



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 15: Topologisches Sortieren

Teil 1: Problemstellung und mathematische Basis

Teil 2: Algorithmus und Implementation



Teil1: Problemstellung und mathematische Basis

Teil2: Software implementierung



Aus einer gegebenen partiellen Ordnung
eine kompatible totale Ordnung produzieren

Un dîner en famille.

by Caran d'Ache



*– Surtout ! ne parlons pas
de l'affaire Dreyfus!*

... Ils en ont parlé ...



Die Problemstellung

Partielle Ordnung: Ordnungsbedingungen zwischen Elementen einer Menge, z.B.

- “Das Abwaschen kommt *vor* der politischen Diskussion”
- “Die Wanderung auf den Üetliberg kommt *vor* dem Essen”
- “Das Medikament muss *vor* dem Essen eingenommen werden”
- “Das Essen kommt *vor* dem Abwaschen”

Totale Ordnung: Eine Sequenz, die alle Elemente der Menge beinhaltet

Kompatibel: Die Sequenz berücksichtigt alle Ordnungsbedingungen

- Üetliberg, Medikament, Essen, Abwaschen, Politik : OK
- Medikament, Üetliberg, Essen, Abwaschen, Politik : OK
- *Politik, Medikament, Essen, Abwaschen, Üetliberg* : not OK

Aus einer gegebenen partiellen Ordnung eine kompatible totale Ordnung produzieren

Warum dieses Beispiel wichtig ist

- Häufiges Problem, präsent in vielen verschiedenen Gebieten
- Interessanter, nicht trivialer (aber auch nicht zu komplizierter) Algorithmus
- Erläutert die Techniken von Algorithmen, Datenstrukturen, Komplexität und anderen Themen der letzten Lektion
- Erklärt Techniken des Software-Engineerings
- Beschreibt wichtige mathematische Konzepte: binäre Relationen und im Speziellen Ordnungsrelationen
- Es ist einfach schön!

Heute: Problemstellung und mathematische Basis

Nächstes Mal: Algorithmus und Konzept im Detail

Topologisches Sortieren: Anwendungsbeispiele

- „Erzeuge aus einem Wörterbuch eine Liste von Definitionen („Glossar“), so dass kein Wort vor seiner Definition steht“
- Erstellung eines kompletten Zeitablaufs für die Ausführung von Aufgaben mit Ordnungsbedingungen
(Häufige Anwendung: „Scheduling“ von Unterhaltungsarbeiten in der Industrie, oft mit tausenden von Einschränkungen)
- Eine neue Version einer Klasse mit neuer Reihenfolge der Features generieren, so dass kein Feature ein anderes, vor ihm deklariertes aufruft

Das Glossar (Wörterbuch)-Beispiel

Ein **Lehrer** ist eine Person, die *Studenten* lehrt

Ein **Student** ist eine Person, die in einer *Universität* studiert

Eine **Universität** ist eine Lehr- und Forschungsinstitution

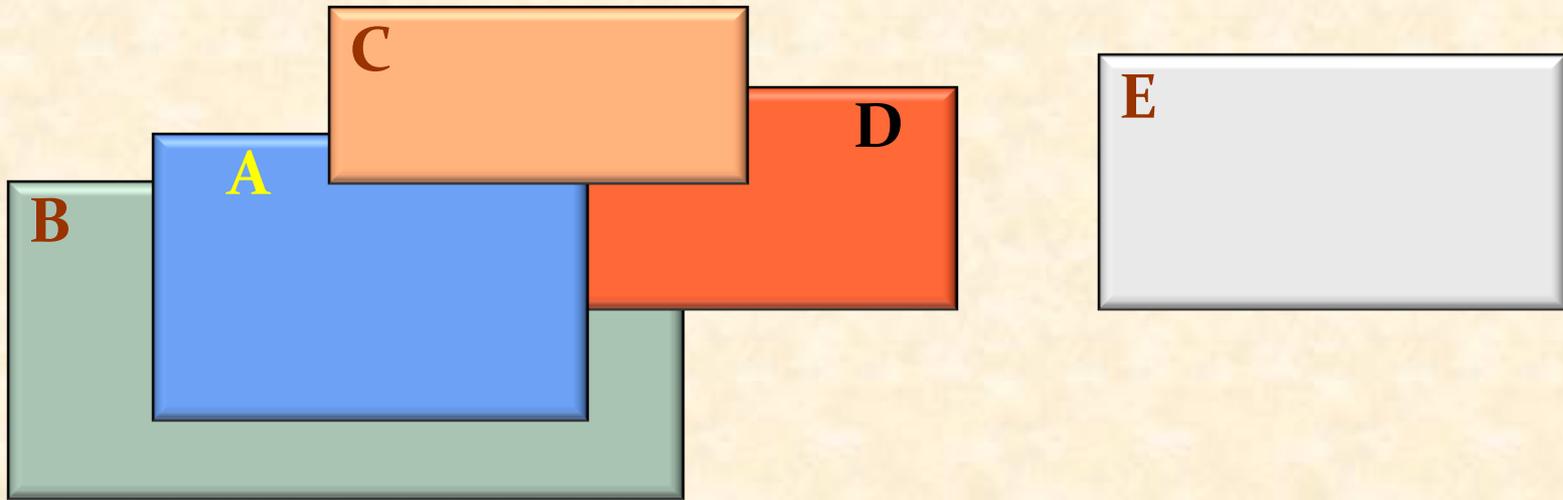
Topologisches Sortieren: Anwendungsbeispiele

- „Erzeuge aus einem Wörterbuch eine Liste von Definitionen („Glossar“), so dass kein Wort vor seiner Definition steht“
- Erstellung eines kompletten Zeitablaufs für die Ausführung von Aufgaben mit Ordnungsbedingungen
(Häufige Anwendung: „Scheduling“ von Unterhaltungsarbeiten in der Industrie, oft mit tausenden von Einschränkungen)
- Eine neue Version einer Klasse mit neuer Reihenfolge der Features generieren, so dass kein Feature ein anderes, vor ihm deklariertes aufruft

Rechtecke mit Überlappungsaufgaben



Bedingungen: [B, A], [D, A], [A, C], [B, D], [D, C]



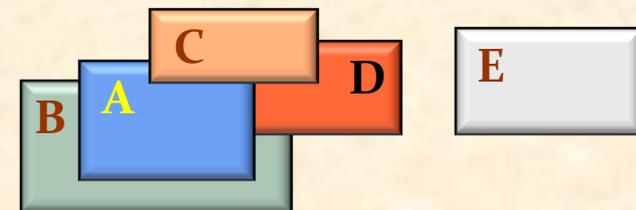
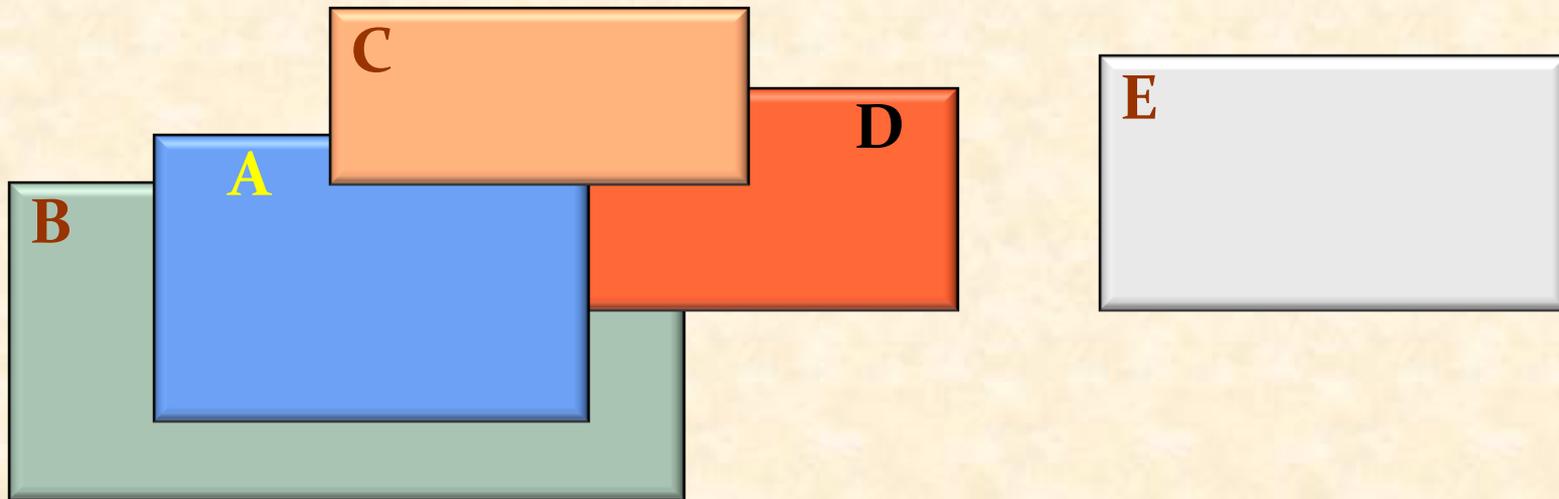
Rechtecke mit Überlappungsaufgaben darstellen



Bedingungen: [B, A], [D, A], [A, C], [B, D], [D, C]

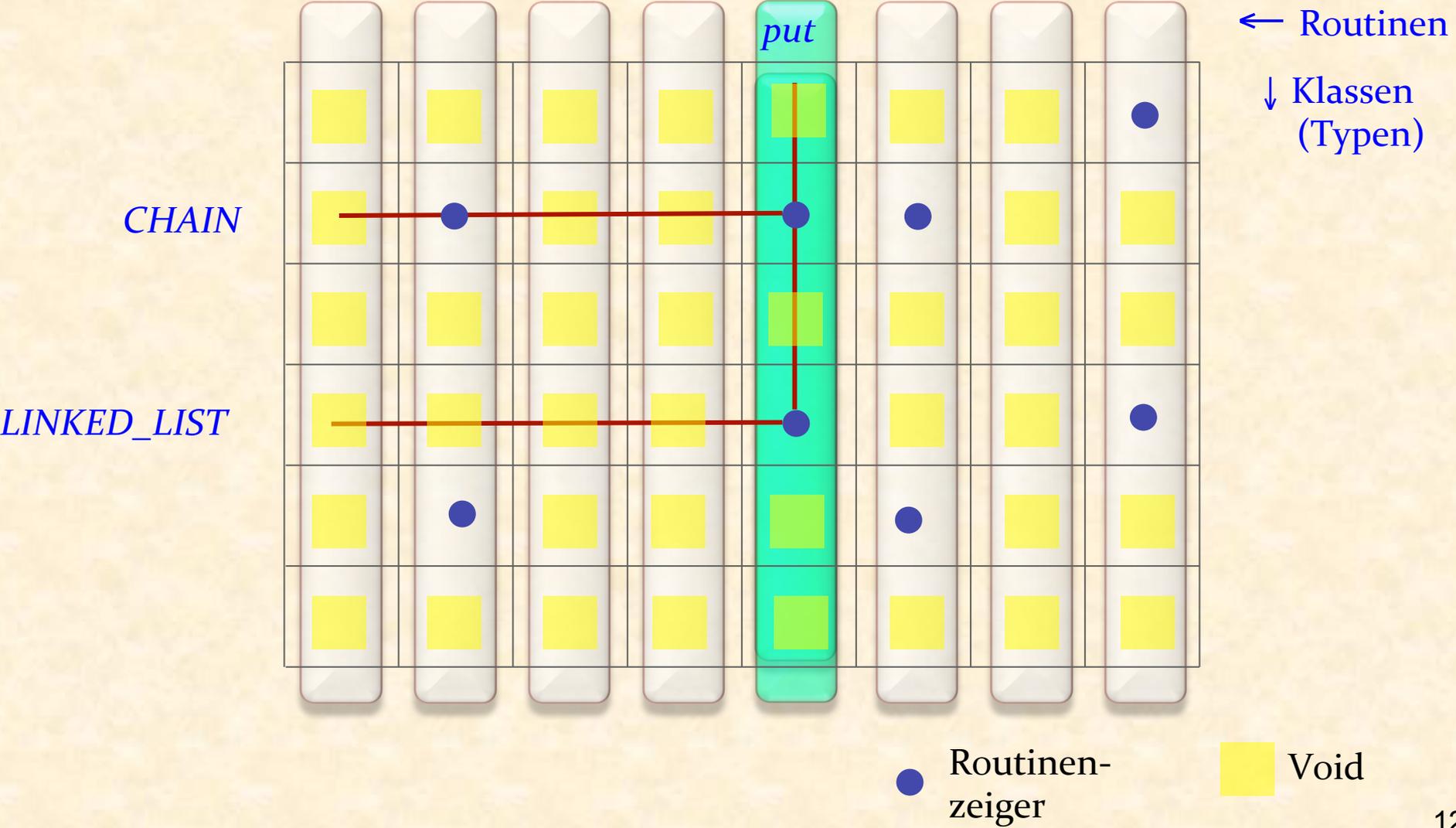
Mögliche Reihenfolge der Anzeige:

B D E A C



Ein Beispiel in EiffelStudio

Um $x.f$ mit dynamischem Binden zu implementieren, brauchen wir eine Tabelle der Routinen



Eine Übung!



Finden Sie im EiffelStudio-Quellcode die Namen der Datenstrukturen, die die Tabelle oder Tabellen der vorhergehenden Folie darstellen



Aus einer gegebenen partiellen Ordnung eine kompatible totale Ordnung produzieren

Partielle Ordnung: Ordnungsbedingung zwischen Elementen einer Menge, z.B.

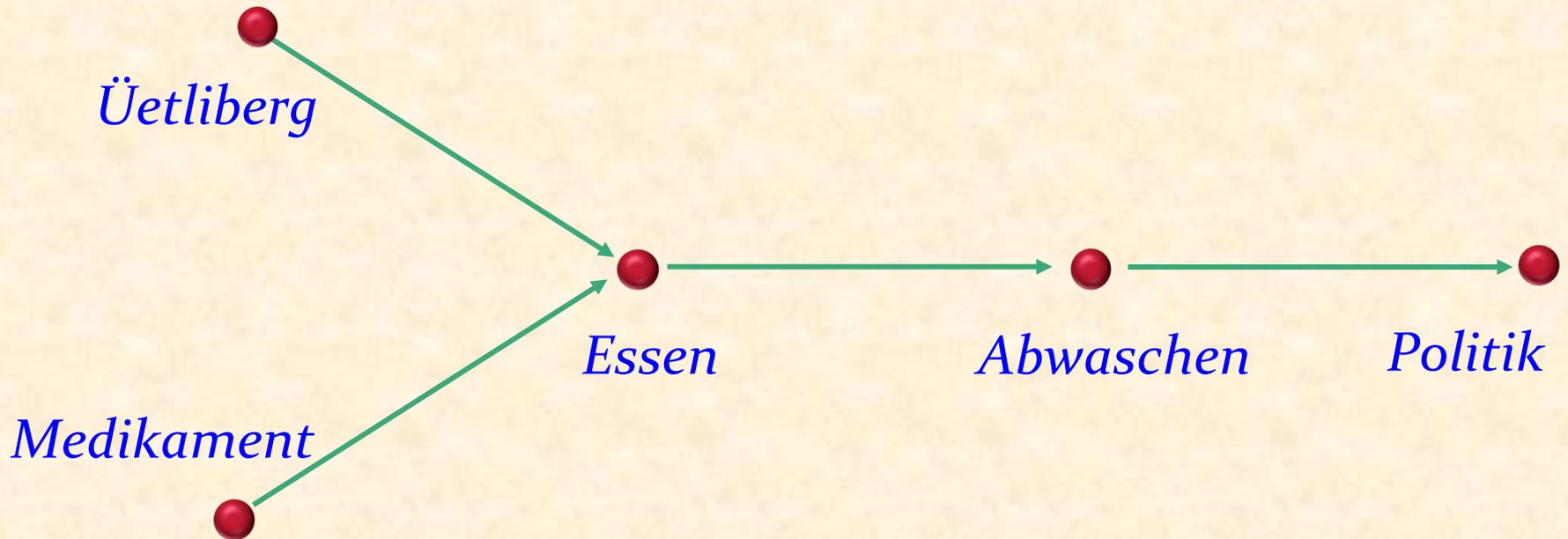
- “Das Abwaschen kommt *vor* der politischen Diskussion”
- “Die Wanderung auf den Üetliberg kommt *vor* dem Essen”
- “Das Medikament muss *vor* dem Essen eingenommen werden”
- “Das Essen kommt *vor* dem Abwaschen”

Totale Ordnung: Eine Sequenz, die alle Elemente der Menge beinhaltet

Kompatibel: Die Sequenz berücksichtigt alle Ordnungsbedingungen

- Üetliberg, Medikament, Essen, Abwaschen, Politik : OK
- Medikament, Üetliberg, Essen, Abwaschen, Politik : OK
- Politik, Medikament, Essen, Abwaschen, Üetliberg : **not OK**

Als Graph dargestellt



- “Das Abwaschen kommt *vor* der politischen Diskussion”
- “Die Wanderung auf den Üetliberg kommt *vor* dem Essen”
- “Das Medikament muss *vor* dem Essen eingenommen werden”
- “Das Essen kommt *vor* dem Abwaschen”



- *“Die Einführung der Rekursion erfordert, dass die Studenten Stapel kennen“*
- *“Abstrakte Datentypen müssen vor Stapel behandelt werden“*
- *“Abstrakte Datentypen beruhen auf Rekursion“*

Die Bedingungen erzeugen einen **Zyklus** (Kreis)

Manchmal gibt es keine Lösung

Ein **Lehrer** ist eine Person, die *Studenten* lehrt

Ein **Student** ist eine Person, die in einer *Universität* studiert

~~Eine **Universität** ist eine Lehr- und Forschungsinstitution~~

Eine **Universität** ist eine Institution, in der *Lehrer*
Studenten lehren

Allgemeine Struktur (1)

Gegeben:

Ein Typ G

Eine Menge von
Elementen vom Typ G

Eine Menge von
Bedingungen zwischen
diesen Elementen

Benötigt:

Eine Aufzählung der
Elemente in einer zu
den Bedingungen
kompatiblen Ordnung

```
class ANORDENBAR [G] feature
  elemente : LIST [G ]

  bedingungen : LIST [TUPLE [G, G ]]

  topsort : LIST [G ]
    -- Alle elemente, in einer Ordnung,
    -- die bedingungen erfüllt.
  ...
  ensure
    kompatibel (Result, bedingungen)
end
```

Ein wenig mathematischer Hintergrund...



Binäre Relationen auf einer Menge

Eine Eigenschaft zwischen zwei Elementen der Menge, die entweder erfüllt oder nicht erfüllt ist.

Beispielrelationen auf einer Menge von Personen *PERSON*:

- *Mutter* : a *Mutter* b ist erfüllt genau dann, wenn a die Mutter von b ist
- *Vater* :
- *Kind* :
- *Schwester* :
- *Geschwister* :

Bemerkung: Relationen werden in grün dargestellt.

Notation: a r b , um auszudrücken, dass r für a und b gilt.

Beispiel: Die *vor*-Relation

Die Menge:

Aufgaben =

{Politik, Essen, Medikament, Abwaschen, Üetliberg}

Die einschränkende Relation:

Abwaschen vor Politik

Üetliberg vor Essen

Medikament vor Essen

Essen vor Abwaschen

“Das Abwaschen kommt *vor* der politischen Diskussion”

“Die Wanderung auf den Üetliberg kommt *vor* dem Essen”

“Das Medikament muss *vor* dem Essen eingenommen werden”

“Das Essen kommt *vor* dem Abwaschen”

Einige spezielle Relationen auf einer Menge X

universal $[X]$: ist für jedes Paar von Elementen in X erfüllt

id $[X]$: ist für jedes Element in X und sich selbst erfüllt

empty $[X]$: ist für kein Paar von Elementen in X erfüllt

Relationen: präzisere mathematische Betrachtung



Wir betrachten eine Relation r auf einer Menge P als:

Kartesisches Produkt

Eine Menge von Paaren aus $P \times P$,
die alle Paare $[x, y]$ enthält, so dass $x \underline{r} y$

Dann heisst $x \underline{r} y$ nichts anderes als $[x, y] \in r$

Siehe Beispiele auf der nächsten Folie

Eine Relation ist eine Menge: Beispiele

son = {[Charles, Elizabeth], [Charles, Philip],
[William, Charles], [Harry, Charles]}

Natürliche Zahlen

id $[\mathbb{N}]$ = {[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], ...}

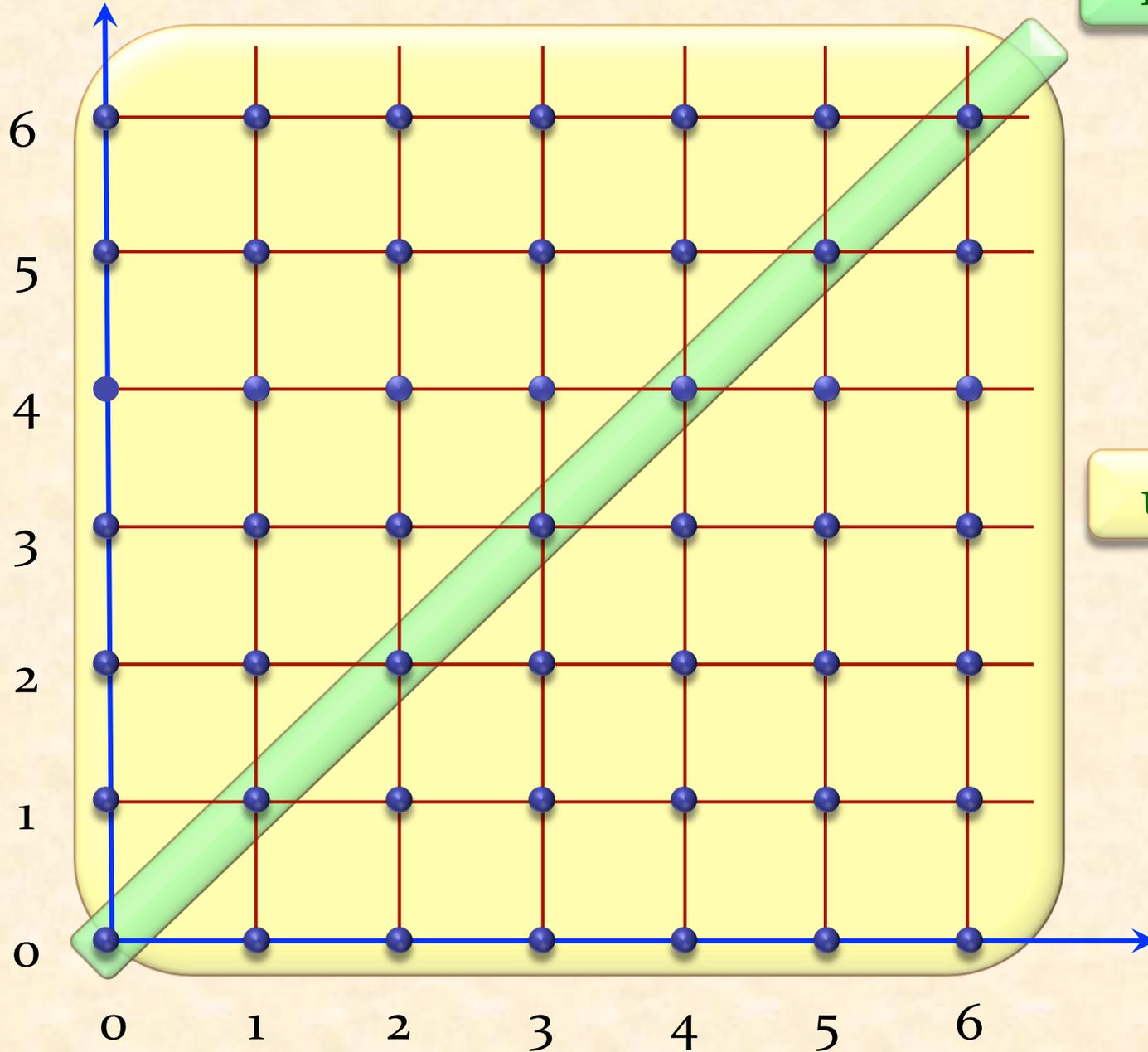
universal $[\mathbb{N}]$ = {[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], ...
[1, 0], [1, 1], [1, 2], [1, 3], [1, 4], ...
[2, 0], [2, 1], [2, 2], [2, 3], [2, 4], ...
... }

= $\mathbb{N} \times \mathbb{N}$

empty $[X]$ = \emptyset

= $\{\}$

Illustration der Relationen



id [N]

universal [N]

Beispiel: Die *vor*-Relation

“Das Abwaschen kommt *vor* der politischen Diskussion”
“Die Wanderung auf den Üetliberg kommt *vor* dem Essen”
“Das Medikament muss *vor* dem Essen eingenommen werden”
“Das Essen kommt *vor* dem Abwaschen”

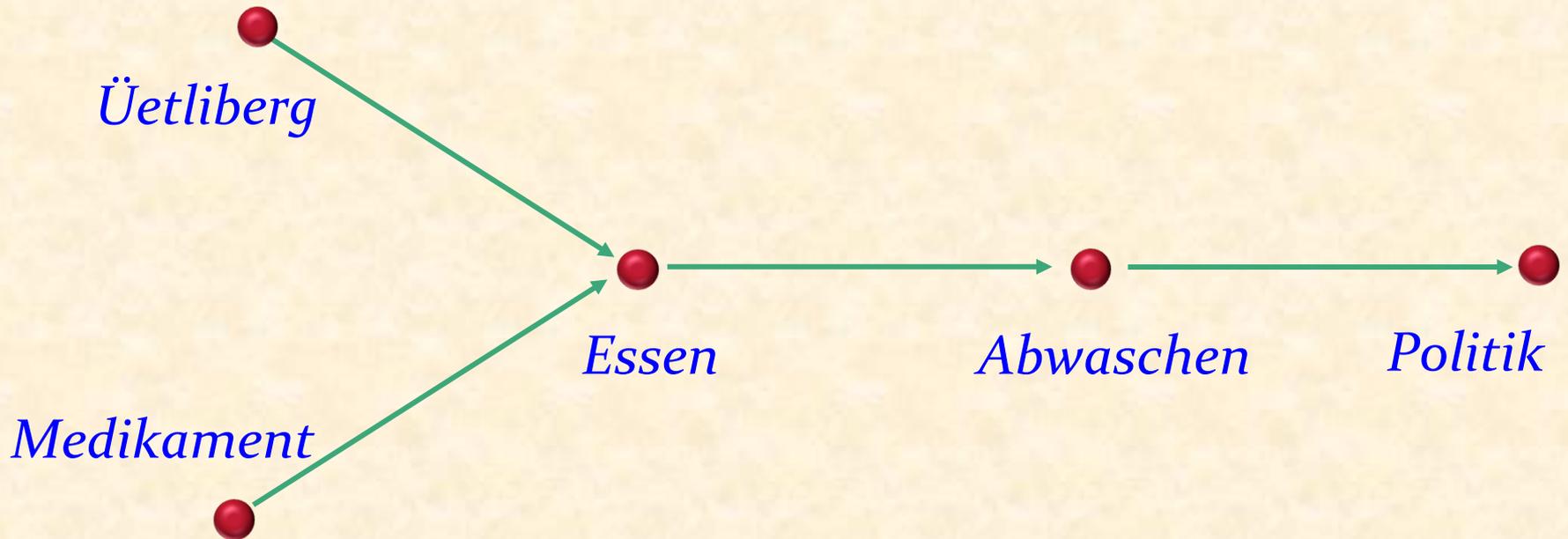
Unsere Menge:

elemente = {Politik, Essen, Medikament,
Abwaschen, Üetliberg}

Die einschränkende Relation:

vor = {[Abwaschen, Politik], [Üetliberg, Essen],
[Medikament, Essen], [Essen, Abwaschen]}

Als Graph dargestellt



- “Das Abwaschen kommt *vor* der politischen Diskussion”
- “Die Wanderung auf den Üetliberg kommt *vor* dem Essen”
- “Das Medikament muss *vor* dem Essen eingenommen werden”
- “Das Essen kommt *vor* dem Abwaschen”

Einfache Operationen auf eine Menge



Konvention: Sie sind ihr eigenes Geschwister.

Gatte = *Ehefrau* \cup *Ehemann*

Geschwister = *Schwester* \cup *Bruder* \cup *id* [*Person*]

Schwester \subseteq *Geschwister*

Vater \subseteq *Vorfahre*

universal [*X*] = *X* x *X*

empty [*X*] = \emptyset

Zusammensetzung (Komposition)

Falls r und s Relationen sind, ist deren **Zusammensetzung**

$$r ; s$$

die Relation t , so dass

$$a \underline{t} c$$

dann und genau dann gilt, wenn ein Element b existiert, so dass

$$a \underline{r} b \wedge b \underline{s} c$$

Besondere Fälle: Potenzen

$$r^1 = r$$

$$r^2 = r ; r$$

$$r^3 = r^2 ; r = r ; r ; r$$

usw.



Mögliche Eigenschaften einer Relation

(Auf einer Menge X . Alle Definitionen müssen für **jedes Element** von X erfüllt sein.)

Total*: $(a \neq b) \Rightarrow ((a \underline{r} b) \vee (b \underline{r} a))$

Reflexiv: $a \underline{r} a$

Irreflexiv: **not** $(a \underline{r} a)$

Symmetrisch: $a \underline{r} b \Rightarrow b \underline{r} a$

Antisymmetrisch: $(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$

Asymmetrisch: **not** $((a \underline{r} b) \wedge (b \underline{r} a))$

Transitiv: $(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

*Die Definition von “total” ist spezifisch für diese Diskussion (es gibt dafür keine Standarddefinition). Die restlichen Begriffe sind standardisiert.

Beispiele (auf einer Menge von Personen)

Geschwister

Reflexiv, symmetrisch, transitiv

Schwester

irreflexiv

Familienoberhaupt

reflexiv, antisymmetrisch

(*a Familienoberhaupt b* heisst, dass *a* das Oberhaupt von *b*'s Familie ist. Ein Oberhaupt pro Familie.)

Mutter

asymmetrisch, irreflexiv

Total: $(a \neq b) \Rightarrow (a \underline{r} b) \vee (b \underline{r} a)$

Reflexiv: $a \underline{r} a$

Irreflexiv: **not** $(a \underline{r} a)$

Symmetrisch: $a \underline{r} b \Rightarrow b \underline{r} a$

Antisymmetrisch: $(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$

Asymmetrisch: **not** $((a \underline{r} b) \wedge (b \underline{r} a))$

Transitiv: $(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

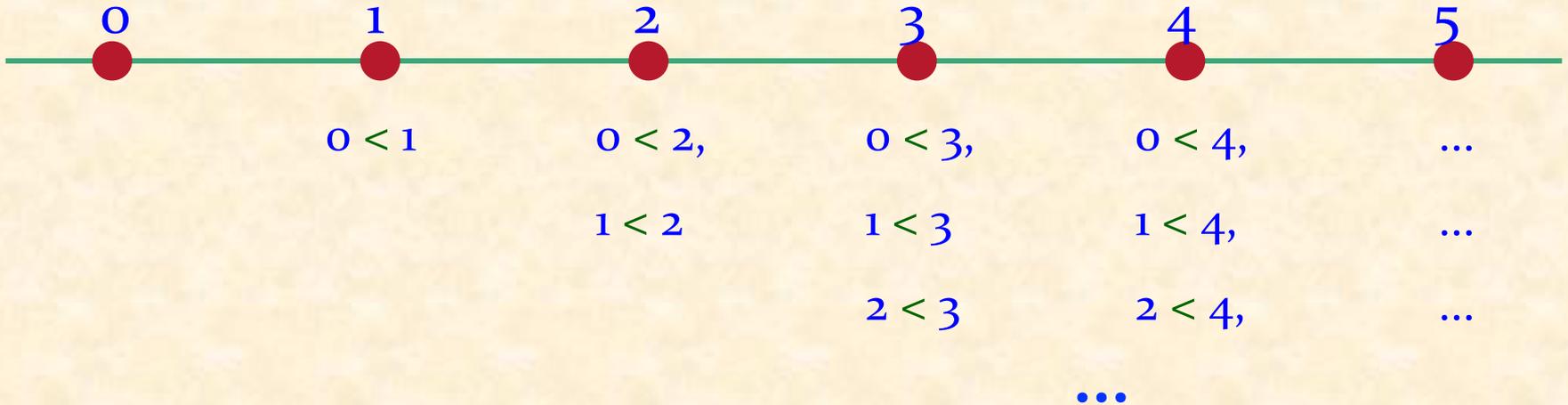
Totale Ordnungsrelation (strikt)

Eine Relation ist eine **strikte totale Ordnung** falls sie folgende Eigenschaften erfüllt:

- Total
- Irreflexiv
- Transitiv

Total:	$(a \neq b) \Rightarrow (a \underline{r} b) \vee (b \underline{r} a)$
Reflexiv:	$a \underline{r} a$
Irreflexiv:	not $(a \underline{r} a)$
Symmetrisch:	$a \underline{r} b \Rightarrow b \underline{r} a$
Antisymmetrisch:	$(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$
Asymmetrisch:	not $((a \underline{r} b) \wedge (b \underline{r} a))$
Transitiv:	$(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

Beispiel: “kleiner als” $<$ auf natürlichen Zahlen





Eine strikte (totale) Ordnungsrelation ist **asymmetrisch**

Totale Ordnungsrelation (strikt)

Eine Relation ist eine **strikte totale Ordnungsrelation**, falls sie folgende Eigenschaften erfüllt:

- Total
- Irreflexiv
- Transitiv

Total: $(a \neq b) \Rightarrow (a \underline{r} b) \vee (b \underline{r} a)$

Reflexiv: $a \underline{r} a$

Irreflexiv: **not** $(a \underline{r} a)$

Symmetrisch: $a \underline{r} b \Rightarrow b \underline{r} a$

Antisymmetrisch: $(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$

Asymmetrisch: **not** $((a \underline{r} b) \wedge (b \underline{r} a))$

Transitiv: $(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

Theorem: Eine strikte (totale) Ordnungsrelation ist **asymmetrisch**.

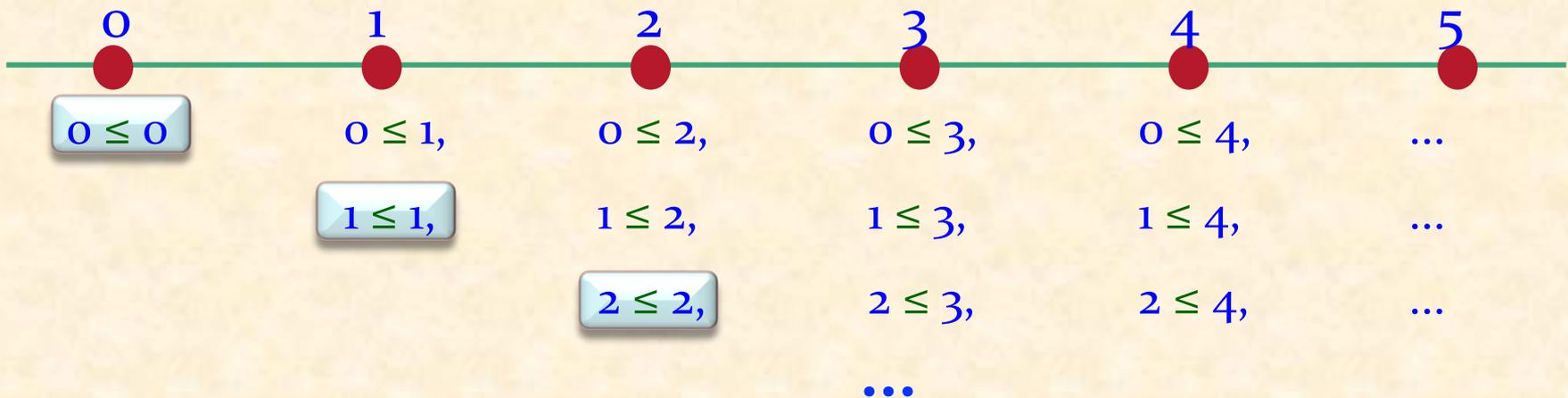
Totale Ordnungsrelation (nicht-strikt)

Eine Relation ist eine **nicht-strikt** Totale Ordnungsrelation hat die folgenden Eigenschaften:

- Total
- Reflexiv
- Transitiv
- **Antisymmetrisch**

Total:	$(a \neq b) \Rightarrow (a \underline{r} b) \vee (b \underline{r} a)$
Reflexiv:	$a \underline{r} a$
Irreflexiv:	not $(a \underline{r} a)$
Symmetrisch:	$a \underline{r} b \Rightarrow b \underline{r} a$
Antisymmetrisch:	$(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$
Asymmetrisch:	not $((a \underline{r} b) \wedge (b \underline{r} a))$
Transitiv:	$(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

Beispiel: “kleiner als oder gleich” \leq auf natürlichen Zahlen



Totale Ordnungsrelation (strikt)

Eine Relation ist eine strikte totale Ordnungsrelation, falls sie folgende Eigenschaften erfüllt:

- Total
- Irreflexiv
- Transitiv

Total: $(a \neq b) \Rightarrow (a \underline{r} b) \vee (b \underline{r} a)$

Reflexiv: $a \underline{r} a$

Irreflexiv: **not** $(a \underline{r} a)$

Symmetrisch: $a \underline{r} b \Rightarrow b \underline{r} a$

Antisymmetrisch: $(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$

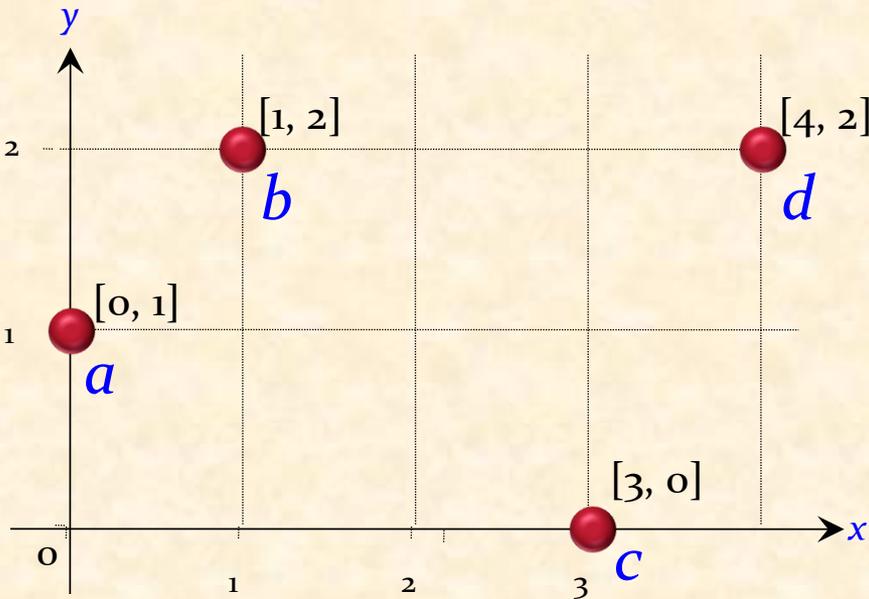
Asymmetrisch: **not** $((a \underline{r} b) \wedge (b \underline{r} a))$

Transitiv: $(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

Partielle Ordnungsrelation (strikt)

Eine Relation ist eine **strikte partielle Ordnungsrelation**, falls sie folgende Eigenschaften erfüllt:

-  total
- irreflexiv
- transitiv



- Total: $(a \neq b) \Rightarrow (a \underline{r} b) \vee (b \underline{r} a)$
- Irreflexiv: **not** $(a \underline{r} a)$
- Symmetrisch: $a \underline{r} b \Rightarrow b \underline{r} a$
- Antisymmetrisch: $(a \underline{r} b) \wedge (b \underline{r} a) \Rightarrow a = b$
- Transitiv: $(a \underline{r} b) \wedge (b \underline{r} c) \Rightarrow a \underline{r} c$

Beispiel: Relation zwischen Punkten in einer Ebene:

$p \textcircled{<} q$ falls:

- $x_p < x_q$
- $y_p < y_q$

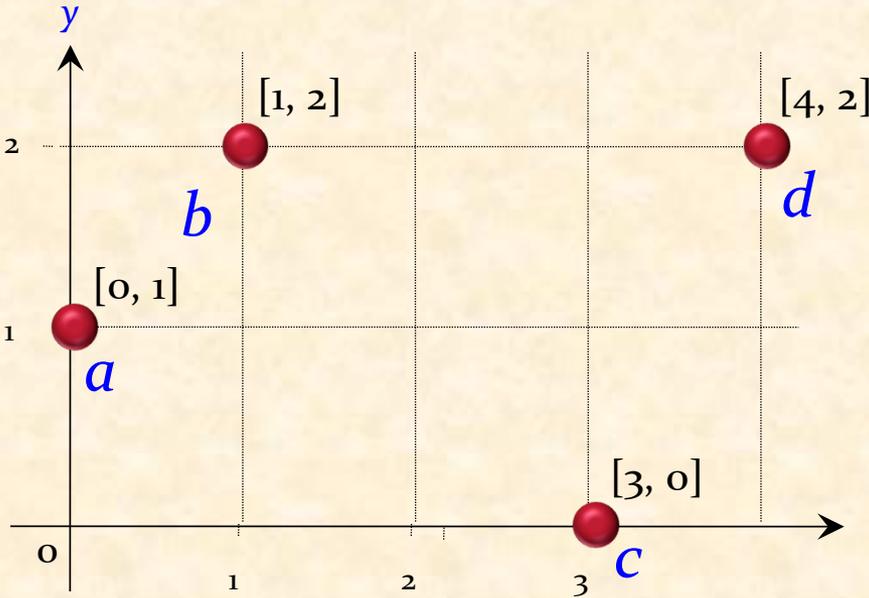


Eine strikte ~~partielle~~ (totale) Ordnungsrelation ist **asymmetrisch**

Eine totale Ordnungsrelation ist (auch!) eine partielle Ordnungsrelation

(“partiell” heisst eigentlich **möglicherweise** partiell)

Beispiel einer partiellen Ordnung



$p \triangleleft q$ falls:

- $x_p < x_q$
- $y_p < y_q$

Hier gilt:

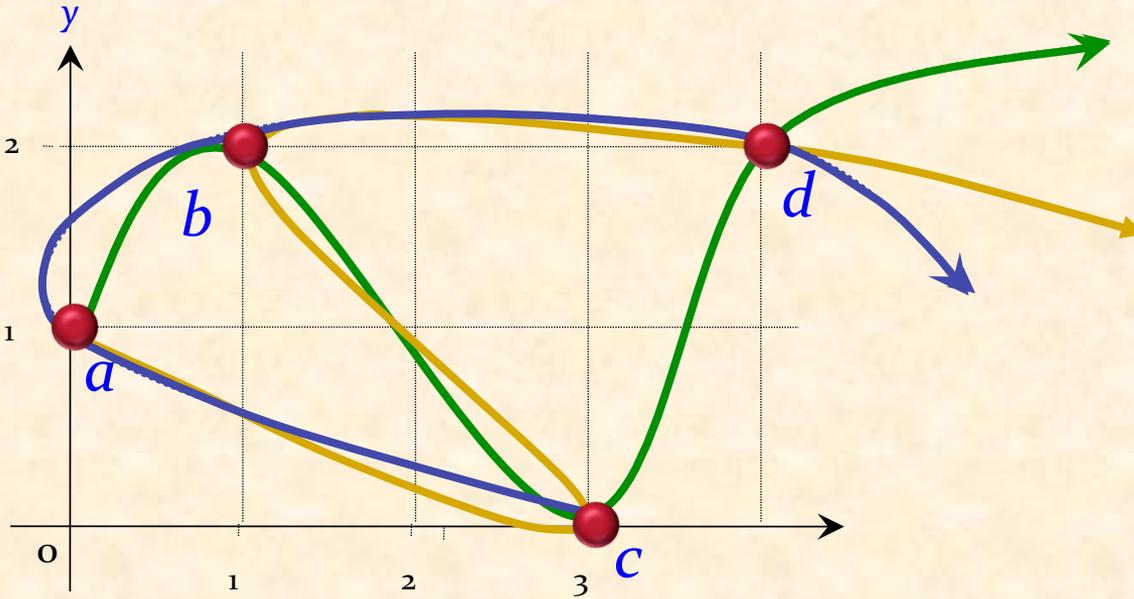
$$a \triangleleft b \quad c \triangleleft d$$

$$a \triangleleft d$$

Keine Verbindung zwischen a und c , b und c :

z.B. Weder $a \triangleleft c$ noch $c \triangleleft a$

Mögliche topologische Sortierungen



a b c d

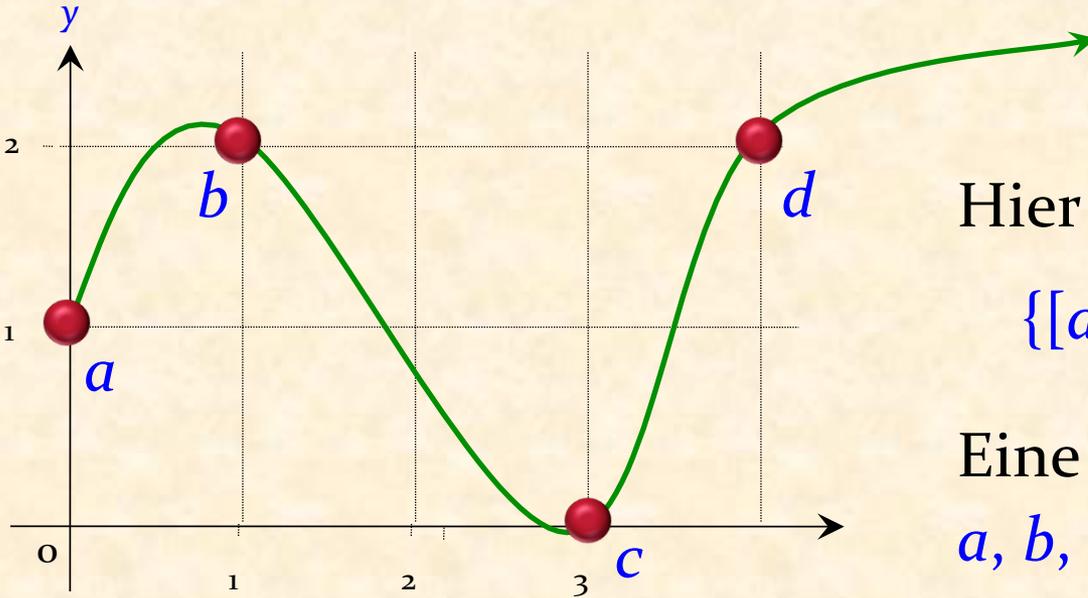
c a b d

a c b d

$a \prec b \quad c \prec d$

$a \prec d$

Topologisches Sortieren verstanden



Hier ist die Relation \prec

$$\{[a, b], [a, d], [c, d]\}$$

Eine Lösung ist:

$$a, b, c, d$$

$$a \prec b \quad c \prec d$$

$$a \prec d$$

Wir suchen eine totale
Ordnungsrelation t , so dass

$$\prec \subseteq t$$



Aus einer partiellen Ordnung eine kompatible totale Ordnung erzeugen

wobei:

Eine partielle Ordnung p ist genau dann kompatibel mit einer totalen Ordnung t , wenn

$$p \subseteq t$$

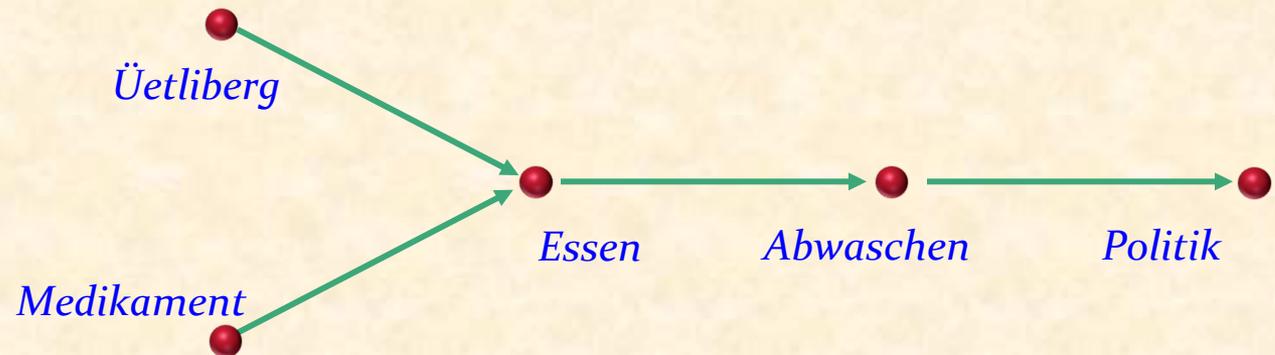
Von Bedingungen zu partiellen Ordnungen

Ist eine durch folgende Menge von Bedingungen definierte Relation, wie zum Beispiel

bedingungen =

$\{[Abwaschen, Politik], [Üetliberg, Essen], [Medikament, Essen], [Essen, Abwaschen]\}$

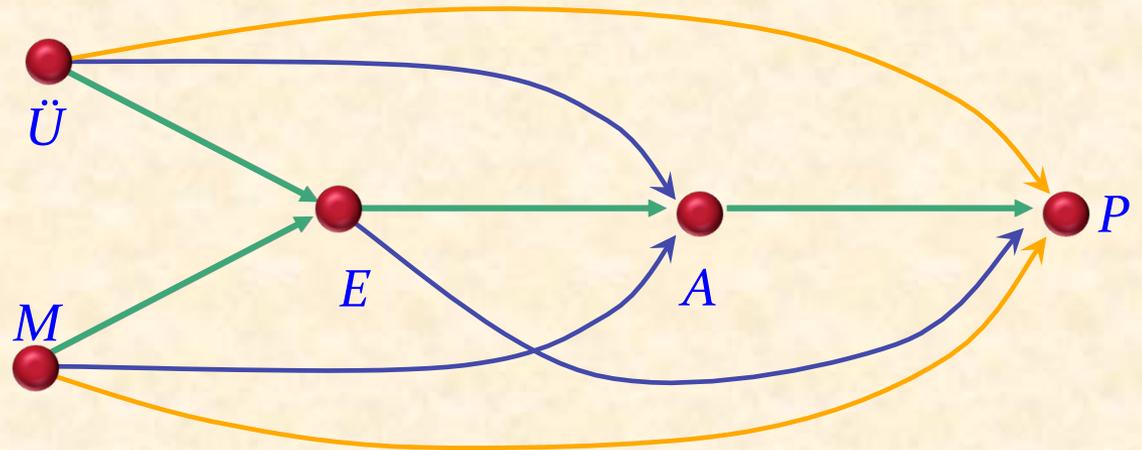
immer eine partielle Ordnung?



Potenzen und transitive Hülle von Relationen



$r^{i+1} = r^i ; r$ wobei $;$ die Zusammensetzung ist



Transitive Hülle (*Transitive closure*)

$$r^+ = r^1 \cup r^2 \cup \dots$$

-- Immer transitiv

r^1

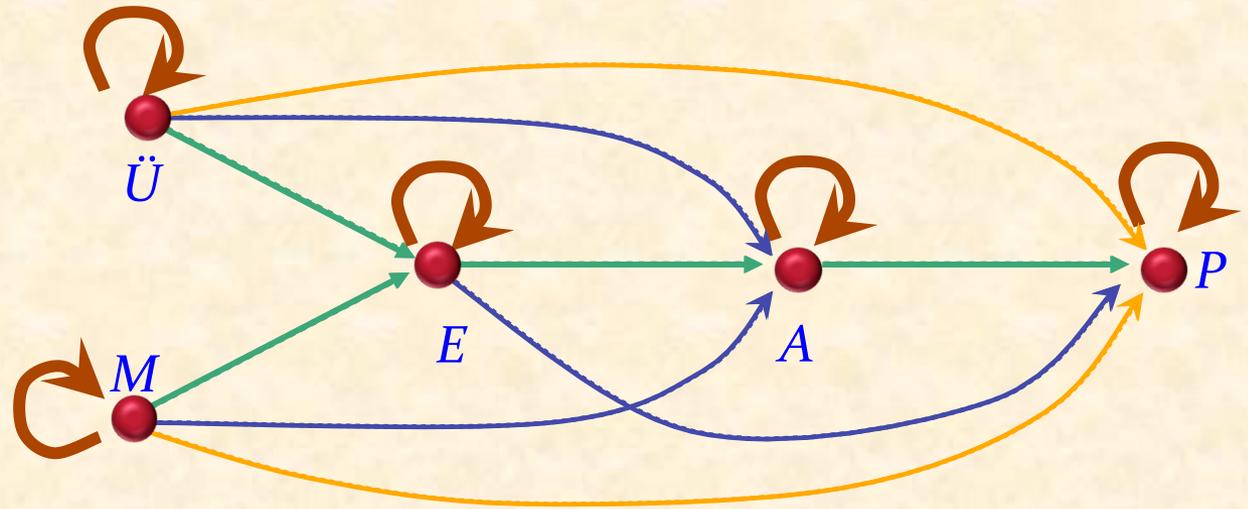
r^2

r^3

Reflexive transitive Hülle

$r^0 = id [X]$ wobei X die zugrunde liegende Menge ist

$r^{i+1} = r^i ; r$ wobei $;$ die Zusammensetzung ist



Transitive Hülle

$r^+ = r^1 \cup r^2 \cup \dots$ immer transitiv

reflexive transitive Hülle:

$r^* = r^0 \cup r^1 \cup r^2 \cup \dots$

-- Immer transitiv **und reflexiv**

r^0

r^1

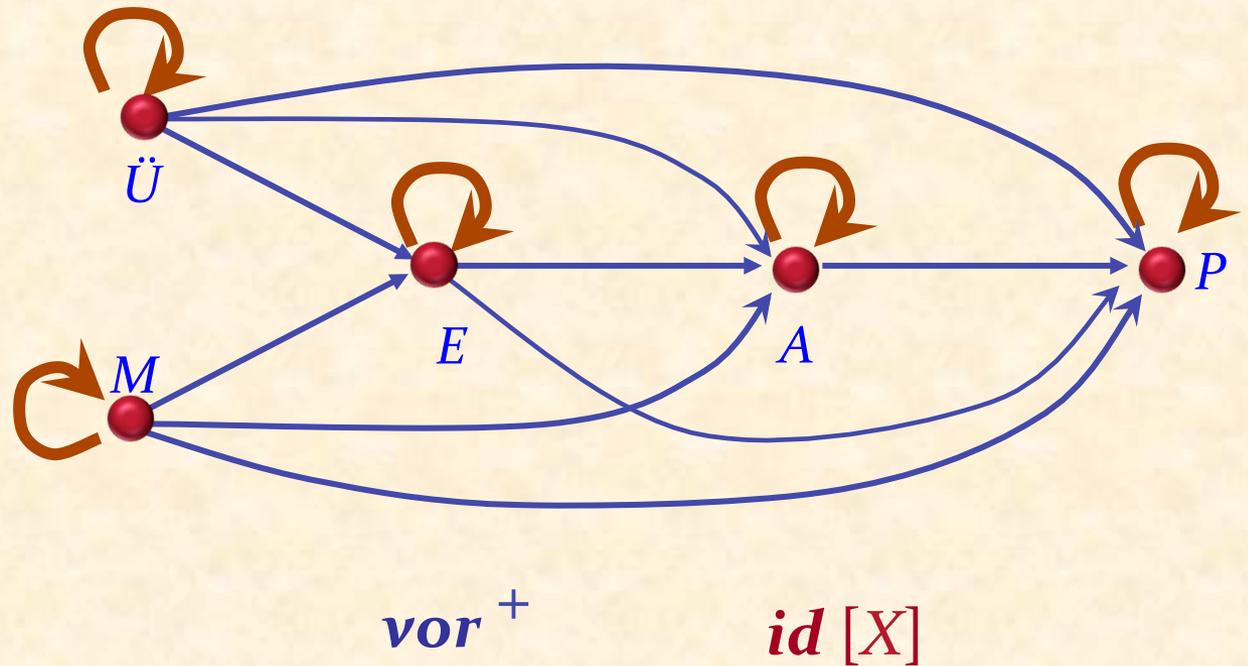
r^2

r^3

Azyklische Relation

Eine Relation r auf einer Menge X ist azyklisch genau dann, wenn:

$$r^+ \cap id[X] = \emptyset$$





Theoreme:

- Jede (strikte) Ordnungsrelation ist azyklisch
- Eine Relation ist azyklisch genau dann wenn ihre transitive Hülle eine (strikte) Ordnung ist

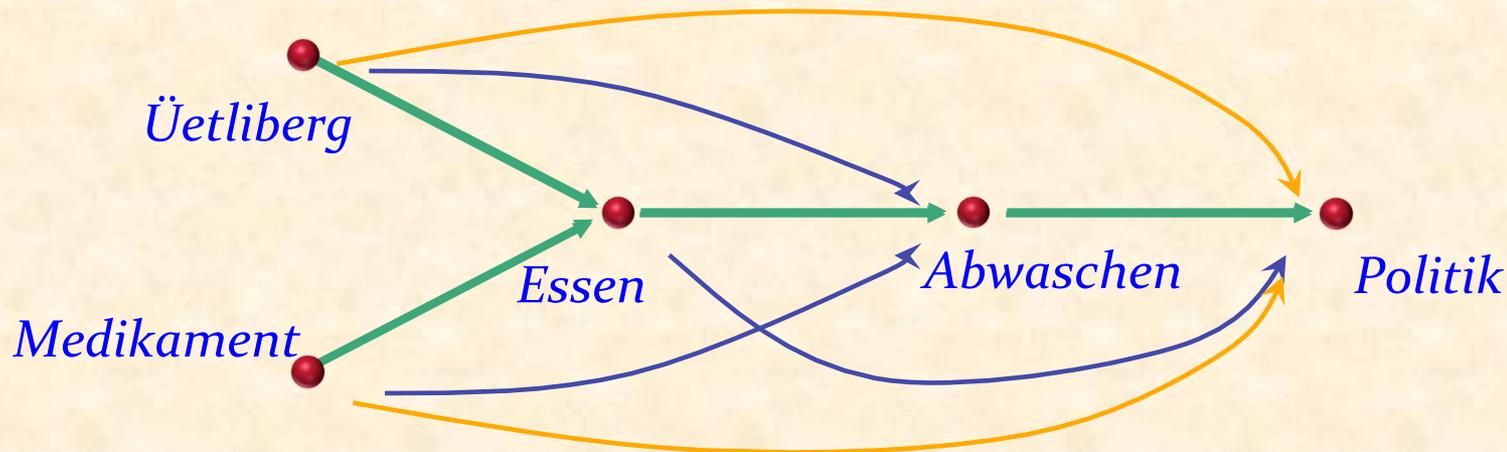
(auch: Genau dann, wenn ihre reflexive transitive Hülle eine nicht-strikte partielle Ordnung ist)

Von den Bedingungen zur partiellen Ordnung

Unsere partielle Ordnungsrelation ist **vor**+

vor =

{[Abwaschen, Politik], [Üetliberg, Essen],
[Medikament, Essen], [Essen, Abwaschen]}





Das Problem des topologischen Sortierens und seine Anwendungen

Mathematischer Hintergrund:

- Relationen als Mengen von Paaren
- Eigenschaften einer Relation
- Ordnungsrelationen: partiell/total, strikt/nicht-strikt
- Transitiv und reflexive transitive Hüllen
- Die Beziehung zwischen **azyklischen** und **Ordnungsrelationen**
- Die Grundidee des topologischen Sortierens

Als Nächstes: Wie man es implementiert:

- Effizient: $O(m + n)$ für m Bedingungen und n Elemente
- Gutes Software-Engineering: effektives API

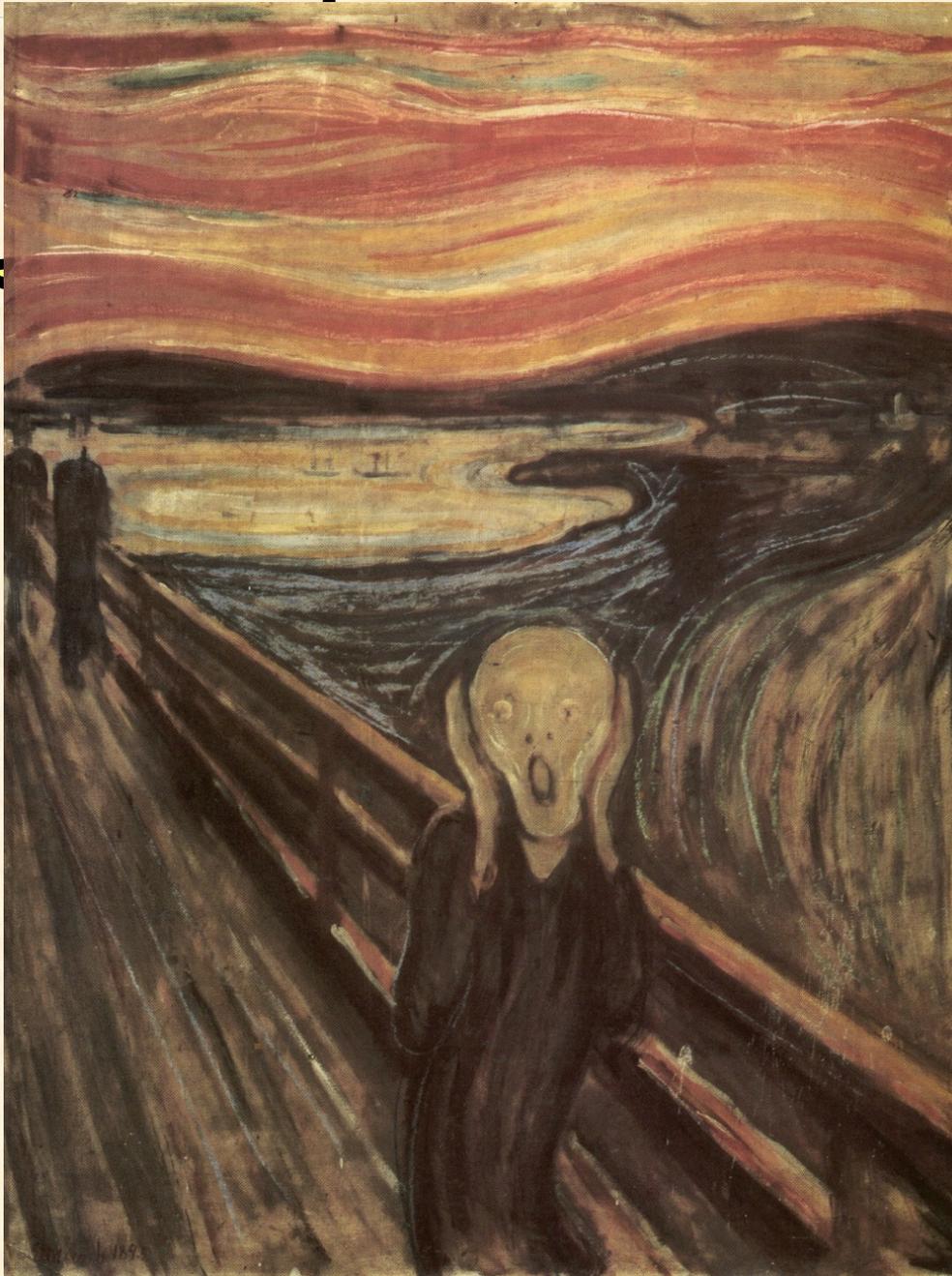


Einführung in die Programmierung

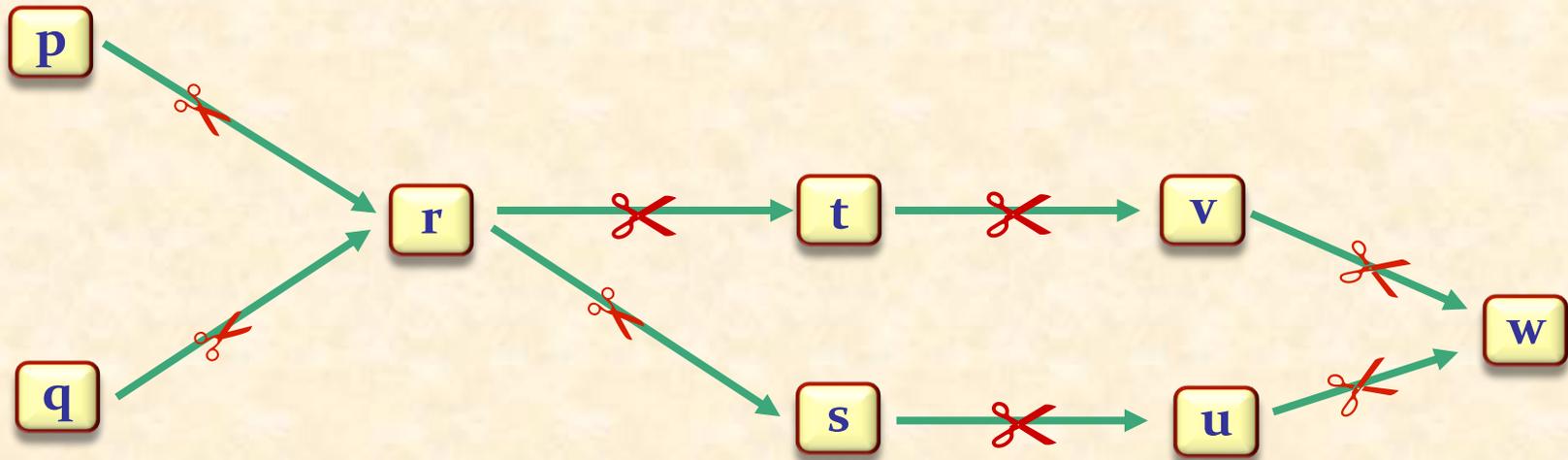
Prof. Dr. Bertrand Meyer

Lektion 15: Topologisches Sortieren
Teil 2: Algorithmus und Implementation

Zurück zu Software



Die Grundidee des Algorithmus



topsort



Allgemeine Struktur (erster Versuch)



Gegeben:

Ein Typ G

Eine Menge von
Elementen des Typs G

Eine Relation *bedingungen*
auf diesen Elemente

Benötigt:

Eine Aufzählung der
Elemente, kompatibel mit
bedingungen

```
class
  TOPOLOGISCH_SORTIERBAR [G ]
feature
  bedingungen : LINKED_LIST [TUPLE [G, G ]]
  elemente : LINKED_LIST [G ]

  topologisch_sortiert : LINKED_LIST [G ]
  require
    zyklus_frei (bedingungen)
  do
    ...
  ensure
    kompatibel (Result, bedingungen)
  end
end
```



Wir benutzen statt einer Funktion *topologisch_sortiert*:

- Eine Prozedur *sortiere*
- Ein Attribut *sequenz* (gesetzt von *sortiere*), welches das Resultat enthält

```
class
  TOPOLOGISCH_SORTIERER [G]
feature
  bedingungen : LINKED_LIST [TUPLE [G, G]]
  elemente : LINKED_LIST [G]

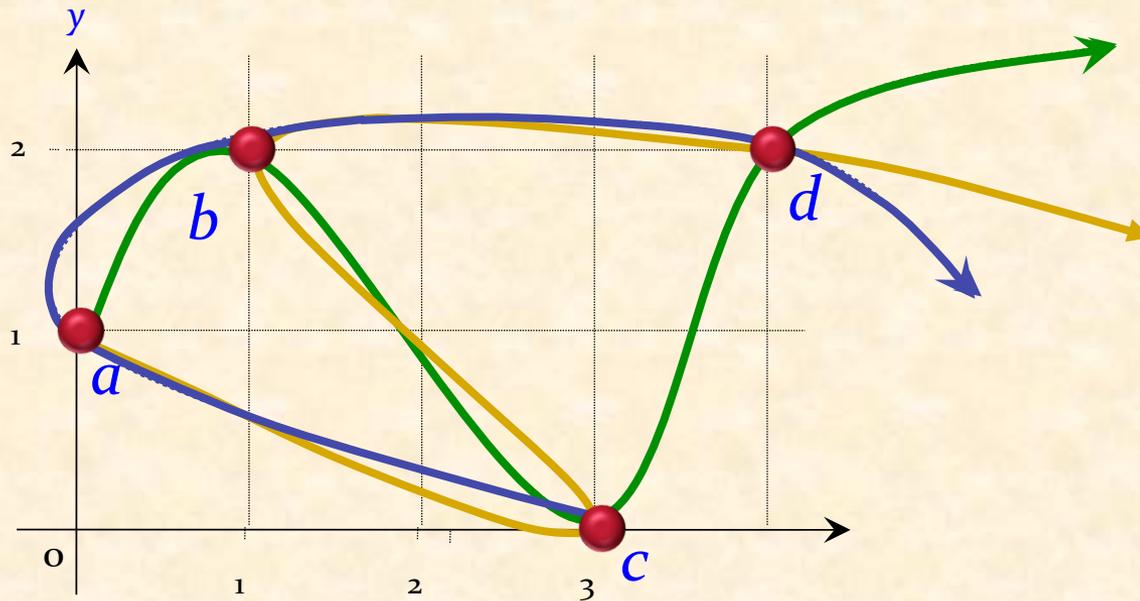
  sequenz : LINKED_LIST [G]

  sortiere
    require
      zyklus_frei (bedingungen)
    do
      ...
    ensure
      kompatibel (sequenz, bedingungen)
    end
end
```

Fehlende Eindeutigkeit



Im Allgemeinen gibt es mehrere Lösungen



a b c d
c a b d
a c b d

In der Praxis benutzt das topologische Sortieren ein Optimierungskriterium, um zwischen möglichen Lösungen zu entscheiden.



Die Relation \odot :

- Muss eine Partielle Ordnung sein: Keine Zyklen in der transitiven Hülle von *bedingungen*
- Das heisst es gibt keine zirkuläre Liste der Form

$$e_0 \odot e_1 \odot \dots \odot e_n \odot e_0$$

Falls es einen solchen Zyklus gibt, existiert keine Lösung für das Problem!

Zyklen



Bei der topologischen Sortierung erhalten wir nicht die eigentliche Relation \prec , sondern eine Relation *bedingungen*, durch eine Menge von Paaren wie

$\{[Abwaschen, Hinaus], [Museum, Essen], [Medikament, Essen], [Essen, Abwaschen]\}$

Die Relation, an der wir interessiert sind, ist:



\prec ist azyklisch gdw *bedingungen* keine Paare der Form

$\{[f_0, f_1], [f_1, f_2], \dots, [f_m, f_0]\}$

enthält

Falls ein solcher Zyklus existiert, kann es keine mit *bedingungen* kompatible Ordnung geben



```
class
  TOPOLOGISCH_SORTIERER [G ]
feature
  bedingungen : LINKED_LIST [TUPLE [G, G]]
  elemente : LINKED_LIST [G ]

  sequenz : LINKED_LIST [G ]

  sortiere
    require
      zyklus_frei (bedingungen)
    do
      ...
    ensure
      kompatibel (sequenz, bedingungen)
    end
end
```



sortiere

require

zyklus_frei (bedingungen)

do

...

ensure

kompatibel (sequenz, bedingungen)

end

Dies nimmt an, dass der Input keine Zyklen beinhaltet

Eine solche Annahme ist in der Praxis nicht durchsetzbar

Im Speziellen: Das Finden von Zyklen ist in Wirklichkeit gleich schwierig wie das topologische Sortieren!

Der Umgang mit Zyklen



Wir nehmen gar nichts an, sondern finden die Zyklen als Nebenprodukt beim topologischen Sortieren

Das Schema für *sortiere* ergibt sich also zu:

```
--"Versuche topologische Sortierung;  
-- Zyklen werden berücksichtigt."
```

```
if "Zyklen gefunden" then  
    "Zyklen ausgeben"  
end
```

Allgemeine Struktur (vorher)

```
class
  TOPOLOGISCH_SORTIERER [G ]
feature
  bedingungen : LINKED_LIST [ TUPLE [G, G ] ]
  elemente : LINKED_LIST [G ]
  sequenz : LINKED_LIST [G ]

  sortiere
    require
      zyklus_frei (bedingungen)
    do
      ...
    ensure
      kompatibel (sequenz, bedingungen)
      “sequenz enthält alle Elemente”
    end
  end
end
```

Allgemeine Struktur (endgültig)

class

TOPOLOGISCH_SORTIERER [G]

feature

bedingungen : LINKED_LIST [TUPLE [G, G]]

elemente : LINKED_LIST [G]

sequenz : LINKED_LIST [G]

sortiere

require

-- Keine Vorbedingung in dieser Version

do

...

ensure

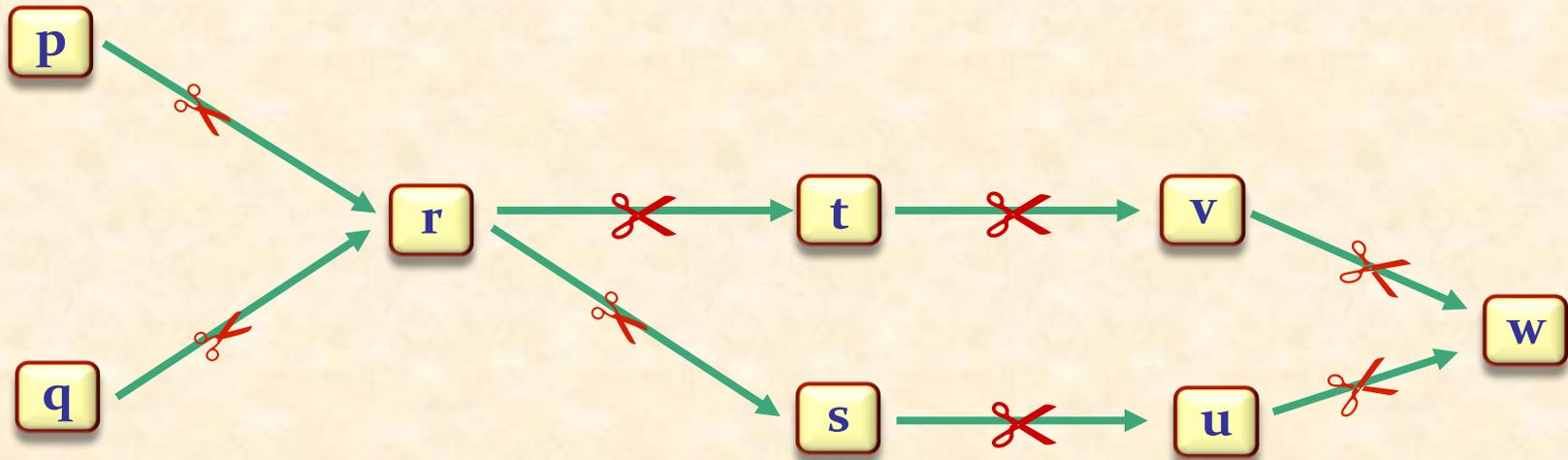
kompatibel (sequenz, bedingungen)

“*sequenz* enthält alle Elemente, die zu Beginn nicht in einen Zyklus involviert sind.”

end

end

Erinnerung: Die Grundidee



topsort





...

loop

“Finde ein Element *next* in *elemente*, für das *bedingungen* keine Paare der Form $[x, next]$ beinhaltet”

sequenz.extend(next)

“Lösche *next* aus *elemente*, und entferne alle Paare der Form $[next, y]$ aus *bedingungen* ”

end



Invariante in unserem ersten Versuch:

“*bedingungen*⁺ enthält keine Zyklen”

Invariante in der verbesserten Architektur:

“*bedingungen*⁺ enthält keine Zyklen, die nicht ursprünglich schon enthalten waren”

Allgemeiner:

bedingungen⁺ ist eine Teilmenge des ursprünglichen *bedingungen*⁺



Falls *bedingungen* ein Paar $[x, y]$, beinhaltet, sagen wir

- x ist ein **Vorgänger** von y
- y ist ein **Nachfolger** von x

Das Schema des Algorithmus



sortiere

do

from

create {...} *sequenz.make*

invariant

"*bedingungen* enthält nur ursprüngliche Zyklen" **and**

"*sequenz* ist kompatibel mit *bedingungen*" **and**

"Alle ursprünglichen Elemente sind in *sequenz* oder *elemente*"

until

Jedes Element von *elemente* hat einen Vorgänger"

loop

next := "Ein Element von *elemente* ohne Vorgängern"

sequenz.extend (*next*)

"Lösche *next* aus *elemente*"

"Lösche alle Paare [*next*, *y*] aus *bedingungen*"

variant

"Grösse von *elemente*"

end

if "Keine Elemente übrig" **then**

"Berichte, dass das topologische Sortieren abgeschlossen ist"

else

"Melde Zyklen in den übrigen *bedingungen* und *elemente*"

end

end

Den Algorithmus implementieren



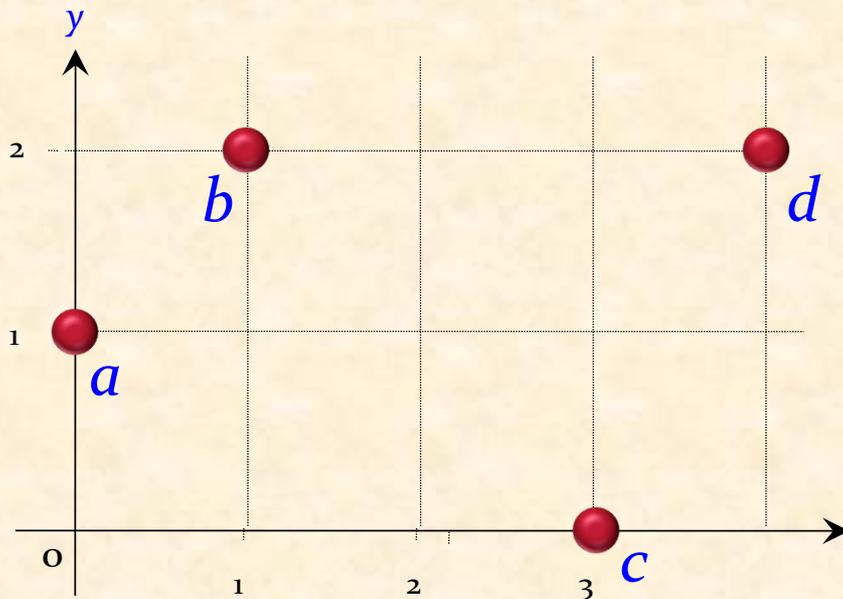
Wir beginnen mit diesen Datenstrukturen, die den Input direkt widerspiegeln:

elemente: LINKED_LIST [G]

bedingungen: LINKED_LIST [TUPLE [G, G]]

(Anzahl Elemente: n)

Anzahl Bedingungen: m)



Beispiel:

elemente = {a, b, c, d}

bedingungen =

{[a, b], [a, d], [b, d], [c, d]}

Datenstrukturen 1: ursprünglich



$elemente = \{a, b, c, d\}$

$bedingungen = \{[a, b], [a, d], [b, d], [c, d]\}$

elemente



n Elemente

bedingungen



m Bedingungen

Effizienz: Das Beste, das wir uns erhoffen können, ist:

$O(m + n)$

Grundoperationen



sortiere
do

from

create {...} *sequenz*.make

invariant

"*bedingungen* enthält nur ursprüngliche Zyklen" and

"*sequenz* ist kompatibel mit *bedingungen*" and

"Alle ursprünglichen Elemente sind in *sequenz* oder *elemente*"

until

Jedes Element von *elemente* hat einen Vorgänger"

loop

next := "Ein Element von *elemente* ohne Vorgängern"

sequenz.extend(*next*)

"Lösche *next* aus *elemente*"

"Lösche alle Paare [*next*, *y*] aus *bedingungen*"

variant

"Grösse von *elemente*"

end

if "Keine Elemente übrig" then

"Berichte, dass das topologische Sortieren abgeschlossen ist"

else

"Melde Zyklen in den übrigen *bedingungen* und *elemente*"

end

end

Die Operationen, die wir brauchen (n mal)



- Herausfinden, ob es ein Element ohne Vorgängern gibt. (Und dann eines davon nehmen)
- Ein gegebenes Element von der Menge der Elemente entfernen
- Alle Bedingungen, die mit einem gegebenen Element beginnen, aus der Menge der Bedingungen entfernen
- Herausfinden, ob noch ein Element vorhanden ist



Das Beste, das wir uns erhoffen können

$$O(m + n)$$

(da wir jede Bedingung und jedes Element mindestens einmal betrachten müssen)

Grundoperationen

sortiere
do

from

create {...} *sequenz*.make

invariant

"*bedingungen* enthält nur ursprüngliche Zyklen" and

"*sequenz* ist kompatibel mit *bedingungen*" and

"Alle ursprünglichen Elemente sind in *sequenz* oder *elemente*"

until

n mal

Jedes Element von *elemente* hat einen Vorgänger"

n mal

loop

next := "Ein Element von *elemente* ohne Vorgängern"

sequenz.extend(*next*)

n mal

"Lösche *next* aus *elemente*"

m mal

"Lösche alle Paare [*next*, *y*] aus *bedingungen*"

variant

"Grösse von *elemente*"

n mal

end

if "Keine Elemente übrig" then

"Berichte, dass das topologische Sortieren abgeschlossen ist"

else

"Melde Zyklen in den übrigen *bedingungen* und *elemente*"

end

end

Datenstrukturen 1: ursprünglich



$elemente = \{a, b, c, d\}$

$bedingungen = \{[a, b], [a, d], [b, d], [c, d]\}$

elemente



n Elemente

bedingungen



m Bedingungen

Effizienz: wenn wir *elemente* und *bedingungen* wie gegeben verwenden, können wir $O(m + n)$ nicht erreichen!

Grundoperationen

sortiere
do

from

create {...} *sequenz*.make

invariant

"*bedingungen* enthält nur ursprüngliche Zyklen" and

"*sequenz* ist kompatibel mit *bedingungen*" and

"Alle ursprünglichen Elemente sind in *sequenz* oder *elemente*"

until

n mal

Jedes Element von *elemente* hat einen Vorgänger"

n mal

loop

next := "Ein Element von *elemente* ohne Vorgängern"

sequenz.extend(*next*)

n mal

"Lösche *next* aus *elemente*"

m mal

"Lösche alle Paare [*next*, *y*] aus *bedingungen*"

variant

"Grösse von *elemente*"

n mal

end

if "Keine Elemente übrig" then

"Berichte, dass das topologische Sortieren abgeschlossen ist"

else

"Melde Zyklen in den übrigen *bedingungen* und *elemente*"

end

end



Eine bessere interne Repräsentation wählen

- Jedes Element hat eine Nummer (Dies erlaubt uns, Arrays zu benutzen)
- Wir repräsentieren *bedingungen* in einer Form, die dem entspricht, was wir von der Struktur wollen:
 - “Finde *next* , so dass *bedingungen* kein Paar der Form $[y, next]$ beinhaltet.”
 - “Gegeben *next* , lösche alle Paare der Form $[next, y]$ aus *bedingungen*.”

Schema des Algorithmus (ohne Invariante und Variante)

sortiere

do

from create {...} *sequenz.make* **until**

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente*”

”Lösche alle Paare [*next*, *y*] aus *bedingungen*”

end

if “Keine Elemente übrig” **then**

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*”

end

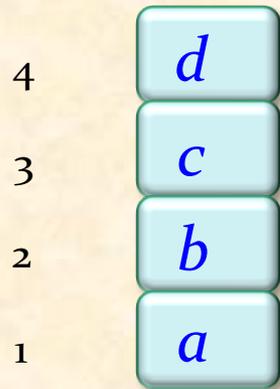
end

Datenstruktur 1: *elemente* repräsentieren



elemente : ARRAY [G]

-- (Ersetzt die ursprüngliche Liste)



elemente

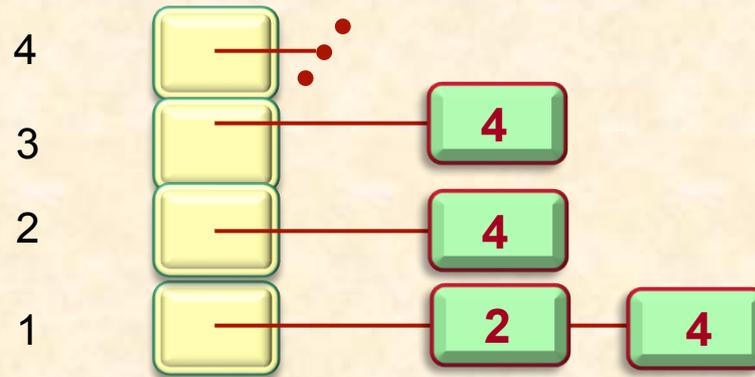
elemente = {*a*, *b*, *c*, *d*}

bedingungen = {[*a*, *b*], [*a*, *d*], [*b*, *d*], [*c*, *d*]}

Datenstruktur 2: *bedingungen* repräsentieren

nachfolger : ARRAY [LINKED_LIST [INTEGER]]

- Elemente, die *nach* einem bestimmten Element
- vorkommen müssen.



nachfolger

elemente = {a, b, c, d}

bedingungen = {[a, b], [a, d], [b, d], [c, d]}

Datenstruktur 3: *bedingungen* repräsentieren



vorgänger_zahl : ARRAY [INTEGER]

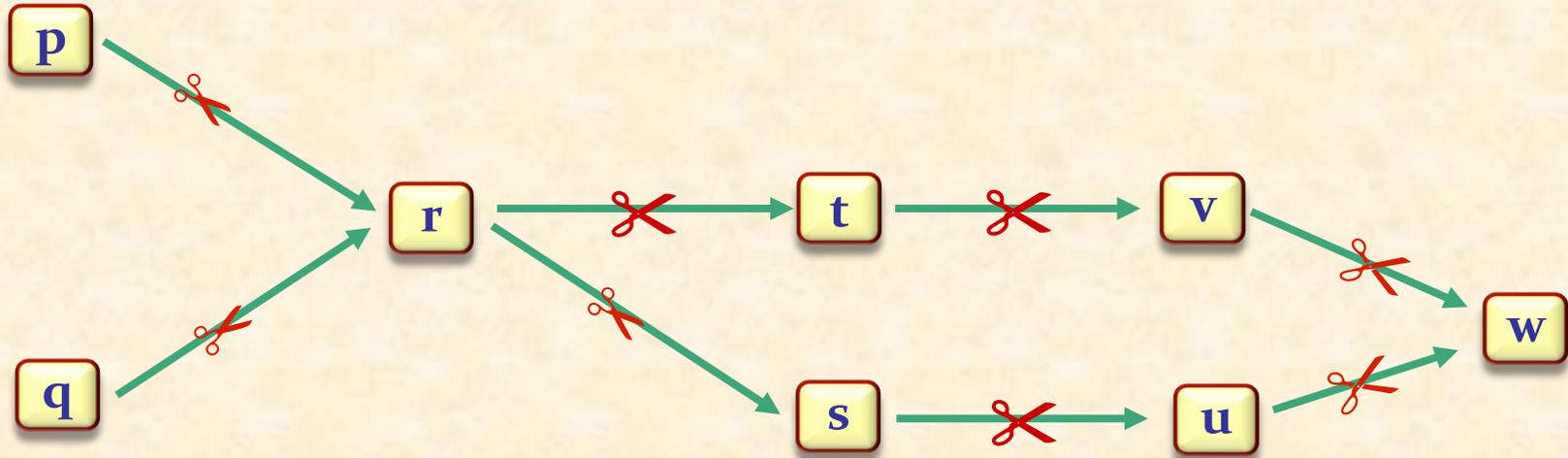
- Anzahl der Elemente, die *vor* einem bestimmten
- Element vorkommen müssen.



elemente = {*a*, *b*, *c*, *d*}

bedingungen = {[*a*, *b*], [*a*, *d*], [*b*, *d*], [*c*, *d*]}

Die Grundidee (nochmals)



topsort



Einen „Kandidaten“ finden (Element ohne Vorgängern)

sortiere

do

from create {...} *sequenz.make* until

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente*”

”Lösche alle Paare [*next*, *y*] aus *bedingungen*”

end

if “Keine Elemente übrig” then

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*”

end

end

Einen Kandidaten finden (1)



Wir implementieren

next := “Ein Element von *elemente* ohne Vorgängern“

als:

Sei *next* ein noch nicht abgearbeiteter Integer, so dass
 $vorgänger_zahl[next] = 0$

Dies benötigt $O(n)$ zum Suchen durch alle Indices: schlecht!

Aber Moment...

Nachfolger löschen



sortiere
do

from create {...} *sequenz.make* **until**

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente*”

”Lösche alle Paare [*next*, *y*] aus *bedingungen*”

end

if “Keine Elemente übrig” **then**

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*”

end

end

Nachfolger löschen



Wir implementieren

“Lösche alle Paare $[next, y]$ aus $bedingungen$ ”

als Schleife über alle Nachfolger von $next$:

```
ziele := nachfolger [next ]
```

```
across ziele as
```

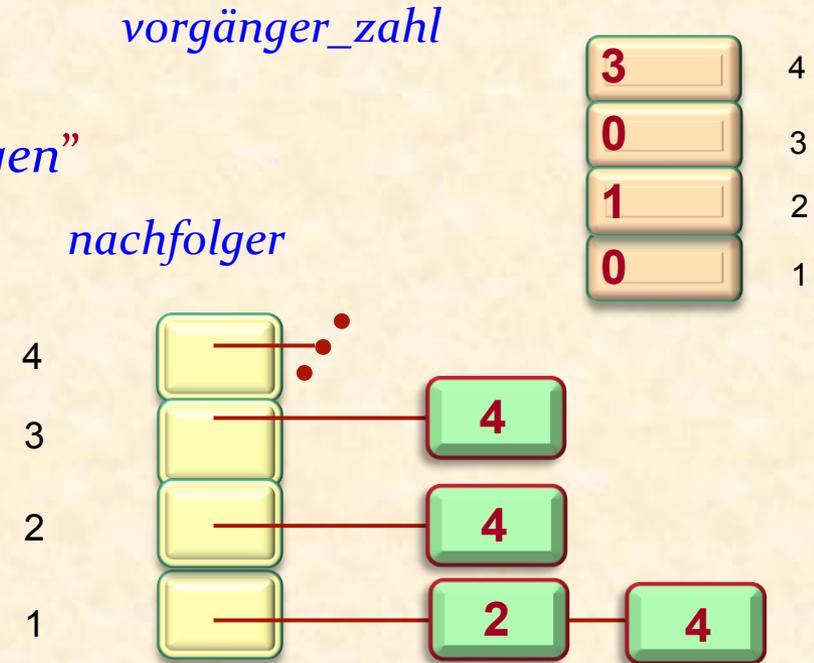
```
  c
```

```
loop
```

```
  x := c.item
```

```
  vorgänger_zahl [x] := vorgänger_zahl [x] - 1
```

```
end
```



Nachfolger löschen



```
ziele := nachfolger [next]
```

```
across ziele as
```

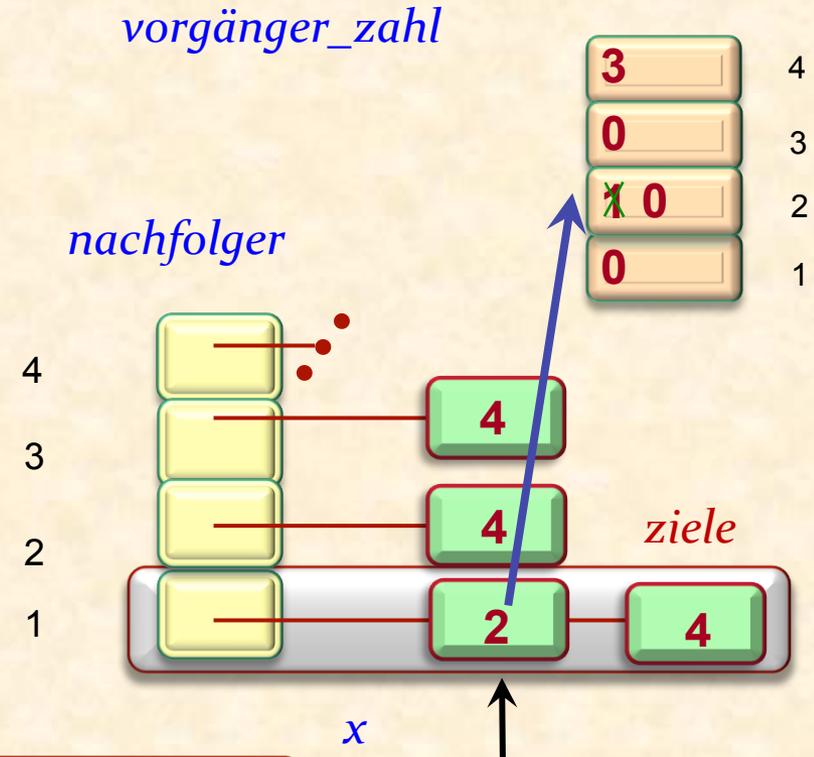
```
  c
```

```
loop
```

```
  x := c.item
```

```
  vorgänger_zahl [x] := vorgänger_zahl [x] - 1
```

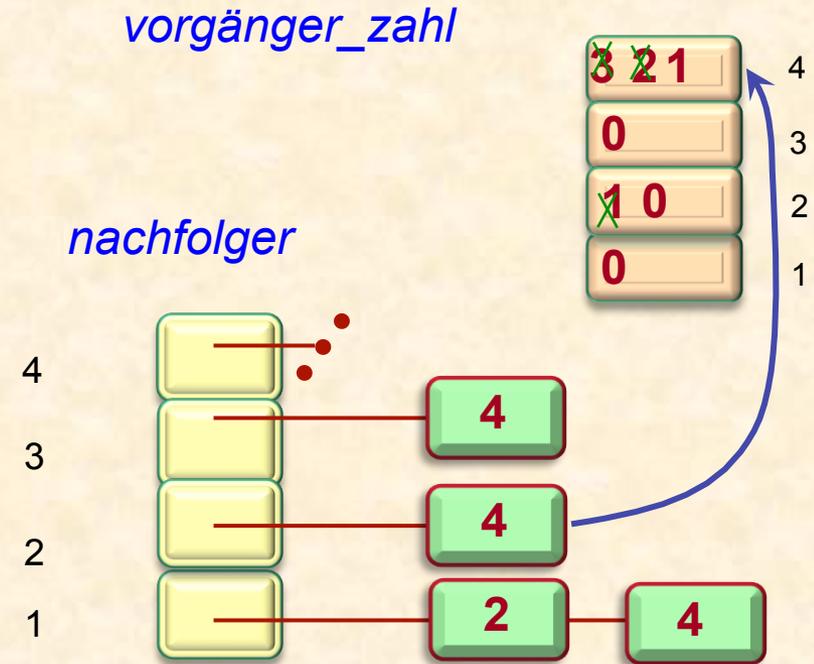
```
end
```



Nachfolger löschen



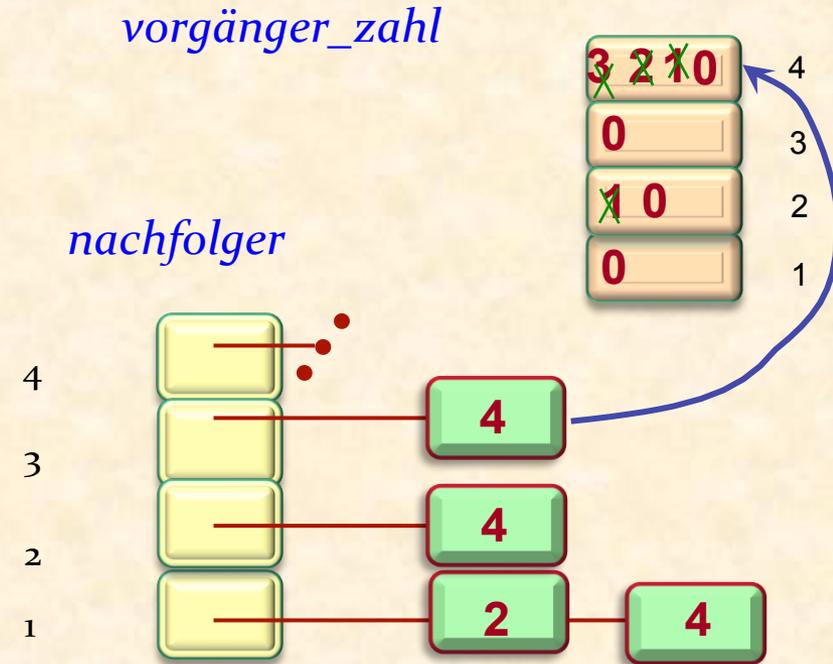
```
ziele := nachfolger [next ]  
across ziele as  
  c  
loop  
  x := c.item  
  vorgänger_zahl [x ] := vorgänger_zahl [x ] - 1  
end
```



Nachfolger löschen



```
ziele := nachfolger [next ]  
across ziele as  
  c  
loop  
  x := c.item  
  vorgänger_zahl [x ] := vorgänger_zahl [x ] - 1  
end
```



Schema des Algorithmus



sortiere
do

from create {...} *sequenz.make* **until**

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente*”

”Lösche alle Paare [*next*, *y*] aus *bedingungen*”

end

if “Keine Elemente übrig” **then**

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*”

end

end

Einen Kandidaten finden (1)



Wir implementieren

$next :=$ “Ein Element von *elemente* ohne Vorgängern“

als:

Sei $next$ ein noch nicht abgearbeiteter Integer, so dass
 $vorgänger_zahl[next] = 0$

Wir haben gesagt:

- “Es scheint $O(n)$ zu benötigen, um durch alle Indizes zu iterieren, aber Moment mal...”

Nachfolger löschen



```
ziele := nachfolger [next ]  
across ziele as
```

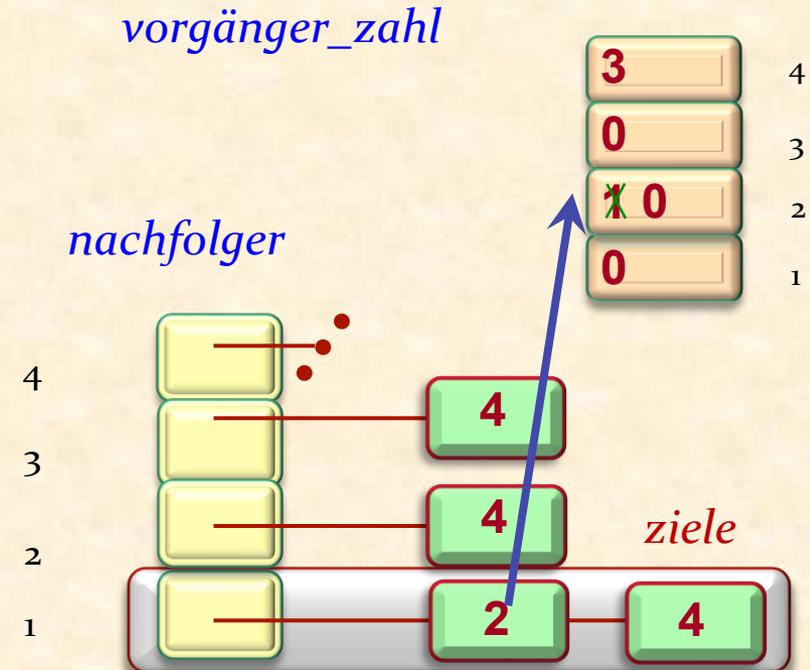
c

```
loop
```

```
  x := c.item
```

```
  vorgänger_zahl [ x ] := vorgänger_zahl [ x ] - 1
```

```
end
```



Einen Kandidaten finden (2): auf der Stelle



Wir ergänzen

```
vorgänger_zahl [x ] := vorgänger_zahl [x ] - 1
```

mit:

```
if vorgänger_zahl [x ] = 0 then
```

```
    -- Wir haben einen Kandidaten gefunden!
```

```
    kandidaten.put (x )
```

```
end
```



kandidaten : *STACK* [*INTEGER*]

-- Elemente ohne Vorgängern



Anstelle eines Stapels kann *kandidaten* auch eine andere **Dispenser**-Datenstruktur sein, z.B. ein Warteschlange

Die Wahl wird bestimmen, welche topologische Sortierung wir erhalten, falls es mehrere Möglichkeiten gibt

Schema des Algorithmus



sortiere
do

from create {...} *sequenz.make* **until**

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente* ”

”Lösche alle Paare [*next*, *y*] aus *bedingungen* ”

end

if “Keine Elemente übrig” **then**

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*.”

end

end

Einen Kandidaten finden (2)

Wir implementieren

next := “Ein Element von *elemente* ohne Vorgängern”

falls *kandidaten* nicht leer ist, als:

next := *kandidaten.item*

Schema des Algorithmus



sortiere
do

from create {...} *sequenz.make* **until**

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente*”

”Lösche alle Paare [*next*, *y*] aus *bedingungen*”

end

if “Keine Elemente übrig” **then**

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*”

end

end

Einen Kandidaten finden (3)

Wir implementieren die Abfrage

Jedes Element von *elemente* hat einen Vorgänger

als

not *kandidaten.is_empty*

Um die Abfrage “**Keine Elemente übrig**” zu implementieren, merken wir uns die Anzahl der abgearbeiteten Elemente und vergleichen diese am Ende mit der ursprünglichen Anzahl Elemente

Erinnerung: die benötigten Operationen (n mal)



- Herausfinden, ob es ein Element ohne Vorgängern gibt. (Und dann eines davon nehmen)
- Ein gegebenes Element von der Menge der Elemente entfernen
- Alle Bedingungen, die mit einem gegebenen Element beginnen, aus der Menge der Bedingungen entfernen
- Herausfinden, ob noch ein Element vorhanden ist

Zyklen detektieren



sortiere
do

from create {...} *sequenz.make* **until**

“Jedes Element von *elemente* hat einen Vorgänger”

loop

next := “Ein Element von *elemente* ohne Vorgängern”

sequenz.extend (*next*)

”Lösche *next* aus *elemente* ”

”Lösche alle Paare [*next*, *y*] aus *bedingungen* ”

end

if “Keine Elemente übrig” **then**

”Berichte, dass das topologische Sortieren abgeschlossen ist”

else

”Melde Zyklen in den übrigen *bedingungen* und *elemente*”

end

end

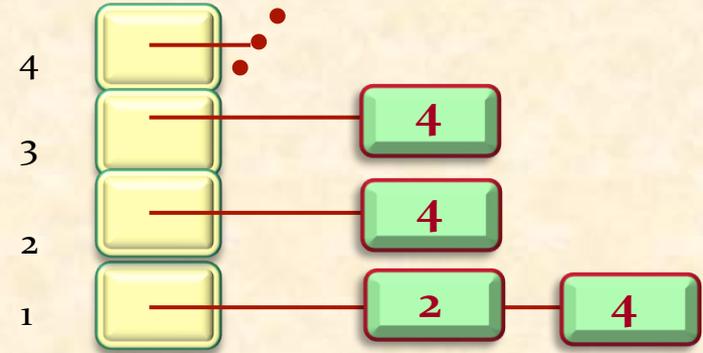


Um die Abfrage “Keine Elemente übrig” zu implementieren, merken wir uns die Anzahl der abgearbeiteten Elemente und vergleichen diese am Ende mit der ursprünglichen Anzahl Elemente

Datenstrukturen: Zusammenfassung

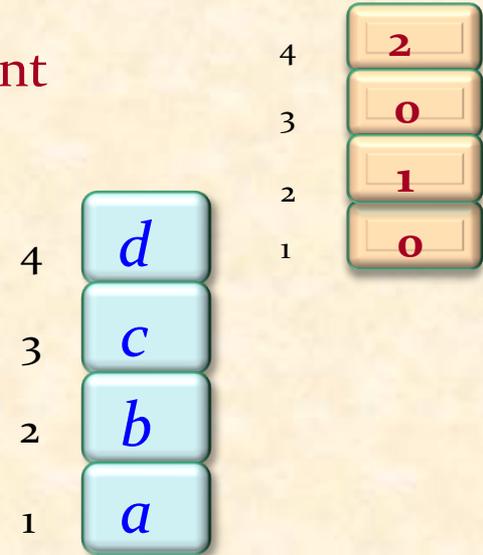
elemente : ARRAY [G]

- Elemente, Ordnung abhängig
- von Bedingungen
- (Ersetzt die ursprüngliche Liste)



nachfolger : ARRAY [LINKED_LIST [INTEGER]]

- Elemente, die nach einem bestimmten Element
- vorkommen müssen

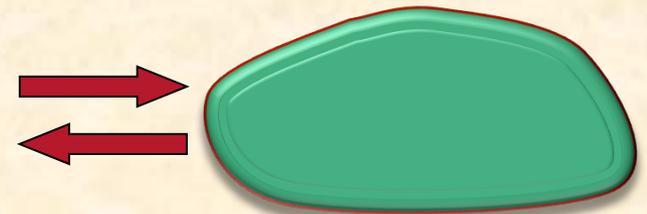


vorgänger_zahl : ARRAY [INTEGER]

- Anzahl Elemente, die vor einem
- bestimmten Element vorkommen müssen

kandidaten : STACK [INTEGER]

- Elemente ohne Vorgängern





Alle Elemente und alle Bedingungen müssen abgearbeitet werden, um diese Datenstrukturen zu erzeugen.

Dies ist $O(m + n)$.

Dies gilt auch für den restlichen Algorithmus.

Datenstrukturen 1: ursprünglich



$elemente = \{a, b, c, d\}$

$bedingungen = \{[a, b], [a, d], [b, d], [c, d]\}$

elemente



n Elemente

bedingungen



m Bedingungen

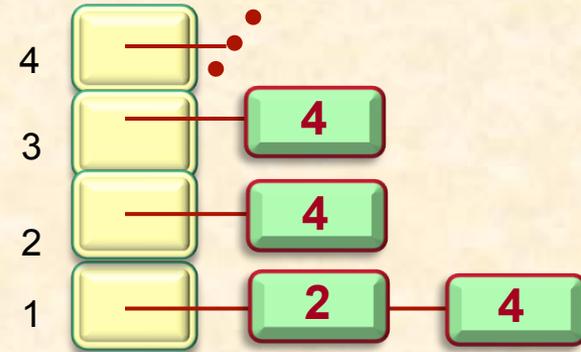
Effizienz: Das Beste, das wir uns erhoffen können:

Wenn wir *elemente* und *bedingungen* wie gegeben verwenden, können wir $O(m + n)$ nicht erreichen!



elemente : *ARRAY* [*G*]

- Elemente, Ordnung abhängig
- von Bedingungen
- (Ersetzt die ursprüngliche Liste)



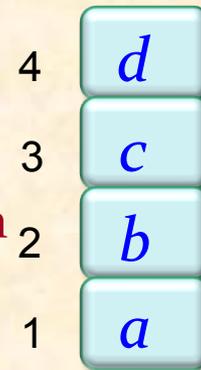
nachfolger : *ARRAY* [*LINKED_LIST* [*INTEGER*]]

- Elemente, die nach einem bestimmten Element
- vorkommen müssen



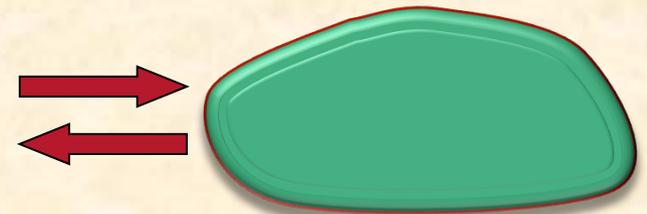
vorgänger_zahl : *ARRAY* [*INTEGER*]

- Anzahl Elemente, die vor einem
- bestimmten Element vorkommen müssen



kandidaten : *STACK* [*INTEGER*]

- Elemente ohne Vorgängern



Programmübersetzung: eine nützliche Heuristik



Die Datenstruktur ist, so wie sie gegeben ist, meist nicht die geeignetste für einen spezifischen Algorithmus

Um einen effizienten Algorithmus zu erhalten, müssen wir sie in eine speziell geeignete Form bringen

Wir können dies “übersetzen” der Daten nennen

Oft ist diese „Übersetzung“ (Initialisierung) genauso teuer wie das wirkliche Abarbeiten, manchmal sogar noch teurer. Aber dies stellt kein Problem dar, falls es die Gesamtkosten reduziert

Eine weitere Lektion



Es kann durchaus OK sein, Informationen in unseren Datenstrukturen zu duplizieren

Ursprüngliche Datenstrukturen: ohne duplizierten Information
elemente



n Elemente

bedingungen



m Bedingungen

Datenstrukturen: mit duplizierter Information

elemente : ARRAY [G]

- Elemente, Ordnung abhängig von Bedingungen
- (Ersetzt die ursprüngliche Liste)

nachfolger : ARRAY [LINKED_LIST [INTEGER]]

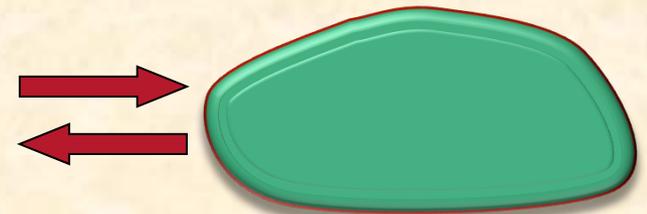
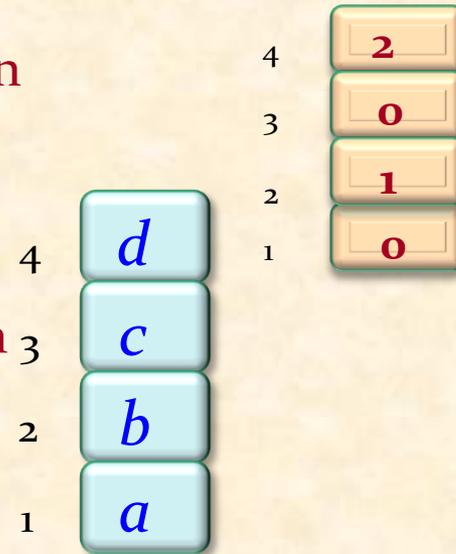
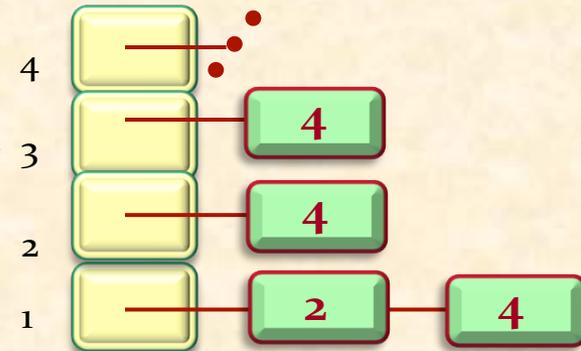
- Elemente, die nach einem
- bestimmten Element vorkommen müssen

vorgänger_zahl : ARRAY [INTEGER]

- Anzahl Elemente, die vor einem
- bestimmten Element vorkommen müssen

kandidaten : STACK [INTEGER]

- Elemente ohne Vorgängern





- Ein sehr interessanter Algorithmus, der für verschiedenste Anwendungen nützlich ist
- Mathematische Basis: Binäre Relationen
- Transitiv Hülle, reflexive transitive Hülle
- Algorithmus: Datenstrukturen anpassen ist der Schlüssel zum Erfolg
- “Übersetzungs”strategie
- Initialisierung kann genauso teuer sein wie das eigentliche Abarbeiten!
- Der Algorithmus ist nicht genug: Wir brauchen ein API! (praktisch, erweiterbar, wiederverwertbar)
- Dies ist der Unterschied zwischen Algorithmen und Software-Engineering!



Gute Algorithmen sind nicht genug

Wir müssen eine Lösung mit einer klaren Schnittstelle (API) zur Verfügung stellen, die einfach zu benutzen ist

Muster (Patterns) in Komponenten überführen