Aussagenlogik

Atomare Aussage
Aussage, die man nicht in kleinere Aussagen zerlegen kann. Z.b. *Es regnet* oder *Die Straße ist nass*. Mit logischen Verknüpfungen lassen sich daraus neue Aussagen erzeugen.

Ziel:
Aussagen formulieren, die entweder wahr oder falsch sind.

Syntax
Definition

Die Formel der Aussagenlogik sind inaktiv definiert.

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

Um Klammern zu sparen, legen wir folgende Prioritäten fest

Operator	**Priorität**
Negation	höchste
UND / ODER	
Äquivalenz	niedrigste

Bsp
Formel der Aussagenlogik sind: *regnet*, *nass*, x, y

Semantik (Bedeutung)
Definition
Eine Belegung einer Formel F der Aussagenlogik ist eine Zuordnung von Wahrheitswerten wahr(1) oder falsch(0) zu den Atomformeln in F.
Daraus ergibt sich der Wahrheitswert einer Formel, der induktiv definiert ist.

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

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

Bsp
Wenn *regnet* eine Atomformel mit der Bedeutung *Es regnet* ist und *nass* eine Atomformel mit der Bedeutung *Die Straße*
ist nass ist, dann bedeutet *regnet* $\wedge$ *nass* $\rightarrow$ Es regnet und die Straße ist nass

Definition

- $\rightarrow$: Implikation
- $\leftrightarrow$: Äquivalenz

Diese sind definiert durch

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

F	G	$F \to G$	$F \leftrightarrow G$
0	0	1	1
0	1	1	0
1	0	0	0
1	1	1	1

Bsp
Die Formel $regnet \to nass$ bedeutet: Wenn *es regnet* Dann *ist es nass*

- *Regnet* kann wahr oder falsch sein
- Wenn *Regnet* wahr ist, dann *ist es nass*

  Dies ist etwas anderes als $Regnet \wedge Nass$

Bsp
Der Betrag y einer Zahl x lässt sich berechnen durch

```
if(x >= 0)
        y = x
else
        y = -x
```

Die Wirkung dieser Anweisung lässt sich beschreiben durch die Formel:
$(x \geq 0 \to y = x) \wedge (\overline{x \geq 0} \to y = -x)$

Definition
Eine Formel F heißt Tautologie, wenn F unter jeder Belegung wahr ist.
Mit dem Symbol T bezeichnet man eine Tautologie. Mit $\perp$
bezeichnen wir eine Formel, die unter jeder Belegung falsch ist (unerfüllbar).

Definition
Wir schreiben $F = G$, wenn F, G unter jeder Belegung die gleichen Wahrheitswerte besitzt.

Rechenregeln

- $F \vee (G \wedge H) = (F \vee G) \wedge (F \vee H)$
- $F \wedge (G \vee H) = (F \wedge G) \vee (F \wedge H)$
- $\neg(F \vee G) = \neg F \wedge \neg G$ (DeMORGAN)
- $\neg(F \wedge G) = \neg F \vee \neg G$ (DeMORGAN)
- $\neg\neg F = F$
- $F \to G = \neg F \vee G = \neg\neg G \vee \neg F = \neg G \to \neg F$ (logische Kontraposition)

Bsp
Regnet $\to$ Nass = $\neg$ Nass $\to \neg$ Regnet

Definition

- Wir schreiben: $A \Rightarrow B$, wenn $A. \to B$ eine Tautologie ist
- Und: $A \Leftrightarrow B$, wenn $A \leftrightarrow$ eine Tautologie ist

Beweistechniken

Ein Beweis ist eine logische Folgerung aus Aussagen, die bereits als wahr bekannt sind.
Bekannte Wahrheiten ⇒ Folgerungen ⇒ Daraus ergeben sich neue, bekannte Wahrheiten ⇒ Neue Folgerungen

Direkter Beweis

Bsp

Wenn $a \in \mathbb{Z}$ eine gerade Zahl ist, dann ist auch a^2 eine gerade Zahl Um dies zu beweisen, benötigen wir eine Definition für den Begriff "gerade ZahL" $\rightarrow$ Eine Zahl $n \in \mathbb{Z}$ heißt gerade, wenn es ein $k \in \mathbb{Z}$ mit der Eigenschaft $n = 2k$ gibt.

Beweis

a ist gerade $\Rightarrow$ es gibt ein $k \in \mathbb{Z}$ mit $a = 2k \Rightarrow a^2 = (2k)^2 = 2 * 2k^2$ für $k \in \mathbb{Z} \Rightarrow a^2$ ist gerade.

Beweis durch Widerspruch

Durch einen Beweis durch Widerspruch wird eine Aussage beweise. Gezeigt wird, dass die Annahme "A ist falsch" zu einem Widerspruch führt.

Bsp.

Die Zahl $\sqrt{2}$ ist irrational.

Definition

Eine Zahl x heißt irrational, wenn es Zahlen $q, x \in \mathbb{Z}$ gibt mit $x = \frac{p}{q}$ wenn a^2 gerade ist, dann ist a gerade.

Beweis
Angenommen $sqrt2$ ist rational. Dann gibt es $p, q \in \mathbb{Z}$ mit $\sqrt{2} = \frac{p}{q}$.
Weiterhin können wir annehmen, dass p, q Teilerfremd sind (durch Kürzen immer möglich!). Durch quadrieren folgt $2q^2 = p^2$. Damit ist p^2 gerade und nach obigem Hilfssatz auch p.
$p = 2k$
Daraus folgt, dass p^2 durch 4 Teilbar ist und damit auch $2q^2$. Daraus folgt, dass q^2 gerade ist. Nach Hilfs-
satz ist damit q gerade. Damit sind p und q nicht Teilerfremd. Widerspruch!

Beweis durch Fallunterscheidung
Damit wird eine Aussage A bewiesen, indem eine Aussage F
(der Fall, nach dem unterschieden wird) gezeigt wird. $F \Rightarrow A$ sowie $\neg F \Rightarrow A$

Weitere Beweistechnik: **Schubfachprinzip**
Wenn $m > n$ Gegenstände auf n Fächer verteilt werden, gibt es mindestens 1 Fach, in dem 2 Gegenstände liegen.

Bsp
In jeder Gruppe aus $n \geq 2$ Personen gibt es zwei, die die gleiche Anzahl Personen aus dieser Gruppe kennen.
Beweis

- Es gibt eine Person, die alle anderen kennt. Dann kennt jede der n Personen $1 \leq k \leq n-1$ andere Personen aus dieser Gruppe
- Es gibt keine Person die alle anderen kennt. Dann kennt jede der n Personen $0 \leq k \leq n-2$ andere Personen aus dieser Gruppe

In beiden Fällen folgt aus dem Schubprinzip: Es gibt zwei Personen, die die gleiche Anzahl Personen aus dieser Gruppe kennen. Damit ist diese Behauptung bewiesen.

Logische Begründung

$$(F \to A) \land (\neg F \to A) \to A$$

$$\text{Wenn } A = 1: \ \neg\neg F = F$$

$$(\neg F \lor A) \land (F \lor A) \to (\neg F \lor 1) \land (\neg\neg F \lor 1) = A \lor (\neg F \land F) \to A$$

$$A \lor \bot \to A = A \to A = \neg A \lor A = A \text{ Tautologie}$$

Kombinatorik

Geordnete Menge
Für eine Menge $\mathbb{A}$ ist $A^n = \{a_1, ..., a_n | a_1, ..., a_n \in \mathbb{A})\}$

Es gilt
$|A^n| = |A^1| * |A^2| * ... * |A^n|$

Bsp
Zahlenschloss mit 4 Stellen und den Ziffern 0..9

- Die Menge der Ziffern $A = \{0..9\}$
- Menge aller Zahlenkombinationen:
  $A^4 = \{(0,0,0,0), (0,0,0,1)...(9,9,9,9)\}$
- Anzahl der Kombinationen: $|A|^4 = 10^4$

Ungeordnete Mengen
$(1,2) \neq (2,1) \{2,1\} = \{1,2\}$
Die Anzahl der Permutationen (Anordnung) von n Elementen bezeichnen wir mit $n!$.

Es gilt:

$$n! = \underbrace{n}_{\text{n Möglichkeiten für 1 Stelle}} * \underbrace{(n-1)}_{\text{2Stelle}} * ... * 1$$

Die Anzahl der aller k-Elementigen Teilmengen einer n-elementigen Menge bezeichnen wir mit $\binom{n}{k}$

Bsp
$A = \{1, 2, 3\}$ Wie viele Mengen gibt es mit?

- 0
- 1
- 2
- 3 Elemente?

$$\binom{3}{0} = 1, \binom{3}{1} = 3, \binom{3}{2} = 3, \binom{3}{3} = 1$$

Es gilt

$$\binom{n}{k} = (n * (n-1) * ... * \frac{n-k+1}{k!}$$

für $0 \leq k \leq n$

$\binom{49}{6}$ Möglichkeiten

O-Notation

Ziel:
Angaben über die Laufzeit oder Platzverbrauch von Algorithmen oder Datenstrukturen zu machen

Algorithmus:
Lösen eines Problemes in endlich vielen Schritten

Motivation
Algorithmus lineare Suche: In einem Feld nach einem Wert suchen. Dazu wird das Feld von links nach rechts durchsucht. Dieser Algorithmus lässt sich in C implementieren

Listing 1: Lineare Suche

```
1
2  int lin_search(int val, int a[], int a_length)
3  {
4          int i;
5          for(i=0; i<a_length; i++)
6          {
7                  if(a[i] == val)
8                  return 1;
9          }
10
11         return 0;
12 }
```

Eine Zeitmessung ist nicht geeigent, um die Laufzeit eines Algorithmuses anzugeben. Stattdesen zählen wir die Anzahl elementarer Anweisungen (z.B. Zuweisungen, Vergleichen...).
Diese können jeweils in beschränkter Zeit ausgeführt werden.
Sei daher $c > 0$, so dass die Laufzeit dieser $ElementrareAnweisung \leq c$ ist. Die Laufzeit des Algorithmus *lineare Suche* ist dann eine Funtkion $g(n)$, wobei n die Länge des Feldes ist, so dass $g(n) \leq n * c + c$

Defintion
Für eine Funktion $f \geq 0$ ist $O(f)$ die Menge aller Funktionen g mit
$O \leq g(n) \leq c * f(n)$ für eine Konstante $c > 0$ und alle hinreichend großen
$n \in N$

$O(f) = \{g \mid$ es gibt ein $n_0 \in \mathbb{N},$ so dass für alle $n > n_0$ und c > 0 gilt: $0 \leq g(n) \leq c*f(n)\}$

$g(n) \leq c * f(n)$ für große $g \in O(f)$

Beispiel:
$17n^2 + 3n + 5log(n)$
$17n^2 + 3n + 5log(n) \in O(n^2)$ da
$17n^2 + 3n + 5log(n) \leq 17n^2 + 3n^2 + 5n^2 = 25n^2$ Konstante $c = 25$ für $n > 0$

$\longrightarrow n^2$ dominiert, d.h. n und log werden nicht beachtet
Ziel ist es, eine möglichst gute und einfache Abschätzung anzugeben.

Typische Laufzeiten
$O(1) \longrightarrow$ Konstante Laufzeit
$O(n) \longrightarrow$Lineare Laufzeit

Listing 2: Lineare Laufzeit

```
1
2
3         for(i=0; i<n; i++)
4         {
5                 a[i] = b[i] + i;
6                 c[i] = b[i] - i;
7                 d[i] = 0;
8         }
```

$g(n) = n * (c_1 + c_1 + c_2) \leq n * 3 * max(c_1, c_2) \leq n * c$ also $g \in O(n)$

$O(2^n) \longrightarrow$ Typisch für Brute-Force-Algorithmen

F = Formel des Aussagenlogik mit **n** Atomformeln. Es gibt dann 2^n
Belegungen für die Formel F.

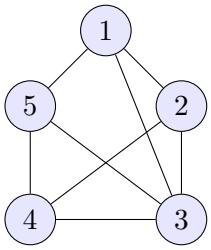
Graphentheorie

13.11.2013

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

- V die Menge der Knoten ist
- E ist die Mengen der Kanten, die aus ungeordneten Paaren $\{u, v\}$ von Knoten besteht

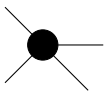
Bsp.
$G = (\{1, 2, 3, 4, 5\}, \{\{1, 2\}\{1, 3\}\{1, 5\}\{2, 3\}\{2, 4\}\{3, 4\}\{4, 5\}\{3, 5\}\}$



Definition
Ein Graph heißt Vollständig, wenn alle Knoten paarweise verbunden sind

Folgerung
Ein vollständiger Graph mit n Knoten besitzt $\binom{n}{2}$ Knoten

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



Satz (Handshake Lemma)
Für jeden Graphen (V, E)gilt: $\sum_{v \in V} deg(v) = Z|E|$

Beweis
Wenn wir jede Kante in der Mitte durchschneiden, ist jeder Knoten v mit genau $deg(v)$ vielen hälften verbunden.
Also ist $deg(v)$ gleich der Anzahl der Kantenhälften und diese ist $2|E|$.

Wege und Kreise

Definition
Ein Weg (von v_1 nach v_k) ist eine endliche Folge von Knoten $v_1, ..., v_k$ mit $\{v_l, v_{v+1}\} \in E$ für $l = 1, k = 1$

Ein Weg heißt *Kreis*, wenn $v_1 = v_2$ (Wenn Weg und Start identisch sind)

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 keine Knoten mehrfach enthält.

Bsp
Ein Hamilton-Kreis ist ein Kreis, der jeden Knoten genau einmal enthält

Für einen Graphen G ist ein *Eulerkreis* ein Kreis, der jede Kante genau einmal erhält.
Satz
Ein Zusammenhängender Graph besitzt einen Euler-Kreis, genau dann, wenn der Graph aller
Knoten gerade ist.
Im Beispiel des *Königsberger Brücken* Problem gibt es keine Lösung, da nicht alle Knoten gerade sind.

Bäume

Definition
Ein *Baum* ist ein zusammenhängender Graph, der keine Kreise enthält.
Ein *Blatt* ist ein Knoten v mit $deg(v) \leq 1$
Für alle u, v bedeutet, es gibt Weg $u \to v$

Satz
Sei $G = (V, E)$ ein Baum. Dann besitzt $G|V| - 1$ Kanten.

Beweis Induktion nach $n = |V|$
$n = 1$: Ein Baum mit 1 Knoten besitzt 0 Kanten

$n \to n + 1$: Sei G ein Baum mit $n + 1$ Knoten. G besitzt ein Blatt (s. HA). Wenn wir in G ein Blatt zusammen mit der zugehörigen Kante entfernen, erhalten wir einen Baum G' mit n Knoten und nach Induktionsvoraussetzung $n - 1$ Kanten. Wenn wir die abgerissene Kante wieder
hinzufügen, erhalten wir den Baum G und dieser besitzt $n-1+1 = n$ Kanten

Definition
Ein Binärer Wurzelbaum ist ein Baum, in dem jeder Knoten, der kein Blatt ist, genau zwei Nachfolger besitzt. Ferner ist genau ein Knoten als Wurzel ausgezeichnet.

Satz
Sei B ein Binärer Wurzelbaum in dem jeder Pfad von
der Wurzel zu einem Blatt die Länge k hat. Dann besitzt B genau 2^k Blätter.

Beweis Induktion nach k
$k = 0$: Ein Binärbaum der nur aus der Wurzel besteht, besitzt $2^0 = 1$ Blatt.

$k \to k + 1$: Wir betrachten einen Binärbaum in der Tiefe $k + 1$. Jeder der beiden Teilbäume, die sich unter der Wurzel befinden sind selbst binäre Wurzelbäume. Da diese jeweils die Tiefe k besitzen, folgt aus der Induktionsvoraussetzung, dass diese Teilbäume jeweils 2^k Blätter besitzten.

Folglich besitzt der Binärbaum der Tiefe $k+1 \rightarrow 2 * 2^k = 2^{k+1}$ Blätter.

Algorithmen

Ein Algorithmus ist ein Verfahren, um ein Problem in einer endlichen Anzahl von Schritten zu lösen.
Ein Algorithmus kann in einer Programmiersprache implementiert werden.

Suchverfahren
Bereits behandelt: **Lineare Suche** (Daten sind hintereinander angeordnet, man sucht, indem man von links nach rechts alles durchsucht)
Laufzeit: O(n)

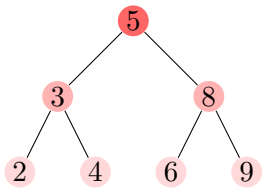
Binäre Suche: Vorausgesetzt wird ein sortiertes Array.
Bei der binären Suche wird nach einem Wert x gesucht, indem zunächst x mit dem Wert in der Mitte des Arrays verglichen wird. Wenn der Wert gefunden wurde, ist der Algorithmus fertig. Sonst wird auf gleiche Weise weiter gesucht in der

- linken Hälfte des Arrays, wenn der gesuchte Wert kleiner als der Wert in der Mitte des Arrays
- rechte Hälfte, wenn der gesuchte Wert größer ist als der Wert in der Mitte des Arrays

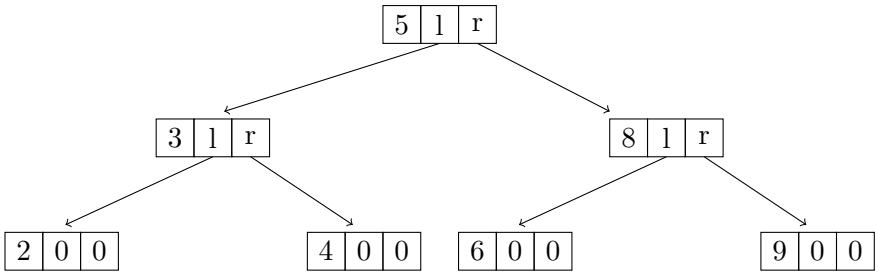
Analyse der Laufzeit: Wir stellen das Verhalten der binären Suche bei erfolgloser Suche als Binärbaum dar. Ferner nehmen wir $n = 2^k$ an. Dabei entsprechen die Knoten, die von der binären Suche ausgeführte Vergleiche, die Kanten dem Weitersuchen im linken bzw, rechten Teilarray. Da in jedem Durchlauf des Algorithmus ein konstanter Aufwand für den Vergleich und das Bestimmen der Teilarrays entsteht, ist die Laufzeit *O(Anzahl Durchläufe)*. Die Anzahl ist die Länge des längsten Pfades von der Wurzel zu einem Blatt in nebenstehendem Binärbaum. Da jedes Blatt

einen Teilarray mit einem Element entspricht, gibt es genau n Blätter. Aus dem oben bewiesenen Satz folgt, dass dieser Binärbaum daher die Tiefe $k = \log(n)$ besitzt. Die Laufzeit der binären Suche liegt folglich in **O(log(n))**

Binäre Suchbäume
Um auch in dynamischen Datenstrukturen zu suchen, lassen sich Suchbäume verwenden. Ein Suchbaum ist ein Binärbaum für die gilt: Jeder in einem Knoten gespeicherter Wert ist größer als alle Werte im linken Teilbaum und kleiner als alle Werte im rechten Teilbaum.



Implementierung in C



Zur Verwaltung eines binären Suchbaumes sind folgende Funktionen nötig.

- **Suche nach einem Wert**

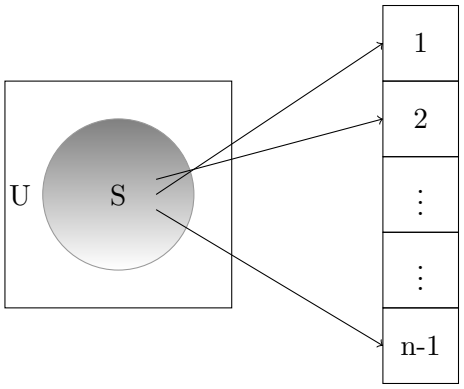
  Dazu wird der Suchbaum, beginnend an der Wurzel, rekursiv durchsucht
  Die Laufzeit ist von der Tiefe des Baumes abhängig. Bei einem vollständig balancierten Baum liegt diese in **O(log n)**, bei einem linear entarteten Baum in **O(n)**.

- **Wert hinzufügen**

  Dazu wird der Baum wie oben durchsucht. Wenn der Wert bereits vorhanden ist, wird die Funktion beendet. Ansonsten wird ein Blatt mit dem neuen Wert hinzugefügt unter dem zuletzt von der Suche besuchten Knoten (links oder rechts).
  Die Laufzeit ist wie bei der Suche (O(log n) bzw. O (n))

- **Wert entfernen**
  Kompliziert

Hashing
Gegeben: Eine Menge U von potenziellen Schlüsseln (z.B. Artikelnummer) und eine Menge $S \subseteq U$ von zu verwaltenden Schlüsseln. Zur Verwaltung der Datensätze wird eine Hashfunktion h von $U \to T$ verwendet, die in eine Hashtabelle T abbildet.



Wenn wir annehmen, dass h injektiv wäre, dann lassen sich folgende Operationen einfach implementieren.

- **Suche:**
  if (T[h(s)] > 0)

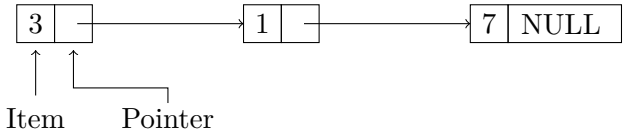
- **Einfügen**
  T [h(s)]++

- **Löschen**
  T[h(s)] = 0

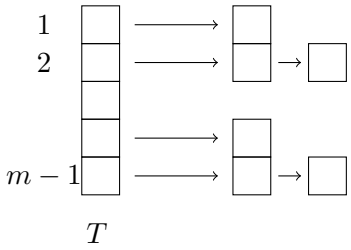
Wegen $|U| >> |S|$ kann h aber nicht injektiv sein (Schubfachprinzip).
Folglich sind Kollisonen möglich, d.h. zwei unterschiedliche Schlüssel $s, s‘$ können den gleichen Hashwert ($h(s) = h(s‘)$) besitzen.

Einfache Lösung zur Behandlung von Kollisionen: Überlauflisten
Eine Liste ist eine dynamische Datenstruktur, bei der jedes
Element der Liste einen Verweis auf ein nachfolgendes Listenelement enthält.

Implementierung in C durch structs und Zeiger



Laufzeit Zugriff auf ein Listenelement: **O(n)**
Um Überlauflisten zu verwenden, verwenden wir eine Tabelle von Listen.

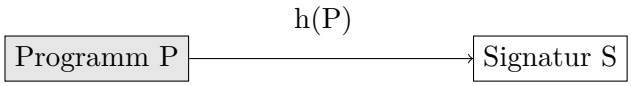


An jeder Position der Hashtabelle werden die dort gespeicherten Einträge in einer Liste veraltet. Wenn die Hashtabelle nicht zu stark befüllt ist (z.B. Anzahl gespeicherter Elemente

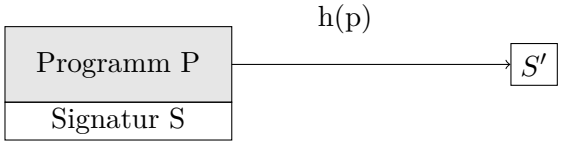
m = $\frac{1}{2}$), dann sind alle Hashoperationen im Mittel in der Zeit **O(1)** möglich.

Mögliche Hashfunktion
h(s) = s mod m

Weitere Anwendungen von Hashing Digitale Unterschrift
Prinzip: Ein Dokument oder ein Programmtext wird als (sehr große) Zahl betrachtet, z.B. indem die Zeichen (Bytes) hintereinander geschrieben als Zahl zu einer geeigneten Basis betrachtet wird. Auf diese Zahl wird eine Hashfunktion angewendet. Der Hashwert ist die Signatur des Dokuments.



Damit lässt sich prüfen, ob das Programm P zu der Signatur S passt



Für $S = S'$ wurde P nicht verändert. Für $S \neq S'$ ist P fehlerhaft (mögliche Fehler bei der Datenübermittlung, P wurde manipuliert..

Anwendung

- Integritätsprüfung von Programmen in der Linux-Paketverwaltung
- Schutz vor Veränderung von Hardware oder Systemeinstellungen (XBox, Motorsteuergeräte, Verbinden von SW-Installation)

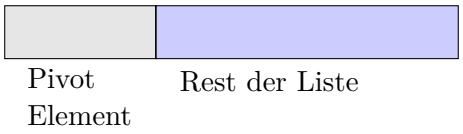
Notwendig ist dazu eine möglichst Kollisionsresistente Hashfunktion. Denn wenn ein Angreifer einen Schadcode P' erzeugen kann mit $h(P) = h(P')$, dann kann dies durch Prüfen des Hashwertes nicht festgestellt werden.

Sortierverfahren

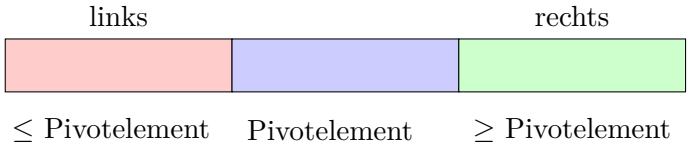
Naive Sortierverfahren haben eine Laufzeit von $O(n^2)$. Ein besseres Verfahren ist Quicksort.

Quicksort führt rekursiv zwei Schritte aus:

- Ein beliebiges Element des zu sortierenden Felds wird als Pivotelement ausgewählt. Bei Listen wird das erste Element ausgewählt, weil der Zugriff darauf in Zeit $O(1)$ möglich ist

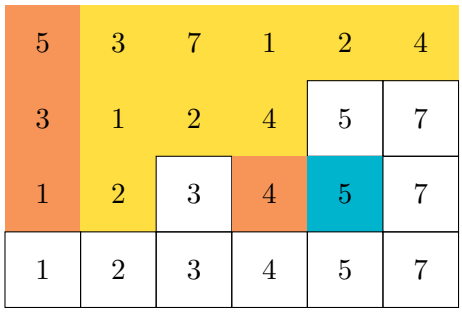


- Die Elemente werden so umgeordnet, dass die Elemente $\neq$ Pivotelemnt vorne, die Elemente $>$ Pivotelement hinten und dazwischen des Pivotelement stehen



- Quicksort wird rekursiv für die Listen l, r aufgerufen, bis die Liste leer ist.

Beispiel



Laufzeitanalyse
Die Laufzeitanalyse von Quicksort ist schwierig, da die Teilliste l, r unterschiedliche Längen haben können. Wir betrachten dazu ein ähnliches Sortierverfahren: Mergesort

Mergesort
Mergesort sortiert wie folgt:

- Die zu sortierente List (bzw. das Array) wird halbiert. Es entstehen Teillisten l, r
- Die Teillisten l, r werden rekursiv sortiert, bis sie leer oder die Länge 1 haben.
- Die sortierten Teillisten werden zu einer sortierten Liste zusammengefügt (Merge-Operation)

Bsp.