Baum

Eingangsvalenzen stets 1 oder 0 (Wurzel)

(Gerichteter) Baum: T = [V,E,r]
ist ein gerichteter Graph [V,E]
mit speziellem Knoten (Wurzel) r $\in$V ,
von dem aus alle anderen Knoten
auf genau einem Weg erreichbar sind.

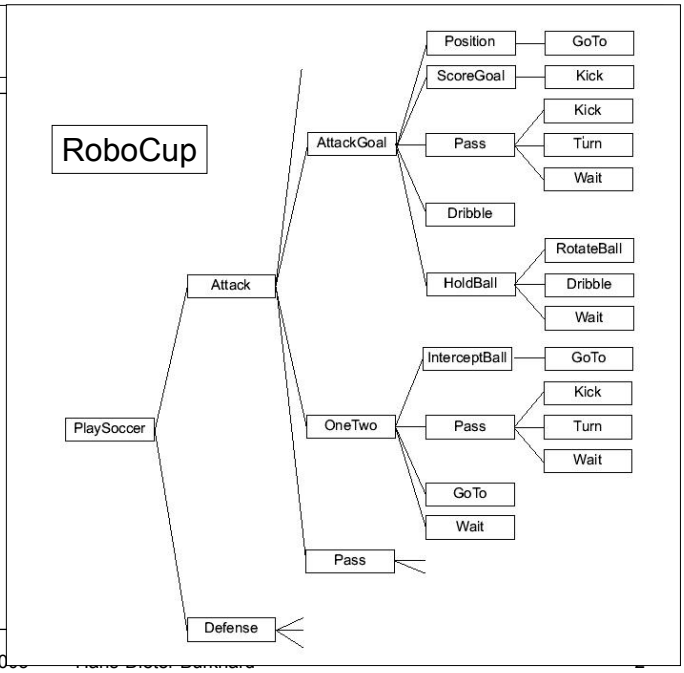
–gerichtet
–zusammenhängend
–ohne Zyklen, ohne Maschen
–Wurzelknoten r eindeutig bestimmt

„Ungerichteter Baum“:
jeder Knoten könnte Wurzelknoten sein

Bäume

Hierarchien:

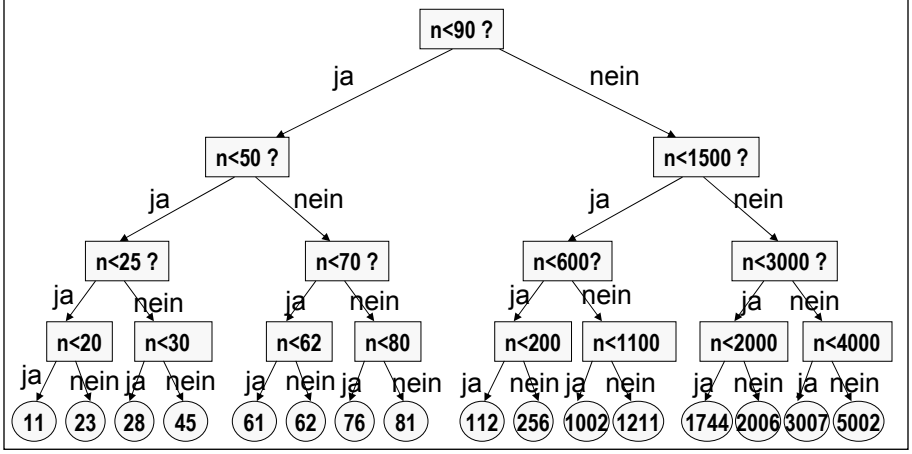
Absichten
und
Unterabsichten



Bäume

Suchbaum

Beschriftungen analog zu Graphen



Baum: Rekursive Definition

Anfang:

Ein Knoten v ist ein Baum: $T=[\{v\}, \emptyset, v]$

Rekursionsschritt:

Wenn v ein Knoten ist

und wenn $T_i = [V_i, E_i, w_i]$ (i=1,...,n) Bäume sind,

deren Knotenmengen paarweise disjunkt sind,

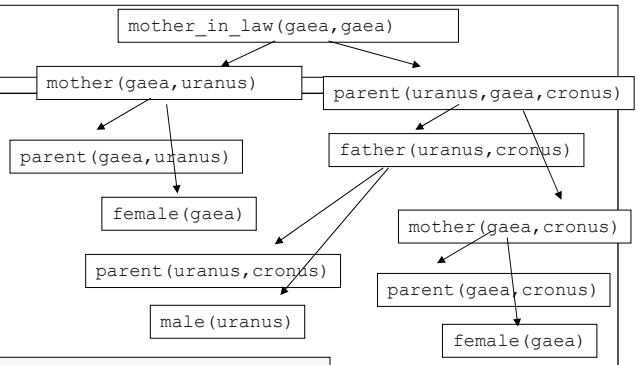
so ist

$T = [V_1 \cup ... \cup V_n \cup \{v\} , E_1 \cup ... \cup E_n \cup \{ [v, w_i] \mid i = 1,...,n \} , v]$

ein Baum.

Bäume

Beweisbaum



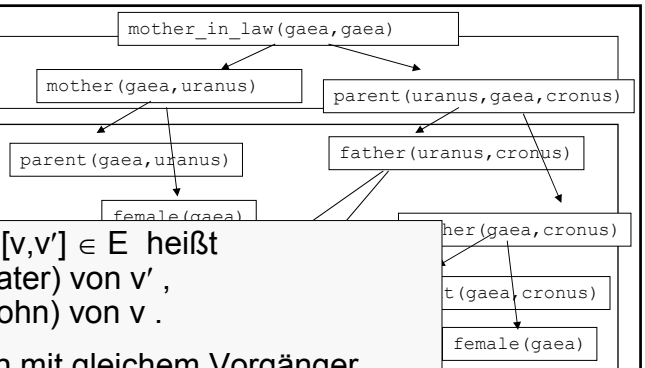
Knoten ohne Nachfolger: *Blätter*.

Knoten mit Nachfolger: *innere Knoten*.

Der Knoten ohne Vorgänger: *Wurzel.*

Tiefe: Entfernung von der Wurzel

Bäume



Für Knoten v, v′ mit [v,v′] ∈ E heißt
v der Vorgänger (Vater) von v′ ,
v′ ein Nachfolger (Sohn) von v .

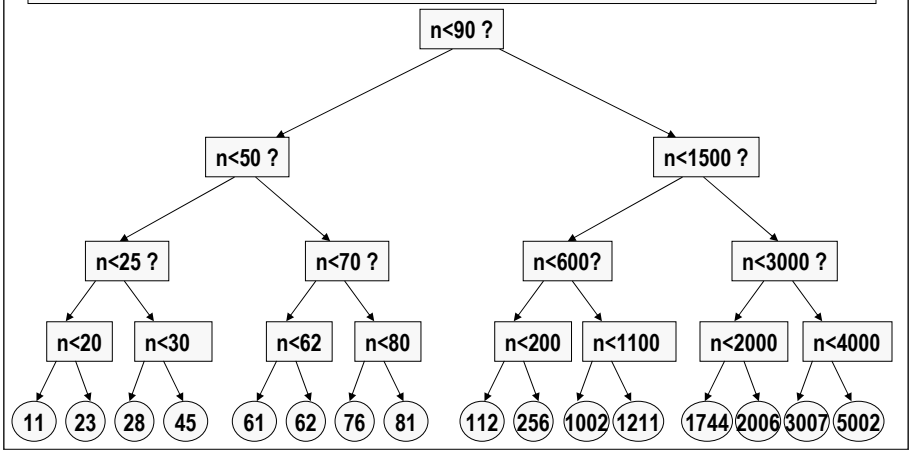
Geschwister: Knoten mit gleichem Vorgänger.

Geordnete Bäume:
Geschwister geordnet (Darstellung: von links nach rechts)

binäre Bäume:
- innere Knoten haben genau 2 Nachfolger
  (weiterhin: geordnet bzgl. linker/rechter Nachfolger
  Unterbäume konsistent mit Ordnung)

„Binärer Suchbaum“

innere Knoten haben genau 2 Nachfolger
(weiterhin: geordnet bzgl. linker/rechter Nachfolger
Unterbäume konsistent mit Ordnung)



Bäume

Bei Tiefe d und Verzweigungszahl b in den Knoten:

- b^d Knoten in jeder Schicht
- Baum der Tiefe d hat dann insgesamt
  $1 + b + b^2 + ... + b^d$ Knoten

letzte Schicht hat mehr Knoten
als der ganze vorherige Baum (falls b >1)

Rekursion für Bäume

Wenn T = [V, E, r] ein Baum ist,
so gilt für jeden Nachfolger v von r :

Der in v beginnende Teilgraph ist ein Baum:

$T|M(v) =_{Def} [\ M(v),\ E \cap M(v) \times M(v),\ v\ ]$ Einschränkung

Rekursive Beweise/Definition in Bäumen

Strukturelle Induktion (vgl. freie Halbgruppen)

Wenn gilt (als Beweis oder Definition):

(A) H gilt für Wurzelknoten r .

(R) Wenn H für Knoten v gilt, so gilt H für alle Nachfolger v‘ von v .

Dann gilt: H gilt für alle Knoten.

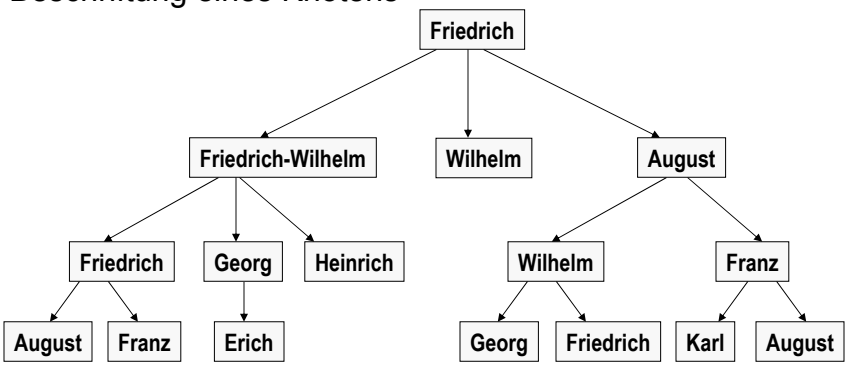
Implementation von Bäumen

- Liste von Paaren [*Knoten, Vorgänger*]
  pre(Sohn,Vater)
- Liste von Paaren [*Knoten, Liste der Nachfolger*]
  succ(Vater,[Sohn-1,...,Sohn-n])
- Rekursiv für binäre Bäume (andere analog) durch Struktur: Knoten,rechter Teilbaum,linker Teilbaum
  (bzw. mit Knotenbeschriftung)
  - tree(Knotenbeschriftung,LinkerBaum,RechterBaum)
  - leerer Baum: nil

Identifikation von Knoten

Unterscheiden

- Identifikator eines Knotens
- Beschriftung eines Knotens

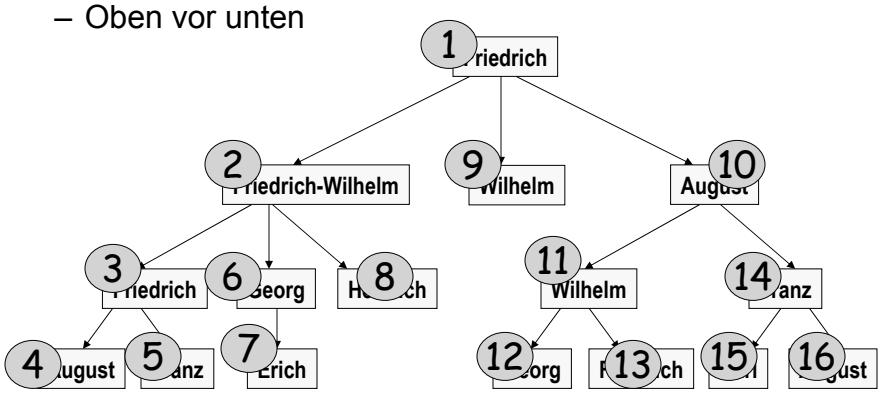


Identifikation von Knoten

Abzählen:

- Links vor rechts
  - Oben vor unten

„Tiefe zuerst"

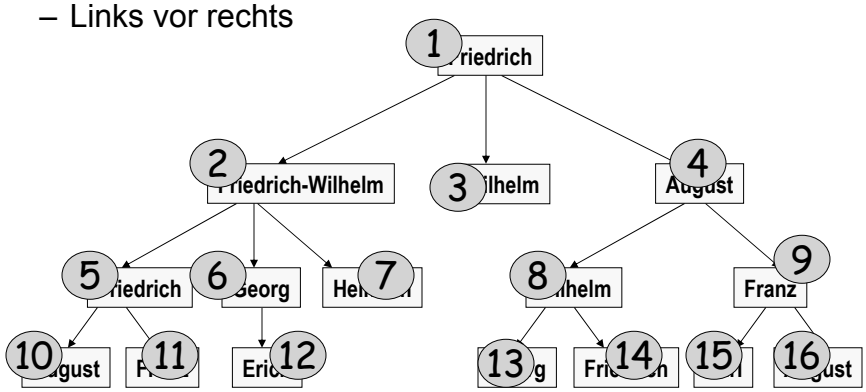


Identifikation von Knoten

Abzählen:

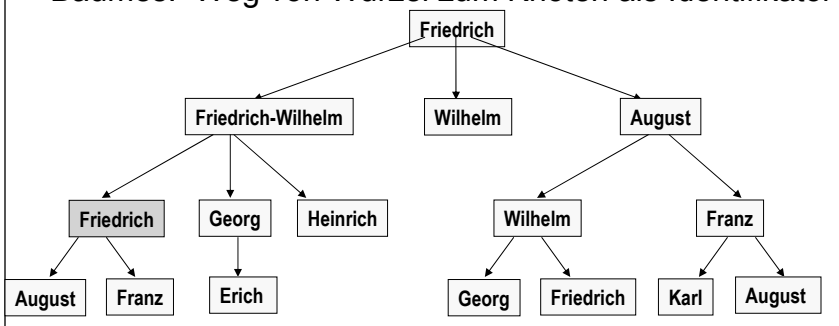
- Oben vor unten
  - Links vor rechts

„Breite zuerst“



Identifikation von Knoten

- Abzählung allein bestimmt nicht Position eines Knotens im Baum
- Eindeutige Identifikation eines Knotens innerhalb eines Baumes: Weg von Wurzel zum Knoten als Identifikator



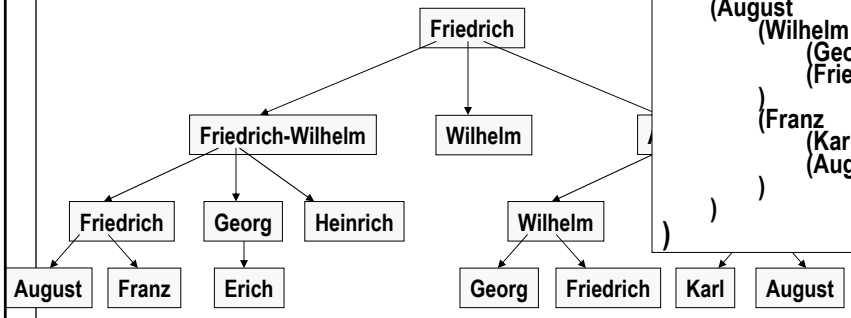
„Friedrich Sohn von Friedrich-Wilhelm dem Sohn von Friedrich“

Sequentialisierung

Rekursive Struktur:

(sohn)

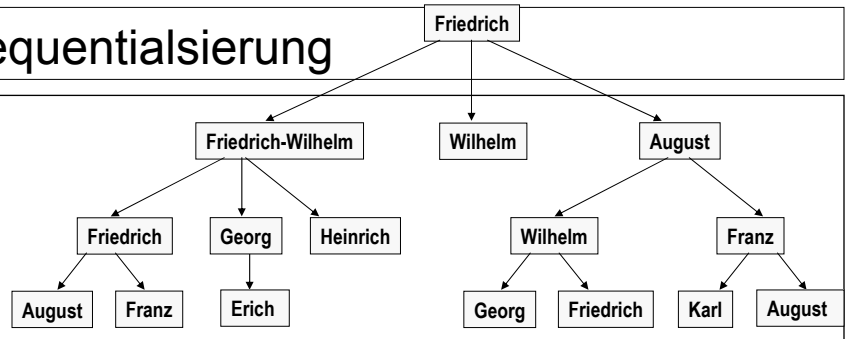
(vater(sohn)(sohn)...(sohn))




```
(Friedrich
      (FriedrichWilhelm
            (Friedrich
                  (August)
                  (Franz)
            )
            (Georg
                  (Erich)
            )
            (Heinrich)
      )
      (Wilhelm)
      (August
            (Wilhelm
                  (Georg)
                  (Friedrich)
            )
            (Franz
                  (Karl)
                  (August)
            )
      )
)
```

(Friedrich(FriedrichWilhelm(Friedrich(August)(Franz))(Georg(Erich))(Heinrich))(Wilhelm)(August(Wilhelm(Georg)(Friedrich))(Franz(Karl)(August))))

Sequentialsierung



(Friedrich(FriedrichWilhelm(Friedrich(August)(Franz))(Georg(Erich))(Heinrich))(Wilhelm)(August(Wilhelm(Georg)(Friedrich))(Franz(Karl)(August))))

Baumstruktur allein durch öffnende (Baumbeginn) und schließende (Baumende) Klammern definiert:

(((( )( ))(( ))( ))( )((( )( ))(( )( ))))

XML= Extensible Markup Language

Beschreibung hierarchischer (baumförmiger) Strukturen

- tags = Klammern
  - <name> für Beginn
  - </name> für Ende
- Weitere Konventionen
  (Inhalte, Attribute)
- Verarbeitungswerkzeuge

- Ursprung:
  electronic publishing
- Keine Formatanweisung
  (html)

http://www.w3schools.com/xml/default.asp

XML= Extensible Markup Language

```
<book>
    <title>My First XML</title>
    <prod id="33-657" media="paper"></prod>
   <chapter>Introduction to XML
        <para>What is HTML</para>
         <para>What is XML</para>
   </chapter>
   <chapter>XML Syntax
        <para>Elements must have a closing tag</para>
        <para>Elements must be properly nested</para>
   </chapter>
</book>
```

- Element: <name> *... Inhalt ...* </name>
- Inhalt: andere Elemente, Text, auch gemischt oder leer

XML= Extensible Markup Language

Daten können als Attribute oder Elemente angegeben werden:

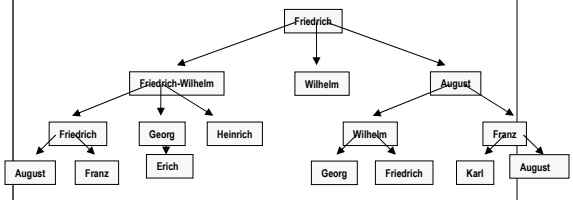
```
<person sex="female">
    <firstname>Anna</firstname>
    <lastname>Smith</lastname>
</person>
```

•Attribut

```
<person>
    <sex>female</sex>
    <firstname>Anna</firstname>
    <lastname>Smith</lastname>
</person>
```

•Element

XML= Extensible Mark



(Friedrich(FriedrichWilhelm(Friedrich(August)(
helm)(August(Wilhelm(Georg)(Friedrich))(Franz

<vater>Friedrich<vater>FriedrichWilhelm<vate
vater>Franz</vater></vater><vater>Georg<vate
nrich</vater></vater><vater>Wilhelm</vater><v
>Georg</vater><vater>Friedrich</vater></vater
vater>August</vater></vater></vater></vater>

```
<vater>Friedrich
    <vater>FriedrichWilhelm
        <vater>Friedrich
            <vater>August</vater>
            <vater>Franz</vater>
        </vater>
        <vater>Georg
            <vater>Erich</vater>
        </vater>
        <vater>Heinrich</vater>
    </vater>
    <vater>Wilhelm</vater>
    <vater>August
        <vater>Wilhelm
            <vater>Georg</vater>
            <vater>Friedrich</vater
            >
        </vater>
        <vater>Franz
            <vater>Karl</vater>
            <vater>August</vater>
        </vater>
    </vater>
</vater>
```

Suche (Retrieval) in einem Baum

Doppelt rekursiv für binären Baum

```
membertree(X,tree(X,_,_)).
membertree(X,tree(_,T1,_)):- membertree(X,T1) .
membertree(X,tree(_,_,T2)) :- membertree(X,T2) .
```

Geordneter binärer Baum

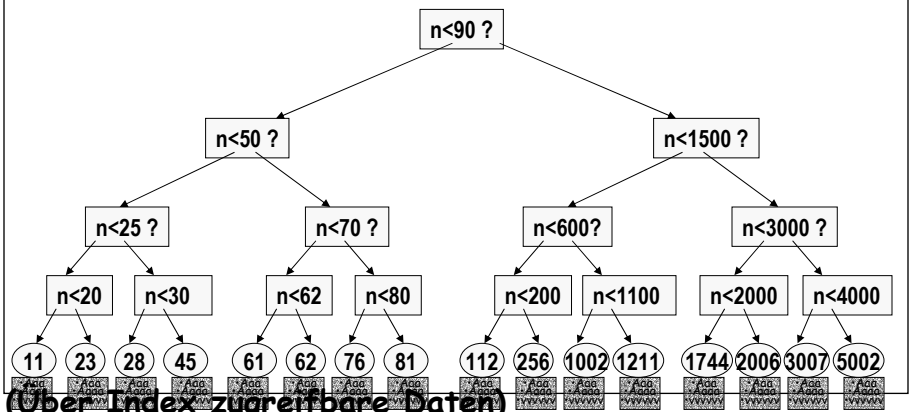
```
node retrieve(int x, node v)
 { if   (v == NIL) return NIL ;
   else if (x == v.value) return v ;
          else if (x < v.value)  return retrieve(x, v.left);
                 else                return retrieve(x, v.right);
 }
```


Suchbaum für Retrieval

Alternative zu Hash-Funktionen.

„Schlüssel“ als Suchbegriff (Knoten-Beschriftung).

Inhalt über Knoten zugreifbar („Anhang“, Verweis).



(Über Index zugreifbare Daten)

Komplexität des Retrievals

Abschätzung: O(log(n))

Tatsächlicher Aufwand abhängig von
- Durchschnittliche Tiefe der Zweige
- Häufigkeit der Suche nach Schlüsseln

Optimierung:
- Baum balancieren bzgl. Tiefe (AVL-Bäume)
  AVL= Adelson-Velskij, Landis, 1962

Wege in einem Graphen

Sei G=[V,E] ein Graph, $v_0 \in V$.

$L(v_0) =_{Def}$ Menge der in v_0 beginnenden Wege $p= v_0 v_1 v_2 ... v_n$

Für endliche gerichtete Graphen G gilt:

- $L(v_0)$ ist eine reguläre Sprache über dem Alphabet V.
- $L(v_0)$ ist endlich gdw. $G|M(v_0)$ azyklisch ist.

$G|M(v_0)$ = von v_0 aus erreichbarer Teilgraph

$G|M(v_0) =_{Def} [M(v_0), E \cap M(v_0) \times M(v_0)]$

$M(v_0) =_{Def}$ Menge der von v_0 erreichbaren Knoten

Erreichbarkeitsbaum

Sei G=[V,E] ein Graph, $v_0 \in V$.

Erreichbare Zustände: $M(v_0) = \{ v \mid v \text{ erreichbar von } v_0 \}$

$L(v_0)$ = Menge der in v_0 beginnenden Wege $p= v_0 v_1 v_2 ... v_n$

„Abwickeln“ des Graphen in v_0 ergibt Erreichbarkeitsbaum:

Aufspalten von Maschen/Zyklen

$T(v_0) = [K, B, k_{v0} , V, \alpha, E, \beta]$ mit

$K = \{ k_p \mid p \in L(v_0) \}$

$\alpha(k_p)$ = der mit p erreichte Knoten

$B = \{ [k_p , k_{pv}] \mid p \in L(v_0) \wedge v \in V \wedge pv \in L(v_0) \}$

$\beta([k_p , k_{pv}])$ = letzte Kante auf Weg pv

$\alpha(k_{v0}) = v_0$

$\alpha(k_p) = v$ für $p= v_0 ... v_n$, $v_n = v$

$\beta([k_p , k_{pv}]) = [v_n,v]$ für $p= v_0 ... v_n$

Erreichbarkeitsbaum

- Für jeden Weg in G=[V,E] eigener Zweig in $T(v_0)$
- Für Knoten k in $T(v_0)$:
  - Name k_p :
    - gemäß Weg p in G
    - Kantenbeschriftungen auf Weg zu k_p in $T(v_0)$
  - Beschriftung $\alpha(k_p)$: bei Weg p in G erreichter Knoten
- $T(v_0)$ endlich gdw. $L(v_0)$ endlich
  (für gerichtete Graphen: gdw. $G|M(v_0)$ azyklisch ist)

Binäre Relationen als Graph darstellen

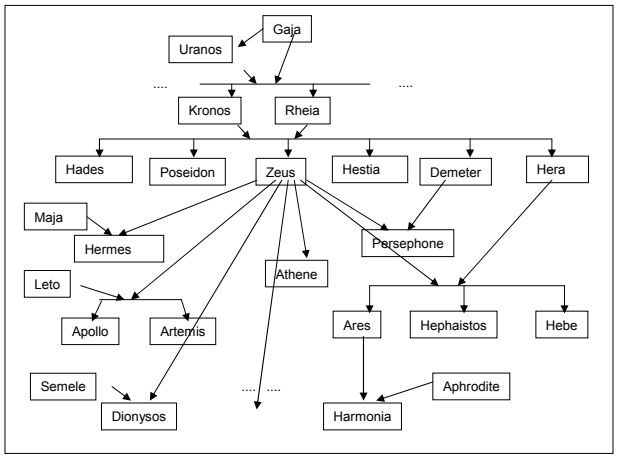
Binäre Relation R über Menge M: $R \subseteq M \times M$

- über natürlichen Zahlen:
  $<, \leq, >, \geq, =, ..., \equiv mod(n)$, ...
- über Mengen:
  $\subset, \subseteq$,..., *gleichmächtig (d.h. card(M)=card(N))*, ...
- über Wörtern einer Sprache:
  suffix, *präfix*, ..., *gleiche Länge, ..., lexikographische Ordnung, …*

Binäre Relationen als Graph darstellen

Binäre Relation R über Menge M: $R \subseteq M \times M$

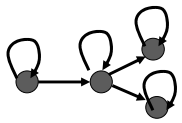
Graph G=[M,R]



Eigenschaften binärer Relationen

Reflexivität **(R)** $\forall a$: aRa

Schlingen $[v,v] \in E$ für alle $v \in V$

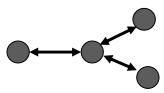


Irreflexivität **(Ir)** $\forall a$: $\neg aRa$

Eigenschaften binärer Relationen

Symmetrie **(S)** $\forall a,b$: $aRb \rightarrow bRa$

Kanten jeweils in beiden Richtungen bzw. ungerichtet

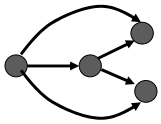


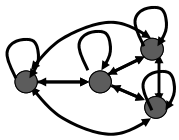
Asymmetrie **(aS)** $\forall a,b$: $aRb \rightarrow \neg bRa$

Antisymmetrie ("identitiv") **(anS)** $\forall a,b$: $aRb \wedge bRa \rightarrow a=b$

Eigenschaften binärer Relationen

Transitivität **(T)** $\forall a,b,c : aRb \wedge bRc \rightarrow aRc$





Direkte Verbindungen zwischen jeweils
allen Knoten auf gerichteten Wegen
(werden bei Darstellung gelegentlich weggelassen)

Eigenschaften binärer Relationen

Konnexität **(K)** $\forall a,b: aRb \vee bRa$

Kanten (in wenigstens einer Richtung)
zwischen allen Knoten
(speziell: Schlingen an allen Knoten)

Linearität **(L)** $\forall a,b: aRb \vee bRa \vee a=b$

Kanten (in wenigstens einer Richtung)
zwischen allen unterschiedlichen Knoten

Anordnungen

- irreflexive Halbordnung: **(Ir)**, **(T)**, (aS) — z.B. $\subset$
- irreflexive Ordnung: **(Ir)**, **(T)**, **(L)**, (aS) — z.B. $<$
- reflexive Quasiordnung: **(R)**, **(T)**
- reflexive Halbordnung: **(R)**, **(T)**, **(anS)** — z.B. $\subseteq$
- reflexive Ordnung: (R), **(T)**, **(anS)**, **(K)** — z.B. $\leq$

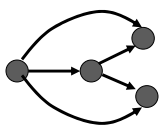
Anordnungen

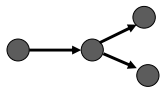
Hasse-Diagramm

Einschränkung des Graphen einer Anordnungsrelation:

Kanten nur von m nach n falls

$mRn \wedge m \neq n \wedge \neg\exists x\,(x \in M \wedge x \neq m \wedge x \neq n \wedge mRx \wedge xRn)$





Anordnungen

Kette: geordnete Teilmenge von M

Halbordnungen: Knoten liegen auf Linien („Ketten“)

„Partielle Ordnung“

Transitivität:
Alle Knoten auf einer Kette direkt verbunden

Ordnungen: alle Knoten liegen auf <u>einer</u> Linie („Kette“)

„Totale Ordnung“

Maximale (bzw. minimale) Elemente in $N \subseteq M$
für Halbordnung R über M :

$$m \in N \wedge \forall x \in N: \; mRx \rightarrow m=x$$

Anfangs-/Endpunkte der Ketten im Teil-Graphen für N

Äquivalenzrelationen

- Äquivalenzrelation **(R)**, **(S)**, **(T)**

Graph zerfällt in stark zusammenhängende Teilgraphen, in denen jeder Knoten mit jedem verbunden ist.

Eine Äquivalenzrelation *R* über einer Menge *M* definiert eine **Zerlegung** von *M* in disjunkte **Klassen**.

Für $a \in M$: $K(a) =_{Def} \{ \; b \mid aRb \; \}$, dabei gilt:

- $K(a) = K(b) \leftrightarrow aRb$
- $K(a) \cap K(b) = \emptyset \leftrightarrow \neg \, aRb$
- $M = \cup \{ \; K(a) \mid a \in M \; \}$

Klassen durch beliebige Repräsentanten eindeutig bestimmt.

Ähnlichkeitsrelation (Verträglichkeitsrelation

- Ähnlichkeitsrelation **(R)**, **(S)**

Graph überdeckt von stark zusammenhängenden Teilgraphen, in denen jeder Knoten mit jedem verbunden ist.

Eine Ähnlichkeitsrelation *R* über einer Menge *M* definiert eine **Überdeckung** $\mathcal{N}$ von *M* durch Mengen $A \subseteq M$ von untereinander ähnlichen Elementen:

$\mathcal{N} =_{Def} \{ A \mid A \subseteq M \wedge A \text{ maximal} \wedge \forall a,b \in A \rightarrow aRb \}$

Im Gegensatz zu Äquivalenzrelationen:

Die Mengen $B(a) =_{Def} \{ b \mid aRb \} \subseteq M$ sind nicht durch Repräsentanten bestimmt. Es kann insbesondere gelten:

aRb und gleichzeitig $B(a) \neq B(b)$

$\neg\, aRb$ und gleichzeitig $B(a) \cap B(b) \neq \emptyset$

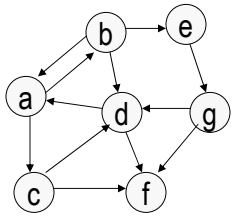
Implementation von Graphen

Und weitere.

ggf. zusätzlich Beschriftungen

Datenstruktur (Inzidenzmatrix, Adjazenzmatrix) als (meist dünn besetzte) Matrix:

- Felder
- Listen

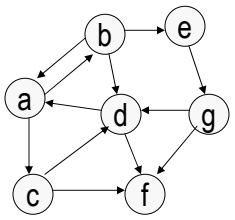


	a	b	c	d	e	f	g
a		1	1				
b	1			1	1		
c				1		1	
d	1					1	
e							1
f							
g				1		1	

Implementation von Graphen

- Datenbank von benachbarten Knoten

kante(knotenname1,knotename2)




```
kante(a,b).
kante(a,c).
kante(b,a).
kante(b,d).
kante(b,e).
kante(c,d).
kante(c,f).
kante(d,a).
kante(d,f).
kante(e,g).
kante(g,d).
kante(g,f).
```

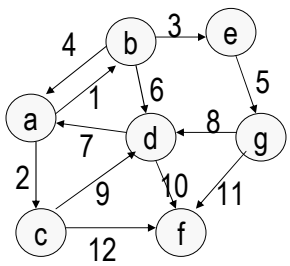
oder kante(kantenname, knotenname1,knotename2)

Implementation von Graphen

- Datenbank von Knoten-/Kantenbeziehungen

eingangskante(knotenname,kantenname)

ausgangskante(knotenname,kantename)

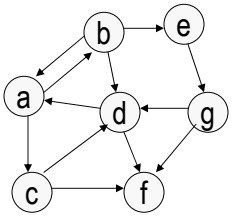



```
eingangskante(b,1).
eingangskante(c,2).
eingangskante(a,4).
eingangskante(d,6).
eingangskante(e,3).
eingangskante(d,9).
eingangskante(f,12).
eingangskante(a,7).
eingangskante(f,10).
eingangskante(g,5).
eingangskante(d,8).
eingangskante(f,11).
```

```
ausgangskante(a,1).
ausgangskante(a,2).
ausgangskante(b,4).
ausgangskante(b,6).
ausgangskante(b,3).
ausgangskante(c,9).
ausgangskante(c,12).
ausgangskante(d,7).
ausgangskante(d,10).
ausgangskante(e,5).
ausgangskante(g,8).
ausgangskante(g,11).
```

Graph als Liste von Adjazenzlisten

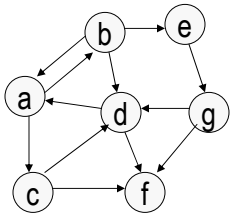
[a:[b,c], b:[a,d,e],c:[d,f],d:[a,f],e:[g],f:[],g:[d,f]]



- Berechnung von Kanten:

```
kante(X,Y,Graph)
        :- member(X:Nachbarn,Graph),member(Y,Nachbarn).
```


Datenbank der Adjazenzlisten



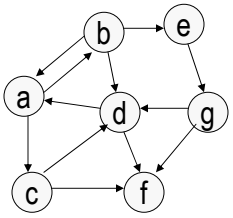

```
adjazenz(a,[b,c]).
adjazenz(b,[a,d,e]).
adjazenz(c,[d,f]).
adjazenz(d,[a,f]).
adjazenz(e,[g]).
adjazenz(f,[]).
adjazenz(g,[d,f]).
```

- Berechnung von Kanten:

```
kante(X,Y) :- adjazenz(X,Nachbarn),member(Y,Nachbarn).
```

Weg in einem Graphen

```
weg(Start,Start,_,[Start]).
weg(Start,Ziel,Graph,[Start|Weg])
  :- kante(Start,Nachbar,Graph),
     weg(Nachbar,Ziel,Graph,Weg).
```



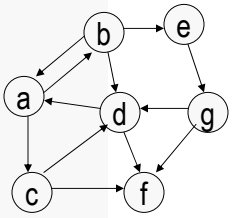

```
?- assert(graph([ a:[b,c], b:[a,d,e],c:[d,f],d:[a,f],e:[g],f:[],g:[d,f] ])).
?- graph(G),weg(a,b,G,W).
G = [a:[b, c], b:[a, d, e], c:[d, f], d:[a, f], e:[g], f:[], g:[d|...]]
W = [a, b] ;
...
W = [a, b, a, b] ;
...
W = [a, b, a, b, a, b] ;
...
W = [a, b, a, b, a, b, a, b]
```


Weg in einem Graphen

```
wegOhneZyklen(Start,Ziel,Graph,Weg)
      :-wegSuchen(Start,Ziel,Graph,Weg,[Start]).

wegSuchen(Start,Start,_,[Start],_).

wegSuchen(Start,Ziel,Graph,[Start|Weg],Bisherige)
  :- kante(Start,Nachbar,Graph),
     not(member(Nachbar,Bisherige)),
     wegSuchen(Nachbar,Ziel,Graph,Weg,[Nachbar|Bisherige]).
```




```
?- graph(G),wegOhneZyklen(a,f,G,W).
G = [a:[b, c], b:[a, d, e], c:[d, f], d:[a, f], e:[g], f:[], g:[d|...]]
W = [a, b, d, f] ;
...
W = [a, b, e, g, d, f] ;
...
W = [a, b, e, g, f] ;
...
W = [a, c, d, f] ;
...
W = [a, c, f] ;
```