Graphen

Algorithmen und Datenstrukturen 1
VU 186.813, 4h, 6 ECTS, SS 2016
Letzte Änderung: 27. April 2016

Grundlegende Definitionen und Anwendungen

Graphen

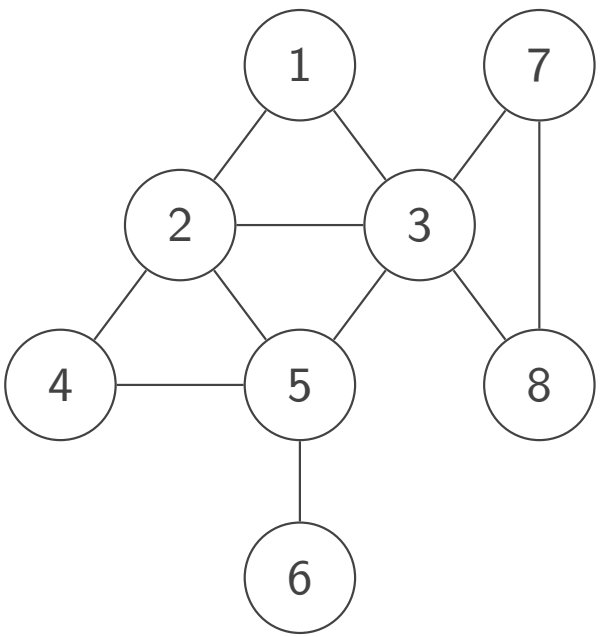
Graphen: Graphen sind ein wichtiges Werkzeug um Netzwerke, Zusammenhänge und Strukturen zu modellieren.

Beispiel: Wiener U-Bahn Linien

Ungerichtete Graphen

Ungerichteter Graph: $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der Kanten zwischen Paaren von Knoten (*edges*).
- Notation für Kante zwischen Knoten a und b: (a, b) bzw. (b, a).
- Alternativ wird auch $a - b$ bzw. $b - a$ verwendet.
- Parameter für Größen: $n = |V|$, $m = |E|$



$V = \{1,2,3,4,5,6,7,8\}$

$E = \{1\text{-}2,1\text{-}3,2\text{-}3,2\text{-}4,2\text{-}5,3\text{-}5,3\text{-}7,3\text{-}8,4\text{-}5,5\text{-}6,7\text{-}8\}$

$n = 8$

$m = 11$

Ungerichtete Graphen: Weitere Definitionen

Adjazent, inzident, Nachbarschaft: Sei $e = (u, v)$ eine Kante in E.

- u und v sind adjazent, d.h. u ist Nachbar von v und v ist Nachbar von u.
- v (bzw. u) und e sind inzident.
- $(u, v) = (v, u)$.

Knotengrad (*degree*): $deg(v)$ bezeichnet den Knotengrad des Knotens v.

- $deg(v)$ entspricht der Anzahl der zu v inzidenten Kanten.
- Es gilt: $\sum_{v \in V} deg(v) = 2 \cdot \mid E \mid$ (Handshaking-Lemma).

Ungerichtete Graphen: Weitere Definitionen

Grundlegende Definitionen:

- Mehrfachkante: Mehrere Kanten zwischen zwei Knoten.
- Schleife: Eine Kante, die einen Knoten mit sich selbst verbindet.

Schlichter Graph: Ein ungerichteter Graph ohne Mehrfachkanten und ohne Schleifen.

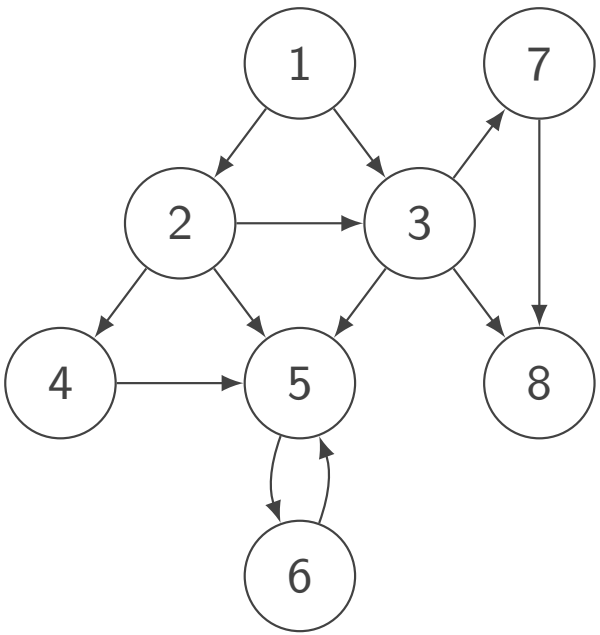
Hinweise:

- In dieser Vorlesung werden, wenn nicht anders verlautbart, schlichte Graphen betrachtet.
- Bei bestimmten Problemstellungen werden gewichtete Graphen verwendet, bei denen Knoten und/oder Kanten eine reelle Zahl zugeordnet bekommen.

Gerichtete Graphen

Gerichteter Graph (Digraph): $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der gerichtete Kanten (*arcs*) zwischen Paaren von Knoten.
- Notation für Kante von a zu b: (a, b) bzw. $a \to b$
- $(a, b) \neq (b, a)$



$V = \{1,2,3,4,5,6,7,8\}$

$E = \{1 \to 2, 1 \to 3, 2 \to 3, 2 \to 4, 2 \to 5, 3 \to 5,$
$3 \to 7, 3 \to 8, 4 \to 5, 5 \to 6, 6 \to 5, 7 \to 8\}$

$n = 8$

$m = 12$

Hinweis: Kanten in entgegengesetzter Richtung sind auch in schlichten Digraphen erlaubt.

Gerichtete Graphen: Weitere Definitionen

Eingangsknotengrad: $deg^-(v)$ ist die Anzahl der eingehenden inzidenten Kanten.

Ausgangsknotengrad: $deg^+(v)$ ist die Anzahl der ausgehenden inzidenten Kanten.

Es gilt: $deg(v) = deg^+(v) + deg^-(v)$.

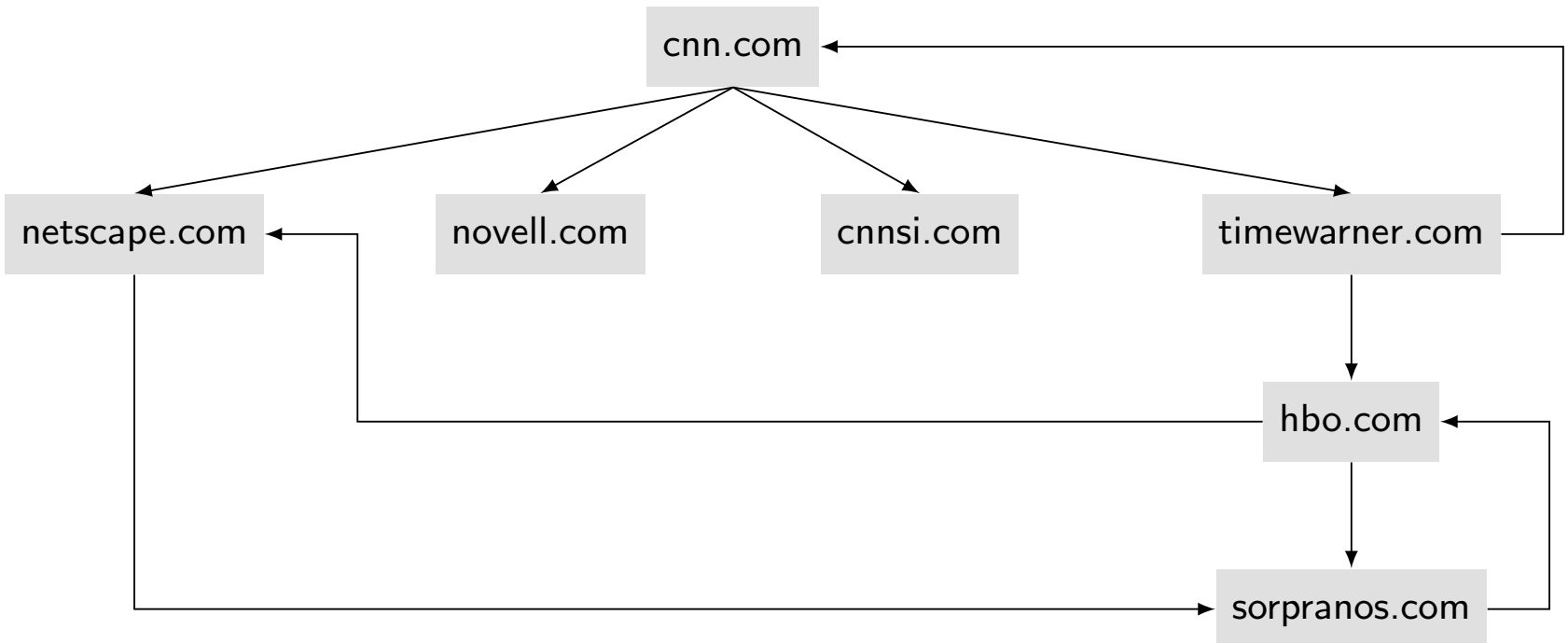
Einige Anwendungen von Graphen

Graph	*Knoten*	*Kanten*
Verkehr	Kreuzungen	Straßen
Netzwerke	Computer	Glasfaserkabel
World Wide Web	Webseiten	Hyperlinks
Sozialer Bereich	Personen	Beziehungen
Nahrungsnetz	Spezies	Räuber-Beute-Beziehung
Software	Funktionen	Funktionsaufrufe
Scheduling	Aufgaben	Ablaufeinschränkungen
elektronische Schaltungen	Gatter	Leitungen

World Wide Web

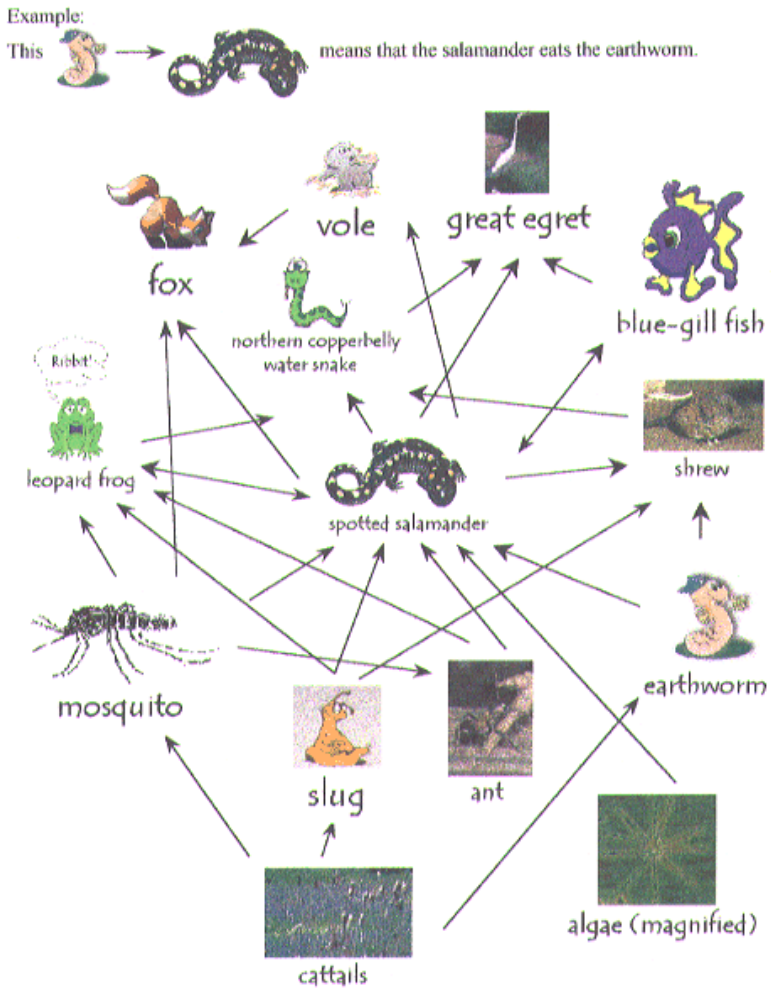
Web Graph:

- Knoten: Webseiten.
- Kante: Hyperlink von einer Seite zur anderen.



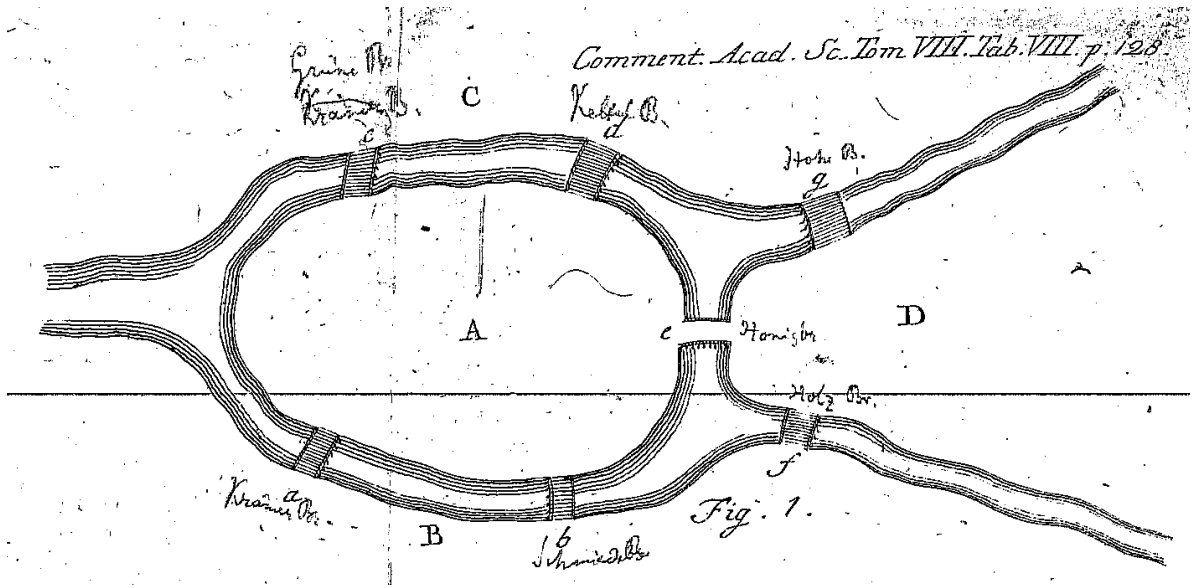
Ökologisches Nahrungsnetz

Nahrungsnetz als Graph: Knoten = Spezies, Kante = von der Beute zum Raubtier.



Königsberger Brückenproblem [Euler 1736]

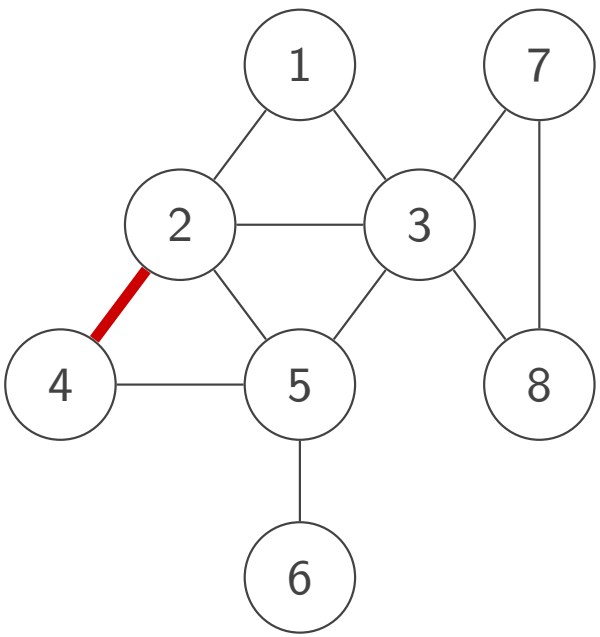
SOLVTIO PROBLEMATIS
AD
GEOMETRIAM SITVS
PERTINENTIS.
AVCTORE
Leonh. Eulero.



Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix: n-mal-n Matrix mit $A_{uv} = 1$ wenn (u, v) eine Kante ist.

- Knoten: 1,2,. . . ,n.
- Zwei Einträge für jede ungerichtete Kante.
- Platzbedarf in $\Theta(n^2)$.
- Überprüfen, ob (u, v) eine Kante ist, hat eine Laufzeit von $\Theta(1)$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(n^2)$.

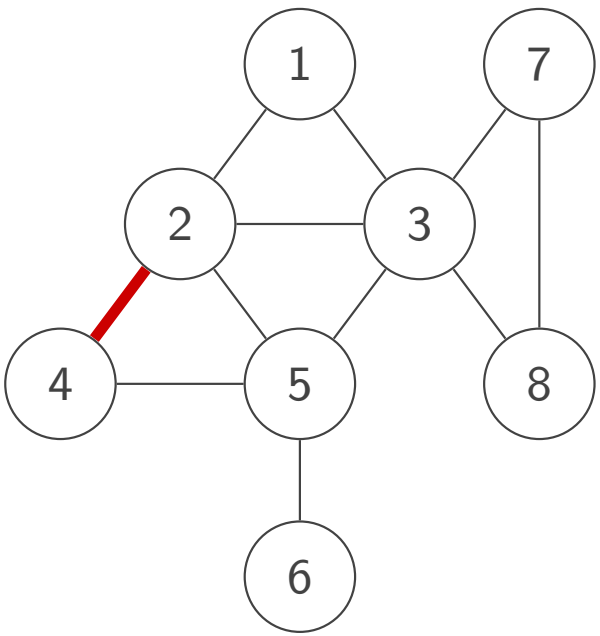


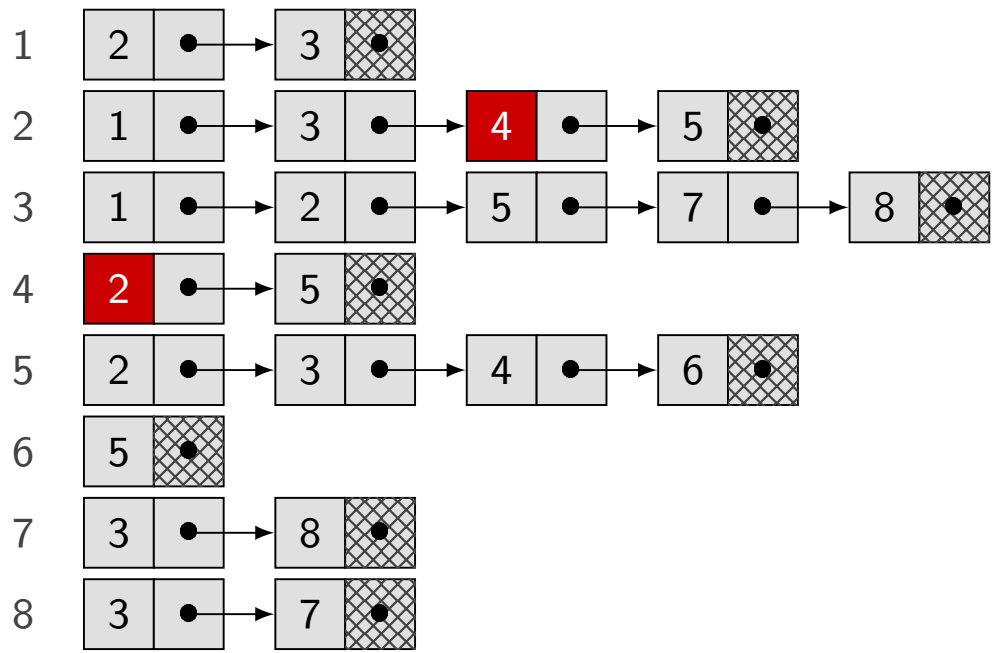
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	**1**	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	**1**	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Repräsentation von Graphen: Adjazenzlisten

Adjazenzlisten: Array von Listen. Index ist die Knotennummer.

- Knoten: 1,2,. . . ,n.
- Zwei Einträge für jede Kante.
- Platzbedarf in $\Theta(m+n)$.
- Überprüfen, ob (u,v) eine Kante ist, hat eine Laufzeit von $O(\deg(u))$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(m+n)$.





Adjazenzmatrix oder Adjazenzlisten

Kantenanzahl:

- Ein Graph kann bis zu $\frac{n(n-1)}{2} = \binom{n}{2} = \Theta(n^2)$ viele Kanten enthalten.
- Für so einen dichten Graphen sind die beiden Darstellungsformen (Adjazenzmatrix oder Adjazenzlisten) vergleichbar.

Praxis:

- Graphen, die sich aus Anwendungen ergeben, enthalten aber oft erheblich weniger Kanten.
- Typischerweise gilt dann $m = O(n)$.
- In diesem Fall ist die Darstellung mittels Adjazenzlisten günstiger.

Hinweis: Wenn wir sagen, dass ein Algorithmus auf Graphen in Linearzeit läuft, gehen wir von einer Darstellung mit Adjazenzlisten aus und betrachten eine Laufzeit von $O(n + m)$.

Pfade und Zusammenhang

Definition: Ein Pfad in einem ungerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \ldots, v_{k\text{-}1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine Kante in E verbunden ist. Die Länge des Pfades ist $k - 1$.

Hinweis: Wir sagen auch: Der Pfad geht von v_1 nach v_k und wir bezeichnen den Pfad als v_1-v_k-Pfad.

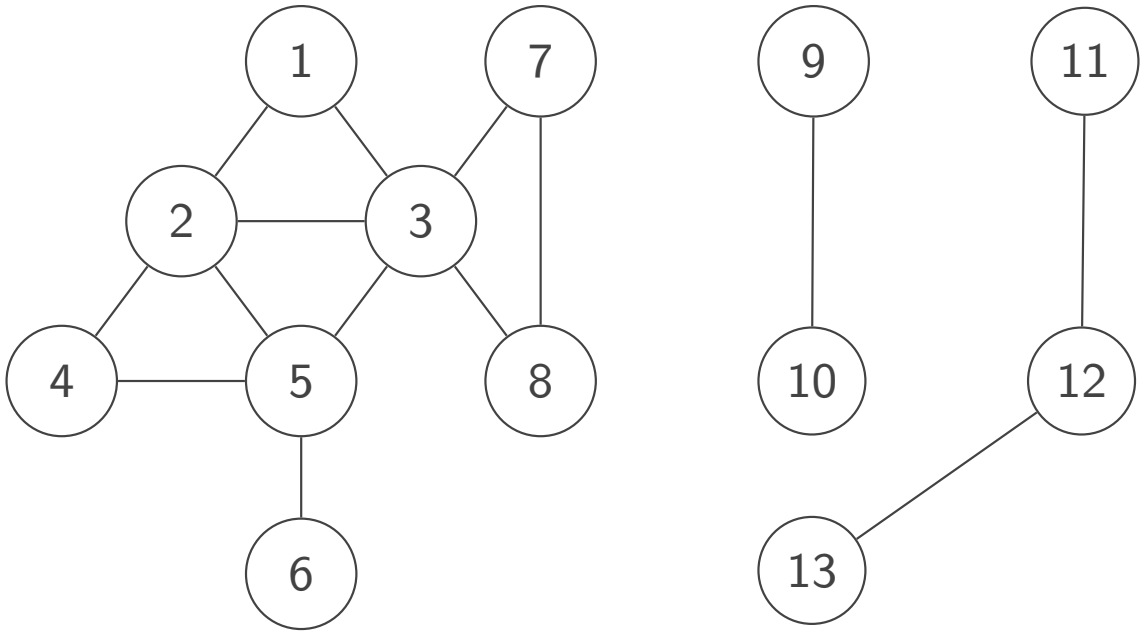
Definition: Knoten u ist von Knoten v in einem Graph G erreichbar, falls G einen u-v-Pfad enthält.

Definition: Ein ungerichteter Graph ist zusammenhängend, wenn jedes Paar von Knoten u und v von einander erreichbar ist.

Definition: Der kürzeste u-v-Pfad zwischen zwei Knoten u und v ist einfach (d.h. es unterscheiden sich alle Knoten).

Zusammenhang: Beispiel

Nicht zusammenhängender Graph:

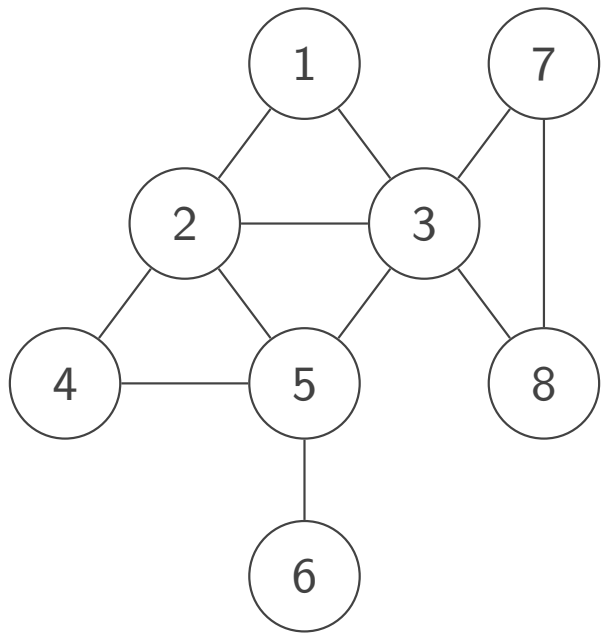


Nicht zusammenhängend: Es gibt zum Beispiel keinen Pfad vom Knoten 1 zu Knoten 10.

Beispiel für Zusammenhang: Die Knoten 1 bis 8 und ihre inzidenten Kanten bilden einen zusammenhängenden Graphen.

Kreis

Definition: Ein Kreis ist ein Pfad $v_1, v_2, \ldots, v_{k\text{-}1}, v_k$ in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k-1$ Knoten alle unterschiedlich sind. Die Länge des Kreises ist $k-1$.



Beispiel für Kreis: C = 1-2-4-5-3-1

Pfade und Kreise in gerichteten Graphen

Pfad: Ein Pfad in einem gerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \ldots, v_{k\text{-}1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine gerichtete Kante (v_i, v_{i+1}) in E verbunden ist.

Hierbei gilt:

- Der Pfad geht von einem Startknoten u zu einen Endknoten v (u-v-Pfad). Die Umkehrung muss aber nicht gelten.
- v kann von u aus erreicht werden, falls ein u-v-Pfad existiert.
- Kürzeste u-v-Pfade sind einfach (d.h. enthalten keinen Knoten doppelt).

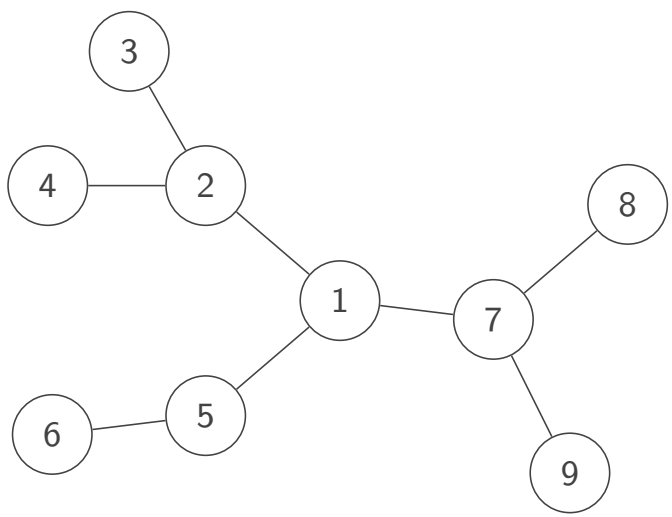
Kreis: Ein gerichteter Kreis ist ein Pfad $v_1, v_2, \ldots, v_{k\text{-}1}, v_k$ in dem $v_1 = v_k$, $k \geq 3$, und die ersten $k - 1$ Knoten alle unterschiedlich sind.

Bäume

Definition: Ein ungerichteter Graph ist ein Baum, wenn er zusammenhängend ist und keinen Kreis enthält.

Theorem: Sei G ein ungerichteter Graph mit n Knoten. Jeweils zwei der nachfolgenden Aussagen implizieren die dritte Aussage:

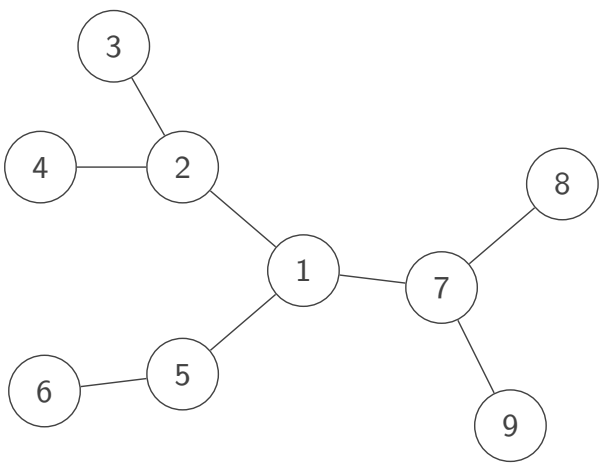
- G ist zusammenhängend.
- G enthält keinen Kreis.
- G hat n-1 Kanten.



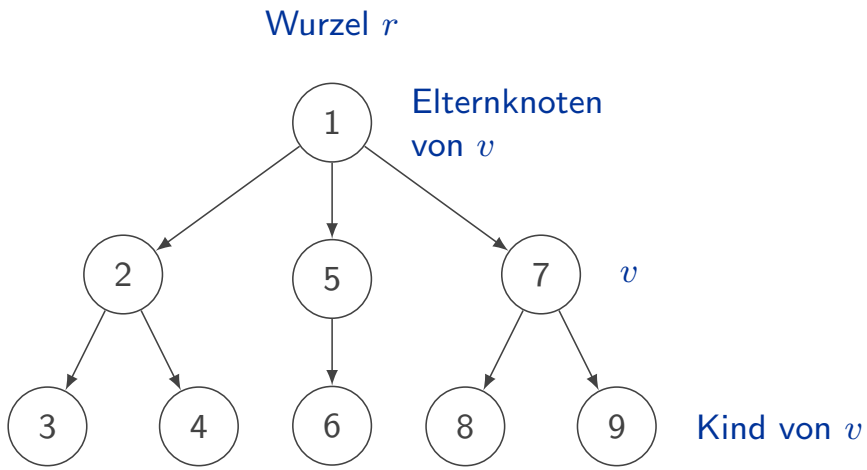
Wurzelbaum (*rooted tree, arborescence*)

Wurzelbaum: Gegeben sei ein Baum T. Wähle einen Wurzelknoten r und gib jeder Kante eine Richtung von r weg.

Bedeutung: Modelliert hierarchische Strukturen.



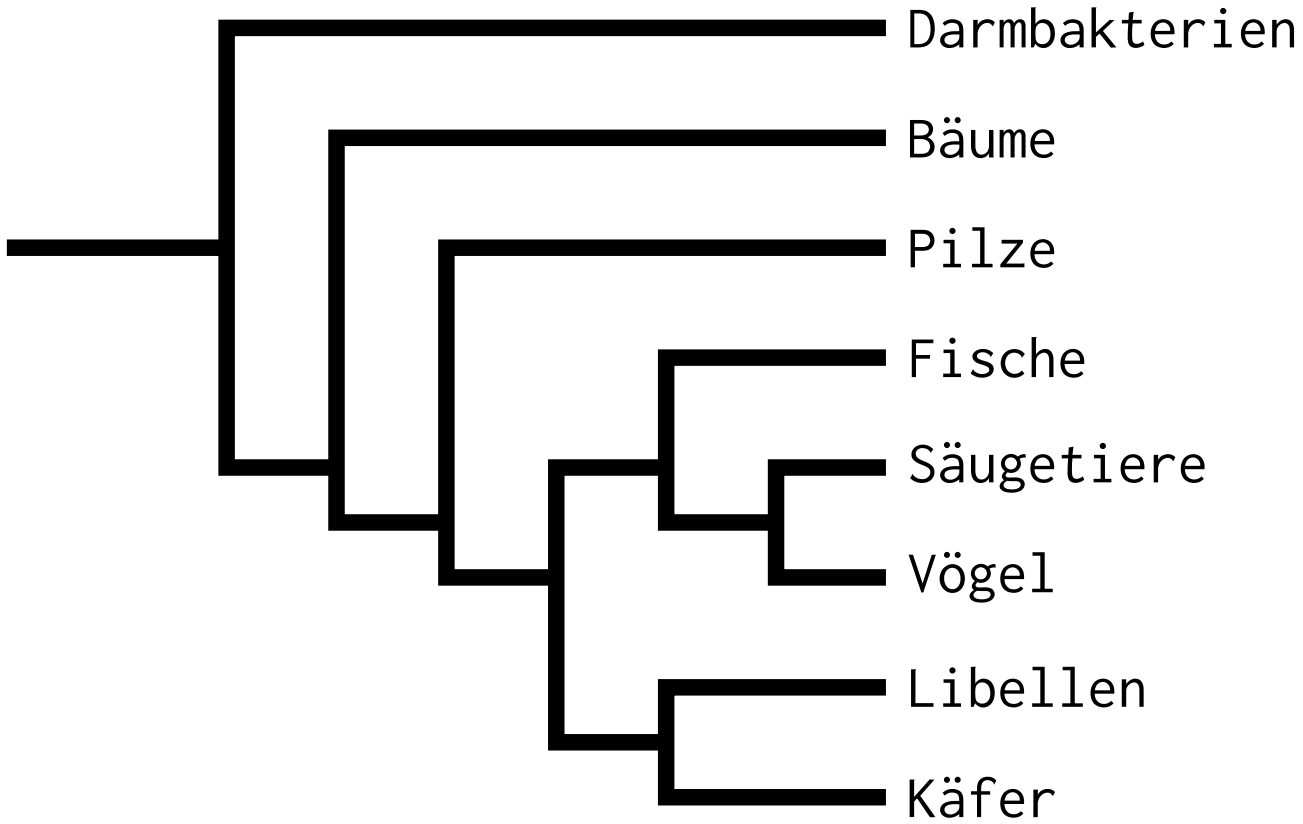
ein Baum



Ein entsprechender Wurzelbaum mit Wurzelknoten 1

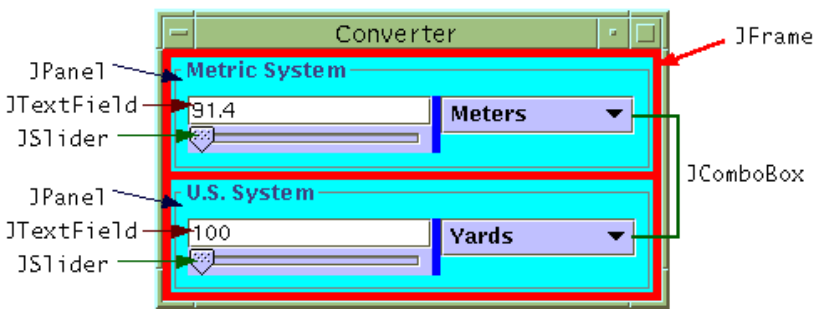
Phylogenetischer Baum

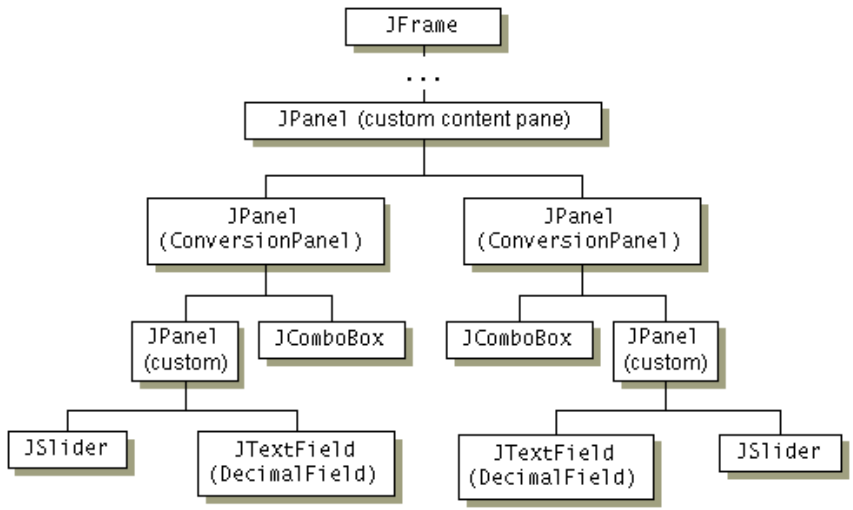
Phylogenetischer Baum: Beschreibt die evolutionären Beziehungen zwischen verschiedenen Arten.



GUI-Hierarchien

GUI-Hierarchien: Beschreiben die Organisation von GUI-Komponenten.





Durchmusterung von Graphen (*Graph Traversal*)

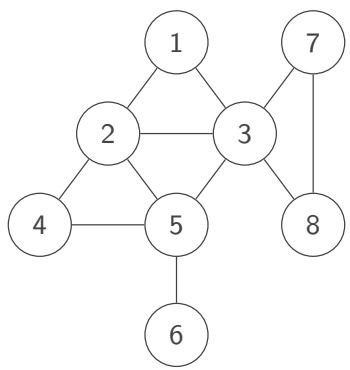
Zusammenhangsproblem in ungerichteten Graphen

s-t Zusammenhangsproblem: Existiert zwischen zwei gegebenen Knoten s und t ein Pfad?

s-t kürzester Pfad: Wie viele Kanten hat der kürzeste Pfad zwischen s und t ($=$ Distanz zwischen s und t)?

Anwendungen:

- Facebook.
- Labyrinth durchschreiten.
- Kevin-Bacon-Zahl.
- Die kleinste Anzahl an Hops (kürzester Pfad) zwischen zwei Knoten in einem Kommunikationsnetzwerk.

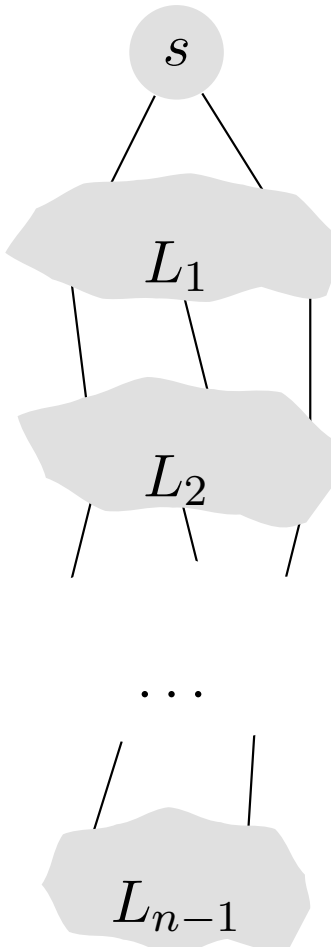


Breitensuche (*Breadth First Search, BFS*)

BFS Ansatz: Untersuche alle Knoten von einem Startknoten s ausgehend in alle möglichen Richtungen, wobei die Knoten Ebene für Ebene abgearbeitet werden.

BFS Algorithmus:

- $L_0 = \{s\}$.
- $L_1 =$ alle Nachbarn von L_0.
- $L_2 =$ alle Knoten, die nicht zu L_0 oder zu L_1 gehören und die über eine Kante mit einem Knoten in L_1 verbunden sind.
- $L_{i+1} =$ alle Knoten, die nicht zu einer vorherigen Ebene gehören und die über eine Kante mit einem Knoten in L_i verbunden sind.



Breitensuche

Theorem: Für jede Ebene $i = 0, 1, \ldots$ gilt, dass L_i alle Knoten mit Distanz i von s beinhaltet. Es existiert ein Pfad von s zu einem Knoten t dann und nur dann, wenn t in einer Ebene aufscheint.

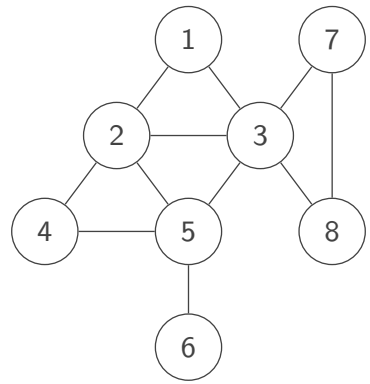
BFS-Baum

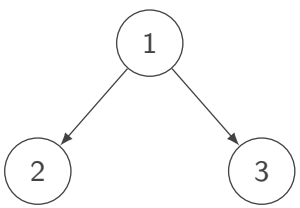
BFS-Baum: Breitensuche erzeugt einen Baum (BFS-Baum), dessen Wurzel ein Startknoten s ist und der alle von s erreichbaren Knoten beinhaltet.

Aufbau: Man startet bei s. Wird nun ein Knoten u in der Ebene L_j gefunden, ist er zu mindestens einem Knoten v der Ebene L_{j-1} benachbart. Ein solcher Knoten v wird ausgewählt und zum Elternknoten von u im BFS-Baum gemacht.

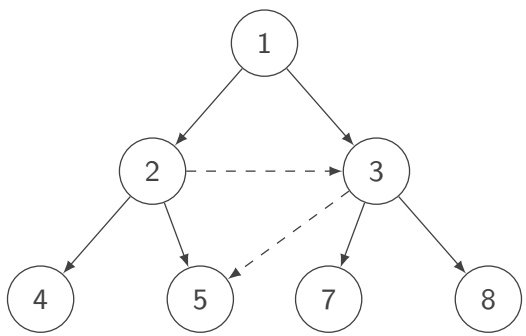
Breitensuche

Eigenschaft: Sei T ein BFS-Baum von $G = (V, E)$ und sei (x, y) eine Kante von G. Dann können sich die Ebenen von x und y höchstens um 1 unterscheiden.

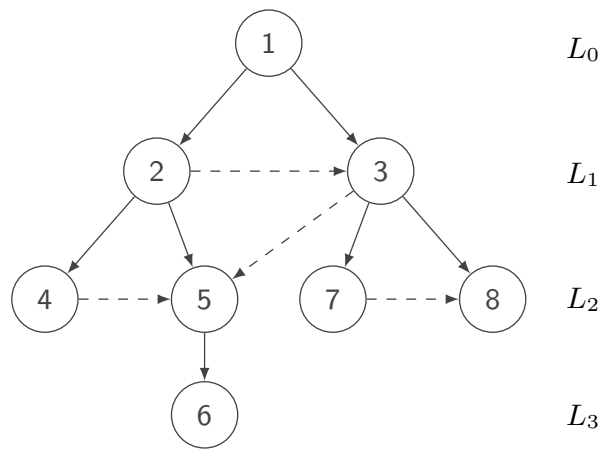




(a)



(b)



(c)

Breitensuche: Implementierung mit einer Queue

Implementierung: Array Discovered, Queue Q, Graph $G = (V, E)$, Startknoten s.

```
BFS(G,s):
Discovered[s] ← true
Discovered[v] ← false für alle anderen Knoten v ∈ V
Q ← s
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    Führe Operation auf u aus (z.B. Ausgabe)
    foreach Kante (u,v) inzident zu u
        if  !Discovered[v]
            Discovered[v] ← true
            Füge v zu Q hinzu
```

Breitensuche: Analyse

Theorem: BFS hat eine Laufzeit von $O(m + n)$.

Laufzeit: Für die Laufzeitabschätzung müssen wir drei Teile betrachten:

- Initialisierung vor der while-Schleife
- while-Schleife
- foreach-Schleife

Breitensuche: Analyse

Initialisierung vor der while-Schleife:

- Jeder Knoten wird genau einmal betrachtet
- Pro Knoten können die Anweisungen in konstanter Zeit ausgeführt werden.
- Daher benötigt die Initialisierung $O(n)$ Zeit.

while-Schleife:

- Jeder Knoten u wird höchstens einmal in Q gegeben, denn nachdem er das erste mal in Q gegeben wird, wird ja Discovered[u]=true gesetzt.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

Breitensuche: Analyse

foreach-Schleife:

- Sei u der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten v in der Adjazenzliste von u betrachtet.
- Das sind genau $deg(u)$ viele. Daher wird die Schleife $deg(u)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} deg(u))$.
- Da $\sum_{u \in V} deg(u) = 2m$, liegt die Laufzeit in $O(n + m)$.

Ergebnis: Etwas vereinfacht können wir die Analyse zusammenfassen:

- BFS betrachtet jeden Knoten einmal und jede Kante höchstens zweimal.
- Daher ergibt sich eine Laufzeit von $O(n + m)$.

Tiefensuche (*Depth First Search, DFS*)

DFS Ansatz: Von einem besuchten Knoten u wird zuerst immer zu einem weiteren noch nicht besuchten Nachbarknoten gegangen (DFS-Aufruf), bevor die weiteren Nachbarknoten von u besucht werden.
DFS Algorithmus: Startknoten s, globales Array Discovered, Graph $G = (V, E)$.

```
DFS(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
DFS1(G,s)

DFS1(G,u):
Discovered[u] ← true
Führe Operation auf u aus (z.B. Ausgabe)
foreach Kante (u,v) inzident zu u
    if  !Discovered[v]
        DFS1(G,v)
```

Tiefensuche: Analyse

Theorem: DFS hat eine Laufzeit von $O(m+n)$.

Laufzeit: Für Laufzeitabschätzung betrachten wir:

- Initialisierung
- foreach-Schleife

Initialisierung:

- Initialisierung vor dem Aufruf von DFS1 in $O(n)$ Zeit.
- DFS1(G,u) wird für jeden Knoten u höchstens einmal aufgerufen.

Tiefensuche: Analyse

foreach-Schleife in DFS1(G,u):

- Es werden alle Knoten v in der Adjazenzliste von u betrachtet. Das sind genau $deg(u)$ viele.
- Daher wird die Schleife $deg(u)$ mal durchlaufen.
- Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit (außer dem rekursiven Aufruf DFS1(G,v), aber dessen Laufzeit wird ja in der Analyse für den Knoten v berücksichtigt).

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} deg(u))$.
- Da $\sum_{u \in V} deg(u) = 2m$, erhalten wir eine Laufzeit von $O(n + m)$.

Tiefensuche: Analyse

Ergebnis: Etwas vereinfacht können wir wie beim BFS auch argumentieren:

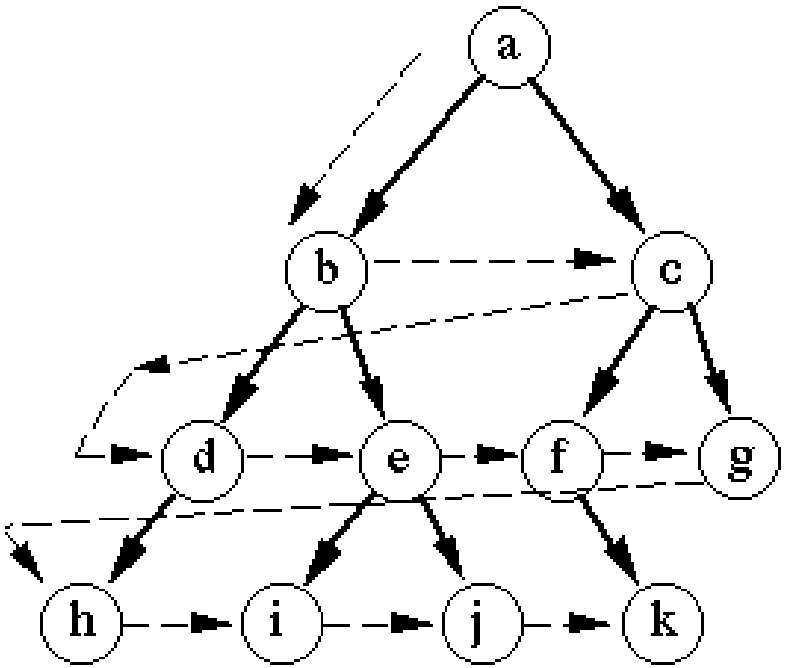
- DFS betrachtet jeden Knoten einmal und jede Kante höchstens zweimal.
- Daher ergibt sich eine Laufzeit von $O(n + m)$.

Durchmusterung: Durchmusterung unterscheidet sich von der bei BFS.

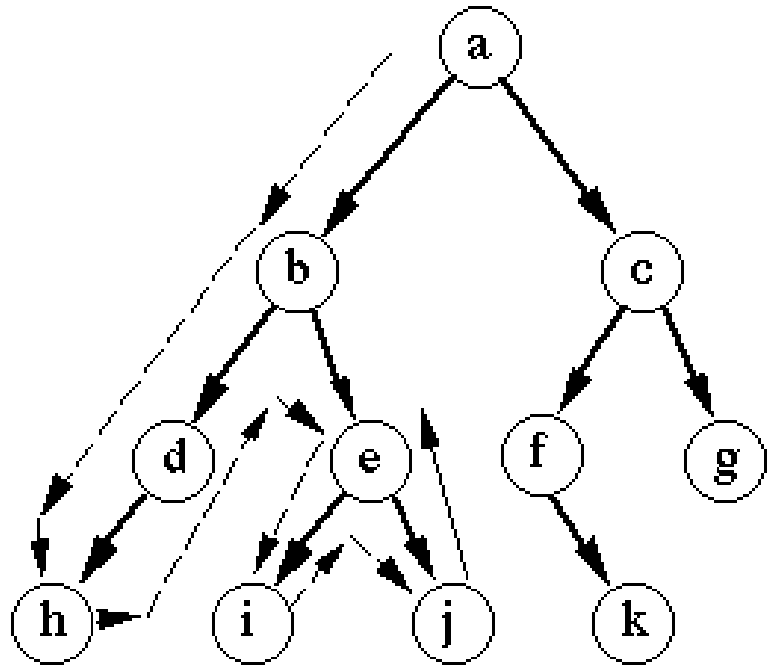
- Es wird zunächst versucht, möglichst weit vom Startknoten weg zu kommen.
- Gibt es in der Nachbarschaft keine möglichen Knoten, dann wird durch den rekursiven Aufstieg bis zu einer möglichen Verzweigung zurückgegangen (Backtracking).

Beispiel

Vergleich: Tiefensuche und Breitensuche im Vergleich.



Breadth-first search



Depth-first search

Durchmusterung:

- Breitensuche: a, b, c, d, e, f, g, ,h ,i, j, k
- Tiefensuche: a, b, d, h, e, i, j, c, f, k, g

Zusammenhangskomponente

Zusammenhang (Wiederholung): Ein ungerichteter Graph ist zusammenhängend, wenn für jedes Paar von Knoten u und v ein Pfad zwischen u und v existiert.

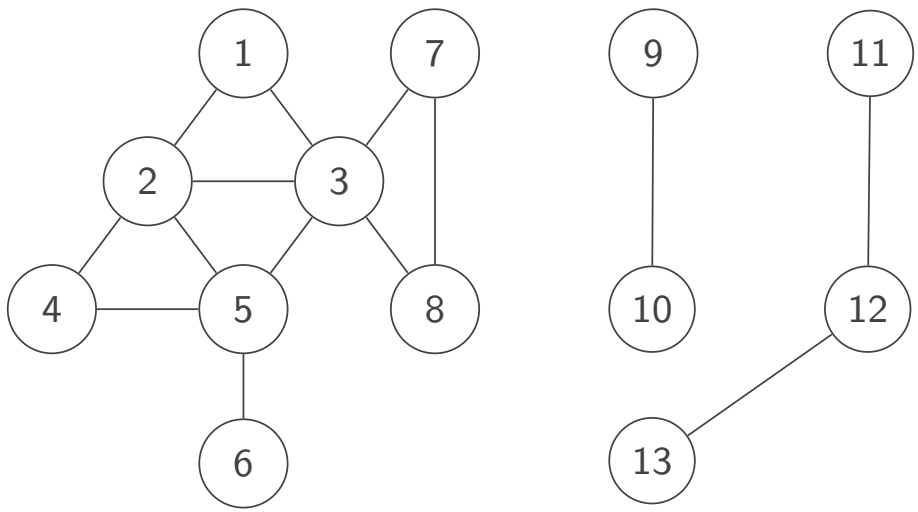
Nicht zusammenhängend: Gibt es zwischen einem Paar von Knoten keinen Pfad, dann ist der Graph nicht zusammenhängend.

Teilgraph: Ein Graph $G_1 = (V_1, E_1)$ heißt Teilgraph von $G_2 = (V_2, E_2)$, wenn seine Knotenmenge V_1 Teilmenge von V_2 und seine Kantenmenge E_1 Teilmenge von E_2 ist, also $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$ gilt.

Zusammenhangskomponente: Einen maximalen zusammenhängenden Teilgraphen eines beliebigen Graphen nennt man Zusammenhangskomponente. Ein nicht zusammenhängender Graph zerfällt in seine Zusammenhangskomponenten.

Zusammenhangskomponente

Beispiel: Ein nicht zusammenhängender Graph mit 3 Zusammenhangskomponenten.



Zusammenhangskomponente

Zusammenhangskomponente: Finde alle Knoten, die von s aus erreicht werden können.

Lösung:

- Rufe DFS(G,u) oder BFS(G,u) auf.
- Ein Knoten u ist von s genau dann erreichbar, wenn Discovered[u]=true ist.

Zusammenhangskomponenten zählen

DFSNUM Algorithmus: Startknoten s, globales Array Discovered, Graph $G = (V, E)$.

```
DFSNUM(G):
Discovered[v] ← false für alle Knoten v ∈ V
i ← 0
foreach Knoten v ∈ V
    if Discovered[v] = false
        i ← i + 1
        DFS1(G, v)
return i
```

Zusammenhangskomponenten zählen

Laufzeit: Die Laufzeit liegt in $O(n+m)$.

Analyse:

- Sei $G=(V,E)$ der gegebene Graph und $G_1=(V_1,E_1),\ldots,G_r=(V_r,E_r)$ seine Zusammenhangskomponenten. Sei $|V|=n$ und $|E|=m$, sowie $|V_i|=n_i$ und $|E_i|=m_i$, für $1\leq i\leq r$.
- Klarerweise gilt $n=n_1+\ldots+n_r$ und $m=m_1+\ldots+m_r$.
- Für jede einzelne Zusammenhangskomponente G_i $(1\leq i\leq r)$ führt der Algorithmus eine Tiefensuche aus. Dies hat eine Laufzeit von $O(n_i+m_i)$.
- Die Initialisierung benötigt $O(n)$ Zeit.
- Insgesamt erhalten wir eine Laufzeit von $O(n+\sum_{i=1}^{r}(n_i+m_i))=O(2n+m)=O(n+m)$.

Zusammenhang in gerichteten Graphen

Suche in gerichteten Graphen

Gerichtete Erreichbarkeit: Gegeben sei ein Knoten s, finde alle Knoten, die von s aus erreicht werden können.

Gerichteter kürzester s-t Pfad: Gegeben seien zwei Knoten s und t, ermittle den kürzesten Pfad von s nach t.

Suche in gerichteten Graphen: BFS und DFS können auch auf gerichtete Graphen angewendet werden.

Beispiel Webcrawler: Starte von einer Webseite s. Finde alle Webseiten, die von s aus direkt oder indirekt verlinkt sind.

Starker Zusammenhang

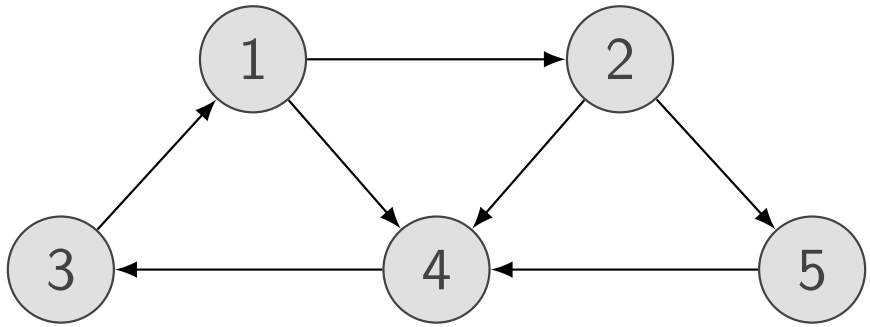
Definition: Knoten u und v in einem gerichteten Graphen sind gegenseitig erreichbar, wenn es einen Pfad von u zu v und einen Pfad von v zu u gibt.

Definition: Ein gerichteter Graph ist stark zusammenhängend, wenn jedes Paar von Knoten gegenseitig erreichbar ist.

Hinweis: Ein gerichteter Graph heißt schwach zusammenhängend, falls der zugehörige ungerichtete Graph (also der Graph, der entsteht, wenn man jede gerichtete Kante durch eine ungerichtete Kante ersetzt) zusammenhängend ist.

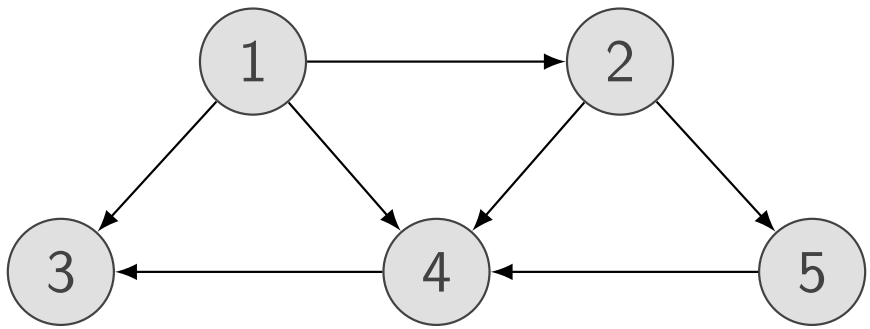
Starker Zusammenhang: Beispiel

Stark zusammenhängend:



Nicht stark zusammenhängend (aber schwach zusammenhängend):

Knoten 1 kann von keinem anderen Knoten erreicht werden, vom Knoten 3 führt kein Pfad weg.

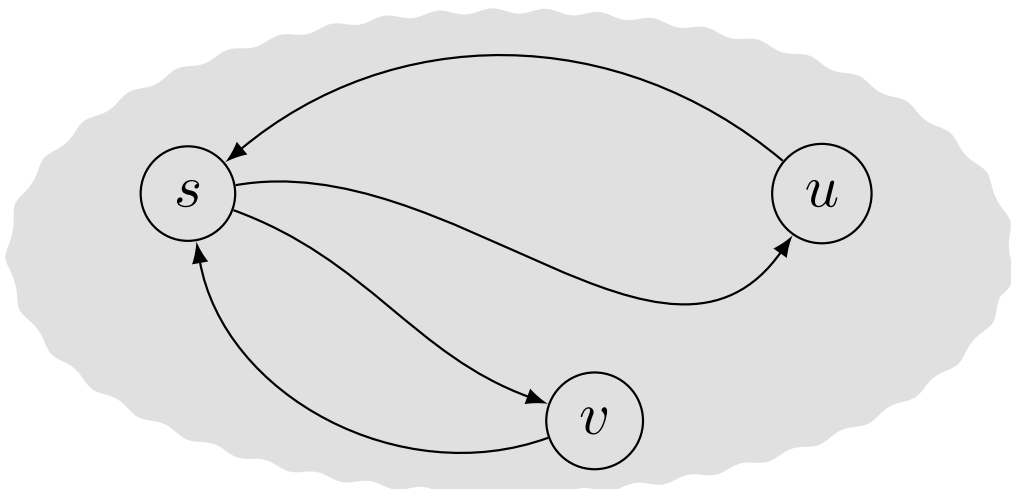


Starker Zusammenhang

Lemma: Sei s ein beliebiger Knoten in einem gerichteten Graphen G. G ist stark zusammenhängend dann und nur dann, wenn jeder Knoten von s aus und s von jedem Knoten aus erreicht werden kann.

Beweis: $\Rightarrow$ Folgt aus der Definition.
Beweis: $\Leftarrow$ Pfad von u zu v: verbinde u-s Pfad mit s-v Pfad.
Pfad von v zu u: verbinde v-s Pfad mit s-u Pfad. □

□ *auch ok, wenn Pfade überlappen*



Starker Zusammenhang: Algorithmus

Theorem: Laufzeit für die Überprüfung, ob G stark zusammenhängend ist, liegt in $O(m + n)$.

Beweis:

- Wähle einen beliebigen Knoten s.
- Führe BFS mit Startknoten s in G aus.
- Führe BFS mit Startknoten s in G^{rev}.
- Gib true zurück dann und nur dann, wenn alle Knoten in beiden BFS-Ausführungen erreicht werden können.
- Korrektheit folgt unmittelbar aus dem vorherigen Lemma. □

□ *umgekehrte Orientierung von jeder Kante in* G

DAGs und Topologische Sortierung

Gerichteter azyklischer Graph (*Directed Acyclic Graph, DAG*)

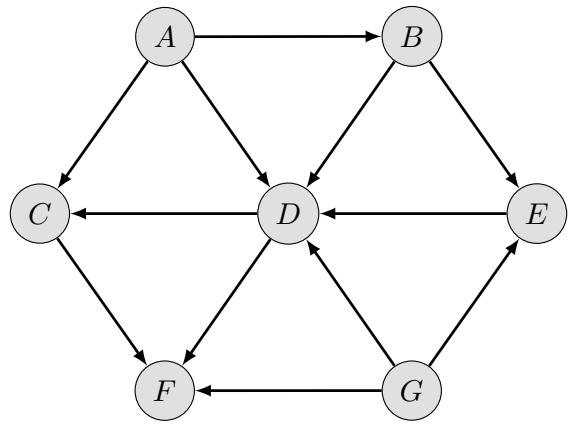
Definition: Ein DAG ist ein gerichteter Graph, der keine gerichteten Kreise enthält.

Beispiel: Reihenfolgebeschränkung: Kante (u, v) bedeutet, Aufgabe u muss vor Aufgabe v bearbeitet werden.

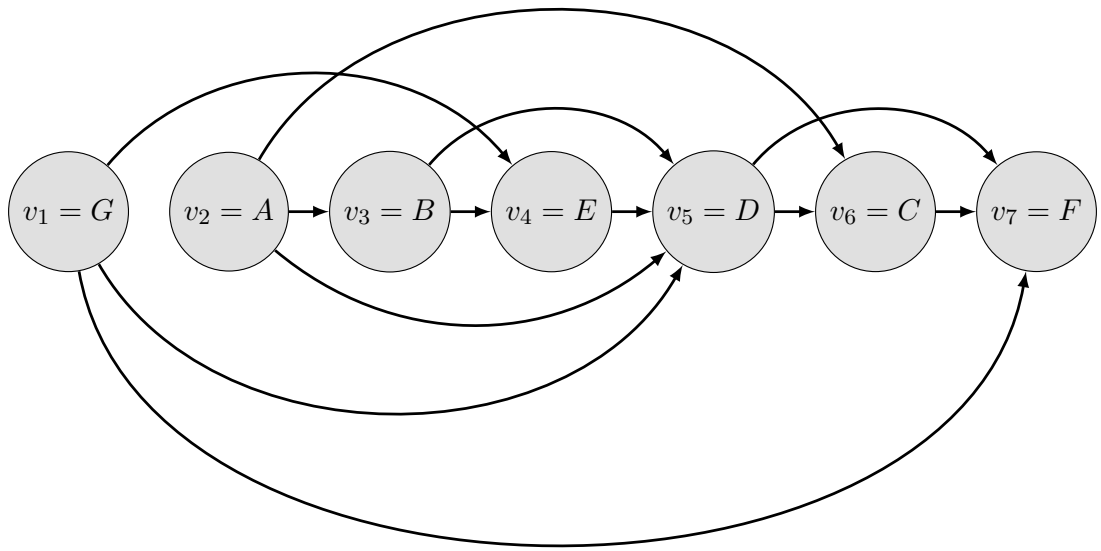
Definition: Eine topologische Sortierung eines gerichteten Graphen $G = (V, E)$ ist eine lineare Ordnung seiner Knoten, bezeichnet mit $v_1, v_2, \ldots, v_n$, sodass für jede Kante (v_i, v_j) gilt, dass $i < j$.

Topologische Sortierung: Beispiel

Ein DAG:



Eine topologische Sortierung:



Reihenfolgebeschränkung

Reihenfolgebeschränkung: Kante (u, v) bedeutet, dass Aufgabe u vor v bearbeitet werden muss.

Anwendungen:

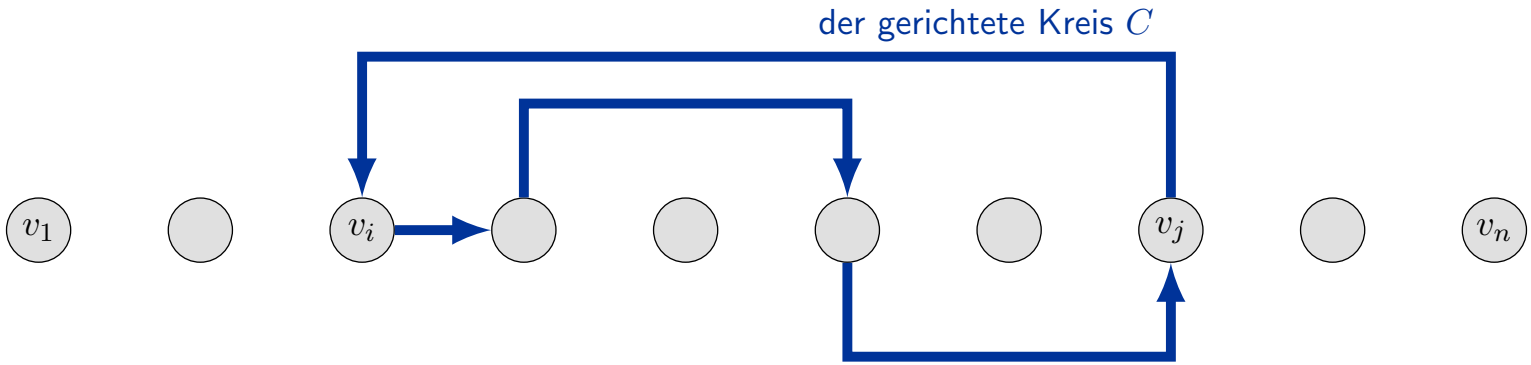
- Voraussetzungen bei Kursen: Kurs u muss vor Kurs v absolviert werden.
- Übersetzung: Modul u muss vor Modul v übersetzt werden.
- Pipeline von Prozessen: Ausgabe von Prozess u wird benötigt, um die Eingabe von v zu bestimmen.

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.

Beweis: (durch Widerspruch)

- Wir nehmen an, dass G eine topologische Sortierung $v_1, \ldots, v_n$ und auch einen gerichteten Kreis C besitzt.
- Sei v_i der Knoten mit dem kleinsten Index in C und sei v_j der Knoten direkt vor v_i in C; daher gibt es die Kante (v_j, v_i).
- Durch die Wahl von i gilt, dass $i < j$.
- Andererseits, da (v_j, v_i) eine Kante ist und $v_1, \ldots, v_n$ eine topologische Sortierung ist, müsste eigentlich $j < i$ sein. Widerspruch. □



die angenommene topologische Sortierung: $v_1, \ldots, v_n$

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.

Frage: Hat jeder DAG eine topologische Sortierung?

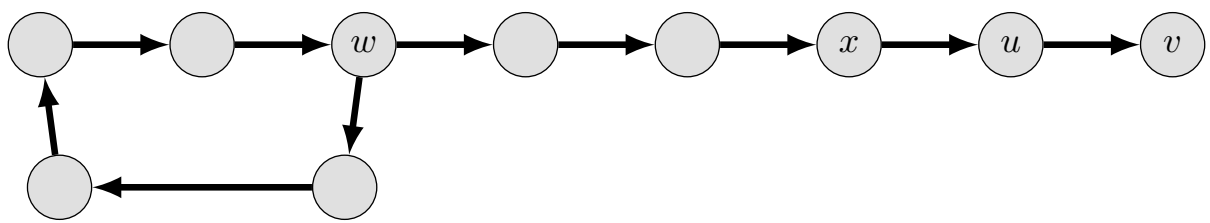
Frage: Wenn ja, wie berechnen wir diese?

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G einen Knoten ohne eingehende Kanten.

Beweis: (durch Widerspruch)

- Wir nehmen an, G ist ein DAG und jeder Knoten hat zumindest eine eingehende Kante.
- Wähle einen beliebigen Knoten v und folge den Kanten von v aus rückwärts. Da v zumindest eine eingehende Kante (u, v) besitzt, können wir rückwärts zu u gelangen.
- Da u zumindest eine eingehende Kante (x, u) hat, können wir rückwärts zu x gelangen.
- Das wird so oft wiederholt, bis man einen Knoten w zweimal besucht.
- Sei C die Sequenz von Knoten die zwischen zwei Besuchen von w durchlaufen wurde. C ist ein Kreis. □



Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine topologische Sortierung.

Beweis: (Induktion auf n)

- Induktionsanfang: wahr wenn $n = 1$.
- Es sei ein DAG mit $n > 1$ Knoten gegeben. Finde einen Knoten v ohne eingehende Kanten.
- Erzeuge $G - \{v\}$, d.h. den Graphen ohne v und ohne alle zu v inzidenten Kanten.
- $G - \{v\}$ ist ein DAG, da das Löschen von v keinen Kreis erzeugen kann.
- Durch die Induktionsbehauptung hat $G - \{v\}$ eine topologische Sortierung.
- Setze v an die erste Stelle in der topologischen Sortierung. Hänge dann Knoten aus G-$\{v\}$ in topologischer Sortierung an. Das ist erlaubt, da v keine eingehenden Kanten hat. □

Topologische Sortierung

Algorithmus: Löschen von Knoten wird mittels Hilfsarray count simuliert. Es wird zusätzlich eine Liste L verwendet.

```
foreach v ∈ V
    count[v] ← 0
foreach v ∈ V
    foreach Kante (v, w) ∈ E
        count[w] ← count[w]+1
foreach v ∈ V
    if count[v] = 0
        Gib v zur Liste L am Anfang hinzu
while L ist nicht leer
    Sei v erstes Element in L, lösche v aus L
    Gib v aus
    foreach Kante (v, w) ∈ E
        count[w] ← count[w]−1
        if count[w] = 0
            Gib w zur Liste L am Anfang hinzu
```

Topologische Sortierung: Laufzeit

Theorem: Algorithmus findet eine topologische Sortierung in $O(n+m)$.

Laufzeit: Dazu betrachten wir die folgenden Teile:

- Initialisierung
  - Erste foreach-Schleife für count.
  - Zwei verschachtelte foreach-Schleifen.
  - Dritte foreach-Schleife für Generierung der Liste.
- while-Schleife (mit foreach-Schleife).

Initialisierung:

- Die erste foreach-Schleife für die Initialisierung von count benötigt $O(n)$ Zeit.
- Bei den verschachtelten foreach-Schleifen wird die innere foreach-Schleife für jeden Knoten v genau $deg^+(v)$ mal ausgeführt. Daher benötigt man dafür $O(n+m)$ Zeit.
- Die Generierung der Liste L durch die dritte foreach-Schleife benötigt $O(n)$ Zeit.
- Daher benötigt die Initialisierung $O(n+m)$ Zeit.

Topologische Sortierung: Analyse

while-Schleife:

- Jeder Knoten v wird höchstens einmal aus L entnommen.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

foreach-Schleife:

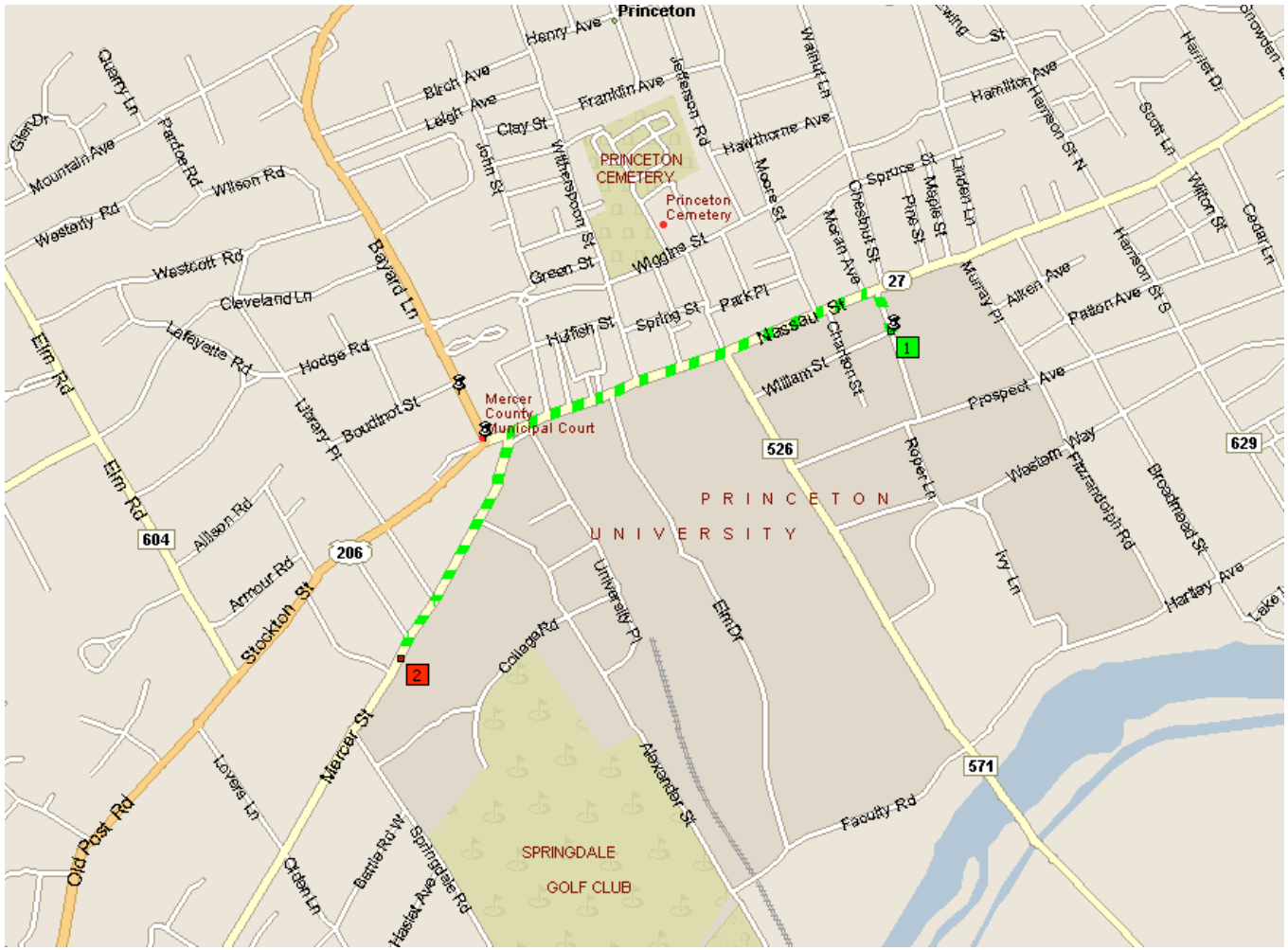
- Sei v der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten w in der Adjazenzliste von v betrachtet.
- Das sind genau $deg^+(v)$ viele. Daher wird die Schleife $deg^+(v)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.
- Jeder Knoten w wird höchstens einmal in L eingefügt.

Topologische Sortierung: Analyse

Gesamt:

- Initialisierung liegt in $O(n + m)$
- while-Schleife liegt in $O(n + m)$
- Daher liegt auch die gesamte Laufzeit in $O(n + m)$

Kürzeste Pfade in einem gewichteten Graphen



Kürzester Pfad vom Informatikinstitut in Princeton zu Einsteins Haus.

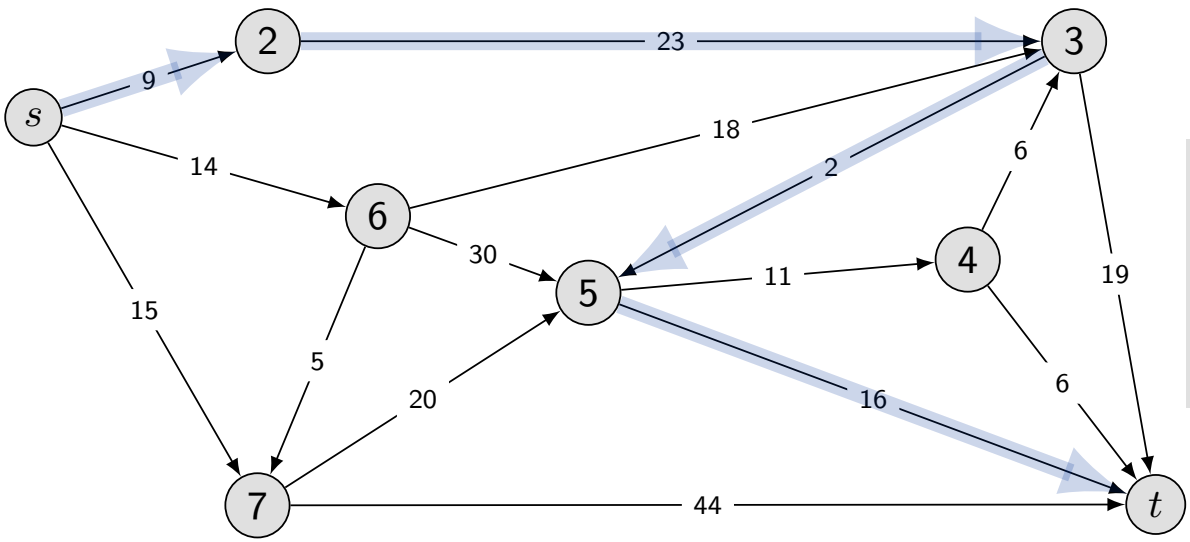
Kürzester Pfad (*Shortest Path Problem*)

Netzwerk für kürzesten Pfad:

- Gerichteter Graph $G = (V, E)$.
- Start s, Ziel t.
- Länge $\ell_e \geq 0$ ist die Länge der Kante e (Gewicht).

Kürzester Pfad: Finde kürzesten gerichteten Pfad von s nach t.

- *Kosten eines Pfades = Summe der Kosten aller Kanten in dem Pfad*



Kosten des Pfades s-2-3-5-t
$= 9 + 23 + 2 + 16$
$= 50$.

Dijkstra 1959

Numerische Mathematik 1, 269—271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. DIJKSTRA

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Algorithmus von Dijkstra

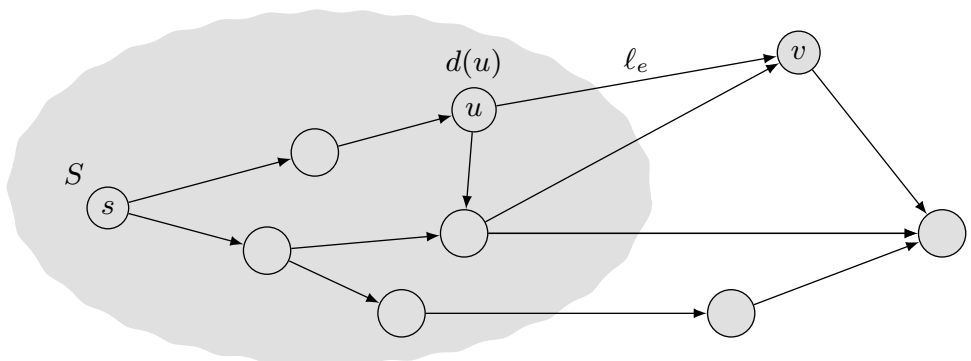
Algorithmus von Dijkstra:

- Verwalte eine Menge von untersuchten Knoten S, für die wir die kürzeste Distanz $d(u)$ von s zu u ermittelt haben.
- Initialisiere $S = \{s\}$, $d(s) = 0$.
- Wähle wiederholt einen nicht untersuchten Knoten v, für den gilt:

$$\pi(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e,$$

Füge v zu S hinzu und setze $d(v) = \pi(v)$.

Kürzester Pfad zu einem u in einem untersuchten Teil des Graphen, gefolgt von einer einzigen Kante (u, v).



Algorithmus von Dijkstra

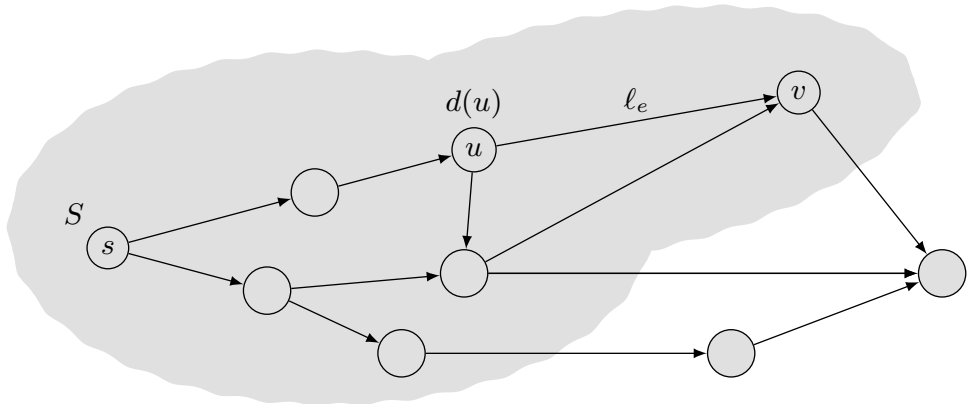
Algorithmus von Dijkstra:

- Verwalte eine Menge von untersuchten Knoten S, für die wir die kürzeste Distanz $d(u)$ von s zu u ermittelt haben.
- Initialisiere $S = \{s\}$, $d(s) = 0$.
- Wähle wiederholt einen nicht untersuchten Knoten v, für den gilt:

$$\pi(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e,$$

Füge v zu S hinzu und setze $d(v) = \pi(v)$.

- *Kürzester Pfad zu einem u in einem untersuchten Teil des Graphen, gefolgt von einer einzigen Kante (u, v).*



Algorithmus von Dijkstra: Korrektheitsbeweis

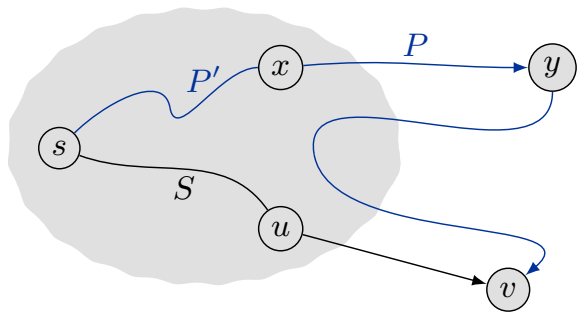
Invariante: Für jeden Knoten $u \in S$, ist $d(u)$ die Länge des kürzesten s-u Pfades.

Beweis: (Durch Induktion auf $|S|$)

Induktionsanfang: $|S| = 1$ ist trivial.

Induktionsbehauptung: Angenommen, wahr für $|S| = k \geq 1$.

- Sei v der nächste zu S hinzugefügte Knoten und sei u-v die gewählte Kante.
- Der kürzeste s-u Pfad plus (u, v) ist ein s-v Pfad der Länge $\pi(v)$.
- Betrachte einen beliebigen s-v Pfad P. Wir werden zeigen, dass er nicht kürzer als $\pi(v)$ ist.
- Sei x-y die erste Kante in P die S verlässt und sei P' der Teilpfad zu x.
- P ist schon zu lange, wenn er S verlässt.



$$\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)$$

□ *Nicht negative Gewichte* □ *Induktionsbehauptung* □ *Definition von* $\pi(y)$ □ *Dijkstra wählt* v *anstatt* y

Dijkstra-Algorithmus: Implementierung

Für jeden nicht untersuchten Knoten wird explizit $\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$ verwaltet.

- Der nächste zu untersuchende Knoten = Knoten mit minimalem $\pi(v)$.
- Wenn v untersucht wird, dann wird für jede inzidente Kante $e = (v, w)$ berechnet

$$\pi(w) = \min\{\pi(w), \pi(v) + \ell_e\}.$$

Effiziente Implementierung: Verwalte eine Vorrangwarteschlange (*priority queue*) von nicht untersuchten Knoten, geordnet nach $\pi(v)$.

Operation	Dijkstra	Array	Priority Queue
Einfügen	n	$O(n)$	$O(\log 1)$
Minimum finden	n	$O(n)$	$O(\log n)$
Schlüssel ändern	m	$O(1)$	$O(\log n)$
Vergleich ob leer	n	$O(1)$	$O(1)$
Gesamt		$O(n^2)$	$O(m \log n)$

Priority Queue (Vorrangwarteschlange)

Priority Queue:

- Eine Priority Queue ist eine Datenstruktur, die eine Menge S von Elementen verwaltet.
- Jedes Element $v \in S$ hat einen dazugehörigen Wert i, der die Priorität von v beschreibt.
- Kleinere Werte repräsentieren höhere Prioritäten.

Operationen: Alle mit Laufzeit in $O(\log n)$.

- Einfügen eines Elements in die Menge S.
- Löschen eines Elements aus der Menge S.
- Finden eines Elements mit dem kleinsten Wert (höchster Priorität).

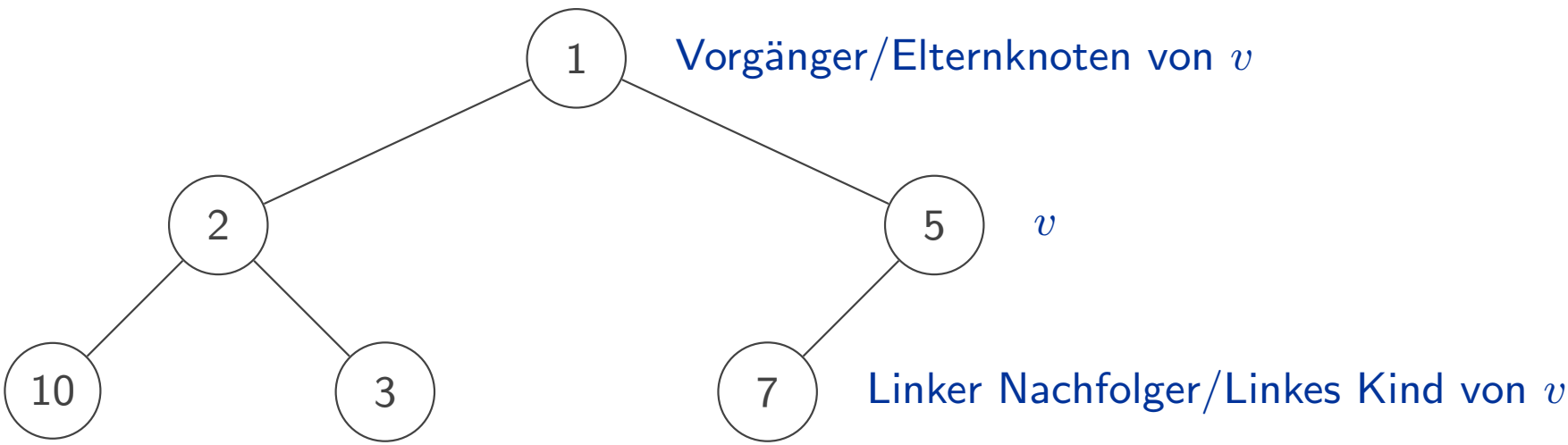
Frage: Wie erreicht man eine Laufzeit in $O(\log n)$?
Antwort: Mit einer bestimmten Datenstruktur, dem Heap.

Heap

Heap: Ein Heap (Min-Heap) ist ein binärer Wurzelbaum, dessen Knoten mit $\leq$ total geordnet sind, sodass gilt:

- Ist u ein linkes oder rechtes Kind von v, dann gilt $v \leq u$ (Heap-Eigenschaft für Min-Heap).
- Alle Ebenen von Knoten bis auf die letzte sind vollständig aufgefüllt.
- Die letzte Ebene des Baumes muss linksbündig aufgefüllt werden.

Beispiel:



Repräsentation eines Heaps

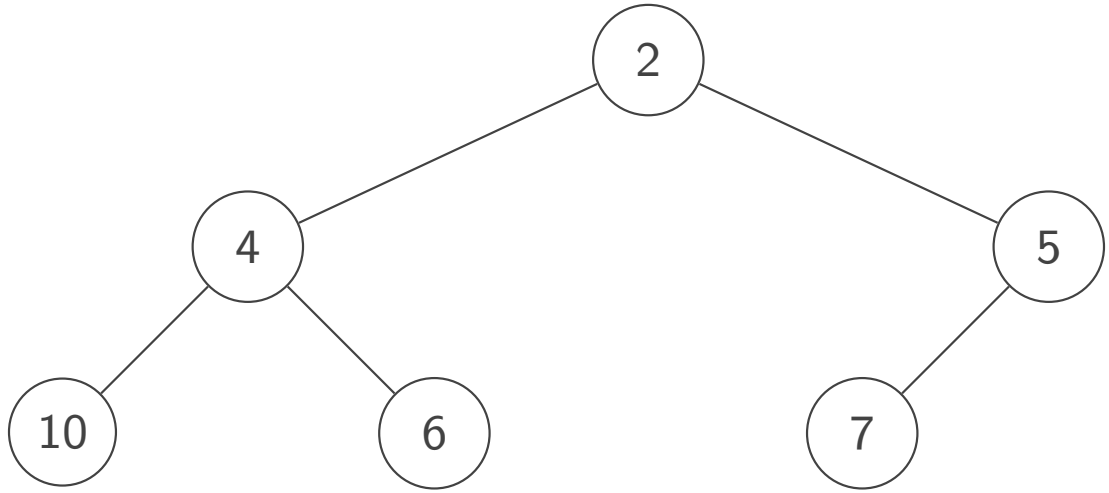
Effiziente Repräsentation: Knoten des Baums ebenenweise in einem Array speichern.

Effiziente Berechnung:

- Die beiden Nachfolgerknoten eines Knotens an der Position k befinden sich an den Positionen $2k$ und $2k+1$. Sein Elternknoten befindet sich an der Position $\lfloor \frac{k}{2} \rfloor$.
- Damit obige Rechnung immer funktioniert, wird das Array ab Index 1 belegt.
- Würde man bei Index 0 anfangen, dann würden sich die Berechnungen folgendermaßen ändern: Nachfolger links auf $2k+1$, Nachfolger rechts auf $2k+2$, Elternknoten auf $\lfloor \frac{k-1}{2} \rfloor$.

Beispiel für Heap-Repräsentation

Heap:



Array: 6 Einträge, erster Platz unbelegt (mit 0 initialisiert).

Index	0	1	2	3	4	5	6
Wert	0	2	4	5	10	6	7

Heapify-up

Einfügen eines neuen Elements: Bei einem Heap mit n Elementen wird das neue Element an Position $n+1$ eingefügt. Wir gehen dabei davon aus, dass noch genügend Plätze im Array frei sind.

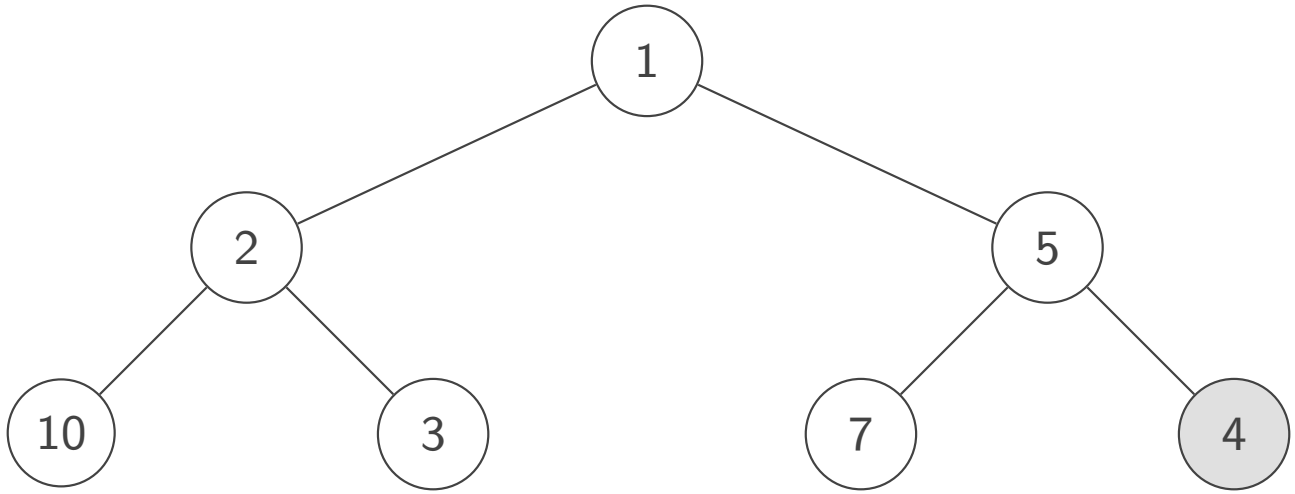
Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element verletzt werden.

Reparieren: Durch Operation Heapify-up (für Heap-Array H an Position i) in $O(\log n)$ Zeit. Aufruf nach dem Einfügen des neuen Elements: Heapify-up(H,n+1).

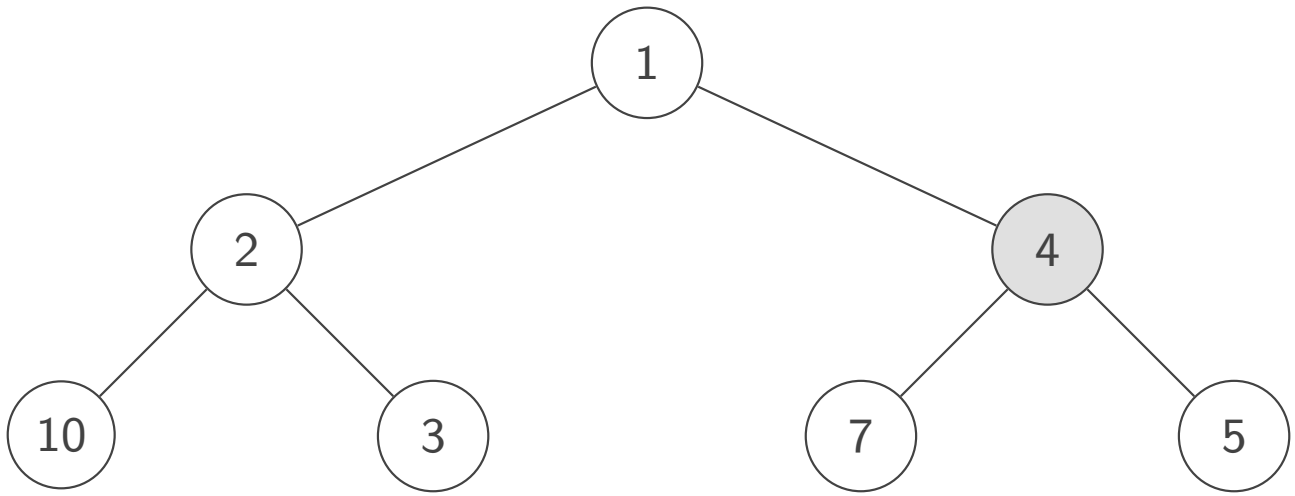
```
Heapify-up(H,i):
if i > 1
    j ← ⌊i/2⌋
    if H[i] < H[j]
        Vertausche die Array-Einträge H[i] und H[j]
        Heapify-up(H,j)
```

Beispiel für Heapify-up

Einfügen von 4:



Verschieben von 4:



Heapify-down

Löschen eines Elements: Element wird an Stelle i gelöscht. Das Element an Stelle n (bei n Elementen) wird an die freie Stelle verschoben.

Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element an der Stelle i verletzt werden.

Reparieren:

- Eingefügtes Element ist zu groß: Benutze Heapify-down, um das Element auf eine untere Ebene zu bringen.
- Eingefügtes Element ist zu klein: Benutze Heapify-up (wie beim Einfügen) von der Stelle i aus.

Hinweis: Beim Heap wird typischerweise die Wurzel entfernt und daher wird dann nur Heapify-down benutzt.

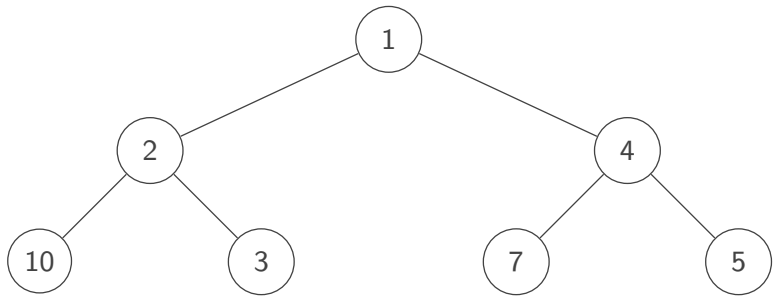
Laufzeit für Löschen: Für Heap-Array H an Position i in $O(\log n)$ Zeit.

Heapify-down

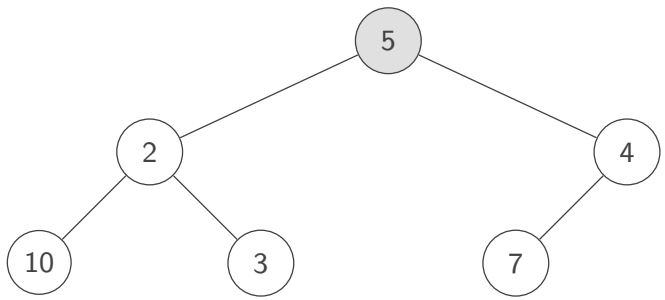
```
Heapify-down(H,i):
n ← length(H)-1
if 2 · i > n
    return
elseif 2 · i < n
    left ← 2 · i, right ← 2 · i + 1
    j ← Index des kleineren Wertes von H[left] und H[right]
else
    j ← 2 · i
if H[j] < H[i]
    Vertausche die Arrayeinträge H[i] und H[j]
    Heapify-down(H,j)
```

Beispiel für Heapify-down

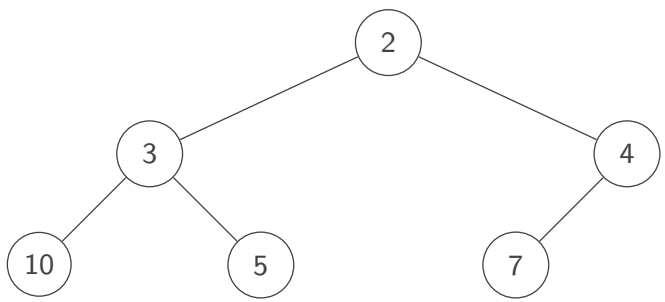
Ursprünglicher Heap:



Löschen von 1, verschieben von 5:



Heapify-down (zwei Mal)



Operationen auf Heap

Operationen auf Heap:

- Insert(H,v): Element v in den Heap H einfügen. Hat der Heap n Elemente, dann liegt die Laufzeit in $O(\log n)$.
- FindMin(H): Findet das Minimum im Heap H. Laufzeit ist konstant (da Wurzel).
- Delete(H,i): Löscht das Element im Heap H an der Stelle i. Für einen Heap mit n Elementen liegt die Laufzeit in $O(\log n)$.
- ExtractMin(H): Kombination von FindMin und Delete und daher in $O(\log n)$.

Erstellen eines Heaps

Erstellen: Das Erstellen eines Heaps aus einem Array A mit Größe n, das noch nicht die Heapeigenschaft erfüllt:

```
Init(A,n):
for i = ⌊n/2⌋ bis 1
    Heapify-down(A,i)
```

Erstellen eines Heaps: Analyse

Laufzeit: $O(n)$

- Vollständiger Binärbaum mit j Stufen hat genau $2^j - 1$ Knoten.
- Heap mit n Schlüsseln hat genau $j = \lceil \log_2(n+1) \rceil$ Stufen.
- Sei $2^{j-1} \leq n \leq 2^j - 1$, d.h. j ist die Anzahl der Stufen des Binärbaums. Auf Stufe k sind höchstens 2^{k-1} Schlüssel.
- Die Anzahl der Vergleiche und Bewegungen zum Einfügen eines Elements von Stufe k ist im Worst-Case proportional zu $j - k$, d.h. proportional zu

$$\begin{aligned}
\sum_{k=1}^{j-1} 2^{k-1}(j-k) &= 2^0(j-1) + 2^1(j-2) + 2^2(j-3) + \ldots + 2^{j-2} \cdot 1 \\
&= \sum_{k=1}^{j-1} k \cdot 2^{j-k-1} = 2^{j-1} \sum_{k=1}^{j-1} \frac{k}{2^k} \\
&\overset{(*)}{\leq} 2^{j-1} \cdot 2 \leq 2n = O(n).
\end{aligned}$$

(*) folgt aus $\sum_{k=1}^{\infty} \frac{k}{2^k} = 2$