Algorithmen und Datenstrukturen

Herbst/Wintersemester 2020

Prof. Dr. Matthias Krause

3. November 2020, 14:17 Uhr

krause@uni-mannheim.de

Einführung

Einführung

Grundlegende Begriffe

Berechnungsprobleme

Definition 1

Berechnungsprobleme Π sind **Relationen** $\Pi \subseteq X \times Y$ zwischen einer Menge X gültiger Eingabedaten und einer Menge Y gültiger Ausgabedaten, $(x, y) \in \Pi$ bedeutet: y is Lösung zur Eingabe x bzgl. Π.

- **Sortierproblem:** Eingabe sind Folgen $\vec{a} = (a_1, \cdots, a_n)$ von Elementen einer geordneten Menge M, Ausgabe zu $\vec{a}$ ist eine sortierende Permutation $\pi \in \mathcal{S}_n$ so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.
  Beispiel: $\vec{a} = (8, 5, 9, 7)$ Ausgabe $\pi(1) = 2, \pi(2) = 4, \pi(3) = 1, \pi(4) = 3$, also $\pi = (1, 2, 4, 3)$ in Zyklenschreibweise.
- **Primzahltest:** Eingaben sind natürliche Zahlen n, Ausgaben sind **ja** (n ist Primzahl) und **nein** (n ist keine Primzahl), also
  Primzahltest$= \{(1, nein), (2, ja), (3, ja), (4, nein), (5, ja), (6, nein), (7, ja), \cdots\}$.

Typische Eingabe- bzw. Ausgabemengen

Im Kontext dieser Vorlesung sind folgende Eingabe- bzw. Ausgabemengen relevant:

(1) **Boolesche Konstanten** $\{0, 1\}$

(2) **Zahlen** (natürlich, ganz, rational, reell)

(3) **Vektoren bzw. Matrizen** mit Komponenten aus (1) oder (2).

(4) **Graphen** (ungerichtet, gerichtet)

(5) **Gewichtete und/oder markierte Graphen** (mit Knotenmarkierungen, mit Kantengewichten usw.)

Eingabelängen

Eingaben $x \in X$ einer Eingabemenge X werden in der Regel eine Eingabelänge $|x| \in \mathbb{N}$ zugeordnet, Beispiele:

- Natürliche Zahlen $x \in \mathbb{N}$, $|x| = \lfloor \log_2(x) \rfloor + 1$ Bitlänge von x,
  Beispiel $|9| = |1001| = 4$, $|91| = |1011011| = 7$.
- $m \times n$-Matrizen M über $\{0,1\}$, $|M| = m \cdot n$.
- Graphen $G = (V, E)$, $|G| = |V|$ oder $|G| = |V| + |E|$ oder $|G| = |E|$ je nach Anwendungszusammenhang.

Die Eingabelänge liefert eine **Partition** von X,

$$X = \bigcup_{n \in \mathbb{N}} X_n,$$

$X_n = \{x \in X, |x| = n\}$ Menge der Eingaben mit Eingabelänge n.

Beispiel $\mathbb{N}$: $X_1 = \{0,1\}$, $X_2 = \{2,3\}$, $X_3 = \{4,5,6,7\}, \cdots, X_n = \{2^{n-1}, \cdots, 2^n - 1\}$.

Algorithmen (allgemein)

- Ein Algorithmus bezieht sich auf einen Akteur, der in der Lage ist, auf einer vorgegenen Menge möglicher Eingabeobjekte Aktionen aus einer vorgegebenen Menge von möglichen Elementaraktionen auszuführen.
- Ein Algorithmus ist eine Handlungsvorschrift für diesen Akteur, in Abhängigkeit vom Eingabeobjekt eine eindeutig bestimmte Folge von Elementaraktionen auszuführen.
- **Beispiele:** Kochrezepte, Anleitungen zum Zusammenbau von IKEA-Möbeln usw.

Unser Kontext: Computer-Algorithmen

- Computer-Algorithmen beziehen sich auf Computer, die auf abgespeicherten Daten Operationen aus einer vorgegebenen Menge von Elementaroperationen (inklusive eines STOPP-Befehls) ausführen können.

- Ein Algorithmus A definiert eine Handlungsanweisung an den Computer, in Abhängigkeit von den gespeicherten Eingabedaten x eine eindeutig bestimmte Folge von Rechenschritten auszuführen (eine Elementaroperation pro Rechenschritt). Diese Folge heißt **Berechnung von** A **auf** x.

- Berechnungen können endlich (abbrechend) oder unendlich sein.

Notation von Algorithmen

- Wir formulieren Algorithmen als Computerprogramme für einen idealisierten Computer, der alle Eigenschaften realer Computer, jedoch einen potentiell unbegrenzten Speicher hat.

- Diese Programme werden in **Pseudocode**, d.h. in einer idealisierten höheren Programmiersprache, notiert, die folgenden Ansprüchen genügen sollte:
  - Algorithmen werden durch Pseudocode-Programme in einer intuitiv eingängigen Weise abgebildet.
  - Pseudocode-Programme können einfach in reale höhere Programmiersprachen (Java, C, Pascal usw,) übertragen werden.

- Wir verwenden den Pseudocode aus dem Buch von Cormen, Leiserson, Rivest, Stein *An Introduction to Algorithms* MIT Press 2009.

Beispiel Insertion Sort

INSERTION SORT(A)

(1) **For** $j \leftarrow 2$ **to** $length(A)$

(2) **do** $key \leftarrow A[j]$

(3) $i \leftarrow j - 1$

(4) **while** $i > 0$ and $key < A[i]$

(5) **do** $A[i + 1] \leftarrow A[i]$

(6) $i \leftarrow i - 1$

(7) $A[i + 1] \leftarrow key$

Lösen von Berechnungsproblemen durch Algorithmen

Definition 2

Ein Algorithmus A **löst** ein gegebenes Berechnungsproblem $\Pi \subseteq X \times Y$, (bzw. ist ein **korrekter** Algorithmus für Π) falls

- A bezieht sich auf eine **Eingabedatenstruktur**, d.h. auf eine Vorschrift, wie Eingaben $x \in X$ im Rechner abgespeichert werden.
- A bezieht sich auf eine **Ausgabedatenstruktur**, d.h. auf eine Vorschrift, wie die im Laufe einer Berechnung berechneten Daten als Ausgaben $y \in Y$ interpretiert werden.
- Auf jeder Eingabe $x \in X$ stoppt der Algorithmus nach endlich vielen Takten und produziert dabei eine Ausgabe $A(x) \in Y$, für die gilt $(x, A(x)) \in \Pi$.

Wir zeigen später, dass **Insertion Sort** ein korrekter Algorithmus für das Sortierproblem ist.

Kosten von Berechnungen

Definition 3

Gegeben ein Algorithmus A, der sich auf Eingaben x aus einer Eingabemenge X bezieht.

- Der **Zeitbedarf** $time_A(x)$ der Berechnung von A auf x ist gleich Summe der Zeitkosten der Rechenschritte der Berechnung von A auf x.
- Die Zuordnung der Zeitkosten zu den Rechenschritten ist modellierungsabhängig, häufiger Ansatz: ein Rechenschritt kostet eine Zeiteinheit.
- Der Speicherplatzbedarf $space_A(x)$ ist gleich der Anzahl der während der Berechnung von A auf x benutzten Speicherplätze.

Das Kostenverhalten von Algorithmen

Sei A ein Algorithmus, der Eingaben aus einer Menge $X = \bigcup_{n \in \mathbb{N}} X_n$ verarbeitet, $X_n = \{x \in X, |x| = n\}$.

- **Worst Case Laufzeit** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \max\{time_A(x), x \in X_n\}.$$

- **Best Case Laufzeit** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \min\{time_A(x), x \in X_n\}.$$

- **Average Case Laufzeit** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \mathbf{E}_{x \in_{P_n} X_n} time_A(x),$$

wobei P_n Wahrscheinlichkeitsverteilung auf X_n.

Entwurf und Analyse von Algorithmen

Algorithmen entwerfen und analysieren heißt

- **Entwurf** eines Algorithmus A für ein vorgegebenes Berechnungsproblem $\Pi \subseteq X \times Y$, $X = \bigcup_{n \in \mathbb{N}} X_n$, $X_n = \{x \in X, |x| = n\}$.
- **Korrektheitsbeweises**: Zeige, dass A auf allen Eingaben $x \in X$ hält, und dass $A(x)$ Lösung zu x bzgl. Π.
- **Laufzeitanalyse**: Bestimme die Wachstumsordnung von $time_A$ und ggf. $space_A$.

Wir analysieren **Insertion Sort**:

Analyse von Insertion Sort

INSERTION SORT(A)

(1) **For** $j \leftarrow 2$ **to** $length(A)$
(2) **do** $key \leftarrow A[j]$
(3) $i \leftarrow j - 1$
(4) **while** $i > 0$ and $key < A[i]$
(5) **do** $A[i+1] \leftarrow A[i]$
(6) $i \leftarrow i - 1$
(7) $A[i+1] \leftarrow key$

- $\cdots$ löst das Sortierproblem auf Eingaben $a_1, \cdots, a_n \in (M, \leq)$ in folgender Weise:
- **Eingabe-Datenstruktur:** Array $A = A[1, \cdots, length(A)]$ mit zusätzlicher Komponente $length(A) = n$. Es gilt $A[i] = a_i$, $i = 1, \cdots, n$.
- **Ausgabe-Datenstruktur:** sortiertes A, mit $A[i] = a_{\pi(i)}$, $i = 1, \cdots, n$ und $a_{\pi(1)} \leq \cdots \leq a_{\pi(n)}$.

Beispiel

INSERTION SORT(A)

(1) **For** $j \leftarrow 2$ **to** *length*(A)
(2) **do** $key \leftarrow A[j]$
(3) $i \leftarrow j - 1$
(4) **while** $i > 0$ and $key < A[i]$
(5) **do** $A[i+1] \leftarrow A[i]$
(6) $i \leftarrow i - 1$
(7) $A[i+1] \leftarrow key$

Beispiel $A = (18, 11, 15, 13, 25)$

Variablen	j	*key*	i
$(18, *, 15, 13, 25)$			11
$(*, 18, 15, 13, 25)$			11
$(11, 18, *, 13, 25)$			15
$(11, *, 18, 13, 25)$			15
$(11, 15, 18, *, 25)$			13
$(11, 15, *, 18, 25)$			13
$(11, *, 15, 18, 25)$			13
$(11, 13, 15, 18, *)$			25

$(11, 13, 15, 18, 25)$

Korrektheitsbeweis Insertion Sort

Theorem 4

Insertion Sort löst das Sortierproblem.

INSERTION SORT(A)

(1) **For** $j \leftarrow 2$ **to** *length*(A)
(2) **do** $key \leftarrow A[j]$
(3) $i \leftarrow j - 1$
(4) **while** $i > 0$ and $key < A[i]$
(5) **do** $A[i+1] \leftarrow A[i]$
(6) $i \leftarrow i - 1$
(7) $A[i+1] \leftarrow key$

Beweis: Zeigen für alle j, $1 \leq j \leq n$, dass $A^j[1] \leq A^j[2] \leq \cdots \leq A^j[j]$.

Hierbei bezeichnet A^j das Feld vor Durchlauf $j+1$.

Induktionsanfang: $j = 1$ offensichtlich, da $A^1 = A$.

Induktionsschritt: Für beliebiges $j > 1$ sei

$$A^{j-1}[1] \leq A^{j-1}[2] \leq \cdots \leq A^{j-1}[j-1].$$

In Durchlauf j werden alle Werte $A^{j-1}[k]$, $k \leq j-1$, für die $A^{j-1}[k] > A^{j-1}[j]$ gilt, um eine Position nach rechts geschoben (**while**-Schleife).

Die freiwerdende Position wird mit $A^{j-1}[j]$ überschrieben (Zeile (7)).

Somit gilt $A^j[1] \leq A^j[2] \leq \cdots \leq A^j[j]$. □

Laufzeitanalyse Insertion Sort

INSERTION SORT(A)

(1) **For** $j \leftarrow 2$ **to** $length(A)$
(2) **do** $key \leftarrow A[j]$
(3) $i \leftarrow j - 1$
(4) **while** $i > 0$ and $key < A[i]$
(5) **do** $A[i+1] \leftarrow A[i]$
(6) $i \leftarrow i - 1$
(7) $A[i+1] \leftarrow key$

Bei **vergleichsbasierten Sortieralgorithmen** zählt man nur die Vergleiche (mit Kosten 1).

Gesamtzeit: $time_{IS}(A) = \sum_{j=2}^{n} t_j$, wobei t_j die Anzahl der Aufruf von Zeile (4) in Durchlauf j ist.

Best Case: $t_j = 1$ für alle j, $2 \leq j \leq n$.

$$time_{IS}(A) = n - 1.$$

Tritt ein wenn A **aufsteigend sortiert**.

Worst Case: $t_j = j - 1$ für alle j, $2 \leq j \leq n$.

$$time_{IS}(A) = \frac{(n-1) \cdot n}{2}.$$

Tritt ein wenn A **absteigend sortiert**.

Einführung

Asymptotisches Wachstum von Funktionen

Die asymptotische Wachstumsordnung für monoton wachsende Funktionen

Man beachte die folgende Äquivalenzrelation.

Definition 5

Zwei monoton wachsende Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ haben die **gleiche asymptotische Wachstumsordnung**, falls die folgenden zwei Bedingungen erfüllt sind:

(i) f **wächst asymptotisch nicht schneller als** g, d.h., es existiert eine Konstante $C \in \mathbb{R}^+$ und ein Startwert $n_0 \in \mathbb{N}$, so dass $f(n) \leq C \cdot g(n)$ für alle $n \geq n_0$,

(ii) f **wächst asymptotisch nicht langsamer als** g, d.h., es existiert eine Konstante $c \in \mathbb{R}^+$ und ein Startwert $n_1 \in \mathbb{N}$, so dass $f(n) \geq c \cdot g(n)$ für alle $n \geq n_1$.

$\Theta(f)$ bezeichnet die **asymptotische Wachstumsordnung** von f, d.h., die Äquivalenzklasse der monoton wachsenden Funktionen, die die gleiche Wachstumsordnung wie f haben.

$O(f)$ bzw. $\Omega(f)$ bezeichnen die Menge von Funktionen, die, laut (i), nicht schneller wachsen als f, bzw., laut (ii), nicht langsamer wachsen als f, $\Theta(f) = O(f) \cap \Omega(f)$.

Einfache Eigenschaften

Lemma 6

Es seien $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ *monoton wachsend und* a *eine beliebige positive Konstante.*

(a) *Es gilt* $\Theta(f) = \Theta(a \cdot f)$,

(b) *Ist* $g \in O(f)$ *so ist* $\Theta(f) = \Theta(f + g)$.

Beweis: Der Beweis von (a) ergibt sich direkt aus Definition 5.

Für (b) genügt es zu zeigen, dass $f + g \in O(f)$, da aus $f \leq f + g$ direkt folgt, dass $f + g \in \Omega(f)$.

Da $g \in O(f)$ existiert eine positive Konstante C uns ein Startwert n_0, so dass $g(n) \leq C \cdot f(n)$ für $n \geq n_0$.

Damit ist $f(n) + g(n) \leq f(n) + C \cdot f(n) = (C + 1) \cdot f(n)$ für $n \geq n_0$. □

Beispiel $5n^2 + 3n + 7 \in \Theta(n^2)$, da $5n^2 \in \Theta(n^2)$ und $3n + 7 \in O(n^2)$.

Motivation

Bemerkung: Die Verwendung von Wachstumsordnungen beinhaltet die Vernachlässigung von multiplikativen Konstanten und additiven Termen nicht schnellerer Wachstumsordnung.

Im gegebenen Kontext ist es Standard, Algorithmen für gegebene Berechnungsprobleme unabhängig von einer konkreten Rechnerumgebung zu entwerfen und zu analysieren.

Da die Kosten pro Elementaroperation hardware- und implementierungsabhängig sind, ist es sinnvoll und allgemeiner Standard, bei der Laufzeitanalyse von Algorithmen lediglich die Wachstumsordnung der Laufzeit als Funktion in der Eingabelänge zu bestimmen.

Typische Wachstumsordnungen

- $\Theta(n)$ **lineares** Wachstum,
- $\Theta(n^2)$ **quadratisches** Wachstum,
- $\Theta(n^3)$ **kubisches** Wachstum,
- $O(1)$ **konstant beschränktes** Wachstum

Weitere typische Wachstumsordnungen

- $n^{O(1)}$ **polynomiell beschränktes** Wachstum, $n^{O(1)} = \bigcup_{k=0}^{\infty} \Theta(n^k)$
- $\Theta(\log(n))$ **logarithmisches** Wachstum

  Diese basisunabhängige Schreibweise ist gerechtfertigt durch das Logarithmengesetz $\log_a(n) = \log_a(b) \cdot \log_b(n)$, d.h. $\log_a(n) = \Theta(\log_b(n))$ für alle $a, b \geq 1$.
- $\exp(\Theta(n))$ **exponentielles** Wachstum, $\exp(\Theta(n)) = \bigcup_{a>0} \Theta(a^n)$

  Diese basisunabhängige Schreibweise ist gerechtfertigt durch das Potenzgesetz $a^n = 2^{\log_2(a) \cdot n}$, d.h. $a^n \in 2^{\Theta(n)}$ für alle $a > 1$.
- $\exp(\Omega(n^\delta))$, $0 < \delta < 1$, **schwach exponentielles** Wachstum
- $\log^{O(1)}(n)$ **polylogarithmische Beschränktheit**

Die o, ω-Notation

Definition 7

Es seien $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ monoton wachsende Funktionen.

- f **wächst asymptotisch echt langsamer als** g, falls für alle Konstanten $c \in \mathbb{R}^+$ ein Startwert $n_0 \in \mathbb{N}$ existiert, so dass $f(n) < c \cdot g(n)$ für alle $n \geq n_0$.
- Mit $o(g)$ wird die Menge aller monoton wachsenden Funktionen, die echt langsamer als g wachsen, bezeichnet.
- f **wächst asymptotisch echt schneller als** g, falls für alle Konstanten $C \in \mathbb{R}^+$ ein Startwert $n_0 \in \mathbb{N}$ existiert, so dass $f(n) > C \cdot g(n)$ für alle $n \geq n_0$.
- Mit $\omega(g)$ wird die Menge aller monoton wachsenden Funktionen, die echt schneller als g wachsen, bezeichnet.

Alternative Charakterisierung von Wachstumsordnungen

Lemma 8

Es seien $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ *monoton wachsende Funktionen.*

(a) ***Falls*** $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ ***so*** $f(n) \in \Theta(g(n))$.

(b) ***Falls*** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ ***so*** $f(n) \in o(g(n))$.

(c) ***Falls*** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ ***so*** $f(n) \in \omega(g(n))$.

Beweis (a): Es sei $C = \lim_{n\to\infty} \frac{f(n)}{g(n)}$.

Dann existiert eine Zahl n_0, so dass $C - 1 \leq \frac{f(n)}{g(n)} \leq C + 1$ für alle $n \geq n_0$.

Also gilt $f(n) \geq (C-1) \cdot g(n)$ und $f(n) \leq (C+1) \cdot g(n)$ für alle $n \geq n_0$.

Das impliziert $f \in O(g)$ und $f \in \Omega(g)$, also $f \in \Theta(g)$.

Beweis Lemma 8 und Anwendung auf Polynome

Beweis (b):

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ bedeutet, dass für **alle** $c > 0$ ein $n_0 \in \mathbb{N}$ existiert, so dass $\frac{f(n)}{g(n)} \leq c$ für alle $n \geq n_0$.

Also existiert für **alle** $c > 0$ ein $n_0 \in \mathbb{N}$ mit $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

Also $f \in o(g)$.

Beweis (c):

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ bedeutet, dass für alle $C \in \mathbb{N}$ ein $n_0 \in \mathbb{N}$ existiert, so dass $\frac{f(n)}{g(n)} \geq C$ für alle $n \geq n_0$.

Also existiert für **alle** $C > 0$ ein $n_0 \in \mathbb{N}$ mit $f(n) \geq C \cdot g(n)$ für alle $n \geq n_0$.

Also $f \in \omega(g)$. $\square$

Anwendung auf Polynome

Es seien p, q zwei Polynome vom Grad k bzw. r, d.h.,

$$p(n) = p_k n^k + p_{k-1} n^{k-1} + \cdots + p_1 n + p_0,$$

$$q(n) = q_r n^r + q_{r-1} n^{r-1} + \cdots + q_1 n + q_0.$$

In der **Schule** wurde für **gebrochen rationale Funktionen** gezeigt:

- Falls $k = r$ so $\lim_{n\to\infty} \frac{p(n)}{q(n)} = \frac{p_k}{q_r}$, (also $p(n) = \Theta(q(n))$).
- Falls $k < r$ so $\lim_{n\to\infty} \frac{p(n)}{q(n)} = 0$, (also $p(n) = o(q(n))$).
- Falls $k > r$ so $\lim_{n\to\infty} \frac{p(n)}{q(n)} = \infty$, (also $p(n) = \omega(q(n))$).

Fazit: Polynome gleichen Grades haben die gleiche Wachstumsordnung.

Polynome größeren Grades wachsen asymptotisch echt schneller als Polynome kleineren Grades.

Exponentielles versus polynomielles Wachstum

Lemma 9

Exponentielle Funktionen wachsen echt schneller als polynomielle, d.h. für alle Konstanten $k \in \mathbb{N}^+$ gilt $n^k = o(e^n)$.

Beweis:

Erinnerung $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.

Betrachten $p(n) = \sum_{i=0}^{k+1} \frac{n^i}{i!}$.

Es gilt $p(n) \in \Theta(n^{k+1})$ und, da $p(n) < e^n$, dass $p(n) \in o(e^n)$.

Also $n^k \in o(p(n))$ und $n^k \in o(e^n)$. $\square$

Bemerkung: Es gilt sogar, dass $n^k = o(e^{n^\delta})$ für alle Konstanten $0 < \delta < 1$.

D.h., polynomielles Wachstum ist asymptotisch echt langsamer als schwach exponentielles Wachstum.

Asymptotischen Wachstum, linear versus logarithmisch

Theorem 10

Logarithmische Funktionen wachsen echt langsamer als lineare, d.h. aus $f(n) \in O(\log(n))$ *folgt, dass* $f(n) \in o(n)$.

Beweis:

Setzen OBdA $f(n) = \ln(n)$.

Wir fixieren ein beliebig kleines $c > 0$ und großes $C > 0$ sodass $1/C \leq c$.

Dann existiert ein Startwert n_0, so dass für alle $n \geq n_0$ gilt $n^C < e^n$,

Also gilt $C \cdot \ln(n) < n$ und damit $\ln(n) < c \cdot n$ für $n \geq n_0$. $\square$

Bemerkung: Man kann zeigen, dass für beliebig kleine $\delta > 0$ und beliebig große $C \in \mathbb{N}$ gilt, dass falls $f(n) \in O(\log^C(n))$, so $f(n) \in o(n^\delta)$.

D.h., polylogarithmisches Wachstum ist echt langsamer als sublineares Wachstum.

Einführung

Laufzeitanalyse rekursiv definierter Algorithmen

Effizientes Mischen sortierter Teilfolgen von $A = A[1, \cdots, n]$

- Gegeben sei ein Eingabefeld A der Länge n für das Sortierproblem.
- In A seien die benachbarten Teilfelder $A[p \cdots q]$ der Länge $n_1 = q - p + 1$ und $A[q + 1 \cdots r]$ der Länge $n_2 = r - q$ bereits sortiert ($1 \leq p \leq q < r \leq n$).
- Der folgende Algorithmus $Merge(A, p, q, r)$ sortiert die entsprechende Teilfolge $A[p \cdots r]$ der Länge $r - p + 1$ in Linearerzeit $O(r - p)$.
- Hierzu werden die Teilfolgen $A[p \cdots q]$ und $A[q + 1 \cdots r]$ zunächst in Hilfsfelder L und R kopiert, die eine zusätzliche rechte Begrenzungsposition mit Eintrag ∞ haben.
- Dann werden aus L und R von links nach rechts jeweils das nächstgrößte Element nach A zurückkopiert.
- $Merge(A, p, q, r)$ bildet die Basis für das effiziente rekursive Sortierverfahren *MergeSort*.

Der Algorithmus $Merge(A, p, q, r)$ mit Beispiel

$\underline{Merge(A, p, q, r)}$

1 **For** $i = 1$ **to** n_1 **do**
  $L[i] = A[p + i - 1]$
2 **For** $j \leftarrow 1$ **to** n_2 **do** $R[j] \leftarrow A[q + j]$
3 $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4 $i \leftarrow 1, j \leftarrow 1$
5 **For** $k \leftarrow p$ **to** r **do**
6 **If** $L[i] \leq R[j]$
7 **then** $A[k] \leftarrow L[i]$; $i \leftarrow i + 1$
8 **else** $A[k] \leftarrow R[j]$; $j \leftarrow j + 1$

Beispiel $Merge(A, 1, 3, 6)$

für $A = [1, 2, 5, 3, 4, 6]$

$L[1, 2, 5, \infty]$	$R[3, 4, 6, \infty]$	$A[1, \cdot, \cdot, \cdot, \cdot, \cdot]$
$L[1, 2, 5, \infty]$	$R[3, 4, 6, \infty]$	$A[1, 2, \cdot, \cdot, \cdot, \cdot]$
$L[1, 2, 5, \infty]$	$R[3, 4, 6, \infty]$	$A[1, 2, 3, \cdot, \cdot, \cdot]$
$L[1, 2, 5, \infty]$	$R[3, 4, 6, \infty]$	$A[1, 2, 3, 4, \cdot, \cdot]$
$L[1, 2, 5, \infty]$	$R[3, 4, 6, \infty]$	$A[1, 2, 3, 4, 5, \cdot]$
$L[1, 2, 5, \infty]$	$R[3, 4, 6, \infty]$	$A[1, 2, 3, 4, 5, 6]$

Mergesort, ein rekursiver Sortieralgorithmus

MergeSort(A, p, r)

```
1  if 1 ≤ p < r ≤ length(A)
2    then q := ⌊(p + r)/2⌋
3         MergeSort(A, p, q)
4         MergeSort(A, q + 1, r)
5         Merge(A, p, q, r)
```

- *MergeSort*(A, p, r) sortiert die Teilfolge $A[p, \cdots, r]$, was direkt aus der Korrektheit von *Merge* folgt.
- Die Anzahl der Vergleiche $T(n)$ für *MergeSort*(A, p, r) genügt der Rekursionsgleichung

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n).$$

- **Frage:** Welche asymptotische Laufzeit hat **MergeSort**?

Beispiel MergeSort

$$[27, 53, 11, 67, 87, 23, 45, 61]$$

$$[27, 53, 11, 67] \quad [87, 23, 45, 61]$$

$$[27, 53] \quad [11, 67] \quad [87, 23] \quad [45, 61]$$

$$[27] \quad [53] \quad [11] \quad [67] \quad [87] \quad [23] \quad [45] \quad [61]$$

$$[27] \leftrightarrow [53] \quad [11] \leftrightarrow [67] \quad [87] \leftrightarrow [23] \quad [45] \leftrightarrow [61]$$

$$[27, 53] \leftrightarrow [11, 67] \quad [23, 87] \leftrightarrow [45, 61]$$

$$[11, 27, 53, 67] \leftrightarrow [23, 45, 61, 87]$$

$$[11, 23, 27, 45, 53, 61, 67, 87]$$

Auflösung der Rekursion $T(n) = a \cdot T(n/b) + f(n)$, $a, b \in \mathbb{N}^+$

Lemma 11

Unter der vereinfachenden Annahme, dass $T(1) = f(1) = 1$ und $n = b^N$ für ein $N \in \mathbb{N}$ gilt $T(n) = \sum_{i=0}^{N} a^i \cdot f\left(\frac{n}{b^i}\right)$.

Induktionsbeweis über N: Für $N = 1$ gilt $T(b) = a \cdot T(1) + f(b) = a^1 \cdot f(1) + a^0 \cdot f(b)$.

Wir fixieren ein beliebiges $N > 1$ und setzen voraus, dass Lemma 11 für alle $N' < N$ gilt.

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) = a \cdot T(b^{N-1}) + a^0 \cdot f\left(\frac{n}{b^0}\right)$$

$$= a \cdot \sum_{i=0}^{N-1} a^i \cdot f\left(\frac{n/b}{b^i}\right) + a^0 \cdot f\left(\frac{n}{b^0}\right) = \sum_{i=0}^{N-1} a^{i+1} \cdot f\left(\frac{n}{b^{i+1}}\right) + a^0 \cdot f\left(\frac{n}{b^0}\right)$$

$$= \sum_{i=1}^{N} a^i \cdot f\left(\frac{n}{b^i}\right) + a^0 \cdot f\left(\frac{n}{b^0}\right) = \sum_{i=0}^{N} a^i \cdot f\left(\frac{n}{b^i}\right). \square$$

Auflösung von $T(n) = a \cdot T(n/b) + f(n)$, $f(n) = n^c$, $c = \log_b(a)$

Wir betrachten den Spezialfall, dass $f(n) = n^c$ mit $c = \log_b(a)$.

Das entspricht der *MergeSort* Rekursion $T(n) = 2 \cdot T(n/2) + n$ mit $a = b = 2$ und $c = 1 = \log_2(2)$.

Man beachte, dass $b^c = a$.

Laut Lemma 11 gilt

$$T(n) = \sum_{i=0}^{N} a^i \cdot \left(\frac{n}{b^i}\right)^c = n^c \cdot \sum_{i=0}^{N} a^i \cdot (b^c)^{-i} = n^c \cdot \sum_{i=0}^{N} a^i \cdot a^{-i} = n^c \cdot (N+1).$$

Folgerung: Die Laufzeit von *MergeSort* ist $\Theta(n \cdot \log(n))$.

Das folgende **Master Theorem** umfasst den allgemeineren Fall, dass $f(n) = n^k$ für beliebige $k > 0$.

Auflösung von $T(n) = a \cdot T(n/b) + f(n)$, das Master Theorem

Theorem 12

Es gelte $T(n) = a \cdot T(n/b) + f(n)$, wobei der Term n/b hier sowohl gleich $\lfloor n/b \rfloor$ als auch gleich $\lceil n/b \rceil$ sein kann.

Es sei $c = \log_b(a)$, d.h. $b^c = a$.

(1) *Wenn $f(n) = \Theta(n^{c-\epsilon})$ für ein beliebiges $\epsilon > 0$, dann $T(n) = \Theta(n^c)$.*

(2) *Wenn $f(n) = \Theta(n^c)$, dann $T(n) = \Theta(n^c \cdot \log(n))$.*

(3) *Wenn $f(n) = \Omega(n^{c+\epsilon})$ für ein beliebiges $\epsilon > 0$, dann $T(n) = \Theta(f(n))$.*

- **Bsp 1:** $T(n) = 2 \cdot T(n/2) + n$, also $c = 1$, $f(n) = n$, also $T(n) = \Theta(n \cdot \log(n))$
- **Bsp 2:** $T(n) = 2 \cdot T(n/2) + n^2$, also $c = 1$, $f(n) = n^2$, also $T(n) = \Theta(n^2)$
- **Bsp 3:** $T(n) = 16 \cdot T(n/2) + n$, also $c = 4$, $f(n) = n$, also $T(n) = \Theta(n^4)$.

Beweisidee Master Theorem, (1)

Wir zeigen die Master Theorem Behauptungen (1) und (3) unter der vereinfachenden Annahme, dass $n = b^N$ und $T(1) = f(1) = 1$.

Für die Master Theorem Behauptung (2) ist das bereits geschehen.

Wir zeigen (1) und nehmen an, dass $f(n) = n^{c-\epsilon}$, $\epsilon > 0$.

Laut Lemma 11 gilt

$$T(n) = \sum_{i=0}^{N} a^i \cdot \left(\frac{n}{b^i}\right)^{c-\epsilon} = n^{c-\epsilon} \cdot \sum_{i=0}^{N} a^i \cdot \frac{b^{\epsilon \cdot i}}{b^{c \cdot i}} = n^{c-\epsilon} \cdot \sum_{i=0}^{N} a^i \cdot \frac{b^{\epsilon \cdot i}}{a^i} = n^{c-\epsilon} \cdot \sum_{i=0}^{N} b^{\epsilon \cdot i}.$$

Das ergibt

$$T(n) = n^{c-\epsilon} \cdot \frac{b^{\epsilon \cdot (N+1)} - 1}{b^{\epsilon} - 1} = C \cdot n^{c-\epsilon} \cdot (b^{\epsilon} \cdot n^{\epsilon} - 1) = C' \cdot n^c - C \cdot n^{c-\epsilon},$$

wobei C, C' Konstanten sind, die nicht von n abhängen, also $T(n) = \Theta(n^c)$. □

Beweisidee Master Theorem, (3)

Wir zeigen (3) und nehmen an, dass $f(n) = n^{c+\epsilon}$, für ein $\epsilon > 0$.

Laut Lemma 11 gilt

$$T(n) = \sum_{i=0}^{N} a^i \cdot \left(\frac{n}{b^i}\right)^{c+\epsilon} = n^{c+\epsilon} \cdot \sum_{i=0}^{N} a^i \cdot \frac{1}{a^i \cdot b^{\epsilon \cdot i}} = n^{c+\epsilon} \cdot \sum_{i=0}^{N} \left(\frac{1}{b^\epsilon}\right)^i = n^{c+\epsilon} \cdot \frac{1 - \left(\frac{1}{b^\epsilon}\right)^{N+1}}{1 - \frac{1}{b^\epsilon}}.$$

Daraus folgt $T(n) = \Theta(n^{c+\epsilon})$, da $\frac{1}{b^\epsilon} < 1$ und dementsprechend

$$1 = \frac{1 - \frac{1}{b^\epsilon}}{1 - \frac{1}{b^\epsilon}} < \frac{1 - \left(\frac{1}{b^\epsilon}\right)^{N+1}}{1 - \frac{1}{b^\epsilon}} < \frac{1}{1 - \frac{1}{b^\epsilon}}. \square$$

Bemerkung: Der vollständige Beweis des Master Theorems ist aufwändiger, da die durch die Verwendung von $\lfloor n/b \rfloor$ und $\lceil n/b \rceil$ in jeder Iteration auftretenden Rundungsfehler berücksichtigt werden müssen.

Einführung

Diskrete Wahrscheinlichkeitsräume

Diskrete endliche Wahrscheinlichkeitsräume

Definition 13

- ... sind Paare (Ω, ρ), wobei Ω eine nichtleere endliche Menge und ρ eine **Wahrscheinlichkeitsverteilung** auf Ω ist, d.h. für $\rho : \Omega \longrightarrow [0, 1]$ gilt $\sum_{x \in \Omega} \rho(x) = 1$.
- Die Elemente $x \in \Omega$ heißen **Elementarereignisse**, Teilmengen $A \subseteq \Omega$ heißen **Ereignisse**.
- Die **Wahrscheinlichkeit** $Pr_{x \in_\rho \Omega}[x \in A]$ **eines Ereignisses** $A \subseteq \Omega$ (kurz $Pr(A)$) ist definiert als
$$Pr_{x \in_\rho \Omega}[x \in A] = \sum_{x \in A} \rho(x).$$
- (Ω, U) heißt **Gleichverteilung** auf Ω, falls $U(x) = \frac{1}{|\Omega|}$ für alle $x \in \Omega$.
- (Ω, χ_x) heißt **Punktverteilung** bezüglich $x \in \Omega$, falls $\chi_x(x') = 1$ falls $x' = x$ und 0 sonst.

Beispiele

- **Faire Münze:** Gleichverteilung auf $\Omega = \{0, 1\}$, $Pr[0] = Pr[1] = \frac{1}{2}$.
- n **unabhängige Münzwürfe:** Gleichverteilung auf $\Omega = \{0, 1\}^n$, $Pr[b_1, \cdots, b_n] = \frac{1}{2^n}$.
- n **unabhängige Münzwürfe:** $Pr[\text{Werfe } k \text{ Mal eine Eins}] = \binom{n}{k} \cdot 2^{-n}$.
- **Fairer Würfel:** Gleichverteilung auf $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- **Fairer Würfel:** $Pr[\text{Würfle eine Primzahl}] = Pr[\{2, 3, 5\}] = \frac{3}{6} = \frac{1}{2}$.
- **Zwei unabhängige Würfel:** Gleichverteilung auf $\Omega = \{1, 2, 3, 4, 5, 6\} \times \{1, 2, 3, 4, 5, 6\}$.
- **Zwei unabhängige Würfel:** $Pr[\text{Pasch}] = Pr[\{(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)\}] = \frac{6}{36} = \frac{1}{6}$.
- **Zwei unabhängige Würfel:** $Pr[\text{Mäxchen}] = Pr[\{(1, 2), (2, 1)\}] = \frac{2}{36} = \frac{1}{18}$.

Weitere Beispiele

Gegeben eine Urne mit n verschiedenen Kugeln $K_1, \ldots, K_n$

- r **Mal Ziehen mit Zurücklegen** $Pr[K_{i_1}, \cdots, K_{i_r}] = \frac{1}{n^r}$.
- r **Mal Ziehen ohne Zurücklegen** $Pr[K_{i_1}, \cdots, K_{i_r}] = \frac{1}{n} \cdot \frac{1}{n-1} \cdots \frac{1}{n-r+1} = \frac{(n-r)!}{n!}$, wobei $i_j \neq i_k$ für alle $1 \leq j \neq k \leq r$.

Informal

- Wahrscheinlichkeitsräume (Ω, ρ) modellieren das Wissen eines Akteurs über den in der Zukunft liegenden Ausgang eines Experiments, wobei Ω die Menge der möglichen Versuchsausgänge bezeichnet.
- Die Wahrscheinlichkeitswerte $\rho(x)$ bezeichnen die Wahrscheinlichkeiten, mit denen der Versuchsausgang $x \in \Omega$ aus Sicht des Akteurs eintritt.
- Das Vorliegen der Gleichverteilung bedeutet **maximale Unsicherheit** des Akteurs über den Versuchsausgang (alle möglichen Versuchsausgänge sind gleichwahrscheinlich).
- Das Vorliegen einer Punktverteilung χ_x, wobei $x \in \Omega$, bedeutet **maximale Sicherheit** des Akteurs über den Versuchsausgang (dieser ist mit Sicherheit x).

Weitere Beispiele

- **Bernoulliverteilung $B(p)$:** $\Omega = \{0, 1\}$, $0 \leq p \leq 1$

$$Pr_{x \in_{B(p)} \{0,1\}}[x = 1] = p, Pr_{x \in_{B(p)} \{0,1\}}[x = 0] = 1 - p.$$

  Beispiel Münzwurf ist $B(\frac{1}{2})$.

- **Binomialverteilung $B(n, \frac{1}{2})$:** $\Omega = \{0, 1, \cdots, n\}$,

$$Pr_{x \in_{B(n, \frac{1}{2})} \Omega}[x = i] = \binom{n}{i} \cdot 2^{-n}.$$

  Erinnerung: $\sum_{i=0}^{n} \binom{n}{i} = 2^n$.

- **Beispiel:** Die Wahrscheinlichkeit, dass bei 8 Münzwürfen 4 Mal Zahl geworfen wird, ist

$$\binom{8}{4} \cdot 2^{-8} = \frac{8!}{4! \cdot 4!} \cdot \frac{1}{256} = \frac{8 \cdot 7 \cdot 6 \cdot 5}{1 \cdot 2 \cdot 3 \cdot 4} \cdot \frac{1}{256} = \frac{70}{256} = \frac{35}{128} = 0.2734375.$$

Rechenregeln für Ereignisse

- Es gilt $\Pr(\emptyset) = 0$ und $\Pr(\Omega) = 1$.
- Schließen sich die Ereignisse $A, B \subseteq \Omega$ gegenseitig aus (d.h. $A \cap B = \emptyset$), so gilt

$$\Pr(A \cup B) = \Pr(A) + \Pr(B).$$

- Für das Komplementärereignis $\bar{A} := \Omega \setminus A$ eines Ereignisses $A \subseteq \Omega$ gilt

$$\Pr(\bar{A}) = 1 - \Pr(A).$$

- Für Ereignisse $A, B \subseteq \Omega$ gilt allgemein

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B).$$

Bedingte Wahrscheinlichkeiten

Bedingte Wahrscheinlichkeiten $Pr(A|B)$ messen die Wahrscheinlichkeit des Eintretens eines Ereignisses $A \subseteq \Omega$ unter der Bedingung, dass das Ereignis $B \subseteq \Omega$ bereits eingetreten ist.

Das Eingetretensein von B entspricht einer Wahrscheinlichkeitsverteilung $\rho|_B$ auf Ω mit:

$$\rho|_B(x) = \begin{cases} \frac{\rho(x)}{\Pr(B)} & \text{falls } x \in B \\ 0 & \text{falls } x \notin B \end{cases}$$

Die Formel der bedingten Wahrscheinlichkeit: Die bedingte Wahrscheinlichkeit $\Pr(A|B)$ des Ereignisses A unter der Bedingung, dass B bereits eingetreten ist, ergibt sich als

$$\Pr(A|B) = \sum_{x \in A} \rho|_B(x) = \sum_{x \in A \cap B} \frac{\rho(x)}{\Pr(B)} = \frac{\Pr(A \cap B)}{\Pr(B)}.$$

Beispiel

- **Bob** würfelt $x \in \{1,2,3,4,5,6\}$ verdeckt mit einem fairen Würfel.
- **Alice** soll raten, ob eine gerade oder eine ungerade Augenzahl gewürfelt wurde, d.h. ob $x \in A_1 = \{2,4,6\}$ oder $x \in A_2 = \{1,3,5\}$. Es gilt $\Pr[A_1] = Pr[A_2] = \frac{1}{2}$.
- **Eve** verrät Alice, dass eine Primzahl gewürfelt wurde, d.h. $x \in B = \{2,3,5\}$.
- Es gilt $\Pr(A_1|B) = \frac{\Pr[\{2\}]}{\Pr[\{2,3,5\}]} = \frac{1/6}{1/2} = \frac{1}{3}$,
- $\Pr(A_2|B) = \frac{\Pr[\{3,5\}]}{\Pr[\{2,3,5\}]} = \frac{1/3}{1/2} = \frac{2}{3}$.
- **Alice** rät **ungerade**, da auf Grund des Symptoms B die Hypothese, dass eine ungerade Zahl gewürfelt wurde, die wahrscheinlichere ist.

Bayessche Entscheidungstheorie

Thomas Bayes (1701-1776) engl. Mathematiker, Philosoph, Pfarrer

- Wir ziehen x zufällig aus Ω und beobachten das **Symptom** $x \in B$, $B \subseteq \Omega$ Ereignis.
- Ω sei partitioniert in n disjunkte Teilmengen $A_1, \cdots, A_n$, die möglichen **Ursachen** (z.B. Krankheiten) des Symptoms B.
- Wir kennen die Wahrscheinlichkeiten $\Pr(A_i)$ und $\Pr(B|A_i)$ für alle $i = 1, \cdots, n$ (z.B. aus medizinischen Lehrbüchern).
- Wir möchten eine möglichst gute Hypothese über das i abgeben, für das $x \in A_i$ gilt, d.h. welche Ursache das Symptom verursacht hat.
- **Lösung Bayes Entscheidung:** Wählen das i, $1 \leq i \leq n$, so dass $\Pr(A_i|B)$ maximal ist.

Die Bayessche Formel

- Es gilt für alle $i = 1, \cdots, n$, dass

$$Pr(A_i|B) = \frac{Pr(A_i \cap B)}{Pr(B)} = \frac{Pr(B|A_i) \cdot Pr(A_i)}{Pr(B)}.$$

- Außerdem gilt

$$Pr(B) = \sum_{j=1}^{n} Pr(B \cap A_j) = \sum_{j=1}^{n} Pr(B|A_j) \cdot Pr(A_j).$$

- Das ergibt die **Bayessche Formel**

$$Pr(A_i|B) = \frac{Pr(B|A_i) \cdot Pr(A_i)}{\sum_{j=1}^{n} Pr(B|A_j) \cdot Pr(A_j)}.$$

- **Bayes Entscheidung:** Wähle i mit $Pr(B|A_i) \cdot Pr(A_i)$ maximal.

Beispiel Bayes'sche Formel

- Die Ärztin stellt bei einem Patienten das seltene Symptom S fest.
- Es ist bekannt, dass nur vier seltenen Krankheiten, nämlich K_1, K_2, K_3, K_4, für S ursächlich seien können.
- K_1 tritt bei einem von hunderttausend Männern im Verlaufe seines Lebens auf, bei K_2 sind es vier, bei K_3 fünf, bei K_4 drei,
  d.h. $Pr(K_1) = \frac{1}{13}$, $Pr(K_2) = \frac{4}{13}$, $Pr(K_3) = \frac{5}{13}$, $Pr(K_4) = \frac{3}{13}$.
- Von zehn an K_1 erkrankten Männern weisen neun das Symptom S auf, bei K_2 sind es drei, bei K_3 fünf, bei K_4 vier,
  d.h. $Pr(S|K_1) = \frac{9}{10}$, $Pr(S|K_2) = \frac{3}{10}$, $Pr(S|K_3) = \frac{5}{10}$, $Pr(S|K_4) = \frac{4}{10}$.
- Also $Pr(S|K_1) \cdot Pr(K_1) = \frac{9}{130}$, $Pr(S|K_2) \cdot Pr(K_2) = \frac{12}{130}$, $Pr(S|K_3) \cdot Pr(K_3) = \frac{25}{130}$, $Pr(S|K_4) \cdot Pr(K_4) = \frac{12}{130}$.
- Die Bayes'sche Hypothese ist K_3, da $Pr[K_3|S]$ maximal.

Unabhängigkeit von Ereignissen

Definition 14

- Ereignisse $A, B \subseteq \Omega$ heißen **unabhängig**, wenn die Wahrscheinlichkeit des Eintretens von A unabhängig davon ist, ob B bereits eingetreten ist, d.h., wenn $Pr(A) = Pr(A|B)$ bzw., was äquivalent dazu ist, wenn $Pr(A \cap B) = Pr(A) \cdot Pr(B)$.
- Eine Menge von Ereignisse $A_1, \cdots, A_n$ heißt **unabhängig** von einer anderen Menge von Ereignissen $B_1, \cdots, B_m$, falls jedes Ereignis A, dass sich per Vereinigung, Durchschnitt oder Komplementbildung aus $A_1, \cdots, A_n$ bilden lässt, unabhängig ist von jedem Ereignis B, dass sich per Vereinigung, Durchschnitt oder Komplementbildung aus $B_1, \cdots, B_m$ bilden lässt
- Eine Menge von Ereignisse $A_1, \cdots, A_n$ heißt **untereinander unabhängig**, falls jede echte nichtleere Teilmenge $\{A_i; i \in I\}$ von diesen Ereignissen unabhängig von der jeweiligen Komplementärmenge $\{A_i; i \notin I\}$ von Ereignissen ist.

Unabhängigkeit von Ereignissen, Erläuterung

Definition 14 ist zwar intuitiv einleuchtend, zeigt aber erstmal keinen effizienten Weg auf, die Unabhängigkeit mehrerer Ereignisse zu nachzuweisen. Das leistet die folgende einfachere Charakterisierung, deren Richtigkeit für Interessierte im Anhang gezeigt wird.

Lemma 15

Ereignisse $A_1, \cdots, A_n$ sind genau dann ***untereinander unabhängig****, wenn für alle nichtleeren Indexmengen $I \subseteq \{1, \cdots, n\}$ gilt, dass*

$$Pr\left(\bigcap_{i \in I} A_i\right) = \prod_{i \in I} Pr(A_i). \square$$

Folgerung: Ereignisse A, B sind genau dann unabhängig, falls $Pr[A \cap B] = Pr[A] \cdot Pr[B]$.

Beweis: $Pr[A] = Pr[A|B] \iff Pr[A] = \frac{Pr[A \cap B]}{Pr[B]} \iff Pr[A] \cdot Pr[B] = Pr[A \cap B]. \square$

Beispiel Unabhängigkeit von Ereignissen

Beispiel: Würfeln mit zwei Würfeln: Ist Ereignis A, dass die Summe der Augenzahlen 6 ist, unabhängig von Ereignis B, dass mindestens eine Augenzahl ungerade ist?

- $A = \{(1,5),(5,1),(4,2),(2,4),(3,3)\}$
  $\Longrightarrow \; Pr[A] = \frac{5}{36}$
- $\bar{B} = \{2,4,6\} \times \{2,4,6\}$
  $\Longrightarrow \; Pr[B] = 1 - \frac{9}{36} = \frac{27}{36} = \frac{3}{4}$
- $A \cap B = \{(1,5),(5,1),(3,3)\}$
  $\Longrightarrow \; Pr[A \cap B] = \frac{3}{36} = \frac{1}{12}$
  $\Longrightarrow \; Pr[A] \cdot Pr[B] = \frac{5}{36} \cdot \frac{3}{4} = \frac{5}{48} \neq \frac{1}{12}.$
- $\Longrightarrow A$ und B **nicht** unabhängig.

Unabhängige Versuche

Definition 16

Versuchsaufbau: Wir führen das Experiment zu einem Wahrscheinlichkeitsraum (Ω, ρ) n-mal unabhängig voneinander hintereinander aus.

Das entspricht dem Wahrscheinlichkeitsraum $(\Omega^n, \rho^{(n)})$, wobei für die entsprechenden Elementarereignisse $(x_1, \cdots, x_n) \in \Omega^n$ gilt

$$\rho^{(n)}(x_1, \cdots, x_n) = \rho(x_1) \cdot \rho(x_2) \cdot \cdots \cdot \rho(x_n).$$

Beispiel: n unabhängige Münzwürfe entsprechen $(\{0,1\}^n, Pr)$ mit

$$Pr[(b_1, \cdots, b_n)] = \frac{1}{2} \cdot \frac{1}{2} \cdots \frac{1}{2} = 2^{-n}.$$

Wir betrachten das Ereignis, dass genau k mal Kopf erscheint, wobei $0 \leq k \leq n$. Dann gilt

$$Pr\left[\sum_{i=1}^{n} x_i = k\right] = 2^{-n}\binom{n}{k}.$$

Zufallsgrößen

Definition 17

Es sei (Ω, ρ) ein Wahrscheinlichkeitsraum. Eine Funktion $X : \Omega \to \mathbb{R}$ heißt **Zufallgröße über** (Ω, ρ).

Wir betrachten hauptsächlich Zufallsgrößen $X : \Omega \longrightarrow \{0, 1, \cdots, n\}$ für geeignete natürliche Zahlen n.

Wichtig: Jede Zufallsgrößen $X : \Omega \longrightarrow \{0, 1, \cdots, n\}$ definiert eine Wahrscheinlichkeitsverteilung auf $\{0, \cdots, n\}$,

$$Pr[i] = Pr[X = i] = Pr_{\omega \in_\rho \Omega}[X(\omega) = i]$$

für alle i, $0 \leq i \leq n$.

Beispiel: Ω ist Menge der Eingaben einer Länge n für einen Algorithmus A. Dann ist $time_A : \Omega \to \mathbb{N}$ eine Zufallsgröße.

Klassische Zufallsgrößen

- Eine Zufallsgröße $X : \Omega \longrightarrow \{0, 1\}$ definiert die Bernoulli-Verteilung $B(p)$ auf $\{0, 1\}$ mit
$$p = Pr[X = 1].$$

- Die Zufallsgröße X über $\{0, 1\}^n$ mit der Gleichverteilung, wobei
$$X(x_1, \cdots, x_n) = x_1 + x_2 + \cdots + x_n,$$
definiert die Binomialverteilung $B(n, \frac{1}{2})$ auf $\{0, 1, \cdots, n\}$ mit
$$Pr\,[X = k] = 2^{-n}\binom{n}{k}.$$

- Betrachten $\mathcal{S}_n$ mit der Gleichverteilung und $X : \mathcal{S}_n \longrightarrow \{1, \cdots, n\}$ mit $X(\sigma) = \sigma(n)$. Dann gilt
$$Pr[X = i] = Pr_{\sigma \in_U \mathcal{S}_n}[\sigma(n) = i] = \frac{1}{n},$$
d.h. X definiert die Gleichverteilung auf $\{1, \cdots, n\}$.

Unabhängige Zufallsgrößen

Definition 18

Zufallsgrößen $X_1, \cdots, X_m : \Omega \longrightarrow \{0, 1, \cdots, n\}$ heißen **unabhängig**, falls für alle Folgen $(i_1, \cdots, i_m)$ mit Elementen aus $\{0, 1, \cdots, n\}$ gilt, dass die Ereignisse $X_1 = i_1, \cdots, X_m = i_m$ untereinander unabhängig sind.

Wichtiges Beispiel: Sei $X : \Omega \longrightarrow \{0, 1, \cdots, n\}$ Zufallsgröße bzgl. (Ω, ρ).

Wir betrachten m unabhängige Versuche des entsprechenden Experiments und messen in jedem Versuch den Wert von X.

Das entspricht den Zufallsfunktionen $X_1, \cdots, X_m : \Omega^m \longrightarrow \{0, 1, \cdots, n\}$ mit

$$X_j(\omega_1, \cdots, \omega_m) = X(\omega_j),$$

d.h. X_j entspricht dem X-Wert des j-then Ausführung des Experiments, $j = 1, \cdots, m$.

Erwartungswerte

Definition 19

Es sei $X : \Omega \longrightarrow \{0, 1, \cdots, n\}$ eine Zufallsgröße über dem Wahrscheinlichkeitsraum (Ω, ρ).

Der Erwartungswert $\mathbf{E}[X]$ von X (bzgl. (Ω, ρ)) ist definiert als

$$\mathbf{E}[X] = \sum_{i=0}^{n} Pr[X = i] \cdot i.$$

- Der Erwartungswert $\mathbf{E}[X]$ bildet den entsprechend der Auftrittswahrscheinlichkeiten gewichteten Mittelwert von X.
- Der Erwartungswert $\mathbf{E}[X]$ ist die **beste Hypothese** über den zu erwartenden Wert $X(\omega)$, wobei $\omega \in \Omega$ den in der Zukunft liegenden Versuchsausgang des Experiments zu (Ω, ρ) bezeichnet (siehe nächste Folien).

Der Abstand zweier Zufallsgrößen

Definition 20

Der **Abstand** $d(X, Y)$ zweier Zufallsgrößen $X, Y : \Omega \longrightarrow \{0, 1, \cdots, n\}$ wird folgendermaßen definiert.

$$d(X, Y) = \sqrt{\sum_{\omega \in \Omega} Pr_{\Omega}[\omega] \cdot (X(\omega) - Y(\omega))^2}.$$

Eine **Hypothese** h über den zu erwartenden Wert $X(\omega)$, $\omega \in \Omega$, entspricht der konstanten Zufallsgröße $h : \Omega \longrightarrow \{0, 1, \cdots, n\}$ mit $h(\omega) = h$ für alle $\omega \in \Omega$.

Die beste Hypothese für X entspricht somit der Konstante h, die den Abstand $d(X, h)$ minimiert.

Statt $d(X, h)$ minimieren wir $d^2(X, h)$ durch Lösen von $\left[d^2(X, h)\right]' = 0$.

Der Erwartungswert ist die beste Hypothese

Es gilt

$$d^2(X,h) = \sum_{\omega\in\Omega} Pr_\Omega[\omega]\cdot(X(\omega)-h))^2 = \sum_{i=0}^{n} Pr[X=i]\cdot(h-i)^2.$$

und

$$\left[d^2(X,h)\right]' = \left[\sum_{i=0}^{n} Pr[X=i]\cdot(h-i)^2\right]' = 2\left(\sum_{i=0}^{n} Pr[X=i]\cdot(h-i)\right)$$

$$= 2\left(\sum_{i=0}^{n} Pr[X=i]\cdot h - \sum_{i=0}^{n} Pr[X=i]\cdot i\right) = 2\cdot(h-\mathbf{E}[X]).$$

D.h. $\left[d^2(X,h)\right]' = 0$ für $h = \mathbf{E}[X]$,

d.h. $d(h)$ ist minimal für $h = \mathbf{E}[X]$.

Rechenregeln für Erwartungswerte, I

Im Folgenden seien $X, Y : \Omega \longrightarrow \{0, 1, \cdots, n\}$ Zufallsgrößen über dem Wahrscheinlichkeitsraum (Ω, ρ) und $a \in \mathbb{R}$.

Lemma 21

Es gilt $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.

Beweis: Es gilt

$$\mathbf{E}[X+Y] = \sum_{i=0}^{2n} Pr[X+Y=i] \cdot i = \sum_{k=0}^{n} \sum_{j=0}^{n} Pr[(X=k) \wedge (Y=j)](k+j)$$

$$= \sum_{k=0}^{n} \left(\sum_{j=0}^{n} Pr[(X=k) \wedge (Y=j)] \right) \cdot k + \sum_{j=0}^{n} \left(\sum_{k=0}^{n} Pr[(X=k) \wedge (Y=j)] \right) \cdot j$$

$$= \sum_{k=0}^{n} Pr[X=k] \cdot k + \sum_{j=0}^{n} Pr[Y=j] \cdot j = \mathbf{E}[X] + \mathbf{E}[Y]. \quad \square$$

Rechenregeln für Erwartungswerte, II

Lemma 22

Es gilt $\mathbf{E}[a \cdot X] = a \cdot \mathbf{E}[X]$.

Beweis: Es gilt

$$\mathbf{E}[a \cdot X] = \sum_{i=0}^{n} Pr[a \cdot X = a \cdot i] \cdot a \cdot i = a \cdot \sum_{i=0}^{n} Pr[X = i] \cdot i = a \cdot \mathbf{E}[X]. \quad \square$$

Rechenregeln für Erwartungswerte, III

Lemma 23

Sind X, Y unabhängig, so gilt $\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$.

Beweis: Es gilt

$$\mathbf{E}[X \cdot Y] = \sum_{i=0}^{n^2} Pr[X \cdot Y = i] \cdot i$$

$$= \sum_{k=0}^{n} \sum_{j=0}^{n} Pr[(X = k) \wedge (Y = j)] \cdot k \cdot j = \sum_{k=0}^{n} \sum_{j=0}^{n} Pr[X = k] \cdot [Y = j] \cdot k \cdot j$$

$$= \left(\sum_{k=0}^{n} Pr[X = k] \cdot k \right) \left(\sum_{j=0}^{n} Pr[Y = j] \cdot j \right) = \mathbf{E}[X] \cdot \mathbf{E}[Y]. \quad \square$$

Beispiele Erwartungswerte

- **Bernoulli-Verteilung:** Sei $X : \Omega \longrightarrow \{0, 1\}$ $B(p)$-verteilt.

$$\mathbf{E}[X] = (1 - p) \cdot 0 + p \cdot 1 = p,$$

- **Binomial-Verteilung:** Sei $X : \Omega \longrightarrow \{0, \cdots, n\}$ $B(n, p)$-verteilt, d.h. $X = \sum_{j=1}^{n} X_j$, wobei X_j untereinander unabhängig und $B(p)$-verteilt. Dann

$$\mathbf{E}[X] = \sum_{j=1}^{n} \mathbf{E}[X_j] = n \cdot p.$$

- **Gleichverteilung:** Sei $X : \Omega \longrightarrow \{1, \cdots, n\}$ gleichverteilt.

$$\mathbf{E}[X] = \sum_{i=1}^{n} Pr[X = i] \cdot i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- **Punktverteilung:** Sei $X : \Omega \longrightarrow \{1, \cdots, n\}$ Punktverteilung mit $X(\omega) = j$ für alle $\omega \in \Omega$. Dann gilt $\mathbf{E}[X] = 1 \cdot j = j$.

Bedingte Erwartungswerte

Es sei (Ω, ρ) Wahrscheinlichkeitsraum, $X : \Omega \longrightarrow \{0, \cdots, n\}$ Zufallsgröße und $B \subseteq \Omega$ ein Ereignis.

Der **bedingte Erwartungswert** $\mathbf{E}[X|B]$ misst den Erwartungswert von X bezüglich der Verteilung $\rho|_B$, d.h. unter der Bedingung dass B bereits eingetreten ist, d.h.

$$\mathbf{E}[X|B] = \sum_{i=0}^{n} Pr[X = i|B] \cdot i.$$

Wir betrachten nun eine Partition von Ω in m disjunkte Ereignisse $A_1, \cdots, A_m$.

D.h. $A_j \cap A_k = \emptyset$ für $j \neq k \in \{1, \cdots, m\}$ und

$$\Omega = \bigcup_{j=1}^{m} A_j.$$

Wir berechnen $\mathbf{E}[X]$ unter der Bedingung, dass wir die bedingten Erwartungswerte $\mathbf{E}[X|A_j]$ kennen.

Formel des bedingten Erwartungswerts

Lemma 24

Es gilt

$$\mathbf{E}[X] = \sum_{j=1}^{m} Pr[A_j] \cdot \mathbf{E}[X|A_j].$$

Beweis: Es gilt

$$\mathbf{E}[X] = \sum_{i=0}^{n} Pr[X = i] \cdot i = \sum_{i=0}^{n} \left(\sum_{j=1}^{m} Pr[(X = i) \cap A_j] \right) \cdot i$$

$$= \sum_{i=0}^{n} \left(\sum_{j=1}^{m} Pr[X = i|A_j] \cdot Pr[A_j] \right) \cdot i = \sum_{j=1}^{m} Pr[A_j] \cdot \left(\sum_{i=0}^{n} Pr[X = i|A_j] \cdot i \right)$$

$$= \sum_{j=1}^{m} Pr[A_j] \cdot \mathbf{E}[X|A_j]. \; \square$$

Sortieralgorithmen

Sortieralgorithmen

Allgemeines zu Sortieralgorithmen

Motivation und grundlegende Begriffe

- Sortieralgorithmen werden in vielen komplexeren Algorithmen als Grundoperation aufgerufen.
- In vielen zeitkritischen Anwendungen stellt das Sortieren großer Mengen von Daten einen Flaschenhals dar.
- In die Suche von Algorithmen zum schnellen Sortieren wurde auf Grund der großen praktischen Relevanz viel Forschungsarbeit investiert.
- **Vergleichsbasierte Sortieralgorithmen** benutzen als Grundoperationen die **Vergleichsoperation** auf der zugrundeliegenden geordneten Menge.
- Sie sind somit für beliebige geordnete Mengen anwendbar **(Beispiele Insertionsort, Mergesort, Heapsort, Quicksort)**.
- **Nicht vergleichsbasierte Sortieralgorithmen** nutzen die konkrete Struktur der zugrundeliegenden geordneten Menge aus **(Beispiele Countingsort, Radixsort).**
- **Inplace Sortieralgorithmen** sortieren das Eingabefeld ohne zusätzliche Felder als Hilfsdatenstruktur zu benutzen.

Sortieralgorithmen

Heapsort

Heapsort, die Heapsort Baumstruktur

Heapsort ist ein **vergleichsbasiertes Inplace** Sortierverfahren.

Es benutzt eine **Binärbaumstruktur** auf auf der Knotenmenge $[n] = \{1, \cdots, n\}$:

- Wurzel ist 1,
- $Parent(i) = \lfloor i/2 \rfloor$ (für $i \geq 2$)
- $Left(i) = 2i$ (für $i \leq n/2$)
- $Right(i) = 2i + 1$ (für $i \leq (n-1)/2$)

Definition 25

Ein Feld A heißt **Heap**, falls für alle $i = 2, \cdots, heapsize(A)$ gilt

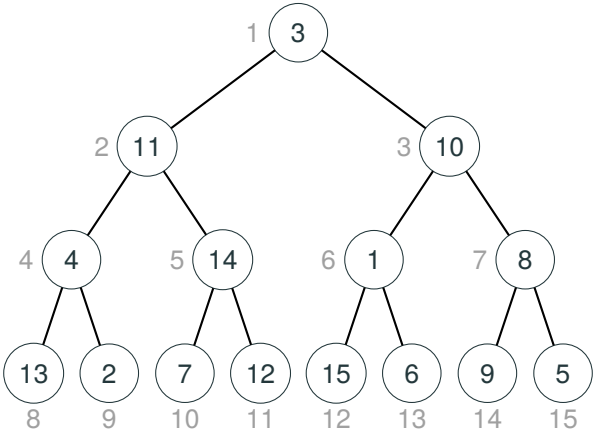
$$A[i] \leq A[Parent(i)].$$

Hierbei bezeichnet $heapsize(A) \leq length(A)$ ein zusätzliches Attribut an A.

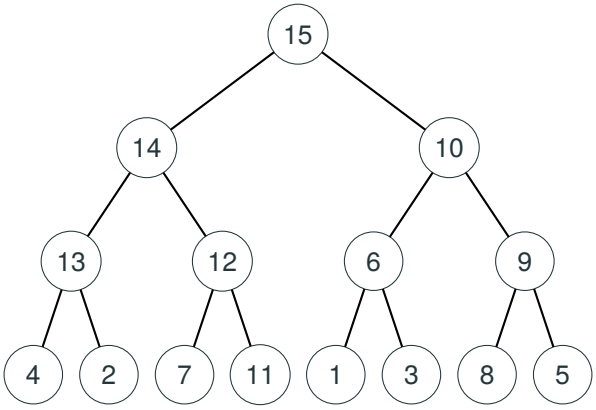
Wichtige Eigenschaft: Ist A eine Heap so gilt $A[1] = \max\{A[i]; 1 \leq i \leq heapsize(A)\}$.

Beispiel Baumstruktur

$A = [3, 11, 10, 4, 14, 1, 8, 13, 2, 7, 12, 15, 6, 9, 5]$



Beispiel Heap



$A = [15, 14, 10, 13, 12, 6, 9, 4, 2, 7, 11, 1, 3, 8, 5]$

Wichtiges Unterprogramm: *Heapify*

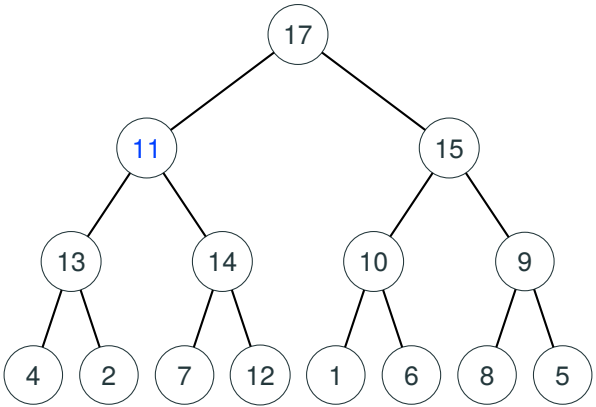
Definition 26

A verletzt die Heap-Eigenschaft in i, falls mindestens eine der folgenden Bedingungen erfüllt ist:

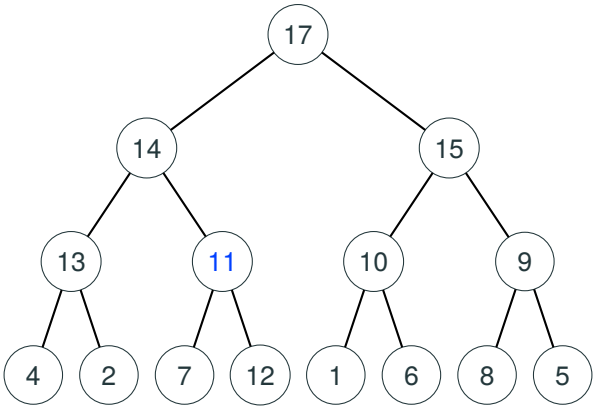
- $Left(i) \leq heapsize(A)$ und $A[Left(i)] > A[i]$
- $Right(i) \leq heapsize(A)$ und $A[Right(i)] > A[i]$.

- $Heapify(A, i)$ ist anwendbar auf A, falls für den Teilbaum mit Wurzel i die **die Heap-Eigenschaft nur in i verletzt** ist.
- $Heapify(A, i)$ stellt die Heap-Eigenschaft im Teilbaum mit Wurzel i wieder her.
- **Idee:** $Heapify(A, i)$ vertauscht das fehlerhaftes $A[i]$ solange mit dem Maximum von $A[Left(i)]$ und $A[Right(i)]$ bis der Fehler beseitigt ist.
- Das erfolgt spätestens nach Erreichen eines Blattes.

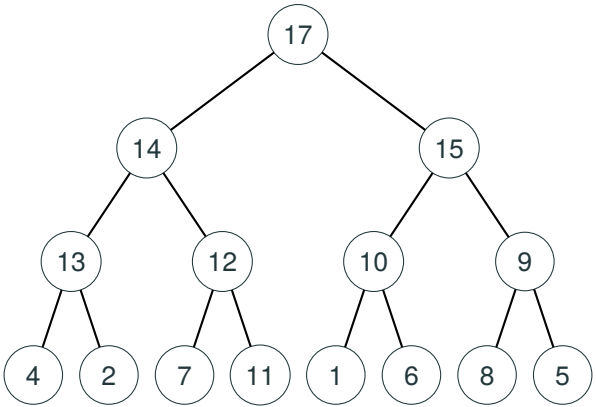
Beispiel *Heapify*$(A, 2)$



Beispiel *Heapify*$(A, 5)$ **in** *Heapify*$(A, 2)$



Beispiel *Heapify*$(A, 2)$ fertig



$Heapify(A, i)$ **(rekursiv)**

1 $max \leftarrow A[i]$, $largest \leftarrow i$

2 **If** $(Left(i) \leq heapsize(A)) \wedge (A[Left(i)] > max)$

3 **then** $largest \leftarrow Left(i)$, $max \leftarrow A[largest]$

4 **If** $(Right(i) \leq heapsize(A)) \wedge (A[Right(i)] > max)$

5 **then** $largest \leftarrow Right(i)$, $max \leftarrow A[largest]$

6 **If** $largest \neq i$

7 **then** $A[largest] \leftarrow A[i]$, $A[i] \leftarrow max$,

8 $Heapify(A, largest)$

Weiteres zur Baumstruktur von $A = (A[1], \cdots, A[n])$

Definition 27

Für Knoten v in einem Binärbaum T bezeichne $height(v, T)$ die **Höhe von v in T**, d.h. die **Länge eines längsten Pfades von v zu einem Blatt in T**.

- Für $i \leq n$ bezeichne $height(i, n)$ die Höhe von Knoten i im Binärbaum zu $\{1, \cdots, n\}$.
- Es gilt $height(i, n) = \max\{k, i \cdot 2^k \leq n\}$.
- Die **Höhe der Wurzel** ist $height(1, n) = \lfloor \log_2(n) \rfloor$.
- **Blätter:** Es gilt $height(i, n) = 0$ für $\lfloor n/2 \rfloor + 1 \leq i \leq n$.
- Für alle k, $0 \leq k \leq \lfloor \log_2(n) \rfloor$ gilt $height(i, n) = k$ für $\lfloor n/2^{k+1} \rfloor + 1 \leq i \leq \lfloor n/2^k \rfloor$.
- Es existieren höchstens $2^{\lfloor \log_2(n) \rfloor - k}$ Zahlen $i \leq n$ mit $height(i, n) = k$.
- Die **Laufzeit** von $Heapify(A, i)$ ist $O(height(i, n)) = O(\log(n))$.

Algorithmus *Buildheap*(A)

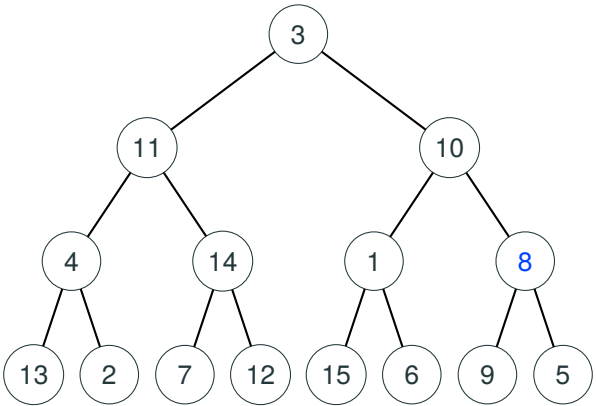
... verwandelt Feld A in einen Heap mit $heapsize(A) = length(A)$.

1 **For** $i \leftarrow \lfloor length(A)/2 \rfloor$ **downto** 1
2 **do** *Heapify*(A, i)
3 $heapsize(A) \leftarrow length(A)$

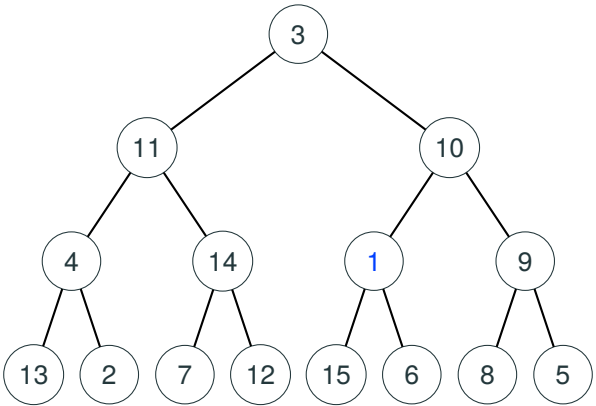
Korrektheit

- *Heapify*(A, i) wird Level für Level in Richtung Wurzel durchgeführt, beginnend mit dem Level direkt oberhalb der Blätter.
- Folglich ist für alle i im Teilbaum mit Wurzel i beim Aufruf von *Heapify*(A, i) die Heap-Eigenschaft nur im Knoten i verletzt.
- Damit ist für alle i der Teilbaum mit Wurzel i nach *Heapify*(A, i) ein Heap.

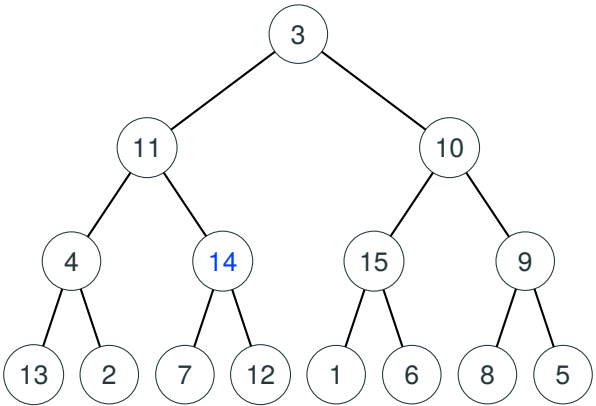
Beispiel Buildheap, *Heapify*$(A, 7)$



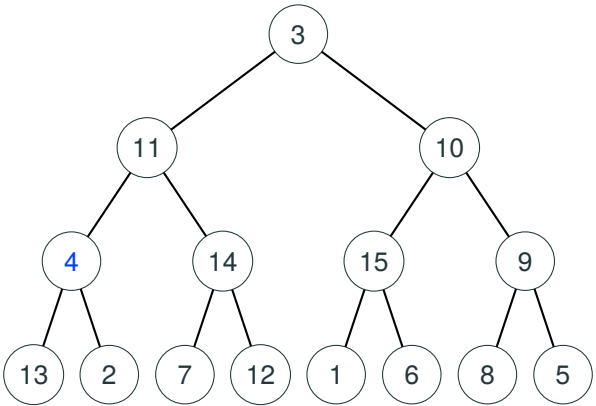
Beispiel Buildheap, *Heapify*$(A, 6)$



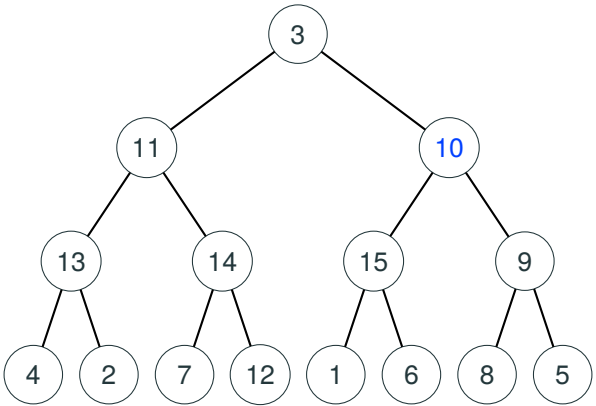
Beispiel Buildheap, *Heapify*$(A, 5)$



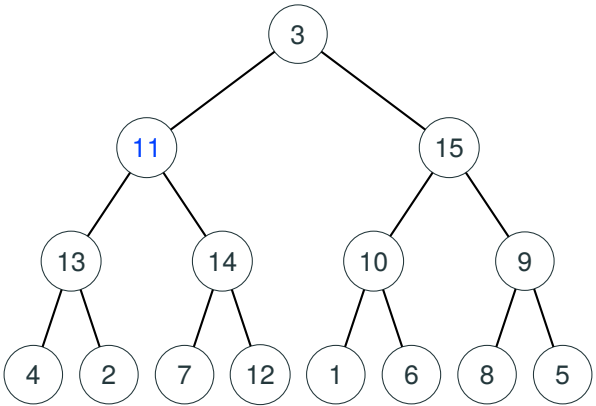
Beispiel Buildheap, *Heapify*$(A, 4)$



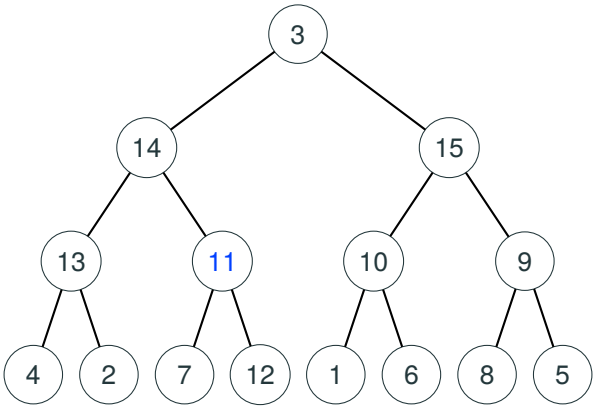
Beispiel Buildheap, *Heapify*$(A, 3)$



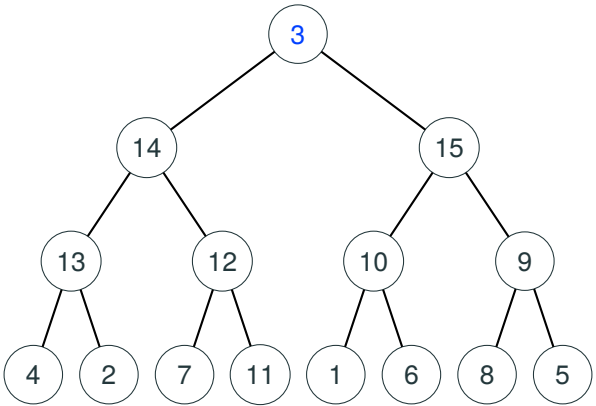
Beispiel Buildheap, *Heapify*$(A, 2)$



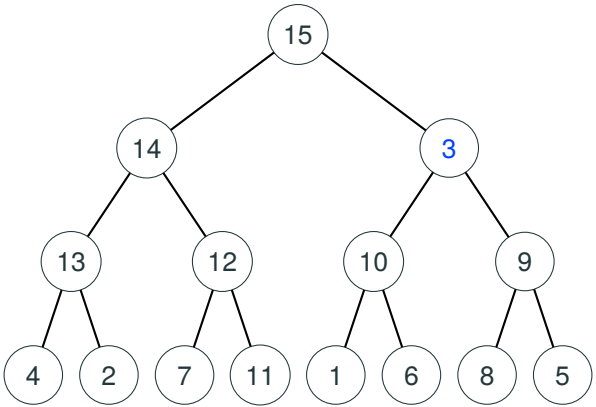
Beispiel Buildheap, *Heapify*$(A, 5)$ in *Heapify*$(A, 2)$



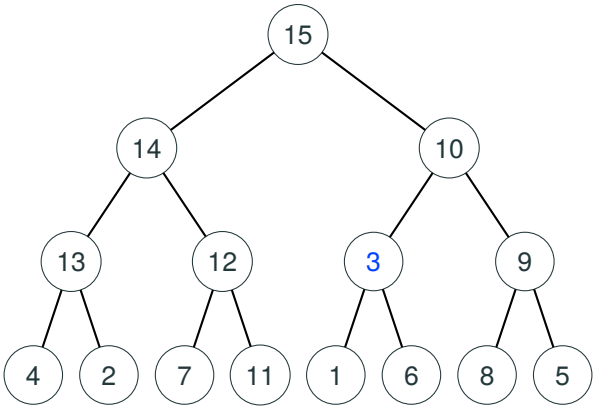
Beispiel Buildheap, *Heapify*$(A, 1)$



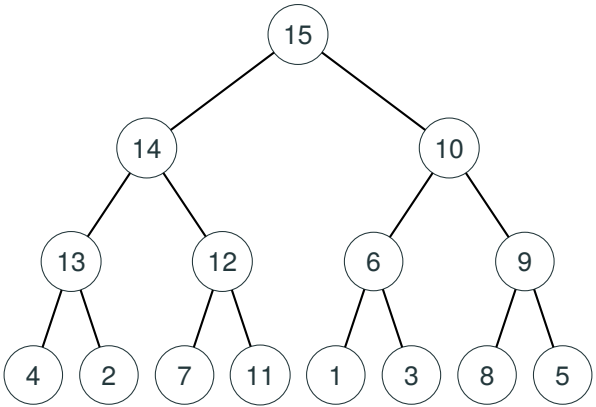
Beispiel Buildheap, *Heapify*$(A, 3)$ in *Heapify*$(A, 1)$



Beispiel Buildheap, *Heapify*$(A, 6)$ in *Heapify*$(A, 3)$ in *Heapify*$(A, 1)$



Beispiel Buildheap, stop



Die Laufzeit von *Buildheap*(A)

- Es bezeichne H_k die Anzahl der Knote der Höhe k.
- Es gilt $H_k = 2^{N-k}$ für $N = \lfloor \log_2(n) \rfloor$.
- Damit ist die Laufzeit von *Buildheap*(A) proportional zu

$$\sum_{k=0}^{N} H_k \cdot k = 2^N \cdot \sum_{k=0}^{N} 2^{-k} \cdot k = \frac{n}{2} \cdot \sum_{k=1}^{N} k \cdot \left(\frac{1}{2}\right)^{k-1} \leq \frac{n}{2} \cdot g(1/2).$$

- Hierbei ist $g(x) = \sum_{k=1}^{\infty} k \cdot x^{k-1}$.
- Es gilt $g(x) = f'(x)$ mit $f(x) = \sum_{k=0}^{\infty} x^k = (1-x)^{-1}$.
- Das heißt, $g(x) = (1-x)^{-2}$,
- Folglich ist $g(1/2) = 4$.
- Also gilt

$$\sum_{k=0}^{N} k \cdot H_k \leq \frac{n}{2} \cdot 4 = 2n = \Theta(n). \; \square$$

Heapsort(A)

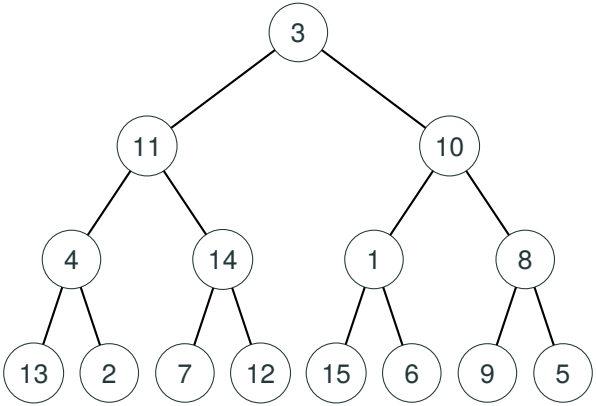
```
1  Buildheap(A)
2  For i ← length(A) downto 2
3     do Exchange(A[1], A[heapsize(A)]),
4        heapsize(A) ← heapsize(A) − 1,
5        Heapify(A, 1)
```

Korrektheit

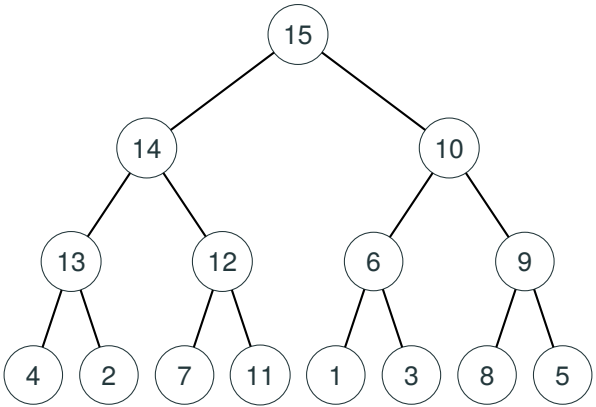
- Nach (1) ist A ein Heap und in $A[1]$ steht das Maximum.
- Nach (3) steht das Maximum in $A[n]$, $n = \mathit{length}(A)$.
- In $A[1 \cdots n-1]$ ist die Heapeigenschaft nur in der Wurzel $A[1]$ verletzt.
- Nach (5) steht in $A[n]$ das Maximum, und $A[1 \cdots n-1]$ ist ein Heap.
- Folglich ist nach jeder Iteration i, $n \geq i \geq 2$, $A[1 \cdots i-1]$ ein Heap und $A[i \cdots n]$ enthält die sortierte Folge der $n - i + 1$ größten Elemente in A.

Beispiel Input *HeapSort*

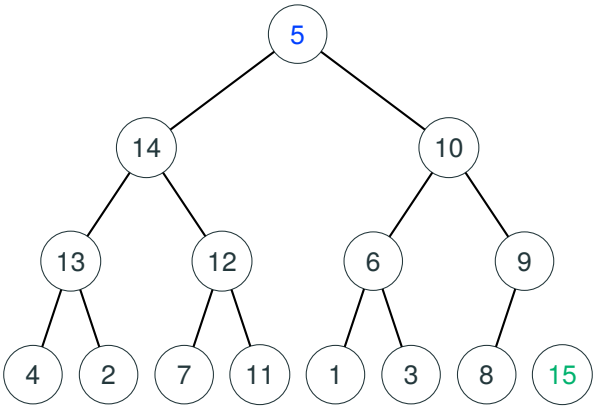
$A = [3, 11, 10, 4, 14, 1, 8, 13, 2, 7, 12, 15, 6, 9, 5]$



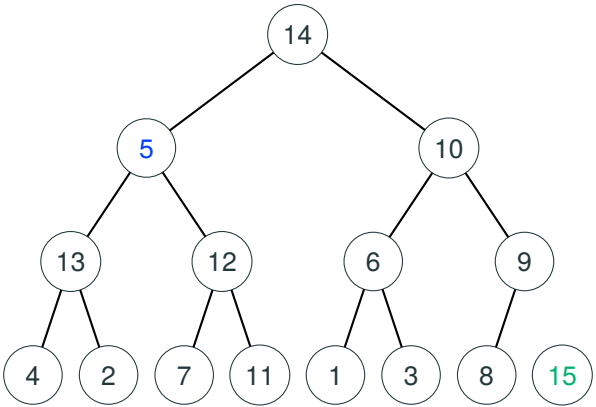
Heap nach *Buildheap*(A) vor Runde $i = 15$



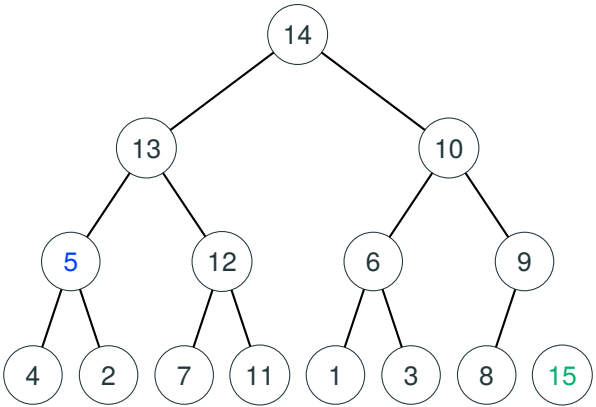
Heapify$(A, 1)$ **in Runde** $i = 15$



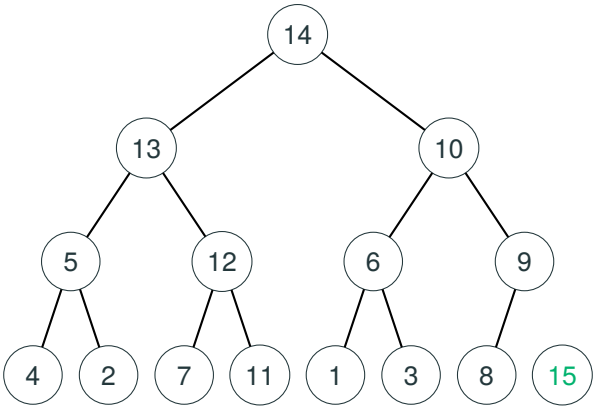
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 15$



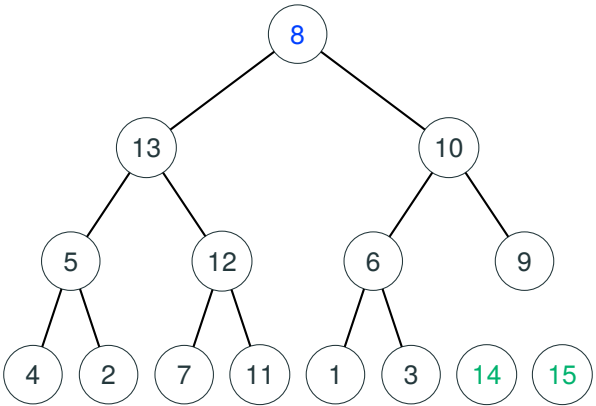
Heapify$(A, 4)$ **in** *Heapify*$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 15$



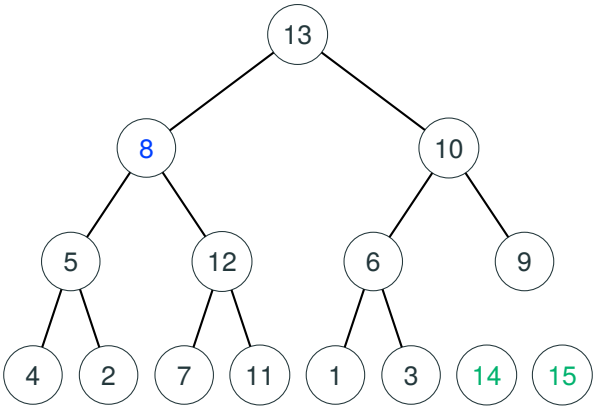
Heap vor $i = 14$



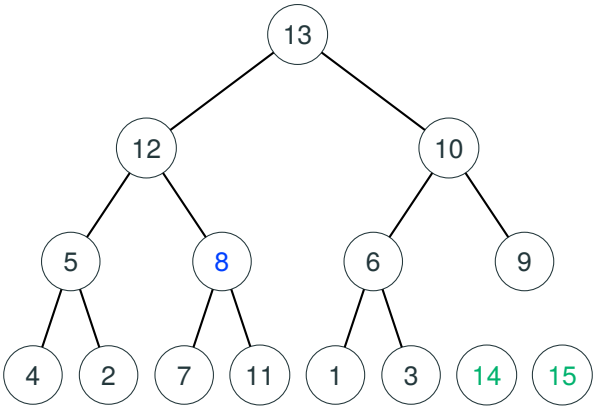
Heapify$(A, 1)$ **in Runde** $i = 14$



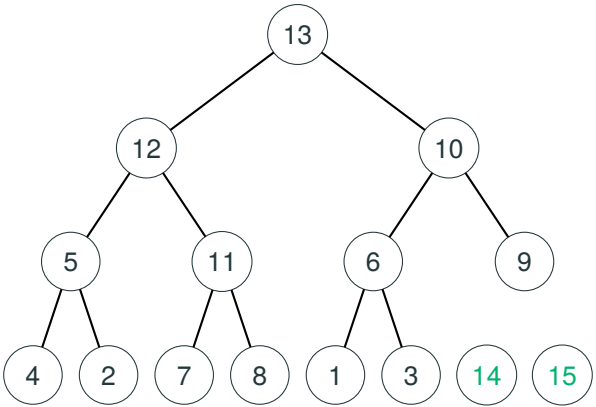
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 14$



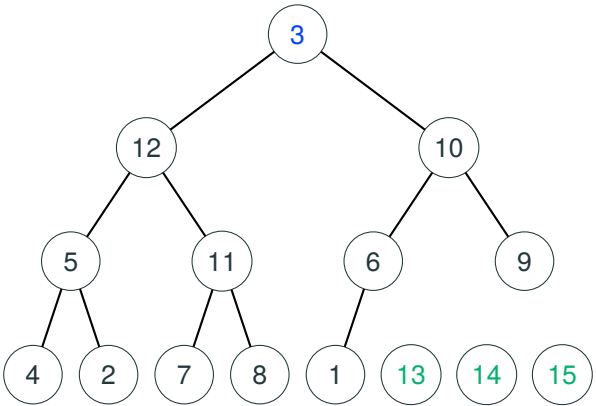
Heapify$(A, 5)$ **in** *Heapify*$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 14$



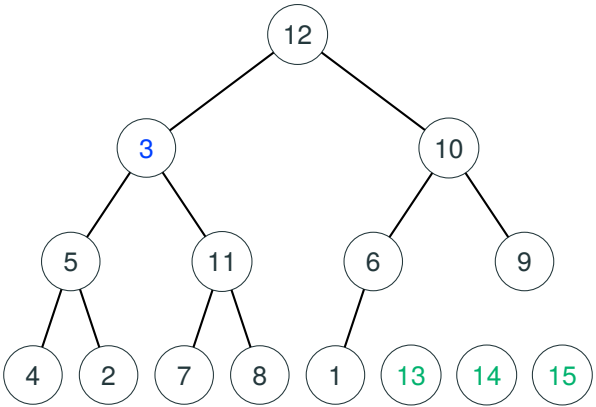
Heap vor Runde $i = 13$



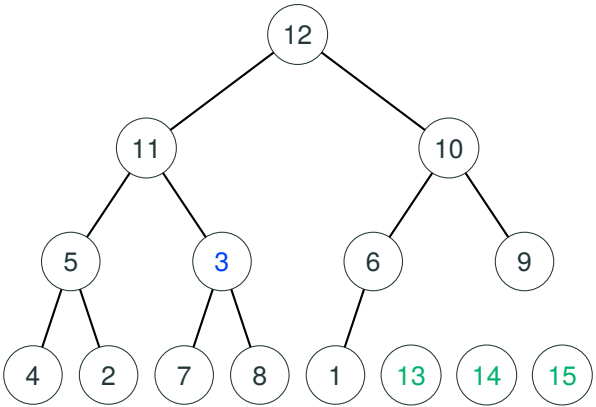
Heapify$(A, 1)$ **in Runde** $i = 13$



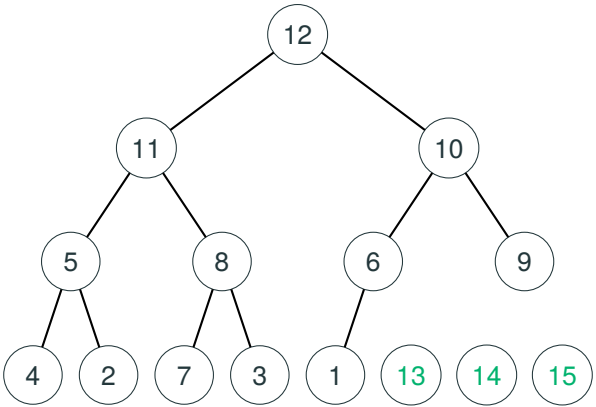
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 13$



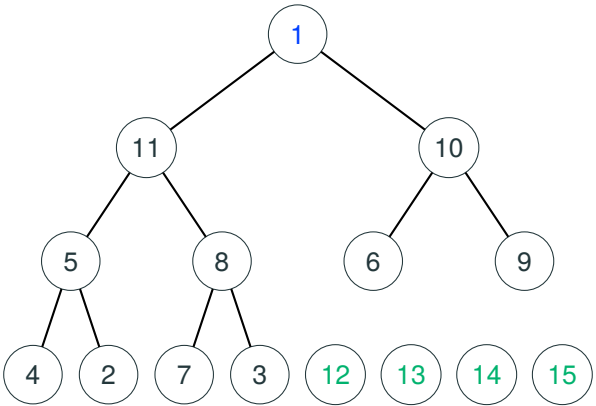
Heapify$(A, 5)$ **in** *Heapify*$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 13$



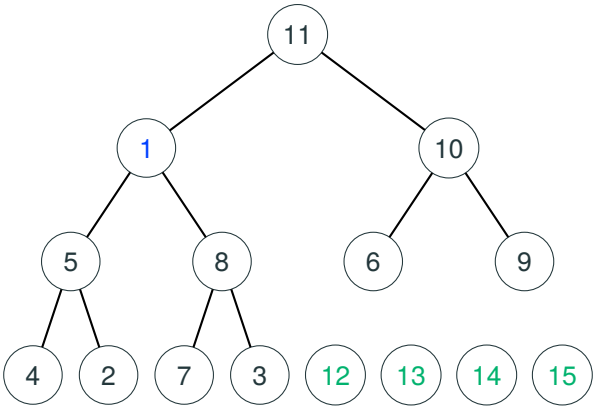
Heap vor Runde $i = 12$



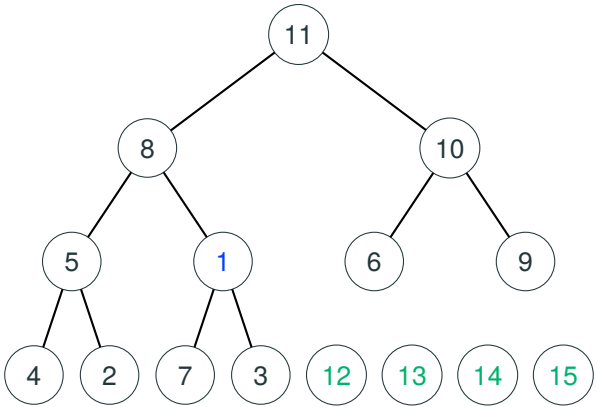
Heapify$(A, 1)$ **in Runde** $i = 12$



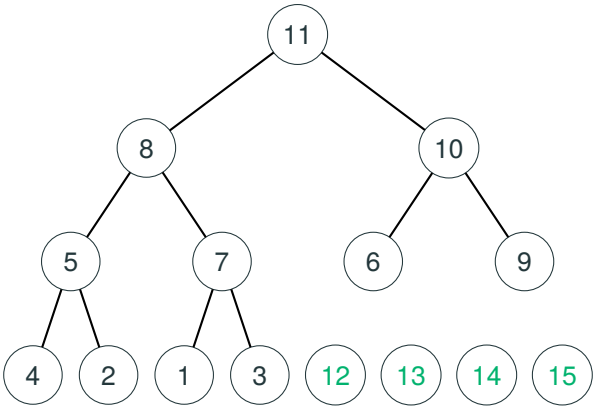
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 12$



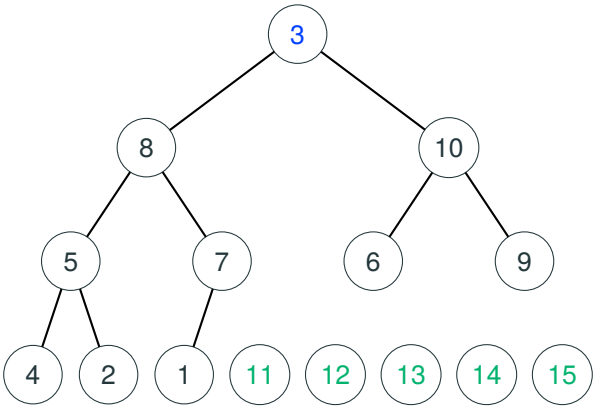
Heapify$(A, 5)$ **in** *Heapify*$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 12$



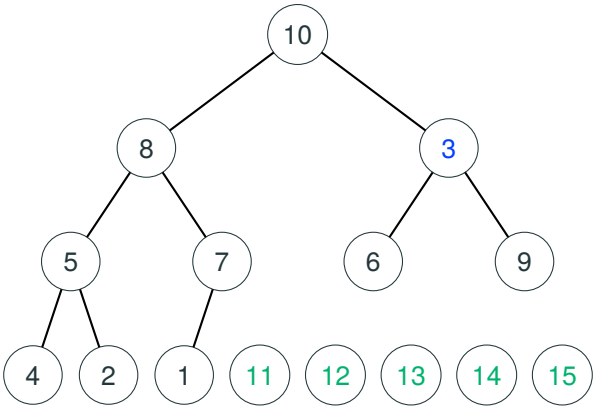
Heap vor Runde $i = 11$



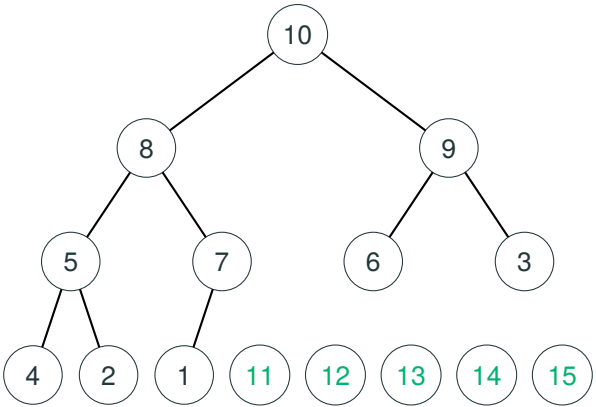
Heapify$(A, 1)$ **in Runde** $i = 11$



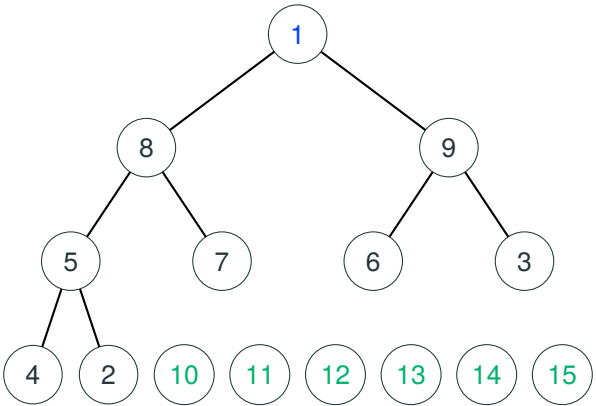
Heapify$(A, 3)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 11$



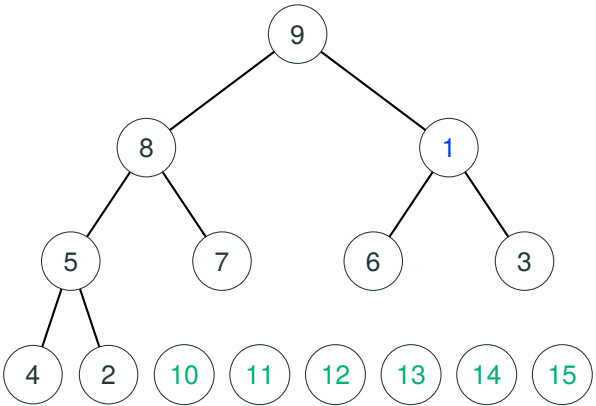
Heap vor Runde $i = 10$



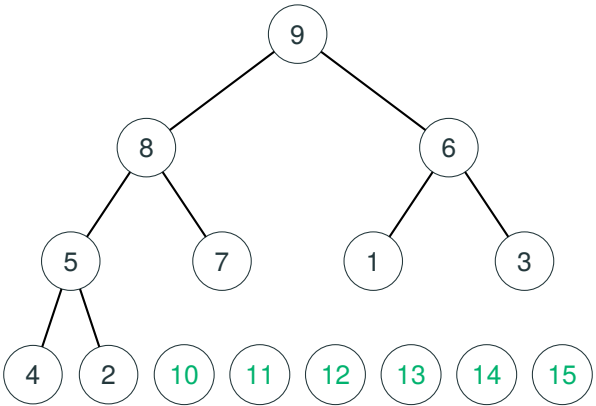
Heapify$(A, 1)$ **in Runde** $i = 10$



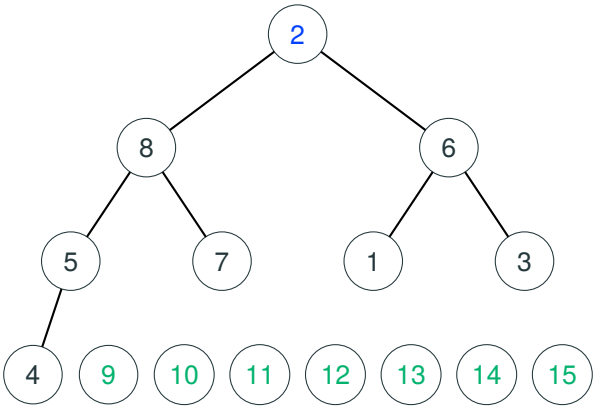
Heapify$(A, 3)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 10$



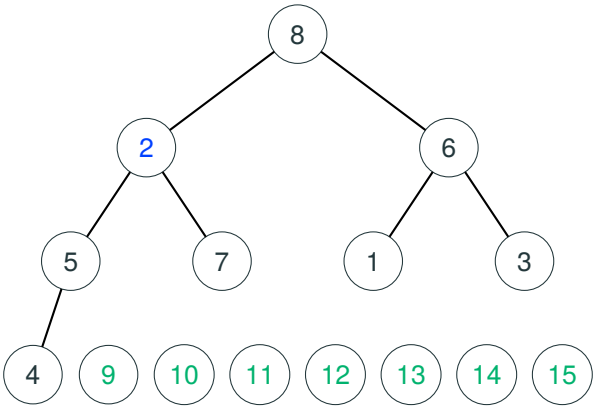
Heap vor Runde $i = 9$



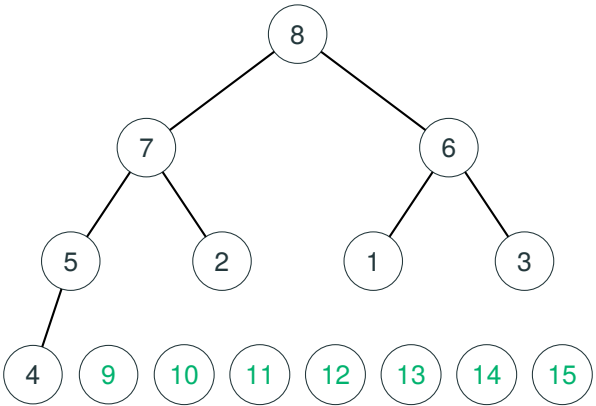
Heapify$(A, 1)$ **in Runde** $i = 9$



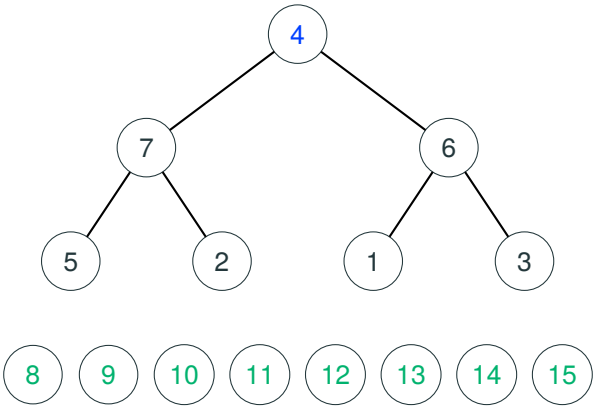
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 9$



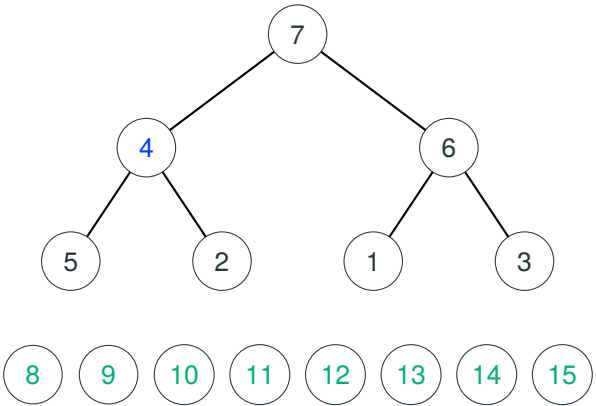
Heap vor Runde $i = 8$



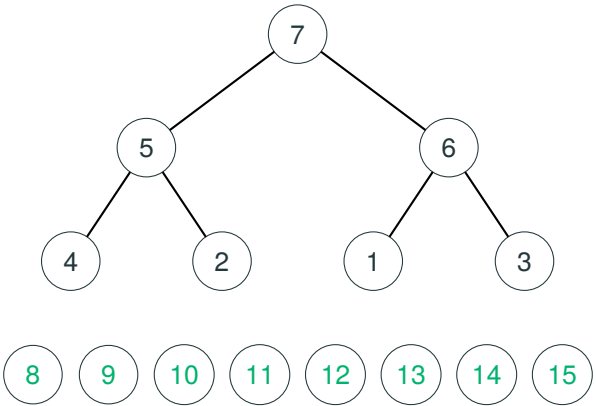
Heapify$(A, 1)$ **in Runde** $i = 8$



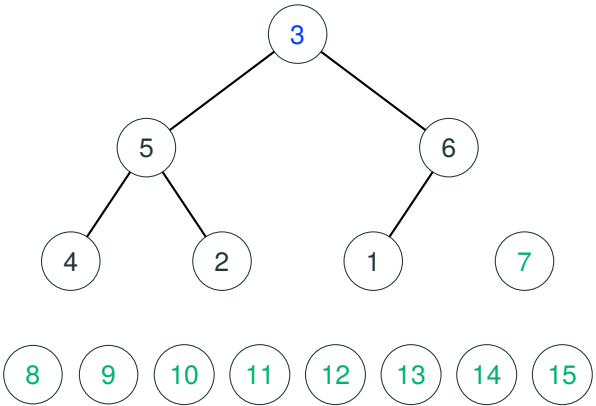
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 8$



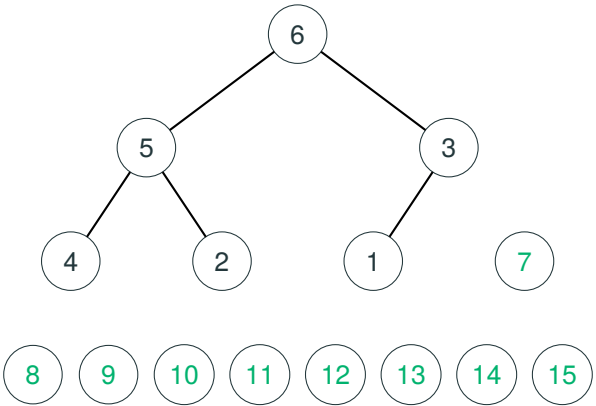
Heap vor Runde $i = 7$



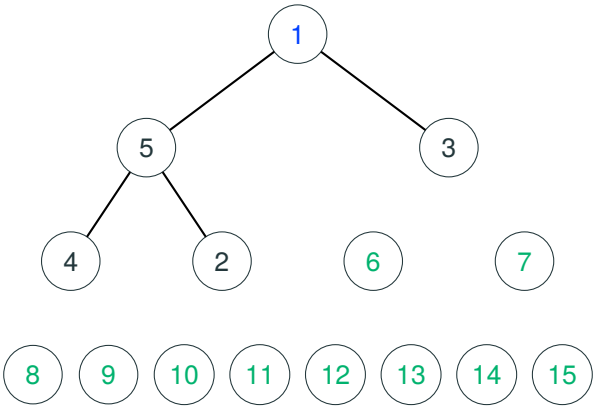
Heapify$(A, 1)$ **in Runde** $i = 7$



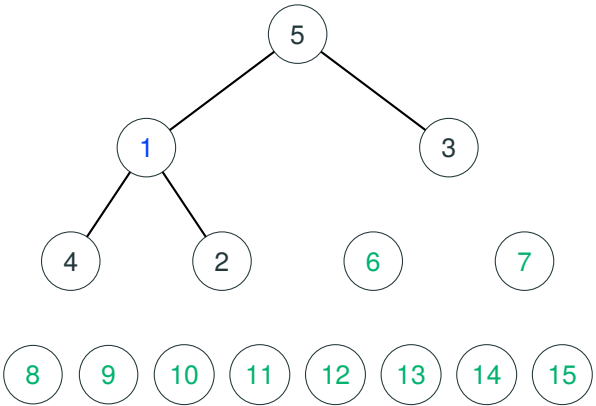
Heap vor Runde $i = 6$



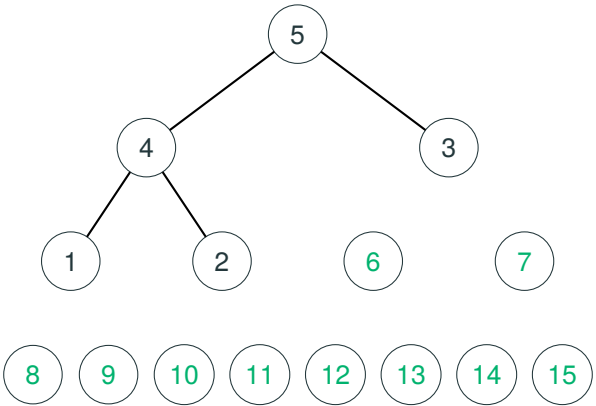
Heapify$(A, 1)$ **in Runde** $i = 6$



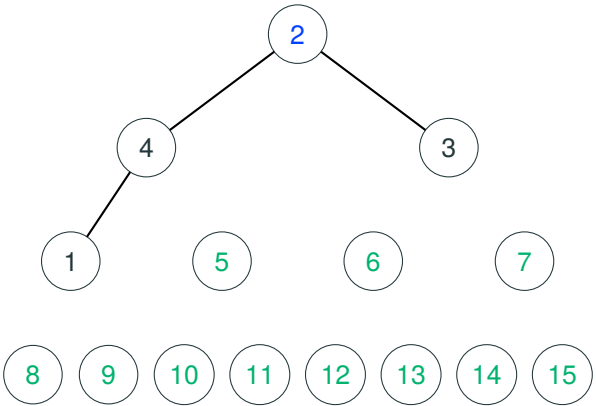
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$ **in Runde** $i = 6$



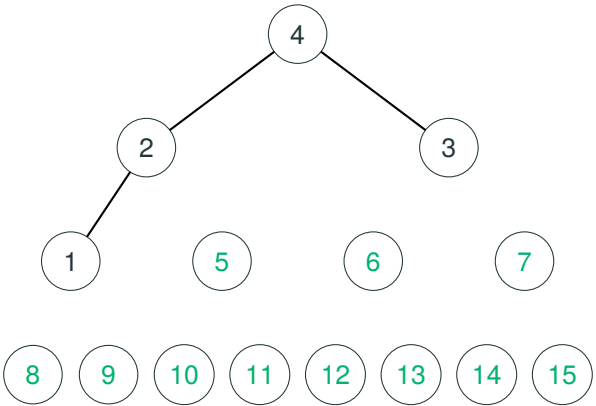
Heap vor Runde $i = 5$



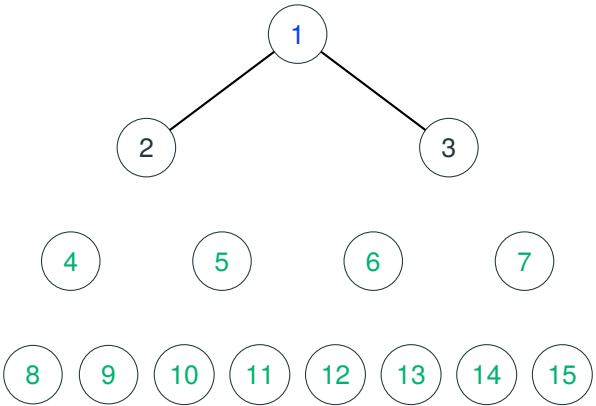
Heapify$(A, 1)$ **in Runde** $i = 5$



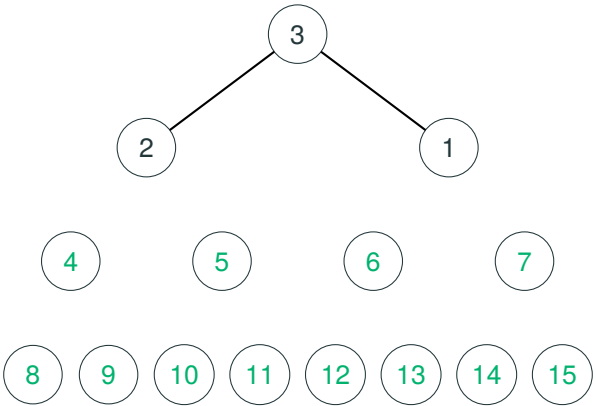
Heap vor Runde $i = 4$



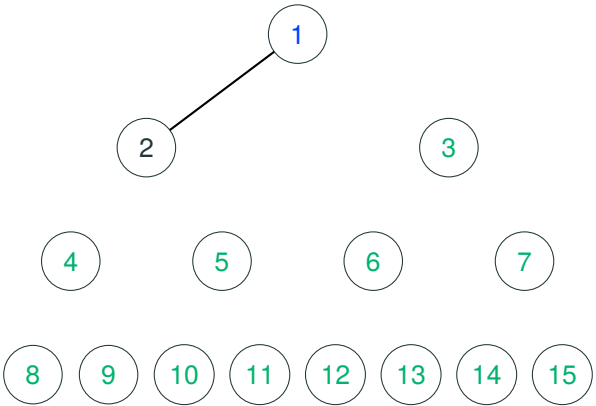
Heapify$(A, 1)$ **in Runde** $i = 4$



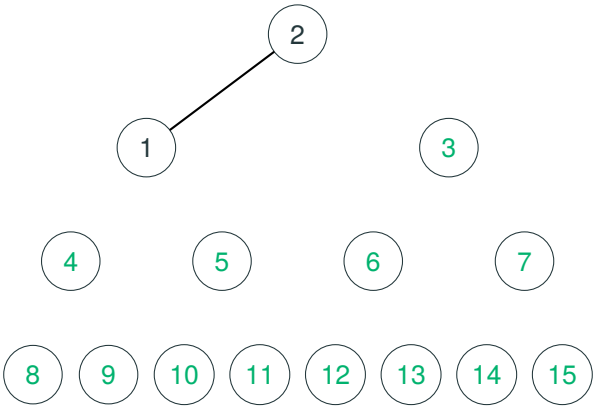
Heap vor Runde $i = 3$



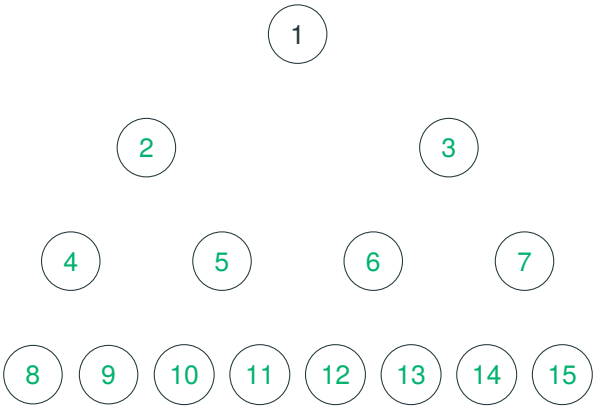
Heapify$(A, 1)$ **in Runde** $i = 3$



Heap vor Runde $i = 2$



Stop in Runde $i = 2$



Die Laufzeit von Heapsort(A) ist $\Theta(n \cdot \log(n))$

(1) *Buildheap*(A)
(2) **For** $i \leftarrow$ *length*(A) **downto** 2
(3) **do** *Exchange*$(A[1], A[$*heapsize*$(A)])$,
(4) *heapsize*$(A) \leftarrow$ *heapsize*$(A) - 1$,
(5) *Heapify*$(A, 1)$

Laufzeit

- Die Laufzeit von Buildheap ist $\Theta(n)$.
- Die Gesamtlaufzeit der For-Schleife ist $\sum_{i=1}^{n-1} \log_2(i) = \Theta\left(\sum_{i=1}^{n-1} \ln(i)\right)$.
- Wir zeigen mittels der **Summationsmethode**, dass

$$\sum_{i=1}^{n-1} \ln(i) = \Theta(n \log(n)).$$

Einschub: Summationsmethode

Theorem 28

Für $a, b \in \mathbb{N}$, $a < b - 1$, und $f : [a, b] \longrightarrow \mathbb{R}^+$ monoton wachsend und integrierbar gilt

$$\int_{x=a}^{b-1} f(x)dx \;\leq\; \sum_{i=a}^{b-1} f(i) \;\leq\; \int_{x=a}^{b} f(x)dx.$$

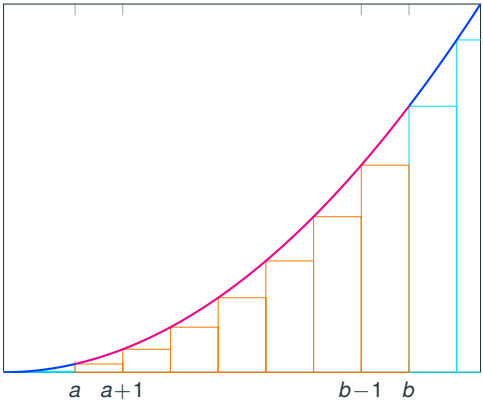
Aus Theorem 28 folgt

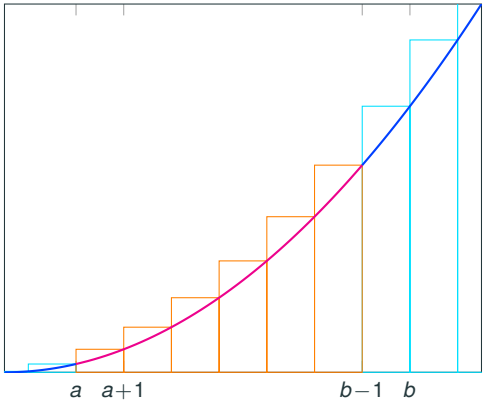
$$(n-1)\cdot(\ln(n-1)-1)+1 \leq \sum_{i=1}^{n-1} \ln(i) \leq n\cdot(\ln(n)-1)+1.$$

Das gilt, da $F(x) = x \cdot (\ln(x) - 1)$ die Stammfunktion von $\ln(x)$, und $F(1) = -1$ ist.

Folglich ist $\sum_{i=1}^{n-1} \ln(i) = \Theta(n \cdot \log(n))$. □

Der Beweis von Theorem 28





$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x)\,dx$$

$$\int_a^{b-1} f(x)\,dx \leq \sum_{i=a}^{b-1} f(i)$$

Sortieralgorithmen

Priority Queues

Priority Queues

... verwalten dynamisch eine Menge A von Elementen (Jobs) i, denen Prioritäten $A[i]$ zugeordnet sind, bezüglich der Operationen

- $Max(A)$: gibt den Job mit maximaler Priorität aus,
- $ExtractMax(A)$: entfernt den Job mit maximaler Priorität aus A,
- $IncreaseKey(A, i, key)$: erhöht die Priorität $A[i]$ von i auf den Wert key,
- $Insert(A, key)$. Fügt zu A ein neues Element mit Priorität key hinzu.

Wir realisieren A durch einen **Heap**, wobei $heapsize(A)$ der aktuellen Anzahl der noch zu erledigenden Jobs in der Queue entspricht.

Alle Operationen setzen die Heapeigenschaft voraus und erhalten diese.

Priority Queue Operationen *Max* und *ExtractMax*

Max(A)

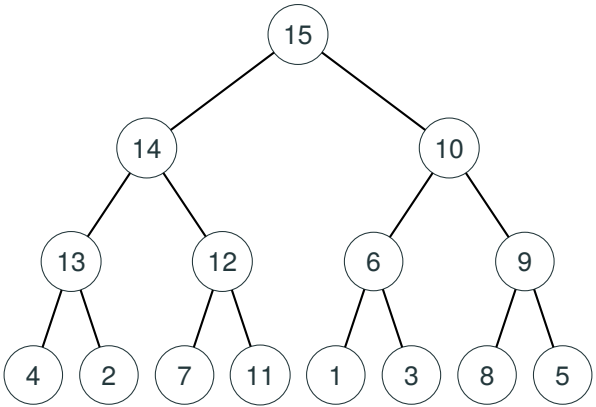
(1) return $A[1]$

ExtractMax(A)

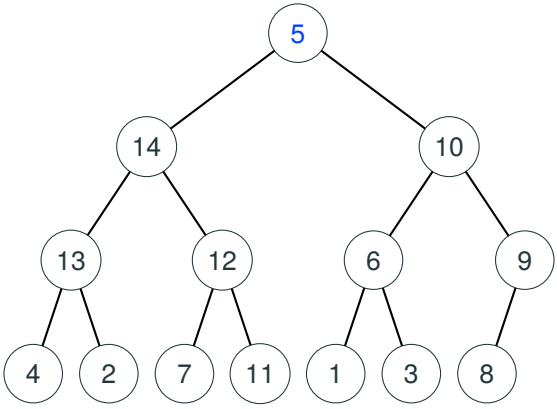
(1) $max \leftarrow A[1]$
(2) $A[1] \leftarrow A[heapsize(A)]$
(3) $heapsize(A)] \leftarrow heapsize(A) - 1$
(4) $Heapify(A, 1)$
(5) **return** *max*

Die Laufzeit von *Max*(A) ist $O(1)$, die von *ExtractMax*(A) ist $\Theta(\log(n))$.

Priority Queue A als Heap

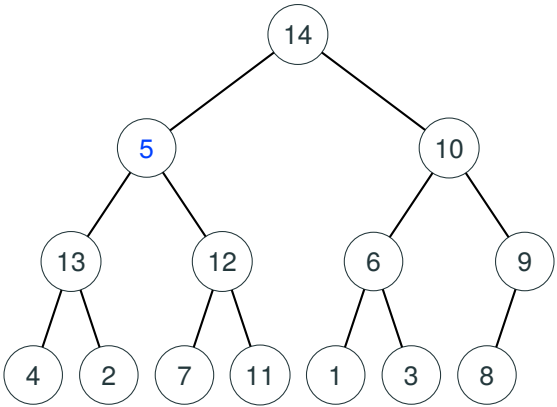


Heapify$(A, 1)$

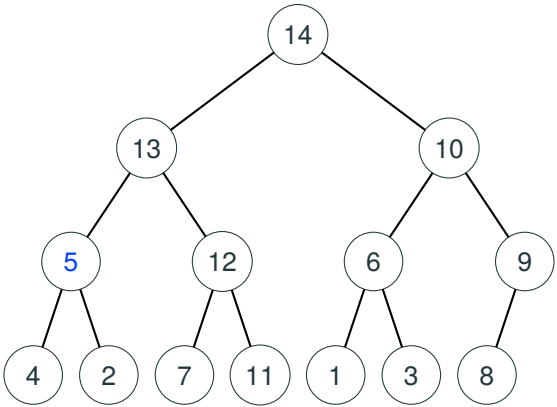


$max = 15$

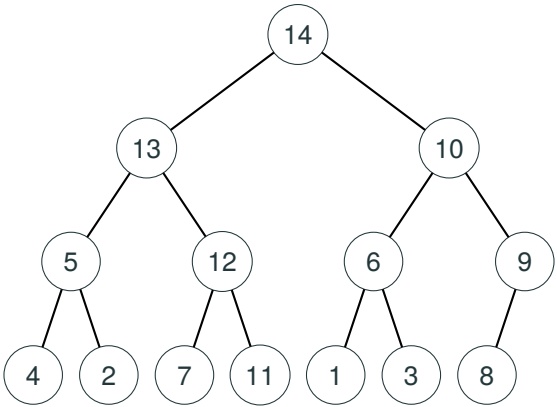
Heapify$(A, 2)$ **in** *Heapify*$(A, 1)$



Heapify$(A, 4)$ **in** *Heapify*$(A, 2)$ **in** *Heapify*$(A, 1)$



Neue Queue



Priority Queue Operationen *IncreaseKey* und *InsertKey*

IncreaseKey(A, i, key)

```
1 if (i ≤ heapsize(A)) ∧ (A[i] < key)
2    then A[i] ← key
3         while (i > 1) ∧ (A[i] > A[Parent(i)])
4            do Exchange(A[i], A[Parent(i)])
5               i ← Parent(i)
```

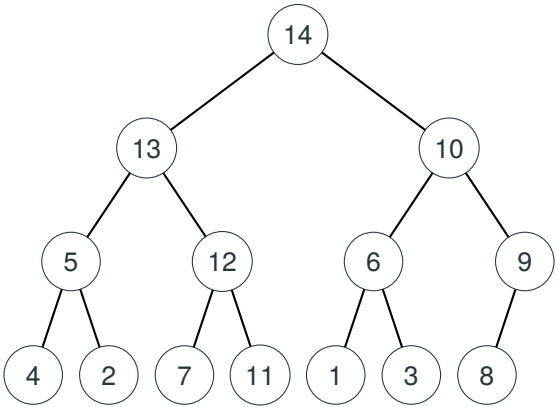
InsertKey(A, key)

```
1 heapsize(A) ← heapsize(A) + 1
2 A[heapsize(A)] ← −∞
3 IncreaseKey(A, heapsize(A), key)
```

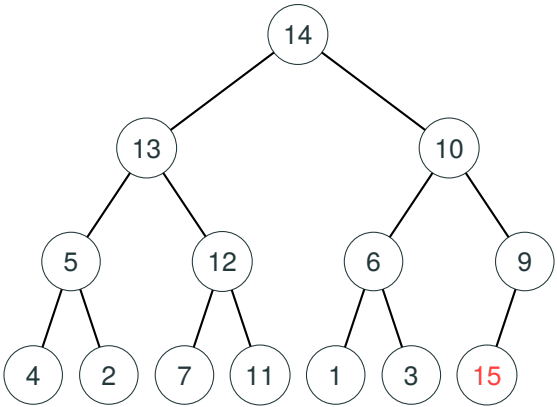
Die **Laufzeit** von *IncreaseKey* und *InsertKey*(A, key) ist $\Theta(\log(n))$.

Korrektheit: Durch *Exchange*$(A[i], A[Parent(i)])$ wird die Heap-Eigenschaft in *Parent*(i) wieder hergestellt, allerdings ggf. in $A[Parent(Parent(i))]$ verletzt.

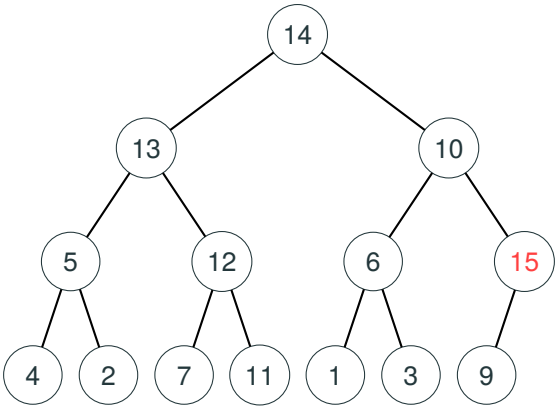
Priority Queue A als Heap



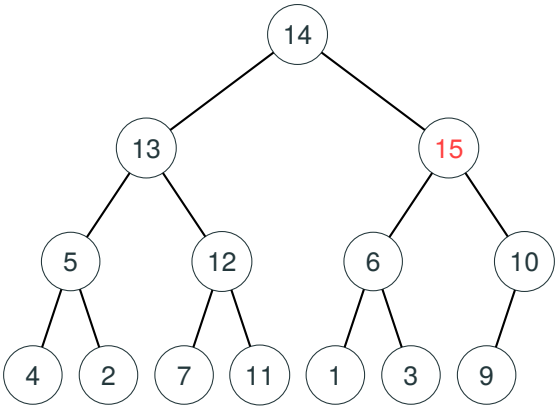
InsertKey(A, i, key) **für** $i = 14$**,** $key = 15$



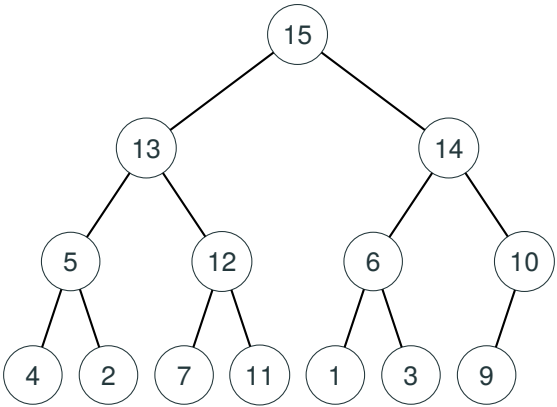
Exchange($A[14], A[7]$)



Exchange($A[7], A[3]$)



Exchange($A[3], A[1]$), **neue Queue**



Sortieralgorithmen

Quicksort

Quicksort $\cdots$

$\cdots$ ist ein rekursives vergleichbasiertes In-Place Sortierverfahren der worst-case Laufzeit $\Theta(n^2)$, das auf Grund seiner geringen average case Laufzeit $c \cdot n \cdot \log_2(n)$, c klein, praktisch sehr bedeutsam ist.

Wir setzen ab jetzt voraus, dass alle Einträge im Eingabefeld A verschieden sind.

Definition 29

Für Eingabefelder A und Indizees i, $1 \leq i \leq \mathit{length}(A)$, gibt

$$\mathit{Rang}_A(A[i]) = |\{k, 1 \leq k \leq \mathit{length}(A), A[k] < A[i]\}| + 1$$

die Position von $A[i]$ im sortierten Feld an.

Beispiel: $A = [27, 91, 18, 7, 121, 44, 13]$, $\mathit{Rang}_A(A[6]) = \mathit{Rang}_A(44) = 5$.

Der Algorithmus Quicksort

$\underline{Quicksort(A,p,r)}$ (für $1 \leq p \leq r \leq length(A)$)

(1) **If** $p < r$
(2) **then** $q = Partition(A,p,r)$
(3) $Quicksort(A,p,q-1)$
(4) $Quicksort(A,q+1,r)$

Erklärung: $Partition(A,p,r)$ wählt das **Pivotelement** $x = A[r]$, berechnet $q = Rang(A[r])$ und permutiert A so, dass die folgenden Bedingungen gelten:

- $A[q] = x$, d.h. das Pivotelement x steht an der richtigen Position.
- Falls $p < q$ so $A[i] < x$ für $i = p, \cdots, q-1$.
- Falls $q < r$ so $A[i] > x$ für $i = q+1, \cdots, r$.

Beispiel:

$[91,18,7,121,44,13,31] \rightarrow [91,18,7,121,44,13,31] \rightarrow [18,7,13,31,91,121,44]$

Beispiel Quicksort

[48, 73, 38, 68, 18, 23, 36, 46, 34, 20, 70, 58, 29, 41]

[38, 18, 23, 36, 34, 20, 29] [41] [48, 73, 68, 46, 70, 58, 61]

[18, 23, 20] [29] [38, 36, 34] [41] [48, 46, 58] [61] [73, 68, 70]

[18] [20] [23] [29] [34] [38, 36] [41] [48, 46] [58] [61] [68] [70] [73]

[18] [20] [23] [29] [34] [36] [38] [41] [46] [48] [58] [61] [68] [70] [73]

[18] [20] [23] [29] [34] [36] [38] [41] [46] [48] [58] [61] [68] [70] [73]

Partition(A, p, r)

(1) $x \leftarrow A[r]$
(2) $i \leftarrow p - 1$
(3) **For** $j \leftarrow p$ **to** r
(4) **do if** $A[j] \leq x$
(5) **then** $i \leftarrow i + 1$
(6) **if** $i < j$ **then**
(7) *exchange*$(A[i], A[j])$
(8) **Output** i

Beispiel

$A[p \cdots r] = [18, 91, 7, 121, 44, 13, 31]$

$x, i, j, i = j$

$[*, 18, 91, 7, 121, 44, 13, 31]$
$[*, 18, 91, 7, 121, 44, 13, 31]$
$[*, 18, 91, 7, 121, 44, 13, 31]$
$[*, 18, 91, 7, 121, 44, 13, 31]$
$[*, 18, 91, 7, 121, 44, 13, 31]$
$[*, 18, 7, 91, 121, 44, 13, 31]$
$[*, 18, 7, 91, 121, 44, 13, 31]$
$[*, 18, 7, 91, 121, 44, 13, 31]$
$[*, 18, 7, 91, 121, 44, 13, 31]$
$[*, 18, 7, 91, 121, 44, 13, 31]$
$[*, 18, 7, 13, 121, 44, 91, 31]$
$[*, 18, 7, 13, 121, 44, 91, 31]$
$[*, 18, 7, 13, 121, 44, 91, 31]$
$[*, 18, 7, 13, 31, 44, 91, 121]$

Ausgabe $i = p + 3$

Korrektheit Partition

Theorem 30

Für alle $j = p, \cdots, r-1$ gilt nach Iteration j, dass

(a) *wenn $i = j$ dann $A[k] < x$ für $k = p, \cdots, i = j$,*
$A = [<, <, \cdots, <, <, *, \cdots, *, x]$.

(b) *wenn $i < j$ dann $A[k] < x$ für $k = p, \cdots, i$ und $A[k] > x$ für $k = i+1, \cdots, j$,*
$A = [<, <, \cdots, <, <, >, \cdots, >, >, *, \cdots, *, x]$.

Beweis per Induktion über j:

Theorem 30 gilt nach Iteration p, da entweder $A[p] < x$ und damit $i = j = p$, d.h.,

$A = [<, *, \cdots, *, x]$ (Fall (a)),

oder $A[p] > x$ und damit $i = p - 1$ und

$A = [>, *, \cdots, *, x]$ (Fall (b)).

Der Beweis von Theorem 30

Induktionsschritt: Betrachten $j,\ p \leq j \leq r-2$, setzen voraus, dass Theorem 30 nach Iteration j wahr ist, und zeigen, dass Theorem 30 auch nach Iteration $j+1$ wahr ist.

- **Fall 1:** Nach Iteration j gilt $i = j$, d.h., $A = [<, \cdots, <, <, *, *, \cdots, *, x])$.
  - **Fall 1.1:** $A[j+1] < x$.

    Dann gilt $i = j+1$ nach Iteration $j+1$, d.h., $A = [<, \cdots, <, <, <, *, \cdots, *, x]$.
  - **Fall 1.2:** $A[j+1] > x$.

    Dann $i = j < j+1$ nach Iteration $j+1$, d.h., $A = [<, <, \cdots, <, <, >, *, \cdots, *, x]$.
- **Fall 2:** Nach Iteration j gilt $i < j$, $A = [<, <, \cdots, <, <, z, >, \cdots, >, >, y, *, \cdots, *, x]$.
  - **Fall 2.1:** $A[j+1] = y < x$. Es gilt $z > x$.

    Während Iteration $j+1$ wird i auf $i+1$ gesetzt und y und z vertauscht.

    $A = [<, <, \cdots, <, <, y, >, \cdots, >, >, z, *, \cdots, *, x]$.
  - **Fall 2.2:** $A[j+1] = y > x$.

    In Iteration $j+1$ werden i and A nicht verändert, d.h.

    $A = [<, <, \cdots, <, <, z, >, \cdots, >, >, y, *, \cdots, *, x]$. □

Die Situation in und nach Iteration r

Fall 1: Nach Iteration $r-1$ gilt $i = j = r-1$, d.h., $A = [<, <, \cdots, <, <, x]$.

Da $A[r] = x$ wird i zu Beginn von Iteration r auf $i+1 = r = j$ gesetzt,

$[<, \cdots, <, <, x] \;\rightarrow\; [<, \cdots, <, <, x]$.

Fall 2: Nach Iteration $r-1$ gilt $i < j = r-1$, $A = [<, \cdots, <, <, z, >, \cdots, >, >, x]$.

Da $A[r] = x$ wird i zu Beginn von Iteration r auf $i+1 < r = j$ gesetzt, d.h.,

$A = [<, \cdots, <, <, z, > \cdots, >, x]$.

Dann wird $\mathit{Exchange}(A[i], A[r])$ ausgeführt, d.h.,

$A = [<, \cdots, <, <, x, >, \cdots, >, z]$.

Da $z > x$ gilt nach Iteration r, dass $i = \mathit{Rang}(x)$ und $A[i] = x$, d.h. Partition ist korrekt. $\square$

Die Worst Case Laufzeit $T(n)$ von Quicksort

Es sei $n = \mathit{length}(A)$ und $A[1], \cdots, A[n]$ untereinander verschieden.

Wir messen die Laufzeit in der Zahl der Vergleiche.

- **Beobachtung 1:** $T(0) = T(1) = 0$.
- **Beobachtung 2:** Die Laufzeit von $\mathit{Partition}(A, p, r)$ ist gleich $r - p + 1$.
- **Beobachtung 3:** Während $\mathit{Quicksort}(A, 1, n)$ wird Partition höchstens n Mal aufgerufen, da sich nach jedem Aufruf die Anzahl der noch zu sortierenden Elemente um eins reduziert, d.h.,

$$T(n) \leq \sum_{i=1}^{n} n - i + 1 = \frac{n(n+1)}{2} = \Theta(n^2).$$

Worst Case Beispiel: Sortierte Folge, Pivotelement ist stets das Maximum, d.h.,

$$T(n) = T(n-1) + T(0) + n = T(n-1) + n = \cdots = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \Theta(n^2).$$

Quicksort: Best Case Beispiel, Rang von Elementen

... bei jedem Aufruf von Partition steht das **mittelstgröße Element ganz rechts**. Dann gilt

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n,$$

wobei $\frac{n}{2} \in \{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil\}$.

Das Master-Theorem liefert $T(n) = \Theta(n \cdot \log(n))$.

Beispiel

$$[48, 27, 63, 92, 39, 74, 51]$$

$$[48, 27, 39]\ [51]\ [63, 92, 74]$$

$$[27]\ [39]\ [48]\ [58]\ [63]\ [74]\ [92]$$

Quicksort: Analyse der Average Case Laufzeit $\mathbf{E}[T(n)]$

- **Fakt:** Für alle Belegungen von $A[1], \cdots, A[n]$ existiert genau eine sortierende Permutation π, so dass $A[\pi(1)] < \cdots < A[\pi(n)]$
- **Modellannahme:** Die Verteilung auf den Eingaben A ist so, dass die sortierenden Permutationenen mit Gleichverteilung auftreten.
- Das bedingt, dass $Pr[Rang(A[n]) = k] = 1/n$ für alle $k = 1, \cdots, n$.

Lemma 31

$\mathbf{E}[T(n)] \leq 2 \cdot n \cdot \ln(n)$.

Beweis: $\mathbf{E}[T(n)] = \sum_{k=1}^{n} Pr[Rang(A[n]) = k] \cdot \mathbf{E}[T(n)|Rang(A[n]) = k]$

$$= \frac{1}{n} \cdot \sum_{k=1}^{n} \mathbf{E}[T(k-1)] + \mathbf{E}[T(n-k)] + n = \frac{2}{n} \cdot \sum_{k=2}^{n-1} \mathbf{E}[T(k)] + n$$

da $T(1) = T(0) = 0$.

Fortsetzung

Wir haben gezeigt, dass $\mathbf{E}[T(n)] = \frac{2}{n} \cdot \sum_{k=2}^{n-1} \mathbf{E}[T(k)] + n$.

Wir zeigen nun per Induktion, dass $\mathbf{E}[T(n)] \leq 2 \cdot n \cdot \ln(n)$.

Für $n = 1$ gilt $\mathbf{E}[T(n)] = 0 \leq 2 \cdot 1 \cdot \ln(1) = 0$.

Laut Induktionsvoraussetzung gilt

$$\mathbf{E}[T(n)] \leq \frac{4}{n} \sum_{k=2}^{n-1} k \cdot \ln(k) + n.$$

Da $F(x) = \frac{1}{2}\left(x^2 \ln(x) - \frac{1}{2}x^2\right)$ die Stammfunktion von $x \cdot \ln(x)$ ist, gilt:

$$\sum_{k=2}^{n-1} k \cdot \ln(k) \leq \int_{x=2}^{n} x \cdot \ln(x) dx = \frac{1}{2}\left(n^2 \ln(n) - \frac{1}{2}n^2\right) - F(2).$$

Also gilt

$$\mathbf{E}[T(n)] \leq 2\left(n \cdot \ln(n) - \frac{1}{2}n\right) + n = 2 \cdot n \cdot \ln(n). \; \square$$

Sortieralgorithmen

Vergleichsbasiertes Sortieren versus Sortieren in Linearzeit

Eine untere Laufzeitschranke für vergleichsbasierte Sortieralgorithmen

Wir ordnen jedem vergleichsbasierten Sortieralgorithmen *Sort* und jeder Eingabelänge n einen binären Entscheidungsbaum T_n mit folgenden Knoten-/Kantenmarkierungen zu:

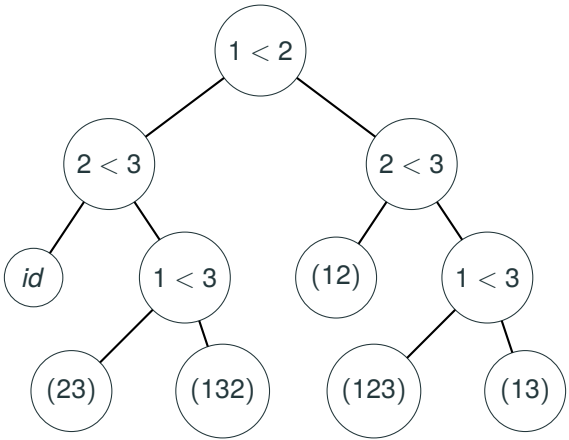
- Innere Knoten: Markierung $i < j$ entsprechend Frage $A[i] \leq A[j]$?
- Linke Nachfolgerkante entspricht JA
- Rechte Nachfolgerkante entspricht NEIN
- Blätter: Sortierende Permutationen $\pi \in \mathcal{S}_n$

Beobachtung 1: Jeder Eingabe $A = (A[1], \cdots, A[n])$ entspricht ein eindeutiger Weg von der Wurzel zu einem Blatt mit Markierung π, so dass π Feld A sortiert, d.h. $A[\pi(1)] \leq \cdots \leq A[\pi(n)]$.

Beobachtung 2: $\mathit{Depth}(T_n)$ ist untere Schranke für die worst-case Laufzeit $\mathit{time}_{\mathit{Sort}}(n)$.

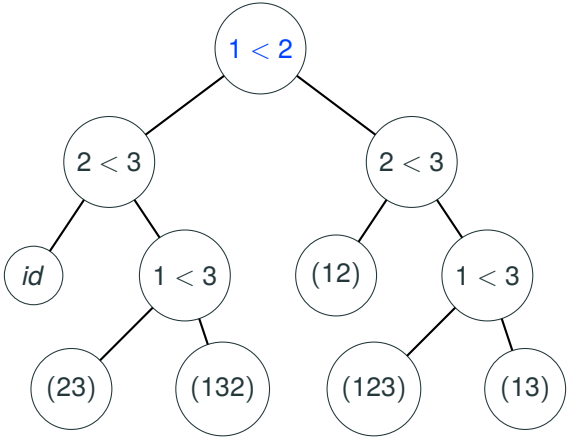
Beobachtung 3: Zu jeder Permutation π existiert Eingabe A, so dass **einzig** π das Feld A sortiert (z.B. $A = (\pi^{-1}(1), \cdots, \pi^{-1}(n))$. Folglich muss zu jedem $\pi \in \mathcal{S}_n$ ein mit π markiertes Blatt existieren.

Beispiel $T(3)$ für *InsertionSort*



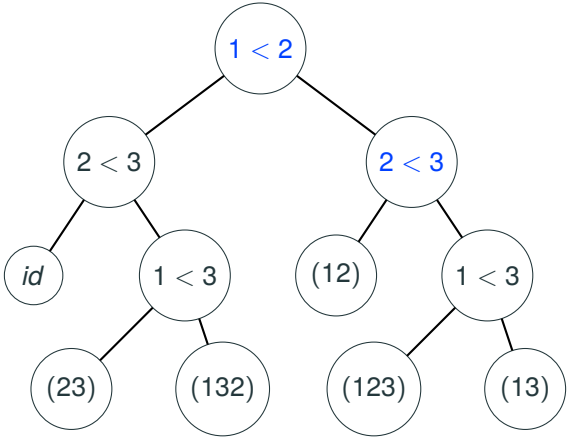
Linke Nachfolgerkante: JA, rechte Nachfolgerkante: NEIN, Permutationen in Zyklenschreibweise

Beispielanwendung $A = [17, 11, 9]$, 1



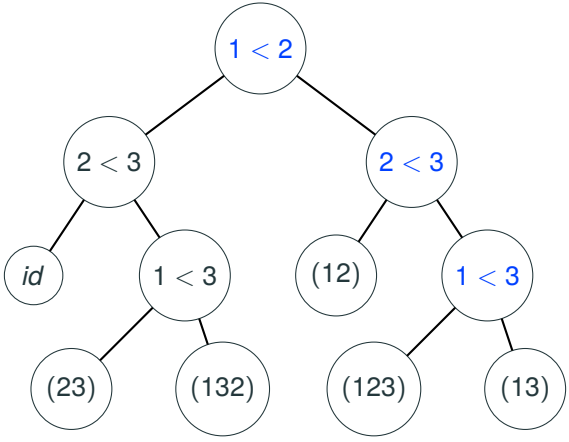
$[17, 11, 9]$

Beispielanwendung $A = [17, 11, 9]$, 2



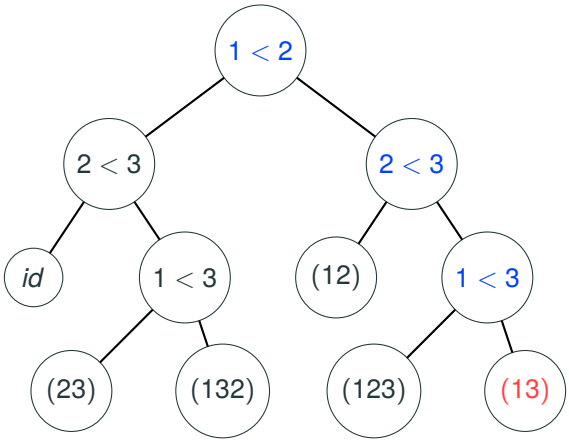
$[17, 11, 9] \rightarrow [11, 17, 9]$

Beispielanwendung $A = [17, 11, 9]$, 3



$[17, 11, 9] \rightarrow [11, 17, 9] \rightarrow [11, 9, 17]$

Beispielanwendung $A = [17, 11, 9]$, 4



$[17, 11, 9] \rightarrow [11, 17, 9] \rightarrow [11, 9, 17] \rightarrow [9, 11, 17]$

Vergleichsbasierte Sortieralgorithmen haben eine Laufzeit von $\Omega(n \cdot \log(n))$

Theorem 32

Jeder vergleichsbasierte Sortieralgorithmus hat eine worst-case Laufzeit der Ordnung $\Omega(n \cdot \log(n))$.

Beweis: Jeder Binärbaum mit S Blättern hat eine Tiefe von mindestens $\lceil \log_2(S) \rceil$, also

$$time_{Sort}(n) \geq \lceil \log_2(n!) \rceil = \Omega(\ln(n!)) = \Omega(n \cdot \log(n)),$$

da $\ln(n!) = \sum_{i=1}^{n} \ln(i) \geq (n-1)(\ln(n-1)-1)+1 = (n-1)\ln(n-1) - n + 2$. $\square$

Folgerung: Vergleichsbasiertes Sortieren in Linearzeit ist nicht möglich,

Aber: Es existieren nicht-vergleichsbasierte Linearzeit Sortieralgorithmen, die die Struktur der Schlüssel ausnutzen.

Sortieren in Linearzeit: *CountingSort*(A, B, k)

Wir betrachten im Folgenden Eingabefelder $A = A[1, \cdots, n]$, bestehend aus Datenobjekten $A[i]$ mit Schlüsseln $A[i].key \in \{0, \cdots, k-1\}$, wobei $k < n$ gelten kann, d.h. Schlüssel kommen in der Regel mehrfach vor.

Definition 33

Ein Sortierverfahren heißt **stabil**, falls für alle $i, j,\ 1 \leq i < j \leq n$, gilt: Ist $A[i].key = A[j].key$ dann steht auch im sortierten Ausgabefeld $A[i]$ links von $A[j]$.

CountingSort(A, B, k) berechnet aus A das Ausgabefeld B mit $B[i] = A[\pi(i)]$ für $i = 1, \cdots, n$, wobei π A bezüglich der Schlüssel $A[i].key$ **stabil** sortiert. Dabei wird ein Hilfsfeld $C = (C[0], \cdots, C[k-1])$ über $\mathbb{N}$ benutzt.

Zusätzliche Parameter

- $h(r) = |\{i, 1 \leq i \leq n, A[i].key = r\}|$ die **Häufigkeit des Schlüssels** r in A.
- $H(r) = \sum_{j=0}^{r} h(j)$ die **Häufigkeit von Schlüsseln der Größe höchstens** r in A.

Beobachtung: Für $r = 1, \cdots, k-1$ gilt $H(r) - H(r-1) = h(r)$.

Idee und Algorithmus CountingSort

Lemma 34

Wir setzen $H(-1) := 0$. Dann ist das Feld B genau dann das ***stabil sortierte Feld bezüglich*** *A, wenn für alle $r = 0, \cdots, k-1$ das Folgende gilt:*

Ist $h(r) > 0$ und sind $i_1 < i_2 < \cdots < i_{h(r)}$ die Positionen zu Schlüssel r, so gilt

$$B[H(r-1)+1] = A[i_1],\ B[H(r-1)+2] = A[i_2], \cdots,\ B[H(r)] = A[i_r]. \square$$

CountingSort(A, B, k)

1 **For** $r \leftarrow 0$ **to** $k-1$ **do** $C[r] \leftarrow 0$
2 **For** $i \leftarrow 1$ **to** n **do** $C[A[i].key] \leftarrow C[A[i].key] + 1$
3 **For** $r \leftarrow 1$ **to** $k-1$ **do** $C[r] \leftarrow C[r-1] + C[r]$
4 **For** $i \leftarrow n$ **downto** 1 **do** $B[C[A[i].key]] \leftarrow A[i]$, $C[A[i].key] \leftarrow C[A[i].key] - 1$.

Laufzeit und Speicherplatz offensichtlich $\Theta(n+k)$.

Korrektheit *CountingSort*(A, B, k)

CountingSort(A, B, k)

(1) **For** $r \leftarrow 0$ **to** $k-1$ **do** $C[r] \leftarrow 0$

(2) **For** $i \leftarrow 1$ **to** n **do**
$C[A[i].key] \leftarrow C[A[i].key] + 1$

(3) **For** $r \leftarrow 1$ **to** $k-1$ **do**
$C[r] \leftarrow C[r-1] + C[r]$

(4) **For** $i \leftarrow n$ **downto** 1 **do**
$B[C[A[i].key]] \leftarrow A[i]$,
$C[A[i].key] \leftarrow C[A[i].key] - 1$.

Korrektheit

- Nach (2) gilt $C[r] = h(r)$, $0 \leq r \leq \cdots, k-1$.
- Nach (3) gilt $C[r] = H(r)$, $0 \leq r \leq \cdots, k-1$.
- Vor Durchlauf i der For-Schleife in (4), $i = n, \cdots, 1$, enthält $C[A[i].key]$ die Position, in die das Element $A[i]$ in B geschrieben werden muss (vor Durchlauf n ist das $H[A[n].key]$). Danach wird der Wert dieser Position um eins erniedrigt.
- Das Durchlaufen der For-Schleife in (4) von links nach rechts garantiert, dass das Verfahren **stabil** ist, d.h., falls $A[i] = A[j]$ für $i < j$ so steht $A[i]$ links von $A[j]$ in B.

Beispiel $CountingSort(A, 15, 5)$, 0

$A = [3, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 0, 0, 0, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 1

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 0, 0, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 2

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 1, 0, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 3

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 2, 0, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 4

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [1, 2, 0, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$**, 5**

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [1, 2, 1, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 6

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [1, 2, 2, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 7

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 2, 2, 1, 0]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 8

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 2, 2, 1, 1]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 9

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 3, 2, 1, 1]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 10

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 3, 2, 2, 1]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 11

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 3, 3, 2, 1]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 12

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 3, 3, 2, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 13

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 3, 3, 2, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 14

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 4, 3, 2, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 15

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 4, 3, 3, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 16

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 4, 4, 3, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 17

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 4, 3, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 18

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 11, 3, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 19

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 11, 14, 2]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 20

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 11, 14, 16]$

$B = [*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beispiel $CountingSort(A, 15, 5)$, 21

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 11, 14, 16]$

$B = [*, *, *, *, *, *, *, *, *, *, 2, *, *, *, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 22

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 10, 14, 16]$

$B = [*, *, *, *, *, *, *, *, *, *, 2, *, *, 3, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 23

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 7, 10, 13, 16]$

$B = [*, *, *, *, *, *, 1, *, *, *, 2, *, *, 3, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 24

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [3, 6, 10, 13, 16]$

$B = [*, *, 0, *, *, *, 1, *, *, *, 2, *, *, 3, *, *]$

Beispiel *CountingSort*$(A, 15, 5)$, 25

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 6, 10, 13, 16]$

$B = [*, *, 0, *, *, *, 1, *, *, *, 2, *, *, 3, *, 4]$

Beispiel $CountingSort(A, 15, 5)$, 26

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 6, 10, 13, 15]$

$B = [*, *, 0, *, *, *, 1, *, *, 2, 2, *, 3, 3, *, 4]$

Beispiel *CountingSort*$(A, 15, 5)$, 27

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 6, 9, 13, 15]$

$B = [*, *, 0, *, *, *, 1, *, *, 2, 2, *, 3, 3, *, 4]$

Beispiel *CountingSort*$(A, 15, 5)$, 28

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 6, 9, 12, 15]$

$B = [*, *, 0, *, *, 1, 1, *, *, 2, 2, *, 3, 3, *, 4]$

Beispiel $CountingSort(A, 15, 5)$, 29

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 5, 9, 12, 15]$

$B = [*, *, 0, *, *, 1, 1, *, *, 2, 2, *, 3, 3, 4, 4]$

Beispiel $CountingSort(A, 15, 5)$, 30

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [2, 5, 9, 12, 14]$

$B = [*, 0, 0, *, *, 1, 1, *, *, 2, 2, *, 3, 3, 4, 4]$

Beispiel $\mathit{CountingSort}(A, 15, 5)$, 31

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [1, 5, 9, 12, 14]$

$B = [*, 0, 0, *, *, 1, 1, *, 2, 2, 2, *, 3, 3, 4, 4]$

Beispiel $CountingSort(A, 15, 5)$, 32

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [1, 5, 8, 12, 14]$

$B = [*, 0, 0, *, *, 1, 1, 2, 2, 2, 2, *, 3, 3, 4, 4]$

Beispiel *CountingSort*$(A, 15, 5)$, 33

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [1, 5, 7, 12, 14]$

$B = [0, 0, 0, *, *, 1, 1, 2, 2, 2, 2, *, 3, 3, 4, 4]$

Beispiel $CountingSort(A, 15, 5)$, 34

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 5, 7, 12, 14]$

$B = [0, 0, 0, *, 1, 1, 1, 2, 2, 2, 2, *, 3, 3, 4, 4]$

Beispiel *CountingSort*$(A, 15, 5)$, 35

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 4, 7, 12, 14]$

$B = [0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, *, 3, 3, 4, 4]$

Beispiel *CountingSort*$(A, 15, 5)$, 36

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 3, 7, 12, 14]$

$B = [0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4]$

Beispiel *CountingSort*$(A, 15, 5)$, 36

$A = [3, 1, 1, 0, 2, 2, 0, 4, 1, 3, 2, 4, 0, 1, 3, 2]$

$C = [0, 3, 7, 11, 14]$

$B = [0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4]$

RadixSort(A)

$\cdots$ sortiert d**-stellige natürliche Zahlen** x, die in k**-närer Darstellung** gegeben sind.

k**-närer Darstellung** heißt:

$x = (x_{d-1}, \cdots, x_0)$, mit $x_r \in \{0, \cdots, k-1\}$, für $r = 0, \cdots, d-1$, und

$$x = \sum_{r=0}^{d-1} x_r \cdot k^r.$$

RadixSort(A)

1 **For** $r \leftarrow 0$ **to** $d-1$
2 **do** Sortiere A stabil bzgl. Ziffer r

Laufzeit $\Theta(d(n+k))$ mit *CountingSort*.

Korrektheit *RadixSort*(A)

Korrektheitsbeweis per Induktion über d:

- **Induktionsanfang** $d = 1$ entspricht *CountingSort*.
- **Induktionsvorausetzung:** Nach Durchlauf $r = d - 2$ ist A bzgl. der Ziffern $d - 2$ bis 0 sortiert.
- **Situation nach Durchlauf** $r = d - 1$**:**

  Wir erhalten eine sortierte Blockzerlegung von A in $A = (A^0, \cdots, A^{k-1})$ entsprechend der führenden Ziffer an Stelle $d - 1$.

  Alle Teilblöcke $A^0, \cdots, A^{k-1}$ sind nach Induktionsvoraussetzung sortiert, da das verwendetes Komponenten Sortierverfahren stabil ist. □

Beispiel *RadixSort*, 0

2178

9422

4709

8705

1403

4088

1731

8199

6413

3715

2443

Beispiel *RadixSort*, 1

2178

9422

4709

8705

1403

4088

1731

8199

6413

3715

2443

Beispiel *RadixSort*, 2

1731

9422

1403

6413

2443

8705

3715

2178

4088

4709

8199

Beispiel *RadixSort*, 3

1403

8705

4709

6413

3715

9422

1731

2443

2178

4088

8199

Beispiel *RadixSort*, 4

4088

2178

8199

1403

9422

2443

6413

8705

4709

3715

1731

Beispiel *RadixSort*, 5

1403

1731

2178

2443

3715

4088

4709

6413

8199

8705

9422

Hashing

Hashing

Grundlegendes

Hashing, Anliegen

- Wir betrachten eine Menge X aus **Datenobjekte** x, die mittels eines Schlüssels $x.key \in U$ identifiziert und verwaltet werden (U Schlüsselraum).
- Wir suchen eine passende Datenstruktur und effiziente Algorithmen, um X bezüglich der folgenden Operationen dynamisch zu verwalten:
  - $Insert(X, x)$ (füge Objekt x zu X hinzu),
  - $Search(X, key)$ (Suche in X nach dem Objekt x mit $x.key = key$),
  - $Delete(X, key)$ (Entferne x mit $x.key = key$ aus X)
- **Problem:** U ist sehr groß, $|X|$ ist viel kleiner als $|U|$.
- **Beispiel:** X sei die Menge der Studierenden an der Uni Mannheim ($|X| \approx 12.000$), die mittels ihrer Matrikelnummer $x.key \in \{0, 10^7 - 1\}$ verwaltet werden.

Triviale Lösungen: Implementieren X als **Feld oder als Liste**.

Implementierung als Feld, Direct Adressing (DA)

T bezeichne ein Feld über dem Indexbereich U.

- $DAInsert(T, x)$:
  1 $T[x.key] \leftarrow x$
- $DASearch(T, key)$
  1 **If** $T[key] \neq NIL$
  2 **then return** $T[key]$
  3 **else return not found**
- $DADelete(T, key)$:
  1 $T[key] \leftarrow NIL$

- **Vorteil:** Alle Operationen haben Laufzeit $O(1)$
- **Nachteil:** U ist i.d.R. unpraktikabel groß.
  Da $|X|$ viel kleiner als $|U|$ ist, wird nur ein winziger Teil des Feldes genutzt.

Implementierung als Liste

Wir speichern die Datenobjekte x als **Listenelemente mit zusätzlichen Zeigern** $x.pred$ **und** $x.succ$ in einer **doppelt verketteten Liste** L, die durch den Zeiger $L.head$ auf das zuletzt eingefügte Element adressiert wird.

- $\underline{ListInsert(L.head, x)}$
  1 $x.prev \leftarrow NIL$
  2 $x.succ \leftarrow L.head$
  3 $(L.head).prev \leftarrow x$
  4 $L.head \leftarrow x$
- $\underline{ListSearch(L.head, key)}$
  1 $search \leftarrow L.head$
  2 **while** $(search.key \neq key) \wedge (search.key \neq NIL)$
  3 **do** $search \leftarrow search.succ$
  4 **If** $(search.key \neq NIL)$
  5 **then return** $search$
  6 **else return not found**

Implementierung als Liste, Fortsetzung

ListDelete$(L.head, key)$

```
1 search ← Search(L, key)
2 If search.key ≠ NIL
3    then (search.prev).succ ← search.succ
4         (search.succ).prev ← search.prev
5    else return not found
```

- **Vorteile:** Die Größe von L ist stets $O(|X|)$, Insert läuft in $O(1)$.
- **Nachteil:** Search und Delete laufen in $\Omega(|L|)$, da ggf. die ganze Liste durchsucht werden muss.

Hashing, Grundidee

- Wir verwenden eine **Hashfunktion**

$$h : U \to \{0, \cdots, m-1\},$$

  wobei m ungefähr der erwarteten Größe von X entspricht.
- **Modellannahme:** $|X| \approx \alpha \cdot m$, α konstant.
- Der Wert $h(key)$ heißt **Hashwert** von key.
- Objekte x werden unter der Adresse $h(x.key)$ abgespeichert.
- **Problem:** Es können **Kollisionen**, d.h. Paare $x \neq x'$ mit $h(x.key) = h(x'.key)$, auftreten.
- Man unterscheidet verschiedene Hashing-Strategien, die sich hauptsächlich im Kollisionsmanagement unterscheiden.
- Hier zunächst: **Hashing by Chaining**:

Hashing

Hashing by Chaining

Definition Hashing by Chaining

- Wir fixieren eine Hashfunktion $h : U \to \{0, \cdots, m-1\}$.
- Wir verwenden eine **Hashtabelle**, d.h. ein Feld $T = T[0, \cdots, m-1]$.
- Jedes $T[j]$ enthält den Zeiger auf eine Liste, die alle bereits eingefügten Elemente $x \in X$ mit $h(x.key) = j$ enthält.
- $\underline{ChInsert(T, x) :} \; ListInsert(T[h(x.key)], x)$.
- $\underline{ChSearch(T, key) :} \; ListSearch(T[h(key)], key)$
- $\underline{ChDelete(T, key) :} \; ListDelete(T[h(key)], key)$
- Die **Laufzeit** von $ChInsert(T, x)$ ist $O(1)$
- Die **Laufzeit** von *ChSearch* und *ChDelete* ist $\Theta(|X|)$.
- Die **Worst Case Laufzeit** tritt ein, wenn $h(x.key)$ für alle $x \in X$ gleich ist.

Beispiel Hashing by Chaining

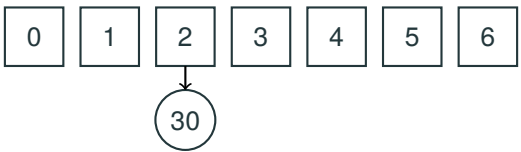
Hashfunktion $h(k) = k \mod 7$

Schlüssel 93, 41, 23, 49, 17, 59, 69, 81, 14, 92, 40, 57, 76, 30

Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

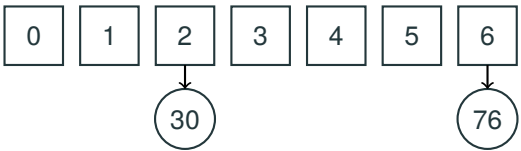
Schlüssel 93, 41, 23, 49, 17, 59, 69, 81, 14, 92, 40, 57, 76



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

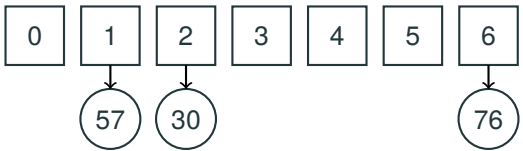
Schlüssel 93, 41, 23, 49, 17, 59, 69, 81, 14, 92, 40, 57



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

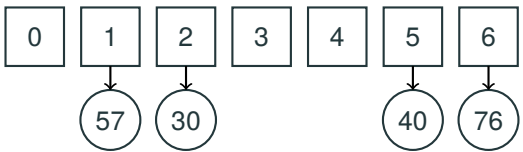
Schlüssel 93, 41, 23, 49, 17, 59, 69, 81, 14, 92, 40



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

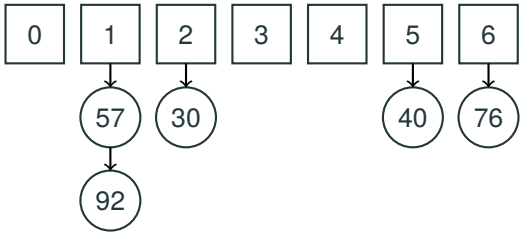
Schlüssel 93, 41, 23, 49, 17, 59, 69, 81, 14, 92



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

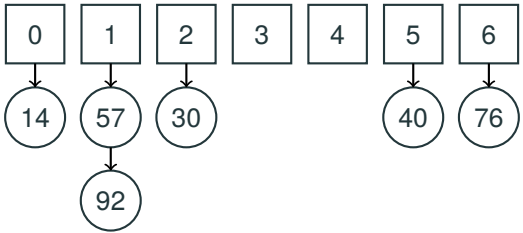
Schlüssel 93, 41, 23, 49, 17, 59, 69, 81, 14



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

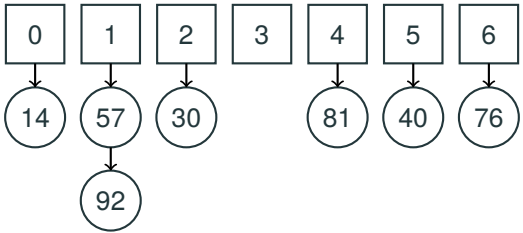
Schlüssel 93, 41, 23, 49, 17, 59, 69, 81



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

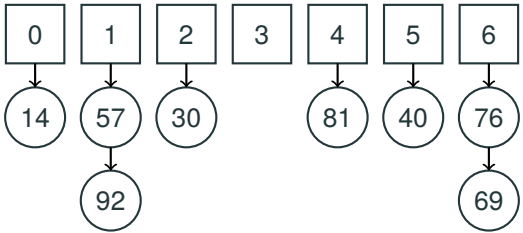
Schlüssel 93, 41, 23, 49, 17, 59, 69



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

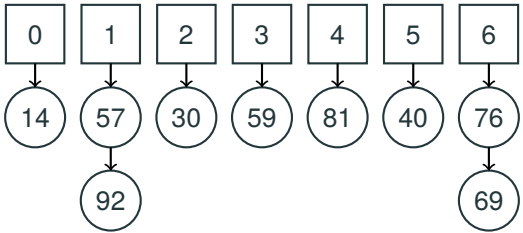
Schlüssel 93, 41, 23, 49, 17, 59



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

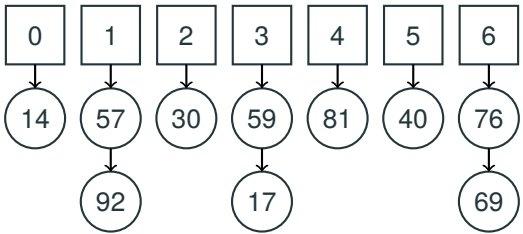
Schlüssel 93, 41, 23, 49, 17



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

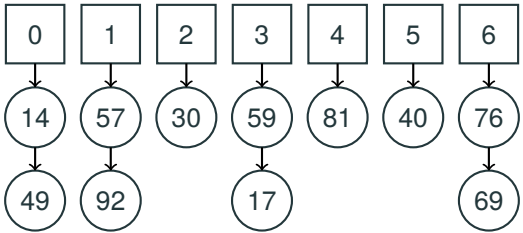
Schlüssel 93, 41, 23, 49



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

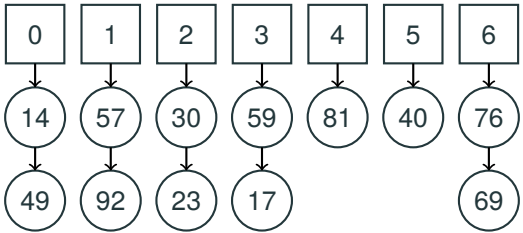
Schlüssel 93, 41, 23



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

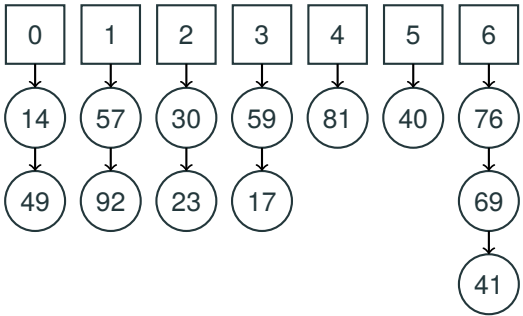
Schlüssel 93, 41



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

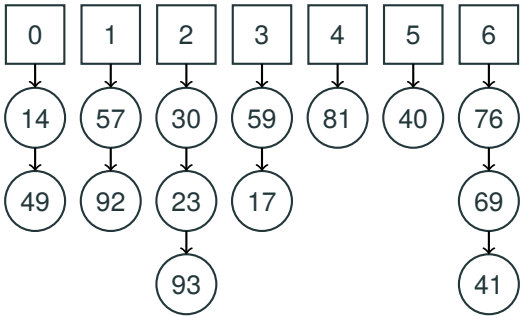
Schlüssel 93



Beispiel Hashing by Chaining

Hashfunktion $h(k) = k \mod 7$

Schlüssel



Simple Uniform Hashing

- **Modellannahme:** Es werden n Objekte zufällig und unabhängig voneinander und unabhängig von der gewählten Hashfunktion $h : U \longrightarrow \{0, \cdots, m-1\}$ gemäß einer Wahrscheinlichkeitsverteilung Pr gezogen.
- Für alle Schlüssel $k \in U$ bezeichnet $Pr[k]$ die Wahrscheinlichkeit für das Elementarereignis, dass das Objekt x mit $x.key = k$ gezogen wird.

Definition 35

Die Hashfunktion h erfüllt die Bedingung des **Simple Uniform Hashing** bezüglich Pr, falls $Pr[h(x.key) = j] = 1/m$ für alle j, $0 \leq j \leq m-1$.

Bemerkung: $Pr[h(x.key) = j] = \sum_{k \in U, h(k)=j} Pr[k]$.

Praktische Bedeutung: Ist in einem Anwendungsszenario die Bedingung des uniformen Hashings gegeben, dann haben die kritischen Operationen Search und Delete eine **konstante Average Case Laufzeit**!

Die Average Case Laufzeit von *ChSearch*(T, key)

- Es bezeichne $X = \{x_1, \cdots, x_n\}$ die Menge der gezogenen Objekte in der Reihenfolge ihres Einfügens in T.
- Für i, $1 \leq i \leq n$ und j, $1 \leq j \leq m$ sei die Zufallsgröße $C_{i,j} \in \{0, 1\}$ definiert als

$$C_{i,j} = 1 \iff h(x_i.key) = j.$$

- Wegen der Bedingung des Simple Uniform Hashing gilt für alle i, $1 \leq i \leq n$ und alle j, $1 \leq j \leq m$, dass

$$Pr[C_{i,j} = 1] = 1/m.$$

- Wir wenden die Formel des bedingten Erwartungswerts an für die Bedingung, dass X das Objekt mit Schlüssel *key* nicht enthält (erfolglose Suche) bzw. doch enthält (erfolgreiche Suche).

Die Average Case Laufzeit von $ChSearch(T, key)$, Erfolglose Suche

Erfolglose Suche: $x_i.key \neq key$ für alle $i = 1, \cdots, n$.

Dann entspricht die Average Case Laufzeit der erwarteter Länge von Liste $T[h(key)]$ plus 1, also

$$1 + \mathbf{E}\left[\sum_{i=1}^{n} C_{i,h(key)}\right] = 1 + \sum_{i=1}^{n} \mathbf{E}\left[C_{i,h(key)}\right]$$

$$= 1 + \sum_{i=1}^{n} Pr[C_{i,h(key)} = 1]$$

$$= 1 + \frac{n}{m} = O(1 + \alpha).$$

Erinnerung: Der Erwartungswert einer Zufallsgröße C mit Werten in $\{0, 1\}$ ist $Pr[C = 1]$.

Die Average Case Laufzeit von *ChSearch*, Erfolgreiche Suche

- **Erfolgreiche Suche:** Es existiert genau ein i, $i = 1, \cdots, n$, so dass $x_i.key = key$.
- Die erwartete Laufzeit von $ChSearch(T, key)$ unter der Bedingung, dass $h(x_i.key) = key$, ist gleich der erwarteten Anzahl von Elementen aus $x_{i+1}, \cdots, x_n$, die auf den gleichen Hashwert abgebildet werden wie x_i, plus einem Schlüsselvergleich für key, also gleich $1 + \frac{n-i}{m}$.
- Die Wahrscheinlichkeit, dass $x_i.key = key$ ist für alle $i = 1, \cdots, n$ gleich, also gleich $\frac{1}{n}$.
- Die erwartete Laufzeit einer erfolgreichen Suche ist also gleich

$$\sum_{i=1}^{n} \frac{1}{n} \left(1 + \frac{n-i}{m}\right) = 1 + \frac{1}{nm} \sum_{i=0}^{n-1} n - i = 1 + \frac{1}{nm} \sum_{i=0}^{n-1} i$$

$$= 1 + \frac{1}{nm} \frac{(n-1)n}{2} = 1 + \frac{n-1}{2m} = O(1 + \alpha).$$

Praktisch eingesetzte Hashfunktionen, die Divisionsmethode

Voraussetzung: $U \subseteq \mathbb{N}$, Schlüssel kodiert als Binärzahlen

- **Divisionsmethode:**

$$h(x) = x \mod m,$$

  m geeignet gewählt.

- **Empfehlung:** m so gewählt, dass $h(x)$ von allen Ziffern von x abhängt, also $m \neq 2^d$.
- **Am Besten:** m Primzahl, die nicht in der Nähe einer Zweierpotenz liegt.
- h erfüllt Simple Uniform Hashing falls

$$Pr[x \mod m = j] \approx 1/m$$

  für alle $j = 0, \cdots, m-1$.

Praktisch eingesetzte Hashfunktionen, die Multiplikationsmethode

- Wir wählen eine geeignete relle Zahl A und setzen

$$h(x) = \lfloor m \cdot (A \cdot x \mod 1) \rfloor.$$

- Hierbei ist für $r \in \mathbb{R}$:

$$r \mod 1 = r - \lfloor r \rfloor.$$

Beispiel: $13.47801 \mod 1 = 0.47801$.

- h erfüllt die Bedingung des Simple Uniform Hashing, falls

$$Pr\left[A \cdot x \mod 1 \in \left[\frac{j}{m}, \frac{j+1}{m}\right]\right] \approx 1/m$$

für alle $j = 0, \cdots, m-1$.

- **Empfehlung:** Goldener Schnitt, $A = \frac{\sqrt{5}-1}{2}$.

Praktisch eingesetzte Hashfunktionen, $\mathbb{Z}_p$-lineare Funktionen

- Es sei $U \subseteq \mathbb{N}$ ein Universum und p eine geeignete Primzahl, die der Anzahl der Hashwerte entspricht.
- Außerdem sei r minimal, so dass $p^r \geq |U|$.
- Wir kodieren Schlüssel $k \in U$ als r-stellige p-näre Zahlen $k = (k_{r-1}, \cdots, k_0)$.
- Zur Definition der Hashfunktion $h_{\vec{a}} : U \longrightarrow \mathbb{Z}_p$ fixieren wir einen Vektor $\vec{a} = (a_{r-1}, \cdots, a_0) \in \mathbb{Z}_p^r$.

$$h_{\vec{a}}(k_{r-1}, \cdots, k_0) = \sum_{i=0}^{r-1} a_i \cdot k_i \mod p.$$

Beispiel: $U = \{0, \cdots, 999\}$, $p = 7$, d.h., $r = 4$, da $7^3 = 343 < 1000 < 2401 = 7^4$.

- Kodieren $k = 472$ (in Dezimaldarstellung) als 4-stellige 7-näre Zahl: $472 = 1 \cdot 343 + 2 \cdot 49 + 4 \cdot 7 + 3$, d.h. $472 \rightarrow (1, 2, 4, 3)$
- Betrachten Hashfunktion $h_{(5,6,1,3)} : U \longrightarrow \mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$.

$$h_{(5,6,1,3)}(1, 2, 4, 3) = 5 \cdot 1 + 6 \cdot 2 + 1 \cdot 4 + 3 \cdot 3 = 5 + 12 + 4 + 9 = 30 \mod 7 = 2.$$

Hashing

Universelles Hashing

Motivation Universelles Hashing

- **Szenario:** Ein bösartiger Gegenspieler wählt in Kenntnis der Hashfunktion h die Schlüssel $k_1, \cdots, k_n$ so, dass $h(k_1) = \cdots = h(k_n)$.
- Dadurch wird die Hashtabelle **maximal unbalanciert**, und die Laufzeiten für Search und Delete werden $\Omega(n)$.
- **Ausweg:** Wählen $h : U \longrightarrow \{0, \cdots, m-1\}$ zufällig aus einer **universellen Familie** $\mathcal{H}$ von Hashfunktionen, wobei das **Wissen um** h **nur dem Betreiber der Hashtabelle** vorbehalten bleibt.

Definition 36

Eine Familie $\mathcal{H}$ von Hashfunktion $h : U \longrightarrow \{0, \cdots, m-1\}$ heißt **universell**, falls für alle $k \neq k' \in U$ gilt:

$$Pr_{h \in \mathcal{H}}[h(k) = h(k')] = \frac{1}{m},$$

wobei h zufällig aus $\mathcal{H}$ gemäß der Gleichverteilung gewählt wird.

Anmerkung: Eine stärkere Forderung wäre $Pr_{h \in \mathcal{H}}[h(k) = a \ \wedge \ h(k') = a'] = \frac{1}{m^2}$ für alle a, a' in $\{0, \cdots, m-1\}$.

Analyse der Listenlänge bei universellem Hashing

- Es sei $X = \{x_1, \cdots, x_n\} \subseteq U$ eine beliebig fixierte Menge von Schlüsseln in der Reihenfolge ihres Einfügens, und $\mathcal{H}$ eine universelle Menge von Hashfunktionen.
- Es sei $C^h_{r,s} = 1$ falls $h(x_r) = h(x_s)$ und $C^h_{r,s} = 0$ falls $h(x_r) \neq (x_s)$.
- Wir schätzen die Länge $|T[j]|$ von Liste $T[j]$ für einen beliebig fixierten Hashwert j, $0 \leq j \leq m-1$, ab.
- Schlechtester Fall $h(x_1) = j$. Dann

$$\mathbf{E}_{h\in\mathcal{H}}\left(|T[j]|\right) = \mathbf{E}_{h\in\mathcal{H}}\left(1 + \sum_{i=2}^{n} C^h_{1,i}\right) = 1 + \sum_{i=2}^{n} \mathbf{E}_{h\in\mathcal{H}}\left(C^h_{1,i}\right) =$$

$$= 1 + \sum_{i=2}^{n} Pr_{h\in\mathcal{H}}\left[C^h_{1,i} = 1\right] = 1 + \sum_{i=2}^{n} Pr_{h\in\mathcal{H}}[h(x_i) = h(x_1)]$$

$$= 1 + \frac{n-1}{m} = O(1+\alpha).$$

Standardbeispiel für universelles Hashing

Es sei p prim und r so gewählt, dass $p^r \geq |U|$.

Wir kodieren die Schlüssel $k \in U$ als r-stellige p-näre Zahlen $k = (k_{r-1}, \cdots, k_0)$ mit Ziffern aus dem endlichen Körper $\mathbb{Z}_p = \{0, \cdots, p-1\}$, und definieren

$$\mathcal{H} = \{h_{\vec{a}} : \mathbb{Z}_p^r \to \mathbb{Z}_p; \vec{a} = (a_{r-1}, \cdots, a_0) \in \mathbb{Z}_p^r\},$$

mit $h_{\vec{a}}(k) = \sum_{i=0}^{r-1} a_i \cdot k_i \mod p$.

Theorem 37

Für alle $k \neq k' \in U$ gilt $Pr_{h_{\vec{a}} \in \mathcal{H}}[h_{\vec{a}}(k) = h_{\vec{a}}(k')] = \frac{1}{p}$.

Beweis: Die Menge der Vektoren $\vec{a}$ mit $h_{\vec{a}}(k) = h_{\vec{a}}(k')$, also $\sum_{i=0}^{r-1}(k_i - k'_i) \cdot a_i \equiv 0 \mod p$, bildet einen Unterraum der Kodimendion 1 (also der Dimension $r-1$) im r-dimensionalen $\mathbb{Z}_p$-Vektorrraum U, d.h.

$$Pr_{h_{\vec{a}} \in \mathcal{H}}[h_{\vec{a}}(k) = h_{\vec{a}}(k')] = \frac{p^{r-1}}{p^r} = \frac{1}{p}. \square$$

Hashing

Offene Adressierung

Offene Adressierung (OA)

- ... ist ein **kollisionsfreies** Hashverfahren für den Fall $n < m$, d.h. $\alpha < 1$.
- OA unterstützt nur *Insert* und *Search*, kein *Delete*.
- OA benutzt eine Hashtabelle $T = T[0, \cdots, m-1]$.
- OA benutzt Hashfunktionen der Art

$$h : U \times \{0, \cdots, m-1\} \to \{0, \cdots, m-1\},$$

  mit $h(u,i) \neq h(u,j)$ für alle $u \in U$ und $i \neq j \in \{0, \cdots, m-1\}$.
- **Das heißt:** Die Hashfunktion h, ordnet jedem Schlüssel $u \in U$ eine Permutation $h(u) = (h(u,0), h(u,1), \cdots, h(u,m-1)$ aus $\mathcal{S}_m$ zu, die als **Probiersequenz** von u bezeichnet wird.
- **Intuition für** *Insert***:** Probiere zunächst $T[h(u,0)]$, falls besetzt dann $T[h(u,1)]$ usw. bis eine freie Feldposition zum Abspeichern gefunden ist.
- Das ist nach spätestens n Schritten der Fall, da $n < m$.

Beispiel Offene Adressierung

Hashfunktion $h(k,i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59, 41, 78, 39, 22, 80

Beispiel Offene Adressierung, $80 + 0 \mod 7 = 3$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59, 41, 78, 39, 22

Beispiel Offene Adressierung, $22 + 0 \mod 7 = 1$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59, 41, 78, 39

Beispiel Offene Adressierung, $39 + 0 \mod 7 = 4$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59, 41, 78

Beispiel Offene Adressierung, $78 + 0 \mod 7 = 1$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59, 41, 78

Beispiel Offene Adressierung, $78 + 1 \mod 7 = 2$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59, 41

Beispiel Offene Adressierung, $41 + 0 \mod 7 = 6$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59

*	22	78	80	39	*	41

Beispiel Offene Adressierung, $59 + 0 \mod 7 = 3$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59

*	22	78	80	39	*	41

Beispiel Offene Adressierung, $59 + 1 \mod 7 = 4$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel 59

*	22	78	80	39	*	41

Beispiel Offene Adressierung, $59 + 2 \mod 7 = 5$

Hashfunktion $h(k, i) = k + i \mod 7,\ i = 0, \cdots, 6$

Schlüssel

*	22	78	80	39	59	41

Die OA Operation *Insert*

OAInsert(T, x) durchsucht T entlang der Probiersequenz $h(x.key)$ bis zur ersten freien Position und schreibt x in diese Position.

OAInsert(T, x)

```
1  j ← 0
2  while T[h(x.key, j)] ≠ NIL and j < m − 1
3        do j ← j + 1
4  if T[h(x.key, j)] = NIL
5     then T[h(x.key, j)] ← x
6     else return error n = m
```

Laufzeit $\Theta(\min\{j, T[h(x.key, j] = NIL\})$.

Die OA Operation Search

OASearch(T, u) ($u \in U$) durchsucht T entlang der Probiersequenz $h(u)$ solange, bis entweder ein Objekt x mit $x.key = u$ gefunden wird (erfolgreiche Suche), oder ein leeres Feld erreicht wird (erfolglose Suche).

OASearch(T, u)

```
1 j ← 0
2 while T[h(u,j)].key ∉ {u, NIL} and j < m − 1
3     do j ← j + 1
4 If T[h(u,j)].key = u
5     then return T[h(u,j)]
6     else return not found
```

Die **Laufzeit** ist nach oben beschränkt durch die Laufzeit einer erfolglosen Suche.

Diese ist gleich der Laufzeit von *OAInsert*(T, x), wobei x das Objekt mit $x.key = u$ ist.

Die **Laufzeit** ist also $\Theta(\min\{j, T[h(u,j] = NIL\})$.

Uniformes Hashing und das Uniform Hashing Lemma

Definition 38

Die Hashfunktion $h : U \to \mathcal{S}_m$ erfüllt bezüglich des Wahrscheinlichkeitsraums (U, Pr) die **Bedingung des uniformen Hashing**, falls die Wahrscheinlichkeit $Pr[h(u) = \sigma]$, dass einem zufälligen Schlüssel $u \in U$ durch h die Probiersequenz $\sigma \in \mathcal{S}_m$ zugeordnet wird, für alle Probiersequenzen σ gleich, und damit gleich $\frac{1}{m!}$, ist.

Eine direkte Konsequenz daraus ist

Lemma 39

Die Hashfunktion $h : U \to \mathcal{S}_m$ erfülle bezüglich des Wahrscheinlichkeitsraums (U, Pr) die Bedingung des uniformen Hashing. Dann gilt für alle r, $1 \leq r < m$, und alle Folgen $J = (j_1, \cdots, j_r)$ von paarweise verschiedenen Werten aus $\{1, \cdots, m\}$ das Folgende:

Die Wahrscheinlichkeit für das Ereignis, dass $h(u, r+1) = j$ unter der Bedingung, dass $h(u, 1) = j_1, h(u, 2) = j_2, \cdots, h(u, r) = j_r$, ist für alle $j \in \{1, \cdots, m\} \setminus J$ gleich, und damit gleich $\frac{1}{m-r}$. $\square$

Die Average Case Laufzeit von OA-Search bei uniformem Hashing

Sei T mit n Objekten belegt. Wir betrachten den Erwartungswert $\mathbf{E}[X]$ der Anzahl X von Versuchen bei **erfolgloser Suche** nach u.

Das entspricht der erwarteten Anzahl von Versuchen bei Einfügen des $(n+1)$-ten Objektes, also

$$\mathbf{E}[X] = \sum_{i=1}^{n+1} i \cdot Pr[X = i] \leq \sum_{i=1}^{\infty} i \cdot Pr[X = i]$$

$$= \sum_{i=1}^{\infty} i \cdot (Pr[X \geq i] - Pr[X \geq i+1]) = \sum_{i=1}^{\infty} (i - (i-1)) \cdot Pr[X \geq i] = \sum_{i=1}^{\infty} Pr[X \geq i].$$

Es gilt $X \geq i$ genau dann, wenn die ersten $i-1$ Versuche gemäß Probiersequenz $h(u)$ auf belegte Felder treffen.

Gemäß dem Uniform Hashing Lemma geschieht das mit Wahrscheinlichkeit $\frac{n}{m}$ beim ersten Versuch, mit Wahrscheinlichkeit $\frac{n-1}{m-1}$ beim zweiten Versuch usw. bis zu Wahrscheinlichkeit $\frac{n-i+2}{m-i+2}$ beim (i-1)ten Versuch.

Average Case Laufzeit OASearch, erfolglose Suche

Also gilt

$$Pr[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-i+2}{m-i+2}.$$

Da $n < m$ gilt für alle $x > 0$, dass

$$\frac{n-x}{m-x} < \frac{n}{m}.$$

Also gilt

$$Pr[X \geq i] \leq (n/m)^{i-1} = \alpha^{i-1},$$

wobei $\alpha = n/m < 1$ den Loadfaktor bezeichnet.

Die erwartete Laufzeit für das Einfügen des $(n+1)$-ten Elementes ergibt sich wiefolgt:

$$\mathbf{E}[X] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha} = \frac{m}{m-n}.$$

Average Case Laufzeit erfolgreiche Suche

- T enthalte n Elemente, darunter das Zielobjekt.
- Die Wahrscheinlichkeit, dass das Zielobjekt als i-tes Objekt eingefügt wurde, sei für alle $i = 1, \cdots, n$ gleich $\frac{1}{n}$.
- Die Averagekosten für das Suchen des i-ten Elementes sind gleich derer bei Einfügen des i-ten Elementes, also kleiner gleich $\frac{m}{m-(i-1)}$.

  Die Average Case Laufzeit ist also kleiner gleich

$$\frac{1}{n} \cdot \sum_{i=1}^{n} \frac{m}{m-i+1} = \frac{m}{n} \cdot \sum_{j=m-n+1}^{m} \frac{1}{j}$$

$$\leq \frac{1}{\alpha} \cdot \int_{x=m-n}^{m} \frac{1}{x} dx$$

$$= \frac{1}{\alpha} \cdot \ln\left(\frac{m}{m-n}\right) = \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right) = \frac{m}{n} \cdot \ln\left(\frac{m}{m-n}\right).$$

Praktikabilität Uniformen Hashings

Uniformes Hashing bedingt $|U| \geq m!$, da für jede Suchsequenz $\sigma \in \mathcal{S}_m$ für mindestens ein $u \in U$ gelten muss, dass $h(u) = \sigma$.

Das bedeutet z.B. für $m = 1024$, dass Schlüssel die Bitlänge von mindestens 9000 aufweisen müssen, für m gleich eine Million beträgt die Schlüssellänge mindestens 18 Millionen, was nicht praktikabel ist.

Praktische OA-Verfahren erfüllen die Bedingung des uniformen Hashings deshalb lediglich approximativ.

OA-Verfahren Lineares Probieren

... basiert auf $h_1 : U \longrightarrow \{0, \cdots, m-1\}$ einfache Hashfunktion.

Dann ist $h : U \times \{0, \cdots, m-1\} \longrightarrow \{0, \cdots, m-1\}$ definiert als

$$h(u, i) = h_1(u) + i \mod m$$

- **Vorteil:** $\cdots$ leicht zu implementieren.
- **Nachteil:** $h_1(u)$ bestimmt Probiersequenz, d.h. es gibt nur m verschiedene Probiersequenzen, d.h. Lineares Probieren ist weit von der Bedingung des Uniform Hashings entfernt.
- **Weiterer Nachteil:** Spezielle Gestalt der Probiersequenzen bewirkt Effekt des *Linear Clusterings*, der die Tabelle ineffizient macht.

Linear Clustering

- Die Tabelle $T = (T[0], \cdots, T[m-1])$ enthalte n Elemente, unter anderem ein i-Cluster, das heißt, es existiert j, so dass $T[j], T[j+1], \cdots, T[j+i-1]$ alle belegt.
- Wir berechnen die Wahrscheinlichkeit p^*, dass das Feld $T[j+i]$ bei Einfügen des $(n+1)$ten Elementes x_{n+1} belegt wird:
- **Uniformes Hashing:** Die Wahrscheinlichkeit der Belegung bei Einfügen von x_{n+1} ist für alle freien Felder gleich, also $p^* = \frac{1}{m-n} = \frac{1}{(1-\alpha)m}$.
- **Lineares Probieren:** Feld $T[j+i]$ wird dann belegt, wenn $h_1(x_{n+1}.key)$ in $\{j, \cdots, j+i\}$ liegt, d.h. $p^* = \frac{i+1}{m}$.
- Für realistische Werte von i, n, m gilt $\frac{1}{1-\alpha} < i+1$, d.h. Cluster werden mit signifikant erhöhter Wahrscheinlichkeit immer größer.

Quadratisches Probieren

... basiert auch auf $h_1 : U \longrightarrow \{0, \cdots, m-1\}$, einer einfachen Hashfunktion.

Dann ist $h : U \times \{0, \cdots, m-1\} \longrightarrow \{0, \cdots, m-1\}$ definiert als

$$h(u, i) = h_1(u) + c_1 \cdot i^2 + c_2 \cdot i \mod m$$

- c_1, c_2 müssen so gewählt sein, dass $c_1 \cdot i^2 + c_2 \cdot i \mod m$ als Abbildung in i bijektiv ist (siehe nächste Seite).
- **Nachteil:** kein *Linear Clustering*, aber $h_1(u)$ bestimmt auch hier die Probiersequenz, d.h. es gibt nur m verschiedene Probiersequenzen, d.h. auch Quadratisches Probieren ist weit von der Bedingung des Uniform Hashings entfernt.

Zur Bijektivität von $c_1 \cdot x^2 + c_2 \cdot x$ auf $\mathbf{Z}_m$

Es gilt, dass, falls m Primzahl, die Funktion $f(x) = c_1 \cdot x^2 + c_2 \cdot x = (c_1 \cdot x + c_2) \cdot x$ für $c_1 \neq 0$ niemals bijektiv ist (da $\mathbb{Z}_m$ ein Körper ist, und sowohl $x = 0$ als auch $x = -c_2 \cdot c_1^{-1}$ auf 0 abgebildet werden).

Es gilt jedoch

Lemma 40

Für $m = p^r$, wobei p Primzahl und $r \geq 2$, ist die Funktion $f(x) = p \cdot x^2 + x$ stets bijektiv über $\mathbb{Z}_m$.

Beweis: Wir zeigen, dass für $f(x) = f(y)$ stets $x = y$ folgt. $f(x) = f(y)$ impliziert

$$0 = p(x^2 - y^2) + x - y = (x - y)(p(x + y) + 1).$$

In $\mathbb{Z}_m$ ist jedoch $p \cdot z + 1$ für alle z invertierbar.

Das gilt, da $gcd(p^r, p \cdot z + 1) = 1$.

Damit folgt aus $f(x) = f(y)$ dass $x - y = 0$. $\square$

Double Hashing $\cdots$

$\cdots$ basiert auf zwei einfachen Hashfunktionen $h_1, h_2 : U \longrightarrow \{0, \cdots, m-1\}$.

Dann ist $h : U \times \{0, \cdots, m-1\} \longrightarrow \{0, \cdots, m-1\}$ definiert als

$$h(u, i) = h_1(u) + h_2(u) \cdot i \mod m.$$

- **Bedingung:** Es muss gewährleistet sein, dass $gcd(m, h_2(u)) = 1$ für alle $u \in U$. Das ist beispielsweise gegeben für m prim und $h_2(u) \neq 0$ für alle $u \in U$, oder $m = 2^d$ und $h_2(u)$ ungerade für alle $u \in U$.
- **Vorteil:** $h_1(u)$ und $h_2(u)$ bestimmen Probiersequenz, d.h. es gibt potentiell quadratisch in m viele verschiedene Probiersequenzen.

Binäre Suchbäume

Binäre Suchbäume

Binäre Suchbäume, das Basismodell

Definition Binärer Suchbäume

Anliegen: Wir verwalten Datenobjekte x gemäß ihrer Schlüssel $x.key \in \mathbb{N}$ in einem binären Baum T, gegeben durch $T.root$.

Wir implementieren die Datenobjekte x als **Knotenelemente** mit den zusätzlichen Zeigern

- $x.left$ Zeiger auf den linken Nachfolger (oder NIL, falls nicht existent)
- $x.right$ Zeiger auf den rechten Nachfolger (oder NIL)
- $x.p$ Zeiger auf den Elternknoten (oder NIL falls $x = T.root$ Wurzel)

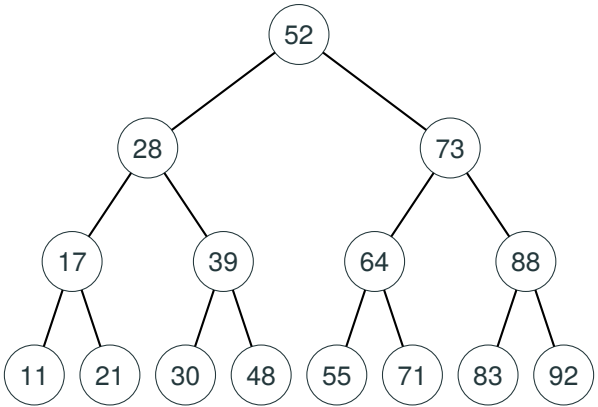
Definition 41

T ist binärer Suchbaum, falls für alle Knoten $x \in T$ gilt:

Ist y im linken Teilbaum an x (d.h. der Teilbaum mit Wurzel $x.left$), dann gilt $y.key \leq x.key$.

Ist y im rechten Teilbaum an x, dann gilt $y.key \geq x.key$.

Beispiel Binärer Suchbaum



Strukturelle Eigenschaften und Verwandtschaftsbeziehungen von Knoten x in T

- **Vater:** $x.p$ (falls $x.p \neq NIL$),
- **Linker Sohn**: $x.left$, **Rechter Sohn**: $x.right$
- T hat **Linkskanten** $(x, x.left)$ und **Rechtskanten** $(x, x.right)$.
- **Test ob x linker (bzw. rechter) Sohn**: $x = x.p.left$ bzw. $x = x.p.right$.
- **Geschwisterknoten**: $x.p.right$ bzw. $x.p.left$, je nachdem ob x linker oder rechter Sohn.
- **Neffen:** $x.p.right.left$ und $x.p.right.right$ bzw. $x.p.left.left$ und $x.p.left.right$, je nachdem ob x linker oder rechter Sohn.
- **Cousins:** $x.p.p.left.left$ und $x.p.p.left.right$, bzw. $x.p.p.right.left$ und $x.p.p.right.right$, je nachdem ob x linker oder rechter Sohn.

Binärer Suchbäume, Operationen

Wir verwalten T dynamisch bezüglich der Operationen

- *InorderTreewalk*(T), gibt die sortierte Folge aller Schlüssel in T aus.
- *Minimum*(T) und *Maximum*(T) (Ausgabe ist Zeiger auf den Knoten mit minimalem bzw. maximalem Schlüssel in T).
- *Search*(T, k), suche x mit $x.key = k$ in T.
- *Predecessor*(T, x) und *Successor*(T, x) (gebe linken bzw. rechten Nachbarn von x bezüglich der sortierten Folge aller T-Schlüssel aus.)
- *Insert*(T, x), fügt x als neuen Knoten in T ein.
- *Delete*(T, x), entferne x aus T.

Voraussetzung: Alle Schlüssel in T sind verschieden.

InorderTreewalk(x)

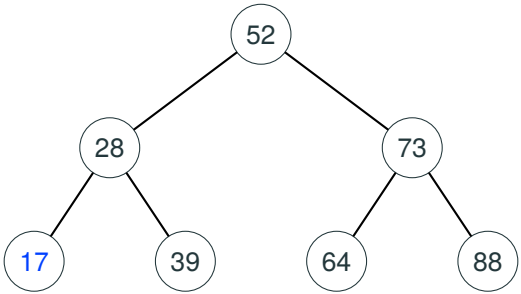
```
1  If x ≠ NIL
2   then InorderTreewalk(x.left)
3        print(x.key)
4        InorderTreewalk(x.right)
```

Laufzeitabschätzung:

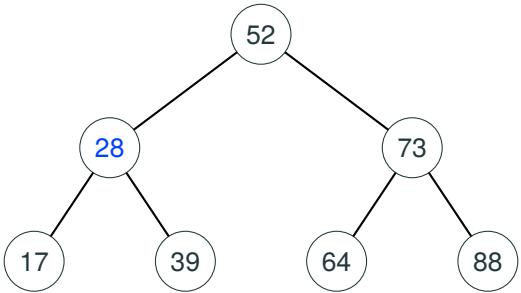
- $t(0) = 1$
- $t(1) = 4$
- $t(n) = 1 + t(k) + 1 + t(n-k-1) = t(k) + t(n-k-1) + 2$ für ein k, $0 \leq k \leq n-1$,

... ergibt $t(n) = 3n + 1$.

Beispiel Inorder Treewalk 1

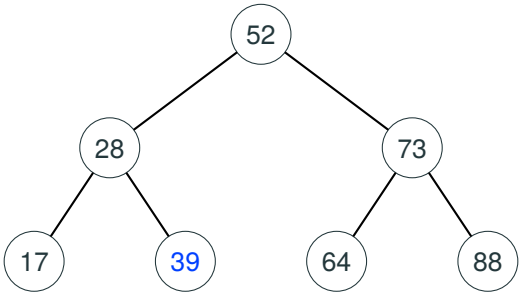


Beispiel Inorder Treewalk 2



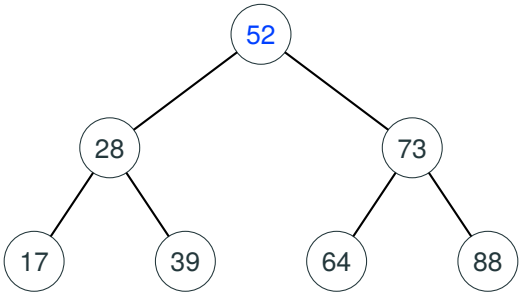
17

Beispiel Inorder Treewalk 3



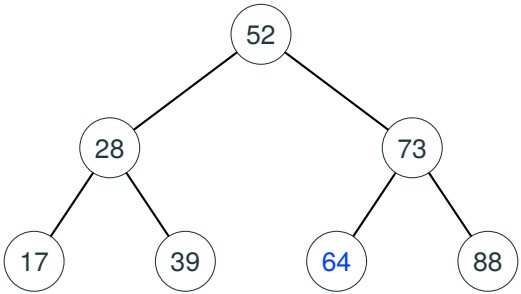
17, 28,

Beispiel Inorder Treewalk 4



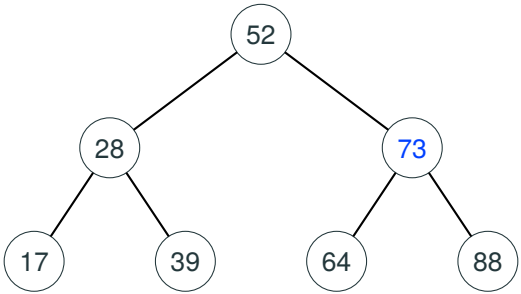
17, 28, 39

Beispiel Inorder Treewalk 5



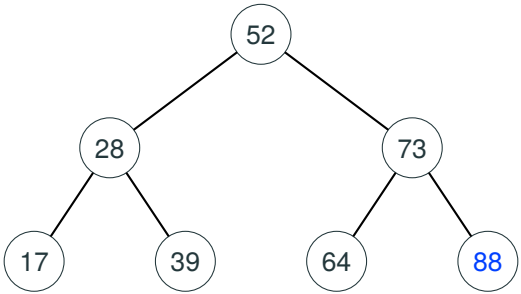
17, 28, 39, 52

Beispiel Inorder Treewalk 6



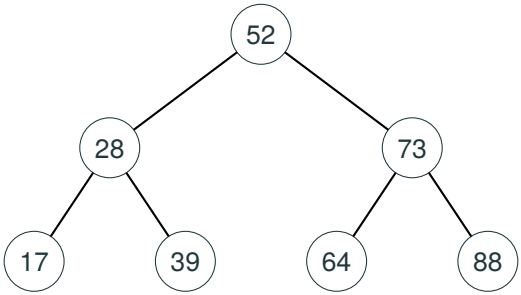
17, 28, 39, 52, 64

Beispiel Inorder Treewalk 7



17, 28, 39, 52, 64, 73

Beispiel Inorder Treewalk 8



17, 28, 39, 52, 64, 73, 88

Minimum und Maximum

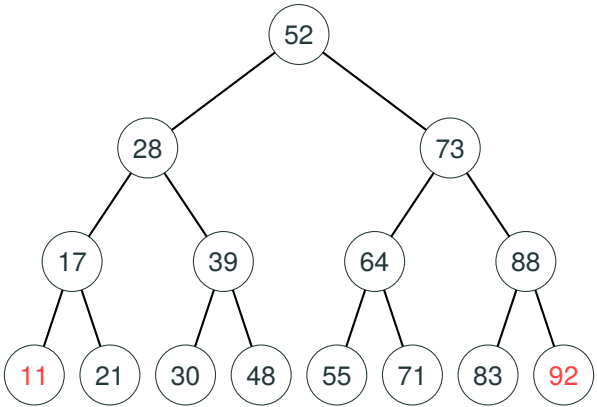
Der minimale bzw. maximale Schlüssel in T steht im linkesten Knoten $Minimum(T)$ bzw. im rechtesten Knoten $Maximum(T)$.

$Minimum(T)$

```
1  x ← T.root
2  If x ≠ NIL
3     then while (x.left ≠ NIL)
4               do x ← x.left
5  return x
```

$Minimum(T)$ berechnet Minimum in Laufzeit $O(height(T))$, $Maximum(T)$ berechnet das Maximum entsprechend.

Beispiel Minimum und Maximum



$Search(T.root, k)$

Grundlegende Beobachtung:

- Zu jedem Schlüssel k existiert ein eindeutig bestimmter Suchpfad zu k in T.
- Dieser startet in der Wurzel $T.root$.
- Erreicht der Suchpfad einen Knoten x mit $x.key \neq k$, so folgt er der linken Kante falls $k < x.key$ und der rechten falls $k > x.key$.
- Der Suchpfad endet entweder in einem Knoten x^* mit $x^*.key = k$, oder an einer (gedacht) auf *NIL* zeigenden Kante.

$Search(T.root, k)$

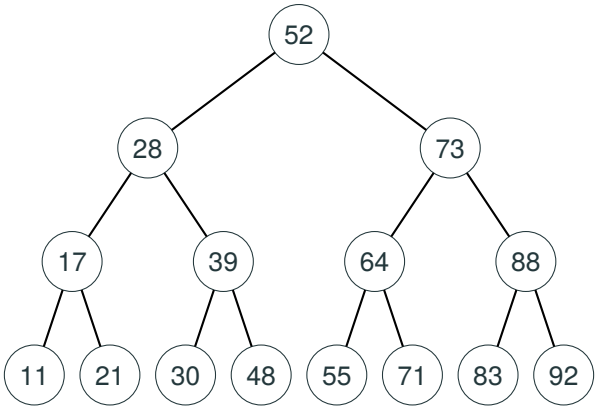
```
1 x ← T.root
2 If (x.key = k) or (x = NIL)
3   then output(x)
4   else if (x.key > k)
5        then Search(x.left, k) else Search(x.right, k)
```

Sequentielles Search

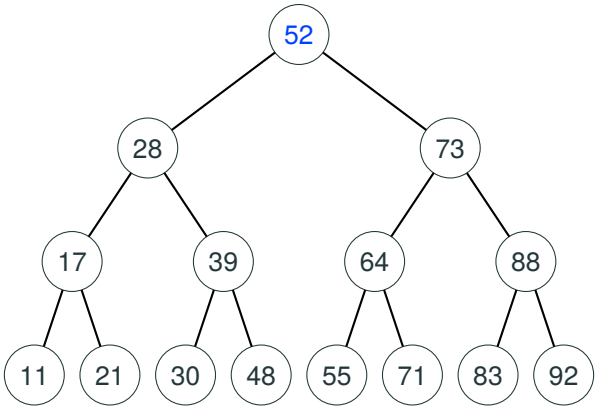
Search$(T.root, k)$

```
1 x ← T.root
2 while (x.key ≠ k) and (x ≠ NIL)
3   do if (x.key > k) then x ← x.left
4                     else x ← x.right
5 output(x)
```

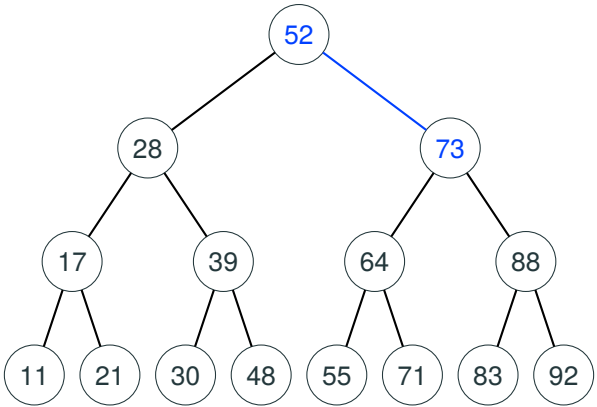
Beispiel *Search*(T, 64), erfolgreich, 0



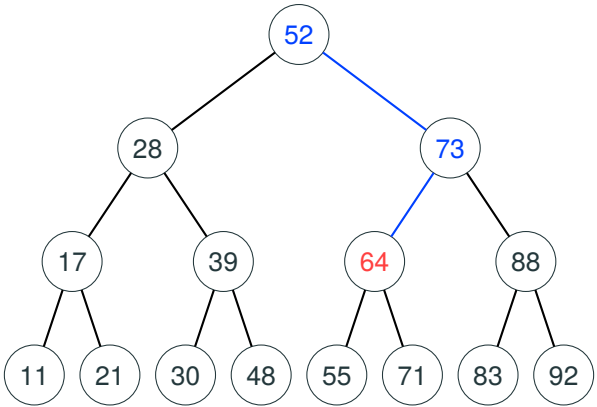
Beispiel *Search*(T, 64), erfolgreich, 1



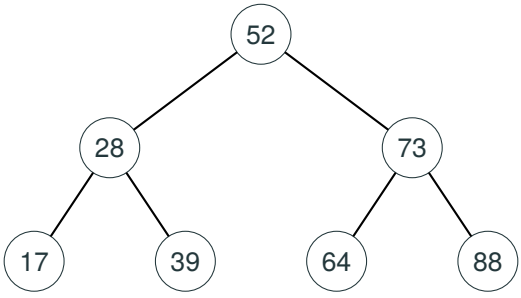
Beispiel *Search*(T, 64), erfolgreich, 2



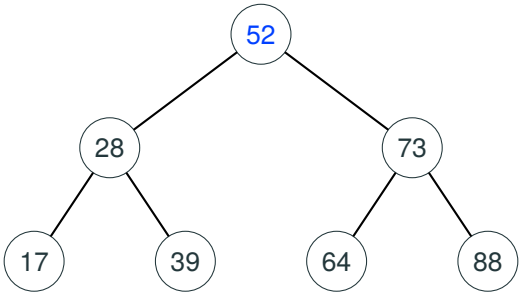
Beispiel *Search*(T, 64), erfolgreich, 3



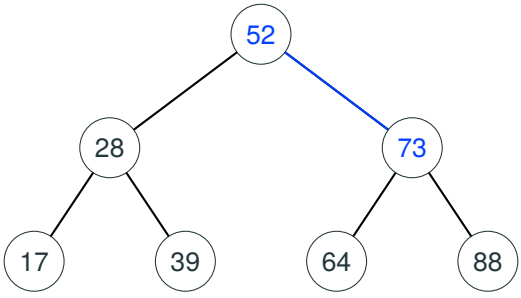
Beispiel *Search*(T, 71), erfolglos, 0



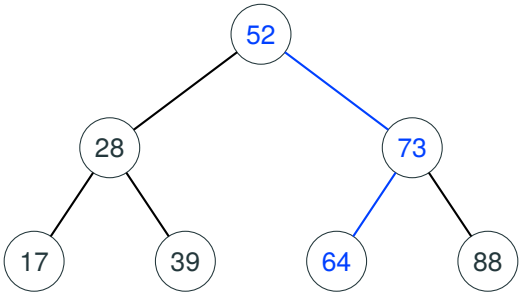
Beispiel *Search*(T, 71), erfolglos, 1



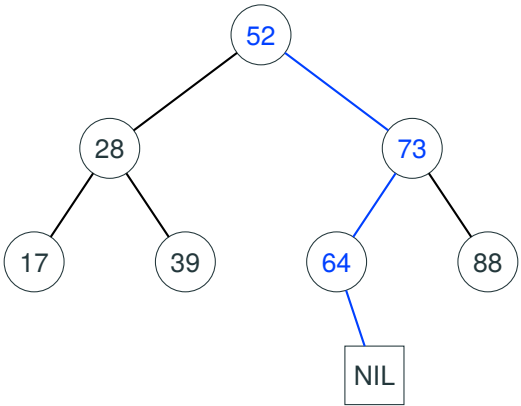
Beispiel *Search*(T, 71), erfolglos, 2



Beispiel *Search*(T, 71), erfolglos, 3



Beispiel *Search*(T, 71), erfolglos, 4

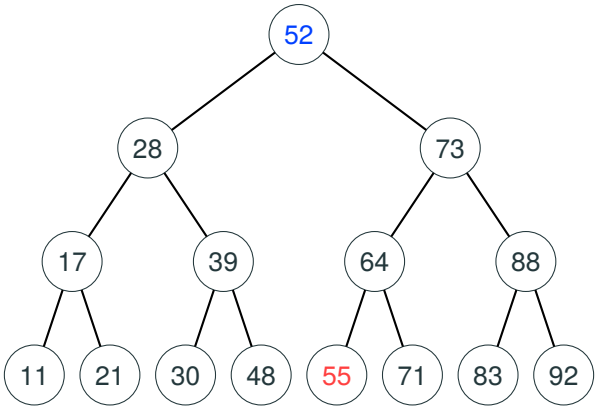


Successor(T, x) **und** *Predecessor*(T, x)

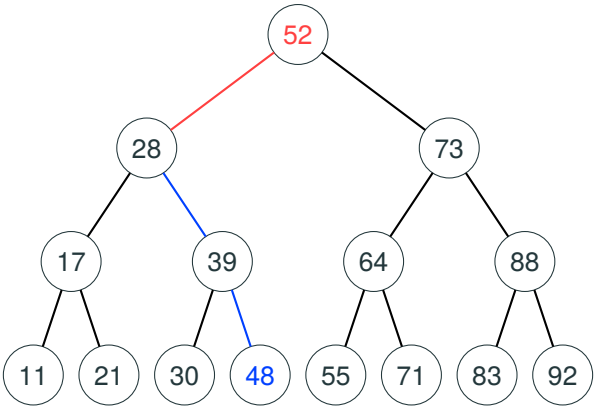
Beobachtungen

- Ist der rechte Teilbaum von x in T nicht leer, so ist der Nachfolger von x das Minimum im rechten Teilbaum von x.
- Entsprechend ist der Vorgänger von x das Maximum im linken Teilbaum von x.
- Ist der rechte Teilbaum von x **leer** und ist x die Wurzel, so hat x keinen Nachfolger.
- Ist der linke Teilbaum von x **leer** und ist x die Wurzel, so hat x keinen Vorgänger.
- Gilt $x.right = NIL$ und $x \neq T.root$ so folge dem Weg von x zur Wurzel. Der erste Knoten y, der über eine Linkskante erreicht wird, ist der Nachfolger. Besteht dieser Weg nur aus Rechtskanten, so ist x das Maximum in T und hat keinen Successor.
- **Begründung:** Der Knoten y ist deshalb der Nachfolger von x, da x der rechteste Knoten im linken Teilbaum an y und damit der Vorgänger von y ist.
- Entsprechendes gilt für den Vorgänger von x.

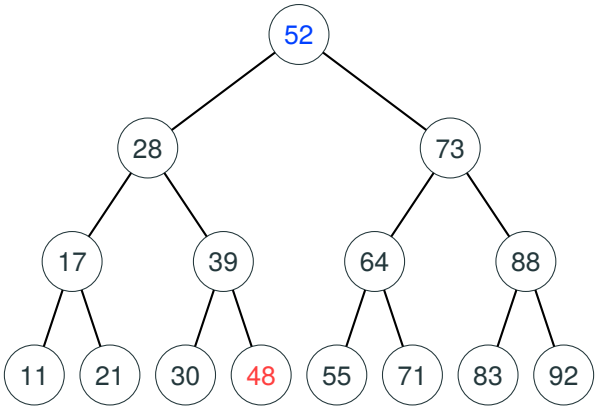
Beispiel $Successor(T.root, 52) = Minimum(T.root.right)$



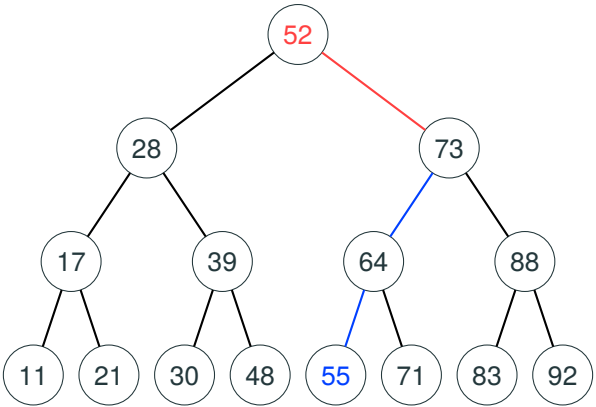
Beispiel *Successor*$(T.root, 48)$



Beispiel $Predecessor(T.root, 52) = Maximum(T.root.left)$



Beispiel *Predecessor*$(T.root, 55)$



$Successor(T, x)$

```
1  If x.right ≠ NIL
2    then return Minimum(x.right)
3    else y ← x.p
4         while (y ≠ NIL) ∧ (x = y.right)
5               do x ← y, y ← y.p
6    return y
```

$Predecessor(T, x)$

```
1  If x.left ≠ NIL
2    then return Maximum(x.right)
3    else y ← x.p
4         while (y ≠ NIL) ∧ (x = y.left)
5               do x ← y, y ← y.p
6    return y
```

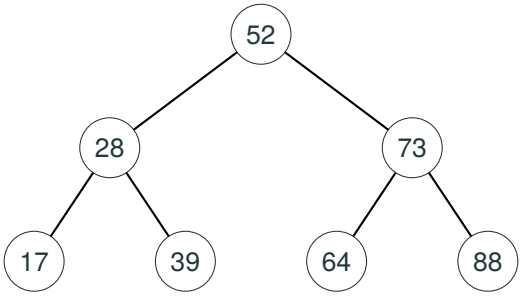
Die Operation Insert

Idee: Folge dem Suchpfad entsprechend $z.key$ und hänge z an die entsprechende NIL-Kante. Laufzeit $O(height(T))$.

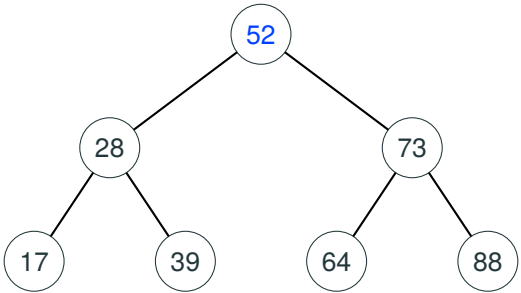
$Insert(T, z)$

```
1  y ← NIL, x ← T.root
2  while (x ≠ NIL)
3       do y ← x
4          If z.key < x.key
5             then x ← x.left else x ← x.right
6  z.p = y
7  If y = NIL then T.root = z
8    elseif z.key < y.key
9         then y.left ← z else y.right ← z
```

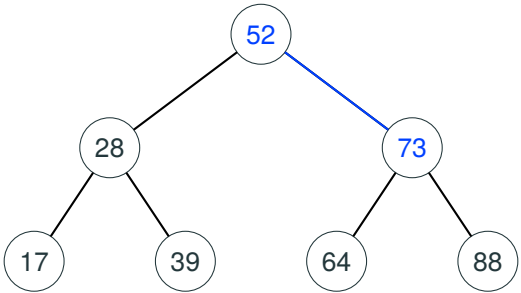
Beispiel *Insert*$(T, 71)$, **0**



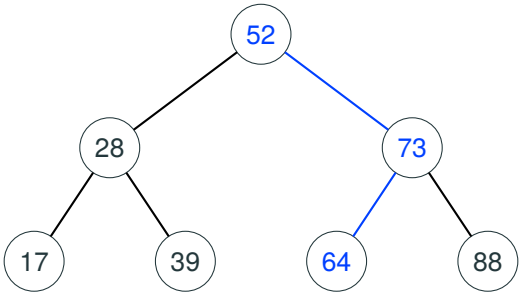
Beispiel *Insert*$(T, 71)$, 1



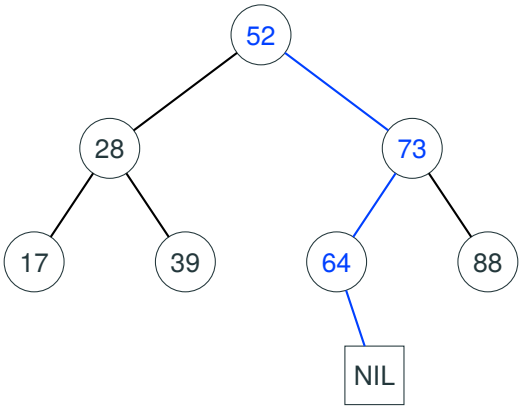
Beispiel *Insert*$(T, 71)$, 2



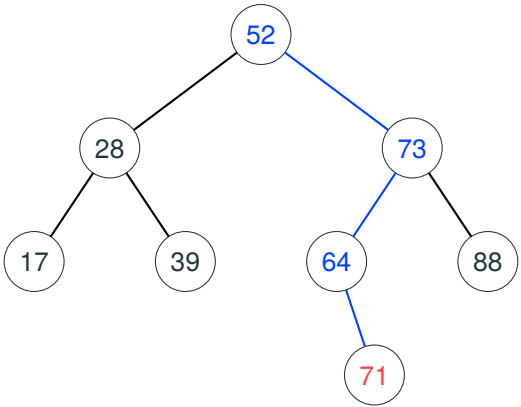
Beispiel *Insert*$(T, 71)$, 3



Beispiel *Insert*$(T, 71)$, 4



Beispiel *Insert*$(T, 71)$, 5



Delete(T, z), informal

Idee

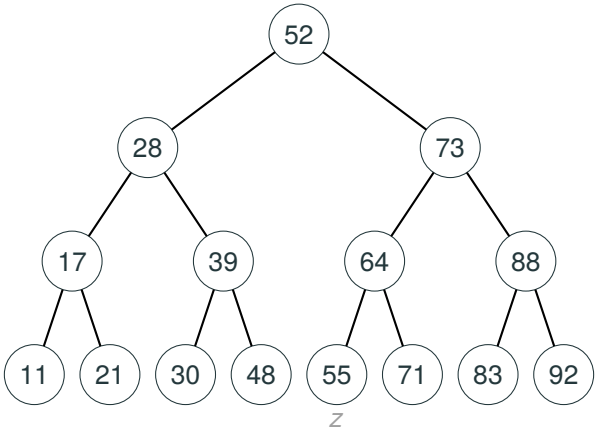
0 Hat Knoten z keinen Sohn, so wird z durch *NIL* ersetzt.

1 Hat z nur einen Sohn x, so wird der Teilbaum mit Wurzel z durch den Teilbaum mit Wurzel x ersetzt.

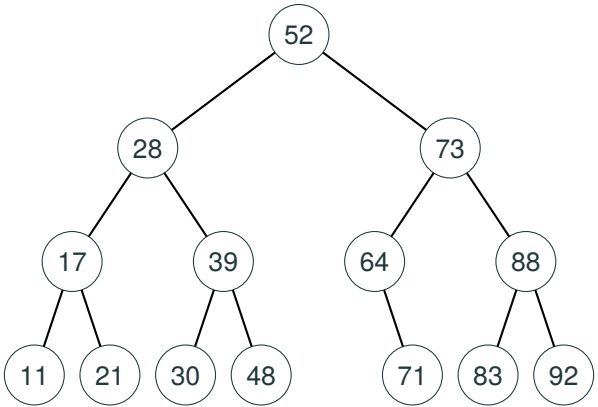
2 Hat z zwei Söhne, so streiche $y = \mathit{Successor}(T, z)$ gemäß Schritt 0 oder 1 (y hat keinen linken Sohn), und überschreibe die Komponenten von z durch die von y.

3 **Laufzeit** $O(\mathit{height}(T))$, wegen der möglichen Berechnung des Nachfolgers.

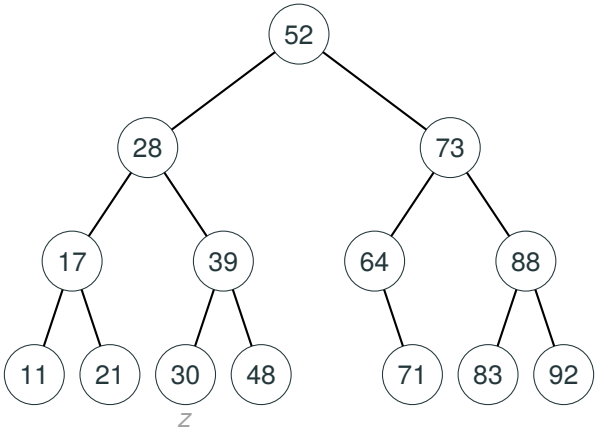
Delete(T, z), $z = 55$



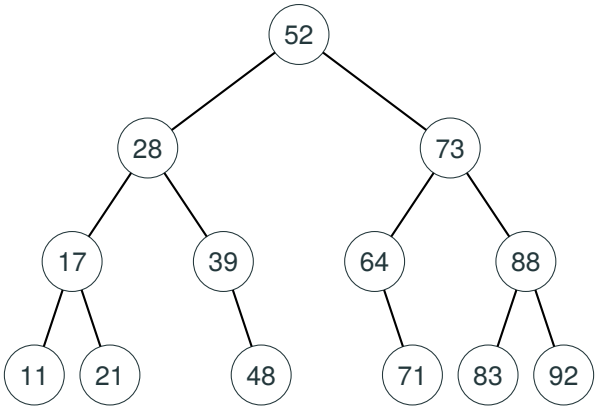
Delete(T, z), $z = 55$



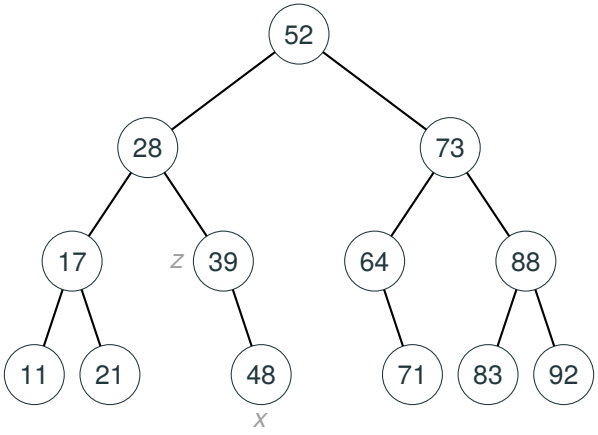
Delete(T, z), $z = 30$



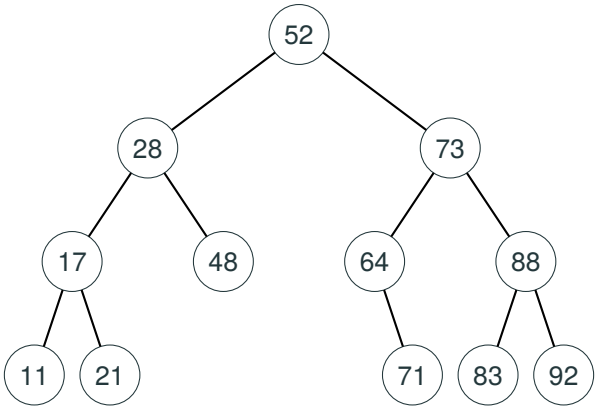
Delete(T, z), $z = 30$



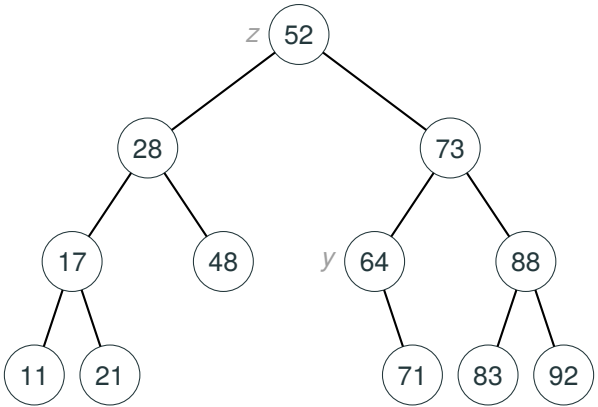
$Delete(T, z)$, $z = 39$, $x = z.right = 48$



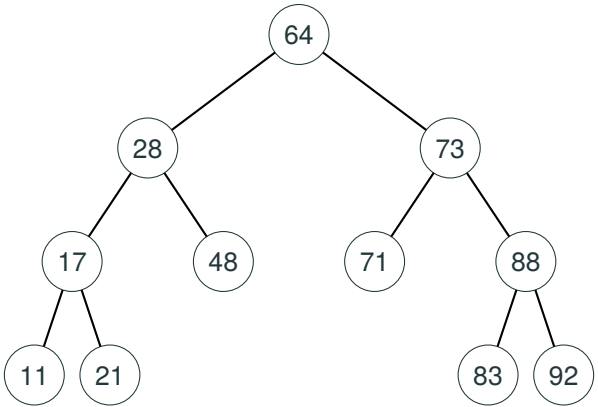
$Delete(T, z)$, $z = 39$, $x = z.right = 48$



$Delete(T, z)$, $z = 52$, $y = Predecessor(T, z) = 64$



$Delete(T, z)$, $z = 52$, $y = Predecessor(T, z) = 64$



Algorithmus Delete

Delete(T, z)

```
 1 If (z.left = NIL) ∨ (z.right = NIL)
 2    then y ← z else y ← Successor(T, z)
 4 If (y.left ≠ NIL)
 5    then x ← y.left else x ← y.right
 7 If (x ≠ NIL)
 8    then x.p ← y.p
 9 If (y.p = NIL)
10    then T.root ← x
11    elseif y = y.p.left
12         then y.p.left ← x else y.p.right ← x
13 If (y ≠ z) then z.key ← y.key
```

Zusammenfassung Binäre Suchbäume

- Alle Operationen auf binären Suchbäumen T laufen in Zeit $O(\mathit{height}(T))$.
- **Gute Nachricht:** Zufällige Suchbäume mit n inneren Knoten haben eine erwartete Tiefe von $O(\log(n))$.

  Genauer: Wenn man die Schlüssel $\pi(1), \cdots, \pi(n)$ für zufällig gewählte $\pi \in \mathcal{S}_n$ in dieser Reihenfolge einfügt, so ist die erwartete Tiefe $O(\log(n))$ (ohne Beweis).
- **Schlechte Nachricht:** Die Worst Case Tiefe von Suchbäumen mit n inneren Knoten ist $n - 1$. Ein böswilliger Gegenspieler kann den Suchbaum unbalanciert und damit langsam machen.
- **Ausweg:** Der Suchbaum wird nach jedem Einfügen und Streichen eines Knotens wieder ausbalanciert, so dass der Suchbaum stets hinreichend balanciert bleibt. Dafür gibt es verschiedene Strategien (B-Bäume, Rot-Schwarz-Bäume, 2-3-Bäume, ...).
- Wir betrachten im Folgenden **Rot-Schwarz-Bäume (Red-Black-Trees)**.

Binäre Suchbäume

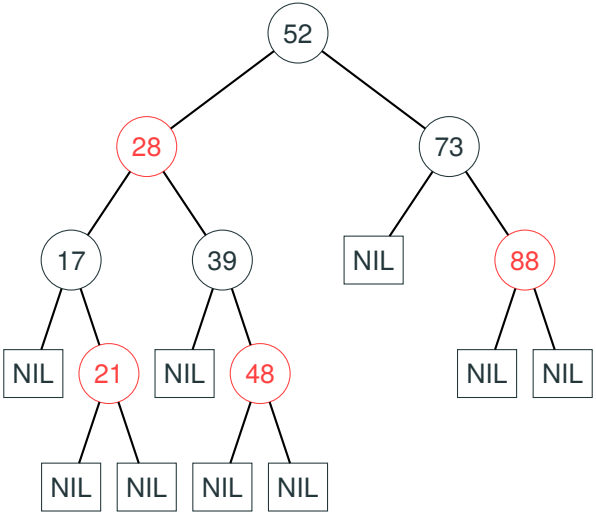
Rot Schwarz Bäume

Rot Schwarz Bäume (Red Black Trees, RB-Bäume)

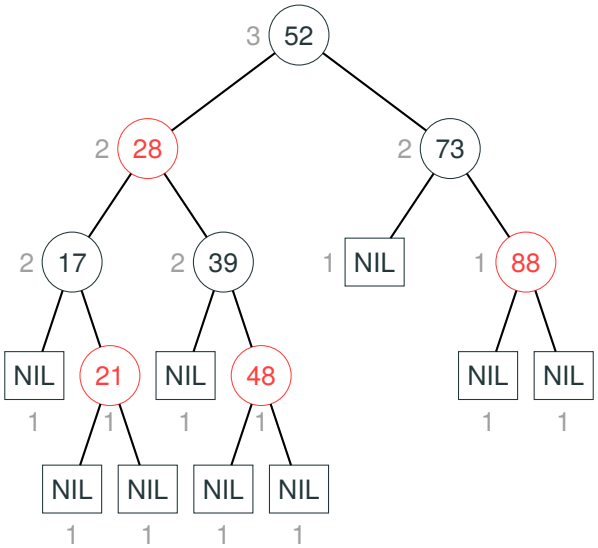
Definition 42

1. RB-Bäume sind binäre Suchbäume, deren innere Knoten x ein zusätzliches Attribut $x.color \in \{red, black\}$ haben.
2. Alle Blätter tragen die Bezeichnung *NIL*, d.h. alle inneren Knoten haben Ausgrad 2.
3. Die Wurzel sowie alle Blätter sind schwarz.
4. Die Söhne roter Knoten sind schwarz.
5. Jeder Weg von der Wurzel $T.root$ zu einem Blatt hat die gleiche Anzahl schwarzer Knoten.
6. $bheight(x)$ bezeichnet die Anzahl schwarzer innerer Knoten auf einem Weg vom Knoten x zu einem Blatt (diese Anzahl ist wegen Regel 5 für alle diese Wege gleich), $bheight(T) = bheight(T.root)$.

Beispiel RB-Baum



Beispiel RB-Baum mit *bheight*



Balanciertheit von RB-Bäumen

Theorem 43

Jeder RB-Baum T mit n inneren Knoten hat höchstens die Tiefe $2 \cdot \log_2(n+1)$.

Der **Beweis von Theorem 43** nutzt

Lemma 44

Für jeden RB-Baum T und jeden Knoten x in T gilt: Der Teilbaum T_x mit Wurzel x von T hat mindestens $2^{bheight(x)-1} - 1$ innere Knoten.

Der **Beweis von Lemma 44** erfolgt per Induktion über die Höhe h von x.

Für $h = 0$ besteht T_x nur aus einem schwarzen NIL-Blatt, das heißt $bheight(x) = 1$ und T_x hat $0 = 2^{1-1} - 1$ innere Knoten.

Beweis von Theorem 43

Für $h > 0$ gilt, dass der linke Nachfolger y und der rechte Nachfolger z von x höchstens die Höhe $h - 1$ und mindestens die Schwarzhöhe $bheight(x) - 1$ haben.

Konkret ist $bheight(y) = bheight(x)$ falls x rot und $bheight(y) = bheight(x) - 1$ falls x schwarz, dasselbe gilt für z.

Nach Induktionsvoraussetzung ist die Anzahl innerer Knoten in T_x mindestens

$$2 \cdot \left(2^{bheight(T)-2} - 1\right) + 1 = 2^{bheight(T)-1} - 1. \square$$

Beweis von Theorem 43: Wegen Regel 4 in Definition 42 gilt $depth(T) \leq 2 \cdot bheight(T) - 2$, also $bheight(T) \geq \frac{depth(T)+2}{2}$.

Das ergibt $n \geq 2^{\frac{depth(T)+2}{2}-1} - 1$, also $\log_2(n+1) \geq \frac{depth(T)+2}{2} - 1 = \frac{depth(T)}{2}$

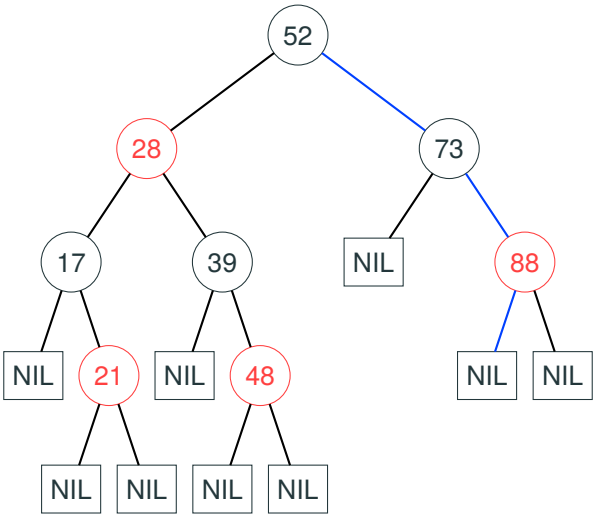
Damit gilt $depth(T) \leq 2 \cdot \log_2(n+1)$. $\square$

Operationen auf RB-Bäumen

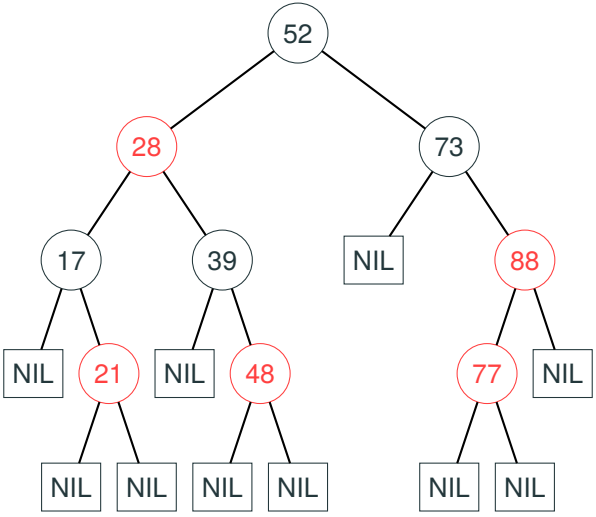
- Die Operationen Search, Minimum, Maximum, Successor und Predecessor gehen wie bei einfachen Binären Suchbäumen.
- *RBInsert*(T, x) fügt x gemäß *Insert*(T, x) in T ein und färbt x rot.

  Dadurch kann ein **Rotfehler** entstehen, d.h. $x.color = x.p.color = RED$.
- *RBDelete*(T, z) streicht z aus T gemäß *Delete*(T, z). Dadurch kann ein **Schwarzfehler** entstehen, d.h. es entsteht eine Knoten x, der eine um Eins kleinere Schwarzhöhe hat als sein Geschwisterknoten.
- *RBInsert*(T, x) und *RBDelete*(T, z) beseitigen die Fehler in der nachfolgenden *FixUp* Phase in Zeit $O(\log(|T|))$ unter Nutzung von **Rotationen**.
- Die Laufzeit aller Operationen ist $O(\log(|T|))$.

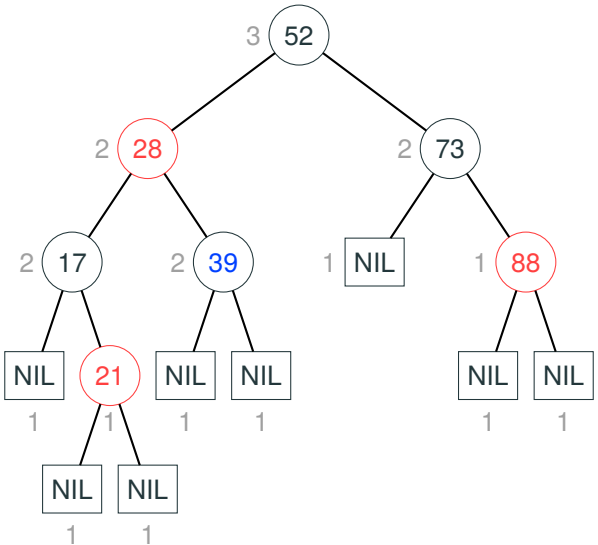
Beispiel Rotfehler *Insert*$(T, 77)$, 1



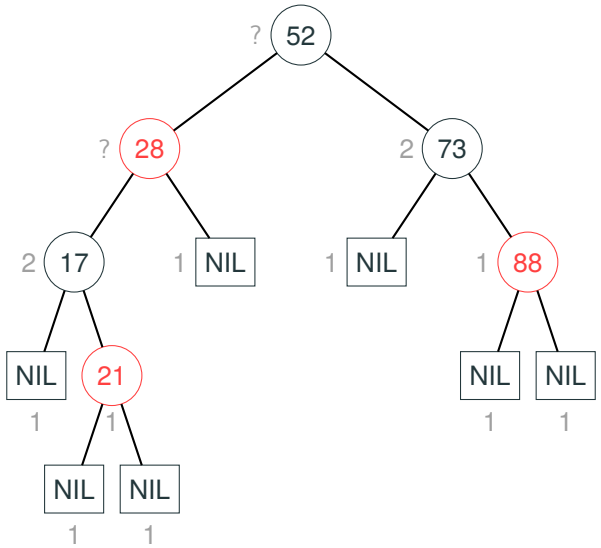
Beispiel Rotfehler *Insert*$(T, 77)$, 2



Beispiel Schwarzfehler *Delete*$(T, 39)$**, 1**



Beispiel Schwarzfehler *Delete*$(T, 39)$, 2



Rotationen

RightRotate(T, y) ist anwendbar falls $y.left = x \neq NIL$

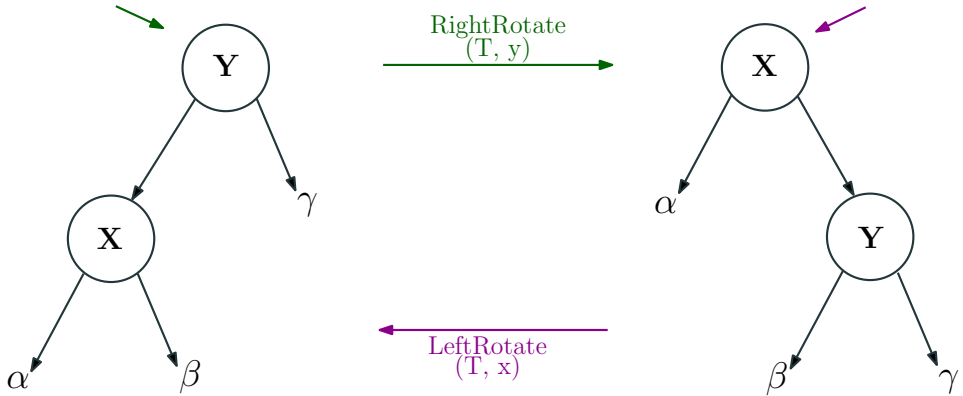
- Es seien α bzw. β linker bzw. rechter Teilbaum an x und γ rechter Teilbaum an y.
- Nach *RightRotate*(T, y) gilt $x.right = y$ und β wird linker Teilbaum von y.

LeftRotate(T, x) ist die Umkehroperation zu *RightRotate*(T, y) und ist anwendbar falls $x.right = y \neq NIL$.

- Es sei α linker Teilbaum an x, β und γ seien linker bzw. rechter Teilbaum an y.
- Nach *LeftRotate*(T, x) gilt $y.left = x$ und β wird rechter Teilbaum von x.

Wichtig: Rotationen erhalten die Suchbaumeigenschaft!

Rotationen



LeftRotate(T, x)

```
 1  y ← x.right
 2  x.right ← y.left
 3  If y.left ≠ NIL then y.left.p ← x
 4  y.p ← x.p
 5  If x.p = NIL then T.root ← y
 6     elseif x = x.p.left
 7           then x.p.left ← y
 8           else x.p.right ← y
 9  y.left ← x
10  x.p ← y
```

Laufzeit $O(1)$.

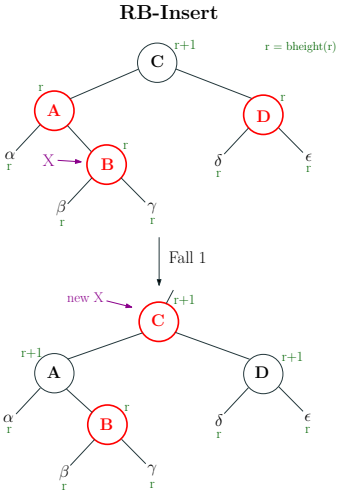
RBInsert(T, x), informale Beschreibung Teil 1

- *RBInsert*(T, x) führt zunächst das normale *Insert*(T, x) aus. Ist $T = \emptyset$ so wird x (als Wurzel) schwarz gefärbt. Ansonsten wird x rot gefärbt.
- Gilt dass $x.p.color = BLACK$ (d.h., x wird an einen schwarzen Knoten gehangen), so ist nichts weiter zu tun.
- Anderenfalls entsteht ein Rotfehler an x, d.h., es gilt $x.p.color = x.color = RED$.
- In der folgenden *RBFixup*-Phase wird dieser Rotfehler schrittweise beseitigt. Zu Beginn jeden Schritts hat der Baum für genau einen Knoten x einen Rotfehler an x.
- Je nach Färbung der Umgebung von x wird dieser Rotfehler in Richtung Wurzel verschoben (Fall 1) oder durch Rotationen und Umfärbung beseitigt (Fall 2 und 3).

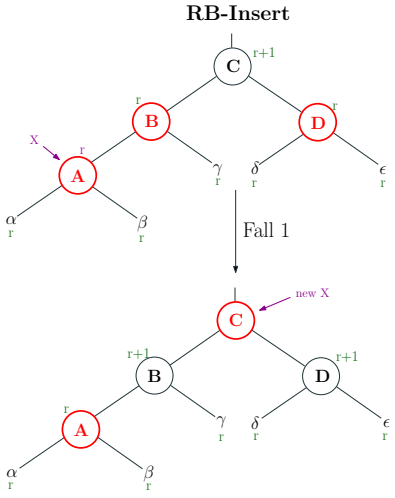
RBInsert(T, x), informale Beschreibung Teil 2

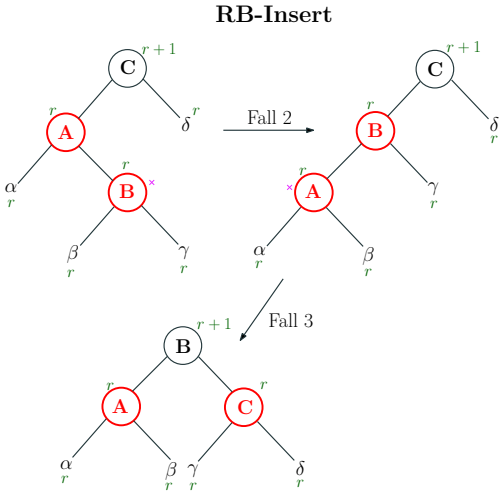
- Fall 1 tritt ein, wenn der Geschwisterknoten w von $x.p$ rot ist.
- Dann werden $x.p$ und y schwarz, und $x.p.p$ rot gefärbt. Dadurch bleibt die Balanciertheit des Baumes erhalten (siehe Bild).
- Sollte $x.p.p$ die Wurzel sein, so kann $x.p.p$ wieder schwarz gefärbt werden, und man erhält einen korrekten RB-Baum.
- Anderenfalls kann ein Rotfehler an $x.p.p$ entstehen (wenn $x.p.p.p$ rot ist). Dann wird x auf $x.p.p$ gesetzt.
- Ist der Geschwisterknoten w von $x.p$ schwarz, so kann durch zwei Rotationen (Fall 2) und Umfärben bzw. durch eine Rotation (Fall 3) und Umfärben der Rotfehler beseitigt werden (siehe Bilder).

RB-Insert



RB-Insert


RB-Insert
C
r+1
B
r
D
r
X
A
r
γ
r
δ
r
ϵ
r
α
r
β
r
Fall 1
new X
C
B
r+1
D
r+1
A
r
γ
r
δ
r
ϵ
r
α
r
β
r


RB-Insert
C
r + 1
A
r
δ
r
α
r
B
r
β
r
γ
r
Fall 2
C
r + 1
B
r
δ
r
A
r
α
r
γ
r
β
r
Fall 3
B
r + 1
A
r
C
r
α
r
β
r
γ
r
δ
r

$RBInsert(T, x)$

```
1  Insert(T, x)
2  x.color ← RED
3  while (x ≠ T.root) ∧ (x.p.color = RED)
4      do if x.p = x.p.p.left
5          then y ← x.p.p.right
6              if y.color = RED (*Fall 1*)
7                  then x.p.color ← BLACK
8                       y.color ← BLACK
9                       x.p.p.color ← RED
10                      x ← x.p.p
```

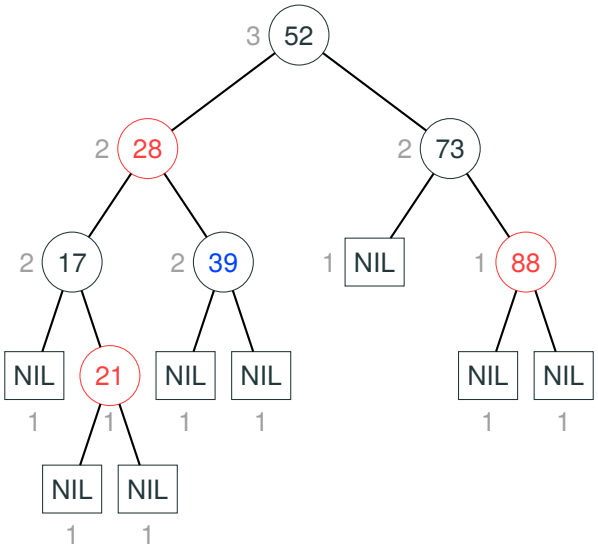
RBInsert(T, x), Fortsetzung

```
11              else (*also y.color = BLACK*)
12                  if x = x.p.right (*Fall 2*)
13                    then x ← x.p
14                         LeftRotate(T, x)
15                  x.p.color ← BLACK (*Fall 3*)
16                  x.p.p.color ← RED
17                  RightRotate(T, x.p.p)
18          else (* also x.p = x.p.p.right*)
19              mache das Gleiche wie im then Teil,
20              nur left und right vertauscht.
21  T.root.color ← BLACK
```

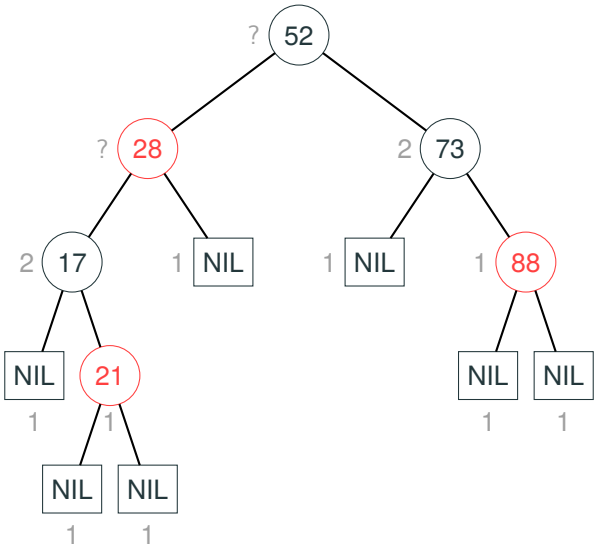
RBDelete(T, z), informale Beschreibung

- *RBDelete*(T, z) führt zunächst *Delete*(T, z) aus. Dabei wird physisch ein innerer Knoten y aus T entfernt, der mindestens einen NIL-Nachfolger hat.
- Ist y rot, so sind beide Söhne NIL-Blätter und y wird durch ein schwarzes NIL-Blatt ersetzt, ohne die Balanciertheit zu stören.
- Hat y einen roten Sohn x, so wird y durch x ersetzt und x schwarz gefärbt, ohne die Balanciertheit zu stören.
- Im verbleibenden Fall ist y schwarz und hat zwei NIL-Söhne. Dann wird y wird durch ein schwarzes NIL-Blatt x ersetzt.
- Allerdings ergibt sich an x ein Schwarzfehler, der durch die Platzierung eines *ExtraBlack* an x künstlich ausgeglichen wird.

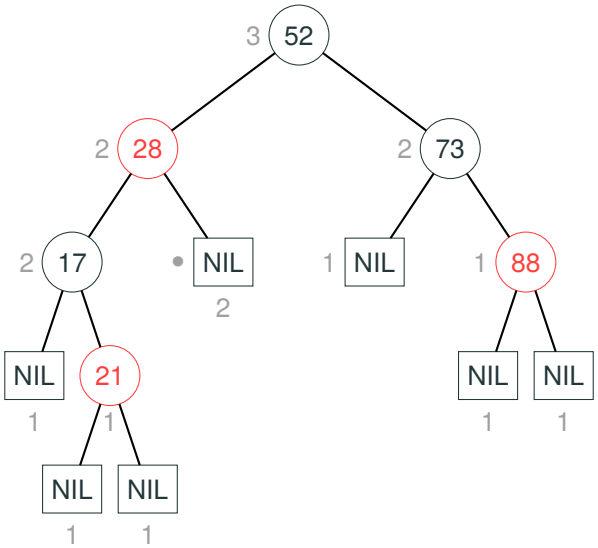
Nochmal Beispiel Schwarzfehler *Delete*$(T, 39)$**, 1**



Nochmal Beispiel Schwarzfehler *Delete*$(T, 39)$**, 2**



Nochmal Beispiel Schwarzfehler *Delete*$(T, 39)$, 3

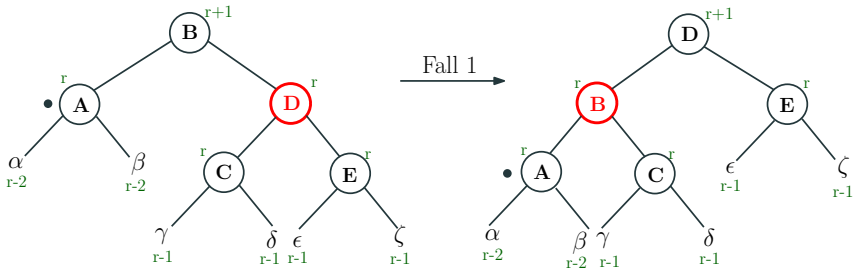


RBDelete(T, z) informale Beschreibung, Fortsetzung

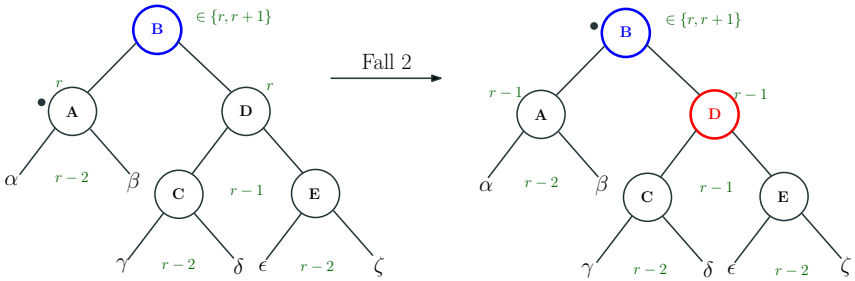
- *RBFixUp* beseitigt das *ExtraBlack* schrittweise. Zu Beginn jedes Schrittes liegt auf genau einem Knoten x das *ExtraBlack*.
- Landet das *ExtraBlack* auf einem roten Knoten, so wird dieser schwarz gefärbt, fertig. Landet es auf der Wurzel, so kann es weggenommen werden.
- Ansonsten wird das *ExtraBlack* in einem Schritt entweder Richtung Wurzel verschoben (Fall 1 und 2) oder durch Rotationen und Umfärbungen beseitigt (Fall 3 und 4).
- Jeder Schritt von sowohl *RBDelete* als auch *RBInsert* benötigt Zeit $O(1)$. Da höchstens $O(\mathit{depth}(T))$ Schritte nötig sind, beträgt die Gesamtlaufzeit $O(\log |T|)$.

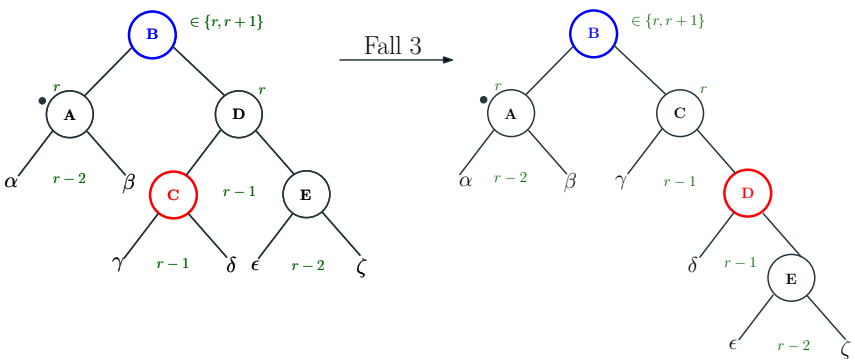
RB-Delete

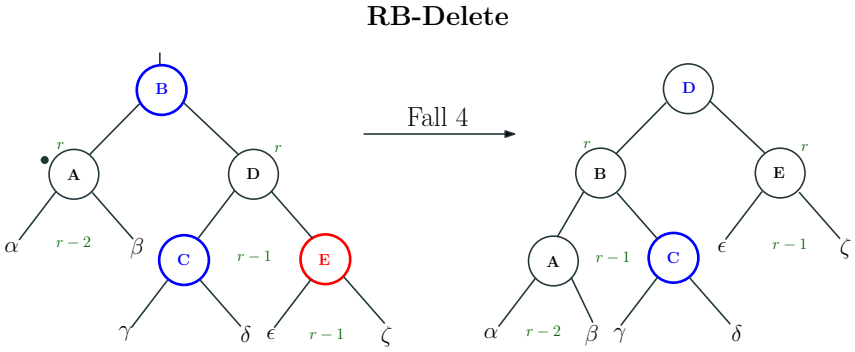
RBDelete(T, x)



RB-Delete


B
$\in \{r, r+1\}$
A
D
C
E
r
r
$r-2$
$r-1$
$r-2$
$r-2$
α
β
γ
δ
ϵ
ζ
Fall 2
B
$\in \{r, r+1\}$
A
D
C
E
$r-1$
$r-1$
$r-2$
$r-1$
$r-2$
$r-2$
α
β
γ
δ
ϵ
ζ


RB-Delete
B
$\in \{r, r+1\}$
r
A
r
D
α
$r-2$
β
C
$r-1$
E
γ
$r-1$
δ
ϵ
$r-2$
ζ
Fall 3
B
$\in \{r, r+1\}$
r
A
r
C
α
$r-2$
β
γ
$r-1$
D
δ
$r-1$
E
ϵ
$r-2$
ζ


RB-Delete
B
A
D
C
E
α
β
γ
δ
ϵ
ζ
r
r − 1
r − 2
Fall 4
D
B
E
A
C
α
β
γ
δ
ϵ
ζ
r
r − 1
r − 2

$RBDelete(T, z)$

```
 1 If (z.left = NIL) ∨ (z.right = NIL)
 2    then y ← z else y ← Successor(T, z)
 4 If (y.left ≠ NIL)
 5    then x ← y.left else x ← y.right
 6 If (x ≠ NIL) then x.p ← y.p
 7 If (y.p = NIL)
 8    then T.root ← x
 9    elseif y = y.p.left
10          then y.p.left ← x else y.p.right ← x
11 If (y ≠ z) then z.key ← y.key
12 If (y.color = BLACK) then RBFixup(T, x)
```

$RBFixup(T, x)$

```
1  while (x ≠ T.root) ∧ (x.color = BLACK)
2      do if (x = x.p.left)
3          then w ← p.x.right
4              if w.color = RED (*1*)
5                  then w.color ← BLACK
6                       x.p.color ← RED
7                       LeftRotate(T, x.p)
8                       w ← x.p.right
9              if w.left.color = w.right.color = BLACK (*2*)
10                 then w.color ← RED
11                      x ← p.x
12                 else
```

RBFixup(T, x) **Fortsetzung**

```
13              else if w.right.color = BLACK (*3*)
14                   then w.left.color ← BLACK
15                        w.color ← RED
16                        RightRotate(T, w)
17                        w ← x.p.right
18                   w.color ← x.p.color (*4*)
19                   x.p.color ← BLACK
20                   w.right.color ← BLACK
21                   LeftRotate(T, x.p)
22                   x ← T.root
23        else (*also x = x.p.right*) mache das Gleiche wie
24             im then Teil, nur left und right vertauscht.
25  x.color ← BLACK
```

Die elementaren Graphalgorithmen Breiten- und Tiefensuche

Datenstrukturen für Graphen

Sei $G = (V, E)$ ein Graph über der endlichen Knotenmenge $V = \{v_1, \cdots, v_n\}$ und der Kantenmenge $E = \{e_1, \cdots, e_m\} \subseteq V \times V$.

Wichtige Datenstrukturen für Graphen sind

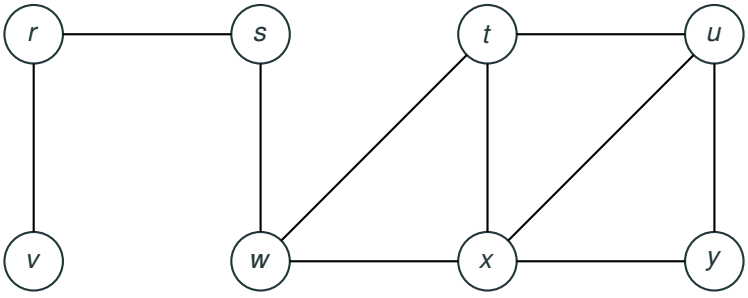
- **Die Kantenliste,** d.h. die Liste aller Kanten von G,
- **Die Adjazenzmatrix** $A(G)$, eine $n \times n$ Matrix über $\{0, 1\}$, definiert durch

$$A(G)_{i,j} = 1 \iff (v_i, v_j) \in E.$$

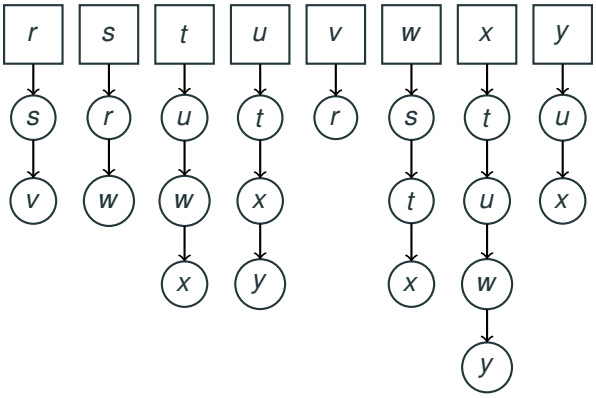
- **Die Adjazenzliste,** d.h. ein Feld $G.Adj = (G.Adj[v])_{v \in V}$, wobei $G.Adj[v]$ auf eine Liste zeigt, die alle zu v adjazenten Knoten w, d.h. alle Knoten w mit $(v, w) \in E$, enthält.

Wichtige Hilfsdatenstrukturen für Graphalgorithmen sind **Queues** und **Stacks**.

Beispielgraph (ungerichtet)



Zugehörige Adjazenzliste



Queues (Warteschlangen)

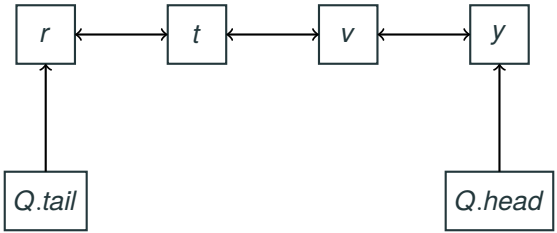
... sind **First-In-First-Out (FIFO)** Speicher, die durch **Felder** oder durch **doppelt verkettete Listen** realisiert werden können.

Wir beschreiben hier eine Realisierung als doppelt verkettete Liste Q mit Zeigern $Q.tail$ (Ende der Schlange) und $Q.head$ (Kopf der Schlange) und den Operationen $Enqueue(Q, x)$ (x stellt sich hinten an) und $Dequeue(Q)$ (der vorderste Kunde wird bearbeitet und verlässt die Warteschlange).

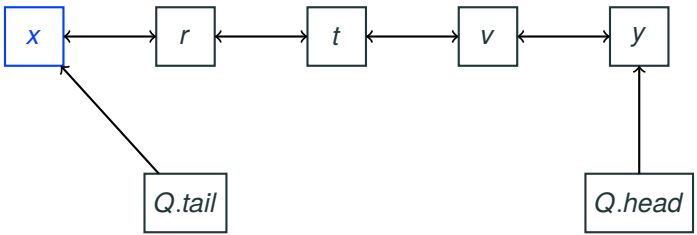
$Enqueue(Q, x)$

```
1 If Q.tail = NIL
2   then Q.head ← Q.tail ← x
3        x.prev ← x.succ ← NIL
4   else x.prev ← Q.tail
5        x.succ ← NIL
6        Q.tail.succ ← x
7        Q.tail ← x
```

Beispiel Queue Q vor $Enqueue(Q, x)$



Beispiel Queue Q nach $Enqueue(Q, x)$

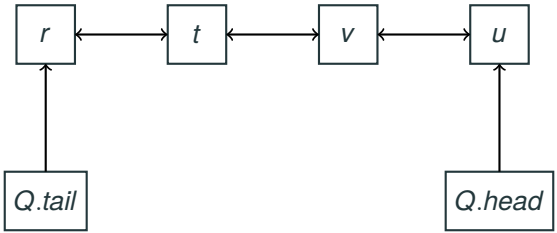


Dequeue(Q)

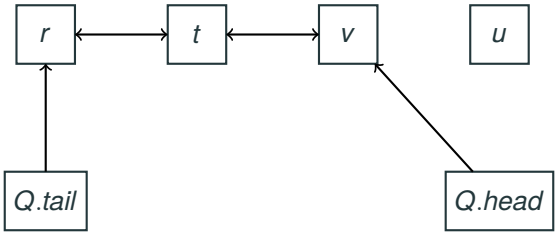
Dequeue(Q)

```
1  u ← Q.head
2  If u ≠ NIL
3    then Q.head ← Q.head.succ
4         If Q.head = NIL
5           then Q.tail ← NIL
6           else Q.head.prev ← NIL
7  return u
```

Beispiel Queue Q vor $Dequeue(Q)$



Beispiel Queue Q nach *Dequeue*(Q)



Stacks (Kellerspeicher)

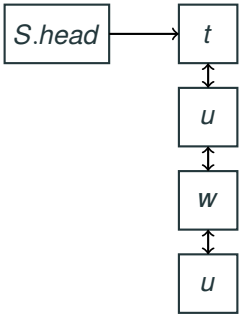
... sind **Last-In-First-Out (LIFO) Speicher**, die durch **Felder** oder **doppelt verkettete Listen** realisiert werden können.

Wir beschreiben die Realisierung als (vertikale) doppelt verkettete Liste S mit Zeigern $S.head$, der Insert Operation $Push(S, x)$ und der Delete-Operation $Pop(S)$.

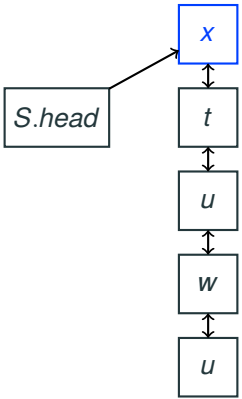
$Push(S, x)$

```
1  If S.head = NIL
2    then S.head ← x
3         x.above ← x.under ← NIL
4    else x.under ← S.head
5         x.above ← NIL
6         S.head.above ← x
7         S.head ← x
```

Stack S vor $Push(S, x)$



Stack S nach $Push(S, x)$

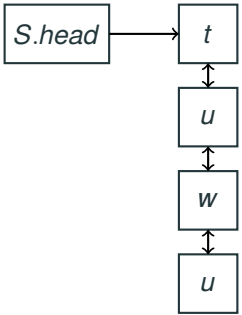


Stack Delete Operation $Pop(S)$

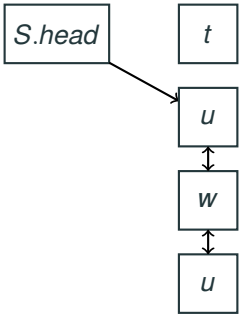
$\underline{Pop(S)}$

```
1 v ← S.head
2 If v ≠ NIL
3   then S.head ← S.head.under
4        If S.head ≠ NIL
5          then S.head.above ← NIL
6 return v
```

Stack S vor $Pop(S)$



Stack S nach $Pop(S)$



Breitensuche $BFS(G,s)$ in Graphen $G = (G.V, G.E)$

Definition 45

Für alle Knoten s, v eines Graphen $G = (V, E)$ bezeichnet $\delta_G(s, v)$ die **Länge eines kürzesten Weges von s nach v** in G.

- $BFS(G,s)$ durchsucht G von einem fixiertem Startpunkt $s \in V$ aus und besucht **alle Knoten aus der Menge $V(s)$ der von s aus erreichbaren Knoten**.
- $BFS(G,s)$ verwaltet dabei für alle $v \in V(s)$ dynamisch einen Wert $v.d \in \mathbb{N} \cup \{\infty\}$.
- Nach Abschluss von $BFS(G,s)$ gilt $v.d = \delta_G(s,v)$.
- Zudem verwaltet $BFS(G,s)$ für jeden Knoten $v \in V(s)$ dynamisch einen Wert $v.\pi \in V \cup \{NIL\}$, und damit einen Baum mit Wurzel s

$$T = \{(v.\pi, v); v \in V(s) \setminus \{s\}, v.\pi \neq NIL\} \subseteq G.E.$$

- Nach Abschluss von $BFS(G,s)$ ist T ist ein **Kürzester-Wege-Baum mit Wurzel s**, d.h. $\delta_T(s,v) = v.d = \delta_G(s,v)$ für alle $v \in V(s)$.

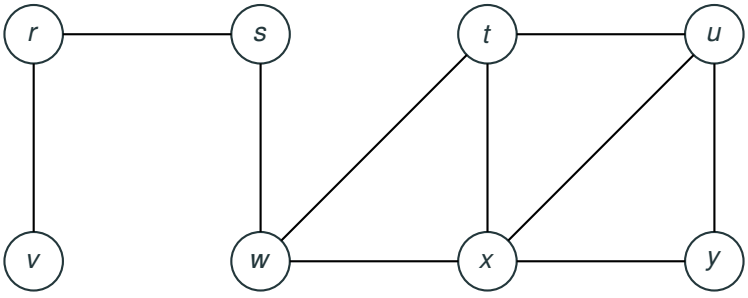
BFS(G,s)

$BFS(G, s)$ benutzt G als Adjazenzliste $G.Adj$ und eine Queue Q und verwaltet für alle $v \in V$ dynamisch eine weitere Komponente $v.color \in \{WHITE, GRAY, BLACK\}$.

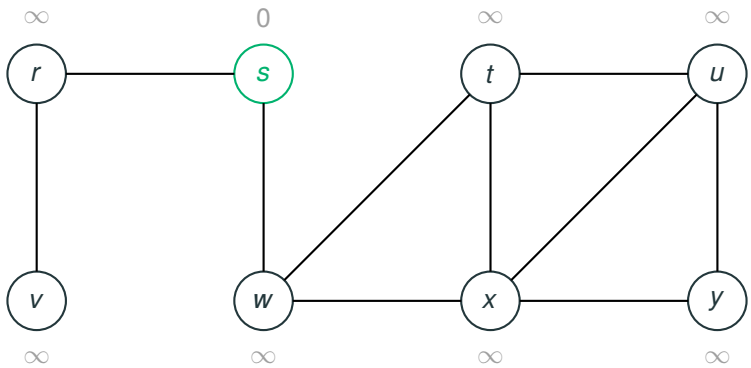
BFS(G,s)

```
 1 For each u ∈ V \ {s}
 2    do u.color ← WHITE, u.d ← ∞, u.π ← NIL
 3 s.color ← GRAY, s.d ← 0, s.π ← NIL
 4 Q ← ∅
 5 Enqueue(Q, s)
 6 while Q ≠ ∅
 7    do u ← Dequeue(Q)
 8       For each v ∈ G.Adj[u]
 9          do if v.color = WHITE
10             then v.color ← GRAY, v.d ← u.d + 1, v.π ← u, Enqueue(Q, v)
11       u.color ← BLACK
```

Beispiel G für $BFS(G, s)$



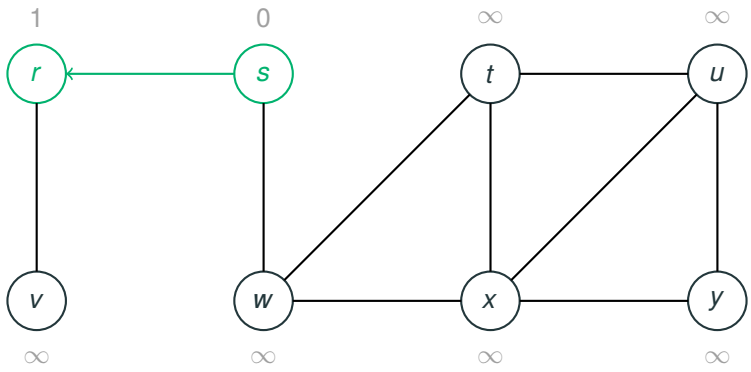
$BFS(G, s)$ **nach Initialisierung**



Queue $Q = (s)$

grey=green, black=blue

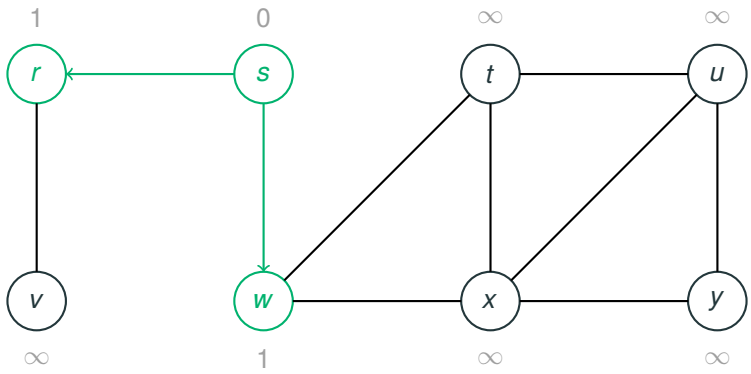
$BFS(G, s)$, **1**



Queue $Q = (r, s)$

grey nodes=green, black nodes =blue, T-edges=green

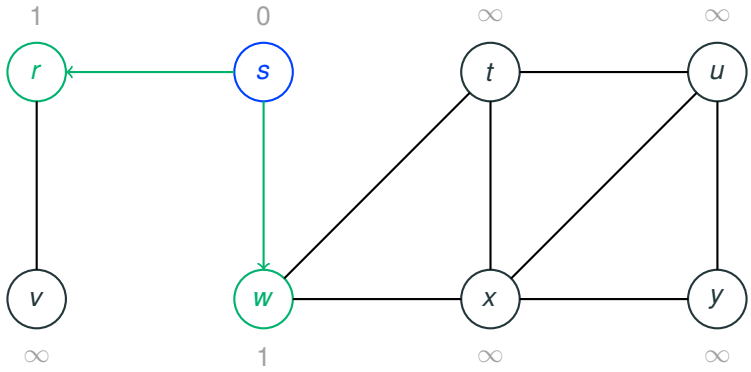
$BFS(G, s)$, 2



Queue $Q = (w, r, s)$

grey nodes=green, black nodes =blue, T-edges=green

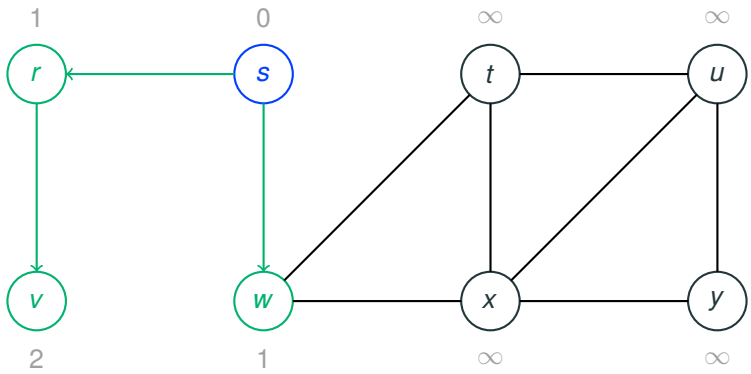
$BFS(G, s)$, 3



Queue $Q = (w, r)$

grey nodes=green, black nodes =blue, T-edges=green

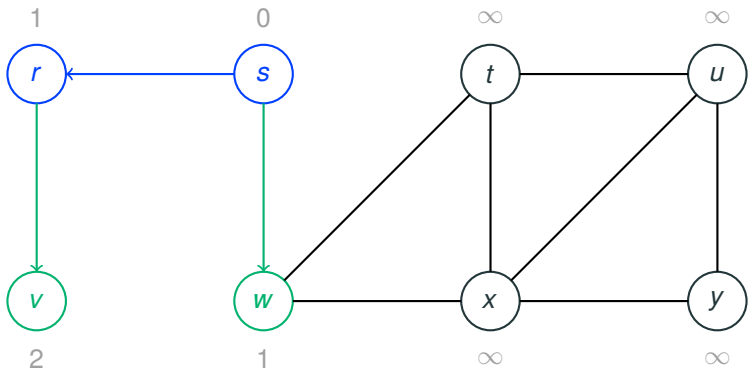
$BFS(G, s)$, 4



Queue $Q = (v, w, r)$

grey nodes=green, black nodes =blue, T-edges=green

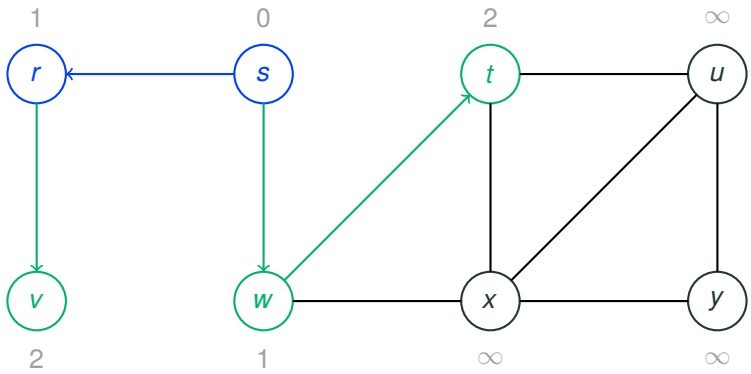
$BFS(G, s)$, 5



Queue $Q = (v, w)$

grey nodes=green, black nodes =blue, T-edges=green or blue

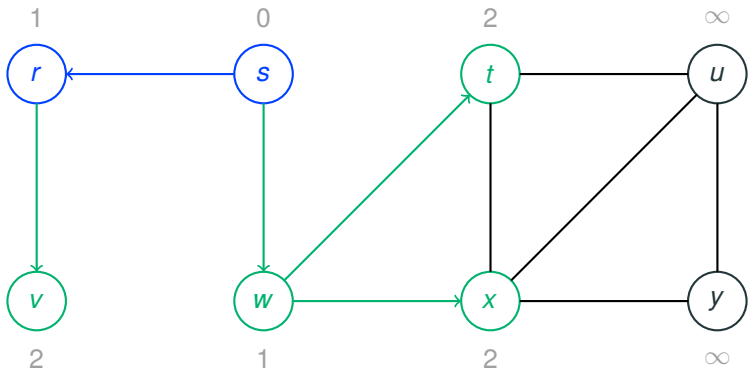
$BFS(G, s)$, 6



Queue $Q = (t, v, w)$

grey nodes=green, black nodes =blue, T-edges=green or blue

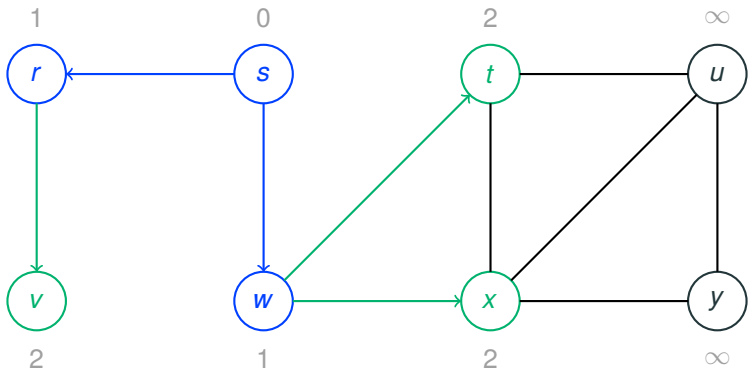
$BFS(G, s)$, 7



Queue $Q = (x, t, v, w)$

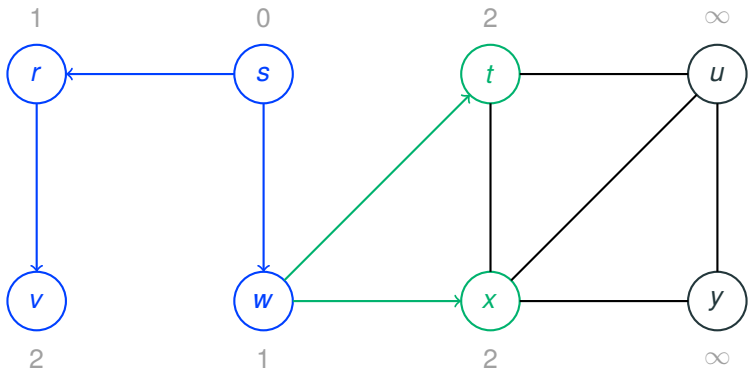
grey nodes=green, black nodes =blue, T-edges=green or blue

$BFS(G, s)$, 8



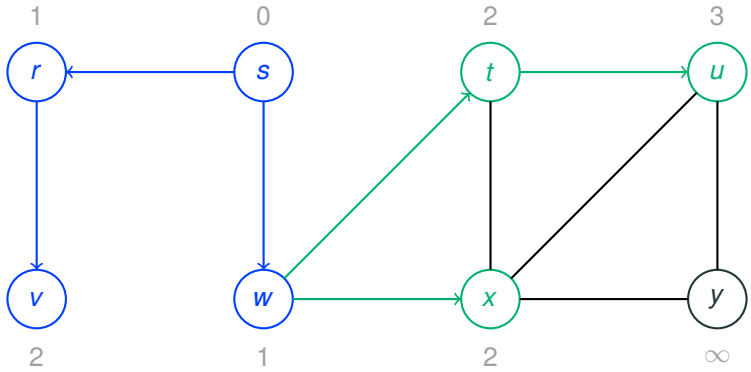
Queue $Q = (x, t, v)$

grey nodes=green, black nodes =blue, T-edges=green or blue



Queue $Q = (x, t)$

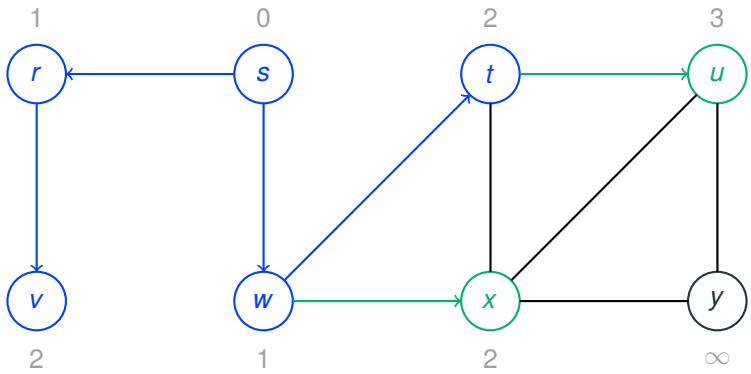
grey nodes=green, black nodes =blue, T-edges=green or blue



Queue $Q = (u, x, t)$

grey nodes=green, black nodes =blue, T-edges=green or blue

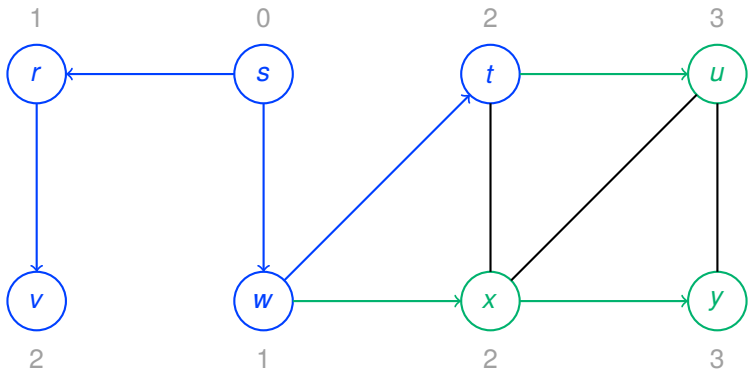
$BFS(G, s)$, 11



Queue $Q = (u, x)$

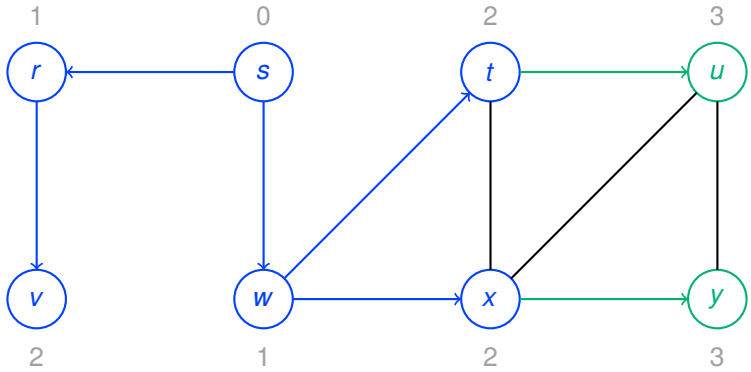
grey nodes=green, black nodes =blue, T-edges=green or blue

$BFS(G, s)$, 12



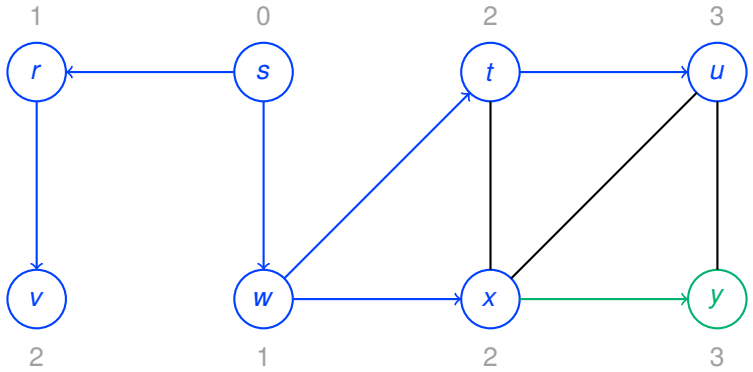
Queue $Q = (y, u, x)$

grey nodes=green, black nodes =blue, T-edges=green or blue



Queue $Q = (y, u)$

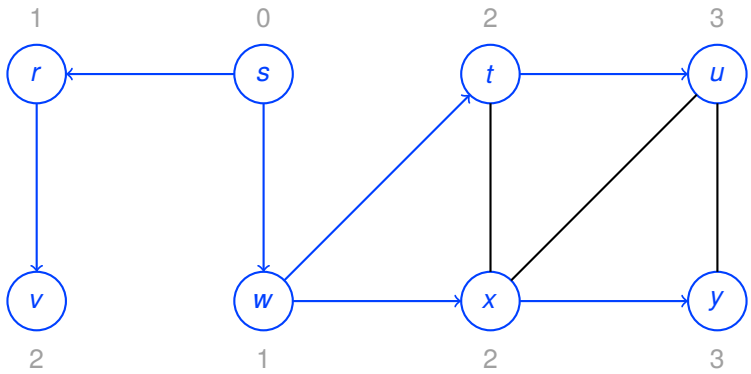
grey nodes=green, black nodes =blue, T-edges=green or blue



Queue $Q = (y)$

grey nodes=green, black nodes =blue, T-edges=green or blue

$BFS(G, s)$, 15



Queue $Q = \emptyset$

black nodes =blue, T-edges=blue

BFS: Laufzeit und Korrektheit

Die **Laufzeit von** $BFS(G, s)$ ist $O(|V| + |E|)$, die Initialisierung kostet $O(|V|)$, die Bearbeitungszeit pro Kante ist $O(1)$.

Korrektheit: Für alle Knoten v, die von $BFS(G, s)$ besucht werden, gebe $nr(v)$ an, als wievielter Knoten v entdeckt, d.h. in die Queue eingefügt wird.

Lemma 46

Für alle von $BFS(G, s)$ besuchten Knoten v, w gilt $nr(v) < nr(w) \Longrightarrow v.d \leq w.d$.

Beweis: Annahme, das Lemma ist falsch. Wir wählen den Knoten v mit minimaler Nummer, für den es einen Knoten w' gibt so dass

$$nr(v) < nr(w') \text{ und } v.d > (w').d \tag{1}$$

Unter allen Knoten w', die (1) erfüllen, sei w der Knoten mit der kleinsten Nummer.

Wegen $s.d = 0$ und $nr(s) = 1$ gilt $0 \leq w.d < v.d$ (also $v \neq s$) und $1 < nr(v) < nr(w)$ (also $w \neq s$).

Der Beweis von Lemma 46

Also existieren $v.\pi$ und $w.\pi$ und es gilt $v.\pi \neq w.\pi$ da $v.d \neq w.d$.

Es gilt aber auch $nr(v.\pi) < nr(w.\pi)$, da v vor w entdeckt wird.

Auf Grund der Minimalität von v folgt daraus $v.\pi.d \leq w.\pi.d$, also $v.d = v.\pi.d + 1 \leq w.\pi.d + 1 = w.d$, Widerspruch. $\square$

BFS Korrektheit

Theorem 47

Für alle $v \in G.V$ gilt nach Abschluss von BFS(G, s), dass $v.d = \delta_G(s, v)$.

Beweis: Wir zeigen zunächst, dass $\delta_G(s, v) \leq v.d$ für alle $v \in V$.

Für alle v mit $v.d = \infty$ ist das trivialerweise wahr.

Für alle v mit $v.d < \infty$ bilden die Kanten $(v.\pi, v), (v.\pi.\pi, v.\pi), \cdots$ einen Weg der Länge $v.d$ von s zu v, woraus direkt $\delta_G(s, v) \leq v.d$ folgt.

Wir müssen zeigen, dass die Menge W aller Knoten w, für die $\delta_G(s, w) < w.d$ gilt, leer ist.

Wir nehmen $W \neq \emptyset$ an und fixieren ein $v \in W$ mit minimalem $\delta_G(s, w) < \infty$.

Sei u der Vorgänger von v auf einem kürzesten Weg von s nach v.

Dann gilt $\delta_G(s, u) = u.d = \delta_G(s, v) - 1$, wegen der Minimalität von v.

Das heißt aber, dass v spätestens bei der Bearbeitung von u entdeckt wird, also $v.d < \infty$.

Der Beweis von Theorem 47

Fall 1: $u = v.\pi$, d.h. v wird von u aus entdeckt.

Dann gilt $v.d = u.d + 1 = \delta_G(s, u) + 1 = \delta_G(s, v)$.

Das steht im Widerspruch zu $v \in W$.

Fall 2: $u \neq v.\pi$, d.h. v wird vor der Bearbeitung von u entdeckt.

Dann gilt $nr(v.\pi) < nr(u)$, d.h. $v.\pi.d \leq u.d$ (Lemma 46).

Folglich gilt auch hier $v.d = v.\pi.d + 1 \leq u.d + 1 = \delta_G(s, v)$. $\square$

Die elementaren Graphalgorithmen Breiten- und Tiefensuche

Tiefensuche (Depth First Search)

Tiefensuche (Depth First Search) $DFS(G)$ in Graphen G

- $DFS(G)$ durchsucht Eingabegraphen G, die als Adjazanzliste $G.Adj$ gegeben sind, nach dem Prinzip des **zuerst in die Tiefe gehen** (Depth First Search)
- $DFS(G)$ benutzt dabei als steuernde Datenstruktur einen **Stack**.
- $DFS(G)$ berechnet einen *DFS Forest* $T_1 \cup \cdots \cup T_s$ mit Wurzeln $u_1, \cdots, u_s$, $s \geq 1$, gegeben durch die Kanten $(v.\pi, v)$, $v \in V \setminus \{u_1, \cdots, u_s\}$.
- $DFS(G)$ berechnet für alle Knoten $v \in G.V$ sogenannte **time stamps** $v.d$ (**discovering time**) und $v.f$ (**finishing time**) und nutzt dazu den Zähler *time*.
- $DFS(G)$ verwaltet dynamisch für jeden Knoten $v \in G.V$ das Attribut $v.color \in \{WHITE, BLACK, GRAY\}$.
- $DFS(G)$ **klassifiziert die Kanten in G als Tree-, Forward-, Back- und Cross-Kanten**.

$DFS(G)$

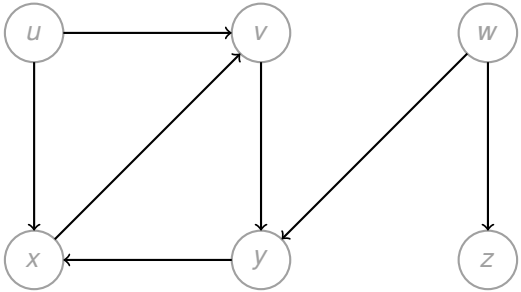
```
1 For each u ∈ G.V
2     do u.color ← WHITE, u.π ← NIL
3 time ← 0; S ← ∅
4 For each u ∈ G.V
5     do if u.color = WHITE
6         then DFSVisit(G, u)
```

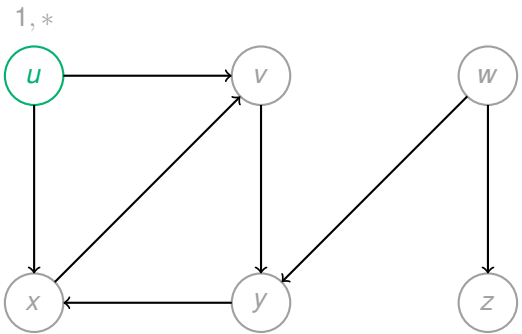
$DFSVisit(G, u)$

```
 1  time ← time + 1
 2  u.d ← time
 3  u.color ← GRAY; push(S, u)
 4  For each v ∈ G.Adj[u]
 5     do if v.color = WHITE
 6           then v.π ← u
 7                DFSVisit(G, v)
 8  time ← time + 1
 9  u.f ← time
10  u.color ← BLACK; pop(S)
```

Anmerkung: Im Stack S wird zur Veranschaulichung die Hierarchie der aktiven rekursiven *DFSVisit*-Aufrufe protokolliert.

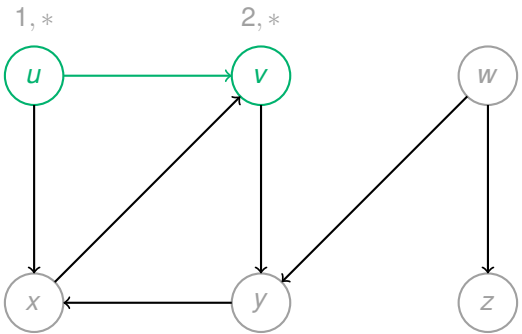
DFS Beispiel





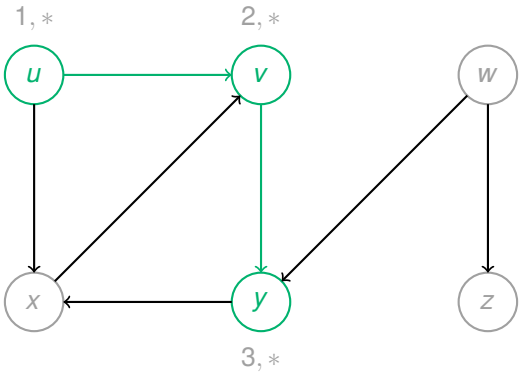
u

$DFS(G) \leftarrow DFSvisit(G, u) \leftarrow Push(v), time = 2$



v

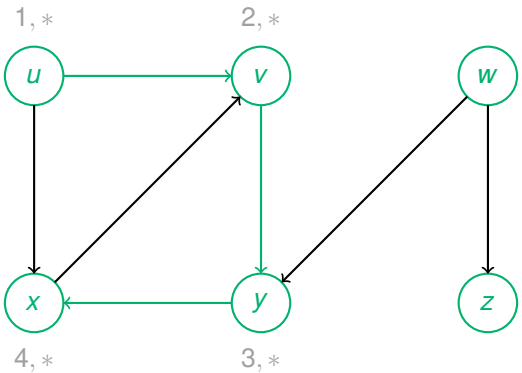
u



y

v

u

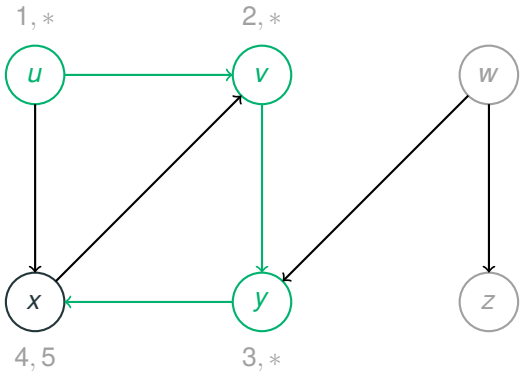


x

y

v

u

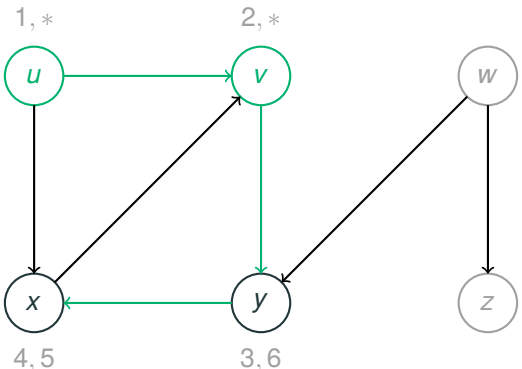


y

v

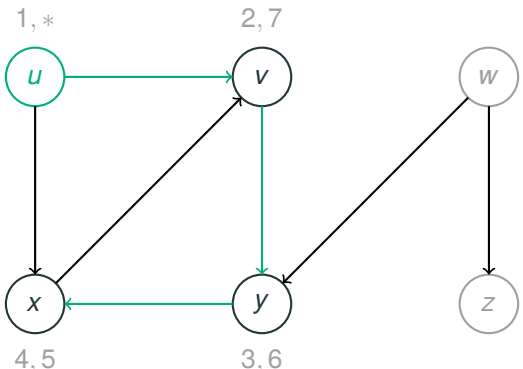
u

$DFS(G) \leftarrow DFSvisit(G, u) \leftarrow Pop, time = 6$



v

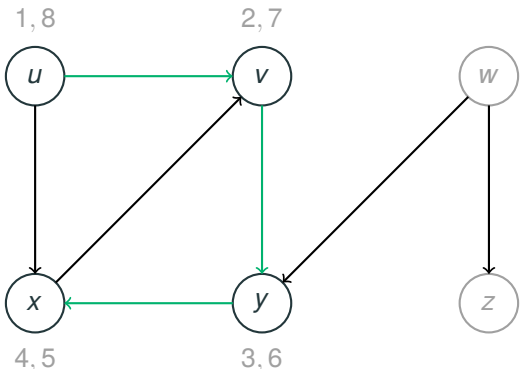
u

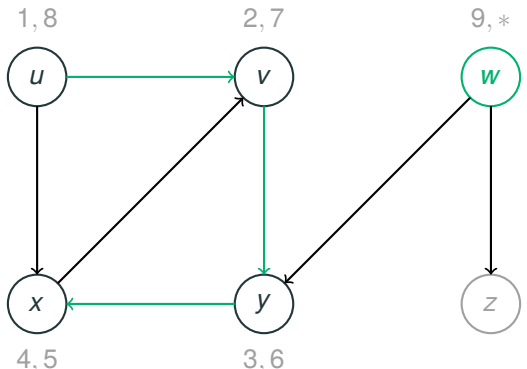


u

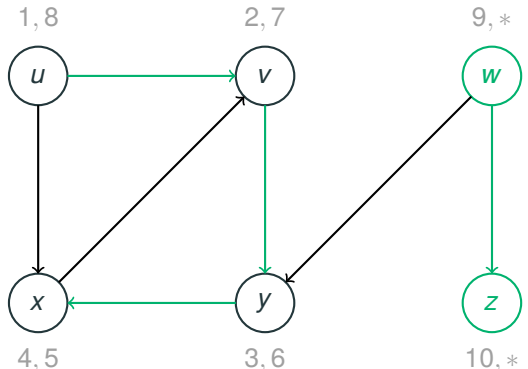
$DFS(G) \leftarrow DFSvisit(G, u) \leftarrow Pop, time = 8$

$\emptyset$



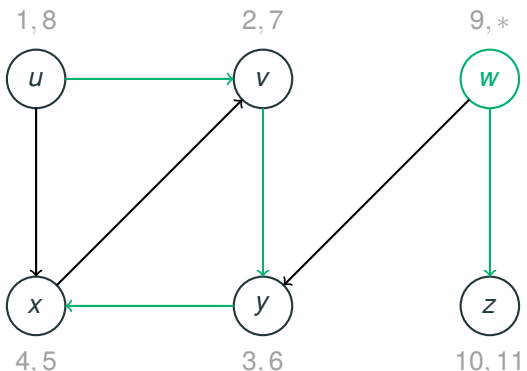


w



z

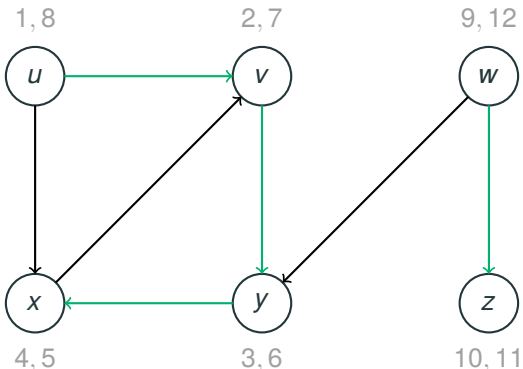
w



w

$DFS(G) \leftarrow DFSvisit(G, w) \leftarrow Pop, time = 12$

$\emptyset$



Eigenschaften von $DFS(G)$

Für alle $v \in G.V$ definiert das **Bearbeitungsintervall** $I(v) = [v.d, v.f]$ von v drei Phasen von $DFS(G)$.

- Ist $time < v.d$, so ist $v \notin S$ und $v.color = WHITE$,
- Ist $time \in I(v)$, so ist $v \in S$ und $v.color = GRAY$,
- Ist $time > v.f$, dann ist $v \notin S$ und $v.color = BLACK$,

Es seien $v \neq w \in G.V$ mit $v.d < w.d$ beliebig fixiert:

- **Fall 1**: Zu $time = w.d$ ist $v.color = GRAY$, d.h. v ist unter w im Stack. Dann ist $v.d < w.d < w.f < v.f$.
- **Fall 2**: Zu $time = w.d$ ist $v.color = BLACK$, d.h. v schon nicht mehr im Stack. Dann ist $v.d < v.f < w.d < w.f$.

Also gilt

Theorem 48

Für alle $v \neq w \in G.V$ gilt entweder $I(v) \cap I(w) = \emptyset$ oder $I(v) \subset I(w)$ oder $I(w) \subset I(v)$. □

Klassifikation von Kanten in G durch $DFS(G)$

$DFS(G)$ ermöglicht eine Partition der Kantenmenge $G.E$ in **Tree-, Back-, Forward- und Crosskanten**, $G.E = T \cup B \cup F \cup C$.

- $(v, w) \in T$, falls $v = w.\pi$,
- $(v, w) \in B$, falls $I(v) \subset I(w)$,
- $(v, w) \in F$, falls $v \neq w.\pi$ und $v.d < w.d$, also $I(w) \subset I(v)$,
- $(v, w) \in C$, falls $v.d > w.f > w.d$, also $I(v) \cap I(w) = \emptyset$.

Theorem 49

Es gilt genau dann $I(w) \subset I(v)$, wenn ein Pfad von T-Kanten von v zu w existiert (Klammertheorem).

Beweis: Für Knoten $u, t \in G.V$ gilt $u = t.\pi$ genau dann, wenn es ein Zeitintervall gibt, in dem u direkt unter t in S liegt. □

$DFS(G)$ mit Kantenklassifizierung

```
 1 For each u ∈ G.V
 2     do u.color ← WHITE, u.π ← NIL
 3 time ← 0; S ← ∅
 4 T ← ∅
 5 B ← ∅
 6 F ← ∅
 7 C ← ∅
 8 For each u ∈ G.V
 9     do if u.color = WHITE
10         then DFSVisit(G, u)
```

$DFSVisit(G, u)$ mit Kantenklassifizierung

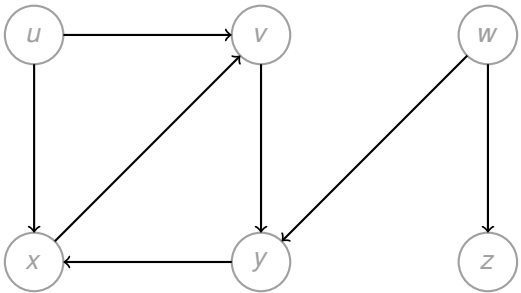
```
 1  time ← time + 1; u.d ← time
 3  u.color ← GRAY; push(S, u)
 4  For each v ∈ G.Adj[u]
 5     do if v.color = WHITE
 6           then v.π ← u; T ← T ∪ {(u, v)}
 7                DFSVisit(G, v)
 8           else EdgeClassify(u, v)
 9  time ← time + 1
10  u.f ← time
11  u.color ← BLACK; pop(S)
```

$EdgeClassify(u, v)$

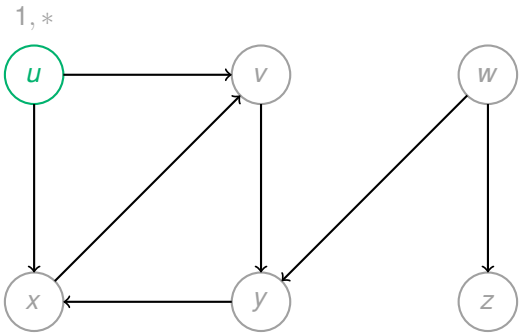
```
1  If v.color = GRAY then B ← B ∪ {(u, v)}
2                    else if u.d < v.d then F ← F ∪ {(u, v)}
3                                      else C ← C ∪ {(u, v)}
```

DFS Beispiel

T, B, F, C



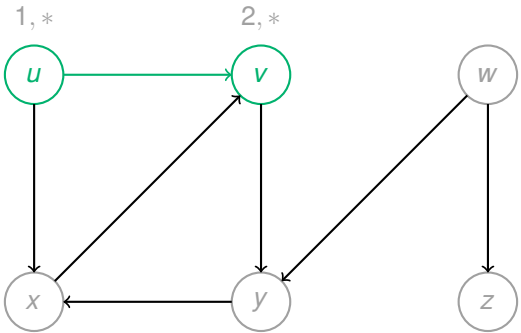
time = 1



T, B, F, C

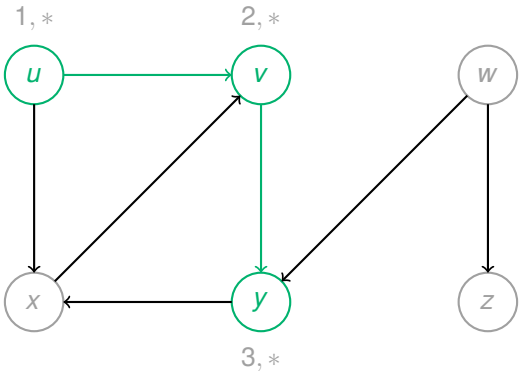
$(u, v) \in T, time = 2$

T, B, F, C



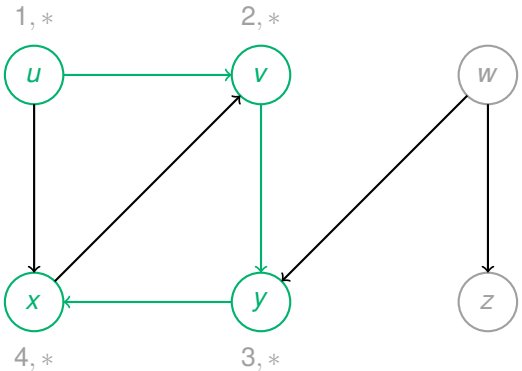
$(v, y) \in T, time = 3$

T, B, F, C



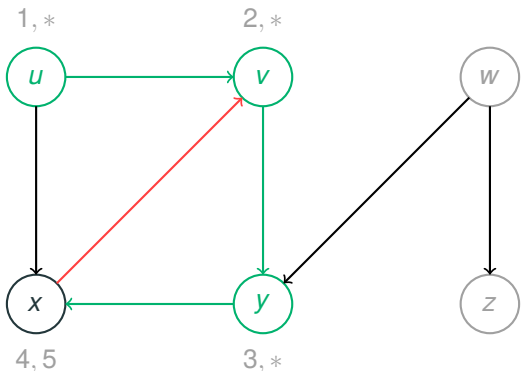
$(y, x) \in T, time = 4$

T, B, F, C



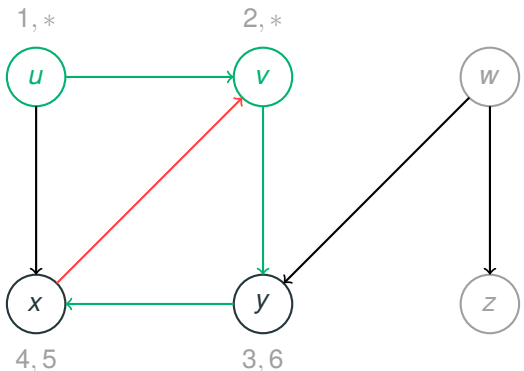
$(x, v) \in B, time = 5$

T, B, F, C

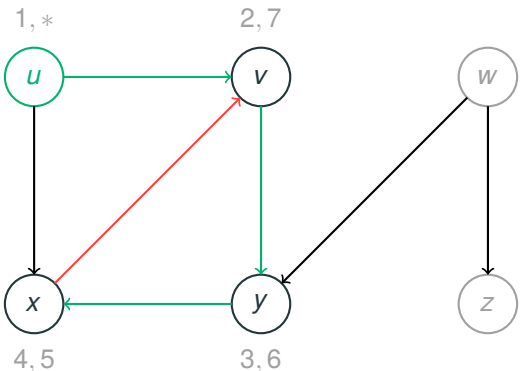


time = 6

T, B, F, C



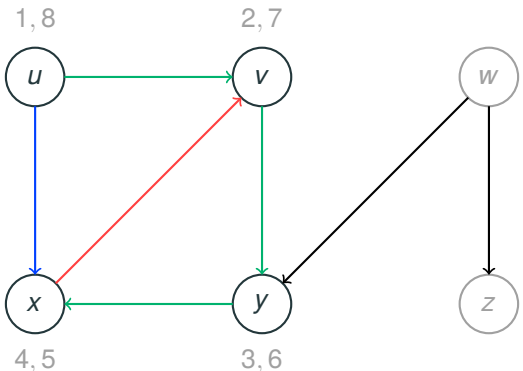
time = 7



T, B, F, C

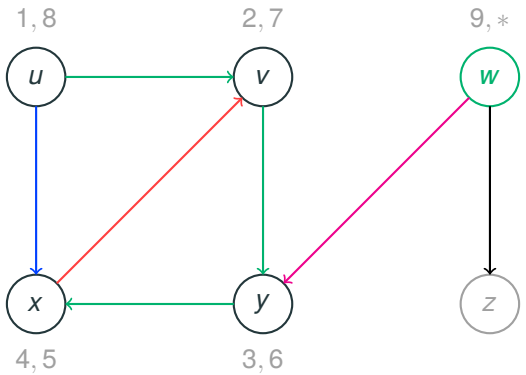
$(u, x) \in F, time = 8$

T, B, F, C



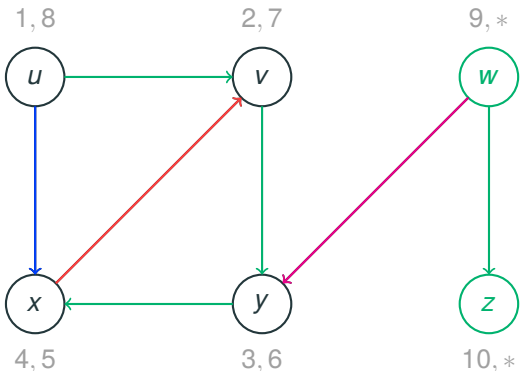
$(w, y) \in C, time = 9$

T, B, F, C

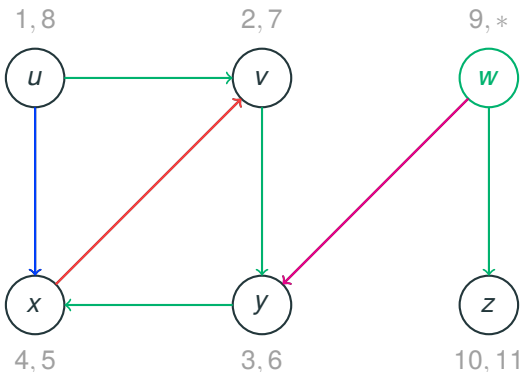


$(w, z) \in T, time = 10$

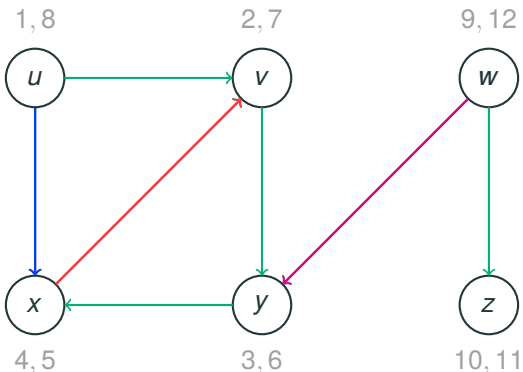
T, B, F, C



T, B, F, C



time = 12



T, B, F, C

Erkennung der Kreisfreiheit mittels $DFS(G)$

Theorem 50

G ist genau dann azyklisch, wenn $DFS(G)$ keine B-Kanten erzeugt.

Beweis: ($\Longrightarrow$) Jede B Kante (v, u) erzeugt einen Kreis, da aus $I(v) \subset I(u)$ folgt, dass ein Weg (aus T-Kanten) von u nach v existiert.

($\Longleftarrow$) G enthalte keine B-Kanten.

Für alle T-, F-, und C-Kanten (v, w) gilt $v.f > w.f$.

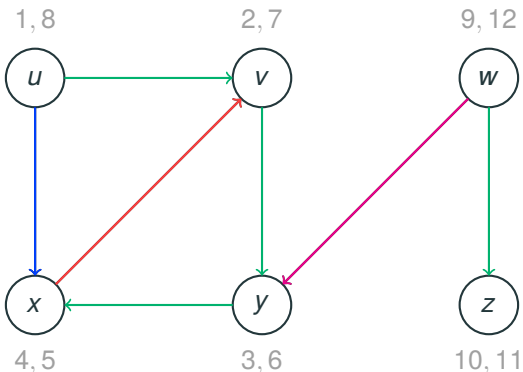
Das heißt, ordnet man V absteigend bezüglich $v.f$, so erhält man eine **topologische Knotensortierung** von G, welche es nur für azyklische Graphen geben kann.

TopologicalSort(G) (*G azyklisch*)

1 $DFS(G)$
2 **Sort** $G.V$ **in decreasing order with respect to** $v.f$**.**

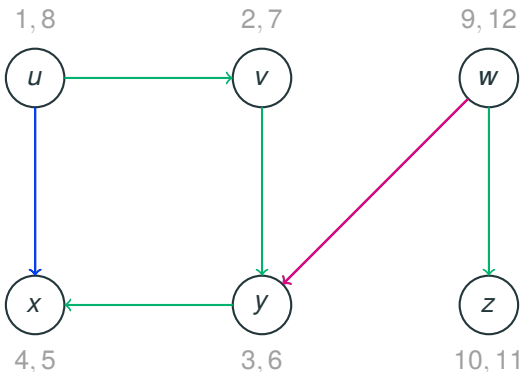
Zyklisches G mit B-Kante

T, B, F, C



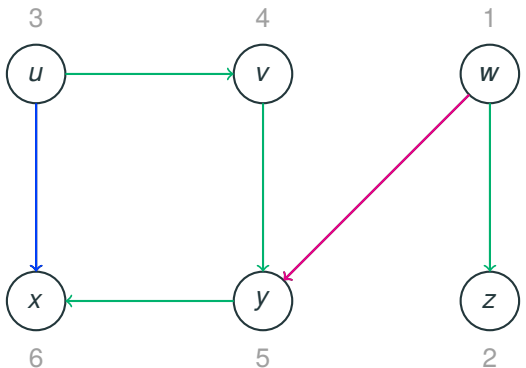
Azyklisches G

T, B, F, C



Topologische Knotensortierung

T, B, F, C



Berechnung der starken Zusammenhangskomponenten

Definition 51

- Knoten $v, w \in G.V$ heißen **stark zusammenhängend**, falls sowohl v von w aus, als auch w von v aus erreichbar ist, d.h. v, w liegen auf einem gerichteten Kreis in G.
- Starker Zusammenhang ist eine **Äquivalenzrelation auf** V, die Restklassen heißen **starke Zusammenhangskomponenten von** G.
- $SCC_G(v)$ bezeichne die starke Zusammenhangskomponente von G, die v enthält.

SimpleSCC(G, v)

1. **Berechne mittels** *DFSVisit*(G, v) **die Menge** $S_1(v)$ **der Knoten, die von** v **aus erreichbar sind.**
2. **Berechne mittels** *DFSVisit*(G^{-1}, v) **die Menge** $S_2(v)$ **der Knoten, von denen aus** v **erreichbar ist.**
3. **Markiere alle Knoten in** $S_1(v) \cap S_2(v)$ **und gebe** $S_1(v) \cap S_2(v)$ **aus**.

Ein besserer SCC-Algorithmus

- **Erinnerung:** G^{-1} entsteht aus G durch Umkehrung der Kantenrichtungen.
- *SimpleSCC* benötigt Zeit $O(|V|(|V|+|E|))$, um alle SCCs von G zu berechnen (wende *SimpleSCC*(v) solange auf unmarkierte Knoten an bis alle Knoten markiert sind).

Beobachtung: Es gibt einen effizienteren SCC-Algorithmus:

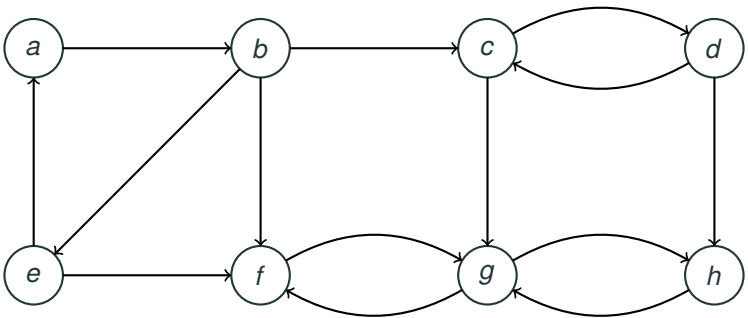
SCC(G)

1 *DFS*(G)
2 *DFS*(G^{-1}), **where** $G.V$ **is in decreasing order w.r.t.** $v.f$
3 **Output the trees of the resulting DFS-forest**

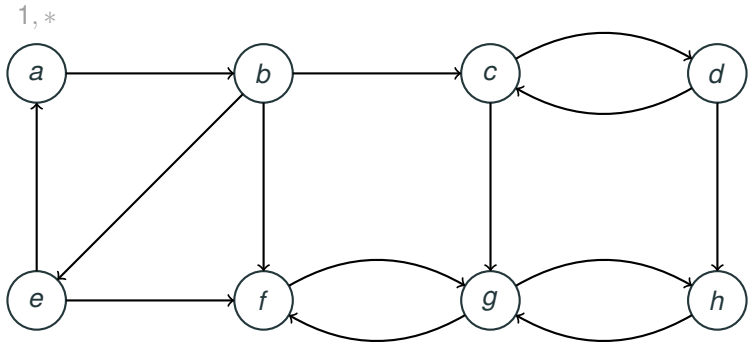
Laufzeit $O(|V|+|E|)$

Korrektheit?

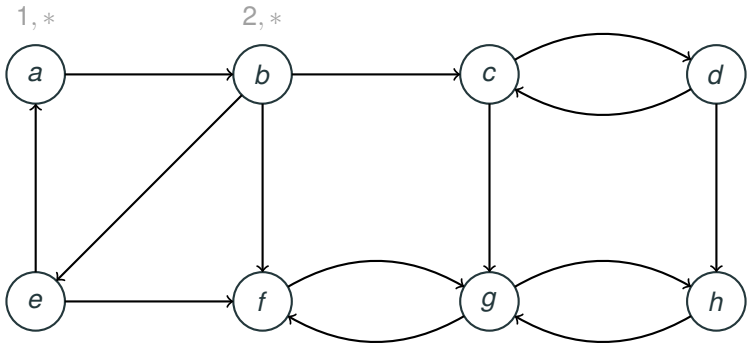
Beispiel für $SCC(G)$



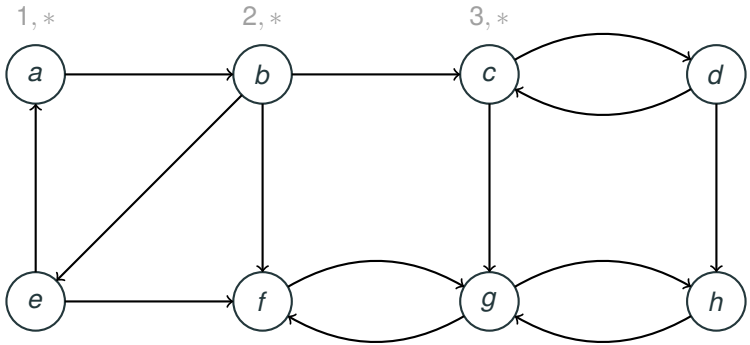
DFS(*G*), **1**



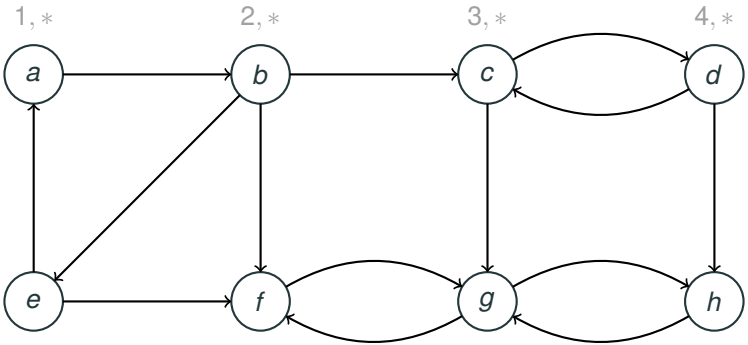
DFS(*G*), **2**



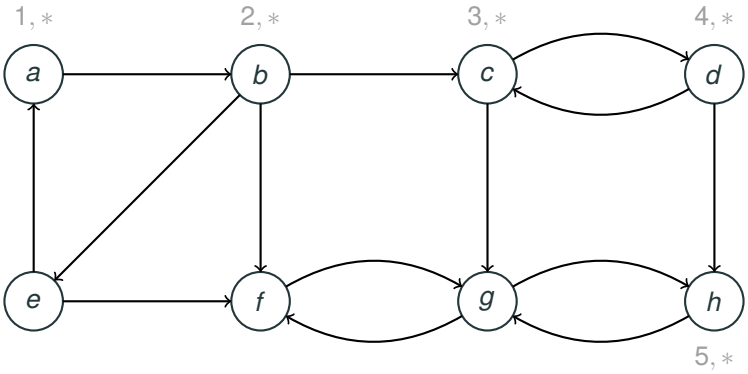
DFS(*G*), **3**



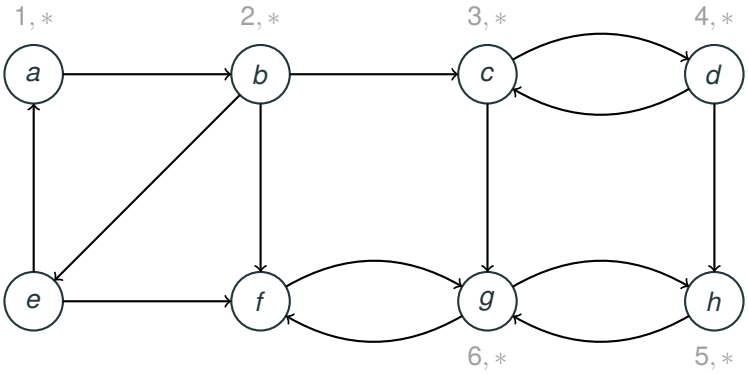
DFS(*G*), **4**



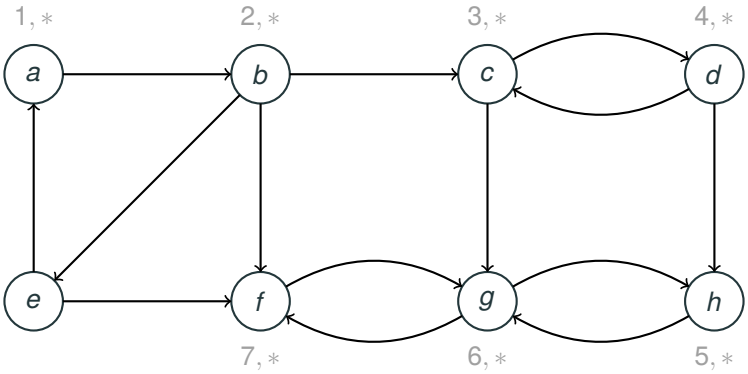
DFS(*G*), **5**



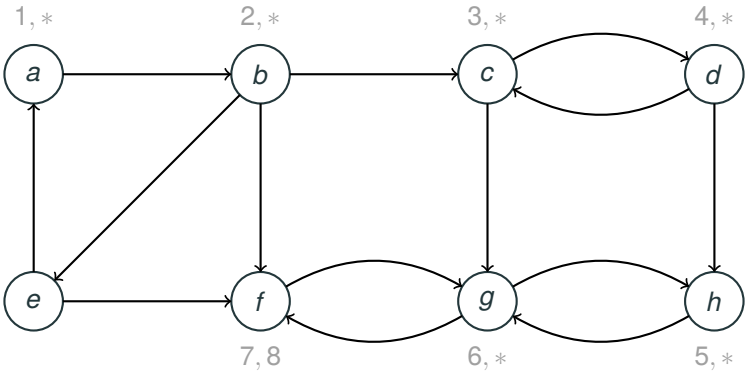
DFS(*G*), **6**



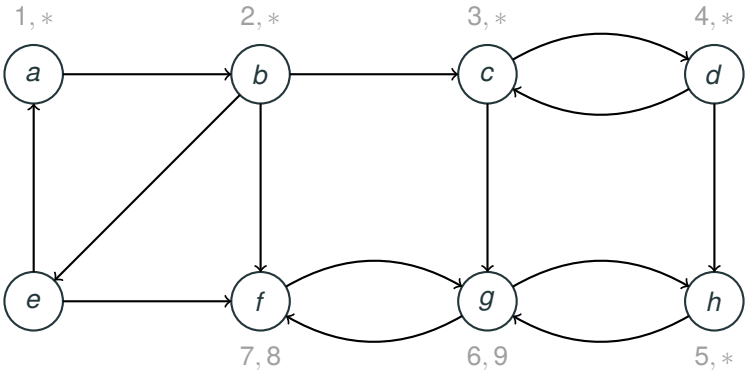
DFS(*G*), **7**



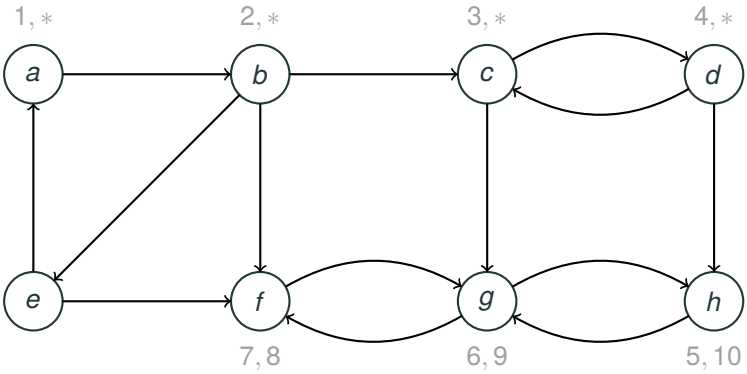
DFS(*G*), **8**



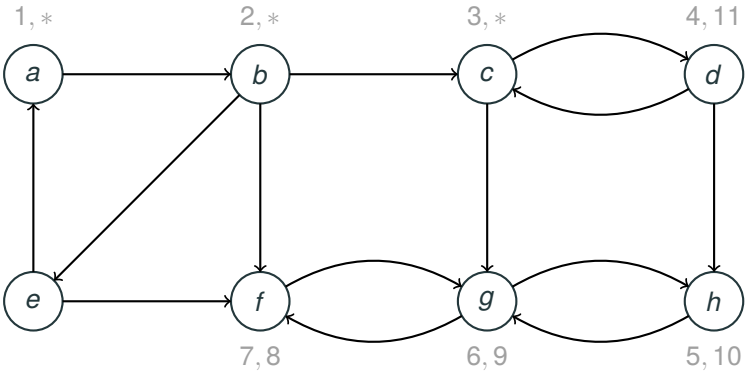
DFS(*G*), **9**



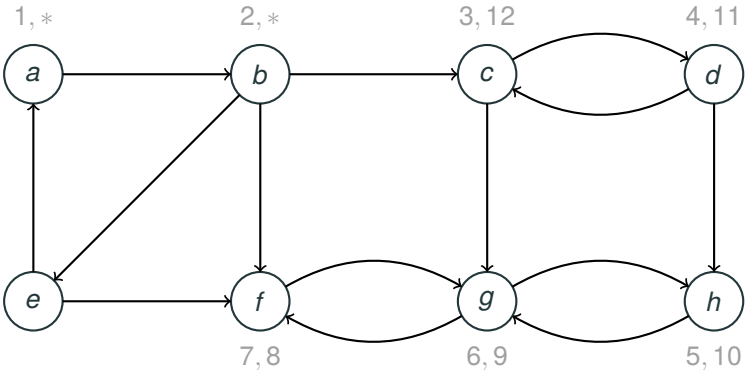
DFS(*G*), **10**



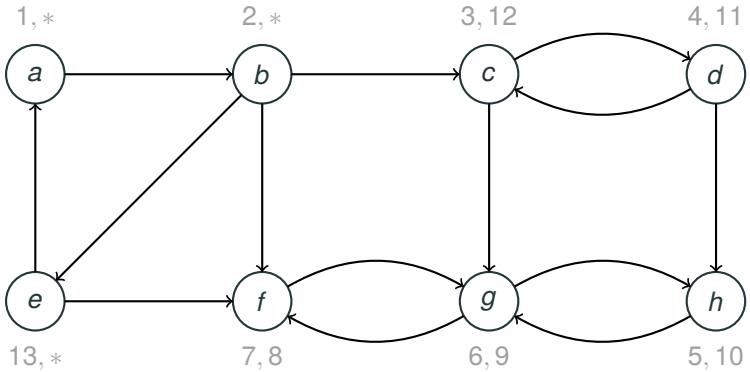
DFS(*G*), **11**



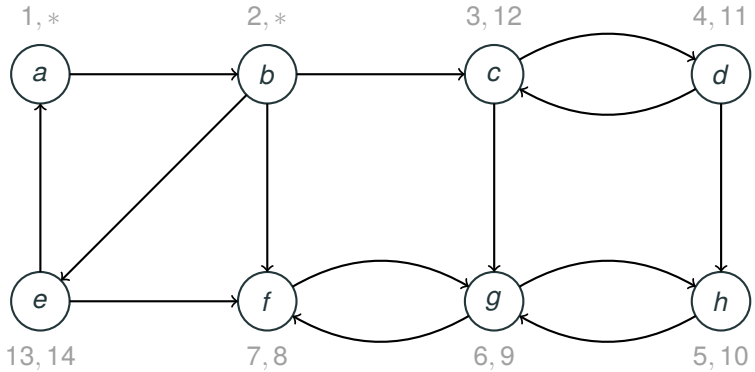
DFS(*G*), **12**



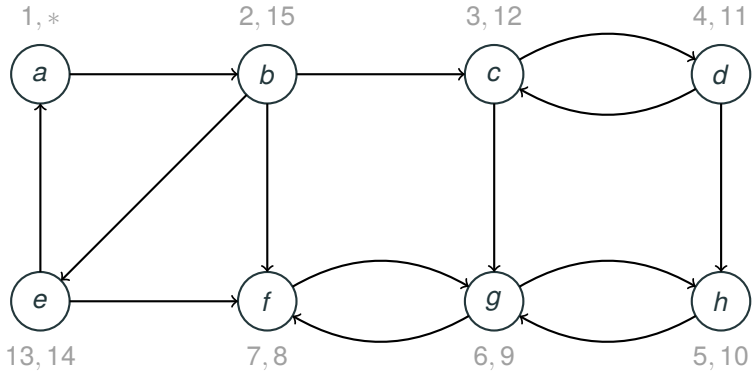
DFS(*G*), 13



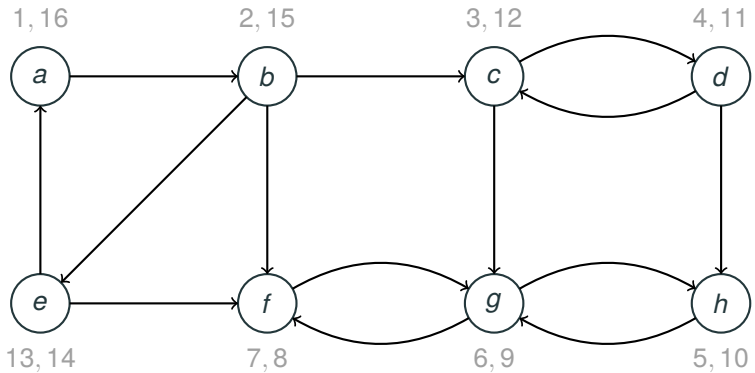
DFS(*G*), **14**



DFS(*G*), **15**

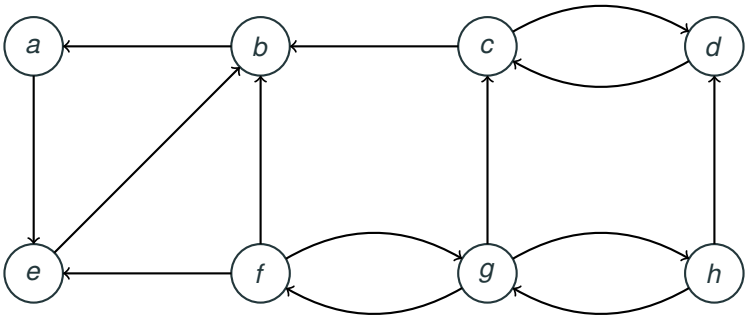


DFS(*G*), **16**



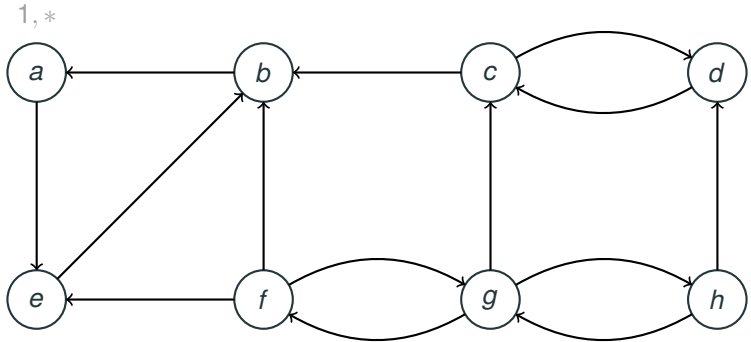
a,b,e,c,d,h,g,f

G^{-1} mit neuer Knotenordnung



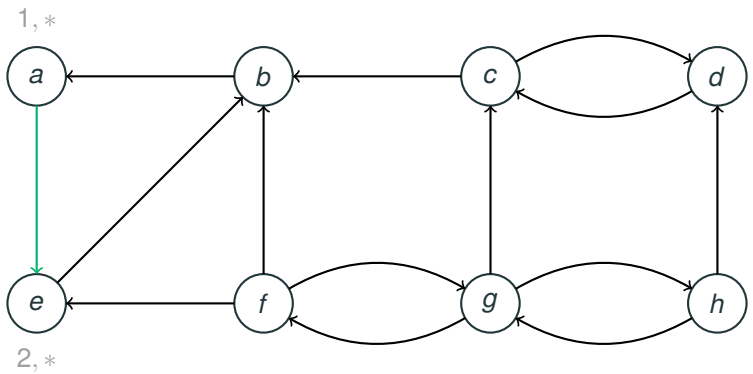
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, a)$ mit neuer Knotenordnung, 1



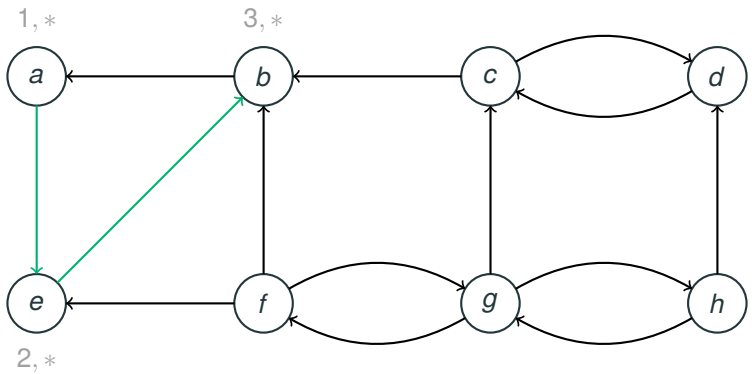
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, a)$ mit neuer Knotenordnung, 2



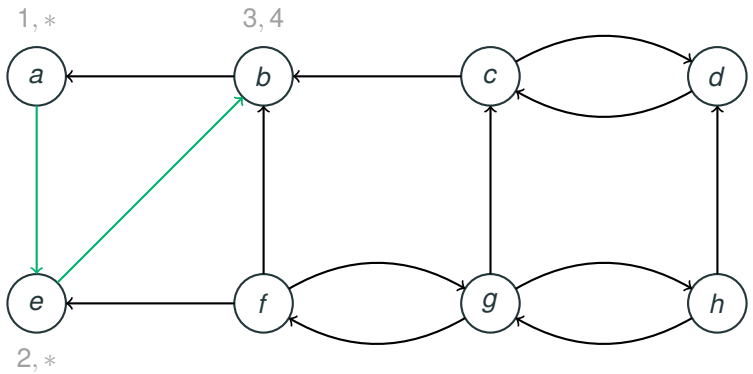
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, a)$ mit neuer Knotenordnung, 3



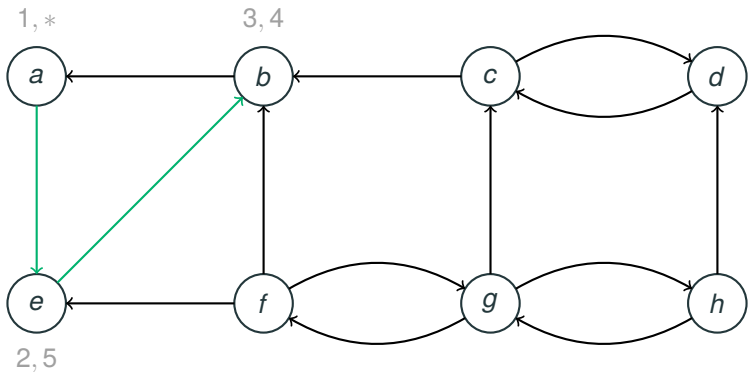
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, a)$ mit neuer Knotenordnung, 4



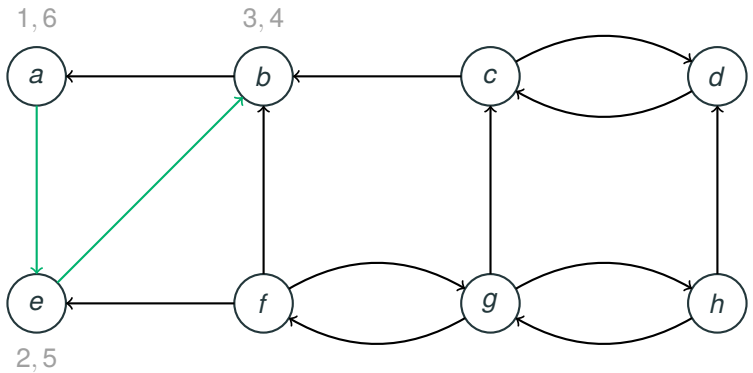
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, a)$ mit neuer Knotenordnung, 5



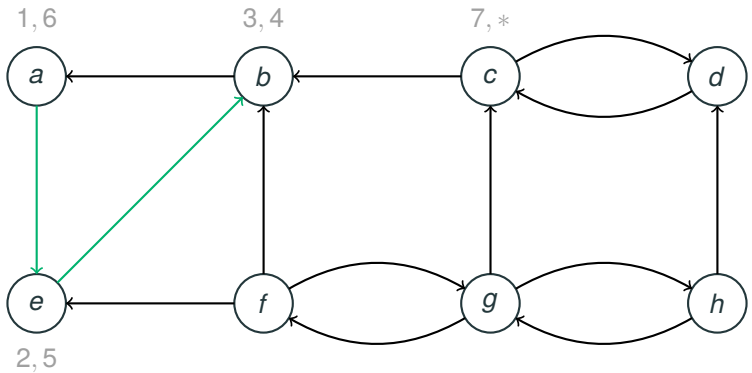
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, a)$ mit neuer Knotenordnung, 6



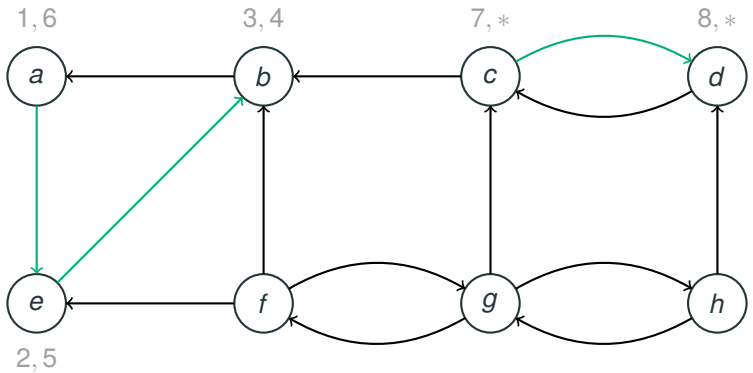
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, c)$ mit neuer Knotenordnung, 7



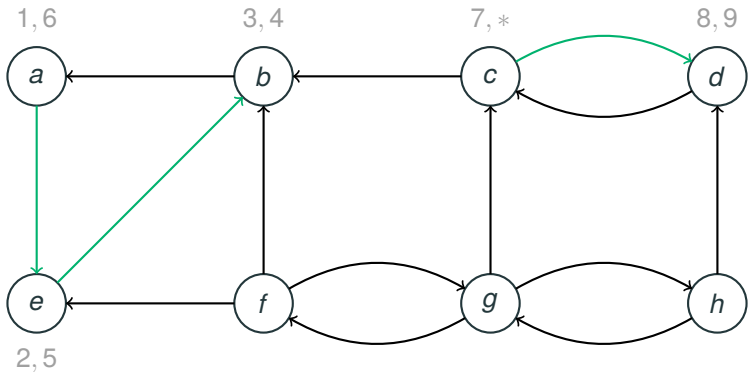
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, c)$ mit neuer Knotenordnung, 8



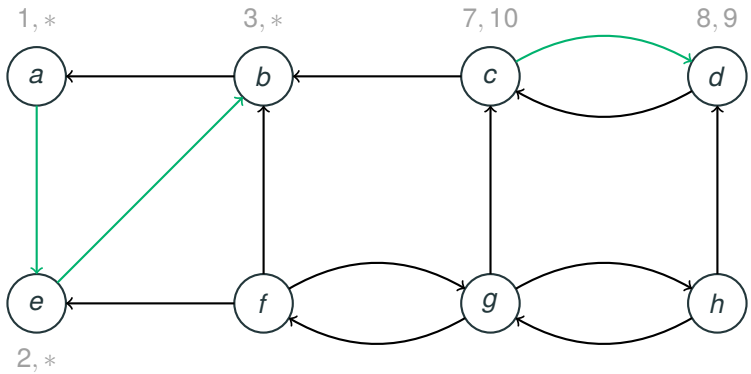
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, c)$ mit neuer Knotenordnung, 9



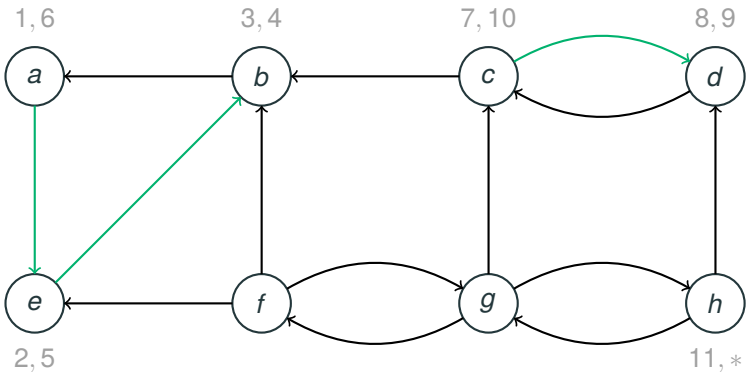
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, c)$ mit neuer Knotenordnung, 10



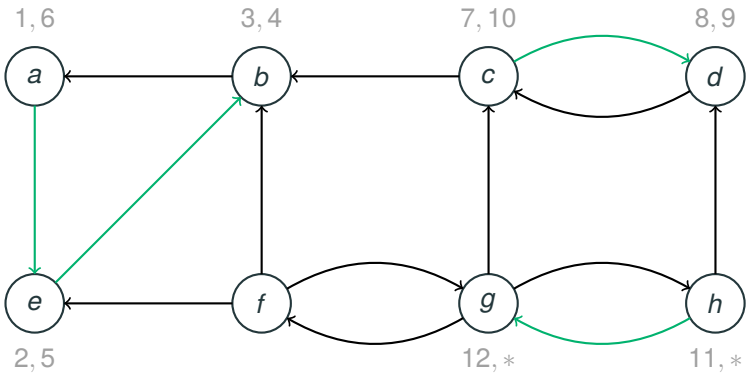
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, h)$ mit neuer Knotenordnung, 11



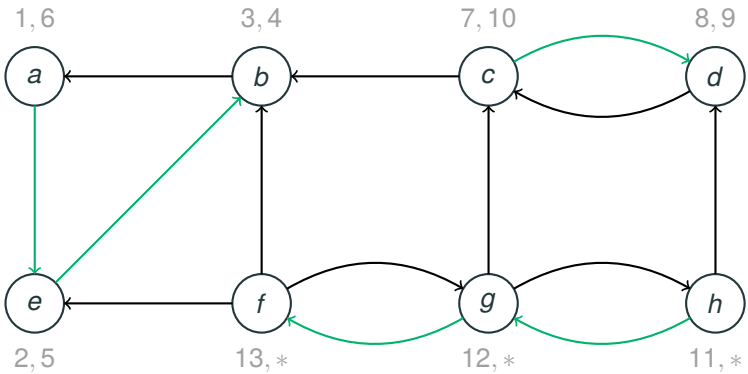
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, h)$ mit neuer Knotenordnung, 12



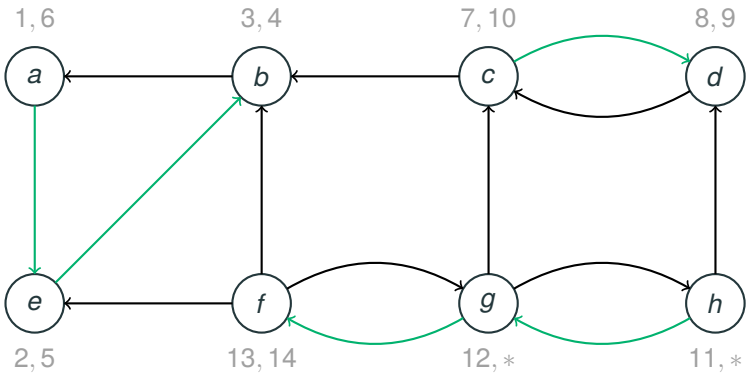
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, h)$ mit neuer Knotenordnung, 13



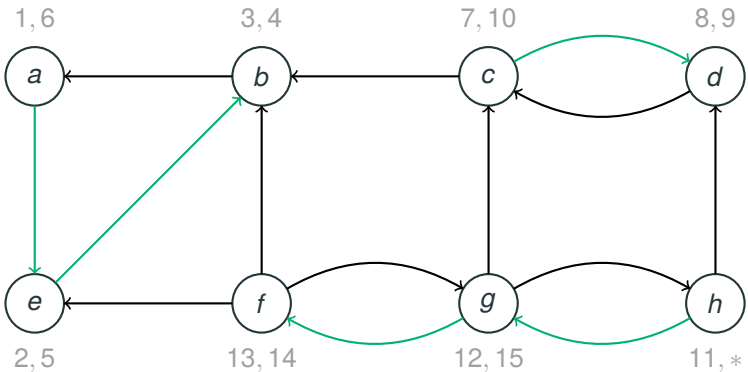
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, h)$ mit neuer Knotenordnung, 14



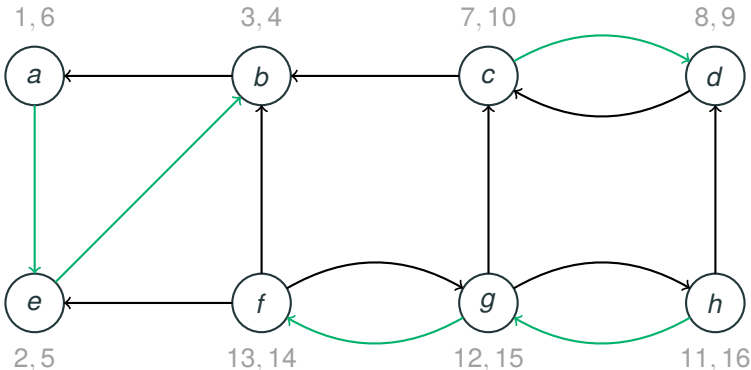
a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, h)$ mit neuer Knotenordnung, 15



a,b,e,c,d,h,g,f

$DFSvisit(G^{-1}, h)$ mit neuer Knotenordnung, 16



SCC: $\{a, b, e\}$, $\{c, d\}$, $\{f, g, h\}$

Korrektheit von $SCC(G)$, I

1 $DFS(G)$
2 $DFS(G^{-1})$, **where $G.V$ is in decreasing order w.r.t. $v.f$**
3 **Output the trees of the resulting DFS-forest**

Theorem 52

Die in Zeile 3 von $SCC(G)$ ausgegebenen DFS-Bäume sind genau die starken Zusammenhangskomponenten von G.

Beweis: Wir nehmen an, dass $DFS(G^{-1})$ bereits $r \geq 0$ DFS-Bäume ausgegeben hat, die allesamt starke Zusammenhangskomponenten von G sind.

Es sei V' die verbleibende Knotenmenge, und $u \in V'$ definiert durch $u.f = \max\{w.f, w \in V'\}$.

D.h., $DFS(G^{-1})$ startet als nächstes $DFSVisit(G^{-1}, u)$.

Korrektheit von $SCC(G)$, Fortsetzung I

Man beachte, dass $SCC_G(u) \subseteq V'$, denn ...

Wäre ein Knoten $w \in SCC_G(u)$ bereits vor dem Start von $DFSVisit(G^{-1}, u)$ entdeckt worden, so würde das auch für u gelten, da u von w aus in G^{-1} erreichbar ist, Widerspruch.

$DFSVisit(G^{-1}, u)$ entdeckt alle Knoten in $SCC_G(u)$, da $SCC_G(u) \subseteq V'$ und u von allen Knoten $w \in SCC_G(u)$ aus in G erreichbar ist.

Angenommen, $DFSVisit(G^{-1}, u)$ entdeckt Knoten $v \notin SCC_G(u)$.

Es gilt $v.f < u.f$, da $v \in V'$.

Fall 1: $u.d < v.d$, also $u.d < v.d < v.f < u.f$, d.h. $I(v) \subset I(u)$.

Dann existiert ein Weg aus T-Kanten von u nach v (Klammertheorem), aber auch ein Weg von v nach u (da $DFSVisit(G^{-1}, u)$ den Knoten v entdeckt), d.h. $v \in SCC_G(u)$, Widerspruch.

Korrektheit von $SCC(G)$, Fortsetzung II

Fall 2: $v.d < u.d$.

Dann ist u beim Start von $DFSVisit(G, v)$ weiß.

Der Knoten u wird während $DFSVisit(G, v)$ entdeckt, da v von u aus in G^{-1} und damit u von v aus in G erreichbar ist.

Also gilt $v.d < u.d < u.f < v.f$ (Klammertheorem), Widerspruch zu $v.f < u.f$! $\square$

Union-Find Datenstrukturen zur dynamischen Verwaltung von Partitionen

Union-Find Datenstrukturen zur dynamischen Verwaltung von Partitionen

Grundlagen von Union-Find Datenstrukturen

Partitionen

Definition 53

Eine Familie $\Pi = \{M_1, \cdots, M_r\}$ von nichtleeren Teilmengen einer gegebenen Menge M heißt **Partition von** M, falls

- $M_i \cap M_j = \emptyset$ für alle $i \neq j$, $1 \leq i,j \leq r$, und
- $\bigcup_{i=1}^{r} M_i = M$.

Fakt: Jede **Äquivalenzrelation** auf einer nichtleeren Mange M definiert eine Partition von M in die Menge der **Äquivalenzklassen**.

Beispiel: Die **Zusammenhangskomponenten** eines ungerichteten bzw. die **starken Zusammenhangskomponenten** eines gerichteten Graphen $G = (V, E)$ entsprechen der Partition von V in die Äquivalenzklassen der Zusammenhangs- bzw. der starken Zusammenhangsrelation.

Dynamische Verwaltung von Partitionen

- Verwalten die Teilmengen einer Partition Π von M durch die Festlegung eines Mengenelementes (bzw. Bezeichners) S.
- S kann ein zusätzliches Datenobjekt oder aber auch ein ausgewähltes Element (Repräsentant) aus der Teilmenge sein.
- Wir verwalten Partitionen dynamisch durch sogenannte **Union-Find** Datenstrukturen, d.h. bezüglich der Operationen
  - $\mathit{Find}(x)$ (Ausgabe: das eindeutig bestimmte $S \in \Pi$ mit $x \in S$)
  - $\mathit{Union}(x, y)$) (Vereinigt die Mengen $\mathit{Find}(x)$ und $\mathit{Find}(y)$ und gibt $\mathit{Find}(x) \cup \mathit{Find}(y)$ eine Bezeichnung $S \in \{\mathit{Find}(x), \mathit{Find}(y)\}$.
- **Initialoperation** $\mathit{MakeSet}(S, x)$ erzeugt $\{x\}$ und bezeichnet diese Menge mit S. Wir schreiben $\mathit{MakeSet}(x)$ falls $S = x$.

Beispiel Zusammenhangskomponenten von ungerichteten Graphen

ConnectedComponents(G)

1 **For all** $v \in V$ **do** *Makeset*(v)
2 **For all** $e = (v, w) \in E$
3 **do if** *Find*$(v) \neq$ *Find*(w)
4 **then** *Union*(v, w)

Lemma 54

ConnectedComponents(G) *berechnet die Zusammenhangskomponenten von* G*.*

Beweis: Wir betrachten die Situation nach *ConnectedComponents*(G), $G = (V, E)$, und zeigen das Folgende:

Für alle $v \neq w \in V$ gilt, dass *Find*$(v) =$ *Find*(w) genau dann, wenn ein Weg p zwischen v und w in G existiert.

Wir fixieren beliebige $v \neq w \in V$ und nehmen an, dass *Find*$(v) =$ *Find*(w).

Beweis von Lemma 54

Sei S die Menge, die durch die Operation $Union(v', w')$, die $Find(v) = Find(w)$ verursacht hat, entsteht.

Dann gilt $(v', w') \in E$ und $S = S_1 \cup S_2$, und vor Ausführung von $Union(v', w')$ gilt $S_1 = Find(v) = Find(v')$ und $S_2 = Find(w) = Find(w')$.

Wir zeigen, dass v und w zusammenhängend sind per Induktion über $|S|$.

Ist $|S| = 2$ so gilt $v = v'$ und $w = w'$ und damit $(v, w) \in E$, also v und w zusammenhängend.

Es sei $|S| > 2$. Es gilt $|S_1| < |S|$ und $|S_2| < |S|$ und damit laut Induktionsvoraussetzung, dass sowohl v und v' als auch w und w' zusammenhängend sind.

Aus $(v', w') \in E$ folgt, dass auch v und w zusammenhängend sind.

Fortsetzung

Wir nehmen andersherum an, dass ein zusammenhängendes Knotenpaar $v \neq w$ existiert mit $\mathit{Find}(v) \neq \mathit{Find}(w)$.

Damit existiert entlang des Weges von v nach w eine Kante $e = (z, z')$ mit $\mathit{Find}(z) \neq \mathit{Find}(z')$, Widerspruch. $\square$

Union-Find Datenstrukturen zur dynamischen Verwaltung von Partitionen

Linked Lists

Linked Lists, Definition und Operationen

- Wir verwalten Elemente $x \in M$ als Knotenelemente mit Zeiger $x.next$ und $x.repr$.
- Wir adressieren Teilmengen $S \subseteq M$ durch ein Mengenelement S mit den Komponenten $S.head$ und $S.tail$.
- $S.head$ und $S.tail$ sind Zeiger auf den Anfang bzw. das Ende der mittels $x.next$ einfach verketteten Liste der Elemente in S.
- Für alle $x \in S$ gelte $x.repr = S$.

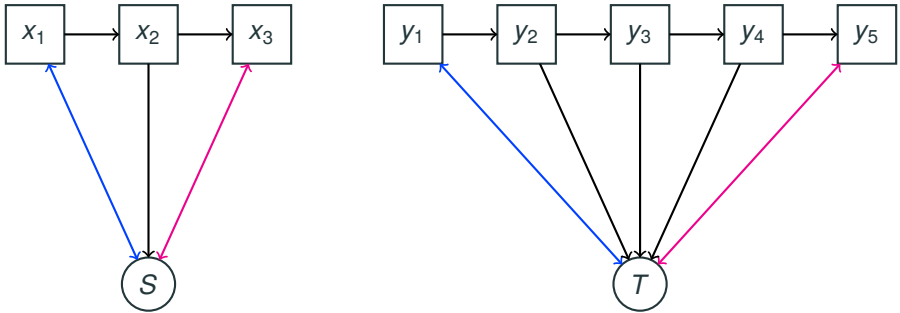
MakeSet(x, S)

```
1 x.repr ← S
2 x.next ← NIL
3 S.head ← S.tail ← x
```

Find(x)

```
1 return x.repr
```

Beispiel Linked List



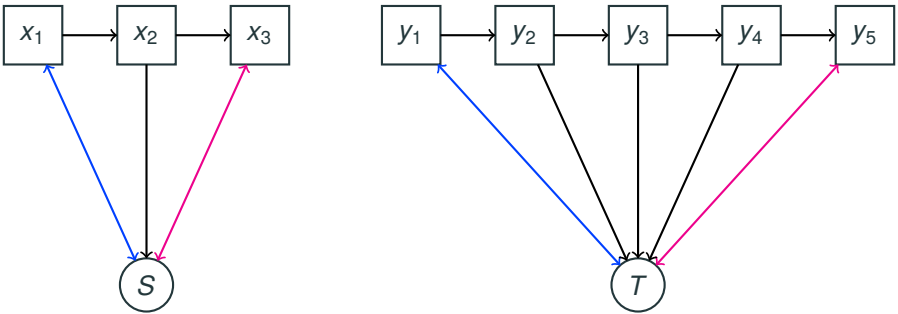
$\leftrightarrow$ Kanten sind *.tail* + *.repr* und *.head* + *.repr*

Linked Lists *SimpleUnion*

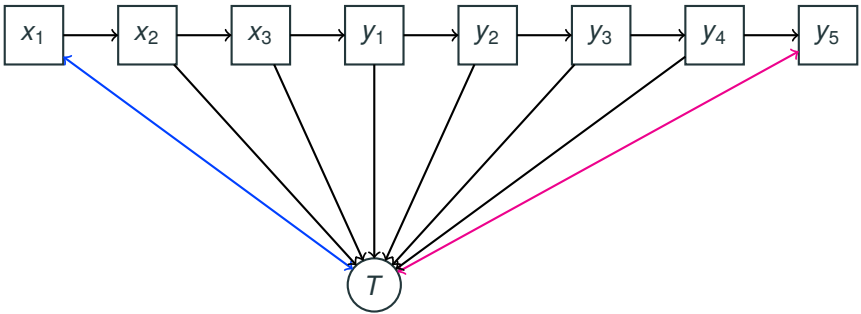
SimpleUnion(x, y)

```
1  help1 ← x.repr.head
2  help2 ← x.repr.tail
3  help1.next ← y.repr.tail (*Hänge Find(x) an Find(y)*)
4  y.repr.tail ← help2
5  while help2 ≠ help1.next
6      do help2.repr ← y.repr (*Überschreiben*)
7          help2 ← help2.next
```

Nochmal Beispiel Linked List, vor *SimpleUnion*(x_2, y_3)



nach *SimpleUnion*(x_2, y_3)



Linked Lists, Kosten

- Die Kosten von *Find* und *Makeset* sind $O(1)$
- Die Kosten von *SimpleUnion*(x, y) sind $\Theta(|Find(x)|)$.
- **Problem:** Eine Folge von $n-1$ Unions nach n *Makesets* kann Kosten $\Theta(n^2)$ haben.
- **Beispiel:** *Union*$(1,2)$, *Union*$(2,3), \cdots$, *Union*$(n-1,n)$ nach *Makeset*$(1,1), \cdots$, *Makeset*(n,n).
- Für alle $i \leq n-1$ gilt direkt vor *Union*$(i, i+1)$, dass $|Find(i)| = i$ und $|Find(i+1)| = 1$, Gesamtkosten

$$\Theta\left(\sum_{i=1}^{n-1} i\right) = \Theta(n^2).$$

- **Ausweg:** Vermerken für Teilmengen $S \subseteq M$ unter $S.size$ die Kardinalität und überschreiben immer die kleinere Menge:

Modified *SimpleUnion*

ModifiedSimpleUnion(x, y)

0 $y.repr.size \leftarrow x.repr.size + y.repr.size$
1 $help1 \leftarrow x.repr.head$
2 $help2 \leftarrow x.repr.tail$
3 $help1.next \leftarrow y.repr.tail$ (*Hänge $Find(x)$ an $Find(y)$*)
4 $y.repr.tail \leftarrow help2$
5 **while** $help2 \neq help1.next$
6 **do** $help2.repr \leftarrow y.repr$ (*Überschreiben*)
7 $help2 \leftarrow help2.next$

Verbesserte Linked Lists

MakeSet(x, S)

1 $x.repr \leftarrow S$
2 $x.next \leftarrow NIL$
3 $S.head \leftarrow S.tail \leftarrow x$
4 $S.size \leftarrow 1$

Union(x, y)

1 **If** $Find(y).size \leq Find(x).size$
2 **then** $ModifiedSimpleUnion(y, x)$
3 **else** $ModifiedSimpleUnion(x, y)$

Verbesserte Linked Lists, Kostenanalyse

Theorem 55

Eine Folge von m Operationen mit n MakeSets kostet höchstens $O(m + n \cdot \log(n))$.

Beweis: Die Gesamtkosten aller *Makesets* und *Finds* ist $O(m)$.

Die Gesamtkosten aller *Unions* sind proportional zur Gesamtzahl aller Überschreibungen von Repräsentantenzeigern.

Der Repräsentantenzeiger jedes Knotenelements x wird durch *Union* höchstens $\lfloor log_2(n) \rfloor$ mal überschrieben.

Das liegt daran, dass bei jeder Überschreibung von $x.repr$ das Element x immer in der kleineren der zu vereinigenden Mengen liegt.

Damit verdoppelt sich die Größe der Menge $Find(x)$ nach jeder Überschreibung, was nur $\lfloor log_2(n) \rfloor$ mal möglich ist, da stets $|Find(x)| \leq n$ □

Union-Find Datenstrukturen zur dynamischen Verwaltung von Partitionen

Disjoint Set Forests

Disjoint Set Forests, Definition und Operationen

- Mengen werden abgelegt als **wurzelgerichtete Bäume** aus **Knotenelementen** x mit **Elternknoten-Zeiger** $x.p$ und einer **Rang-Komponente** $x.rank$, die die Höhe von x im Baum bestimmt.
- Der **Repräsentant einer Menge** ist die Wurzel des entsprechenden Baums.

MakeSet(x)

```
1 x.p ← x
2 x.rank ← 0
```

Find(x)

```
1 while x ≠ x.p
2       do x ← x.p
3 output x
```

Disjoint Set Forests, *Union* by Rank

Informal: Die Union Operation hängt die Wurzel der *flacheren* Menge and die Wurzel der *höheren* Menge.

$\underline{Union(x, y)}$

1 $Link(Find(x), Find(y))$

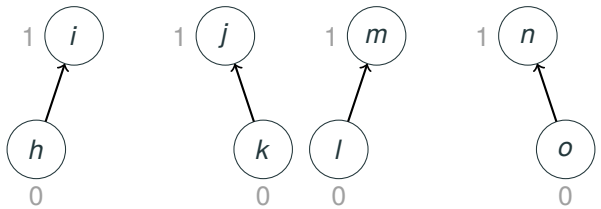
$\underline{Link(x, y)}$

1 **If** $x.rank \leq y.rank$
2 **then** $x.p \leftarrow y$
3 **else** $y.p \leftarrow x$
4 **If** $x.rank = y.rank$
5 **then** $y.rank \leftarrow y.rank + 1$

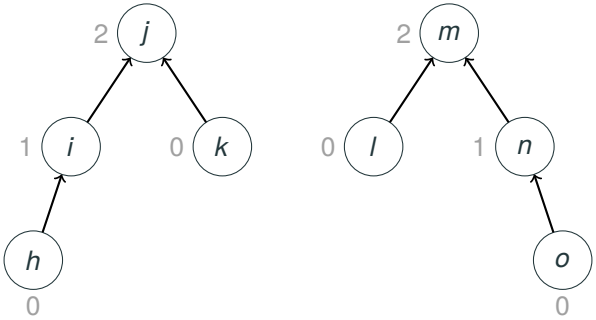
Beobachtung: Die Kosten von $Find(x)$ sind $O(Find(x).rank - x.rank)$.

Beispiel Disjoint Set Forest, nach *Makeset*($h \cdots o$)

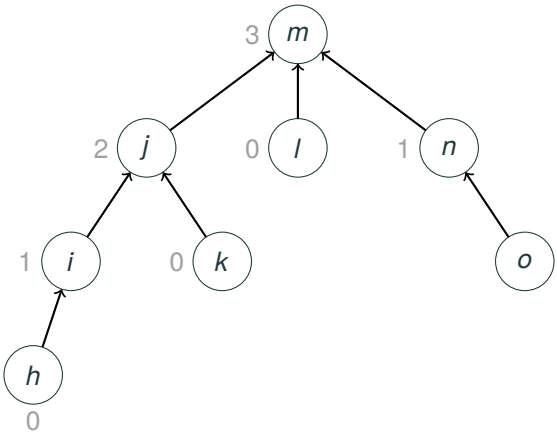
Nach $Union(h, i)$**,** $Union(k, j)$**,** $Union(l, m)$**,** $Union(o, n)$



Nach $Union(h, k)$, $Union(n, m)$



Nach *Union*(i, n)



Kostenanalyse Disjoint Set Forests

Theorem 56

Eine Folge von m Operationen, unter denen n MakeSets sind, kostet $O(m \cdot \log(n))$.

Beweis: Es sei S eine der erzeugten Mengen, $root(S)$ bezeichne die Wurzel von S.

Wir zeigen per Induktion, dass $|S| \geq 2^r$ mit $r = root(S).rank$.

Das impliziert $x.rank \leq \log_2(n)$ für alle x, und somit den Beweis des Satzes.

Falls $rank(S) = 0$ so ist $|S| = 1$.

Falls $r > 0$, so ergibt sich S als $S = X \cup Y \cup Z$.

Hierbei werde $root(S).rank$ auf r gesetzt durch die Operation $X \cup Y$, während Z alles umfasst, was danach an $root(S)$ gehangen wurde.

Somit gilt $rank(X) = rank(Y) = r - 1$, $root(S) = root(Y)$, und $rank(z) < r$ für alle $z \in Z$.

Also gilt $|Y| \geq 2^{r-1}$ und $|X| \geq 2^{r-1}$, also $|S| \geq 2^r$. $\square$

*Disjoint Set Forests mit Path Compression

Idee: Wir hängen während $Find(x)$ alle Knoten auf dem Weg von x zur Wurzel an die Wurzel $Find(x)$ dran.

$\underline{Find(x)}$

```
1 If x ≠ x.p
2    then x.p ← Find(x.p)
3    else output x
```

Theorem 57

Bei Verwendung von Path Compression kostet eine Folge von m Operationen, unter denen n MakeSets sind, $O(m \cdot \alpha(n))$. □

Ohne **Beweis** hier, $\alpha(n)$ ist eine extrem langsam wachsende Funktion, $\alpha(x) \leq 4$ für $x \leq 16^{512}$ (d.h., für alle praktisch auftretenden x).

(Siehe Cormen/Leiserson/Rivest/Stein Kapitel 21.4).

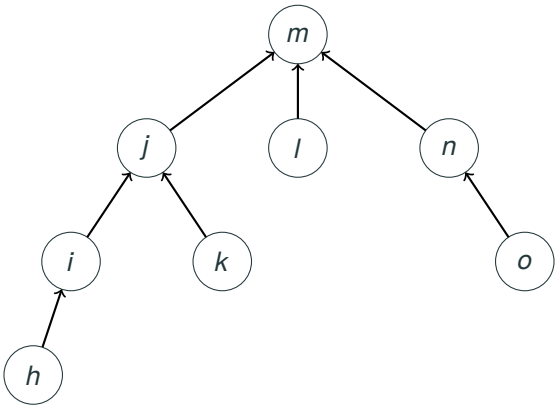
*Path Compression sequentiell

Find(x)

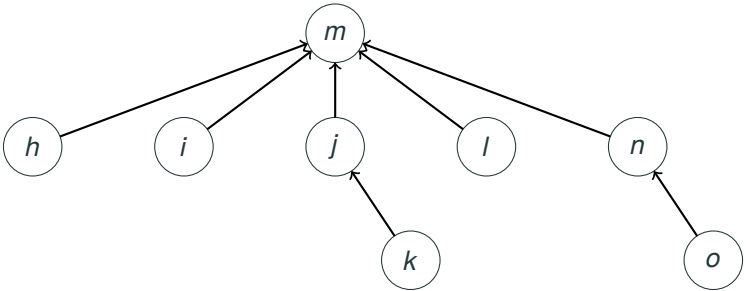
```
1 help1 ← x
2 while help1 ≠ help1.p
3       do help1 ← help1.p
4 while x ≠ help1
5       do help2 ← x.p
6          x.p ← help1
7          x ← help2
8 return help1
```

Bemerkung: Bei Verwendung von Path Compression sollte die Komponente $x.rank$ nicht verwendet werden, da Path Compression die Beziehung zwischen der Höhe von x im aktuellen Mengenbaum und $x.rang$ ggf. zerstört.

Beispiel Path Compression, vor *Find*(*h*)



Nach *Find*(*h*)



Berechnung Minimaler Spannbäume (MSTs) in gewichteten ungerichteten Graphen

Berechnung Minimaler Spannbäume (MSTs) in gewichteten ungerichteten Graphen

Ein generischer MST-Algorithmus

Minimale Spannbäume in gewichteten ungerichteten Graphen

Definition 58

Sei $G = (V, E, w)$ ein gewichteter ungerichteter zusammenhängender Graph mit einer Kantengewichtsfunktion $w : E \longrightarrow \mathbb{R}$.

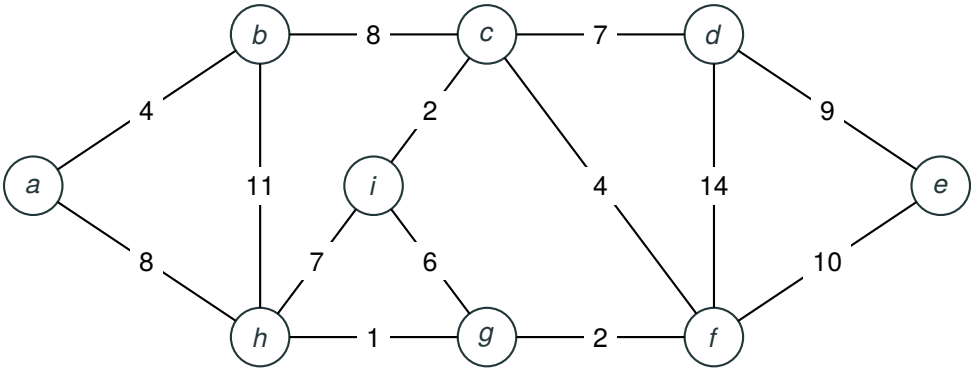
- Eine Menge von Kanten $T \subseteq E$ heißt **Spannbaum von** G, falls (V, T) ein Baum, d.h., ein **zusammenhängender und kreisfreier** Untergraph von G ist.
- Ein Spannbaum T heißt **Minimaler Spannbaum (MST)** von $G = (V, E, w)$, falls $w(T) \leq w(T')$ für alle Spannbäume T' von G, wobei $w(T) = \sum_{e \in T} w(e)$.

Erinnerung 1. Semester FGdI:

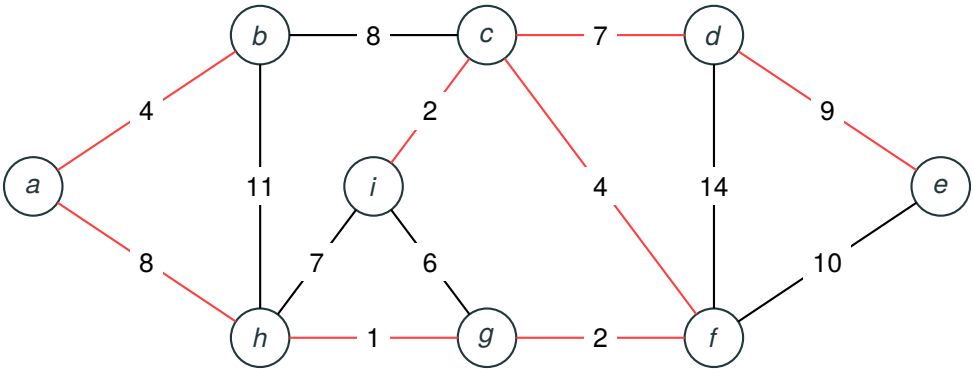
Lemma 59

Ein kreisfreier Graph $G = (V, E')$ mit weniger als $|V| - 1$ Kanten ist nicht zusammenhängend und ein zusammenhängender Graph $G = (V, E')$ mit mehr als $|V| - 1$ Kanten ist nicht kreisfrei. Bäume haben somit stets $|V| - 1$ Kanten. □

Beispielgraph



Beispielgraph mit MST A



$A = \{(a,b),(a,h),(c,f),(g,h),(f,g),(c,i),(c,d),(d,e)\}$

$w(A) = 37$

Ein generischer MST-Algorithmus

Definition 60

Sei $A \subseteq E$ eine Teilmenge eines MSTs für $G = (V, E, w)$.

Eine Kante $e \in E \setminus A$ heißt **sicher (safe)** für A, falls $A \cup \{e\}$ auch Teilmenge eines MSTs für $G = (V, E, w)$ ist.

GenericMST(G)

```
1 A ← ∅
2 while |A| < |V| − 1
3     do choose e ∈ E safe for A
4         A ← A ∪ {e}
5 return A
```

Der Algorithmus ist per Definition korrekt.

Wichtige Frage: Wie berechnet man sichere Kanten?

Charakterisierung sicherer Kanten

Definition 61

Sei $G = (V, E, w)$ gewichteter ungerichteter Graph und $A \subseteq E$.

- Eine Partition $\{S, V \setminus S\}$ von V heißt **Schnitt** von V.
- $(u, v) \in E$ heißt **Crosskante** für $\{S, V \setminus S\}$, falls $u \in S$ und $v \notin S$, oder $u \notin S$ und $v \in S$.
- A **respektiert** $\{S, V \setminus S\}$, falls A keine Crosskanten für $\{S, V \setminus S\}$ enthält.
- Eine Crosskante $e = (u, v) \in E$ heißt **leichte Crosskante** für $\{S, V \setminus S\}$, falls $w(e) \leq w(e')$ für alle Crosskanten e' für $\{S, V \setminus S\}$ aus E.

Theorem 62

Sei $A \subseteq E$ eine echte Teilmenge eines MSTs für $G = (V, E, w)$, die den Schnitt $\{S, V \setminus S\}$ respektiert, und e sei eine leichte Crosskante für $\{S, V \setminus S\}$. Dann ist e sicher für A.

Der Beweis von Theorem 62

Es sei T ein MST für G, der A aber nicht e enthält. Enthielte T neben A auch e, so wäre nichts zu zeigen.

Da $T \cup \{e\}$ zusammenhängend und $|T \cup \{e\}| = |V|$, enthält gemäß Lemma 59 die Kantenmenge $T \cup \{e\}$ einen Kreis K, der über e von S nach $V \setminus S$ führt und deshalb neben e eine weitere Crosskante $e' \in T$ enthalten muss, die von $V \setminus S$ zurück nach S führt.

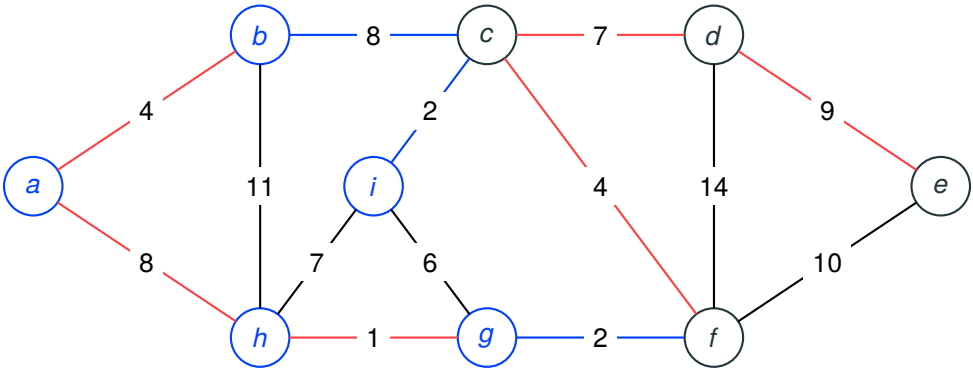
Wir zeigen, dass der Graph $G = (V, T')$ mit der Kantenmenge $T' = (T \cup \{e\}) \setminus \{e'\}$ zusammenhängend ist und fixieren dazu zwei beliebige Knoten $v \neq v' \in V$.

Enthält der Weg p zwischen v und v' in T die Kante e, so erhalten wir einen Weg p' zwischen v und v' in T', indem wir in p die Kante e durch das Kreisfragment $K \setminus \{e\}$ ersetzen.

Da zudem $|T'| = |T| = |V| - 1$, ist T' ein Spannbaum (Lemma 59).

Da $w(e) \leq w(e')$, gilt $w(T') \leq w(T)$, also ist T' auch ein MST, der zudem neben A auch e enthält. □

Schnitte, respektierendes A und Crosskanten



$S = \{a, b, g, h, i\}$

$A = \{(a, b), (a, h), (c, f), (g, h), (c, d), (d, e)\}$

Crosskanten $\{(b, c), (c, i), (f, g)\}$

Ein einfacher MST-Algorithmus für $G = (V, E, w)$

SimpleMST(G)

1 Fixiere beliebiges $v \in V$, setze $S \leftarrow \{v\}$, $A \leftarrow \emptyset$.
2 For $i \leftarrow 1$ to $|V| - 1$ do
3 Finde leichteste Kante $(v, w) \in E$ mit $v \in S$, $w \notin S$.
4 Setze $S \leftarrow S \cup \{w\}$, $A \leftarrow A \cup \{(v, w)\}$.
5 Gebe A aus

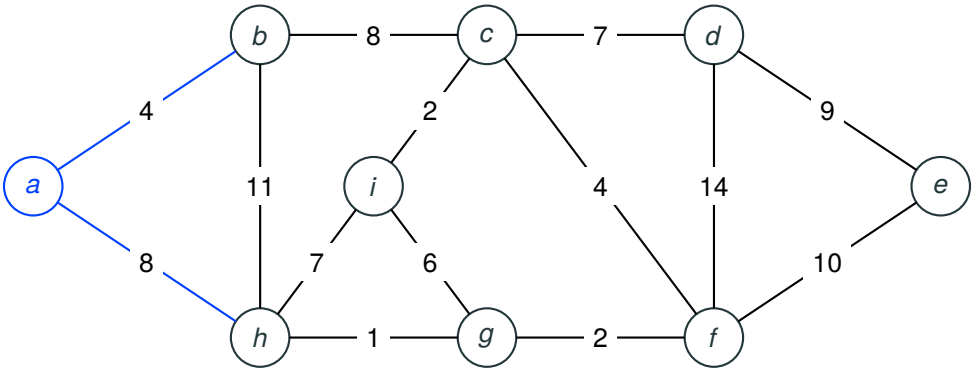
Korrektheit: In 3 wird stets eine leichteste Crosskante für $(S, V \setminus S)$ gewählt.

$S \leftarrow S \cup \{w\}$ in Schritt 4 stellt sicher, dass A stets den Schnitt $(S, V \setminus S)$ respektiert.

Laufzeit: Schritt 3 erfordert ggf. das Durchsuchen der gesamten Kantenmenge und damit Kosten von $\Theta(|E|)$. Das liefert **Gesamtkosten** von $\Theta(|V| \cdot |E|)$.

Frage: Gibt es schnellere MST-Algorithmen?

Beispiel *SimpleMST*(G)

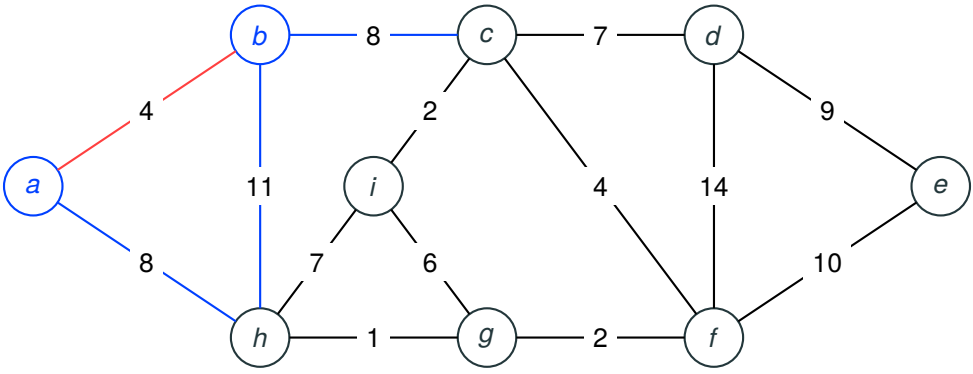


$S = \{a\}$,

$A = \emptyset$,

Cross Edges $\{(a,b),(a,h)\}$

Step 1

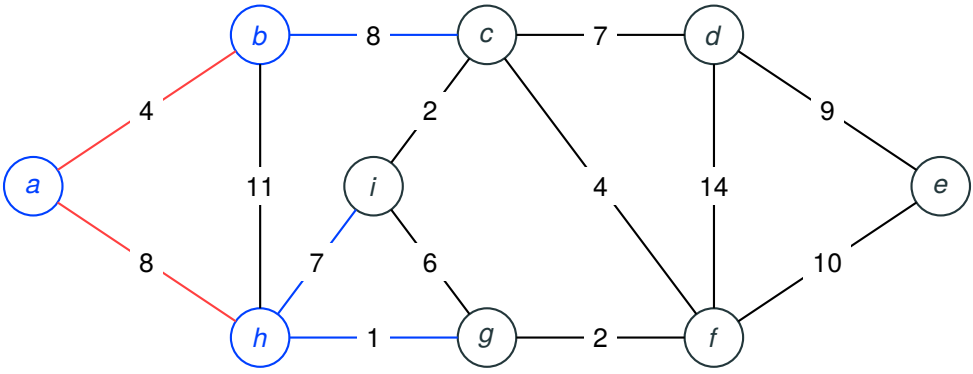


$S = \{a, b\}$,

$A = \{(a, b)\}$,

Cross Edges $\{(a, h), (b, c), (b, h)\}$

Step 2

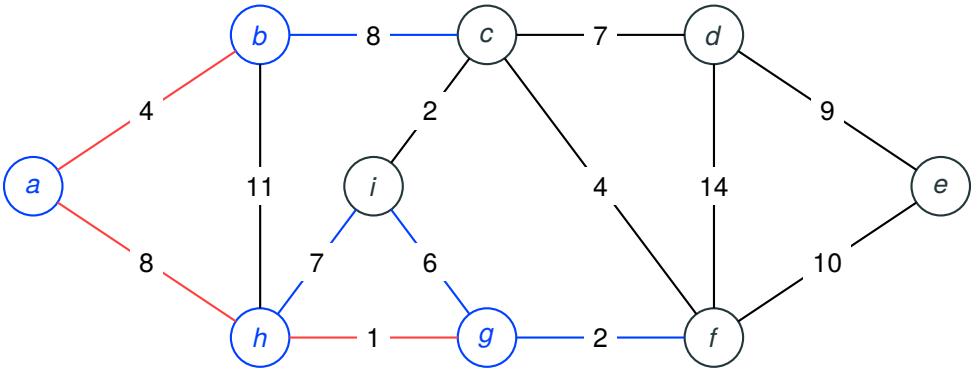


$S = \{a, b, h\}$,

$A = \{(a, b), (a, h)\}$,

Cross Edges $\{(b, c), (h, i), (h, g)\}$

Step 3

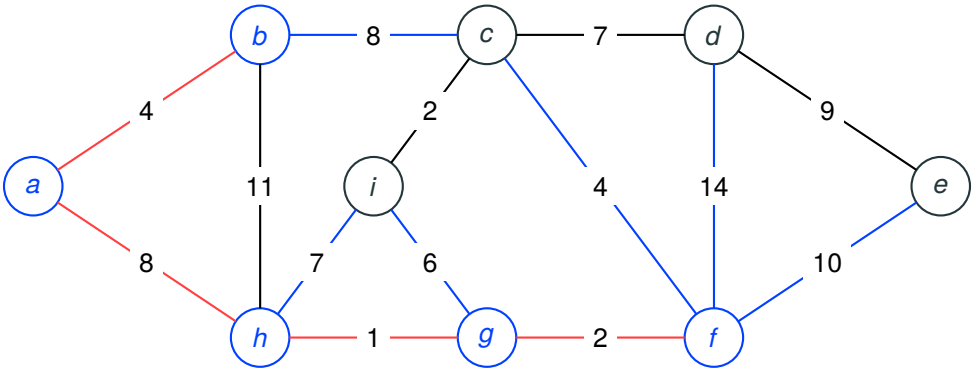


$S = \{a, b, h, g\}$,

$A = \{(a, b), (a, h), (g, h)\}$,

Cross Edges $\{(b, c), (h, i), (g, i), (f, g)\}$

Step 4

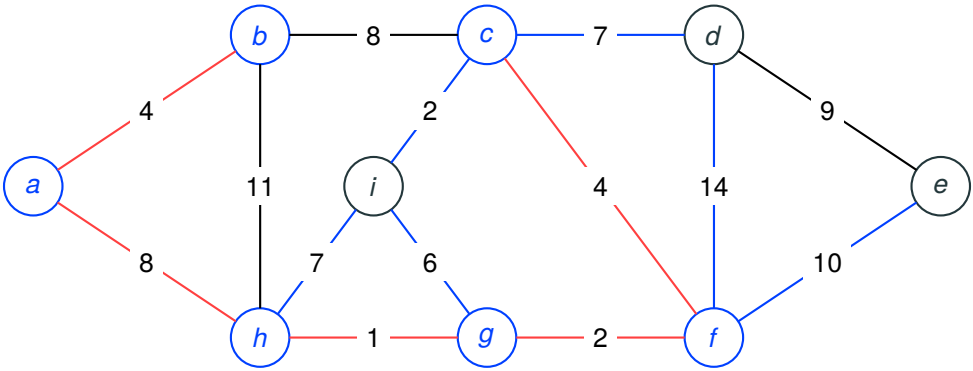


$S = \{a, b, h, g, f\}$,

$A = \{(a, b), (a, h), (g, h), (f, g)\}$,

Cross Edges $\{(b, c), (h, i), (g, i), (c, f), (d, f), (e, f)\}$

Step 5

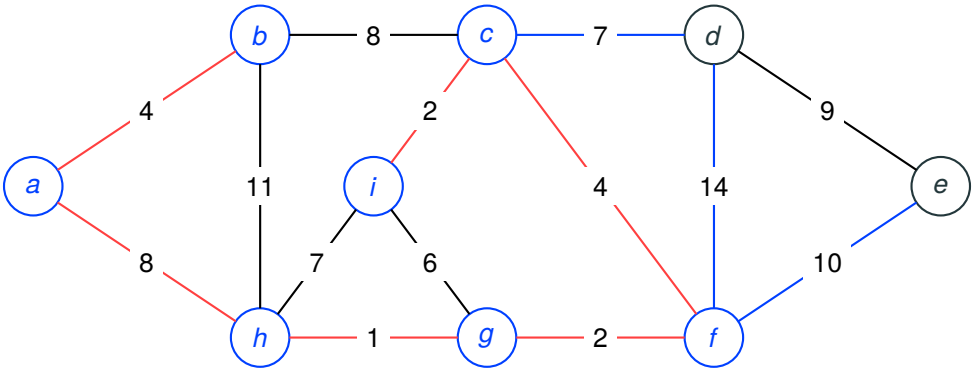


$S = \{a, b, h, g, f, c\}$,

$A = \{(a, b), (a, h), (c, f), (g, h), (f, g)\}$,

Cross Edges $\{(h, i), (g, i), (d, f), (e, f), (c, i), (c, d)\}$

Step 6

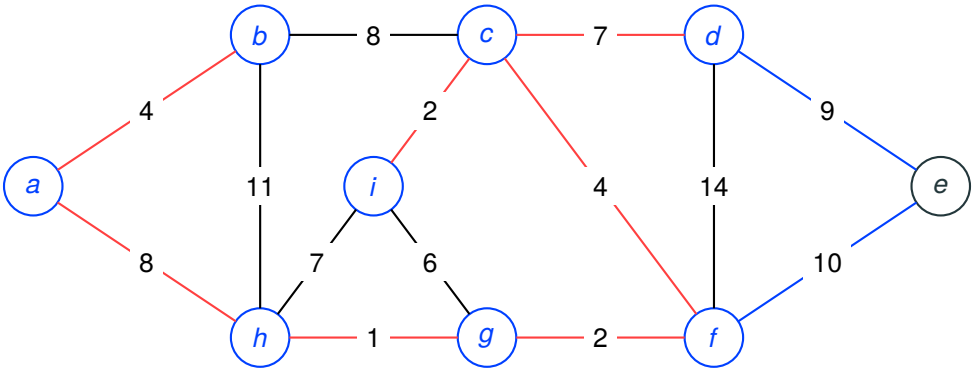


$S = \{a, b, h, g, f, c, i\}$,

$A = \{(a, b), (a, h), (c, f), (g, h), (f, g), (c, i)\}$,

Cross Edges $\{(d, f), (e, f), (c, d)\}$

Step 7

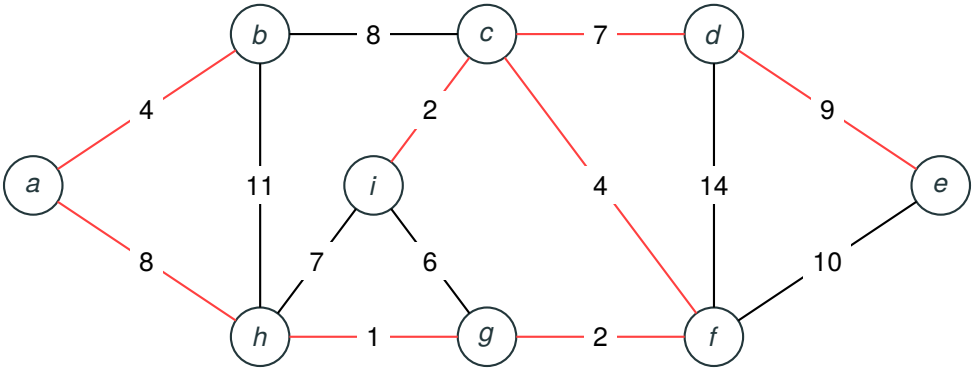


$S = \{a, b, h, g, f, c, i, d\}$,

$A = \{(a, b), (a, h), (c, f), (g, h), (f, g), (c, i), (c, d)\}$,

Cross Edges $\{(d, e), (e, f)\}$

Step 8



$A = \{(a,b),(a,h),(c,f),(g,h),(f,g),(c,i),(c,d),(d,e)\}$,

Berechnung Minimaler Spannbäume (MSTs) in gewichteten ungerichteten Graphen

Der MST-Algorithmus von Kruskal

Der MST-Algorithmus von Kruskal, Idee

Gegeben $G = (V, E, w)$, so wird ein MST A gemäß dem generischen Algorithmus entsprechend folgender Regel aufgebaut:

- Am Anfang gilt $A = \emptyset$.
- Wenn $|A| < |V| - 1$ so zerfällt A in mehr als eine Zusammenhangskomponente (Lemma 59)
- Die Zusammenhangskomponenten von A werden als Union-Find Datenstruktur verwaltet.
- In jeder Iteration gilt $A \leftarrow A \cup \{e\}$, wobei e die leichteste Kante ist, die zwei der Zusammenhangskomponenten von A verbindet.
- Nach Theorem 62 ist damit A stets Teilmenge eines MST und e stets sicher für A.
- Das wird erreicht, indem die Kanten nach aufsteigendem Gewicht geordnet werden.
- In dieser Reihefolge wird für jede Kante e getestet, ob sie zwei verschiedene Zusammenhangskomponenten von A verbindet.
- In diesem Fall wird e zu A hinzugefügt.

MSTKruskal(G), $G = (V, E, w)$ zusammenhängend

MSTKruskal(G)

```
1 A ← ∅
2 For each v ∈ V
3     do MakeSet(v)
4 Sort E nondecreasingly w.r.t w
5 For each e = (u, v) ∈ G.E
6     do if Find(u) ≠ Find(v)
7           then A ← A ∪ {e}
8                Union(u, v)
9 return A
```

Laufzeit: Die Laufzeit für das Sortieren in Zeile 4 ist $O(|E| \cdot \log |E|)$.

Die Gesamtlaufzeit der **For** Schleife in den Zeilen 5-8 ist ebenfalls $O(|E| \cdot \log |E|)$, falls Linked Lists oder Disjoint Set Trees als Union-Find Datenstruktur verwendet werden.

Korrektheits von MST-Kruskal

Theorem 63

Für alle Kanten $e = (u, v) \in E$ ist das Folgende wahr:

Gilt beim Durchlauf von Zeile 6 mit e, dass $Find(u) \neq Find(v)$, so ist e eine leichte Crosskante bezüglich eines Schnittes, der die bis dahin berechnete Menge $A \subseteq E$ respektiert.

Damit ist $A \cup \{e\}$ nach Theorem 62 stets eine Teilmenge eines MST.

Außerdem ist (V, A) nach Beendigung des Algorithmus zusammenhängend und damit nach Lemma 59 ein MST.

Beweis: Wir betrachten eine beliebige Kante $e = (u, v) \in E$, für die beim Durchlauf von Zeile 6 $Find(u) \neq Find(v)$ gilt.

Wir setzen per Induktion voraus, dass die bislang in Zeile 8 berechnete Partition der Knotenmenge den Zusammenhangskomponenten des Graphen (V, A) entspricht.

Der Beweis von Theorem 63

Das gilt offensichtlich für die erste Kante aus E mit minimalem Gewicht.

Wir erhalten einen A respektierender Schnitt $(S, V \setminus S)$, bezüglich dessen $e = (u, v)$ eine Crosskante ist, indem wir die Zusammenhangskomponente von u nach S und die restlichen Zusammenhangskomponenten nach $V \setminus S$ verlagern.

Alle Kanten $e' = (u', v')$, für die $w(e') < w(e)$ gilt, haben Zeile 6 vor e durchlaufen, und nach deren Durchlauf gilt stets, dass $\mathit{Find}(u') = \mathit{Find}(v')$.

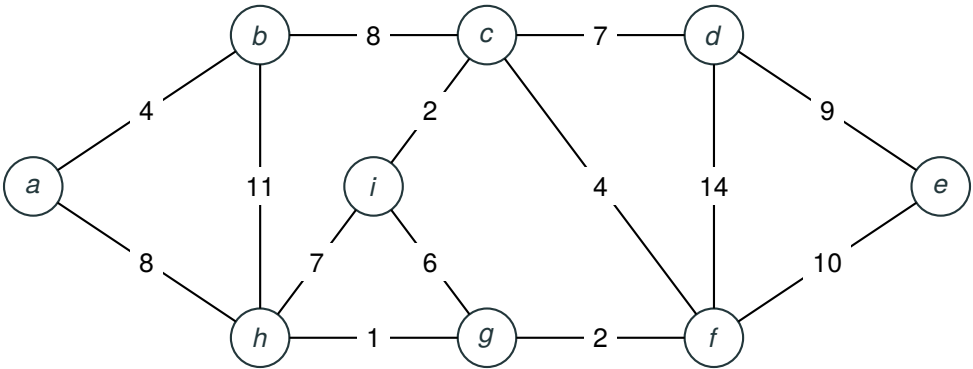
Also ist e leichte Crosskante bezüglich $(S, V \setminus S)$, da e' keine Crosskante bezüglich $(S, V \setminus S)$ sein kann.

Damit werden zu A stets sichere Kanten hinzugefügt, d.h., A ist stets eine Teilmenge eines MST.

Zudem ist nach Beendigung des Algorithmus $\mathit{Find}(u) = \mathit{Find}(v)$ für alle $e = (u, v)$ in E.

Daraus folgt, dass dann (V, A) zusammenhängend und damit ein MST ist. $\square$

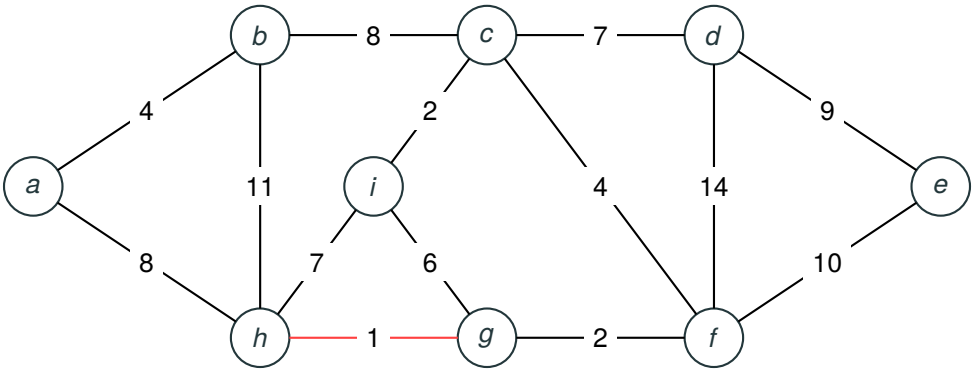
Beispiel *MSTKruskal*(G)



$(g,h),(c,i),(f,g),(a,b),(c,f)(g,i),(c,d),(h,i),(a,h),(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}$

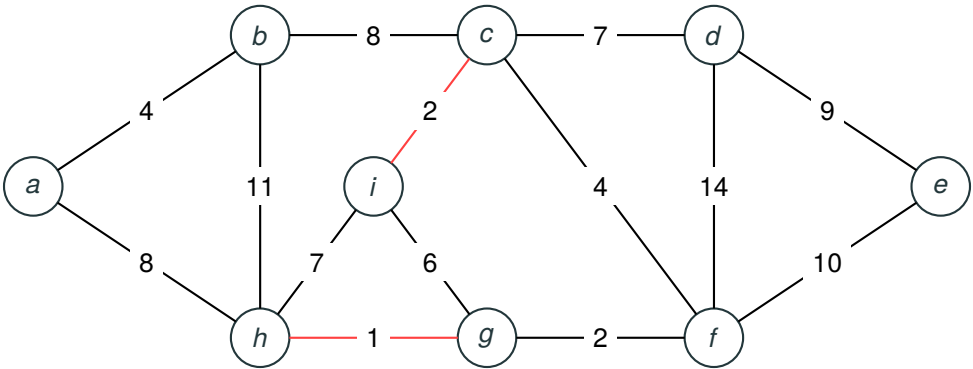
Step 1



$(c,i),(f,g),(a,b),(c,f)(g,i),(c,d),(h,i),(a,h),(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g,h\},\{i\}$

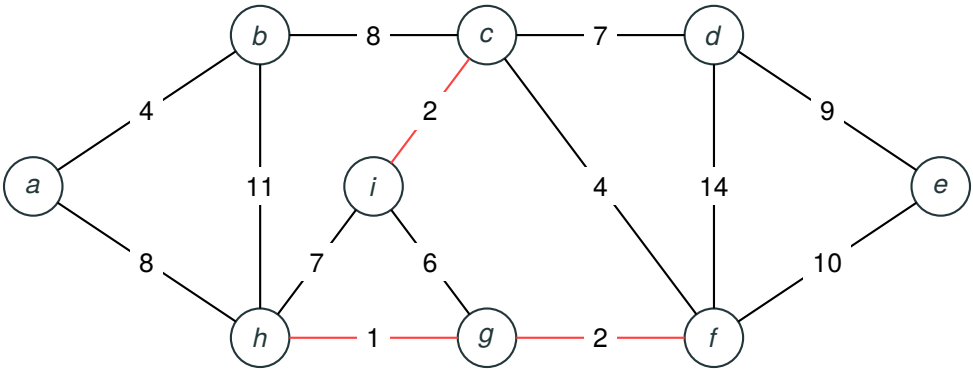
Step 2



$(f,g),(a,b),(c,f)(g,i),(c,d),(h,i),(a,h),(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a\},\{b\},\{c,i\},\{d\},\{e\},\{f\},\{g,h\}$

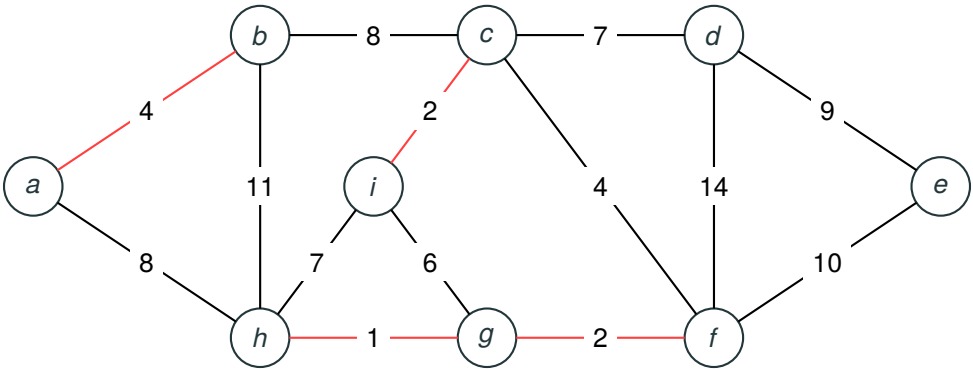
Step 3



$(a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)$

Partition $\{a\}, \{b\}, \{c,i\}, \{d\}, \{e\}, \{f,g,h\}$

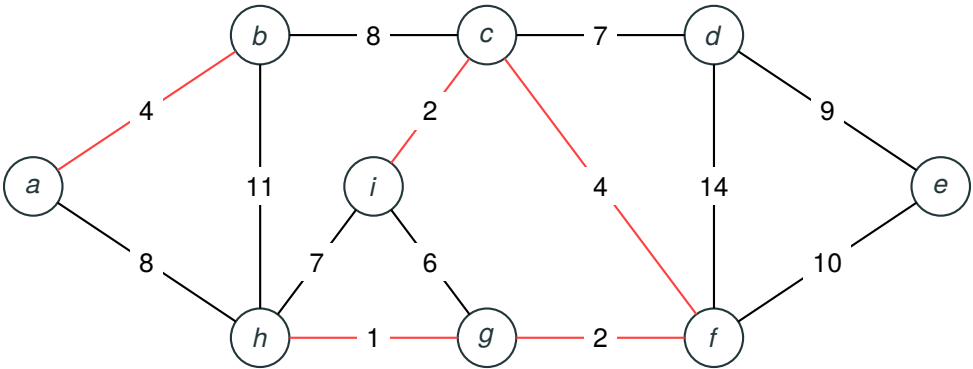
Step 4



$(c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)$

Partition $\{a,b\}, \{c,i\}, \{d\}, \{e\}, \{f,g,h\}$

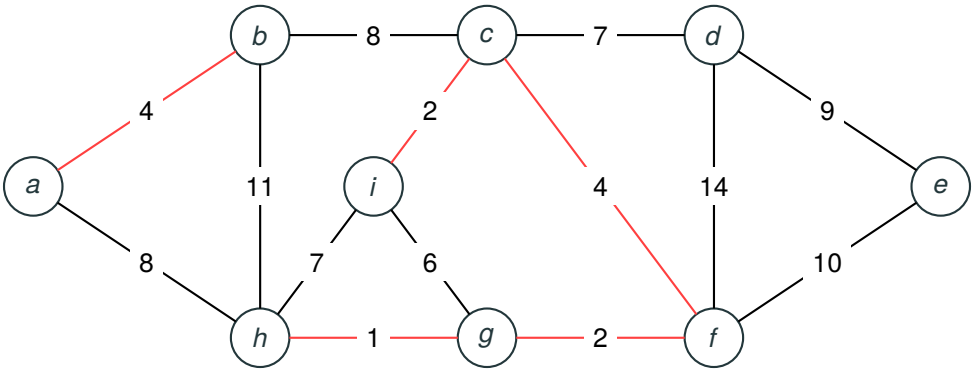
Step 5



$(g,i),(c,d),(h,i),(a,h),(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a,b\},\{c,i,f,g,h\},\{d\},\{e\}$

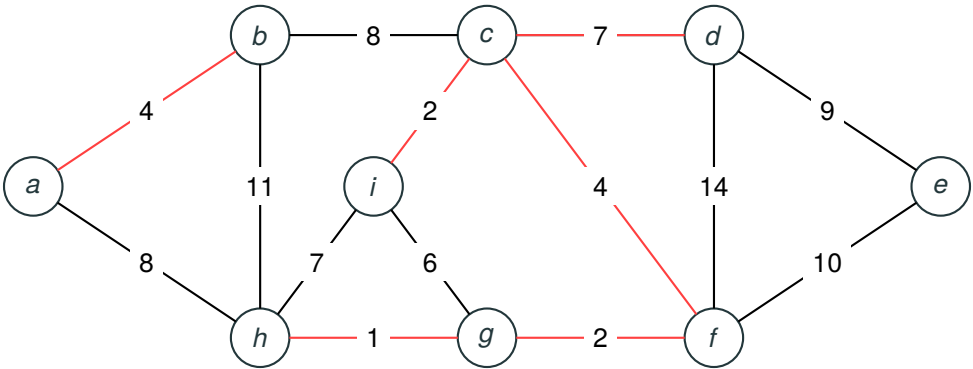
Step 6



$(c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)$

Partition $\{a,b\}, \{c,i,f,g,h\}, \{d\}, \{e\}$

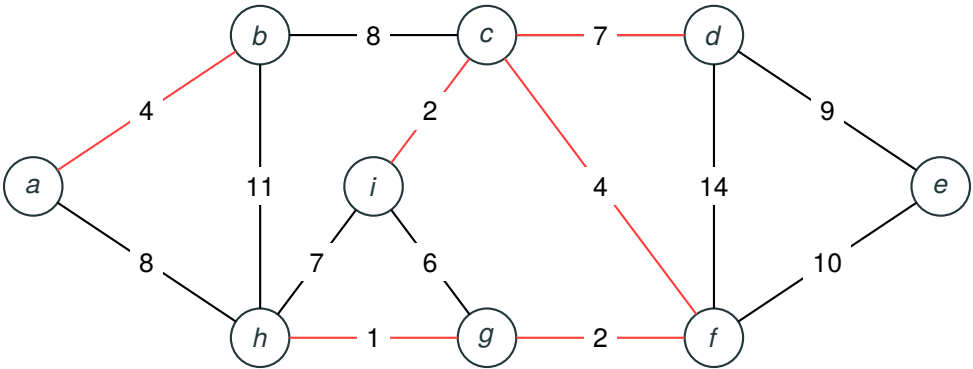
Step 7



$(h,i),(a,h),(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a,b\},\{c,i,f,g,h,d\},\{e\}$

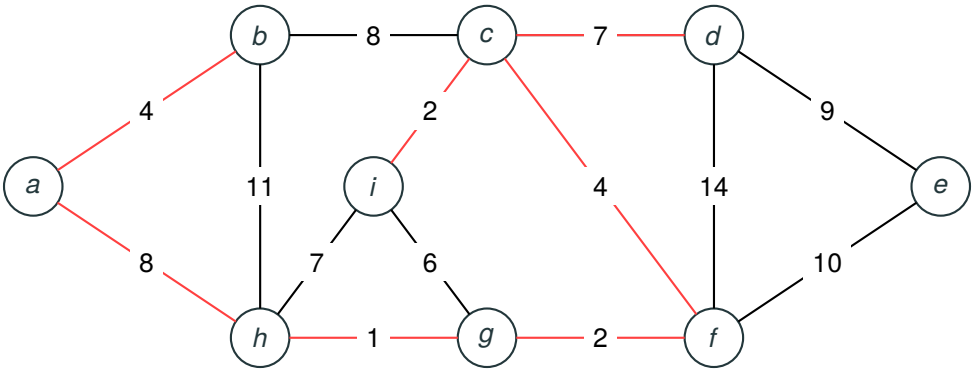
Step 8



$(a,h),(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a,b\},\{c,i,f,g,h,d\},\{e\}$

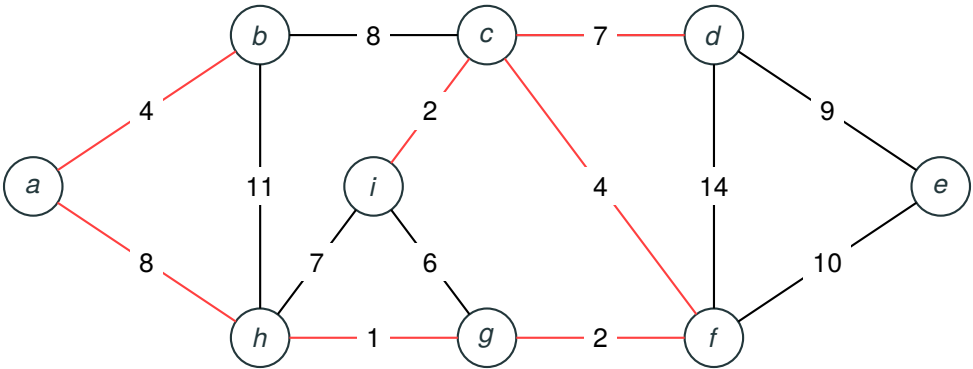
Step 9



$(b,c),(d,e),(e,f),(b,h),(d,f)$

Partition $\{a,b,c,i,f,g,h,d\},\{e\}$

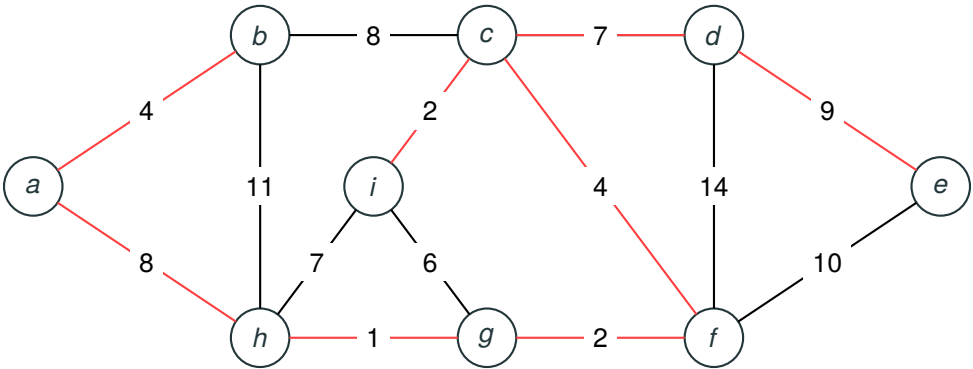
Step 10



$(d,e),(e,f),(b,h),(d,f)$

Partition $\{a,b,c,i,f,g,h,d\},\{e\}$

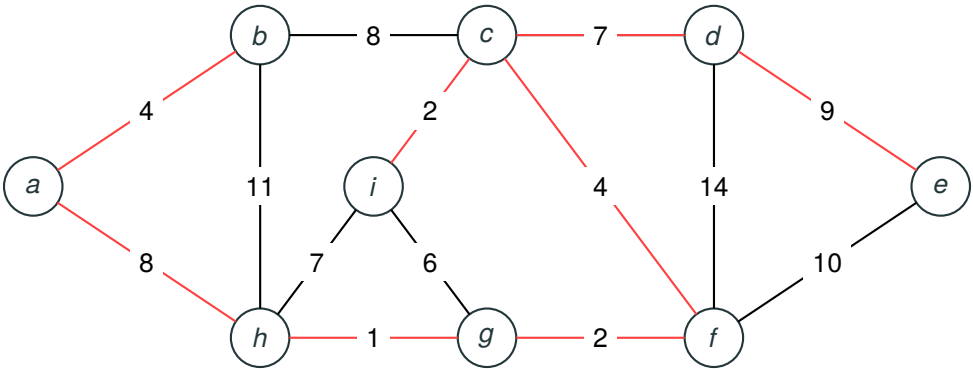
Step 11



$(e, f), (b, h), (d, f)$

Partition $\{a, b, c, i, f, g, h, d, e\}$

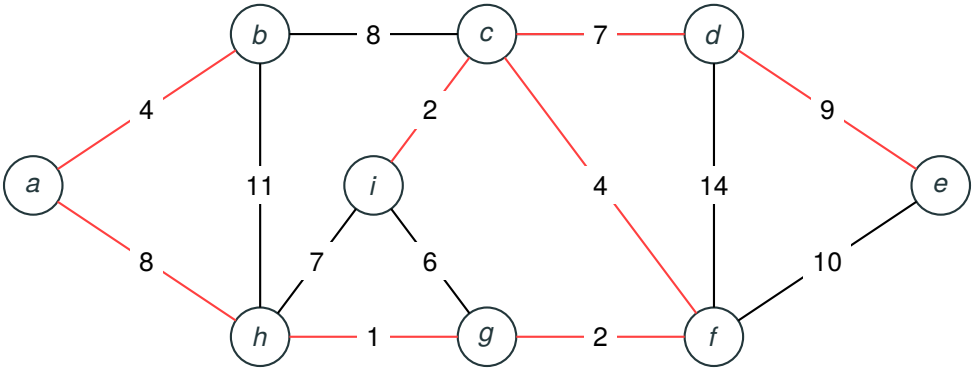
Step 12



$(b,h),(d,f)$

Partition $\{a,b,c,i,f,g,h,d,e\}$

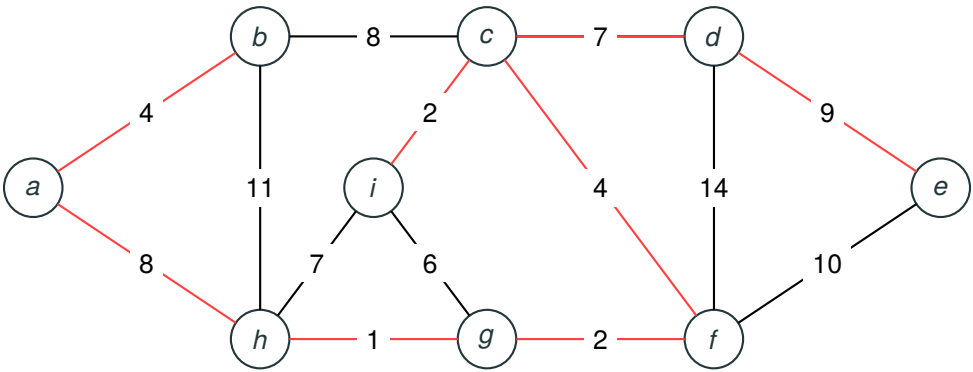
Step 13



(d, f)

Partition $\{a, b, c, i, f, g, h, d, e\}$

Step 14



Berechnung Minimaler Spannbäume (MSTs) in gewichteten ungerichteten Graphen

Der MST-Algorithmus von Prim

Der Algorithmus von Prim, die Idee

- Der Algorithmus von Prim verwaltet dynamisch eine Partition S, Q von V und für jeden Knoten $v \in Q$ eine Information $\pi(v) \in S \cup \{NIL\}$ und eine Information $v.key \in \mathbb{R} \cup \{\infty\}$.
- Am Anfang ist $S = \emptyset$, sowie $\pi(v) = NIL$ und $v.key = \infty$ für alle $v \in Q = V$.
- Dann wird $r.key = 0$ und $S = \{r\}$ gesetzt, wobei $r \in V$ ein beliebig fixierte Knoten ist.
- Im weiteren Verlauf gilt für alle $v \in Q$, dass $v.key = \infty$ genau dann, wenn es keine Kante von v nach S gibt.
- Ist $v.key < \infty$ so bezeichnet $(v, v.\pi)$, $v.\pi \in S$, stets die **leichteste Kante** von v nach S, und es gilt $v.key = w((v, v.\pi))$.
- Der Algorithmus von Prim wählt in jeder seiner $|V| - 1$ Iterationen den Knoten $v \in Q$ mit dem minimalen $v.key$ Wert und fügt $(v, v.\pi)$ zu A und v zu S hinzu.
- **Damit wird stets die leichteste Crosskante von S, Q zu A hinzugefügt**, der Algorithmus ist also korrekt.

$MSTPrim(G, w, r)$, $G = (V, E, w)$, $r \in V$

$MSTPrim(G, w, r)$

```
 1 For all v ∈ V
 2     do v.key ← ∞, v.π ← NIL
 3 r.key ← 0
 4 Q ← V
 5 S ← A ← ∅
 6 while Q ≠ ∅
 7     do u ← min(Q) (*bzgl. u.key*)
 8        if u ≠ r then A ← A ∪ {(u, u.π)}
 9        S ← S ∪ {u}, Q ← Q \ {u}
10        For all v ∈ Adj_G[u]
11            do if (v ∈ Q) ∧ (v.key > w(u, v))
12                  then v.key ← w(u, v)
13                       v.π ← u
```

Korrektheit und Laufzeit von $MSTPrim(G, w, r)$

(1) Zur Korrektheit genügt es zu zeigen, dass nach jeder Iteration für alle $v \in Q$ gilt, dass $v.key = \min\{w(v,u); u \in S\}$.

(2) Nach Iteration 1 gilt $v.key = w(r,v)$ für alle $v \in Adj_G[r]$, Aussage (1) ist somit wahr da $S = \{r\}$.

(3) Wir setzen nun voraus, dass für $i > 1$ Aussage (1) nach den Iterationen $1, \cdots, i-1$ wahr ist. Die Aktualisierung von $v.key$ in Iteration i entsprechend dem neuen Knoten u in S wird in den Zeilen 10-13 korrekt vorgenommen.

(4) Zur Optimierung der Laufzeit wird Q als eine auf einem MinHeap beruhende Priority Queue organisiert.

(5) Damit kosten die Operationen in Zeile 7 und Zeile 12 jeweils $O(\log(|V|)$, wobei Zeile 7 $|V|$ Mal und Zeile 12 höchstens $|E|$ Mal ausgeführt wird.

(6) Damit beträgt die Gesamtlaufzeit $O(|V| \cdot \log(|V|) + |E| \cdot \log(|V|))$.

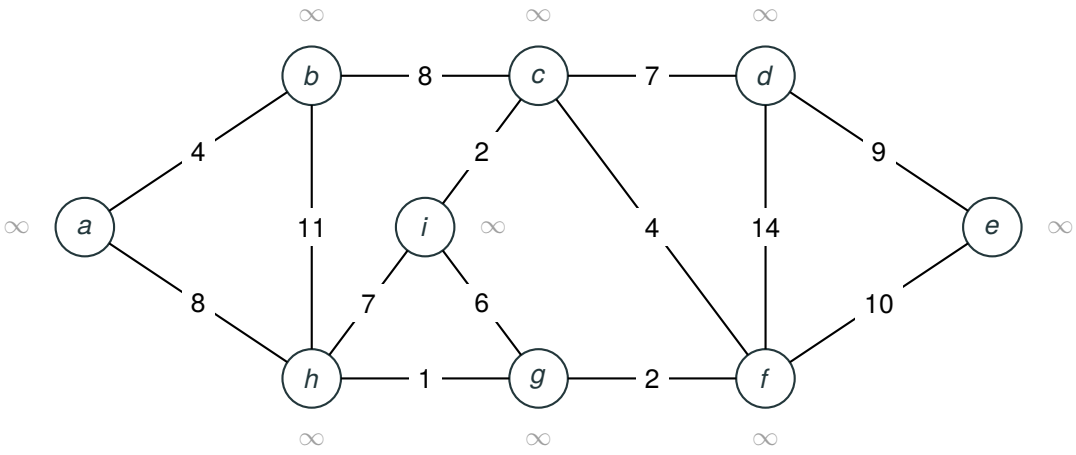
(7) Bei Verwendung sogenannter **Fibonacci-Heaps** kann die Laufzeit auf $O(|V| \cdot \log(|V|) + |E|)$ gedrückt werden.

$MSTPrim(G, w, r)$, $r \in V$, mit MinHeap auf Q

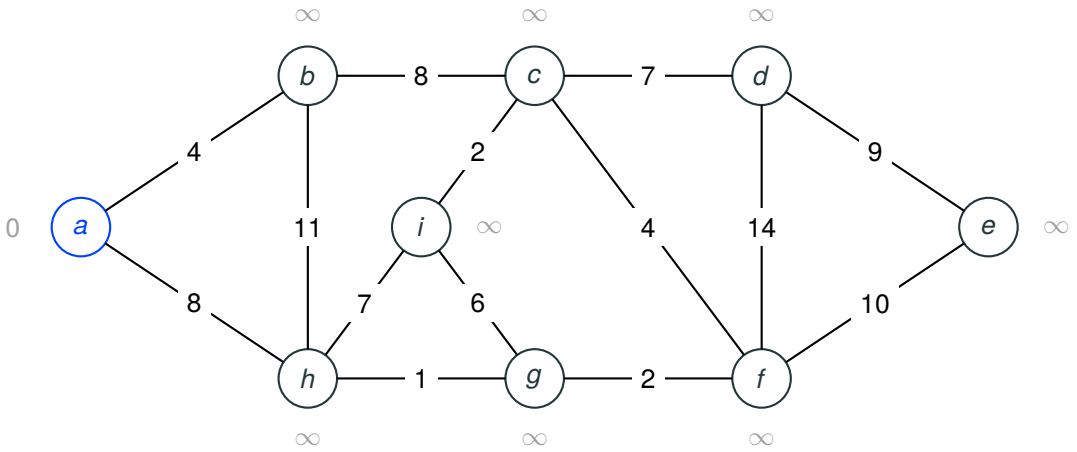
$MSTPrim(G, w, r)$

```
 1 For all v ∈ V
 2     do v.key ← ∞, v.π ← NIL
 3 r.key ← 0
 4 Q ← BuildHeap(V)
 5 S ← A ← ∅
 6 while heapsize(Q) > 0
 7    do u ← ExtractMin(Q)
 8       if u ≠ r then A ← A ∪ {(u, π(u))}
 9       S ← S ∪ {u}
10       For all v ∈ Adj_G[u]
11            do if (v ∈ Q) ∧ (v.key > w(u, v))
12                 then Decrease(Q, v.key, w(u, v))
13                      v.π ← u
```

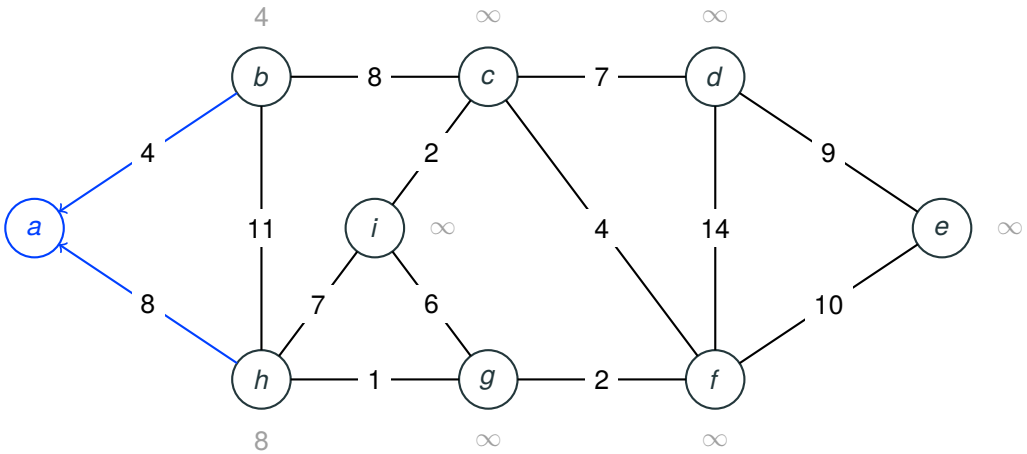
Beispiel $MSTPrim(G, a)$



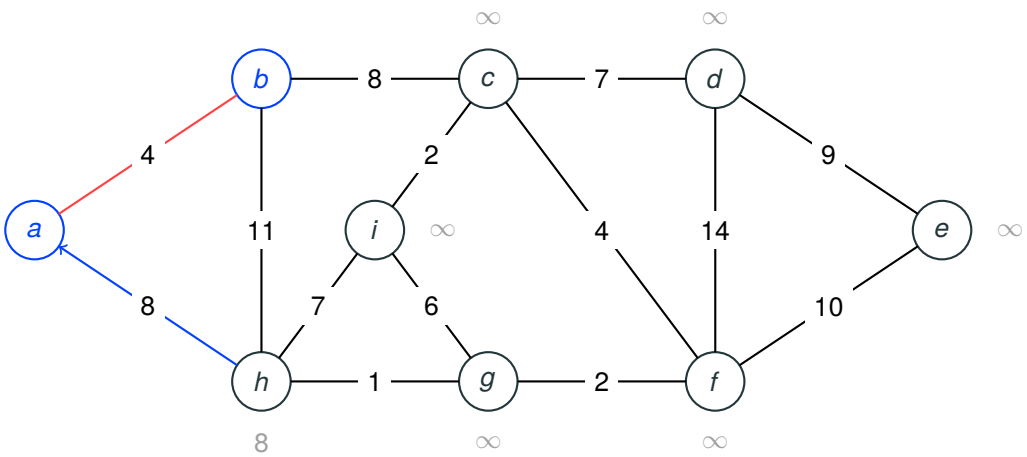
Step 0.1



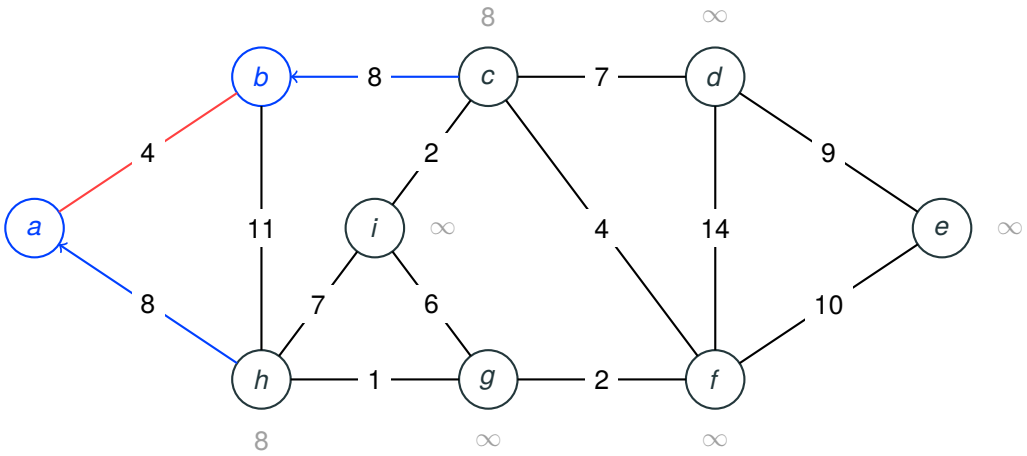
Step 0.2



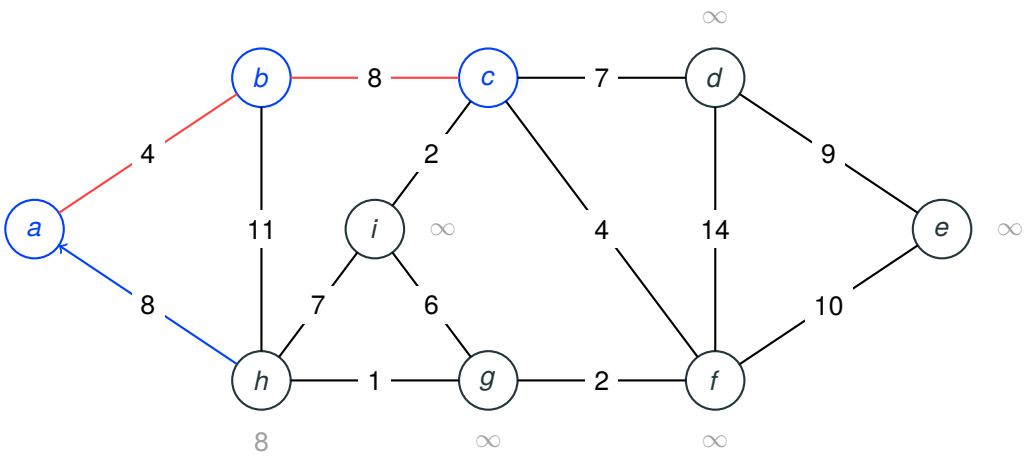
Step 1.1



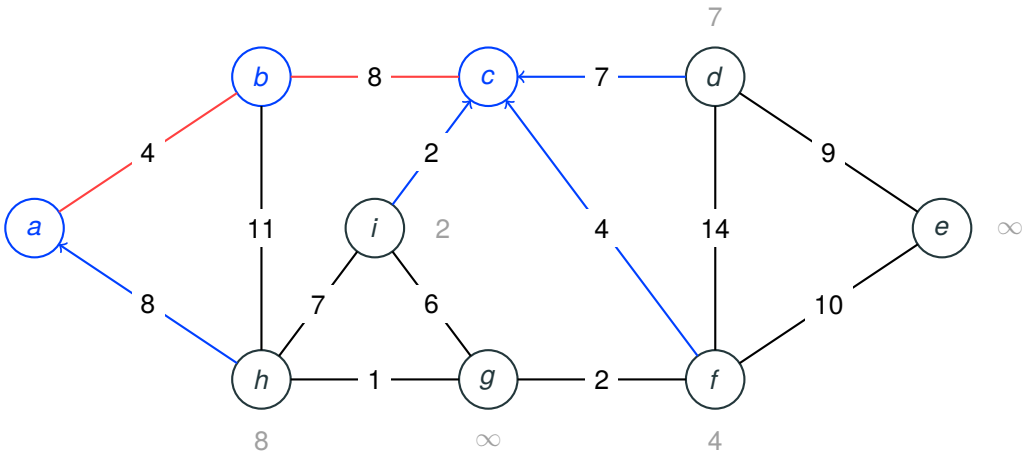
Step 1.2



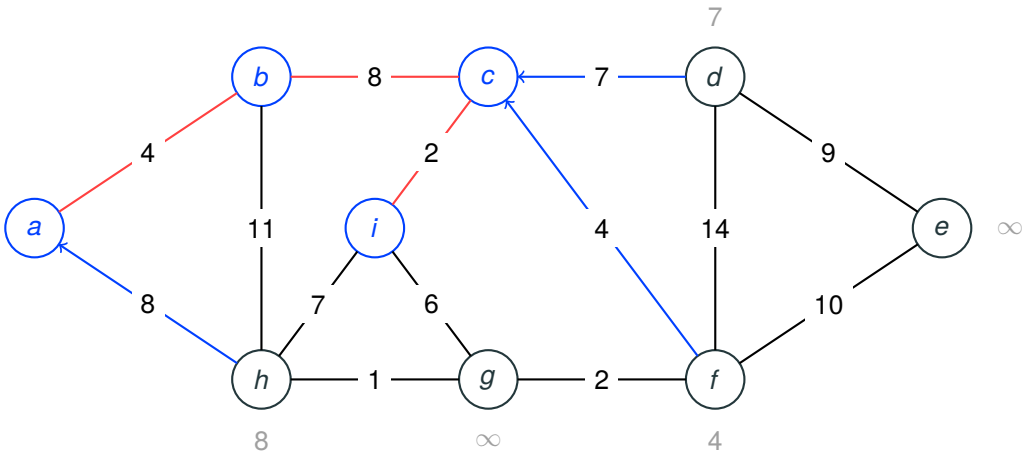
Step 2.1



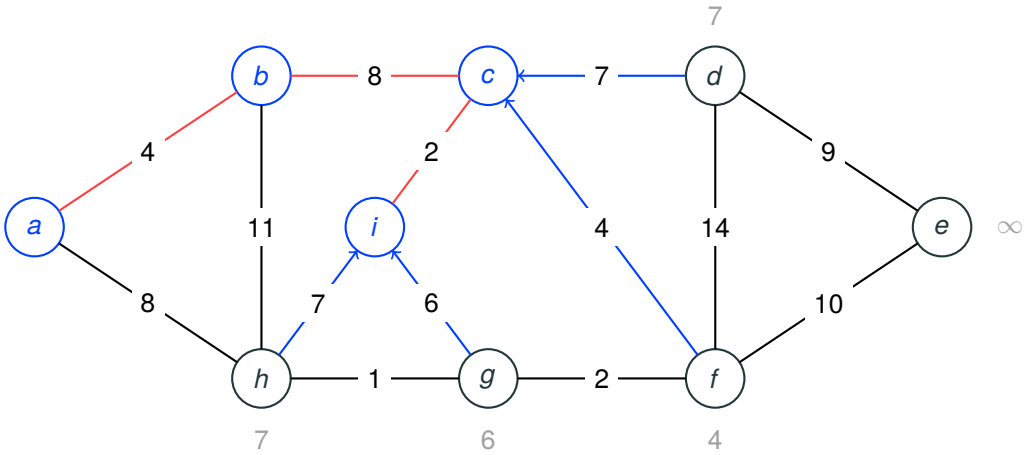
Step 2.2



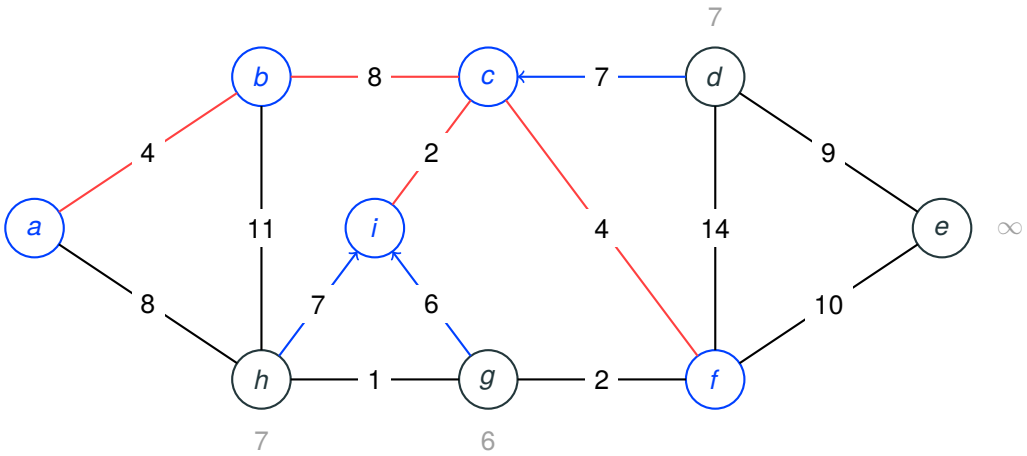
Step 3.1



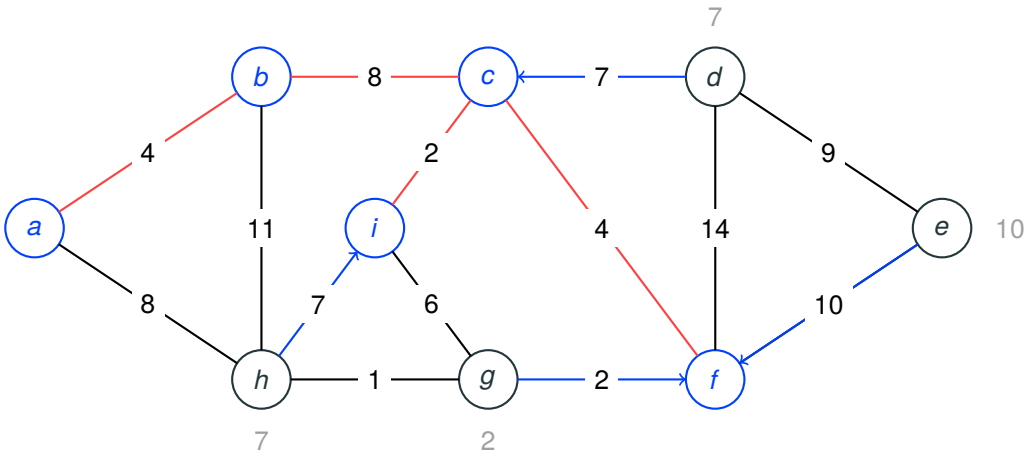
Step 3.2



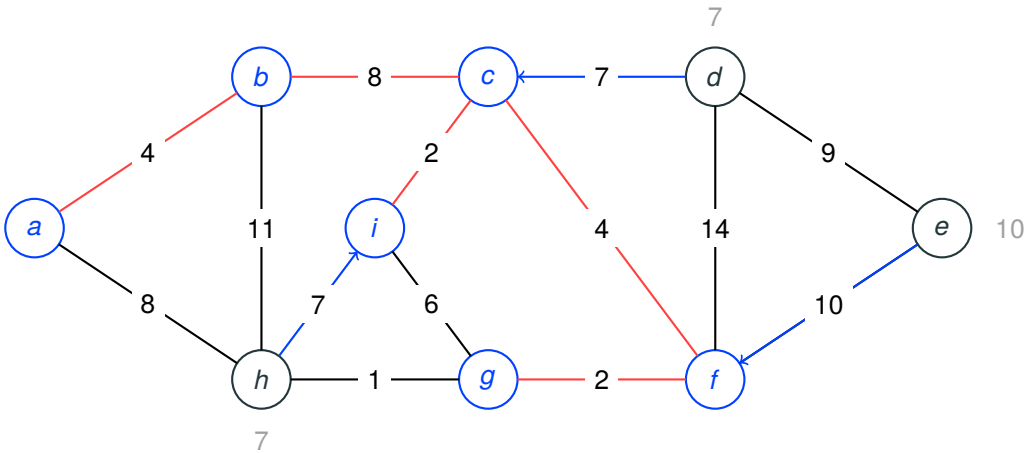
Step 4.1



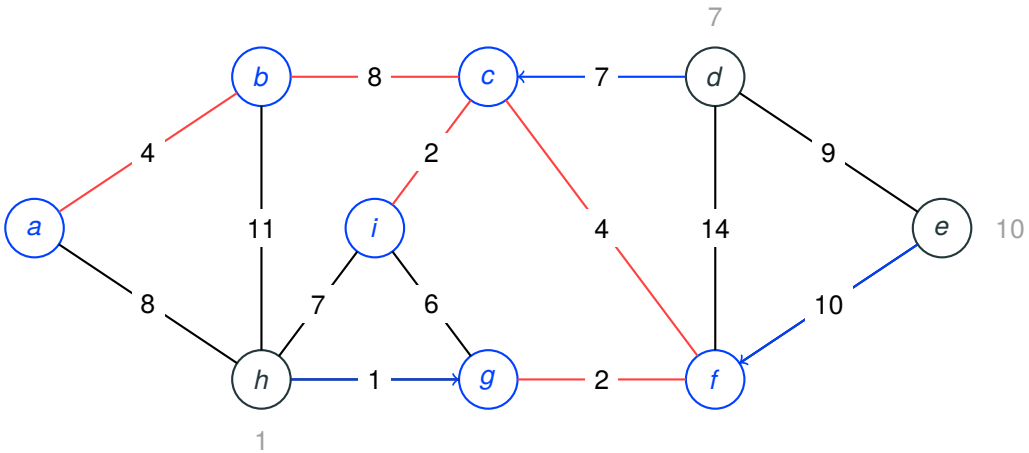
Step 4.2



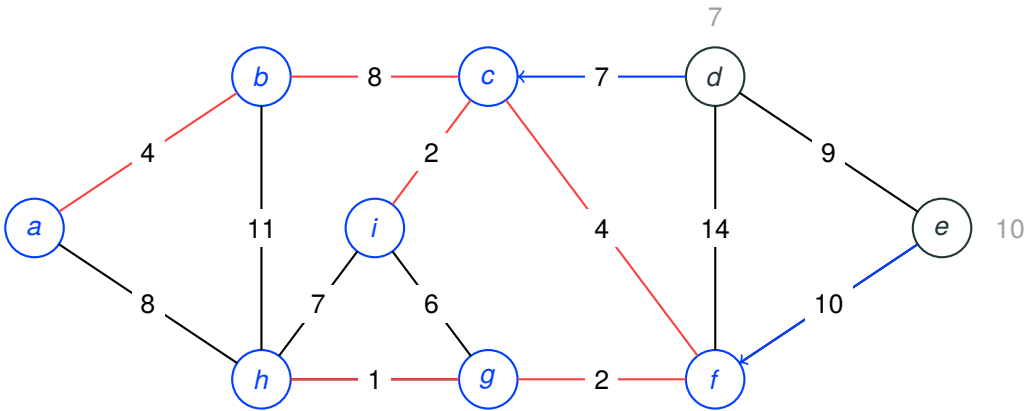
Step 5.1



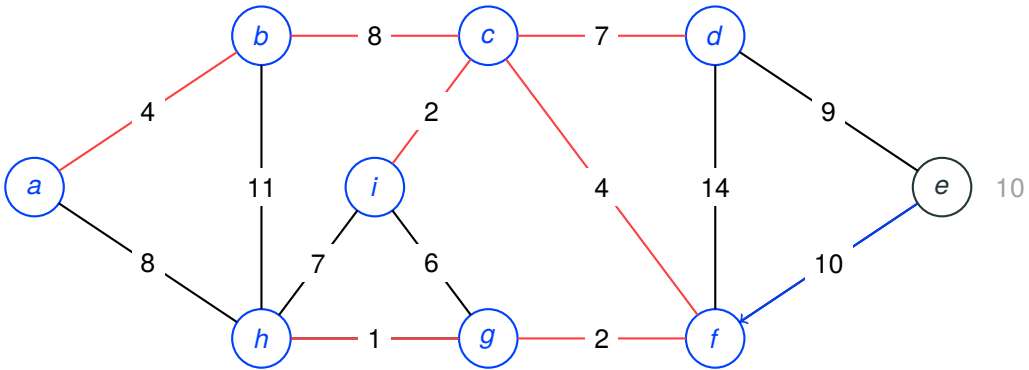
Step 5.2



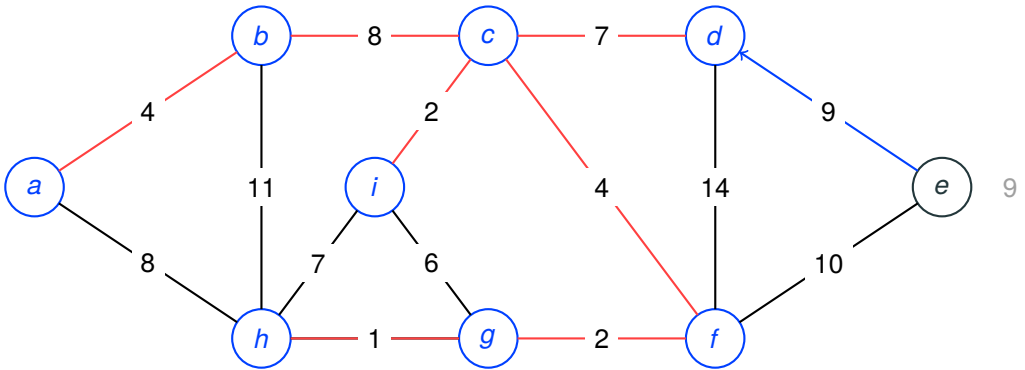
Step 6.1



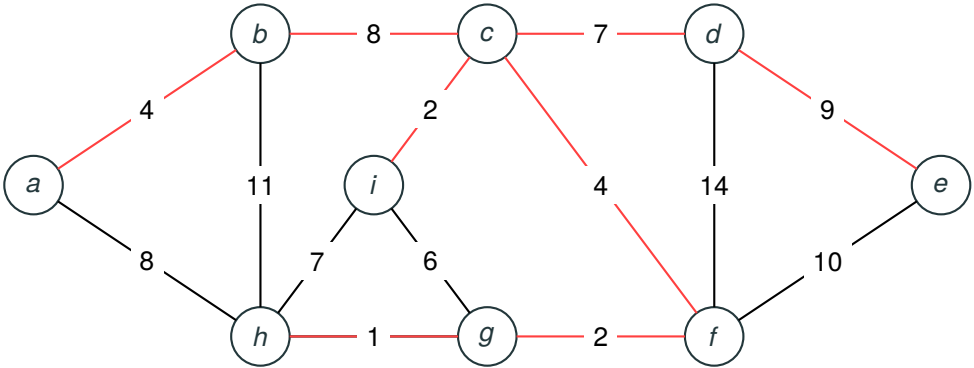
Step 7.1



Step 7.2



Step 8.1



Kürzeste Wege in gewichteten gerichteten Graphen, Single Source Shortest Path

Kürzeste Wege in gewichteten gerichteten Graphen, Single Source Shortest Path

Single Source Shortest Path, Grundlagen

Kürzeste Wege in gewichteten Graphen

Es sei $G = (G.V, G.E, w)$ ein Graph mit Kantengewichtsfunktion $w : E \longrightarrow \mathbb{R}$.

- **Metrik auf $G.V$:** Für alle $u \neq v \in G.V$ sei

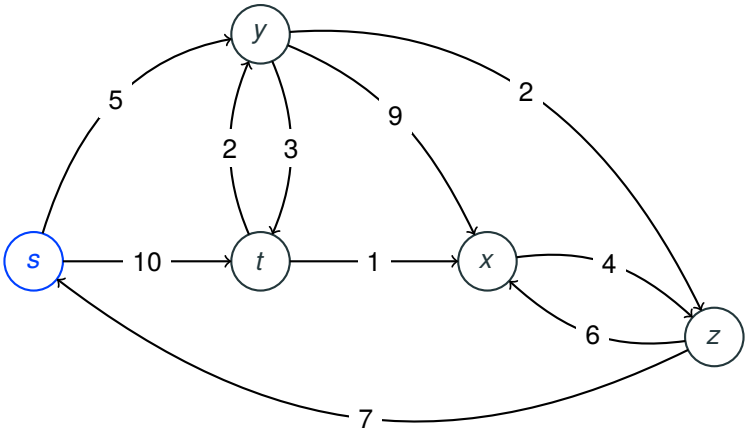
$$\delta_G(u, v) = \min\{w(p), p \text{ gerichteter Weg von } u \text{ nach } v\}.$$

- $\delta_G(u, v) = -\infty$ bedeutet, dass es **beliebig kurze Wege** von u nach v gibt.
- Das ist genau dann der Fall, wenn es einen Weg von u nach v gibt, der einen **Kreis mit negativem Gewicht** enthält.
- $\delta_G(u, v) = \infty$ bedeutet, dass es **keinen Weg** von u nach v in G gibt.
- Ist $-\infty < \delta_G(u, v) < \infty$, so heißt ein gerichteter Pfad p von u nach v mit $w(p) = \delta_G(u, v)$ **kürzester Weg von u nach v**.

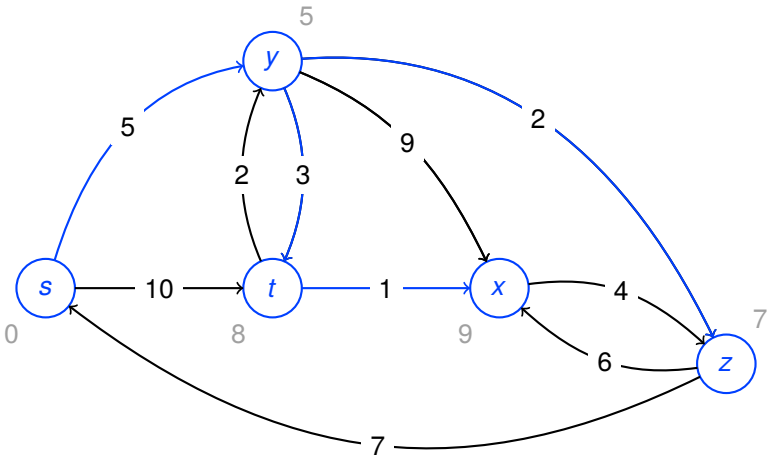
Kürzeste Wege Probleme

- Das **Single Pair Shortest Path Problem:** Berechne für **ein Knotenpaar** $u \neq v \in G.V$ einen kürzesten Weg von u nach v (bzw. dessen Länge $\delta_G(u, v)$).
- Das **All Pairs Shortest Path Problem:** Berechne für **alle Knotenpaare** $u \neq v \in G.V$ einen kürzesten Weg von u nach v (bzw. dessen Länge $\delta_G(u, v)$).
- Das **Single Source Shortest Path Problem:** Berechne für **alle Knotenpaare** $s \neq v \in G.V$ einen kürzesten Weg von s nach v (bzw. dessen Länge $\delta_G(s, v)$), wobei s ein **fixierter Startknoten** ist.
- **Ausgabe-Datenstruktur** des **Single Source Shortest Path Problem:** Für G, s wird ein **Kürzeste-Wege-Baum** G_π berechnet.
- D.h. G_π ist ein Unterbaum von G mit Wurzel s, der genau die Knoten enthält, die von s aus erreichbar sind, und alle Wege in G_π entsprechen kürzesten Wegen in G.

Beispiel Eingabe G, w, s für das Single Source Shortest Path Problem



Ausgabe Kürzeste-Wege-Baum mit Wurzel *s*



Eigenschaften kürzester Wege: Kreisfreiheit

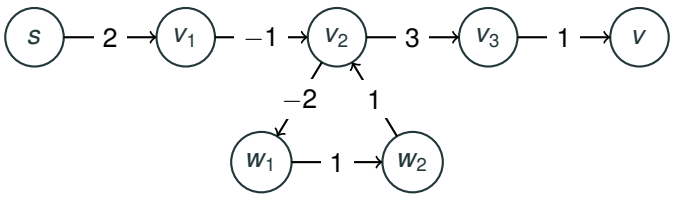
Lemma 64

*Von der Wurzel $s \in G.V$ sei **kein Kreis mit negativem Gewicht** in G erreichbar.*

Sei $v \in G.V$ von s aus erreichbar.

*Dann existiert ein kürzester Weg p von s nach v, der **einfach (also kreisfrei)** ist, also höchstens $|V| - 1$ Kanten besitzt.*

Beweis: Enthält p einen Kreis, so hat dieser nicht negatives Gewicht und kann deswegen gestrichen werden. □



Eigenschaften kürzester Wege: Kreisfreiheit

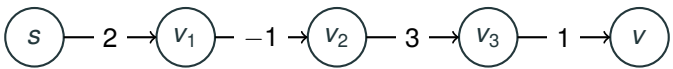
Lemma 65

*Von der Wurzel $s \in G.V$ sei **kein Kreis mit negativem Gewicht** in G erreichbar.*

Sei $v \in G.V$ von s aus erreichbar.

*Dann existiert ein kürzester Weg p von s nach v, der **einfach (also kreisfrei)** ist, also höchstens $|V| - 1$ Kanten besitzt.*

Beweis: Enthält p einen Kreis, so hat dieser nicht negatives Gewicht und kann deswegen gestrichen werden. □



Eigenschaften kürzester Wege: Monotonie

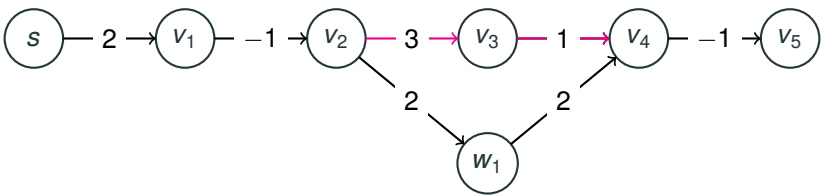
Lemma 66

*Sei $p = (v_0, \cdots, v_k)$ ein **kürzester Weg** von $v_0 = s$ nach $v_k = v$.*

*Dann ist für alle $0 \leq i < j \leq k$ der **Teilweg** $(v_i, \cdots, v_j)$ von p ein **kürzester Weg** von v_i nach v_j.*

Beweis:

Gäbe es einen kürzeren Weg q von v_i nach v_j, so wäre $(v_0, \cdots, v_{i-1}, q, v_{j+1}, \cdots, v_k)$ ein Weg von s nach v, der kürzer als p ist, Widerspruch. □



Single-Source-Shortest-Path: Datenstrukturen und Initialisierung

Die hier diskutierten Single-Source-Shortest-Path Algorithmen für Eingabegraphen $G = (V, E, w)$ verwenden die folgenden Datenstrukturen.

- Knoten $v \in V$ tragen das **Distanzattribut** $v.d \in \mathit{Reals} \cup \{\mathit{infty}\}$.
- Knoten $v \in V$ tragen zudem ein **Zeigerattribut** $v.\pi \in V \cup \{\mathit{NIL}\}$.
- Die Attribute definieren stets einen **Untergraphen** $G_\pi = (V_\pi, E_\pi)$ **von** G mit $V_\pi = \{v \in V; v.d \neq \infty\}$ und $E_\pi = \{(v.\pi, v); v \in V, v.\pi \neq \mathit{NIL}\}$.

G wird initialisiert durch

$\underline{\mathit{Initialize}(G, s)}$

```
1 For all v ∈ G.V
2     do v.d ← ∞
3        v.π ← NIL
4 s.d ← 0
```

Laufzeit $O(|V|)$

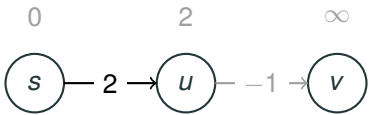
Die Grundoperationen *Relax*, 1

Relax(u, v, w)

```
1 If v.d > u.d + w(u,v)
2    then v.d ← u.d + w(u,v) (* Relaxiere (u,v) *)
3         v.π ← u
```

Laufzeit $O(1)$

Mittels *Relax*(u, v, w) wird v von u aus entdeckt (falls $u.d < \infty$ und $v.d = \infty$),



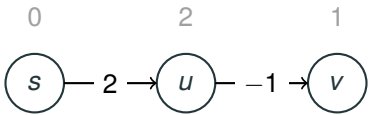
Die Grundoperationen *Relax*, 2

Relax(u, v, w)

```
1 If v.d > u.d + w(u, v)
2    then v.d ← u.d + w(u, v) (* Relaxiere (u, v) *)
3         v.π ← u
```

Laufzeit $O(1)$

Mittels *Relax*(u, v, w) wird v von u aus entdeckt (falls $u.d < \infty$ und $v.d = \infty$),

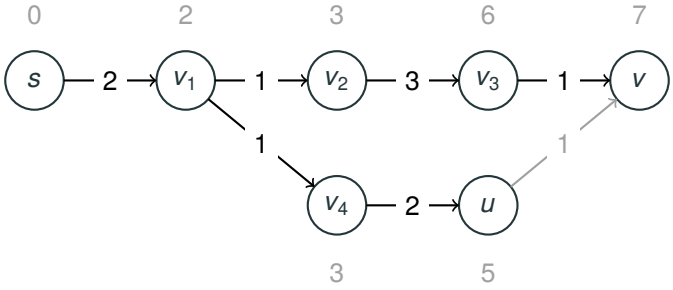


Die Grundoperationen *Relax*, 3

Relax(u, v, w)

```
1 If v.d > u.d + w(u,v)
2    then v.d ← u.d + w(u,v) (* Relaxiere (u,v) *)
3         v.π ← u
```

Mittels *Relax*(u, v, w) wird ein kürzerer Wegs von s nach v (nämlich über u) entdeckt, (falls $\infty > v.d > u.d + w(u, v)$)

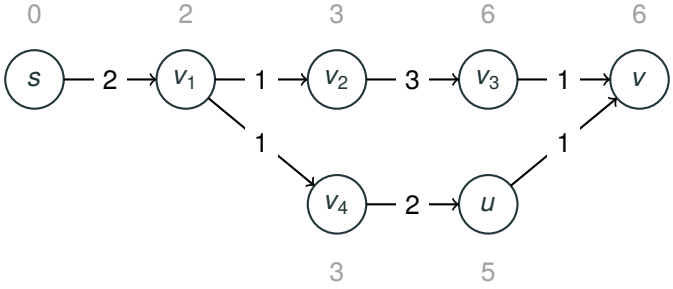


Die Grundoperationen *Relax*, 4

Relax(u, v, w)

1 **If** $v.d > u.d + w(u, v)$
2 **then** $v.d \leftarrow u.d + w(u, v)$ (* Relaxiere (u, v) *)
3 $v.\pi \leftarrow u$

Mittels *Relax*(u, v, w) wird ein kürzerer Wegs von s nach v (nämlich über u) entdeckt, (falls $\infty > v.d > u.d + w(u, v)$)



Eigenschaften von *Initialize* und *Relax*

Die hier vorgestellten Single-Source-Shortest-Path Algorithmen sind **Relax-basiert**, d.h.,:

Auf alle Eingaben $G = (V, E, w)$ und $s \in V$ wird, nach *Initialize*(G, s), lediglich **eine Folge von Relax-Operationen** angewendet.

Das folgende Theorem analysiert das Verhalten des Teilgraphen $G_\pi = (E_\pi, V_\pi)$, mit $V_\pi = \{v \in V; v.d \neq \infty\}$ und $E_\pi = \{(v.\pi, v); v \in V_\pi \setminus \{s\}\}$ für Relax-basierte Algorithmen.

Theorem 67

In G sei von s aus ***kein Kreis mit negativem Gewicht*** *erreichbar.*

Wir wenden Initialize(G, s) *und eine Folge* $R_1, \cdots, R_m$ *von Relax- Operationen auf G an.*

Dann ist G_π *stets ein* ***Baum mit Wurzel*** *s.*

Außerdem gilt für alle $v \in V_\pi$*, dass* $v.d = \delta_{G_\pi}(s, v) \geq \delta_G(s, v)$*.*

Der Beweis von Theorem 67

Wir nehmen an, dass $E_\pi \neq \emptyset$.

Erinnerung: Ein gerichteter Graph ist genau dann ein Baum mit Wurzel s, wenn **jeder Knoten auf genau einem** Weg von s aus erreichbar ist.

Wir wissen, dass jeder Knoten in G_π einen Ingrad von höchstens Eins hat.

Daraus folgt, dass G_π eine **disjunkte Vereinigung von gerichteten Bäumen und einfachen gerichteten Kreisen** ist.

Die Wurzeln der beteiligten Bäume sind dabei genau die Knoten vom Ingrad 0.

Da es in G_π genau einen Knoten mit Ingrad 0 gibt (nämlich s), genügt es zu zeigen, dass G_π azylisch ist.

Denn dann besteht G_π aus genau einem Baum mit Wurzel s.

Die Struktur von G_π

Wir zeigen, dass G_π keine Kreise hat, indem wir folgendes zeigen:

Jeder Kreis $K = (v_1, \cdots, v_k, v_1)$ in G_π muss negatives Gewicht haben.

OBdA sei (v_k, v_1) die Kante in K, die zuletzt relaxiert wird.

Vor $\mathit{Relax}(v_k, v_1, w)$ sind $(v_1, v_2), \cdots, (v_{k-1}, v_k)$ bereits relaxiert, also

$$v_k.d = v_{k-1}.d + w(v_{k-1}, v_k) = v_{k-2} + w(v_{k-2}, v_{k-1}) + w(v_{k-1}, v_k)$$

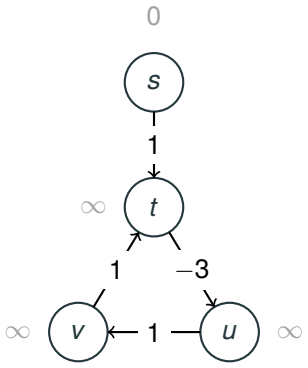
$$= \cdots = v_1.d + \sum_{i=2}^{k} w(v_{i-1}, v_i).$$

Außerdem gilt $v_1.d > v_k.d + w(v_k, v_1)$, d.h.

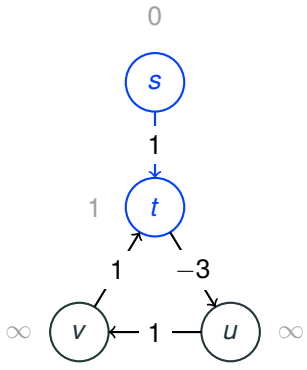
$$v_1.d > v_1.d + \sum_{i=2}^{k} w(v_{i-1}, v_i) + w(v_k, v_1) = v_1.d + w(K).$$

Also $w(K) < 0$, Widerspruch, da K von s aus erreichbar ist. □

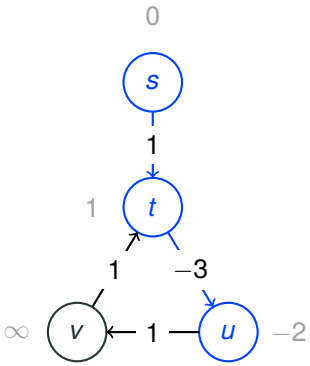
Beispiel G_π mit negativem Kreis



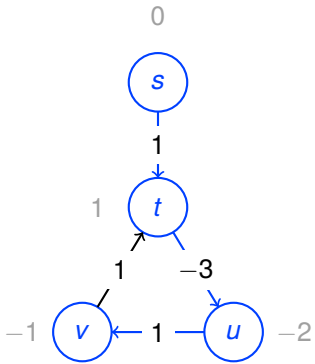
$Relax(s, t)$



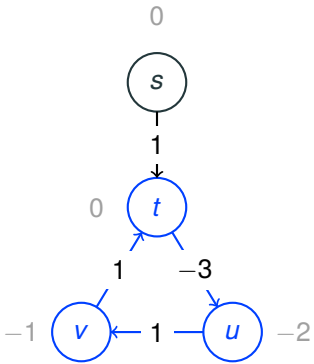
$Relax(s, t)$, $Relax(t, u)$



$Relax(s,t)$, $Relax(t,u)$, $Relax(u,v)$



$Relax(s,t)$, $Relax(t,u)$, $Relax(u,v)$, $Relax(v,t)$



Kürzeste Wege in gewichteten gerichteten Graphen, Single Source Shortest Path

Der Bellman-Ford Algorithmus

Der Bellman-Ford Algorithmus

Idee: Relaxieren $|V| - 1$ Mal alle Kanten im Eingabegraphen $G = (V, E, w)$.

In Runde $|V|$ wird getestet, ob es noch Kanten $(u, v) \in E$ mit $v.d > u.d + w(u, v)$ gibt.

Falls ja, ist von s aus ein Kreis mit negativem Gewicht erreichbar.

BellmanFord(G, w, s)

```
1  Initialize(G, s)
2  For i ← 1 to |G.V| − 1
3     do for all (u, v) ∈ E
4        do Relax(u, v, w)
5  For all (u, v) ∈ E
6     do if v.d > u.d + w(u, v)
7        then return false, STOP
7  return true
```

Die **Laufzeit** beträgt offensichtlich $O(|V||E|)$.

Die Korrektheit des Bellman-Ford Algorithmus

Theorem 68

BellmanFord(G, w, s) *gibt genau dann* ***true*** *aus, wenn von* s *aus kein Kreis mit negativem Gewicht in* G *erreichbar ist.*

In diesem Fall ist G_π *ein Kürzester-Wege-Baum mit Wurzel* s*.*

Beweis

Fall 1: Von s aus ist kein Kreis mit negativem Gewicht in G erreichbar.

Laut Theorem 67 ist dann G_π ein Baum mit Wurzel s.

Wir fixieren einen beliebigen von s aus in G erreichbaren Knoten v und zeigen, dass $v.d = \delta_G(s, v)$.

Es sei $p = (e_1, \cdots, e_k) \subseteq E$ ein kreisfreier kürzester Weg von s nach v in G.

Hierbei gilt $k \leq |V| - 1$, und es sei $e_i = (v_{i-1}, v_i)$ für $i = 1, \cdots, k$, wobei $v_0 = s$ uns $v_k = v$.

Korrektheit des Bellman-Ford Algorithmus, II

Wir zeigen per Induktion über r, $0 \leq r \leq k$, dass nach Runde r der **For**-Schleife in Zeilen 2-4 gilt, dass $v_r.d = \delta_G(s, v_r.d)$.

Iteration 0 entspreche hier der Initialisierung.

Die Induktionsbehauptung gilt offensichtlich für $r = 0$, da $v_0.d = s.d = \delta_G(s, s) = 0$.

Es sei nun $r > 0$.

Vor Iteration r gilt laut Induktionsvoraussetzung, dass $v_{r-1}.d = \delta_G(s, v_{r-1})$.

Damit gilt $\delta_G(s, v_r) = v_{r-1}.d + w(v_{r-1}, v_r)$.

Also wird spätestens durch die Operation $\mathit{Relax}(v_{r-1}, v_r, w)$ in Iteration r der Wert $v_r.d$ auf den kleinstmöglichen Wert $\delta_G(s, v_r)$ gesetzt.

Korrektheit des Bellman-Ford Algorithmus, III

Damit gilt nach Runde $|V| - 1$, dass für alle von s aus erreichbaren Knoten v gilt, dass $v.d = \delta_G(s, v)$.

G_π ist also ein Kürzester-Wege-Baum in G mit Wurzel s.

Für alle $e = (u, v) \in E$ muss stets $\delta_G(s, v) \leq \delta_G(s, u) + w(u, v)$ gelten.

Anderenfalls gäbe es einen Weg von s nach v, der kürzer als $\delta_G(s, v)$ ist, nämlich den von s über u zu v, Widerspruch zur Definition von $\delta_G(s, v)$.

Damit gilt nach Runde $|V| - 1$ für alle $e = (u, v) \in E$, dass $v.d \leq u.d + w(u, v)$.

D.h., der Algorithmus gibt **true** aus.

Fall 2: Von s aus sei ein Kreis $K = (v_1, \cdots, v_k, v_1)$ mit $w(K) < 0$ erreichbar.

Korrektheit des Bellman-Ford Algorithmus, IV

Da alle Knoten in K von s aus mittels einfacher Wege mit höchstens $|V| - 1$ Kanten erreichbar sind, gilt nach den ersten $|V| - 1$ Runden, dass $v_i.d < \infty$ für alle $i = 1, \cdots, k$.

Annahme: Die Ausgabe ist **true**.

Dann gilt

$$v_1.d \leq v_k.d + w(v_k, v_1)$$

sowie

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

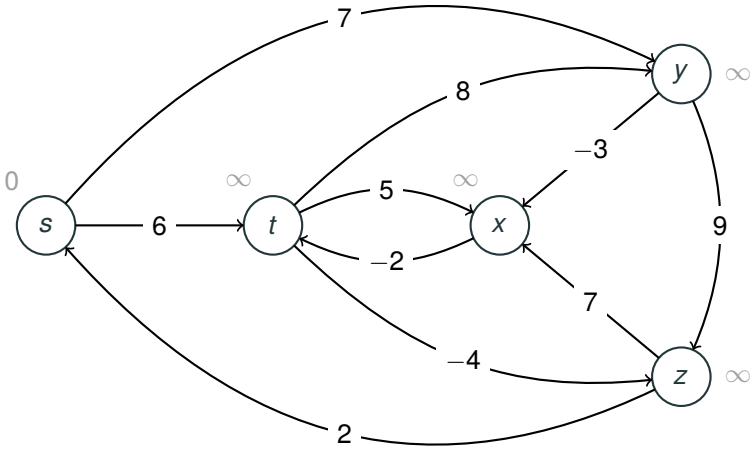
für alle $i = 2, \cdots, k$.

Summiert man diese k Ungleichungen, so erhält man

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_i.d + w(K),$$

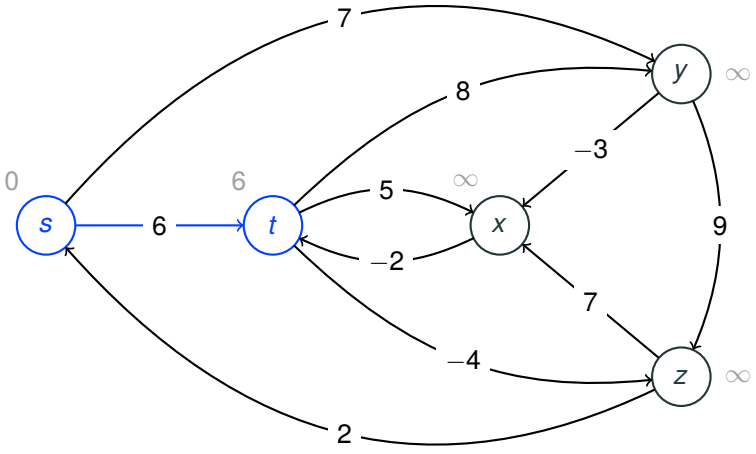
also $w(K) \geq 0$, Widerspruch! $\square$

Beispiel *BellmanFord*(G, s)



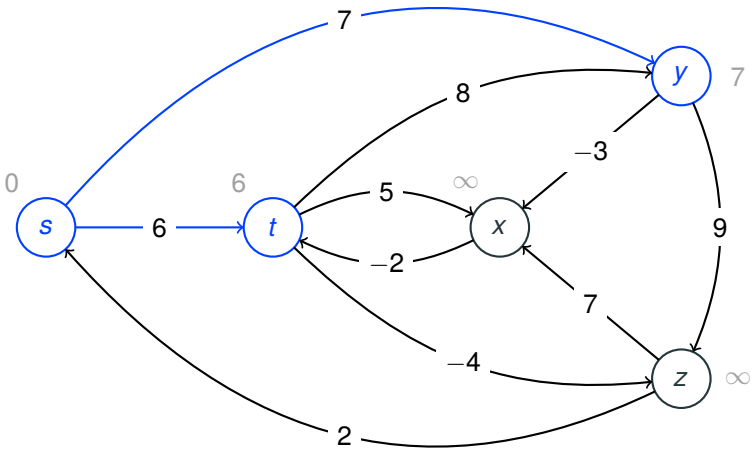
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 (s, t)



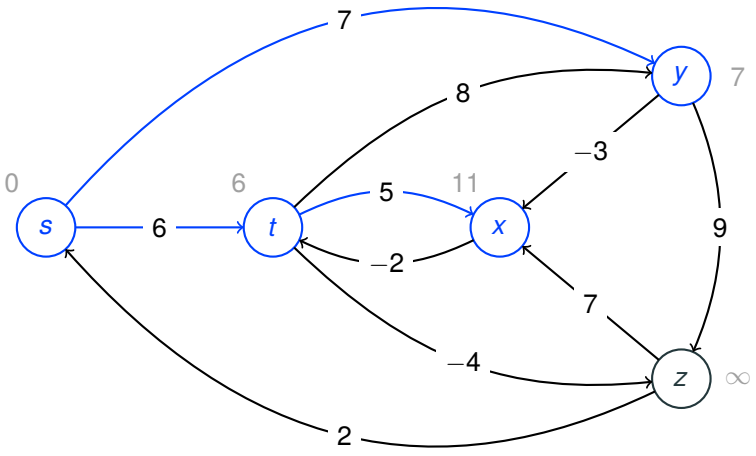
Kanten $(s,t), (s,y), (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,s), (z,x)$

Runde 1 $(s,t),(s,y)$



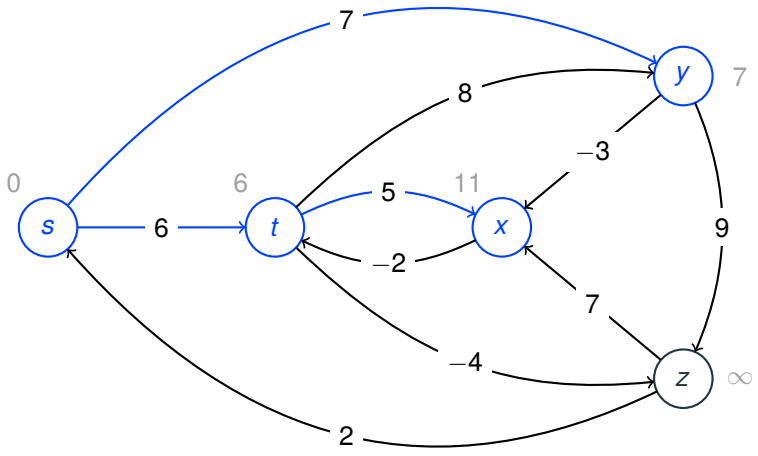
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x)$



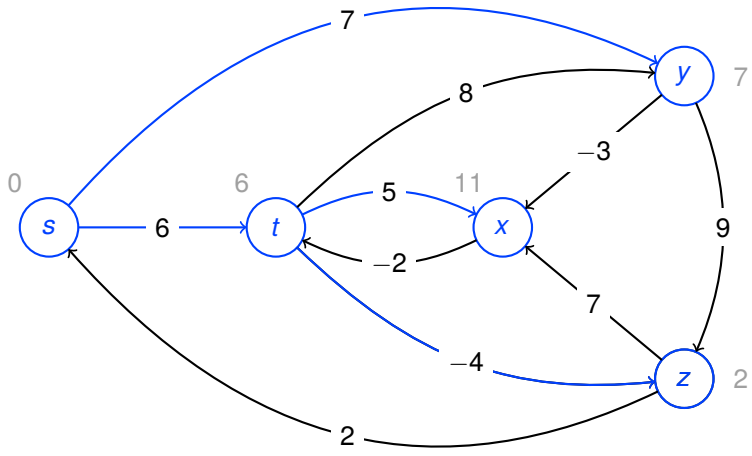
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y)$



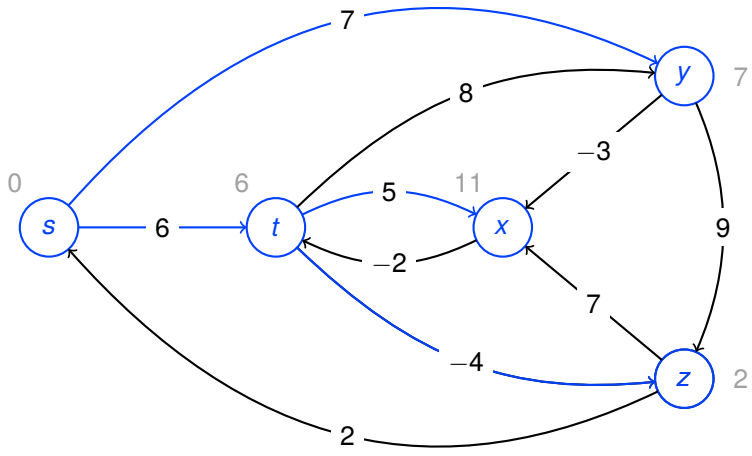
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y),(t,z)$



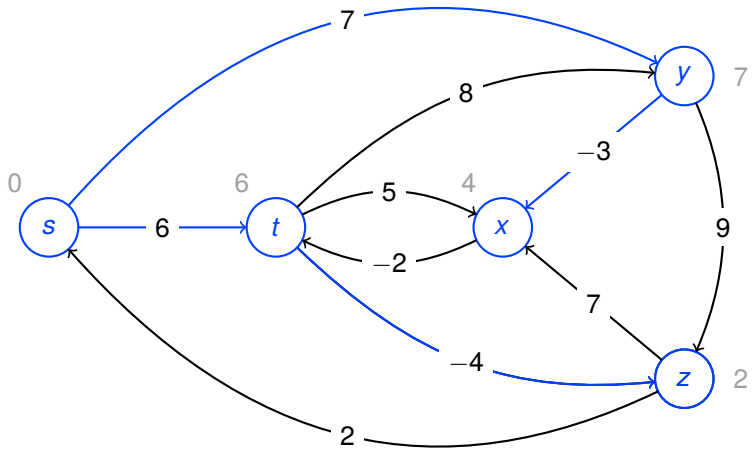
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t)$



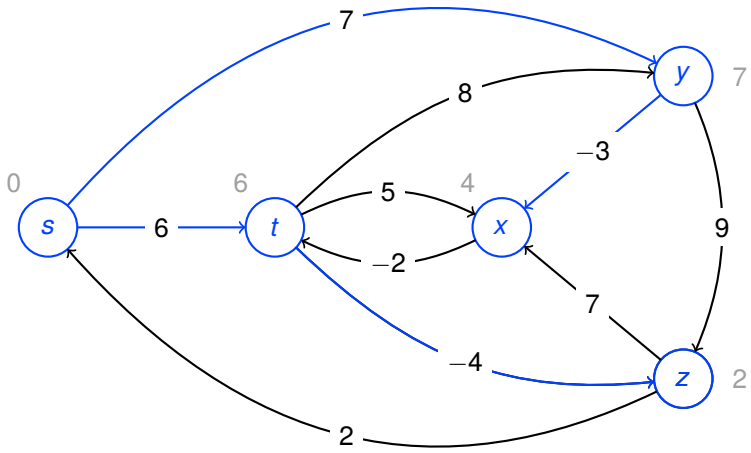
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x)$



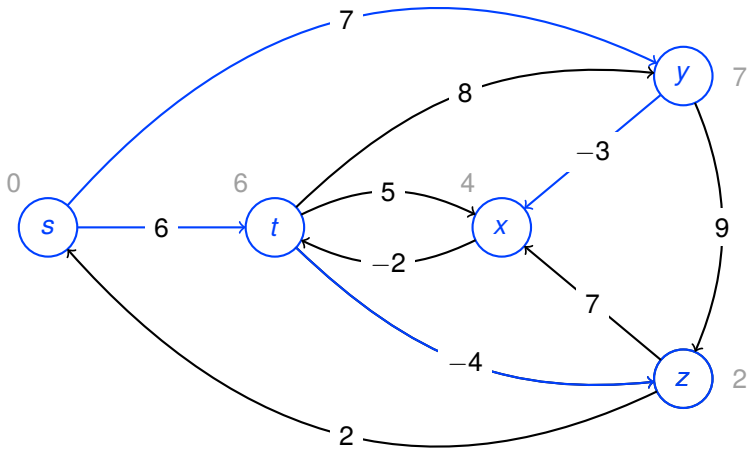
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z)$



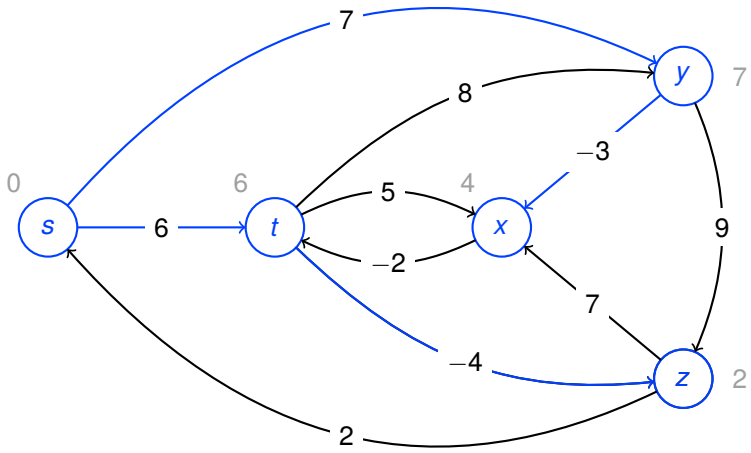
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s)$



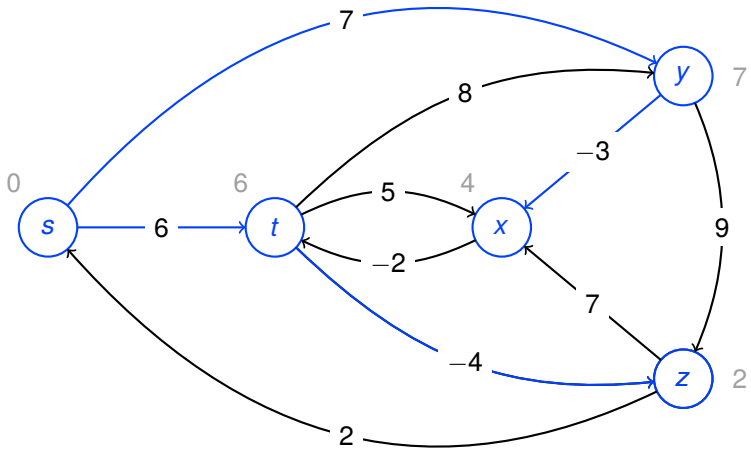
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 1 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$



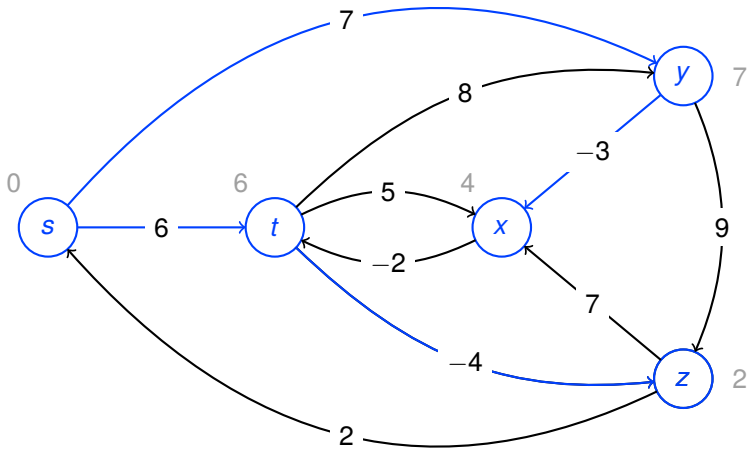
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 (s, t)



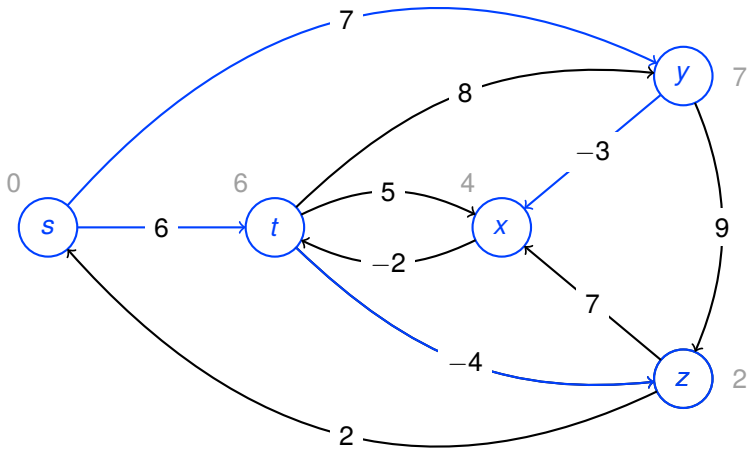
Kanten $(s,t), (s,y), (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,s), (z,x)$

Runde 2 $(s,t),(s,y)$



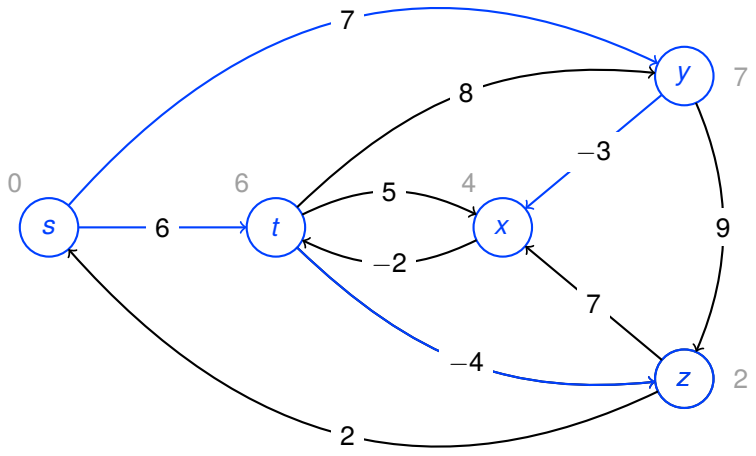
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x)$



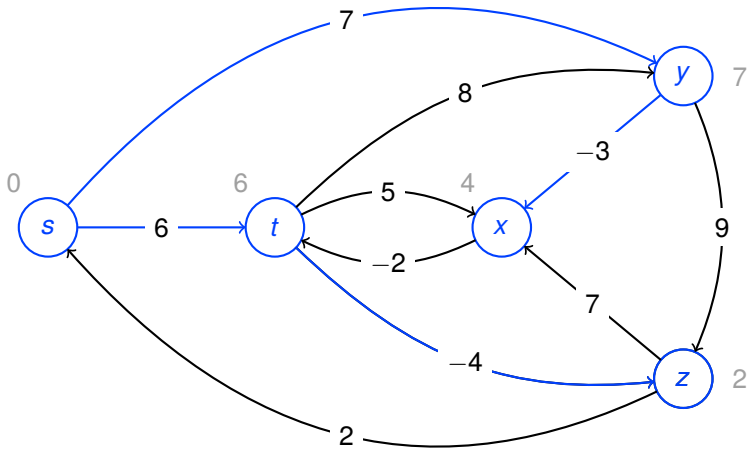
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y)$



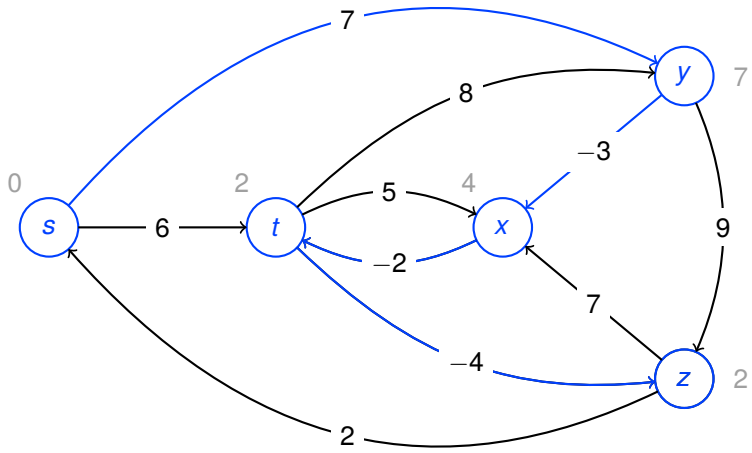
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y),(t,z)$



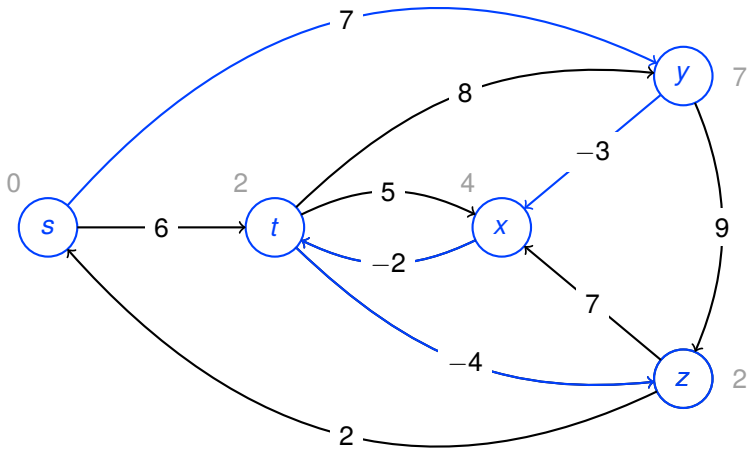
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t)$



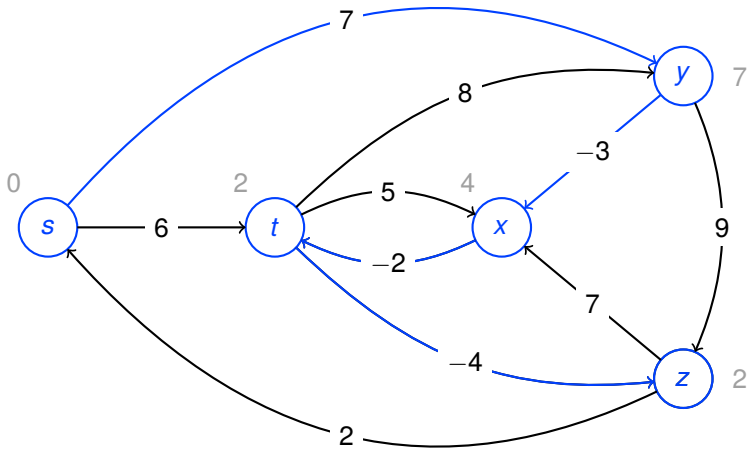
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x)$



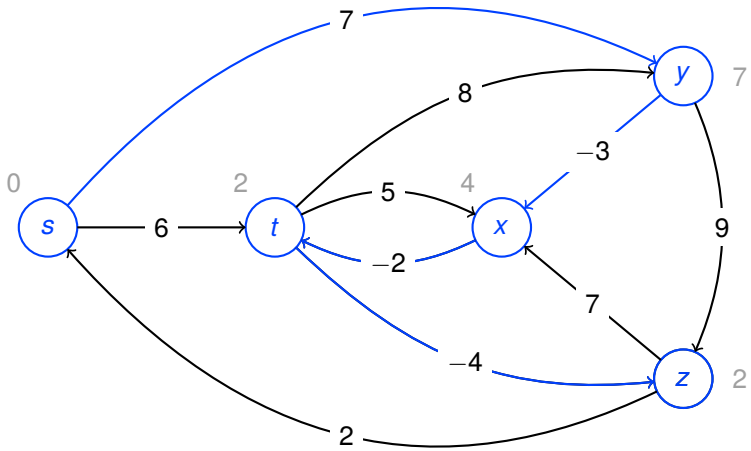
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z)$



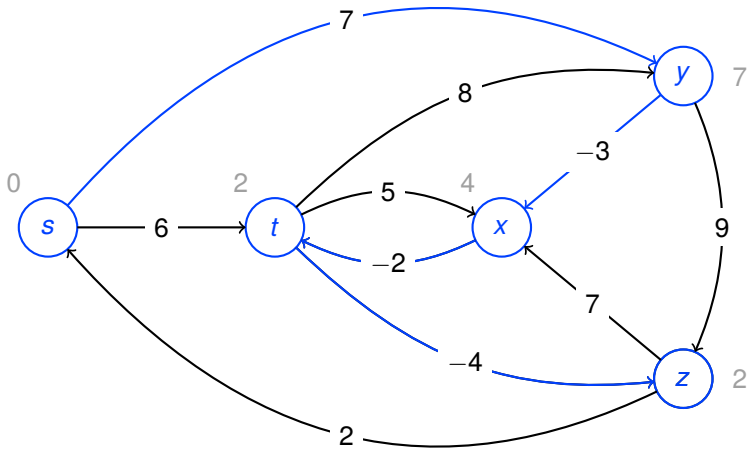
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s)$



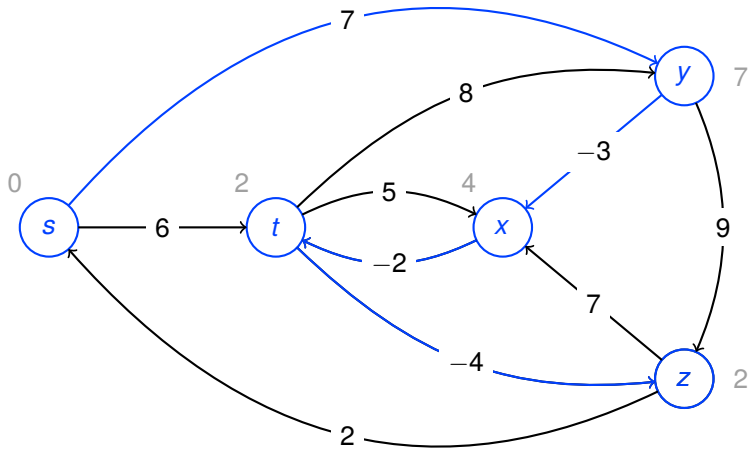
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 2 $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$



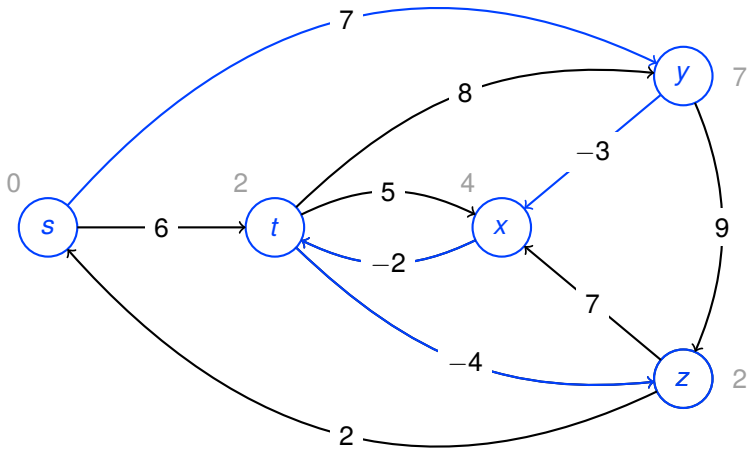
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 3 (s, t)



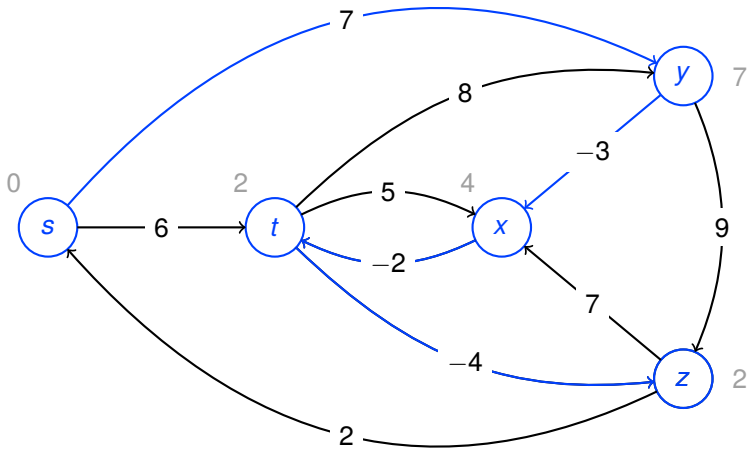
Kanten $(s,t), (s,y), (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,s), (z,x)$

Runde 3 $(s,t),(s,y)$



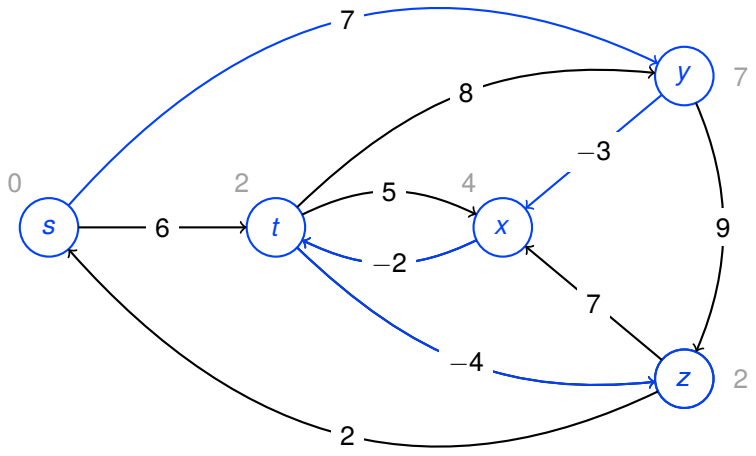
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 3 $(s,t),(s,y),(t,x)$



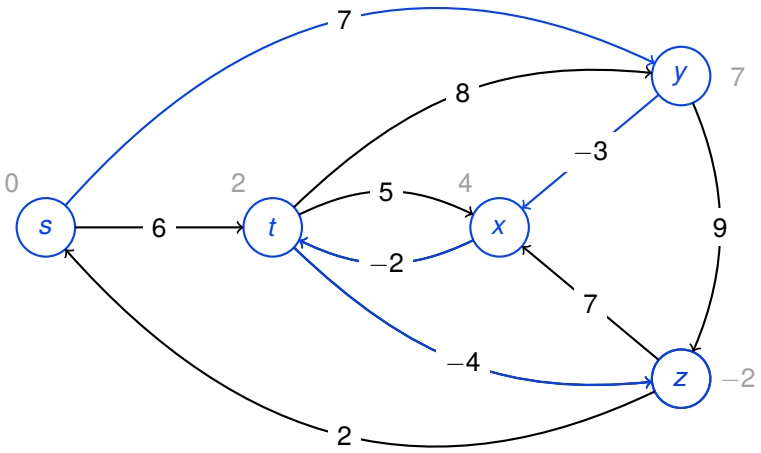
Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 3 $(s,t),(s,y),(t,x),(t,y)$



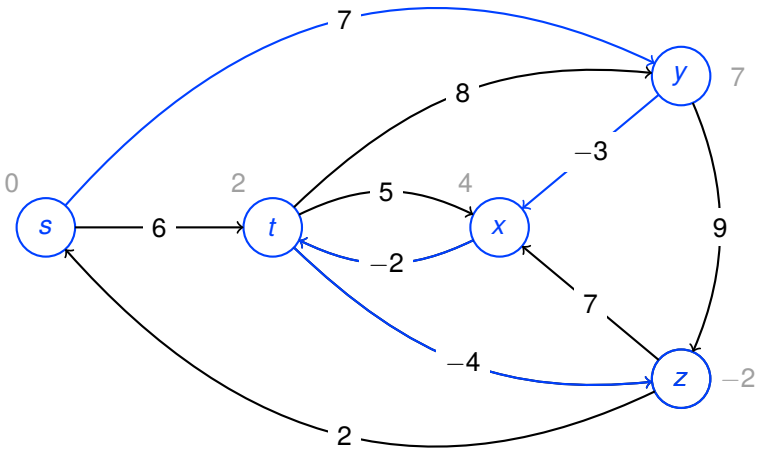
Kanten $(s,t),(s,y),(t,x)(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Runde 3 $(s,t),(s,y),(t,x),(t,y),(t,z)$



Kanten $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,s),(z,x)$

Ausgabe true



Die restlichen 5 Relax in Runde 3 und die in Runde 4 bringen keine Veränderung mehr.

Kürzeste Wege in gewichteten gerichteten Graphen, Single Source Shortest Path

Der Algorithmus von Dijkstra

Der Algorithmus von Dijkstra, die Idee

- Der schnellere Algorithmus $\mathit{Dijkstra}(G, w, s)$ läuft nur auf Graphen mit nichtnegativen Kantengewichten.
- $\mathit{Dijkstra}(G, w, s)$ läuft in $|V|$ Runden.
- $\mathit{Dijkstra}(G, w, s)$ verwaltet dynamisch eine Partition von V in Q und $V \setminus Q$, wobei Q als **Min Priority Queue** bezüglich $v.d$ organisiert ist.
- Initial gilt $Q = V$.
- In jeder Runde wird der Knoten u mit dem minimalen $u.d$-Wert aus Q entfernt und alle von u ausgehenden Kanten relaxiert und die Priority Queue entsprechend aktualisiert.
- Damit wird jeder Knoten einmal aus der Queue entfernt und jede Kante höchstens einmal relaxiert. Beides kostet $O(\log(|V|))$.
- Damit ist die **Gesamtlaufzeit** $O((|V| + |E|) \cdot \log(|V|))$.
- Nichttrivial ist der Korrektheitsbeweis.

Der Algorithmus

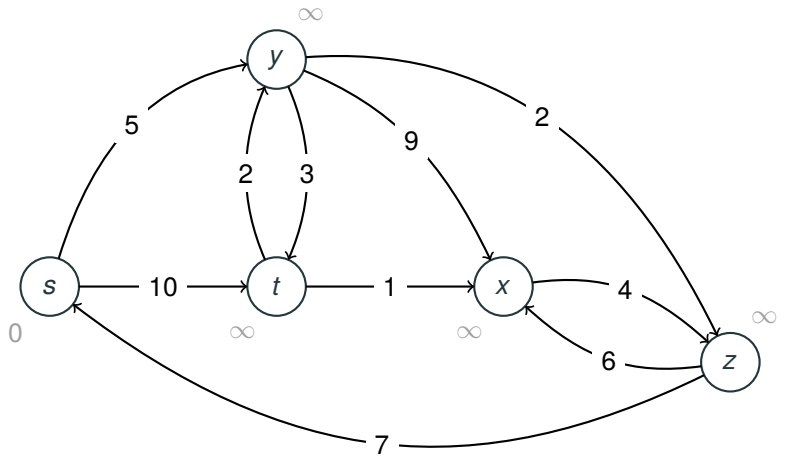
Sei $G = (V, E, w)$ ein gewichteter Graph mit nichtnegativer Kantgengewichtsfunktion $w : E \longrightarrow \mathbb{R}^{\geq 0}$ und $s \in V$.

Dijkstra(G, w, s)

```
1  Initialize(G, s)
2  Q ← BuildHeap(V) (*Min.Priority Queue bzgl. v.d *)
3  while heapsize(Q) > 0
4      do u ← MinExtract(Q) (* Bearbeite u *)
5          For all v ∈ G.Adj[u]
6              do Relax(u, v, w), DecreaseKey(Q, v, v.d)
```

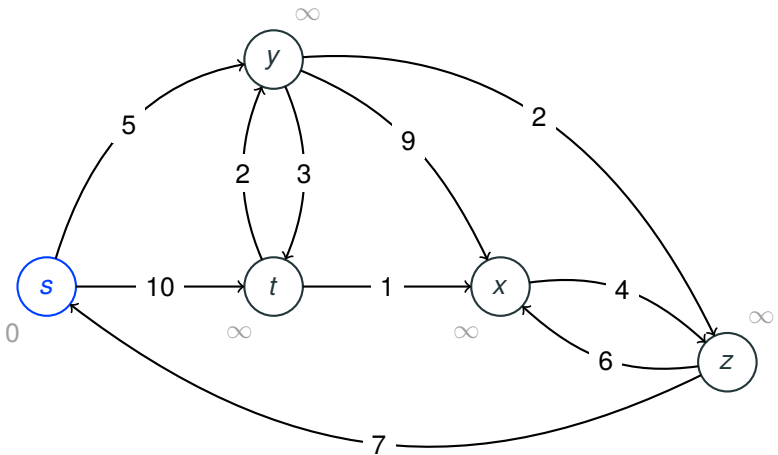
Laufzeit: $O((|E| + |V|) \cdot \log |V|)$.

Beispiel *Dijkstra*(G, w, s)



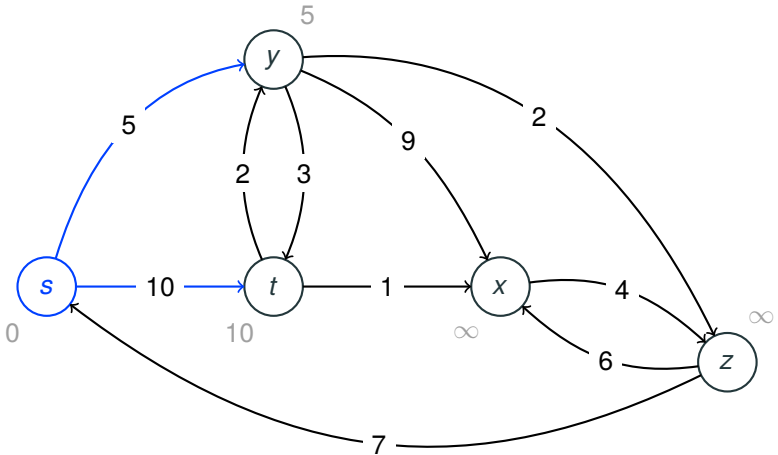
$Q = V$, $V \setminus Q = \emptyset$

Runde 1, *ExtractMin*(Q)



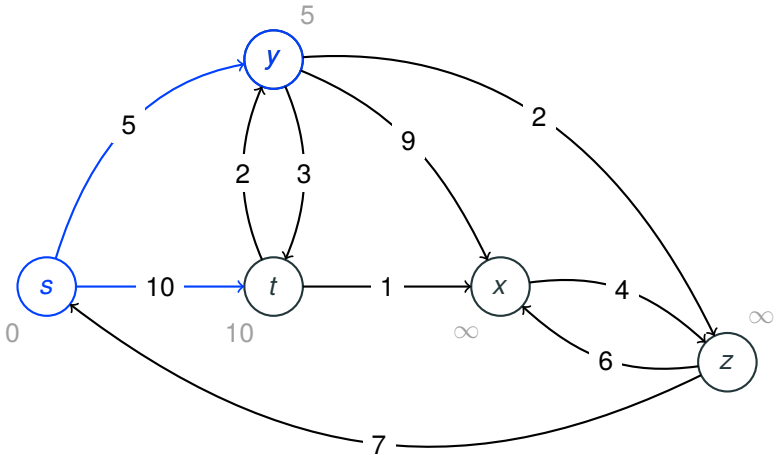
$Q = \{t, x, y, z\}$, $V \setminus Q = \{s\}$

Runde 1, *Relax*



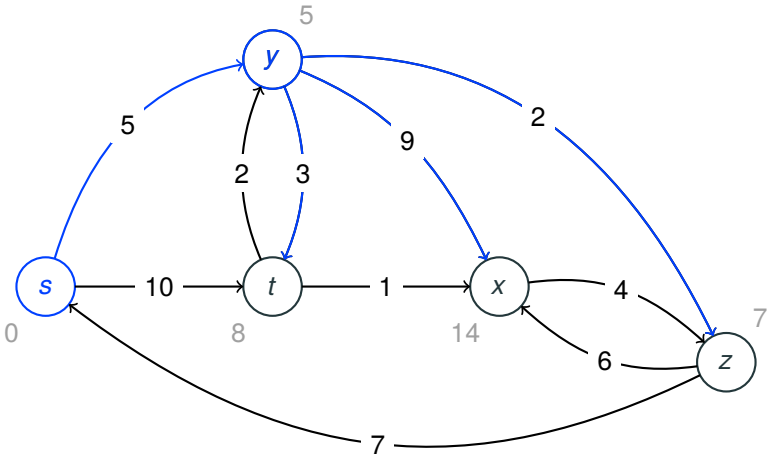
$Q = \{t, x, y, z\}$, $V \setminus Q = \{s\}$

Runde 2, *ExtractMin*



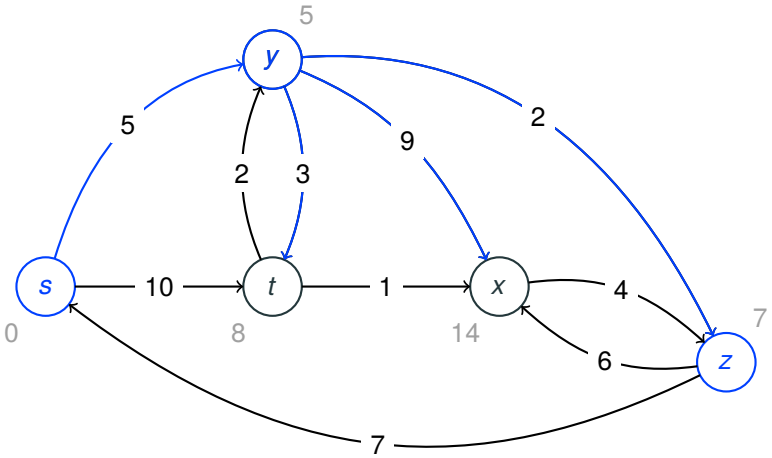
$Q = \{t, x, z\}$, $V \setminus Q = \{s, y\}$

Runde 2, *Relax*



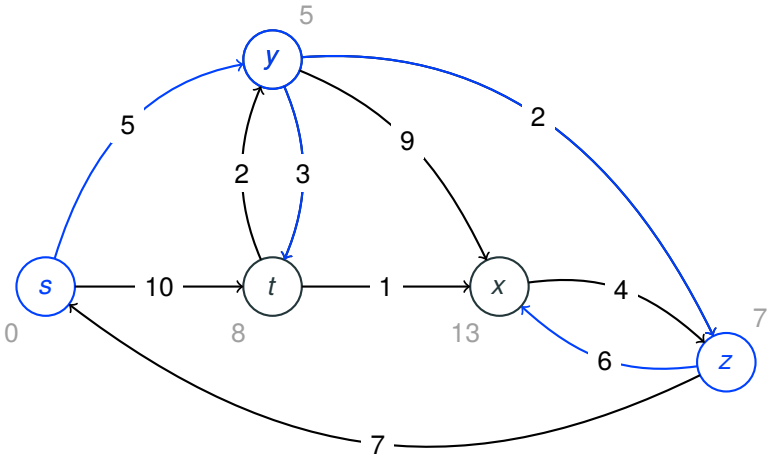
$Q = \{t, x, z\}$, $V \setminus Q = \{s, y\}$

Runde 3, *ExtractMin*



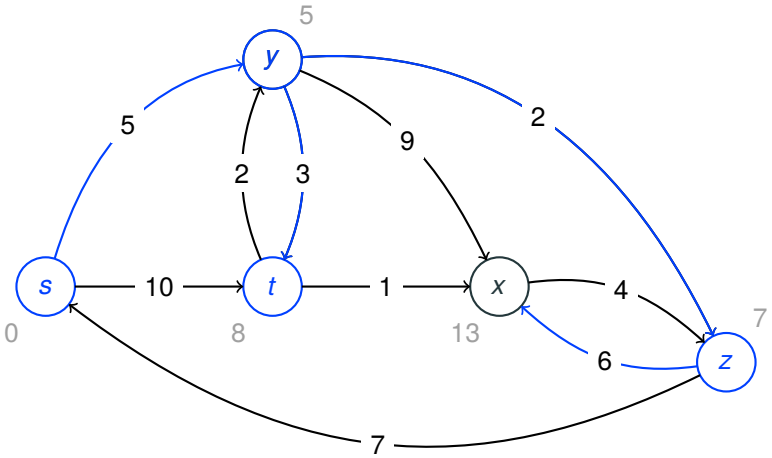
$Q = \{t, x\}$, $V \setminus Q = \{s, y, z\}$

Runde 3, *Relax*



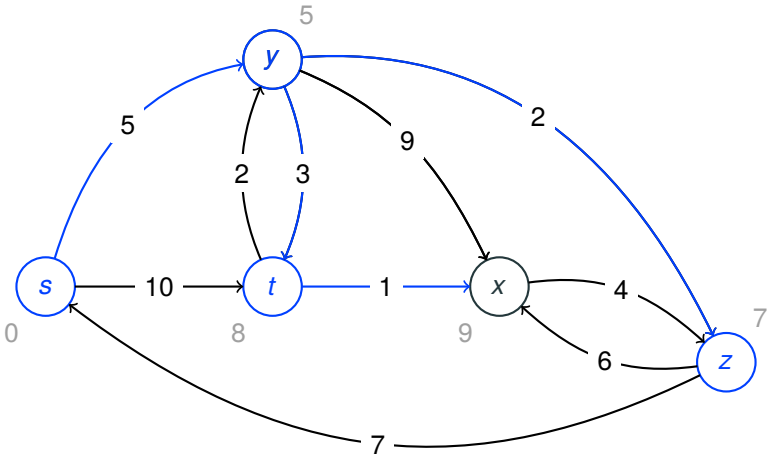
$Q = \{t, x\}$, $V \setminus Q = \{s, y, z\}$

Runde 4, *ExtractMin*



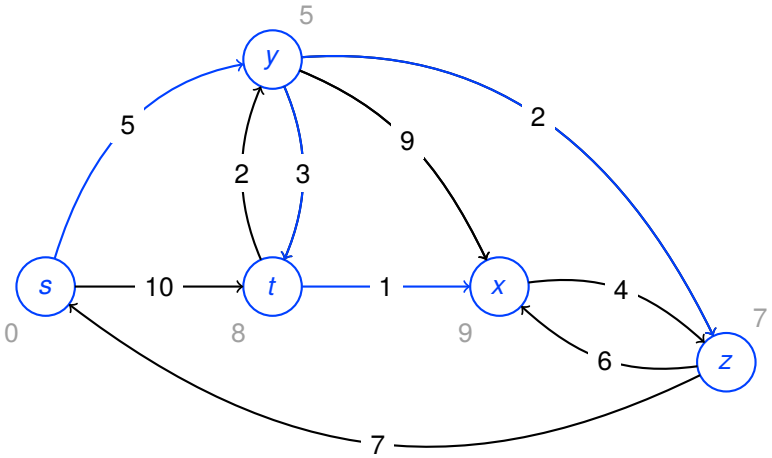
$Q = \{x\}$, $V \setminus Q = \{s, y, z, t\}$

Runde 4, *Relax*



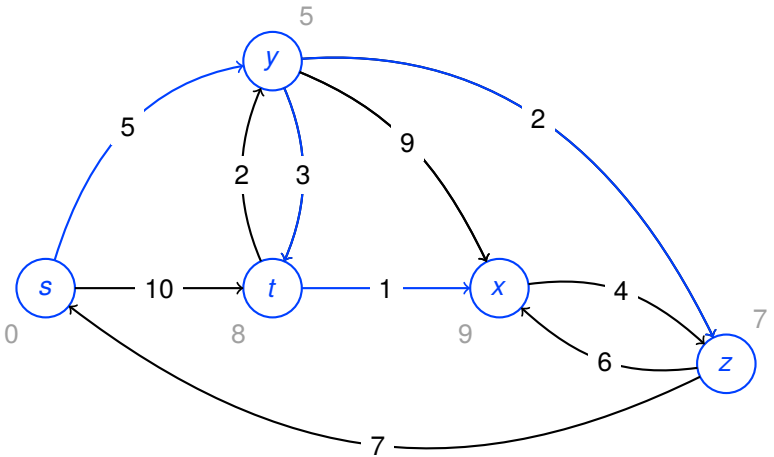
$Q = \{x\}$, $V \setminus Q = \{s, y, z, t\}$

Runde 5, *ExtractMin*



$Q = \emptyset$, $V \setminus Q = \{s, y, z, t, x\}$

Runde 5, *Relax*



Die Korrektheit des Algorithmus von Dijkstra, I

Theorem 69

Der von Dijkstra(G, w, s) ausgegebene Graph G_π ist ein Kürzeste-Wege-Baum mit Wurzel s.

Beweis: Da $w(e) \geq 0$ für alle $e \in E$ ist $G_\pi = (V_\pi, E_\pi)$ ein Baum mit Wurzel s.

Annahme: Es gibt Knoten v, für die $v.d > \delta_G(s, v)$ gilt.

Unter all diesen Knoten bezeichne v^* den Knoten, der zuerst bearbeitet wird.

Wir betrachten die Situation direkt vor dem Durchlauf der **while** Schleife, in dem v^* bearbeitet wird.

In dieser Situation gilt $v^* \in Q$ und $v^*.d = \min\{v.d, v \in Q\}$.

Da $\delta_G(s, v^*) < \infty$ existiert ein kürzester Weg p von s zu v^*.

Wir bezeichnen mit u^* den letzten Knoten entlang p, der nicht in Q liegt, und mit $\tilde{v} \in Q$ seinen Nachfolger entlang p.

Korrektheit des Algorithmus von Dijkstra, II

Da $u^* \notin Q$ gilt $u^*.d = \delta_G(s, u^*)$, da u^* vor v^* bearbeitet wurde.

Fall 1: $\tilde{v} = v^*$.

Dann wird (u^*, v^*) während der Bearbeitung von u^* relaxiert, und es gilt danach

$$\delta_G(s, v^*) = \delta_G(s, u^*) + w(u^*, v^*) = u^*.d + w(u^*, v^*) = v^*.d.$$

Fall 2: $\tilde{v} \neq v^*$.

Dann gilt mit dem gleichen Argument wie in Fall 1, dass $\tilde{v}.d = \delta_G(s, \tilde{v})$.

Da alle Kanten von G nichtnegatives Gewicht haben, gilt zudem

$$\tilde{v}.d = \delta_G(s, \tilde{v}) \leq \delta_G(s, v^*) < v^*.d.$$

Folglich hätte $\tilde{v}$ statt v^* als nächster zur Bearbeitung anstehender Knoten gewählt werden müssen, Widerspruch zur Wahl von v^*. □

Kürzeste Wege in gewichteten gerichteten Graphen, All Pairs Shortest Paths

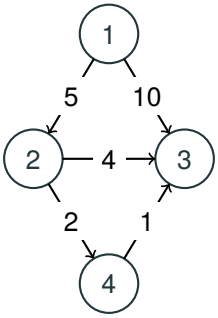
Kürzeste Wege in gewichteten gerichteten Graphen, All Pairs Shortest Paths

Einführung in das Problem

Das All Pairs Shortest Path Problem

- **Eingabe:** Gerichteter gewichteter Graph $G = (V, E, w)$ mit Gewichtsfunktion $w : E \longrightarrow \mathbb{R}$, wobei in diesem Abschnitt $V = \{1, \cdots, n\}$. 55
- Wir setzen voraus, dass G **keine Kreise mit negativem Gewicht** enthält.
- **Eingabe-Datenstruktur:** Gewichtsmatrix $W = (w(i,j))_{i,j=1}^{n}$, wobei $w(i,j) = \infty$, falls $(i,j) \notin E$.
- **Ausgabe 1:** Entfernungsmatrix $\Delta_G = (\delta_G(i,j))_{i,j=1}^{n}$.
- **Ausgabe 2:** Zeigermatrix $\Pi_G = (\pi_{i,j})_{i,j=1}^{n}$.
- Hierbeit ist für alle i, j, $1 \leq i, j \leq n$, der Knoten $\pi_{i,j}$ der Vorgänger von j entlang eines kürzesten Weges von i nach j.
- Falls j von i aus nicht erreichbar ist, sei $\pi_{i,j} = \mathit{NIL}$.
- Ferner gelte $\pi_{i,i} = i$.

Beispielgraph mit W und Π



$$W = \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}, \quad \Pi = \begin{pmatrix} 1 & 1 & 1 & N \\ N & 2 & 2 & 2 \\ N & N & 3 & N \\ N & N & 4 & 4 \end{pmatrix}.$$

Ausgabe kürzester Wege

Mittels der folgenden Prozedur können kürzeste Wege effizient ausgegeben werden.

PrintPath(Π, i, j)

```
1  If π_{i,j} ≠ NIL
2    then if i ≠ j
3           then PrintPath(Π, i, π_{i,j}), Print(j)
4           else Print(i)
2    else Print(j not reachable from i)
```

Kürzeste Wege in gewichteten gerichteten Graphen, All Pairs Shortest Paths

Ansatz Dynamische Programmierung

Grundlagen

- Für alle $k \geq 0$ und $i \neq j \in V$ sei $d_{i,j}^k$ die **Länge eines kürzesten Weges von i nach j, der höchstens k Kanten enthält**.
- **Beobachtung 1:** $d_{i,i}^k = 0$ für alle $i \in V$ und $k \geq 0$.
- Es sei $D^{(k)} = (d_{i,j}^k)_{i,j=1}^n$.
- **Beobachtung 2:** Es gilt $d_{i,j}^0 = \infty$ für alle $i \neq j \in V$.
- **Beobachtung 3:** Für alle $i \neq j \in V$ gilt $d_{i,j}^1 = w(i,j)$, d.h.,

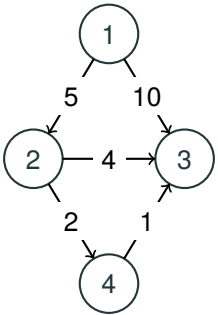
$$D^{(1)} = W.$$

- **Beobachtung 4:** Für alle $i \neq j \in V$ gilt $d_{i,j}^{n-1} = \delta_G(i,j)$, d.h.,

$$D^{(n-1)} = \Delta_G.$$

- **Bleibt Frage:** Wie berechnet man D^k effizient aus D^{k-1}, $2 \leq k \leq n-1$?

Beispiel mit $D^{(0)}$, $D^{(1)}$, $D^{(2)}$, $D^{(3)}$



$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}, \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 5 & 9 & 7 \\ \infty & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 5 & 8 & 7 \\ \infty & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}.$$

Berechnung von $D^{(k)}$

Lemma 70

Für alle $i \neq j \in V$ und $k \geq 2$ gilt $d_{i,j}^k = \min\{d_{i,r}^{k-1} + w_{r,j}; r = 1, \cdots, n\}$.

Beweis

Fall 1: $d_{i,j}^k < d_{i,j}^{k-1}$, d.h., es existiert ein kürzester Weg p von i nach j mit k Kanten, der kürzer ist als alle Wege von i nach j mit weniger als k Kanten.

Per Definition gilt $d_{i,j}^k = w(p)$.

Es sei r^* der Vorgänger von j entlang p.

Da $k \geq 2$ gilt $r^* \notin \{i, j\}$.

Sowohl der Teilweg von p von i bis r^*, als auch die Kante (r^*, j) sind kürzeste Wege.

Also gilt **für alle** $r = 1, \cdots, n$**,** $r \notin \{i, j\}$, dass

$$d_{i,j}^k = d_{i,r^*}^{k-1} + w_{r^*,j} \leq d_{i,r}^{k-1} + w_{r,j}.$$

Berechnung von $D^{(k)}$, Fortsetzung

Für $r = i$ **gilt** $d_{i,j}^k < d_{i,i}^{k-1} + w(i,j) = 0 + d_{i,j}^1$, da $d_{i,j}^1 \leq d_{i,j}^{k-1}$.

Für $r = j$ **gilt** $d_{i,j}^k < d_{i,j}^{k-1} + w(j,j) = d_{i,j}^{k-1} + 0$.

Insgesamt gilt also $d_{i,j}^k = d_{i,r^*}^{k-1} + w_{r^*,j} \leq d_{i,r}^{k-1} + w(r,j)$ für alle r, $1 \leq r \leq n$.

Fall 2: $d_{i,j}^k = d_{i,j}^{k-1}$, d.h. es gibt einen kürzesten Weg p von i nach j mit weniger als k Kanten.

Dann ist $d_{i,j}^k = d_{i,j}^{k-1} = w(p) = d_{i,j}^{k-1} + w(j,j) = d_{i,j}^{k-1} + 0$.

Für alle r, $1 \leq r \leq n$, gilt zudem $d_{i,j}^k \leq d_{i,r}^{k-1} + w(r,j)$, da für kein $r \neq j$ ein Weg q mit k Kanten und letztem Knoten r vor j existiert mit $w(q) < w(p) = d_{i,j}^{k-1}$.

Also gilt auch hier $d_{i,j}^k = \min\{d_{i,r}^{k-1} + w_{r,j}; r = 1, \cdots, n\}$. $\square$

SlowAllPairsShortestPaths(W, n)

Für $n \times n$-Matrizen A, B mit Einträgen in $\mathbb{R} \cup \{\infty\}$ definieren wir eine $n \times n$-Matrix $A \circ B$

$$(A \circ B)_{i,j} = \min\{a_{i,r} + b_{r,j}; 1 \leq r \leq n\}.$$

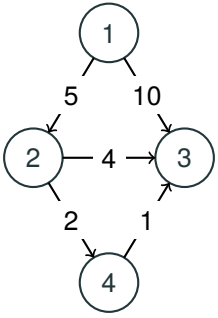
Das ergibt einen Algorithmus der Laufzeit $O(n^4)$ für All-Pairs-Shortest-Paths.

SlowAllPairsShortestPaths(W, n)

```
1 D ← W
2 For i ← 2 to n − 1
3     do D ← D ∘ W
4 Output D
```

Eingabe ist die Gewichtsmatrix W und $n = |V|$, **Ausgabe** die Matrix Δ_G.

Nochmal das Beispiel mit W



$$W = \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}.$$

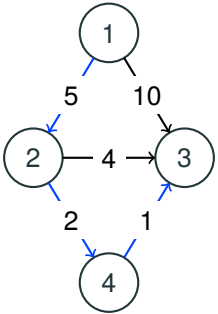
$$D^{(2)} = W \circ W = \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 5 & 9 & 7 \\ \infty & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}.$$

$\min\{0+10, 5+4, 10+0, \infty+1\} = \min\{10, 9, \infty\} = 9$

$$D^{(3)} = D^{(2)} \circ W = \begin{pmatrix} 0 & 5 & 9 & 7 \\ \infty & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 5 & 8 & 7 \\ 1 & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}.$$

$\min\{0+10, 5+4, 9+0, 7+1\} = \min\{10, 9, 8\} = 8$

Beispiel mit Δ



$$\Delta = \begin{pmatrix} 0 & 5 & 8 & 7 \\ 1 & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}$$

Kürzeste Wege in gewichteten gerichteten Graphen, All Pairs Shortest Paths

Ein schnelleres Verfahren

Algebraische Grundlagen

- Wir fixieren einen Ring $(R, \oplus, \otimes)$.
- D.h. $\oplus$ und $\otimes$ sind assoziative Operationen auf R, es gilt das Distributivgesetz, und es existieren neutrale Elemente 0_R für $\oplus$ und 1_R für $\otimes$.
- Für jeden Ring $(R, \oplus, \otimes)$ ist für $n \geq 1$ der Matrizenring $(M_n(R), \oplus, \circ)$ der $n \times n$-Matrizen mit Koeffizienten aus R definiert.
- Hierbei ist $\oplus$ ist die komponentenweise Addition von Matrizen und $\circ$ die Matrixmultiplikation,

$$(A \circ B)_{i,j} = \bigoplus_{r=1}^{n} A_{i,r} \otimes B_{r,j}.$$

- Neutrale Elemente sind die 0-Matrix bzw. die Einheitsmatrix.
- Die Matrixmultiplikation ist assoziativ, die Berechnung nach der Standardmethode kostet $O(n^3)$.

Ein schnelleres Verfahren für Δ_G

- Wir betrachten den Ring $(\mathbb{R} \cup \{\infty\}, \oplus, \otimes)$ mit
  - **Addition** $x \oplus y = \min\{x, y\}$, (**Neutrales Element** $0_R = \infty$)
  - **Multiplikation** $x \otimes y = x + y$, (**Neutrales Element** $1_R = 0$).
- **Beobachtung:** Die Matrixmultiplikation $\circ$ im Matrixring $M_n(\mathbb{R} \cup \{\infty\})$ entspricht genau der Matrixoperation $A \circ B$, die im Kontext von *SlowAllPairsShortestPath* definiert wurde.
- Wir erhalten den schnelleren Algorithmus *FastAllPairsShortestPath* zur Berechnung von $D^{(n-1)}$ durch Ausnutzung der Assoziativität von $\circ$.
- Die Matrix $D^{(n-1)} = W^{n-1}$ kann durch $O(\log(n))$ Quadrierungen

$$D \leftarrow D \circ D$$

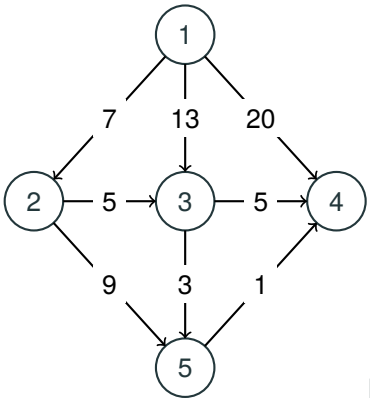
  berechnet werden.
- **Gesamtlaufzeit** $O(n^3 \cdot \log(n))$ im Vergleich zu $O(n^4)$ für *SlowAllPairsShortestPath*.

FastAllPairsShortestPaths(W, n)

```
1 s ← 0
2 D ← W
3 while 2^s < n − 1
4       do D ← D ∘ D
5          s ← s + 1
6 Output D
```

... berechnet $\Delta_G = D^{(2^s)} = D^{(n-1)}$ in Zeit $O(|V|^3 \cdot \log(|V|))$.

Beispiel mit W



$$W = \begin{pmatrix} 0 & 7 & 13 & 20 & \infty \\ \infty & 0 & 5 & \infty & 9 \\ \infty & \infty & 0 & 5 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

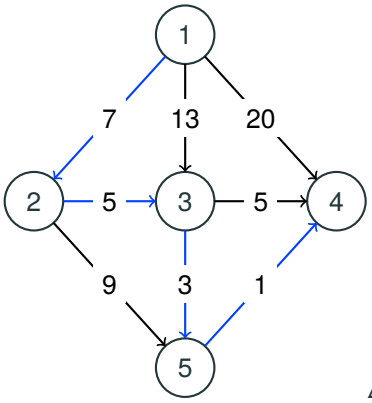
$$D = W \circ W = \begin{pmatrix} 0 & 7 & 13 & 20 & \infty \\ \infty & 0 & 5 & \infty & 9 \\ \infty & \infty & 0 & 5 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 0 & 7 & 13 & 20 & \infty \\ \infty & 0 & 5 & \infty & 9 \\ \infty & \infty & 0 & 5 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 7 & 12 & 18 & 16 \\ \infty & 0 & 5 & 10 & 8 \\ \infty & \infty & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$\min\{\infty + 20, \infty + \infty, 0 + 5, 5 + 0, 3 + 1\} = \min\{\infty, 5, 4\} = 4$$

$$D^{(4)} = D \circ D = \begin{pmatrix} 0 & 7 & 12 & 18 & 16 \\ \infty & 0 & 5 & 10 & 8 \\ \infty & \infty & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 0 & 7 & 12 & 18 & 16 \\ \infty & 0 & 5 & 10 & 8 \\ \infty & \infty & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 7 & 8 & 16 & 15 \\ \infty & 0 & 5 & 9 & 8 \\ \infty & \infty & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$\min\{0 + 18,\ 7 + 10,\ 12 + 4,\ 18 + 0,\ 16 + 1\} = \min\{18, 17, 16\} = 16$$

Beispiel mit Δ



$$\Delta = \begin{pmatrix} 0 & 7 & 8 & 16 & 15 \\ \infty & 0 & 5 & 9 & 8 \\ \infty & \infty & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Kürzeste Wege in gewichteten gerichteten Graphen, All Pairs Shortest Paths

Die Berechnung der Zeigermatrizen

Berechnung von Π_G, Vorbetrachtungen

Definition 71

Die Matrix $\Pi^{(k)} = (\pi_{i,j}^{(k)})_{i,j=1}^{n}$ sei für alle $k \geq 1$ wiefolgt definiert.

Für alle i, $1 \leq i \leq n$, sei $\pi_{i,i}^{(k)} = i$.

Für alle $i \neq j$, $1 \leq i,j \leq n$, sei $\pi_{i,j}^{(k)}$ der Vorgänger von j entlang eines kürzesten Weges von i nach j mit höchstens k Kanten.

Falls kein derartiger Weg existiert, sei $\pi_{i,j}^{(k)} = \mathit{NIL}$.

Aussehen von $\Pi^{(1)}$

- Für $(i,j) \in E$ gilt $\pi_{i,j}^{(1)} = i$,
- es gilt $\pi_{i,i}^{(1)} = i$,
- für $i \neq j$ und $(i,j) \notin E$ gilt $\pi_{i,j}^{(1)} = \mathit{NIL}$.

Es gilt $\Pi_G = \Pi^{(n-1)}$.

Problemstellung und die Operation *Link*

- **Problem:** Effiziente Berechnung von $\Pi^{(s+t)}$ aus $\Pi^{(s)}$ und $\Pi^{(t)}$.
- **Idee:** Besteht ein kürzester Weg mit höchstens $s+t$ Kanten von i nach j aus dem kürzester Weg mit höchstens s Kanten von i nach r^* und und dem kürzester Weg mit höchstens t Kanten von r^* nach j, so ist $\Pi_{i,j}^{(s+t)} = \Pi_{r^*,j}^{(t)}$ falls $r^* \neq j$ und $\Pi_{i,j}^{(s+t)} = \Pi_{i,j}^{(s)}$ falls $r^* = j$.
- Auf dieser Idee beruht die Operation $Link(A, B, \Pi^A, \Pi^B)$.
- $Link(A, B, \Pi^A, \Pi^B)$ ist für $n \times n$-Matrizen Entfernungsmatrizen A, B mit Einträgen in $\mathbb{R} \cup \{\infty\}$ und $n \times n$-Zeigermatrizen Π^A, Π^B mit Einträgen in $\{1, \cdots, n\} \cup \{NIL\}$ definiert.
- $Link(A, B, \Pi^A, \Pi^B)$ berechnet als Ausgabe die Zeigermatrix $\Pi^{A \circ B}$ zur Entfernungsmatrix $A \circ B$.
- *Link* benutzt die Prozedur $SelectMin(x_1, \cdots, x_n)$, die für $(x_1, \cdots, x_n) \in (\mathbb{R} \cup \{\infty\})^n$ einen Index r^* mit $x_{r^*} = \min\{x_1, \cdots, x_n\}$ ausgibt.

Die Operation *Link*

$\underline{Link(A, B, \Pi^A, \Pi^B)}$

```
1  For i ← 1 to n do
2    for j ← 1 to n do
3      if i = j
4        then Π^{A∘B}_{i,j} ← i
5        else if min{a_{i,r} + b_{r,j}; 1 ≤ r ≤ n} = ∞
6               then Π^{A∘B}_{i,j} ← NIL
7               else r* ← SelectMin(a_{i,r} + b_{r,j}; 1 ≤ r ≤ n)
8                    if j = r*
9                      then Π^{A∘B}_{i,j} ← Π^A_{i,j}
10                     else Π^{A∘B}_{i,j} ← Π^B_{r*,j}
```

Laufzeit: $O(n)$ für jeden Eintrag $\Pi^{A\circ B}_{i,j}$, also $O(n^3)$.

Beobachtung: Für alle s, t, $1 \leq s, t \leq n-1$, gilt $\Pi^{(s+t)} = Link(D^{(s)}, D^{(t)}, \Pi^{(s)}, \Pi^{(t)})$.

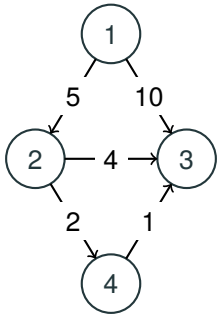
Simultane Berechnung von Δ_G und Π_G (Fast)

FastAllPairsShortestPaths(W, n)

1 $D \leftarrow W$
2 **Compute** $\Pi^{(1)}$ from W
3 $\Pi \leftarrow \Pi^{(1)}$
4 $s \leftarrow 0$
4 **While** $2^s < n - 1$
5 **do** $\Pi \leftarrow \mathit{Link}(D, D, \Pi, \Pi)$
6 $D \leftarrow D \circ D$
6 $s \leftarrow s + 1$
7 **Output** D
8 **Output** Π

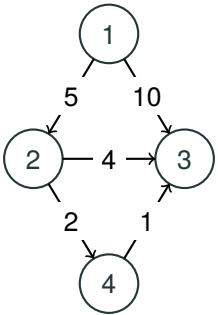
Gesamtlaufzeit $O(|V|^3 \cdot \log(|V|))$.

Beispielgraph mit W und $\Pi^{(1)}$



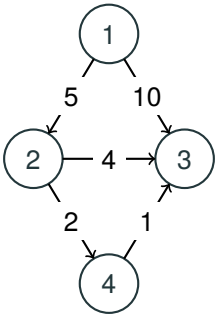
$$W = \begin{pmatrix} 0 & 5 & 10 & \infty \\ \infty & 0 & 4 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}, \quad \Pi^{(1)} = \begin{pmatrix} 1 & 1 & 1 & N \\ N & 2 & 2 & 2 \\ N & N & 3 & N \\ N & N & 4 & 4 \end{pmatrix}.$$

Beispielgraph mit $D^{(2)}$ und $\Pi^{(2)}$



$$D^{(2)} = \begin{pmatrix} 0 & 5 & 9 & 7 \\ \infty & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}, \quad \Pi^{(2)} = \begin{pmatrix} 1 & 1 & 2 & 2 \\ N & 2 & 4 & 2 \\ N & N & 3 & N \\ N & N & 4 & 4 \end{pmatrix}.$$

Beispielgraph mit $D^{(3)}$ und $\Pi^{(3)}$



$$D^{(3)} = \begin{pmatrix} 0 & 5 & 8 & 7 \\ \infty & 0 & 3 & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix}, \quad \Pi^{(3)} = \begin{pmatrix} 1 & 1 & 4 & 2 \\ N & 2 & 4 & 2 \\ N & N & 3 & N \\ N & N & 4 & 4 \end{pmatrix}.$$

Kürzeste Wege in gewichteten gerichteten Graphen, All Pairs Shortest Paths

Der Floyd-Warshall Algorithmus

Ein alternativer dynamischer Ansatz

- **Eingabe** gewichtete Graphen $G = (V, E, w)$ mit Kantengewichtsfunktion $w : E \longrightarrow \mathbb{R}$ ohne Kreise negativen Gewichts, $V = \{1, \cdots, n\}$.
- **Strukturierung des Suchraums:**

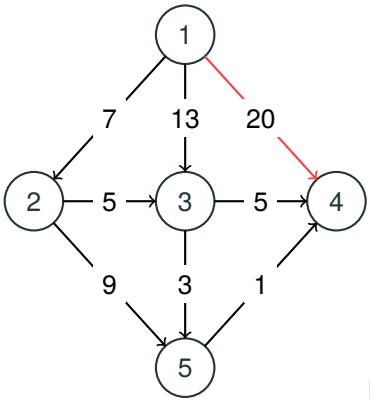
  Für alle $1 \leq i, j \leq n$, und $0 \leq k \leq n$ sei $Path_{i,j}^k$ **die Menge aller Wege von** i **nach** j **in** G**, die nur Zwischenknoten aus** $\{1, \cdots, k\}$ **haben**,

  $$e_{i,j}^k = \min\{w(p), p \in Path_{i,j}^k\},$$

  bzw. $e_{i,j}^k = \infty$ falls $Path_{i,j}^k = \emptyset$.
- Für alle $k \geq 0$ sei $E^{(k)} = (e_{i,j}^k)_{i,j=1}^n$.
- **Beobachtung 1:** Es gilt $E^{(0)} = W$
- **Beobachtung 2:** Es gilt $E^{(n)} = \Delta_G$.
- **Idee des Floyd-Warshall Algorithmus:**

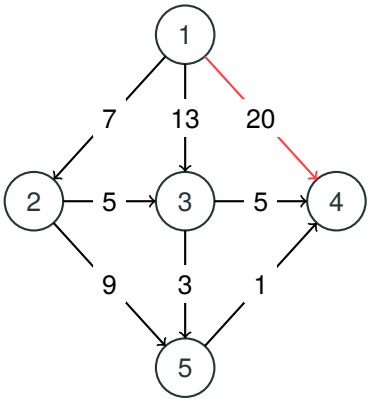
  Effiziente Berechnung von $E^{(k)}$ aus $E^{(k-1)}$ für $k > 0$.

Beispiel mit $W, E^{(0)}, E^{(1)}$



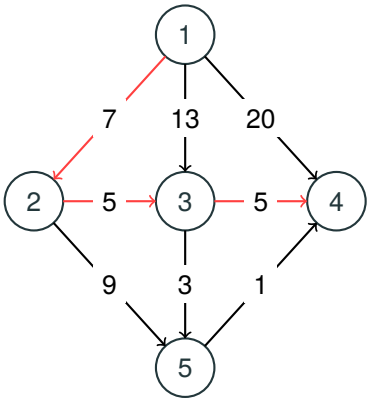
$$W = E^{(0)} = E^{(1)} = \begin{pmatrix} 0 & 7 & 13 & 20 & \infty \\ \infty & 0 & 5 & \infty & 9 \\ \infty & \infty & 0 & 5 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Beispiel mit $E^{(2)}$



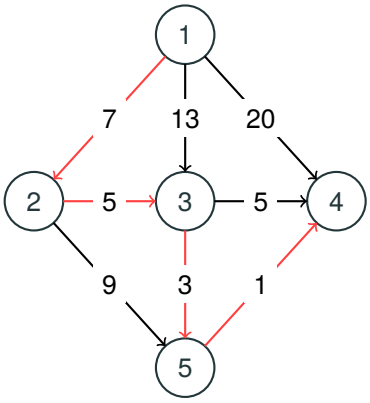
$$E^{(2)} = \begin{pmatrix} 0 & 7 & 12 & 20 & 16 \\ \infty & 0 & 5 & \infty & 9 \\ \infty & \infty & 0 & 5 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Beispiel mit $E^{(3)}, E^{(4)}$



$$E^{(3)} = E^{(4)} = \begin{pmatrix} 0 & 7 & 12 & 17 & 15 \\ \infty & 0 & 5 & 10 & 8 \\ \infty & \infty & 0 & 5 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Beispiel mit $E^{(5)}$



$$E^{(5)} = \begin{pmatrix} 0 & 7 & 12 & 16 & 15 \\ \infty & 0 & 5 & 10 & 8 \\ \infty & \infty & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Effiziente Berechnung von $E^{(k)}$ aus $E^{(k-1)}$ für $k > 0$

Lemma 72

Für alle $i \neq j$, $1 \leq i, j \leq n$, und $k > 0$ gilt $e^k_{i,i} = 0$ und

$$e^k_{i,j} = \min\{e^{k-1}_{i,j}, e^{k-1}_{i,k} + e^{k-1}_{k,j}\}.$$

Beweis: Es gibt entweder einen kürzesten Weg p in $Path^k_{i,j}$, der k **als echten Zwischenknoten** enthält, oder nicht.

Falls ja, kann p in einen kürzesten Weg aus $Path^{k-1}_{i,k}$ und einen kürzesten aus $Path^{k-1}_{k,j}$ aufgespalten werden.

Also $e^k_{i,j} = e^{k-1}_{i,k} + e^{k-1}_{k,j}$.

Falls nein gilt $e^k_{i,j} = e^{k-1}_{i,j}$. □

Die entsprechende Berechnung von Π_G

Definition 73

Für alle $k \geq 0$ sei die Matrix $\Pi^{[k]} = (\pi_{i,j}^{[k]})_{i,j=1}^{n}$ definiert durch

- $\pi_{i,i}^{[k]} = i$ für alle $1 \leq i \leq n$,
- $\pi_{i,j}^{[k]} = \mathit{NIL}$ falls $\mathit{Path}_{i,j}^{k} = \emptyset$,
- $\pi_{i,j}^{[k]}$ ist der Vorgänger von j auf einem kürzesten Weg aus $\mathit{Path}_{i,j}^{k}$, falls $i \neq j$ und $\mathit{Path}_{i,j}^{k} \neq \emptyset$.

- **Aussehen von $\Pi^{[0]}$:** Es gilt $\pi_{i,j}^{0} = i$ falls $(i,j) \in E$, ansonsten $\pi_{i,j}^{0} = \mathit{NIL}$.
- **Beobachtung 1:** Es gilt $\Pi^{[n]} = \Pi_G$.

Beobachtung 2: Für $k > 0$ und $i \neq j$ ist lässt sich $\pi_{i,j}^{k}$ **in** $O(1)$ **aus** $\Pi^{[k-1]}$ **berechnen**, denn

- $\pi_{i,j}^{k} = \pi_{i,j}^{k-1}$ falls $e_{i,j}^{k} = e_{i,j}^{k-1}$,
- $\pi_{i,j}^{k} = \pi_{k,j}^{k-1}$ falls $e_{i,j}^{k} < e_{i,j}^{k-1}$

Der Algorithmus

FloydWarshall(W)

1 **Compute** $\Pi^{[0]}$ from W
2 $E^{(0)} \leftarrow W$
3 **For** $k \leftarrow 1$ **to** n
4 **do compute** $E^{(k)}$ **from** $E^{(k-1)}$ **and** $\Pi^{[k]}$ **from** $\Pi^{[k-1]}$
5 **Output** $E^{(n)}$
6 **Output** $\Pi^{[n]}$

Beobachtung: Für alle $k > 0$ lässt sich $E^{(k)}$ aus $E^{(k-1)}$ und $\Pi^{[k]}$ aus $\Pi^{[k-1]}$ in $O(|V|^2)$ berechnen.

Gesamtlaufzeit: $O(|V|^3)$.