Scriptum Grundlagen der Informatik

Markus Klemm.net

WS 2013/2014

Inhaltsverzeichnis

1 Aussagenlogik

Ziel Aussagen formulieren die entweder wahr oder falsch sind. Eine atomare Aussage ist eine Aussage, die sich nicht in kleinere Bestandteile zerlegen lässt, z.B.: es regnet oder die Straße ist nass. Mit logischen Verknüpfungen lassen sich daraus neue Aussagen erzeugen.

Jede atomare Aussage ist eine Formel der Aussagenlogik. Diese heißen Atomformeln. Atomformeln bezeichnen wir mit Kleinbuchstaben oder Wörtern 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
$\neg$	höchste
$\wedge\vee$	
$\Rightarrow\leftrightarrow$	
$\equiv$	Niedrigste

Bsp: Formeln der Aussagenlogik sind : regnet, nass, $x, y, x\wedge y, \neg X, x\vee y$. Keine Formeln der Aussagenlogik: $\wedge x, x \wedge \vee y$

1.1 Semantik

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 wahr genau dann, wenn F wahr oder G wahr sind. Die Formel $\neg F$ ist wahr genau dann, wenn F falsch ist. Die Formel

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

Bsp: Wenn regnet eine Atomformel mit der Bedeutung „es regnet ist“ und nass eine Atomformel mit der Bedeutung „die Straße ist nass“, dann bedeutet $regnet \wedge nass$ „es regnet und die Straße ist nass“. Wenn regnet, nass mit wahr belegt werden, ist auch $regnet \wedge nass$ wahr.

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

$$F \Rightarrow G = \neg F \vee G$$
$$F \leftrightarrow G = (F \Rightarrow G) \wedge (G \Rightarrow F)$$

Bsp: Der Beitrag y einer Zahl x lässt sich berechnen durch

```
1 if (x>=0)
2 {
3     y=x
4 }
5 else
6 {
7     y=-x
8 }
```

Die Wirkung dieser Anweisung lässt sich beschreiben durch die Formel

$$(x \geq 0 \Rightarrow y = x) \wedge (\neg(x \geq 0) \Rightarrow y = -x)$$

Definition Eine Formel F heißt Tautologie, wenn F unter jeder Belegung wahr ist. Mit dem Symbol T bezeichnen wir 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 besitzen.

Rechenregeln

$$F \vee (G \wedge H) = (F \vee G) \wedge (F \vee H)$$
$$F \wedge (G \vee H) = (F \wedge G) \vee (F \wedge H)$$
$$\neg(F \vee G) = \neg F \wedge \neg G$$
$$\neg(F \wedge G) = \neg F \vee \neg G$$
$$\neg\neg F = F$$

Beispiel (logische Kontraposition)
$F \Rightarrow G = \neg F \vee G = \neg\neg G \vee \neg F = \neg G \Rightarrow \neg F$

Definition Wir schreiben $A \Rightarrow B$, wenn $A \rightarrow B$m eine Tautologie ist und $A \Leftrightarrow B$, wenn $A \leftrightarrow B$ eine Tautologie ist.

Beweistechniken Ein Beweis ist eine logische Folgerung aus Aussagen die bereits als wahr bekannt sind.

Direkter Beweis Beispiel:

Satz: 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“ . Dazu Def: Eine Zahl $n \in \mathbb{Z}$ heißt gerade, wenn es ein $k \in \mathbb{Z}$ gibt, mit $n = 2k$

Beweis: a gerade $\Rightarrow$ es gibt ein $K \in \mathbb{Z}$ mit $a = 2k \Rightarrow a^2 = (2k)^2 = 2 \cdot 2k^2$ für ein $k \in \mathbb{Z}$

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.

Beispiel: Satz: Die Zahl $\sqrt{2}$ ist irrational.
Dazu: Definition: Eine Zahl x heißt rational, wenn es Zahlen $p, q \in \mathbb{Z}$ gibt mit $X = \frac{p}{q}$
Hilfssatz (o. Beweis): Wenn a^2 gerade ist, dann ist a gerade.

Beweis: Angenommen $\sqrt{2}$ 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 obigen Hilfssatz auch p. Daraus folgt, dass p^2 durch 4 teilbar ist. Und damit auch $2q^2$. Daraus folgt, dass q^2 gerade ist. Nach Hilfssatz ist damit q gerade. Damit sind p, q nicht teilerfremd, Widerspruch.

Aufgabe Zeigen Sie, dass $(\neg A \rightarrow \perp) \leftrightarrow A$ eine Tautologie ist.

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

Schubfachunterscheidung Wenn $m > n$ Gegenstände auf n Fächer verteilt werden, gibt es mindestens ein Fach, in dem zwei Gegenstände liegen.

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

Beweis: (Fallunterscheidung und Schubfachprinzip)

1. Fall: 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.

2. Fall: Es gibt keine Person, die alle anderen kennt. Dann kennt jeder der n Personen $k \leq n-2$ aus dieser Gruppe

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

Logische Begründung des Beweises durch Fallunterscheidung:

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

1.2 Kombinatorik

Geordnete Mengen Für eine Menge A ist $A^n = \{(a_1, ..., a_n) | a_1, ..., a_n \in A\}$
Es gilt: $|A^n| = |A|^n$

Beispiel: Zahlenschloss mit 4 Stellen und Ziffern 0...9.
Menge der Ziffern $A = \{0, ..., 9\}$
Menge aller Kombinationen $A^4 = \{(0,0,0,0), (0,0,0,1), ..., (9,9,9,9)\}$
Anzahl Kombinationen $|A|^4 = 10^4$

Ungeordnete Mengen Die Anzahl der Permutationen (Anordnungen) von n Elementen bezeichnen wir mit $n!$.
Es gilt: $n! = n \cdot (n-1) \cdot ... \cdot 1$

Die Anzahl aller k-elementigen Teilmengen einer n-elementigen Menge bezeichnen wir mit $\binom{n}{k}$
Es gilt $\binom{n}{k} = \frac{n \cdot (n-1) \cdot ... \cdot (n-k+1)}{k!}$ für $0 \leq k \leq n$

O-Notation Ziel: Angaben über Laufzeit oder Platzverbrauch von Algorithmen oder Datenstrukturen zu machen.

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:

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

Eine Zeitmessung ist nicht geeignet, um die Laufzeit eines Algorithmus anzugeben. Stattdessen zählen wir die Anzahl elementarer Anweisungen (z.B. Zuweisungen, Vergleicher, ...). Diese können jeweils in beschränkter Zeit ausgeführt werden. Sei daher $c > 0$, so dass die Laufzeit dieser elementaren Anweisungen $\leq c$ ist. Die Laufzeit des Algorithmus lineare Suche ist dann eine Funktion $g(n)$, wobei n die Länge des Feldes ist, so dass:

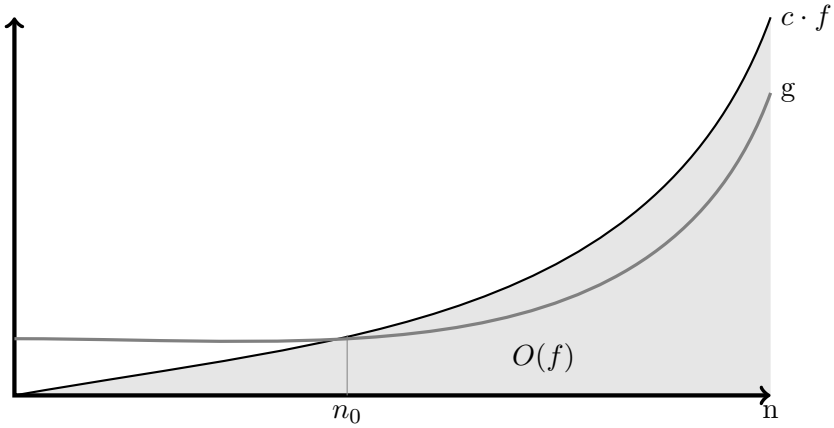
$$g(n) \leq n \cdot c + c$$

Wir ignorieren zunächst den zweiten Summanden und vereinfachen dies zu $g(n) \leq n \cdot c$. Für unterschiedliche Implementierungen und unterschiedlich schnelle Computer ergeben

sich unterschiedliche Konstanten c. Dies legt es nahe, Funktionen die sich nur in einer Konstanten c unterscheiden, in einer Menge zusammenzufassen.

Definition: Für eine Funktion $f \geq 0$ ist $O(f)$ die Menge aller Funktionen g mit $0 \leq g(n) \leq c \cdot f(n)$ für eine Konstante $c > 0$ und alle hinreichend großen $n \in \mathbb{N}$.

$$O(f) = \{g | \text{ es gibt ein } n_0 \in \mathbb{N} \text{ und ein } c > 0, \text{ so dass } \forall n > n_0 \text{ gilt: } 0 \leq g(n) \leq c \cdot f(n)\}$$



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

Beispiel: $17n^2 + 3n + 5\log(n) \in O(n^2)$, da $17n^2 + 3n + 5\log(n) \leq 17n^2 + 3n^2 + 5n^2 = \underbrace{25}_{c} n^2$ für alle $n > 0$

Ziel ist es, stets eine möglichst gute und einfache Abschätzung anzugeben.

Typische Laufzeiten:

- $O(1)$: Konstante Laufzeit
  Beispiel:

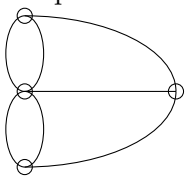
```
1 a=1;
2 b=2;
3 c=3;
```

  LF insgesamt: $3 \cdot c \leq 1 \cdot (3 \cdot c) \in O(1)$

- $O(n)$: Lineare Laufzeit
- $O(2^n)$: Brute-Force-Algorithmen

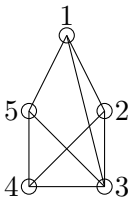
Graphen Königsberger Brückenproblem Gibt es einen Rundweg der über jede Brücke genau einmal führt?

Graph zum Königsberger Brückenproblem



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

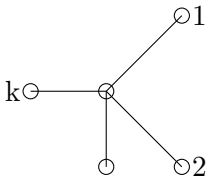
- V die Menge der Knoten ist
- E die Menge der Kanten ist, die aus ungeordneten Paaren $\{U, V\}$ von Knoten besteht.

Beispiel: 

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

Definition Ein Graph heißt vollständig, wenn alle Knoten paarweise verbunden sind.
Folgerung: Ein vollständiger Graph mit n Knoten besitzt $\binom{n}{2}$ Kanten.

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 $\sum_{v \in V} deg(v) = 2|E|$

Beweis: Wenn wir jede Kante in der Mtite durchschneiden, ist jeder Knoten v mit genau $deg(v)$ Hälften verbunden. Also ist $\sum_{v \in V} dev(v)$ gleich der Anzahl der Kantenhälften und diese ist $2|E|$.

Weg und Kreise

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

Ein Weg heißt Kreis, wenn $v_1 = v_r$.
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.

Beispiel
Ein Hamiltonkreis ist ein Kreis, der jeden Koten genau einmal enthält.
Für einen Graphen G ist ein Euler-Kreis ein Kreis, der jede Kante genau einmal enthält.

Satz (Euler)
Ein zusammenhängender Graph besitzt einen Euler-Kreis genau dann, wenn der Grad aller Knoten gerade ist. (Königsberger Brückenproblem nicht lösbar)

Bäume Definition: Ein Baum ist ein zusammenhängender Graph, der keine Kreise enthält. Ein Blatt ist ein Knoten v mit $deg(v) \leq 1$.

Satz: Sei $G = (V, E)$ ein Baum. Dann besitzt G $|V| - 1$ Kanten.
Beweis (Induktion nach $n = |V|$)
$n = 1$: Ein Baum mit einem 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 Induktionsvorraussetzung $n - 1$ 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 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 Induktionsvorraussetzung, dass diese Teilbäume jeweils 2^k Blätter besitzen. Folglich besitzt der Binärbaum der Tiefe $k = 1$, $2 \cdot 2^k = 2^{k+1}$ Blätter.

Algorithmus 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, Laufzeit $O(n)$

Binäre Suche: Vorraussetzung: 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 weitergesucht in der

- linken Hälfte des Array, wenn der gesuchte Wert kleiner als der Wert in der Mitte des Arrays

- rechten Hälfte des Array, wenn der gesuchte Wert größer als der Wert in der Mitte des Arrays

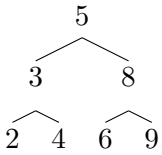
ist.
Beispiel: Suche nach 17 in dem Array

0	1	2	3	4	5	6	7	8	9
2	5	6	7	10	16	22	30	31	50

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 den 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 der Durchläufe ist die Länge des längsten Pfades von der Wurzel zu einem Blatt. Da jedes Blatt einem Teilarray mit einem Element entspricht, gibt es genau n Blätter.

Aus dem oben bewiesenen Satz folgt, dass dieser Binärbaum daher die Tiefe $k = \log_{(2)} n$ 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 den gilt:
Jeder in ein Knoten gespeicherter Wert ist größer als alle Werte im linken Teilbaum und kleiner als alle Werte im rechtem Teilbaum.



Implementierung in C

Zur Verwendung eines binären Suchbaumes (jeder Knoten hat ≤ 2 Blätter) sind folgende Funktionen nötig:

- Suche nach einen Wert
  Dazu wird der Suchbaum, beginnend an der Wurzel, rekursiv durchsucht. Die Laufzeit ist von der Tiefe des Baums abhängig. Bei einem vollständig balancierten Baum liegt diese in $O(\log n)$, bei einem linear entarteten Baum in $O(n)$.

- Werthinzufügen
  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 zulest von der Suche besuchten Knoten (links oder rechts). LZ: Wie Suche.

- Wert entfernen: Kompliziert

Hashing Gegeben: Eine Menge U von potenziellen Schlüsseln (z.B. Artikelnummer), Menge $S \subseteq U$ von zu verwaltenden Schlüsseln.
Zur Verwaltung der Datensätze wird eine Hashfunktion $h : U \mapsto T$ verwendet, die in eine Hashtabelle T abbildet. Wenn wir annehmen, dass h injektiv wäre,
dann lassen sich folgende Operationen implementieren:

- Suche if(T[h(s)] > 0)
- Einfügen T[h(s)]++
- Löschen T[h(s)]=0

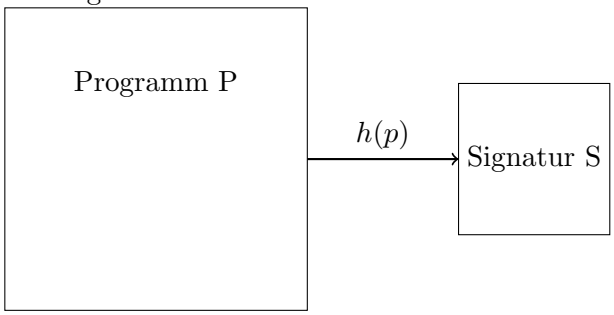
Wegen $|U| >> |S|$ kann h aber nicht injektiv sein (Schubfachprinzip). Folglich sind Kollisionen möglich, d.h. zwei unterschiedliche Schlüssel s, s' können den gleichen Hashwert $(h(s) = h(s'))$ besitzen.

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.

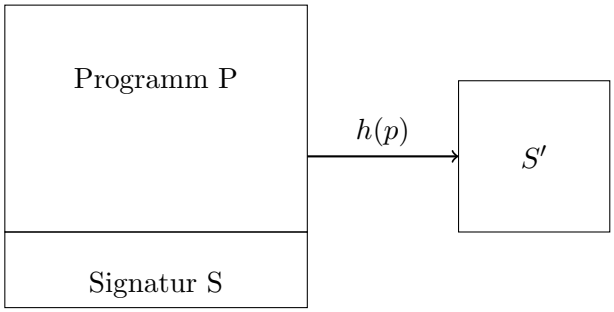
Implementierung in C durch structs und Zeiger
Laufzeit Zugriff auf ein Listenelement $O(n)$.

Um Überlauflisten zu verwenden, verwenden wir eine Tabelle von Listen. An jeder Position der Hashtabelle werden die dort gespeicherten Einträge in einer Liste verwaltet. Wenn die Hashtabelle nicht zu stark befüllt ist (z.B. Anzahl gespeicherte Elemente $n = \frac{1}{2}$), 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. in dem 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 Dokumentes.



Damit lässt sich Prüfen ob das Programm P zu der Signatur S passt.

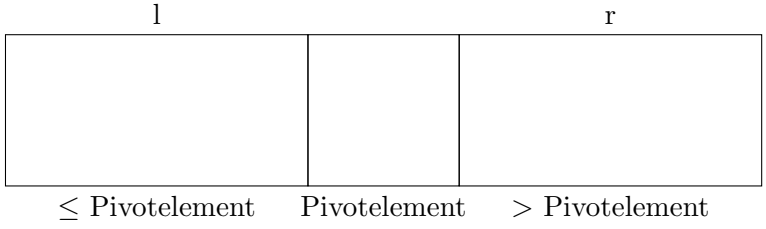


Für $S = S'$ wurde P nicht verändert. Für $S \neq S'$ ist P fehlerhaft (möglich Ursachen: Fehler bei der Datenübermittlung, P wurde manipuliert).
Anwendung: Integritätsprüfung von Programmen in der Linux-Paketverwaltung.
Schutz vor Veränderung von Hardware oder Systemeinstellungen. (Digital Rights Managment, Motorsteuergeräte)
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 das Prüfen des Hashwertes nicht festgestellt werden.

Sortierverfahren Naive Sortierverfahren haben eine Laufzeit in $O(n^2)$. Ein besseres Verfahren ist Quicksort führt rekursiv zwei Schritte aus:

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

2. Die Elemente werden so umgeordnet das die Elemente $\leq$ Pivotelement vorne, die Elemente $>$ Pivotelement hinten und dazwischen das Pivotelement stehen.

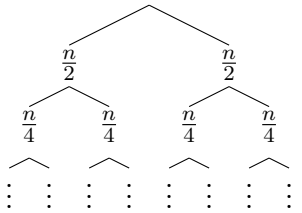


3. Quicksort wird rekursiv für die Listen l,r aufgerufen, bis die Liste leer ist.

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

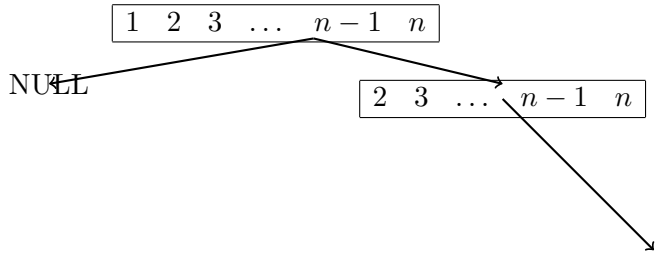
1. Die zu sortierende Liste (bzw. das Array) wird halbiert. Es entstehen Teillisten l,r.
2. Die Teillisten l,r werden rekursiv sortiert, bis sie leer oder die Länge 1 haben.
3. Die sortierten Teillisten werden zu einer sortierten Liste zusammengefügt (Merge-Operation).

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

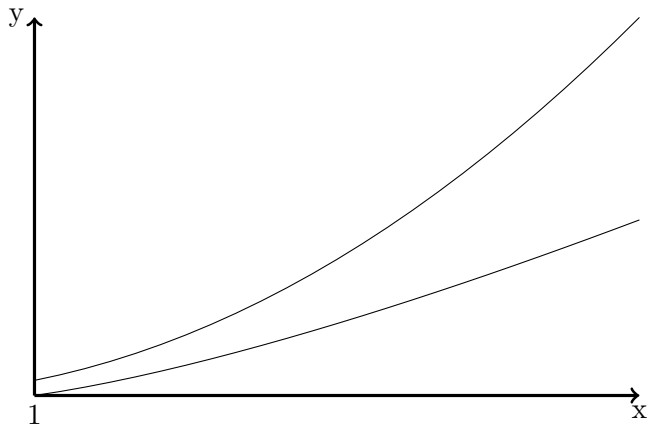


Zum Erzeugen der Hälften und den Zusammenfügen fällt der Aufwand $O(|\text{linke Hälfte}| + |\text{rechte Hälfte}|)$ an.
Auf jeder Ebene des Baumes sind das $O(n)$. Die Tiefe des Baumes ist n. Satz aus der Vorlesung $k = \log n_2$. Die Laufzeit von Mergesort liegt folglich in $O(n \log n)$.
Dies liegt nahe am theoretischen Optimum für Sortierverfahren.

Die mittlere Laufzeit von Quicksort liegt ebenfalls in $O(n \log n)$.
Worstcase Laufzeit von Quicksort:
Wir stellen die rekursiven Aufrufe von Quicksort für eine sortierte Liste als Binbaum dar:



Anzahl der Vergleiche beim Sortieren einer sortierten Liste:
$n - 1 + n - 2 + ... + 1 = \frac{(n-1)n}{2} \in O(n^2)$
Die Laufzeit von Quicksort für eine sortierte Liste liegt daher in $O(n^2)$.



Platzbedarf:
Quicksort kann ohne zusätzlichen Speicherplatz (in-place) implementiert werden.
Mergesort braucht zusätzlichen Speicherplatz für die Merge-Operation.

Graphalgorithmen Datenstrukturen zur Repräsentation von Graphen:

- Adjazenzmatrix: Die ist eine Matrix (a_{uv}) mit $a_{uv} = \begin{cases} 1 & \text{für} \quad \{u,v\} \in E \\ 0 & \text{sonst} \end{cases}$
- Adjasenzliste: Array, das an der Position $u \in V$ eine Liste aller Knoten $v \in V$ enthält mit $\{u,v\} \in E$

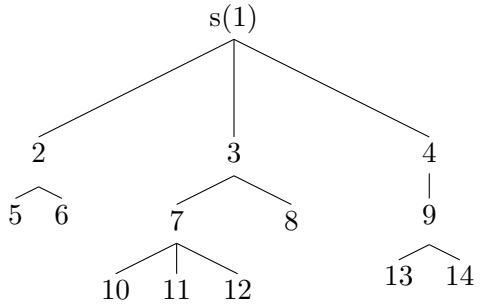
Speicherbedarf:
Adjazenzmatrix $O(|V|^2)$
Adjazenzliste $O(|V| + |E|)$
Für Bäume verbraucht die Adjazenzliste den Platz $O(|V| + |V| - 1) = O(|V|)$ und damit weniger Platz als die Adjazenzmatrix.
Für beliebige Graphen gilt ferner $O(|V| + |E|) \subseteq O(|V|^{\frac{|V| \cdot |V-1|}{2}}) = O(|V|^2)$.

Rechenzeit:	Adjazenzmatrix	Adjazenzliste
Feststellen ob $\{u,v\} \in E$	$O(1)$	$O(\deg u)$
Nachbarn eines Knotens u bestimmen	$O(\|V\|)$	$O(\deg u)$

Folgerung: Adjazenzliste meistens effizienter als Adjazenzmatrix.

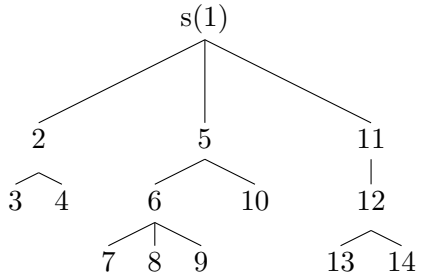
Breitensuche Die Breitensuche durchsucht alle Knoten eines Graphen, beginnend mit einem Startknoten, indem schichtweise die Nachbarn eines Knotens, danach die Nachbarn dieser Knoten usw. besucht werden.

Beispiel: Breitensuche in einem Wurzelbaum
Die Suche beginnt im Knoten s (Zahl gibt Reihenfolge des Besuchs an)



Tiefensuche Die Tiefensuche durchsucht alle Knoten eines Graphen, beginnend mit einem Startknoten, indem Pfade verfolgt werden, bis ein Knoten keine unbesuchten Nachbarn mehr besitzt. Erst wenn das Ende eines Pfades erreicht ist, werden unbesuchte Nachbarn des zuletzt besuchten Knotens besucht (Backtracking).

Beispiel: Tiefensuche in einem Wurzelbaum



Warteschlange: FIFO-Datenstruktur (First in, first out)

Entnehmen ← ☐☐☐☐☐☐☐☐ ← Einfügen

Effiziente Implementierung: Einfach verkette Liste mit zusätzlichen Zeiger auf das letzte Element

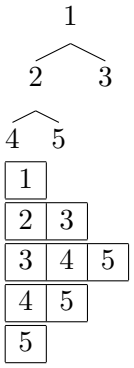
→ ☐ → ☐ → ☐ → ☐ ←

Entnehmen $O(1)$

Algorithmus Breitensuche (für zusammenhängende Graphen)

1. Füge Startknoten einer Warteschlange hinzu
2. Solange die Warteschlange nicht leer ist:
   - Entferne das erste Element v von der Warteschlange
   - Füge noch unbesuchte Nachbarn von v der Warteschlange hinzu

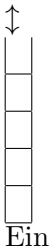
Beispiel:



Laufzeit: Jeder Knoten v wird genau einmal aus der Warteschlange entfernt und für $\deg v$ Nachbarn geprüft, ob diese in die Warteschlange eingefügt werden müssen. Aufwand dazu, wenn eine Adjazenzliste verwendet wird:

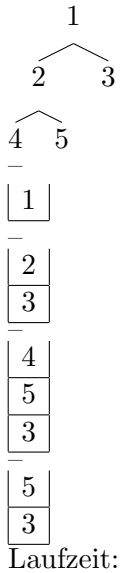
$$O(\sum_{v \in V}(1 + \deg v)) = O(|V| + \sum_{v \in V} \deg v) = O(|V| + 2|E|) \overset{\text{Handshake-L.}}{=} O(|V| + |E|)$$

Tiefensuche: Aus der Implementierung der Breitensuche erhalten wir eine Tiefensuche, wenn anstelle der Warteschlange ein Stack verwendet wird.
Stack: LIFO-Datenstruktur (Last in, first out)



Ein Stack kann durch eine einfach verkettete Liste realisiert werden.
Einfügen, Entnehmen: $O(1)$

Beispiel:



Laufzeit: Gleiche Laufzeit wie die Breitensuche, da nur die Reihenfolge, in der die Knoten besucht werden, unterschiedlich ist.

Gerichtete Graphen und binäre Relationen

Definition Ein gerichteter Graph ist ein Paar $G = (V, E)$ mit $E \subseteq V \times V$.
Beispiel: $G = (V, E)$ mit $V = \{1, 2, 3\}, E = \{(1, 1), (1, 2), (2, 1), (2, 3)\}$
$1_{\circlearrowleft} \leftrightarrow 2 \rightarrow 3$

Definition Sei E eine binäre Relation auf V (d.h. $E \subseteq V \times V$).

- Die reflexive Hülle von E ist die Relation $E \cup \{(v, v) | v \in V\}$
- Die symmetrische Hülle von E ist die Relation $E \cup \{(v, u) | (u, v) \in E\}$
- Die transitive Hülle von E ist die Relation

$$\bigcup_{n \geq 1} E^n$$

  oder gleichwertig

$$E \cup \{(v, w) | \exists v_1, \ldots v_k \in V : (v, v_1) \in E, (v_1, v_2) \in E, \ldots, (v_k, w) \in E\}$$

- Die reflexive und transitive Hülle von E ist die Vereinigung der reflexiven und der transitiven Hülle von E.
- Aus den Labyrinth kann ein Graph $G = (V, E)$ erzeugt werden. Daraus kann man einen gerichteten Graphen $G' = (V, E')$ mit $E' = \{(u, v) | \{u, v\} \in E\} \cup \{(v, u) | \{u, v\} \in E\}$ erzeugen.

Es gibt einen Weg zum Ausgang, wenn (Startposition, Ausgangsknoten) $\in$ reflexiv-transitive Hülle von E'.
Die reflexiv-transitive Hülle einer Relation lässt sich mit einer Breiten- oder Tiefensuche berechnen.

Topologische Sortierung

Problem Wie lassen sich Arbeitsschritte, zwischen denen eine Abhängigkeit besteht, in eine Reihenfolge bringen?

Definition Sei $G = (V, E)$ ein DAG (directed acyclic graph).
Eine topologische Sortierung von G ist eine Nummerierung von G mit Nummer $(n) <$ Nummer (r) für $(u, v) \in E$.

Beispiel: Eine topologische Sortierung kann durch eine Tiefensuche bestimmt werden:

```
1  topsort:
2      for v ∈ V
3          markiere v mit weiss
4      for v ∈ V
5          tiefensuche(v)
6
7  tiefensuche(v):
8      v grau: Fehler ("Graph enthaelt Kreis")
9      v weiss: markiere v mit grau
10         for {u|(v, u) ∈ E}
11             tiefensuche(u)
12         markiere v mit schwarz und fuege v an den Kopf einer Liste
```

Laufzeit: $O(|V| + |E|)$ (wie bei Tiefensuche)

Codierungstheorie

Problem Daten können bei der Übertragung verändert werden. Wie können die Fehler erkannt und gegebenenfalls korrigiert werden?

Häufige Lösung zur Fehlererkennung: Prüfsummen

Paritätsprüfung Für eine Folge von Bits $b_1, \ldots, b_{n-1} \in \{0, 1\}$ wird das Paritätsbit

$$b_n = (\sum_{i=1}^{n-1} b_i) \mod 2$$

berechnet. Das erhaltene Codewort ist $b_1, \ldots, b_n$.

Beispiel:
00000 ist ein Codewort

00001 ist kein Codewort
10100 ist ein Codewort
11100 ist kein Codewort

In der endlichen Gruppe $\mathbb{Z}_2, t)$ gilt $1 + 1 \equiv 0(\mod 2)$. Aus

$$\sum_{i=1}^{n-1} b_i \equiv b_n(\mod 2)$$

folgt daher

$$\sum_{i=1}^{n} b_i \equiv 0(\mod 2)$$

Die Menge der Codewörter lässt sich damit darstellen durch

$$\{(b_1, \ldots, b_n) \in \{0,1\}^n | \sum_{i=1}^{n} b_i \equiv 0(\mod 2)\}$$

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

Beweis: Seien $b_1, \ldots, b_n$ ein Codewort und $b'_1, \ldots, b'_n$ ein Wort, das sich an Stelle k von $b_1, \ldots, b_n$ unterscheidet. Angenommen $b'_1, \ldots, b'_n$ ist ein Codwort. Dann gilt $0 \equiv \sum_{i=1}^{n} b'_i \equiv \sum_{i=1}^{n} b_i + (b'_k - b_k) \equiv 0 + b'_k - b_k \equiv \underbrace{1}_{\text{da } b'_k \neq b_k}$ Widerspruch, q.e.d.

Anwendungen

- Fehlererkennung in Hardware (Speicher, Festplatte, Netzwerke)
- 7-bit ASCII (Bit 8 als Paritätsbit)

Ferner kann das Verfahren zur Rekonstrutktion 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 $1 \leq k \leq n$

$$b_k \equiv \sum_{i \neq k} b_i(\mod 2)$$

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.

ISBN-Code Um festzustellen, ob eine Buchnummer falsch eingetippt wurde, enhält der ISBN-Code eine Prüfsumme.
Beispiel: $\underbrace{382741826}_{\text{9-st. BuchNr}}\ \underbrace{7}_{\text{Prüfziffer}}$
Für die Prüfziffer gilt:
$z_{10} = \sum_{i=1}^{9}(i \cdot z_i) \mod 11$
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

$$\{(z_1, \ldots, z_{10}) | z_1, \ldots, z_9 \in \{0, \ldots, 9\}, z_{10} \in \{0, \ldots, 10\}, \sum_{i=1}^{10} i \cdot z_1 \equiv 0(\mod 11)\}$$

Satz Der ISBN-Code ist 1-fehlererkennend.

Beweis: Seien $z_1, \ldots, z_{10}$ ein Codewort und $z'_1 \ldots z'_{10}$ ein Wort, das sich an Stelle k von $z_1, \ldots, z_{10}$ unterscheidet. Angenommen $z'_1 \ldots z'_{10}$ ist ein Codewort. Dann gilt

$$0 \equiv \sum_{i=1}^{10} i \cdot z'_i \equiv \sum_{i=1}^{10} i \cdot z_1 + k \cdot (z'_k - z_k) \equiv 0 + k(z'_k - z_k)$$

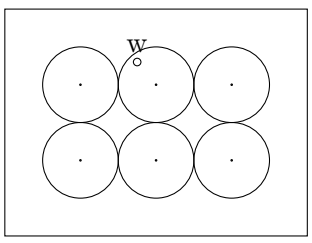
Da 11 eine Primzahl ist, ist $(\mathbb{Z}_{11}, +, \cdot)$ ein Körper. Daher können wir durch k teilen und erhalten

$$0 \equiv z'_k - z_k$$

woraus $z'_k \equiv z_k$ folgt, Widerspruch, q.e.d.

Satz Der ISBN-Code erkennt Vertauschungen von Ziffern (insbesondere Zahlendreher) ohne Beweis.

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).



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

Satz

1. Ein Code ist k-fehlererkennend genau dann wenn sein Minimalabstand mindestens $k+1$ ist.

2. Ein Code ist k-fehlerkorrigierend genau dann wenn sein Minimalabstand mindestens $2k+1$ ist.

Beweis:

1. „⇒“ Sei v ein Codewort und w ein Wort mit $d(v,w) \leq k$. Wenn der Code k-fehlererkennend ist, darf es demnach kein anderes Codewort v' geben, welches einen Abstand $d(v,v') \leq k$ hat, da ansonsten v' selbst kein Codewort mehr wäre, sondern ein Wort mit k Fehlern. Daraus folgt, dass der Minimalabstand $d(v,v')$ des Codes mindestens $k+1$ ist.

   „⇐“ Für eine Codewort v sei $s(v) = \{w|d(v,w) \leq k\}$. Aus $d(v,v') \geq k+1$ (Für alle Codewörter v, v' für die gilt: $v \neq v'$):

   - $\{v\} \cap s(v') = \varnothing$
   - $\{v'\} \cap s(v) = \varnothing$

   (denn anonsten wäre der Abstand $\leq k$)
   Jedes fehlerhafte Wort $w \in s(v)$, dass sich in höchstens k Stellen von v unterscheidet, kann somit erkannt werden.

2. ($\Rightarrow$) Seien v ein Codewort und w ein Wort mit $d(v,w) \leq k$. Wenn der Code k-fehlerkorrigierend ist, darf es nur ein Codewort v mit $d(v,w) \leq k$ geben. Folglich muss $d(v',w) \geq k+1$ für alle Codewörter $v' \neq v$ gelten, woraus $d(v,v') \geq 2k+1$ folgt.
   Daraus folgt, dass der Minimalabstand des Codes mindestens $2k+1$ ist.

   ($\Leftarrow$) Für ein Codewort v sei $s(v) = \{w|d(v,w) \leq k\}$. Aus $d(v,v') \geq 2k+1$ für Codewörter v, v' mit $v \neq v'$ folgt $S(v) \wedge S(v') = \varnothing$ (denn angenommen $S(v) \wedge S(v') \neq \varnothing$, dann hätten v, v' einen Abstand $\leq 2k$). Jedes $w \in S(v)$ lässt sich daher eindeutig zu v decodieren.

Beispiel: Der Minimalabstand des Parity-Check-Doe ist 2, da sich zwei Codewörter in mindestens 2 Stellen unterscheiden. Folglich ist der Code 1-fehlererkennend und 0-fehlerkorrigierend.

Naiver Ansatz zur Fehlerkorrektur: Nachricht mehrfach senden.
Beispiel $0 \to 000, 1 \to 111$
Der Code $\{000, 111\}$ hat Minimalabstand 3 und ist daher 1-fehlerkorrigierend.
Nachteil: Platzverschwendung

Effizienter sind lineare Codes. Diese verwenden Matrix-Vektor-Operationen.
Beispiel: Mit $A = (\underbrace{1 \dots 1}_{n})$ lässt sich der Parity-Check-Code beschreiben durch $\{w \in \{0,1\}^n | Aw^T = 0\}$

Definition Ein Code heißt linear, wenn es eine Matrix A gibt, so dass sich der Code darstellen lässt durch $\{w | Aw^T = 0\}$.
Folgerung:

Satz Ein binärer linearer Code C ist ein Vektorraum über $\mathbb{Z}_2(+, \cdot)$.
Beweis (Skizze):
Abgeschlossenheit der Vektoraddition: Zu zeigen: $w_1, w_2 \in C \Rightarrow w_1 + w_2 \in C$.
Aus $w_1, w_2 \in C$ folgt $Aw_1^T = 0, Aw_2^T = 0$ und daraus folgt $0 = Aw_1^T + Aw_2^T = A(w_1 + w_2)^T$ und daraus folgt $w_1 + w_2 \in C$. Restliche Vektorraumaxiome folgen entsprechend.
Da ein linearer Code ein Vektorraum ist, besitzt er eine Basis.

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

Beispiel: Sei $G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$. Die Zeilen von G sind linearer unabhängig und es gilt $(b_1 b_2 b_3) \cdot \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = (b_1\ b_2\ b_3\ (b_1 + b_2 + b_3))$

Die Zeilen von G bilden daher eine Basis für den Parity-Check-Code der Länge 4 und G ist dessen Generatormatrix

Für den Minimalabstand eines linearen Code gilt:

Satz Sei C ein linearer Code. Der Minimalabstand von C ist $\min\{d(c,0) | c \in C, c \neq 0\}$.

Beweis: Seien $v, w \in C$ mit $v \neq w$. Da C ein Vektorraum ist, gilt $v - w \in C$. Ferner unterscheidet sich $v - w$ von 0 an genau den Stellen, an denen sich v, w unterscheiden. Daraus folgt $d(v,w) = d(v - w, 0)$ und damit $\min\{d(v,w) | v, w \in C, v \neq w\} = \min\{d(c,0) | c \in C, c \neq 0\}$.

Idee zur Konstruktion fehlerkorrigierender Codes: Mehrere Prüfsummen einfügen.

Beispiel $G = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$. Der erzeugte Code ist $C = \{bG | b \in \{0,1\}^3\}$ und $(b_1 b_2 b_3) \cdot G = (\underbrace{b_1\ b_2\ b_3}_{\text{Nachrichtenbits}}\ \underbrace{(b_1 + b_2)\ (b_1 + b_3)\ (b_2 + b_3)}_{\text{Prüfbits}})$.
Der erzeugte Code besteht aus 3 Nachrichtenbits und 3 Prüfbits. Der Minimalabstand ist 3, daher ist der Code 1-fehlerkorrigierend.