Mulitiplikation (mit Skalaren):
Zu zeigen: $\alpha \in \mathbb{Z}_2, w \in C \Rightarrow \alpha \cdot w \in C$
Dies gilt, da $\alpha \cdot w = \begin{cases} 0 & \text{für } \alpha = 0 \\ w & \text{für } \alpha = 1 \end{cases}$ und $0, w \in C$.

- Andere Vektoraxiome folgen entsprechend.

Da ein linearer Code ein Vektorraum ist, besitzt er eine Basis[1].

[1] Eine Menge von linear unabhängigen Vektoren, mit der sich alle anderen Punkte erzeugen lassen

Def.: Sei C ein linearer Code. Eine Matrix G, deren Zeilen eine Basis von C bilden heißt *Generatormatrix*.

Bsp.: Sei $G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$. Da die Zeilen von G linear unabhängig sind [1], bilden sie eine Basis eines Vektorraumes. Folglich ist G eine Generatormatrix.

Es gilt $\begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} b_1 & b_2 & b_3 & \sum_{i=1}^{3} b_i \end{pmatrix}$ (alle Linearkombinationen der Matrix G).

Der durch G induzierte Code ist daher $C = \left\{ \begin{pmatrix} b_1 & b_2 & b_3 & \sum_{i=1}^{3} b_i \end{pmatrix} | b_1, b_2, b_3 \in \{0,1\} \right\}$.

Dies ist der Parity-Check-Code der Länge 4.

Wenn C ein linearer Code mit Parity-Check-Matrix A (Achtung $C \neq A$) und Generatormatrix G ist, dann können wir G zum Codieren und A zum Decodieren verwenden.

Bsp.: Wir betrachten den Parity-Check-Code der Länge 4.
Wir wollen die Nachricht $(1\ 0\ 1)$ codieren:

Das entsprechende Codewort ist $\begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$.

Zum Dekodieren berechnen wir $\begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = 0$.

In diesem Fall wurde die Nachricht fehlerfrei übertragen.

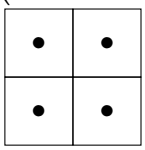
Vorteil dieses Ansatzes: Mit Hilfe einer Generatormatrix lassen sich beliebige weitere Codes definieren. Dies wird insbesondere zur Konstruktion von fehlerkorrigierenden Codes genutzt.

Eine auf diese Weise systematisch konstruierbare Klasse von Codes sind die Hemming-Codes.
Die Generatormatrix eines Hemming-Codes besteht aus allen Vektoren in $\{0,1\}^n$ außer dem Nullvektor. Auf diese Weise lassen sich fehlerkorrigierende Codes erzeugen.
Die Codierung und Decodierung (einschließlich der Fehlerkorrektur) lässt sich für Hamming-Codes sehr effizient durch Matrix-Vektor-Operationen realisieren. Es gibt dazu eine umfangreiche Theorie.
Die Hamming-Codes sind eine weit verbreitete und genutzte Klasse von fehlerkorrigierenden Codes.
Hamming-Codes sind perfekte Codes.

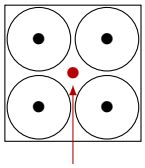
Anhand der Parity-Check Generatormatrix lässt sich die Struktur einer Generatormatrix erkennen: In den ersten Spalten steht die Einheitsmatrix, um im Codewort wiederden Input wieder zu bekommen. Alle weiteren Spalten sind zum verifizieren da.

[1] Nachweis lineare Unabhängigkeit (allgemein):
$\sum a_i v_i = 0$. Vektoren sind dann unabhängig, wenn Gleichung nur lösbar, wenn alle $a_i = 0$.

$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right)$. Dem enstprechend auch dieser (Hemming-)Code $\left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{array}\right)$



perfekter Code



nicht perfekter Code