8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

Rekursive Tiefensuche (depth-first search)

```
void visitDF(Vertex v, Graph g)  {
        Set<Vertex> visited = ∅;
        visitDF(v, g, visited);
}


void visitDF(Vertex v, Graph g, Set<Vertex> visited) {
         visited.add(v);

        // Bearbeite v:
        println(v);

        for ( jeden adjazenten Knoten w von v )
                if ( ! visited.contains(w) )  // w noch nicht besucht
                        visitDF(w, g, visited);
}
```

visitDF(v, g) startet von Knoten v eine Tiefensuche im Graph g.

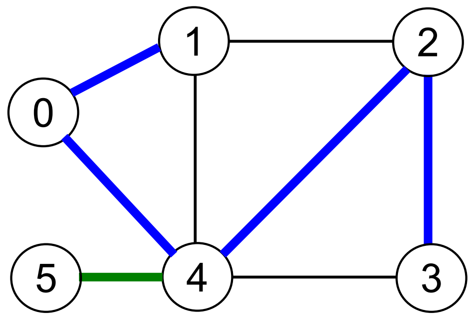
visited ist die Menge aller bereits besuchten Knoten.

Wichtig zur Vermeidung von Endlosschleifen.

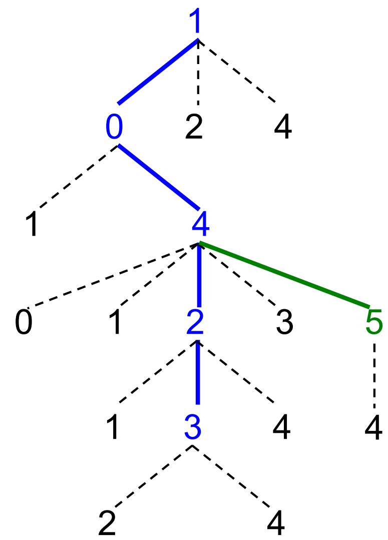
visitDF(v, g, visited) besucht Knoten v im Graphen g und ist rekursiv.

Beispiel für rekursive Tiefensuche

- Tiefensuche mit Start bei Knoten 1



Aufrufstruktur

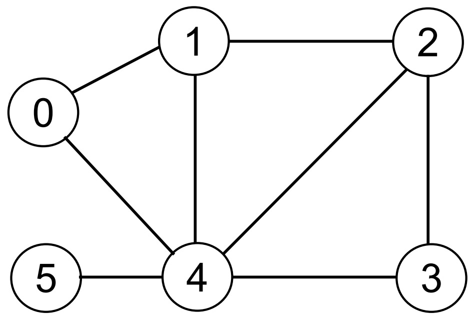


Besuchsreihenfolge:
1, 0, 4, 2, 3, 5

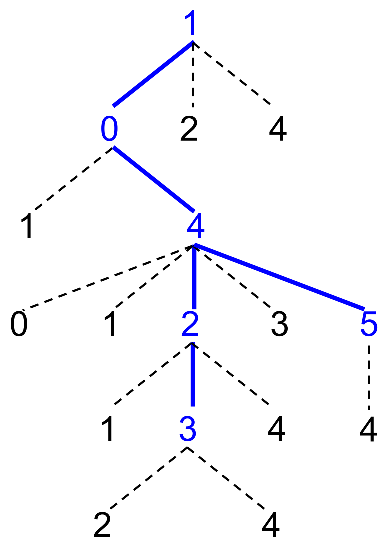
- die Nachbarn eines Knoten werden in numerischer Reihenfolge durchlaufen.
- Bereits besuchte Nachbarknoten sind durch eine gestrichelte Kante verbunden und gehören nicht zur Aufrufstruktur.

Tiefensuchbaum

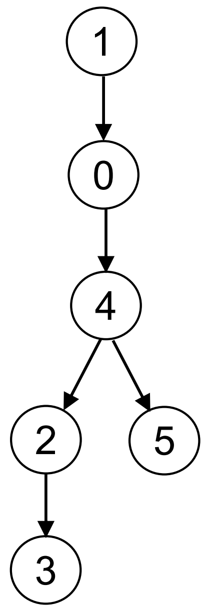
- Die Aufrufstruktur bildet mit den besuchten Knoten den sogenannten Tiefensuchbaum.



Aufrufstruktur



Tiefensuchbaum



Iterative Tiefensuche mit einem Keller

```
void visitDF(Vertex v, Graph g)  {
        Set<Vertex> visited = ∅;
        visitDF(v, g, visited);
}

void visitDF(Vertex v, Graph g, Set<Vertex> visited) {
        Stack<Vertex> stk;
        stk.push(v);

        while( ! stk.empty() )  {
                v = stk.pop();
                if (visited.contains(v) )
                        continue;

                visited.add(v);

                // Bearbeite v:
                println(v);

                for ( jeden adjazenten Knoten w von v )
                        if ( ! visited.contains(w) )
                                stk.push(w);
        }
}
```

visitDF startet von Knoten v eine Tiefensuche.

visitDF besucht Knoten v im Graphen g, wobei die bereits besuchten Knoten in visited abgespeichert werden.

Im Keller stk werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

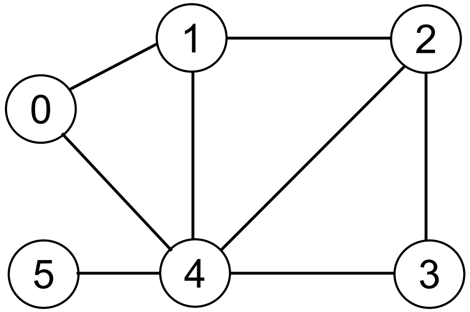
Die Tiefensuche wird erreicht durch die LIFO-Organisation des Kellers.

Um die gleiche Besuchsreihenfolge wie bei der rekursiven Funktion zu erreichen, müssen in der for-Schleife die Nachbarn in umgekehrter Reihenfolge bearbeitet werden.

Beachte: Gleiche Knoten können mehrfach eingekellert werden. Das ließe sich durch eine weitere Knotenmarkierung verhindern:
- nicht besucht und nicht im Keller,
- nicht besucht und im Keller,
- besucht.

Allerdings würde sich eine andere Besuchsreihenfolge als bei der rekursiven Funktion ergeben.

Tiefensuche mit Stack



Keller

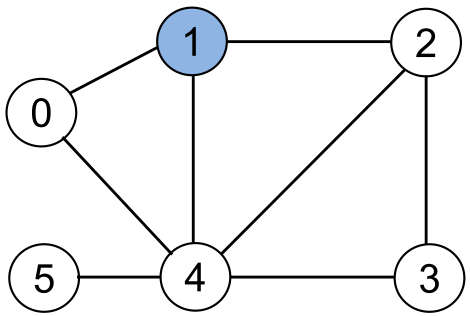
Besuchsreihenfolge:
1, 0, 4, 2, 3, 5

Bitte ausfüllen

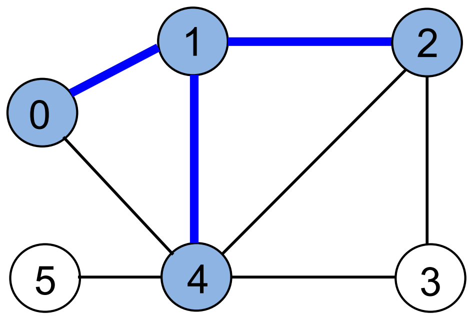
8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

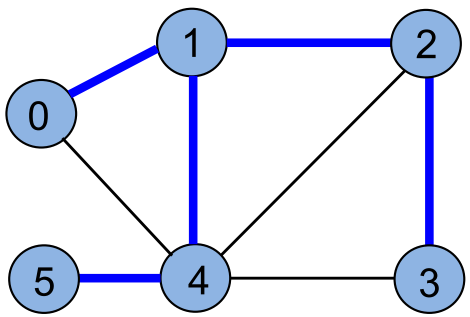
Breitensuche - Beispiel



1) Beginne mit Knoten 1.



2) Besuche alle Knoten, die über genau eine Kante von Knoten 1 erreichbar sind.



3) Besuche alle Knoten, die über genau 2 Kanten von Knoten 1 erreichbar sind.

Besuchsreihenfolge:
1, 0, 2, 4, 3, 5

Breitensuche (breadth-first search) mit einer Schlange

```
void visitBF(Vertex v, Graph g) {
	Set<Vertex> visited = ∅;
	visitBF(v, g, visited);
}

void visitBF(Vertex v, Graph g, Set<Vertex> visited) {
	Queue<Vertex> q;
	q.add(v);

	while( ! q.empty() ) {
		v = q.remove();
		if ( visited.contains(v))
			continue;

		visited.add(v);

		// Bearbeite v:
		println(v);

		for ( jeden adjazenten Knoten w von v )
			if ( ! visited.contains(w) )
				q.add(w);
	}
}
```

visitBF startet von Knoten v eine Breitensuche im Graphen g.

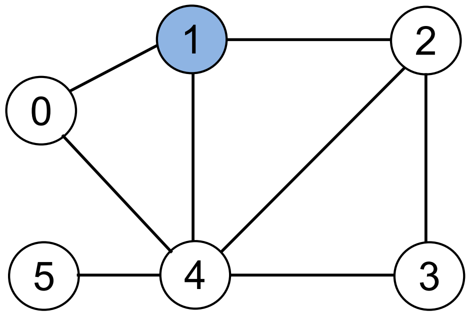
visitBF besucht Knoten v im Graphen g, wobei die bereits besuchten Knoten in der Menge visited abgespeichert werden.

In der Schlange q werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

Die Breitensuche wird erreicht durch die FIFO-Organisation der Schlange

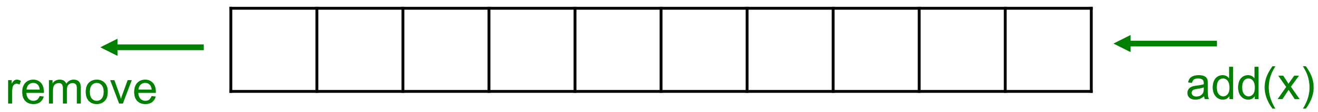
Beachte: Gleiche Knoten können mehrfach in die Schlange eingereiht werden.
Das ließe sich durch eine weitere Knotenmarkierung verhindern:
- nicht besucht,
- nicht besucht und in der Schlange,
- besucht.

Breitensuche mit Queue



Bitte ausfüllen

Schlange



Besuchsreihenfolge:
1, 0, 2, 4, 3, 5

8. Elementare Algorithmen für Graphen

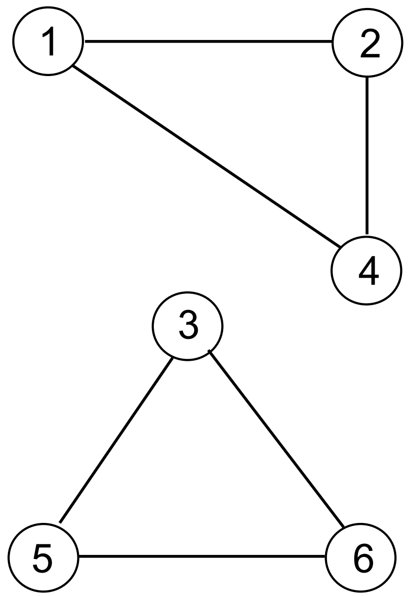
- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

Tiefen- bzw. Breitensuche bei mehreren Zusammenhangskomponenten

- Hat ein Graph mehrere Zusammenhangskomponenten, dann ist nicht garantiert, dass es von jedem Knoten v einen Weg zu jedem anderen Knoten w gibt.
- Sollen alle Knoten besucht werden, dann muss visitDF (Tiefensuche) bzw. visitBF (Breitensuche) mehrere Male mit allen noch nicht besuchten Knoten als Startknoten aufgerufen werden.

```
void visitAllNodes() {

    Set<Vertex> visited = ∅;

    for (jeden Knoten v)
        if (! visited.contains(v) )
            visit(v, g, visited);
}
```

visit kann visitDF oder visitBF sein.

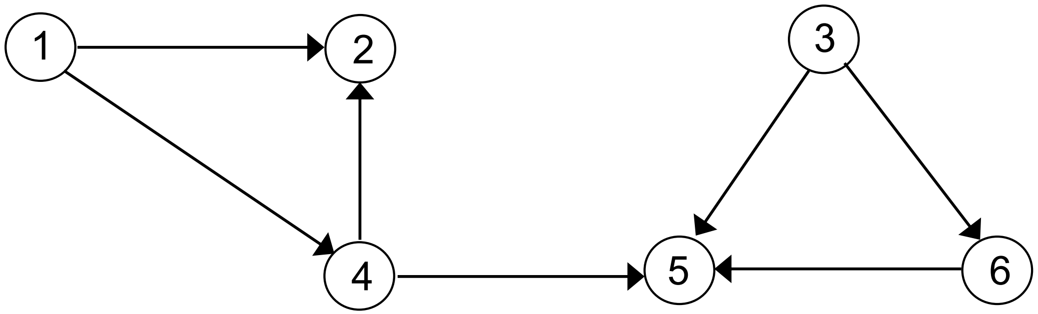


Tiefensuchwald bei Tiefensuche mit mehreren Zusammenhangskomponenten

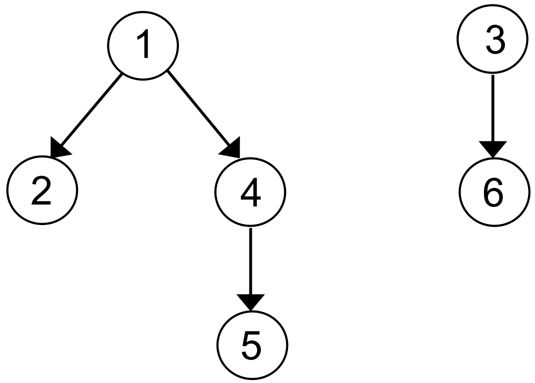
- Bei der Tiefensuche in einem Graphen mit mehreren Zusammenhangskomponenten entsteht dann der sogenannte Tiefensuchwald.

```
void visitDFAllNodes() {

    Set<Vertex> visited = ∅;

    for (jeden Knoten v)
        if (! visited.contains(v))
            visitDF(v, g, visited);
}
```

Gerichteter Graph (mit mehreren starken Zusammenhangskomponenten):



Tiefensuchwald (Knoten werden nach ihrer Nummerierung aufsteigend besucht):



Analyse

- Hier: Analyse der Tiefensuche über den kompletten Graphen.
  Bei der Breitensuche ergibt sich dieselbe Komplexität.

- Jeder Knoten wird genau 1-mal besucht.

- Jede Kante wird genau 2-mal bei ungerichteten Graphen bzw. genau 1-mal bei gerichteten Graphen in einer der for-Schleifen betrachtet.

  Wird die besucht-Menge visited als boolsches Feld implementiert und werden für den Graph Adjazenzlisten verwendet, ist der Aufwand für den Durchlauf aller for-Schleifen höchstens O(|E|).

- Damit ergibt sich insgesamt:

$$T = O(|V| + |E|)$$

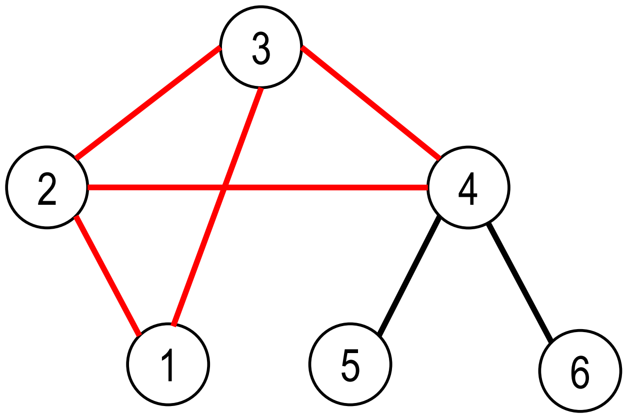
```
void visitDFAllNodes() {
    Set<Vertex> visited = ∅;

    for (jeden Knoten v)
        if (! visited.contains(v) )
            visitDF(v, g, visited);
}

void visitDF(Vertex v, Graph g, Set<Vertex> visited) {
    visited.add(v);

    // Bearbeite v:
    println(v);

    for ( jeden adjazenten Knoten w von v )
        if ( ! visited.contains(w) )
            visitDF(w, g, visited);
}
```

8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

Zyklen in ungerichteten Graphen

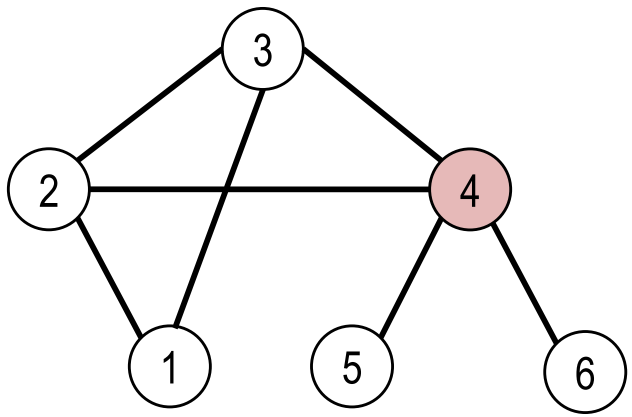
- Ein Zyklus in einem ungerichteten Graphen ist ein Weg mit gleichem Anfangs- und Endknoten, wobei alle Kanten unterschiedlich sind.



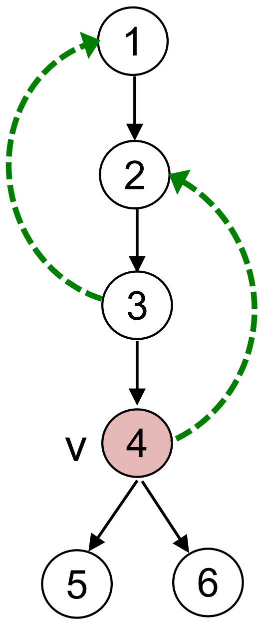
- Alle <u>einfachen</u> Zyklen:
  (d.h. außer Start- und Endknoten sind alle Knoten unterschiedlich)
  - 1, 2, 3, 1
  - 2, 3, 4, 2
  - 1, 2, 4, 3, 1
- Beachte, dass Zyklen sich nicht ändern, falls ein anderer Startknoten gewählt wird:

  Z.B. sind Zyklus 1, 2, 3, 1 und Zyklus 2, 3, 1, 2 identisch

Erkennung von Zyklen mit Tiefensuche (1)



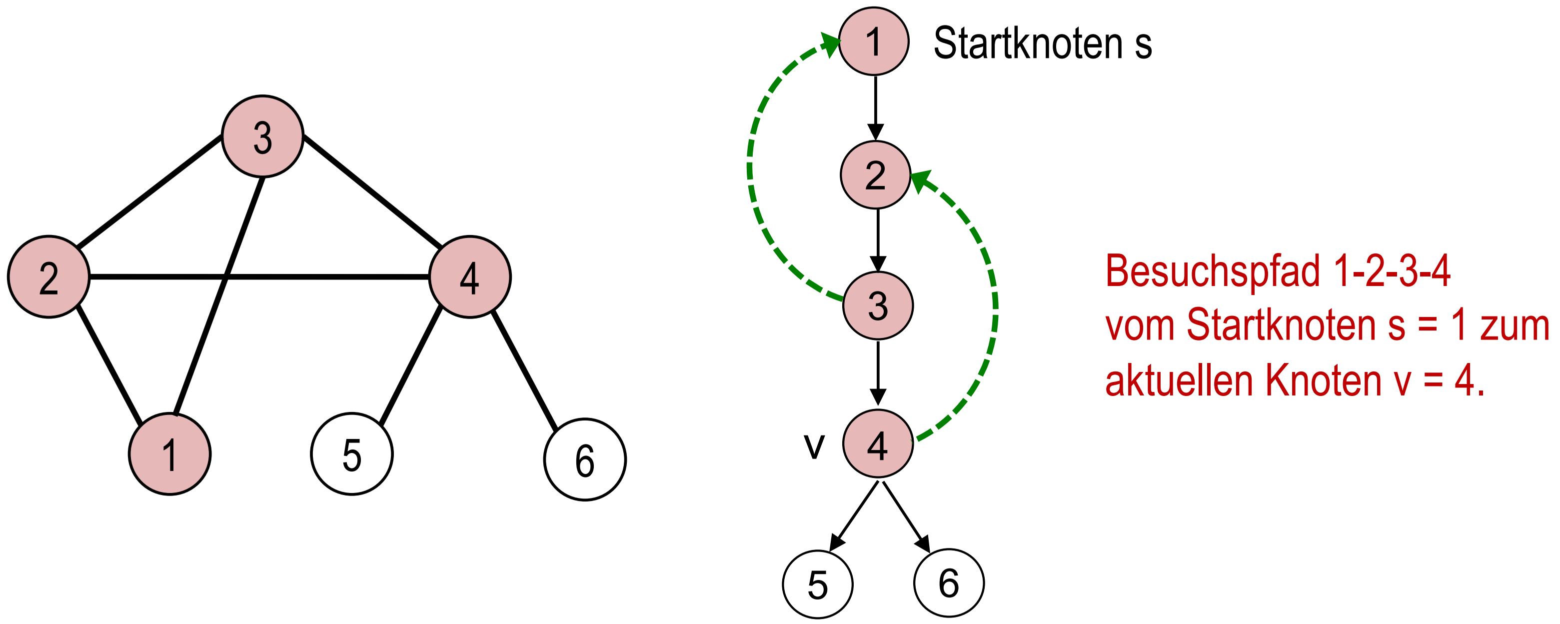
Tiefensuchbaum mit Startknoten 1



Rückwärtskante zu einem bereits besuchten Knoten, der indirekter Vorfahre ist.

- Sobald in der Tiefensuche ein Knoten v besucht wird (z.B. v = 4), dann werden alle Nachbarknoten w betrachtet. Es gibt genau drei Fälle:
  (1) w ist bereits besucht und direkter Vorfahre (w = 3)
  (2) w ist bereits besucht und indirekter Vorfahre (Rückwärtskante zu w = 2)
  (3) w ist noch nicht besucht (w = 5 und w = 6)
- Nur Knoten der Kategorie (2) sind für Zyklen relevant.

Erkennung von Zyklen mit Tiefensuche (2)



- Idee für Algorithmus:
  - Führe in der rekursiven Tiefensuche den Besuchspfad vom Startknoten s zum aktuellen Knoten v als zusätzlichen Parameter mit.
  - Sobald ein Nachbarknoten w von v als besucht erkannt wurde, lässt sich anhand des Besuchspfads feststellen, ob w ein indirekter Vorfahre ist und damit ein Zyklus vorliegt.
  - Darüber hinaus kann der Zyklus mit Hilfe des Besuchspfads ausgegeben werden.

Zyklenerkennung mit rekursiver Tiefensuche

```
void searchUndirectedCycle(Graph g)  {
        Set<Vertex> visited = ∅;
        Stack<Vertex> path = empty;

        for (jeden Knoten s)
                if (! visited.contains(s))
                        searchUndirectedCycle(s, g, visited, path);
}

void searchUndirectedCycle(Vertex v, Graph g, Set<Vertex> visited, Stack<Vertex> path) {
        visited.add(v);
        vp  = path.top(); // direkter Vorfahre von v
        path.push(v);

        for ( jeden Nachbar-Knoten w von v ) {
                if ( ! visited.contains(w) )  // w noch nicht besucht
                        searchUndirectedCycle(w, g, visited, path);
                else if (w ≠  vp) { // Knoten w ist indirekter Vorfahre
                        println("Zyklus erkannt");
                        Zyklus mit Hilfe von path ausgeben;
                }
        }

        path.pop();   // Besuch von v ist erledigt.  Daher wird v aus path entfernt.
}
```

Graph g ist ein ungerichteter Graph

path ist der aktuelle Besuchspfad vom Startknoten s zum aktuellen Knoten v.

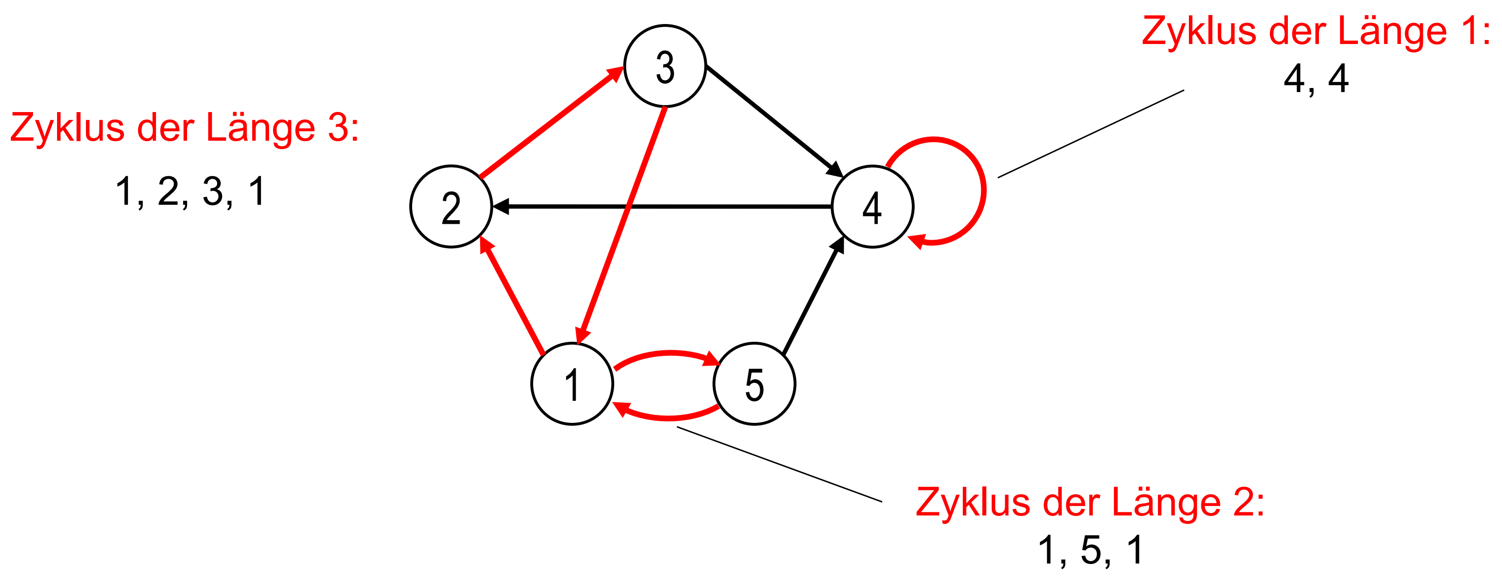
path wird als Stack verwaltet.

8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

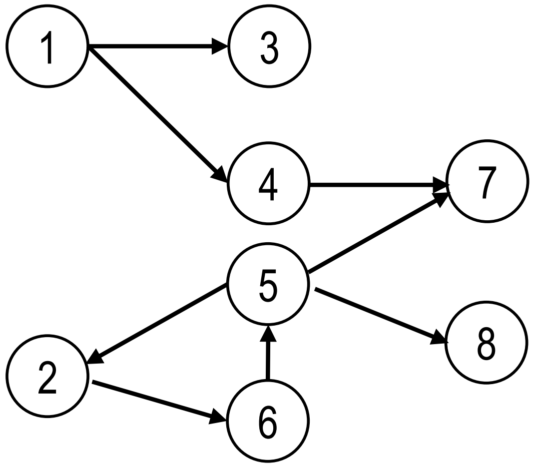
Zyklen in gerichteten Graphen

- Ein Zyklus in einem gerichteten Graphen ist ein Weg mit gleichem Anfangs- und Endknoten.

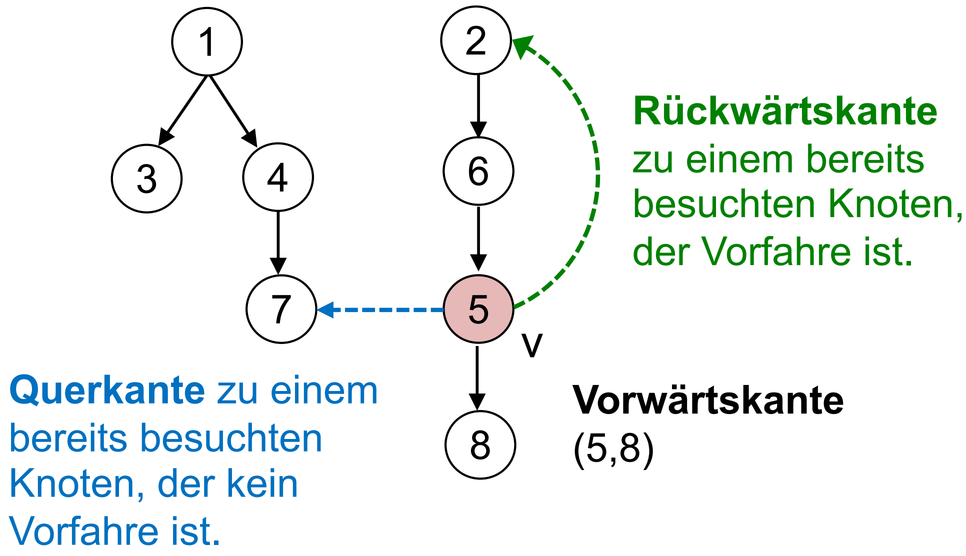


- Alle <u>einfachen</u> Zyklen (d.h. außer Start- und Endknoten sind alle Knoten unterschiedlich):
  - 1, 2, 3, 1
  - 2, 3, 4, 2
  - 4, 4
  - 1, 5, 1

Erkennung von Zyklen mit Tiefensuche (1)

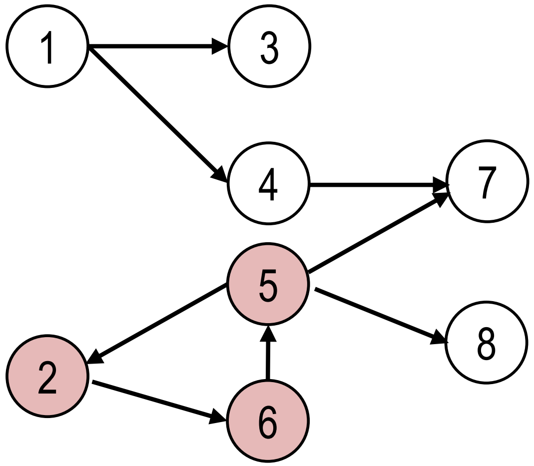


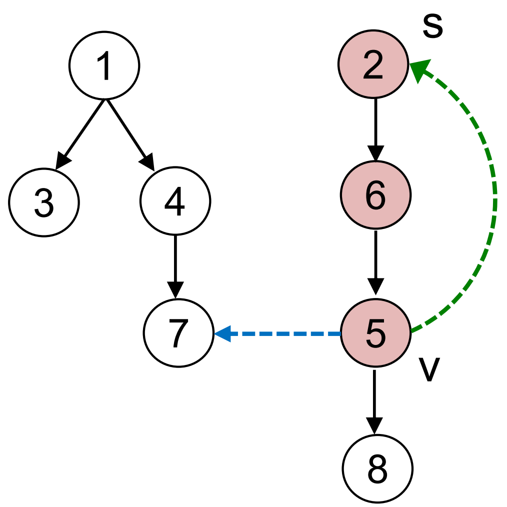
Tiefensuchwald (schwarze Kanten)



- Sobald in der Tiefensuche ein Knoten v besucht wird (z.B. v = 5), dann werden alle Nachfolgerknoten w betrachtet. Es gibt genau drei Fälle:
  - (1) w ist bereits besucht und (direkter oder indirekter) Vorfahre. Rückwärtskante zu w = 2.
  - (2) w ist bereits besucht und kein Vorfahre. Querkante zu w = 7.
  - (3) w ist noch nicht besucht. Vorwärtskante zu w = 8.
- Nur Knoten der Kategorie (1) sind für Zyklen relevant.

Erkennung von Zyklen mit Tiefensuche (2)





Besuchspfad vom Startknoten s zum aktuellen Knoten v.

- Idee für Algorithmus:
  - Führe in der rekursiven Tiefensuche den Besuchspfad vom Wurzelknoten s zum aktuellen Knoten v als zusätzlichen Parameter mit.
  - Sobald ein Nachbarknoten w von v als besucht erkannt wurde,
    lässt sich anhand des Besuchspfads feststellen, ob ein Zyklus vorliegt.
  - Darüber hinaus kann der Zyklus mit Hilfe des Besuchspfads ausgegeben werden.

Zyklenerkennung mit rekursiver Tiefensuche

```
void searchDirectedCycle(Graph g)  {
        Set<Vertex> visited = ∅;
        Stack<Vertex> path = empty;
        Set<Vertex> nodeInPath = ∅;

        for (jeden Knoten s)
                if (! visited.contains(s))
                        searchDirectedCycle(s, g, visited, path, nodeInPath);
}

void searchDirectedCycle(Vertex v, Graph g, Set<Vertex> visited, Stack<Vertex> path, Set<Vertex> nodeInPath) {
        visited.add(v);
        path.push(v);
        nodeInPath.add(v);

        for ( jeden Nachfolger w von v )
                if ( ! visited.contains(w) )  // w noch nicht visited
                        searchDirectedCycle(w, g, visited, path, nodeInPath);
                else if (nodeInPath.contains(w)) { // Kante (v,w) ist Rückwartskante
                        println("Zyklus erkannt");
                        Zyklus mit Hilfe von path ausgeben;
                }
        }
        path.pop();
        nodeInPath.remove(v);
}
```

Graph g ist ein gerichteter Graph

path ist der aktuelle Besuchspfad vom Startknoten s zum aktuellen Knoten v. path ist als Stack realisiert.

nodeInPath ist die Menge aller Knoten im Besuchspfad path. Dient der effizienteren Überprüfung, ob ein Knoten im Pfad vorhanden ist.

Tipp: path und nodeInPath lassen sich als LinkedHashSet aus der Java API zusammenfassen!

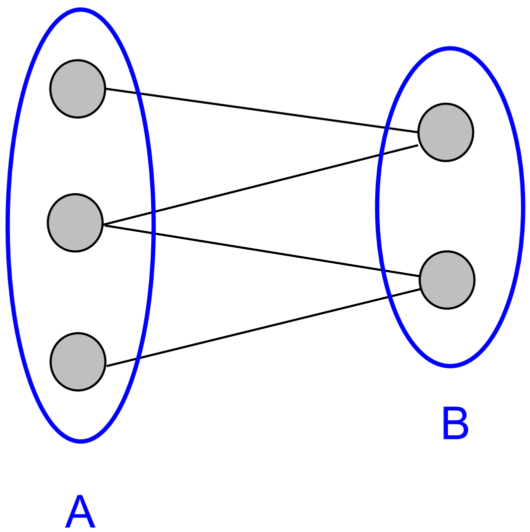
Besuch von v ist erledigt. Daher wird v aus path und nodeInPath entfernt.

8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

Bipartiter Graph

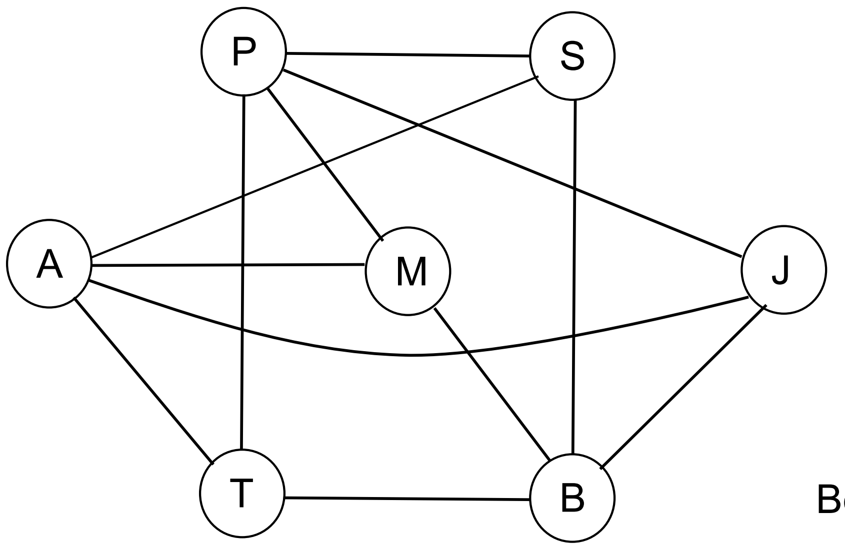
- Ein ungerichteter Graph G = (V, E) heißt bipartit, falls sich V disjunkt in A und B zerlegen lässt, so dass für jede Kante $(u,v) \in E$ gilt:
  - $u \in A$ und $v \in B$ oder
  - $u \in B$ und $v \in A$.



Es gibt nur Kanten zwischen A und B.

Beispiel Sympathiegraph

- Für eine Menge von Personen V ist ein Sympathiegraph G = (V, E) gegeben.
- (u,v) ist eine Kante in E gdw. sich u und v gegenseitig sympathisch finden.



Beispiel aus [Vöcking], Kap. 7

- Gesucht ist (für eine Show) eine Zerlegung von V in zwei disjunkte Personengruppen A und B, die sich nicht untereinander sympathisch finden.
- Also: Ist G ein bipartiter Graph?
  Wenn ja, wie sieht die disjunkte Zerlegung in A und B aus?

Tiefensuche zur Prüfung der Bipartitheit

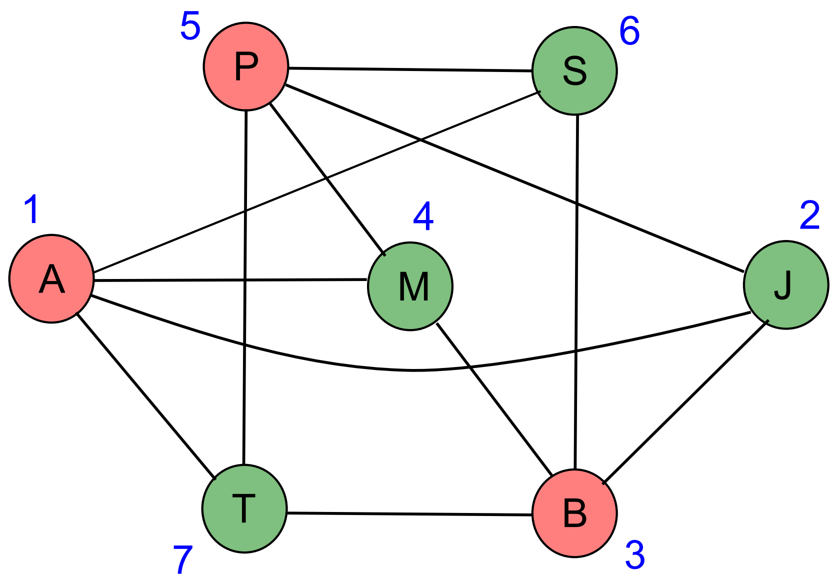
- Besuche mit Tiefensuche alle Knoten und färbe abwechselnd mit rot oder grün ein bzw. prüfe ob Färbung OK.

```
void pruefeBipartheit() {
    for (jeden Knoten v)
        if (v hat noch keine Farbe)
            färbe(v, “rot”))
    print(“Graph ist bipartit”);
}
```

```
void färbe(Vertex v, Farbe f) {
    if (v bereits gefärbt)  {
        if (Farbe von v ist nicht f) {
            print(“Graph ist nicht bipartit”);
            beende Prüfung;
        }
    } else {
        färbe v mit f ein;
        for ( jeden Nachbarn w von v )
            färbe(w, flip(f));
    }
}
```

flip(“rot”) = “grün” und
flip(“grün”) = “rot” .

Tiefensuche mit alphabetischer Reihenfolge.
Reihenfolge der Färbung in blau.



8. Elementare Algorithmen für Graphen

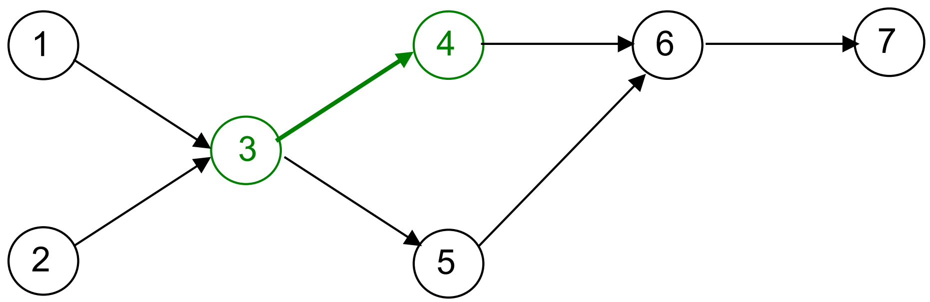
- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Zyklenerkennung bei ungerichteten Graphen
- Zyklenerkennung bei gerichteten Graphen
- Bipartiter Graph
- Topologisches Sortieren

Definition topologische Sortierung

- Eine Folge $v_0, v_1, v_2, \ldots, v_{n-1}$ aller Knoten eines gerichteten Graphen G heißt topologische Sortierung, falls für alle Knoten u, v folgende Bedingung gilt:

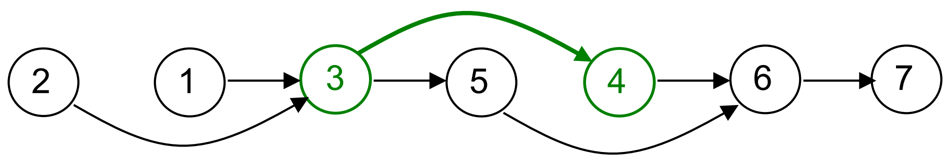
  falls (u,v) eine Kante in G ist,
  dann steht u vor v in der Folge $v_0, v_1, v_2, \ldots, v_{n-1}$.

- Beispiel:



(3,4) ist Kante

topologisch sortiert:



3 steht vor 4

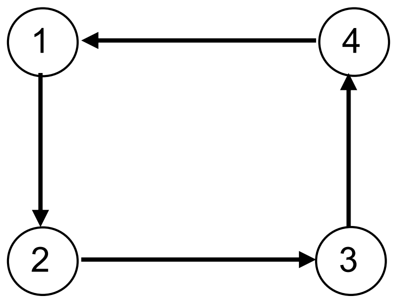
- Anschaulich:

  Der Graph lässt sich so auf eine Linie „verbiegen",
  dass die Pfeile nur nach rechts gehen.

Eigenschaften und Anwendungen

Eigenschaften:

- Falls der Graph einen Zyklus enthält, dann existiert keine topologische Sortierung und umgekehrt.

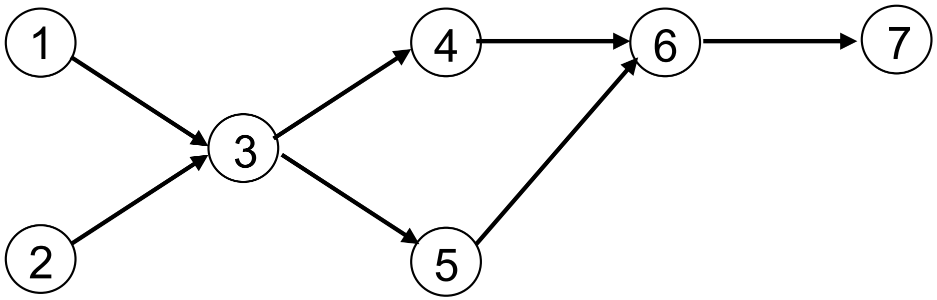


- Falls ein Graph topologisch sortiert werden kann, dann ist im allgemeinen die topologische Sortierung nicht eindeutig.

Typische Anwendung:

- Ermittle eine Durchführungsreihenfolge für die Aktivitäten oder auch Prozesse in einem Vorranggraphen.

Topologische Sortierung mit Hilfe einer Queue (1)



inDegree und
Queue einzeichnen!

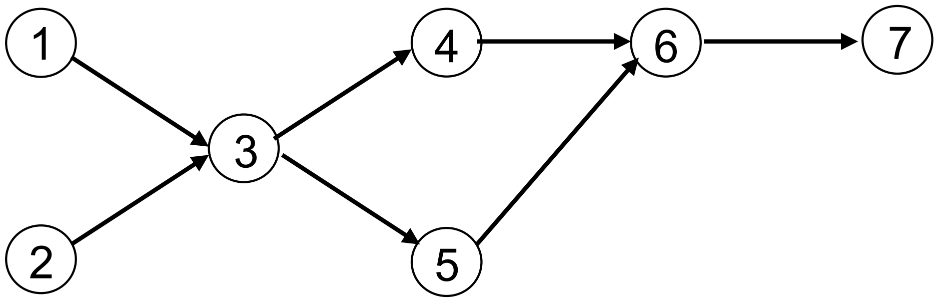
- Speichere für jeden Knoten v:
  inDegree[v] = Anzahl der noch nicht besuchten Vorgänger
- Halte alle noch nicht besuchten Knoten v mit inDegree[v] == 0
  als Kandidat in einer Queue q.
- Queue teilt die Knoten in 3 Klassen ein:
  - Knoten links der Queue-Knoten sind bereits besucht
  - die Knoten aus der Queue können als nächstes besucht werden.
  - Die Knoten rechts der Queue-Knoten sind noch nicht registriert.
- Ähnelt einer Breitensuche

Topologische Sortierung mit Hilfe einer Queue (2)

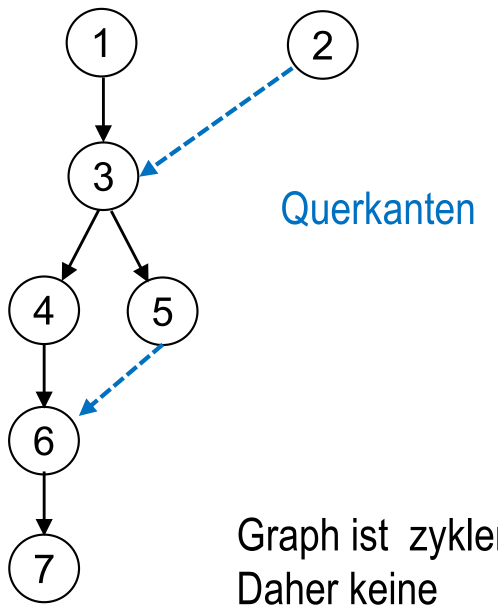
```
List<Vertex> topSort(DiGraph g) {
	List<Vertex> ts;  // topologisch sortierte Folge
	int[ ] inDegree;   // Anz. noch nicht besuchter Vorgänger
	Queue<Vertex> q;

	for (jeden Knoten v) {
		inDegree[v] = Anzahl der Vorgänger;
		if (inDegree[v]  == 0)
			q.add(v);
	}

	while (! q.empty() ) {
		v = q.remove();
		ts.add(v);
		for ( jeden Nachfolger w von v )
			if (--inDegree[w] == 0)
				q.add(w);
	}

	if (ts.size() != Anzahl Knoten in g)
		return null;		// Graph zyklisch;
	else
		return ts;
}
```

Topologische Sortierung mit Tiefensuche

- Bei einem zyklenfreien Graph lässt sich die topologische Sortierung unmittelbar aus Reihenfolge der Tiefensuche ablesen!
- Wie?



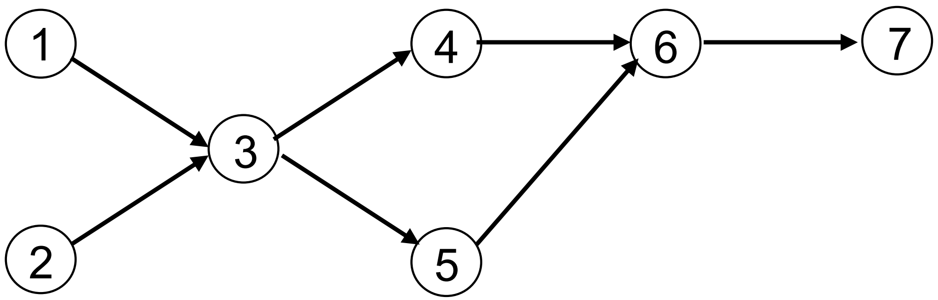
Tiefensuchwald
(Vorwärtskanten in schwarz)

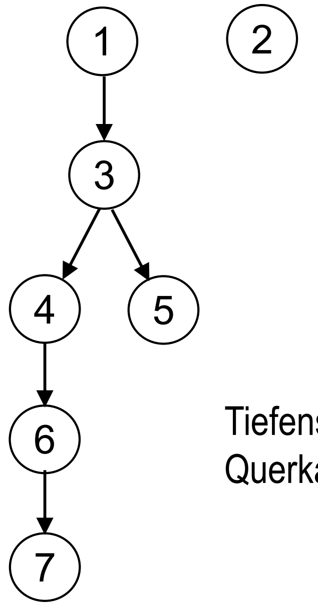


Graph ist zyklenfrei.
Daher keine
Rückwärtskanten.

InOrder- und PostOrder-Reihenfolge

- Bei der rekursive Tiefensuche gibt es zwei wichtige Reihenfolgen der besuchten Knoten:
  – PreOrder: jeder Knoten wird, sobald er besucht wird (Eintritt in die rekursive Besuchsmethode), in eine Liste angehängt. Entspricht der bisher besprochenen Reihenfolge.
  – PostOrder: jeder Knoten wird, sobald der Besuch des Knotens beendet ist (Verlassen der rekursiven Besuchsmethode), in eine Liste angehängt.





Tiefensuchwald
Querkanten weggelassen

- PreOrder: 1, 3, 4, 6, 7, 5, 2
- PostOrder: 7, 6, 4, 5, 3, 1, 2

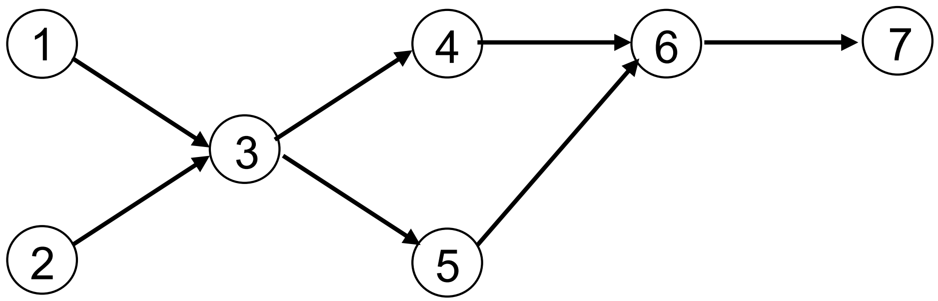
- Beachte, dass die Reihenfolge der betrachteten Knoten in den for-Schleifen der Tiefensuche hier willkürlich auf eine numerische Reihenfolge festgelegt wurde.
  Im Beispiel: Zuerst 1 und dann 2 und zuerst 4 und dann 5.
- Eine andere Reihenfolge geht selbstverständlich auch, was im allgemeinen eine andere PreOrder- bzw. PostOrder-Reihenfolge ergibt.

Topologische Sortierung mit Tiefensuche

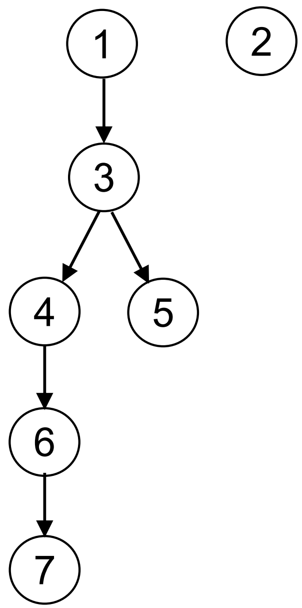
Algorithmus:

(1) Prüfe mit Tiefensuche, ob Graph zyklenfrei ist (Algo. Seite 8-24)
(2) und ermittle dabei eine PostOrder-Reihenfolge.
(3) Die invertierte PostOrder-Reihenfolge ist eine toplogische Sortierung.

Beispiel:



Tiefensuchwald
(Querkanten weggelassen)



(1) Tiefensuche findet keine Rückwärtskanten.
Daher ist Graph zyklenfrei.

(2) PostOrder-Reihenfolge: 7, 6, 4, 5, 3, 1, 2

(3) Invertierte PostOrder-Reihenfolge ist eine
topologische Sortierung: 2, 1, 3, 5, 4, 6, 7

Wieso ist invertierte PostOrder-Reihenfolge eine topologische Sortierung?

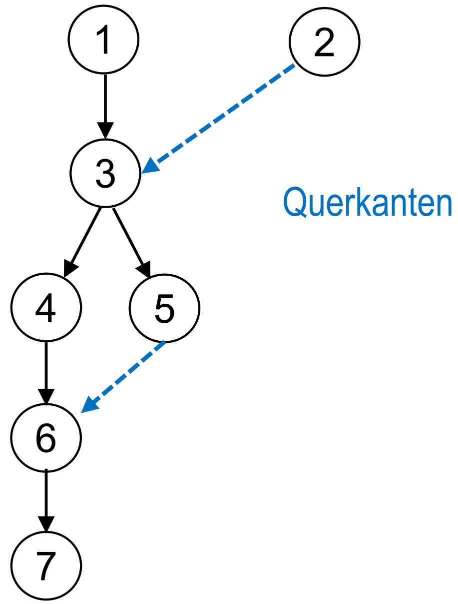
Satz:

Falls ein gerichteter Graph zyklenfrei ist, dann ist eine invertierte PostOrder-Reihenfolge eine topologische Sortierung.

Beweis:

- Sei (v,w) eine beliebige gerichtete Kante des Graphen. Wir müssen zeigen, dass v vor w in der invertierten PostOrder-Reihenfolge steht.
- Sobald in der Tiefensuche v besucht wird, gibt es für den Nachfolgerknoten w genau zwei Fälle:
    1. w wurde schon besucht.
       (v,w) kann keine Rückwärtskante sein, da der Graph zyklenfrei ist. Daher muss (v,w) eine Querkante sein.
       Dann wurde w schon in die PostOrder-Reihenfolge eingefügt und steht damit vor v.
    2. w wurde noch nicht besucht.
       Dann ist (v,w) eine Vorwärtskante.
       Dann wird als nächstes w mit einem rekursiven Aufruf besucht.
       w wird dann vor v in die PostOrder-Reihenfolge eingefügt.
- Damit steht v vor w in der invertierten PostOrderReihenfolge.

Tiefensuchwald mit Querkanten



PostOrder-Reihenfolge:
7, 6, 4, 5, 3, 1, 2