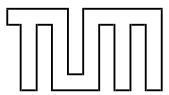
Grundlagen: Algorithmen und Datenstrukturen

Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2013



Übersicht

Vorlesungsdaten

- Titel: “Grundlagen: Algorithmen und Datenstrukturen” / GAD
- 3 SWS Vorlesung + 2 SWS Übung
- Modul: IN0007, ECTS: 6 Credit Points
- Vorlesungszeiten:

  Dienstag 14:15 – 15:45 Uhr (MI Hörsaal 1 + Int. Hs. 1)
  Donnerstag 12:15 – 13:00 Uhr (MI Hörsaal 1 + Int. Hs. 1)
- Webseite: http://www14.in.tum.de/lehre/2013SS/gad/
- Voraussetzung:
  Modul IN0001: Einführung in die Informatik 1
- Klausur:
  Gesamtklausur am Samstag, 27.07.2013 (11:30–14:00 Uhr)
  Wiederholungsklausur am Freitag, 27.09.2013 (11:00-13:30 Uhr)

Zielgruppe

- Bachelor Informatik
- Bachelor Wirtschaftsinformatik
- Bachelor Bioinformatik
- Bachelor Informatik: Games Engineering
- Andere Studiengänge mit Neben-/Zweitfach Informatik
- Masterstudiengang Angewandte Informatik
- Aufbaustudium Informatik
- Schülerstudium

- planmäßig im 2. Fachsemester

Dozent / Kontaktdaten

- Hanjo Täubig

  Lehrstuhl für Effiziente Algorithmen
  (Lehrstuhlinhaber: Prof. Dr. Ernst W. Mayr)

- eMail: taeubig@in.tum.de

- Web: http://www14.in.tum.de/personen/taeubig/

- Telefon: 089 / 289-17740

- Raum: 03.09.039 (3. Stock, Finger 9)

- Sprechstunde: Mittwoch 13-14 Uhr
  (oder nach Vereinbarung)

Übung

- 2 SWS Tutorübungen
- 42 Gruppen an 14 verschiedenen Terminen
- jeweils maximal 16-20 Teilnehmer
- Anmeldung über TUMonline:
  https://campus.tum.de/
- Übungsleitung:

  Jeremias Weihmann (weihmann@in.tum.de)
- Webseite:

  http://www14.in.tum.de/lehre/2013SS/gad/uebung/

Inhalt

- Grundlagen der Komplexitätsanalyse
- Sequenzrepräsentation durch dynamische Felder und Listen
- binäre Bäume und Algorithmen
- binäre Suchbäume und balancierte Suchbäume
- Prioritätswarteschlangen
- Hashing
- Sortieren und sortierbasierte Algorithmen
- Graph-Repräsentation und einfache Graphalgorithmen
- Pattern Matching
- Datenkompression

Grundlage

- Inhalt der Vorlesung basiert auf dem Buch

  K. Mehlhorn, P. Sanders:
  Algorithms and Data Structures – The Basic Toolbox
  (Springer, 2008)

  http://www.mpi-inf.mpg.de/~mehlhorn/Toolbox.html

- Vorlage für die Slides:

  Slides aus SS'08 von Prof. Dr. Christian Scheideler bzw.
  Slides aus SS'09 von Prof. Dr. Helmut Seidl

Weitere Literatur

- Cormen, Leiserson, Rivest, Stein:
  Introduction to Algorithms
- Goodrich, Tamassia:
  Algorithm Design: Foundations, Analysis, and Internet Examples
- Heun:
  Grundlegende Algorithmen
  Einführung in den Entwurf und die Analyse effizienter Algorithmen
- Kleinberg, Tardos:
  Algorithm Design
- Schöning:
  Algorithmik
- Sedgewick:
  Algorithmen in Java. Teil 1-4

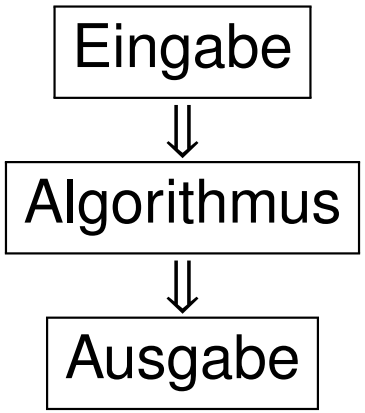
Übersicht

Übersicht

Algorithmus - Definition

Definition

Ein Algorithmus ist eine formale Handlungsvorschrift zur Lösung von Instanzen eines Problems in endlich vielen Schritten.



Algorithmus - Analogien und Beispiele

- Kochrezept
  - Eingabe: Zutaten
  - Algorithmus: Rezept
  - Ausgabe: Essen
- Bauanleitung
  - Eingabe: Einzelteile
  - Algorithmus: Bauanleitung
  - Ausgabe: Fertiges Möbelstück
- Weitere Beispiele
  - Weg aus dem Labyrinth
  - Zeichnen eines Kreises

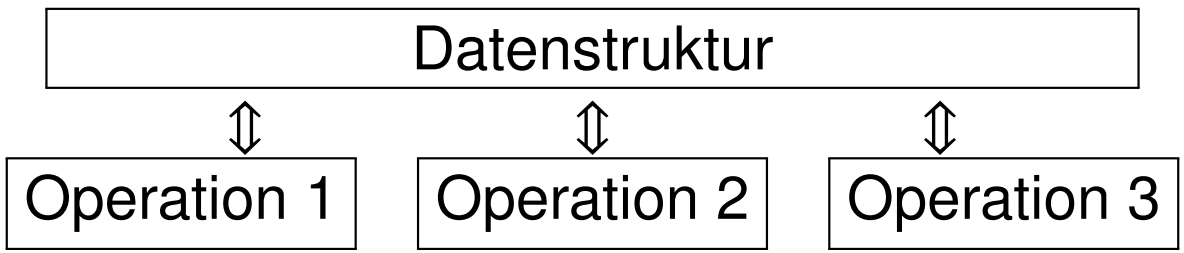
Datenstruktur - Definition

Definition

Eine Datenstruktur ist ein formalisiertes Objekt, das

- der Speicherung und
- der Verwaltung von bzw.
- dem Zugriff auf Daten

in geeigneter Weise dient, wobei die Daten dabei zweckdienlich angeordnet, kodiert und miteinander verknüpft werden.



Datenstruktur - Beispiele

- Lexikon
  Operation: Suche(Name)
  (Algorithmus mit Eingabe ⟨Name⟩,
  Ausgabe Information zu ⟨Name⟩)

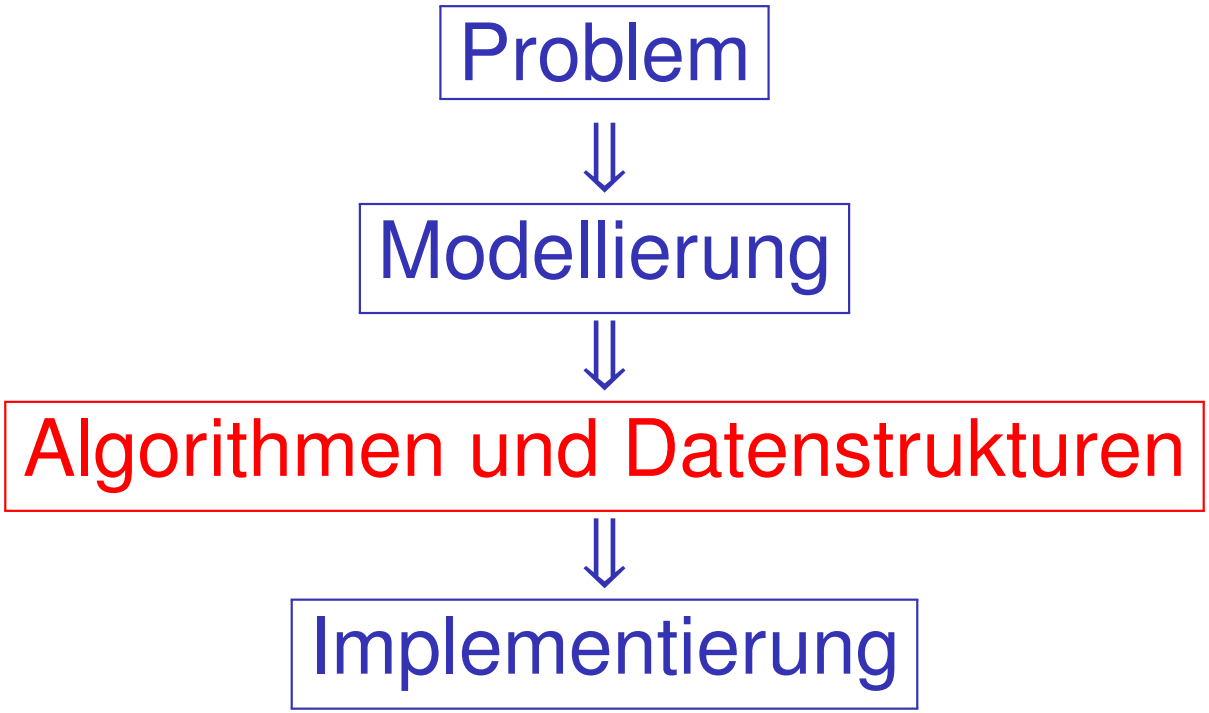
- Kalender
  Operation: Wochentag(Datum)
  (Algorithmus mit Eingabe ⟨Datum⟩,
  Ausgabe Wochentag zum ⟨Datum⟩)

Abstrakter Datentyp:

- legt fest, welche Operationen was tun (Semantik),
- aber nicht wie (konkrete Implementierung)

$\Rightarrow$ Kapselung durch Definition einer Schnittstelle

Softwareentwicklung



- Abstraktion vom genauen Problem (Vereinfachung)
- geeignete Auswahl von Algorithmen / Datenstrukturen
- Grundsätzliche Probleme: Korrektheit, Komplexität, Robustheit / Sicherheit, aber vor allem Effizienz

Effizienz

im Sinn von

- Laufzeit
- Speicheraufwand
- Festplattenzugriffe
- Energieverbrauch

Kritische Beispiele:

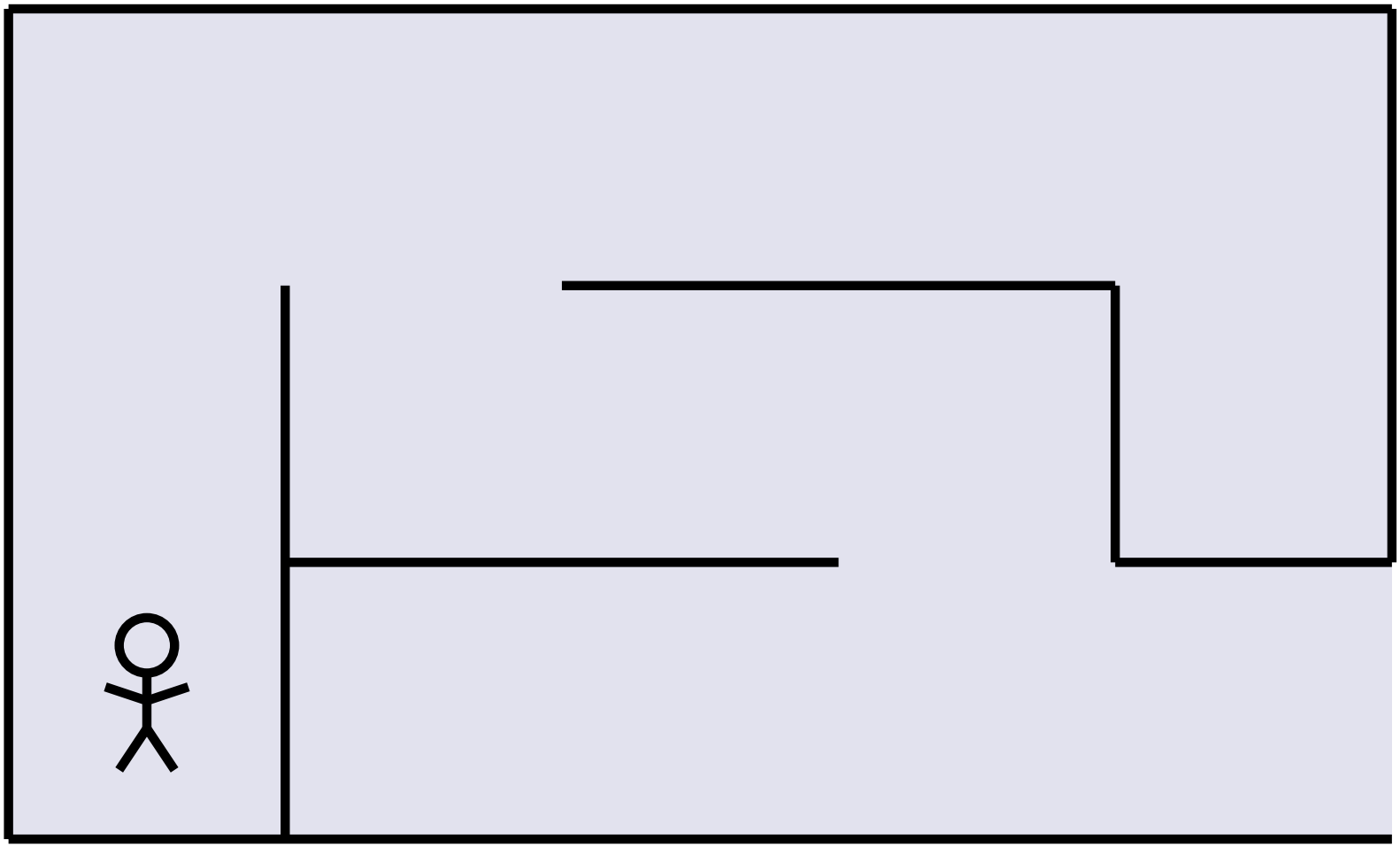
- Riesige Datenmengen (Bioinformatik)
- Echtzeitanwendungen (Spiele, Flugzeugsteuerung)

Ziel der Vorlesung:

Grundstock an effizienten Algorithmen und Datenstrukturen für Standardprobleme

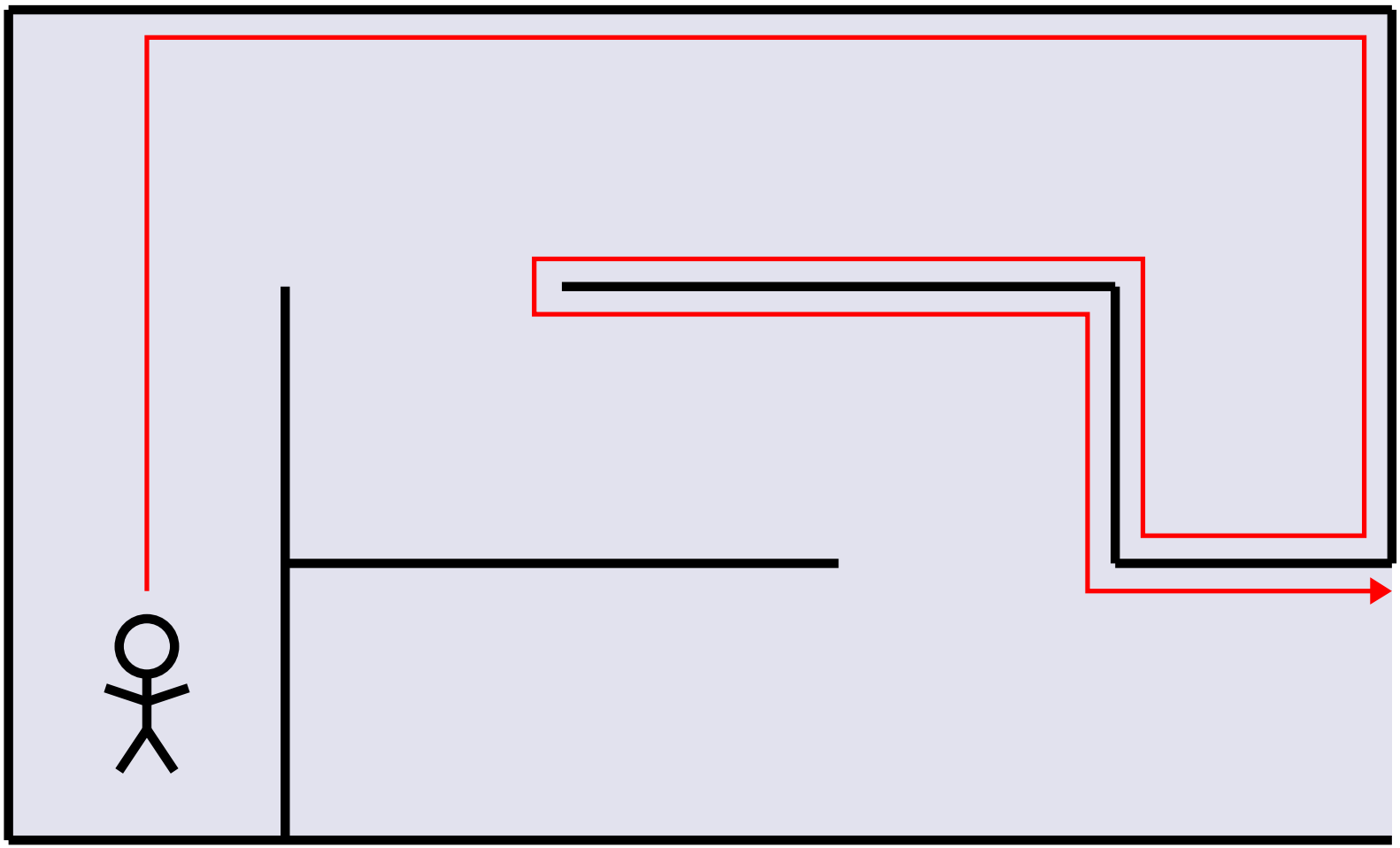
Übersicht

Weg aus dem Labyrinth



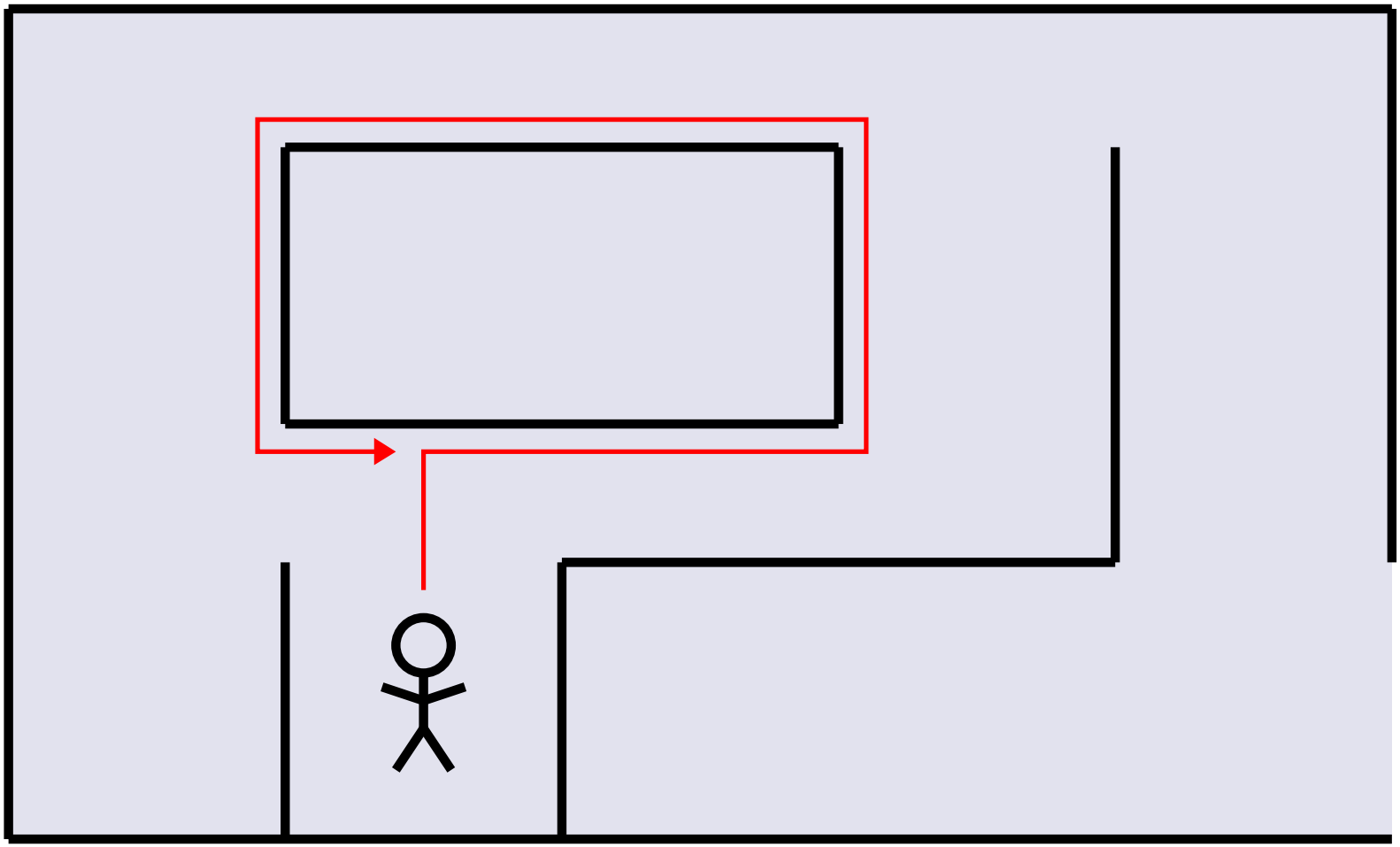
Problem: Es ist dunkel!

Weg aus dem Labyrinth



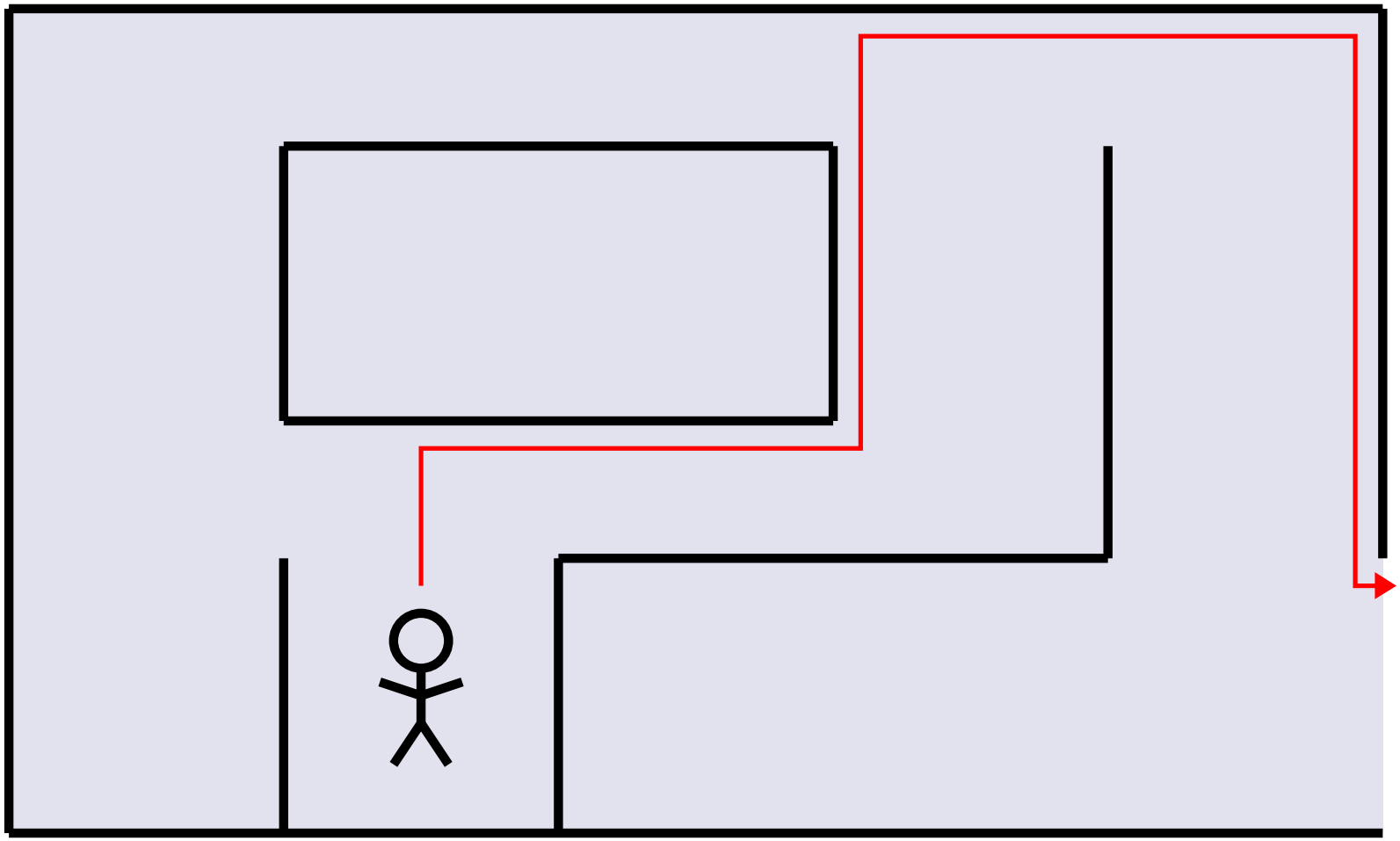
1. Versuch: mit einer Hand immer an der Wand lang

Weg aus dem Labyrinth



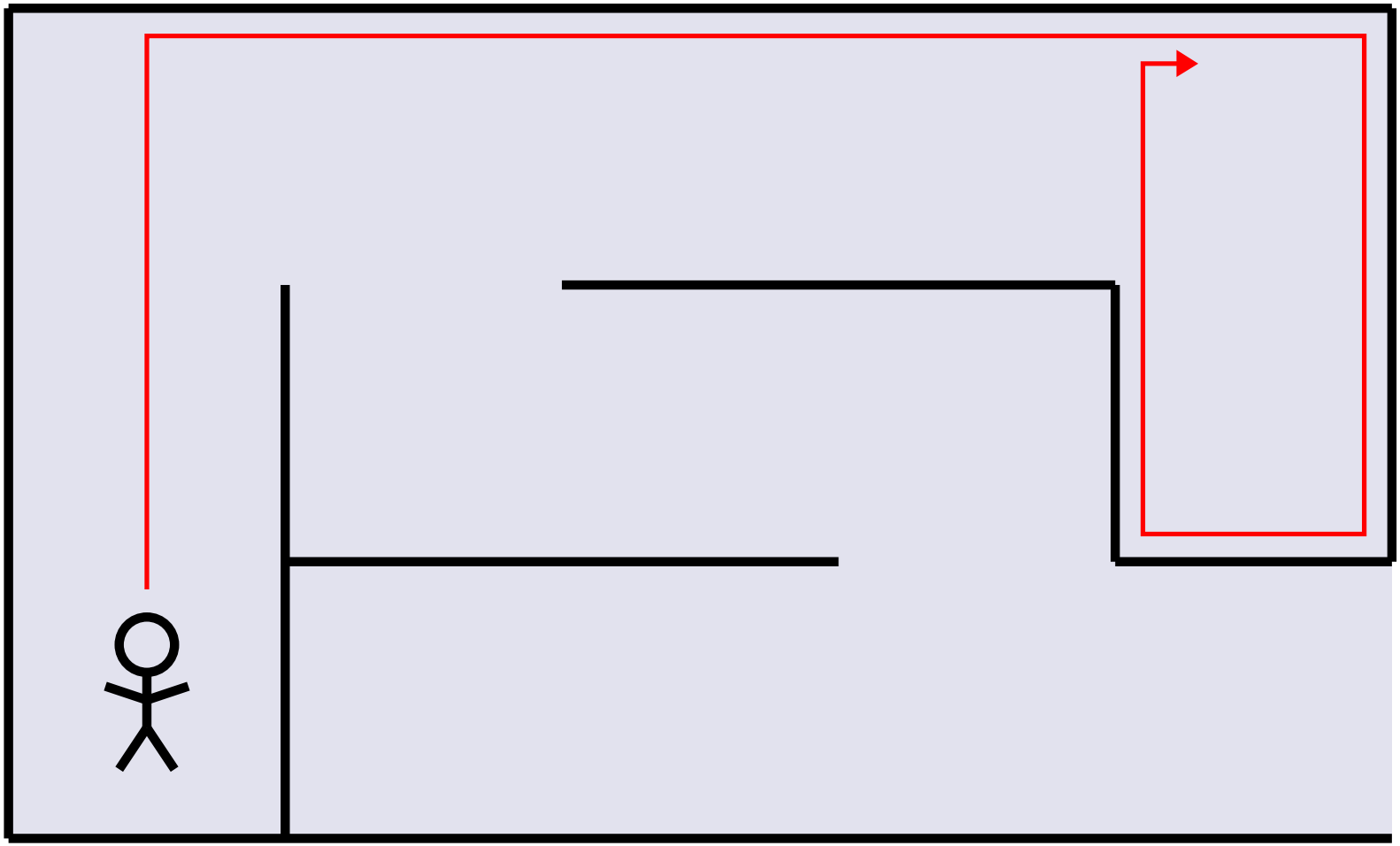
Problem: Inseln werden endlos umkreist

Weg aus dem Labyrinth



2. Versuch: gerade bis zur Wand, der Wand folgen bis man wieder in dieselbe Richtung läuft, dann wieder gerade bis zur Wand usw.

Weg aus dem Labyrinth



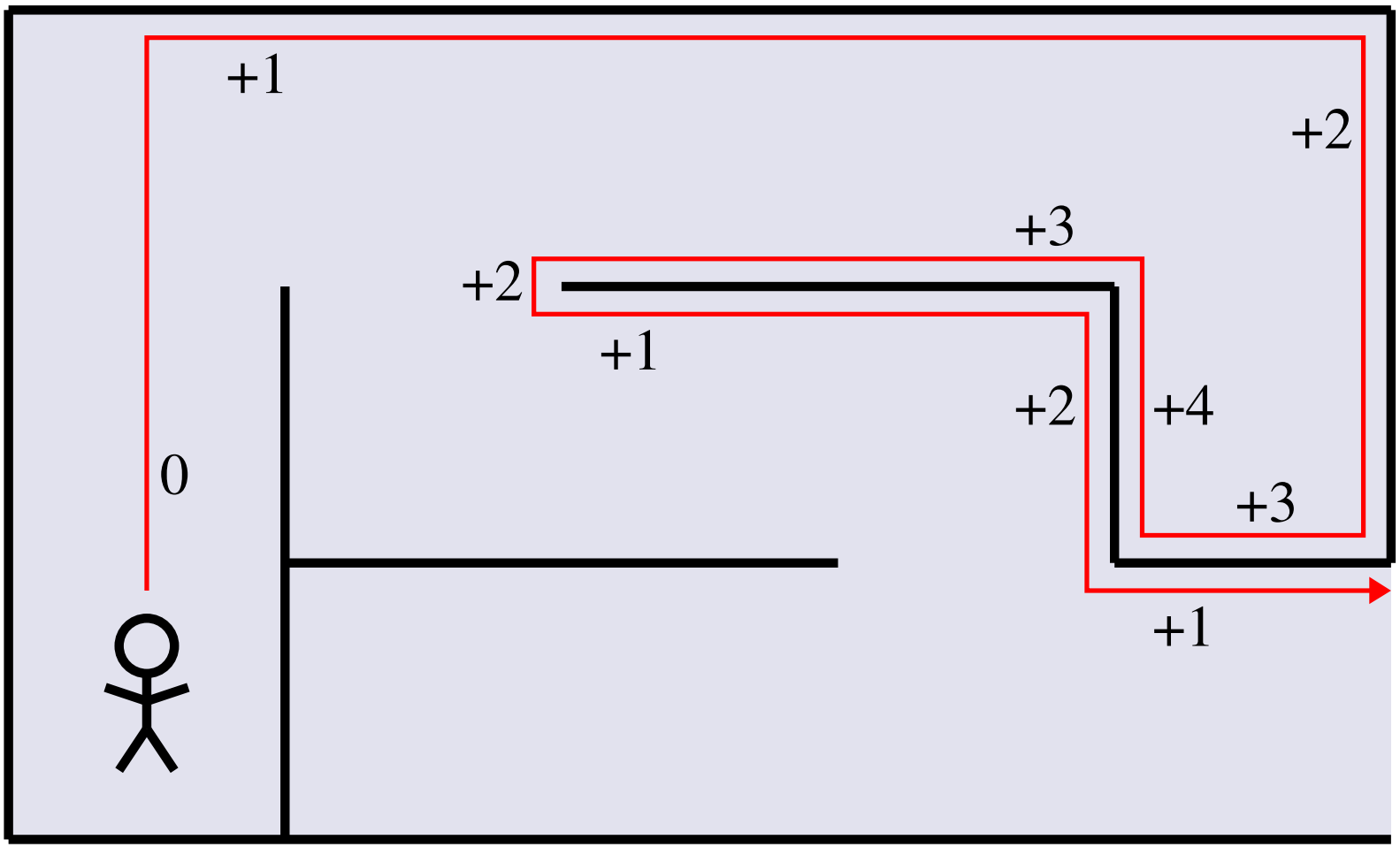
Problem: Jetzt laufen wir im ersten Beispiel im Kreis

Pledge-Algorithmus

Labyrinth-Suche

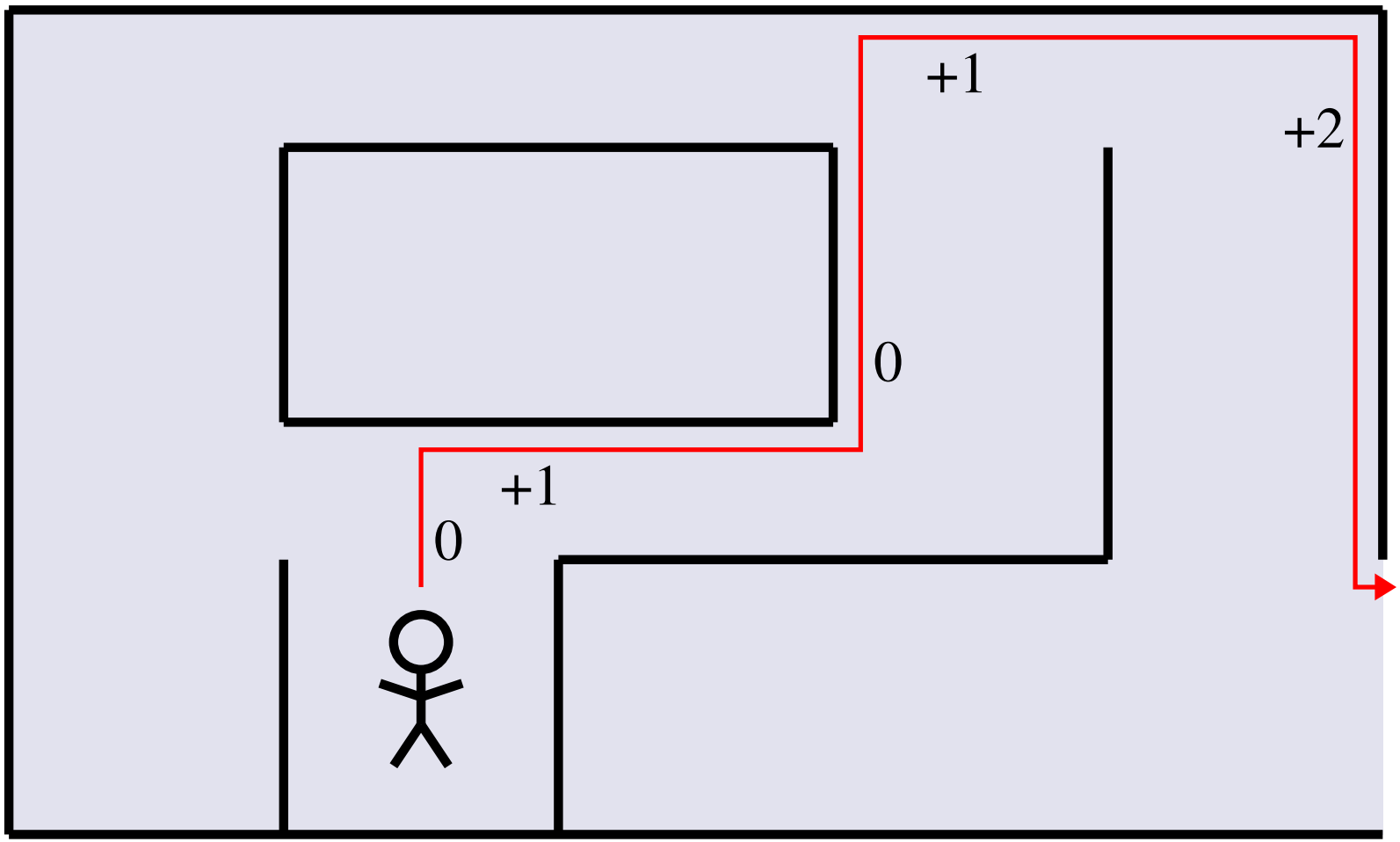
```
Setze Umdrehungszähler auf 0;
repeat
    repeat
        Gehe geradeaus;
    until Wand erreicht;
    Drehe nach rechts;
    Inkrementiere Umdrehungszähler;
    repeat
        Folge dem Hindernis mit einer Hand;
        dabei: je nach Drehrichtung Umdrehungszähler
        inkrementieren / dekrementieren;
    until Umdrehungszähler=0;
until Ausgang erreicht;
```

Weg aus dem Labyrinth



1. Beispiel funktioniert

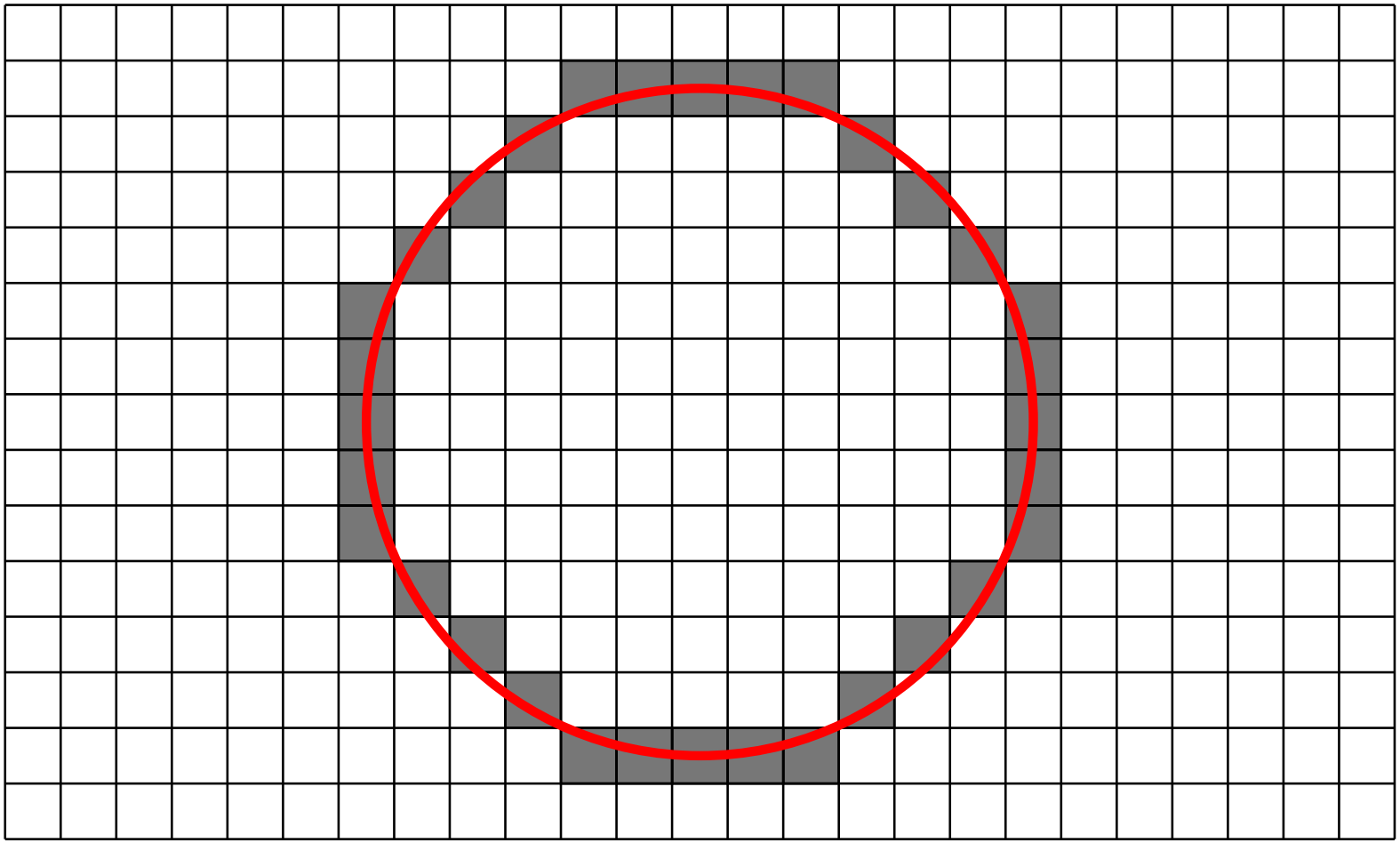
Weg aus dem Labyrinth



2. Beispiel funktioniert auch

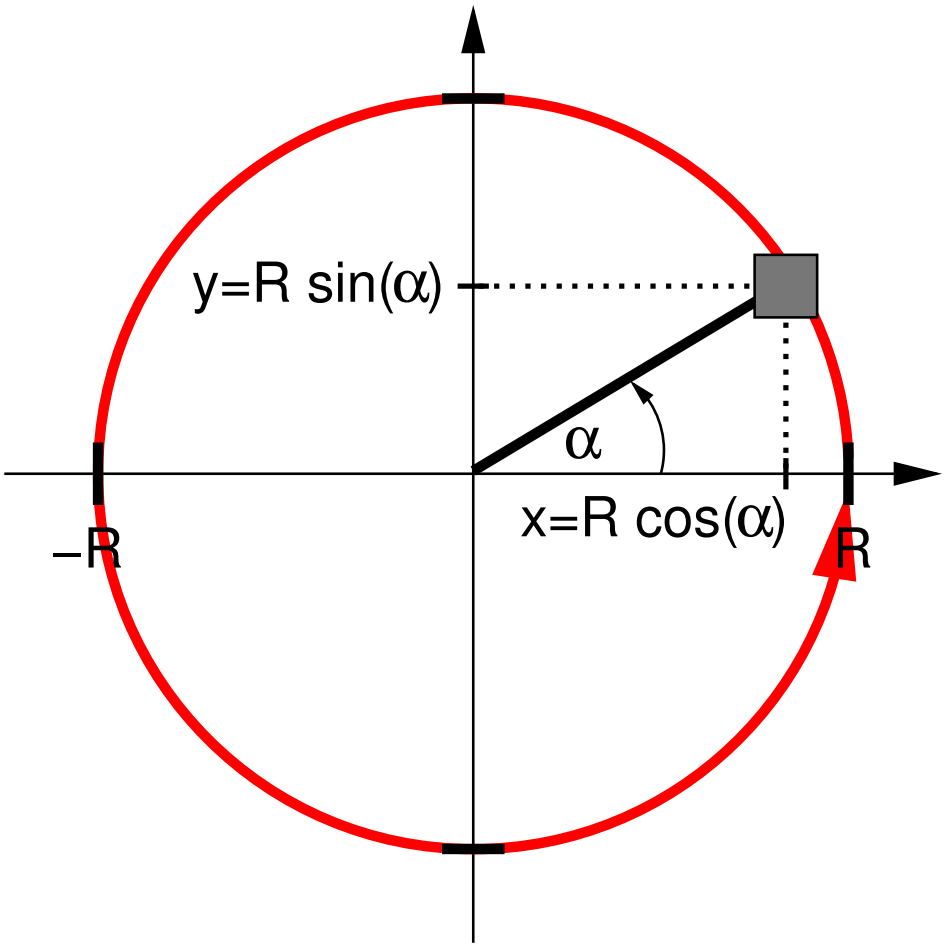
Kreis zeichnen

Wie kann ein Computer einen Kreis zeichnen?



Kreis zeichnen: mit Winkelfunktionen

Naiver Ansatz: eine Runde wie mit dem Zirkel



Verwendung von sin() und cos() für $\alpha = 0 \ldots 2\pi$

Kreis zeichnen: mit Winkelfunktionen

Kreis1

Eingabe : Radius R
Pixelanzahl n

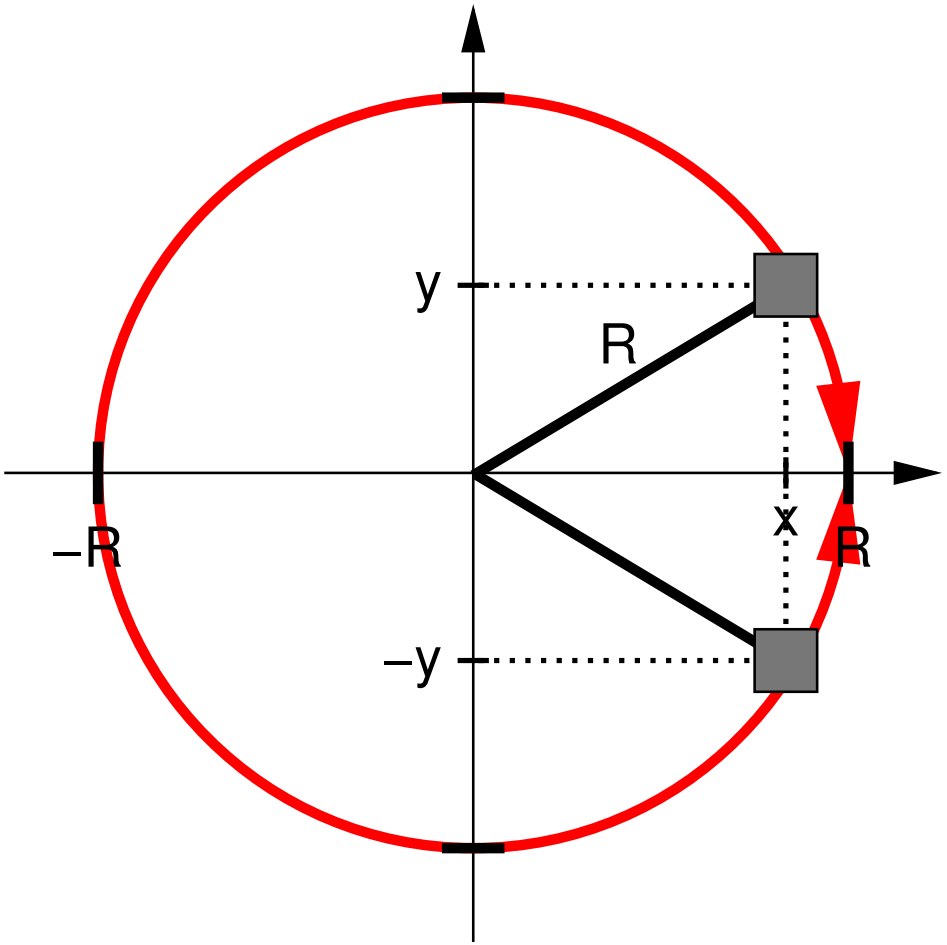
for $i = 0; i < n; i{+}{+}$ **do**
  plot($R * \cos(2\pi * i/n), R * \sin(2\pi * i/n)$);

Kreisumfang: $u = 2\pi \cdot R$
$\Rightarrow$ Bei Pixelbreite von 1 Einheit reicht $n = \lceil 2\pi R \rceil$.

Problem: sin() und cos() sind teuer!

Kreis zeichnen: mit Wurzelfunktion

Schnellerer Ansatz: $x^2 + y^2 = R^2$ bzw. $y = \pm\sqrt{R^2 - x^2}$



1 Pixel pro Spalte für oberen / unteren Halbkreis

Kreis zeichnen: mit Wurzelfunktion

Kreis2

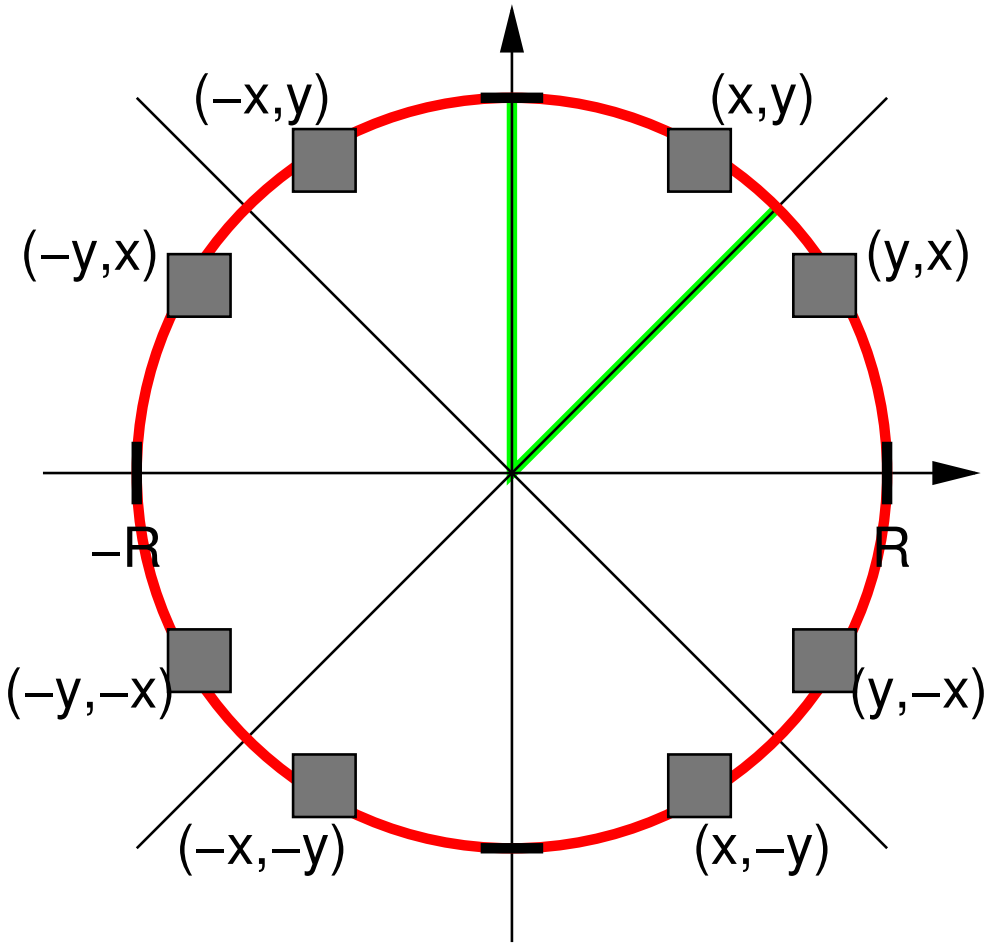
Eingabe : radius R

for $x = -R; x \leq R; x{+}{+}$ **do**
    $y = \text{sqrt}(R * R - x * x)$;
    plot(x, y);
    plot(x, -y);

Problem: sqrt() ist auch noch relativ teuer!

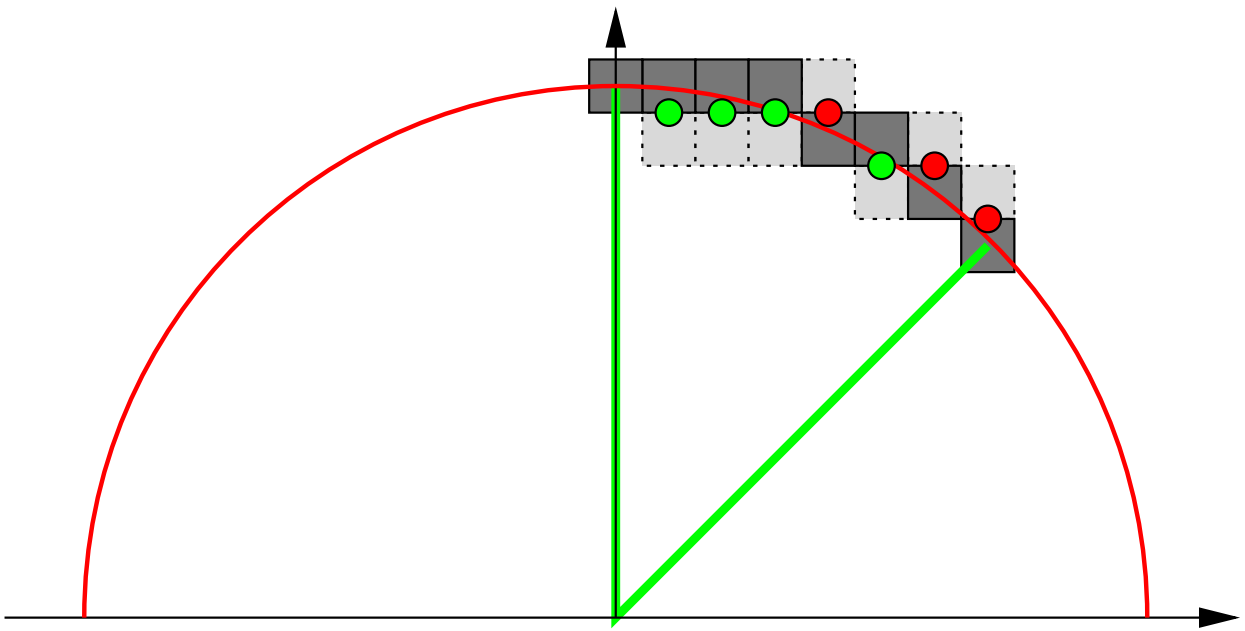
Kreis zeichnen: mit Multiplikation

Besserer Ansatz: Ausnutzung von Spiegelachsen



Kreis zeichnen: mit Multiplikation

- betrachtetes Kreissegment: Anstieg zwischen 0 und -1
- 2 Fälle für nächstes Pixel: nur rechts oder rechts unten
- Entscheidungskriterium:
  Grundlinienmittelpunkt des rechten Nachbarpixels innerhalb vom Kreis? ja: $x++$ nein: $x++;\quad y--$



Kreis zeichnen: mit Multiplikation

- Test, ob (x, y) innerhalb des Kreises:

$$F(x, y) := x^2 + y^2 - R^2 < 0$$

- Mittelpunkt des ersten Quadrats: $(x, y) = (0, R)$
- Position seines Grundlinienmittelpunkts: $(0, R - \frac{1}{2})$
- Grundlinienmittelpunkt für Pixel rechts daneben:
  $F(1, R - \frac{1}{2}) = 1^2 + (R - \frac{1}{2})^2 - R^2 = \frac{5}{4} - R < 0$?
- Update:

$$\begin{aligned}
F(x+1, y) &= (x+1)^2 + y^2 - R^2 = (x^2 + 2x + 1) + y^2 - R^2 \\
F(x+1, y) &= F(x, y) + 2x + 1 \\
F(x+1, y-1) &= (x+1)^2 + (y-1)^2 - R^2 \\
&= (x^2 + 2x + 1) + (y^2 - 2y + 1) - R^2 \\
F(x+1, y-1) &= F(x, y) + 2x - 2y + 2
\end{aligned}$$

Kreis zeichnen: mit Multiplikation

Bresenham1

$x = 0; \quad y = R;$
$\text{plot}(0, R); \quad \text{plot}(R, 0); \quad \text{plot}(0, -R); \quad \text{plot}(-R, 0);$
$F = \frac{5}{4} - R;$
while $x < y$ **do**
  if $F < 0$ **then**
    $F = F + 2 * x + 1;$
  else
    $F = F + 2 * x - 2 * y + 2;$
    $y = y - 1;$
  $x = x + 1;$
  $\text{plot}(x, y); \quad \text{plot}(-x, y); \quad \text{plot}(-y, x); \quad \text{plot}(-y, -x);$
  $\text{plot}(y, x); \quad \text{plot}(y, -x); \quad \text{plot}(x, -y); \quad \text{plot}(-x, -y);$

Es geht sogar noch etwas schneller!

Kreis zeichnen: mit Addition / Subtraktion

- Ersetzung der Korrekturterme für F:

$$F = F + 2x + 1 \quad \rightarrow \quad F = F + d_E$$
$$F = F + 2x - 2y + 2 \quad \rightarrow \quad F = F + d_{SE}$$

mit $d_E = 2x + 1$ und $d_{SE} = 2x - 2y + 2$

- Anfangswerte:

$$\begin{aligned} d_E(0, R) &= 2 \cdot 0 + 1 = 1 \\ d_{SE}(0, R) &= 2 \cdot 0 - 2 \cdot R + 2 = 2 - 2 \cdot R \end{aligned}$$

- Updates nach rechts (E) und nach unten rechts (SE):

$$\begin{aligned} d_E(x+1, y) &= 2 \cdot (x+1) + 1 = d_E(x, y) + 2 \\ d_{SE}(x+1, y) &= 2 \cdot (x+1) - 2 \cdot y + 2 = d_{SE}(x, y) + 2 \\ d_E(x+1, y-1) &= 2 \cdot (x+1) + 1 = d_E(x, y) + 2 \\ d_{SE}(x+1, y-1) &= 2 \cdot (x+1) - 2 \cdot (y-1) + 2 = d_{SE}(x, y) + 4 \end{aligned}$$

Kreis zeichnen: mit Addition / Subtraktion

- Der Bruch $\frac{5}{4}$ kann durch 1 ersetzt werden, weil sich F immer um eine ganze Zahl ändert.
- D.h.

$$F = \frac{5}{4} - R + k < 0$$

ist äquivalent zu

$$F = 1 - R + k < 0$$

- Vorteil:

nur noch ganze Zahlen!

Kreis zeichnen: mit Addition / Subtraktion

Bresenham2

$x = 0; \quad y = R; \quad \text{plot}(0, R); \text{plot}(R, 0); \text{plot}(0, -R); \text{plot}(-R, 0);$
$F = 1 - R;$
$d_E = 1; \quad d_{SE} = 2 - R - R;$
while $x < y$ **do**
  if $F < 0$ **then**
    $F = F + d_E;$
    $d_{SE} = d_{SE} + 2;$
  else
    $F = F + d_{SE};$
    $y = y - 1;$
    $d_{SE} = d_{SE} + 4;$
  $x = x + 1; \quad d_E = d_E + 2;$
  $\text{plot}(x, y); \quad \text{plot}(-x, y); \quad \text{plot}(-y, x); \quad \text{plot}(-y, -x);$
  $\text{plot}(y, x); \quad \text{plot}(y, -x); \quad \text{plot}(x, -y); \quad \text{plot}(-x, -y);$

Bresenham-Algorithmus

- Ab Anfang der 1960er Jahre hat Jack Bresenham Algorithmen zur Linien- und Kreisdarstellung entwickelt.
- Diese verwenden nur einfache Additionen ganzer Zahlen.
- Sie sind damit deutlich schneller als die naiven Ansätze.

Multiplikation langer Zahlen

Schulmethode:

- gegeben Zahlen a und b
- multipliziere a mit jeder Ziffer von b
- addiere die Teilprodukte

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

Aufwand

- Wenn die Zahlen klein sind, ist der Aufwand ok.
- Aber wenn die Zahlen sehr lang sind, kann man das Produkt dann schneller ausrechnen als mit der Schulmethode?

$\Rightarrow$ Wie wollen wir die Zeit oder den Aufwand überhaupt messen?

- Am besten nicht in Sekunden, die irgendein Rechner braucht, denn das könnte für einen anderen Rechner eine ganz andere Zahl sein.
- Außerdem werden die Computer ja von Generation zu Generation immer schneller und leistungsfähiger.

$\Rightarrow$ Wir zählen Grundoperationen: Operationen, die man in einem einzigen Schritt bzw. in einer konstanten Zeiteinheit ausführen kann.

Grundoperation

- Multiplikation von zwei Ziffern: $x \cdot y = ?$

  Das Ergebnis besteht aus (höchstens) zwei Ziffern u (Zehnerstelle) und v (Einerstelle), also

  $$x \cdot y = 10 \cdot u + v$$

- Addition von drei Ziffern: $x + y + z = ?$

  Auch hier besteht das Ergebnis aus (höchstens) zwei Ziffern u (Zehnerstelle) und v (Einerstelle), also

  $$x + y + z = 10 \cdot u + v$$

  Wir benutzen hier drei Ziffern als Summenden, weil wir später Überträge berücksichtigen wollen.

Analyse der Addition

- *Zahl plus Zahl:*

$$\begin{array}{cccccc} & 6 & 9 & 1 & 7 & \\ & 4 & 2 & 6 & 9 & \\ 1 & 1 & 0 & 1 & & \\ \hline 1 & 1 & 1 & 8 & 6 & \\ \hline\hline \end{array}$$

Zur Addition zweier Zahlen mit jeweils n Ziffern brauchen wir n Additionen von 3 Ziffern, also n Grundoperationen.

Ergebnis: Zahl mit $n + 1$ Ziffern

Analyse des Teilprodukts

- *Zahl mal Ziffer:*

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

Zur Multiplikation einer Zahl bestehend aus n Ziffern mit einer einzelnen Ziffer brauchen wir

- n Multiplikationen von 2 Ziffern und
- $n+1$ Additionen von 3 Ziffern, wobei in der letzten Spalte eigentlich nichts addiert werden muss,

also $2n[+1]$ Grundoperationen.

Ergebnis: Zahl mit $n+1$ Ziffern

Analyse des Produkts

- *Zahl mal Zahl:*

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

Zur Multiplikation zweier Zahlen mit jeweils n Ziffern brauchen wir

- n Multiplikationen einer n-Ziffern-Zahl mit einer Ziffer, also $n \cdot (2n[+1]) = 2n^2[+n]$ Grundoperationen
- Zwischenergebnisse sind nicht länger als das Endergebnis ($2n$ Ziffern), also $n - 1$ Summen von Zahlen mit $2n$ Ziffern, also $(n-1) \cdot 2n = 2n^2 - 2n$ Grundoperationen

Insgesamt: $4n^2 - [2]n$ Grundoperationen

Analyse des Produkts

- *Zahl mal Zahl:*

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

Genauer:

- Beim Aufsummieren der Zwischenergebnisse muss man eigentlich jeweils nur Zahlen bestehend aus $n+1$ Ziffern addieren. Das ergibt $(n-1)(n+1) = n^2 - 1$ Grundoperationen.

Insgesamt hätte man damit $3n^2[+n] - 1$ Grundoperationen.

Geht es besser?

Frage:

- Ist das überhaupt gut?
- Vielleicht geht es ja schneller?
- Was wäre denn überhaupt eine signifikante Verbesserung?
- Vielleicht irgendetwas mit $2n^2$?
- Das würde die Zeit auf ca. $2/3$ des ursprünglichen Werts senken.
- Aber bei einer Verdoppelung der Zahlenlänge hätte man immer noch eine Vervierfachung der Laufzeit.

- Wir werden diese Frage später beantworten ...

Algorithmen-Beispiele

- Rolf Klein und Tom Kamphans:
  Der Pledge-Algorithmus: Wie man im Dunkeln aus einem Labyrinth entkommt
- Dominik Sibbing und Leif Kobbelt:
  Kreise zeichnen mit Turbo
- Arno Eigenwillig und Kurt Mehlhorn:
  Multiplikation langer Zahlen (schneller als in der Schule)
- Diese und weitere Beispiele:

Taschenbuch der Algorithmen (Springer, 2008)

Übersicht

Übersicht

Effizienzmessung

Ziel:

- Beschreibung der Performance von Algorithmen
- möglichst genau, aber in kurzer und einfacher Form

Exakte Spezifikation der Laufzeit eines Algorithmus
(bzw. einer DS-Operation):

- Menge $\mathcal{I}$ der Instanzen
- Laufzeit des Algorithmus T : $\mathcal{I} \mapsto \mathbb{N}$

Problem: T sehr schwer exakt bestimmbar / beschreibbar

Lösung: Gruppierung der Instanzen (meist nach Größe)

Eingabekodierung

Bei Betrachtung der Länge der Eingabe:

Vorsicht bei der Kodierung!

Beispiel (Primfaktorisierung)

Gegeben: Zahl $x \in \mathbb{N}$

Gesucht: Primfaktoren von x (Primzahlen $p_1, \ldots, p_k$ mit $x = \prod_{i=1}^{k} p_i^{e_i}$)

Bekannt als hartes Problem (wichtig für RSA-Verschlüsselung!)

Eingabekodierung - Beispiel Primfaktorisierung

Beispiel (Primfaktorisierung)

Trivialer Algorithmus

Teste von $y = 2$ bis $\lfloor \sqrt{x} \rfloor$ alle Zahlen, ob diese x teilen und wenn ja, dann bestimme wiederholt das Ergebnis der Division bis die Teilung nicht mehr ohne Rest möglich ist

Laufzeit: $\sqrt{x}$ Teilbarkeitstests und höchstens $\log_2 x$ Divisionen

- Unäre Kodierung von x (x Einsen als Eingabe):
  Laufzeit polynomiell bezüglich der Länge der Eingabe
- Binäre Kodierung von x ($\lceil \log_2 x \rceil$ Bits):
  Laufzeit exponentiell bezüglich der Länge der Eingabe

Eingabekodierung

Betrachtete Eingabegröße:

- Größe von Zahlen: Anzahl Bits bei binärer Kodierung
- Größe von Mengen / Folgen: Anzahl Elemente

Beispiel (Sortieren)

Gegeben: Folge von Zahlen $a_1, \ldots, a_n \in \mathbb{N}$

Gesucht: sortierte Folge der Zahlen

Größe der Eingabe: n

Manchmal Betrachtung von mehr Parametern:

- Größe von Graphen: Anzahl Knoten und Anzahl Kanten

Effizienzmessung

Sei $\mathcal{I}_n$ die Menge der Instanzen der Größe n eines Problems.

Effizienzmaße:

- Worst case:
$$t(n) = \max\{T(i) : i \in \mathcal{I}_n\}$$
- Average case:
$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$
- Best case:
$$t(n) = \min\{T(i) : i \in I_n\}$$

(Wir stellen sicher, dass max und min existieren und dass $\mathcal{I}_n$ endlich ist.)

Vor- und Nachteile der Maße

- worst case:
  liefert Garantie für die Effizienz des Algorithmus,
  evt. sehr pessimistisch
- average case:
  beschreibt durchschnittliche Laufzeit, aber nicht unbedingt übereinstimmend mit dem "typischen Fall" in der Praxis,
  ggf. Verallgemeinerung durch Wahrscheinlichkeitsverteilung
- best case:
  Vergleich mit worst case liefert Aussage über die Abweichung innerhalb der Instanzen gleicher Größe,
  evt. sehr optimistisch

Exakte Formeln für $t(n)$ sind meist sehr aufwendig bzw. nicht möglich!

$\Rightarrow$ betrachte asymptotisches Wachstum ($n \to \infty$)

Asymptotische Notation

- $f(n)$ und $g(n)$ haben gleiche Wachstumsrate, falls für große n das Verhältnis der beiden nach oben und unten durch Konstanten beschränkt ist, d.h.,

$$\exists c, d \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad c \leq \frac{f(n)}{g(n)} \leq d$$

- $f(n)$ wächst schneller als $g(n)$, wenn es für alle positiven Konstanten c ein n_0 gibt, ab dem $f(n) \geq c \cdot g(n)$ für $n \geq n_0$ gilt, d.h.,

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Beispiel

n^2 und $5n^2 - 7n$ haben gleiche Wachstumsrate, da für alle $n \geq 2$ $1 \leq \frac{5n^2-7n}{n^2} \leq 5$ gilt. Beide wachsen schneller als $n^{3/2}$.

Asymptotische Notation

- Warum die Betrachtung der Wachstumsrate und die Forderung nur für genügend große n?

  Ziel effizienter Algorithmen: Lösung großer Probleminstanzen

  gesucht: Verfahren, die für große Instanzen noch effizient sind

  Für große n sind Verfahren mit kleinerer Wachstumsrate besser.

- Warum Verzicht auf konstante Faktoren?

  Unser Maschinenmodell ist nur eine Abstraktion von echten Computern und kann die eigentliche Rechenzeit sowieso nur bis auf konstante Faktoren bestimmen.

  Deshalb ist es in den meisten Fällen sinnvoll, Algorithmen mit gleichem Wachstum erstmal als gleichwertig zu betrachten.

- Außerdem: Laufzeitangabe durch einfache Funktionen

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

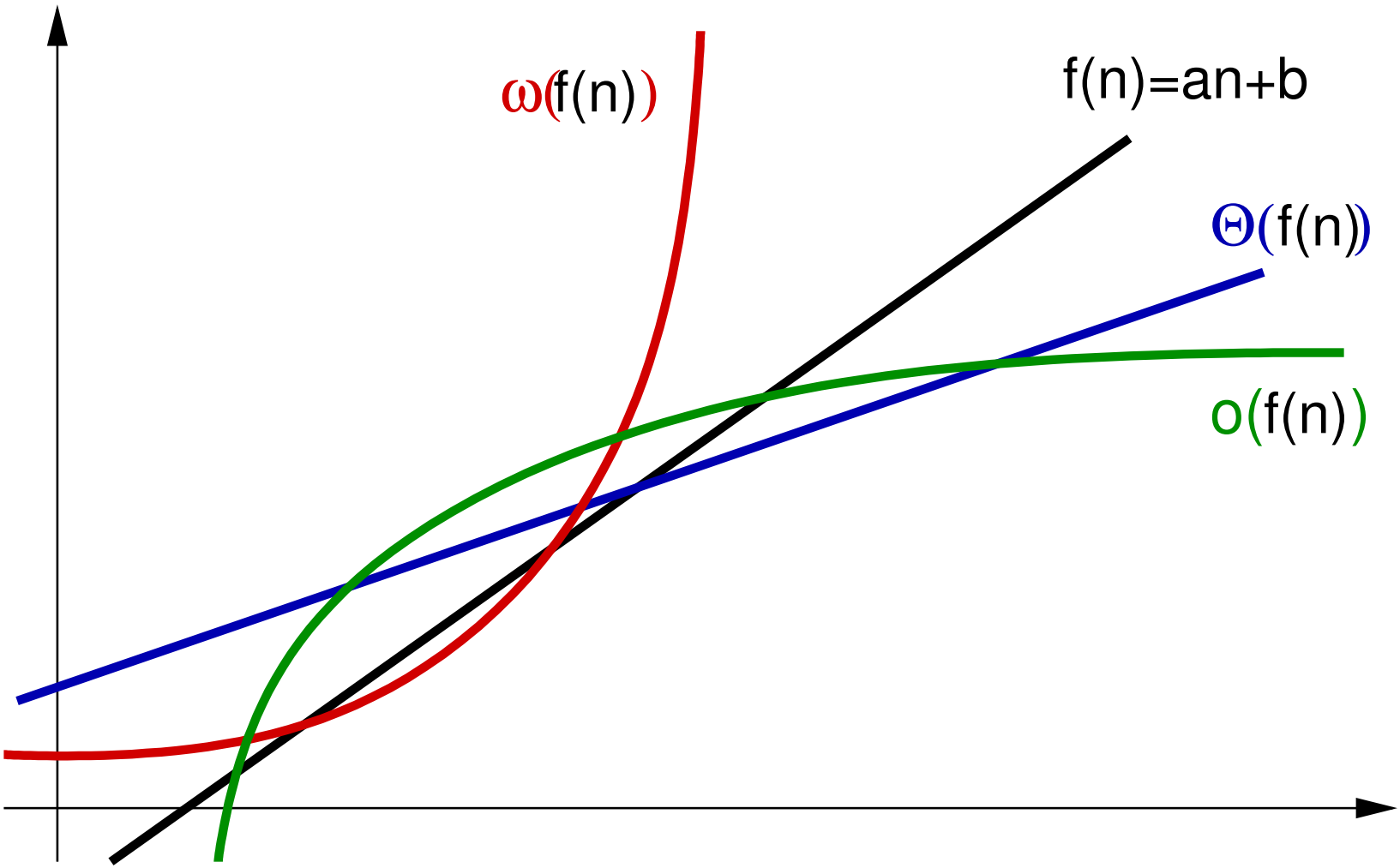
$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$
$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$
$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 > 0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

Asymptotische Notation



Asymptotische Notation

Beispiel

- $5n^2 - 7n \in \mathcal{O}(n^2), \quad n^2/10 + 100n \in \mathcal{O}(n^2), \quad 4n^2 \in \mathcal{O}(n^3)$
- $5n^2 - 7n \in \Omega(n^2), \quad n^3 \in \Omega(n^2), \quad n \log n \in \Omega(n)$
- $5n^2 - 7n \in \Theta(n^2)$
- $\log n \in o(n), \quad n^3 \in o(2^n)$
- $n^5 \in \omega(n^3), \quad 2^{2n} \in \omega(2^n)$

Asymptotische Notation als Platzhalter

- statt $g(n) \in \mathcal{O}(f(n))$ schreibt man auch oft $g(n) = \mathcal{O}(f(n))$
- für $f(n) + g(n)$ mit $g(n) \in o(h(n))$ schreibt man auch $f(n) + g(n) = f(n) + o(h(n))$
- statt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$ schreibt man auch $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$

Beispiel

$$n^3 + n \quad = \quad n^3 + o(n^3) \quad = \quad (1 + o(1))n^3 \quad = \quad \mathcal{O}(n^3)$$

$\mathcal{O}$-Notations"gleichungen" sollten nur von links nach rechts gelesen werden!

Übersicht

Wachstum von Polynomen

Lemma

Sei p ein Polynom der Ordnung k bzgl. der Variable n, also

$$p(n) = \sum_{i=0}^{k} a_i \cdot n^i \quad \text{mit} \quad a_k > 0.$$

Dann ist

$$p(n) \in \Theta(n^k).$$

Wachstum von Polynomen

Beweis.

Zu zeigen: $p(n) \in \mathcal{O}(n^k)$ und $p(n) \in \Omega(n^k)$

$p(n) \in \mathcal{O}(n^k)$:

Für $n \geq 1$ gilt:

$$p(n) \quad \leq \quad \sum_{i=0}^{k} |a_i| \cdot n^i \quad \leq \quad n^k \sum_{i=0}^{k} |a_i|$$

Also ist die Definition

$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

mit $c = \sum_{i=0}^{k} |a_i|$ und $n_0 = 1$ erfüllt.

Wachstum von Polynomen

Beweis.

$p(n) \in \Omega(n^k)$:

$$A = \sum_{i=0}^{k-1} |a_i|$$

Für positive n gilt dann:

$$p(n) \quad \geq \quad a_k n^k - A n^{k-1} \quad = \quad \frac{a_k}{2} n^k + n^{k-1}\left(\frac{a_k}{2} n - A\right)$$

Also ist die Definition

$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

mit $c = a_k/2$ und $n_0 > 2A/a_k$ erfüllt. □

Rechenregeln für $\mathcal{O}$-Notation

Für Funktionen $f(n)$ (bzw. $g(n)$) mit der Eigenschaft $\exists n_0 > 0 : \ \forall n \geq n_0 : \ f(n) > 0$ gilt:

Lemma

- $c \cdot f(n) \in \Theta(f(n))$ *für jede Konstante* $c > 0$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Die Ausdrücke sind auch korrekt für Ω *statt* $\mathcal{O}$.

Vorsicht, der letzte heißt dann

- $\Omega(f(n) + g(n)) = \Omega(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Aber: Vorsicht bei induktiver Anwendung!

Induktions"beweis"

Behauptung:

$$\sum_{i=1}^{n} i = \mathcal{O}(n)$$

"Beweis": Sei $f(n) = n + f(n-1)$ und $f(1) = 1$.

Ind.anfang: $f(1) = \mathcal{O}(1)$

Ind.vor.: Es gelte $f(n-1) = \mathcal{O}(n-1)$

Ind.schritt: Dann gilt

$$f(n) = n + f(n-1) = n + \mathcal{O}(n-1) = \mathcal{O}(n)$$

Also ist

$$f(n) = \sum_{i=1}^{n} i = \mathcal{O}(n)$$

FALSCH!

Rechenregeln für $\mathcal{O}$-Notation

Lemma

Seien f und g differenzierbar.

Dann gilt

- *falls $f'(n) \in \mathcal{O}(g'(n))$, dann auch $f(n) \in \mathcal{O}(g(n))$*
- *falls $f'(n) \in \Omega(g'(n))$, dann auch $f(n) \in \Omega(g(n))$*
- *falls $f'(n) \in o(g'(n))$, dann auch $f(n) \in o(g(n))$*
- *falls $f'(n) \in \omega(g'(n))$, dann auch $f(n) \in \omega(g(n))$*

Umgekehrt gilt das im Allgemeinen nicht!

Rechenbeispiele für $\mathcal{O}$-Notation

Beispiel

- 1. Lemma:
  - $n^3 - 3n^2 + 2n \in \mathcal{O}(n^3)$
  - $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2/2 + n/2) = \mathcal{O}(n^2)$
- 2. Lemma:
  Aus $\log n \in \mathcal{O}(n)$ folgt $n \log n \in \mathcal{O}(n^2)$.
- 3. Lemma:
  - $(\log n)' = 1/n$, $(n)' = 1$ und $1/n \in \mathcal{O}(1)$.
    $\Rightarrow \log n \in \mathcal{O}(n)$

Übersicht

von Neumann / Princeton-Architektur

John von Neumann
Quelle: http://wikipedia.org/

1945 János / Johann / John von Neumann:
Entwurf eines Rechnermodells: EDVAC
(Electronic Discrete Variable Automatic Computer)

Programm und Daten teilen sich
einen gemeinsamen Speicher

Besteht aus

- Arithmetic Logic Unit (ALU):
  Rechenwerk / Prozessor
- Control Unit: Steuerwerk (Befehlsinterpreter)
- Memory: eindimensional adressierbarer Speicher
- I/O Unit: Ein-/Ausgabewerk

Random Access Machine (RAM)

Was ist eigentlich ein Rechenschritt?

$\Rightarrow$ Abstraktion mit Hilfe von Rechnermodellen

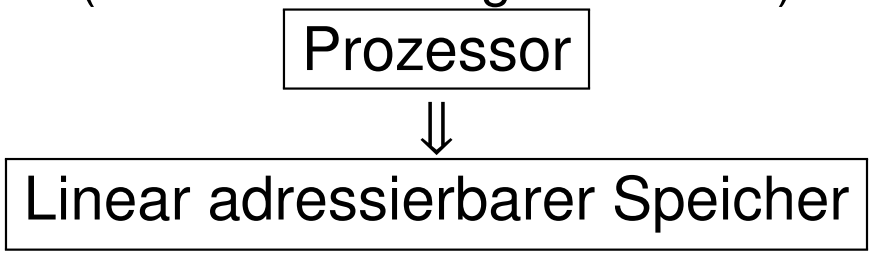
Bsp. Turingmaschine:

kann aber nicht auf beliebige Speicherzellen zugreifen,
nur an der aktuellen Position des Lese-/Schreibkopfs

1963 John Shepherdson, Howard Sturgis (u.a.):

Random Access Machine (RAM)

(beschränkte Registeranzahl)



(unbeschränkt)

RAM: Speicher

- Unbeschränkt viele Speicherzellen (words) $S[0], S[1], S[2], \ldots$, von denen zu jedem Zeitpunkt nur endlich viele benutzt werden
- aber: Speicherung beliebig großer Zahlen würde zu unrealistischen Algorithmen führen
- Jede Speicherzelle kann bei Eingabegröße n eine Zahl mit $\mathcal{O}(\log n)$ Bits speichern (angemessen: für konstant große Zellen würde man nur einen Faktor $\mathcal{O}(\log n)$ bei der Rechenzeit erhalten).

$\Rightarrow$ Gespeicherte Werte stellen polynomiell in n (Eingabegröße) beschränkte Zahlen dar.

- sinnvoll für Speicherung von Array-Indizes

Begrenzter Parallelismus:

- sequentielles Maschinenmodell, aber
- Verknüpfung logarithmisch vieler Bits in konstanter Zeit

RAM: Aufbau

Prozessor:

- Beschränkte Anzahl an Registern $R_1, \ldots, R_k$
- Instruktionszeiger zum nächsten Befehl

Programm:

- Nummerierte Liste von Befehlen
  (Adressen in Sprungbefehlen entsprechen dieser Nummerierung)

Eingabe:

- steht in Speicherzellen $S[1], \ldots, S[R_1]$

Modell / Reale Rechner:

- unendlicher / endlicher Speicher
- Abhängigkeit / Unabhängigkeit der Größe der Speicherzellen von der Eingabegröße

RAM: Befehle

Annahme:

- Jeder Befehl dauert genau eine Zeiteinheit.
- Laufzeit ist Anzahl ausgeführter Befehle

Befehlssatz:

- Registerzuweisung:

  $R_i := c$ Registerzuweisung für eine Konstante c

  $R_i := R_j$ Zuweisung von einem Register an ein anderes

- Speicherzugriff: (alle Zugriffe benötigen gleich viel Zeit)

  $R_i := S[R_j]$ lesend: lädt Inhalt von $S[R_j]$ in R_i

  $S[R_j] := R_i$ schreibend: speichert Inhalt von R_i in $S[R_j]$

RAM: Befehle

- Arithmetische / logische Operationen:

  $R_i := R_j \,\mathrm{op}\, R_k$ binäre Rechenoperation
  $\mathrm{op} \in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, \ldots\}$ oder
  $\mathrm{op} \in \{<, \leq, =, \geq, >\}$: 1$\hat{=}$wahr, 0$\hat{=}$falsch

  $R_i := \mathrm{op}\, R_j$ unäre Rechenoperation
  $\mathrm{op} \in \{-, \neg\}$

- Sprünge:

  jump x — Springe zu Position x

  jumpz x R_i — Falls $R_i = 0$, dann springe zu Position x

  jumpi R_j — Springe zur Adresse aus R_j

Das entspricht Assembler-Code von realen Maschinen!

Maschinenmodell

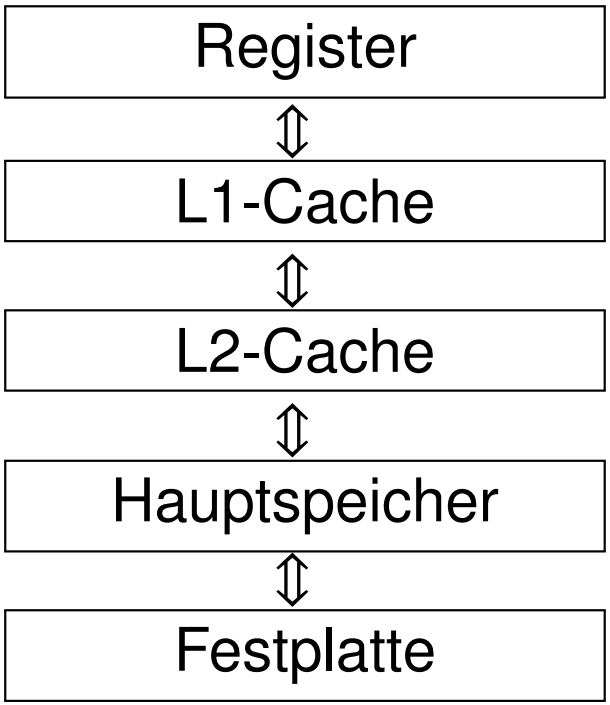
RAM-Modell

- Modell für die ersten Computer
- entspricht eigentlich der Harvard-Architektur (separater Programmspeicher)
- Sein Äquivalent zur Universellen Turing-Maschine, das Random Access Stored Program (RASP) Modell entspricht der von Neumann-Architektur und hat große Ähnlichkeit mit üblichen Rechnern.

Aber: Speicherhierarchien und Multicore-Prozessoren erfordern Anpassung des RAM-Modells

$\Rightarrow$ Algorithm Engineering, z.B. External-Memory Model

Speicherhierarchie



schneller, kleiner

langsamer, größer

$\Rightarrow$ External-Memory Model

- begrenzter schneller Speicher mit M Zellen
- unbegrenzter (langsamer) externer Speicher
- I/O-Operationen transferieren B aufeinanderfolgende Worte

Pseudocode

Maschinencode / Assembler umständlich
$\Rightarrow$ besser: Programmiersprache wie Pascal, C++, Java, ...

Variablendeklarationen:

T v;	Variable v vom Typ T
T v=x;	initialisiert mit Wert x

Variablentypen:

- int, boolean, char, double, ...
- Klassen T, Interfaces I
- T[n]: Feld von Elementen vom Typ T
  (indexiert von 0 bis $n-1$)

Pseudocode-Programme

Allokation und Deallokation von Speicherobjekten:

- v = new T($v_1, \ldots, v_k$); // implizit wird Konstruktor für T aufgerufen

Sprachkonstrukte: (C: Bedingung, I,J: Anweisungen)

- $v = A$; // Ergebnis von Ausdruck A wird an Variable v zugewiesen
- **if** (C) I **else** J
- **do** I **while** (C); **while**(C) I
- **for**($v = a$; $v < e$; v++) I
- **return** v;

Übersetzung in RAM-Befehle

- Zuweisungen lassen sich in konstant viele RAM-Befehle übersetzen, z.B.
  $a := a + bc$

  $R_1 := R_b * R_C;$
  $R_a := R_a + R_1$

  - R_a, R_b, R_c: Register, in denen a, b und c gespeichert sind

- **if** (C) I **else** J

  eval(C); jumpz sElse R_c; trans(I); jump sEnd; trans(J)

  - eval(C): Befehle, die die Bedingung C auswerten und das Ergebnis in Register R_c hinterlassen
  - trans(I), trans(J): übersetzte Befehlsfolge für I und J
  - sElse: Adresse des 1. Befehls in trans(J)
    sEnd: Adresse des 1. Befehls nach trans(J)

Übersetzung in RAM-Befehle

- **repeat** I **until** C

  trans(I); eval(C); jumpz sI R_c

  - trans(I): übersetzte Befehlsfolge für I
  - eval(C): Befehle, die die Bedingung C auswerten und das Ergebnis in Register R_c hinterlassen
  - sI: Adresse des ersten Befehls in trans(I)

- andere Schleifen lassen sich damit simulieren:

  while(C) I $\longrightarrow$ **if** C **then repeat** I **until** $\neg$C

  for($i = a$; $i \leq b$; i++) I $\longrightarrow$ $i := a$; **while**($i \leq b$) I; i++

Übersicht

Laufzeitanalyse

Was wissen wir?

- $\mathcal{O}$-Kalkül
  ($\mathcal{O}(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$, ...)
- RAM-Modell
  (load, store, jump, ...)
- Pseudocode
  (if-else, while, new, ...)

Wie analysieren wir damit Programme?

worst case

Berechnung der worst-case-Laufzeit:

- $T(I)$ sei worst-case-Laufzeit für Konstrukt I
- $T(\text{elementare Zuweisung}) = \mathcal{O}(1)$
- $T(\text{elementarer Vergleich}) = \mathcal{O}(1)$
- $T(\textbf{return } x) = \mathcal{O}(1)$
- $T(\textbf{new } \text{Typ}(\ldots)) = \mathcal{O}(1) + \mathcal{O}(T(\text{Konstruktor}))$
- $T(I_1; I_2) = T(I_1) + T(I_2)$
- $T(\textbf{if } (C)\ I_1\ \textbf{else}\ I_2) = \mathcal{O}(T(C) + \max\{T(I_1), T(I_2)\})$
- $T(\textbf{for}(i = a; i < b; i++)\ I) = \mathcal{O}\left(\sum_{i=a}^{b-1}(1 + T(I))\right)$
- $T(e.m(\ldots)) = \mathcal{O}(1) + T(ss)$, wobei ss Rumpf von m

Beispiel: Vorzeichenausgabe

`signum`(x)

Input : Zahl $x \in \mathbb{R}$
Output : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

```
if x < 0 then
  return -1
if x > 0 then
  return 1
return 0
```

Wir wissen:

$T(x < 0) = \mathcal{O}(1)$
$T(\textbf{return } \textit{-1}) = \mathcal{O}(1)$
$T(\textbf{if } (C)\ I) = \mathcal{O}(T(C) + T(I))$

Also: $T(\textbf{if } (x < 0) \textbf{ return } -1) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$

Beispiel: Vorzeichenausgabe

`signum`(x)

Input : Zahl $x \in \mathbb{R}$
Output : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

if $x < 0$ **then**	$\mathcal{O}(1)$
⌊ **return** *-1*	
if $x > 0$ **then**	$\mathcal{O}(1)$
⌊ **return** *1*	
return *0*	$\mathcal{O}(1)$

$$\mathcal{O}(1 + 1 + 1) = \mathcal{O}(1)$$

Beispiel: Minimumsuche

`minimum`(A, n)

Input : Zahlenfolge in $A[0], \ldots, A[n-1]$
n: Anzahl der Zahlen
Output : Minimum der Zahlen

min $= A[0]$;	$\mathcal{O}(1)$
for $(i = 1; i < n; i{+}{+})$ **do**	$\mathcal{O}(\sum_{i=1}^{n-1}(1 + T(I)))$
⌊ **if** $A[i] <$ min **then** min $= A[i]$	$\mathcal{O}(1)$ ↗
return min	$\mathcal{O}(1)$

$$\mathcal{O}\left(1 + \left(\sum_{i=1}^{n-1} 1\right) + 1\right) = \mathcal{O}(n)$$

Beispiel: BubbleSort

Sortieren durch Aufsteigen

Vertausche in jeder Runde in der (verbleibenden) Eingabesequenz (hier vom Ende in Richtung Anfang) jeweils zwei benachbarte Elemente, die nicht in der richtigen Reihenfolge stehen

Beispiel

5	10	19	1	14	3
5	10	19	1	3	14
5	10	1	19	3	14
5	1	10	19	3	14
1	5	10	19	3	14

1	5	10	3	19	14
1	5	3	10	19	14
1	**3**	5	10	19	14
1	3	**5**	10	14	19
1	3	5	**10**	**14**	**19**

Beispiel: Sortieren

`BubbleSort`(A, n)

Input : n: Anzahl der Zahlen
$A[0], \ldots, A[n-1]$: Zahlenfolge
Output : Sortierte Zahlenfolge A

for $(i = 0; i < n-1; i{+}{+})$ **do**	$O(\sum_{i=0}^{n-2} T(l_1))$
for $(j = n-2; j \geq i; j{-}{-})$ **do**	$O(\sum_{j=i}^{n-2} T(l_2))$
if $A[j] > A[j+1]$ **then**	$O(1 + T(l_3))$
$x = A[j];$	$O(1)$
$A[j] = A[j+1];$	$O(1)$
$A[j+1] = x;$	$O(1)$

$$O\left(\sum_{i=0}^{n-2}\sum_{j=i}^{n-2} 1\right)$$

Beispiel: Sortieren

$$
\begin{aligned}
\sum_{i=0}^{n-2} \sum_{j=i}^{n-2} 1 &= \sum_{i=0}^{n-2} (n-i-1) \\
&= \sum_{i=1}^{n-1} i \\
&= \frac{n(n-1)}{2} \\
&= \frac{n^2}{2} - \frac{n}{2} \\
&= O\left(n^2\right)
\end{aligned}
$$

Beispiel: Binäre Suche

`BinarySearch`(A, n, x)

Input : n: Anzahl der (sortierten) Zahlen
$A[0], \ldots, A[n-1]$: Zahlenfolge
x: gesuchte Zahl

Output : Index der gesuchten Zahl

$\ell = 0;$	$\mathcal{O}(1)$
$r = n - 1;$	$\mathcal{O}(1)$
while $(\ell \leq r)$ **do**	$\mathcal{O}(\sum_{i=1}^{k} T(l))$
$\quad m = \lfloor (r + \ell)/2 \rfloor;$	$\quad \mathcal{O}(1)$ $\uparrow$
$\quad$ **if** $A[m] == x$ **then return** $m;$	$\quad \mathcal{O}(1)$
$\quad$ **if** $A[m] < x$ **then** $\ell = m + 1;$	$\quad \mathcal{O}(1)$
$\quad$ **else** $r = m - 1;$	$\quad \mathcal{O}(1)$
return -1	$\mathcal{O}(1)$

$$\mathcal{O}\left(\sum_{i=1}^{k} 1\right) = \mathcal{O}(k)$$

Beispiel: Binäre Suche

Aber: Wie groß ist die Anzahl der Schleifendurchläufe k?

Größe des verbliebenen Suchintervalls $(r - \ell + 1)$ nach Iteration i:

$$\begin{aligned} s_0 &= n \\ s_{i+1} &\leq \lfloor s_i/2 \rfloor \end{aligned}$$

Bei $s_i < 1$ endet der Algorithmus.

$$\Rightarrow \quad k \leq \log_2 n$$

Gesamtkomplexität: $\mathcal{O}(\log n)$

Beispiel: Bresenham-Algorithmus

Bresenham1

$x = 0; \quad y = R;$	$\mathcal{O}(1)$
plot$(0, R)$; plot$(R, 0)$; plot$(0, -R)$; plot$(-R, 0)$;	$\mathcal{O}(1)$
$F = \frac{5}{4} - R;$	$\mathcal{O}(1)$
while $x < y$ **do**	$\mathcal{O}(\sum_{i=1}^{k} T(I))$
if $F < 0$ **then**	
$F = F + 2 * x + 1;$	
else	
$F = F + 2 * x - 2 * y + 2;$	alles $\mathcal{O}(1)$
$y = y - 1;$	
$x = x + 1;$	
plot(x, y); plot$(-x, y)$; plot$(-y, x)$; plot$(-y, -x)$;	
plot(y, x); plot$(y, -x)$; plot$(x, -y)$; plot$(-x, -y)$;	

Wie groß ist Anzahl Schleifendurchläufe k? $\mathcal{O}\left(\sum_{i=1}^{k} 1\right) = \mathcal{O}(k)$

Beispiel: Bresenham-Algorithmus

- Betrachte dazu die Entwicklung der Werte der Funktion

$$\varphi(x, y) = y - x$$

- Anfangswert: $\varphi_0(x, y) = R$
- Monotonie: verringert sich pro Durchlauf um mindestens 1
- Beschränkung: durch die **while**-Bedingung $x < y$ bzw. $0 < y - x$

$\Rightarrow$ maximal R Runden

Beispiel: Fakultätsfunktion

fakultaet(n)

Input : $n \in \mathbb{N}_+$
Output : $n!$

if $(n == 1)$ **then**	$\mathcal{O}(1)$
⌊ **return** 1	$\mathcal{O}(1)$
else	
⌊ **return** $n * fakultaet(n-1)$	$\mathcal{O}(1 + \ldots?)$

- $T(n)$: Laufzeit von fakultaet(n)
- $T(1) = \mathcal{O}(1)$
- $T(n) = T(n-1) + \mathcal{O}(1)$

$\Rightarrow$ $T(n) = \mathcal{O}(n)$

Übersicht

Average Case Complexity

Uniforme Verteilung:
(alle Instanzen gleichwahrscheinlich)

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$

Tatsächliche Eingabeverteilung kann in der Praxis aber stark von uniformer Verteilung abweichen.

Dann

$$t(n) = \sum_{i \in \mathcal{I}_n} p_i \cdot T(i)$$

Aber: meist schwierig zu berechnen!

Beispiel: Binärzahl-Inkrementierung

increment(A)

Input : Array A mit Binärzahl in $A[0] \ldots A[n-1]$,
in $A[n]$ steht eine 0

Output : inkrementierte Binärzahl in $A[0] \ldots A[n]$

```
i = 0;
while (A[i] == 1) do
    A[i] = 0;
    i = i + 1;
A[i] = 1;
```

Durchschnittliche Laufzeit für Zahl mit n Bits?

Beispiel: Binärzahl-Inkrementierung

Analyse:

- Sei $\mathcal{I}_n$ die Menge der n-Bit-Instanzen.
- Für die Hälfte ($1/2$) der Zahlen $x_{n-1} \dots x_0 \in \mathcal{I}_n$ ist $x_0 = 0$

$\Rightarrow$ 1 Schleifendurchlauf

- Für die andere Hälfte gilt $x_0 = 1$.
  Bei diesen gilt wieder für die Hälfte (also insgesamt $1/4$) der Zahlen $x_1 x_0 = 01$

$\Rightarrow$ 2 Schleifendurchläufe

- Für den Anteil $(1/2)^k$ der Zahlen gilt $x_{k-1} x_{k-2} \dots x_0 = 01 \dots 1$

$\Rightarrow$ k Schleifendurchläufe

Beispiel: Binärzahl-Inkrementierung

Durchschnittliche Laufzeit:

$$
\begin{aligned}
t(n) &= \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i) \\
&= \frac{1}{|\mathcal{I}_n|} \sum_{k=1}^{n} \frac{|\mathcal{I}_n|}{2^k} \cdot O(k) \\
&= \sum_{k=1}^{n} O\left(\frac{k}{2^k}\right) \\
&= O\left(\sum_{k=1}^{n} \frac{k}{2^k}\right) \stackrel{?}{=} O(1)
\end{aligned}
$$

Beispiel: Binärzahl-Inkrementierung

Lemma

$$\sum_{k=1}^{n} \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$$

Beweis

Induktionsanfang:

Für $n = 1$ gilt: $\sum_{k=1}^{1} \frac{k}{2^k} = \frac{1}{2} \leq 2 - \frac{1+2}{2^1}$ ✓

Induktionsvoraussetzung:

Für n gilt: $\sum_{k=1}^{n} \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$

Beispiel: Binärzahl-Inkrementierung

Beweis.

Induktionsschritt: $n \to n+1$

$$\begin{aligned}
\sum_{k=1}^{n+1} \frac{k}{2^k} &= \left(\sum_{k=1}^{n} \frac{k}{2^k}\right) + \frac{n+1}{2^{n+1}} \\
&\leq 2 - \frac{n+2}{2^n} + \frac{n+1}{2^{n+1}} \qquad \text{(laut Ind.vor.)} \\
&= 2 - \frac{2(n+2)}{2^{n+1}} + \frac{n+1}{2^{n+1}} = 2 - \frac{2n+4-n-1}{2^{n+1}} \\
&= 2 - \frac{n+3}{2^{n+1}} \\
&= 2 - \frac{(n+1)+2}{2^{n+1}}
\end{aligned}$$

□

Übersicht

Zufallsvariable

Definition

Für einen Wahrscheinlichkeitsraum mit Ergebnismenge Ω nennt man eine Abbildung $X : \Omega \mapsto \mathbb{R}$ (numerische) Zufallsvariable.

Eine Zufallsvariable über einer endlichen oder abzählbar unendlichen Ergebnismenge heißt diskret.

Der Wertebereich diskreter Zufallsvariablen

$$W_X := X(\Omega) = \{x \in \mathbb{R} \mid \exists \omega \in \Omega \text{ mit } X(\omega) = x\}$$

ist ebenfalls endlich bzw. abzählbar unendlich.

Schreibweise: $\Pr[X = x] := \Pr\left[X^{-1}(x)\right] = \sum_{\omega \in \Omega \mid X(\omega) = x} \Pr[\omega]$

Zufallsvariable

Beispiel

Wir ziehen aus einem Poker-Kartenspiel mit 52 Karten (13 von jeder Farbe) eine Karte.

Wir bekommen bzw. bezahlen einen bestimmten Betrag, je nachdem welche Farbe die Karte hat, z.B. 4 Euro für Herz, 7 Euro für Karo, −5 Euro für Kreuz und −3 Euro für Pik.

Wenn wir ein As ziehen, bekommen wir zusätzlich 1 Euro.

$\Omega = \{\heartsuit A, \heartsuit K, \ldots, \heartsuit 2, \diamondsuit A, \diamondsuit K, \ldots, \diamondsuit 2, \clubsuit A, \clubsuit K, \ldots, \clubsuit 2, \spadesuit A, \spadesuit K, \ldots, \spadesuit 2\}$.

X sei der Geldbetrag den wir bekommen bzw. bezahlen.

$W_X = \{-5, -4, -3, -2, 4, 5, 7, 8\}$

$\Pr[X = -3] = \Pr[\spadesuit K] + \ldots + \Pr[\spadesuit 2] = 12/52 = 3/13$

Erwartungswert

Definition

Für eine diskrete Zufallsvariable X ist der Erwartungswert definiert als

$$\mathbb{E}[X] := \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega]$$

sofern $\sum_{x \in W_X} |x| \cdot \Pr[X = x]$ konvergiert (absolute Konvergenz).

Bei endlicher Ereignismenge und gleichwahrscheinlichen Ereignissen entspricht der Erwartungswert dem Durchschnitt:

$$\mathbb{E}[X] = \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \frac{1}{|\Omega|} = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} X(\omega)$$

Erwartungswert

Beispiel

(Beispiel wie zuvor)

$$\begin{aligned}\mathbb{E}[X] &= 4 \cdot \frac{12}{52} + 5 \cdot \frac{1}{52} + 7 \cdot \frac{12}{52} + 8 \cdot \frac{1}{52} \\ &\quad + (-5) \cdot \frac{12}{52} + (-4) \cdot \frac{1}{52} + (-3) \cdot \frac{12}{52} + (-2) \cdot \frac{1}{52} = \frac{43}{52}\end{aligned}$$

Wir bekommen also im Erwartungswert $\frac{43}{52}$ Euro pro gezogener Karte.

Grundlagen zu diskreter Wahrscheinlichkeitstheorie findet man z.B. in folgendem Buch:

Th. Schickinger, A. Steger
Diskrete Strukturen – Band 2
(Wahrscheinlichkeitstheorie und Statistik)
Springer-Verlag, 2001.

Erwartungswert

Beispiel

Münze werfen, bis sie zum ersten Mal Kopf zeigt

Zufallsvariable k: Anzahl der Versuche

k ungerade: Spieler bezahlt etwas an die Bank
k gerade: Spieler bekommt etwas von der Bank

Zufallsvariable X: Gewinnbetrag der Bank

Variante 1: Spieler bezahlt / bekommt k Euro
$\mathbb{E}[X]$ existiert (absolute Konvergenz)

Variante 2: Spieler bezahlt / bekommt 2^k Euro
$\mathbb{E}[X]$ existiert nicht (keine Konvergenz)

Variante 3: Spieler bezahlt / bekommt $\frac{2^k}{k}$ Euro
$\mathbb{E}[X]$ existiert nicht (Konvergenz, aber keine absolute)

Erwartungswert zusammengesetzter Zufallsvariablen

Satz (Linearität des Erwartungswerts)

Für Zufallsvariablen $X_1, \ldots, X_n$ und

$$X := a_1 X_1 + \ldots + a_n X_n$$

mit $a_1, \ldots, a_n \in \mathbb{R}$ gilt

$$\mathbb{E}[X] = a_1 \mathbb{E}[X_1] + \ldots + a_n \mathbb{E}[X_n].$$

Interessant ist für uns vor allem der einfache Fall:

$$X := X_1 + \ldots + X_n$$

mit

$$\mathbb{E}[X] = \mathbb{E}[X_1] + \ldots + \mathbb{E}[X_n].$$

Beispiel: Suche in statischer Liste

- gegeben: Liste mit Elementen $1, \ldots, m$
- search(i): lineare Suche nach Element i ab Listenanfang

s_i Position von Element i in der Liste (1$\hat{=}$Anfang)

p_i Wahrscheinlichkeit für Zugriff auf Element i

Erwartete Laufzeit der Operation search(i) mit zufälligem i:

$$\mathbb{E}[T(\text{search}(i))] = O\left(\sum_i p_i s_i\right)$$

Erwartete Laufzeit $t(n)$ für n Zugriffe bei statischer Liste:

$$t(n) = \mathbb{E}[T(n \times \text{search}(i))] = n \cdot \mathbb{E}[T(\text{search}(i))] = O\left(n \sum_i p_i s_i\right)$$

Beispiel: Suche in statischer Liste

Optimale Anordnung?

$\Rightarrow$ wenn für alle Elemente i, j mit $p_i > p_j$ gilt, dass $s_i < s_j$, d.h. die Elemente nach Zugriffswahrscheinlichkeit sortiert sind

o.B.d.A. seien die Indizes so, dass $p_1 \geq p_2 \geq \ldots \geq p_m$

- Optimale Anordnung: $s_i = i$
- Optimale erwartete Laufzeit: $\text{opt} = \sum_i p_i \cdot i$

Einfach: wenn die Zugriffswahrscheinlichkeiten bekannt sind
$\Rightarrow$ optimale erwartete Laufzeit durch absteigende Sortierung nach p_i

Problem: was wenn die Wahrscheinlichkeiten p_i unbekannt sind?

Beispiel: Suche in selbstorganisierender Liste

Move-to-Front Rule:

Verschiebe nach jeder erfolgreichen Suche das gefundene Element an den Listenanfang

Bsp.: Ausführung von search(4) ergibt

Beispiel: Suche in selbstorganisierender Liste

Erwartete Laufzeit $t(n)$ bei dynamischer Liste:

$$\mathbb{E}[T(\text{search}(i))] = O\left(\sum_i p_i \cdot \mathbb{E}[s_i]\right)$$

Satz

Ab dem Zeitpunkt, wo auf jedes Element mindestens einmal zugegriffen wurde, ist die erwartete Laufzeit der search-Operation unter Verwendung der Move-to-Front Rule höchstens $2 \cdot \text{opt}$.

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Betrachte zwei feste Elemente i und j

t_0 Zeitpunkt der letzten Suchoperation auf i oder j

- bedingte Wahrscheinlichkeit: $\Pr[A \mid B] = \dfrac{\Pr[A \wedge B]}{\Pr[B]}$
- $\Pr[C \mid (C \vee D)] = \dfrac{\Pr[C \wedge (C \vee D)]}{\Pr[C \vee D]} = \dfrac{\Pr[C]}{\Pr[C \vee D]}$
- $\Pr[\text{search}(j) \text{ bei } t_0 \mid \text{search}(i \vee j) \text{ bei } t_0] = \dfrac{p_j}{p_i + p_j}$
- mit Wsk. $\dfrac{p_i}{p_i + p_j}$ steht i vor j und mit Wsk. $\dfrac{p_j}{p_i + p_j}$ steht j vor i

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Betrachte nun nur ein festes Element i

- Definiere Zufallsvariablen $X_j \in \{0,1\}$ für $j \neq i$:

$$X_j = 1 \quad \Leftrightarrow \quad j \text{ vor } i \text{ in der Liste}$$

- Erwartungswert:

$$\begin{aligned} \mathbb{E}[X_j] &= 0 \cdot \Pr[X_j = 0] + 1 \cdot \Pr[X_j = 1] \\ &= \Pr[\text{letzte Suche nach } i \,/\, j \text{ war nach } j] \\ &= \frac{p_j}{p_i + p_j} \end{aligned}$$

Beispiel: Suche in selbstorganisierender Liste

Beweis.

- Listenposition von Element i: $1 + \sum_{j \neq i} X_j$
- Erwartungswert der Listenposition von Element i:

$$\begin{aligned} \mathbb{E}[s_i] &= \mathbb{E}\left[1 + \sum_{j \neq i} X_j\right] \\ &= 1 + \mathbb{E}\left[\sum_{j \neq i} X_j\right] = 1 + \sum_{j \neq i} \mathbb{E}\left[X_j\right] \\ \mathbb{E}[s_{i,\mathrm{MTF}}] &= 1 + \sum_{j \neq i} \frac{p_j}{p_i + p_j} \end{aligned}$$

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Erwartete Laufzeit der search-Operation:

$$
\begin{aligned}
\mathbb{E}[T_{\mathrm{MTF}}] &= \sum_i p_i \left(1 + \sum_{j \neq i} \frac{p_j}{p_i + p_j}\right) \\
&= \sum_i \left(p_i + \sum_{j \neq i} \frac{p_i p_j}{p_i + p_j}\right) = \sum_i \left(p_i + 2 \sum_{j<i} \frac{p_i p_j}{p_i + p_j}\right) \\
&= \sum_i p_i \left(1 + 2 \sum_{j<i} \frac{p_j}{p_i + p_j}\right) \leq \sum_i p_i \left(1 + 2 \sum_{j<i} 1\right) \\
&\leq \sum_i p_i \cdot (2i - 1) < \sum_i p_i \cdot 2i = 2 \cdot \mathrm{opt}
\end{aligned}
$$

□

Übersicht

Sequenzen

Sequenz: lineare Struktur

$$s = \langle e_0, \ldots, e_{n-1} \rangle$$

(Gegensatz: verzweigte Struktur in Graphen, fehlende Struktur in Hashtab.)

Klassische Repräsentation:

- (Statisches) Feld / Array:
  direkter Zugriff über $s[i]$
  - Vorteil: Zugriff über Index, homogen im Speicher
  - Nachteil: dynamische Größenänderung schwierig

- Liste:
  indirekter Zugriff über Nachfolger / Vorgänger
  - Vorteil: Einfügen / Löschen von Teilsequenzen
  - Nachteil: kein Zugriff per Index, Elemente über Speicher verteilt

Sequenzen

Operationen:

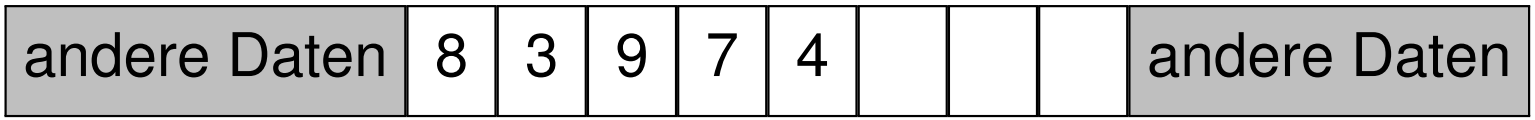
- $\langle e_0, \ldots, e_{n-1} \rangle [i]$ liefert Referenz auf e_i
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{get}(i) = e_i$
- $\langle e_0, \ldots, e_{i-1}, e_i, \ldots, e_{n-1} \rangle .\mathrm{set}(i, e) = \langle e_0, \ldots, e_{i-1}, e, \ldots, e_{n-1} \rangle$
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{pushBack}(e) = \langle e_0, \ldots, e_{n-1}, e \rangle$
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{popBack}() = \langle e_0, \ldots, e_{n-2} \rangle$
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{size}() = n$

Übersicht

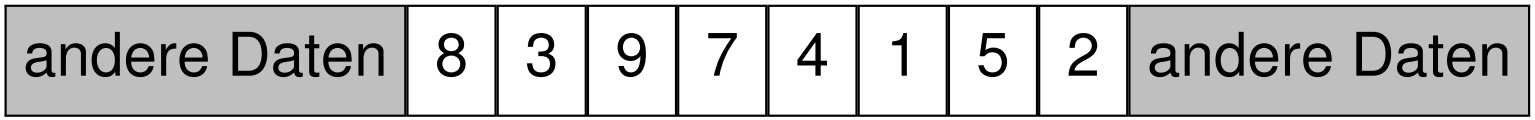
Sequenz als Feld

Problem: beschränkter Speicher

- Feld:



- pushBack(1), pushBack(5), pushBack(2):



- pushBack(6): voll!

Sequenz als Feld

Problem:

- Beim Anlegen des Felds ist nicht bekannt, wieviele Elemente es enthalten wird
- Nur Anlegen von statischen Feldern möglich
  ($s =$ new ElementTyp[w])

Lösung: Datenstruktur für dynamisches Feld

Dynamisches Feld

Erste Idee:

- Immer dann, wenn Feld s nicht mehr ausreicht:
  generiere neues Feld der Größe $w + c$ für ein festes c

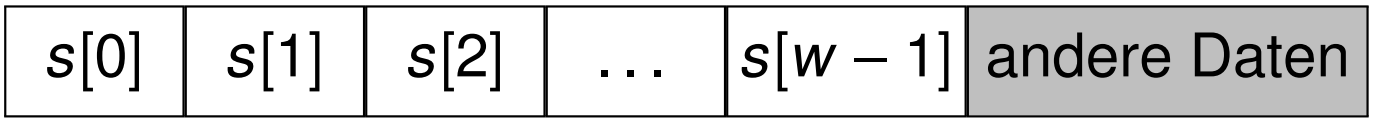
$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	andere Daten

$\Downarrow$ Kopieren in neues größeres Feld

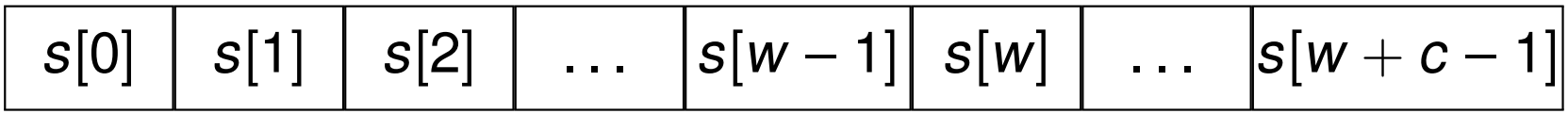
$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	$s[w]$	$\ldots$	$s[w+c-1]$

Dynamisches Feld

Zeitaufwand für Erweiterung: $\Theta(w)$



$\Downarrow$ Kopieren in neues größeres Feld



Zeitaufwand für n pushBack Operationen:

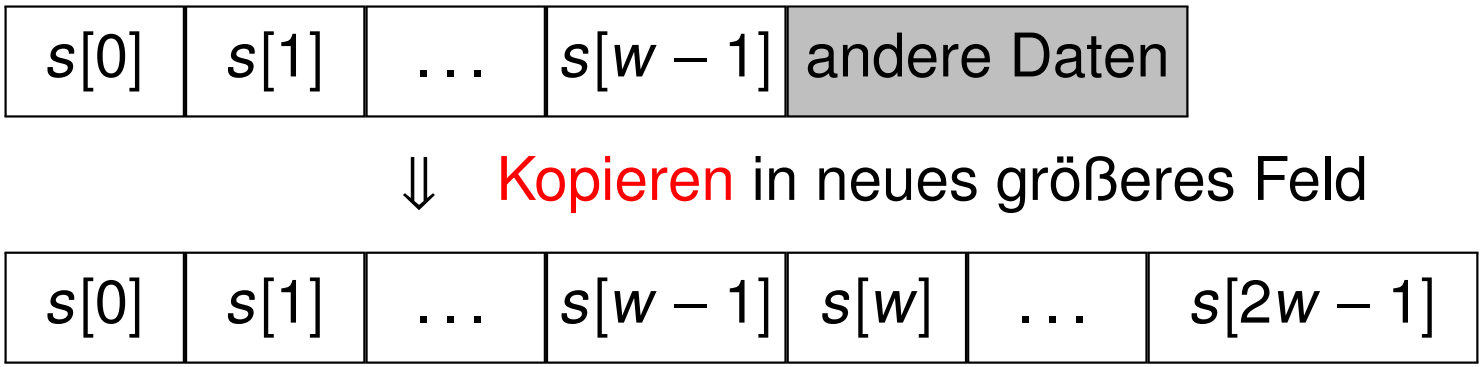
- Aufwand von $\Theta(w)$ nach jeweils c Operationen (wobei w immer größer wird)
- Gesamtaufwand:

$$\Theta\left(\sum_{i=1}^{n/c} c \cdot i\right) = \Theta\left(n^2\right)$$

Dynamisches Feld

Bessere Idee:

- Immer dann, wenn Feld s nicht mehr ausreicht:
  generiere neues Feld der doppelten Größe $2w$



- Immer dann, wenn Feld s zu groß ist ($n \leq w/4$):
  generiere neues Feld der halben Größe $w/2$

Dynamisches Feld

Implementierung

Klasse **UArray** mit den Methoden:

- ElementTyp **get**(int i)
- **set**(int i, ElementTyp& e)
- int **size**()
- void **pushBack**(ElementTyp e)
- void **popBack**()
- void **reallocate**(int new_w)

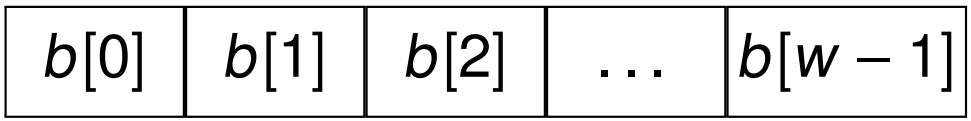
- ElementTyp& **[**int i**]**
  auch möglich, aber Referenz nur bis zur nächsten Größenänderung des Felds gültig

Dynamisches Feld

Implementierung

Klasse **UArray** mit den Elementen:

- $\beta = 2$ // Wachstumsfaktor
- $\alpha = 4$ // max. Speicheroverhead
- $\mathbf{w} = 1$ // momentane Feldgröße
- $\mathbf{n} = 0$ // momentane Elementanzahl
- $\mathbf{b} =$ new ElementTyp[w] // statisches Feld



Dynamisches Feld

Implementierung

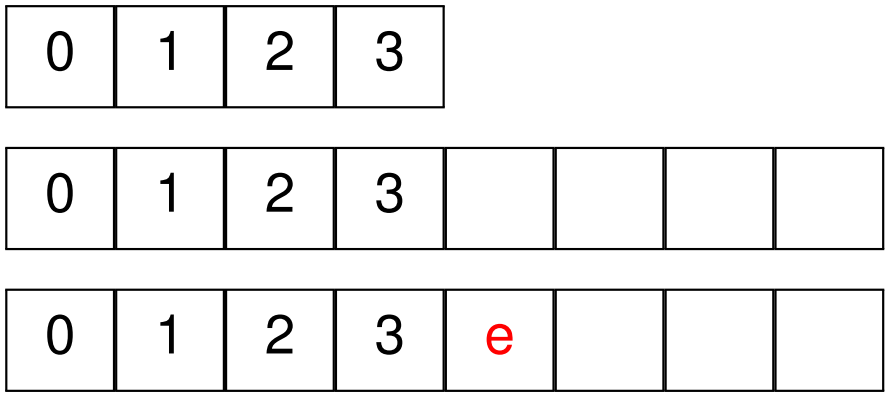
```
ElementTyp get(int i) {
    assert(0≤i<n);
    return b[i];
}

set(int i, ElementTyp& e) {
    assert(0≤i<n);
    b[i] = e;
}

int size() {
    return n;
}
```

Dynamisches Feld

Implementierung

```
void pushBack(ElementTyp e) {
    if (n==w)
        reallocate(β*n);
    b[n]=e;
    n++;
}
```

n=4, w=4



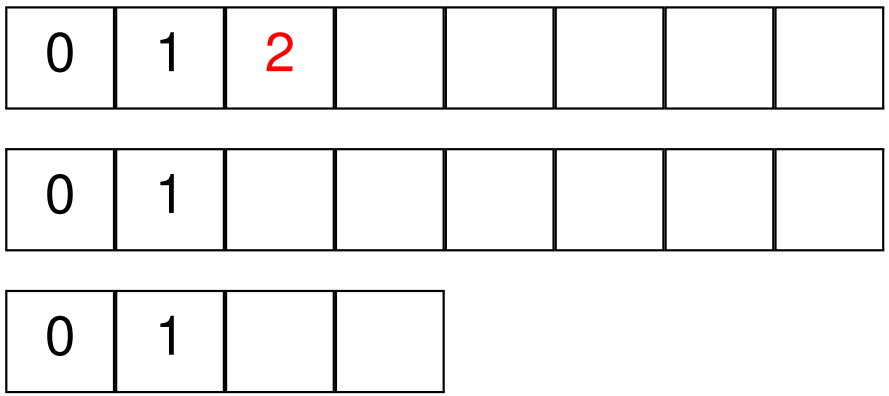
n=5, w=8

Dynamisches Feld

Implementierung

```
void popBack() {
    assert(n>0);
    n−−;
    if (α*n ≤ w ∧ n>0)
        reallocate(β*n);
}
```

n=3, w=8



n=2, w=4

Dynamisches Feld

Implementierung

```
void reallocate(int new_w) {
    w = new_w;
    ElementTyp[ ] new_b = new ElementTyp[new_w];
    for (i=0; i<n; i++)
        new_b[i] = b[i];
    b = new_b;
}
```

Dynamisches Feld

Wieviel Zeit kostet eine Folge von n pushBack-/popBack-Operationen?

Erste Idee:

- einzelne Operation kostet $\mathcal{O}(n)$
- Schranke kann nicht weiter gesenkt werden, denn reallocate-Aufrufe kosten jeweils $\Theta(n)$

$\Rightarrow$ also Gesamtkosten für n Operationen beschränkt durch $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$

Dynamisches Feld

Wieviel Zeit kostet eine Folge von n pushBack-/popBack-Operationen?

Zweite Idee:

- betrachtete Operationen sollen direkt aufeinander folgen
- zwischen Operationen mit reallocate-Aufruf gibt es immer auch welche ohne

$\Rightarrow$ vielleicht ergibt sich damit gar nicht die n-fache Laufzeit einer Einzeloperation

Lemma

Betrachte ein anfangs leeres dynamisches Feld s.

Jede Folge $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ von pushBack- und popBack-Operationen auf s kann in Zeit $O(n)$ bearbeitet werden.

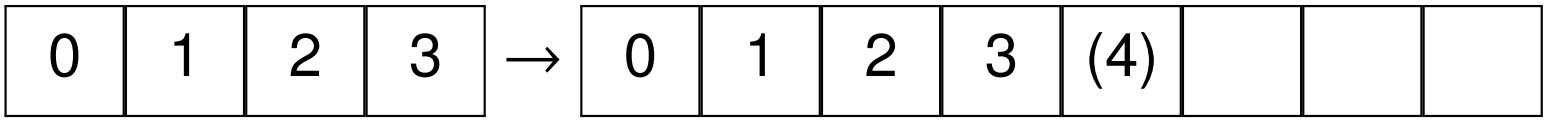
Dynamisches Feld

$\Rightarrow$ nur durchschnittlich konstante Laufzeit pro Operation

- Kosten teurer Operationen werden mit Kosten billiger Operationen verrechnet.
- Man nennt das dann amortisierte Kosten bzw. amortisierte Analyse.
- In diesem Beispiel hätten wir also eine amortisierte Laufzeit von $O(1)$ für die pushBack- und die popBack-Operation.

Dynamisches Feld: Analyse

- Feldverdopplung:



- Feldhalbierung:

- nächste Verdopplung: nach $\geq n$ pushBack-Operationen
- nächste Halbierung: nach $\geq n/2$ popBack-Operationen

Dynamisches Feld: Analyse

Formale Verrechnung: Zeugenzuordnung

- reallocate kann eine Vergrößerung oder Verkleinerung sein
- reallocate als Vergrößerung auf n Speicherelemente:
  es werden die $n/2$ vorangegangenen pushBack-Operationen zugeordnet
- reallocate als Verkleinerung auf n Speicherelemente:
  es werden die n vorangegangenen popBack-Operationen zugeordnet

$\Rightarrow$ kein pushBack / popBack wird mehr als einmal zugeordnet

Dynamisches Feld: Analyse

- Idee: verrechne reallocate-Kosten mit pushBack/popBack-Kosten (ohne reallocate)
  - Kosten für pushBack / popBack: $\mathcal{O}(1)$
  - Kosten für reallocate(k*n): $\mathcal{O}(n)$
- Konkret:
  - $\Theta(n)$ Zeugen pro reallocate(k$*$n)
  - verteile $\mathcal{O}(n)$-Aufwand gleichmäßig auf die Zeugen
- Gesamtaufwand: $\mathcal{O}(m)$ bei m Operationen

Dynamisches Feld: Analyse

Kontenmethode

- günstige Operationen zahlen Tokens ein
- teure Operationen entnehmen Tokens
- Tokenkonto darf nie negativ werden!

Dynamisches Feld: Analyse

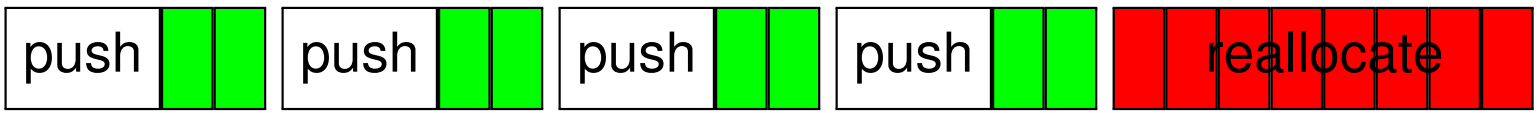
Kontenmethode

- günstige Operationen zahlen Tokens ein
  - pro pushBack 2 Tokens
  - pro popBack 1 Token
- teure Operationen entnehmen Tokens
  - pro reallocate(k∗n) −n Tokens
- Tokenkonto darf nie negativ werden!
  - Nachweis über Zeugenargument

Dynamisches Feld: Analyse

Tokenlaufzeit (Reale Kosten + Ein-/Auszahlungen)

- Ausführung von pushBack / popBack kostet 1 Token
  - Tokenkosten für pushBack: 1+2=3 Tokens
  - Tokenkosten für popBack: 1+1=2 Tokens
- Ausführung von reallocate(k∗n) kostet n Tokens
  - Tokenkosten für reallocate(k∗n): n−n=0 Tokens



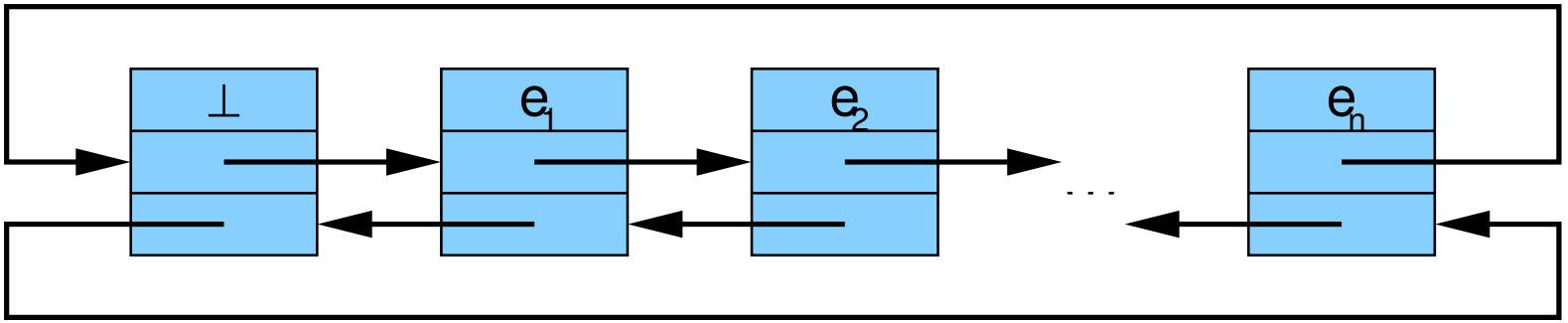
- Gesamtlaufzeit = $\mathcal{O}$(Summe der Tokenlaufzeiten)

Übersicht

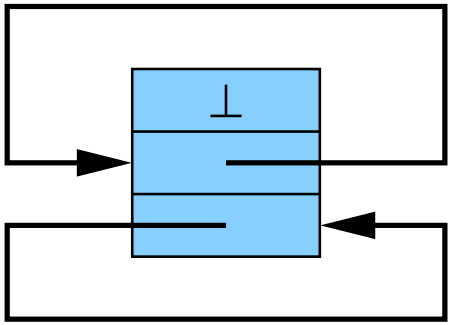
Doppelt verkettete Liste

Einfache Verwaltung:

durch Dummy-Element h ohne Inhalt ($\perp$):

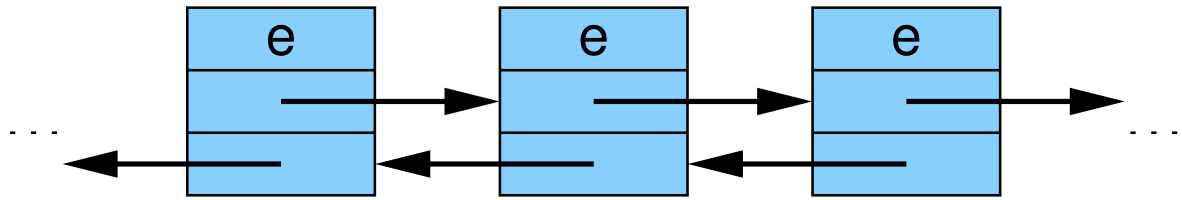


Anfangs:



Doppelt verkettete Liste

```
type Handle: pointer → Item<Elem>;

type Item<Elem> {
    Elem e;
    Handle next;
    Handle prev;
}

class List<Elem> {
    Item<Elem> h;       // initialisiert mit ⊥ und Zeigern auf sich selbst
    ...weitere Variablen und Methoden ...
}
```



Invariante:

next→prev == prev→next == this

Doppelt verkettete Liste

Zentrale statische Methode: splice(Handle a, Handle b, Handle t)

- Bedingung:
  - $\langle a, \ldots, b \rangle$ muss Teilsequenz sein (a=b erlaubt)
  - b nicht vor a (also Dummy h nicht zwischen a und b)
  - t nicht in Teilliste $\langle a, \ldots, b \rangle$, aber evt. in anderer Liste
- splice entfernt $\langle a, \ldots, b \rangle$ aus der Sequenz und fügt sie hinter Item t an

Für

$$\langle e_1, \ldots, a', a, \ldots, b, b', \ldots, t, t', \ldots, e_n \rangle$$

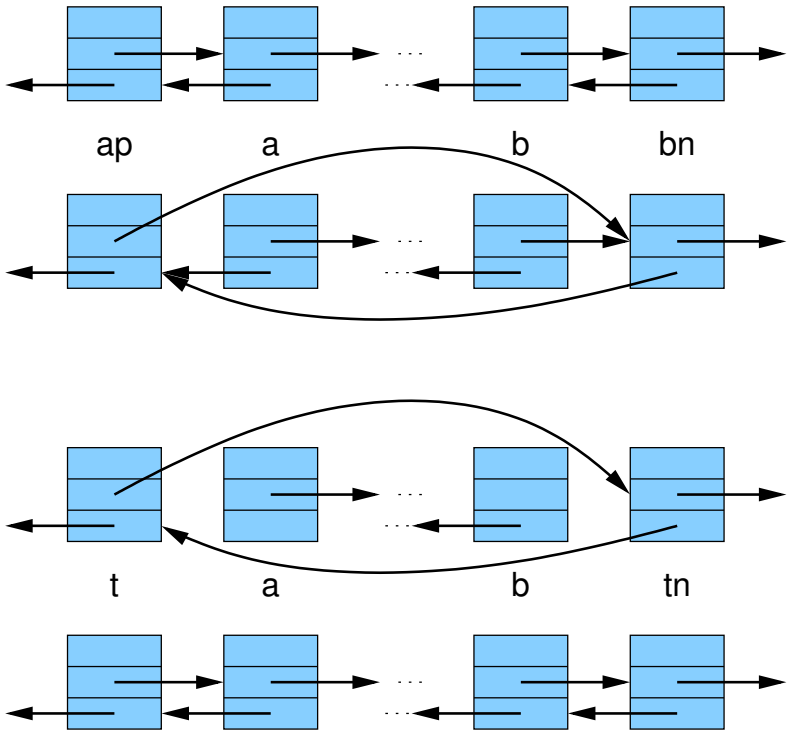
liefert splice(a,b,t)

$$\langle e_1, \ldots, a', b', \ldots, t, a, \ldots, b, t', \ldots, e_n \rangle$$

Doppelt verkettete Liste

Methoden

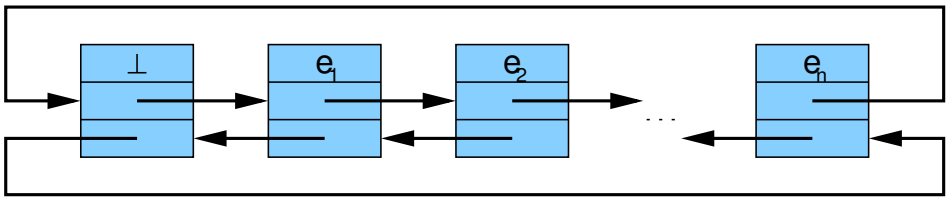
```
(static) splice(Handle a, b, t) {
    // schneide ⟨a,...,b⟩ heraus
    Handle ap = a→prev;
    Handle bn = b→next;
    ap→next = bn;
    bn→prev = ap;

    // füge ⟨a,...,b⟩ hinter t ein
    Handle tn = t→next;
    b→next = tn;
    a→prev = t;
    t→next = a;
    tn→prev = b;
}
```

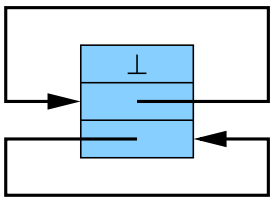


Doppelt verkettete Liste

Methoden

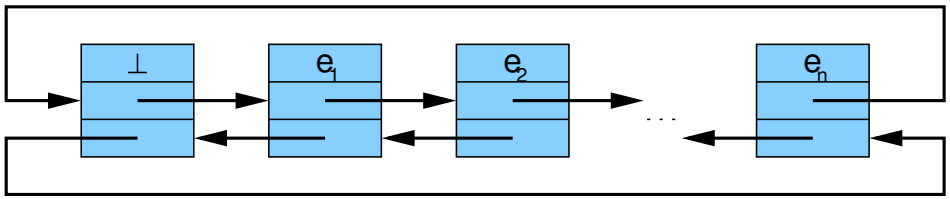
```
Handle head() {
    return address of h;
}

boolean isEmpty() {
    return (h.next == head());
}

Handle first() {
    return h.next;        // evt. → ⊥
}

Handle last() {
    return h.prev;        // evt. → ⊥
}
```





Doppelt verkettete Liste

Methoden




```
(static) moveAfter (Handle b, Handle a) {
   splice(b, b, a);       // schiebe b hinter a
}

moveToFront (Handle b) {
   moveAfter(b, head());       // schiebe b ganz nach vorn
}

moveToBack (Handle b) {
   moveAfter(b, last());       // schiebe b ganz nach hinten
}
```

Doppelt verkettete Liste

Methoden

Löschen und Einfügen von Elementen:

mittels separater Liste freeList
⇒ bessere Laufzeit (Speicherallokation teuer)

```
(static) remove(Handle b) {
    moveAfter(b, freeList.head());
}

popFront() {
    remove(first());
}

popBack() {
    remove(last());
}
```

Doppelt verkettete Liste

Methoden

```
(static) Handle insertAfter(Elem x, Handle a) {
    checkFreeList();       // u.U. Speicher allokieren
    Handle b = freeList.first();
    moveAfter(b, a);
    b→e = x;
    return b;
}

(static) Handle insertBefore(Elem x, Handle b) {
    return insertAfter(x, b→prev);
}

pushFront(Elem x) {    insertAfter(x, head());   }

pushBack(Elem x) {    insertAfter(x, last());   }
```

Doppelt verkettete Liste

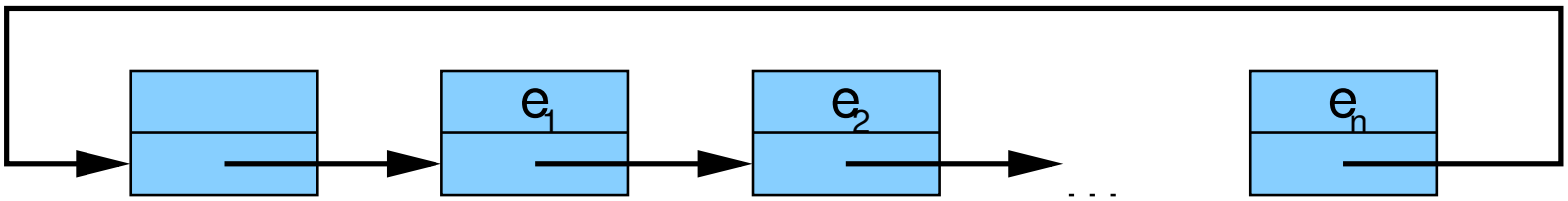
Manipulation ganzer Listen:

Trick: verwende Dummy-Element

```
Handle findNext(Elem x, Handle from) {
    h.e = x;
    while (from→e != x)
        from = from→next;
    [h.e = ⊥;]
    return from;
}
```

Einfach verkettete Liste

```
type SHandle: pointer → SItem<Elem>;

type SItem<Elem> {
    Elem e;
    SHandle next;
}
```

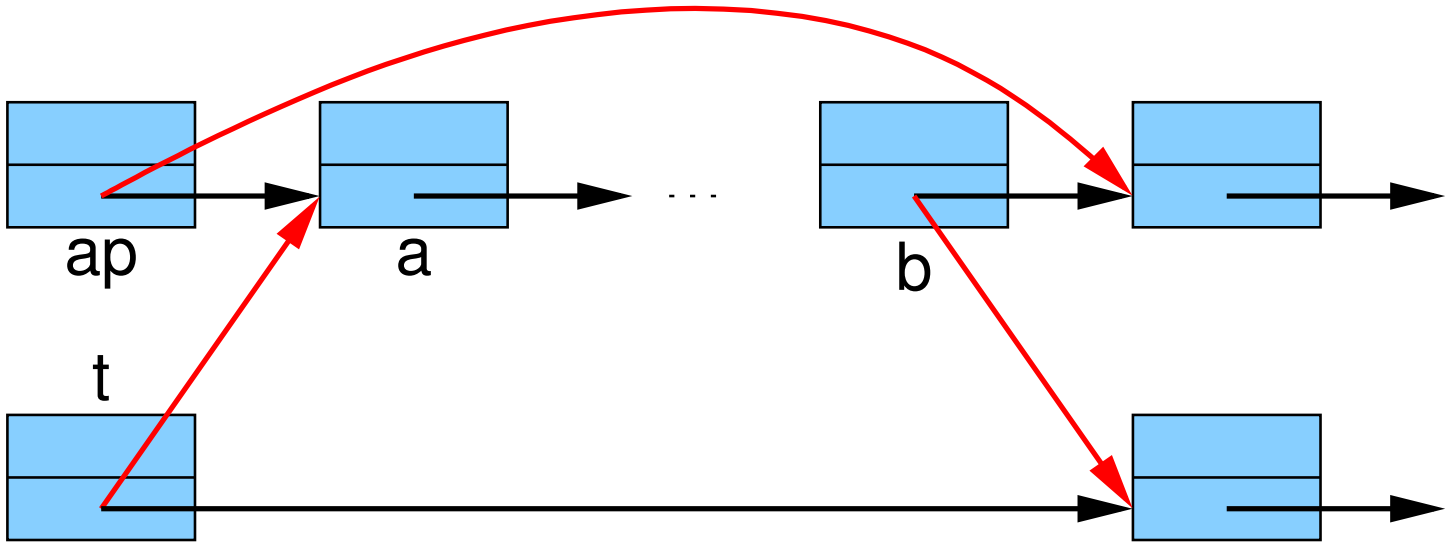



```
class SList<Elem> {
    SItem<Elem> h;
    . . . weitere Variablen und Methoden . . .
}
```

Einfach verkettete Liste

```
(static) splice(SHandle ap, SHandle b, SHandle t) {
    SHandle a = ap→next;
    ap→next = b→next;
    b→next = t→next;
    t→next = a;
}
```

Wir brauchen hier den Vorgänger ap von a!



Einfach verkettete Liste

- findNext sollte evt. auch nicht den nächsten Treffer, sondern dessen Vorgänger liefern
  (damit man das gefundene SItem auch löschen kann, Suche könnte dementsprechend erst beim Nachfolger des gegebenen SItems starten)

- auch einige andere Methoden brauchen ein modifiziertes Interface

- sinnvoll: Pointer zum letzten Item

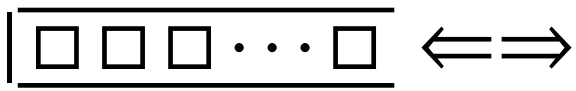
$\Rightarrow$ pushBack in $\mathcal{O}(1)$

Übersicht

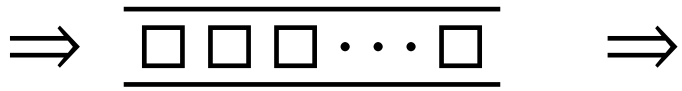
Stacks und Queues

Grundlegende sequenzbasierte Datenstrukturen:

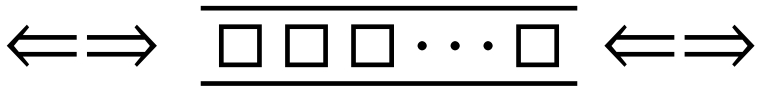
- Stack (Stapel)



- (FIFO-)Queue (Schlange)



- Deque (double-ended queue)



Stacks und Queues

Stack-Methoden:

- pushBack (bzw. push)
- popBack (bzw. pop)
- last (bzw. top)

Queue-Methoden:

- pushBack
- popFront
- first

Stacks und Queues

Warum spezielle Sequenz-Typen betrachten, wenn wir mit der bekannten Datenstruktur für Listen schon alle benötigten Operationen in $\mathcal{O}(1)$ haben?

- Programme werden lesbarer und einfacher zu debuggen, wenn spezialisierte Zugriffsmuster explizit gemacht werden.
- Einfachere Interfaces erlauben eine größere Breite von konkreten Implementationen (hier z.B. platzsparendere als Listen).
- Listen sind ungünstig, wenn die Operationen auf dem Sekundärspeicher (Festplatte) ausgeführt werden.

  Sequentielle Zugriffsmuster können bei entsprechender Implementation (hier z.B. als Arrays) stark vom Cache profitieren.

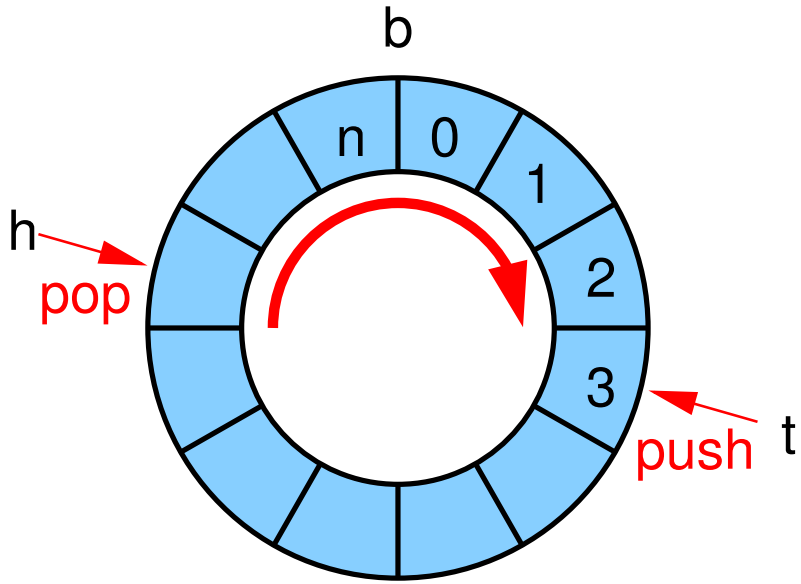
Stacks und Queues

Spezielle Umsetzungen:

- Stacks mit beschränkter Größe ⇒ Bounded Arrays
- Stacks mit unbeschränkter Größe ⇒ Unbounded Arrays
- oder: Stacks als einfach verkettete Listen
  (top of stack = front of list)
- (FIFO-)Queues: einfach verkettete Listen mit Zeiger auf letztes Element (eingefügt wird am Listenende, entnommen am Listenanfang, denn beim Entnehmen muss der Nachfolger bestimmt werden)
- Deques ⇒ doppelt verkettete Listen
  (einfach verkettete reichen nicht)

Beschränkte Queues

```
class BoundedFIFO<Elem> {
    const int n;     // Maximale Anzahl
    Elem[n+1] b;
    int h=0;              // erstes Element
    int t=0;       // erster freier Eintrag
}
```

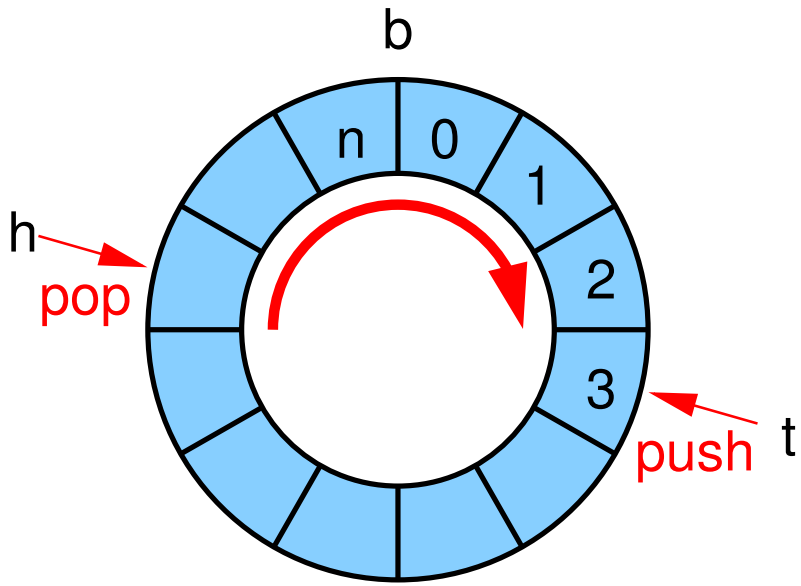


- Queue besteht aus den Feldelementen $h \ldots t-1$
- Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)

Beschränkte Queues

Methoden

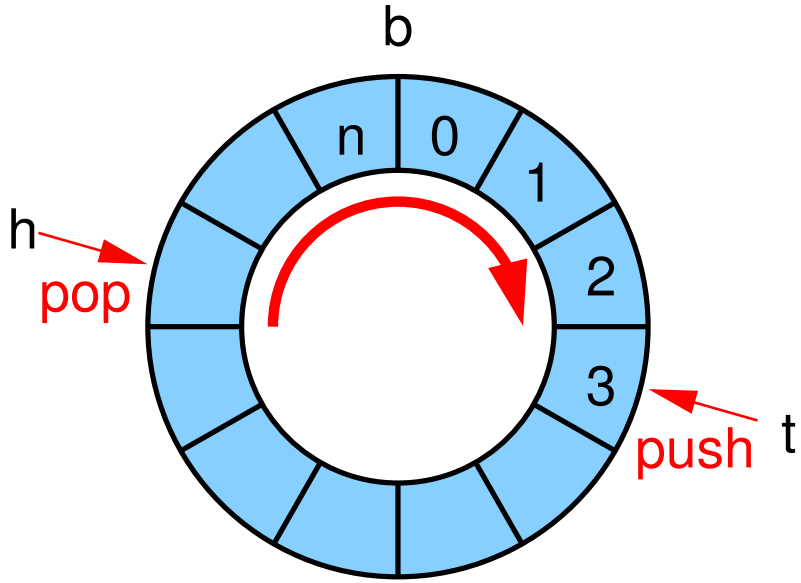
```
boolean isEmpty() {
    return (h==t);
}

Elem first() {
    assert(!isEmpty());
    return b[h];
}

int size() {
    return (t-h+n+1)%(n+1);
}
```



Beschränkte Queues

Methoden

```
pushBack(Elem x) {
    assert(size()<n);
    b[t]=x;
    t=(t+1)%(n+1);
}

popFront() {
    assert(!isEmpty());
    h=(h+1)%(n+1);
}

int size() {
    return (t-h+n+1)%(n+1);
}
```



Beschränkte Queues

- Struktur kann auch als Deque verwendet werden
- Zirkuläre Arrays erlauben auch den indexierten Zugriff:

```
Elem Operator [int i] {
    return b[(h+i)%(n+1)];
}
```

- Bounded Queues / Deques können genauso zu Unbounded Queues / Deques erweitert werden wie Bounded Arrays zu Unbounded Arrays

Übersicht

Sortierte Sequenz

S: sortierte Sequenz

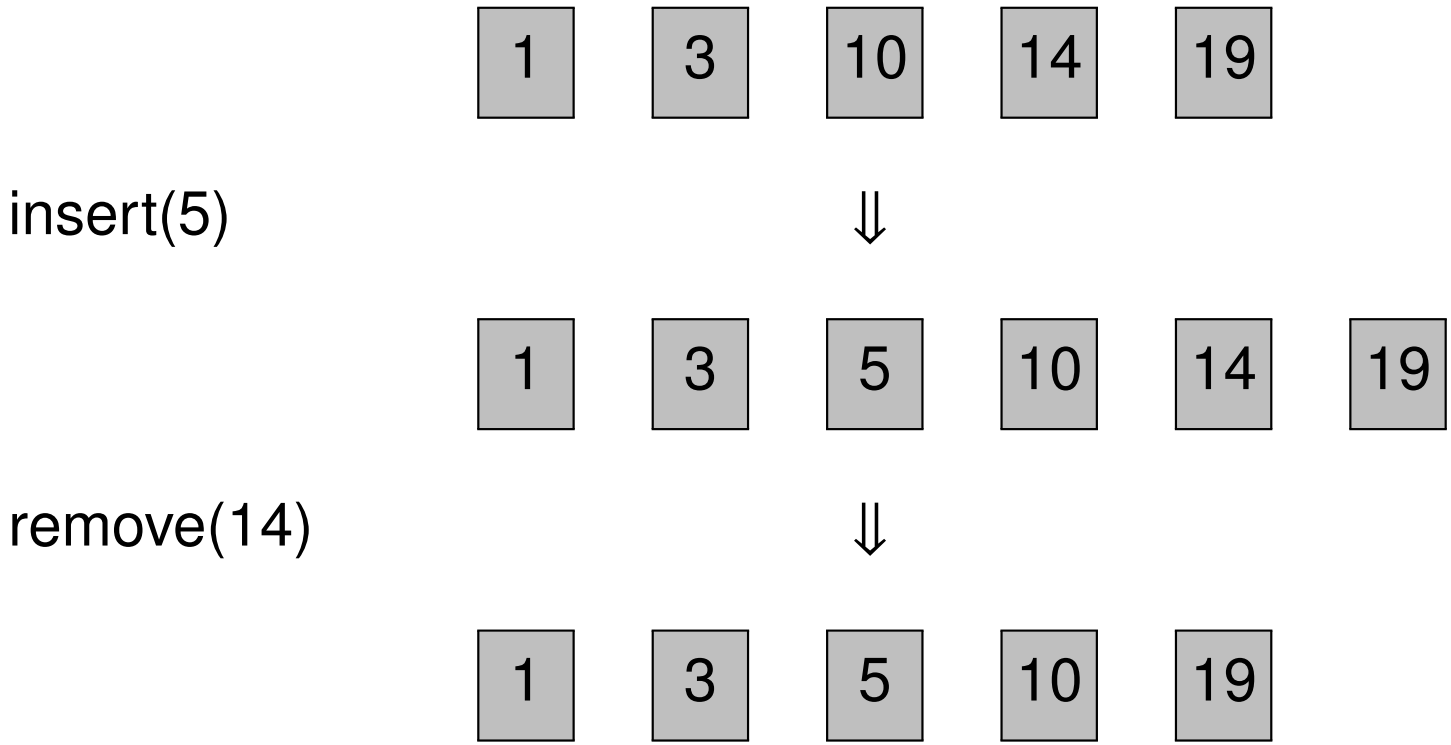
Jedes Element e identifiziert über key(e)

Operationen:

- $\langle e_1, \ldots, e_n \rangle$.insert$(e) = \langle e_1, \ldots, e_i, e, e_{i+1}, \ldots, e_n \rangle$
  für das i mit key$(e_i) <$ key$(e) <$ key(e_{i+1})
- $\langle e_1, \ldots, e_n \rangle$.remove$(k) = \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle$
  für das i mit key$(e_i) = k$
- $\langle e_1, \ldots, e_n \rangle$.find$(k) = e_i$
  für das i mit key$(e_i) = k$

Sortierte Sequenz

Problem:
Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung



Sortierte Sequenz

Realisierung als Liste

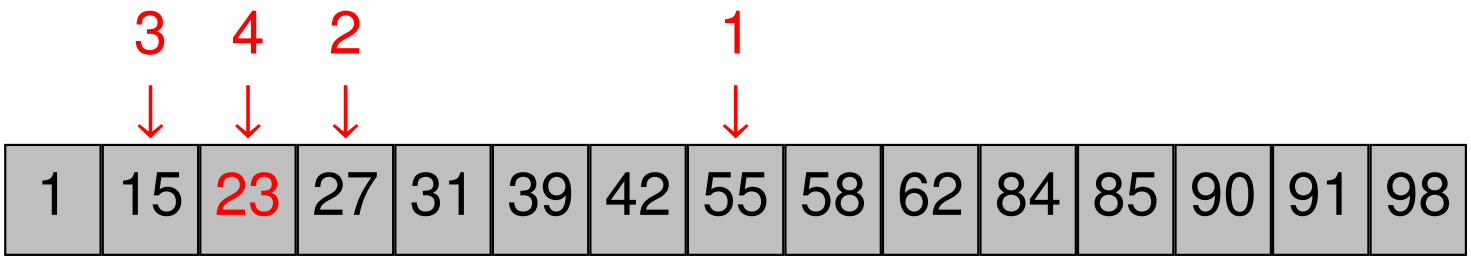
- insert und remove kosten zwar eigentlich nur konstante Zeit, müssen aber wie find zunächst die richtige Position finden
- find auf Sequenz der Länge n kostet $\mathcal{O}(n)$ Zeit, damit ebenso insert und remove

Realisierung als Feld

- find kann mit binärer Suche in Zeit $\mathcal{O}(\log n)$ realisiert werden
- insert und remove kosten $\mathcal{O}(n)$ Zeit für das Verschieben der nachfolgenden Elemente

Binäre Suche

find(23):



In einer n-elementigen Sequenz kann ein beliebiges Element mit maximal $2 + \lfloor \log_2 n \rfloor$ Vergleichen gefunden werden.

Übersicht

Übersicht

Assoziative Arrays / Wörterbücher

- Assoziatives Array / Wörterbuch (dictionary) S:
  speichert eine Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

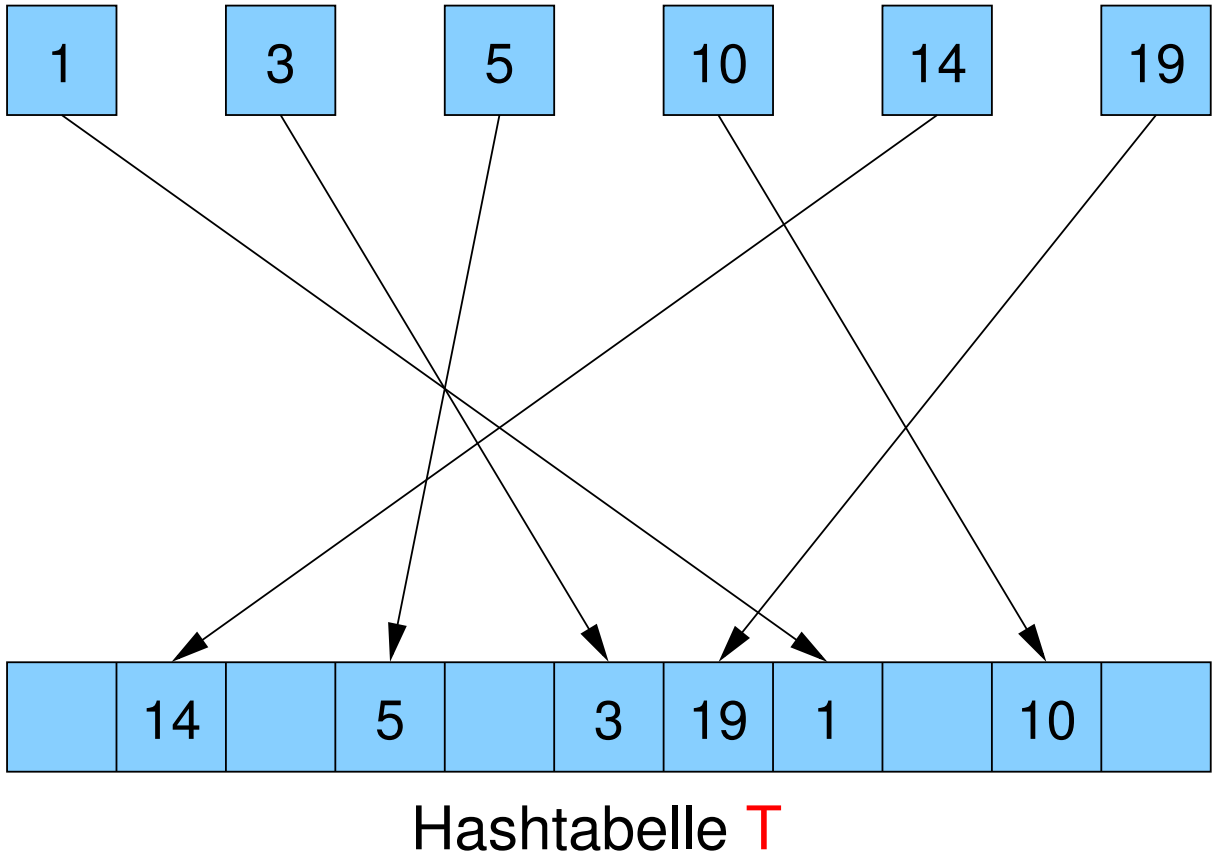
- S.insert(Elem e): $S := S \cup \{e\}$
- S.remove(Key k): $S := S \setminus \{e\}$,
  wobei e das Element mit key(e) $= k$ ist
- S.find(Key k):
  gibt das Element $e \in S$ mit key(e) $= k$ zurück, falls es existiert,
  sonst $\perp$ (entspricht Array-Indexoperator [ ], daher der Name)

Hashfunktion und Hashtabelle

Hashfunktion h :
Key $\mapsto \{0, \ldots, m-1\}$

$|\text{Key}| = N$

gespeicherte
Elemente: n



Hashfunktion

Anforderungen:

- schneller Zugriff (Zeiteffizienz)
- platzsparend (Speichereffizienz)
  (z.B. surjektive Abbildung möglicher Schlüssel auf die Adressen)
- gute Streuung bzw. Verteilung der Elemente
  über die ganze Tabelle
- Idealfall: Element e direkt in Tabelleneintrag t[h(key(e))]

$\Rightarrow$ find, insert und remove in konstanter Zeit

(genauer: plus Zeit für Berechnung der Hashfunktion)

Hashing

Annahme: perfekte Streuung

```
insert(Elem e) {
   T[h(key(e))] = e;
}

remove(Key k) {
   T[h(k)] = null;
}

Elem find(Key k) {
   return T[h(k)];
}
```

statisches Wörterbuch: nur find
dynamisches Wörterbuch: insert, remove und find

Kollisionen

In der Praxis:

- perfekte Zuordnung zwischen den gespeicherten Schlüsseln und den Adressen der Tabelle nur bei statischem Array möglich
- leere Tabelleneinträge
- Schlüssel mit gleicher Adresse (Kollisionen)

Wie wahrscheinlich ist eine Kollision?

- Geburtstagsparadoxon: In einer Menge von 23 zufällig ausgewählten Personen gibt es mit Wahrscheinlichkeit $> 50\%$ zwei Leute, die am gleichen Tag Geburtstag feiern.
- Bei zufälliger Abbildung von 23 Schlüsseln auf die Adressen einer Hashtabelle der Größe 365 gibt es mit Wahrscheinlichkeit $> 50\%$ eine Kollision.

Wahrscheinlichkeit von Kollisionen

- Hilfsmittel: $\forall x \in \mathbb{R}: \quad 1 + x \leq \sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$

$\Rightarrow$ $\forall x \in \mathbb{R}: \quad \ln(1+x) \leq x$ (da $\ln(x)$ monoton wachsend ist)

- Pr[keine Kollision beim i-ten Schlüssel] $= \frac{m-(i-1)}{m}$ für $i \in [1 \ldots n]$

$$\begin{aligned} \Pr[\text{keine Kollision}] &= \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \\ &= e^{\left[\sum_{i=0}^{n-1} \ln(1-\frac{i}{m})\right]} \leq e^{\left[\sum_{i=0}^{n-1} (-\frac{i}{m})\right]} = e^{\left[-\frac{n(n-1)}{2m}\right]} \end{aligned}$$

(da e^x monoton wachsend ist)

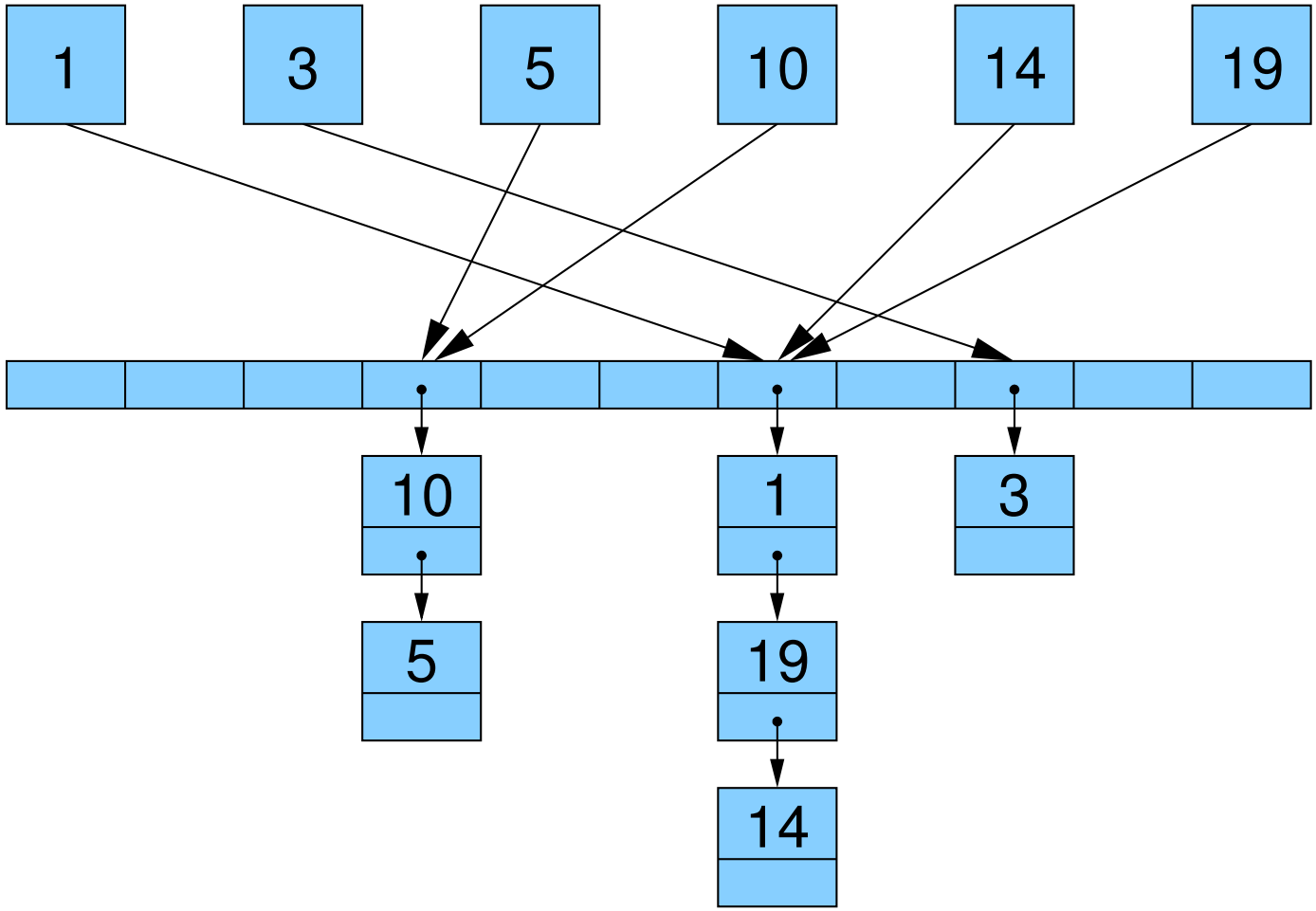
$\Rightarrow$ Bei gleichverteilt zufälliger Hashposition für jeden Schlüssel tritt für $n \in \omega(\sqrt{m})$ mit Wahrscheinlichkeit $1 - o(1)$ mindestens eine Kollision auf.

Übersicht

Dynamisches Wörterbuch

Hashing with Chaining:

Feld von Listen oder Zeigern



unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with Chaining:

```
List<Elem>[m] T;

insert(Elem e) {
    T[h(key(e))].insert(e);
}

remove(Key k) {
    T[h(k)].remove(k);
}

find(Key k) {
    T[h(k)].find(k);
}
```

Hashing with Chaining

- Platzverbrauch: $\mathcal{O}(n+m)$
- insert benötigt konstante Zeit
- remove und find müssen u.U. eine ganze Liste scannen
- im worst case sind alle Elemente in dieser Liste

$\Rightarrow$ im worst case ist Hashing with chaining nicht besser als eine normale Liste

Hashing with Chaining

Gibt es Hashfunktionen, die garantieren, dass alle Listen kurz sind?

- nein, für jede Hashfunktion gibt es eine Adresse, der mindestens N/m mögliche Schlüssel zugeordnet sind
  (erweitertes Schubfachprinzip / pigeonhole principle)
- Meistens ist $n < N/m$ (weil N riesig ist).
- In diesem Fall kann die Suche zum Scan aller Elemente entarten.

$\Rightarrow$ Auswege

- Average-case-Analyse
- Randomisierung
- Änderung des Algorithmus
  (z.B. Hashfunktion abhängig von aktuellen Schlüsseln)

Hashing with Chaining

Betrachte als Hashfunktionsmenge die Menge aller Funktionen, die die Schlüsselmenge (mit Kardinalität N) auf die Zahlen $0, \ldots, m-1$ abbilden.

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + n/m)$.

Unrealistisch: es gibt m^N solche Funktionen und man braucht $\log_2(m^N) = N \log_2 m$ Bits, um eine Funktion zu spezifizieren.

$\Rightarrow$ widerspricht dem Ziel, den Speicherverbrauch von N auf n zu senken!

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei remove(k) oder find(k)
- Laufzeit ist Konstante plus Zeit für Scan der Liste
  also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$
- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = i$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge $\mathbb{E}[X]$

$$
\begin{aligned}
&= \mathbb{E}\left[\sum_{e \in S} X_e\right] = \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
&= \sum_{e \in S} \Pr[X_e = 1] = \sum_{e \in S} 1/m = n/m
\end{aligned}
$$

□

Übersicht

c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \ldots, m-1\}$ heißt c-universell, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$\left|\{h \in H : \; h(x) = h(y)\}\right| \leq \frac{c}{m}|H|.$$

D.h. bei zufälliger Auswahl der Hashfunktion $h \in H$ gilt

$$\forall \{x, y\}_{(x \neq y)} : \quad \Pr[h(x) = h(y)] \leq \frac{c}{m}$$

1-universelle Familien nennt man universell.

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + c \cdot n/m)$.

Beweis.

- Betrachte festen Schlüssel k
- Zugriffszeit X ist $O(1 +$ Länge der Liste $T[h(k)])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$ zeigt an, ob e auf die gleiche Position wie k gehasht wird

c-Universal Hashing with Chaining

Beweis.

- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = h(k)$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in S} X_e\right] \\
&= \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
&= \sum_{e \in S} \Pr[X_e = 1] \leq \sum_{e \in S} c/m = n \cdot c/m
\end{aligned}$$

□

Beispiele für c-universelles Hashing

Einfache c-universelle Hashfunktionen?

Annahme: Schlüssel sind Bitstrings einer bestimmten Länge

Wähle als Tabellengröße m eine Primzahl

$\Rightarrow$ dann ist der Restklassenring modulo m (also $\mathbb{Z}_m$) ein Körper, d.h. es gibt zu jedem Element außer für die Null genau ein Inverses bzgl. Multiplikation

- Sei $w = \lfloor \log_2 m \rfloor$.
- unterteile die Bitstrings der Schlüssel in Teile zu je w Bits
- Anzahl der Teile sei k
- interpretiere jeden Teil als Zahl aus dem Intervall $[0, \ldots, 2^w - 1]$
- interpretiere Schlüssel x als k-Tupel solcher Zahlen:

$$\mathbf{x} = (x_1, \ldots, x_k)$$

Familie für universelles Hashing

Definiere für jeden Vektor

$$\mathbf{a} = (a_1, \ldots, a_k) \in \{0, \ldots, m-1\}^k$$

mittels Skalarprodukt

$$\mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{k} a_i x_i$$

eine Hashfunktion von der Schlüsselmenge
in die Menge der Zahlen $\{0, \ldots, m-1\}$

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \mod m$$

Familie für universelles Hashing

Satz

Wenn m eine Primzahl ist, dann ist

$$H = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0, \ldots, m-1\}^k \right\}$$

eine [1-]universelle Familie von Hashfunktionen.

Oder anders:
das Skalarprodukt zwischen einer Tupeldarstellung des Schlüssels und einem Zufallsvektor modulo m definiert eine gute Hashfunktion.

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

$\Rightarrow$ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits

- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \ldots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 269 - 1] = \{0, \ldots, 268\}$, spezifiziert z.B. $\mathbf{a} = (2, 4, 261, 16)$

$\Rightarrow$ Hashfunktion:

$$h_{\mathbf{a}}(\mathbf{x}) = (2x_1 + 4x_2 + 261x_3 + 16x_4) \bmod 269$$

$$h_{\mathbf{a}}(\mathbf{x}) = (2 \cdot 11 + 4 \cdot 7 + 261 \cdot 4 + 16 \cdot 3) \bmod 269 = 66$$

Eindeutiges a_j

Beweis

- Betrachte zwei beliebige verschiedene Schlüssel $\mathbf{x} = \{x_1, \ldots, x_k\}$ und $\mathbf{y} = \{y_1, \ldots, y_k\}$
- Wie groß ist $\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})]$?
- Sei j ein Index (von evt. mehreren möglichen) mit $x_j \neq y_j$ (muss es geben, sonst wäre $\mathbf{x} = \mathbf{y}$)

$\Rightarrow$ $(x_j - y_j) \not\equiv 0 \bmod m$
d.h., es gibt genau ein multiplikatives Inverses $(x_j - y_j)^{-1}$

$\Rightarrow$ gegeben Primzahl m und Zahlen $x_j, y_j, b \in \{0, \ldots m-1\}$ hat jede Gleichung der Form

$$a_j(x_j - y_j) \equiv b \bmod m$$

eine eindeutige Lösung: $a_j \equiv (x_j - y_j)^{-1} b \bmod m$

Wann wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

Wenn man alle Variablen a_i außer a_j festlegt, gibt es exakt eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$, denn

$$
\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \quad &\Leftrightarrow \quad \sum_{i=1}^{k} a_i x_i \equiv \sum_{i=1}^{k} a_i y_i && \mod m \\
&\Leftrightarrow \quad a_j(x_j - y_j) \equiv \sum_{i \neq j} a_i (y_i - x_i) && \mod m \\
&\Leftrightarrow \quad a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j} a_i (y_i - x_i) && \mod m
\end{aligned}
$$

Wie oft wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

- Es gibt m^{k-1} Möglichkeiten, Werte für die Variablen a_i mit $i \neq j$ zu wählen.
- Für jede solche Wahl gibt es genau eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
- Für $\mathbf{a}$ gibt es insgesamt m^k Auswahlmöglichkeiten.
- Also

$$\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}$$

□

Familie für k-universelles Hashing

Definiere für $a \in \{1, \ldots, m-1\}$ die Hashfunktion

$$h'_a(\mathbf{x}) = \sum_{i=1}^{k} a^{i-1} x_i \bmod m$$

(mit $x_i \in \{0, \ldots, m-1\}$)

Satz

Für jede Primzahl m ist

$$H' = \{h'_a : \ a \in \{1, \ldots, m-1\}\}$$

eine k-universelle Familie von Hashfunktionen.

Familie für k-universelles Hashing

Beweisidee:

Für Schlüssel $\mathbf{x} \neq \mathbf{y}$ ergibt sich folgende Gleichung:

$$\begin{aligned} h'_a(\mathbf{x}) &= h'_a(\mathbf{y}) \\ h'_a(\mathbf{x}) - h'_a(\mathbf{y}) &\equiv 0 \bmod m \\ \sum_{i=1}^{k} a^{i-1}(x_i - y_i) &\equiv 0 \bmod m \end{aligned}$$

Anzahl der Nullstellen des Polynoms in a ist durch den Grad des Polynoms beschränkt (Fundamentalsatz der Algebra), also durch $k-1$.

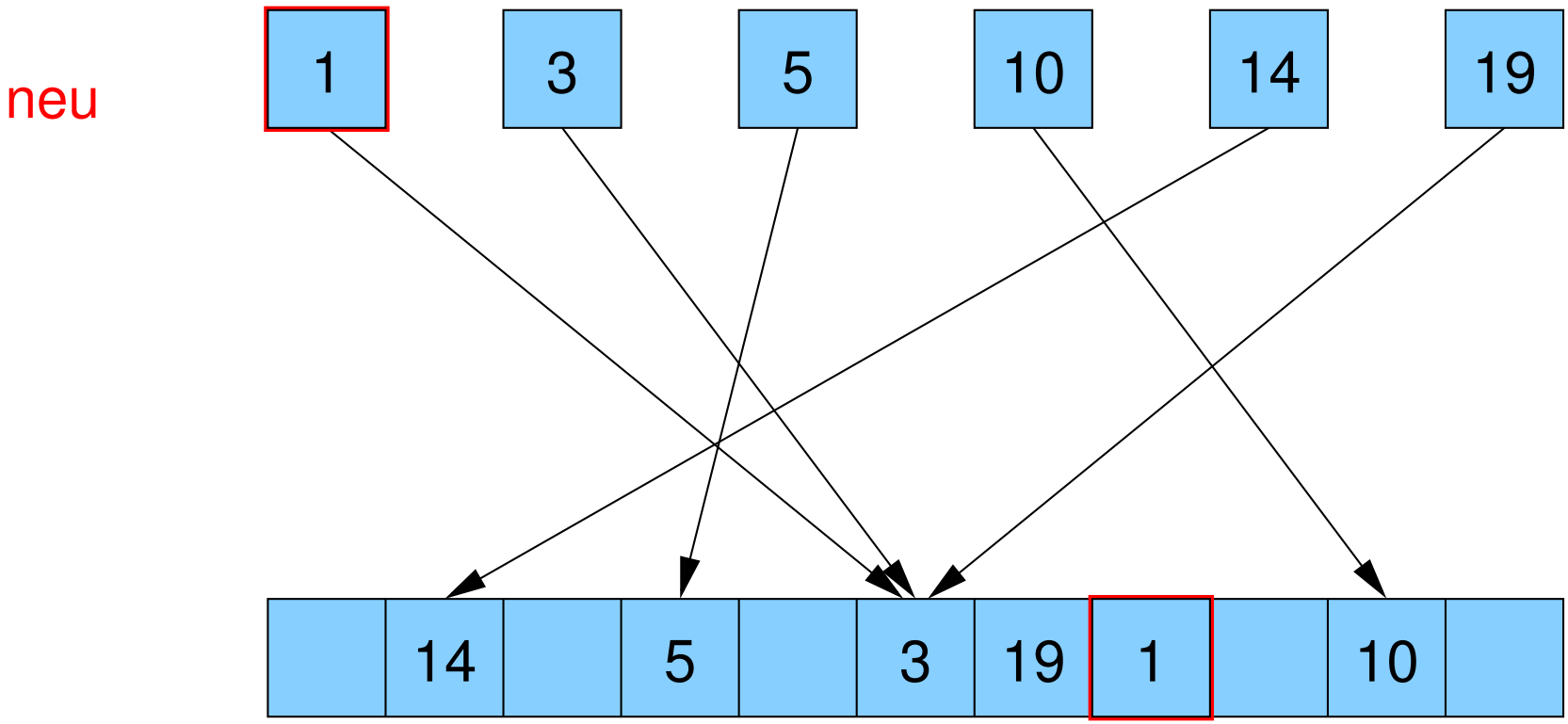
Falls $k \leq m$ können also höchstens $k-1$ von $m-1$ möglichen Werten für a zum gleichen Hashwert für $\mathbf{x}$ und $\mathbf{y}$ führen.

Aus $\Pr[h(\mathbf{x}) = h(\mathbf{y})] \leq \frac{\min\{k-1, m-1\}}{m-1} \leq \frac{k}{m}$ folgt, dass H' k-universell ist.

Übersicht

Dynamisches Wörterbuch

Hashing with Linear Probing:



Speichere Element e im ersten freien Ort T[i], T[$i+1$], T[$i+2$], ... mit $i == h(\text{key}(e))$
(Ziel: Folgen besetzter Positionen möglichst kurz)

Hashing with Linear Probing

```
Elem[m] T;                    // Feld sollte genügend groß sein

insert(Elem e) {
    i = h(key(e));
    while (T[i] ≠ null ∧ T[i] ≠ e)
        i = (i+1) % m;
    T[i] = e;
}

find(Key k) {
    i = h(k);
    while (T[i] ≠ null ∧ key(T[i]) ≠ k)
        i = (i+1) % m;
    return T[i];
}
```

Hashing with Linear Probing

Vorteil:

Es werden im Gegensatz zu Hashing with Chaining (oder auch im Gegensatz zu anderen Probing-Varianten) nur zusammenhängende Speicherzellen betrachtet.

$\Rightarrow$ Cache-Effizienz!

Hashing with Linear Probing

Problem: Löschen von Elementen

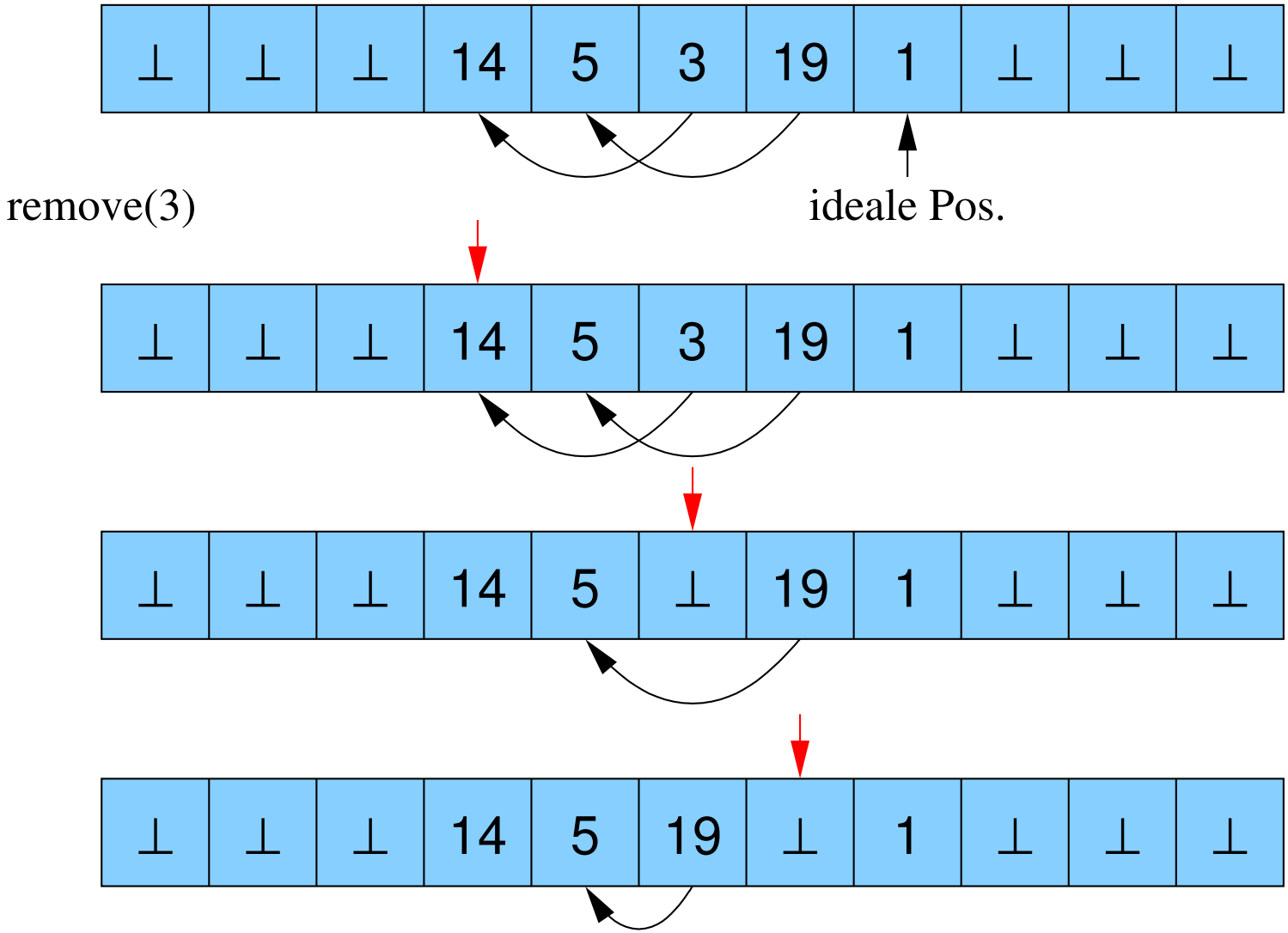
1. Löschen verbieten
2. Markiere Positionen als gelöscht
   (mit speziellem Zeichen $\neq \perp$)
   Suche endet bei $\perp$, aber nicht bei markierten Zellen

   Problem: Anzahl echt freier Zellen sinkt monoton
   $\Rightarrow$ Suche wird evt. langsam oder periodische Reorganisation
3. Invariante sicherstellen:

   Für jedes $e \in S$ mit idealer Position $i = h(key(e))$ und aktueller Position j gilt:
   T[i], T[i+1],. . . ,T[j] sind besetzt

Hashing with Linear Probing

Löschen / Aufrechterhaltung der Invariante



Übersicht

Dynamisches Wörterbuch

Problem: Hashtabelle ist zu groß oder zu klein
(sollte nur um konstanten Faktor von Anzahl der Elemente abweichen)

Lösung: Reallokation

- Wähle geeignete Tabellengröße
- Wähle neue Hashfunktion
- Übertrage Elemente auf die neue Tabelle

Dynamisches Wörterbuch

Problem: Tabellengröße m sollte prim sein
(für eine gute Verteilung der Schlüssel)

Lösung:

- Für jedes k gibt es eine Primzahl in $\left[k^3, (k+1)^3\right]$
- Jede Zahl $z \leq (k+1)^3$, die nicht prim ist, muss einen Teiler $t \leq \sqrt{(k+1)^3} = (k+1)^{3/2}$ haben.
- Für eine gewünschte ungefähre Tabellengröße m' (evt. nicht prim) bestimme k so, dass $k^3 \leq m' \leq (k+1)^3$
- Größe des Intervalls:
  $(k+1)^3 - k^3 + 1 = (k^3 + 3k^2 + 3k + 1) - k^3 + 1 = 3k^2 + 3k + 2$
- Für jede Zahl $j = 2, \ldots, (k+1)^{3/2}$:
  streiche die Vielfachen von j in $[k^3, (k+1)^3]$
- Für jedes j kostet das Zeit $\left((k+1)^3 - k^3 + 1\right)/j \in O(k^2/j)$

Dynamisches Wörterbuch

- Hilfsmittel: Wachstum der harmonischen Reihe

$$\ln n \quad \leq \quad H_n = \sum_{i=1}^{n} \frac{1}{i} \quad \leq \quad 1 + \ln n$$

- insgesamt:

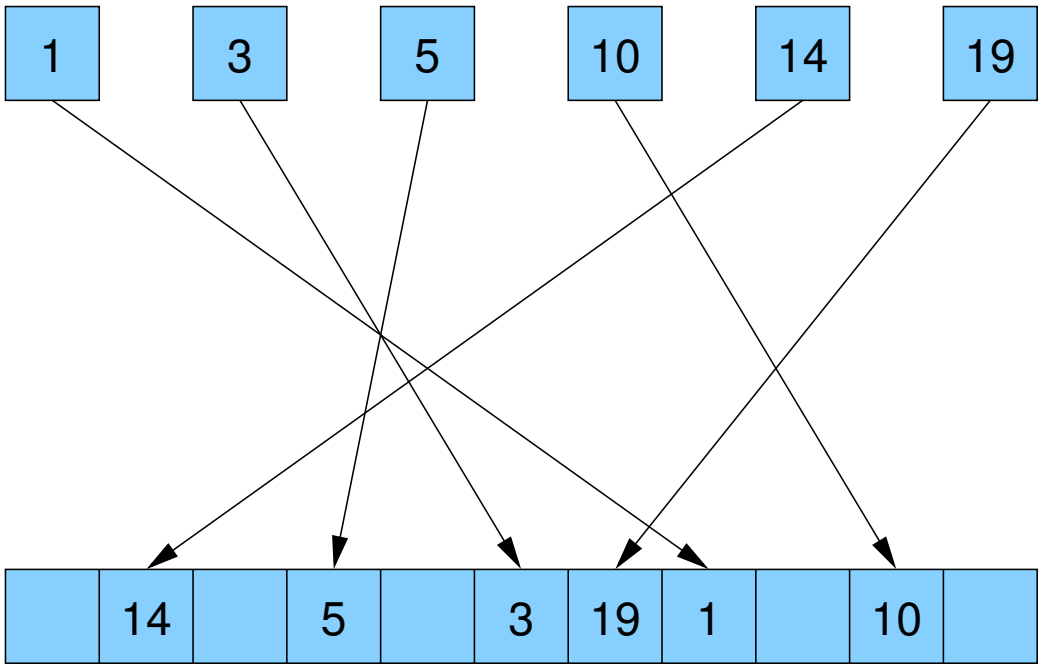
$$\begin{aligned}
\sum_{2 \leq j \leq (k+1)^{3/2}} O\left(\frac{k^2}{j}\right) \quad &\leq \quad k^2 \sum_{2 \leq j \leq (k+1)^{3/2}} O\left(\frac{1}{j}\right) \\
&\in \quad k^2 \cdot O\left(\ln\left((k+1)^{3/2}\right)\right) \\
&\in \quad O(k^2 \ln k) \\
&\in \quad o(m)
\end{aligned}$$

$\Rightarrow$ Kosten zu vernachlässigen im Vergleich zur Initialisierung der Tabelle der Größe m (denn m ist kubisch in k)

Übersicht

Perfektes Hashing für statisches Wörterbuch

- bisher: konstante *erwartete* Laufzeit
  falls m im Vergleich zu n genügend groß gewählt wird
  (nicht ausreichend für Real Time Scenario)
- Ziel: konstante Laufzeit im worst case für find()
  durch perfekte Hashtabelle ohne Kollisionen
- Annahme: statische Menge S von n Elementen



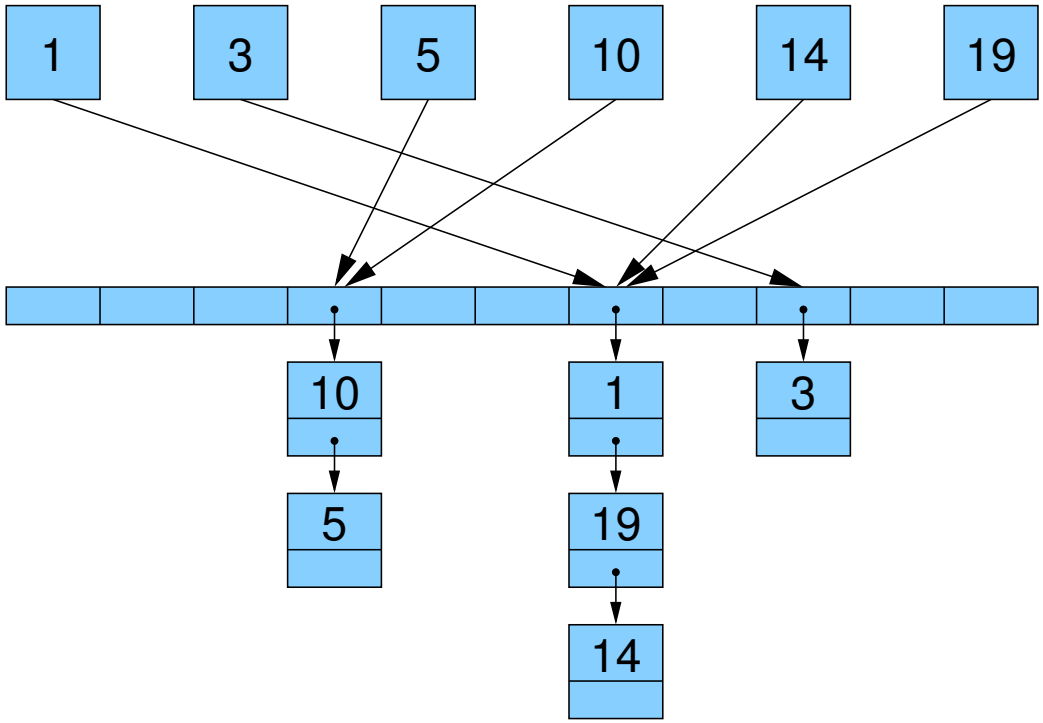
Statisches Wörterbuch

- S: feste Menge von n Elementen mit Schlüsseln k_1 bis k_n
- H_m: c-universelle Familie von Hashfunktionen auf $\{0, \ldots, m-1\}$
  (Hinweis: 2-universelle Familien existieren für alle m)
- $C(h)$ für $h \in H_m$: Anzahl Kollisionen in S für h, d.h.

$$C(h) = \left|\{(x,y) : x, y \in S,\ x \neq y,\ h(x) = h(y)\}\right|$$

Beispiel:

$C(h) = 2 + 6 = 8$



Erwartete Anzahl von Kollisionen

Lemma

Für die Anzahl der Kollisionen gilt: $\mathbb{E}[C(h)] \leq cn(n-1)/m.$

Beweis.

- Definiere $n(n-1)$ Indikator-Zufallsvariablen $X_{ij}(h)$:
  Für $i \neq j$ sei $X_{ij}(h) = 1 \Leftrightarrow h(k_i) = h(k_j)$.
- Dann ist $C(h) = \sum_{i \neq j} X_{ij}(h)$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i \neq j} X_{ij}\right] = \sum_{i \neq j} \mathbb{E}\left[X_{ij}\right] = \sum_{i \neq j} \Pr\left[X_{ij} = 1\right] \leq n(n-1) \cdot c/m$$

$\Rightarrow$ Für quadratische Tabellengröße ist die erwartete Anzahl von Kollisionen (und damit die erwartete worst-case-Laufzeit für find) eine Konstante. □

Markov-Ungleichung

Satz (Markov-Ungleichung)

Für jede nichtnegative Zufallsvariable X und Konstante $k > 0$ gilt:

$$\Pr[X \geq k] \leq \frac{\mathbb{E}[X]}{k} \qquad \textit{und für } \mathbb{E}[X] > 0\textit{:} \quad \Pr[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$$

Beweis.

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{z \in \mathbb{R}} z \cdot \Pr[X = z] \\
&\geq \sum_{z \geq k \cdot \mathbb{E}[X]} z \cdot \Pr[X = z] \geq \sum_{z \geq k \cdot \mathbb{E}[X]} k \cdot \mathbb{E}[X] \cdot \Pr[X = z] \\
&\geq k \cdot \mathbb{E}[X] \cdot \Pr[X \geq k \cdot \mathbb{E}[X]]
\end{aligned}$$

□

Hashfunktionen mit wenig Kollisionen

Lemma

Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt:

$$C(h) \leq 2cn(n-1)/m$$

Beweis.

- Aus Lemma $\mathbb{E}[C(h)] \leq cn(n-1)/m$ und Markov-Ungleichung $\Pr[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$ folgt für $n \geq 2$:

$$\Pr[C(h) \geq 2cn(n-1)/m] \quad \leq \quad \Pr\left[C(h) \geq 2\mathbb{E}[C(h)]\right] \quad \leq \quad \frac{1}{2}$$

$\Rightarrow$ Für höchstens die Hälfte der Funktionen ist $C(h) \geq \frac{2cn(n-1)}{m}$

$\Rightarrow$ Für mindestens die Hälfte der Funktionen ist $C(h) \leq \frac{2cn(n-1)}{m}$ $(n \geq 1)$

□

Hashfunktionen ohne Kollisionen

Lemma

Wenn $m \geq cn(n-1)+1$, dann bildet mindestens die Hälfte der Funktionen $h \in H_m$ die Schlüssel injektiv in die Indexmenge der Hashtabelle ab.

Beweis.

- Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt $C(h) < 2$.
- Da $C(h)$ immer eine gerade Zahl sein muss, folgt aus $C(h) < 2$ direkt $C(h) = 0$.

$\Rightarrow$ keine Kollisionen (bzw. injektive Abbildung)

□

- Wähle zufällig $h \in H_m$ mit $m \geq cn(n-1)+1$
- Prüfe auf Kollision $\Rightarrow$ behalten oder erneut wählen

$\Rightarrow$ Nach durchschnittlich 2 Versuchen erfolgreich

Statisches Wörterbuch

Ziel: lineare Tabellengröße

Idee: zweistufige Abbildung der Schlüssel

- Wähle Hashfunktion h mit wenig Kollisionen (≈ 2 Versuche)

$\Rightarrow$ 1. Stufe bildet Schlüssel auf Buckets von konstanter durchschnittlicher Größe ab

- Wähle Hashfunktionen h_ℓ ohne Kollisionen (≈ 2 Versuche pro h_ℓ)

$\Rightarrow$ 2. Stufe benutzt quadratisch viel Platz für jedes Bucket, um alle Kollisionen aus der 1. Stufe aufzulösen

Statisches Wörterbuch

- B_ℓ^h: Menge der Elemente in S, die h auf ℓ abbildet, $\ell \in \{0, \ldots, m-1\}$ und $h \in H_m$
- b_ℓ^h: Kardinalität von B_ℓ^h, also $b_\ell^h := \left|B_\ell^h\right|$
- Für jedes ℓ führen die Schlüssel in B_ℓ^h zu $b_\ell^h(b_\ell^h - 1)$ Kollisionen
- Also ist die Gesamtzahl der Kollisionen

$$C(h) = \sum_{\ell=0}^{m-1} b_\ell^h(b_\ell^h - 1)$$

Perfektes statisches Hashing: 1. Stufe

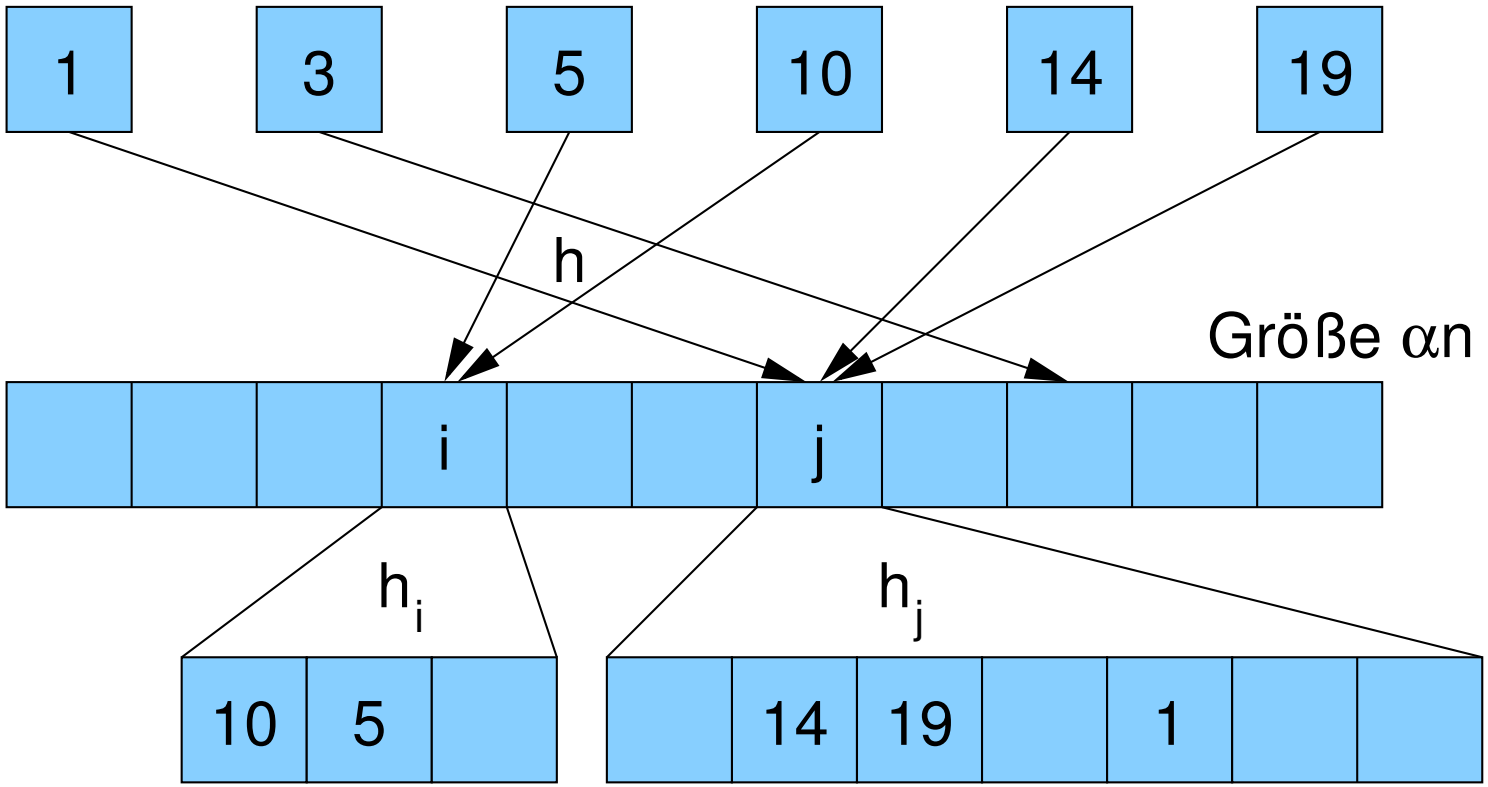
- 1. Stufe der Hashtabelle soll linear viel Speicher verwenden, also $\lceil \alpha n \rceil$ Adressen, wobei wir Konstante α später festlegen.
- Wähle Funktion $h \in H_{\lceil \alpha n \rceil}$, um S in Teilmengen B_ℓ aufzuspalten.

  Wähle h dabei so lange zufällig aus $H_{\lceil \alpha n \rceil}$ aus, bis gilt:

$$C(h) \leq \frac{2cn(n-1)}{\lceil \alpha n \rceil} \leq \frac{2cn(n-1)}{\alpha n} \leq \frac{2cn}{\alpha}$$

  Da das für mindestens die Hälfte der Hashfunktionen in $H_{\lceil \alpha n \rceil}$ gilt (vorletztes Lemma), erwarten wir dafür ≤ 2 Versuche.
- Für jedes $\ell \in \{0, \ldots, \lceil \alpha n \rceil - 1\}$ seien B_ℓ die Elemente, die durch h auf Adresse ℓ abgebildet werden und $b_\ell = |B_\ell|$ deren Anzahl.

Perfektes statisches Hashing



Perfektes statisches Hashing: 2. Stufe

- Für jedes B_ℓ:

  Berechne $m_\ell = cb_\ell(b_\ell - 1) + 1$.

  Wähle zufällig Funktion $h_\ell \in H_{m_\ell}$, bis h_ℓ die Menge B_ℓ injektiv in $\{0, \ldots, m_\ell - 1\}$ abbildet (also ohne Kollisionen).
  Mindestens die Hälfte der Funktionen in H_{m_ℓ} tut das.

- Hintereinanderreihung der einzelnen Tabellen ergibt eine Gesamtgröße der Tabelle von $\sum_\ell m_\ell$

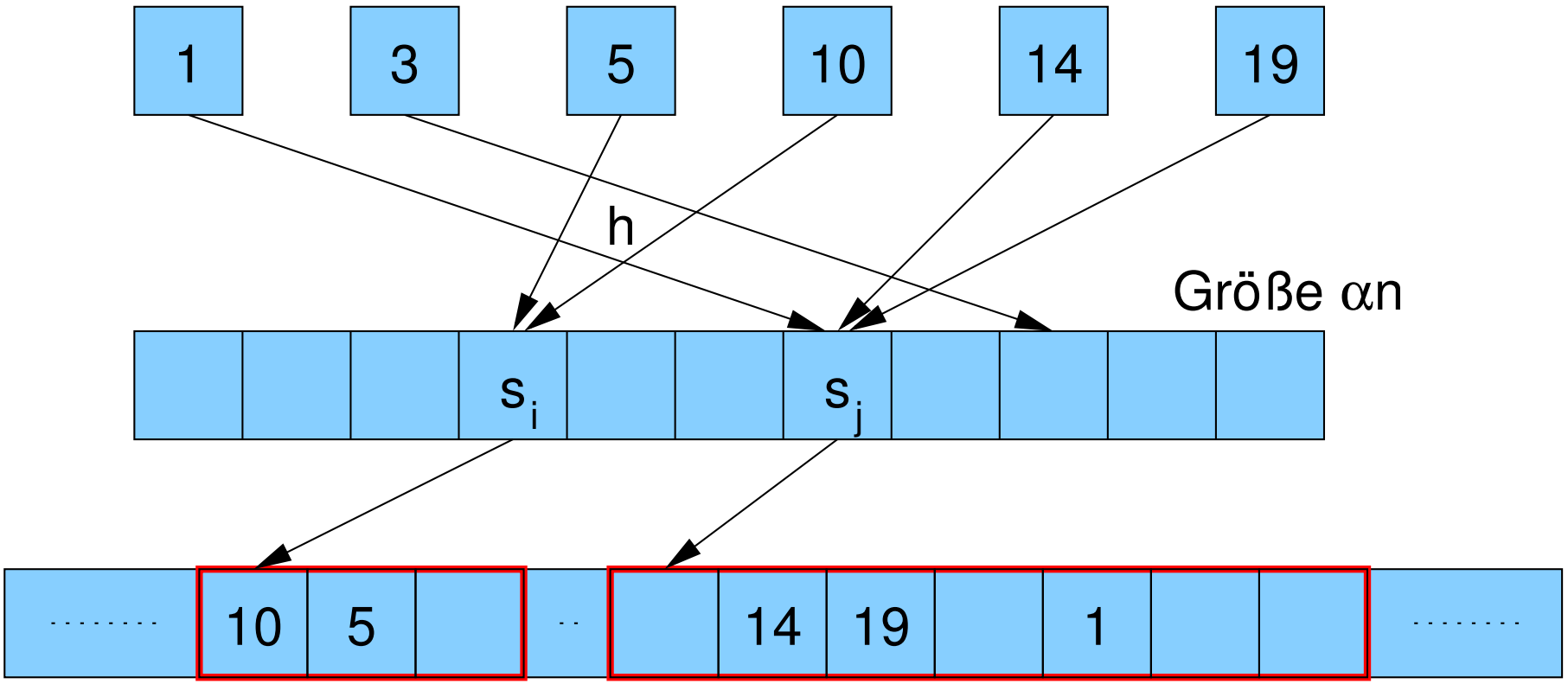
- Teiltabelle für B_ℓ beginnt an Position $s_\ell = m_0 + m_1 + \ldots + m_{\ell-1}$ und endet an Position $s_\ell + m_\ell - 1$

- Für gegebenen Schlüssel x, berechnen die Anweisungen

$$\ell = h(x); \quad \text{return } s_\ell + h_\ell(x);$$

  dann eine injektive Funktion auf der Menge S.

Perfektes statisches Hashing



Perfektes statisches Hashing

- Die Funktion ist beschränkt durch:

$$\begin{aligned}
\sum_{\ell=0}^{\lceil \alpha n \rceil - 1} m_\ell &= \sum_{\ell=0}^{\lceil \alpha n \rceil - 1} (c \cdot b_\ell (b_\ell - 1) + 1) \qquad \text{(siehe Def. der } m_\ell\text{'s)} \\
&\leq c \cdot C(h) + \lceil \alpha n \rceil \\
&\leq c \cdot 2cn/\alpha + \alpha n + 1 \\
&\leq (2c^2/\alpha + \alpha)n + 1
\end{aligned}$$

- Zur Minimierung der Schranke betrachte die Ableitung

$$\begin{aligned}
f(\alpha) &= (2c^2/\alpha + \alpha)n + 1 \\
f'(\alpha) &= (-2c^2/\alpha^2 + 1)n
\end{aligned}$$

$\Rightarrow$ $f'(\alpha) = 0$ liefert $\alpha = \sqrt{2}c$

$\Rightarrow$ Adressbereich: $0 \ldots 2\sqrt{2}cn$

Perfektes statisches Hashing

Satz

Für eine beliebige Menge von n Schlüsseln kann eine perfekte Hashfunktion mit Zielmenge $\{0,\ldots,2\sqrt{2}cn\}$ *in linearer erwarteter Laufzeit konstruiert werden.*

- Da wir wissen, dass wir für beliebige m eine 2-universelle Familie von Hashfunktionen finden können, kann man also z.B. eine Hashfunktion mit Adressmenge $\{0,\ldots,4\sqrt{2}n\}$ in linearer erwarteter Laufzeit konstruieren. (Las Vegas-Algorithmus)

- Unsere Minimierung hat nicht den Platz berücksichtigt, der benötigt wird, um die Werte s_ℓ, sowie die ausgewählten Hashfunktionen h_ℓ zu speichern.

$\Rightarrow$ Berechnung von α sollte eigentlich angepasst werden (entsprechend Speicherplatz pro Element, pro m_ℓ und pro h_ℓ)

Perfektes dynamisches Hashing

Kann man perfekte Hashfunktionen auch dynamisch konstruieren?

ja, z.B. mit Cuckoo Hashing

- 2 Hashfunktionen h_1 und h_2
- 2 Hashtabellen T_1 und T_2
- bei find und remove jeweils in beiden Tabellen nachschauen
- bei insert abwechselnd beide Tabellen betrachten, das zu speichernde Element an die Zielposition der aktuellen Tabelle schreiben und wenn dort schon ein anderes Element stand, dieses genauso in die andere Tabelle verschieben usw.
- evt. Anzahl Verschiebungen durch $2 \log n$ beschränken, um Endlosschleife zu verhindern
  (ggf. kompletter Rehash mit neuen Funktionen h_1, h_2)

Übersicht

Probleme beim linearen Sondieren

- Offene Hashverfahren allgemein:

  Erweiterte Hashfunktion $h(k, i)$ gibt an, auf welche Adresse ein Schlüssel k abgebildet werden soll, wenn bereits i Versuche zu einer Kollision geführt haben

- Lineares Sondieren (Linear Probing):

$$h(k, i) = (h(k) + i) \bmod m$$

- Primäre Häufung (primary clustering):
  tritt auf, wenn für Schlüssel k_1, k_2 mit unterschiedlichen Hashwerten $h(k_1) \neq h(k_2)$ ab einem bestimmten Punkt i_1 bzw. i_2 die gleiche Sondierfolge auftritt:

$$\exists i_1, i_2 \quad \forall j : \quad h(k_1, i_1 + j) = h(k_2, i_2 + j)$$

Probleme beim quadratischen Sondieren

- Quadratisches Sondieren (Quadratic Probing):

$$h(k,i) = \left(h(k) + c_1 i + c_2 i^2\right) \bmod m \qquad (c_2 \neq 0)$$

$$\text{oder:} \quad h(k,i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

- $h(k,i)$ soll möglichst surjektiv auf die Adressmenge $\{0,\ldots,m-1\}$ abbilden, um freie Positionen auch immer zu finden. Bei

$$h(k,i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

z.B. durch Wahl von m prim $\wedge$ $m \equiv 3 \bmod 4$

- Sekundäre Häufung (secondary clustering):
tritt auf, wenn für Schlüssel k_1, k_2 mit gleichem Hashwert $h(k_1) = h(k_2)$ auch die nachfolgende Sondierfolge gleich ist:

$$\forall i: \quad h(k_1,i) = h(k_2,i)$$

Double Hashing

- Auflösung der Kollisionen der Hashfunktion h
  durch eine zweite Hashfunktion h':

$$h(k,i) = [h(k) + i \cdot h'(k)] \bmod m$$

  wobei für alle k gelten soll, dass $h'(k)$ teilerfremd zu m ist,

$$\begin{array}{lrcl} \text{z.B.} & h'(k) & = & 1 + k \bmod m - 1 \\ \text{oder} & h'(k) & = & 1 + k \bmod m - 2 \end{array}$$

  für Primzahl m

- primäre und sekundäre Häufung werden weitgehend vermieden,
  aber nicht komplett ausgeschlossen

Übersicht

Statisches Wörterbuch

Lösungsmöglichkeiten:

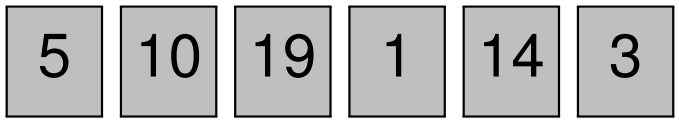
- Perfektes Hashing
  - Vorteil: Suche in konstanter Zeit
  - Nachteil: keine Ordnung auf Elementen, d.h. Bereichsanfragen (z.B. alle Namen, die mit 'A' anfangen) teuer

- Speicherung der Daten in sortiertem Feld
  - Vorteil: Bereichsanfragen möglich
  - Nachteil: Suche teurer (logarithmische Zeit)

Sortierproblem

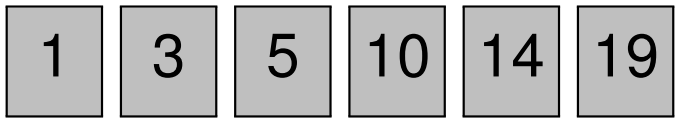
- gegeben: Ordnung $\leq$ auf der Menge möglicher Schlüssel
- Eingabe: Sequenz $s = \langle e_1, \ldots, e_n \rangle$

  Beispiel:



- Ausgabe: Permutation $s' = \langle e'_1, \ldots, e'_n \rangle$ von s,
  so dass $\text{key}(e'_i) \leq \text{key}(e'_{i+1})$ für alle $i \in \{1, \ldots, n\}$

  Beispiel:



Übersicht

SelectionSort

Sortieren durch Auswählen

Wähle das kleinste Element aus der (verbleibenden) Eingabesequenz und verschiebe es an das Ende der Ausgabesequenz

Beispiel

5	10	19	1	14	3
1	10	19	5	14	3
1	5	19	10	14	3
1	**3**	19	10	14	5
1	3	10	19	14	5

1	3	**5**	19	14	10
1	3	5	14	19	10
1	3	5	**10**	19	14
1	3	5	10	**14**	19
1	3	5	10	14	**19**

SelectionSort

Sortieren durch Auswählen

```
selectionSort(Element[ ] a, int n) {
    for (int i = 0; i < n; i++)
        // verschiebe min{a[i],...,a[n – 1]} nach a[i]
        for (int j = i + 1; j < n; j++)
            if (a[i] > a[j])
                swap(a[i], a[j]);
}
```

Zeitaufwand:

- Minimumsuche in Feld der Größe i: $\Theta(i)$
- Gesamtzeit: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
- Vorteil: einfach zu implementieren
- Nachteil: quadratische Laufzeit

InsertionSort

Sortieren durch Einfügen

Nimm ein Element aus der Eingabesequenz und füge es an der richtigen Stelle in die Ausgabesequenz ein

Beispiel

5	10	19	1	14	3
5	10	19	1	14	3
5	**10**	19	1	14	3
5	10	**19**	1	14	3
5	10	1	19	14	3

5	1	10	19	14	3
1	5	10	19	14	3
1	5	10	**14**	19	3
1	...	←	...	3	19
1	**3**	5	10	14	19

InsertionSort

Sortieren durch Einfügen

```
insertionSort(Element[ ] a, int n) {
    for (int i = 1; i < n; i++)
        // verschiebe a_i an die richtige Stelle
        for (int j = i − 1; j ≥ 0; j − −)
            if (a[j] > a[j + 1])
                swap(a[j], a[j + 1]);
}
```

Zeitaufwand:

- Einfügung des i-ten Elements an richtiger Stelle: $\mathcal{O}(i)$
- Gesamtzeit: $\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$
- Vorteil: einfach zu implementieren
- Nachteil: quadratische Laufzeit

Einfache Verfahren

SelectionSort

- mit besserer Minimumstrategie worst case Laufzeit $\mathcal{O}(n \log n)$ erreichbar
  (mehr dazu in einer späteren Vorlesung)

InsertionSort

- mit besserer Einfügestrategie worst case Laufzeit $\mathcal{O}(n \log^2 n)$ erreichbar
  ($\rightarrow$ ShellSort)

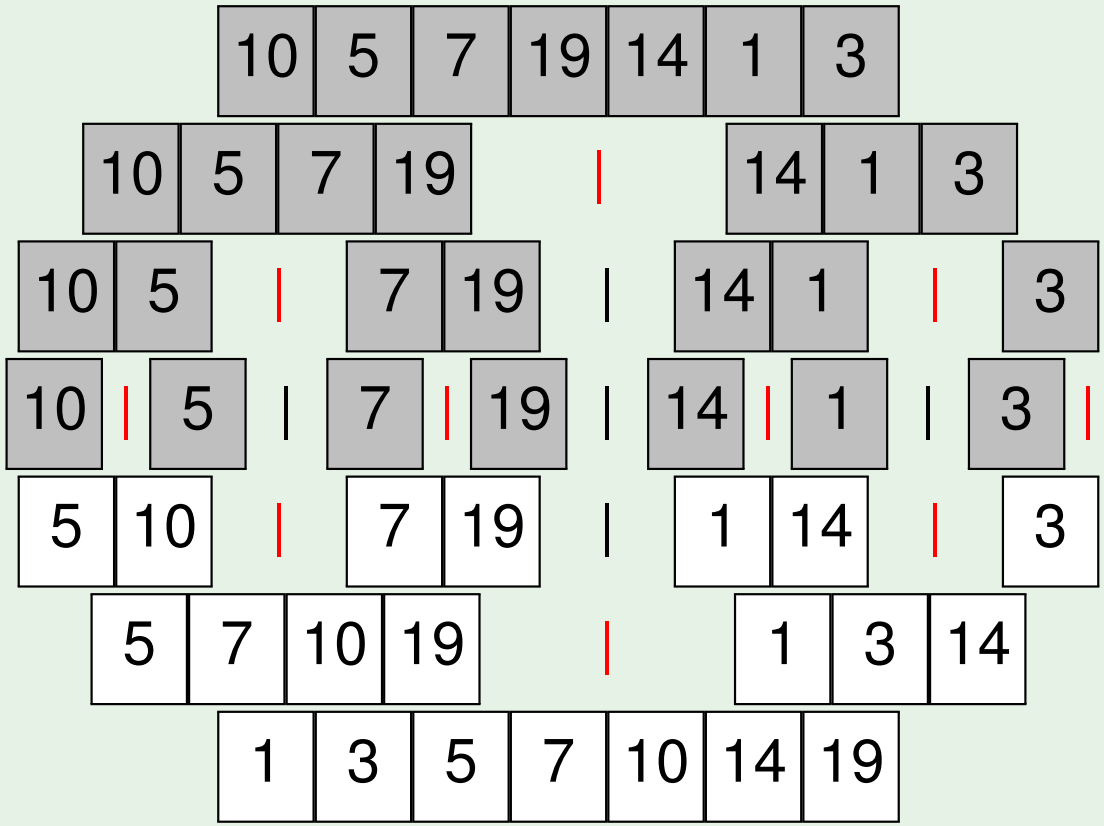
Übersicht

MergeSort

Sortieren durch Verschmelzen

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden

Beispiel



MergeSort

Sortieren durch Verschmelzen

```
mergeSort(Element[ ] a, int l, int r) {
  if (l == r) return;              // nur ein Element
  m = ⌊(r + l)/2⌋;                 // Mitte
  mergeSort(a, l, m);              // linken Teil sortieren
  mergeSort(a, m + 1, r);          // rechten Teil sortieren
  j = l;   k = m + 1;              // verschmelzen
  for i = 0 to r – l do
    if (j > m) { b[i] = a[k];  k++; }          // linker Teil leer
    else
      if (k > r) { b[i] = a[j];  j++; }        // rechter Teil leer
      else
        if (a[j] ≤ a[k]) { b[i] = a[j];  j++; }
        else { b[i] = a[k];  k++; }
  for i = 0 to r – l do  a[l + i] = b[i];      // zurückkopieren
}
```

MergeSort

Sortieren durch Verschmelzen

Beispiel (Verschmelzen)

10	5	7	19	|	14	1	3											
j→			m		k→								i→					
5	7	10	19	|	1	3	14	1										
5	7	10	19	|		3	14	1	3									
5	7	10	19	|			14	1	3	5								
	7	10	19	|			14	1	3	5	7							
		10	19	|			14	1	3	5	7	10						
			19	|			14	1	3	5	7	10	14					
			19	|				1	3	5	7	10	14	19				

MergeSort

Sortieren durch Verschmelzen

Zeitaufwand:

- $T(n)$: Laufzeit bei Feldgröße n
- $T(1) = \Theta(1)$
  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

$\Rightarrow$ $T(n) \in \mathcal{O}(n \log n)$
(folgt aus dem sogenannten Master-Theorem)

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- nichtrekursive Unterprogrammaufrufe sind einfach zu analysieren (separate Analyse des Funktionsaufrufs und Einsetzen)
- rekursive Aufrufstrukturen liefern Rekursionsgleichungen

$\Rightarrow$ Funktionswert wird in Abhängigkeit von Funktionswerten für kleinere Argumente bestimmt
gesucht: nichtrekursive / geschlossene Form

Anwendung: Divide-and-Conquer-Algorithmen

- gegeben: Problem der Größe $n = b^k$ ($k \in \mathbb{N}_0$)
- falls $k \geq 1$:
  - zerlege das Problem in d Teilprobleme der Größe n/b
  - löse die Teilprobleme (d rekursive Aufrufe)
  - setze aus den Teillösungen die Lösung zusammen
- falls $k = 0$ bzw. $n = 1$: investiere Aufwand a zur Lösung

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

- Betrachte den Aufwand für jede Rekursionstiefe
- Anfang: Problemgröße n
- Level für Rekursionstiefe i: d^i Teilprobleme der Größe n/b^i

$\Rightarrow$ Gesamtaufwand auf Rekursionslevel i:

$$d^i c \frac{n}{b^i} = cn\left(\frac{d}{b}\right)^i \qquad \text{(geometrische Reihe)}$$

$d < b$ Aufwand sinkt mit wachsender Rekursionstiefe; *erstes* Level entspricht konstantem Anteil des Gesamtaufwands

$d = b$ Gesamtaufwand für jedes Level gleich groß; maximale Rekursionstiefe logarithmisch, Gesamtaufwand $\Theta(n \log n)$

$d > b$ Aufwand steigt mit wachsender Rekursionstiefe; *letztes* Level entspricht konstantem Anteil des Gesamtaufwands

Analyse rekursiver Funktionen

Divide-and-Conquer-Algorithmen

Geometrische Folge: $(a_i)_{i \in \mathbb{N}}$

Verhältnis benachbarter Folgenglieder konstant: $q = a_{i+1}/a_i$

n-te Partialsumme der geometrischen Reihe:

$$s_n = \sum_{i=0}^{n} a_i = a_0 + \ldots + a_n = a_0 + a_0 q + a_0 q^2 + \ldots a_0 q^n$$

Wert:

$$s_n = a_0 \frac{q^{n+1} - 1}{q - 1} = a_0 \frac{q^{n+1} - 1}{q - 1} \quad \text{für } q \neq 1$$

bzw.

$$s_n = a_0(n+1) \quad \text{für } q = 1$$

Master-Theorem

Lösung von Rekursionsgleichungen

Satz (vereinfachtes Master-Theorem)

Seien a, b, c, d positive Konstanten und $n = b^k$ mit $k \in \mathbb{N}$.

Betrachte folgende Rekursionsgleichung:

$$r(n) = \begin{cases} a & \text{falls } n = 1, \\ cn + d \cdot r(n/b) & \text{falls } n > 1. \end{cases}$$

Dann gilt:

$$r(n) = \begin{cases} \Theta(n) & \text{falls } d < b, \\ \Theta(n \log n) & \text{falls } d = b, \\ \Theta(n^{\log_b d}) & \text{falls } d > b. \end{cases}$$

Übersicht

Untere Schranke

MergeSort hat Laufzeit $O(n \log n)$ im worst case.

insertionSort kann so implementiert werden, dass es im best case nur lineare Laufzeit hat

Gibt es Sortierverfahren mit Laufzeit besser als $O(n \log n)$ im worst case, z.B. $O(n)$ oder $O(n \log \log n)$?

$\Rightarrow$ nicht auf der Basis einfacher Schlüsselvergleiche

Entscheidungen: $x_i < x_j \quad \rightarrow \quad$ ja / nein

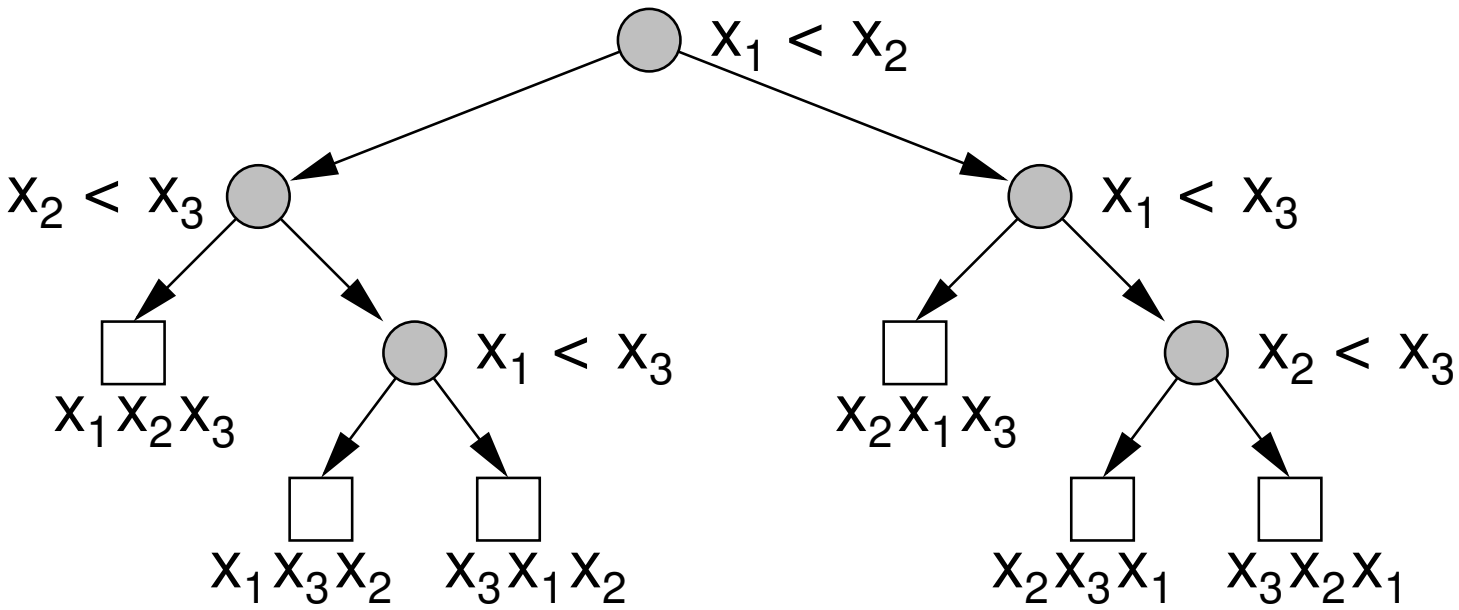
Satz

Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst case mindestens $n \log n - O(n) \in \Theta(n \log n)$ *Vergleiche.*

Untere Schranke

Vergleichsbasiertes Sortieren

Entscheidungsbaum mit Entscheidungen an den Knoten:



Untere Schranke

Vergleichsbasiertes Sortieren

muss insbesondere auch funktionieren, wenn alle n Schlüssel verschieden sind

$\Rightarrow$ Annahme: alle verschieden

Wieviele verschiedene Ergebnisse gibt es?
$\Rightarrow$ alle Permutationen:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+O\left(\frac{1}{n}\right)\right) \geq \frac{n^n}{e^n}\sqrt{2\pi n}$$

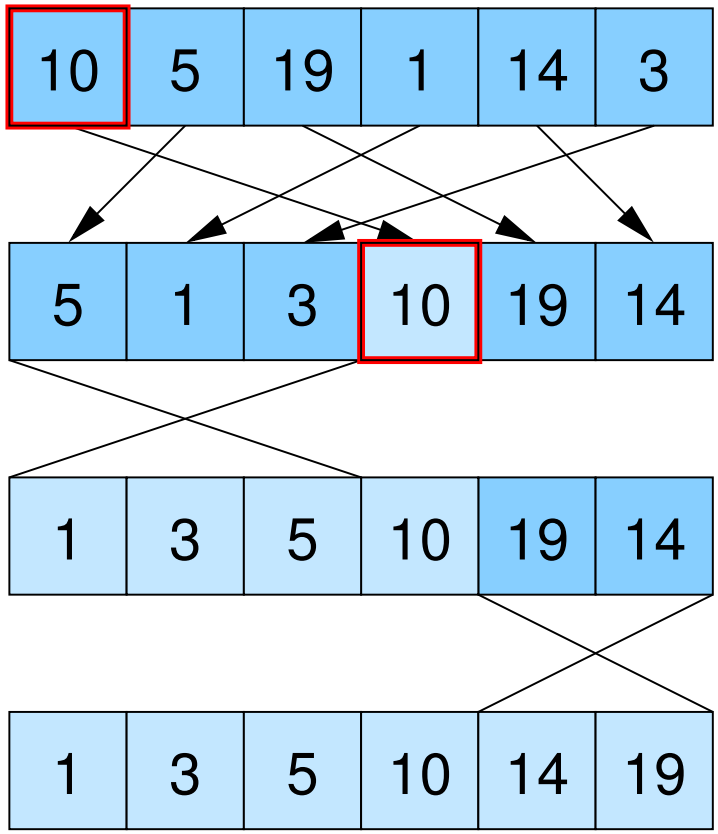
Binärbaum der Höhe h hat höchstens 2^h Blätter bzw.
Binärbaum mit b Blättern hat mindestens Höhe $\log_2 b$

$$\Rightarrow h \geq \log_2(n!) \geq n\log n - n\log e + \tfrac{1}{2}\log(2\pi n)$$

Übersicht

QuickSort

Idee:
Aufspaltung in zwei Teilmengen, aber nicht in der Mitte der Sequenz wie bei MergeSort, sondern getrennt durch ein Pivotelement



QuickSort: Algorithmus

```
quickSort(Element[ ] a, int ℓ, int r) {
    // a[ℓ ... r]: zu sortierendes Feld
    if (ℓ < r) {
        p = a[r];   // Pivot
        int i = ℓ – 1;   int j = r;
        do {   // spalte Elemente in a[ℓ, ..., r – 1] nach Pivot p
            do { i++ } while (a[i] < p);
            do { j– – } while (j ≥ ℓ  ∧  a[j] > p);
            if (i < j) swap(a[i], a[j]);
        } while (i < j);
        swap (a[i], a[r]);   // Pivot an richtige Stelle
        quickSort(a, ℓ, i – 1);
        quickSort(a, i + 1, r);
    }
}
```

QuickSort: Laufzeit

Problem:

- im Gegensatz zu MergeSort kann die Aufteilung in Teilprobleme unbalanciert sein (also nur im Optimalfall eine Halbierung)
- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)

Lösungen:

- wähle zufälliges Pivotelement:
  Laufzeit $O(n \log n)$ mit hoher Wahrscheinlichkeit
- berechne Median (mittleres Element):
  mit Selektionsalgorithmus, später in der Vorlesung

QuickSort

Laufzeit bei zufälligem Pivot-Element

- Zähle Anzahl Vergleiche (Rest macht nur konstanten Faktor aus)
- $\bar{C}(n)$: erwartete Anzahl Vergleiche bei n Elementen

Satz

Die erwartete Anzahl von Vergleichen für QuickSort ist

$$\bar{C}(n) \leq 2n \ln n \leq 1.39 n \log_2 n$$

QuickSort

Beweis.

- Betrachte sortierte Sequenz $\langle e'_1, \ldots, e'_n \rangle$
- nur Vergleiche mit Pivotelement
- Pivotelement ist nicht in den rekursiven Aufrufen enthalten

$\Rightarrow$ e'_i und e'_j werden höchstens einmal verglichen und zwar dann, wenn e'_i oder e'_j Pivotelement ist

QuickSort

Beweis.

- Zufallsvariable $X_{ij} \in \{0, 1\}$
- $X_{ij} = 1 \quad \Leftrightarrow \quad e'_i$ und e'_j werden verglichen

$$\begin{aligned}
\bar{C}(n) &= \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right] \\
&= \sum_{i<j} 0 \cdot \Pr\left[X_{ij} = 0\right] + 1 \cdot \Pr\left[X_{ij} = 1\right] \\
&= \sum_{i<j} \Pr\left[X_{ij} = 1\right]
\end{aligned}$$

QuickSort

Lemma

$$\Pr[X_{ij} = 1] = 2/(j - i + 1)$$

Beweis.

- Sei $M = \{e'_i, \ldots, e'_j\}$
- Irgendwann wird ein Element aus M als Pivot ausgewählt.
- Bis dahin bleibt M immer zusammen.
- e'_i und e'_j werden genau dann *direkt* verglichen, wenn eines der beiden als Pivot ausgewählt wird
- Wahrscheinlichkeit:

$$\Pr[e'_i \text{ oder } e'_j \text{ aus } M \text{ ausgewählt}] = \frac{2}{|M|} = \frac{2}{j - i + 1}$$

□

QuickSort

Beweis.

$$
\begin{aligned}
\bar{C} &= \sum_{i<j} \Pr\left[X_{ij} = 1\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\leq 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{1}{k} = 2(n-1) \sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n - 1) \\
&\leq 2(n-1)(1 + \ln n - 1) \leq 2n \ln n = 2n \ln(2) \log_2(n)
\end{aligned}
$$

□

QuickSort

Verbesserte Version ohne Check für Array-Grenzen

```
qSort(Element[ ] a, int ℓ, int r) {
    while (r − ℓ ≥ n0) {
        j = pickPivotPos(a, ℓ, r);
        swap(a[ℓ],a[j]);      p = a[ℓ];
        int i = ℓ;      int j = r;
        repeat {
            while (a[i] < p) do i++;
            while (a[j] > p) do j−−;
            if (i ≤ j) { swap(a[i], a[j]);   i++;   j−−; }
        } until (i > j);
        if (i < (ℓ + r)/2) { qSort(a, ℓ, j);   ℓ = i; }
        else               { qSort(a, i, r);   r = j; }
    }
    insertionSort(a, ℓ, r);
}
```

Übersicht

Rang-Selektion

- Bestimmung des kleinsten und größten Elements ist mit einem einzigen Scan über das Array in Linearzeit möglich
- Aber wie ist das beim k-kleinsten Element, z.B. beim $\lfloor n/2 \rfloor$-kleinsten Element (Median)?

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

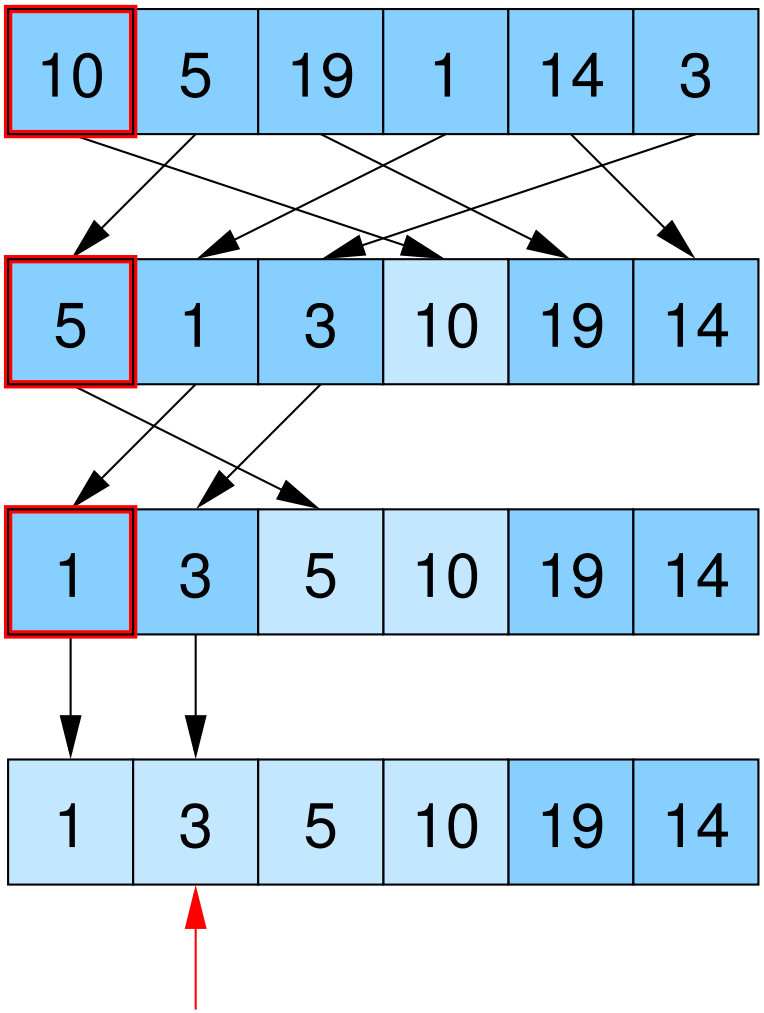
Naive Lösung: Sortieren und k-tes Element ausgeben

$\Rightarrow$ Zeit $O(n \log n)$

Geht das auch schneller?

QuickSelect

Ansatz: ähnlich zu QuickSort, aber nur eine Seite betrachten



QuickSelect

Methode analog zu QuickSort

```
Element quickSelect(Element[ ] a, int l, int r, int k) {
  // a[l...r]: Restfeld,    k: Rang des gesuchten Elements
  if (r == l) return a[l];
  int z = zufällige Position in {l,...,r};   swap(a[z], a[r]);
  Element p = a[r];   int i = l – 1;   int j = r;
  do {   // spalte Elemente in a[l,...,r – 1] nach Pivot p
    do i++ while (a[i] < p);
    do j– – while (a[j] > p && j ≠ l);
    if (i < j) swap(a[i], a[j]);
  } while (i < j);
  swap (a[i], a[r]);   // Pivot an richtige Stelle
  if (k < i) return quickSelect(a, l, i – 1, k);
  if (k > i) return quickSelect(a, i + 1, r, k);
  else return a[k];   // k == i
}
```

QuickSelect

Alternative Methode

```
Element select(Element[ ] s, int k) {
   assert(|s| ≥ k);
   Wähle p ∈ s zufällig (gleichverteilt);

   Element[ ] a := {e ∈ s : e < p};
   if (|a| ≥ k)
      return select(a,k);

   Element[ ] b := {e ∈ s : e = p};
   if (|a| + |b| ≥ k)
      return p;

   Element[ ] c := {e ∈ s : e > p};
   return select(c,k – |a| – |b|);
}
```

QuickSelect

Alternative Methode

Beispiel

s	k	a b c
$\langle 3,1,4,1,5,9,2,6,5,3,5,8,9\rangle$	7	$\langle 1,1\rangle\ \langle 2\rangle\ \langle \underline{3,4,5,9,6,5,3,5,8,9}\rangle$
$\langle 3,4,5,9,6,5,3,5,8,9\rangle$	4	$\langle \underline{3,4,5,5,3,5}\rangle\ \langle 6\rangle\ \langle 9,8,9\rangle$
$\langle 3,4,5,5,3,5\rangle$	4	$\langle 3,4,3\rangle\ \langle \underline{5,5,5}\rangle\ \langle\rangle$

In der sortierten Sequenz würde also an 7. Stelle das Element 5 stehen.

Hier wurde das mittlere Element als Pivot verwendet.

QuickSelect

teilt das Feld jeweils in 3 Teile:

- a Elemente kleiner als das Pivot
- b Elemente gleich dem Pivot
- c Elemente größer als das Pivot

$T(n)$: erwartete Laufzeit bei n Elementen

Satz

Die erwartete Laufzeit von QuickSelect ist linear: $T(n) \in \mathcal{O}(n)$.

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als $2/3$ der aktuellen Feldgröße sind:

$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

Erwartete Zeit bei n Elementen

- linearer Aufwand außerhalb der rekursiven Aufrufe: cn
- Pivot gut (Wsk. $1/3$): Restaufwand $\leq T(2n/3)$
- Pivot schlecht (Wsk. $2/3$): Restaufwand $\leq T(n-1) < T(n)$

QuickSelect

Beweis.

$$
\begin{aligned}
T(n) &\leq cn + p \cdot T(n \cdot 2/3) + (1-p) \cdot T(n) \\
p \cdot T(n) &\leq cn + p \cdot T(n \cdot 2/3) \\
T(n) &\leq cn/p + T(n \cdot 2/3) \\
&\leq cn/p + c \cdot (n \cdot 2/3)/p + T(n \cdot (2/3)^2) \\
&\ldots \quad \text{wiederholtes Einsetzen} \\
&\leq (cn/p)(1 + 2/3 + 4/9 + 8/27 + \ldots) \\
&\leq \frac{cn}{p} \cdot \sum_{i \geq 0} (2/3)^i \\
&\leq \frac{cn}{1/3} \cdot \frac{1}{1-2/3} = 9cn \in \mathcal{O}(n)
\end{aligned}
$$

□

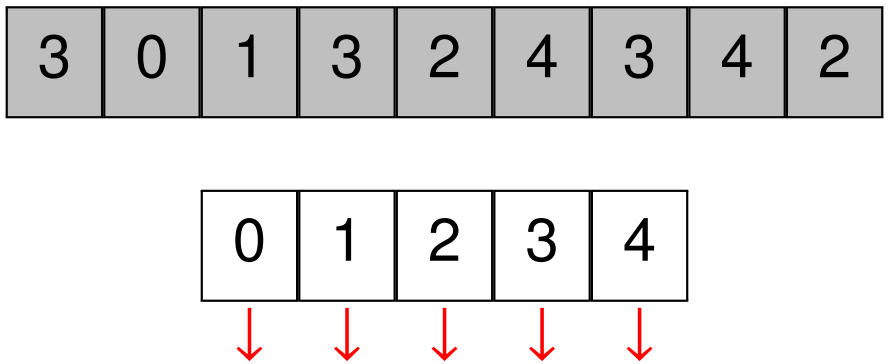
Übersicht

Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $\mathcal{O}(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
- z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer bzw. das erste Zeichen. Nur bei Gleichheit des Anfangs kommt es auf weitere Ziffern / Zeichen an.

- Annahme: Elemente sind Zahlen im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)



Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
   Sequence<Elem>[ ] b = new Sequence<Elem>[K];
   foreach (e ∈ s)
      b[key(e)].pushBack(e);
   return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n + K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n + K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

- verwende K-adische Darstellung der Schlüssel
- Annahme:
  Schlüssel sind Zahlen aus $\{0, \ldots, K^d - 1\}$
  repräsentiert durch d Ziffern aus $\{0, \ldots, K - 1\}$
- sortiere zunächst entsprechend der niedrigstwertigen Ziffer mit kSort und dann nacheinander für immer höherwertigere Stellen
- behalte Ordnung der Teillisten bei

RadixSort

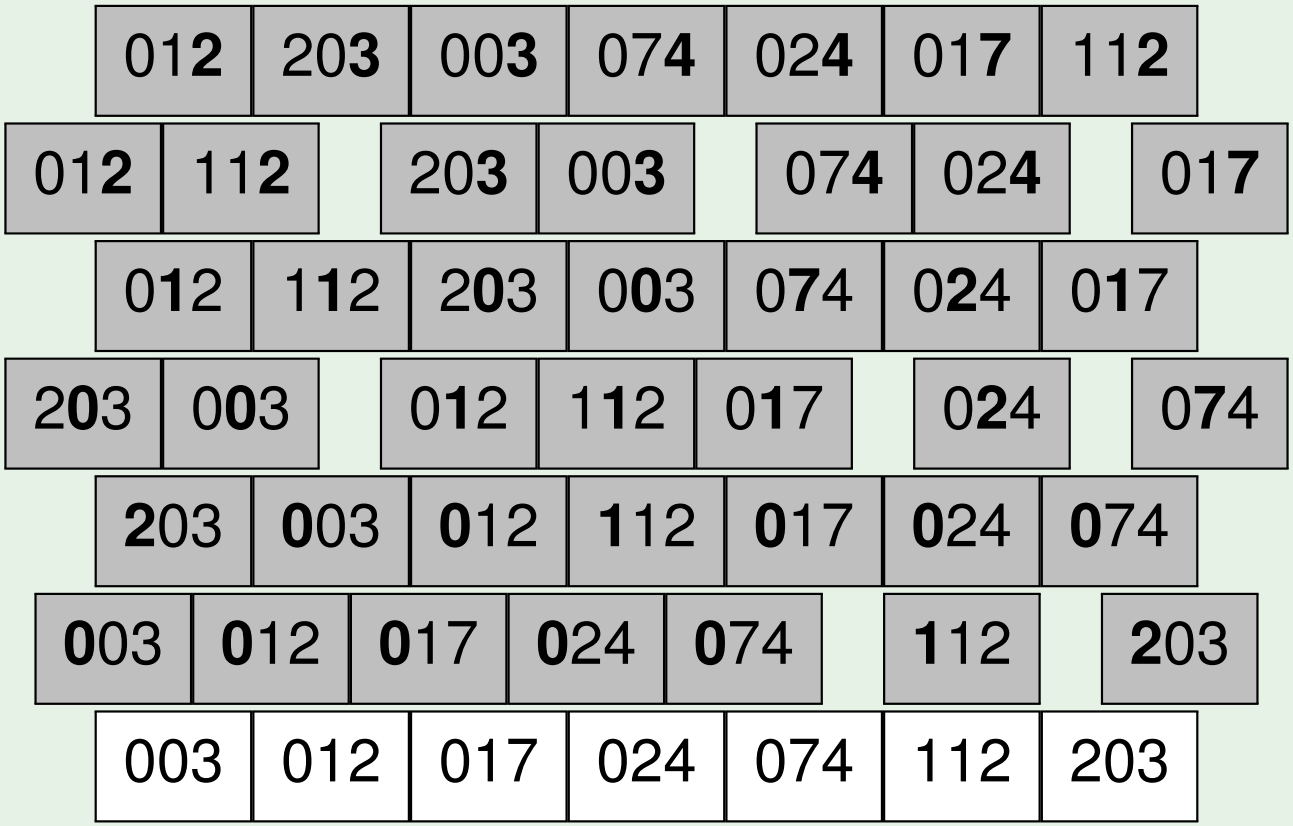
```
radixSort(Sequence<Elem> s) {
    for (int i = 0; i < d; i++)
        kSort(s,i);   // sortiere gemäß key_i(x)
        // mit key_i(x) = (key(x)/K^i) mod K
}
```

Verfahren funktioniert, weil kSort stabil ist:
Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1 \ldots 0$ während der Sortierung nach Ziffer i

Laufzeit: $\mathcal{O}(d(n+K))$ für n Schlüssel aus $\{0,\ldots,K^d-1\}$

RadixSort

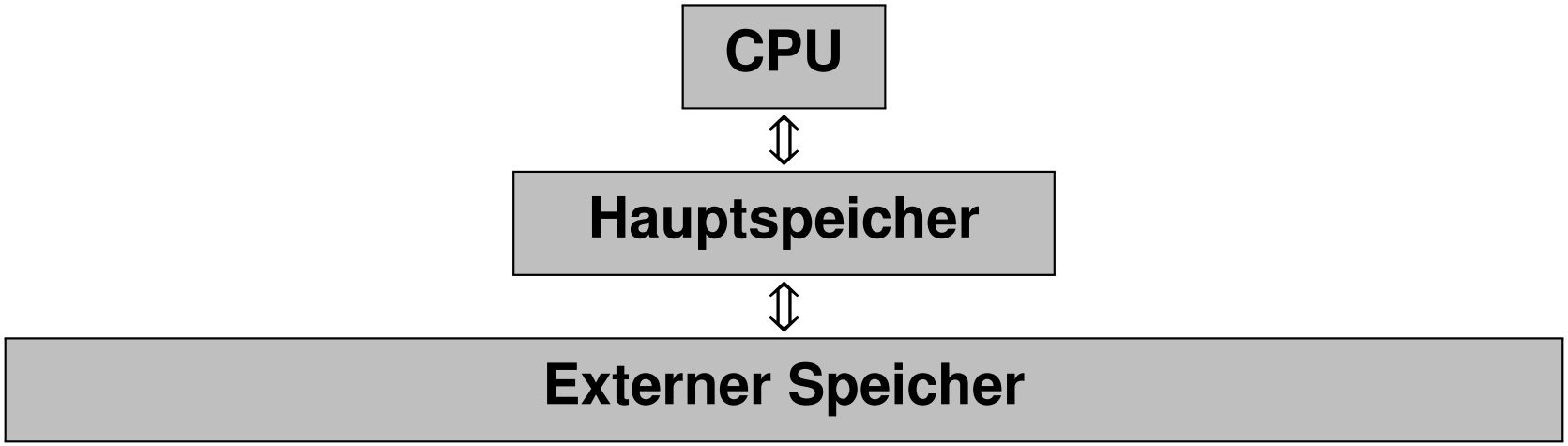
Beispiel



Übersicht

Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Lösung: Verwende MergeSort

Vorteil: MergeSort verwendet oft konsekutive Elemente (Scanning)
(geht auf Festplatte schneller als Random Access-Zugriffe)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar (sonst z.B. Auffüllen mit maximalem Schlüssel)

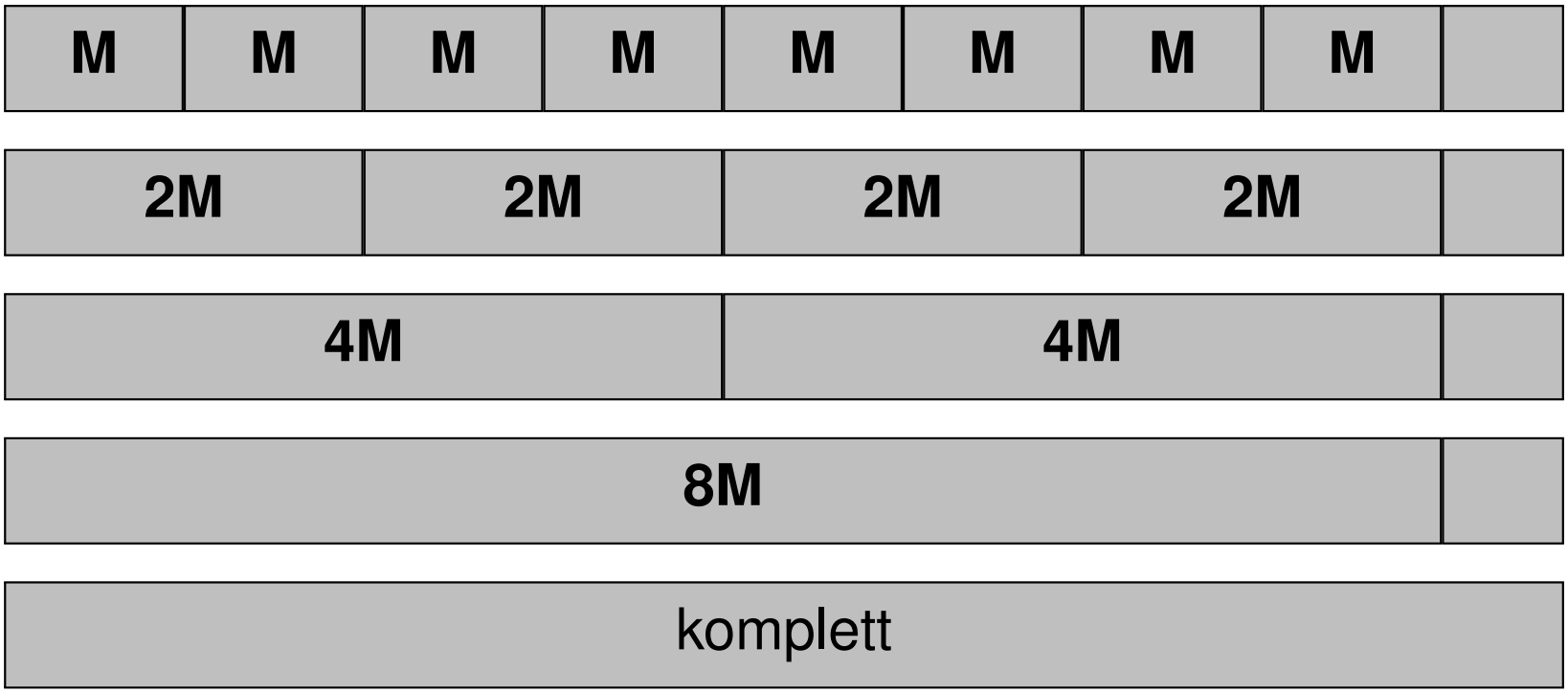
Run Formation Phase:

- Lade Teilfelder der Größe M in den Speicher (Annahme: $B \mid M$),
- sortiere sie mit einem in-place-Sortierverfahren,
- schreibe das sortierte Teilfeld wieder zurück auf die Festplatte
- benötigt n/B Blocklese- und n/B Blockschreiboperationen
- Laufzeit: 2n/B Transfers
- ergibt sortierte Bereiche (Runs) der Größe M

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $O(\lceil \log(n/M) \rceil)$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

$\Rightarrow$ $(2n/B)(1 + \lceil \log_2(n/M) \rceil)$ Block-Transfers

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich mehr als zwei Runs (k) mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow$ $(k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

$\Rightarrow$ $(2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

Übersicht

Übersicht

Prioritätswarteschlangen

M: Menge von Elementen

prio(e): Priorität von Element e

Operationen:

- M.build($\{e_1, \ldots, e_n\}$): $M = \{e_1, \ldots, e_n\}$
- M.insert(Element e): $M = M \cup e$
- Element M.min(): gib ein e mit minimaler Priorität prio(e) zurück
- Element M.deleteMin():
  entferne Element e mit minimalem Wert prio(e)
  und gib es zurück

Adressierbare Prioritätswarteschlangen

Zusätzliche Operationen für adressierbare Priority Queues:

- Handle insert(Element e): wie zuvor, gibt aber ein Handle (Referenz / Zeiger) auf das eingefügte Element zurück
- remove(Handle h): lösche Element spezifiziert durch Handle h
- decreaseKey(Handle h, int k):
  reduziere Schlüssel / Priorität des Elements auf Wert k
  (je nach Implementation evt. auch um Differenz k)
- M.merge(Q): $M = M \cup Q$; $Q = \emptyset$;

Prioritätswarteschlangen mit Listen

Priority Queue mittels unsortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $\mathcal{O}(n)$
- insert(Element e): Zeit $\mathcal{O}(1)$
- min(), deleteMin(): Zeit $\mathcal{O}(n)$

Priority Queue mittels sortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $\mathcal{O}(n \log n)$
- insert(Element e): Zeit $\mathcal{O}(n)$
- min(), deleteMin(): Zeit $\mathcal{O}(1)$

$\Rightarrow$ Bessere Struktur als eine Liste notwendig!

Übersicht

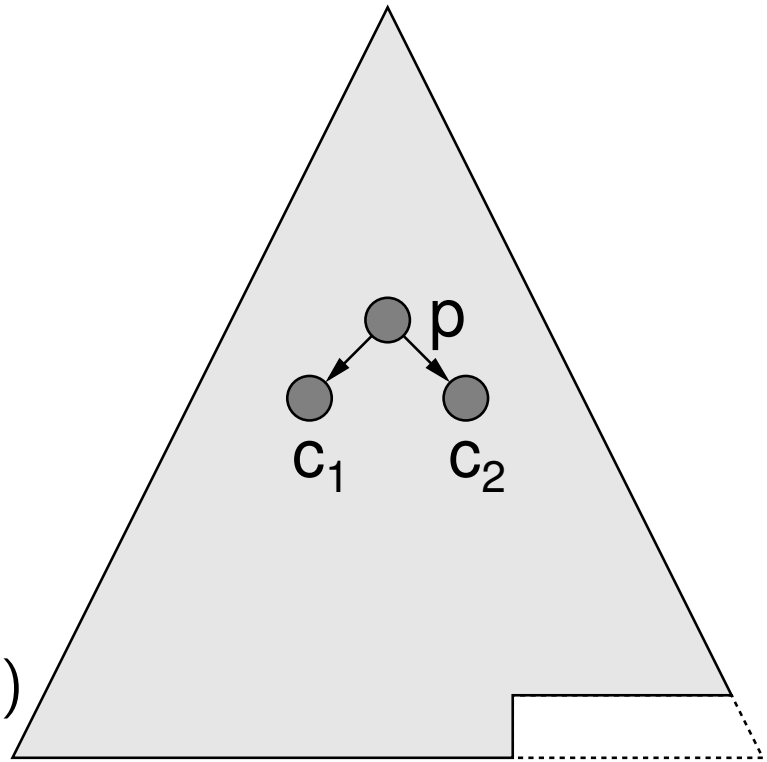
Binärer Heap

Idee: verwende Binärbaum

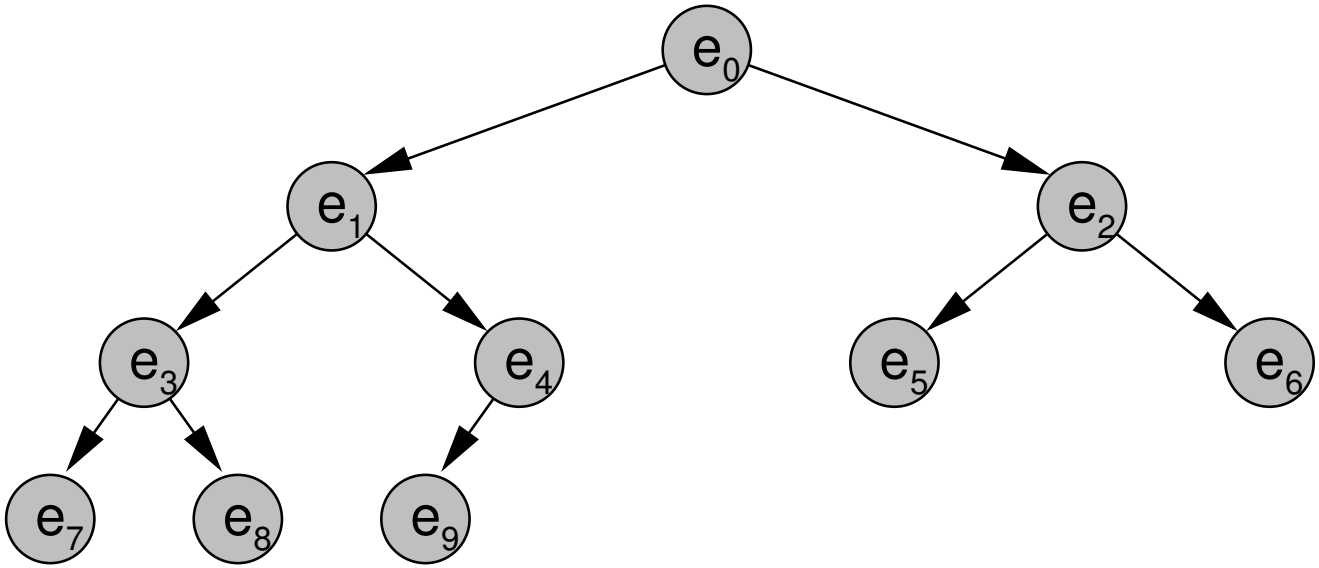
Bewahre zwei Invarianten:

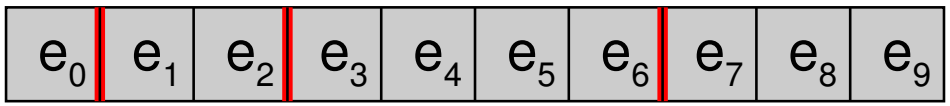
- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

$$\text{prio}(p) \leq \min\{\text{prio}(c1), \text{prio}(c2)\})$$



Binärer Heap als Feld





- Kinder von Knoten $H[i]$ in $H[2i+1]$ und $H[2i+2]$
- Form-Invariante: $H[0] \ldots H[n-1]$ besetzt
- Heap-Invariante: $H[i] \leq \min\{H[2i+1], H[2i+2]\}$

Binärer Heap als Feld

insert(e)

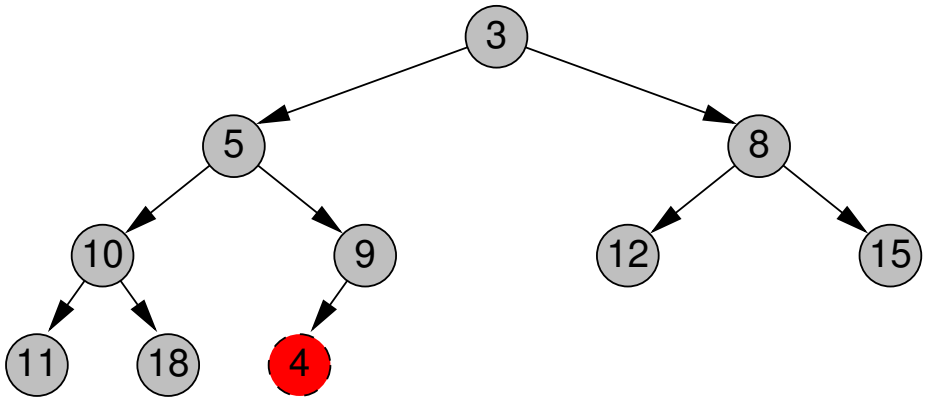
- Form-Invariante: $H[n] = e;$ siftUp(n); n++;
- Heap-Invariante:

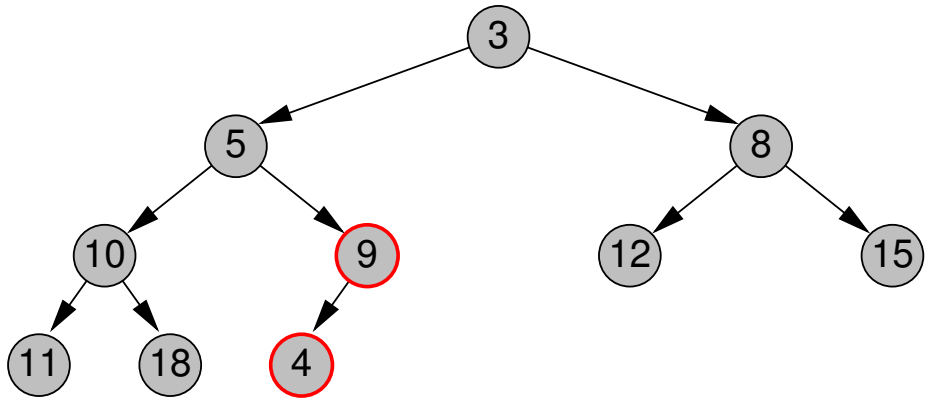
  vertausche e mit seinem Vater bis
  prio($H[\lfloor (k-1)/2 \rfloor]$) $\leq$ prio(e) für e in $H[k]$ (oder e in $H[0]$)

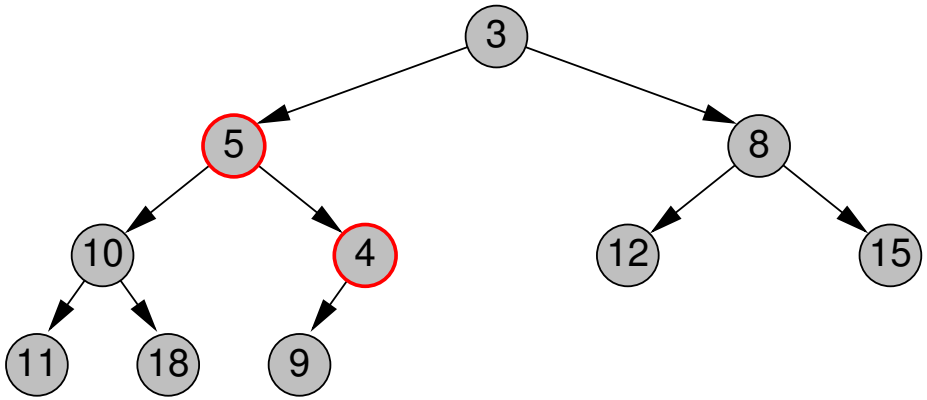
  siftUp(i) {
  while ($i > 0 \ \wedge$ prio($H[\lfloor (i-1)/2 \rfloor]$) $>$ prio($H[i]$)) {
  swap($H[i]$, $H[\lfloor (i-1)/2 \rfloor]$);
  $i = (i-1)/2;$
  }
  }

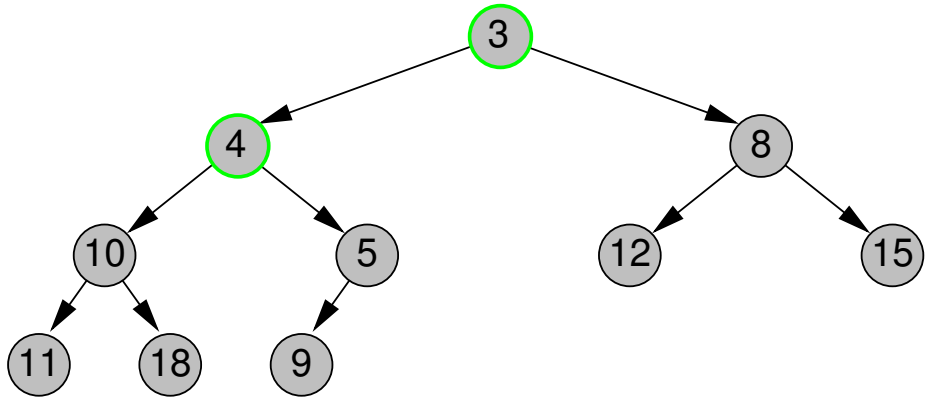
- Laufzeit: $O(\log n)$

Heap - siftUp()


3
5
8
10
9
12
15
11
18
4


3
5
8
10
9
12
15
11
18
4


3
5
8
10
4
12
15
11
18
9


3
4
8
10
5
12
15
11
18
9

Binärer Heap als Feld

deleteMin()

- Form-Invariante:
  $e = H[0]$;
  $n--$;
  $H[0] = H[n]$;
  siftDown(0);
  return e;

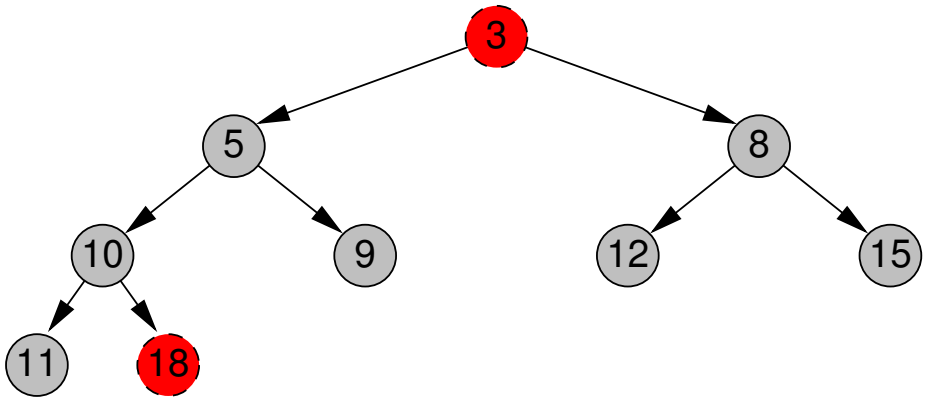
- Heap-Invariante: (siftDown)
  vertausche e (anfangs Element in $H[0]$) mit dem Kind, das die kleinere Priorität hat, bis e ein Blatt ist oder
  $\text{prio}(e) \leq \min\{\text{prio}(c_1(e)), \text{prio}(c_2(e))\}$.

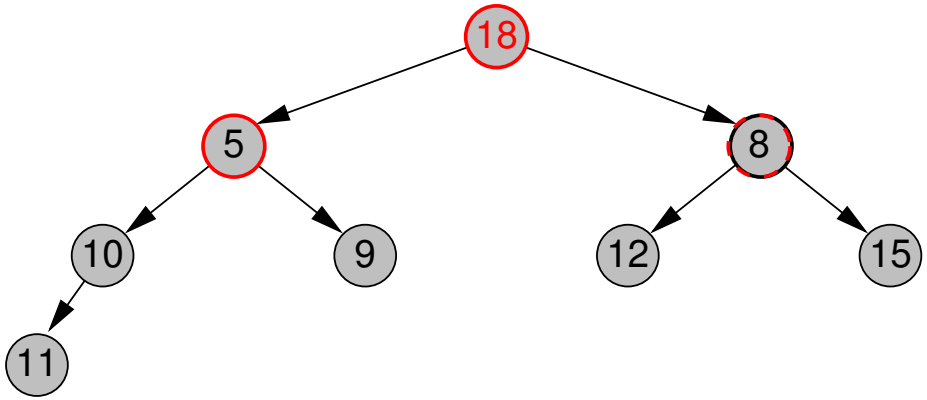
- Laufzeit: $O(\log n)$

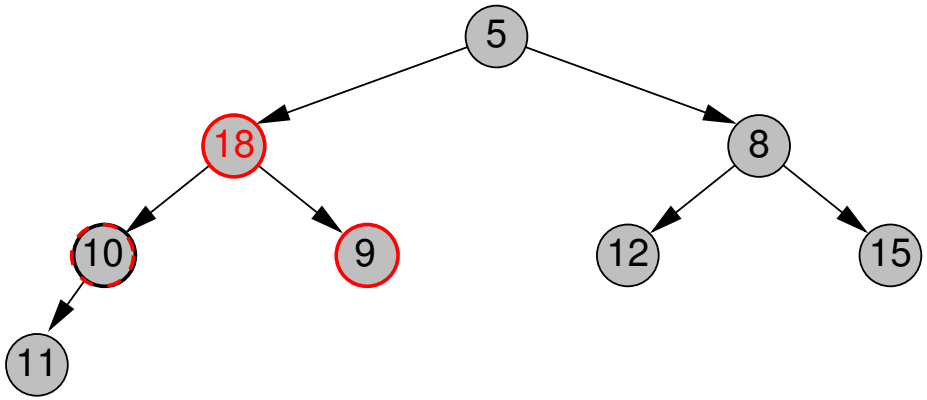
Binärer Heap als Feld

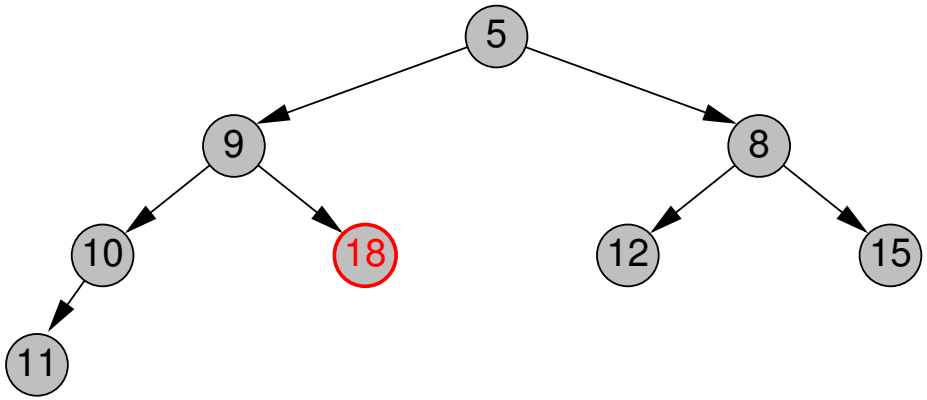
```
siftDown(i) {
    int m;
    while (2i + 1 < n) {
        if (2i + 2 ≥ n)
            m = 2i + 1;
        else
            if (prio(H[2i + 1]) < prio(H[2i + 2]))
                m = 2i + 1;
            else m = 2i + 2;
        if (prio(H[i]) ≤ prio(H[m]))
            return;
        swap(H[i], H[m]);
        i = m;
    }
}
```

Heap - siftDown()









Binärer Heap / Aufbau

build($\{e_0, \ldots, e_{n-1}\}$)

- naiv:

  Für alle $i \in \{0, \ldots, n-1\}$:
  insert(e_i)

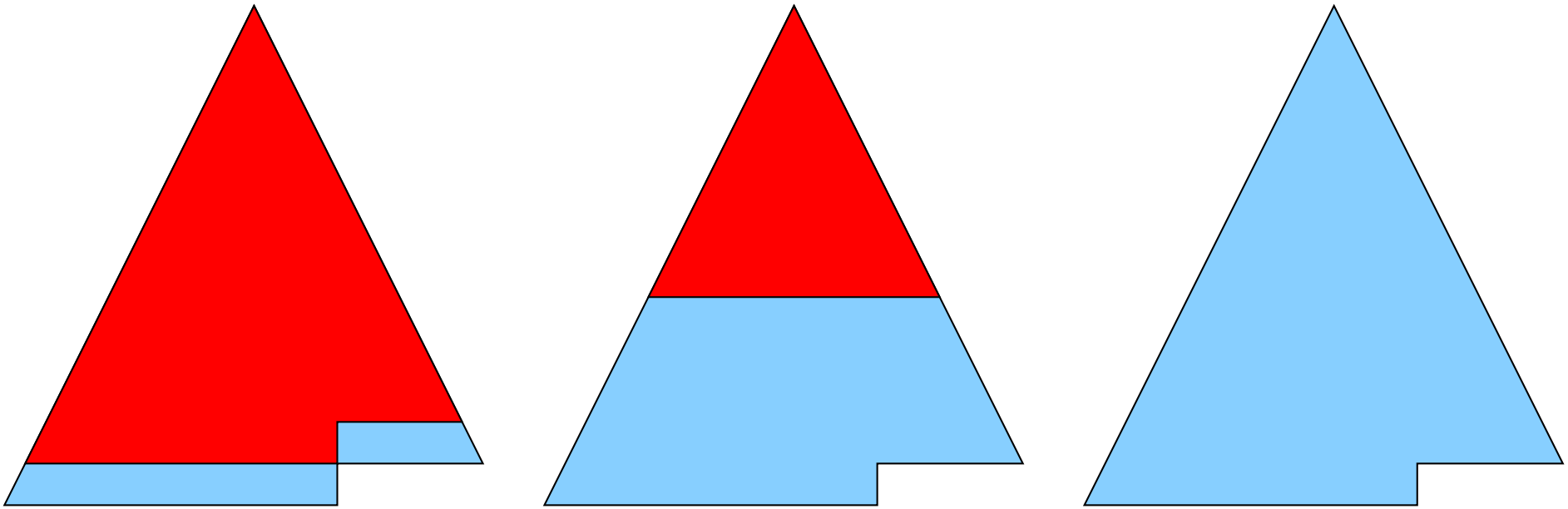
$\Rightarrow$ Laufzeit: $\Theta(n \log n)$

Binärer Heap / Aufbau

build($\{e_0, \ldots, e_{n-1}\}$)

effizient:

- Für alle $i \in \{0, \ldots, n-1\}$:
  $H[i] := e_i$.
- Für alle $i \in \left\{\left\lfloor \frac{n}{2} \right\rfloor - 1, \ldots, 0\right\}$:
  siftDown(i)



Binärer Heap / Aufbau

Laufzeit:

- $k = \lfloor \log n \rfloor$: Baumtiefe (gemessen in Kanten)
- siftDown-Kosten von Level ℓ aus proportional zur Resttiefe $(k - \ell)$
- Es gibt $\leq 2^\ell$ Knoten in Tiefe ℓ.

$$O\left(\sum_{0 \leq \ell < k} 2^\ell (k - \ell)\right) \subseteq O\left(2^k \sum_{0 \leq \ell < k} \frac{k - \ell}{2^{k-\ell}}\right) \subseteq O\left(2^k \sum_{j \geq 1} \frac{j}{2^j}\right) \subseteq O(n)$$

$$\begin{aligned}
\sum_{j \geq 1} j \cdot 2^{-j} &= \sum_{j \geq 1} 2^{-j} + \sum_{j \geq 2} 2^{-j} + \sum_{j \geq 3} 2^{-j} + \ldots \\
&= 1 \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{2} \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{4} \cdot \sum_{j \geq 1} 2^{-j} + \ldots \\
&= (1 + 1/2 + 1/4 + \ldots) \sum_{j \geq 1} 2^{-j} = 2 \cdot 1 = 2
\end{aligned}$$

Laufzeiten des Binären Heaps

- min(): $\mathcal{O}(1)$
- insert(e): $\mathcal{O}(\log n)$
- deleteMin(): $\mathcal{O}(\log n)$
- build($e_0, \ldots, e_{n-1}$): $\mathcal{O}(n)$
- M.merge(Q): $\Theta(n)$

Adressen bzw. Feldindizes in array-basierten Binärheaps können nicht als Handles verwendet werden, da die Elemente bei den Operationen verschoben werden
$\Rightarrow$ ungeeignet als adressierbare PQs (kein remove bzw. decreaseKey)

HeapSort

Verbesserung von SelectionSort:

- erst build($e_0, \ldots, e_{n-1}$): $\mathcal{O}(n)$
- dann $n \times$ deleteMin():
  vertausche in jeder Runde erstes und letztes Heap-Element,
  dekrementiere Heap-Größe und führe siftDown(0) durch:
  $\mathcal{O}(n \log n)$

$\Rightarrow$ sortiertes Array entsteht von hinten,
ansteigende Sortierung kann mit Max-Heap erzeugt werden

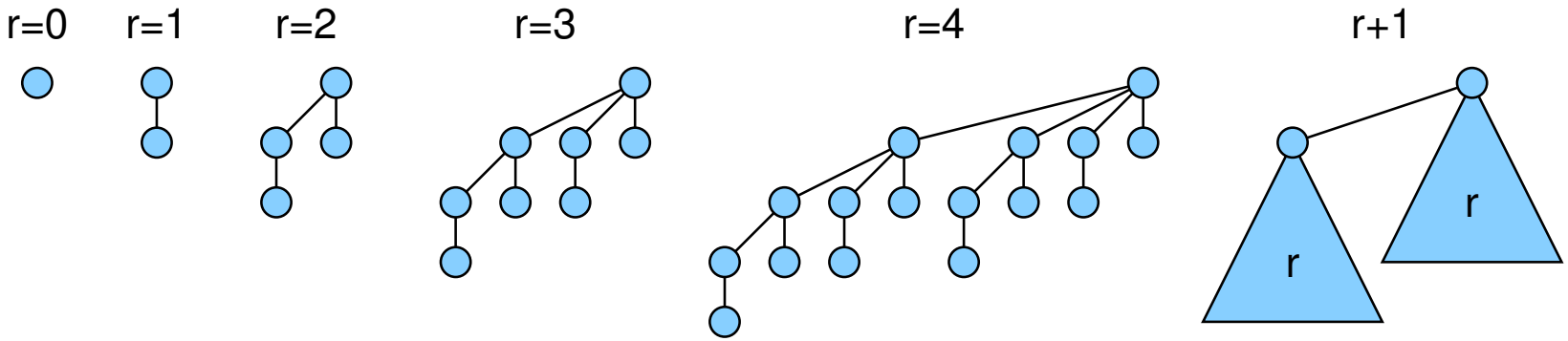
- in-place, aber nicht stabil
- Gesamtlaufzeit: $\mathcal{O}(n \log n)$

Übersicht

Binomial-Bäume

Binomial Heaps bestehen aus Binomial-Bäumen

- Form-Invariante:



- Heap-Invariante:

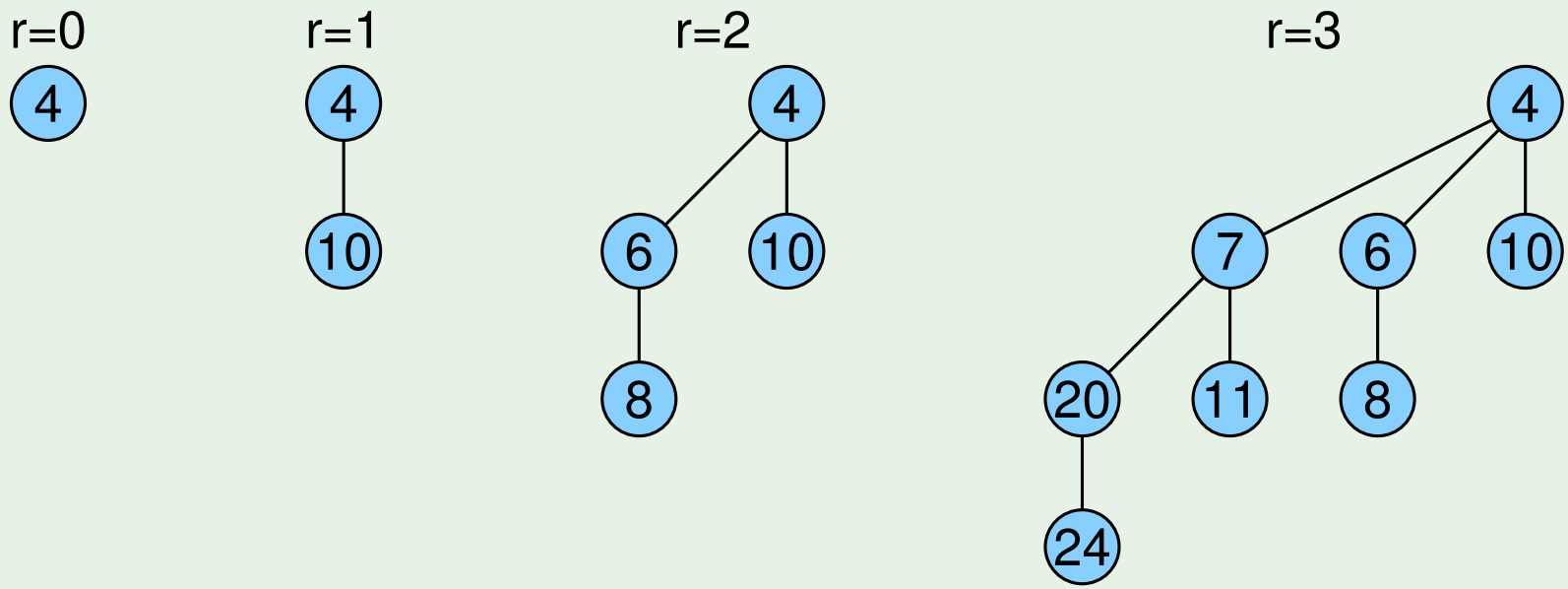
$$\text{prio(Vater)} \leq \text{prio(Kind)}$$

Elemente der Priority Queue werden in Heap Items gespeichert, die eine feste Adresse im Speicher haben und damit als Handles dienen können (im Gegensatz zu array-basierten Binärheaps)

Binomial-Bäume

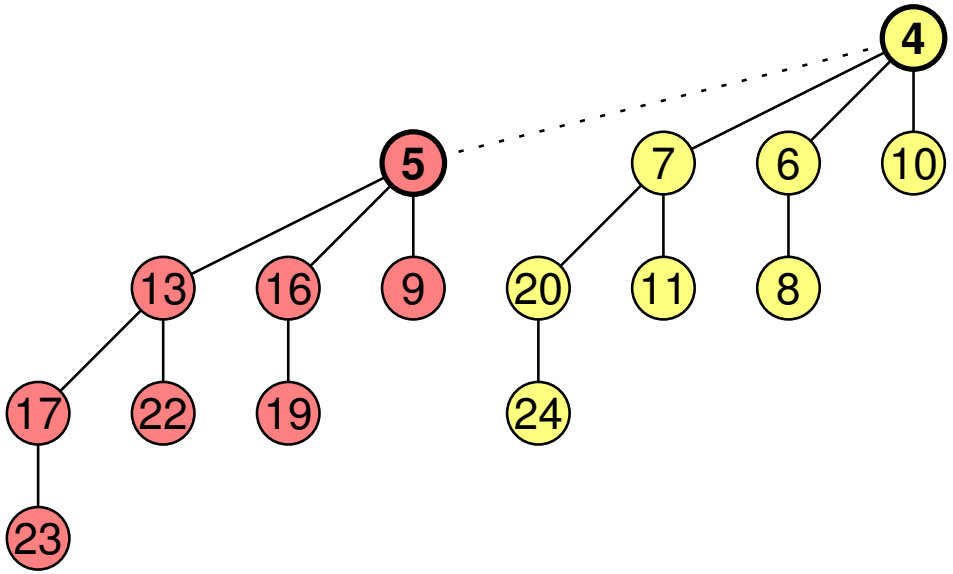
Beispiel

Korrekte Binomial-Bäume:

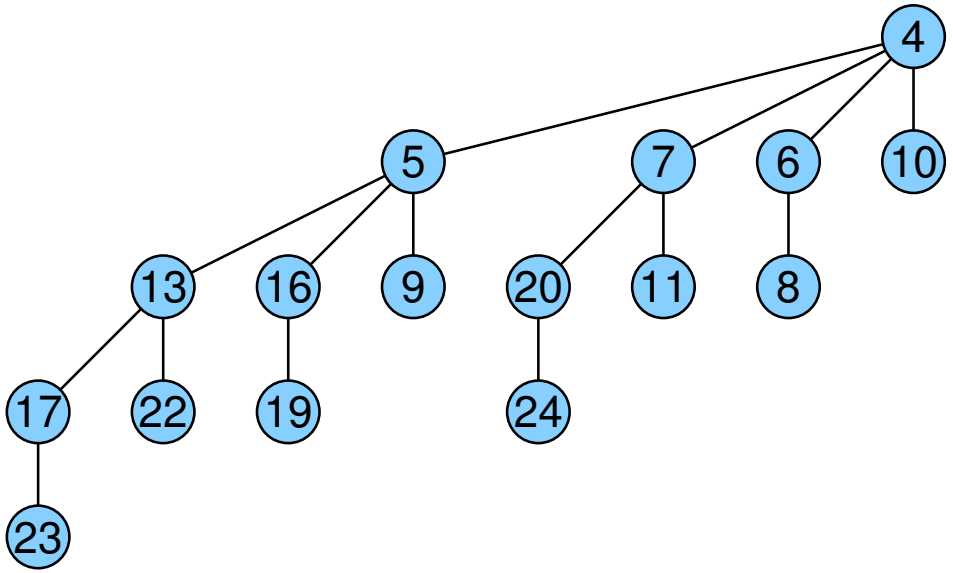


Binomial-Baum: Merge

Wurzel mit größerem Wert wird neues Kind der Wurzel mit kleinerem Wert! (Heap-Bedingung)

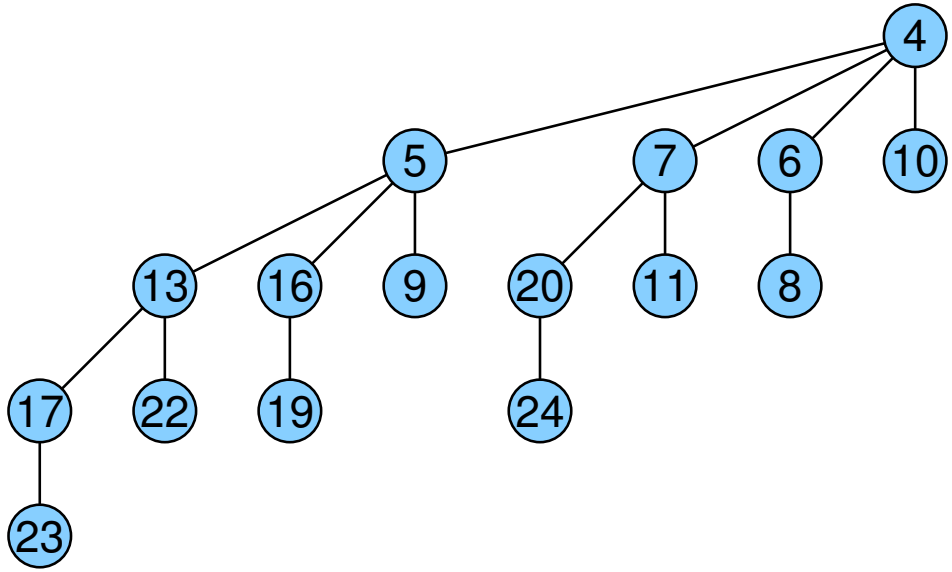


aus zwei B_{r-1} wird ein B_r

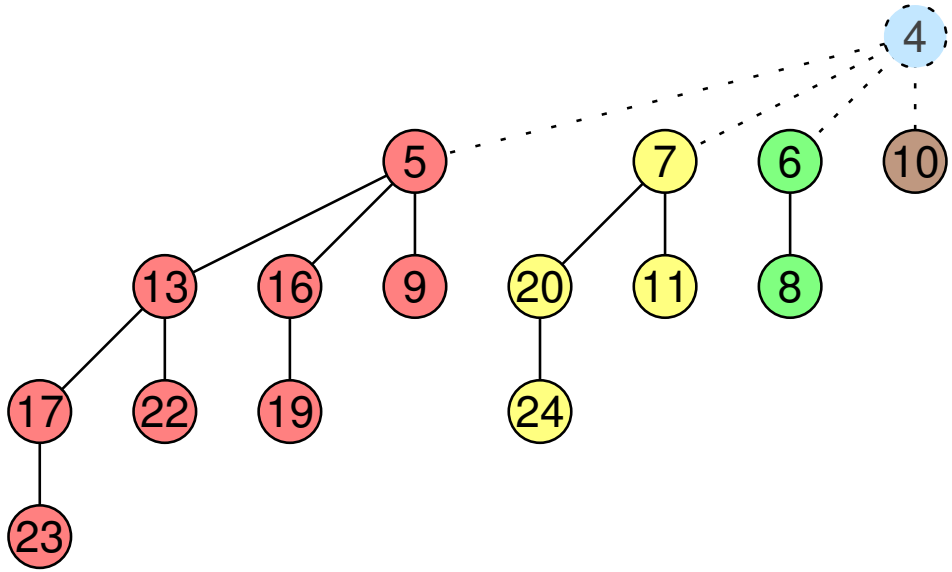


Binomial-Baum: Löschen der Wurzel (deleteMin)

aus einem B_r



werden $B_{r-1}, \ldots, B_0$



Binomial-Baum: Knotenanzahl

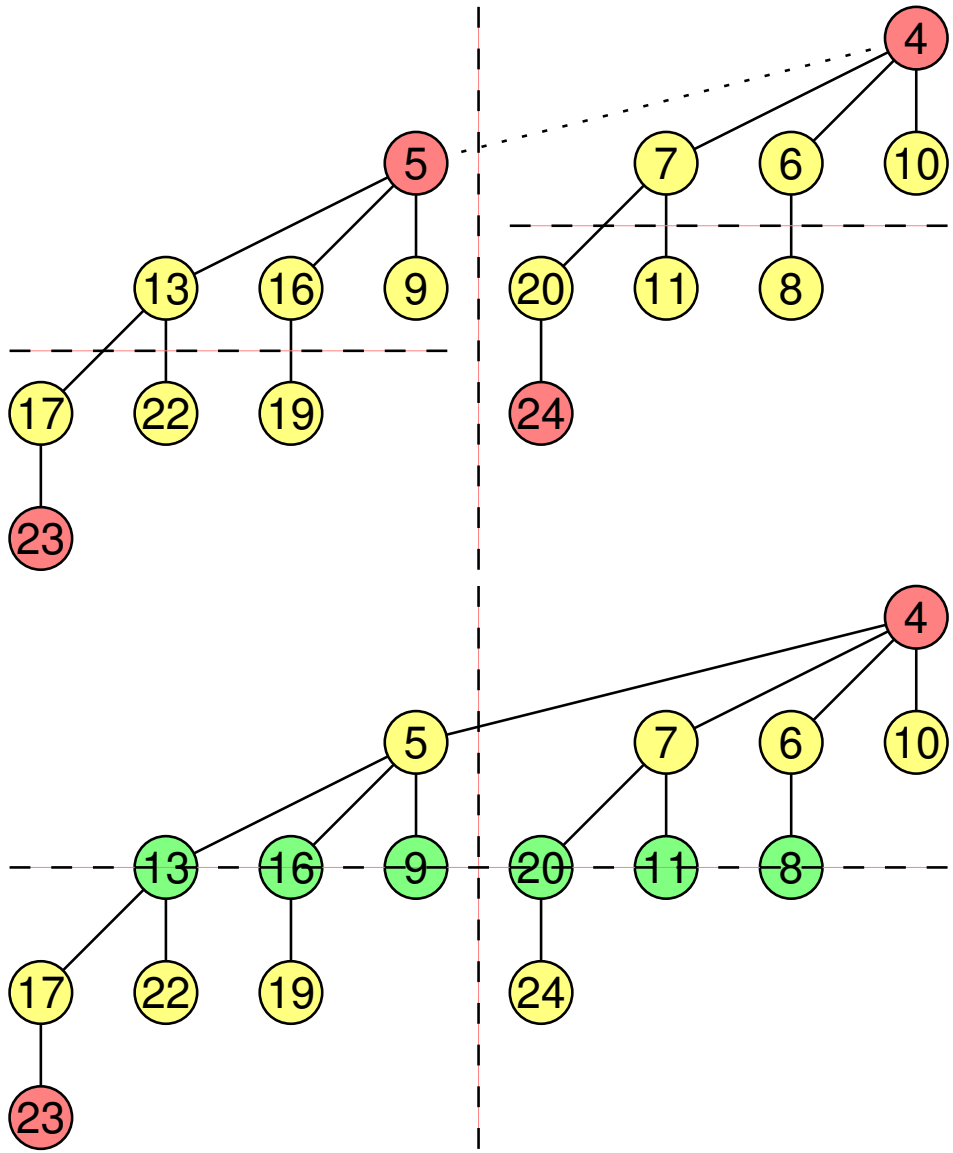
B_r hat auf Level $k \in \{0, \ldots, r\}$ genau $\binom{r}{k}$ Knoten

Warum?

Bei Bau des B_r aus 2 B_{r-1} gilt:

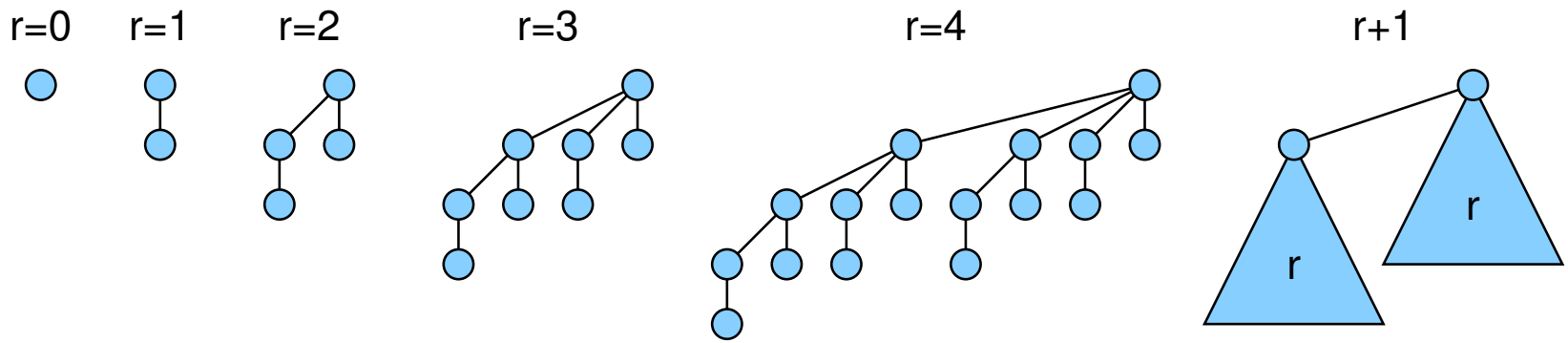
$$\binom{r}{k} = \binom{r-1}{k-1} + \binom{r-1}{k}$$

Insgesamt: B_r hat 2^r Knoten



Binomial-Bäume

Eigenschaften von Binomial-Bäumen:



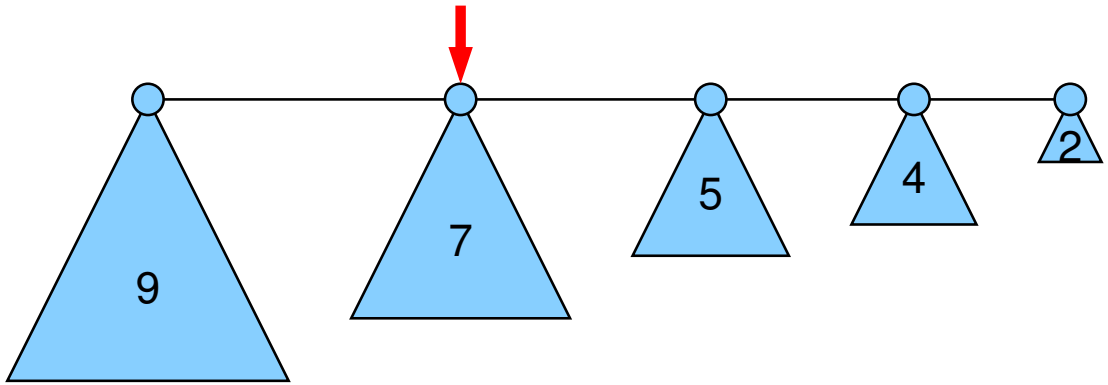
Binomial-Baum vom Rang r

- hat Höhe r (gemessen in Kanten)
- hat maximalen Grad r (Wurzel)
- hat auf Level $\ell \in \{0, \ldots, r\}$ genau $\binom{r}{\ell}$ Knoten
- hat $\sum_{\ell=0}^{r} \binom{r}{\ell} = 2^r$ Knoten
- zerfällt bei Entfernen der Wurzel in r Binomial-Bäume von Rang 0 bis $r - 1$

Binomial Heap

Binomial Heap:

- verkettete Liste von Binomial-Bäumen
- pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem Prioritätswert

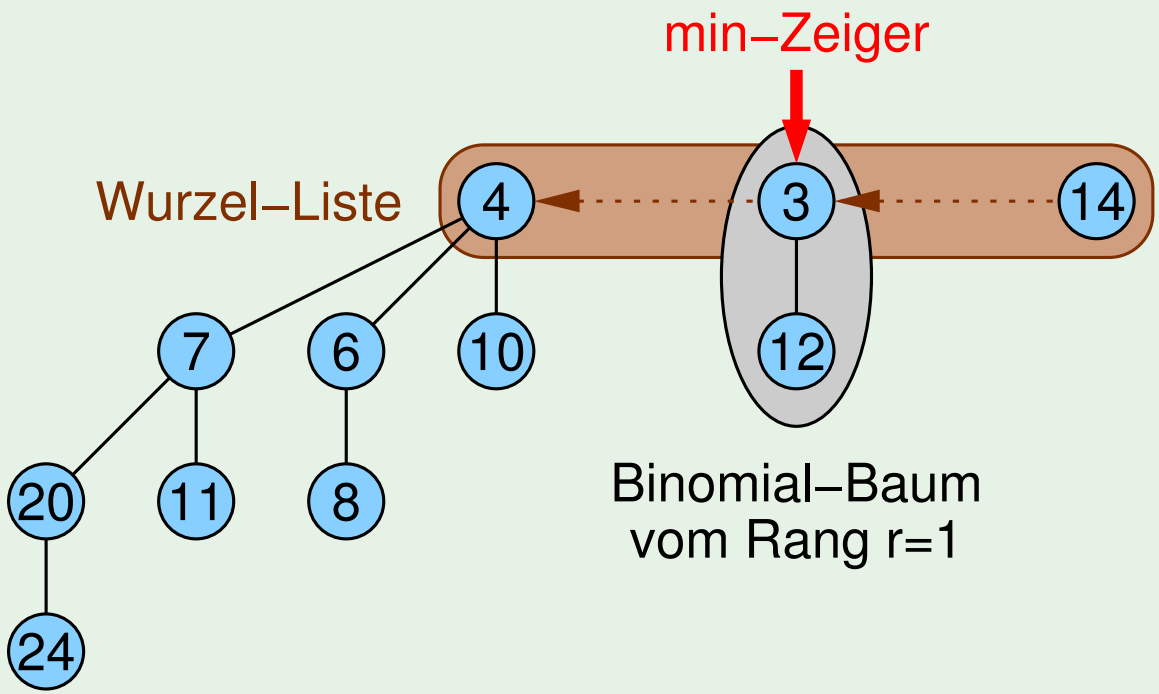


(Zahlen: Ränge)

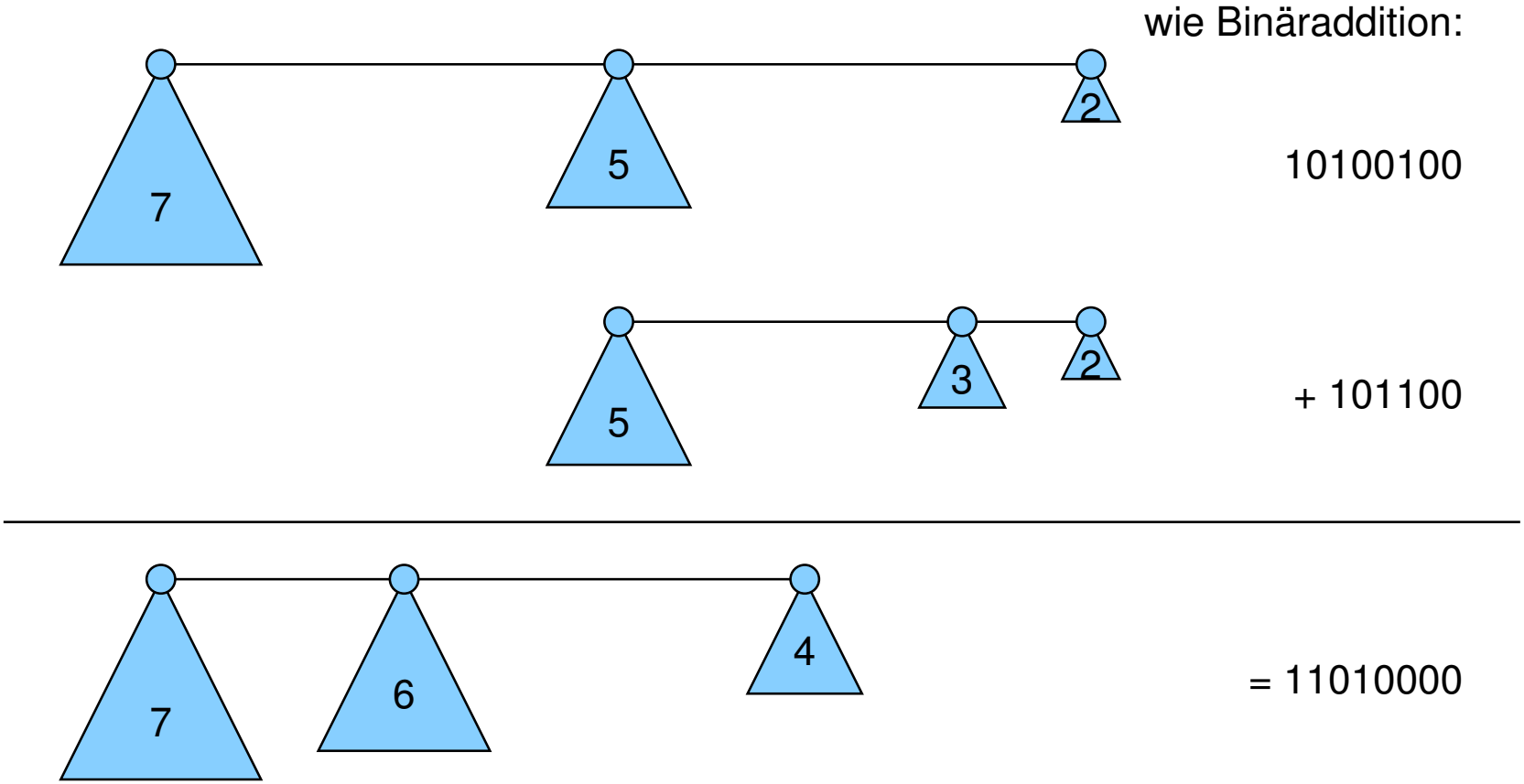
Binomial Heap

Beispiel

Korrekter Binomial Heap:



Merge von zwei Binomial Heaps



Aufwand für Merge: $\mathcal{O}(\log n)$

Binomial Heaps

B_i: Binomial-Baum mit Rang i

Operationen:

- merge: $O(\log n)$
- insert(e): Merge mit B_0, Zeit $O(\log n)$
- min(): spezieller Zeiger, Zeit $O(1)$
- deleteMin():
  sei das Minimum in B_i,
  durch Löschen der Wurzel zerfällt der Binomialbaum in $B_0, \ldots, B_{i-1}$
  Merge mit dem restlichen Binomial Heap kostet $O(\log n)$

Binomial Heaps

Weitere Operationen:

- decreaseKey(h, k): siftUp-Operation in Binomial-Baum für das Element, auf das h zeigt,
  dann ggf. noch min-Zeiger aktualisieren

  Zeit: $\mathcal{O}(\log n)$

- remove(h): Sei e das Element, auf das h zeigt.
  Setze prio(e) $= -\infty$ und wende siftUp-Operation auf e an bis e in der Wurzel, dann weiter wie bei deleteMin

  Zeit: $\mathcal{O}(\log n)$

Bessere Laufzeit mit Fibonacci-Heaps

Fibonacci-Heaps

Verbesserung von Binomial Heaps mit folgenden Kosten:

- min, insert, merge: $\mathcal{O}(1)$ (worst case)
- decreaseKey: $\mathcal{O}(1)$ (amortisiert)
- deleteMin, remove: $\mathcal{O}(\log n)$ (amortisiert)

Wir werden darauf bei den Graph-Algorithmen zurückgreifen.

Übersicht

Übersicht

Vergleich Wörterbuch / Suchstruktur

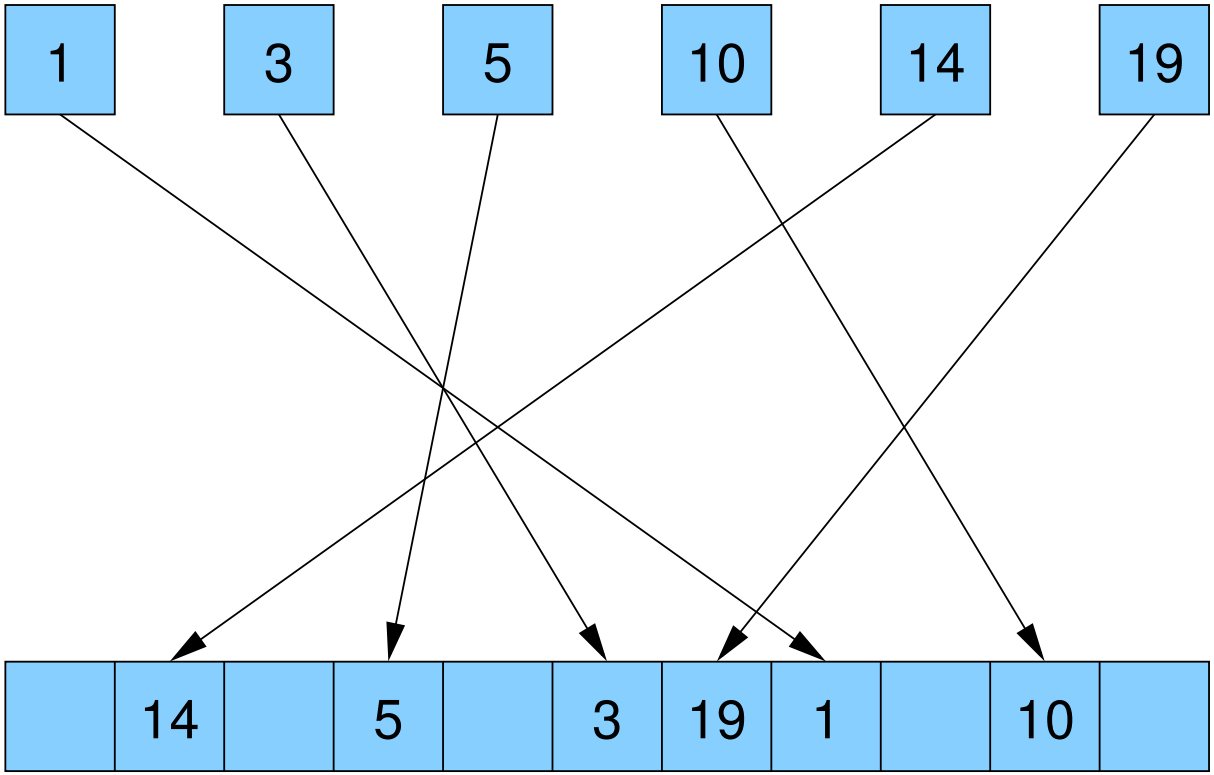
- S: Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

- S.insert(Elem e): $S = S \cup \{e\}$
- S.remove(Key k): $S = S \setminus \{e\}$,
  wobei e das Element mit key(e) $==$ k ist
- S.find(Key k): (Wörterbuch)
  gibt das Element $e \in S$ mit key(e) $==$ k zurück, falls es existiert, sonst null
- S.locate(Key k): (Suchstruktur)
  gibt das Element $e \in S$ mit minimalem Schlüssel key(e) zurück, für das key(e) $\geq$ k

Vergleich Wörterbuch / Suchstruktur

- Wörterbuch effizient über Hashing realisierbar



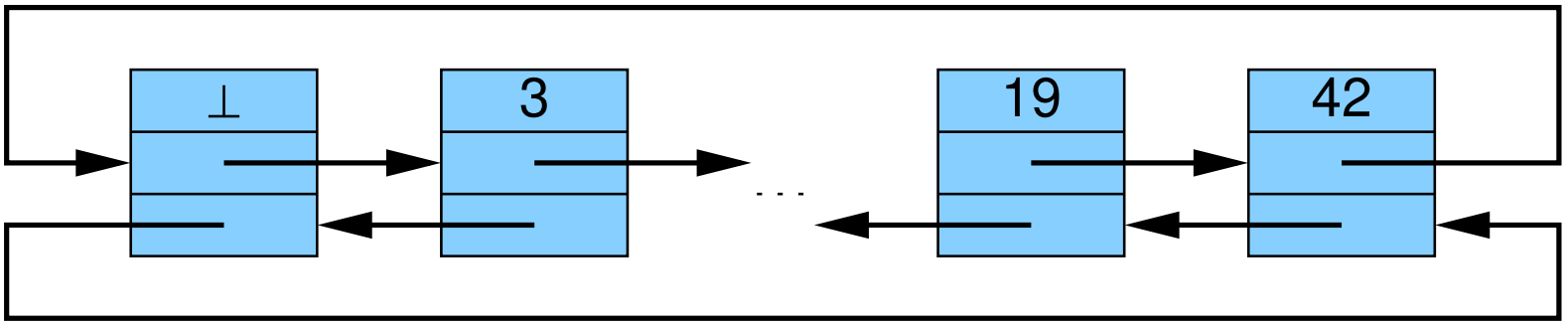
- Hashing zerstört die Ordnung auf den Elementen

$\Rightarrow$ keine effiziente locate-Operation

$\Rightarrow$ keine Intervallanfragen

Suchstruktur

Erster Ansatz: sortierte Liste



Problem:

- insert, remove, locate kosten im worst case $\Theta(n)$ Zeit

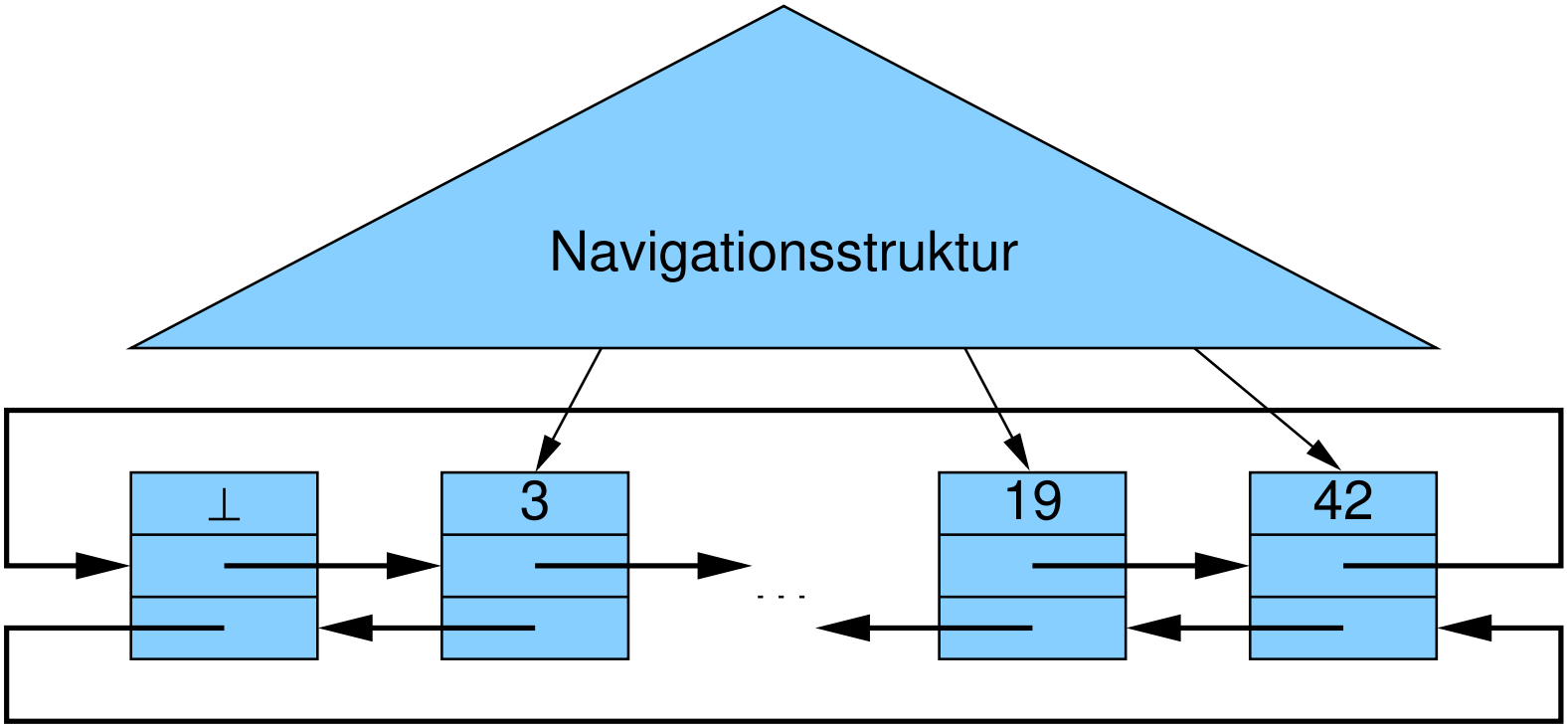
Einsicht:

- wenn locate effizient implementierbar, dann auch die anderen Operationen

Suchstruktur

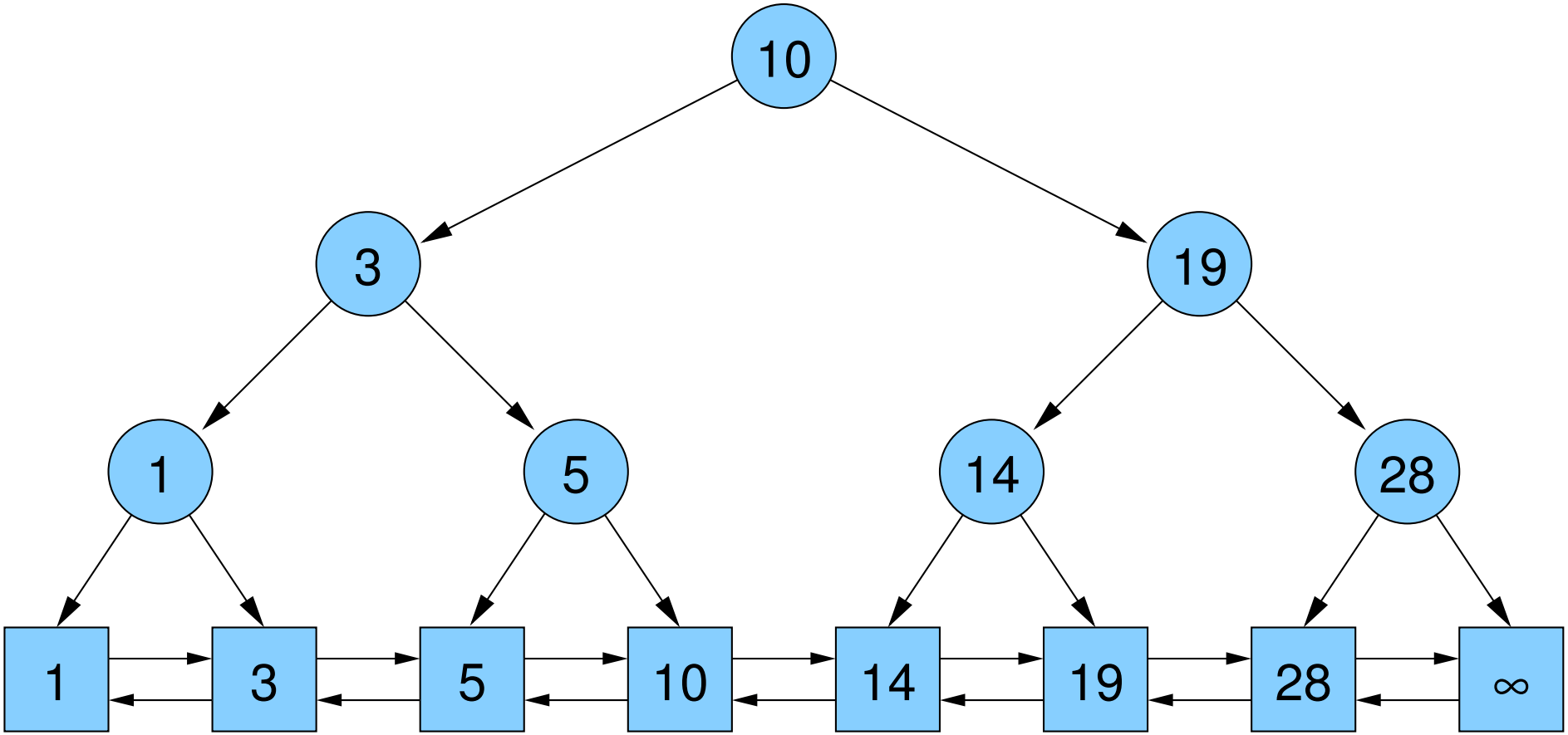
Idee:

- füge Navigationsstruktur hinzu, die locate effizient macht



Übersicht

Binärer Suchbaum (ideal)

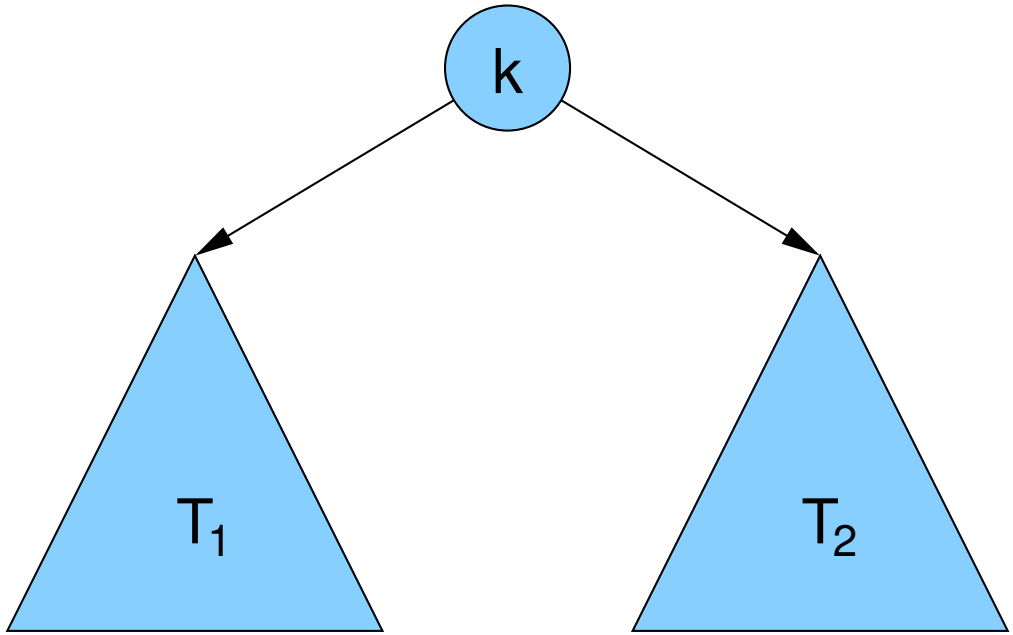


Binärer Suchbaum

Suchbaum-Regel:

Für alle Schlüssel
k_1 in T_1 und k_2 in T_2:

$$k_1 \leq k < k_2$$



locate-Strategie:

- Starte in Wurzel des Suchbaums
- Für jeden erreichten Knoten v:

  Falls key(v) $\geq k_{\text{gesucht}}$, gehe zum linken Kind von v,
  sonst gehe zum rechten Kind

Binärer Suchbaum

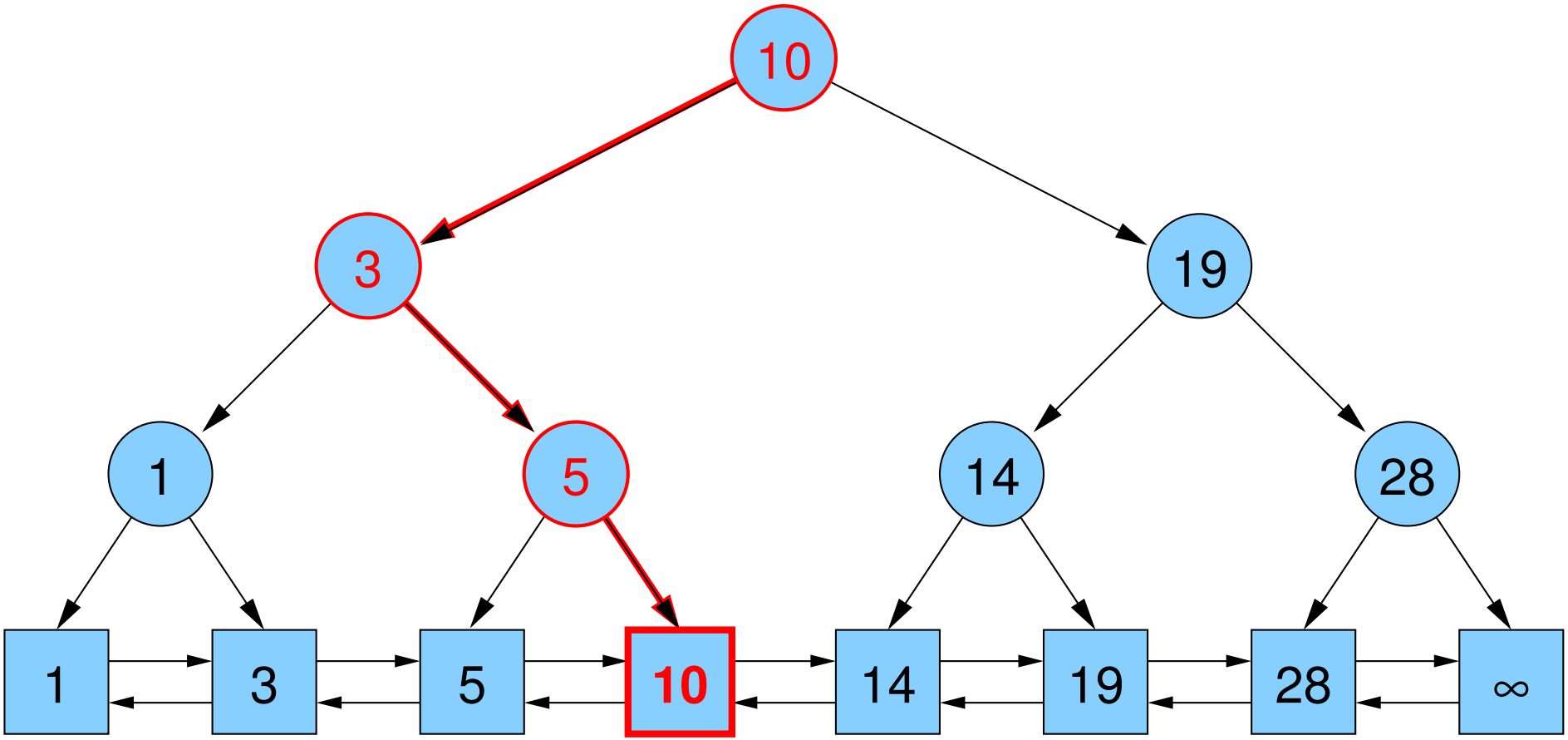
Formal: für einen Baumknoten v sei

- key(v) der Schlüssel von v
- $d(v)$ der Ausgangsgrad (Anzahl Kinder) von v

- Suchbaum-Invariante: $k_1 \leq k < k_2$
  (Sortierung der linken und rechten Nachfahren)
- Grad-Invariante: $d(v) \leq 2$
  (alle Baumknoten haben höchstens 2 Kinder)
- Schlüssel-Invariante:
  (Für jedes Element e in der Liste gibt es *genau einen* Baumknoten v mit key(v) $==$ key(e))

Binärer Suchbaum / locate

locate(9)



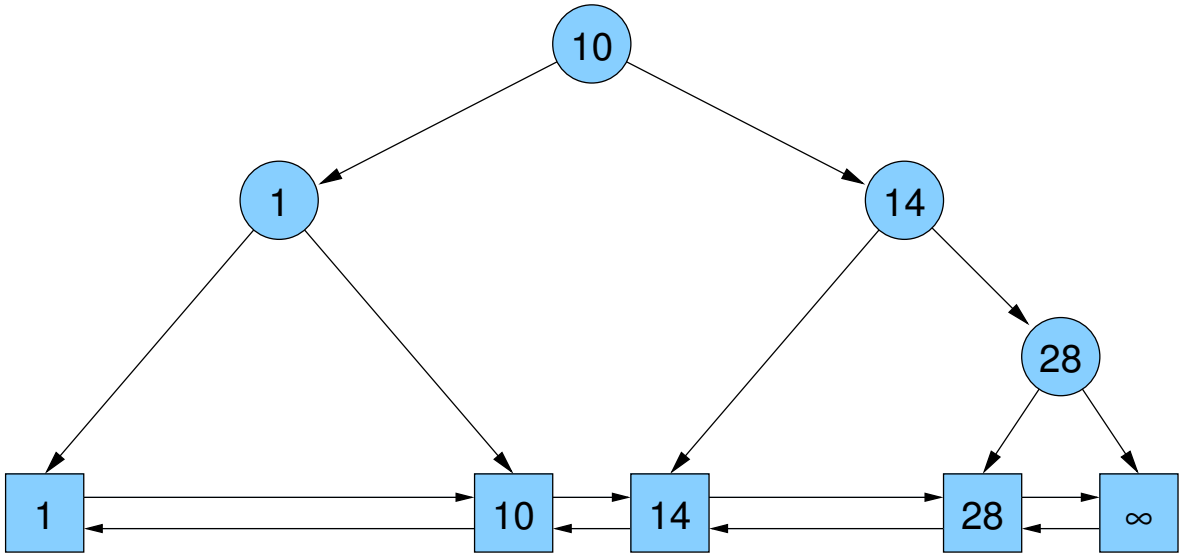
Binärer Suchbaum / insert, remove

Strategie:

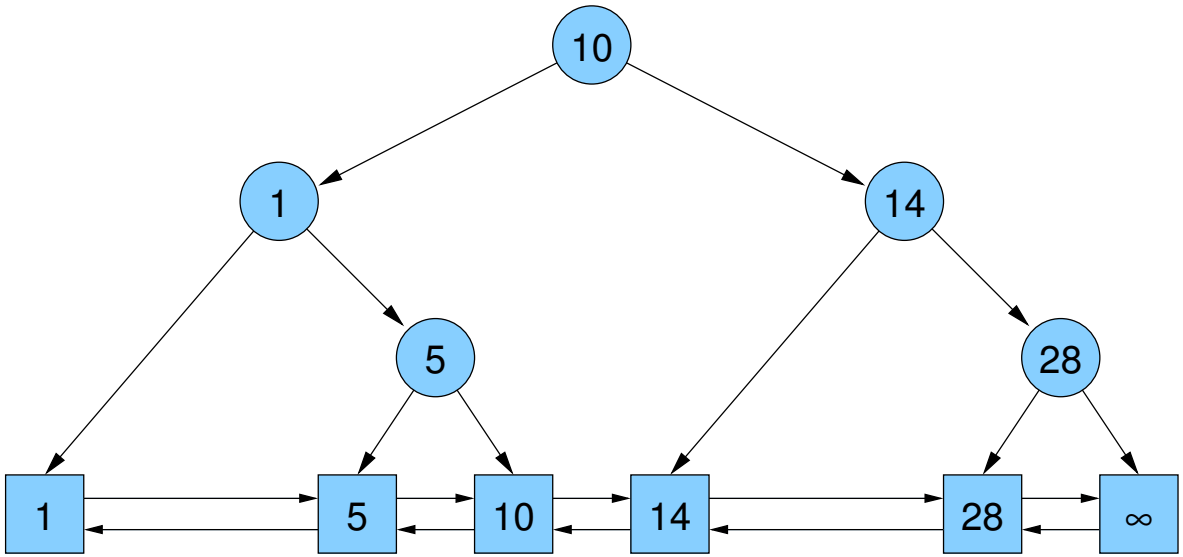
- insert(e):
  - erst wie locate(key(e)) bis Element e' in Liste erreicht
  - falls key(e') $>$ key(e):
    füge e vor e' ein, sowie ein neues Suchbaumblatt für e und e' mit key(e) als Splitter Key, so dass Suchbaum-Regel erfüllt

- remove(k):
  - erst wie locate(k) bis Element e in Liste erreicht
  - falls key(e) $= k$, lösche e aus Liste und Vater v von e aus Suchbaum und
  - setze in dem Baumknoten w mit key(w) $= k$ den neuen Wert key(w) $=$ key(v)

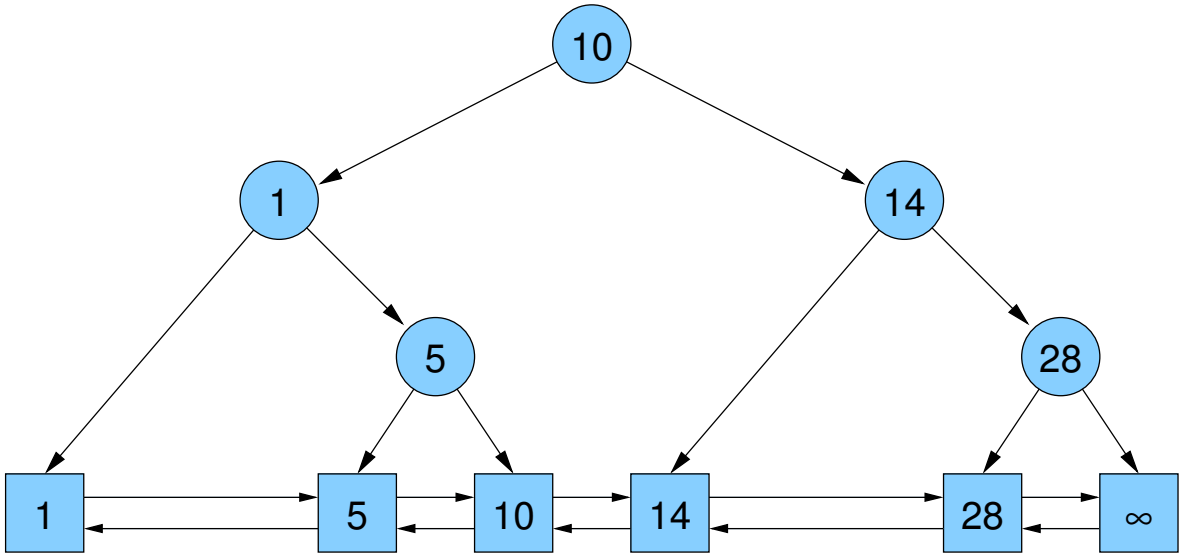
Binärer Suchbaum / insert, remove



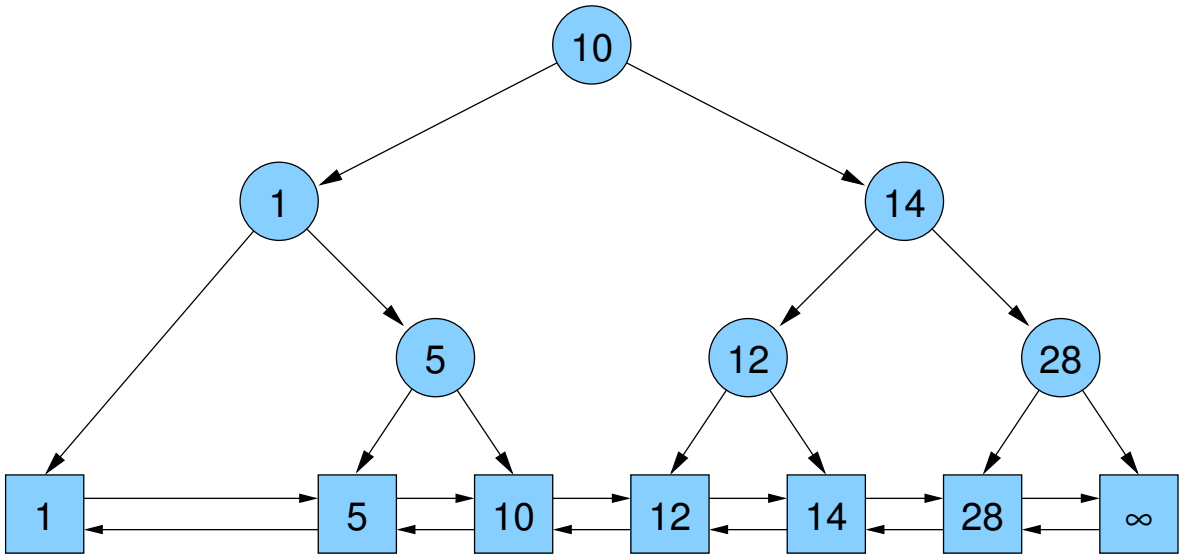
insert(5)



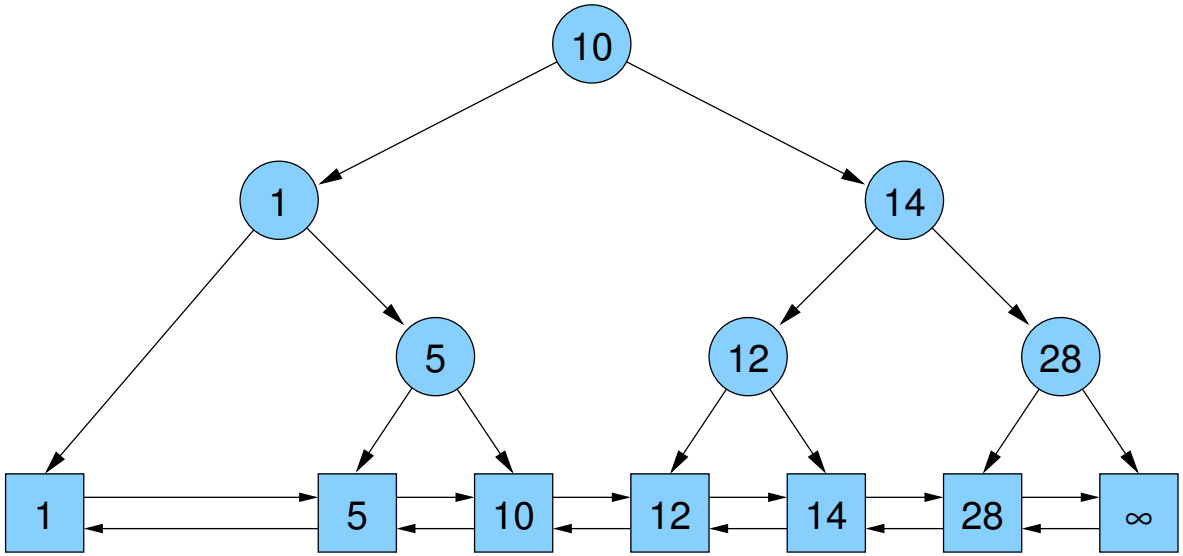
Binärer Suchbaum / insert, remove



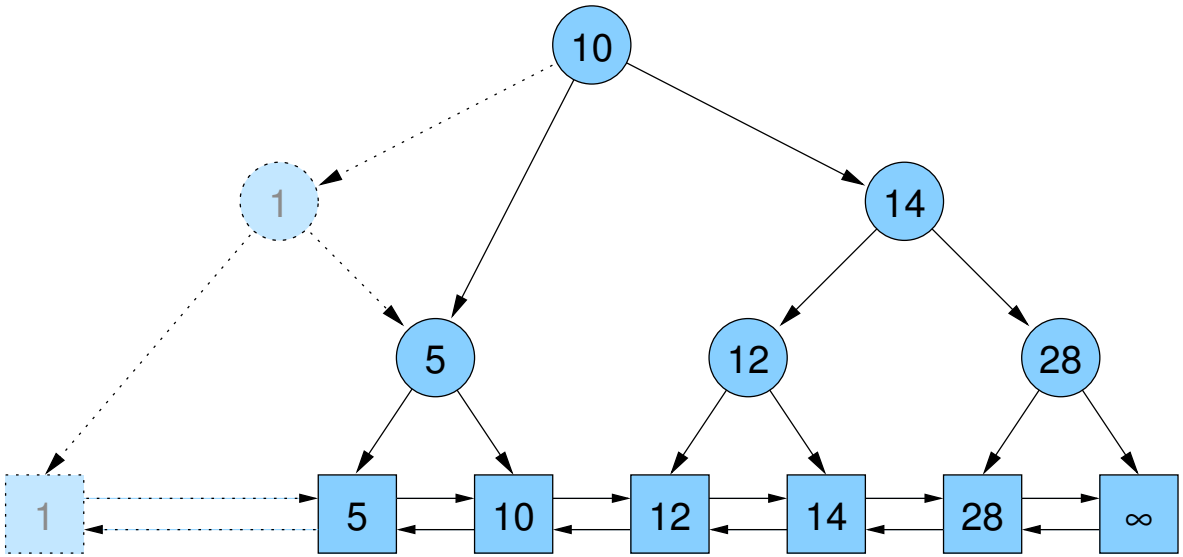
insert(12)



Binärer Suchbaum / insert, remove

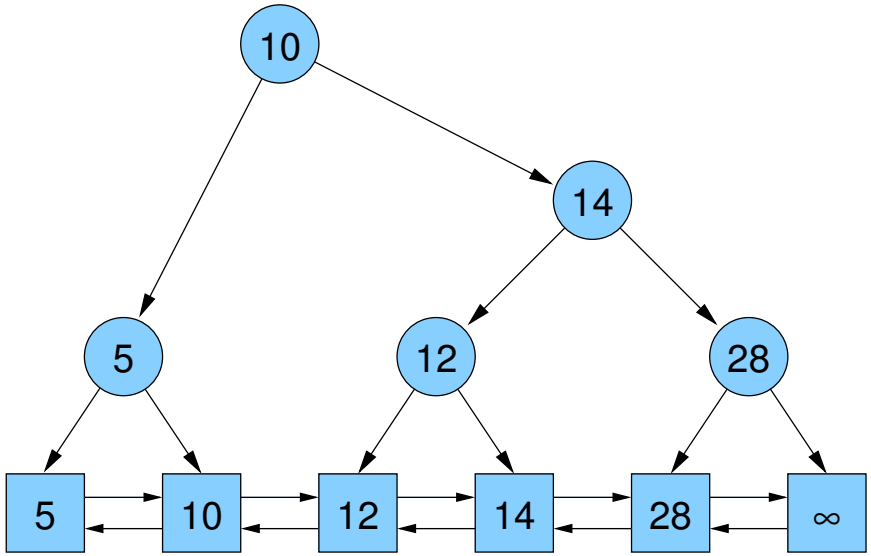


remove(1)

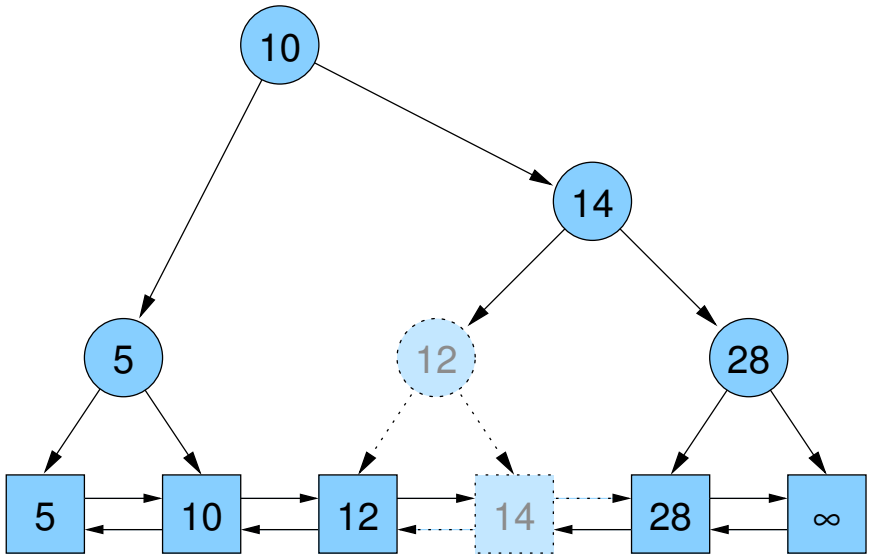


Binärer Suchbaum / insert, remove

1

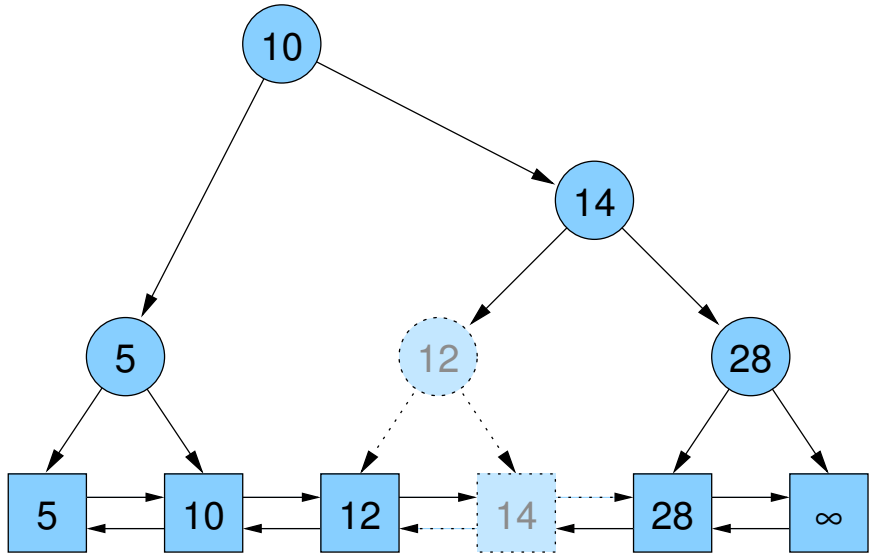


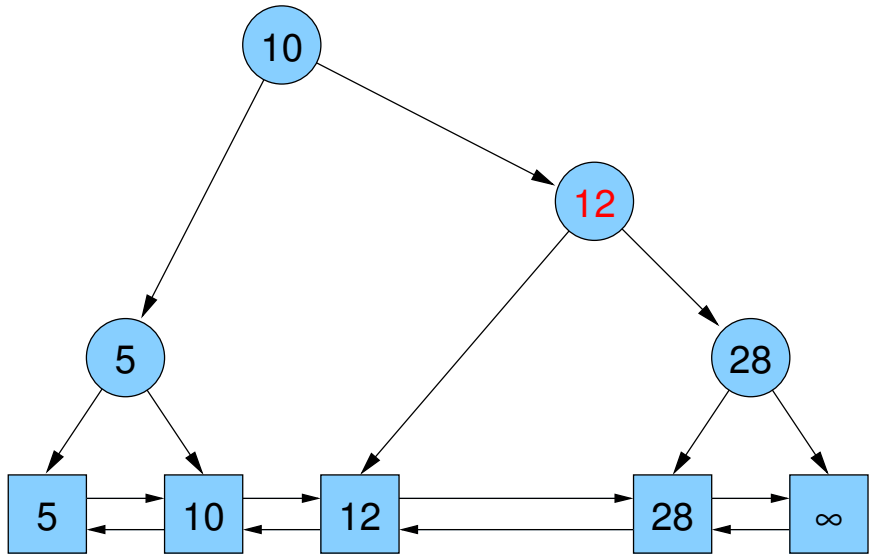
remove(14)



Binärer Suchbaum / insert, remove

remove(14)





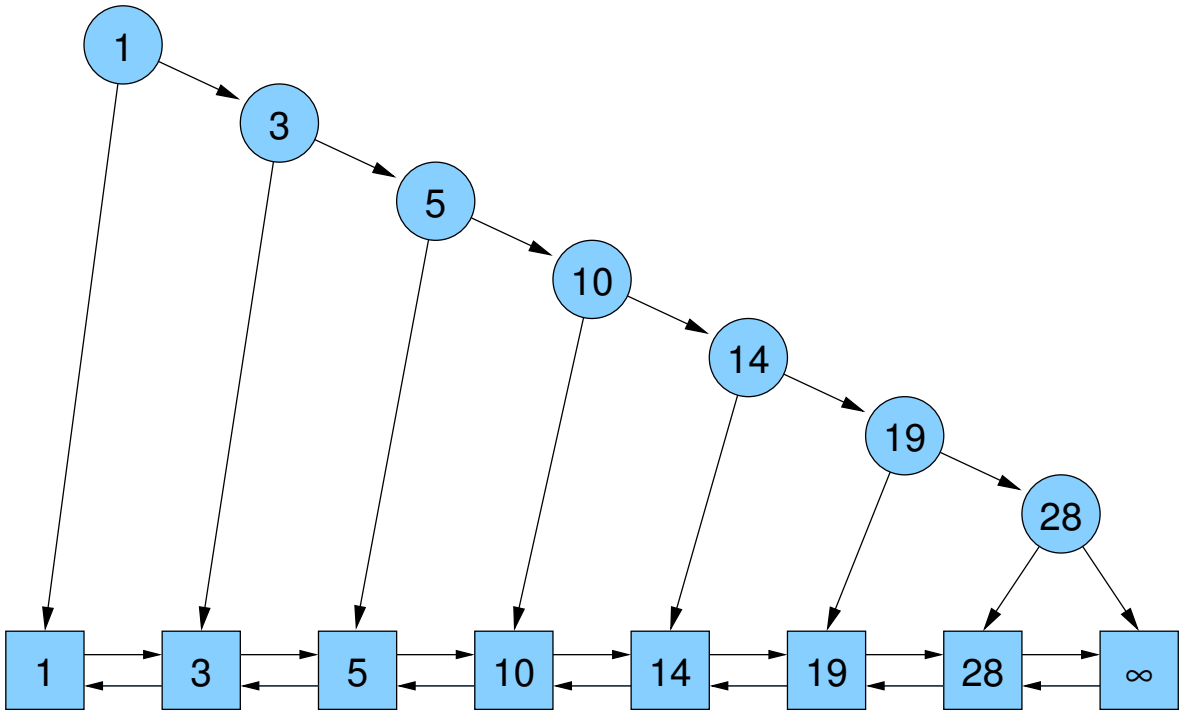
Binärer Suchbaum / worst case

Problem:

- Baumstruktur kann zur Liste entarten
- Höhe des Baums kann linear in der Anzahl der Elemente werden

$\Rightarrow$ locate kann im worst case Zeitaufwand $\Theta(n)$ verursachen

Beispiel: Zahlen werden in sortierter Reihenfolge eingefügt



Übersicht

AVL-Bäume

Balancierte binäre Suchbäume

Strategie zur Lösung des Problems:

- Balancierung des Baums

G. M. **A**delson-**V**elsky & E. M. **L**andis (1962):

- Beschränkung der Höhenunterschiede für Teilbäume auf $[-1, 0, +1]$

$\Rightarrow$ führt nicht unbedingt zu einem idealen unvollständigen Binärbaum (wie wir ihn von array-basierten Heaps kennen), aber zu einem hinreichenden Gleichgewicht

AVL-Bäume

Balancierte binäre Suchbäume

Worst Case: Fibonacci-Baum

- Laufzeit der Operation hängt von der Baumhöhe ab
- Was ist die schlimmste Höhe bei gegebener Anzahl von Elementen?
- bzw: Wieviel Elemente hat ein Baum bei gegebener Höhe h mindestens?

$\Rightarrow$ Für mindestens ein Kind hat der Unterbaum Höhe $h - 1$.
Im worst case hat das andere Kind Höhe $h - 2$ (kleiner geht nicht wegen Höhendifferenzbeschränkung)

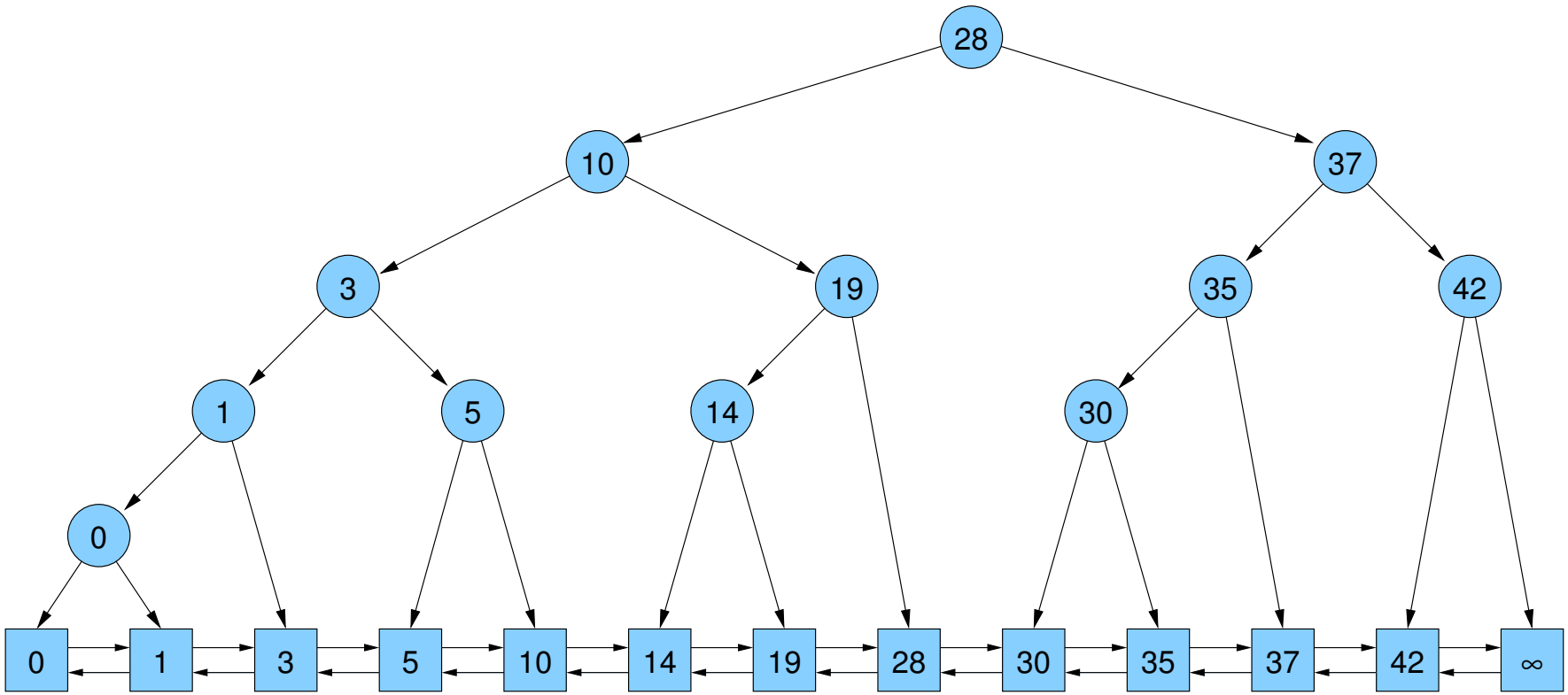
$\Rightarrow$ Anzahl der (inneren) Knoten entspricht den Fibonacci-Zahlen:

$$F_n = F_{n-1} + F_{n-2}$$

AVL-Bäume

Balancierte binäre Suchbäume

Worst Case: Fibonacci-Baum



AVL-Bäume

Balancierte binäre Suchbäume

Worst Case: Fibonacci-Baum

- Fibonacci-Baum der Stufe 0 ist der leere Baum
- Fibonacci-Baum der Stufe 1 ist ein einzelner Knoten
- Fibonacci-Baum der Stufe $h + 1$ besteht aus einer Wurzel, deren Kinder Fibonacci-Bäume der Stufen h und $h - 1$ sind

Explizite Formel:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

$\Rightarrow$ Die Anzahl der Elemente ist exponentiell in der Höhe bzw. die Höhe ist logarithmisch in der Anzahl der Elemente.

AVL-Bäume

Balancierte binäre Suchbäume

Operationen auf einem AVL-Baum:

- insert und remove können zunächst zu Binärbäumen führen, die die Balance-Bedingung für die Höhendifferenz der Teilbäume verletzen

$\Rightarrow$ Teilbäume müssen umgeordnet werden, um das Kriterium für AVL-Bäume wieder zu erfüllen (Rebalancierung / Rotation)

- Dazu wird an jedem Knoten die Höhendifferenz der beiden Unterbäume vermerkt ($-1, 0, +1$, mit 2 Bit / Knoten)

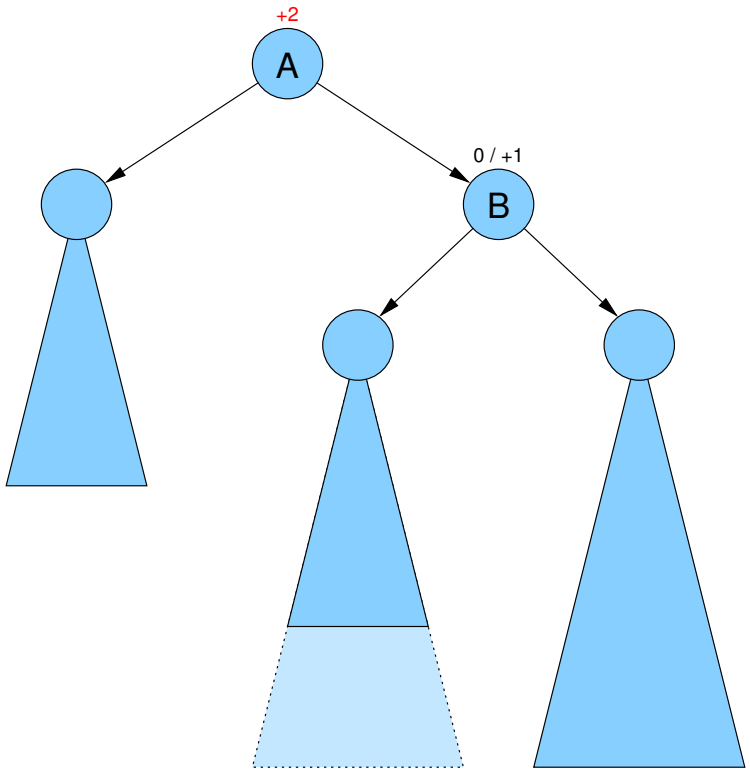
(Hausaufgabe ...)

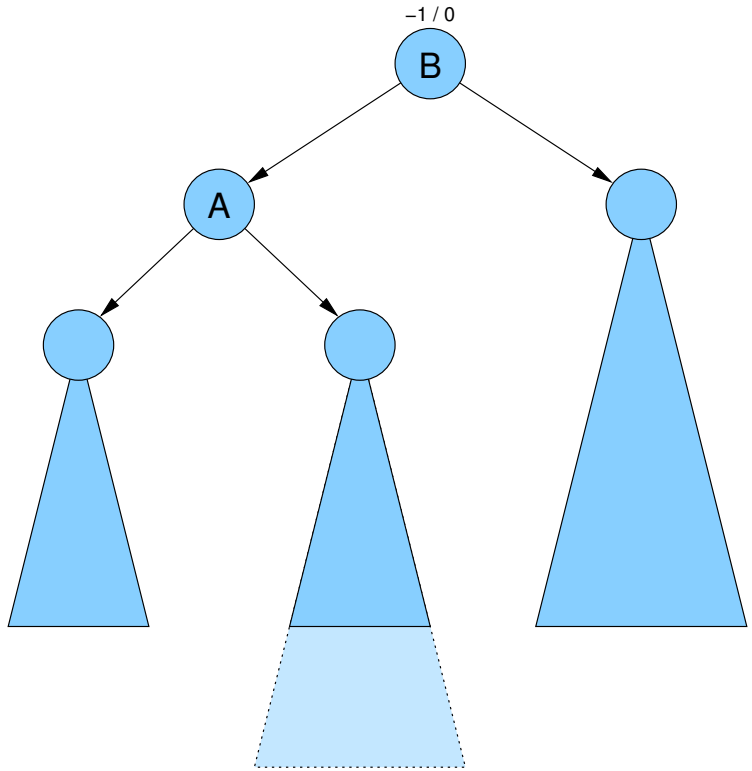
- Operationen locate, insert und remove haben Laufzeit $O(\log n)$

AVL-Bäume

Balancierte binäre Suchbäume

Einfachrotation:

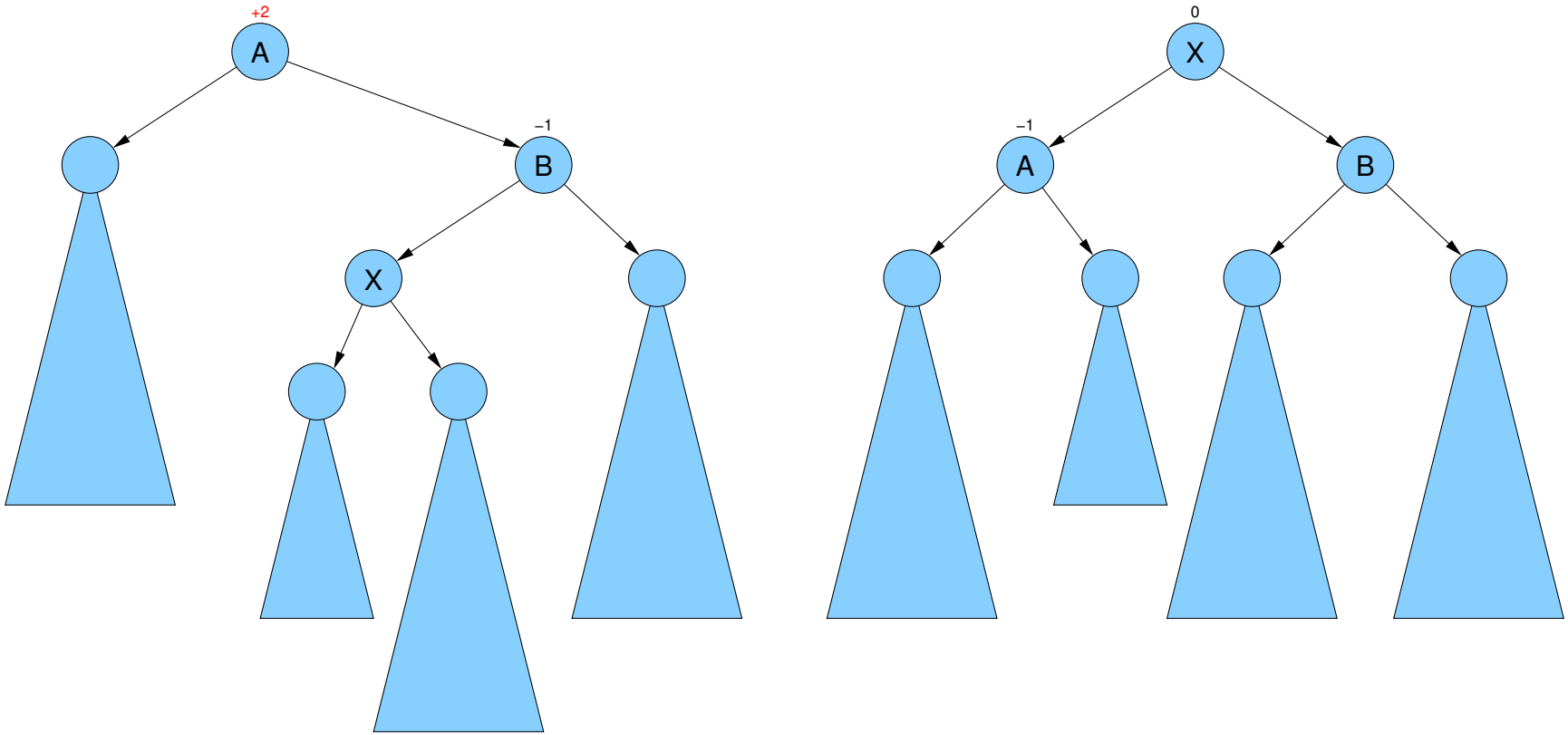




AVL-Bäume

Balancierte binäre Suchbäume

Doppelrotation:



Übersicht

(a,b)-Baum

Problem: Baumstruktur kann zur Liste entarten

Lösung: (a,b)-Baum

Idee:

- $d(v)$: Ausgangsgrad (Anzahl Kinder) von Knoten v
- $t(v)$: Tiefe (in Kanten) von Knoten v
- Form-Invariante:
  alle Blätter in derselben Tiefe: $t(v) = t(w)$ für Blätter v, w
- Grad-Invariante:
  Für alle internen Knoten v (außer Wurzel) gilt:

$$a \leq d(v) \leq b \qquad \text{(wobei } a \geq 2 \text{ und } b \geq 2a-1\text{)}$$

Für Wurzel r: $2 \leq d(r) \leq b$ (außer wenn nur 1 Blatt im Baum)

(a,b)-Baum

Lemma

Ein (a,b)-Baum für $n \geq 1$ Elemente hat Tiefe $\leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$.

Beweis.

- Baum hat $n+1$ Blätter $\quad (+1$ wegen ∞-Dummy)
- Im Fall $n \geq 1$ hat die Wurzel Grad ≥ 2,
  die anderen inneren Knoten haben Grad $\geq a$.

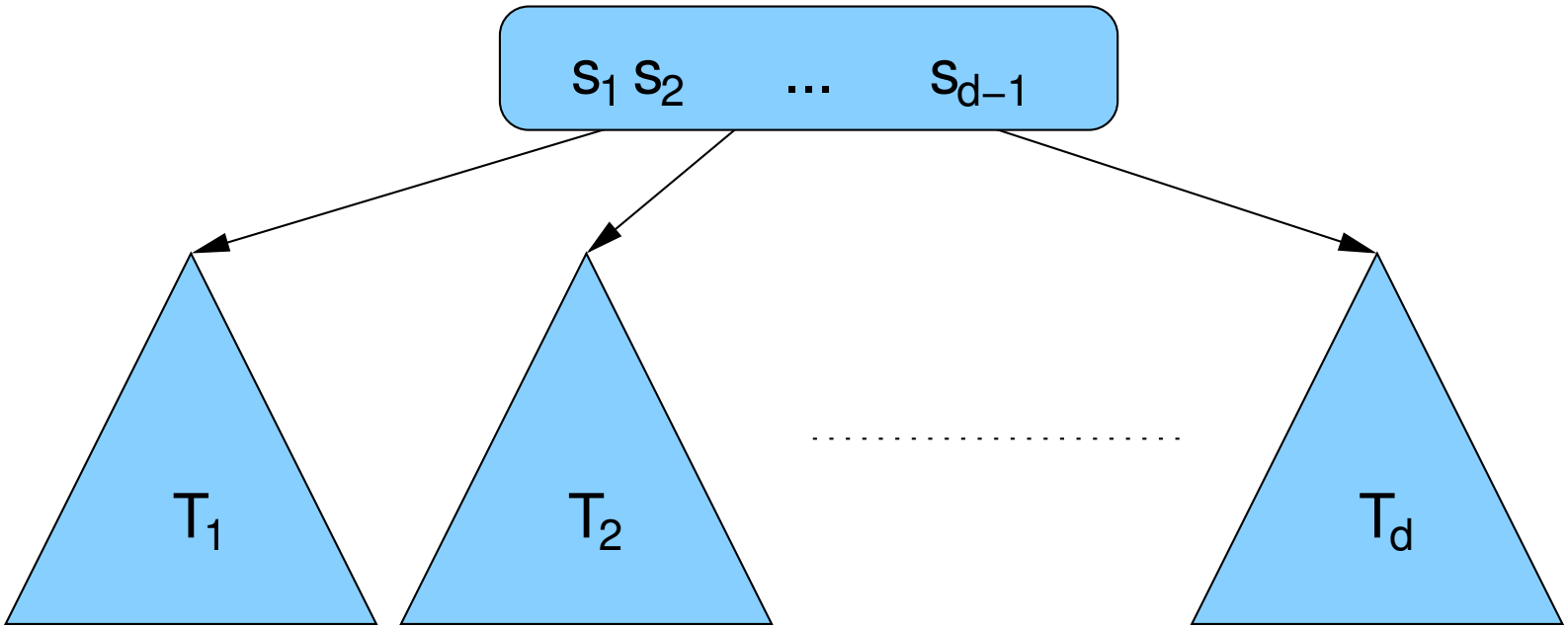
$\Rightarrow$ Bei Tiefe t gibt es $\geq 2a^{t-1}$ Blätter

- $n+1 \geq 2a^{t-1} \quad \Leftrightarrow \quad t \leq 1 + \log_a \frac{n+1}{2}$
- Da t eine ganze Zahl ist, gilt $\quad t \leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$.

□

(a,b)-Baum: Split-Schlüssel

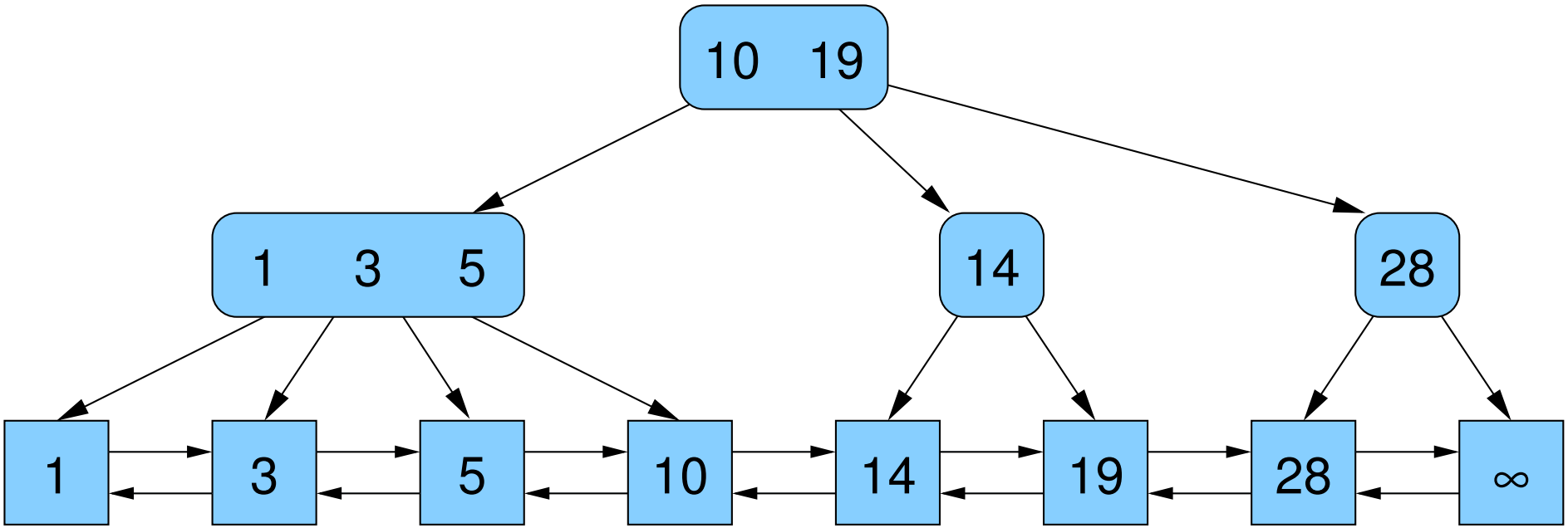
- Jeder Knoten v enthält ein sortiertes Array von $d(v) - 1$ Split-Schlüsseln $s_1, \ldots, s_{d(v)-1}$



- (a,b)-Suchbaum-Regel:
  Für alle Schlüssel k in T_i und k' in T_{i+1} gilt:
  $k \leq s_i < k'$ bzw. $s_{i-1} < k \leq s_i$ $\quad (s_0 = -\infty,\ s_d = \infty)$

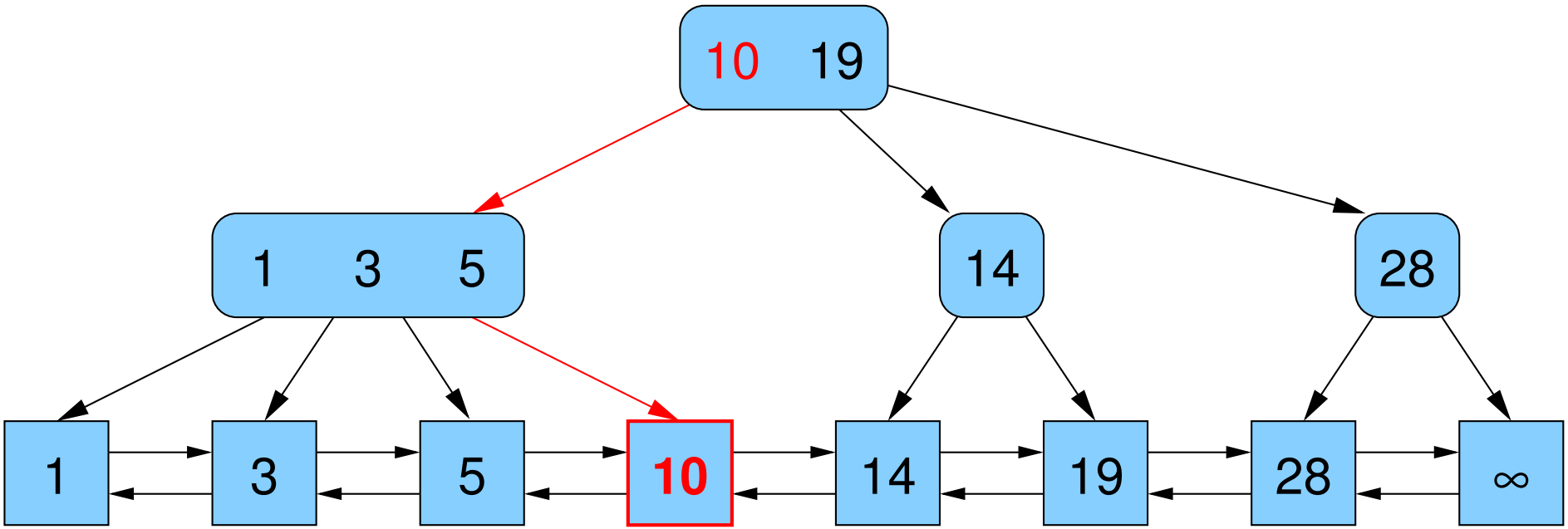
(a,b)-Baum

Beispiel:



(a,b)-Baum

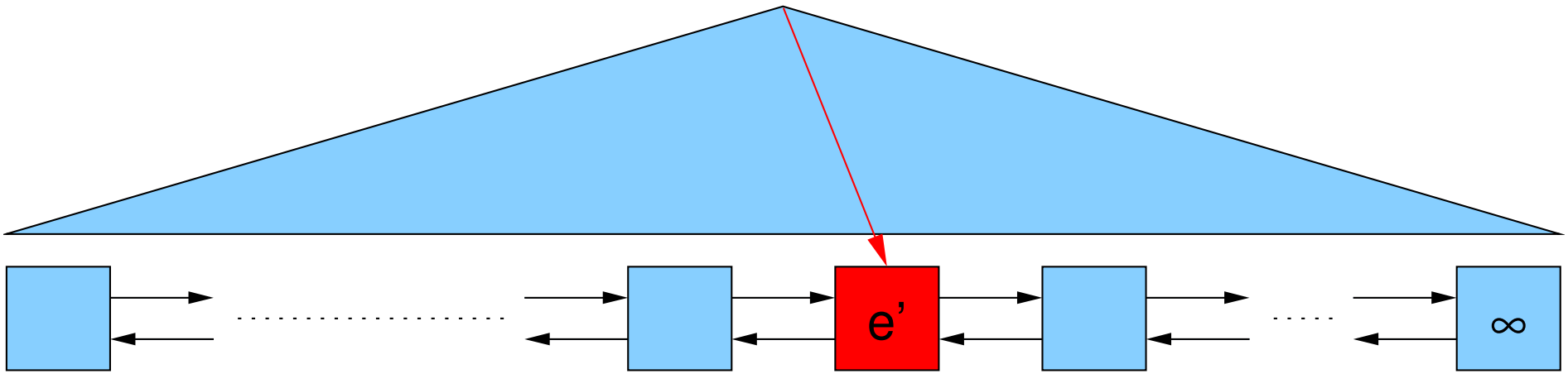
locate(9)



(a,b)-Baum

insert(e)

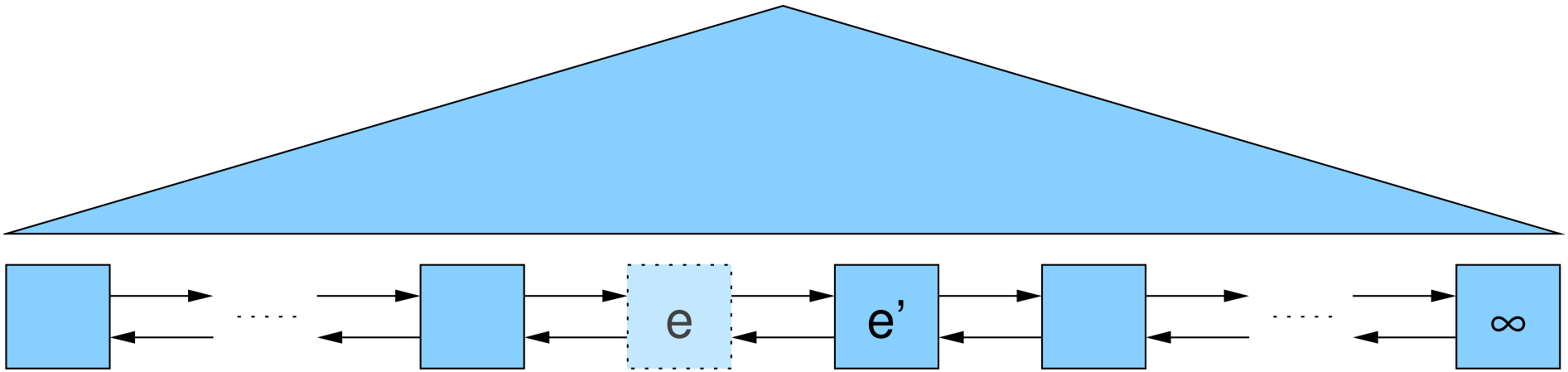
- Abstieg wie bei locate(key(e)) bis Element e' in Liste erreicht
- falls key(e')>key(e), füge e vor e' ein



(a,b)-Baum

insert(e)

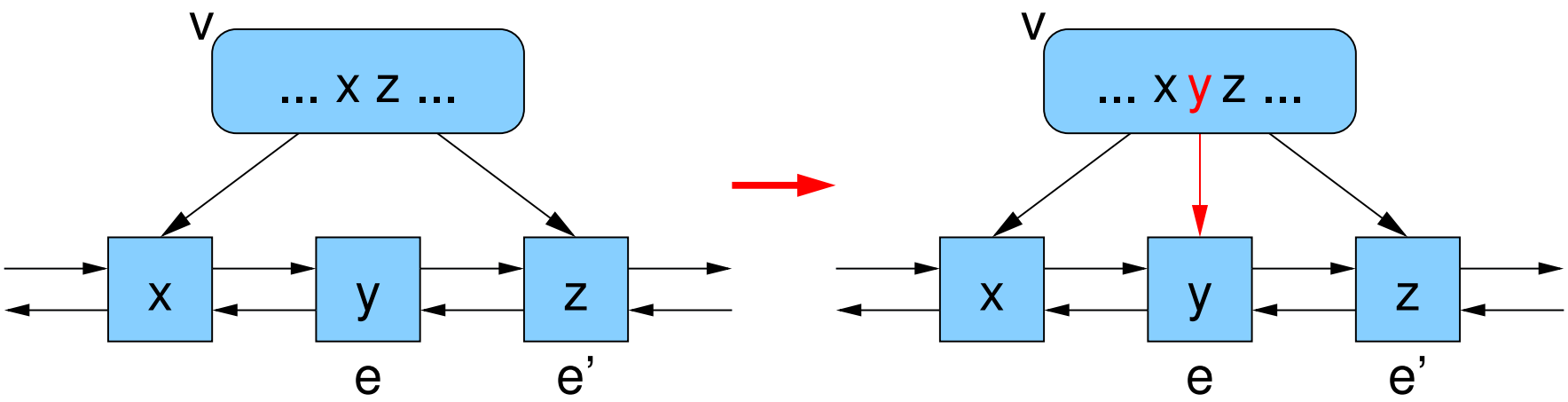
- Abstieg wie bei locate(key(e)) bis Element e' in Liste erreicht
- falls key(e')>key(e), füge e vor e' ein



(a,b)-Baum

insert(e)

- füge key(e) und Handle auf e in Baumknoten v über e ein
- falls $d(v) \leq b$, dann fertig

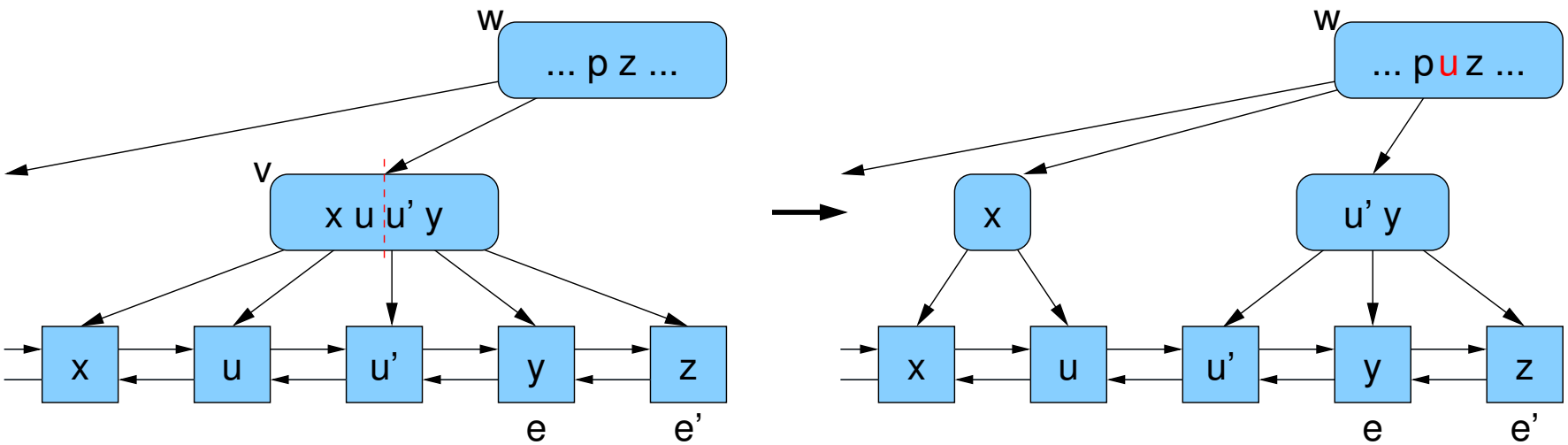


(a,b)-Baum

insert(e)

- füge key(e) und Handle auf e in Baumknoten v über e ein
- falls $d(v) > b$, dann teile v in zwei Knoten auf und
- verschiebe den Splitter (größter Key im linken Teil) in den Vaterknoten

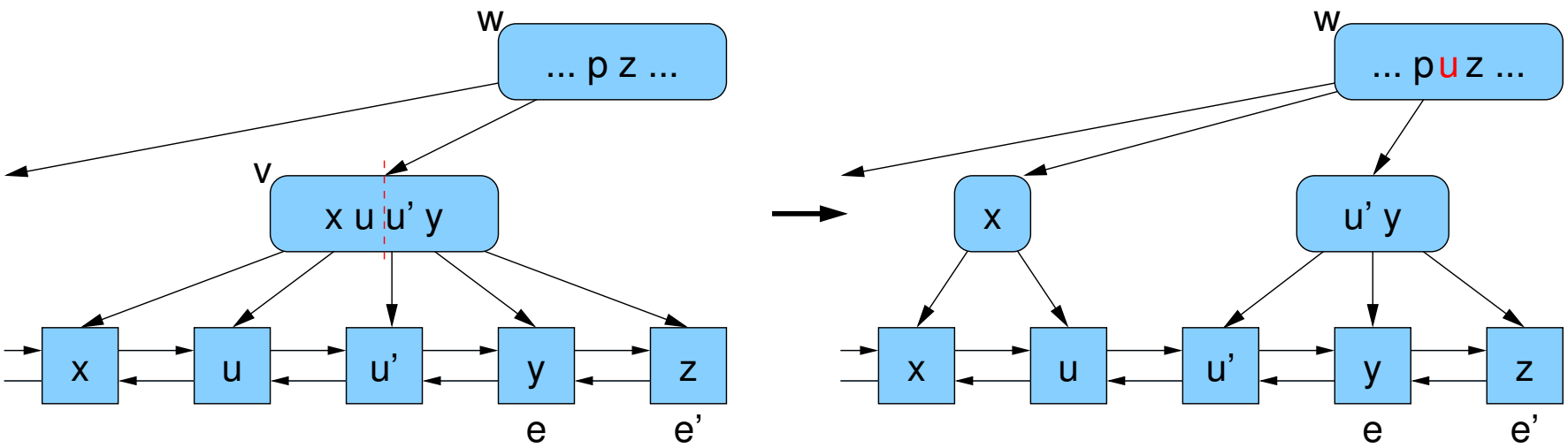
Beispiel: $(2,4)$-Baum



(a,b)-Baum

insert(e)

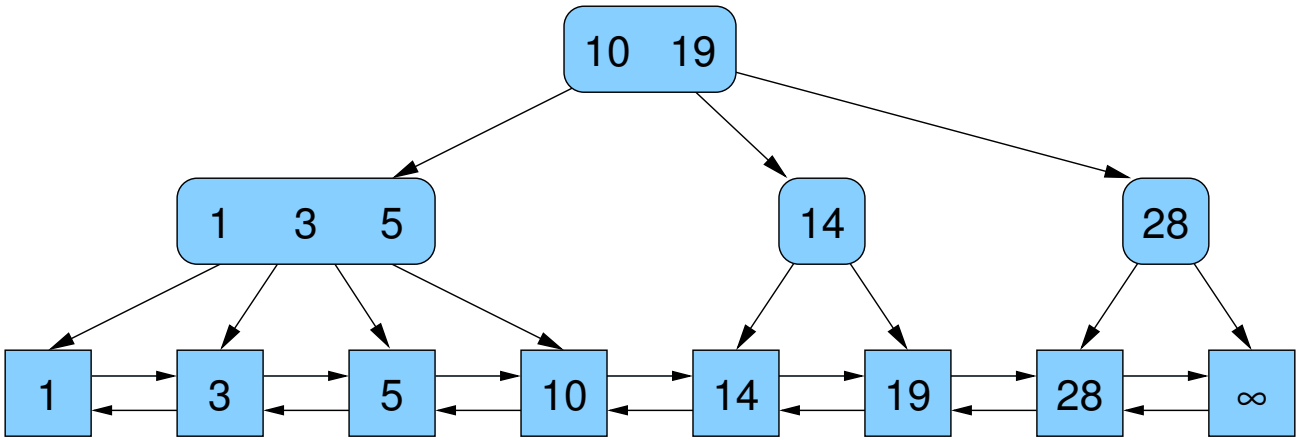
- falls $d(w) > b$, dann teile w in zwei Knoten auf usw.
  bis Grad $\leq b$
  oder Wurzel aufgeteilt wurde



(a,b)-Baum / insert

$a = 2, b = 4$

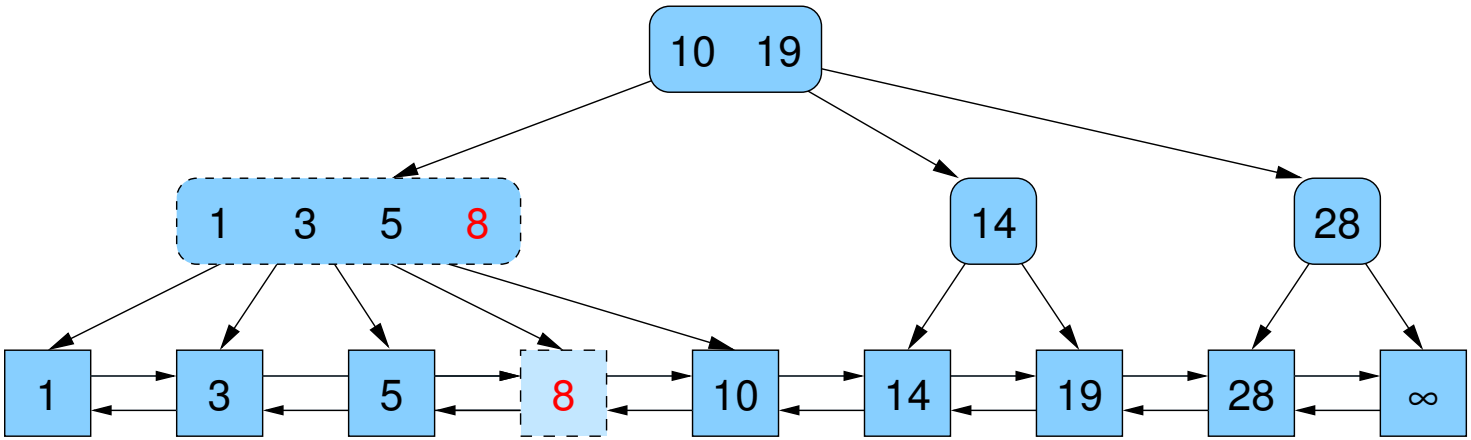
insert(8)



(a,b)-Baum / insert

$a = 2, b = 4$

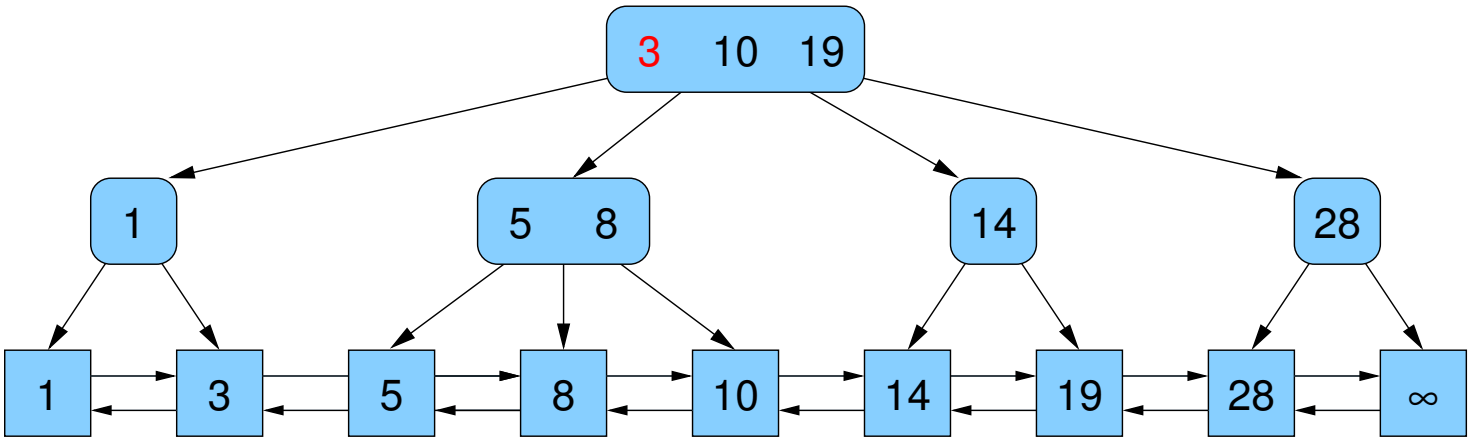
insert(8)



(a,b)-Baum / insert

$a = 2, b = 4$

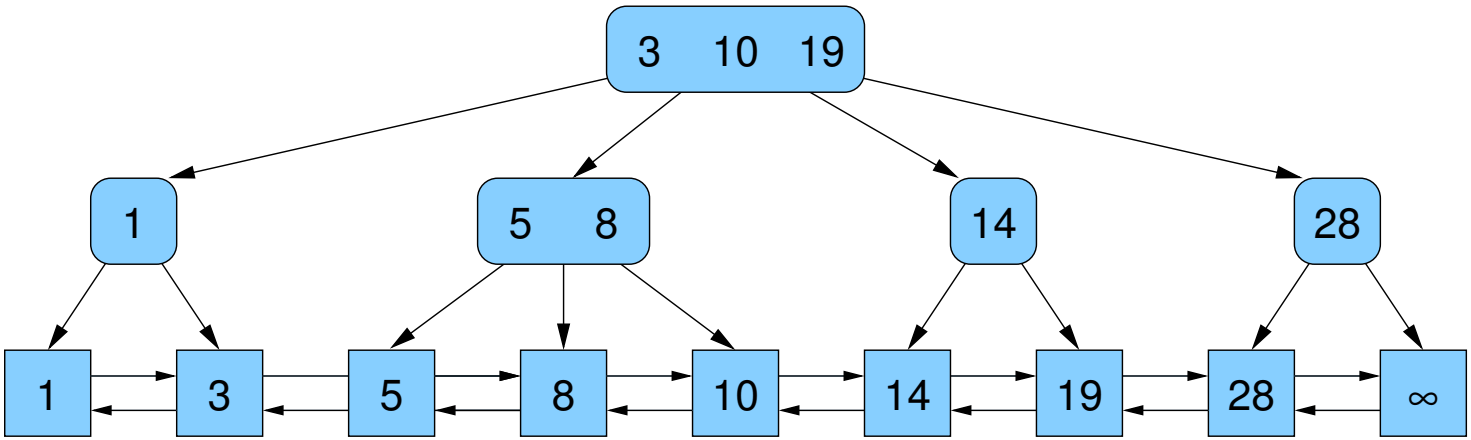
insert(8)



(a,b)-Baum / insert

$a = 2, b = 4$

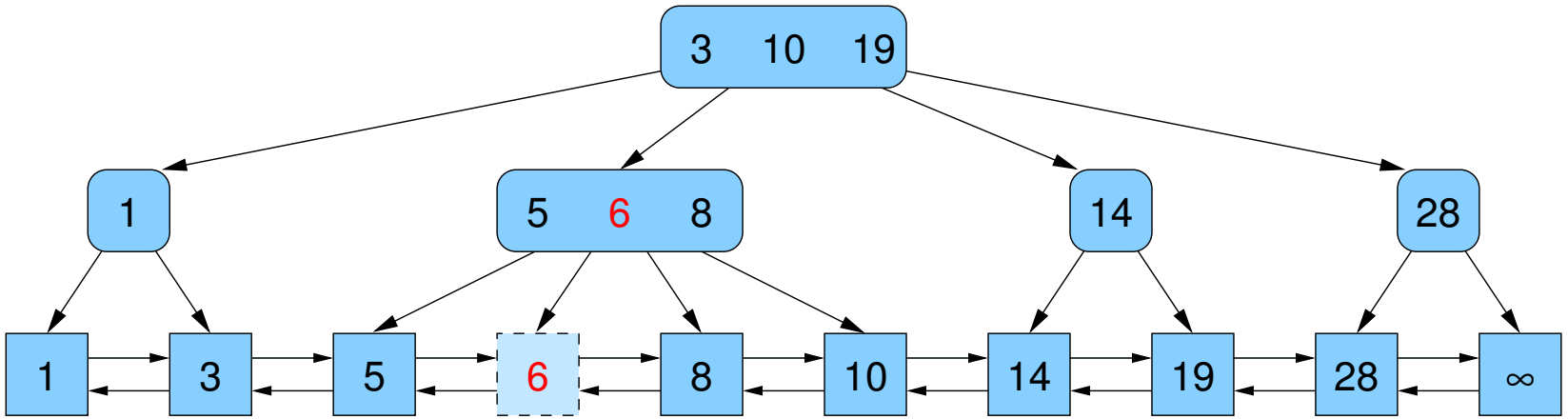
insert(6)



(a,b)-Baum / insert

$a = 2, b = 4$

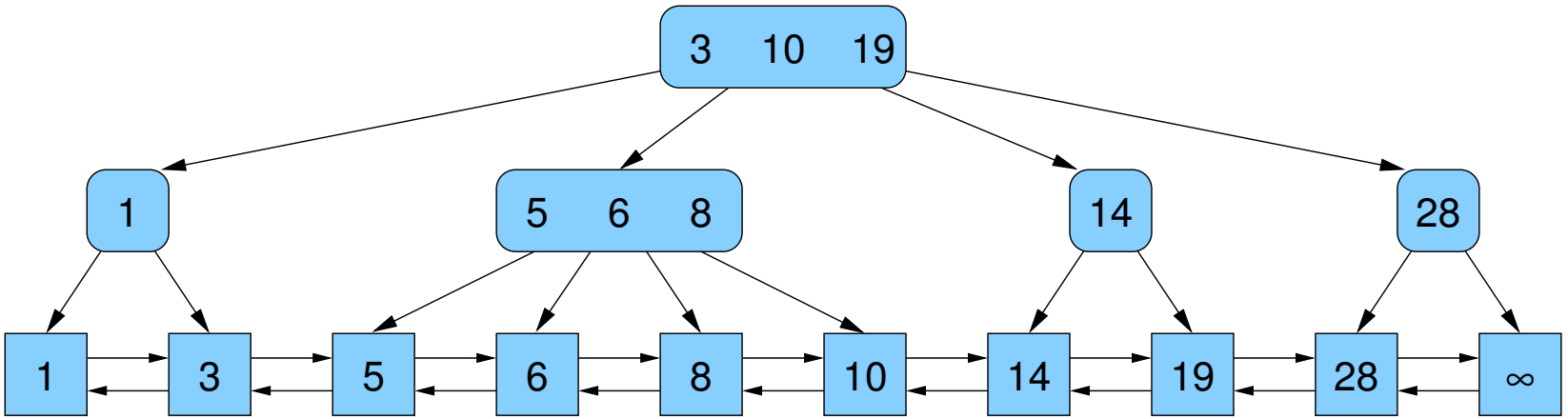
insert(6)



(a,b)-Baum / insert

$a = 2, b = 4$

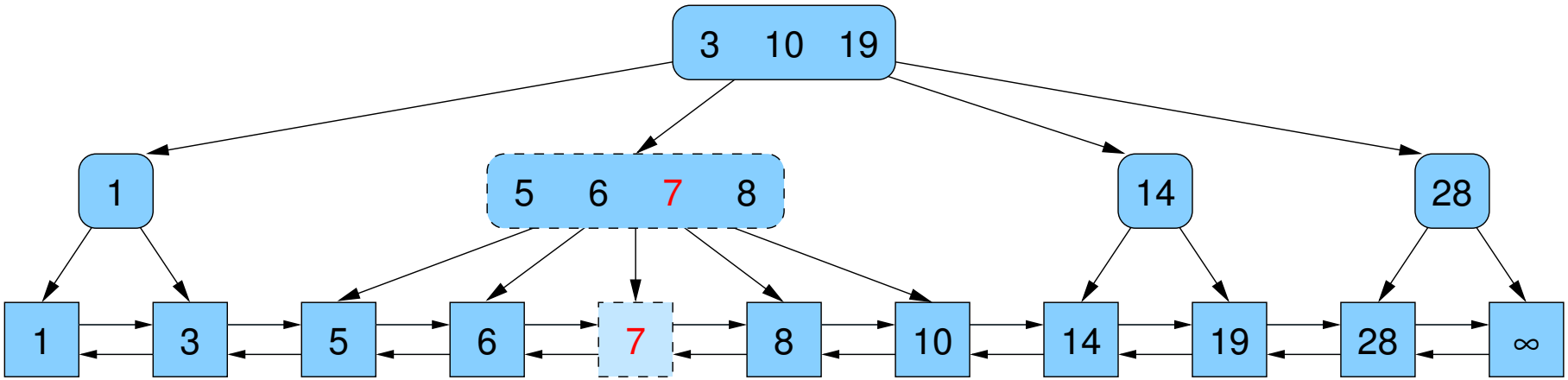
insert(7)



(a,b)-Baum / insert

$a = 2, b = 4$

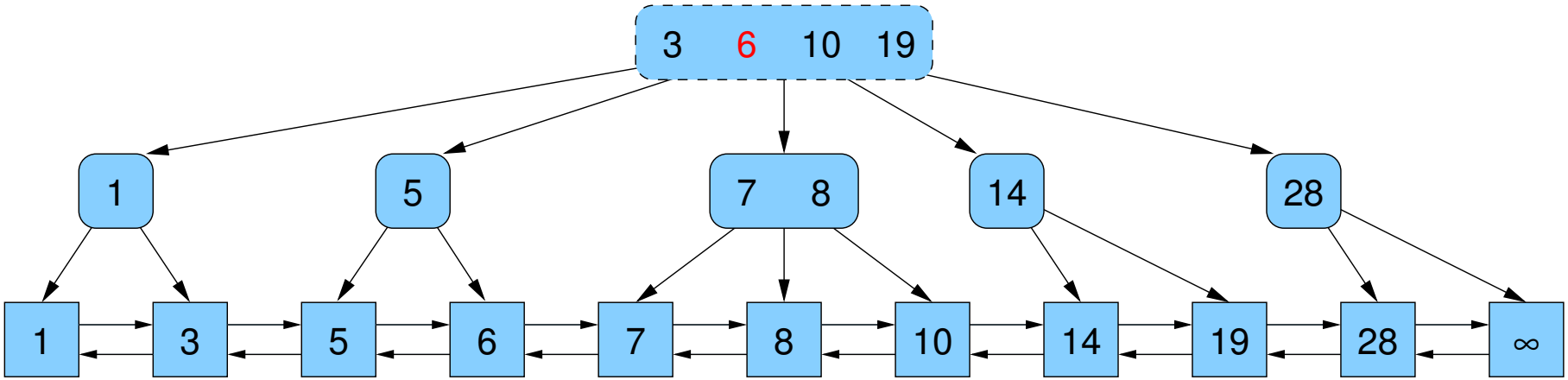
insert(7)



(a,b)-Baum / insert

$a = 2, b = 4$

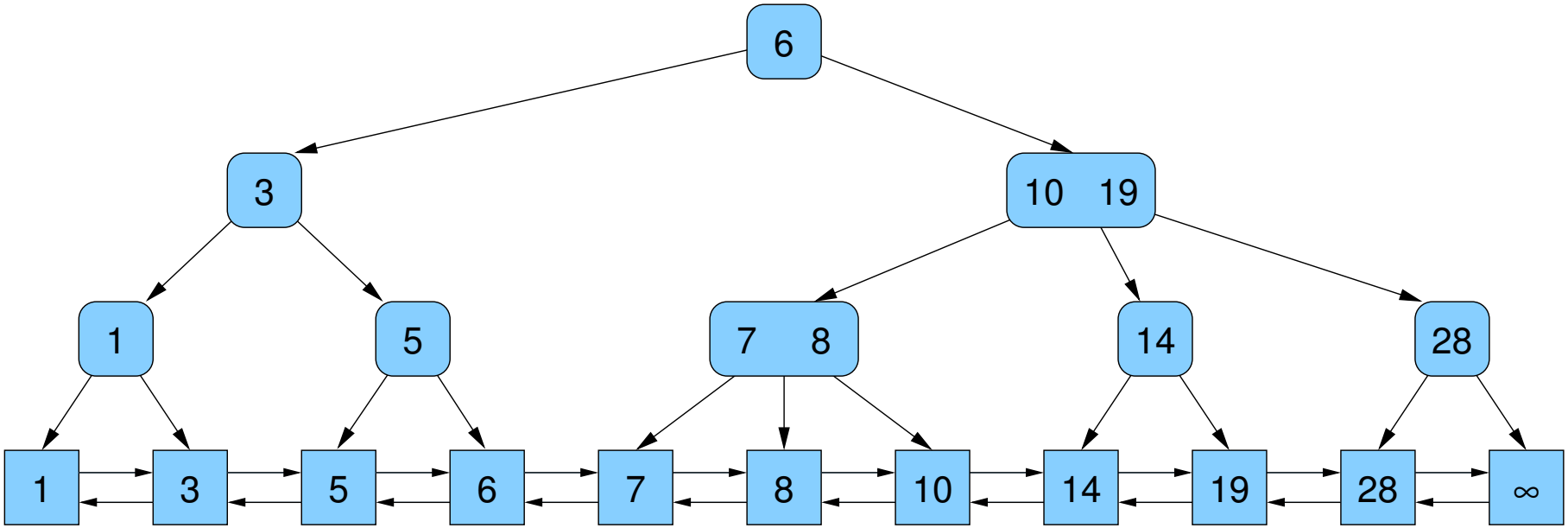
insert(7)



(a,b)-Baum / insert

$a = 2, b = 4$

insert(7)



(a,b)-Baum / insert

Form-Invariante

- alle Blätter haben dieselbe Tiefe, denn
  neues Blatt wird auf der Ebene der anderen eingefügt und
  im Fall einer neuen Wurzel erhöht sich die Tiefe aller Blätter um 1

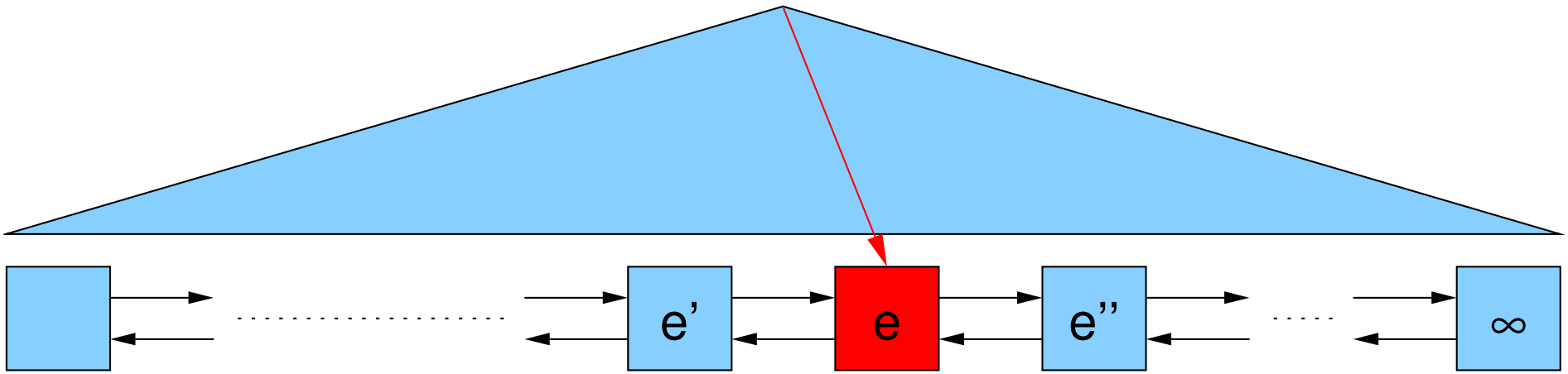
Grad-Invariante

- insert splittet Knoten mit Grad $b+1$ in zwei Knoten
  mit Grad $\lfloor (b+1)/2 \rfloor$ und $\lceil (b+1)/2 \rceil$
- wenn $b \geq 2a-1$, dann sind beide Werte $\geq a$
- wenn Wurzel Grad $b+1$ erreicht und gespalten wird,
  wird neue Wurzel mit Grad 2 erzeugt

(a,b)-Baum

remove(k)

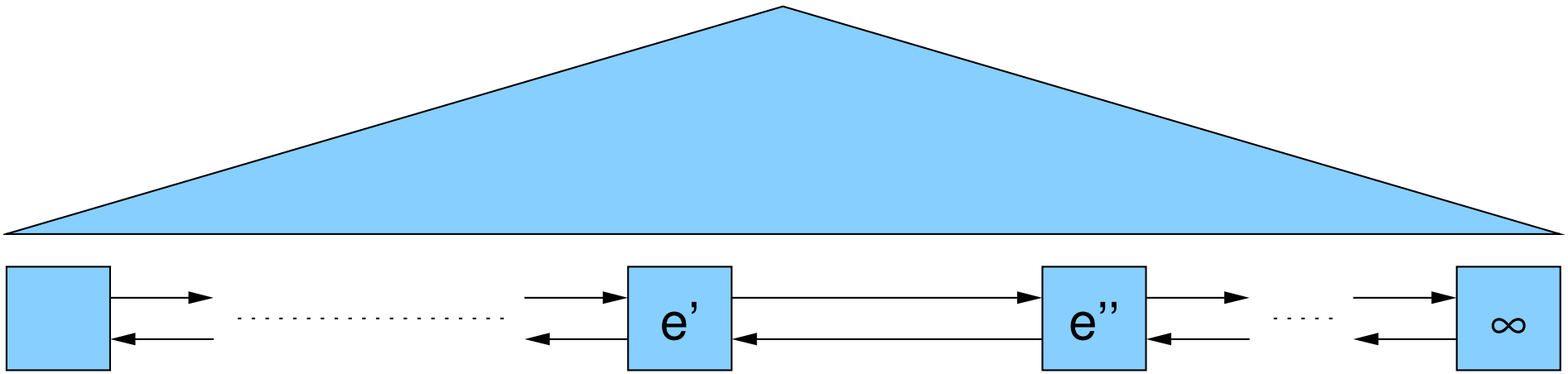
- Abstieg wie bei locate(k) bis Element e in Liste erreicht
- falls key(e) $= k$, entferne e aus Liste (sonst return)



(a,b)-Baum

remove(k)

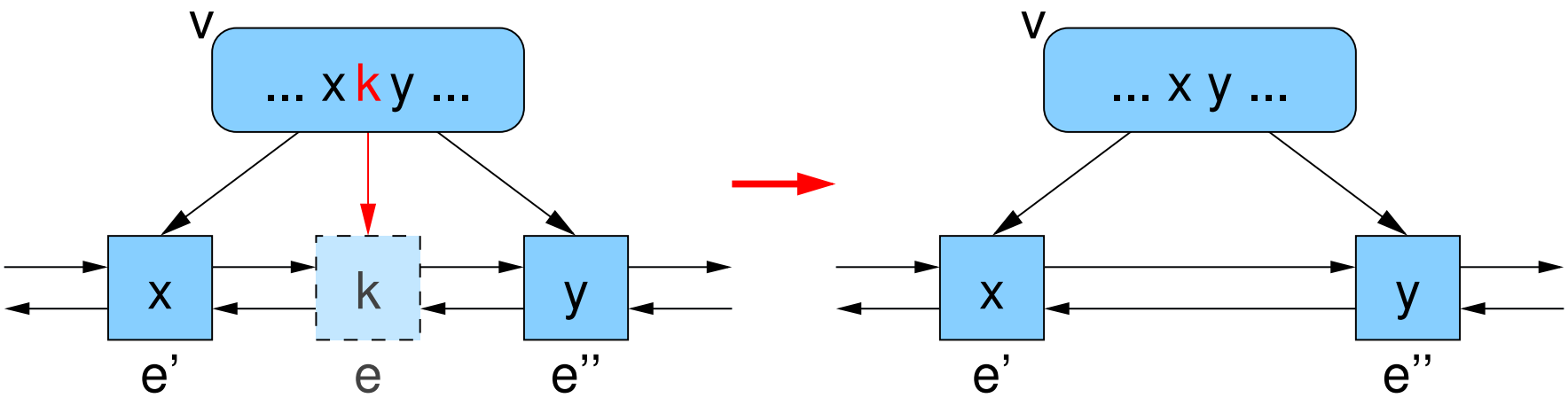
- Abstieg wie bei locate(k) bis Element e in Liste erreicht
- falls key(e) $= k$, entferne e aus Liste (sonst return)



(a,b)-Baum

remove(k)

- entferne Handle auf e und Schlüssel k vom Baumknoten v über e (wenn e rechtestes Kind: Schlüsselvertauschung wie bei binärem Suchbaum)
- falls $d(v) \geq a$, dann fertig

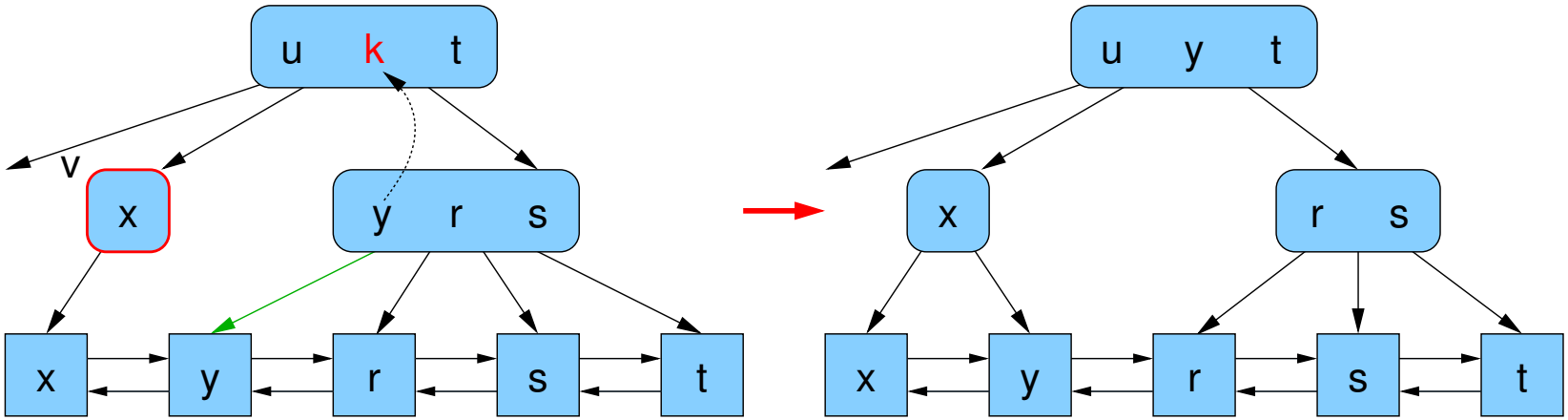


(a,b)-Baum

remove(k)

- falls $d(v) < a$ und ein direkter Nachbar v' von v hat Grad $> a$, nimm Kante von v'

Beispiel: $(2,4)$-Baum

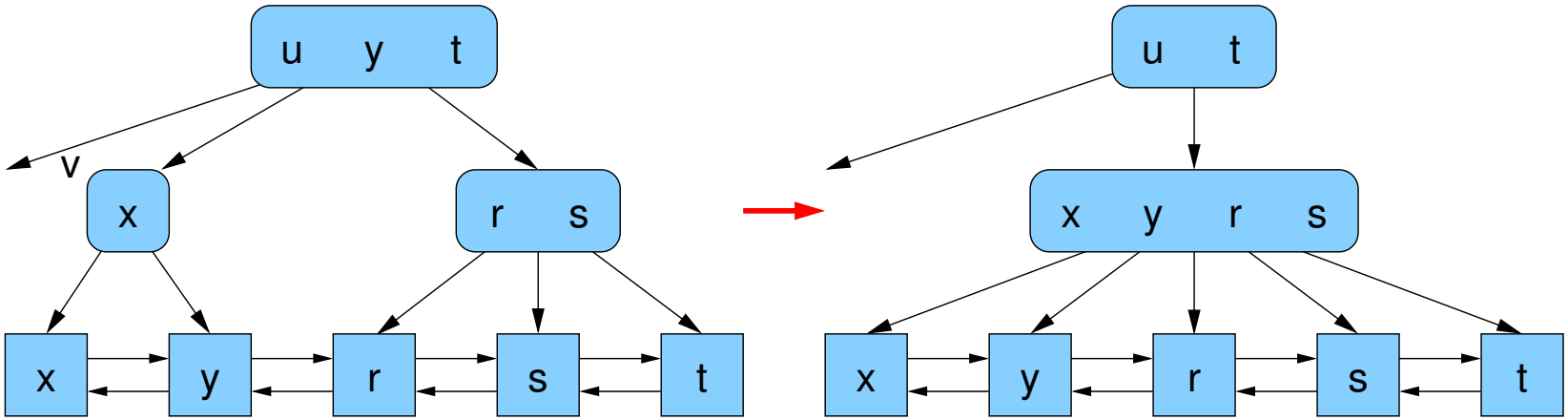


(a,b)-Baum

remove(k)

- falls $d(v) < a$ und kein direkter Nachbar von v hat Grad $> a$, merge v mit Nachbarn

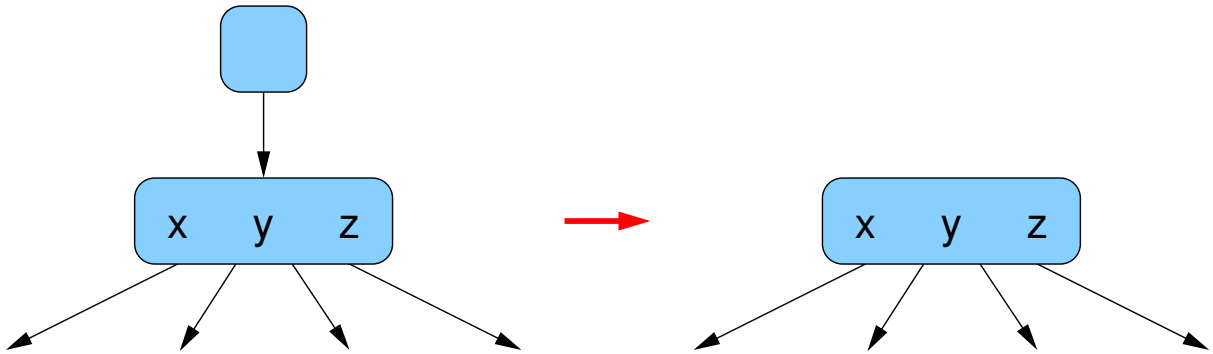
Beispiel: $(3,5)$-Baum



(a,b)-Baum

remove(k)

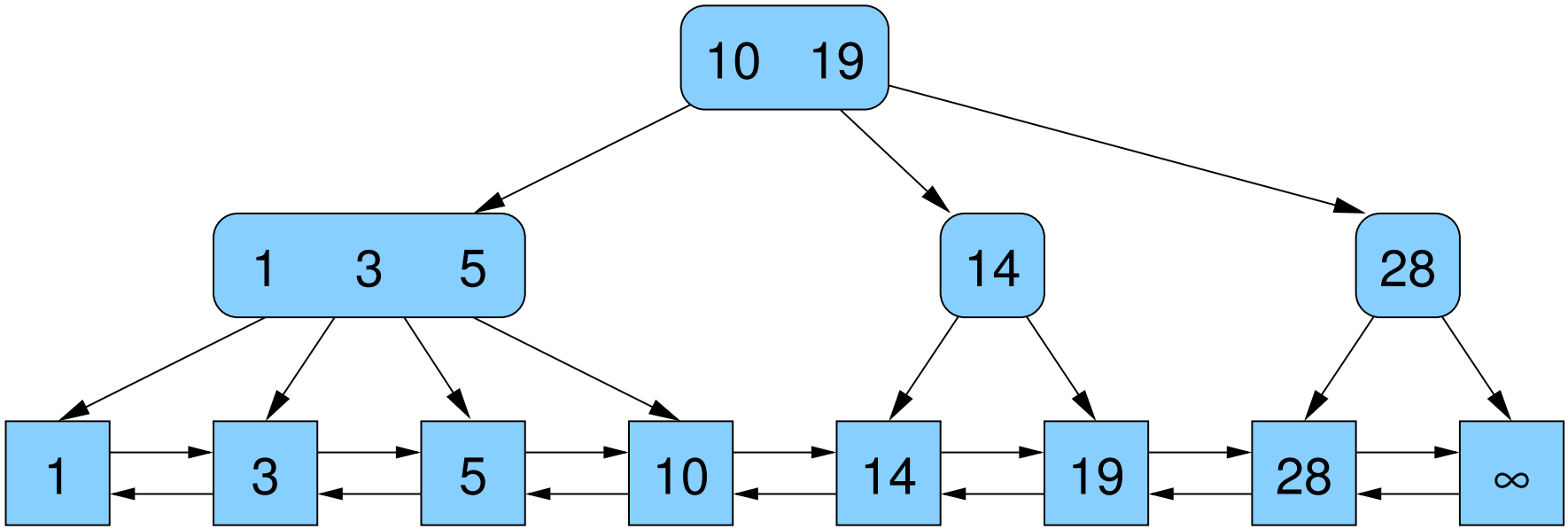
- Verschmelzungen können sich nach oben fortsetzen, ggf. bis zur Wurzel
- falls Grad der Wurzel < 2: entferne Wurzel
  neue Wurzel wird das einzige Kind der alten Wurzel



(a,b)-Baum / remove

$a = 2, b = 4$

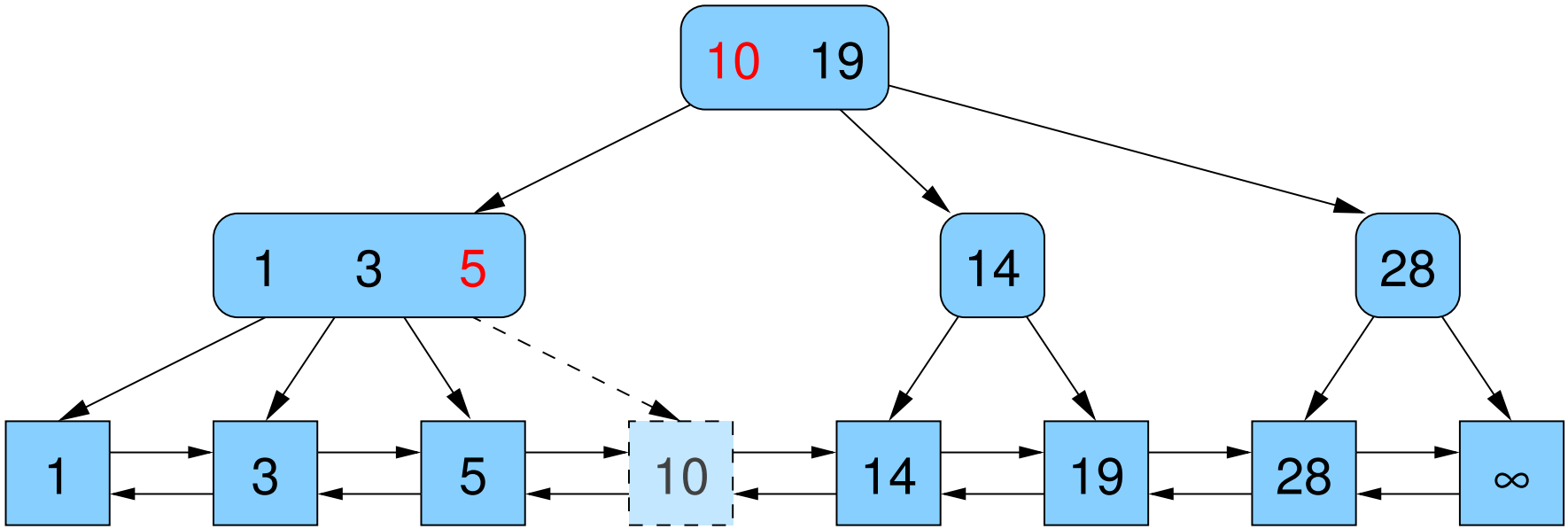
remove(10)



(a,b)-Baum / remove

$a = 2, b = 4$

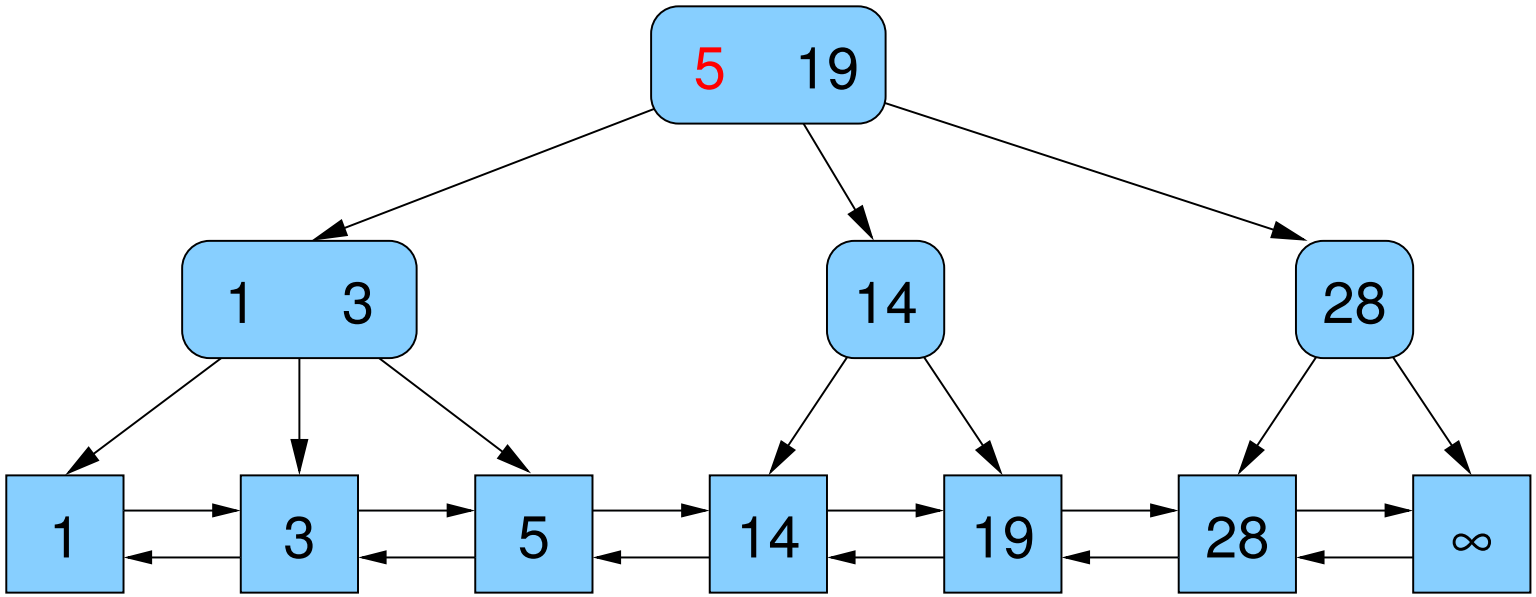
remove(10)



(a,b)-Baum / remove

$a = 2, b = 4$

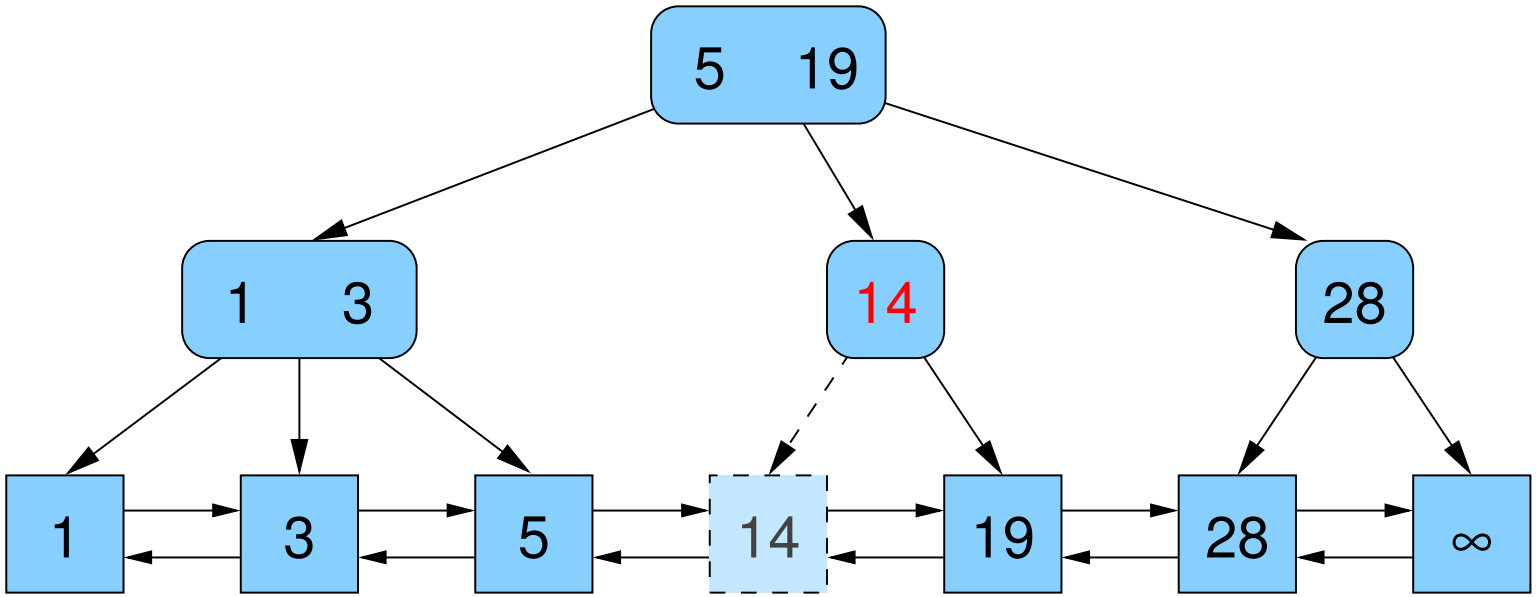
remove(10)



(a,b)-Baum / remove

$a = 2, b = 4$

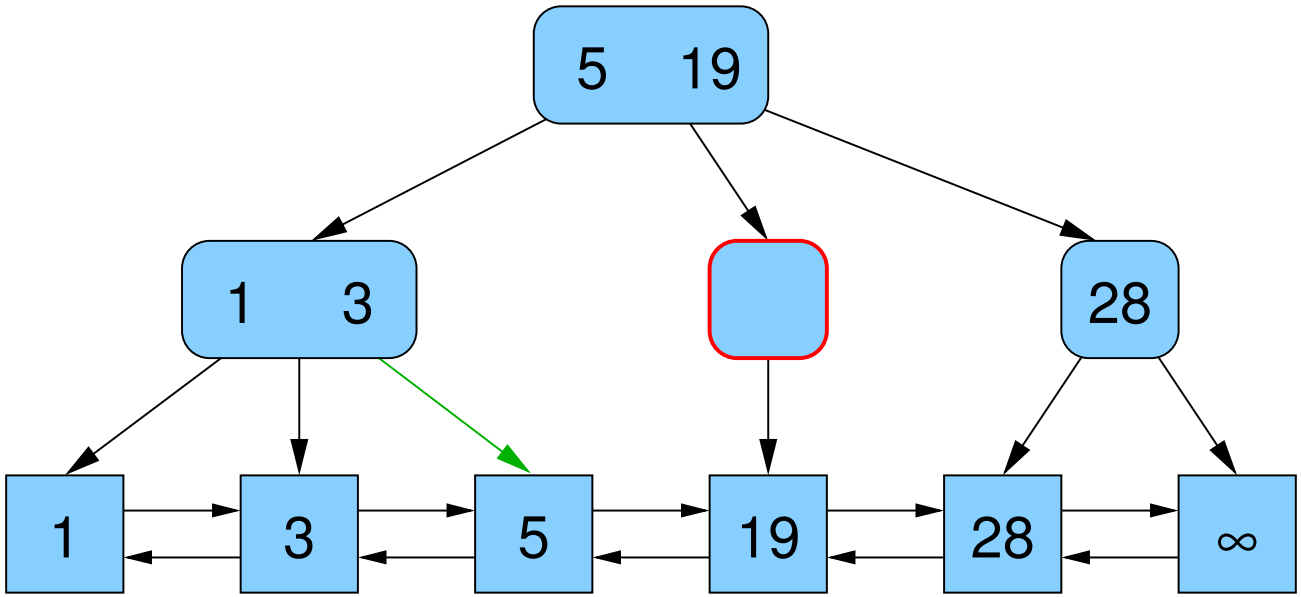
remove(14)



(a,b)-Baum / remove

$a = 2, b = 4$

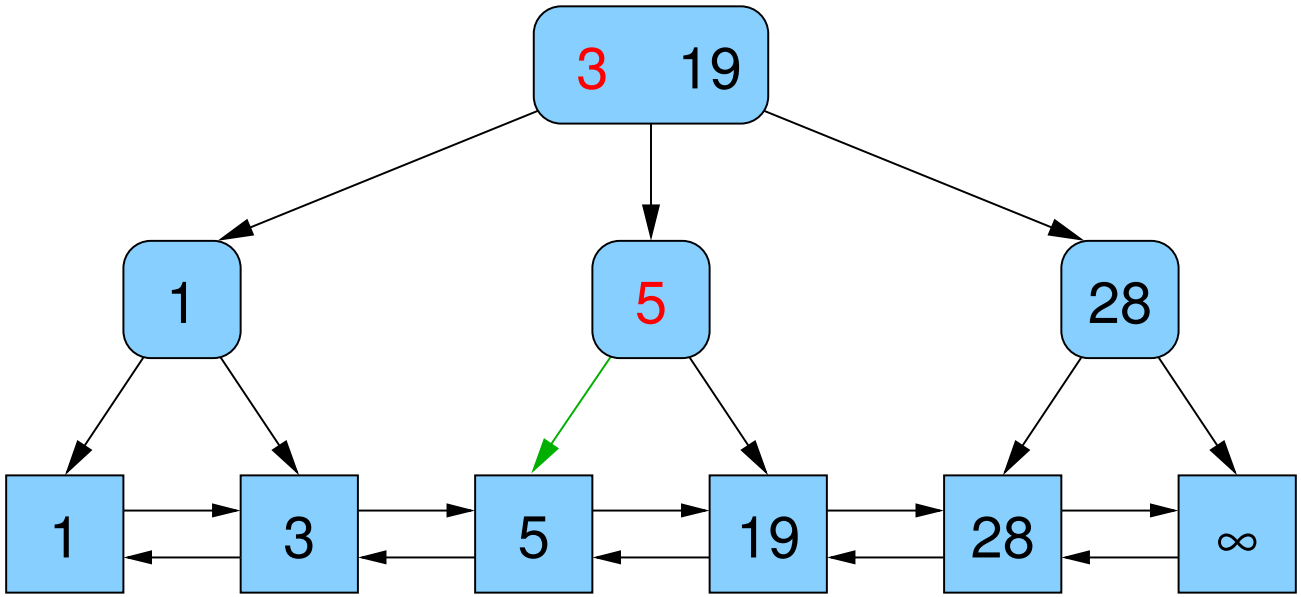
remove(14)



(a, b)-Baum / remove

$a = 2, b = 4$

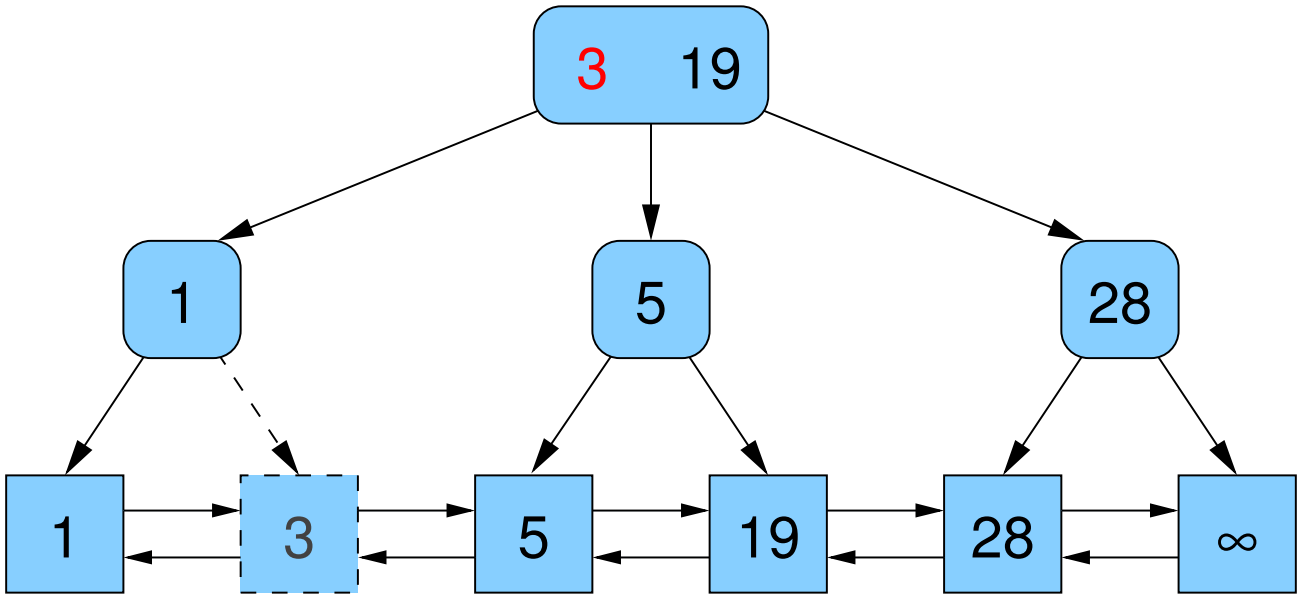
remove(14)



(a,b)-Baum / remove

$a = 2, b = 4$

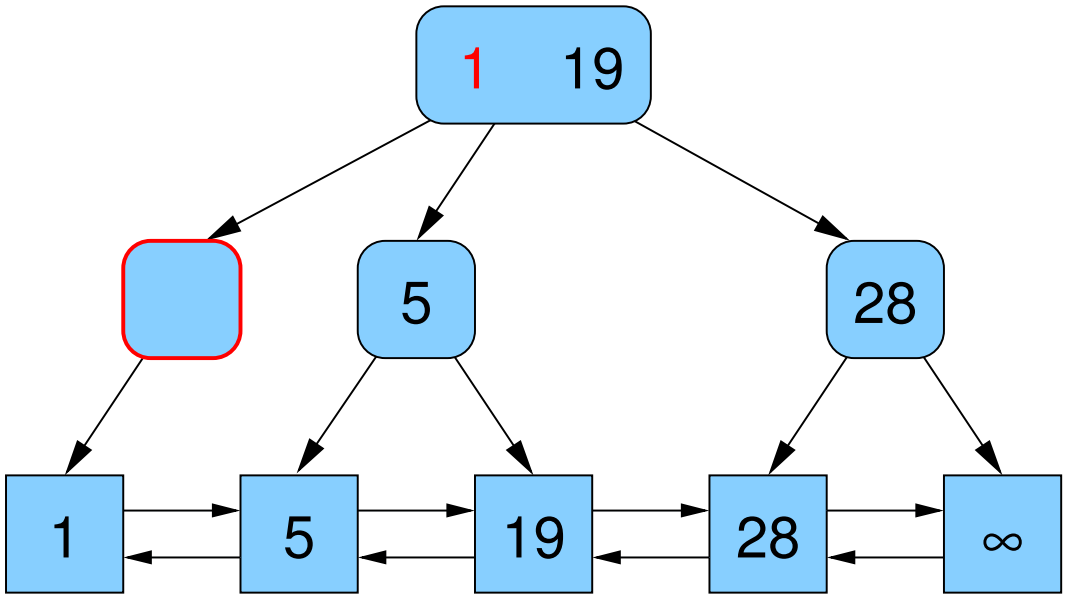
remove(3)



(a,b)-Baum / remove

$a = 2, b = 4$

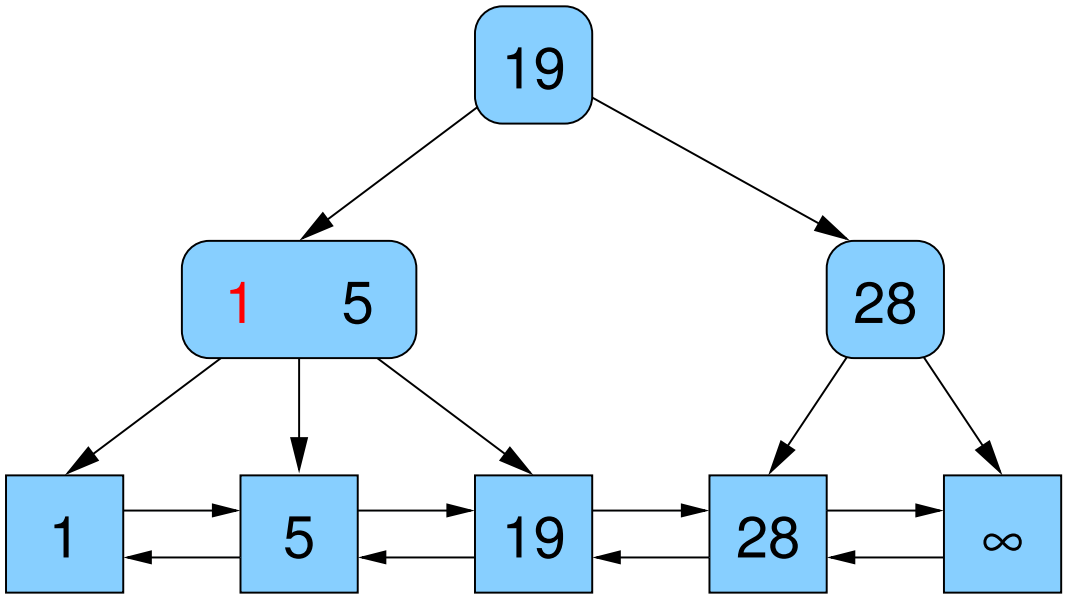
remove(3)



(a,b)-Baum / remove

$a = 2, b = 4$

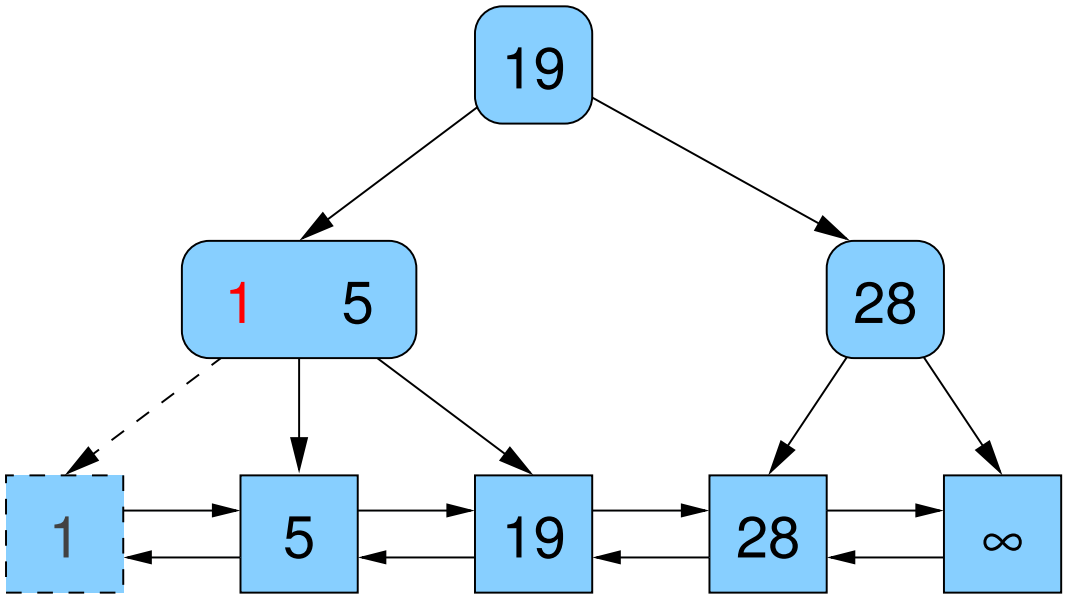
remove(3)



(a,b)-Baum / remove

$a = 2, b = 4$

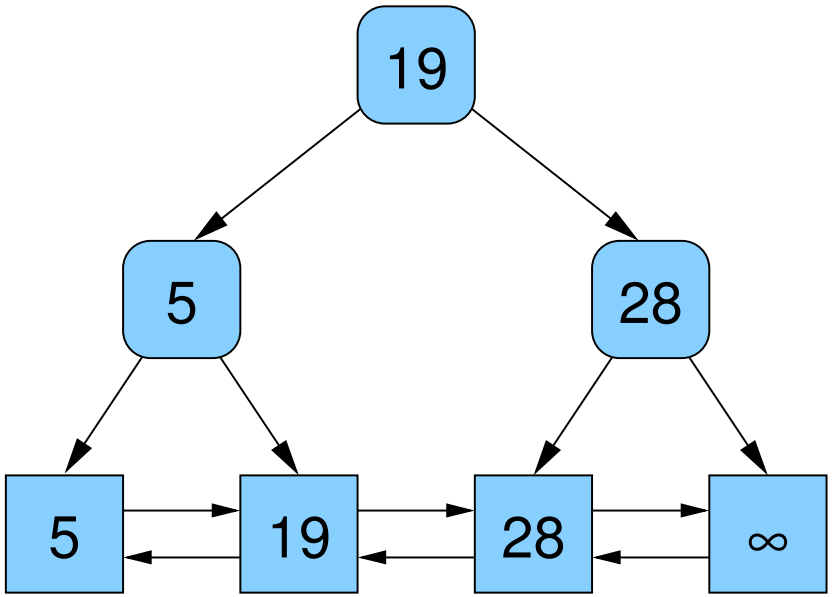
remove(1)



(a,b)-Baum / remove

$a = 2, b = 4$

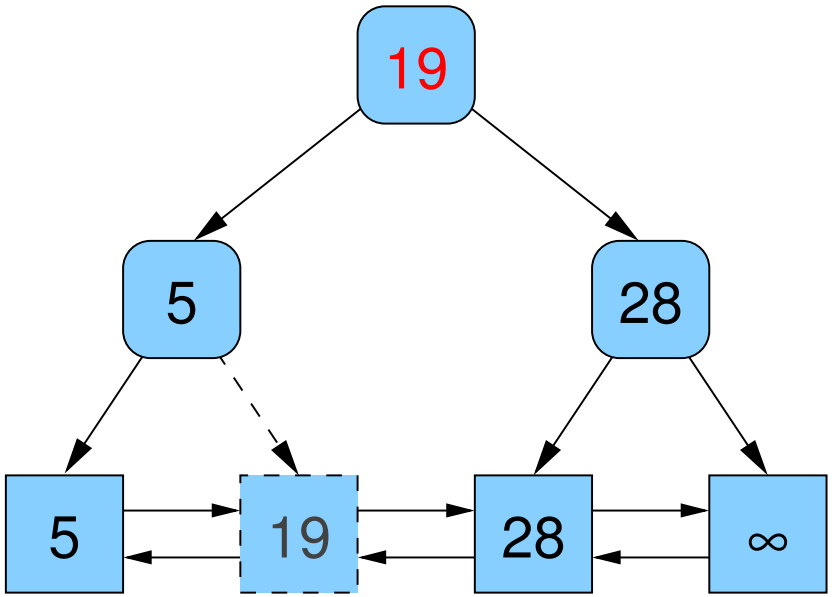
remove(1)



(a,b)-Baum / remove

$a = 2, b = 4$

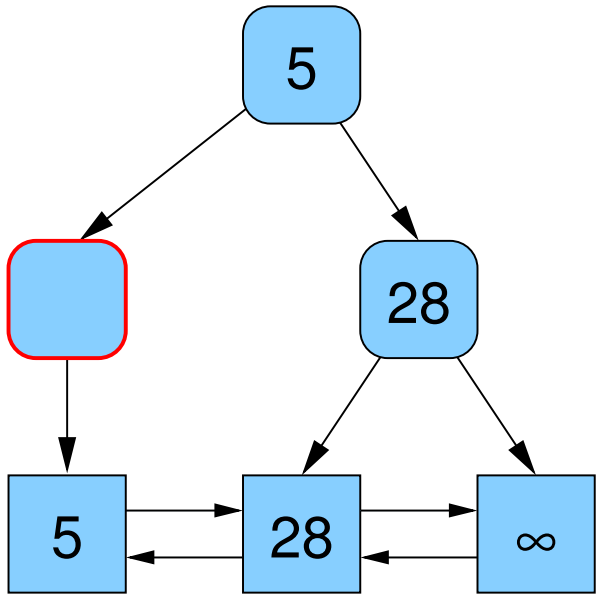
remove(19)



(a,b)-Baum / remove

$a = 2, b = 4$

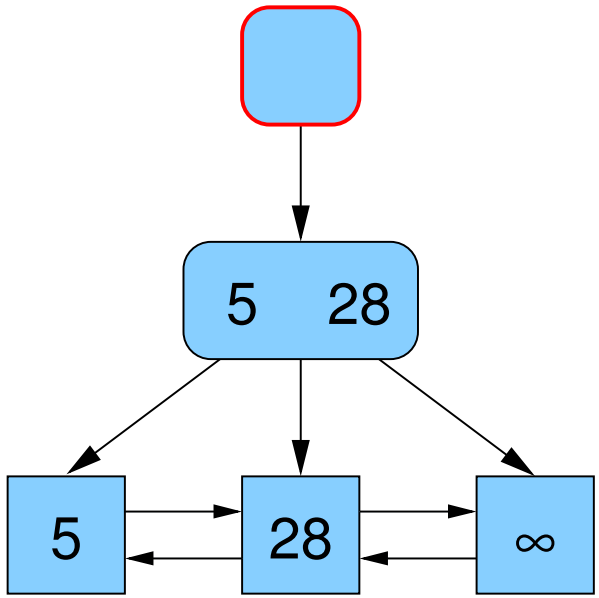
remove(19)



(a,b)-Baum / remove

$a = 2, b = 4$

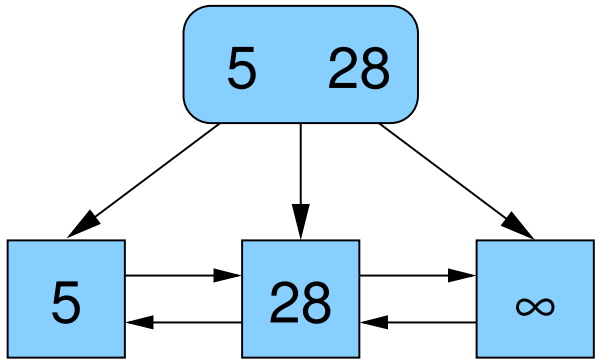
remove(19)



(a,b)-Baum / remove

$a = 2, b = 4$

remove(19)



(a,b)-Baum / remove

Form-Invariante

- alle Blätter behalten dieselbe Tiefe
- falls alte Wurzel entfernt wird, verringert sich die Tiefe aller Blätter

Grad-Invariante

- remove verschmilzt Knoten, die Grad $a-1$ und a haben
- wenn $b \geq 2a-1$, dann ist der resultierende Grad $\leq b$
- remove verschiebt eine Kante von Knoten mit Grad $> a$ zu Knoten mit Grad $a-1$, danach sind beide Grade in $[a,b]$
- wenn Wurzel gelöscht, wurden vorher die Kinder verschmolzen, Grad vom letzten Kind ist also $\geq a$ (und $\leq b$)

Weitere Operationen im (a,b)-Baum

- min / max-Operation

  verwende first / last-Methode der Liste, um das kleinste bzw. größte Element auszugeben

  Zeit: $\mathcal{O}(1)$

- Range queries (Bereichsanfragen)

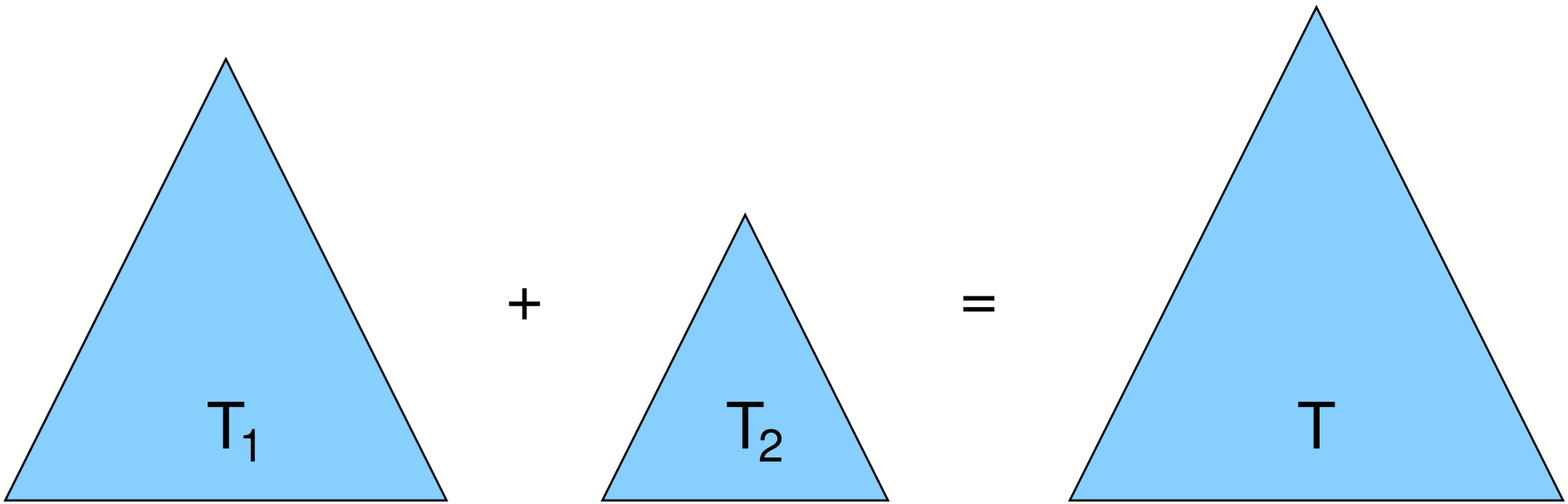
  suche alle Elemente im Bereich $[x,y]$:
  - führe locate(x) aus und
  - durchlaufe die Liste, bis Element $> y$ gefunden wird

  Zeit: $\mathcal{O}(\log n + \text{Ausgabegröße})$

- Konkatenation / Splitting

Konkatenation von (a,b)-Bäumen

- verknüpfe zwei (a,b)-Bäume T_1 und T_2 mit s_1 bzw. s_2 Elementen und Höhe h_1 bzw. h_2 zu (a,b)-Baum T
- Bedingung: Schlüssel in $T_1 \leq$ Schlüssel in T_2



Konkatenation von (a,b)-Bäumen

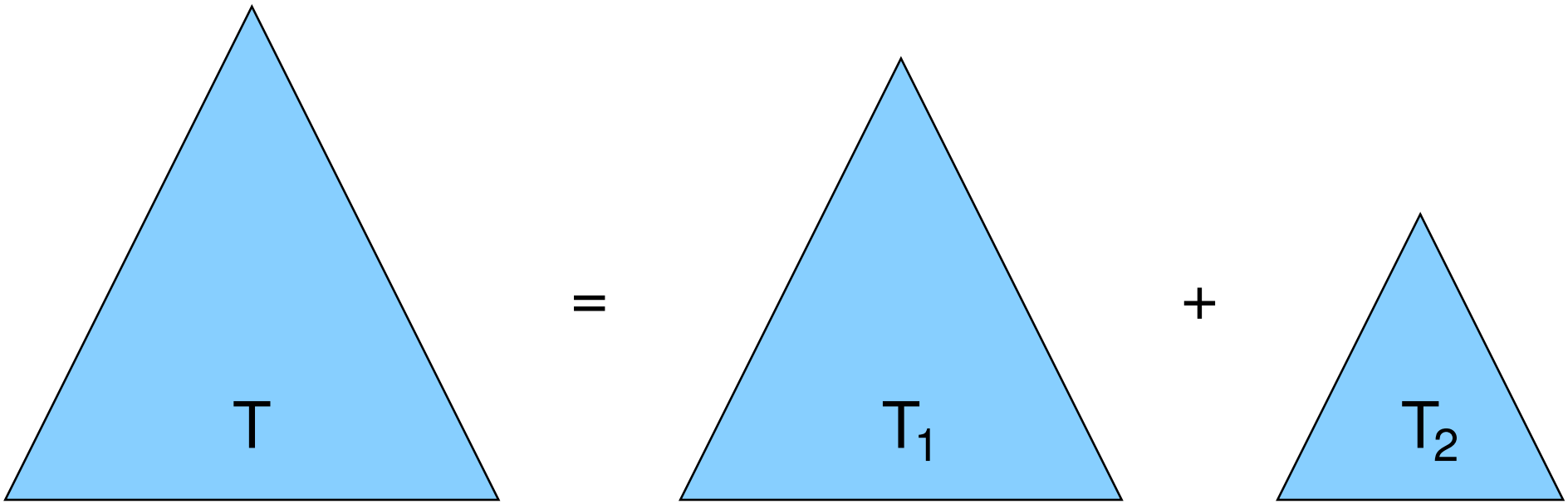
- lösche in T_1 das ∞-Dummy-Element
- wenn danach dessen Vater-Knoten $< a$ Kinder hat,
  dann behandle dies wie bei remove
- verschmelze die Wurzel des niedrigeren Baums mit dem entsprechenden äußersten Knoten des anderen Baums, der sich auf dem gleichen Level befindet
- wenn dieser Knoten danach $> b$ Kinder hat,
  dann behandle dies wie bei insert

$\Rightarrow$ falls Höhe der Bäume explizit gespeichert: Zeit $O(1 + |h_1 - h_2|)$

ansonsten (mit Höhenbestimmung): Zeit $O(1 + \max\{h_1, h_2\})$
$\subseteq O(1 + \log(\max\{s_1, s_2\}))$

Aufspaltung eines (a,b)-Baums

- spalte (a,b)-Baum T bei Schlüssel k in zwei (a,b)-Bäume T_1 und T_2 auf



Aufspaltung eines (a,b)-Baums

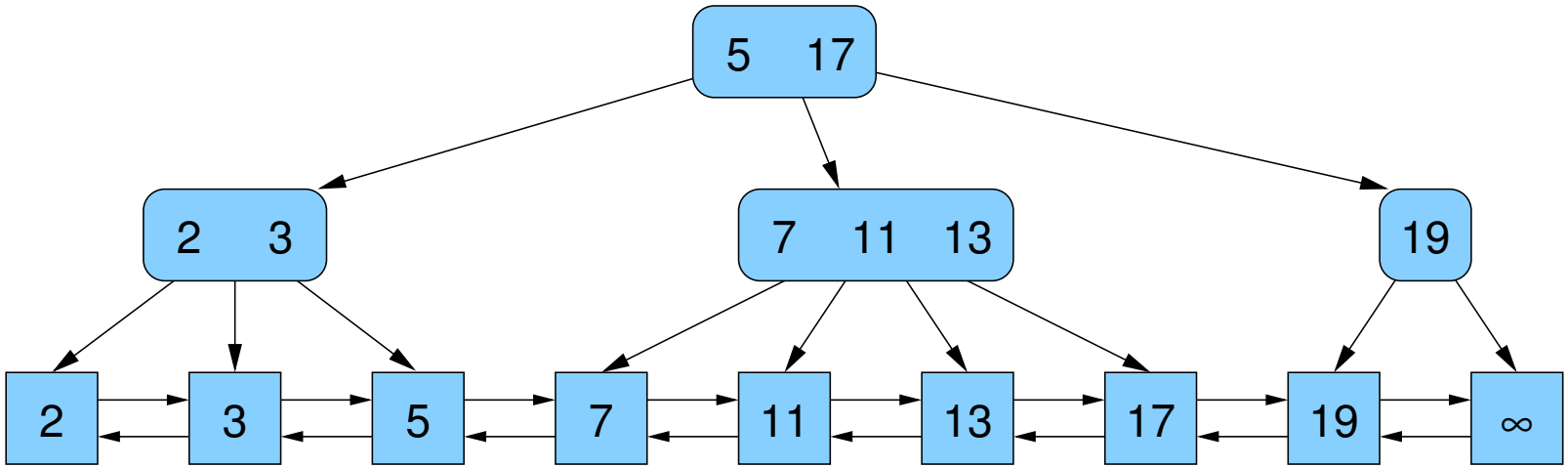
- Sequenz $q = \langle w, \ldots, x, y, \ldots, z \rangle$ soll bei Schlüssel y in Teile $q_1 = \langle w, \ldots, x \rangle$ und $q_2 = \langle y, \ldots, z \rangle$ aufgespalten werden
- betrachte Pfad von Wurzel zum Blatt y
- spalte auf diesem Pfad jeden Knoten v in zwei Knoten v_ℓ und v_r
- v_ℓ bekommt Kinder links vom Pfad,
  v_r bekommt Kinder rechts vom Pfad
  (evt. gibt es Knoten ohne Kinder)
- Knoten mit Kind(ern) werden als Wurzeln von (a,b)-Bäumen interpretiert
- Konkatenation der linken Bäume zusammen mit einem neuen ∞-Dummy ergibt einen Baum für die Elemente bis x
- Konkatenation von $\langle y \rangle$ zusammen mit den rechten Bäumen ergibt einen Baum für die Elemente ab y

Aufspaltung eines (a,b)-Baums

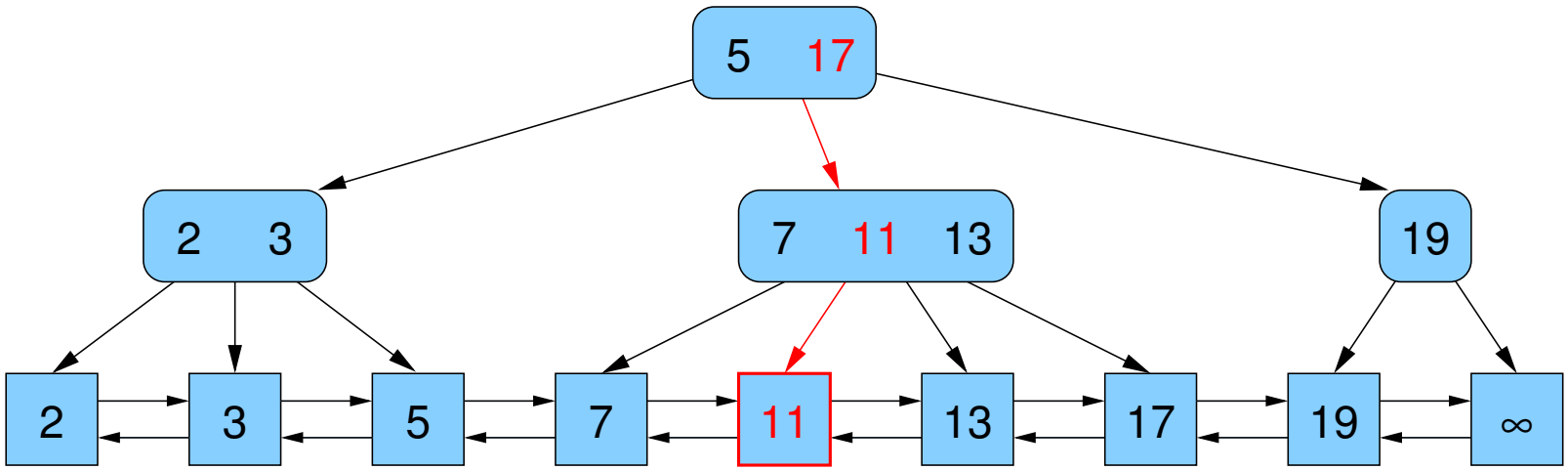
- diese $\mathcal{O}(\log n)$ Konkatenationen können in Gesamtzeit $\mathcal{O}(\log n)$ erledigt werden
- Grund: die linken Bäume haben echt monoton fallende, die rechten echt monoton wachsende Höhe
- Seien z.B. $r_1, r_2, \ldots, r_k$ die Wurzeln der linken Bäume und $h_1 > h_2 > \ldots > h_k$ deren Höhen
- verbinde zuerst r_{k-1} und r_k in Zeit $\mathcal{O}(1 + h_{k-1} - h_k)$, dann r_{k-2} mit dem Ergebnis in Zeit $\mathcal{O}(1 + h_{k-2} - h_{k-1})$, dann r_{k-3} mit dem Ergebnis in Zeit $\mathcal{O}(1 + h_{k-3} - h_{k-2})$ usw.
- Gesamtzeit:

$$\mathcal{O}\left(\sum_{1 \le i < k} (1 + h_i - h_{i+1})\right) = \mathcal{O}(k + h_1 - h_k) \in \mathcal{O}(\log n)$$

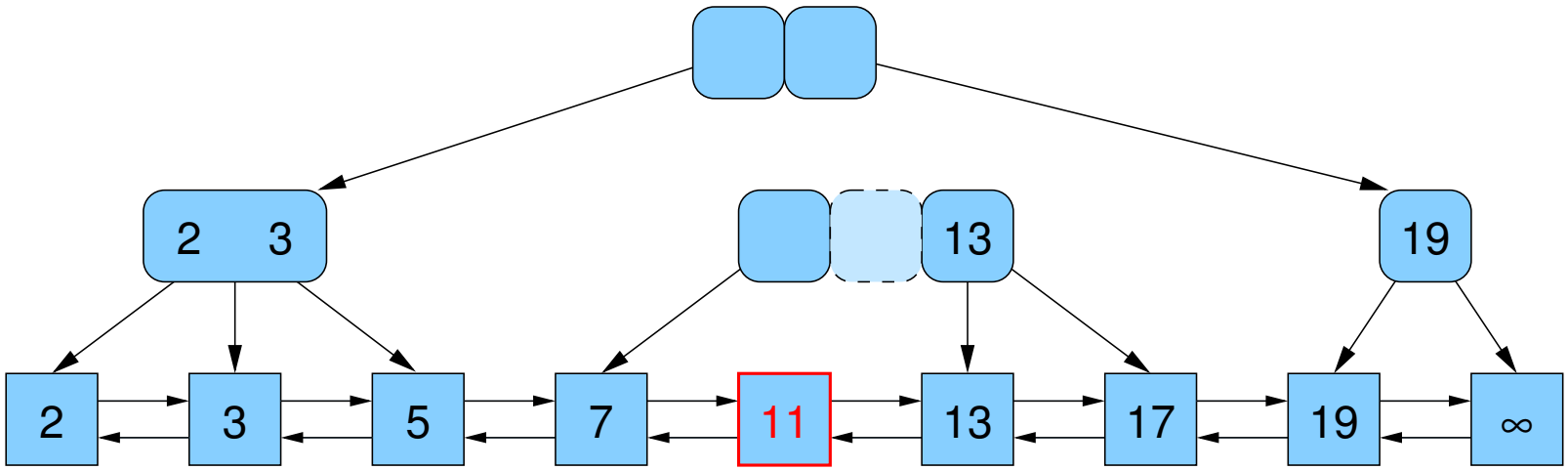
Aufspaltung eines (a, b)-Baums



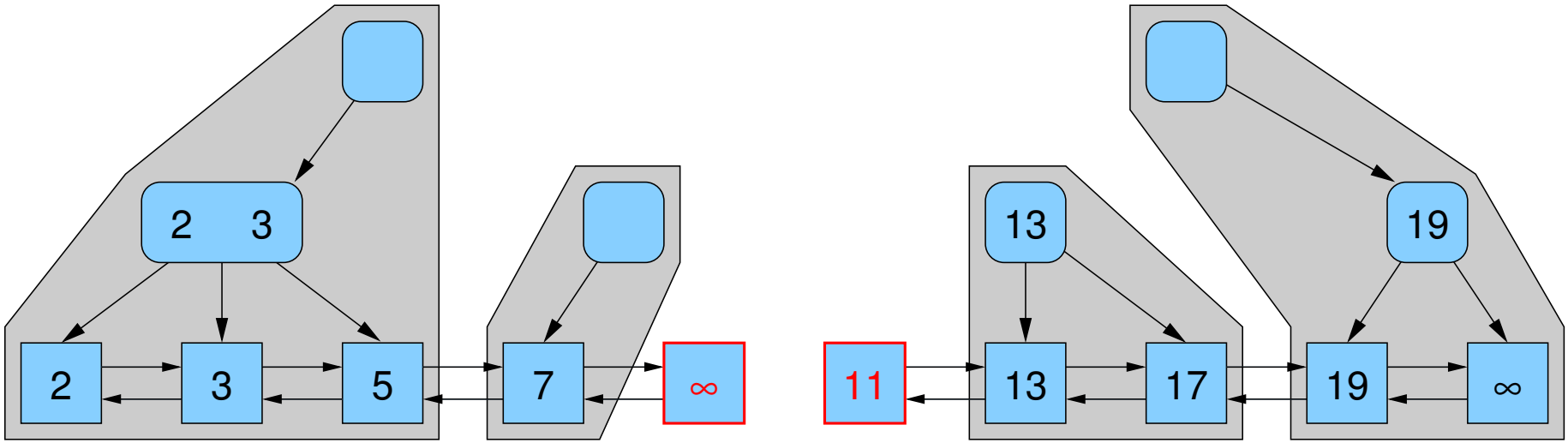
Aufspaltung eines (a, b)-Baums



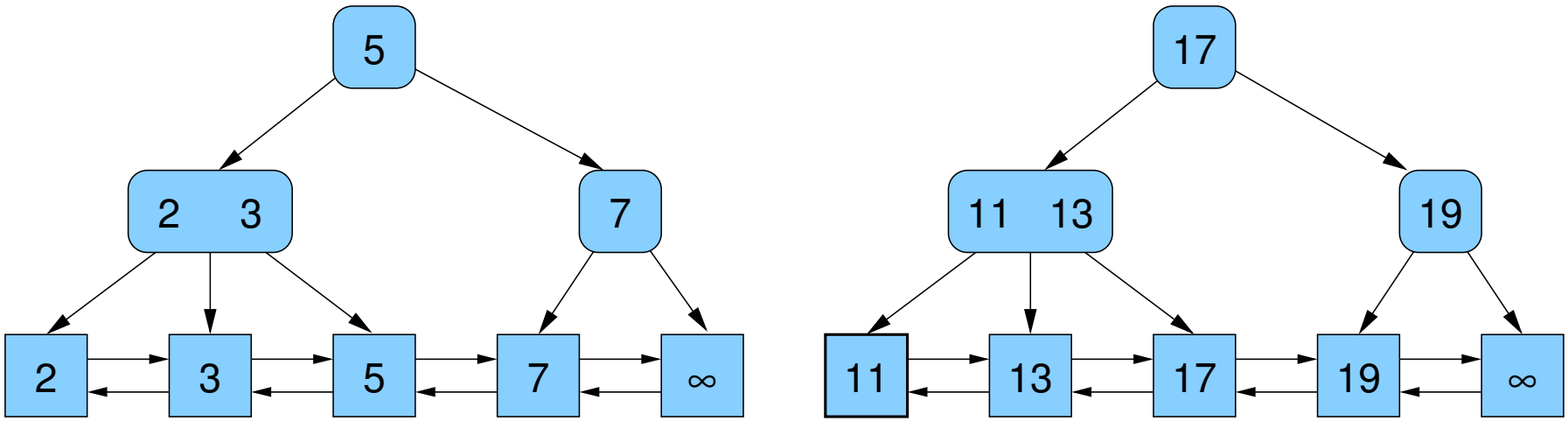
Aufspaltung eines (a, b)-Baums



Aufspaltung eines (a,b)-Baums



Aufspaltung eines (a, b)-Baums



Effizienz von insert / remove-Folgen

Satz

Es gibt eine Folge von n insert- und remove-Operationen auf einem anfangs leeren $(2,3)$-Baum, so dass die Gesamtanzahl der Knotenaufspaltungen und -verschmelzungen in $\Omega(n \log n)$ ist.

Beweis: siehe Übung

Effizienz von insert / remove-Folgen

Satz

Für (a,b)-Bäume, die die erweiterte Bedingung $b \geq 2a$ erfüllen, gilt:

Für jede Folge von n insert- und remove-Operationen auf einem anfangs leeren (a,b)-Baum ist die Gesamtanzahl der Knotenaufspaltungen und -verschmelzungen in $O(n)$.

Beweis: amortisierte Analyse, nicht in dieser Vorlesung

Übersicht

Übersicht

Netzwerk

Objekt bestehend aus

- Elementen und
- Interaktionen bzw. Verbindungen zwischen den Elementen

eher *informales* Konzept

- keine exakte Definition
- Elemente und ihre Verbindungen können ganz unterschiedlichen Charakter haben
- manchmal manifestiert in real existierenden Dingen, manchmal nur gedacht (virtuell)

Beispiele für Netzwerke

- Kommunikationsnetze: Internet, Telefonnetz
- Verkehrsnetze: Straßen-, Schienen-, Flug-, Nahverkehrsnetz
- Versorgungsnetzwerke: Strom, Wasser, Gas, Erdöl
- wirtschaftliche Netzwerke: Geld- und Warenströme, Handel
- biochemische Netzwerke: Metabolische und Interaktionsnetzwerke
- biologische Netzwerke: Gehirn, Ökosysteme
- soziale / berufliche Netzwerke: virtuell oder explizit (Communities)
- Publikationsnetzwerke: Zitationsnetzwerk, Koautor-Netzwerk

Graph

formales / abstraktes Objekt bestehend aus

- Menge von Knoten V (engl. vertices, nodes)
- Menge von Kanten E (engl. edges, lines, links), die jeweils ein Paar von Knoten verbinden
- Menge von Eigenschaften der Knoten und / oder Kanten

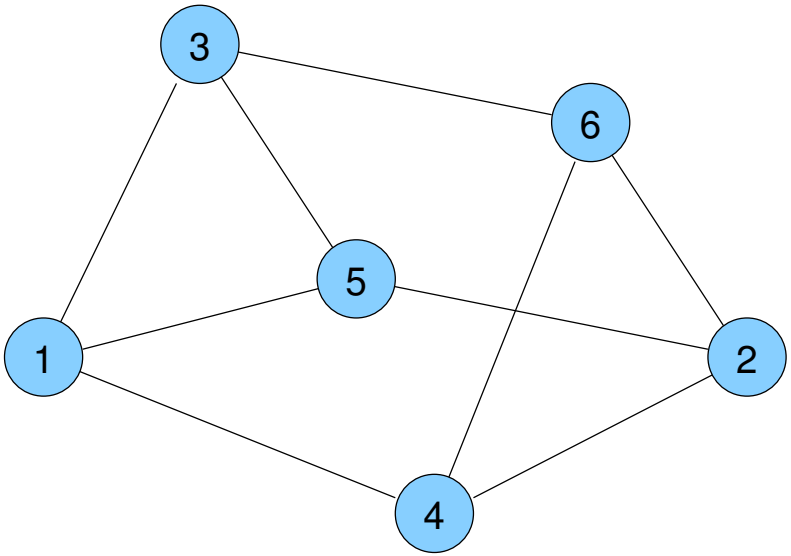
Notation:

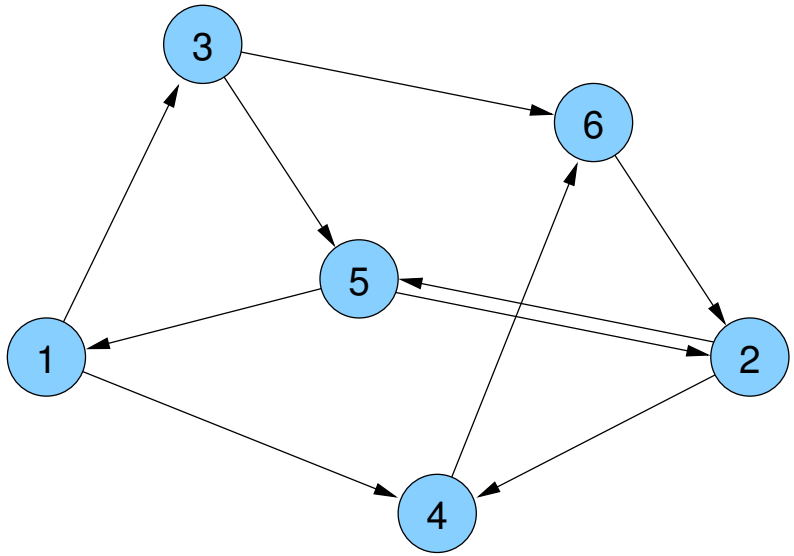
- $G = (V, E)$
  manchmal auch $G = (V, E, w)$ im Fall gewichteter Graphen
- Anzahl der Knoten: $n = |V|$
  Anzahl der Kanten: $m = |E|$

Gerichtete und ungerichtete Graphen

Kanten bzw. Graphen

- ungerichtet: $E \subseteq \{\{v, w\} : v \in V, w \in V\}$
  (ungeordnetes Paar von Knoten bzw. 2-elementige Teilmenge)
- gerichtet: $E \subseteq \{(v, w) : v \in V, w \in V\}$, also $E \subseteq V \times V$
  (geordnetes Paar von Knoten)





Gerichtete und ungerichtete Graphen

Anwendungen:

- Ungerichtete Graphen:

  symmetrische Beziehungen (z.B. $\{v, w\} \in E$ genau dann, wenn Person v und Person w verwandt sind)

- Gerichtete Graphen:

  asymmetrische Beziehungen (z.B. $(v, w) \in E$ genau dann, wenn Person v Person w mag)

  kreisfreie Beziehungen (z.B. $(v, w) \in E$ genau dann, wenn Person v Vorgesetzter von Person w ist

hier:

- Modellierung von ungerichteten durch gerichtete Graphen
- Ersetzung ungerichteter Kanten durch je zwei antiparallele gerichtete Kanten

Nachbarn: Adjazenz, Inzidenz, Grad

- Sind zwei Knoten v und w durch eine Kante e verbunden, dann nennt man
  - v und w adjazent bzw. benachbart
  - v und e inzident (ebenso w und e)

- Anzahl der Nachbarn eines Knotens v: Grad $\deg(v)$

  bei gerichteten Graphen:
  - Eingangsgrad: $\deg^-(v) = |\{(w,v) \in E\}|$
  - Ausgangsgrad: $\deg^+(v) = |\{(v,w) \in E\}|$

Annahmen

- Graph (also Anzahl der Knoten und Kanten) ist endlich
- Graph ist einfach, d.h. E ist eine Menge und keine Multimenge (anderenfalls heißt G Multigraph)
- Graph enthält keine Schleifen (Kanten von v nach v)

Gewichtete Graphen

In Abhängigkeit vom betrachteten Problem wird Kanten und / oder Knoten oft eine Eigenschaft (z.B. eine Farbe oder ein numerischer Wert, das Gewicht) zugeordnet (evt. auch mehrere), z.B.

- Distanzen (in Längen- oder Zeiteinheiten)
- Kosten
- Kapazitäten / Bandbreite
- Ähnlichkeiten
- Verkehrsdichte

Wir nennen den Graphen dann

- knotengewichtet bzw.
- kantengewichtet

Beispiel: $w : E \mapsto \mathbb{R}$
Schreibweise: $w(e)$ für das Gewicht einer Kante $e \in E$

Wege, Pfade und Kreise

- Weg (engl. *walk*) in einem Graphen $G = (V, E)$: alternierende Folge von Knoten und Kanten $x_0, e_1, \ldots, e_k, x_k$, so dass
  - $\forall i \in [0, k] : x_i \in V$ und
  - $\forall i \in [1, k] : e_i = \{x_{i-1}, x_i\}$ bzw. $e_i = (x_{i-1}, x_i) \in E$.
- *Länge* eines Weges: Anzahl der enthaltenen Kanten

- Ein Weg ist ein Pfad, falls er (in sich) kantendisjunkt ist, falls also gilt: $e_i \neq e_j$ für $i \neq j$.
- Ein Pfad ist ein einfacher Pfad, falls er (in sich) knotendisjunkt ist, falls also gilt: $x_i \neq x_j$ für $i \neq j$.

- Ein Weg heißt *Kreis* (engl. *cycle*), falls $x_0 = x_k$.

Operationen

Graph G: Datenstruktur (Typ / Klasse, Variable / Objekt) für Graphen

Node: Datenstruktur für Knoten, Edge: Datenstruktur für Kanten

Operationen:

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

Operationen

Anzahl der Knoten konstant

$\Rightarrow$ $V = \{0, \ldots, n-1\}$

- (Knotenschlüssel durchnummeriert)

Anzahl der Knoten variabel

- Hashing kann verwendet werden für ein Mapping der n Knoten in den Bereich $\{0, \ldots, \mathcal{O}(n)\}$

$\Rightarrow$ nur konstanter Faktor der Vergrößerung gegenüber statischer Datenstruktur

Übersicht

Graphrepräsentation

Darstellung von Graphen im Computer?

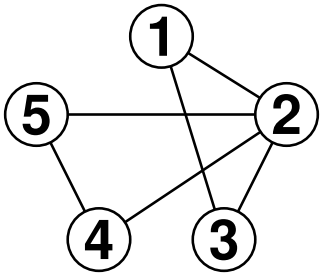
Vor- und Nachteile bei z.B. folgenden Fragen:

- Sind zwei gegebene Knoten v und w adjazent?
- Was sind die Nachbarn eines Knotens?
- Welche Knoten sind (direkte oder indirekte) Vorgänger bzw. Nachfolger eines Knotens v in einem gerichteten Graphen?
- Wie aufwendig ist das Einfügen oder Löschen eines Knotens bzw. einer Kante?

Graphrepräsentationen

- Kantenliste
- Adjazenzmatrix
- Inzidenzmatrix
- Adjazenzarray
- Adjazenzliste
- implizit

Kantenliste



$\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{2,5\},\{4,5\}$

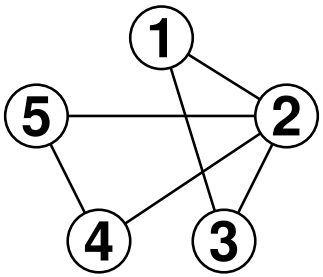
Vorteil:

- Speicherbedarf $\mathcal{O}(m+n)$
- Einfügen von Knoten und Kanten in $\mathcal{O}(1)$
- Löschen von Kanten per Handle in $\mathcal{O}(1)$

Nachteil:

- *G*.find(Key *i*, Key *j*): im worst case $\Theta(m)$
- *G*.remove(Key *i*, Key *j*): im worst case $\Theta(m)$
- Nachbarn nur in $\mathcal{O}(m)$ feststellbar

Adjazenzmatrix



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

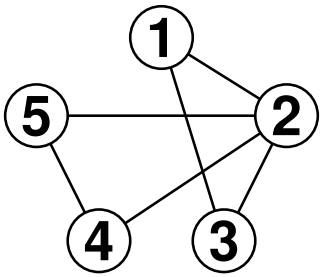
Vorteil:

- in $\mathcal{O}(1)$ feststellbar, ob zwei Knoten Nachbarn sind
- ebenso Einfügen und Löschen von Kanten

Nachteil:

- kostet $\Theta(n^2)$ Speicher, auch bei Graphen mit $o(n^2)$ Kanten
- Finden aller Nachbarn eines Knotens kostet $\mathcal{O}(n)$
- Hinzufügen neuer Knoten ist schwierig

Inzidenzmatrix

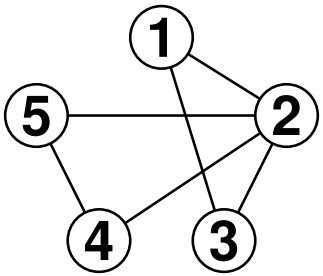


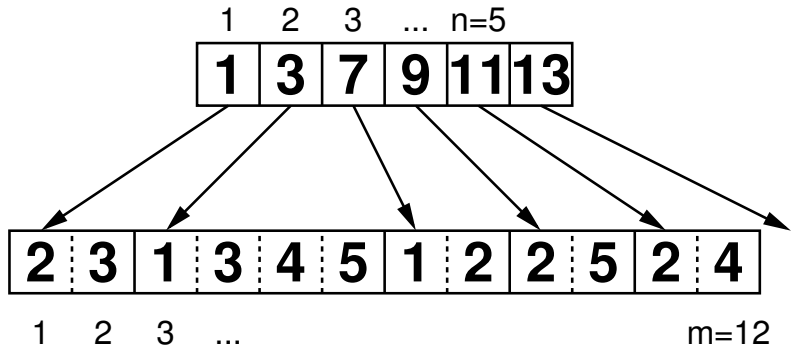
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Nachteil:

- kostet $\Theta(mn)$ Speicher

Adjazenzarray





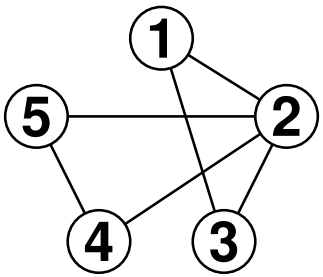
Vorteil:

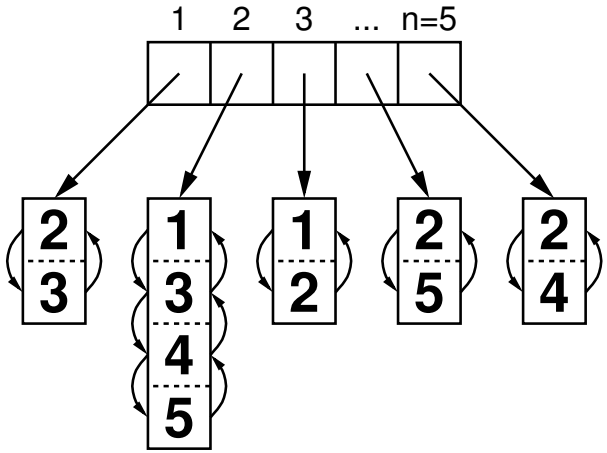
- Speicherbedarf:
  gerichtete Graphen: $n + m + \Theta(1)$
  (hier noch kompakter als Kantenliste mit $2m$)
  ungerichtete Graphen: $n + 2m + \Theta(1)$

Nachteil:

- Einfügen und Löschen von Kanten ist schwierig, deshalb nur für *statische* Graphen geeignet

Adjazenzliste





Unterschiedliche Varianten:
einfach/doppelt verkettet, linear/zirkulär

Vorteil:

- Einfügen von Kanten in $\mathcal{O}(d)$ oder $\mathcal{O}(1)$
- Löschen von Kanten in $\mathcal{O}(d)$ (per Handle in $\mathcal{O}(1)$)
- mit unbounded arrays etwas cache-effizienter

Nachteil:

- Zeigerstrukturen verbrauchen relativ viel Platz und Zugriffszeit

Adjazenzliste + Hashtabelle

- speichere Adjazenzliste (Liste von adjazenten Knoten bzw. inzidenten Kanten zu jedem Knoten)
- speichere Hashtabelle, die zwei Knoten auf einen Zeiger abbildet, der dann auf die ggf. vorhandene Kante verweist

Zeitaufwand:

- *G*.find(Key *i*, Key *j*): $\mathcal{O}(1)$ (worst case)
- *G*.insert(Edge *e*): $\mathcal{O}(1)$ (im Mittel)
- *G*.remove(Key *i*, Key *j*): $\mathcal{O}(1)$ (im Mittel)

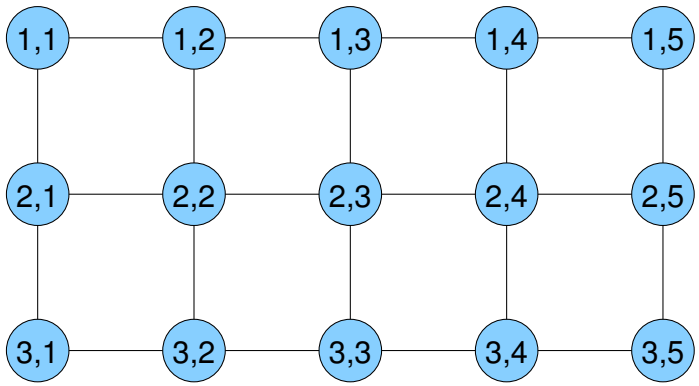
Speicheraufwand: $\mathcal{O}(n+m)$

Implizite Repräsentation

Beispiel: Gitter-Graph (grid graph)

- definiert durch zwei Parameter k und ℓ

$$
\begin{aligned}
V &= [1,\ldots,k] \times [1,\ldots,\ell] \\
E &= \Big\{((i,j),(i,j')) \in V^2 : |j-j'| = 1\Big\} \cup \\
&\quad \Big\{((i,j),(i',j)) \in V^2 : |i-i'| = 1\Big\}
\end{aligned}
$$



- Kantengewichte könnten in 2 zweidimensionalen Arrays gespeichert werden:
  eins für waagerechte und eins für senkrechte Kanten

Übersicht

Graphtraversierung

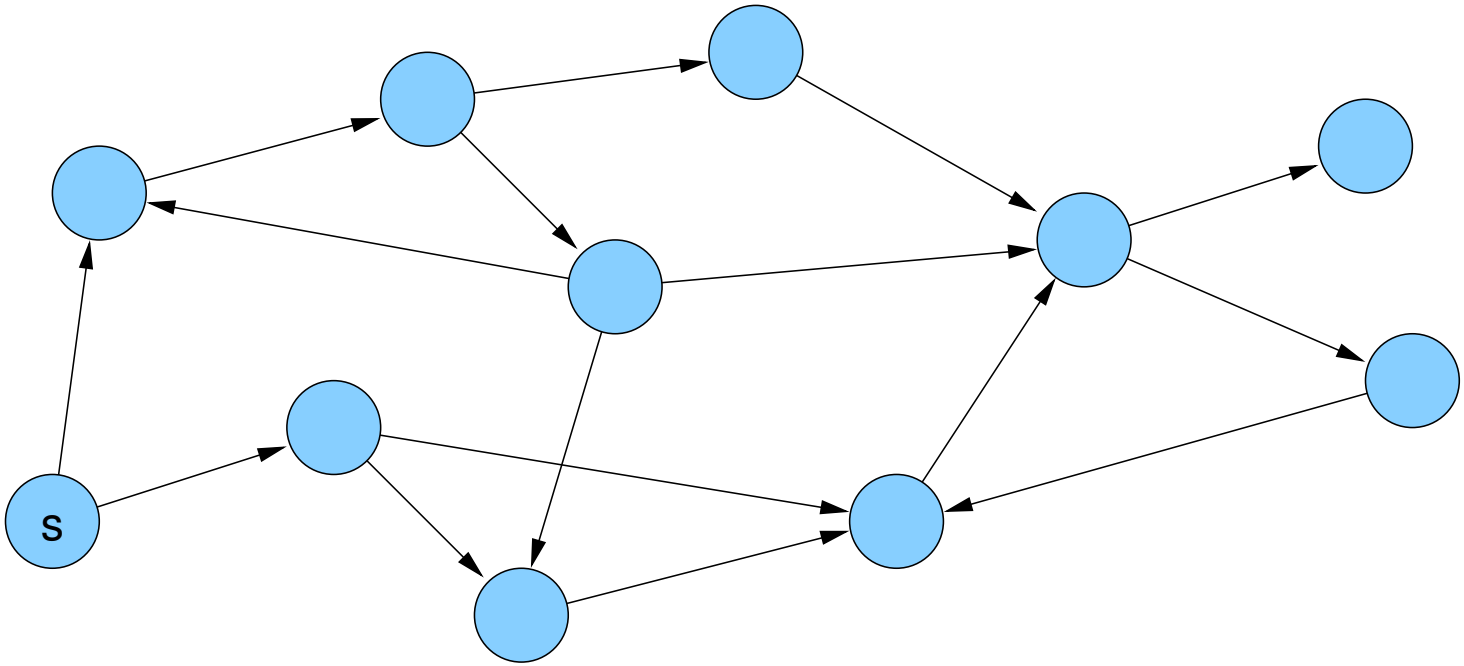
Problem:

Wie kann man die Knoten eines Graphen systematisch durchlaufen?

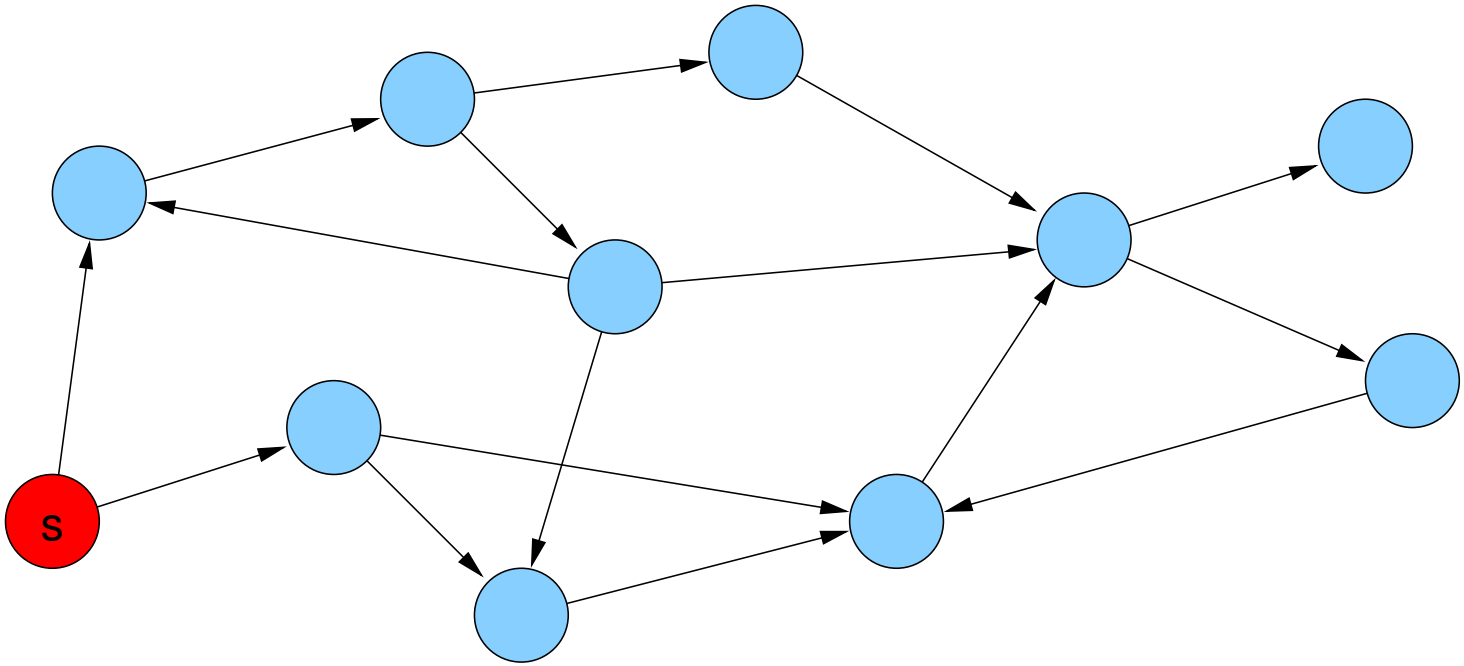
Grundlegende Strategien:

- Breitensuche (breadth-first search, BFS)
- Tiefensuche (depth-first search, DFS)

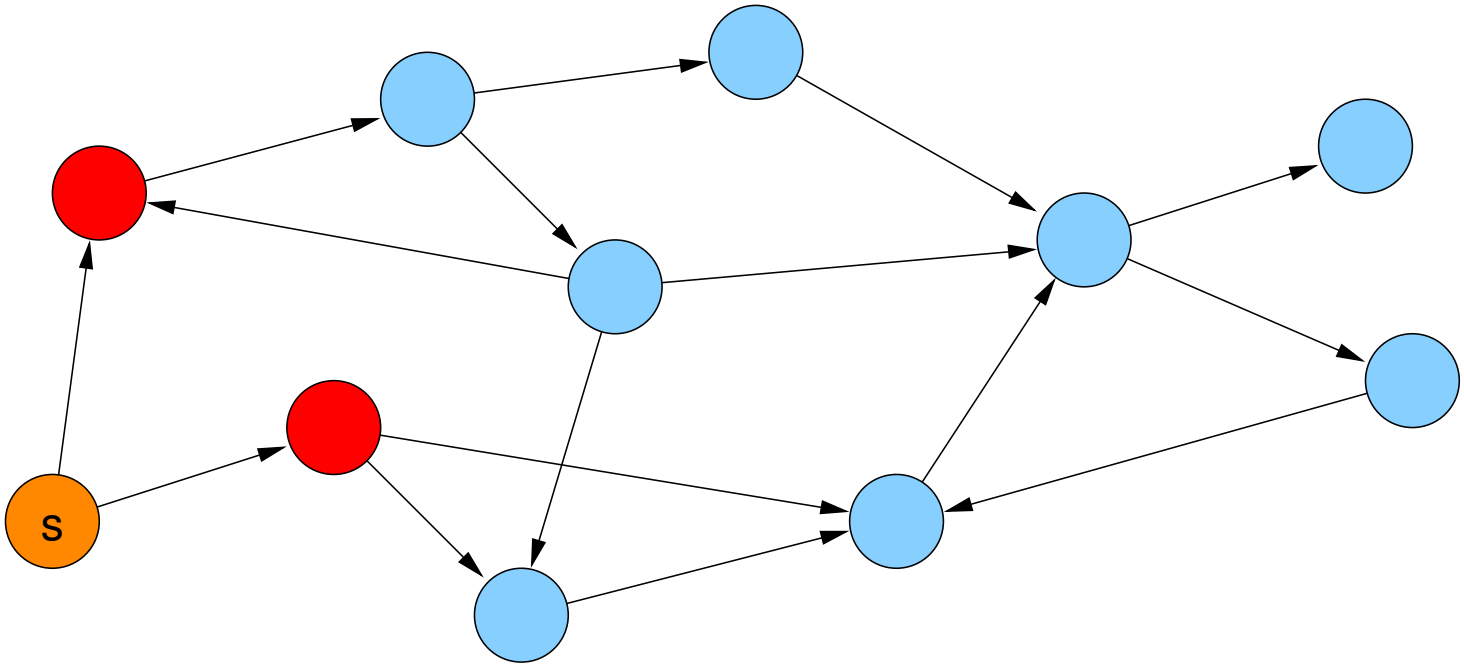
Breitensuche



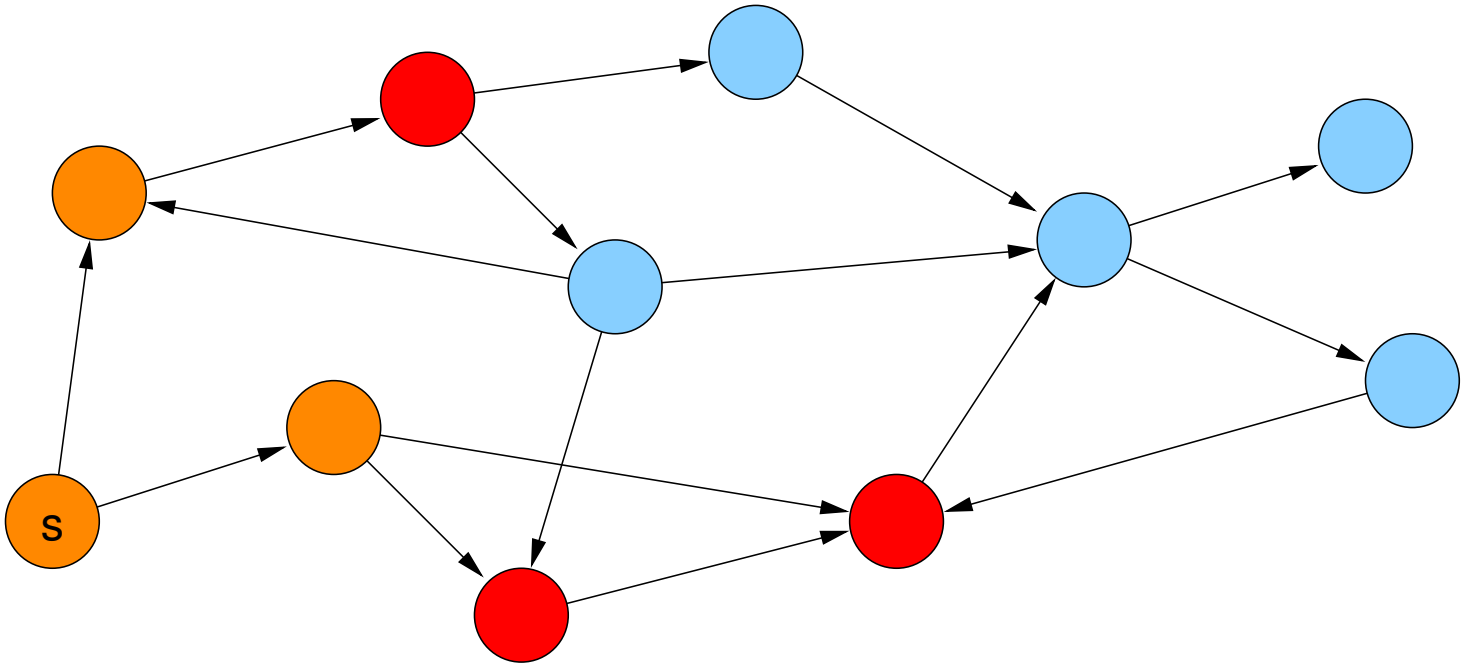
Breitensuche



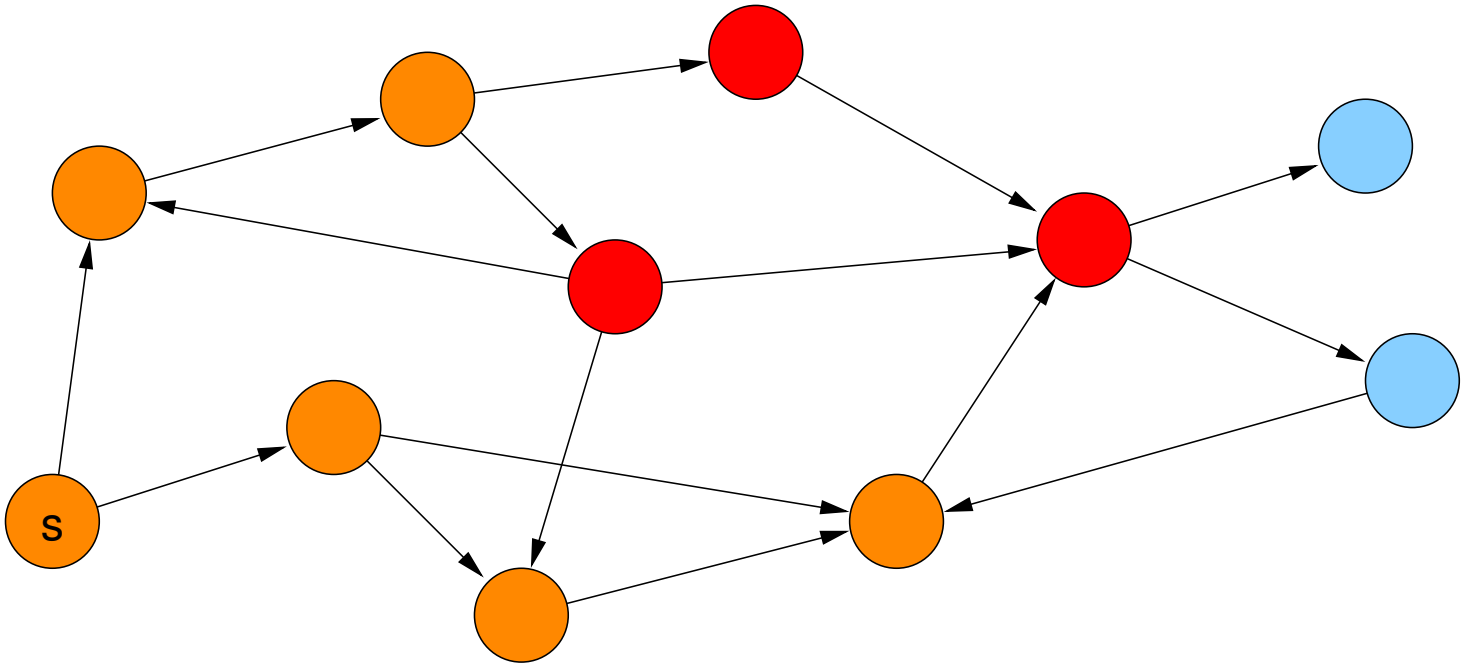
Breitensuche



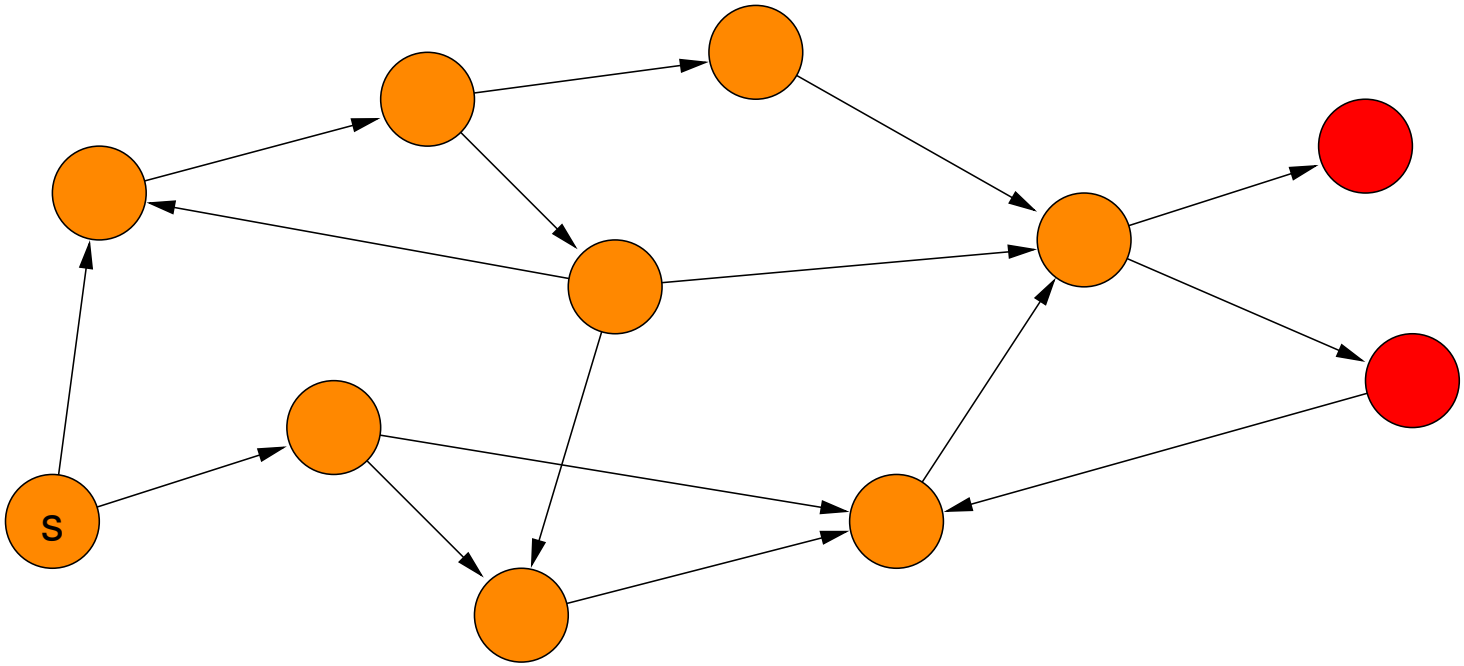
Breitensuche



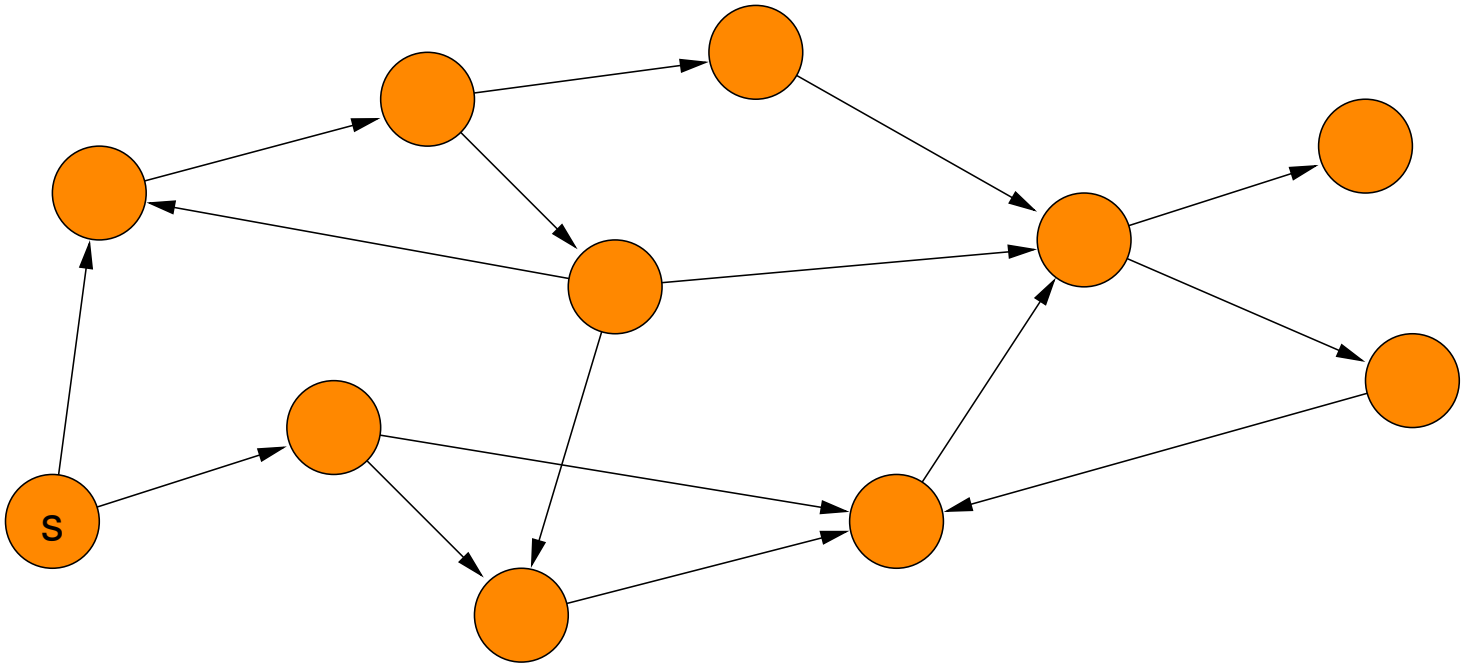
Breitensuche



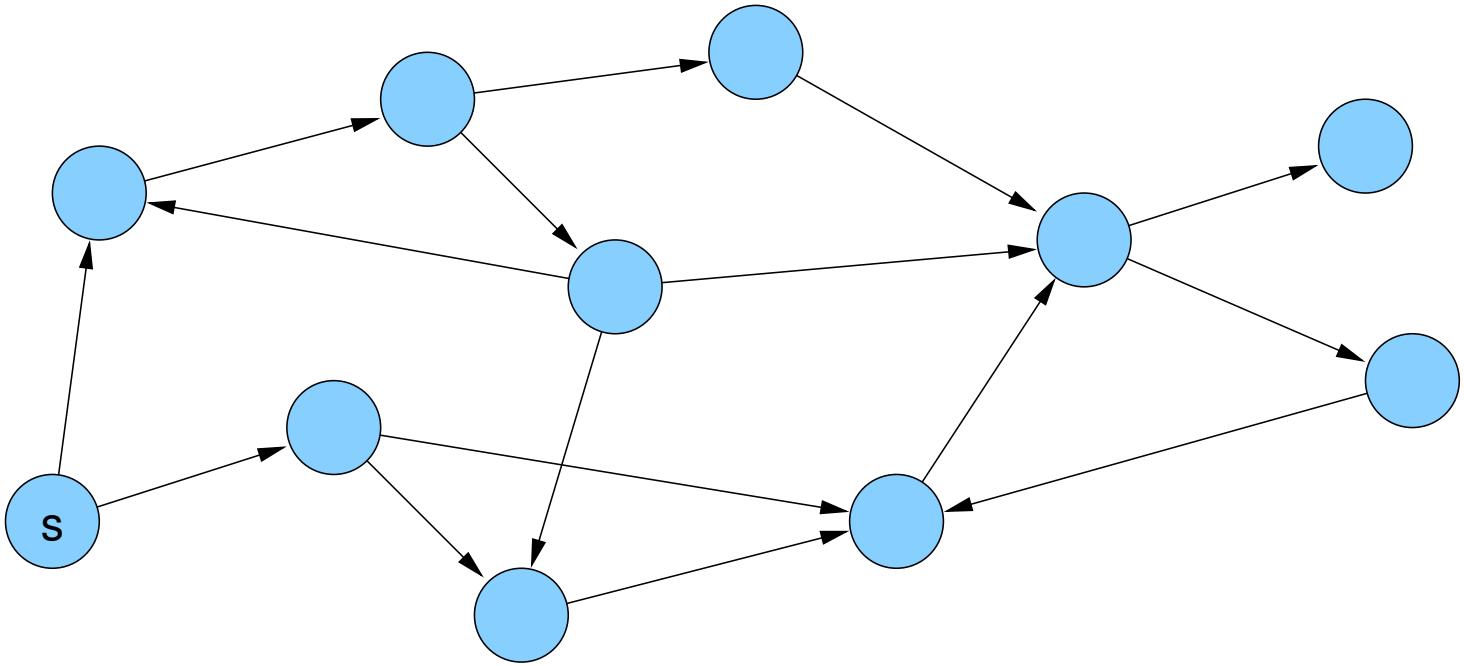
Breitensuche



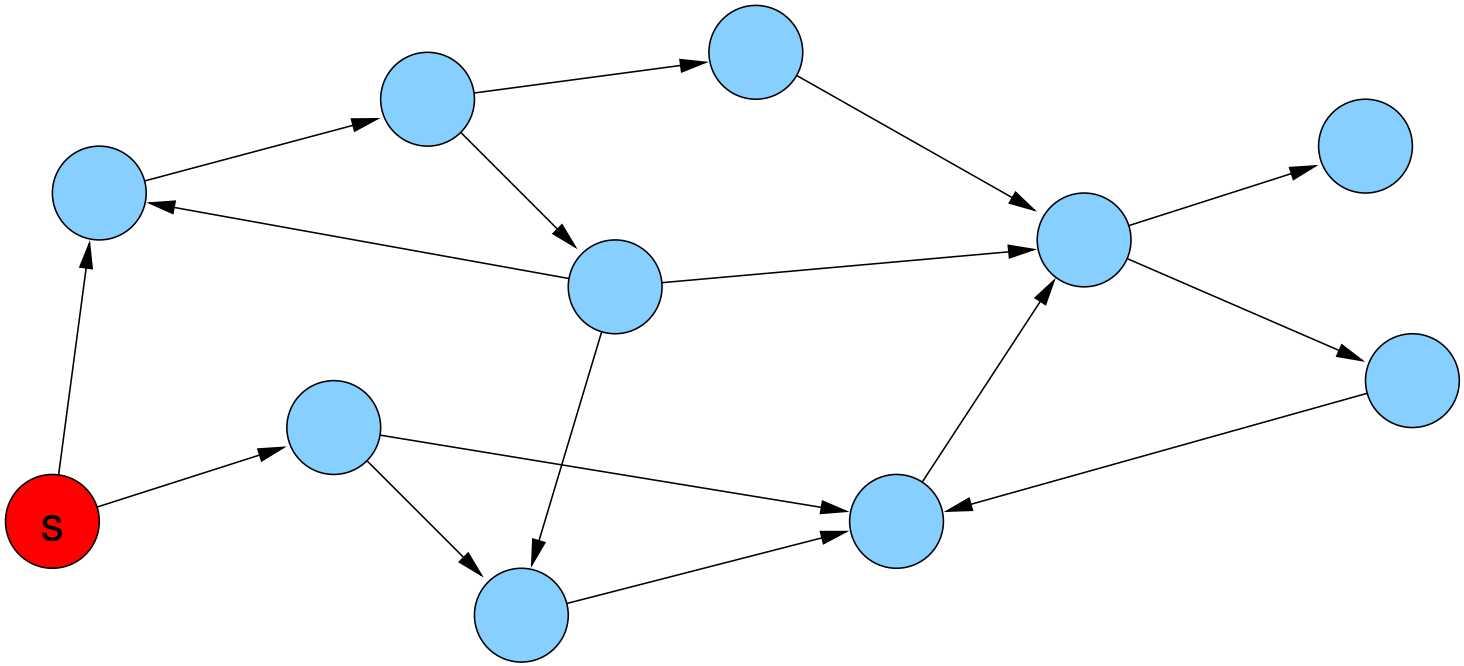
Breitensuche



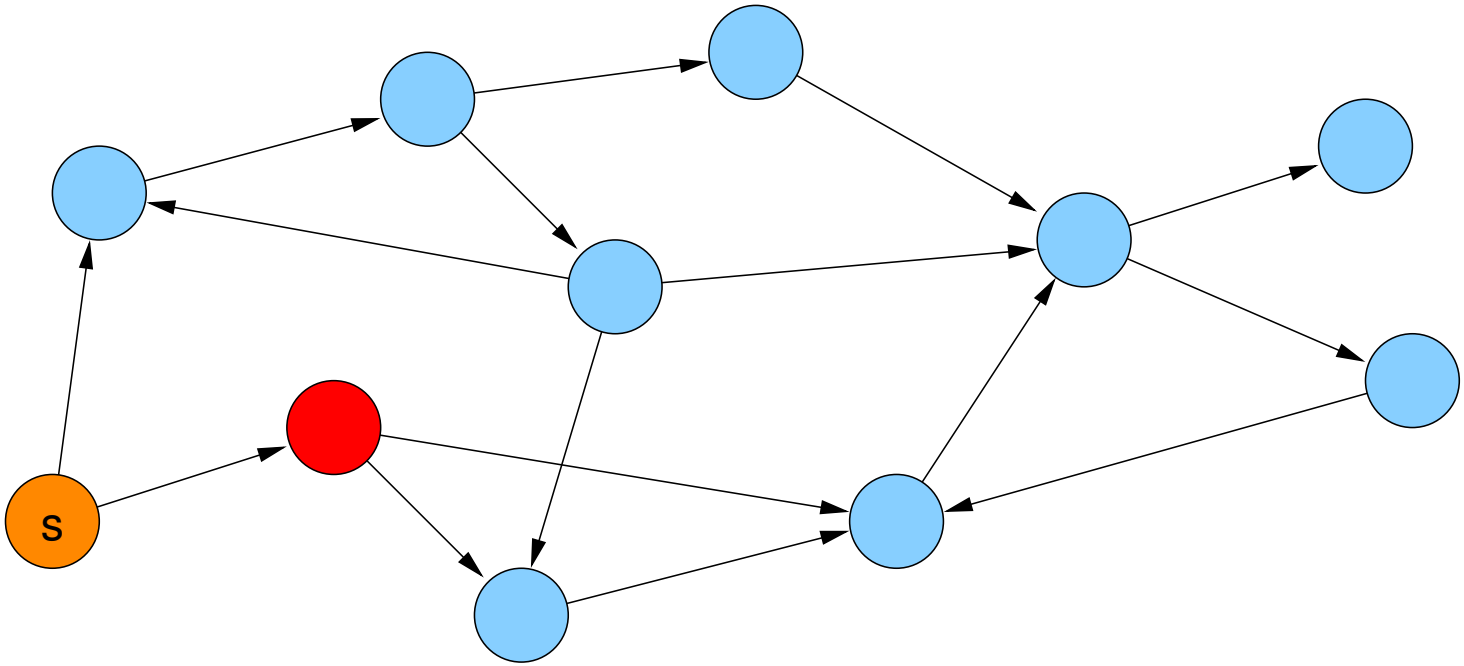
Tiefensuche



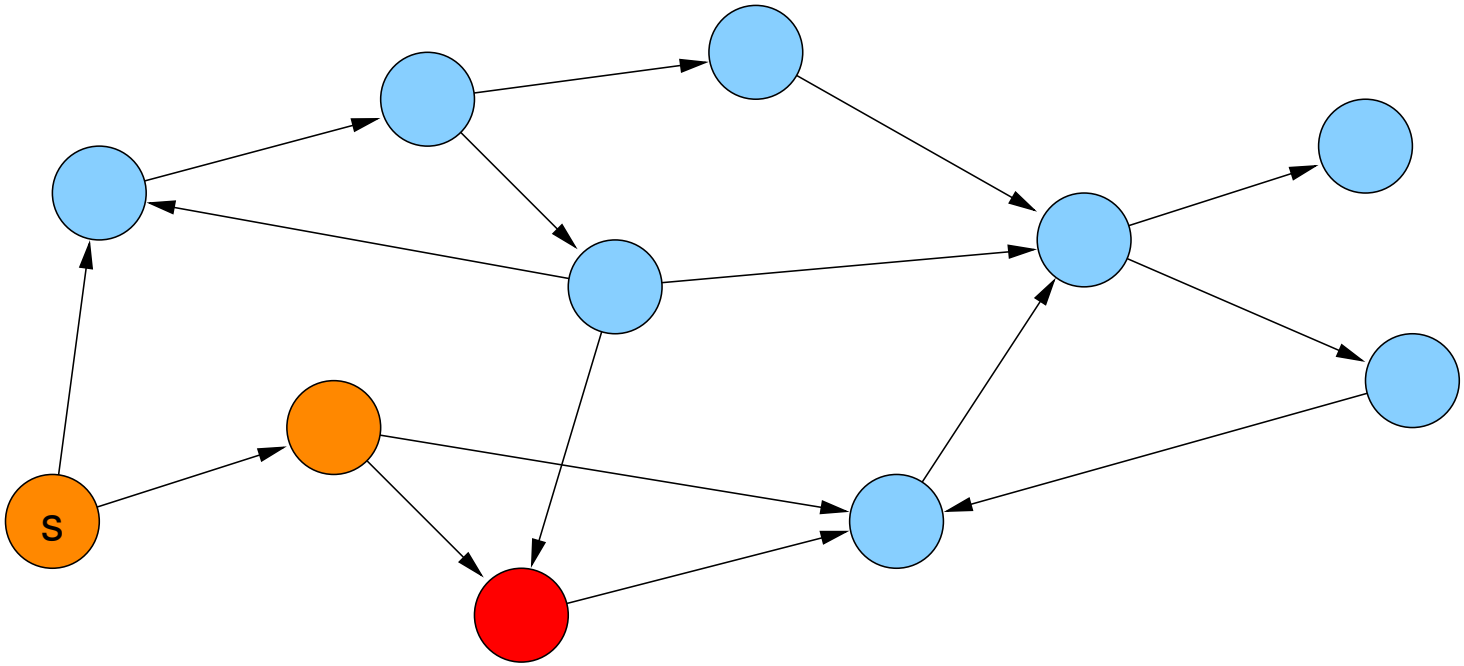
Tiefensuche



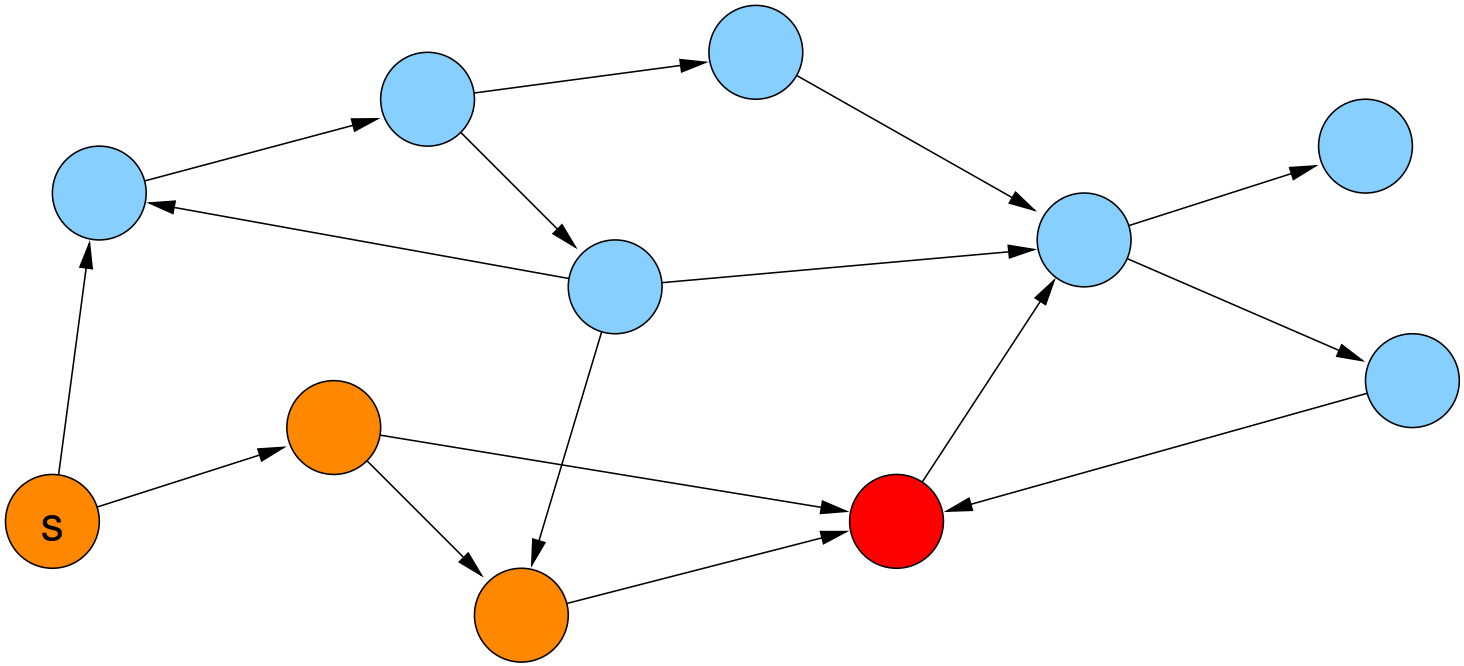
Tiefensuche



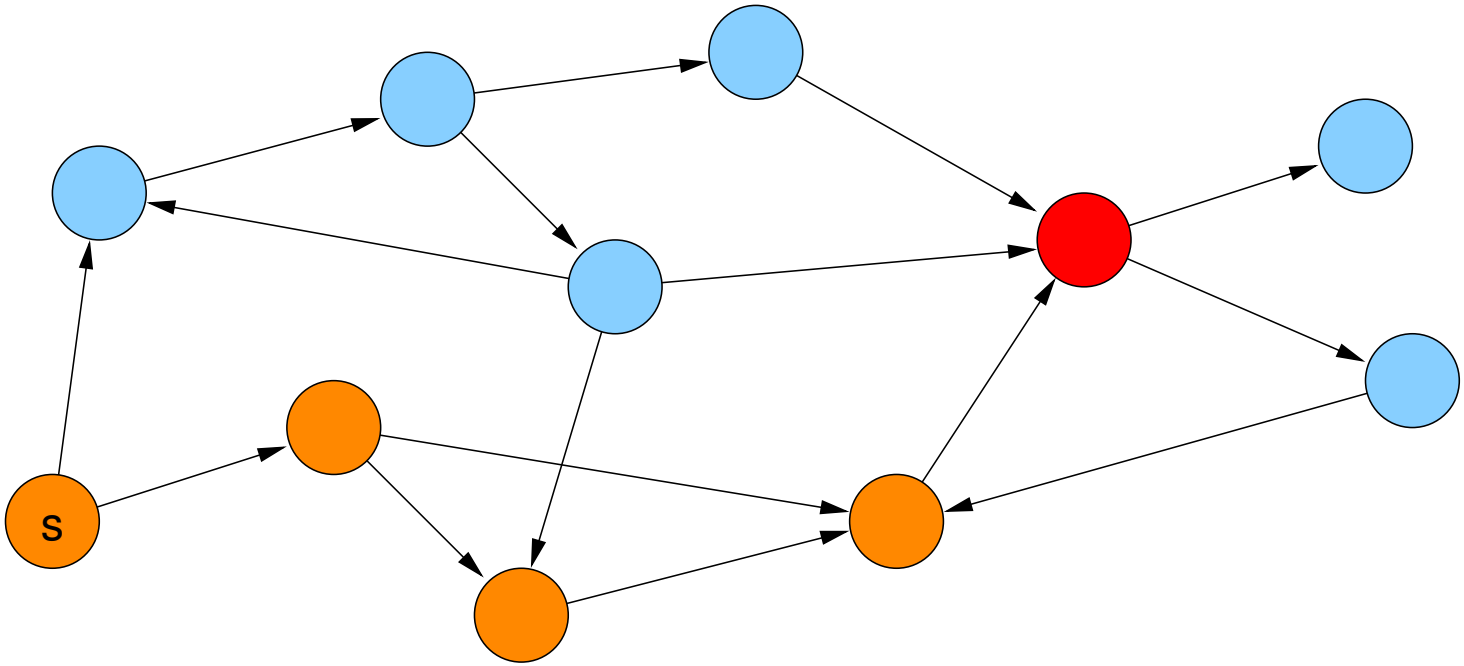
Tiefensuche



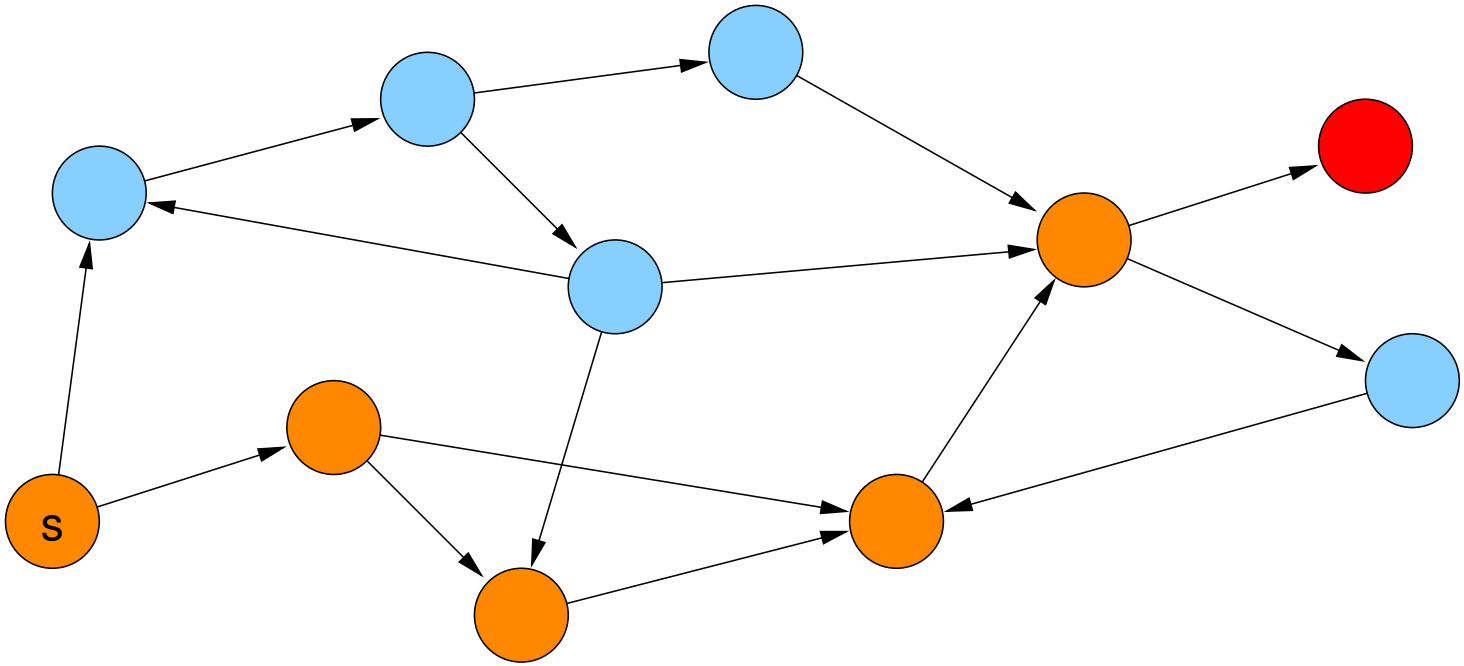
Tiefensuche



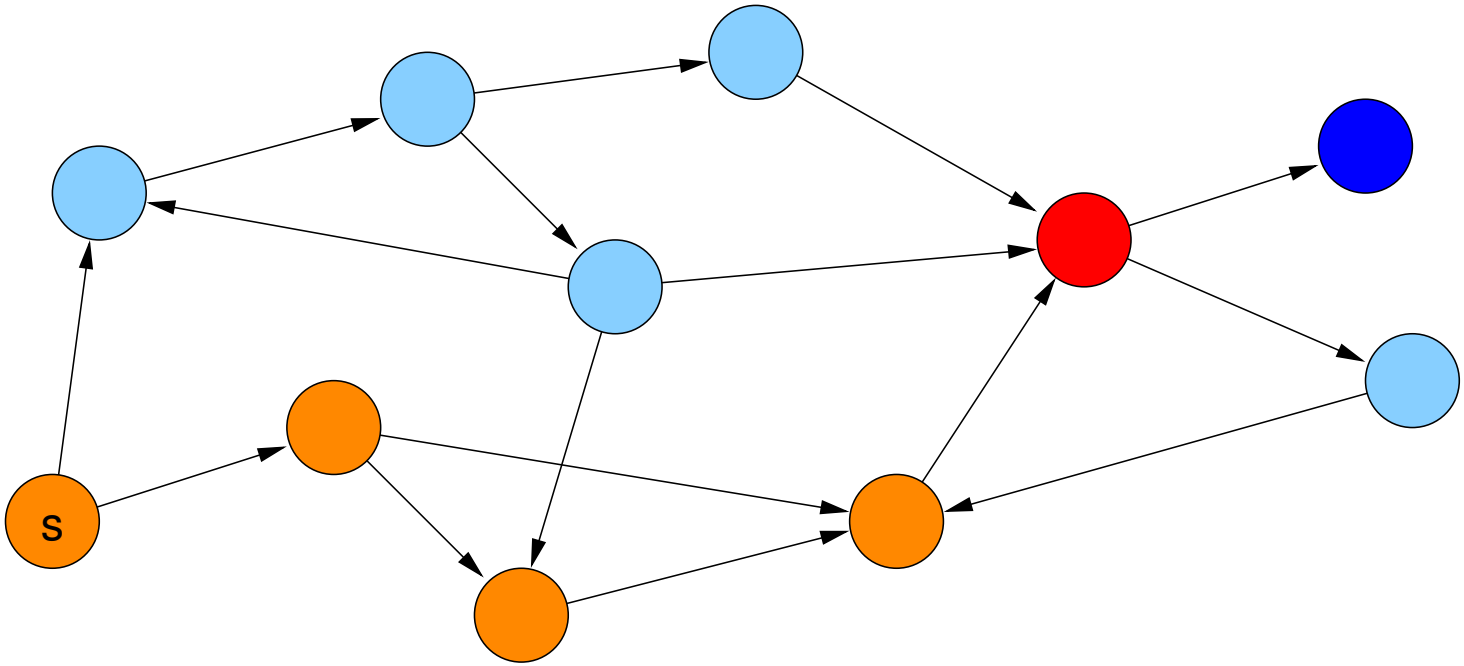
Tiefensuche



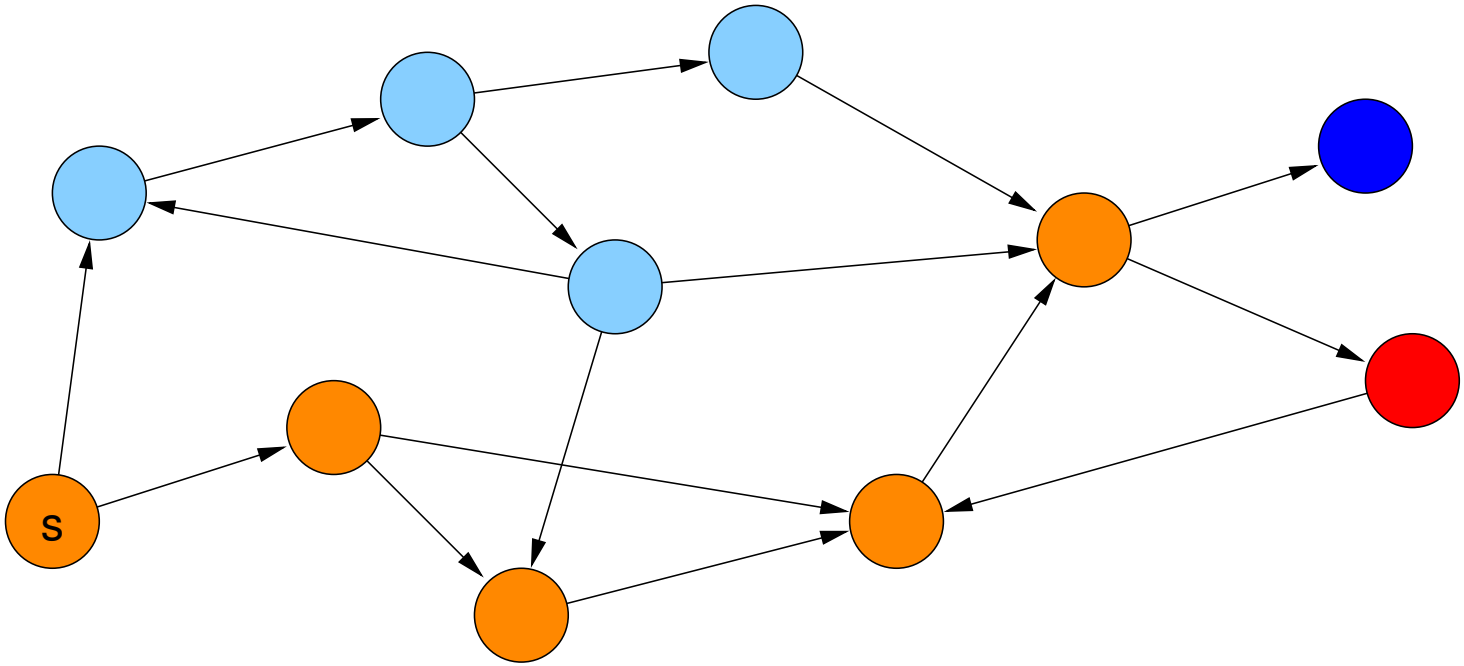
Tiefensuche



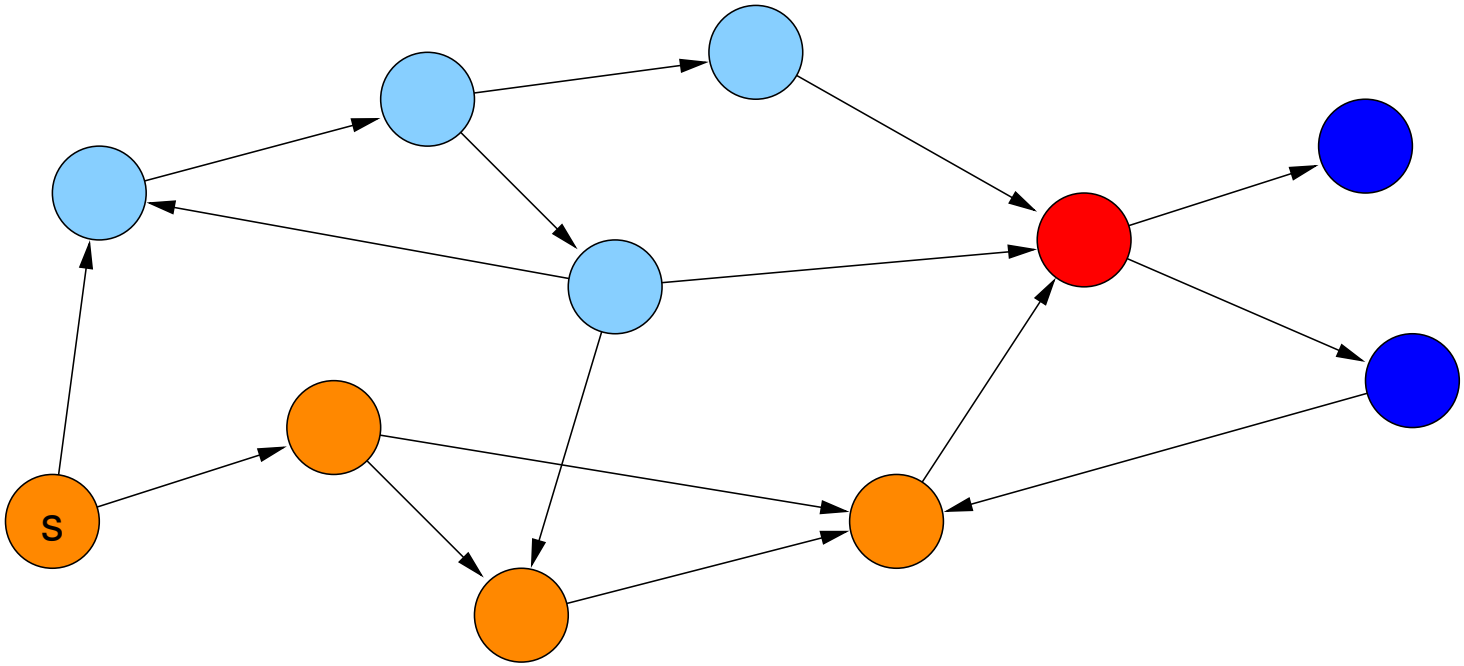
Tiefensuche



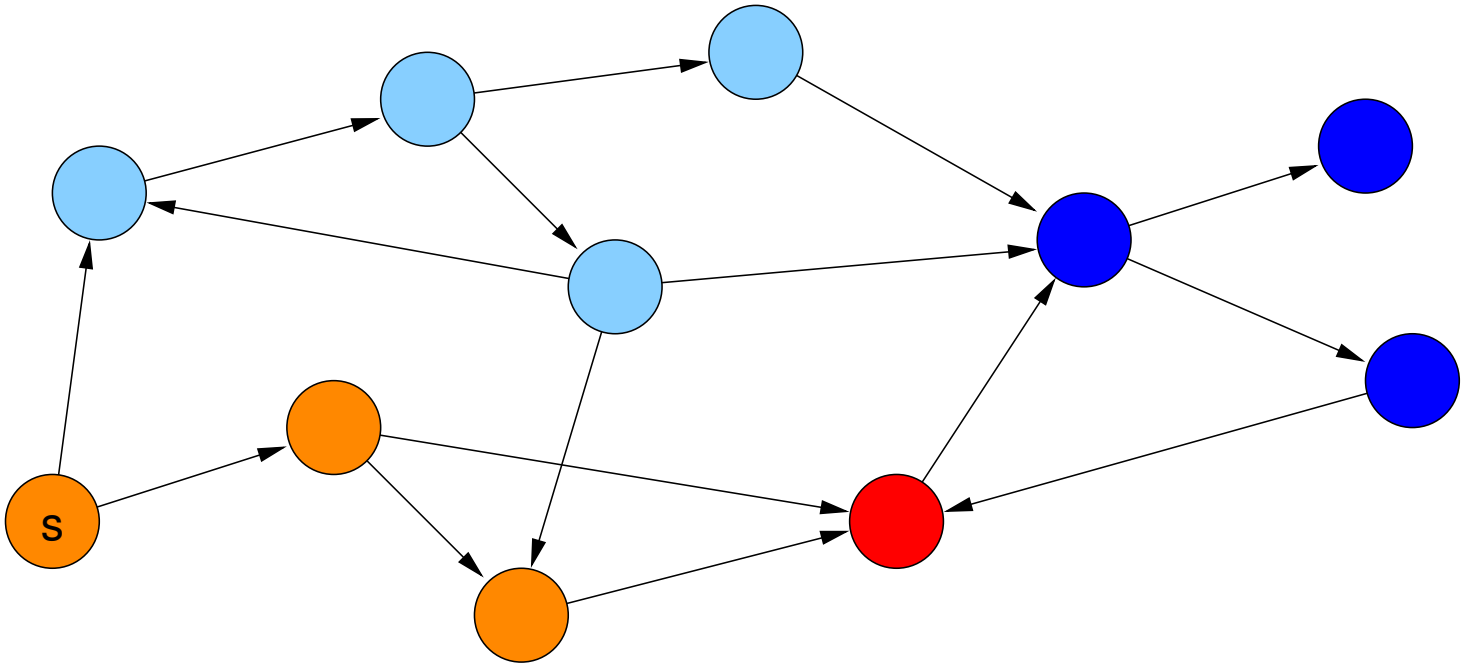
Tiefensuche



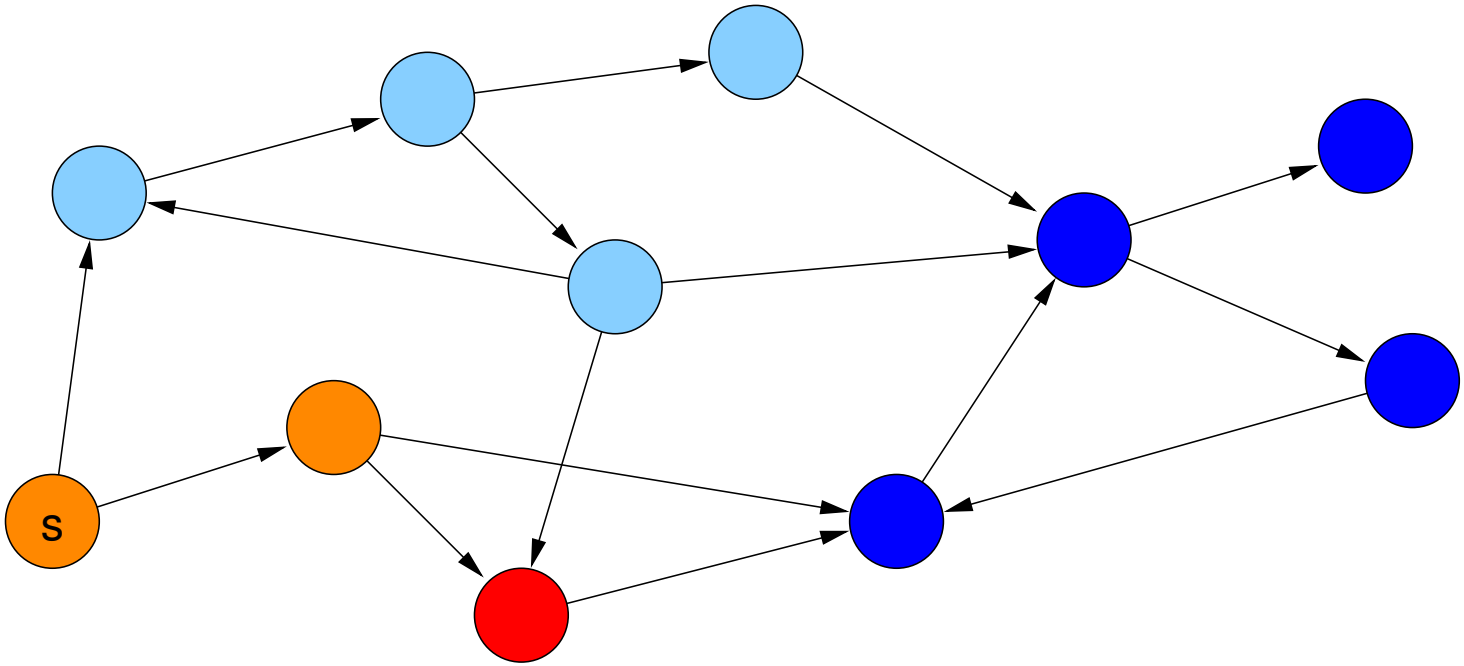
Tiefensuche



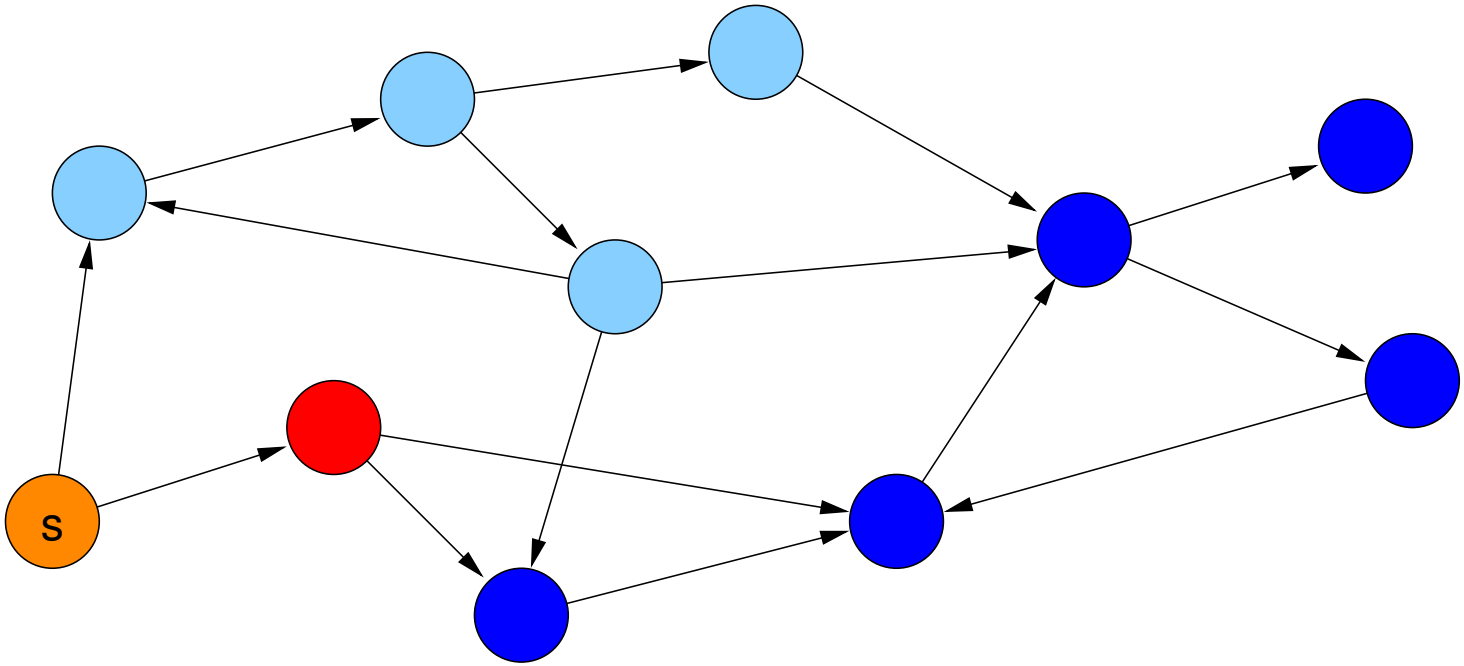
Tiefensuche



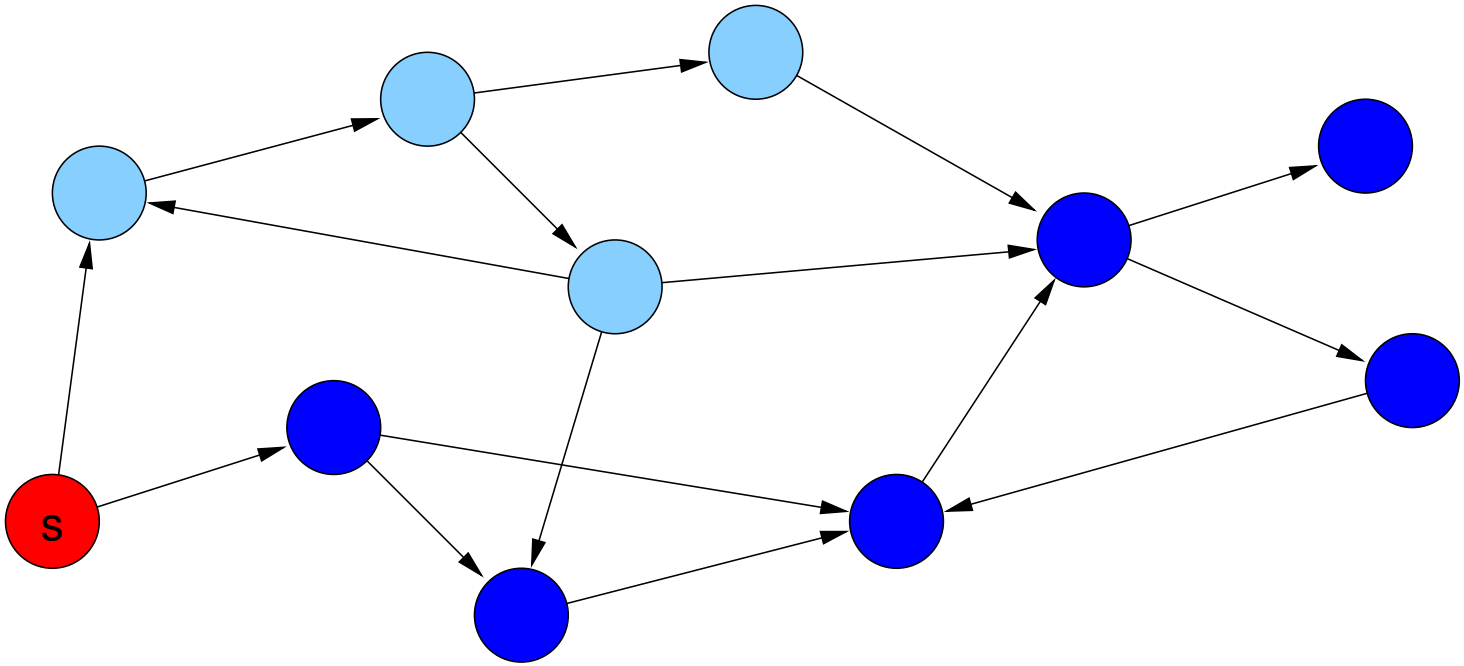
Tiefensuche



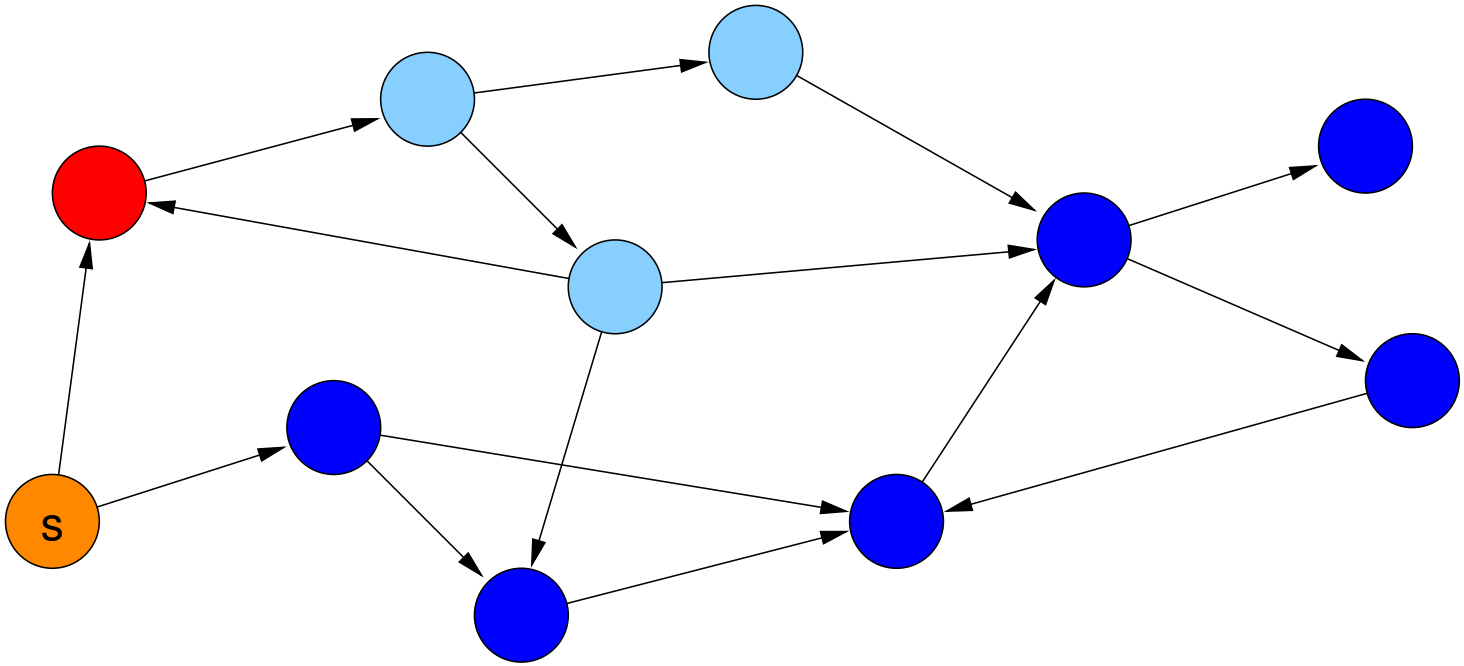
Tiefensuche



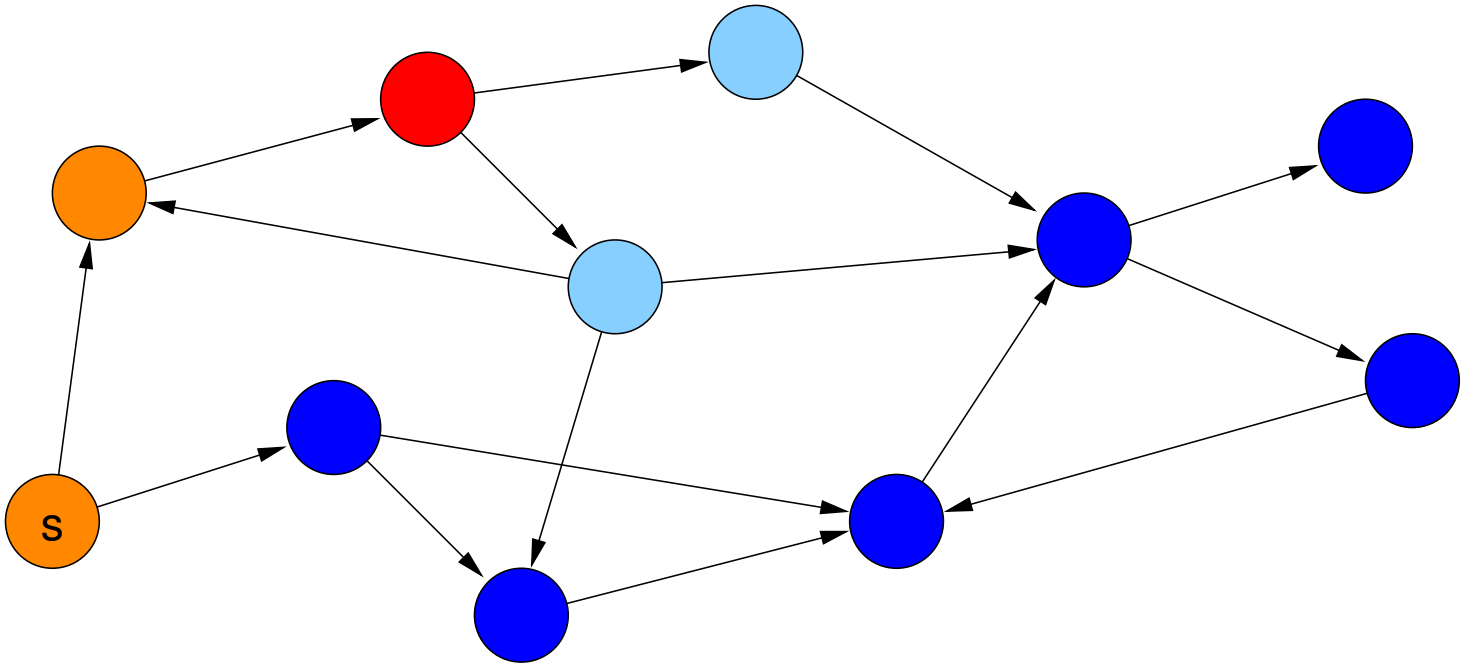
Tiefensuche



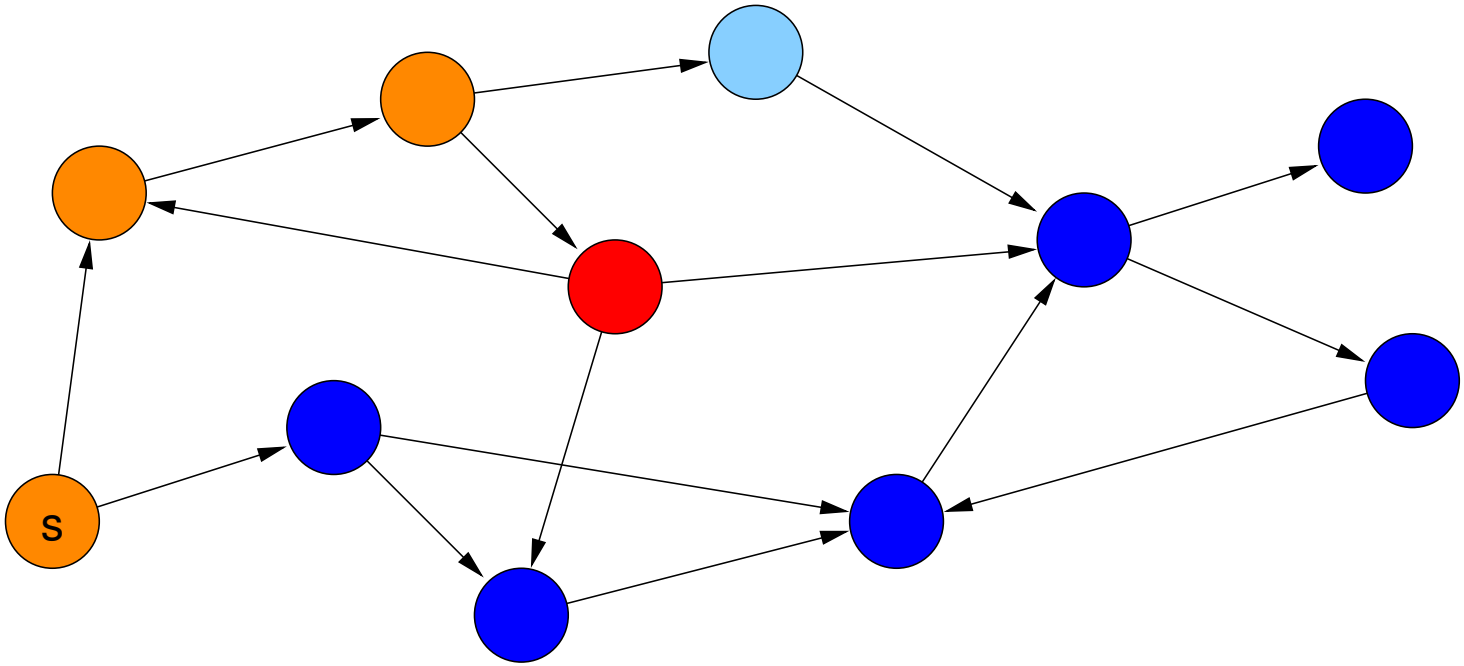
Tiefensuche



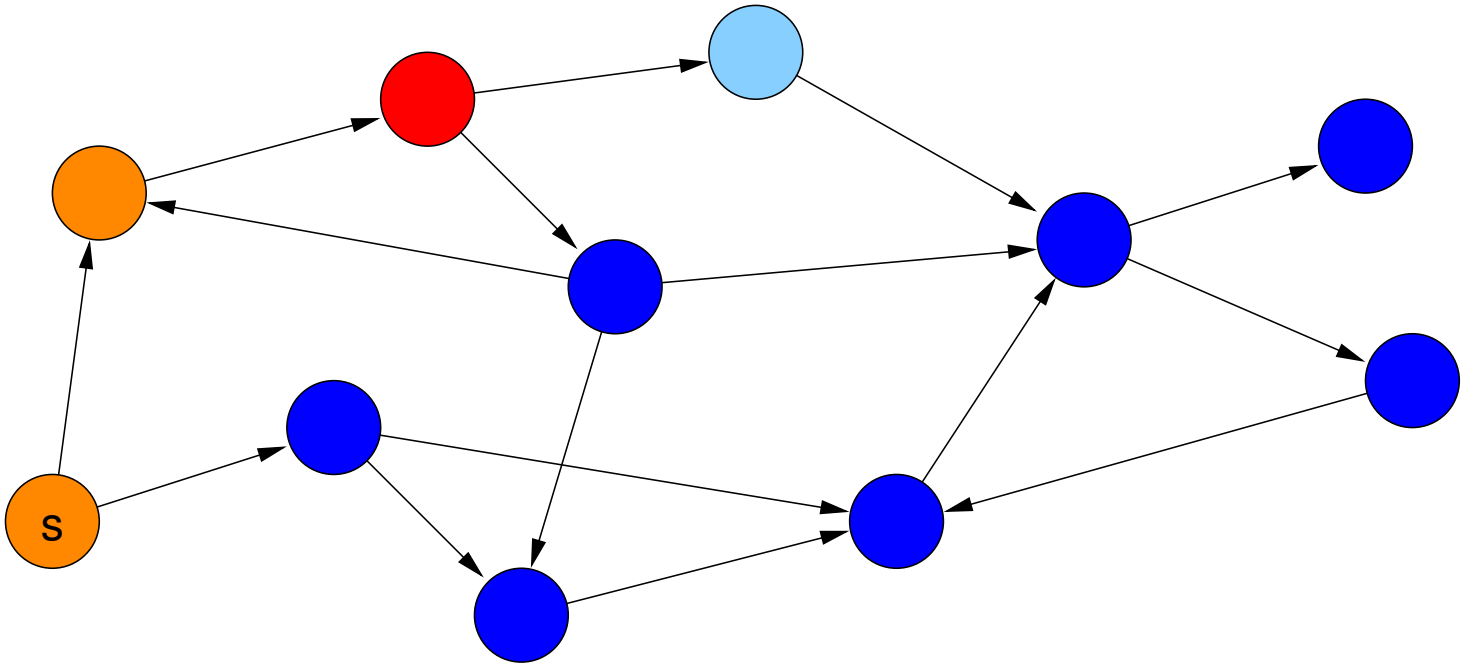
Tiefensuche



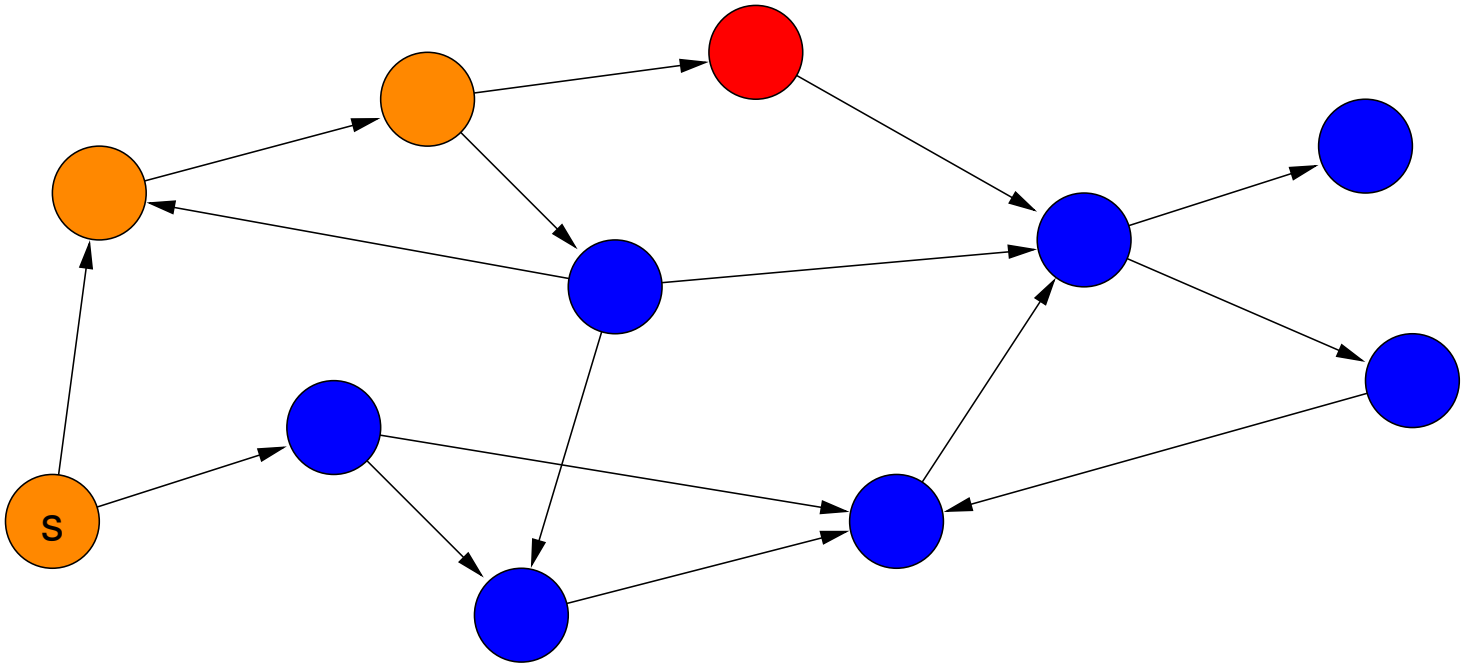
Tiefensuche



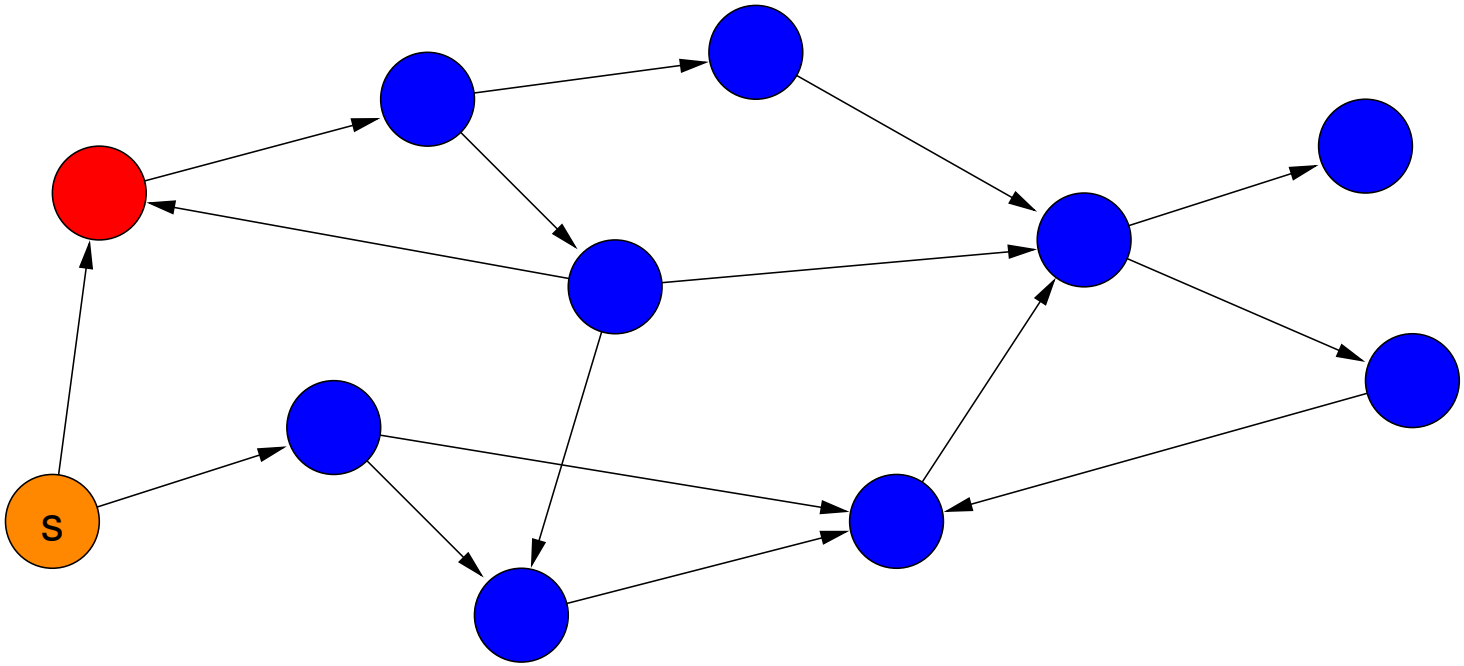
Tiefensuche



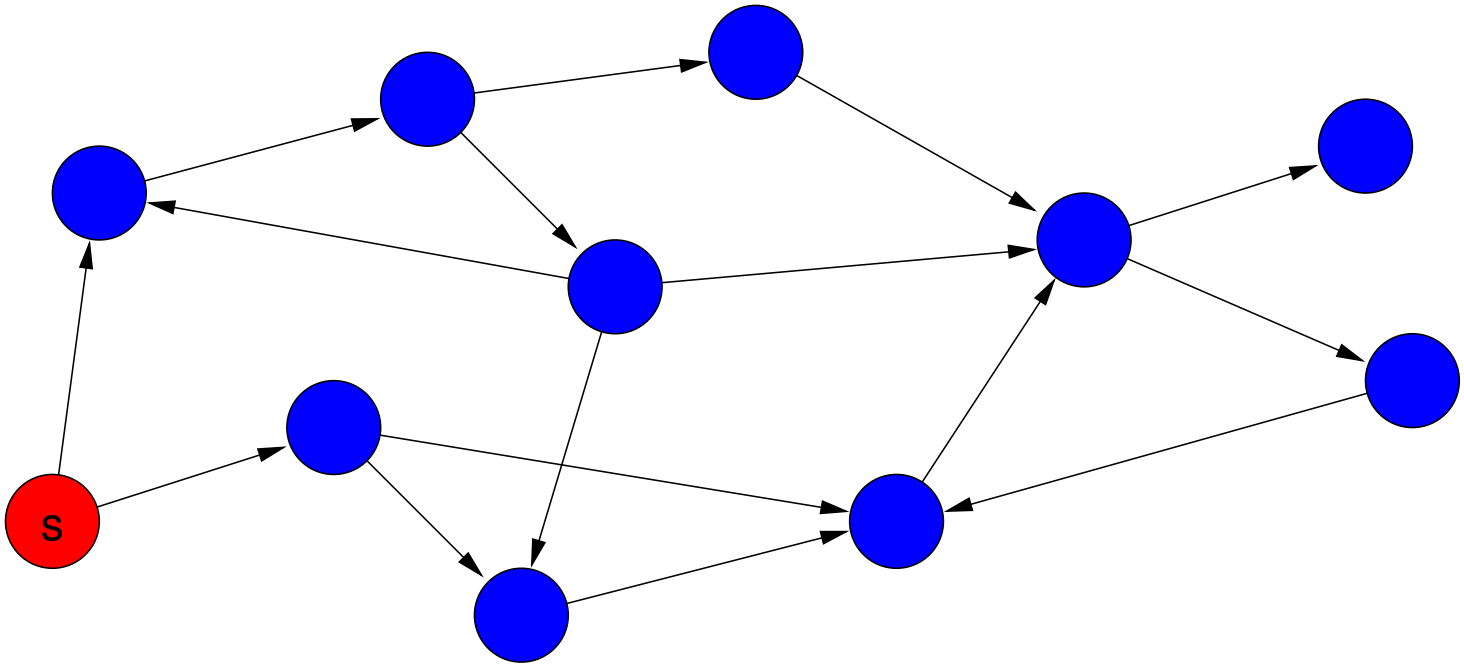
Tiefensuche



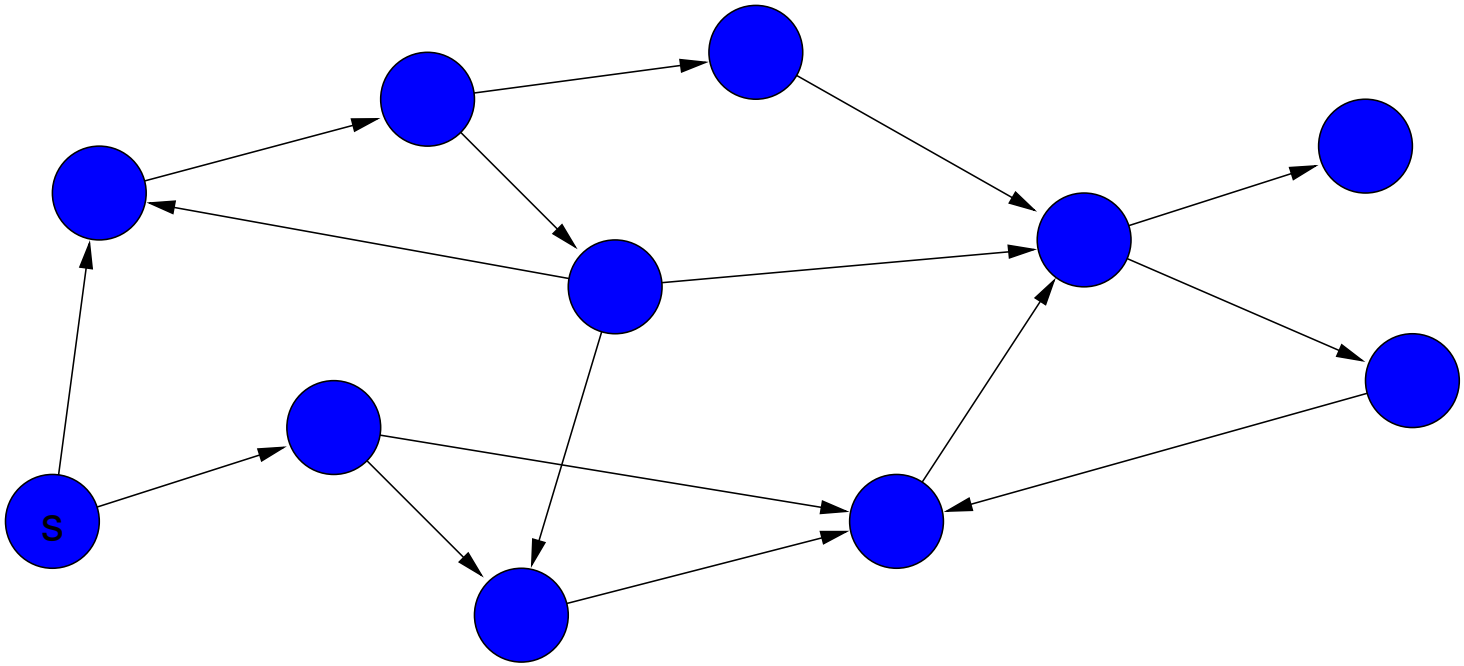
Tiefensuche



Tiefensuche

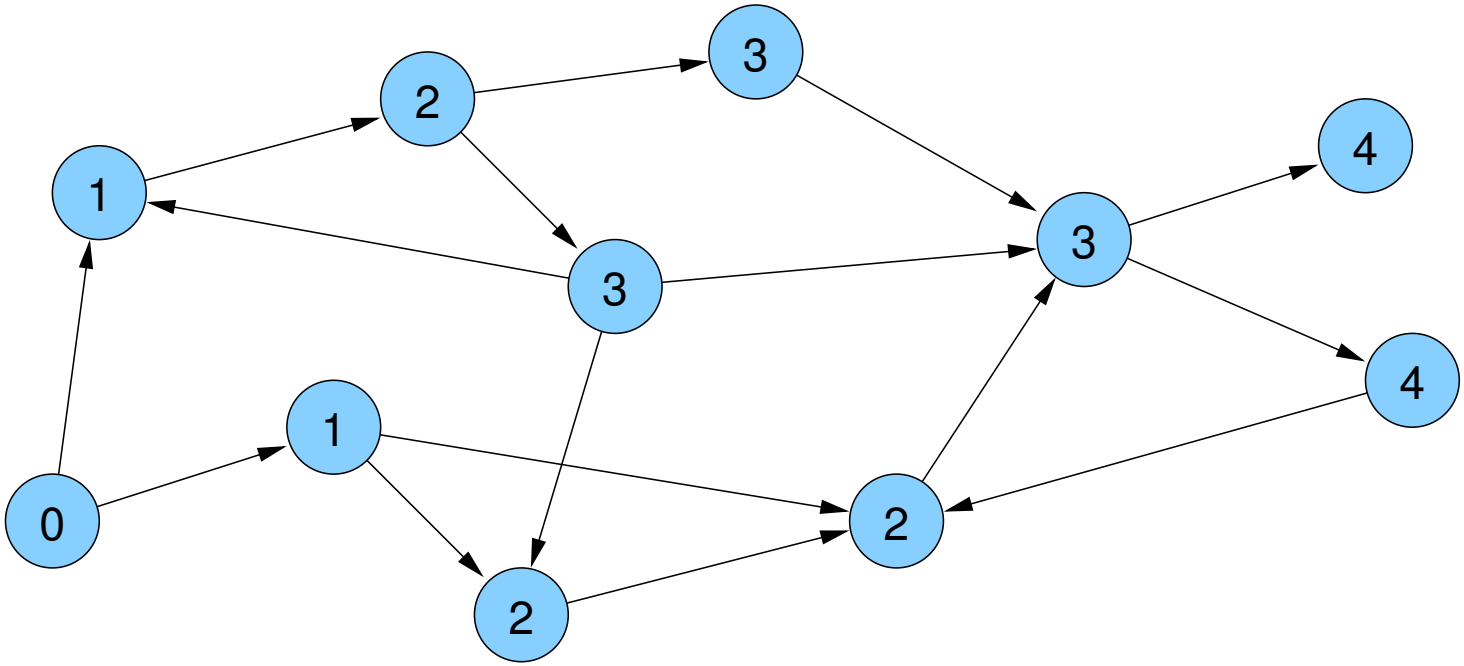


Tiefensuche



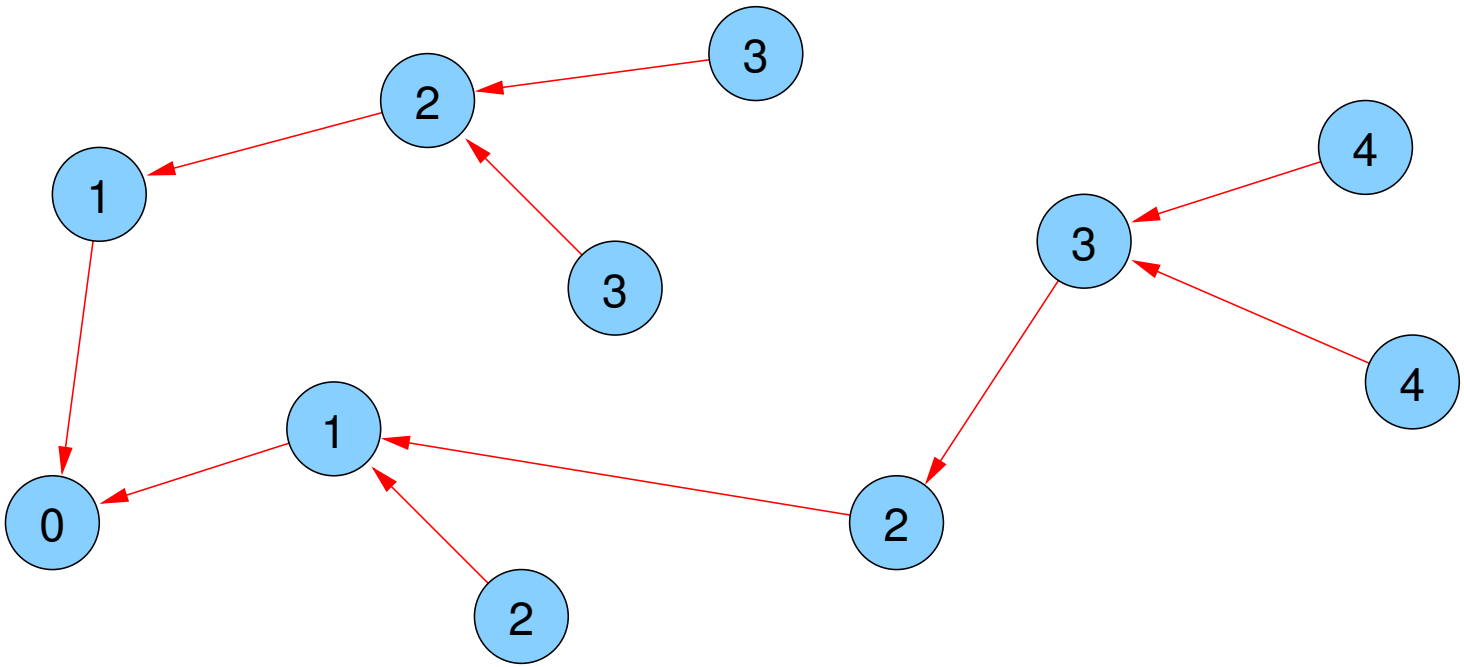
Breitensuche

- $d(v)$: Distanz von Knoten v zu s ($d(s) = 0$)



Breitensuche

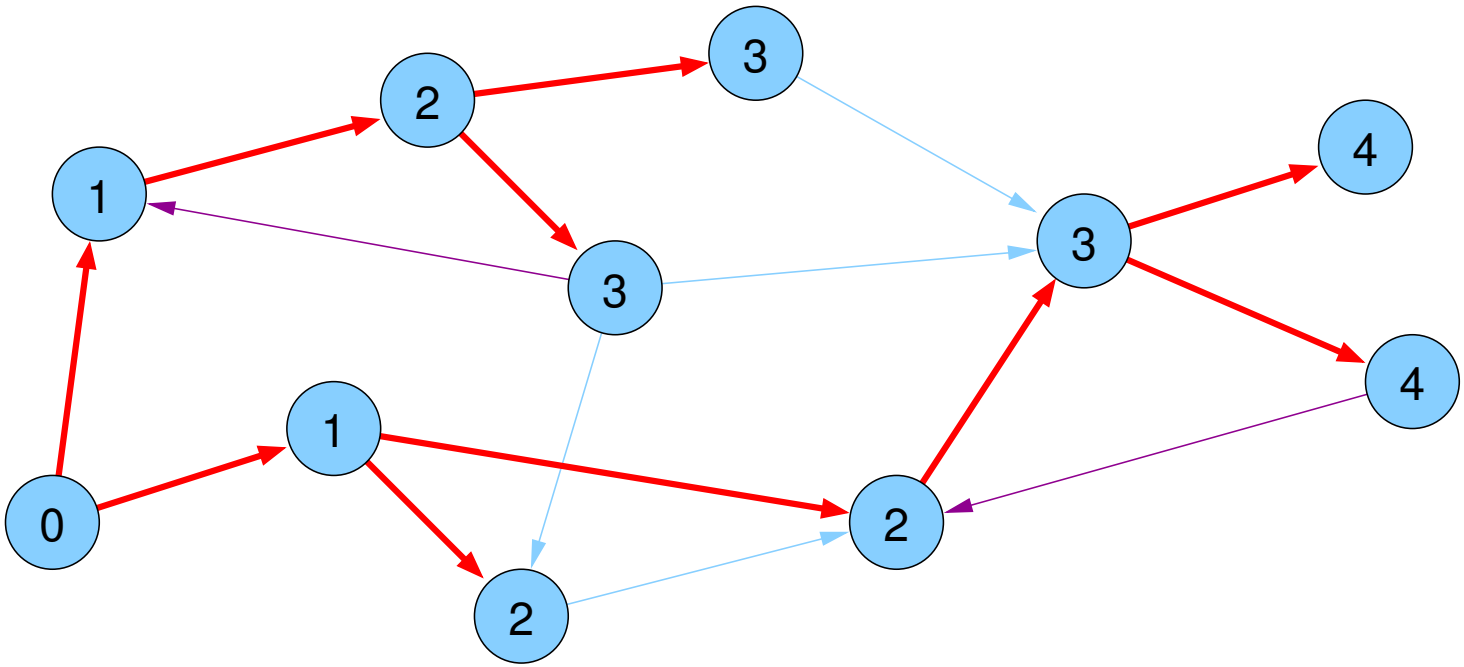
- parent(v): Knoten, von dem v entdeckt wurde
- parent wird beim ersten Besuch von v gesetzt ($\Rightarrow$ eindeutig)



Breitensuche

Kantentypen:

- Baumkanten: zum Kind
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige

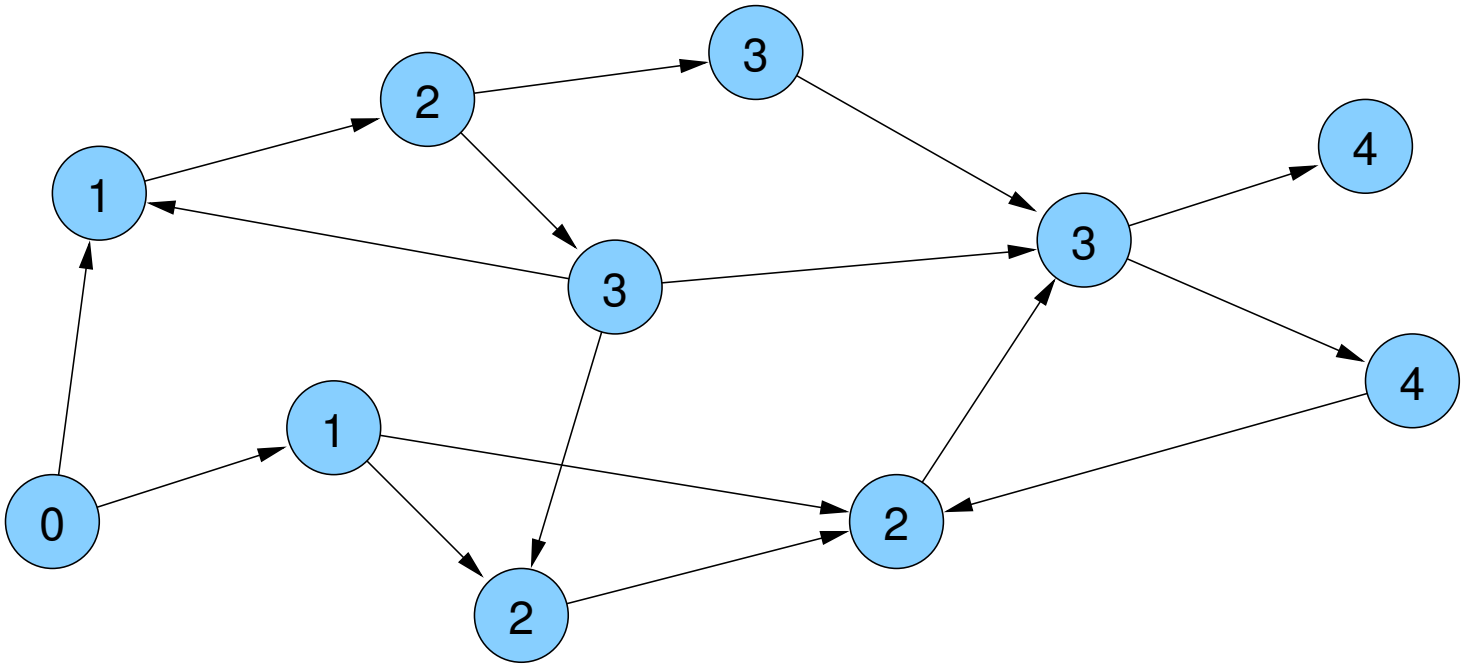


Breitensuche

```
BFS(Node s) {
    d[s] = 0;
    parent[s] = s;
    List<Node> q = ⟨s⟩;
    while ( !q.empty() ) {
        u = q.popFront();
        foreach ((u, v) ∈ E) {
            if (parent[v] == null) {
                q.pushBack(v);
                d[v] = d[u]+1;
                parent[v] = u;
            }
        }
    }
}
```

Breitensuche

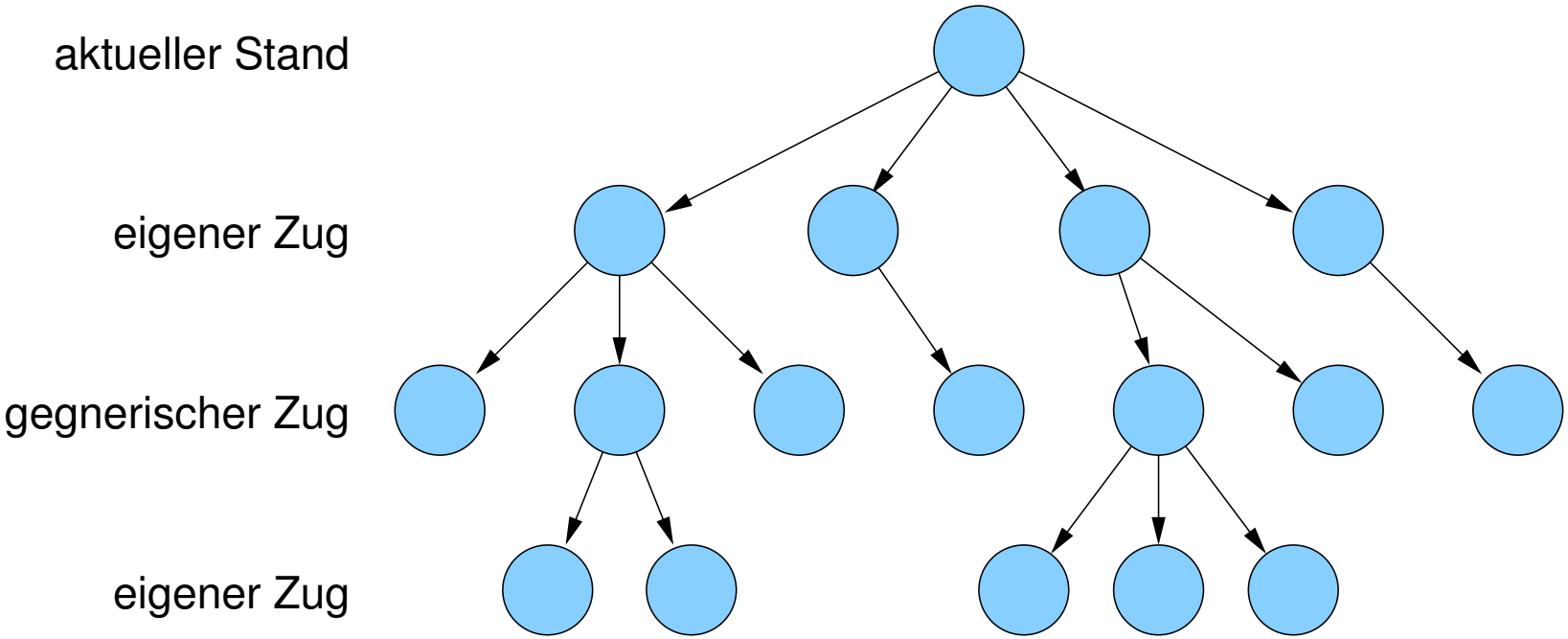
Anwendung: Single Source Shortest Path (SSSP) Problem
in ungewichteten Graphen



Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen

Exploration des Spielbaums



Problem: halte Aufwand zur Suche eines guten Zuges in Grenzen

Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen

- Standard-BFS: verwendet FIFO-Queue

  ebenenweise Erkundung

  aber: zu teuer!

- Best-First Search: verwendet Priority Queue

  (z.B. realisiert durch binären Heap)

  Priorität eines Knotens wird durch eine Güte-Heuristik des repräsentierten Spielzustands gegeben

Tiefensuche

Übergeordnete Methode (falls nicht alle Knoten erreicht werden)

```
foreach (v ∈ V)
    Setze v auf nicht markiert;

init();
foreach (s ∈ V)
    if (s nicht markiert) {
        markiere s;
        root(s);
        DFS(s,s);
    }
```

Tiefensuche

```
DFS(Node u, Node v) {
   foreach ((v, w) ∈ E)
      if (w ist markiert)
         traverseNonTreeEdge(v,w);
      else {
         traverseTreeEdge(v,w);
         markiere w;
         DFS(v,w);
      }
   backtrack(u,v);
}
```

Tiefensuche

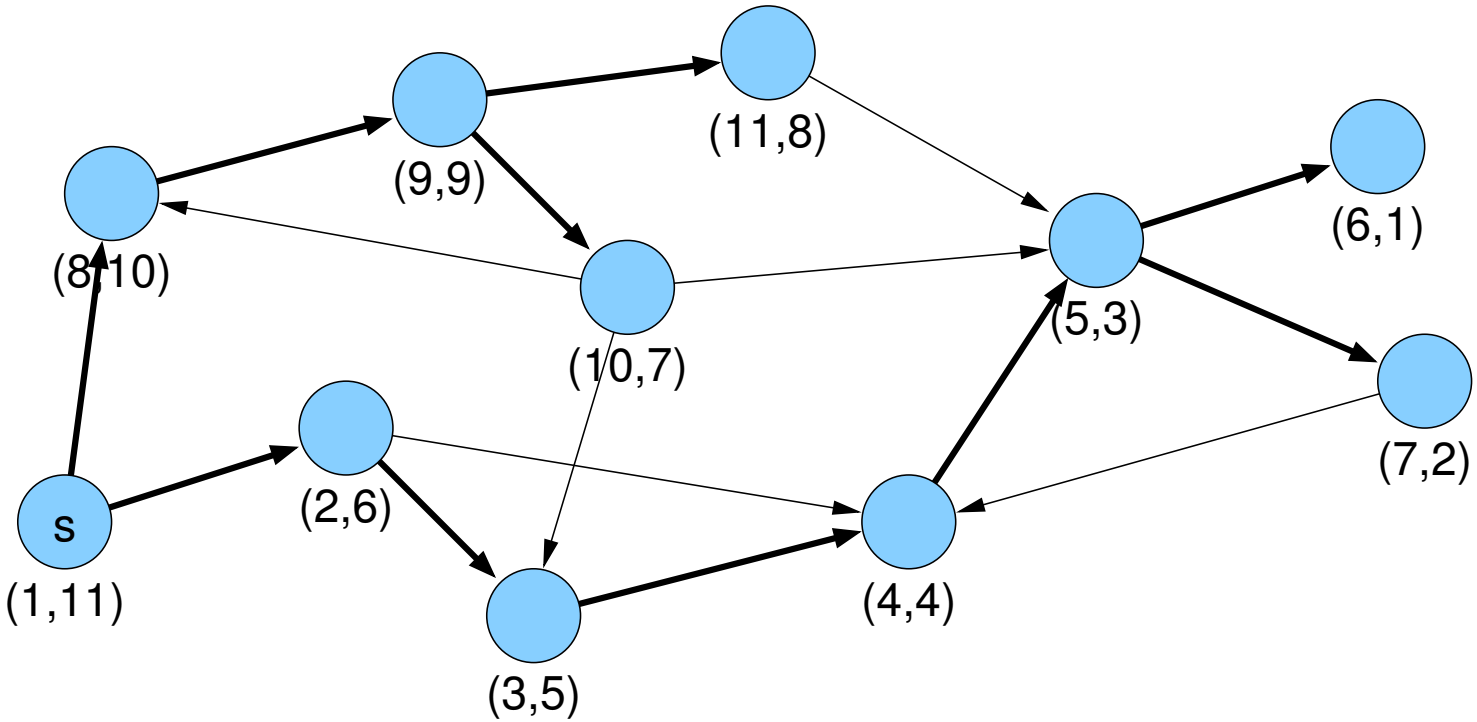
Variablen:

- int[ ] dfsNum; // Explorationsreihenfolge
- int[ ] finishNum; // Fertigstellungsreihenfolge
- int dfsCount, finishCount; // Zähler

Methoden:

- init() { dfsCount = 1; finishCount = 1; }
- root(Node s) { dfsNum[s] = dfsCount; dfsCount++; }
- traverseTreeEdge(Node v, Node w)
  { dfsNum[w] = dfsCount; dfsCount++; }
- traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v)
  { finishNum[v] = finishCount; finishCount++; }

Tiefensuche



DFS-Nummerierung

Beobachtung:

- Knoten im DFS-Rekursionsstack (aktiven Knoten) sind bezüglich dfsNum aufsteigend sortiert

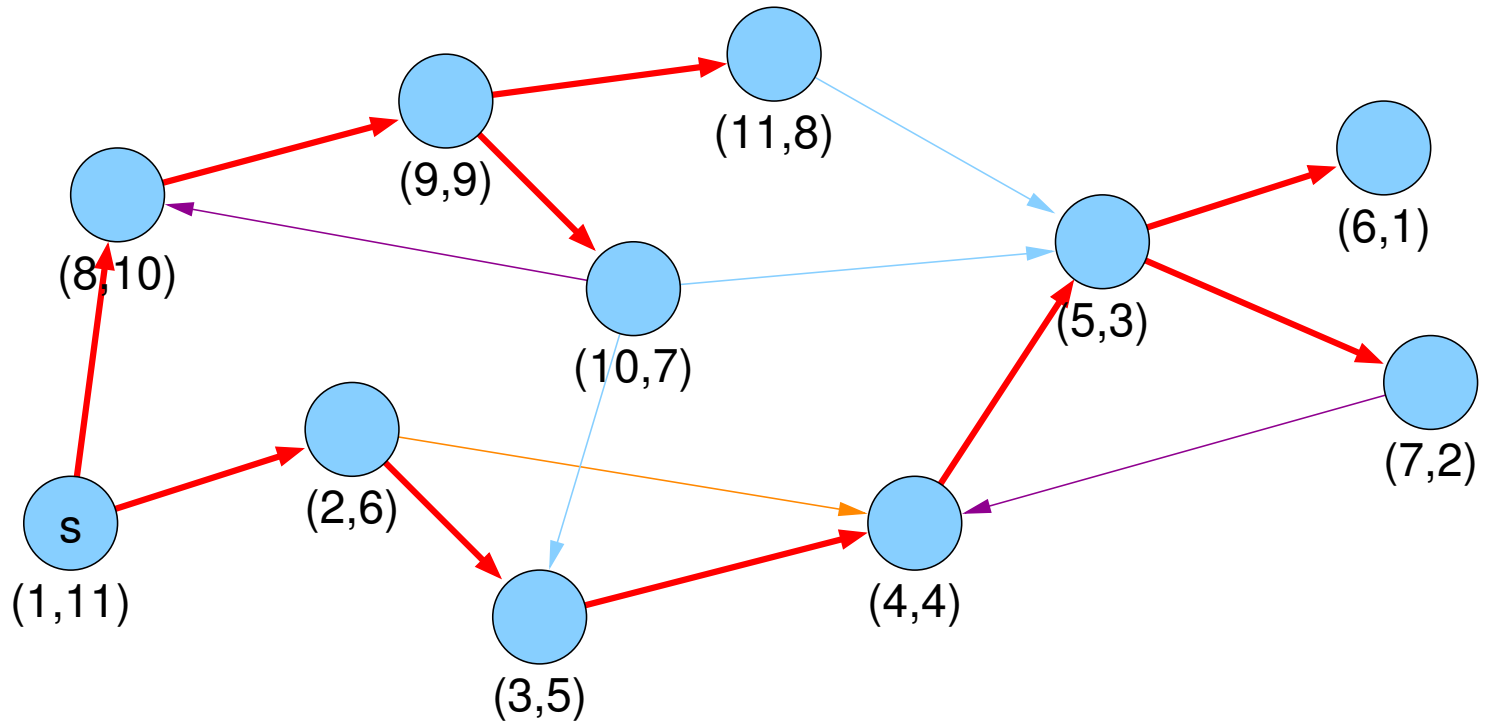
Begründung:

- dfsCount wird nach jeder Zuweisung von dfsNum inkrementiert
- neue aktive Knoten haben also immer die höchste dfsNum

DFS-Nummerierung

Kantentypen:

- Baumkanten: zum Kind
- Vorwärtskanten: zu einem Nachfahren
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige



DFS-Nummerierung

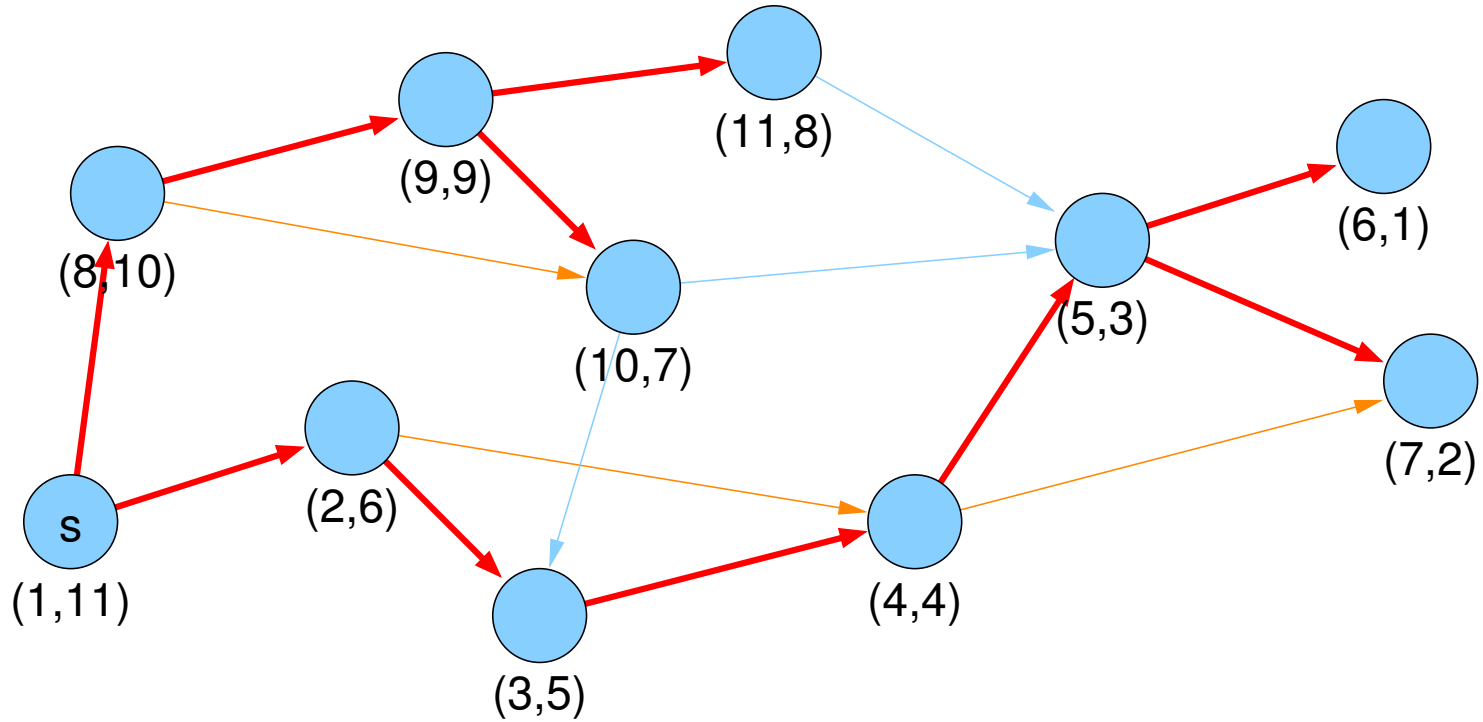
Beobachtung für Kante (v, w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DAG-Erkennung per DFS

Anwendung:

- Erkennung von azyklischen gerichteten Graphen (engl. directed acyclic graph / DAG)



- keine gerichteten Kreise

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- (2)⇒(3): Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuzkanten.
  Für alle gilt (3).
- (3)⇒(2): Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- $\neg(2) \Rightarrow \neg(1)$: Wenn Rückwärtskante (v,w) existiert, gibt es einen gerichteten Kreis ab Knoten w (und G ist kein DAG).
- $\neg(1) \Rightarrow \neg(2)$: Wenn es einen gerichteten Kreis gibt, ist mindestens eine von der DFS besuchte Kante dieses Kreises eine Rückwärtskante (Kante zu einem schon besuchten Knoten, dieser muss Vorfahr sein).

□

Zusammenhang in Graphen

Definition

Ein ungerichteter Graph heißt zusammenhängend, wenn es von jedem Knoten einen Pfad zu jedem anderen Knoten gibt.

Ein maximaler zusammenhängender induzierter Teilgraph wird als Zusammenhangskomponente bezeichnet.

Die Zusammenhangskomponenten eines ungerichteten Graphen können mit DFS oder BFS in $\mathcal{O}(n+m)$ bestimmt werden.

Knoten-Zusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt k-fach zusammenhängend (oder genauer gesagt k-*knoten*zusammenhängend), falls

- $|V| > k$ und
- für jede echte Knotenteilmenge $X \subset V$ mit $|X| < k$ der Graph $G - X$ zusammenhängend ist.

Bemerkung:

- "zusammenhängend" ist im wesentlichen gleichbedeutend mit "1-knotenzusammenhängend"

  Ausnahme: Graph mit nur einem Knoten ist zusammenhängend, aber nicht 1-zusammenhängend

Artikulationsknoten und Blöcke

Definition

Ein Knoten v eines Graphen G heißt Artikulationsknoten (engl. *cut-vertex*), wenn sich die Anzahl der Zusammenhangskomponenten von G durch das Entfernen von v erhöht.

Definition

Die Zweifachzusammenhangskomponenten eines Graphen sind die maximalen Teilgraphen, die 2-fach zusammenhängend sind.

Ein Block ist ein maximaler zusammenhängender Teilgraph, der keinen Artikulationsknoten enthält.

D.h. die Menge der Blöcke besteht aus den Zweifachzusammenhangskomponenten, den Brücken (engl. *cut edges*), sowie den isolierten Knoten.

Blöcke und DFS

Modifizierte DFS nach R. E. Tarjan:

- num[v]: DFS-Nummer von v
- low[v]: minimale Nummer num[w] eines Knotens w, der von v aus über beliebig viele (≥ 0) Baumkanten (abwärts), evt. gefolgt von einer einzigen Rückwärtskante (aufwärts) erreicht werden kann

- low[v]: Minimum von
  - num[v]
  - low[w], wobei w ein Kind von v im DFS-Baum ist (Baumkante)
  - num[w], wobei $\{v, w\}$ eine Rückwärtskante ist

Artikulationsknoten und DFS

Lemma

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

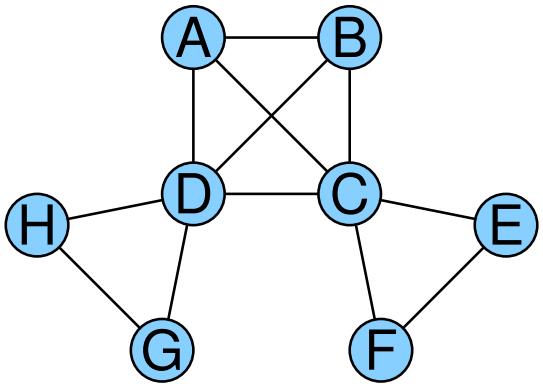
- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und es ein Kind b von a mit $\texttt{low[b]} \geq \texttt{num[a]}$ gibt.*

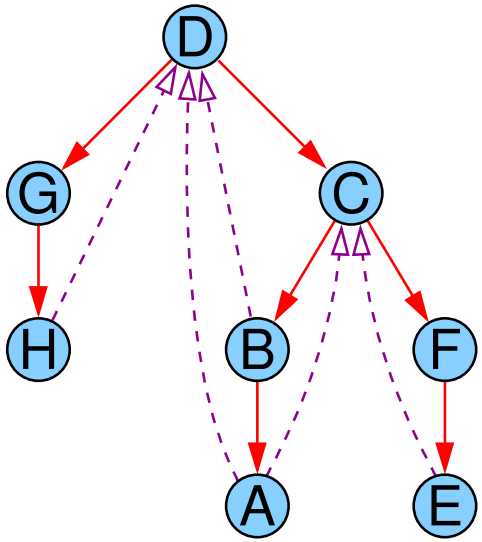
Beweisidee

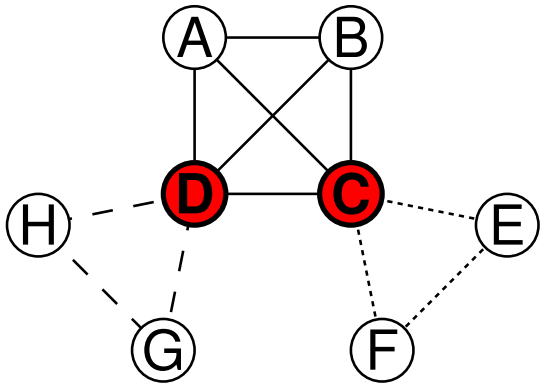
Der Algorithmus beruht auf der Tatsache, dass in Zweifach(knoten)zusammenhangskomponenten zwischen jedem Knotenpaar mindestens zwei (knoten-)disjunkte Wege existieren. Das entspricht einem Kreis.

Artikulationsknoten und Blöcke per DFS

- bei Aufruf der DFS für Knoten v wird `num[`v`]` bestimmt und `low[`v`]` mit `num[`v`]` initialisiert
- nach Besuch eines Nachbarknotens w: Update von `low[`v`]` durch Vergleich mit
  - `low[`w`]` nach Rückkehr vom rekursiven Aufruf, falls (v, w) eine Baumkante war
  - `num[`w`]`, falls (v, w) eine Rückwärtskante war

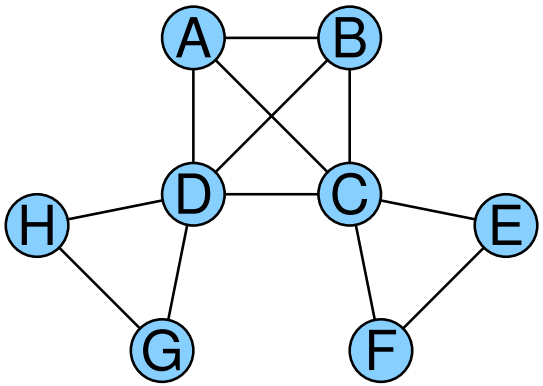


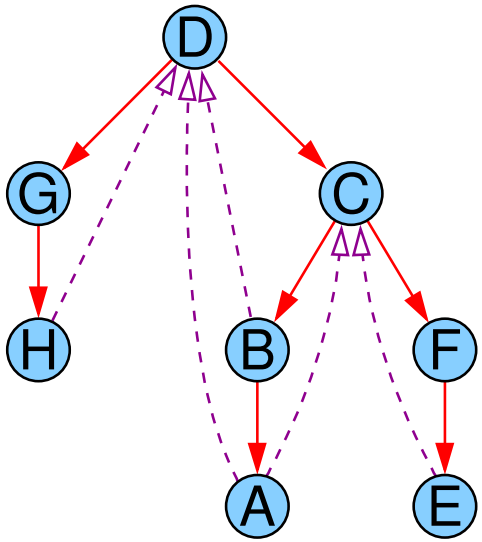


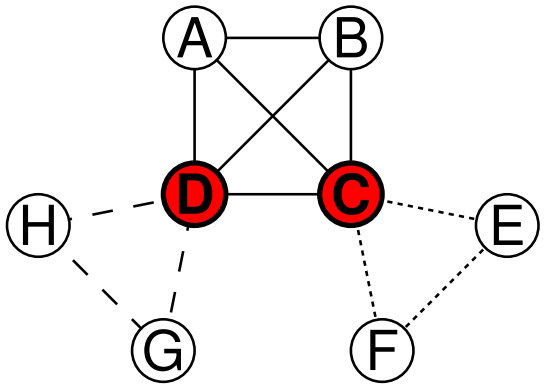


Artikulationsknoten und Blöcke per DFS

- Kanten werden auf einem anfangs leeren Stack gesammelt
- Rückwärtskanten kommen direkt auf den Stack (ohne rek. Aufruf)
- Baumkanten kommen vor dem rekursiven Aufruf auf den Stack
- nach Rückkehr von einem rekursiven Aufruf werden im Fall $\texttt{low}[w] \geq \texttt{num}[v]$ die obersten Kanten vom Stack bis einschließlich der Baumkante $\{v, w\}$ entfernt und bilden den nächsten Block







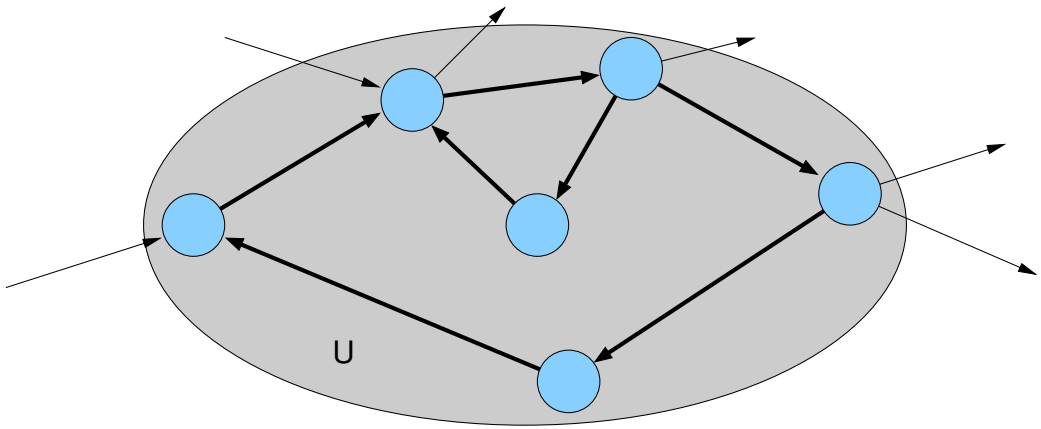
Starke Zusammenhangskomponenten

Definition

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

Knotenteilmenge $U \subseteq V$ heißt stark zusammenhängend genau dann, wenn für alle $u, v \in U$ ein gerichteter Pfad von u nach v in G existiert.

Für Knotenteilmenge $U \subseteq V$ heißt der induzierte Teilgraph $G[U]$ starke Zusammenhangskomponente von G, wenn U stark zusammenhängend und (inklusions-)maximal ist.



Starke Zusammenhangskomponenten

Beobachtungen:

- Knoten $x, y \in V$ sind stark zusammenhängend, falls beide Knoten auf einem gemeinsamen gerichteten Kreis liegen (oder $x = y$).

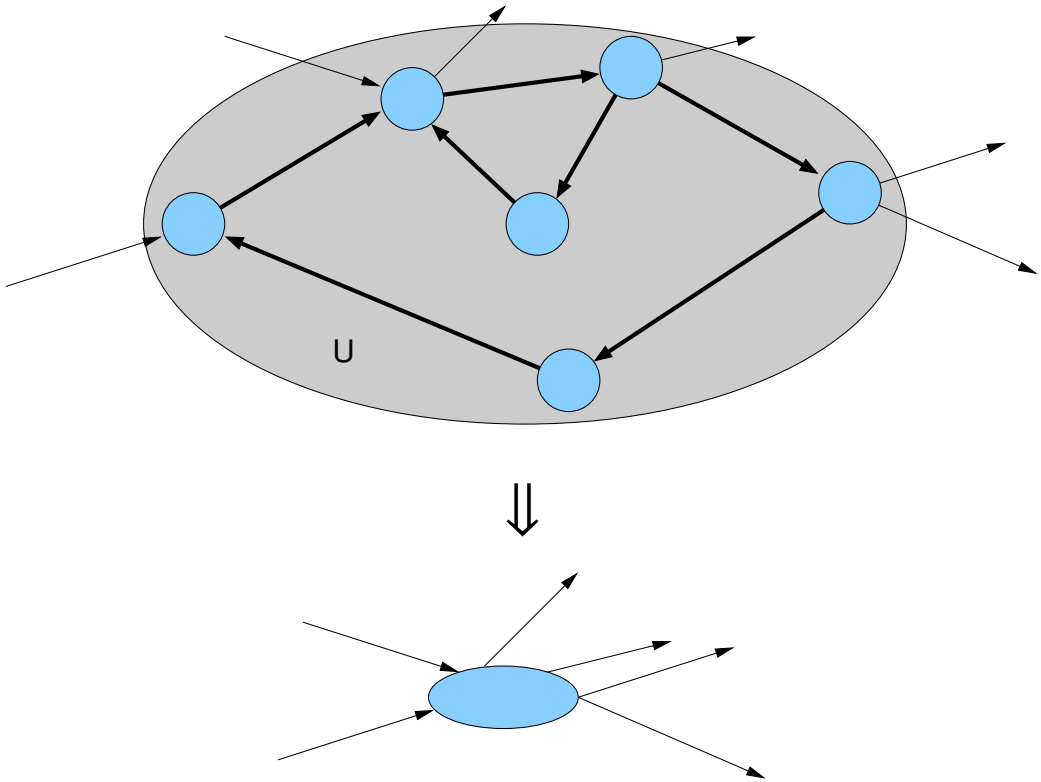
- Die starken Zusammenhangskomponenten bilden eine Partition der Knotenmenge.

  (im Gegensatz zu 2-Zhk. bei ungerichteten Graphen, wo nur die Kantenmenge partitioniert wird, sich aber zwei verschiedene 2-Zhk. in einem Knoten überlappen können)

Starke Zusammenhangskomponenten

Beobachtungen:

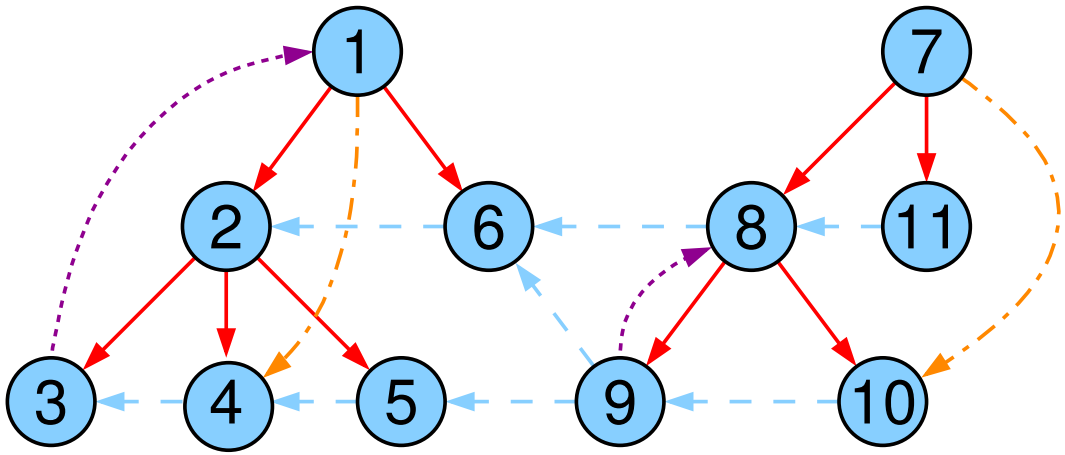
- Schrumpft man alle starken Zusammenhangskomponenten zu einzelnen (Super-)Knoten, ergibt sich ein DAG.



Starke Zhk. und DFS / Variante 1

Modifizierte DFS nach R. E. Tarjan

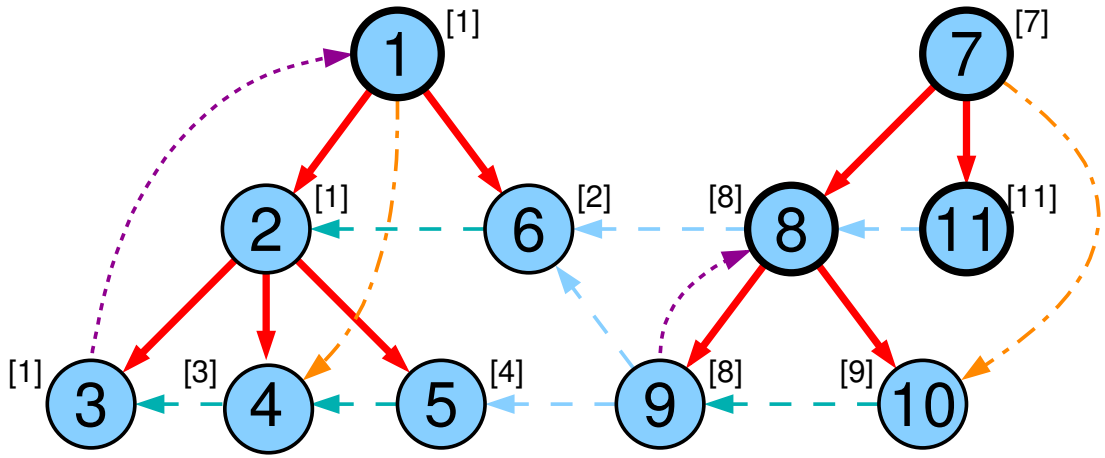
- `num[v]`: DFS-Nummer von v
- In einer starken Zhk. heißt der Knoten mit kleinster DFS-Nummer Wurzel der starken Zhk.
- `low[v]`: minimales `num[w]` eines Knotens w, der von v aus über beliebig viele (≥ 0) Baumkanten abwärts, evt. gefolgt von einer einzigen Rückwärtskante oder einer Querkante zu einer ZHK, deren Wurzel echter Vorfahre von v ist, erreicht werden kann



Starke Zhk. und DFS / Variante 1

Modifizierte DFS nach R. E. Tarjan

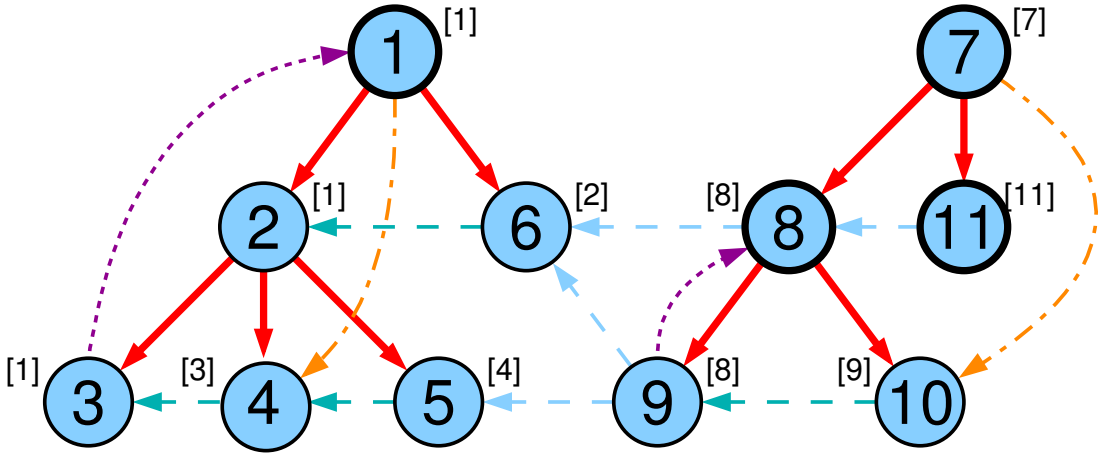
- `low[v]`: Minimum von
  - `num[v]`
  - `low[w]`, wobei w ein Kind von v im DFS-Baum ist (Baumkante)
  - `num[w]`, wobei $\{v, w\}$ eine Rückwärtskante ist
  - `num[w]`, wobei $\{v, w\}$ eine Querkante ist und die Wurzel der starken Zusammenhangskomponente von w ist Vorfahre von v

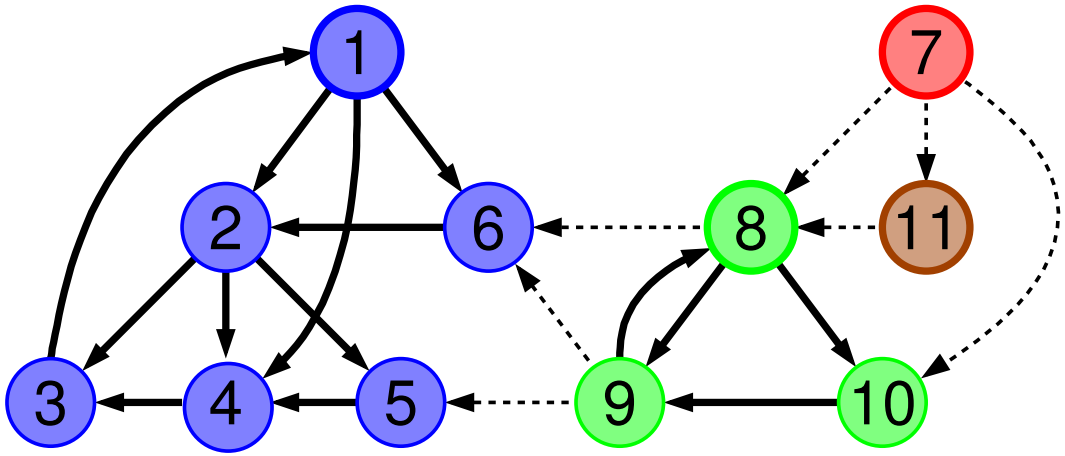


Starke Zhk. und DFS / Variante 1

Modifizierte DFS nach R. E. Tarjan

- Knoten v ist genau dann Wurzel einer starken Zusammenhangskomponente, wenn $\texttt{num}[v] = \texttt{low}[v]$





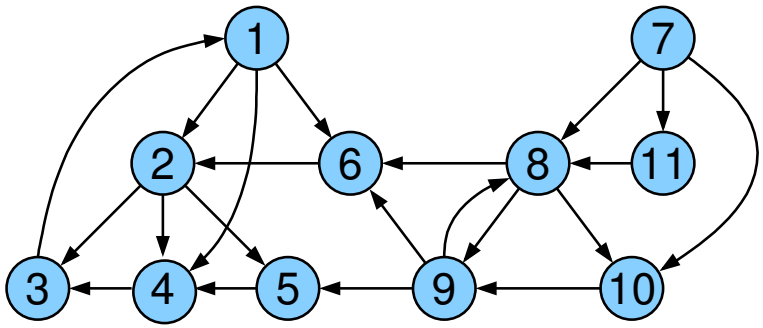
Starke Zhk. und DFS / Variante 2

Idee:

- beginne mit Graph ohne Kanten, jeder Knoten ist eigene SCC
- füge nach und nach einzelne Kanten ein

$\Rightarrow$ aktueller (current) Graph $G_c = (V, E_c)$

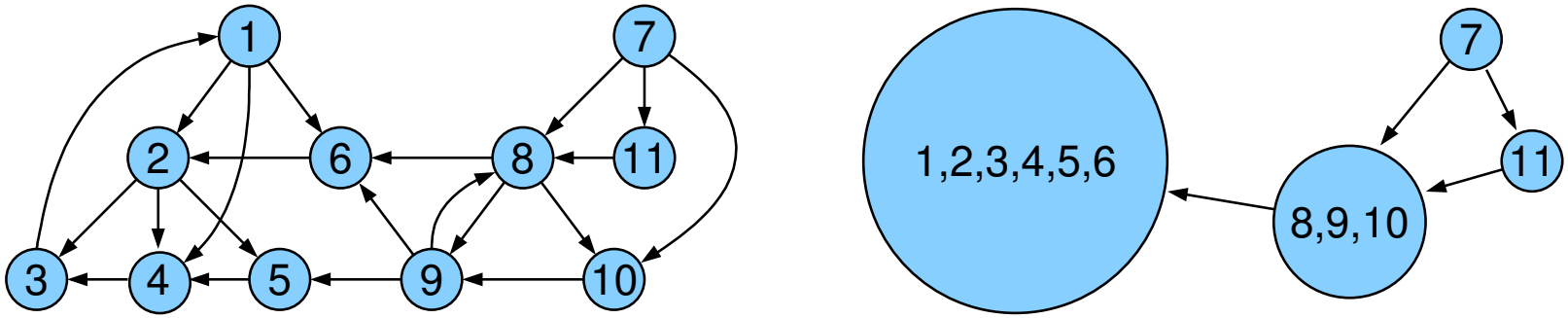
- Update der starken Zusammenhangskomponenten (SCCs)



Starke Zhk. und DFS / Variante 2

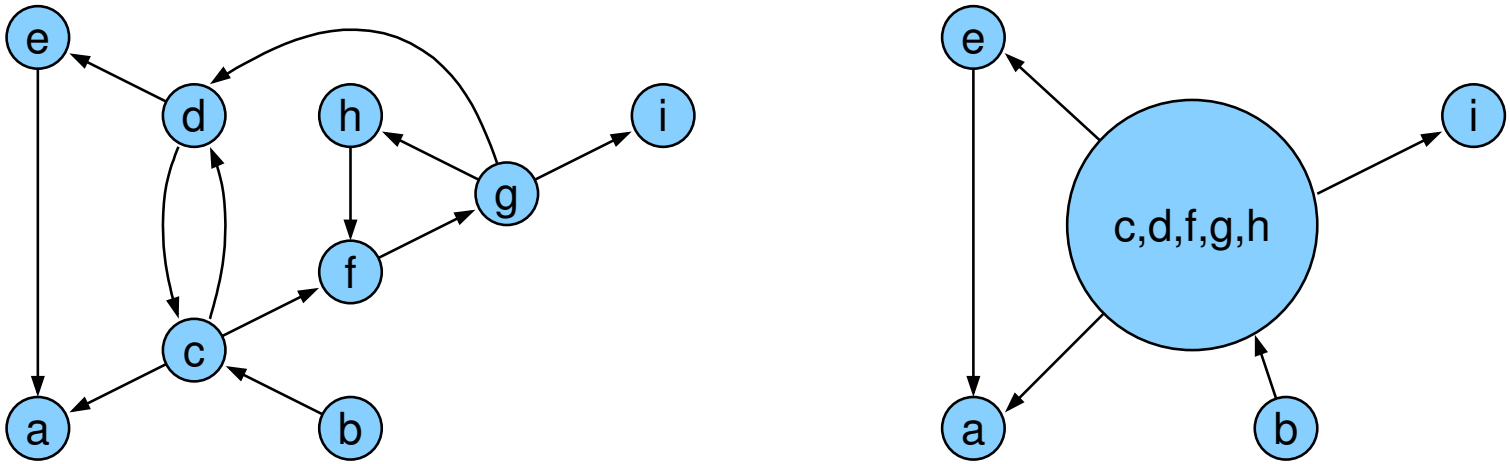
Idee:

- betrachte geschrumpften (shrunken) Graph G_c^s:
  Knoten entsprechen SCCs von G_c, Kante (C, D) genau dann, wenn es Knoten $u \in C$ und $v \in D$ mit $(u, v) \in E_c$ gibt
- geschrumpfter Graph G_c^s ist ein DAG
- Ziel: Aktualisierung des geschrumpften Graphen beim Einfügen



Starke Zhk. und DFS / Variante 2

Geschrumpfter Graph
(Beispiel aus Mehlhorn / Sanders)



Starke Zhk. und DFS / Variante 2

Update des geschrumpften Graphen nach Einfügen einer Kante:

3 Möglichkeiten:

- beide Endpunkte gehören zu derselben SCC

$\Rightarrow$ geschrumpfter Graph unverändert

- Kante verbindet Knoten aus zwei verschiedenen SCCs, aber schließt keinen Kreis

$\Rightarrow$ SCCs im geschrumpften Graph unverändert, aber eine Kante wird im geschrumpften Graph eingefügt (falls nicht schon vorhanden)

- Kante verbindet Knoten aus zwei verschiedenen SCCs und schließt einen oder mehrere Kreise

$\Rightarrow$ alle SCCs, die auf einem der Kreise liegen, werden zu einer einzigen SCC verschmolzen

Starke Zhk. und DFS / Variante 2

Prinzip:

- Tiefensuche
  - V_c schon markierte (entdeckte) Knoten
  - E_c schon gefundene Kanten
- 3 Arten von SCC: unentdeckt, offen, geschlossen
- unentdeckte Knoten haben Ein- / Ausgangsgrad Null in G_c

$\Rightarrow$ zunächst bildet jeder Knoten eine eigene unentdeckte SCC, andere SCCs enthalten nur markierte Knoten

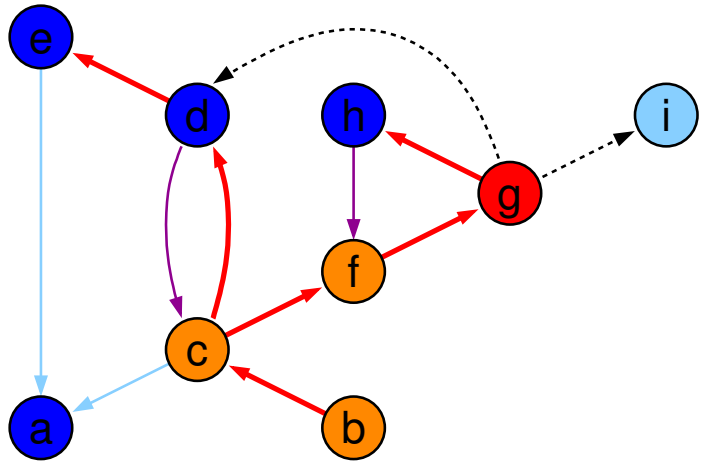
- SCCs mit mindestens einem aktiven Knoten (ohne finishNum) heißen offen
- SCC heißt geschlossen, falls sie nur fertige Knoten (mit finishNum) enthält
- Knoten in offenen / geschlossenen SCCs heißen offen / geschlossen

Starke Zhk. und DFS / Variante 2

- Knoten in geschlossenen SCCs sind immer fertig (mit finishNum)
- Knoten in offenen SCCs können fertig oder noch aktiv (ohne finishNum) sein
- Repräsentant einer SCC: Knoten mit kleinster dfsNum

Starke Zhk. und DFS / Variante 2

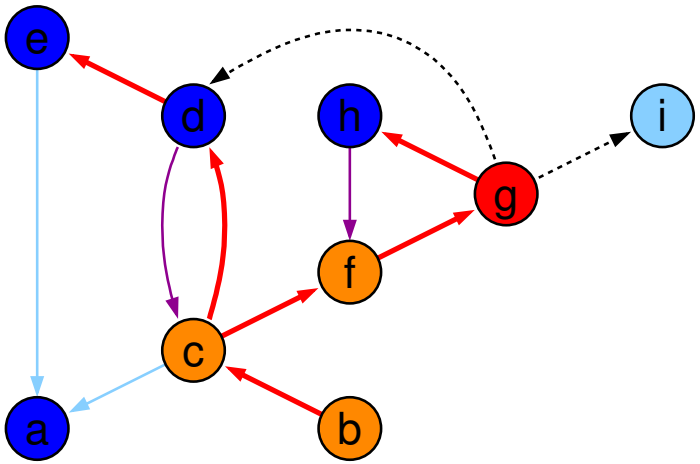
DFS-Snapshot:

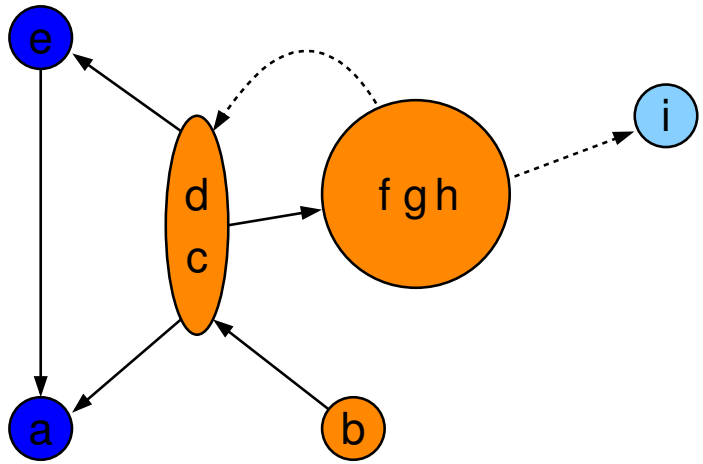


- erste DFS startete bei Knoten a, zweite bei b
- aktueller Knoten ist g, auf dem Rekursionsstack liegen b, c, f, g
- (g, d) und (g, i) wurden noch nicht exploriert
- (d, c) und (h, f) sind Rückwärtskanten
- (c, a) und (e, a) sind Querkanten
- (b, c), (c, d), (d, e), (c, f), (f, g) und (g, h) sind Baumkanten

Starke Zhk. und DFS / Variante 2

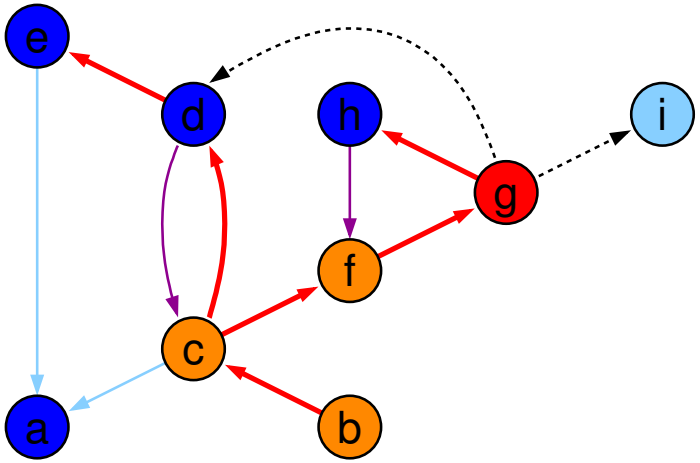
DFS-Snapshot mit geschrumpftem Graph:

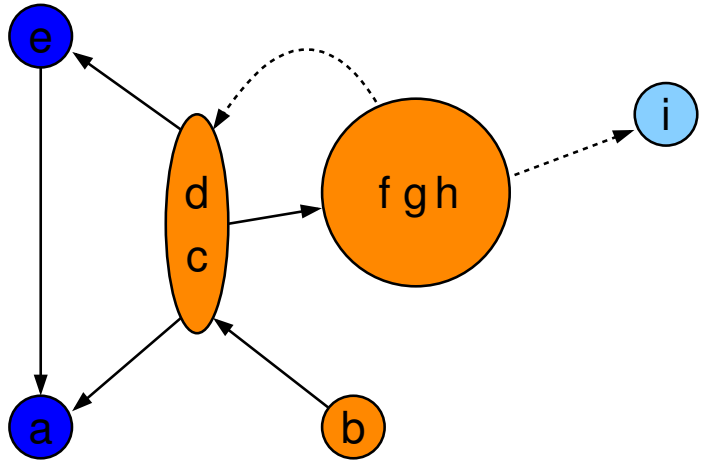




- unentdeckt: $\{i\}$ offen: $\{b\}$, $\{c, d\}$, $\{f, g, h\}$ geschlossen: $\{a\}$, $\{e\}$
- offene SCCs bilden Pfad im geschrumpften Graph
- aktueller Knoten gehört zur letzten SCC
- offene Knoten wurden in Reihenfolge b, c, d, f, g, h erreicht und werden von den Repräsentanten b, c und f genau in die offenen SCCs partitioniert

Starke Zhk. und DFS / Variante 2





Beobachtungen (Invarianten für G_c):

1. Pfade aus geschlossenen SCCs führen immer zu geschlossenen SCCs
2. Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen SCCs

   offene Komponenten bilden Pfad im geschrumpften Graph
3. Knoten der offenen SCCs in Reihenfolge der DFS-Nummern werden durch Repräsentanten in die offenen SCCs partitioniert

Starke Zhk. und DFS / Variante 2

Geschlossene SCCs von G_c sind auch SCCs in G:

- Sei v geschlossener Knoten und S / S_c seine SCC in G / G_c.
- zu zeigen: $S = S_c$
- G_c ist Subgraph von G, also $S_c \subseteq S$
- somit zu zeigen: $S \subseteq S_c$

- Sei w ein Knoten in S.

$\Rightarrow$ $\exists$ Kreis C durch v und w.

- Invariante 1: alle Knoten von C sind geschlossen und somit erledigt (alle ausgehenden Kanten exploriert)
- C ist in G_c enthalten, also $w \in S_c$
- damit gilt $S \subseteq S_c$, also $S = S_c$

Starke Zhk. und DFS / Variante 2

Vorgehen:

- Invarianten 2 und 3 helfen bei Verwaltung der offenen SCCs
- Knoten in offenen SCCs auf Stack oNodes
  (in Reihenfolge steigender dfsNum)
- Repräsentanten der offenen SCCs auf Stack oReps
- zu Beginn Invarianten gültig (alles leer)
- vor Markierung einer neuen Wurzel sind alle markierten Knoten erledigt, also keine offenen SCCs, beide Stacks leer
  dann: neue offene SCC für neue Wurzel s,
  s kommt auf beide Stacks

Starke Zhk. und DFS / Variante 2

Prinzip: betrachte Kante $e = (v, w)$

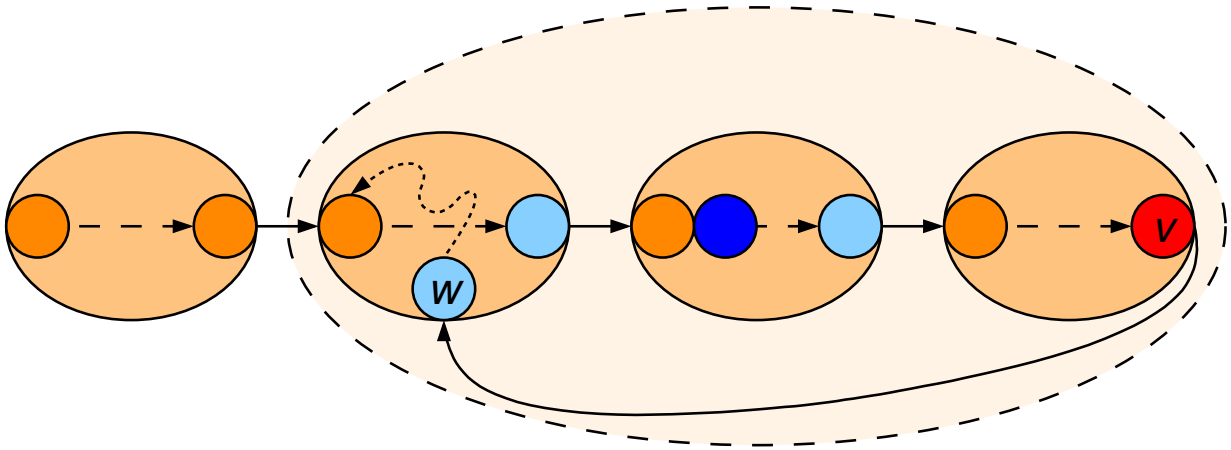
- Kante zu unbekanntem Knoten w (Baumkante):
  neue eigene offene SCC für w (w kommt auf oNodes und oReps)
- Kante zu Knoten w in geschlossener SCC (Nicht-Baumkante):
  von w gibt es keinen Weg zu v, sonst wäre die SCC von w noch nicht geschlossen (geschlossene SCCs sind bereits komplett), also SCCs unverändert
- Kante zu Knoten w in offener SCC (Nicht-Baumkante):
  falls v und w in unterschiedlichen SCCs liegen, müssen diese mit allen SCCs dazwischen zu einer einzigen SCC verschmolzen werden (durch Löschen der Repräsentanten)

Wenn Knoten keine ausgehenden Kanten mehr hat:

- Knoten fertig
- wenn Knoten Repräsentant seiner SCC ist, dann SCC schließen

Starke Zhk. und DFS / Variante 2

Vereinigung offener SCCs im Kreisfall:



- offene SCC entsprechen Ovalen, Knoten sortiert nach dfsNum
- alle Repräsentanten offener SCCs liegen auf Baumpfad zum aktuellen Knoten v in SCC S_k
- Nicht-Baumkante (v, w) endet an Knoten w in offener SCC S_i mit Repräsentant r_i
- Pfad von w nach r_i muss existieren (innerhalb SCC S_i)

$\Rightarrow$ Kante (v, w) vereinigt $S_i, \ldots, S_k$

Starke Zhk. und DFS / Variante 2

- init() {
  component = new int[n];
  oReps = ⟨ ⟩;
  oNodes = ⟨ ⟩;
  dfsCount = 1;
  }

- root(Node w) / traverseTreeEdge(Node v, Node w) {
  oReps.push(w); // Repräsentant einer neuen ZHK
  oNodes.push(w); // neuer offener Knoten
  dfsNum[w] = dfsCount;
  dfsCount++;
  }

Starke Zhk. und DFS / Variante 2

- traverseNonTreeEdge(Node v, Node w) {
  if (w ∈ oNodes) // verschmelze SCCs
  while (dfsNum[w] < dfsNum[oReps.top()])
  oReps.pop();
  }

- backtrack(Node u, Node v) {
  if (v == oReps.top()) { // v Repräsentant?
  oReps.pop(); // ja: entferne v
  do { // und offene Knoten bis v
  w = oNodes.pop();
  component[w] = v;
  } while (w!=v);
  }
  }

Starke Zhk. und DFS / Variante 2

Zeit: $\mathcal{O}(n+m)$

Begründung:

- init, root: $\mathcal{O}(1)$
- traverseTreeEdge: $(n-1) \times \mathcal{O}(1)$
- backtrack, traverseNonTreeEdge:
  da jeder Knoten höchstens einmal in oReps und oNodes landet,
  insgesamt $\mathcal{O}(n+m)$
- DFS-Gerüst: $\mathcal{O}(n+m)$
- gesamt: $\mathcal{O}(n+m)$

Übersicht

Kürzeste Wege

Zentrale Frage: Wie kommt man am schnellsten von A nach B?

Kürzeste Wege

Zentrale Frage: Wie kommt man am schnellsten von A nach B?

Fälle:

- Kantenkosten 1
- DAG, beliebige Kantenkosten
- beliebiger Graph, positive Kantenkosten
- beliebiger Graph, beliebige Kantenkosten

Kürzeste-Wege-Problem

gegeben:

- gerichteter Graph $G = (V, E)$
- Kantenkosten $c : E \mapsto \mathbb{R}$

2 Varianten:

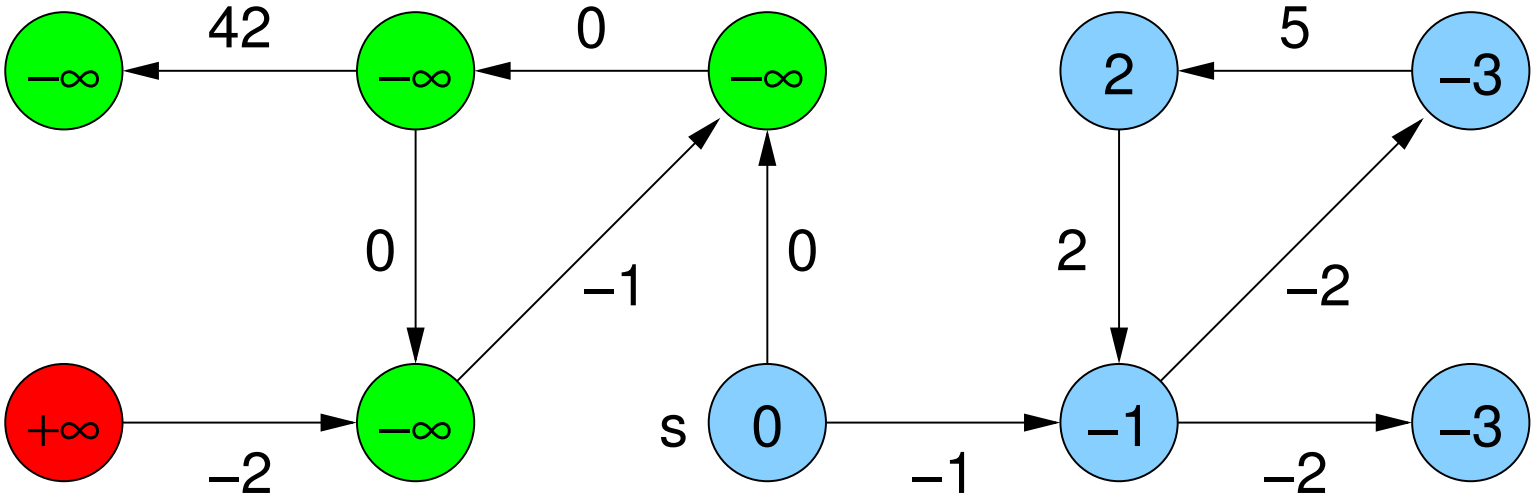
- SSSP (single source shortest paths):

  kürzeste Wege von einer Quelle zu allen anderen Knoten

- APSP (all pairs shortest paths):

  kürzeste Wege zwischen allen Paaren

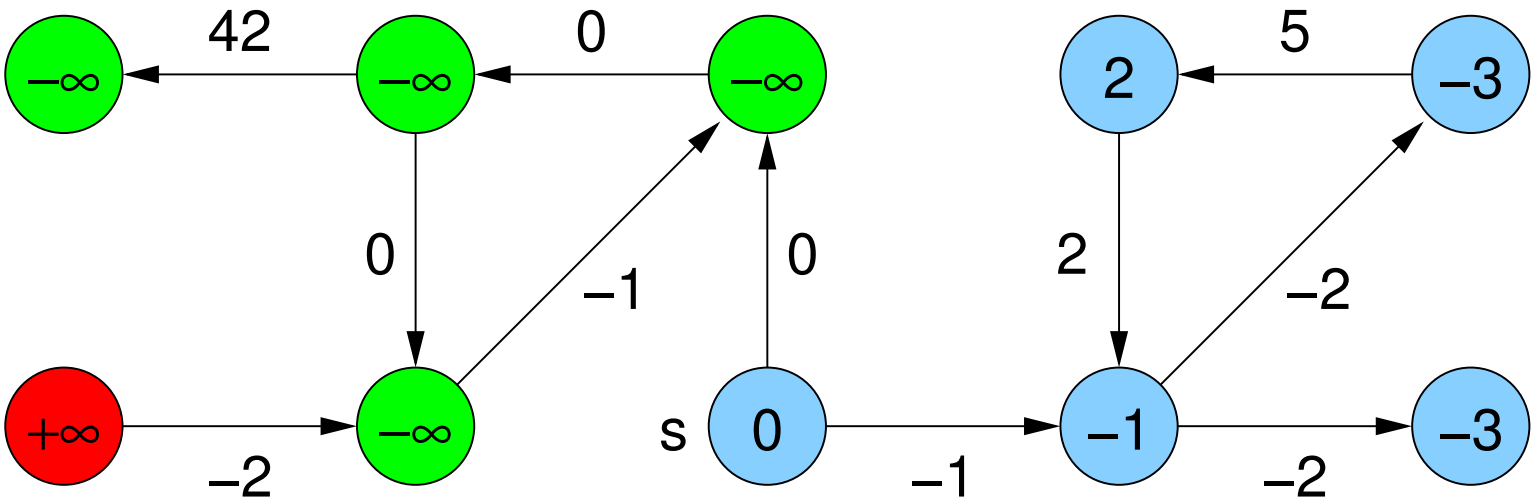
Distanzen



$\mu(s,v)$: Distanz von s nach v

$$\mu(s,v) = \begin{cases} +\infty & \text{kein Weg von } s \text{ nach } v \\ -\infty & \text{Weg beliebig kleiner Kosten von } s \text{ nach } v \\ \min\{c(p) : p \text{ ist Weg von } s \text{ nach } v\} \end{cases}$$

Distanzen



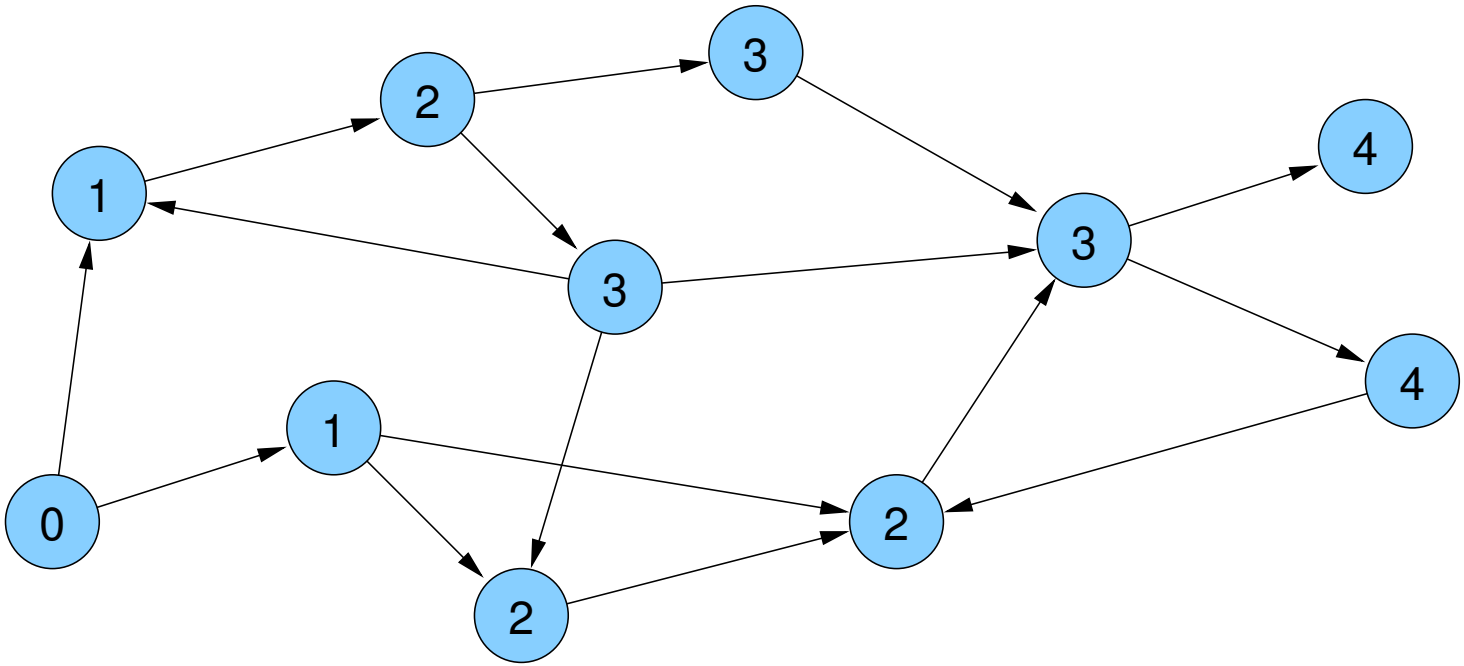
Wann sind die Kosten $-\infty$?

wenn es einen Kreis mit negativer Gewichtssumme gibt
(hinreichende und notwendige Bedingung)

Kürzeste Wege bei uniformen Kantenkosten

Graph mit Kantenkosten 1:

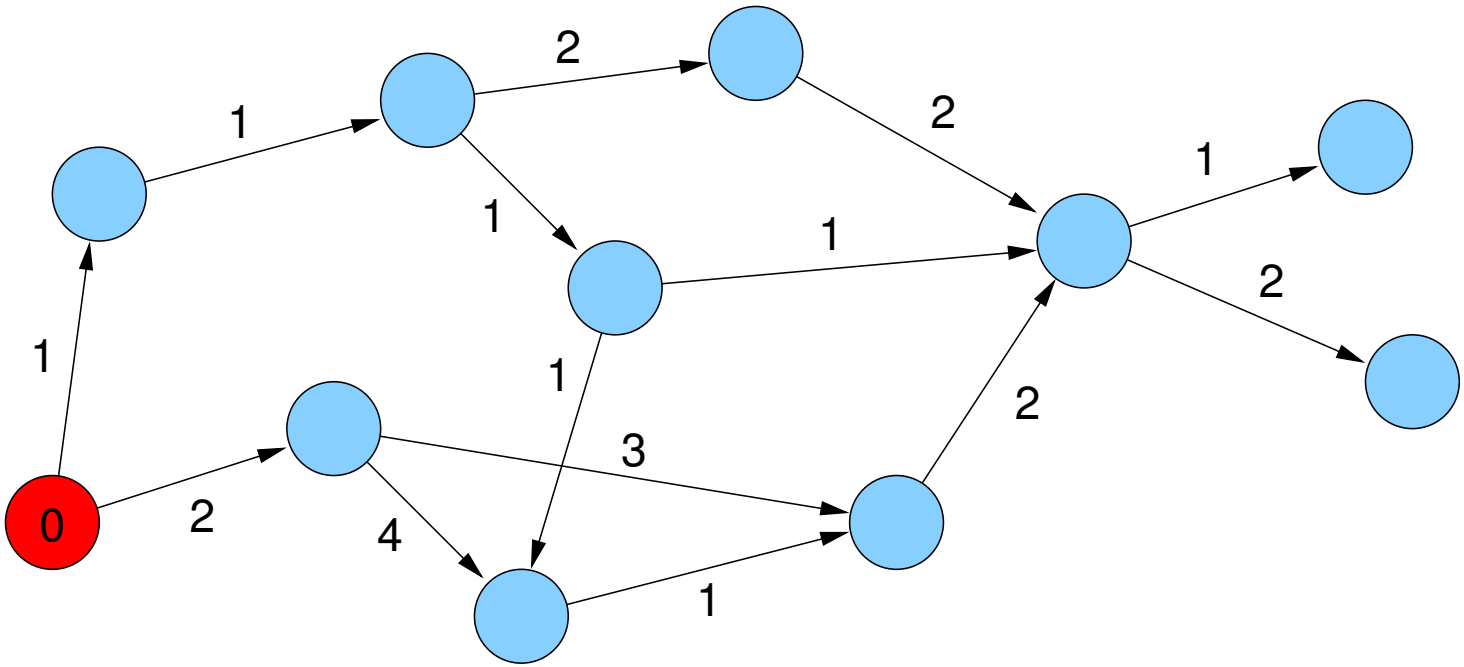
$\Rightarrow$ Breitensuche (BFS)



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

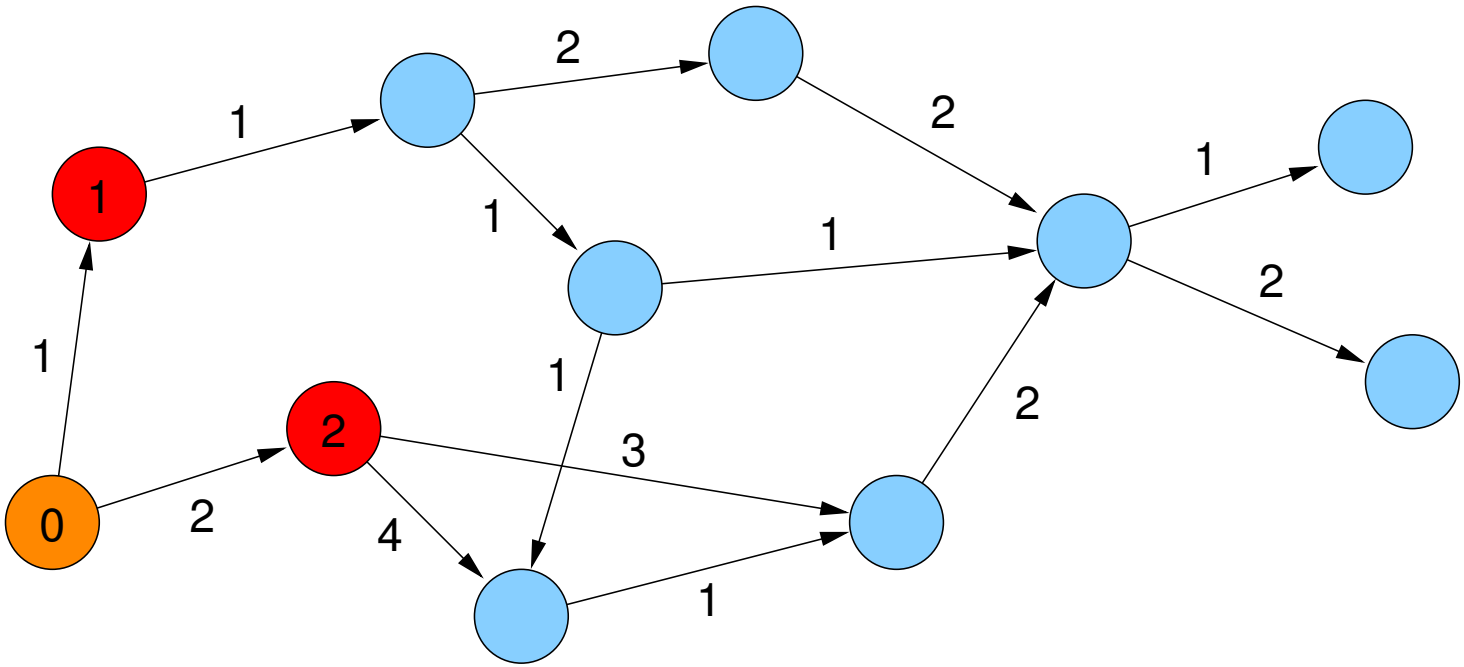
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

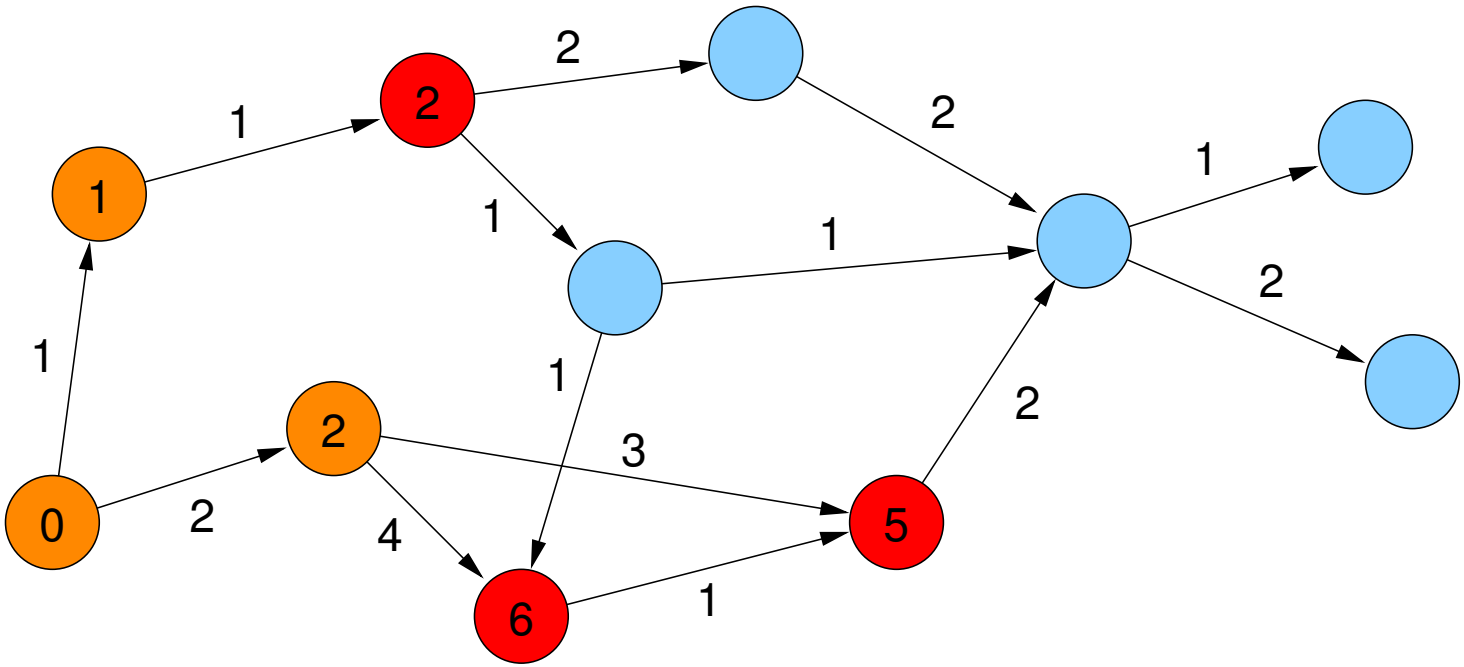
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

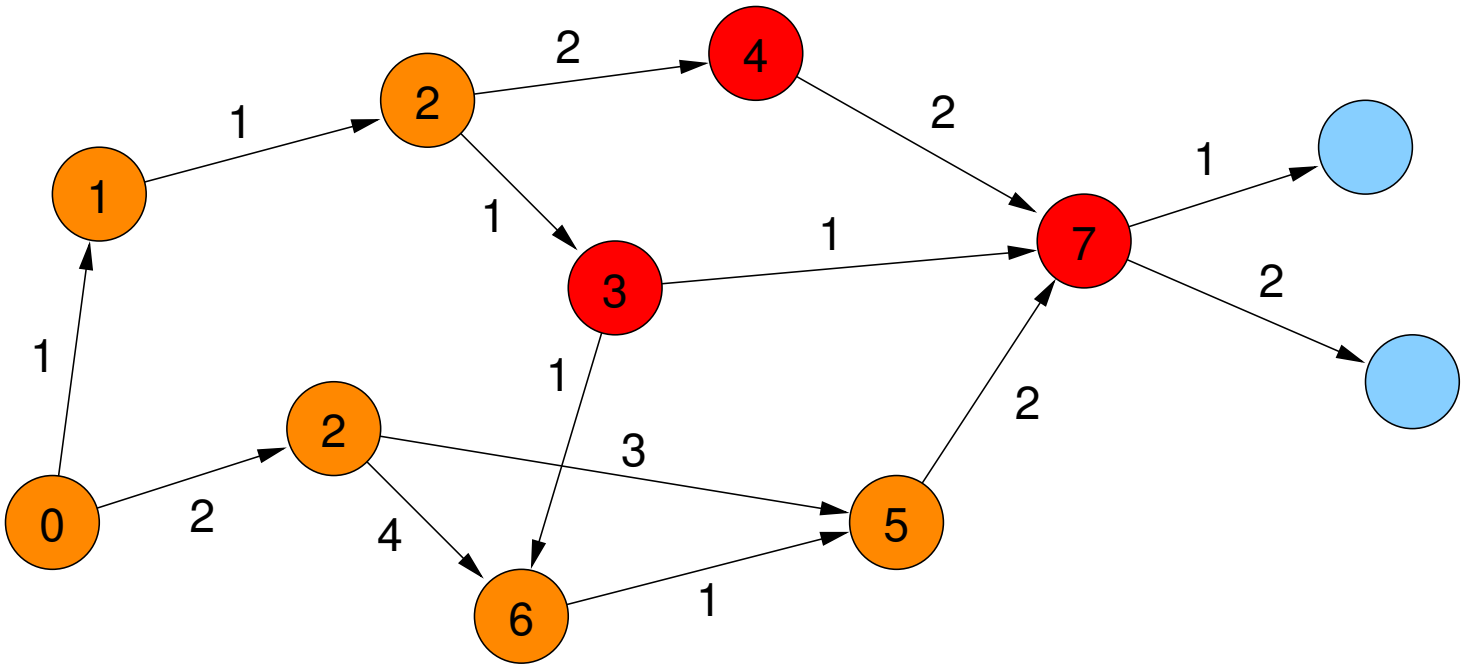
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

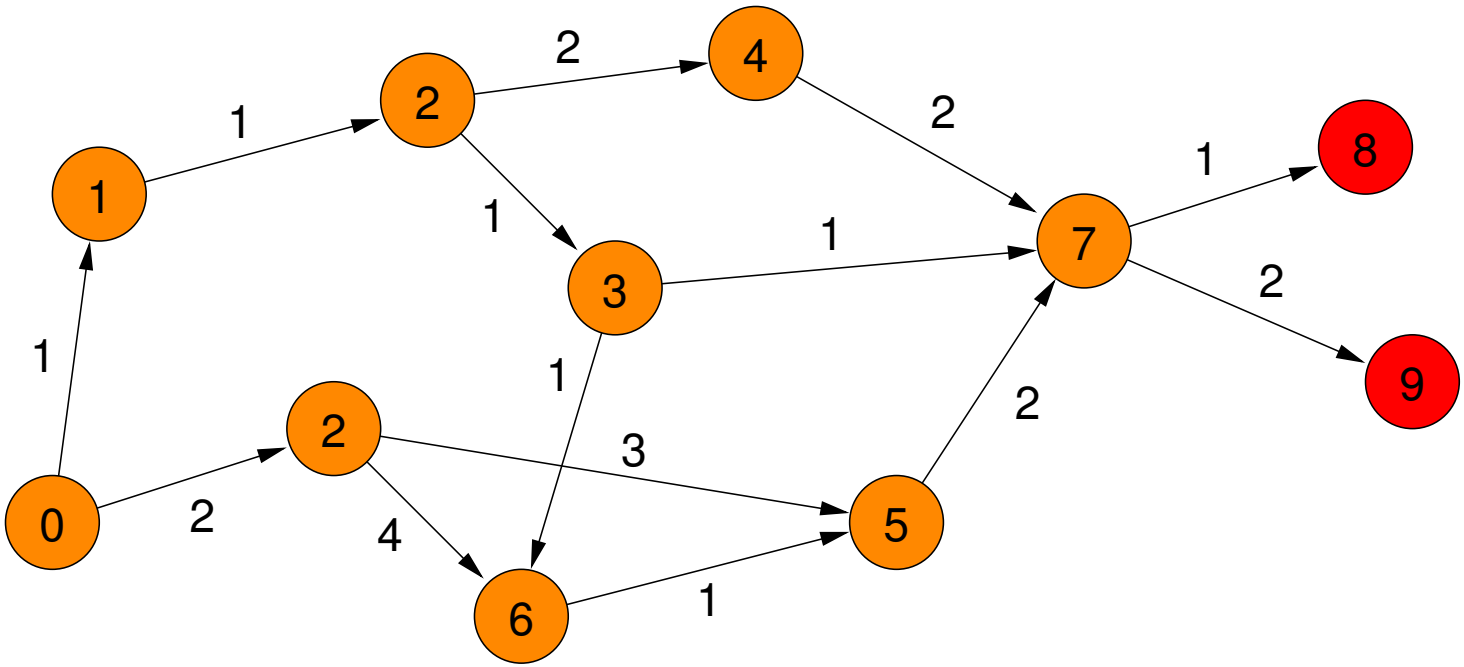
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

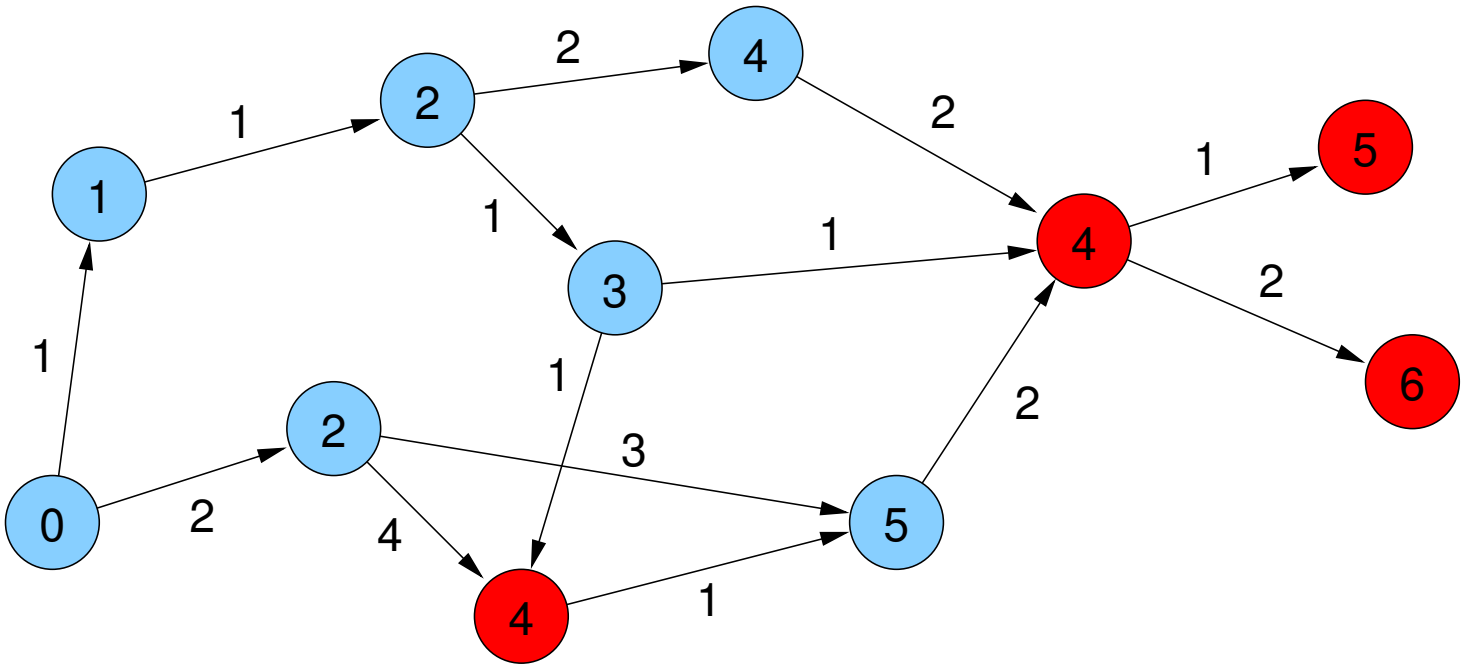
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Einfache Breitensuche funktioniert nicht.

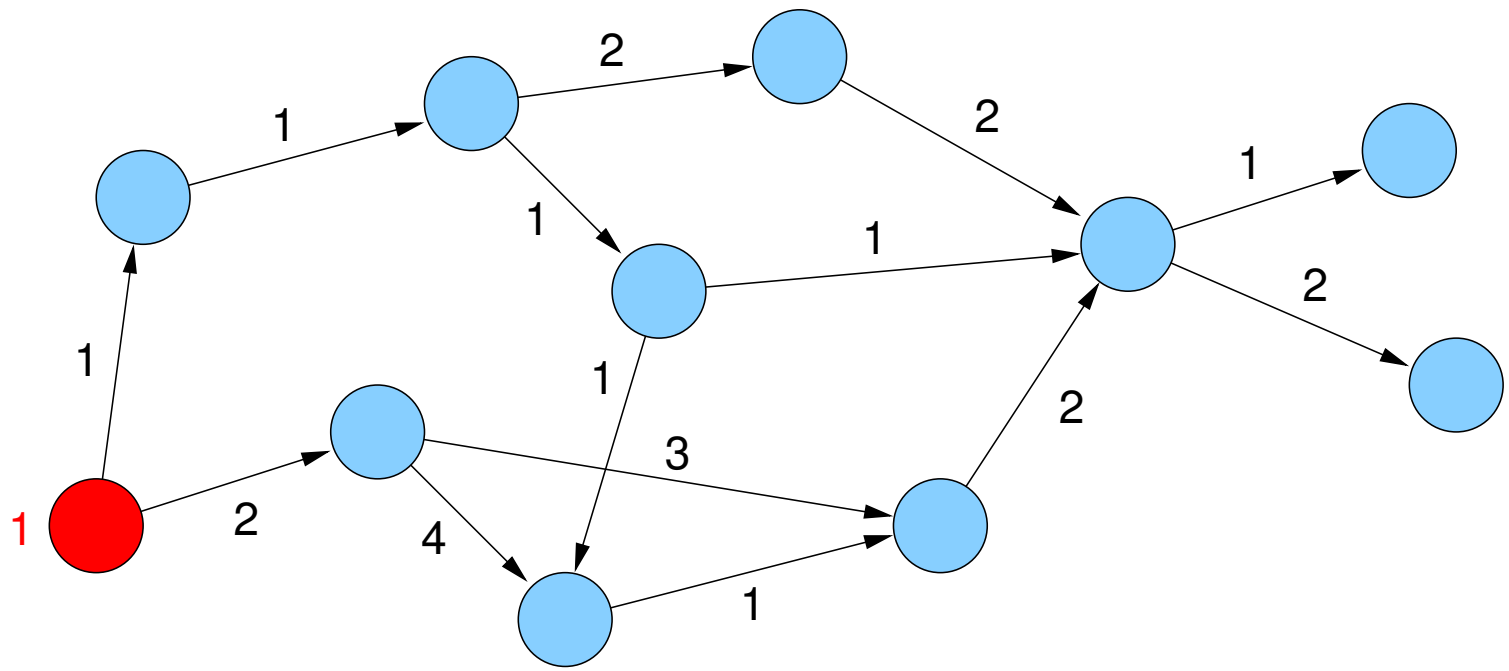


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

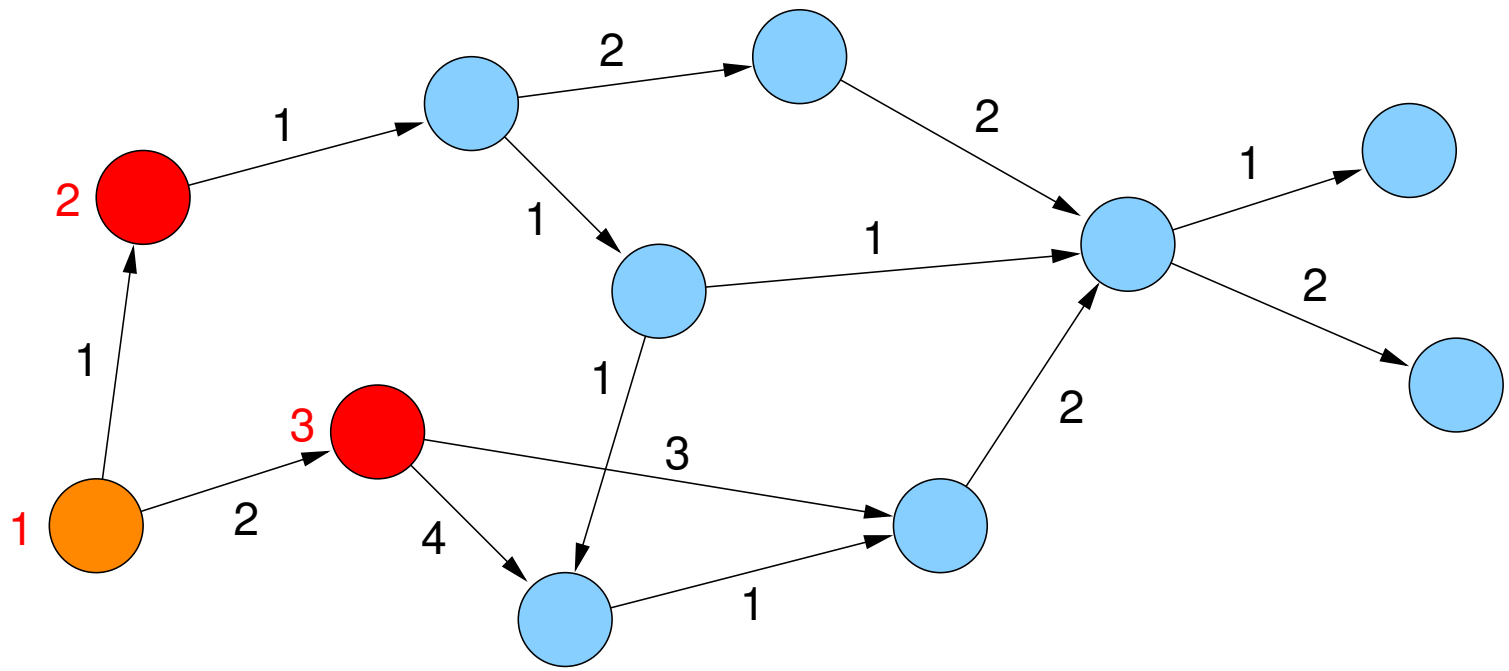


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

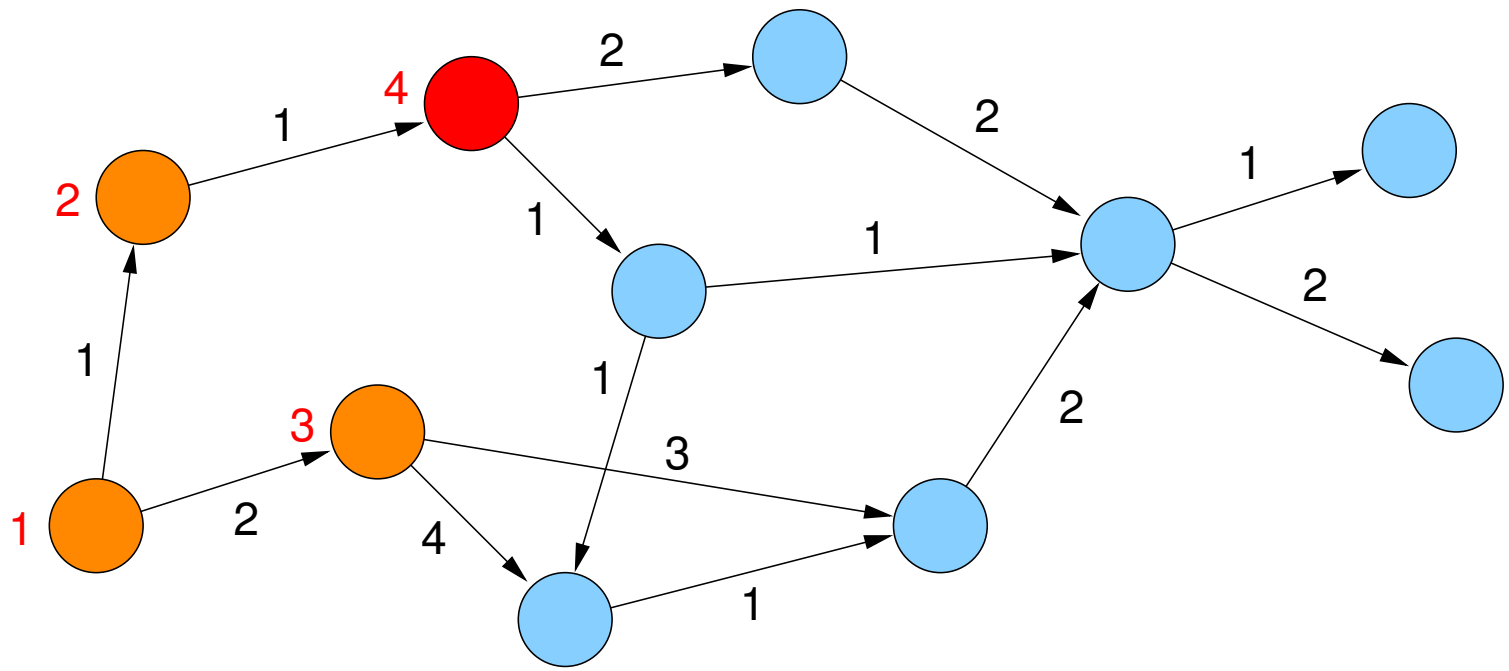


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

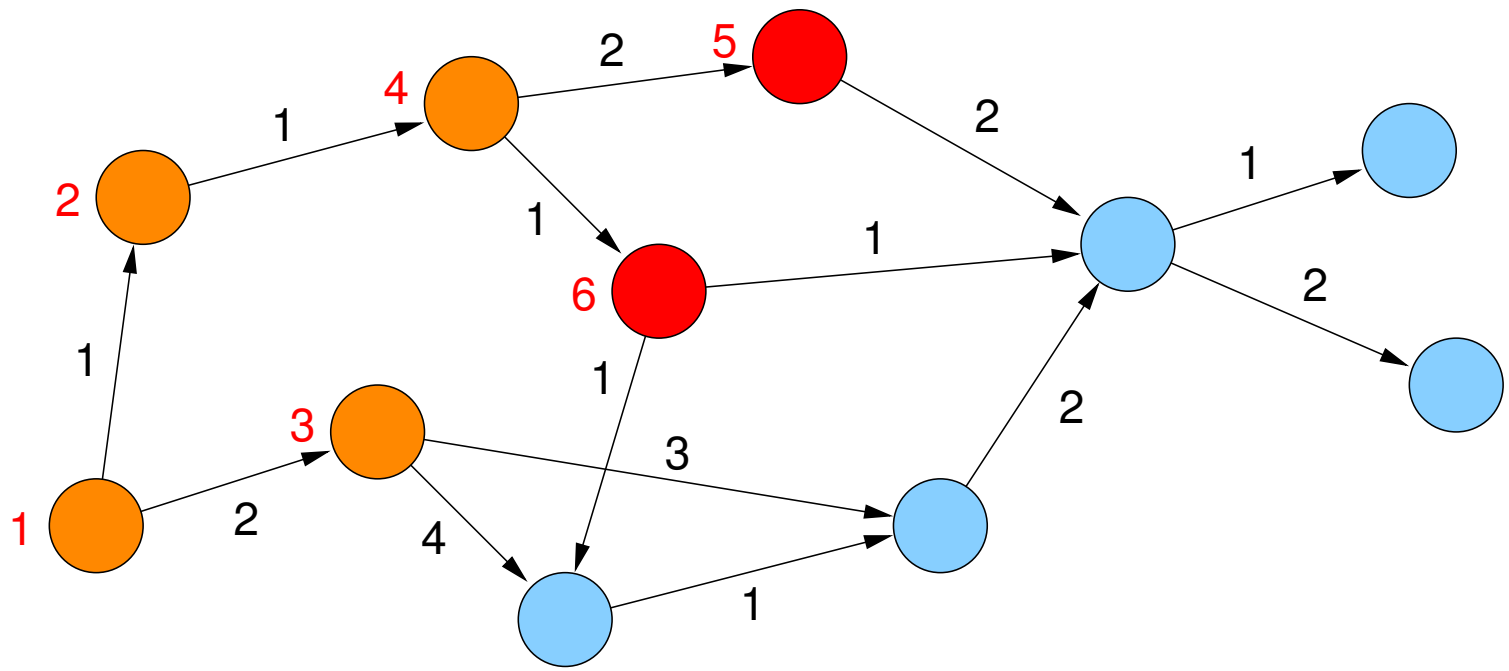


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

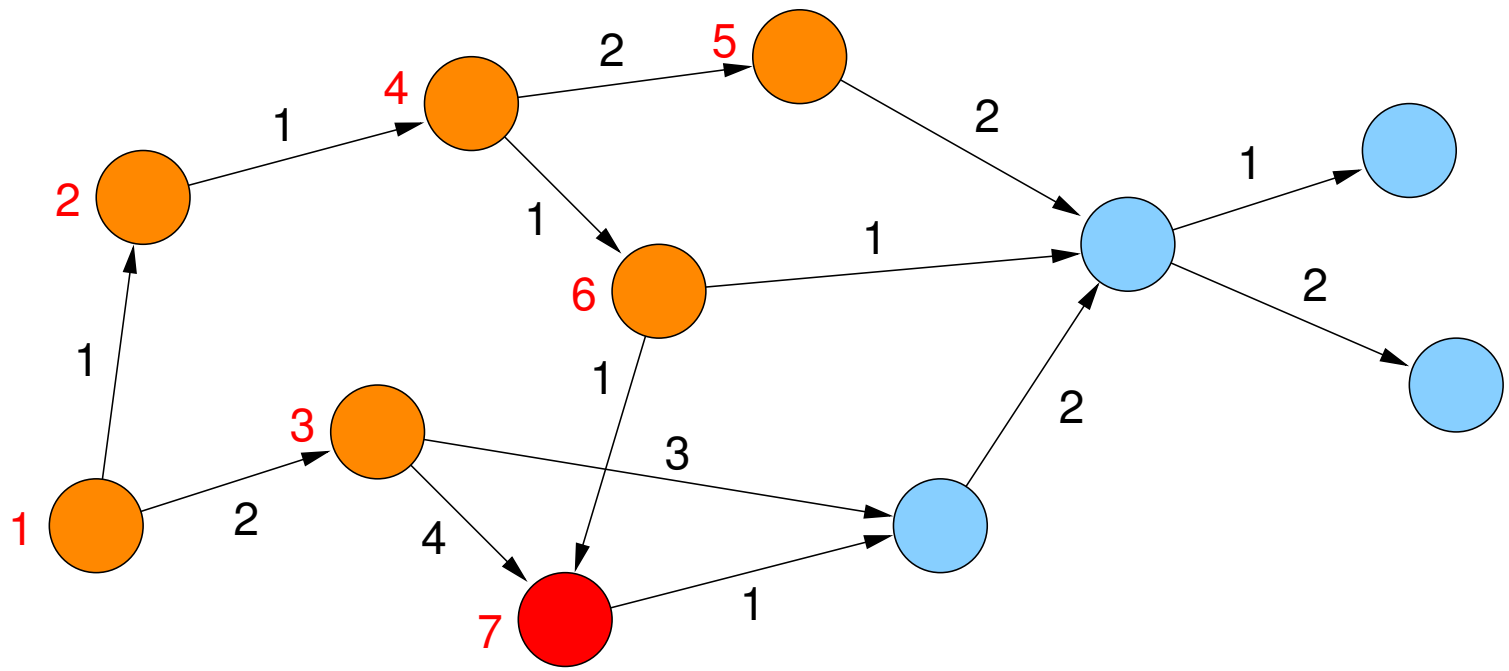


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

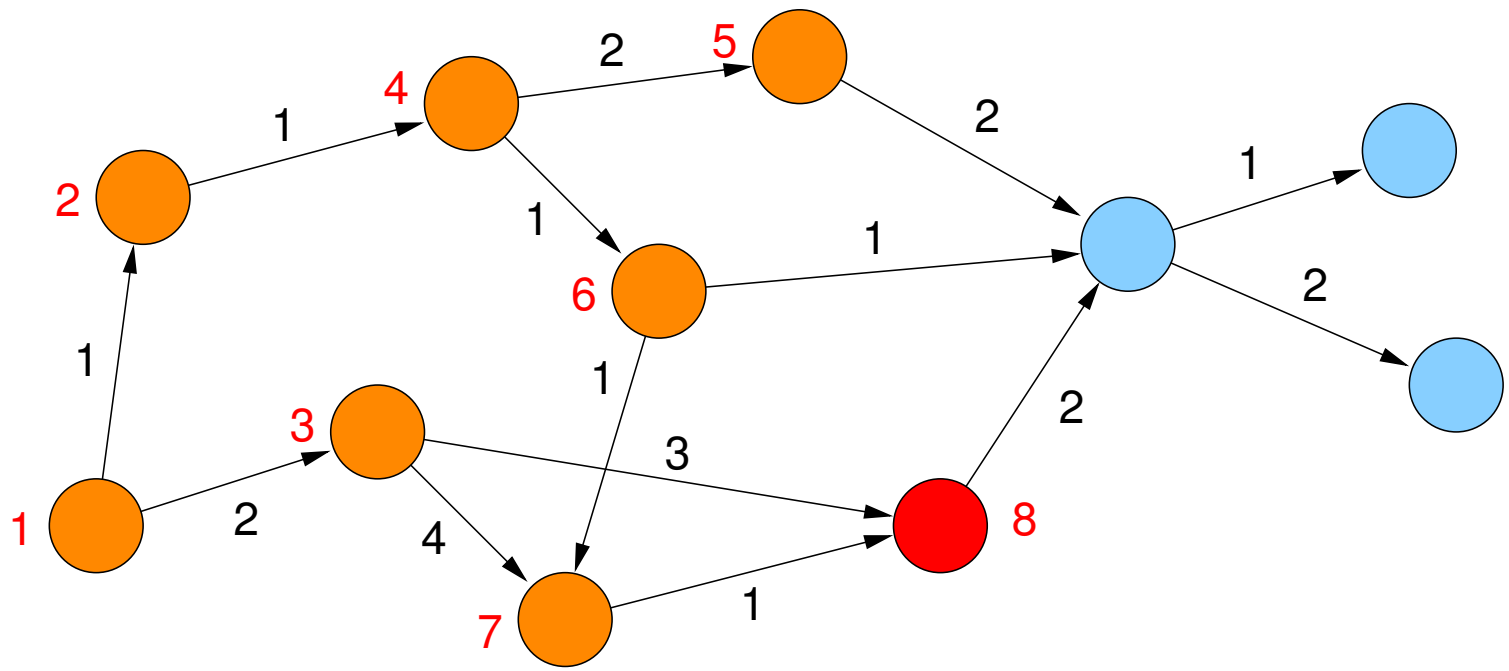


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

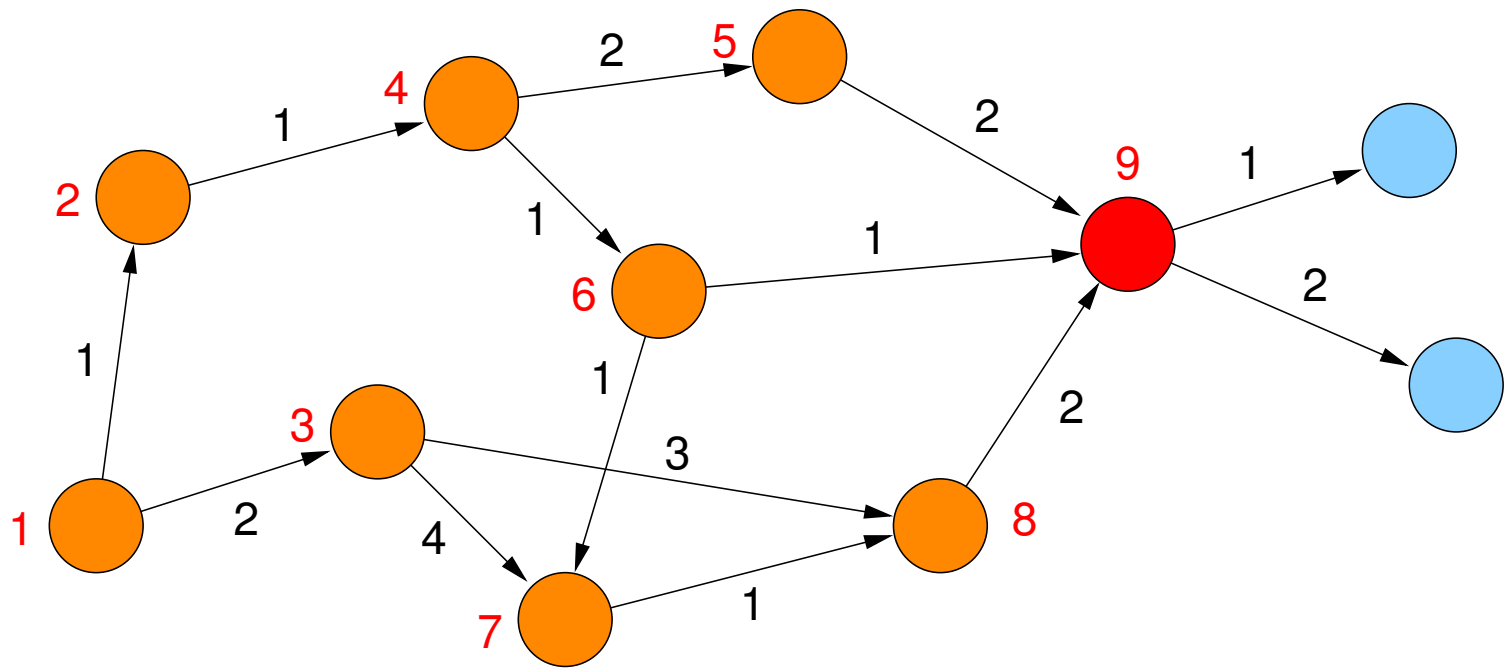


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

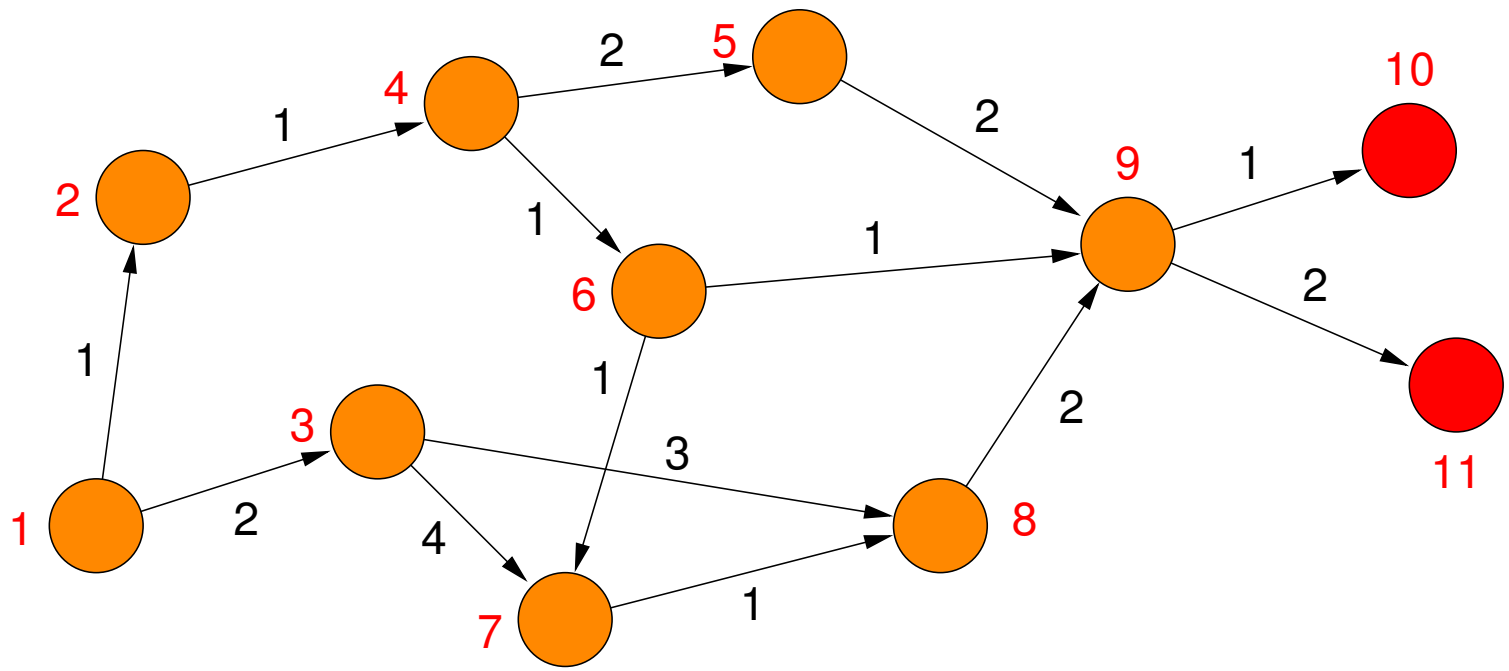


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

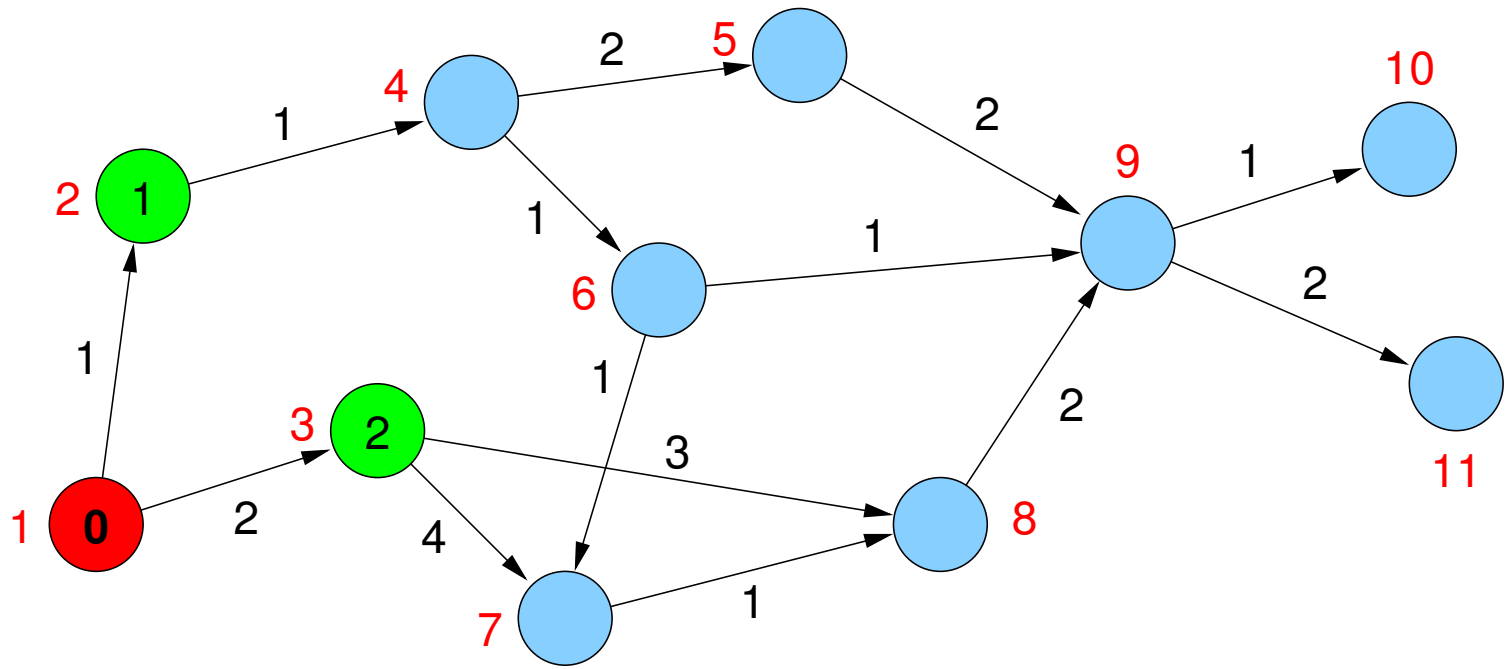


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

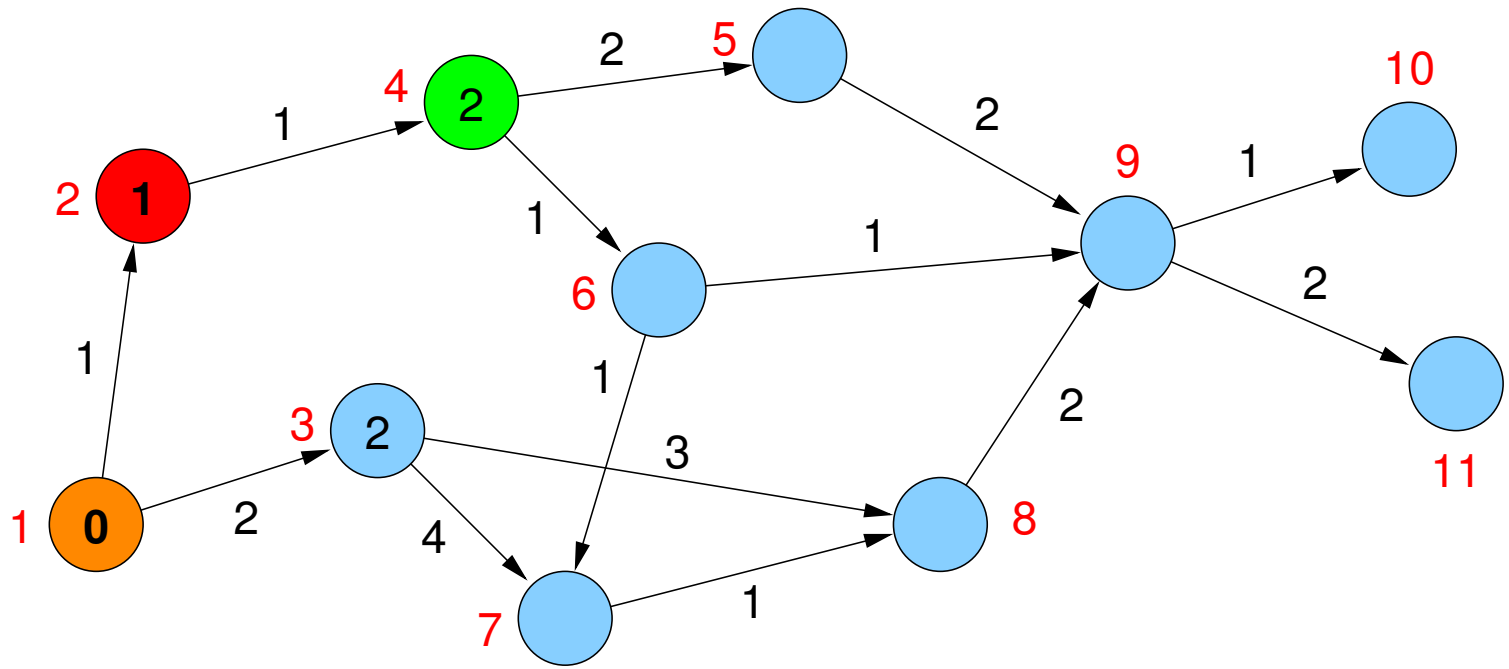


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

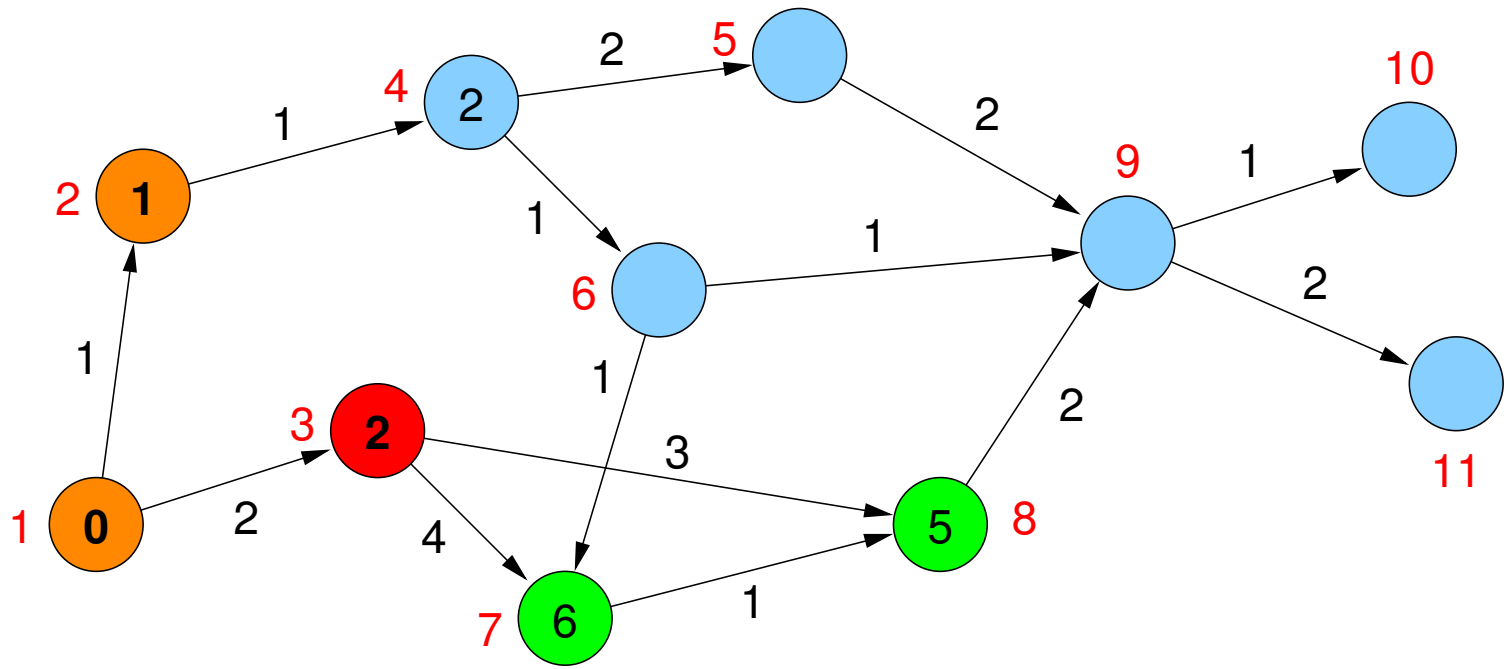


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

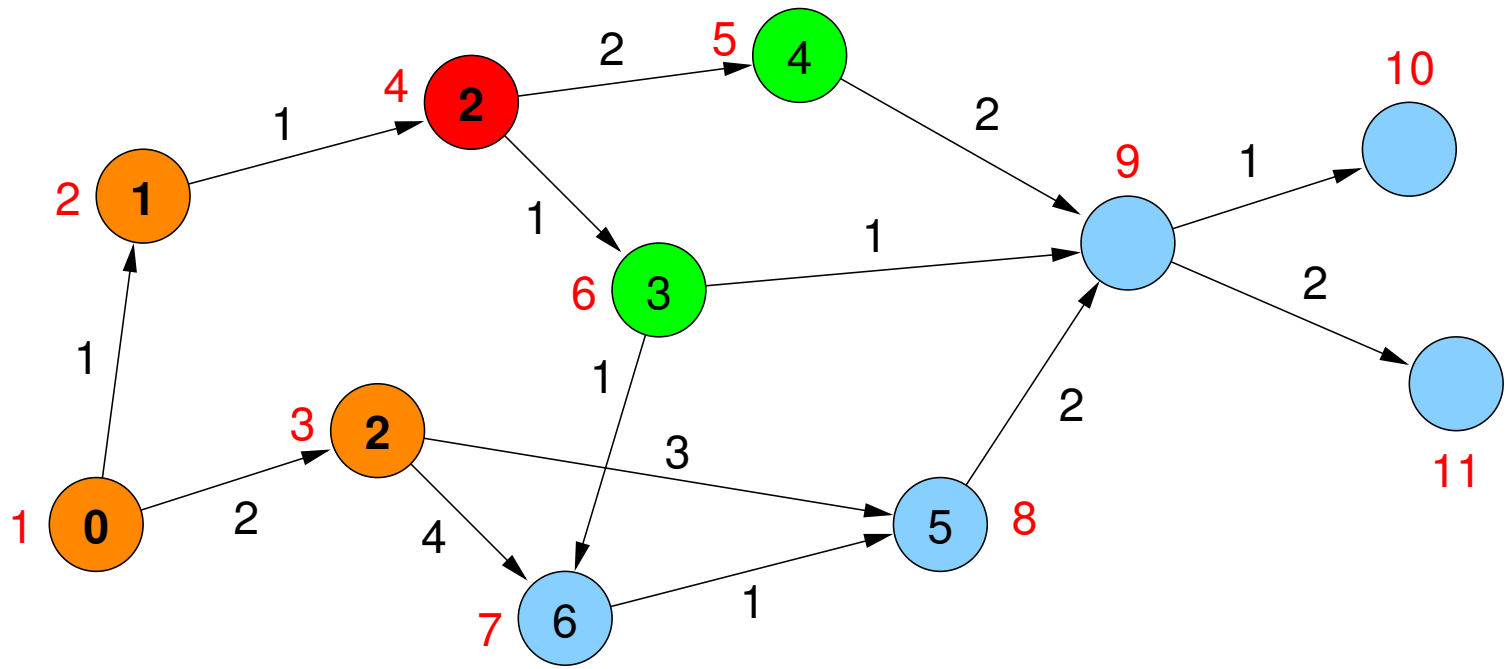


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

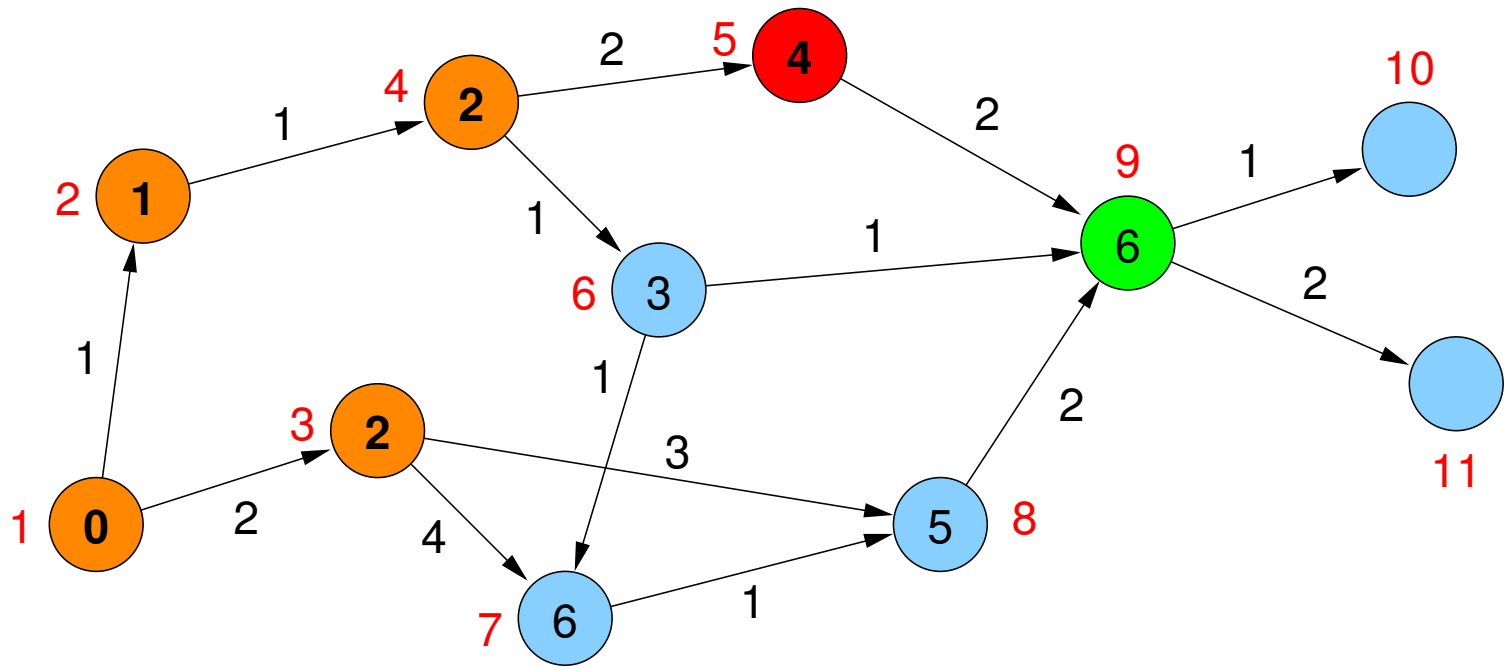


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

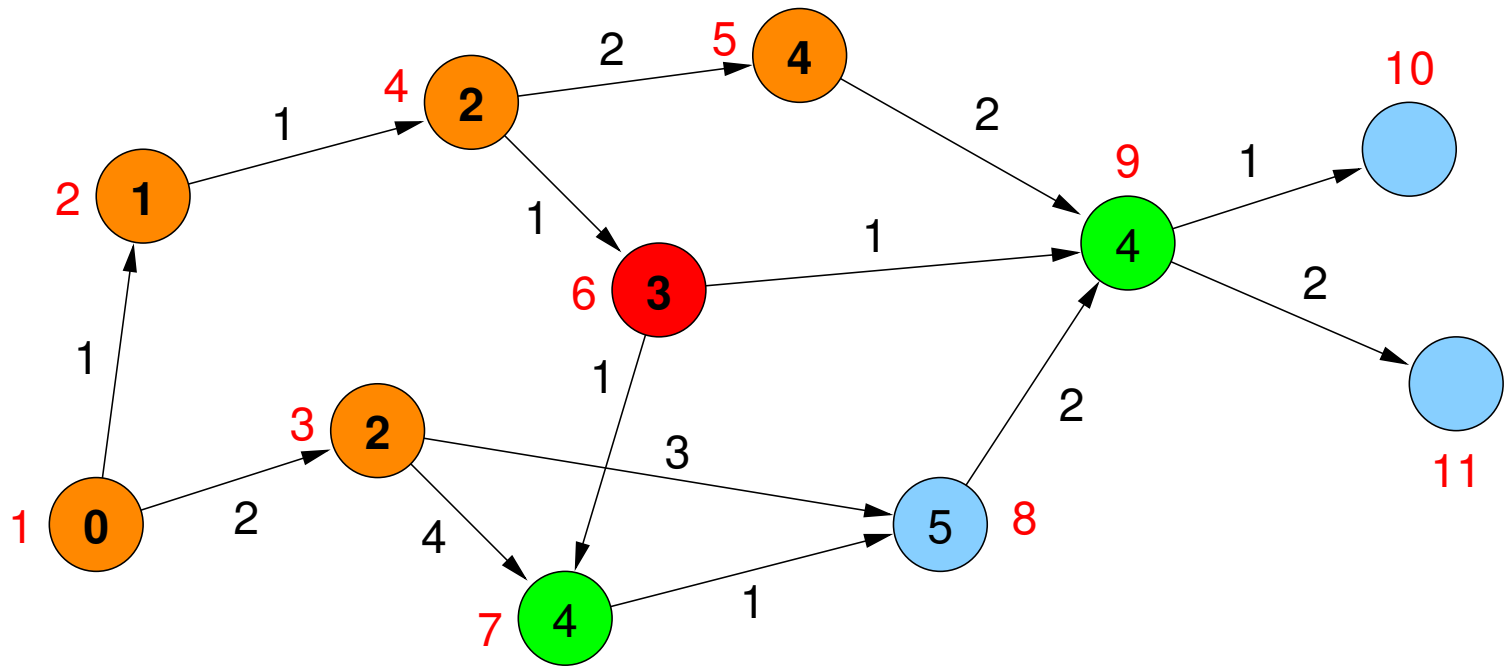


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

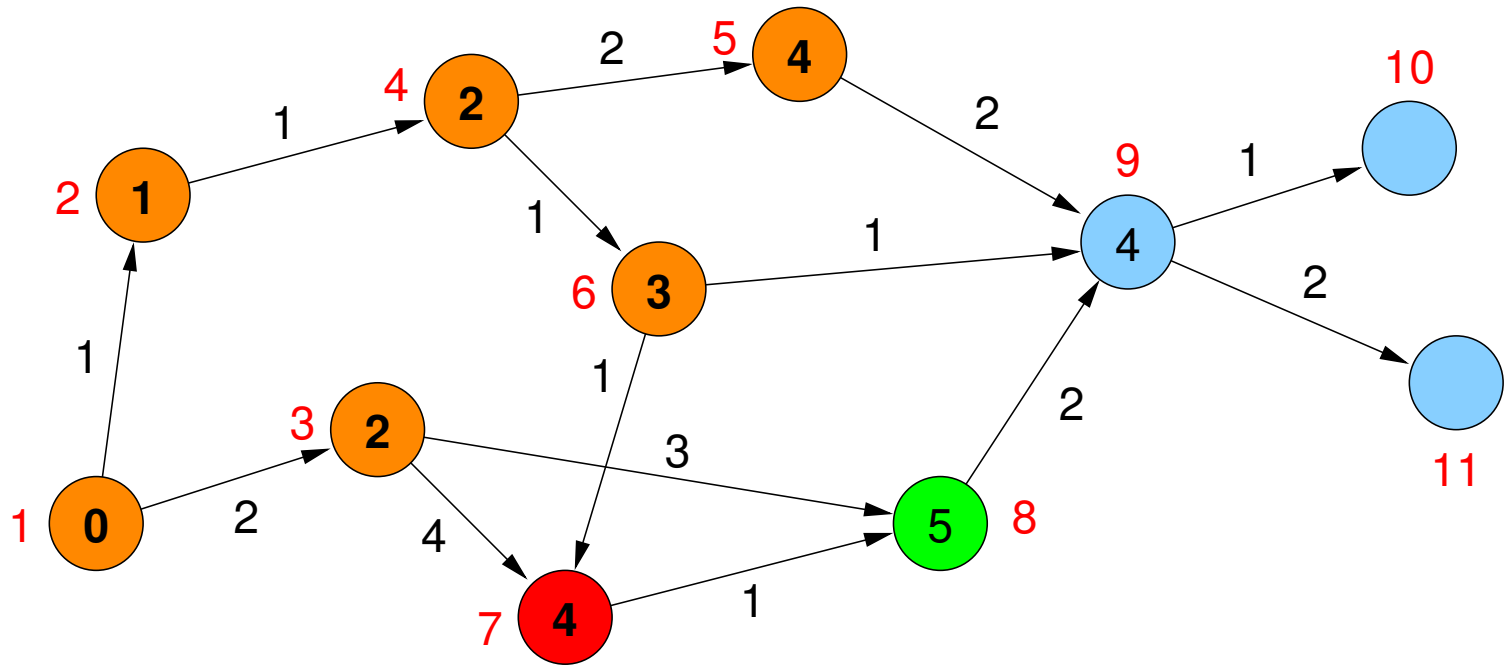


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

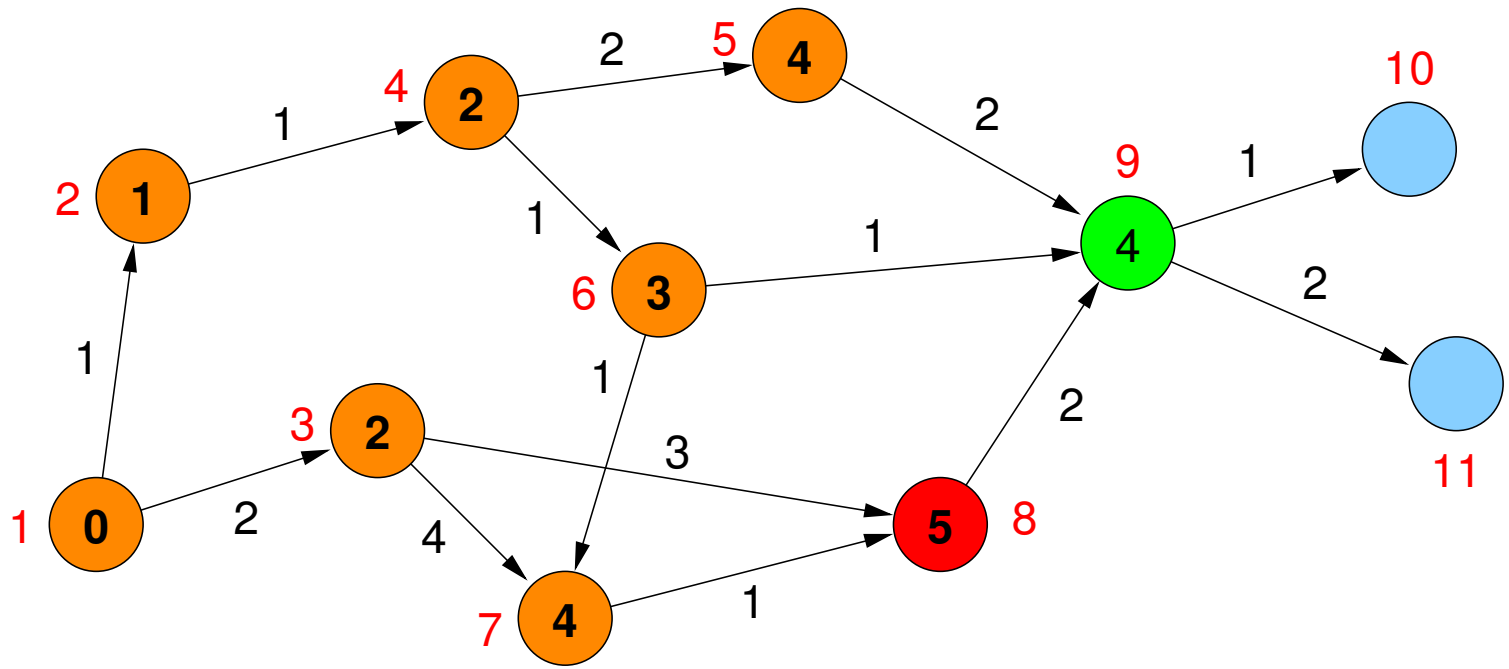


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

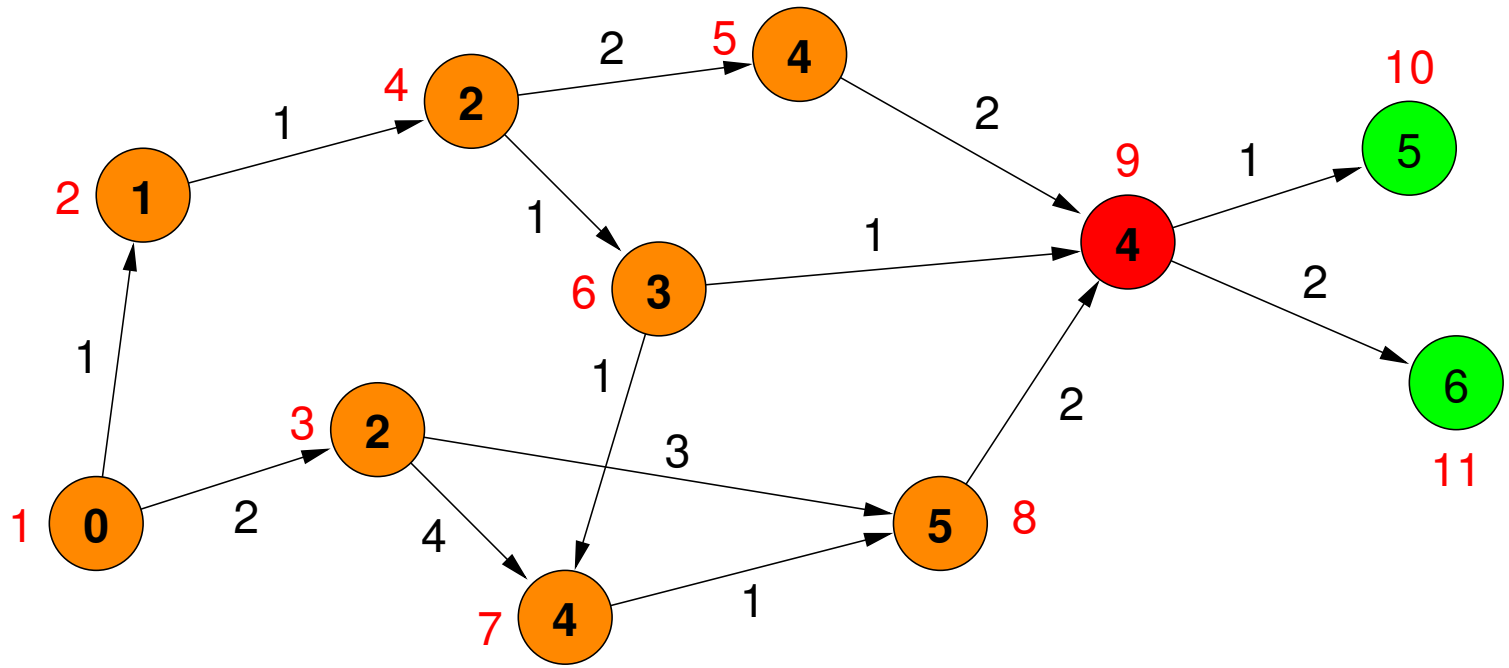


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

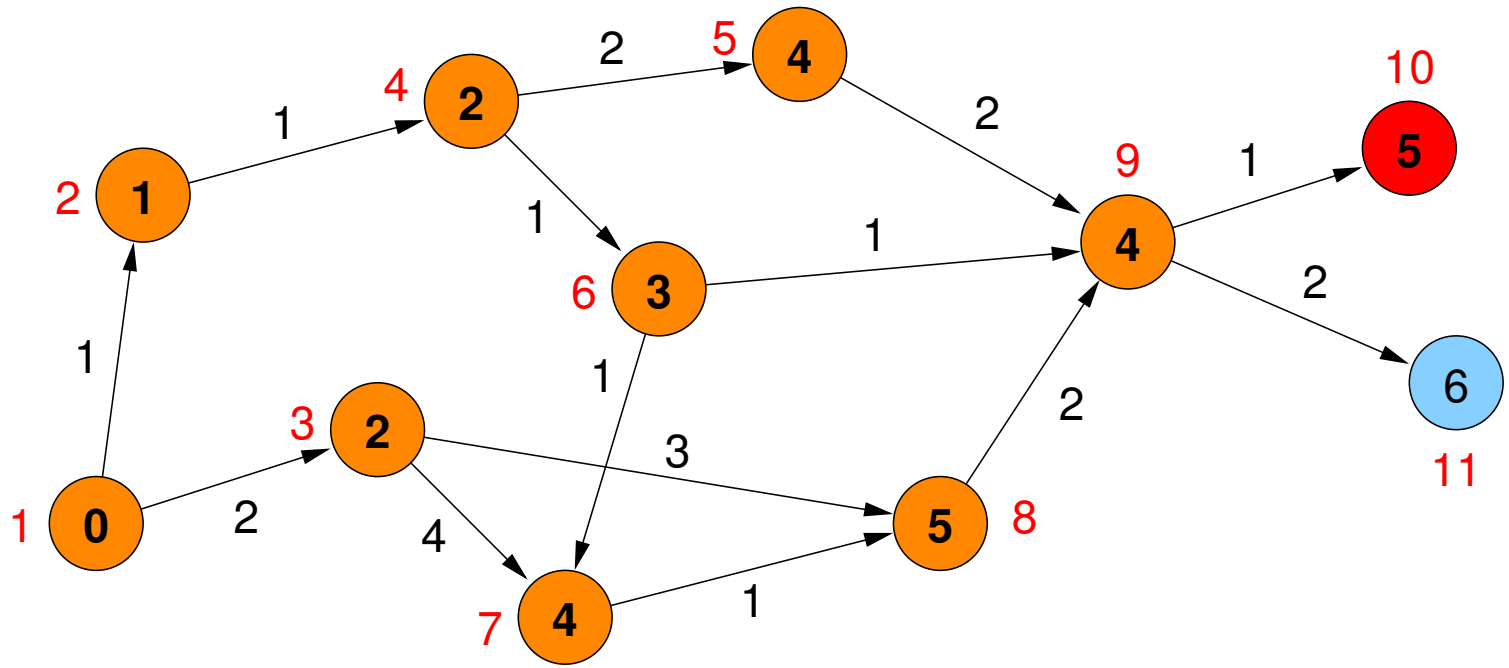


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

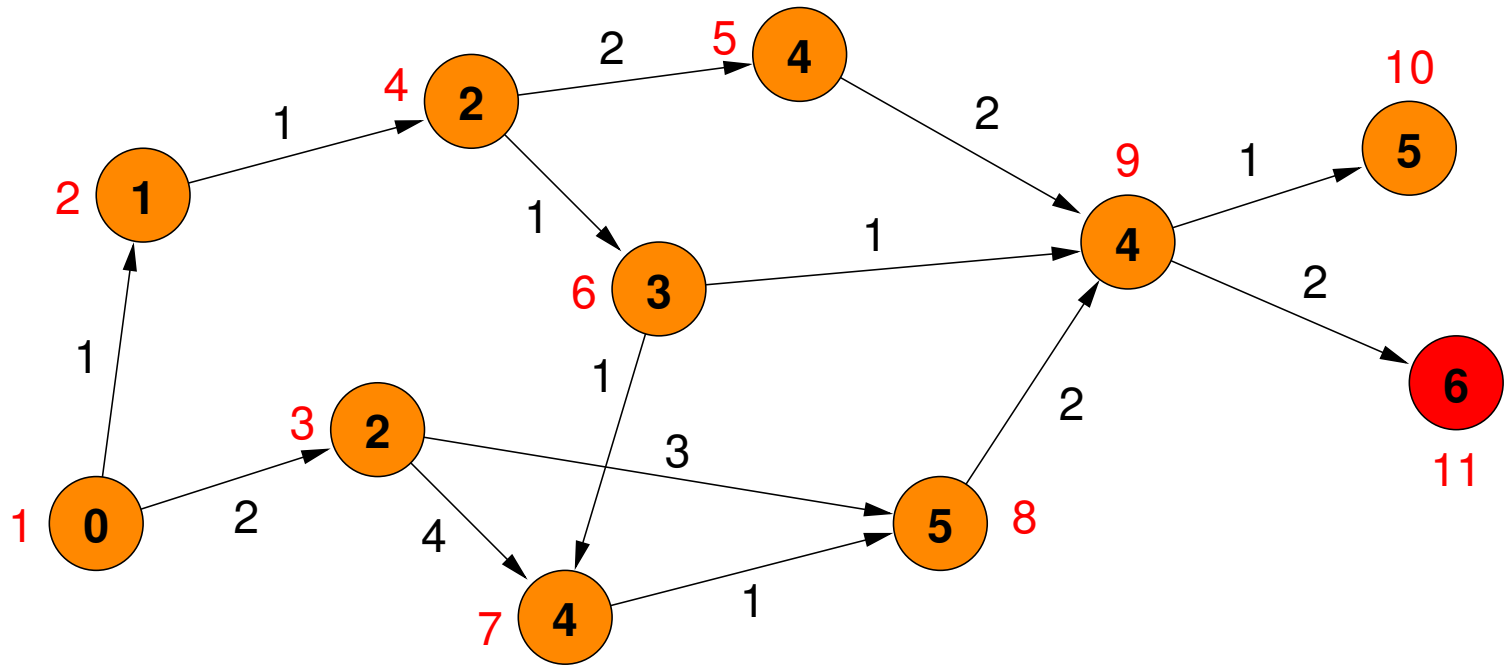


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Topologische Sortierung – warum funktioniert das?

- betrachte einen kürzesten Weg von s nach v
- der ganze Pfad beachtet die topologische Sortierung
- d.h., die Distanzen werden in der Reihenfolge der Knoten vom Anfang des Pfades zum Ende hin betrachtet
- damit ergibt sich für v der richtige Distanzwert

- ein Knoten x kann auch nie einen Wert erhalten, der echt kleiner als seine Distanz zu s ist
- die Kantenfolge von s zu x, die jeweils zu den Distanzwerten an den Knoten geführt hat, wäre dann ein kürzerer Pfad (Widerspruch)

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Allgemeine Strategie:

- Anfang: setze $d(s) = 0$ und
  für alle anderen Knoten v setze $d(v) = \infty$
- besuche Knoten in einer Reihenfolge, die sicherstellt, dass mindestens ein kürzester Weg von s zu jedem v in der Reihenfolge seiner Knoten besucht wird
- für jeden besuchten Knoten v aktualisiere die Distanzen der Knoten w mit $(v, w) \in E$, d.h. setze

$$d(w) = \min\{\, d(w),\ d(v) + c(v, w) \,\}$$

Kürzeste Wege in DAGs

Topologische Sortierung

- verwende FIFO-Queue q
- verwalte für jeden Knoten einen Zähler für die noch nicht markierten eingehenden Kanten
- initialisiere q mit allen Knoten, die keine eingehende Kante haben (Quellen)
- nimm nächsten Knoten v aus q und
  markiere alle $(v, w) \in E$, d.h. dekrementiere Zähler für w
- falls der Zähler von w dabei Null wird, füge w in q ein
- wiederhole das, bis q leer wird

Kürzeste Wege in DAGs

Topologische Sortierung

Korrektheit

- Knoten wird erst dann nummeriert, wenn alle Vorgänger nummeriert sind

Laufzeit

- für die Anfangswerte der Zähler muss der Graph einmal traversiert werden $O(n+m)$
- danach wird jede Kante genau einmal betrachtet

$\Rightarrow$ gesamt: $O(n+m)$

Test auf DAG-Eigenschaft

- topologische Sortierung erfasst genau dann alle Knoten, wenn der Graph ein DAG ist
- bei gerichteten Kreisen erhalten diese Knoten keine Nummer

Kürzeste Wege in DAGs

DAG-Strategie

1. Topologische Sortierung der Knoten

   Laufzeit $\mathcal{O}(n+m)$

2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

   Laufzeit $\mathcal{O}(n+m)$

Gesamtlaufzeit: $\mathcal{O}(n+m)$

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

- beliebiger Graph
  (gerichtet oder ungerichtet, muss diesmal kein DAG sein)
- mit nicht-negativen Kantengewichten

$\Rightarrow$ keine Knoten mit Distanz $-\infty$

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- wie bei Breitensuche, jedoch diesmal auch mit Distanzen $\neq 1$

Lösung:

- besuche Knoten in der Reihenfolge der kürzesten Distanz zum Startknoten s

Kürzeste Pfade: SSSP / Dijkstra

Dijkstra-Algorithmus

Input : $G = (V, E)$, $c : E \to \mathbb{R}$, $s \in V$
Output : Distanzen $d(s, v)$ zu allen $v \in V$

$P = \emptyset$; $T = V$;
$d(s, v) = \infty$ for all $v \in V \setminus s$;
$d(s, s) = 0$; $pred(s) = \perp$;
while $P \neq V$ **do**
  $v = \text{argmin}_{v \in T}\{d(s, v)\}$;
  $P = P \cup v$; $T = T \setminus v$;
  forall the $(v, w) \in E$ **do**
    if $d(s, w) > d(s, v) + c(v, w)$ **then**
      $d(s, w) = d(s, v) + c(v, w)$;
      $pred(w) = v$;

Dijkstra-Algorithmus für SSSP

Input : $G = (V, E)$, $c : E \to \mathbb{R}_{\geq 0}$, $s \in V$
Output : Distanzen $d[v]$ von s zu allen $v \in V$

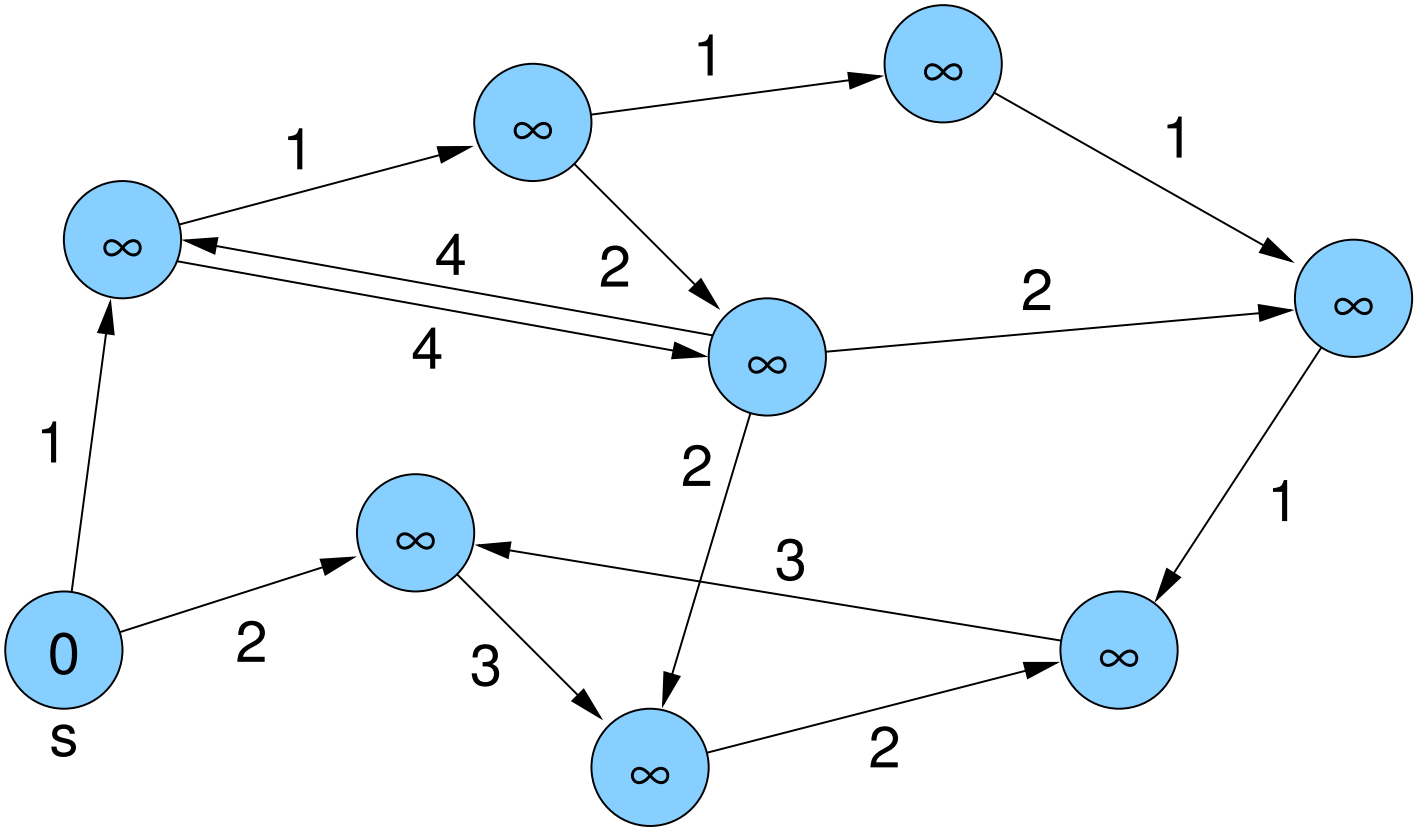
$d[v] = \infty$ for all $v \in V \setminus s$;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $\neg pq.empty()$ **do**
  $v = pq$.deleteMin();
  forall the $(v, w) \in E$ **do**
    newDist $= d[v] + c(v, w)$;
    if $newDist < d[w]$ **then**
      $pred[w] = v$;
      if $d[w] == \infty$ **then** pq.insert$(w,$ newDist$)$;
      else pq.decreaseKey$(w,$ newDist$)$;
      $d[w =$ newDist;

Dijkstra-Algorithmus

- setze Startwert $d(s, s) = 0$ und zunächst $d(s, v) = \infty$
- verwende Prioritätswarteschlange, um die Knoten zusammen mit ihren aktuellen Distanzen zuspeichern
- am Anfang nur Startknoten (mit Distanz 0) in Priority Queue
- dann immer nächsten Knoten v (mit kleinster Distanz) entnehmen, endgültige Distanz dieses Knotens v steht nun fest
- betrachte alle Nachbarn von v,
  füge sie ggf. in die PQ ein bzw.
  aktualisiere deren Priorität in der PQ

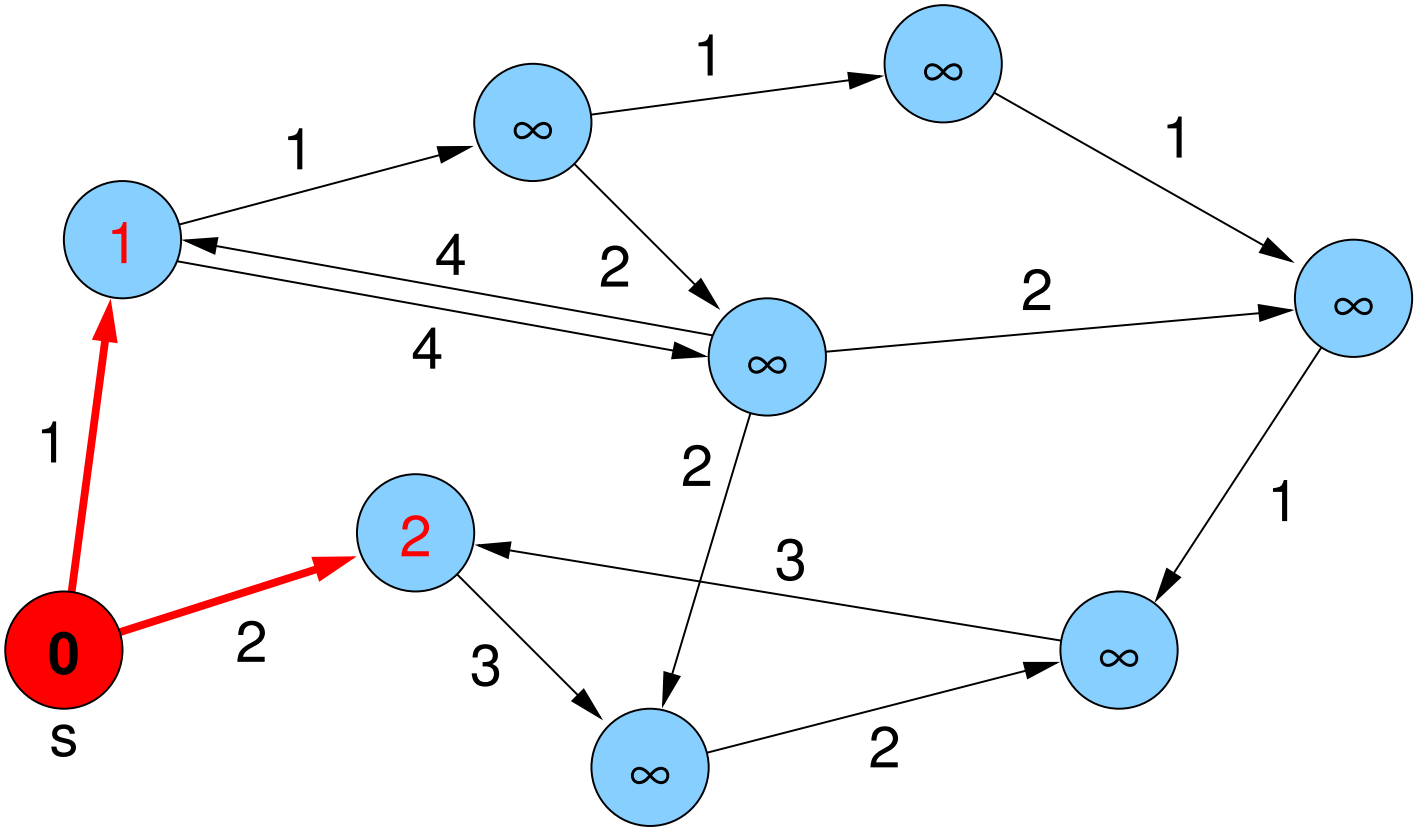
Dijkstra-Algorithmus

Beispiel:



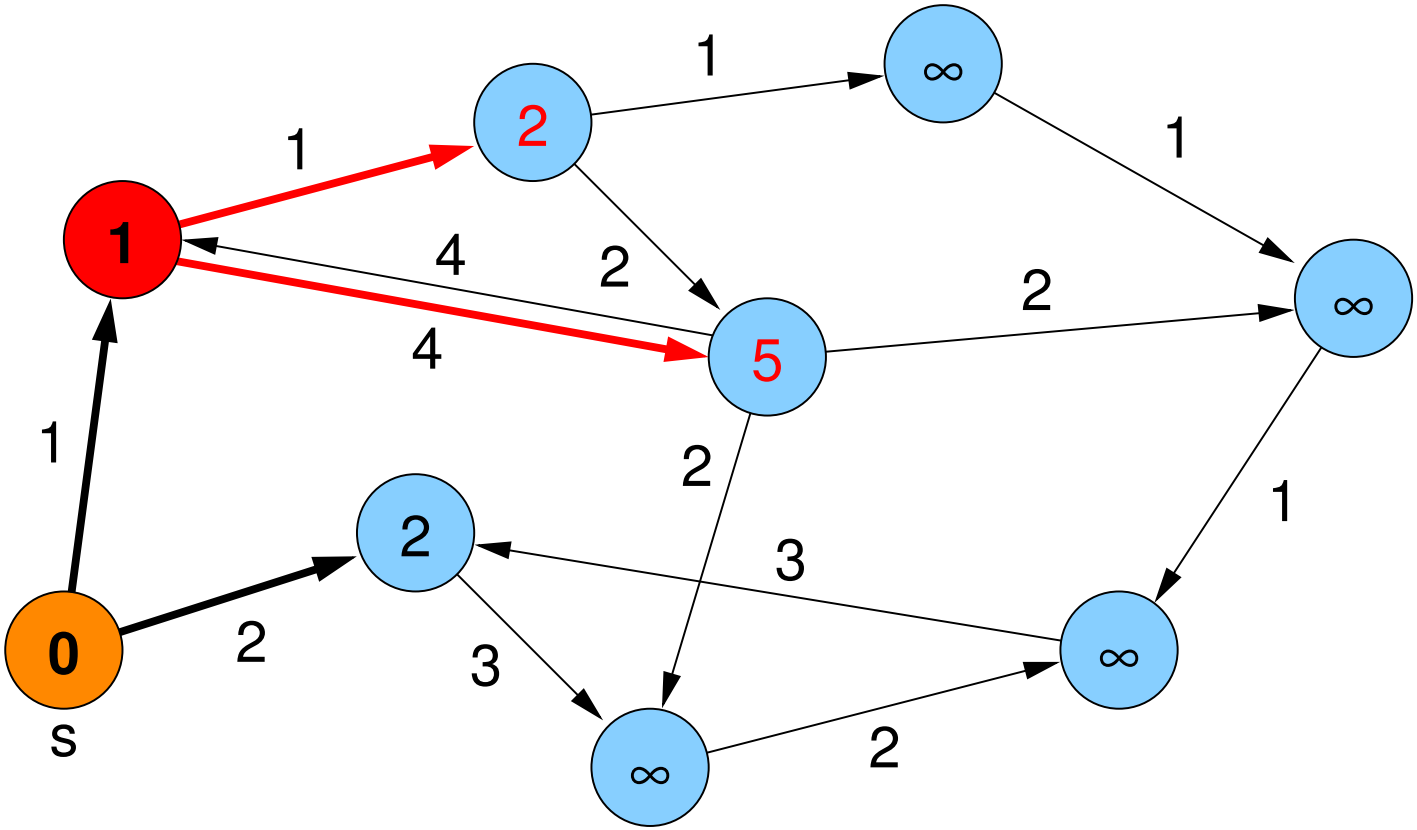
Dijkstra-Algorithmus

Beispiel:



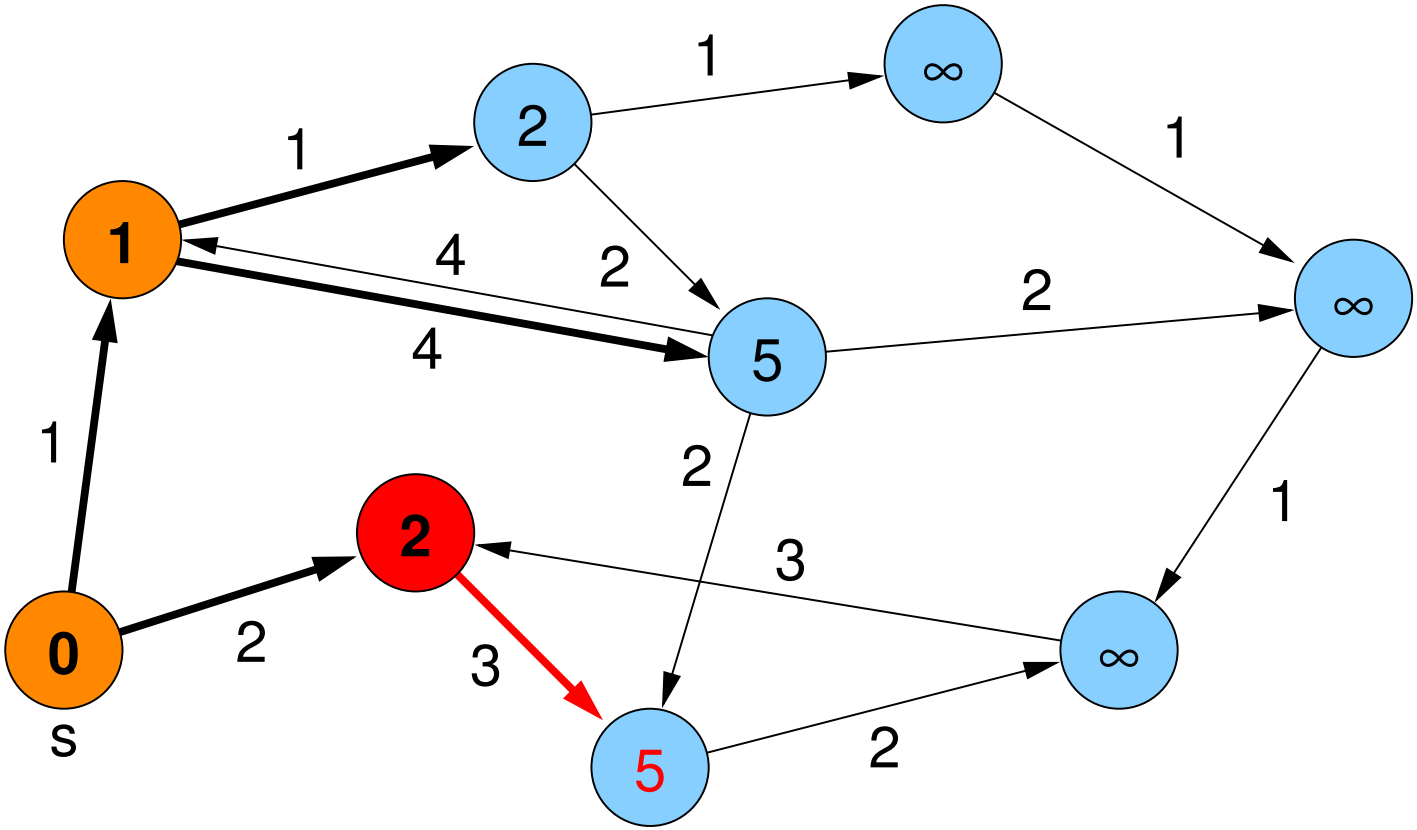
Dijkstra-Algorithmus

Beispiel:



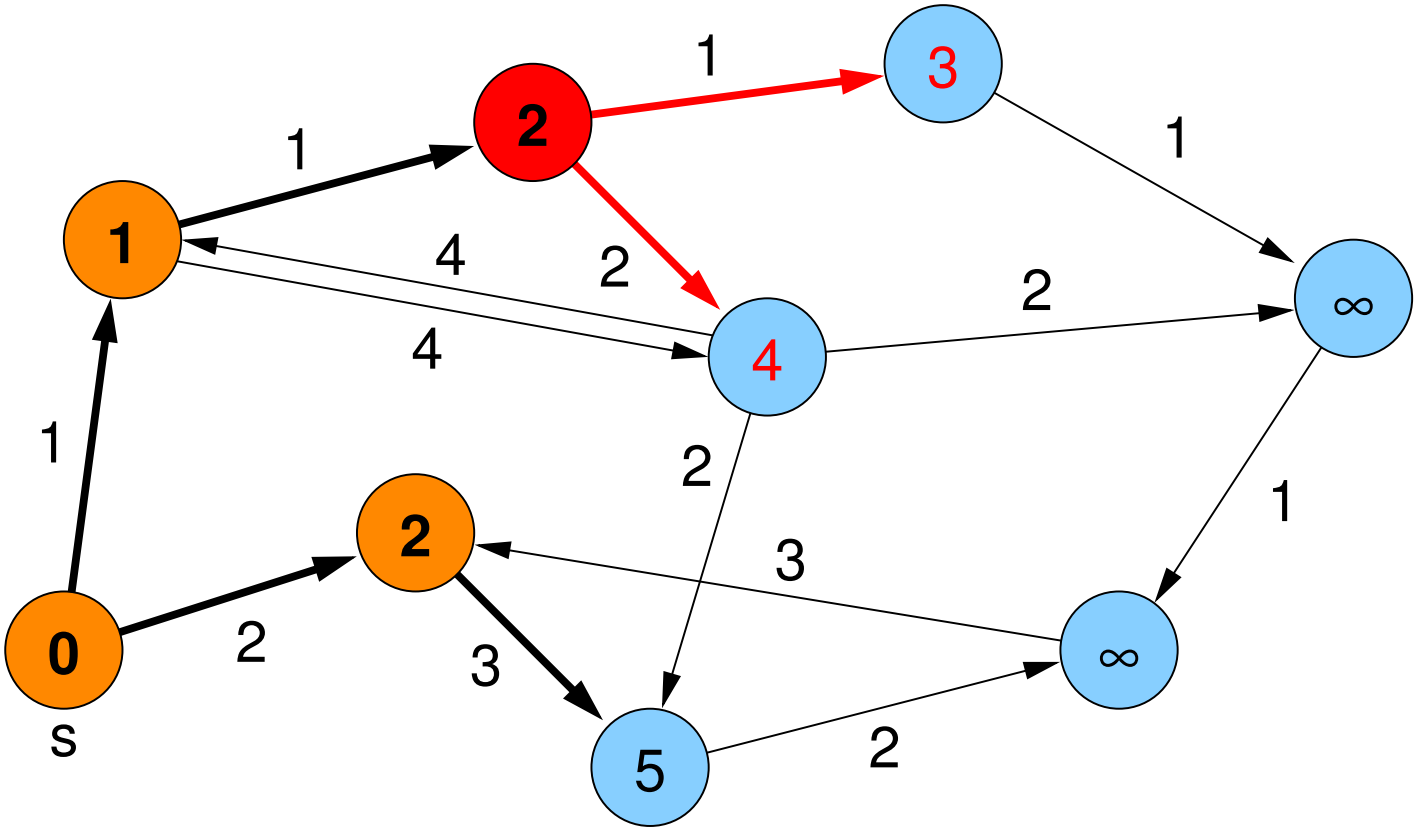
Dijkstra-Algorithmus

Beispiel:



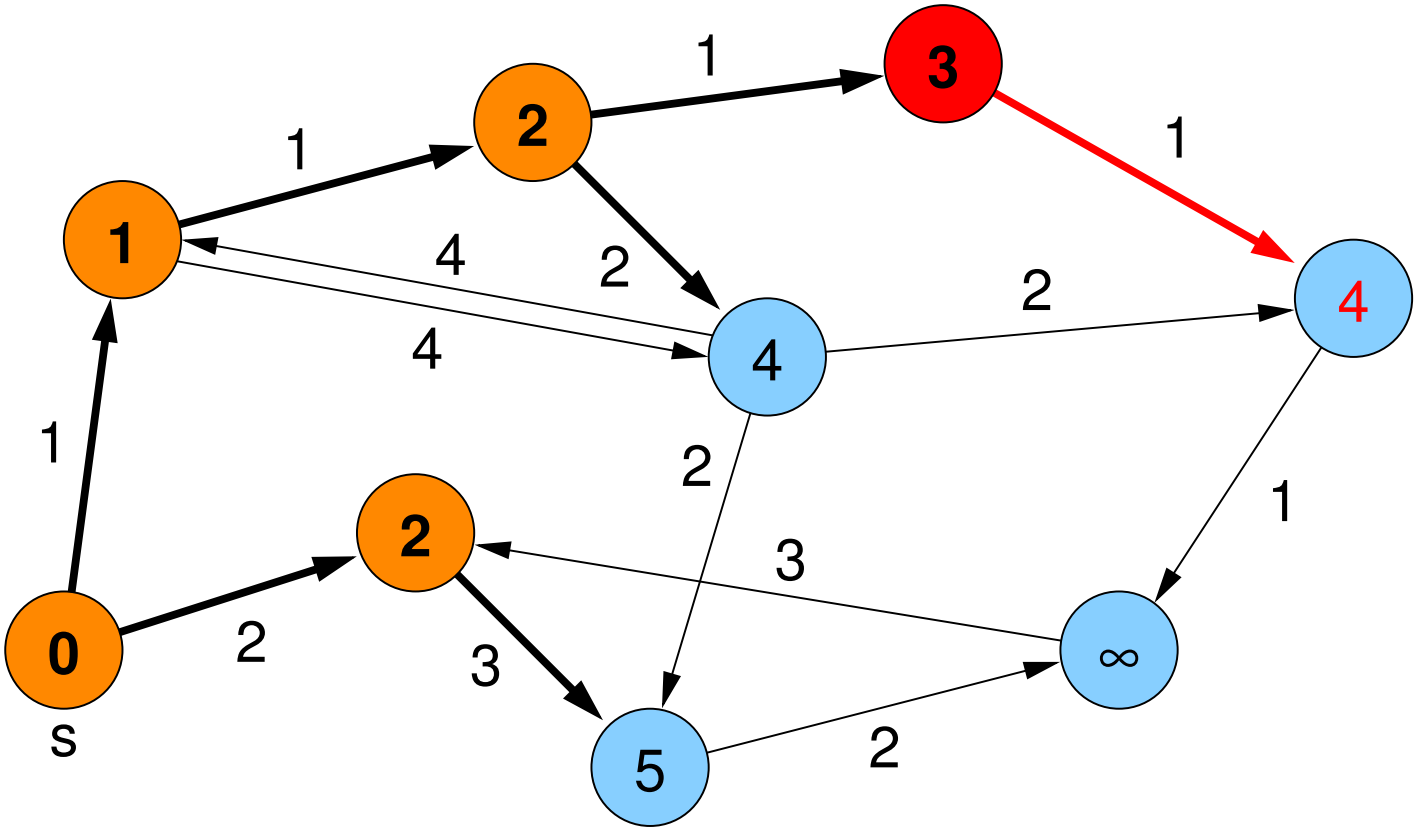
Dijkstra-Algorithmus

Beispiel:



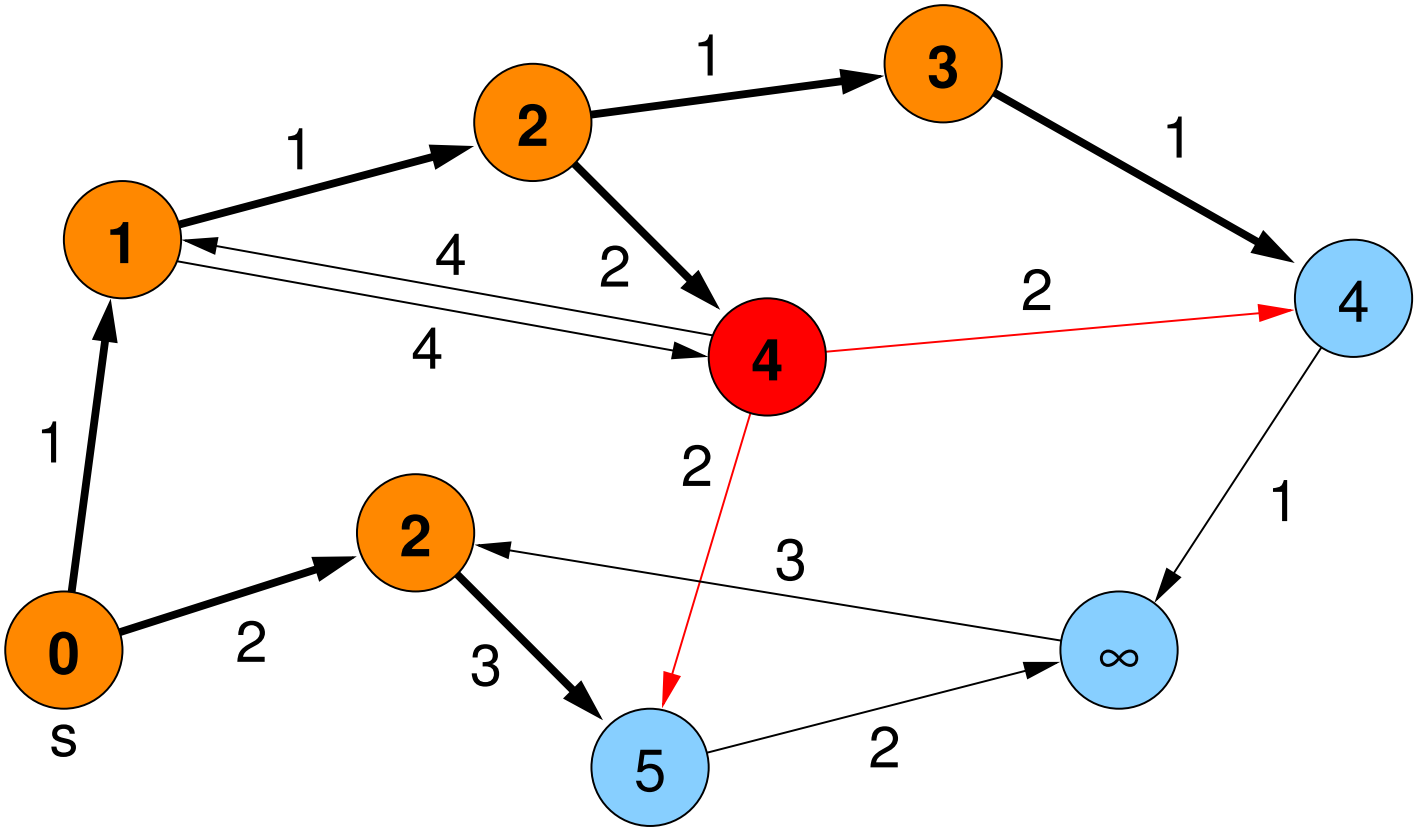
Dijkstra-Algorithmus

Beispiel:



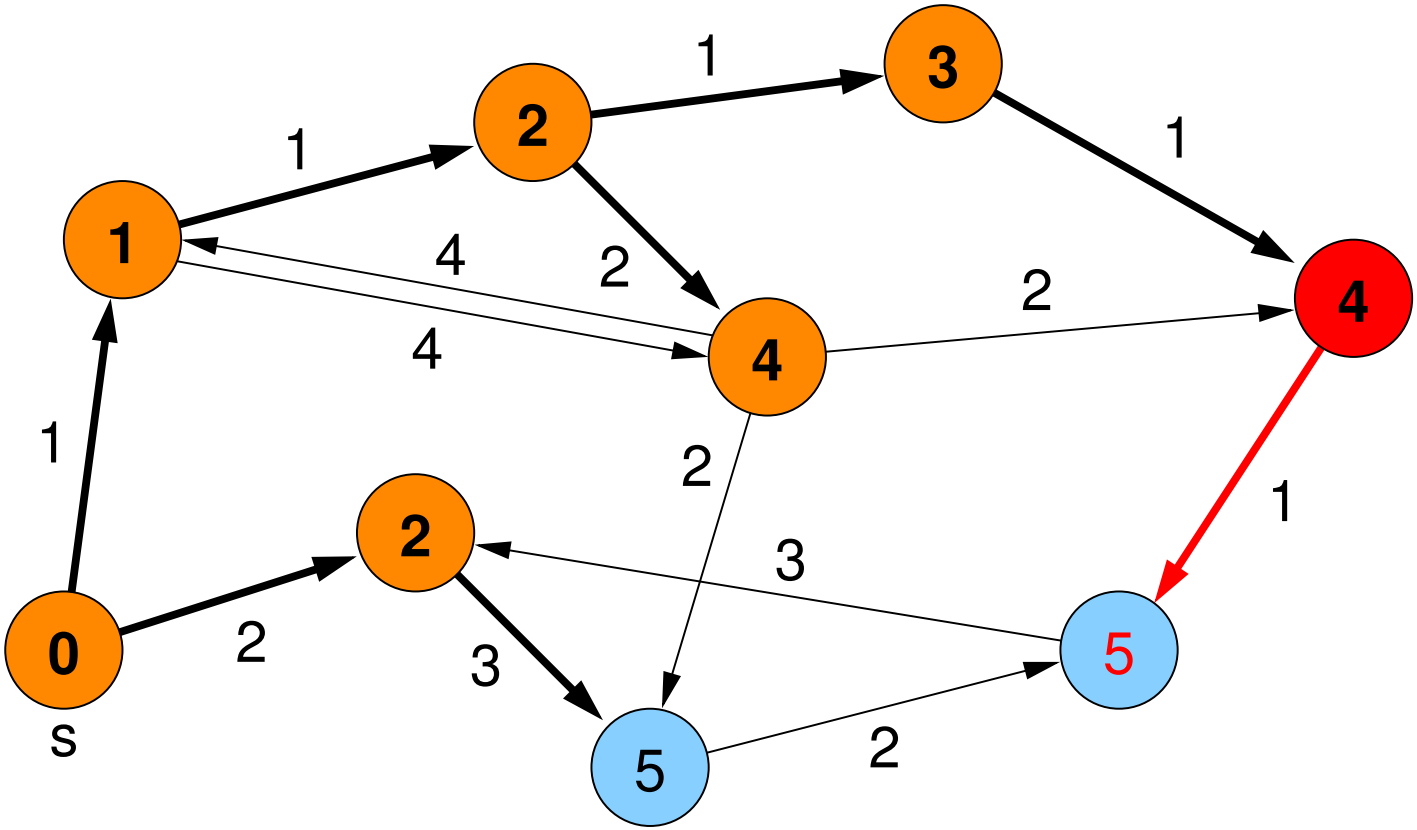
Dijkstra-Algorithmus

Beispiel:



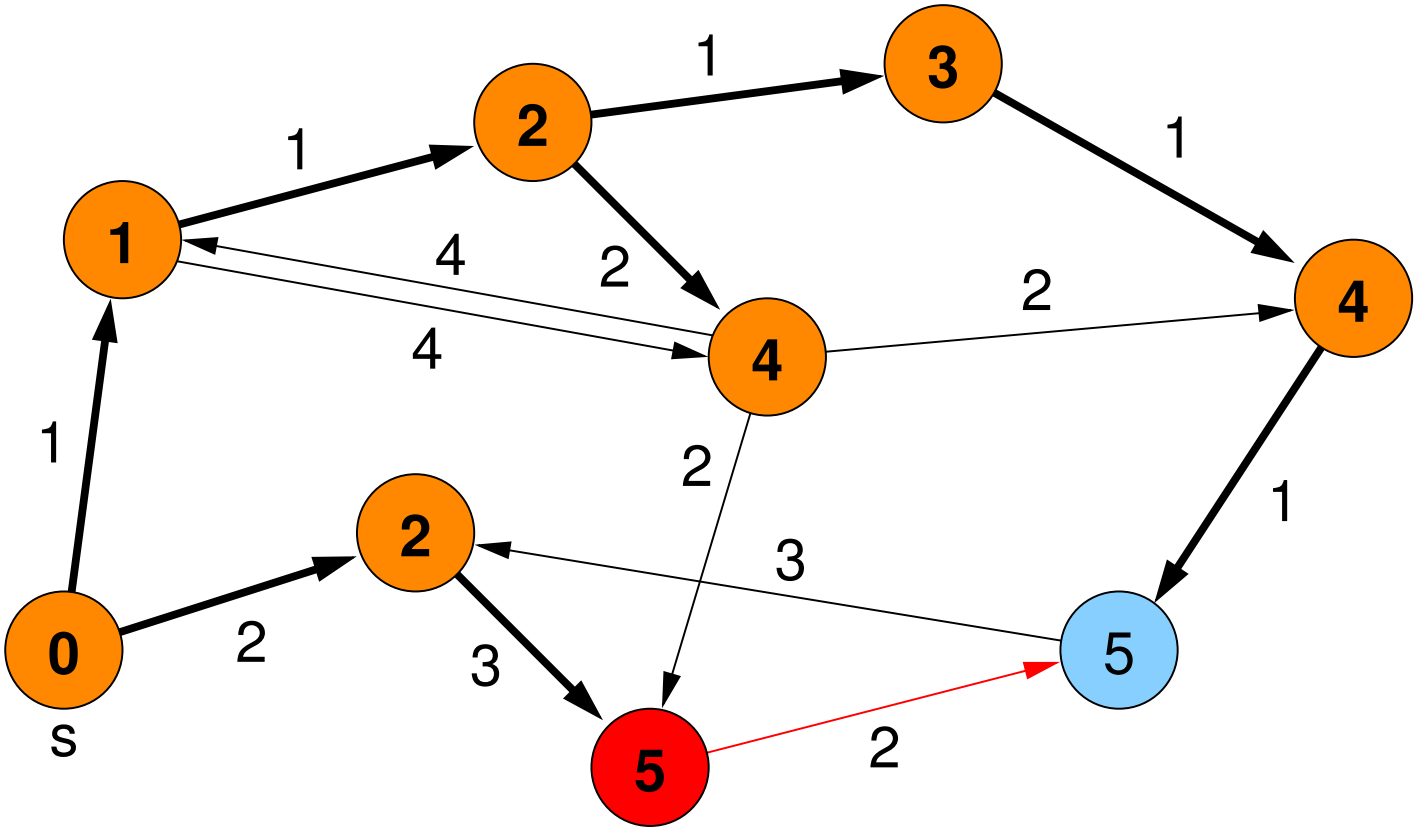
Dijkstra-Algorithmus

Beispiel:



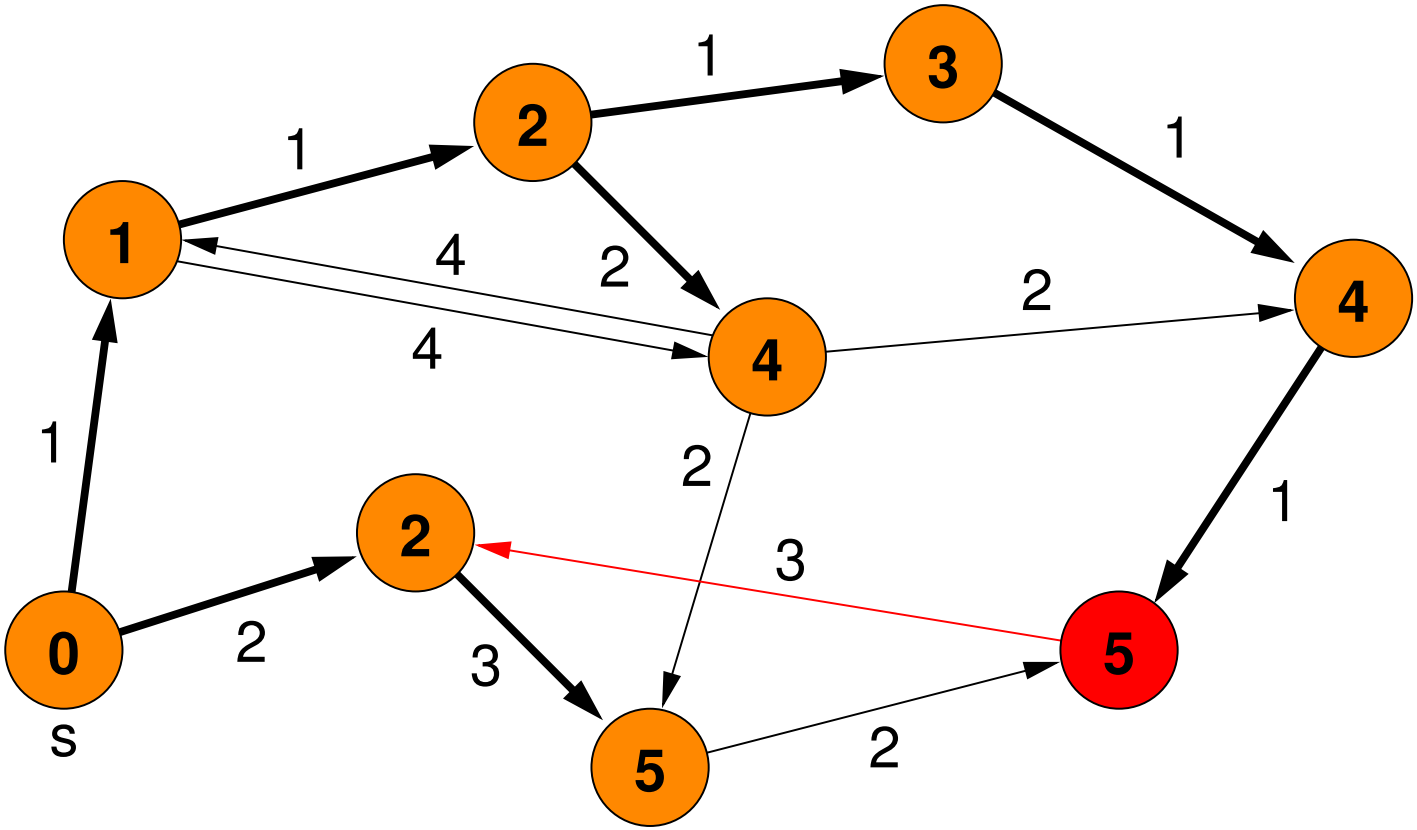
Dijkstra-Algorithmus

Beispiel:



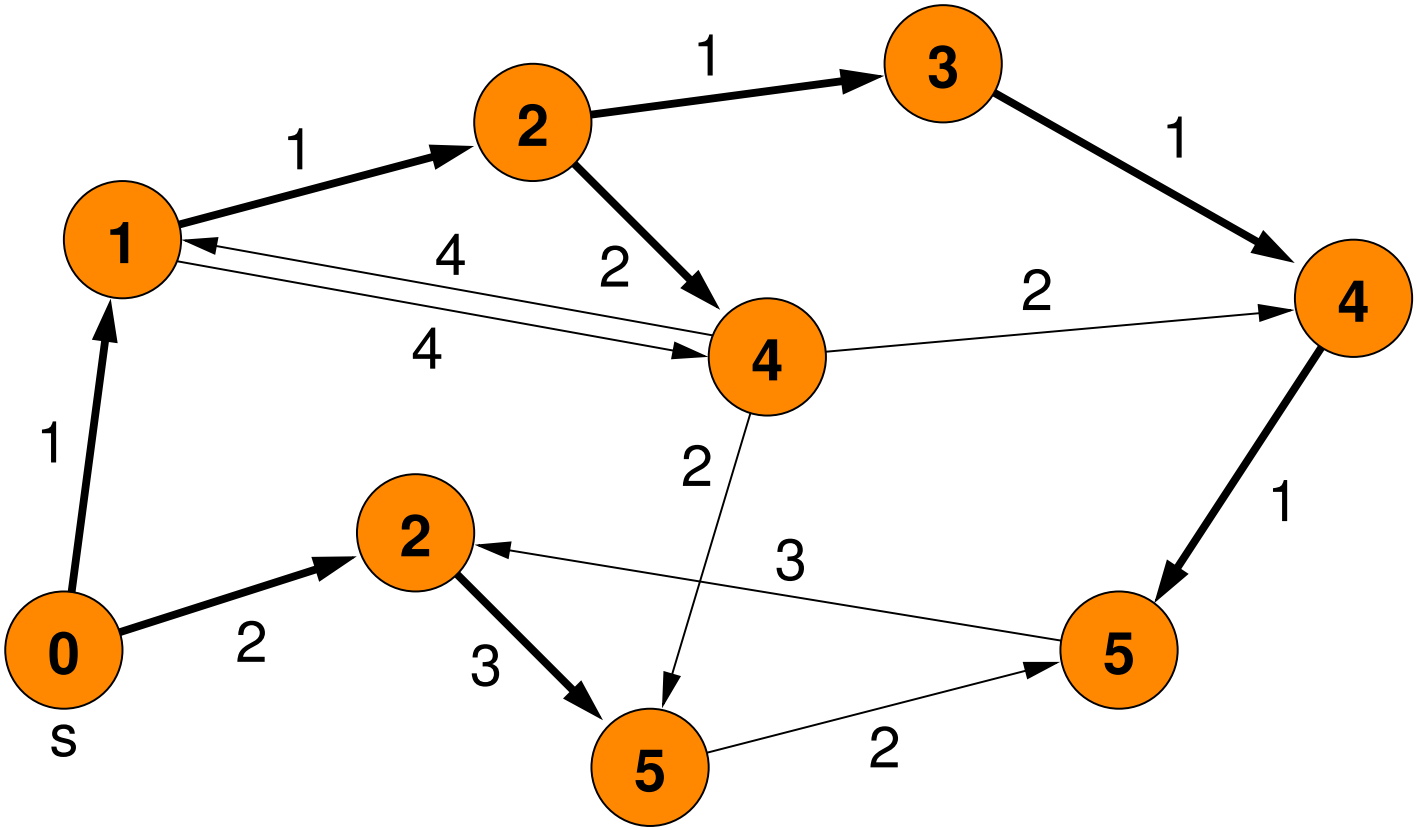
Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Korrektheit:

- Annahme: Algorithmus liefert für w einen zu kleinen Wert $d(s, w)$
- sei w der erste Knoten, für den die Distanz falsch festgelegt wird (kann nicht s sein, denn die Distanz $d(s, s)$ bleibt immer 0)
- kann nicht sein, weil $d(s, w)$ nur dann aktualisiert wird, wenn man über einen von s schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann
- d.h. $d(s, v)$ müsste schon falsch gewesen sein (Widerspruch zur Annahme, dass w der erste Knoten mit falscher Distanz war)

Dijkstra-Algorithmus

- Annahme: Algorithmus liefert für w einen zu großen Wert $d(s, w)$
- sei w der Knoten mit der kleinsten (wirklichen) Distanz, für den der Wert $d(s, w)$ falsch festgelegt wird (wenn es davon mehrere gibt, der Knoten, für den die Distanz zuletzt festgelegt wird)
- kann nicht sein, weil $d(s, w)$ immer aktualisiert wird, wenn man über einen von s schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann (dabei steht $d(s, v)$ immer schon fest, so dass auch die Länge eines kürzesten Wegs über v zu w richtig berechnet wird)
- d.h., entweder wurde auch der Wert von v falsch berechnet (Widerspruch zur Def. von w) oder die Distanz von v wurde noch nicht festgesetzt
- weil die berechneten Distanzwerte monoton wachsen, kann letzteres nur passieren, wenn v die gleiche Distanz hat wie w (auch Widerspruch zur Def. von w)

Dijkstra-Algorithmus

- Datenstruktur: Prioritätswarteschlange
  (z.B. Fibonacci Heap: amortisierte Komplexität $\mathcal{O}(1)$ für `insert` und `decreaseKey`, $\mathcal{O}(\log n)$ `deleteMin`)

- Komplexität:
  - $n \times \mathcal{O}(1)$ `insert`
  - $n \times \mathcal{O}(\log n)$ `deleteMin`
  - $m \times \mathcal{O}(1)$ `decreaseKey`

  $\Rightarrow \mathcal{O}(m + n \log n)$

- aber: nur für nichtnegative Kantengewichte(!)

Monotone Priority Queues

Beobachtung:

- aktuelles Distanz-Minimum der verbleibenden Knoten ist beim Dijkstra-Algorithmus monoton wachsend

Monotone Priority Queue

- Folge der entnommenen Elemente hat monoton steigende Werte
- effizientere Implementierung möglich, falls Kantengewichte ganzzahlig

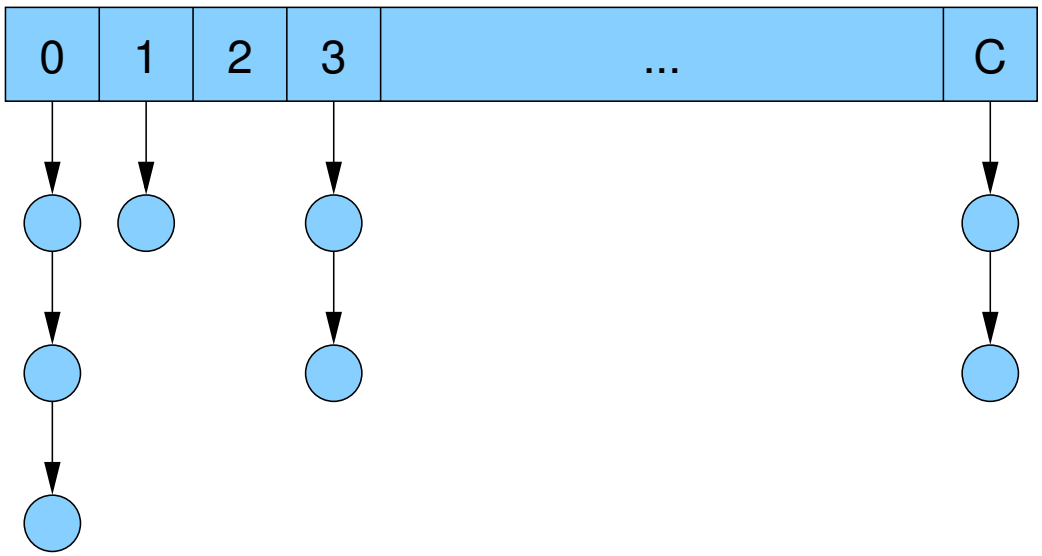
Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

$\Rightarrow$ enthaltene Distanzwerte immer im Bereich $[d, d + C]$

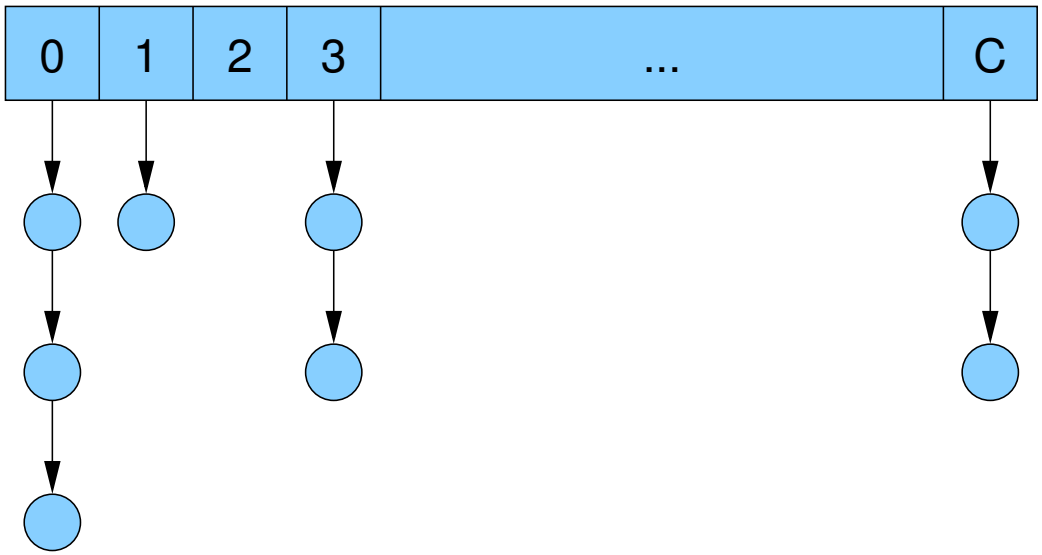
Bucket Queue

- Array B aus $C + 1$ Listen
- Variable d_{min} für aktuelles Distanzminimum mod$(C + 1)$



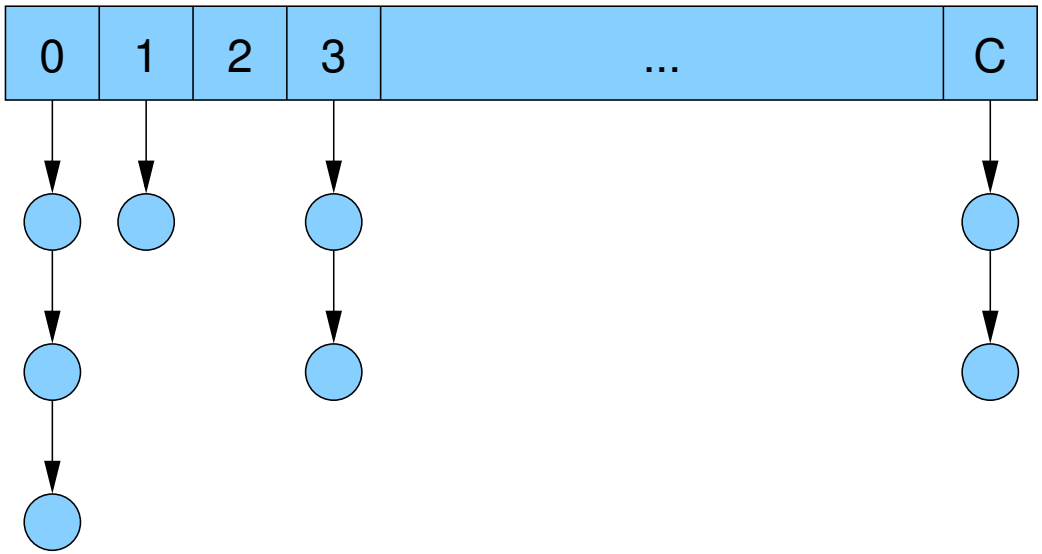
Bucket Queue

- jeder Knoten v mit aktueller Distanz $d[v]$ in Liste $B[d[v] \bmod (C+1)]$
- alle Knoten in Liste $B[d]$ haben dieselbe Distanz, weil alle aktuellen Distanzen im Bereich $[d, d+C]$ liegen



Bucket Queue / Operationen

- insert(v): fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($\mathcal{O}(1)$)
- decreaseKey(v): entfernt v aus momentaner Liste ($\mathcal{O}(1)$ falls Handle auf Listenelement in v gespeichert) und fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($\mathcal{O}(1)$)
- deleteMin(): solange $B[d_{\min}] = \emptyset$, setze $d_{\min} = (d_{\min}+1) \bmod (C+1)$.
  Nimm dann einen Knoten u aus $B[d_{\min}]$ heraus ($\mathcal{O}(C)$)



Dijkstra mit Bucket Queue

- insert, decreaseKey: $\mathcal{O}(1)$
- deleteMin: $\mathcal{O}(C)$
- Dijkstra: $\mathcal{O}(m + C \cdot n)$

- lässt sich mit Radix Heaps noch verbessern
- verwendet exponentiell wachsende Bucket-Größen
- Details in der Vorlesung Effiziente Algorithmen und Datenstrukturen
- Laufzeit ist dann $\mathcal{O}(m + n \log C)$

Beliebige Graphen mit beliebigen Gewichten

Gegeben:

- beliebiger Graph mit beliebigen Kantengewichten

$\Rightarrow$ Anhängen einer Kante an einen Weg kann zur Verkürzung des Weges (Kantengewichtssumme) führen (wenn Kante negatives Gewicht hat)

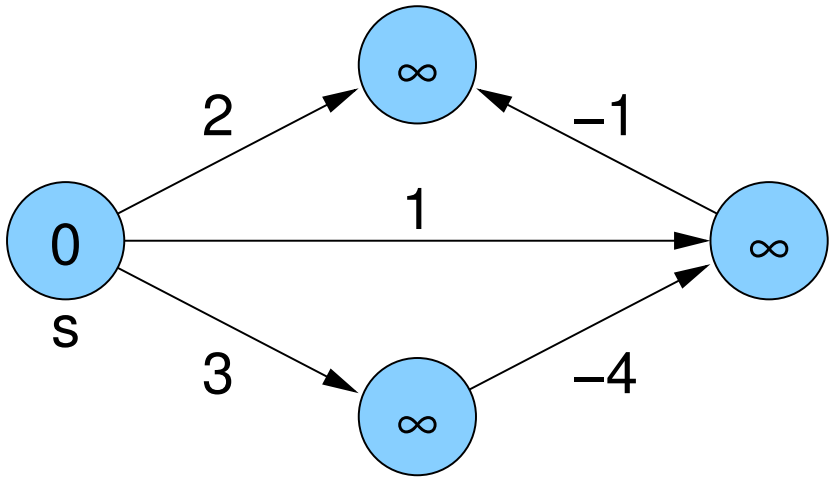
$\Rightarrow$ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- Dijkstra kann nicht mehr verwendet werden, weil Knoten nicht unbedingt in der Reihenfolge der kürzesten Distanz zum Startknoten s besucht werden

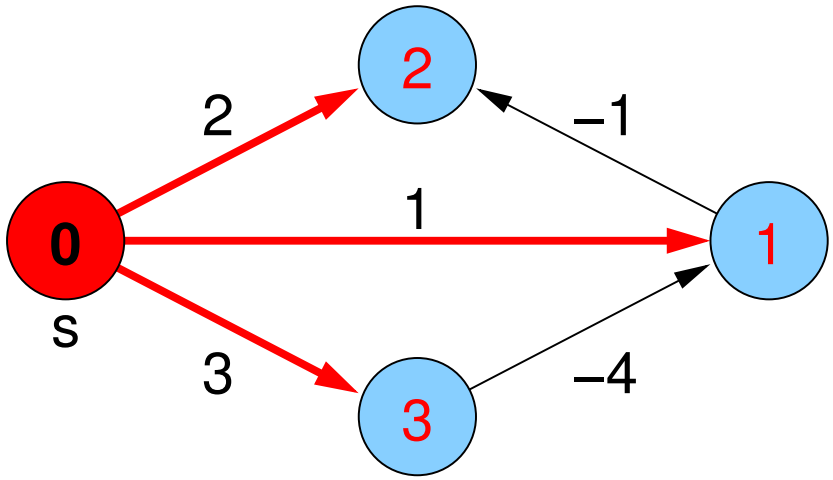
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



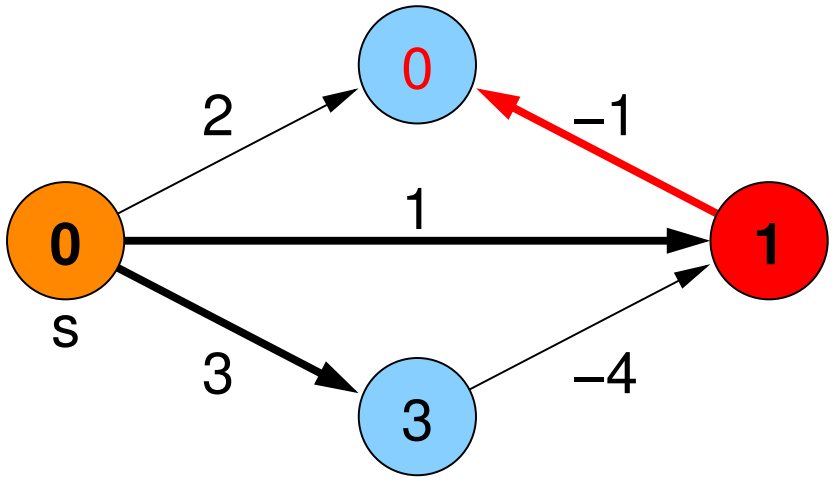
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



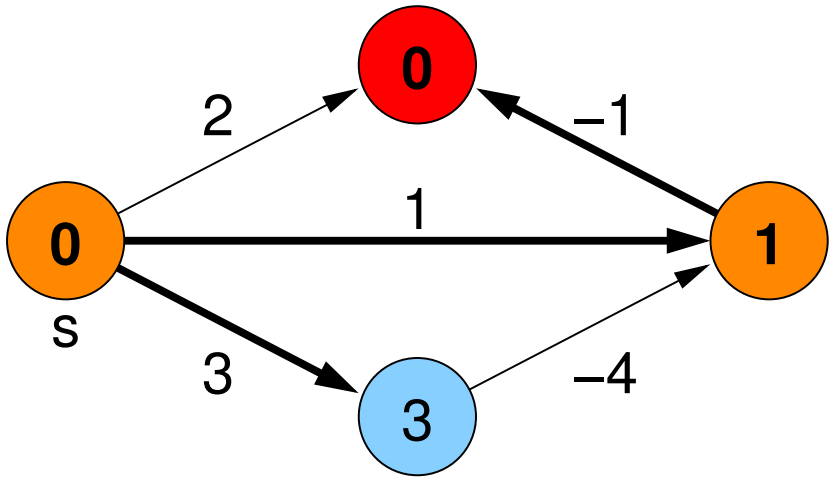
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



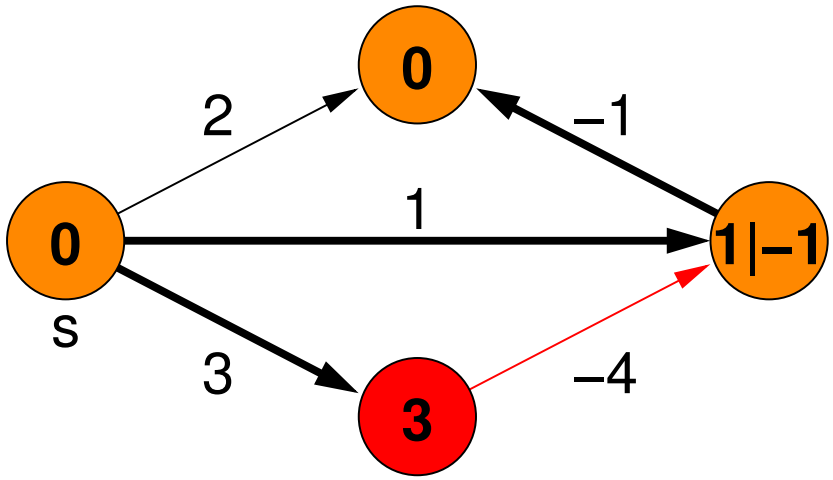
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



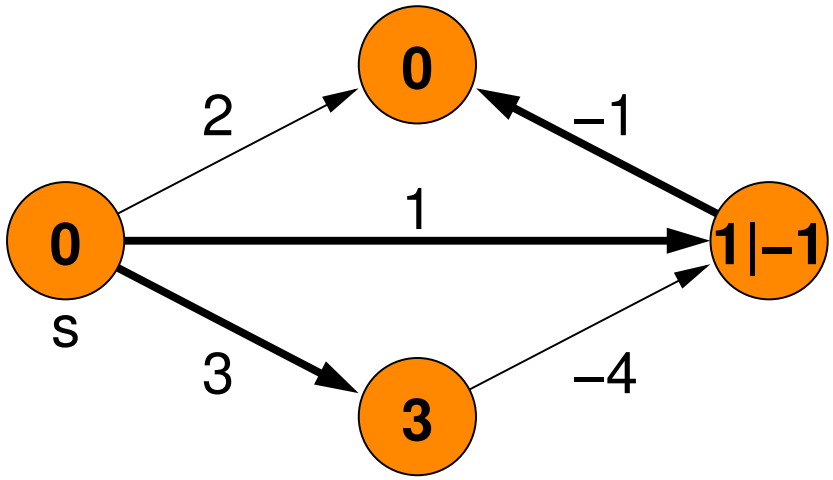
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



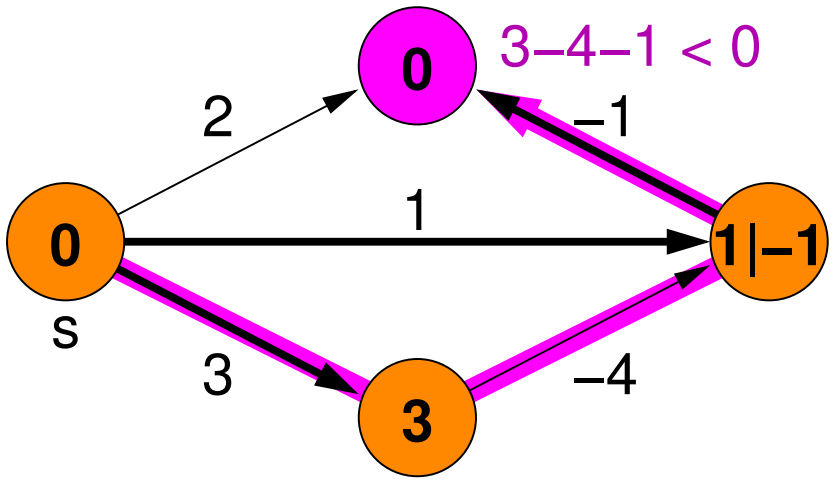
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Lemma

Für jeden von s erreichbaren Knoten v mit $d(s,v) > -\infty$ gibt es einen einfachen Pfad (ohne Kreis) von s nach v der Länge $d(s,v)$.

Beweis.

Betrachte kürzesten Weg mit Kreis(en):

- Kreis mit Kantengewichtssumme > 0 nicht enthalten:
  Entfernen des Kreises würde Kosten verringern
- Kreis mit Kantengewichtssumme $= 0$:
  Entfernen des Kreises lässt Kosten unverändert
- Kreis mit Kantengewichtssumme < 0:
  Distanz von s ist $-\infty$

□

Bellman-Ford-Algorithmus

Folgerung

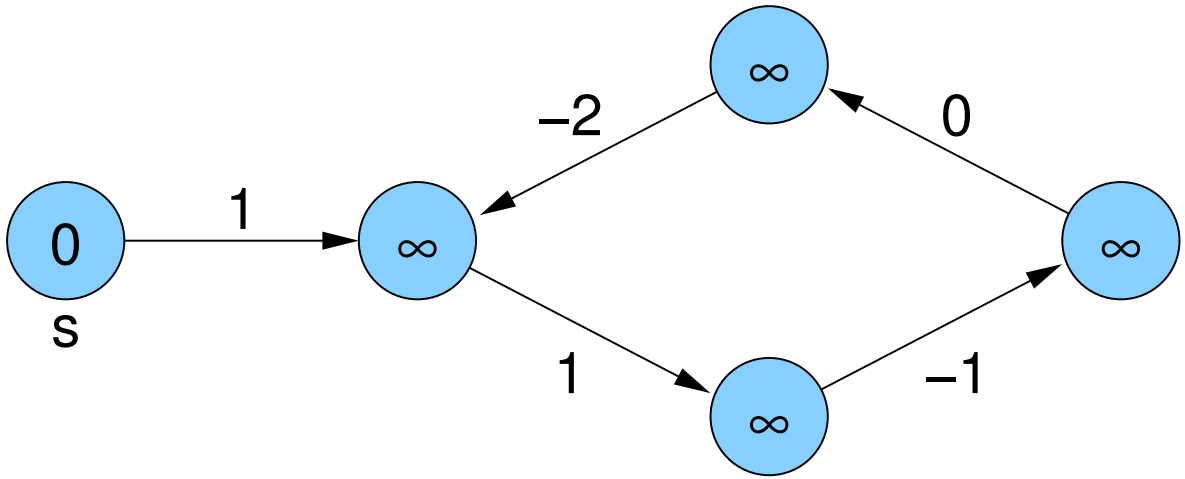
In einem Graph mit n Knoten gibt es für jeden erreichbaren Knoten v mit $d(s,v) > -\infty$ einen kürzesten Weg bestehend aus $< n$ Kanten zwischen s und v.

Strategie:

- anstatt kürzeste Pfade in Reihenfolge wachsender Gewichtssumme zu berechnen, betrachte sie in Reihenfolge steigender Kantenanzahl
- durchlaufe (n-1)-mal alle Kanten im Graph und aktualisiere die Distanz
- dann alle kürzesten Wege berücksichtigt

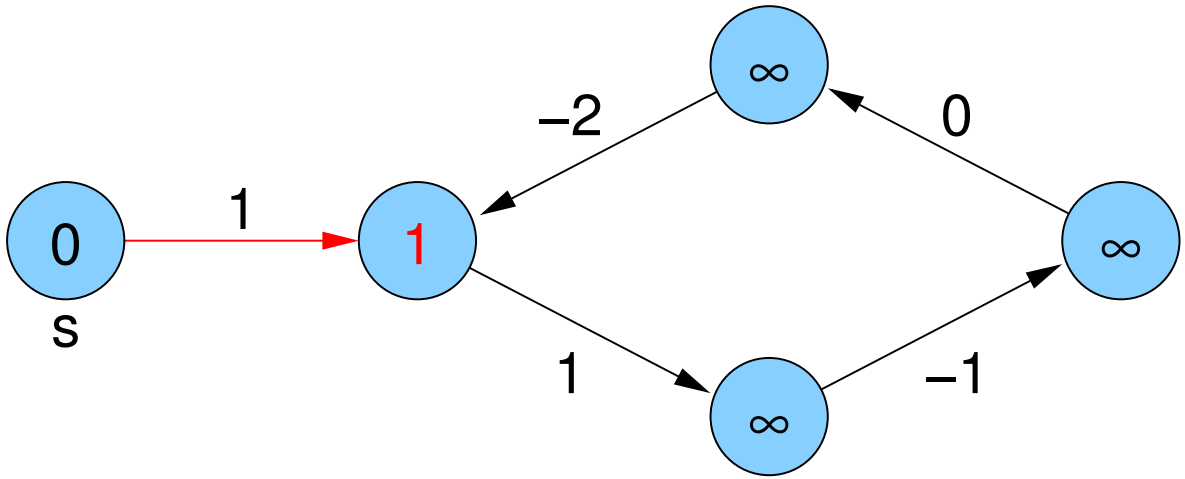
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



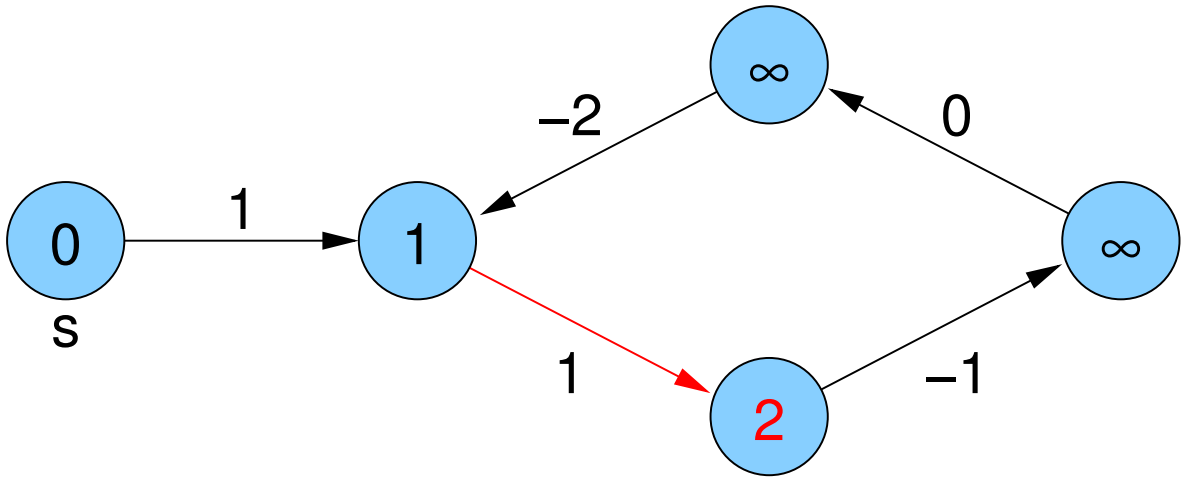
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



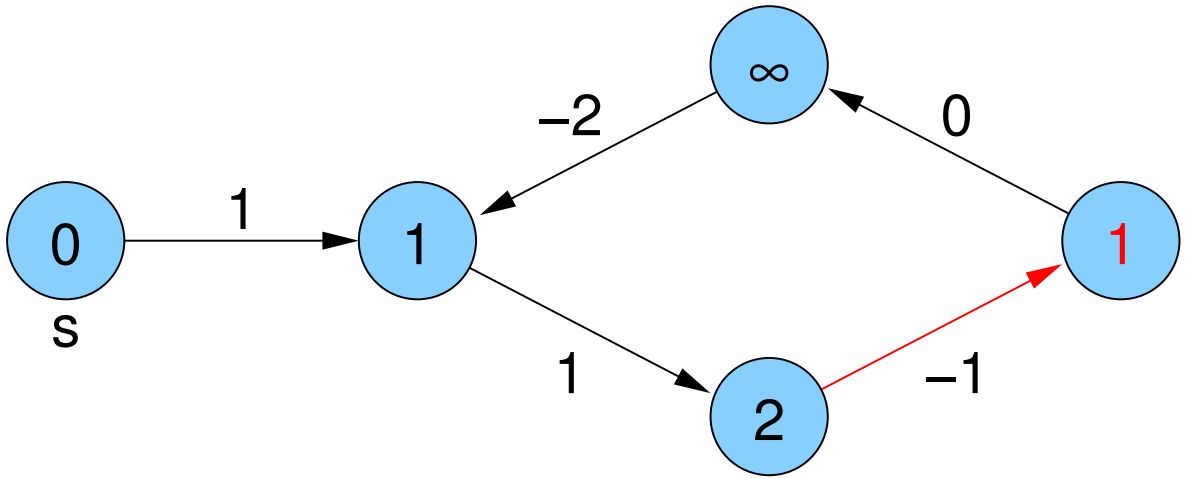
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



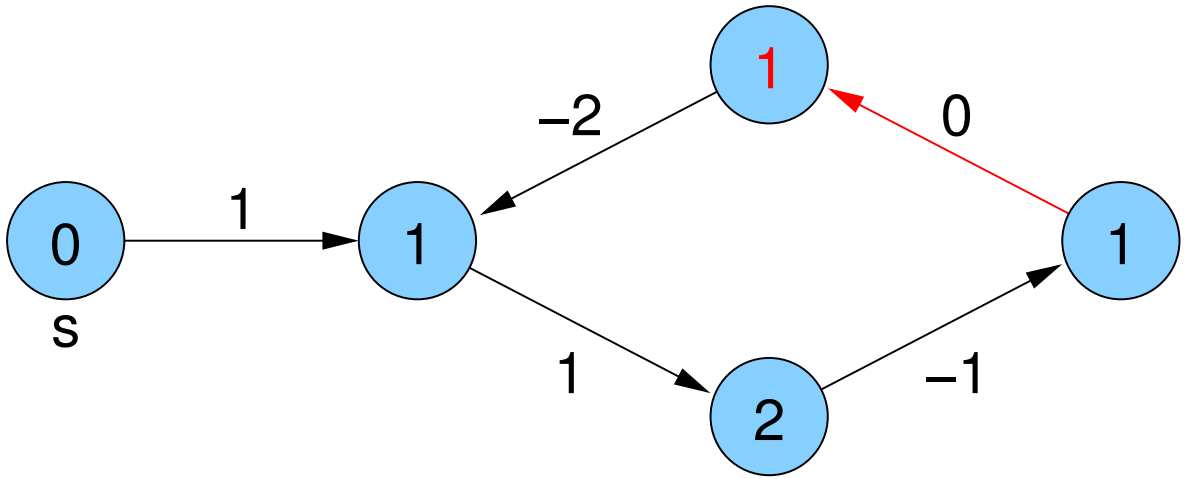
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



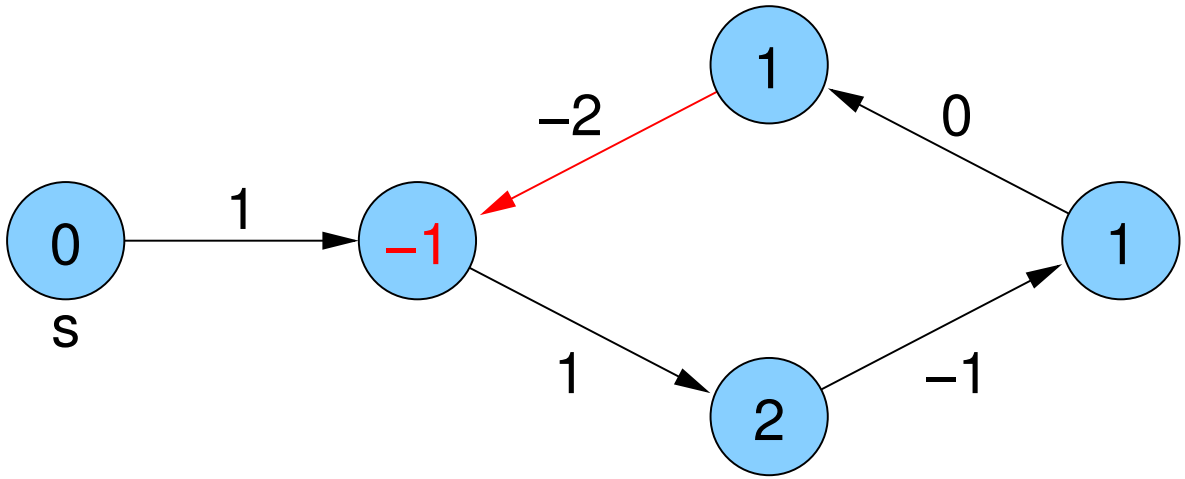
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



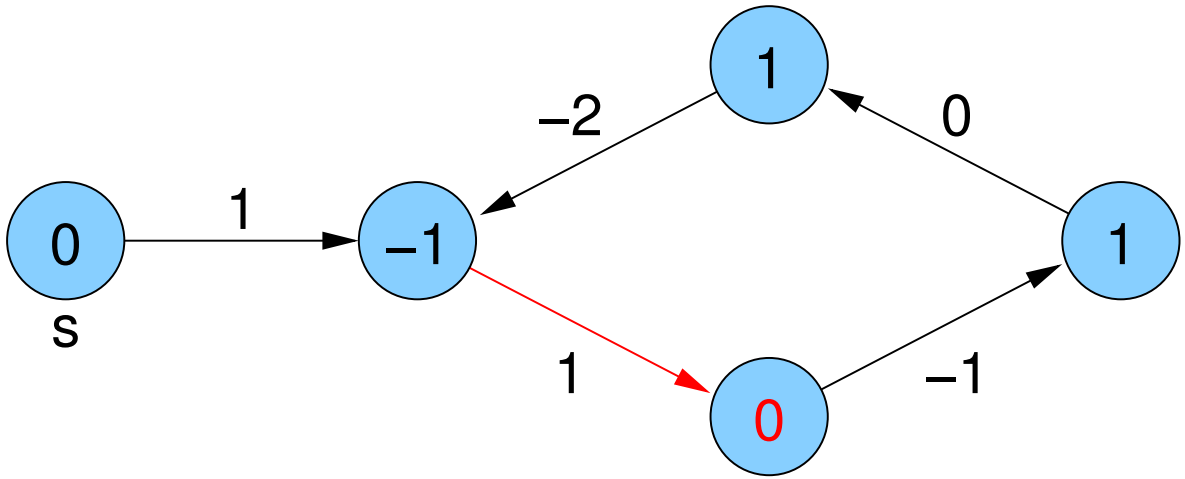
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Keine Distanzverringerung mehr möglich:

- Annahme: zu einem Zeitpunkt gilt für alle Kanten (v, w) $d[v] + c(v, w) \geq d[w]$

$\Rightarrow$ (per Induktion) für alle Knoten w und jeden Weg p von s nach w gilt: $d[s] + c(p) \geq d[w]$

- falls sichergestellt, dass zu jedem Zeitpunkt für kürzesten Weg p von s nach w gilt $d[w] \geq c(p)$, dann ist $d[w]$ zum Schluss genau die Länge eines kürzesten Pfades von s nach w (also korrekte Distanz)

Bellman-Ford-Algorithmus

Zusammenfassung:

- keine Distanzerniedrigung mehr möglich
  ($d[v] + c(v,w) \geq d[w]$ für alle w):
  fertig, alle $d[w]$ korrekt für alle w

- Distanzerniedrigung möglich selbst noch in n-ter Runde
  ($d[v] + c(v,w) < d[w]$ für ein w):
  Es gibt einen negativen Kreis, also Knoten w mit Distanz $-\infty$.

Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[s] = s;
  for (int i = 0; i < n − 1; i++) {   // n − 1 Runden
    foreach (e = (v, w) ∈ E)
      if (d[v] + c(e) < d[w]) {   // kürzerer Weg?
        d[w] = d[v] + c(e);
        parent[w] = v;
      }
  }
  foreach (e = (v, w) ∈ E)
    if (d[v] + c(e) < d[w]) {   // kürzerer Weg in n-ter Runde?
      infect(w);
}
```

Bellman-Ford-Algorithmus

```
infect(Node v) {   // −∞-Knoten
  if (d[v] > −∞) {
    d[v] = −∞;
    foreach (e = (v, w) ∈ E)
      infect(w);
  }
}
```

Gesamtlaufzeit: $\mathcal{O}(m \cdot n)$

Bellman-Ford-Algorithmus

Bestimmung der Knoten mit Distanz $-\infty$:

- betrachte alle Knoten, die in der n-ten Phase noch Distanzverbesserung erfahren
- aus jedem Kreis mit negativem Gesamtgewicht muss mindestens ein Knoten dabei sein
- jeder von diesen Knoten aus erreichbare Knoten muss Distanz $-\infty$ bekommen
- das erledigt hier die infect-Funktion
- wenn ein Knoten zweimal auftritt (d.h. der Wert ist schon $-\infty$), wird die Rekursion abgebrochen

Bellman-Ford-Algorithmus

Bestimmung eines negativen Zyklus:

- bei den oben genannten Knoten sind vielleicht auch Knoten, die nur an negativen Kreisen über ausgehende Kanten angeschlossen sind, die selbst aber nicht Teil eines negativen Kreises sind
- Rückwärtsverfolgung der parent-Werte, bis sich ein Knoten wiederholt
- Kanten vom ersten bis zum zweiten Auftreten bilden einen negativen Zyklus

Bellman-Ford-Algorithmus

Ursprüngliche Idee der Updates vorläufiger Distanzwerte stammt von Lester R. Ford Jr.

Verbesserung (Richard E. Bellman / Edward F. Moore):

- verwalte FIFO-Queue von Knoten, zu denen ein kürzerer Pfad gefunden wurde und deren Nachbarn am anderen Ende ausgehender Kanten noch auf kürzere Wege geprüft werden müssen
- wiederhole: nimm ersten Knoten aus der Queue und prüfe für jede ausgehende Kante die Distanz des Nachbarn

  falls kürzerer Weg gefunden, aktualisiere Distanzwert des Nachbarn und hänge ihn an Queue an (falls nicht schon enthalten)
- Phase besteht immer aus Bearbeitung der Knoten, die am Anfang des Algorithmus (bzw. der Phase) in der Queue sind (dabei kommen während der Phase schon neue Knoten ans Ende der Queue)

Kürzeste einfache Pfade bei beliebigen Kantengewichten

Achtung!

Fakt

Die Suche nach kürzesten einfachen Pfaden
(also ohne Knotenwiederholungen / Kreise)
in Graphen mit beliebigen Kantengewichten
(also möglichen negativen Kreisen)
ist ein $\mathcal{NP}$-vollständiges Problem.

(Man könnte Hamilton-Pfad-Suche damit lösen.)

All Pairs Shortest Paths (APSP)

gegeben:

- Graph mit beliebigen Kantengewichten,
  der aber keine negativen Kreise enthält

gesucht:

- Distanzen / kürzeste Pfade zwischen allen Knotenpaaren

Naive Strategie:

- n-mal Bellman-Ford-Algorithmus
  (jeder Knoten einmal als Startknoten)

$\Rightarrow$ $O(n^2 \cdot m)$

APSP / Kantengewichte

Bessere Strategie:

- reduziere n Aufrufe des Bellman-Ford-Algorithmus auf n Aufrufe des Dijkstra-Algorithmus

Problem:

- Dijkstra-Algorithmus funktioniert nur für nichtnegative Kantengewichte

Lösung:

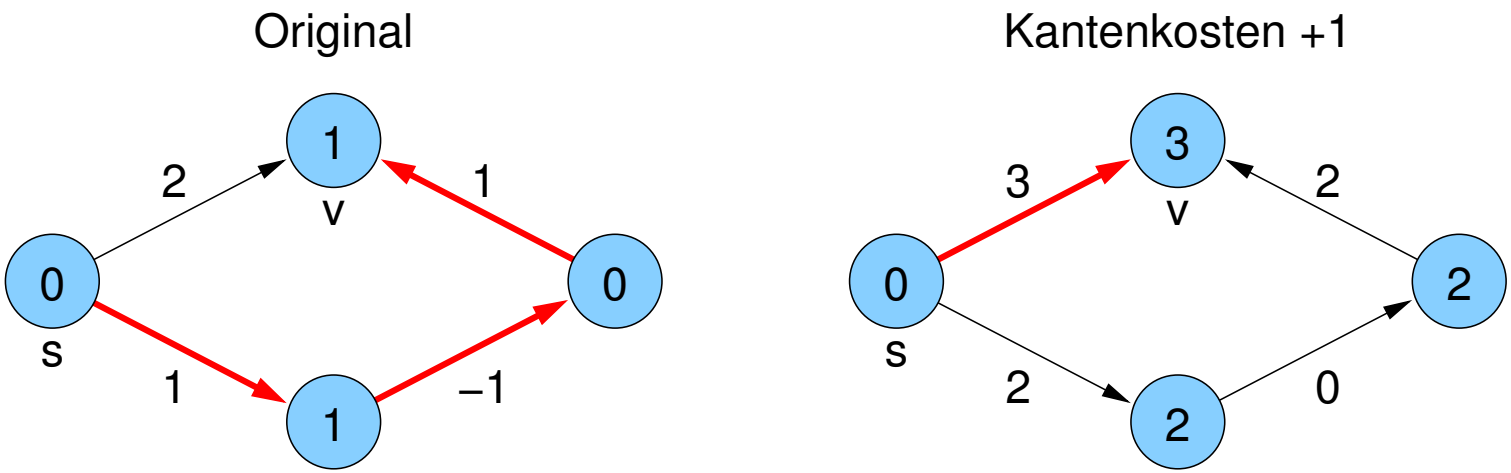
- Umwandlung in nichtnegative Kantenkosten ohne Verfälschung der kürzesten Wege

Naive Modifikation der Kantengewichte

Naive Idee:

- negative Kantengewichte eliminieren, indem auf jedes Kantengewicht der gleiche Wert c addiert wird

$\Rightarrow$ verfälscht kürzeste Pfade



Knotenpotential

Sei $\Phi : V \mapsto \mathbb{R}$ eine Funktion, die jedem Knoten ein Potential zuordnet.

Modifizierte Kantenkosten von $e = (v, w)$:

$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$

Lemma

Seien p und q Wege von v nach w in G.

$c(p)$ und $c(q)$ bzw. $\bar{c}(p)$ und $\bar{c}(q)$ seien die aufsummierten Kosten bzw. modifizierten Kosten der Kanten des jeweiligen Pfads.

Dann gilt für jedes Potential Φ:

$$\bar{c}(p) < \bar{c}(q) \quad \Leftrightarrow \quad c(p) < c(q)$$

Knotenpotential

Beweis.

Sei $p = (v_1, \ldots, v_k)$ beliebiger Weg und $\forall i: \; e_i = (v_i, v_{i+1}) \in E$

Es gilt:

$$\begin{aligned} \bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) \\ &= \sum_{i=1}^{k-1} (\Phi(v_i) + c(e_i) - \Phi(v_{i+1})) \\ &= \Phi(v_1) + c(p) - \Phi(v_k) \end{aligned}$$

d.h. modifizierte Kosten eines Pfads hängen nur von ursprünglichen Pfadkosten und vom Potential des Anfangs- und Endknotens ab.
(Im Lemma ist $v_1 = v$ und $v_k = w$) □

Potential für nichtnegative Kantengewichte

Lemma

Annahme:

- *Graph hat keine negativen Kreise*
- *alle Knoten von s aus erreichbar*

Sei für alle Knoten v das Potential $\Phi(v) = d(s, v)$.

Dann gilt für alle Kanten e: $\bar{c}(e) \geq 0$

Beweis.

- für alle Knoten v gilt nach Annahme: $d(s, v) \in \mathbb{R}$ (also $\neq \pm\infty$)
- für jede Kante $e = (v, w)$ ist

$$\begin{aligned} d(s, v) + c(e) &\geq d(s, w) \\ d(s, v) + c(e) - d(s, w) &\geq 0 \end{aligned}$$

□

Johnson-Algorithmus für APSP

- füge neuen Knoten s und Kanten (s, v) für alle v hinzu mit $c(s, v) = 0$

$\Rightarrow$ alle Knoten erreichbar

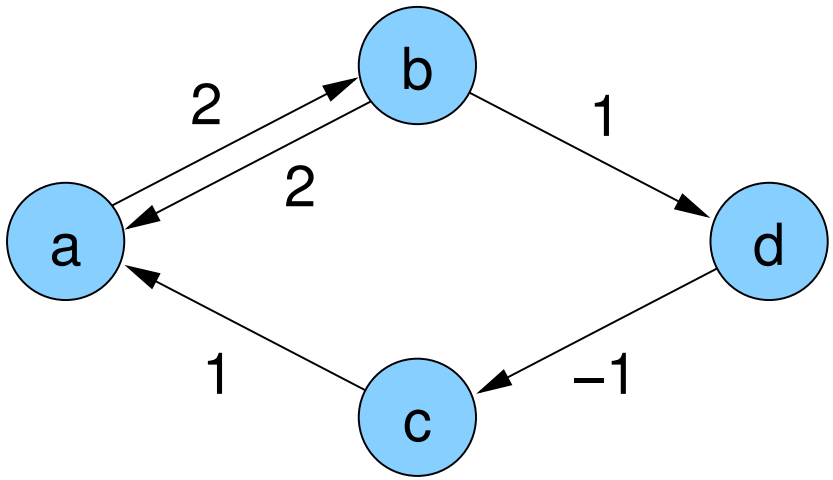
- berechne $d(s, v)$ mit Bellman-Ford-Algorithmus
- setze $\Phi(v) = d(s, v)$ für alle v
- berechne modifizierte Kosten $\bar{c}(e)$

$\Rightarrow$ $\bar{c}(e) \geq 0$, kürzeste Wege sind noch die gleichen

- berechne für alle Knoten v die Distanzen $\bar{d}(v, w)$ mittels Dijkstra-Algorithmus mit modifizierten Kantenkosten auf dem Graph ohne Knoten s
- berechne korrekte Distanzen $d(v, w) = \bar{d}(v, w) + \Phi(w) - \Phi(v)$

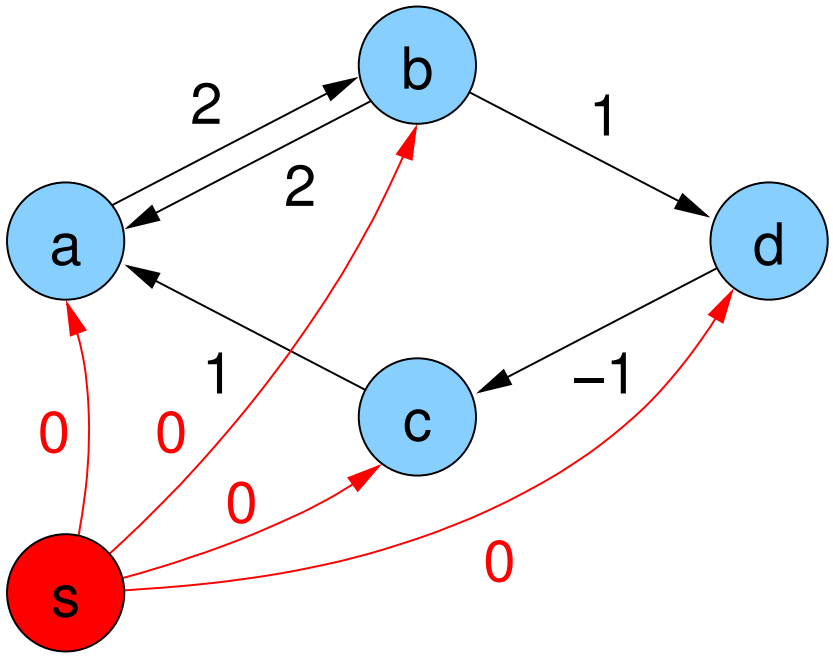
Johnson-Algorithmus für APSP

Beispiel:



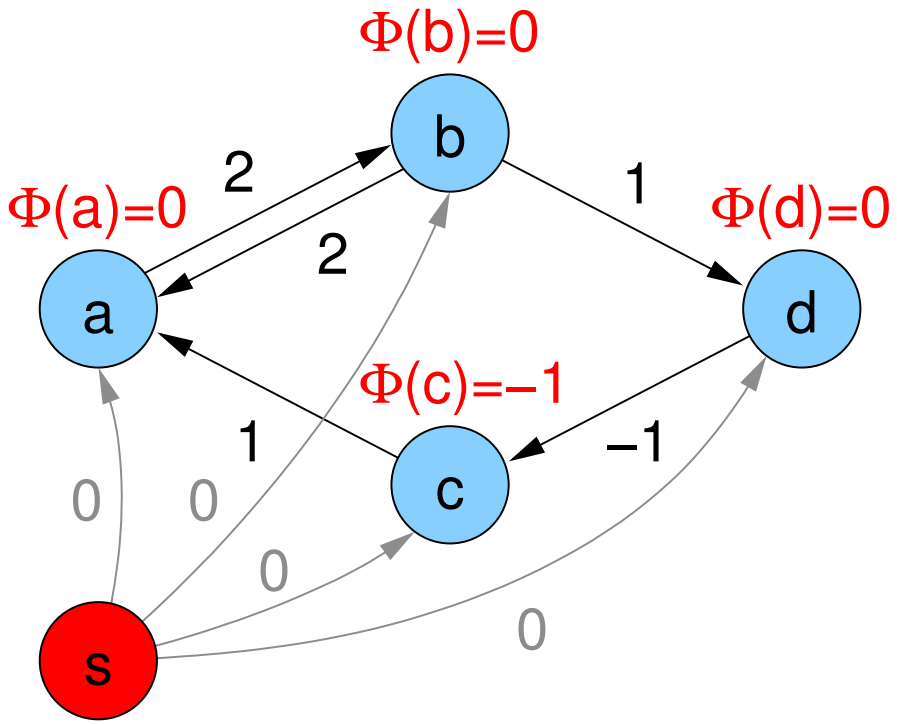
Johnson-Algorithmus für APSP

1. künstlicher Startknoten s:



Johnson-Algorithmus für APSP

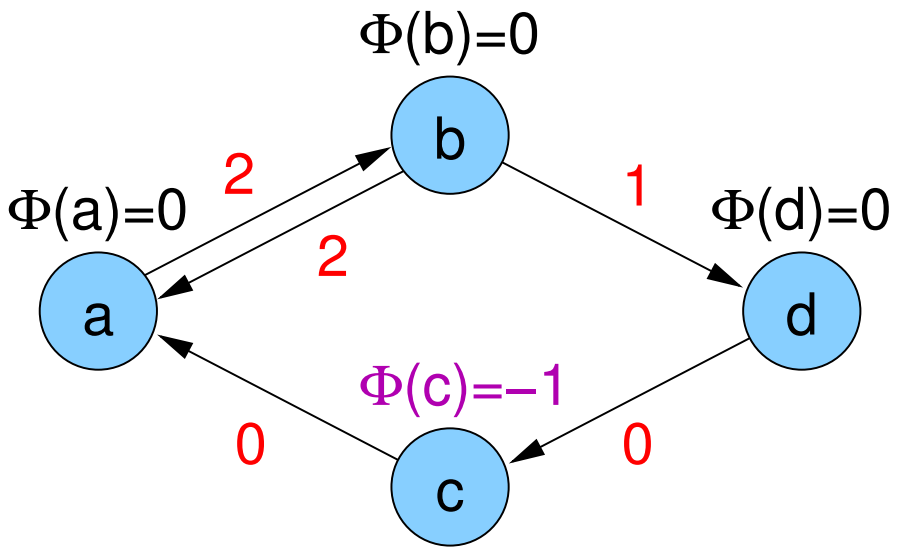
2. Bellman-Ford-Algorithmus auf s:



Johnson-Algorithmus für APSP

3. $\bar{c}(e)$-Werte für alle $e = (v, w)$ berechnen:

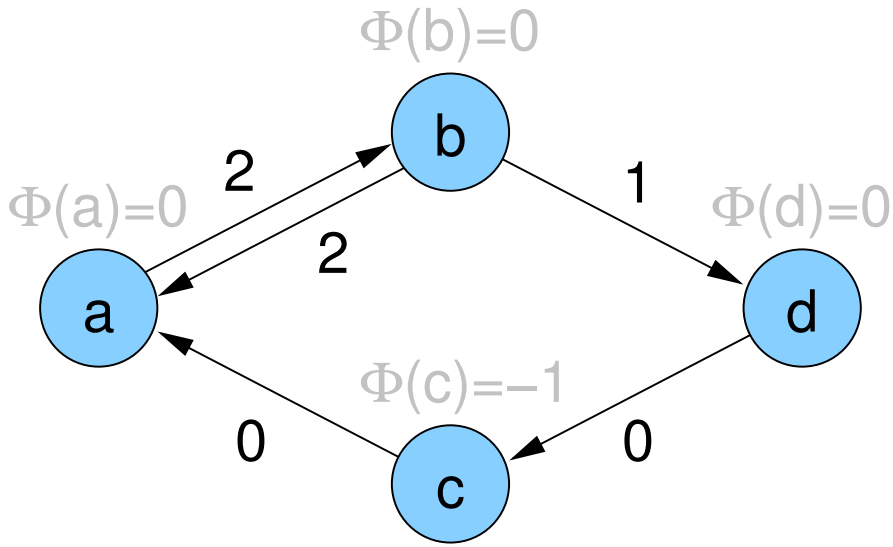
$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$



Johnson-Algorithmus für APSP

4. Distanzen $\bar{d}$ mit modifizierten Kantengewichten via Dijkstra:

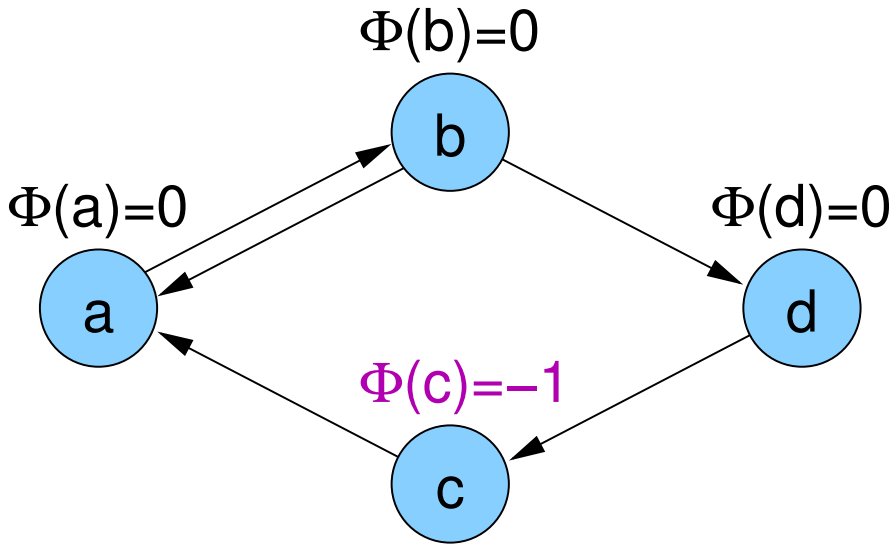
$\bar{\mathbf{d}}$	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0



Johnson-Algorithmus für APSP

5. korrekte Distanzen berechnen: $d(v,w) = \bar{d}(v,w) + \Phi(w) - \Phi(v)$

d	*a*	*b*	*c*	*d*
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	−1	0



Johnson-Algorithmus für APSP

Laufzeit:

$$\begin{aligned} T_{\text{Johnson}}(n,m) &= \mathcal{O}(T_{\text{Bellman-Ford}}(n+1, m+n) + n \cdot T_{\text{Dijkstra}}(n,m)) \\ &= \mathcal{O}((m+n)\cdot(n+1) + n\cdot(n\log n + m)) \\ &= \mathcal{O}(m\cdot n + n^2 \log n) \end{aligned}$$

(bei Verwendung von Fibonacci Heaps)

Floyd-Warshall-Algorithmus für APSP

Grundlage:

- geht der kürzeste Weg von u nach w über v, dann sind auch die beiden Teile von u nach v und von v nach w kürzeste Pfade zwischen diesen Knoten
- Annahme: alle kürzesten Wege bekannt, die nur über Zwischenknoten mit Index kleiner als k gehen

$\Rightarrow$ kürzeste Wege über Zwischenknoten mit Indizes bis einschließlich k können leicht berechnet werden:

  - entweder der schon bekannte Weg über Knoten mit Indizes kleiner als k
  - oder über den Knoten mit Index k (hier im Algorithmus der Knoten v)

Floyd-Warshall-Algorithmus für APSP

Floyd-Warshall APSP Algorithmus

Input : Graph $G = (V, E)$, $c : E \mapsto R$
Output : Distanzen $d(u, v)$ zwischen allen $u, v \in V$

for $u, v \in V$ **do**
  $d(u, v) = \infty$; $\text{pred}(u, v) = \perp$;
for $v \in V$ **do** $d(v, v) = 0$;
for $(u, v) \in E$ **do**
  $d(u, v) = c(u, v)$; $\text{pred}(u, v) = u$;
for $v \in V$ **do**
  for $\{u, w\} \in V \times V$ **do**
    if $d(u, w) > d(u, v) + d(v, w)$ **then**
      $d(u, w) = d(u, v) + d(v, w)$;
      $\text{pred}(u, w) = \text{pred}(v, w)$;

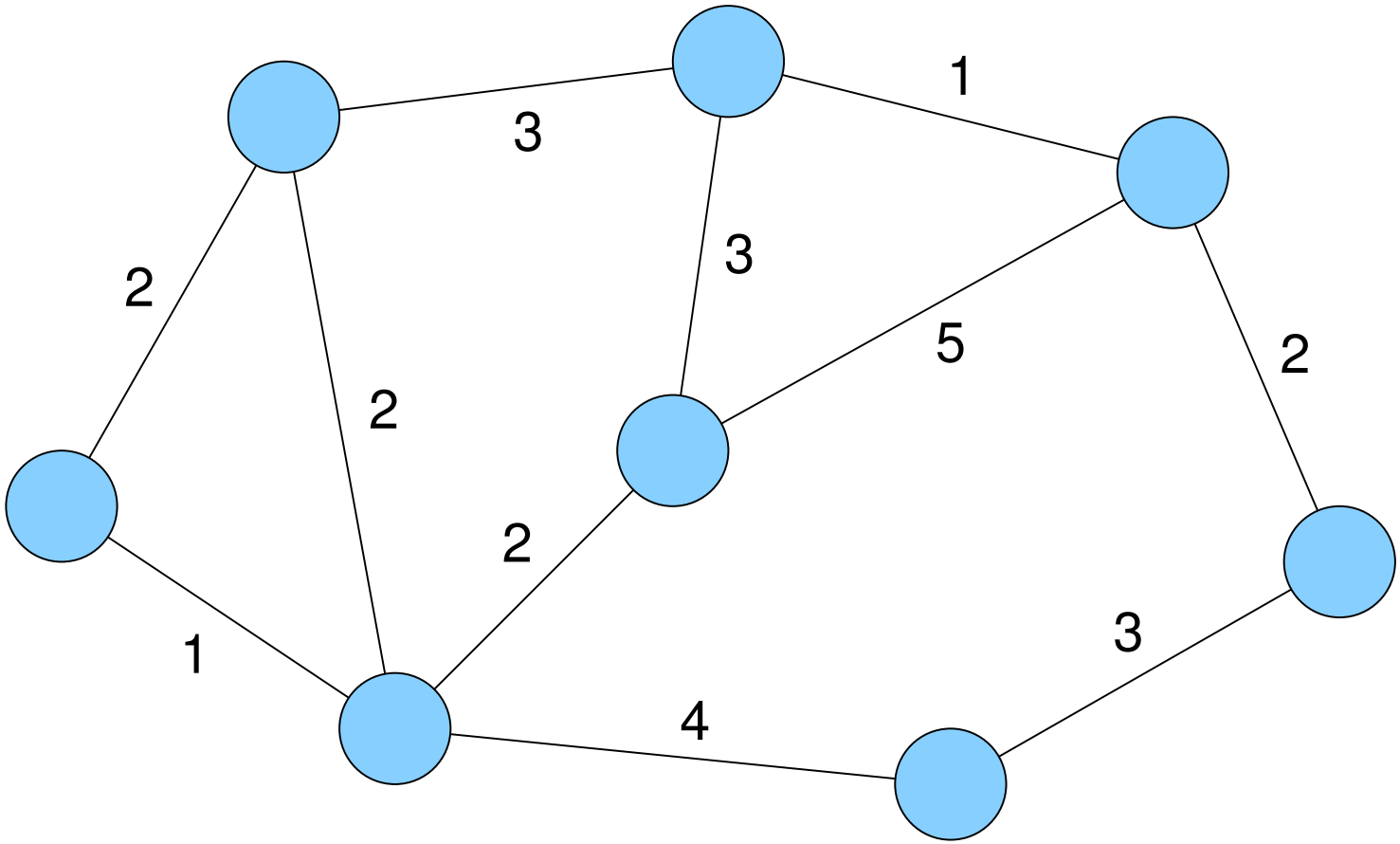
Floyd-Warshall-Algorithmus für APSP

- Komplexität: $O(n^3)$
- funktioniert auch, wenn Kanten mit negativem Gewicht existieren
- Kreise negativer Länge werden nicht direkt erkannt und verfälschen das Ergebnis, sind aber indirekt am Ende an negativen Diagonaleinträgen der Distanzmatrix erkennbar

Übersicht

Minimaler Spannbaum

Frage: Welche Kanten nehmen, um mit minimalen Kosten alle Knoten zu verbinden?



Minimaler Spannbaum

Eingabe:

- ungerichteter Graph $G = (V, E)$
- Kantenkosten $c : E \mapsto \mathbb{R}_+$

Ausgabe:

- Kantenteilmenge $T \subseteq E$, so dass Graph (V, T) verbunden und $c(T) = \sum_{e \in T} c(e)$ minimal

Beobachtung:

- T formt immer einen Baum
  (wenn Kantengewichte echt positiv)

$\Rightarrow$ Minimaler Spannbaum (MSB) / Minimum Spanning Tree (MST)

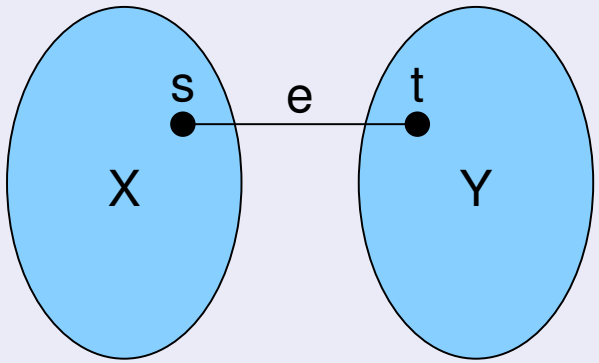
Minimaler Spannbaum

Lemma

Sei

- (X, Y) *eine Partition von* V *(d.h.* $X \cup Y = V$ *und* $X \cap Y = \emptyset$*) und*
- $e = \{s, t\}$ *eine Kante mit minimalen Kosten mit* $s \in X$ *und* $t \in Y$*.*

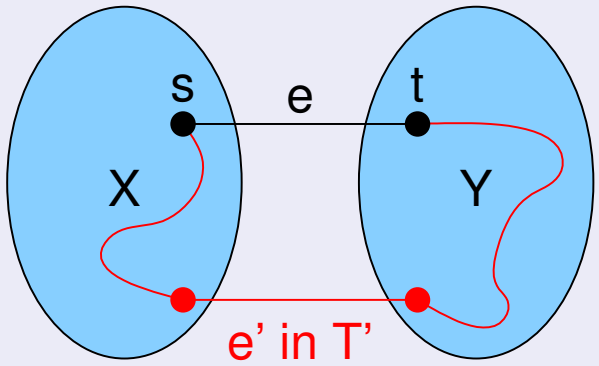
Dann gibt es einen minimalen Spannbaum T, der e enthält.



Minimaler Spannbaum

Beweis.

- gegeben X, Y und $e = \{s, t\}$: (X, Y)-Kante minimaler Kosten
- betrachte beliebigen MSB T', der e nicht enthält
- betrachte Verbindung zwischen s und t in T', darin muss es mindestens eine Kante e' zwischen X und Y geben



- Ersetzung von e' durch e führt zu Baum T'', der höchstens Kosten von MSB T' hat (also auch ein MSB ist)

□

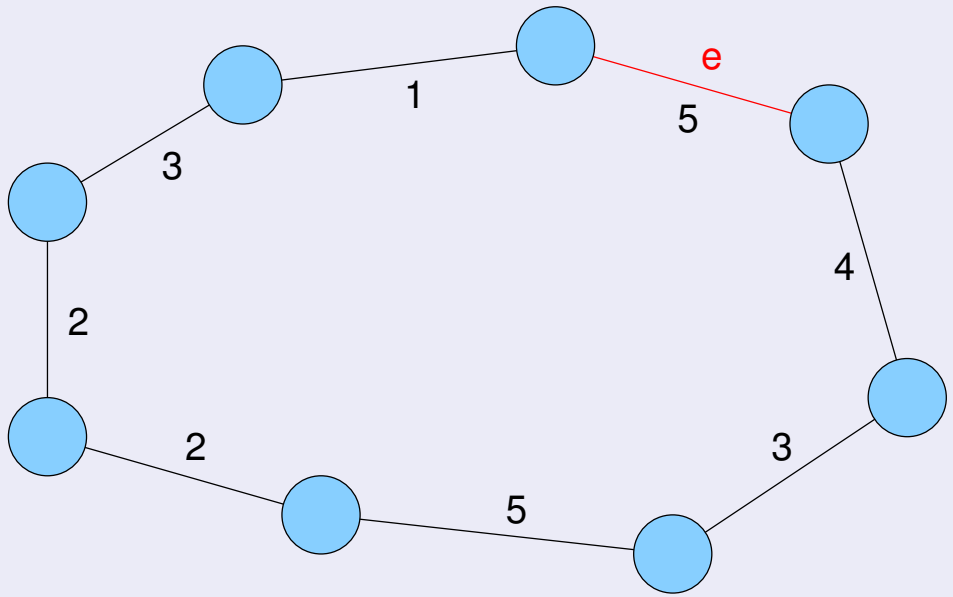
Minimaler Spannbaum

Lemma

Betrachte

- *beliebigen Kreis C in G*
- *eine Kante e in C mit maximalen Kosten*

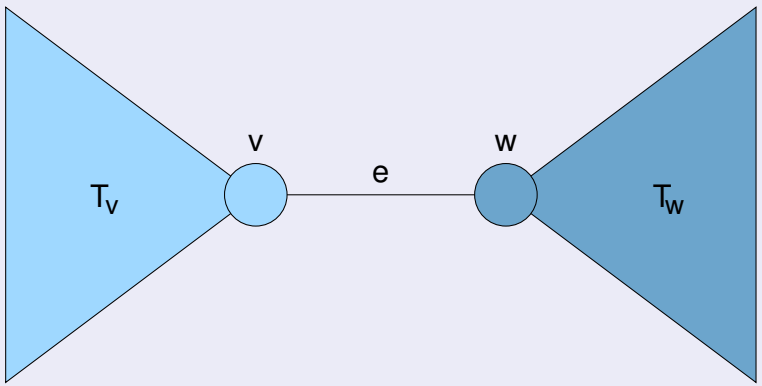
Dann ist jeder MSB in G ohne e auch ein MSB in G

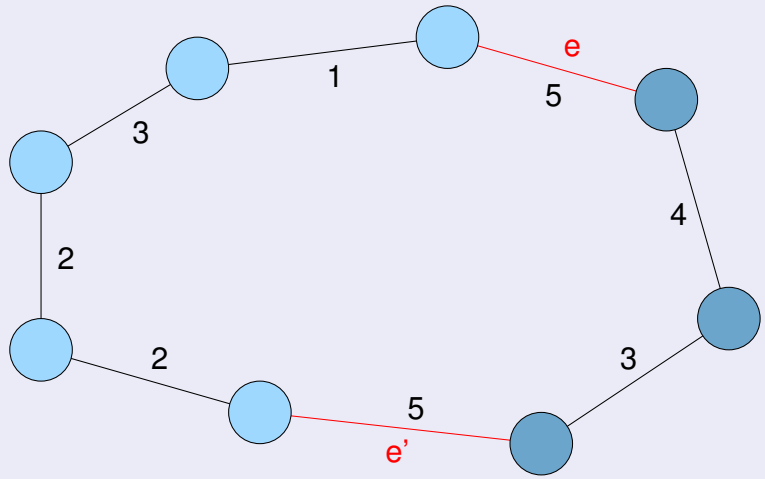


Minimaler Spannbaum

Beweis.

- betrachte beliebigen MSB T in G
- Annahme: T enthält e



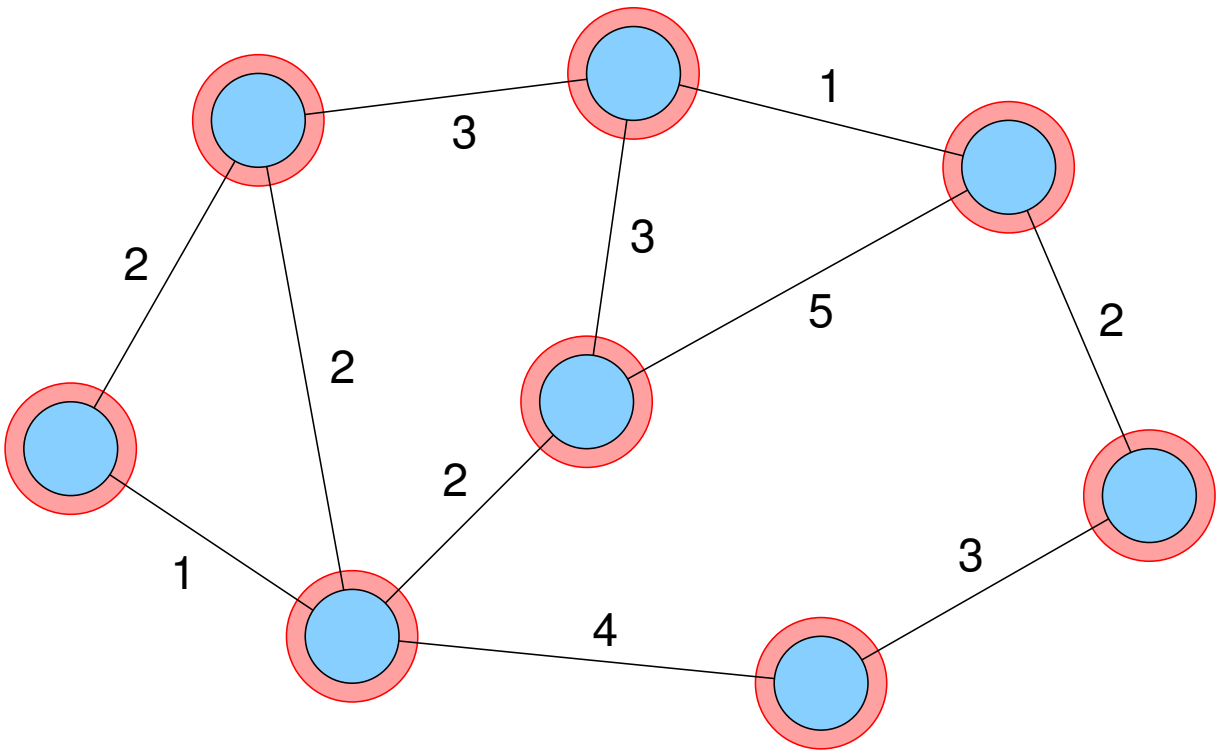


- es muss (mindestens) eine weitere Kante e' in C geben, die einen Knoten aus T_v mit einem Knoten aus T_w verbindet
- Ersetzen von e durch e' ergibt einen Baum T' dessen Gewicht nicht größer sein kann als das von T, also ist T' auch MSB

Minimaler Spannbaum

Regel:

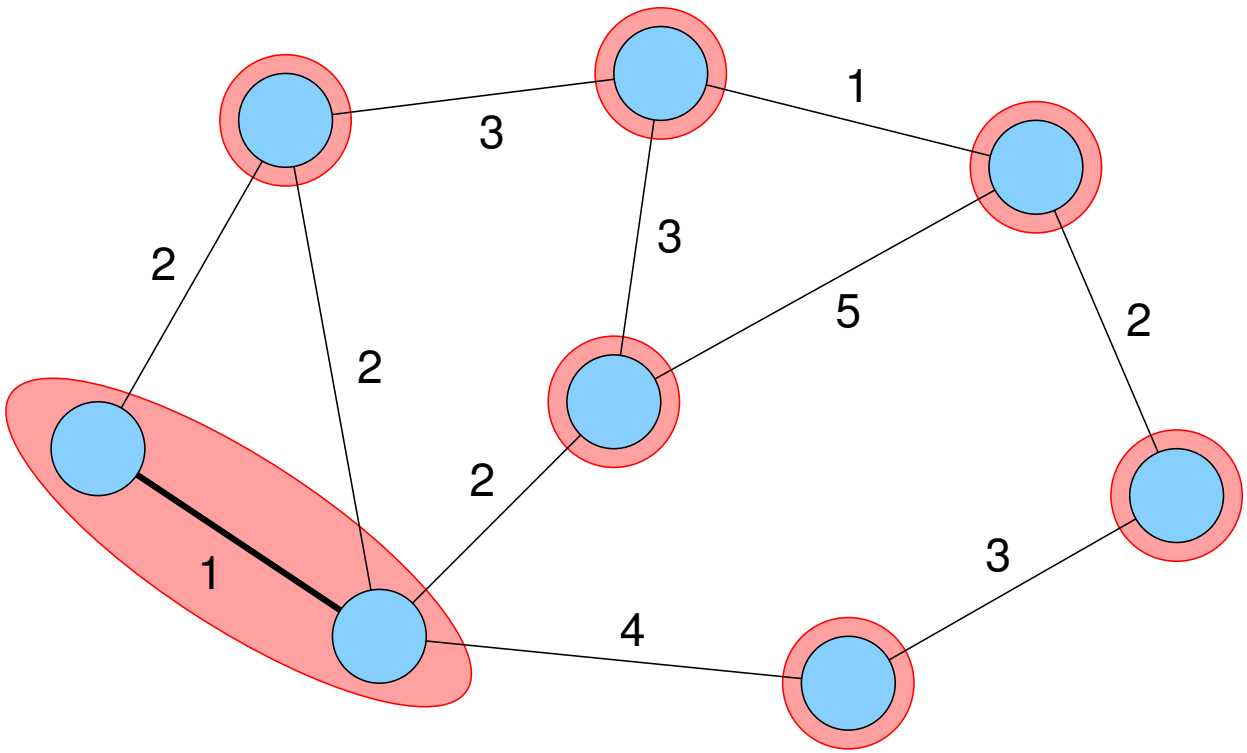
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

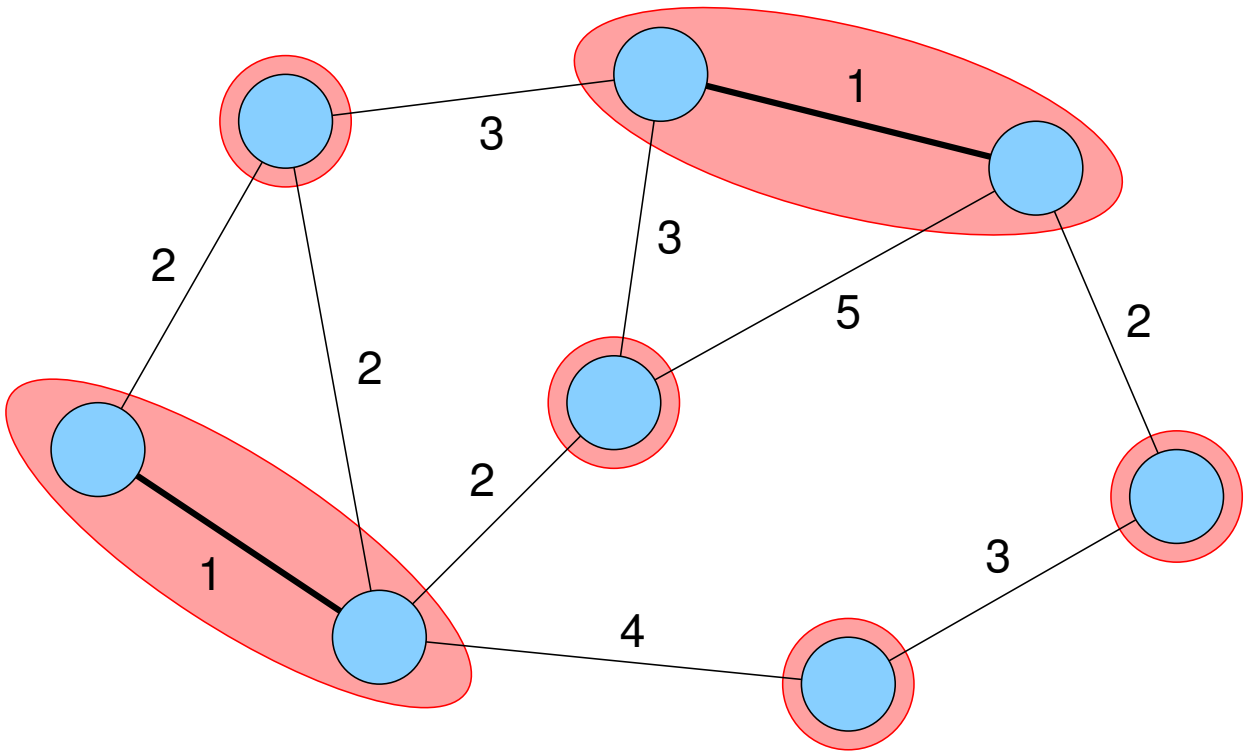
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

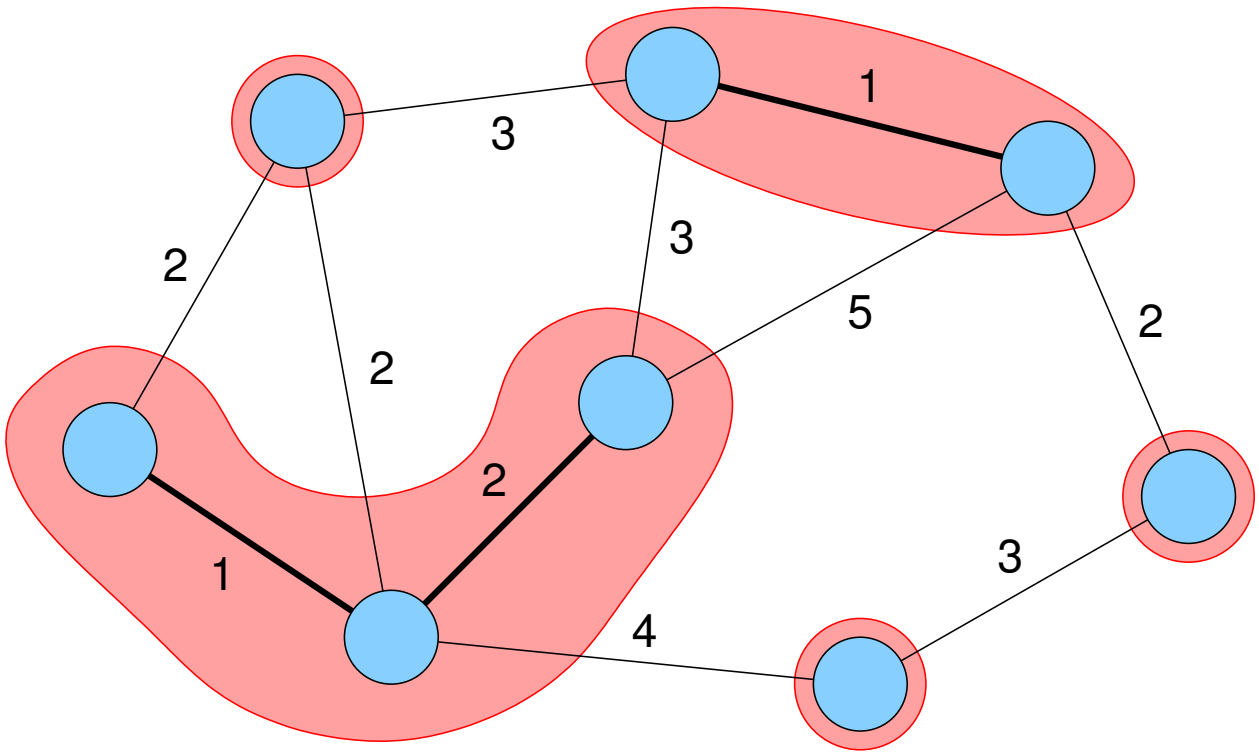
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

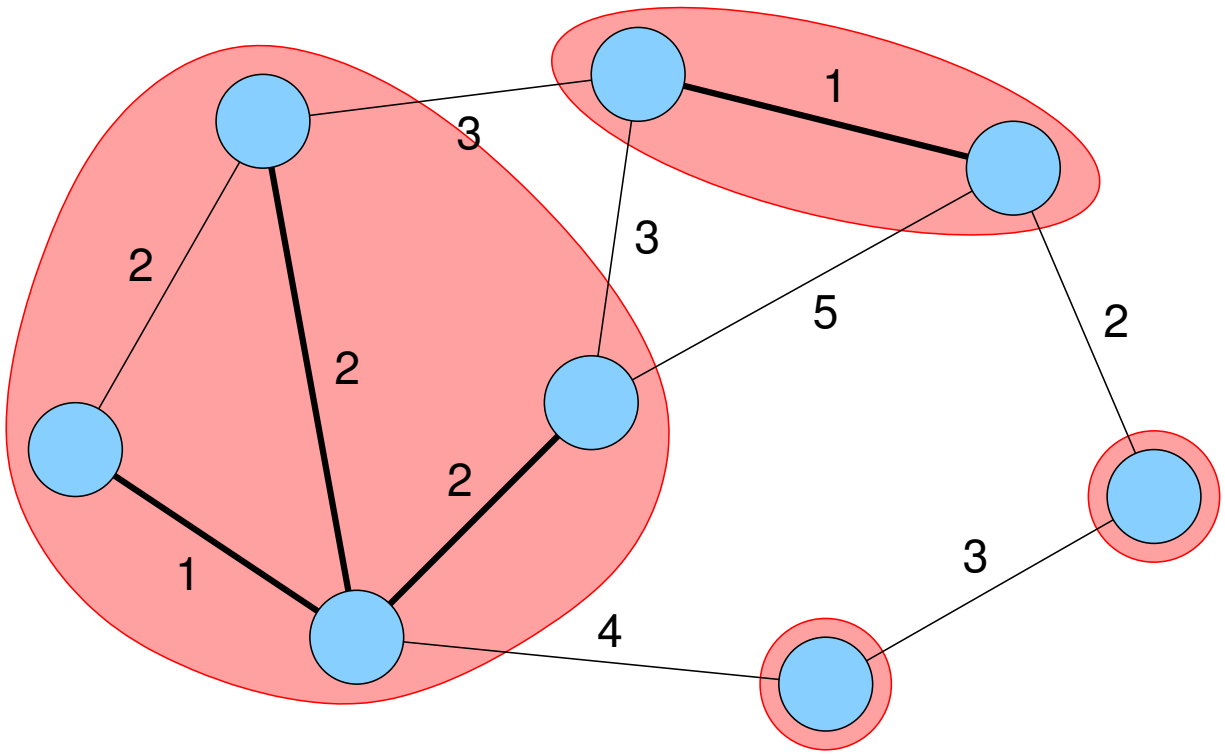
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

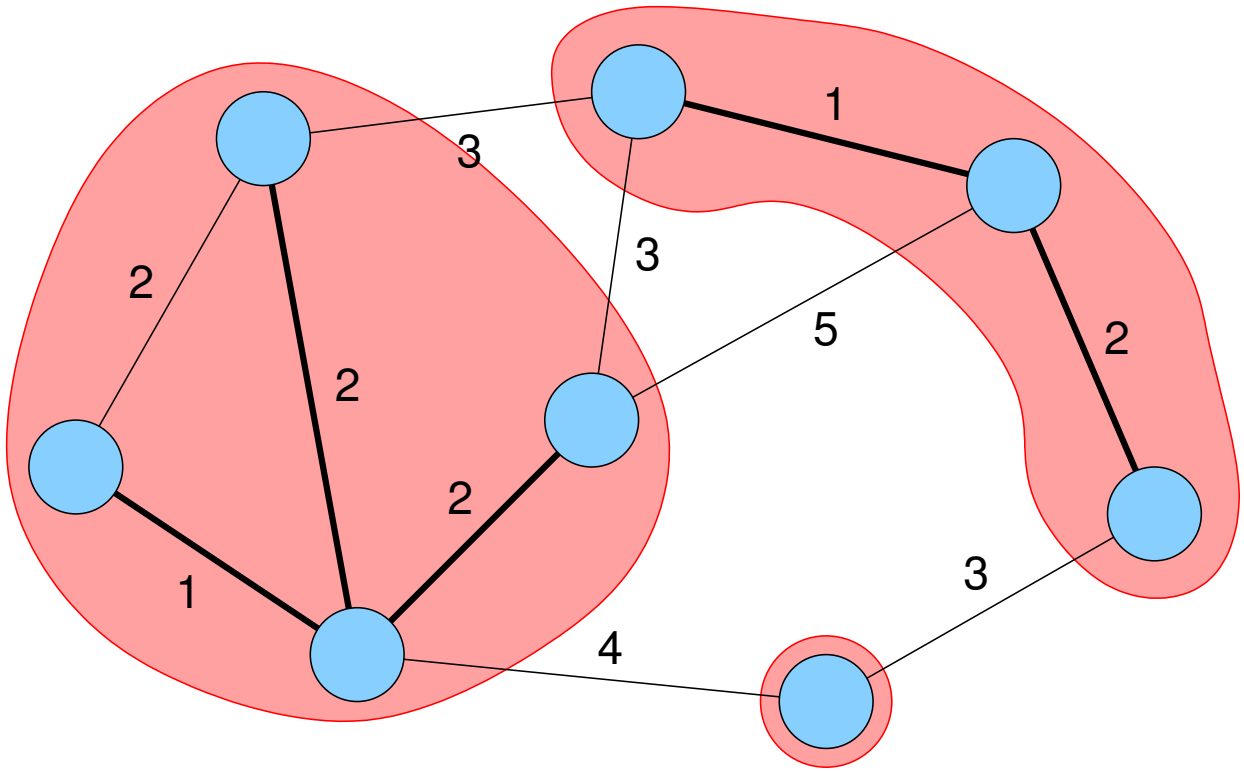
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

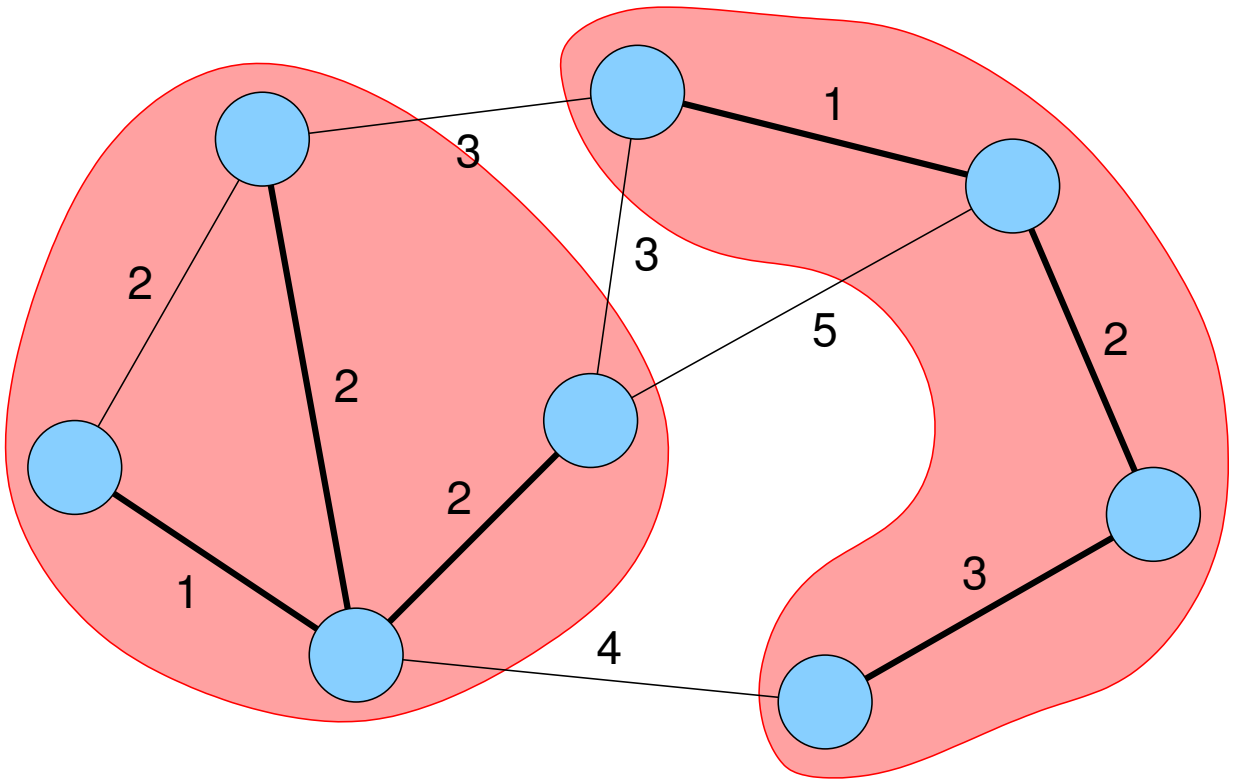
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

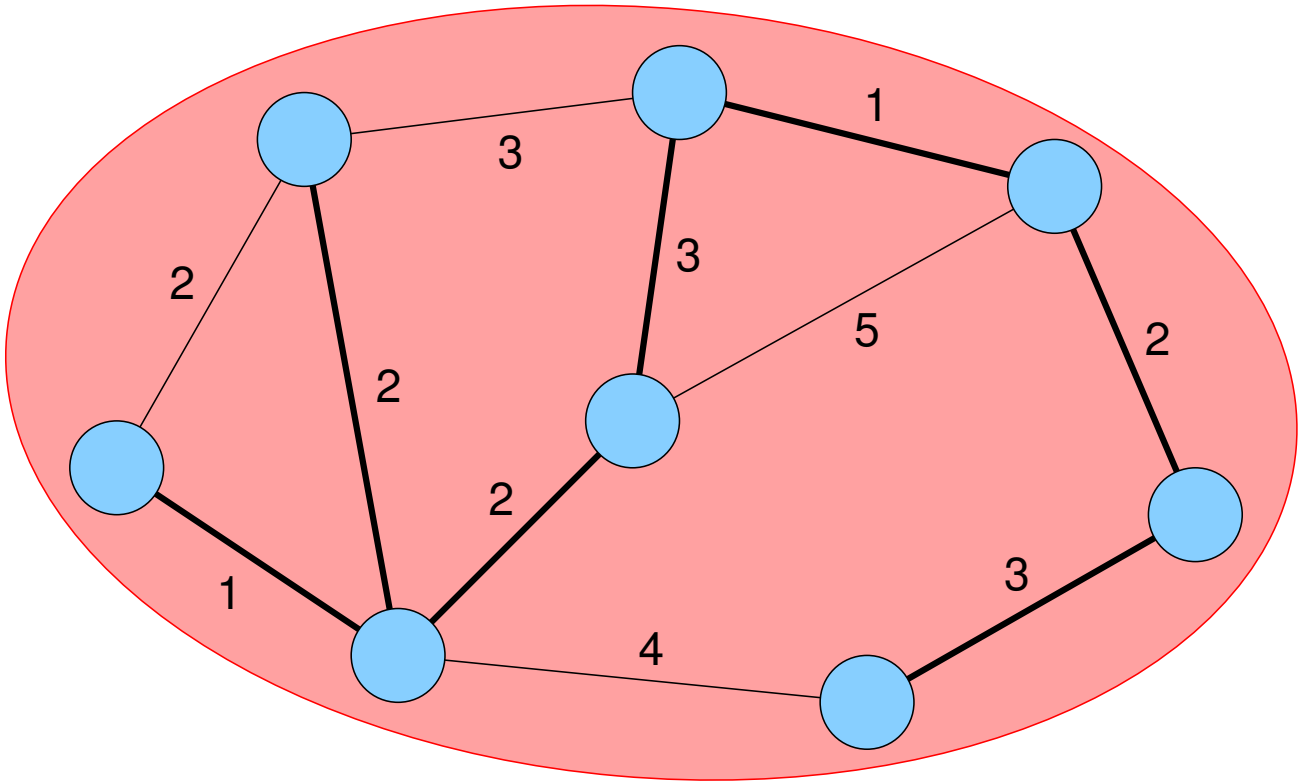
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

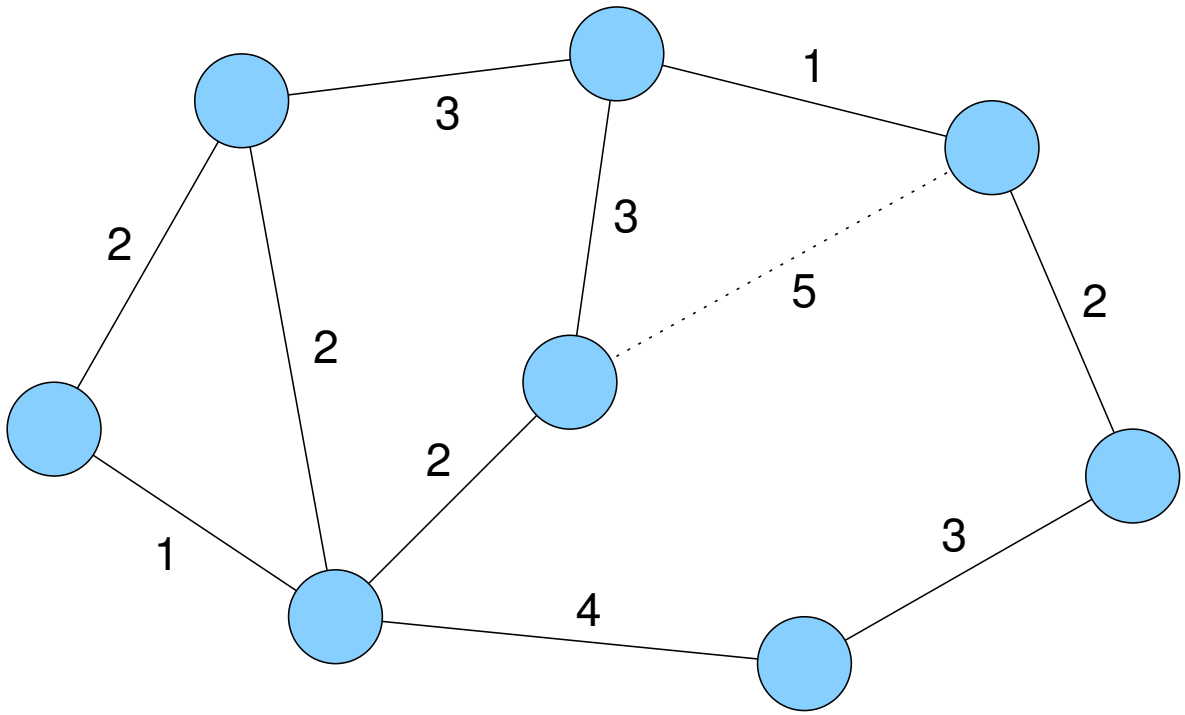
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

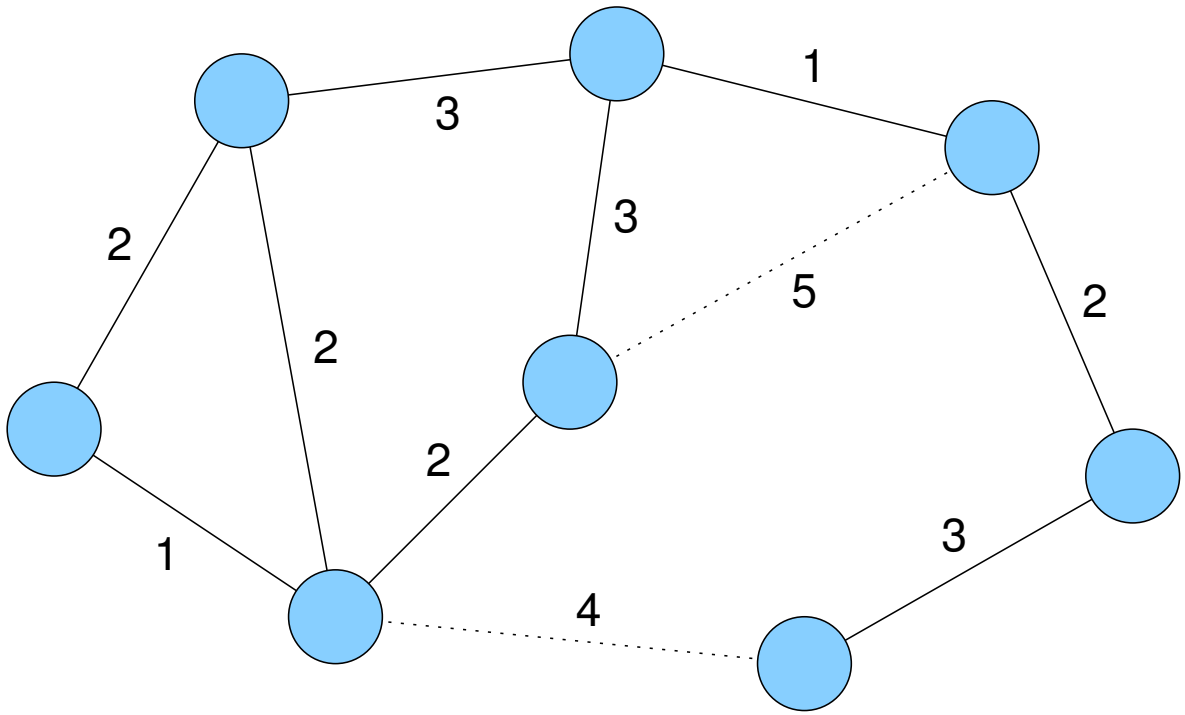
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

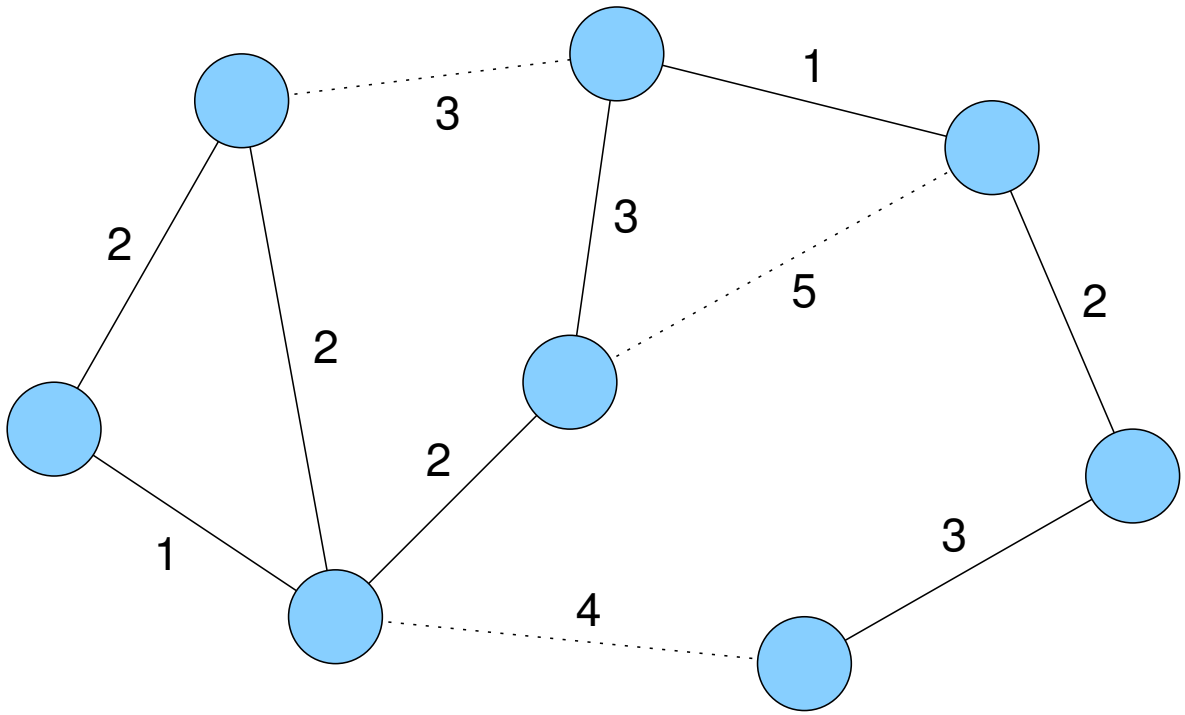
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

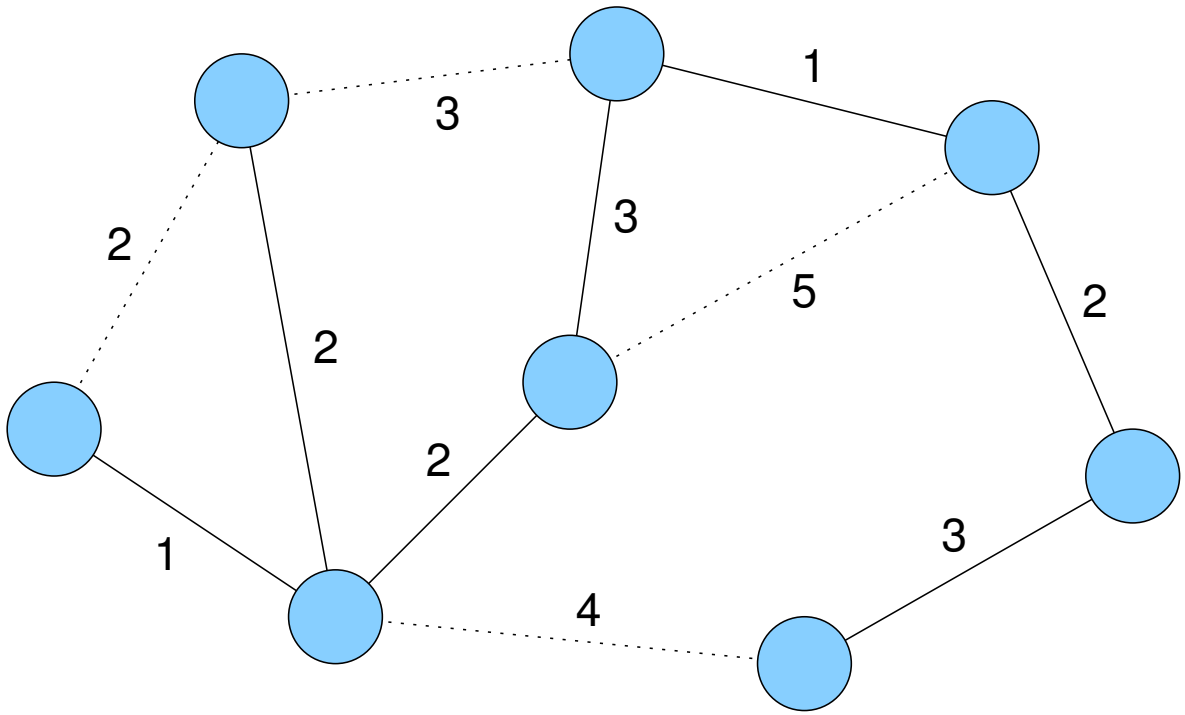
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

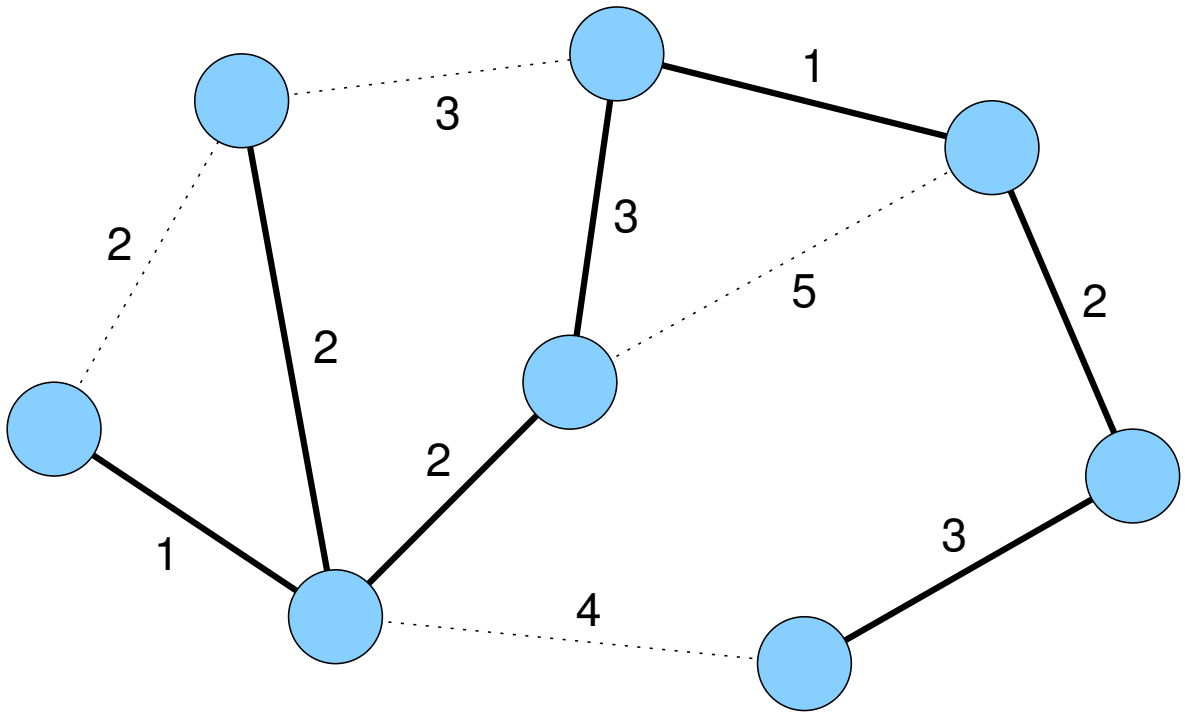
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Problem: Wie implementiert man die Regeln effizient?

Strategie aus dem ersten Lemma:

- sortiere Kanten aufsteigend nach ihren Kosten
- setze $T = \emptyset$ (leerer Baum)
- teste für jede Kante $\{u, v\}$ (in aufsteigender Reihenfolge), ob u und v schon in einer Zusammenhangskomponente (also im gleichen Baum) sind
- falls nicht, füge $\{u, v\}$ zu T hinzu (nun sind u und v im gleichen Baum)

Algorithmus von Kruskal

```
Set<Edge> MST_Kruskal (V, E, c) {
    T = ∅;
    S = sort(E);   // aufsteigend sortieren
    foreach (e = {u, v} ∈ S)
        if (u und v in verschiedenen Bäumen in T)
            T = T ∪ e;
    return T;
}
```

Problem:

- Umsetzung des Tests auf gleiche / unterschiedliche Zusammenhangskomponente

Union-Find-Datenstruktur

Union-Find-Problem:

- gegeben sind (disjunkte) Mengen von Elementen
- jede Menge hat genau einen Repräsentanten
- union soll zwei Mengen vereinigen, die durch ihren jeweiligen Repräsentanten gegeben sind
- find soll zu einem gegebenen Element die zugehörige Menge in Form des Repräsentanten finden

Anwendung:

- Knoten seien nummeriert von 0 bis $n - 1$
- Array int parent[n], Einträge verweisen Richtung Repräsentant
- anfangs parent[i]=i für alle i

Union-Find-Datenstruktur

```
int find(int i) {
    if (parent[i] == i)    return i;    // ist i Wurzel des Baums?
    else {    // nein
        k = find( parent[i] );    // suche Wurzel
        parent[i] = k;    // zeige direkt auf Wurzel
        return k;    // gibt Wurzel zurück
    }
}

union(int i, int j) {
    int ri = find(i);
    int rj = find(j);    // suche Wurzeln
    if (ri ≠ rj)
        parent[ri] = rj;    // vereinigen
}
```

Algorithmus von Kruskal

```
Set<Edge> MST_Kruskal (V, E, c) {
    T = ∅;
    S = sort(E);   // aufsteigend sortieren
    for (int i = 0; i < |V|; i++)
        parent[i] = i;
    foreach (e = {u, v} ∈ S)
        if (find(u) ≠ find(v)) {
            T = T ∪ e;
            union(u, v);   // Bäume von u und v vereinigen
        }
    return T;
}
```

Gewichtete union-Operation mit Pfadkompression

- Laufzeit von find hängen von der Höhe des Baums ab
- deshalb wird am Ende von find jeder Knoten auf dem Suchpfad direkt unter die Wurzel gehängt, damit die Suche beim nächsten Mal direkt zu diesem Knoten kommt (Pfadkompression)
- weiterhin sollte bei union der niedrigere Baum unter die Wurzel des höheren gehängt werden (gewichtete Vereinigung)

$\Rightarrow$ Höhe des Baums ist dann $O(\log n)$

Gewichtete union-Operation

```
union(int i, int j) {
    int ri = find(i);
    int rj = find(j);    // suche Wurzeln
    if (ri ≠ rj)
        if (height[ri] < height[rk])
            parent[ri] = rj;
        else {
            parent[rj] = ri;
            if (height[ri] == height[rk])
                height[ri]++;
        }
}
```

union / find - Kosten

Situation:

- Folge von union / find -Operationen auf einer Partition von n Elementen, darunter $n - 1$ union-Operationen

Komplexität:

- amortisiert $\log^* n$ pro Operation, wobei

$$\log^* n = \min\{i \geq 1 : \underbrace{\log \log \ldots \log}_{i-\text{mal}} n \leq 1\}$$

- bessere obere Schranke: mit inverser Ackermannfunktion (Vorlesung Effiziente Algorithmen und Datenstrukturen I)
- Gesamtkosten für Kruskal-Algorithmus: $\mathcal{O}(m \log n)$ (Sortieren)

Algorithmus von Prim

Problem: Wie implementiert man die Regeln effizient?

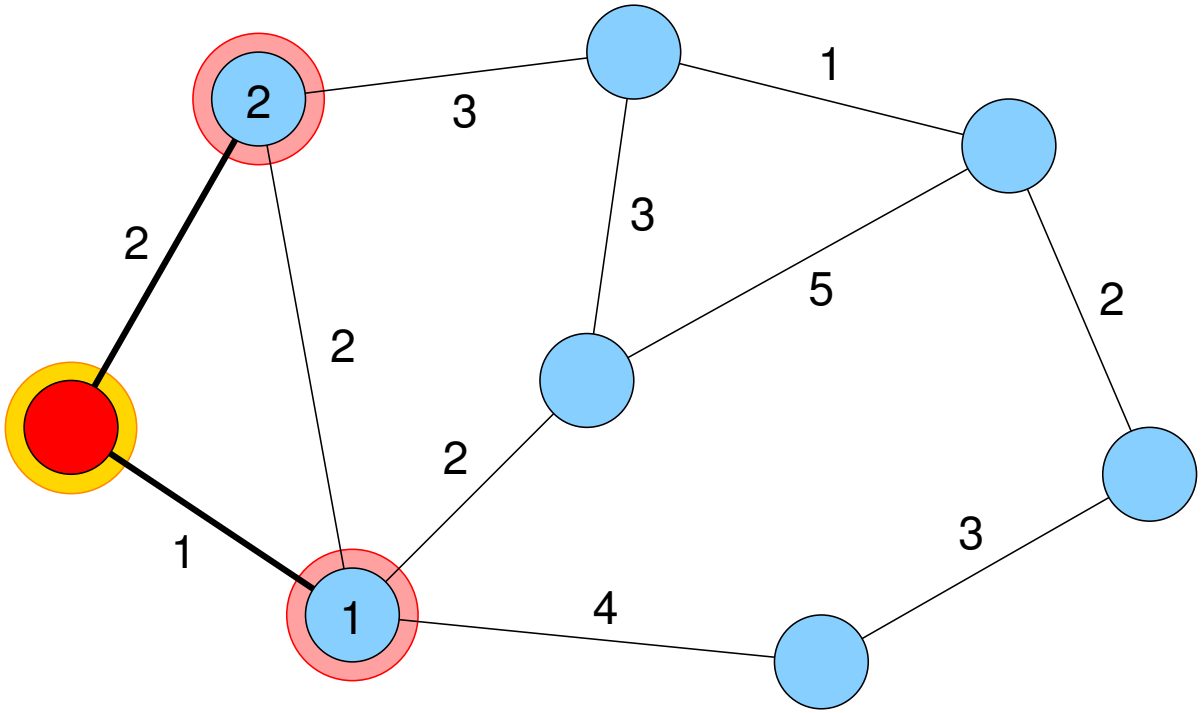
Alternative Strategie aus dem ersten Lemma:

- betrachte wachsenden Baum T, anfangs bestehend aus beliebigem einzelnen Knoten s
- füge zu T eine Kante mit minimalem Gewicht von einem Baumknoten zu einem Knoten außerhalb des Baums ein (bei mehreren Möglichkeiten egal welche)

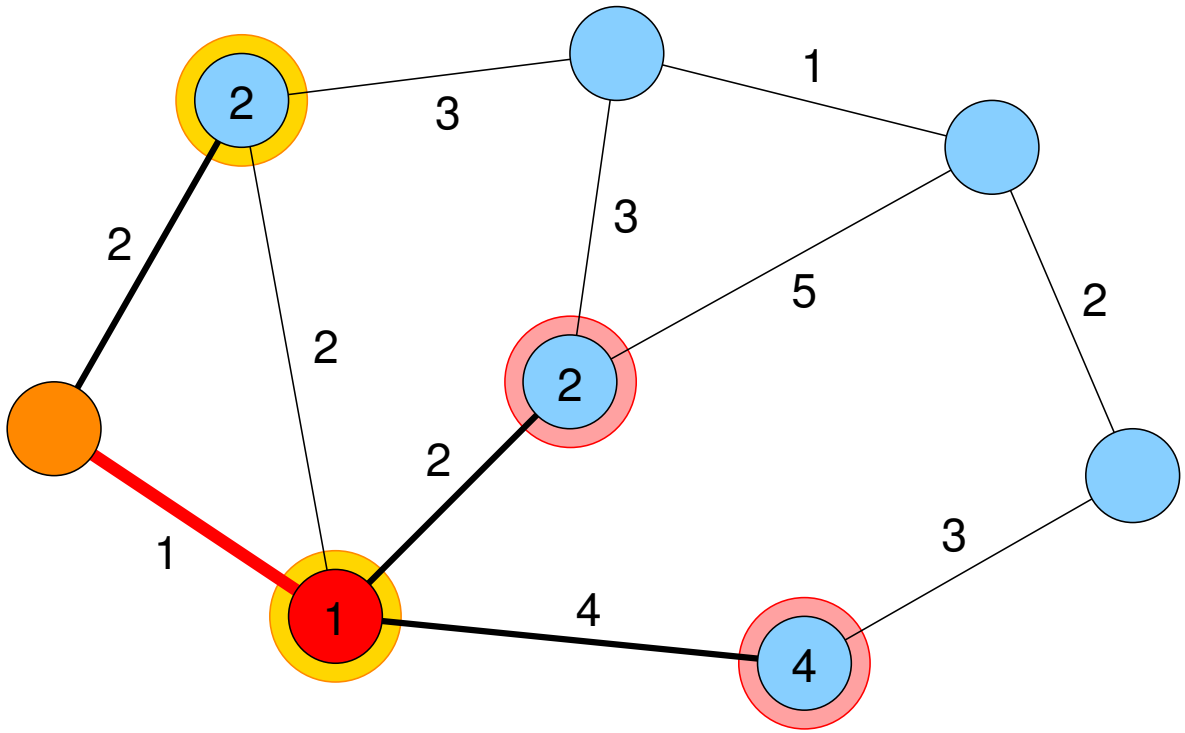
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle n Knoten im Baum

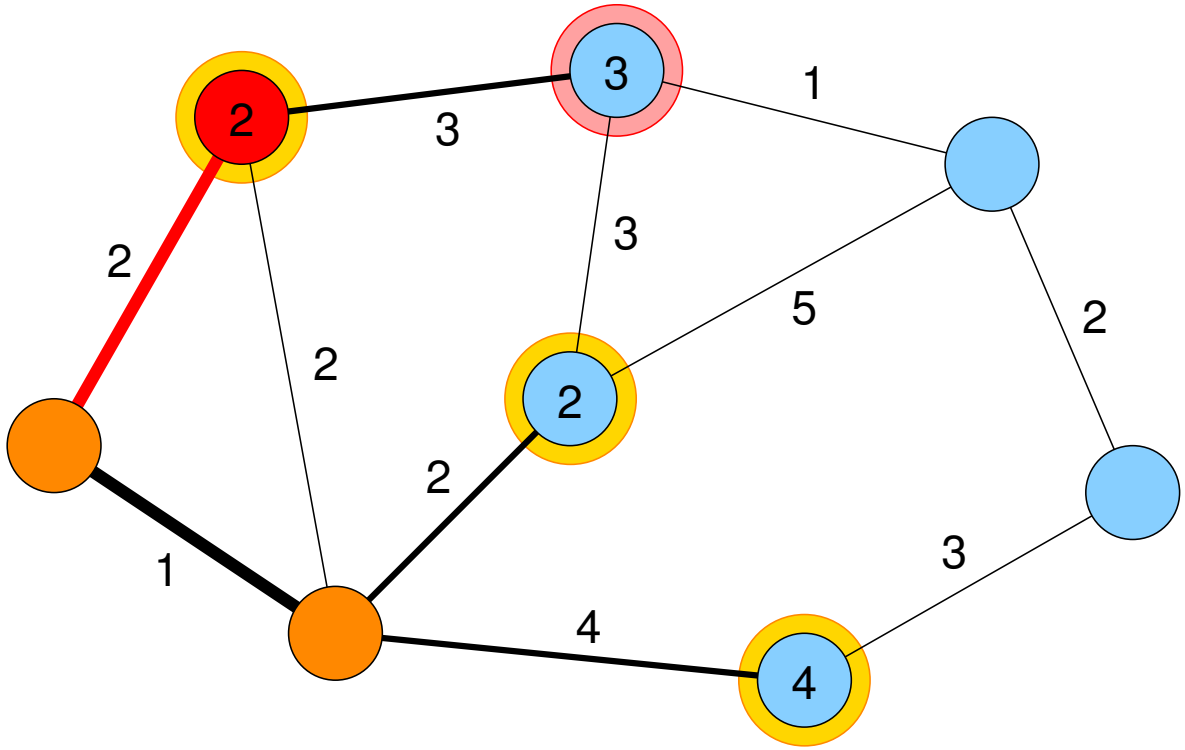
Algorithmus von Prim



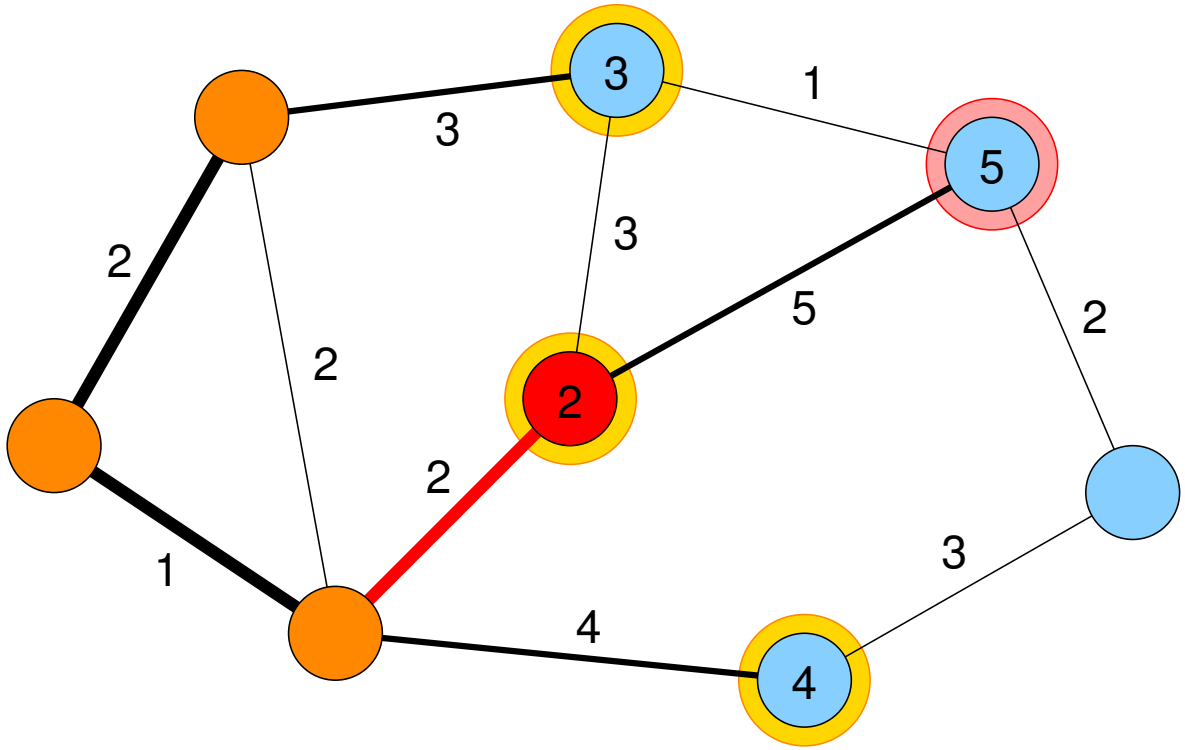
Algorithmus von Prim



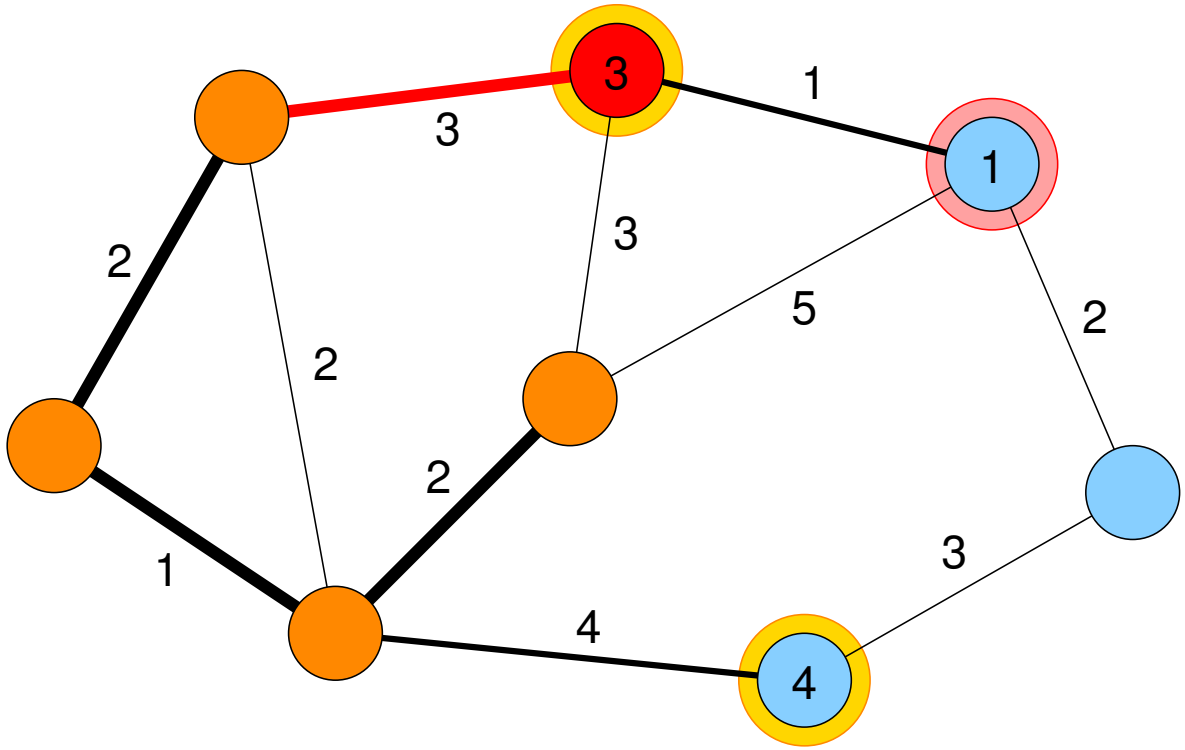
Algorithmus von Prim



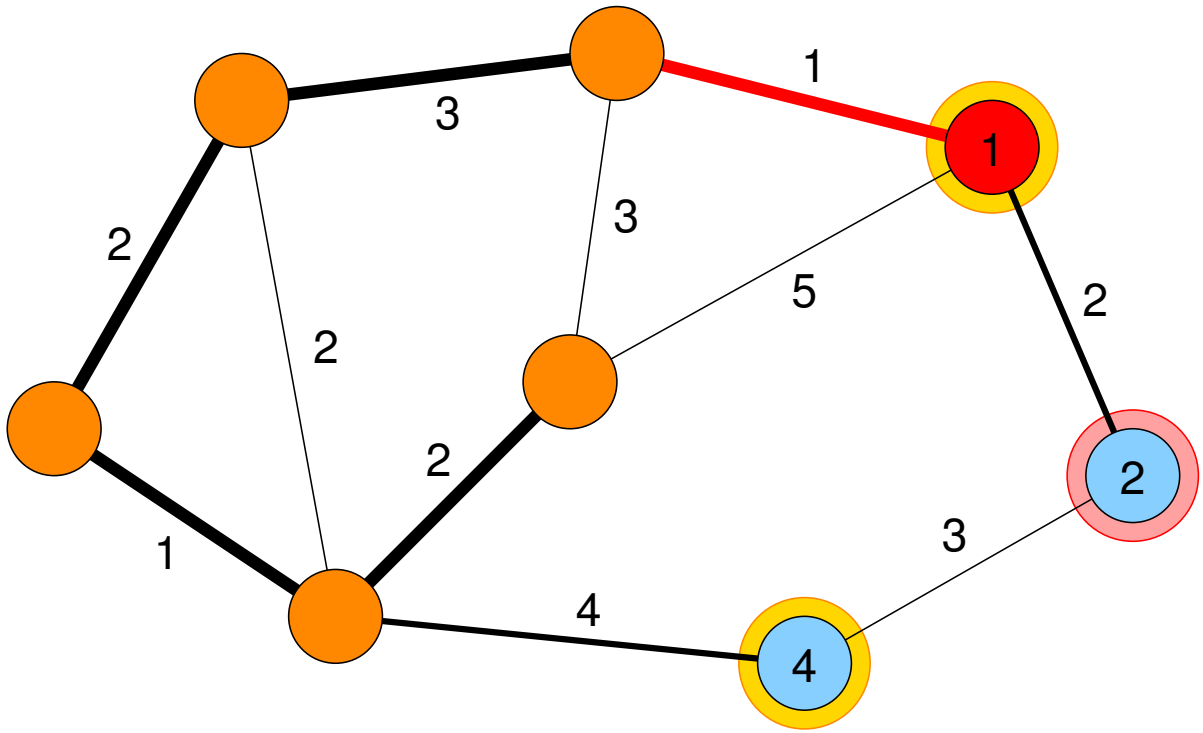
Algorithmus von Prim



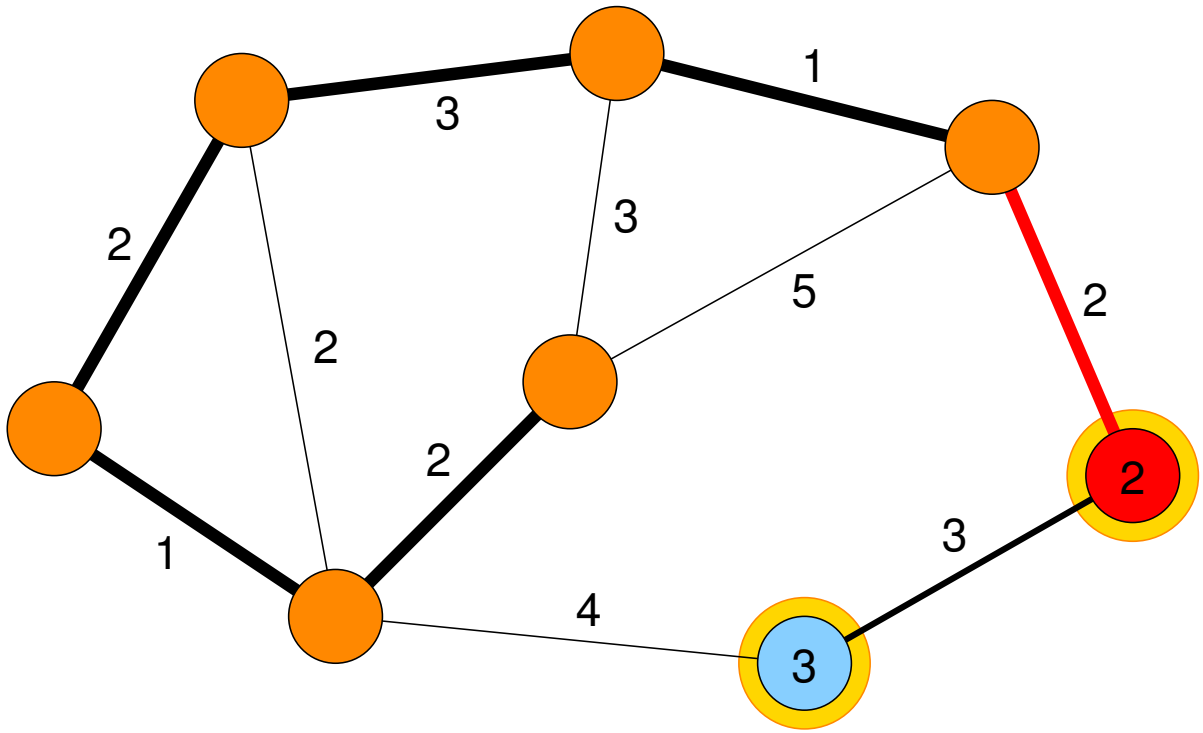
Algorithmus von Prim



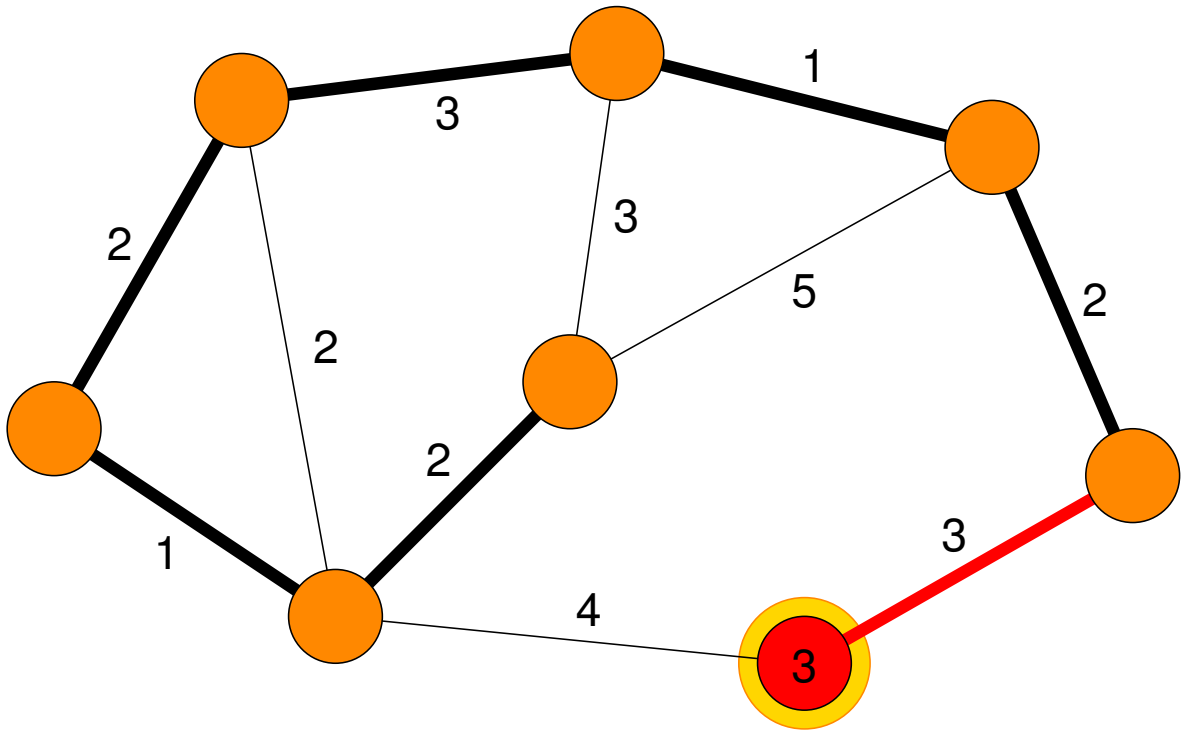
Algorithmus von Prim



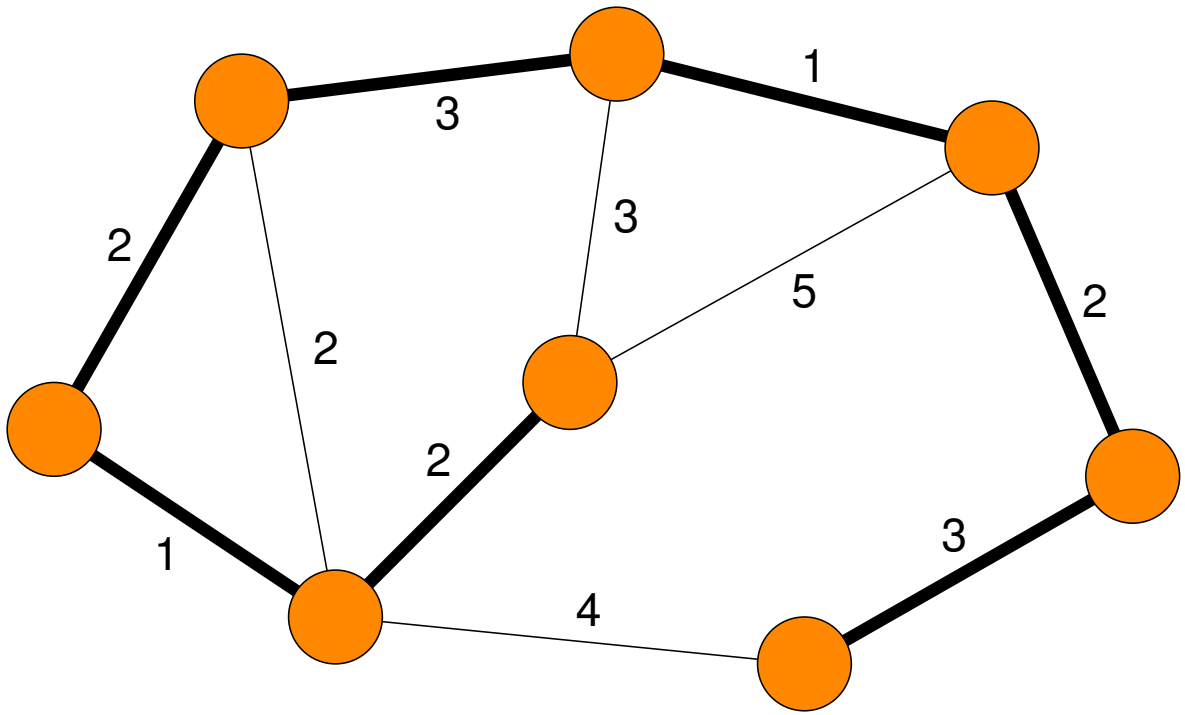
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim



Jarnik-Prim-Algorithmus für MSB

Input : $G = (V, E)$, $c : E \to \mathbb{R}_+$, $s \in V$
Output : Minimaler Spannbaum zwischen allen $v \in V$ (in Array pred)

```
d[v] = ∞ for all v ∈ V \ s;
d[s] = 0;   pred[s] = ⊥;
pq = ⟨⟩;   pq.insert(s, 0);
while ¬pq.empty() do
    v = pq.deleteMin();
    forall the {v, w} ∈ E do
        newWeight = c(v, w);
        if newWeight < d[w] then
            pred[w] = v;
            if d[w] == ∞ then pq.insert(w, newWeight);
            else if w ∈ pq then
                pq.decreaseKey(w, newWeight);
            d[w] = newWeight;
```

Jarnik-Prim-Algorithmus

Laufzeit:

$$\mathcal{O}\Big(n \cdot (T_{\text{insert}}(n) + T_{\text{deletMin}}(n)) + m \cdot T_{\text{decreaseKey}}(n)\Big)$$

Binärer Heap:

- alle Operationen $\mathcal{O}(\log n)$, also
- gesamt: $\mathcal{O}((m+n)\log n)$

Fibonacci-Heap: amortisierte Kosten

- $\mathcal{O}(1)$ für insert und decreaseKey,
- $\mathcal{O}(\log n)$ deleteMin
- gesamt: $\mathcal{O}(m + n\log n)$

Übersicht

Alphabet, Wörter, Wortlänge, Wortmengen

Definition

Ein Alphabet Σ ist eine endliche Menge von Symbolen.

Wörter über Σ sind endliche Folgen von Symbolen aus Σ
(meist $w = w_0 \cdots w_{n-1}$ oder $w = w_1 \cdots w_n$).

Notation:

$\lvert w \rvert$	Länge des Wortes w (Anzahl der Zeichen in w)
ε	leeres Wort (Wort der Länge 0)
Σ^*	Menge aller Wörter über Σ
Σ^+	Menge aller Wörter der Länge ≥ 1 über Σ ($\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$)
Σ^k	Menge aller Wörter über Σ der Länge k

Präfix, Suffix, Teilwort

Definition

$[a : b] := \{n \in \mathbb{Z} \mid a \leq n \wedge n \leq b\}$ für $a, b \in \mathbb{Z}$

Sei $w = w_1 \cdots w_n$ ein Wort der Länge n über Σ, dann heißt

- w' Präfix von w, wenn $w' = w_1 \cdots w_\ell$ mit $\ell \in [0 : n]$
- w' Suffix von w, wenn $w' = w_\ell \cdots w_n$ mit $\ell \in [1 : n+1]$
- w' Teilwort von w, wenn $w' = w_i \cdots w_j$ mit $i, j \in [1 : n]$

Für $w' = w_i \cdots w_j$ mit $i > j$ soll gelten $w' = \epsilon$.

Das leere Wort ε ist also Präfix, Suffix und Teilwort eines jeden Wortes.

Textsuche

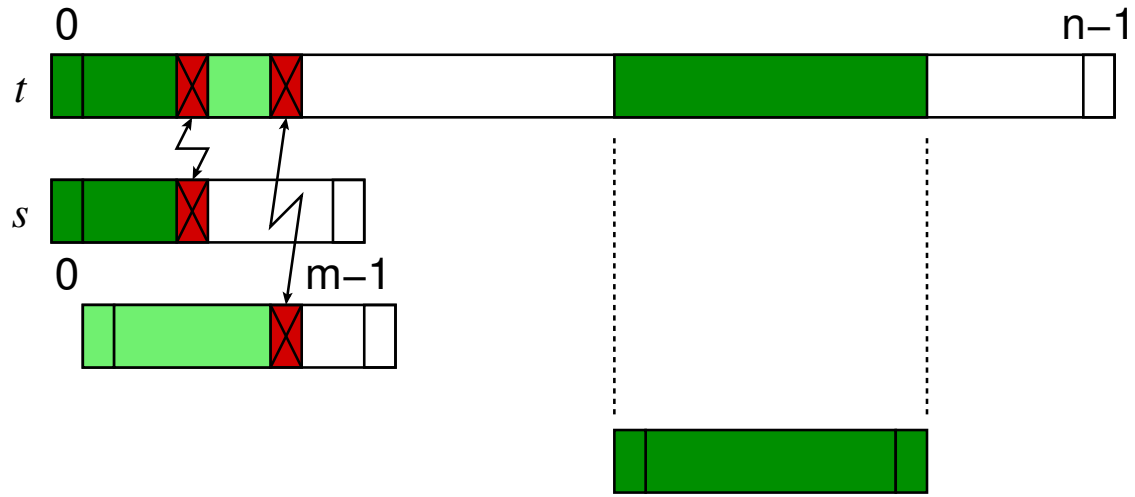
Problem:

Gegeben: Text $t \in \Sigma^*$; $|t| = n$;
Suchwort $s \in \Sigma^*$; $|s| = m \leq n$

Gesucht: $\exists\, i \in [0 : n - m]$ mit $t_i \cdots t_{i+m-1} = s$?
(bzw. alle solchen Positionen i)

Übersicht

Naiver Algorithmus



- Suchwort *s* Zeichen für Zeichen mit Text *t* vergleichen
- Wenn zwei Zeichen nicht übereinstimmen (Mismatch), dann *s* um eine Position „nach rechts“ schieben und erneut *s* mit *t* vergleichen
- Vorgang wiederholen, bis *s* in *t* gefunden wird oder bis klar ist, dass *s* in *t* nicht enthalten ist.

Naiver Algorithmus: Beispiele

t	=	a	a	a	a	a	a	a	a	a	a
		a	a	a	b						
			a	a	a	b					
				a	a	a	b				

t	=	a	a	b	a	a	b	a	a	b	a	a	b	a	a	b
		a	a	b	a	a	a								
			a	a	b	a	a	a							
				a	a	b	a	a	a						
					a	a	b	a	a	a					

Naiver Algorithmus: Implementation

```
bool NaiveSearch (char t[], int n, char s[], int m)
```

```
int i := 0, j := 0;
while (i ≤ n − m) do
    while (t[i + j] = s[j]) do
        j++;
        if (j = m) then
            return TRUE;
    i++;
    j := 0;
return FALSE;
```

Analyse des naiven Algorithmus

- zähle Vergleiche von Zeichen,
- äußere Schleife wird $(n - m + 1)$-mal durchlaufen,
- die innere Schleife wird maximal m-mal durchlaufen.
- maximale Anzahl von Vergleichen: $(n - m + 1)m$,
- Laufzeit: $\mathcal{O}(nm)$

Übersicht

Bessere Idee

- frühere erfolgreiche Vergleiche von zwei Zeichen ausnutzen

- Idee:

  Suchwort so weit nach rechts verschieben, dass in dem Bereich von t, in dem bereits beim vorherigen Versuch erfolgreiche Zeichenvergleiche durchgeführt wurden, nun nach dem Verschieben auch wieder die Zeichen in diesem Bereich übereinstimmen

Rand und eigentlicher Rand

Definition

Ein Wort r heißt Rand eines Wortes w, wenn r sowohl Präfix als auch Suffix von w ist.

Bemerkung: Für jedes Wort w sind damit immer auch das leere Wort ε und w selbst Ränder von w.

Ein Rand r eines Wortes w heißt eigentlicher Rand, wenn $r \neq w$ und wenn es außer w selbst keinen längeren Rand gibt.

Rand und eigentlicher Rand

Beispiel

Das Wort $w = \texttt{aabaabaa}$ besitzt folgende Ränder:

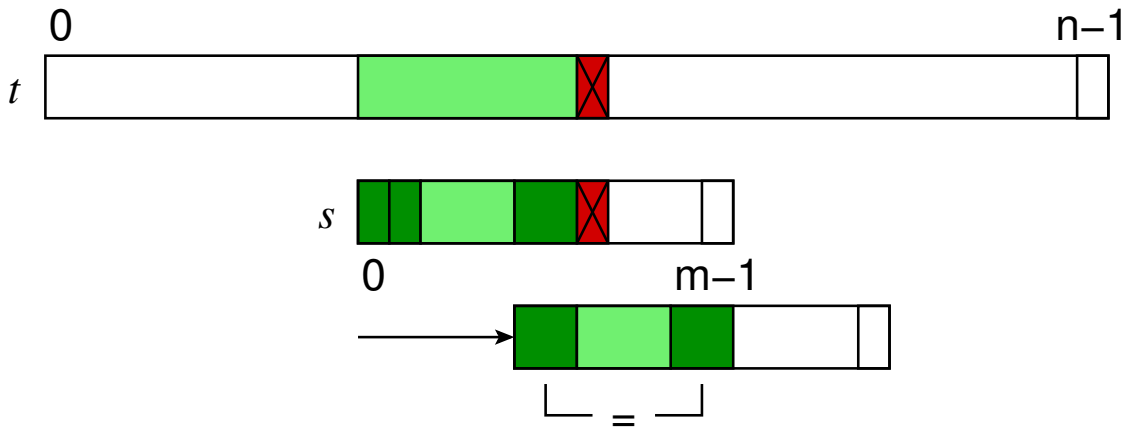
- ε
- $\texttt{a}$
- $\texttt{aa}$
- $\texttt{aabaa}$
- $\texttt{aabaabaa} = w$

Der eigentliche Rand ist $\texttt{aabaa}$.

Man beachte, dass sich bei der Darstellung eines Rands im Wort das entsprechende Präfix und Suffix in der Mitte des Worts überlappen können.

Shift-Idee

- Pattern s so verschieben, dass im bereits gematchten Bereich wieder Übereinstimmung herrscht.
- Dazu müssen überlappendes Präfix und Suffix dieses Bereichs übereinstimmen.



Shifts und sichere Shifts

Definition

Ein Verschiebung der Anfangsposition i des zu suchenden Wortes (d.h. eine Erhöhung des Index $i \to i'$) heißt Shift.

Ein Shift von $i \to i'$ heißt sicherer Shift, wenn s nicht als Teilwort von t an der Position $k \in [i+1 : i'-1]$ vorkommt, d.h. $s \neq t_k \cdots t_{k+m-1}$ für alle $k \in [i+1 : i'-1]$.

- Sinn eines sicheren Shifts: dass man beim Verschieben des Suchworts kein eventuell vorhandenes Vorkommen von s in t überspringt.

Sichere Shifts

Definition

Sei $\partial(s)$ der eigentliche Rand von s und sei

$$border[j] = \begin{cases} -1 & \text{für } j = 0 \\ |\partial(s_0 \cdots s_{j-1})| & \text{für } j \geq 1 \end{cases}$$

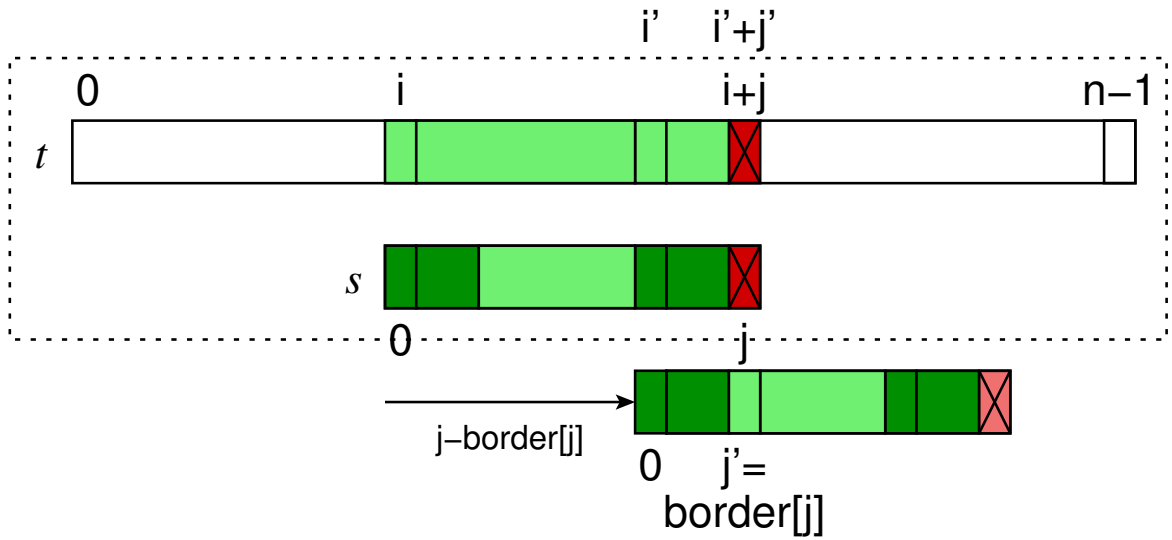
die Länge des eigentlichen Rands des Präfixes der Länge j.

Lemma

Ist das Präfix der Länge j gematcht (also gilt $s_k = t_{i+k}$ für alle $k \in [0 : j-1]$) und haben wir ein Mismatch an der nächsten Position j ($s_j \neq t_{i+j}$), dann ist der Shift $i \to i + j - border[j]$ sicher.

Sichere Shifts

Shift um j – border[j]



Sichere Shifts

Beweis.

- (siehe Skizze)

$$\begin{aligned} s_0 \cdots s_{j-1} &= t_i \cdots t_{i+j-1}, \\ s_j &\neq t_{i+j} \end{aligned}$$

- Der eigentliche Rand von $s_0 \cdots s_{j-1}$ hat die Länge $\mathrm{border}[j]$.
- Verschiebt man s um $j - \mathrm{border}[j]$ nach rechts, so liegt der linke Rand von $s_0 \cdots s_{j-1}$ nun genau da, wo vorher der rechte Rand lag, d.h. im Präfix/Suffix-Überlappungsbereich besteht Übereinstimmung zwischen Präfix, Suffix und Text.
- Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt (außer $s_0 \cdots s_{j-1}$ selbst), ist dieser Shift sicher.

□

KMP-Algorithmus

```
bool KMP(char t[], int n, char s[], int m)
int border[m + 1];
compute_borders(int border[], int m, char s[]);
int i := 0, j := 0;
while i ≤ n − m do
    while t[i + j] = s[j] do
        j++;
        if j = m then
            return TRUE;
    i := i + (j − border[j]) ;          // Es gilt j − border[j] > 0
    j := max{0, border[j]};
return FALSE;
```

Laufzeit des KMP-Algorithmus: erfolglose Vergleiche

Nach erfolglosem Vergleich (Mismatch) wird $(i + j)$ nie kleiner:

- Seien dazu i und j die Werte vor einem erfolglosen Vergleich und i' und j' die Werte nach einem erfolglosen Vergleich.
- Wert vor dem Vergleich: $i + j$
- Wert nach dem Vergleich:
  $i' + j' = (i + j - border[j]) + (max\{0, border[j]\})$.
- Fallunterscheidung: $border[j]$ negativ oder nicht.
  - $border[j] < 0$, also $border[j] = -1$, dann muss $j = 0$ sein.
    Das bedeutet $i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1$.
  - $border[j] \geq 0$, dann gilt $i' + j' = i + j$
- Also wird $i + j$ nach einem erfolglosen Vergleich nicht kleiner.

Laufzeit des KMP-Algorithmus

- Nach jedem erfolglosen Vergleich wird $i \in [0 : n - m]$ erhöht.
- Im Verlaufe des Algorithmus wird i nie erniedrigt wird.

$\Rightarrow$ maximal $n - m + 1$ erfolglose Vergleiche

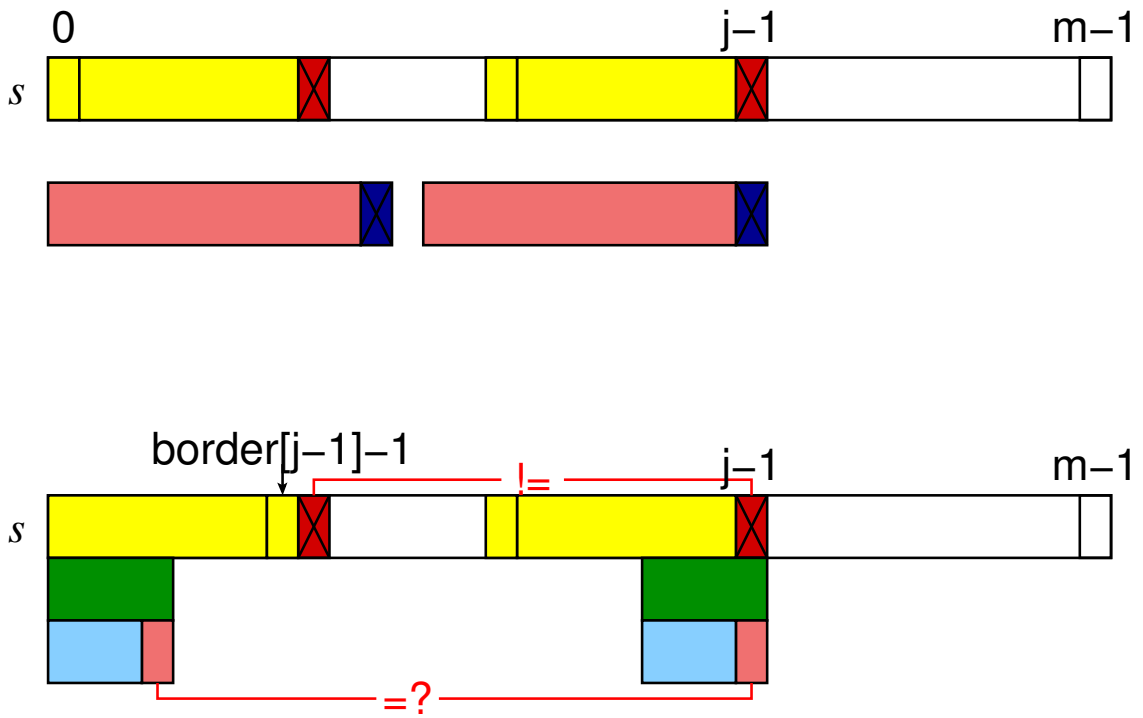
- Nach einem erfolgreichen Vergleich wird $i + j$ um 1 erhöht.
- maximal n erfolgreiche Vergleiche, da $i + j \in [0 : n - 1]$.

- Somit werden insgesamt maximal $2n - m + 1$ Vergleiche ausgeführt.

Berechnung der border-Tabelle

- In der *border*[]-Tabelle wird für jedes Präfix $s_0 \cdots s_{j-1}$ der Länge $j \in [0 : m]$ des Suchstrings s der Länge m gespeichert, wie groß dessen eigentlicher Rand ist.
- Initialisierung: $border[0] = -1$ und $border[1] = 0$
- Annahme: $border[0], \ldots, border[j-1]$ sind schon berechnet
- Ziel: Berechnung von $border[j]$
  (Länge des eigentlichen Randes von Präfix der Länge j)

Berechnung der border-Tabelle



Berechnung der border-Tabelle

- Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$, da nach Definition $s_0 \cdots s_{k-1}$ auch ein Rand von $s_0 \cdots s_{j-2}$ ist (oberer Teil der Abbildung).
- Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j-1] + 1$.
- Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist.
- Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt betrachteten Randes dieses Wortes.
- Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$.

Berechnung der border-Tabelle

- Nun testen wir, ob sich dieser Rand von $s_0 \cdots s_{j-2}$ zu einem eigentlichen Rand von $s_0 \cdots s_{j-1}$ erweitern lässt.
- Dies wiederholen wir solange, bis wir einen Rand gefunden haben, der sich zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt.
- Falls sich kein Rand von $s_0 \cdots s_{j-2}$ zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt, so ist der eigentliche Rand von $s_0 \cdots s_{j-1}$ das leere Wort und wir setzen $\mathit{border}[j] = 0$.

Algorithmus zur Berechnung der border-Tabelle

`compute_borders(int` *border*`[], int` *m*`, char` *s*`[])`

```
border[0] := −1;
border[1] := 0;
int i := 0;
for (int j := 2; j ≤ m; j++) do
    // Beachte, dass hier gilt: i = border[j − 1]
    while (i ≥ 0) && (s[i] ≠ s[j − 1]) do
        i := border[i];
    i++;
    border[j] := i;
```

Laufzeit der Berechnung der border-Tabelle

- Es kann maximal $m-1$ erfolgreiche Vergleiche geben, da jedes Mal $j \in [2:m]$ um 1 erhöht und nie erniedrigt wird.

- Für die Anzahl erfolgloser Vergleiche betrachten wir den Wert i. Zu Beginn ist $i = 0$.
- i wird genau $(m-1)$ Mal um 1 erhöht, da die for-Schleife $(m-1)$ Mal durchlaufen wird.
- Im Fall eines erfolglosen Vergleichs wird i um mindestens 1 erniedrigt.
- i kann maximal $(m-1)+1 = m$ Mal erniedrigt werden, da immer $i \geq -1$ gilt. Es gibt also höchstens m erfolglose Vergleiche.

- Gesamtzahl der Vergleiche $\leq 2m-1$

Laufzeit des KMP-Algorithmus

Theorem

Der Algorithmus von Knuth, Morris und Pratt benötigt maximal $2n + m$ Vergleiche, um festzustellen, ob ein Muster s der Länge m in einem Text t der Länge n enthalten ist.

Der Algorithmus lässt sich leicht derart modifizieren, dass er alle Positionen der Vorkommen von s in t ausgibt, ohne dabei die asymptotische Laufzeit zu erhöhen.

Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt
Fast Pattern Matching in Strings

SIAM Journal on Computing 6(2):323–350, 1977.

Übersicht

Übersicht

Datenkompression

Problem:

- Dateien enthalten oft viel Redundanz (z.B. Wiederholungen) und nehmen mehr Speicherplatz ein als erforderlich

$\Rightarrow$ mit Wissen über die Struktur der Daten und Informationen über die Häufigkeit von Zeichen bzw. Wörtern kann man die Datei so kodieren, dass sie weniger Platz benötigt (Kompression)

Präfixcodes

Definition

Ein Präfixcode (auch *präfixfreier Code*) ist ein Code, bei dem kein Codewort ein Präfix eines anderen Codeworts ist
(kein Codewort taucht als Anfang eines anderen Codeworts auf).

Vorteil:

- $\Rightarrow$ wenn man den codierten Text von vorn abläuft, merkt man sofort, wenn das aktuelle Codewort zu Ende ist
- bei einem Code, der die Präfix-Eigenschaft nicht erfüllt, wird u.U. erst an einer späteren Position klar, welches Codewort weiter vorn im Text gemeint war oder evt. ist die Dekodierung mehrdeutig

Präfixcodes

Beispiele:

- Der Code $\{a \mapsto 0, b \mapsto 01, c \mapsto 10\}$ ist kein Präfixcode, weil das Codewort für a als Präfix des Codeworts für b auftaucht.

  So wäre z.B. unklar, ob 010 für ac oder für ba steht.

  Für 0110 wäre zwar am Ende des Codes klar, dass dieser nur für bc stehen kann, allerdings wäre nach dem Ablaufen der ersten 0 noch nicht klar, ob diese für a steht, oder den Anfang des Codes für b darstellt. Das sieht man erst, nach dem man die folgenden 11 gesehen hat.

- Der Code $\{a \mapsto 0, b \mapsto 10, c \mapsto 11\}$ ist ein Präfixcode, weil kein Codewort als Präfix eines anderen Codeworts auftaucht.

Optimimale Kodierung

Eingabe:

- Wahrscheinlichkeitsverteilung p auf Alphabet A

Ausgabe:

- optimaler Präfixcode

$$f : A \mapsto \{0,1\}^*$$

für die Kodierung von A bei Verteilung p, d.h.
minimale erwartete Codelänge pro Eingabezeichen:

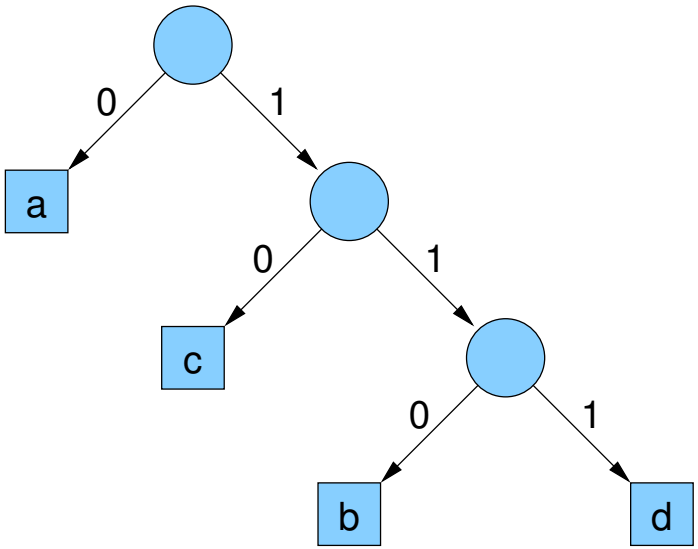
$$\sum_{x \in A} |f(a)| \cdot p(a)$$

Baumdarstellung

Beobachtung:

- Präfixcodes lassen sich als Baum darstellen
- Baumkanten sind mit Zeichen des Codes beschriftet (hier Bits 0 und 1, also Binärbaum)
- an den Blättern stehen die kodierten Zeichen aus A

Beispiel:



Alphabet: $A = \{a, b, c, d\}$

Kodierung:

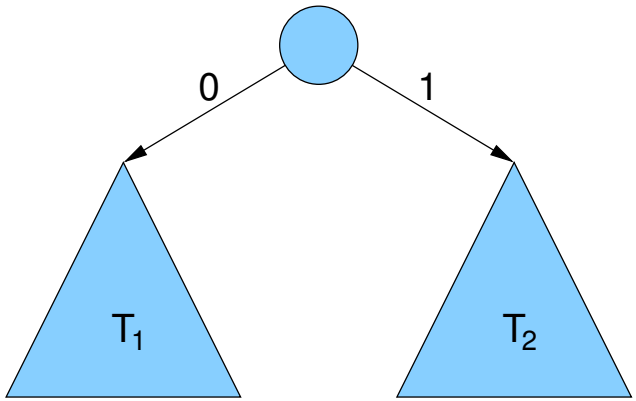
- $f(a) = 0$
- $f(b) = 110$
- $f(c) = 10$
- $f(d) = 111$

Huffman Code

Huffman Code: optimale Kodierung

Strategie:

- anfangs ist jedes Zeichen in A ein Baum für sich (also Wald aus $|A|$ Bäumen)
- Wiederhole bis nur noch ein Baum übrig ist
  - bestimme 2 Bäume T_1 und T_2 mit kleinster Summe ihrer Zeichenwahrscheinlichkeiten $\sum_{a \in T_{1/2}} p(a)$
  - verbinde T_1 und T_2 zu neuem Baum



Huffman Code / Beispiel

Zeichen $x \in A$	a	b	c	d	_
Wahrscheinlichkeit $p(x)$	0,35	0,1	0,2	0,2	0,15