

Effiziente Algorithmen

Master of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WiSe 2022/23

Einleitung

- *Bewertung von Algorithmen*
- *Berechenbarkeitstheorie*
- *Komplexitätstheorie*
- *Exakte Algorithmen für schwere Probleme*

Entwurfsmethoden

- *divide and conquer*
- *dynamic programming*
- *greedy*
- *local search*

Sortieren

- *Quick-Sort – unterschiedliche Varianten*
- *Heap-Sort*
- *Untere Schranke*
- *Counting-/Radix-Sort*
- *Bucket-Sort*

Auswahlproblem (Median-Berechnung)

- *randomisiert*
- *worst-case*

Graphalgorithmen

- *Tiefen- und Breitensuche und deren Anwendungen*
- *Zusammenhangsprobleme*
- *Kürzeste Wege*
- *Minimaler Spannbaum*
- *Netzwerkfluss*
- *Matching*

Spezielle Graphklassen

- *Bäume und Co-Graphen*
- *Chordale Graphen*
- *Vergleichbarkeitsgraphen (comparability graph)*
- *Planare Graphen*

Vorrangwarteschlangen

- *Linksbäume*
- *Binomial-Queues*
- *Fibonacci-Heaps*

Suchbäume

- *AVL-Bäume*
- *Rot-Schwarz-Bäume*
- *B-Bäume*
- *Splay-Bäume*
- *Tries*

Amortisierte Laufzeitanalysen

- *dynamic tables*
- *Selbstanordnende lineare Listen*
- *Splay-Bäume*
- *Fibonacci-Heaps*
- *Union-Find-Datenstruktur*

Algorithmen für moderne Hardware

- *Lokalität der Daten*
- *Externe Datenstrukturen*

Algorithmen für geometrische Probleme

- *Scan-Line-Prinzip*
- *Konvexe Hülle*
- *Polygon-Triangulierung*
- *Voronoi-Diagramme*
- *Datenstrukturen*

Randomisierte Algorithmen

- *Identitätstest*
- *Primzahltest*

- T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen*. Springer Spektrum
- T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms*. MIT Press
- Robert Sedgewick: *Algorithms*. Addison-Wesley
- Uwe Schöning: *Algorithmen - kurz gefasst*. Spektrum Akademischer Verlag
- Volker Heun: *Grundlegende Algorithmen*. Vieweg Verlag
- A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Datastructures and Algorithms*. Addison-Wesley

- F. Gurski, I. Rothe, J. Rothe, E. Wanke: *Exakte Algorithmen für schwere Graphenprobleme*. Springer Verlag
- Shimon Even: *Graph Algorithms*. Computer Science Press
- Uwe Schöning: *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag
- C.H. Papadimitriou: *Computational Complexity*. Addison-Wesley
- Juraj Hromkovič: *Randomisierte Algorithmen*. Vieweg + Teubner Verlag
- Rolf Wanka: *Approximationsalgorithmen*. Teubner B.G.
- I. Gerdes, F. Klawonn, R. Kruse: *Evolutionäre Algorithmen*. Vieweg Verlag

- Einleitung
- Entwurfsmethoden
- Sortieren
- Auswahlproblem
- *Graphalgorithmen*
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

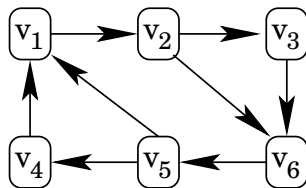
Graphalgorithmen

- *Grundlagen*
- Tiefen- und Breitensuche
- Zusammenhangsprobleme
- Minimale Spannbäume
- Kürzeste Wege
- Netzwerkfluss
- Matching-Probleme

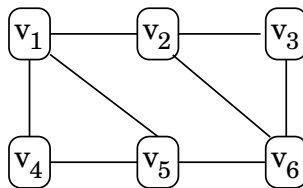
Ein *gerichteter Graph* $G = (V, E)$ besteht aus

- einer endlichen Menge von *Knoten* $V = \{v_1, \dots, v_n\}$ und
- einer Menge von gerichteten *Kanten* $E \subseteq V \times V$.

Bei einem *ungerichteten Graphen* $G = (V, E)$ sind die Kanten ungeordnete Paare:
 $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$



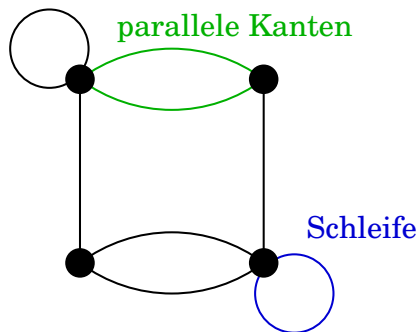
gerichteter Graph



ungerichteter Graph

Wir bezeichnen im Folgenden $|V| = \mathcal{V}$ und $|E| = \mathcal{E}$.

Übung: Sehen Sie sich unsere Definition von ungerichteten Graphen noch einmal genau an. Überlegen Sie, ob parallele Kanten oder Schleifen möglich sind.



Wieviele Kanten kann ein Graph maximal haben?

Parallele Kanten sind nicht möglich, da die Kanten über eine Teilmenge definiert sind und in Mengen keine Elemente mehrfach vorkommen:

$$E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$$

Da in der Definition $u \neq v$ gefordert wird, sind auch keine Schleifen möglich.

Möchte man parallele Kanten oder Schleifen zulassen, kann man die Definition mittels Multimengen formulieren.

Wenn es keine parallelen Kanten gibt, dann kann es maximal

$$\frac{\mathcal{V} \cdot (\mathcal{V} - 1)}{2}$$

viele Kanten geben, also gilt: $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$

Die Kanten eines Graphen können gewichtet sein, um z.B. Längen oder Zeiten beschreiben zu können.

Gewichtete Graphen G werden wir als 3-Tupel $G = (V, E, c)$ angeben, wobei

- V die Knotenmenge,
- E die Kantenmenge und
- $c : E \rightarrow \mathbb{R}$ eine Funktion bezeichnet, die jeder Kante e eine Zahl $c(e)$ zuordnet.

Graphen verwenden wir überall dort, wo ein Sachverhalt darstellbar ist durch

- eine Menge von Objekten (Entitäten)
- und Beziehungen zwischen den Objekten.

Beispiele:

- *Routenplanung*: Städte sind durch Straßen verbunden; die Straßen haben eine gewisse Länge, die als Kantengewicht angegeben wird.
- *Kursplanung*: Kurse setzen andere Kurse voraus.
- *Produktionsplanung*: Produkte werden aus Einzelteilen und Teilprodukten zusammengesetzt.
- *Schaltkreisanalyse*: Bauteile sind durch elektrische Leitungen verbunden.
- *Spiele*: Der Zustand/Status eines Spiels wird durch einen Spielzug geändert.

Sei $G = (V, E)$ ein gerichteter Graph und seien $u, v \in V$.

- Sei $e = (u, v)$ eine Kante. Knoten u ist der *Startknoten*, v der *Endknoten* von e . Die Knoten u und v sind *adjazent*, Knoten u bzw. v und Kante e sind *inzident*.
- Der *Eingangsgrad* von u , geschrieben $\text{indeg}(u)$, ist die Anzahl der in u einlaufenden Kanten.

Der *Ausgangsgrad* von u ist die Anzahl der aus u auslaufenden Kanten und wird mit $\text{outdeg}(u)$ bezeichnet.

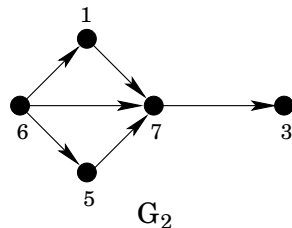
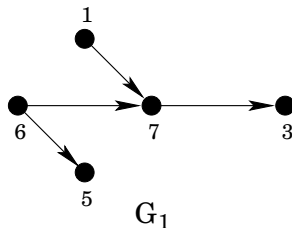
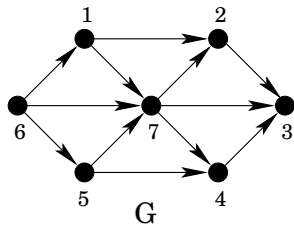
- $p = (v_0, v_1, \dots, v_k)$ ist ein *gerichteter Weg* in G der Länge k von Knoten u nach Knoten w , falls gilt: $v_0 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.
- Der gerichteter Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.
- Ein gerichteter Weg $p = (v_0, v_1, \dots, v_k, v_0)$ heißt *Kreis*, falls alle Kanten (v_{i-1}, v_i) und (v_k, v_0) paarweise disjunkt sind.

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

- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von G , geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.

Unten: G_1 und G_2 sind Teilgraphen von G .

- Ein Graph $G' = (V', E')$ heißt **induzierter Teilgraph** von G , falls $V' \subseteq V$ und $E' = E \cap (V' \times V')$ gilt. Der Graph $G|_{V'}$ bezeichnet den durch V' induzierten Teilgraphen von G .
- Unten: G_1 ist kein induzierter Teilgraph von G , da die Kanten $(6, 1)$ und $(5, 7)$ fehlen. G_2 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G .



Sei $G = (V, E)$ ein ungerichteter Graph und seien $u, v \in V$.

- Sei $e = \{u, v\}$ eine Kante. Die Knoten u und v sind *adjazent*, Knoten u bzw. v und Kante e sind *inzident*. Knoten u und v sind die *Endknoten* der Kante e .
- Der *Knotengrad* von u , geschrieben $\deg(u)$, ist die Anzahl der zu u inzidenten Kanten.
- $p = (v_0, v_1, \dots, v_k)$ ist ein *ungerichteter Weg* in G der Länge k von Knoten u nach Knoten w , falls gilt: $v_0 = u$, $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 \leq i \leq k$.
- Der ungerichteter Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.
- Ein ungerichteter Weg $p = (v_0, v_1, \dots, v_k, v_0)$ heißt *Kreis*, falls alle Kanten, also $\{v_{i-1}, v_i\}$ und $\{v_k, v_0\}$, paarweise disjunkt sind.

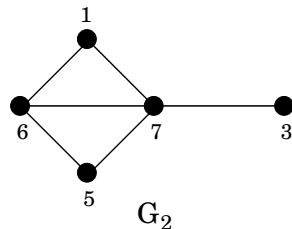
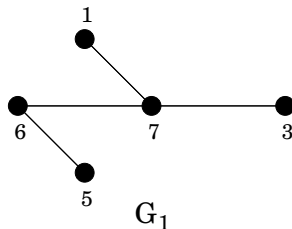
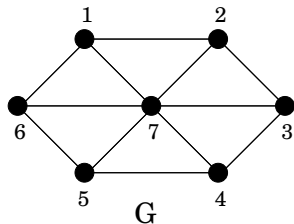
Begriffe: ungerichtete Graphen

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

- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von G , geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.

Unten: G_1 und G_2 sind Teilgraphen von G .

- Ein Graph $G' = (V', E')$ heißt **induzierter Teilgraph** von G , falls $V' \subseteq V$ und $E' = E \cap \{\{u, v\} \mid u, v \in V', u \neq v\}$ gilt.
- Unten: G_1 ist kein induzierter Teilgraph von G , da die Kanten $\{1, 6\}$ und $\{5, 7\}$ fehlen. G_2 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G .

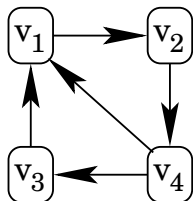


Speicherung von Graphen

Sei n die Anzahl der Knoten in dem Graphen $G = (V, E)$. Die *Adjazenz-Matrix* für G ist eine $n \times n$ -Matrix $A_G = (a_{ij})$ mit

$$a_{ij} = \begin{cases} 0, & \text{falls } (v_i, v_j) \notin E, \\ 1, & \text{falls } (v_i, v_j) \in E. \end{cases}$$

Beispiel:



gerichtet:

A	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	0
4	1	0	1	0

ungerichtet:

A	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

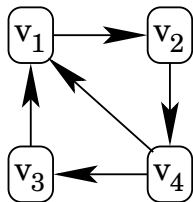
Frage: Wie lange dauert die Operation

- „falls Knoten u mit Knoten v verbunden ist ...“
- „für alle zu Knoten u benachbarten Knoten v ...“

Speicherung von Graphen

Bei einer *Adjazenz-Liste* werden für jeden Knoten v eines Graphen $G = (V, E)$ in einer doppelt verketteten Liste $Adj[v]$ alle von v ausgehenden Kanten gespeichert.

Beispiel:



gerichtet:

$$Adj[v_1] = \{v_2\}$$

$$Adj[v_2] = \{v_4\}$$

$$Adj[v_3] = \{v_1\}$$

$$Adj[v_4] = \{v_1, v_3\}$$

ungerichtet:

$$Adj[v_1] = \{v_2, v_3, v_4\}$$

$$Adj[v_2] = \{v_1, v_4\}$$

$$Adj[v_3] = \{v_1, v_4\}$$

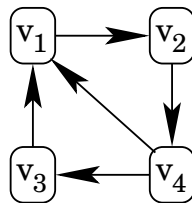
$$Adj[v_4] = \{v_1, v_2, v_3\}$$

Frage: Wie lange dauert die Operation

- „falls Knoten u mit Knoten v verbunden ist ...“
- „für alle zu Knoten u benachbarten Knoten v ...“

Speicherung von Graphen

Bei einem *Adjazenz-Array* werden alle Kanten in einem Array abgelegt, alle zu einem Knoten inzidenten Kanten liegen hintereinander im Array.



gerichtet:

i	1	2	3	4	5
edgeOf	0	1	2	3	5

i	0	1	2	3	4	5
target	2	4	1	1	3	-

ungerichtet:

i	1	2	3	4	5
edgeOf	0	3	5	7	10

i	0	1	2	3	4	5	6	7	8	9	10
target	2	3	4	1	4	1	4	1	2	3	-

Vergleich: Sei $G = (V, E)$ ein Graph mit n Knoten und m Kanten. Zur Speicherung von G wird folgender Platz benötigt:

- Adjazenz-Matrix: $\mathcal{O}(n^2)$
 - Geeignet für *dichte* Graphen (dense graphs).
 - Adjazenz-Matrizen unterstützen sehr gut Aufgaben wie: Falls Knoten u und v adjazent sind, tue ...
- Adjazenz-Liste und -Array: $\mathcal{O}(n + m)$
 - Auch geeignet für *dünn besetzte* Graphen (sparse graphs).
 - Adjazenz-Listen und -Arrays unterstützen sehr gut das Verfolgen von Kanten: Für alle zu Knoten u inzidenten Kanten tue ...

Hinzufügen und Löschen von Knoten werden nicht unterstützt. Es gibt aber voll dynamische Datenstrukturen ...

Graphalgorithmen

- Grundlagen
- *Tiefen- und Breitensuche*
- Zusammenhangsprobleme
- Minimale Spannbäume
- Kürzeste Wege
- Netzwerkfluss
- Matching-Probleme

Aufgabe: Die Suche soll alle von einem gegebenen Startknoten s aus erreichbaren Knoten finden.

Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle aus s auslaufenden Kanten zu D hinzu
solange D nicht leer ist:
 entnehme eine Kante (u, v) aus D
 falls der Knoten v als „unbesucht“ markiert ist:
 markiere Knoten v als „besucht“
 füge alle aus v auslaufenden Kanten zu D hinzu

Anmerkung: In der Datenstruktur D speichern wir diejenigen Kanten, von deren Endknoten vielleicht noch unbesuchte Knoten erreicht werden können.

Laufzeit:

- Jede Kante wird höchstens einmal in D eingefügt.
 - Jeder Knoten wird höchstens einmal inspiziert.
- ⇒ Die Laufzeit ist proportional zur Anzahl der vom Startknoten aus erreichbaren Knoten und Kanten, also $\mathcal{O}(\mathcal{V} + \mathcal{E})$.

Der Typ der Datenstruktur bestimmt die *Durchlaufordnung*:

- Stack (last in, first out): *Tiefensuche*
- Liste (first in, first out): *Breitensuche*

Wir unterscheiden die Kanten bei einem gerichteten Graphen nach der Rolle, die sie bei der Tiefensuche spielen.

- **Baumkante:** Kante $(u, v) \in E$, der die Tiefensuche folgt, wo also während der Abarbeitung von $\text{dfs}(u)$ ein Aufruf $\text{dfs}(v)$ erfolgt.
- **Vorwärtskante:** Kante $(u, v) \in E$ mit $\text{dfb}[v] > \text{dfb}[u]$, die keine Baumkante ist.
- **Querkante:** Eine Kante $(u, v) \in E$ mit $\text{dfb}[v] < \text{dfb}[u]$ und $\text{dfe}[v] < \text{dfe}[u]$.
- **Rückwärtskante:** Kante $(u, v) \in E$ mit $\text{dfb}[v] < \text{dfb}[u]$ und $\text{dfe}[v] > \text{dfe}[u]$.

Rekursive Tiefensuche:

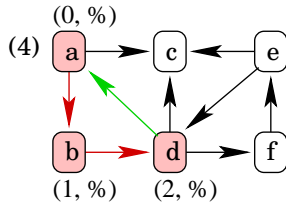
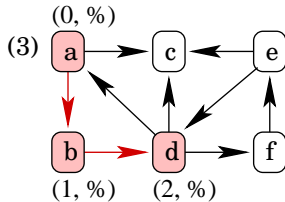
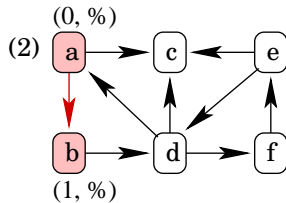
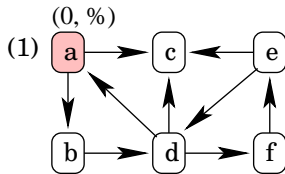
Sei $s \in V$ ein beliebiger Knoten.

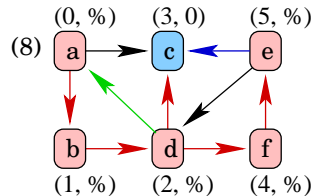
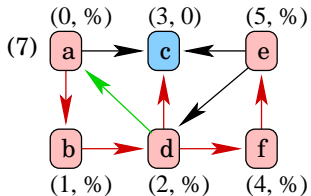
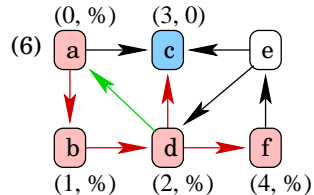
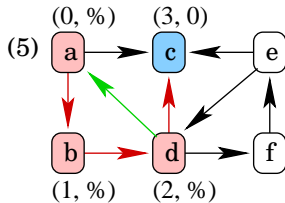
markiere alle Knoten als „unbesucht“
 $\text{dfbZähler} := 0$
 $\text{dfeZähler} := 0$
 $\text{dfs}(s)$

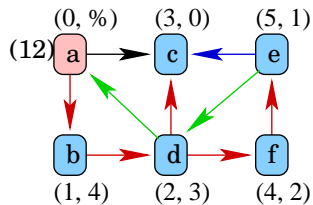
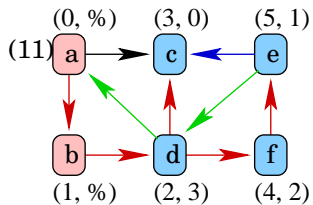
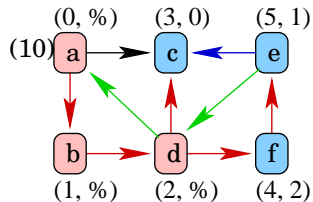
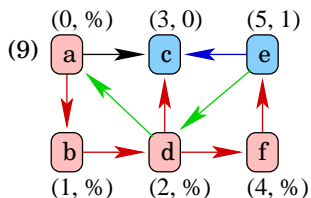
Diese Art der Tiefensuche, wo die Suche bei einem gegebenen Startknoten beginnt, bezeichnen wir als einfache Tiefensuche.

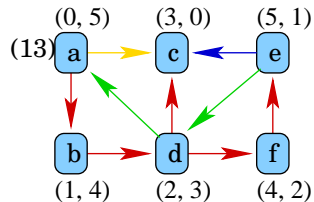
$\text{dfs}(u)$

```
markiere  $u$  als „besucht“  
 $\text{dfb}[u] := \text{dfbZähler}$   
 $\text{dfbZähler} := \text{dfbZähler} + 1$   
betrachte alle Kanten  $(u, v) \in E$ :  
    falls Knoten  $v$  als „unbesucht“ markiert ist:  
         $\text{dfs}(v)$   
 $\text{dfe}[u] := \text{dfeZähler}$   
 $\text{dfeZähler} := \text{dfeZähler} + 1$ 
```









- Baumkanten sind rot markiert.
- Rückwärtskanten sind grün markiert.
- Querkanten sind blau markiert.
- Vorwärtskanten sind gelb markiert.

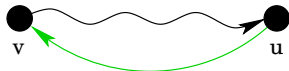
Tiefensuche: (Suche beginnt nicht bei einem ausgezeichneten Startknoten.)

markiere alle Knoten als unbesucht
solange ein unbesuchter Knoten v existiert:
dfs(v)

Satz: Der gerichtete Graph G enthält einen Kreis \iff die Tiefensuche auf G liefert eine Rückwärtskante.

Beweis:

“ \Leftarrow “ Für eine Rückwärtskante (u, v) gilt: $dfb(u) > dfb(v)$ und $dfe(u) < dfe(v)$.
Außerdem existiert ein Weg von v nach u .



Gäbe es keinen Weg von v nach u , dann wäre $dfs(v)$ beendet, bevor $dfs(u)$ aufgerufen wird, also $dfe(v) < dfe(u)$. ⚡⚡

Beweis: (Fortsetzung)

“ \Rightarrow “ Sei $C = (v_1, v_2, \dots, v_k, v_1)$ ein Kreis in G .

O.B.d.A. sei v_1 der Knoten aus C , der von der Tiefensuche zuerst besucht wird, also $dfb(v_1) < dfb(v_k)$.

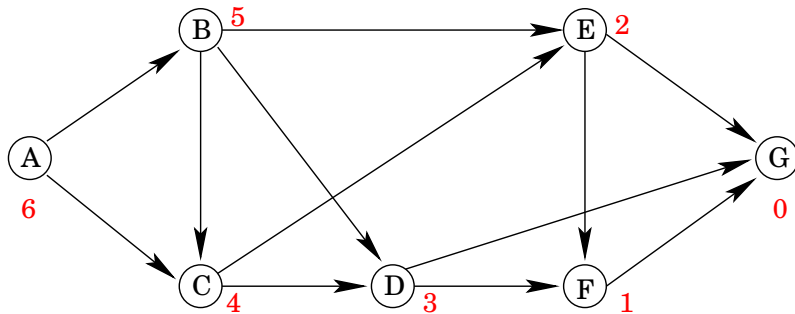
$dfs(v_1)$ wird erst beendet, wenn alle von v_1 aus erreichbaren Knoten, also insbesondere v_k , besucht und abgearbeitet wurden. Daher gilt $dfe(v_1) > dfe(v_k)$.

Also ist (v_k, v_1) eine Rückwärtskante.

Anwendungen: Topologische Sortierung

Gegeben: Ein gerichteter Graph $G = (V, E)$.

Gesucht: Eine Nummerierung $\pi(v_1), \dots, \pi(v_n)$ der Knoten, so dass gilt:
 $(u, v) \in E \Rightarrow \pi(u) > \pi(v)$



Algorithmus: Tiefensuche

G ist kreisfrei \Rightarrow dfe-Nummern sind topologische Sortierung!

Korrektheit: Für alle Kanten $(u, v) \in E$ muss $dfe(u) > dfe(v)$ gelten.

- Wenn G kreisfrei ist, treten keine Rückwärtskanten auf.

- (u, v) ist eine Baumkante

$\rightarrow dfe(u) > dfe(v) \checkmark$

Denn: $dfs(u)$ wird erst beendet, wenn $dfs(v)$ bereits abgeschlossen ist.



- (u, v) ist eine Vorwärtskante

$\rightarrow dfe(u) > dfe(v) \checkmark$

Denn: $dfs(v)$ ist bereits beendet, während bei $dfs(u)$ die Vorwärtskante entdeckt wird.



- (u, v) ist eine Querkante

$\rightarrow dfe(u) > dfe(v)$ nach Definition \checkmark

Anmerkung: Wenn G nicht kreisfrei ist, also einen Kreis enthält, dann existiert keine topologische Sortierung der Knoten.

Graphalgorithmen:

- Grundlagen
- Tiefen- und Breitensuche
- *Zusammenhangsprobleme*
- Minimale Spannbäume
- Kürzeste Wege
- Netzwerkfluss
- Matching-Probleme

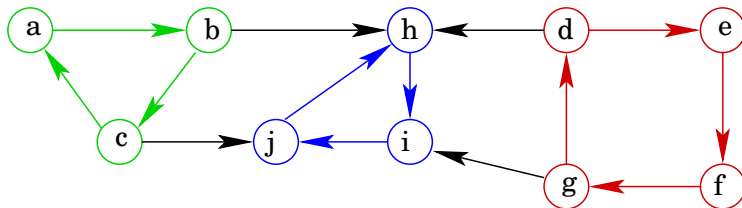
Sei $G = (V, E)$ ein gerichteter Graph.

- G ist *stark zusammenhängend*, wenn es zwischen jedem Knotenpaar einen Weg in G gibt.

Also: Für alle $u, v \in V$ existiert ein Weg von u nach v und ein Weg von v nach u in G .

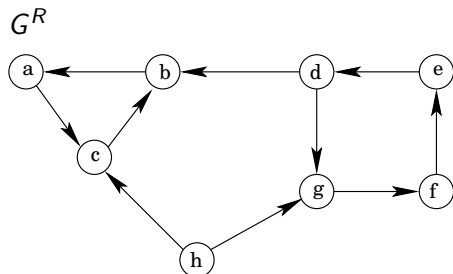
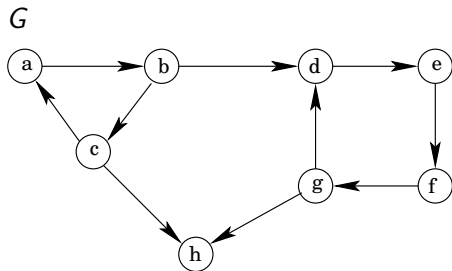
- Eine *starke Zusammenhangskomponente* von G ist ein bzgl. der Knotenmenge maximaler, stark zusammenhängender, induzierter Teilgraph von G .

Beispiel:



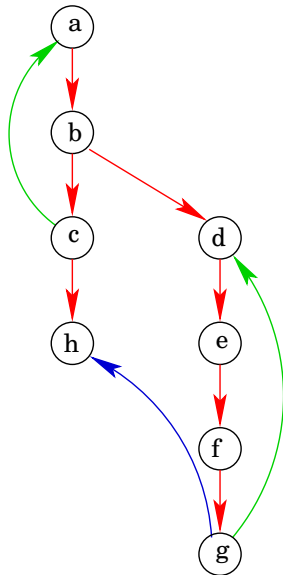
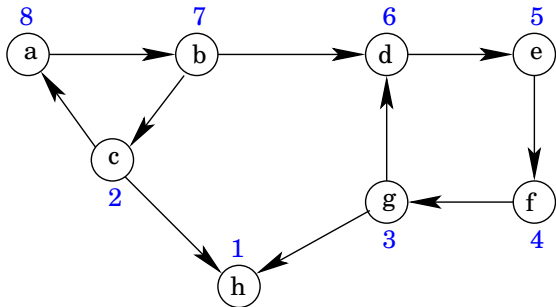
Algorithmus: Gegeben sei ein gerichteter Graph $G = (V, E)$.

- 1 Tiefensuche auf G mit dfe-Nummerierung der Knoten.
- 2 Berechne $G^R = (V, E^R)$ mit $E^R = \{(v, u) \mid (u, v) \in E\}$.
- 3 Tiefensuche auf G^R , wobei immer mit dem Knoten v mit *größtem dfe-Index* aus *Schritt 1* gestartet wird.



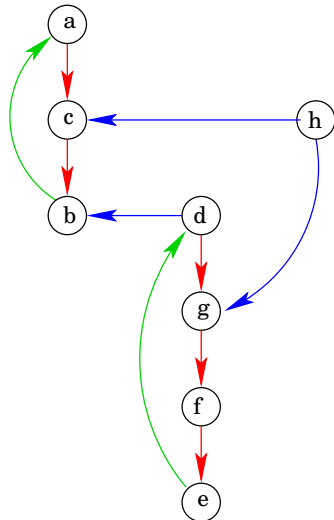
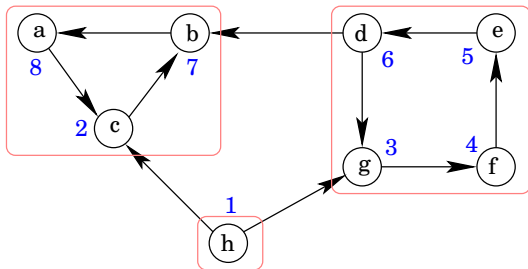
Jeder Baum des in Schritt 3 entstandenen Waldes entspricht einer starken Zusammenhangskomponente.

Beispiel: Eine mögliche Tiefensuche auf G mit Startknoten a liefert:

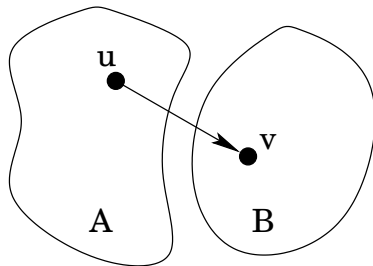


Zusammenhangsprobleme

Beispiel: Tiefensuche auf G^R , wobei immer mit dem Knoten v mit *größtem dfe-Index* aus dem vorigen Schritt gestartet wird, liefert:



Warum ist es wichtig, in Schritt 3 den Knoten mit dem größten dfe-Index zu wählen?



Wenn eine Kante $(u, v) \in E$ existiert, die eine starke ZHK A mit einer starken ZHK B verbindet, dann gilt nach dem ersten Schritt: $dfe(u) > dfe(v)$

Eine Kante von B nach A kann also nicht existieren, da sonst A und B eine einzige starke ZHK bilden würden.

- falls die Tiefensuche in B beginnt:
 - Tiefensuche in B ist beendet, bevor Tiefensuche in A startet.
 - $dfe(u) > dfe(v)$
- falls die Tiefensuche in A beginnt:
 - Die Tiefensuche erreicht irgendwann B, und B wird komplett abgearbeitet, bevor die Tiefensuche in A fortgesetzt wird.
 - $dfe(u) > dfe(v)$

Korrektheit:

Wir stellen zunächst fest, dass die Graphen G und G^R dieselben starken Zusammenhangskomponenten haben, denn:

- $u \overset{G}{\rightsquigarrow} v \Rightarrow v \overset{G^R}{\rightsquigarrow} u$

Wenn ein Weg von u nach v in G existiert, dann existiert in G^R ein Weg von v nach u .

- $v \overset{G}{\rightsquigarrow} u \Rightarrow u \overset{G^R}{\rightsquigarrow} v$

Wenn ein Weg von v nach u in G existiert, dann existiert in G^R ein Weg von u nach v .

Korrektheit: (Fortsetzung)

Wir bestimmen die starken Zshg.-komponenten durch Tiefensuche auf G^R . Betrachten wir zwei starke Zshg.-komponenten A und B:

- Eine Kante $(u, v) \in E$ von A nach B wird in G^R umgedreht.
 - In G kann keine Kante von B nach A existieren, da sonst A und B eine große Zshg.-komponente bilden würde.
 - Es existieren in G^R also keine Kanten von A nach B.
 - Zunächst wird eine Tiefensuche in A gestartet, da aufgrund unserer Vorüberlegung $dfe(u) > dfe(v)$ gilt.
- Der Tiefensuchbaum umfasst genau die Knoten in A.

Anschließend sind die Knoten von A als besucht markiert und werden nie wieder besucht. Also werden auch die Knoten von B durch eine spätere Tiefensuche korrekt ausgegeben.

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

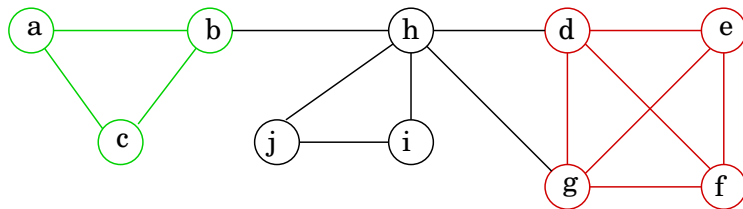
- G heißt *zusammenhängend*, wenn es zwischen jedem Knotenpaar $u, v \in V$ einen Weg in G gibt.
- G heißt *k -fach zusammenhängend*, wenn es zwischen jedem Knotenpaar k knotendisjunkte Wege gibt.

Das heißt, außer die Start- und Endknoten sind alle auf den Wegen liegenden Knoten paarweise verschieden.

- Eine *k -fache Zusammenhangskomponente* von G ist ein maximaler k -fach zusammenhängender, induzierter Teilgraph von G .
- Ein Knoten u ist ein *Schnittpunkt*, wenn der Graph G ohne Knoten u (und damit auch ohne die zu u inzidenten Kanten) mehr Zusammenhangskomponenten hat als G .

Englisch: *cut point* oder *articulation point*.

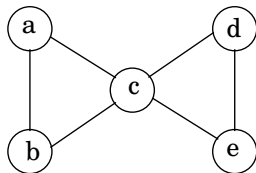
Beispiel: Der folgende Graph ist zusammenhängend.



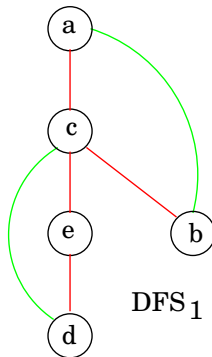
- Eine der 2-fach Zusammenhangskomponenten ist der grün markierte Teilgraph.
- Der rot markierte Teilgraph stellt sogar eine 3-fach Zusammenhangskomponente dar.
- Knoten b und h sind Schnittpunkte: Wird einer dieser Knoten entfernt, ist der Restgraph nicht mehr zusammenhängend.

Wir werden im Folgenden solche Schnittpunkte berechnen. Ein Graph ohne Schnittpunkt ist mindestens 2-fach zusammenhängend.

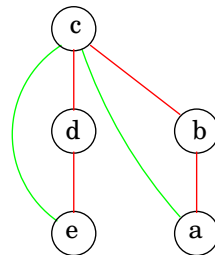
Tiefensuche auf ungerichteten Graphen G :



Graph G



DFS₁



DFS₂

- (a) Es existieren keine Querkanten.
- (b) Die Wurzel w eines DFS-Baums ist ein Schnittpunkt, wenn w mehr als einen Teilbaum hat, denn: G ohne w zerfällt wegen Punkt (a) in mehrere Teile, siehe Knoten c bei DFS₂.

- (c) Ein innerer Knoten v ist *kein* Schnittpunkt, wenn aus jedem Teilbaum von v eine Rückwärtskante zu einem Vorgänger von v geht, denn: Wenn v aus G entfernt wird, existiert über die Rückwärtskante ein alternativer Weg in den Teilbaum.
- Knoten c in DFS_1 ist ein Schnittpunkt: Vom rechten Teilbaum geht zwar eine Rückwärtskante zu einem Vorgänger von c im Baum, aber vom linken Teilbaum nicht.

Idee des Algorithmus: Tiefensuche auf G .

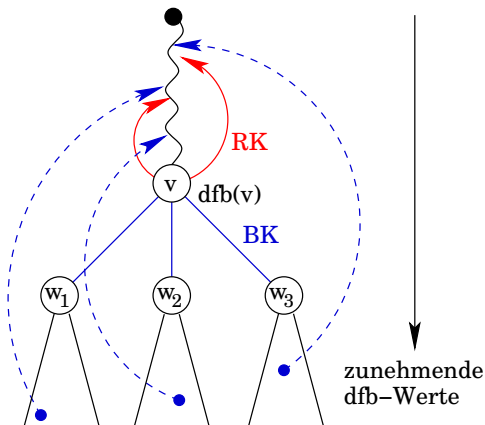
- Berechne für Knoten v am Ende von $\text{dfs}(v)$:

$$\text{low}(v) := \min \left\{ \text{dfb}(v), \min_{(v,z) \in RK} \{ \text{dfb}(z) \}, \min_{(v,w) \in BK} \{ \text{low}(w) \} \right\}$$

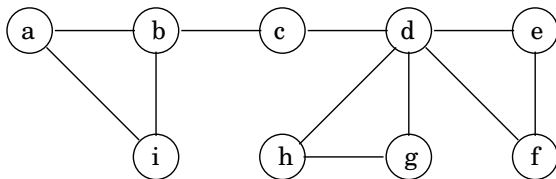
- „Wie hoch geht es im Baum über Rückwärtskanten?“

Anschaulich:

$$low(v) := \min \left\{ dfb(v), \min_{(v,z) \in RK} \{dfb(z)\}, \min_{(v,w) \in BK} \{low(w)\} \right\}$$



Zusammenhangsprobleme



$$\text{low}(f) = \min(6, 4, \%) = 4$$

$$\text{low}(e) = \min(5, \%, 4) = 4$$

$$\text{low}(h) = \min(8, 4, \%) = 4$$

$$\text{low}(g) = \min(7, \%, 4) = 4$$

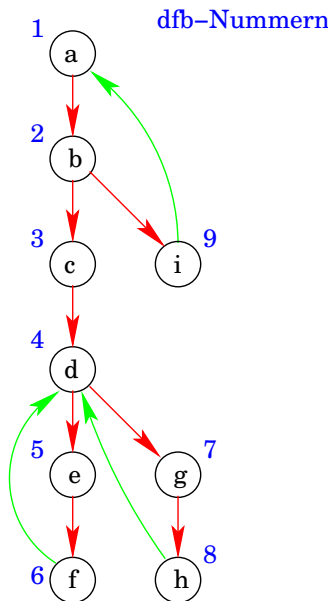
$$\text{low}(d) = \min(4, \%, 4, 4) = 4$$

$$\text{low}(c) = \min(3, \%, 4) = 3$$

$$\text{low}(i) = \min(9, 1, \%) = 1$$

$$\text{low}(b) = \min(2, \%, 1) = 1$$

$$\text{low}(a) = \min(1, \%, 1) = 1$$



Algorithmus:

```
markiere alle Knoten als „unbesucht“  
dfbZähler := 1  
initialisiere einen leeren Stack  
für alle Knoten  $v \in V$ :  
    falls  $v$  als „unbesucht“ markiert ist  
        ZSuche( $v$ )
```

Vorbemerkungen zur Funktion ZSuche:

- Zeile 10: Ein Knoten v ist ein Schnittpunkt, wenn ein Kind w von v existiert mit $low(w) \geq dfb(v)$
- Der Wert $low(v)$ wird in den Zeilen 4, 11 und 13 berechnet.
- Zeile 8: Der Vorgänger $pred[v]$ wird vermerkt, um in Zeile 12 Rückwärts- von Baumkanten unterscheiden zu können.

ZSuche(v)

1. markiere v als „besucht“
2. $dfb[v] := dfbZähler$
3. $dfbZähler := dfbZähler + 1$
4. $low[v] := dfb[v]$
5. betrachte alle mit v inzidenten Kanten $\{v, w\} \in E$:
6. lege $\{v, w\}$ auf Stack, falls noch nicht geschehen
7. falls w als „unbesucht“ markiert ist:
8. $pred[w] := v$
9. ZSuche(w) // berechnet $low[w]$
10. falls $low[w] \geq dfb[v]$: alle Kanten bis $\{v, w\}$ vom Stack nehmen und als Komponente ausgeben
11. $low[v] := \min(low[v], low[w])$
12. sonst: falls $w \neq pred[v]$:
13. $low[v] := \min(low[v], dfb[w])$

Graphalgorithmen:

- Grundlagen
- Tiefen- und Breitensuche
- Zusammenhangsprobleme
- *Minimale Spannbäume*
- Kürzeste Wege
- Netzwerkfluss
- Matching-Probleme

Definition: Ein **Spannbaum** T eines Graphen $G = (V, E)$ ist ein zusammenhängender Teilgraph $T = (V, E_T)$ von G mit $|V| - 1$ Kanten.

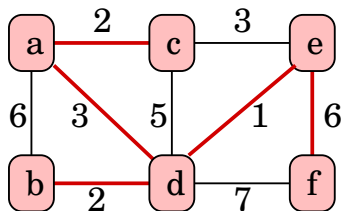
Anmerkung: Da T zusammenhängend ist, aber nur $|V| - 1$ Kanten hat, ist T kreisfrei und daher ist T ein Baum.

Gegeben: Ein ungerichteter, zusammenhängender Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \rightarrow \mathbb{R}^+$.

Gesucht: Ein Spannbaum $T = (V, E_T)$ von G mit minimalen Kosten:

$$c(T) = \sum_{e \in E_T} c(e)$$

Beispiel:



Zur Zeit bester Algorithmus stammt von Karger, Klein und Tarjan: A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM. 1995.

Aus dem Titel geht hervor, dass es sich um einen randomisierten Algorithmus mit einer erwarteten Laufzeit in $\mathcal{O}(\mathcal{V} + \mathcal{E})$ handelt.

Motivation:

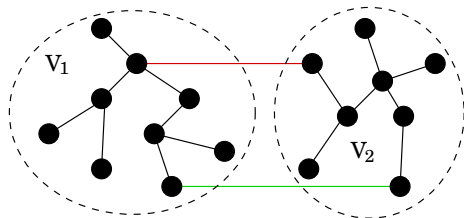
- *Verkabelung*: Alle Häuser sollen ans Telefonnetz angeschlossen werden und aus Kostengründen sollen dazu möglichst wenige Kabel verlegt werden.
- *Stromversorgung* von elektrischen Bauteilen auf einer Platine.
- *Routing*:
 - CISCO IP Multicast
 - Spanning Tree Protocol
- ☺ Es werden nur die Straßen repariert, so dass nach wie vor alle Häuser erreichbar sind.

Beobachtung: Sei (V_1, V_2) eine disjunkte Zerlegung der Knotenmenge V , also $V_1 \cap V_2 = \emptyset$ und $V_1 \cup V_2 = V$. Dann gilt: Ein minimaler Spannbaum enthält die billigste Kante $e = \{u, v\} \in E$ mit $u \in V_1$ und $v \in V_2$.

Beweis durch Widerspruch:

Wir nehmen an, kein minimaler Spannbaum enthält die billigste Kante zwischen V_1 und V_2 .

Betrachte **Kante** eines minimalen Spannbaums, die einen Knoten aus V_1 mit einem Knoten aus V_2 verbindet.



- Durch Hinzunahme der **billigsten Kante** zwischen V_1 und V_2 entsteht ein Kreis.
- Durch Streichen der **teueren Kante** zwischen V_1 und V_2 entsteht ein Spannbaum, dessen Gewicht sogar geringer ist als der ursprüngliche Spannbaum. ⚡⚡

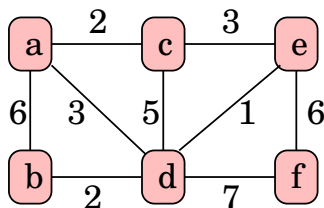
Sei $G = (V, E, c)$ ein ungerichteter Graph mit Kostenfunktion $c : E \rightarrow \mathbb{R}^+$.

```
A := ∅  
for each vertex  $v \in V$  do  
    makeSet( $v$ )  
  
sort the edges of  $E$  by non-decreasing weight  
for each edge  $(u, v) \in E$  do  
     $r_u := \text{findSet}(u)$   
     $r_v := \text{findSet}(v)$   
    if  $r_u \neq r_v$  then  
         $A := A \cup \{(u, v)\}$   
        union( $r_u, r_v$ )
```

Nach Ablauf des Algorithmus enthält die Menge A die Kanten eines minimalen Spannbaums.

Zunächst werden die Kanten anhand ihrer Gewichtung sortiert. Anschließend wird in jedem Schritt eine Kante $\{u, v\}$ aus der sortierten Liste entfernt, geprüft, ob die Endknoten u und v in verschiedenen Mengen liegen und ggf. diese Mengen verschmolzen.

Beispiel:



sorted(E)	Knotenmengen
initial	$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$
$\{d, e\}$	$\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}$
$\{a, c\}$	$\{a, c\}, \{b\}, \{d, e\}, \{f\}$
$\{b, d\}$	$\{a, c\}, \{b, d, e\}, \{f\}$
$\{a, d\}$	$\{a, c, b, d, e\}, \{f\}$
$\{c, e\}$	unverändert
$\{c, d\}$	unverändert
$\{a, b\}$	unverändert
$\{e, f\}$	$\{a, c, b, d, e, f\}$
$\{d, f\}$	unverändert

Wir benötigen eine Datenstruktur, mit der wir effizient Mengen verwalten können. Insbesondere sollen die Funktionen `findSet` und `union` unterstützt werden.

Union-Find-Datenstruktur:

- Zur Speicherung einer disjunkten Zerlegung einer Menge

$$S = X_1 \cup X_2 \cup \dots \cup X_k \text{ mit } X_i \cap X_j = \emptyset \text{ für } i \neq j.$$

- Speichere jede Klasse X_i in einem Baum.
- Der Repräsentant einer Klasse X_i ist die Wurzel des Baums.
- Die Funktion `findSet(v)` liefert den Repräsentanten des Baums, in dem Knoten v gespeichert ist.
- Damit ein Knoten schnell gefunden werden kann, werden Zeiger auf alle Elemente $v \in S$ gespeichert.
- Die Funktion `union` hängt den kleineren/flacheren Baum an die Wurzel des größeren/tieferen an.

In unserem Beispiel:

sorted(E)	Mengenrepräsentation durch Bäume					
initial	a	b	c	d	e	f
$\{d, e\}$	a	b	c	d ↑ e	f	
$\{a, c\}$	a ↑ c	b	d ↑ e		f	
$\{b, d\}$	a ↑ c	b ↗ d		e ↘ d		f

Fortsetzung:

sorted(E)	Mengenrepräsentation durch Bäume
$\{b, d\}$	
$\{a, d\}$	
...	...

Die Funktion `makeSet(v)` initialisiert den Vorgänger und den Rang der Wurzel:

```
pred[v] := 0  
rang[v] := 0
```

Wir speichern zu jedem Knoten dessen Rang, um eine Vereinigung zweier Bäume nach ihrer Höhe durchführen zu können. Aktualisiert wird nur der Rang der Wurzel bei `union(u, v)`:

```
a := find(u)  
b := find(v)  
wenn rang[a] < rang[b]  
    dann pred[a] := b  
sonst pred[b] := a  
    wenn rang[a] = rang[b]  
        dann rang[a] := rang[a] + 1
```


Übung: Zeigen Sie mittels vollständiger Induktion über die Höhe h eines Baums, dass die Höhe der so entstehenden Bäume nur logarithmisch in der Anzahl der Knoten ist. Also: Ein Baum der Höhe h hat mindestens 2^h viele Knoten.

Folgerung: Die Laufzeit einer `findSet`-Operation ist logarithmisch in der Anzahl der Knoten beschränkt.

Laufzeit:

- | | |
|--|---|
| • Initialisierung: | $\mathcal{O}(\mathcal{V})$ |
| • Sortieren der Kanten: | $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{E}))$ |
| • Schleifendurchläufe mal Aufwand <code>findSet</code> : | $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$ |
| • Gesamtlaufzeit: | $\overline{\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))}$ |

Übung: Zeigen Sie, dass $\mathcal{O}(\log(\mathcal{E})) = \mathcal{O}(\log(\mathcal{V}))$ gilt.

Induktion über die Höhe h des Baums.

- Induktionsanfang $h = 0$: Der Baum T besteht nur aus dem Wurzelknoten, also gilt $\text{size}(T) = 1 \geq 2^0 = 1$
- Induktionsschritt $h \rightsquigarrow h + 1$: Dieser Fall kann nur auftreten, wenn zwei gleich hohe Bäume T_1 und T_2 aneinander gehängt werden.

$$\begin{aligned}\text{size}(T) &= \text{size}(T_1) + \text{size}(T_2) \\ &\geq 2^h + 2^h = 2 \cdot 2^h = 2^{h+1} \quad \checkmark\end{aligned}$$

Wir hatten bereits festgestellt, dass $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$ gilt. Also gilt:

$$\begin{aligned}\mathcal{O}(\log(\mathcal{E})) &= \mathcal{O}(\log(\mathcal{V}^2)) \\ &= \mathcal{O}(2 \cdot \log(\mathcal{V})) = \mathcal{O}(\log(\mathcal{V}))\end{aligned}$$

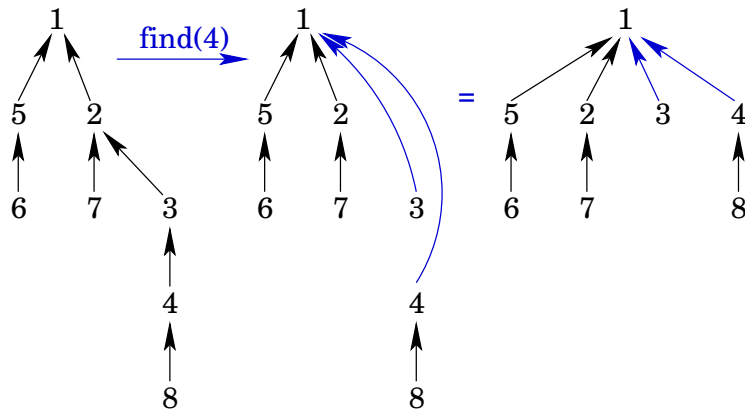
Anmerkung: Kosten der findSet-Operation sind abhängig von Höhe der Bäume.

- Es wäre günstig, alle Knoten direkt an die Wurzel zu hängen.
- Das aber würde die union-Operation teurer machen als bisher.
- Stattdessen: Verkürze während der findSet-Operation die Pfadlängen.

→ findSet(x) mit Pfadkomprimierung:

```
res := x
z := x
while pred[res] ≠ 0 do
    res = pred[res]
while pred[z] ≠ res do
    tmp := z
    z := pred[z]
    pred[tmp] := res
return res
```

Beispiel:



- Die Pfadkomprimierung macht die `findSet`-Methode ungefähr doppelt so teuer wie vorher, die asymptotische Laufzeit wird nicht größer.
- Eine amortisierte Laufzeitanalyse liefert: Kruskals Algorithmus hat für alle praktischen Eingaben eine lineare Laufzeit.

Übung: Welche Laufzeiten haben die Operationen

- Vereinigung zweier Mengen $A \cup B$,
- Schnitt $A \cap B$ zweier Mengen und
- Differenz zweier Mengen $A - B$,

wenn die Mengen A und B mittels sortierter, doppelt verketteter Listen abgespeichert werden?

Sortierte Listen: Um ein Element zu finden, muss die Liste einmal durchlaufen werden. Einhängen, Anhängen und Entfernen eines Elements erfolgt dann in konstanter Zeit.

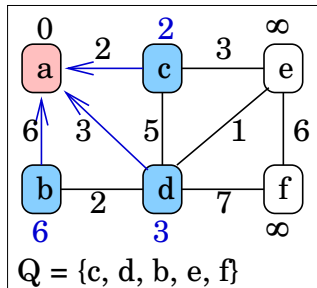
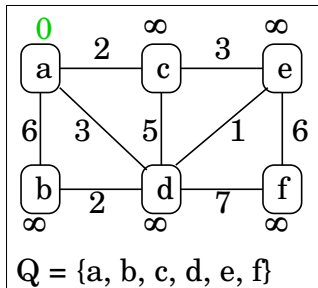
- Vereinigung: Merge Algorithm for two Sorted Sequences
 - Durchlaufe mit jeweils einem Zeiger die sortierten Listen.
 - Falls die Zeiger auf verschiedene Werte zeigen, nimm den kleineren Wert in die Ergebnis-Liste auf und setze den entsprechenden Zeiger auf das nächste Element.
 - Zeigen beide Zeiger auf das gleiche Element, nimm das Element nur einmal in die Ergebnis-Liste auf und setze beide Zeiger weiter.
 - Laufzeit $\mathcal{O}(|A| + |B|)$
- Schnittmenge: Intersection Algorithm for two Sorted Lists
 - Durchlaufe mit jeweils einem Zeiger die sortierten Listen.
 - Falls die Zeiger auf verschiedene Werte zeigen, verwirf den kleineren Wert und setze den entsprechenden Zeiger auf das nächste Element.
 - Zeigen beide Zeiger auf das gleiche Element, nimm das Element nur einmal in die Ergebnis-Liste auf und setze beide Zeiger weiter.
 - Laufzeit $\mathcal{O}(|A| + |B|)$.
- Differenz: analog erhalten wir als Laufzeit $\mathcal{O}(|A| + |B|)$.

Sei Q eine Datenstruktur zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

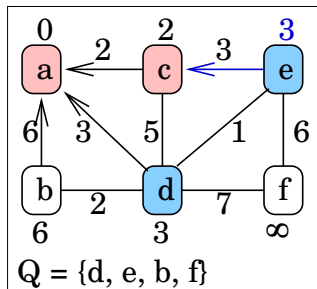
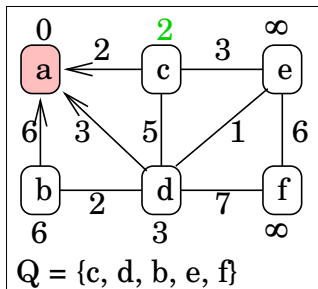
```
Q := V
key[v] := ∞ for all v ∈ V
key[s] := 0 for some arbitrary s ∈ V
while Q ≠ ∅ do
    u := ExtractMin(Q)
    for all v ∈ Adj(u) do
        if v ∈ Q and c((u, v)) < key[v]
        then key[v] := c((u, v))
            π[v] := u
```

- Q nennt man Vorrang-Warteschlange oder priority queue.
- $\pi[v]$ speichert den Vorgänger (predecessor) des Knoten v .

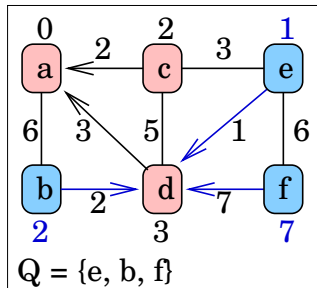
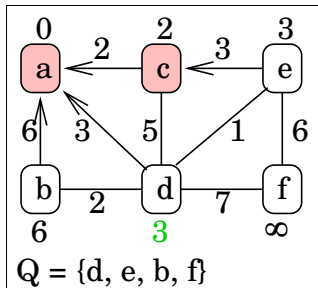
Prims Algorithmus: Beispiel



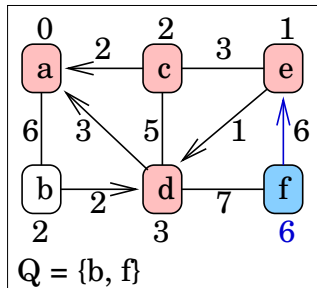
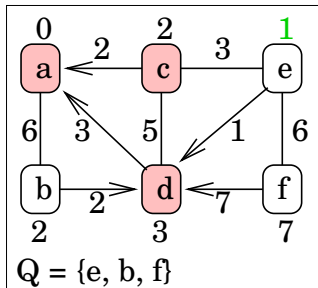
Prims Algorithmus: Beispiel



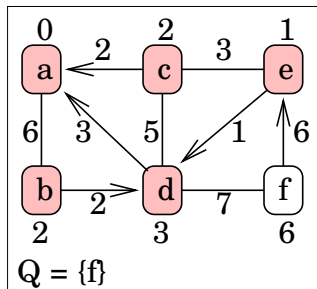
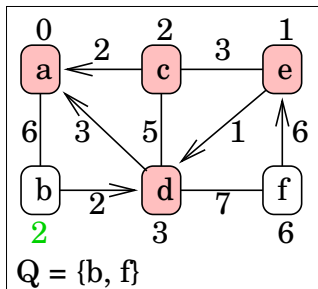
Prims Algorithmus: Beispiel



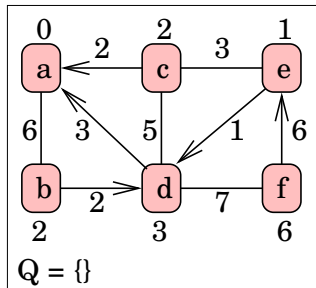
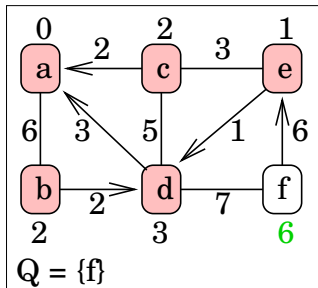
Prims Algorithmus: Beispiel



Prims Algorithmus: Beispiel



Prims Algorithmus: Beispiel



Laufzeit:

$$T = \Theta(\mathcal{V}) \cdot T_{ExtractMin} + \Theta(\mathcal{E}) \cdot T_{DecreaseKey}$$

Je nach verwendeter Datenstruktur ergeben sich unterschiedliche Laufzeiten:

- implementiere Q mittels Array:
 - $T_{ExtractMin} \in \mathcal{O}(\mathcal{V})$
 - $T_{DecreaseKey} \in \mathcal{O}(1)$
 - \Rightarrow Laufzeit $\mathcal{O}(\mathcal{V}^2)$
- implementiere Q als Binär-Heap:
 - $T_{ExtractMin} \in \mathcal{O}(\log(\mathcal{V}))$
 - $T_{DecreaseKey} \in \mathcal{O}(\log(\mathcal{V}))$
 - \Rightarrow Laufzeit $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$
- implementiere Q als Fibonacci-Heap:
 - $T_{ExtractMin} \in \mathcal{O}(\log(\mathcal{V}))$
 - $T_{DecreaseKey} \in \mathcal{O}(1)$
 - \Rightarrow Laufzeit $\mathcal{O}(\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))$

Graphalgorithmen:

- Grundlagen
- Tiefen- und Breitensuche
- Zusammenhangsprobleme
- Minimale Spannbäume
- *Kürzeste Wege*
- Netzwerkfluss
- Matching-Probleme

Gegeben: Ein (un-)gerichteter, zusammenhängender Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \rightarrow \mathbb{R}^+$.

Varianten:

- *Single source shortest paths:* Suche von einem gegebenen Knoten $s \in V$ die kürzesten Wege zu allen anderen Knoten.
→ Algorithmen von Dijkstra oder Bellman/Ford
- *All pairs shortest paths:* Suche für jedes Paar $(u, v) \in V \times V$ den kürzesten Weg von u nach v .
→ Floyd/Warshall Algorithmus

Motivation:

- *Reiseplanung*: Finde den kürzesten Weg zum Urlaubsort.
- *Kostenminimierung*: Finde Zugverbindung
 - mit möglichst kurzer Reisezeit,
 - bei der man möglichst wenig umsteigen muss, oder
 - die möglichst preiswert ist.

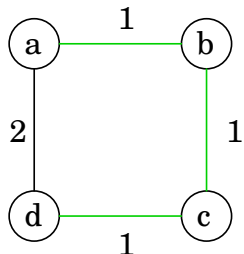
⇒ z.B. Fahrplanberechnung der Deutschen Bahn AG
- *Routing im Internet*: OSPF (Open Shortest Path First)

Anmerkung: Das Problem, einen einfachen längsten Weg zu finden, ist NP-vollständig.

Reduktion: Hamilton Path \leq_p Longest Path

Graph $G = (V, E)$ contains a hamilton path if and only if there is a simple longest path of length $|V| - 1$.

Warum benötigen wir einen neuen Algorithmus zur Berechnung kürzester Wege? Sind die kürzesten Wege nicht bereits durch einen minimalen Spannbaum gegeben?

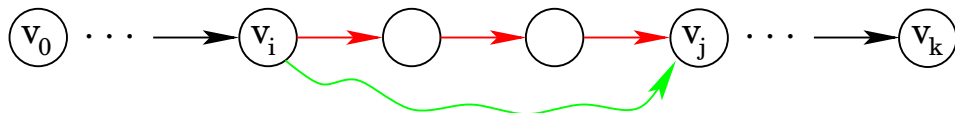


- Der minimale Spannbaum im neben stehenden Graphen ist eindeutig und durch die grün markierten Kanten gekennzeichnet.
- Der kürzeste Weg von a nach d ist allerdings durch die Kante $\{a, d\}$ gegeben, die nicht zum minimalen Spannbaum gehört.

Optimale Sub-Struktur: Ein Teilweg eines kürzesten Weges ist ebenfalls ein kürzester Weg.

Theorem: Sei $(v_0, v_1, v_2, \dots, v_k)$ ein kürzester Weg von v_0 nach v_k . Dann ist jeder Teilweg von v_i nach v_j für $0 \leq i < j \leq k$ auch ein kürzester Weg von v_i nach v_j .

Beweis: Cut and paste.



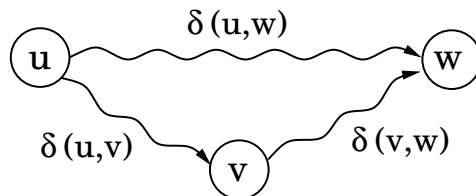
Gäbe es einen kürzeren Weg von v_i nach v_j , dann könnte auch der Weg von v_0 nach v_k verkürzt werden. ⚡⚡

Bezeichne $\delta(u, v)$ die Länge des kürzesten Wegs von u nach v . Falls kein Weg von u nach v existiert, gelte $\delta(u, v) = \infty$.

Dreiecksungleichung: Für alle Knoten $u, v, w \in V$ gilt:

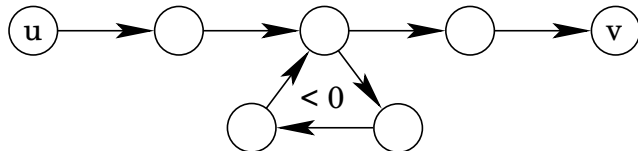
$$\delta(u, w) \leq \delta(u, v) + \delta(v, w)$$

Beweis:



Wäre $\delta(u, v) + \delta(v, w) < \delta(u, w)$, dann wäre $\delta(u, w)$ nicht die Länge des kürzesten Weges von u nach w .

Enthält ein Graph einen Kreis negativer Länge, dann existieren einige kürzeste Wege nicht, da der Kreis bei jedem Durchlaufen die Weglänge verkürzt.



Bezeichnungen hier wie bisher:

- Eine endliche Folge von Knoten $p = (v_1, v_2, \dots, v_m)$ heißt Weg, falls $(v_{i-1}, v_i) \in E$ für $i \in \{2, \dots, m\}$.
- Ein Weg heißt einfach, wenn alle v_i paarweise verschieden sind, wenn also kein Knoten mehrfach vorkommt.
- Suchen wir *einfache* Wege, dann sind diese auch bei Kreisen negativer Länge wohldefiniert.

Manchmal werden Wege anders definiert:

Eine endliche Folge von Kanten $(u_0, v_0), \dots, (u_m, v_m)$ mit $v_{i-1} = u_i$ für alle $i \in \{1, \dots, m\}$ nennt man Weg, wenn keine Kante mehrfach vorkommt.

Bei einer solchen Definition existieren auch dann kürzeste Wege, wenn es Kreise gibt, auf denen die Summe der Kantengewichte negativ ist. Denn solche Kreise dürften nicht mehrfach durchlaufen werden, wenn ein kürzester Weg gesucht ist.

Kürzeste (einfache) Wege ist NP-vollständig!

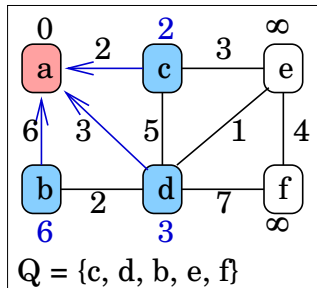
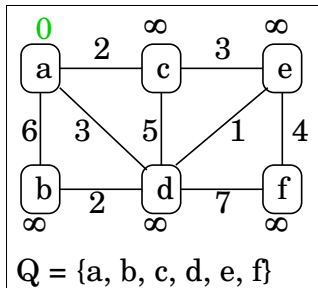
Reduktion: Längste (einfache) Wege \leq_p kürzeste (einfache) Wege

Multipliziere alle Kantengewichte mit -1 , dann wird aus einem längsten Weg ein kürzester Weg.

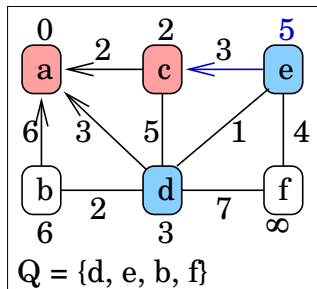
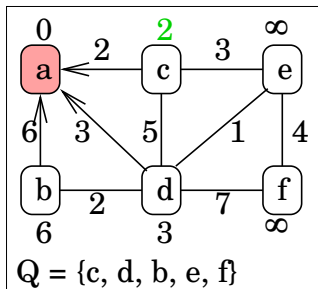
Single source shortest paths: Sei Q eine Datenstruktur zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

```
Q := V
S := ∅
d[v] := ∞ for all v ∈ V
d[s] := 0 for some arbitrary s ∈ V
while Q ≠ ∅ do
    u := ExtractMin(Q)
    S := S ∪ {u}
    for all v ∈ Adj(u) do
        if v ∈ Q and d[v] > d[u] + c((u, v))
        then d[v] := d[u] + c((u, v))
            π[v] := u
```

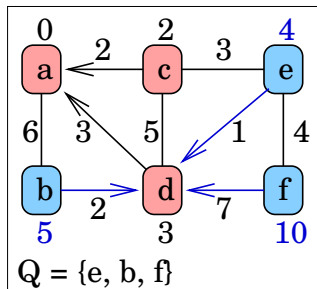
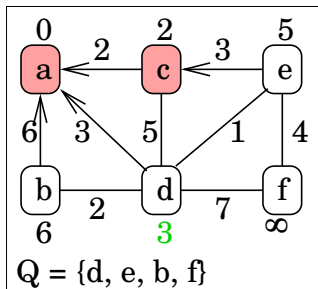
Dijkstras Algorithmus: Beispiel



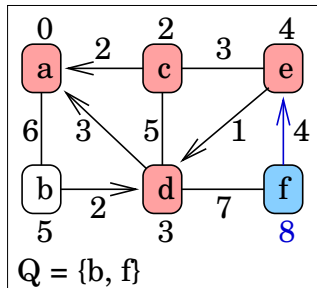
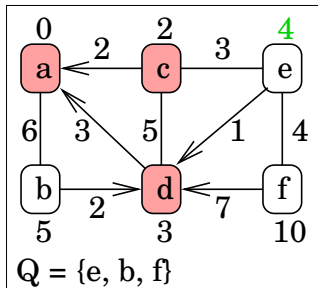
Dijkstras Algorithmus: Beispiel



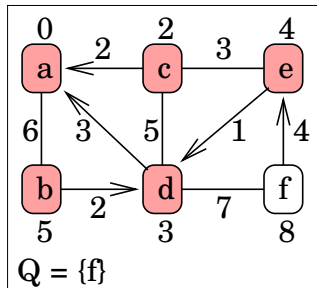
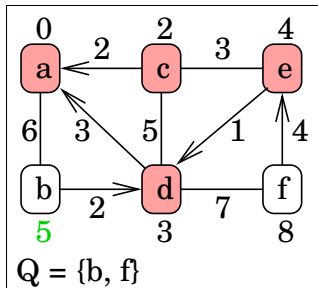
Dijkstras Algorithmus: Beispiel



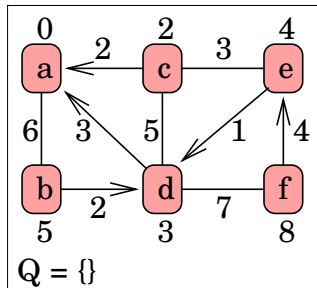
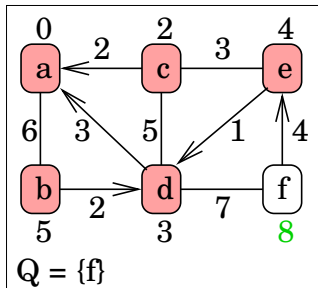
Dijkstras Algorithmus: Beispiel



Dijkstras Algorithmus: Beispiel



Dijkstras Algorithmus: Beispiel



Laufzeit: Gleiche Laufzeit wie Prim's Algorithmus!

$$T = \Theta(\mathcal{V}) \cdot T_{ExtractMin} + \Theta(\mathcal{E}) \cdot T_{DecreaseKey}$$

Implementiere Q als

- Array: Laufzeit $\mathcal{O}(\mathcal{V}^2)$
- Binär-Heap: Laufzeit $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$
- Fibonacci-Heap: Laufzeit $\mathcal{O}(\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))$

Ungewichtete Graphen: Wird die Länge eines Weges durch die Anzahl der Kanten bestimmt, reicht eine modifizierte Breitensuche zur Bestimmung der kürzesten Wege. Es müssen nur die Vorgänger abgespeichert werden, um die Wege zu markieren.

Lemma: Während der Ausführung des Algorithmus gilt stets $d[v] \geq \delta(s, v)$ für alle $v \in V$.

Beweis durch Widerspruch:

- Sei v der Knoten, für den während der Ausführung des Algorithmus zum ersten Mal $d[v] < \delta(s, v)$ gilt.
- Außerdem sei u der Knoten, durch den $d[v]$ verkleinert wurde:
 $d[v] := d[u] + c((u, v))$

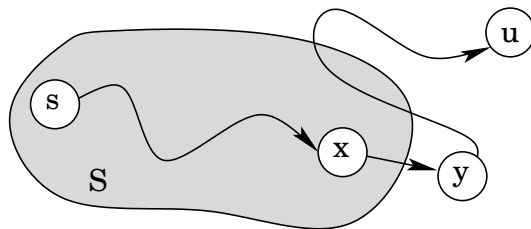
Dann gilt:

$d[v]$	$<$	$\delta(s, v)$	aufgrund der Annahme
	\leq	$\delta(s, u) + \delta(u, v)$	wegen der Dreiecksungleichung
	\leq	$\delta(s, u) + c((u, v))$	kürzester Weg \leq konkreter Weg
	\leq	$d[u] + c((u, v))$	v verletzt Invariante zum 1. Mal

Theorem: Dijkstras Algorithmus liefert $d[v] = \delta(s, v)$ für alle Knoten $v \in V$.

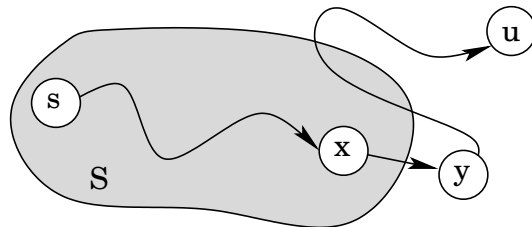
Beweis: Wir zeigen, dass $d[v] = \delta(s, v)$ gilt, wenn v zur Menge S hinzu genommen wird. Da die Werte von $d[v]$ nur kleiner werden können, ist damit die Aussage gezeigt.

Angenommen, u ist der erste Knoten, der zu S hinzu genommen wird, für den $d[u] \neq \delta(s, u)$ gilt.



Sei y der erste Knoten aus $V - S$ auf einem kürzesten Weg von s nach u , und sei x der Vorgänger von y auf diesem Weg.

(Fortsetzung)



- Es gilt $d[x] = \delta(s, x)$, da u der erste Knoten ist, der die Invariante verletzt.
- Da Teilwege von kürzesten Wegen auch kürzeste Wege sind, wurde $d[y]$ auf $\delta(s, x) + c((x, y)) = \delta(s, y)$ gesetzt, als die Kante (x, y) nach Hinzunahme von x zu S untersucht wurde.

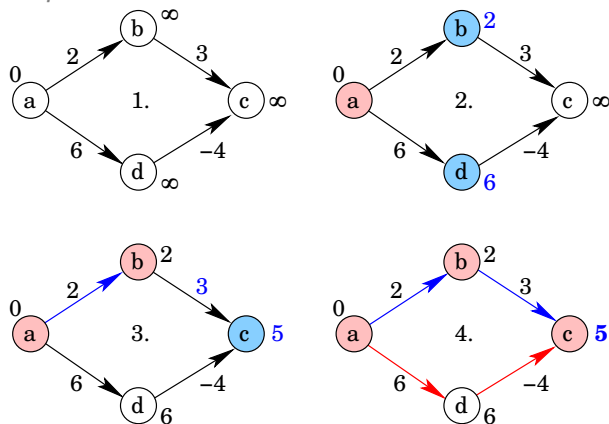
Also gilt $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$, denn $c : E \rightarrow \mathbb{R}^+$.

- Aber es gilt $d[u] \leq d[y]$, weil der Algorithmus u wählt.
- Also: $d[y] = \delta(s, y) = \delta(s, u) = d[u]$ ⚡⚡

Dijkstras Algorithmus

Anmerkung: Dijkstras Algorithmus arbeitet nur korrekt, wenn im Graphen keine negativen Kantengewichte existieren.

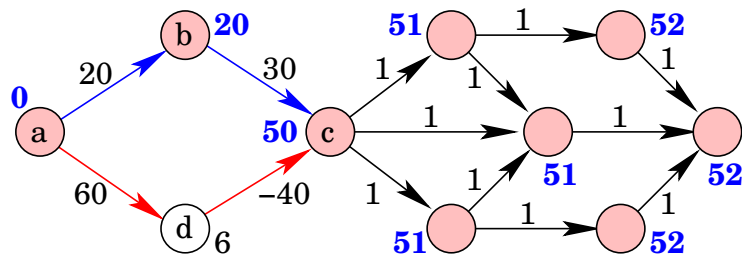
Beispiel:



Frage:

Warum erweitern wir Dijkstra nicht so, dass die Distanz bei c geändert wird, wenn ein kürzerer Weg gefunden wird?

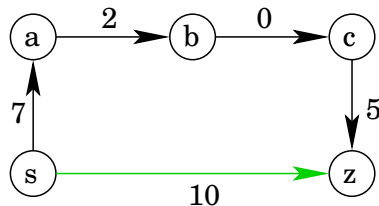
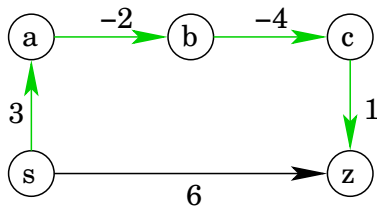
Antwort: Alle von *c* aus erreichbaren und schon berechneten kürzesten Wege müssen erneut berechnet werden. → Laufzeit steigt enorm



Weitere Frage: Wie werden Endlosschleifen bei Kreisen negativer Länge verhindert?

Frage: Warum addieren wir nicht einfach den Betrag des kleinsten Kantengewichts auf alle Kantengewichte auf und berechnen dann mittels Dijkstras Algorithmus die kürzesten Wege?

Weil es nicht funktioniert, wie folgendes Beispiel zeigt.



- Der kürzeste Weg zwischen Startknoten s und Zielknoten z ist jeweils grün markiert.
- Links ist der Originalgraph zu sehen, im rechten Graphen wurde auf jedes Kantengewicht der Wert 4 addiert.

Lösung des Single-Source-Shortest-Paths-Problems:

- Im Gegensatz zu Dijkstras Algorithmus sind negative Kantengewichte erlaubt.
- gegeben: Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \rightarrow \mathbb{R}$ und ein Startknoten $s \in V$.

```
d[v] := ∞ for all v ∈ V
d[s] := 0 for some arbitrary s ∈ V
for i := 1 to V - 1 do
    for each edge (u, v) ∈ E do
        if d[v] > d[u] + c((u, v))
            then d[v] := d[u] + c((u, v))
for each edge (u, v) ∈ E do
    if d[v] > d[u] + c((u, v))
        then report: „negative-weight cycle exists“
```

Anmerkung:

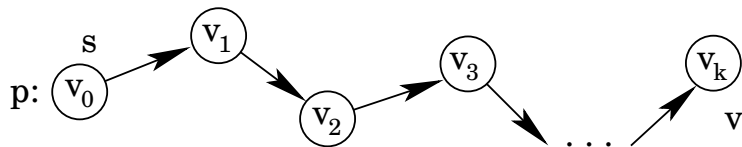
- Ein kürzester Weg, der keine negativen Kreise enthält, besucht keinen Knoten zweimal.
- Daher besteht ein solcher Weg aus höchstens $\mathcal{V} - 1$ Kanten.

Laufzeit:

- Die äußere Schleife wird $(\mathcal{V} - 1)$ -mal durchlaufen.
 - Die innere Schleife wird \mathcal{E} -mal durchlaufen.
 - Alle Operationen der inneren Schleife kosten Zeit $\mathcal{O}(1)$.
- Gesamte Laufzeit in $\Theta(\mathcal{V} \cdot \mathcal{E})$.

Korrektheit: Der Graph G enthalte keine negativen Kreise.

- Sei $v \in V$ ein beliebiger Knoten, und betrachte einen kürzesten Weg p von s nach v mit minimaler Anzahl Kanten.



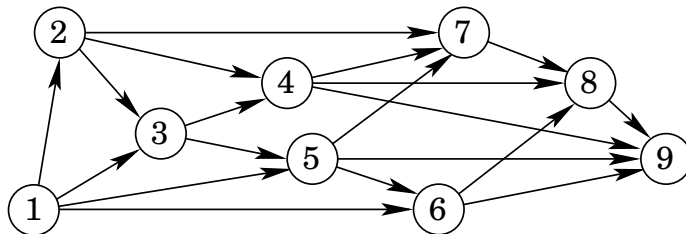
Da p kürzester Weg ist: $\delta(s, v_i) = \delta(s, v_{i-1}) + c((v_{i-1}, v_i))$.

- Initial gilt: $d[v_0] = 0 = \delta(s, v_0)$
- Wir betrachten die Durchläufe der äußeren Schleife:
 - nach 1. Durchlauf durch E gilt: $d[v_1] = \delta(s, v_1)$.
 - nach 2. Durchlauf durch E gilt: $d[v_2] = \delta(s, v_2)$.
 - \vdots
 - nach k . Durchlauf durch E gilt: $d[v_k] = \delta(s, v_k)$.
- Ein kürzester Weg besteht aus höchstens $\mathcal{V} - 1$ Kanten.

Anmerkung: Wenn ein Wert $d[v]$ nach $V - 1$ Durchläufen der äußeren Schleife nicht konvergiert, dann enthält Graph G einen negativen Kreis, der von s aus erreichbar ist.

Berechnung kürzester Wege in gerichteten, azyklischen Graphen in Zeit $\Theta(V + E)$:

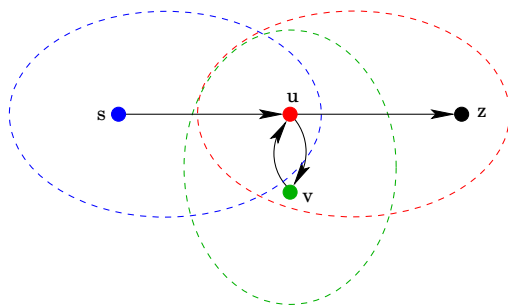
- Berechne topologische Sortierung der Knoten.



- Durchlaufe die Knoten $u \in V$ entsprechend dieser Reihenfolge und relaxiere die Kanten von $v \in Adj[u]$: falls $d[v] > d[u] + c(u, v)$ dann $d[v] := d[u] + c(u, v)$

Unsere Voraussetzung, dass ein kürzester Weg, der keine negativen Kreise enthält, keinen Knoten zweimal besucht, ist nicht immer richtig, siehe Elektromobilität:

- Positive Kantengewichte stellen einen Energieverbrauch dar, negative Kantengewichte entsprechen einem Energiegewinn, z.B. weil bergab gefahren wird.
- Kreise mit negativer Länge kann es natürlich nicht geben, sonst wären unsere Energieprobleme gelöst.
- Wenn es schnell (v) und langsam (u) ladende Strom-Tankstellen gibt, kann es sinnvoll sein, einen Knoten doppelt zu besuchen.



Die gestrichelten Linien geben jeweils die Grenzen an, bis zu denen eine voll geladene Batterie ausreicht.

Lade bei u nur so viel auf, dass v erreicht wird. Lade bei v voll auf. Lade bei u so viel nach, dass z erreicht wird.

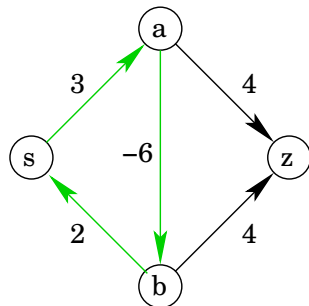
All pairs shortest paths:

Gegeben: Gewichteter Graph $G = (V, E, c)$ mit $c : E \rightarrow \mathbb{R}$.

Achtung: Negative Kantengewichte sind erlaubt!

Gesucht: $\forall u, v \in V$ der kürzeste Weg von u nach v .

Voraussetzung: G enthält keine negativen Kreise!



Im nebenstehenden Graphen hat der grün markierte Kreis (s, a, b) negative Länge.

Daher kann der kürzeste Weg von s nach z immer weiter verkleinert werden, indem der Kreis ein weiteres Mal durchlaufen wird.

In ungerichteten Graphen stellt bereits eine einzige negative Kante einen Kreis negativer Länge dar.

Idee: Sei d_{ij}^k die Länge eines kürzesten Weges von i nach j , der nur über Knoten mit Nummern kleiner gleich k läuft. Dann gilt:

- $d_{ij}^0 = c(i, j)$
- $d_{ij}^k = \min \{ d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1} \}$

Algorithmus: dynamische Programmierung, sei $V = \{1, 2, \dots, n\}$.

```

 $d[i, j] := \infty$  for each  $i, j \in V, i \neq j$ 
 $d[i, i] := 0$  for each  $i \in V$ 
 $d[i, j] := c(i, j)$  for each  $(i, j) \in E$ 
for  $k := 1$  to  $n$  do
  for  $i := 1$  to  $n$  do
    for  $j := 1$  to  $n$  do
      if  $d[i, j] > d[i, k] + d[k, j]$ 
         $d[i, j] := d[i, k] + d[k, j]$ 

```

Anmerkungen:

- Laufzeit: $\mathcal{O}(V^3)$
- Wie wird ein Kreis negativer Länge erkannt?
Ein negativer Wert d_{ii} auf der Diagonalen bedeutet, dass mindestens ein Kreis negativer Länge vorhanden ist.
- Bei Graphen ohne negative Kanten könnte auch für jede Quelle ein Aufruf von Dijkstra durchgeführt werden. Warum wird das nicht gemacht?
 - Der Programmieraufwand wäre größer, da Dijkstra eine Priority-Queue benötigt.
 - Die in der Groß-O-Notation verborgenen Konstanten sind viel größer als bei Floyd/Warshalls Algorithmus.
- Warum wird kein 3D-Array d benötigt?
 - Weil in einer Runde k immer nur auf die Werte der letzten Runde $k - 1$ zugegriffen wird. Bei der Matrix-Ketten-Multiplikation oder dem CYK-Algorithmus wurden auch weiter zurück liegende Teilergebnisse benötigt, hier nicht.

Transitiver Abschluss

Gegeben: Ein gerichteter Graph $G = (V, E)$.

Gesucht: Transitiver Abschluss $G^* = (V, E^*)$ von G .

In G^* existiert eine Kante (v_i, v_j) genau dann, wenn in G ein Weg zwischen v_i und v_j existiert. Sei auch hier $V = \{1, 2, \dots, n\}$.

```
tii := 1
tij := 1 for each (vi, vj) ∈ E
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      tij := tij ∨ (tik ∧ tkj)
```

→ Laufzeit: $\mathcal{O}(n^3)$

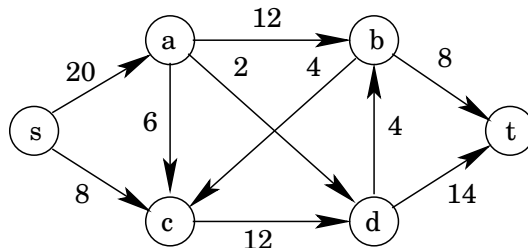
Graphalgorithmen:

- Grundlagen
- Tiefen- und Breitensuche
- Zusammenhangsprobleme
- Minimale Spannbäume
- Kürzeste Wege
- *Netzwerkfluss*
- Matching-Probleme

- Gegeben:**
- Ein gewichteter Graph $G = (V, E, c)$ mit einer Kostenfunktion $c : E \rightarrow \mathbb{Q}^+$.
 - Eine Quelle $s \in V$ und eine Senke $t \in V$, wobei $s \neq t$ gilt.

Gesucht: Maximaler Fluss im Netzwerk von Quelle s nach Senke t .

Beispiel:



Eine *Flussfunktion* f eines Netzwerks $H = (V, E, c, \mathbb{Q}^+, s, t)$ ist eine Funktion $f : E \rightarrow \mathbb{Q}^+$ mit

- **Kapazitätsbeschränkung:** $\forall e \in E : 0 \leq f(e) \leq c(e)$
- **Flusserhaltung:** $\forall v \in V - \{s, t\} :$

$$\sum_{e \in \text{In}(v)} f(e) = \sum_{e \in \text{Out}(v)} f(e)$$

Der *Fluss* $F(f)$ im Netzwerk $H = (V, E, c, \mathbb{Q}^+, s, t)$ ist definiert als

$$F(f) = \sum_{e \in \text{In}(t)} f(e) - \sum_{e \in \text{Out}(t)} f(e)$$

oder analog:

$$F(f) = \sum_{e \in \text{Out}(s)} f(e) - \sum_{e \in \text{In}(s)} f(e)$$

Sei $H = (V, E, c, \mathbb{Q}^+, s, t)$ ein Netzwerk, und sei $S \subset V$ eine Teilmenge der Knoten mit $s \in S$ und $t \notin S$, sei $\bar{S} = V - S$. Dann definieren wir:

- $E(S, \bar{S}) := \{(x, y) \mid x \in S, y \in \bar{S}\} \cap E$
- $E(\bar{S}, S) := \{(x, y) \mid x \in \bar{S}, y \in S\} \cap E$
- Der *Schnitt des Netzwerks* H bzgl. der Knotenmenge S ist $H(S) := E(S, \bar{S}) \cup E(\bar{S}, S)$.

Also: In $H(S)$ sind all die Kanten, die aus der Menge S in die Menge \bar{S} laufen und umgekehrt.

- Die *Kapazität des Schnitts* $H(S)$ ist definiert als

$$c(S) := \sum_{e \in E(S, \bar{S})} c(e).$$

Achtung: Berücksichtigt nur Kanten von S nach \bar{S} .

Lemma:

$$F(f) = \sum_{e \in E(S, \bar{S})} f(e) - \sum_{e \in E(\bar{S}, S)} f(e)$$

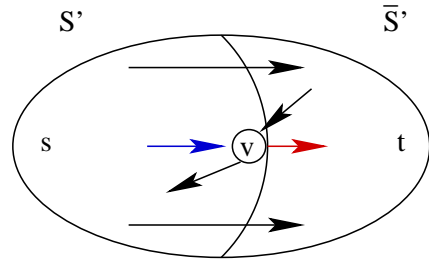
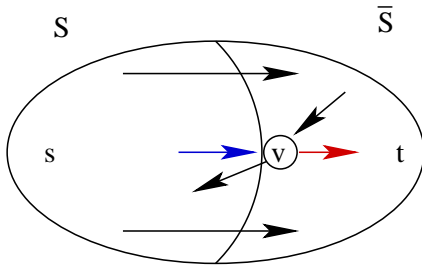
Beweis: Induktion über $|S|$.

Induktionsanfang: $S = \{s\}$

$$\begin{aligned} F(f) &\stackrel{!}{=} \sum_{E(S, \bar{S})} f(e) - \sum_{E(\bar{S}, S)} f(e) \\ &= \sum_{e \in \text{Out}(s)} f(e) - \sum_{e \in \text{In}(s)} f(e) \end{aligned}$$

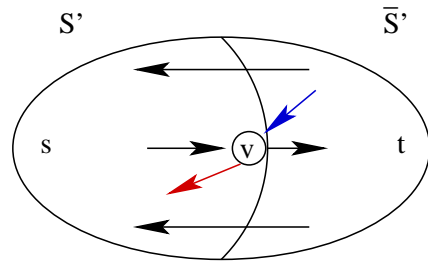
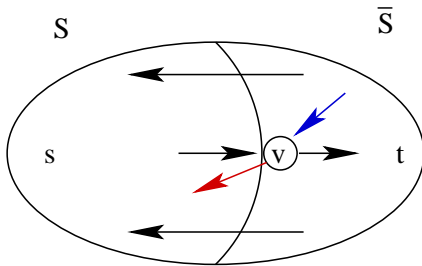
→ O.K.: Gilt nach Definition von Netzwerkfluss.

(1)



$$\sum_{E(S \cup \{v\}, \bar{S} - \{v\})} f(e) = \sum_{E(S, \bar{S})} f(e) - \sum_{\substack{e=(u,v) \in E \\ u \in S}} f(e) + \sum_{\substack{e=(v,w) \in E \\ w \in \bar{S}}} f(e)$$

(2)



$$\sum_{E(\bar{S}-\{v\}, S \cup \{v\})} f(e) = \sum_{E(\bar{S}, S)} f(e) - \sum_{\substack{e=(v,w) \in E \\ w \in S}} f(e) + \sum_{\substack{e=(u,v) \in E \\ u \in \bar{S}}} f(e)$$

$$\sum_{E(S \cup \{v\}, \bar{S} - \{v\})} f(e) = \sum_{E(S, \bar{S})} f(e) - \underbrace{\sum_{\substack{e=(u,v) \in E \\ u \in S}} f(e)}_A + \underbrace{\sum_{\substack{e=(v,w) \in E \\ w \in \bar{S}}} f(e)}_B$$

$$\sum_{E(\bar{S} - \{v\}, S \cup \{v\})} f(e) = \sum_{E(\bar{S}, S)} f(e) - \underbrace{\sum_{\substack{e=(v,w) \in E \\ w \in S}} f(e)}_C + \underbrace{\sum_{\substack{e=(u,v) \in E \\ u \in \bar{S}}} f(e)}_D$$

Subtrahieren wir die beiden Gleichungen, dann gilt:

$$\begin{aligned} F(f) &\stackrel{!}{=} \sum_{E(S \cup \{v\}, \bar{S} - \{v\})} f(e) - \sum_{E(\bar{S} - \{v\}, S \cup \{v\})} f(e) \\ &= \sum_{E(S, \bar{S})} f(e) - \sum_{E(\bar{S}, S)} f(e) + (B + C) - (A + D) \end{aligned}$$

$$A + D = \sum_{\substack{e=(u,v) \in E \\ u \in S}} f(e) + \sum_{\substack{e=(u,v) \in E \\ u \in \bar{S}}} f(e) = \sum_{e \in \text{In}(v)} f(e)$$

$$B + C = \sum_{\substack{e=(v,w) \in E \\ w \in \bar{S}}} f(e) + \sum_{\substack{e=(v,w) \in E \\ w \in S}} f(e) = \sum_{e \in \text{Out}(v)} f(e)$$

Da für jeden Knoten $v \in V - \{s, t\}$ die Flusserhaltung

$$\sum_{e \in \text{In}(v)} f(e) = \sum_{e \in \text{Out}(v)} f(e)$$

gilt, erhalten wir $(B + C) - (A + D) = 0$.

Also: Der Fluss über den alten Schnitt ist gleich dem Fluss über den neuen Schnitt.

Satz: Für jede Teilmenge $S \subset V$ mit $s \in S$ und $t \notin S$ gilt:

$$F(f) \leq c(S)$$

Beweis:

$$\begin{aligned} F(f) &\stackrel{\text{Lemma}}{=} \sum_{E(S, \bar{S})} f(e) - \sum_{E(\bar{S}, S)} f(e) \\ 0 \leq f(e) &\leq c(e) \leq \sum_{E(S, \bar{S})} c(e) - \sum_{E(\bar{S}, S)} f(e) \\ &\stackrel{\text{Definition}}{=} c(S) - \sum_{E(\bar{S}, S)} f(e) \\ 0 \leq f(e) &\leq c(S) \end{aligned}$$

Wegen $F(f) \leq c(S)$ gilt also insbesondere:

$$\max_f \{F(f)\} \leq \min_S \{c(S)\}$$

Damit erhalten wir das *Max-Flow Min-Cut Theorem*:

$$F(f) = c(S) \quad \implies \quad F(f) \text{ ist maximal, } c(S) \text{ ist minimal}$$

Definition: Ein *Pfad* in einem Netzwerk $G = (V, E)$ ist eine Knotenfolge $P = u_1, \dots, u_k$, so dass für jedes Knotenpaar u_i, u_{i+1} , $1 \leq i \leq k - 1$, entweder (u_i, u_{i+1}) oder (u_{i+1}, u_i) eine gerichtete Kante in G ist.

- (u_i, u_{i+1}) ist eine *Vorwärtskante* und
- (u_{i+1}, u_i) eine *Rückwärtskante* im Pfad P .

Idee der meisten Flussalgorithmen:

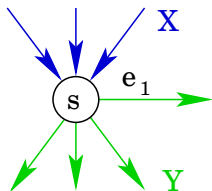
- ① Starte mit beliebiger Flussfunktion f , z.B. $f(e) := 0 \ \forall e \in E$
- ② Suche Pfad $P = u_1, \dots, u_k$ mit $u_1 = s, u_k = t$ und
 - für alle Vorwärtskanten $e = (u_i, u_{i+1})$ gilt: $f(e) < c(e)$
sei $\Delta(e) := c(e) - f(e)$
 - für alle Rückwärtskanten $e = (u_{i+1}, u_i)$ gilt: $f(e) > 0$
sei $\Delta(e) := f(e)$

Sei $\Delta(P)$ das kleinste $\Delta(e)$ über alle Kanten e im Pfad P .

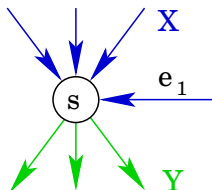
- ③ Erhöhe den Fluss um $\Delta(P)$.
 - für eine Vorwärtskante e also: $f'(e) := f(e) + \Delta(P)$
 - für eine Rückwärtskante e also: $f'(e) := f(e) - \Delta(P)$
- ④ Wiederhole die Schritte 2 und 3 solange es einen Pfad P mit $\Delta(P) > 0$ gibt.

Aufgrund der Wahl von $\Delta(P)$ ist klar, dass in Schritt 3 die Kapazitätsbeschränkung pro Kante eingehalten wird.

Um zu zeigen, dass der Fluss vergrößert wurde, betrachten wir zwei Fälle. Sei e_1 die zu s inzidente Kante, deren Fluss geändert wurde.



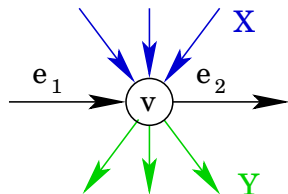
$$\begin{aligned}
 F(f') &= \sum_{e \in \text{Out}(s)} f'(e) - \sum_{e \in \text{In}(s)} f'(e) \\
 &= \sum_{e \in Y} f(e) + f'(e_1) - \sum_{e \in X} f(e) \\
 &= \sum_{e \in Y} f(e) + f(e_1) + \Delta(P) - \sum_{e \in X} f(e) \\
 &= F(f) + \Delta(P) > F(f)
 \end{aligned}$$



$$\begin{aligned}
 F(f') &= \sum_{e \in \text{Out}(s)} f'(e) - \sum_{e \in \text{In}(s)} f'(e) \\
 &= \sum_{e \in Y} f(e) - \left(\sum_{e \in X} f(e) + f'(e_1) \right) \\
 &= \sum_{e \in Y} f(e) - \left(\sum_{e \in X} f(e) + f(e_1) - \Delta(P) \right) \\
 &= F(f) + \Delta(P) > F(f)
 \end{aligned}$$

Bei der Flusserhaltung im Knoten v unterscheiden wir vier Fälle. Seien e_1 und e_2 die zu v inzidenten Kanten, deren Fluss geändert wurde.

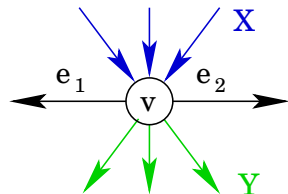
Fall 1:



Über e_1 fließt zusätzlich $\Delta(P)$ in den Knoten v ein, was über e_2 aber auch wieder abfließt.

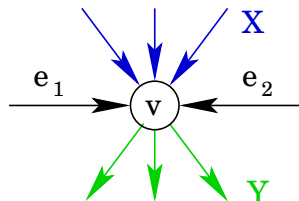
$$\begin{aligned}\sum_{e \in \text{In}(v)} f'(e) &= \sum_{e \in \text{In}(v)} f(e) + \Delta(P) \\ \sum_{e \in \text{Out}(v)} f'(e) &= \sum_{e \in \text{Out}(v)} f(e) + \Delta(P)\end{aligned}$$

Fall 2:



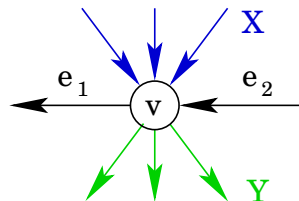
Bei der Kante e_1 wird der Fluss um $\Delta(P)$ verringert, bei der Kante e_2 um $\Delta(P)$ vergrößert, so dass in Summe bei den ausgehenden Kanten keine Änderung erfolgt.

Fall 3:



Bei der Kante e_1 wird der Fluss um $\Delta(P)$ vergrößert, bei der Kante e_2 um $\Delta(P)$ verringert, so dass in Summe bei den einlaufenden Kanten keine Änderung erfolgt.

Fall 4:



Über e_2 fließt zusätzlich $\Delta(P)$ in den Knoten v ein, was über e_1 aber auch wieder abfließt.

$$\sum_{e \in \text{In}(v)} f'(e) = \sum_{e \in \text{In}(v)} f(e) + \Delta(P)$$

$$\sum_{e \in \text{Out}(v)} f'(e) = \sum_{e \in \text{Out}(v)} f(e) + \Delta(P)$$

Ein Pfad P mit $\Delta(P) > 0$ ist ein *zunehmender Pfad*.

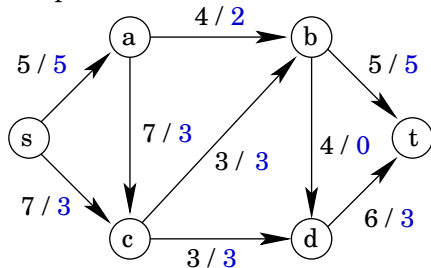
Sei $G_R(f) = (V, E_R)$ der *Restgraph* zu f , der alle noch möglichen Flussvergrößerungen beschreibt. Für jede Kante $e \in E$ von u nach v gibt es im Restgraphen

- eine Kante von u nach v mit Gewicht $c(e) - f(e)$, falls $c(e) > f(e)$,
- und eine Kante von v nach u mit Gewicht $f(e)$, falls $f(e) > 0$.

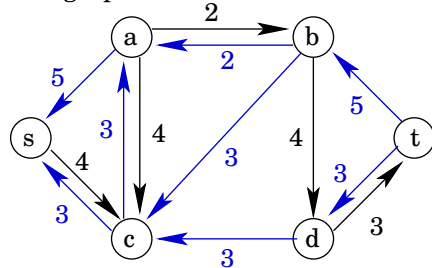
Bemerkung: Es müssen auch rückwärtsgerichtete Kanten betrachtet werden, damit eine Flussvergrößerung berechnet werden kann, siehe dazu im Beispiel auf der nächsten Folie den Graphen G . Es findet sich dort kein Weg, der nur Vorwärtskanten enthält, bei dem der Fluss vergrößert werden könnte.

Beispiel:

Graph:



Restgraph:



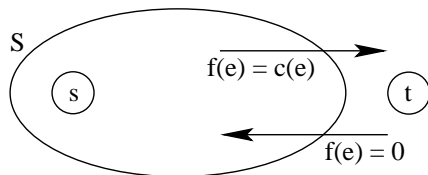
Bemerkung: Jeder Weg von s nach t im Restgraphen $G_R(f)$ ist ein zunehmender Pfad in G .

hier: $P = s \xrightarrow{4} c \xrightarrow{3} a \xrightarrow{2} b \xrightarrow{4} d \xrightarrow{3} t$ mit $\Delta(P) = 2$

Satz: $F(f)$ ist maximal \iff in $G = (V, E)$ gibt es keinen zunehmenden Pfad

Beweis:

- “ \Rightarrow “ Klar. Denn gäbe es einen zunehmenden Pfad, dann könnte $F(f)$ vergrößert werden und $F(f)$ wäre nicht maximal.
- “ \Leftarrow “ Wenn es in G keinen zunehmenden Pfad gibt, dann gibt es in $G_R(f)$ keinen Weg von s nach t . Sei S die Menge der Knoten, die in $G_R(f)$ von s aus erreichbar sind, sei \bar{S} die Menge der übrigen Knoten und betrachte den Graphen G :



Dann gilt für jede Kante

- $e \in E(S, \bar{S})$: $f(e) = c(e)$
- $e \in E(\bar{S}, S)$: $f(e) = 0$

Damit ist $F(f) = c(S)$, und somit ist der Fluss maximal.

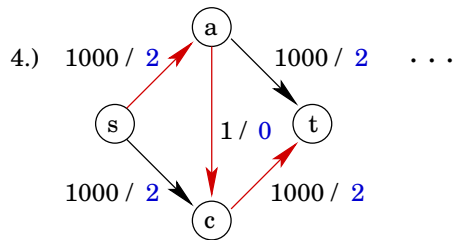
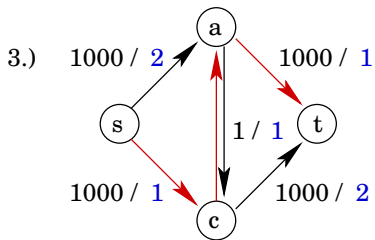
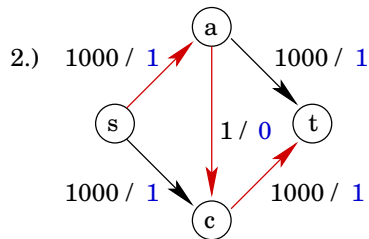
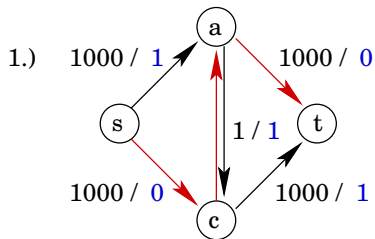
Idee von Ford-Fulkerson (1956): Suche einen beliebigen zunehmenden Pfad P (z.B. mittels Tiefensuche) und erhöhe den Fluss entlang des Pfades um $\Delta(P)$.

Bemerkungen:

- ohne Beweis¹: Der Algorithmus terminiert nicht immer für irrationale Kapazitäten.
- ohne Beweis: Der Algorithmus konvergiert für irrationale Kapazitäten nicht einmal unbedingt gegen F_{\max} .
- Für ganzzahlige Kapazitäten ist die Anzahl der Flussvergrößerungen durch F_{\max} beschränkt.

Daher: Die Anzahl der Flussvergrößerungen ist abhängig von den Kantengewichten, und nicht allein abhängig von der Größe des Graphen!

¹Uri Zwick: The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. Theoretical Computer Science (1995).



Frage: Ist die Laufzeit polynomiell in der Eingabegröße?

Antwort: Die Laufzeit ist **nicht** polynomiell in der Eingabegröße!

Zahlen werden binär kodiert, haben also nur logarithmische Länge bezogen auf den Wert der Zahl. Die Zahl 65.534 kann mit $\lfloor \log_2(65534) \rfloor + 1 = 16$ Bit dargestellt werden.

Die Laufzeit des Algorithmus hängt aber vom Wert der Zahl ab, also in unserem Beispiel:

$$2^{\log_2(65534)} = 65534$$

Die Laufzeit kann also exponentiell sein, abhängig davon, wie groß die Zahlenwerte im Vergleich zur Größe des Graphen sind.

Idee: Wähle immer einen zunehmenden Pfad mit der minimalen Anzahl Kanten, wie ihn bspw. eine Breitensuche liefert.

Lemma: Der Netzwerkfluss-Algorithmus nach Edmonds und Karp benötigt höchstens $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ Iterationen.

Damit erhalten wir als *Laufzeit* insgesamt:

- Anzahl Iterationen: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$
- Breitensuche zum Finden kürzester Pfade: $\mathcal{O}(\mathcal{E})$
- Gesamte Laufzeit: $\mathcal{O}(\mathcal{E}^2 \cdot \mathcal{V})$

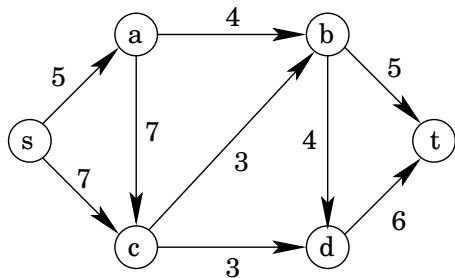
Beweis: Sei d die Distanz von der Quelle s zur Senke t im aktuellen Restgraphen. Wir werden zeigen, dass

- d niemals kleiner wird, und
- nach $\mathcal{O}(\mathcal{E})$ Iterationen der Wert von d um mindestens 1 wächst.

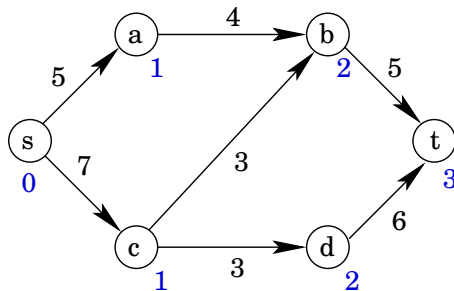
Netzwerkfluss nach Edmonds und Karp

Der dem Restgraphen $G_R = (V, E_R)$ zugeordnete Schichtengraph $L(G_R)$ ist derjenige Teilgraph von G_R , der nur Kanten $(u, v) \in E_R$ enthält, für die $\delta(s, v) = \delta(s, u) + 1$ gilt. $L(G_R)$ enthält für jeden Knoten $v \in V$ alle Pfade von s nach v minimaler Länge.

Restgraph:



Schichtengraph:



Für einen Pfad P minimaler Länge in $G_R(f)$ gilt also: Die Kanten von P liegen in $L(G_R(f))$.

Nun wird der Fluss entlang des Pfades P um $\Delta(P)$ erhöht. Der neue Restgraph $G_R(f')$ unterscheidet sich von $G_R(f)$ dadurch,

- dass mindestens eine Kante auf P gesättigt ist und daher in $G_R(f')$ nicht vorhanden ist.
- dass evtl. weitere, allerdings rückwärts gerichtete Kanten hinzugekommen sind.

Betrachten wir nun einen Pfad Q minimaler Länge in $G_R(f')$:

- Falls Q auch in $L(G_R(f))$ enthalten ist, dann muss Q dieselbe Länge wie P haben.
- Falls Q nicht in $L(G_R(f))$ enthalten ist, muss die Länge von Q um mindestens 1 größer sein als die Länge von P , da nur rückwärts gerichtete Kanten hinzugekommen sind.

Die Pfadlänge wird also nicht kleiner.

Betrachten wir eine Folge von Flussvergrößerungen.

- Ein Restgraph G_R enthält höchstens $2 \cdot \mathcal{E}$ Kanten, da zu jeder Kante des gegebenen Netzwerks G höchstens eine weitere, entgegengesetzt gerichtete Kante hinzukommt.
- Nach spätestens $\mathcal{O}(\mathcal{E})$ Iterationen ist jede Kante entfernt worden, so dass eine rückwärts gerichtete Kante genutzt werden muss und die Pfadlänge um mindestens 1 größer wird.
- Die Distanz von s nach t kann höchstens \mathcal{V} -mal erhöht werden, da ein kürzester Weg keine Knoten doppelt besucht, oder anders gesagt, die maximale Pfadlänge ist $\mathcal{V} - 1$.

Damit haben wir gezeigt, dass der Netzwerkfluss-Algorithmus nach Edmonds und Karp höchstens $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ Iterationen benötigt.

Idee von Dinic: Finde alle kürzesten Wege mit gleicher Anzahl von Kanten in einer Runde.

- 1 Bilde den Schichtengraphen $L(G_R(f))$. Dieser kann mittels Breitensuche in Zeit $\mathcal{O}(\mathcal{E})$ aufgebaut werden.
- 2 Bestimme eine Flussfunktion f' für den Schichtengraphen $L(G_R(f))$, die nicht durch einen zunehmenden Pfad von s nach t erhöhbar ist und addiere f' zu f hinzu.

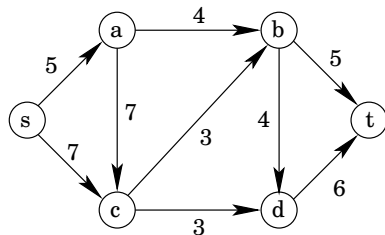
Mit anderen Worten: Jeder Weg von s nach t im Schichtengraphen enthält eine gesättigte Kante. Wir nennen f' einen blockierenden Fluss für den Schichtengraphen.

- 3 Wiederhole Schritt 1 und 2 solange, bis der Fluss f maximal ist, also bis es keinen Weg von s nach t im Schichtengraphen $L(G_R(f))$ gibt.

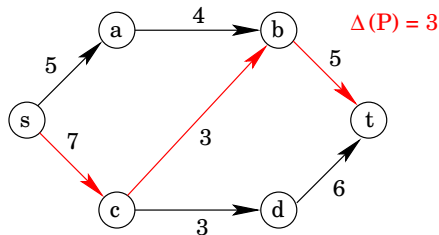
Aber wie findet man einen blockierenden Fluss?

Beispiel: Finden einer nicht-vergrößerbaren Flussfunktion.

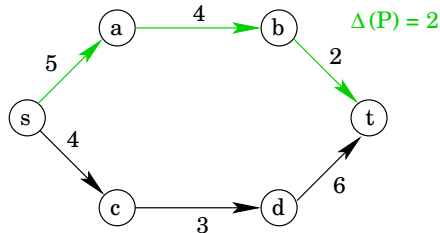
Restgraph:



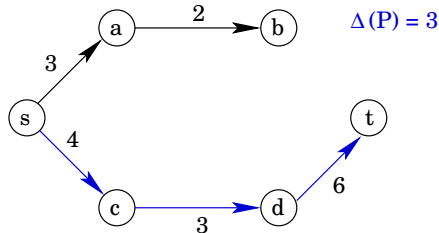
1. Schichtengraph:



2. Schichtengraph:



3. Schichtengraph:



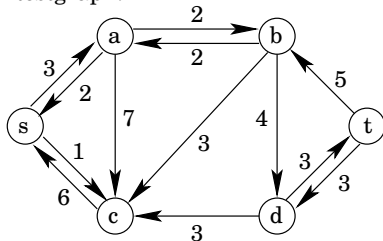
Finden einer nicht-vergrößerbaren Flussfunktion:

- Wir modifizieren die Tiefensuche so, dass die Suche endet, wenn der Knoten t erreicht wurde, oder wenn ein Knoten ohne auslaufende Kanten erreicht wurde.
→ Laufzeit: $\mathcal{O}(\mathcal{V})$
- Starte eine modifizierte Tiefensuche bei s :
 - Wenn der Knoten t durch die Tiefensuche erreicht wird, wurde ein zunehmender Pfad P von s nach t gefunden:
 - Verkleinere die Kapazitäten der Kanten auf P um $\Delta(P)$.
 - Mindestens eine Kante erhält die Kapazität 0: Entferne diese Kante aus dem Schichtengraph.
 - Wenn Knoten t nicht erreicht wird, entferne die letzte Kante des Weges, damit keine Endlosschleifen auftreten.→ Laufzeit: $\mathcal{O}(\mathcal{V})$
- Starte die Tiefensuche erneut. → Da in jedem Durchlauf eine Kante entfernt wird, gibt es höchstens $\mathcal{O}(\mathcal{E})$ Iterationen.

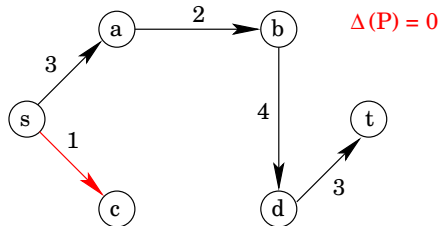
Laufzeit: $\mathcal{O}(\mathcal{E} \cdot \mathcal{V})$

weiter im *Beispiel*:

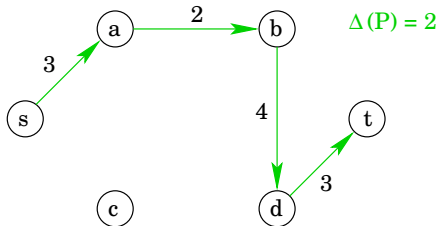
Restgraph:



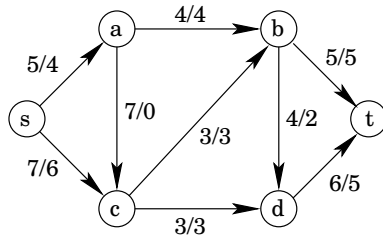
1. Schichtengraph:



2. Schichtengraph:



maximaler Fluss:



Laufzeit:

- Nach jeder Iteration der Schritte 1 und 2 des Algorithmus erhöht sich die Anzahl der Kanten in den kürzesten Wegen um mindestens 1.

Da ein kürzester Weg keinen Knoten mehrfach besucht, gibt es höchstens $\mathcal{O}(\mathcal{V})$ Iterationen.

- Wir hatten bereits festgestellt, dass jede Ausführung einer solchen Iteration $\mathcal{O}(\mathcal{E} \cdot \mathcal{V})$ Zeit kostet.

- Insgesamt ergibt sich also als Laufzeit: $\mathcal{O}(\mathcal{E} \cdot \mathcal{V}^2)$

Laufzeit von Edmonds/Karp zum Vergleich: $\mathcal{O}(\mathcal{E}^2 \cdot \mathcal{V})$

Aber es geht noch besser!

Sei $G = (V, E, c, s, t)$ ein Netzwerk. Eine Funktion $f : E \rightarrow \mathbb{R}^+$ heißt *Präfluss*, falls gilt:

- $0 \leq f(u, v) \leq c(u, v)$ für alle $(u, v) \in E$
- Für alle Knoten $v \in V - \{s, t\}$ kann ein Überschuss $\text{excess}_f(v)$ vorhanden sein:

$$\text{excess}_f(v) := \sum_{e \in \text{In}(v)} f(e) - \sum_{e \in \text{Out}(v)} f(e) \geq 0$$

Jeder Fluss f ist auch ein Präfluss, da $\text{excess}_f(v) = 0$ für alle Knoten $v \in V - \{s, t\}$ gilt. Im Folgenden sei $G_R(f) = (V, E_f, c_f)$ der Restgraph zu Graph G und Präfluss f .

Ein Knoten $v \in V - \{s, t\}$ heißt *aktiv* (active), wenn gilt: $\text{excess}_f(v) > 0$

Der Algorithmus von Goldberg und Tarjan wendet zwei Prozeduren **PUSH** und **RELABEL** an, um den Überschuss bei aktiven Knoten abzubauen.

Mittels $\text{PUSH}(u, v)$ wird ein Teil des Überschusses von Knoten u zu Knoten v verschoben.

Für die Ausführung müssen drei Bedingungen erfüllt sein:

- $\text{excess}_f(u) > 0$: Knoten u ist aktiv, d.h. an Knoten u liegt ein Überschuss vor.
- $(u, v) \in E_f$, also $c_f(u, v) > 0$: Im Restgraphen gibt es eine Kante von u nach v , über die der Überschuss weitergeleitet werden kann.
- $\text{height}(u) > \text{height}(v)$: Der Fluss kann nur von einem höheren zu einem niedrigeren Knoten fließen.

Der Wert, der verschoben wird, ist $\min\{\text{excess}_f(u), c_f(u, v)\}$.

Mittels **RELABEL**(u) kann der Knoten u angehoben werden, und zwar soweit, dass die Höhe um eins größer ist, als die Höhe des niedrigsten Nachbarknotens.

Für die Ausführung müssen zwei Bedingungen erfüllt sein:

- $excess_f(u) > 0$: Es muss einen Grund für die Erhöhung geben, d.h. es muss ein Überschuss abgetragen werden. Daher werden nur aktive Knoten angehoben.
- $height(u) \leq height(v)$ für alle Kanten $(u, v) \in E_f$: Alle zu u benachbarten Knoten liegen höher oder gleich hoch.

Durch das Anheben des Knotens u wird $height(u)$ auf den folgenden Wert gesetzt:

$$height(u) := \min\{height(v) + 1 \mid (u, v) \in E_f\}$$

Eine Kante $(u, v) \in E_f$ heißt **zulässig** (admissible), wenn $height(u) = height(v) + 1$ gilt.

GOLDBERG-TARJAN-ALGORITHM()

```
for all edges  $(u, v) \in E$  do  $f(u, v) := 0$   
for all neighbors  $v$  of  $s$  do  $f(s, v) := c(s, v)$   
for all nodes  $v \in V - \{s\}$  do  $height(v) := 0$   
 $height(s) := \mathcal{V}$   
while  $\exists$  active node  $v \in V - \{s, t\}$  do  
    if  $\exists$  admissible edge  $(v, w) \in E_f$   
        then PUSH( $v, w$ )  
        else RELABEL( $v$ )
```

PUSH(v, w)

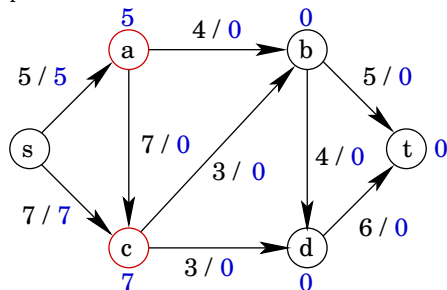
```
if  $(v, w) \in E$  then  
     $f(v, w) := f(v, w) + \min\{excess_f(v), c_f(v, w)\}$   
else  $f(v, w) := f(v, w) - \min\{excess_f(v), c_f(v, w)\}$ 
```

RELABEL(v)

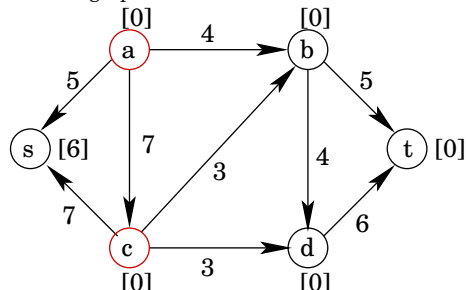
```
 $height(v) := \min\{height(w) + 1 \mid (v, w) \in E_f\}$ 
```

Push-Relabel-Algorithmen

preflow



residual graph

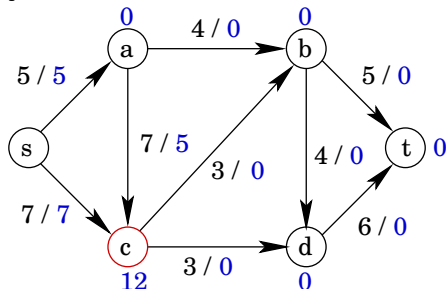


→ RELABEL(a) und PUSH(a, c)

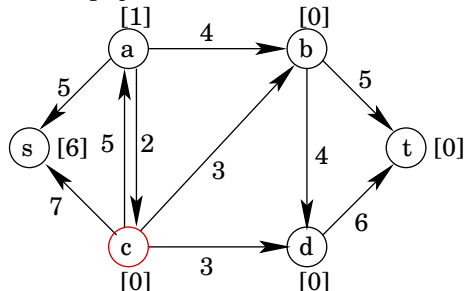
Damit der Überschuss in Richtung t abgebaut werden kann, werden Knoten immer nur soweit angehoben, dass der Überschuss in Richtung der kleinsten Höhe abfließen kann. Wir werden sehen, dass dadurch die Höhen der Knoten untere Schranken für die Länge der kürzesten Wege im Restgraphen zur Senke t sind.

Push-Relabel-Algorithmen

preflow



residual graph



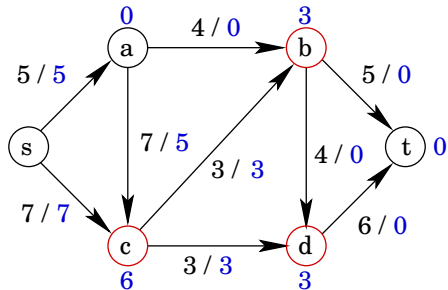
→ RELABEL(c) und PUSH(c, b) und PUSH(c, d)

Invariante: Nach jeder PUSH- oder RELABEL-Operation gilt:

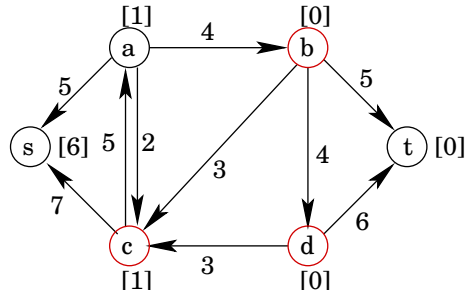
- f ist ein Präfluss.
- $height(u) \leq height(v) + 1$ für jede Kante $(u, v) \in E_f$ im Restgraphen.

Push-Relabel-Algorithmen

preflow



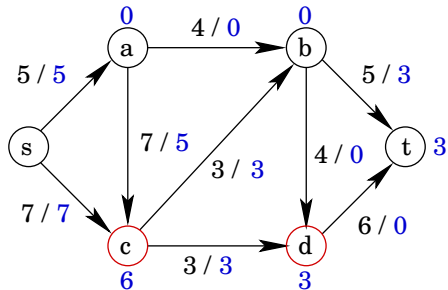
residual graph



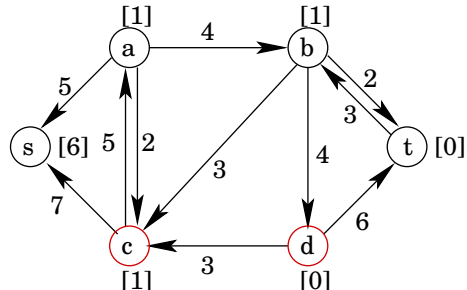
→ RELABEL(b) und PUSH(b, t)

Push-Relabel-Algorithmen

preflow



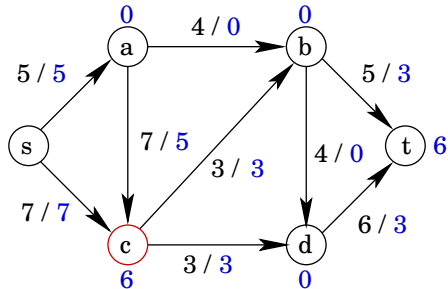
residual graph



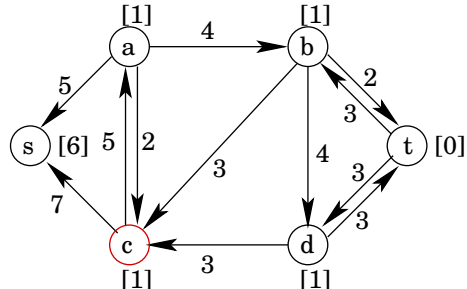
→ RELABEL(d) und PUSH(d, t)

Push-Relabel-Algorithmen

preflow



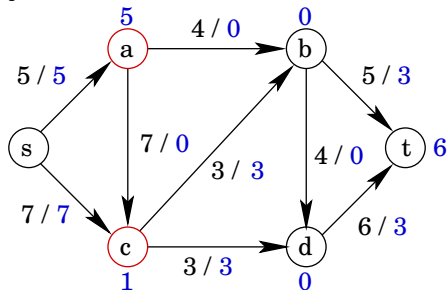
residual graph



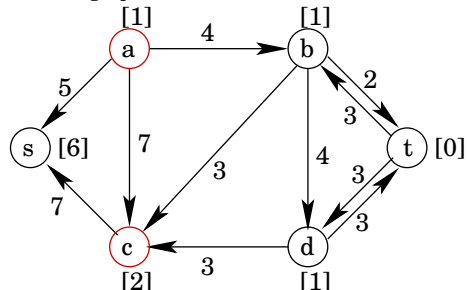
→ RELABEL(c) und PUSH(c, a)

Push-Relabel-Algorithmen

preflow



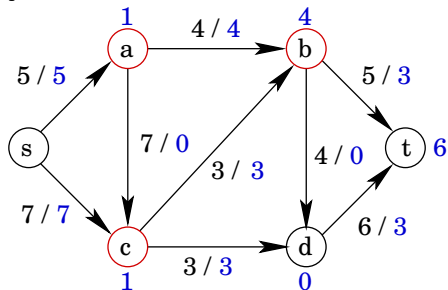
residual graph



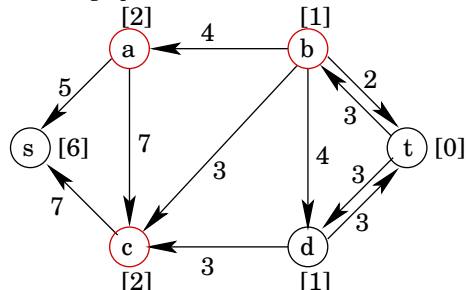
→ RELABEL(a) und PUSH(a, b)

Push-Relabel-Algorithmen

preflow



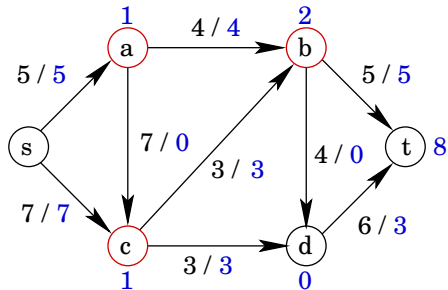
residual graph



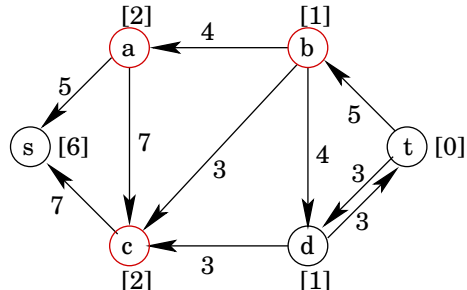
→ PUSH(b, t)

Push-Relabel-Algorithmen

graph



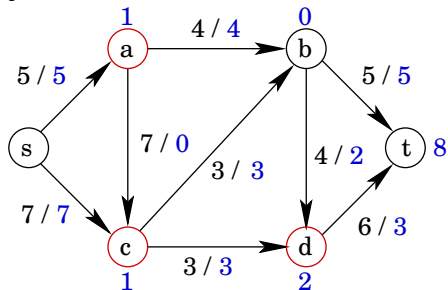
residual graph



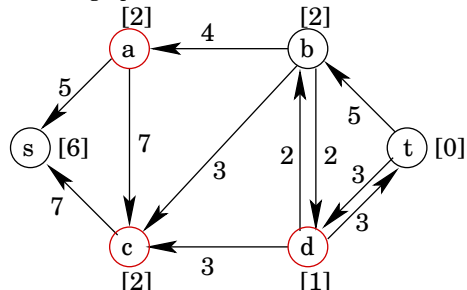
→ RELABEL(b) und PUSH(b, d)

Push-Relabel-Algorithmen

preflow



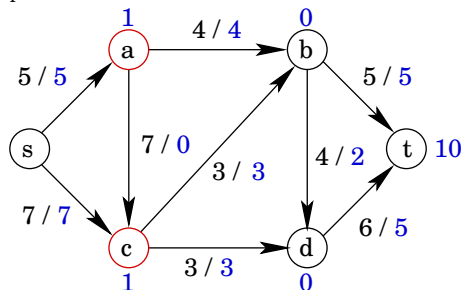
residual graph



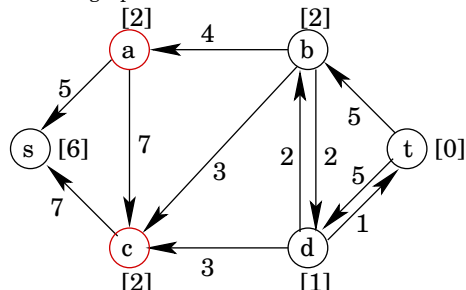
$\rightarrow \text{PUSH}(d, t)$

Push-Relabel-Algorithmen

preflow



residual graph



Der Wert der Flussfunktion ist bereits maximal. Der Algorithmus muss aber noch die Überschüsse in den Knoten a und c abbauen. Dazu werden a und c immer weiter angehoben, bis der Überschuss in Richtung s abgebaut werden kann.

Invariante:

Nach jeder PUSH- oder RELABEL-Operation gilt: f ist ein Präfluss.

RELABEL:

- Der Präfluss wird nicht verändert.

PUSH(u, v):

- Da niemals mehr als der Überschuss $excess_f(u)$ in Knoten u abgebaut wird, bleibt $excess_f(u) \geq 0$ auch nach der Operation erhalten.
- Da der Präfluss an Knoten v erhöht wird, und bereits vor dem Erhöhen $excess_f(v) \geq 0$ galt, gilt dies auch nach dem PUSH.

Invariante:

Nach jeder PUSH- oder RELABEL-Operation gilt für jede Kante $(u, v) \in E_f$ im Restgraphen: $height(u) \leq height(v) + 1$

RELABEL(u):

- Vor dem Erhöhen galt $height(u) \leq height(v) + 1$ für alle Kanten $(u, v) \in E_f$.
- Durch das Erhöhen wird $height(u)$ auf den Wert $\min\{height(v) + 1 \mid (u, v) \in E_f\}$ gesetzt.

PUSH(u, v):

- Es kann eine neue Kante (v, u) im Restgraphen entstehen.
 - Vor der Ausführung von PUSH galt $height(u) = height(v) + 1$, da nur zulässige Kanten ausgewählt werden.
- Daher gilt nachher: $height(v) = height(u) - 1 \leq height(u) + 1$

Senke t ist im Restgraphen von Quelle s aus nicht erreichbar. Beweis durch Widerspruch:

- Annahme: Es existiert ein einfacher gerichteter Weg von s nach t im Restgraphen $G_R(f)$.
- Sei $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_l = t$ ein solcher Weg.
- Dann gilt $height(v_i) \leq height(v_{i+1}) + 1$ für $0 \leq i < l$.
- Also gilt $height(s) \leq height(t) + l$ und $l < \mathcal{V}$.
- ⚡ Initial wird $height(s) = \mathcal{V}$ und $height(t) = 0$ gesetzt, und t wird niemals angehoben.

Der Algorithmus terminiert, wenn es keine aktiven Knoten gibt:

- Dann gilt $excess_f(u) = 0$ für alle Knoten $u \in V - \{s, t\}$, und daher ist f ein Fluss.
 - Außerdem gibt es einen Schnitt S , so dass $s \in S$ und $t \in \bar{S}$ ist.
- Also ist f maximal.

Wir müssen noch zeigen, dass der Algorithmus terminiert.

Sei f ein Präfluss. Wenn $\text{excess}_f(u) > 0$ für einen Knoten u gilt, dann ist Quelle s im Restgraphen von u aus erreichbar, d.h. der Überschuss kann in Richtung Quelle abgebaut werden.

- Sei U die Menge aller von u aus erreichbaren Knoten im Restgraphen. Zu zeigen: $s \in U$
- Nach Definition von $\text{excess}_f(v)$ gilt:

$$\sum_{v \in U} \text{excess}_f(v) = \sum_{v \in U} \left(\sum_{e \in \text{In}(v)} f(e) - \sum_{e \in \text{Out}(v)} f(e) \right)$$

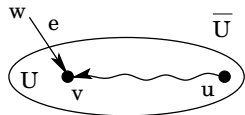
Dabei gilt:

- $e \in U \times U$ kommt positiv und negativ vor: Beitrag 0
- $e \in \overline{U} \times \overline{U}$ kommt nicht vor
- $e \in \overline{U} \times U$ kommt nur positiv vor: Beitrag $f(e)$
- $e \in U \times \overline{U}$ kommt nur negativ vor: Beitrag $-f(e)$

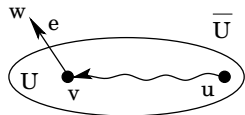
$$\rightarrow \sum_{v \in U} \text{excess}_f(v) = \sum_{e \in E \cap (\overline{U} \times U)} f(e) - \sum_{e \in E \cap (U \times \overline{U})} f(e)$$

Auf der letzten Folie hatten wir festgestellt:

$$\sum_{v \in U} excess_f(v) = \sum_{e \in E \cap (\bar{U} \times U)} f(e) - \sum_{e \in E \cap (U \times \bar{U})} f(e)$$



Es muss $f(w, v) = 0$ gelten, da sonst die Kante $(v, w) \in E_f$ erzeugt würde, und damit $w \in U$ gelten würde.



Es muss $f(v, w) = c(e)$ gelten, damit die Kante (v, w) nicht im Restgraphen liegt, denn sonst würde $w \in U$ gelten.

$$\sum_{v \in U} excess_f(v) = \sum_{e \in E \cap (\bar{U} \times U)} 0 - \sum_{e \in E \cap (U \times \bar{U})} c(e) = - \sum_{e \in E \cap (U \times \bar{U})} c(e)$$

Auf der letzten Folie hatten wir festgestellt:

$$\sum_{v \in U} \text{excess}_f(v) = - \sum_{e \in E \cap (U \times \bar{U})} c(e)$$

Annahme: $s \notin U$

Da $\text{excess}_f(v) \geq 0$ für alle Knoten $v \in V - \{s, t\}$ gilt, muss gelten:

$$\sum_{v \in U} \text{excess}_f(v) = - \sum_{e \in E \cap (U \times \bar{U})} c(e) \geq 0$$

Da $c(e) \geq 0$ für alle Kanten $e \in E$ gilt, kann nur gelten:

$$\sum_{v \in U} \text{excess}_f(v) = - \sum_{e \in E \cap (U \times \bar{U})} c(e) = 0 \quad \textcolor{red}{\text{⚡}}$$

Widerspruch, da $\text{excess}_f(u) > 0$ ist und $u \in U$ gilt.

Ein Überschuss kann also zur Quelle s abgebaut werden. Nun zeigen wir, dass die Knoten nicht beliebig angehoben werden.

Es gilt: $height(v) \leq 2\mathcal{V} - 1$ für alle $v \in V$.

- Sei v ein aktiver Knoten.
- Dann gibt es einen Weg $v = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_l = s$ der Länge $l \leq \mathcal{V} - 1$.
- Außerdem gilt $height(v_i) \leq height(v_{i+1}) + 1$ für $0 \leq i < l$.
- Da initial $height(s) = \mathcal{V}$ gesetzt wird und Knoten s nicht angehoben wird, gilt:
 $height(v) \leq height(s) + \mathcal{V} - 1 = 2\mathcal{V} - 1$

Es werden höchstens $\mathcal{O}(\mathcal{V}^2)$ RELABEL-Operationen durchgeführt, denn jeder der \mathcal{V} Knoten kann höchstens $2\mathcal{V} - 1$ mal angehoben werden.

Operation $\text{PUSH}(u, v)$ heißt *sättigend*, falls $c_f(u, v) \leq \text{excess}_f(u)$ gilt, ansonsten heißt die Operation *nicht sättigend*.

Es werden $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ sättigende PUSH -Operationen durchgeführt:

- Wir betrachten eine beliebige Kante $e = (u, v)$.
- Wird Kante e saturiert, dann gilt $\text{height}(u) > \text{height}(v)$.
- Kante e wird aus dem Restgraphen entfernt und ist erst dann wieder enthalten, wenn ein $\text{PUSH}(v, u)$ ausgeführt wurde.
- Dazu muss $\text{height}(v) > \text{height}(u)$ gelten, also muss $\text{height}(v)$ um mindestens zwei erhöht worden sein.
- Damit also ein weiteres Mal ein $\text{PUSH}(u, v)$ Kante e sättigen kann, muss auch $\text{height}(u)$ um mindestens zwei steigen.
- $\text{height}(u)$ kann höchstens auf $2\mathcal{V} - 1$ steigen, also kann Kante e höchstens $\frac{2\mathcal{V}-1}{2} < \mathcal{V}$ mal saturiert werden.
- Da es im Restgraphen höchstens $2 \cdot \mathcal{E}$ viele Kanten gibt, kann es nur $2\mathcal{E} \cdot \mathcal{V}$ sättigende PUSH -Operationen geben.

Um zu zeigen, dass $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$ nicht sättigende PUSH-Operationen durchgeführt werden, benötigen wir folgende Potenzialfunktion:

$$\Phi(f) := \sum_{v: \text{excess}_f(v) > 0} \text{height}(v)$$

nicht sättigende PUSH(v, w)-Operation:

- Der Wert der Potenzialfunktion wird um $\text{height}(v)$ verringert, da der Überschuss an Knoten v komplett abgebaut wird.
- Allerdings wird der Wert ggf. um $\text{height}(w)$ erhöht, da der Überschuss nun bei Knoten w ist, und Knoten w vorher evtl. keinen Überschuss hatte.
- Da aber $\text{height}(v) > \text{height}(w)$ sein muss, wird der Wert insgesamt um mindestens eins erniedrigt.

sättigende $\text{PUSH}(v, w)$ -Operation:

- Wir wissen bereits, dass $\text{height}(w) \leq 2V - 1$ ist, also wird der Wert der Potenzialfunktion um höchstens $2V - 1$ erhöht.
- Da es nur $2V \cdot \mathcal{E}$ sättigende PUSH -Operationen gibt, kann der Potenzialanstieg nach oben mit $4V^2 \cdot \mathcal{E}$ abgeschätzt werden.

$\text{RELABEL}(v)$:

- Da der Knoten v angehoben wird, steigt auch der Wert der Potenzialfunktion.
- Insgesamt ist der Potenzialanstieg durch $(2V - 1) \cdot V \in \mathcal{O}(V^2)$ beschränkt, da $\text{height}(v) \leq 2V - 1$ ist und es V viele Knoten gibt.

Wir können also festhalten:

- Der Potenzialanstieg durch sättigende PUSH-Operationen kann nach oben mit $4\mathcal{V}^2 \cdot \mathcal{E}$ abgeschätzt werden.
- Der Potenzialanstieg durch RELABEL-Operationen ist durch $2\mathcal{V}^2$ nach oben beschränkt.
- Bei jeder nicht sättigenden PUSH-Operation wird der Wert der Potenzialfunktion um mindestens eins erniedrigt.

Für die Anzahl der nicht sättigenden PUSH-Operationen np gilt:

$$np \leq 2\mathcal{V}^2 + 4\mathcal{V}^2 \cdot \mathcal{E} \in \mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$$

Die Laufzeit des Goldberg-Tarjan-Algorithmus ist also $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$:

- $\mathcal{O}(\mathcal{V}^2)$ RELABEL-Operationen
- $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ sättigende PUSH-Operationen
- $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$ nicht sättigende PUSH-Operationen

Wir haben noch nicht festgelegt, wie der aktive Knoten gewählt wird. Diese Wahl hat entscheidenden Einfluss auf die Laufzeit:

- Wähle von allen aktiven Knoten denjenigen Knoten aus, dessen *height*-Wert am größten ist. \rightarrow Laufzeit: $\mathcal{O}(\mathcal{V}^2 \cdot \sqrt{\mathcal{E}})$

Wir zeigen nur: Laufzeit $\mathcal{O}(\mathcal{V}^3)$

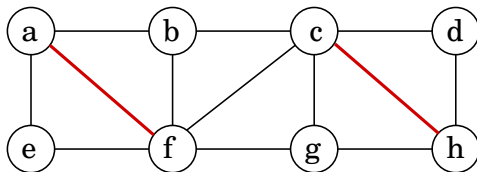
- Ein Knoten v wird solange gewählt, bis er durch eine nicht sättigende PUSH-Operation nicht mehr aktiv ist.
- Knoten v kann erst wieder aktiv werden, wenn für mindestens einen Knoten w der Wert $height(w)$ erhöht wurde.
- Nach \mathcal{V} vielen nicht sättigenden PUSH-Operationen ohne zwischenzeitlicher RELABEL-Operation gibt es keine aktiven Knoten mehr und der Algorithmus terminiert.
- Da es nur $\mathcal{O}(\mathcal{V}^2)$ viele RELABEL-Operationen gibt, kann es nur $\mathcal{O}(\mathcal{V}^3)$ viele nicht sättigende PUSH-Operationen geben.
- Verwalte die aktiven Knoten in einer FIFO-Liste.
 \rightarrow Laufzeit: $\mathcal{O}(\mathcal{V}^3)$

Graphalgorithmen:

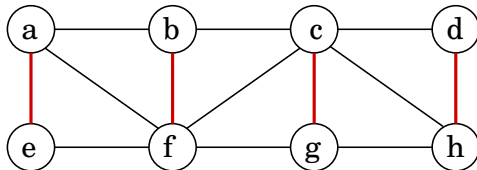
- Grundlagen
- Tiefen- und Breitensuche
- Zusammenhangsprobleme
- Minimale Spannbäume
- Kürzeste Wege
- Netzwerkfluss
- *Matching-Probleme*

Sei $G = (V, E)$ ein ungerichteter Graph. Ein *Matching* ist eine Teilmenge $M \subseteq E$ der Kanten, so dass keine zwei Kanten aus M einen gemeinsamen Endknoten haben:

- *Maximales Matching* → nicht vergrößerbares Matching



- *Maximum-Matching* → ein größtes Matching



²Paarung, Abgleich, Anpassung

Matching-Probleme sind in vielen Bereichen zu lösen:

- Stundenplanerstellung: Finde eine Zuordnung zwischen Raum, Fach und Lehrer.
→ dreidimensionales Matching, NP-vollständig
- Sind die Zuordnungen auf disjunkten Knotenmengen zu berechnen, also z.B. wenn
 - Studenten auf Studienplätze verteilt werden sollen, oder
 - den Mitarbeitern Tätigkeiten zugewiesen werden sollen,so spricht man von bipartiten Matchings.

Ein ungerichteter Graph $G = (V, E)$ heißt *bipartit*, wenn sich die Knotenmenge in 2 Mengen $V_1, V_2 \subseteq V$ zerlegen lässt, so dass gilt:

- $V_1 \cup V_2 = V$ und $V_1 \cap V_2 = \emptyset$
- Für alle $\{u, v\} \in E$ gilt: $u \in V_1, v \in V_2$ oder $u \in V_2, v \in V_1$

zunächst: Berechne ein größtes Matching für bipartite Graphen mittels Flussalgorithmus.

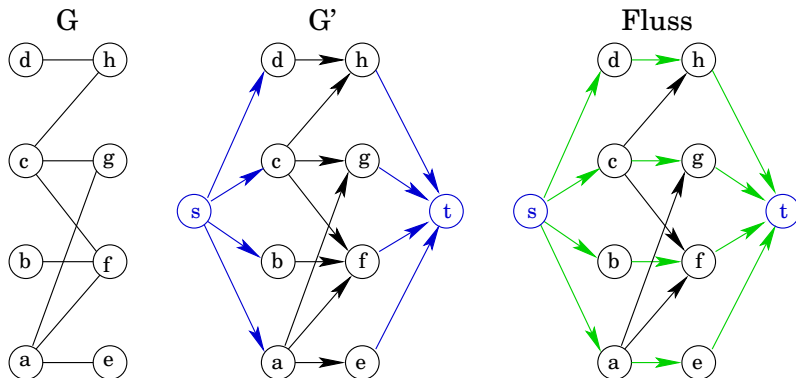
Konstruiere zu dem gegebenen bipartiten Graphen $G = (V, E)$ mit $V = V_1 \cup V_2$ das Netzwerk $G' = (V', E', c)$ mit

- $V' = V \cup \{s, t\}$
- $E' = \{s\} \times V_1 \cup \{(u, v) \in V_1 \times V_2 \mid \{u, v\} \in E\} \cup V_2 \times \{t\}$
- $c : E' \rightarrow \mathbb{N}$ mit $c(e) = 1$ für alle Kanten $e \in E'$.

Algorithmus

- Bestimme für G' einen maximalen Fluss durch zunehmende Pfade P mit $\Delta(P) = 1$.
 - Jede Kante in G' hat den Flusswert 0 oder 1.
- Die Kanten $\{u, v\} \in E$ mit $f((u, v)) = 1$ bilden ein größtes Matching in G .

Matching in bipartiten Graphen



größtes Matching: Alle Kanten $\{u, v\} \in E$ mit $f((u, v)) = 1$

- Für alle $v \in V_1$ gilt: Falls $f(s, v) = 1$ ist, dann hat genau eine auslaufende Kante $e = (v, w)$ den Fluss $f(e) = 1$.
- Für alle $w \in V_2$ gilt: Falls $f(w, t) = 1$ ist, dann hat genau eine einlaufende Kante $e = (v, w)$ den Fluss $f(e) = 1$.
- Also gilt: $f(s, v) = 1 \Rightarrow f(v, w) = 1 \Rightarrow f(w, t) = 1$ liefert Matching-Kante (v, w) .

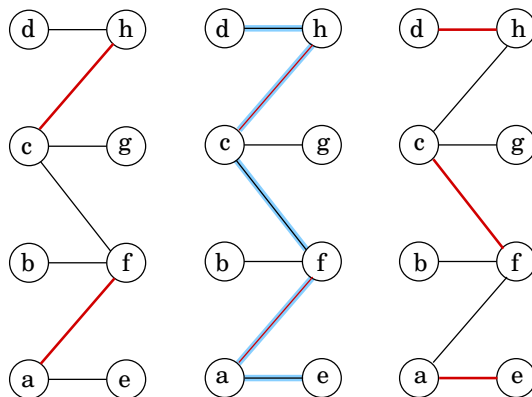
Es geht auch ohne Flussalgorithmen!

Sei $G = (V, E)$ ein Graph, und sei $M \subseteq E$ ein Matching.

- Die Kanten aus M heißen *gebundene Kanten*, die übrigen sind die *freien Kanten*.
- Die Endknoten der gebundenen Kanten heißen *gebundene Knoten*, alle übrigen Knoten sind *freie Knoten*.
- Ein Weg $P = (v_1, \dots, v_k)$, der abwechselnd aus freien und gebundenen Kanten besteht, heißt *alternierender Weg*.
- Ein alternierender Weg $P = (v_1, \dots, v_k)$ heißt *zunehmend alternierender Weg*, wenn $v_1 \neq v_k$ gilt und beide Endknoten v_1 und v_k mit keiner gebundenen Kante inzident sind.

Wir nutzen zunehmend alternierende Wege, um ein bestehendes Matching zu vergrößern. Beachte: $|M| \leq \mathcal{V}/2$

Matching in bipartiten Graphen



Das bestehende Matching M kann vergrößert werden, indem

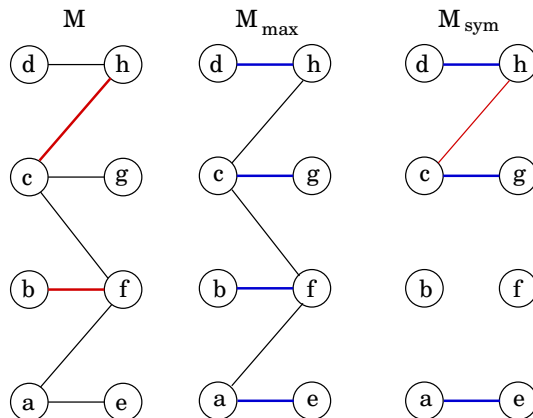
- ein zunehmend alternierender Weg P gesucht wird,
- und dann alle gebundenen Kanten in P aus M entfernt werden und alle freien Kanten in P zu M hinzugefügt werden.

Matching in bipartiten Graphen

Sei M ein beliebiges Matching, und sei M_{max} ein größtes Matching. Die *symmetrische Differenz*

$$M_{sym} = (M_{max} - M) \cup (M - M_{max})$$

enthält genau die Kanten, die nur in einem der beiden Matchings vorkommen.



Durch fortlaufendes Finden zunehmend alternierender Wege kann ein größtes Matching berechnet werden:

- Für $M_{sym} = (M_{max} - M) \cup (M - M_{max})$ gilt: Jeder Knoten im Graphen G ist mit höchstens zwei Kanten aus M_{sym} inzident: eine Kante aus M_{max} , und eine Kante aus M
 - Die Anzahl der aus M_{max} stammenden Kanten ist größer als die Anzahl der aus M stammenden Kanten: $|M_{max}| > |M|$ und in M_{sym} sind alle Kanten aus $M_{max} \cup M$, die nicht in beiden Mengen sind.
 - $G_{sym} = (V, M_{sym})$ besteht aus knotendisjunkten Wegen bzw. Kreisen, die abwechselnd Kanten aus M_{max} und M benutzen.
 - Jeder Kreis, falls einer existiert, hat genau so viele Kanten aus M_{max} wie aus M .
- Da $|M_{max}| > |M|$ ist, muss es Wege geben, die eine Kante mehr aus M_{max} als aus M haben, und dies sind zunehmend alternierende Wege.

Wir stellen fest:

- Wenn ein zunehmend alternierender Weg existiert, kann das bisherige Matching vergrößert werden.
- Falls kein solcher Weg gefunden wird, terminiert der Algorithmus und hat ein größtes Matching berechnet.

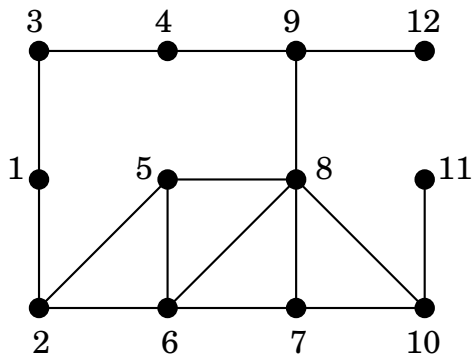
In bipartiten Graphen können zunehmend alternierende Wege mittels Breitensuche in Zeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$ berechnet werden. Daher ergibt sich für obigen Algorithmus die Laufzeit:

- maximal $\mathcal{V}/2$ Aufrufe der Breitensuche
- insgesamt: $\mathcal{O}(\mathcal{V}^2 + \mathcal{V} \cdot \mathcal{E})$ oder $\mathcal{O}(\mathcal{V}^3)$ wegen $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$

Der Algorithmus von Hopcroft und Karp findet in jedem Durchlauf alle kürzesten, zunehmend alternierenden Wege und benötigt daher nur $\sqrt{\mathcal{V}}$ Durchläufe.

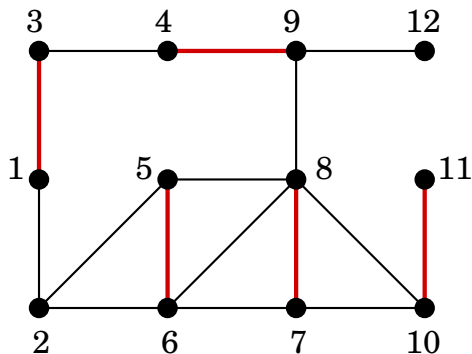
→ Laufzeit $\mathcal{O}(\sqrt{\mathcal{V}} \cdot \mathcal{E})$

Matching auf allgemeinen Graphen



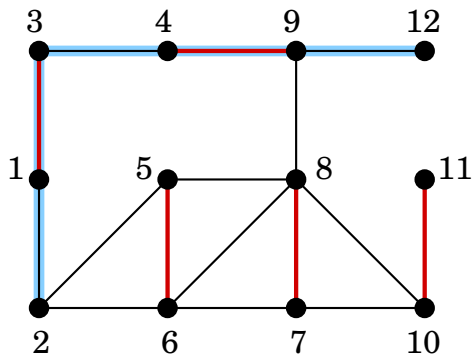
→ bestimme ein initiales Matching

Matching auf allgemeinen Graphen



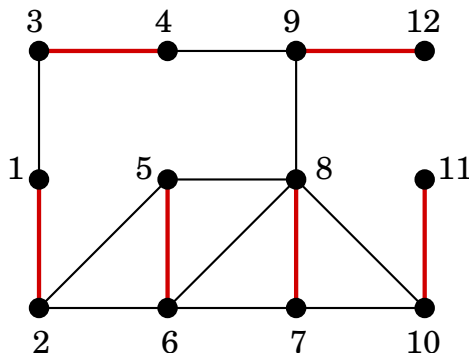
→ bestimme einen zunehmend alternierenden Weg

Matching auf allgemeinen Graphen



→ vergrößere das Matching entlang dieses Weges

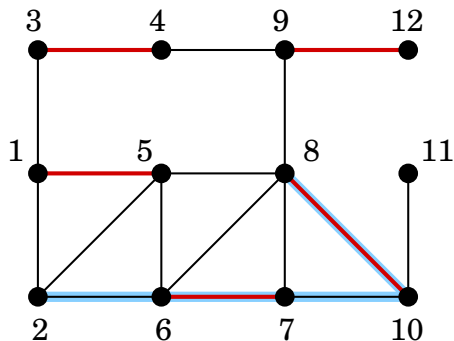
Matching auf allgemeinen Graphen



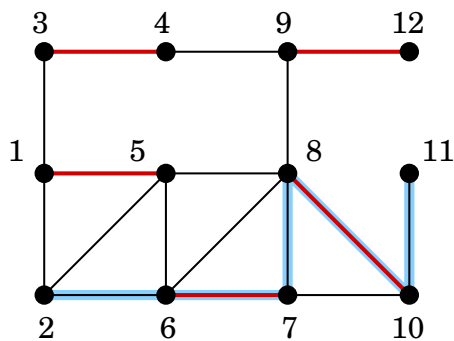
Leider hilft uns die Breitensuche allein nun nicht weiter. Es kann vorkommen, dass ein Knoten auf einem Weg gerader Länge und auf einem anderen Weg ungerader Länge erreichbar ist.

Matching auf allgemeinen Graphen

Starte Breitensuche bei einem freien Knoten: 2



Der Weg $P_1 = (2, 6, 7, 10, 8)$ hat die Länge 4, also gerade Länge.



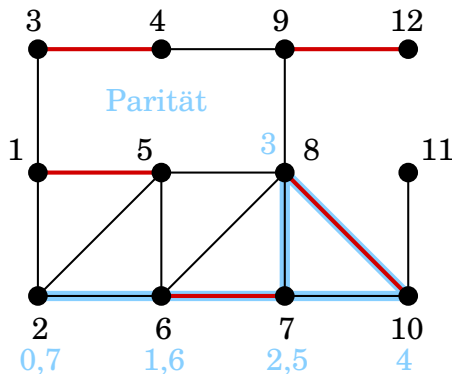
Der Weg $P_2 = (2, 6, 7, 8)$ hat die Länge 3, also ungerade Länge.

Der Weg P_1 kann nicht zu einem zunehmend alternierenden Weg erweitert werden! Würde die Breitensuche bei Knoten 7 anders verzweigen, würde ein solcher Weg gefunden.

Matching auf allgemeinen Graphen

Idee: Die Breitensuche darf jeden Knoten zweimal besuchen, einmal für gerade Entfernung und einmal für ungerade Entfernung.

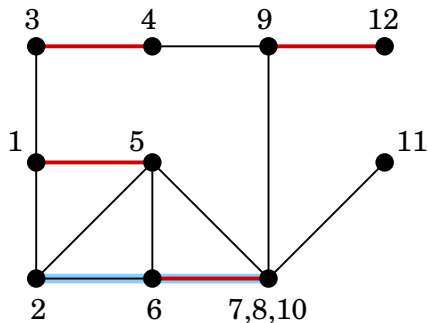
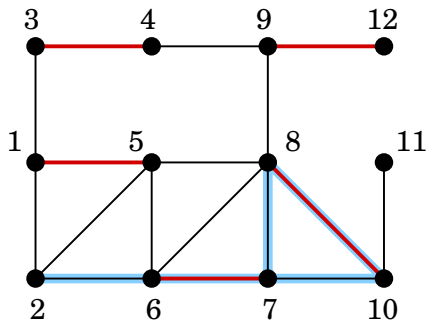
Das ist leider keine gute Idee: Es kann dann passieren, dass wir einen Kreis ungerader Länge komplett durchlaufen, und anschließend einen Teil des Weges rückwärts gehen, da sich die Paritäten der Knoten umgedreht haben. Ein solcher Weg ist natürlich nicht zunehmend alternierend.



Der Weg $P_3 = (2, 6, 7, 8, 10, 7, 6, 2)$ läuft einmal komplett durch den Kreis $(7, 8, 10, 7)$ und läuft dann zum Startknoten der Breitensuche zurück. Dies ist möglich, da alle Knoten einmal für gerade und einmal für ungerade Entfernung besucht werden dürfen.

Matching auf allgemeinen Graphen

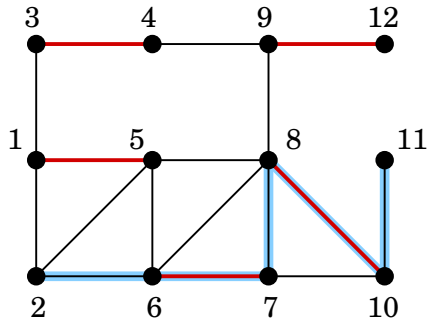
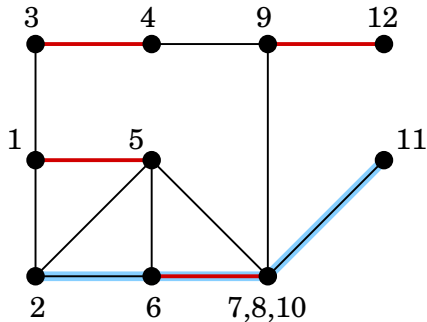
Ein Kreis ungerader Länge heißt *Blüte*. Wird bei der Suche nach einem zunehmend alternierenden Weg eine Blüte gefunden, so schrumpfen wir die Blüte auf einen Knoten zusammen: Jede Kante, die zuvor mit einem Knoten der Blüte verbunden war, führt jetzt zu diesem neuen Knoten.



Der Weg kann zu einem zunehmend alternierenden Weg erweitert werden.

Matching auf allgemeinen Graphen

Nachdem ein zunehmend alternierender Weg gefunden wurde, müssen alle geschrumpften Blüten wieder expandiert werden.

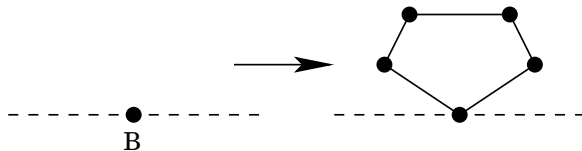


Dazu muss nur bestimmt werden, wie der zunehmend alternierende Weg innerhalb der expandierten Blüte fortgesetzt wird. Das dies immer möglich ist, zeigt die folgende Überlegung.

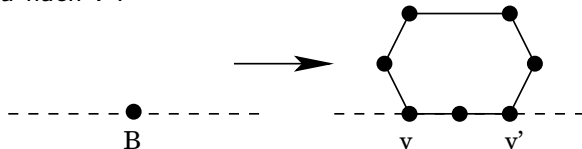
Matching auf allgemeinen Graphen

Sei P ein zunehmend alternierender Weg in G ausgehend von u , der eine Blüte B passiert. Es gibt zwei Fälle:

- P berührt die Blüte nur in einem Knoten. Dann bleibt der Pfad unverändert.

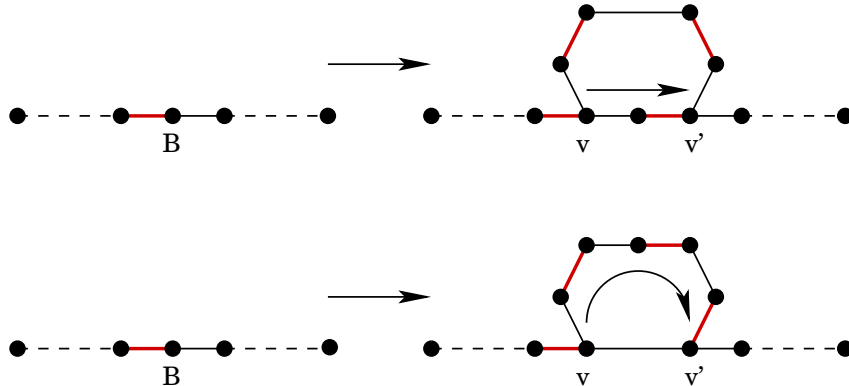


- P betritt die Blüte in einem Knoten v und verlässt sie wieder in einem anderen Knoten v' . In der Blüte gibt es zwei Pfade von v nach v' : einen mit gerader Länge und einen mit ungerader Länge. Einer der beiden Pfade erhält die Parität des Abstands von u nach v' .



Matching auf allgemeinen Graphen

Expandieren von Blüten: Für den Weg durch eine Blüte gibt es zwei Möglichkeiten. Je nachdem, welche Parität notwendig ist, wird der korrekte Pfad gewählt.



Laufzeit von Edmond's Algorithmus: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$

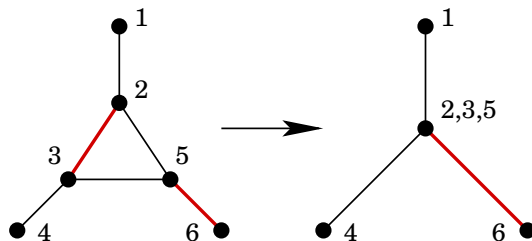
- Alle Operationen in einer Runde dauern Zeit $\mathcal{O}(\mathcal{E})$:
 - Alternierenden Weg suchen mittels Breitensuche von einem freien Knoten aus: $\mathcal{O}(\mathcal{V} + \mathcal{E})$
 - Auf einem Pfad können alle Blüten zusammen höchstens \mathcal{E} Kanten besitzen. Das Schrumpfen einer Blüte ist abhängig von der Größe des Kreises. Demnach können alle Blüten in einem Durchlauf in $\mathcal{O}(\mathcal{E})$ Schritten geschrumpft werden.
- In jeder Runde wird das Matching größer, also gibt es höchstens $\mathcal{V}/2$ viele Runden.

Auch diese Laufzeit kann noch verbessert werden:

S. Micali, V. Vazirani (1980): An $\mathcal{O}(\sqrt{\mathcal{V}} \cdot \mathcal{E})$ algorithm for finding maximum matching in general graphs. 21st Annual Symposium on Foundations of Computer Science: 17–27

Matching auf allgemeinen Graphen

Anmerkung: Es genügt nicht, zunächst alle Blüten zu suchen und zu schrumpfen, um anschließend eine Breitensuche durchzuführen:



Im obigen Beispiel enthält der geschrumpfte Graph keinen zunehmend alternierenden Pfad, wohl aber der Ausgangsgraph.

- Einleitung
- Entwurfsmethoden
- Sortieren
- Auswahlproblem
- Graphalgorithmen
- *Spezielle Graphklassen*
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

Schwere Graphenprobleme auf speziellen Graphklassen

- *Einleitung*
- Bäume und Co-Graphen
- Chordale Graphen
- Vergleichbarkeitsgraphen (Comparability Graph)
- Planare Graphen

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

- Die Anzahl der Knoten in einer Maximum-Clique (also einer Clique mit maximaler Kardinalität) wird mit $\omega(G)$ bezeichnet und heißt *Cliquenzahl* (clique number).
- Seien C_1, \dots, C_k Cliques in G , die G partitionieren, also $V = C_1 \cup \dots \cup C_k$ und $C_i \cap C_j = \emptyset$ für alle $1 \leq i, j \leq k$ und $i \neq j$. Dann heißt C_1, \dots, C_k eine *Cliquen-Partitionierung* der Größe k .
- Die Größe der kleinstmöglichen Cliques-Partitionierung wird mit $\theta(G)$ bezeichnet und heißt *Cliquen-Überdeckungszahl* (clique cover number).

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

- Sei I eine Teilmenge der Knoten aus V , die paarweise nicht adjazent sind. Man nennt I eine *unabhängige Menge* (independent set) oder eine *stabile Menge* (stable set).
- Die Anzahl der Knoten einer unabhängigen Menge mit maximaler Kardinalität wird mit $\alpha(G)$ bezeichnet und heißt *Unabhängigkeitszahl* oder *Stabilitätszahl* (stability number).
- Seien I_1, \dots, I_k unabhängige Mengen von G . Stellen diese unabhängigen Mengen I_1, \dots, I_k eine Partitionierung des Graphen G dar, also $V = I_1 \cup \dots \cup I_k$ und $I_i \cap I_j = \emptyset$ für alle $1 \leq i, j \leq k$ und $i \neq j$, so spricht man von einer *k-Färbung* (k -coloring) von G .
- Das minimale k , für welches eine k -Färbung von G existiert, wird mit $\chi(G)$ bezeichnet und heißt *Färbungszahl* oder *chromatische Zahl* (chromatic number).

Allgemein gilt:

- $\omega(G) \leq \chi(G)$
Alle Knoten einer Clique müssen mit verschiedenen Farben gefärbt sein.
- $\alpha(G) \leq \theta(G)$
Alle Knoten einer unabhängigen Menge müssen in verschiedenen Cliquen einer Cliquen-Partitionierung liegen.
- $\omega(G) = \alpha(\overline{G})$
Eine Clique maximaler Kardinalität in G entspricht einer unabhängigen Menge maximaler Kardinalität in \overline{G} .
- $\chi(G) = \theta(\overline{G})$
Eine k -Färbung in G entspricht einer Cliquen-Partitionierung in \overline{G} .

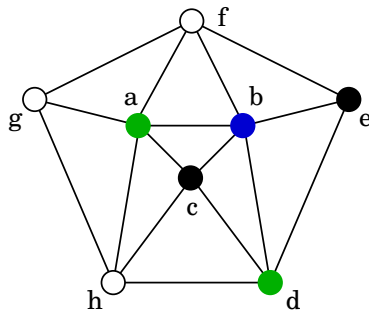
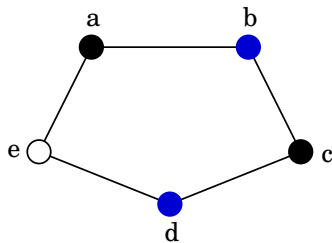
Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißen *isomorph* zueinander, wenn eine bijektive Abbildung $f : V_1 \rightarrow V_2$ existiert, wobei $\{u, v\} \in E_1 \iff \{f(u), f(v)\} \in E_2$ gelten muss. Wir schreiben dann $G_1 \simeq G_2$.

Übung

- Geben Sie Graphen an, bei denen $\omega(G) < \chi(G)$ gilt.
- Geben Sie Graphen an, bei denen $\alpha(G) < \theta(G)$ gilt.
- Erstellen Sie den Komplementgraphen zu P_4 , also einem Weg über vier Knoten.
- Erstellen Sie den Komplementgraphen zu C_5 , also einem Kreis über fünf Knoten.

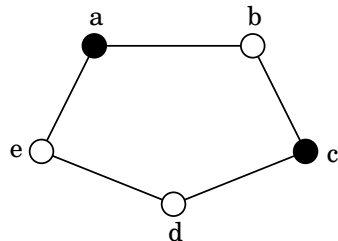
Geben Sie Graphen an, bei denen $\omega(G) < \chi(G)$ gilt.

Links: Ein Kreis ungerader Länge hat nur Cliques der Größe 2, ist aber nur dreifärbbar.



Rechts: Das innere Dreieck a, b, c stellt eine 3-Clique dar und muss daher mit drei Farben gefärbt werden. Die Farben der Knoten d und e ergeben sich automatisch, für f benötigen wir eine vierte Farbe.

Geben Sie Graphen an, bei denen $\alpha(G) < \theta(G)$ gilt.



Die Knoten a und c stellen ein maximum independent set der Größe zwei dar, eine Cliquesüberdeckung besteht aus mindestens drei Cliques, zum Beispiel $\{a, b\}$, $\{c, d\}$, $\{e\}$.

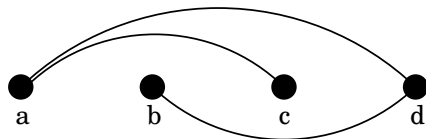
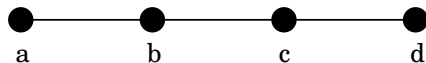
Hier fällt auf, dass beim Kreis C_5 sowohl $\omega(G) < \chi(G)$ als auch $\alpha(G) < \theta(G)$ gilt. Ist das Zufall?

Wir wissen bereits, dass ganz allgemein $\omega(G) = \alpha(\overline{G})$ und $\chi(G) = \theta(\overline{G})$ gilt.

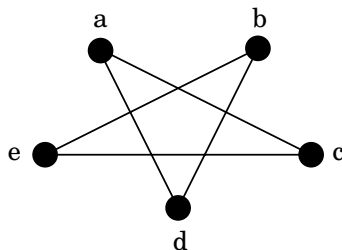
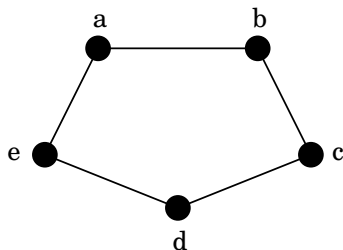
Also gilt hier: $\omega(C_5) = \alpha(\overline{C_5}) < \chi(C_5) = \theta(\overline{C_5})$

Betrachten wir den C_5 auf der nächsten Folie genauer.

Der P_4 und der Komplementgraph $\overline{P_4}$. Die Graphen sind isomorph zueinander.



Der C_5 und der Komplementgraph $\overline{C_5}$. Die Graphen sind isomorph zueinander.



Eine Grapheigenschaft A ist eine Untermenge der Menge aller Graphen, so dass für je zwei isomorphe Graphen G_1 und G_2 gilt: $G_1 \in A \iff G_2 \in A$.

Ein Graph G hat die Eigenschaft A , falls $G \in A$ ist.

- Eine Grapheigenschaft heißt *monoton*, falls für jeden Graphen mit dieser Eigenschaft gilt, dass auch jeder seiner Subgraphen diese Eigenschaft hat.
- Eine Grapheigenschaft heißt *hereditär*, falls für jeden Graphen mit dieser Eigenschaft gilt, dass auch jeder seiner induzierten Subgraphen diese Eigenschaft hat.

Jede monotone Eigenschaft ist auch hereditär. Aber nicht jede hereditäre Eigenschaft ist auch monoton: Die Eigenschaft eines Graphen, vollständig zu sein, ist hereditär, aber nicht monoton.

Schwere Graphenprobleme auf speziellen Graphklassen

- Einleitung
- *Bäume und Co-Graphen*
- Chordale Graphen
- Vergleichbarkeitsgraphen (Comparability Graph)
- Planare Graphen

Ein zusammenhängender Graph $G = (V, E)$ mit $|E| = |V| - 1$ Kanten ist ein *Baum* und insbesondere kreisfrei. Ein kreisfreier Graph ist ein *Wald*.

Für Bäume und Wälder gilt: $\omega(G) = \chi(G) = 2$, falls $|E| > 0$

Frage: Ist die Eigenschaft Baum/Wald monoton/hereditär?

Ein Graph $G = (V, E)$ heißt *bipartit*, falls $V = A \cup B$ gilt mit $A \cap B = \emptyset$ und $\{u, v\} \in E \Rightarrow (u \in A, v \in B) \vee (v \in A, u \in B)$.

Ein Graph ist genau dann bipartit, wenn er keinen Kreis ungerader Länge enthält.

Für bipartite Graphen gilt: $\omega(G) = \chi(G) = 2$, falls $|E| > 0$

Frage: Ist die Eigenschaft bipartit monoton/hereditär?

Ist die Eigenschaft Baum monoton/hereditär?

- Die Eigenschaft Baum ist nicht monoton: Wenn man Kanten aus einem Baum entfernt, zerfällt der Graph und wird zu einem Wald.
- Die Eigenschaft Baum ist nicht hereditär: Wenn man einen Knoten v mit $\deg(v) \geq 2$ entfernt, zerfällt der Baum in mehrere Teile.

Ist die Eigenschaft Wald monoton/hereditär?

- Die Eigenschaft Wald ist monoton und damit auch hereditär.

Ist die Eigenschaft bipartit monoton/hereditär?

- Die Eigenschaft bipartit ist monoton und damit auch hereditär: Durch das Entfernen einer Kante oder eines Knotens kann kein Kreis ungerader Länge entstehen. Dazu müssten Kanten $\{u, v\}$ mit $u, v \in A$ oder $u, v \in B$ existieren.

Es seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei Graphen.

- Die *disjunkte Vereinigung* von G_1 und G_2 ist definiert durch

$$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2).$$

- Die *disjunkte Summe* von G_1 und G_2 ist definiert durch

$$G_1 \times G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{\{v_1, v_2\} \mid v_1 \in V_1, v_2 \in V_2\}).$$

Ein Graph $G = (V, E)$ ist ein *Co-Graph*, falls er sich aus den folgenden drei Regeln aufbauen lässt:

- Der Graph mit genau einem Knoten (Bezeichnung $G = \bullet$) ist ein Co-Graph.
- Die disjunkte Vereinigung zweier Co-Graphen ist ein Co-Graph.
- Die disjunkte Summe zweier Co-Graphen ist ein Co-Graph.

Co-Graphen waren ursprünglich anders definiert:

- Der Graph mit genau einem Knoten (Bezeichnung $G = \bullet$) ist ein Co-Graph.
- Die disjunkte Vereinigung zweier Co-Graphen ist ein Co-Graph.
- Ist G ein Co-Graph, dann ist auch \overline{G} ein Co-Graph.

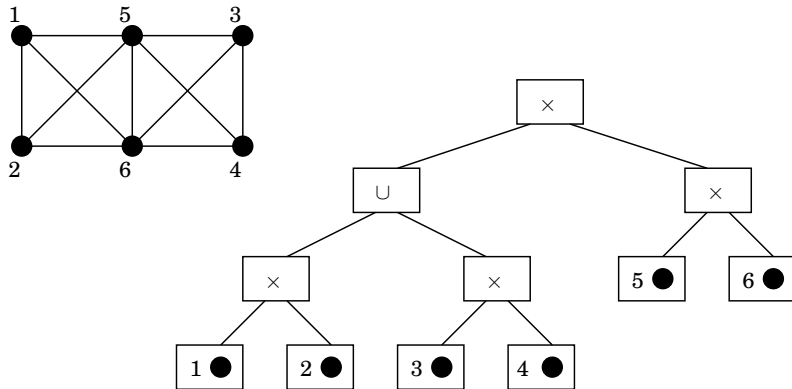
Beide Definitionen sind äquivalent. Die ursprüngliche Definition erklärt aber besser den Namen der Graphklasse: Complement-Reducible-Graph

Lemma: Für zwei Co-Graphen G_1 und G_2 gilt:

$$G_1 \times G_2 = \overline{\overline{G_1} \cup \overline{G_2}}$$

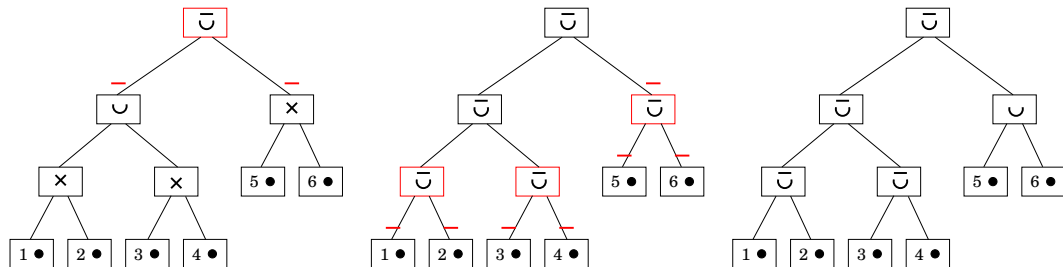
Beobachtung: Ein Graph ist genau dann ein Co-Graph, wenn er keinen P_4 als induzierten Teilgraphen enthält.

Der Aufbau eines Co-Graphen kann in einer Baumstruktur dargestellt werden: Links der Co-Graph, rechts der Co-Baum.



Übung: Stellen Sie den Co-Graphen als Baum über die Operationen der alternativen Definition dar, also mittels disjunkter Vereinigung und Komplementbildung.

Lösung: Beginne in der Wurzel mit der Ersetzung von \times -Knoten zu $\bar{\cup}$ -Knoten.

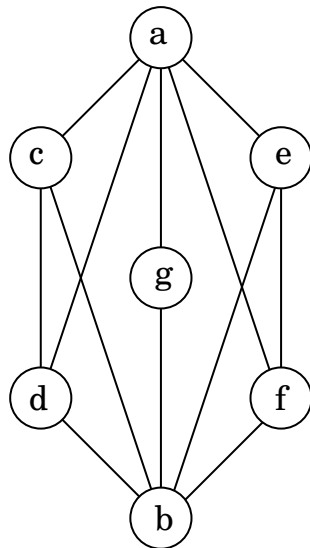


Satz: Zu einem gegebenen Graphen $G = (V, E)$ kann in Zeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$ entschieden werden, ob G ein Co-Graph ist und im positiven Fall auch ein Co-Baum für G bestimmt werden. (A linear recognition algorithm for cographs: Corneil, Perl, Stewart, 1985)

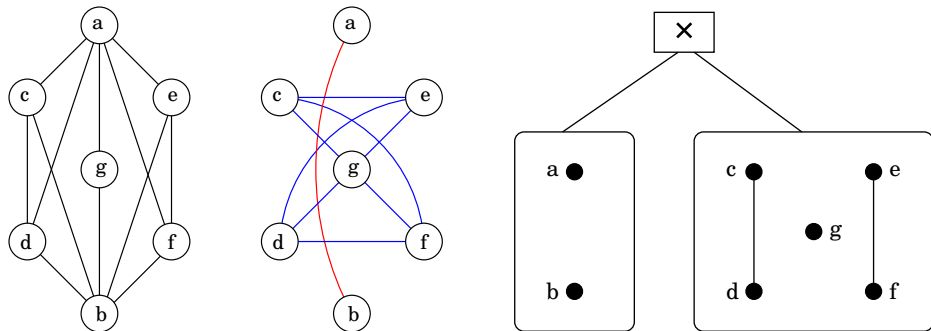
Erstellen eines Co-Baums: Sei G ein Co-Graph. Iteriere die folgenden Schritte, bis die Graphen nur noch aus einem Knoten bestehen.

- Falls G nicht zusammenhängend ist, dann seien G_1, \dots, G_k seine Zusammenhangskomponenten. Erstelle rekursiv den Co-Baum für jede Komponente G_i und erstelle einen \cup -Knoten als Vorgängerknoten.
- Sonst betrachte den Komplementgraphen \overline{G} und seine Zusammenhangskomponenten $\overline{G}_1, \dots, \overline{G}_k$. Erstelle rekursiv den Co-Baum für jeden Komplementgraphen G_i und erstelle einen \times -Knoten als Vorgänger.

Übung: Erstellen Sie für den folgenden Co-Graphen den Co-Baum.



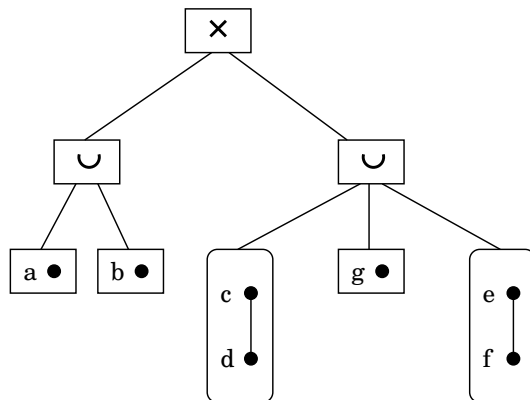
Lösung: Da der Graph G zusammenhängend ist, erstelle den Komplementgraphen \overline{G} , ermittle dessen Zusammenhangskomponenten $\overline{G} = \overline{G_1} \cup \overline{G_2}$ und füge einen \times -Knoten als Vorgänger ein.



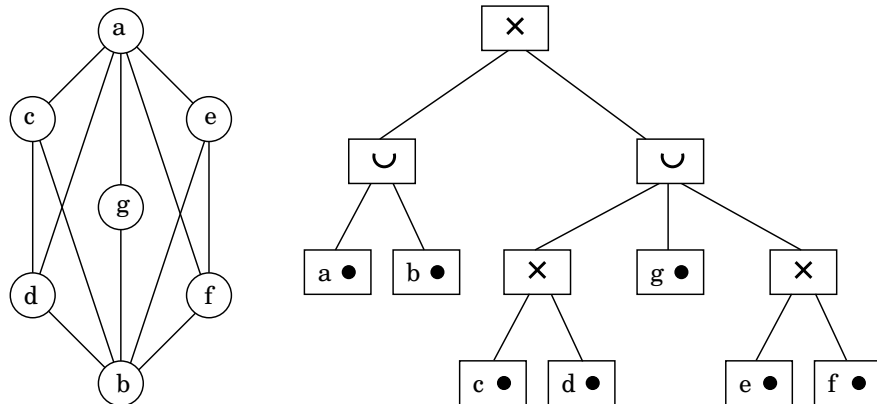
Es gilt: $\overline{G} = \overline{G_1} \cup \overline{G_2} \equiv \overline{\overline{G}} = \overline{\overline{G_1} \cup \overline{G_2}}$, also gilt $\overline{\overline{G}} = G = G_1 \times G_2$

Setze das Verfahren mit den Graphen G_1 und G_2 fort.

Fortsetzung: Da die Graphen G_1 und G_2 jeweils nicht zusammenhängend sind, betrachten wir deren Zusammenhangskomponenten und fügen jeweils einen \cup -Knoten als Vorgänger ein.



Schließlich erhalten wir den folgenden Co-Baum.



Bemerkung: Da bei einem zusammenhängenden Co-Graphen G die letzte Operation immer die \times -Operation ist, hat G einen Durchmesser von höchstens 2, d.h. der längste kürzeste Weg zwischen 2 Knoten hat höchstens die Länge 2.

Im Folgenden sei G ein Co-Graph und $T = (V_T, E_T, w)$ ein Co-Baum zu G mit der Wurzel w . Für einen Knoten $v \in V_T$ sei T_v der Teilbaum von T mit Wurzel v und $G[v]$ sei der durch T_v definierte Co-Graph.

Independent Set:

- Für jedes Blatt v in T setze $\alpha(G[v]) = 1$.
- Für jeden mit \cup gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\alpha(G[v]) = \alpha(G[v_1]) + \alpha(G[v_2])$
- Für jeden mit \times gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\alpha(G[v]) = \max\{\alpha(G[v_1]), \alpha(G[v_2])\}$

Clique:

- Für jedes Blatt v in T setze $\omega(G[v]) = 1$.
- Für jeden mit \cup gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\omega(G[v]) = \max\{\omega(G[v_1]), \omega(G[v_2])\}$
- Für jeden mit \times gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\omega(G[v]) = \omega(G[v_1]) + \omega(G[v_2])$

Färbung:

- Für jedes Blatt v in T setze $\chi(G[v]) = 1$.
- Für jeden mit \cup gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\chi(G[v]) = \max\{\chi(G[v_1]), \chi(G[v_2])\}$
- Für jeden mit \times gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\chi(G[v]) = \chi(G[v_1]) + \chi(G[v_2])$

Partition in Cliques:

- Für jedes Blatt v in T setze $\theta(G[v]) = 1$.
- Für jeden mit \cup gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\theta(G[v]) = \theta(G[v_1]) + \theta(G[v_2])$
- Für jeden mit \times gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\theta(G[v]) = \max\{\theta(G[v_1]), \theta(G[v_2])\}$

Die hier gezeigten Algorithmen haben Laufzeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$:

- Erstellung des Co-Baums in $\mathcal{O}(\mathcal{V} + \mathcal{E})$.
- Bottom-Up-Durchlauf der Knoten des Co-Baums in $\mathcal{O}(\mathcal{V} + \mathcal{E})$.

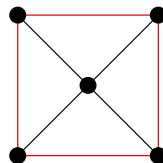
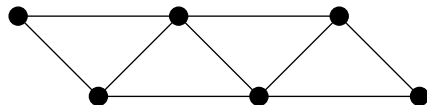
Schwere Graphenprobleme auf speziellen Graphklassen

- Einleitung
- Bäume und Co-Graphen
- *Chordale Graphen*
- Vergleichbarkeitsgraphen (Comparability Graph)
- Planare Graphen

Ein Graph heißt *chordal*, falls er keinen induzierten Teilgraphen enthält, der isomorph zum Kreis C_k mit $k \geq 4$ ist. In anderen Worten: Jeder Kreis der Länge mindestens vier besitzt eine Sehne. Eine *Sehne* ist eine Kante zwischen zwei auf dem Kreis nicht benachbarten Knoten.

Beispiele:

- Der vollständige Graph K_n ist chordal.
- Ein Wald ist chordal.
- Der linke Graph ist chordal, der rechte nicht:



Frage: Ist die Eigenschaft *chordal* monoton/hereditär?

Die Eigenschaft *chordal* ist nicht monoton:

- Wenn man eine Sehne entfernt, zerstört man die Eigenschaft.

Die Eigenschaft *chordal* ist hereditär:

- Annahme: Durch das Entfernen von Knoten entsteht ein Kreis der Länge größer gleich vier, der keine Sehne hat.
- Durch das Hinzufügen des Knotens und der inzidenten Kanten entsteht keine Sehne in dem Kreis.
- Also ist schon der ursprüngliche Graph nicht chordal. ⚡

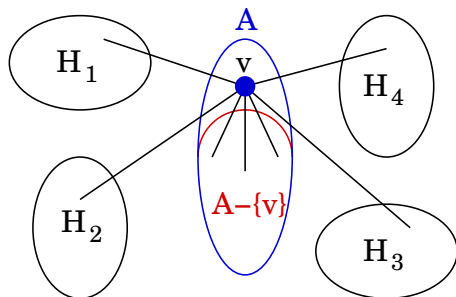
Sei $G = (V, E)$ ein Graph und $u, v \in V$ zwei Knoten.

- Eine Knotenmenge $A \subset V - \{u, v\}$ heißt *$u - v$ -trennend*, falls u und v in G , aber nicht in $G - A$ durch einen Pfad verbunden sind.
- Eine Knotenmenge $A \subset V$ heißt *trennend*, falls es Knoten $u, v \in V$ gibt, so dass A $u - v$ -trennend ist.

Lemma: Sei A eine inklusionsweise-minimal trennende Knotenmenge in einem zusammenhängenden Graphen $G = (V, E)$. Dann ist jeder Knoten in A zu jeder Komponente von $G - A$ verbunden.

Beweis: Da A minimal ist, gilt für jedes $v \in A$, dass der Graph $G - (A - \{v\})$ noch zusammenhängend ist. Da es zwischen den Komponenten H_1, \dots, H_k von $G - A$ keine Kanten gibt, sind die H_i 's in $G - (A - \{v\})$ alle mit v verbunden, also insbesondere in G . (siehe auch nächste Folie)

noch zum Beweis:



- $G - (A - \{v\})$ ist noch zusammenhängend
- $G - A$ zerfällt in H_1, \dots, H_4

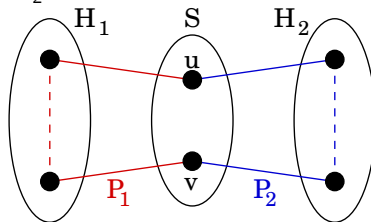
Definition: Wir bezeichnen die Menge der adjazenten Knoten eines Knotens u in einem Graphen $G = (V, E)$ als Nachbarschaft und schreiben $N_G(u) := \{v \mid \{u, v\} \in E\}$.

Falls aus dem Kontext heraus klar ist, welcher Graph G gemeint ist, so schreiben wir $N(u)$ anstelle von $N_G(u)$. Für $A \subseteq V$ definieren wir $N_G(A) := \bigcup_{u \in A} N_G(u)$.

Satz: Ein Graph ist chordal \iff jede inklusionsweise-minimal trennende Knotenmenge induziert eine Clique. (Dirac, 1961)

\Rightarrow Sei S eine solche trennende Menge in G .

- Es gibt mindestens zwei Komponenten H_1 und H_2 in $G - S$.
- Alle Knoten aus S haben Nachbarn in H_1 und H_2 . (Lemma)
- Seien $u, v \in S$, und seien P_1, P_2 zwei $u - v$ -Pfade **minimaler Länge**, die nur innere Knoten aus H_1 bzw. H_2 benutzen.



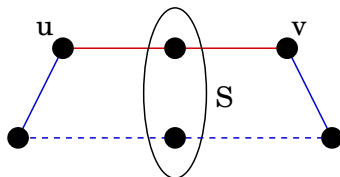
- Der Kreis $P_1 \cup P_2$ hat Länge \geq vier und daher eine Sehne.
- Zwischen H_1 und H_2 gibt es keine Kanten, die Pfade P_1, P_2 haben **minimale Länge**, es gibt also keine Sehne innerhalb von H_1 oder H_2 , also muss die Kante $\{u, v\} \in E$ existieren.

Fortsetzung

Satz: Ein Graph ist chordal \iff jede inklusionsweise-minimal trennende Knotenmenge induziert eine Clique. (Dirac, 1961)

\Leftarrow Sei C ein Kreis der Länge ≥ 4 in G und seien $u, v \in C$ zwei auf C nicht benachbarte Knoten.

- Ist $\{u, v\} \in E$, so gibt es eine Sehne. \checkmark
- Sonst sei S eine minimal $u - v$ -trennende Knotenmenge.
- S enthält je einen Knoten auf beiden $u - v$ -Pfaden in C .



- Da S eine Clique ist, sind diese durch eine Kante verbunden. \checkmark

Ein Knoten in einem Graphen heißt *Simplizialknoten*, falls seine Nachbarschaft vollständig verbunden ist, also eine Clique bildet.

Lemma: Jeder chordale Graph G besitzt einen Simplicialknoten, und falls der Graph nicht vollständig ist, besitzt er sogar **zwei nicht benachbarte** Simplicialknoten. (Dirac, 1961)

Beweis: Durch vollständige Induktion nach $n = |V|$.

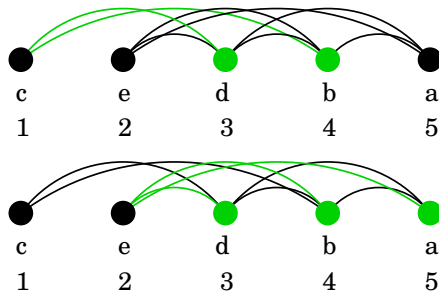
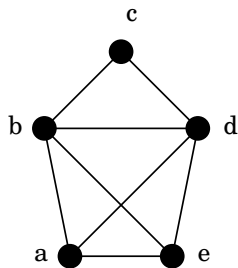
- Aussage ist klar für $n = 1, 2, 3$ oder vollständige Graphen. ✓
- Sonst: G besitzt zwei nicht benachbarte Knoten u und v .
- Sei S eine inklusionsweise-minimal $u - v$ -trennende Knotenmenge, $G - S$ zerfalle in Komponenten H_1, \dots, H_k .
- S induziert Clique, also haben nach Induktionsannahme die chordalen Graphen $G[H_i \cup S]$ je einen Simplicialknoten in H_i :
 - Entweder gibt es zwei nicht benachbarte simpliciale Knoten, von denen einer in H_i liegt, da S eine Clique ist.
 - oder $G[H_i \cup S]$ ist Clique, also jeder Knoten aus H_i simplicial.
- Diese Simplicialknoten in H_i sind auch simplicial in G , weil für die Nachbarschaft $N(H_i)$ gilt: $N(H_i) \subseteq H_i \cup S$

Die Knoten eines chordalen Graphen $G = (V, E)$ mit n Knoten lassen sich wie folgt anordnen.

Eine totale Ordnung $\sigma : V \rightarrow [n]$ heißt *perfektes Knoten-Eliminationsschema* PES, falls jeder Knoten $v \in V$ simplicial in $G[\{u \in V \mid \sigma(u) > \sigma(v)\}]$ ist.

anders gesagt: Die Anordnung (v_1, \dots, v_n) der Knoten aus G heißt PES, wenn die Mengen $\{v_j \in N(v_i) \mid j > i\}$ Cliques sind.

Beispiel:



Satz: Ein Graph ist genau dann chordal, wenn er ein perfektes Knoten-Eliminationsschema besitzt. (Fulkerson, Gross, 1965)

⇒ Sei G ein chordaler Graph und s ein Simplicialknoten.

- Das PES beginnt mit (s, \dots) .
- Auch $G - \{s\}$ ist chordal. (weil hereditär)
- Wähle sukzessiv Simplicialknoten im restlichen Graphen aus und hänge den Knoten an das bisherige PES an.
- Falls keiner gefunden wird, ist der Graph nicht chordal.

⇐ Sei G ein Graph mit PES σ und sei C ein Kreis in G .

- Sei $u \in C$ der erste Knoten des Kreises bezüglich σ .
- Nach Definition des PES sind insbesondere seine beiden Nachbarn auf C verbunden.
- Damit hat der Kreis eine Sehne.

Die Existenz eines PES ist also hinreichend für die Chordalität eines Graphen.

Obige naive Implementierung zur Berechnung eines PES hat Laufzeit $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$.
Mittels einer lexikographischen Breitensuche kann ebenfalls ein PES berechnet werden, aber effizienter.

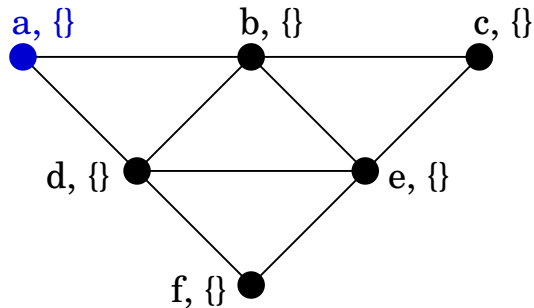
LEX-BFS ist eine normale Breitensuche mit Kollisionsregel:

```
for all  $v \in V$  do  
     $label[v] := \emptyset$   
  
for  $i := n$  downto 1 do  
    pick unnumbered vertex  $v$  with lexicographically largest label  
     $\sigma(v) := i$   
    for all unnumbered vertices  $w$  adjacent to  $v$  do  
        append  $label[w]$  by  $i$ 
```

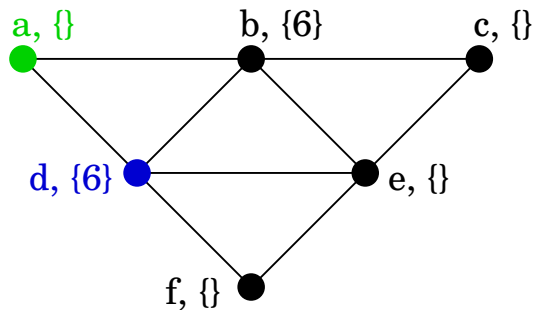
Es gilt $(a_1, \dots, a_q) \leq_L (b_1, \dots, b_p)$, falls es ein j gibt mit $a_i \leq b_i$ für $1 \leq i \leq j$ und $(q < p$ oder $a_{j+1} < b_{j+1})$ gilt.

So gilt bspw. $(6, 3, 2, 1) \leq_L (6, 4, 1)$ und $(6, 4, 2) \leq_L (6, 4, 2, 1)$.

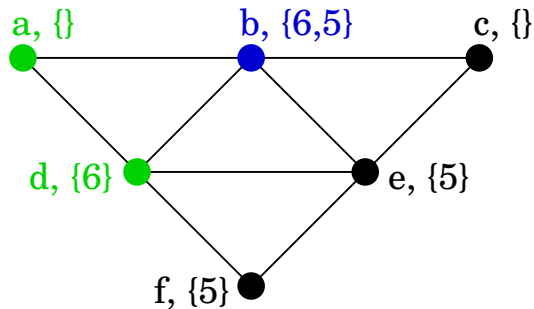
Beispiel: $i = 6$, wähle Knoten $a \rightarrow \sigma = (a)$



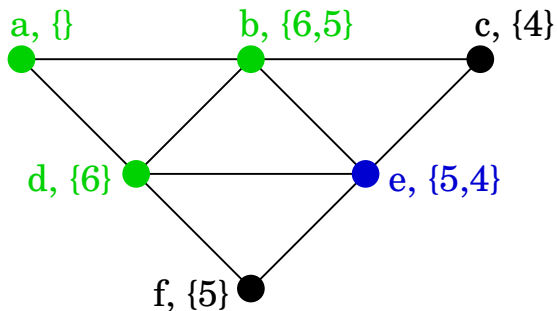
Beispiel: $i = 5$, wähle Knoten $d \rightarrow \sigma = (d, a)$



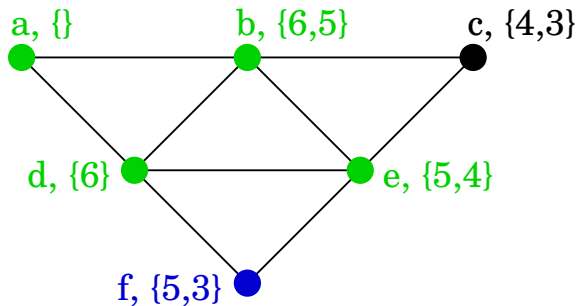
Beispiel: $i = 4$, wähle Knoten $b \rightarrow \sigma = (b, d, a)$



Beispiel: $i = 3$, wähle Knoten $e \rightarrow \sigma = (e, b, d, a)$



Beispiel: $i = 2$, wähle Knoten $f \rightsquigarrow \sigma = (f, e, b, d, a)$



Nach einem weiteren Schritt erhalten wir (c, f, e, b, d, a) als Ordnung. Man prüft leicht nach, dass diese Ordnung ein PES ist.

Satz: Der Algorithmus LEX-BFS liefert bei Anwendung auf einen Graphen G genau dann ein PES, wenn G chordal ist. (ohne Beweis)

maximum clique:

$K := \emptyset$

for all node u in order of σ **do**

 compute $X_u := \{v \in V \mid \{u, v\} \in E, \sigma(v) > \sigma(u)\}$

if $|K| < |\{u\} \cup X_u|$ **then**

$K := \{u\} \cup X_u$

output K

maximum independent set:

- sei y_1 der erste Knoten bezüglich σ
- induktiv: sei y_i der erste auf y_{i-1} folgende Knoten in σ , der nicht in der Menge $X_{y_1} \cup \dots \cup X_{y_{i-1}}$ ist
- die Knoten y_1, \dots, y_t bilden eine unabhängige Menge

minimum clique cover:

Die Mengen $\{y_i\} \cup X_{y_i} - X_{y_1} - \dots - X_{y_{i-1}}$ bilden eine minimale Cliques-Überdeckung.

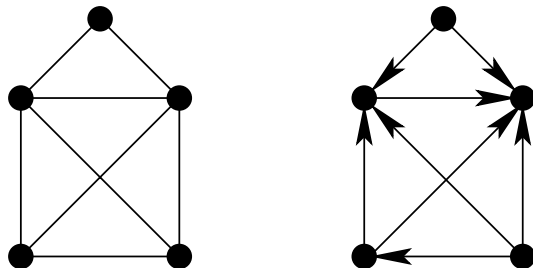
Optimalität gilt wegen $t \leq \alpha(G) \leq \theta(G) \leq t$.

Schwere Graphenprobleme auf speziellen Graphklassen

- Einleitung
- Bäume und Co-Graphen
- Chordale Graphen
- *Vergleichbarkeitsgraphen (Comparability Graph)*
- Planare Graphen

Ein gerichteter Graph $G = (V, E)$ heißt *transitiv*, wenn für je drei Knoten $u, v, w \in V$ gilt: Falls $(u, v) \in E$ ist und $(v, w) \in E$ ist, dann ist auch $(u, w) \in E$.

Ein ungerichteter Graph heißt *Vergleichbarkeitsgraph* (comparability graph), wenn eine Orientierung der Kanten existiert, so dass der orientierte Graph transitiv ist.



Links Vergleichbarkeitsgraph, rechts eine mögliche transitive Orientierung dazu.

Wir führen für diesen Abschnitt eine andere Definition von Graphen ein, die auf M.C. Golumbic zurück geht:

- Ein Graph $G = (V, E)$ besteht aus einer anti-reflexiven binären Relation E über einer endlichen Knotenmenge V . Die Elemente von E bestehen aus geordneten Paaren von Knoten.

Alle Graphen sind also Schleifen-frei und haben keine mehrfachen Kanten.

- Wir definieren $ab \in E^{-1} \iff ba \in E$.
- Ein Graph ist ungerichtet, wenn $E = E^{-1}$ gilt, d.h. ein ungerichteter Graph besteht aus gerichteten Kanten, aber zu jeder Kante gibt es eine entgegengesetzt gerichtete Kante.

Ein ungerichteter Graph $G = (V, E)$ heißt Vergleichbarkeitsgraph, wenn es einen Graphen (V, F) gibt, so dass gilt:

$$F \cap F^{-1} = \emptyset \qquad F \cup F^{-1} = E \qquad F^2 \subseteq F$$

wobei $F^2 = \{ac \mid ab, bc \in F\}$ definiert sei.

Comparability-Graphen

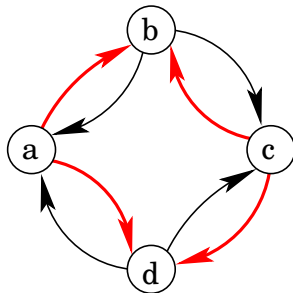
Beispiel: ungerichteter Graph $G = (V, E)$

schwarze Kanten: F

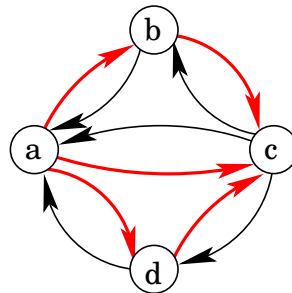
rote Kanten: F^{-1}

$$E = F \cup F^{-1}$$

$$F \cap F^{-1} = \emptyset$$



$$F^2 = \emptyset$$



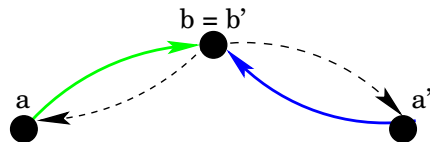
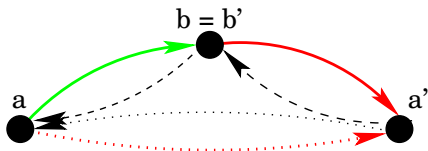
$$F^2 = \{ca\}$$

Definiere Relation Γ auf Knotenmenge E eines ungerichteten Graphen $G = (V, E)$ wie folgt. Sei $ab \in E$ und $a'b' \in E$.

$$ab \Gamma a'b' \iff (a = a' \wedge bb' \notin E) \text{ oder } (b = b' \wedge aa' \notin E).$$

Wir sagen: ab fordert $a'b'$.

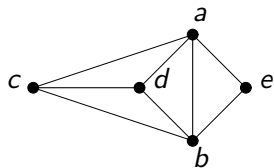
zweiter Fall: Würde ab und $b'a'$ mit $b = b'$ als Richtung der Kanten gewählt, müsste auch aa' gerichtet werden, was aber nicht möglich ist, da $aa' \notin E$ gilt.



Die reflexive, transitive Hülle Γ^* von Γ ist eine Äquivalenzrelation auf E und partitioniert E in *Implikationsklassen* von G . Dabei liegen zwei Kanten ab und cd genau dann in einer Implikations-klasse, wenn gilt: $ab = a_0b_0 \Gamma a_1b_1 \Gamma \dots \Gamma a_kb_k = cd$ mit $k \geq 0$.

Bezeichne $\Phi(G)$ die Menge der Implikationsklassen von G , und bezeichne $\hat{A} = A \cup A^{-1}$ den symmetrischen Abschluss von A .

Beispiel:



$$A_1 = \{ab\}$$

$$A_2 = \{cd\}$$

$$A_3 = \{ac, ad, ae\}$$

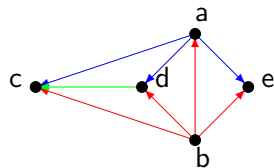
$$A_4 = \{bc, bd, be\}$$

$$A_1^{-1} = \{ba\}$$

$$A_2^{-1} = \{dc\}$$

$$A_3^{-1} = \{ca, da, ea\}$$

$$A_4^{-1} = \{cb, db, eb\}$$



$$\hat{A}_1 = A_1 \cup A_1^{-1}$$

$$\hat{A}_2 = A_2 \cup A_2^{-1}$$

$$\hat{A}_3 = A_3 \cup A_3^{-1}$$

$$\hat{A}_4 = A_4 \cup A_4^{-1}$$

$\Phi(G) = \{\hat{A}_1, \hat{A}_2, \hat{A}_3, \hat{A}_4\}$ ist Menge der Implikationsklassen von G

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

- Eine Partition der Kantenmenge $E = \hat{B}_1 \uplus \dots \uplus \hat{B}_k$ heißt *Decomposition* (Zerlegung) von E , wenn B_i eine Implikationsklasse von $\hat{B}_i \uplus \dots \uplus \hat{B}_k$ darstellt.
- Eine Menge von Kanten $\{(x_1, y_1), \dots, (x_k, y_k)\}$ heißt *Dekompositionsschema* für G , wenn eine Dekomposition $E = \hat{B}_1 \uplus \dots \uplus \hat{B}_k$ existiert mit $(x_i, y_i) \in B_i$.

Berechnung eines Dekompositionsschemas für $G = (V, E)$:

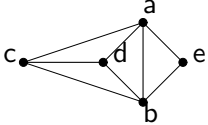
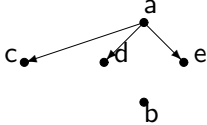
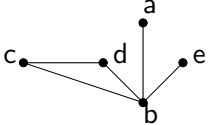
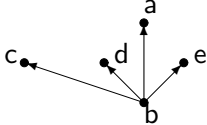
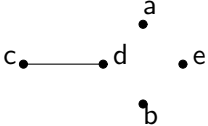
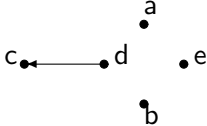
S0 Initial sei $i := 1$ und $E_i := E$

S1 Wähle eine Kante $e_i = (x_i, y_i) \in E_i$

S2 Zähle die Implikationsklasse B_i von E_i auf, die (x_i, y_i) enthält

S3 Definiere $E_{i+1} := E_i \setminus \hat{B}_i$

S4 Falls $E_{i+1} \neq \emptyset$, dann $i := i + 1$ und Sprung nach S1

i	$G_i = (V, E_i)$	(x_i, y_i)	(V, B_i)
1		ac	
2		bc	
3		dc	

Der folgende Satz zeigt, dass der Algorithmus zur Berechnung einer Dekomposition korrekt ist.

Satz: Sei $G = (V, E)$ ein ungerichteter Graph mit Dekomposition $E = \hat{B}_1 \uplus \dots \uplus \hat{B}_k$, dann sind folgende Aussagen äquivalent:

- G ist ein Vergleichbarkeitsgraph
- $A \cap A^{-1} = \emptyset$ für alle Implikationsklassen A von E
- $B_i \cap B_i^{-1} = \emptyset$ für $i = 1, 2, \dots, k$ (ohne Beweis)

Anmerkung: Eine transitive Orientierung ist kreisfrei! Angenommen, es gäbe einen Kreis $C = (v_1, \dots, v_n)$ mit den Kanten $v_1 v_2, v_2 v_3, \dots, v_n v_1$. Dann gilt:

- $v_1 v_3 \in F$, weil $v_1 v_2 \in F$ und $v_2 v_3 \in F$
 - $v_1 v_4 \in F$, weil $v_1 v_3 \in F$ und $v_3 v_4 \in F$
 - $v_1 v_5 \in F$, weil $v_1 v_4 \in F$ und $v_4 v_5 \in F$ usw.
 - bis schließlich gilt: $v_1 v_n \in F$, weil $v_1 v_{n-1} \in F$ und $v_{n-1} v_n \in F$
- ⚡ $v_1 v_n \in F$ und $v_n v_1 \in F$

Sei $G = (V, E)$ ein Vergleichbarkeitsgraph.

maximum clique: Alle Knoten auf einem Weg stellen eine Clique dar.

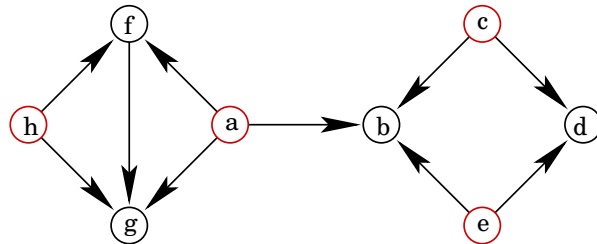
- berechne eine transitive Orientierung $G' = (V, E')$
- berechne eine topologische Sortierung zu G'
- bestimme induktiv einen längsten Weg: $dist(v_1) := 0$, dann

$$dist(v_i) := \max_{\substack{j=1, \dots, i-1 \\ (v_j, v_i) \in E'}} \{dist(v_j) + 1\}$$

maximum independent set: Sei $V = \{v_1, \dots, v_n\}$.

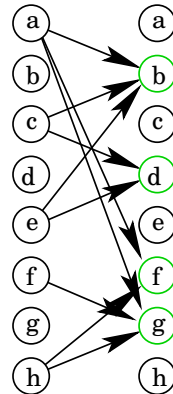
- berechne eine transitive Orientierung $G' = (V, E')$
- berechne einen bipartiten Graphen $G'' = (A \cup B, E'')$ mit
 - $A = \{x_i \mid v_i \in V\}$
 - $B = \{y_i \mid v_i \in V\}$
 - $E'' = \{\{x_i, y_j\} \mid (v_i, v_j) \in E'\}$ (siehe auch nächste Folie)
- berechne ein minimum vertex cover T in dem bipartiten Graphen G''
- dann ist $S := V - T$ ein maximum independent set

Beispiel:



Independent-Set

Vertex-Cover

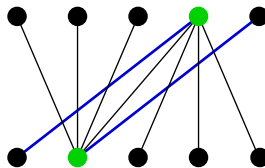


Frage: Wie berechnet man ein Minimum Vertex Cover auf bipartiten Graphen?
Allgemein ist das Problem NP-vollständig.

Ein Minimum Vertex Cover auf bipartiten Graphen kann mittels eines Maximum Matchings berechnet werden. (König, 1931)

Sei M ein Maximum Matching auf dem bipartiten Graphen G .

Klar: Ein Vertex Cover für G hat mindestens $|M|$ viele Knoten, da jede Matching-Kante durch das Vertex Cover abgedeckt sein muss.



Die blauen Kanten stellen ein Maximum Matching dar, die grünen Knoten ein Minimum Vertex Cover.

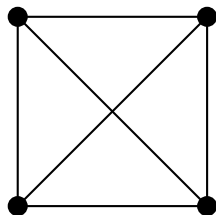
König 1931: In einem bipartiten Graphen ist die Größe eines größten Matchings gleich der Größe einer minimalen Knotenüberdeckung. (Beweis über alternierende Pfade)

Schwere Graphenprobleme auf speziellen Graphklassen

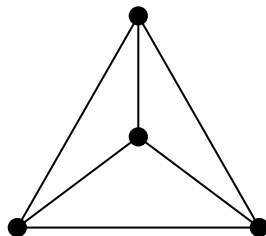
- Einleitung
- Bäume und Co-Graphen
- Chordale Graphen
- Vergleichbarkeitsgraphen (Comparability Graph)
- *Planare Graphen*

Definition: Ein Graph, der kreuzungsfrei in der Ebene gezeichnet werden kann, heißt planar. → planare Einbettung des Graphen

planarer Graph K_4 :

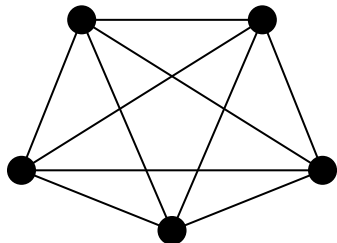


planare Einbettung des K_4 :

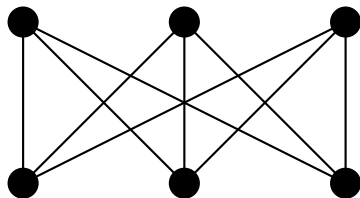


Alle einfachen Graphen mit höchstens 4 Knoten sind planar!

K_5 :



$K_{3,3}$:

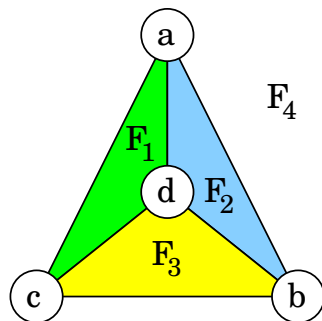


allgemein:

- K_n bezeichnet den vollständigen Graphen mit n Knoten
- $K_{n,m}$ bezeichnet den vollständig bipartiten Graphen $G = (V, E)$ mit $V = A \cup B$, $E = \{\{u, v\} \mid u \in A, v \in B\}$, $A \cap B = \emptyset$, $|A| = n$ und $|B| = m$.

Eulersche Polyeder-Formel: Für einen einfachen, zusammenhängenden, planaren Graphen $G = (V, E)$, der kreuzungsfrei in der Ebene gezeichnet ist, gilt $\mathcal{V} - \mathcal{E} + f = 2$, wobei f die Anzahl Flächen (faces) bezeichnet.

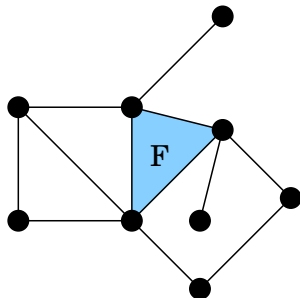
Beweis:



- Für einen Baum gilt: $\mathcal{E} = \mathcal{V} - 1$ und $f = 1$. Also gilt $\mathcal{V} - \mathcal{E} + f = 2$.
An dieser Stelle im Beweis geht die Voraussetzung *zusammenhängend* ein.
- Entferne nacheinander aus jedem Kreis in G je eine Kante: Jedesmal wird \mathcal{E} und f um 1 kleiner, die Differenz $\mathcal{V} - \mathcal{E} + f$ bleibt konstant. □

Korollar 1: Für einen planaren Graphen $G = (V, E)$ gilt: $\mathcal{E} \leq 3 \cdot \mathcal{V} - 6$

Beweis: Betrachte eine planare Einbettung von G mit f Flächen.



- Jede Kante e ist die Grenze von höchstens 2 Flächen.
→ p Kanten begrenzen maximal $2p$ Flächen.
- Dabei wird jede Fläche F mindestens dreimal gezählt, da F durch mindestens drei Kanten begrenzt ist. Beachte die Voraussetzung *einfacher Graph*.

$$\rightarrow f \leq \frac{2 \cdot \mathcal{E}}{3} \iff 3 \cdot f \leq 2 \cdot \mathcal{E}$$

Nach Eulers Polyeder-Formel gilt $\mathcal{V} - \mathcal{E} + f = 2$. Multiplizieren wir beide Seiten der Polyeder-Formel mit 3, dann erhalten wir:

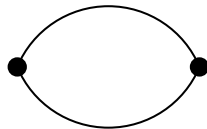
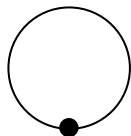
$$3 \cdot \mathcal{V} - 3 \cdot \mathcal{E} + 3 \cdot f = 6 \iff 3 \cdot f = 6 - 3 \cdot \mathcal{V} + 3 \cdot \mathcal{E}$$

Jetzt nutzen wir die Ungleichung $3 \cdot f \leq 2 \cdot \mathcal{E}$ als Abschätzung in der modifizierten Polyeder-Formel $3 \cdot f = 6 - 3 \cdot \mathcal{V} + 3 \cdot \mathcal{E}$ und erhalten:

$$3 \cdot f = 6 - 3 \cdot \mathcal{V} + 3 \cdot \mathcal{E} \leq 2 \cdot \mathcal{E} \implies \mathcal{E} \leq 3 \cdot \mathcal{V} - 6 \quad \square$$

Anmerkung: Planare Graphen $G = (V, E)$ sind dünne Graphen, es gilt also $\mathcal{E} \in \mathcal{O}(\mathcal{V})$.

Obiges Korollar gilt nur für einfache Graphen. Würden wir auch Schlingen oder Mehrfachkanten zulassen, dann könnte eine Fläche von weniger als drei Kanten begrenzt werden.



Korollar 2: Für einen bipartiten, einfachen, planaren Graphen $G = (V, E)$ gilt:
 $\mathcal{E} \leq 2 \cdot \mathcal{V} - 4$

Beweis: In bipartiten Graphen hat jeder Kreis eine gerade Länge. Also muss bei bipartiten Graphen sogar $4 \cdot f \leq 2 \cdot \mathcal{E}$ gelten, denn jede Fläche F wird dort mindestens viermal gezählt.

Nach der Eulerschen Polyeder-Formel gilt $\mathcal{V} - \mathcal{E} + f = 2$. Multiplizieren wir beide Seiten der Gleichung mit 4 erhalten wir:

$$4 \cdot \mathcal{V} - 4 \cdot \mathcal{E} + 4 \cdot f = 8 \iff 4 \cdot f = 8 - 4 \cdot \mathcal{V} + 4 \cdot \mathcal{E}$$

Nutzen wir $4 \cdot f \leq 2 \cdot \mathcal{E}$, dann erhalten wir:

$$4 \cdot f = 8 - 4 \cdot \mathcal{V} + 4 \cdot \mathcal{E} \leq 2 \cdot \mathcal{E} \implies 2 \cdot \mathcal{E} \leq 4 \cdot \mathcal{V} - 8$$

Dividieren wir beide Seiten durch 2, dann erhalten wir die zu beweisende Aussage. □

Satz: Der K_5 ist nicht planar.

Beweis: Im K_5 ist jeder der fünf Knoten mit jedem der vier anderen Knoten durch eine Kante verbunden. Da wir dabei jede Kante doppelt zählen, muss es also $\frac{4 \cdot 5}{2} = 10$ Kanten geben.

Nach Korollar 1 darf es aber höchstens $3 \cdot \mathcal{V} - 6 = 9$ Kanten geben. □

Satz: Jeder planare Graph besitzt einen Knoten v , der höchstens den Grad 5 hat.

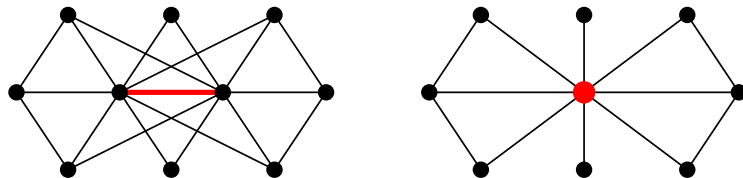
Beweis: Hätte jeder Knoten einen Knotengrad von mindestens 6, dann gäbe es mindestens $\frac{6 \cdot \mathcal{V}}{2} = 3 \cdot \mathcal{V}$ viele Kanten in dem Graphen, was nach Korollar 1 nicht sein kann. □

Satz: Der $K_{3,3}$ ist nicht planar.

Beweis: Im $K_{3,3}$ ist jeder Knoten mit drei anderen Knoten über eine Kante verbunden, es gibt also $\frac{6 \cdot 3}{2} = 9$ Kanten, nach Korollar 2 dürfte es aber nur $2 \cdot \mathcal{V} - 4 = 8$ Kanten geben. □

Wir wollen im Folgenden zu einem Graphen G testen, ob G planar ist. Beim Test spielen die Graphen K_5 und $K_{3,3}$ eine wichtige Rolle.

Kantenkontraktion:



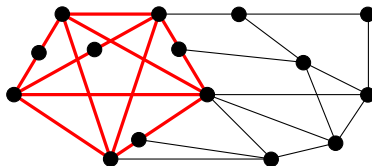
Ein Graph, der durch Löschen von Knoten oder Kanten oder durch Kantenkontraktionen aus G entsteht, heißt Graph-Minor von G .

Satz von Wagner: Jeder einfache Graph ist genau dann planar, wenn er weder den K_5 noch den $K_{3,3}$ als Minor enthält. (ohne Beweis)

Ein Graph H ist eine *Unterteilung* von G , wenn H aus G entsteht, indem Kanten von G durch einfache Wege über neu eingefügte Knoten ersetzt werden:

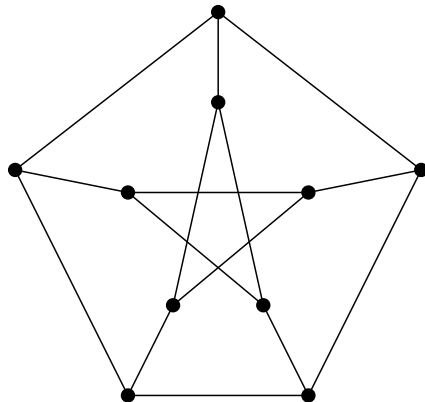


Alle neu eingefügten Knoten haben also den Grad 2. Ein Teilgraph G' eines gegebenen Graphen G wird *Kuratowski-Teilgraph* genannt, wenn G' eine Unterteilung des K_5 oder des $K_{3,3}$ ist.



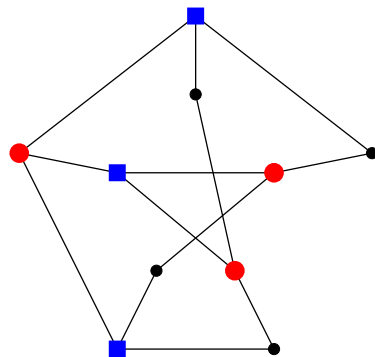
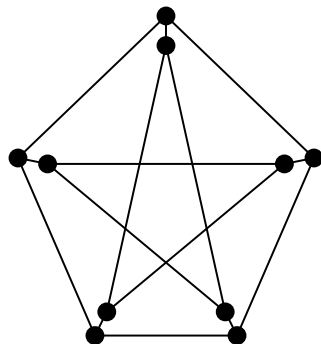
Satz von Kuratowski: Jeder einfache Graph ist genau dann planar, wenn er keinen Kuratowski-Teilgraphen enthält, wenn er also keine Unterteilung des K_5 oder des $K_{3,3}$ als Teilgraphen enthält. (ohne Beweis)

Frage: Ist der Peterson-Graph planar?



Antwort: Der Peterson-Graph ist nicht planar.

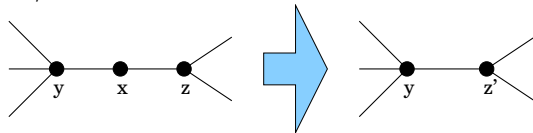
links: der K_5 entsteht durch Kantenkontraktion aus dem Peterson-Graph, enthält also den K_5 als Minor (\nexists zu Satz von Wagner)



rechts: der Peterson-Graph enthält einen Teilgraphen, der ein Unterteilungsgraph des $K_{3,3}$ ist (\nexists zu Satz von Kuratowski)

Aus dem Satz von Kuratowski ergibt sich ein Planaritätstest:

- kontrahiere Kanten, die zu einem Knoten mit Grad zwei inzident sind



- überprüfe im kontrahierten Graphen alle 5- bzw. 6-elementigen Teilmengen darauf, ob sie einen K_5 oder $K_{3,3}$ bilden

Anmerkungen:

- Laufzeit ist polynomiell aber nicht effizient.
- Nur Planaritätstest, liefert keine planare Einbettung.
- Effiziente Algorithmen:
 - Path-Addition-Algorithmus von Hopcroft und Tarjan
Realisierung in linearer Zeit mittels Datenstruktur PQ-Bäume
 - Vertex-Addition-Algorithmus von Lempel, Even, Cederbaum

Satz: Jeder planare Graph kann mit sechs Farben gefärbt werden.

Beweis: Es gibt einen Knoten v mit Knotengrad höchstens 5.

- Wir berechnen eine Färbung für $G - \{v\}$ mit 6 Farben.
- Anschließend fügen wir v mit den dazu gehörenden Kanten wieder ein.

Da der Knoten v in G höchstens 5 Nachbarn hat, ist noch eine Farbe für v frei. □

Satz: Jeder planare Graph kann mit fünf Farben gefärbt werden.

Beweis: Betrachte eine Einbettung von G in die Ebene.

Fall 1: Wenn ein Knoten v mit Knotengrad höchstens vier in G existiert, dann färben wir $G - \{v\}$ mit 5 Farben und fügen Knoten v mit den dazu gehörenden Kanten wieder ein.

Da Knoten v nur maximal vier Nachbarn hat, ist noch eine Farbe für Knoten v frei.

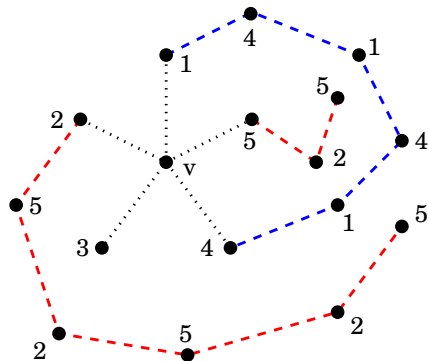
Fall 2: Alle Knoten haben mindestens den Grad fünf. Sei v ein Knoten mit Grad fünf. Auch hier berechnen wir zunächst eine Färbung von $G - \{v\}$ mit 5 Farben.

Falls die Nachbarn von v mit nur vier verschiedenen Farben gefärbt sind, kann v mit den dazu gehörenden Kanten eingefügt und mit der verbleibenden freien Farbe gefärbt werden.

Knotenfärbung planarer Graphen

Seien nun also die Nachbarn w_1, \dots, w_5 von v mit fünf verschiedenen Farben gefärbt, dabei sei w_i mit Farbe i gefärbt.

Falls w_1 und w_4 in dem durch die Knoten mit Farben 1 und 4 induzierten Teilgraphen H in verschiedenen Komponenten H_1 und H_4 liegen: Vertausche in H_1 die Farben 1 und 4, färbe v mit Farbe 1 und füge v mit den dazu gehörenden Kanten wieder ein.



Falls w_1 und w_4 in H verbunden sind, betrachte den durch die Knoten mit Farben 2 und 5 induzierten Teilgraphen H' . Die Knoten w_2 und w_5 können in H' nicht verbunden sein.

Vertausche die Farben 2 und 5 in der Komponente von w_2 in H' . Färbe v mit Farbe 2 und füge v mit den dazu gehörenden Kanten wieder ein. □

Satz: Jeder planare Graph kann mit vier Farben gefärbt werden.

- Dieser Satz wurde 1852 von *Augustus de Morgan* formuliert.
- *Alfred Kempe* liefert 1879 den ersten „Beweis“.
- 1890 fand *Percy Heawood* einen Fehler in Kempes Beweis.
- 1977 wurde der Satz von *Appel* und *Haken* bewiesen.
 - riesige Anzahl von Fallunterscheidungen nötig
 - Fallunterscheidungen wurden durch Computer gelöst
 - daher bei einigen Mathematikern umstritten
- 1995 geben *Robertson, Sanders, Seymour* und *Thomas* einen wesentlich kürzeren Beweis, allerdings werden immer noch Computer genutzt.

Aber: Es ist auch für planare Graphen NP-vollständig, zu entscheiden, ob drei Farben zur Knotenfärbung ausreichen.

3-Knotenfärbung für planare Graphen ist NP-vollständig

3-Knotenfärbung für planare Graphen ist in NP:

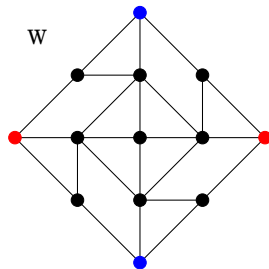
- Bestimme für den gegebenen Graphen $G = (V, E)$ nicht-deterministisch eine Zuordnung $f : V \rightarrow \{1, 2, 3\}$ von Farben zu Knoten.
- Prüfe für jede Kante $\{u, v\} \in E$, ob $f(u) \neq f(v)$ gilt.

3-Knotenfärbung \leq_p 3-Planar-Knotenfärbung

Zunächst benötigen wir eine Einbettung des Graphen G in die Ebene, und zwar so, dass sich in einem Punkt höchstens zwei Kanten schneiden.

Eine solche Kreuzung zweier Kanten wird durch eine Kopie des rechts stehenden Graphen W ersetzt.

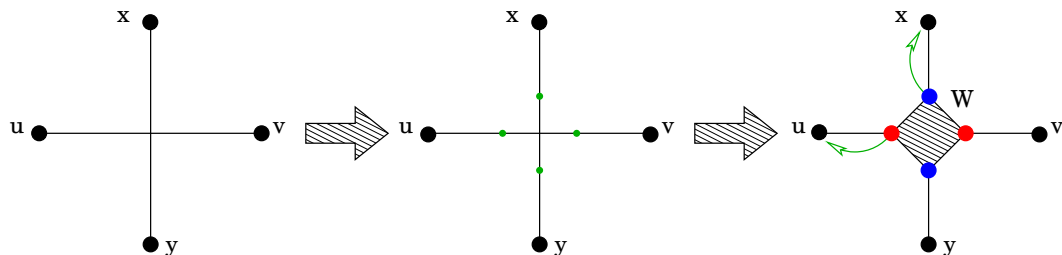
Die gefärbten „Eckknoten“ sind von zentraler Bedeutung.



3-Knotenfärbung für planare Graphen ist NP-vollständig

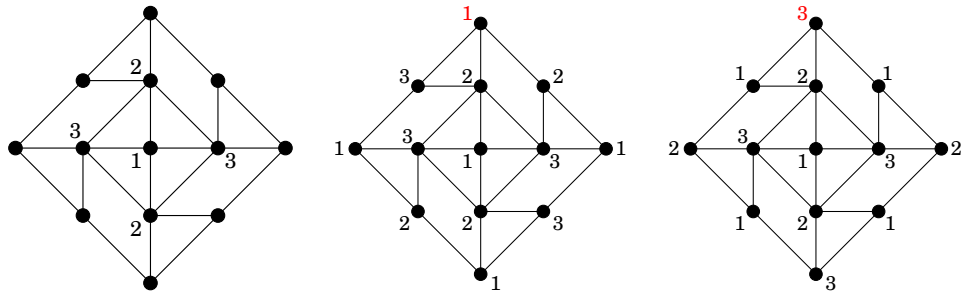
Die Ersetzung einer Kreuzung der Kanten $\{u, v\}$ und $\{x, y\}$ erfolgt in drei Schritten:

- Zunächst werden die beiden Kanten, die sich kreuzen, aufgeteilt.
- Dann wird eine Kopie des Graphen W eingefügt.
- Schließlich werden u und x mit je einem Knoten aus W verschmolzen.



3-Knotenfärbung für planare Graphen ist NP-vollständig

Um die obige Aussage zu verstehen, färben wir zunächst den mittleren Knoten mit einer Farbe. Die umliegenden Knoten müssen dann abwechselnd mit den anderen zwei Farben gefärbt werden, siehe linke Abbildung.

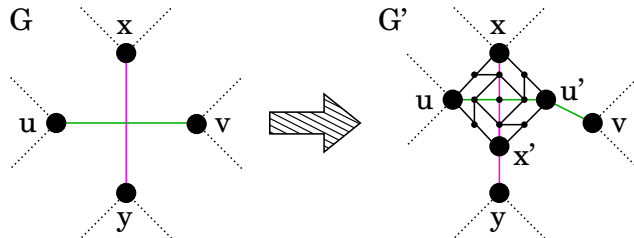


Der obere Knoten kann dann mit einer von zwei Farben gefärbt werden. Die Färbung der verbleibenden Knoten ist damit festgelegt: im mittleren Bild im Uhrzeigersinn, im rechten Bild entgegen dem Uhrzeigersinn.

Andere Färbungen ergeben sich, wenn der mittlere Knoten anders gefärbt wird oder seine umliegenden Knoten.

3-Knotenfärbung für planare Graphen ist NP-vollständig

Der aus Graph G durch obige Transformation entstehende Graph G' ist planar, in polynomieller Zeit berechenbar und es gilt: G ist 3-färbbar $\iff G'$ ist 3-färbbar



\Rightarrow G ist 3-färbbar

Es existiert eine Färbung f' für G' , sodass $f(u) = f'(u) = f'(u')$ und $f(x) = f'(x) = f'(x')$ gilt, und $f(v) = f'(v)$ und $f(y) = f'(y)$.

\Leftarrow G' ist 3-färbbar

Für jede Färbung f' gilt $f'(u) = f'(u')$ und $f'(x) = f'(x')$, daher ist f' eingeschränkt auf die Knotenmenge V eine Färbung von G .

- Einleitung
- Entwurfsmethoden
- Sortieren
- Auswahlproblem
- Graphalgorithmen
- Spezielle Graphklassen
- *Vorrangwarteschlangen*
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

Vorrangwarteschlangen

- *Motivation*
- Linksbäume
- Binomial-Queues
- Fibonacci-Heaps

Die Algorithmen von

- *Dijkstra* zur Berechnung *kürzester Wege* sowie
- *Prim* zur Berechnung *minimaler Spannbäume*

verwenden eine Datenstruktur zum Speichern von Knoten mit Operationen *ExtractMin* und *DecreaseKey*.

Wir kennen drei mögliche Implementierungen:

	Array	Linked List	Binary Heap
ExtractMin	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
DecreaseKey	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

DecreaseKey setzt voraus, dass jeweils ein Zeiger auf jedes Element vorhanden ist, um das Element effizient zu finden, um also keine Zeit mit Suchen zu verschwenden.

Zugriffsmethoden: (komplett)

	Linked List	Binary Heap
MakeHeap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\log n)$
Minimum	$\Theta(n)$	$\Theta(1)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$
Union	$\Theta(1)$	$\Theta(n)$
DecreaseKey	$\Theta(1)$	$\Theta(\log n)$
Delete	$\Theta(1)$	$\Theta(\log n)$

- Auch bei **Delete** muss das Element direkt zugreifbar sein.
- Lassen sich Vorteile der verketteten Listen und Heaps vereinigen?
- Können die rot markierten Laufzeiten verbessert werden?

Folgende Operationen muss eine Priority Queue unterstützen:

- `MakeHeap()` erzeugt einen neuen Heap ohne Elemente.
- `Insert(H, x)` fügt Knoten x in Heap H ein.
- `Minimum(H)` liefert einen Zeiger auf den Knoten mit minimalem Element im Heap H .
- `ExtractMin(H)` entfernt das minimale Element aus Heap H und liefert einen Zeiger auf den Knoten.
- `Union(H_1, H_2)` erzeugt einen neuen Heap, der die Elemente aus H_1 und H_2 enthält.
- `DecreaseKey(H, x, k)(*)` weist dem Knoten x im Heap H einen neuen, kleineren Wert k zu.
- `Delete(H, x)(*)` entfernt Knoten x aus Heap H .

(*) Der Zeiger auf x muss bekannt sein, um Suchen zu vermeiden.

Vorrangwarteschlangen

- Motivation
- *Linksbäume*
- Binomial-Queues
- Fibonacci-Heaps

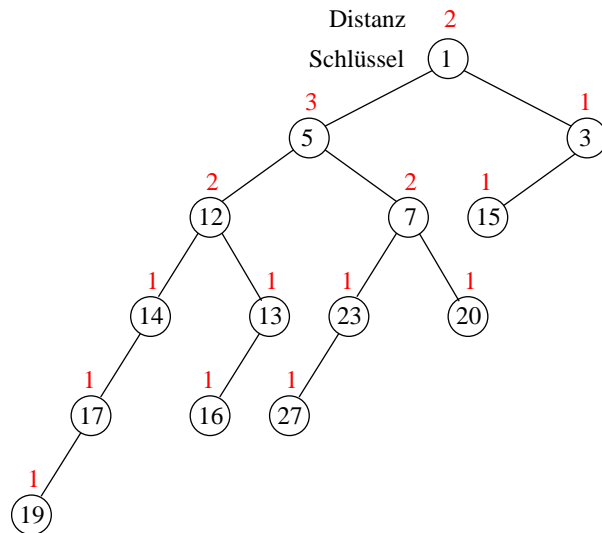
Ein Baum mit $N + 1$ Blättern und N inneren Knoten ist balanciert, wenn jedes Blatt eine Tiefe aus $\mathcal{O}(\log(N))$ hat.

Implementierung von Priority-Queues: Es reicht eine wesentlich schwächere Forderung, um obige Operationen in der vorgegebenen Zeit auszuführen.

Linksbäume (leftist trees) sind binäre, heap-geordnete, links-rechts-geordnete Bäume, die in ihren inneren Knoten Elemente mit Schlüsseln speichern.

In unseren Abbildungen sind die Blätter nicht gezeichnet.

Beispiel:



Eigenschaften:

- Der Schlüsselwert eines Knoten ist immer kleiner als die Schlüsselwerte aller seiner Kinder: Min-Heap!
- Jeder Knoten besitzt einen Distanzwert.
 - Blätter: Der Distanzwert ist 0.
 - Innere Knoten: Distanzwert des rechten Kindes plus 1.
 - Distanzwert rechtes Kind \leq Distanzwert linkes Kind.
- Aufgrund der letzten Bedingung hat der Linksbaum seinen Namen: Der Baum „hängt“ nach links, er ist links tiefer als rechts.

Wir werden zeigen: Sei w ein Knoten mit $dist(w) = k$. Dann enthält der Teilbaum mit Wurzel w mindestens 2^k Knoten.

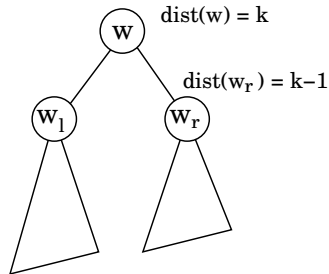
Daher gilt: In einem Linksbaum hat das rechteste Blatt eine Tiefe von höchstens $\mathcal{O}(\log(N))$.

Lemma: Sei w ein Knoten mit $\text{dist}(w) = k$. Dann enthält der Teilbaum mit Wurzel w mindestens 2^k Knoten.

Beweis mittels vollständiger Induktion:

I.A. Für $\text{dist}(w) = 0$ gilt: w ist ein Blatt und der Teilbaum mit Wurzel w besteht aus $2^0 = 1$ Knoten. ✓

I.S. Der Knoten w habe den linken Nachfolger w_l und den rechten Nachfolger w_r .



Dann gilt $k = \text{dist}(w) = \text{dist}(w_r) + 1$ und $\text{dist}(w_l) \geq \text{dist}(w_r)$ nach Definition.

Damit haben die Teilbäume mit den Wurzeln w_l und w_r nach I.V. mindestens 2^{k-1} Knoten.

Der Teilbaum mit Wurzel w enthält also mindestens $2^{k-1} + 2^{k-1} = 2^k$ Knoten. ✓

MAKEHEAP(): Erzeuge ein Blatt mit Distanzwert 0.

→ Laufzeit: $\mathcal{O}(1)$

MINIMUM(*D*): Gib das Wurzel-Element zurück.

→ Laufzeit: $\mathcal{O}(1)$

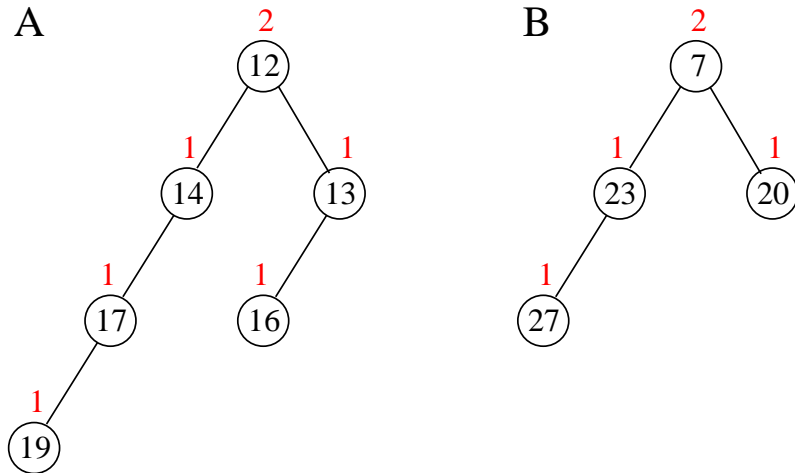
Alle anderen Operationen lassen sich auf das Verschmelzen von zwei Linksbäumen zurückführen!

$\text{UNION}(D_1, D_2)$:

rekursiv

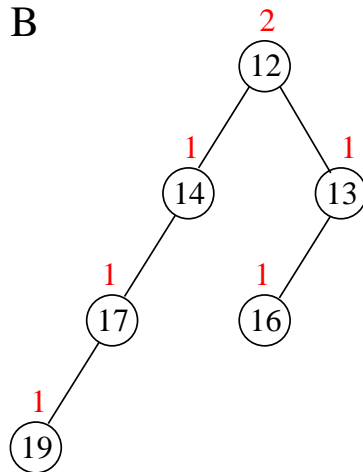
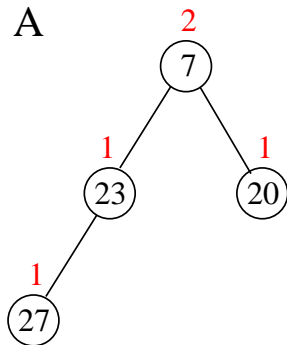
- Wenn D_1 oder D_2 ein Blatt ist, dann ist das Ergebnis der Linksbaum D_2 bzw. D_1 .
- o.B.d.A: Der Schlüssel an Wurzel von D_1 ist kleiner als der Schlüssel an Wurzel von D_2 – sonst tausche Linksbäume.
 - (a) $\text{UNION}(D_1.\text{RECHTS}, D_2)$
 - (b) Ist die Distanz vom (neuen) rechten Teilbaum von D_1 größer als die Distanz vom linken Teilbaum von D_1 , dann werden die Teilbäume von D_1 vertauscht.

UNION(A,B): Initial



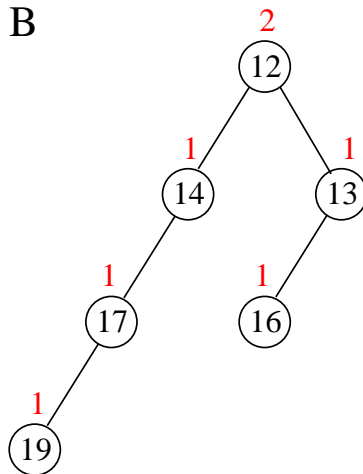
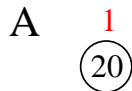
→ Bäume vertauschen

UNION(A,B): Bäume vertauscht



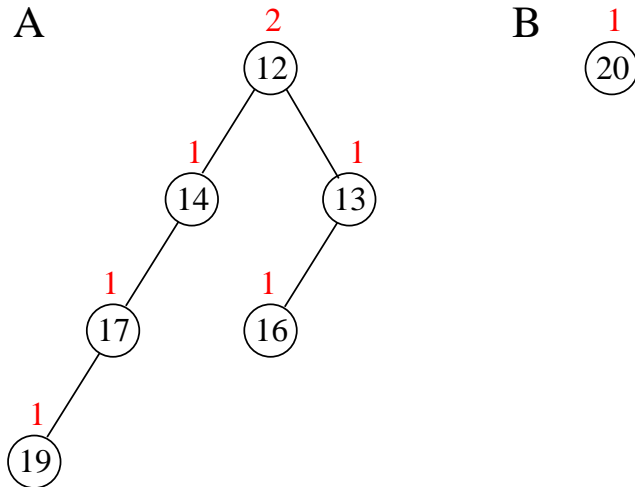
→ verschmelzen von A.rechts und B

$\text{UNION}(A,B)$: verschmelzen von A und B



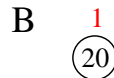
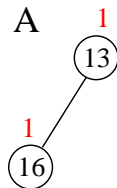
→ Bäume vertauschen

UNION(A,B): Bäume vertauscht

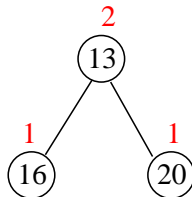


→ verschmelzen von A.rechts und B

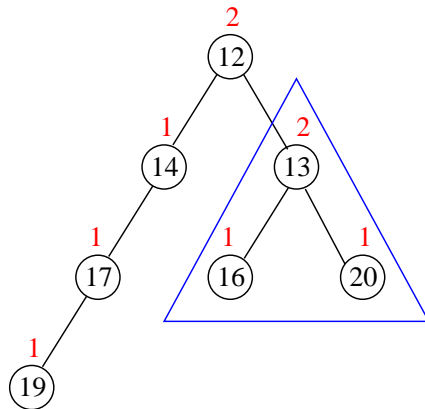
$\text{UNION}(A,B)$: verschmelzen von A und B



jetzt besteht A.rechts nur noch aus einem Blatt
→ Ergebnis der letzten Rekursion:

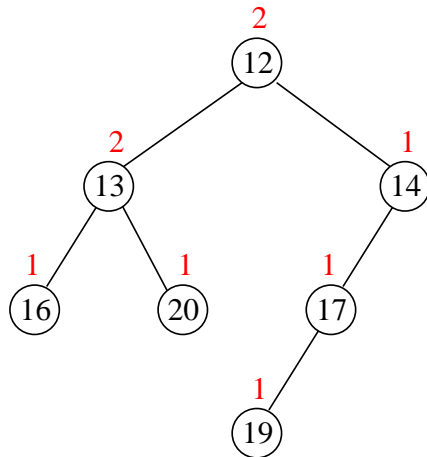


$\text{UNION}(A,B)$: vorläufiges Ergebnis der zweiten Rekursion

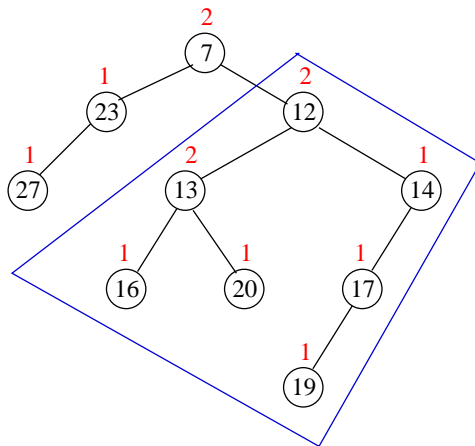


Distanzwert rechts (neu) größer als Distanzwert links \rightarrow vertausche die beiden Teilbäume

$\text{UNION}(A,B)$: Ergebnis der zweiten Rekursion

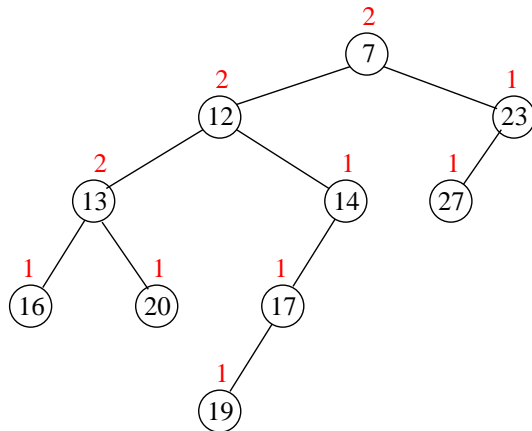


$\text{UNION}(A,B)$: vorläufiges Ergebnis der ersten Rekursion



Distanzwert rechts (neu) größer als Distanzwert links \rightarrow vertausche die beiden Teilbäume

UNION(A,B): Ergebnis der ersten Rekursion



Laufzeit: Ist beschränkt durch die Summe der Längen der beiden Pfade von der Wurzel zum jeweils rechtesten Blatt. $\rightarrow \mathcal{O}(\log(N))$

INSERT(D, x):

- Erzeuge einen Linksbaum E , der nur einen einzigen inneren Knoten x mit Distanz 1 hat.
 - UNION(D, E)
- Laufzeit: $\mathcal{O}(\log(N))$

EXTRACTMIN(D):

- Entferne die Wurzel.
 - Verschmelze die beiden Teilbäume der Wurzel.
- Laufzeit: $\mathcal{O}(\log(N))$

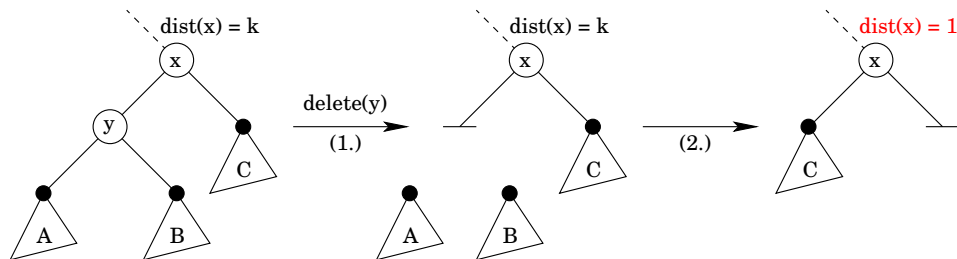
DELETE(D, x):

- Der Linksbaum zerfällt beim Entfernen von x in den oberhalb von x liegenden Teilbaum und in den linken und rechten Teilbaum.
- Ersetze den Knoten x durch ein Blatt b .
- Tausche ggf. b mit seinem Geschwister, um links den Teilbaum mit größerer Distanz anzuordnen.
- Adjustiere die Distanzwerte von b bis zur Wurzel.
- Verschmelze die drei Teilbäume miteinander.

→ Laufzeit: $\mathcal{O}(\log(N))$

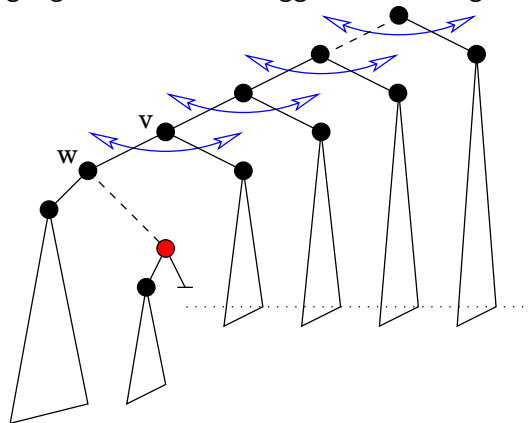
Warum? Das Adjustieren der Distanzen und das Vertauschen von Teilbäumen auf dem Pfad zur Wurzel ist doch proportional zur Höhe des Baumes, also $\Theta(n)$, oder?

Beispiel: Im folgenden Ausschnitt eines Linksbiums wird Knoten y gelöscht.



- Zunächst werden die Teilbäume A und B vom zu löschenden Knoten abgetrennt. Diese werden später mittels `UNION` dem Linksbium wieder hinzugefügt.
- Da der Vorgänger x einen nicht-leeren, rechten Teilbaum C hat, müssen der linke und rechte Nachfolger von x getauscht und die Distanz von x korrigiert werden.
- Korrigiere Vorgängerdistanzen, falls x rechter Nachfolger ist.

Wenn beim Korrigieren der Vorgängerdistanzen ein Knoten w erreicht wird, der linker Nachfolger seines Vorgängers v ist, müssen ggf. Teilbäume getauscht werden.



Dies erfolgt aber nur, wenn der rechte Teilbaum von v tiefer ist als der linke. Setzt sich das Tauschen bis zur Wurzel fort, ist der Baum bis w im Wesentlichen balanciert und hat nur logarithmische Tiefe.

DECREASEKEY(D, x, k):

- DELETE(D, x)
 - Ändere den Schlüssel von x auf k .
 - INSERT(D, x)
- Laufzeit: $\mathcal{O}(\log(N))$

Zusammenfassung

	Linked List	Binary Heap	Linksbaum
MakeHeap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
DecreaseKey	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Delete	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Übung:

- Fügen Sie die Werte 1,2,3,4,5,6,7 in dieser Reihenfolge in einen initial leeren Linksbaum ein.
- Führen Sie anschließend `EXTRACTMIN()` und danach `DECREASEKEY(6, 1)` aus.
- Geben Sie nach jedem Schritt den resultierenden Linksbaum an.
- Kann ein Linksbaum zu einer linearen Liste entarten? Begründen Sie Ihre Antwort.

Vorrangwarteschlangen

- Motivation
- Linksbäume
- *Binomial-Queues*
- Fibonacci-Heaps

Binomialbäume sind heap-geordnete Bäume, die in allen Knoten Elemente mit Schlüsseln speichern. (Min-Heap)

Ein Binomialbaum vom Typ

- B_0 besteht aus genau einem Knoten.
- B_{i+1} , $i \geq 0$, besteht aus zwei Kopien der Binomialbäume vom Typ B_i , indem man die Wurzel der einen Kopie zum Kind der Wurzel der anderen Kopie macht.

Ihren Namen verdanken die Binomial-Queues B_k der Tatsache, dass auf der Ebene i die Anzahl der Knoten gerade $\binom{k}{i}$ beträgt.

Binomial Heap

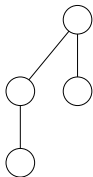
B_0



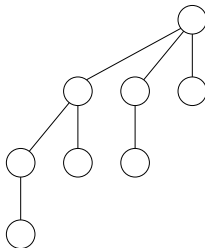
B_1



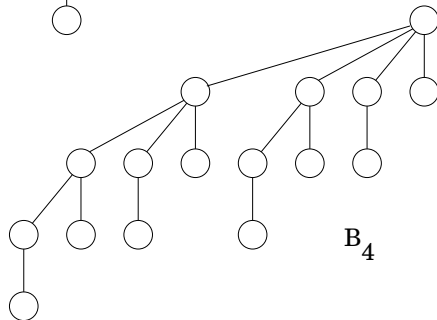
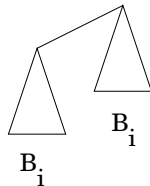
B_2



B_3



B_{i+1}



Eigenschaften eines Binomialbaums vom Typ B_k :

- 1 B_k hat 2^k Knoten und Höhe k .
- 2 In Ebene i hat B_k genau $\binom{k}{i}$ viele Knoten.
- 3 Teilbäume der Wurzel sind vom Typ $B_{k-1}, B_{k-2}, \dots, B_0$.

Beweis: Induktion nach k . Für $k = 0$ gelten die Aussagen.

- 1 B_k entsteht, indem ein B_{k-1} unter einen anderen B_{k-1} gehängt wird, also gilt:
 - Anzahl Knoten: $2^{k-1} + 2^{k-1} = 2^k$
 - $\text{depth}(B_k) = \text{depth}(B_{k-1}) + 1 = (k-1) + 1 = k$.
- 2 Sei $D(k, i)$ die Anzahl der Knoten in Tiefe i von B_k . Da ein B_{k-1} unter einen anderen B_{k-1} gehängt wird, gilt:
$$D(k, i) = D(k-1, i) + D(k-1, i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$
- 3 siehe Bild auf vorheriger Folie
 - aus B_0 und B_0 entsteht B_1 , also ein Teilbaum B_0
 - aus B_1 und B_1 entsteht B_2 , also ein Teilbaum B_1 usw.

Speichern von n Elementen: Die Anzahl der benötigten Bäume ist gleich der Anzahl der Einsen in der Binärdarstellung von n . Sei $n = (d_{m-1} \dots d_0)$. Dann gilt:

$$\text{Baum vom Typ } B_j \text{ wird benötigt} \iff d_j = 1$$

Beispiel: Um $n = 13 = (1101)_2$ in einem Binomial-Heap zu speichern, werden Bäume vom Typ B_3 , B_2 und B_0 benötigt.

Eine Menge von Bäumen nennt man in der Graphentheorie (und nicht nur da 😊) einen *Wald*.

Ein *Binomial-Heap vom Typ D_n* ist die Repräsentation einer Menge mit n Elementen durch Binomialbäume.

- Binomial-Heaps werden auch Binomial-Queues genannt.
- Die Anzahl der Bäume ist höchstens $\log(n)$.
- Die Wurzeln der Bäume werden in einer verketteten Liste gespeichert.

MakeHeap() : Erzeuge eine neue, leere Wurzelliste.

→ Laufzeit: $\Theta(1)$

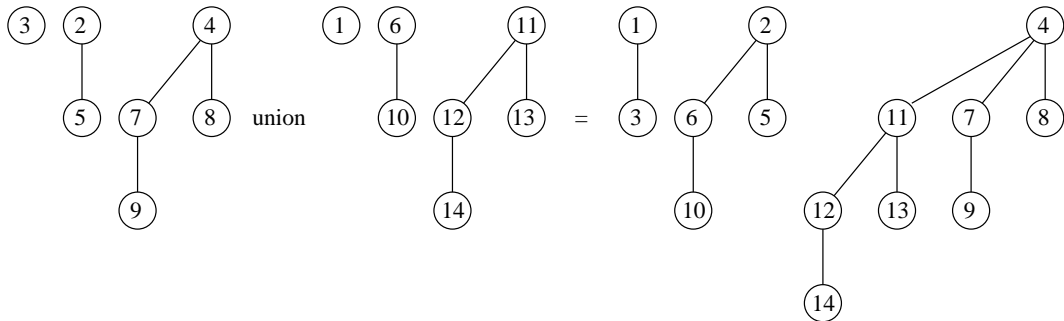
Minimum(H) : Durchsuche die Wurzelliste.

→ Laufzeit: $\Theta(\log(n))$

$\text{Union}(H_1, H_2)$: Vereinige die Bäume aus beiden Binomial-Queues wie folgt:

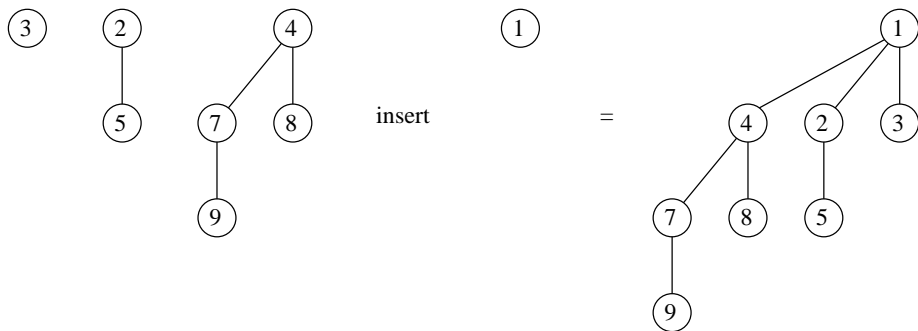
- Zwei Bäume vom Typ B_0 ergeben einen Baum vom Typ B_1 .
- Zwei Bäume vom Typ B_1 ergeben einen Baum vom Typ B_2 .
- usw.

analog zur Addition zweier Dualzahlen



→ Laufzeit: $\Theta(\log(n + m))$

$\text{Insert}(H, x)$: Vereinige H und den Baum vom Typ B_0 , der den Knoten x enthält.

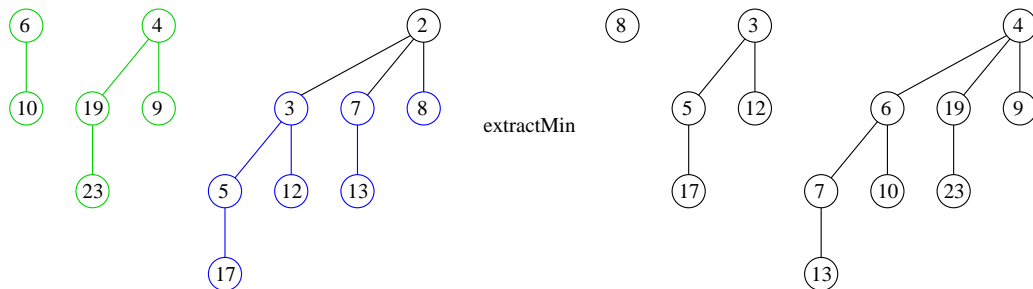


→ Laufzeit: $\Theta(\log(n))$

ExtractMin(H):

- Sei B_j der Baum, der das minimale Element in seiner Wurzel speichert.
- Sei D_{n-2^j} der Wald ohne Baum B_j , unten grün markiert.
- Sei D_{2^j-1} der Wald, wenn im Baum B_j die Wurzel entfernt wird, unten blau markiert.

⇒ Vereinige D_{n-2^j} und D_{2^j-1} .



→ Laufzeit: $\Theta(\log(n))$

DecreaseKey(H, x, k):

- Setze den Schlüssel von Knoten x auf den Wert k .
- **UpHeap(H, x):** Laufe im Baum hoch und vertausche den Knoten mit seinem Vorgänger, falls der Vorgänger einen größeren Wert speichert.

→ Laufzeit: $\Theta(\log(n))$

Delete(H, x):

- DecreaseKey($H, x, -\infty$)
- ExtractMin(H)

→ Laufzeit: $\Theta(\log(n))$

Zusammenfassung

Operation	Linked List	Binary Heap	Links-baum	Binomial Heap
MAKEHEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
MINIMUM	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
EXTRACTMIN	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
UNION	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
DECREASEKEY	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
DELETE	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Übung:

- Fügen Sie die Werte 1,2,3,4,5,6,7 in dieser Reihenfolge in einen initial leeren Binomial Heap ein.
- Führen Sie anschließend `EXTRACTMIN()` und danach `DECREASEKEY(6, 1)` aus.
- Geben Sie nach jedem Schritt den resultierenden Binomial Heap an.

Vorrangwarteschlangen

- Motivation
- Linksbäume
- Binomial-Queues
- *Fibonacci-Heaps*

Noch nicht zufriedenstellend beschleunigte Operationen:

- INSERT
- DECREASEKEY
- DELETE

Idee der Fibonacci-Heaps:

- Verzichte beim Einfügen und Löschen auf die Vereinigung der Bäume.
- Hole dies erst bei EXTRACTMIN nach.
- Vermerke nach jeder Operation das minimale Element.

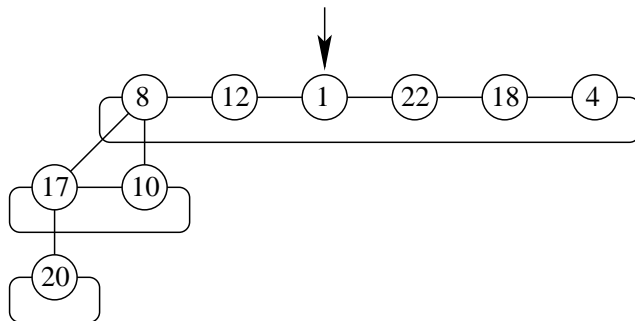
Fibonacci-Heap:

- Sammlung heap-geordneter Bäume. (Min-Heap)
 - Struktur implizit durch erklärte Operationen definiert.
- ⇒ Jede mit den bereitgestellten Operationen aufbaubare Struktur ist ein Fibonacci-Heap.

Implementierung:

- Die Wurzeln der Bäume sind doppelt zyklisch verkettet.
- Zeiger auf das kleinste Element der Wurzelliste.
- Kinder der Knoten ebenfalls doppelt zyklisch verkettet.

Beispiel:



MAKEHEAP(): Erzeuge leeren Fibonacci-Heap: NULL-Zeiger

→ Laufzeit: $\Theta(1)$

MINIMUM(*H*): Element, worauf Minimalzeiger zeigt.

→ Laufzeit: $\Theta(1)$

UNION(H_1, H_2):

- Hänge die Wurzellisten von H_1 und H_2 aneinander.
- Setze neuen Minimalzeiger auf Minimum der Minimalelemente von H_1 und H_2 .

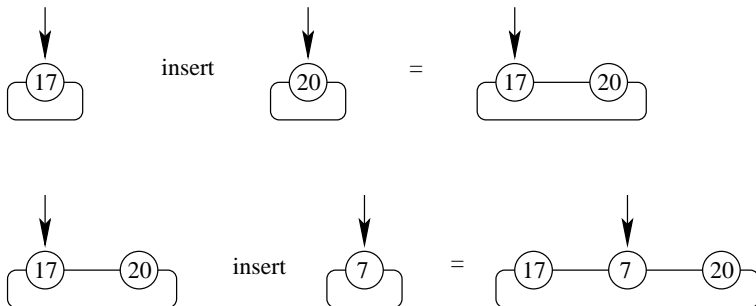
→ Laufzeit: $\Theta(1)$

Weiteres Implementierungsdetail: Jeder Knoten hat einen *Rang* und ein *Markierungsfeld*.

- Der *Rang* entspricht der Anzahl der Kinder.
- Der Sinn des *Markierungsfeldes* wird bei DECREASEKEY erklärt.

INSERT(H, x):

- Erzeuge einen Fibonacci-Heap H' , der nur x enthält.
- Der Rang von x ist 0, das Element ist nicht markiert.
- Vereinige H und H' mit der Operation UNION.



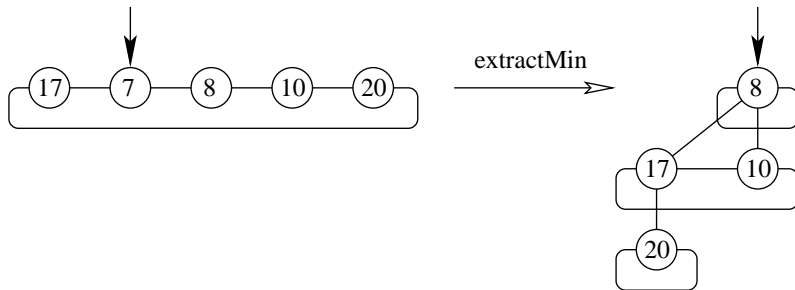
→ Laufzeit: $\Theta(1)$

EXTRACTMIN(H):

- Entferne den Minimalknoten u aus der Wurzelliste und bilde eine neue Wurzelliste durch Einhängen der Liste der Kinder von u an Stelle von u .
→ Laufzeit $\Theta(1)$, weil die Kinder zyklisch verkettet sind.
- Consolidate: Verschmelze nun solange zwei Bäume vom selben Rang, bis die Wurzelliste nur noch Bäume mit paarweise verschiedenem Rang enthält.
 - Beim Verschmelzen (kein Union!) zweier Bäume B und B' vom Rang i entsteht ein Baum vom Rang $i + 1$.
 - Ist das Element in der Wurzel v von B größer als das Element in der Wurzel v' von B' , dann wird v ein Kind von v' und das Markierungsfeld von v wird auf nicht markiert gesetzt.
 - Aktualisiere auch den Minimalzeiger.

Laufzeit: $\mathcal{O}(n)$ im worst-case, aber amortisiert nur $\mathcal{O}(\log(n))$.

Beispiel:

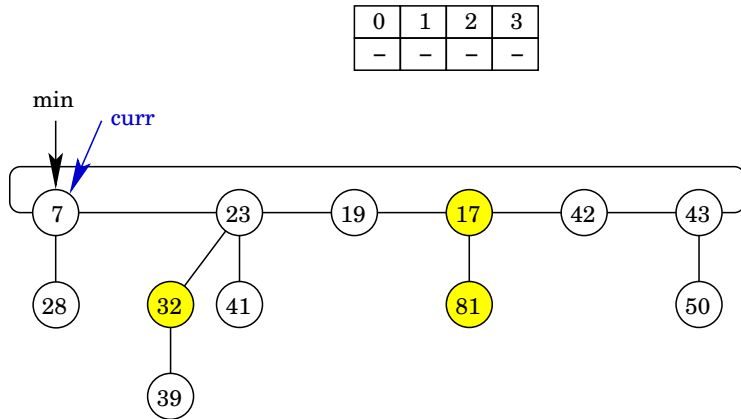


Effiziente Implementierung des Consolidate-Schrittes:

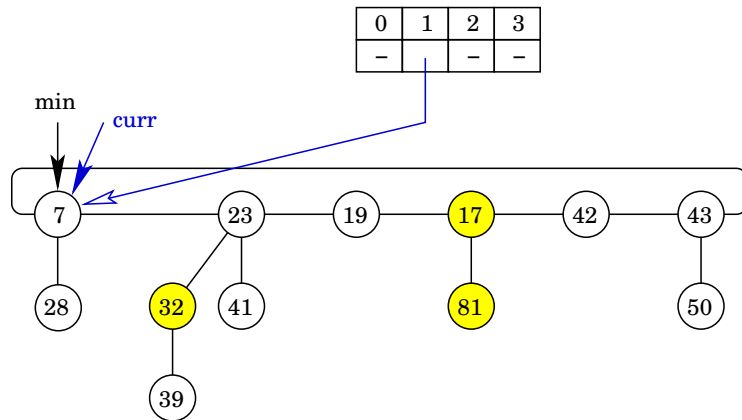
- verwende temporäres Array: Element $A[i]$ zeigt auf einen Knoten x , so dass x die Wurzel eines Baums mit Rang i gilt.
- durchlaufe die Wurzelliste beginnend beim `min`-Zeiger:
 - sei `curr` der Zeiger auf den aktuell untersuchten Knoten
 - falls $A[\text{rang}(\text{curr})] == \text{NULL}$ ist, füge `curr` hier ein
 - sonst: verschmelze die Bäume und füge den neuen Baum bei der Position $\text{rang}(\text{curr}) + 1$ ein \rightarrow evtl. Überlauf behandeln

Fibonacci-Heap

Consolidate: initial

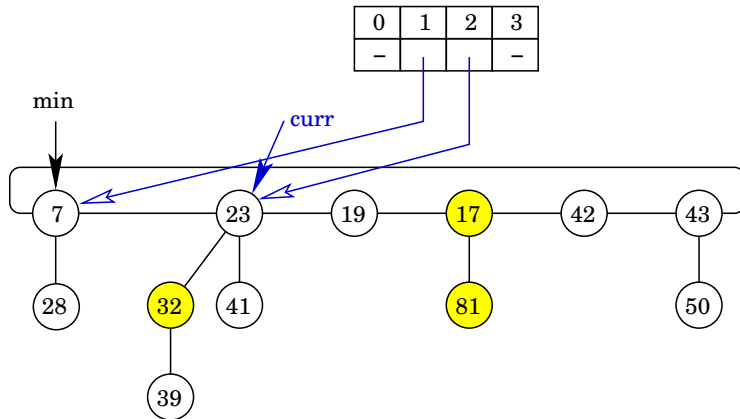


Consolidate: trage Baum mit Rang 1 in das Array ein



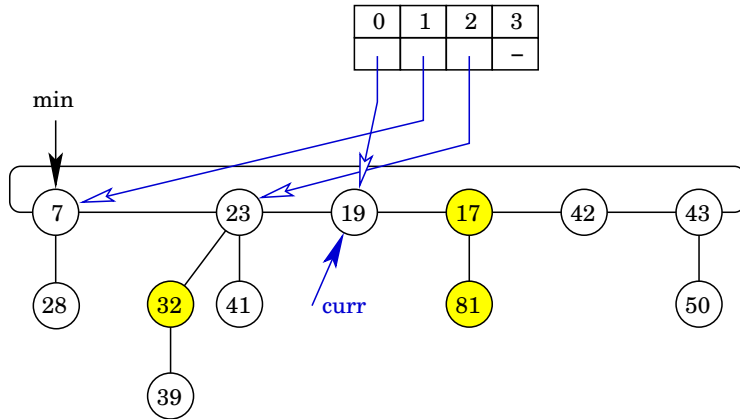
dann: setze Zeiger curr auf das nächste Element der Wurzelliste

Consolidate: trage Baum mit Rang 2 in das Array ein



dann: setze Zeiger curr auf das nächste Element der Wurzelliste

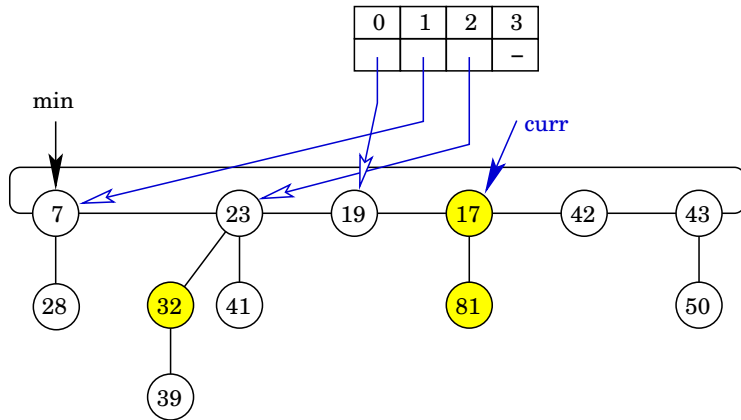
Consolidate: trage Baum mit Rang 0 in das Array ein



dann: setze Zeiger curr auf das nächste Element der Wurzelliste

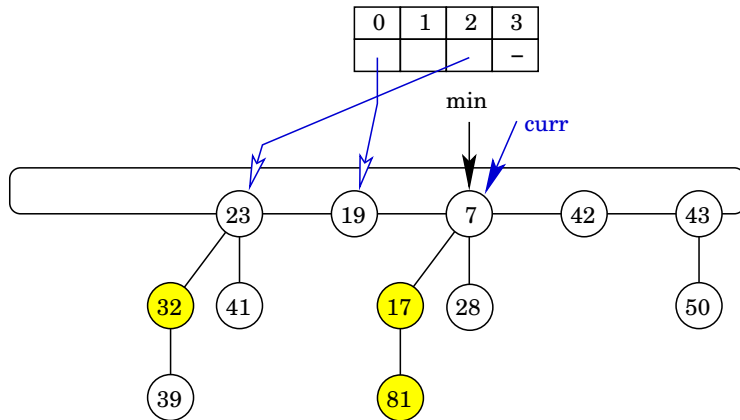
Fibonacci-Heap

Consolidate: A[1] ist bereits belegt



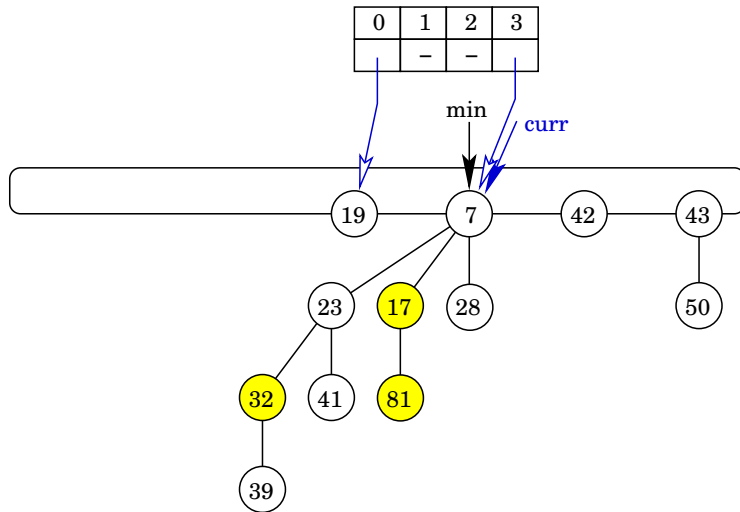
also: verschmelze die Bäume mit Wurzeln 7 und 17

Consolidate: $A[2]$ ist bereits belegt \rightarrow Überlauf behandeln



also: verschmelze die Bäume mit Wurzeln 7 und 23

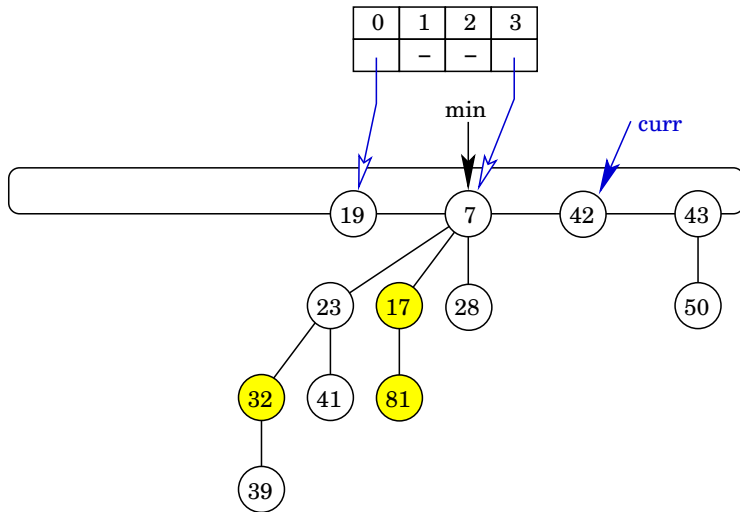
Consolidate: trage neuen Baum mit Rang 3 in das Array ein



dann: setze Zeiger curr auf das nächste Element der Wurzelliste

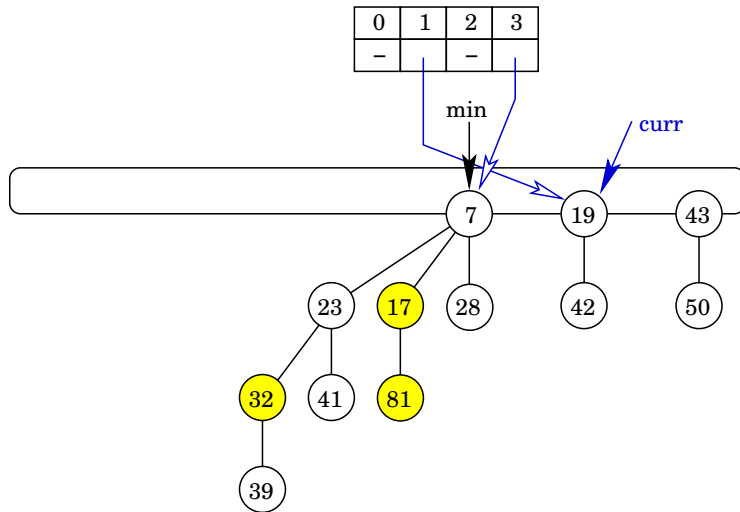
Fibonacci-Heap

Consolidate: A[0] ist bereits belegt



also: verschmelze die Bäume mit Wurzeln 19 und 42

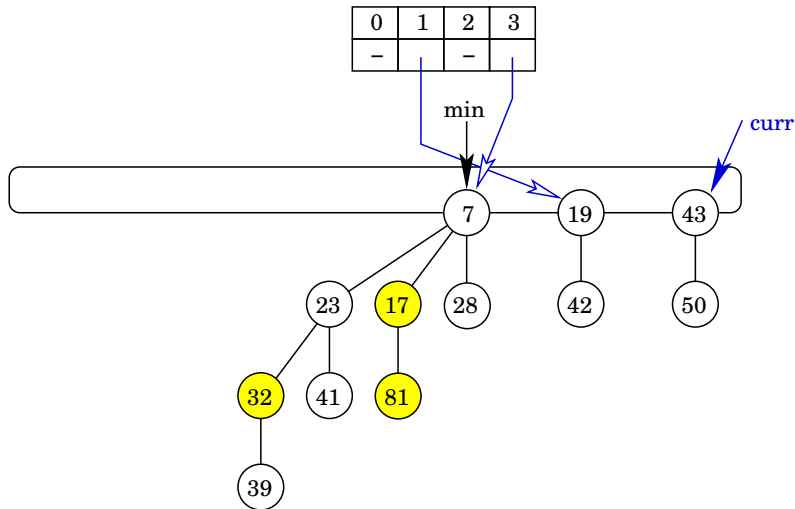
Consolidate: trage neuen Baum mit Rang 1 in das Array ein



dann: setze Zeiger curr auf das nächste Element der Wurzelliste

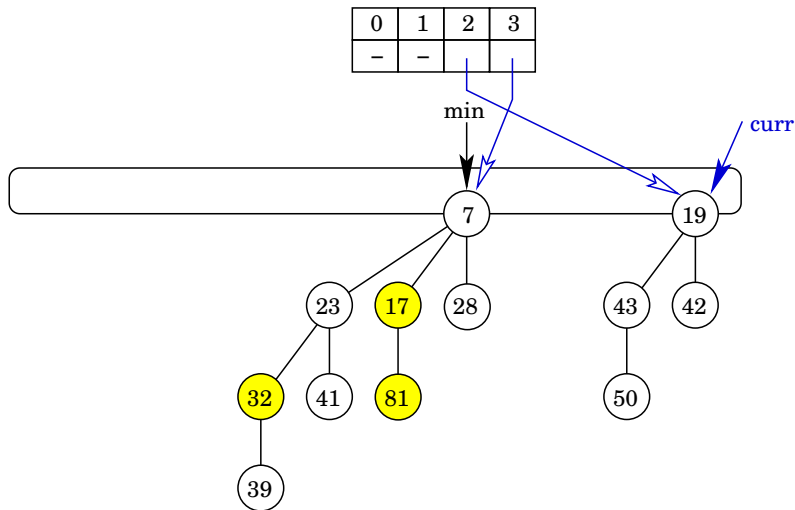
Fibonacci-Heap

Consolidate: A[1] ist bereits belegt



also: verschmelze die Bäume mit Wurzeln 19 und 43

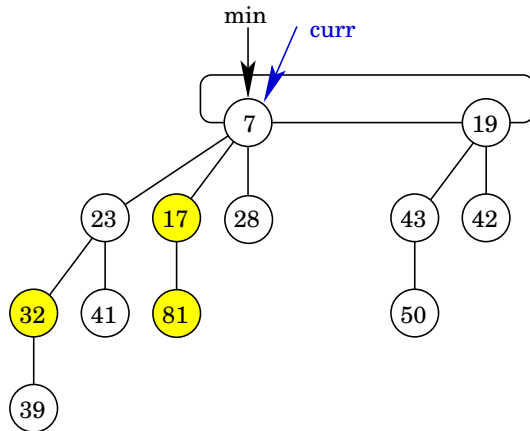
Consolidate: trage neuen Baum mit Rang 2 in das Array ein



dann: setze Zeiger curr auf das nächste Element der Wurzelliste

Fibonacci-Heap

Consolidate: Ende, denn curr zeigt wieder auf min



Laufzeit?

DECREASEKEY(H, x, k):

- Trenne x von seinem Elter.
- Verkleinere den Wert auf k .
- Hänge den heap-geordneten Baum in die Wurzelliste.
- Aktualisiere den Minimalzeiger.

→ Laufzeit: $\Theta(1)$

Wie verhindern wir, dass die Bäume zu „dünn“ werden, also zuviele Knoten verlieren?

Es soll verhindert werden, dass mehr als zwei Kinder von seinem Elter abgetrennt werden:

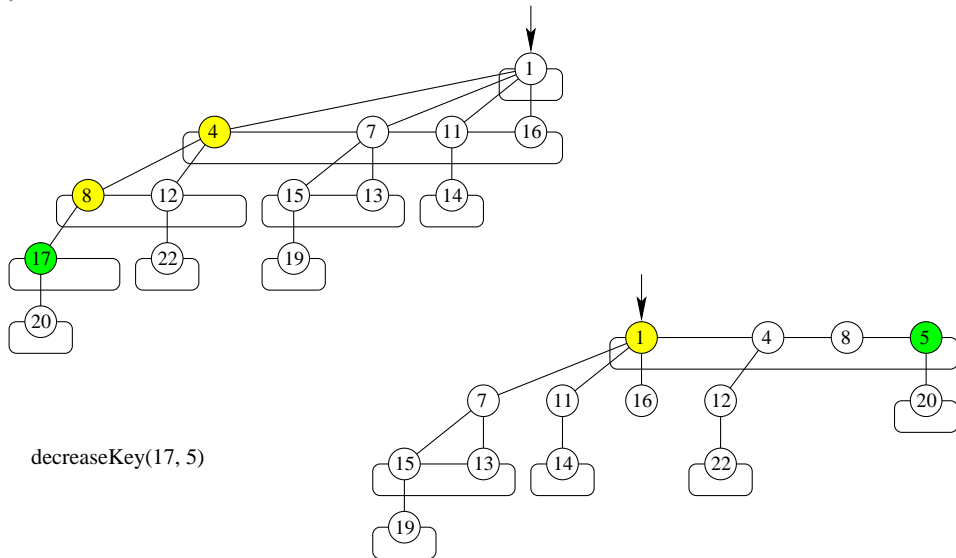
- Beim Abtrennen eines Knotens p von seinem Elter v wird v markiert.
- War v bereits markiert, wird auch v von seinem Elter v' abgetrennt und v' markiert, usw.
- Alle abgetrennten Teilbäume werden in die Wurzelliste aufgenommen, die Markierungen der jeweiligen Wurzeln werden gelöscht.

Laufzeit:

- $\mathcal{O}(\log(n))$ im worst-case
- $\mathcal{O}(1)$ amortisiert

Fibonacci-Heap

Beispiel:



DELETE(H, x): Setze x auf einen sehr kleinen Schlüssel mittels **DECREASEKEY** und führe dann **EXTRACTMIN**(H) aus.

- $\mathcal{O}(n)$ im worst-case
- $\mathcal{O}(\log(n))$ amortisiert

Amortisierte Laufzeit:

- Durchschnittliche, maximale Kosten einer Operation bei einer beliebigen Folge von Operationen.
- Idee: Eine einzelne **EXTRACTMIN**-Operation kann zwar sehr lange dauern, aber dann wird die Datenstruktur „aufgeräumt“ und die nächste Ausführung dauert dann nicht so lange.

Zusammenfassung

Operation	Linked List ^(*)	Binary Heap ^(*)	Binomial Heap ^(*)	Fibonacci Heap ^(**)
MAKEHEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
EXTRACTMIN	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
UNION	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
DECREASEKEY	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

(*) worst-case-Laufzeit

(**) amortisierte Laufzeit

Übung:

- Fügen Sie die Werte 1,2,3,4,5,6,7 in dieser Reihenfolge in einen initial leeren Fibonacci-Heap ein.
- Führen Sie anschließend `EXTRACTMIN()` und danach `DECREASEKEY(6, 1)` aus.
- Geben Sie nach jedem Schritt den resultierenden Fibonacci-Heap an.

- Einleitung
- Entwurfsmethoden
- Sortieren
- Auswahlproblem
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- *Suchbäume*
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

- *Motivation*
- AVL-Bäume
- Rot-Schwarz-Bäume
- B-Bäume
- Splay-Bäume
- Tries

Binäre Suchbäume sind links-rechts-geordnete binäre Bäume. Wir implementieren die Knoten eines Binärbaums wie folgt:

- `parent` ist ein Verweis auf den Eltern-Knoten.
- `left` zeigt auf das linke Kind, `right` auf das rechte.
- Der Wert `key` enthält den Schlüssel, der in der Regel eine natürliche Zahl ist.
- `value` speichert den eigentlichen Datensatz.

```
template <typename T>
struct Knoten {
    int key;
    T value;
    Knoten *left;
    Knoten *right;
    Knoten *parent;
};
```

Suchbaum-Eigenschaft:

- Für jeden Knoten y im linken Teilbaum von Knoten x gilt: $y.key \leq x.key$
- Für jeden Knoten y im rechten Teilbaum von Knoten x gilt: $y.key > x.key$

Suchbäume unterstützen folgende Operationen:

- Einfügen (Insert)
- Entfernen (Delete)
- Suchen (Search)
- Minimum/Maximum-Suche
- Vorgänger (Predecessor), Nachfolger (Successor)

Laufzeit aller Operationen ist proportional zur Höhe des Baums.

Aufgrund der Suchbaum-Eigenschaft können die Werte des Baums nach Schlüsseln sortiert ausgegeben werden. Dazu wird der Baum in In-Order-Reihenfolge durchlaufen.

```
inorderTreeWalk(Knoten x)
    if  $x \neq \text{NIL}$ 
        inorderTreeWalk(x.left)
        report(x.value)
        inorderTreeWalk(x.right)
```

Alternativen Methoden: Post-Order und Pre-Order.

In-Order-Durchlauf:

```
string Suchbaum<T>::inorder() {  
    if (root == nullptr)  
        return "--- leer ---";  
    return root->toString();  
}
```

```
string Knoten<T>::toString() {  
    ostringstream os;  
  
    if (left != nullptr)  
        os << left->toString() << ",";  
    os << value.toString();  
    if (right != nullptr)  
        os << "," << right->toString();  
    return os.str();  
}
```

Gegeben: ein Suchbaum T und ein Schlüssel k

Gesucht: ein Knoten x mit Schlüssel k

```
treeSearch(Knoten x, int k)
    if ( $x = \text{NIL}$ )  $\vee$  ( $x.\text{key} = k$ )
        return x
    if  $k < x.\text{key}$ 
        return treeSearch(x.left, k)
    return treeSearch(x.right, k)
```

Falls kein Knoten mit Schlüssel k enthalten ist, soll das Ergebnis der Suche NIL sein.
Gibt es mehrere Knoten mit Schlüssel k , so wird einer dieser Knoten berichtet.

```
Knoten<T> * Suchbaum<T>::find(int key) {  
    if (root == nullptr)  
        return nullptr;  
    return root->find(key);  
}  
  
Knoten<T> * Knoten<T>::find(int key) {  
    if (this->key == key)  
        return this;  
    if (key < this->key) {  
        if (left == nullptr)  
            return nullptr;  
        return left->find(key);  
    } else {  
        if (right == nullptr)  
            return nullptr;  
        return right->find(key);  
    }  
}
```

Das Minimum befindet sich links unten im Baum:

```
Knoten<T> * Suchbaum<T>::minimum() {  
    if (root == nullptr)  
        return nullptr;  
    return minimum(root);  
}
```

```
Knoten<T> * Suchbaum<T>::minimum(Knoten<T> *t) {  
    while (t->left != nullptr)  
        t = t->left;  
    return t;  
}
```

Das Maximum befindet sich rechts unten im Baum:

```
Knoten<T> * Suchbaum<T>::maximum() {  
    if (root == nullptr)  
        return nullptr;  
    return maximum(root);  
}
```

```
Knoten<T> * Suchbaum<T>::maximum(Knoten<T> *t) {  
    while (t->right != nullptr)  
        t = t->right;  
    return t;  
}
```

Direkten Vorgänger bestimmen:

- *Gegeben:* ein Suchbaum T und ein Schlüssel k
- *Gesucht:* der Knoten mit nächst kleinerem Schlüssel

Wir gehen hier davon aus, dass kein Schlüssel mehrfach vorkommt.

Algorithmus:

- Zunächst müssen wir einen Knoten x mit Schlüssel k mittels `find()` finden.
- Wenn der gefundene Knoten x einen linken Teilbaum besitzt, ist der Vorgänger der maximale Wert des linken Teilbaums.
- Sonst muss der Weg zur Wurzel durchlaufen werden, bis der erste Schlüssel gefunden wird, der kleiner als k ist.

Wird kein solcher Schlüssel gefunden, dann ist k der kleinste Schlüssel im Baum und es gibt keinen Vorgänger.

direkten Vorgänger bestimmen

```
Knoten<T> * Suchbaum<T>::pred(int key) {  
    Knoten<T> *n = find(key);  
  
    if (n == nullptr)  
        return nullptr;  
  
    if (n->left != nullptr)  
        return maximum(n->left);  
  
    n = n->parent;  
    while (n != nullptr) {  
        if (n->key < key)  
            return n;  
        n = n->parent;  
    }  
    return nullptr;  
}
```

direkten Nachfolger bestimmen

```
Knoten<T> * Suchbaum<T>::succ(int key) {  
    Knoten<T> *n = find(key);  
  
    if (n == nullptr)  
        return nullptr;  
  
    if (n->right != nullptr)  
        return minimum(n->right);  
  
    n = n->parent;  
    while (n != nullptr) {  
        if (n->key > key)  
            return n;  
        n = n->parent;  
    }  
    return nullptr;  
}
```


Beim Einfügen eines Wertes `val` mit Schlüssel `key` in den Baum muss die Suchbaum-Eigenschaft erhalten bleiben. Wir gehen daher wie folgt rekursiv vor:

- Ist der Teilbaum leer, so wird ein neuer Knoten mit Wert `val` und Schlüssel `key` angelegt und als Wurzel des Teilbaums gespeichert.
- Andernfalls muss der neue Knoten entweder im linken oder im rechten Teilbaum der Wurzel `x` eingefügt werden.
 - Falls `key` größer ist als `x.key`, füge den neuen Knoten im rechten Teilbaum `x.right` ein.
 - Sonst füge den neuen Knoten im linken Teilbaum `x.left` ein.

Einfügen eines Elements

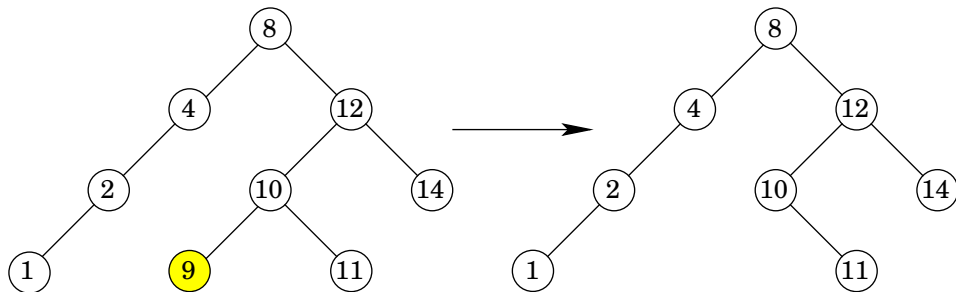
```
void Suchbaum<T>::insert(int key, T val) {
    if (root == nullptr)
        root = new Knoten<T>(key, val);
    else root->insert(key, val);
}

void Knoten<T>::insert(int key, T val) {
    if (key <= this->key) {
        if (left != nullptr)
            left->insert(key, val);
        else left = new Knoten(key, val, this);
    } else {
        if (right != nullptr)
            right->insert(key, val);
        else right = new Knoten(key, val, this);
    }
}
```

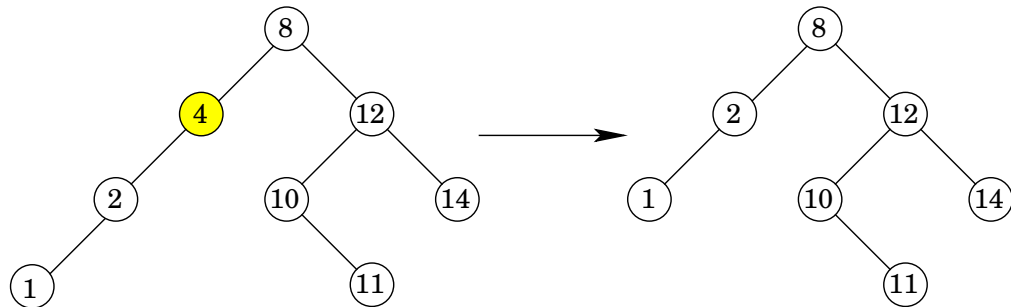
Wir unterscheiden drei Fälle: Wenn der zu entfernende Knoten

- ein Blatt ist, dann kann das Blatt einfach gelöscht werden.
Achtung: Beim Elternteil `e` das Löschen des entsprechenden Nachfolgers `e.left` bzw. `e.right` nicht vergessen!
- nur ein Kind besitzt, dann kann der Knoten durch dieses Kind ersetzt werden.
Achtung: Anpassen des `parent`-Eintrags beim Kind nicht vergessen!
- zwei Kinder besitzt, suchen wir zunächst den direkten Nachfolger des Knotens. Der direkte Nachfolger besitzt kein linkes Kind. Wir ersetzen den Knoten durch seinen direkten Nachfolger und entfernen den direkten Nachfolger wie oben aus dem Baum.

Fall 1: Der zu löschende Knoten ist ein Blatt

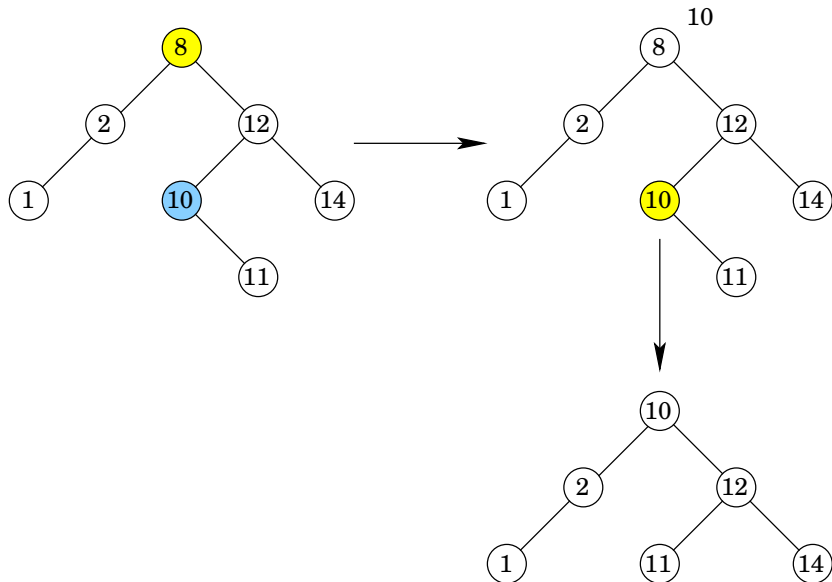


Fall 2: Der zu löschende Knoten hat nur ein Kind.



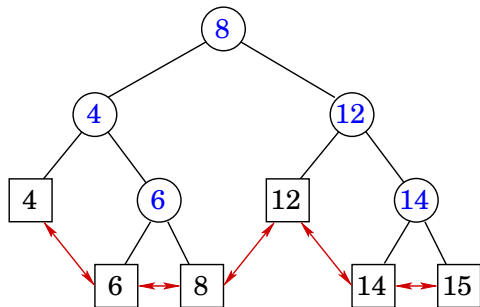
Löschen eines Elements

Fall 3: Der zu löschende Knoten hat zwei Kinder.



Blattsuchbäume: Die Werte werden nur in den Blättern des Baums gespeichert. Innere Knoten enthalten nur „Wegweiser“ für die Suche, und zwar den maximalen Wert des linken Teilbaums.

Zusätzlich können alle Blätter doppelt verkettet werden. Dadurch können Bereichsanfragen der Form „Welche Werte liegen zwischen k_1 und k_2 ?“ schnell beantwortet werden.

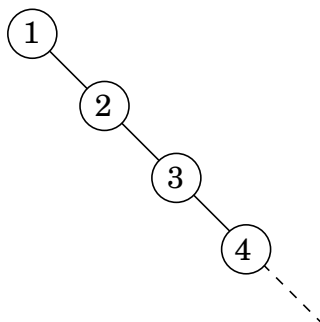


Beispiel: $k_1 = 5, k_2 = 13$

- Suche den Wert k_1 : Die Suche endet im Knoten mit dem Wert 6.
- Laufe die verkettete Liste entlang, bis ein Wert $\geq k_2$ erreicht wird, und gib alle Werte auf diesem Weg aus.

- Implementieren Sie einen Suchbaum mit folgenden Methoden:
 - `insert(T val)` fügt den Wert `val` in den Suchbaum ein.
 - `bool contains(T val)` sucht den Wert `val` im Suchbaum.
 - `string toString()` liefert eine sortierte Ausgabe des Baums.
- Schreiben Sie ein Programm zum automatischen Testen des Suchbaums.
- Fügen Sie nacheinander $2^{15}, 2^{16}, 2^{17}, \dots, 2^{25}$ zufällige Werte in den Suchbaum ein.
- Ändert sich das Laufzeitverhalten, wenn die eingefügten Werte aufsteigend sortiert sind?

Durch das Einfügen von aufsteigend sortierten Werten entsteht ein zu einer Liste degenerierter Baum.



Beim Einfügen eines Wertes muss also immer bis ans Ende der Liste gelaufen werden. Als Laufzeit erhalten wir daher:

$$T(n) = \sum_{i=1}^n i \in \Theta(n^2)$$

Das Einfügen von zufälligen Werten geht deutlich schneller. Beim erfolgreichen Suchen eines Wertes werden so viele Vergleiche benötigt, wie der gesuchte Knoten von der Wurzel entfernt ist.

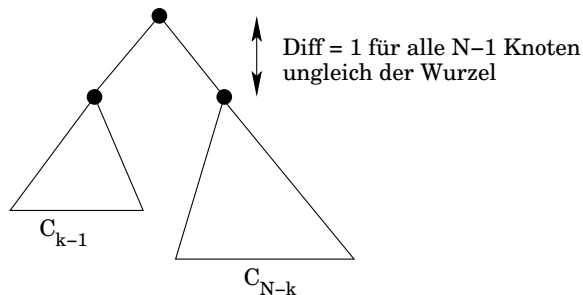
Die interne Pfadlänge (internal path length) eines Baums ist:

$$\sum_v \text{Distanz von Knoten } v \text{ zur Wurzel}$$

Wenn wir die interne Pfadlänge durch die Anzahl der Knoten N im Baum dividieren, so erhalten wir die durchschnittliche Anzahl der benötigten Vergleiche.

Sei C_N die durchschnittliche interne Pfadlänge eines binären Suchbaums mit N Knoten. Dann gilt:

$$\begin{aligned} C_1 &= 1 \\ C_N &= N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}) \end{aligned}$$



Analog zu der Rekursionsformel von Quicksort gilt:

$$C_N \in \Theta(N \log N)$$

Dividieren wir durch N so erhalten wir die durchschnittliche Anzahl Vergleiche bei einer erfolgreichen Suche: $\Theta(\log N)$

Die Analyse einer erfolglosen Suche ist komplizierter, liefert aber dieselbe asymptotische Laufzeitabschätzung.

- Motivation
- *AVL-Bäume*
- Rot-Schwarz-Bäume
- B-Bäume
- Splay-Bäume
- Tries

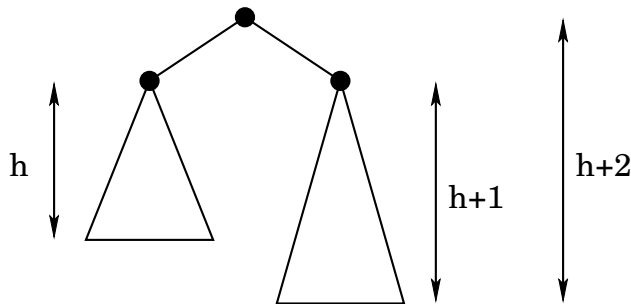
Ziel: Wir wollen auch im worst-case eine logarithmisch beschränkte Laufzeit für das Suchen, Einfügen und Löschen von Elementen. Die Höhe des Suchbaums muss also auf $\mathcal{O}(\log n)$ beschränkt werden.

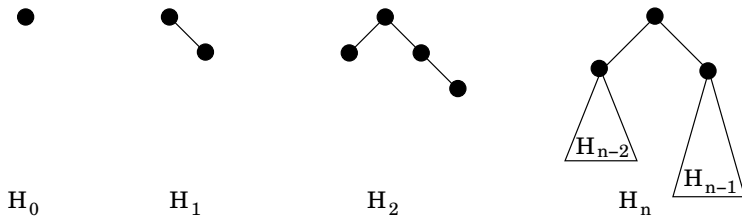
Ein Suchbaum ist *höhenbalanciert*, wenn für jeden inneren Knoten u gilt, dass sich die Höhe des rechten und linken Teilbaumes von u höchstens um 1 unterscheidet. Solche Bäume werden auch AVL-Bäume genannt, nach deren Entwicklern Adelson-Velskij und Landis.

Der Balance-Wert ist definiert als Höhe des rechten Teilbaums minus der Höhe des linken Teilbaums. Solange der Balance-Wert an jedem inneren Knoten eines Baums gleich -1 , 0 oder $+1$ ist, ist der Baum ein AVL-Baum.

Die Höhenbedingung stellt sicher, dass AVL-Bäume mit n Knoten eine Höhe aus $\mathcal{O}(\log n)$ haben:

- Höhe $h = 0$: minimale Knotenanzahl ist 1.
- Höhe $h = 1$: minimale Knotenanzahl ist 2.
- Einen AVL-Baum der Höhe $h + 2$ mit minimaler Knotenanzahl erhält man, wenn man je einen AVL-Baum mit Höhe $h + 1$ und Höhe h mit minimaler Knotenanzahl wie folgt zu einem Baum der Höhe $h + 2$ zusammenfügt:





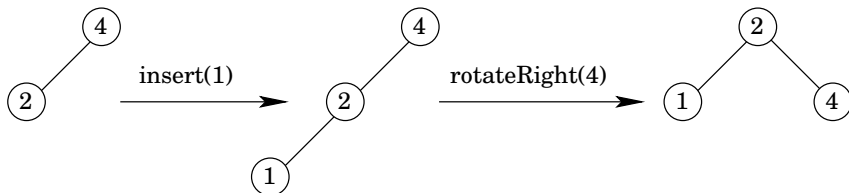
Sei $k(H_i)$ die Anzahl der Knoten in Baum H_i und sei F_i die i -te Fibonacci-Zahl. Dann gilt:

$$\begin{aligned}k(H_0) &= 1 \geq F_2 \\k(H_1) &= 2 \geq F_3 \\k(H_n) &= 1 + k(H_{n-2}) + k(H_{n-1}) \geq F_{n+2}\end{aligned}$$

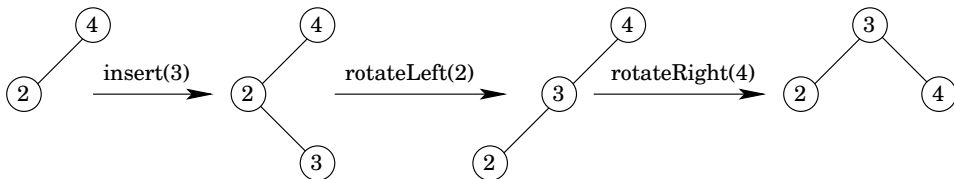
Ein AVL-Baum der Höhe h hat also wenigstens F_{h+2} Knoten. Aus der Mathematik wissen wir: $F_n \approx 1.618^n$

Die Anzahl der Knoten in einem höhenbalancierten Baum wächst also exponentiell mit der Höhe. \rightarrow Ein AVL-Baum mit n Knoten hat eine Höhe aus $\mathcal{O}(\log n)$.

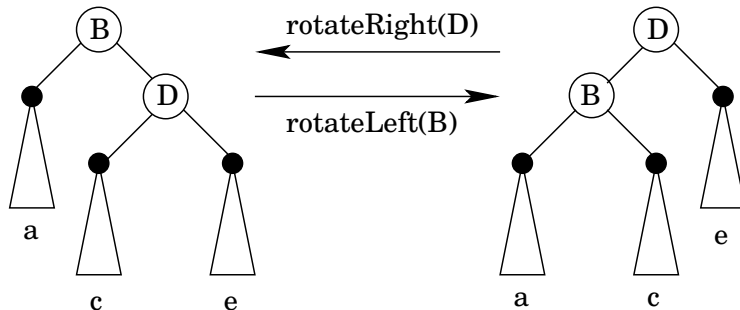
Beim Einfügen eines Wertes kann die Balance zerstört werden. Wir gleichen das durch sogenannte Rotationen aus.



Je nachdem, ob der neue Knoten als linker oder rechter Nachfolger eingefügt wird, ist zuvor noch eine gegensätzliche Rotation notwendig:



Rotationen können in konstanter Zeit ausgeführt werden, da nur einige Zeiger „verbogen“ werden müssen.



Wie werden nach einer Links-Rotation die Balance-Werte neu berechnet? $nBal$ steht für new Balance, $oBal$ für old Balance.

$$nBal(B) = ht(c) - ht(a)$$

$$oBal(B) = ht(D) - ht(a) = (1 + \max\{ht(c), ht(e)\}) - ht(a)$$

Wir subtrahieren die zweite Gleichung von der ersten:

$$nBal(B) - oBal(B) = ht(c) - (1 + \max\{ht(c), ht(e)\}) + ht(a) - ht(a)$$

Die beiden letzten Terme heben sich auf, außerdem addieren wir $oBal(B)$ auf beiden Seiten und stellen ein wenig um:

$$nBal(B) = oBal(B) - 1 - (\max\{ht(c), ht(e)\} - ht(c))$$

Da $\max(x, y) - z = \max(x - z, y - z)$ gilt, erhalten wir:

$$nBal(B) = oBal(B) - 1 - \max\{ht(c) - ht(c), ht(e) - ht(c)\}$$

Da $ht(e) - ht(c) = oBal(D)$ ist, erhalten wir also:

$$nBal(B) = oBal(B) - 1 - \max\{0, oBal(D)\}$$

Durch eine entsprechende Rechnung zeigt man:

$$nBal(D) = oBal(D) - 1 + \min\{nBal(B), 0\}$$

Für die Balance-Werte nach einer Rechts-Rotation gilt:

$$\begin{aligned}nBal(B) &= oBal(B) + 1 - \min(oBal(D), 0) \\nBal(D) &= oBal(D) + 1 + \max(nBal(B), 0)\end{aligned}$$

Durch die Rotationen können sich die Balance-Werte bei den Vorgänger-Knoten ändern. Daher müssen evtl. alle Knoten bis hinauf zur Wurzel angepasst werden. Dies wird einfach beim Abbau der Rekursion erledigt.

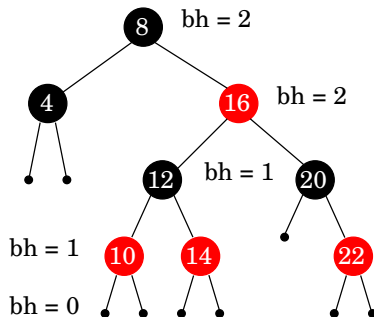
```
bool AvlTree<T>::insert(Node<T> *node, T val) {  
    bool deeper = false;  
  
    if (val < node->value) {  
        if (node->left == nullptr) {  
            // if node has no left child then create new node  
            node->left = new Node<T>(val, node);  
            if (node->right != nullptr) {  
                node->balance = 0;  
                deeper = false;  
            } else {  
                node->balance = -1;  
                deeper = true;  
            }  
        } else {  
            // insert val in left subtree recursively  
            ....  
        }  
    }  
}
```

```
deeper = insert(node->left, val);
if (deeper) {
    node->balance -= 1;
    if (node->balance == 0)
        deeper = false;
    if (node->balance == -2) {
        if (node->left->balance == -1) {
            rotateRight(node);
        } else {
            rotateLeft(node->left);
            rotateRight(node);
        }
        deeper = false;
    }
}
}
}
} else ... // val > node->value similarly
return deeper;
```

- Motivation
- AVL-Bäume
- *Rot-Schwarz-Bäume*
- B-Bäume
- Splay-Bäume
- Tries

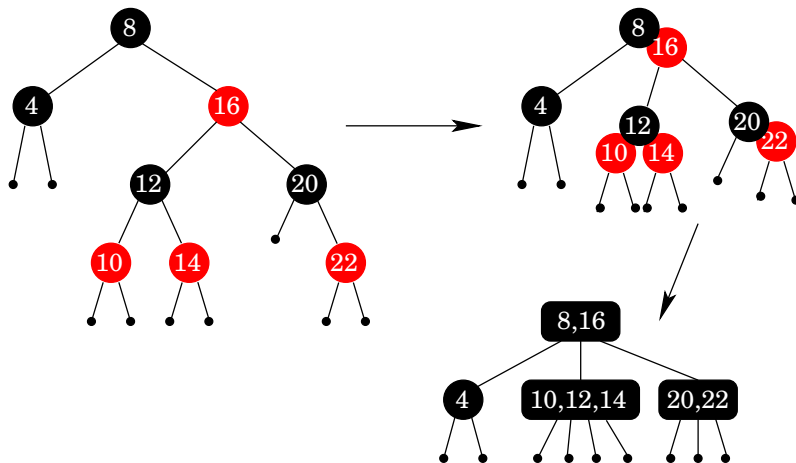
Balancierte Binärbäume, die im Gegensatz zu AVL-Bäumen nur ein zusätzliches Bit pro Knoten als Zusatzinformation benötigen:

- Jeder Knoten ist entweder rot oder schwarz.
- Die Wurzel und die Blätter (NIL's) sind schwarz.
- Wenn ein Knoten rot ist, dann ist sein Vorgänger schwarz.
- Jeder einfache, abwärtsgerichtete Weg von einem Knoten x zu einem Blatt hat die gleiche Anzahl schwarzer Knoten: $bh(x)$.



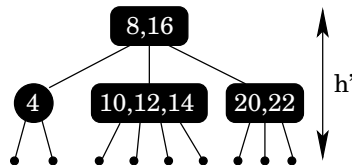
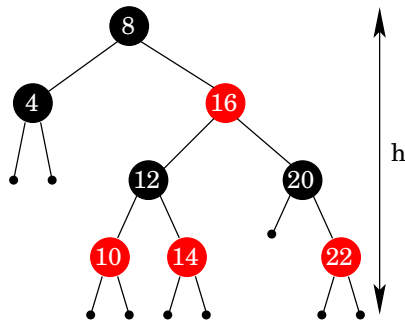
Satz: Ein Rot-Schwarz-Baum mit n inneren Knoten hat höchstens eine Höhe von $h \leq 2 \cdot \log_2(n + 1)$.

Idee: Verschmelze die roten Knoten mit deren schwarzen Eltern.



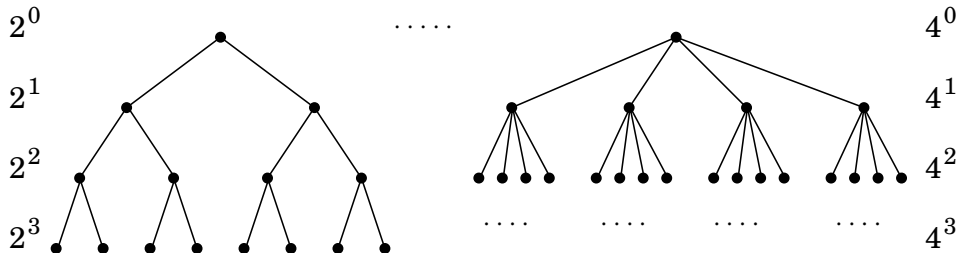
Dadurch erhalten wir einen balancierten Baum, dessen Knoten 2, 3 oder 4 Kinder haben. Dieser so entstandene Baum habe Höhe h' .

Dann gilt $h' \geq h/2$, da höchstens jeder zweite Knoten auf einem Weg von der Wurzel zu einem Blatt rot ist.



Ein Rot-Schwarz-Baum mit n inneren Knoten hat genau $n + 1$ viele Blätter. Der entstandene 2-3-4-Baum hat daher auch $n + 1$ Blätter.

Wie man in folgender Abbildung sieht, hat ein 2-3-4-Baum der Höhe t mindestens $2^t - 1$ und höchstens $\frac{4^t - 1}{4 - 1}$ innere Knoten.



In unserem Fall gilt also $2^{h'} - 1 \leq n \leq \frac{4^{h'} - 1}{4 - 1}$.

$$n \geq 2^{h'} - 1 \equiv n + 1 \geq 2^{h'} \equiv \log_2(n + 1) \geq h'$$

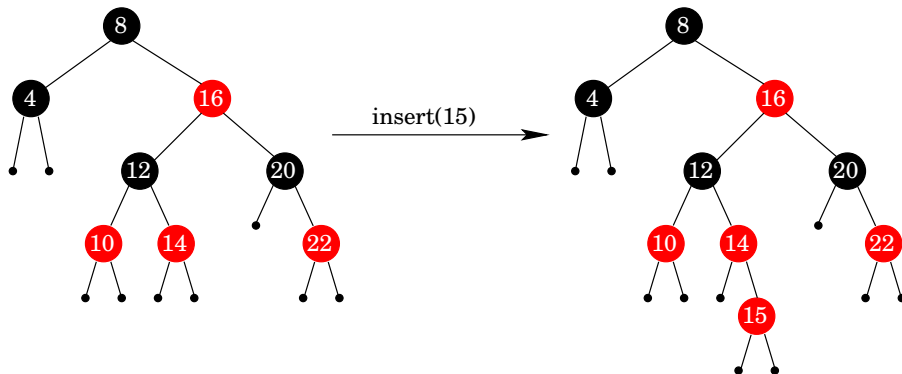
Mit $h' \geq h/2$ gilt also: $\log_2(n + 1) \geq h' \geq h/2$.

Damit erhalten wir $2 \cdot \log_2(n + 1) \geq h$.

Einfügen: (Idee)

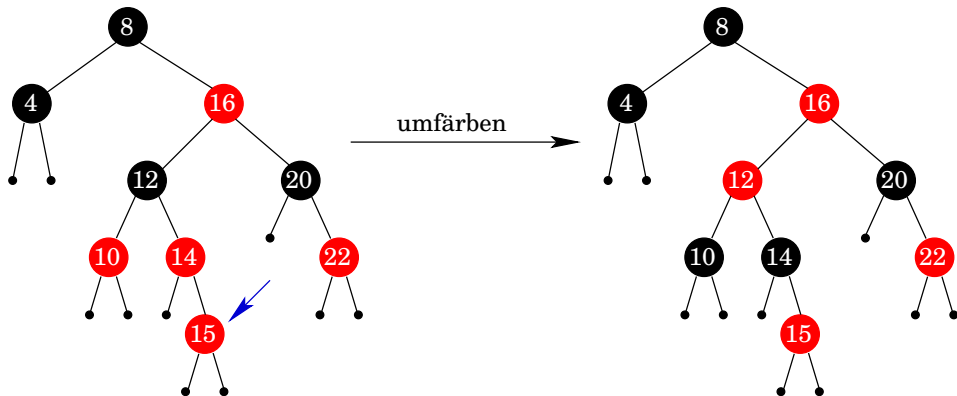
- Füge den Wert x in den Baum ein.
- Färbe den neuen Knoten rot.

→ Nur die Rot-Schwarz-Eigenschaft kann verletzt sein!



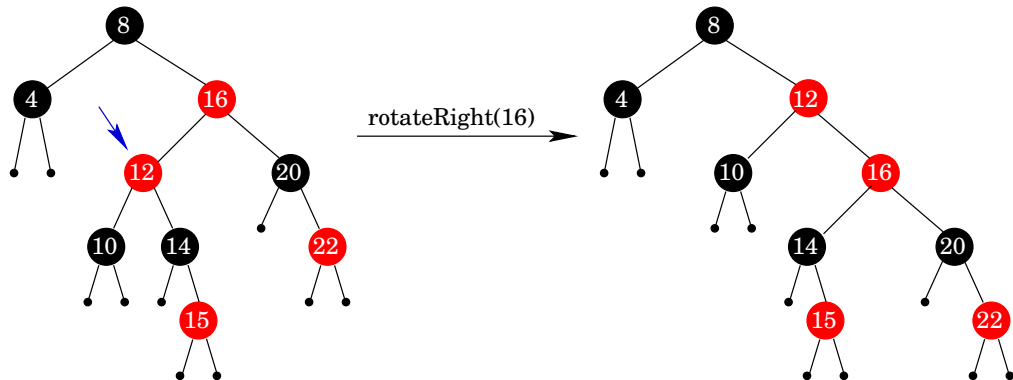
Versuche durch umfärben von Knoten und rotieren von Teilbäumen diese Verletzung hinauf zur Wurzel zu schieben.

Wenn der Vorgänger und dessen Geschwister beide rot gefärbt sind, dann färbe beide Knoten schwarz und färbe den Vor-Vorgänger rot.

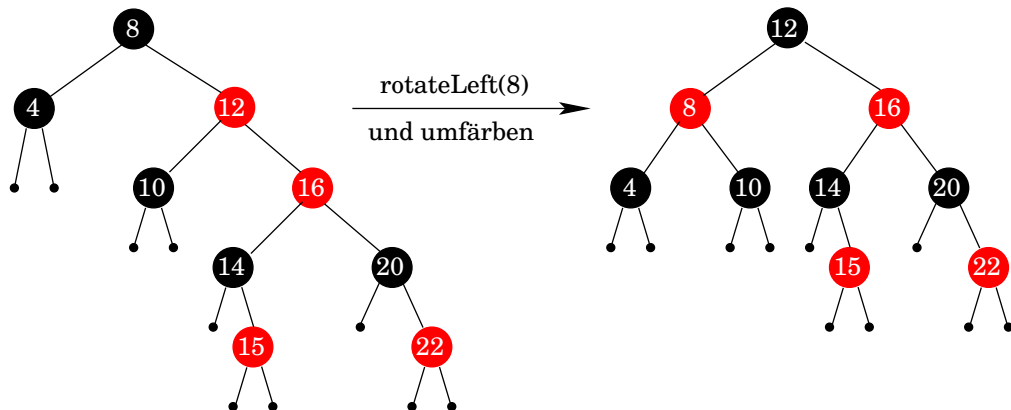


Dadurch ist die Verletzung der Rot-Schwarz-Bedingung weiter zur Wurzel gewandert und das Korrigieren ist bei dem Knoten mit Wert 12 fortzusetzen.

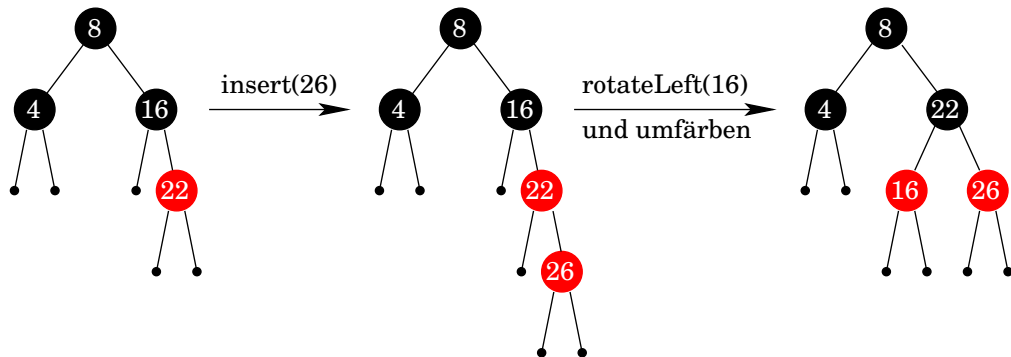
Nun benötigen wir eine andere Strategie: Wir wollen den Baum so rotieren, dass die schwarzen Knoten mit den Werten 4, 10, 14 und 20 auf einer Ebene im Baum liegen. Dazu führen wir zunächst eine Rechtsrotation bei 16 aus, anschließend eine Linksrotation bei 8.



Nun die Linksrotation bei 8, anschließend noch die Knoten mit den Werten 8 und 12 umfärben: fertig!



Eine solche Doppelrotation ist nicht immer notwendig. Im folgenden Beispiel wird nur eine einfache Rotation durchgeführt.



Wäre der neue Knoten linker Nachfolger seines Vorgängers, z.B. wenn wir den Wert 20 eingefügt hätten, dann hätte zunächst eine Rechtsrotation bei 22 stattfinden müssen, bevor die Linksrotation bei 16 durchgeführt wird.

Implementierung:

Wenn noch kein Knoten gespeichert ist, wird ein neuer, schwarz gefärbter Knoten als Wurzel eingefügt; der Knoten hat keinen Vorgänger.

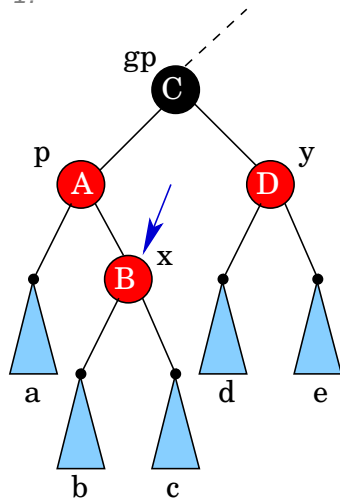
```
void RedBlackTree<T>::insert(T val) {  
    if (root == nullptr) {  
        root = new Node<T>(val);  
        root->col = BLACK;  
        return;  
    }  
    ....  
}
```

Ansonsten fügen wir mittels der schon bekannten Methode `insert` einen neuen, rot gefärbten Knoten in den Baum ein und müssen anschließend die Rot-Schwarz-Eigenschaft wieder herstellen.

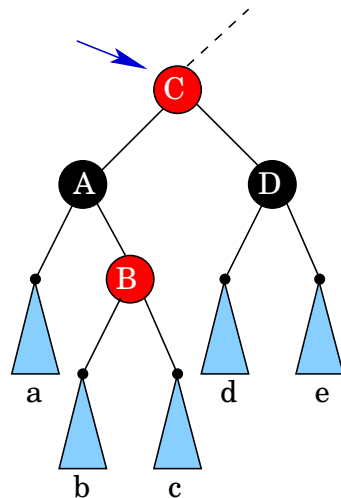

```
Node<T> *x = insert(root, val);
while (x != root && x->parent->col == RED) {
    Node<T> *p = x->parent;    // parent of x
    Node<T> *gp = p->parent;  // grandparent

    if (p == gp->left) {
        Node<T> *y = gp->right; // sibling of parent
        if (y != nullptr && y->col == RED) {
            // Fall 1
        } else {
            if (x == p->right)
                // Fall 2+3
            else // Fall 3
        }
    } else // analog, left/right vertauscht
}
root->col = BLACK;
}
```

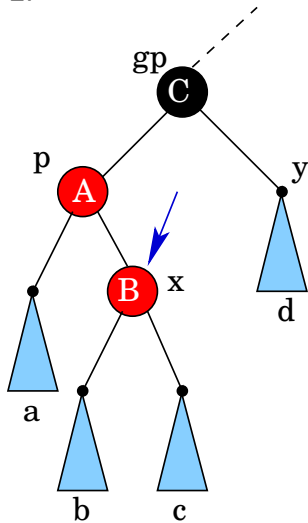
Fall 1:



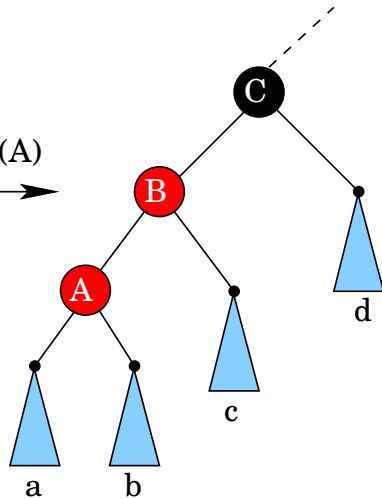
umfärben
→



Fall 2:

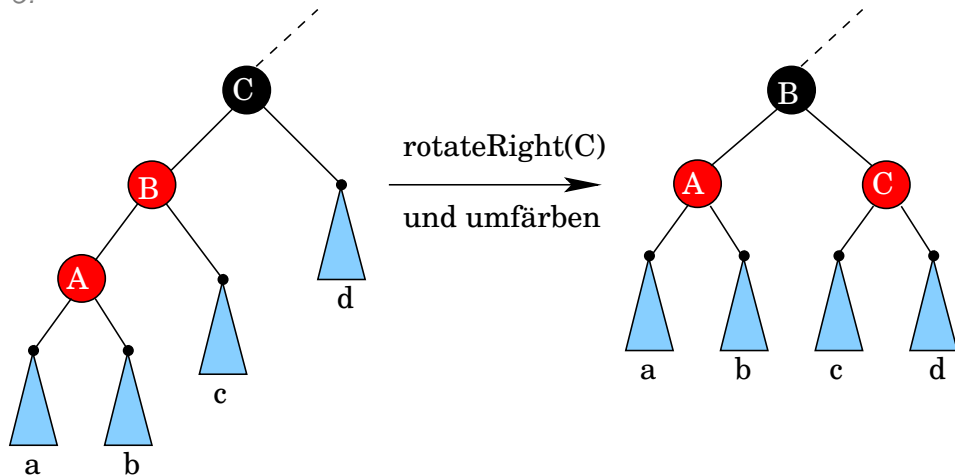


rotateLeft(A)



Anschließend direkt weiter mit Fall 3.

Fall 3:



Anschließend ist die Transformation abgeschlossen. Es können keine weiteren Verletzungen der Rot-Schwarz-Eigenschaft vorhanden sein.

Übung:

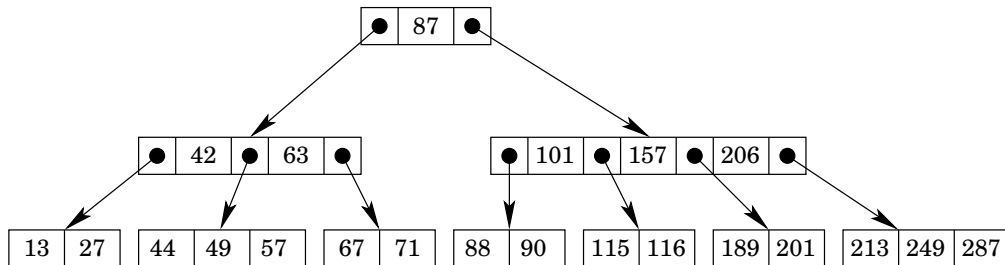
- Implementieren Sie einen AVL- und einen Rot-Schwarz-Baum, die jeweils die Methoden `insert`, `contains` und `toString` unterstützen.
- Schreiben Sie jeweils ein Programm zum automatischen Testen des entsprechenden Suchbaums.
- Fügen Sie nacheinander $2^{15}, 2^{16}, 2^{17}, \dots, 2^{25}$ zufällige Werte in die Suchbäume ein und messen Sie die Laufzeiten.

Vergleichen Sie die Laufzeiten auch mit den Laufzeiten Ihrer Implementierung eines einfachen, nicht-balancierten Suchbaums.

- Wie ändert sich das Laufzeitverhalten, wenn die eingefügten Werte aufsteigend sortiert sind?

- Motivation
- AVL-Bäume
- Rot-Schwarz-Bäume
- *B-Bäume*
- Splay-Bäume
- Tries

Wir beschränken uns nicht länger auf binäre Suchbäume. In dem Beweis zur Tiefe von Rot-Schwarz-Bäumen haben wir bereits einen B-Baum der Ordnung 4 kennengelernt: den 2-3-4-Baum.



Jeder innere Knoten eines 2-3-4-Baums hat mindestens 2 und höchstens 4 Kinder.

Der B-Baum wurde 1972 von Rudolf Bayer und Edward M. McCreight entwickelt.
(Quelle: wikipedia)

Ein B-Baum der Ordnung m hat folgende Eigenschaften:

- Alle Blätter befinden sich in gleicher Tiefe.
- Alle inneren Knoten außer der Wurzel haben mindestens $\lceil m/2 \rceil$ Kinder. In einem nicht leeren Baum hat die Wurzel mindestens 2 Kinder.
- Jeder innere Knoten hat höchstens m Kinder. Ein Knoten mit k Kindern speichert $k - 1$ Schlüsselwerte.
- Alle Schlüsselwerte eines Knotens sind aufsteigend sortiert.
- Seien k_1, \dots, k_s die Schlüssel eines inneren Knotens. Dann gibt es die Zeiger z_0, z_1, \dots, z_s auf die Kinder und es gilt:
 - z_0 zeigt auf einen Teilbaum mit Werten kleiner als k_1 .
 - z_i für $i = 1, \dots, s - 1$ zeigt auf einen Teilbaum mit Werten größer als k_i und kleiner als k_{i+1} .
 - z_s zeigt auf einen Teilbaum mit Werten größer als k_s .

Suchen eines Elements:

Die Suche nach einem Schlüssel k in einem B-Baum ist eine natürliche Verallgemeinerung der Suche in einem Suchbaum:

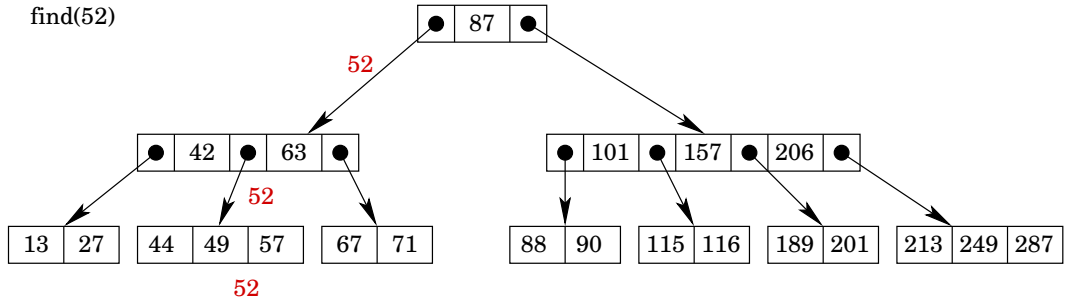
- Man beginnt mit der Wurzel, die die Schlüssel k_1, \dots, k_l speichert, und bestimmt den kleinsten Index i mit $x \leq k_i$, falls es ein solches i gibt. Ansonsten ist x größer als der größte Schlüssel k_l .
 - Ist $x = k_i$, dann wurde x gefunden.
 - Ist $x < k_i$, so wird die Suche bei z_{i-1} fortgesetzt.
 - Ist $x > k_i$, dann wird die Suche bei z_i fortgesetzt.

Dies wird solange wiederholt, bis man x gefunden hat, oder an einem Blatt erfolglos endet.

- Die Suche nach dem kleinsten Index i mit $x \leq k_i$ kann mit linearer Suche erfolgen.

B-Bäume

find(52)

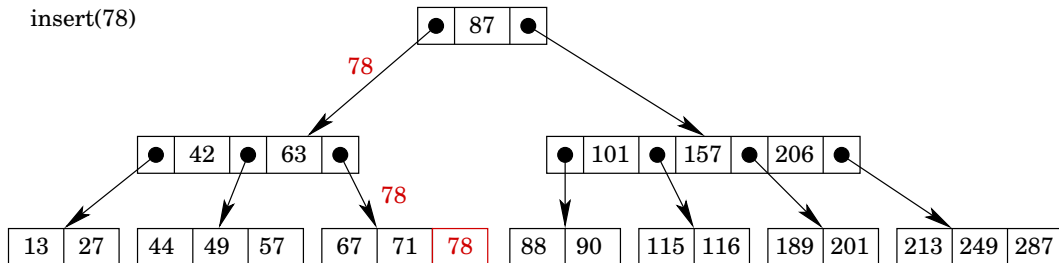


Einfügen eines Wertes:

Bestimme zunächst mittels der Suche das Blatt, in dem der Wert abgelegt werden muss. Wir unterscheiden zwei Fälle:

- Fall 1: Das Blatt hat noch nicht die maximale Anzahl $m - 1$ von Schlüsseln gespeichert.
Dann fügen wir den Wert entsprechend der Sortierung ein.

insert(78)



- Fall 2: Das Blatt hat bereits die maximale Anzahl $m - 1$ von Schlüsseln gespeichert.

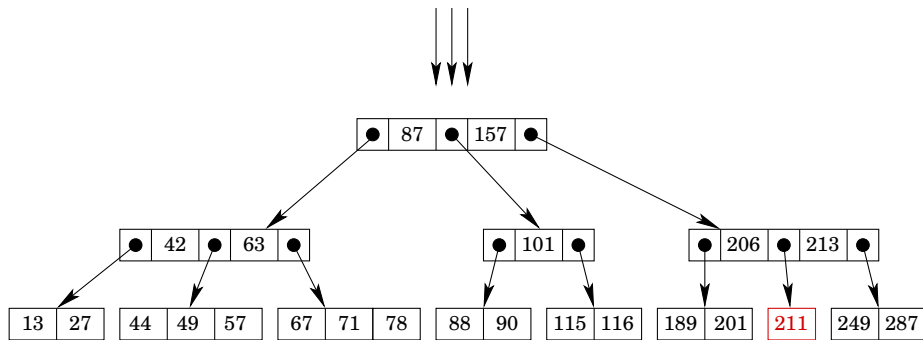
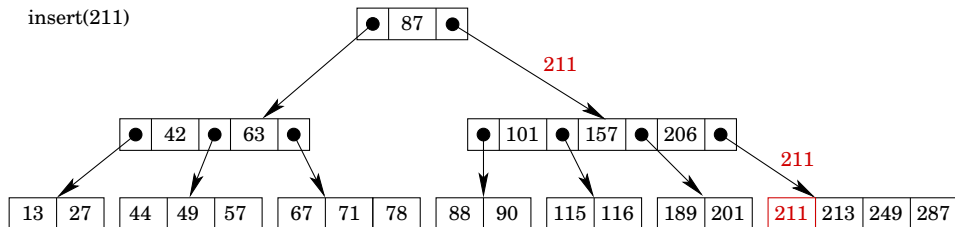
In diesem Fall ordnen wir den Wert wie im Fall 1 entsprechend seiner Größe ein und teilen anschließend den zu groß gewordenen Knoten in der Mitte auf. Das mittlere Element wird in den Vorgänger-Knoten eingefügt.

Dieses Teilen wird solange längs des Suchpfades bis zur Wurzel fortgesetzt, bis ein Knoten erreicht ist, der noch nicht die maximale Anzahl von Schlüsseln speichert, oder bis die Wurzel erreicht wird.

Muss die Wurzel geteilt werden, so schafft man eine neue Wurzel, die den mittleren Schlüssel als einzigen Schlüssel speichert.

B-Bäume

insert(211)



Ein B-Baum der Ordnung m mit Höhe h hat die minimale Blattanzahl, wenn seine Wurzel nur 2 und jeder innere Knoten nur $\lceil m/2 \rceil$ viele Kinder hat. Daher ist die minimale Blattanzahl

$$N_{\min} = 2 \cdot \lceil m/2 \rceil^{h-1}.$$

Die Blattanzahl wird maximal, wenn jeder innere Knoten die maximal mögliche Anzahl von Kindern hat. Somit ist die maximale Blattanzahl

$$N_{\max} = m^h.$$

Also gilt:

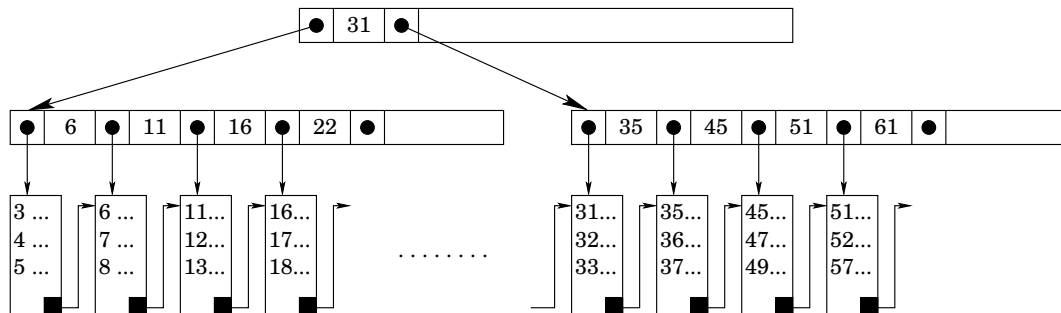
$$N_{\min} = 2 \cdot \lceil m/2 \rceil^{h-1} \leq n+1 \leq m^h = N_{\max}$$

und somit $h \leq 1 + \log_{\lceil m/2 \rceil}(\frac{n+1}{2})$ und $\log_m(n+1) \leq h$.

B-Bäume sind also balanciert.

Für den Einsatz in Datenbanksystemen werden die B-Bäume geringfügig modifiziert:

- Die Blätter werden zu einer linearen Liste verkettet.
- Werte werden nur in den Blättern gespeichert: Blattsuchbaum
- Innere Knoten enthalten nur Hinweise: kleinster Schlüsselwert im rechten Teilbaum



- Motivation
- AVL-Bäume
- Rot-Schwarz-Bäume
- B-Bäume
- *Splay-Bäume*
- Tries

Splay-Bäume im Vergleich zu AVL- und Rot-Schwarz-Bäumen:

- Es sind keine zusätzlichen Informationen wie Balance-Wert oder Farbe notwendig.
 - speicher-effizienter
 - weniger Programmieraufwand
- Strukturanpassung an unterschiedliche Zugriffshäufigkeiten:
 - Oft angefragte Schlüssel werden in Richtung Wurzel bewegt.
 - Selten angefragte Schlüssel wandern zu den Blättern hinab.
- Die Zugriffshäufigkeiten sind vorher nicht bekannt.

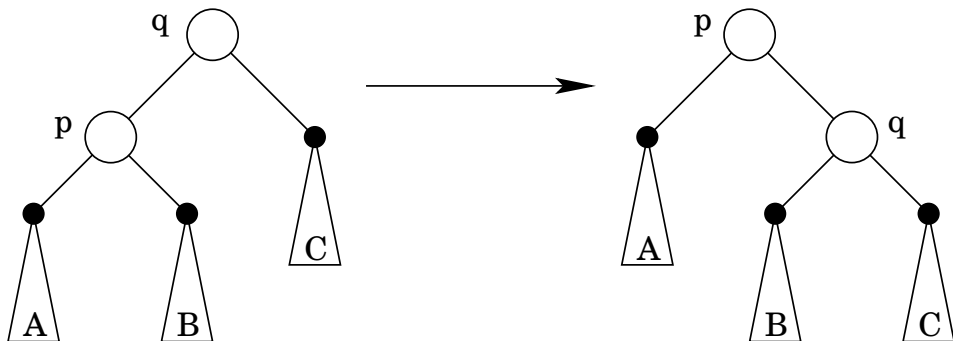
Splay-Bäume sind *selbstanordnende Suchbäume*: Jeder Schlüssel x , auf den zugegriffen wurde, wird mittels Umstrukturierungen zur Wurzel bewegt. Zugleich wird erreicht, dass sich die Längen aller Pfade zu Schlüssel x auf dem Suchpfad zu x etwa halbieren.

Die Splay-Operation: Sei T ein Suchbaum und x ein Schlüssel. Dann ist $\text{splay}(T, x)$ der Suchbaum, den man wie folgt erhält:

- Schritt 1: Suche nach x in T . Sei p der Knoten, bei dem die erfolgreiche Suche endet, falls x in T vorkommt.
Ansonsten sei p der Vorgänger des Blattes, an dem die Suche nach x endet, falls x nicht in T vorkommt.
- Schritt 2: Wiederhole die folgenden Operationen zig, zig-zig und zig-zag beginnend bei p solange, bis sie nicht mehr ausführbar sind, weil p Wurzel geworden ist.

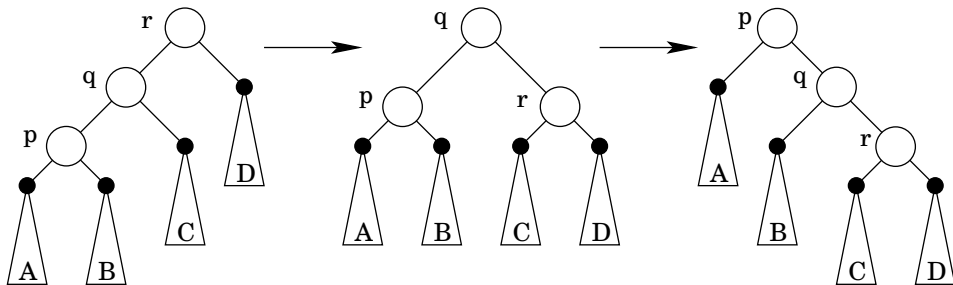
Fall 1: p hat den Vorgänger q und q ist die Wurzel.

Dann führe die Operation `zig` aus: eine Rotation nach links oder rechts, die p zur Wurzel macht.



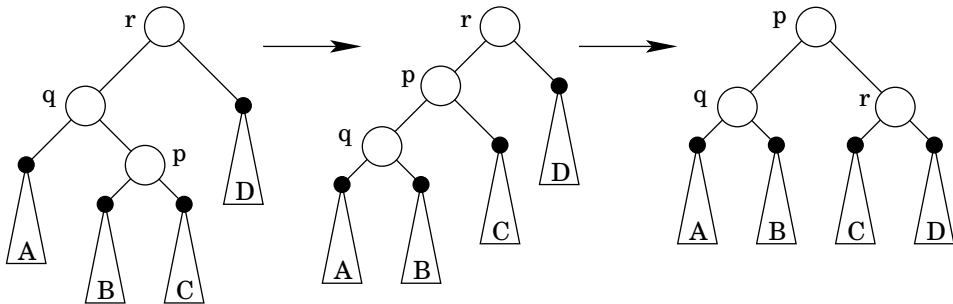
Fall 2: p hat den Vorgänger q und den Vor-Vorgänger r . Außerdem sind p und q beides rechte oder beides linke Nachfolger.

Dann führe die Operation zig-zig aus: zwei aufeinanderfolgende Rotationen in dieselbe Richtung, die p zwei Ebenen hinaufbewegen.



Fall 3: p hat Vorgänger q und Vor-Vorgänger r , wobei p linker Nachfolger von q und q rechter Nachfolger von r ist, bzw. p ist rechter Nachfolger von q und q ist linker Nachfolger von r .

Dann führe die Operation zig-zag aus: zwei Rotationen in entgegengesetzter Richtung, die p zwei Ebenen hinaufbewegen.



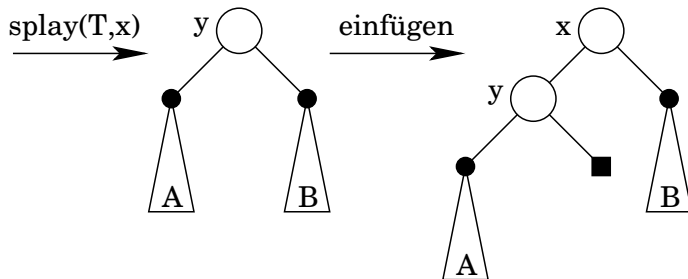
Anmerkungen:

- Kommt x in T vor, so erzeugt $\text{splay}(T, x)$ einen Suchbaum, der den Schlüssel x in der Wurzel speichert.
- Kommt x nicht in T vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel x unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

Suchen: Um nach x in T zu suchen wird $\text{splay}(T, x)$ ausgeführt und x an der Wurzel gesucht.

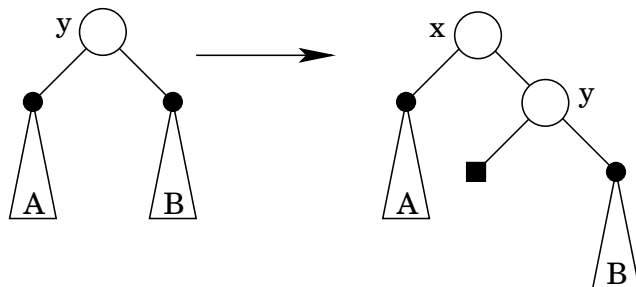
Einfügen: Um x in T einzufügen, rufe $\text{splay}(T, x)$ auf. Ist x nicht in der Wurzel, so füge wie folgt eine neue Wurzel mit x ein.

- Falls der Schlüssel der Wurzel von T kleiner als x ist:



Dieses Vorgehen ist korrekt aufgrund obiger zweiter Anmerkung: Kommt x nicht in T vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel x unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

- Falls der Schlüssel der Wurzel von T größer als x ist:



Dieses Vorgehen ist korrekt aufgrund obiger zweiter Anmerkung.

Das Ausführen einer beliebigen Folge von m Such-, Einfüge- und Entferne-Operationen, in der höchstens n -mal eingefügt wird und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $\mathcal{O}(m \cdot \log(n))$ Schritte, siehe Kapitel amortisierte Laufzeitanalyse.

Entfernen: Um x aus T zu entfernen, rufe $\text{splay}(T, x)$ auf.

Ist x in der Wurzel, dann sei L der linke und R der rechte Teilbaum unter der Wurzel.

Rufe $\text{splay}(L, \infty)$ auf. Dadurch entsteht ein Teilbaum mit dem größten Schlüssel von L an der Wurzel und einem leeren rechten Teilbaum. Ersetze den rechten leeren Teilbaum durch R .

Hinweis: Die Ausführung der Operation $\text{splay}(T, x)$ schließt immer die Suche nach x ein.

Übung:

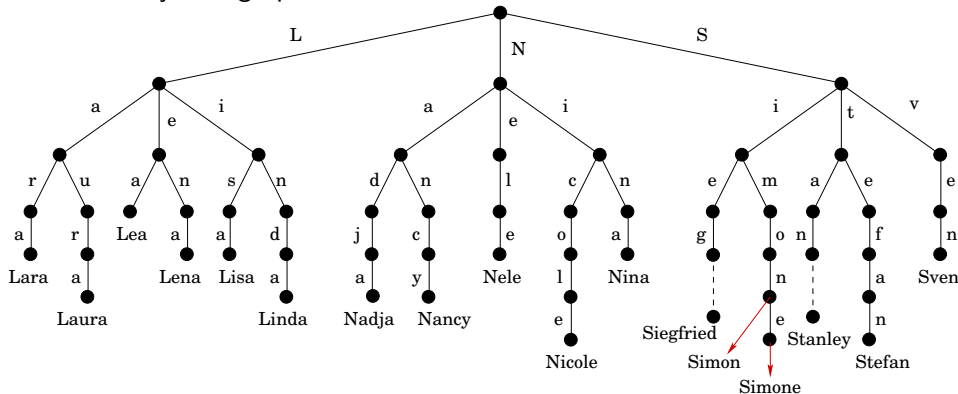
- Fügen Sie nacheinander die Schlüssel 1,2,3,4,5,6 und 7 in einen anfangs leeren Splay-Baum ein.
- Führen Sie anschließend eine Suche nach dem Schlüssel 1 aus.
- Geben Sie nach jedem Schritt den resultierenden Splay-Baum an.
- Welche Schritte sind schnell ausführbar, welche dauern länger?

- Motivation
- AVL-Bäume
- Rot-Schwarz-Bäume
- B-Bäume
- Splay-Bäume
- *Tries*

Digitale Suchbäume:

- Ein Schlüssel, der aus einzelnen Zeichen eines Alphabets besteht, wird in Teile zerlegt.
- Baum wird anhand der Schlüsselteile aufgebaut: Bits, Ziffern oder Zeichen eines Alphabets oder Zusammenfassung der Grundelemente wie Silben der Länge k .
- Eine Suche im Baum nutzt nicht die totale Ordnung der Schlüssel, sondern erfolgt durch Vergleich von Schlüsselteilen.
- Jede unterschiedliche Folge von Teilschlüsseln ergibt einen eigenen Suchweg im Baum.
- Alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg. Digitale Suchbäume werden daher auch Präfix-Bäume genannt.

Trie kommt von dem Begriff **Retrieval**. Zur Unterscheidung von „tree“ und „trie“, wird „trie“ oft wie „try“ ausgesprochen.



Suchen eines Wertes $w = w_1 w_2 \dots w_n$ in einem Trie:

- Verzweige in Ebene i anhand des Zeichens w_i .
- Wert ist nicht gespeichert, falls Verzweigung nicht existiert.

Problem: Wörter, die Präfix eines anderen Wortes sind, wie Andrea/Andreas, Simon/Simone, Stefan/Stefanie usw.

- Jeder Knoten hat einen `value`- und einen `children`-Eintrag.
- `value` speichert einen Zeiger auf den zum Schlüssel gehörenden Eintrag.
- `children` speichert die Zeiger auf Nachfolger.

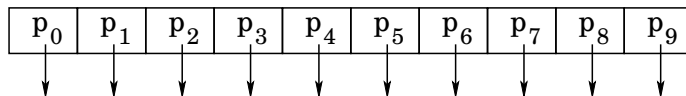
Einfügen eines Wertes $w = w_1 w_2 \dots w_n$ in einen Trie:

- Beginne an der Wurzel.
- In Ebene i betrachte das Zeichen w_i :
 - Falls kein Kind mit Namen w_i vorhanden ist: Erzeuge ein Kind mit Namen w_i .
 - Gehe zu Kind w_i und fahre fort mit Ebene $i + 1$.

Zunächst:

- Jeder children-Eintrag eines Tries vom Grad m ist ein eindimensionaler Vektor mit m Zeigern.
- Jedes Element im Vektor ist einem Zeichen oder einer Zeichenkombination zugeordnet.

Beispiel: Knoten eines Tries mit Ziffern als Schlüsselteilen.



- Implizite Zuordnung von Ziffer/Zeichen zu Zeiger. p_i gehört zur Ziffer i . Tritt Ziffer i in der betreffenden Position auf, so verweist p_i auf den Nachfolgerknoten. Kommt i in der betreffenden Position nicht vor, so ist p_i mit NULL belegt.

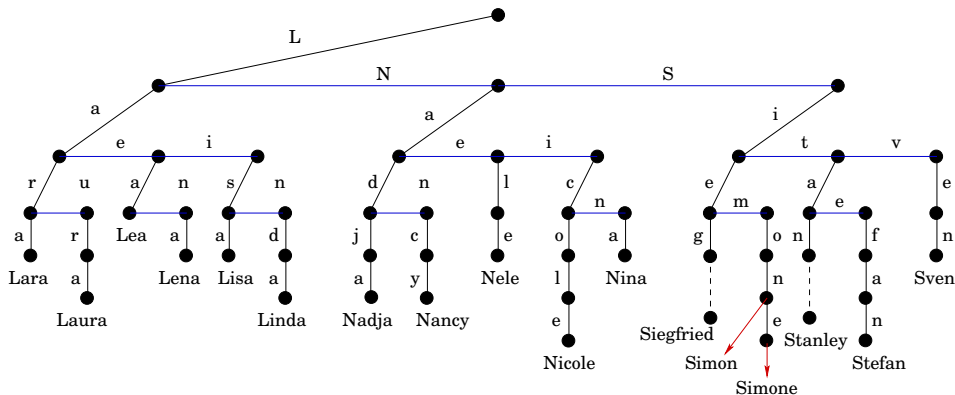
Beobachtungen:

- Die Höhe eines Tries wird durch den längsten abgespeicherten Schlüssel bestimmt.
- Die Gestalt des Baumes hängt von der Verteilung der Schlüssel ab, nicht von der Reihenfolge ihrer Abspeicherung.
- Knoten, die nur NULL-Zeiger besitzen, werden nicht angelegt.
- Schlechte Speicherplatzausnutzung aufgrund dünn besetzter Knoten und vieler Einweg-Verzweigungen in der Nähe der Blätter.

Lösung: Realisierung als binärer Baum oder mittels Hashing.

Realisierung als binärer Baum:

Speichere in jedem Knoten nur Verweise auf zwei Knoten: Einen zum ersten Kind und einen zum Geschwister.



Problem: Zugriff auf Kind mit bestimmten Namen nicht mehr in konstanter Zeit möglich.

Realisierung mittels Hashing:

- Speichere nur Referenzen auf Eltern-Knoten.
- Nutze Hash-Funktion, um zu einem Kind mit einem bestimmten Namen zu gelangen.
- Die Größe der Hash-Tabelle richtet sich nach der Größe des Wörterbuchs.

Näheres dazu in: Volker Heun. Grundlegende Algorithmen. Vieweg Verlag, 2003.

Übung:

- Implementieren Sie einen Splay-Baum und einen Trie, die jeweils die Methoden `insert`, `contains` und `toString` unterstützen.
- Schreiben Sie jeweils ein Programm zum automatischen Testen des entsprechenden Suchbaums.
- Fügen Sie nacheinander $2^{15}, 2^{16}, 2^{17}, \dots, 2^{25}$ zufällige Werte in die Suchbäume ein und messen Sie die Laufzeiten.
Vergleichen Sie die Laufzeiten mit den Laufzeiten Ihrer anderen Implementierungen von Suchbäumen.
- Ändert sich das Laufzeitverhalten, wenn die eingefügten Werte aufsteigend sortiert sind?