

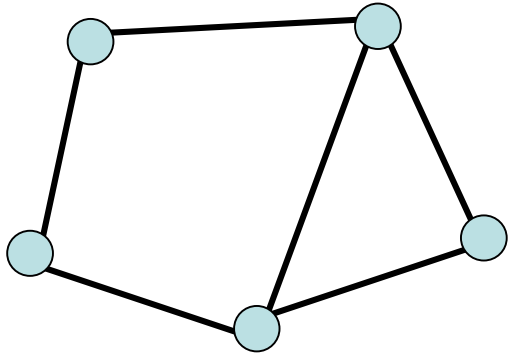
Proseminar  
Effiziente Algorithmen  
Kapitel 7: Graphdurchlauf

Prof. Dr. Christian Scheideler  
WS 2017

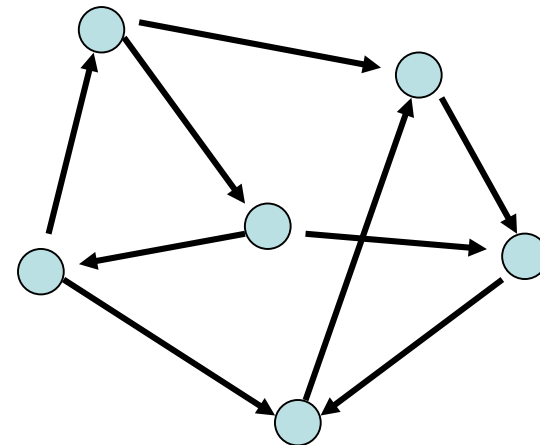
# Graphen

Graph  $G=(V,E)$  besteht aus

- Knotenmenge  $V$  
- Kantenmenge  $E$  



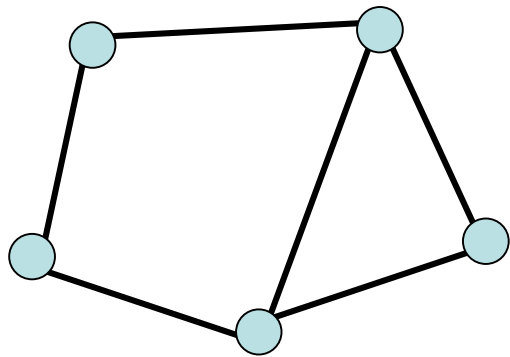
ungerichteter Graph



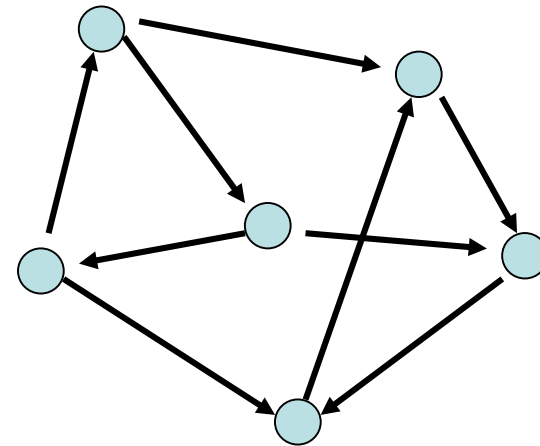
gerichteter Graph

# Graphen

- **Ungerichteter Graph:** Kante repräsentiert durch Teilmenge  $\{v,w\} \subseteq V$
- **Gerichteter Graph:** Kante repräsentiert durch Paar  $(v,w) \in V \times V$  (bedeutet  $v \rightarrow w$ )



ungerichteter Graph



gerichteter Graph

# Graphen

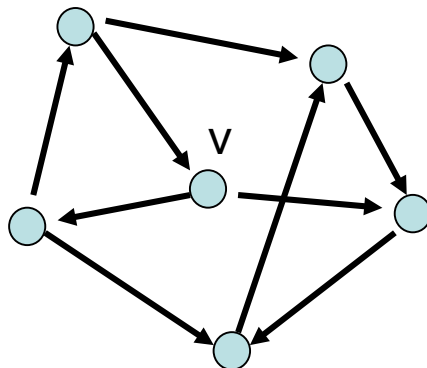
## Anwendungen:

- **Ungerichtete Graphen:** Symmetrische Beziehungen jeglicher Art (z.B.  $\{v,w\} \in E$  genau dann, wenn Distanz zwischen  $v$  und  $w$  maximal 1 km)
- **Gerichtete Graphen:** asymmetrische Beziehungen (z.B.  $(v,w) \in E$  genau dann, wenn Person  $v$  Person  $w$  liebt)

# Graphen

## Definition 7.1:

- a) Ein gerichteter Graph  $G$  ist ein Tupel  $G=(V,E)$  mit Knotenmenge  $V$  und Kantenmenge  $E \subseteq V \times V$
- b) Sei  $G=(V,E)$  ein gerichteter Graph, sei  $v \in V$ :
- $N(v) = \{w \in V \mid (v,w) \in E\}$ : Menge der Nachfolger von  $v$  in  $G$
  - $N^-(v) = \{u \in V \mid (u,v) \in E\}$ : Menge der Vorgänger von  $v$  in  $G$
  - $\text{deg}_{\text{out}}(v) = |N(v)|$ : ausgehender Grad von  $v$
  - $\text{deg}_{\text{in}}(v) = |N^-(v)|$ : eingehender Grad von  $v$

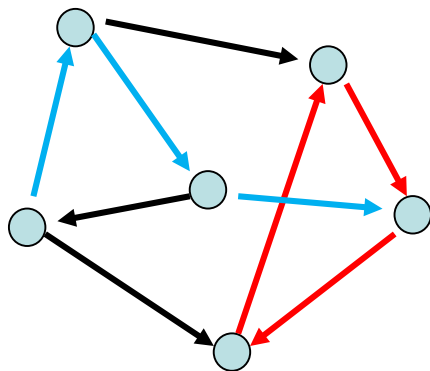


$$\text{deg}_{\text{in}}(v) = 1$$
$$\text{deg}_{\text{out}}(v) = 2$$

# Graphen

## Definition 7.2:

- a) Sei  $G=(V,E)$  ein gerichteter Graph.
- Eine Folge  $P=(v_1, \dots, v_k)$  mit  $(v_i, v_{i+1}) \in E$  für alle  $i \in [k-1]$  ist ein **Pfad** in  $G$ .
  - Ein **Kreis**  $C$  ist ein Pfad  $P=(v_1, \dots, v_k)$  mit  $v_1=v_k$ .
  - Ein Pfad (Kreis) ist **einfach**, falls  $v_i \neq v_j$  für alle  $i \neq j \in [k]$  (bzw.  $i \neq j \in [k-1]$ ).
  - Ein Pfad (oder Kreis) kann auch durch eine Folge von Kanten angegeben werden.
- b) Die **Länge** eines Pfades  $P$  (Kreises  $C$ ) ist definiert als die Anzahl der Kanten in  $P$  (bzw.  $C$ ).

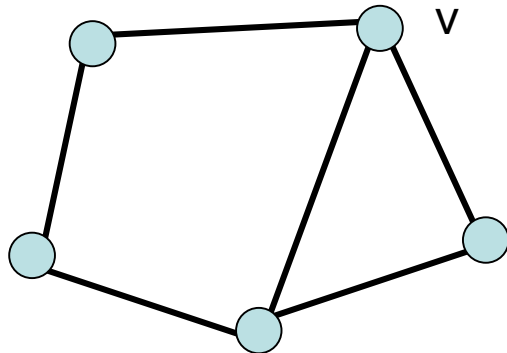


- einfacher Pfad
- einfacher Kreis

# Graphen

## Definition 7.3:

- a) Ein ungerichteter Graph  $G$  ist ein Tupel  $G=(V,E)$  mit Knotenmenge  $V$  und Kantenmenge  $E \subseteq \wp^2(V)$
- b) Sei  $G=(V,E)$  ein ungerichteter Graph, sei  $v \in V$ :
- $N(v)=\{w \in V \mid \{v,w\} \in E\}$ : Menge der Nachbarn von  $v$  in  $G$
  - $\deg(v)=|N(v)|$ : Grad von  $v$
  - Eine Folge  $P=(v_1, \dots, v_k)$  mit  $\{v_i, v_{i+1}\} \in E$  für alle  $i \in [k-1]$  ist ein Pfad in  $G$ .
  - Ein Kreis  $C$  ist ein Pfad  $P=(v_1, \dots, v_k)$  mit  $v_1=v_k$ .



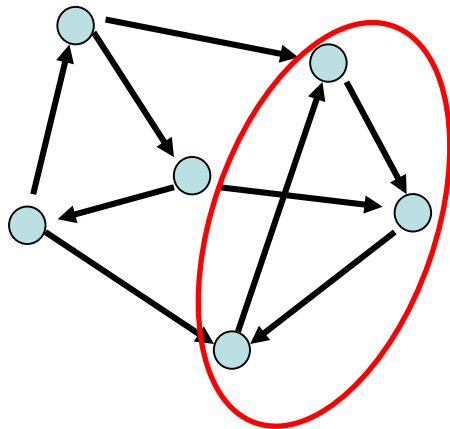
$$\deg(v) = 3$$

# Graphen

## Definition 7.4:

Sei  $G=(V,E)$  ein ungerichteter (bzw. gerichteter) Graph.

- $n=|V|$ : Anzahl der Knoten
- $m=|E|$ : Anzahl der Kanten
- $v \rightsquigarrow_G w$ : es gibt einen Pfad von  $v$  nach  $w$  in  $G$
- Eine Menge  $U \subseteq V$  heißt (starke) Zusammenhangskomponente von  $G$ , falls für alle Knoten  $v, w \in U$  gilt, dass  $v \rightsquigarrow_G w$ , und es keine Übermenge von  $U$  mit dieser Eigenschaft gibt, das heißt,  $U$  maximal mit dieser Eigenschaft ist.
- $G$  heißt (stark) zusammenhängend, falls  $V$  eine (starke) Zusammenhangskomponente in  $G$  ist.



starke Zusammenhangskomponente

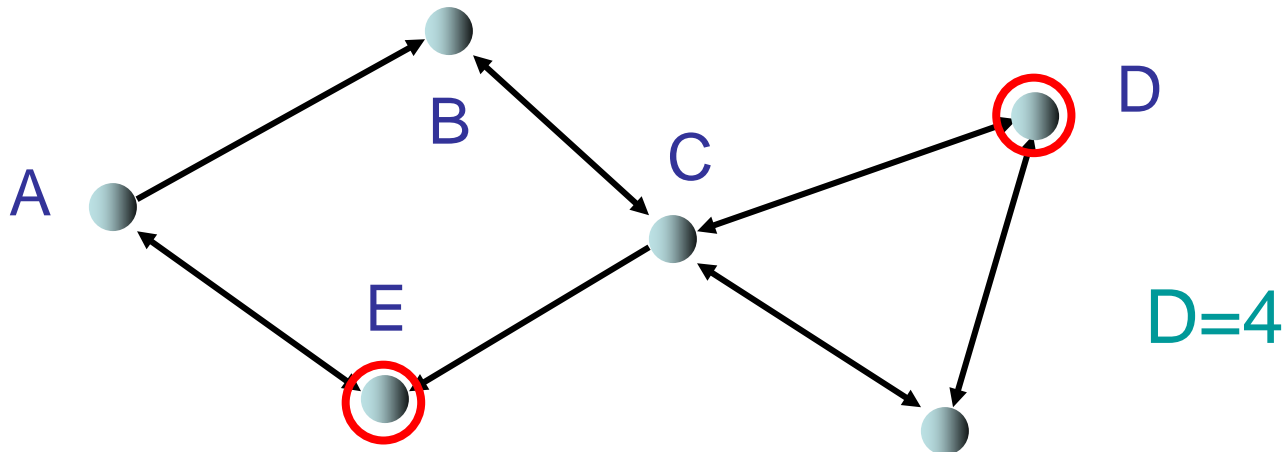


# Graphen

## Definition 7.5:

Sei  $G=(V,E)$  ein ungerichteter Graph.

- Distanz  $\delta(v,w)$ : Länge eines kürzesten Weges von  $v$  nach  $w$  in  $G$
- $D=\max_{v,w} \delta(v,w)$ : Durchmesser von  $G$
- $\alpha(G) = \min_{U \subseteq V, |U| \leq \lceil |V|/2 \rceil} |N(U)|/|U|$ : Expansion von  $G$

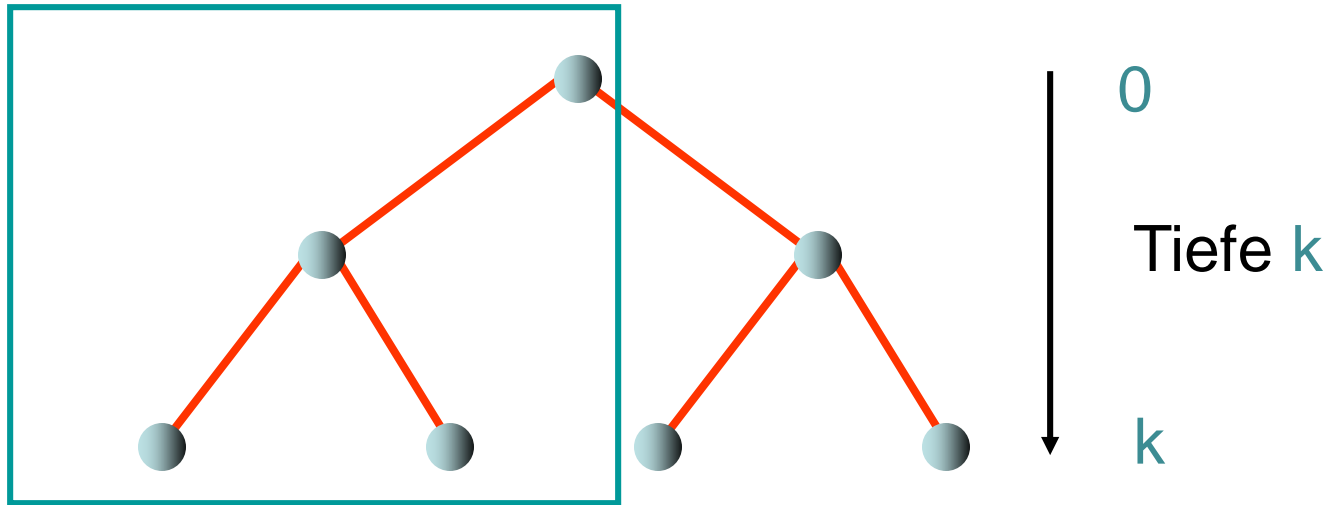


# Lineare Liste



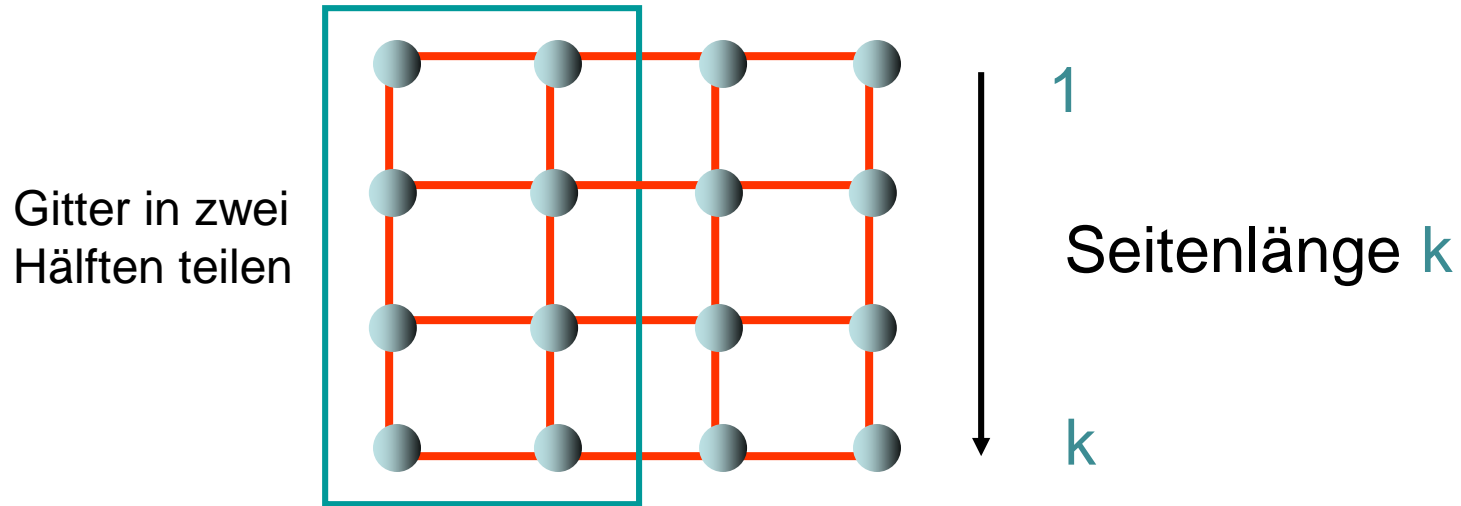
- Grad 2
- Hoher Durchmesser ( $n-1$  für  $n$  Knoten)
- Niedrige Expansion (  $\alpha(\text{Liste}) = 2/n$  )

# Vollständiger binärer Baum



- $n=2^{k+1}-1$  Knoten, Grad 3
- Durchmesser ist  $2k \sim 2 \log_2 n$
- Niedrige Expansion (  $\alpha(\text{Binärbaum})=2/n$  )

# 2-dimensionales Gitter



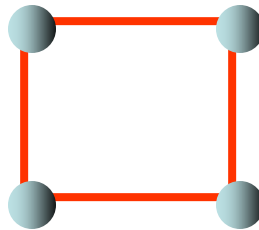
- $n = k^2$  Knoten, maximaler Grad 4
- Durchmesser ist  $2(k-1) \sim 2\sqrt{n}$
- Expansion ist  $\sim 2/\sqrt{n}$

# Hypercube

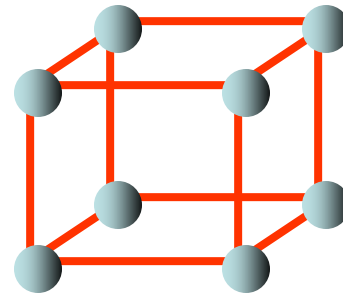
- Knoten:  $(x_1, \dots, x_d) \in \{0, 1\}^d$
- Kanten:  $\forall i: (x_1, \dots, x_d) \rightarrow (x_1, \dots, 1-x_i, \dots, x_d)$



$d=1$



$d=2$



$d=3$

Grad  $d$ , Durchmesser  $d$ , Expansion  $1/\sqrt{d}$

# Expander

Satz 7.6: Für jeden Graph  $G$  ist die Expansion  $\alpha(G)$  höchstens 1.

Satz 7.7: Es gibt Familien von Graphen **konstanten** Grades mit **konstanter** Expansion. Diese heißen **Expander**.

Beispiel: Gabber-Galil Graph

- Knotenmenge:  $(x,y) \in \{0, \dots, n-1\}^2$
- $(x,y) \rightarrow (x, x+y), (x, x+y+1), (x+y, y), (x+y+1, y) \pmod n$

# Operationen auf Graphen

$G=(V,E)$ : Graph-Variable für gerichteten Graph

- **Node**: DS für Knoten, **Edge**: DS für Kanten

Operationen:

- **G.insert**( $e$ : Edge):  $E:=E \cup \{e\}$
- **G.remove**( $i,j$ : Key):  $E:=E \setminus \{e\}$  für die Kante  $e=(v,w)$  mit  $\text{Key}(v)=i$  und  $\text{Key}(w)=j$
- **G.insert**( $v$ : Node):  $V:=V \cup \{v\}$
- **G.remove**( $i$ : Key): sei  $v \in V$  der Knoten mit  $\text{Key}(v)=i$ .  
 $V:=V \setminus \{v\}$ ,  $E:=E \setminus \{(x,y) \mid x=v \vee y=v\}$
- **G.find**( $i$ : Key): gib Knoten  $v$  aus mit  $\text{Key}(v)=i$
- **G.find**( $i,j$ : Key): gib Kante  $(v,w)$  aus mit  $\text{Key}(v)=i$  und  $\text{Key}(w)=j$

# Operationen auf Graphen

Anzahl der Knoten oft **fest**. In diesem Fall:

- $V = \{1, \dots, n\}$  (Knoten hintereinander nummeriert, identifiziert durch ihre Keys)

Relevante Operationen:

- $G.insert(e: Edge): E := E \cup \{e\}$
- $G.remove(i, j: Key): E := E \setminus \{e\}$  für die Kante  $e = (i, j)$
- $G.find(i, j: Key):$  gib Kante  $e = (i, j)$  aus



# Operationen auf Graphen

Im folgenden: Konzentration auf statische Anzahl an Knoten.

Parameter für Laufzeitanalyse:

- $n$ : Anzahl Knoten
- $m$ : Anzahl Kanten
- $d$ : maximaler Knotengrad (maximale Anzahl ausgehender Kanten von Knoten)

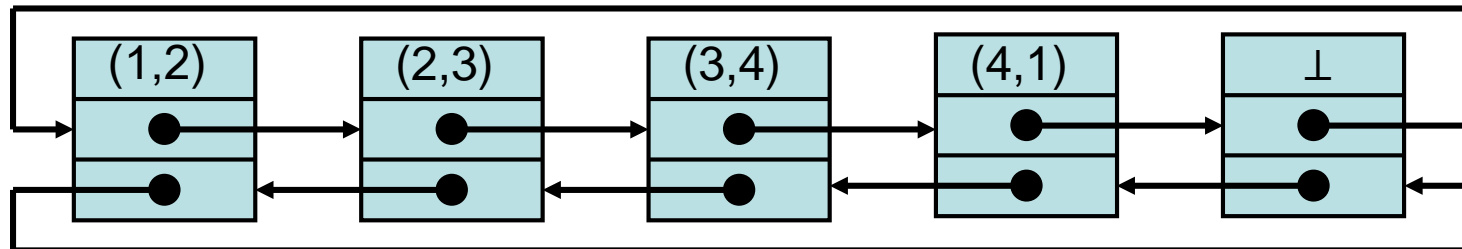
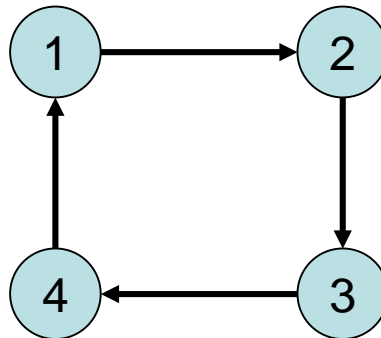
# Graphrepräsentationen

Wir betrachten folgende Repräsentationen:

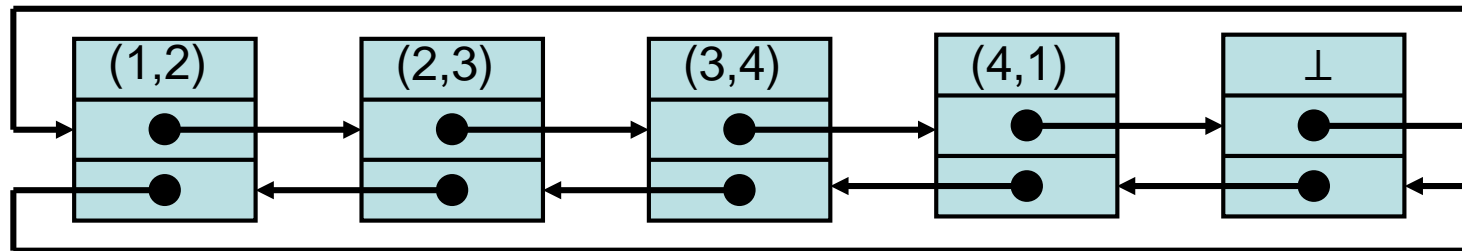
1. Sequenz von Kanten
2. Adjazenzfeld
3. Adjazenzliste
4. Adjazenzmatrix
5. Adjazenzliste + Hashtabelle

# Graphrepräsentationen

## 1: Sequenz von Kanten



# Sequenz von Kanten

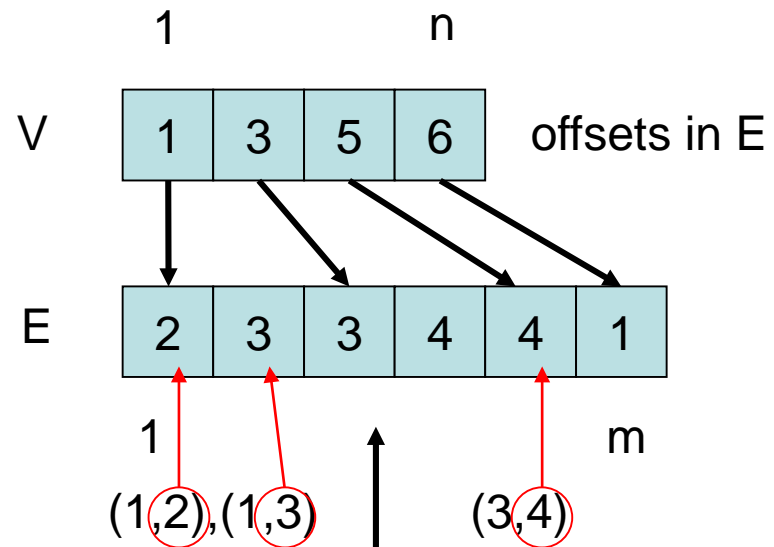
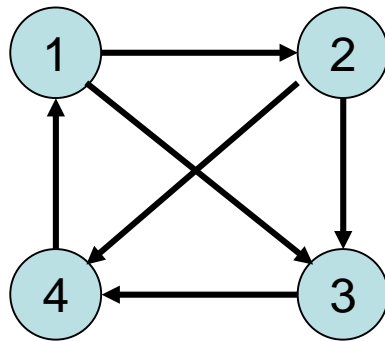


Zeitaufwand:

- **G.find**(*i,j*: Key):  $\Theta(m)$  im worst case
- **G.insert**(*e*: Edge):  $O(1)$
- **G.remove**(*i,j*: Key):  $\Theta(m)$  im worst case

# Graphrepräsentationen

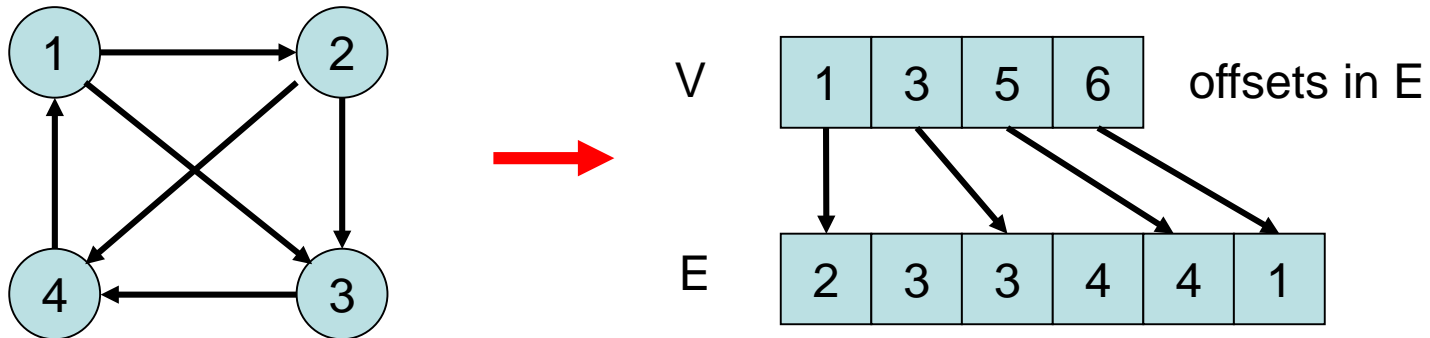
## 2: Adjazenzfeld



Hier: nur Zielkeys

In echter DS: **E**: Array [1..m] of **Edge**

# Adjazenzfeld

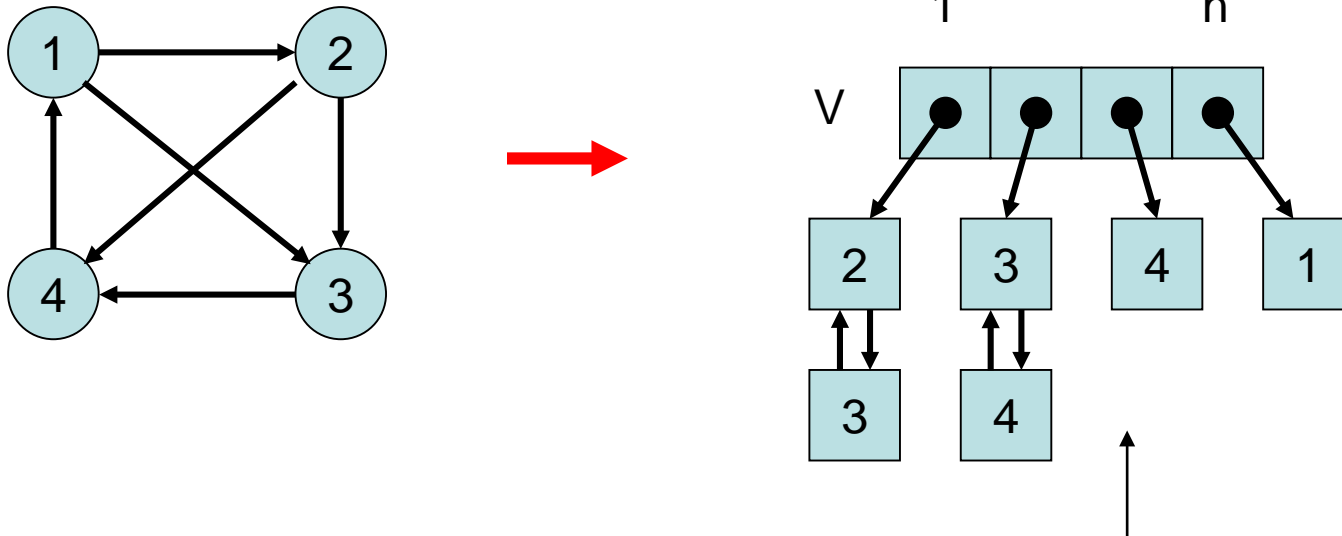


## Zeitaufwand:

- **G.find**(i,j: Key): Zeit  $O(d)$
- **G.insert**(e: Edge): Zeit  $O(m)$  (worst case)
- **G.remove**(i,j: Key): Zeit  $O(m)$  (worst case)

# Graphrepräsentationen

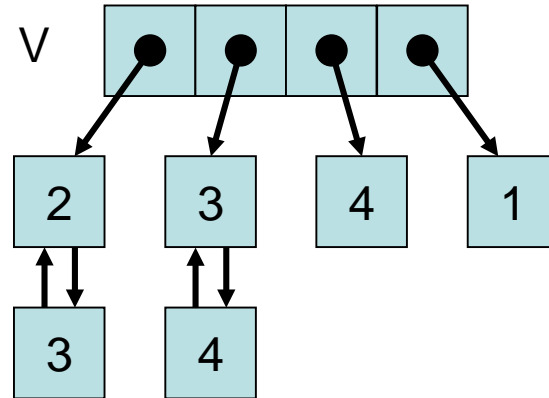
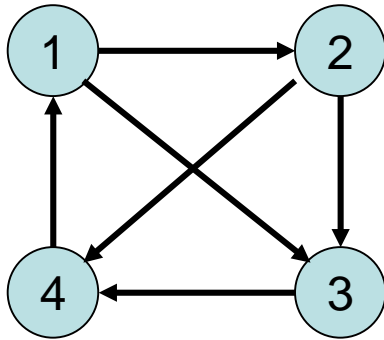
## 3: Adjazenzliste



Hier: nur Zielkeys

In echter DS:  $V$ : Array  $[1..n]$  of List of Edge

# Adjazenzliste



## Zeitaufwand:

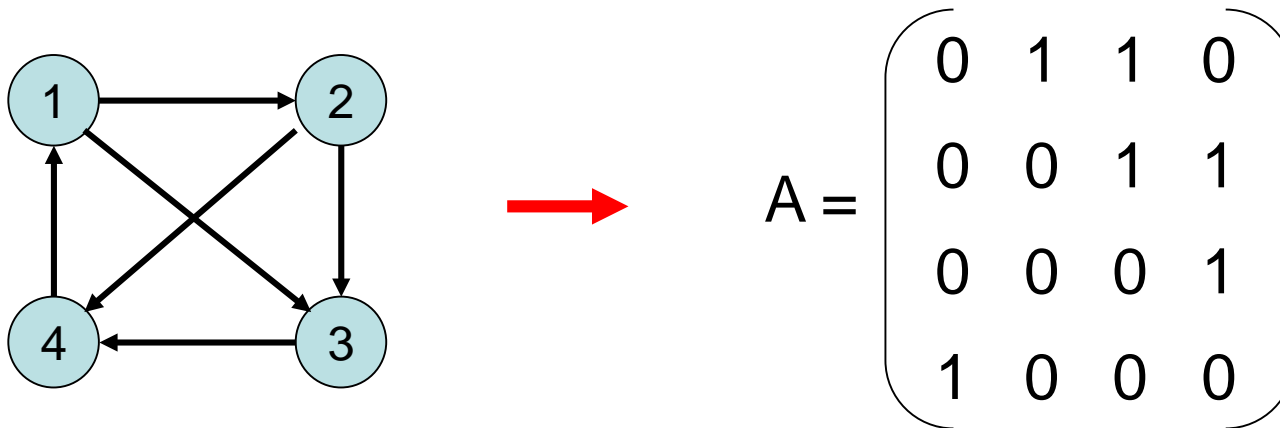
- **G.find**(i,j: Key): Zeit  $O(d)$
- **G.insert**(e: Edge): Zeit  $O(1)$
- **G.remove**(i,j: Key): Zeit  $O(d)$

Problem: d kann groß sein!



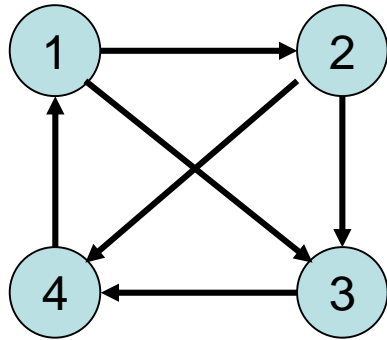
# Graphrepräsentationen

## 4: Adjazenzmatrix



- $A[i,j] \in \{0,1\}$  (bzw. Zeiger auf **Edge**)
- $A[i,j]=1$  genau dann, wenn  $(i,j) \in E$

# Adjazenzmatrix



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

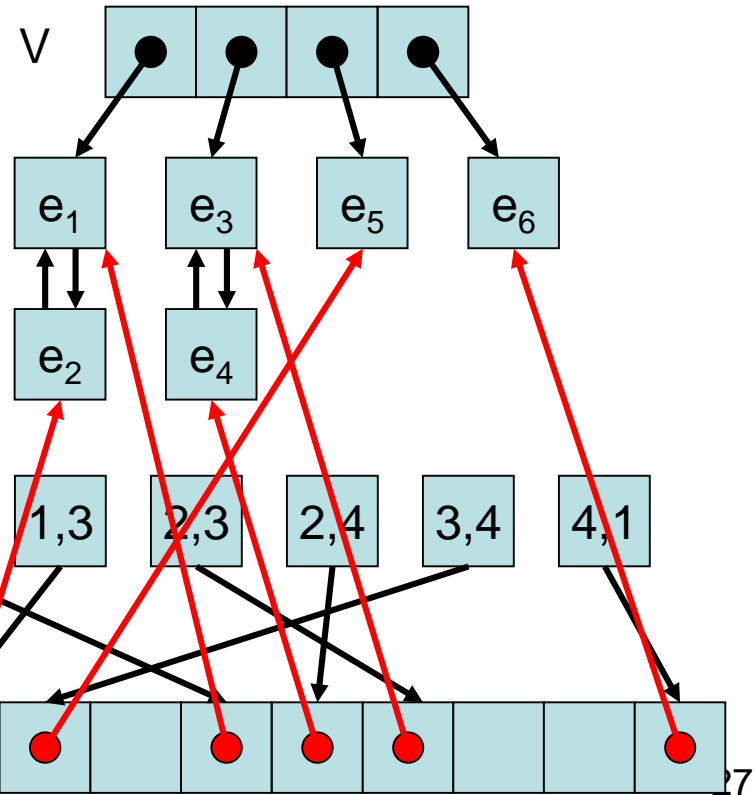
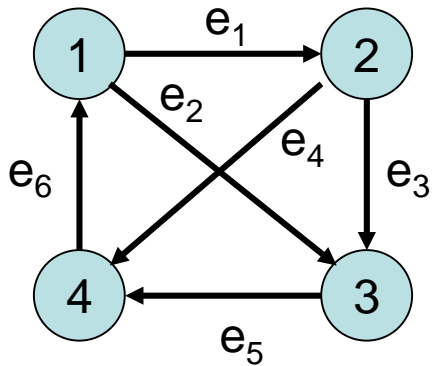
## Zeitaufwand:

- **G.find**(i,j: Key): Zeit  $O(1)$
- **G.insert**(e: Edge): Zeit  $O(1)$
- **G.remove**(i,j: Key): Zeit  $O(1)$

Aber: Speicher-  
aufwand  $O(n^2)$

# Graphrepräsentationen

## 5: Adjazenzliste + Hashtabelle

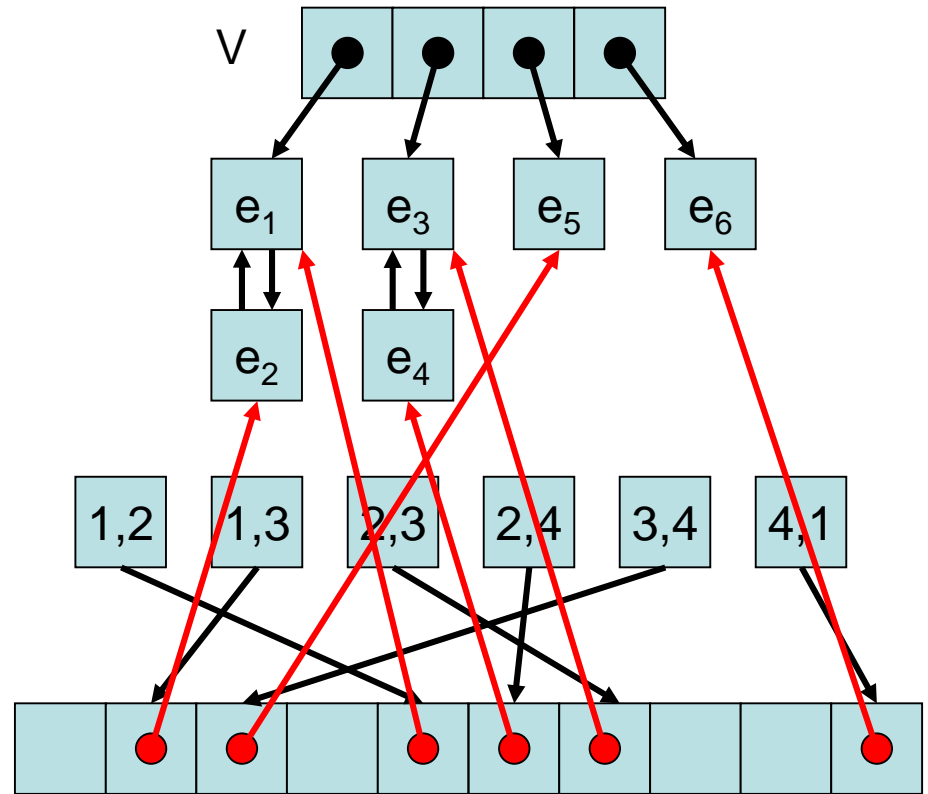


z.B. Kuckuckshashing+ →

# Adjazenzliste+Hashtabelle

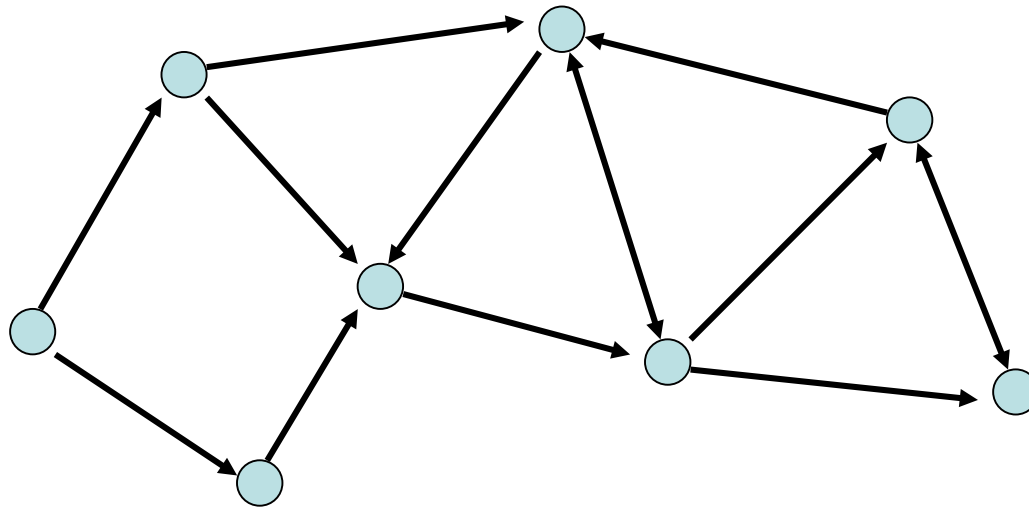
## Zeitaufwand:

- **G.find**(i,j: Key):  
O(1) (m.h.W.)
- **G.insert**(e: Edge):  
O(1) (m.h.W.)
- **G.remove**(i,j: Key):  
O(1) (m.h.W.)
- Speicher: O(n+m)



# Graphdurchlauf

**Zentrale Frage:** Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?



# Graphdurchlauf

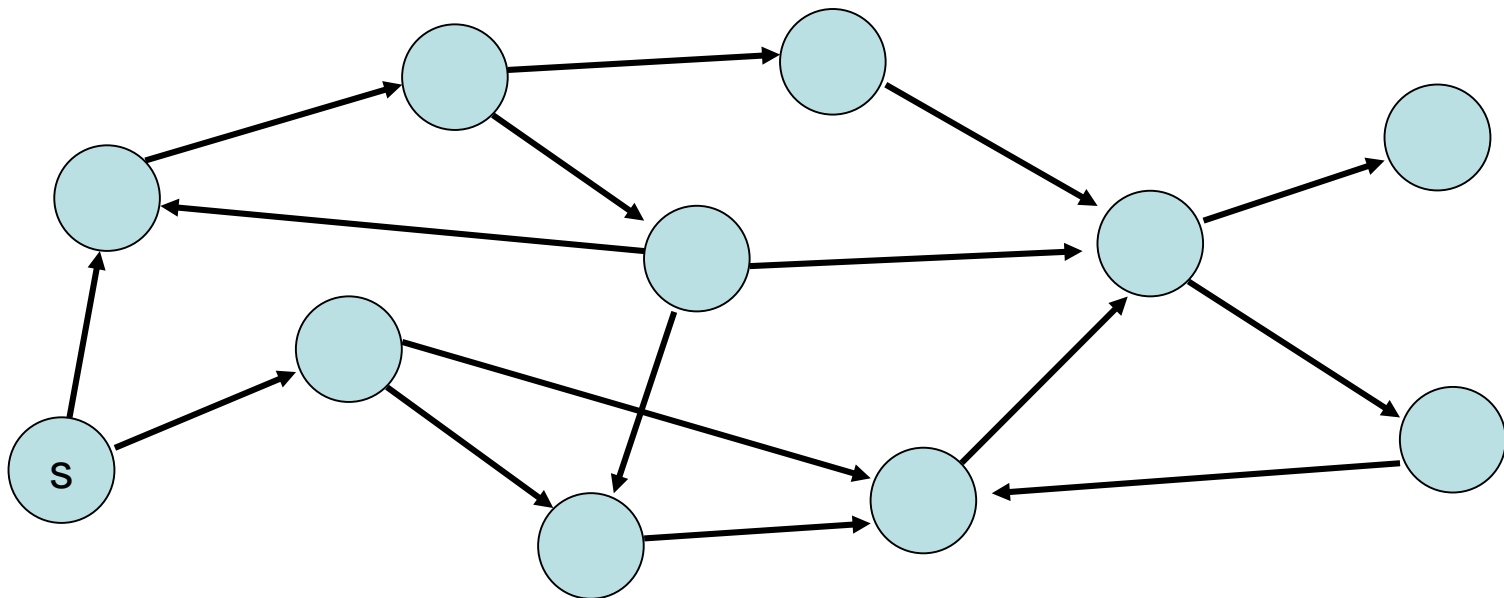
**Zentrale Frage:** Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?

**Grundlegende Strategien:**

- Breitensuche
- Tiefensuche

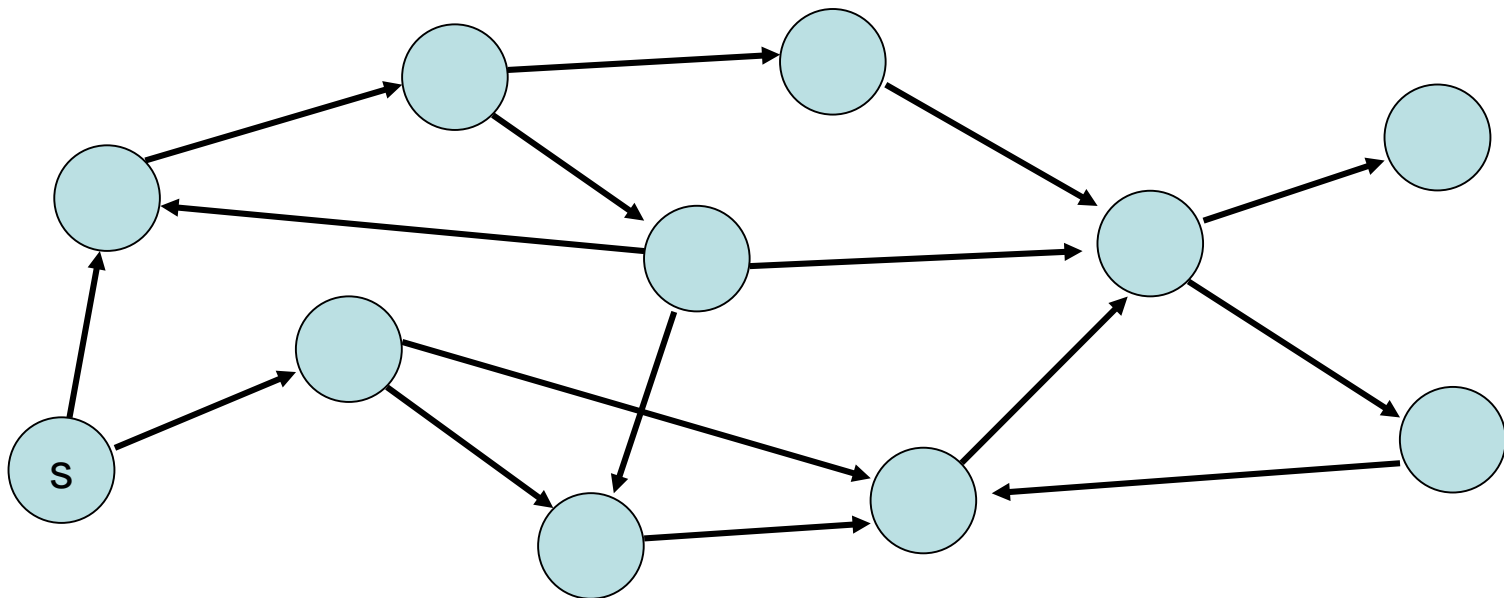
# Breitensuche

- Starte von einem Knoten **s**
- Exploriere Graph Distanz für Distanz



# Tiefensuche

- Starte von einem Knoten  $s$
- Exploriere Graph in die Tiefe  
( ● : aktuell, ● : noch aktiv, ● : fertig)

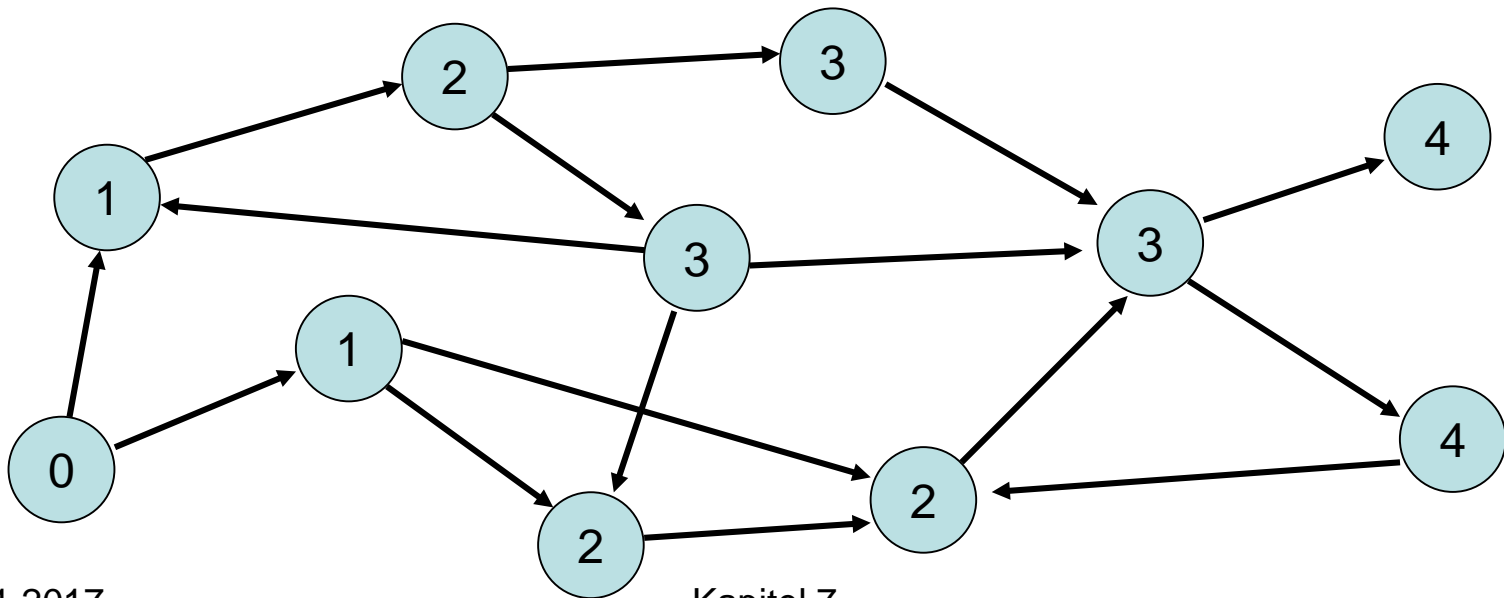




# Breitensuche

- $d(v)$ : Distanz von Knoten  $v$  zu  $s$  ( $d(s)=0$ )
- $\text{parent}(v)$ : Knoten, von dem  $v$  besucht

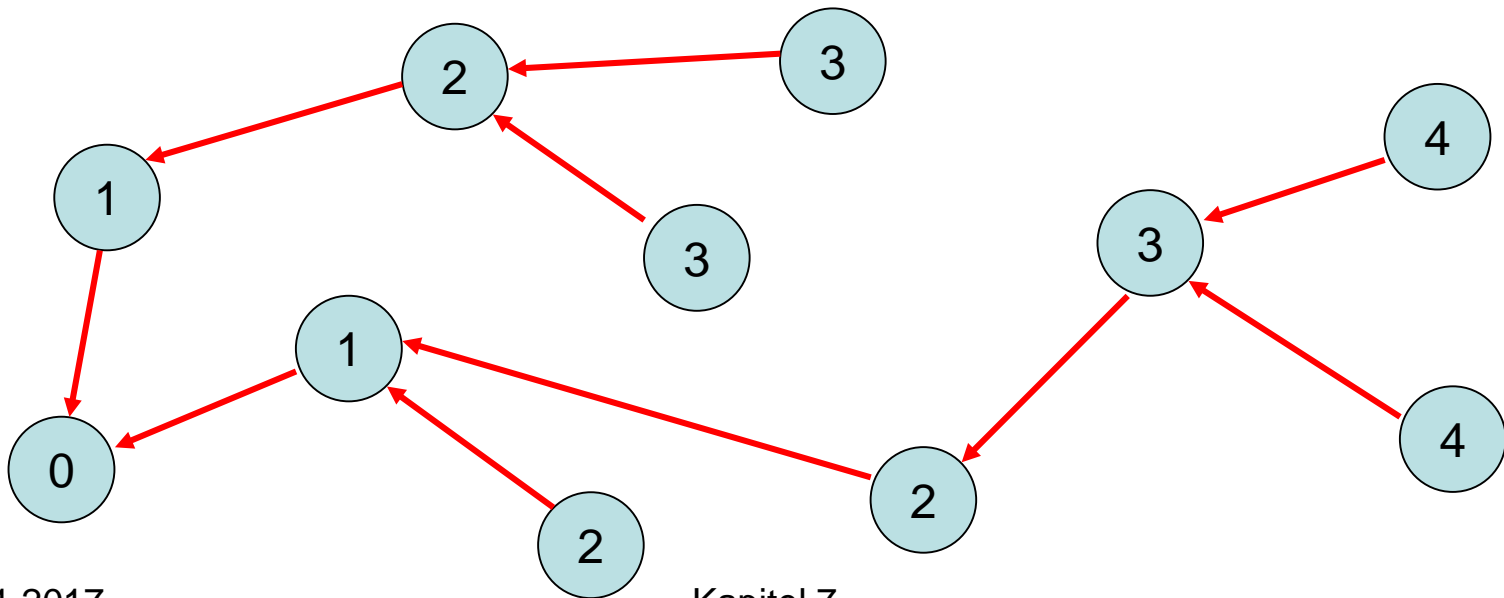
Distanzen:



# Breitensuche

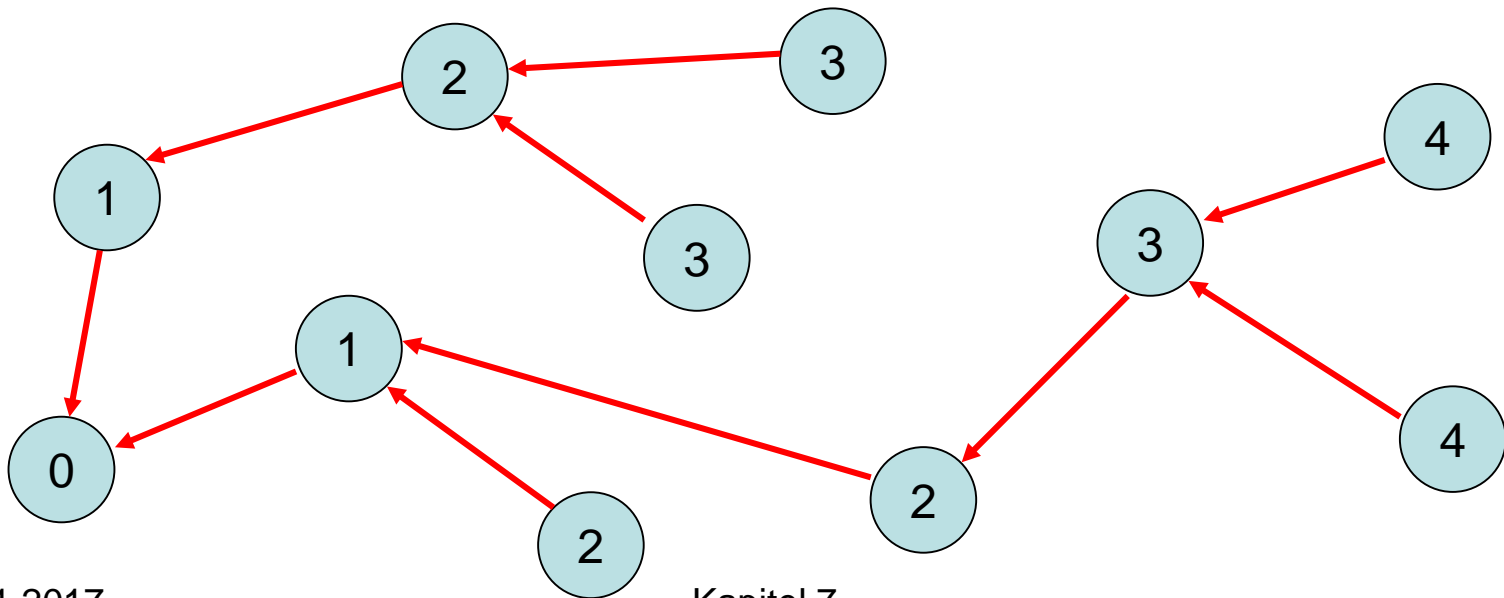
- $d(v)$ : Distanz von Knoten  $v$  zu  $s$  ( $d(s)=0$ )
- $\text{parent}(v)$ : Knoten, von dem  $v$  besucht

Mögliche Parent-Beziehungen in rot:



# Breitensuche

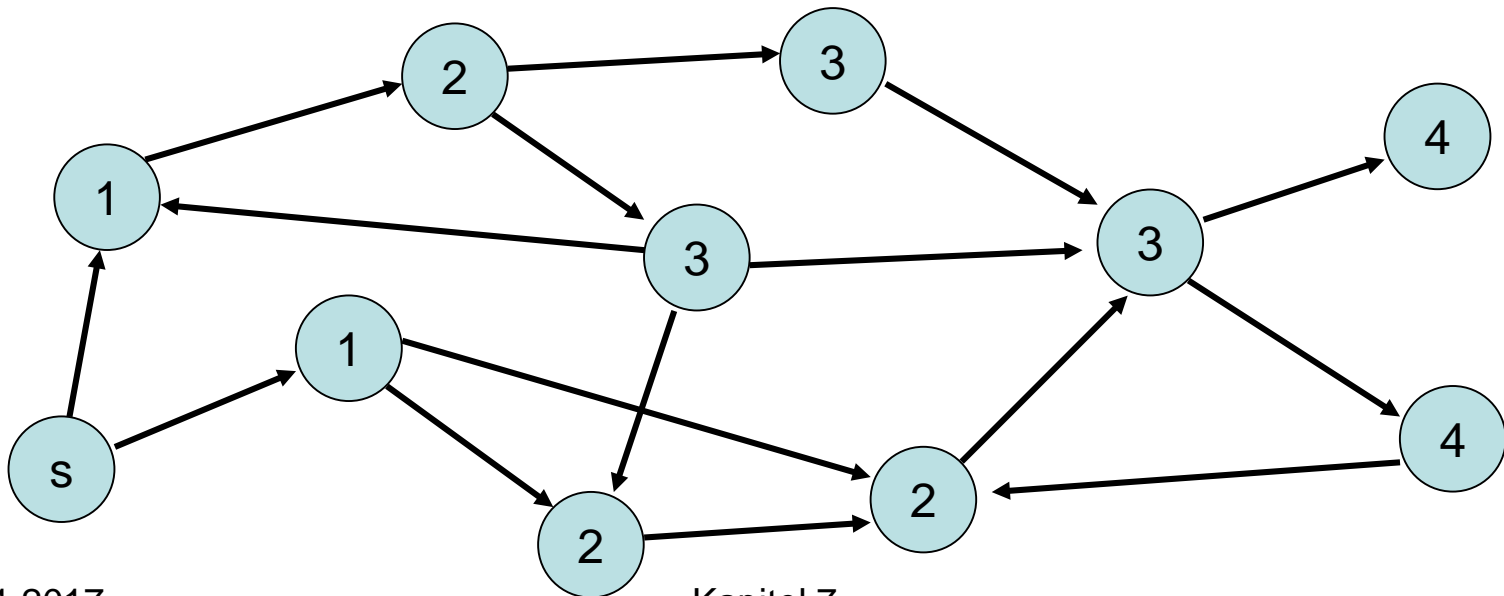
Parent-Beziehung eindeutig: wenn Knoten  $v$  zum erstenmal besucht wird, wird  $\text{parent}(v)$  gesetzt und  $v$  markiert, so dass  $v$  nicht nochmal besucht wird



# Breitensuche

Kantentypen:

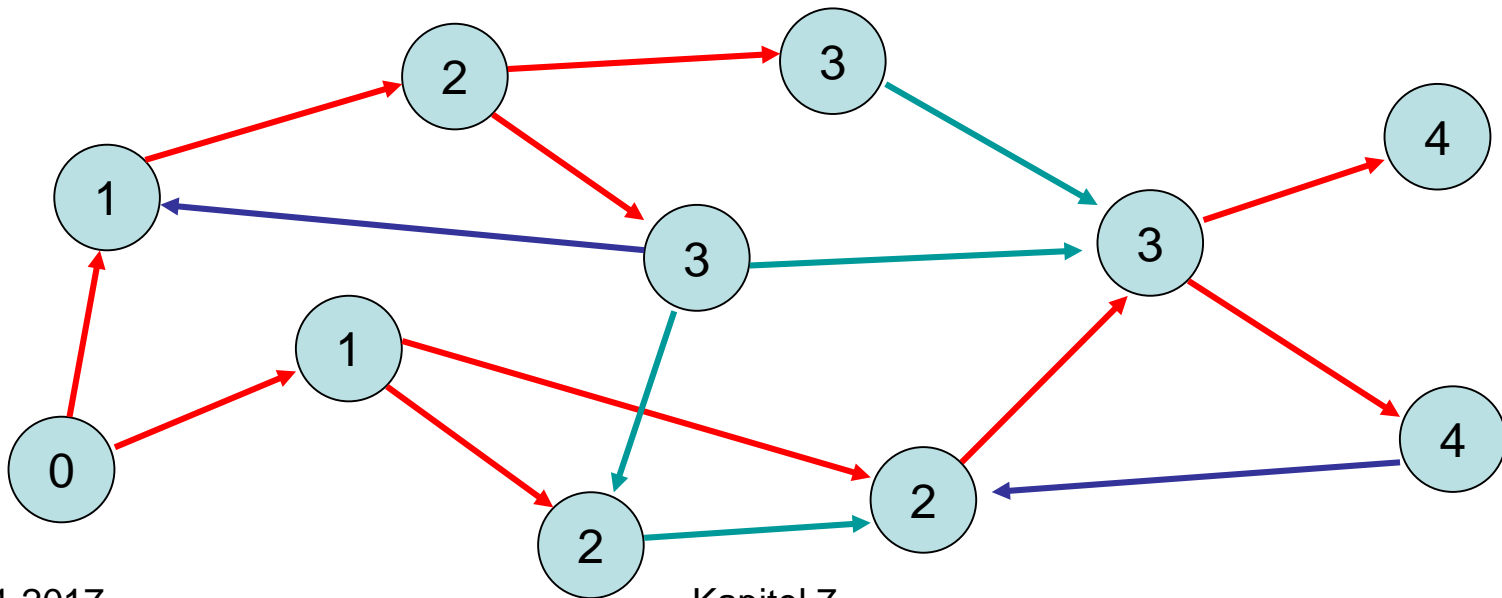
- **Baumkante:** zum Kind
- **Rückwärtskante:** zu einem Vorfahr
- **Kreuzkante:** alle sonstige Kanten



# Breitensuche

Kantentypen:

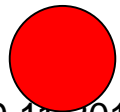
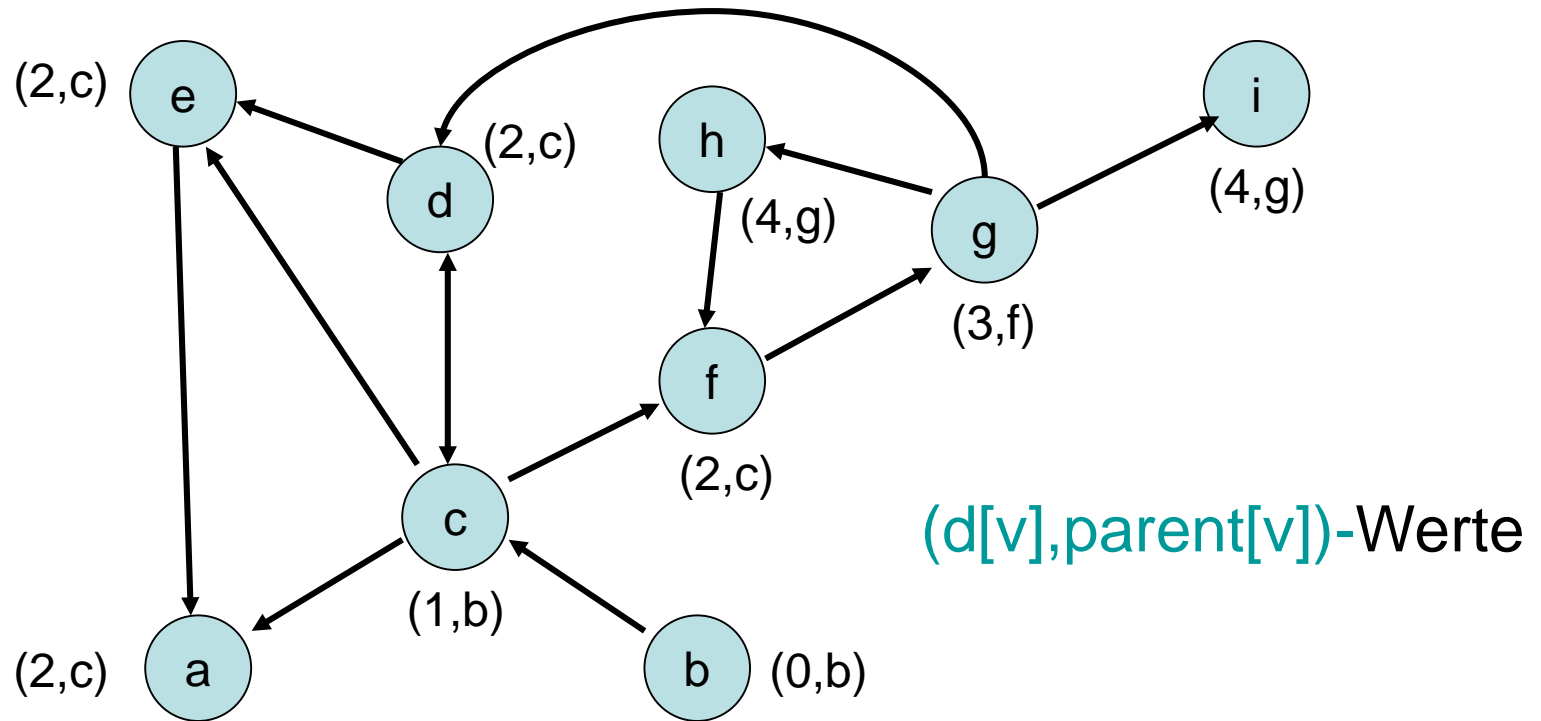
- **Baumkante:** zum Kind
- **Rückwärtskante:** zu einem Vorfahr
- **Kreuzkante:** alle sonstige Kanten



# Breitensuche

```
Procedure BFS(s: Node)
  d =  $\langle \infty, \dots, \infty \rangle$ : Array [1..n] of IN
  parent =  $\langle \perp, \dots, \perp \rangle$ : Array [1..n] of Node
  d[Key(s)]:=0           // s hat Distanz 0 zu sich
  parent[Key(s)]:=s     // s ist sein eigener Vater
  q:=<s>: List of Node // q:Queue zu besuchender Knoten
  while q  $\neq$  <> do    // solange q nicht leer
    u:= q.popFront()  // nimm Knoten nach FIFO-Regel
    foreach (u,v) $\in$ E do
      if parent(Key(v))= $\perp$  then // v schon besucht?
        q.pushBack(v) // nein, dann in q hinten einfügen
        d[Key(v)]:=d[Key(u)]+1
        parent[Key(v)]:=u
```

# BFS(b)

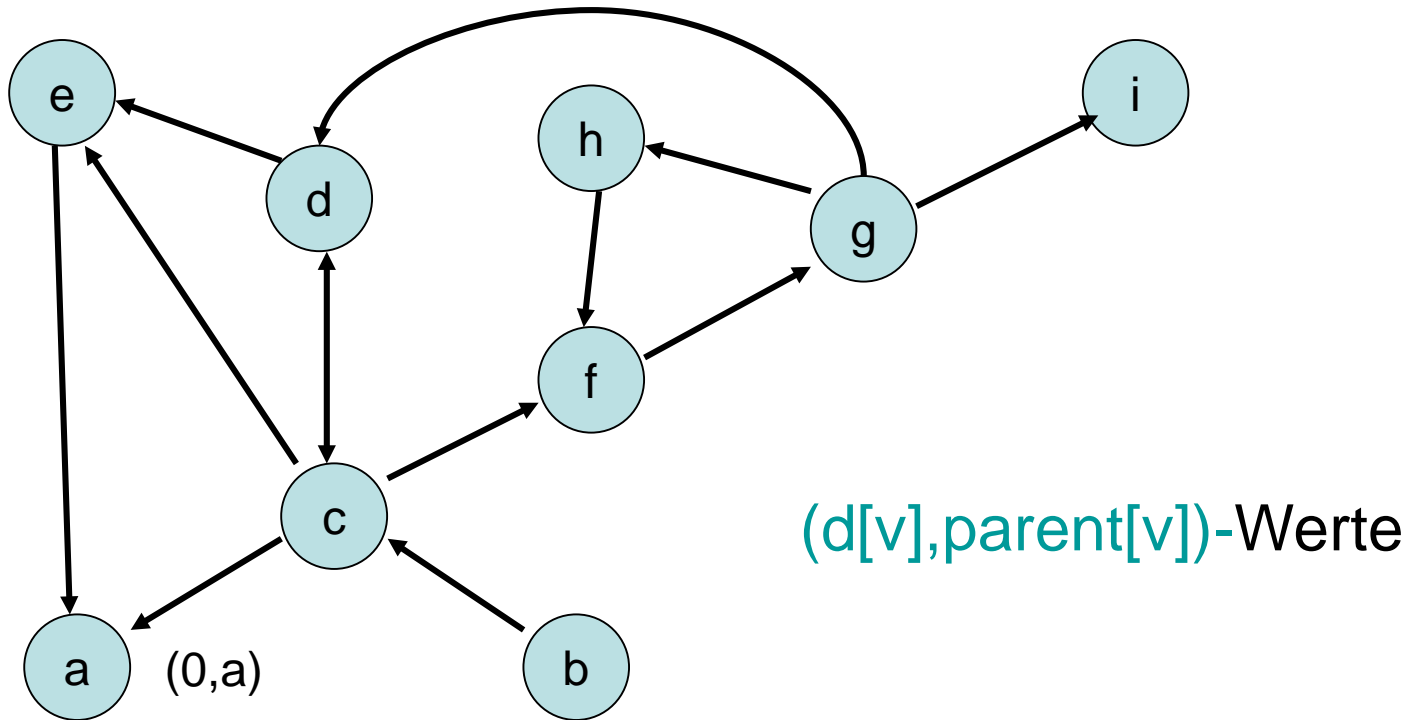


: besucht, noch in  $q$



: besucht, nicht mehr in  $q$

# BFS(a)

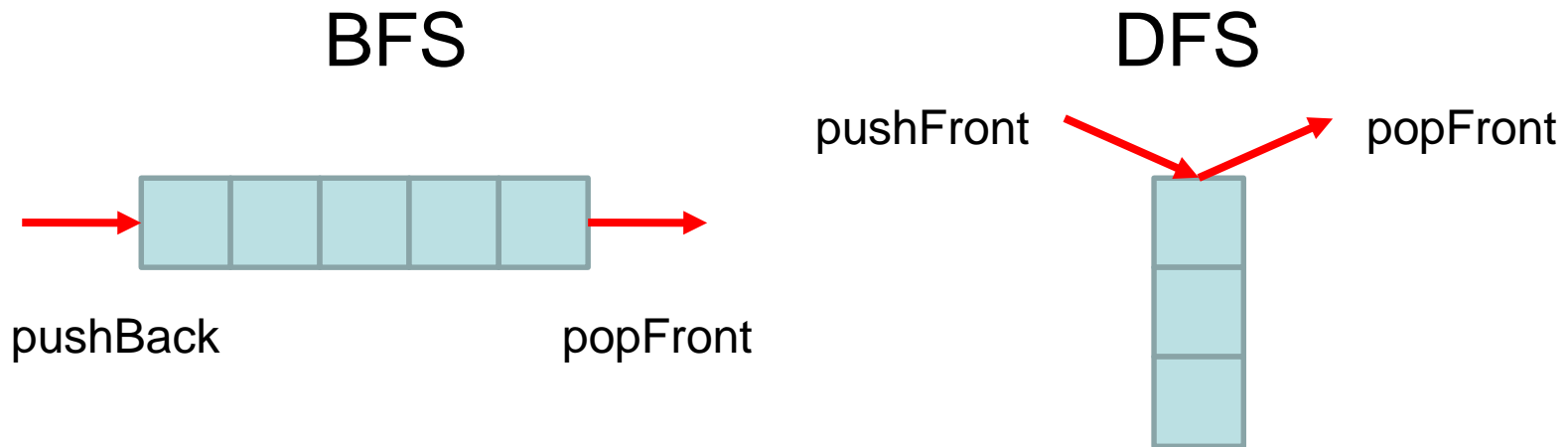


Von **a** kein anderer Knoten erreichbar.



# Tiefensuche

**Einsicht:** BFS-Algorithmus kann einfach in einen DFS-Algorithmus umgewandelt werden, wenn anstelle einer FIFO Queue für die Knotenbehandlung ein Stack verwendet wird.

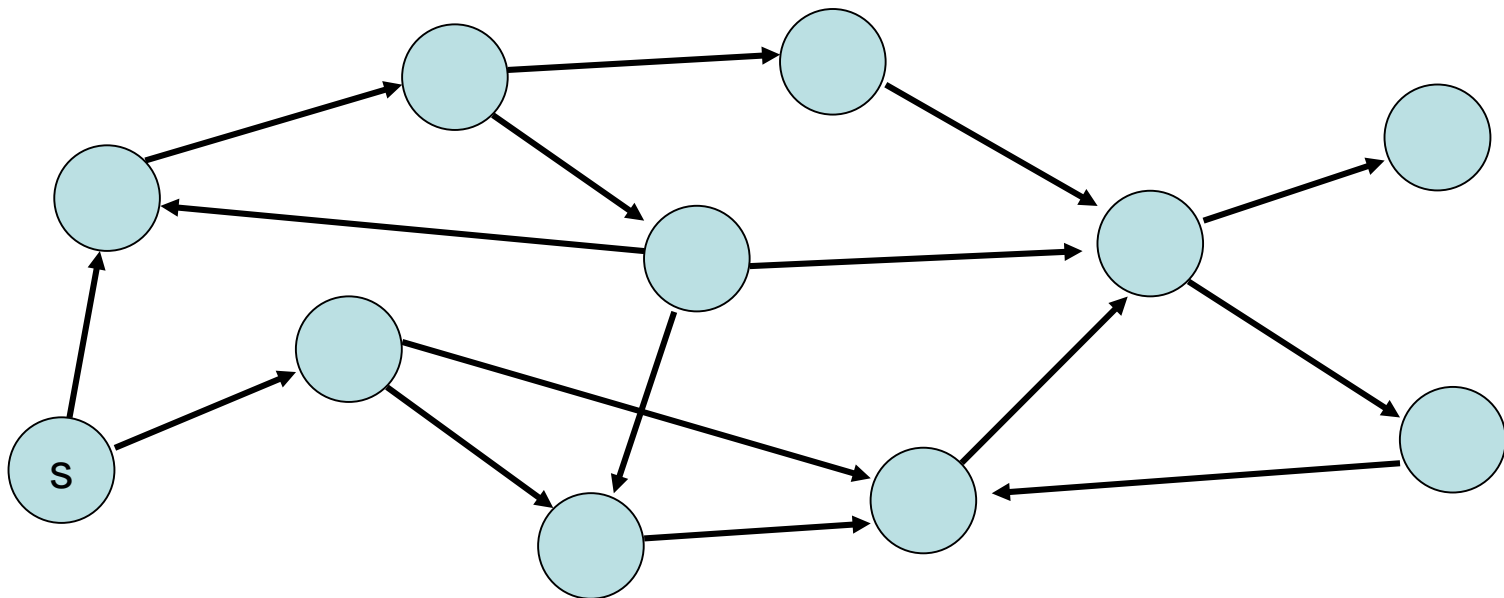


# Tiefensuche

```
Procedure DFS(s: Node)
  d = <∞, ..., ∞>: Array [1..n] of IN
  parent = <⊥, ..., ⊥>: Array [1..n] of Node
  d[Key(s)]:=0           // s hat Distanz 0 zu sich
  parent[Key(s)]:=s     // s ist sein eigener Vater
  q:=<s>: List of Node // q:Stack zu besuchender Knoten
  while q ≠ <> do      // solange q nicht leer
    u:= q.popFront()  // nimm Knoten nach FIFO-Regel
    foreach (u,v)∈E do
      if parent[Key(v)]≠⊥ then // v schon besucht?
        q.pushFront(v) // nein, dann in q vorne einfügen
        d[Key(v)]:=d[Key(u)]+1
        parent[Key(v)]:=u
```

# Tiefensuche

- Starte von einem Knoten **s**
- Exploriere Graph in die Tiefe  
( ● : aktuell, ● : im Stack, ● : fertig)



# Tiefensuche – Alternatives Schema

Übergeordnete Prozedur:

unmark all nodes

init()

foreach  $s \in V$  do // stelle sicher, dass alle Knoten besucht werden

if  $s$  is not marked then

mark  $s$

root( $s$ )

DFS( $s,s$ ) //  $s$ : Startknoten

Procedure DFS( $u,v$ : Node) //  $u$ : Vater von  $v$

foreach  $(v,w) \in E$  do

if  $w$  is marked then traverseNonTreeEdge( $v,w$ )

else traverseTreeEdge( $v,w$ )

mark  $w$

DFS( $v,w$ )

backtrack( $u,v$ )

Prozeduren in rot: noch zu spezifizieren

# DFS-Nummerierung

## Variablen:

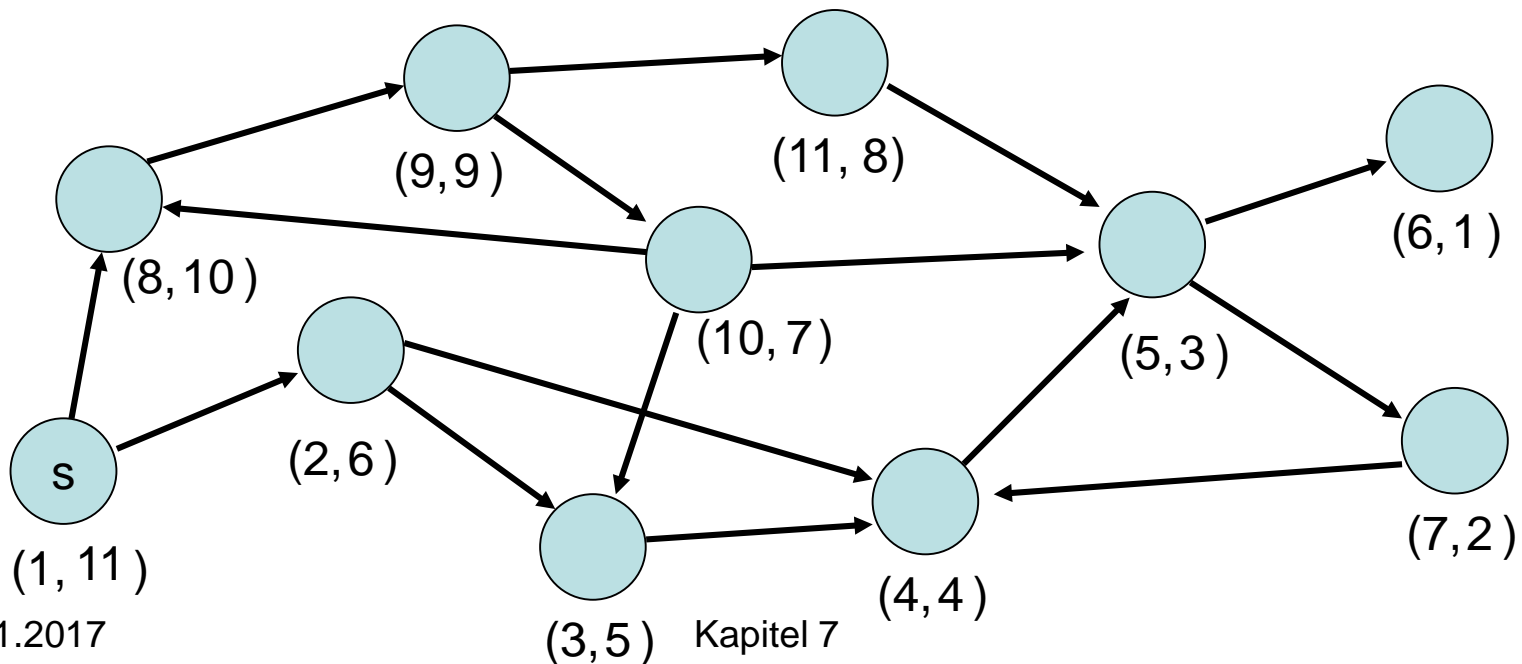
- `dfsNum`: Array [1..n] of IN // Zeitpunkt wenn Knoten 
- `finishTime`: Array [1..n] of IN // Zeitpunkt wenn Knoten  → 
- `dfsPos`, `finishingTime`: IN // Zähler

## Prozeduren:

- `init()`:  
`dfsPos:=1; finishingTime:=1`
- `root(s)`:  
`dfsNum[s]:=dfsPos; dfsPos:=dfsPos+1`
- `traverseTreeEdge(v,w)`:  
`dfsNum[w]:=dfsPos; dfsPos:=dfsPos+1`
- `traverseNonTreeEdge(v,w)`:  
-
- `Backtrack(u,v)`:  
`finishTime[v]:=finishingTime; finishingTime:=finishingTime+1`

# DFS-Nummerierung


- Exploriere Graph in die Tiefe  
( ● : aktuell, ● : noch aktiv, ● : fertig)
- Paare  $(i,j)$ :  $i$ : dfsNum,  $j$ : finishTime



# DFS-Nummerierung

Ordnung  $<$  auf den Knoten:

$$u < v, \text{dfsNum}[u] < \text{dfsNum}[v]$$

**Lemma 7.8:** Die Knoten im DFS-Rekursionsstack (alle  Knoten) sind sortiert bezüglich  $<$ .

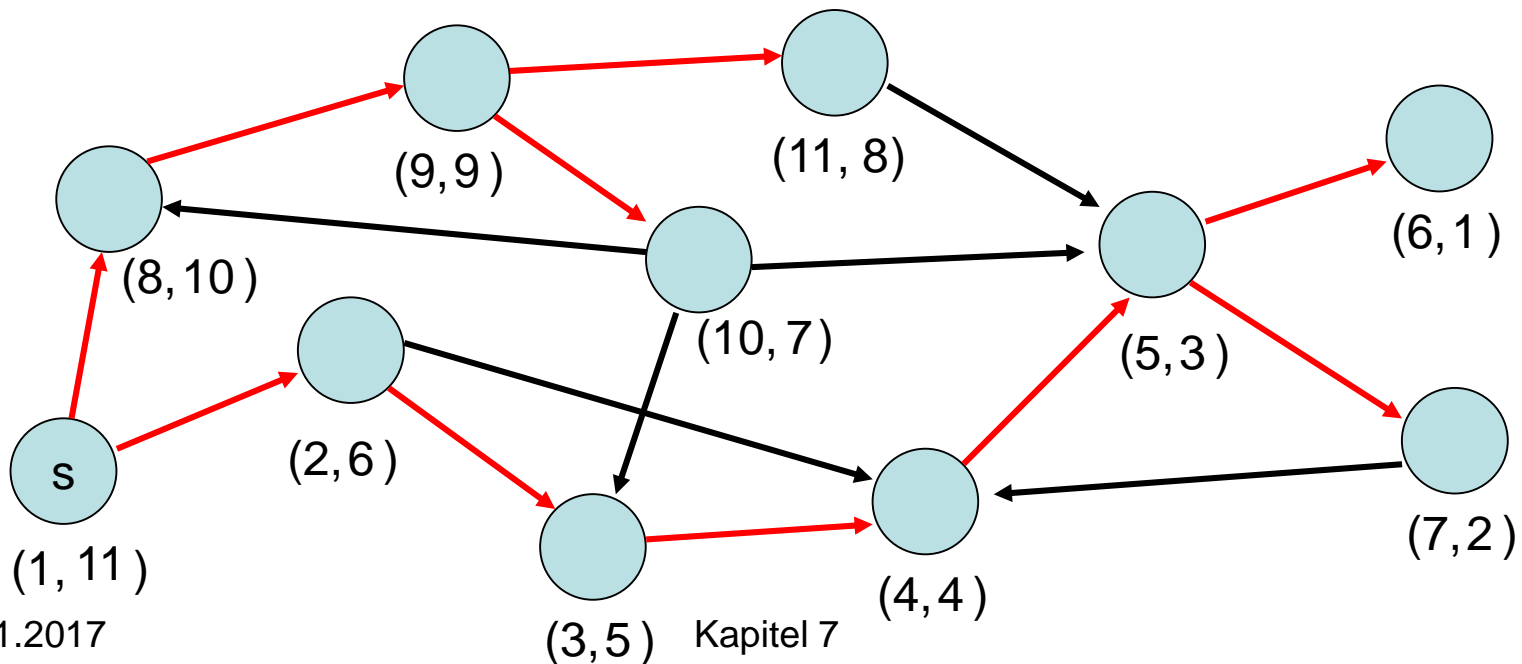
**Beweis:**

dfsPos wird nach jeder Zuweisung von dfsNum erhöht. Jeder neue aktive Knoten hat also immer die höchste dfsNum.

# DFS-Nummerierung

## Überprüfung von Lemma 7.8:

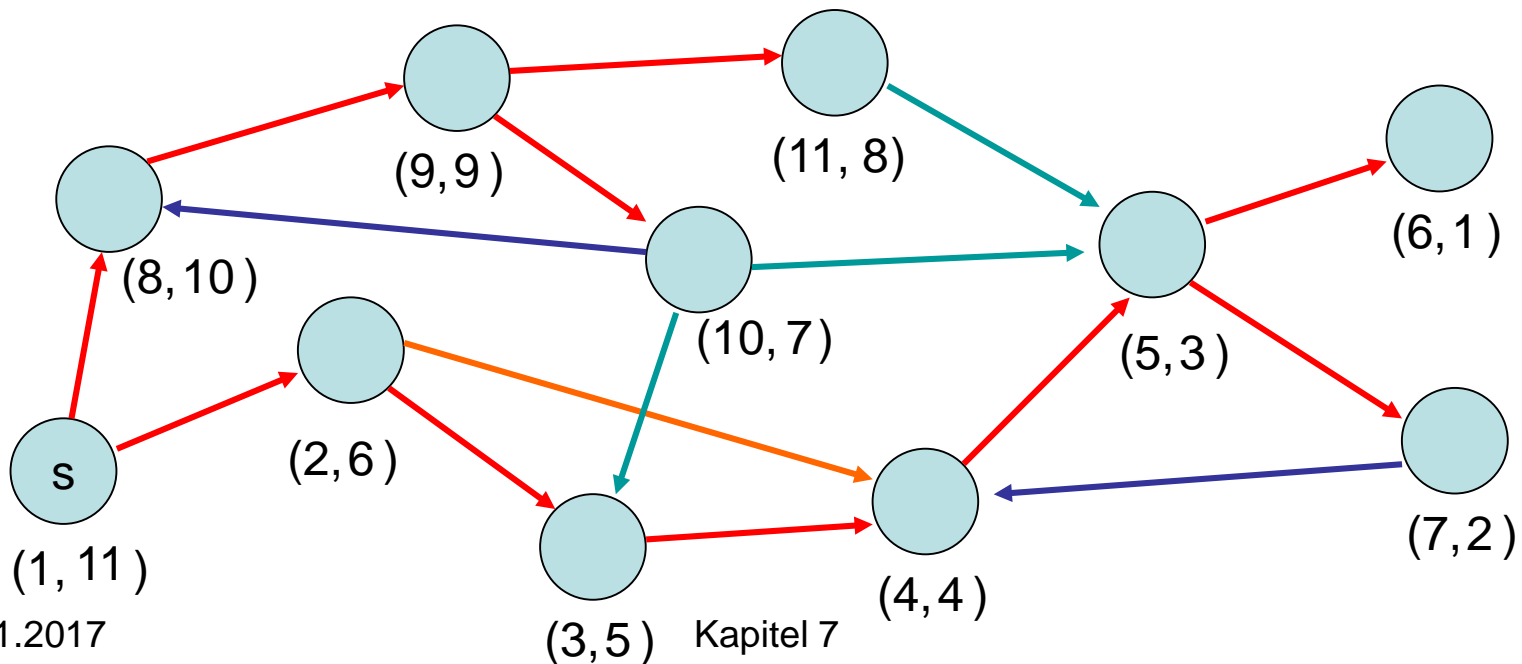
- Rekursionsstack: roter Pfad von  $s$
- Paare  $(i,j)$ :  $i$ : dfsNum,  $j$ : finishTime





# DFS-Nummerierung

- **Baumkante:** zum Kind
- **Vorwärtskante:** zu einem Nachkommen
- **Rückwärtskante:** zu einem Vorfahr
- **Kreuzkante:** alle sonstige Kanten



# DFS-Nummerierung

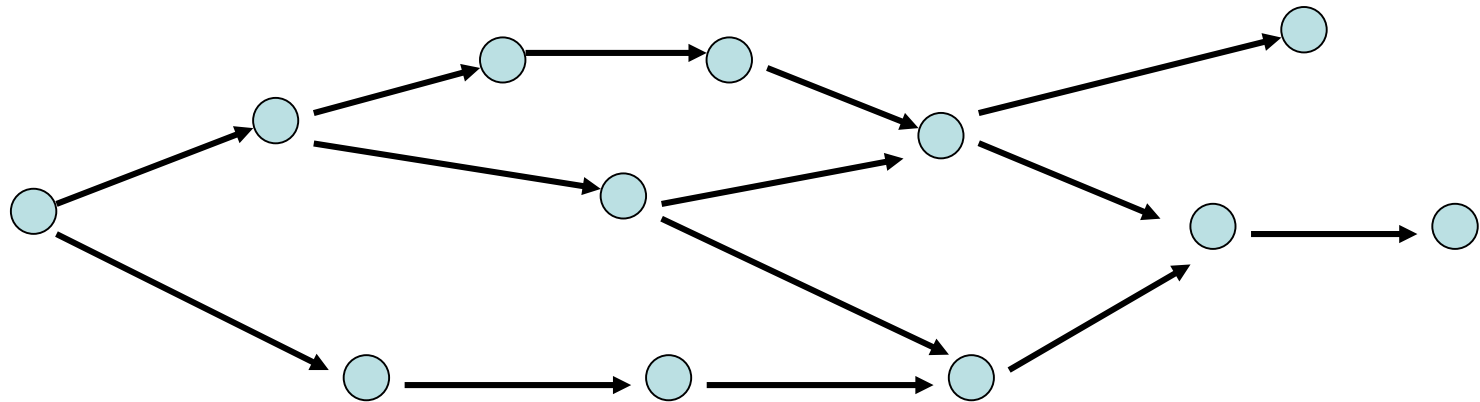
Beobachtung für Kante  $(v,w)$ :

Kantentyp	$\text{dfsNum}[v] < \text{dfsNum}[w]$	$\text{finishTime}[v] > \text{finishTime}[w]$
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

# DFS-Nummerierung

## Anwendung:

- Erkennung eines azyklischen gerichteten Graphen (engl. DAG)



Merkmale: keine gerichtete Kreise

# DFS-Nummerierung

Lemma 7.9: Das Folgende ist äquivalent:

1.  $G$  ist ein DAG
2. DFS enthält keine Rückwärtskante
3.  $\forall (v,w) \in E: \text{finishTime}[v] > \text{finishTime}[w]$

Beweis:

2. , 3.: folgt aus Tabelle

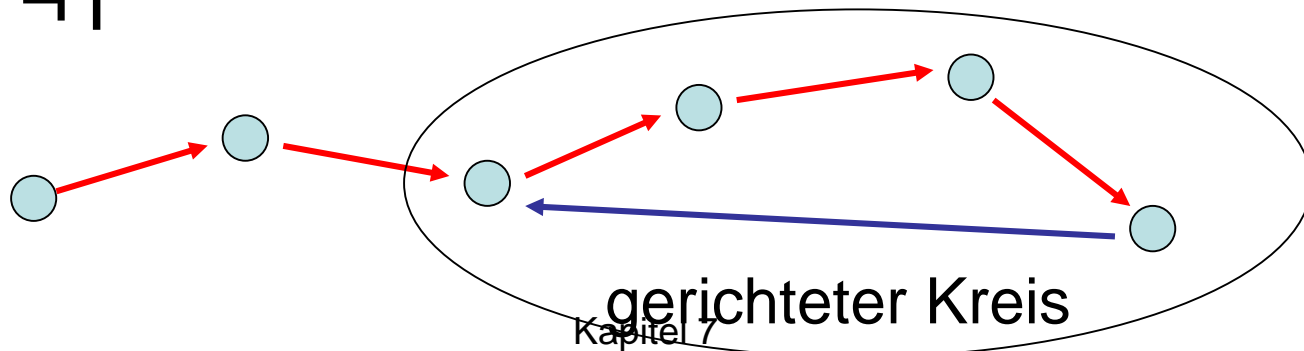
# DFS-Nummerierung

Lemma 7.9: Das Folgende ist äquivalent:

1.  $G$  ist ein DAG
2. DFS enthält keine Rückwärtskante
3.  $\forall (v,w) \in E: \text{finishTime}[v] > \text{finishTime}[w]$

Beweis:

$\neg 2 \Rightarrow \neg 1$



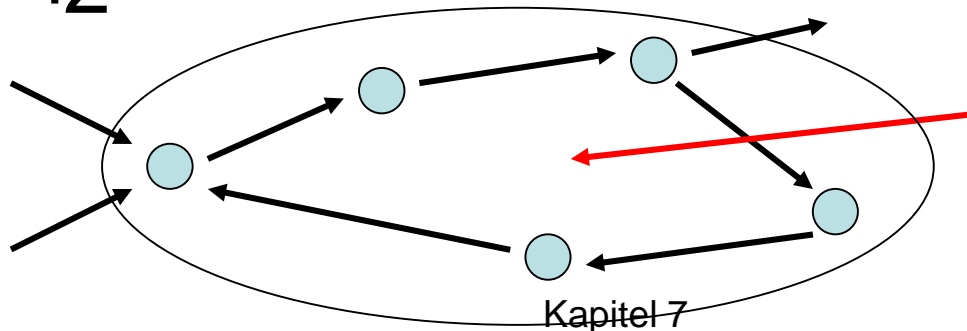
# DFS-Nummerierung

Lemma 7.9: Das Folgende ist äquivalent:

1.  $G$  ist ein DAG
2. DFS enthält keine Rückwärtskante
3.  $\forall (v,w) \in E: \text{finishTime}[v] > \text{finishTime}[w]$

Beweis:

$\neg 1 \Rightarrow \neg 2$



Eine davon  
Rückwärtskante

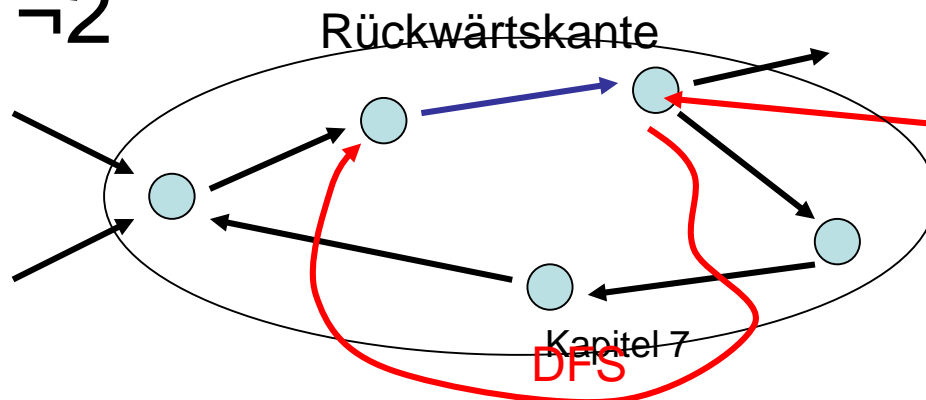
# DFS-Nummerierung

Lemma 7.9: Das Folgende ist äquivalent:

1.  $G$  ist ein DAG
2. DFS enthält keine Rückwärtskante
3.  $\forall (v,w) \in E: \text{finishTime}[v] > \text{finishTime}[w]$

Beweis:

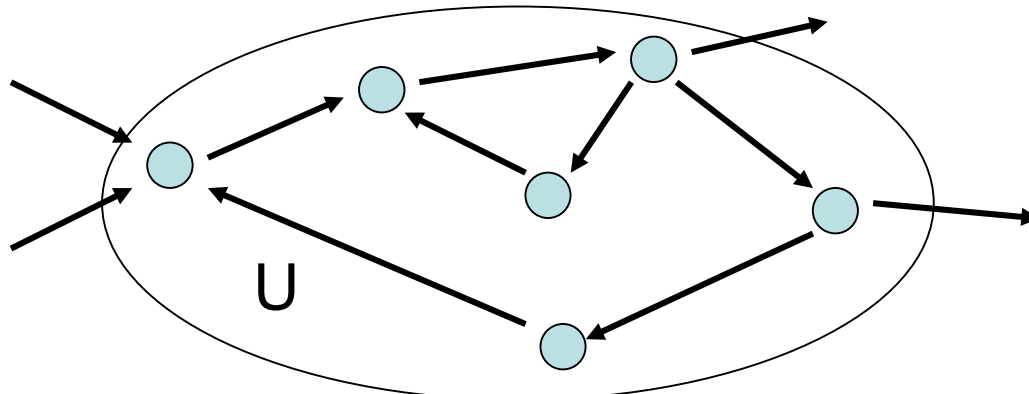
$\neg 1 \Rightarrow \neg 2$



Annahme: Erster von DFS besuchter Knoten im Kreis

# Starke ZHKs

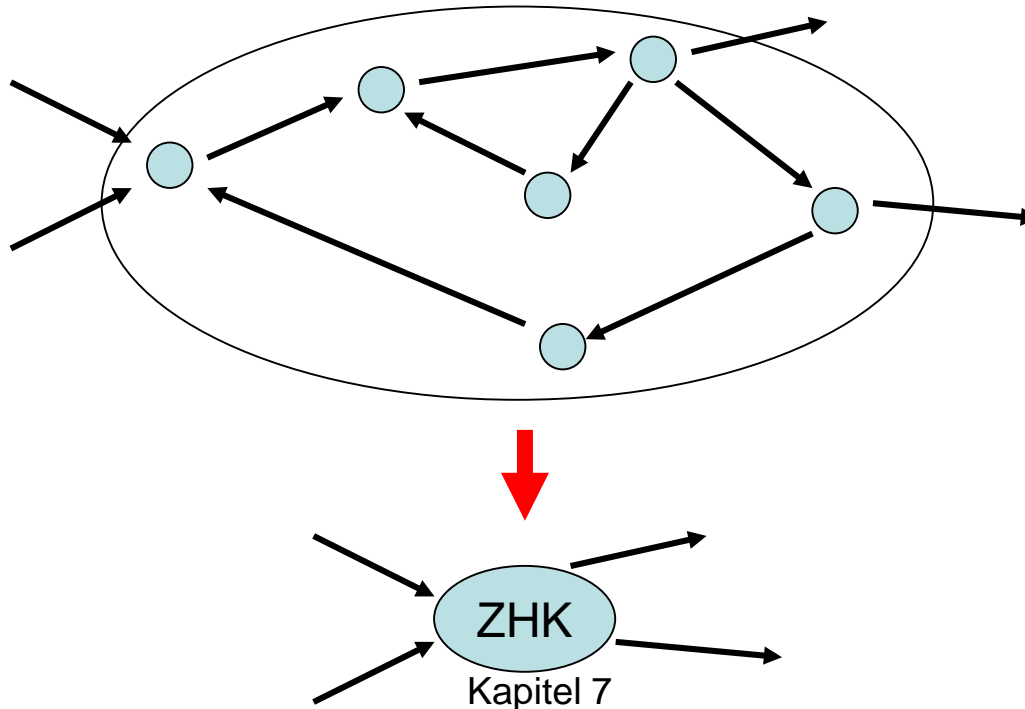
Erinnerung: Sei  $G=(V,E)$  ein gerichteter Graph.  $U\subseteq V$  ist eine **starke Zusammenhangskomponente** (ZHK) von  $V$ , falls es für alle  $u,v\in U$  einen gerichteten Weg von  $u$  nach  $v$  in  $G$  gibt und  $U$  maximal ist



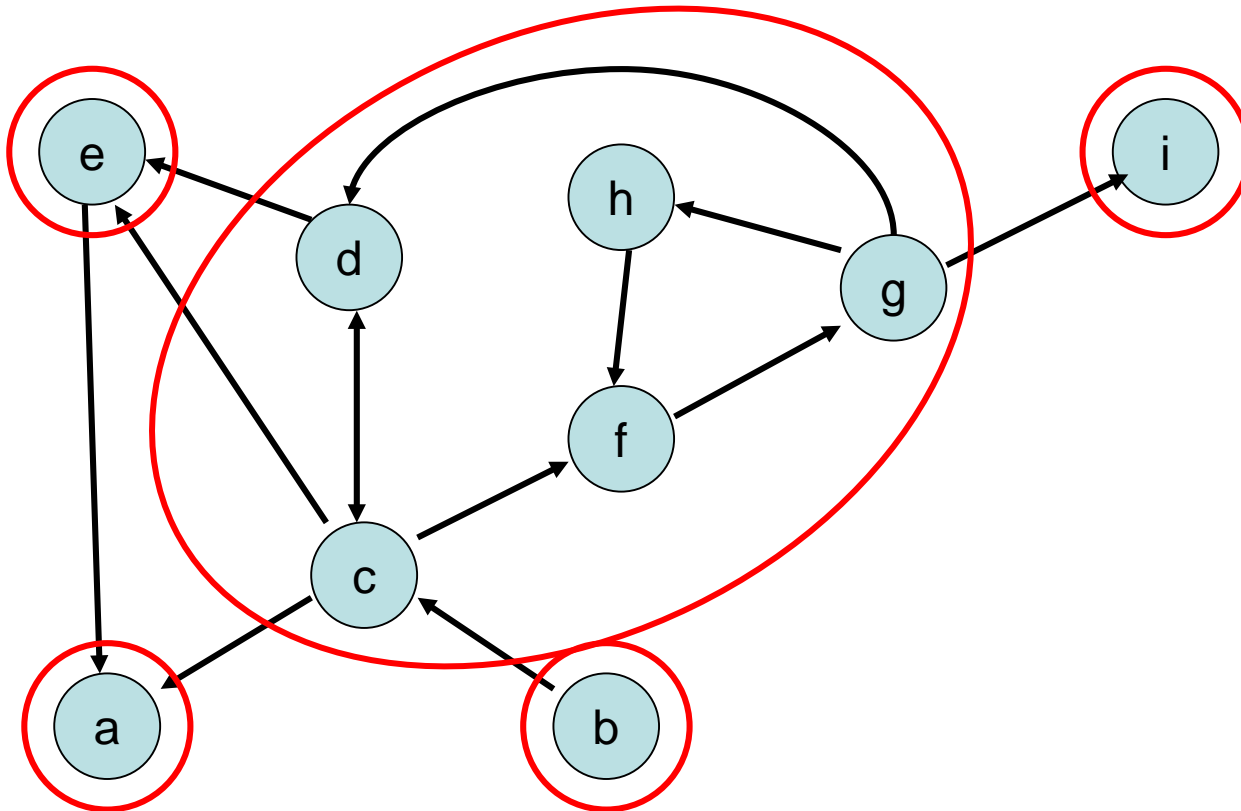


# Starke ZHKs

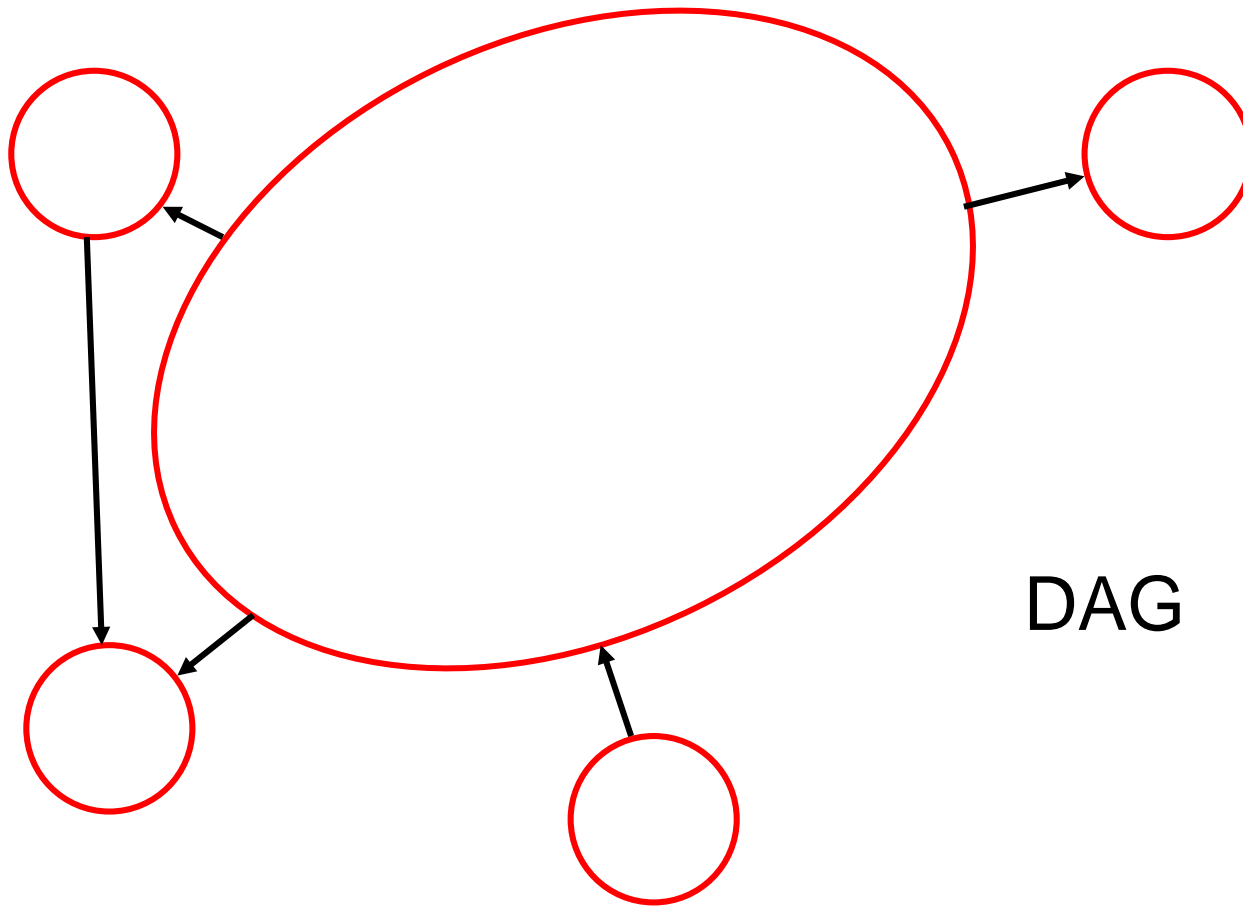
**Beobachtung:** Schrumpft man starke ZHKs zu einzelnen Knoten, dann ergibt sich DAG.



# Starke ZHKs - Beispiel



# Starke ZHKs - Beispiel



# Starke ZHKs

**Ziel:** Finde alle starken ZHKs im Graphen in  $O(n+m)$  Zeit ( $n$ : #Knoten,  $m$ : #Kanten)

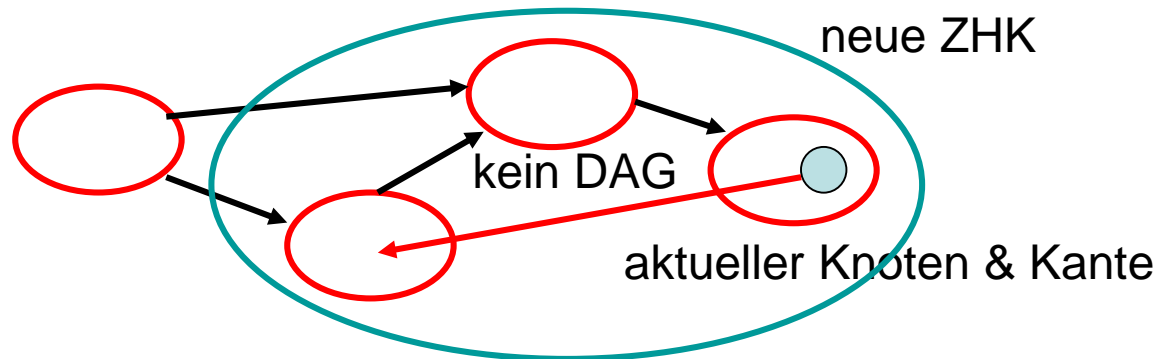
**Strategie:** Verwende DFS-Verfahren mit **component:** Array  $[1..n]$  of  $1..n$

**Am Ende:**  $\text{component}[v]=\text{component}[w]$  ,  
 $v$  und  $w$  sind in derselben starken ZHK

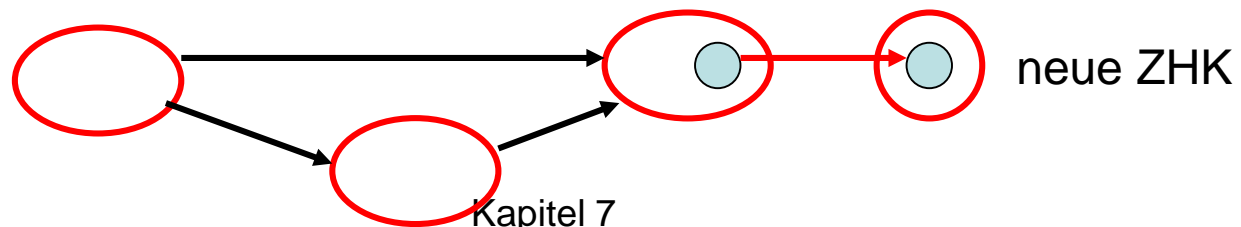
# Starke ZHKs

- Betrachte DFS auf  $G=(V,E)$
- Sei  $G_c=(V_c,E_c)$  bereits besuchter Teilgraph von  $G$
- Ziel: bewahre starke ZHKs in  $G_c$
- Idee:

a)

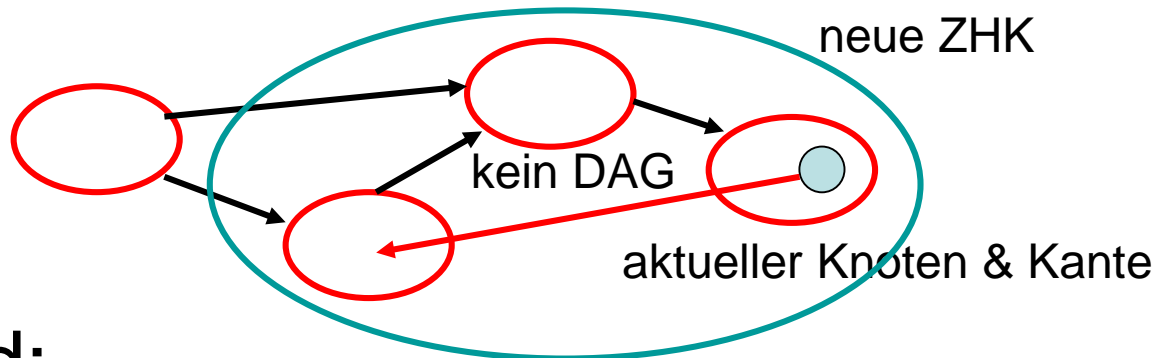


b)

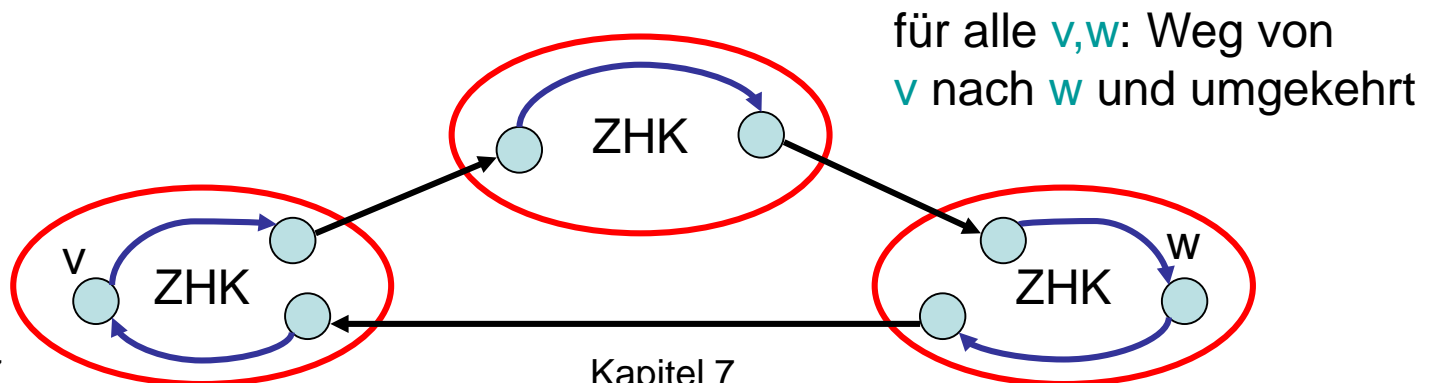


# Starke ZHKs

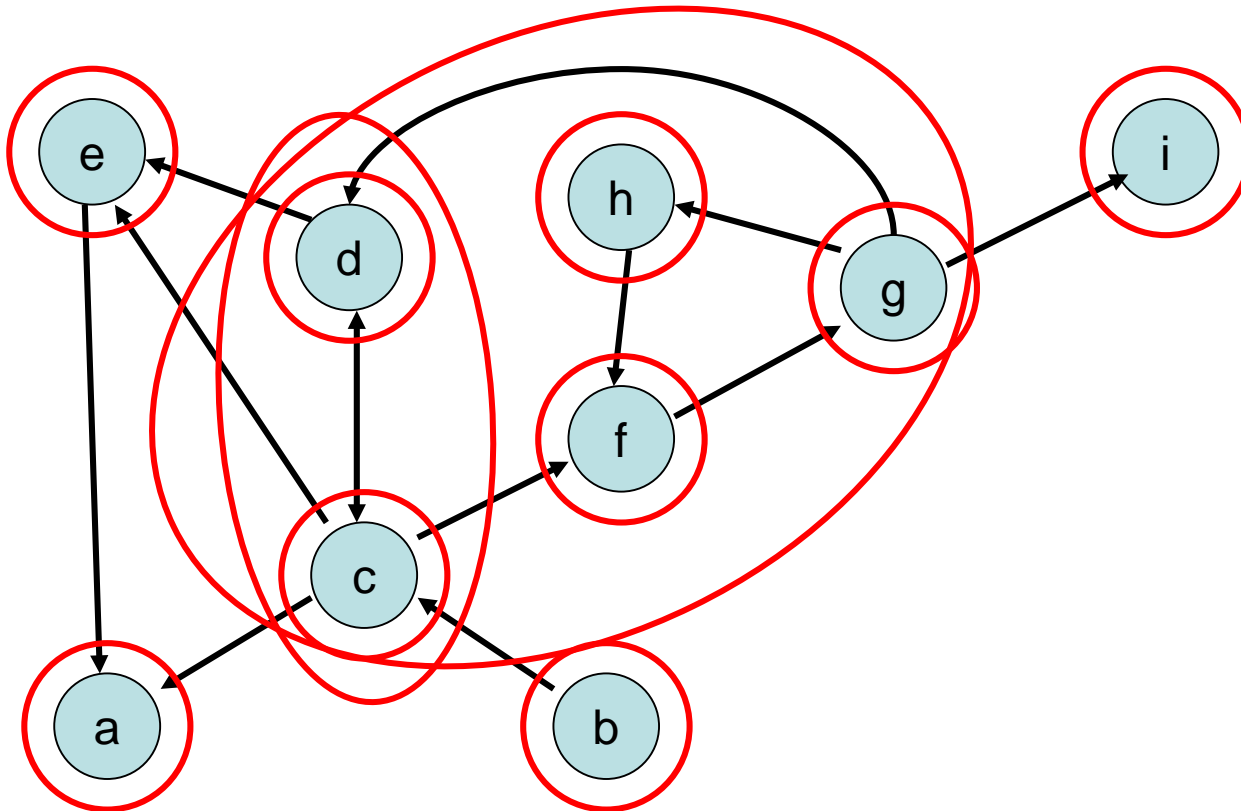
Warum ZHKs zu einer zusammenfassbar?



Grund:






# Starke SHKs - Beispiel



Problem: wie fasst man ZHKs effizient zusammen?

# Starke ZHKs

## Definition:

-   : unfertiger Knoten
-  : fertiger Knoten
- Eine ZHK in  $G$  heißt **offen**, falls sie noch unfertige Knoten enthält. Sonst heißt sie (und ihre Knoten) **geschlossen**.
- **Repräsentant** einer ZHK: Knoten mit kleinster dfsNum.



# Starke ZHKs

## Beobachtungen:

1. Alle Kanten aus geschlossenen Knoten führen zu geschlossenen Knoten.
2. Der Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs.
3. Betrachte die Knoten in offenen ZHKs sortiert nach DFS-Nummern. Die Repräsentanten partitionieren diese Folge in die offenen ZHKs.

# Starke ZHKs

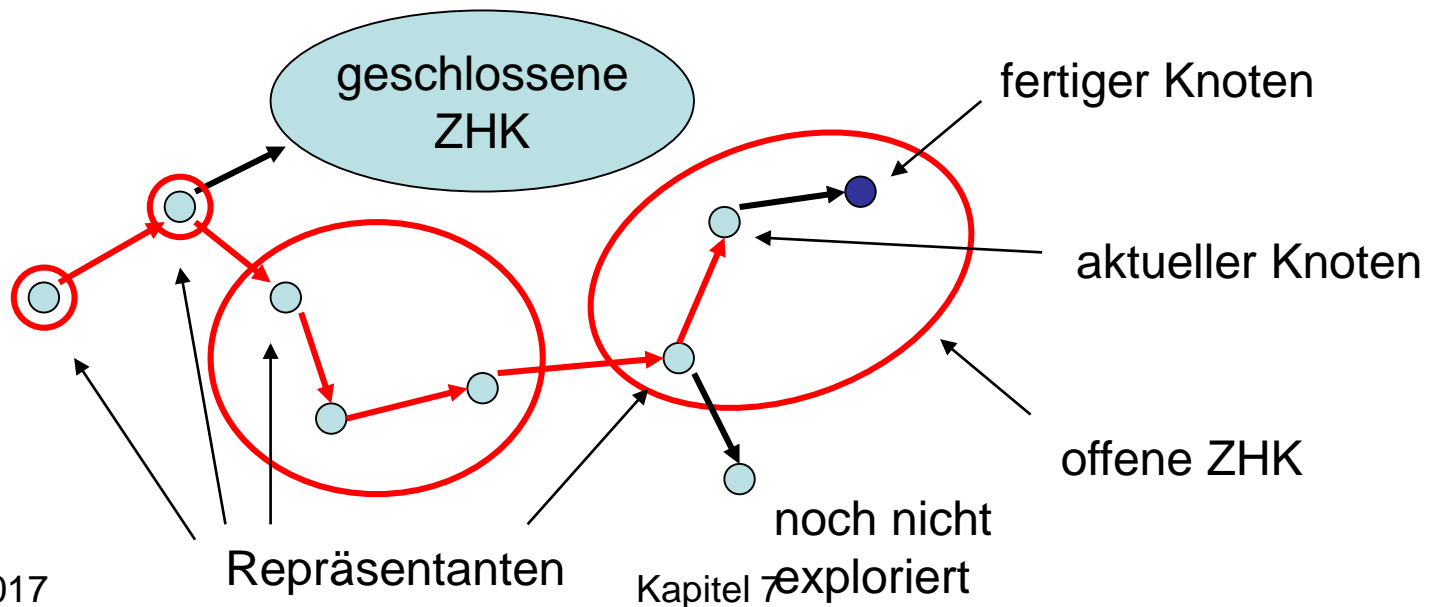
Beobachtungen sind **Invarianten**:

1. Alle Kanten aus geschlossenen Knoten führen zu geschlossenen Knoten.
2. Der Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs.
3. Betrachte die Knoten in offenen ZHKs sortiert nach DFS-Nummern. Die Repräsentanten partitionieren diese Folge in die offenen ZHKs.

# Starke ZHKs

Beweis über vollständige Induktion.

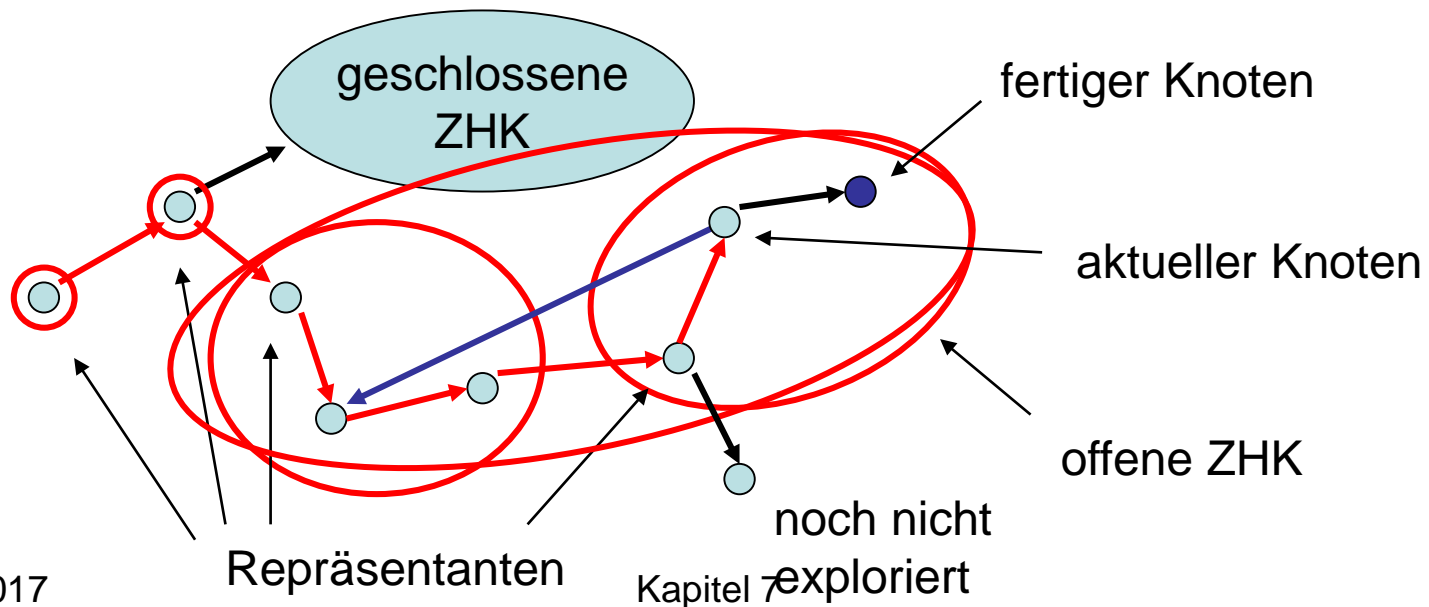
- Anfangs gelten alle Invarianten
- Wir betrachten verschiedene Fälle



# Starke ZHKs

Beweis über vollständige Induktion.

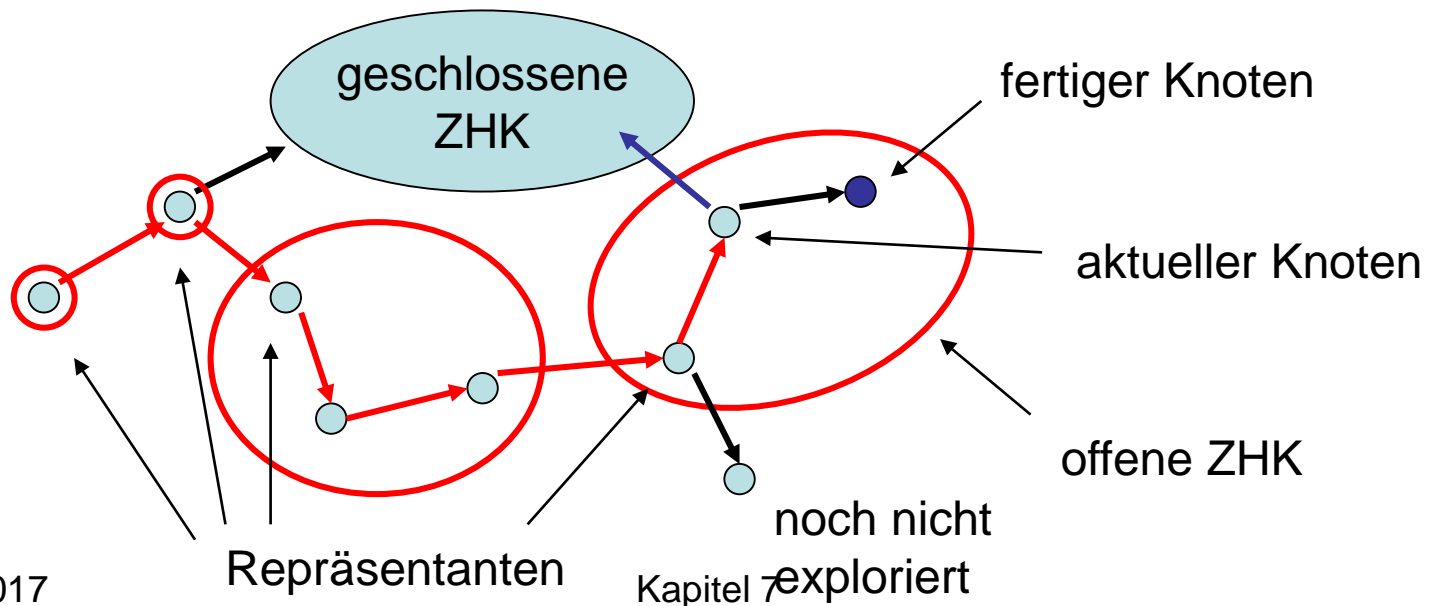
- Anfangs gelten alle Invarianten
- Fall 1: Kante zu unfertigem Knoten



# Starke ZHKs

Beweis über vollständige Induktion.

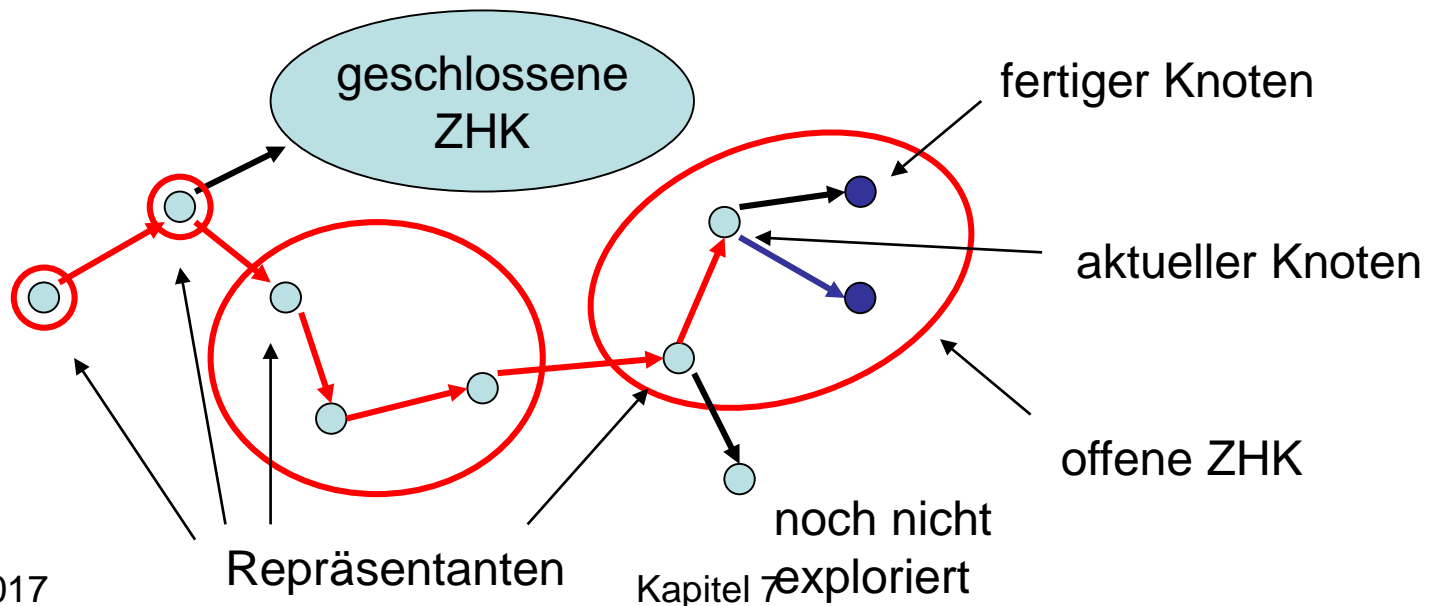
- Anfangs gelten alle Invarianten
- Fall 2: Kante zu geschlossenem Knoten



# Starke ZHKs

Beweis über vollständige Induktion.

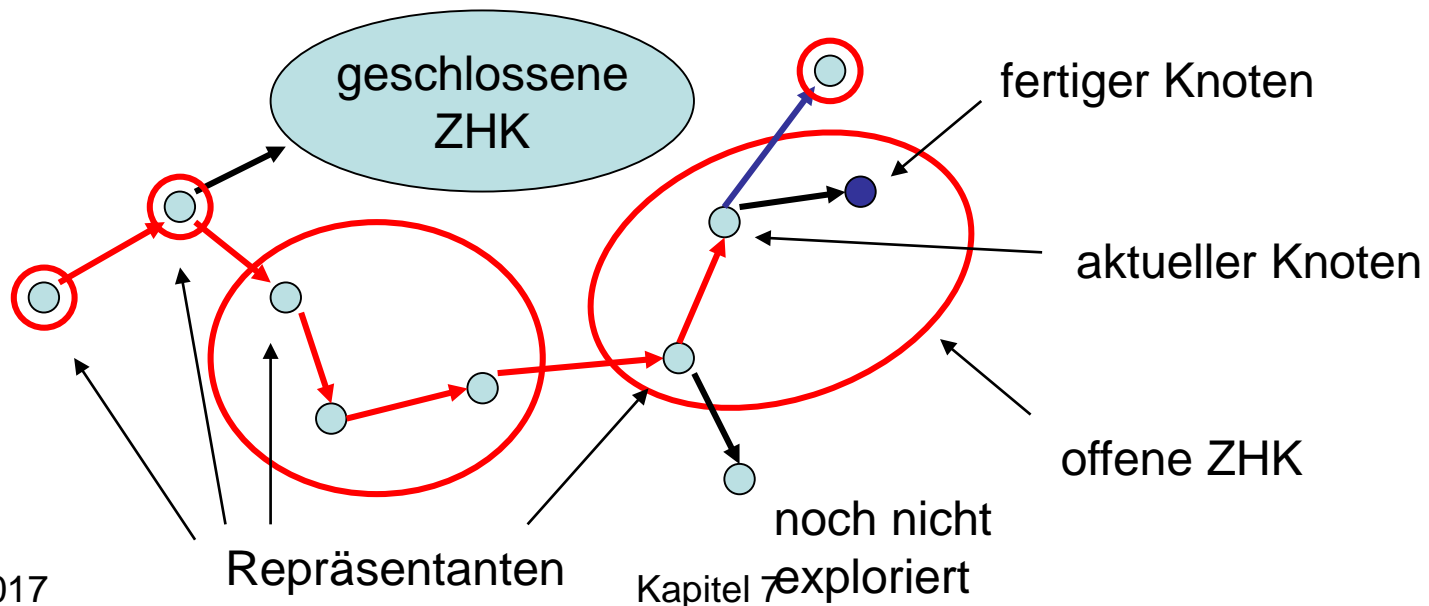
- Anfangs gelten alle Invarianten
- Fall 3: Kante zu fertigem Knoten



# Starke ZHKs

Beweis über vollständige Induktion.

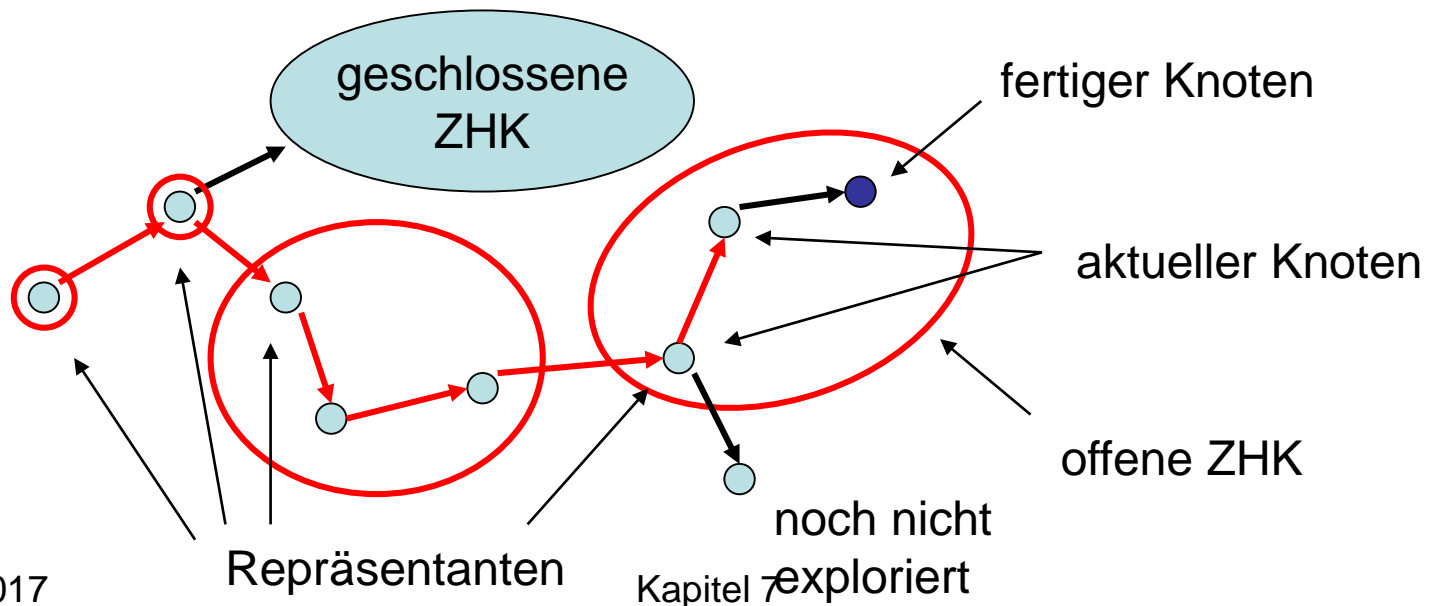
- Anfangs gelten alle Invarianten
- Fall 4: Kante zu nicht exploriertem Knoten



# Starke ZHKs

Beweis über vollständige Induktion.

- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert

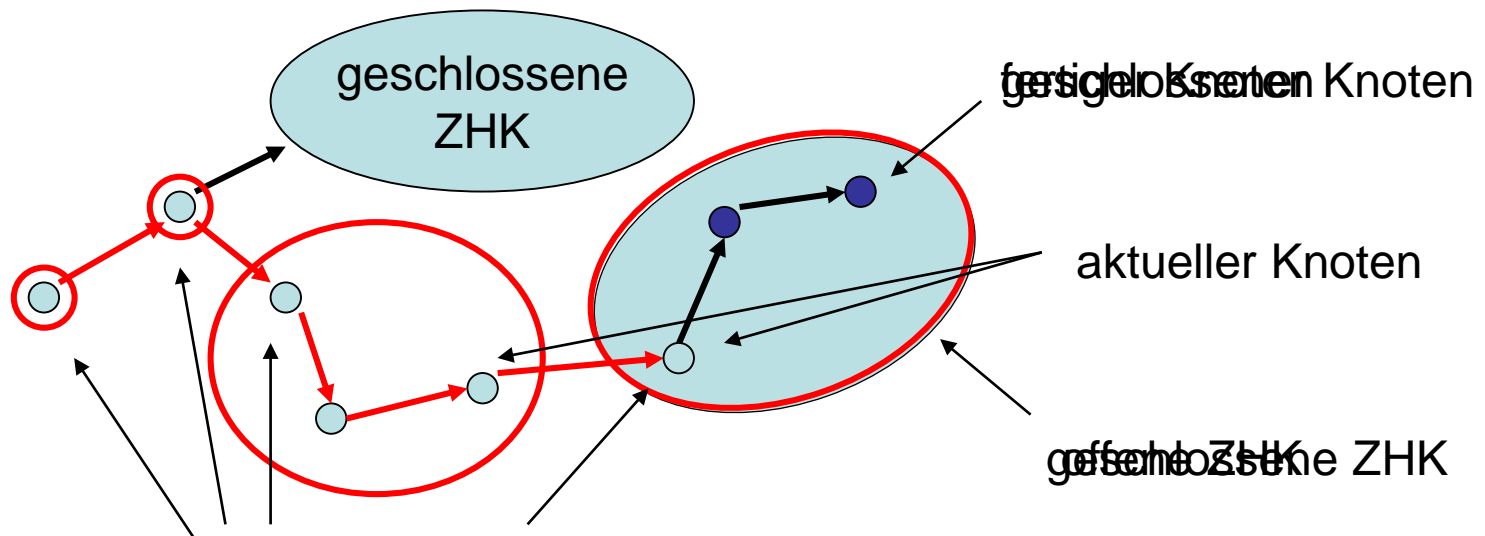




# Starke ZHKs

Beweis über vollständige Induktion.

- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



# Starke ZHKs

Lemma 7.10: Eine geschlossene ZHK in  $G_c$  ist eine ZHK in  $G$ .

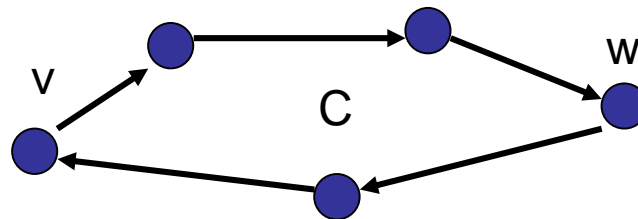
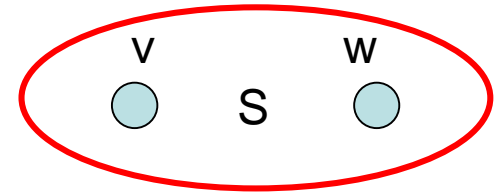
Beweis:

- $v$ : geschlossener Knoten
- $S$ : ZHK in  $G$ , die  $v$  enthält
- $S_c$ : ZHK in  $G_c$ , die  $v$  enthält
- Es gilt:  $S_c \subseteq S$
- Zu zeigen:  $S \subseteq S_c$

# Starke ZHKs

Beweis von Lemma 7.10:

- $w$ : beliebiger Knoten in  $S$
- Es gibt gerichteten Kreis  $C$  durch  $v$  und  $w$
- **Invariante 1**: alle Knoten in  $C$  geschlossen

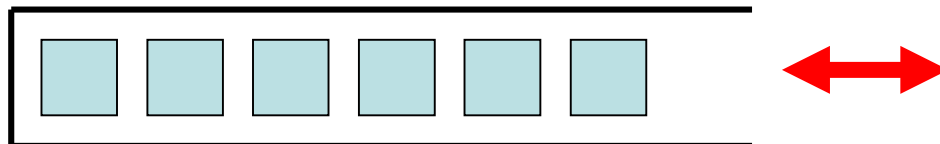


- Da alle Kanten geschlossener Knoten exploriert worden sind, ist  $C$  in  $G_c$  und daher  $w \in S_c$

# Starke ZHKs

Invarianten 2 und 3: einfache Methode, um offene ZHKs in  $G_c$  zu repräsentieren:

- Wir verwalten Folge **oNodes** aller offenen (nicht geschl.) Knoten in steigender DFS-Nummer und eine Teilfolge **oReps** aller offenen ZHK-Repräsentanten
- **Stack** ausreichend für beide Folgen



# Starke ZHKs

init:

```
component: Array [1..n] of NodeId  
oReps = <>: Stack of NodeId  
oNodes = <>: Stack of NodeId  
dfsPos:=1
```

root(w) oder traverseTreeEdge(v,w):

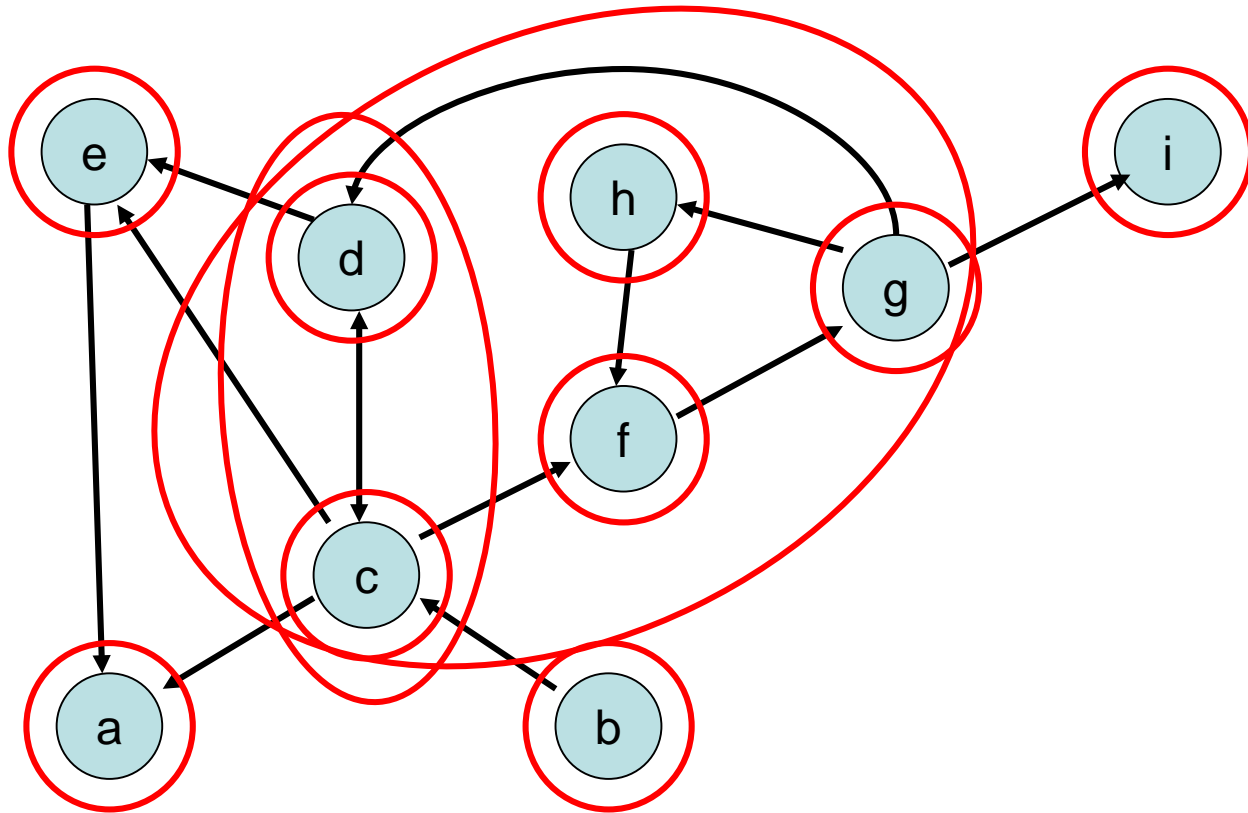
```
oReps.push(w) // neue ZHK  
oNodes.push(w) // neuer offener Knoten  
dfsNum[w]:=dfsPos; dfsPos:=dfsPos+1
```

# Starke ZHKs

```
traverseNonTreeEdge(v,w):  
  if  $w \in oNodes$  then // kombiniere ZHKs  
    while  $dfsNum[w] < dfsNum[oReps.top()]$  do  
       $oReps.pop()$ 
```

```
backtrack(u,v):  
  if  $v = oReps.top()$  then // v Repräsentant?  
     $oReps.pop()$  // ja: entferne v  
    repeat // und offene Knoten bis v  
       $w := oNodes.pop()$   
       $component[w] := v$   
    until  $w = v$ 
```

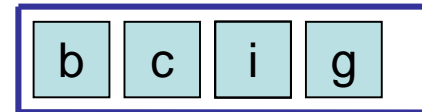
# Starke SHKs - Beispiel



oNodes:



oReps:



# Starke ZHKs

Satz 7.11: Der DFS-basierte Algorithmus für starke ZHKs benötigt  $O(n+m)$  Zeit.

Beweis:

- `init`, `root`, `traverseTreeEdge`: Zeit  $O(1)$
- `Backtrack`, `traverseNonTreeEdge`: da jeder Knoten nur höchstens einmal in `oReps` und `oNodes` landet, insgesamt Zeit  $O(n)$
- DFS-Gerüst: Zeit  $O(n+m)$



# Probleme

- 10004: Bicoloring
- 10067: Playing with Wheels
- 10099: The Tourist Guide
- 117: The Postal Worker Rings Once
- 125: Numbering Paths
- 192: Synchronous Design

Hausaufgabe:

- 10199: Tourist Guide