# Teil II

# Grundlagen

# Ziele der Vorlesung

## Wissen:

- Algorithmische Prinzipien verstehen und anwenden
- Grundlegende Algorithmen kennen lernen
- Grundlegende Datenstrukturen kennen lernen
- Bewertung von Effizienz und Korrektheit

## Methodenkompetenz:

- für Entwurf von effizienten und korrekten Algorithmen
- zur Analyse von Algorithmen
- zur Umsetzung auf dem Computer

# Ziele der Vorlesung

## Wissen:

- Algorithmische Prinzipien verstehen und anwenden
- Grundlegende Algorithmen kennen lernen
- Grundlegende Datenstrukturen kennen lernen
- Bewertung von Effizienz und Korrektheit

## Methodenkompetenz:

- für Entwurf von effizienten und korrekten Algorithmen
- zur Analyse von Algorithmen
- zur Umsetzung auf dem Computer

# Übersicht der Inhalte

Grundlagen:

1. **Einführung in Algorithmen und Datenstrukturen**
   Motivation, Definitionen, Einordnung
2. **Grundlagen von Algorithmen**
   Darstellung, elementare Bausteine, Pseudocode
3. **Grundlagen von Datenstrukturen**
   Primitive Datentypen, Felder, abstrakte Datentypen
4. **Grundlagen der Korrektheit von Algorithmen**
   Verifikation, Testen, Sortieren
5. **Grundlagen der Effizienz von Algorithmen**
   Komplexitätsanalyse, Sortieren
6. **Grundlagen des Algorithmen-Entwurfs**
   Entwurfs-Prinzipien

# Übersicht der Inhalte

Fortgeschrittene Algorithmen und Datenstrukturen:

7. **Fortgeschrittene Datenstrukturen**
   Bäume, Graphen, Priority-Queue
8. **Such-Algorithmen**
   Elementare Suchmethoden, Suchbäume
9. **Graph-Algorithmen**
   Elementare Algorithmen, kürzeste Pfade, Spannbaum
10. **Numerische Algorithmen**
    Matrizen-Operationen, Fast Fourier Transform

# Übersicht der Inhalte

Ausgewählte Themen (je nach verfügbarer Zeit):

11. **Datenkompression**
    Huffmann-Codes, JPEG
12. **Kryptographie**
    symmetrische und asymmetrische Verschlüsselungsverfahren

# Was ist ein Algorithmus?

## Duden online:

„Rechenvorgang nach einem bestimmten (sich wiederholenden) Schema“

Beispiele für Algorithmen bereits in der Antike, etwa der Euklidsche Algorithmus zur Berechnung des ggT:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*

# Was ist ein Algorithmus?

## Duden online:

„Rechenvorgang nach einem bestimmten (sich wiederholenden) Schema“

Beispiele für Algorithmen bereits in der Antike, etwa der Euklidsche Algorithmus zur Berechnung des ggT:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*

# Was ist ein Algorithmus?

## M. Broy: Informatik: Eine grundlegende Einführung

„Ein Algorithmus ist ein Verfahren

- mit einer **präzisen** (d.h. in einer genau festgelegten Sprache abgefassten),
- **endlichen** Beschreibung,
- unter Verwendung
  - **effektiver** (d.h. tatsächlich ausführbarer),
  - **elementarer** (Verarbeitungs-) Schritte.“

# Was ist ein Algorithmus?

## H. Rogers:
## Theory of Recursive Functions and Effective Computability

„Ein Algorithmus ist eine

- deterministische Handlungsvorschrift,
- die auf eine bestimmte Klasse von Eingaben angewendet werden kann,
- und für jede dieser Eingaben eine korrespondierende Ausgabe liefert.“

Im weiteren Verlauf des Buches wird mathematische Theorie zur ↑Berechenbarkeit entwickelt

↑theoretische Informatik

# Was ist ein Algorithmus?

## Mathematische Definition Algorithmus

Eine Berechnungsvorschrift zur Lösung eines Problems heißt Algorithmus genau dann, wenn

- eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert,
- die für jede Eingabe, die eine Lösung besitzt, terminiert.

Alan Turing (1936): Turingmaschine als mathematisches Modell eines Computers

↑theoretische Informatik

# 3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

**mathematisch:**

Ein Problem beschreibt eine Funktion $f : E \to A$, mit

$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

**Beispiele:**

# 3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

**mathematisch:**
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

**Beispiele:**

# 3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

**mathematisch:**
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

**Beispiele:**

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position $P$.

# 3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

**mathematisch:**
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

**Beispiele:**

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position $P$.

# 3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

**mathematisch:**
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

**Beispiele:**

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position $P$.

# Algorithmus

Ein Algorithmus ist ein exaktes Verfahren zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus berechnet eine Funktion $f$.

Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain

Abu Abdallah Muhamed ibn Musa al-Chwarizmi, ca. 780–835

# Algorithmus

**Beobachtung:**
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

**Beweisidee:**

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

# Algorithmus

**Beobachtung:**
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

**Beweisidee**:

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

# Algorithmus

Das exakte Verfahren besteht i.a. darin, eine Abfolge von elementaren Einzelschritten der Verarbeitung festzulegen.

**Beispiel:** Alltagsalgorithmen

| *Resultat* | *Algorithmus* | *Einzelschritte* |
|---|---|---|
| Pullover | Strickmuster | eine links, eine rechts, eine fallen lassen |
| Kuchen | Rezept | nimm 3 Eier . . . |
| Konzert | Partitur | Noten |

# Beispiel: Euklidischer Algorithmus

**Problem:** geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\mathrm{ggT}(a, b)$.

**Algorithmus:**

1. Falls $a = b$, brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze $a$ durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze $b$ durch $b - a$ und setze Berechnung in Schritt 1 fort.

# Beispiel: Euklidischer Algorithmus

**Warum geht das?**

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

# Beispiel: Euklidischer Algorithmus

**Warum geht das?**

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a-b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

# Beispiel: Euklidischer Algorithmus

**Warum geht das?**

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a-b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

$$\begin{aligned} a-b &= (q_a - q_b) \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a &= (q'_{a-b} + q'_b) \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

# Beispiel: Euklidischer Algorithmus

Hier sind $q_a, q_b, q'_{a-b}, q'_b \in \mathbb{Z}$.

**Warum geht das?**

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a-b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

$$\begin{aligned} a-b &= (q_a - q_b) \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a &= (q'_{a-b} + q'_b) \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

Das heißt $g$ ist Teiler von $a-b, b$ und $g'$ ist Teiler von $a, b$.

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

# Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

# Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

# Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

# Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

# Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

# Analyse von Algorithmen

**Entscheidende Fragestellungen:**

- Darstellung $\longrightarrow$ Kapitel 2
- Robustheit und Korrektheit $\longrightarrow$ Kapitel 4
- Effizienz und Komplexität $\longrightarrow$ Kapitel 5
- Entwurfstechniken $\longrightarrow$ Kapitel 6

# Definition Datenstruktur

## Definition Datenstruktur (nach Prof. Eckert)

Eine Datenstruktur ist eine

- **logische Anordnung** von Datenobjekten,
- die **Informationen repräsentieren**,
- den **Zugriff** auf die repräsentierte Information über **Operationen** auf Daten ermöglichen und
- die Information **verwalten**.

# Beispiel Datenstruktur

**Stapel** (oder Englisch: **Stack**), z.B. Pizza-Stapel

Operationen:

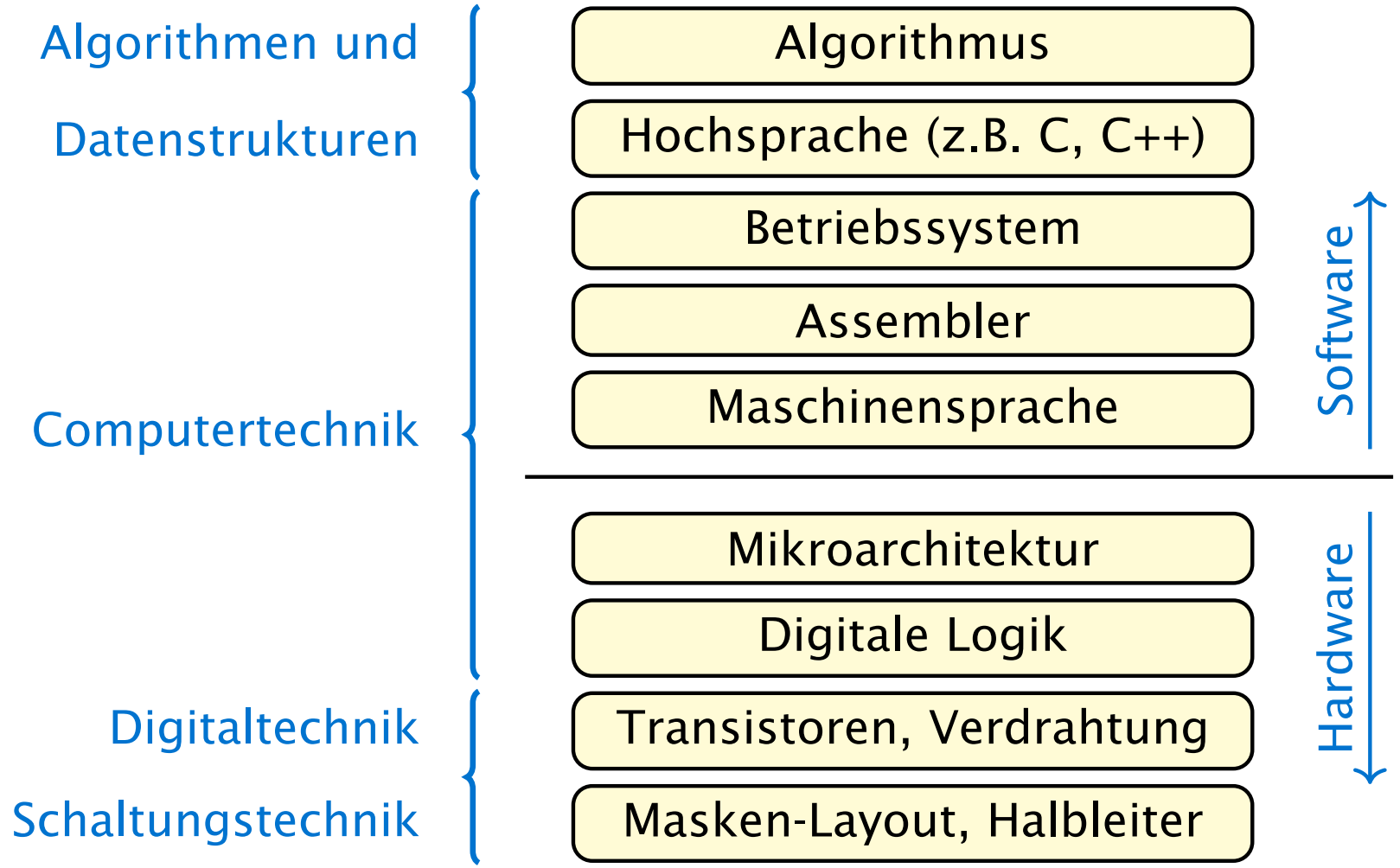
- Element auf Stapel legen – **push**
- Element von Stapel nehmen – **pop**

Operationen jeweils nur auf oberstem Element!

# Weitere Beispiele von Datenstrukturen

- **Felder**, **Listen**, **Stack**, **Queue** $\longrightarrow$ Kapitel 3
- **Bäume**, **Graphen** $\longrightarrow$ Kapitel 7, 8, 9

# Wie funktioniert ein Computer?




*Schema nach Prof. Diepold: Grundlagen der Informatik.*

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

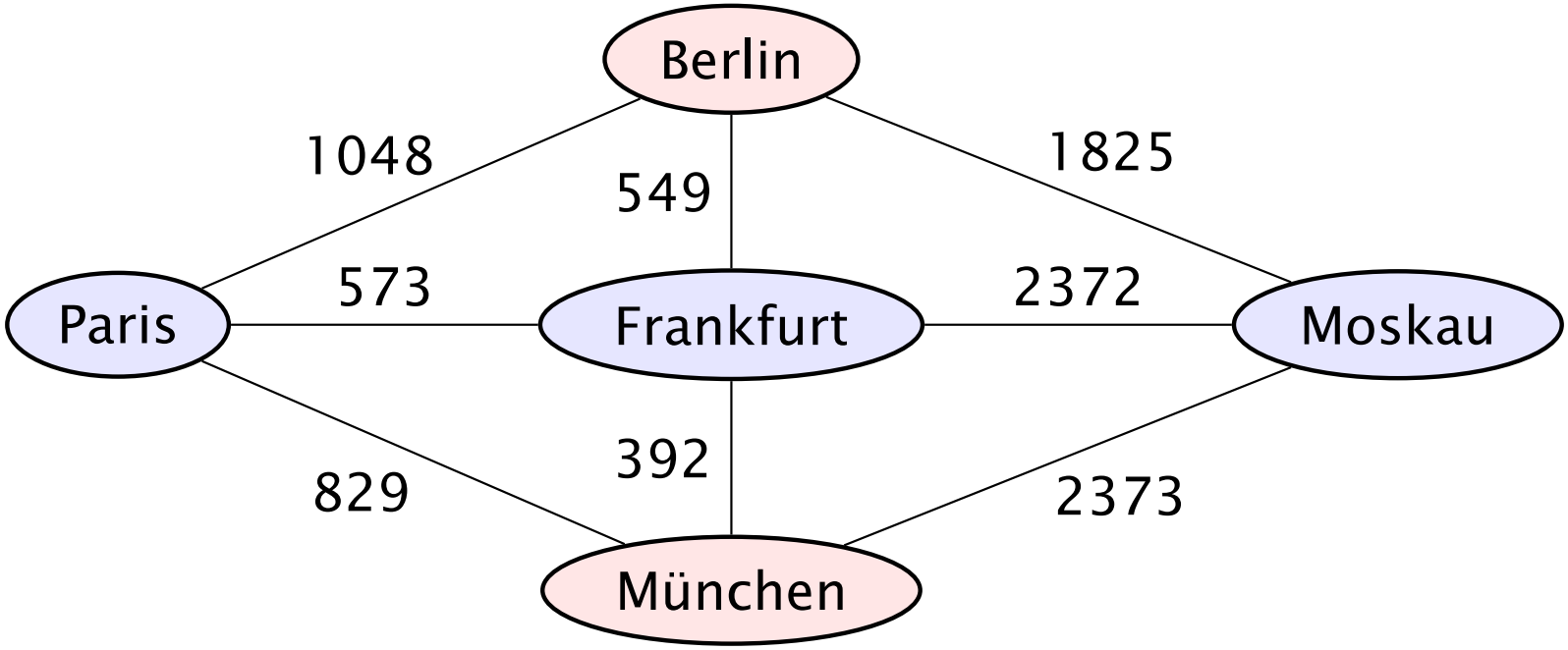
Berlin → → München

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)

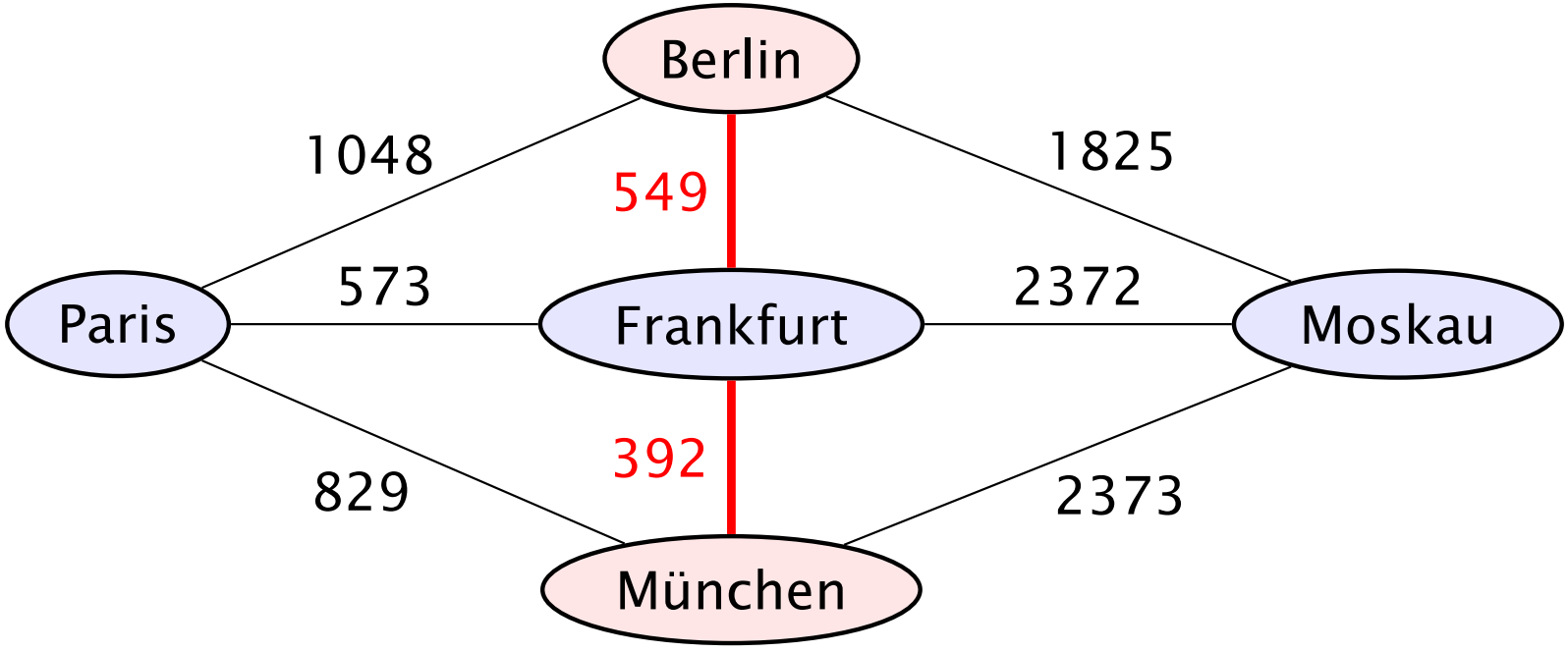


# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)



# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)
- **Aufruf des Programms:** Betriebssystem (z.B. Linux)

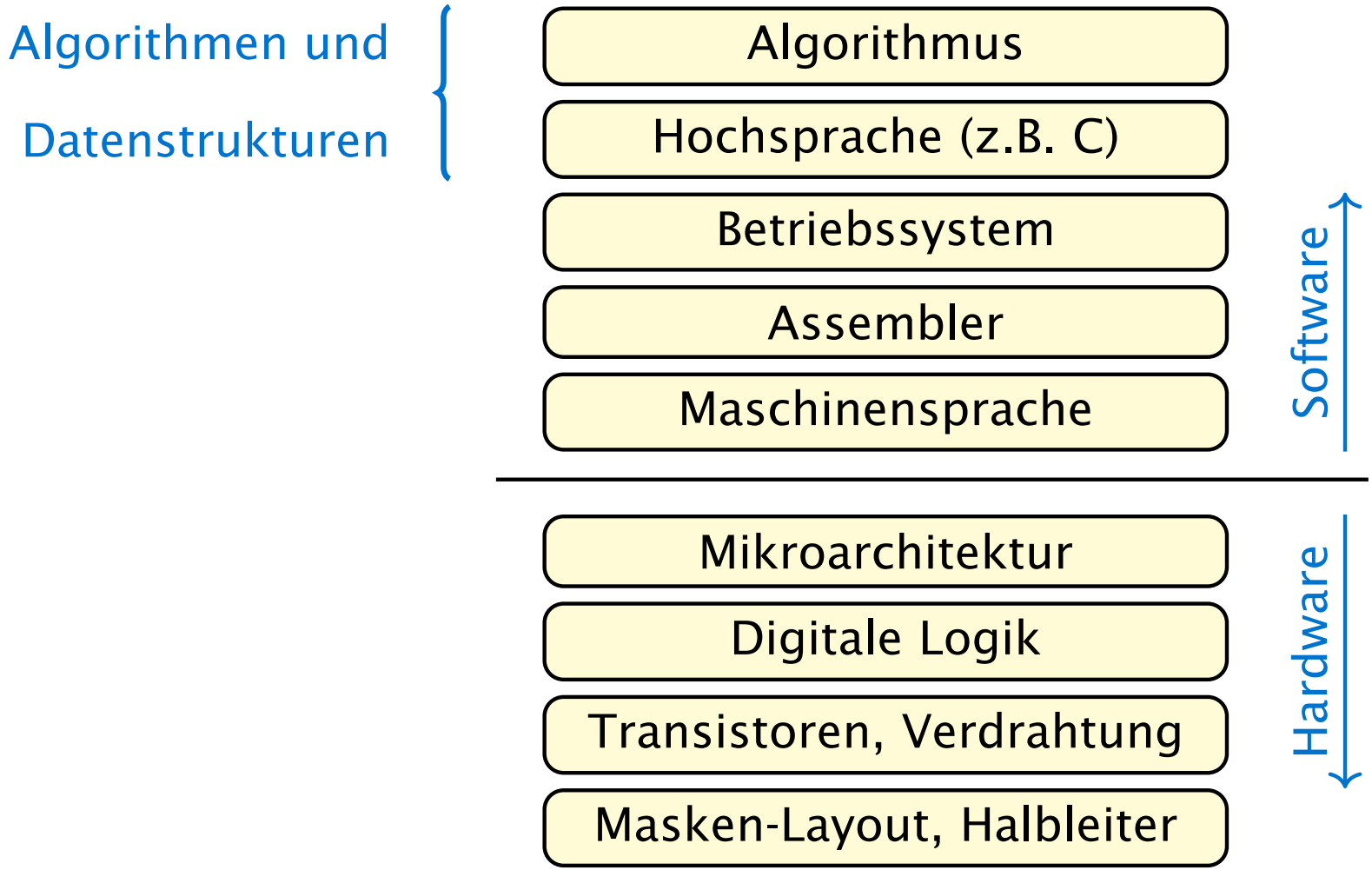
# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)
- **Aufruf des Programms:** Betriebssystem (z.B. Linux)
- **Ausführung des Programms:** Computer (z.B. Laptop)

# Einordnung Algorithmen und Datenstrukturen




*Schema nach Prof. Diepold: Grundlagen der Informatik.*

# Wie beschreibt man Algorithmen?

**Algorithmus: bestimme Maximum von zwei Zahlen**

- Input: Zahlen $a, b$
- Output: Zahl $x = \max(a, b)$

**Problem:** präzise Beschreibung der Schritte

# Wie beschreibt man Algorithmen?

**Algorithmus: bestimme Maximum von zwei Zahlen**

- Input: Zahlen $a, b$
- Output: Zahl $x = \max(a, b)$

**Problem:** präzise Beschreibung der Schritte

**Lösung:** Pseudocode

```
Algorithmus: max(a,b)
Input:  a,b
   x=a
   Falls b>a dann
     x=b
   Ende Falls
Output:  x
```

# Darstellung von Algorithmen I

## Pseudocode

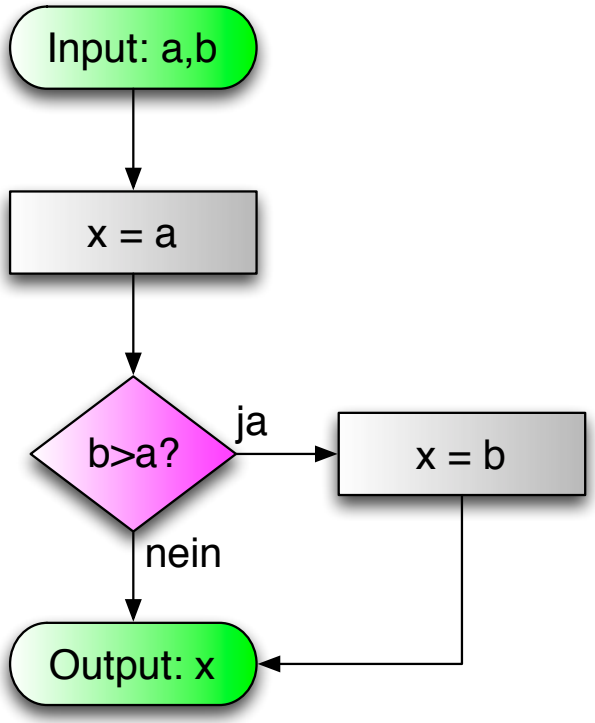
- informelle Veranschaulichung von Algorithmus
- nicht von Rechner ausführbar
- nicht standardisiert

```
Algorithmus: max(a,b)
Input:  a,b
   x=a
   Falls b>a dann
     x=b
   Ende Falls
Output:  x
```

# Darstellung von Algorithmen II

## Flussdiagramm

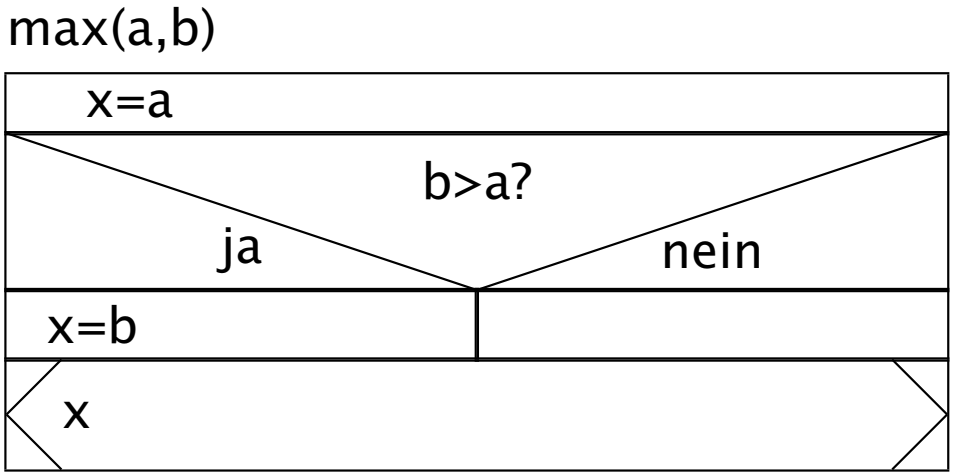
- graphische Darstellung als Ablaufdiagramm, nicht ausführbar
- normiert als DIN 66001



# Darstellung von Algorithmen III

## Struktogramm

- Diagramm zur Strukturdarstellung, nicht ausführbar
- eingeführt von Nassi/Shneiderman 1973, normiert als DIN 66261



# Darstellung von Algorithmen IV

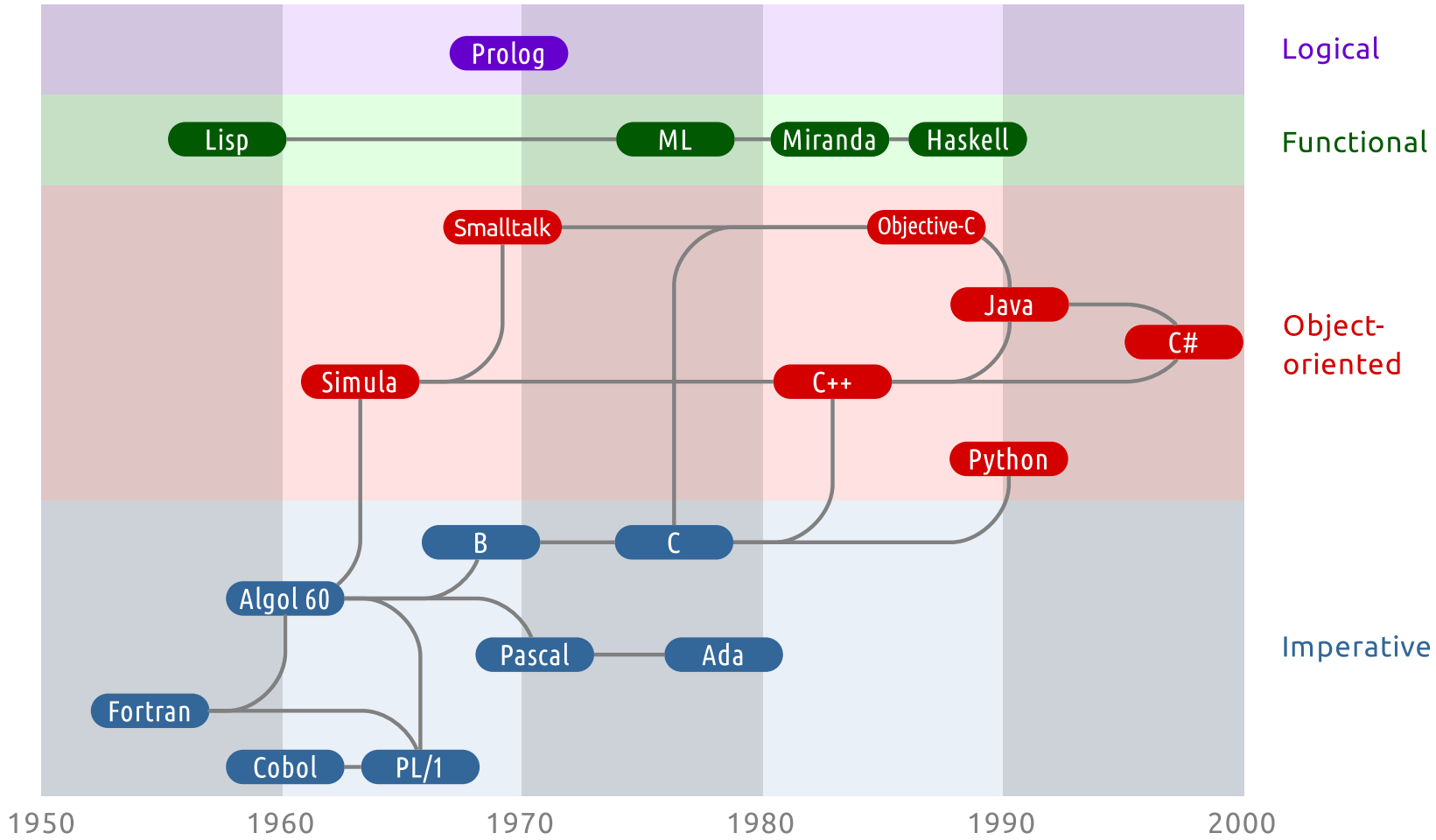
## Programmiersprache

- formale Sprache zur Beschreibung von Algorithmen
- fest definierte Syntax
- ein Compiler/Interpreter wandelt Programm in ausführbare Form für Rechner um
- Beispiele: Assembler, C, Java

- Algorithmus in C:

```
1 int max(int a, int b) {
2     int x = a;
3     if (b > a)
4         x = b;
5     return x;
6 }
```

# Programmiersprachen Übersicht




*Grafik von Alexandru Duliu.*

# Äquivalenz von Algorithmen-Beschreibungen

## Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

Die fehlende Zutat für eine präzise Algorithmendefinition ist die Definition eines *elementaren Einzelschrittes*. Dies wird üblicherweise durch ein Maschinenmodells gemacht (z.B. Turingmaschine). Es gibt auch sehr esoterische Maschinenmodelle (z.B. *billiard ball computer*).

- alle gängigen Programmiersprachen leisten dasselbe
- jeder Computer ist äquivalent

Die These postuliert, dass alle diese Maschinenmodelle equivalent ist.

- formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

# Äquivalenz von Algorithmen-Beschreibungen

## Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

Die fehlende Zutat für eine präzise Algorithmendefinition ist die Definition eines *elementaren Einzelschrittes*. Dies wird üblicherweise durch ein Maschinenmodells gemacht (z.B. Turingmaschine). Es gibt auch sehr esoterische Maschinenmodelle (z.B. *billiard ball computer*).

- alle gängigen Programmiersprachen leisten dasselbe
- jeder Computer ist äquivalent

Die These postuliert, dass alle diese Maschinenmodelle equivalent ist.

- formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

# Bausteine von Algorithmen

## Elementare Bausteine

„Normale“ Algorithmen lassen sich mit

**vier elementaren Bausteinen**

darstellen:

1. Elementarer Verarbeitungsschritt (z.B. Zuweisung an Variable)
2. Sequenz (elementare Schritte nacheinander)
3. Bedingter Verarbeitungsschritt (z.B. if/else)
4. Wiederholung (z.B. while-Schleife)

# 1. Elementarer Verarbeitungsschritt

**Beispiele**

- `a = a - b // weist Variable a den Wert a-b zu`
- `return a // liefert den Wert von a zurueck`

Achtung: manche Verarbeitungsschritte sehen elementar aus, sind es aber nicht!

- `sortiere Liste L // nicht elementar`
- `finde kuerzesten Pfad in G // nicht elementar`

# 2. Sequenz

Sequenz ist eine Aneinanderreihung von elementaren Verarbeitungsschritten

Abgrenzung der Schritte mittels **Semikolon (;)**

**Beispiel**

- `x = 5; // Zuweisung von Wert 5 an Variable x`
- `x = x + 2; // Wert von x  ist nun 7`

Um Ausnahmen zu vermeiden, wird Semikolon auch verwendet, wenn kein weiterer Schritt folgt

# 3. Bedingter Verarbeitungsschritt

Ausführung des Verarbeitungsschrittes nur wenn Bedingung erfüllt ist

**Beispiele:**

- ```
  if (a > b) // Bedingung wird in Klammern notiert
      a = a - b;
  ```
- ```
  if (a > b)
      a = a - b;
  else // falls Bedingung nicht erfuellt
      b = b - a;
  ```

Einrückung verdeutlicht logische Ebenen

# 3. Bedingter Verarbeitungsschritt

falls mehr als ein Verarbeitungsschritt bedingt ausgeführt werden soll, Markierung durch einen Block **{ ... }** mit geschweiften Klammern

**Beispiel**

```
if (x == 0) {
    x = 5;
    x = x + 2;
} // if Block ist hier zu Ende
else {
    x = x - 1;
} // else Block ist hier zu Ende
```

auch einzelne Schritte können in einen Block gefasst werden

# 4. Wiederholung

wiederholte Ausführung von Verarbeitungsschritt/Block solange Bedingung erfüllt ist (auch while Schleife genannt)

**Beispiele**

- ```
  while (x != 0) // Bedingung in Klammern
      x = x - 1;
  ```
- ```
  while (b > 0) { // Block fuer mehrere Schritte
      if (a > b)
          a = a - b;
      else
          b = b - a;
  } // while Block ist hier zu Ende
  ```

# 4. Wiederholung

Es gibt auch andere Schleifentypen: do-while Schleife:

```
do {
    x = x - 1;
} while (x != 0); // Vorsicht by floats!!!
```

for-Schleife:

```
for i=1 to 10
    print(i); // gibt Wert von i aus
```

Achtung, Syntax der for-Schleife ist in C komplexer!

```
for (i=1; i <= 10; i++) // echte C Syntax
    print(i);
```

# Beispiel: Euklidscher Algorithmus

- Einrücken oder geschweifte Klammern {, } kennzeichnen **Blockstruktur**

```
1  euklid(a,b)
2      if (a == 0)
3          return b;
4      while (b > 0) {
5          if (a > b)
6              a = a - b;
7          else
8              b = b - a;
9      }
10     return a;
```

Euklidscher Algorithmus

# Beispiel: Euklidscher Algorithmus

- Einrücken oder geschweifte Klammern {, } kennzeichnen **Blockstruktur**
- In C so nicht möglich

```
1 euklid(a,b)
2     if (a == 0)
3         return b;
4     while (b > 0)
5         if (a > b)
6             a = a - b;
7         else
8             b = b - a;
9     return a;
```

Euklidscher Algorithmus

# Euklidscher Algorithmus

**Input:** Natürliche Zahlen $a, b$

**Output:** $\mathrm{ggT}(a, b)$

1. Falls $a = 0$ liefere $b$ zurück
2. Solange $b > 0$ wiederhole

   Falls $a > b$ setze $a = a - b$

   sonst setze $b = b - a$
3. Liefere $a$ zurück

# Euklidscher Algorithmus

**Input:** Natürliche Zahlen $a, b$

**Output:** $\mathrm{ggT}(a, b)$

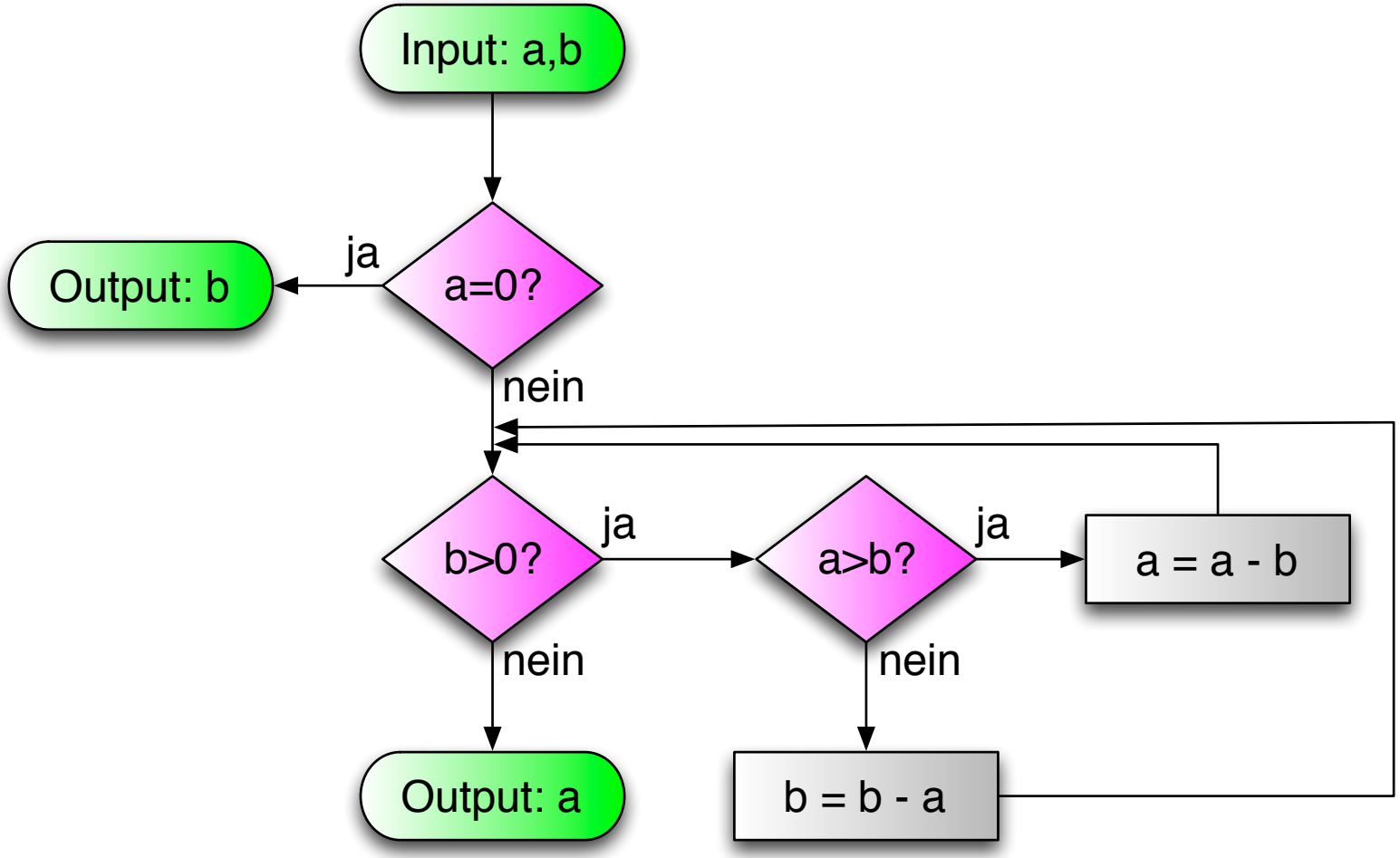
1. Falls $a = 0$ liefere $b$ zurück
2. Solange $b > 0$ wiederhole

   Falls $a > b$ setze $a = a - b$

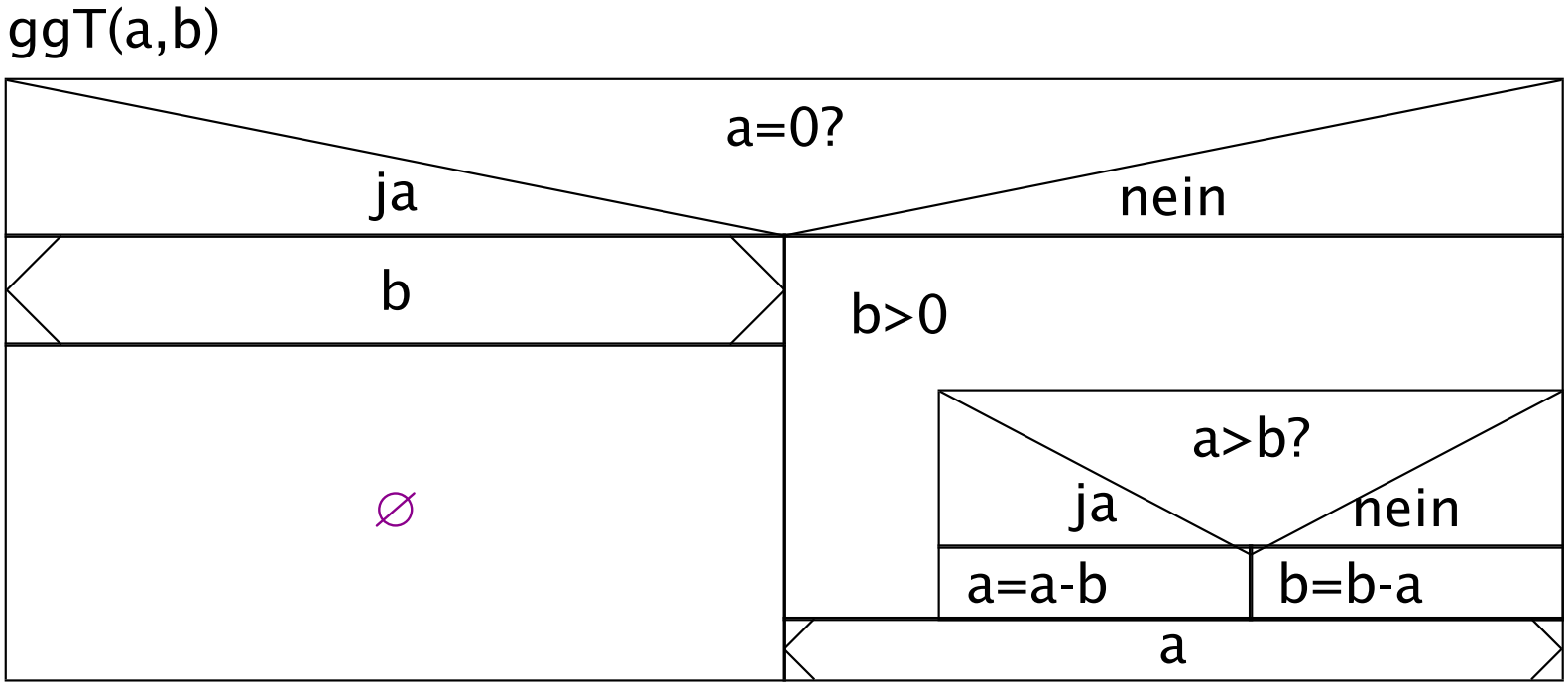
   sonst setze $b = b - a$
3. Liefere $a$ zurück

**⟶ das ist Pseudocode!**

# Euklidscher Algorithmus als Flussdiagramm



# Euklidscher Algorithmus als Struktogramm



# Euklidscher Algorithmus als Pseudocode

```
1  Input: natuerliche Zahlen a, b
2  Output: ggT(a,b)
3  euklid(a,b)
4      if (a == 0)
5          return b;
6      while (b > 0) { // Hauptschleife
7          if (a > b)
8              a = a - b;
9          else
10             b = b - a;
11     }
12     return a;
```

Euklidscher Algorithmus

# Euklidscher Algorithmus als C

```
1  int ggT(int a, int b)
2  {
3      if (a==0)
4          return b;
5      while (b>0) {
6          if (a>b)
7              a = a - b;
8          else
9              b = b - a;
10     }
11     return a;
12 }
```

# Euklidscher Algorithmus als Python

```
1 def ggT(a, b):
2     if a == 0:
3         return b
4     while b > 0:
5         if a > b:
6             a = a - b
7         else:
8             b = b - a
9     return a
```

# Darstellung von Algorithmen in der Vorlesung

viele Möglichkeiten der Darstellung!

- alle vernünftigen Darstellungen sind äquivalent
- jede Darstellung hat Vor- und Nachteile

für die Vorlesung: **Pseudocode im C Stil**

Zusatzmaterial für viele Beispiele aus der Vorlesung:

- `http://www.brpreiss.com/books/opus7/`
- Beispiele in:
  - Python
  - C++
  - Java
  - C#
  - und vieles mehr...

# Beispiel: Fibonacci Zahlen

## Fibonacci Folge

Die **Fibonacci Folge** ist eine Folge natürlicher Zahlen $f_1, f_2, f_3, \ldots$, für die gilt

$$f_n = f_{n-1} + f_{n-2} \qquad \text{für } n \geq 3$$

mit Anfangswerten $f_1 = 1$, $f_2 = 1$.

- eingesetzt von Leonardo Fibonacci zur Beschreibung von Wachstum einer Kaninchenpopulation
- Folge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- berechenbar z.B. via Rekursion

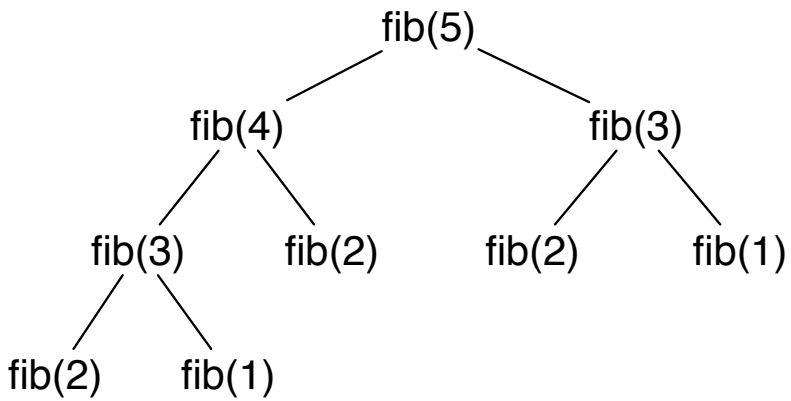
# Beispiel: Fibonacci Funktion

```
Input: Index n der Fibonaccifolge
Output: Wert f_n
fib(n)
   if (n == 1 || n == 2) {
      return 1;
   }
   else {
      // rekursiver Aufruf
      return fib(n-1) + fib(n-2);
   }
```

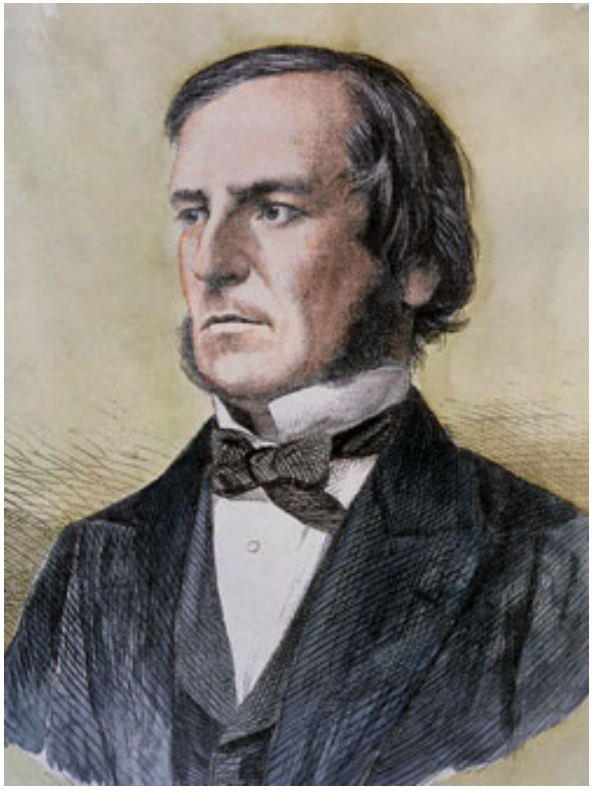
# Beispiel: Fibonacci Funktion

```
Input: Index n der Fibonaccifolge
Output: Wert f_n
fib(n)
   if (n == 1 || n == 2) {
      return 1;
   }
   else {
      // rekursiver Aufruf
      return fib(n-1) + fib(n-2);
   }
```

Aufrufstruktur für fib(5):



# George Boole



Englischer Mathematiker (1815-1864)

# Logische Werte

**Boolesche Logik**: Logik mit zwei Werten

Repräsentationen:

- 1 und 0
- W und F (in Englisch: T und F)
- L und O

Mengensymbol $\mathbb{B}$

- $\mathbb{B} = \{0, 1\} = \{F, W\} = \{O, L\}$

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:** $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

**Disjunktion:** $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

**Negation:** $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:** $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

**Disjunktion:** $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

**Negation:** $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

**Wahrheitstabelle:**

| $a$ | $b$ | $a \wedge b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:** $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

**Disjunktion:** $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

**Negation:** $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

**Wahrheitstabelle:**

| $a$ | $b$ | $a \wedge b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $a$ | $b$ | $a \vee b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:** $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

**Disjunktion:** $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

**Negation:** $\neg : \mathbb{B} \rightarrow \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

**Wahrheitstabelle:**

| $a$ | $b$ | $a \wedge b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $a$ | $b$ | $a \vee b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $a$ | $\neg a$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

# Weitere Verknüpfungen I

**NAND:** $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \uparrow b = \neg(a \wedge b)$
- mit NAND lassen sich NOT, OR, AND erzeugen

**NOR:** $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \downarrow b = \neg(a \vee b)$
- mit NOR lassen sich ebenso NOT, OR, AND erzeugen

**XOR:** $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- auch exklusiv oder genannt
- erzeugbar aus $\neg(a \wedge b) \wedge (a \vee b)$ (siehe Übung)

**Wahrheitstabelle:**

| $a$ | $b$ | $a \uparrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Weitere Verknüpfungen I

**NAND:** $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \uparrow b = \neg(a \wedge b)$
- mit NAND lassen sich NOT, OR, AND erzeugen

**NOR:** $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \downarrow b = \neg(a \vee b)$
- mit NOR lassen sich ebenso NOT, OR, AND erzeugen

**XOR:** $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- auch exklusiv oder genannt
- erzeugbar aus $\neg(a \wedge b) \wedge (a \vee b)$ (siehe Übung)

**Wahrheitstabelle:**

| $a$ | $b$ | $a \uparrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $a$ | $b$ | $a \downarrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

# Weitere Verknüpfungen I

**NAND:** $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \uparrow b = \neg(a \wedge b)$
- mit NAND lassen sich NOT, OR, AND erzeugen

**NOR:** $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \downarrow b = \neg(a \vee b)$
- mit NOR lassen sich ebenso NOT, OR, AND erzeugen

**XOR:** $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- auch exklusiv oder genannt
- erzeugbar aus $\neg(a \wedge b) \wedge (a \vee b)$ (siehe Übung)

**Wahrheitstabelle:**

| $a$ | $b$ | $a \uparrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $a$ | $b$ | $a \downarrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| $a$ | $b$ | $a \oplus b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Weitere Verknüpfungen II

**Implikation:** $\Rightarrow: \ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- oft verwendet für mathematische Sätze:
  „$a$ impliziert $b$“, „aus $a$ folgt $b$“
- Beispiel: „aus $n < 3$ folgt $n < 5$“
- erzeugbar aus $\neg a \vee b$

**Wahrheitstabelle:**

| $a$ | $b$ | $a \Rightarrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Äquivalenz:** $\Leftrightarrow: \ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- oft verwendet für mathematische Sätze:
  „$a$ gilt genau dann, wenn $b$ gilt“,
  „$a$ und $b$ sind äquivalent“
- Beispiel: „$f$ ist bijektiv genau dann,
  wenn $f$ injektiv und surjektiv ist“
- erzeugbar aus $(a \wedge b) \vee (\neg a \wedge \neg b)$

# Weitere Verknüpfungen II

**Implikation:** $\Rightarrow: \ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- oft verwendet für mathematische Sätze:
  „$a$ impliziert $b$“, „aus $a$ folgt $b$“
- Beispiel: „aus $n < 3$ folgt $n < 5$“
- erzeugbar aus $\neg a \vee b$

**Wahrheitstabelle:**

| $a$ | $b$ | $a \Rightarrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Äquivalenz:** $\Leftrightarrow: \ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- oft verwendet für mathematische Sätze:
  „$a$ gilt genau dann, wenn $b$ gilt“,
  „$a$ und $b$ sind äquivalent“
- Beispiel: „$f$ ist bijektiv genau dann,
  wenn $f$ injektiv und surjektiv ist“
- erzeugbar aus $(a \wedge b) \vee (\neg a \wedge \neg b)$

| $a$ | $b$ | $a \Leftrightarrow b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Rangfolge und Rechenregeln

**Rangfolge:**

- NICHT vor UND
- UND vor ODER

**Beispiel**

$$\neg 0 \vee 1 \wedge 0 = (\neg 0) \vee (1 \wedge 0) = 1 \vee 0 = 1$$

**De Morgan-Gesetze:**

- $\neg(a \wedge b) = \neg a \vee \neg b$
- $\neg(a \vee b) = \neg a \wedge \neg b$

# Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

**Beispiele:**

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

**Beispiele:**

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

**Beispiele:**

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

**Beispiele:**

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Was sind primitive Datentypen?

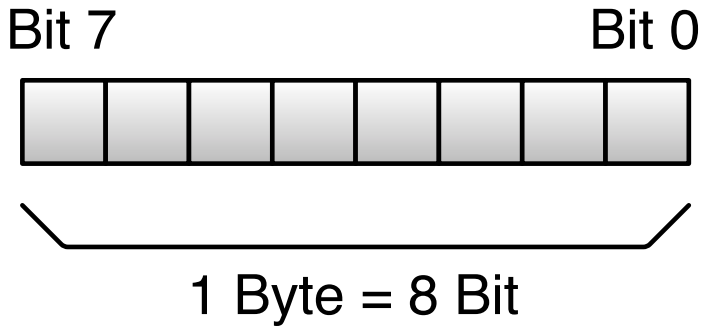
**Primitive Datentypen**
Wir bezeichnen grundlegende, in Programmiersprachen eingebaute Datentypen als primitive Datentypen.

Durch Kombination von primitiven Datentypen lassen sich zusammengesetzte Datentypen bilden.

Beispiele für primitive Datentypen in C:

- `int` für ganze Zahlen
- `float` für floating point Zahlen
- `bool` für logische Werte

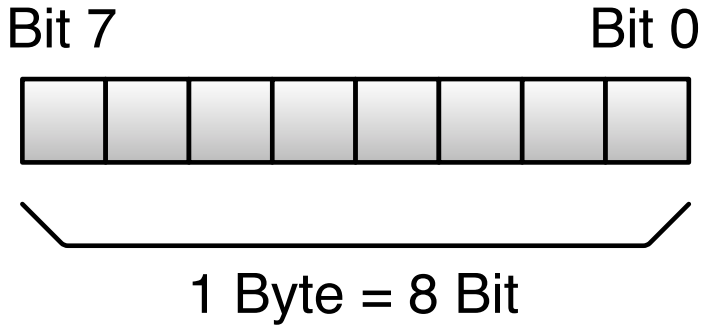
# Bits und Bytes




Bytes als Maßeinheit für Speichergrössen (nach IEC, traditionell):

- $2^{10}$ Bytes $= 1024$ Bytes $= 1$ KiB, ein Kilo Byte (Kibi Byte)
- $2^{20}$ Bytes $= 1$ MiB, ein Mega Byte (bzw. MebiByte)
- $2^{30}$ Bytes $= 1$ GiB, ein Giga Byte (bzw. GibiByte)
- $2^{40}$ Bytes $= 1$ TiB, ein Tera Byte (bzw. TebiByte)
- $2^{50}$ Bytes $= 1$ PiB, ein Peta Byte (bzw. PebiByte)
- $2^{60}$ Bytes $= 1$ EiB, ein Exa Byte (bzw. ExbiByte)

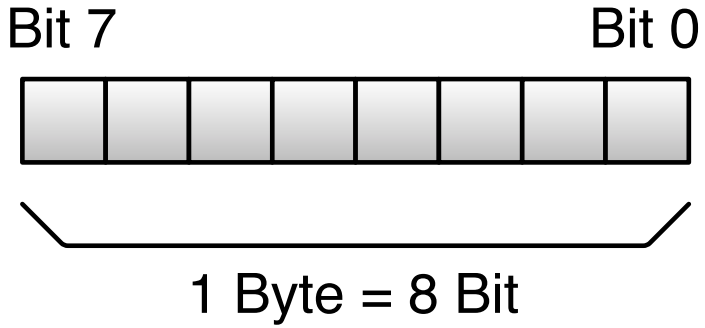
# Bits und Bytes




Bytes als Maßeinheit für Speichergrössen (nach IEC, metrisch):

- $10^3$ Bytes $= 1000$ Bytes $= 1$ kB, ein kilo Byte (großes B)
- $10^6$ Bytes $= 1$ MB, ein Mega Byte
- $10^9$ Bytes $= 1$ GB, ein Giga Byte
- $10^{12}$ Bytes $= 1$ TB, ein Tera Byte
- $10^{15}$ Bytes $= 1$ PB, ein Peta Byte
- $10^{18}$ Bytes $= 1$ EB, ein Exa Byte

# Bits und Bytes




Bytes als Maßeinheit für Speichergrössen (nach IEC, metrisch):

- $10^3$ Bytes $= 1000$ Bytes $= 1$ kB, ein kilo Byte (großes B)
- $10^6$ Bytes $= 1$ MB, ein Mega Byte
- $10^9$ Bytes $= 1$ GB, ein Giga Byte
- $10^{12}$ Bytes $= 1$ TB, ein Tera Byte
- $10^{15}$ Bytes $= 1$ PB, ein Peta Byte
- $10^{18}$ Bytes $= 1$ EB, ein Exa Byte

**Hinweis:** auch Bits werden als Maßangabe verwendet, z.B. 16 Mbit oder 16 Mb (kleines b).

# Primitive Datentypen in C-ähnlichen Sprachen

Wir betrachten im Detail primitive Datentypen für:

1. natürliche Zahlen *(unsigned integers)*
2. ganze Zahlen *(signed integers)*
3. floating point Zahlen *(floats)*

# Zahldarstellung

**Dezimalsystem:**

- Basis $b = 10$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7,8,9\}$
- Beispiel: $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

**Binärsystem:**

- Basis $b = 2$
- Koeffizienten $c_n \in \{0,1\}$
- Beispiel: $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$

# Zahldarstellung

**Dezimalsystem:**

- Basis $b = 10$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7,8,9\}$
- Beispiel: $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

**Binärsystem:**

- Basis $b = 2$
- Koeffizienten $c_n \in \{0,1\}$
- Beispiel: $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$

# Zahldarstellung

**Oktalsystem:**

- Basis $b = 8\ (= 2^3)$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7\}$
- Beispiel: $173_8 = 1 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 = 123_{10}$

**Hexadezimalsystem:**

- Basis $b = 16\ (= 2^4)$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- Beispiel: $7B_{16} = 7 \cdot 16^1 + B \cdot 16^0 = 123_{10}$

# Wie viele Ziffern pro Zahl?

**Problem**

Gegeben Zahl $z \in \mathbb{N}$, wie viele Ziffern $m$ werden bezüglich Basis $b$ benötigt?

**Lösung:** $m = \lfloor \log_b(z) \rfloor + 1$

**Erläuterung:** ($a \in \mathbb{R}$)

- $\lfloor a \rfloor = \text{floor}(a) =$ größte ganze Zahl kleiner gleich $a$
- $\lceil a \rceil = \text{ceil}(a) =$ kleinste ganze Zahl größer gleich $a$

$$a - 1 < \lfloor a \rfloor \leq a \leq \lceil a \rceil < a + 1$$

- $\log_b(z) = \frac{\ln(z)}{\ln(b)}$, wobei „ln“ der natürliche Logarithmus ist

# Wie viele Ziffern pro Zahl?

**Lösung:** $m = \lfloor \log_x(z) \rfloor + 1$

**Beispiele:** $z = 123$

- Basis $b = 10$: $m = \lfloor \log_{10}(123) \rfloor + 1 = \lfloor 2.0899\ldots \rfloor + 1 = 3$
- Basis $b = 2$: $m = \lfloor \log_2(123) \rfloor + 1 = \lfloor 6.9425\ldots \rfloor + 1 = 7$
- Basis $b = 8$: $m = \lfloor \log_8(123) \rfloor + 1 = \lfloor 2.3141\ldots \rfloor + 1 = 3$
- Basis $b = 16$: $m = \lfloor \log_{16}(123) \rfloor + 1 = \lfloor 1.7356\ldots \rfloor + 1 = 2$

# Größte Zahl pro Anzahl Ziffern?

**Problem**

Gegeben Basis $b$ und $m$ Ziffern, was ist die größte darstellbare Zahl?

**Lösung:** $z_{max} = x^m - 1$

**Beispiele:**

- $b = 2, m = 4$: $z_{max} = 2^4 - 1 = 15 = 1111_2$
- $b = 2, m = 8$: $z_{max} = 2^8 - 1 = 255 = 11111111_2$
- $b = 16, m = 2$: $z_{max} = 16^2 - 1 = 255 = FF_{16}$

# Natürliche Zahlen in C-ähnlichen Sprachen

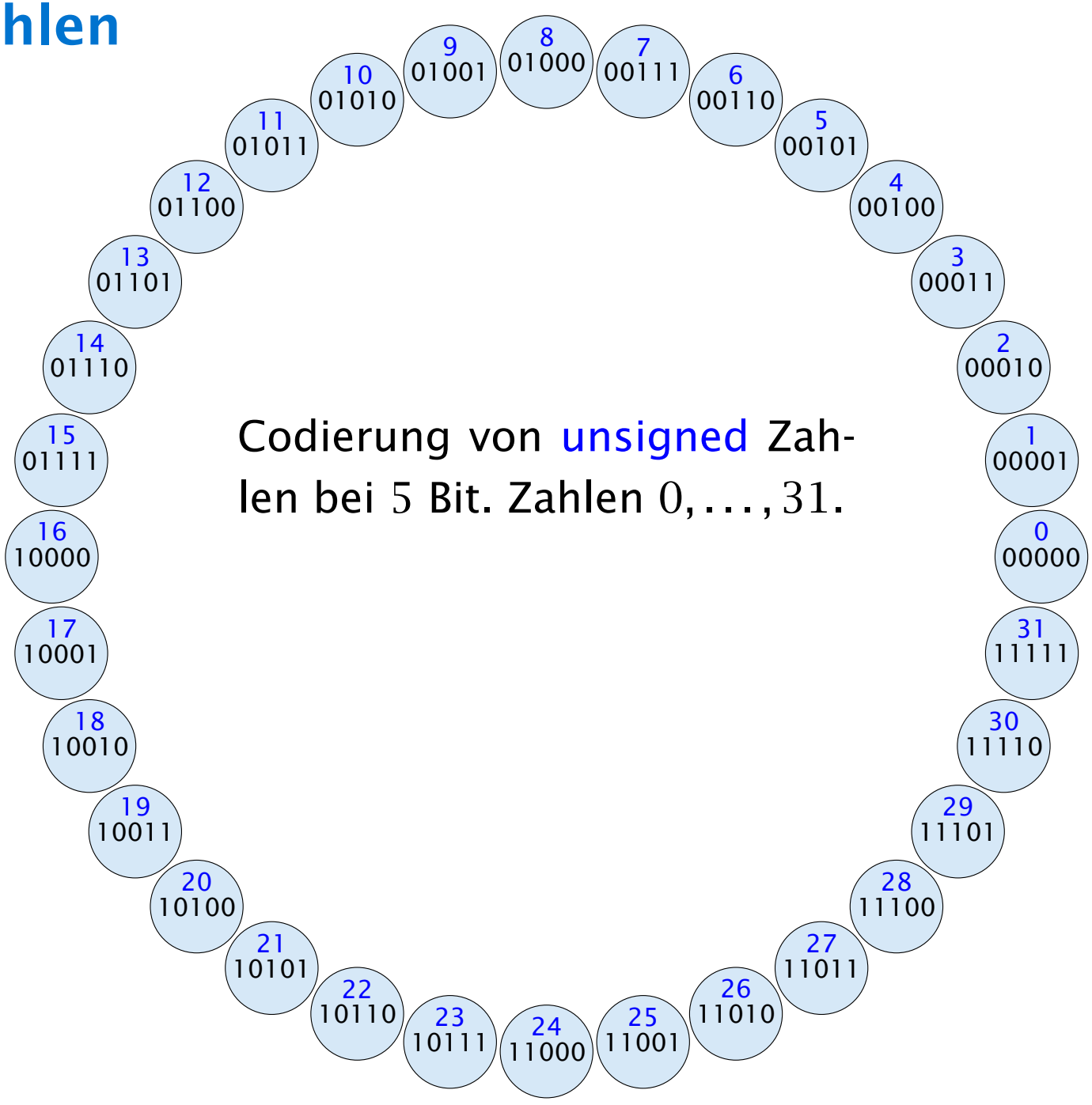
**Natürliche Zahlen**

In Computern verwendet man Binärdarstellung mit einer fixen Anzahl Ziffern (genannt Bits).

Die primitiven Datentypen für natürliche Zahlen sind:

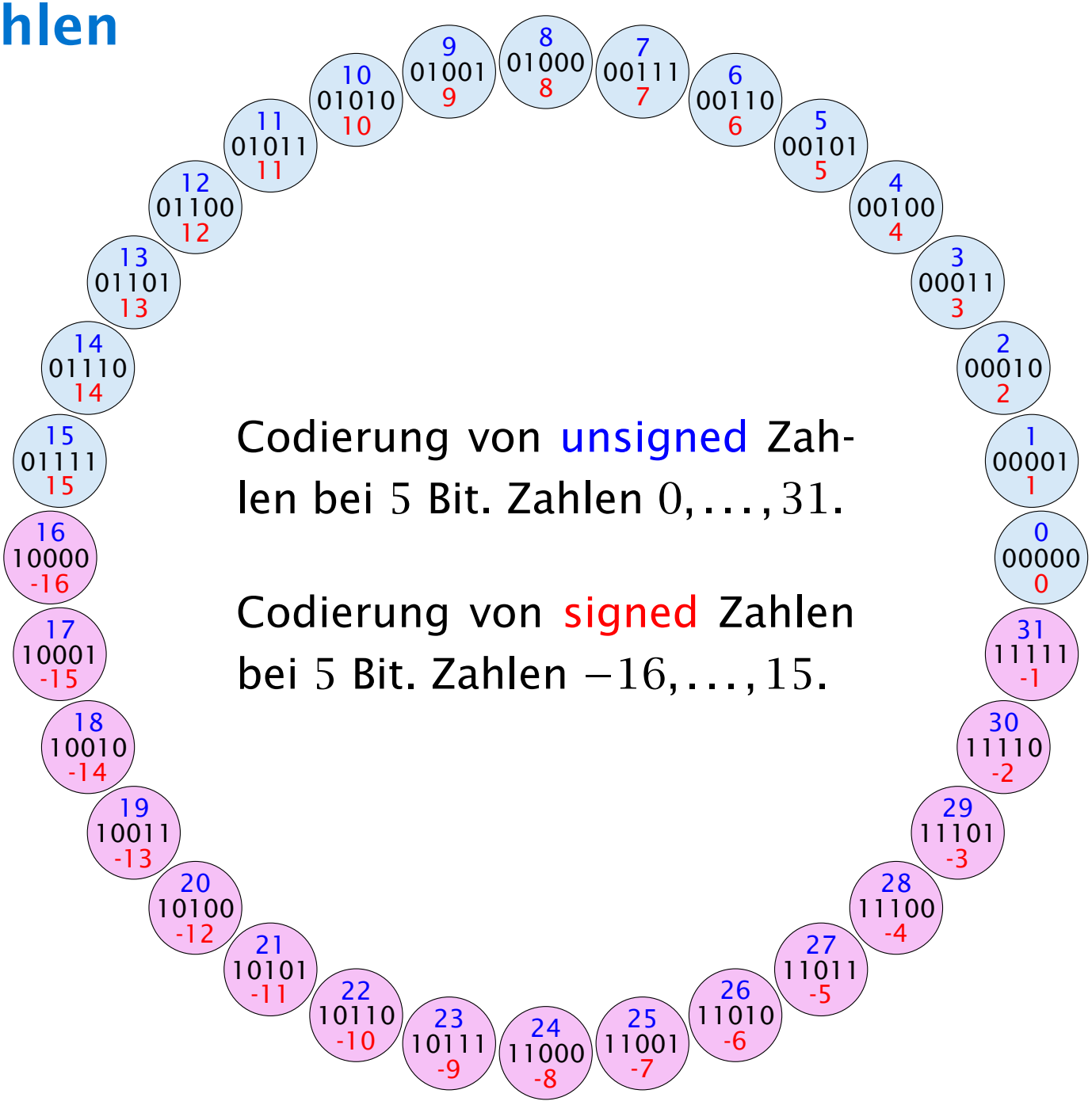
- **8 Bits** (ein Byte), darstellbare Zahlen: $\{0,\ldots,255\}$
  in C: `unsigned char`
- **16 Bits**, darstellbare Zahlen: $\{0,\ldots,65\,535\}$
  in C: `unsigned short`
- **32 Bits**, darstellbare Zahlen: $\{0,\ldots,4\,294\,967\,295\}$
  in C: `unsigned long`
- **64 Bits**, darstellbare Zahlen: $\{0,\ldots,2^{64}-1\}$
  in C: `unsigned long long`

# Negative Zahlen




Codierung von unsigned Zahlen bei $5$ Bit. Zahlen $0, \ldots, 31$.

# Negative Zahlen




Codierung von unsigned Zahlen bei $5$ Bit. Zahlen $0, \ldots, 31$.

Codierung von signed Zahlen bei $5$ Bit. Zahlen $-16, \ldots, 15$.

# Negative Zahlen

| Bitfolge | | Zahl |
|---|---|---|
| $x = \langle x_{n-1}, \ldots, x_0 \rangle$ | $\underset{f^{-1}}{\overset{f}{\rightleftarrows}}$ | $\sum_{i=0}^{n-1} x_i 2^i$ |
| $x = \langle x_{n-1}, \ldots, x_0 \rangle$ | $\underset{f_{ZK}^{-1}}{\overset{f_{ZK}}{\rightleftarrows}}$ | $-x_{n-1} 2^n + \sum_{i=0}^{n-1} x_i 2^i$ |

# Negative Zahlen

## Definition

In 2-Komplement Darstellung mit $n$ bits repräsentiert die Bitfolge

$$x = \langle x_{n-1}, x_{n-2}, \ldots, x_1, x_0 \rangle$$

die Zahl $f_{\mathrm{ZK}}(x) = -x_{n-1}2^n + \sum_{i=0}^{n-1} x_i 2^i$.

## Beobachtungen

- Zahlen mit $x_{n-1} = 1$ sind negativ; andere positiv (Vorzeichenbit)
- positive Zahlen: $0, \ldots, 2^{n-1} - 1$
  negative Zahlen: $-1, \ldots, -2^{n-1}$
- $f_{\mathrm{ZK}}(x) = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$.

# Negative Zahlen

Beachte, dass die Regel für die Zahl $-2^{n-1}$ nicht gelten kann, da $+2^{n-1}$ im 2er-Komplement mit $n$ bits nicht darstellbar ist. Für die Null gilt die Regel nur wenn man die Addition $1 + f(\bar{x})$ modulo $2^n$ ausführt.

**Vorzeichenwechsel**

Sei $x = \langle x_{n-1}, \ldots, x_0 \rangle$ ein Bitfolge mit $\langle x_{n-2}, \ldots, x_0 \rangle \neq \langle 0, \ldots, 0 \rangle$.
Die Repräsentation für die Zahl $-f_{\mathrm{ZK}}(x)$ im 2er Komplement (d.h. $f_{\mathrm{ZK}}^{-1}(-f_{\mathrm{ZK}}(x))$) erhält man durch

$$f^{-1}(f(\bar{x}) + 1)$$

wobei $\bar{x} = \langle \bar{x}_{n-1}, \ldots, \bar{x}_0 \rangle$ die invertierte Bitfolge bezeichnet.

D.h. man invertiert die Bitfolge und addiert $1$ auf die sich ergebende Zahl.

# Negative Zahlen

**Beweis**

1. Fall: $x_{n-1} = 1$, d.h. $f_{\mathrm{ZK}}(x)$ negativ

$$-f_{\mathrm{ZK}}(x) = 2^n - \sum_{i=0}^{n-1} x_i 2^i = 1 + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} x_i 2^i$$

$$= 1 + \sum_{i=0}^{n-1} (1 - x_i) 2^i = 1 + \sum_{i=0}^{n-1} \bar{x}_i 2^i$$

$$= 1 + f(\bar{x})$$

Da $1 + f(\bar{x}) < 2^{n-1}$ liefert Anwendung von $f^{-1}$ oder $f_{\mathrm{ZK}}^{-1}$ die gleiche Bitfolge.

# Negative Zahlen

**Beweis**

2. Fall: $x_{n-1} = 0$, d.h. $f_{\mathrm{ZK}}(x)$ strikt positiv

$$-f_{\mathrm{ZK}}(x) = -\sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{n-1} (1-x_i)2^i - \sum_{i=0}^{n-1} 2^i - 1 + 1$$

$$= 1 + \sum_{i=0}^{n-1} \bar{x}_i 2^i - 2^n = 1 + f(\bar{x}) - 2^n$$

Für eine Zahl $z$ die das höchstwertige Bit $n-1$ gesetzt hat gilt $f_{\mathrm{ZK}}^{-1}(z - 2^n) = f^{-1}(z)$. Dies gilt für $1 + f(\bar{x})$.

# Negative Zahlen

**Definition**

Die Restklasse $[a]_m$ enthält alle $z \in \mathbb{Z}$ die bei Division durch $m$ den gleichen Rest lassen.

$a$ heißt Repräsentant der Restklasse. Eine Restklasse hat viele verschiedene Repräsentanten.

Für eine Restklasse $M \subseteq \mathbb{Z}$ nennen wir $a \in M$ mit $0 \leq a < m$ den Standardrepräsentanten der Restklasse.

**Beispiel**

- $[2]_8 = [42]_8 = [-78]_8$

# Negative Zahlen

## Rechnen mit Restklassen

Man kann mit Restklassen rechnen. Die Multiplikation / Addition / Subtraktion etc. wird repräsentantenweise ausgeführt. Die Wahl des Repräsentanten ist unwichtig!!!!!!!!!

**Beispiele:**

- $[2]_8 \cdot [7]_8 = [2 \cdot 7]_8 = [6]_8$
- $[-6]_8 \cdot [23]_8 = [-6 \cdot 23]_8 = [-138]_8 = [-18]_8 = [6]_8$
- $[7]_8 + [8]_8 = [15]_8 = [-1]_8$

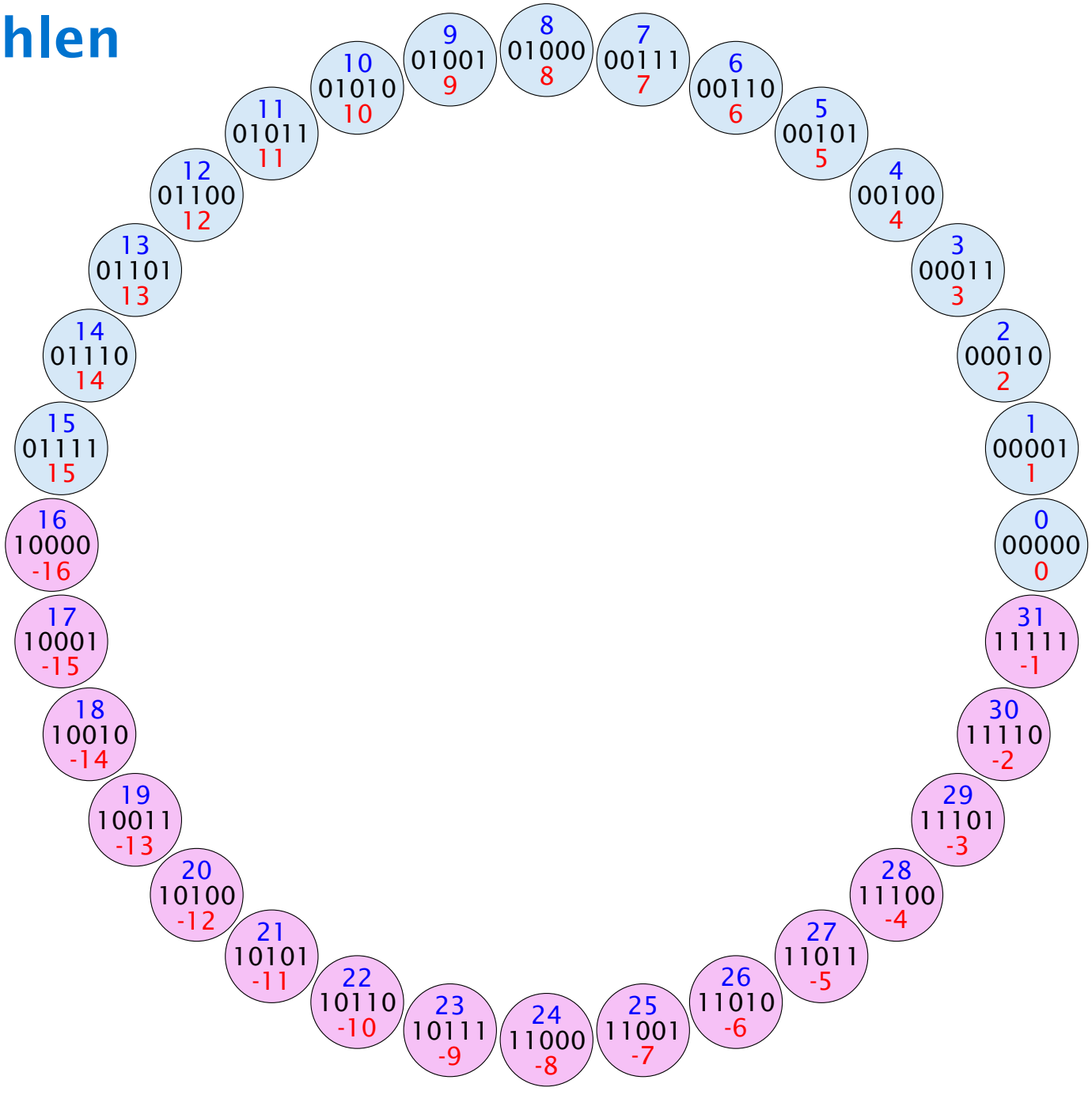
# Negative Zahlen

Die Hardware Implementierung von Addition / Multiplikation etc. implementiert eigentlich eine Operation auf Restklassen modulo $2^n$, wobei $n$ der Bitlänge entspricht.

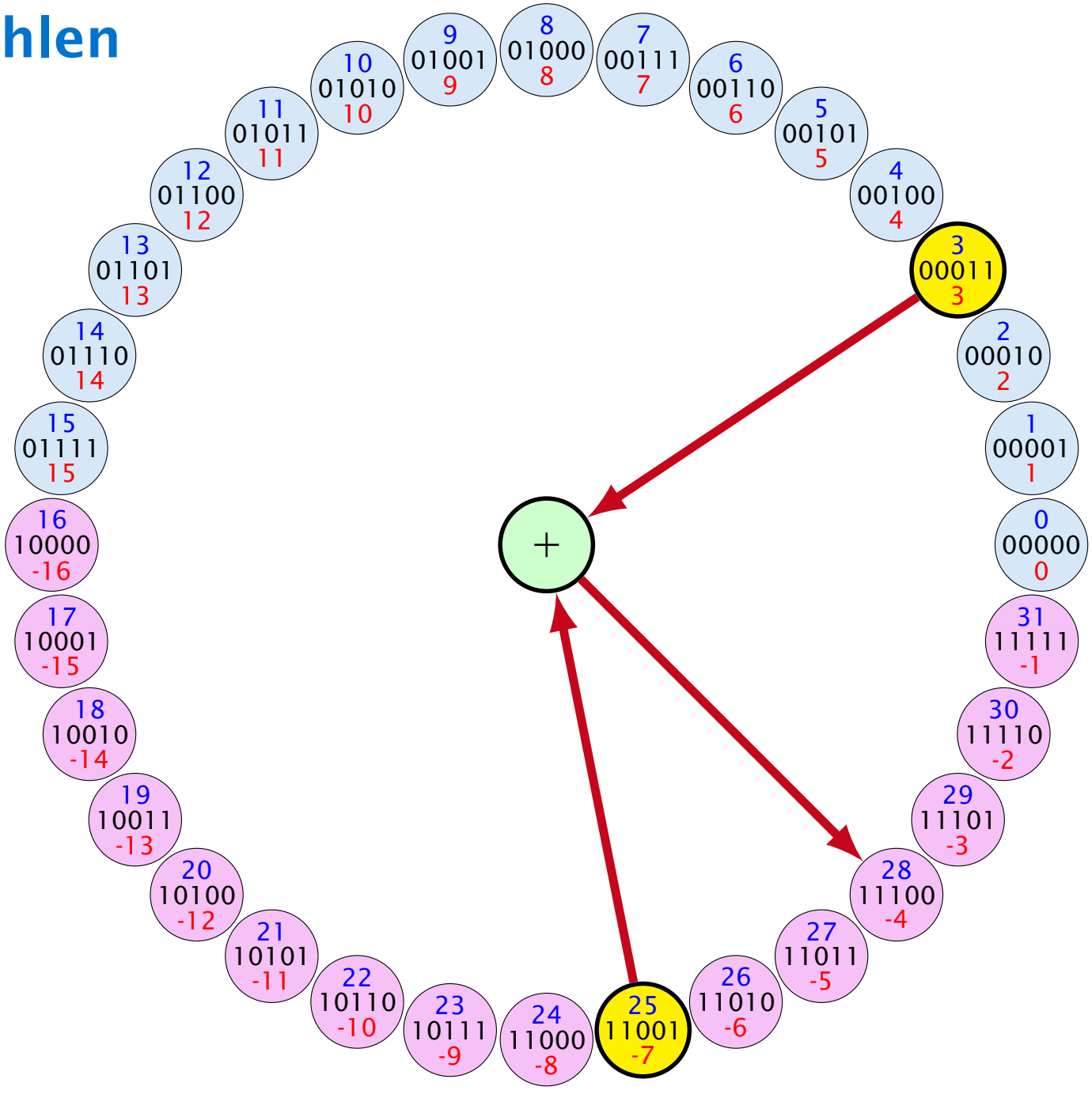
Im Prinzip wird eine Operation (Addition / Subtraktion / Multiplikation / Ganzahldivision) ausgeführt, und dann werden überzählige Bits verworfen (d.h., dass Ergebnis wird modulo $2^n$ genommen).

Durch das Verwenden des 2er-Komplements kann man für signed und unsigned Datentypen, (im wesentlichen) die gleiche Hardware benutzen.

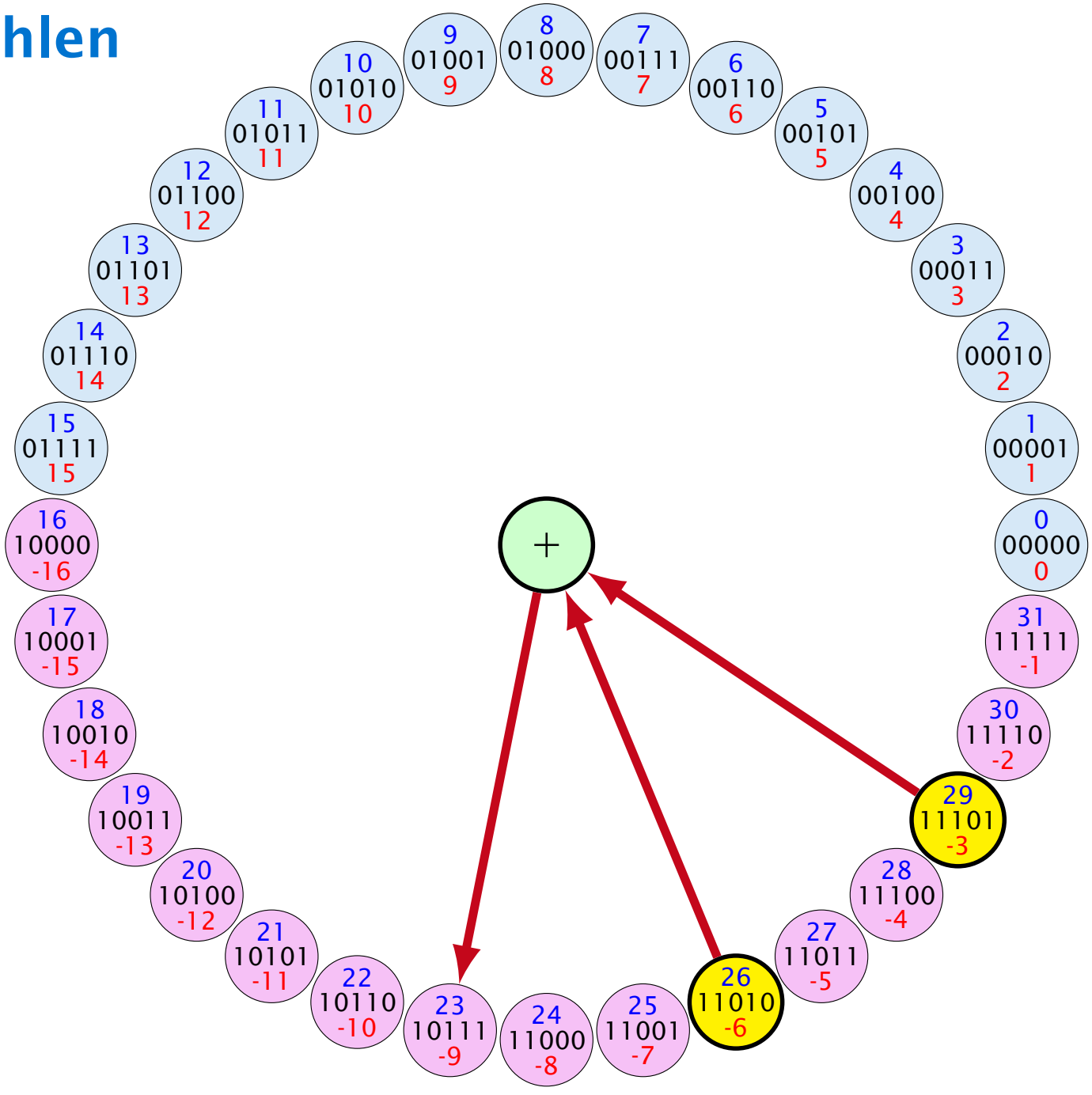
# Negative Zahlen



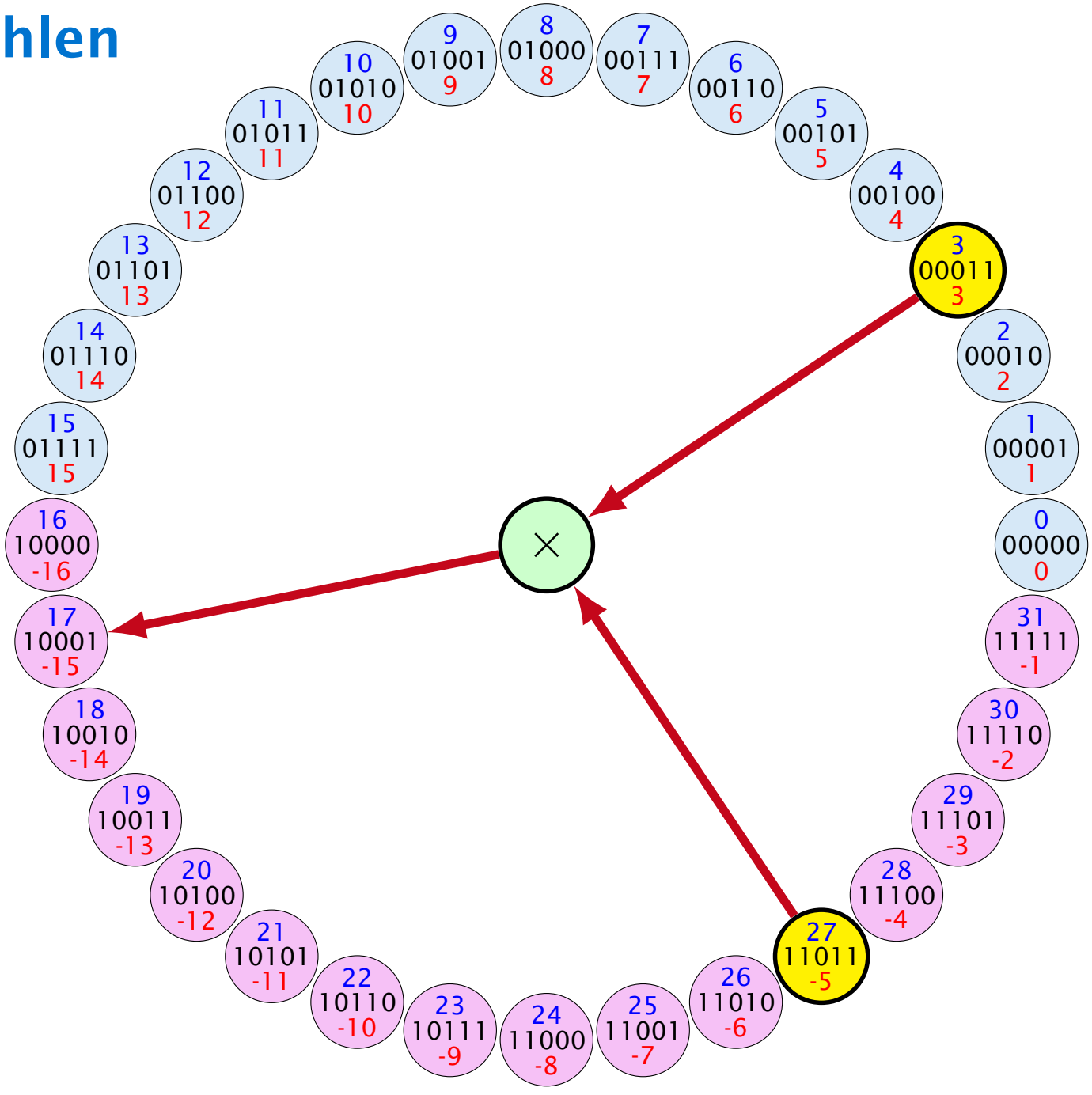
# Negative Zahlen



# Negative Zahlen



# Negative Zahlen



# Ganze Zahlen in C-ähnlichen Sprachen

**Ganze Zahlen:**

Die primitiven Datentypen für ganze Zahlen sind:

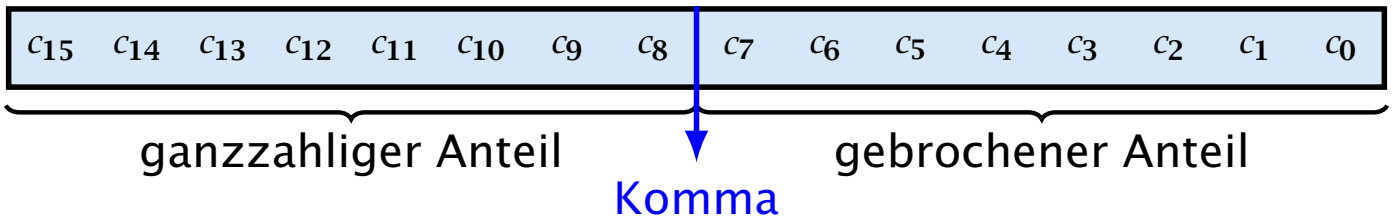
- **8 Bits:** `unsigned char` $\{0,\ldots,255\}$
  `signed char` $\{-128,\ldots,127\}$
- **16 Bits:** `unsigned short` $\{0,\ldots,65535\}$
  `signed short` $\{-32768,\ldots,32767\}$
- **32 Bits:** `unsigned long` $\{0,\ldots,2^{32}-1\}$
  `signed long` $\{-2^{31},\ldots,2^{31}-1\}$
- **64 Bits:** `unsigned long long` $\{0,\ldots,2^{64}-1\}$
  `signed long long` $\{-2^{63},\ldots,2^{63}-1\}$
- `signed` kann weggelassen werden (ausser bei `char`!)
- `unsigned int` und `signed int` sind je nach System 16, 32 oder 64 Bit

Dies sind nur Mindestvorgaben. Auf der Cray ist ein `short` z.B. 64 Bit lang.

# Rationale Zahlen I

**Festkommadarstellung:** Komma an fester Stelle in Zahl

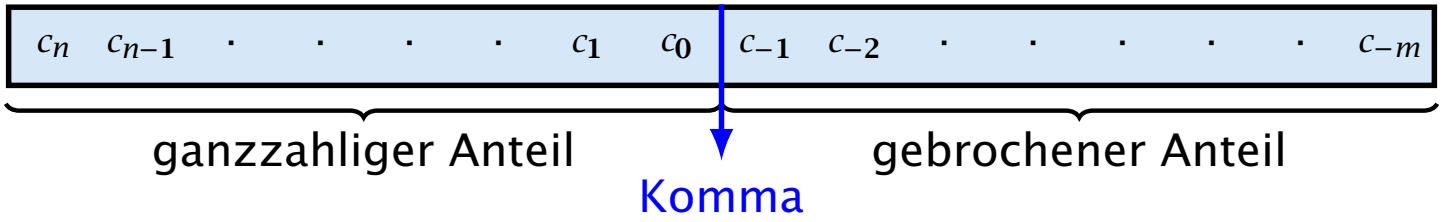
Beispiel mit $n = 16$:




Nachteile:

- weniger große Zahlen darstellbar
- feste Genauigkeit der Nachkommastellen

# Rationale Zahlen II




**Interpretation** für $r \in \mathbb{Q}$:

$$r = c_n \cdot 2^n + \ldots + c_0 \cdot 2^0 + c_{-1}2^{-1} + \ldots + c_{-m} \cdot 2^{-m}$$

mit $n+1$ Vorkomma- und $m$ Nachkommaziffern

**Beispiel:**

$$11.01_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$$
$$= 2 + 1 + 0 + \tfrac{1}{4} = 3.25_{10}$$

# Floating Point Zahlen I

**Wissenschaftliche Notation:** $x = a \cdot 10^b$ für $x \in \mathbb{R}$, wobei:

- $a \in \mathbb{R}$ mit $1 \leq |a| < 10$
- $b \in \mathbb{Z}$

**Beispiele:**

- $-2.7315 \cdot 10^2$ °C absoluter Nullpunkt
- $1.5 \cdot 10^9$ Hz Taktfrequenz A8X Prozessor

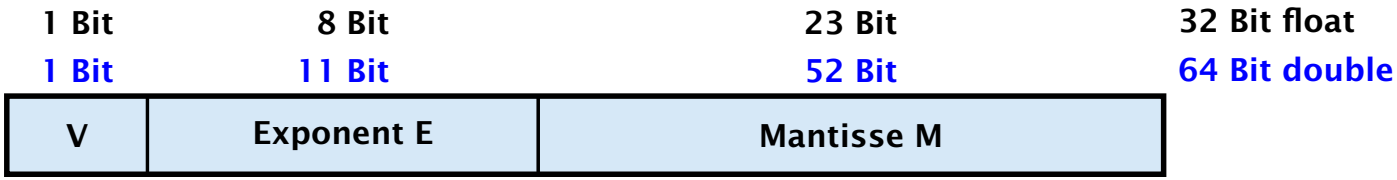
**Drei Bestandteile:**

- Vorzeichen
- Mantisse $|a|$ (bestimmt die Genauigkeit)
- Exponent $b$ (bestimmt Größe des Wertebereichs)

Problem: bei fester Länge der Mantisse (z.B. 3 Ziffern)

- zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!

# Floating Point Zahlen II




- wissenschaftliche Darstellung mit Basis 2

$$f = (-1)^V \cdot (1 + M) \cdot 2^{E-bias}$$

- Vorzeichen Bit $V$
- Mantisse $M$ hat immer die Form `1.abc`, also wird erste Stelle weggelassen („hidden bit“)
- Exponent $E$ wird vorzeichenlos abgespeichert, verschoben um $bias$
  - bei 32 bit: $bias = 127$, bei 64 bit: $bias = 1023$

# Floating Point Zahlen III

## Übliche Floating Point Formate:

| *Bit* | *Vorz.* | *Exp.* | *Mant.* | *Dezimal stellen* | *Bereich* |
|---|---|---|---|---|---|
| 32 | 1 Bit | 8 Bit | 23 Bit | $\sim 7$ | $\pm 2 \cdot 10^{-38}$ bis $\pm 2 \cdot 10^{38}$ |
| 64 | 1 Bit | 11 Bit | 52 Bit | $\sim 15$ | $\pm 2 \cdot 10^{-308}$ bis $\pm 2 \cdot 10^{308}$ |
| 80 | 1 Bit | 15 Bit | 64 Bit | $\sim 19$ | $\pm 1 \cdot 10^{-4932}$ bis $\pm 1 \cdot 10^{4932}$ |

In C:

`float` (32 Bit), `double` (64 Bit), `long double` (80 Bit)

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: `(0.1 + 0.2 == 0.3)` ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: (0.1 + 0.2 == 0.3) ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: `(0.1 + 0.2 == 0.3)` ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: `(0.1 + 0.2 == 0.3)` ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

# Definition Datenstruktur

**Definition Datenstruktur (nach Prof. Eckert)**
Eine Datenstruktur ist eine

- **logische Anordnung** von Datenobjekten,
- die **Informationen** repräsentieren,
- den **Zugriff** auf die repräsentierte Information über **Operationen** auf Daten ermöglichen und
- die Information **verwalten**.

Zwei Hauptbestandteile:

- Datenobjekte
  z.B. definiert über primitive Datentypen
- Operationen auf den Objekten
  z.B. definiert als Funktionen

# Primitive Datentypen in C

Natürliche Zahlen, z.B. `unsigned short`, `unsigned long`

- Wertebereich: bei $n$ Bit von 0 bis $2^n - 1$
- Operationen: `+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`

Ganze Zahlen, z.B. `int`, `long`

- Wertebereich: bei $n$ Bit von $-2^{n-1}$ bis $2^{n-1} - 1$
- Operationen: `+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`

Floating Point Zahlen, z.B. `double`, `float`

- Wertebereich: abhängig von Größe
- Operationen: `+`, `-`, `*`, `/`, `<`, `==`, `!=`, `>`

Logische Werte, `bool`

- Wertebereich: `true`, `false`
- Operationen: `&&`, `||`, `!`, `==`, `!=`

# Definition Feld

**Definition Feld**

Ein Feld $F$ ist eine Folge von $n$ Datenelementen $(d_i)_{i=1,\ldots,n}$,

$$F = d_1, d_2, \ldots, d_n$$

mit $n \in \mathbb{N}_0$.

Die Datenelemente $d_i$ sind beliebige Datentypen (z.B. primitive).

**Beispiele:**

- $F$ sind die natürlichen Zahlen von 1 bis 10, aufsteigend geordnet:

$$F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

- Ist $n = 0$, so ist das Feld leer.

# Definition Feld

**Operationen**

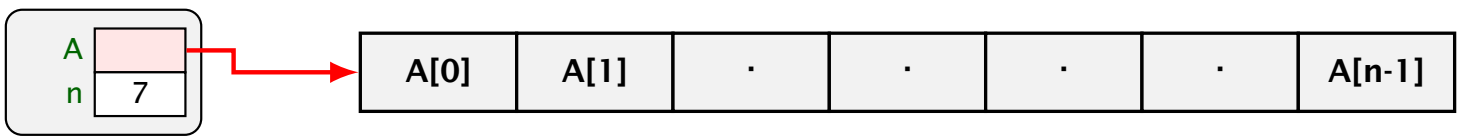
- `F.initialize()`: initialisiere leeres Feld `A`
- `F.elementAt(i)`: Zugriff auf `i`-tes Element von `A`.
  `i` muss zwischen `1` und `F.size()` liegen.
- `F.insert(d,i)`: füge Element `d` an Position `i` in Feld `A` ein.
  `i` muss zwischen `1` und `F.size()+1` liegen
- `F.erase(i)`: entferne `i`-tes Element aus Feld `A`.
- `F.size()`: gibt die Anzahl der Elemente des Feldes zurück

Ein abstrakter Datentyp wird im wesentlichen durch die auf ihn anwendbaren Operationen definiert.

# Feld als Array

**Repräsentation von Feld durch Array der Länge $n$**

- Datenelemente werden in Array gespeichert
- einfacher Zugriff über index-operator (`A[i]`)
- Hinzufügen/Löschen schwierig...




Achtung: Indizierung des Arrays startet bei 0!

`A[i]` enthält Element `i+1` des Feldes

# Eigenschaften von Arrays

Feld F mit Länge n als Array

**Vorteile:**

- direkter Zugriff auf Elemente in konstanter Zeit mittels A[i]
- sequentielles Durchlaufen sehr einfach

**Nachteile:**

- Verlängern des Feldes aufwendig
- Hinzufügen und Löschen von Elementen aufwendig

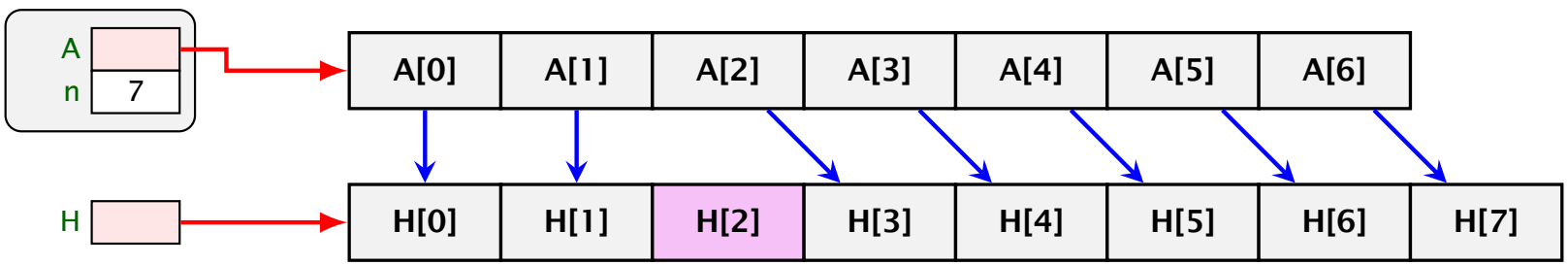
# Hinzufügen eines Elementes

**Gegeben:** Feld F, Länge n, via Array implementiert

**Gewünscht:** Feld F, zusätzliches Element $d_i$ an Position $i$; Elemente an Positionen $\geq i$ werden auf nächsthöhere Position verschoben

- neuen Speicher der Größe n+1 reservieren
- altes Array in neuen Speicher kopieren

**Beispiel:** `F.insert(12,3)` (einfügen an Position 3)



# Implementierung: Feld via Array

```
 3 class Feld {
 4     int  n;
 5     int* A;
 6
 7   public:
 8     Feld() {
 9         n = 0;
10         A = new int[n];
11     }
```

# Implementierung: Feld via Array

```
13      void insert(int d, int i) {
14          int* H = new int[n+1];
15          for (int j=0; j<i-1; j++) {
16              H[j] = A[j];
17          }
18          H[i-1] = d;
19          for (int j=i-1; j<n; j++) {
20              H[j+1] = A[j];
21          }
22          delete[] A;
23          A = H;
24          n++;
25      }
```

# Implementierung: Feld via Array

```
27      void erase(int i) {
28          int* H = new int[n-1];
29          for (int j=0; j<i-1; j++) {
30              H[j] = A[j];
31          }
32          for (int j=i; j<n; j++) {
33              H[j-1] = A[j];
34          }
35          delete[] A;
36          A = H;
37          n--;
38      }
```

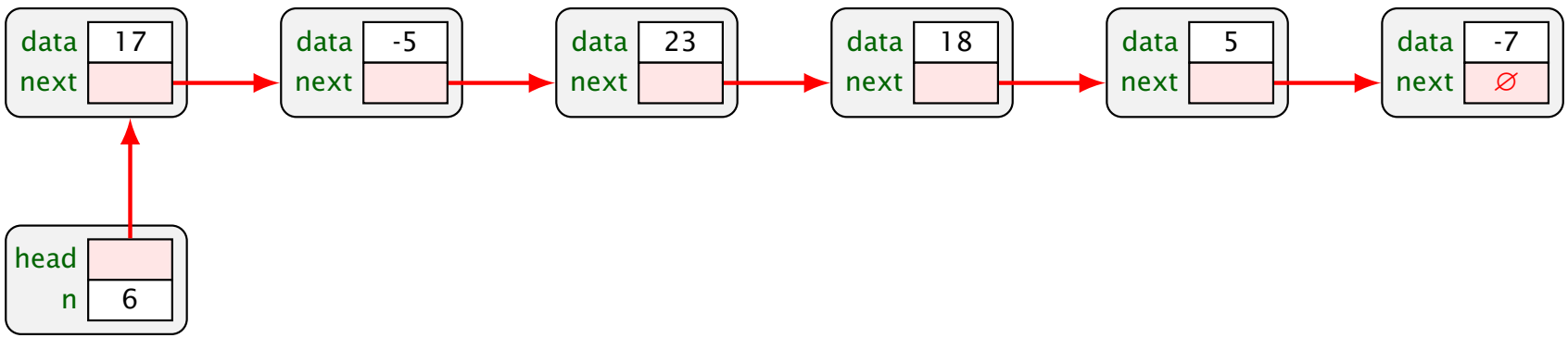
# Implementierung: Feld via Array

```
40      int elementAt(int i) {
41          return A[i-1];
42      }
43
44      int size() {
45          return n;
46      }
47  };
```

# Feld als einfach verkettete Liste

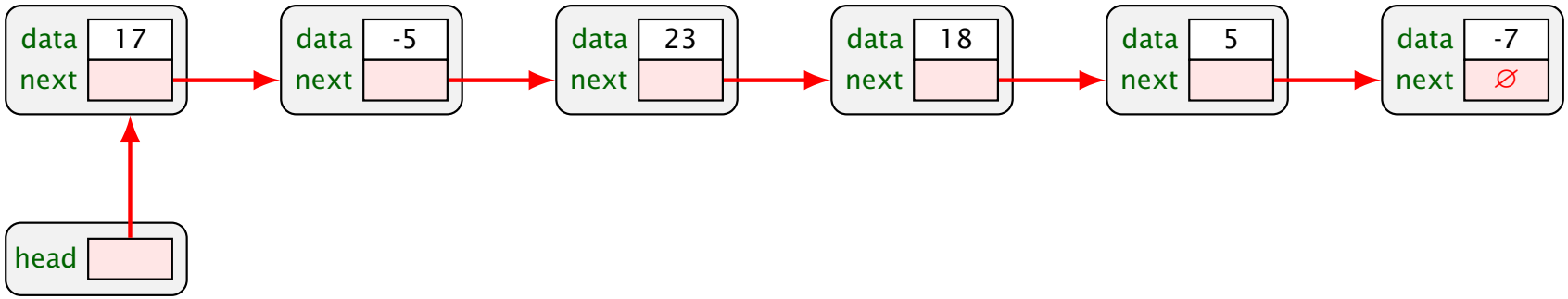
**Repräsentation von Feld als verkettete Liste**

- dynamische Anzahl von Datenelementen
- in linearer Reihenfolge gespeichert (nicht notwendigerweise zusammenhängend!)
- mit Referenzen oder Zeigern verkettet




**auf Englisch:** *linked list*

# Verkettete Liste




Folge von miteinander verbundenen Elementen

jedes Element besteht aus

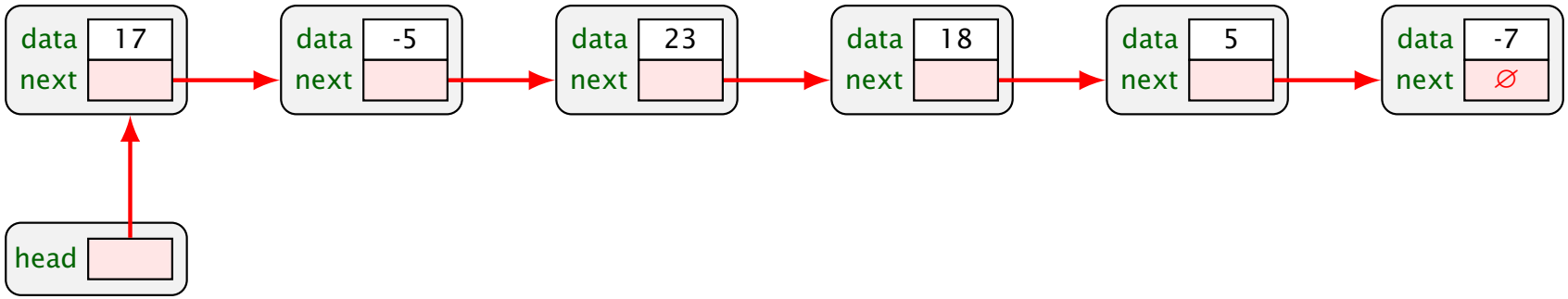
- **data**: Wert des Feldes an Position i
- **next**: Referenz auf das nächste Element

head ist Referenz auf erstes Element der Liste

letztes Element hat keinen Nachfolger

- symbolisiert durch null-Referenz

# Verkettete Liste




Folge von miteinander verbundenen Elementen

jedes Element besteht aus

- **data**: Wert des Feldes an Position i
- **next**: Referenz auf das nächste Element

head ist Referenz auf erstes Element der Liste

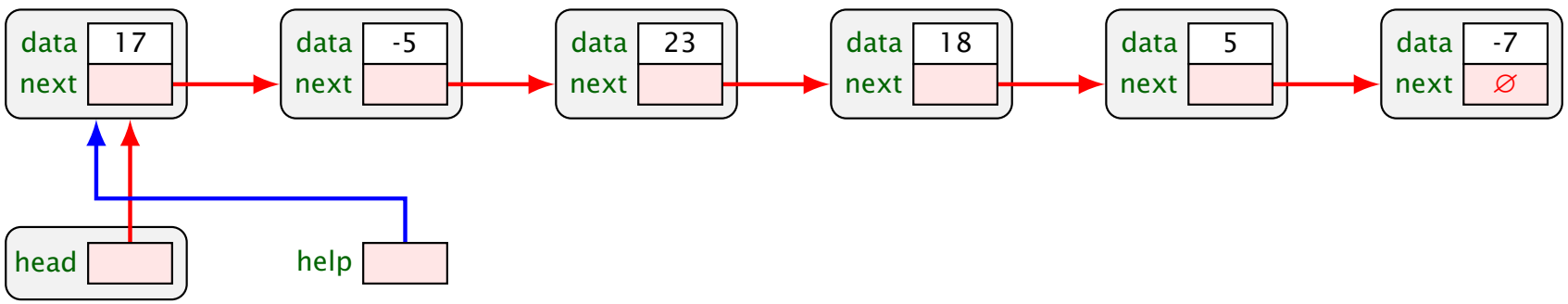
letztes Element hat keinen Nachfolger

- symbolisiert durch null-Referenz

# Verketteter Liste – Zugriff auf Element

**Zugriff auf Element i:**

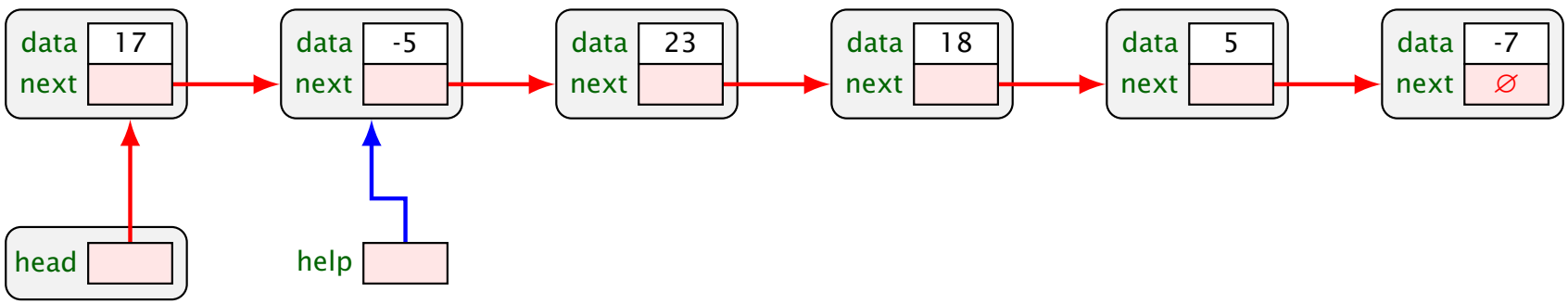
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i`-ten Element



# Verketteter Liste – Zugriff auf Element

**Zugriff auf Element i:**

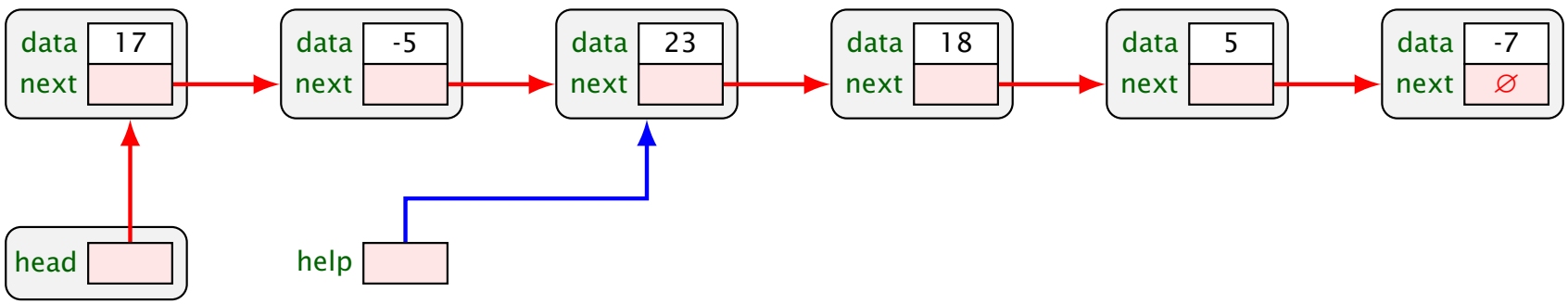
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i`-ten Element



# Verketteter Liste – Zugriff auf Element

**Zugriff auf Element i:**

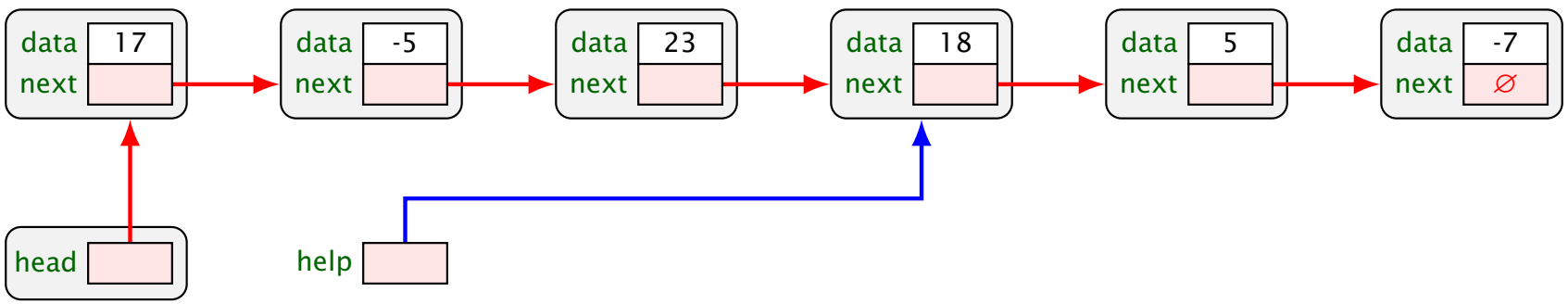
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i`-ten Element



# Verketteter Liste – Zugriff auf Element

**Zugriff auf Element i:**

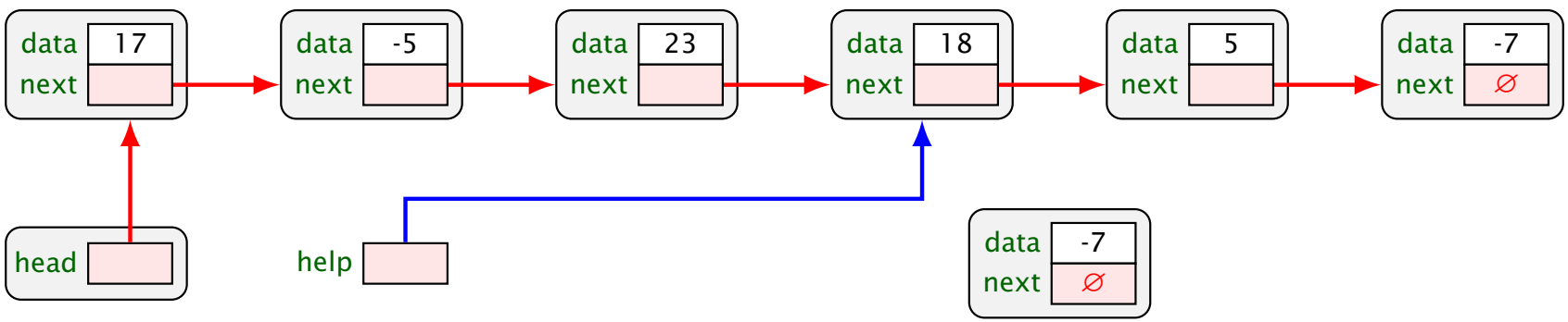
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i`-ten Element



# Verketteter Liste – Einfügen nach Referenz

**Einfügen nach Element i:**

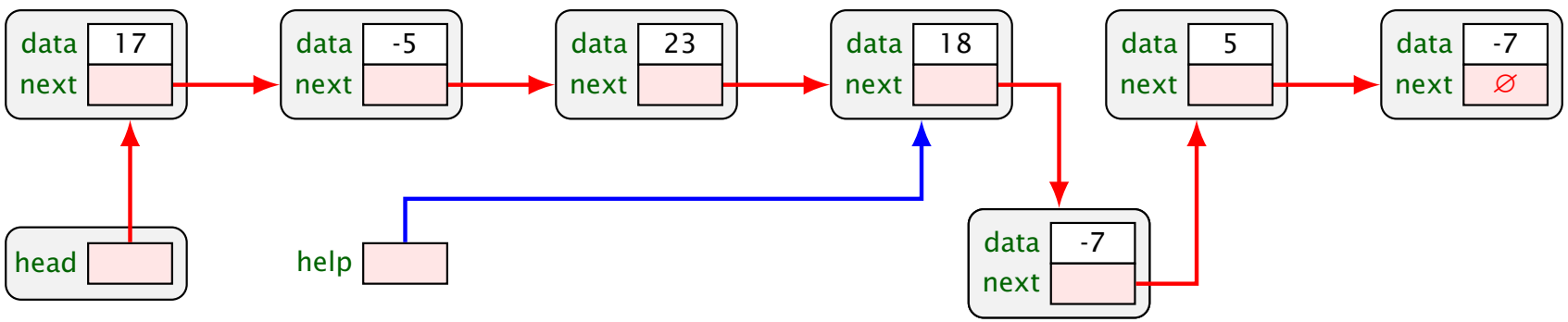
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i-1`-ten Element
- Referenzen umsetzen



# Verketteter Liste – Einfügen nach Referenz

**Einfügen nach Element i:**

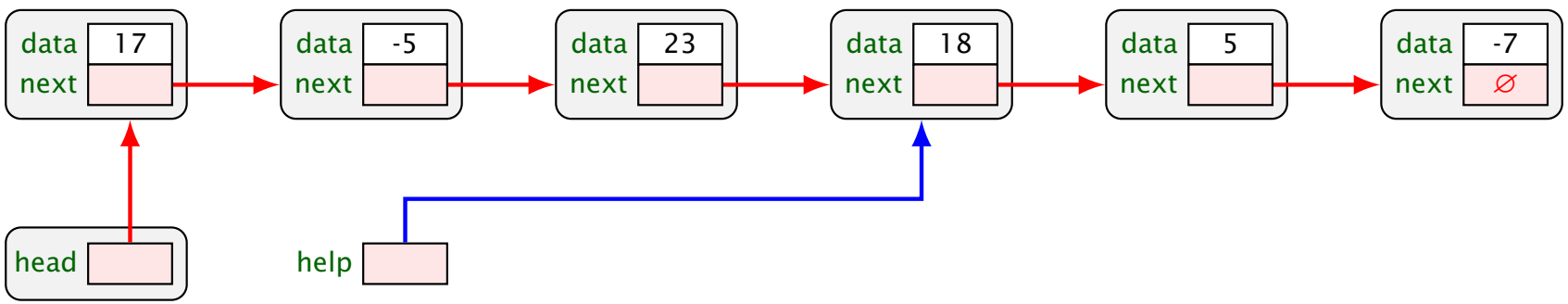
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i-1`-ten Element
- Referenzen umsetzen



# Verketteter Liste – Löschen nach Referenz

**Einfügen nach Element i:**

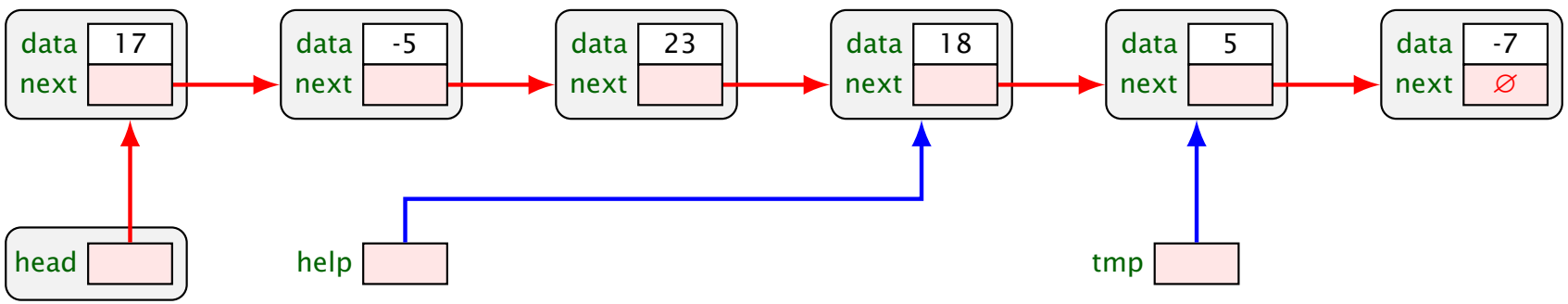
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i-1`-ten Element
- Referenzen umsetzen + Speicher freigeben



# Verketteter Liste – Löschen nach Referenz

**Einfügen nach Element i:**

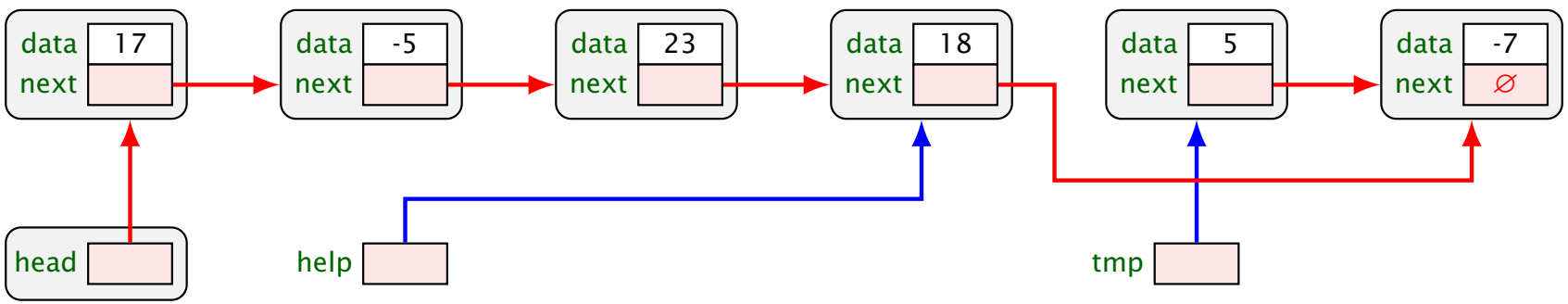
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i-1`-ten Element
- Referenzen umsetzen + Speicher freigeben



# Verketteter Liste – Löschen nach Referenz

**Einfügen nach Element i:**

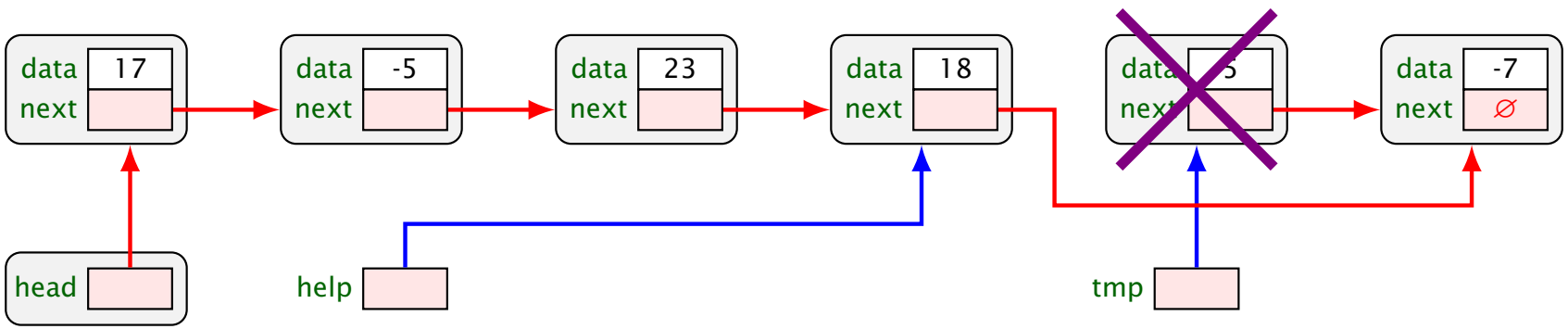
- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i-1`-ten Element
- Referenzen umsetzen + Speicher freigeben



# Verketteter Liste – Löschen nach Referenz

**Einfügen nach Element i:**

- beginne bei `head`-Referenz
- "vorhangeln" entlang `next`-Referenzen bis zum `i-1`-ten Element
- Referenzen umsetzen + Speicher freigeben



# Implementierung: Feld via Liste

```
 4 struct Node {
 5     int data;
 6     Node *next;
 7
 8     Node(int d, Node* n) {
 9         data = d;
10         next = n;
11     }
12 };
```

# Implementierung: Feld via Liste

```
14 class Feld {
15     int n;
16     Node *head;
17   public:
18
19     Feld() { // erzeuge leeres Feld
20         n    = 0;
21         head = NULL;
22     }
23
24     int size() { return n; }
25
26     int elementAt(int i) {
27         Node* h = head;
28         while (i-- > 1)
29             h = h->next;
30         return h->data;
31     }
```

# Implementierung: Feld via Liste

```
33      void insert(int d, int i) {
34          Node* tmp = new Node(d, NULL);
35          n++;
36          if (i == 1) {
37              tmp->next = head;
38              head      = tmp;
39              return;
40          }
41          Node* h = head;
42          i--;
43          while (i-- > 1)
44              h = h->next;
45          tmp->next = h->next;
46          h->next   = tmp;
47      }
```

# Implementierung: Feld via Liste

```
49      void erase(int i) {
50          n--;
51          if (i == 1) {
52              Node* tmp = head;
53              head = head->next;
54              delete tmp;
55              return;
56          }
57          Node* h = head;
58          i--;
59          while (i-- > 1)
60              h = h->next;
61          Node* tmp = h->next;
62          h->next = h->next->next;
63          delete tmp;
64      }
65  }; // end class
```

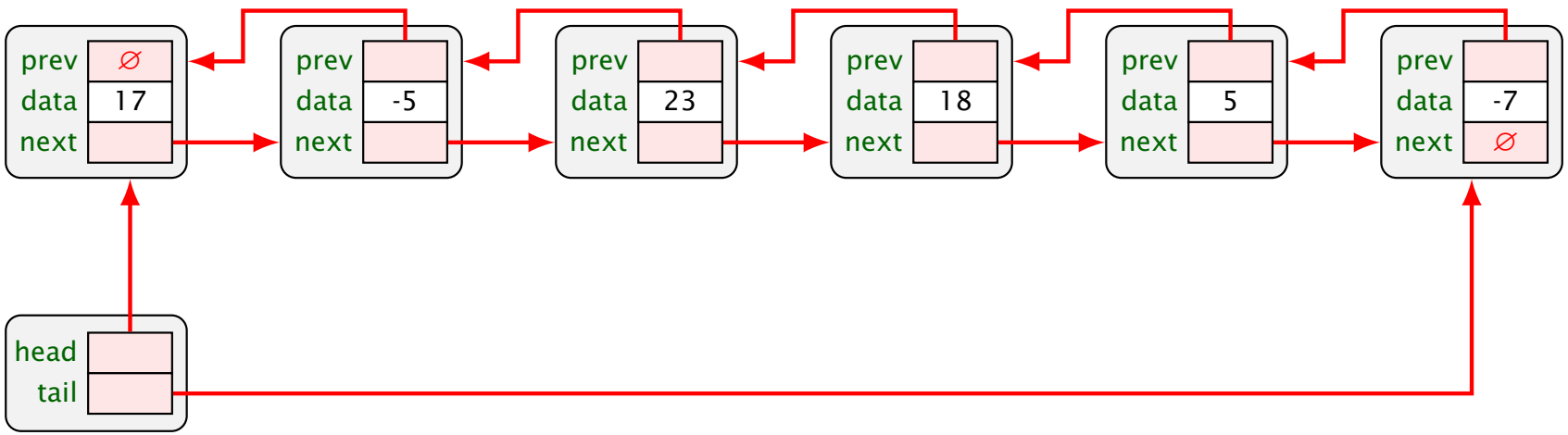
# Gegenüberstellung Array und verkettete Liste

| Array | Verkettete Liste |
|---|---|
| ⊕ Direkter Zugriff auf i-tes Element | ⊖ Zugriff auf i-tes Element erfordert i Iterationen |
| ⊕ sequentielles Durchlaufen sehr einfach | ⊕ sequentielles Durchlaufen sehr einfach |
| ⊖ statische Länge, kann Speicher verschwenden | ⊕ dynamische Länge |
| | ⊖ zusätzlicher Speicher für Zeiger benötigt |
| ⊖ Einfügen/Löschen erfordert erheblich Kopieraufwand | ⊕ Einfügen/Löschen einfach |

# Feld als doppelt verkettete Liste

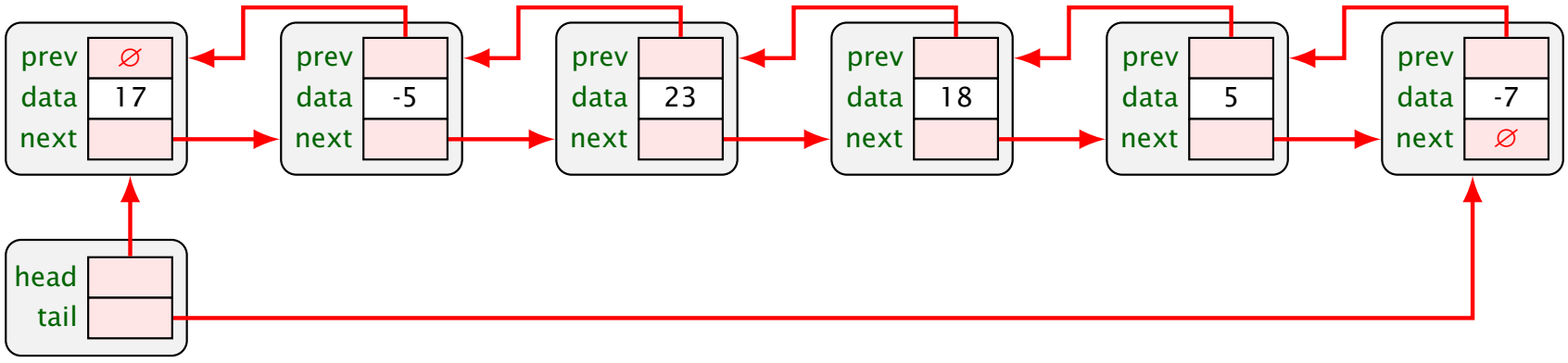
**Repräsentation von Feld A als doppelt verkettete Liste**

- verkettete Liste
- jedes Element mit Referenzen doppelt verkettet




**auf Englisch:** *doubly linked list*

# Doppelt verkettete Liste




Folge von miteinander verbundenen Elementen

Jedes Element besteht aus

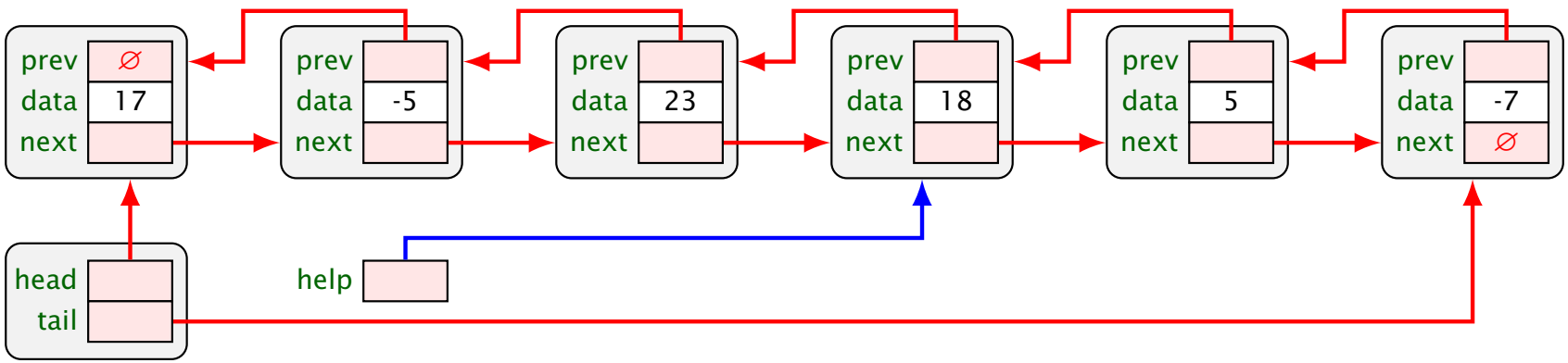
- `data`: Wert des Feldes
- `next`: Referenz auf das nächste Element
- `prev`: Referenz auf das vorherige Element

`head`/`tail` sind Referenzen auf erstes/letztes Element; diese haben keinen Nachfolger bzw. keinen Vorgänger.

# Operationen auf doppelt verketteter Liste

**Löschen von Element i:**

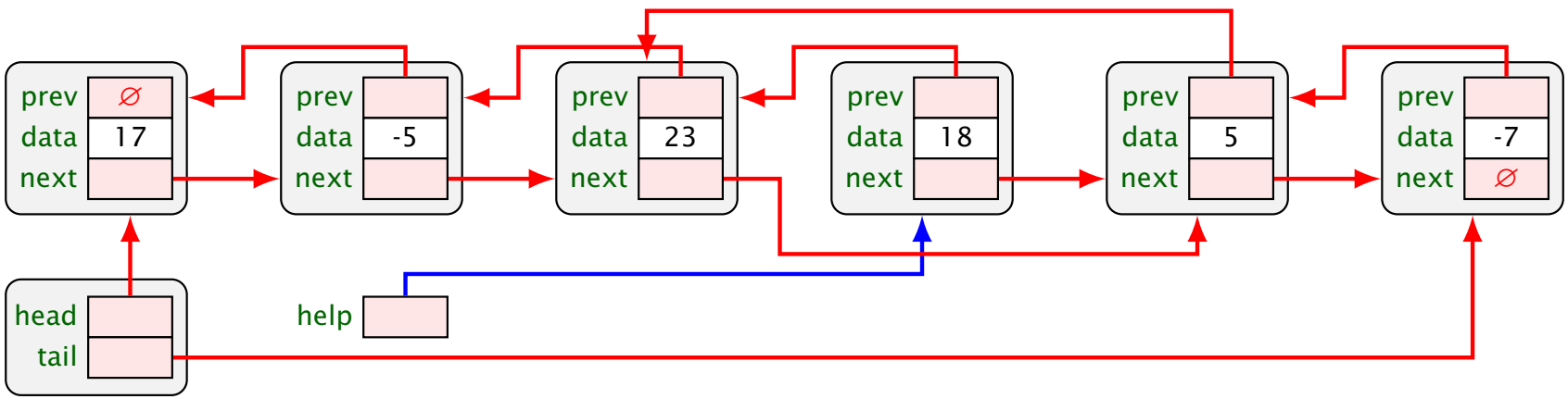
- Zugriff auf Element i
- umhängen von Referenzen
- Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

**Löschen von Element i:**

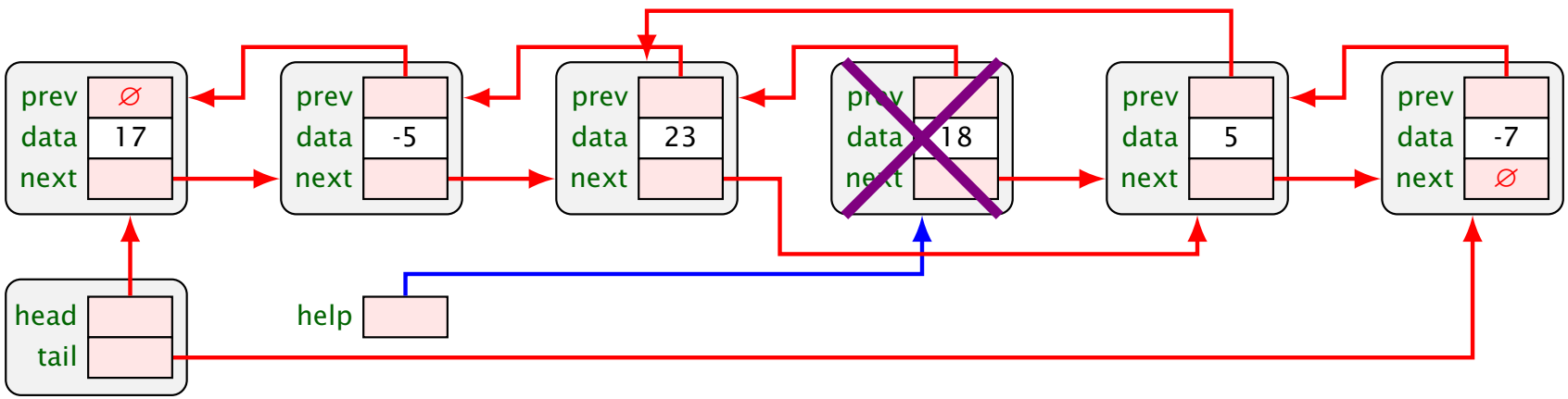
- Zugriff auf Element i
- umhängen von Referenzen
- Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

**Löschen von Element i:**

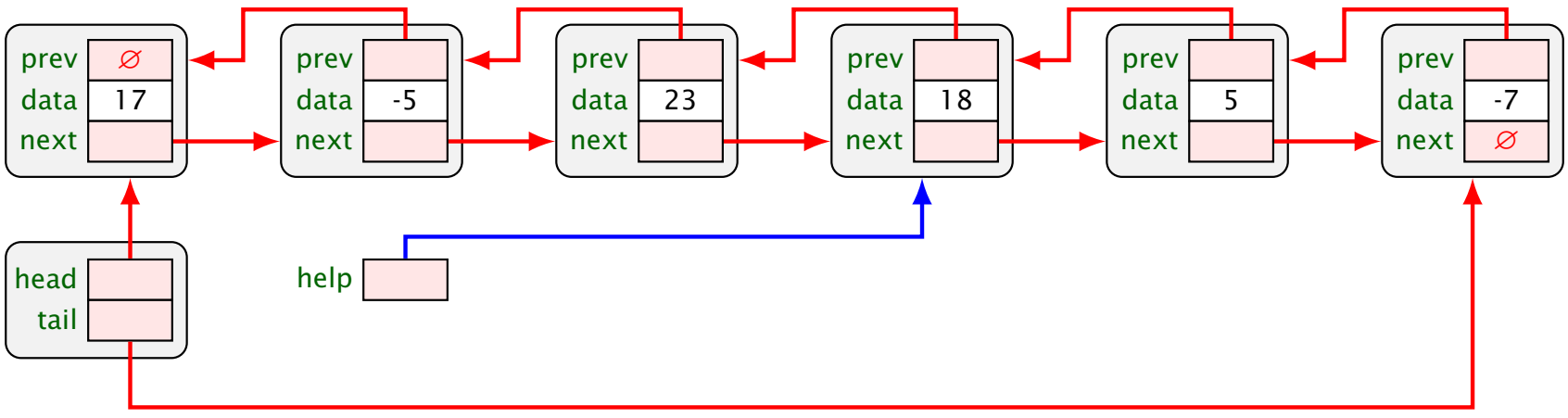
- Zugriff auf Element i
- umhängen von Referenzen
- Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

**Einfügen von Element an Stelle i:**

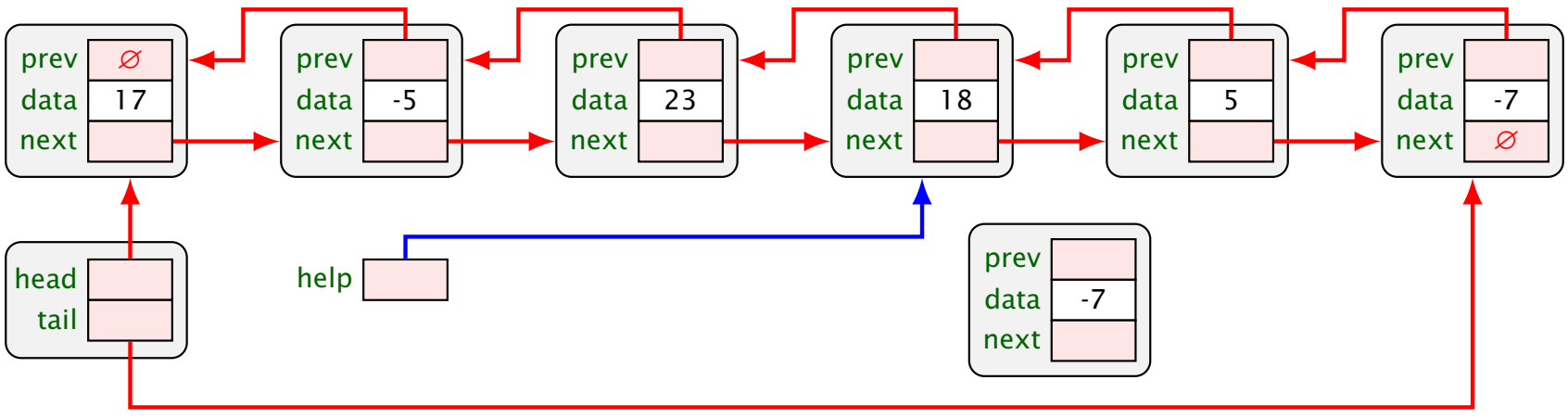
- Zugriff auf Element `i-1`
- umhängen von Referenzen



# Operationen auf doppelt verketteter Liste

**Einfügen von Element an Stelle i:**

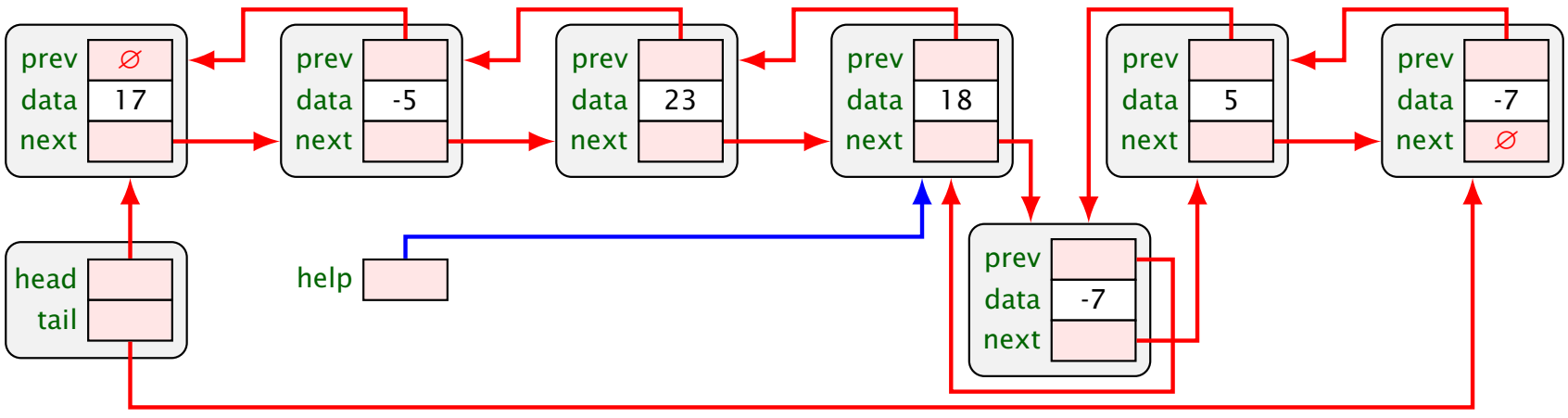
- Zugriff auf Element `i-1`
- umhängen von Referenzen



# Operationen auf doppelt verketteter Liste

**Einfügen von Element an Stelle i:**

- Zugriff auf Element `i-1`
- umhängen von Referenzen



# Eigenschaften doppelt verkettete Liste

Doppelt verkettete Liste

Vorteile:

- Durchlauf in beiden Richtungen möglich
- Einfügen/Löschen potentiell einfacher, da man sich Vorgänger nicht extra merken muss

Nachteile:

- zusätzlicher Speicher erforderlich für zwei Referenzen
- Referenzverwaltung komplizierter und fehleranfällig

# Zusammenfassung Felder

Ein **Feld** A kann repräsentiert werden als:

- **Array**
- **verkettete Liste** (linked list)
- **doppelt verkettete Liste** (doubly linked list)

**Eigenschaften:**

- einfach und flexibel
- aber manche Operationen aufwendig

# Zusammenfassung Felder

Ein **Feld** A kann repräsentiert werden als:

- **Array**
- **verkettete Liste** (linked list)
- **doppelt verkettete Liste** (doubly linked list)

**Eigenschaften:**

- einfach und flexibel
- aber manche Operationen aufwendig

# Definition Abstrakter Datentyp

**Abstrakter Datentyp (englisch: abstract data type, ADT)** Ein abstrakter Datentyp ist ein mathematisches **Modell** für bestimmte Datenstrukturen mit vergleichbarem Verhalten.

Ein abstrakter Datentyp wird indirekt definiert über

- mögliche Operationen auf ihm sowie
- mathematische Bedingungen (oder: constraints) über die Auswirkungen der Operationen (u.U. auch die Kosten der Operationen).

# Beispiel abstrakter Datentyp: abstrakte Variable

Abstrakte Variable V ist eine veränderliche Dateneinheit mit zwei Operationen

- `load(V)` liefert einen Wert
- `store(V, x)` wobei `x` ein Wert ist

und der Bedingung

- `load(V)` liefert immer den Wert `x` der letzten Operation `store(V, x)`

# Beispiel abstrakter Datentyp: abstrakte Liste (Teil 1)

Abstrakte Liste L ist ein Datentyp

mit Operationen

- `pushFront(L, x)` liefert eine Liste
- `front(L)` liefert ein Element
- `rest(L)` liefert eine Liste

und den Bedingungen

- ist x Element, L Liste, dann liefert `front(pushFront(L, x))` das Element x.
- ist x Element, L Liste, dann liefert `rest(pushFront(L, x))` die Liste L.

# Beispiel abstrakter Datentyp: abstrakte Liste (Teil 2)

Abstrakte Liste `L`. Weitere Operationen sind

- `isEmpty(L)` liefert `true` oder `false`
- `initialize()` liefert eine Listeninstanz

mit den Bedingungen

- `initialize()` $\neq$ `L` für jede Liste `L` (d.h. jede neue Liste ist separat von alten Listen)
- `isEmpty(initialize()) == true` (d.h. eine neue Liste ist leer)
- `isEmpty(pushFront(L, x)) == false` (d.h. eine Liste ist nach einem pushFront nicht leer)

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein Stack ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem Last In – First Out (LIFO) Prinzip mit den Eigenschaften

- löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- nur das **letzte Element** darf manipuliert werden.

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein Stack ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem Last In – First Out (LIFO) Prinzip mit den Eigenschaften

- löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- nur das **letzte Element** darf manipuliert werden.

**Operationen auf Stacks:**

- `push:` legt ein Element auf den Stack (einfügen)
- `pop:` entfernt das letzte Element vom Stack (löschen)
- `top:` liefert das letzte Stack-Element
- `isEmpty:` liefert `true` falls Stack leer
- `initialize:` Stack erzeugen und in Anfangszustand (leer) setzen

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein Stack ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem Last In – First Out (LIFO) Prinzip mit den Eigenschaften

- löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- nur das **letzte Element** darf manipuliert werden.

# Definition Stack (exakter)

Stack S ist ein abstrakter Datentyp mit Operationen

- `pop(S)` liefert einen Wert
- `push(S, x)` wobei `x` ein Wert

mit der Bedingung

- ist `x` Wert und `V` Variable, dann ist die Sequenz `push(S, x); V = pop(S);` äquivalent zu `V = x;`

# Definition Stack (exakter)

Stack S ist ein abstrakter Datentyp mit Operationen

- `pop(S)` liefert einen Wert
- `push(S, x)` wobei `x` ein Wert

mit der Bedingung

- ist `x` Wert und `V` Variable, dann ist die Sequenz `push(S, x); V = pop(S);` äquivalent zu `V = x;`

sowie der Operation

- `top(S)` liefert einen Wert

mit der Bedingung

- ist `x` Wert und `V` Variable, dann ist die Sequenz `push(S, x); V = top(S);` äquivalent zu `push(S, x); V = x;`

# Definition Stack (exakter, Teil 2)

Stack S. Weitere Operationen sind

- `isEmpty(S)` liefert `true` oder `false`
- `initialize()` liefert eine Stackinstanz

mit den Bedingungen

- `initialize() ≠ S` für jeden Stack `S` (d.h. jeder neue Stack ist separat von alten Stacks)
- `isEmpty(initialize()) == true` (d.h. ein neuer Stack ist leer)
- `isEmpty(push(S, x)) == false` (d.h. ein Stack nach push ist nicht leer)

# Anwendungsbeispiele Stack

Call-Stack bei Funktionsaufrufen

Einfache Vorwärts- / Rückwärts Funktion in Software

- z.B. im Internet-Browser

Syntaxanalyse eines Programms

- z.B. zur Erkennung von Syntax-Fehlern durch Compiler

Auswertung arithmetischer Ausdrücke

# Auswertung arithmetischer Ausdrücke

Gegeben sei ein vollständig geklammerter, einfacher arithmetischer Ausdruck mit Bestandteilen Zahl, +, *, =

**Beispiel:** (3 * (4 + 5)) =

# Auswertung arithmetischer Ausdrücke

Gegeben sei ein vollständig geklammerter, einfacher arithmetischer Ausdruck mit Bestandteilen Zahl, +, *, =

**Beispiel:** (3 * (4 + 5)) =

**Schema:**

- arbeite Ausdruck von links nach rechts ab, speichere jedes Zeichen ausser ) und = in Stack S
- bei ) werte die 3 obersten Elemente von S aus, dann entferne die passende Klammer ( vom Stack S und speichere Ergebnis in Stack S
- bei = steht das Ergebnis im obersten Stack-Element von S

# Beispiel: Auswertung arithmetischer Ausdrücke

$$(((3 + 1) \cdot 7) + (14 - 3))$$

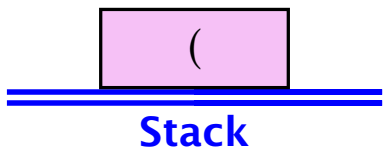
# Beispiel: Auswertung arithmetischer Ausdrücke

**Stack**

| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

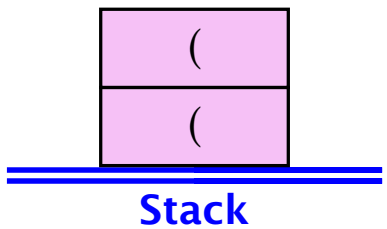
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

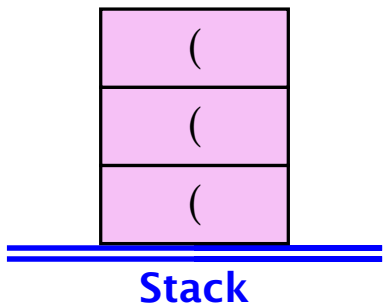
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

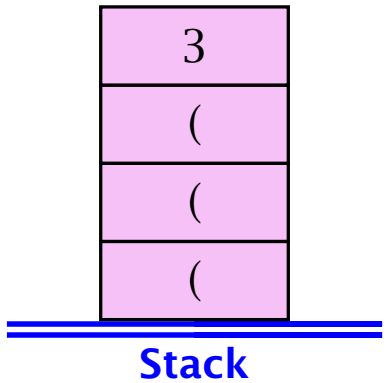
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

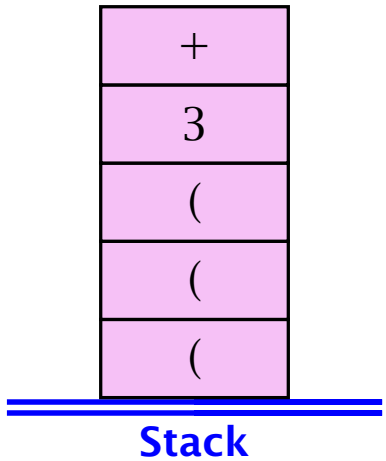
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

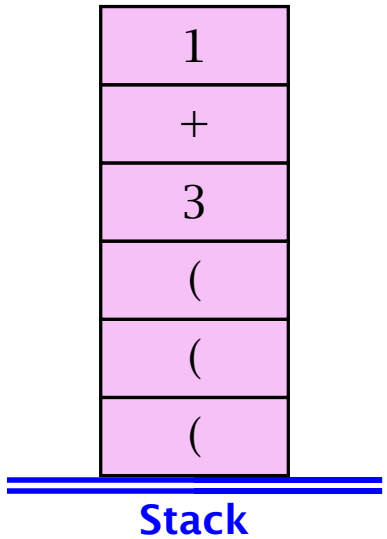
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

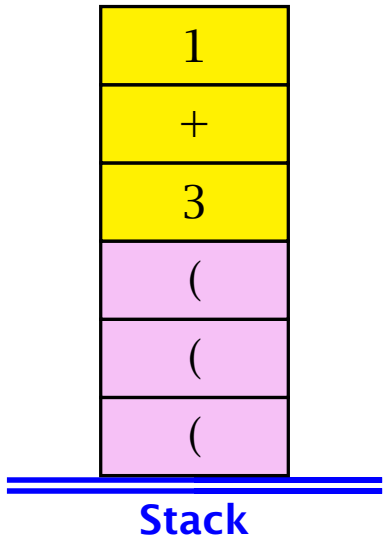
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

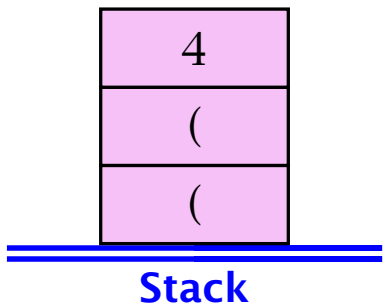
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

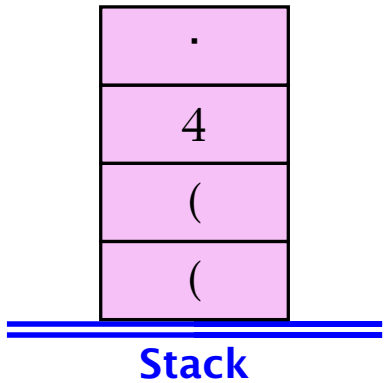
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

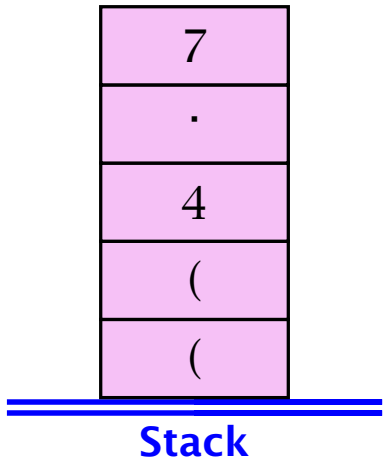
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

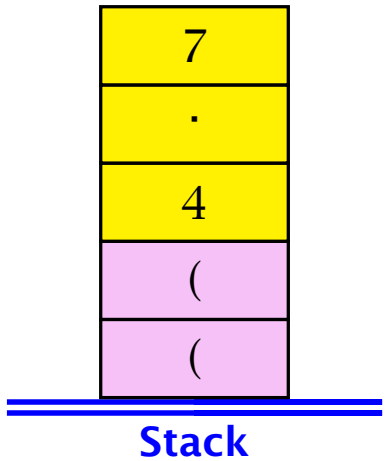
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

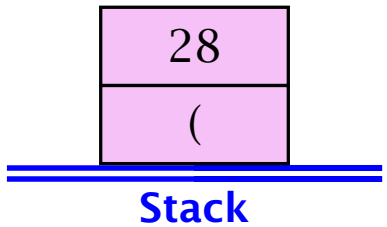
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

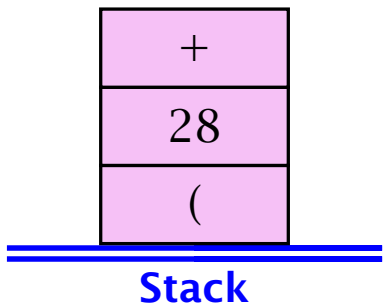
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

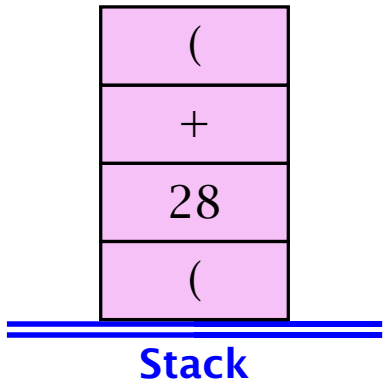
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

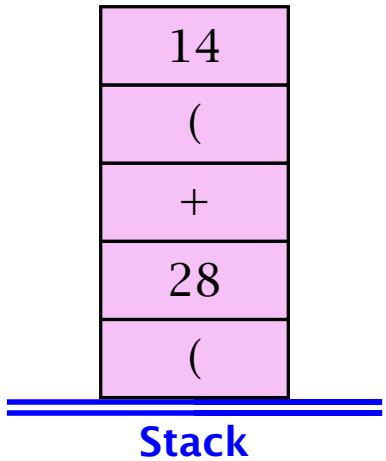
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

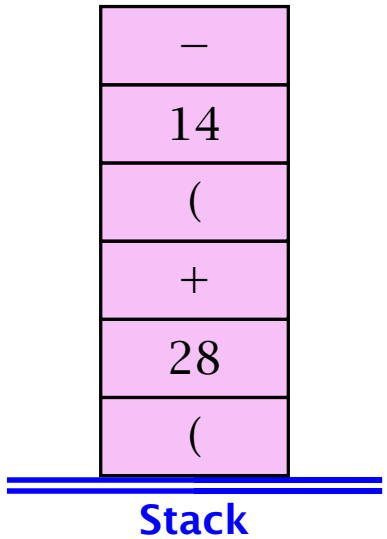
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

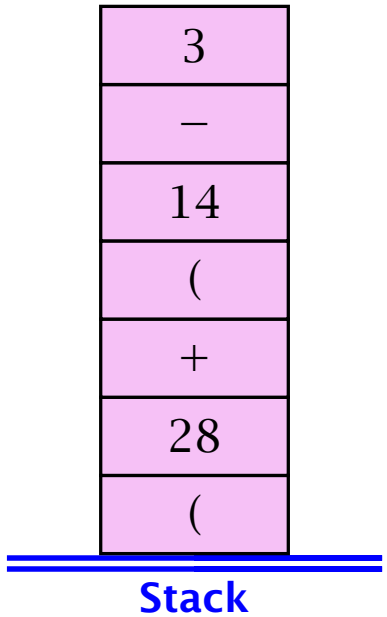
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

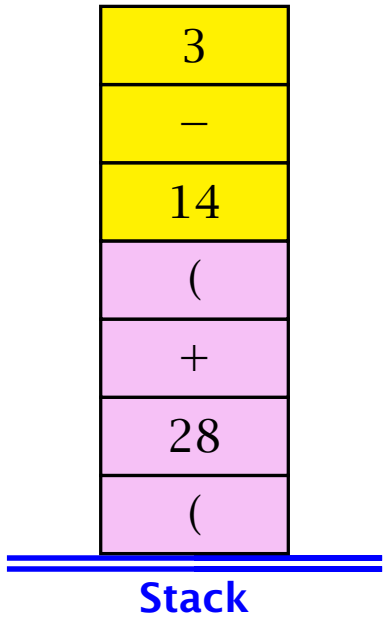
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

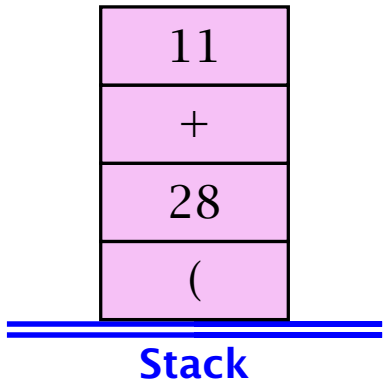
# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

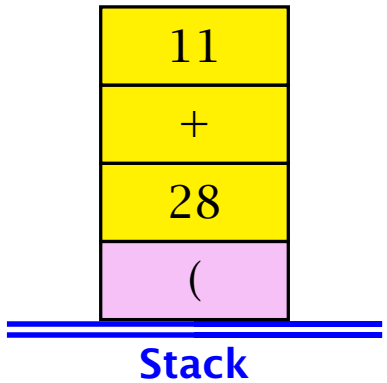
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

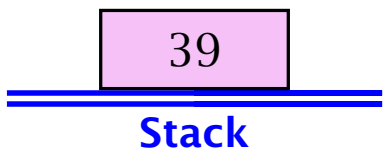
# Beispiel: Auswertung arithmetischer Ausdrücke




| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |

$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke




| ( | ( | ( | 3 | + | 1 | ) | · | 7 | ) | + | ( | 14 | − | 3 | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Implementation Stack

**Stack ist abstrakter Datentyp.**

- Implementation ist nicht festgelegt
- nur Operationen und Bedingungen sind festgelegt

# Implementation Stack

**Stack ist abstrakter Datentyp.**

- Implementation ist nicht festgelegt
- nur Operationen und Bedingungen sind festgelegt

Stack kann auf viele Arten implementiert werden, zum Beispiel als:

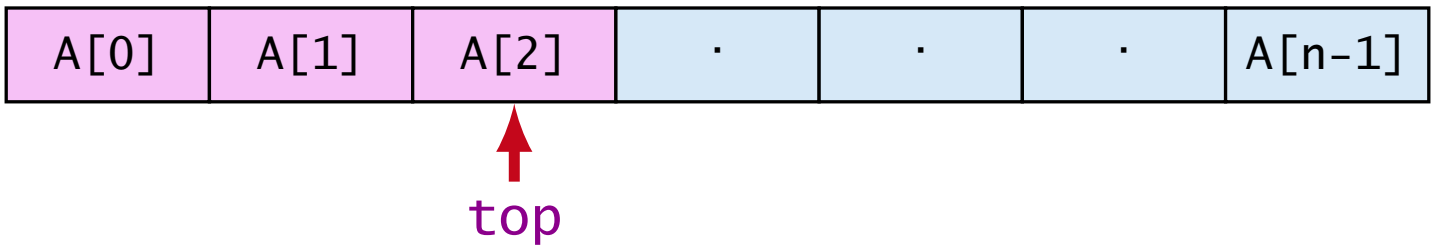
- Array
- verkettete Liste

# Implementation Stack via Array

Stack-Elemente im Array (Länge n) speichern

oberstes Stack-Element merken mittels Variable top

falls Stack **leer** ist top == -1

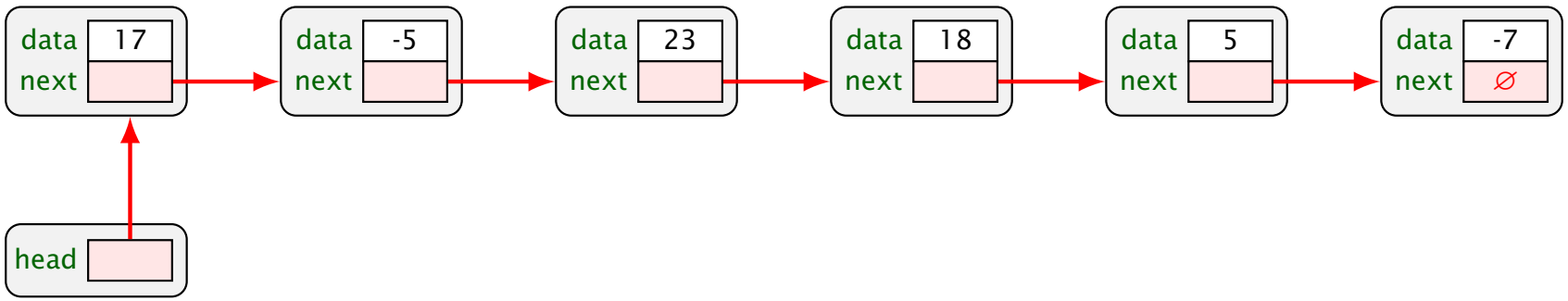



- push(x) inkrementiert top und speichert x in A[top]
- pop() liefert A[top] zurück und dekrementiert top
- top() liefert A[top] zurück

# Implementation Stack als verkettete Liste

Stack-Elemente speichern in verketteter Liste

oberstes Stack-Element wird durch head-Referenz markiert




- `push(x)` fügt Element an erster Position ein
- `pop()` liefert Element an erster Position zurück und entfernt es
- `top()` liefert Element an erster Position zurück

# Zusammenfassung Stack

Stack ist abstrakter Datentyp als Metapher für einen Stapel

- wesentliche Operationen: **push, pop**

Implementation als Array

- fixe Größe (entweder Speicher verschwendet oder zu klein)
- push, pop sehr effizient

Implementation als verkettete Liste

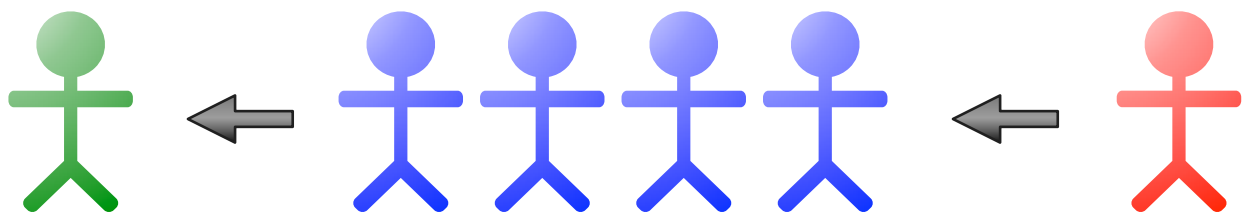
- dynamische Größe, aber Platz für Zeiger "verschwendet"
- push, pop effizient
- eventuell nicht cache-effizient

# Definition Queue

**Queue (oder deutsch: Warteschlange)**

Eine Queue ist ein abstrakter Datentyp. Sie beschreibt eine spezielle Listenstruktur nach dem First In – First Out (FIFO) Prinzip mit den Eigenschaften

- einfügen ist nur **am Ende** der Liste erlaubt,
- entfernen ist nur **am Anfang** der Liste erlaubt.



# Definition Queue

**Queue (oder deutsch: Warteschlange)**
Eine Queue ist ein abstrakter Datentyp. Sie beschreibt eine spezielle Listenstruktur nach dem First In – First Out (FIFO) Prinzip mit den Eigenschaften

- einfügen ist nur **am Ende** der Liste erlaubt,
- entfernen ist nur **am Anfang** der Liste erlaubt.

**Operationen auf Queues:**

- `enqueue`: fügt ein Element am Ende der Schlange hinzu
- `dequeue`: entfernt das erste Element der Schlange
- `isEmpty`: liefert `true` falls Queue leer
- `initialize`: Queue erzeugen und in Anfangszustand (leer) setzen

# Definition Queue (exakter)

Queue `Q` ist ein abstrakter Datentyp mit Operationen

- `dequeue(Q)` liefert einen Wert
- `enqueue(Q, x)` wobei `x` ein Wert
- `isEmpty(Q)` liefert `true` oder `false`
- `initialize` liefert eine Queue Instanz

und mit Bedingungen

- ist `x` Wert, `V` Variable, `Q` **leere** Queue, dann ist Sequenz
  `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x`.
- sind `x,y` Werte, `V` Variable und `Q` Queue, dann ist Sequenz
  `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)`
  äquivalent zu `enqueue(Q,x); V=dequeue(Q); enqueue(Q, y)`
- `initialize() ≠ Q` für jede Queue `Q`
- `isEmpty(initialize()) == true`
- `isEmpty(enqueue(Q, x)) == false`

# Definition Queue (exakter)

Queue `Q` ist ein abstrakter Datentyp mit Operationen

- `dequeue(Q)` liefert einen Wert
- `enqueue(Q, x)` wobei `x` ein Wert
- `isEmpty(Q)` liefert `true` oder `false`
- `initialize` liefert eine Queue Instanz

und mit Bedingungen

- ist `x` Wert, `V` Variable, `Q` **leere** Queue, dann ist Sequenz `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x`.
- sind `x,y` Werte, `V` Variable und `Q` Queue, dann ist Sequenz `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)` äquivalent zu `enqueue(Q,x); V=dequeue(Q); enqueue(Q, y)`
- `initialize() ≠ Q` für jede Queue `Q`
- `isEmpty(initialize()) == true`
- `isEmpty(enqueue(Q, x)) == false`

# Definition Queue (exakter)

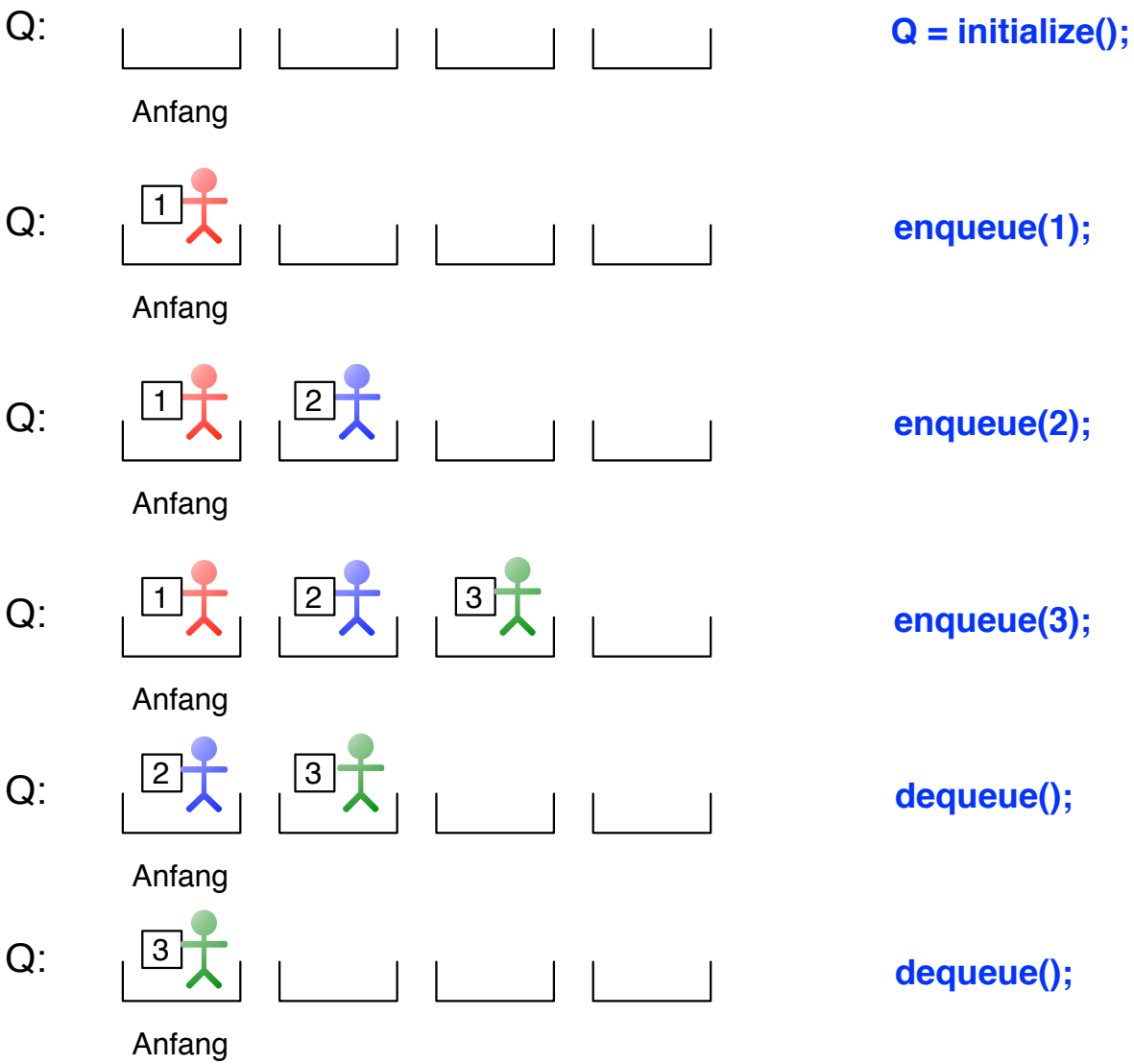
Queue `Q` ist ein abstrakter Datentyp mit Operationen

- `dequeue(Q)` liefert einen Wert
- `enqueue(Q, x)` wobei `x` ein Wert
- `isEmpty(Q)` liefert `true` oder `false`
- `initialize` liefert eine Queue Instanz

und mit Bedingungen

- ist `x` Wert, `V` Variable, `Q` **leere** Queue, dann ist Sequenz `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x`.
- sind `x,y` Werte, `V` Variable und `Q` Queue, dann ist Sequenz `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)` äquivalent zu `enqueue(Q,x); V=dequeue(Q); enqueue(Q, y)`
- `initialize() ≠ Q` für jede Queue Q
- `isEmpty(initialize()) == true`
- `isEmpty(enqueue(Q, x)) == false`

# Beispiel: Queue



# Anwendungsbeispiele Queue

- Druckerwarteschlange
- Playlist von iTunes (oder ähnlichem Musikprogramm)
- Kundenaufträge bei Webshops
- Warteschlange für Prozesse im Betriebssystem (Multitasking)

# Anwendungsbeispiel Stack und Queue

**Palindrom**
Ein Palindrom ist eine Zeichenkette, die von vorn und von hinten gelesen gleich bleibt.

**Beispiel**: Reittier

Erkennung ob Zeichenkette ein Palindrom ist

- ein **Stack** kann die Reihenfolge der Zeichen umkehren
- eine **Queue** behält die Reihenfolge der Zeichen

# Palindromerkennung

```
1  Input: Zeichenkette str mit Laenge n
2  Output: true falls str Palindrom; sonst false
3
4  Stack S;
5  Queue Q;
6
7  i = 0;
8  while (i<n)
9      S.push(str[i]);
10     Q.enqueue(str[i]);
11     i++;
12 i = 0;
13 while (i<n)
14     s = S.pop();
15     q = Q.dequeue();
16     if (s != q) return false;
17     i++;
18 return true;
```

# Implementation Queue

Auch Queue ist abstrakter Datentyp.

- Implementation ist nicht festgelegt
- nur Operationen und Bedingungen sind festgelegt

Queue kann auf viele Arten implementiert werden, zum Beispiel als:

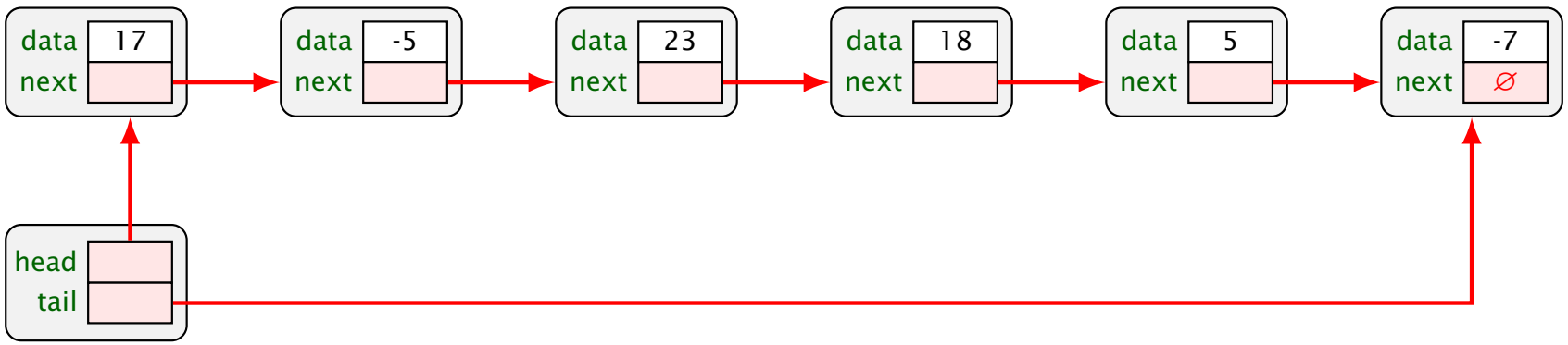
- verkettete Liste
- Array

# Implementation Queue als verkettete Liste

Queue-Elemente speichern in verketteter Liste

Anfang der Queue wird durch `head`-Referenz markiert

Ende der Queue wird durch extra `tail`-Referenz markiert



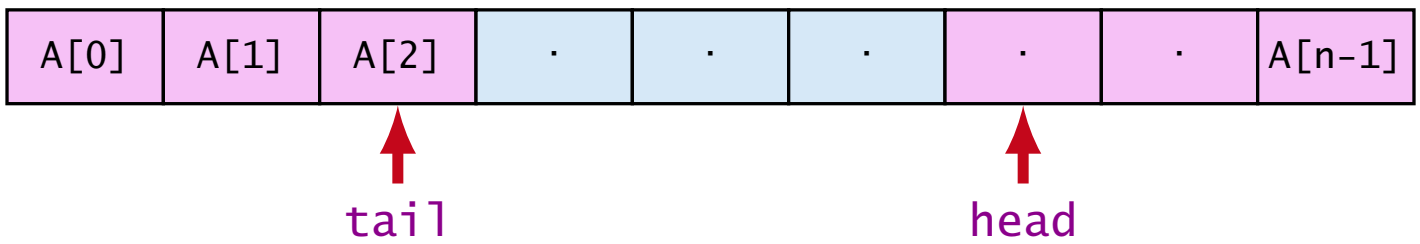

- `enqueue(x)` fügt Element bei `head`-Referenz ein
- `dequeue()` liefert Element bei `tail`-Referenz zurück und entfernt es

# Implementation Queue via Array

Queueelemente in Array (Länge n) speichern

Anfang der Queue wird durch Index head markiert

Ende der Queue wird durch Index tail markiert




- enqueue(x) fügt Element bei Index (tail+1)%n ein
- dequeue liefert Element bei Index head zurück und entfernt es durch Inkrement von head (head=(head+1)%n)

# Implementation Queue als zwei Stacks

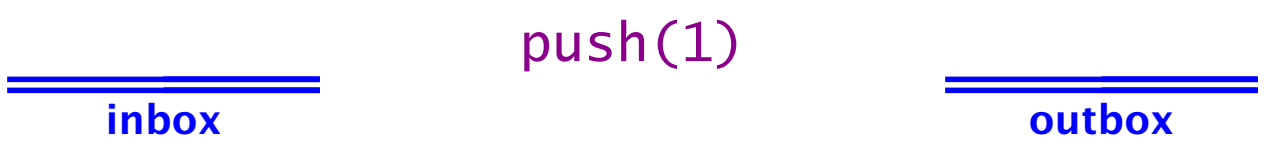
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

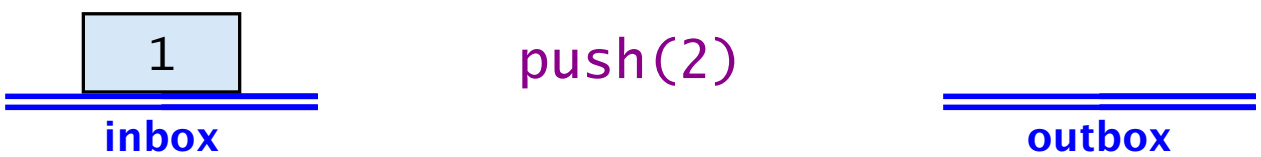
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

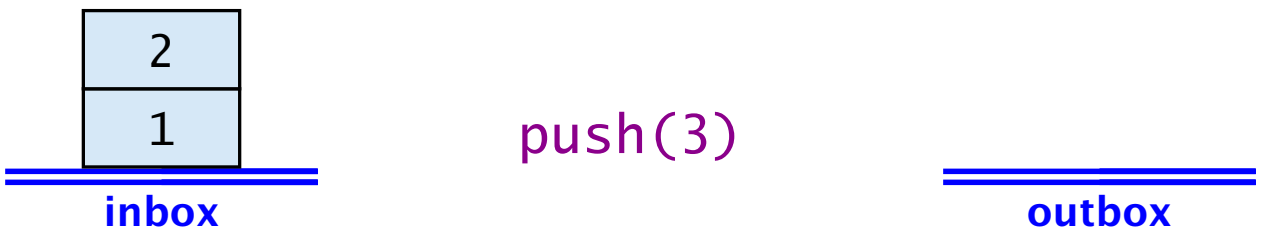
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück

# Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück

# Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück

# Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück

# Implementation Queue als zwei Stacks

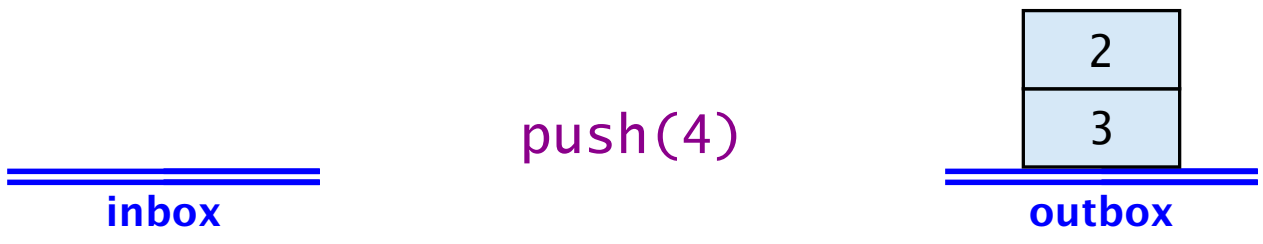
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

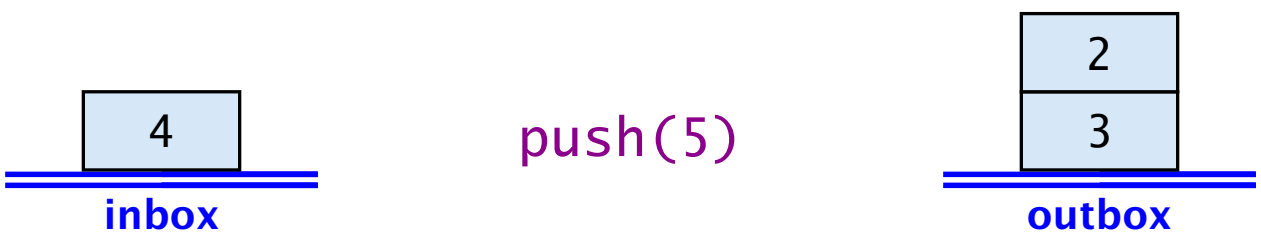
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück

# Implementation Queue als zwei Stacks

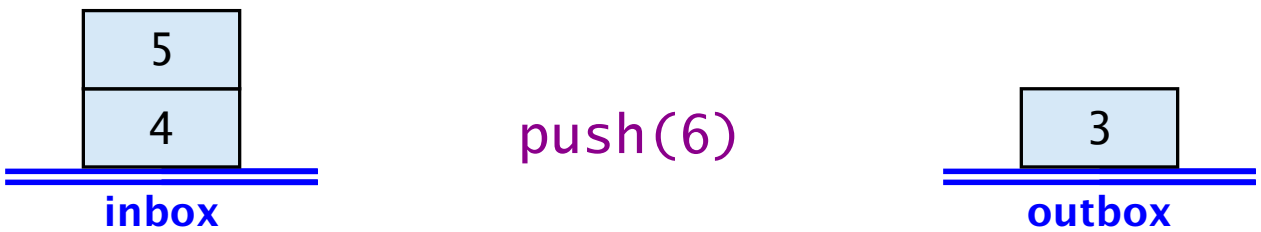
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

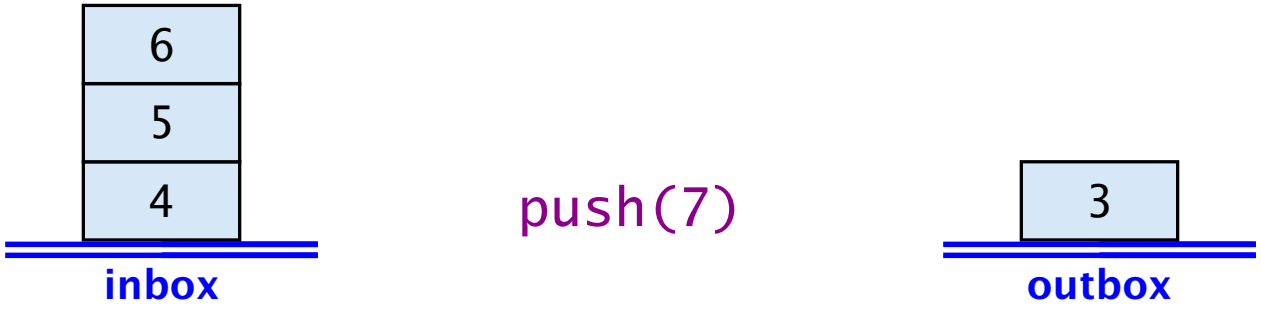
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

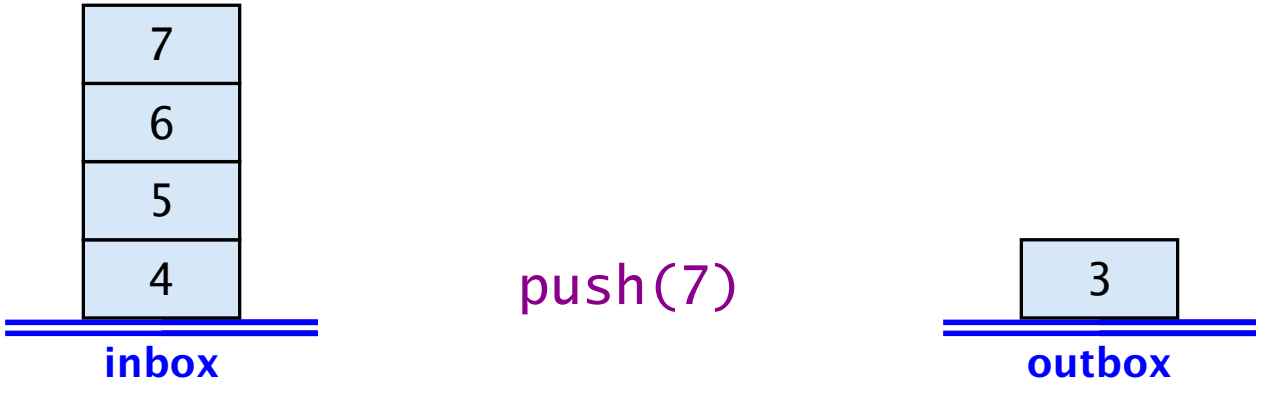
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- falls `outbox` leer, kopiere alle Elemente von `inbox` zu outbox: `outbox.push( inbox.pop() )`
- `Q.dequeue()` liefert `outbox.pop()` zurück



# Zusammenfassung Queue

Queue ist abstrakter Datentyp als Metapher für eine Warteschlange

- wesentliche Operationen: **enqueue, dequeue**

Implementation als verkettete Liste

- dynamische Größe, aber Platz für Referenzen "verschwendet"
- enqueue, dequeue effizient
- nicht cache-effizient

Implementation als Array

- fixe Größe (entweder Speicher verschwendet oder zu klein)
- enqueue, dequeue sehr effizient

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $O(n^2)$“.
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche“.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$“.
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche“.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$“.
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche“.

# 5 Effizienz von Algorithmen

**Wie messen wir?**

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$“.
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche“.

# 5 Effizienz von Algorithmen

**Eingabelänge**

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

# 5 Effizienz von Algorithmen

**Eingabelänge**

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

# 5 Effizienz von Algorithmen

**Eingabelänge**

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an bits)
- die Anzahl der Argumente

**Example 1**

Angenommen $n$ Zahlen aus dem Bereich $\{1, \ldots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge $n$ ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

# 5 Effizienz von Algorithmen

**Eingabelänge**

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an bits)
- die Anzahl der Argumente

**Example 1**

Angenommen $n$ Zahlen aus dem Bereich $\{1, \ldots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge $n$ ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

# 5 Effizienz von Algorithmen

**Eingabelänge**

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an bits)
- die Anzahl der Argumente

**Example 1**

Angenommen $n$ Zahlen aus dem Bereich $\{1, \ldots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge $n$ ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

# Rechenmodell

**Wie messen wir**

# Rechenmodell

**Wie messen wir**

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

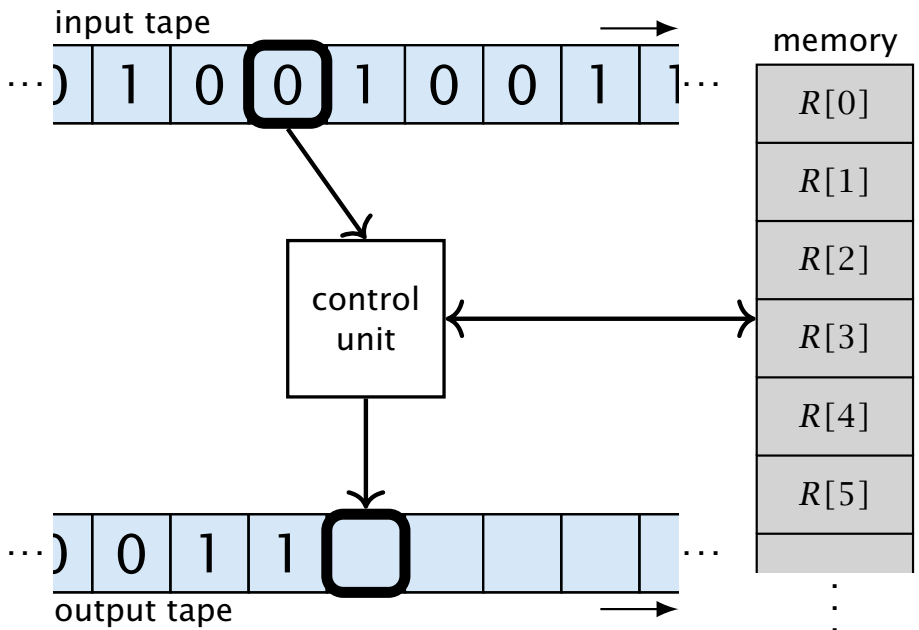
# Rechenmodell

**Wie messen wir**

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

# Random Access Machine (RAM)

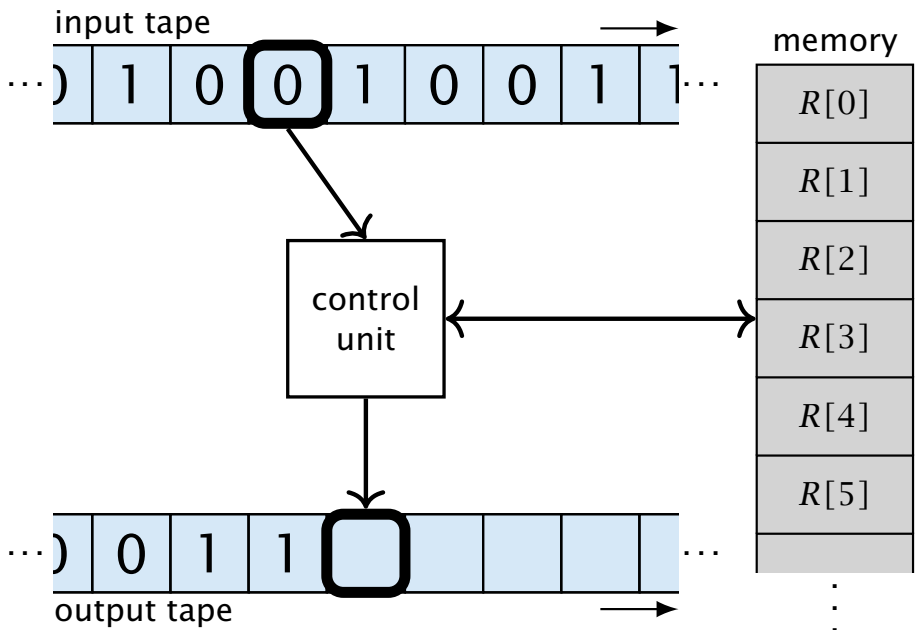
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.




Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

# Random Access Machine (RAM)

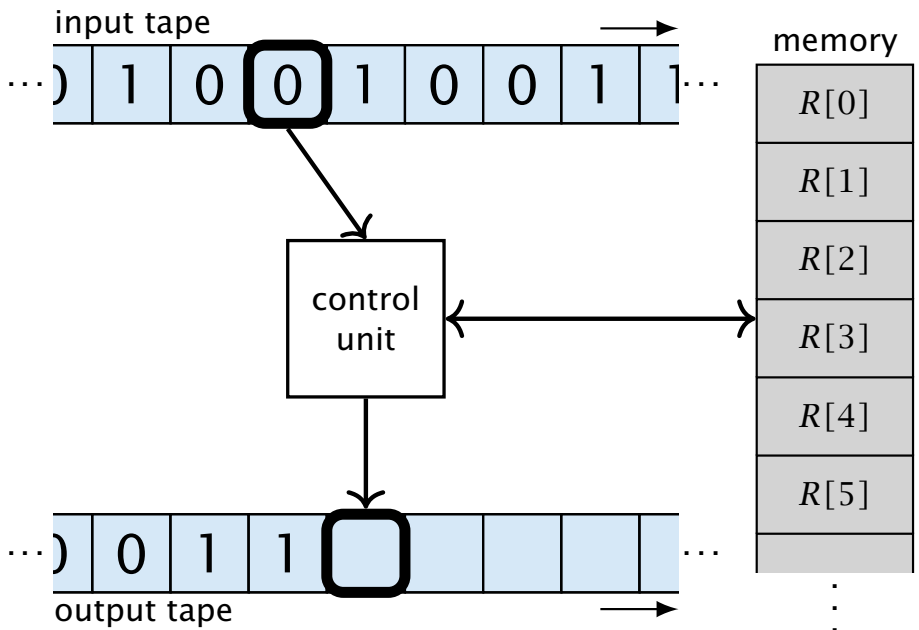
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.




Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

# Random Access Machine (RAM)

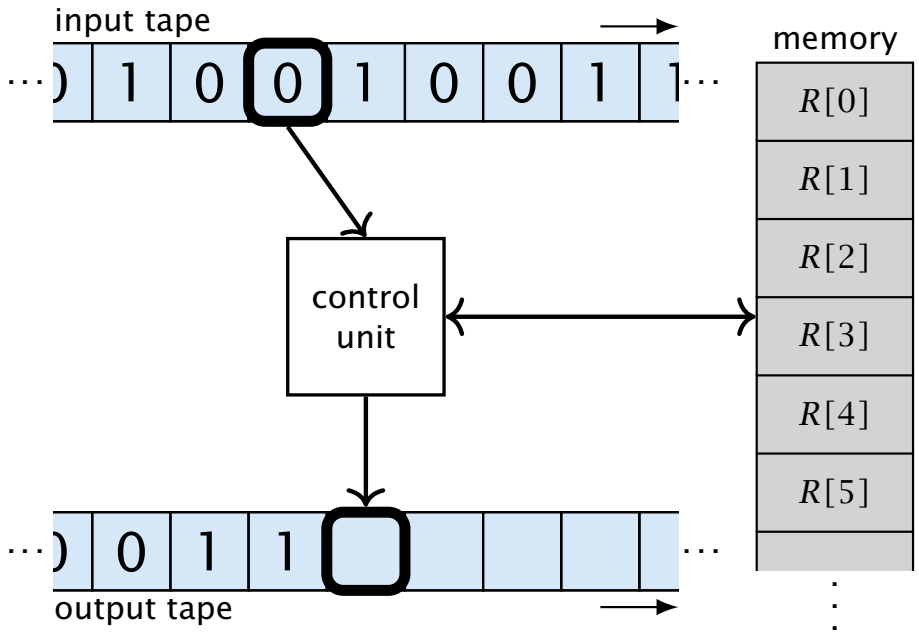
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.




Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

# Random Access Machine (RAM)

- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.




Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

# Random Access Machine (RAM)

## Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[i] := R[j]$
  - $R[i] := R[j] + R[k]$
- **indirekte** Adressierung
  - $R[i] := R[j] + R[k]$
    - Lese den Inhalt des $R[j]$-ten Registers in das $i$-te Register
  - $R[i] := R[j] + R[k]$
    - Lese den Inhalt des $i$-ten Registers in das $j$-te Register

# Random Access Machine (RAM)

## Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - LOAD $i$
  - STORE $i$
- **indirekte** Adressierung
  - LOAD* $i$
  - STORE* $i$

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
- **indirekte** Adressierung

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
- **indirekte** Adressierung

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirekte** Adressierung

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[j]$ := $R[i]$
  - $R[j]$ := 4
- **indirekte** Adressierung

# Random Access Machine (RAM)

## Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[j]$ := $R[i]$
  - $R[j]$ := $4$
- indirekte Adressierung

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirekte** Adressierung
  - $R[j] := R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das $j$-te Register.
  - $R[R[i]] := R[j]$
    lädt den Inhalt des $j$-ten Registers in das $R[i]$-te Register

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirekte** Adressierung
  - $R[j] := R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das $j$-te Register.
  - $R[R[i]] := R[j]$
    lädt den Inhalt des $j$-ten Registers in das $R[i]$-te Register

# Random Access Machine (RAM)

**Operationen**

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ $i$
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE $i$
- Registertransfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirekte** Adressierung
  - $R[j] := R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das $j$-te Register.
  - $R[R[i]] := R[j]$
    lädt den Inhalt des $j$-ten Registers in das $R[i]$-te Register

# Random Access Machine (RAM)

## Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - jump $x$
    springe zur Position $x$ im Programm
    setze Befehlszähler auf $x$
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - jumpz $x$ $R[i]$
    springe zu $x$ falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - jumpi $i$
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

# Random Access Machine (RAM)

## Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` $x$
    springe zur Position $x$ im Programm
    setze Befehlszähler auf $x$
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` $x$ $R[i]$
    springe zu $x$ falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` $i$
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+, -, \times, /$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

# Random Access Machine (RAM)

## Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` $x$
    springe zur Position $x$ im Programm
    setze Befehlszähler auf $x$
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` $x$ $R[i]$
    springe zu $x$ falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` $i$
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+, -, \times, /$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

# Random Access Machine (RAM)

## Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` $x$
    springe zur Position $x$ im Programm
    setze Befehlszähler auf $x$
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` $x$ $R[i]$
    springe zu $x$ falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` $i$
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+, -, \times, /$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

# Random Access Machine (RAM)

## Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` $x$
    springe zur Position $x$ im Programm
    setze Befehlszähler auf $x$
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` $x$ $R[i]$
    springe zu $x$ falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` $i$
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i] := R[j] + R[k];$
    $R[i] := -R[k];$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

# Random Access Machine (RAM)

## Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` $x$
    springe zur Position $x$ im Programm
    setze Befehlszähler auf $x$
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` $x$ $R[i]$
    springe zu $x$ falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` $i$
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i]$ := $R[j]$ + $R[k]$;
    $R[i]$ := -$R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

# Rechenmodell

Man nimmt normalerweise an, dass jeder Befehl eine Zeiteinheit kostet.

# Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

$$C_{\text{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

$$C_{\text{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

$$C_{\text{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

| | |
|---|---|
| $C(x)$ | Kosten für Eingabe $x$ |
| $\|x\|$ | Eingabelänge von $x$ |
| $I_n$ | Menge der Eingaben mit Länge $n$ |

# Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

$$C_{\text{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

$$C_{\text{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

$$C_{\text{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

| | |
|---|---|
| $C(x)$ | Kosten für Eingabe $x$ |
| $\|x\|$ | Eingabelänge von $x$ |
| $I_n$ | Menge der Eingaben mit Länge $n$ |

# Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

$$C_{\text{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

$$C_{\text{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

$$C_{\text{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

| | |
|---|---|
| $C(x)$ | Kosten für Eingabe $x$ |
| $\lvert x\rvert$ | Eingabelänge von $x$ |
| $I_n$ | Menge der Eingaben mit Länge $n$ |

# Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

$$C_{\text{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

$$C_{\text{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

$$C_{\text{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

| | |
|---|---|
| $C(x)$ | Kosten für Eingabe $x$ |
| $\lvert x\rvert$ | Eingabelänge von $x$ |
| $I_n$ | Menge der Eingaben mit Länge $n$ |

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von $n$. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von $n$. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von $n$. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von $n$. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

## Formale Definition

Sei $f$ eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als $f$ wachsen)

# Asymptotische Notation

## Formale Definition

Sei $f$ eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als $f$ wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als $f$ wachsen)

# Asymptotische Notation

## Formale Definition

Sei $f$ eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als $f$ wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als $f$ wachsen)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (Funktionen die asymptotisch gleiches Wachstum wie $f$ haben)

# Asymptotische Notation

## Formale Definition

Sei $f$ eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als $f$ wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als $f$ wachsen)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (Funktionen die asymptotisch gleiches Wachstum wie $f$ haben)
- $o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotische langsamer als $f$ wachsen)

# Asymptotische Notation

## Formale Definition

Sei $f$ eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als $f$ wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als $f$ wachsen)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (Funktionen die asymptotisch gleiches Wachstum wie $f$ haben)
- $o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotische langsamer als $f$ wachsen)
- $\omega(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch schneller als $f$ wachsen)

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). $f$ und $g$ seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). $f$ und $g$ seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). $f$ und $g$ seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). $f$ und $g$ seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). $f$ und $g$ seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
- $g \in \omega(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

# Asymptotische Notation

**Missbrauch dieser Notation**

1. Man schreibt $f = \mathcal{O}(g)$, anstatt $f \in \mathcal{O}(g)$. Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt $f(n) = \mathcal{O}(g(n))$, anstatt $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

# Asymptotische Notation

**Missbrauch dieser Notation**

1. Man schreibt $f = \mathcal{O}(g)$, anstatt $f \in \mathcal{O}(g)$. Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt $f(n) = \mathcal{O}(g(n))$, anstatt $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

# Asymptotische Notation

**Missbrauch dieser Notation**

1. Man schreibt $f = \mathcal{O}(g)$, anstatt $f \in \mathcal{O}(g)$. Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt $f(n) = \mathcal{O}(g(n))$, anstatt $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

# Asymptotische Notation

Man kann einen Ausdruck mit asymptotischer Notation als **Menge** ansehen:

$$n^2 \cdot \mathcal{O}(n) + \mathcal{O}(\log n)$$

repräsentiert

$$\Big\{ f : \mathbb{N} \to \mathbb{R}^+ \mid f(n) = n^2 \cdot g(n) + h(n) \quad \text{mit } g(n) \in \mathcal{O}(n) \text{ und } h(n) \in \mathcal{O}(\log n) \Big\}$$

# Asymptotische Notation

**Lemma 2**

*Seien $f, g$ Funktionen mit $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ (das gleiche für $g$). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* $c$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für $\Omega$ und $\Theta$.*

# Asymptotische Notation

**Lemma 2**

*Seien $f, g$ Funktionen mit $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ (das gleiche für $g$). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* $c$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für $\Omega$ und $\Theta$.*

# Asymptotische Notation

**Lemma 2**

*Seien $f, g$ Funktionen mit $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ (das gleiche für $g$). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* $c$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für $\Omega$ und $\Theta$.*

# Asymptotische Notation

**Lemma 2**

*Seien $f, g$ Funktionen mit $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ (das gleiche für $g$). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* $c$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für $\Omega$ und $\Theta$.*

# Asymptotische Notation

**Lemma 2**

*Seien $f, g$ Funktionen mit $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ (das gleiche für $g$). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* $c$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für $\Omega$ und $\Theta$.*

# Rechenregel für $\mathcal{O}$-Notation

Zu zeigen:

$$T_1(n) + T_2(n) = \mathcal{O}\big(\max(f(n), g(n))\big)$$

für $T_1(n) \in \mathcal{O}(f(n))$ und $T_2(n) \in \mathcal{O}(g(n))$.

- da $T_1(n) = \mathcal{O}(f(n))$, gibt es $c_1 > 0$ und $n_1 \in \mathbb{N}$ mit $T_1(n) \leq c_1 f(n)$ für $n \geq n_1$
- da $T_2(n) = \mathcal{O}(g(n))$, gibt es $c_2 > 0$ und $n_2 \in \mathbb{N}$ mit $T_2(n) \leq c_2 g(n)$ für $n \geq n_2$
- Setze $n_0 := \max(n_1, n_2)$, dann ist für $n \geq n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2) \max(f(n), g(n)) \quad \checkmark \end{aligned}$$

# Rechenregel für $\mathcal{O}$-Notation

Zu zeigen:

$$T_1(n) + T_2(n) = \mathcal{O}\big(\max(f(n), g(n))\big)$$

für $T_1(n) \in \mathcal{O}(f(n))$ und $T_2(n) \in \mathcal{O}(g(n))$.

- da $T_1(n) = \mathcal{O}(f(n))$, gibt es $c_1 > 0$ und $n_1 \in \mathbb{N}$ mit $T_1(n) \leq c_1 f(n)$ für $n \geq n_1$
- da $T_2(n) = \mathcal{O}(g(n))$, gibt es $c_2 > 0$ und $n_2 \in \mathbb{N}$ mit $T_2(n) \leq c_2 g(n)$ für $n \geq n_2$
- Setze $n_0 := \max(n_1, n_2)$, dann ist für $n \geq n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2) \max(f(n), g(n)) \quad \checkmark \end{aligned}$$

# Laufzeiten

| *Funktion* | *Eingabelänge n* | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $f(n)$ | $10$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
| $\log n$ | 33ns | 66ns | 0.1µs | 0.1µs | 0.2µs | 0.2µs | 0.2µs | 0.3µs |
| $\sqrt{n}$ | 32ns | 0.1µs | 0.3µs | 1µs | 3.1µs | 10µs | 31µs | 0.1ms |
| $n$ | 100ns | 1µs | 10µs | 0.1ms | 1ms | 10ms | 0.1s | 1s |
| $n \log n$ | 0.3µs | 6.6µs | 0.1ms | 1.3ms | 16ms | 0.2s | 2.3s | 27s |
| $n^{3/2}$ | 0.3µs | 10µs | 0.3ms | 10ms | 0.3s | 10s | 5.2min | 2.7h |
| $n^2$ | 1µs | 0.1ms | 10ms | 1s | 1.7min | 2.8h | 11d | 3.2y |
| $n^3$ | 10µs | 10ms | 10s | 2.8h | 115d | 317y | $3.2 \cdot 10^5$y | |
| $1.1^n$ | 26ns | 0.1ms | $7.8 \cdot 10^{25}$y | | | | | |
| $2^n$ | 10µs | $4 \cdot 10^{14}$y | | | | | | |
| $n!$ | 36ms | $3 \cdot 10^{142}$y | | | | | | |

1 Operation = 10ns; 100MHz

Alter des Universums: ca. $13.8 \cdot 10^9$y

# Typische Laufzeitklassen

$\mathcal{O}$-Notation erlaubt Klassifizierung der Effizienz von Algorithmen

**$\Theta(1)$: konstante Laufzeit**

- unabhängig von Problemgröße
- *Beispiel:* Löschen von erstem Element in verketteter Liste

**$\Theta(\log n)$: logarithmische Laufzeit**

- Laufzeit wächst langsamer als Problemgröße
- typisch für Divide & Conquer Algorithmen
- *Beispiel:* Suchen in sortierter Liste

**$\Theta(n)$: lineare Laufzeit**

- Laufzeit wächst vergleichbar zur Problemgröße
- jedes Eingabe-Element erfordert $\mathcal{O}(1)$ Arbeit
- *Beispiele:* Suchen in unsortierter Liste, Löschen von Element im Array

# Typische Laufzeitklassen

## $\Theta(n \log n)$: "loglinear" Laufzeit

- Laufzeit wächst schneller als Problemgröße
- typisch für Divide & Conquer
- *Beispiele:* Quicksort, FFT

## $\Theta(n^2)$: quadratische Laufzeit

- typisch für Algorithmen, die Element paarweise kombinieren
- *Beispiele:* Insertion Sort, Matrix-Vektor Multiplikation

## $\Theta(n^3)$: kubische Laufzeit

- *Beispiel:* Matrix-Matrix Multiplikation

## $\Theta(2^n)$: exponentielle Laufzeit

- auch als "unlösbar" bezeichnet (intractable)
- *Beispiel:* Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

Sofern sich die angegebenen Laufzeiten auf ein Problem beziehen (und nicht auf einen konkreten Algorithmus zu Lösung eines Problems) handelt es sich um einen typischen Lösungsansatz für das jeweilige Problem. Zum Beispiel hat das Standardverfahren für die Matrixmultiplikation eine Laufzeit von $\Theta(n^3)$. Es gibt aber bessere Verfahren.

Ein wichtiges offenes Problem ist es ob es z.B. für das TSP-Problem einen Algorithmus mit polynomieller Laufzeit gibt.

# Asymptotische Notation

**Hinweise**

- Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- Für beliebige Konstanten $a, b$ gilt $\log_a n = \Theta(\log_b n)$. Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- Für diese Vorlesung: $\log n = \log_2 n$, d.h., wir nehmen $2$ als Standardbasis für den Logarithmus.

# Asymptotische Notation

**Hinweise**

- Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- Für beliebige Konstanten $a, b$ gilt $\log_a n = \Theta(\log_b n)$. Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- Für diese Vorlesung: $\log n = \log_2 n$, d.h., wir nehmen $2$ als Standardbasis für den Logarithmus.

# Asymptotische Notation

**Hinweise**

- Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- Für beliebige Konstanten $a, b$ gilt $\log_a n = \Theta(\log_b n)$. Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- Für diese Vorlesung: $\log n = \log_2 n$, d.h., wir nehmen $2$ als Standardbasis für den Logarithmus.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes $n$ überlegen.
- Aber:

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes $n$ überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes $n$ überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes $n$ überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes $n$ überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

# Komplexität der elementaren Bausteine

**Elementarer Verarbeitungsschritt:**

- $\mathcal{O}(1)$

**Sequenz:**

- Addition in $\mathcal{O}$-Notation

**Bedingter Verarbeitungsschritt:**

- Maximum von Komplexität von if und else Block, sowie
- $\mathcal{O}(1)$ für Auswertung der Bedingung

**Wiederholung:**

- Anzahl Wiederholungen multipliziert mit Komplexität Schleifenkörper, sowie
- Anzahl Wiederholungen + 1 multipliziert mit $\mathcal{O}(1)$ für Auswertung der Schleifenbedingung

# Komplexität Datenstrukturenoperationen

**Feld via Array:**

- elementAt $\mathcal{O}(1)$, insert $\mathcal{O}(n)$, erase $\mathcal{O}(n)$

**Feld via LinkedList:**

- elementAt $\mathcal{O}(n)$, insert $\mathcal{O}(n)$, erase $\mathcal{O}(n)$

**Stack via Array:**

- push, pop, top alle $\mathcal{O}(1)$

**Stack via LinkedList:**

- push, pop, top alle $\mathcal{O}(1)$

**Queue via LinkedList:**

- enqueue, dequeue beide $\mathcal{O}(1)$

# Sortieren durch Einfügen

**Gegeben:** eine Folge von ganzen Zahlen.

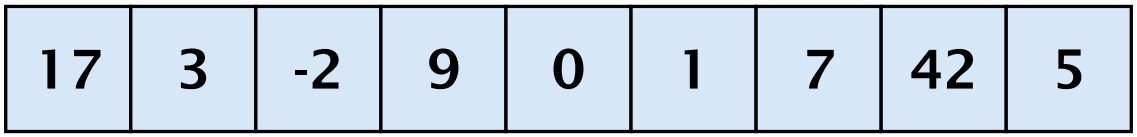
**Gesucht:** die zugehörige aufsteigend sortierte Folge.

**Idee:**

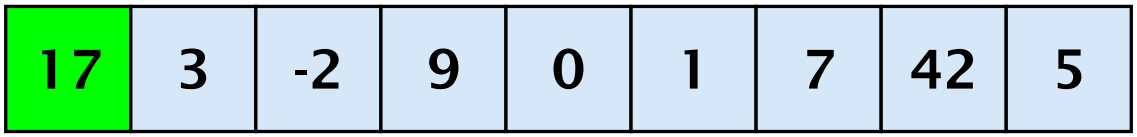
- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

$\Longrightarrow$ Sortieren durch Einfügen (InsertionSort)

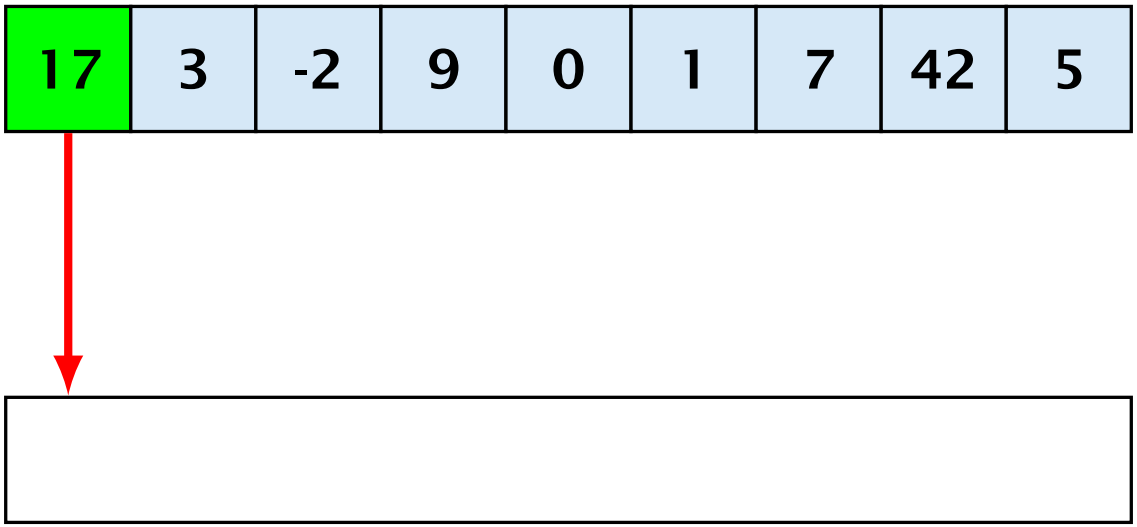
# Beispiel



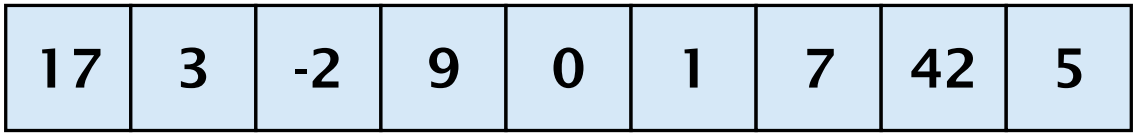
# Beispiel



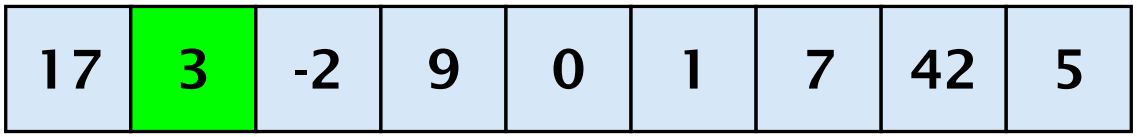
# Beispiel



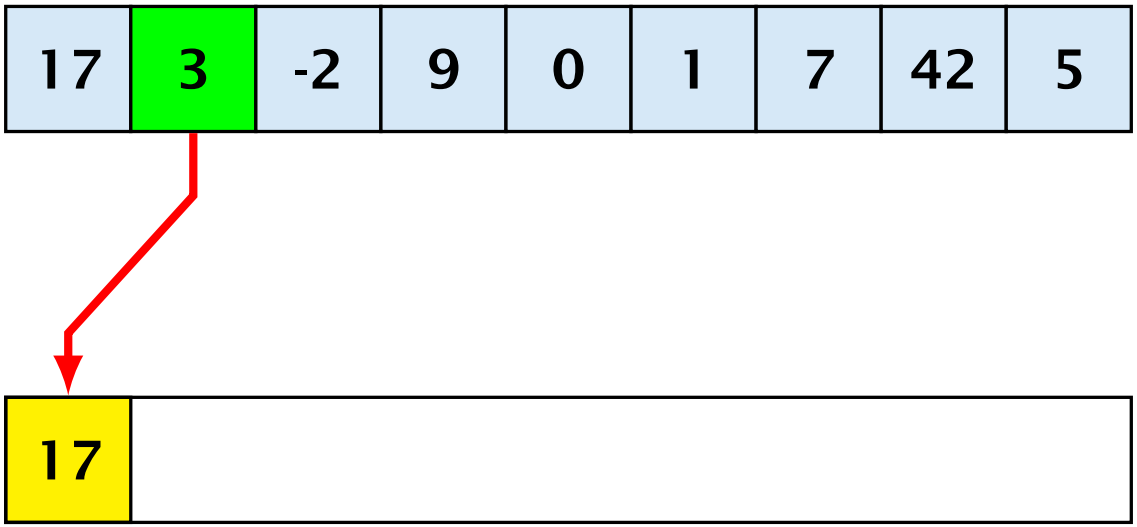
# Beispiel



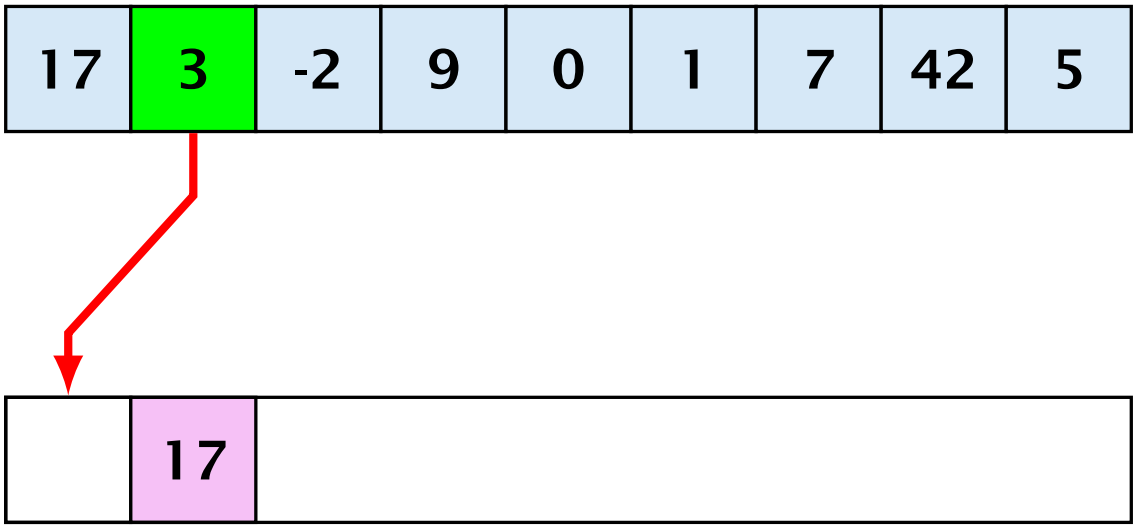
# Beispiel



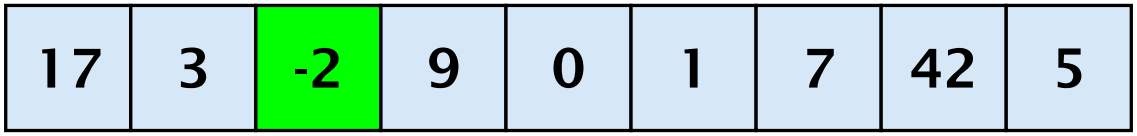
# Beispiel



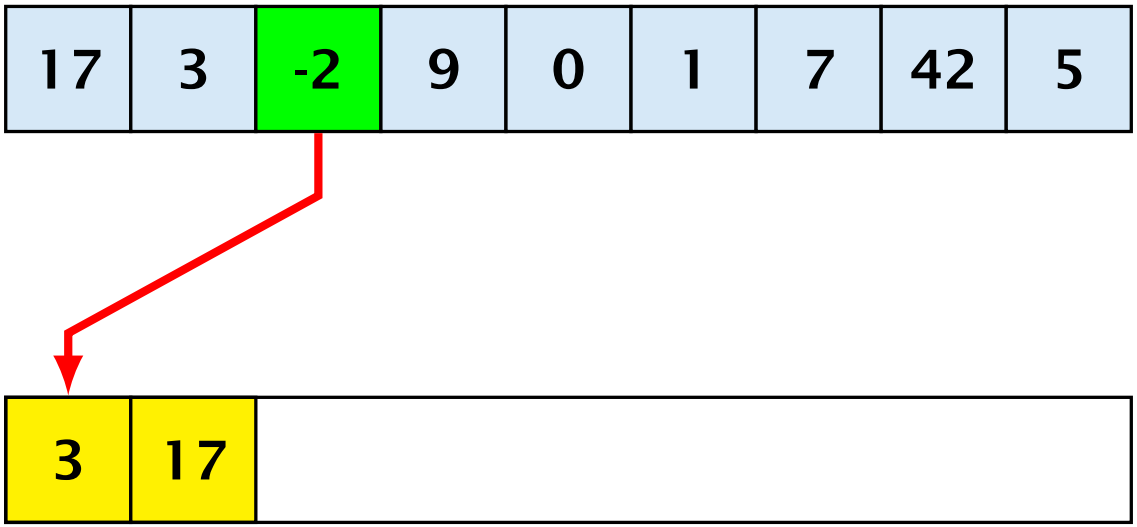
# Beispiel



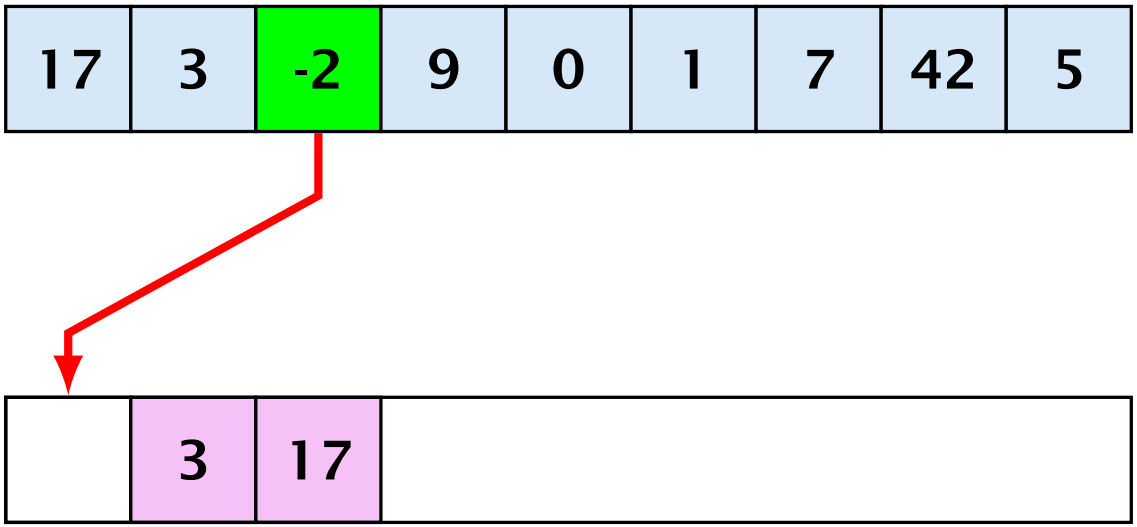
# Beispiel



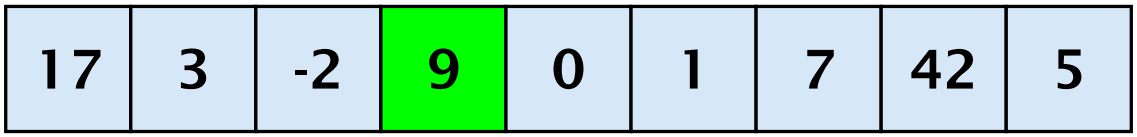
# Beispiel



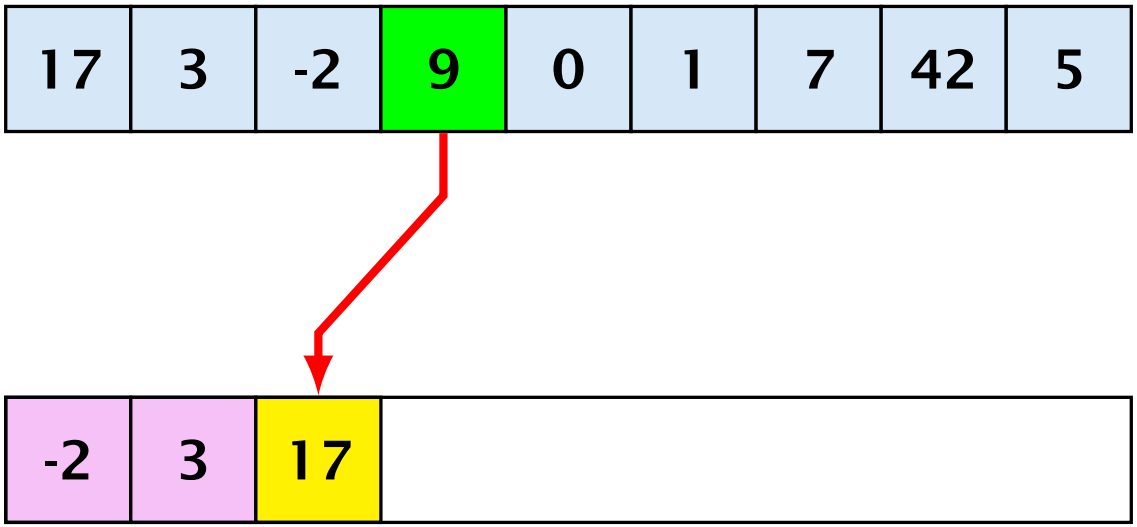
# Beispiel



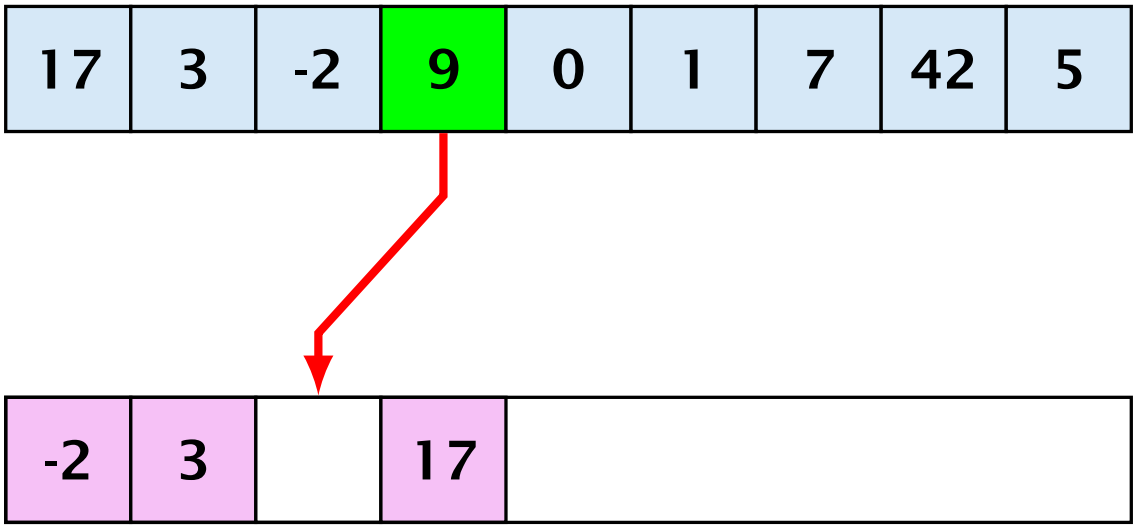
# Beispiel



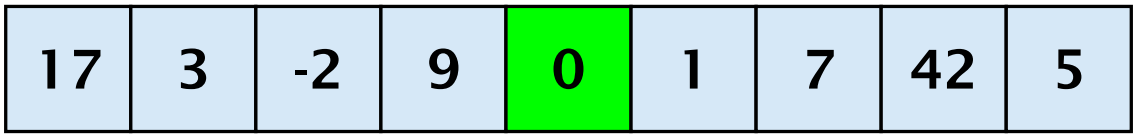
# Beispiel



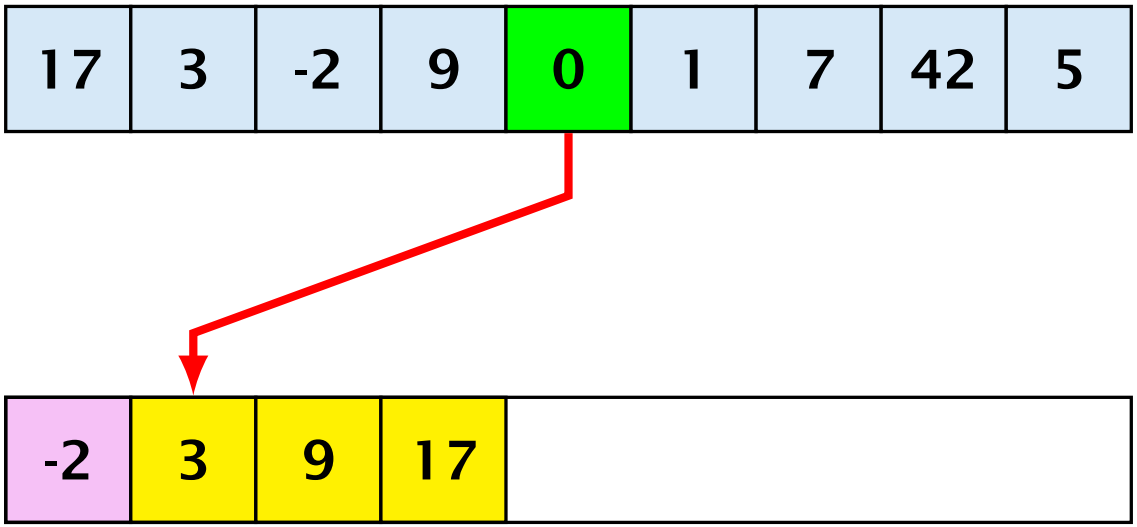
# Beispiel



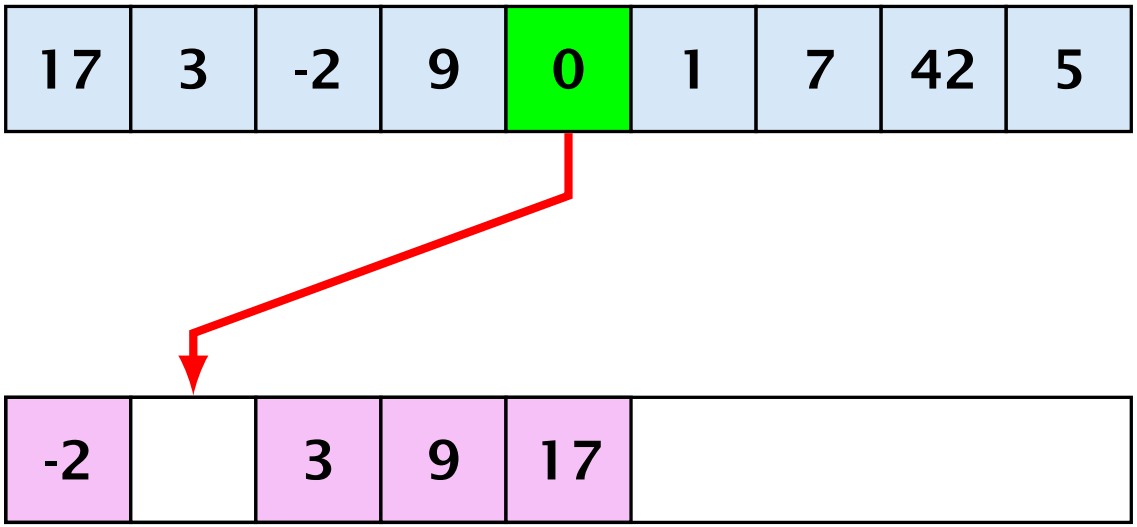
# Beispiel



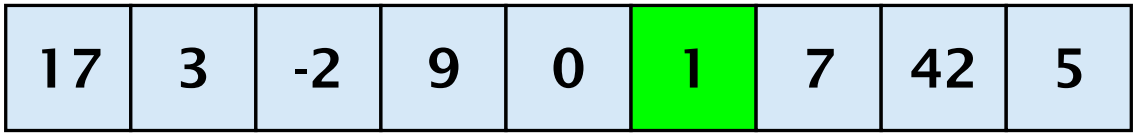
# Beispiel

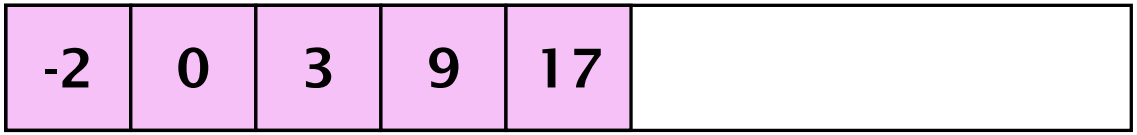


# Beispiel

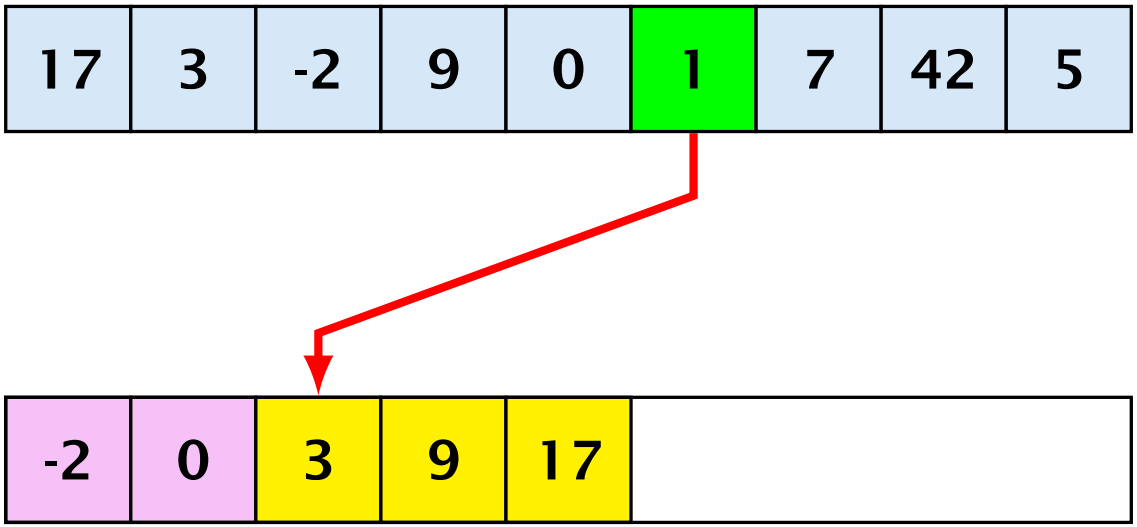


# Beispiel

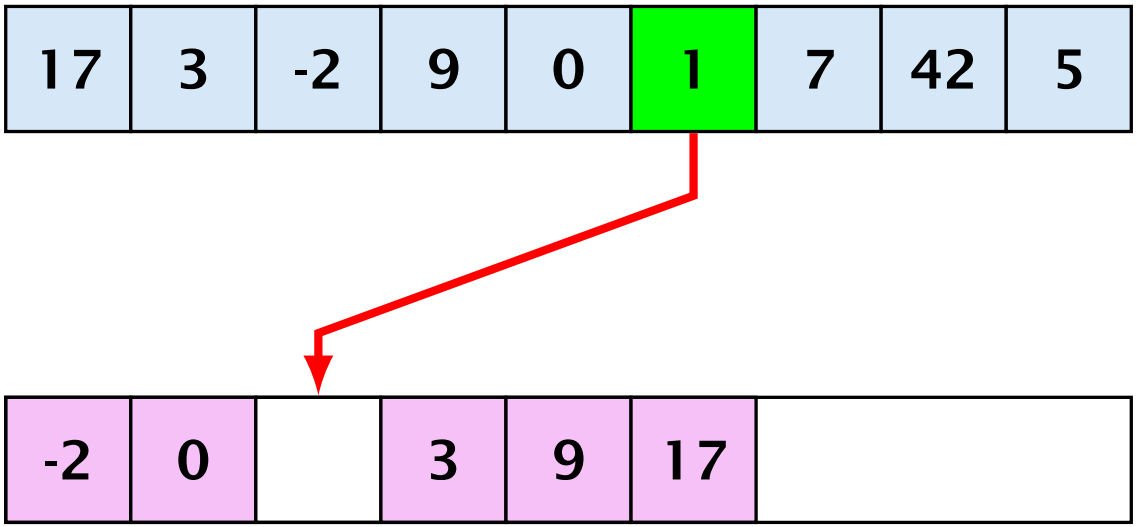


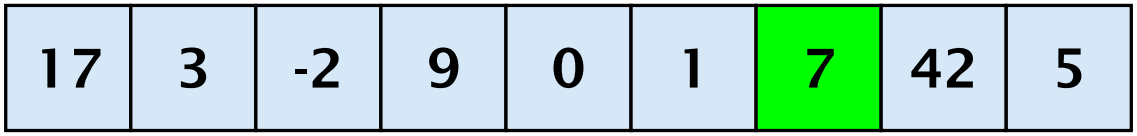
# Beispiel

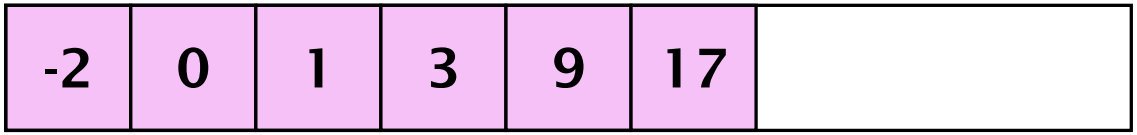


# Beispiel

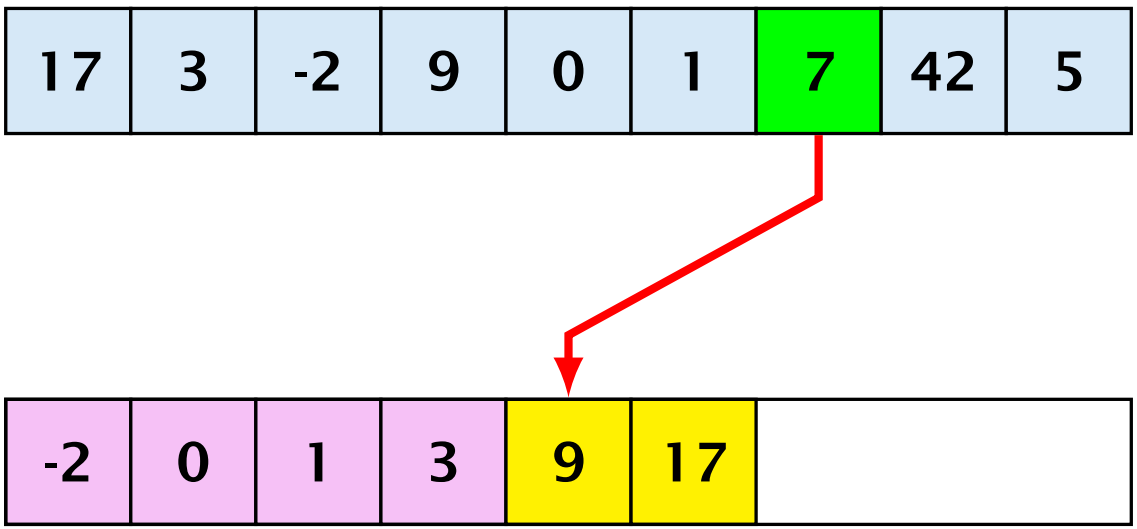


# Beispiel

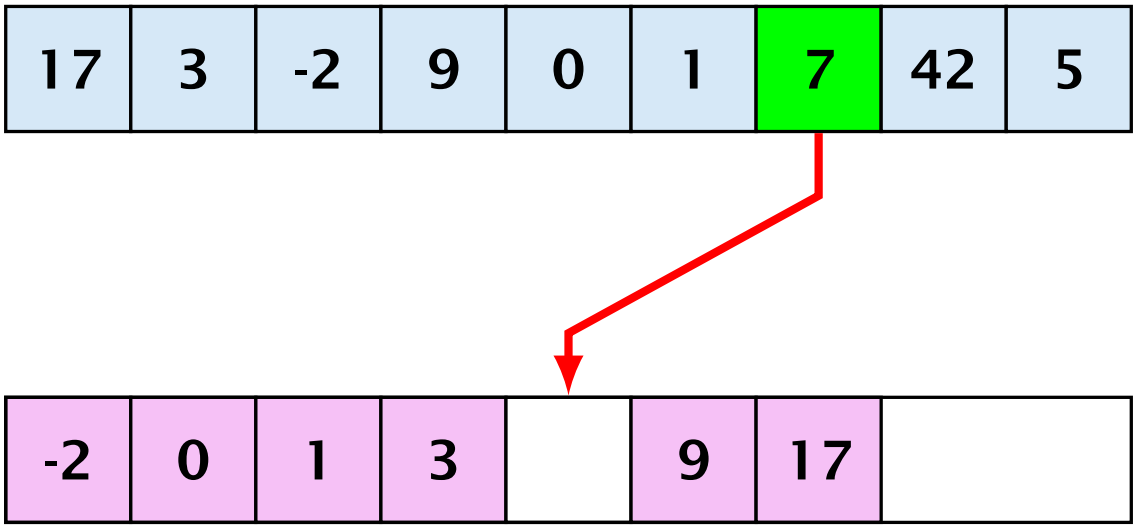


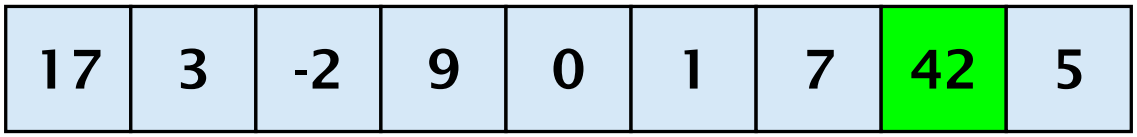
# Beispiel

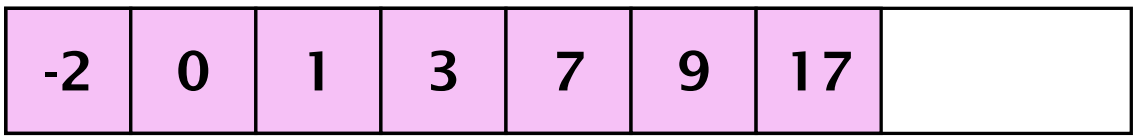


# Beispiel

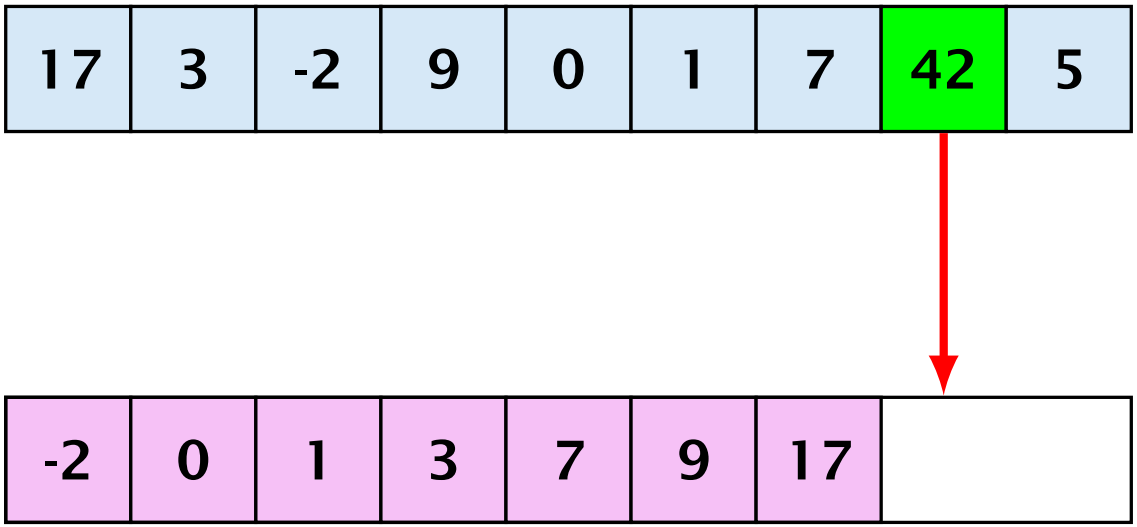


# Beispiel

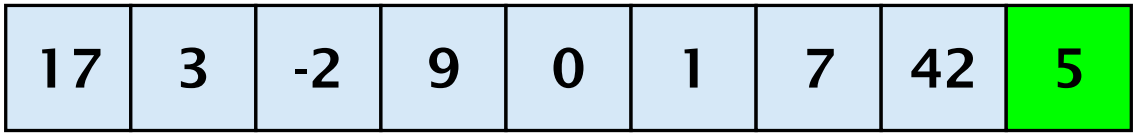


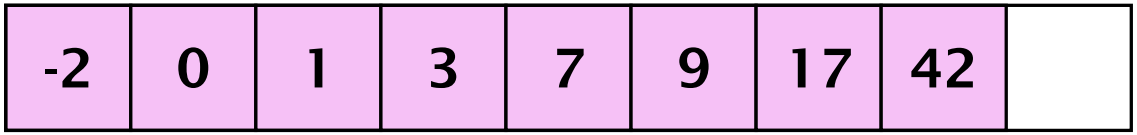



# Beispiel

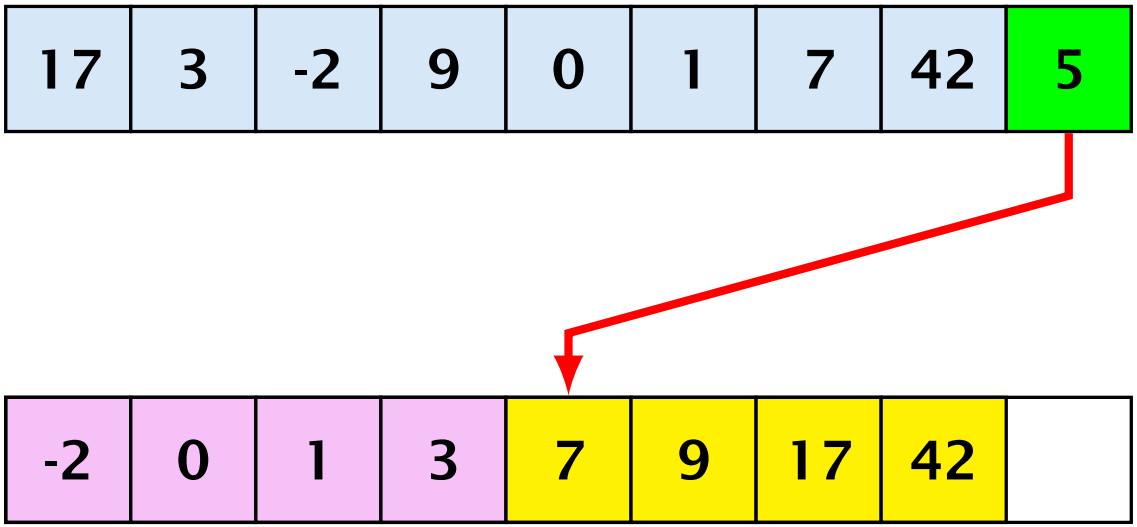


# Beispiel

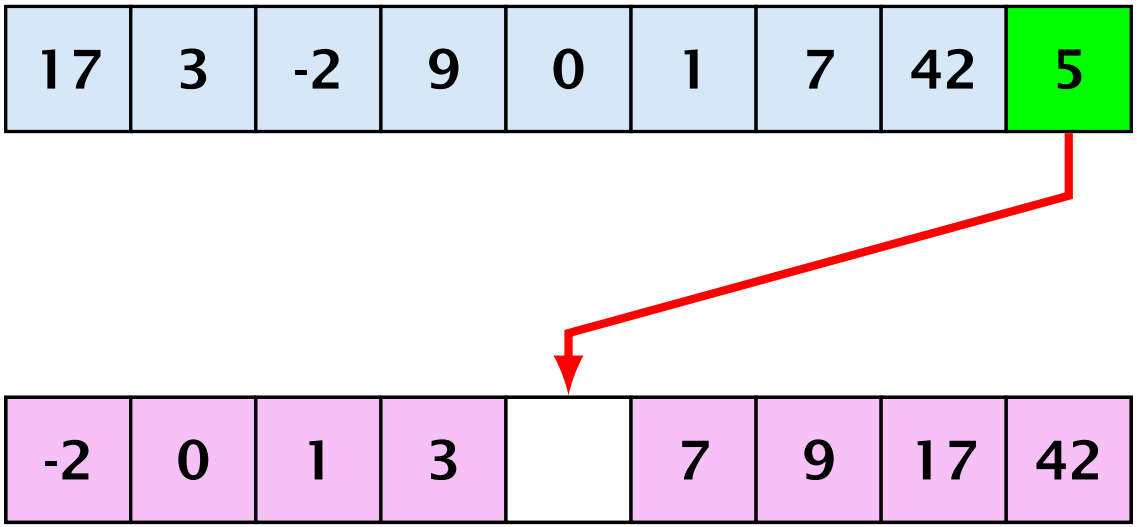


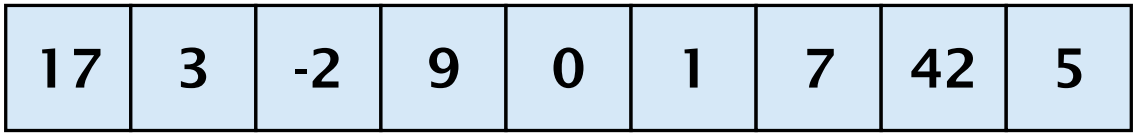
# Beispiel

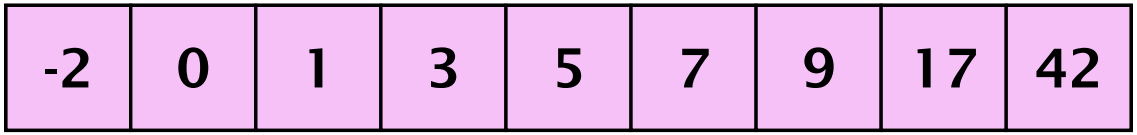


# Beispiel



# Beispiel



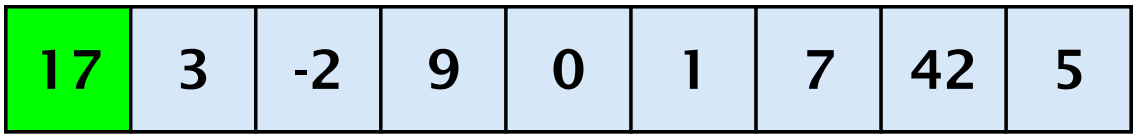
# Beispiel

**Wir brauchen kein zweites Array:**

| 17 | 3 | -2 | 9 | 0 | 1 | 7 | 42 | 5 |
|---|---|---|---|---|---|---|---|---|

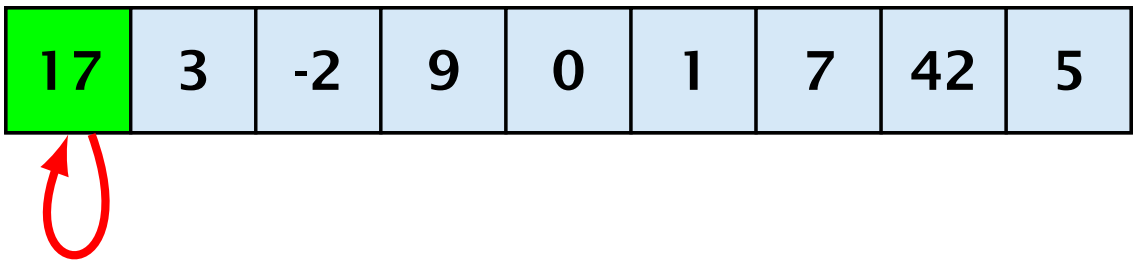
# Beispiel

**Wir brauchen kein zweites Array:**



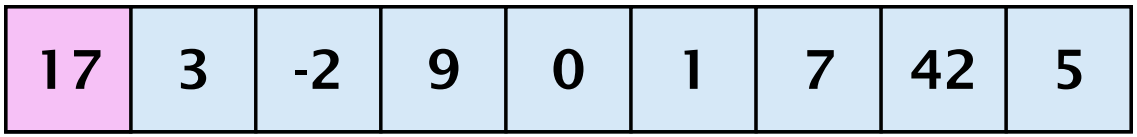
# Beispiel

**Wir brauchen kein zweites Array:**



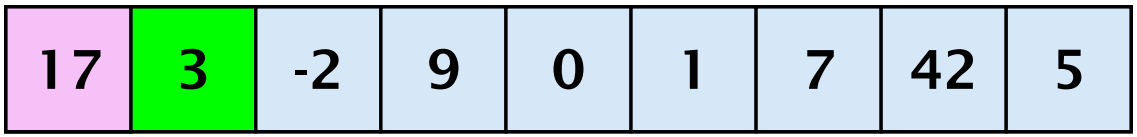
# Beispiel

**Wir brauchen kein zweites Array:**



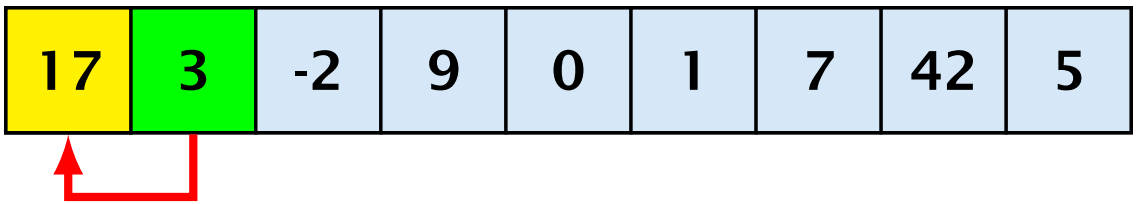
# Beispiel

**Wir brauchen kein zweites Array:**



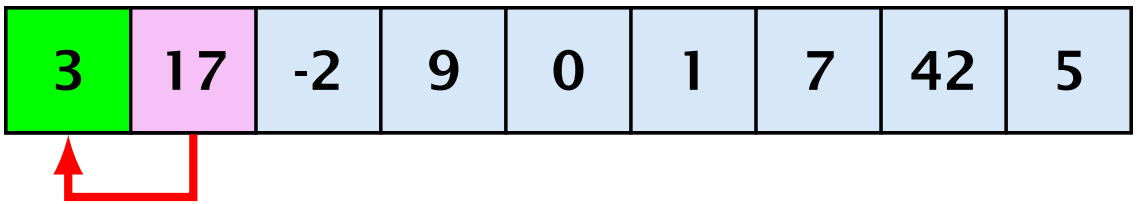
# Beispiel

**Wir brauchen kein zweites Array:**



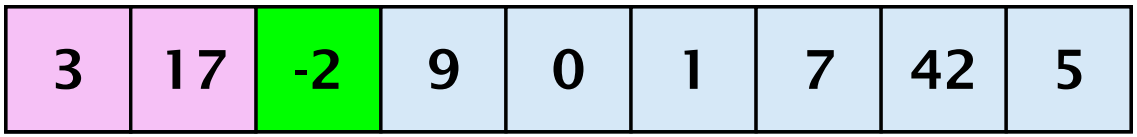
# Beispiel

**Wir brauchen kein zweites Array:**



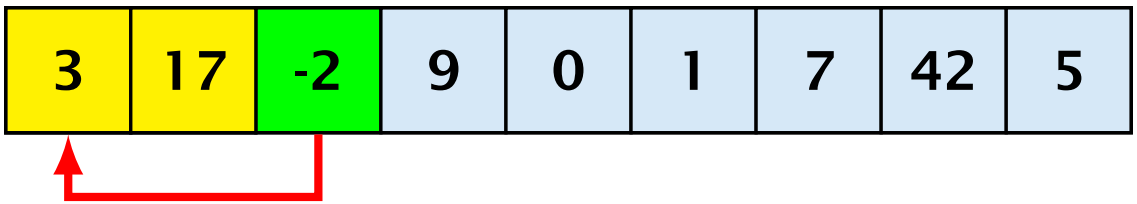
# Beispiel

**Wir brauchen kein zweites Array:**



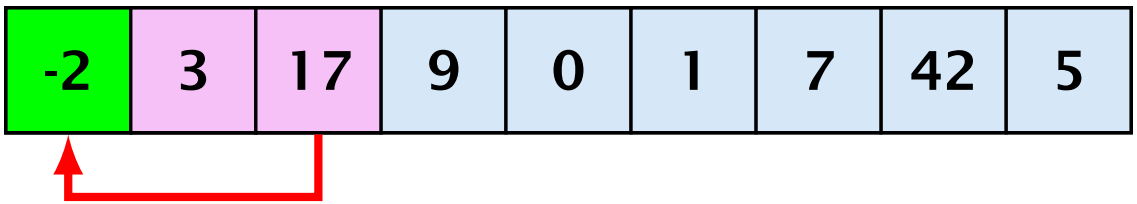
# Beispiel

**Wir brauchen kein zweites Array:**



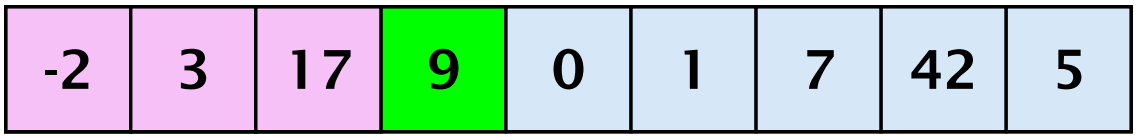
# Beispiel

**Wir brauchen kein zweites Array:**



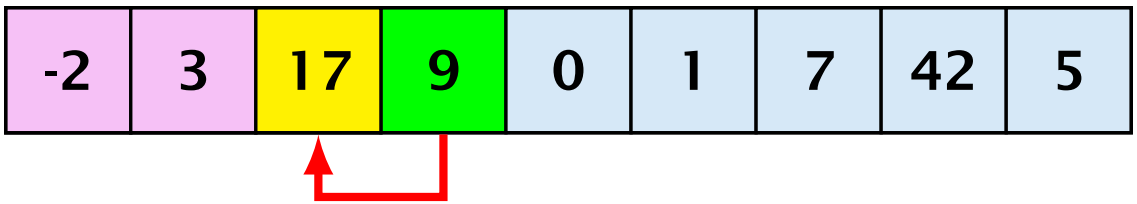
# Beispiel

**Wir brauchen kein zweites Array:**



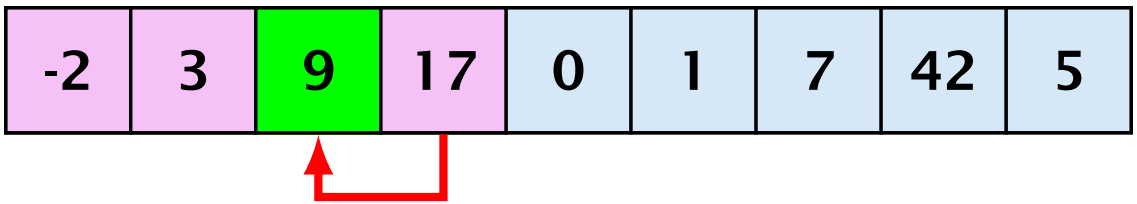
# Beispiel

**Wir brauchen kein zweites Array:**



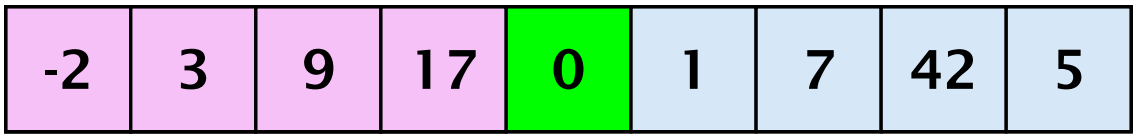
# Beispiel

**Wir brauchen kein zweites Array:**



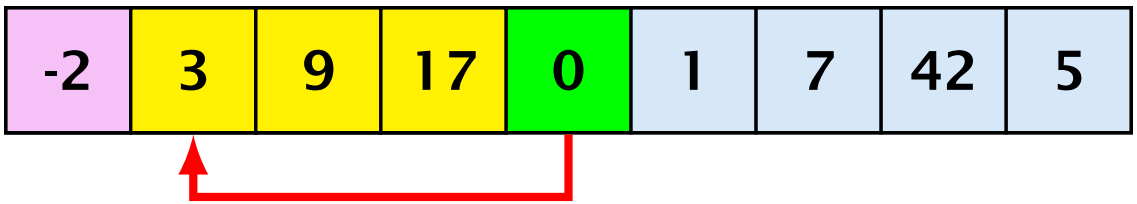
# Beispiel

**Wir brauchen kein zweites Array:**



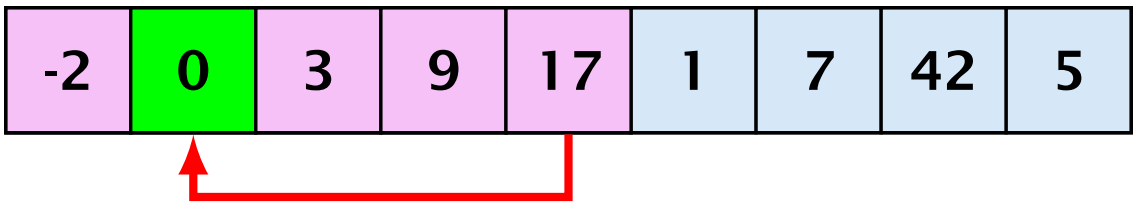
# Beispiel

**Wir brauchen kein zweites Array:**



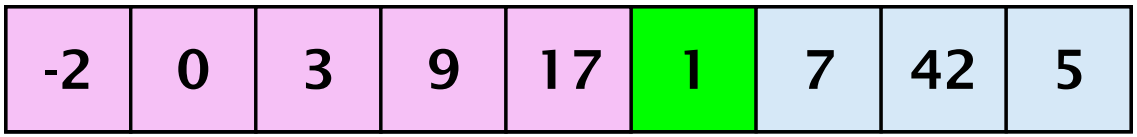
# Beispiel

**Wir brauchen kein zweites Array:**



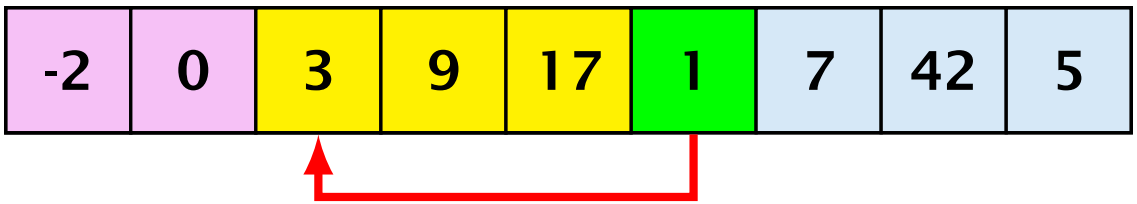
# Beispiel

**Wir brauchen kein zweites Array:**



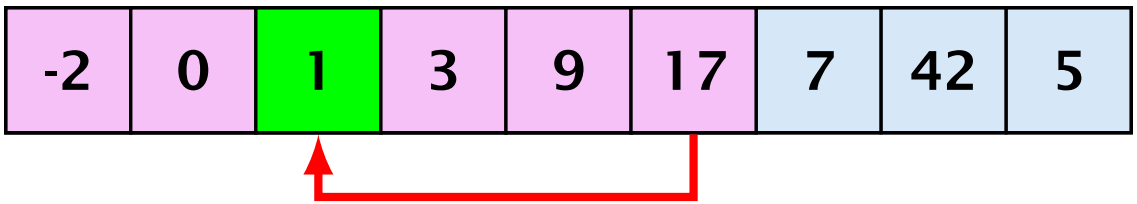
# Beispiel

**Wir brauchen kein zweites Array:**



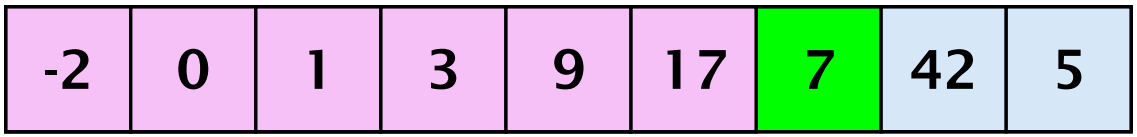
# Beispiel

**Wir brauchen kein zweites Array:**



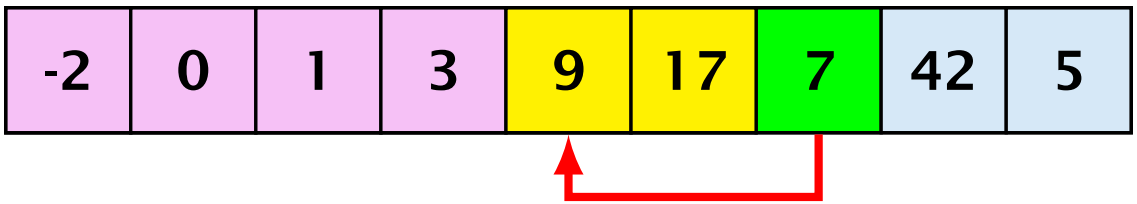
# Beispiel

**Wir brauchen kein zweites Array:**



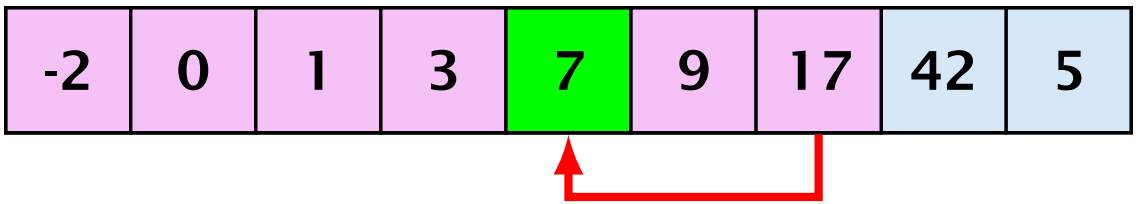
# Beispiel

**Wir brauchen kein zweites Array:**



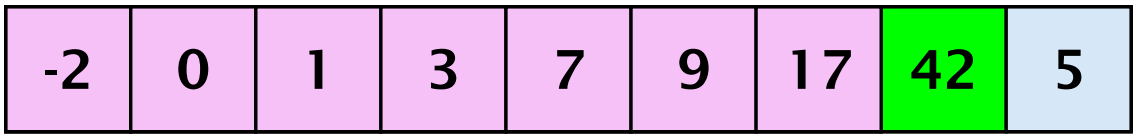
# Beispiel

**Wir brauchen kein zweites Array:**



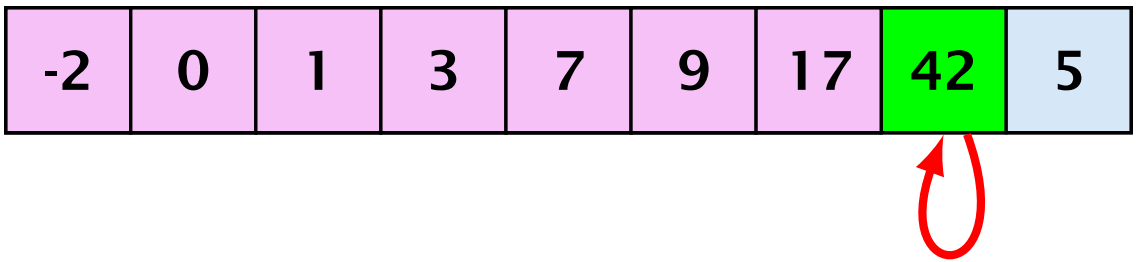
# Beispiel

**Wir brauchen kein zweites Array:**



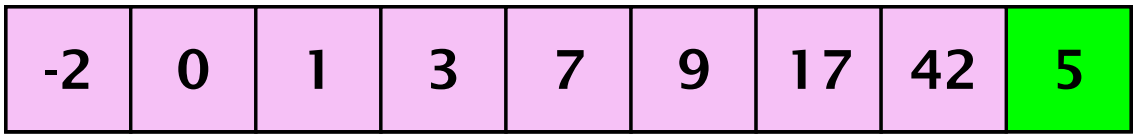
# Beispiel

**Wir brauchen kein zweites Array:**



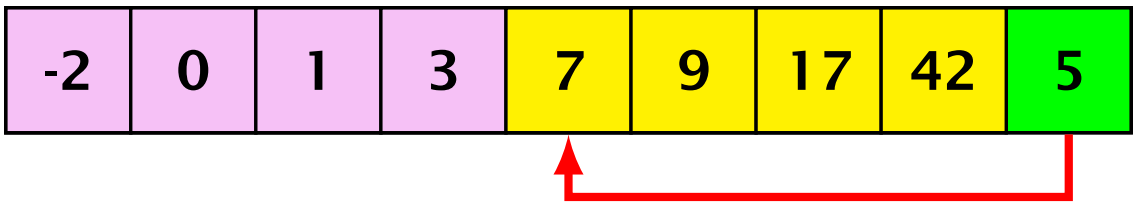
# Beispiel

**Wir brauchen kein zweites Array:**



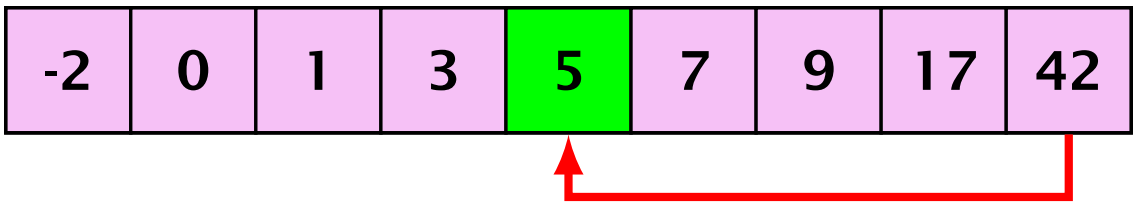
# Beispiel

**Wir brauchen kein zweites Array:**



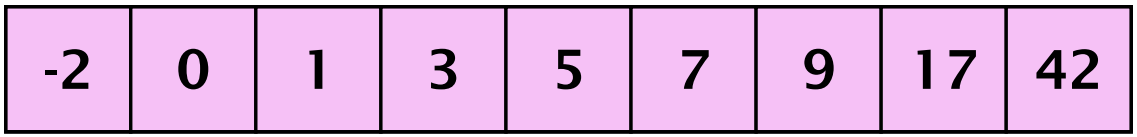
# Beispiel

**Wir brauchen kein zweites Array:**



# Beispiel

**Wir brauchen kein zweites Array:**



# Algorithmus: Insertion Sort

```
1  Input: Array A mit n Elementen
2  Output: Array A aufsteigend sortiert
3
4  InsertionSort(A)
5      j = 0;
6      while (j < n)
7          key = A[j];
8          i = j-1;
9          while (i >= 0 && A[i] > key)
10             A[i+1] = A[i];
11             i--;
12         A[i+1] = key;
13         j++;
```

InsertionSort

# Laufzeit InsertionSort

| *Kostenübersicht* | | |
|---|---|---|
| ***Zeile*** | ***Kosten*** | ***Anzahl*** |
| 5 | $\mathcal{O}(1)$ | $1$ |
| 6 | $\mathcal{O}(1)$ | $n+1$ |
| 7 | $\mathcal{O}(1)$ | $n-1$ |
| 8 | $\mathcal{O}(1)$ | $t_j$ |
| 9 | $\mathcal{O}(1)$ | $t_j-1$ |
| 10 | $\mathcal{O}(1)$ | $t_j-1$ |
| 11 | $\mathcal{O}(1)$ | $n$ |
| 12 | $\mathcal{O}(1)$ | $n$ |

```
1  Input: Array A mit Laenge n
2  Output: sortiertes Array
3
4  InsertionSort(A)
5    j = 0;
6    while (j < n)
7      key = A[j];
8      i = j-1;
9      while (i>=0 && A[i]>key)
10       A[i+1] = A[i];
11       i--;
12     A[i+1] = key;
13     j++;
```

$t_j$ bezeichnet Anzahl der Abfragen der while-Bedingung in Zeile 8 für Durchlauf $j$

# Laufzeit InsertionSort

$$\mathcal{O}(1) + (n+1)\mathcal{O}(1) + (n-1)\mathcal{O}(1) + \mathcal{O}(1)\sum_{j=0}^{n-1} t_j +$$

$$\mathcal{O}(1)\sum_{j=0}^{n-1}(t_j - 1) + \mathcal{O}(1)\sum_{j=0}^{n-1}(t_j - 1) + n\mathcal{O}(1) + n\mathcal{O}(1)$$

$$= \mathcal{O}(1)n + \mathcal{O}(1)\sum_{j=0}^{n-1} t_j$$

$$= \mathcal{O}\left(n + \sum_{j=0}^{n-1} t_j\right)$$

Die Laufzeit hängt stark vom Input ab.

# Laufzeit InsertionSort

**Best-case:**

Wenn das Array sortiert wird ist, ist $t_j = 1$.

$\Longrightarrow$ Laufzeit: $\mathcal{O}(n)$.

**Worst-case:**

Wenn das Array absteigend sortiert ist, ist $t_j = j + 1$.

$\Longrightarrow$ Laufzeit: $\mathcal{O}(n^2)$.

**Beobachtung:**

Wenn ein Element höchstens $h$ Positionen von seiner Zielposition entfernt ist, dann ist $t_j \leq h + 1$.

$\Longrightarrow$ Laufzeit: $\mathcal{O}(hn)$.

**Gegeben:** Array `A` ganzer Zahlen; Element `x`

**Gesucht:** Wo kommt `x` in `A` vor?

**Naives Vorgehen:**

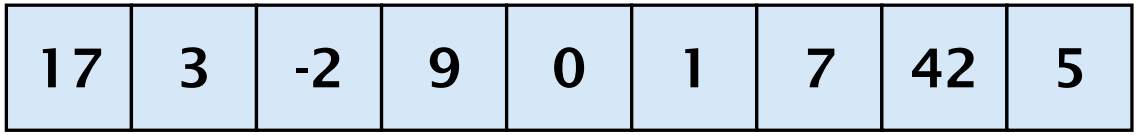
- Vergleiche `x` der Reihe nach mit `A[0]`, `A[1]`, usw.
- Finden wir `i` mit `A[i] == x`, geben wir `i` aus.
- Andernfalls geben wir `-1` aus: „Element nicht gefunden“!

# Naives Suchen

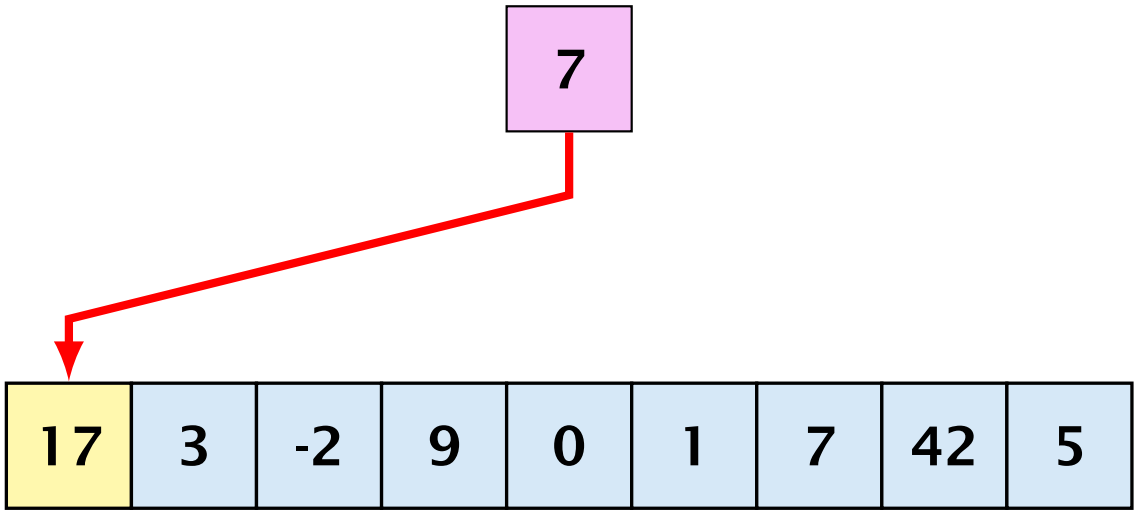
```
1  Input:  Array A mit Laenge n; Element x
2  Output: i mit A[i] == x falls existent
3          sonst -1
4
5  find(A,x)
6      i = 0;
7      while (i < n && A[i] != x)
8          i++;
9      if (i == n)
10         return -1;
11     else
12         return i;
```

Naives Suchen

# Beispiel

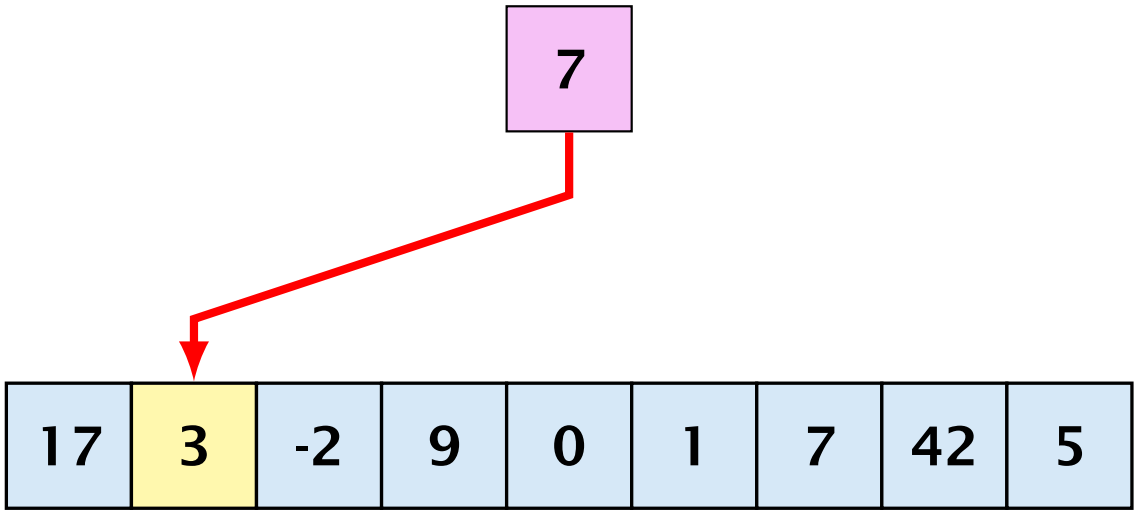


# Beispiel



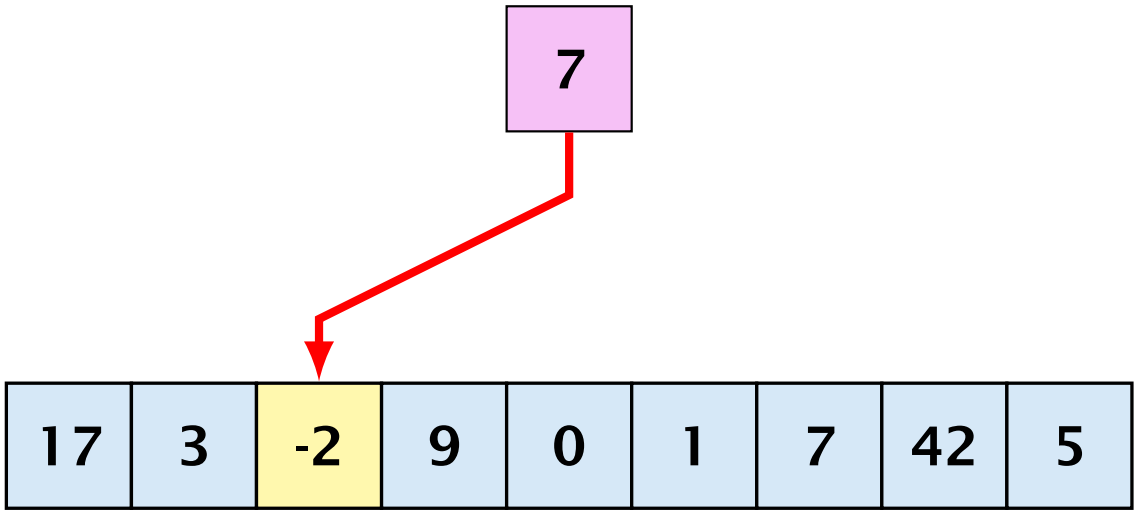

no

# Beispiel



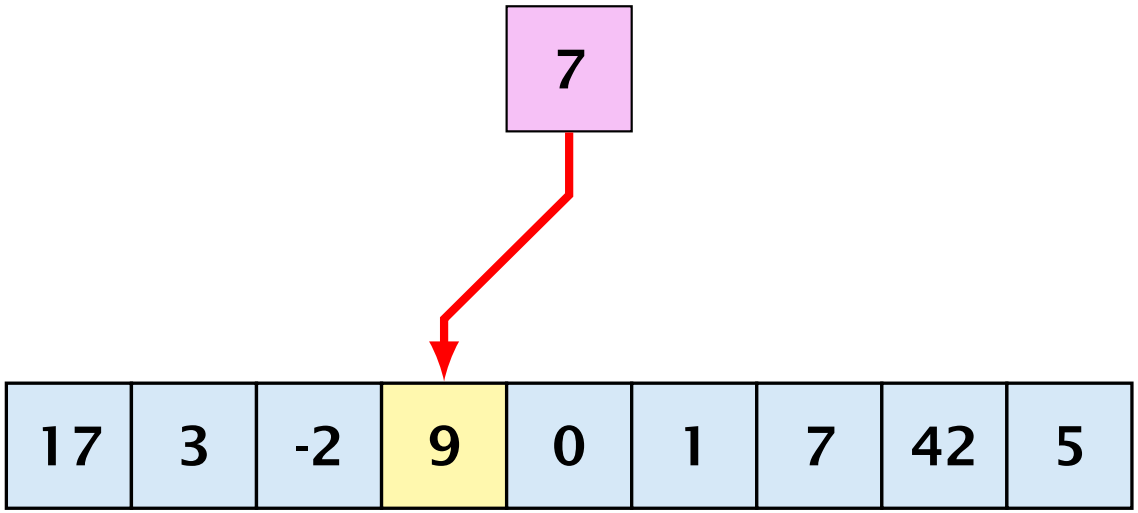

no

# Beispiel



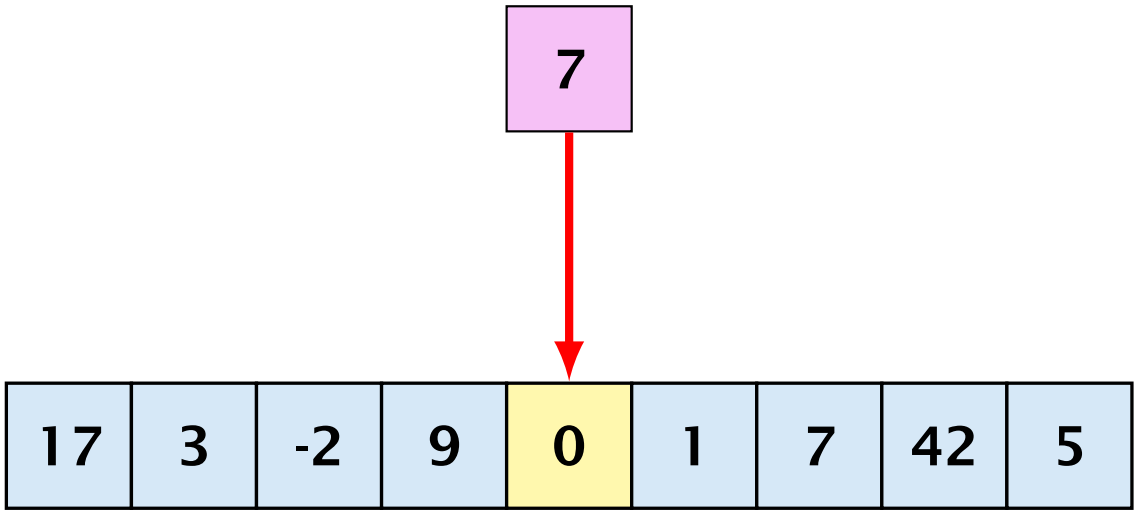

no

# Beispiel



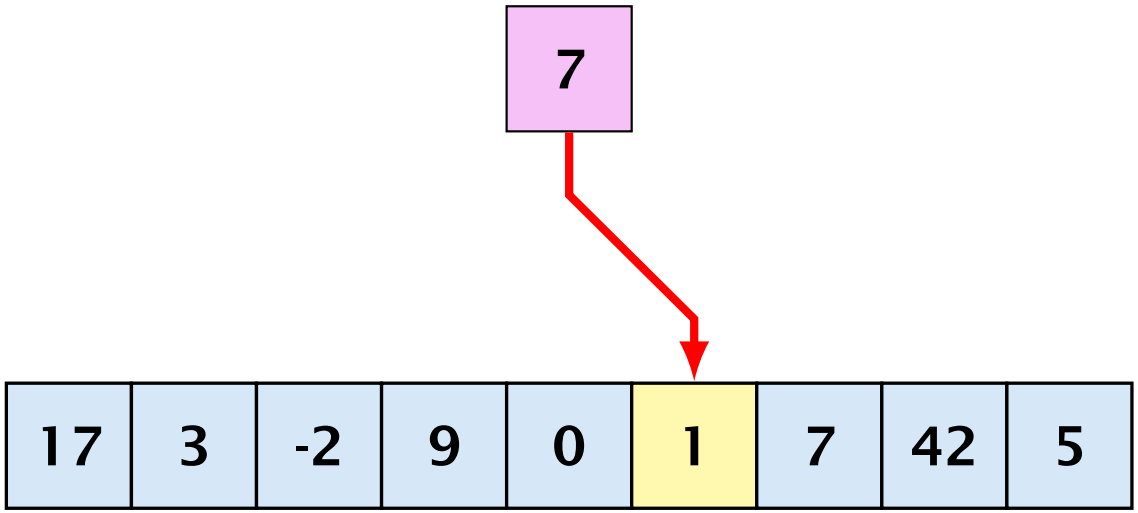

no

# Beispiel



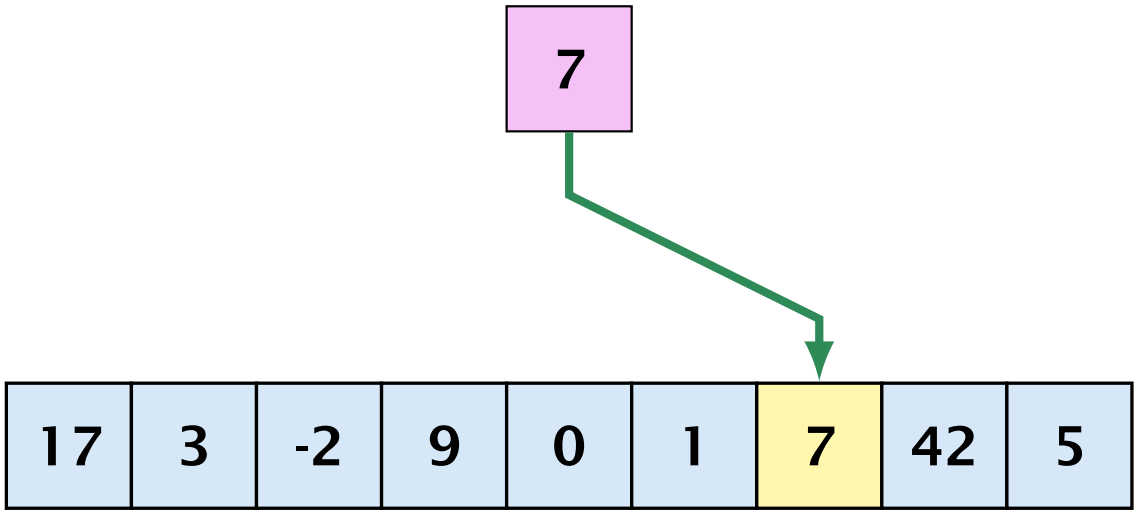

no

# Beispiel




no

# Beispiel




yes

# Laufzeit Naives Suchen

**Best-case:**

Wenn $x$ an Position $0$.

$\Longrightarrow$ Laufzeit: $\mathcal{O}(1)$.

**Worst-case:**

Wenn $x$ nicht vorkommt.

$\Longrightarrow$ Laufzeit: $\mathcal{O}(n)$.

**...geht das besser?**

# Binäre Suche

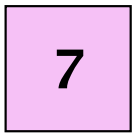
**Annahme:** Input ist sortiert.

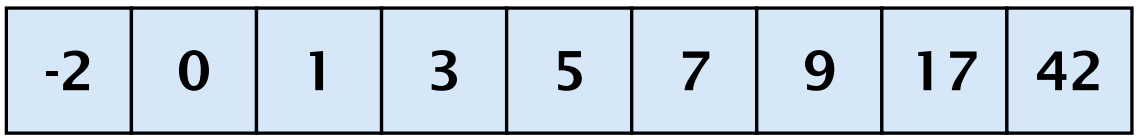
**Idee:**

- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

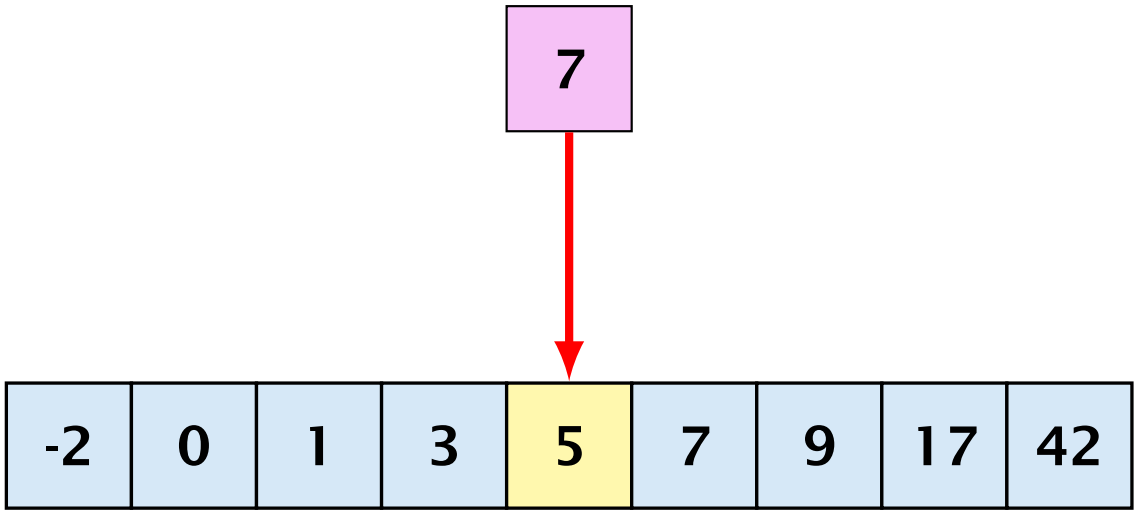
# Beispiel




- wir benötigen nur drei Vergleiche

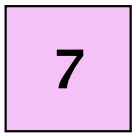
# Beispiel




no

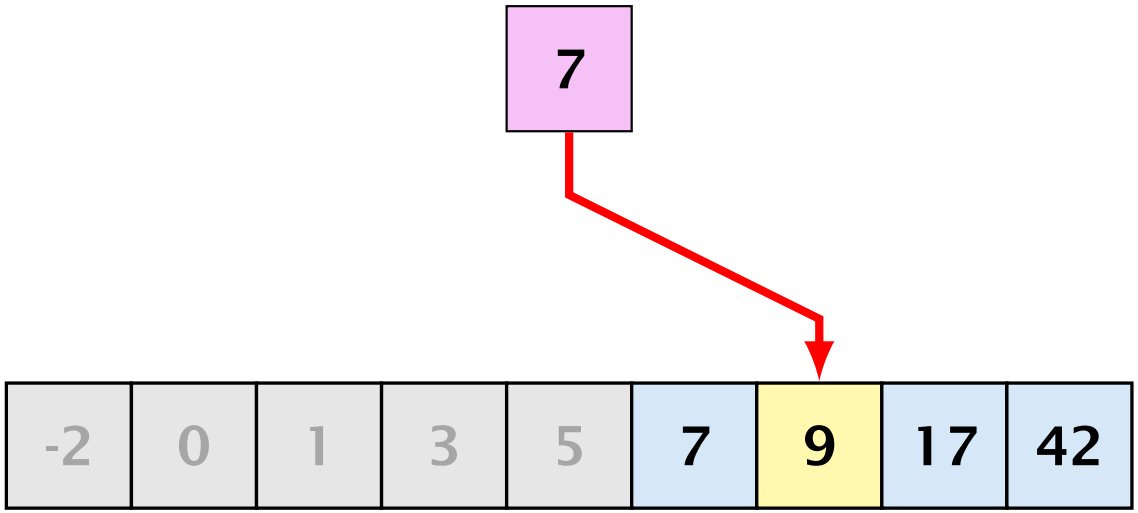
- wir benötigen nur drei Vergleiche

# Beispiel



- wir benötigen nur drei Vergleiche

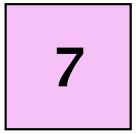
# Beispiel




no

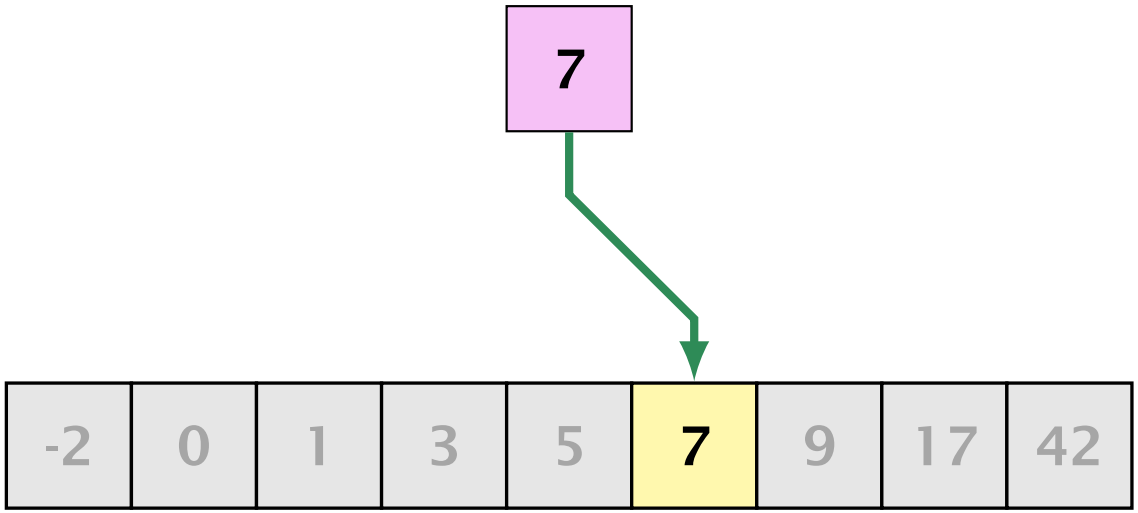
- wir benötigen nur drei Vergleiche

# Beispiel



- wir benötigen nur drei Vergleiche

# Beispiel




yes

- wir benötigen nur drei Vergleiche

# Implementierung

```
1  Input:  sortiertes Array A; Element x;
2          linker Index nl; rechter Index nr; nl<=nr
3  Output: Index i; nl ≤ i ≤ nr mit A[i]==x falls existent
4          sonst -1
5  find(A, x, nl, nr) // Inputlaenge ist n=nr-nl+1
6      t = (nl + nr) / 2;
7      if (A[t] == x)
8          return t;
9      if (nl == nr)
10         return -1;
11     if (x > A[t])
12         return find(A, x, t+1, nr);
13     if (nl < t)
14         return find(A, x, nl, t-1);
15     return -1;
```

# Laufzeit Binäre Suche

**Laufzeit:**

$$T(n) = \begin{cases} \mathcal{O}(1) & \texttt{A[t] == x} \\ \mathcal{O}(1) & \texttt{nl == nr} \\ \mathcal{O}(1) + T(nr - t) & \texttt{x > A[t]} \\ \mathcal{O}(1) + T(t - nl) & \texttt{nl < t} \end{cases}$$

**oder**

$$T(n) \leq \begin{cases} \mathcal{O}(1) & n = 1 \\ \mathcal{O}(1) + T(\lfloor n/2 \rfloor) & \text{sonst} \end{cases}$$

# Laufzeit Binäre Suche

**Lösen der Rekursionsgleichung:**

$$T(n) \leq \begin{cases} \mathcal{O}(1) & n = 1 \\ \mathcal{O}(1) + T(\lfloor n/2 \rfloor) & \text{sonst} \end{cases}$$

Üblicherweise nur für $n = 2^k$; z.B. durch vollständige Induktion.

# Rekurrenzen

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

# Rekurrenzen

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

# Methoden

1. **Raten+Induktion**
   Man rät die richtige Lösung und beweist die Korrektheit mittels vollständiger Induktion. Man benötigt Erfahrung um richtig zu raten...
2. **Mastertheorem**
   Für die meisten Rekurrenzen gibt es ein allgemeines Theorem, dass die asymptotisch korrekte Laufzeit für die jeweilige Rekurrenz angibt.

# Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

# Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**für $n = 2^k$:**

Für diesen Fall können wir stattdessen die folgende Rekursionsgleichung betrachten:

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

# Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**für $n = 2^k$:**

Für diesen Fall können wir stattdessen die folgende Rekursionsgleichung betrachten:

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Man rät die richtige Lösung und beweist, dass durch Einsetzen, dass diese Lösung korrekt ist.

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$:

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \\ &\leq a \log n + b \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:** $T(n) \leq a \log n + b$.

**Beweis.** (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für $n$:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \\ &\leq a \log n + b \end{aligned}$$

  Gilt falls $a \geq c_1$.

# Mastertheorem

**Lemma 3**

*Seien $a \geq 1, b \geq 1$ und $\epsilon > 0$ Konstanten. Betrachte die Rekurrenz*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \ .$$

*1. Fall:*

*Falls $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ gilt $T(n) = \Theta(n^{\log_b a})$.*

*2. Fall:*

*Falls $f(n) = \Theta(n^{\log_b(a)} \log^k n)$ gilt $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$, $k \geq 0$.*

# Mastertheorem

Wir beweisen das Mastertheorem für den Fall $n = b^\ell$, und nehmen an, dass der nichtrekursive Fall für Problemgröße $1$ Kosten $1$ verursacht.

# Der Rekursionsbaum

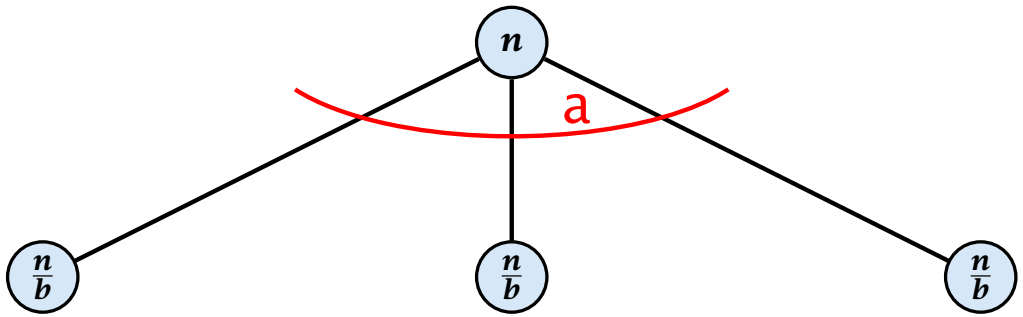
Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

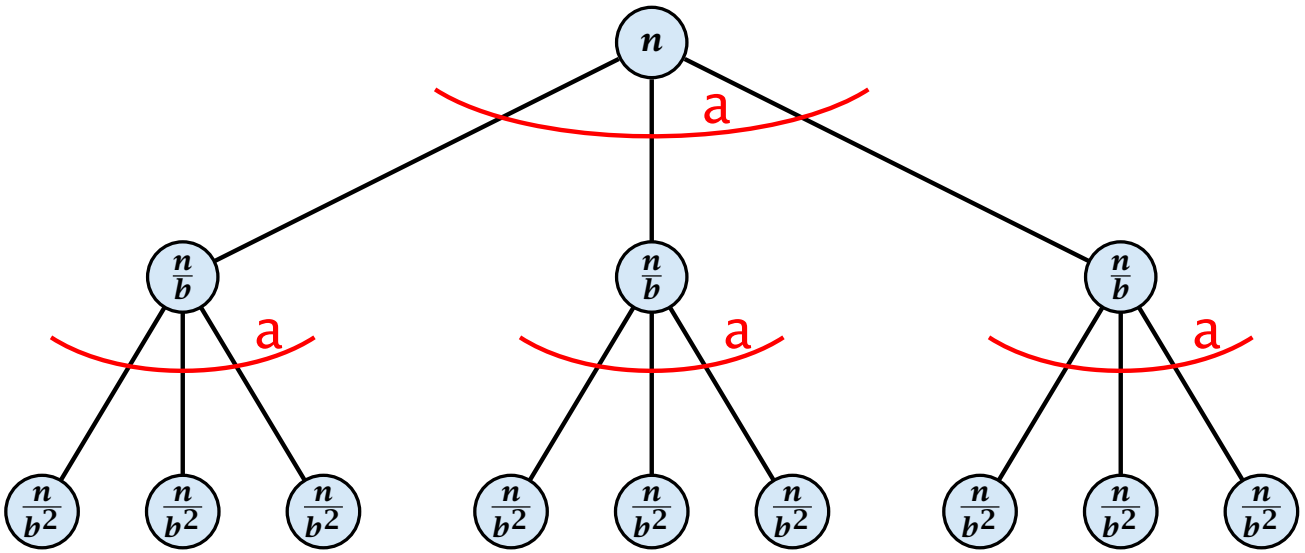
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



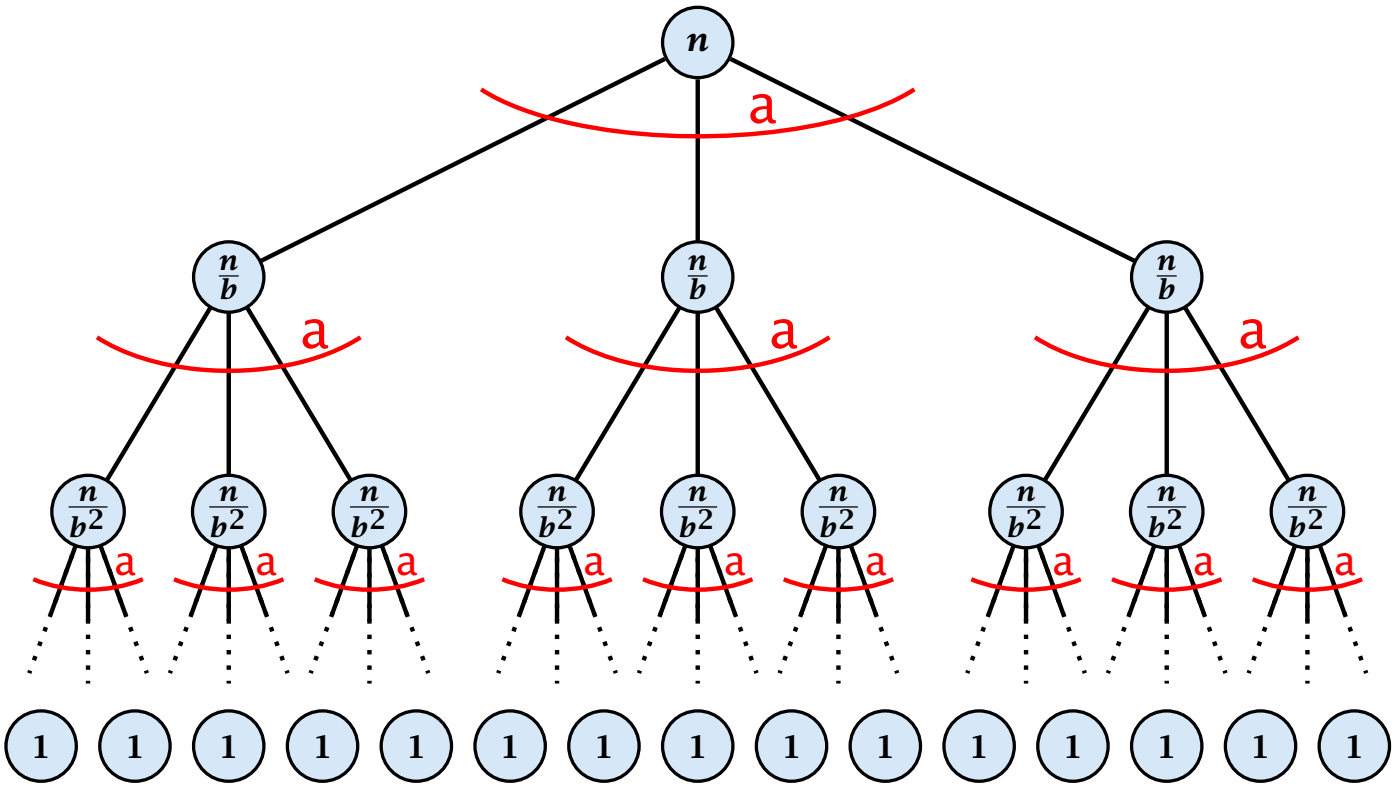
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



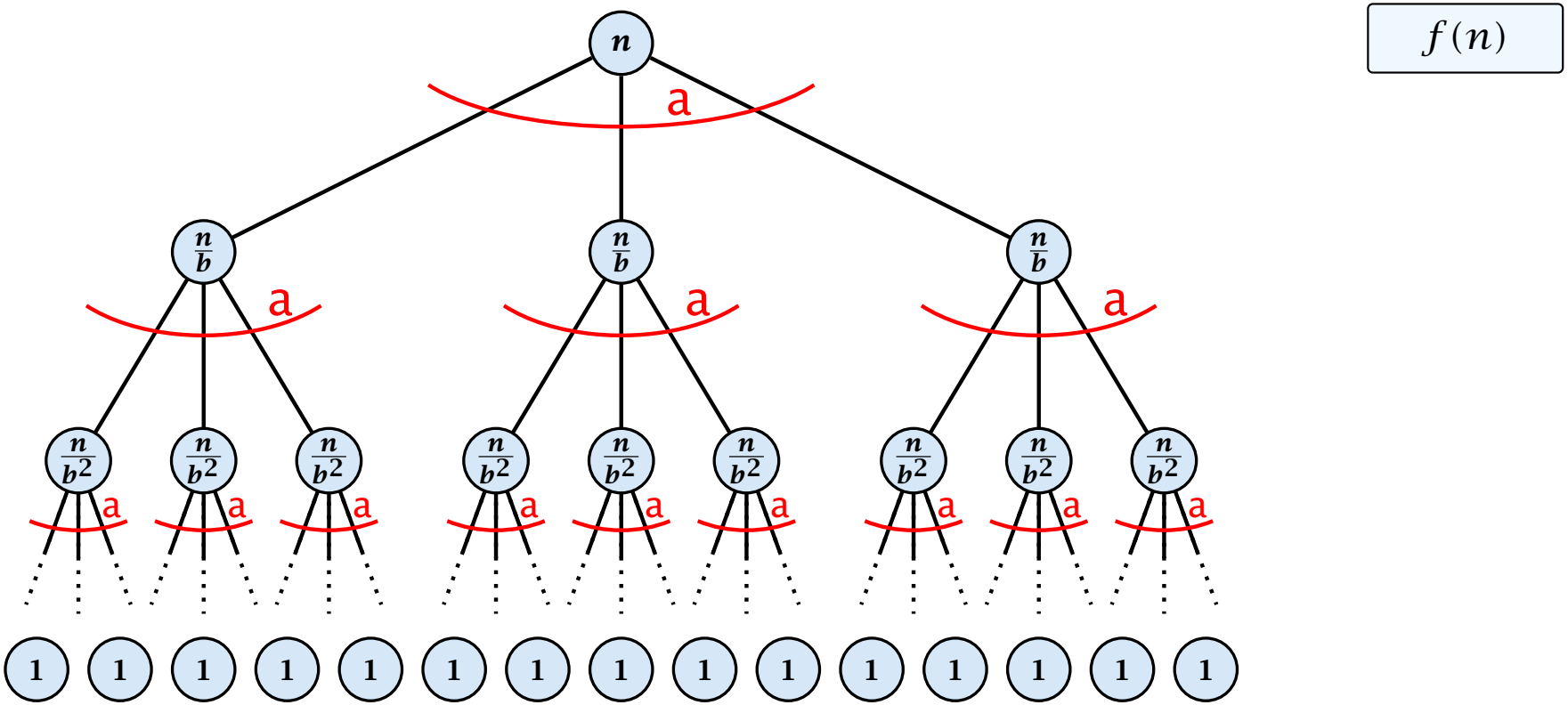
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



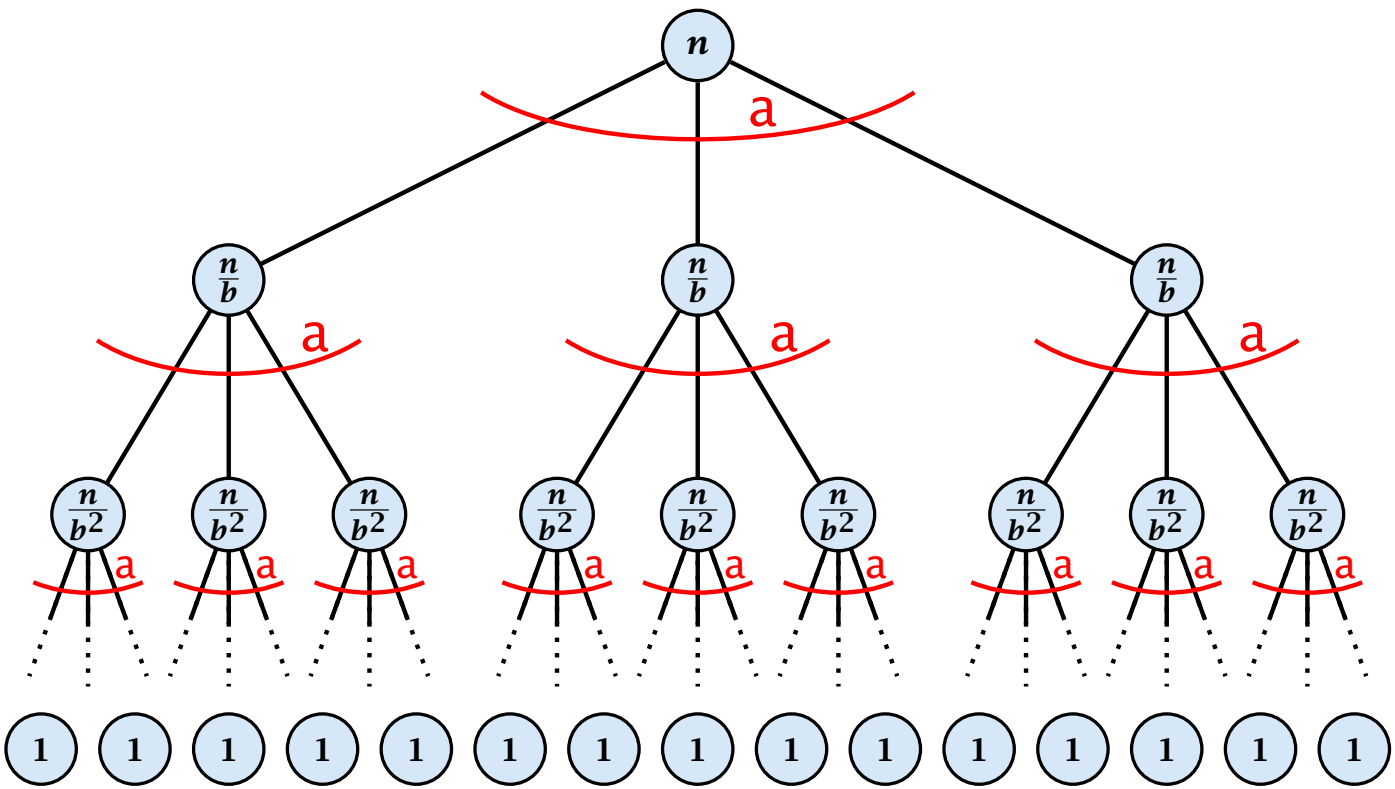
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



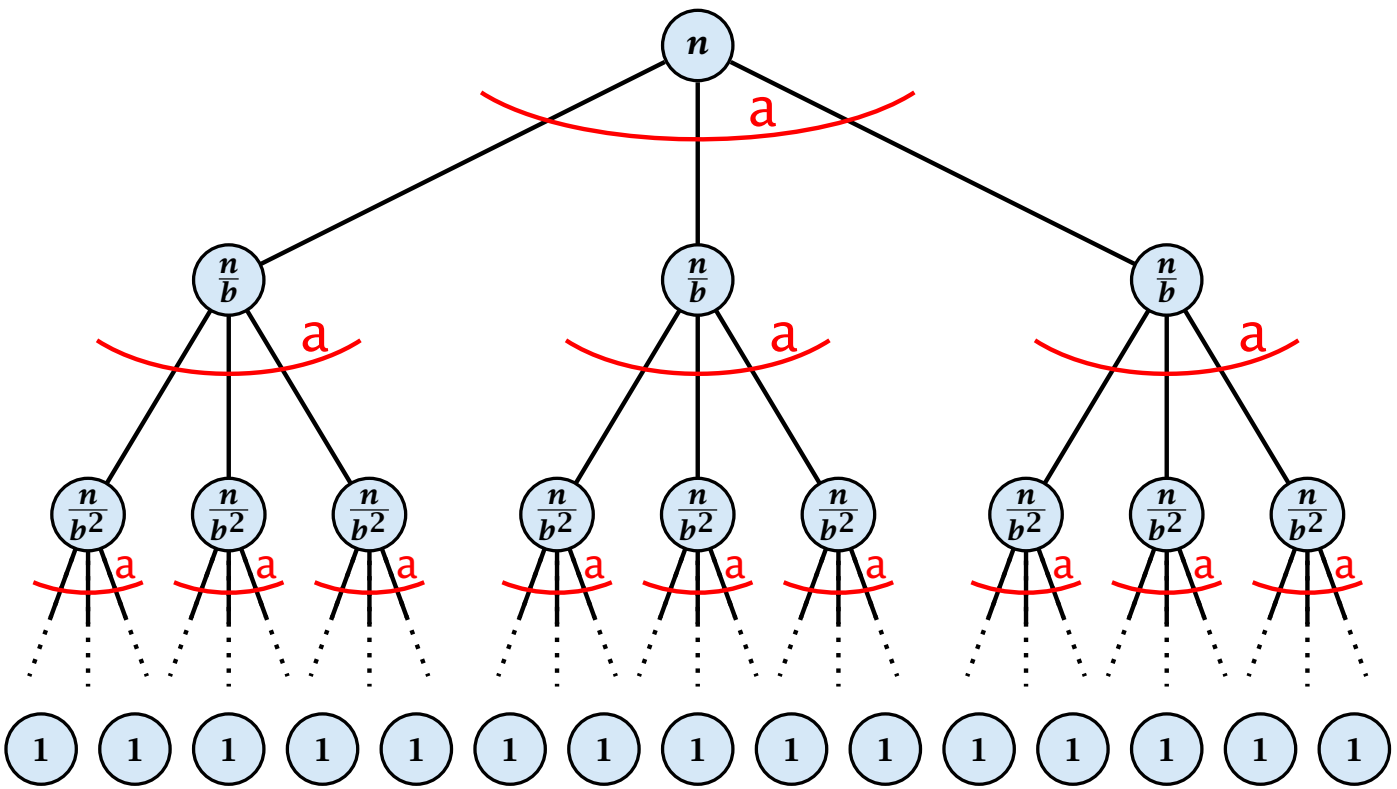
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



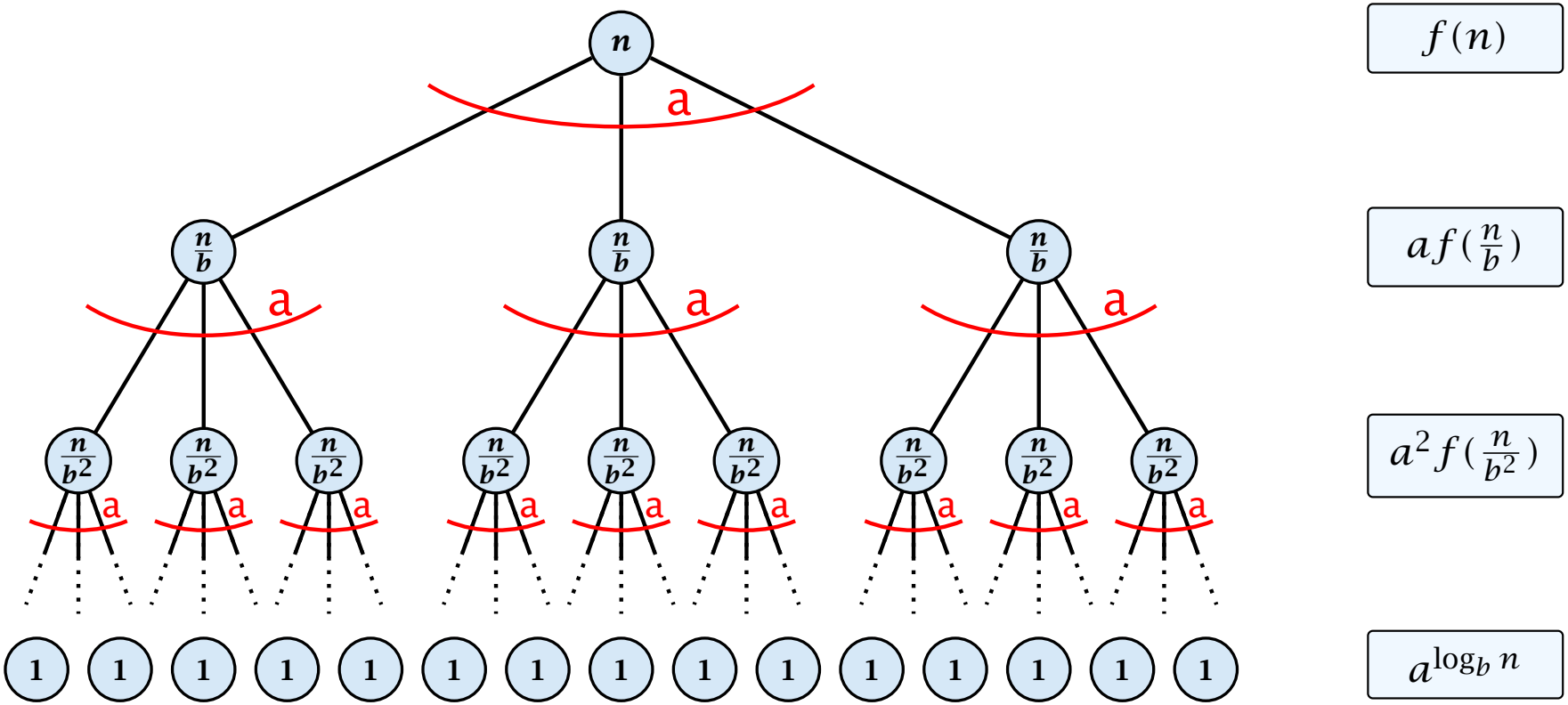

$f(n)$

$af(\frac{n}{b})$

$a^2 f(\frac{n}{b^2})$

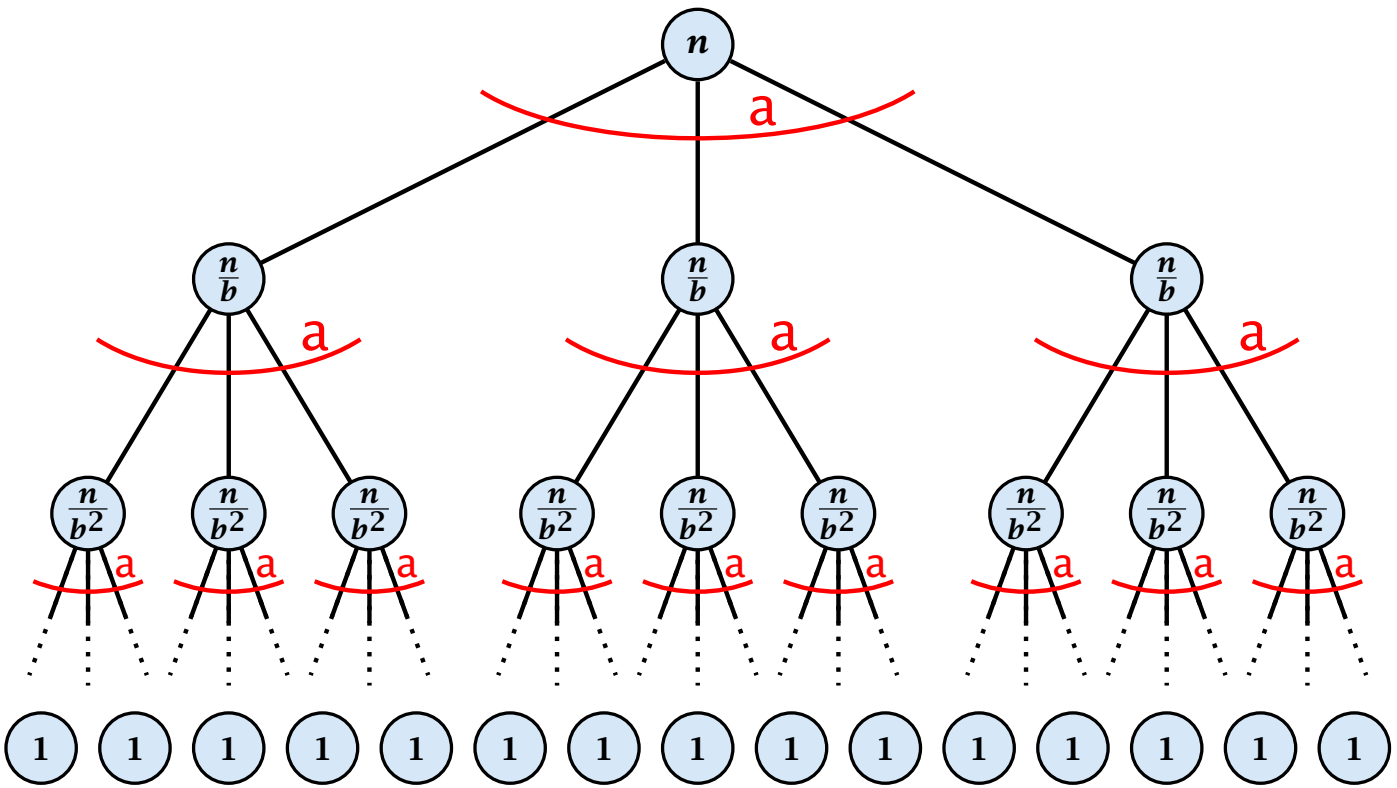
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:




$f(n)$

$af(\frac{n}{b})$

$a^2 f(\frac{n}{b^2})$

$a^{\log_b n}$

$=$

$n^{\log_b a}$

# Mastertheorem

Das heißt unsere Kosten sind

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

**Fall 1.** Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

**Fall 1.** Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a}$$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

**Fall 1.** Sei $f(n) \leq c n^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

**Fall 1.** Sei $f(n) \leq c n^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

$$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^\epsilon\right)^i$$

$$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^\epsilon\right)^i$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^{\epsilon}\right)^i$$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1)$$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^\epsilon\right)^i$$

$$= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1)$$

$$= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1)$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^\epsilon)^i$$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1)$$

$$= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1)$$

$$= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)$$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^\epsilon)^i$$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1)$$

$$= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1)$$

$$= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)$$

Also,

$$T(n) \leq \left(\frac{c}{b^\epsilon - 1} + 1\right) n^{\log_b(a)}$$

**Fall 1.** Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$$= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^\epsilon)^i$$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1)$$

$$= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1)$$

$$= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)$$

Also,

$$T(n) \leq \left(\frac{c}{b^\epsilon - 1} + 1\right) n^{\log_b(a)}$$

$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a})$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

$$T(n) - n^{\log_b a}$$

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}$$

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1
\end{aligned}
$$

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n)$$

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n) \qquad \boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n).}$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

Fall 2. Sei $f(n) \geq c n^{\log_b a}$.

$$T(n) - n^{\log_b a}$$

Fall 2. Sei $f(n) \geq c n^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 2. Sei $f(n) \geq c n^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}$$

Fall 2. Sei $f(n) \geq c n^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}$$

$$= c n^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Fall 2. Sei $f(n) \geq c n^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= c n^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= c n^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$T(n) = \Omega(n^{\log_b a} \log_b n)$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$T(n) = \Omega(n^{\log_b a} \log_b n) \qquad \boxed{\Rightarrow T(n) = \Omega(n^{\log_b a} \log n).}$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq c n^{\log_b a} (\log_b(n))^k$.

$$T(n) - n^{\log_b a}$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq c n^{\log_b a} (\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b \left(\frac{n}{b^i}\right)\right)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq c n^{\log_b a} (\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b \left(\frac{n}{b^i}\right)\right)^k$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell - 1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \boxed{\sum_{i=1}^{\ell} i^k} \approx \tfrac{1}{k}\ell^{k+1}$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k$$

$$\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}$$

**Fall 2.** Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k$$

$$\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}$$

$$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n).$$

# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:

# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

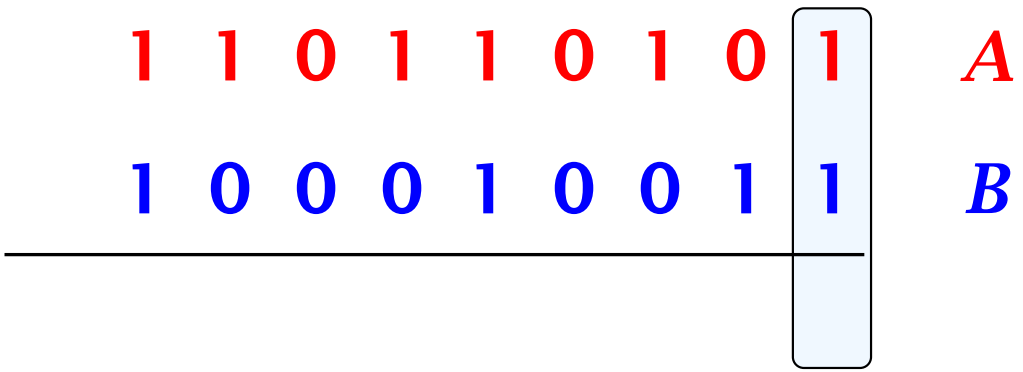
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:

$$\begin{array}{rrrrrrrrrr} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline \end{array}$$

# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

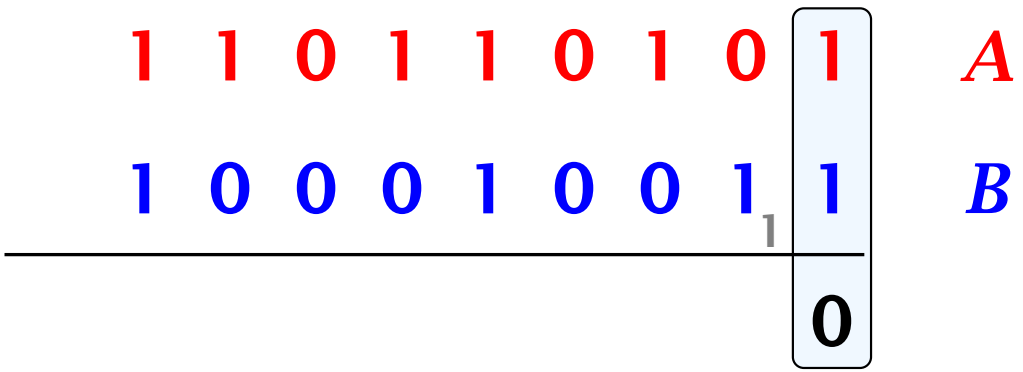
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

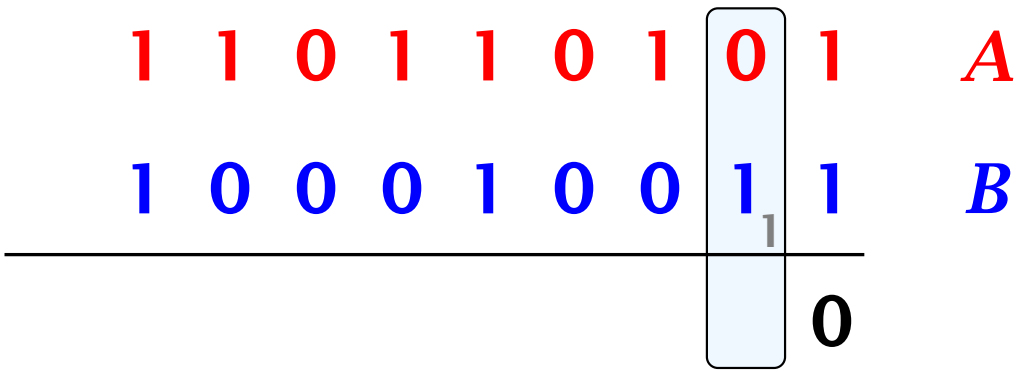
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

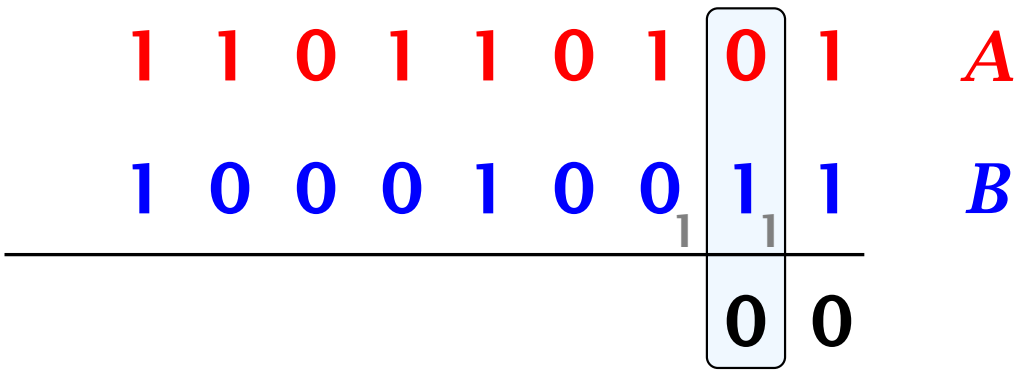
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

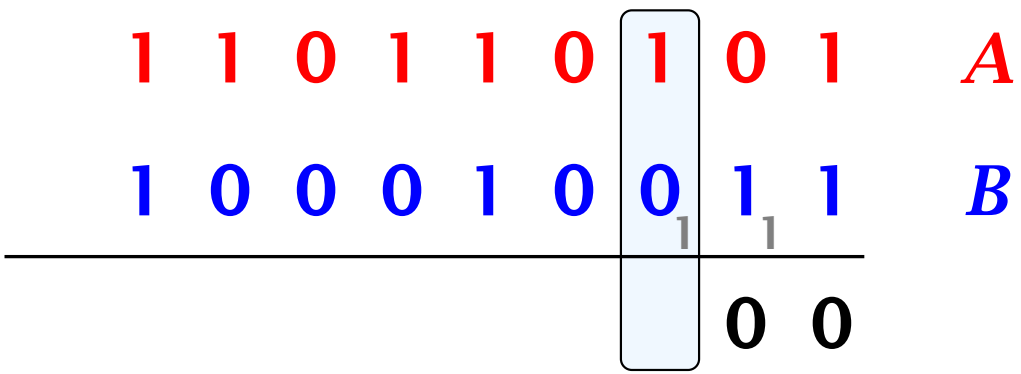
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

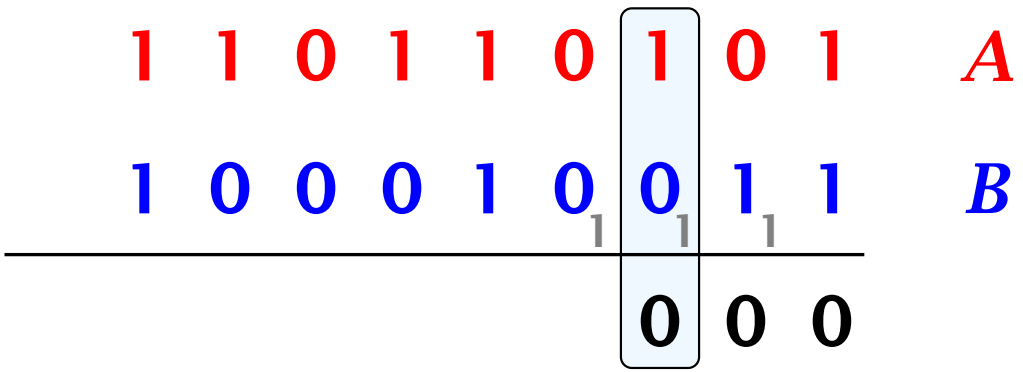
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

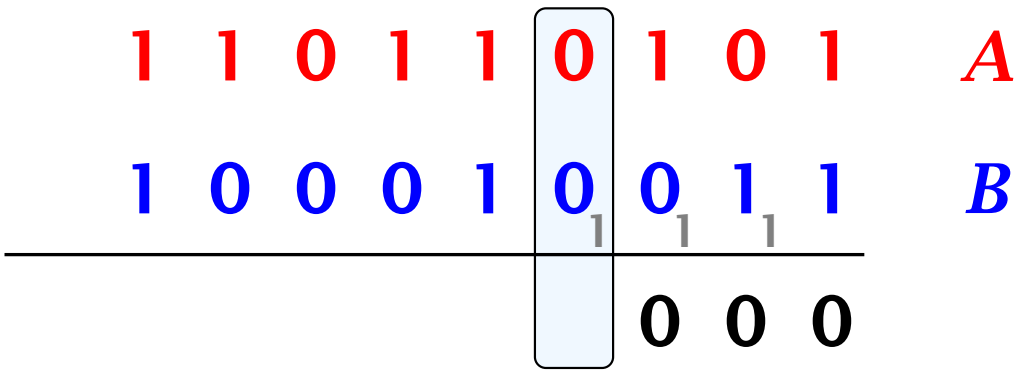
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

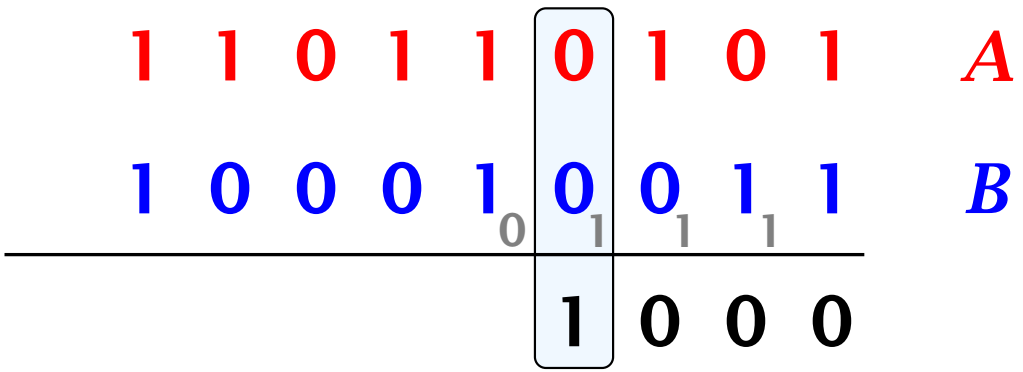
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

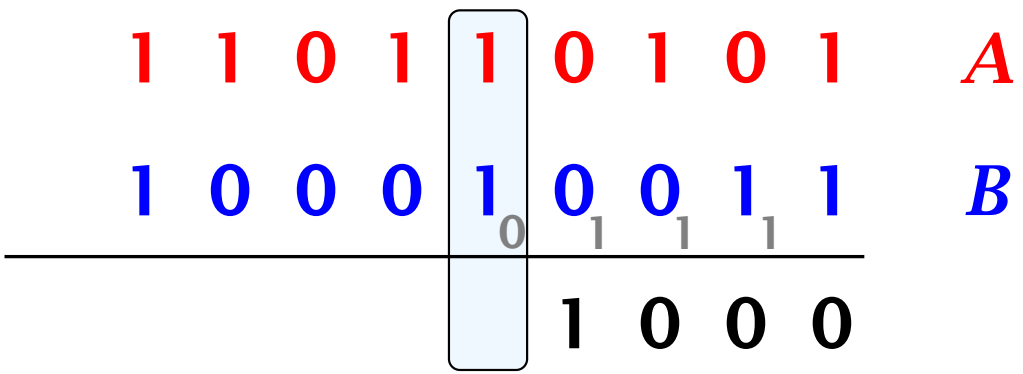
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

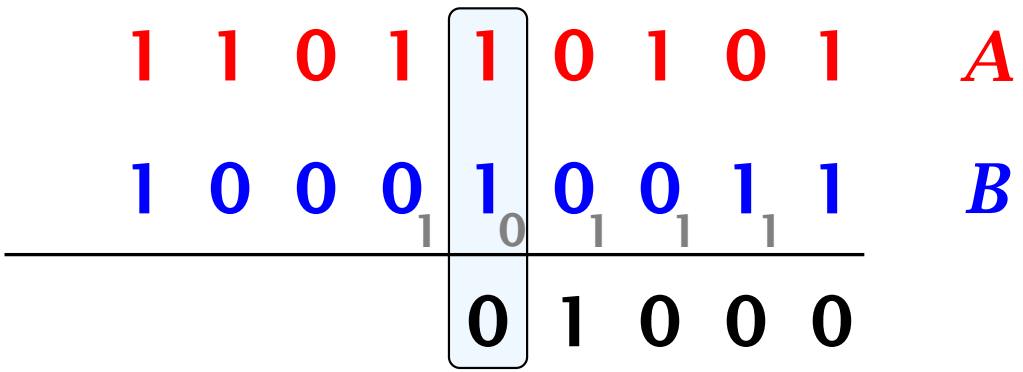
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

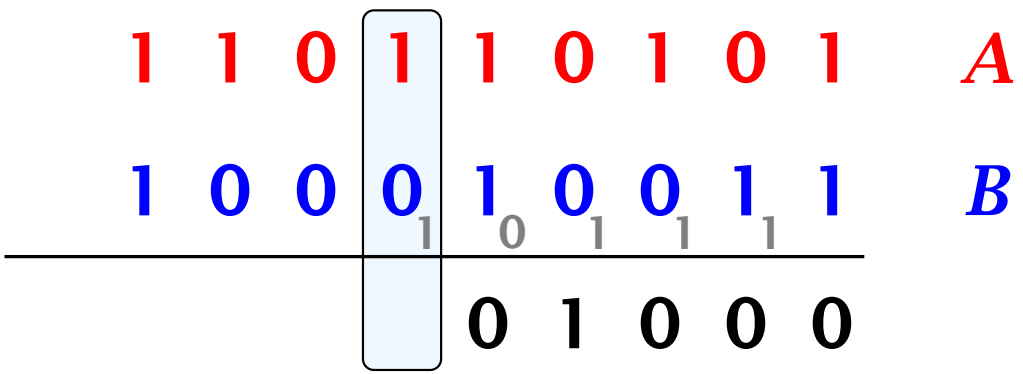
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

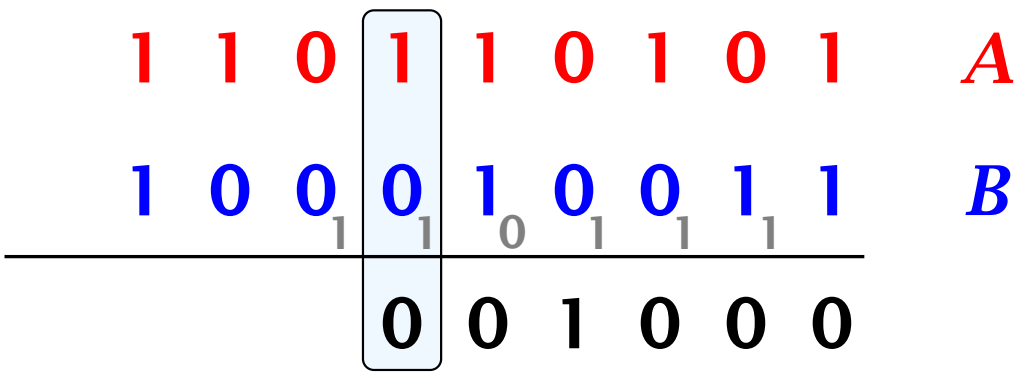
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

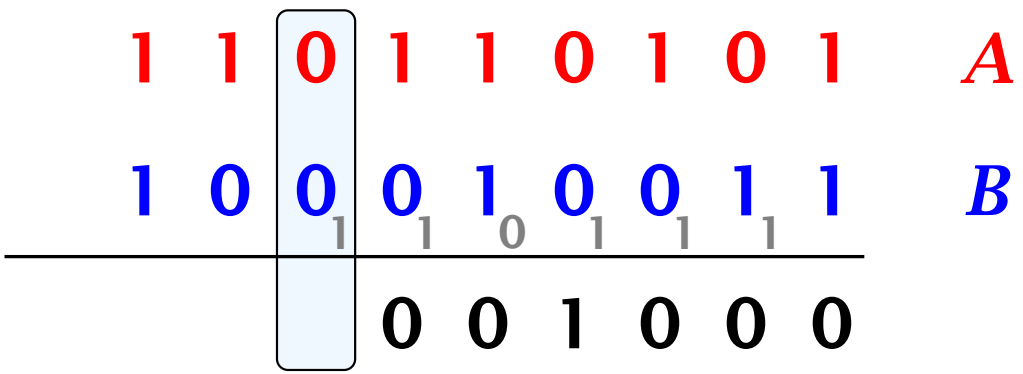
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

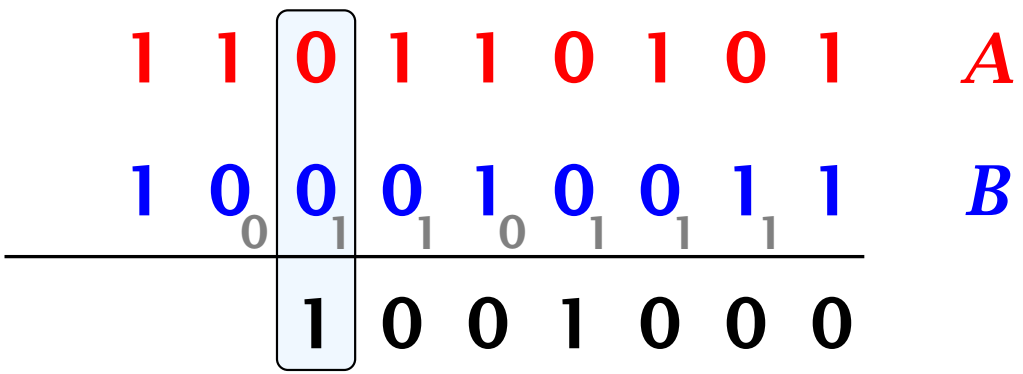
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

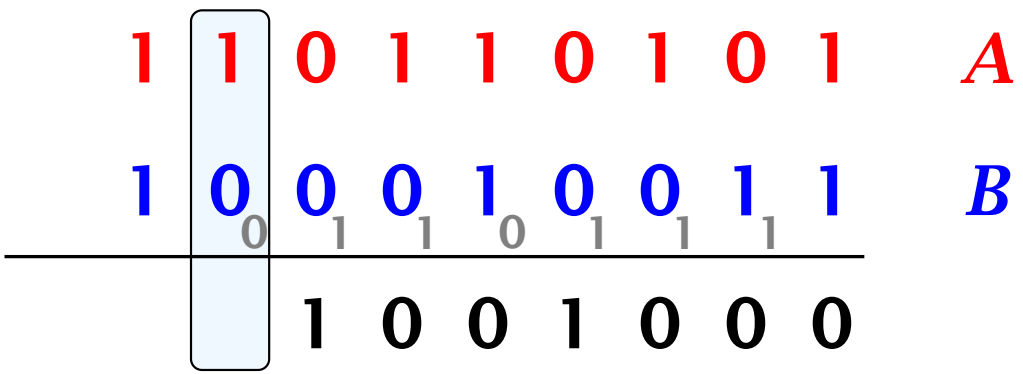
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

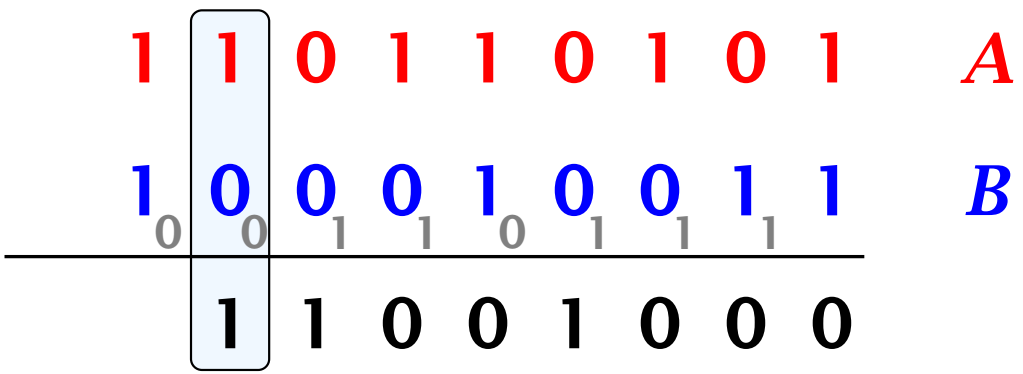
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

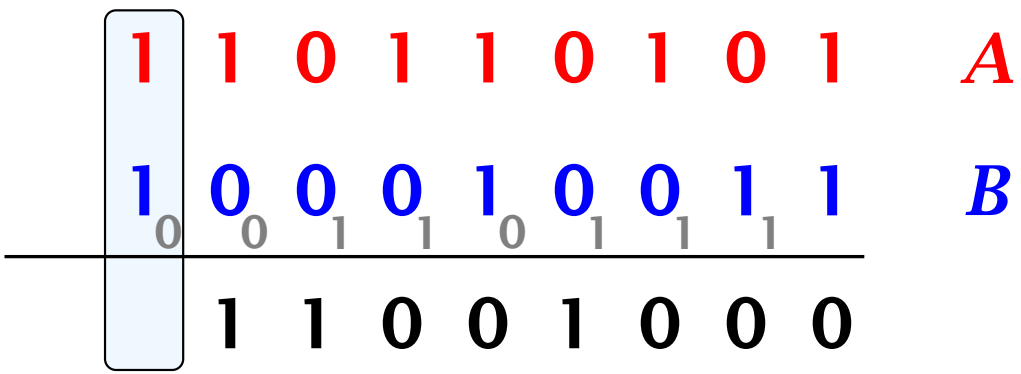
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

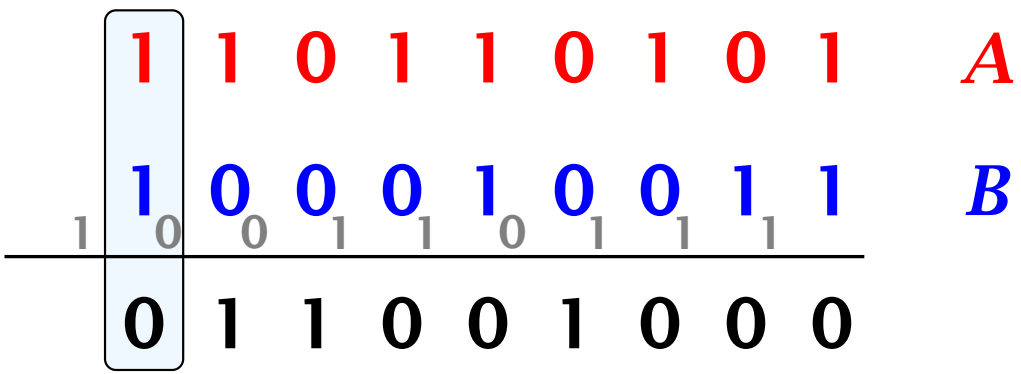
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

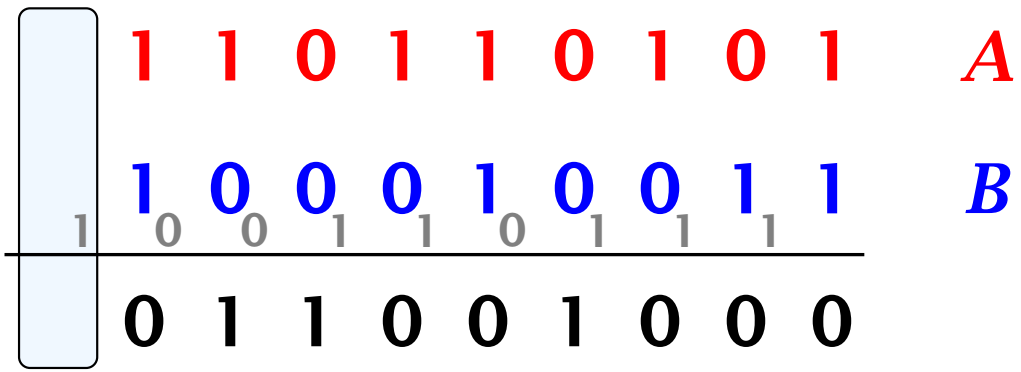
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

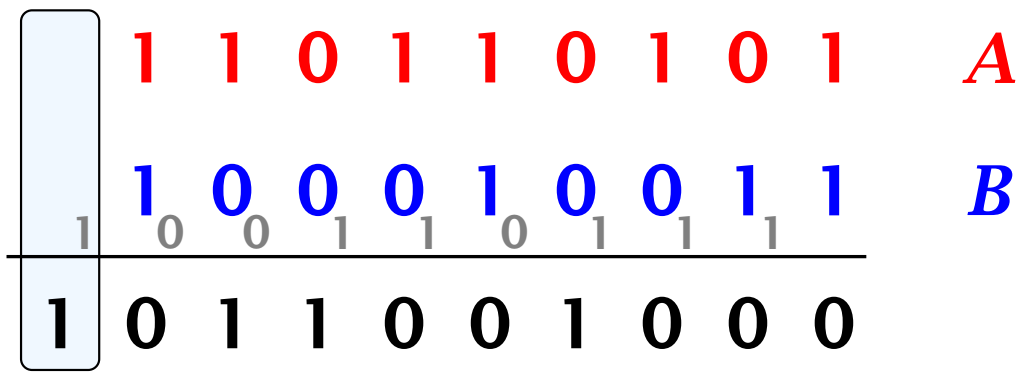
Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:



# Beispiel: Integermultiplikation

Angenommen wir möchten zwei $n$-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen $A$ and $B$ addieren können:

$$\begin{array}{cccccccccc} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & & \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \end{array}$$

Das heißt wir können zwei $n$-bit Integer in Zeit $\mathcal{O}(n)$ addieren.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \\ \times\ 1\ 0\ 1\ 1 \\ \hline \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\underline{1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ \boxed{1}}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ \boxed{1} \\ \hline 1\ 0\ 0\ 0\ 1 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ \boxed{1}\ 1 \\ \hline 1\ 0\ 0\ 0\ 1 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{rrrrrcrrrr} 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\ \hline & & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & & & & & & 0 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{rrrrrcrrrr} 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\ \hline & & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{rrrrrcrrrr} 1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\ \hline & & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{rrrrrcrrrr} 1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\ \hline & & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\ & & & & & & & & & 0 & 0 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{rrrrrrrrrr} 1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\ \hline & & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$\begin{array}{rrrrrrrrrr} 1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\ \hline & & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

$$
\begin{array}{rrrrrcrrrr}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 0 & 1 & \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & & & & & 0 & 0 & 0
\end{array}
$$

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrcrrrr}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 0 & 1 & \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & \\
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrcrrrr}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 0 & 1 & \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & \\
\hline
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrcrrrr}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 0 & 1 & \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & &
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

**Laufzeit:**

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrcrrrr}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 0 & 1 & \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & &
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

**Laufzeit:**

- Zwischenergebnisse berechnen: $\mathcal{O}(nm)$.

# Beispiel: Integermultiplikation

Angenommen wir möchten ein $n$-bit Integer $A$ und ein $m$-bit Integer $B$ ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrcrrrr}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 0 & 1 & \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & &
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

**Laufzeit:**

- Zwischenergebnisse berechnen: $\mathcal{O}(nm)$.
- Addieren von $m$ Zahlen der Länge $\leq 2n$: $\mathcal{O}((m+n)m) = \mathcal{O}(nm)$.

# Beispiel: Integermultiplikation

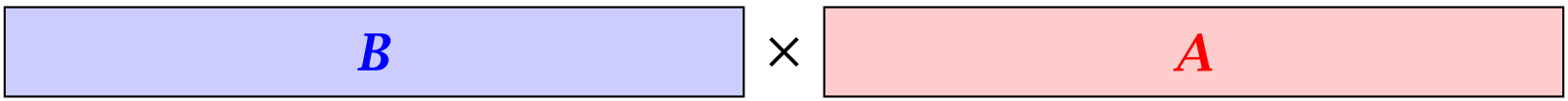
**Ein rekursiver Ansatz:**

Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.

# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

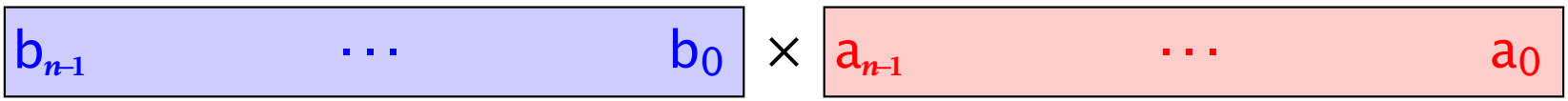
Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.



# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

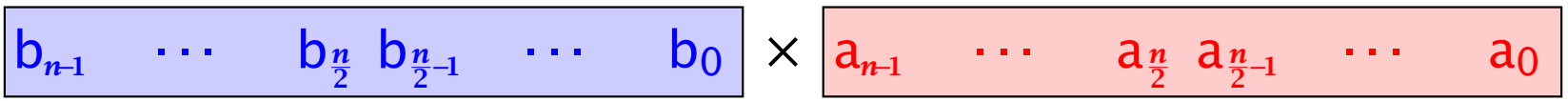
Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.



# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

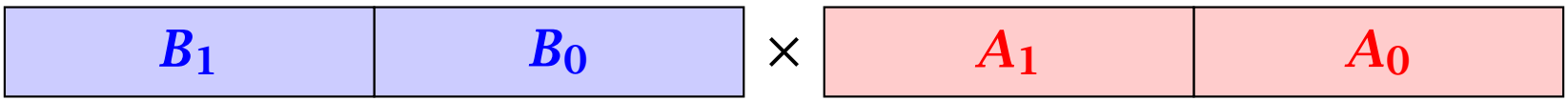
Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.



# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

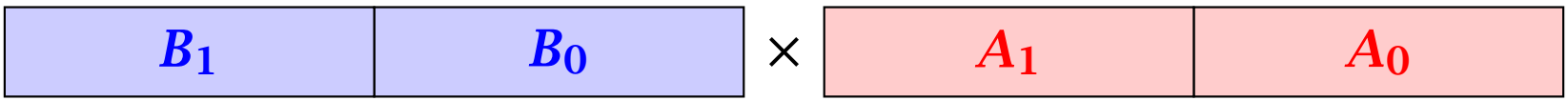
Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.



# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.



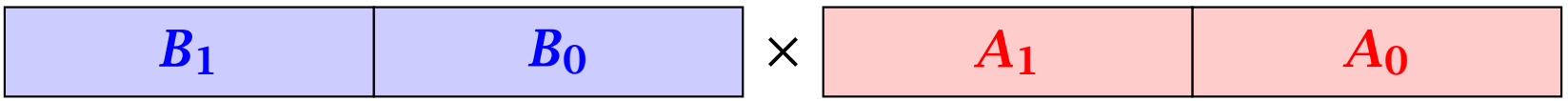

Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

Angenommen die Integer $A$ und $B$ haben Länge $n = 2^k$.




Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Also,

$$A \cdot B = A_1 B_1 \cdot 2^n + (A_1 B_0 + A_0 B_1) \cdot 2^{\frac{n}{2}} + A_0 B_0$$

# Beispiel: Integermultiplikation

```
1  Input:  Zahlen A, B, repraesentiert durch Bitarrays
2          der Laenge n
3  Output: Bitarray, das A*B enthaelt
4
5  mult(A, B, n)
6      if (n == 1)
7          return new int(A[0]*B[0]);
8      split(A,A0,A1);
9      split(B,B0,B1);
10     Z2 = mult(A1,B1,n/2);
11     Z1 = mult(A0,B1,n/2) + mult(A1,B0,n/2);
12     Z0 = mult(A0,B0,n/2);
13     return Z2*2^n + Z1*2^(n/2) + Z0;
```

Die Addition + addiert Bitarrays. Das funktioniert in C++ nicht direkt, aber man kann stattdessen z.B. eine Funktion add(A,B) benutzen. Wenn man den Algorithmus nach C++ übertragen möchte ist das Speichermanagement etwas unübersichtlich, da man jedes Zwischenergebnis vor Funktionsende wieder von Hand löschen muss.

Wir erhalten folgende Rekurrenzgleichung:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall $a = 4$, $b = 2$, und $f(n) = \Theta(n)$. Also, Fall 1, da $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall $a = 4$, $b = 2$, und $f(n) = \Theta(n)$. Also, Fall 1, da $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

Wir erhalten Laufzeit $\mathcal{O}(n^2)$.

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall $a = 4$, $b = 2$, und $f(n) = \Theta(n)$. Also, Fall 1, da $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

Wir erhalten Laufzeit $\mathcal{O}(n^2)$.

$\Longrightarrow$ Nicht besser als „Schulmethode“.

# Beispiel: Integermultiplikation

We can use the following identity to compute $Z_1$:

# Beispiel: Integermultiplikation

We can use the following identity to compute $Z_1$:

$$Z_1 = A_1 B_0 + A_0 B_1$$

# Beispiel: Integermultiplikation

We can use the following identity to compute $Z_1$:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - A_1B_1 - A_0B_0 \end{aligned}$$

# Beispiel: Integermultiplikation

We can use the following identity to compute $Z_1$:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1B_1}_{= Z_2} - \underbrace{A_0B_0}_{= Z_0} \end{aligned}$$

# Beispiel: Integermultiplikation

We can use the following identity to compute $Z_1$:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \qquad\qquad \underbrace{= Z_2} \quad \underbrace{= Z_0} \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1B_1} - \underbrace{A_0B_0} \end{aligned}$$

```
1  Input:  Zahlen A, B, repraesentiert durch Bitarrays
2          der Laenge n
3  Output: Bitarray, das A*B enthaelt
4
5  mult(A, B, n)
6      if (n == 1)
7          return new int(A[0]*B[0]);
8      split(A,A0,A1);
9      split(B,B0,B1);
10     Z2 = mult(A1,B1,n/2);
11     Z1 = mult(A0+A1,B0+B1,n/2)-Z0-Z2;
12     Z0 = mult(A0,B0,n/2);
13     return Z2*2^n + Z1*2^(n/2) + Z0;
```

# Beispiel: Integermultiplikation

Zeile 11 ist leider nicht korrekt, da A0 + A1 bzw. B0 + B1 eventuell $n/2+1$ bits haben können. Wenn man eine $n/2+1$-bit Zahl X in das höchstwertige Bit $X_{\frac{n}{2}}$ und die restlichen Bits $\bar{X}$ zerlegt kann man folgendes nutzen:

```
...
 X = A0 + A1;
 Y = B0 + B1;
Z1 = X_{n/2}*Y_{n/2}*2^n + (X_{n/2}*Ȳ + Y_{n/2}*X̄)*2^{n/2} + mult(X̄,Ȳ) - Z0 - Z2;
...
```

Laufzeit hierfür ist $T(n/2)+\mathcal{O}(n)$.

# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

**Master Theorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

**Master Theorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

**Master Theorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

**Master Theorem:** Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

# Algorithmenentwurf

Kein Patentrezept zum Entwurf von Algorithmen!

- insbesondere Ableitung von Algorithmus aus Spezifikation nicht automatisierbar

Programmieren ist **kreative** Tätigkeit

- “The Art of Computer Programming” (D. Knuth)

Unterstützung durch Algorithmenmuster

- auch **Design Patterns** genannt
- “best practice”

# Divide and Conquer

**Definition: Divide and Conquer**
Divide and Conquer ist die rekursive Rückführung eines zu lösenden Problems auf mehrere identische Problem mit kleinerer Eingabemenge.

**Divide and Conquer**: zu deutsch “Teile und herrsche”

**Prinzip:**

- teile große Aufgabe in mehrere kleine Teilaufgaben
- rufe denselben Algorithmus rekursiv auf den Teilaufgaben auf

# Divide and Conquer

1. **Teile** gegebene Aufgabe in mehrere getrennte Teilaufgaben
   - **löse** Teilaufgaben einzeln
   - **setze** Lösung der Gesamtaufgabe aus Teillösungen zusammen

2. Wende dieselbe Technik auf jede Teilaufgabe an, dann auf deren Teilaufgaben etc., bis die Teilaufgabe so klein ist, dass Lösung explizit berechnet werden kann

3. Jede Teilaufgabe sollte von derselben Art sein wie die Gesamtaufgabe, so dass der gleiche Algorithmus rekursiv aufgerufen werden kann

# Divide and Conquer

```
1  Input: Aufgabe A
2
3  DivideAndConquer(A)
4      if (A klein)
5          loese A explizit;
6      else
7          teile A in Teilaufgaben A_1,...,A_n;
8          DivideAndConquer(A_1)
9          ...
10         DivideAndConquer(A_n)
11         berechne Loesung fuer A aus Lsgn fuer A_1,...,A_n
```

# Divide and Conquer

- Berechnung der Fibonacii-Zahlen (untypisch, da Teilprobleme Größe $n-$ und $n-2$ haben).
- Binäre Suche (nur ein Teilproblem der Größe $\leq n/2$).
- Karatsuba für die Multiplikation großer Zahlen.
- MergeSort
- QuickSort
- Fast Fourier Transformation (FFT)
- Medianberechnung
- ...

# Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der nach dem Divide and Conquer Prinzip arbeitet

John von Neumann (1945)

# Divide and Conquer: MergeSort

Sei $L$ verkette Liste mit $n$ natürlichen Zahlen $a_i \in \mathbb{N}$.

**Aufgabe:** sortiere $L$ in aufsteigender Reihenfolge.

Lösung mit Divide and Conquer-Muster: **MergeSort**

Idee:

- **Divide:** teile $L$ auf in zwei gleich große Teillisten
- **Rekursion:** rufe MergeSort rekursiv für die zwei Teillisten auf
- **Conquer:** setze die Teillisten zusammen (**merge** bzw. mischen)

Wann ist Teilliste **"klein"**, d.h. wann löst man explizit?
→ Teilliste mit nur **einem** Element → sortiert!

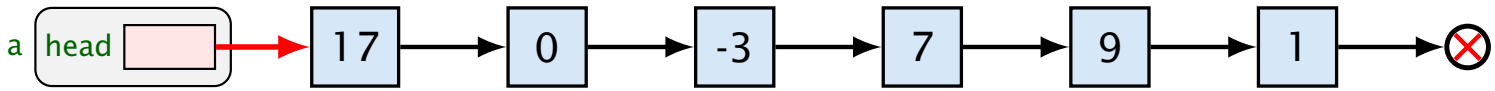
# Divide and Conquer: MergeSort

```
1 List* mergeSort(List* a) {
2     // returns a list with elements from a sorted
3     // may delete the list pointed to by a
4     if (a->length() <= 1) return a;
5     List* b = a->half();
6     a = mergeSort(a);
7     b = mergeSort(b);
8     return merge(a,b);
9 }
```

# Divide and Conquer: MergeSort

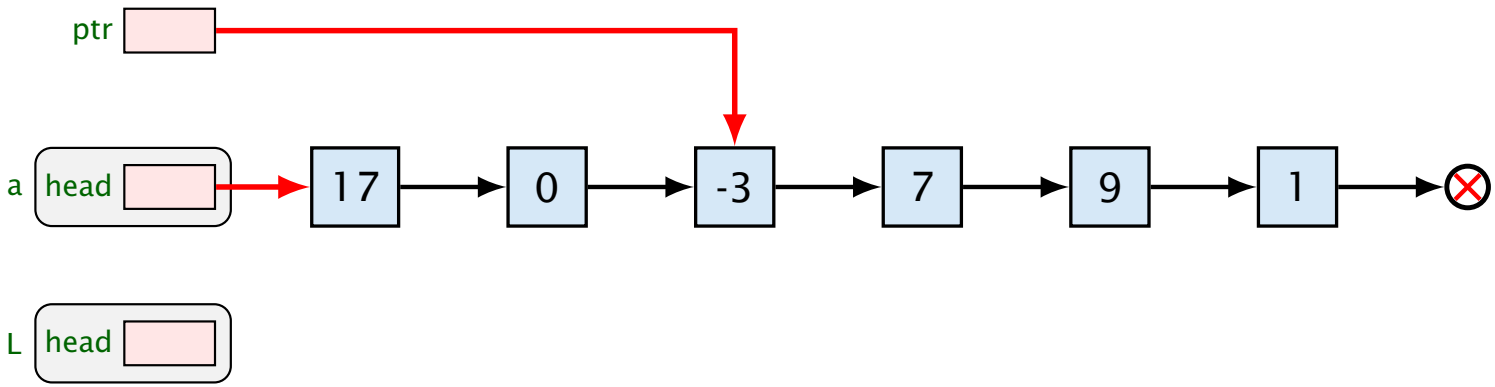
```
1  List* half() {
2      // removes elements from second half of list
3      // and returns these as a new list
4      Node* ptr = head;
5      for (int i=0; i<length()/2-1; i++)
6          ptr = ptr->next;
7      List* L = new List(ptr->next);
8      ptr->next = NULL;
9      return L;
10 }
```

# Beispiel – Halbieren

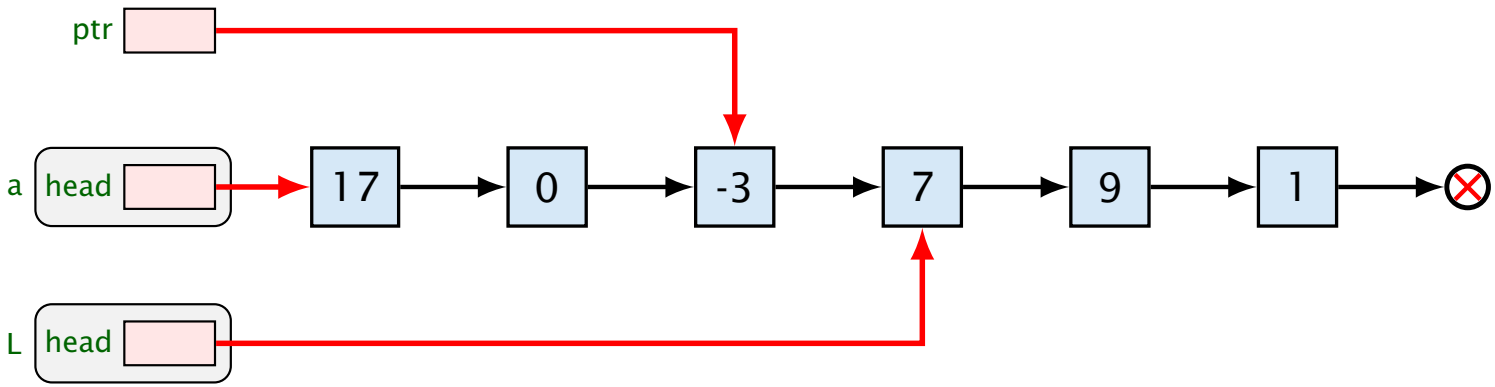



```
a.half()
```

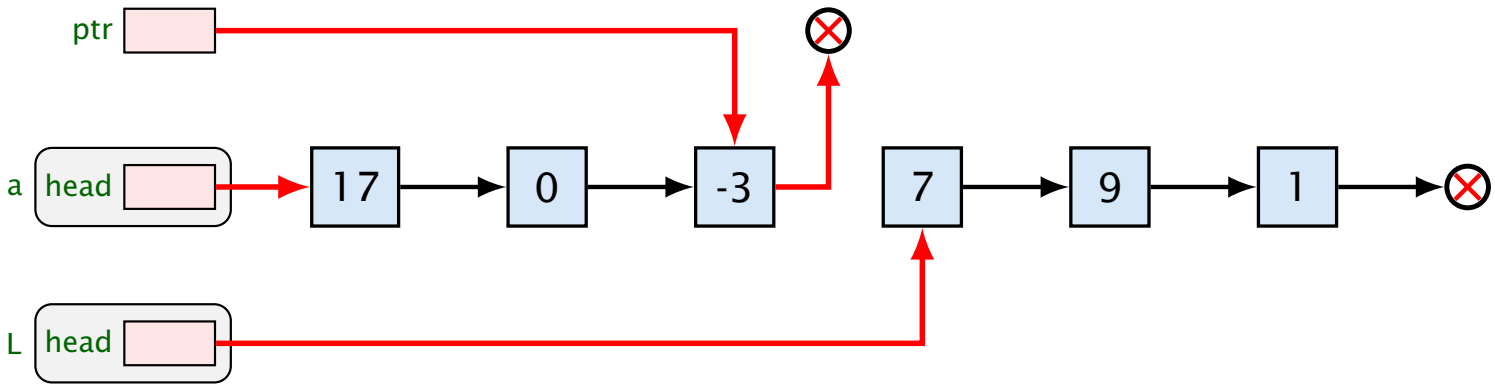
# Beispiel – Halbieren



# Beispiel – Halbieren

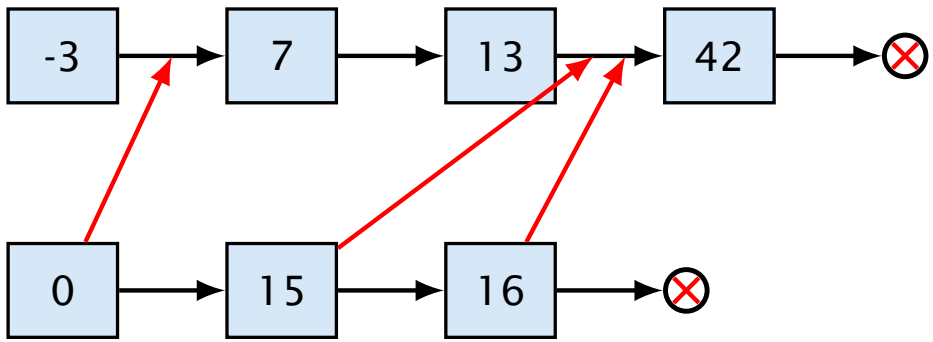


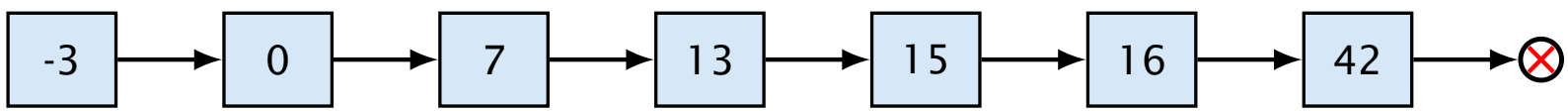
# Beispiel – Halbieren



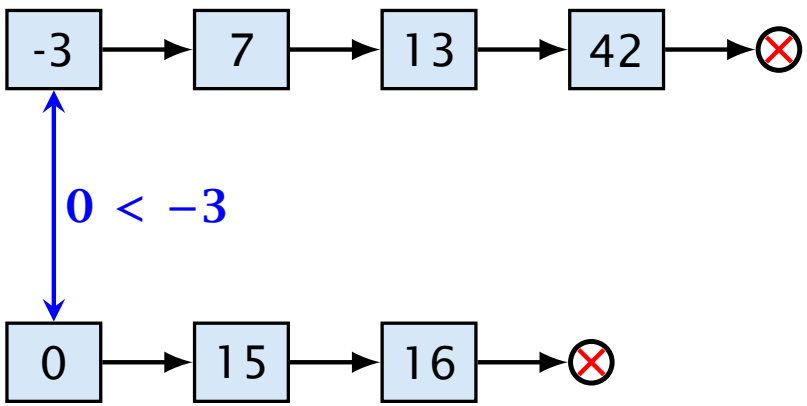
# Beispiel – Mischen

Hier benutzen wir das Symbol ⊗ für das `null`-Objekt.

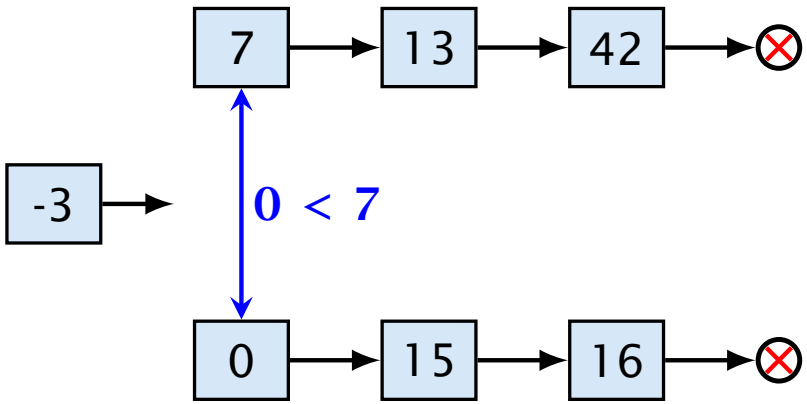


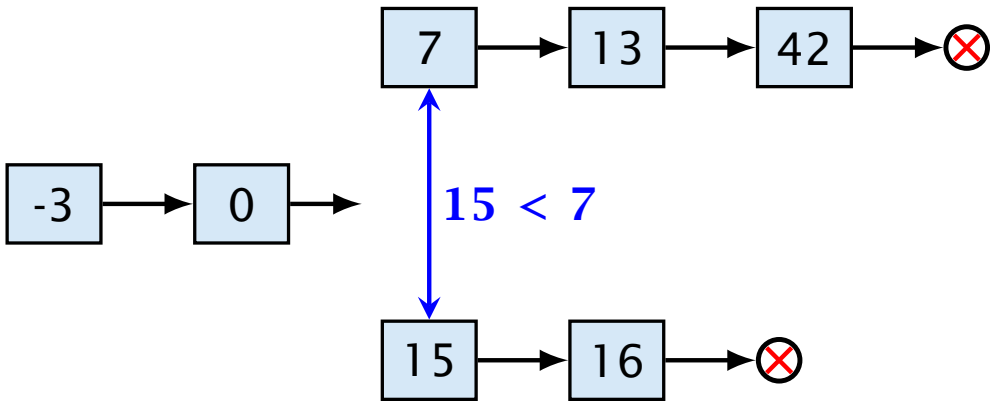
# Beispiel – Mischen



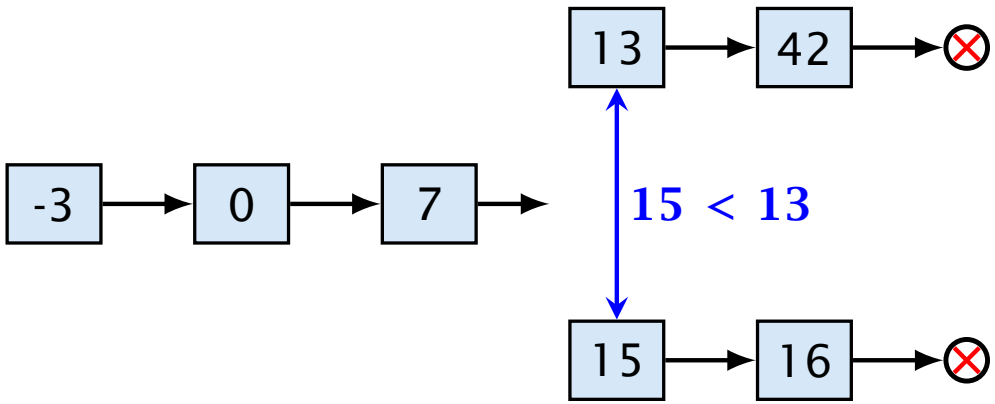
# Beispiel – Mischen



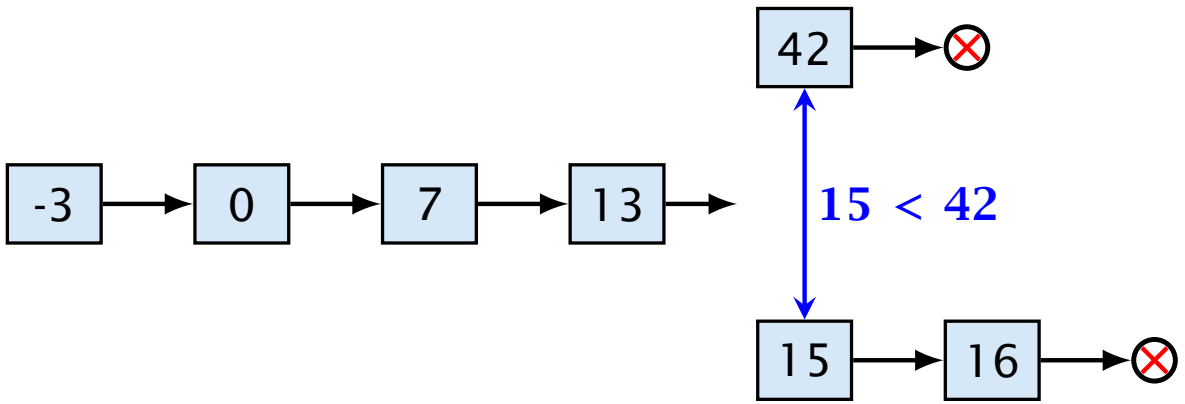
# Beispiel – Mischen



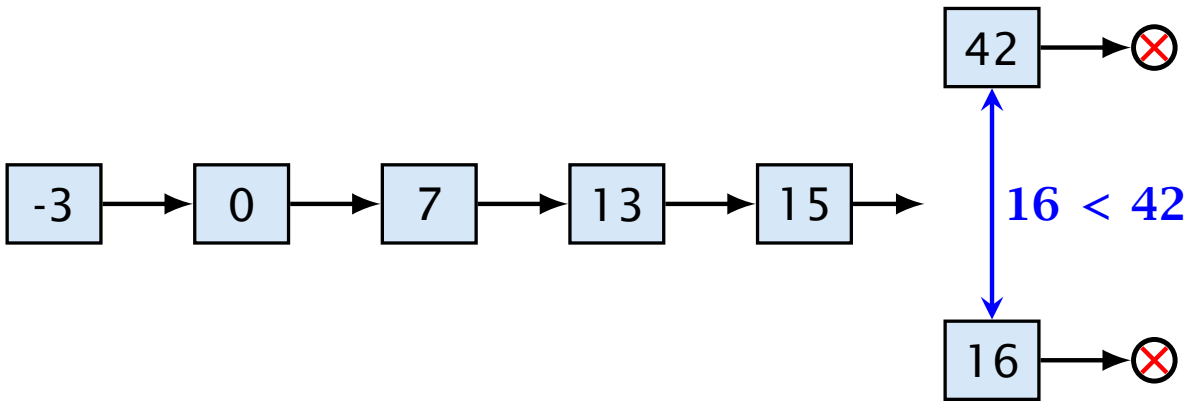
# Beispiel – Mischen



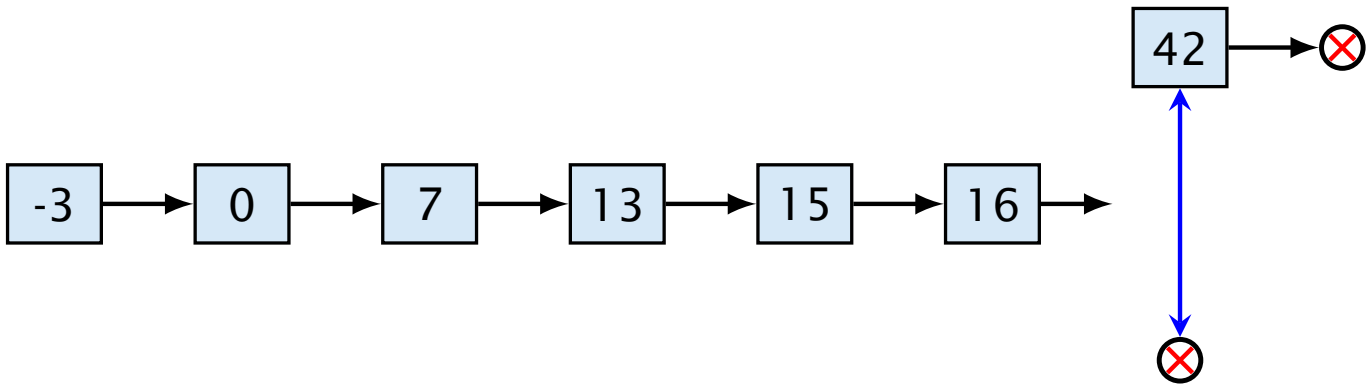
# Beispiel – Mischen



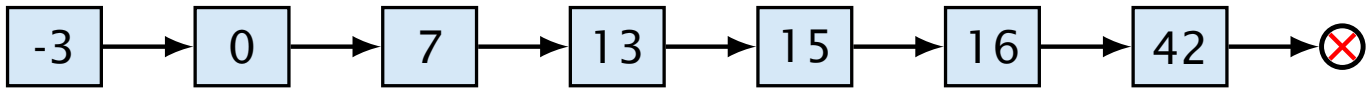
# Beispiel – Mischen



# Beispiel – Mischen



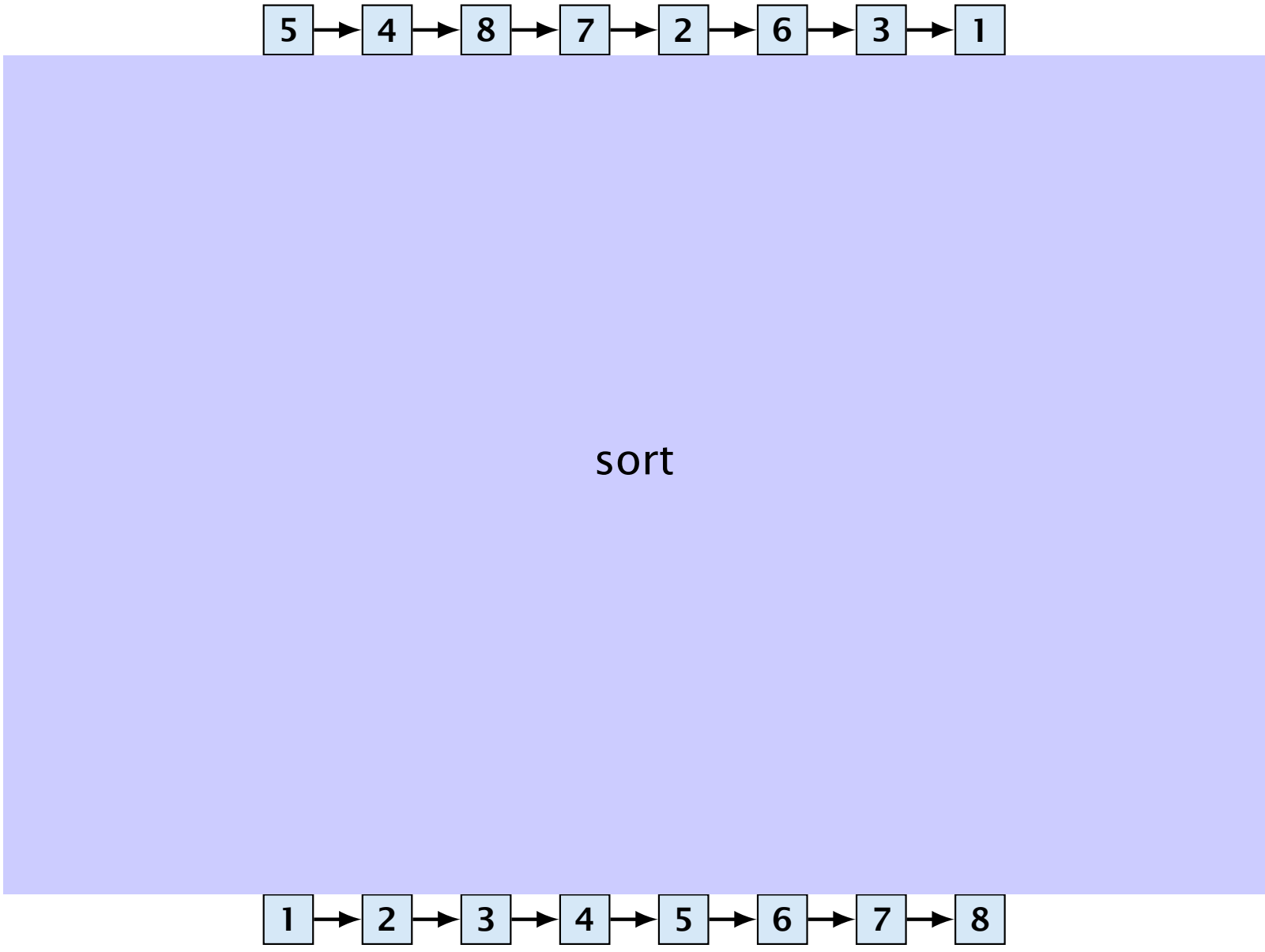
# Beispiel – Mischen



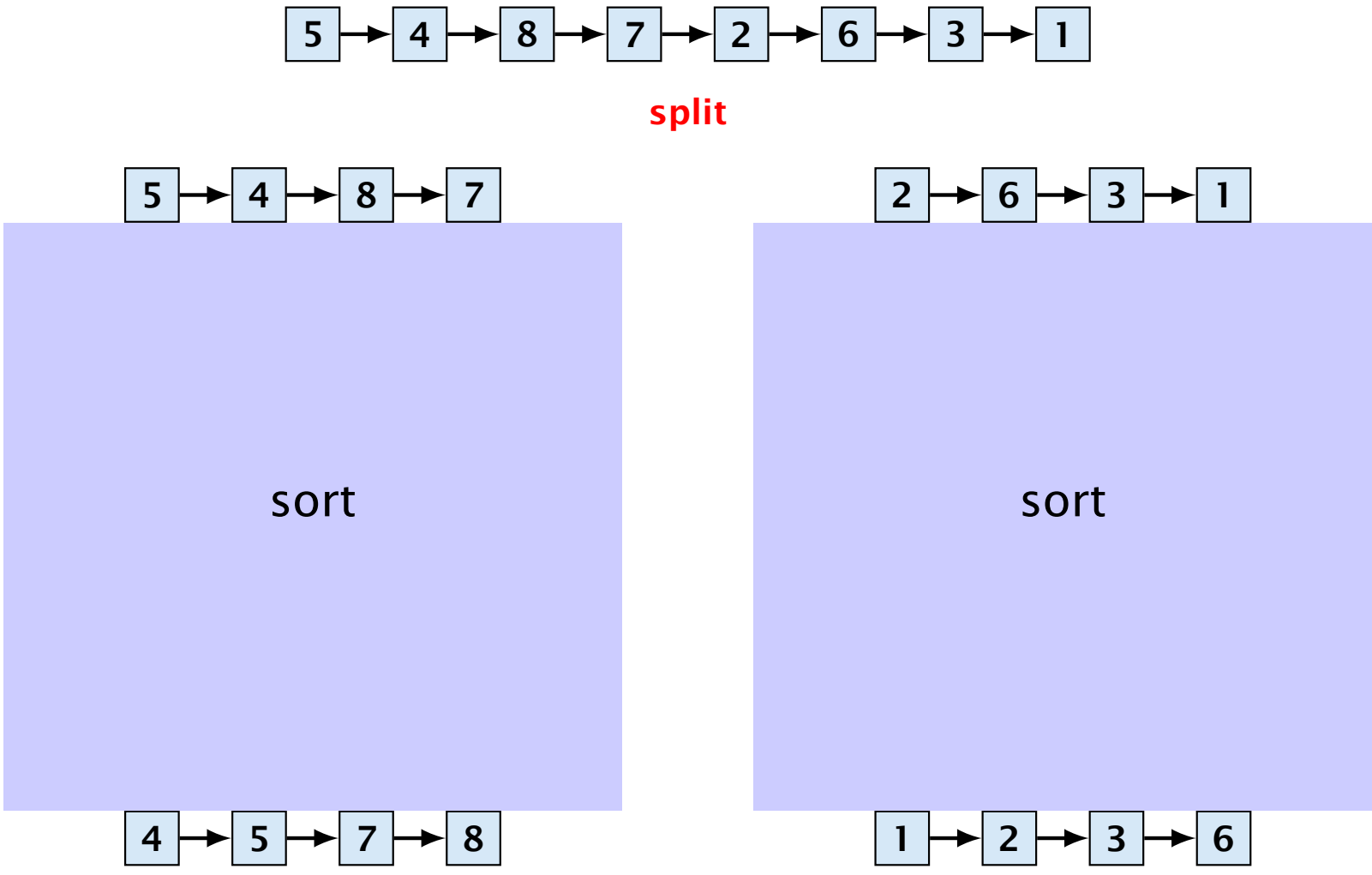
# Divide and Conquer: MergeSort

```
1  List* merge(List* a, List* b) {
2      // returns a list with elements from a and b
3      // the lists a and b may be deleted
4      if (a->empty()) { delete a; return b; }
5      if (b->empty()) { delete b; return a; }
6      List* h;
7      if (a->elementAt(0) < b->elementAt(0))
8          h = a;
9      else
10         h = b;
11     // remove element from h and recurse
12     int d = h->removeFront();
13     List* L = merge(a,b);
14     L->insertFront(d);
15     return L;
16 }
```

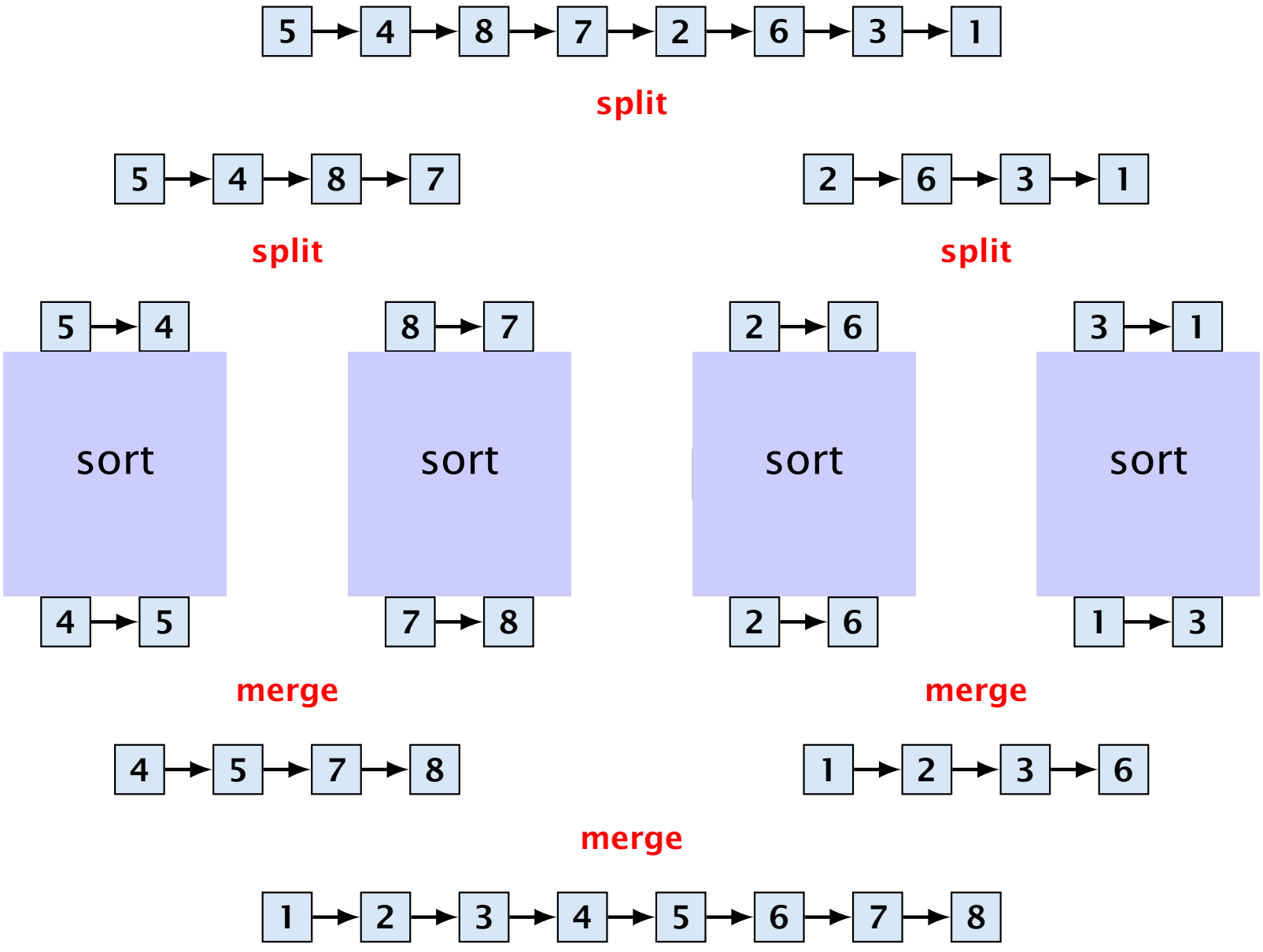
# Mergesort



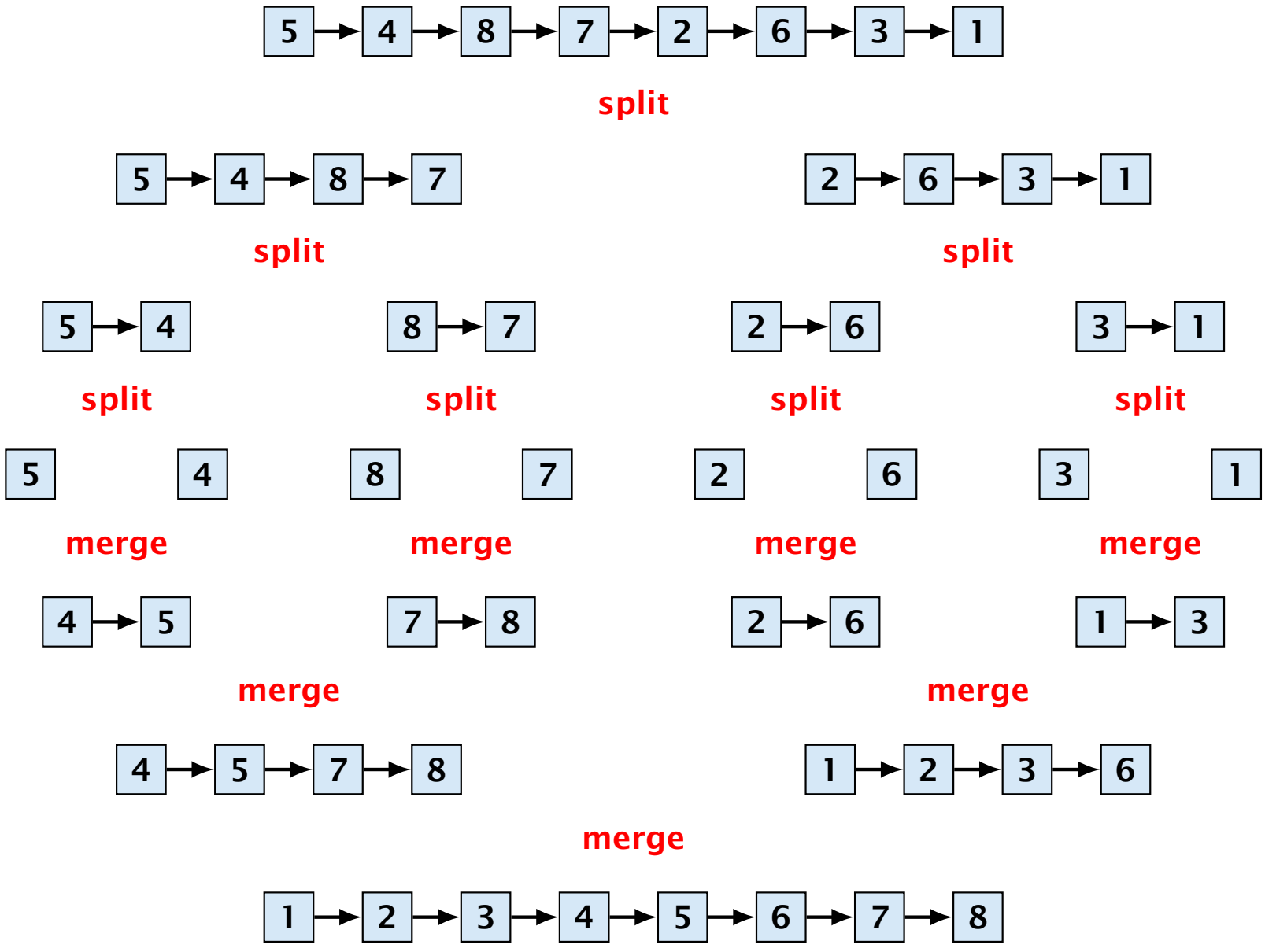
# Mergesort


5 4 8 7 2 6 3 1
split
5 4 8 7
2 6 3 1
sort
sort
4 5 7 8
1 2 3 6
merge
1 2 3 4 5 6 7 8

# Mergesort


5 4 8 7 2 6 3 1
split
5 4 8 7
split
2 6 3 1
split
5 4
8 7
2 6
3 1
sort
sort
sort
sort
4 5
7 8
2 6
1 3
merge
merge
4 5 7 8
1 2 3 6
merge
1 2 3 4 5 6 7 8

# Mergesort


5 4 8 7 2 6 3 1
split
5 4 8 7
2 6 3 1
split
split
5 4
8 7
2 6
3 1
split
split
split
split
5
4
8
7
2
6
3
1
merge
merge
merge
merge
4 5
7 8
2 6
1 3
merge
merge
4 5 7 8
1 2 3 6
merge
1 2 3 4 5 6 7 8

# MergeSort

**Eigenschaften**

- Merge Sort benötigt zusätzlichen Speicher in Funktion `merge`; insgesamt $n$ zusätzliche Elemente falls $A$ Länge $n$
- best und worst case sind identisch
- die meiste Arbeit steckt in **merge**; ein Aufruf von merge auf zwei Listen der Länge $n/2$ Kosten Zeit $\mathcal{O}(n)$.

# MergeSort

**Rekurrenzgleichung:**

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

**Laufzeit:** $\mathcal{O}(n \log n)$

# Sortieralgorithmen Zusammenfassung

**Insertion Sort**

- in-place
- Komplexität $\mathcal{O}(n^2)$, best case: $\mathcal{O}(n)$
- Komplexität $\mathcal{O}(hn)$ falls jedes Element nur $h$ Positionen von Zielposition entfernt

**MergeSort**

- benötigt zusätzlichen Speicher
- Komplexität $\mathcal{O}(n \log n)$

**QuickSort**

- in-place
- Komplexität im Mittel $\mathcal{O}(n \log n)$, worst case: $\mathcal{O}(n^2)$

# Algorithmenmuster: Greedy

**greedy** = “gierig”, “gefräßig”

**Greedy Prinzip:**

- Lösung eines Problems durch schrittweise Erweiterung der Lösung ausgehend von Startlösung
- in jedem Schritt wähle den bestmöglichen Schritt (ohne Berücksichtigung zukünftiger Schritte) ⇒ **greedy**

gefundene Lösung muss nicht immer optimal sein!

# Algorithmenmuster: Greedy

```
1 Input: Aufgabe A
2
3 Greedy(A)
4     S = {}; // Loesung
5     while (S keine Loesung)
6         waehle bestmoeglichen Erweiterungschritt s
7         erweitere S mit s
```

# Greedy: Beispiel Wechselgeld I

**Problem:** Herausgabe von Wechselgeld

- **Voraussetzung:** übliche Euro-Münzen 2€, 1€, 50ct, 20ct, 10ct, 5ct, 2ct und 1ct
- **Aufgabe:** Wechselgeld-Herausgabe mit möglichst wenig Münzen

**Beispiel:** Preis €1.11, bezahlt mit 2€-Münze. Wechselgeld: 89ct

Minimum Anzahl Münzen: 6

$$89\text{ct} = 50\text{ct} + 20\text{ct} + 10\text{ct} + 5\text{ct} + 2\text{ct} + 2\text{ct}$$

# Greedy: Beispiel Wechselgeld II

```
1  Input: Betrag b
2
3  Wechselgeld(b)
4      printf("%d = ",b);
5      count = 0;
6      while (count < b)
7          // make greedy choice
8          waehle groesste Muenze s mit count + s <= b
9          printf("%d ",s);
10         count += s;
```

**Achtung:** Abhängig vom Geldsystem liefert dieser Algorithmus nicht immer optimale Lösung!

- **Beispiel:** Münzen: 5ct, 4ct, 1ct. **Betrag:** 8ct
- **Greedy-Lösung:** 8ct = 5ct + 1ct + 1ct + 1ct
- **Optimale Lösung:** 8ct = 4ct + 4ct

# Von Greedy lösbare Probleme

**Voraussetzungen** für Anwendbarkeit von Greedy:

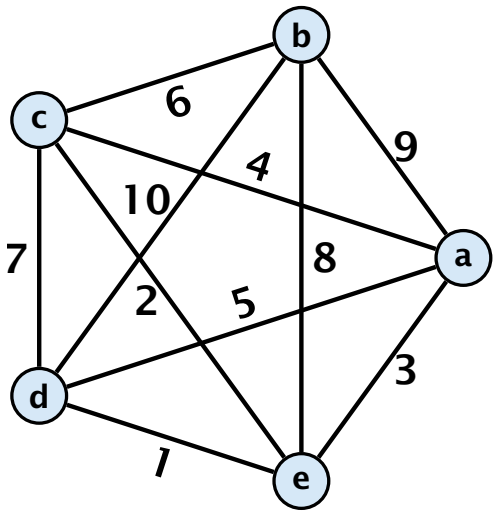
- Lösungen lassen sich schrittweise durch Hinzufügen von Elementen aufbauen, beginnend bei leerer Lösung
- Bewertungsfunktion für partielle und vollständige Lösung
- Gesucht wird eine/die optimale Lösung

# Anwendung Greedy: Glasfasernetz

**Problemstellung:** Aufbau von möglichst billigem Glasfasernetz zwischen $n$ Knoten $K_1, \ldots, K_n$, so dass alle Knoten miteinander verbunden sind (u.U. mit Umweg)

**Input:**

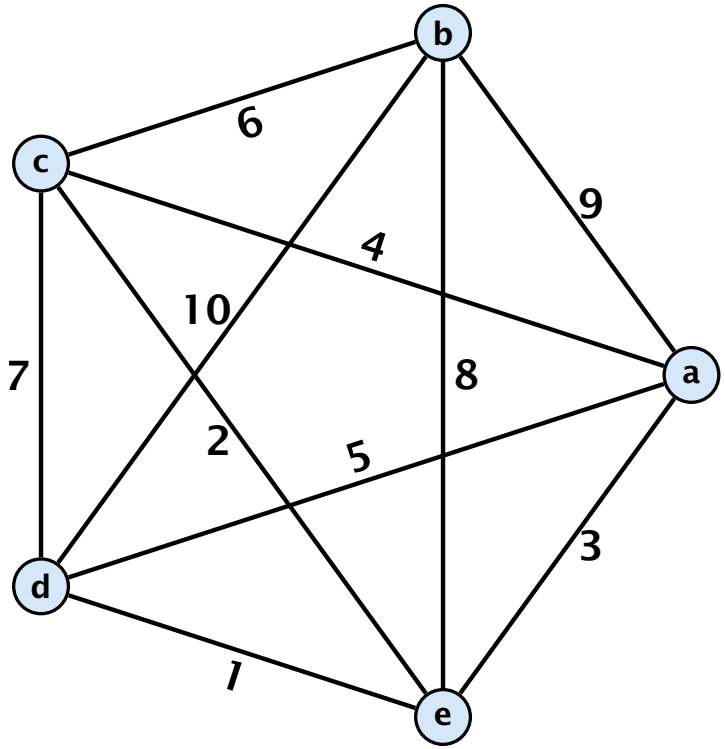
- Knoten $a, b, c, \ldots$
- Kosten $d_{ij} > 0$ für direkte Verbindung zwischen $i$ und $j$ für $i \neq j$.




**Output:** Teilmenge aller Verbindungen, so dass

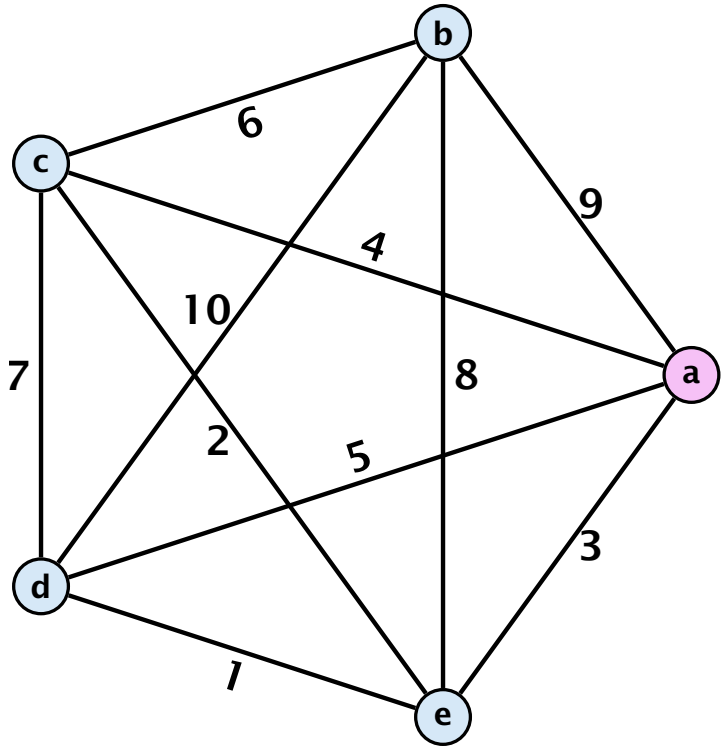
- alle Knoten verbunden sowie
- minimale Kosten

# Beispiel: Glasfasernetz



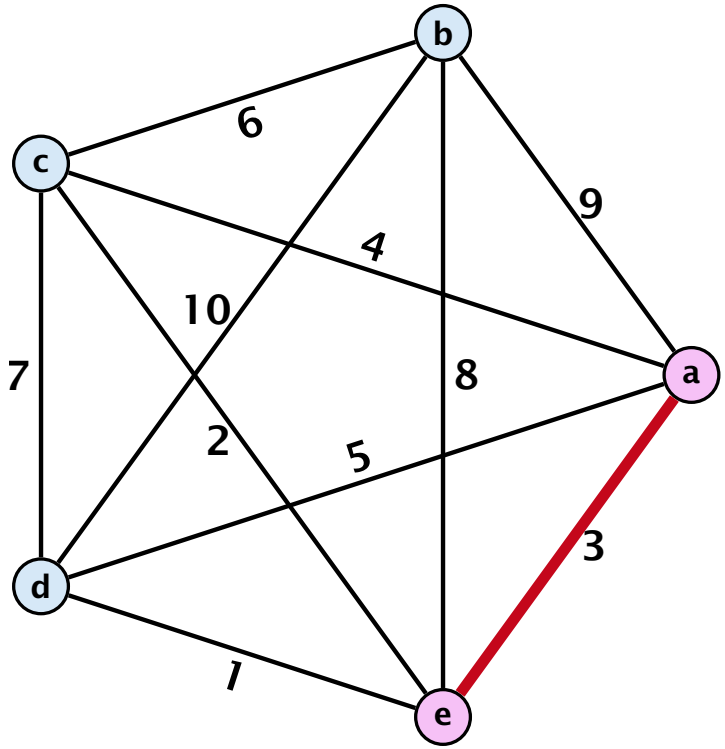

- Knoten $a, b, c, d, e$
- Kosten $d_{ij}$
- repräsentiert als gewichteter, ungerichteter Graph

# Beispiel: Glasfasernetz



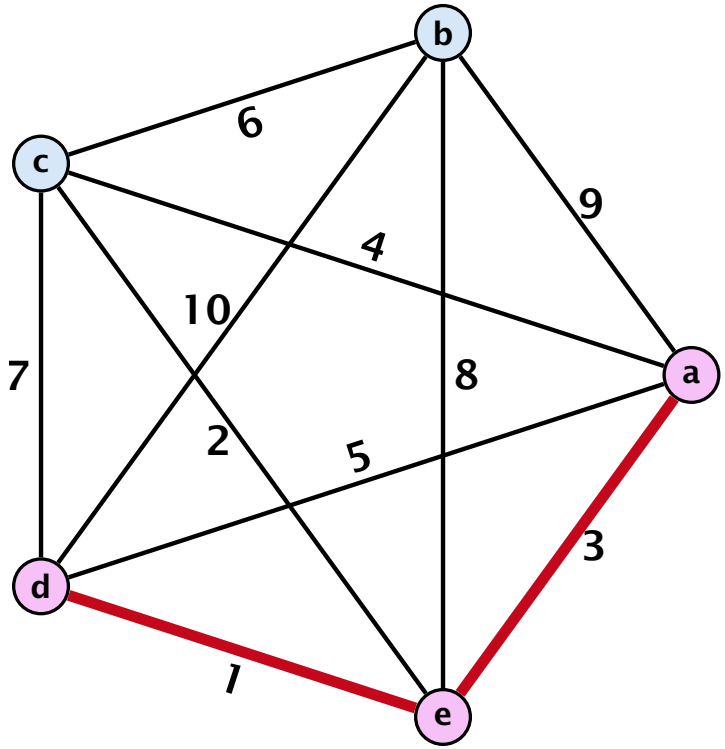

- Startknoten $a$
- beste Verbindung aus $\{a\}$ führt zu $e$ (Kosten 3)

# Beispiel: Glasfasernetz



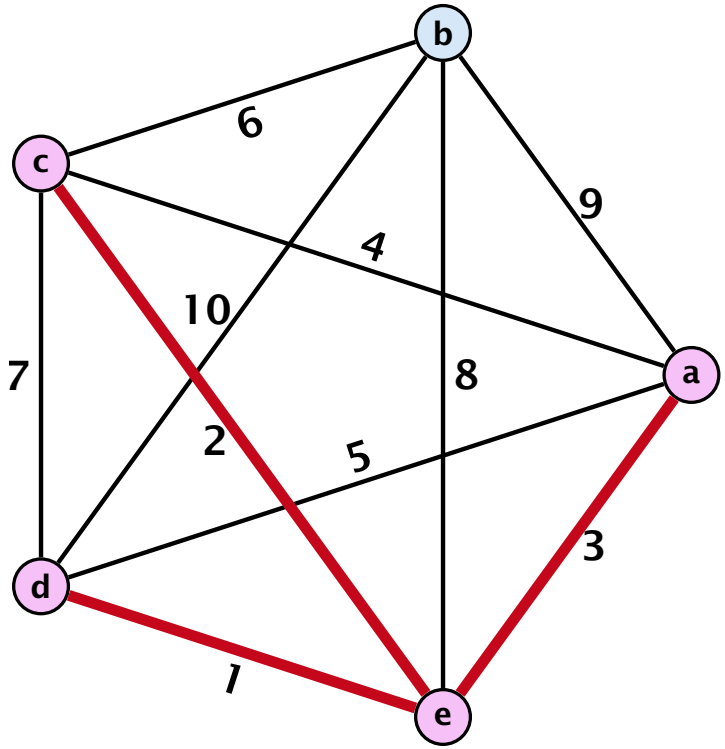

- Menge $\{a, e\}$
- beste Verbindung aus $\{a, e\}$ führt zu $d$ (Kosten 1)

# Beispiel: Glasfasernetz



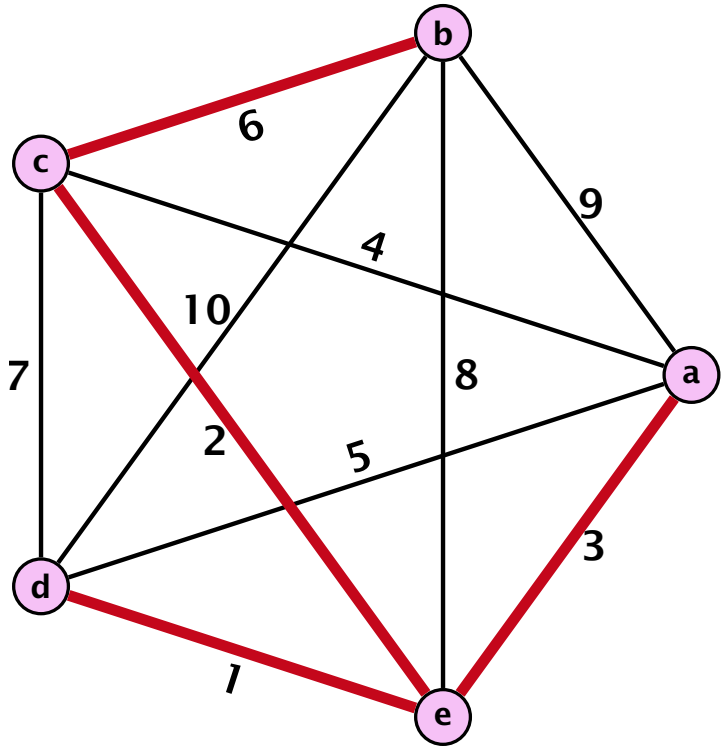

- Menge $\{a, d, e\}$
- beste Verbindung aus $\{a, d, e\}$ führt zu $c$ (Kosten 2)

# Beispiel: Glasfasernetz




- Menge $\{a, c, d, e\}$
- beste Verbindung aus $\{a, c, d, e\}$ führt zu $b$ (Kosten 6)

# Beispiel: Glasfasernetz




- alle Knoten verbunden
- Algorithmus fertig

**Ergebnis:** ein sog. minimaler Spannbaum (minimum spanning tree) des Graphen

# Glasfasernetz: Algorithmus

```
1 Input:  Array K von n Knoten, Kostenfunktion d(i,j)
2 Output: minimaler Spannbaum B
3
4 Glasfasernetz(K, d)
5     B = K[1];
6     while (B nicht Spannbaum)
7         suche billigste Kante aus B;
8         fuege Kante und Knoten zu B hinzu;
```

Komplexität naiver Implementation: $O(n^3)$

$\Rightarrow$ geht besser, (später in der Vorlesung)

# Algorithmenmuster: Brute Force

**Brute Force:**

- erzeuge all in Frage kommenden Lösungskandidaten
- überprüfe für jeden Kandidaten ob es eine zulässige Lösung ist
- ggfs. (bei Optimierungsproblemen) bestimme zulässige Lösung mit minimalen Kosten/maximalem Profit

**Eigenschaften:**

- sehr einfach zu implementieren
- häufig sehr schlechte Laufzeit

Ein Programmierer sollte wissen ob man ein Problem via Brute-Force lösen kann, oder ob die Laufzeit dafür zu schlecht ist.

# Algorithmenmuster: Brute Force

```
1 Input:  Problem P
2 Output: zulaessige Loesung fuer P
3
4 BruteForce(P)
5     S = first(P)
6     while (S != NULL)
7         if (S valid for P)
8             return S
9         S = next(P)
```

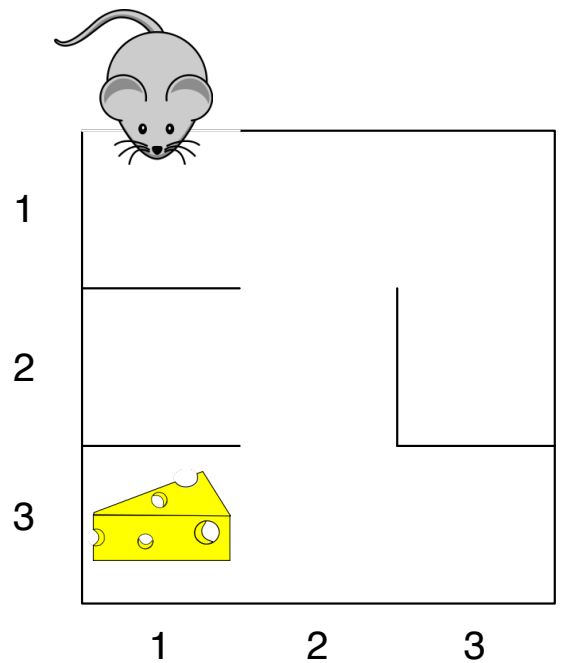
# Algorithmenmuster: Backtracking

**Backtracking:** systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten

# Algorithmenmuster: Backtracking

**Backtracking:** systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten

**Paradebeispiel:** Labyrinth. Wie findet Maus den Käse?

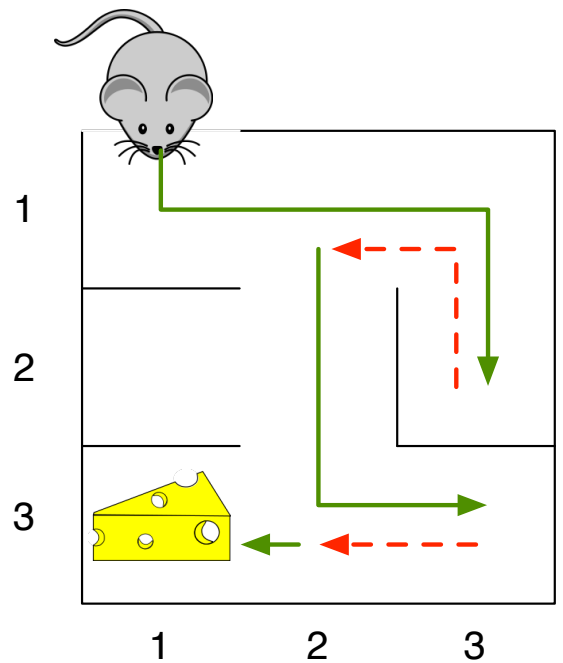


# Backtracking: Labyrinth I

**Problem:** Wie findet Maus den Käse?

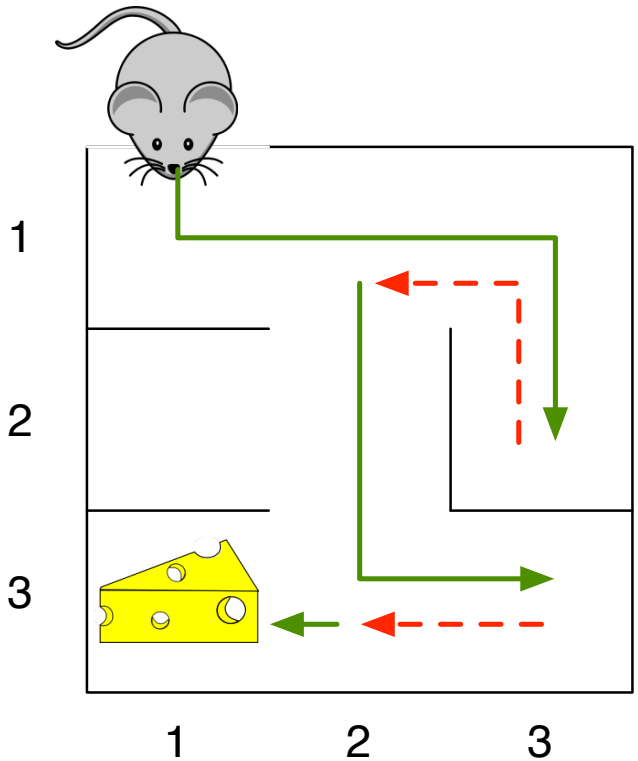
**Lösung:**

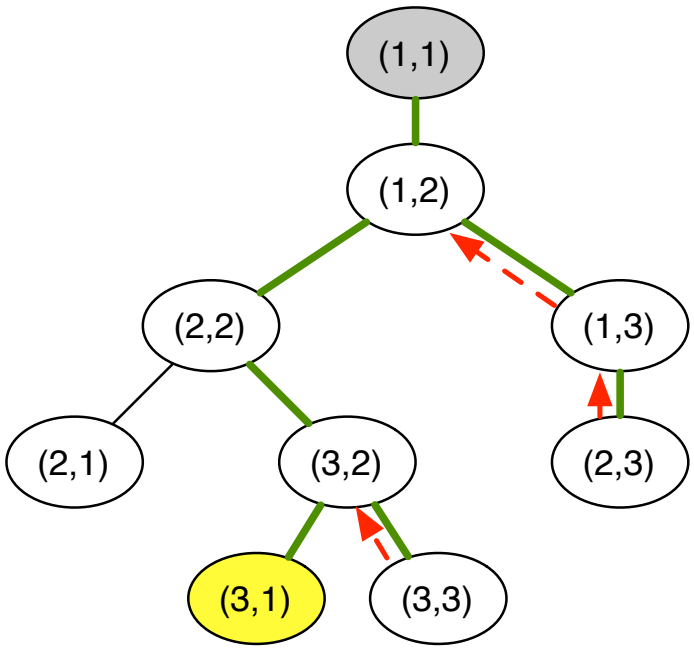
- systematisches Abgehen des Labyrinths
- Zurückgehen falls Sackgasse (daher: **Backtracking**)
  ⇒ “trial and error”



# Backtracking: Labyrinth II

Mögliche Wege repräsentiert als **Baum**:



# Algorithmenmuster: Backtracking

**Voraussetzungen:**

- Lösungs(teil)raum repräsentiert als **Konfiguration** $K$
- $K_0$ ist Startkonfiguration
- jede Konfiguration $K_i$ kann **direkt erweitert** werden
- für jede Konfiguration ist entscheidbar, ob Lösung

```
1 Input: Konfiguration K
2
3 Backtrack(K)
4     if (K Loesung)
5         gib K aus;
6     else
7         foreach (Erweiterung K' von K)
8             Backtrack(K')
```

# Algorithmenmuster: Backtracking

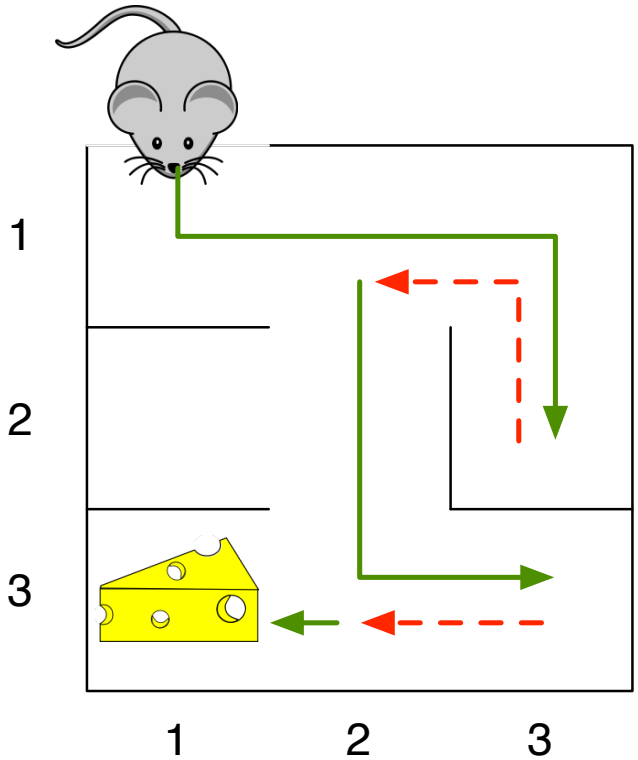
**Voraussetzungen:**

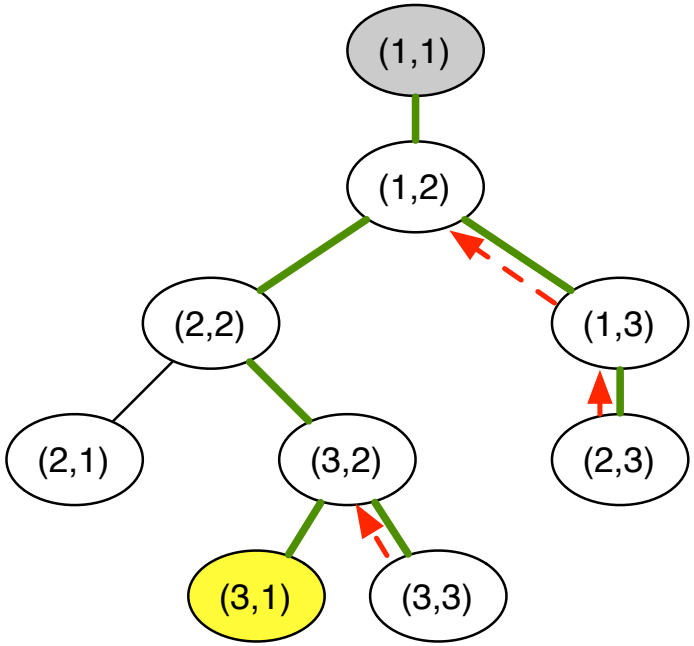
- Lösungs(teil)raum repräsentiert als **Konfiguration** $K$
- $K_0$ ist Startkonfiguration
- jede Konfiguration $K_i$ kann **direkt erweitert** werden
- für jede Konfiguration ist entscheidbar, ob Lösung

```
1 Input: Konfiguration K
2
3 Backtrack(K)
4     if (K Loesung)
5         gib K aus;
6     else
7         foreach (Erweiterung K' von K)
8             Backtrack(K')
```

⇒ initialer Aufruf mittels Backtrack($K_0$)

# Backtracking: Konfigurationen




**Konfiguration** z.B. repräsentiert als **Pfad** im Baum

# Backtracking: Eigenschaften

**Terminierung** von Backtracking:

- nur wenn Lösungsraum **endlich**
- nur wenn sichergestellt dass Konfigurationen **nicht wiederholt getestet** werden

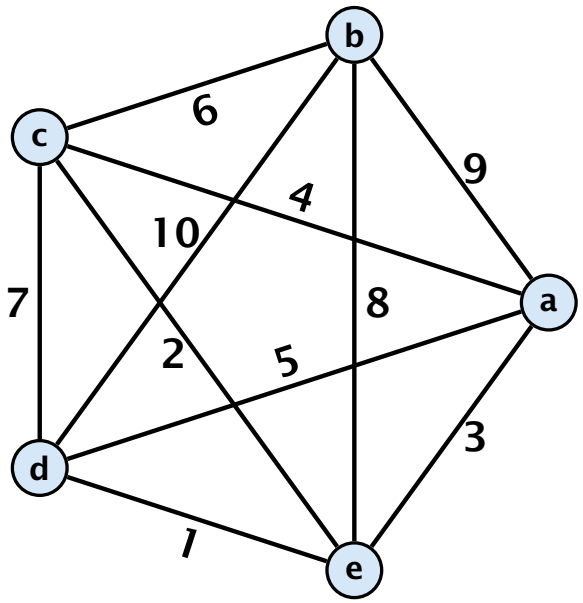
**Komplexität** von Backtracking:

- direkt abhängig von Größe des Lösungsraums
- meist **exponentiell**, also $O(2^n)$, oder schlimmer!
- nur für kleine Probleme wirklich anwendbar

# Backtracking Beispiel: Traveling Salesman

**Traveling Salesman Problem:**

- $n$ Städte
- finde kürzeste Rundreise, die alle Städte exakt einmal besucht (ausser Start- und Zielort (identisch))




⇒ Lösung z.B. mit Algorithmenmuster Backtracking

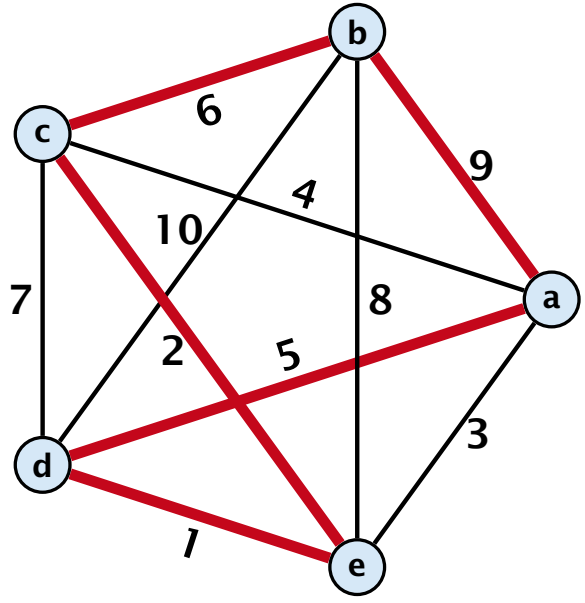
# Traveling Salesman Problem: Algorithmus mit Backtracking

```
1  Input: n Staedte, Rundreise trip
2
3  TSP(trip)
4      if (trip besucht jede Stadt)
5          erweitere trip um Reise zum Standort;
6          gebe trip und Kosten aus;
7      else
8          foreach (bislang unbesuchte Stadt s)
9              trip' = trip erweitert um s
10             TSP(trip');
```

# Traveling Salesman Problem: Beispiel

Bei $n$ Städten mit fixiertem Start-/Zielort gibt es $(n-1)!$ Rundreisen (hier: 5 Städte $\Rightarrow 4! = 24$ Rundreisen).

Laufzeit von **TSP** hier ist $\mathcal{O}((n-1)!)$




hier: kürzeste Rundreise hat Länge 23

- z.B. über Route $a \to b \to c \to e \to d \to a$

# Backtracking Beispiel: Acht-Damen-Problem

**Acht-Damen-Problem:**

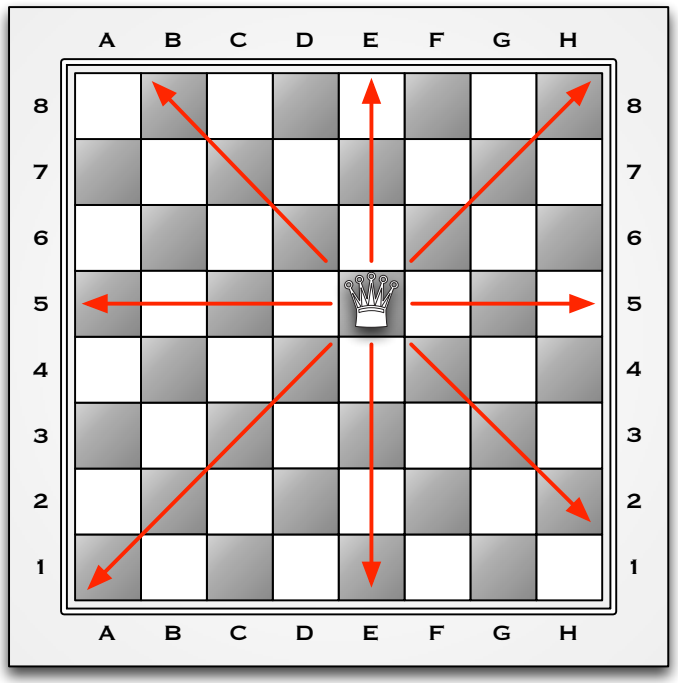
- suche alle Konfigurationen von 8 Damen auf Schachbrett
- so dass keine Dame eine andere bedroht

# Backtracking Beispiel: Acht-Damen-Problem

**Acht-Damen-Problem:**

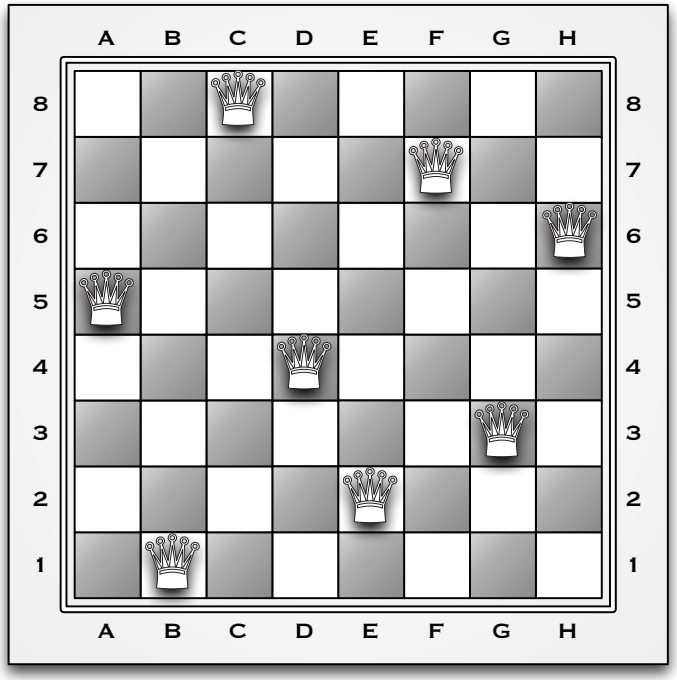
- suche alle Konfigurationen von 8 Damen auf Schachbrett
- so dass keine Dame eine andere bedroht

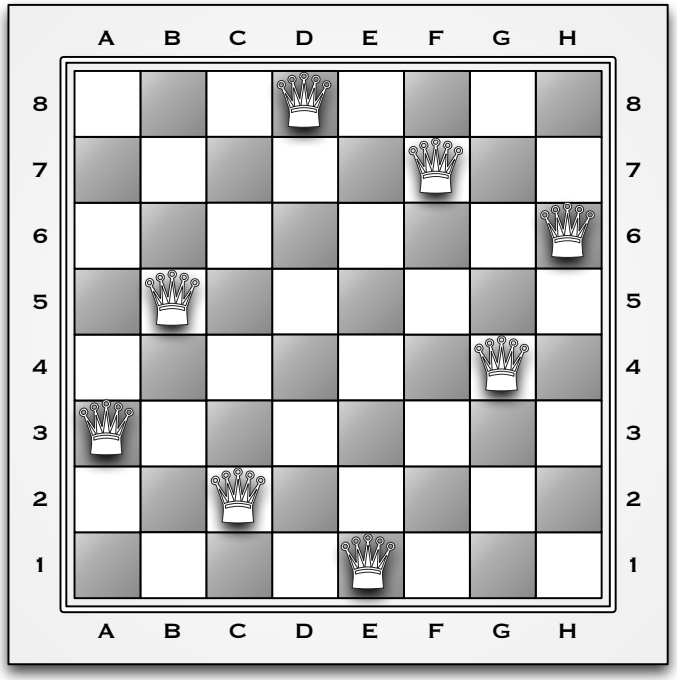
**Dame** auf Schachbrett:



# Acht-Damen-Problem

Zwei der möglichen Lösungen:




**Beobachtung:** jeweils nur eine Dame pro Zeile/Spalte

→ Lösung z.B. mit Algorithmenmuster Backtracking

# Acht-Damen-Problem: Algorithmus mit Backtracking

```
1  Input: Zeilenindex i
2
3  AchtDamen(i)
4      if (i == 9)
5          gib Loesung aus;
6          return;
7      for h=1 to 8
8          if (Feld in Zeile i, Spalte h nicht bedroht)
9              setze Dame auf Feld (i,h);
10             AchtDamen(i+1);
11             entferne Dame von Feld (i,h);
```

# Acht-Damen-Problem

- es gibt **92 Lösungen** für das Acht-Damen-Problem
- das Problem lässt sich auf $n$ Damen auf einem $n \times n$ Schachbrett ausweiten
  - Anzahl Lösungen wächst stark
  - z.B. für $n = 13$ gibt es $73712$ Lösungen
- ähnliche Spiele, wie z.B. **Sudoku**, lassen sich entsprechend lösen

# Dynamisches Programmieren

**Dynamisches Programmieren**

- einsetzbar für Probleme, deren optimale Lösung sich aus optimalen Lösungen von Teilproblemen zusammensetzt (z.B. Rekursion)

**Prinzip:**

- statt Rekursion berechnet man vom kleinsten Teilproblem "aufwärts"
- Zwischenergebnisse werden in **Tabellen** gespeichert

# Beispiel: Fibonacci Zahlen

**Fibonacci Folge**

Die Fibonacci Folge ist eine Folge natürlicher Zahlen $f_1, f_2, f_3, \ldots$, für die gilt

$$f_n = f_{n-1} + f_{n-2} \qquad \text{für } n \geq 3$$

mit Anfangswerten $f_1 = 1$, $f_2 = 1$.

- eingesetzt von Leonardo Fibonacci zur Beschreibung von Wachstum einer Kaninchenpopulation
- Folge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- berechenbar z.B. via Rekursion

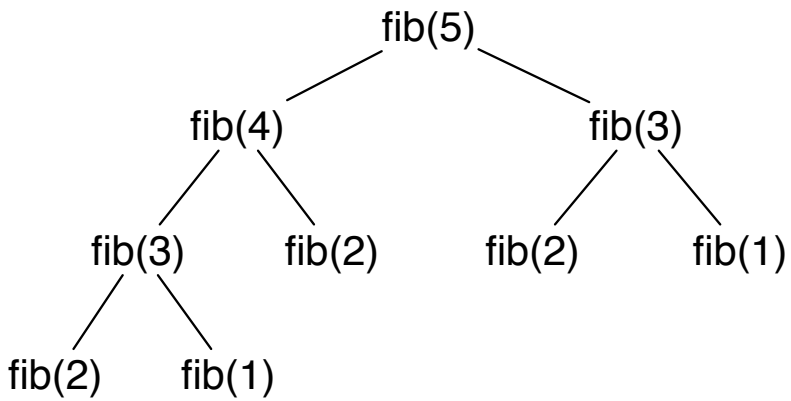
# Beispiel: Fibonacci Funktion

```
Input: Index n der Fibonaccifolge
Output: Wert f_n
fib(n)
   if (n == 1 || n == 2) {
      return 1;
   }
   else {
      // rekursiver Aufruf
      return fib(n-1) + fib(n-2);
   }
```

# Beispiel: Fibonacci Funktion

```
Input: Index n der Fibonaccifolge
Output: Wert f_n
fib(n)
   if (n == 1 || n == 2) {
      return 1;
   }
   else {
      // rekursiver Aufruf
      return fib(n-1) + fib(n-2);
   }
```

Aufrufstruktur für fib(5):



# Fibonacci Funktion: dynamisch programmiert

```
1  Input:  Index n der Fibonaccifolge
2  Output: F_n
3
4  FibDyn(n)
5      fib = new long[n+1];
6      fib[1] = 1;
7      fib[2] = 1;
8      for (k=3; k <= n; k++)
9          fib[k] = fib[k-1] + fib[k-2];
10     res = fib[n];
11     delete fib;
12     return fib[n];
```

- Komplexität dynamisch programmiert: $O(n)$

# Definition: Ungerichteter Graph

**Definition: Ungerichteter Graph**

Ein ungerichteter Graph ist ein Paar $G = (V, E)$ mit

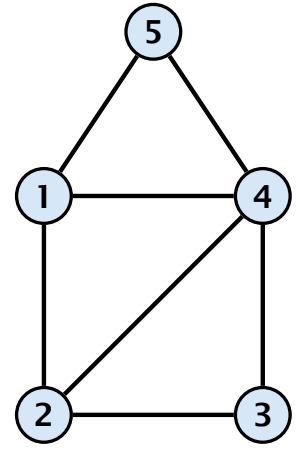
- $V$ endliche Menge der Knoten
- $E \subseteq \{\{u, v\} : u, v \in V\}$ Menge der Kanten

auf Englisch:

- Knoten = vertices
- Kanten = edges

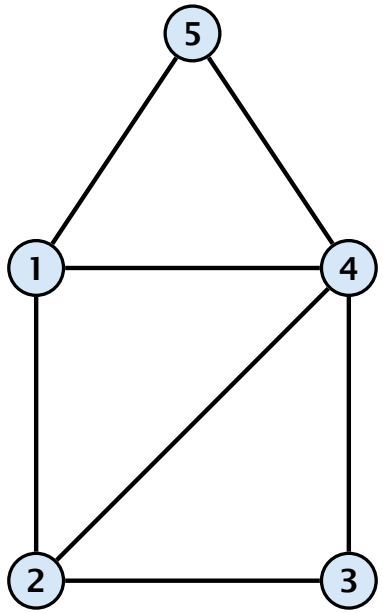
es ist $\{u, v\} = \{v, u\}$, d.h. Richtung der Kante spielt keine Rolle

**Beispiel:**




Manchmal erlaubt man auch Schleifen (self-loops); dann muss man $E$ als Multimenge modellieren.

# Ungerichteter Graph: Beispiel




- Graph $G_u = (V_u, E_u)$
- Knoten $V_u = \{1, 2, 3, 4, 5\}$
- Kanten $E_u = \{\{1, 2\}, \{1, 4\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$

# Definition: Gerichteter Graph

**Definition: Gerichteter Graph**

Ein gerichteter Graph ist ein Paar $G = (V, E)$ mit

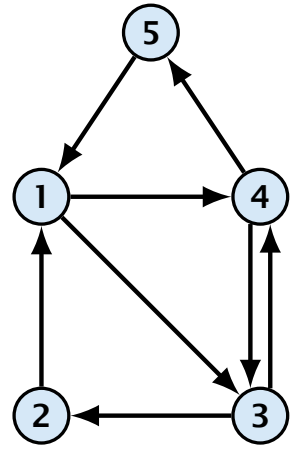
- $V$ endliche Menge der Knoten
- $E \subseteq V \times V$ Menge der Kanten

$E \subseteq \{(u, v) : u, v \in V\} = V \times V$

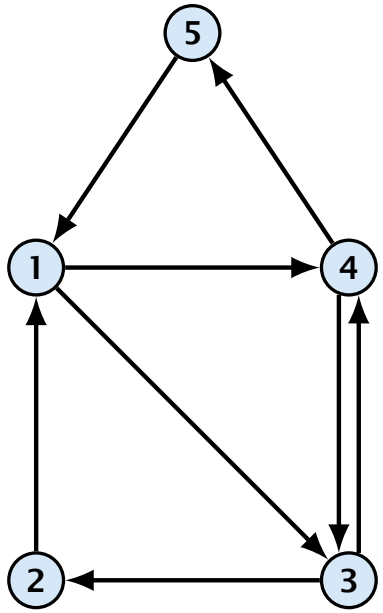
es ist $(u, v) \neq (v, u)$, d.h. Richtung der Kante spielt eine Rolle

hier sind Schleifen möglich, d.h. Kanten der Form $(u, u)$ für $u \in V$

**Beispiel:**



# Gerichteter Graph: Beispiel




- Graph $G_g = (V_g, E_g)$
- Knoten $V_g = \{1, 2, 3, 4, 5\}$
- Kanten
  $E_g = \{(1,3), (1,4), (2,1), (3,2), (3,4), (4,3), (4,5), (5,1)\}$

# Definition: Gewichteter Graph

**Definition: Gewichteter Graph**

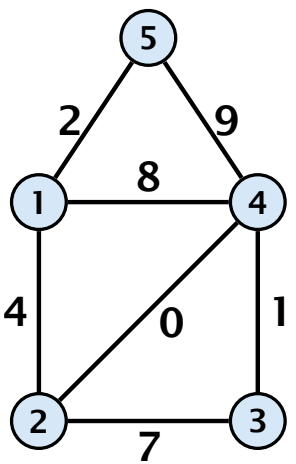
Ein gewichteter Graph ist ein Graph $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to \mathbb{R}$.

der Graph $G$ kann gerichtet oder ungerichtet sein

je nach Anwendung kann ein verschiedener Wertebereich für die Funktion $w$ gewählt werden

- z.B. $\mathbb{R}$ oder $\mathbb{N}_0$

**Beispiel:**



# Eigenschaften von Graphen I

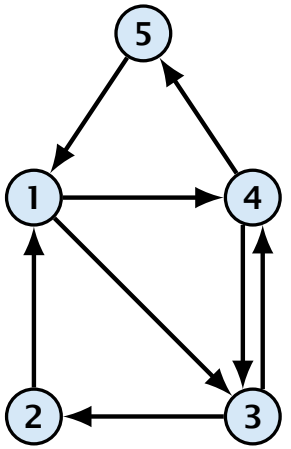
Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

- Ist $(u, v) \in E$ bzw. $\{u, v\} \in E$ für $u, v \in V$, so heißt $v$ adjazent zu $u$.
- Ein Knoten $v$ und eine Kante $e = (x, v)$ or $e = \{x, v\}$ heißen inzident.
- Sei $G$ gerichteter Graph:
  - die Anzahl der eintretenden Kanten in $v$ heißt Eingangsgrad von $v$, $\text{indeg}(v) = \left|\{v' : (v', v) \in E\}\right|$
  - die Anzahl der austretetenden Kanten heißt Ausgangsgrad von $v$, $\text{outdeg}(v) = \left|\{v' : (v, v') \in E\}\right|$
- Sei $G$ ungerichteter Graph:
  - die Anzahl der eintretenden bzw. austretenden Kanten von $v$ heißt Grad (englisch: degree) von $v$ oder kurz $\deg(v)$.

# Eigenschaften von Graphen: Beispiel

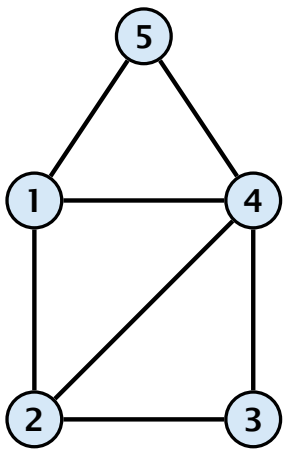
**Beispiel gerichteter Graph:**

- Knoten 1 ist adjazent zu Knoten 2
- Knoten 2 ist adjazent zu Knoten 3
- $\text{outdeg}(4) = 2$, $\text{indeg}(4) = 2$
- $\text{indeg}(2) = 1$, $\text{outdeg}(2) = 1$




**Beispiel ungerichteter Graph:**

- Knoten 4 ist adjazent zu Knoten 2
- Knoten 2 ist adjazent zu Knoten 4
- $\deg(2) = 3$
- $\deg(5) = 2$



# Eigenschaften von Graphen II

Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

- Seien $v, v' \in V$. Ein Pfad von $v$ nach $v'$ ist eine Folge von Knoten $(v_0, v_1, \ldots, v_k) \subset V$ mit
  - $v_0 = v$, $v_k = v'$
  - $(v_i, v_{i+1}) \in E$ bzw. $\{v_i, v_{i+1}\} \in E$ für $i = 0, \ldots, k-1$

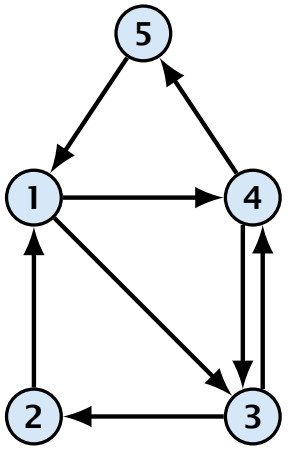
  $k$ heißt Länge des Pfades.
- Ein Pfad heißt einfach, falls alle Knoten des Pfades paarweise verschieden sind.

Gibt es einen Pfad von $u$ nach $v$, so heißt $v$ erreichbar von $u$.

# Eigenschaften von Graphen II: Beispiel

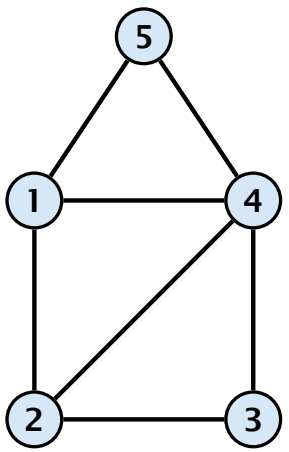
**Beispiel gerichteter Graph:**

- $(2, 1, 3)$ ist ein einfacher Pfad der Länge 2
- $(1, 4, 5, 1)$ ist ein Pfad der Länge 3, aber nicht einfach
- $5$ ist erreichbar von $1$




**Beispiel ungerichteter Graph:**

- $(5, 2, 4, 3, 2)$ ist ein Pfad der Länge 4, aber nicht einfach
- $(1, 2, 3, 4)$ ist ein einfacher Pfad der Länge 3
- $3$ ist erreichbar von $1$



# Eigenschaften von Graphen III

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

- Ein Pfad $(v_0, \ldots, v_k)$ heißt Zyklus, falls $v_0 = v_k$.
- Ein Zyklus $(v_0, \ldots, v_k)$ heißt Kreis, falls $v_1, \ldots, v_k$ paarweise verschieden sind.

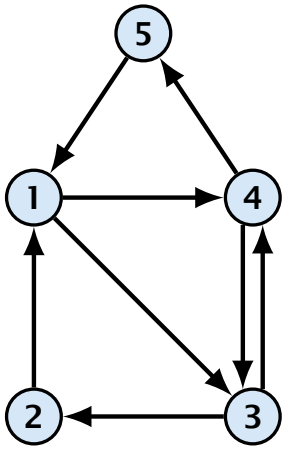
Zyklen der Länge 1 oder 2 heißen trivial und werden häufig nicht betrachtet.

Ein Graph ohne Zyklen heißt azyklisch.

# Eigenschaften von Graphen III: Beispiel

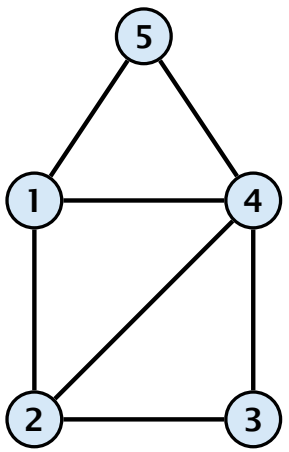
**Beispiel gerichteter Graph:**

- $(2, 1, 3, 4, 3, 2)$ ist ein Zyklus, aber nicht einfach
- $(2, 1, 3, 2)$ ist ein einfacher Zyklus




**Beispiel ungerichteter Graph:**

- $(2, 1, 3, 2)$ ist ein Zyklus
- $(1, 5, 4, 1)$ ist ein Zyklus



# Eigenschaften von Graphen IV

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

- $G$ heißt stark zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine starke Zusammenhangskomponente von $G$ ist ein maximaler zusammenhängender Untergraph von $G$.
  - alternativ: Äquivalenzklassen der Knoten bezüglich Relation "gegenseitig erreichbar"

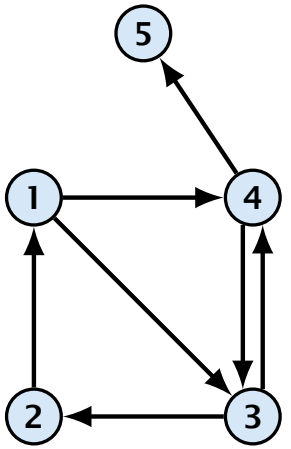
Sei $G = (V, E)$ ungerichteter Graph.

- $G$ heißt zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine Zusammenhangskomponente von $G$ ist ein maximaler zusammenhängender Untergraph von $G$.
  - alternativ: Äquivalenzklassen der Knoten bezüglich Relation "erreichbar von"

# Eigenschaften von Graphen IV: Beispiel

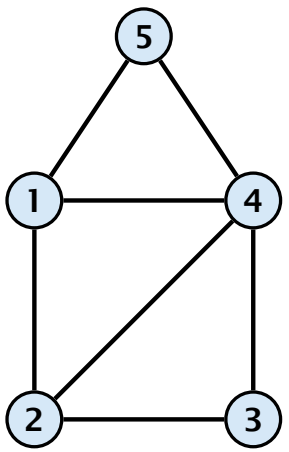
**Beispiel gerichteter Graph:**

- Graph ist nicht stark zusammenhängend (z.B. $3$ nicht erreichbar von $5$)
- starke Zusammenhangskomponenten: $\{1,2,3,4\}$ und $\{5\}$




**Beispiel ungerichteter Graph:**

- Graph ist zusammenhängend
- nur eine Zusammenhangskomponente: $\{1,2,3,4,5\}$



# Darstellung von Graphen: Adjazenzmatrizen

**Adjazenzmatrix**

Sei $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$. Die Adjazenzmatrix von $G$ speichert die vorhandenen Kanten in einer $n \times n$ Matrix $A \in \mathbb{R}^{n \times n}$ mit

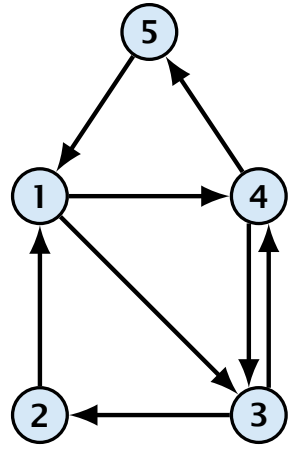
- $A(i, j) = 1$ falls Kante von Knoten $v_i$ zu $v_j$ existiert
- $A(i, j) = 0$ falls keine Kante von Knoten $v_i$ zu $v_j$ existiert

für $i, j \in \{1, \ldots, n\}$.

# Eigenschaften von Graphen: Adjazenzmatrizen

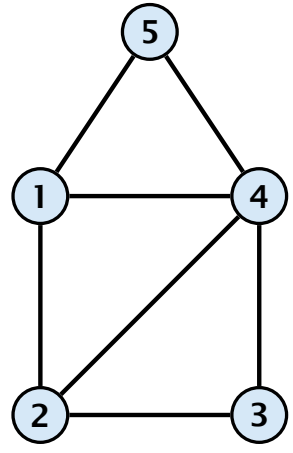
**Beispiel gerichteter Graph:**

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




**Beispiel ungerichteter Graph:**

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



# Adjazenzmatrizen: Eigenschaften

Eigenschaften von Adjazenzmatrizen zu Graph $G = (V, E)$

- sinnvoll wenn der Graph nahezu vollständig ist (d.h. fast alle möglichen Kanten tatsächlich in $E$ liegen)
- Speicherkomplexität: $O(|V|^2)$
- bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch
- bei gewichteten Graphen kann man statt der 1 in der Matrix das Gewicht der Kante eintragen

# Darstellung von Graphen: Adjazenzlisten

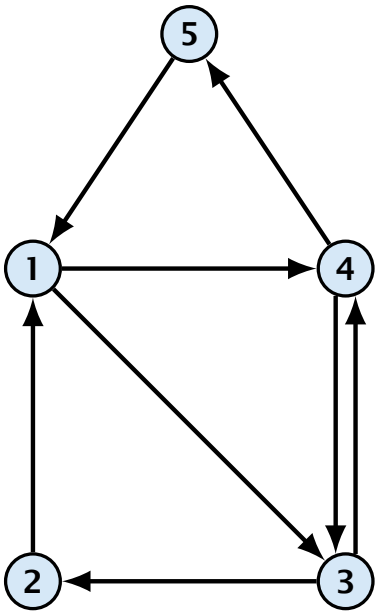
**Adjazenzliste**

Sei $G = (V, E)$ gerichteter Graph. Eine Adjazenzliste von $G$ sind $|V| + 1$ verkettete Listen, so daß

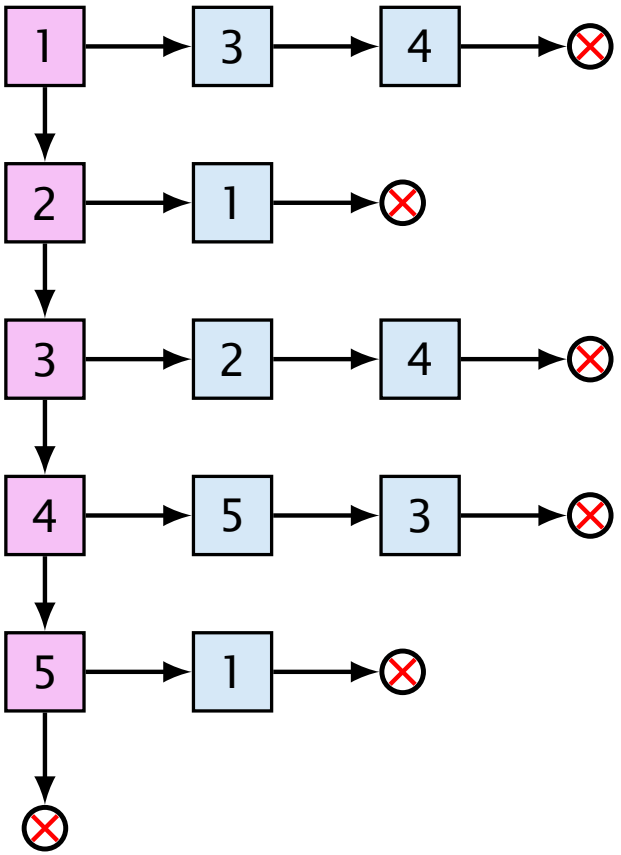
- die erste Liste alle Knoten enthält
- für jeden Knoten $v$ eine Liste angelegt wird mit allen Knoten, die durch eine von $v$ austretende Kante zu erreichen sind

# Adjazenzliste: Beispiel

## Graph




## Adjazenzliste



# Adjazenzliste: Eigenschaften

Eigenschaften von Adjazenzlisten zu Graph $G = (V, E)$

- sinnvoll bei dünn besetzten Graphen mit wenigen Kanten
- Speicherkomplexität: $O(|V| + |E|)$
- bei **ungerichteten** Graphen gleiches Verfahren
  - allerdings muß jede Kante zweimal gespeichert werden
- bei **gewichteten** Graphen kann man die Gewichte mit in den verketteten Listen der jeweiligen Knoten speichern

# Komplexität der Darstellungen

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

| Operation | Adjazenzmatrix | Adjazenzliste |
|---|---|---|
| Kante einfügen | $\mathcal{O}(1)$ | $\mathcal{O}(\|V\|)$ |
| Kante löschen | $\mathcal{O}(1)$ | $\mathcal{O}(\|V\|)$ |
| Knoten einfügen | $\mathcal{O}(\|V\|^2)$ | $\mathcal{O}(1)$ |
| Knoten löschen | $\mathcal{O}(\|V\|^2)$ | $\mathcal{O}(\|V\| + \deg(v))$ |

- falls Größe im Vorhinein bekannt, kann Knoten löschen/einfügen bei Adjanzenzmatrix effizienter implementiert werden
- Löschen von Knoten ist immer aufwendig, da auch alle Kanten von/zu diesem Knoten gelöscht werden müssen

# Algorithmen auf Graphen

**Ausblick** auf Algorithmen auf Graphen:

- Traversierung (Durchlaufen) von allen Knoten
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- kürzester Pfad zwischen Knoten in Graphen
- minimaler Spannbaum (minimum spanning tree, MST)

# Bäume

Bäume sind alltägliches Mittel zur Strukturierung:

- Stammbaum
- Hierarchie in Unternehmen
- Systematik in der Biologie
- etc.

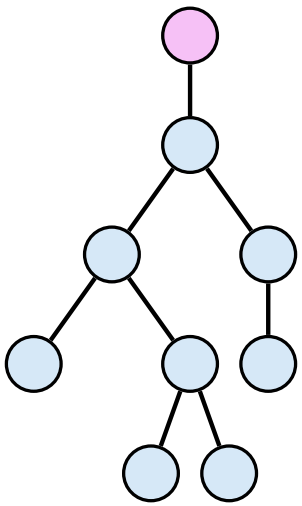
In Informatik:

- Bäume sind spezielle Graphen
- Wurzel oben!

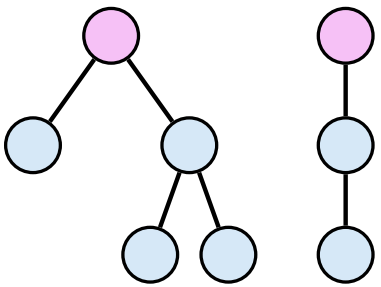
# Definition Wald/Baum

**Definition: Wald und Baum**

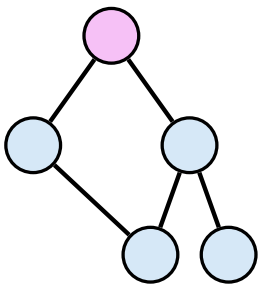
- Ein azyklischer ungerichteter Graph heißt auch Wald.
- Ein zusammenhängender, azyklischer ungerichteter Graph heißt auch Baum.



Baum



Wald



kein Baum

# Eigenschaften von Bäumen

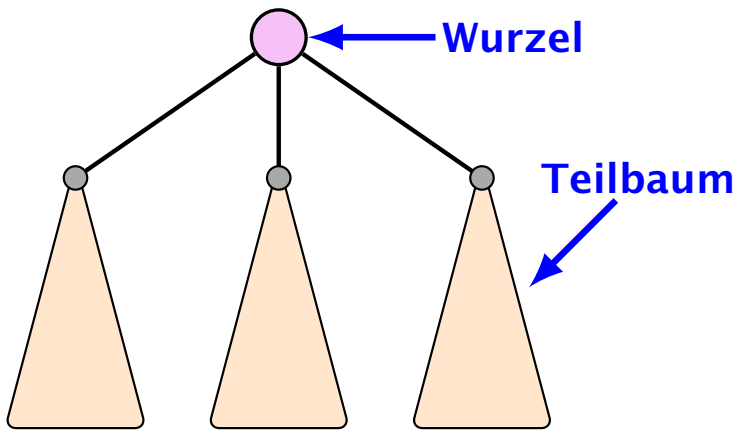
Sei $G = (V, E)$ ein Baum.

- jedes Paar von Knoten $u, v \in V$ ist durch einen einzigen Pfad verbunden
- $G$ ist zusammenhängend, aber wenn eine Kante aus $E$ **entfernt** wird, ist $G$ nicht mehr zusammenhängend
- $G$ ist azyklisch, aber wenn eine Kante zu $E$ **hinzugefügt** wird, ist $G$ nicht mehr azyklisch
- es gilt $|E| = |V| - 1$

# Wurzel von Bäumen

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

- genau ein Knoten $w \in V$ wird als Wurzel ausgezeichnet
- entfernt man $w$ erhält man einen Wald von Teilbäumen

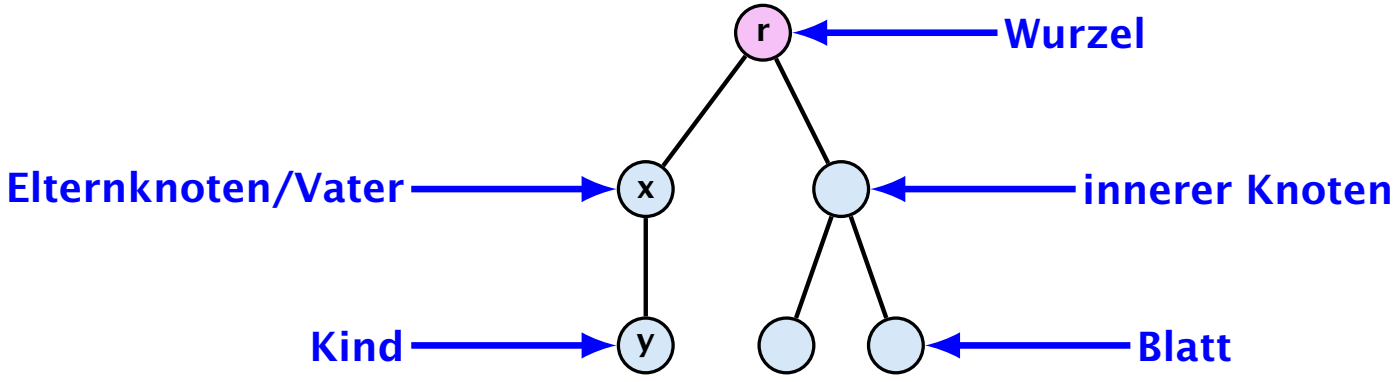



**Hinweis:** manchmal wird zwischen “freiem” und “gewurzeltem” Baum unterschieden!

# Weitere Begriffe bei Bäumen I

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

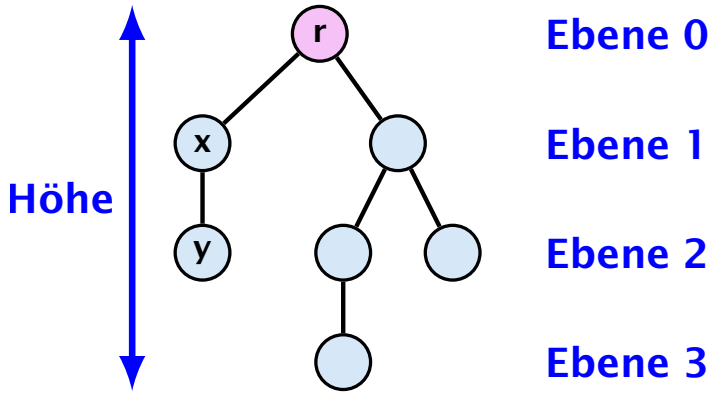
- jeder Knoten $v \in V$ mit $v \neq w$ ist mit genau einer Kante mit seinem Elternknoten $x \in V$ (oder: direkter Vorgänger) verbunden
- $v$ wird dann als Kind (oder: direkter Nachfolger) von $x \in V$ bezeichnet
- ein Knoten ohne Kinder heißt Blatt, alle anderen Knoten heißen innere Knoten



# Weitere Begriffe bei Bäumen II

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

- Anzahl der Kinder von Knoten $x \in V$ heißt auch Grad von $x$. (Achtung: Grad in Graph $G$ ist anders definiert!)
- Länge des Pfades von Wurzel $w$ zu Knoten $x \in V$ heißt Tiefe von $x$
- alle Knoten gleicher Tiefe bilden eine Ebene des Baumes $G$
- maximale Tiefe eines Knotens heißt Höhe des Baumes. (manchmal ist Höhe auch als Anzahl der Ebenen definiert)

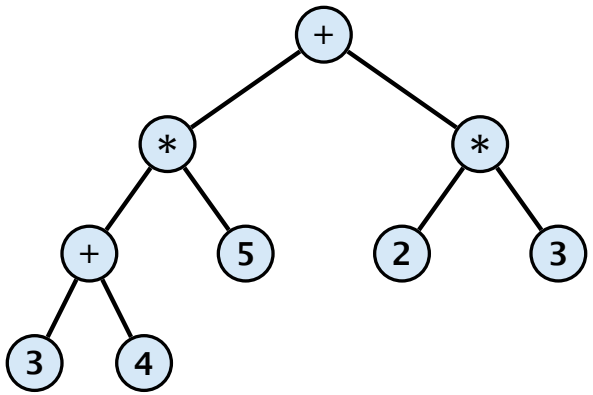


# Bäume: Beispiele

**arithmetischer Ausdruck**

$$(3 + 4) * 5 + 2 * 3$$

repräsentiert als Baum:




**hierarchisches Dateisystem**

- Windows z.B. “C:\”
- Unix “/”

**Suchbaum** → später in der Vorlesung

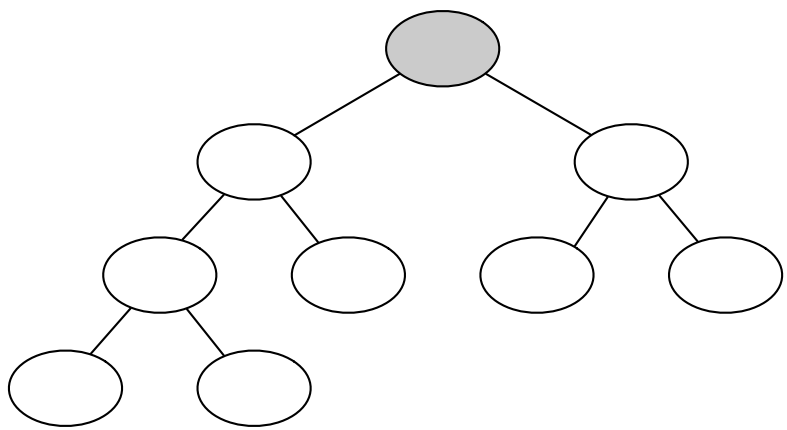
# Besondere Bäume

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

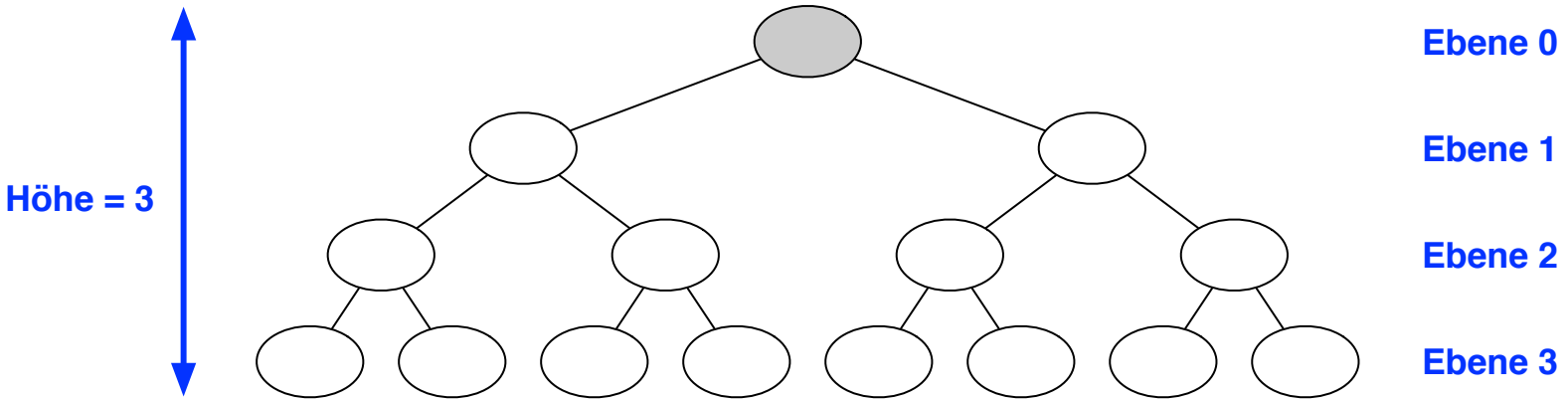
- sind die Kinder jedes Knotens in bestimmter Reihenfolge angeordnet, heißt $G$ geordneter Baum
- ist die Anzahl $n$ der Kinder jedes Knotens vorgegeben, heißt $G$ $n$-ärer Baum

Wichtiger Spezialfall:

- ist $G$ geordnet und hat jeder Knoten maximal zwei Kinder, heißt $G$ Binärbaum



# Beispiel: Binärbaum




Binärbaum mit Höhe 3, 8 Blättern und 7 inneren Knoten.

- Binärbaum heißt vollständig, wenn jede Ebene die maximale Anzahl an Knoten enthält
- ein vollständiger Binärbaum der Höhe $k$ hat $2^{k+1} - 1$ Knoten, davon $2^k$ Blätter
- Beweis per Induktion

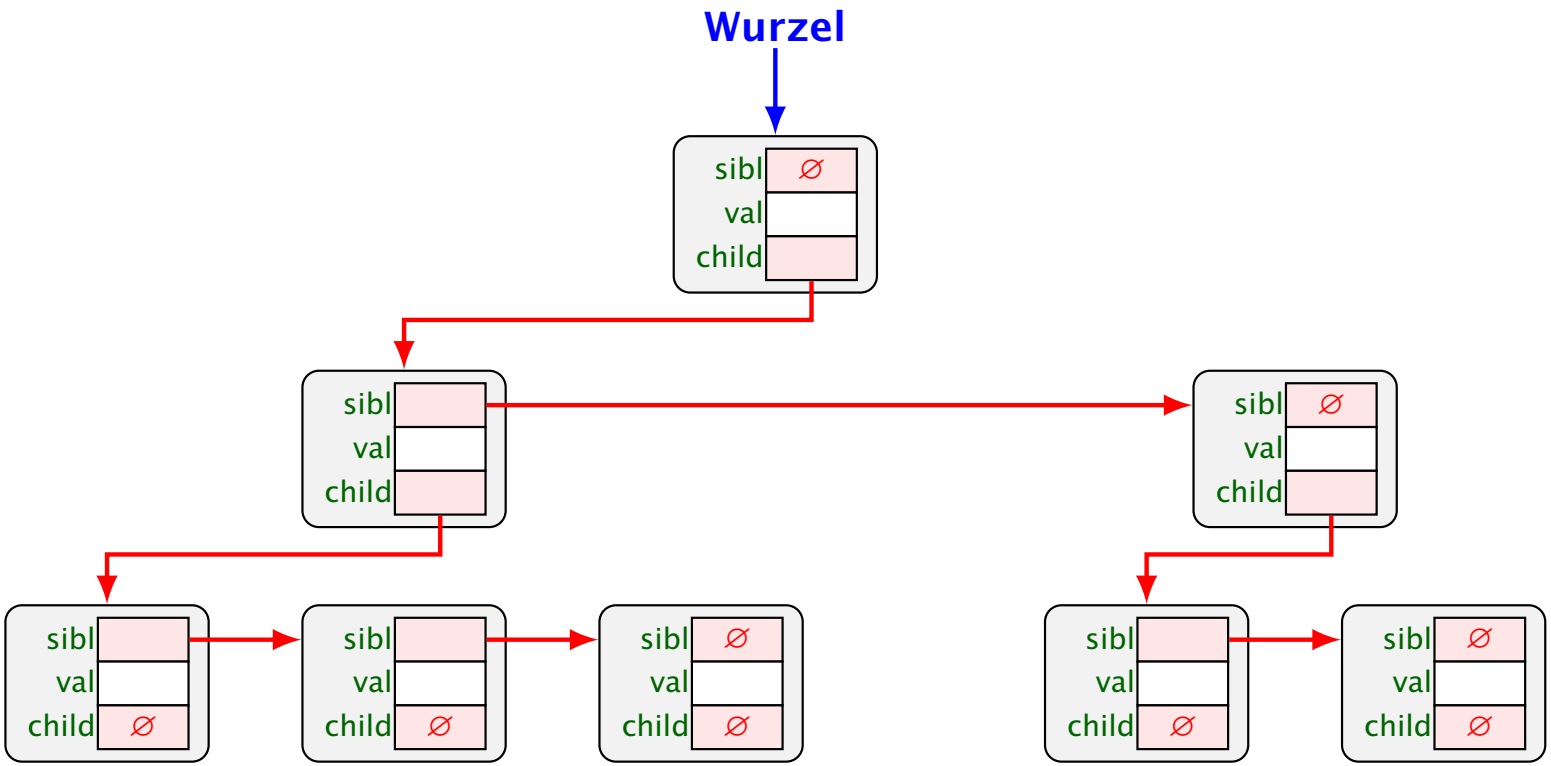
# Darstellung von Bäumen: als Graph

Bäume sind Graphen → Darstellung als

- Adjazenzmatrix
- Adjazenzliste

⇒ leider meist nicht effizient (sowohl Laufzeit als auch Speicher)

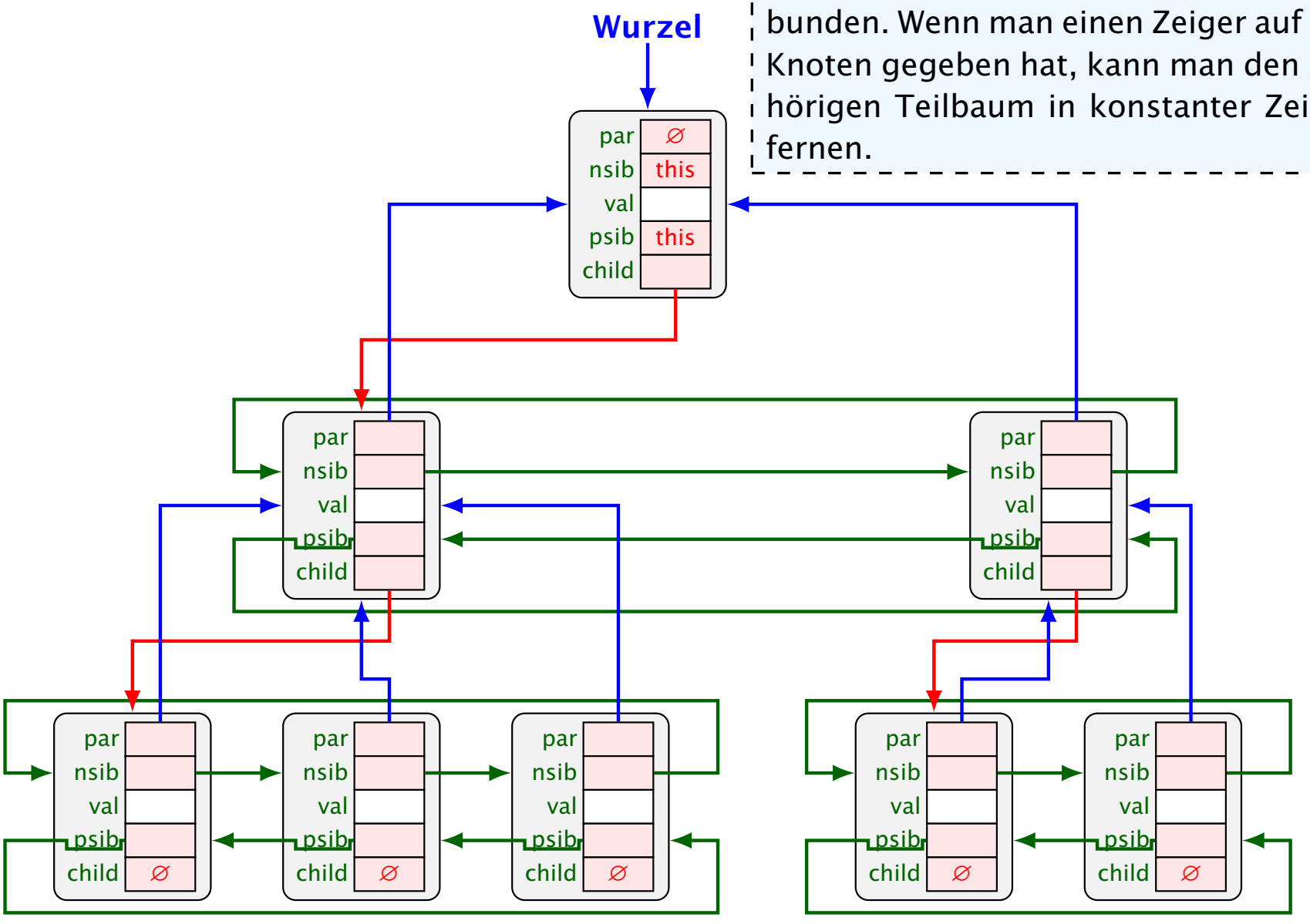
# Darstellung von Bäumen: verkettete Liste



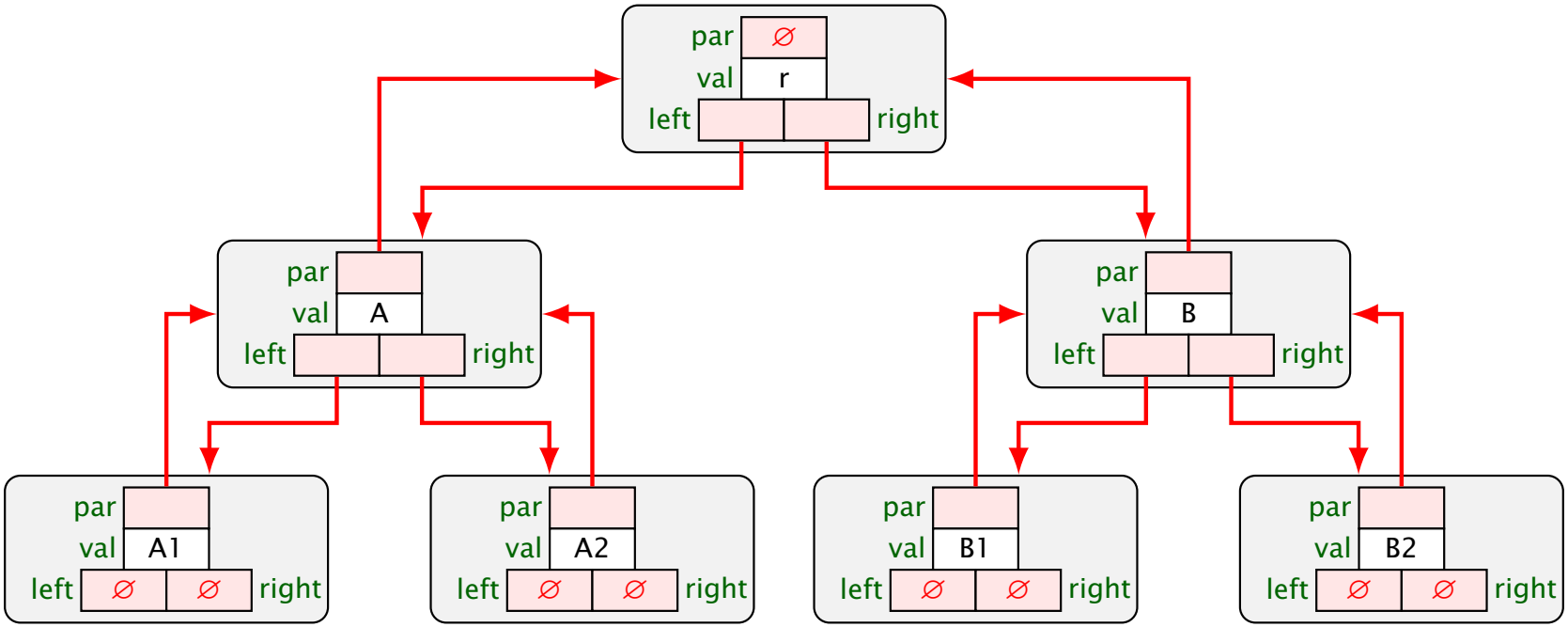

**Nachteil**: nur Navigation nach unten möglich.

# Darstellung von Bäumen: doppelt verkettet

Geschwister sind über zyklische Liste verbunden. Wenn man einen Zeiger auf einen Knoten gegeben hat, kann man den zugehörigen Teilbaum in konstanter Zeit entfernen.



# Darstellung von Binärbäumen



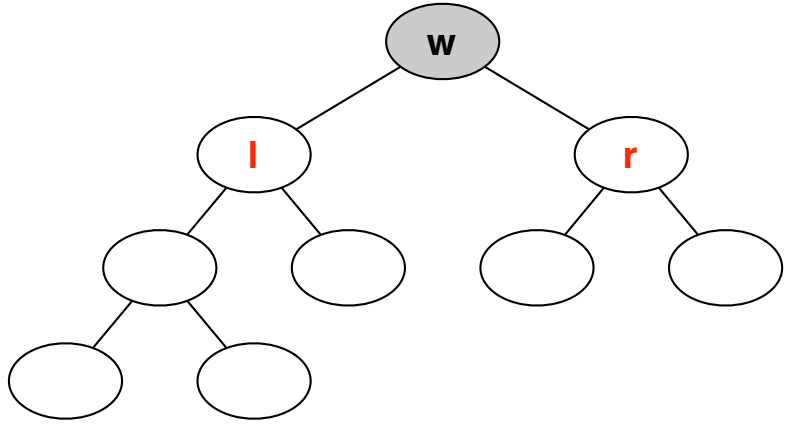

Bei Binärbäumen hat man üblicherweise für beide Nachfolger eine explizite Referenz.

# Traversierung von Binärbäumen

Sei $G = (V, E)$ Binärbaum.

In **welcher Reihenfolge** durchläuft man $G$?

- **Wurzel** zuerst
- danach linker oder rechter Kind-Knoten $l$ bzw. $r$?
- falls $l$: danach Kindknoten von $l$ oder zuerst $r$?
- falls $r$: danach Kindknoten von $r$ oder zuerst $l$?



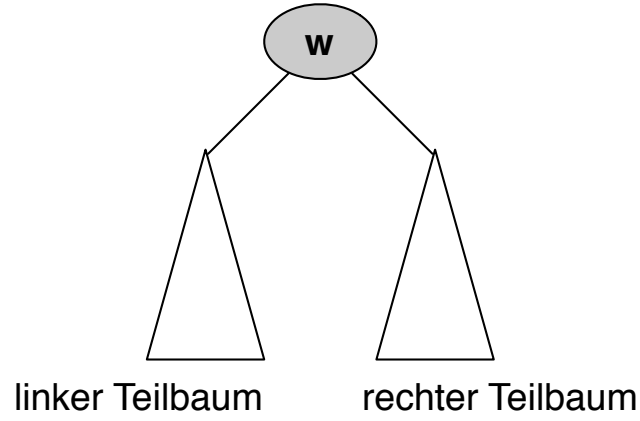

⇒ falls zuerst in die Tiefe: Depth-first search (DFS)

⇒ falls zuerst in die Breite: Breadth-first search (BFS)

# DFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

Tiefensuche (Depth-first search, DFS) gibt es in 3 Varianten:




1. Pre-order Reihenfolge
   - besuche Wurzel
   - durchlaufe linken Teilbaum
   - durchlaufe rechten Teilbaum
2. In-order Reihenfolge
   - durchlaufe linken Teilbaum
   - besuche Wurzel
   - durchlaufe rechten Teilbaum
3. Post-order Reihenfolge
   - durchlaufe linken Teilbaum
   - durchlaufe rechten Teilbaum
   - besuche Wurzel

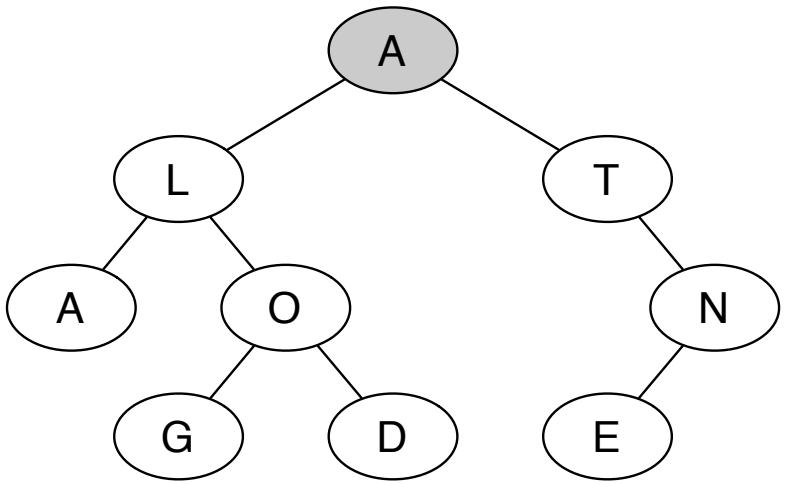
# Pre-order Traversierung

Pre-order Reihenfolge:

- besuche Wurzel
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

**Beispiel:**

A, L, A, O, G, D, T, N, E



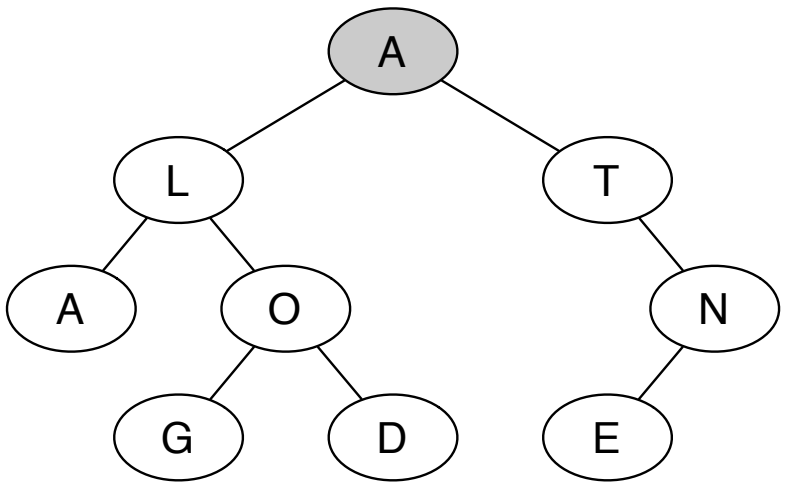
# In-order Traversierung

Inorder Reihenfolge:

- durchlaufe linken Teilbaum
- besuche Wurzel
- durchlaufe rechten Teilbaum

**Beispiel:**

A, L, G, O, D, A, T, E, N



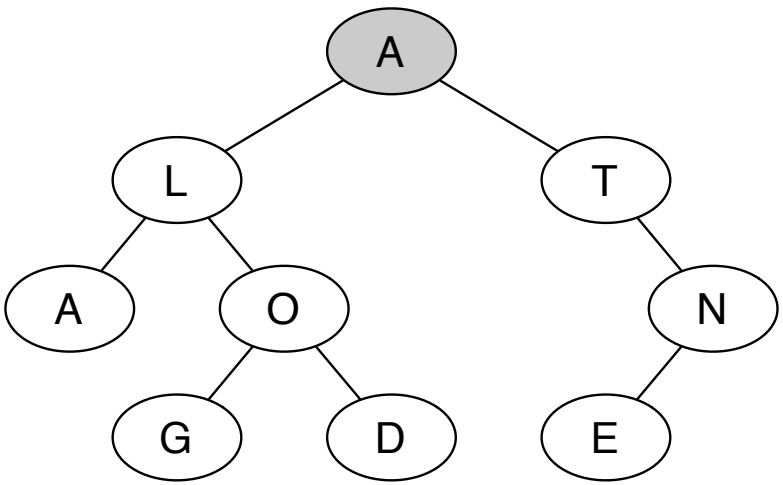
# Post-order Traversierung

Postorder Reihenfolge:

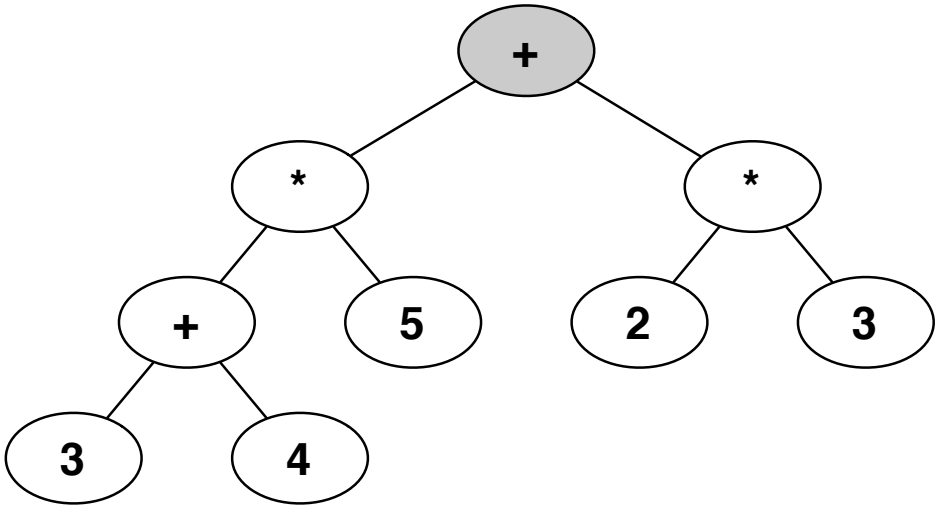
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

**Beispiel:**

A, G, D, O, L, E, N, T, A



# Beispiel: Arithmetischer Term

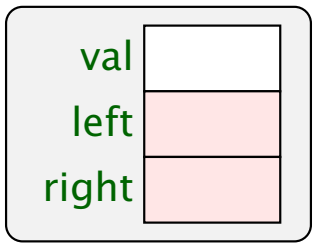



Traversierung:

- **Pre-order:** $+ * + 3\ 4\ 5 * 2\ 3$
- **In-order:** $3 + 4 * 5 + 2 * 3$
- **Post-order:** $3\ 4 + 5 * 2\ 3 * +$

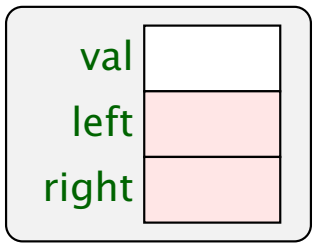
# Implementierung DFS

```
1 preorder(TreeNode* v)
2     if (v == NULL)
3         return
4     print(v->val);
5     preorder(v->left);
6     preorder(v->right);
```



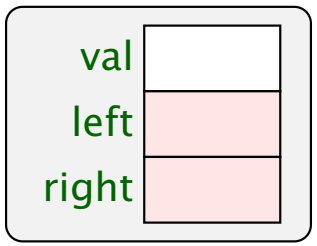
# Implementierung DFS

```
1 inorder(TreeNode* v)
2     if (v == NULL)
3         return
4     preorder(v->left);
5     print(v->val);
6     preorder(v->right);
```



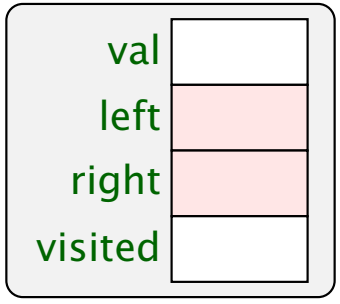
# Implementierung DFS

```
1 postorder(TreeNode* v)
2     if (v == NULL)
3         return
4     postorder(v->left);
5     postorder(v->right);
6     print(v->val);
```



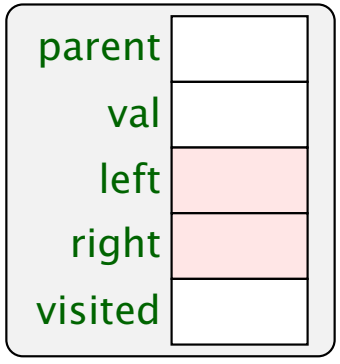
# Implementierung DFS mit Stack

```
1  postorder(TreeNode* v)
2      stack.push(v);
3      while (!stack.empty())
4          v = stack.top();
5          if (v->left) && !v->left->visited)
6              stack.push(v->left);
7          else if (v->right && !v->right->visited)
8              stack.push(v->right);
9          else
10             print(v->val);
11             v->visited = true;
12             stack.pop();
```



# Implementierung DFS ohne Stack

```
1  postorder(TreeNode* v)
2      TreeNode* k = v;
3      while (true)
4          if (k->left && !k->left->visited)
5              k = k->left;
6          else if (k->right && !k->right->visited)
7              k = k->right;
8          else
9              print(k->val);
10             k->visited = true;
11             if (k == v) break;
12             else k = k->parent;
```

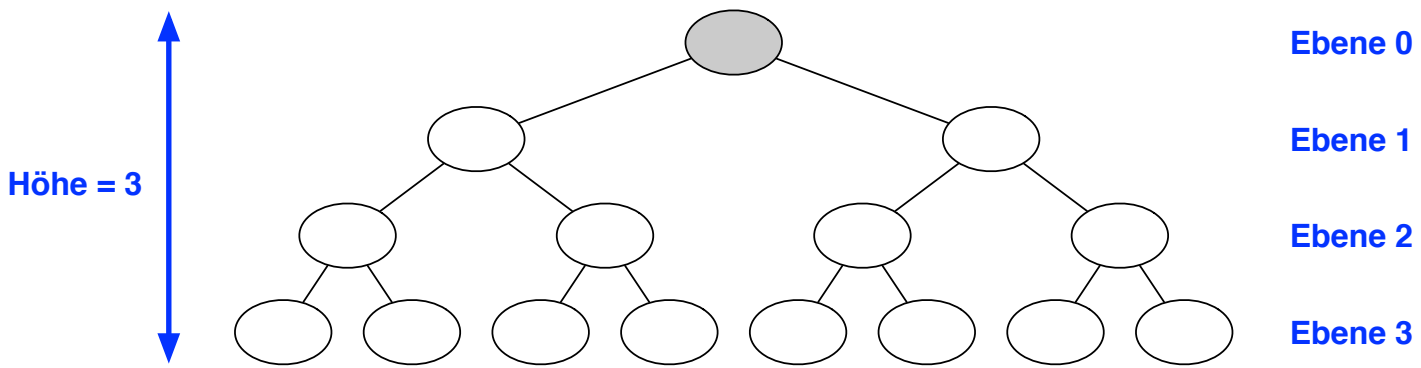


# BFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

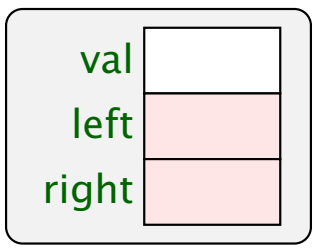
Breitensuche (Breadth-first search, BFS):

- besuche Wurzel
- für alle Ebenen von 1 bis Höhe
  - besuche alle Knoten aktueller Ebene



# Implementierung BFS Traversierung

```
1 bfs(TreeNode* v)
2     queue.enqueue(v);
3     while (!queue.empty())
4         v = queue.dequeue();
5         print(v.val);
6         if (v->left)
7             queue.enqueue(v->left);
8         if (v->right)
9             queue.dequeue(v->right);
```

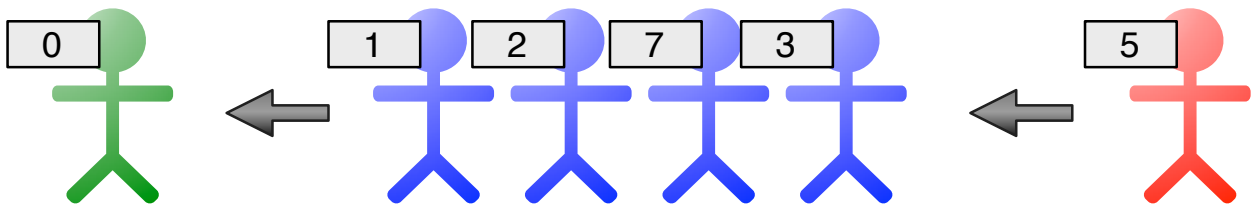


# Definition Priority Queue

**Definition Priority Queue**

Eine Priority Queue ist ein abstrakter Datentyp. Sie beschreibt einen **Queue-artigen** Datentyp für eine Menge von Elementen mit **zugeordnetem Schlüssel** und unterstützt die Operationen

- Einfügen von Element mit Schlüssel in die Queue,
- Entfernen von Element mit minimalem Schlüssel aus der Queue,
- Ansehen des Elementes mit minimalem Schlüssel in der Queue.




- entsprechend gibt es auch eine Priority Queue mit Entfernen/Ansehen von Elememt mit **maximalem Schlüssel**

# Definition Priority Queue (abstrakter)

Priority Queue P ist ein abstrakter Datentyp mit Operationen

- **insert(P, x)** wobei x ein Element
- **extractMin(P)** liefert ein Element
- **minimum(P)** liefert ein Element
- **isEmpty(P)** liefert `true` or `false`
- **initialize** liefert eine Priority Queue Instanz

und mit Bedingungen

- **isEmpty(initialize()) == `true`**
- **isEmpty(insert(P, x)) == `false`**
- **minimum(initialize())** ist nicht erlaubt (Fehler)
- **extractMin(initialize())** ist nicht erlaubt (Fehler)

*(Fortsetzung nächste Folie)*

# Definition Priority Queue (abstrakter)

Fortsetzung Bedingungen Priority Queue P:

- **minimum(insert(P, x))** liefert zurück
  - falls `P == initialize()`, dann `x`
  - sonst: `min(x, minimum(P))`

- **extractMin(insert(P, x))**
  - falls `x == minimum(insert(P, x))`, dann liefert es `x` zurück und hinterlässt `P` im Originalzustand
  - sonst liefert es `extractMin(P)` zurück und hinterlässt `P` im Zustand `insert(extractMin(P), x)`

(entsprechend für die Priority Queue mit maximalem Schlüssel)

# Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k)**:
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

# Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k):**
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

# Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

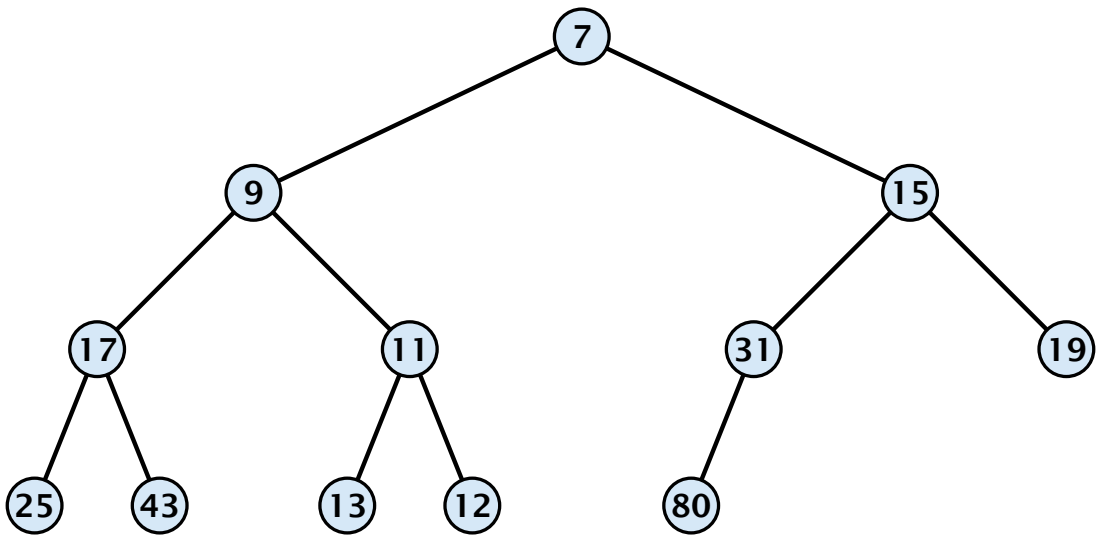
- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k):**
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

# Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

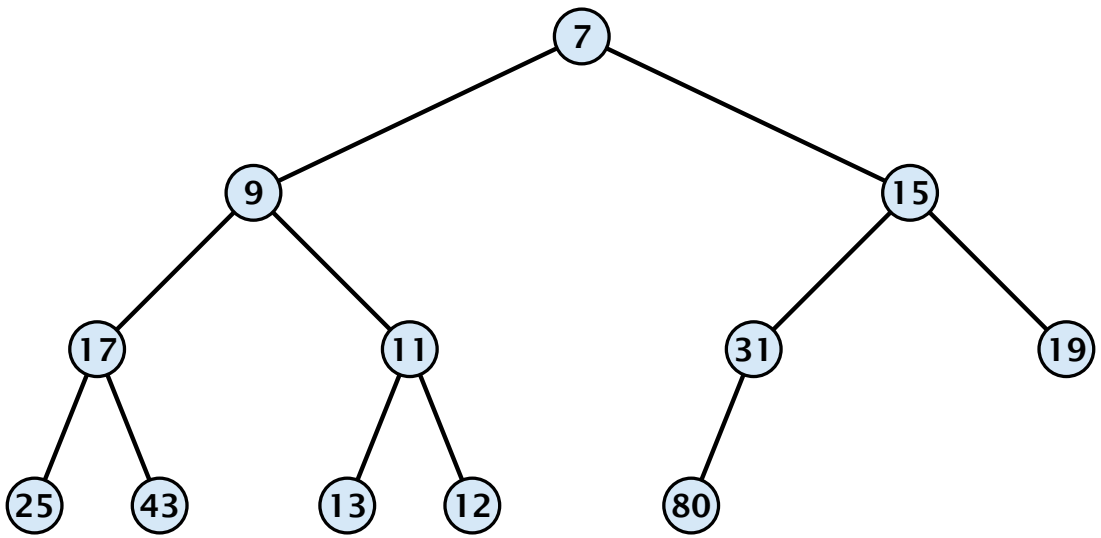
- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k)**:
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

# 7.3 Priority Queues



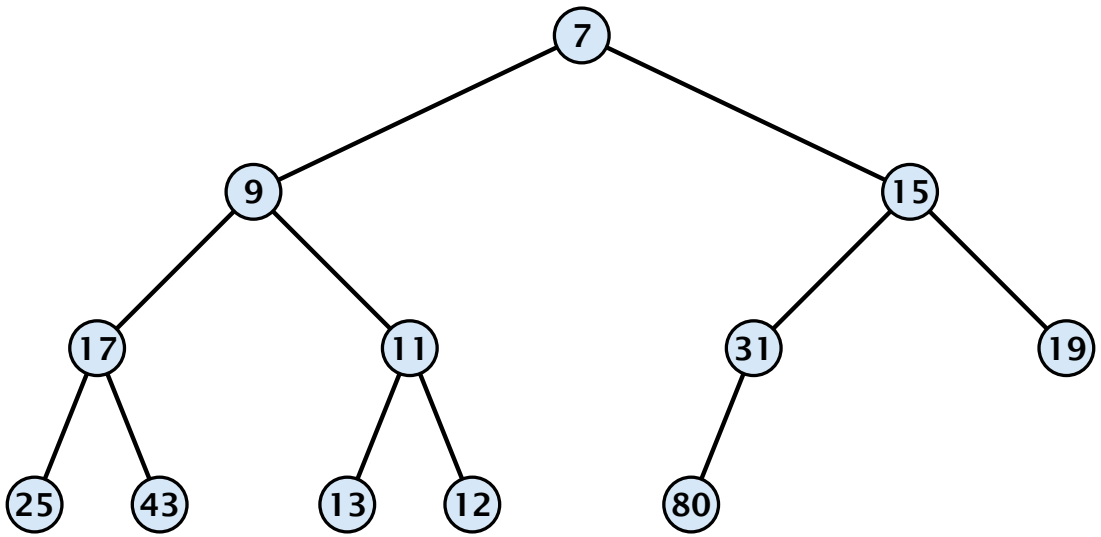
# 7.3 Priority Queues

- **Idee:** Speichere Elemente in einem fast vollständigen Binärbaum.



# 7.3 Priority Queues

- **Idee:** Speichere Elemente in einem fast vollständigen Binärbaum.
- Heapeigenschaft: Der Schlüssel eines Elements ist nicht größer als der Schlüssel eines Kindes.



# Binäre Heaps

**Operationen:**

- **minimum():** gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** überprüfe ob Zeiger auf Wurzel NULL ist. Zeit $\mathcal{O}(1)$.

# Binäre Heaps

**Operationen:**

- **minimum():** gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** überprüfe ob Zeiger auf Wurzel NULL ist. Zeit $\mathcal{O}(1)$.

# Binäre Heaps

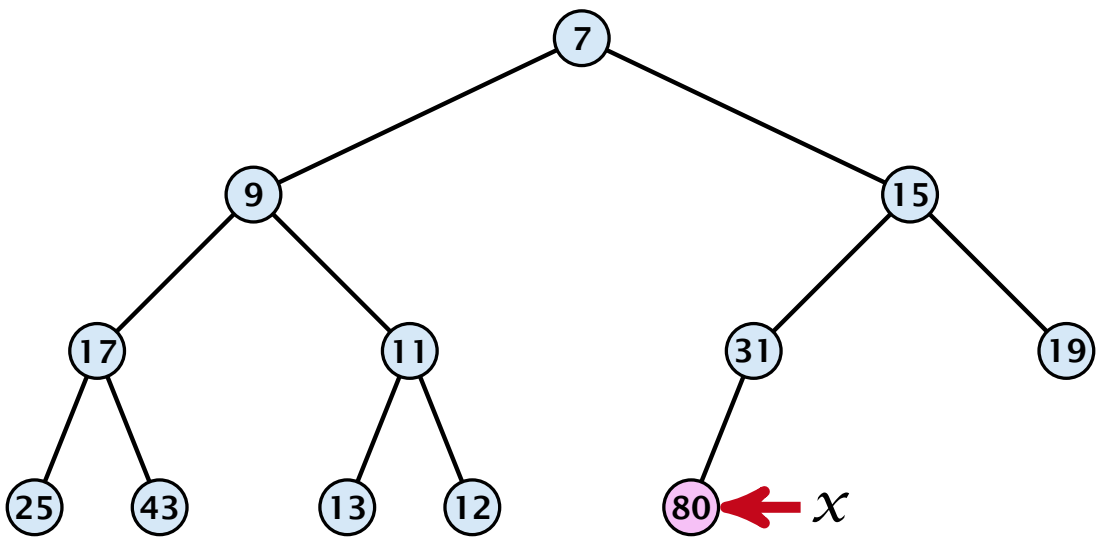
**Operationen:**

- **minimum():** gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** überprüfe ob Zeiger auf Wurzel NULL ist. Zeit $\mathcal{O}(1)$.

# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Berechne Vorgänger von $x$ (letztes Element wenn $x$ gelöscht wird) in Zeit $\mathcal{O}(\log n)$.



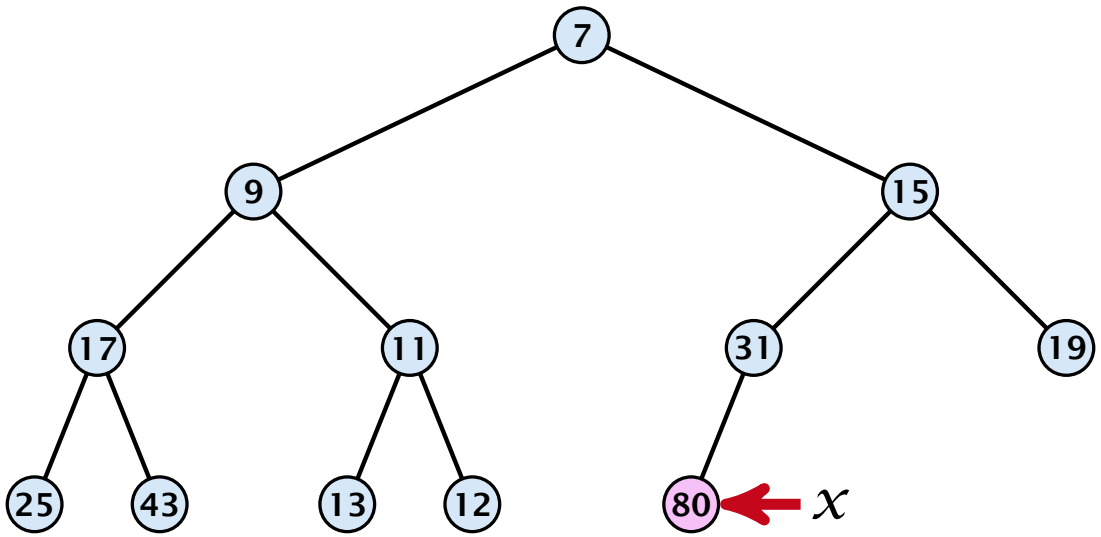
# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Berechne Vorgänger von $x$ (letztes Element wenn $x$ gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur nocht rechte Kanten.



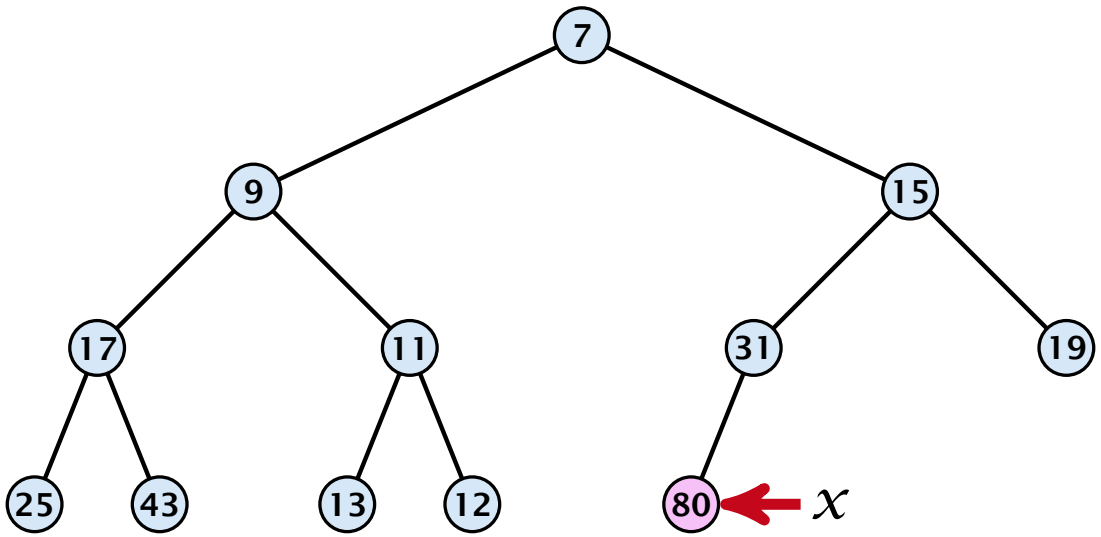
# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Berechne Vorgänger von $x$ (letztes Element wenn $x$ gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

  gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

  wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur nocht rechte Kanten.



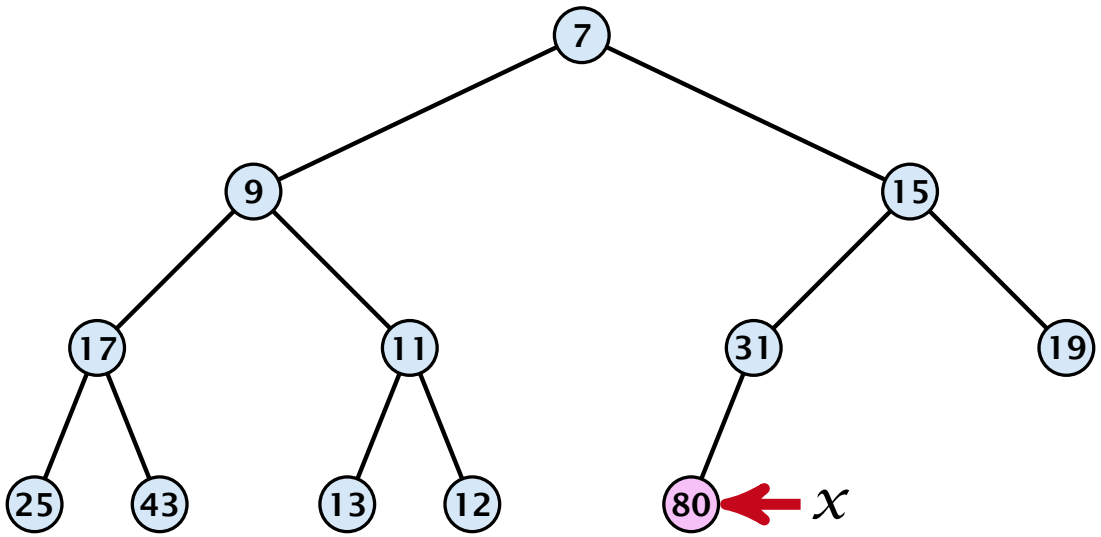
# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Berechne Vorgänger von $x$ (letztes Element wenn $x$ gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

  gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

  wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur nocht rechte Kanten.



# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Wir können Nachfolger von $x$ (letztes Element wenn ein Element eingefügt wird) in Zeit $\Theta(\log n)$ berechnen.




Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

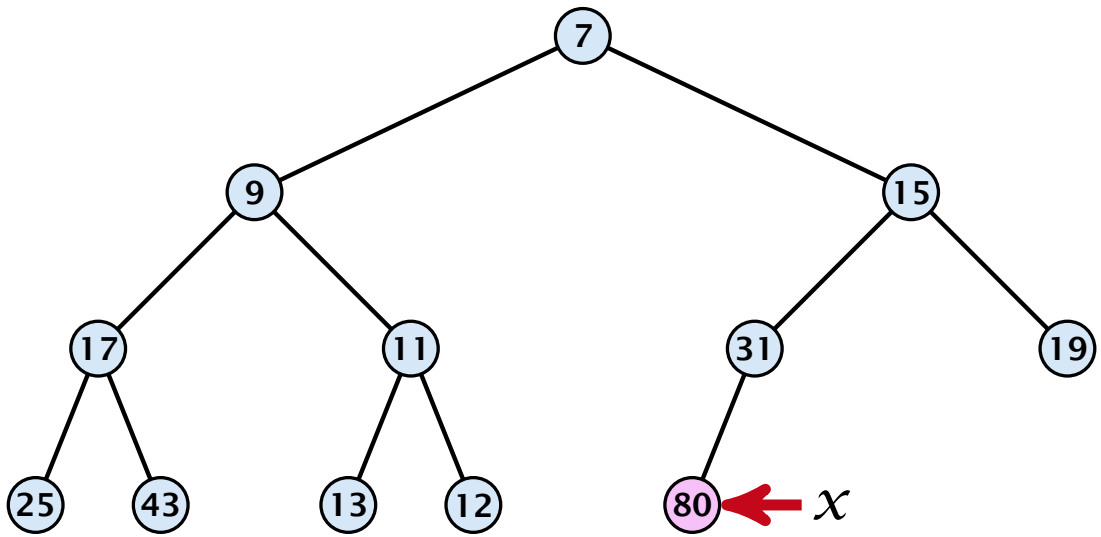
# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Wir können Nachfolger von $x$ (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

falls man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch linke Kanten.




Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

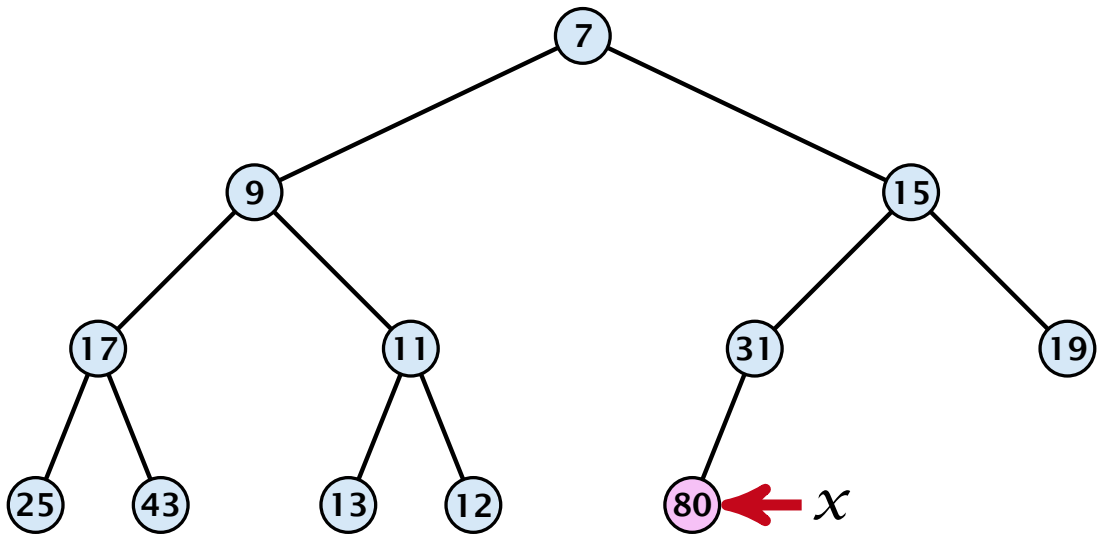
# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Wir können Nachfolger von $x$ (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

  gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

  falls man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch linke Kanten.




Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

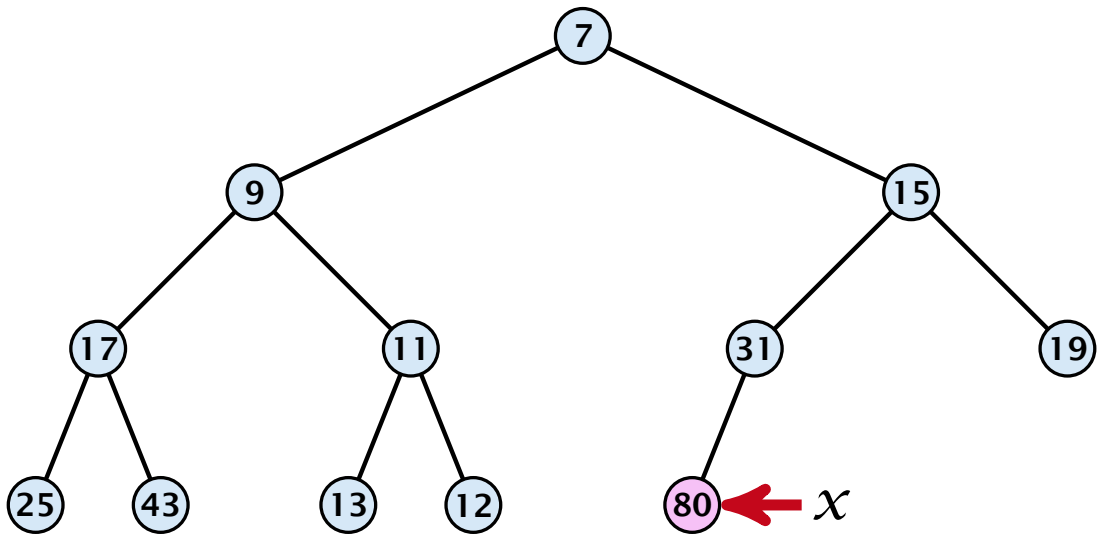
# 7.3 Priority Queues

Verwalte Zeiger auf letztes Element $x$.

- Wir können Nachfolger von $x$ (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

  gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

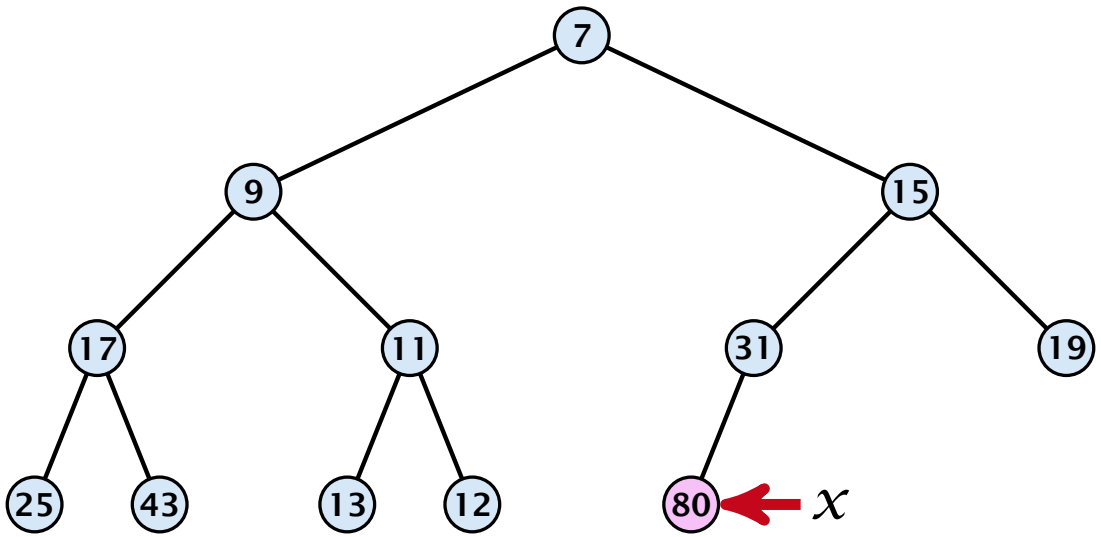
  falls man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch linke Kanten.




Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

# Einfügen

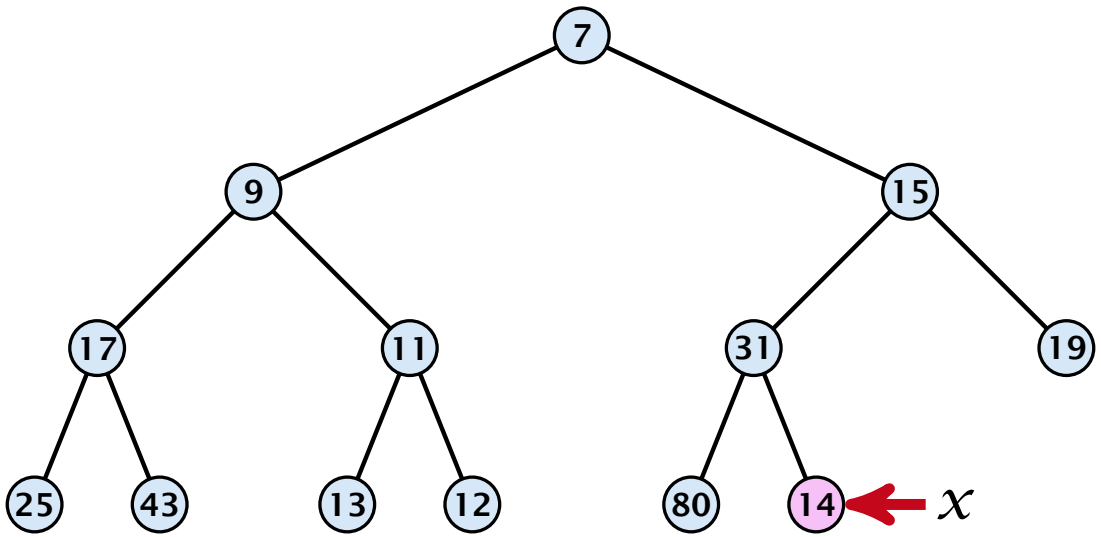
1. Füge Element am Nachfolger von $x$ ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.




Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

# Einfügen

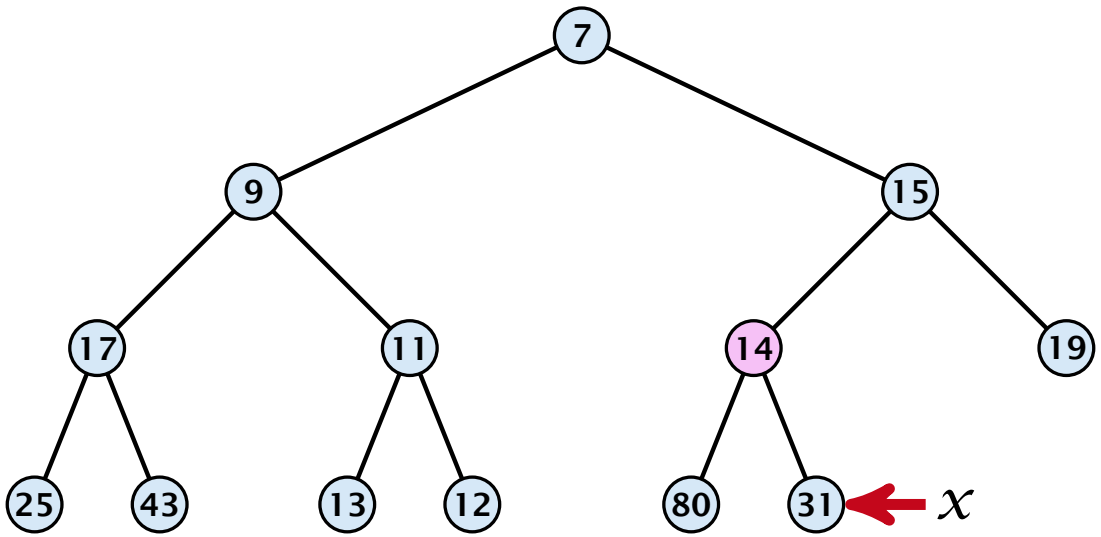
1. Füge Element am Nachfolger von $x$ ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.




Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

# Einfügen

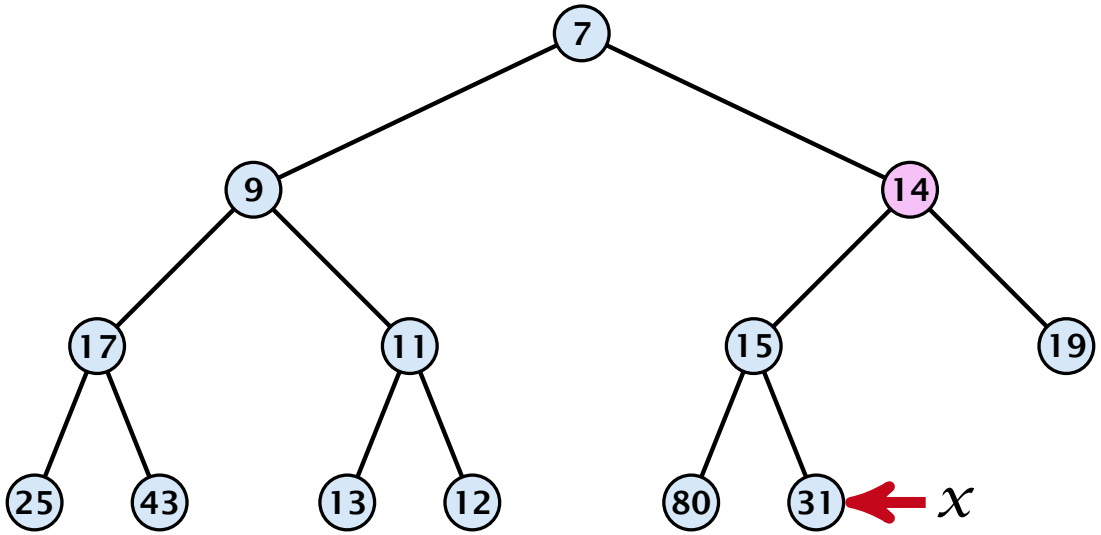
1. Füge Element am Nachfolger von $x$ ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.




Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

# Einfügen

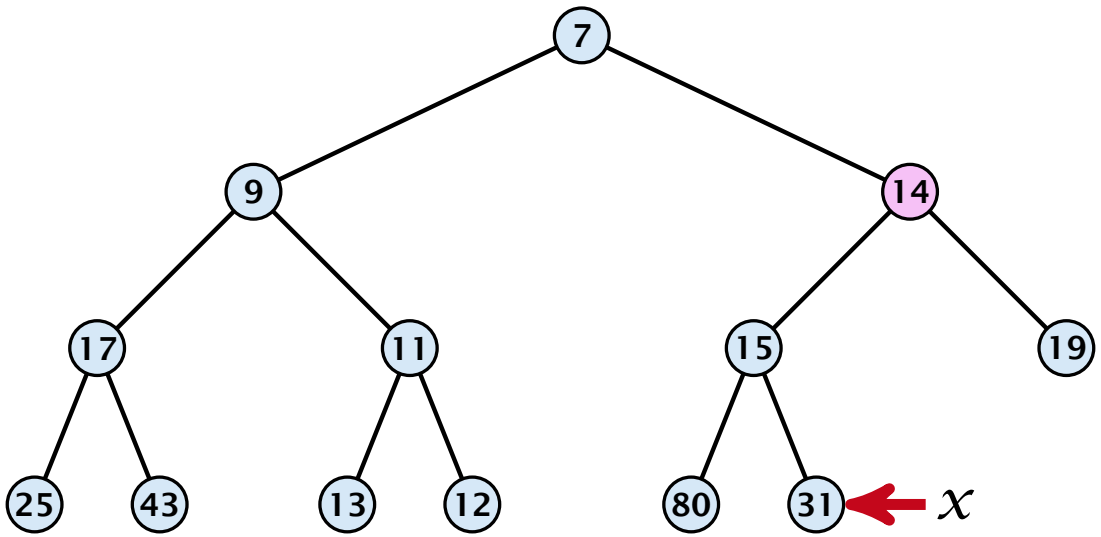
1. Füge Element am Nachfolger von $x$ ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.




Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

# Einfügen

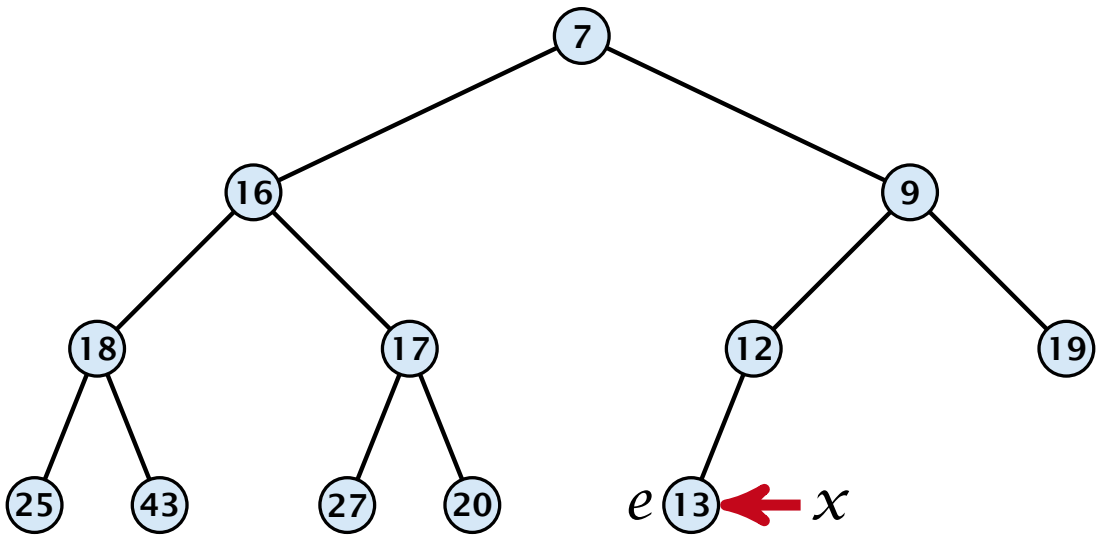
1. Füge Element am Nachfolger von $x$ ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.




Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

# Löschen

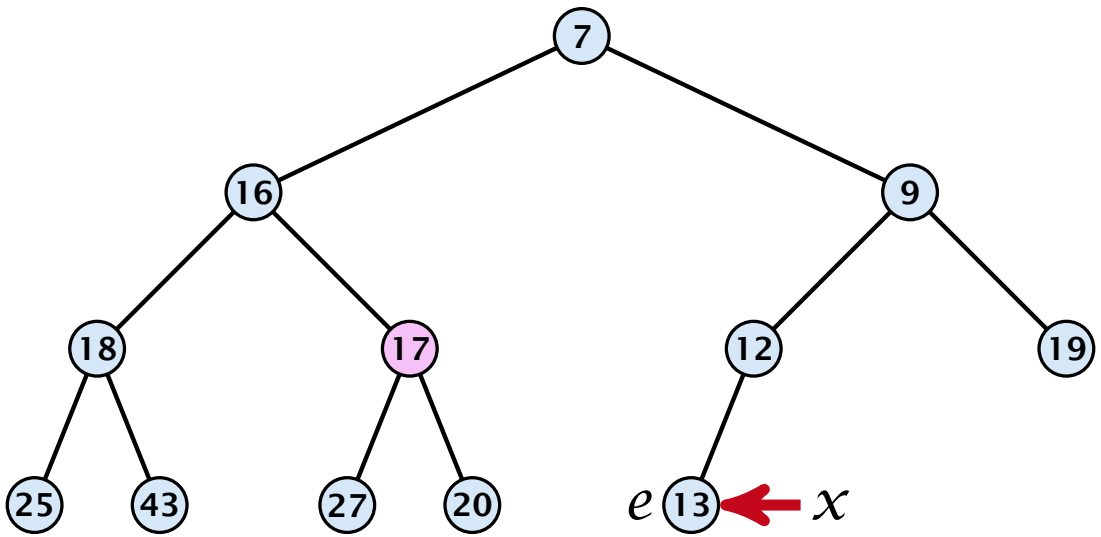
1. Vertausche zu löschenes Element mit dem letzen Element $e$.
2. Stelle die Heapeigenschaft für Element $e$ wieder her.




An der neuen Position wandert $e$ entweder auf oder abwärts (aber nicht beides).

# Löschen

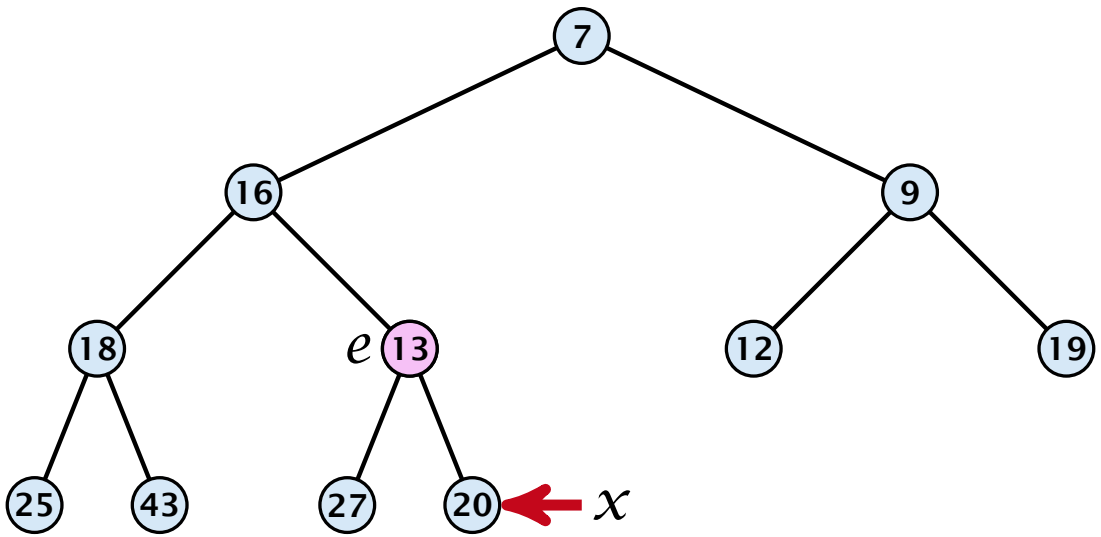
1. Vertausche zu löschenes Element mit dem letzen Element $e$.
2. Stelle die Heapeigenschaft für Element $e$ wieder her.




An der neuen Position wandert $e$ entweder auf oder abwärts (aber nicht beides).

# Löschen

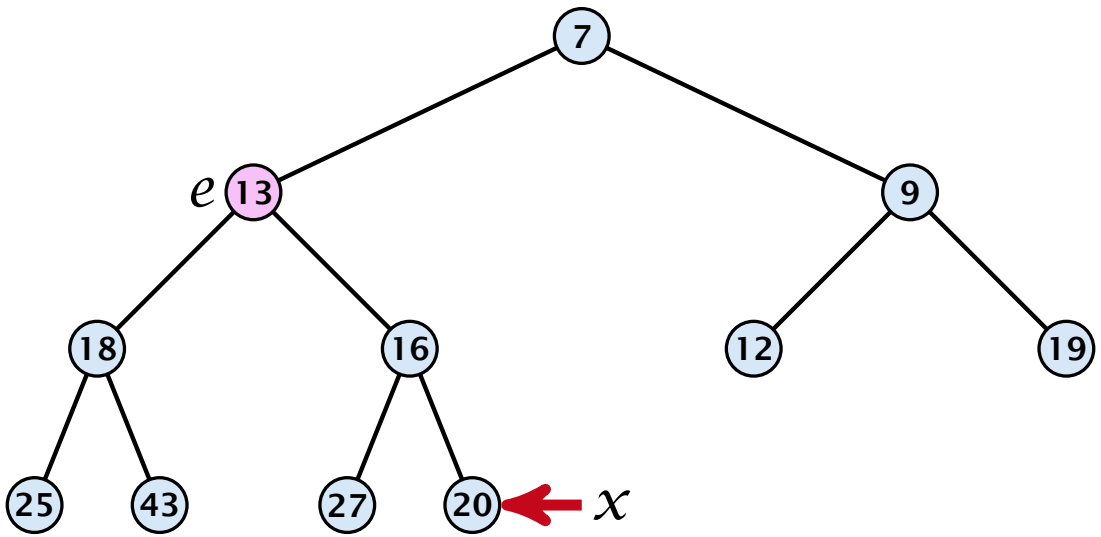
1. Vertausche zu löschenes Element mit dem letzen Element $e$.
2. Stelle die Heapeigenschaft für Element $e$ wieder her.




An der neuen Position wandert $e$ entweder auf oder abwärts (aber nicht beides).

# Löschen

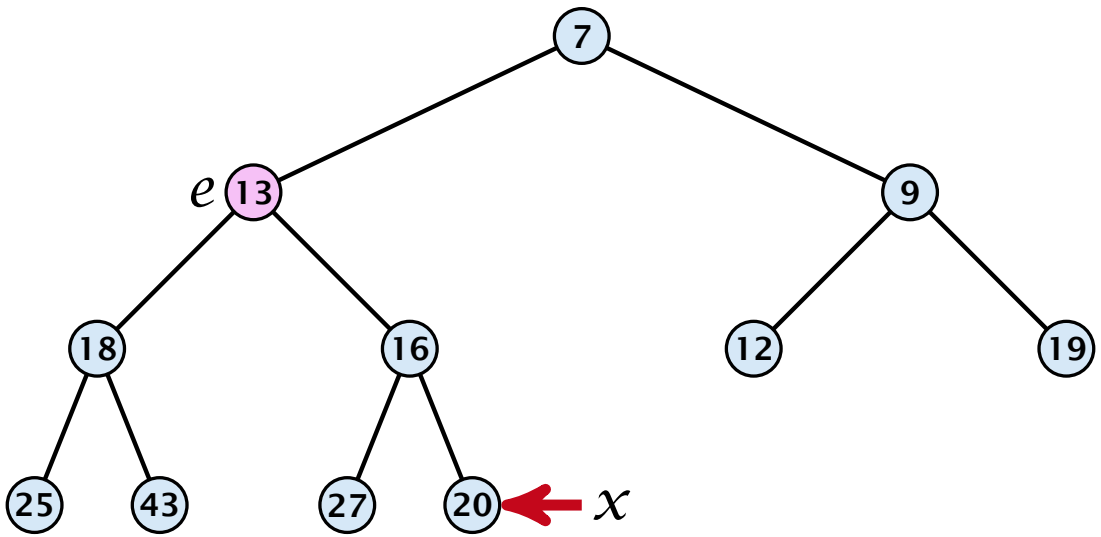
1. Vertausche zu löschenes Element mit dem letzen Element $e$.
2. Stelle die Heapeigenschaft für Element $e$ wieder her.




An der neuen Position wandert $e$ entweder auf oder abwärts (aber nicht beides).

# Löschen

1. Vertausche zu löschenes Element mit dem letzen Element $e$.
2. Stelle die Heapeigenschaft für Element $e$ wieder her.




An der neuen Position wandert $e$ entweder auf oder abwärts (aber nicht beides).

# Binäre Heaps

**Operationen:**

- **minimum():** Gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** Überprüfe ob Wurzelzeiger NULL. Zeit $\mathcal{O}(1)$.
- **insert($k$):** füge bei Nachfolger von $x$ ein. bubble up. Zeit $\mathcal{O}(\log n)$.
- **delete($h$):** tausche mit $x$; bubble up **or** sift-down. Zeit $\mathcal{O}(\log n)$.

# Binäre Heaps

```
1 void bubbleUp(TreeNode* v) {
2     while (v->parent && v->parent->val > v->val)
3         swap(v,v->parent);
4 }
```

Vertausche mit Elternknoten solange dein Wert kleiner als Wert des Elternknotens.

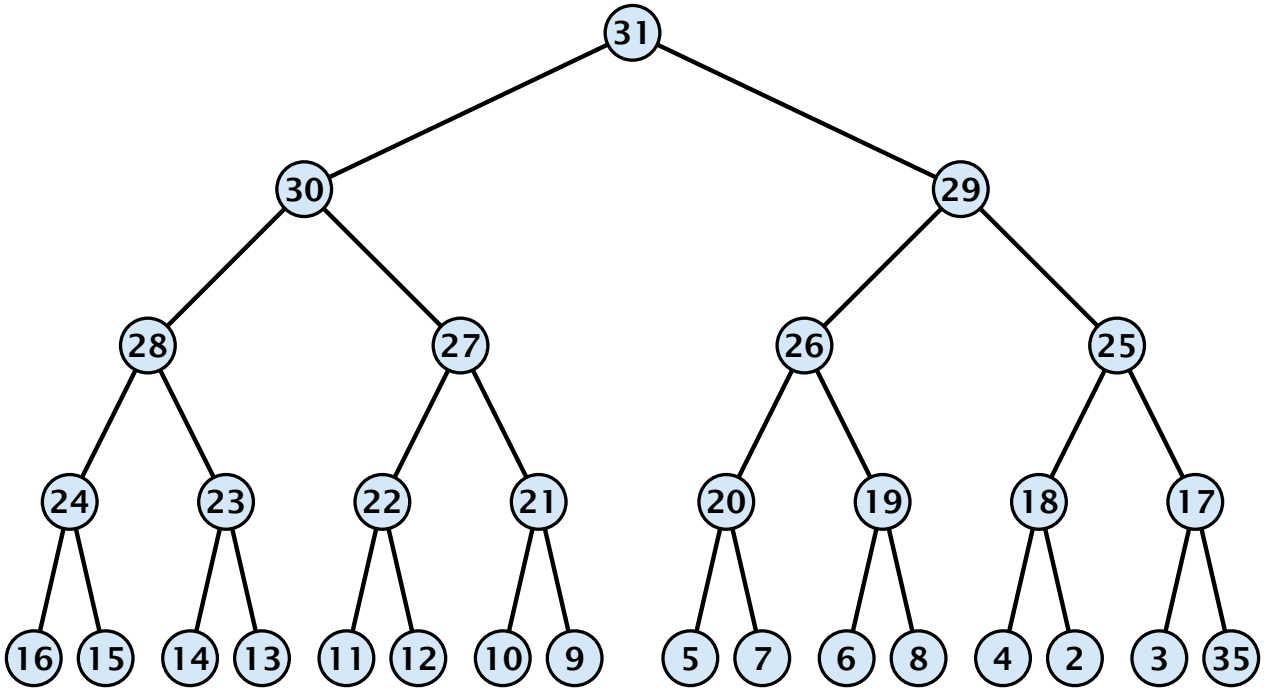
# Binäre Heaps

```
1  void siftDown(TreeNode* v) {
2      TreeNode* m = NULL;
3      while (true) {
4          int minValue = v->val;
5          if (v->left && minValue > v->left->val) {
6              minValue = v->left->val;
7              m = v->left;
8          }
9          if (v->right && minValue > v->right->val) {
10             minValue = v->right->val;
11             m = v->right;
12         }
13         if (v->val != minValue)
14             swap(v,m);
15         else break;
16     }
17 }
```

Vertausche mit dem kleineren der Kinder, solange der Wert kleiner als dein eigener ist.

# Build Heap

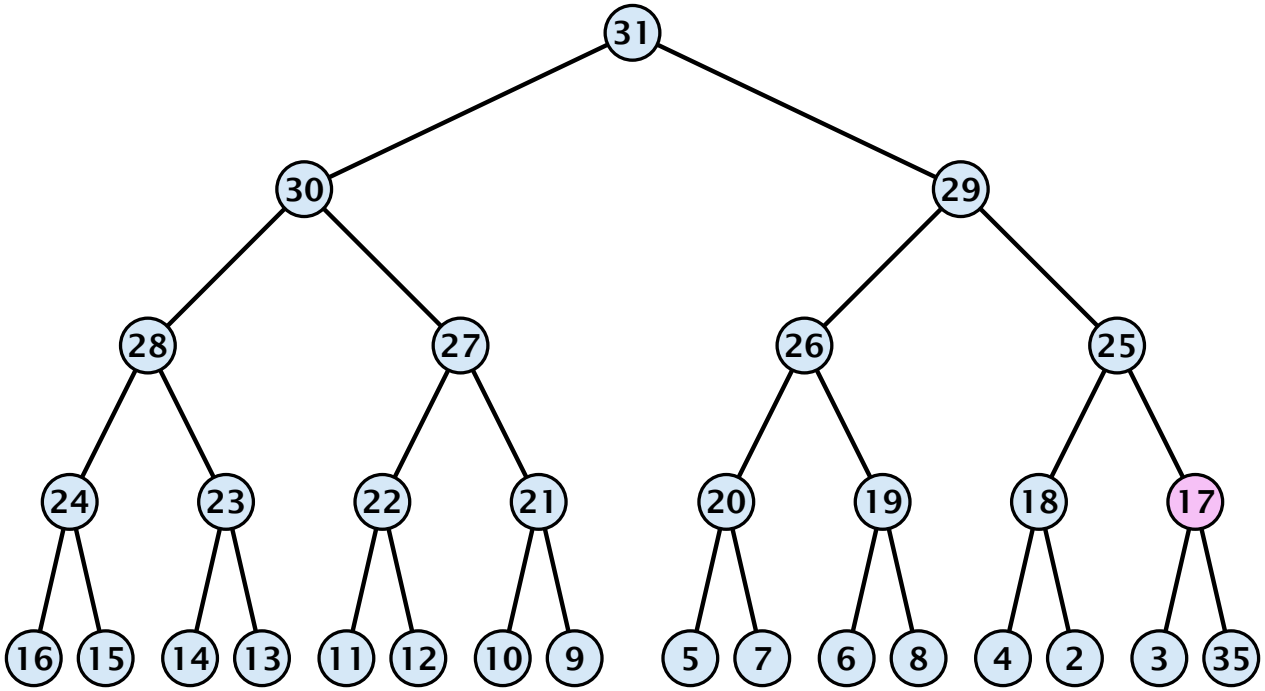
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

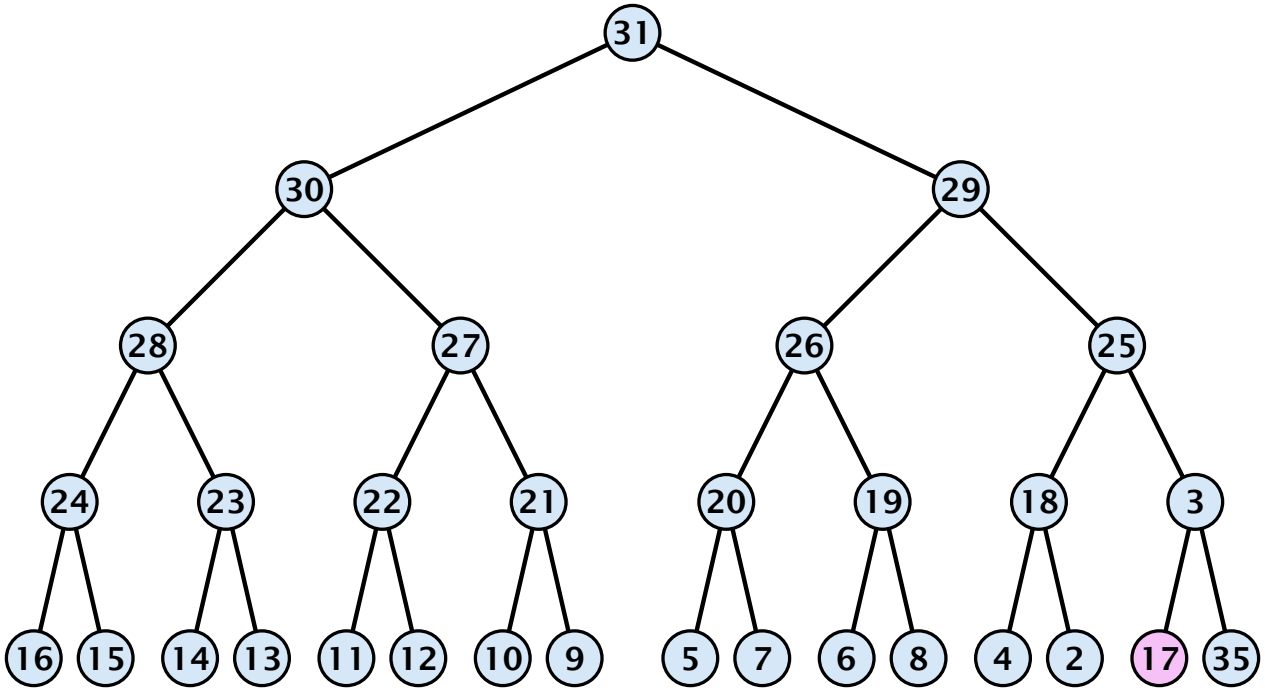
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

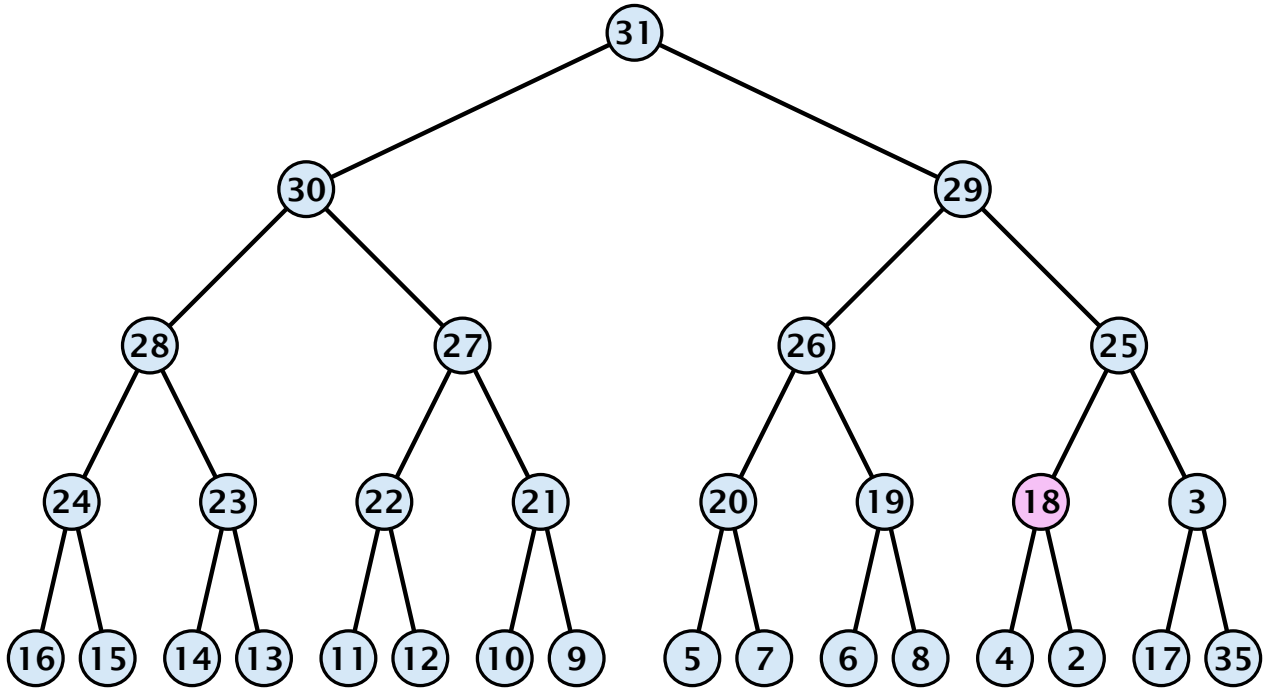
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

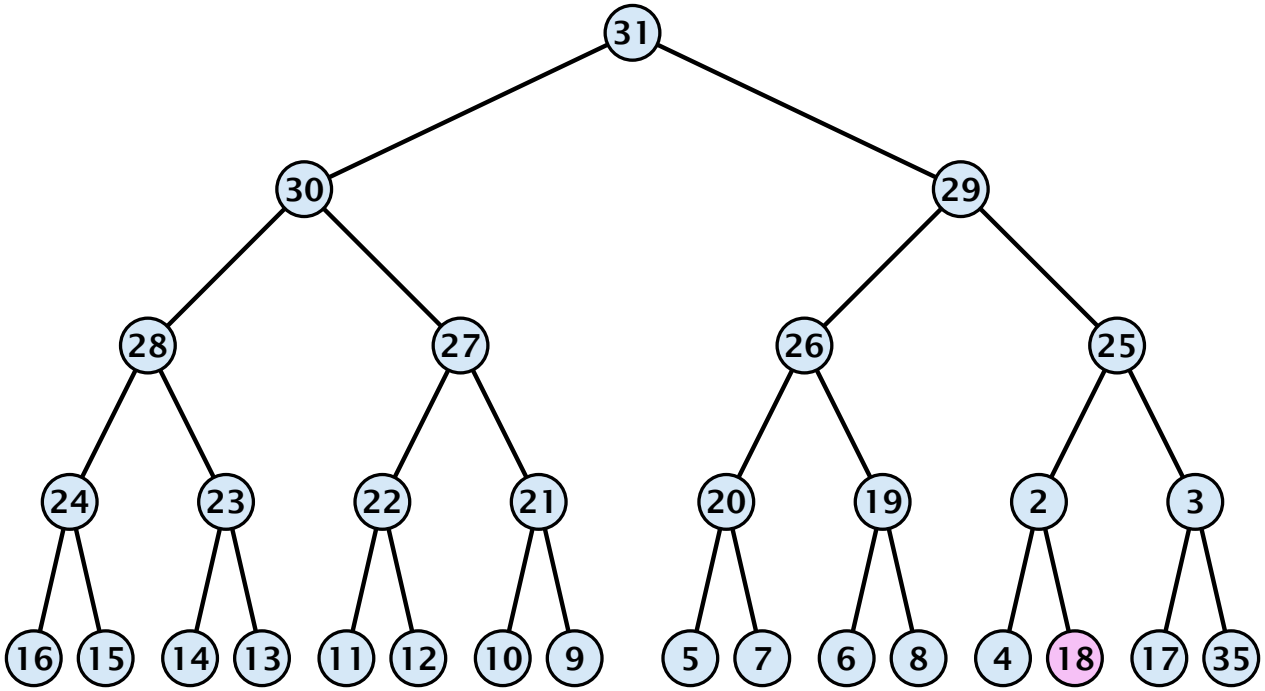
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

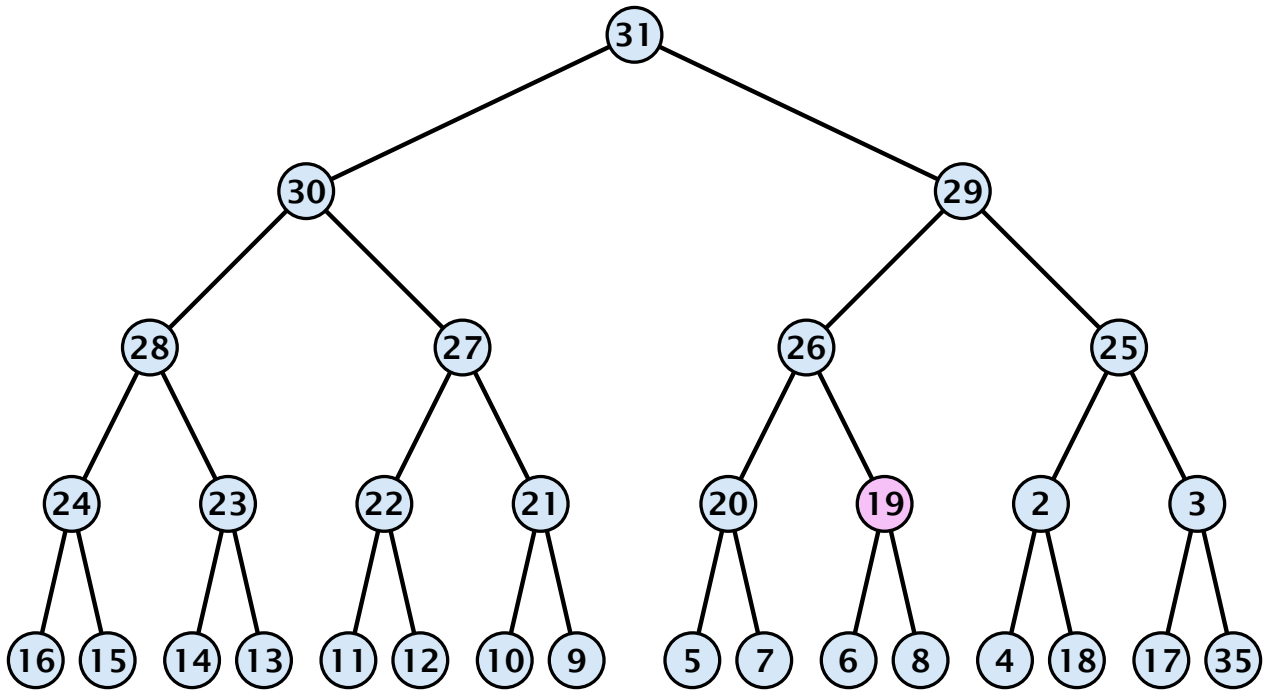
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

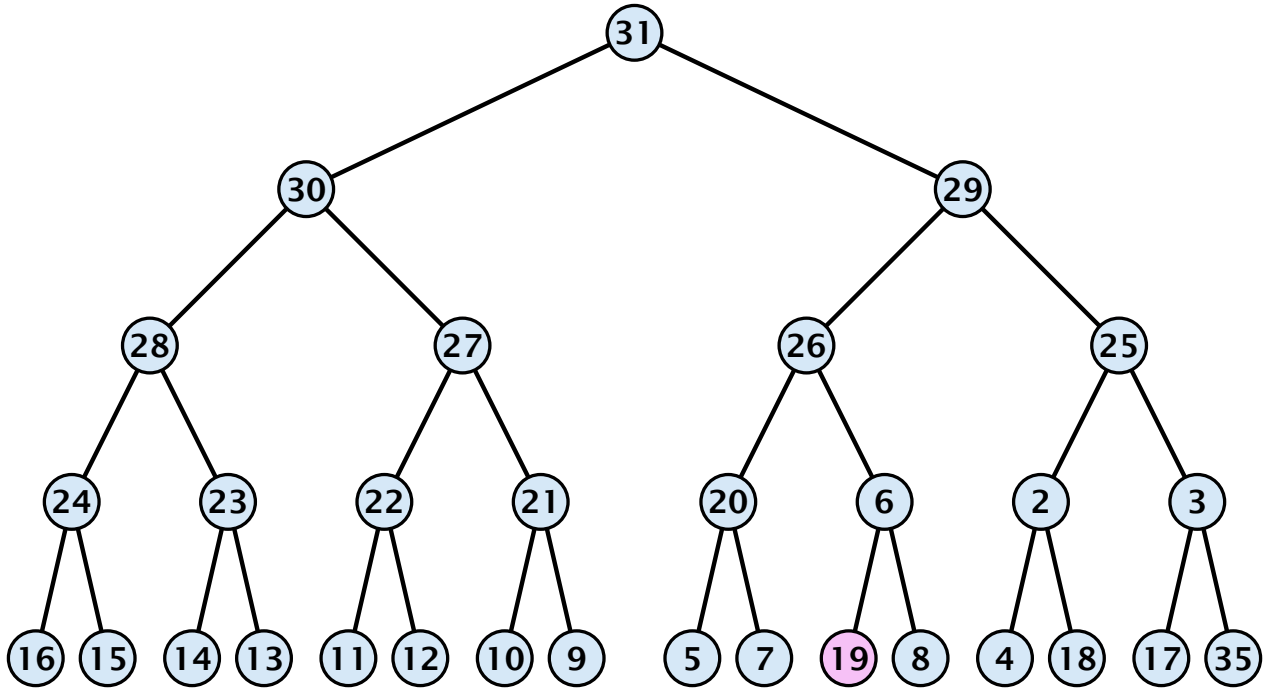
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

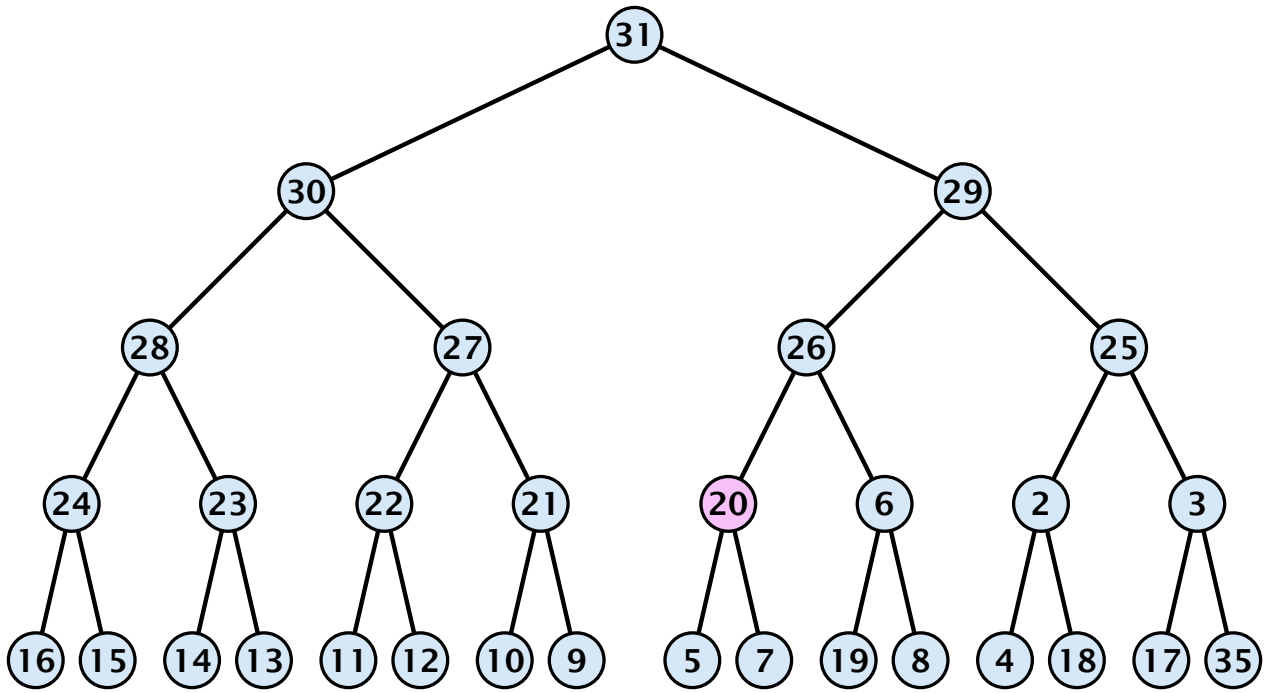
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

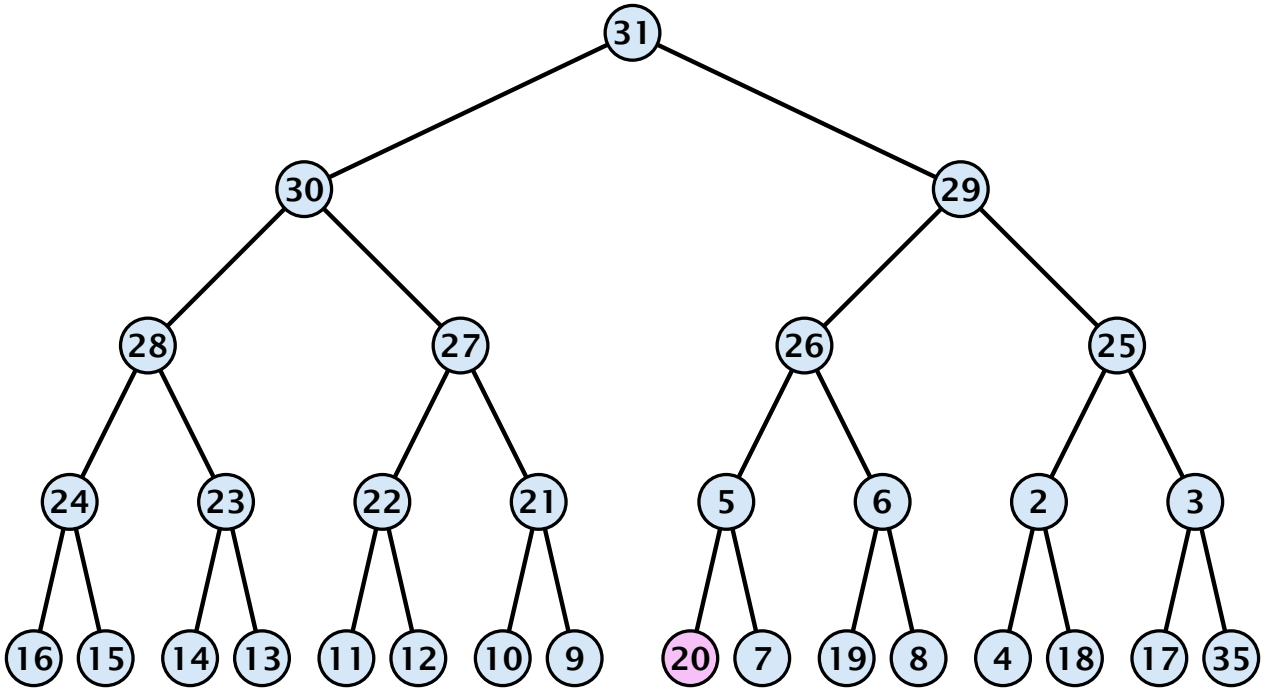
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

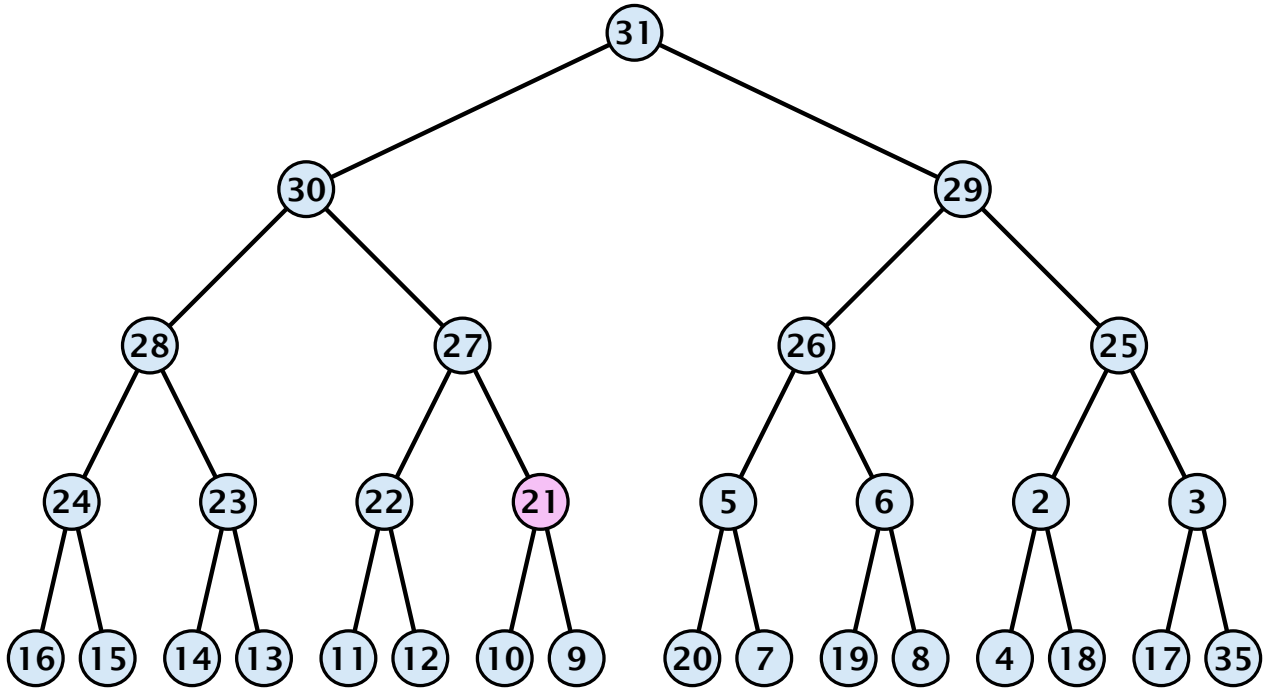
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

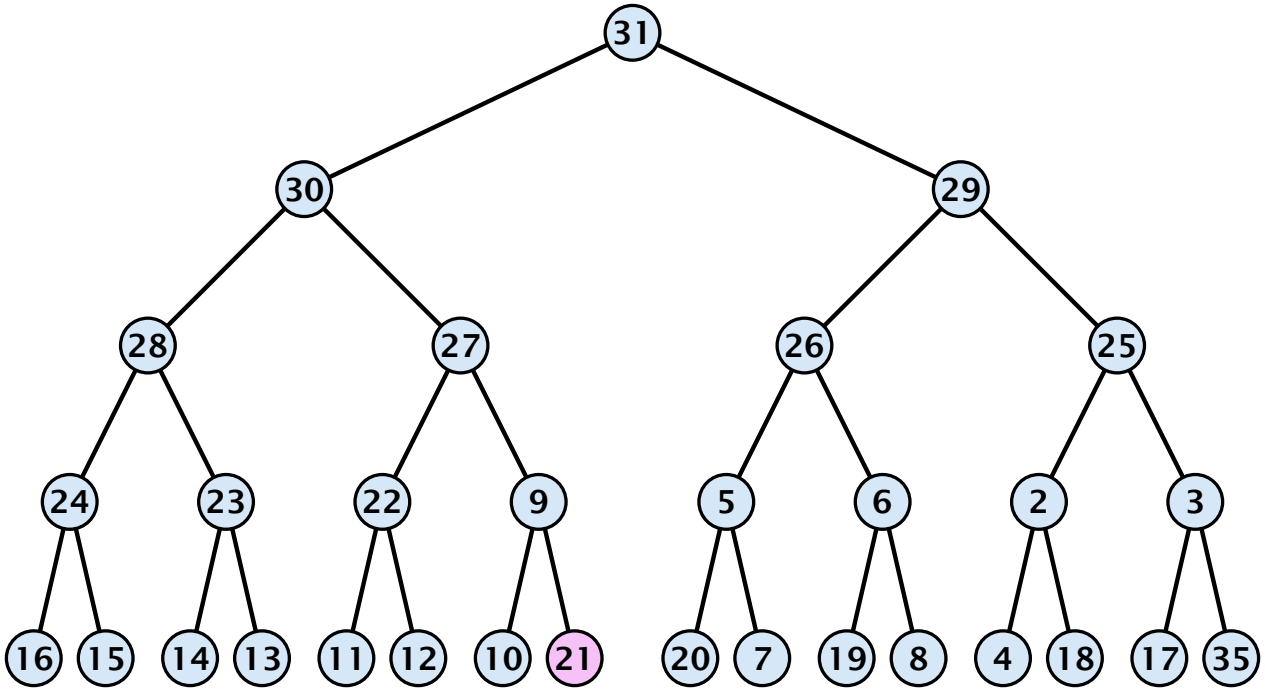
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

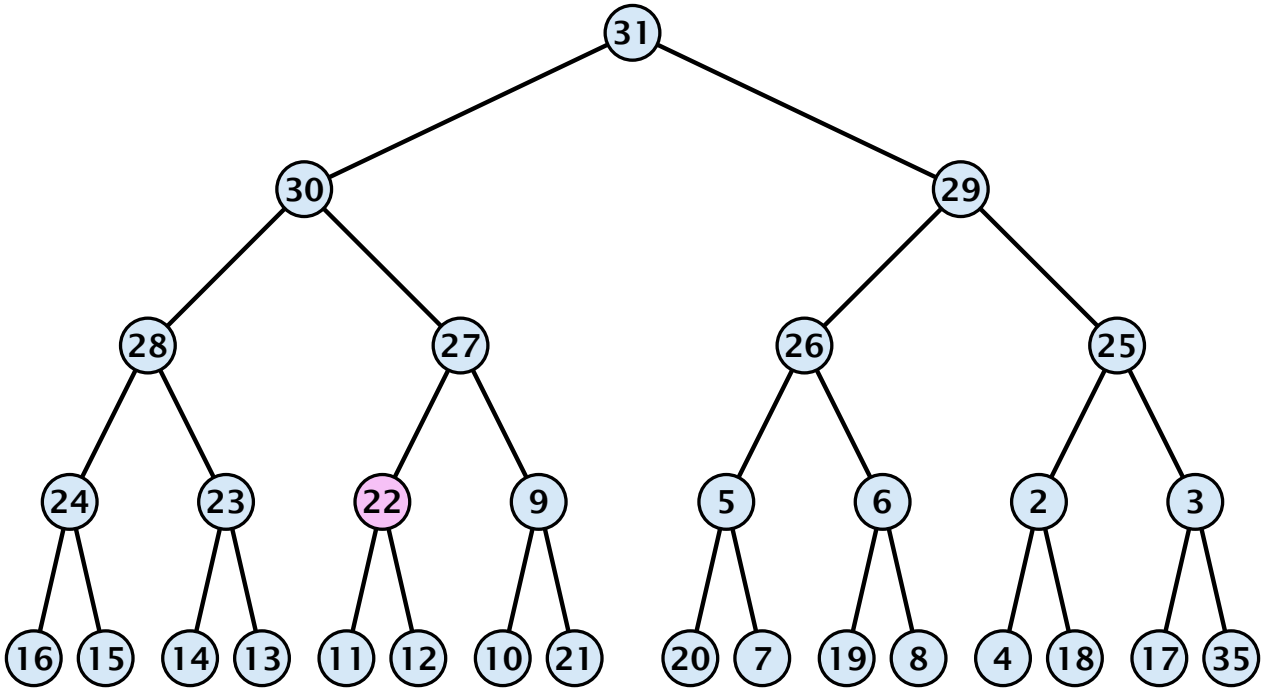
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

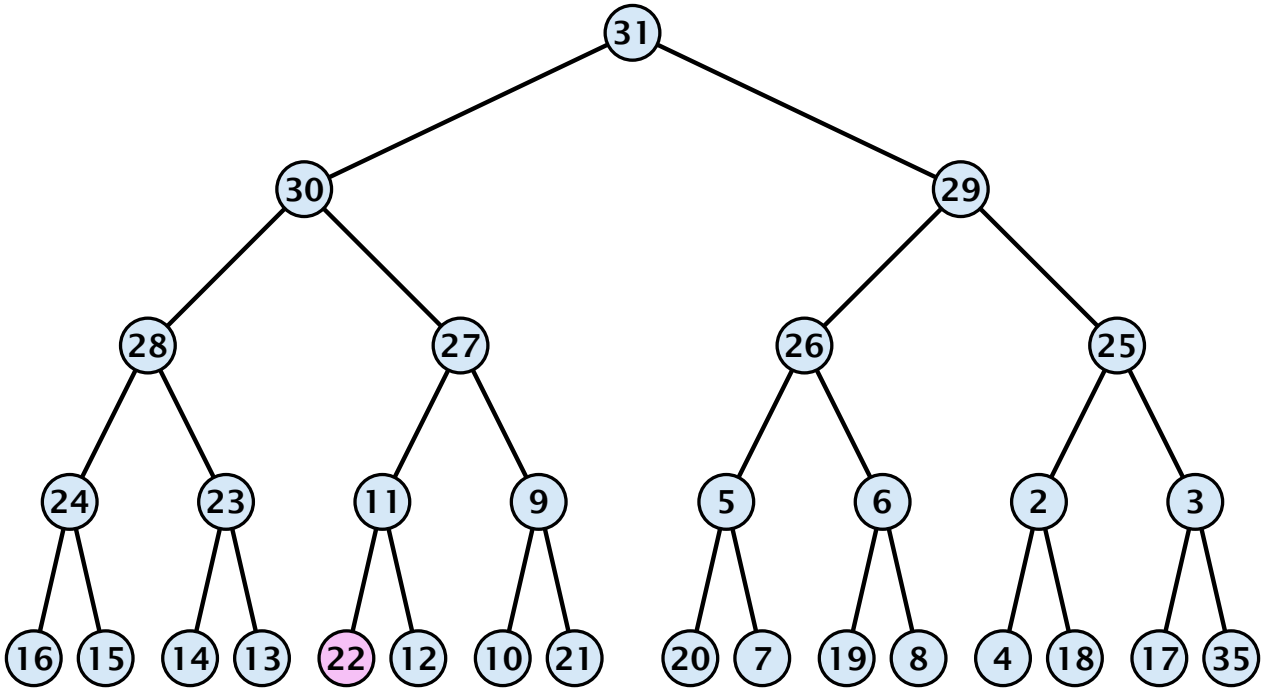
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

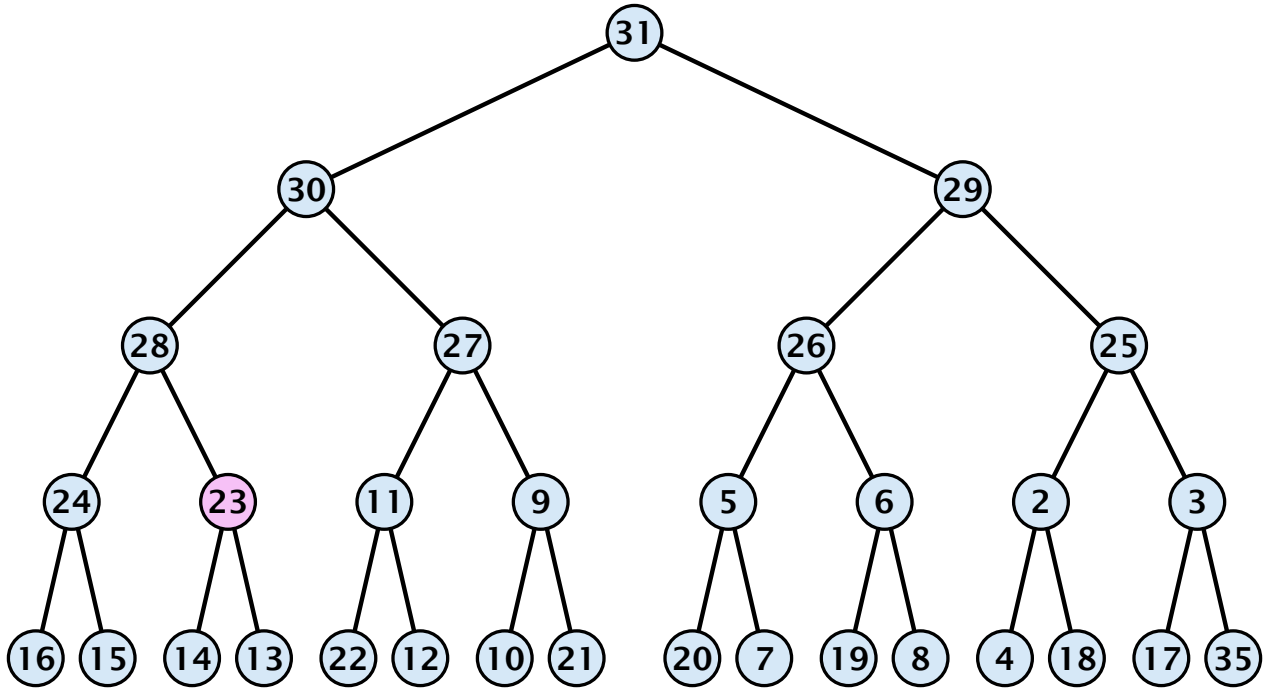
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

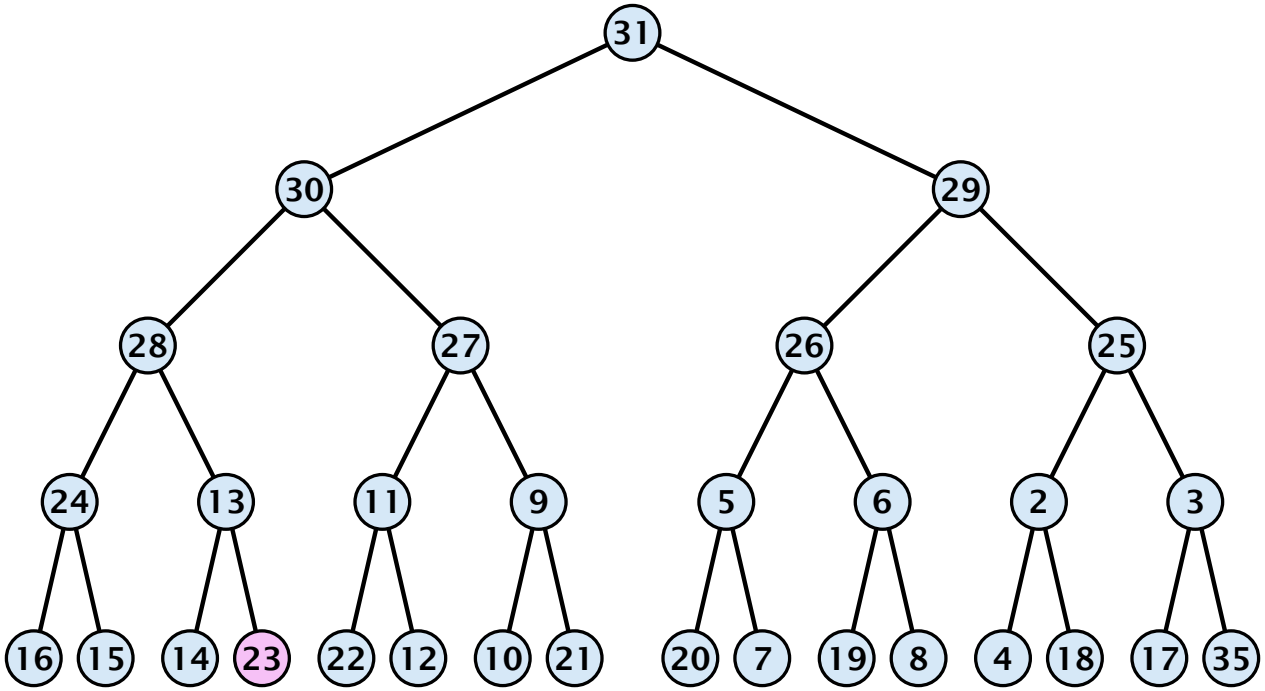
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

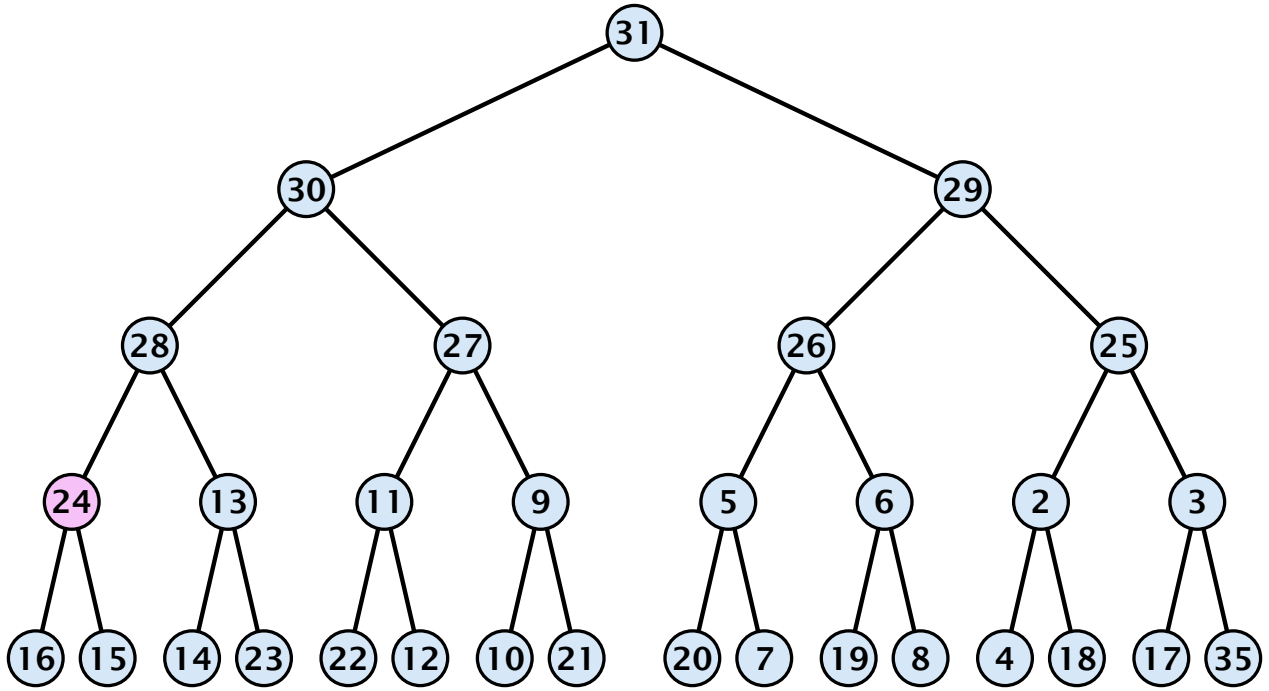
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

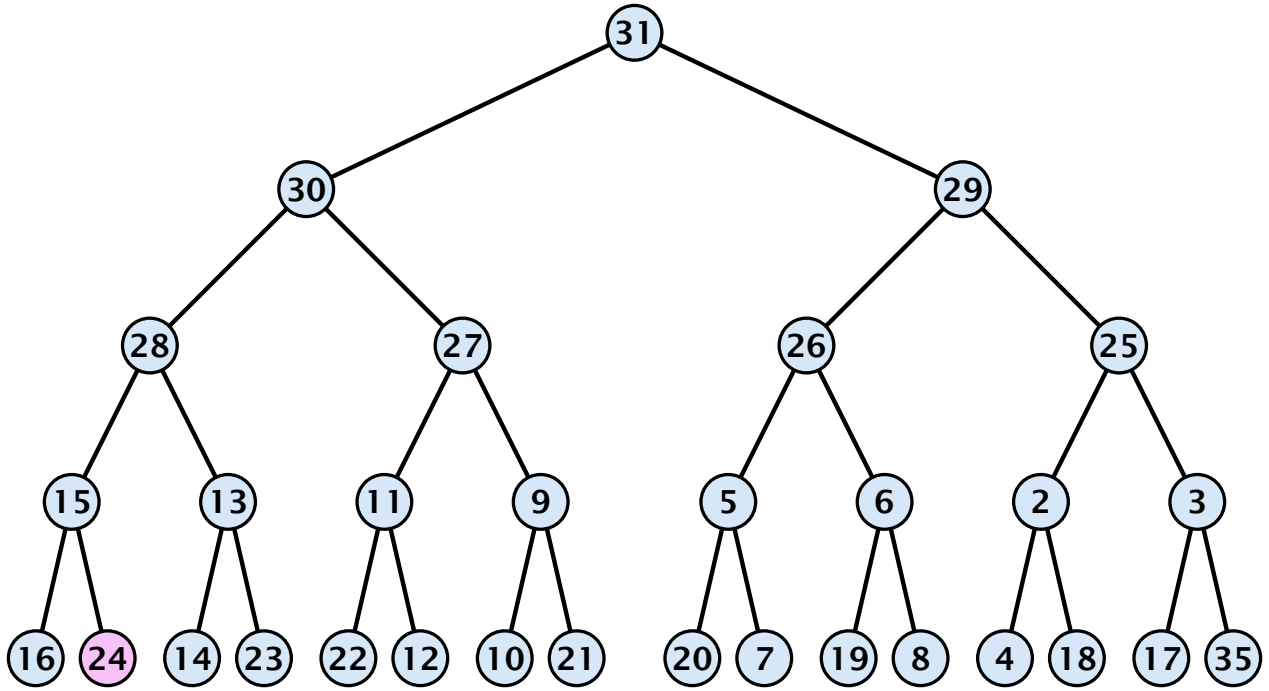
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

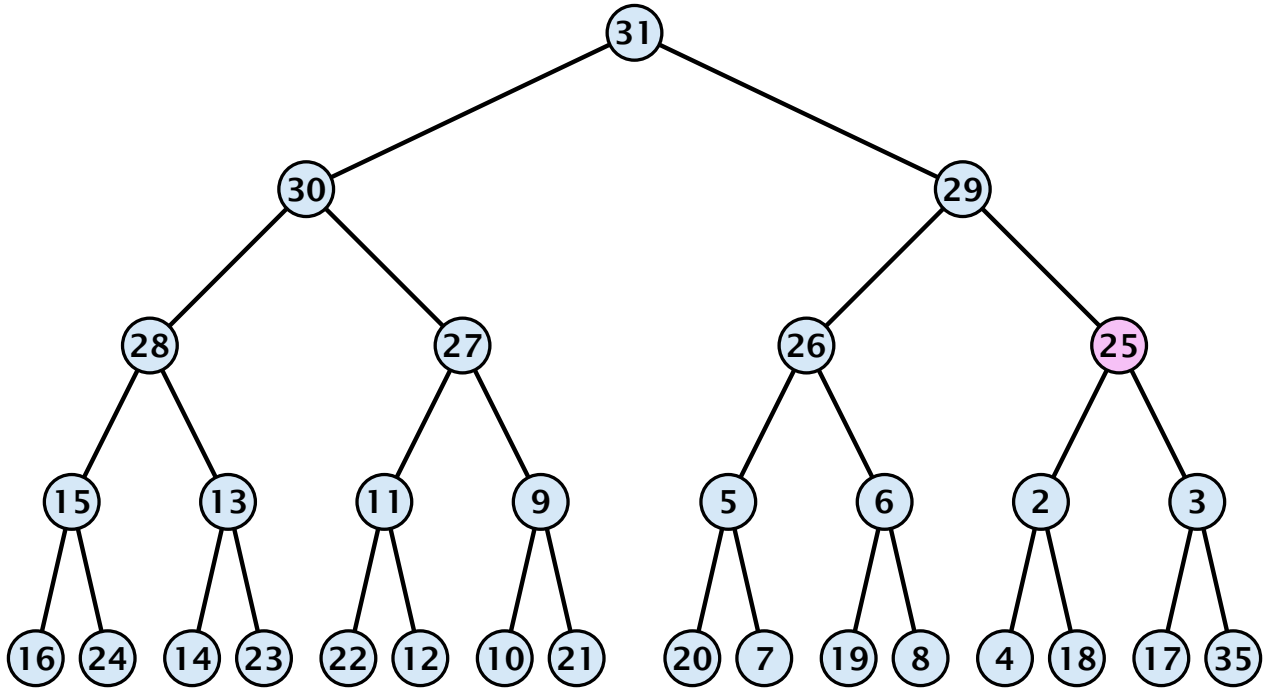
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

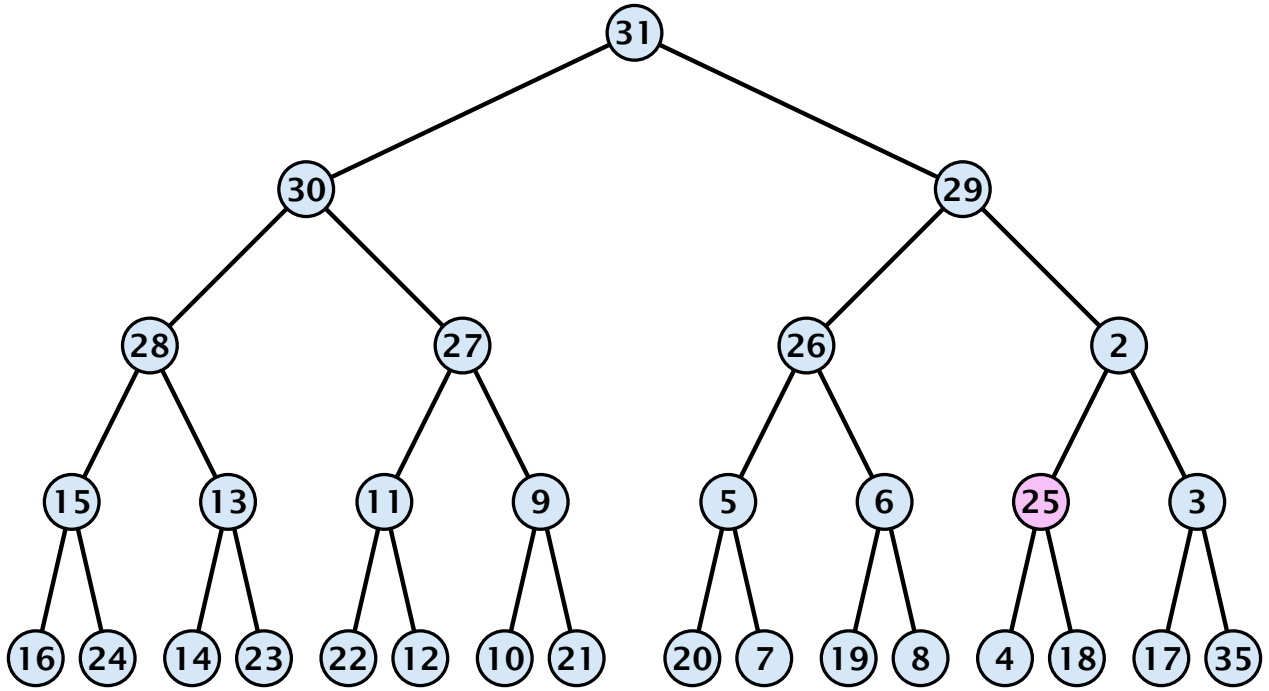
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

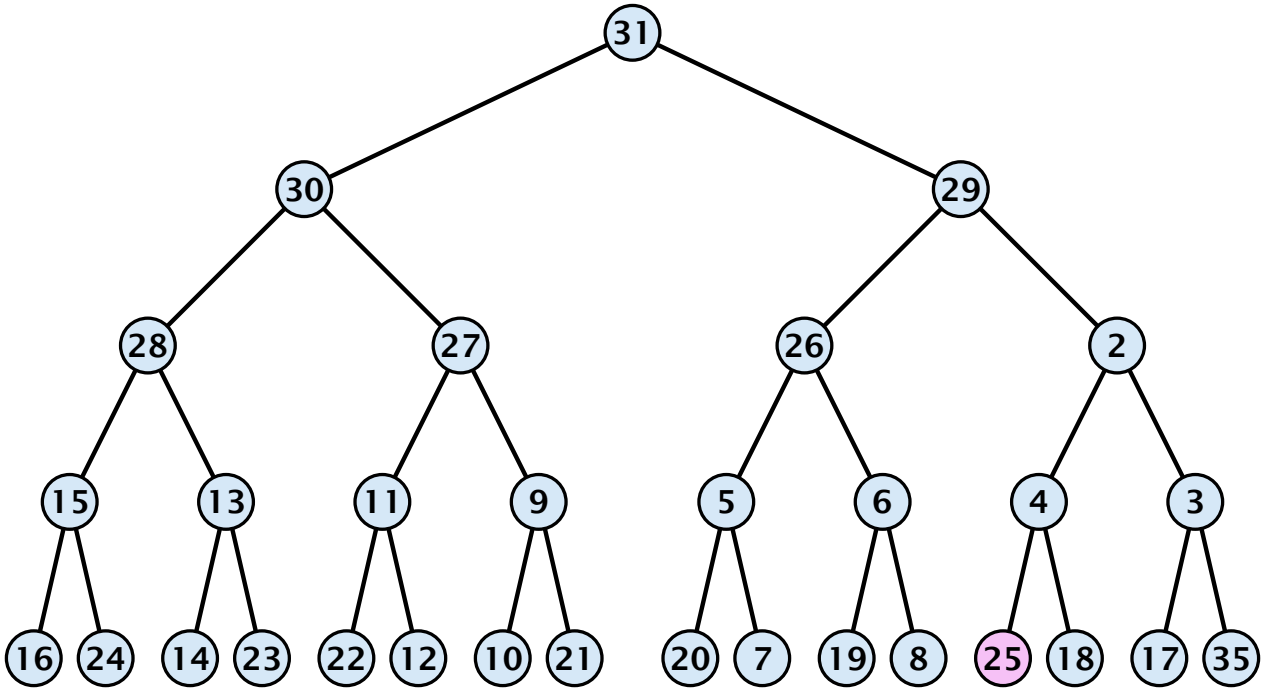
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

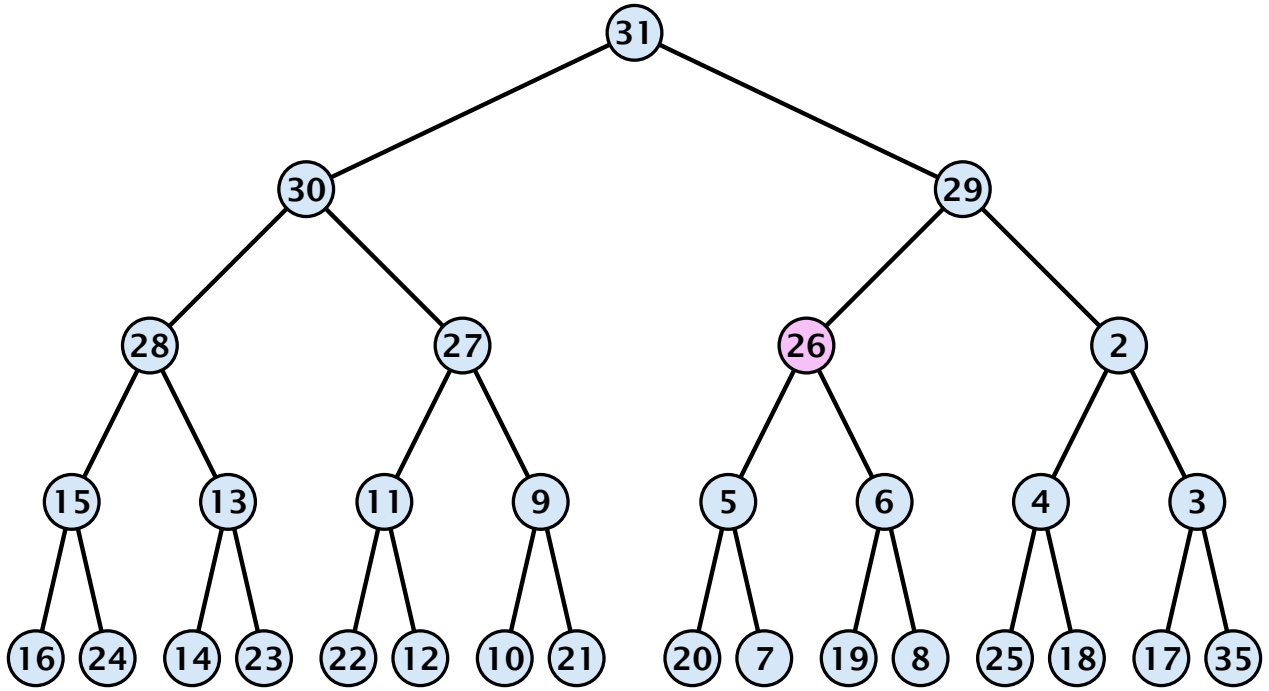
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

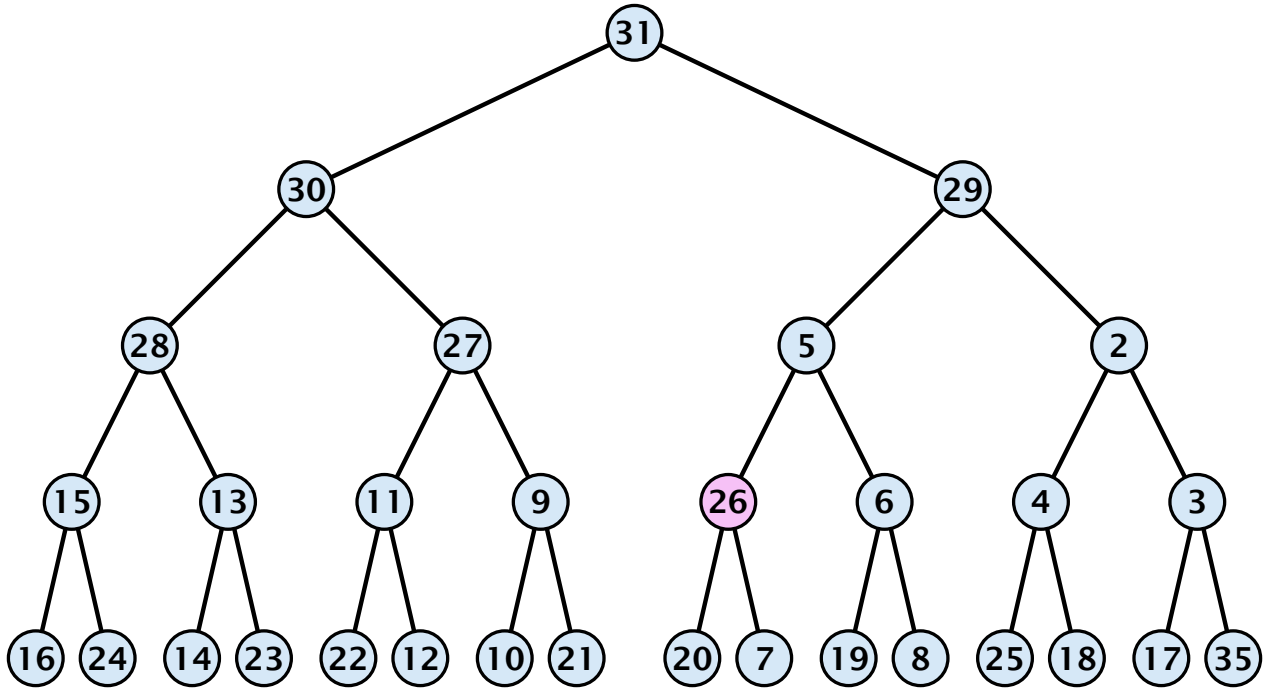
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

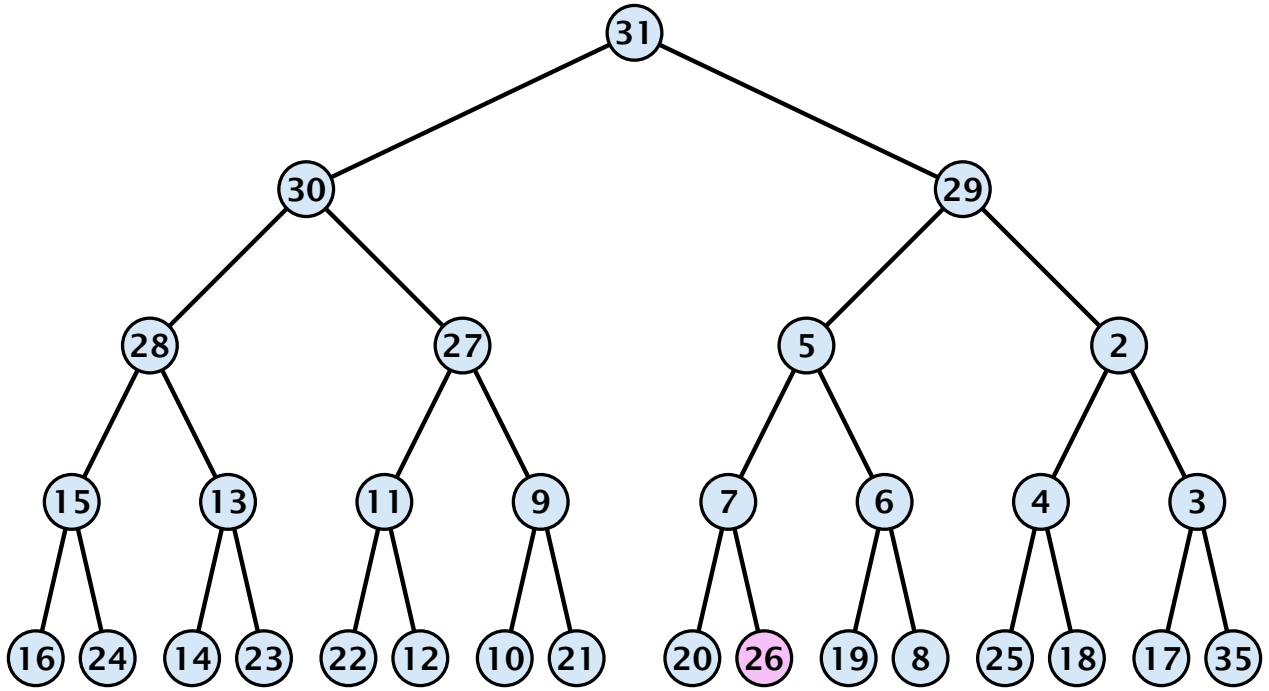
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

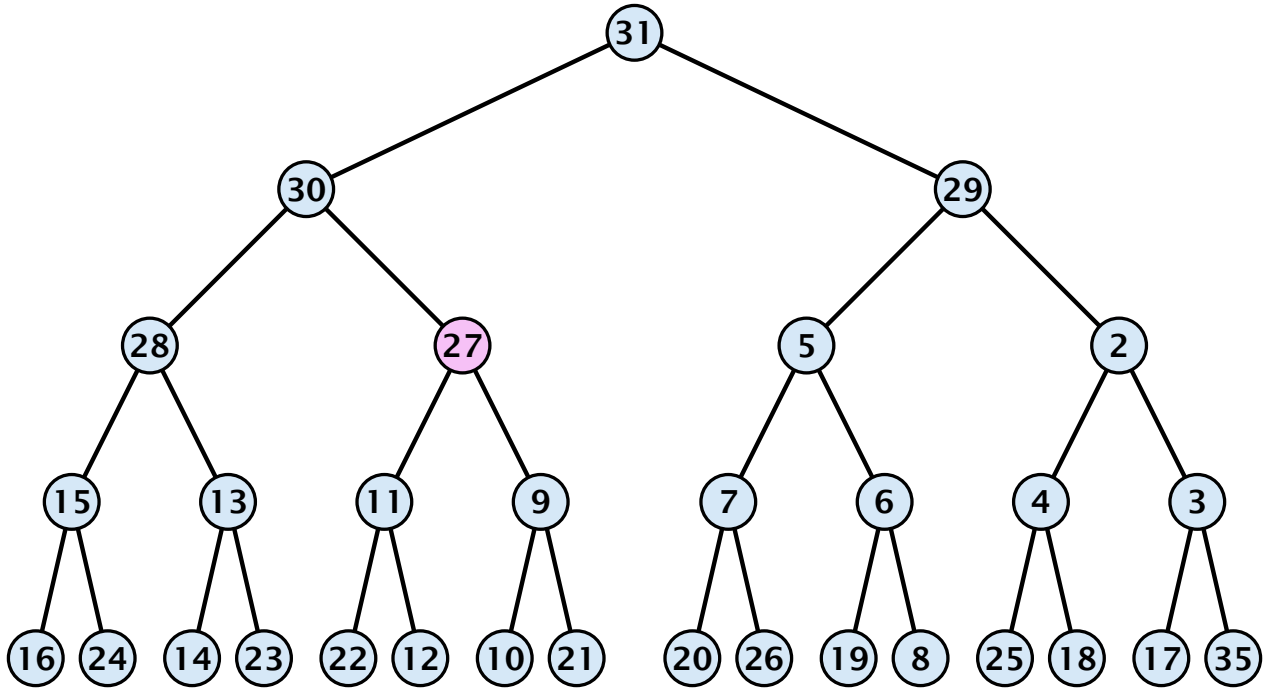
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

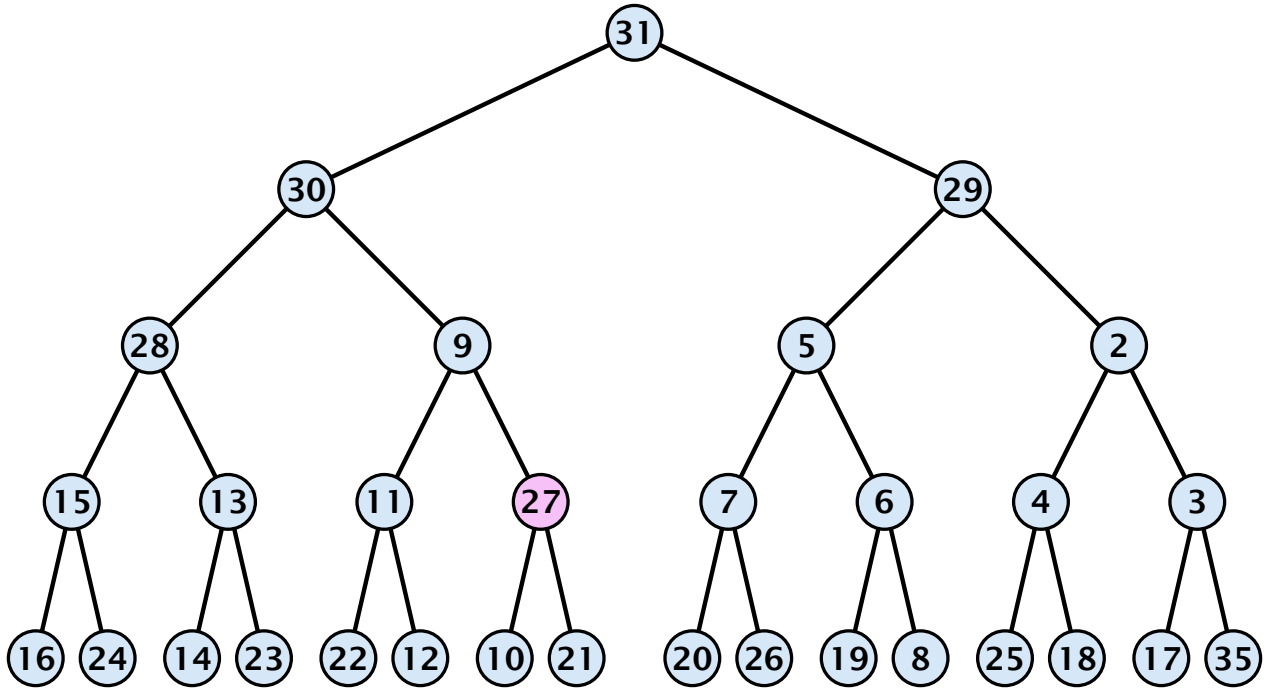
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

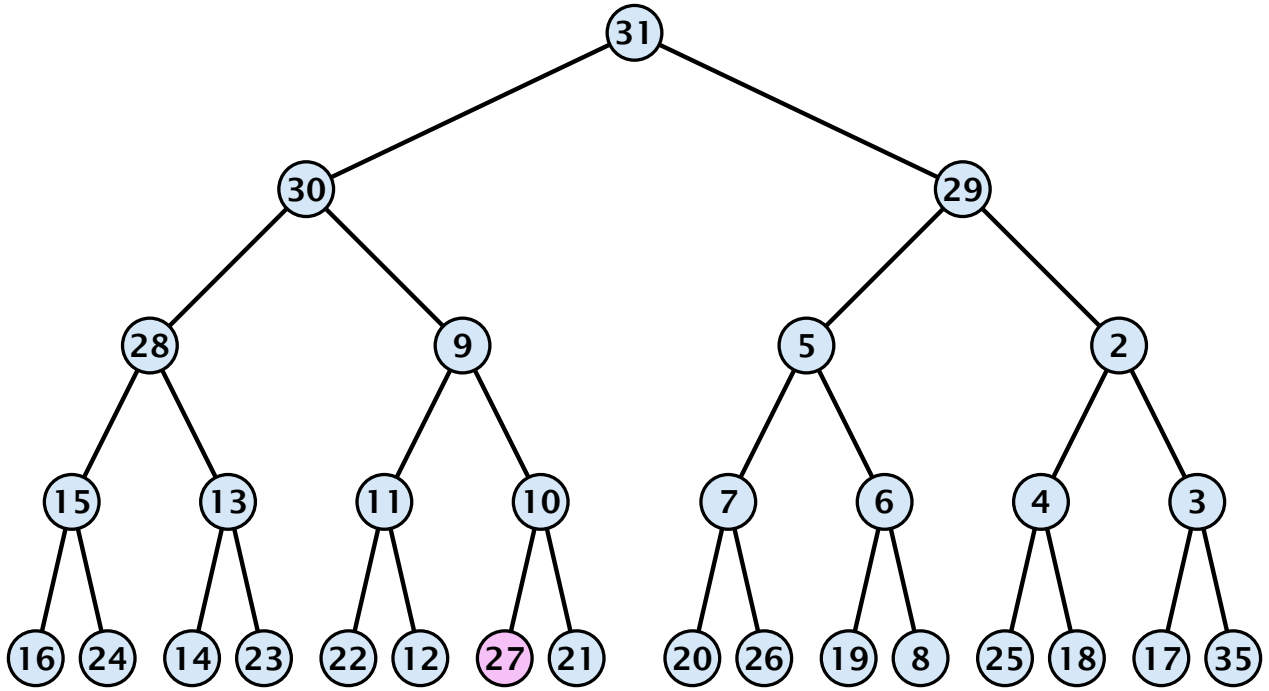
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

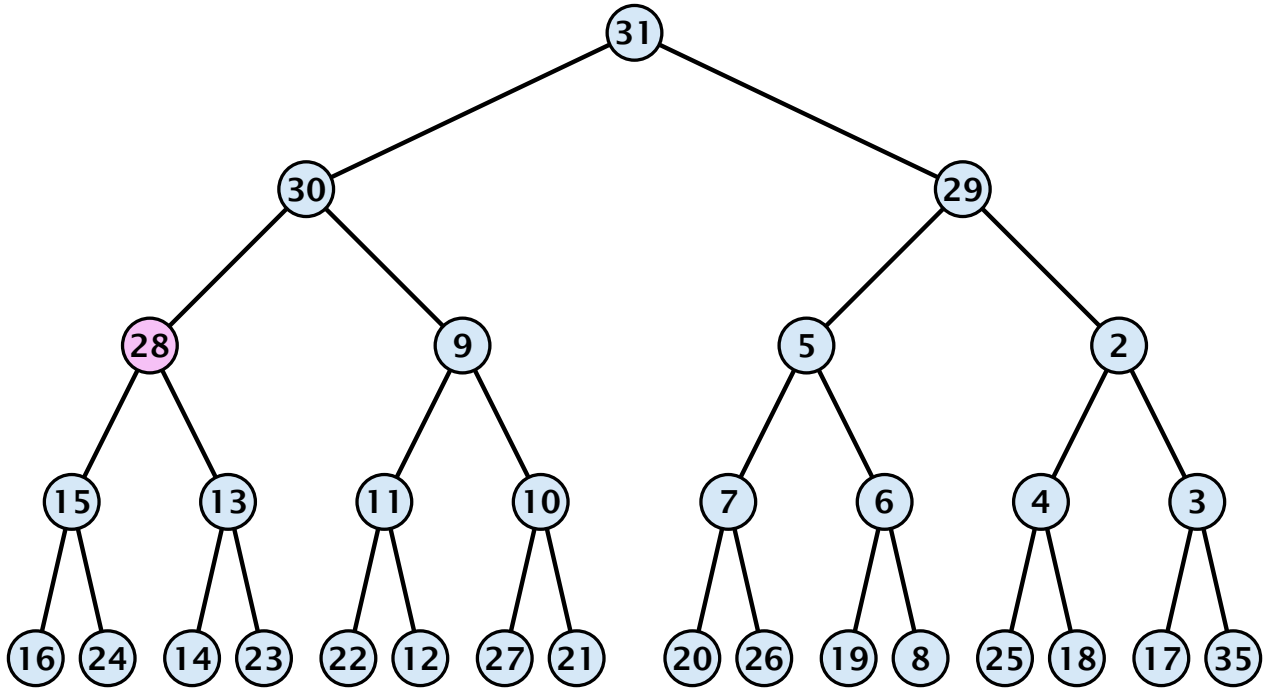
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

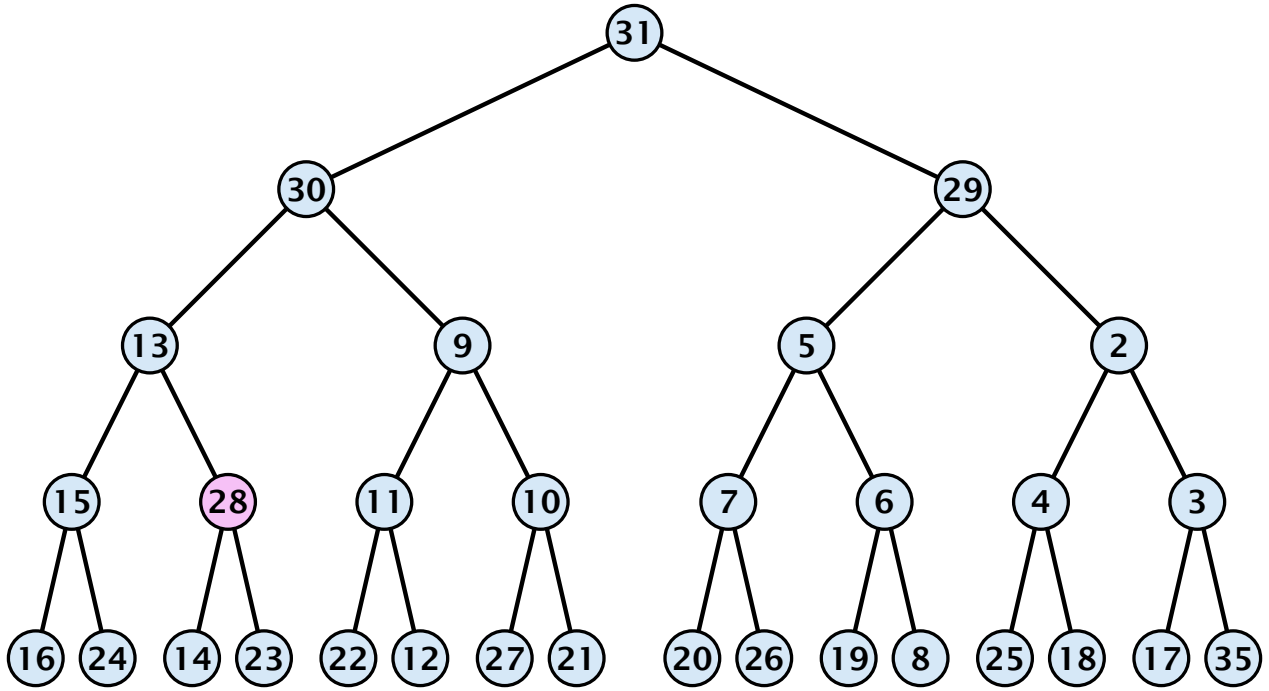
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

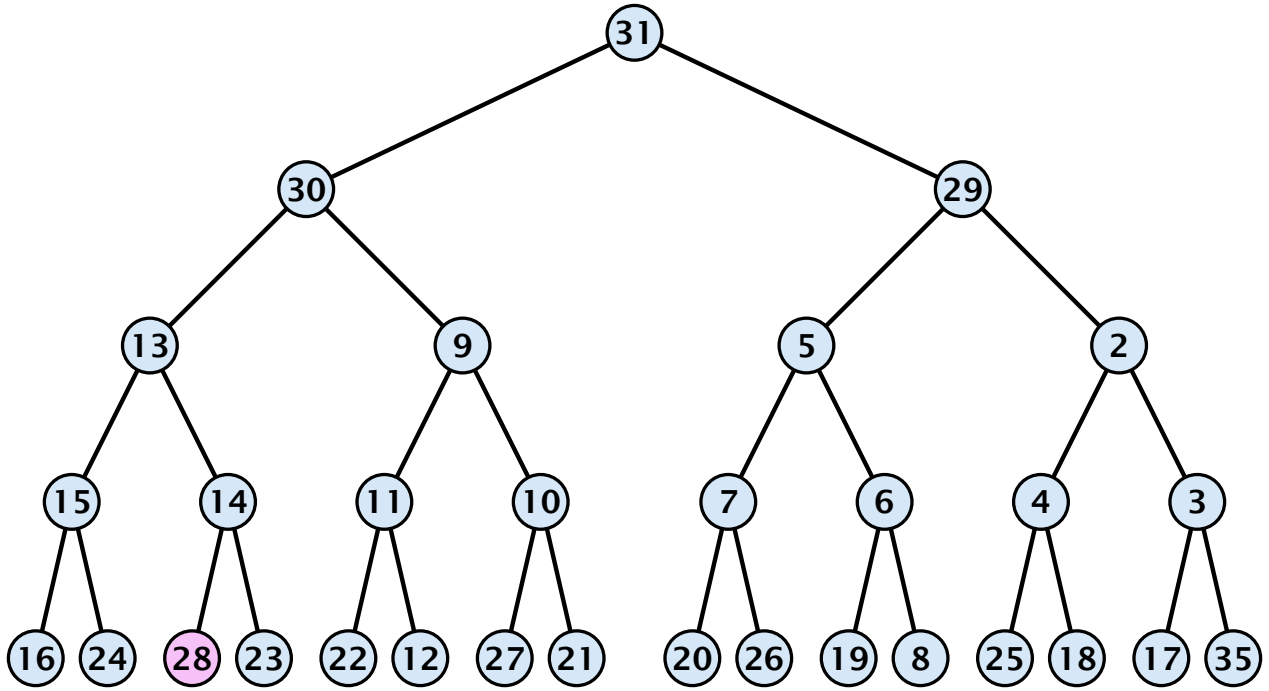
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

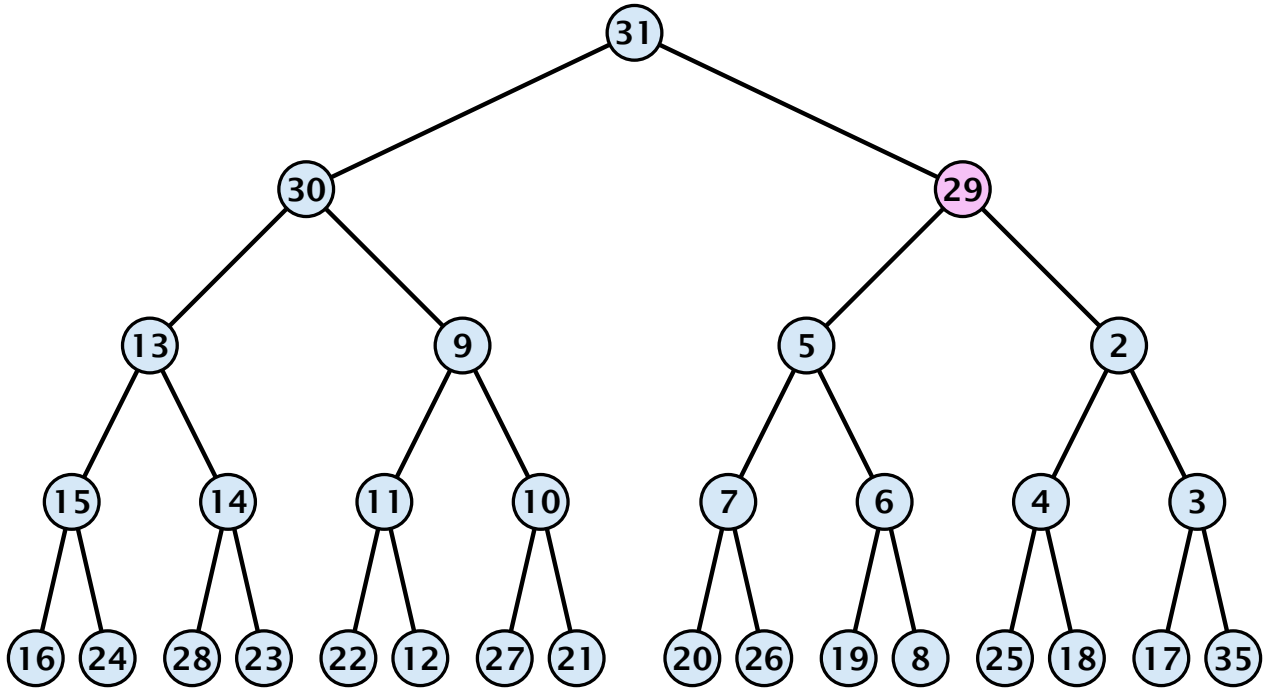
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

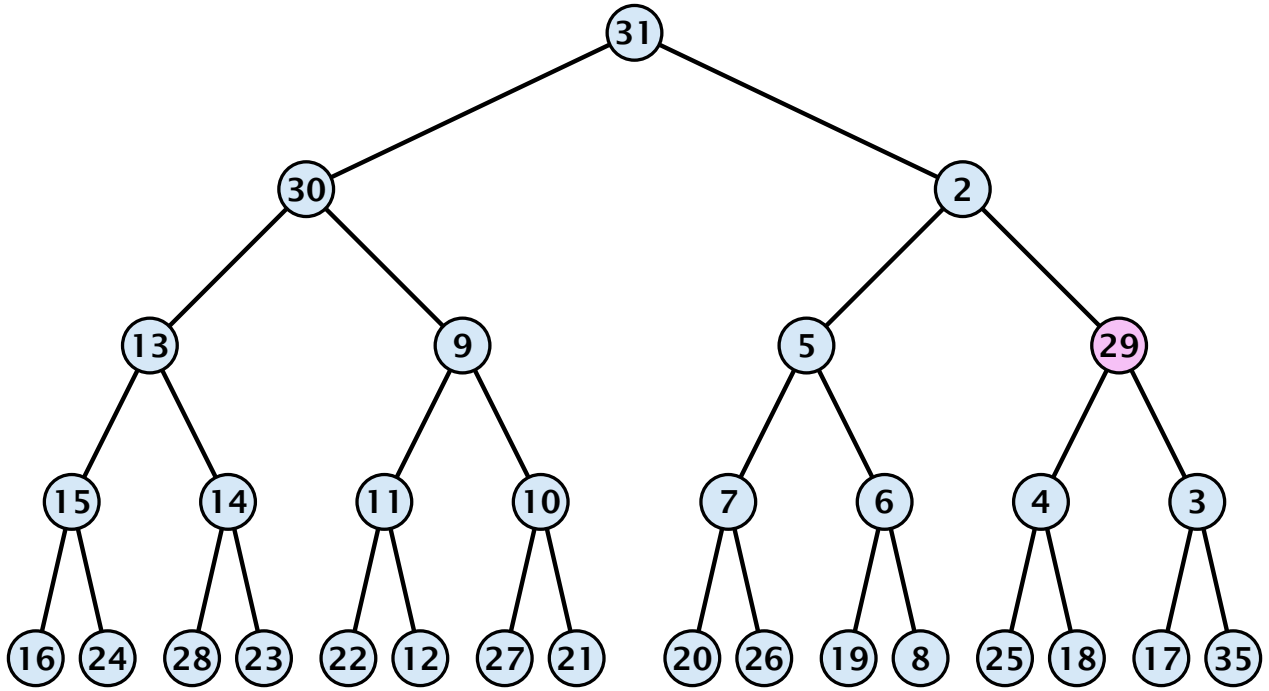
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

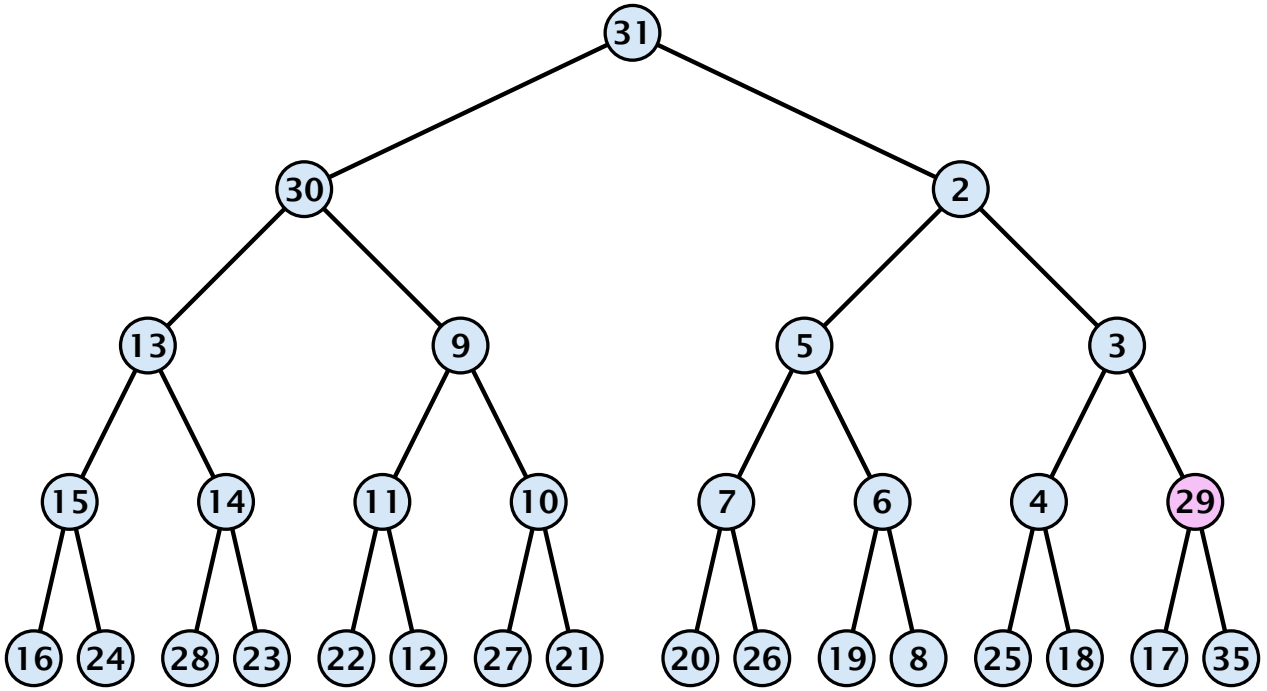
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

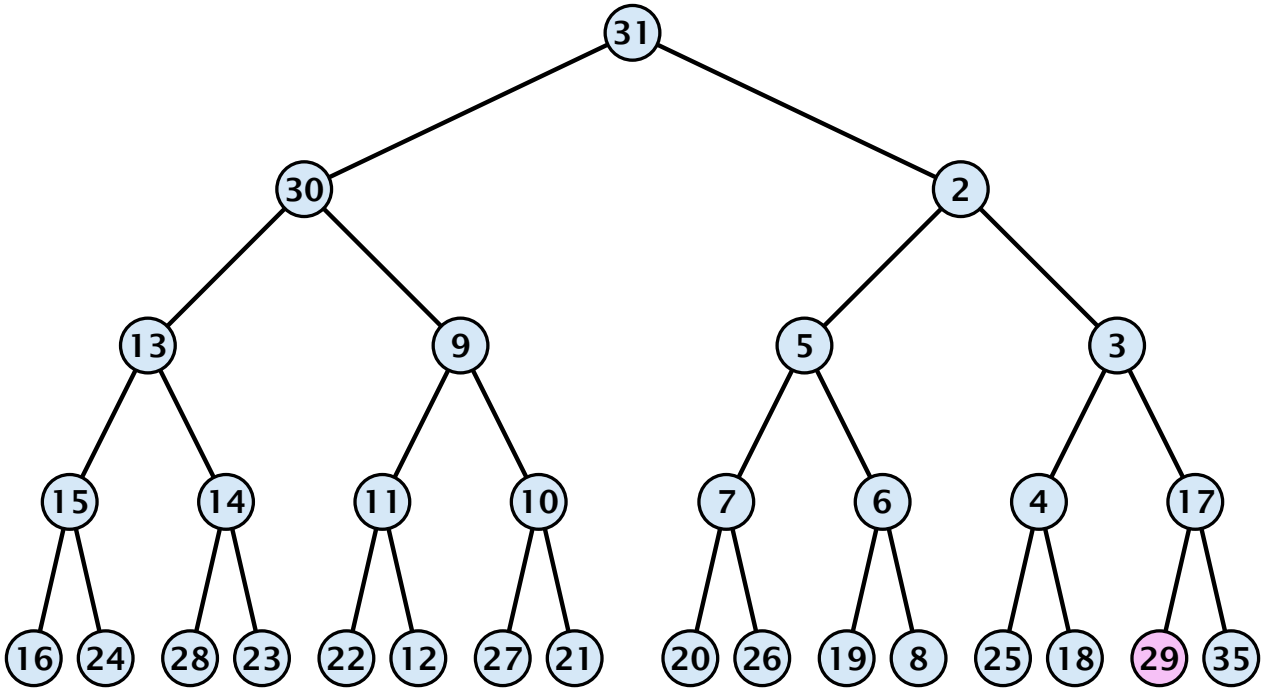
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

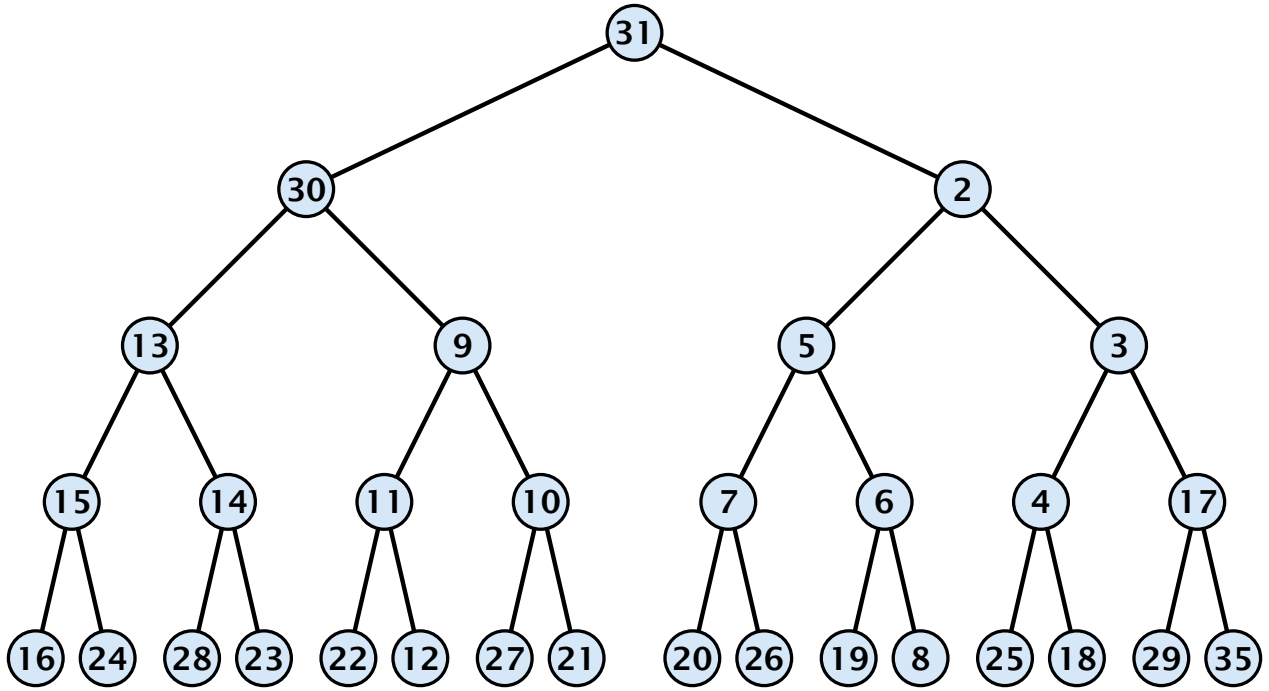
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

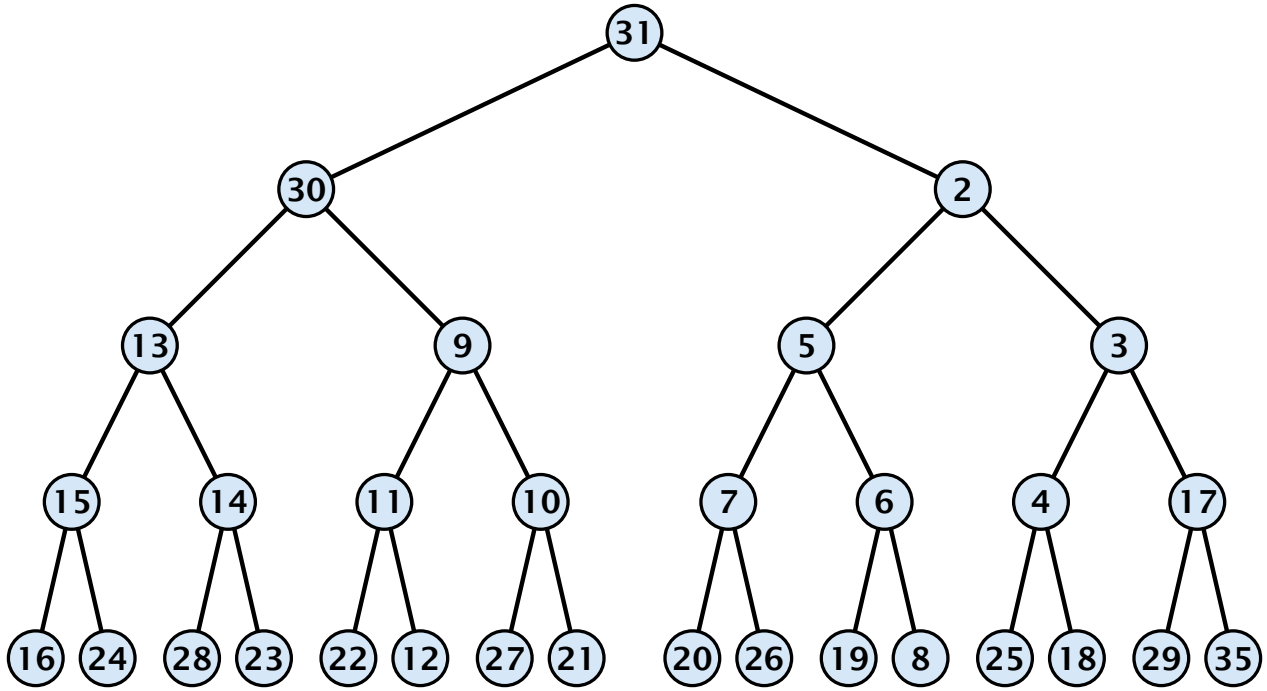
Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

# Build Heap

Wir können einen Heap in Linearzeit erzeugen:




$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Die Analyse auf der vorherigen Folie betrachtet nur die Kosten für die sift-down Operationen.

1. Wie bekommt man alle Schlüssel in eine Baumstruktur?
2. Wie realisiert man die Reihenfolge der sift-down Operationen?

**2.** kann mit Hilfe einer BFS-Suche realisiert werden; man startet eine BFS und verkettet die Baumknoten gemäß der Reihenfolge.

Auch **1.** kann man durch eine BFS-artige Generierung des Baumes erreichen.

```
1 insertCompleteBinary(A, n)
2     Queue q;
3     q.enqueue(&root);
4     for (i=0; i<n; i++)
5         TreeNode*  n = new TreeNode(A[i]);
6         TreeNode** t = q.dequeue();
7         n->parent = *t;
8         q.enqueue(&(n->left));
9         q.enqueue(&(n->right));
```

# Binäre Heaps

**Operationen:**

- **minimum():** Gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** Überprüfe ob Wurzelzeiger NULL. Zeit $\mathcal{O}(1)$.
- **insert($k$):** füge bei Nachfolger von $x$ ein. bubble up. Zeit $\mathcal{O}(\log n)$.
- **delete($h$):** tausche mit $x$; bubble up **or** sift-down. Zeit $\mathcal{O}(\log n)$.
- **build($x_1, \ldots, x_n$):** Füge Elemente beliebig ein; führe sift-down-Operationen durch; starte mit den untersten Leveln. Zeit $\mathcal{O}(n)$.

# Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des $i$-ten Elementes findet man an Position $[\frac{i-1}{2}]$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von $x$ zu finden ist viel einfacher als auf den vorherigen Folien. $x$ wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

# Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des $i$-ten Elementes findet man an Position $\lfloor \frac{i-1}{2} \rfloor$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von $x$ zu finden ist viel einfacher als auf den vorherigen Folien. $x$ wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

# Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des $i$-ten Elementes findet man an Position $\lfloor \frac{i-1}{2} \rfloor$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von $x$ zu finden ist viel einfacher als auf den vorherigen Folien. $x$ wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

# Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des $i$-ten Elementes findet man an Position $\lfloor \frac{i-1}{2} \rfloor$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von $x$ zu finden ist viel einfacher als auf den vorherigen Folien. $x$ wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

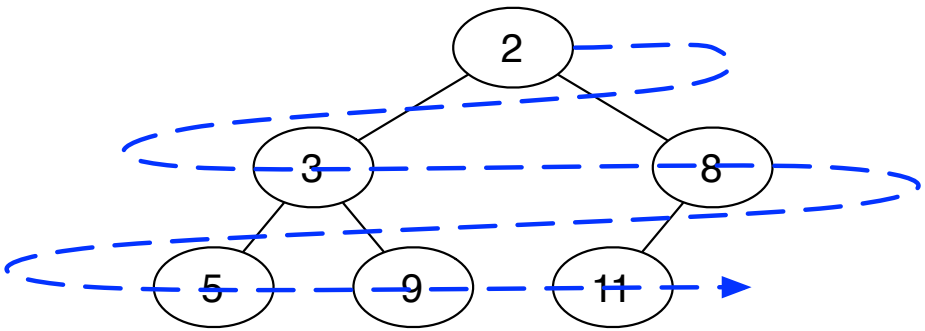
# Binärbaum als Array

- vollständiger Binärbaum Höhe $k$ hat $2^{k+1}-1$ Knoten

  $\Rightarrow$ speichere Knoten von oben nach unten, von links nach rechts in Array

  $\Rightarrow$ maximale Größe des Arrays: $2^{k+1}-1$

- Beispiel fast vollständiger Binärbaum:

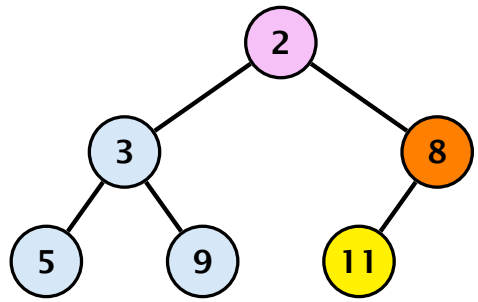


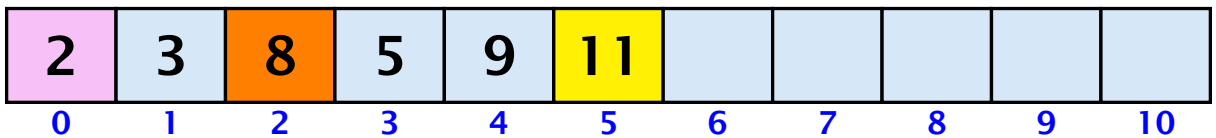
# Binärbaum als Array

Wurzel an Position $0$.

Knoten an Position $i$:

- Elternknoten an Position $\lfloor (i-1)/2 \rfloor$
- linkes Kind an Position $2i+1$;
- rechtes Kind an Position $2i+2$

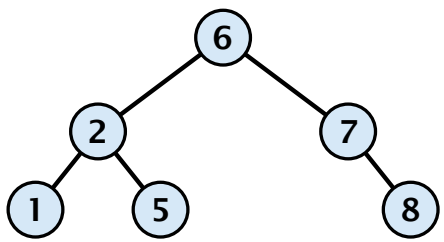


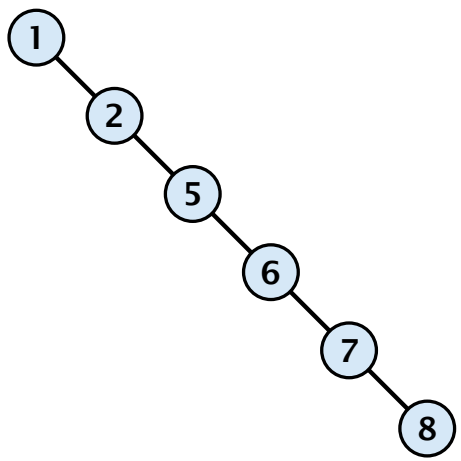



# 7.4 Binäre Suchbäume

Ein binärer Suchbaum speichert Elemente in einem binären Baum. Jeder Baumknoten enthält ein Element. Alle Elemente im linken Teilbaum eines Knotens $v$ haben einen kleineren Schlüssel als $\text{key}[v]$; Elemente im rechten Teilbaum haben einen größeren Schlüssel. (Annahme: alle Schlüssel sind unterschiedlich).

**Beispiele:**



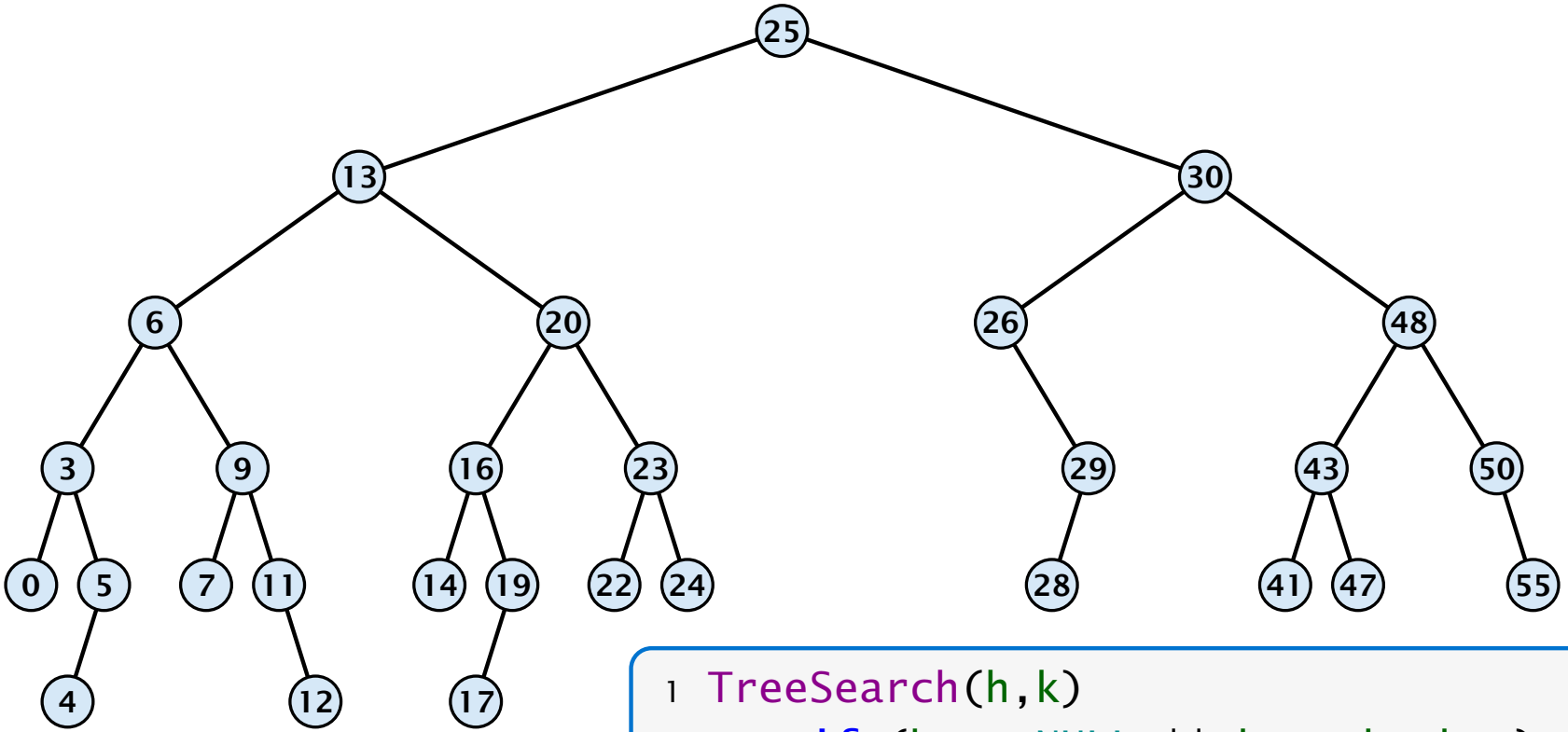



# 7.4 Binäre Suchbäume

Das Handle ist üblicherweise ein Zeiger auf den Baumknoten, der das Objekt enthält.

- **`T.insert(x)`** Fügt Objekt `x` ein; `T` darf kein Objekt mit Schlüssel `key[x]` enthalten.
- **`T.delete(h)`** Entfernt durch handle `h` referenziertes Objekt aus `T`.
- **`T.search(k)`** Gibt `handle` auf in `T` gespeichertes Objekt mit Schlüssel `k` zurück falls existent. Sonst `NULL`.
- **`T.successor(h)`** Gibt `handle` auf Nachfolger von durch `h` referenziertes Objekt zurück; falls existent. Sonst `NULL`.
- **`T.predecessor(h)`** Gibt `handle` auf Vorgänger von durch `h` referenziertes Objekt zurück; falls existent. Sonst `NULL`.
- **`T.minimum()`** Gibt `handle` auf in `T` gespeichertes Objekt mit kleinstem Schlüssel zurück falls existent. Sonst `NULL`.
- **`T.maximum()`** Gibt `handle` auf in `T` gespeichertes Objekt mit größtem Schlüssel zurück falls existent. Sonst `NULL`.

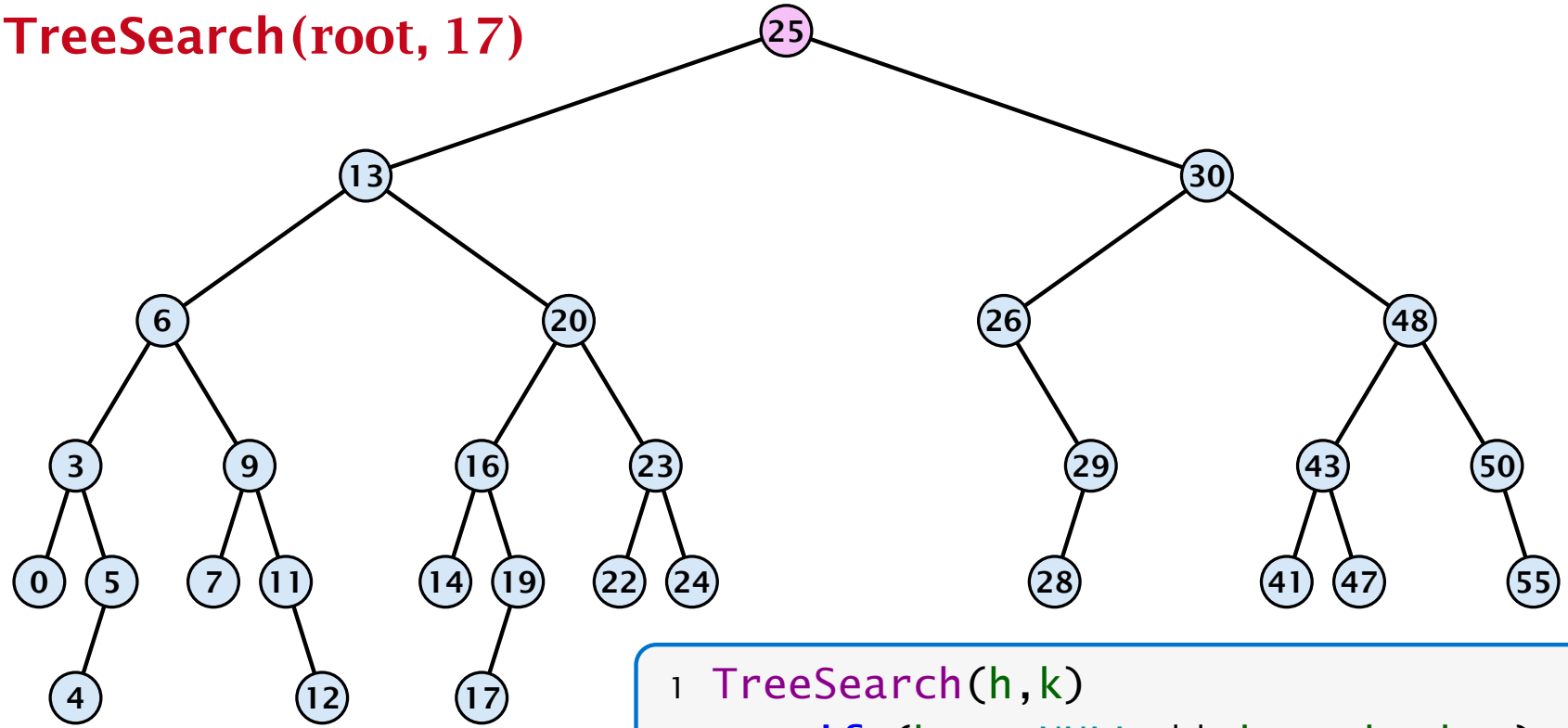
# Binäre Suchbäume: Suchen




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

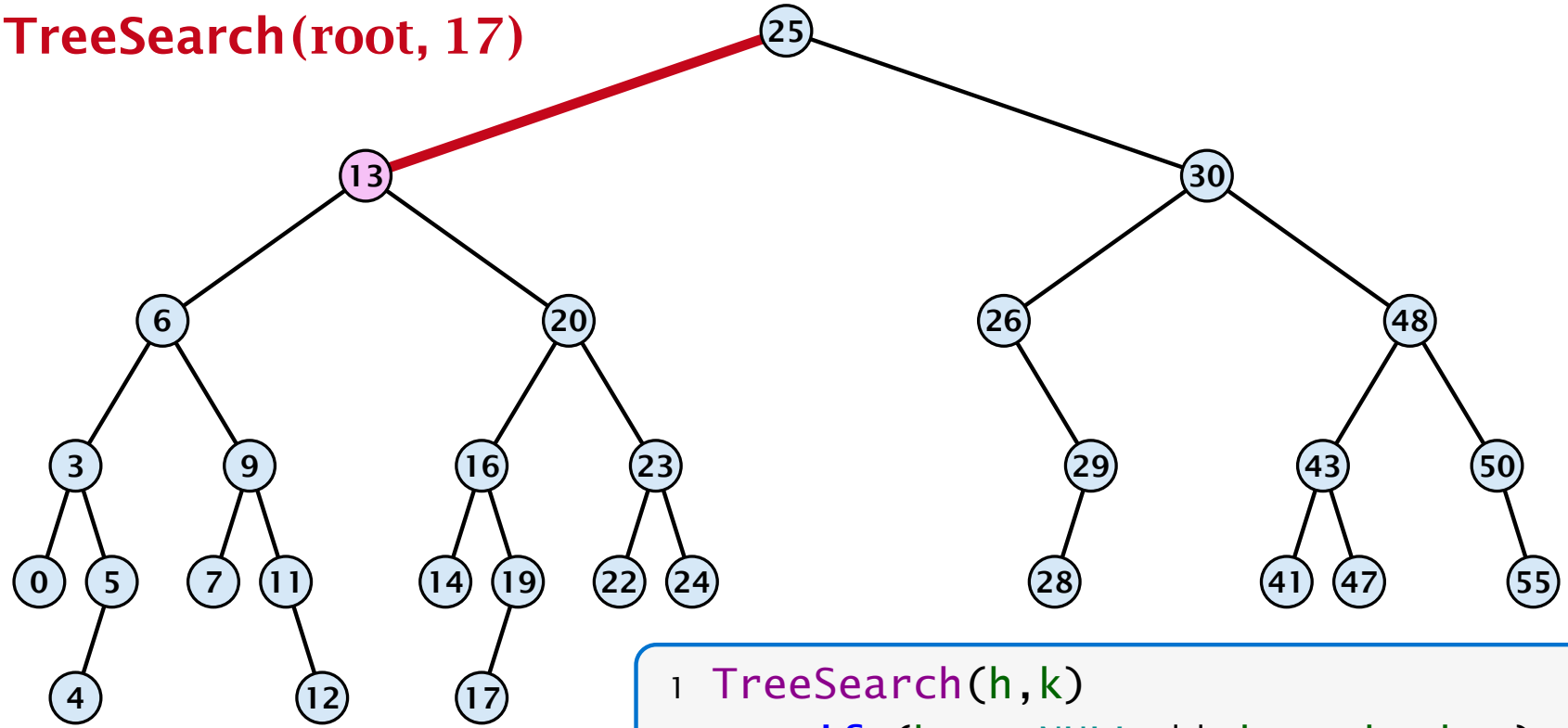
**TreeSearch(root, 17)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

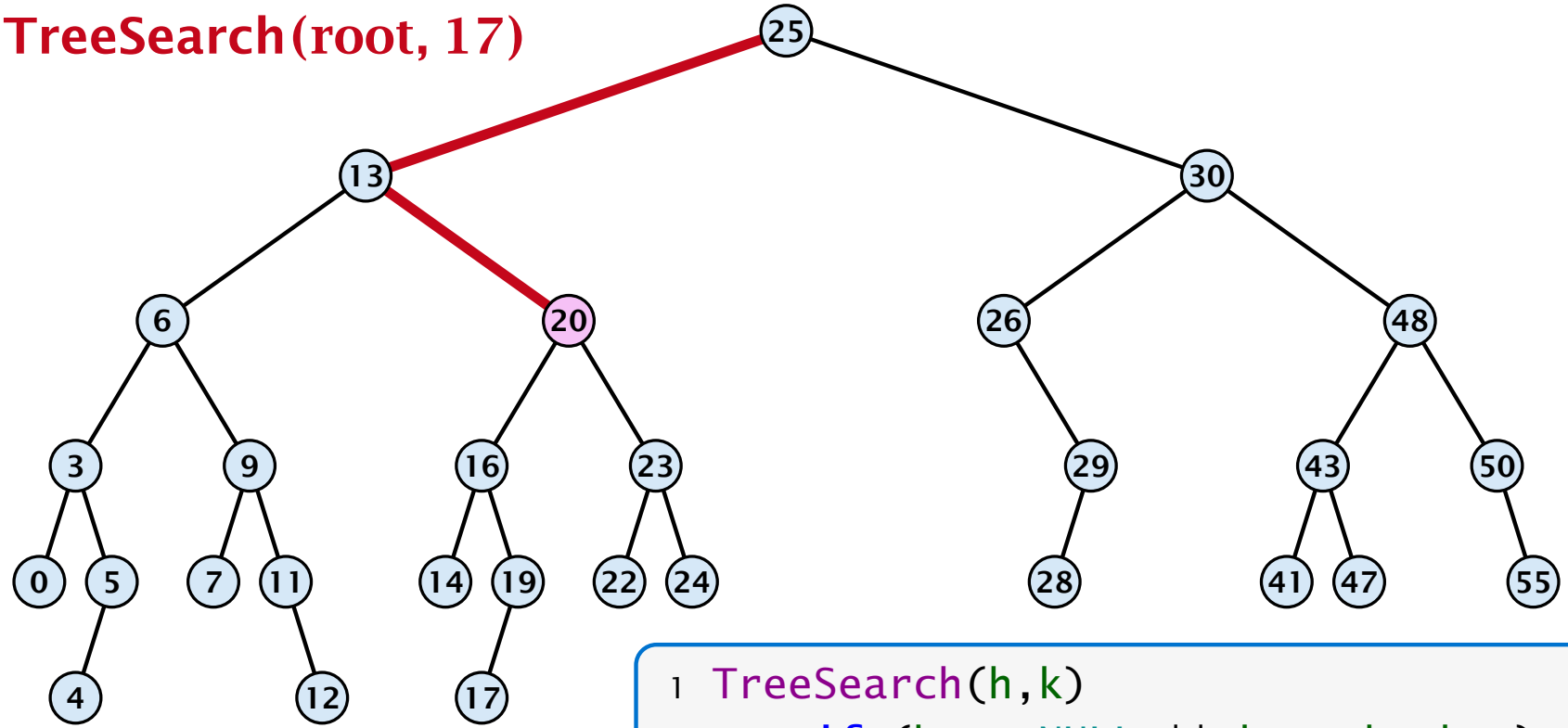
**TreeSearch(root, 17)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

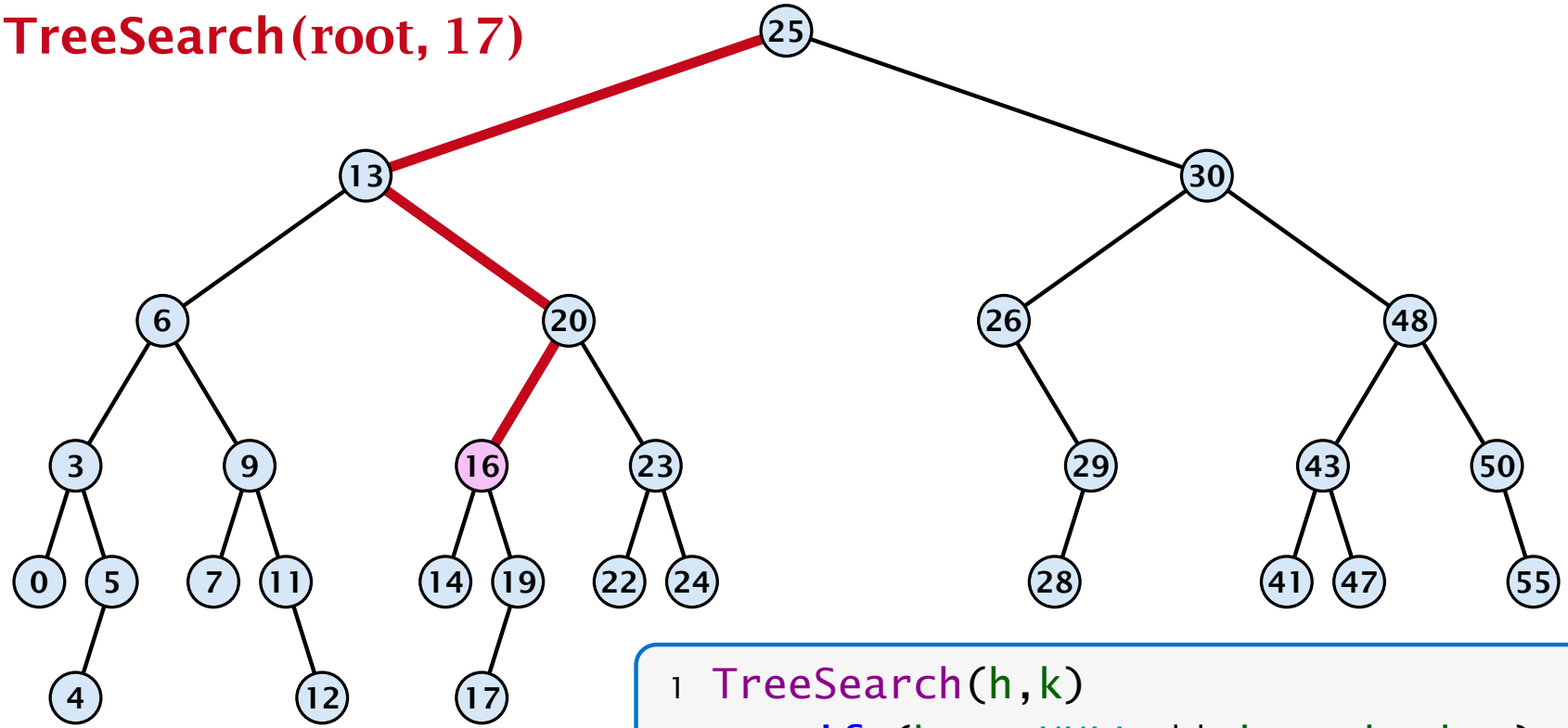
**TreeSearch(root, 17)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

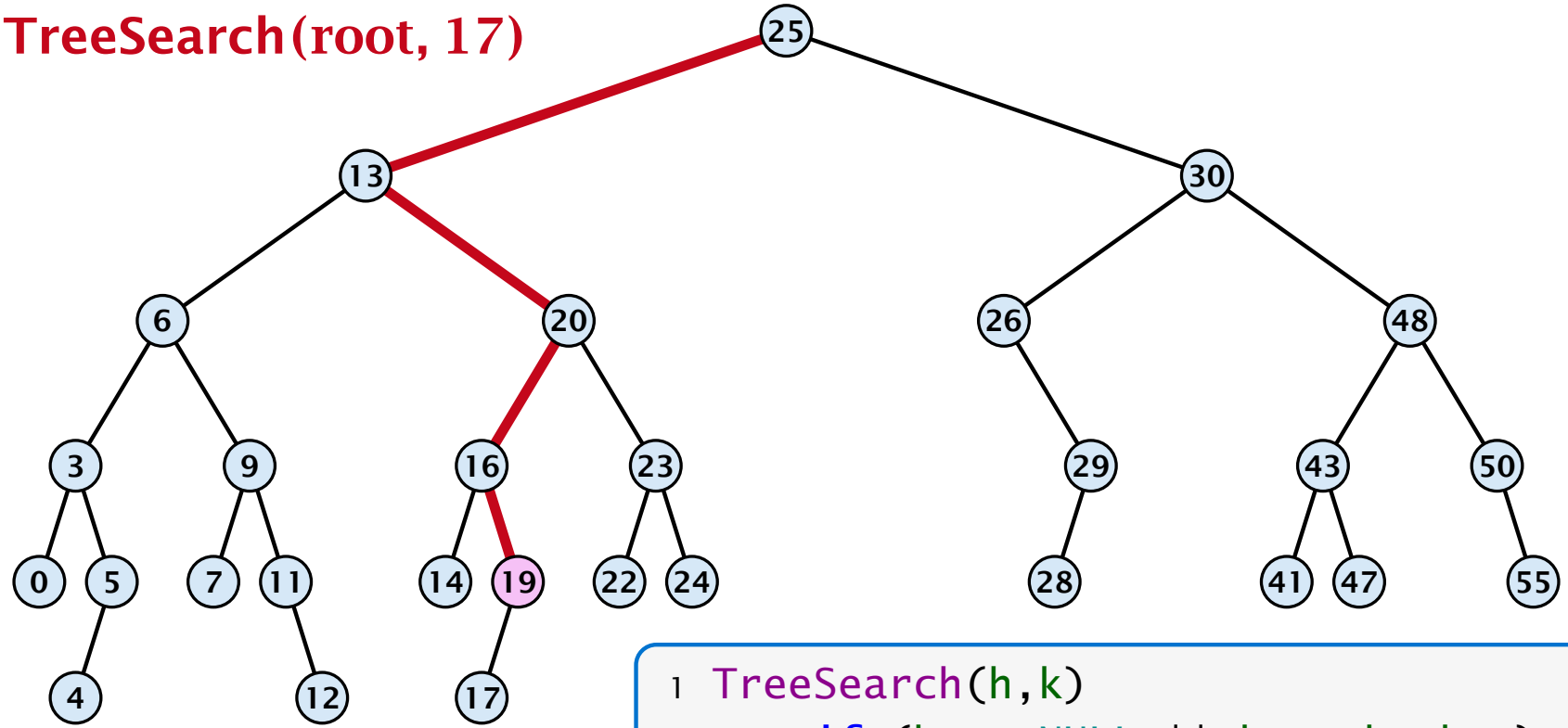
**TreeSearch(root, 17)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

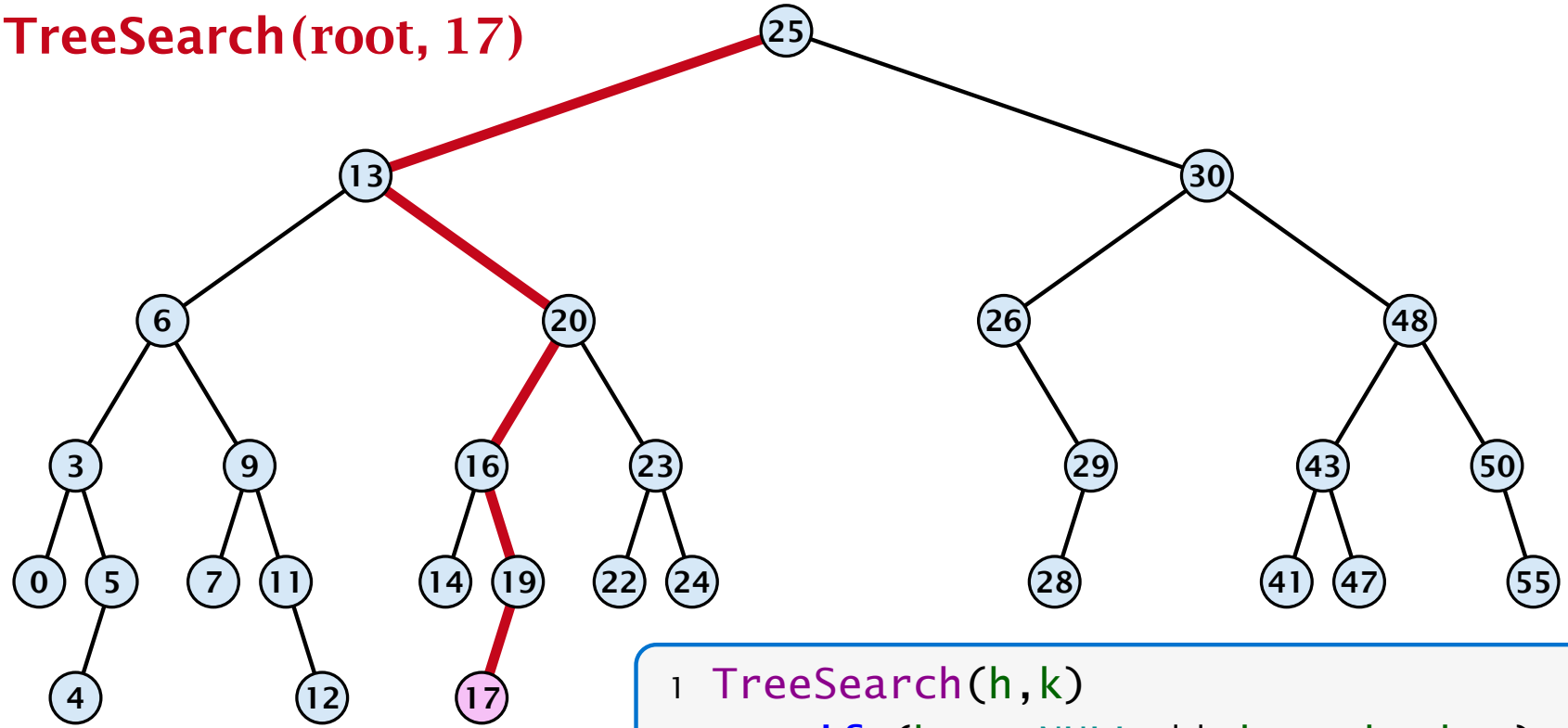
**TreeSearch(root, 17)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

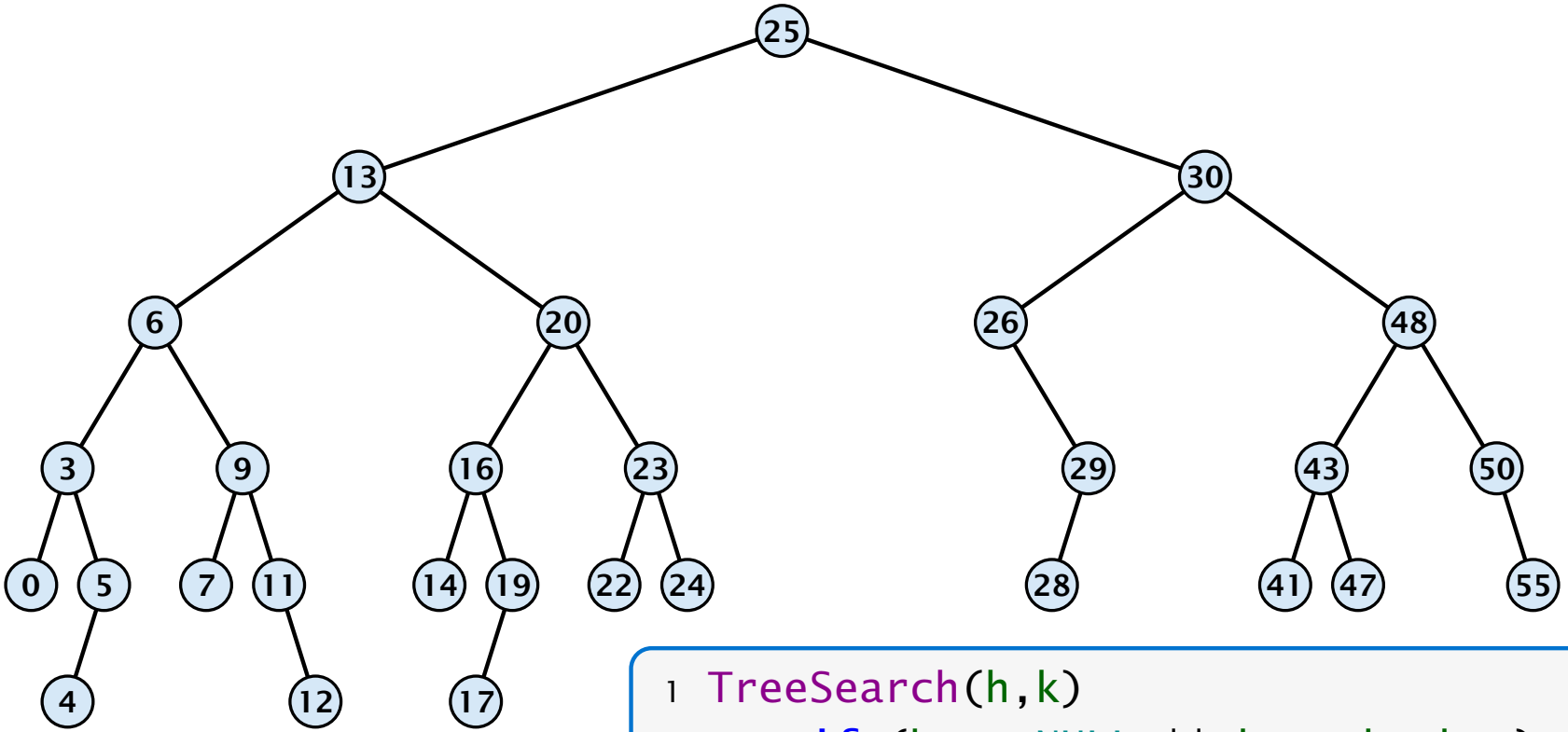
# Binäre Suchbäume: Suchen

**TreeSearch(root, 17)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

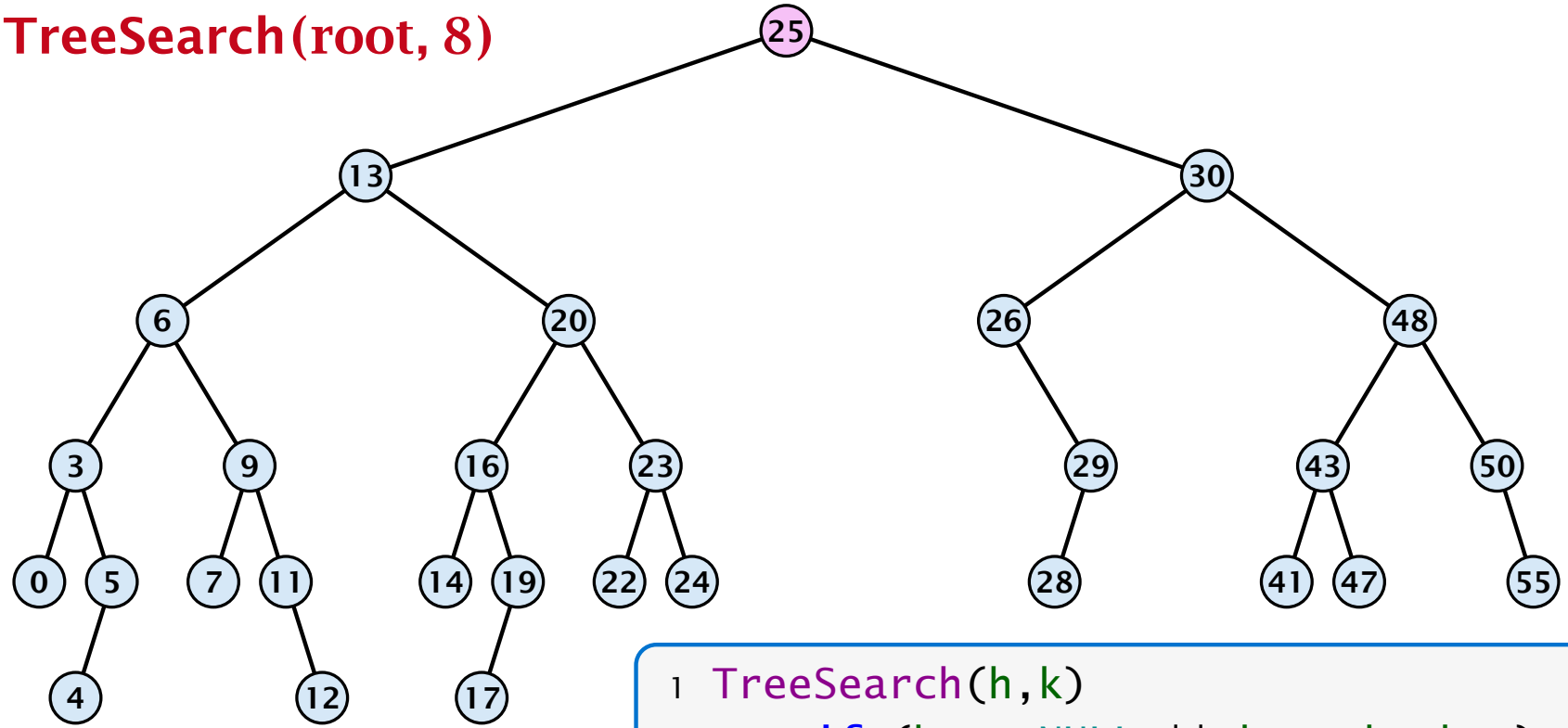
# Binäre Suchbäume: Suchen




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

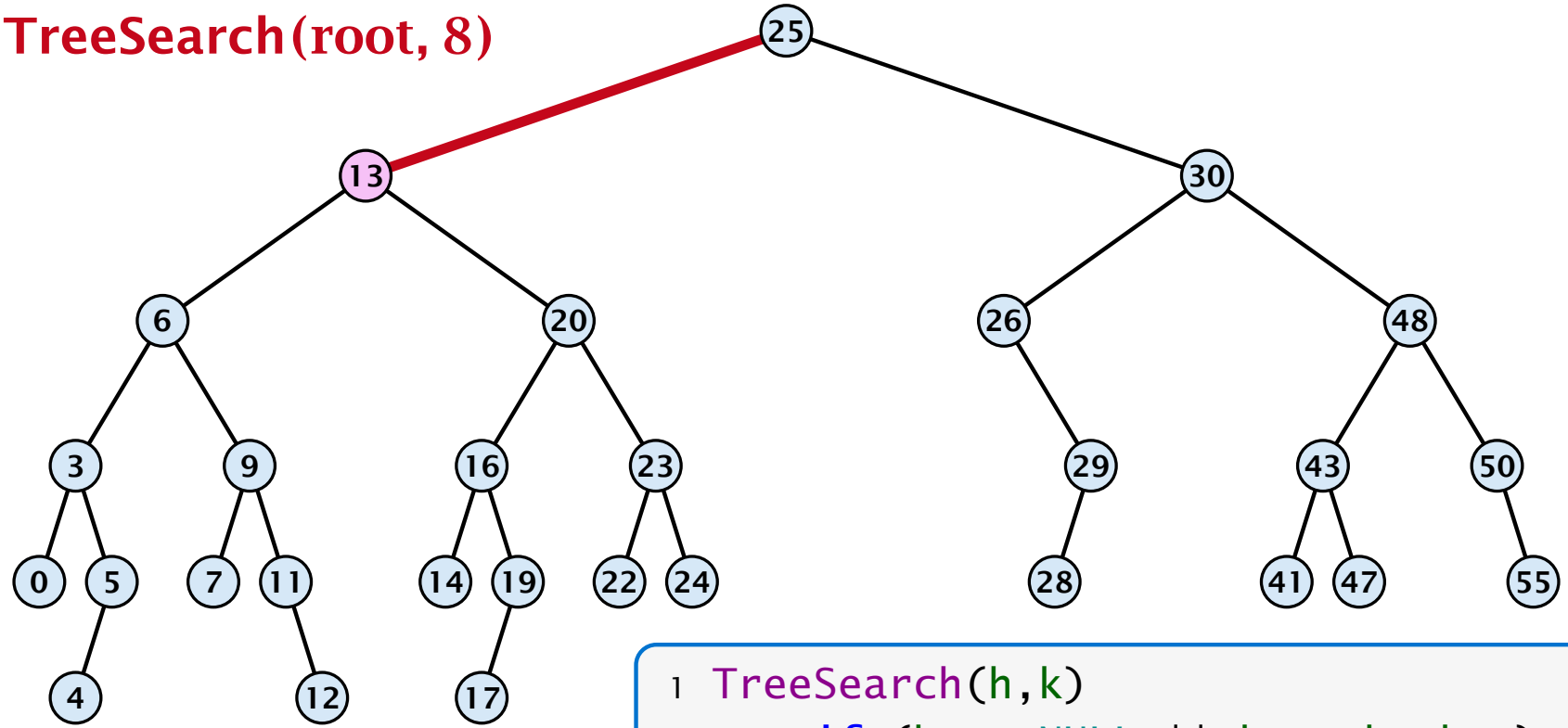
**TreeSearch(root, 8)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

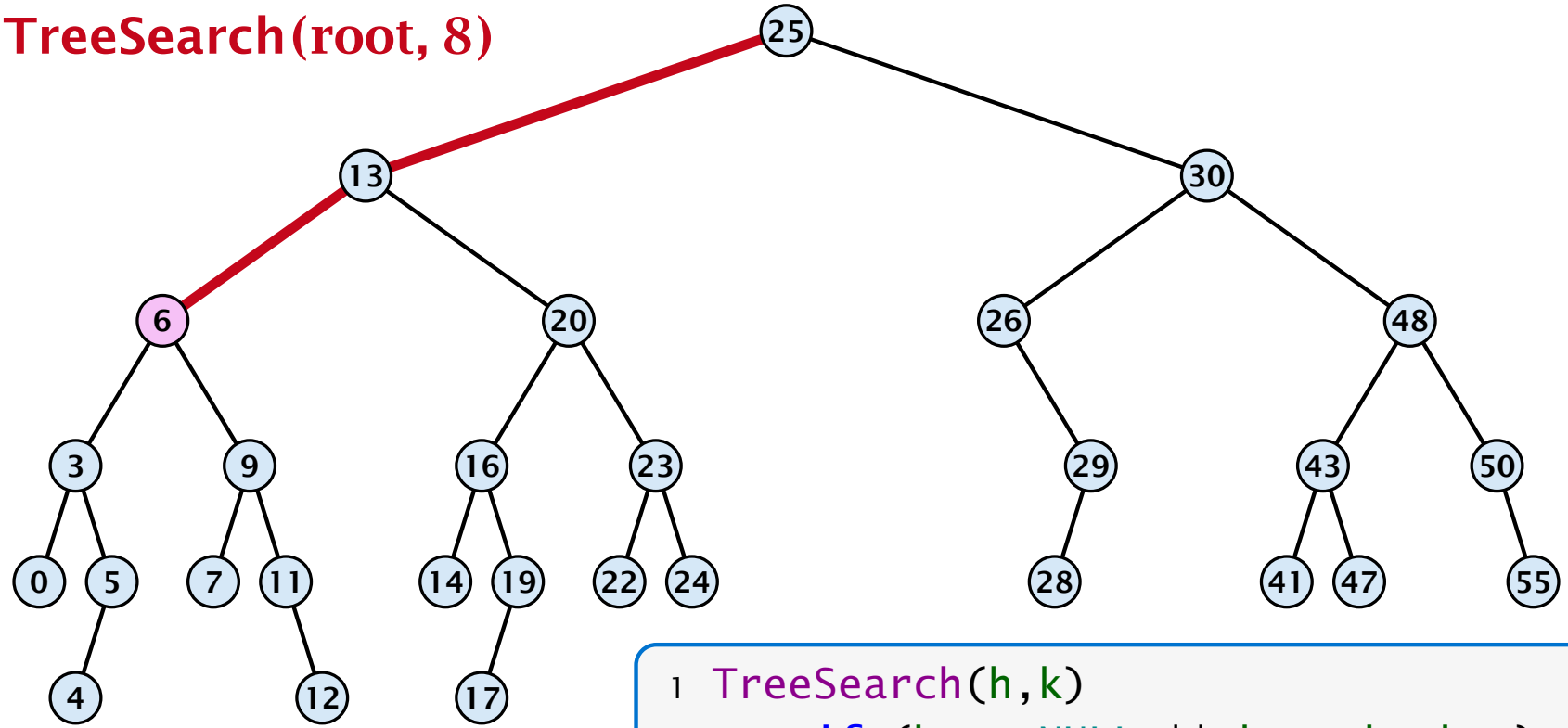
**TreeSearch(root, 8)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

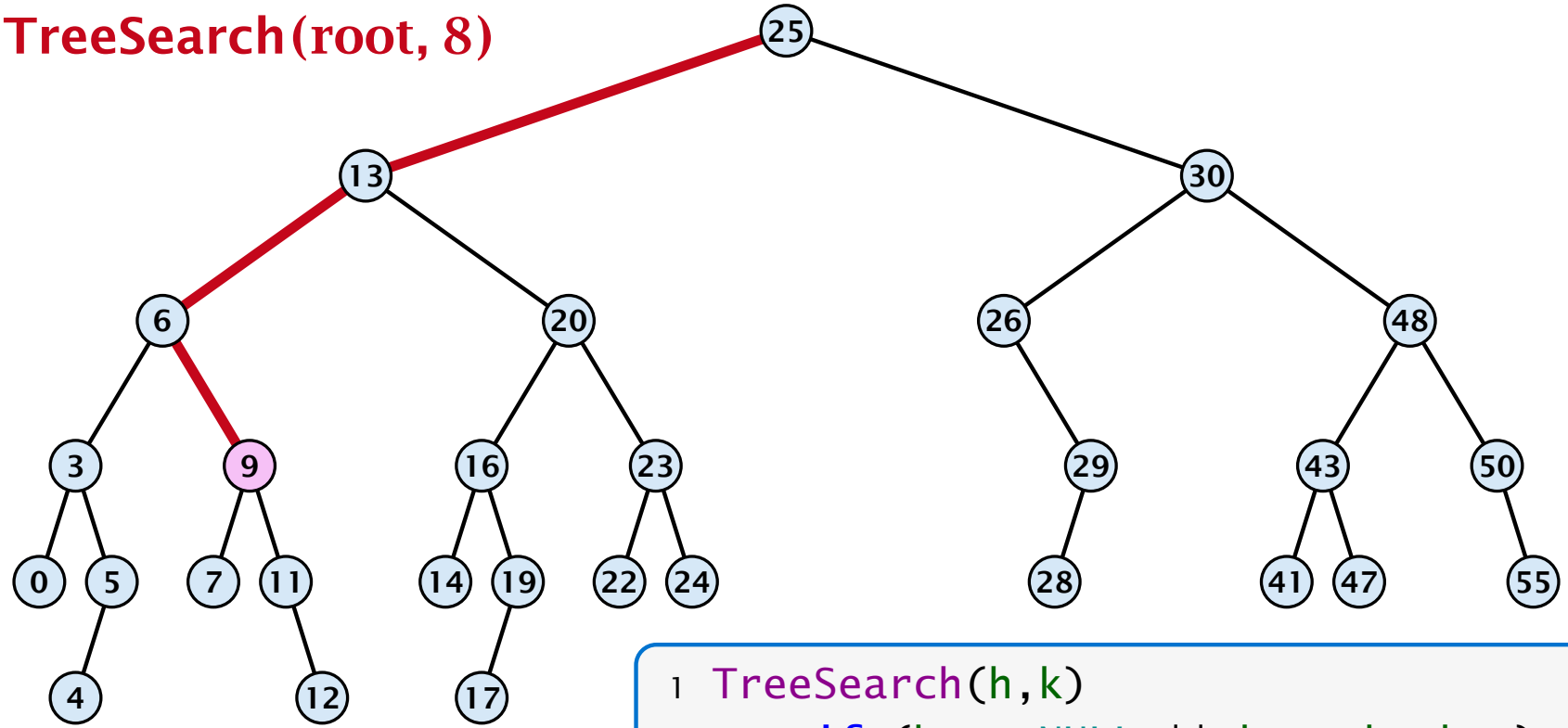
**TreeSearch(root, 8)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

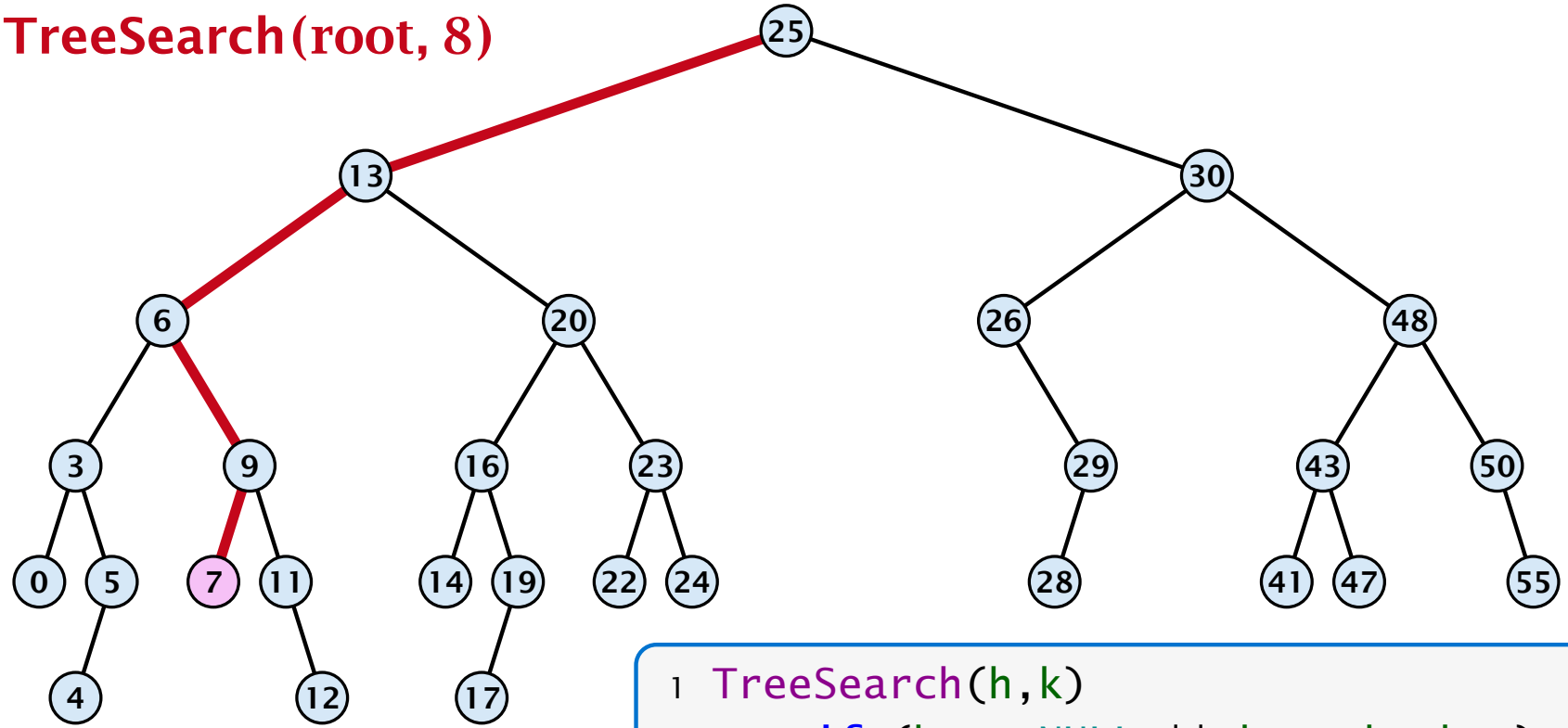
**TreeSearch(root, 8)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Suchen

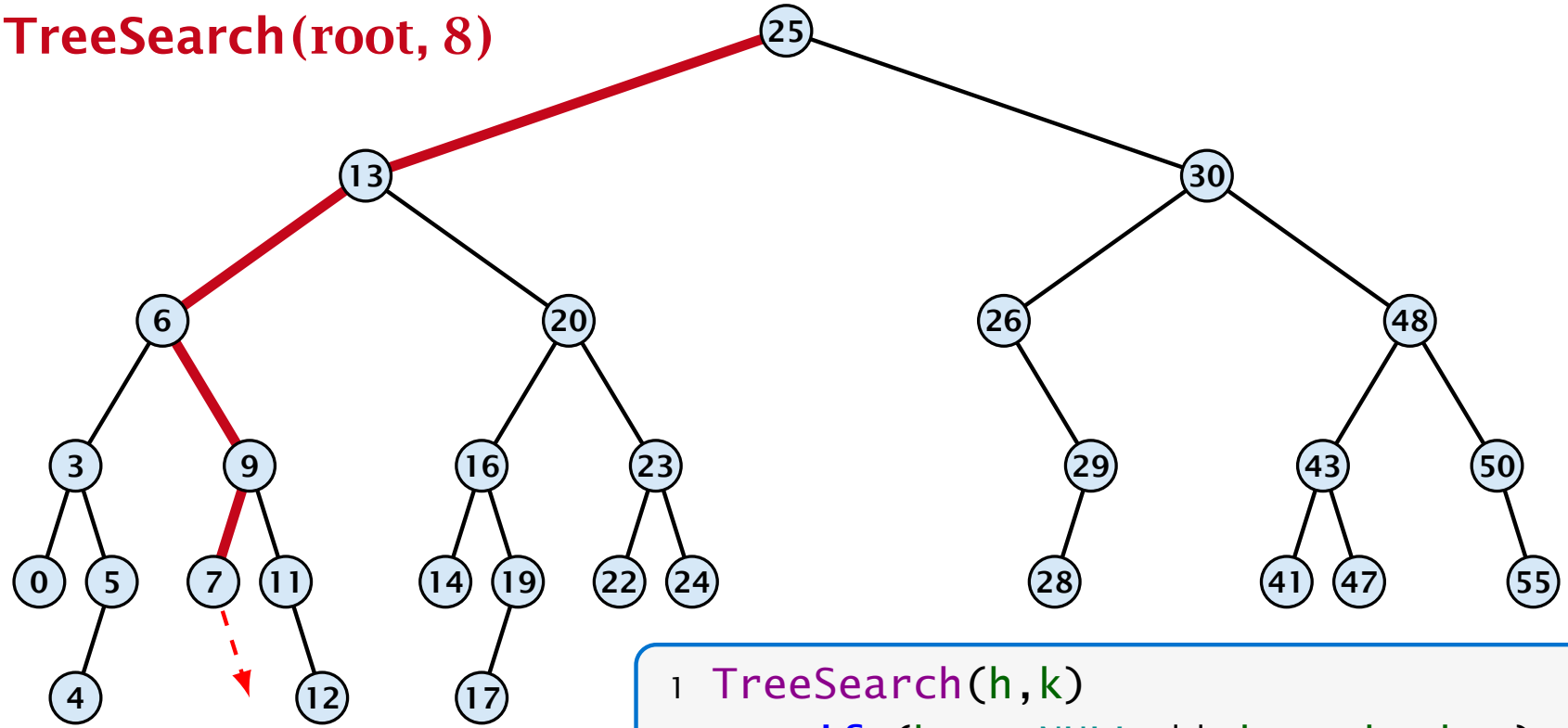
**TreeSearch(root, 8)**




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

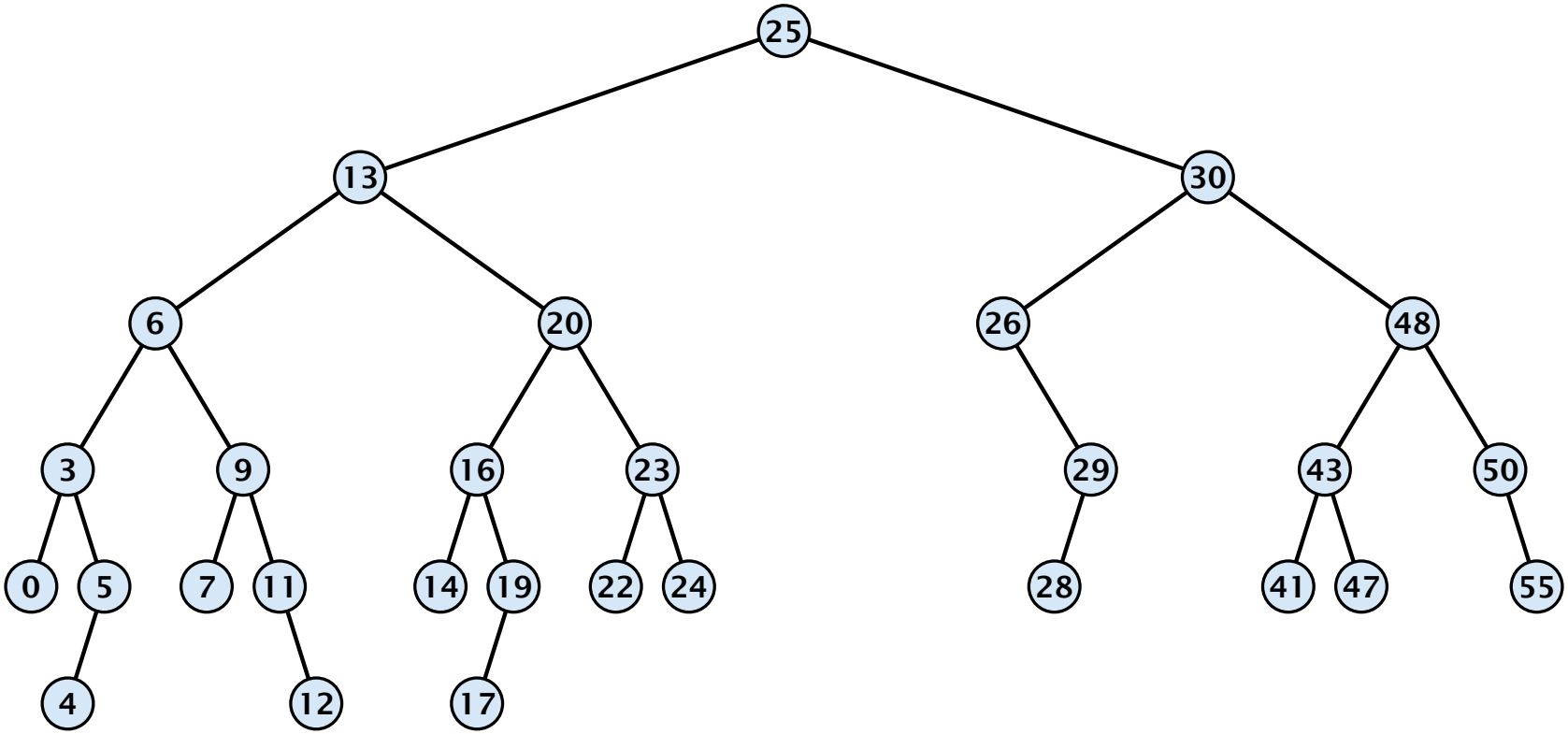
# Binäre Suchbäume: Suchen

**TreeSearch(root, 8)**



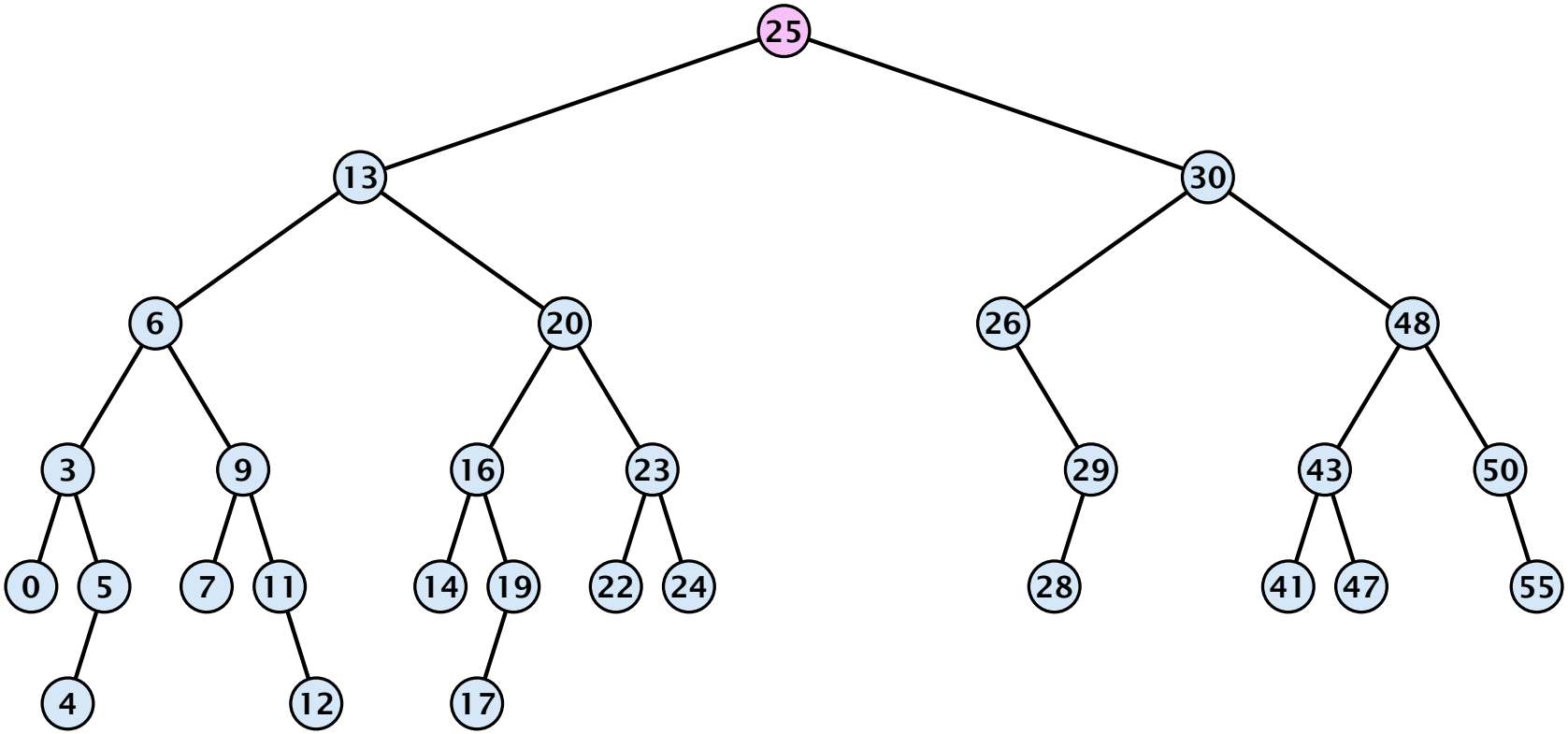

```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

# Binäre Suchbäume: Minimum



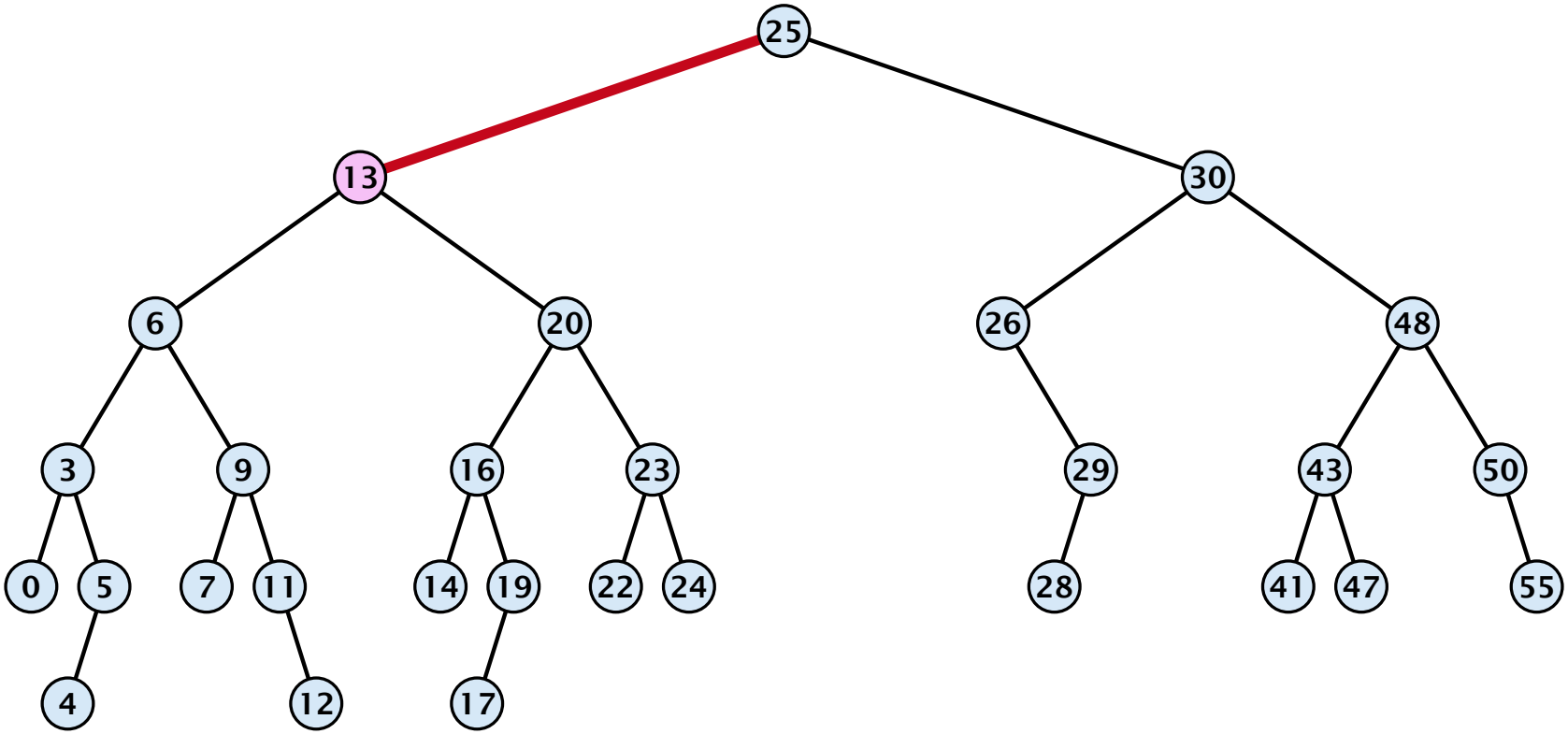

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

# Binäre Suchbäume: Minimum



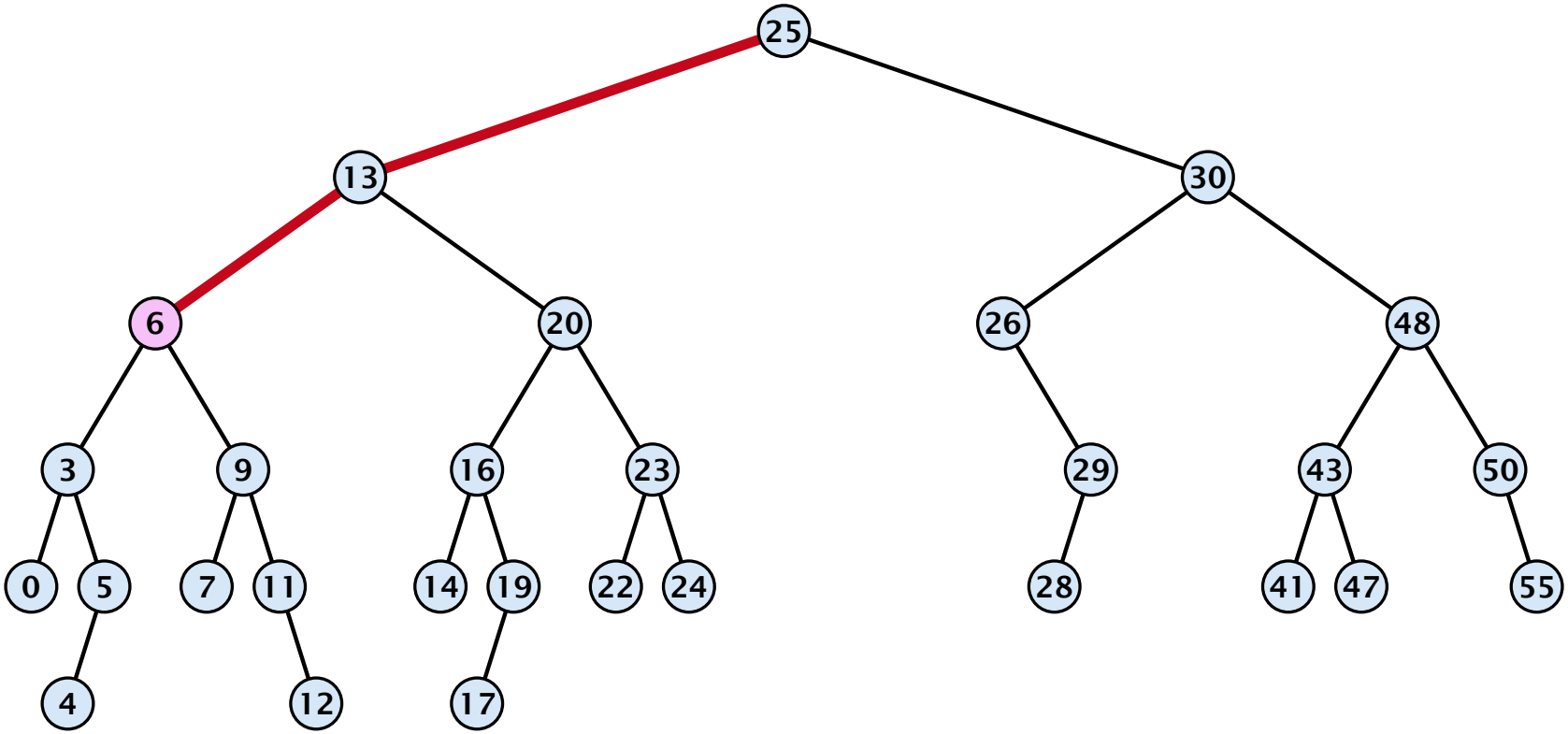

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

# Binäre Suchbäume: Minimum



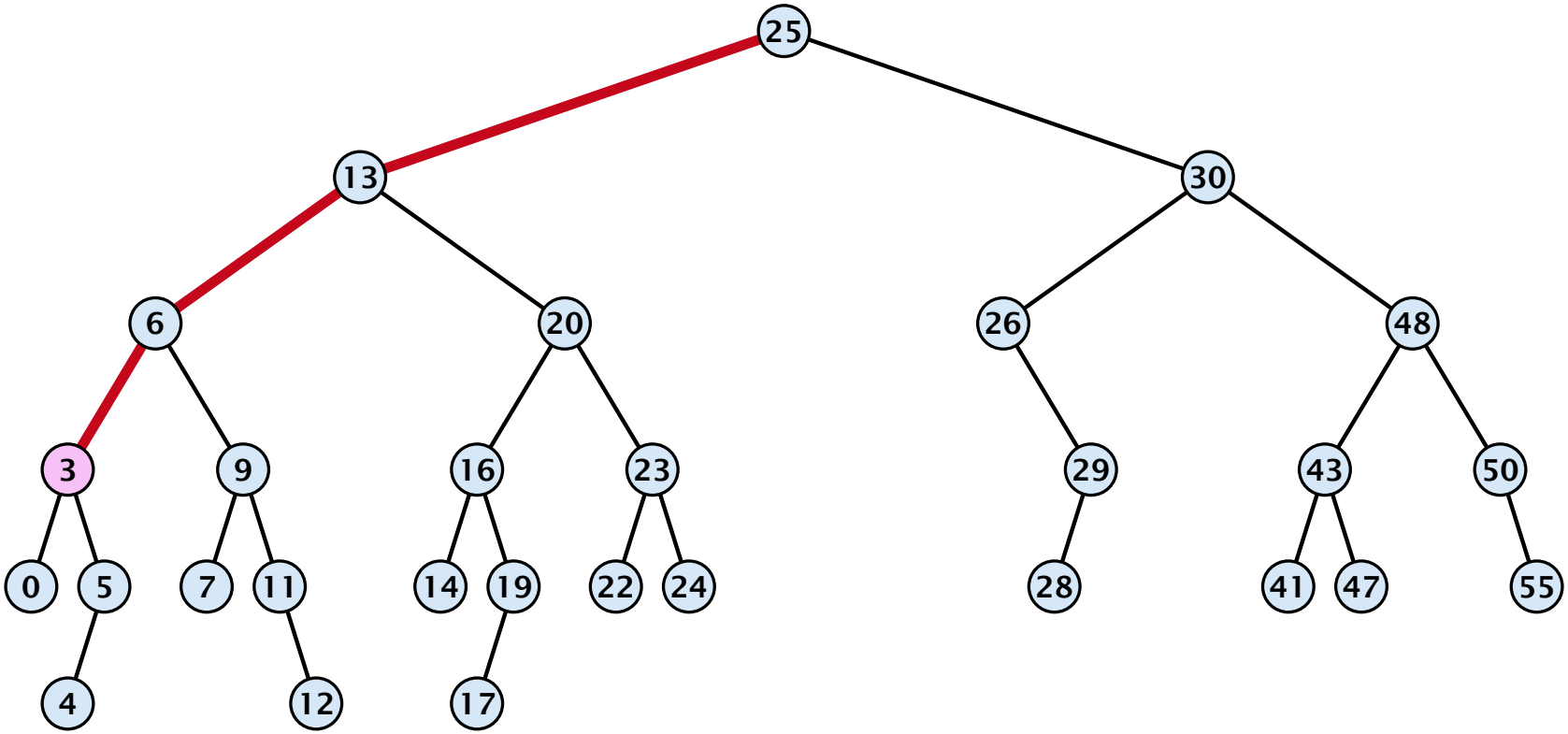

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

# Binäre Suchbäume: Minimum



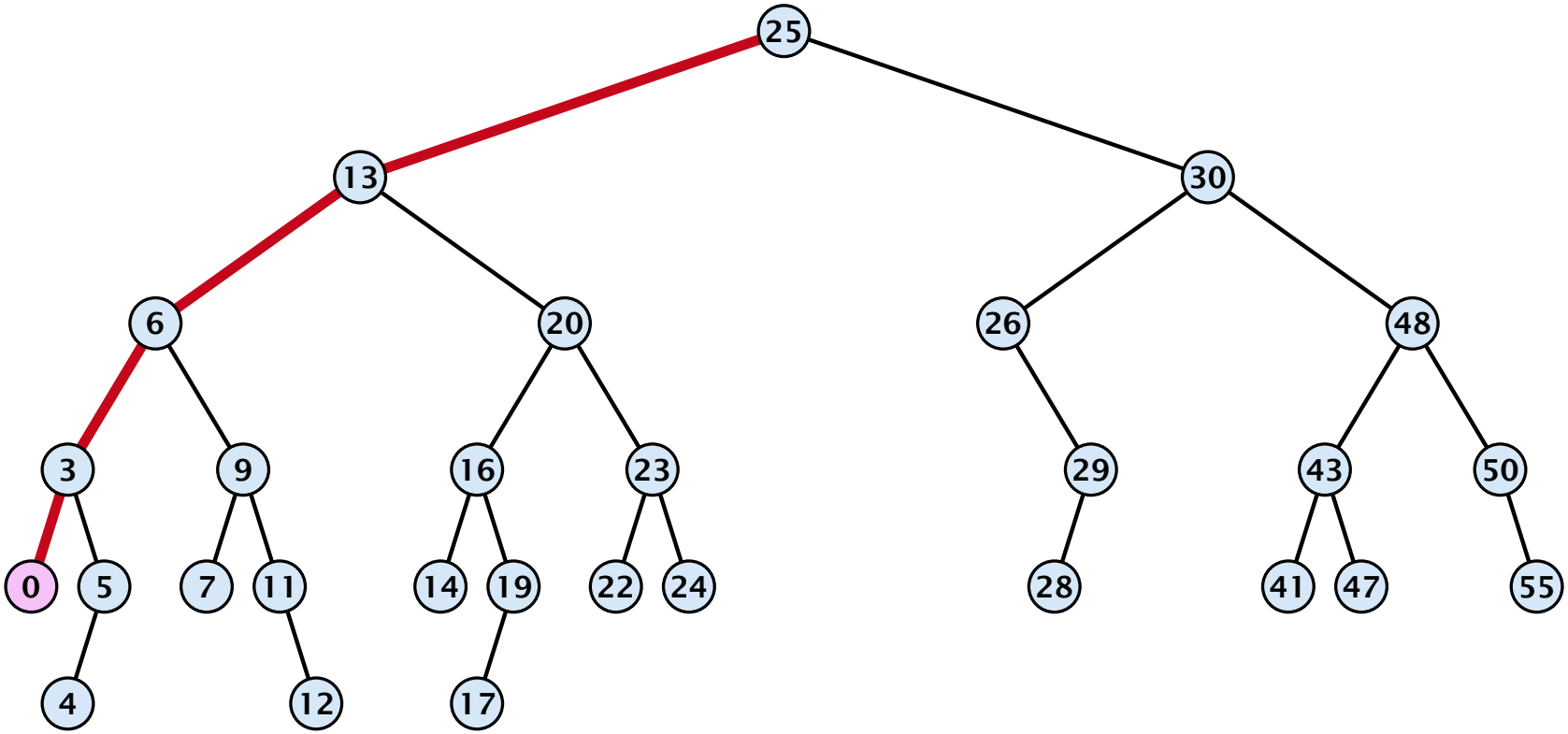

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

# Binäre Suchbäume: Minimum



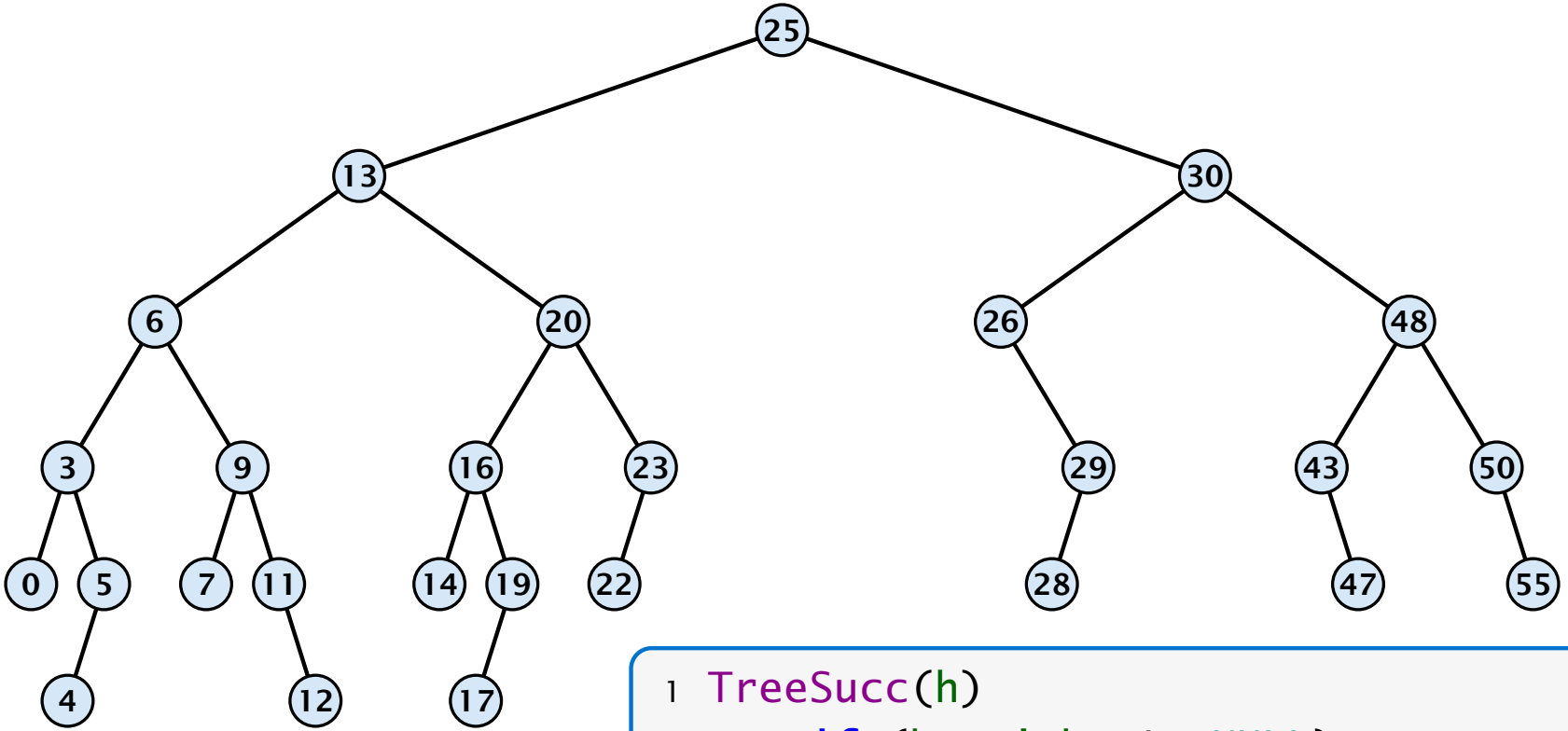

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

# Binäre Suchbäume: Minimum



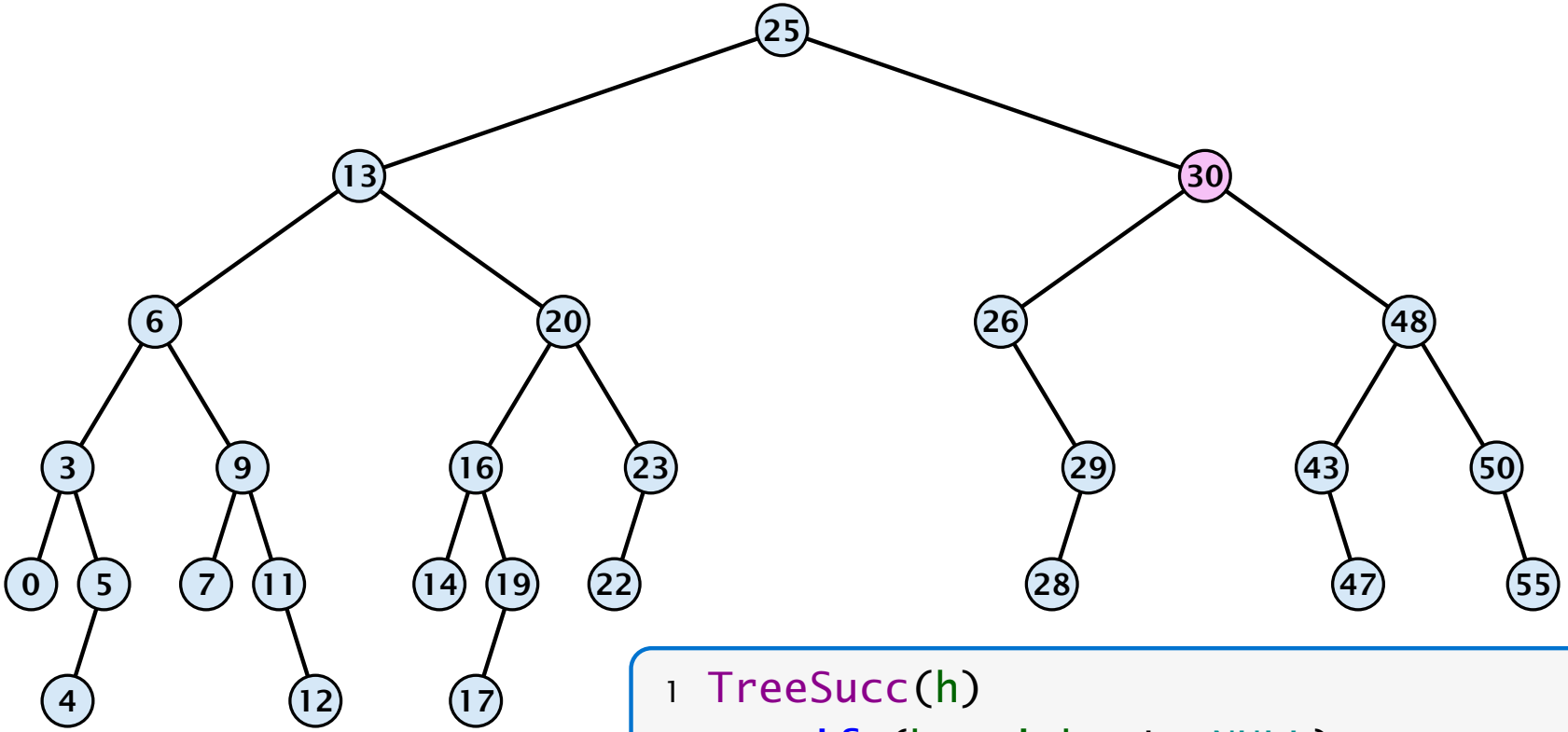

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

# Binäre Suchbäume: Nachfolger



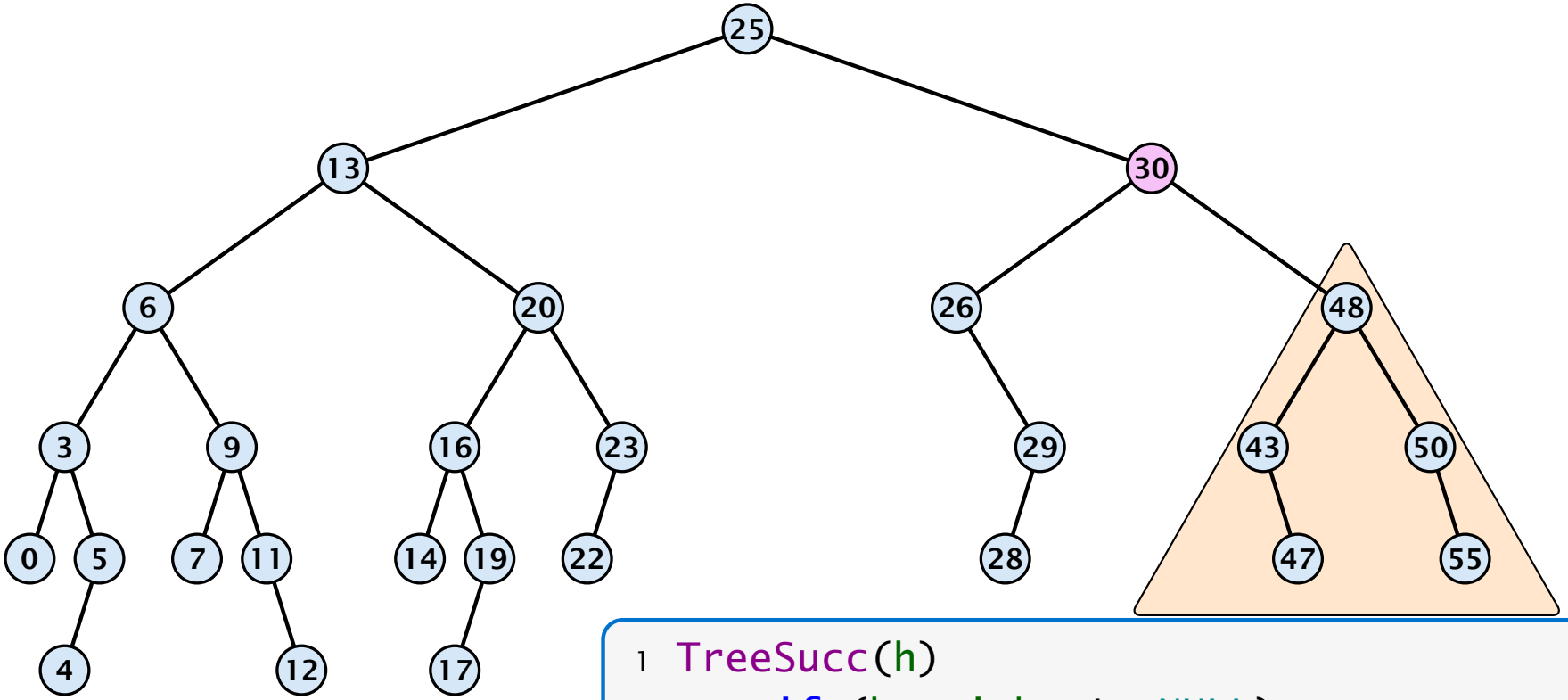

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

# Binäre Suchbäume: Nachfolger



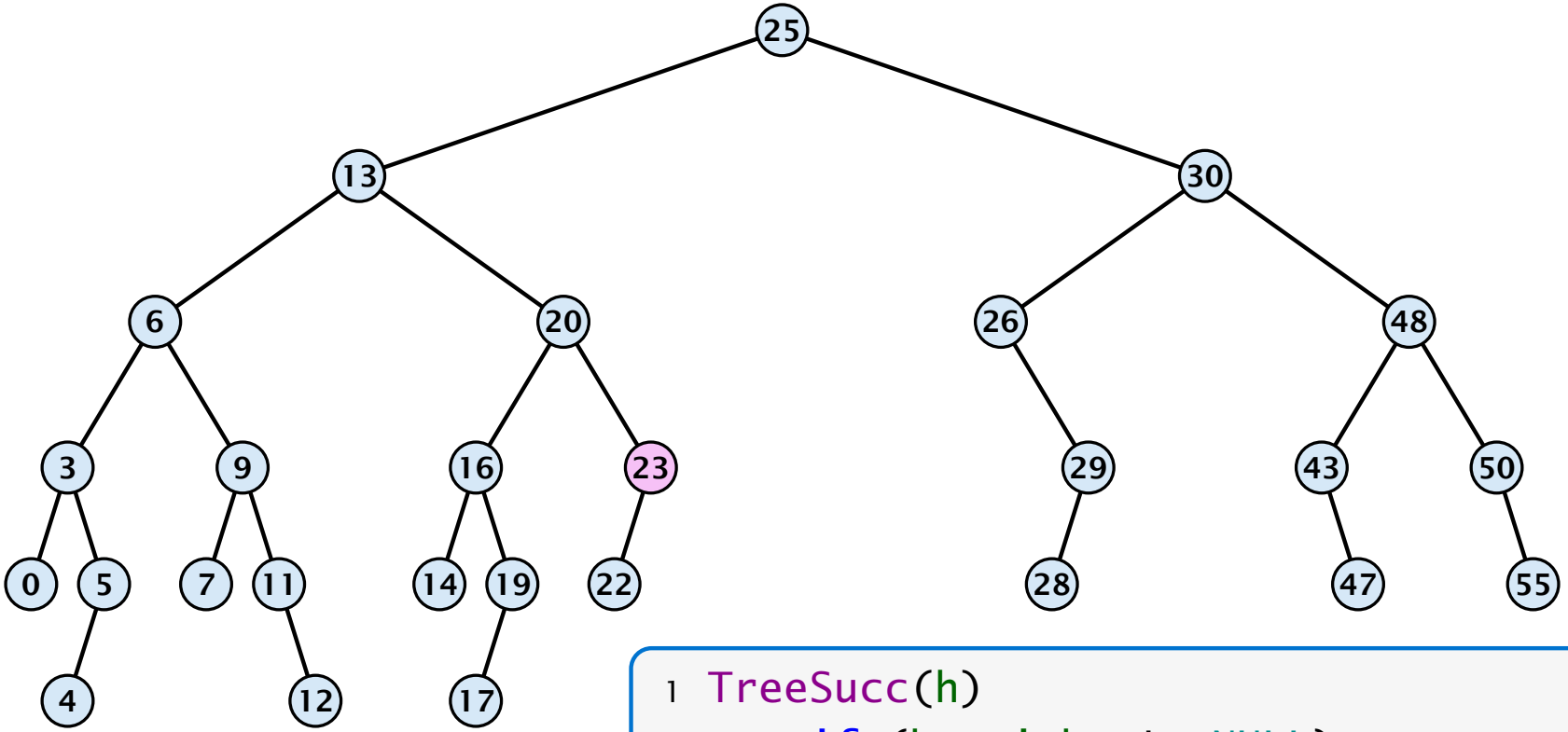

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

# Binäre Suchbäume: Nachfolger



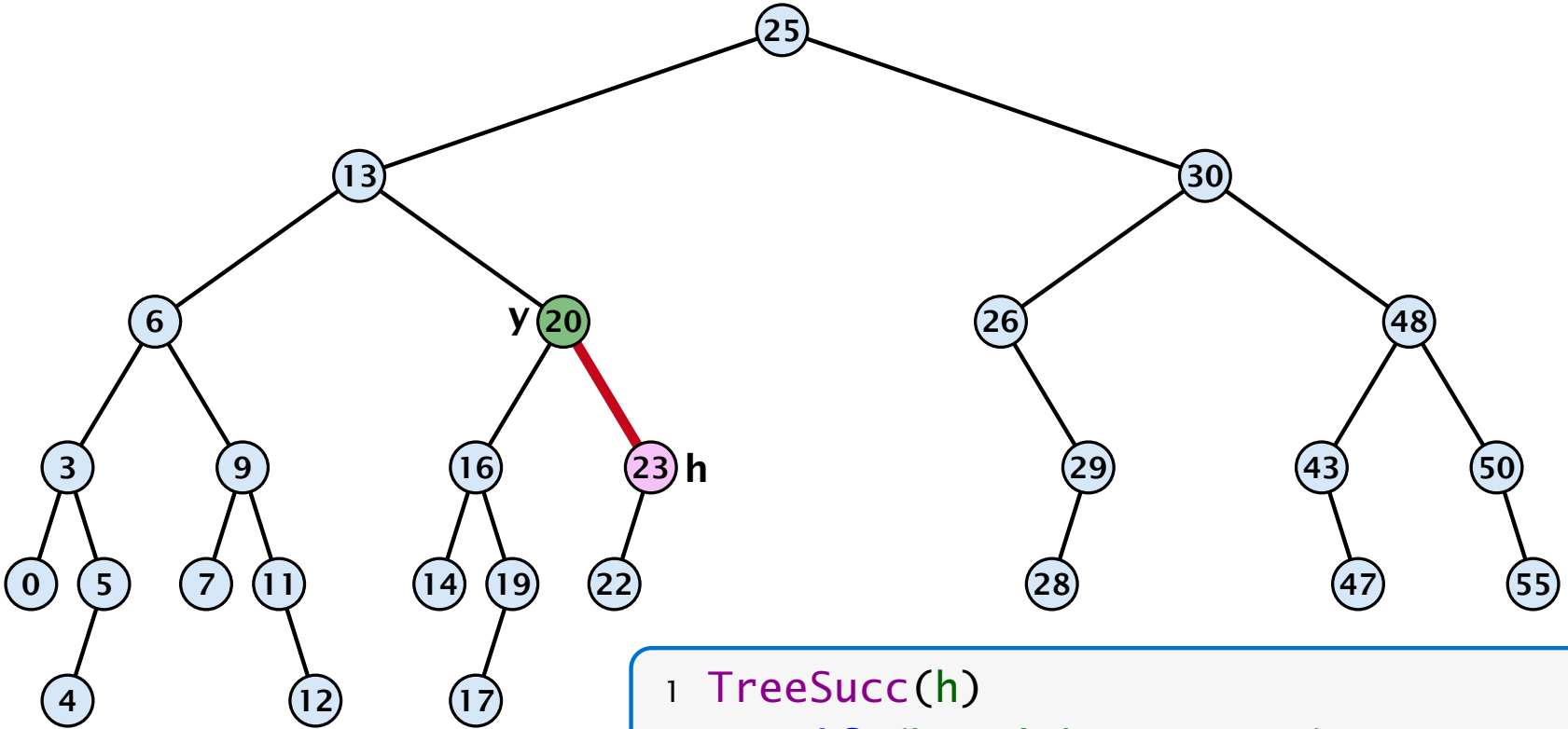

```
1  TreeSucc(h)
2      if (h->right != NULL)
3          return TreeMin(h->right);
4      y = h->parent;
5      while (y != NULL && h == y->right)
6          h = y; y = h->parent;
7      return y;
```

# Binäre Suchbäume: Nachfolger



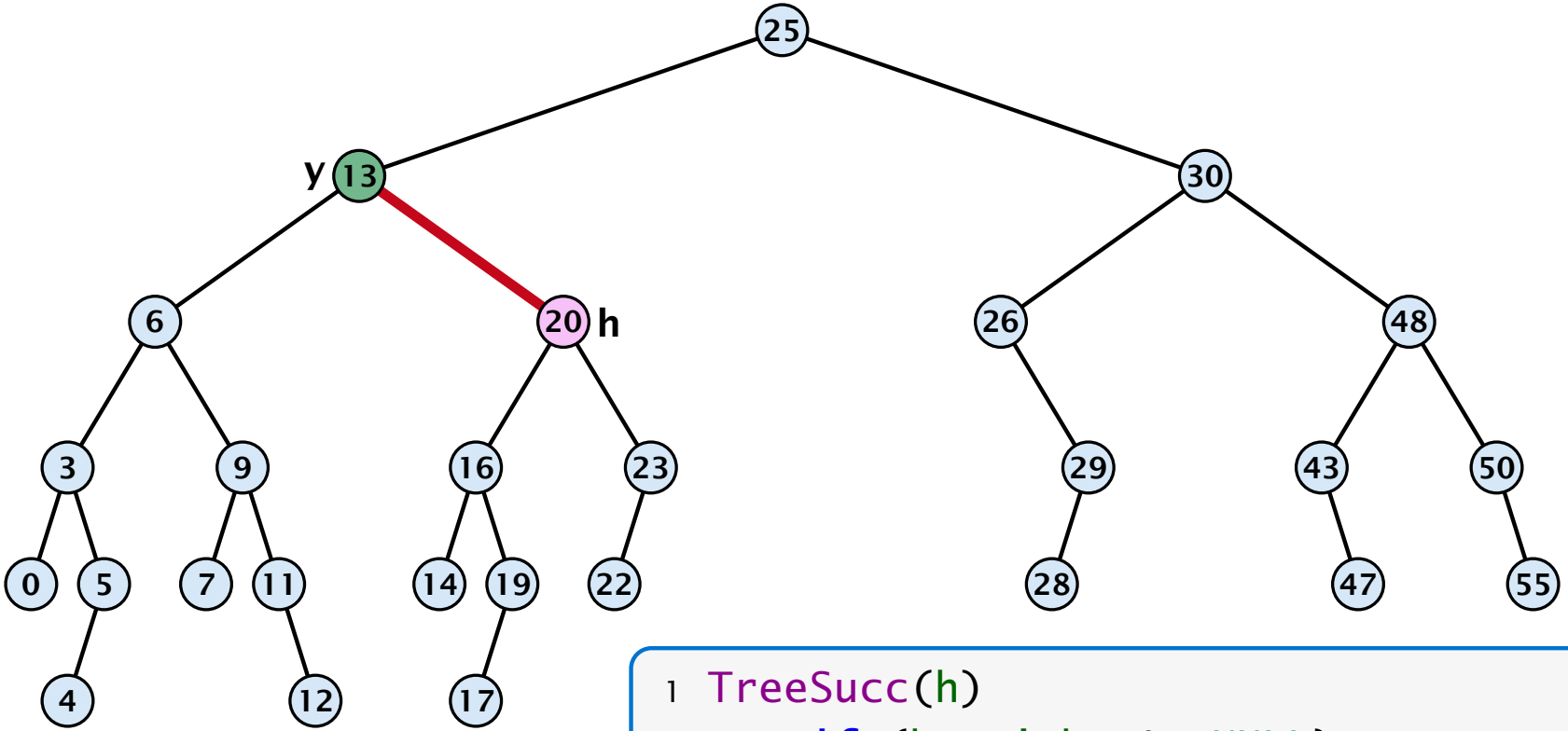

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

# Binäre Suchbäume: Nachfolger



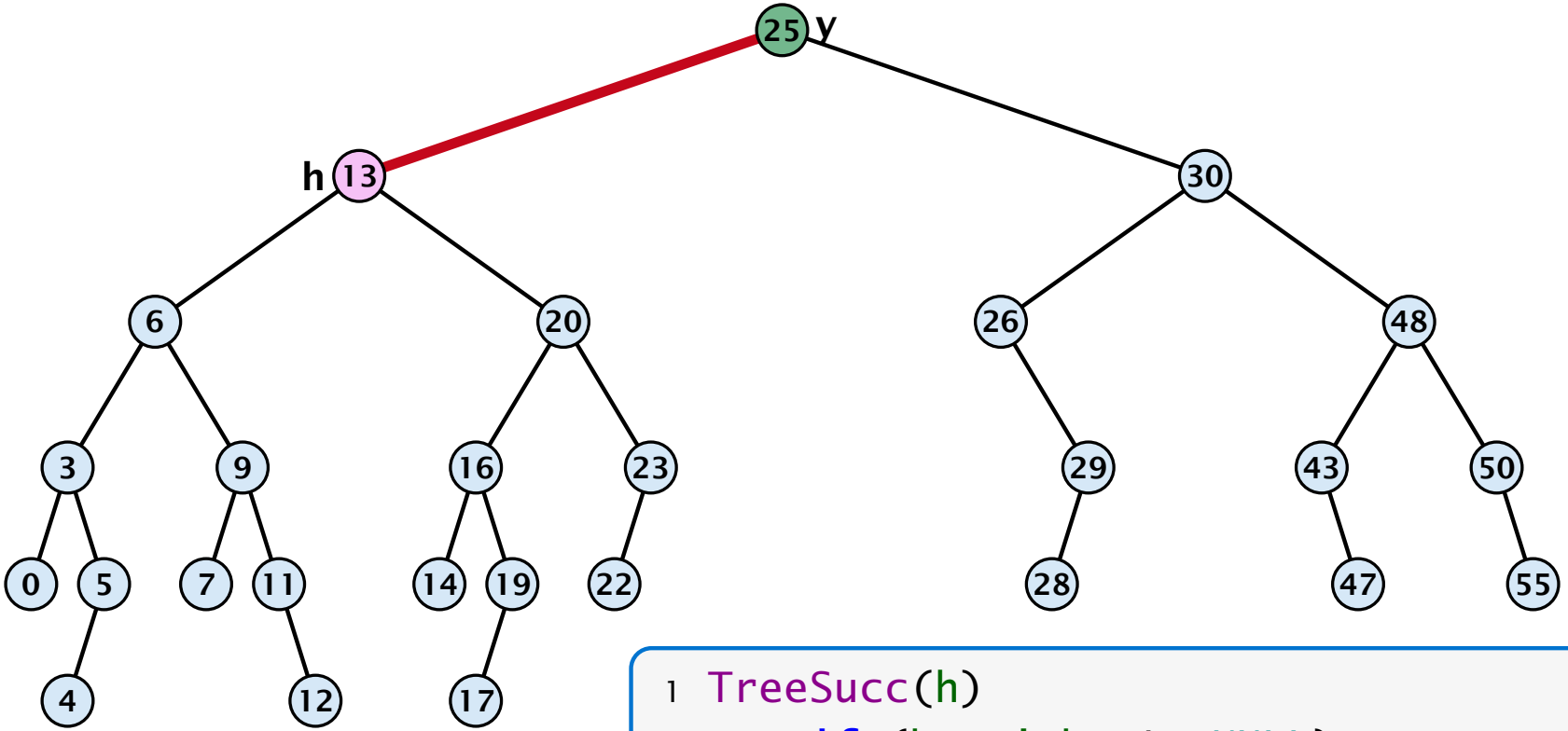

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

# Binäre Suchbäume: Nachfolger



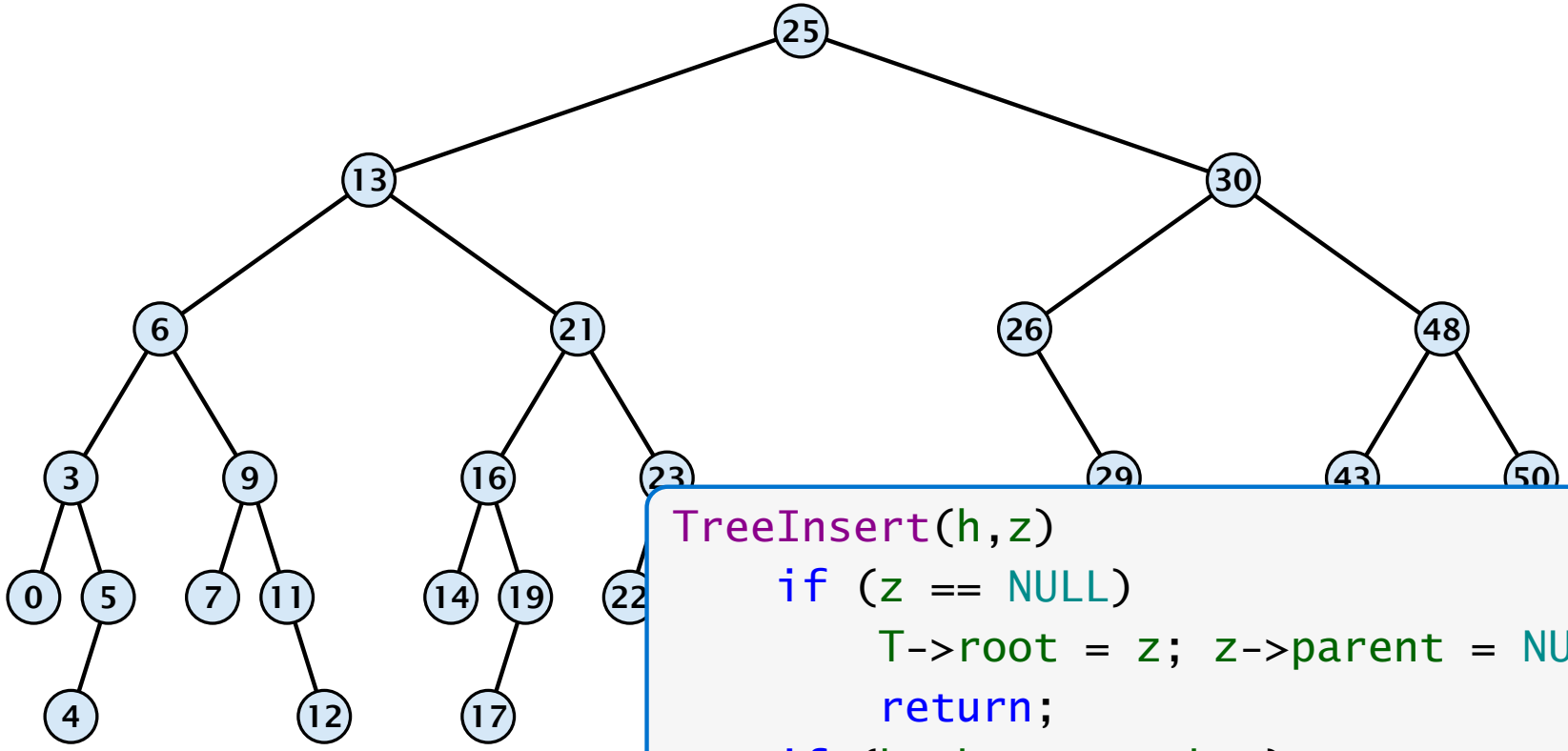

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

# Binäre Suchbäume: Nachfolger




```
1  TreeSucc(h)
2      if (h->right != NULL)
3          return TreeMin(h->right);
4      y = h->parent;
5      while (y != NULL && h == y->right)
6          h = y; y = h->parent;
7      return y;
```

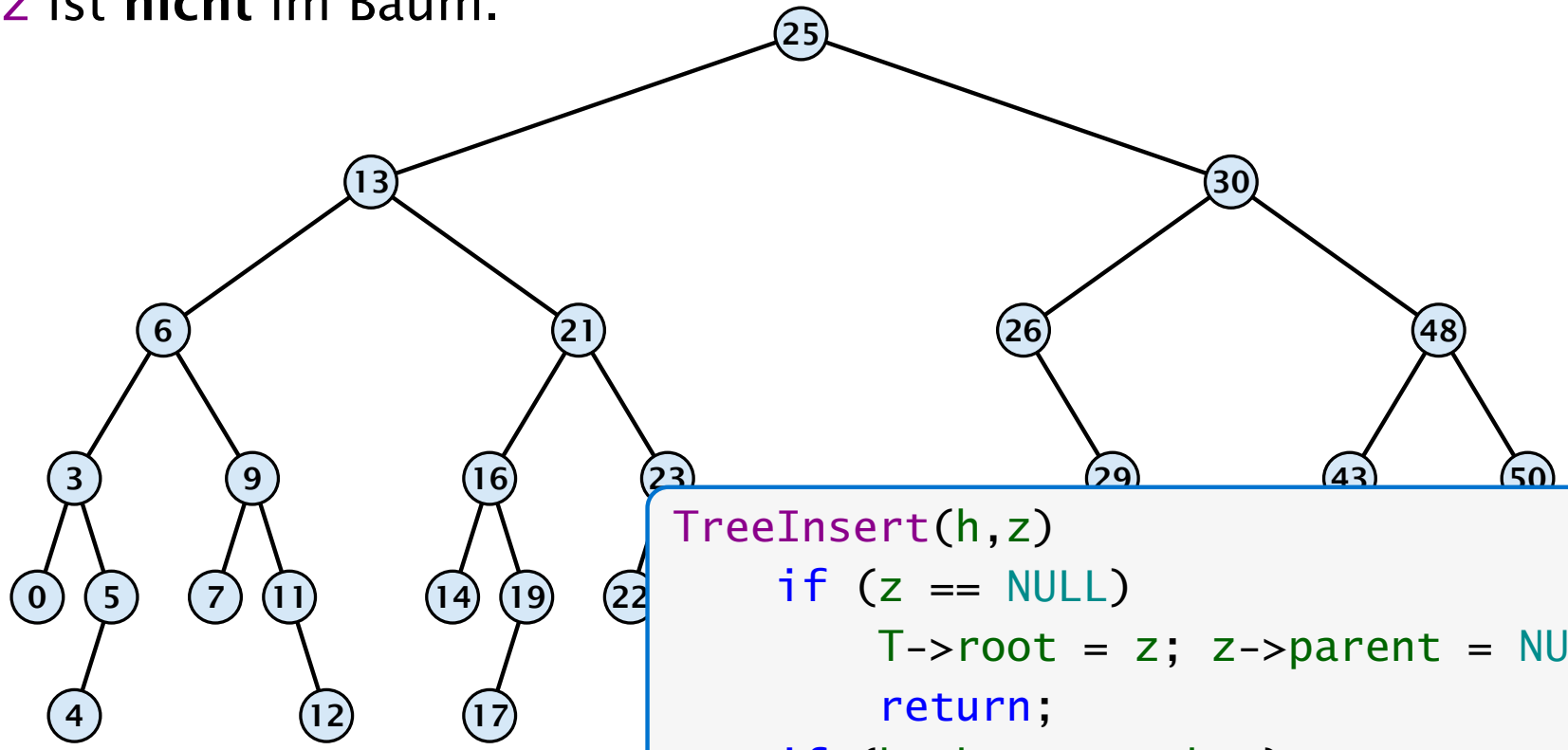
# Binäre Suchbäume: Einfügen




```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

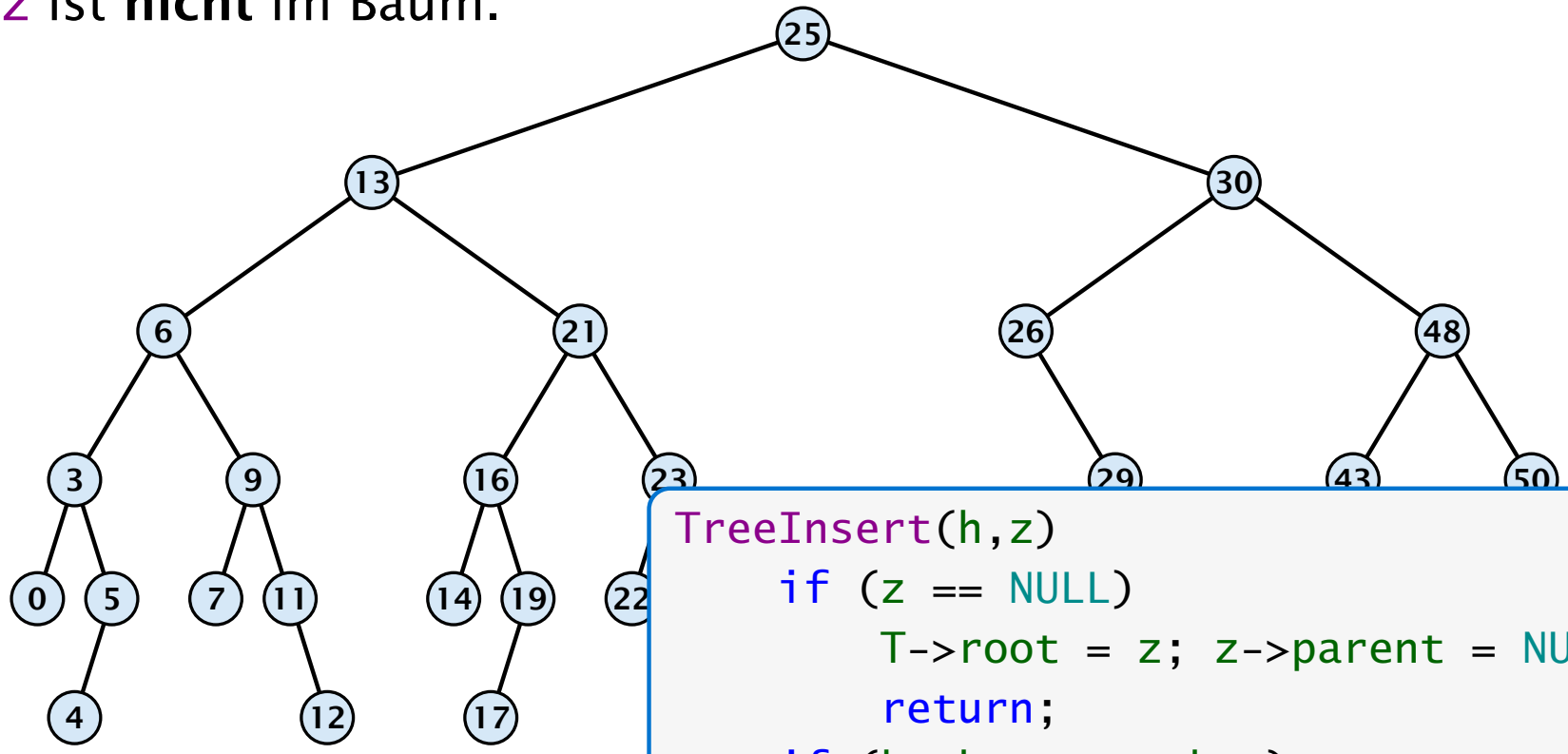
z ist **nicht** im Baum.




```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.



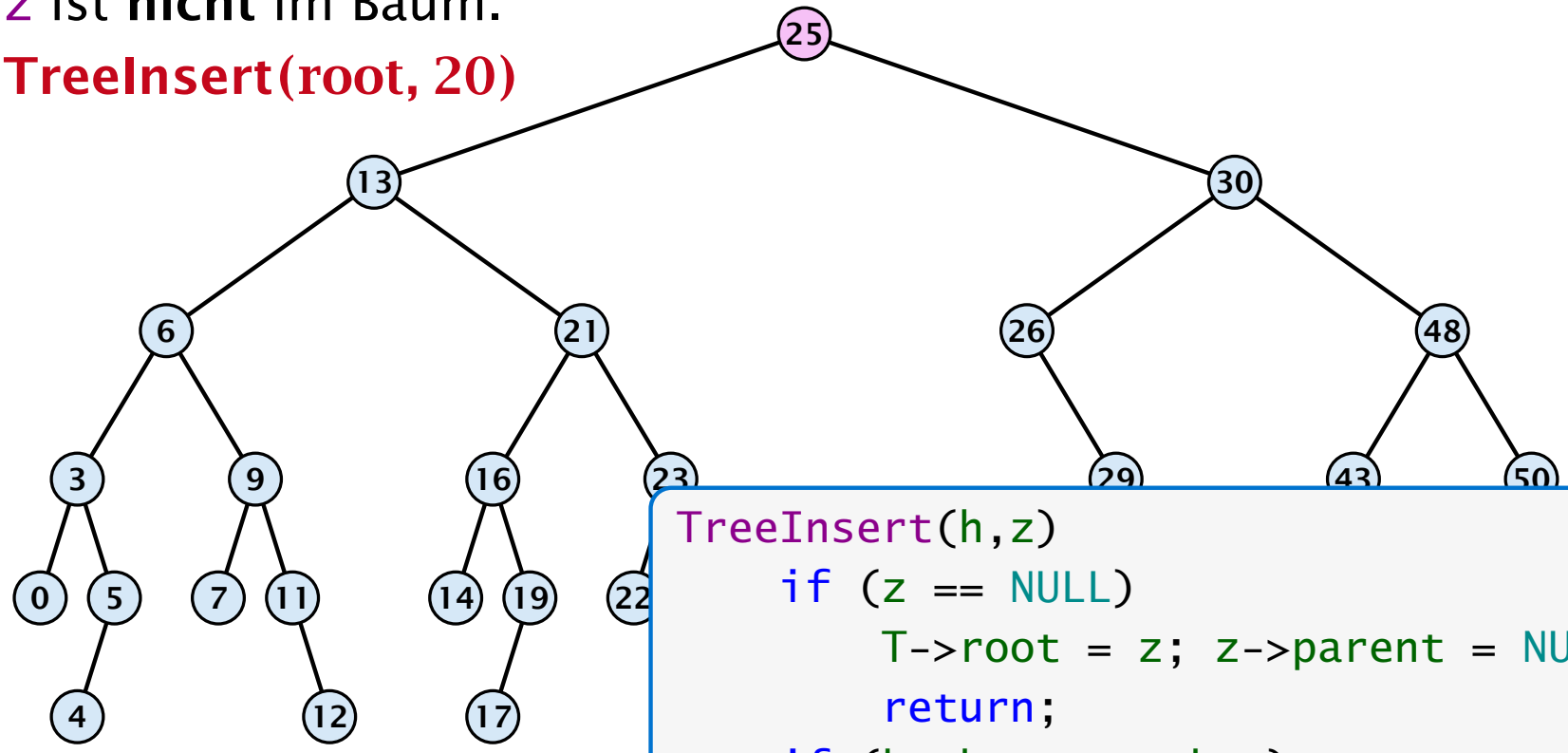

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.

**TreeInsert(root, 20)**



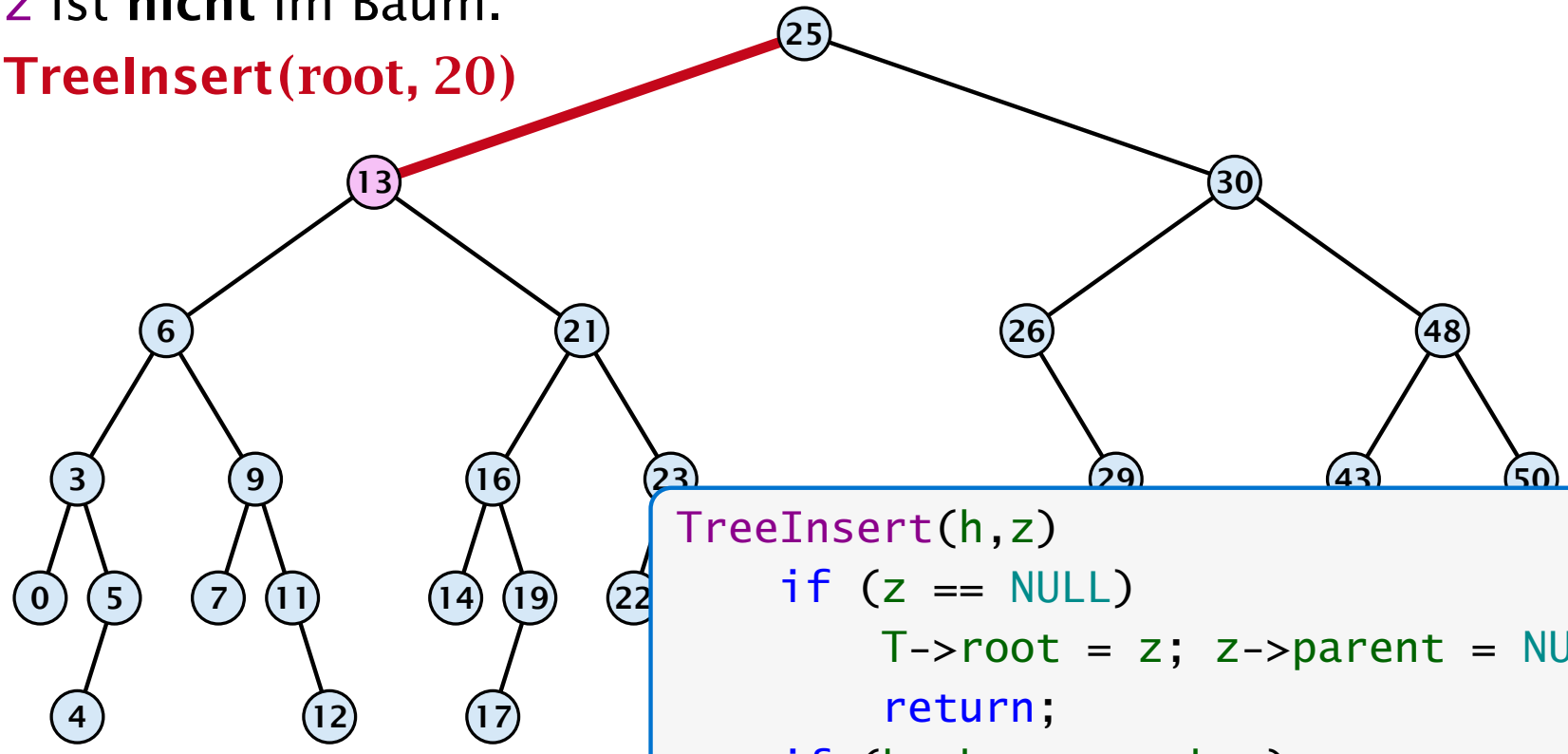

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.

**TreeInsert(root, 20)**



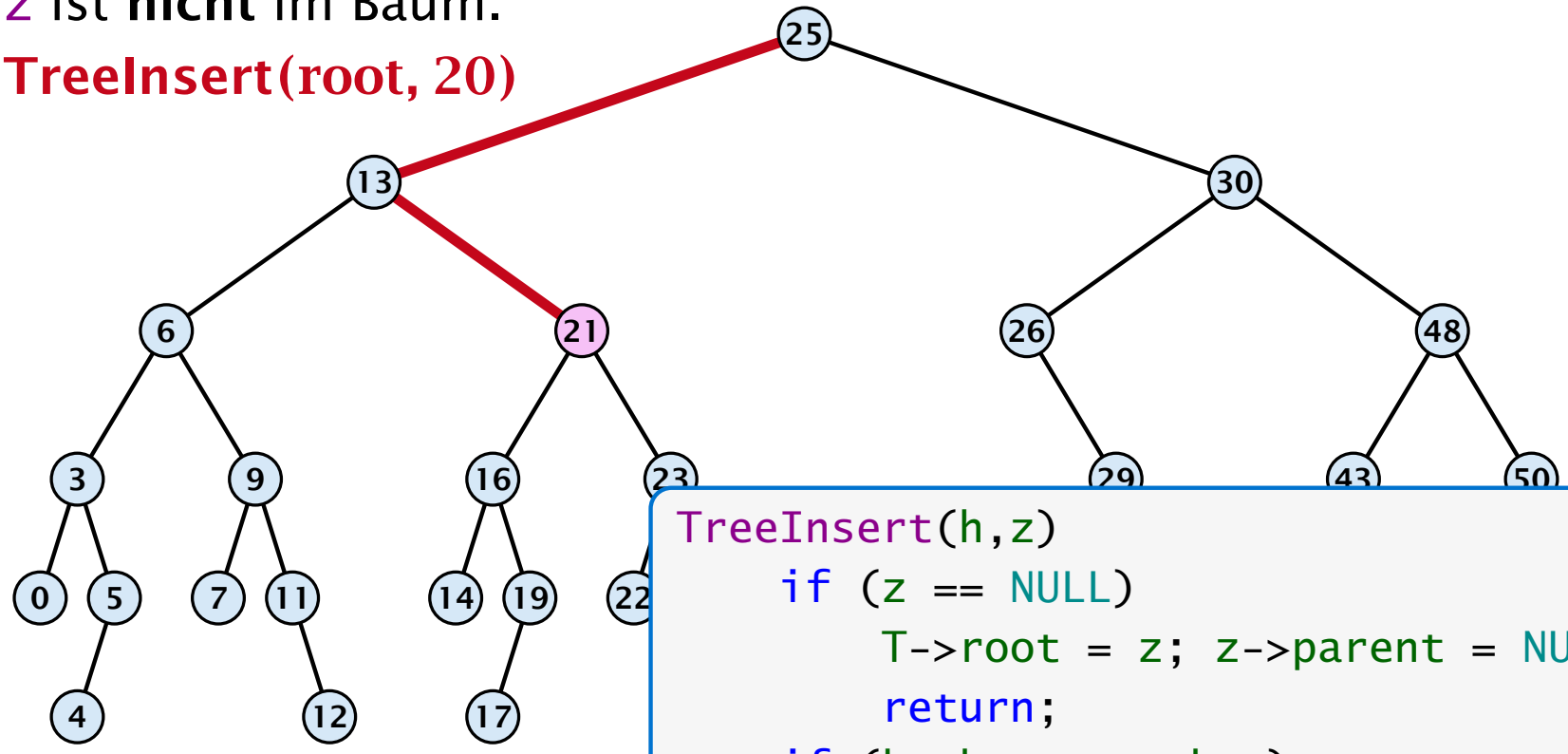

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.

**TreeInsert(root, 20)**



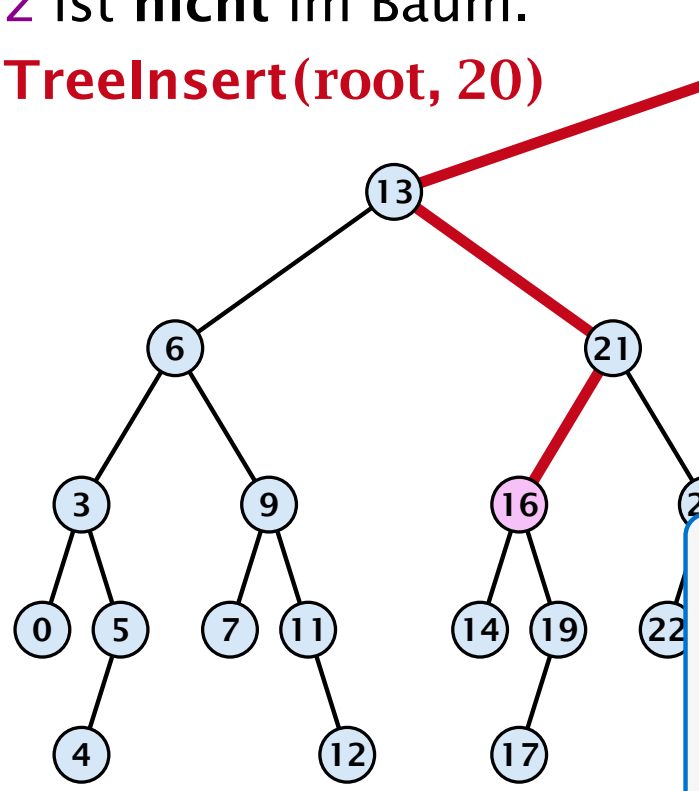

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.

**TreeInsert(root, 20)**



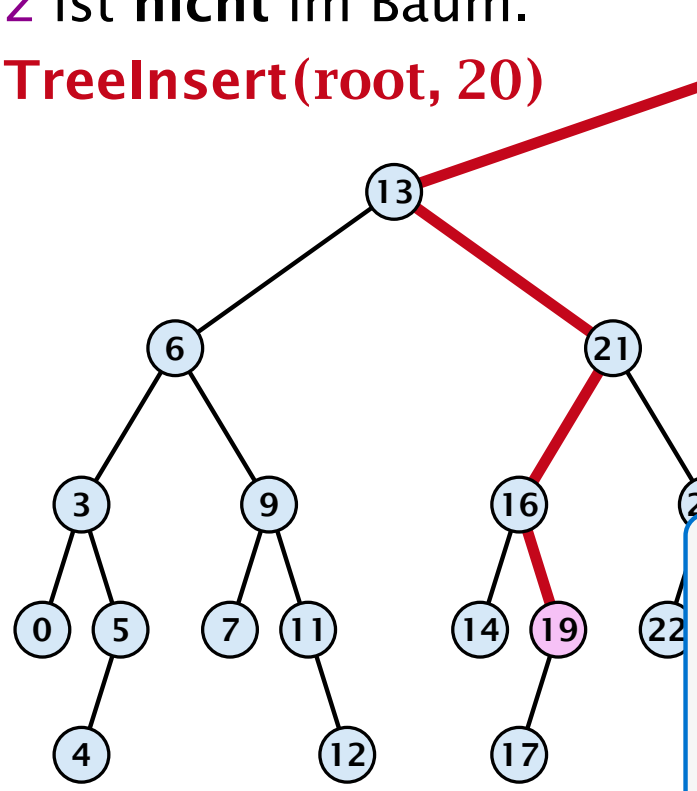

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.

**TreeInsert(root, 20)**



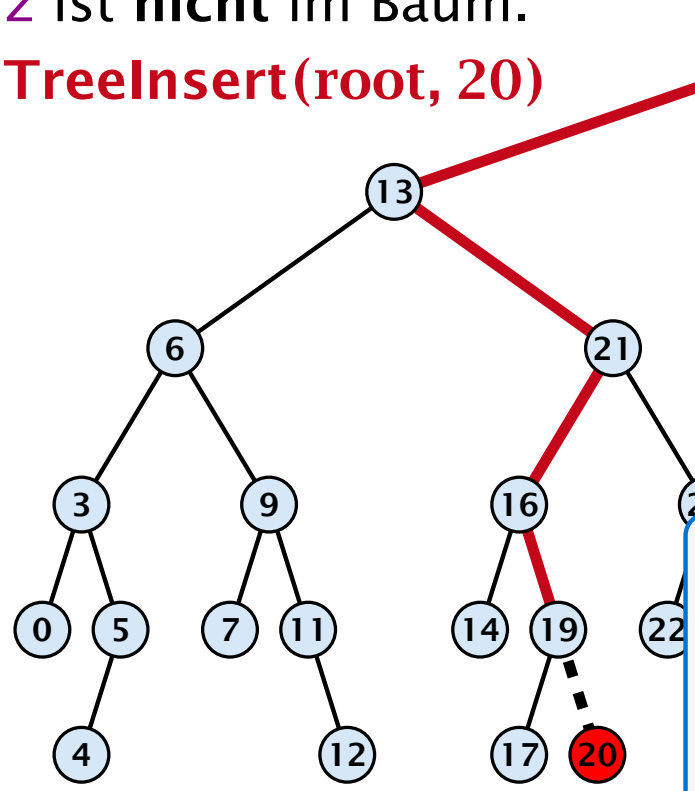

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Einfügen

$z$ ist **nicht** im Baum.

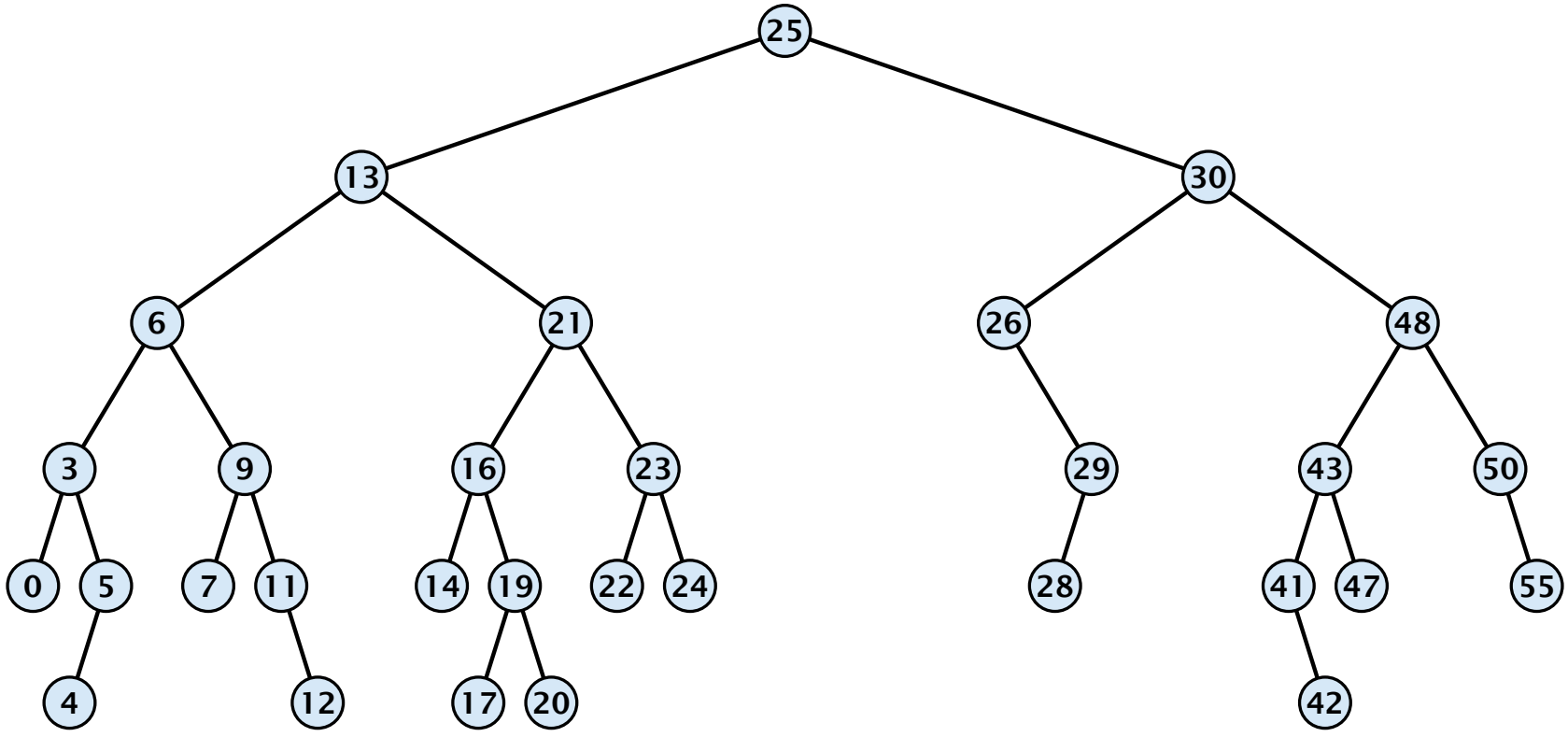
**TreeInsert(root, 20)**



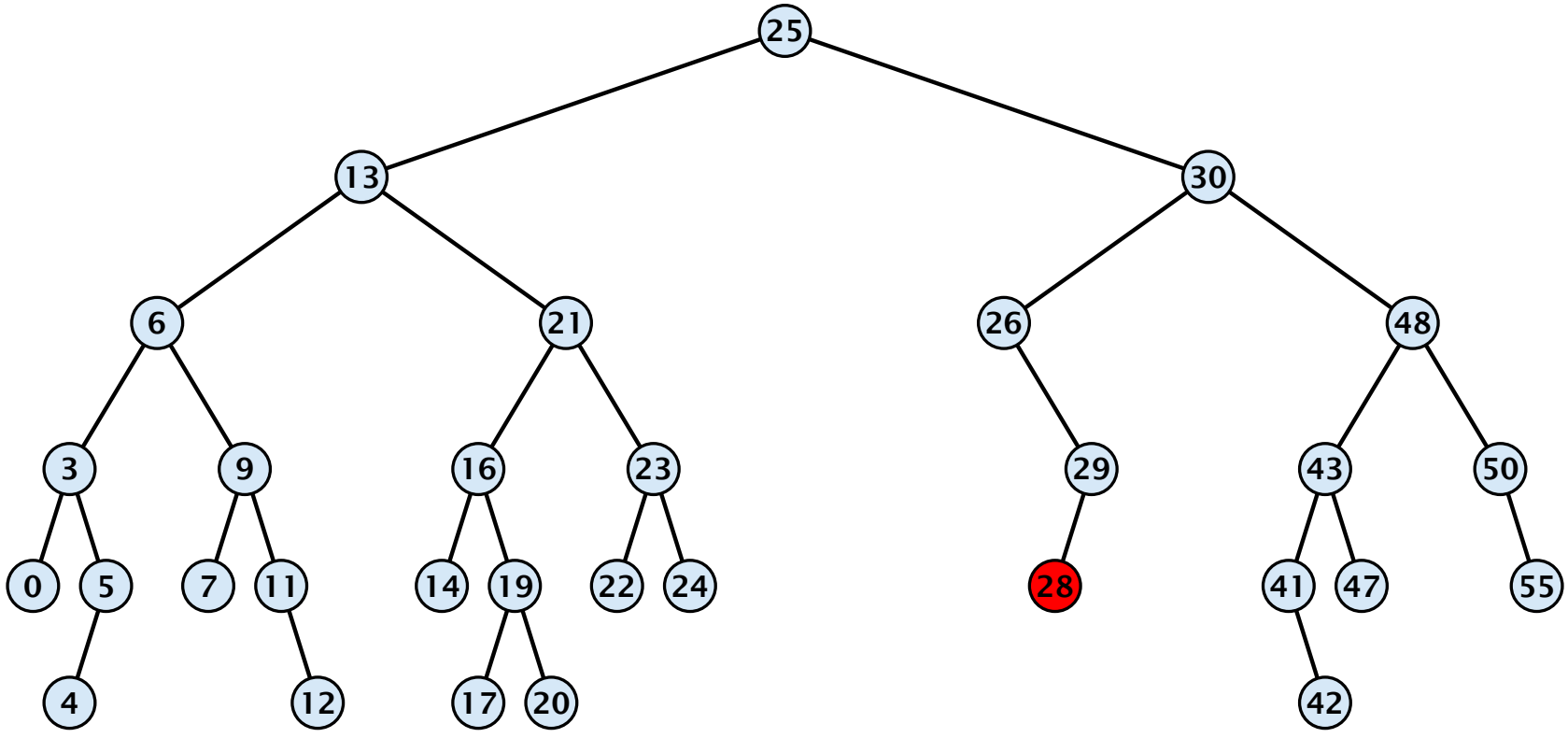

Suche nach $z$. Suche endet an NULL-Zeiger. Hier wird $z$ eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

# Binäre Suchbäume: Löschen



# Binäre Suchbäume: Löschen

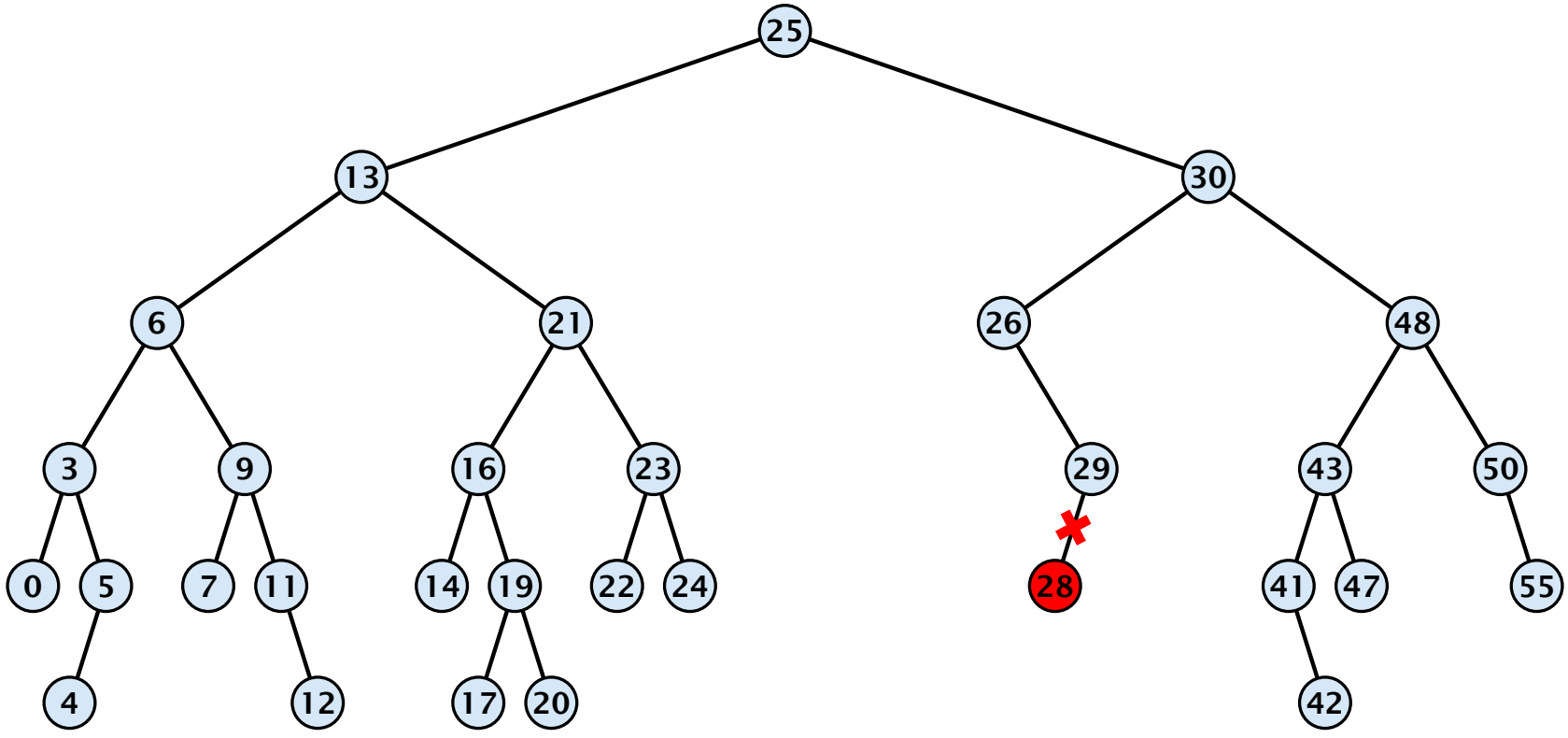



## Fall 1:

Element hat keine Kinder

- Der Kindzeiger am Elternknoten wird auf NULL gesetzt.

# Binäre Suchbäume: Löschen

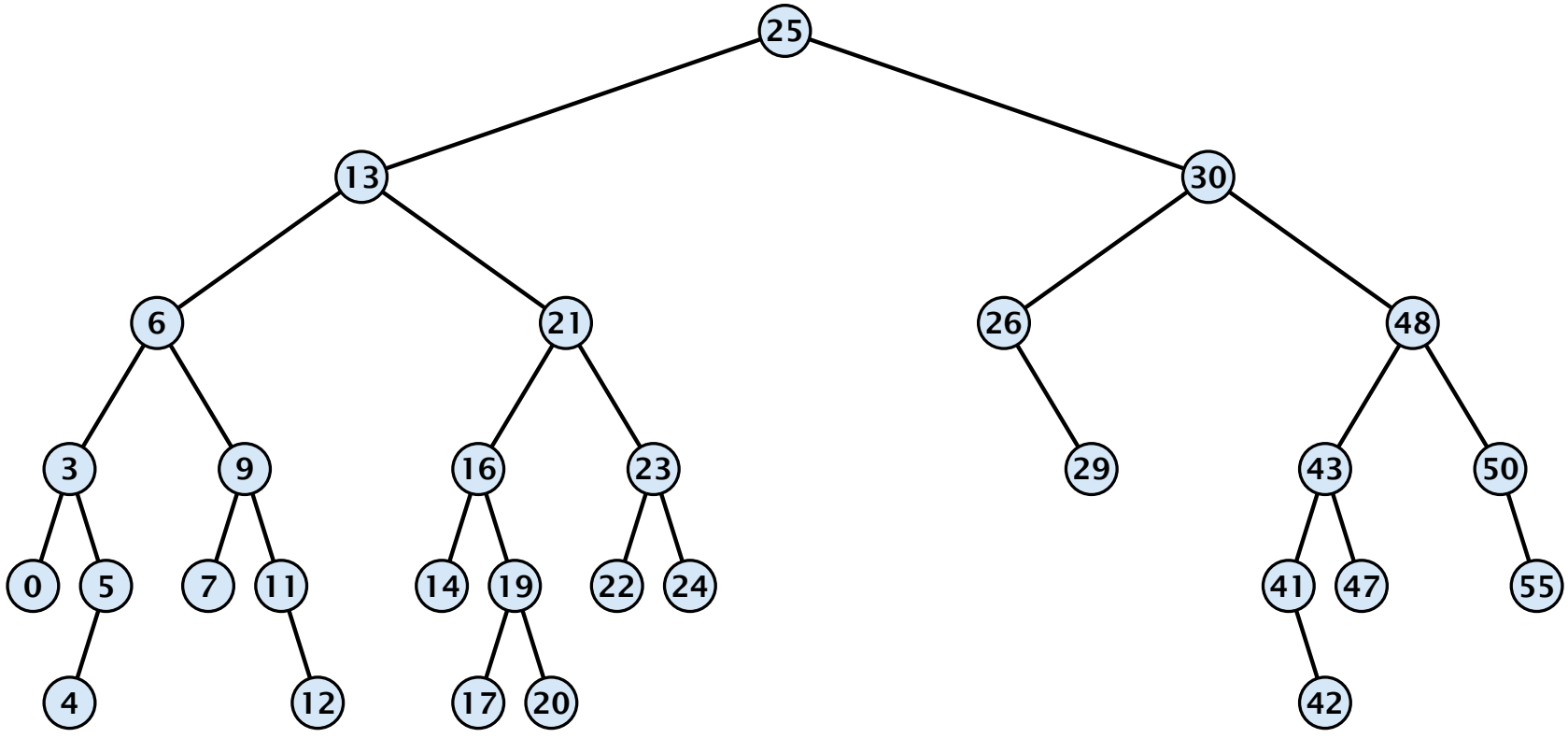



## Fall 1:

Element hat keine Kinder

- Der Kindzeiger am Elternknoten wird auf NULL gesetzt.

# Binäre Suchbäume: Löschen

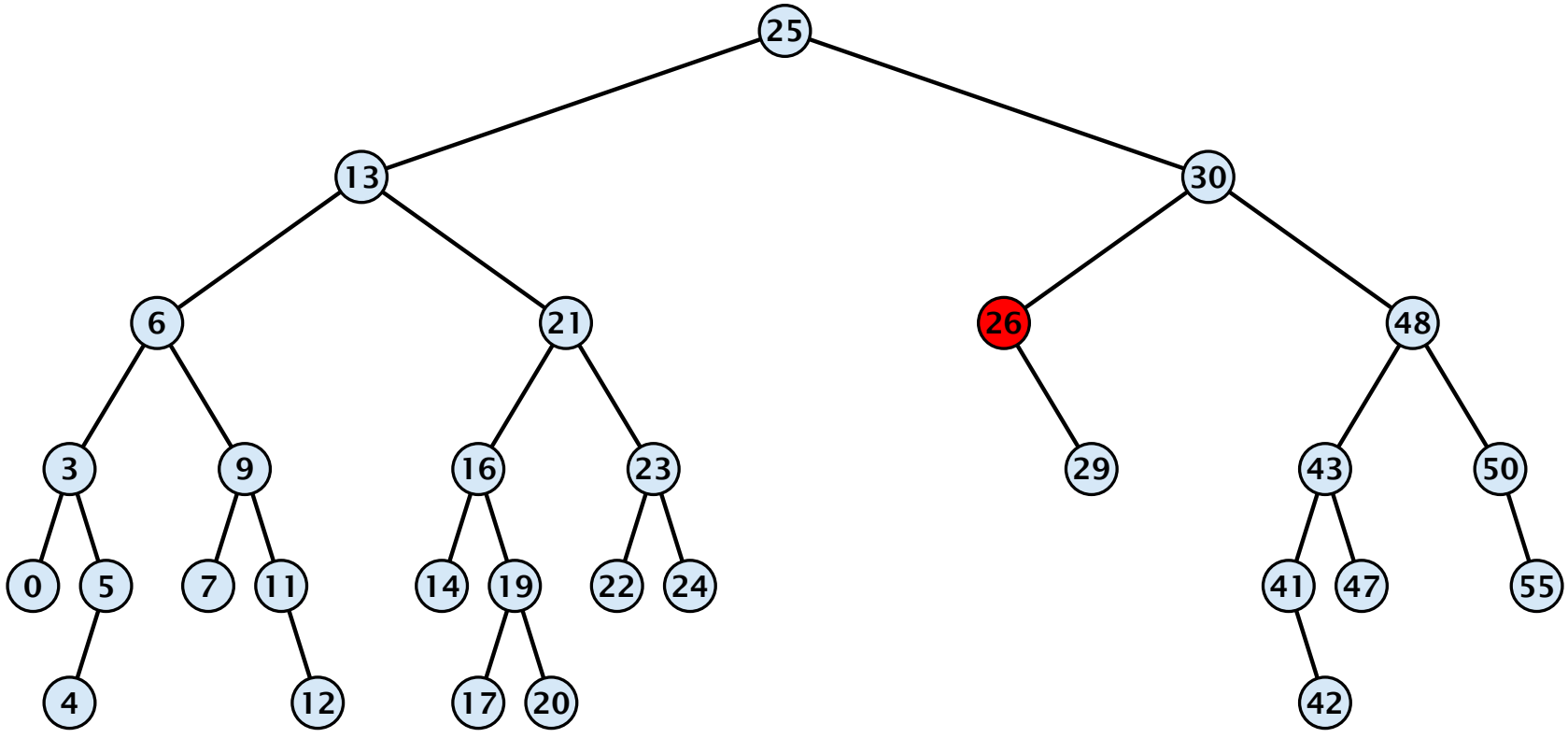



## Fall 1:

Element hat keine Kinder

- Der Kindzeiger am Elternknoten wird auf NULL gesetzt.

# Binäre Suchbäume: Löschen

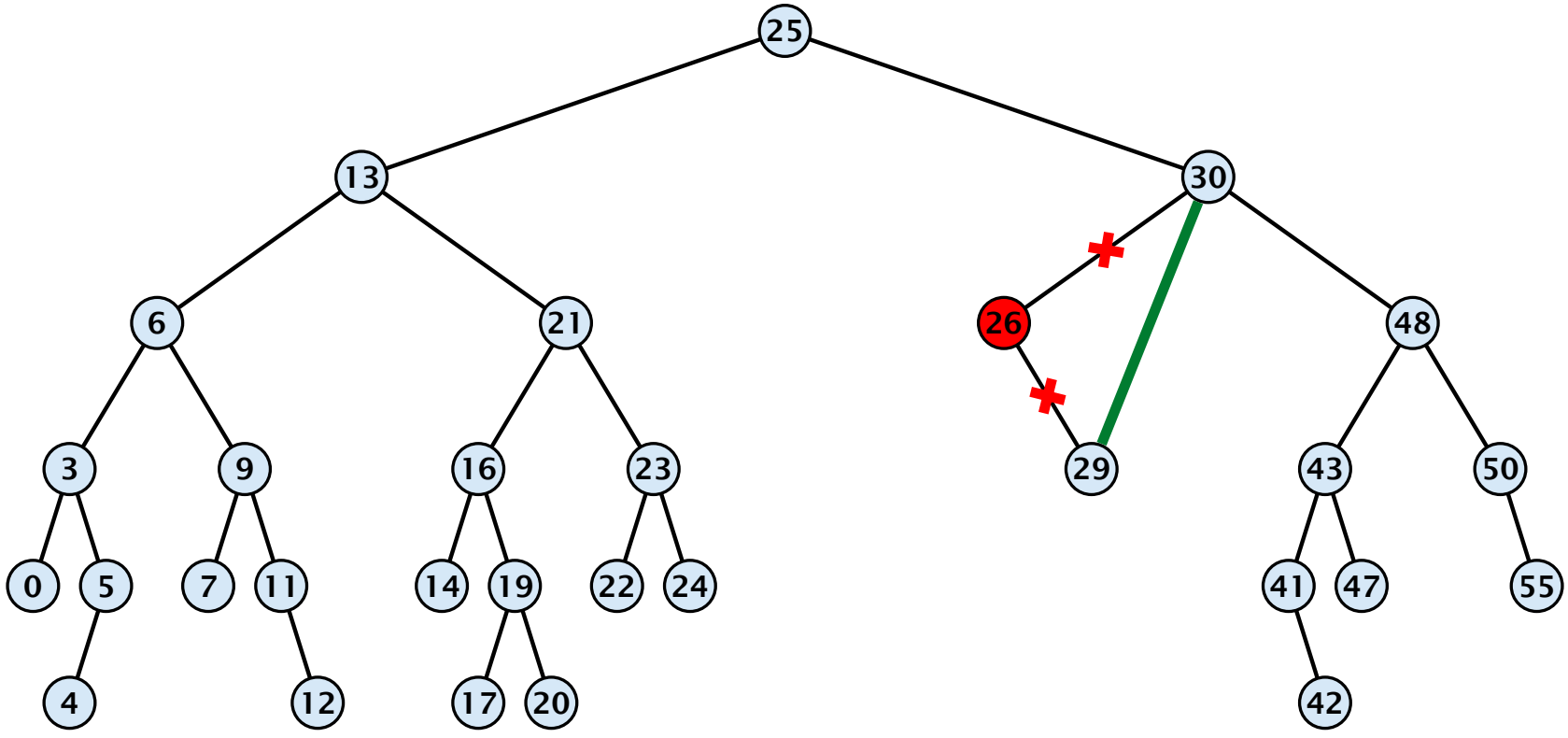



## Fall 2:

Element hat genau ein Kind

- Überbrücke Element indem man Elternknoten mit Kind verbindet.

# Binäre Suchbäume: Löschen

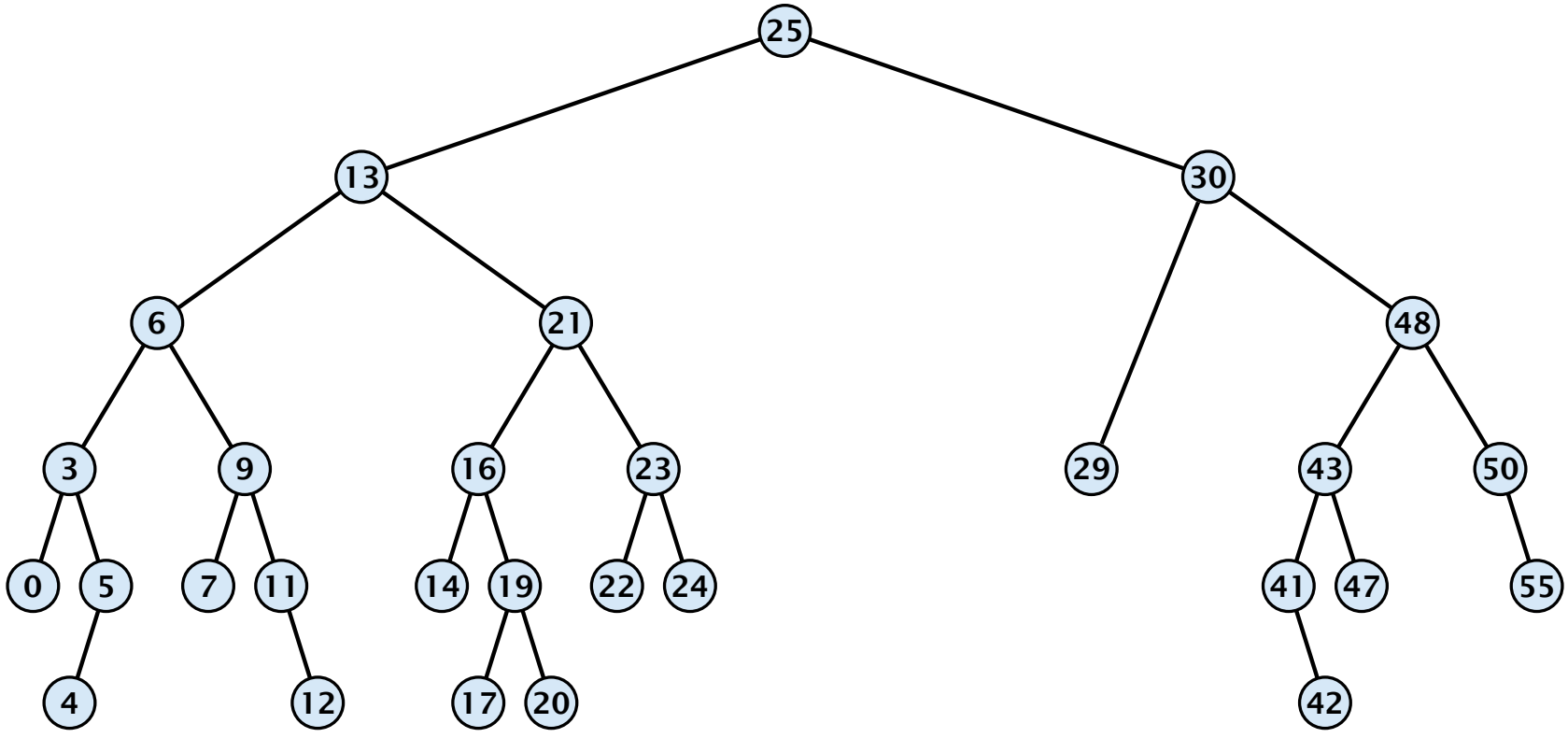



## Fall 2:

Element hat genau ein Kind

- Überbrücke Element indem man Elternknoten mit Kind verbindet.

# Binäre Suchbäume: Löschen

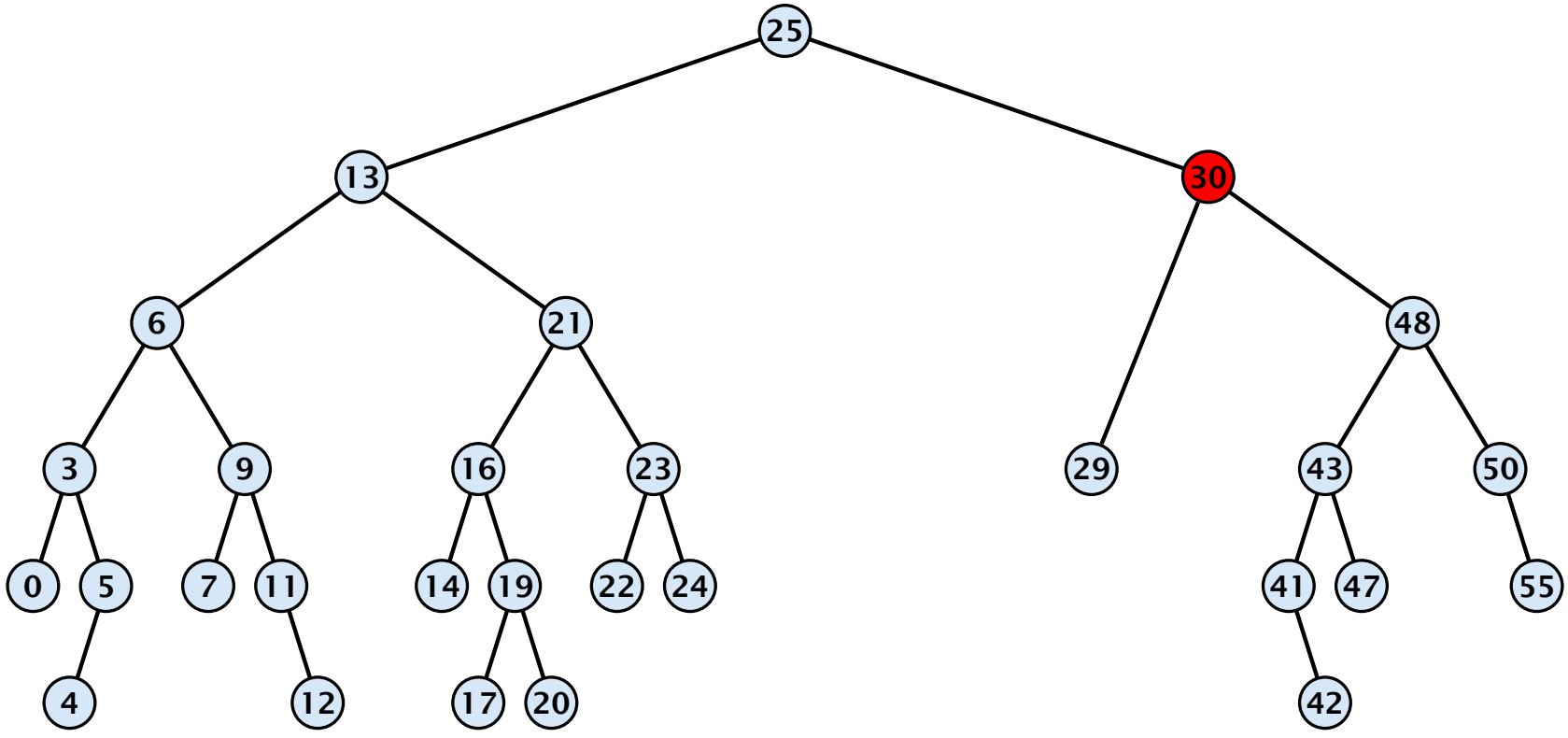



## Fall 2:

Element hat genau ein Kind

- Überbrücke Element indem man Elternknoten mit Kind verbindet.

# Binäre Suchbäume: Löschen

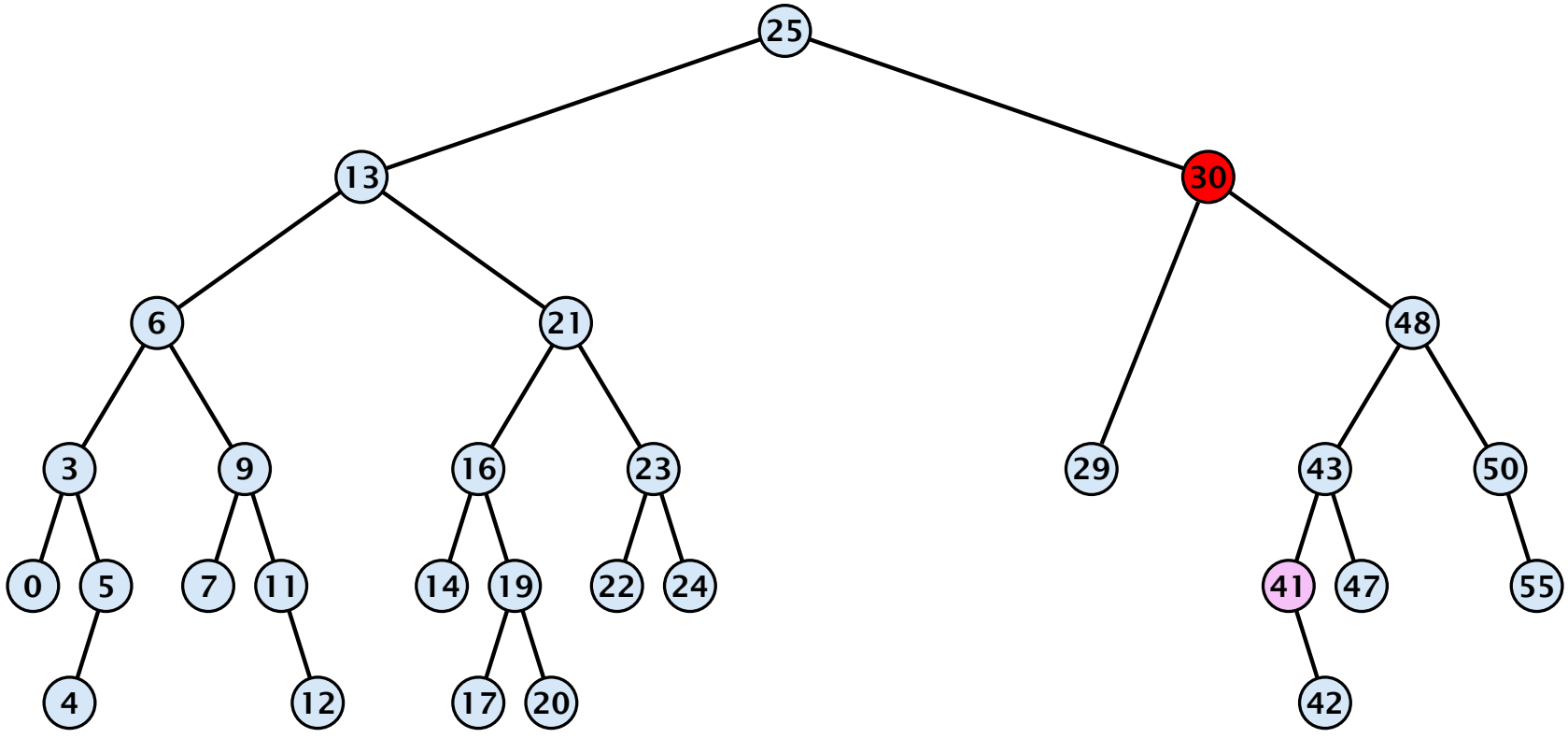



## Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

# Binäre Suchbäume: Löschen

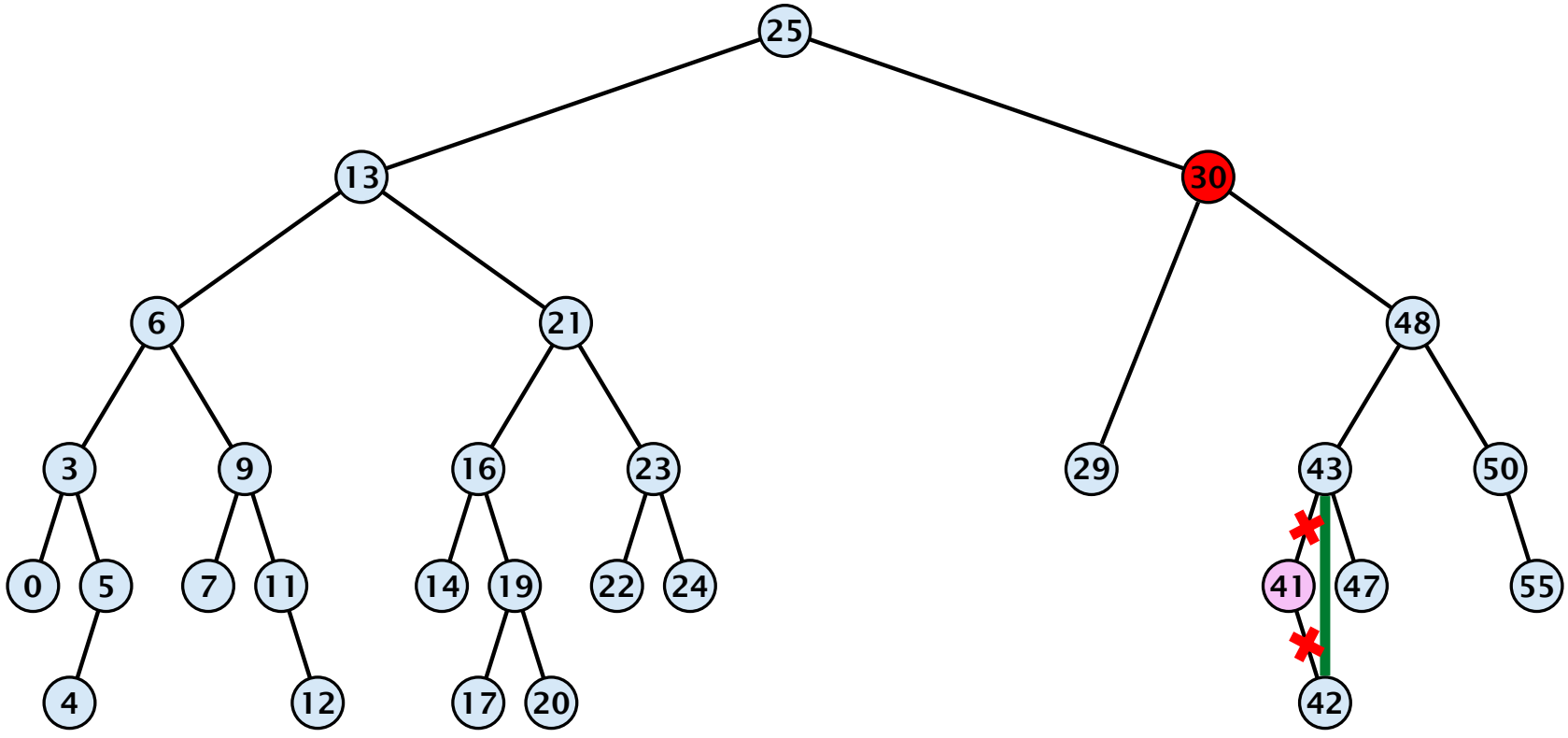



## Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

# Binäre Suchbäume: Löschen

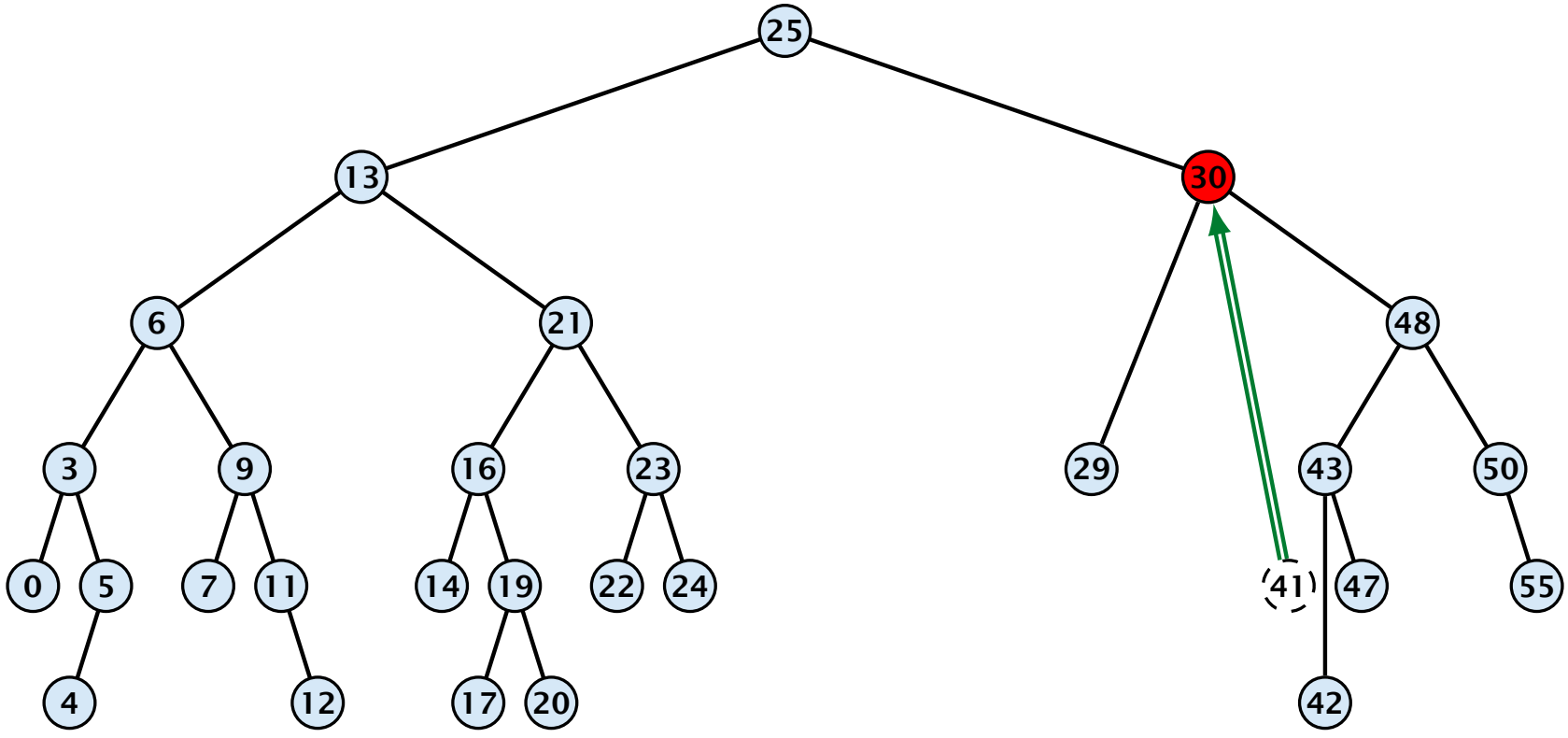



## Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

# Binäre Suchbäume: Löschen

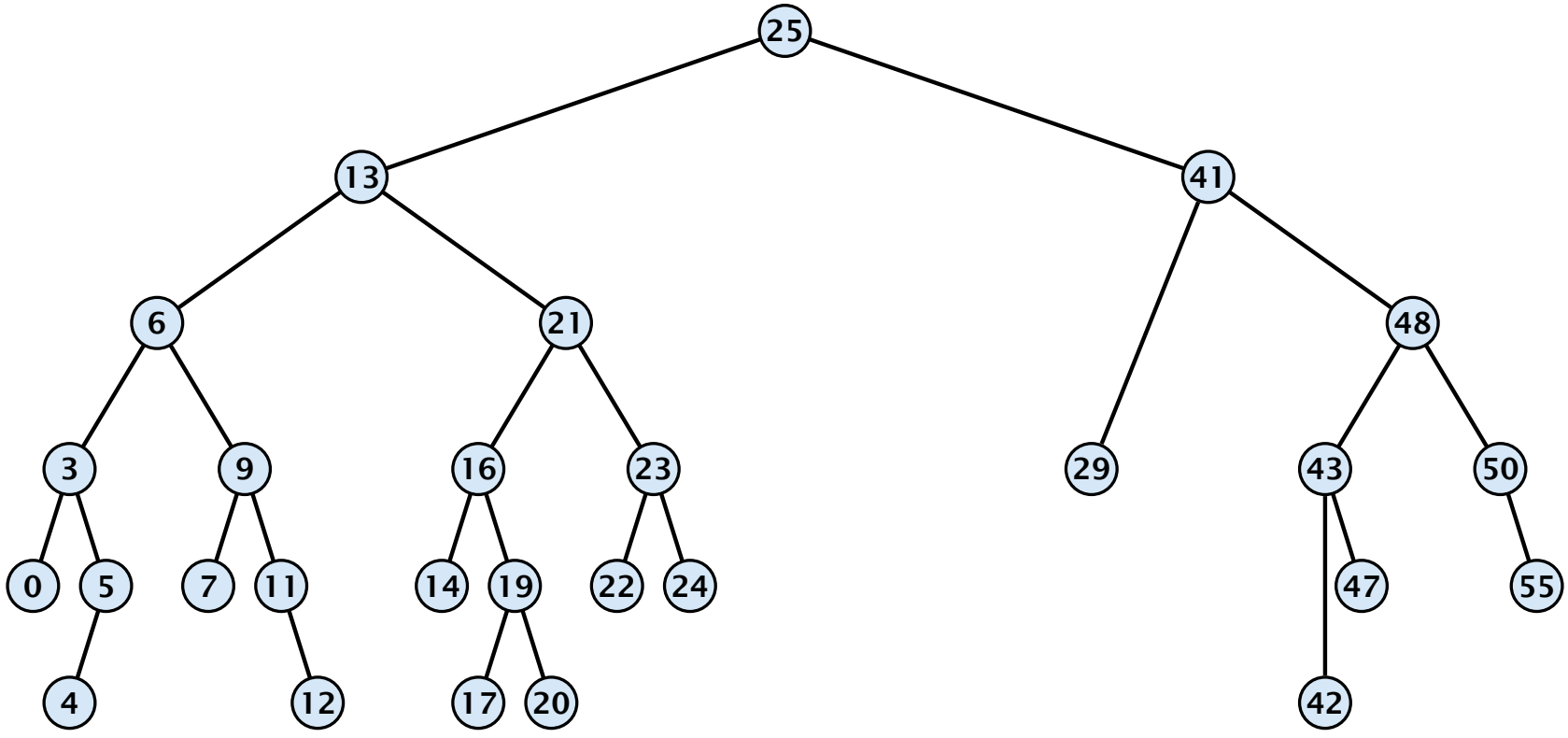



## Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

# Binäre Suchbäume: Löschen

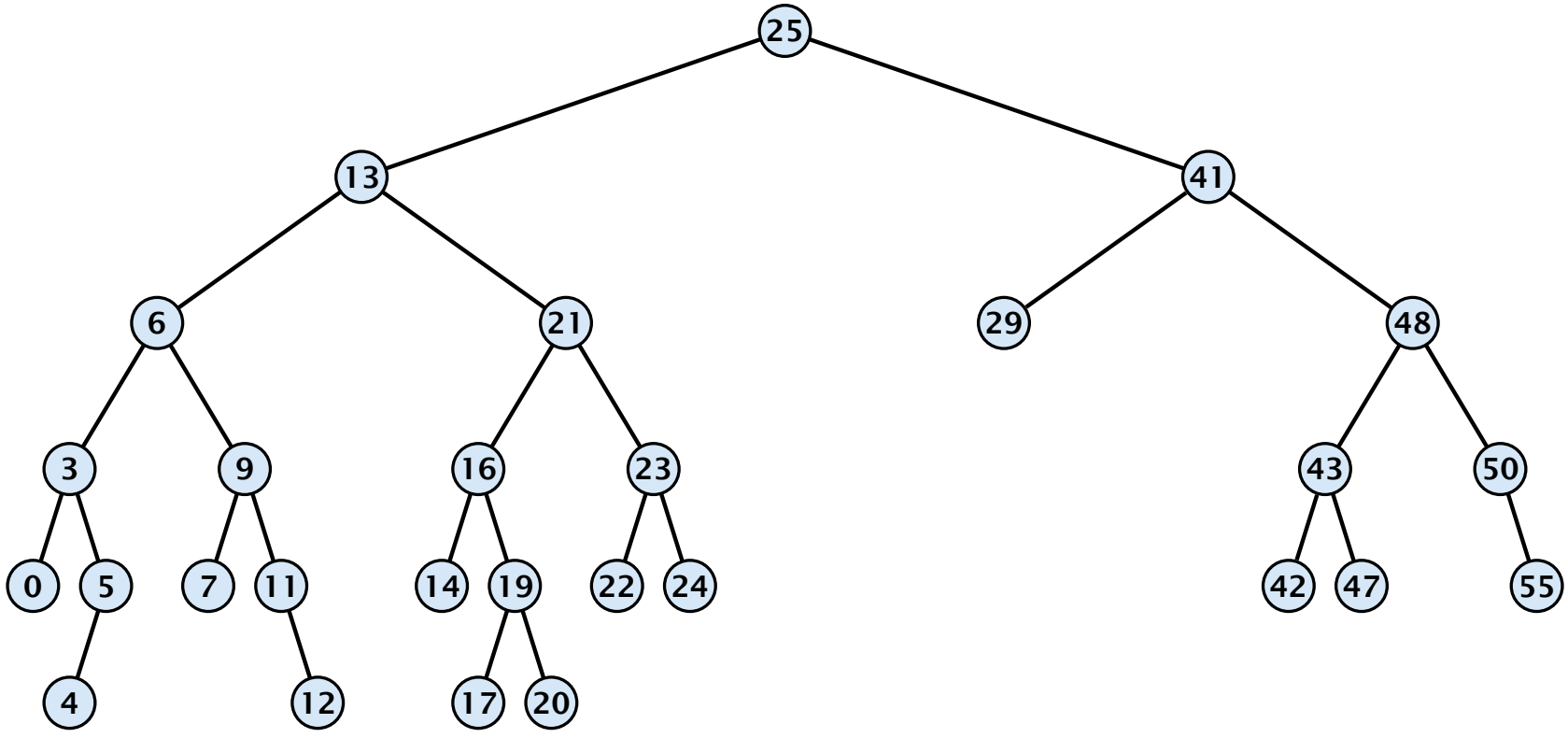



## Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

# Binäre Suchbäume: Löschen




## Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

# Binäre Suchbäume: Löschen

```
1  TreeDelete(z)
2      if (z->left == NULL || z->right == NULL)
3          y = z;
4      else y = TreeSucc(z);
5      if (y->left != NULL)
6          x = y->left;
7      else x = y->right;
8      if (x != NULL)
9          x->parent = y->parent;
10     if (y->parent == NULL)
11         T->root = x;
12     else
13         if (y->parent->left == y)
14             y->parent->left  = x;
15         else
16             y->parent->right = x;
17     if (y != z) replace z with y
```

# Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei $h$ die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

**Balancierte Suchbäume**
Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

# Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei $h$ die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

**Balancierte Suchbäume**

Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

# Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei $h$ die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

**Balancierte Suchbäume**

Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

# Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei $h$ die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

**Balancierte Suchbäume**
Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

# Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei $h$ die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

**Balancierte Suchbäume**
Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

# 7.5 AVL-Bäume

**Definition 4**

AVL-Bäume sind binäre Suchbäume, die die folgende Balancierungsbedingung erfüllen: Für jeden Knoten $v$

$$|\text{height}(\text{left sub-tree}(v)) - \text{height}(\text{right sub-tree}(v))| \leq 1 \ .$$

**Lemma 5**

*Ein AVL-Baum der Höhe $h$ enthält mindestens $F_{h+2} - 1$ und höchstens $2^h - 1$ interne Knoten, wobei $F_n$ die $n$-te Fibonaccizahl ist ($F_0 = 0$, $F_1 = 1$), und $h$ die Höhe des Baumes bezeichnet.*

In einem AVL-Baum werden Schlüssel nur an internen Knoten gespeichert. Die Blattknoten sind sogenannte dummy-leafs, die einfach ein nicht vorhandenes Kind symbolisieren.

Zusätzlich hat jeder interne Knoten genau zwei Kinder.

# 7.5 AVL-Bäume

### Definition 4

AVL-Bäume sind binäre Suchbäume, die die folgende Balancierungsbedingung erfüllen: Für jeden Knoten $v$

$$|\text{height}(\text{left sub-tree}(v)) - \text{height}(\text{right sub-tree}(v))| \leq 1 \ .$$

### Lemma 5

*Ein AVL-Baum der Höhe $h$ enthält mindestens $F_{h+2} - 1$ und höchstens $2^h - 1$ interne Knoten, wobei $F_n$ die $n$-te Fibonaccizahl ist ($F_0 = 0$, $F_1 = 1$), und $h$ die Höhe des Baumes bezeichnet.*

In einem AVL-Baum werden Schlüssel nur an internen Knoten gespeichert. Die Blattknoten sind sogenannte dummy-leafs, die einfach ein nicht vorhandenes Kind symbolisieren.

Zusätzlich hat jeder interne Knoten genau zwei Kinder.

# AVL-Bäume

**Beweis.**

Die obere Schranke folgt, da ein Binärbaum der Höhe $h$ nur

$$\sum_{j=0}^{h-1} 2^j = 2^h - 1$$

interne Knoten enthalten kann.

# AVL trees

## Beweis (cont.)

**Induktionsanfang:**

1. ein AVL-Baum der Höhe $h = 1$ enthält mindestens einen internen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. ein AVL-Baum der Höhe $h = 2$ enthält mindestens zwei interne Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

# AVL trees

## Beweis (cont.)

**Induktionsanfang:**

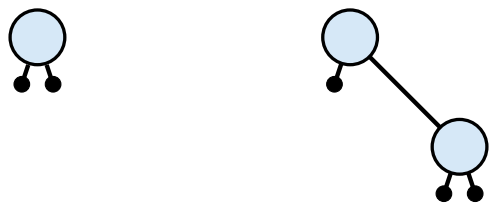
1. ein AVL-Baum der Höhe $h = 1$ enthält mindestens einen internen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. ein AVL-Baum der Höhe $h = 2$ enthält mindestens zwei interne Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

# AVL trees

## Beweis (cont.)

**Induktionsanfang:**

1. ein AVL-Baum der Höhe $h = 1$ enthält mindestens einen internen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. ein AVL-Baum der Höhe $h = 2$ enthält mindestens zwei interne Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

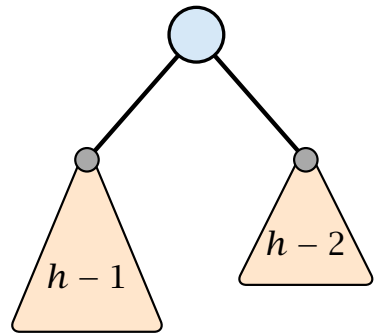


**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.

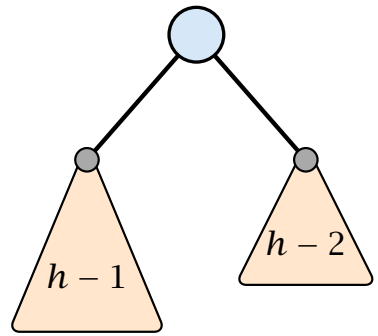
**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.



**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.

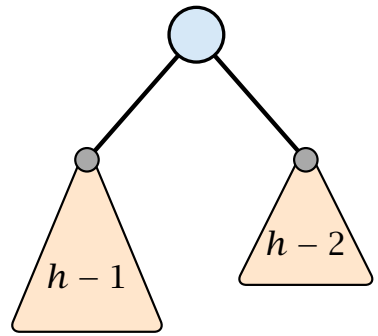



Sei

$$g_h := 1 + \text{minimale Größe von AVL-Baum mit Höhe } h \ .$$

**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.



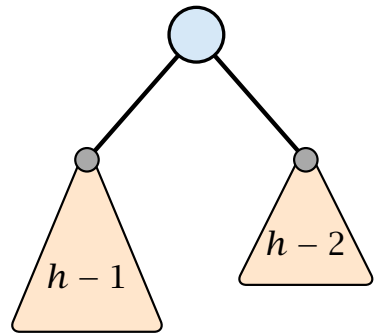

Sei

$$g_h := 1 + \text{minimale Größe von AVL-Baum mit Höhe } h \ .$$

Dann

**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.




Sei

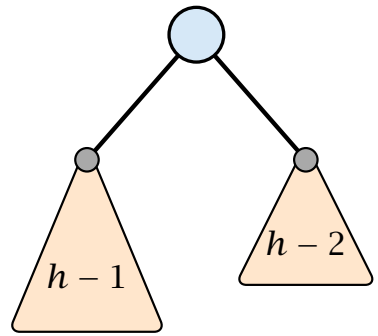
$$g_h := 1 + \text{minimale Größe von AVL-Baum mit Höhe } h \ .$$

Dann

$$g_1 = 2 \qquad\qquad = F_3$$

**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.




Sei

$$g_h := 1 + \text{minimale Größe von AVL-Baum mit Höhe } h \ .$$

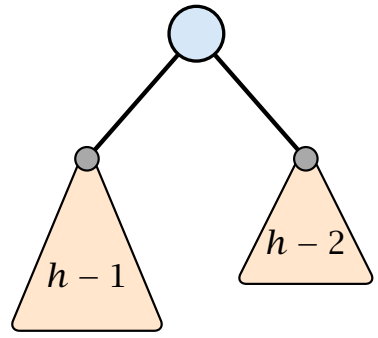
Dann

$$g_1 = 2 \qquad = F_3$$

$$g_2 = 3 \qquad = F_4$$

**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.




Sei

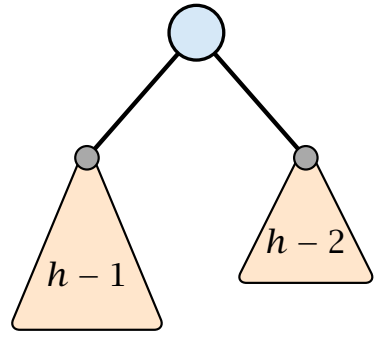
$$g_h := 1 + \text{minimale Größe von AVL-Baum mit Höhe } h \ .$$

Dann

$$\begin{aligned} g_1 &= 2 && = F_3 \\ g_2 &= 3 && = F_4 \\ g_h - 1 &= 1 + g_{h-1} - 1 + g_{h-2} - 1\,, && \text{also} \end{aligned}$$

**Induktionsschritt ($h-1, h-2 \to n$):**

Ein minimaler AVL-Baum der Höhe $h \geq 2$ hat eine Wurzel mit zwei Teilbäumen — einer mit Höhe $h-1$ und einer mit Höhe $h-2$. Beide sind minimal.




Sei

$$g_h := 1 + \text{minimale Größe von AVL-Baum mit Höhe } h \ .$$

Dann

$$\begin{aligned} g_1 &= 2 && = F_3 \\ g_2 &= 3 && = F_4 \\ g_h - 1 &= 1 + g_{h-1} - 1 + g_{h-2} - 1\,, && \text{also} \\ g_h &= g_{h-1} + g_{h-2} && = F_{h+2} \end{aligned}$$

# 7.5 AVL-Bäume

Ein AVL-Baum der Höhe $h$ enthält mindestens $F_{h+2} - 1$ interne Knoten.

Da

$$n + 1 \geq F_{h+2} = \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \ ,$$

erhalten wir

$$n \geq \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \ ,$$

und daher $h = \mathcal{O}(\log n)$.

# 7.5 AVL-Bäume

Wir müssen die Balancierungsbedingung aufrechterhalten.

Dafür speichern wir an jedem internen Knoten $v$ die Balance des Knotens. Sei $v$ ein Baumknoten mit linkem Kind $c_\ell$ und rechtem Kind $c_r$.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \ ,$$

wobei $T_{c_\ell}$ und $T_{c_r}$, die Teilbäume mit Wurzeln $c_\ell$ und $c_r$ sind.

# 7.5 AVL-Bäume

Wir müssen die Balancierungsbedingung aufrechterhalten.

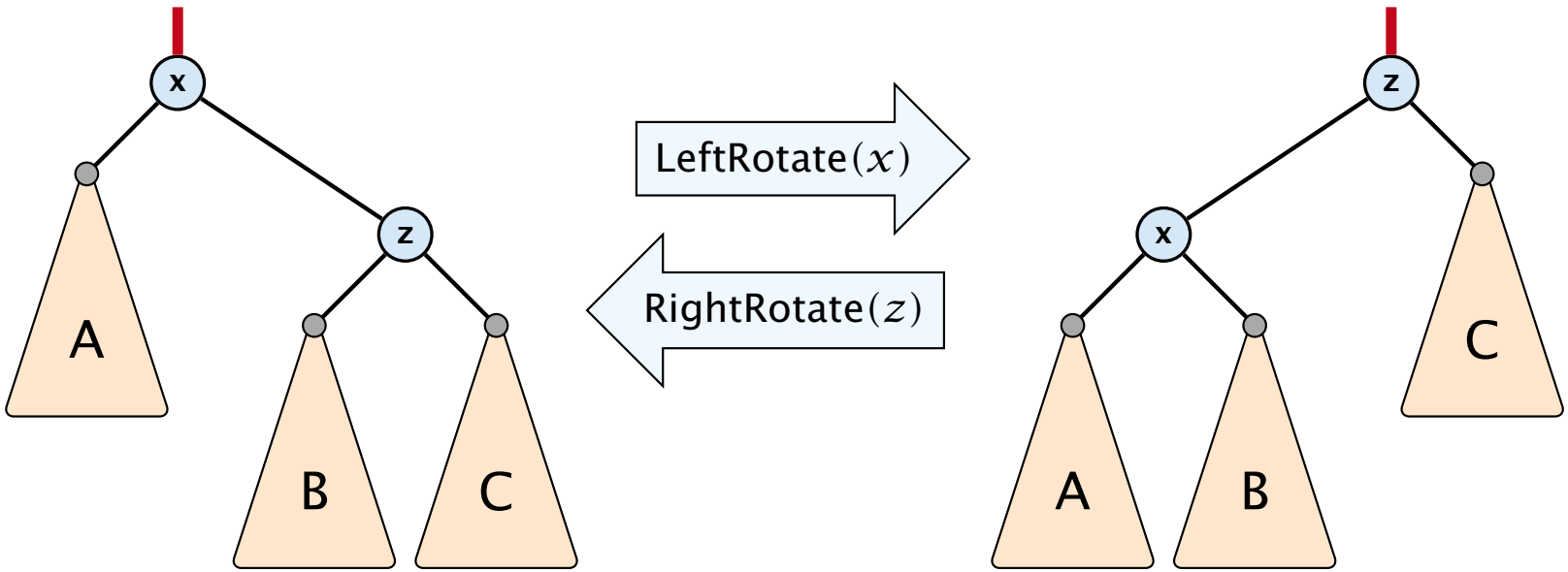
Dafür speichern wir an jedem internen Knoten $v$ die Balance des Knotens. Sei $v$ ein Baumknoten mit linkem Kind $c_\ell$ und rechtem Kind $c_r$.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \ ,$$

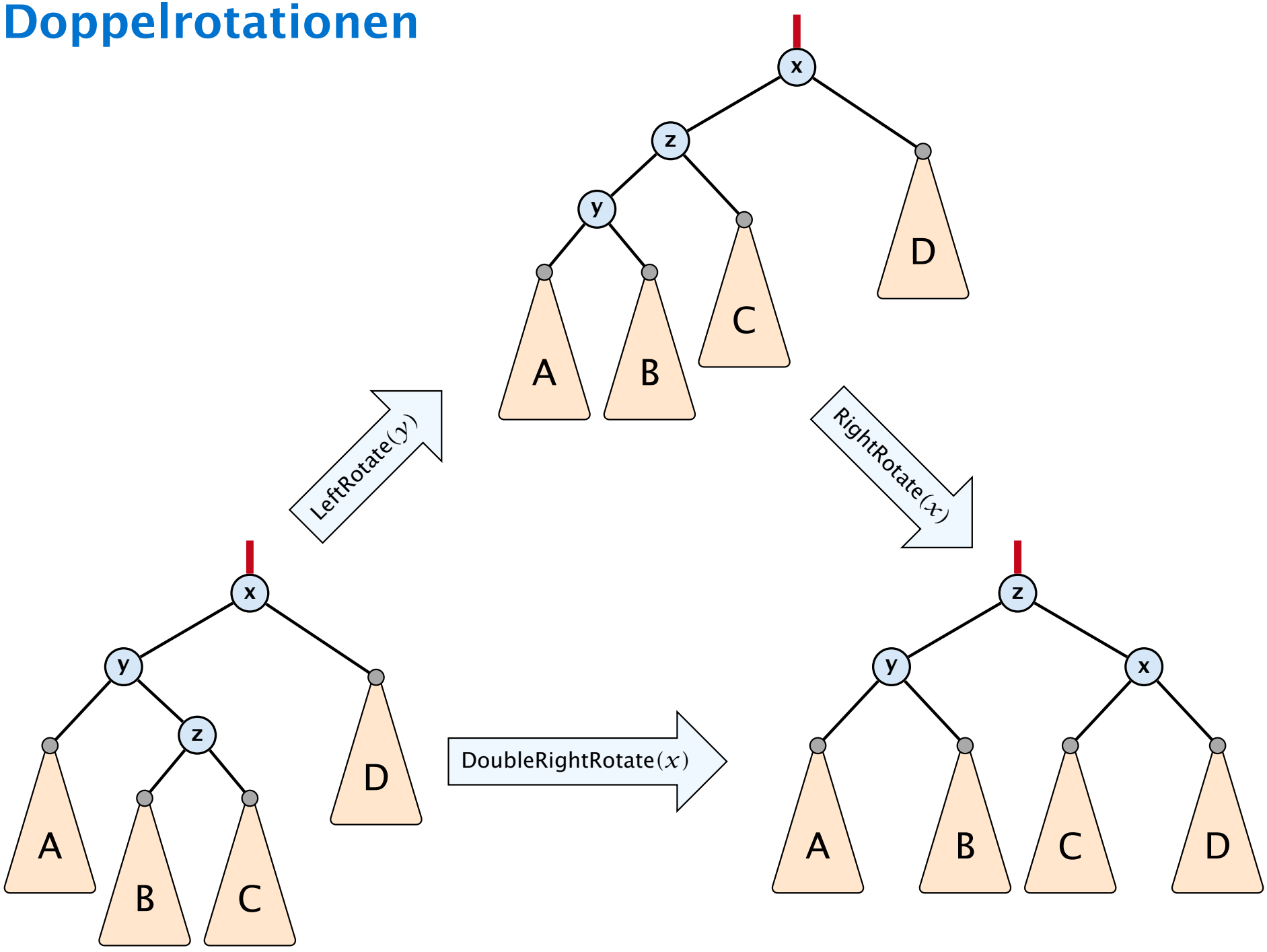
wobei $T_{c_\ell}$ und $T_{c_r}$, die Teilbäume mit Wurzeln $c_\ell$ und $c_r$ sind.

# Rotationen

Die Balancierungsbedingung wird durch Rotationen aufrechterhalten:



# Doppelrotationen



Rotationen sind lokale Operationen.

Wenn man einen Zeiger auf einen Baumknoten gegeben hat kann man eine Rotation um diesen Knoten in konstanter Zeit durchführen (das setzt natürlich voraus, dass man `parent`-Zeiger an jedem Baumknoten hat).

# AVL-Bäume: Einfügen

Beachte, dass vor dem Einfügen $w$ ein Level über dem Blattlevel lag, da $x$ ein Dummyblatt ersetzt hat, dass ein Kind von $w$ war.

- Füge wie in einem binären Suchbaum ein.

# AVL-Bäume: Einfügen

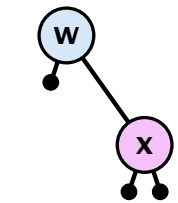
Beachte, dass vor dem Einfügen $w$ ein Level über dem Blattlevel lag, da $x$ ein Dummyblatt ersetzt hat, dass ein Kind von $w$ war.

- Füge wie in einem binären Suchbaum ein.
- Sei $w$ der Elternknoten des neuen Knotens $x$.

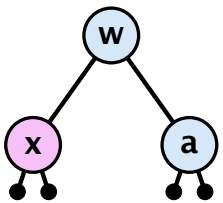
# AVL-Bäume: Einfügen

Beachte, dass vor dem Einfügen $w$ ein Level über dem Blattlevel lag, da $x$ ein Dummyblatt ersetzt hat, dass ein Kind von $w$ war.

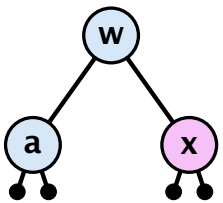
- Füge wie in einem binären Suchbaum ein.
- Sei $w$ der Elternknoten des neuen Knotens $x$.
- Es gilt einer der folgenden Fälle:



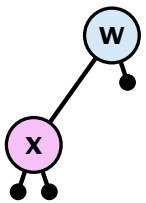
$\text{bal}(w) = -1$



$\text{bal}(w) = 0$



$\text{bal}(w) = 0$

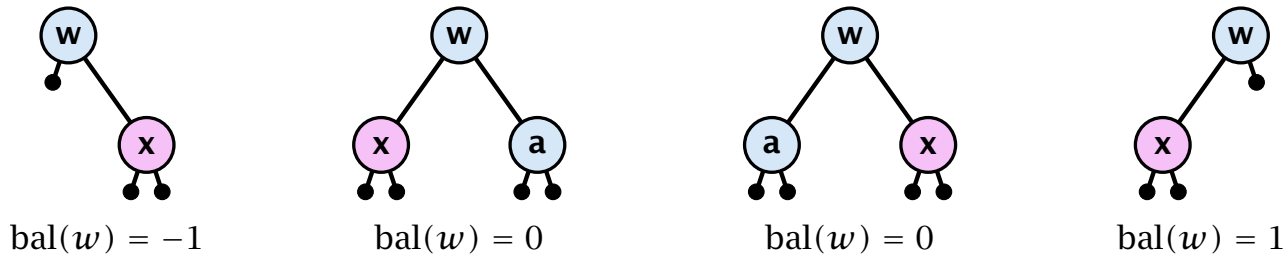


$\text{bal}(w) = 1$

# AVL-Bäume: Einfügen

Beachte, dass vor dem Einfügen $w$ ein Level über dem Blattlevel lag, da $x$ ein Dummyblatt ersetzt hat, dass ein Kind von $w$ war.

- Füge wie in einem binären Suchbaum ein.
- Sei $w$ der Elternknoten des neuen Knotens $x$.
- Es gilt einer der folgenden Fälle:

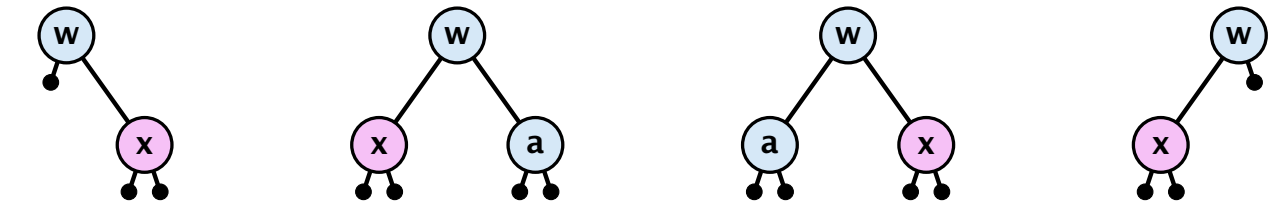



- Falls $\mathrm{bal}[w] \neq 0$, hat $T_w$ seine Höhe geändert; die Balancierungsbedingung könnte an Vorgängern von $w$ verletzt sein.

# AVL-Bäume: Einfügen

Beachte, dass vor dem Einfügen $w$ ein Level über dem Blattlevel lag, da $x$ ein Dummyblatt ersetzt hat, dass ein Kind von $w$ war.

- Füge wie in einem binären Suchbaum ein.
- Sei $w$ der Elternknoten des neuen Knotens $x$.
- Es gilt einer der folgenden Fälle:




$\mathrm{bal}(w) = -1$ $\mathrm{bal}(w) = 0$ $\mathrm{bal}(w) = 0$ $\mathrm{bal}(w) = 1$

- Falls $\mathrm{bal}[w] \neq 0$, hat $T_w$ seine Höhe geändert; die Balancierungsbedingung könnte an Vorgängern von $w$ verletzt sein.
- Wir rufen `AVL_fix_up_insert(w->parent)` um diese wiederherzustellen.

# AVL-Bäume: Einfügen

Diese Bedingungen gelten insbesondere für den ersten Aufruf AVL-fix-up-insert(parent[$w$]).

**Invariante zu Beginn von AVL-fix-up-insert($v$):**

1. Die Balancierungsbedingung gilt für alle Nachfolger von $v$.
2. Ein Knoten wurde in $T_c$ eingefügt, wobei $c$ ein Kind von $v$ ist.
3. $T_c$ hat seine Höhe um 1 erhöht (sonst hätten wir die fix-up Prozedur schon beendet).
4. Die Balance am Knoten $c$ erfüllt balance$[c] \in \{-1, 1\}$. Dies gilt, da ansonsten der Teilbaum $T_c$ seine Höhe nicht geändert hätte.

# AVL-Bäume: Einfügen

Diese Bedingungen gelten insbesondere für den ersten Aufruf AVL-fix-up-insert(parent[$w$]).

**Invariante zu Beginn von AVL-fix-up-insert($v$):**

1. Die Balancierungsbedingung gilt für alle Nachfolger von $v$.
2. Ein Knoten wurde in $T_c$ eingefügt, wobei $c$ ein Kind von $v$ ist.
3. $T_c$ hat seine Höhe um 1 erhöht (sonst hätten wir die fix-up Prozedur schon beendet).
4. Die Balance am Knoten $c$ erfüllt balance$[c] \in \{-1, 1\}$. Dies gilt, da ansonsten der Teilbaum $T_c$ seine Höhe nicht geändert hätte.

# AVL-Bäume: Einfügen

Diese Bedingungen gelten insbesondere für den ersten Aufruf AVL-fix-up-insert(parent[$w$]).

**Invariante zu Beginn von AVL-fix-up-insert($v$):**

1. Die Balancierungsbedingung gilt für alle Nachfolger von $v$.
2. Ein Knoten wurde in $T_c$ eingefügt, wobei $c$ ein Kind von $v$ ist.
3. $T_c$ hat seine Höhe um 1 erhöht (sonst hätten wir die fix-up Prozedur schon beendet).
4. Die Balance am Knoten $c$ erfüllt balance$[c] \in \{-1, 1\}$. Dies gilt, da ansonsten der Teilbaum $T_c$ seine Höhe nicht geändert hätte.

# AVL-Bäume: Einfügen

Diese Bedingungen gelten insbesondere für den ersten Aufruf AVL-fix-up-insert(parent[$w$]).

**Invariante zu Beginn von AVL-fix-up-insert($v$):**

1. Die Balancierungsbedingung gilt für alle Nachfolger von $v$.
2. Ein Knoten wurde in $T_c$ eingefügt, wobei $c$ ein Kind von $v$ ist.
3. $T_c$ hat seine Höhe um 1 erhöht (sonst hätten wir die fix-up Prozedur schon beendet).
4. Die Balance am Knoten $c$ erfüllt balance$[c] \in \{-1, 1\}$. Dies gilt, da ansonsten der Teilbaum $T_c$ seine Höhe nicht geändert hätte.

# AVL-Bäume: Einfügen

Diese Bedingungen gelten insbesondere für den ersten Aufruf AVL-fix-up-insert($\text{parent}[w]$).

**Invariante zu Beginn von AVL-fix-up-insert($v$):**

1. Die Balancierungsbedingung gilt für alle Nachfolger von $v$.
2. Ein Knoten wurde in $T_c$ eingefügt, wobei $c$ ein Kind von $v$ ist.
3. $T_c$ hat seine Höhe um 1 erhöht (sonst hätten wir die fix-up Prozedur schon beendet).
4. Die Balance am Knoten $c$ erfüllt $\text{balance}[c] \in \{-1, 1\}$. Dies gilt, da ansonsten der Teilbaum $T_c$ seine Höhe nicht geändert hätte.

# AVL-Bäume: Einfügen

```
1 AVL_fix_up_insert(v)
2     if (v->balance ∈ {-2,2})
3         v = DoRotationInsert(v);
4     if (v->balance == 0)
5         return;
6     AVL_fix_up_insert(v->parent);
```

Wir zeigen, dass dieses Verfahren korrekt ist, und dass es höchstens eine Rotation ausführt.

# AVL-Bäume: Einfügen

```
1  DoRotationInsert(v)
2      if (v->balance == -2) // insert in right sub-tree
3          if (v->right->balance ∈ {0,-1})
4              v = LeftRotate(v);
5          else
6              v = DoubleLeftRotate(v);
7      else // insert in left sub-tree
8          if (v->left->balance ∈ {0,1})
9              v = RightRotate(v);
10         else
11             v = DoubleRightRotate(v);
12     return v;
```

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Die Invariante für die fix-up Routine gilt solange wie keine Rotationen durchgeführt werden.

Wir zeigen, dass nach einer Rotation **all** Balancebedingungen erfüllt sind.

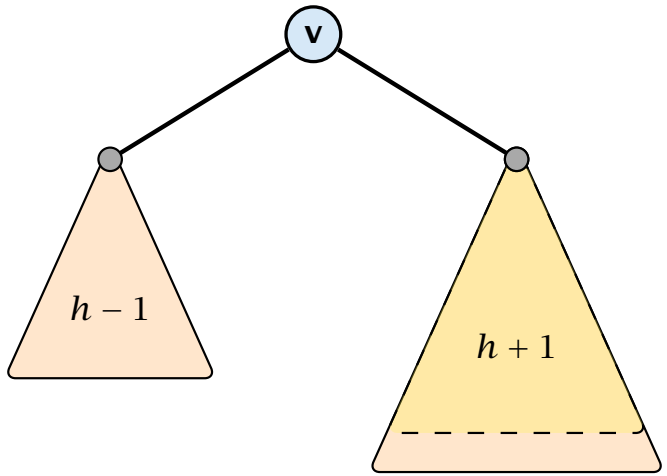
Wir zeigen dass nach einer Rotation an $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ immer noch ihre Bedingung erfüllen.
- Die Höhe des Teilbaums $T_v$ die gleiche ist wie vor der Einfügeoperation.

Wir betrachten nur den Fall, dass in den rechten Teilbaum von $v$ eingefügt wurde. Der andere Fall ist symmetrisch.

# AVL-Bäume: Einfügen

Wir haben die folgende Situation:

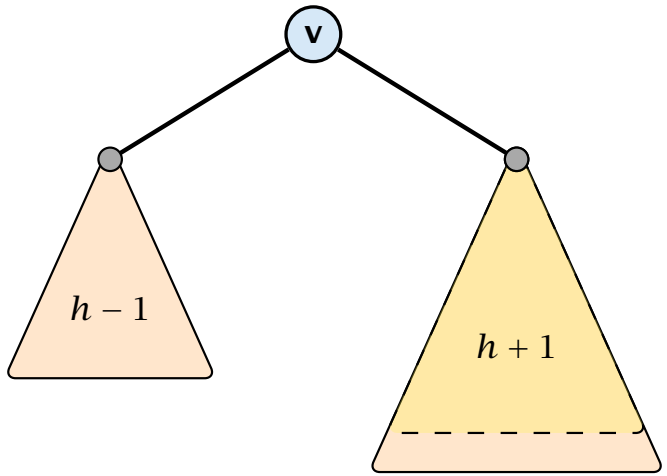



Der rechte Teilbaum von $v$ hat seine Höhe erhöht. Dadurch ensteht die Balance von $-2$ am Knoten $v$.

Vor der Einfügeoperation war die Höhe von $T_v$ gleich $h+1$.

# AVL-Bäume: Einfügen

Wir haben die folgende Situation:




Der rechte Teilbaum von $v$ hat seine Höhe erhöht. Dadurch ensteht die Balance von $-2$ am Knoten $v$.

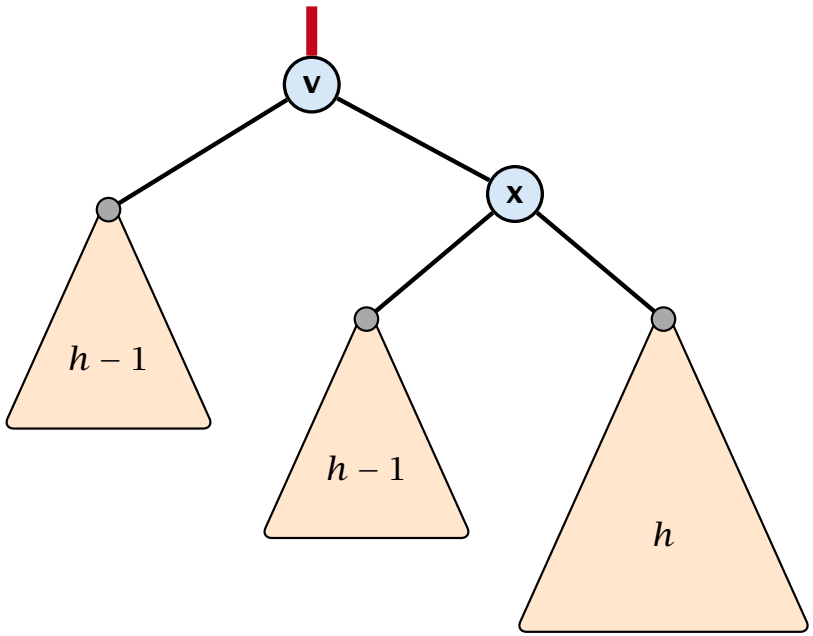
Vor der Einfügeoperation war die Höhe von $T_v$ gleich $h+1$.

# Fall 1: balance[right[$v$]] $= -1$

Linksrotation um $v$

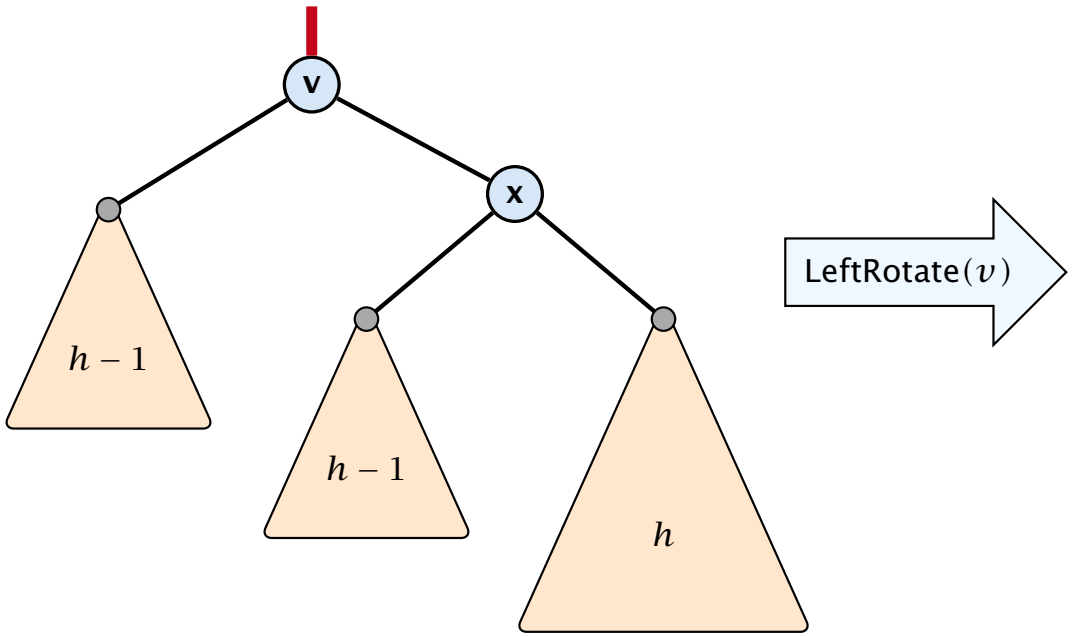
# Fall 1: balance[right[$v$]] $= -1$

Linksrotation um $v$



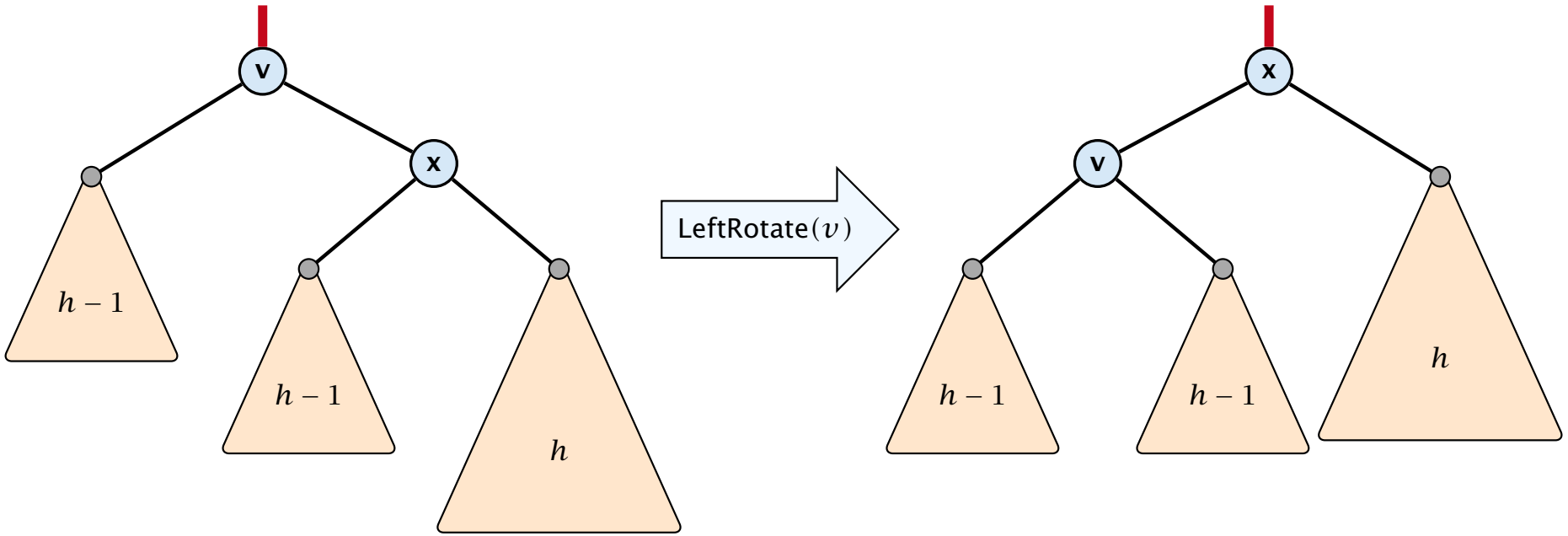
# Fall 1: balance[right[$v$]] $= -1$

Linksrotation um $v$



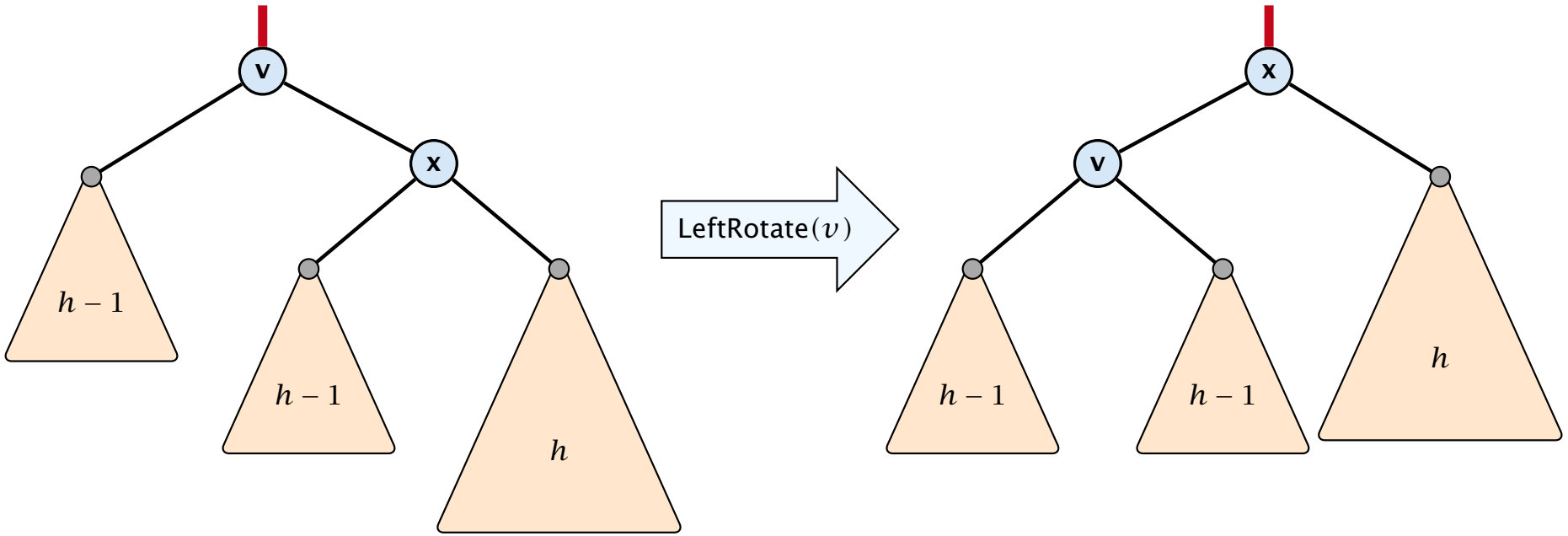
# Fall 1: balance[right[$v$]] = $-1$

Linksrotation um $v$



# Fall 1: balance[right[$v$]] $= -1$

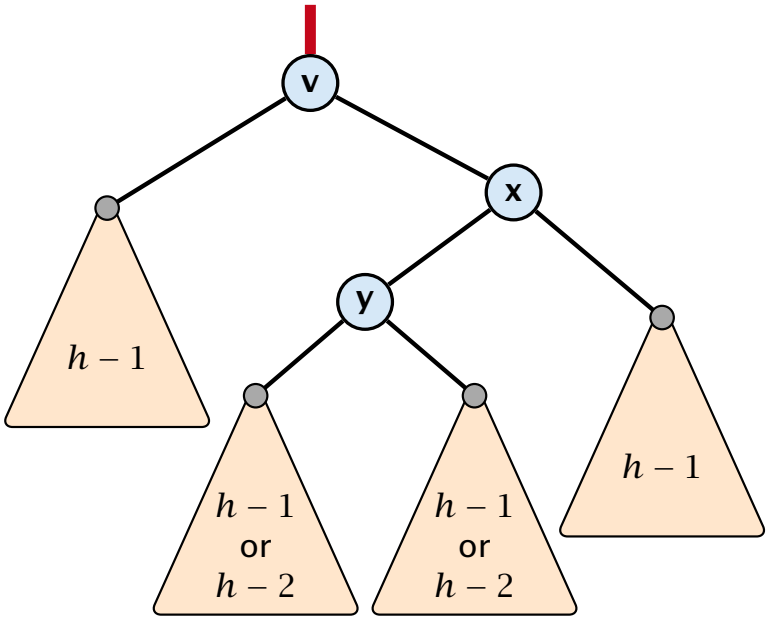
Linksrotation um $v$



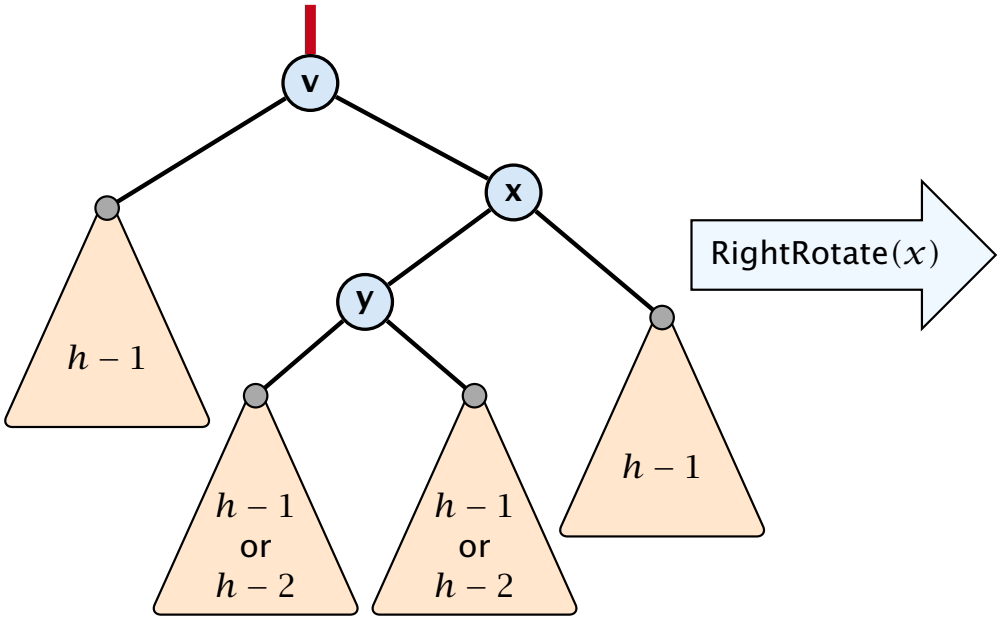

Der Teilbaum $T_v$ hat jetzt Höhe $h + 1$ wie vor dem Einfügen.

# Fall 2: balance[right[$v$]] = 1

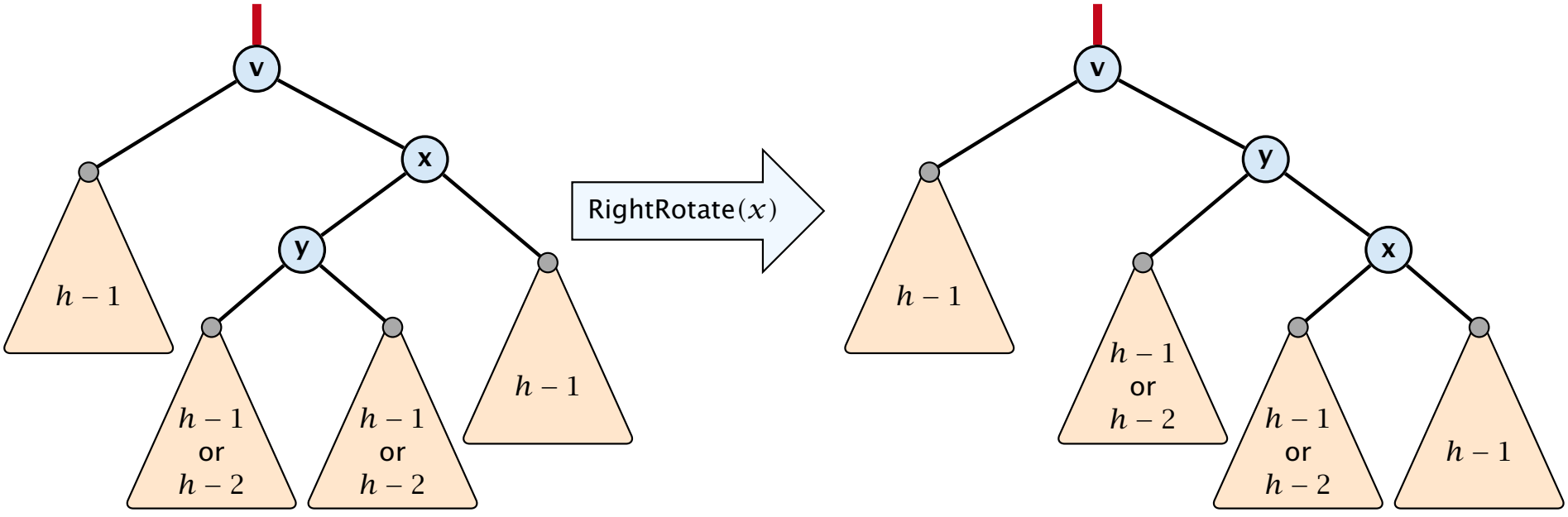
# Fall 2: balance[right[$v$]] = 1



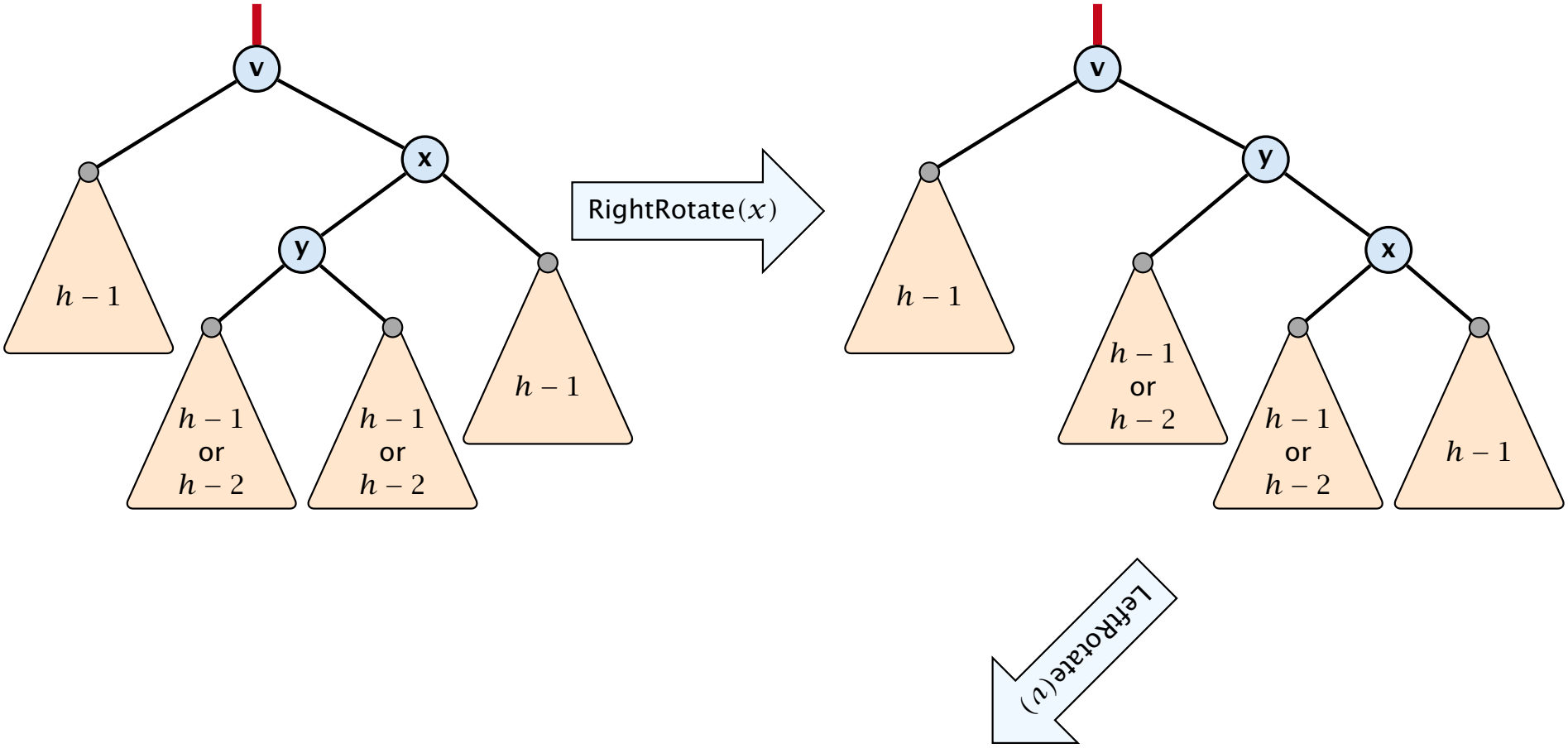
# Fall 2: balance[right[$v$]] = 1



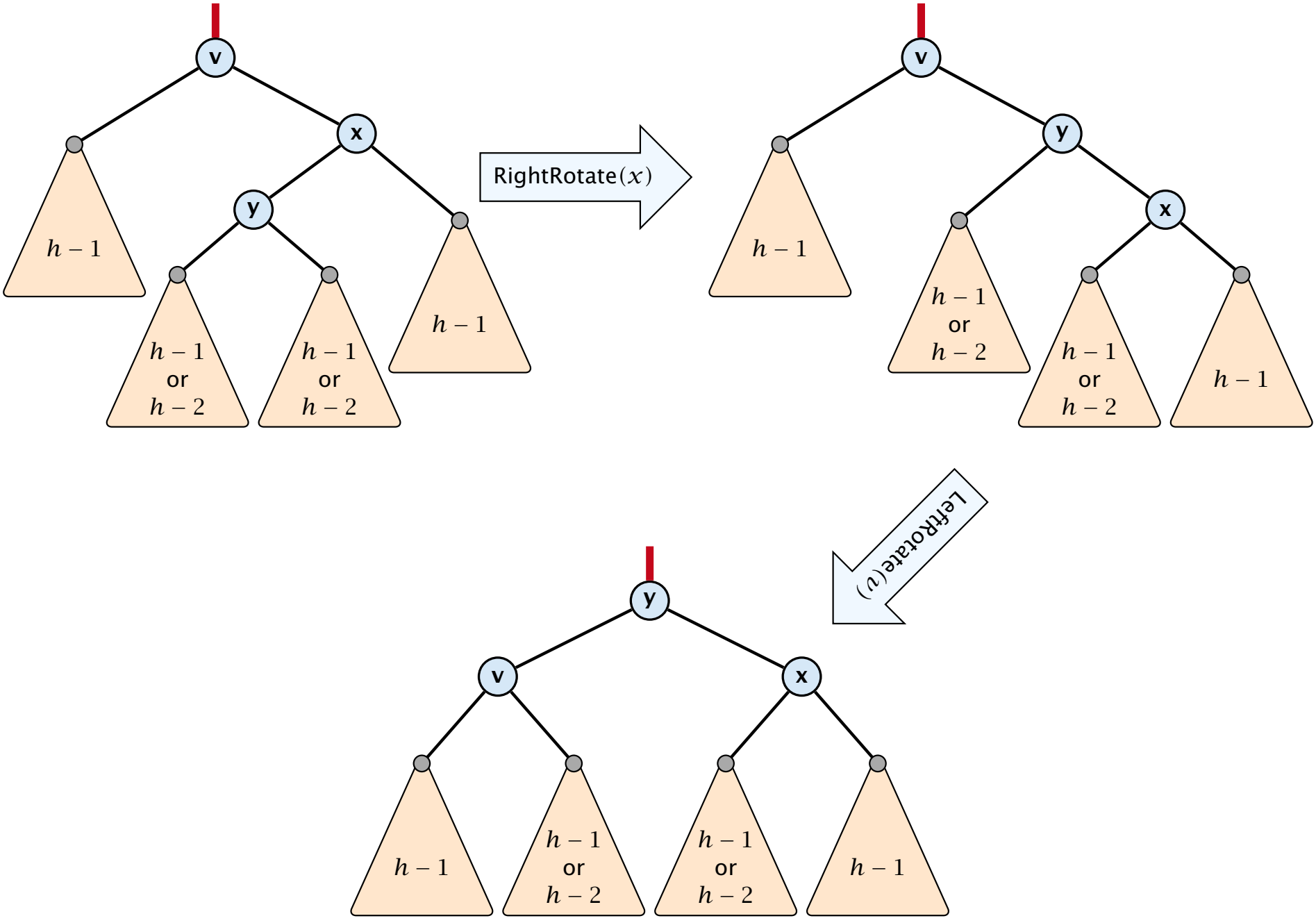
# Fall 2: balance[right[$v$]] = 1



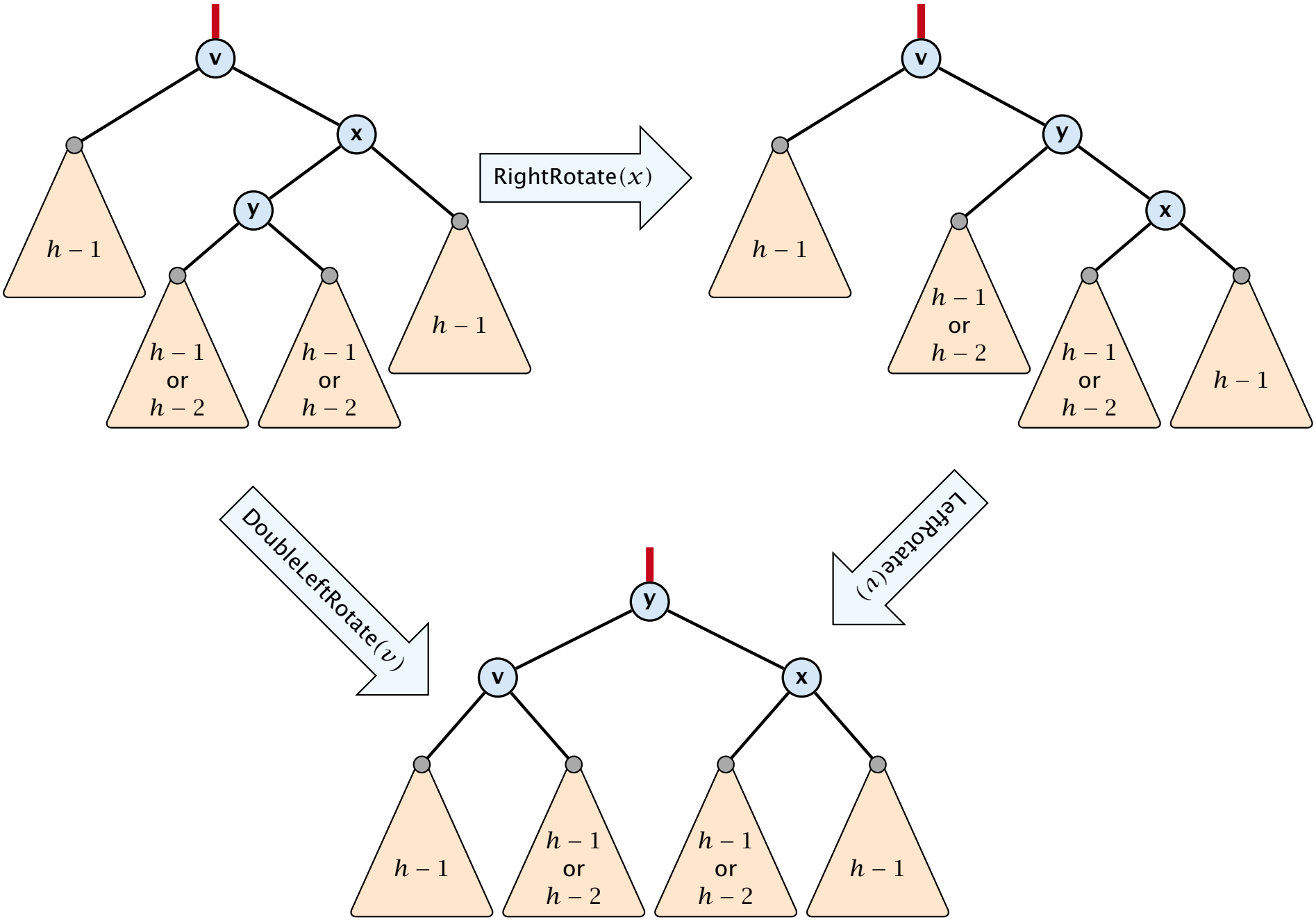
# Fall 2: balance[right[$v$]] = 1



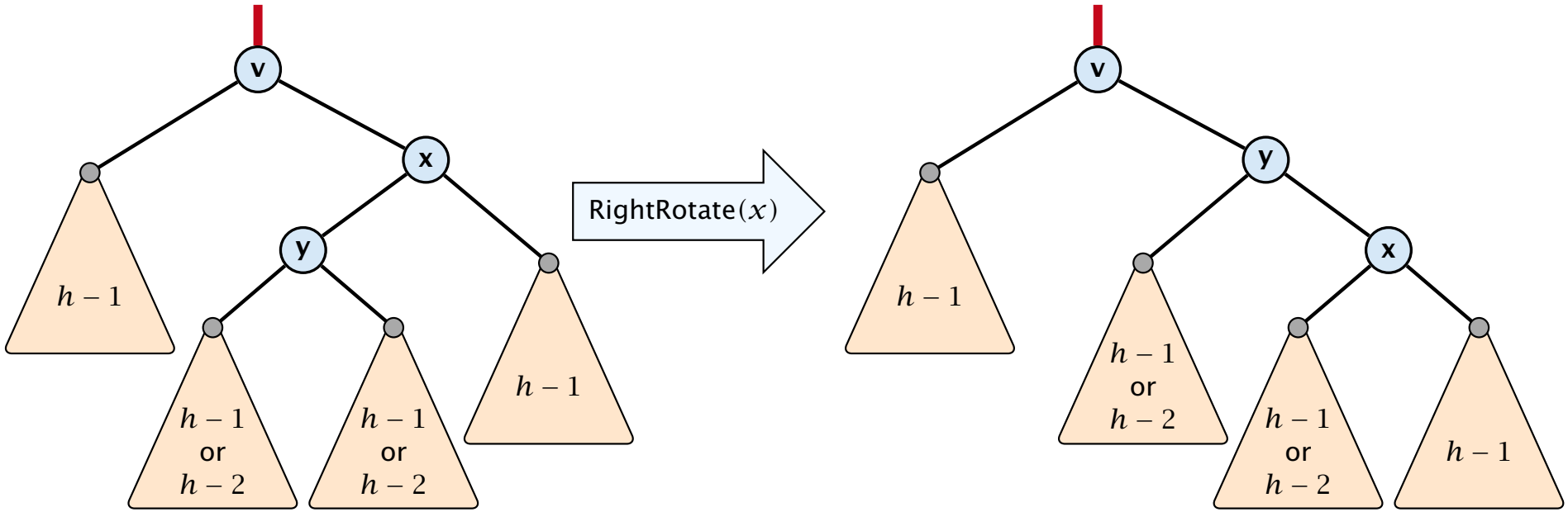
# Fall 2: balance[right[$v$]] = 1

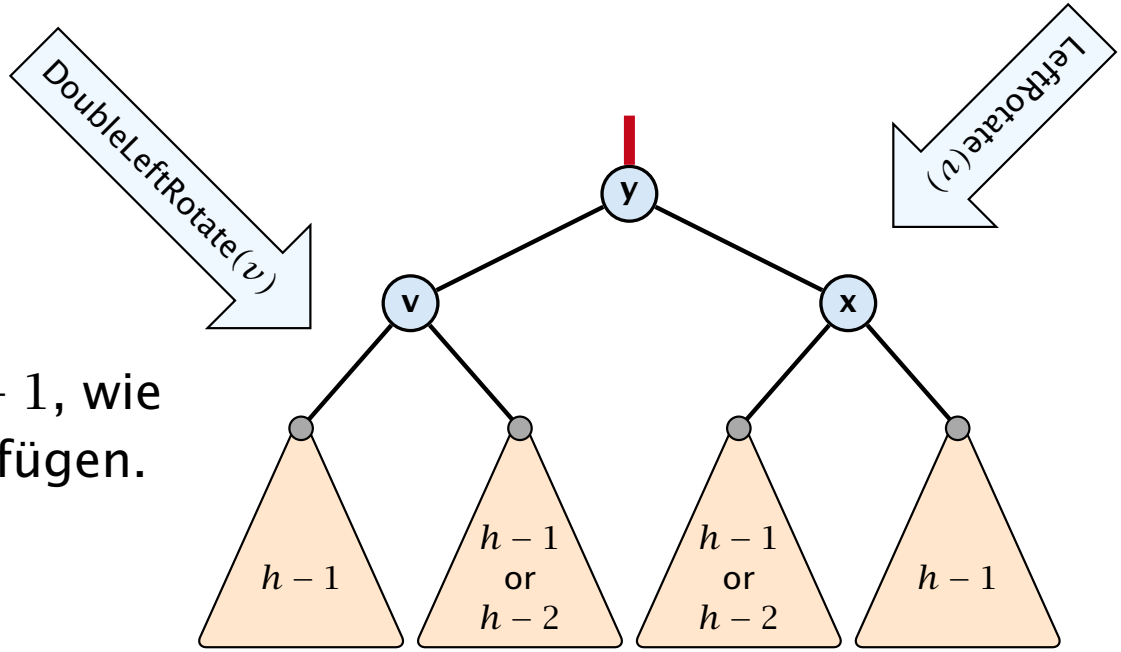


# Fall 2: balance[right[$v$]] = 1



# Fall 2: balance[right[$v$]] = 1




Höhe ist $h+1$, wie vor dem Einfügen.

# AVL-Bäume: Löschen

- Löschen wie im normalen Suchbaum.
- Sei $v$ der Elternknoten des überbrückten Knotens.
- Die Balancierungsbedingung kann an $v$, oder an Vorgängern von $v$ verletzt sein, da einer der Teilbäume der Kinder von $v$ seine Höhe verändert hat.
- Initial, ist der Knoten $c$—die neue Wurzel des Teilbaums der sich geändert hat—entweder ein Dummyblatt oder ein Knoten mit zwei Dummyblättern als Kindern.



Case 1



Case 2

In beiden Fällen gilt $\mathrm{bal}[c] = 0$.

- Wir nutzen AVL-fix-up-delete($v$) um die Balancebedingungen wiederherzustellen.

# AVL-Bäume: Löschen

- Löschen wie im normalen Suchbaum.
- Sei $v$ der Elternknoten des überbrückten Knotens.
- Die Balancierungsbedingung kann an $v$, oder an Vorgängern von $v$ verletzt sein, da einer der Teilbäume der Kinder von $v$ seine Höhe verändert hat.
- Initial, ist der Knoten $c$—die neue Wurzel des Teilbaums der sich geändert hat—entweder ein Dummyblatt oder ein Knoten mit zwei Dummyblättern als Kindern.




Case 1




Case 2

In beiden Fällen gilt $\mathrm{bal}[c] = 0$.

- Wir nutzen AVL-fix-up-delete($v$) um die Balancebedingungen wiederherzustellen.

# AVL-Bäume: Löschen

- Löschen wie im normalen Suchbaum.
- Sei $v$ der Elternknoten des überbrückten Knotens.
- Die Balancierungsbedingung kann an $v$, oder an Vorgängern von $v$ verletzt sein, da einer der Teilbäume der Kinder von $v$ seine Höhe verändert hat.
- Initial, ist der Knoten $c$—die neue Wurzel des Teilbaums der sich geändert hat—entweder ein Dummyblatt oder ein Knoten mit zwei Dummyblättern als Kindern.



Case 1



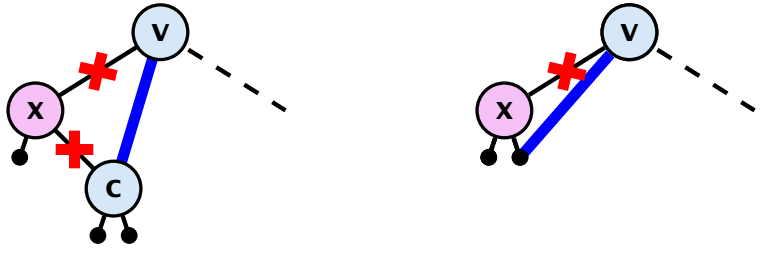
Case 2

In beiden Fällen gilt $\mathrm{bal}[c] = 0$.

- Wir nutzen AVL-fix-up-delete($v$) um die Balancebedingungen wiederherzustellen.

# AVL-Bäume: Löschen

- Löschen wie im normalen Suchbaum.
- Sei $v$ der Elternknoten des überbrückten Knotens.
- Die Balancierungsbedingung kann an $v$, oder an Vorgängern von $v$ verletzt sein, da einer der Teilbäume der Kinder von $v$ seine Höhe verändert hat.
- Initial, ist der Knoten $c$—die neue Wurzel des Teilbaums der sich geändert hat—entweder ein Dummyblatt oder ein Knoten mit zwei Dummyblättern als Kindern.



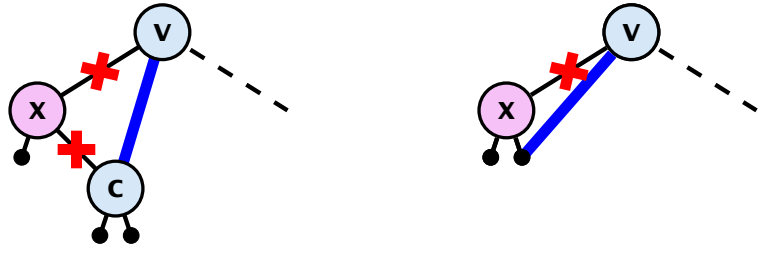

Case 1 Case 2

In beiden Fällen gilt $\mathrm{bal}[c] = 0$.

- Wir nutzen AVL-fix-up-delete($v$) um die Balancebedingungen wiederherzustellen.

# AVL-Bäume: Löschen

- Löschen wie im normalen Suchbaum.
- Sei $v$ der Elternknoten des überbrückten Knotens.
- Die Balancierungsbedingung kann an $v$, oder an Vorgängern von $v$ verletzt sein, da einer der Teilbäume der Kinder von $v$ seine Höhe verändert hat.
- Initial, ist der Knoten $c$—die neue Wurzel des Teilbaums der sich geändert hat—entweder ein Dummyblatt oder ein Knoten mit zwei Dummyblättern als Kindern.




Case 1 Case 2

In beiden Fällen gilt $\mathrm{bal}[c] = 0$.

- Wir nutzen AVL-fix-up-delete($v$) um die Balancebedingungen wiederherzustellen.

# AVL-Bäume: Löschen

**Invariante zu Beginn von AVL-fix-up-delete($v$):**

1. Die Balancebedingungen gelten für alle Nachfolger von $v$.
2. Ein Knoten ist im Teilbaum $T_c$ entfernt worden, wobei $c$ entweder linkes oder rechtes Kind von $v$ ist
3. $T_c$ hat seine Höhe um eins reduziert.
4. Die Balance am Knoten $c$ erfüllt $\text{balance}[c] = 0$. Dies gilt, da wir zeigen werden, dass im Fall $\text{balance}[c] \in \{-1, 1\}$ der Baum $T_c$ seine Höhe nicht geändert hat und das deshalb die Prozedur schon abgebrochen worden wäre.

# AVL-Bäume: Löschen

**Invariante zu Beginn von AVL-fix-up-delete($v$):**

1. Die Balancebedingungen gelten für alle Nachfolger von $v$.
2. Ein Knoten ist im Teilbaum $T_c$ entfernt worden, wobei $c$ entweder linkes oder rechtes Kind von $v$ ist
3. $T_c$ hat seine Höhe um eins reduziert.
4. Die Balance am Knoten $c$ erfüllt $\text{balance}[c] = 0$. Dies gilt, da wir zeigen werden, dass im Fall $\text{balance}[c] \in \{-1, 1\}$ der Baum $T_c$ seine Höhe nicht geändert hat und das deshalb die Prozedur schon abgebrochen worden wäre.

# AVL-Bäume: Löschen

**Invariante zu Beginn von AVL-fix-up-delete($v$):**

1. Die Balancebedingungen gelten für alle Nachfolger von $v$.
2. Ein Knoten ist im Teilbaum $T_c$ entfernt worden, wobei $c$ entweder linkes oder rechtes Kind von $v$ ist
3. $T_c$ hat seine Höhe um eins reduziert.
4. Die Balance am Knoten $c$ erfüllt $\text{balance}[c] = 0$. Dies gilt, da wir zeigen werden, dass im Fall $\text{balance}[c] \in \{-1, 1\}$ der Baum $T_c$ seine Höhe nicht geändert hat und das deshalb die Prozedur schon abgebrochen worden wäre.

# AVL-Bäume: Löschen

**Invariante zu Beginn von AVL-fix-up-delete($v$):**

1. Die Balancebedingungen gelten für alle Nachfolger von $v$.
2. Ein Knoten ist im Teilbaum $T_c$ entfernt worden, wobei $c$ entweder linkes oder rechtes Kind von $v$ ist
3. $T_c$ hat seine Höhe um eins reduziert.
4. Die Balance am Knoten $c$ erfüllt $\text{balance}[c] = 0$. Dies gilt, da wir zeigen werden, dass im Fall $\text{balance}[c] \in \{-1, 1\}$ der Baum $T_c$ seine Höhe nicht geändert hat und das deshalb die Prozedur schon abgebrochen worden wäre.

# AVL-Bäume: Löschen

```
1 AVL_fix_up_delete(v)
2     if (v->balance ∈ {-2,2})
3         v = DoRotationDelete(v);
4     if (v->balance ∈ {-1,1})
5         return;
6     AVL_fix_up_delete(v->parent);
```

Wir zeigen, dass dies korrekt ist. Eventuell benötigen wir aber eine logarithmische Anzahl an Rotationen.

# AVL-Bäume: Löschen

```
1 AVL_fix_up_delete(v)
2     if (v->balance ∈ {-2,2})
3         v = DoRotationDelete(v);
4     if (v->balance ∈ {-1,1})
5         return;
6     AVL_fix_up_delete(v->parent);
```

Wir zeigen, dass dies korrekt ist. Eventuell benötigen wir aber eine logarithmische Anzahl an Rotationen.

# AVL-Bäume: Löschen

```
1  DoRotationDelete(v)
2      if (v->balance == -2) // deletion in left sub-tree
3          if (v->right->balance ∈ {0,-1})
4              v = LeftRotate(v);
5          else
6              v = DoubleLeftRotate(v);
7      else // deletion in right sub-tree
8          if (v->left->balance ∈ {0,1})
9              v = RightRotate(v);
10         else
11             v = DoubleRightRotate(v);
12     return v;
```

Beachte, dass die Fallunterscheidung im zweiten Level (bal[right[$v$]] und bal[left[$v$]]) nicht bzgl. des Kindes $c$ gemacht wird, dessen Teilbaum $T_c$ sich geändert hat Dies ist ein Unterschied zu AVL-fix-up-insert().

# AVL-Bäume: Löschen

Die Invariante der fix-up Routine gilt solange keine Rotationen erfolgt sind.

Wir zeigen, dass nach Rotation um $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ ihre Bedingung immer noch erfüllen
- Falls jetzt $\text{balance}[v] \in \{-1, 1\}$ können wir aufhören, da der Teilbaum $T_v$ die gleiche Höhe hat wie vor der Löschoperation.

Wir betrachten nur den Fall dass der entfernte Knoten im rechten Teilbaum von $v$ war. Der andere Fall ist symmetrisch.

# AVL-Bäume: Löschen

Die Invariante der fix-up Routine gilt solange keine Rotationen erfolgt sind.

Wir zeigen, dass nach Rotation um $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ ihre Bedingung immer noch erfüllen
- Falls jetzt $\text{balance}[v] \in \{-1, 1\}$ können wir aufhören, da der Teilbaum $T_v$ die gleiche Höhe hat wie vor der Löschoperation.

Wir betrachten nur den Fall dass der entfernte Knoten im rechten Teilbaum von $v$ war. Der andere Fall ist symmetrisch.

# AVL-Bäume: Löschen

Die Invariante der fix-up Routine gilt solange keine Rotationen erfolgt sind.

Wir zeigen, dass nach Rotation um $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ ihre Bedingung immer noch erfüllen
- Falls jetzt $\text{balance}[v] \in \{-1, 1\}$ können wir aufhören, da der Teilbaum $T_v$ die gleiche Höhe hat wie vor der Löschoperation.

Wir betrachten nur den Fall dass der entfernte Knoten im rechten Teilbaum von $v$ war. Der andere Fall ist symmetrisch.

# AVL-Bäume: Löschen

Die Invariante der fix-up Routine gilt solange keine Rotationen erfolgt sind.

Wir zeigen, dass nach Rotation um $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ ihre Bedingung immer noch erfüllen
- Falls jetzt $\text{balance}[v] \in \{-1, 1\}$ können wir aufhören, da der Teilbaum $T_v$ die gleiche Höhe hat wie vor der Löschoperation.

Wir betrachten nur den Fall dass der entfernte Knoten im rechten Teilbaum von $v$ war. Der andere Fall ist symmetrisch.

# AVL-Bäume: Löschen

Die Invariante der fix-up Routine gilt solange keine Rotationen erfolgt sind.

Wir zeigen, dass nach Rotation um $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ ihre Bedingung immer noch erfüllen
- Falls jetzt $\text{balance}[v] \in \{-1, 1\}$ können wir aufhören, da der Teilbaum $T_v$ die gleiche Höhe hat wie vor der Löschoperation.

Wir betrachten nur den Fall dass der entfernte Knoten im rechten Teilbaum von $v$ war. Der andere Fall ist symmetrisch.

# AVL-Bäume: Löschen

Die Invariante der fix-up Routine gilt solange keine Rotationen erfolgt sind.

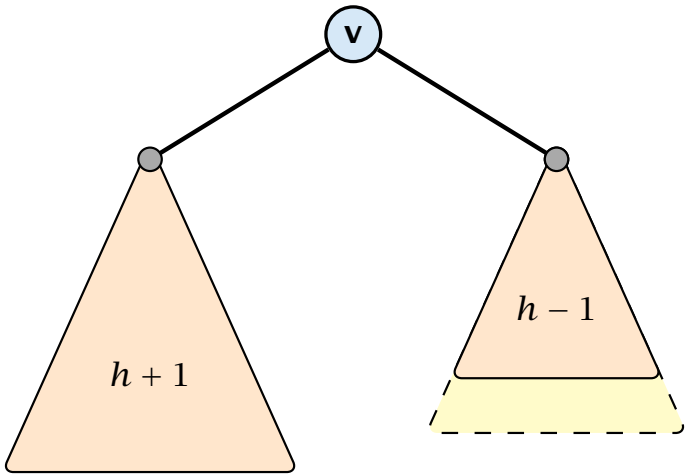
Wir zeigen, dass nach Rotation um $v$:

- $v$ seine Balancebedingung erfüllt.
- Alle Kinder von $v$ ihre Bedingung immer noch erfüllen
- Falls jetzt $\text{balance}[v] \in \{-1, 1\}$ können wir aufhören, da der Teilbaum $T_v$ die gleiche Höhe hat wie vor der Löschoperation.

Wir betrachten nur den Fall dass der entfernte Knoten im rechten Teilbaum von $v$ war. Der andere Fall ist symmetrisch.

# AVL-Bäume: Löschen

Folgende Situation:

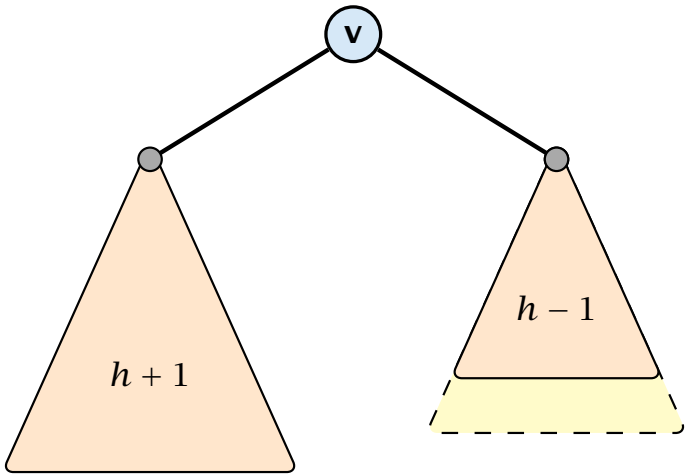



Der rechte Teilbaum von $v$ hat seine Höhe reduziert. Dies ergibt eine Balance von $2$ für $v$.

Vor der Löschoperation war die Höhe von $T_v$ gleich $h+2$.

# AVL-Bäume: Löschen

Folgende Situation:

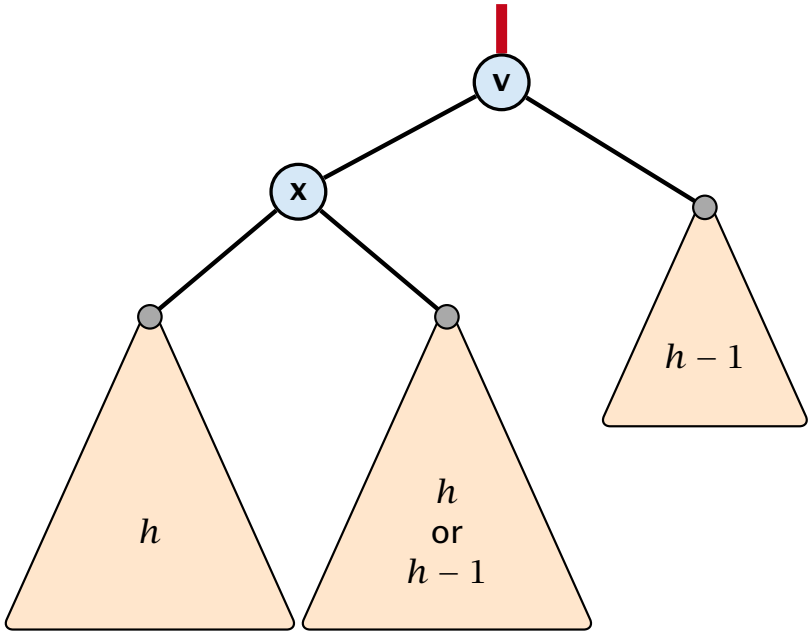



Der rechte Teilbaum von $v$ hat seine Höhe reduziert. Dies ergibt eine Balance von $2$ für $v$.

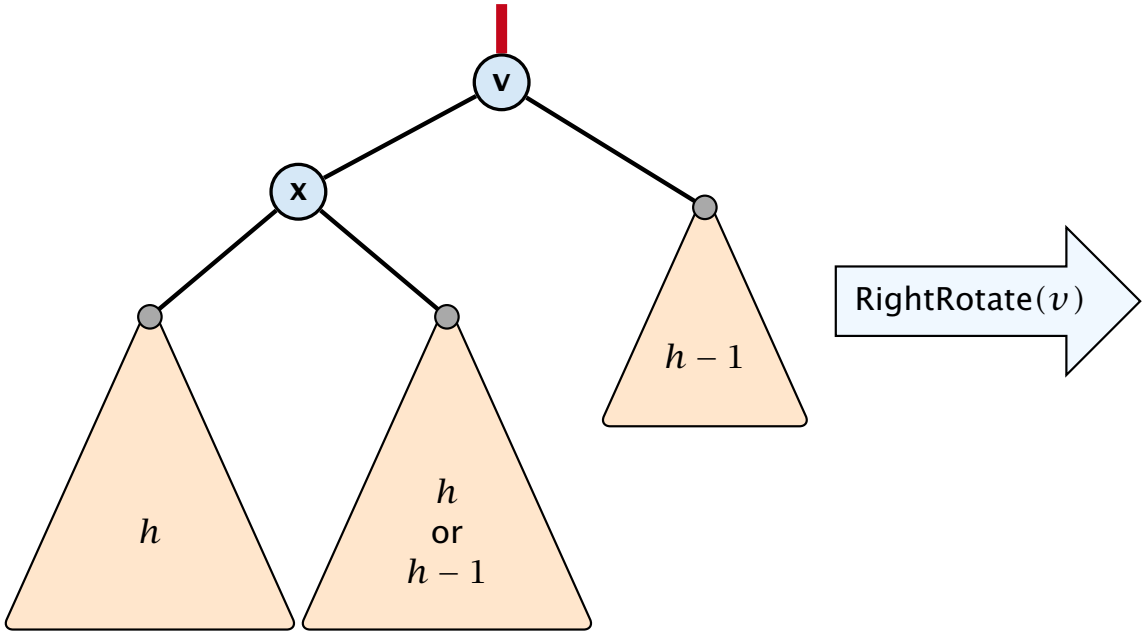
Vor der Löschoperation war die Höhe von $T_v$ gleich $h+2$.

# Fall 1: balance[left[$v$]] $\in \{0, 1\}$

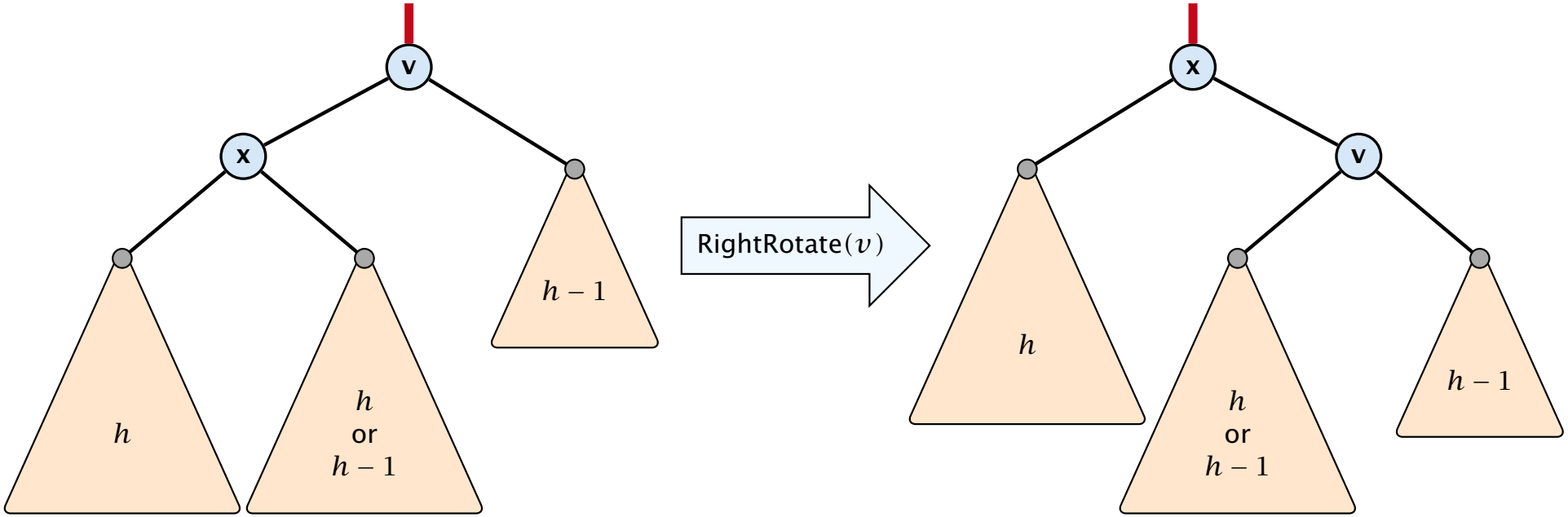
# Fall 1: balance[left[$v$]] $\in \{0, 1\}$



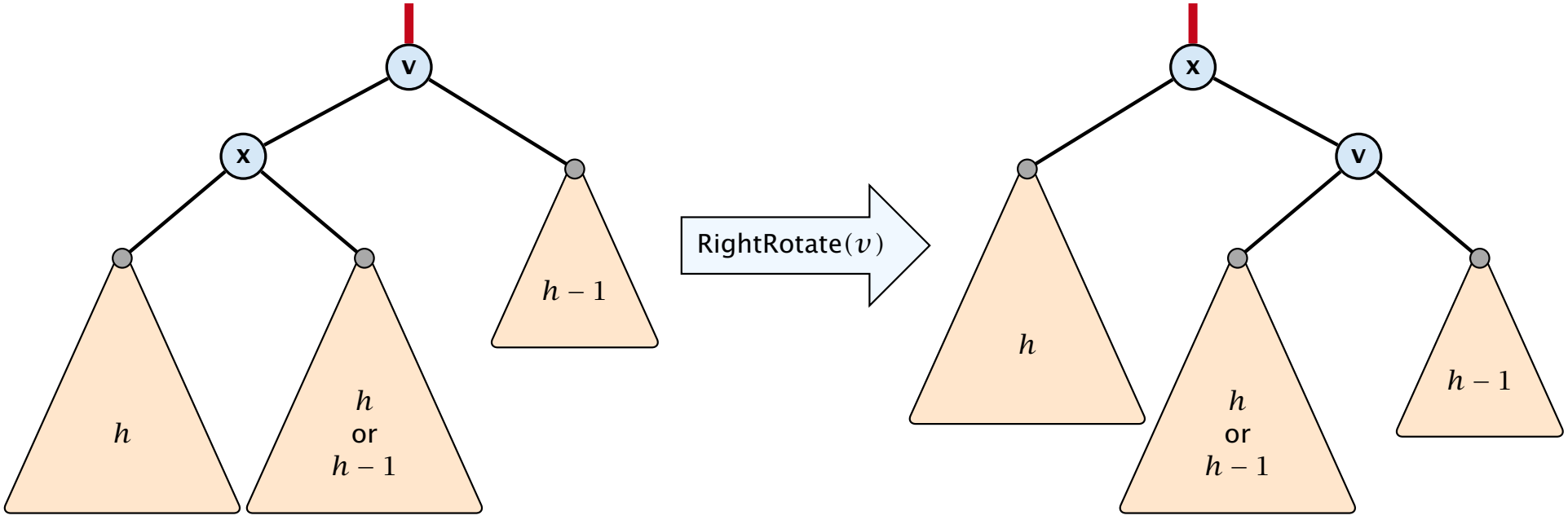
# Fall 1: balance[left[$v$]] $\in \{0, 1\}$



# Fall 1: balance[left[$v$]] $\in \{0, 1\}$

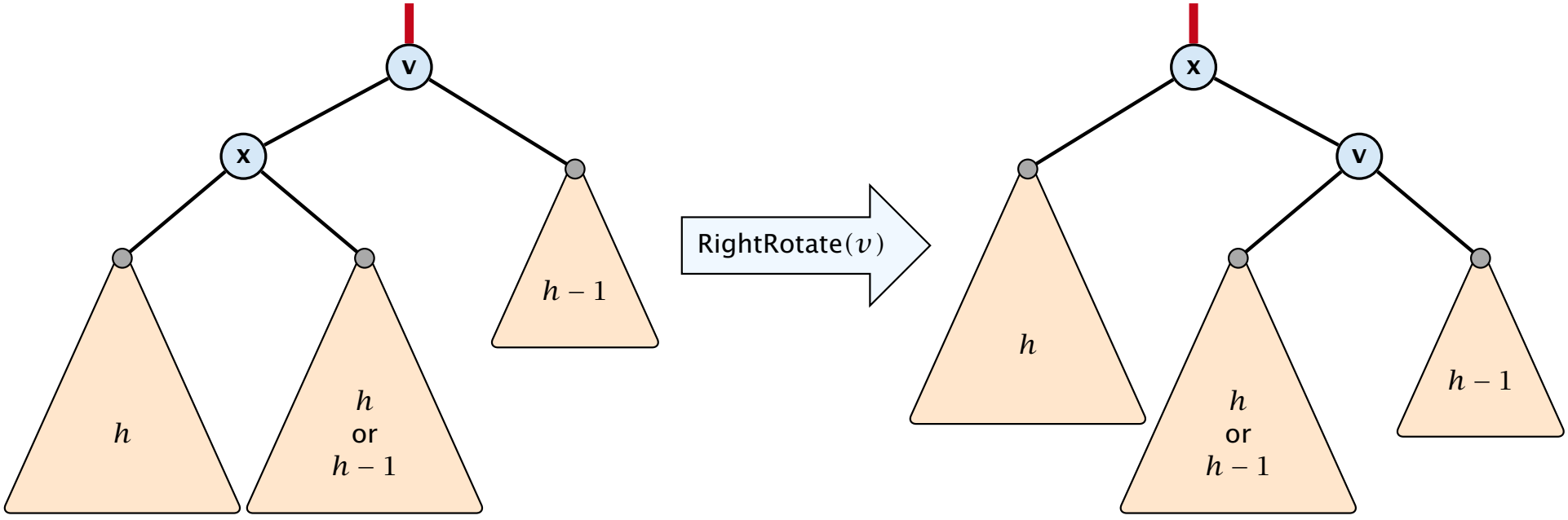


# Fall 1: balance[left[$v$]] $\in \{0, 1\}$




Falls der mittlere Teilbaum Höhe $h$ hat, hat der Gesamtaum Höhe $h + 2$ wie vor der Operation. Die Iteration bricht ab, da die Balance an der Wurzel nicht Null ist.

# Fall 1: balance[left[$v$]] $\in \{0, 1\}$

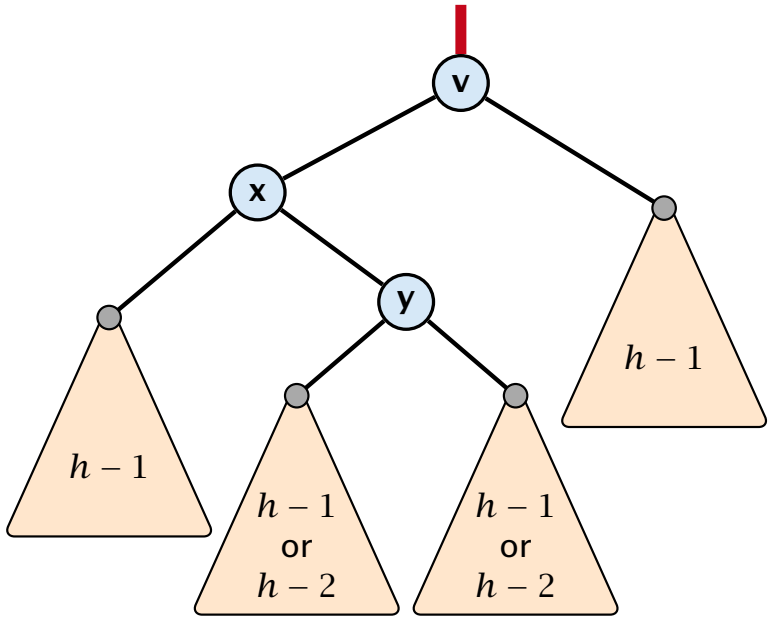



Falls der mittlere Teilbaum Höhe $h$ hat, hat der Gesamtaum Höhe $h + 2$ wie vor der Operation. Die Iteration bricht ab, da die Balance an der Wurzel nicht Null ist.

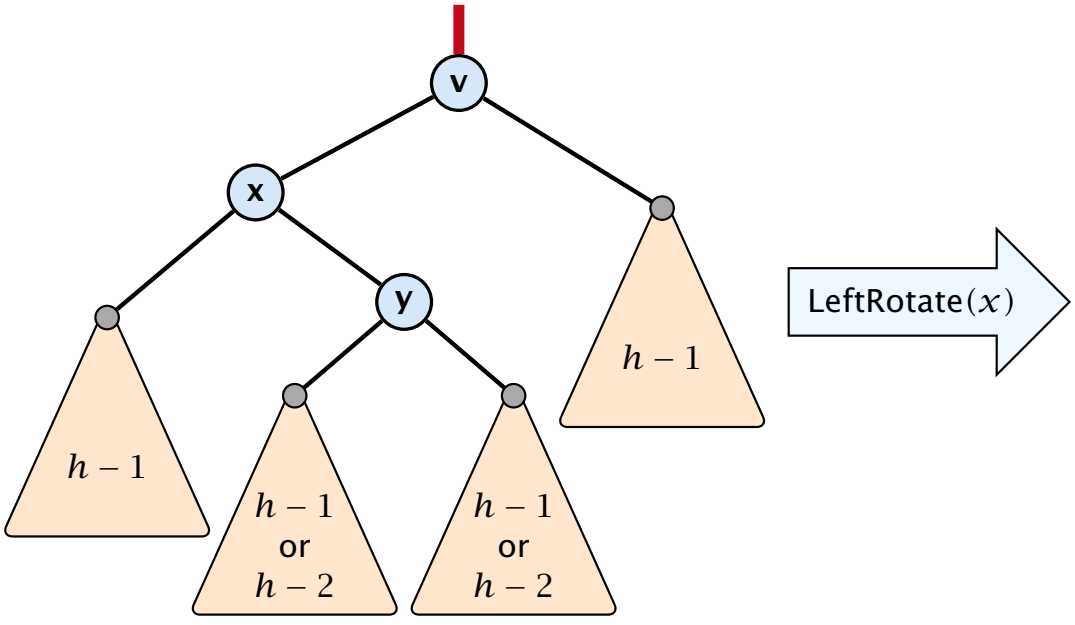
Falls der mittlere Teilbaum Höhe $h - 1$ hat, hat der Gesamtbaum die Höhe von $h + 2$ auf $h + 1$ reduziert. Wir machen mit der fix-up Routine weiter.

# Fall 2: balance[left[$v$]] $= -1$

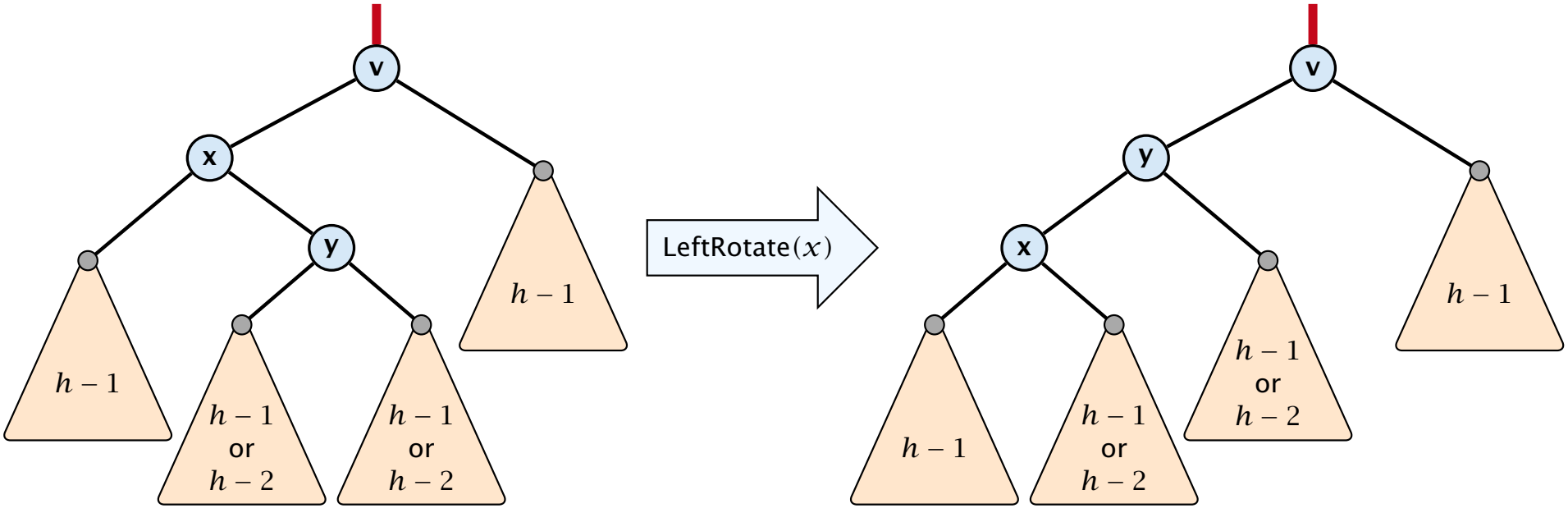
# Fall 2: balance[left[$v$]] = $-1$



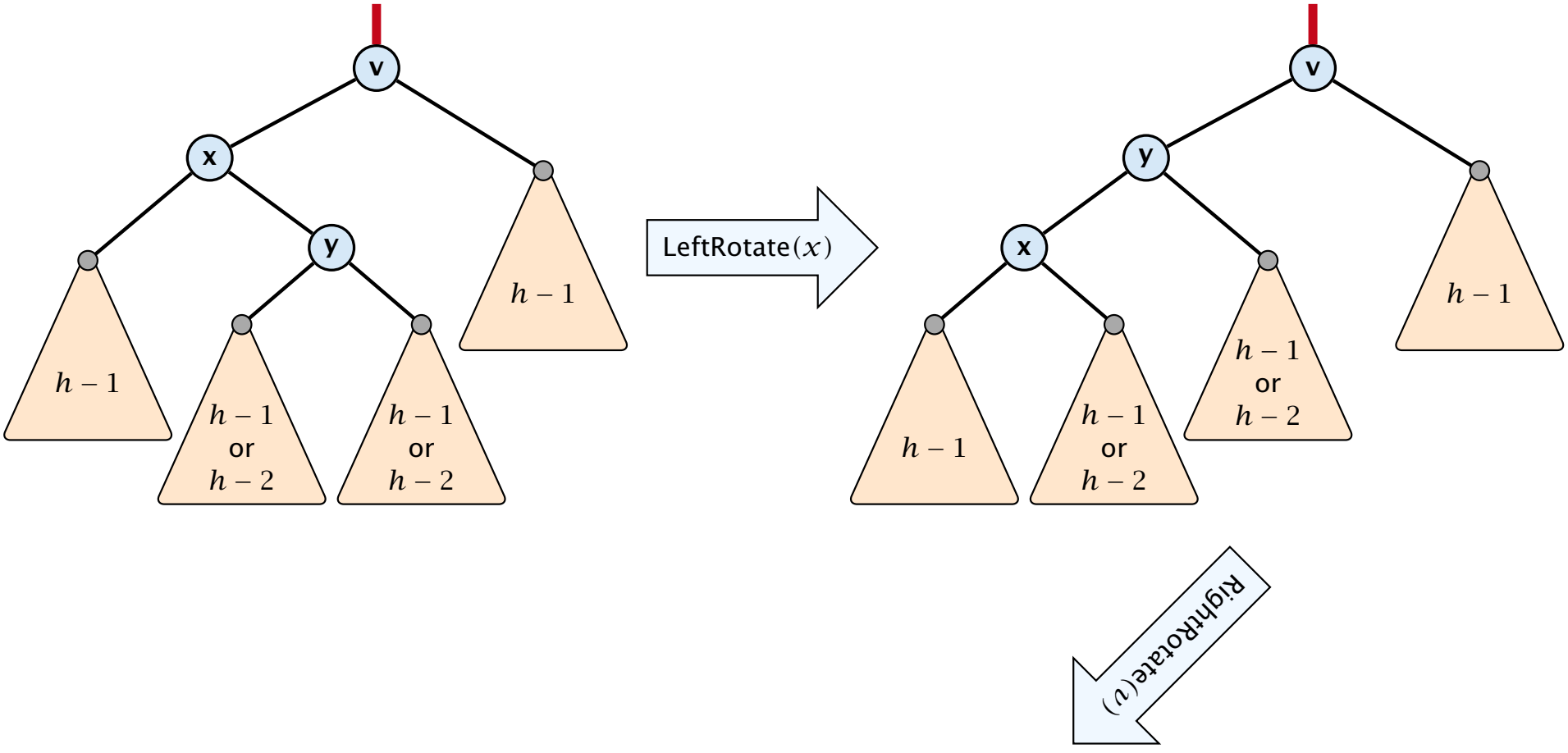
# Fall 2: balance[left[$v$]] = $-1$



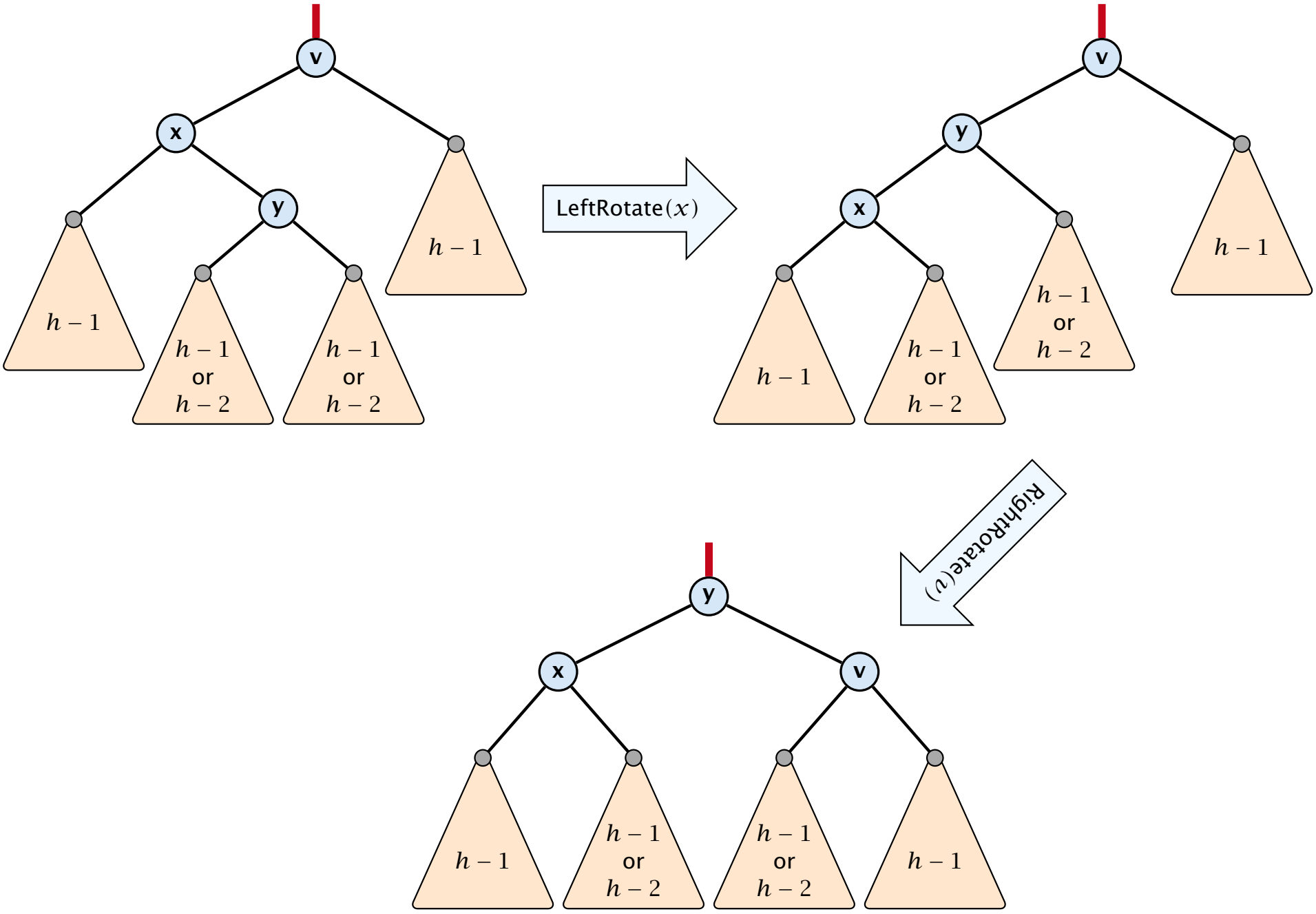
# Fall 2: balance[left[$v$]] = $-1$



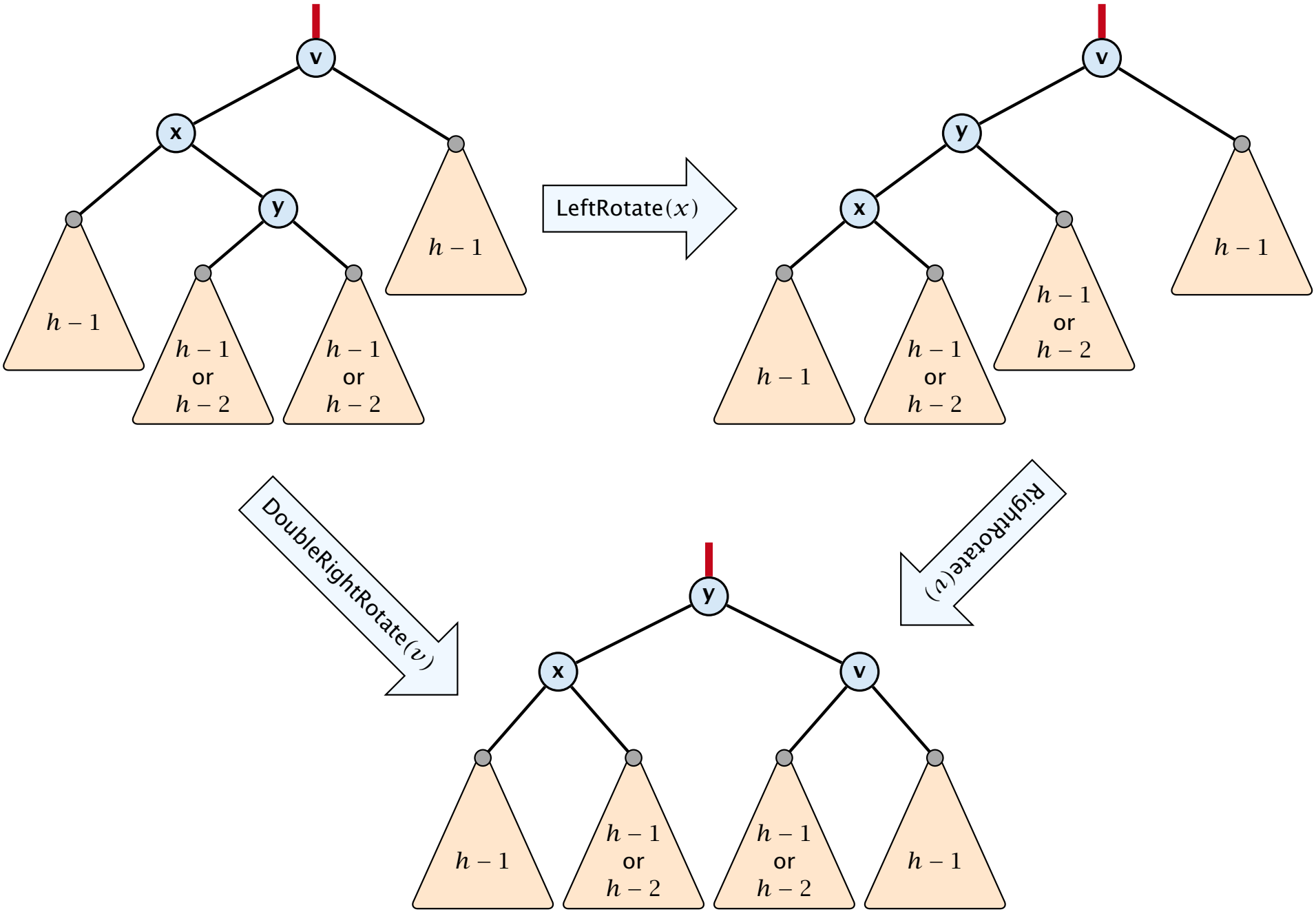
# Fall 2: balance[left[$v$]] = $-1$



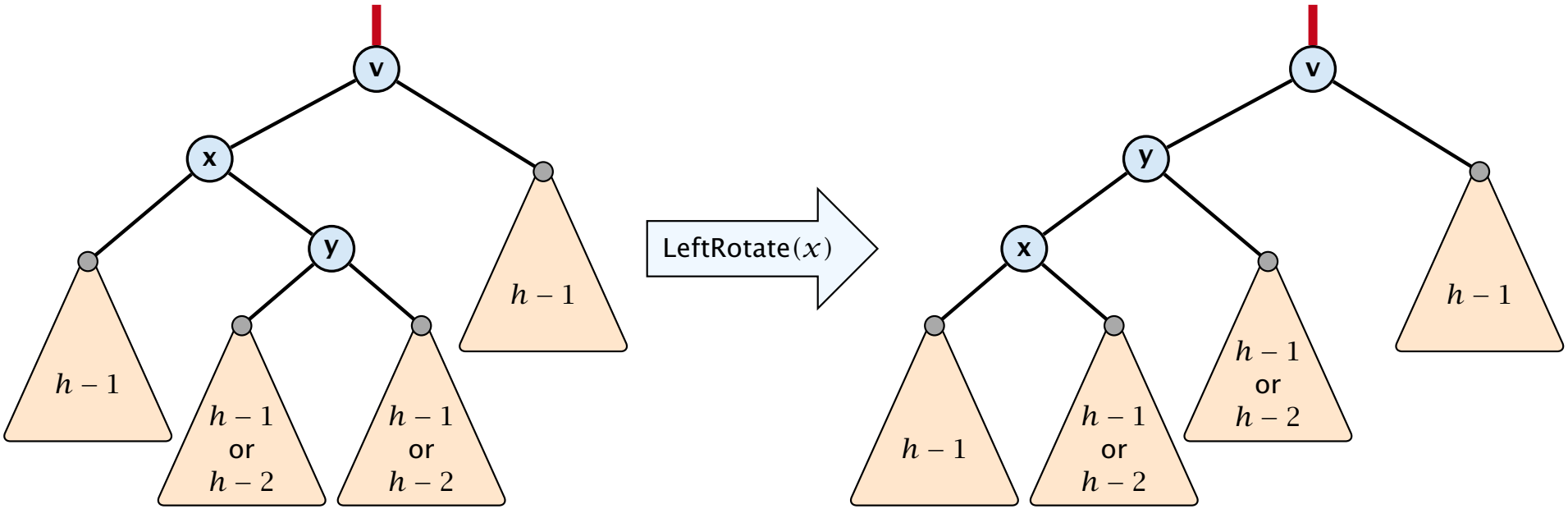
# Fall 2: balance[left[$v$]] = $-1$



# Fall 2: balance[left[$v$]] = $-1$



# Fall 2: balance[left[$v$]] $= -1$




Teilbaum hat Höhe $h + 1$ (kleiner als vorher). Die Balance an $y$ ist Null. Wir machen weiter.

# Traversierung von Graphen

Die Algorithmen für eine Breiten- und Tiefensuche auf Bäumen lassen sich auf Graphen verallgemeinern.

**Tiefensuche**
`DFSvisit(v)` besucht alle (noch nicht besuchten) Knoten, die von `v` aus erreichbar sind.

`v->dfsNum` codiert hinterher die Reihenfolge in der Knoten besucht wurden.

Zusätzlich speichert `v->finNum` die Reihenfolge, in der die rekursiven Aufrufe beendet wurden.

# Tiefensuche

```
1 DFS()
2     dfsCount = 1;
3     finCount = 1;
4     foreach v ∈ V
5         v->state  = initial;
6         v->parent = NULL;
7     foreach (s ∈ V)
8         if s->state == initial
9             DFSvisit(s)
```

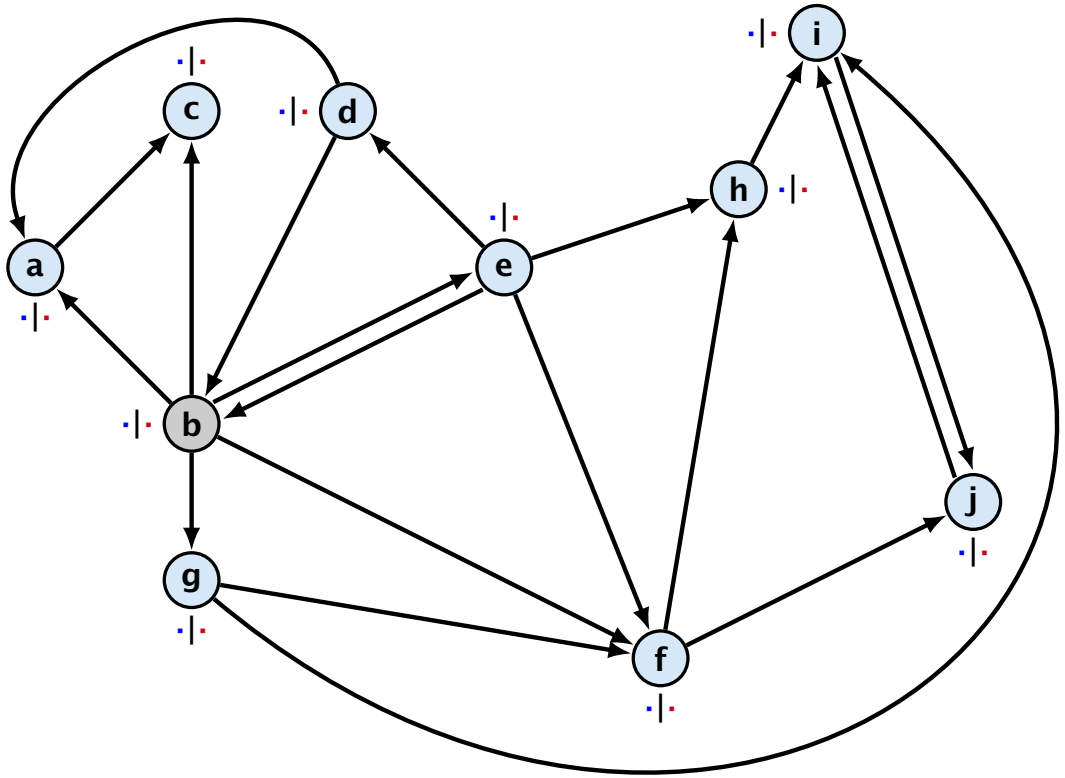
# Tiefensuche – rekursiv

```
1 DFSvisit(v)
2     v->state  = active;
3     v->dfsNum = dfsCount++;
4     foreach x ∈ N[v] // iteriere ueber Nachbarn
5         if (x->state == initial)
6             x->parent = v;
7             DFSvisit(x);
8     v->state  = finished;
9     v->finNum = finCount++;
```

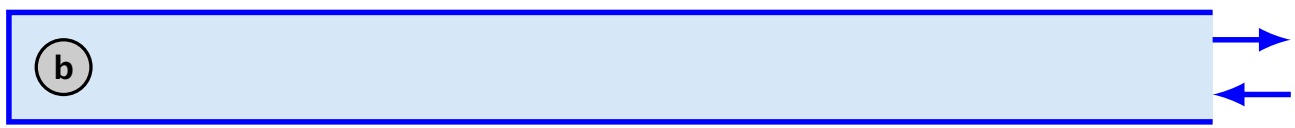
# Tiefensuche – mit Stack

```
1  DFSvisit(s)
2      Stack S; S.push(s);
3      while (!S.empty()) {
4          v = S.top();
5          if (v->state == initial)
6              v->state  = active;
7              v->dfsNum = dfsCount++;
8              foreach x ∈ N(v)
9                  if (x->state == active)        //(v,x) back edge
10                 else if (x->state == finished)//(v,x) cross edge
11                 else // (x->parent,x) forward edge
12                     x->parent = v;
13                     S.push(x);
14         else
15             v->state  = finished;
16             v->finNum = finCount++;
17             S.pop();
18     }
```

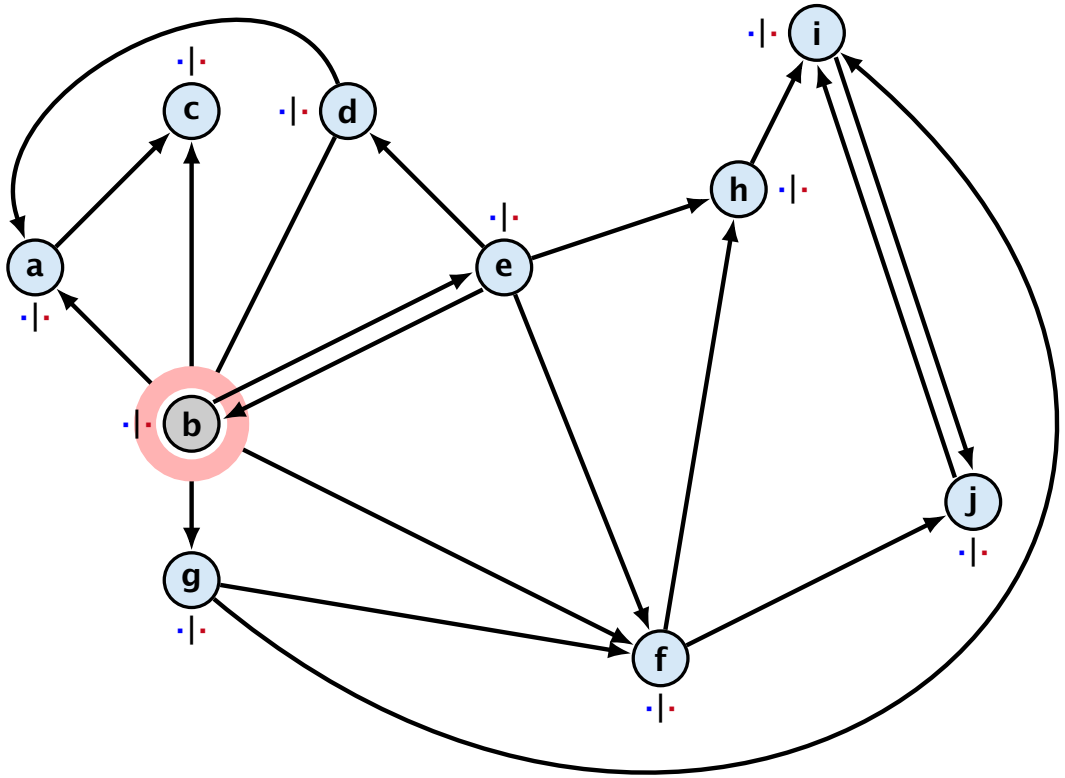
# DFS


i
c
d
h
a
e
b
j
g
f

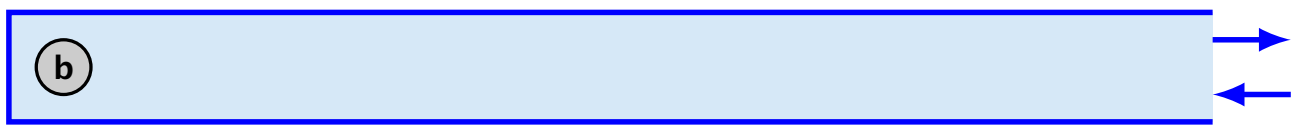
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b

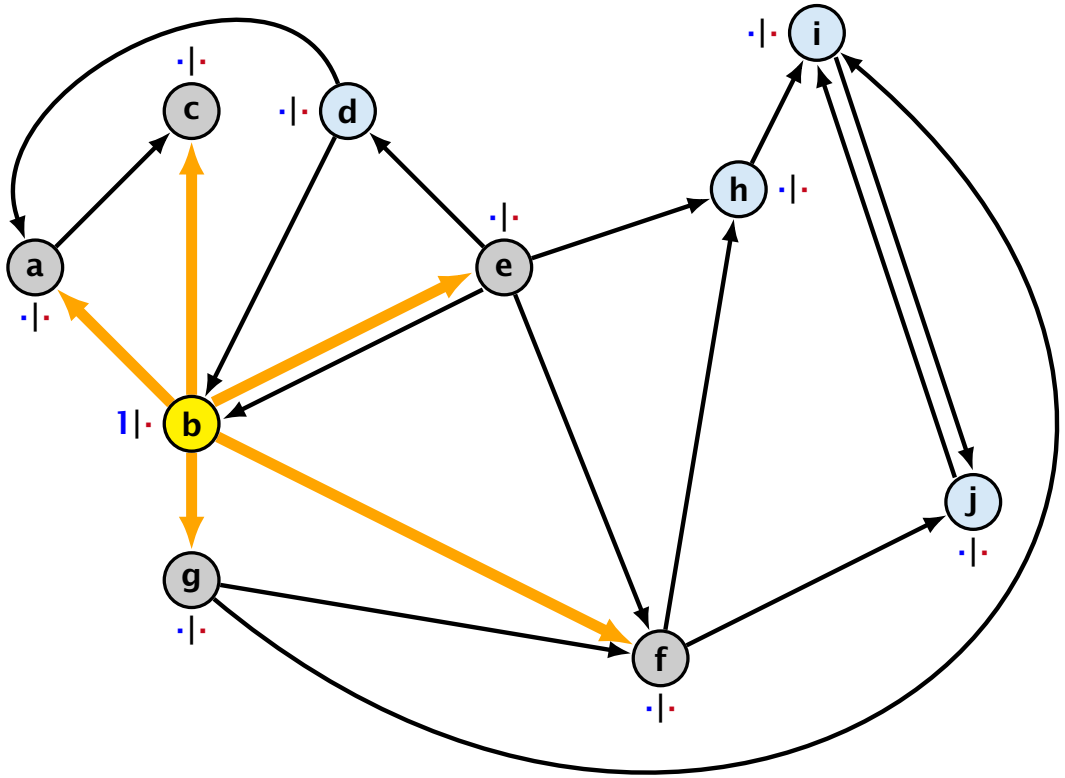
# DFS


i
c
d
h
a
e
b
j
g
f

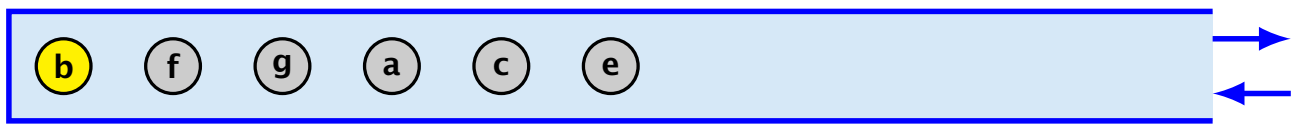
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b

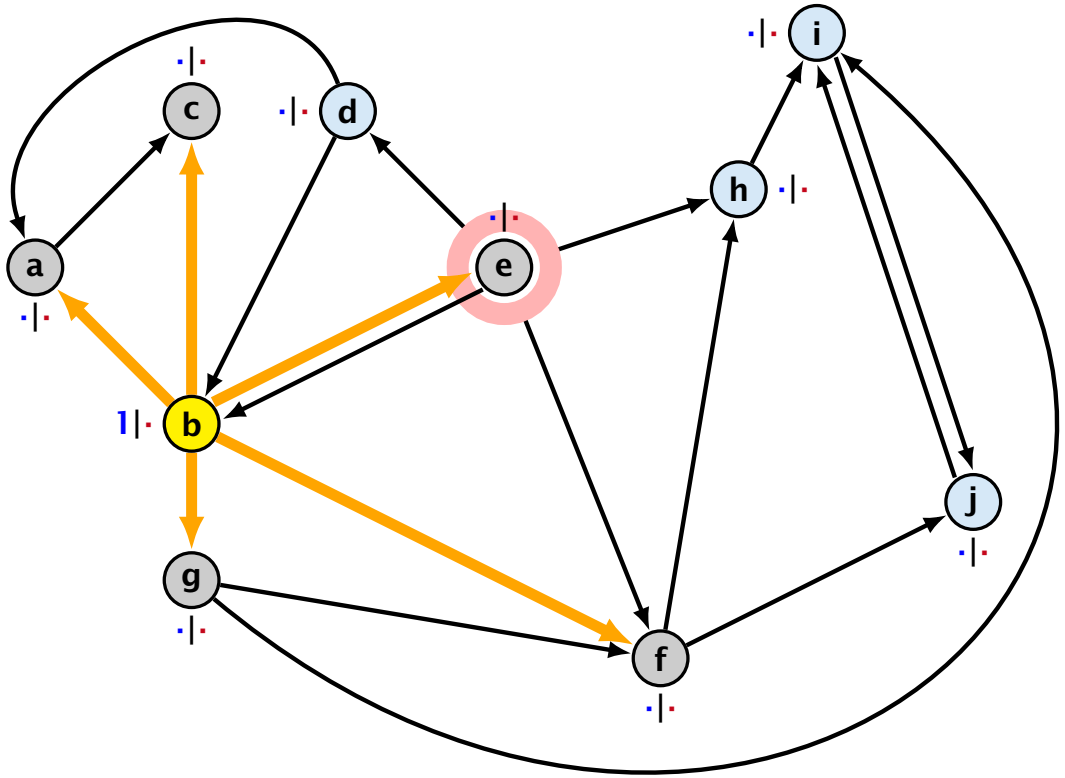
# DFS


i
c
d
h
a
e
1
b
j
g
f

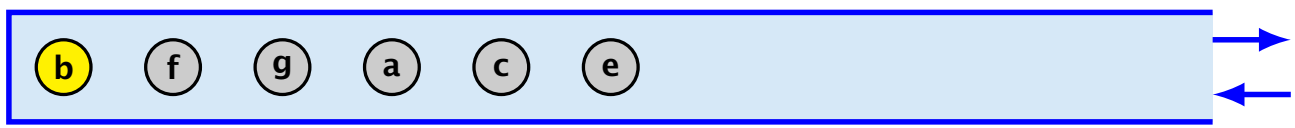
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e

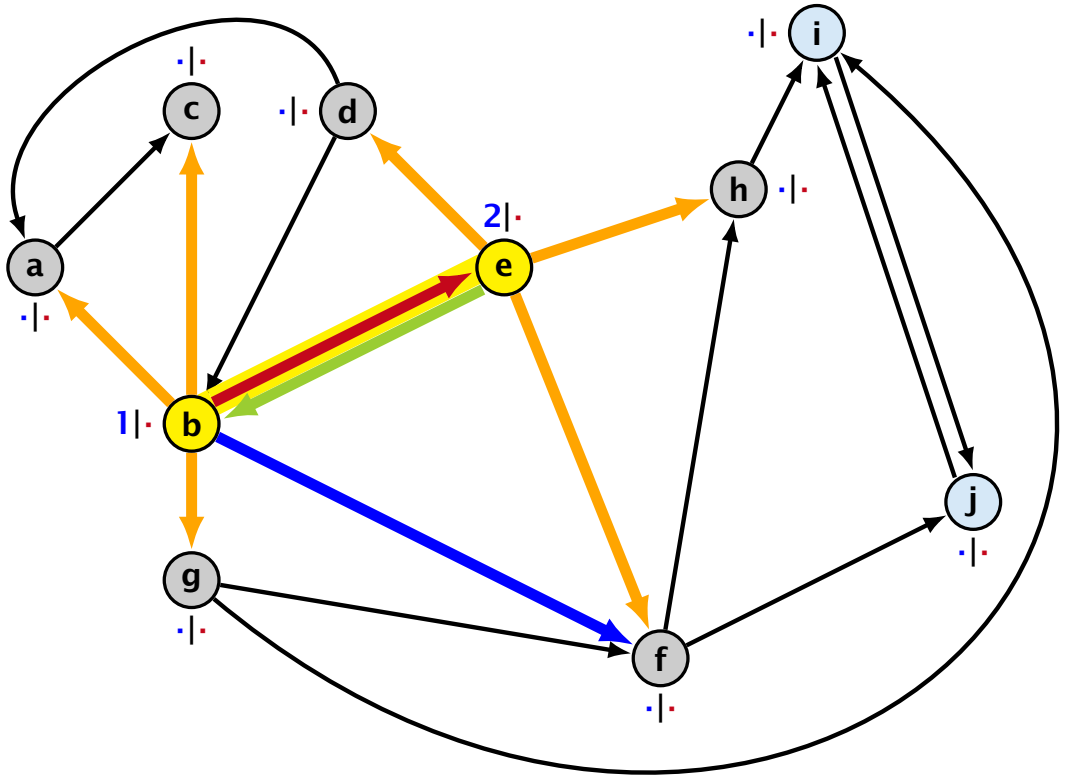
# DFS


i
c
d
h
a
e
1
b
j
g
f

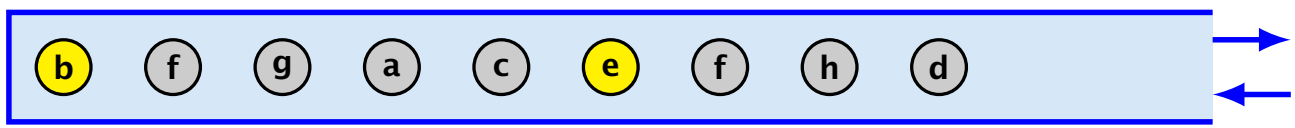
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e

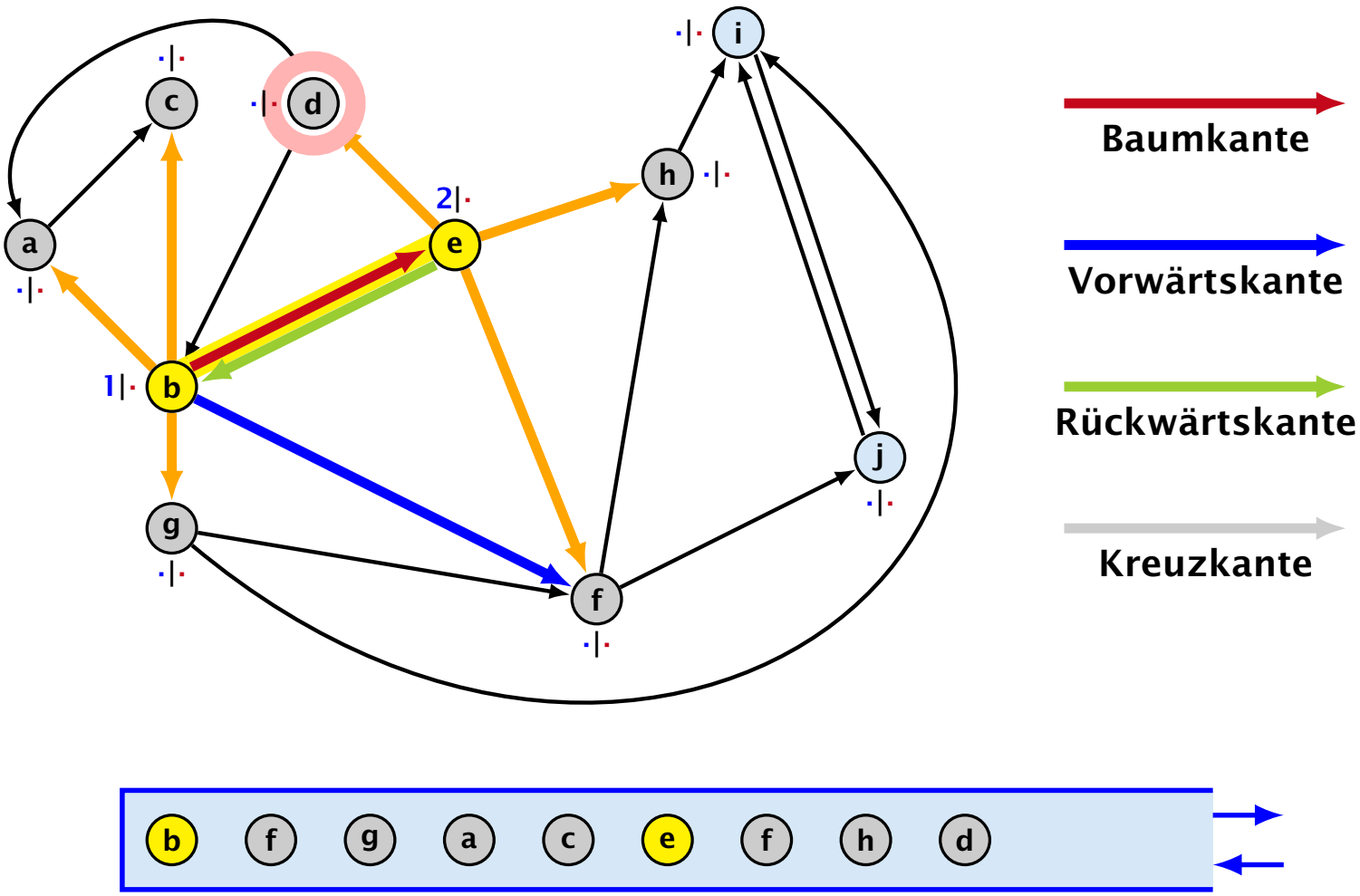
# DFS


i
c
d
h
2
e
a
1
b
j
g
f

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

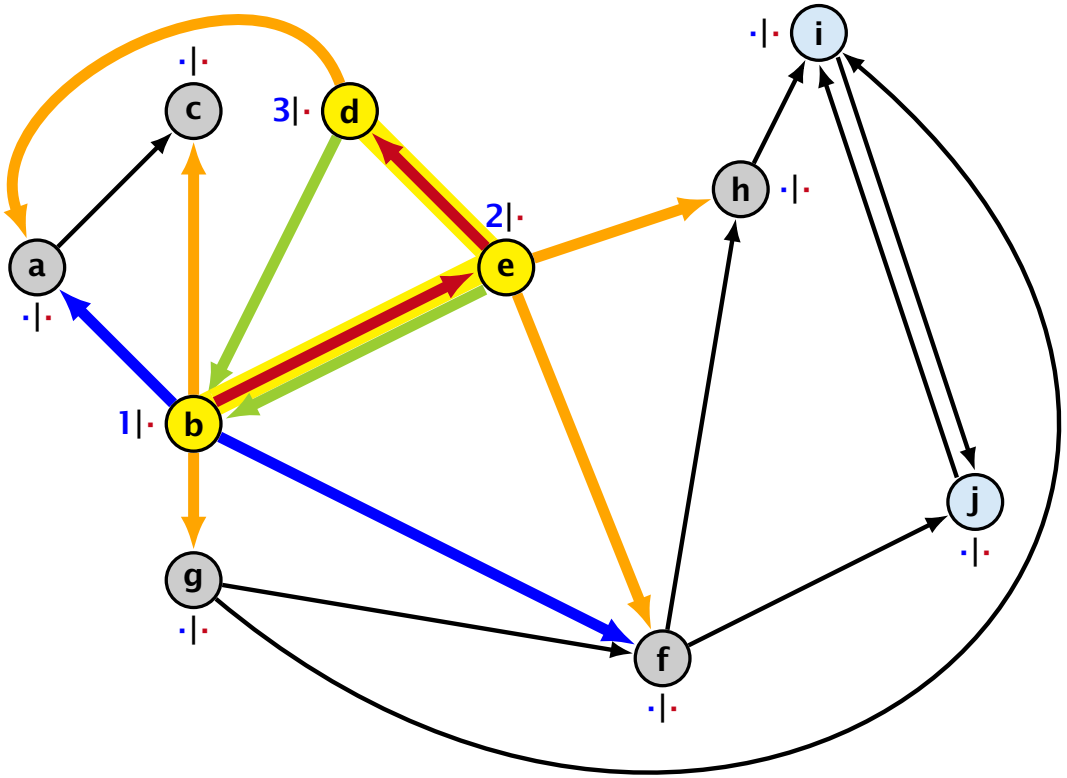

b
f
g
a
c
e
f
h
d

# DFS

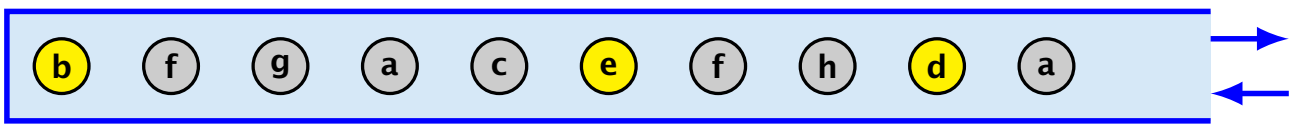


a
b
c
d
e
f
g
h
i
j
1|·
2|·
·|·
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b f g a c e f h d

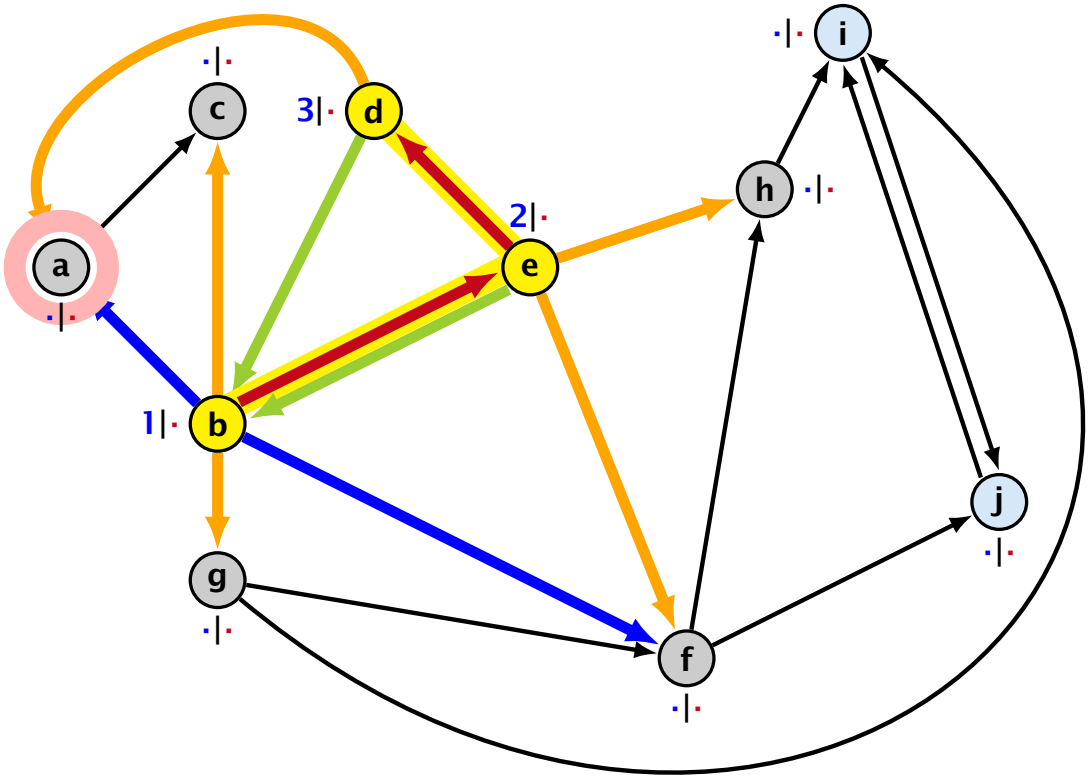
# DFS


·|·
c
3|·
d
·|·
i
h
·|·
2|·
e
a
·|·
1|·
b
j
·|·
g
·|·
f
·|·

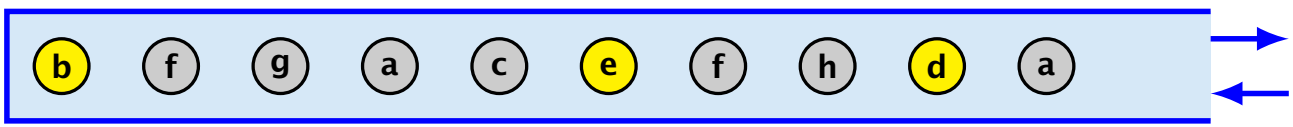
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a

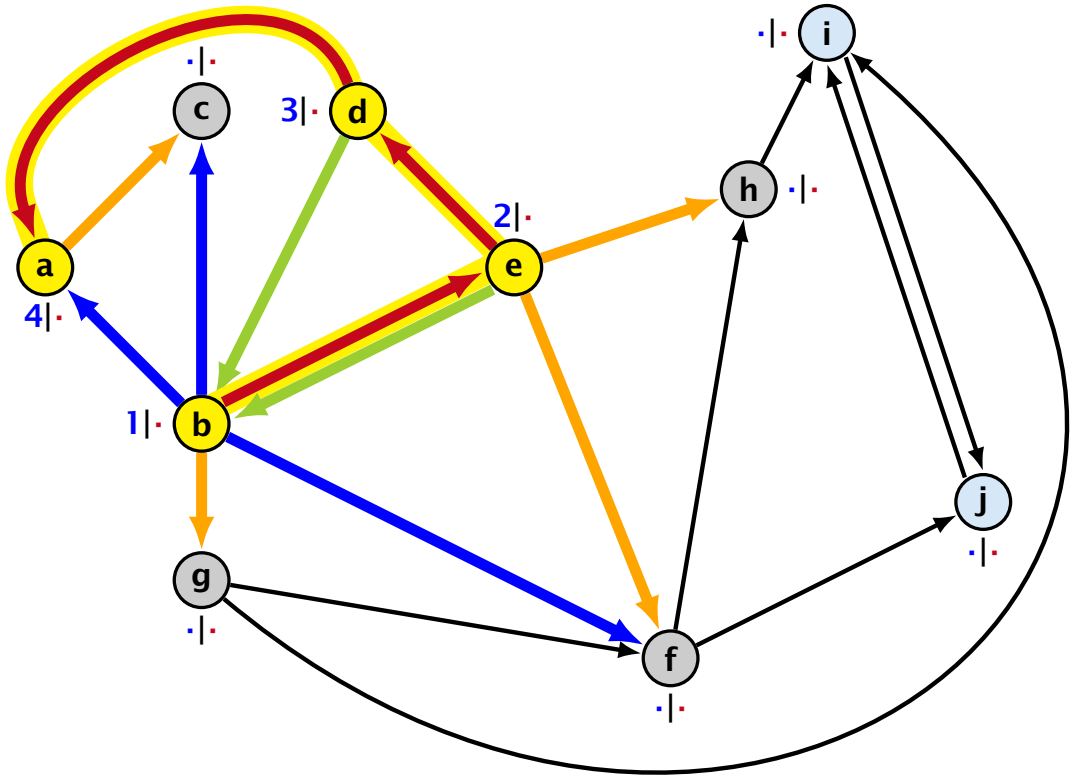
# DFS


·|·
c
3|·
d
·|·
i
h
·|·
2|·
e
a
·|·
1|·
b
j
·|·
g
·|·
f
·|·

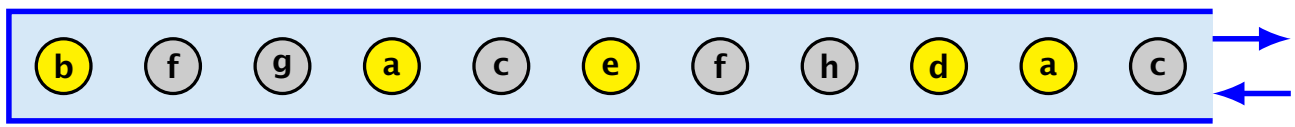
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a

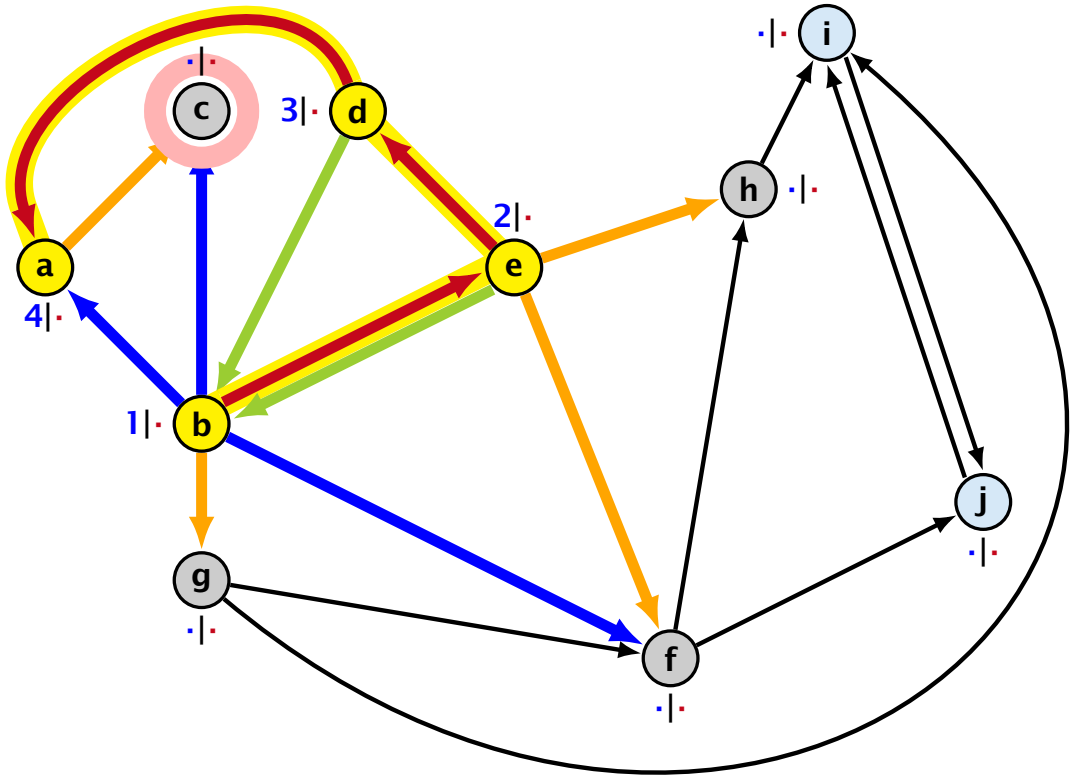
# DFS


·|·
c
3|·
d
·|·
i
h
·|·
2|·
a
e
4|·
1|·
b
j
·|·
g
·|·
f
·|·

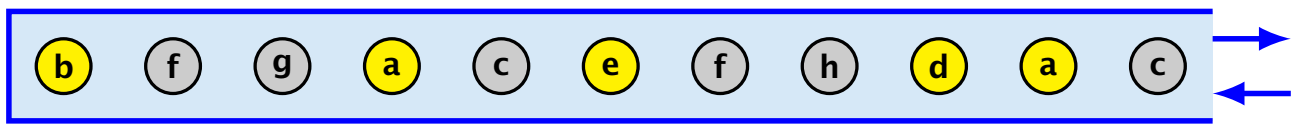
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a
c

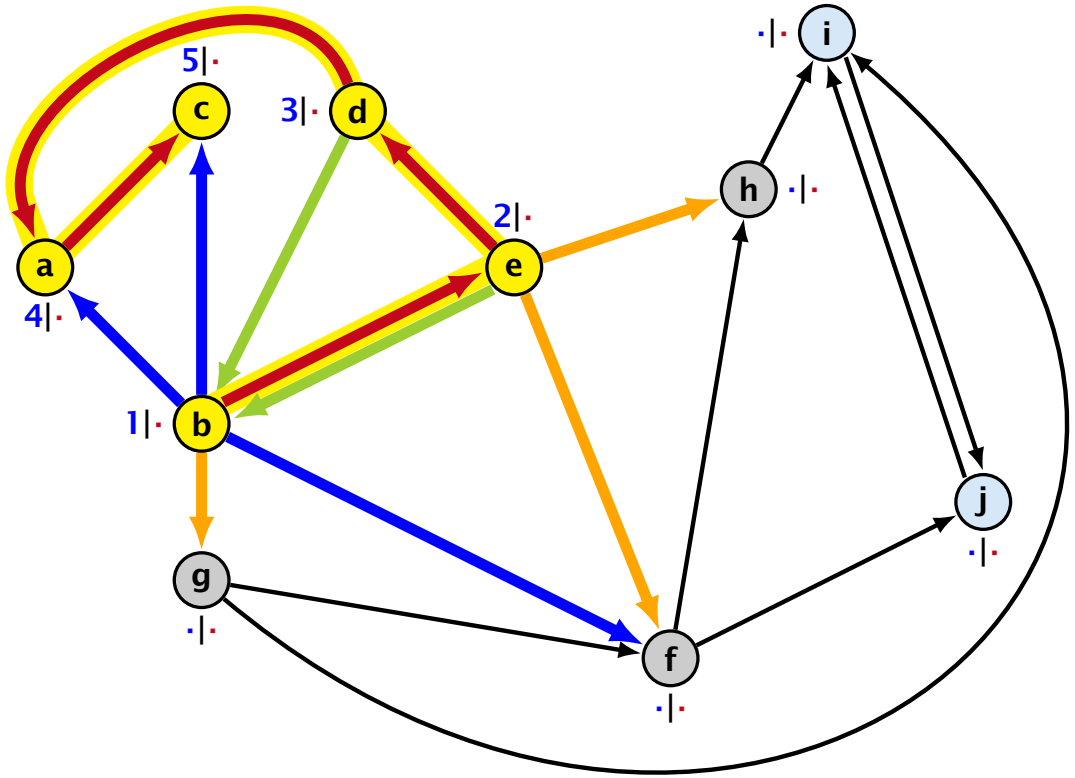
# DFS


·|·
c
3|·
d
·|·
i
h
·|·
2|·
a
e
4|·
1|·
b
j
·|·
g
·|·
f
·|·

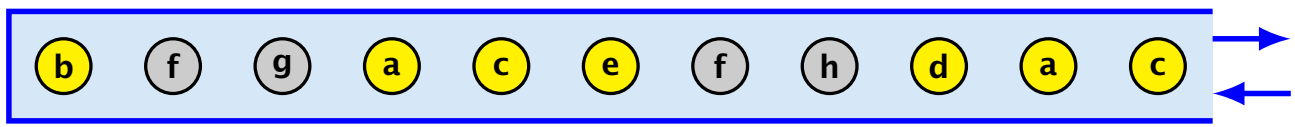
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a
c

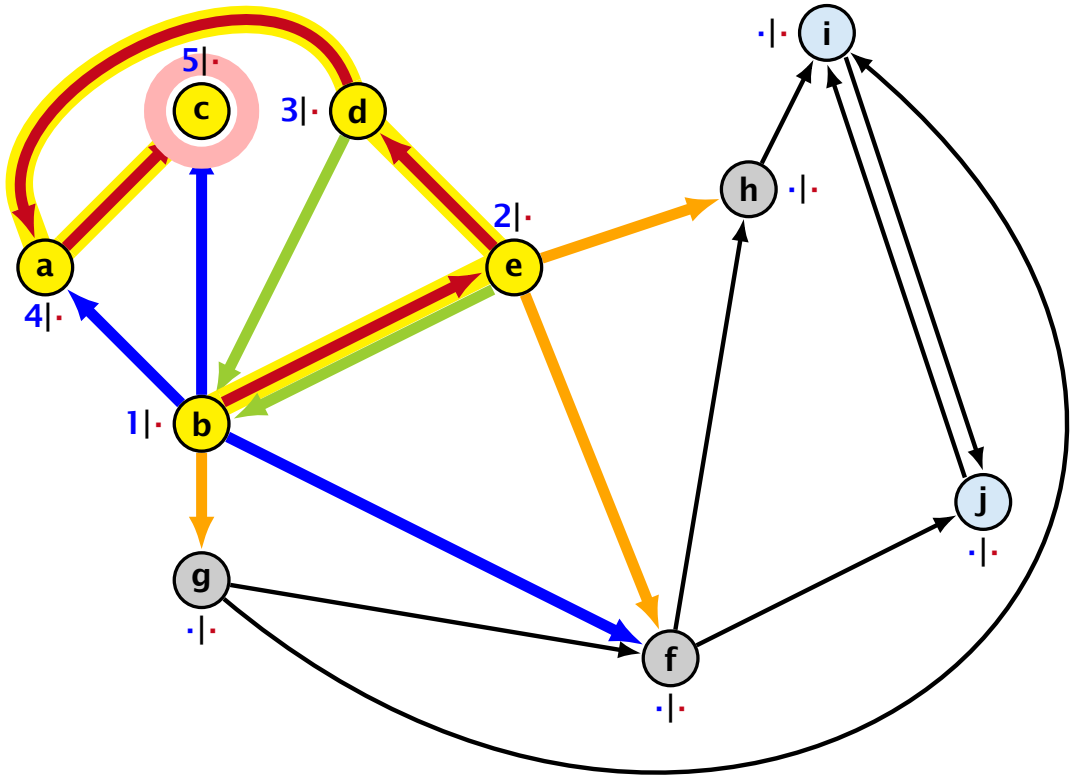
# DFS


5|·
3|·
2|·
4|·
1|·
·|·
a
b
c
d
e
f
g
h
i
j

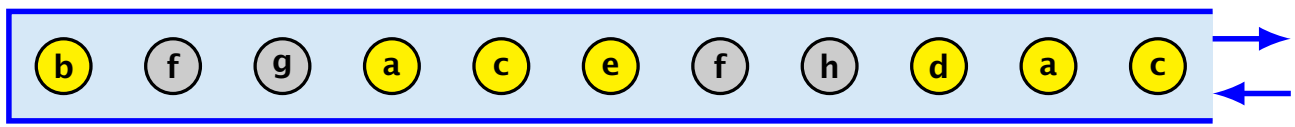
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a
c

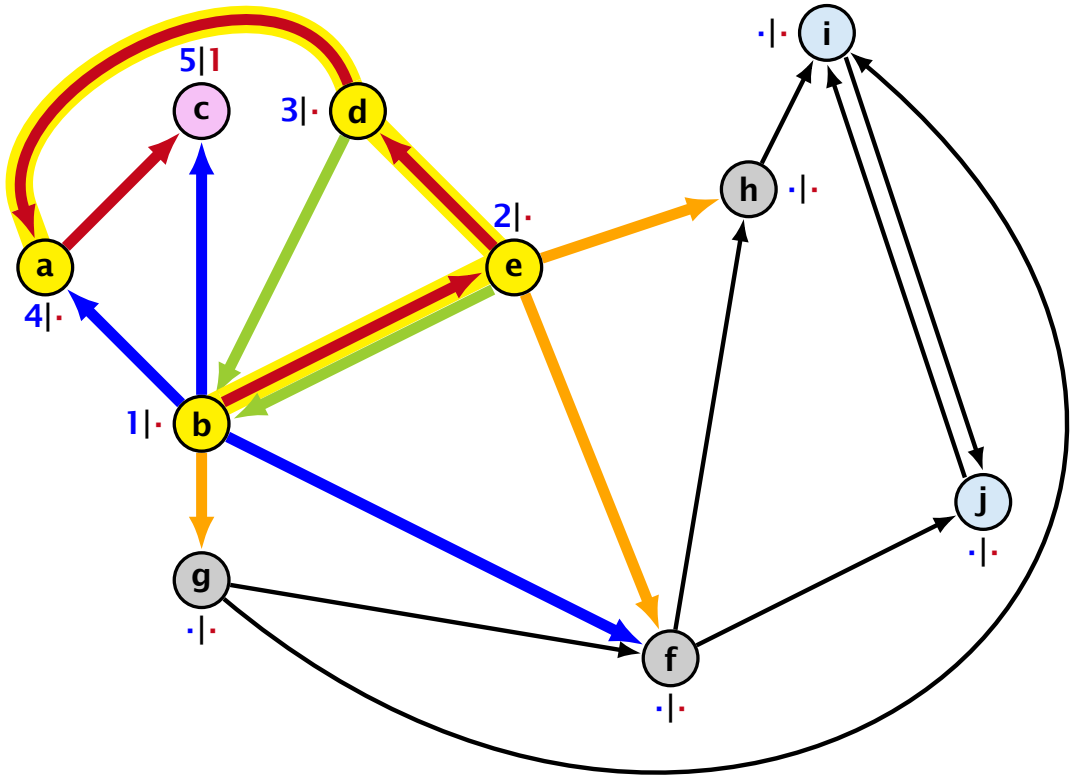
# DFS


5|·
c
3|·
d
·|·
i
h
·|·
2|·
a
e
4|·
1|·
b
j
·|·
g
·|·
f
·|·

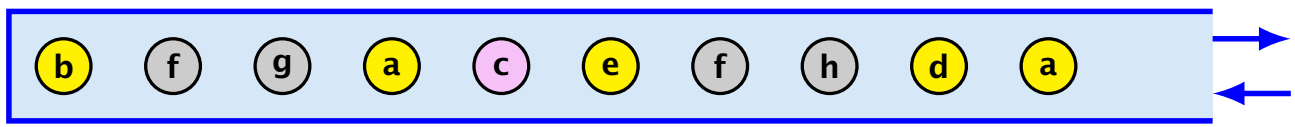
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a
c

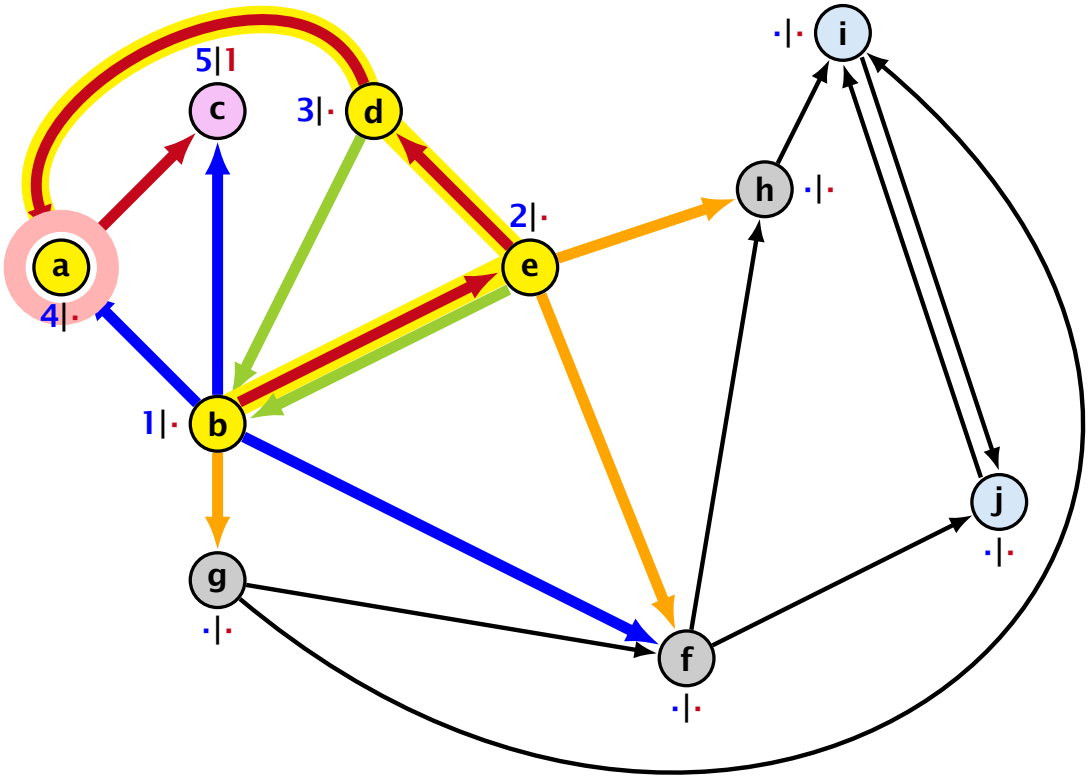
# DFS


5|1
c
3|·
d
·|·
i
h
·|·
2|·
e
a
4|·
1|·
b
g
·|·
f
·|·
j
·|·

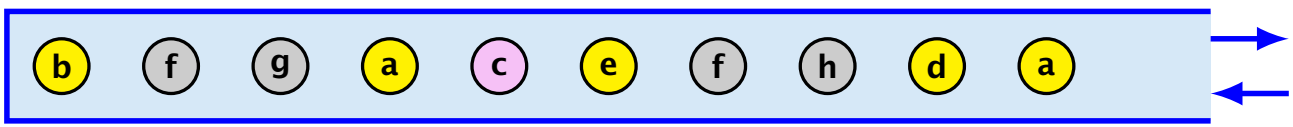
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a

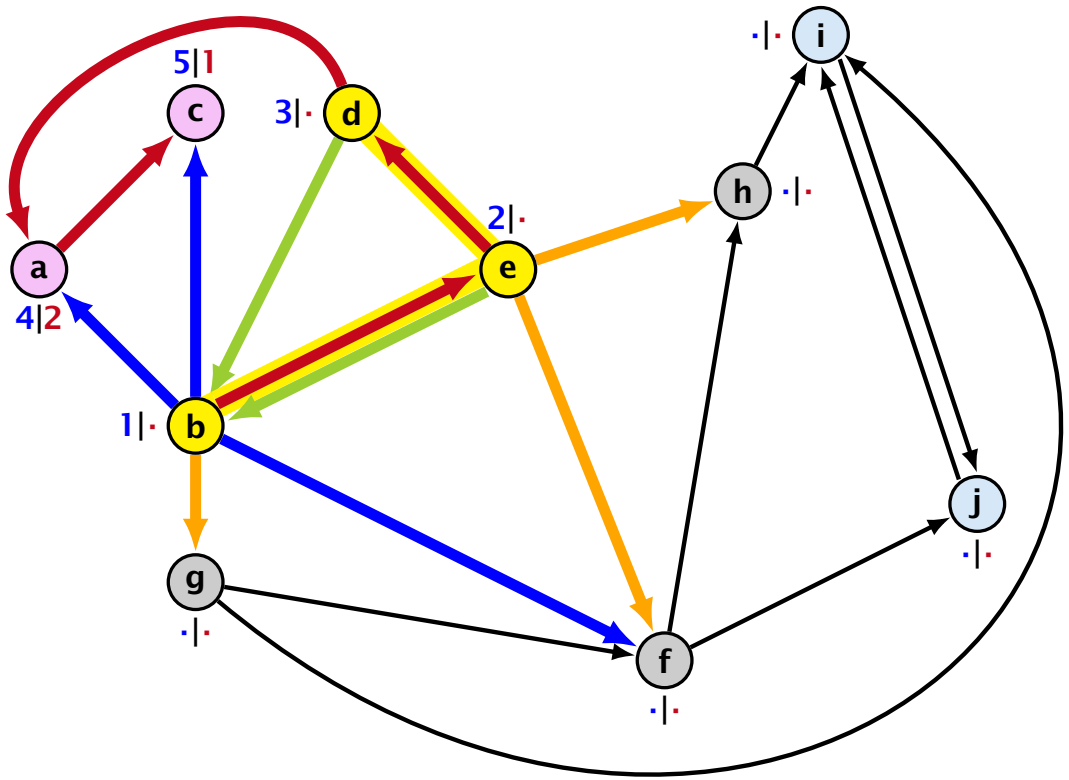
# DFS


5|1
c
3|·
d
·|·
i
2|·
h
·|·
a
e
4|·
1|·
b
g
·|·
j
·|·
f
·|·

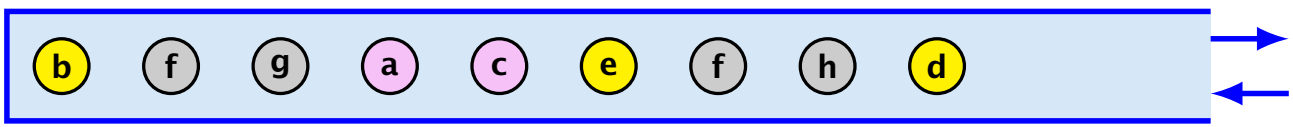
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d
a

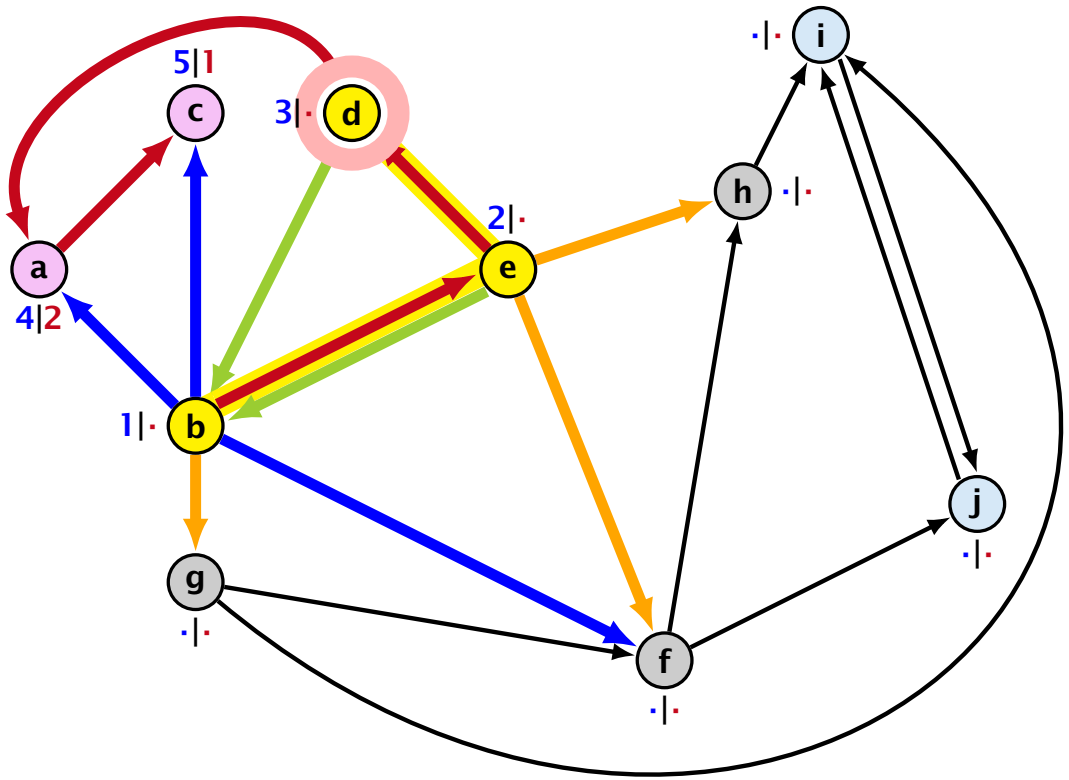
# DFS


5|1
c
3|·
d
·|·
i
h
·|·
2|·
e
a
4|2
1|·
b
j
·|·
g
·|·
f
·|·

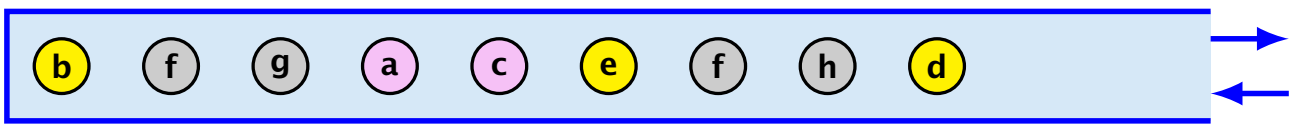
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d

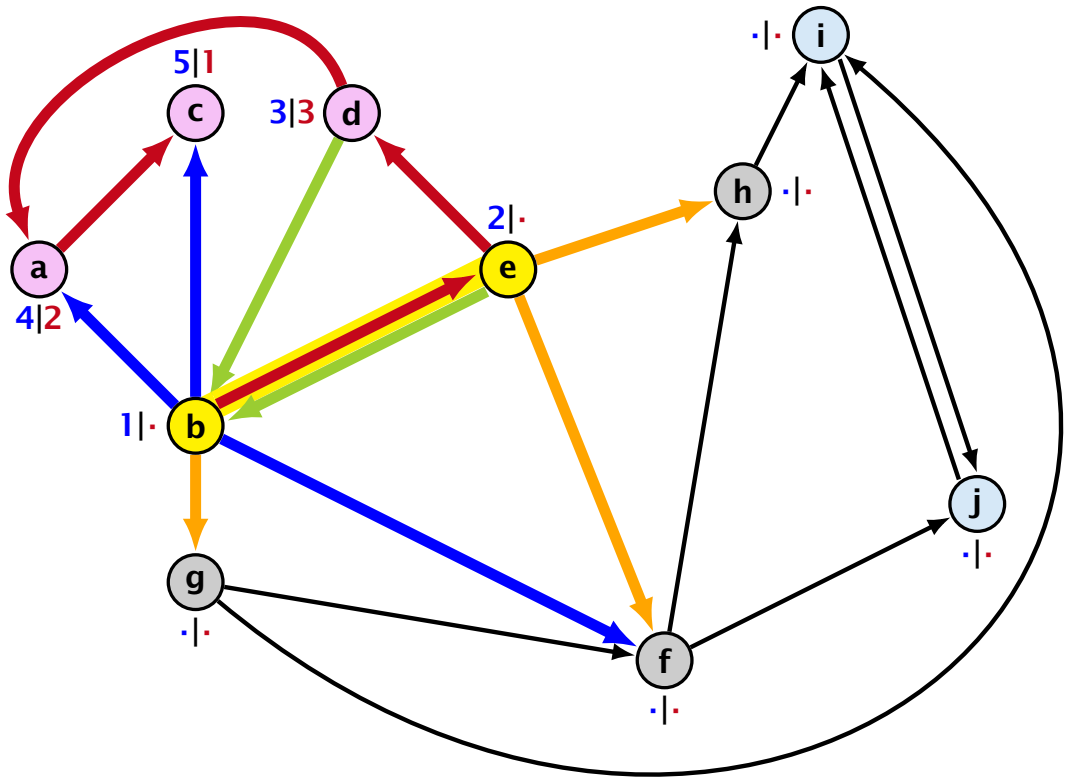
# DFS


5|1
c
3|·
d
i
·|·
h
·|·
2|·
e
a
4|2
1|·
b
j
·|·
g
·|·
f
·|·

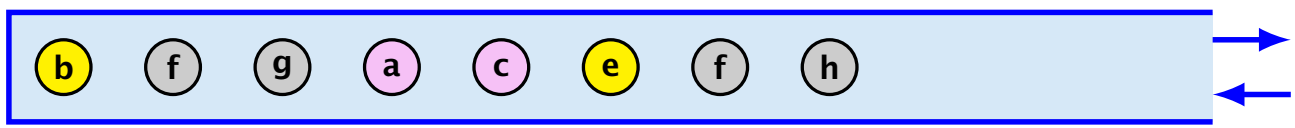
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
d

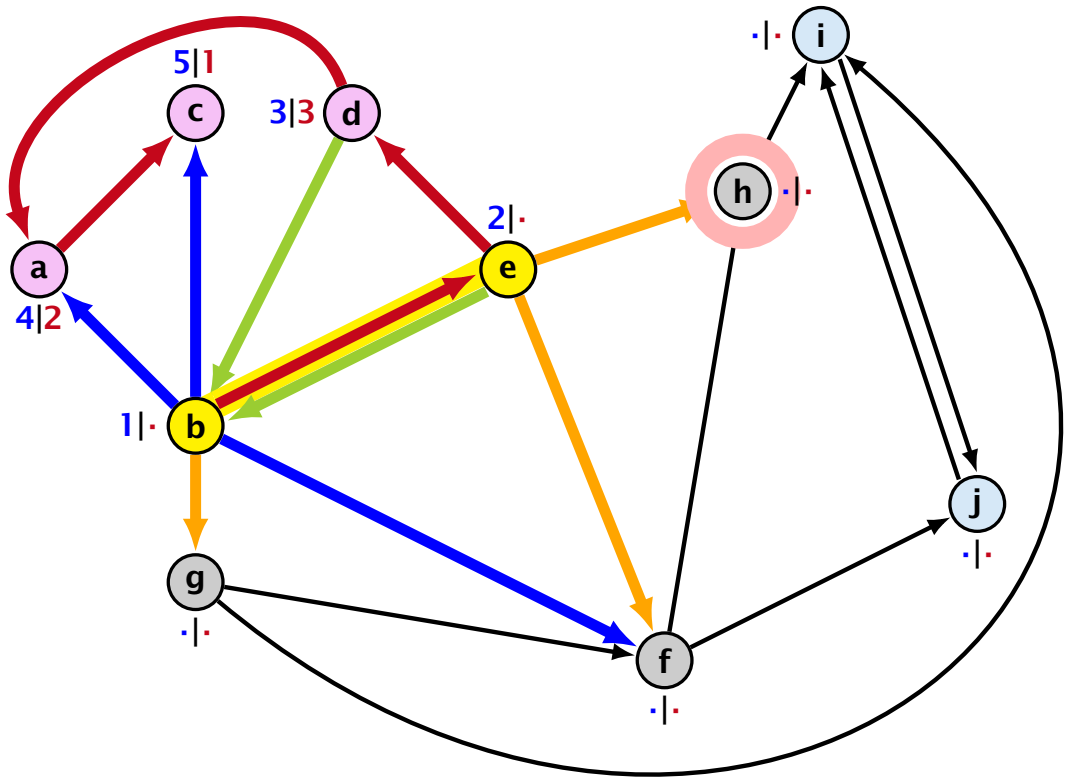
# DFS


5|1
c
3|3
d
i
·|·
h
·|·
2|·
e
a
4|2
1|·
b
j
·|·
g
·|·
f
·|·

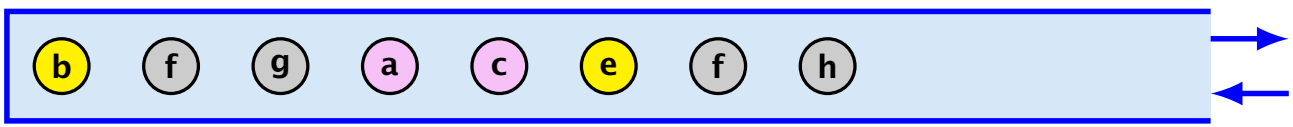
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h

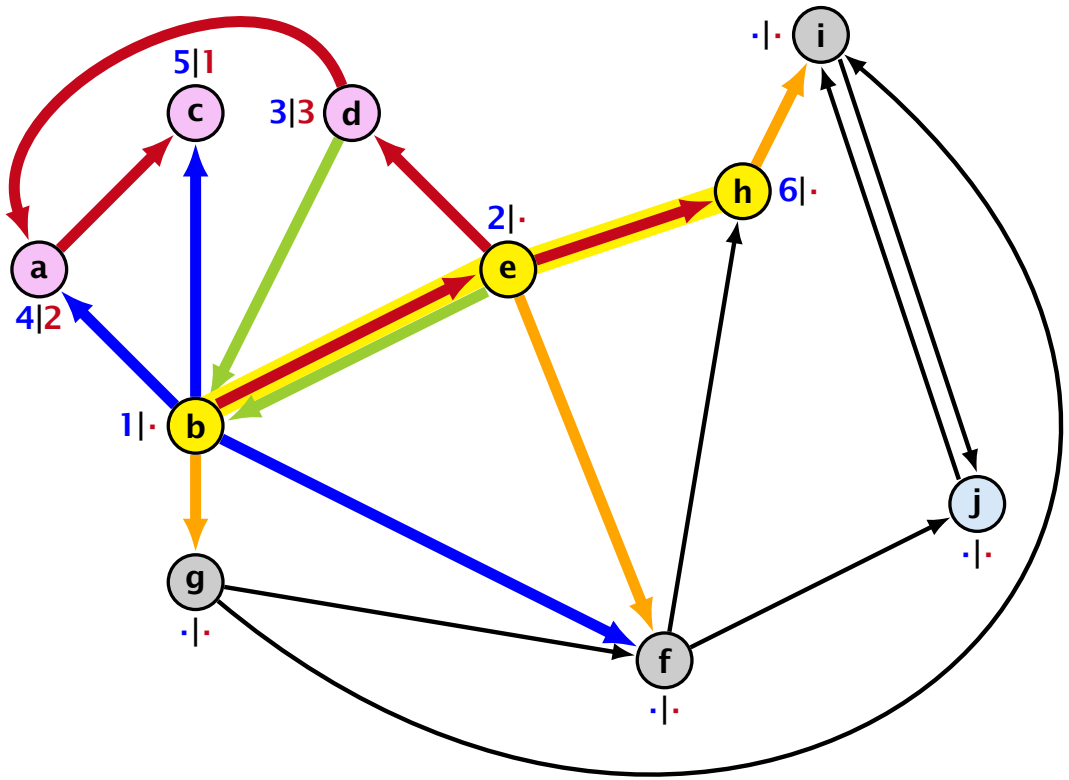
# DFS


5|1
c
3|3
d
i
·|·
h
·|·
2|·
e
a
4|2
1|·
b
j
·|·
g
·|·
f
·|·

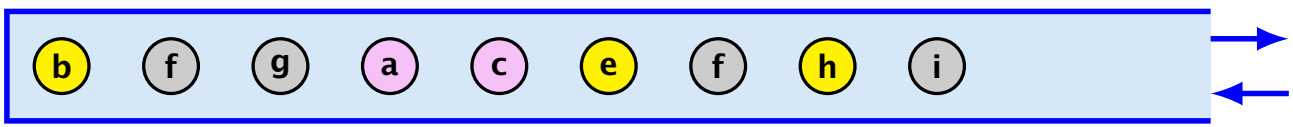
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h

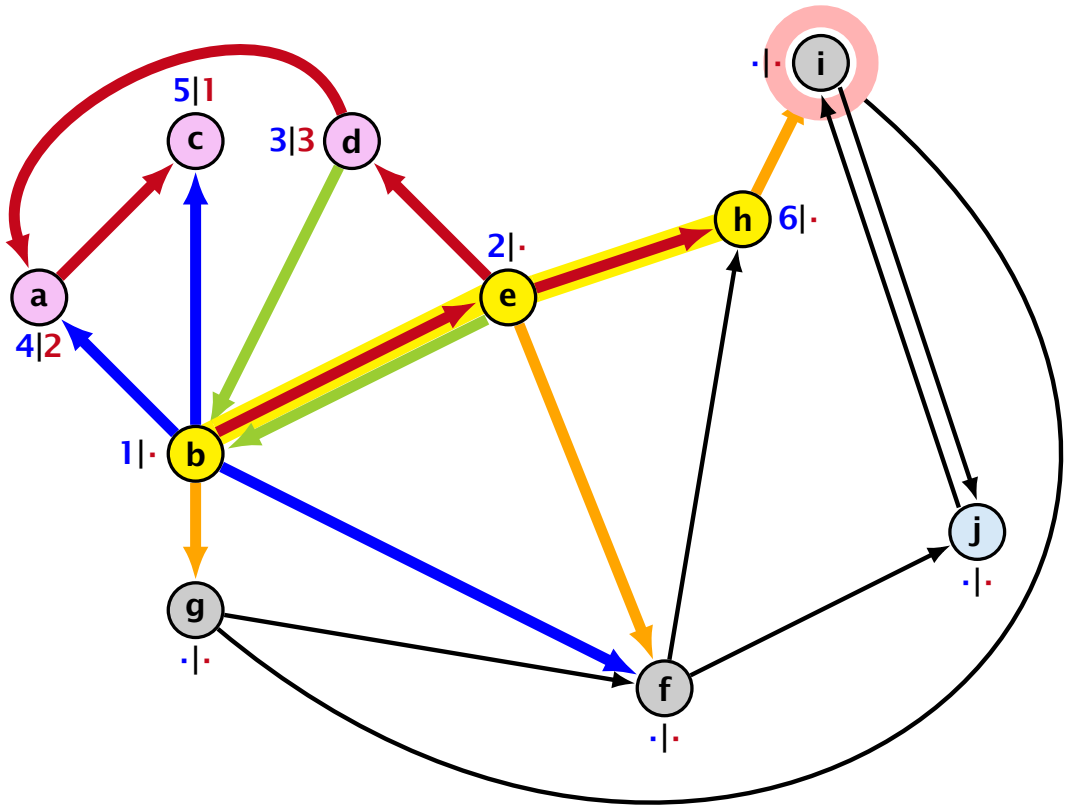
# DFS


5|1
c
3|3
d
·|·
i
a
4|2
2|·
e
h
6|·
1|·
b
j
·|·
g
·|·
f
·|·

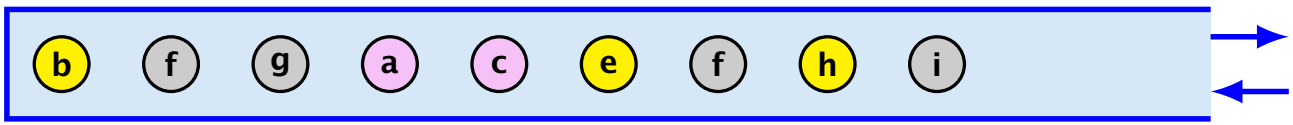
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i

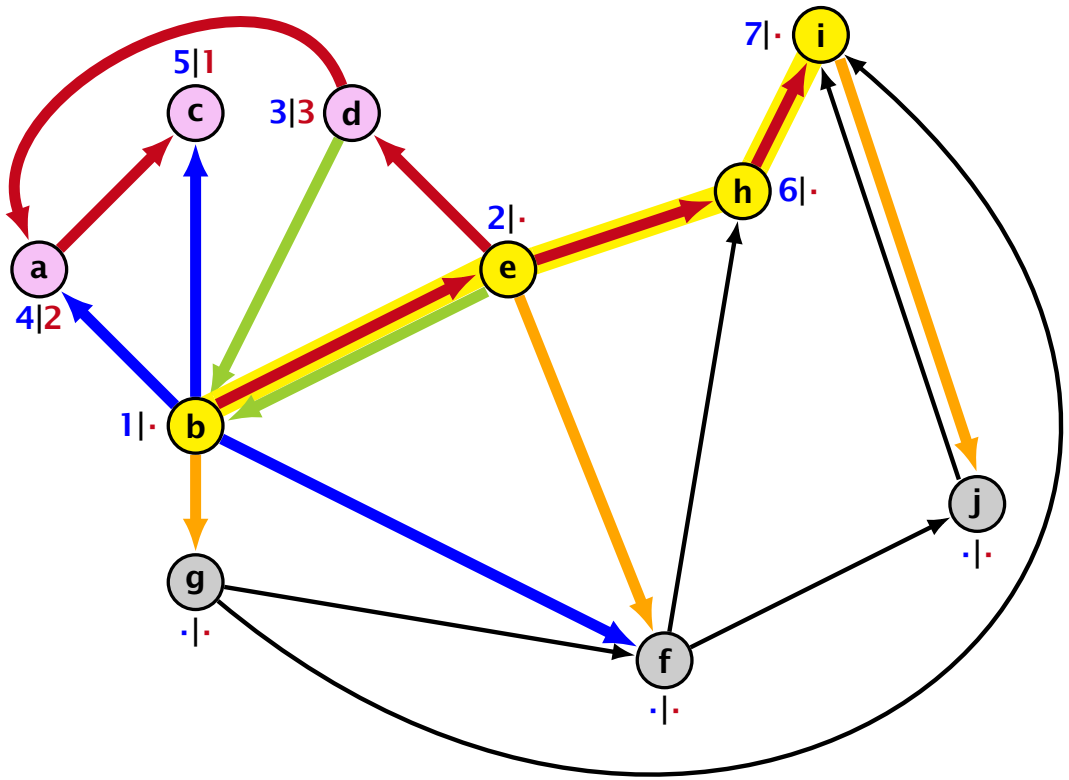
# DFS


5|1
c
3|3
d
i
·|·
h
6|·
2|·
e
a
4|2
1|·
b
j
·|·
g
·|·
f
·|·

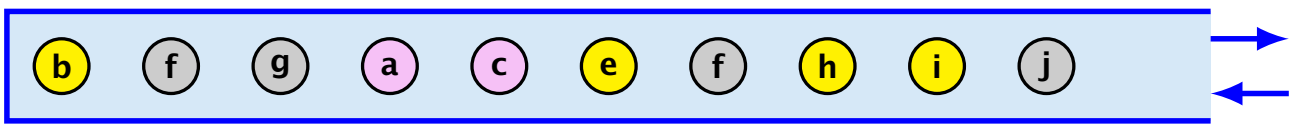
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i

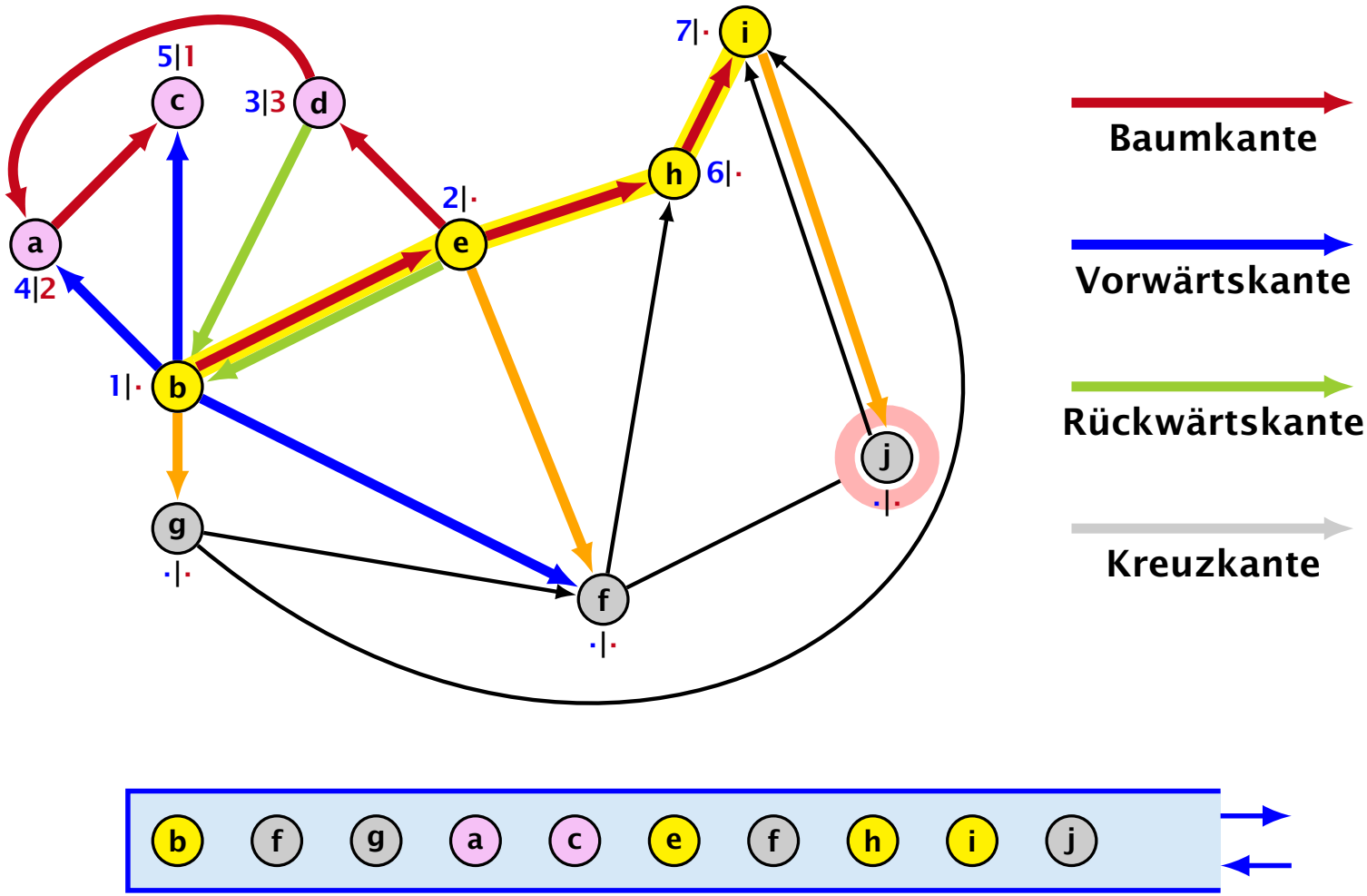
# DFS


5|1
c
3|3
d
7|·
i
a
4|2
2|·
e
h
6|·
1|·
b
g
·|·
f
·|·
j
·|·

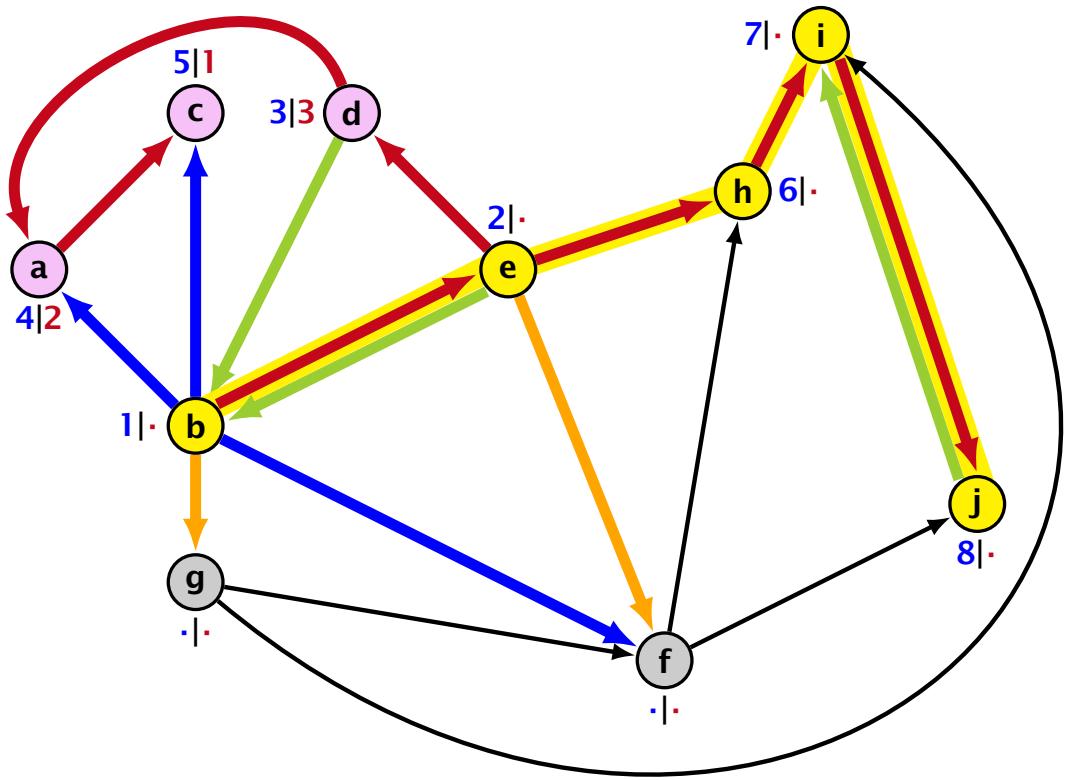
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i
j

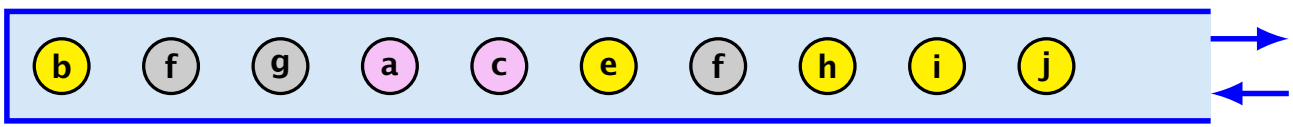
# DFS


5|1
c
3|3
d
7|·
i
h
6|·
2|·
e
a
4|2
1|·
b
g
·|·
f
·|·
j
·|·
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b
f
g
a
c
e
f
h
i
j

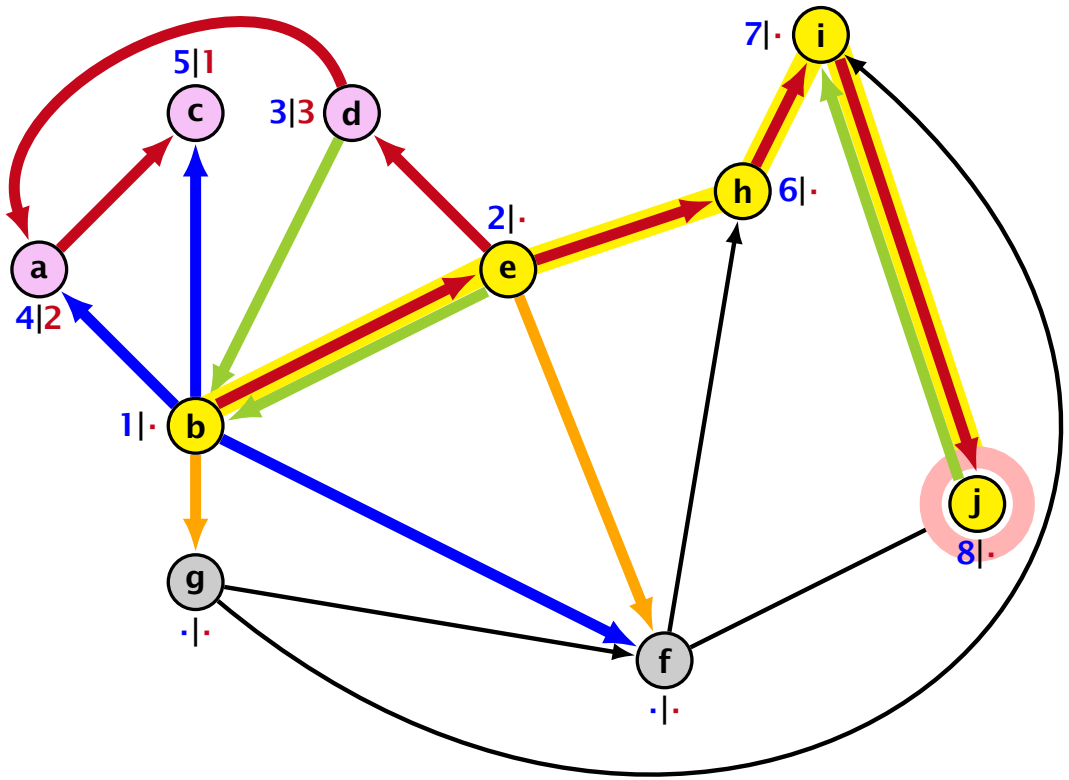
# DFS


5|1
c
3|3
d
7|·
i
h
6|·
2|·
e
a
4|2
1|·
b
j
8|·
g
·|·
f
·|·

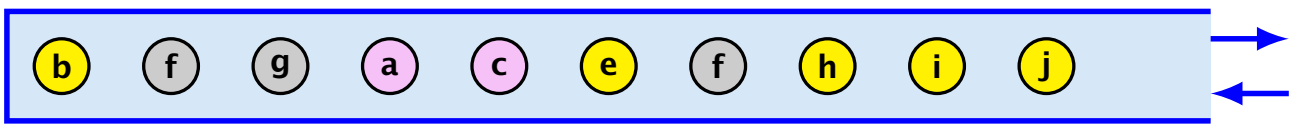
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i
j

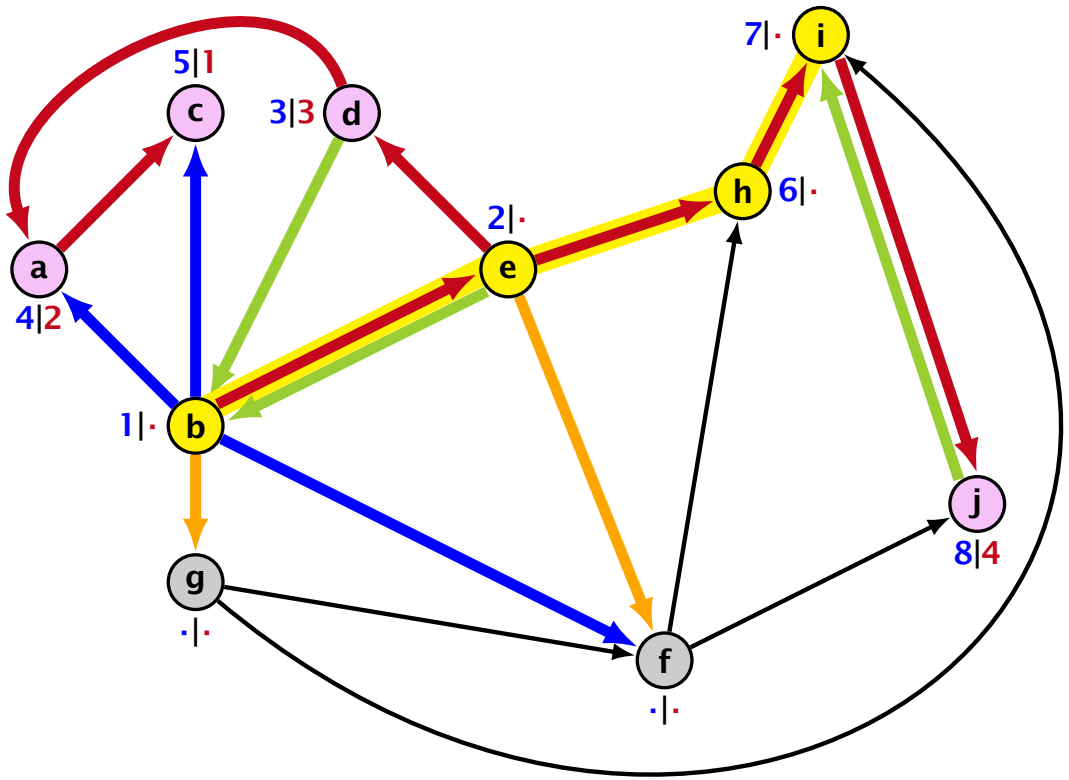
# DFS


5|1
c
3|3
d
7|·
i
a
4|2
2|·
e
h
6|·
1|·
b
j
8|·
g
·|·
f
·|·

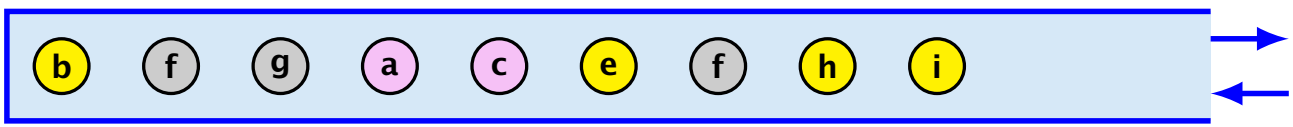
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i
j

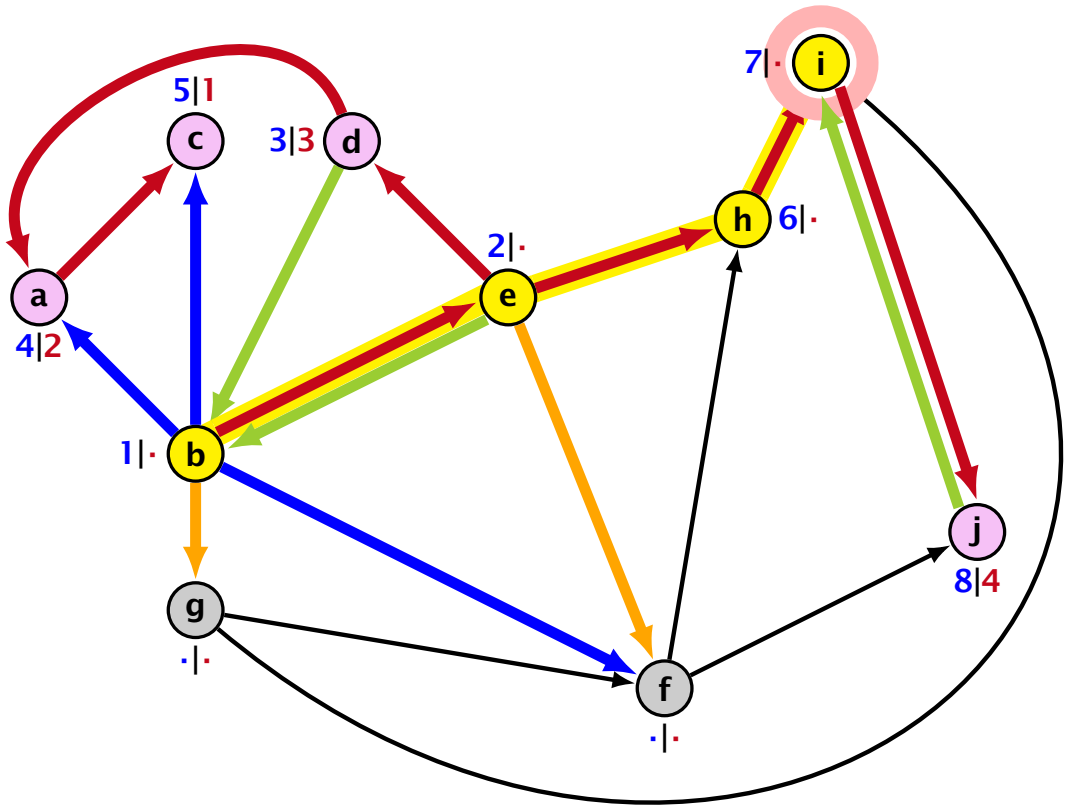
# DFS


5|1
c
3|3
d
7|·
i
a
4|2
2|·
e
h
6|·
1|·
b
j
8|4
g
·|·
f
·|·

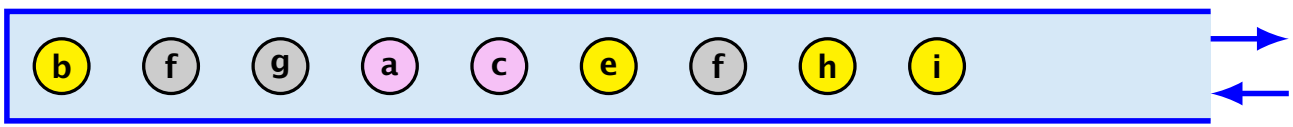
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i

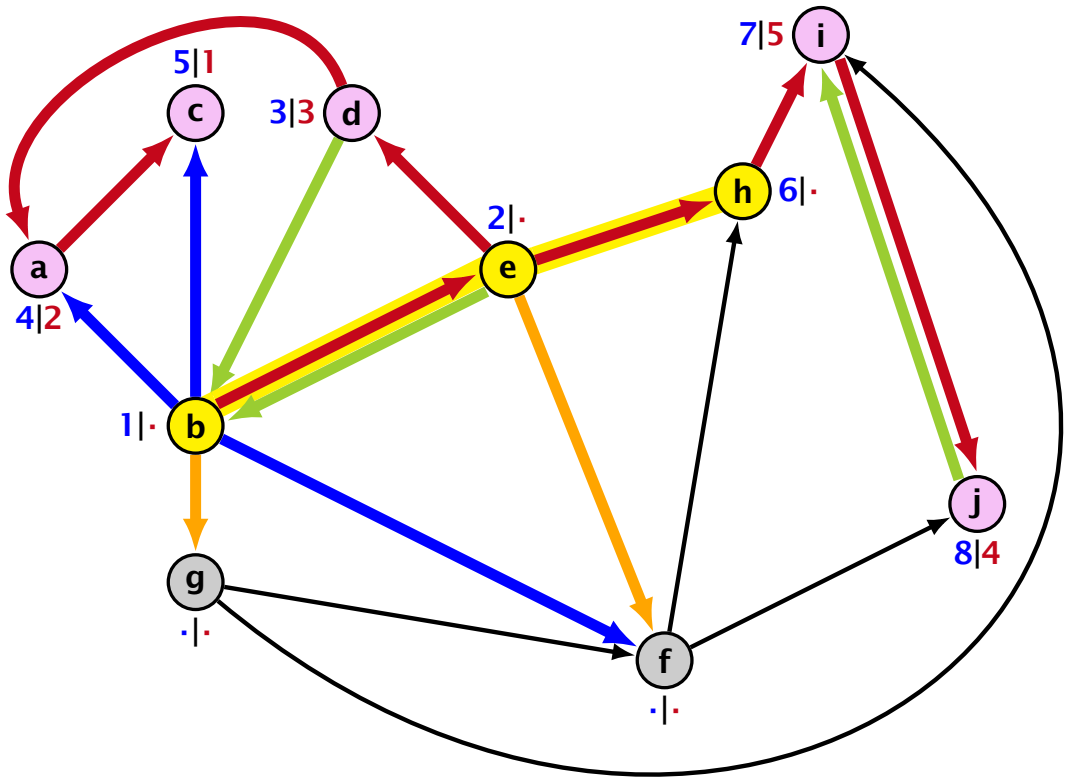
# DFS


5|1
c
3|3
d
7|·
i
h
6|·
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
·|·

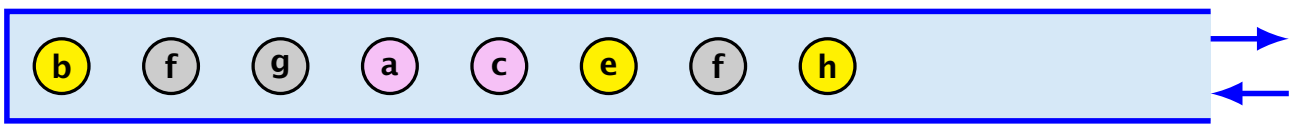
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h
i

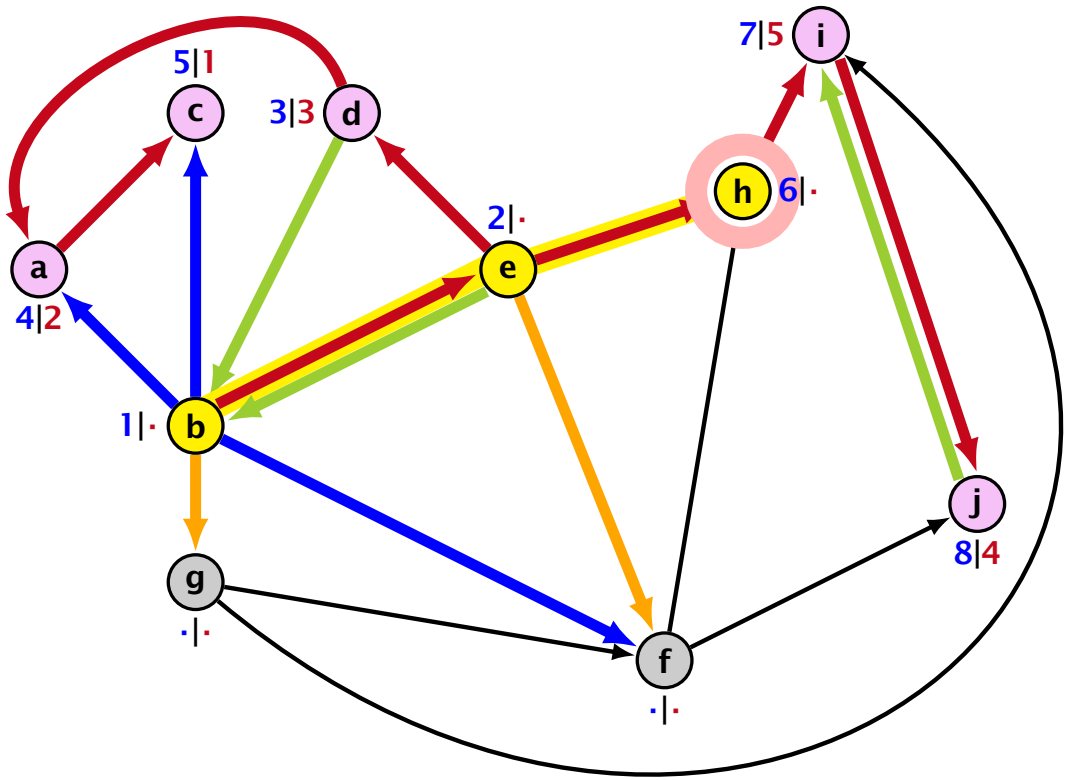
# DFS


5|1
c
3|3
d
7|5
i
a
4|2
2|·
e
h
6|·
1|·
b
j
8|4
g
·|·
f
·|·

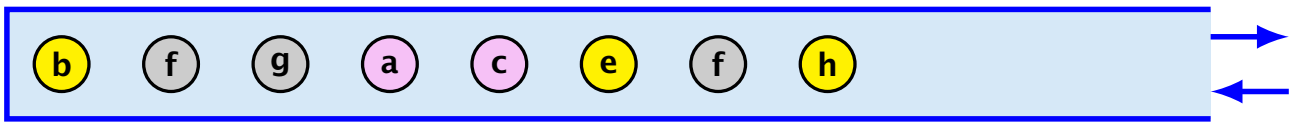
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h

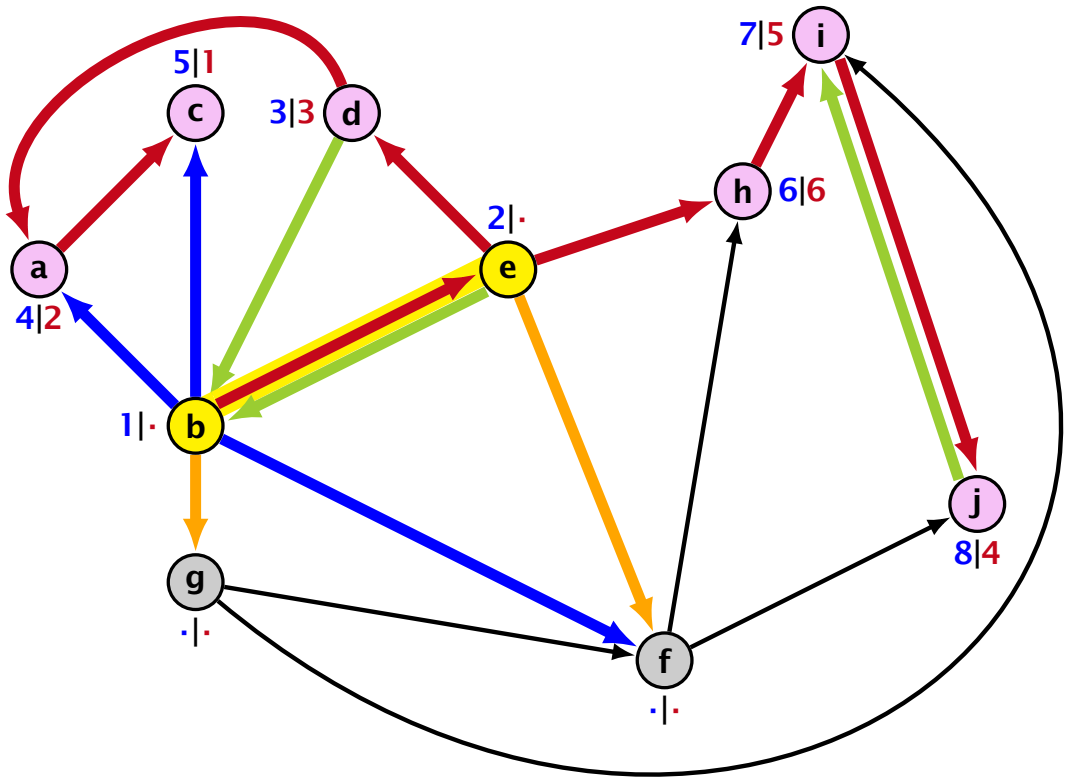
# DFS


5|1
c
3|3
d
7|5
i
h
6|·
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
·|·

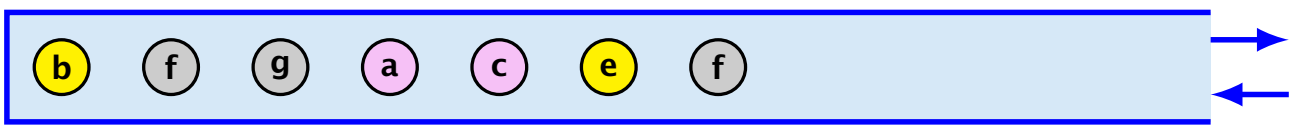
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f
h

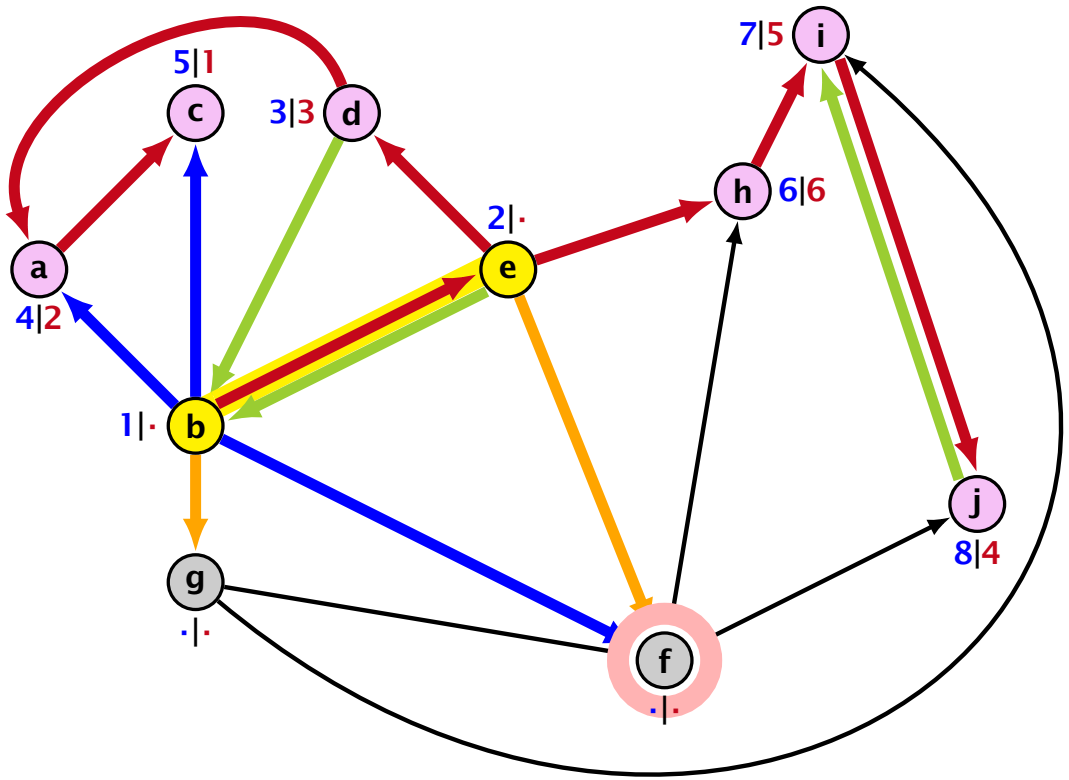
# DFS


7|5
i
5|1
c
3|3
d
h
6|6
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
·|·

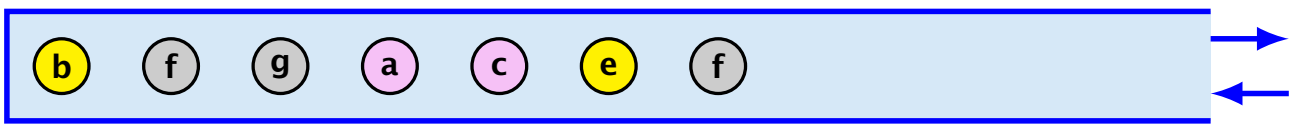
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f

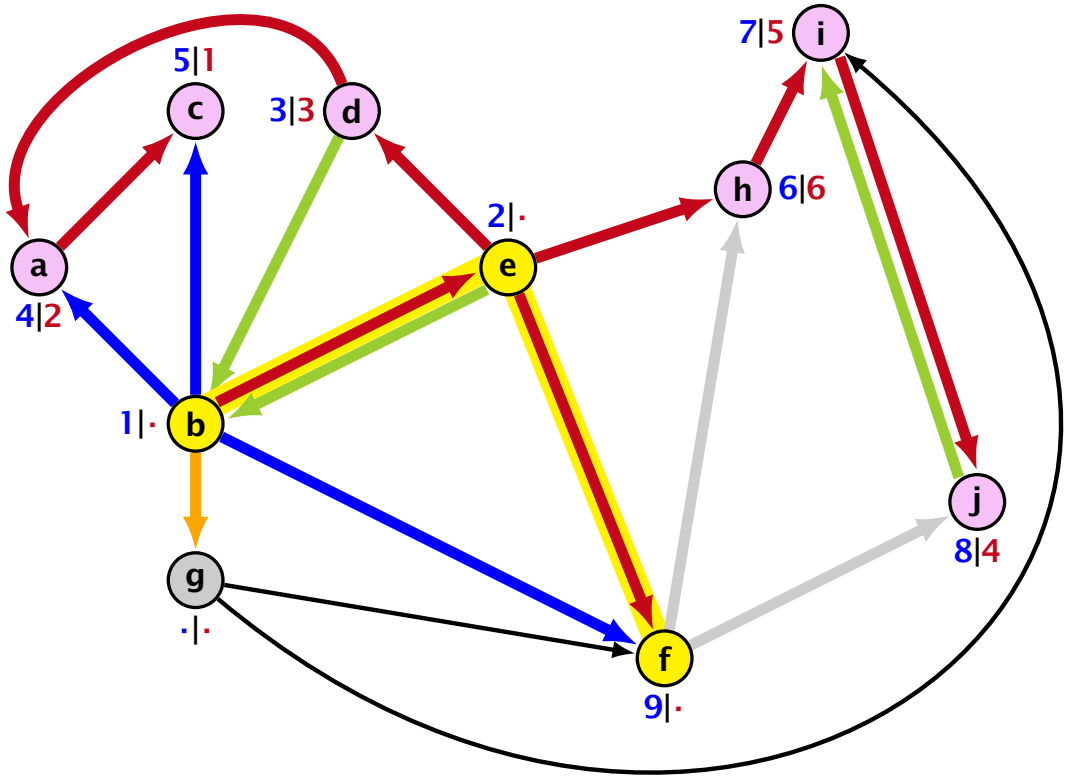
# DFS


7|5
i
5|1
c
3|3
d
h
6|6
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
·|·

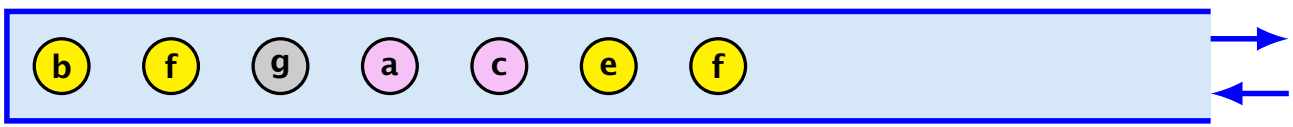
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f

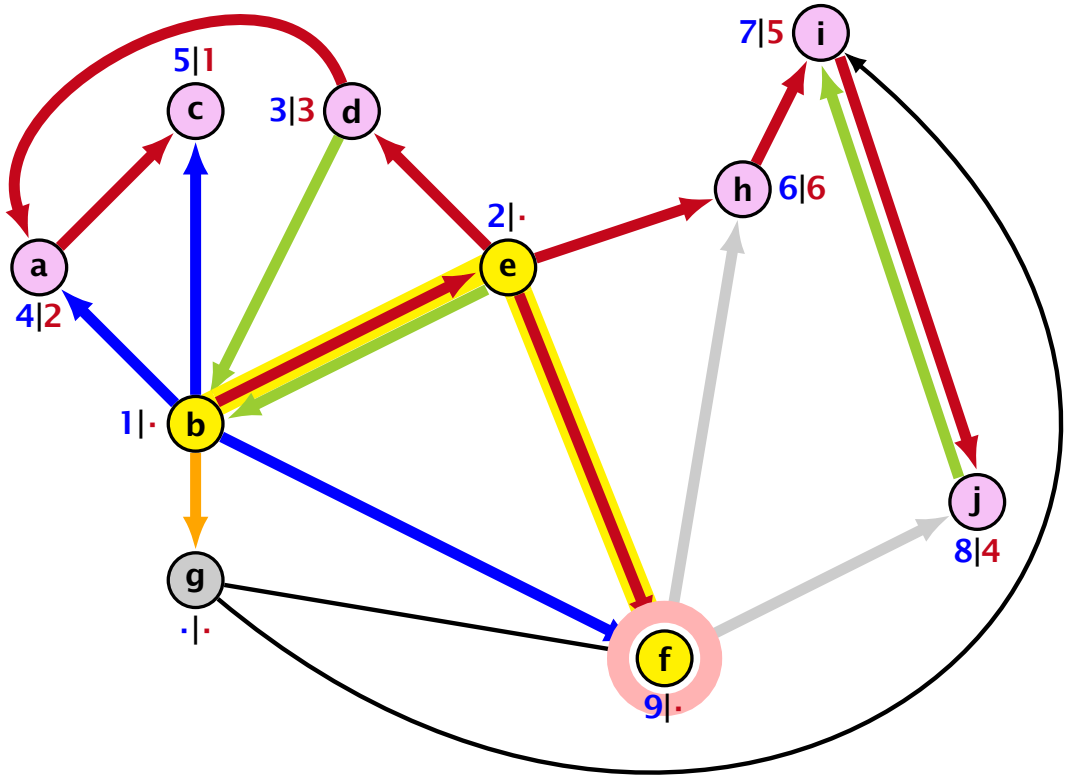
# DFS


7|5
i
5|1
c
3|3
d
h
6|6
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
9|·

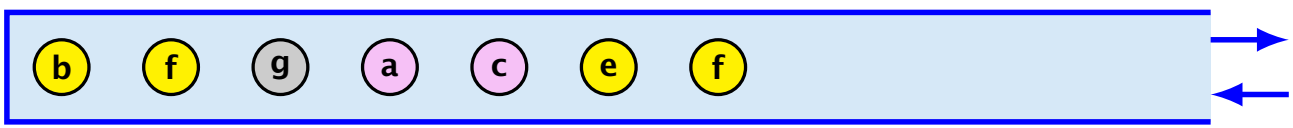
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f

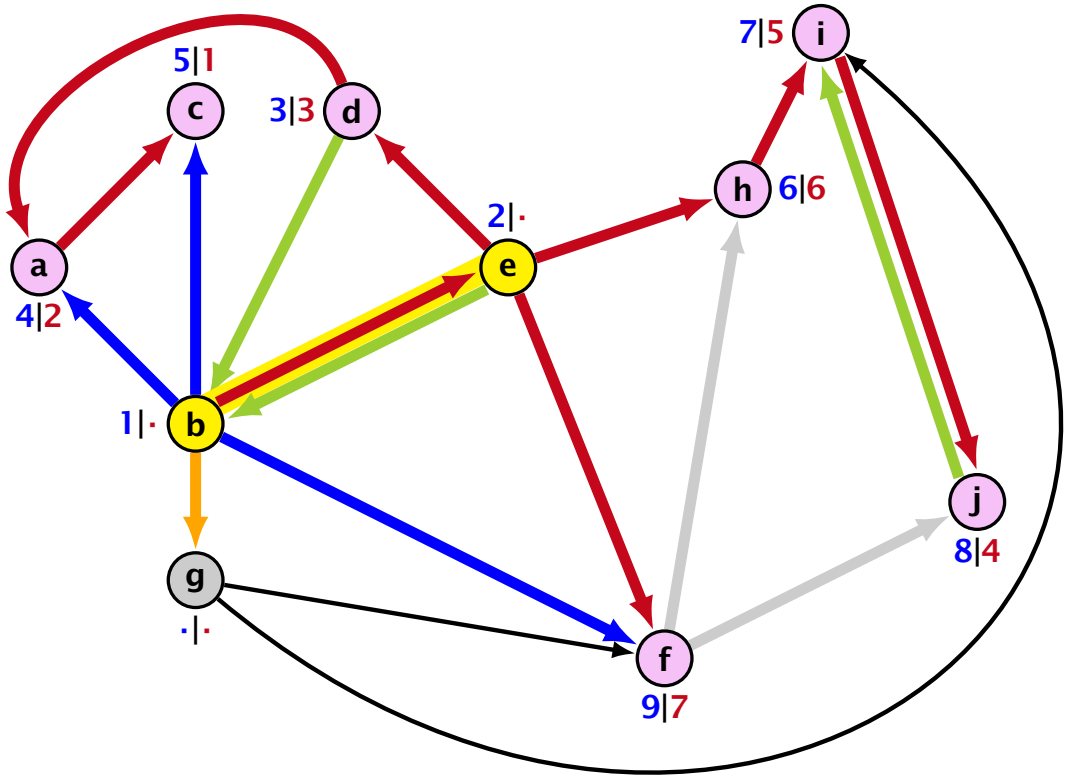
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
9|·

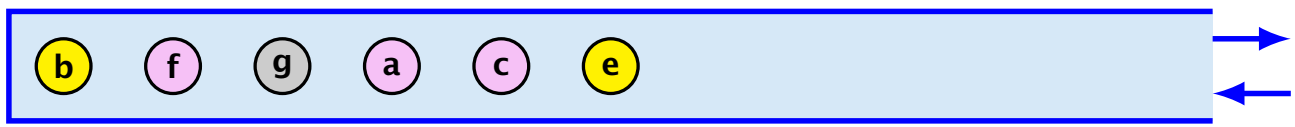
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e
f

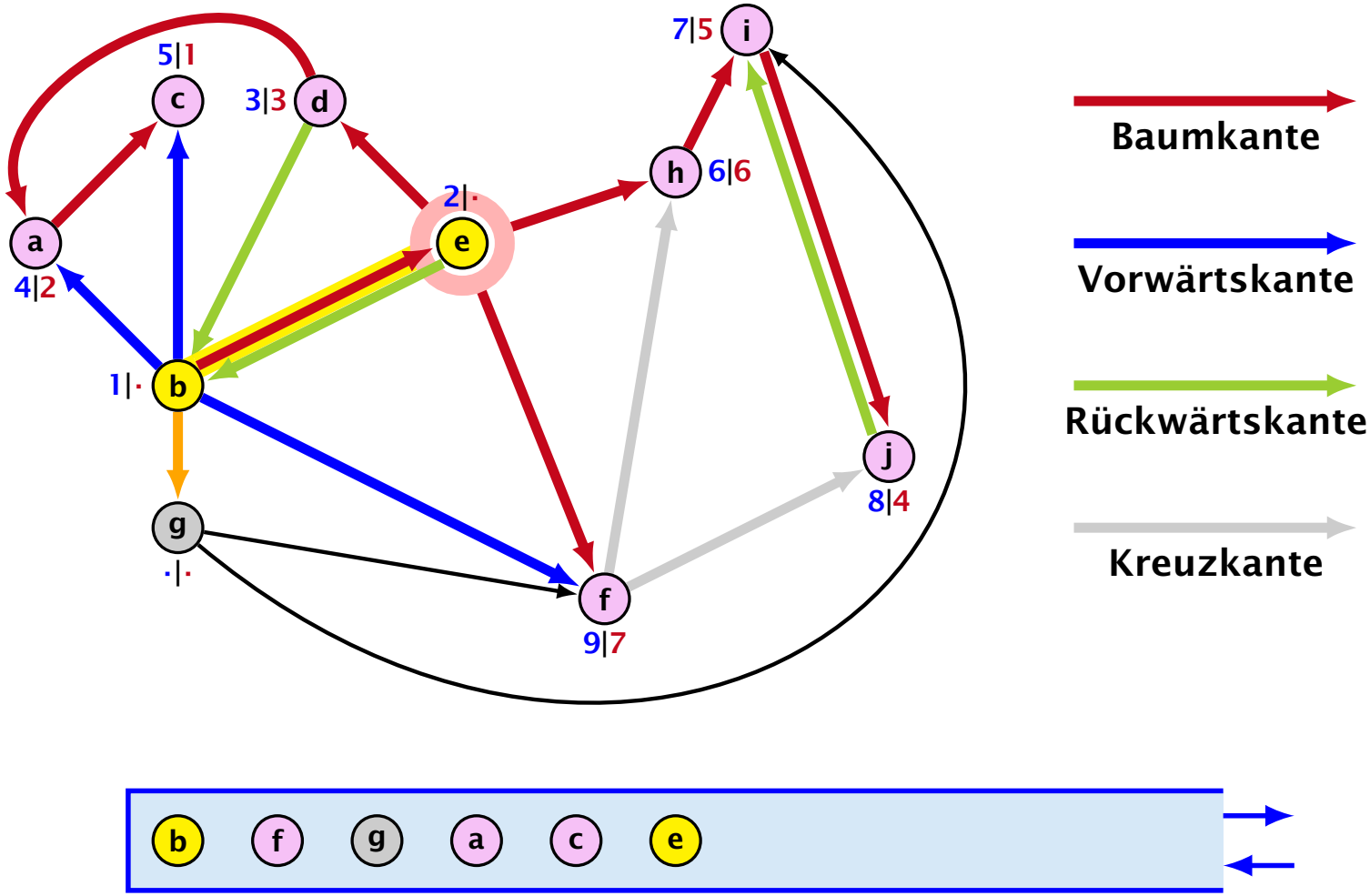
# DFS


a
4|2
5|1
c
3|3
d
1|·
b
2|·
e
h
6|6
7|5
i
j
8|4
g
·|·
f
9|7

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

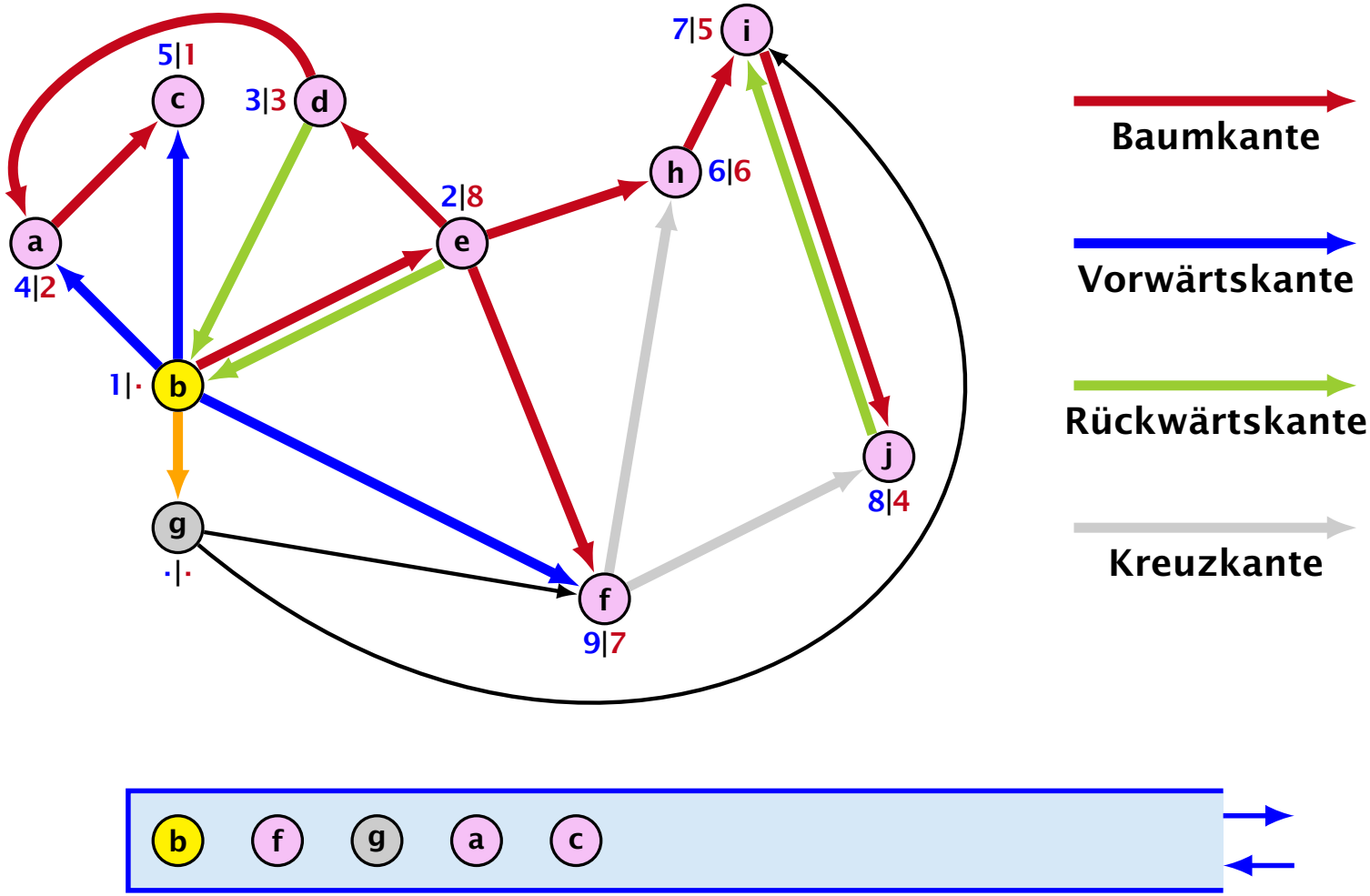

b
f
g
a
c
e

# DFS

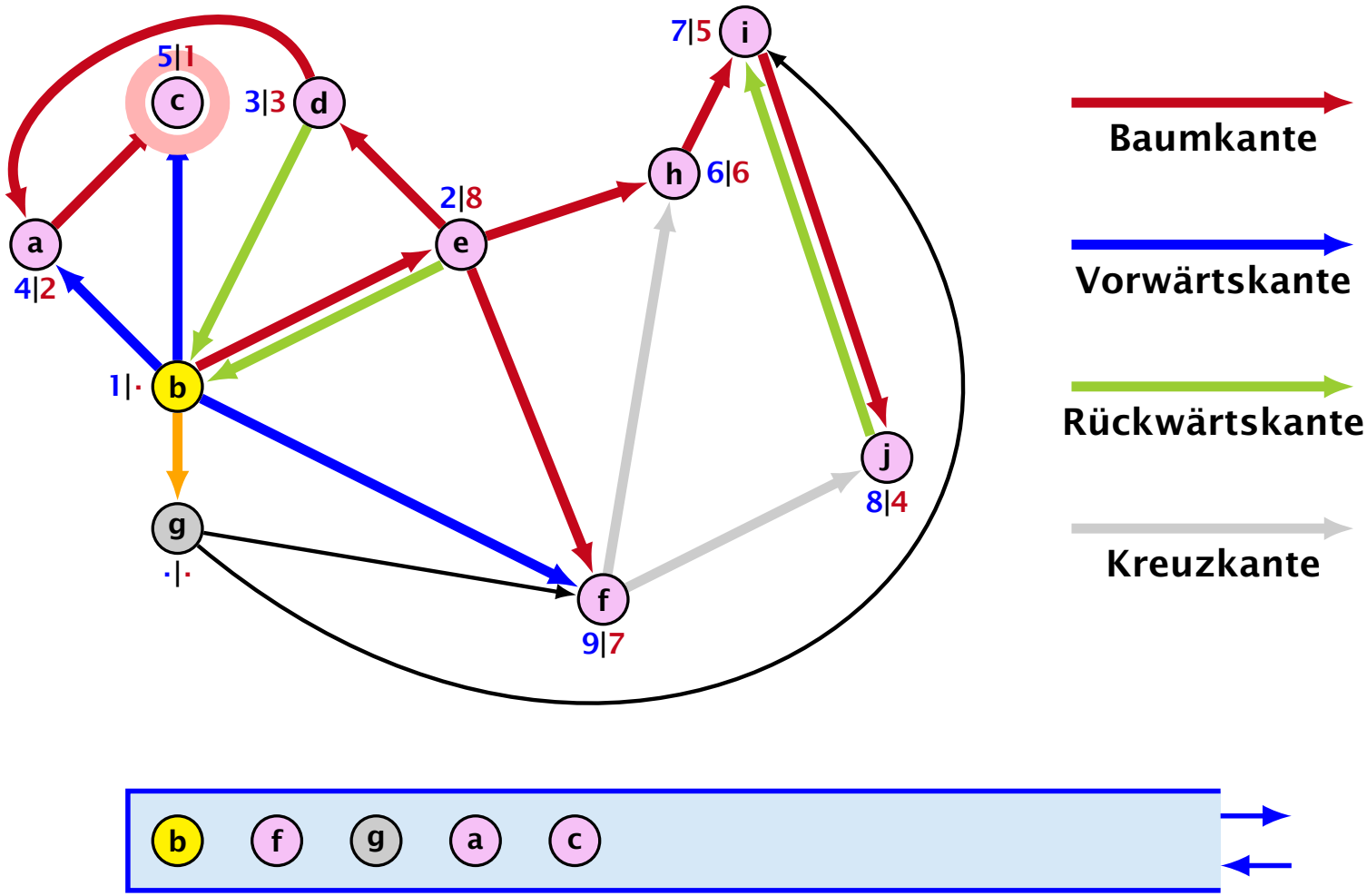

5|1
c
3|3
d
7|5
i
h
6|6
2|·
e
a
4|2
1|·
b
j
8|4
g
·|·
f
9|7
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a
c
e

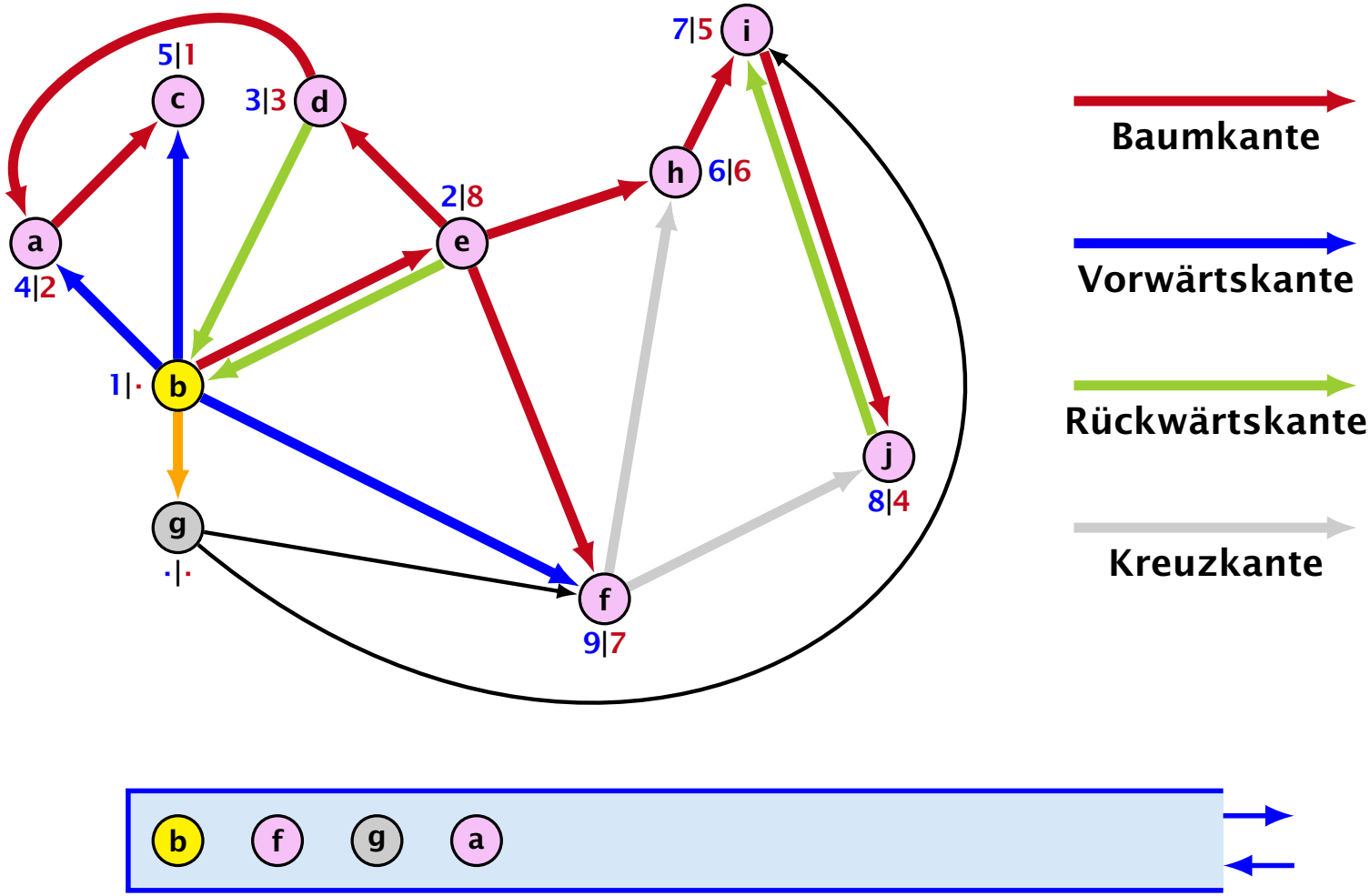
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|8
e
a
4|2
1|·
b
g
·|·
f
9|7
j
8|4
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b
f
g
a
c

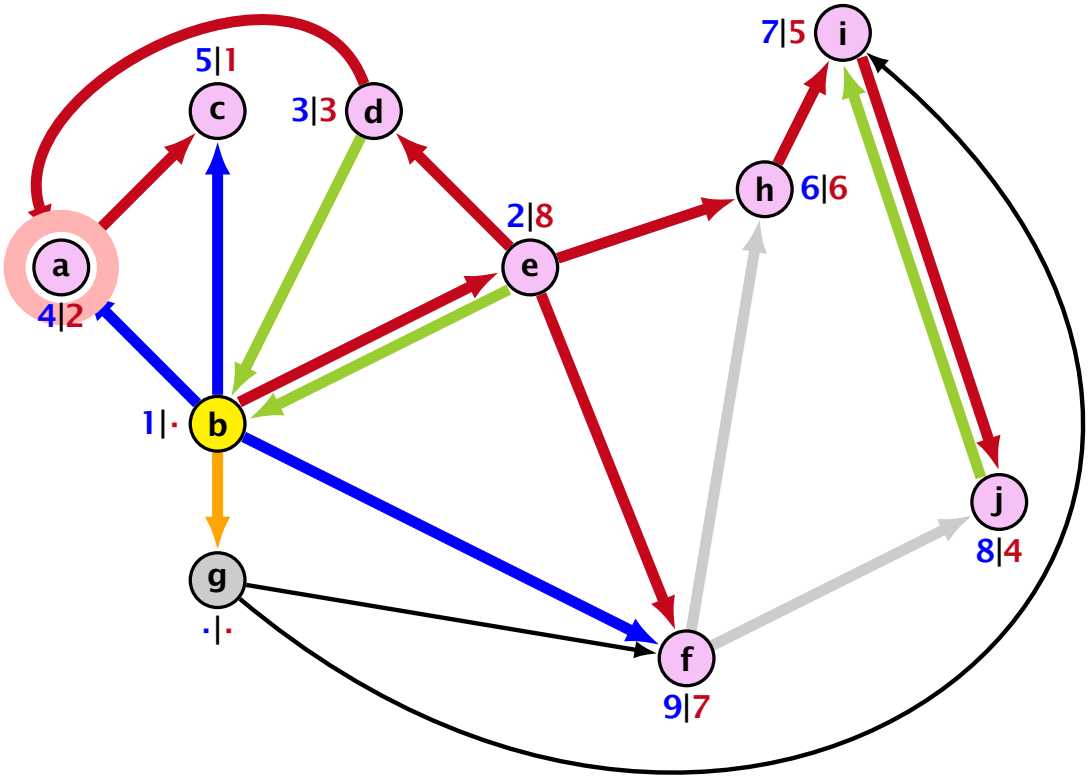
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|8
e
a
4|2
1|·
b
g
·|·
f
9|7
j
8|4
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b
f
g
a
c

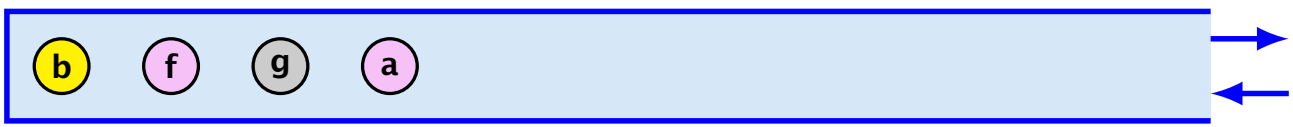
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|8
e
a
4|2
1|·
b
g
·|·
f
9|7
j
8|4
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b
f
g
a

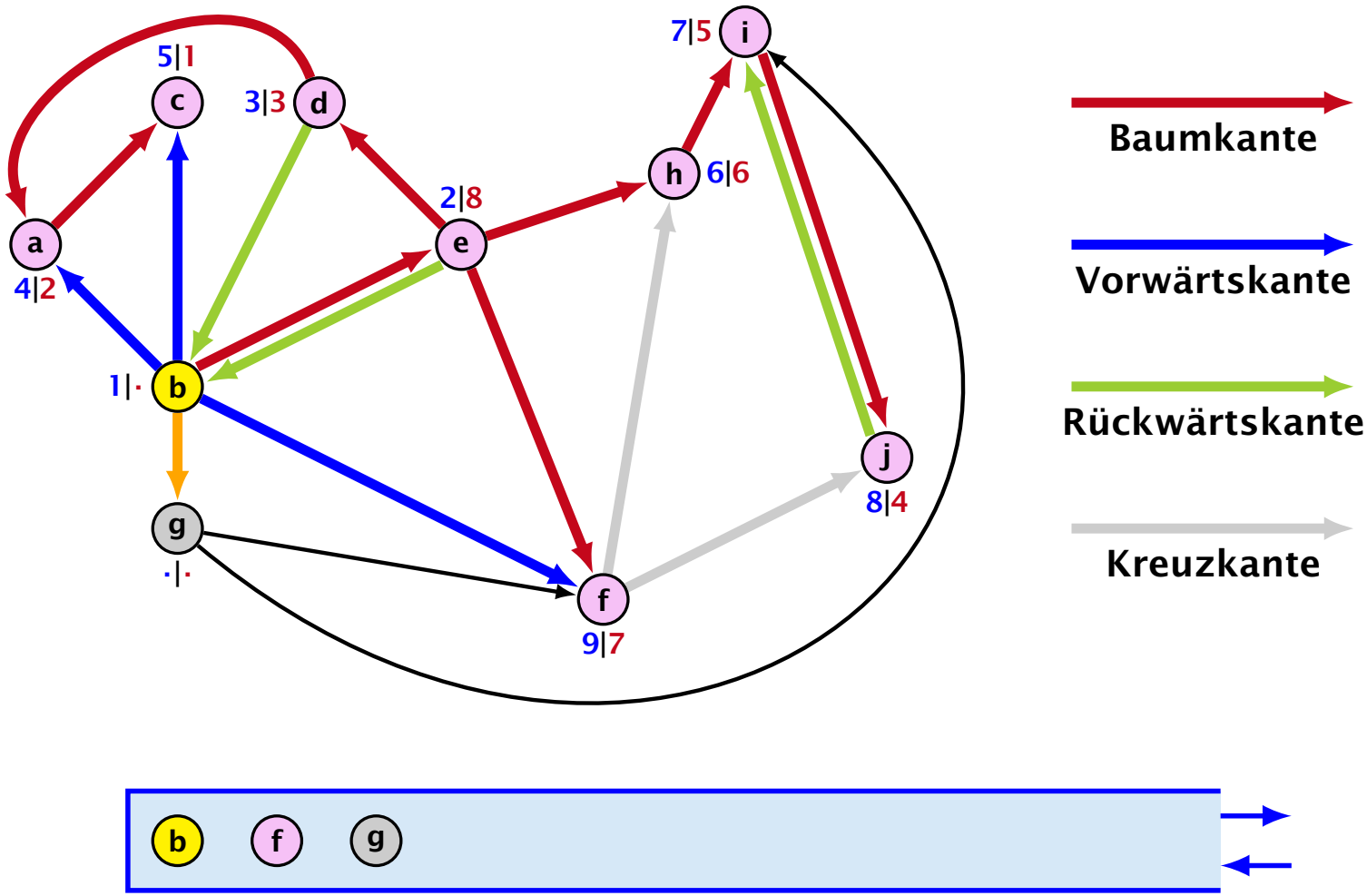
# DFS


5|1
c
3|3
d
7|5
i
2|8
e
h
6|6
a
4|2
1|·
b
j
8|4
g
·|·
f
9|7

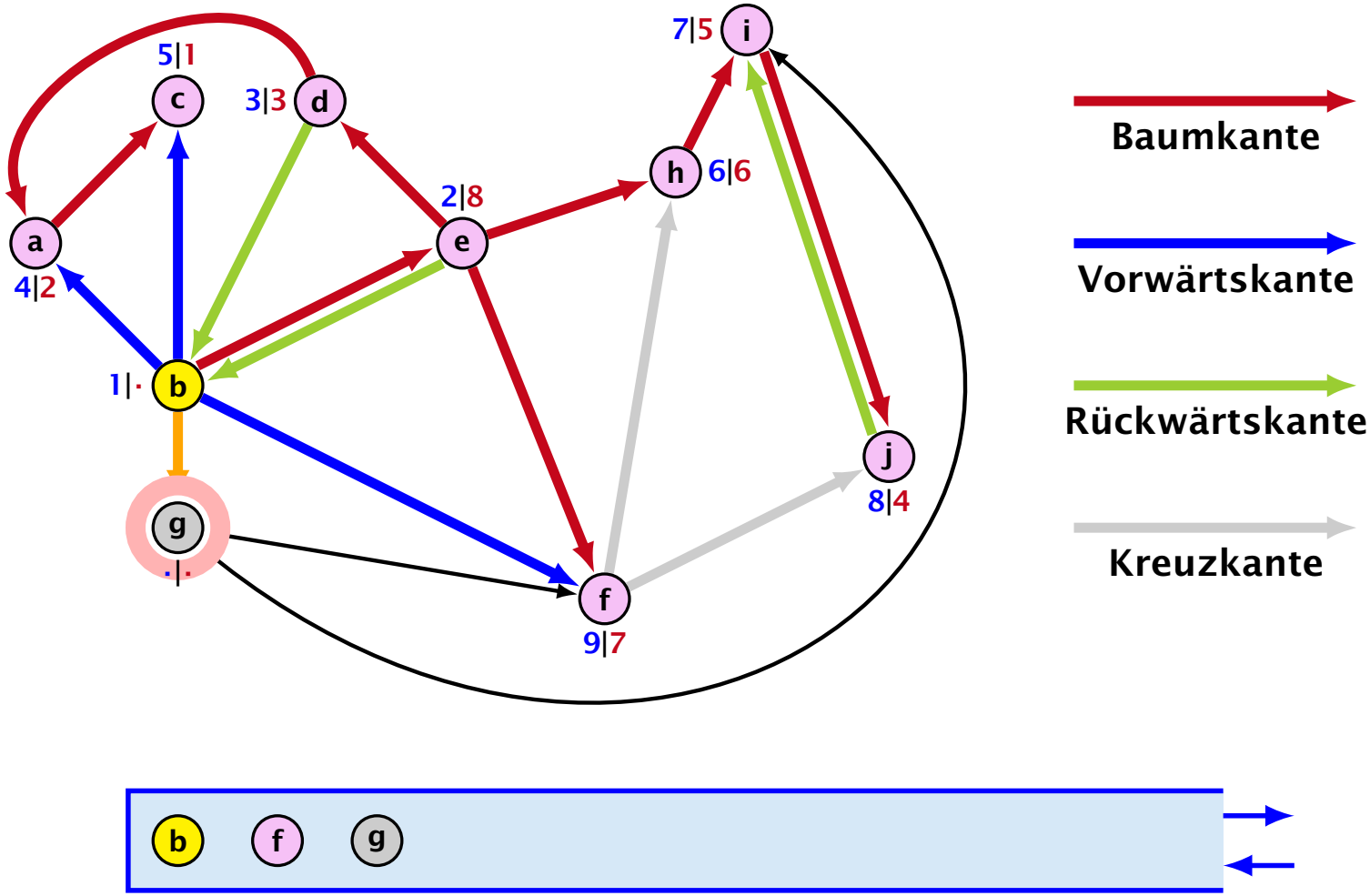
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g
a

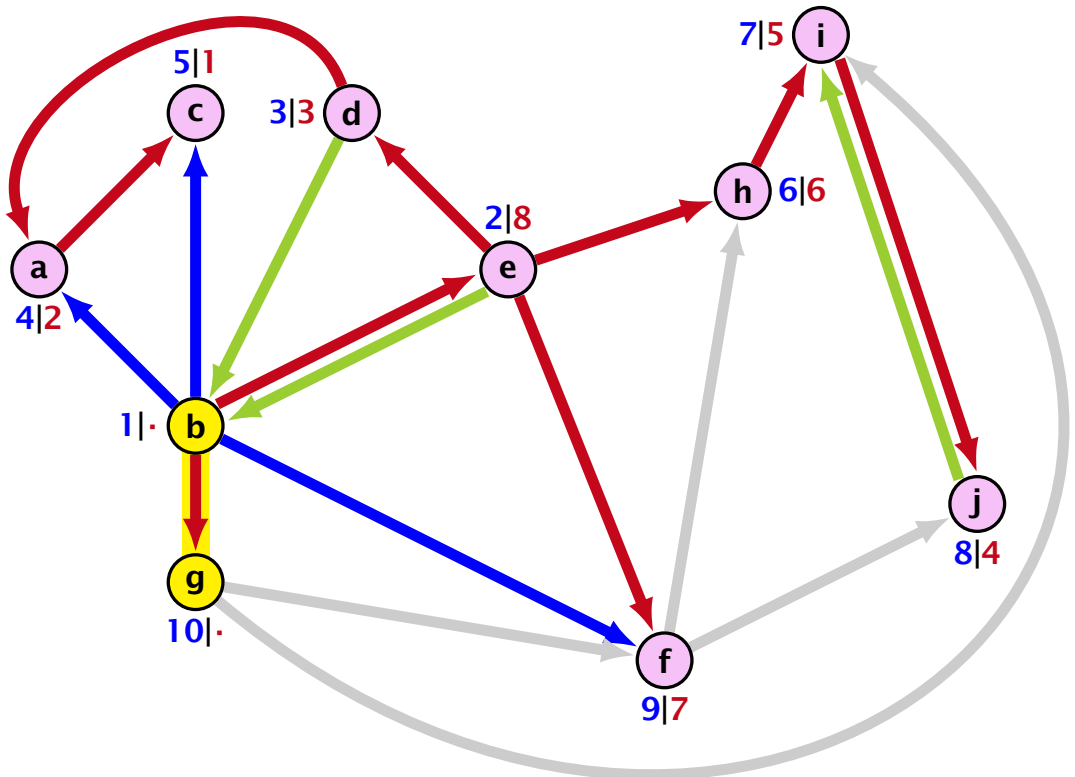
# DFS


a
4|2
c
5|1
d
3|3
e
2|8
h
6|6
i
7|5
j
8|4
b
1|·
g
·|·
f
9|7
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b
f
g

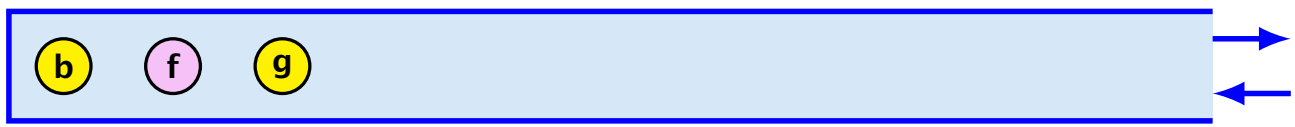
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|8
e
a
4|2
1|·
b
g
·|·
f
9|7
j
8|4
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
b
f
g

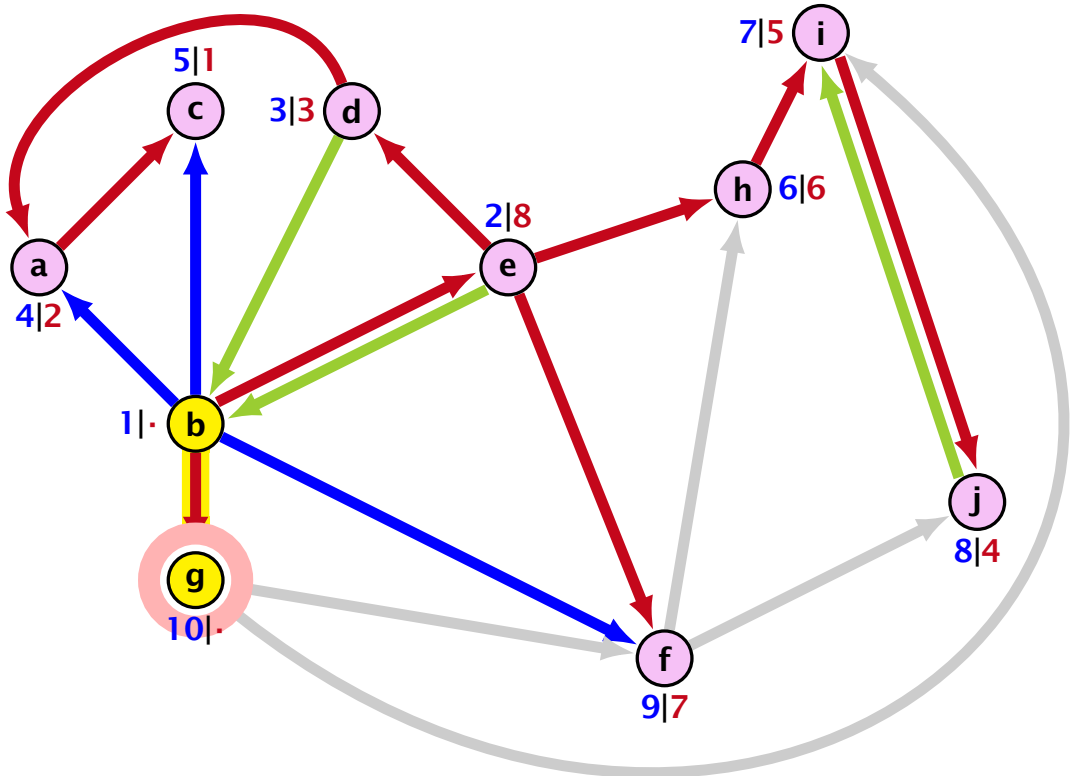
# DFS


5|1
c
3|3
d
7|5
i
a
4|2
2|8
e
h
6|6
1|·
b
j
8|4
g
10|·
f
9|7

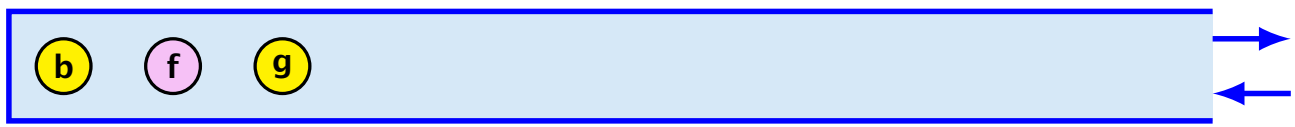
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g

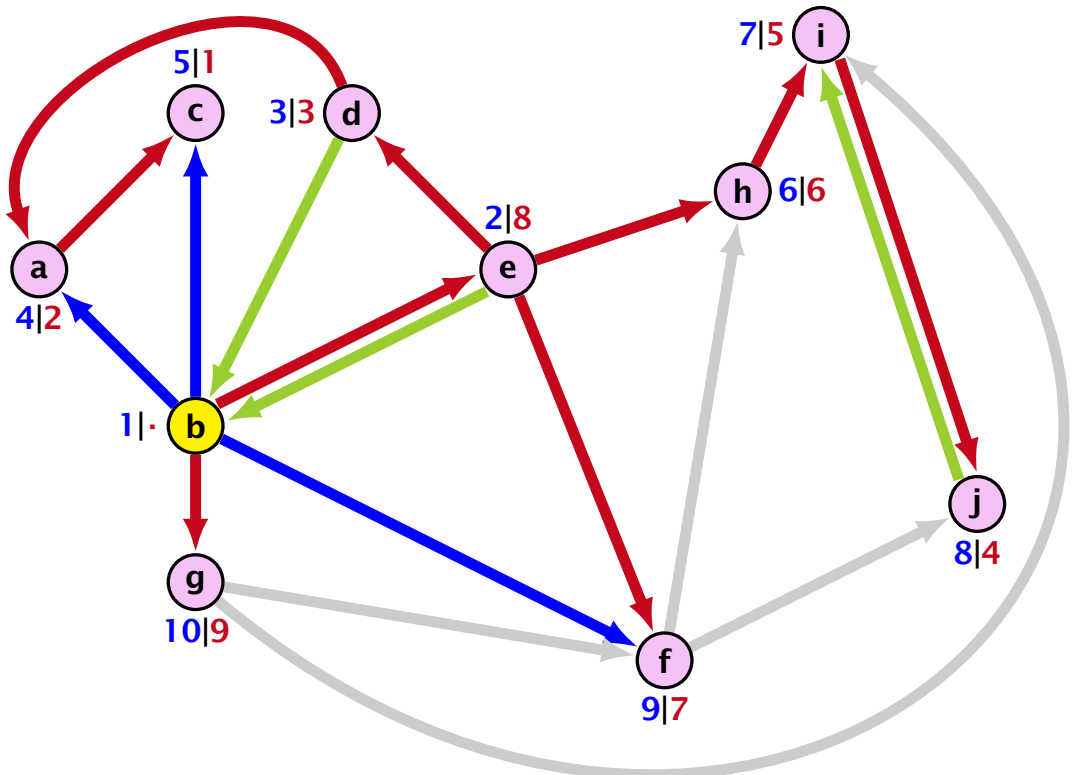
# DFS


5|1
c
3|3
d
7|5
i
a
4|2
2|8
e
h
6|6
1|·
b
j
8|4
g
10|·
f
9|7

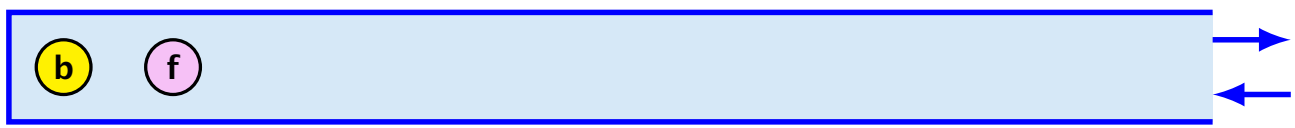
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f
g

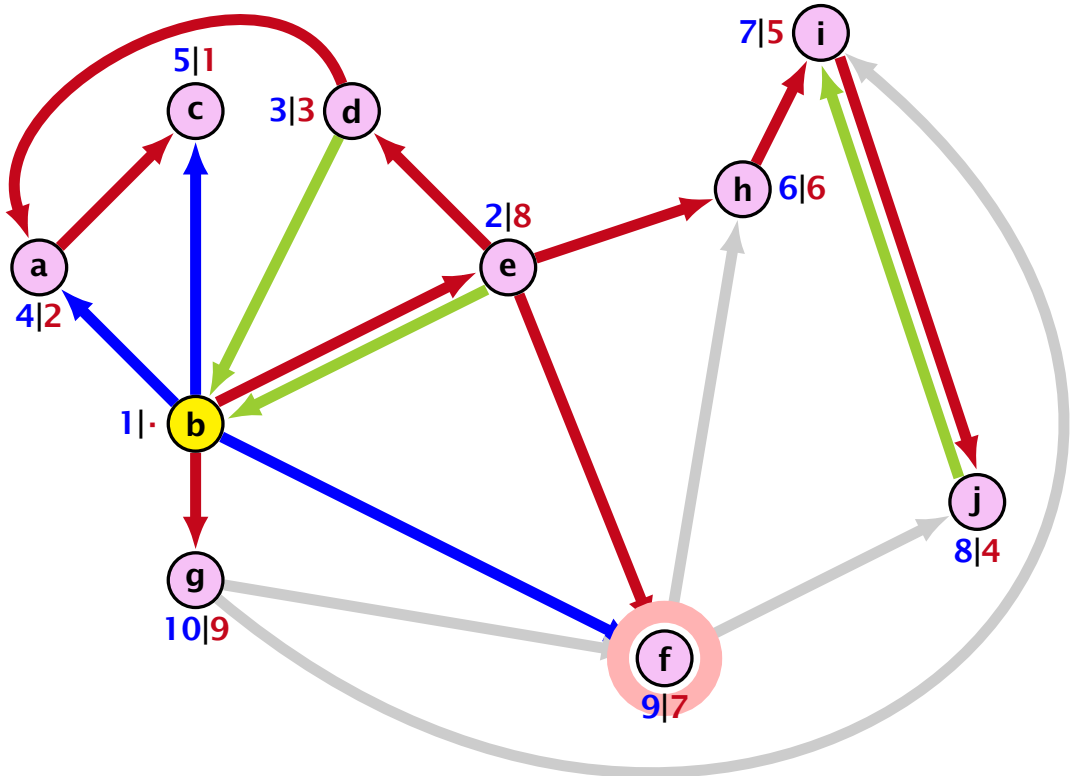
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|8
e
a
4|2
1|·
b
g
10|9
f
9|7
j
8|4

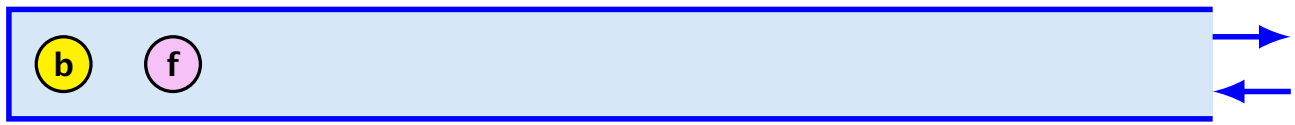
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f

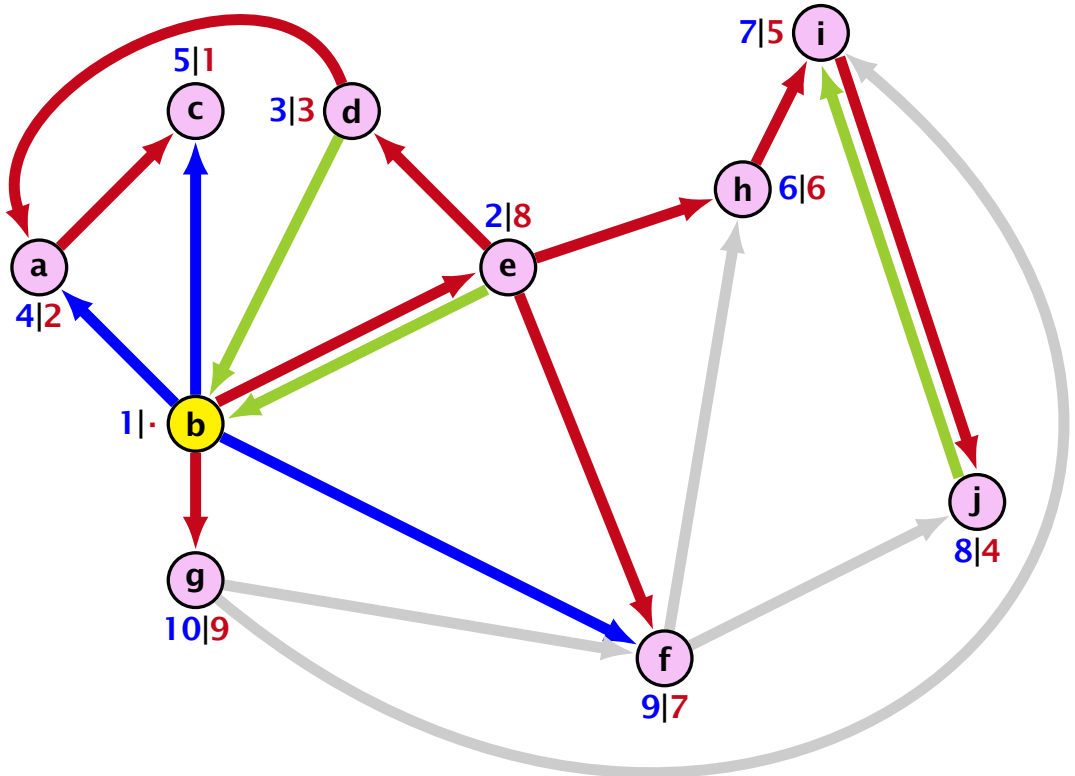
# DFS


5|1
c
3|3
d
7|5
i
h
6|6
2|8
e
a
4|2
1|·
b
g
10|9
f
9|7
j
8|4

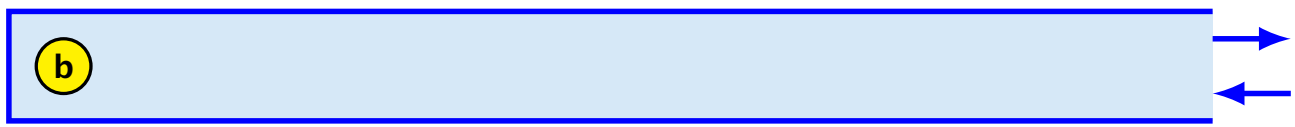
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b
f

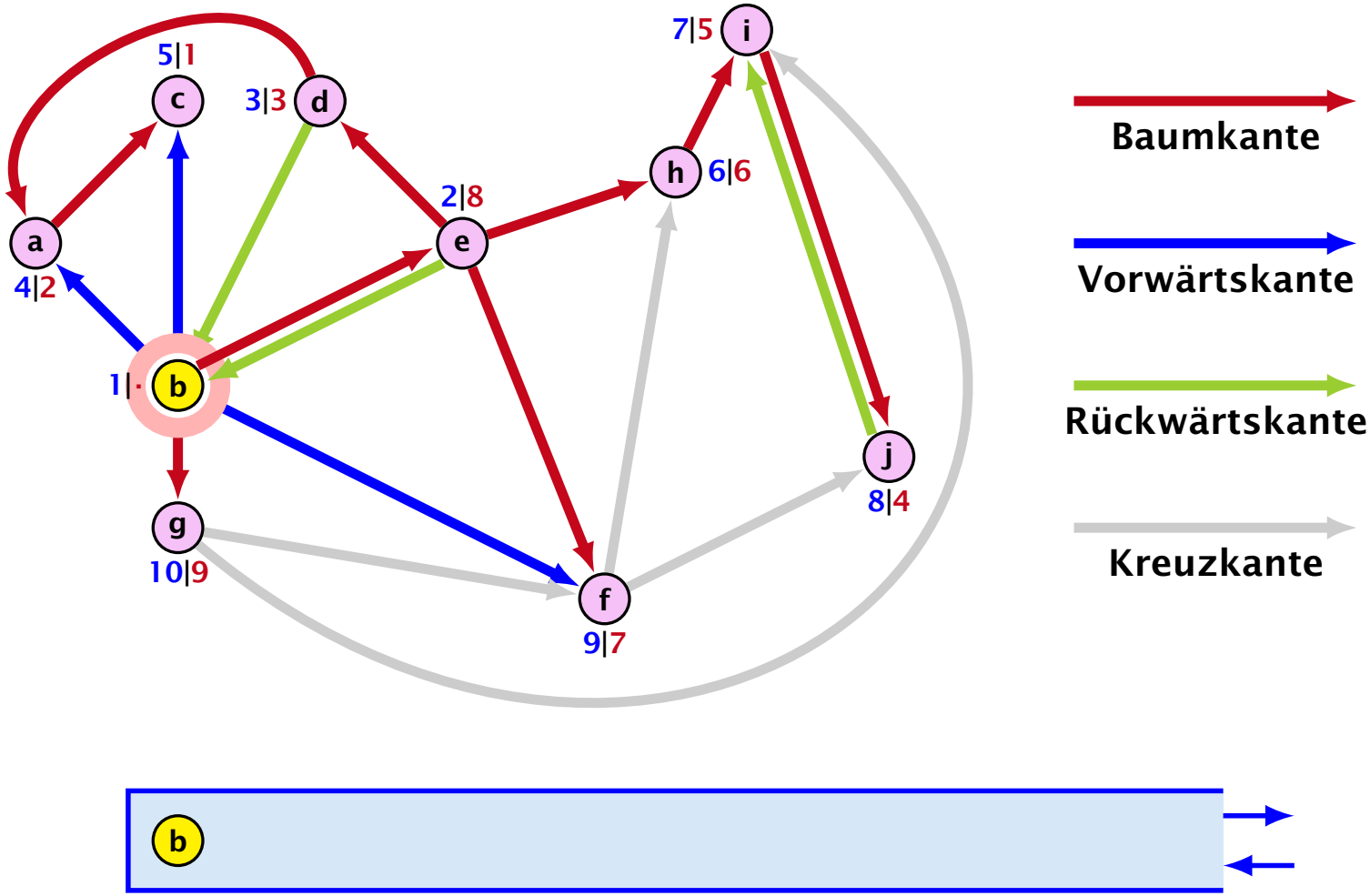
# DFS


a
4|2
c
5|1
d
3|3
e
2|8
h
6|6
i
7|5
j
8|4
b
1|·
g
10|9
f
9|7

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

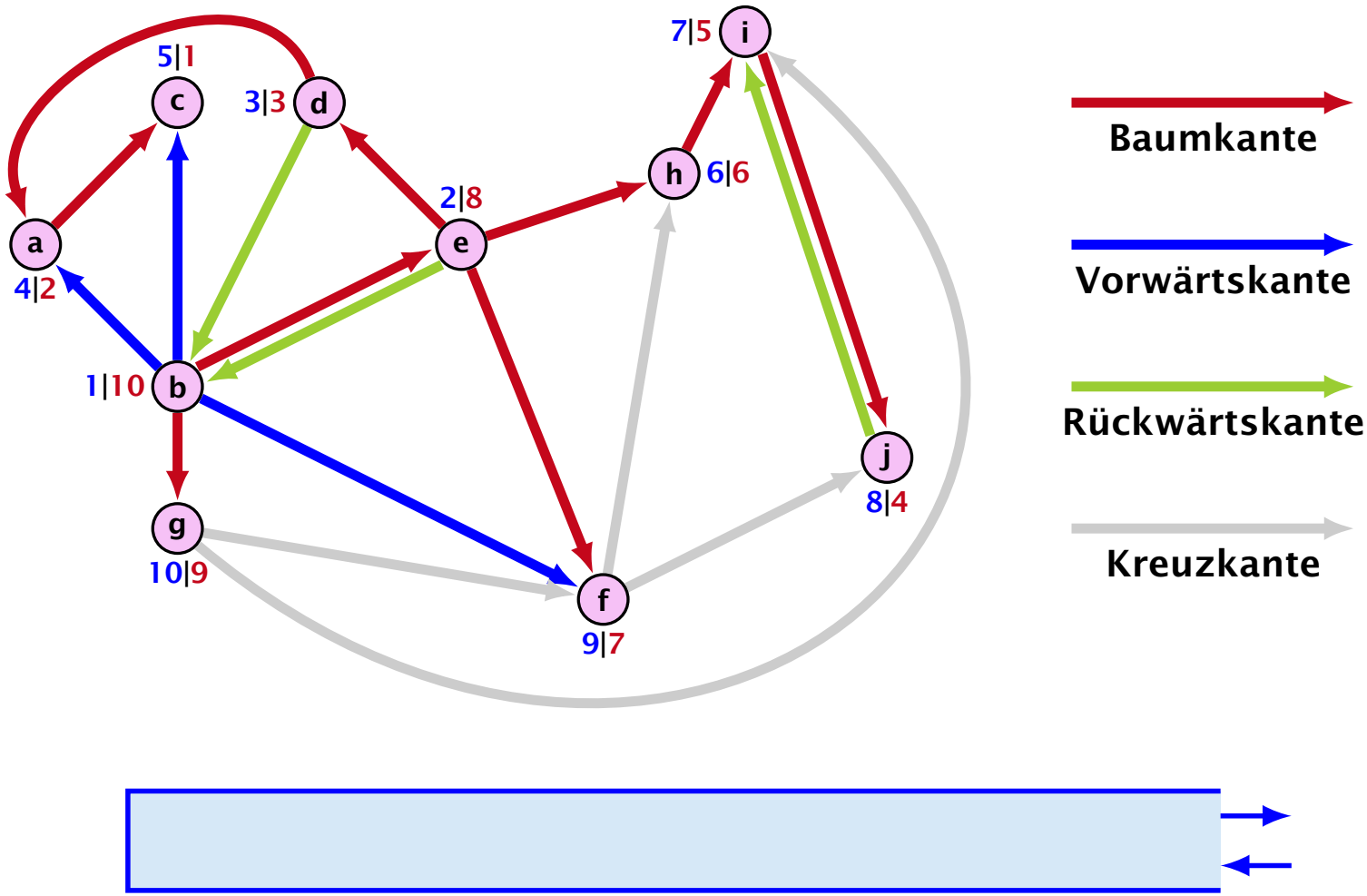

b

# DFS

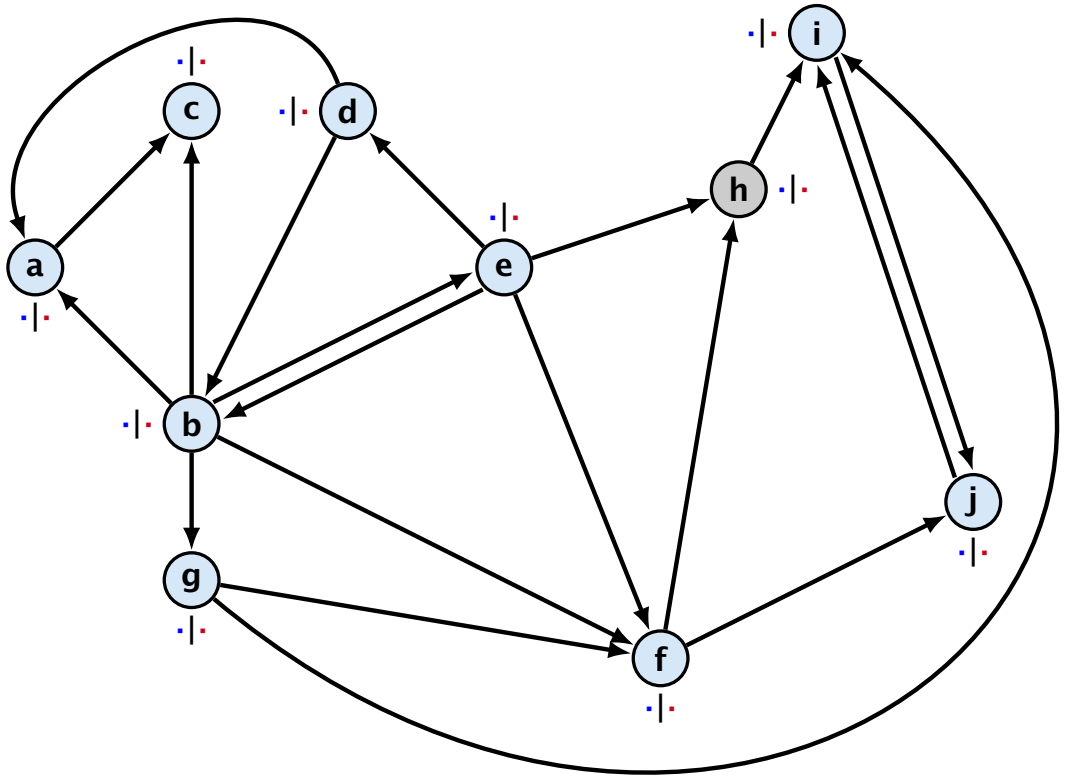

a
4|2
c
5|1
d
3|3
e
2|8
h
6|6
i
7|5
j
8|4
b
1|·
g
10|9
f
9|7
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b

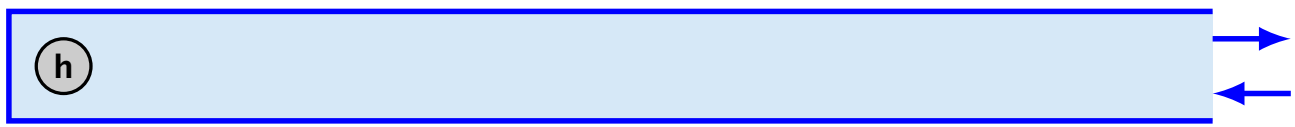
# DFS


a
4|2
b
1|10
c
5|1
d
3|3
e
2|8
f
9|7
g
10|9
h
6|6
i
7|5
j
8|4
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

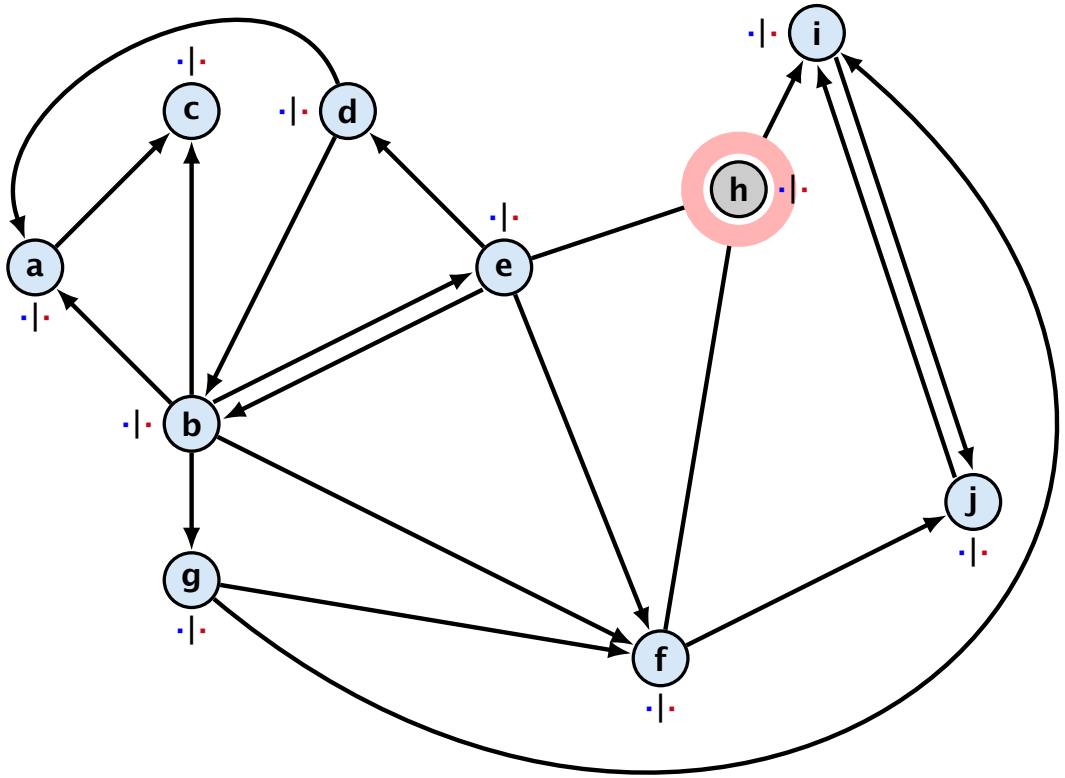
# DFS


a
b
c
d
e
f
g
h
i
j

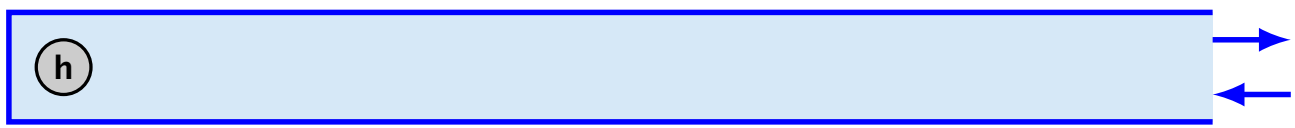
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h

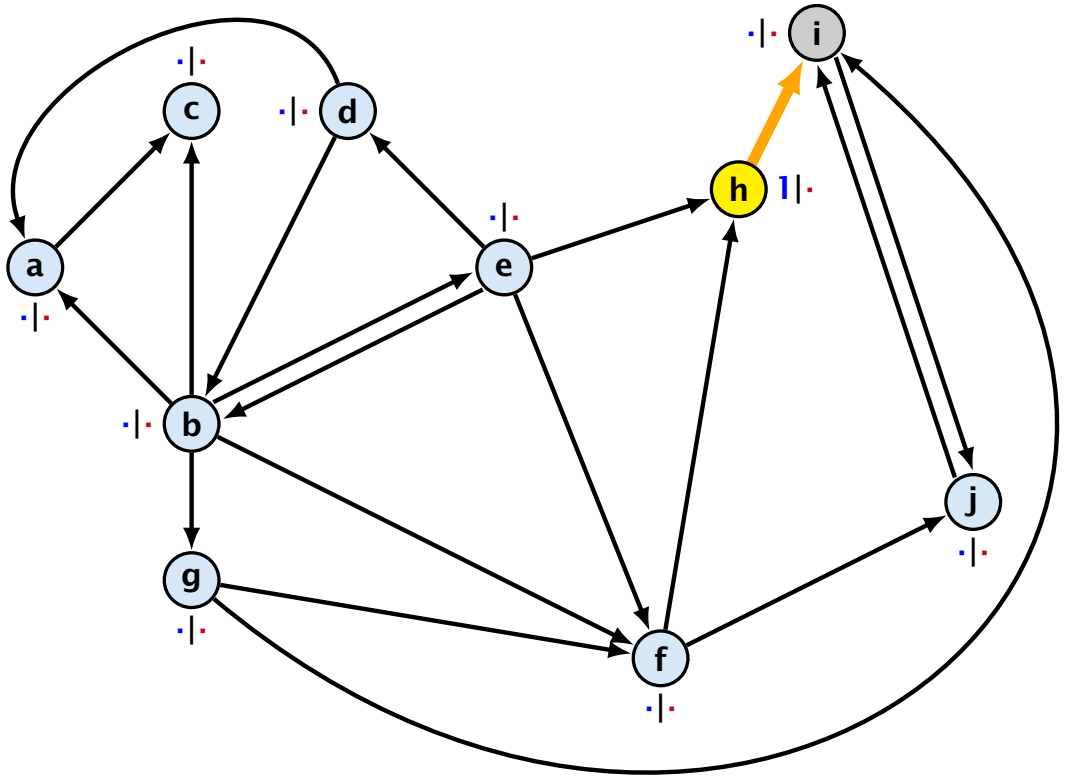
# DFS


i
c
d
h
a
e
b
j
g
f

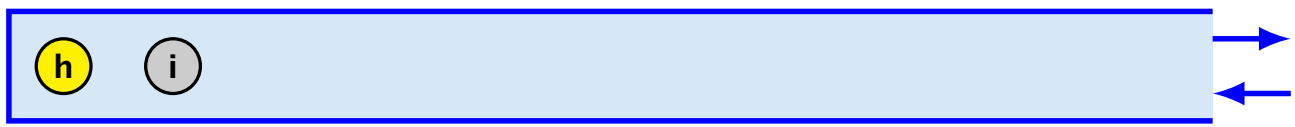
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h

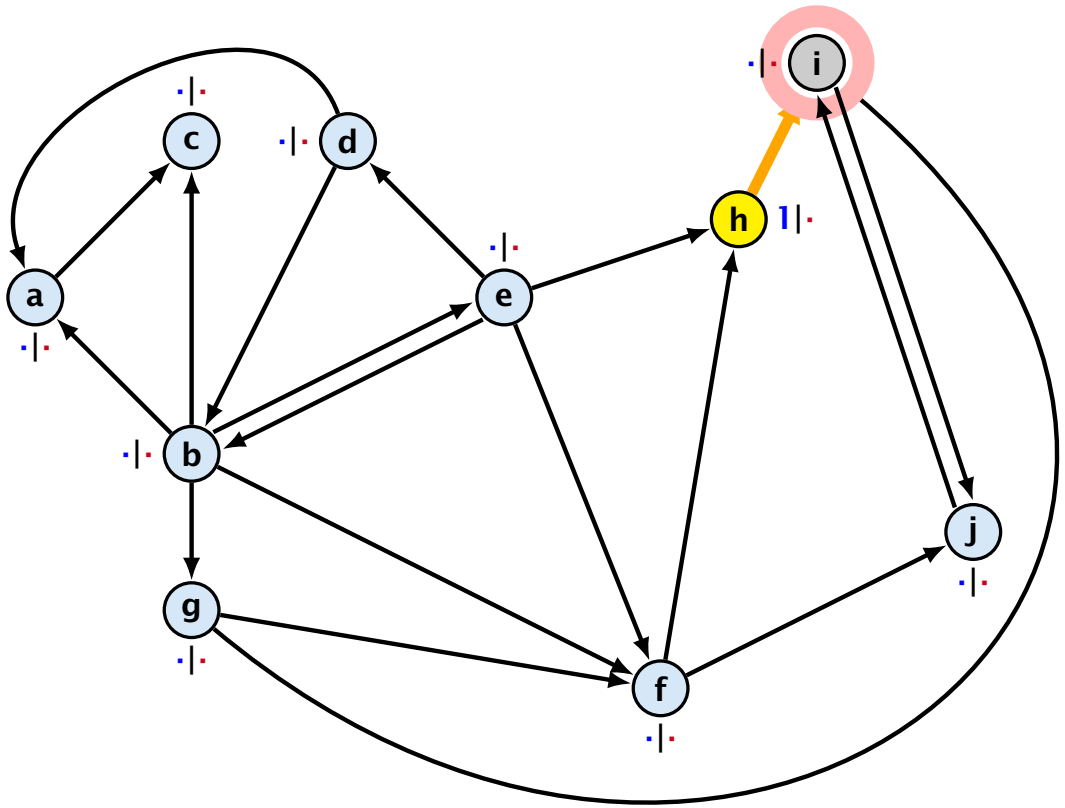
# DFS


i
c
d
h
1
a
e
b
j
g
f

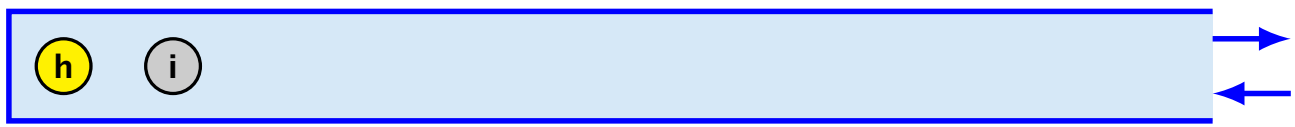
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h
i

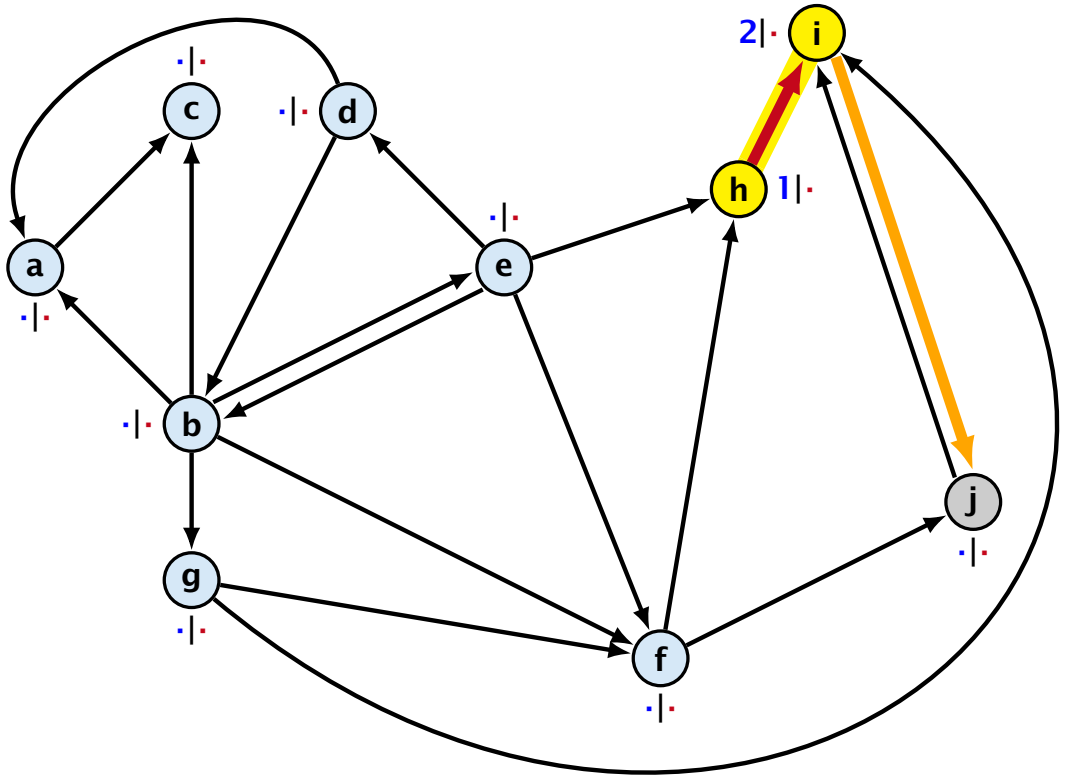
# DFS


i
c
d
h
1
a
e
b
j
g
f

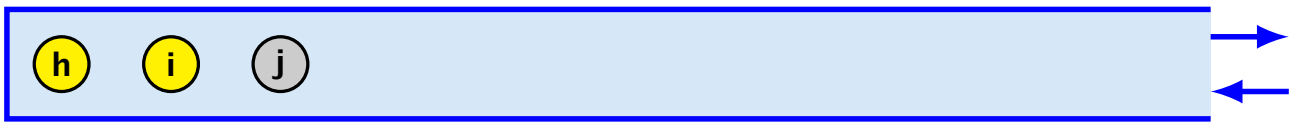
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h
i

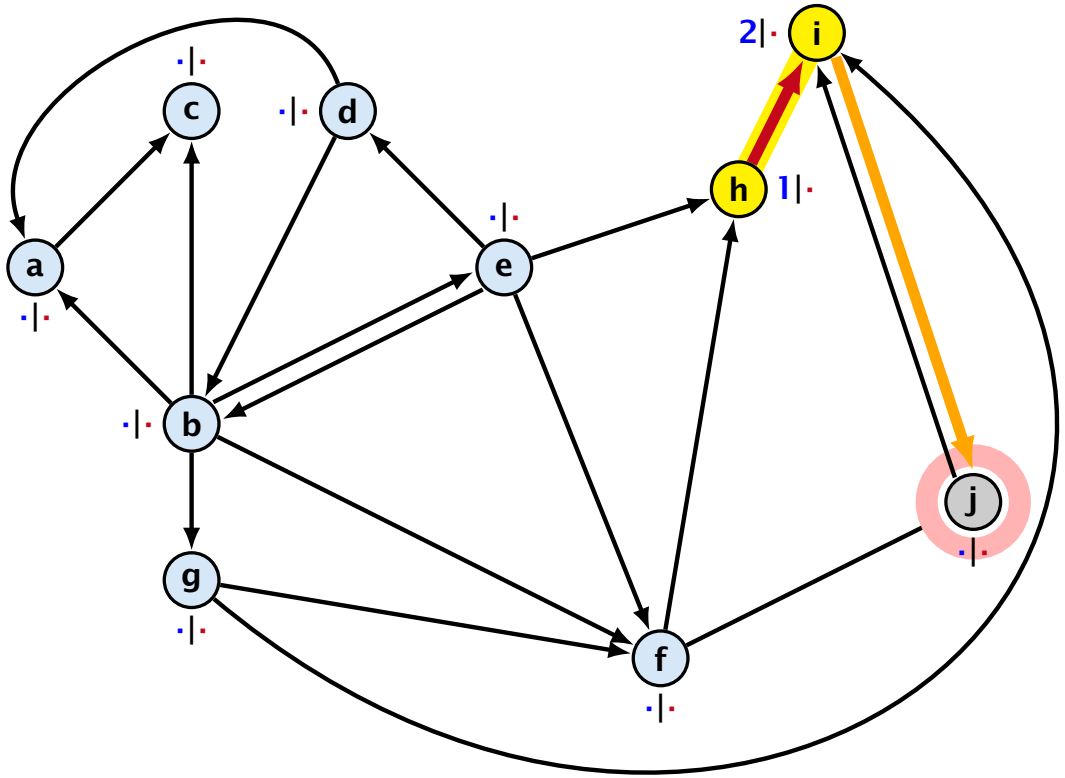
# DFS


2|·
i
h
1|·
c
d
a
e
b
g
f
j
·|·

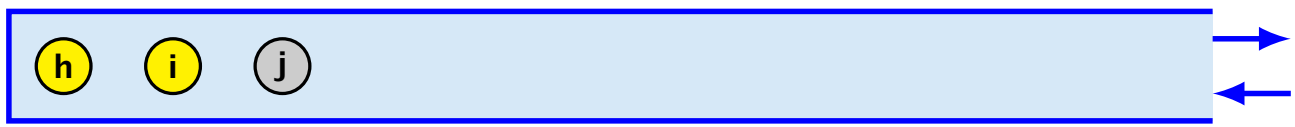
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h
i
j

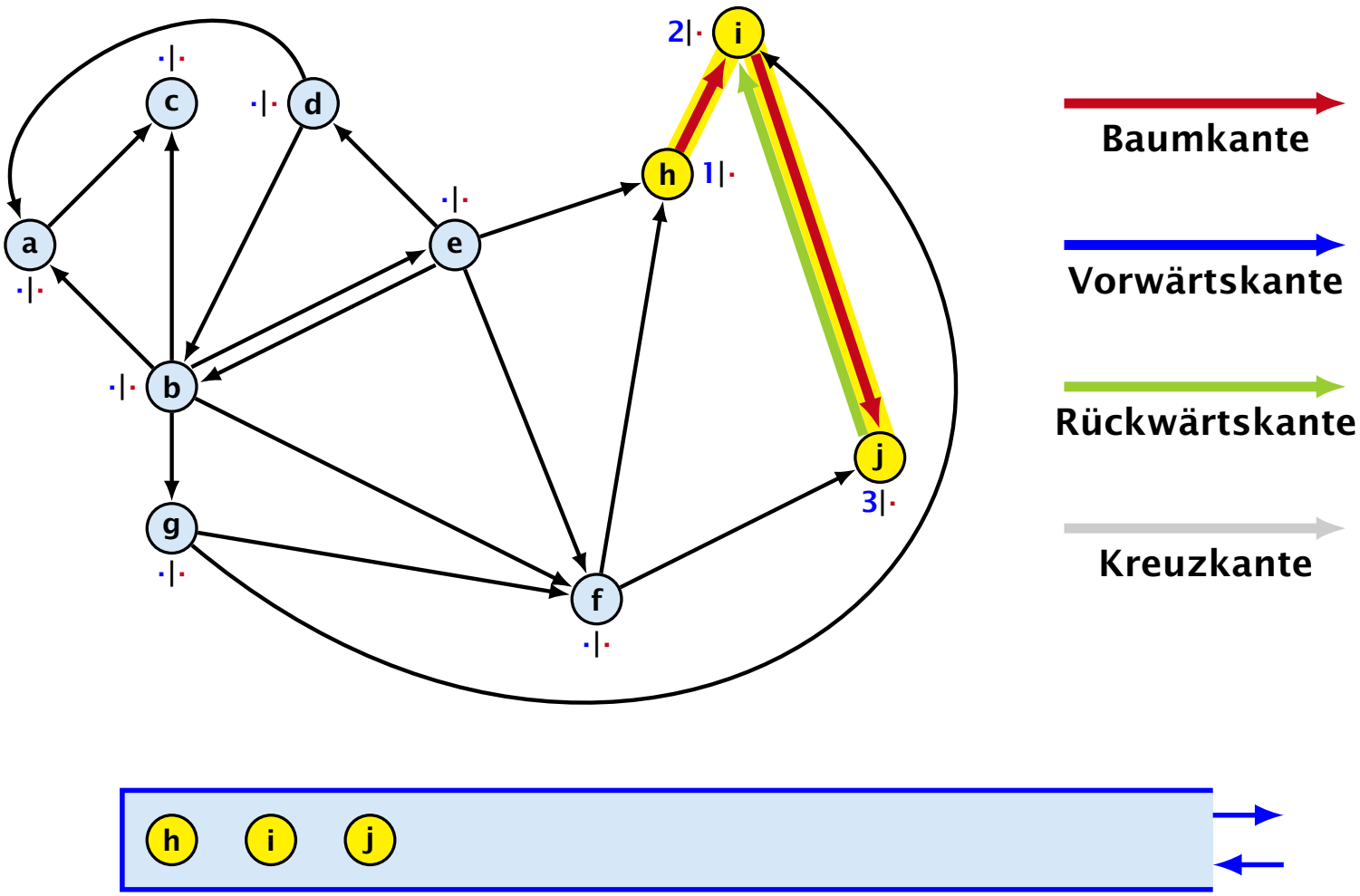
# DFS


2|·
i
h
1|·
c
d
a
e
b
g
f
j

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

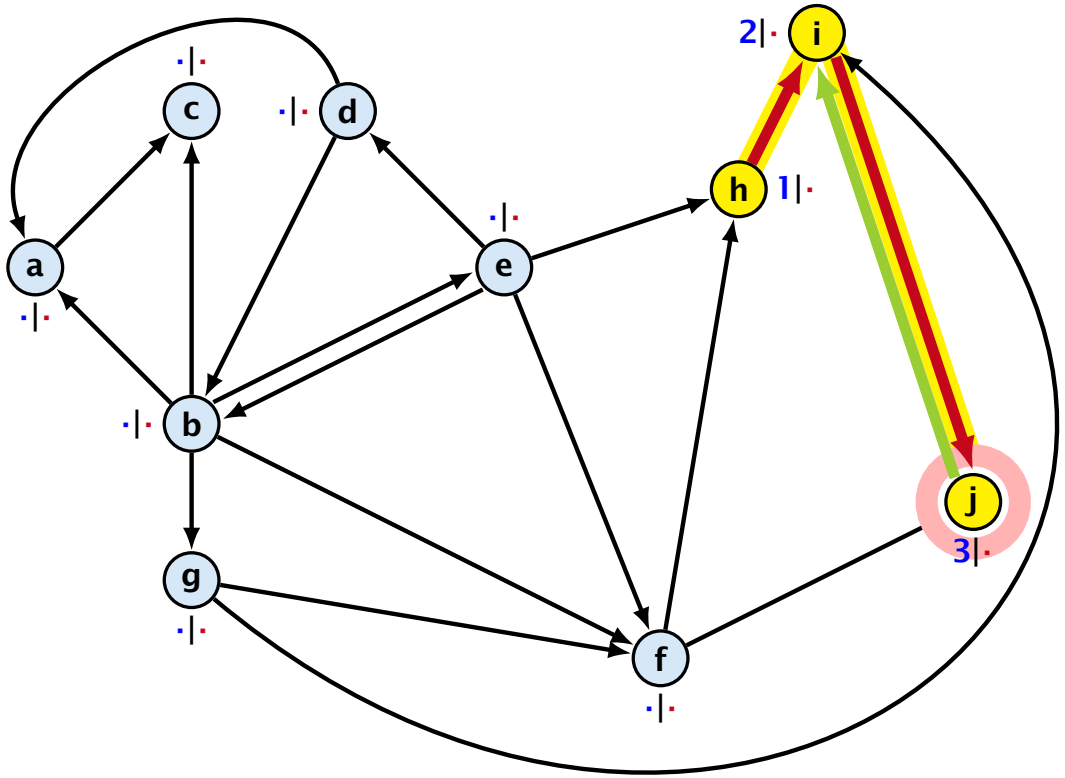

h
i
j

# DFS



a
b
c
d
e
f
g
h
i
j
·|·
1|·
2|·
3|·
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

# DFS

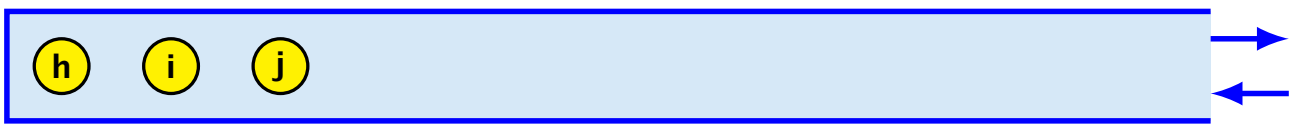



Baumkante

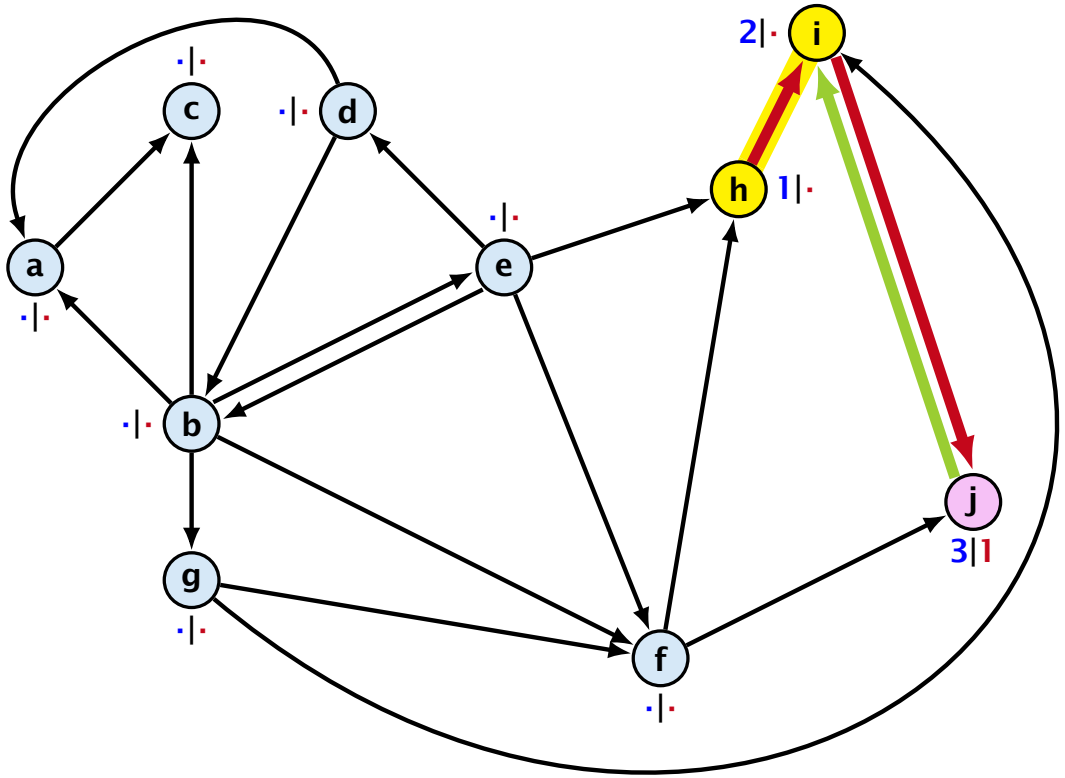
Vorwärtskante

Rückwärtskante

Kreuzkante



# DFS

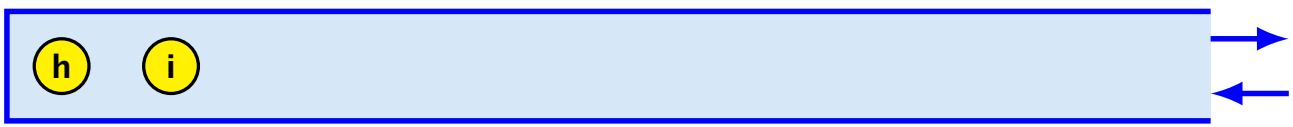



Baumkante

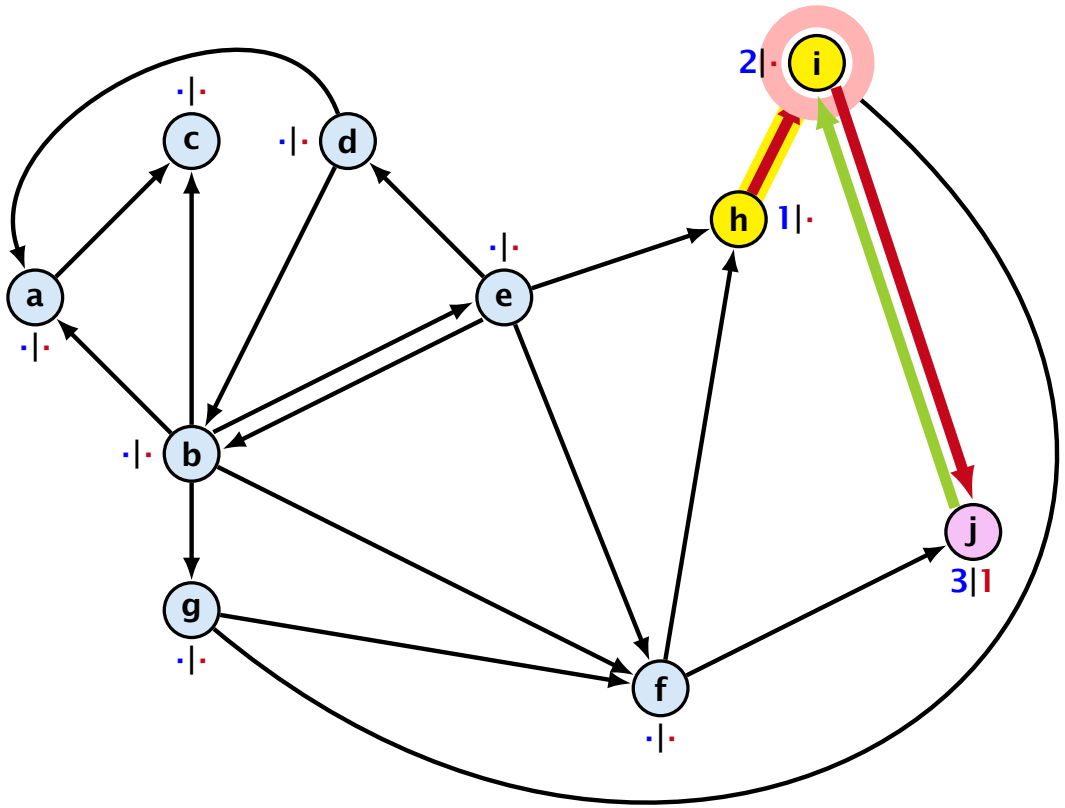
Vorwärtskante

Rückwärtskante

Kreuzkante



# DFS

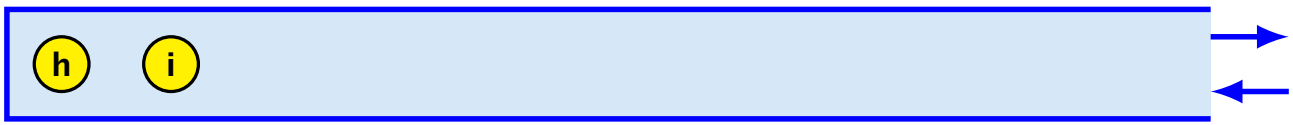



Baumkante

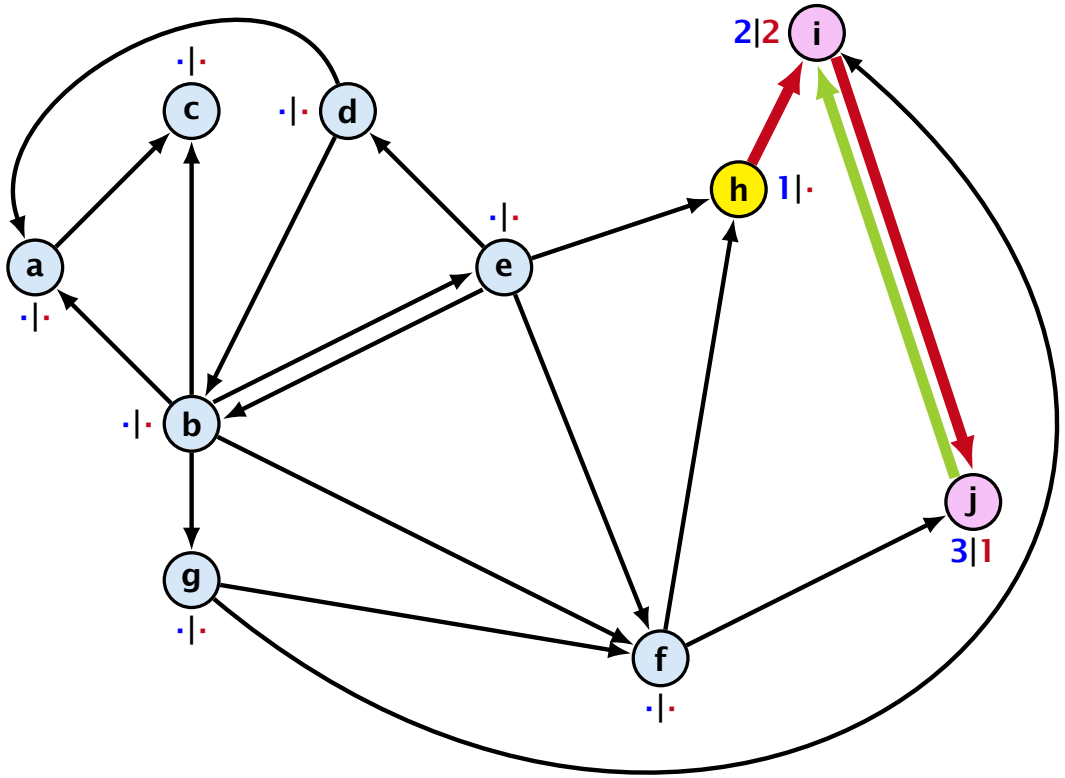
Vorwärtskante

Rückwärtskante

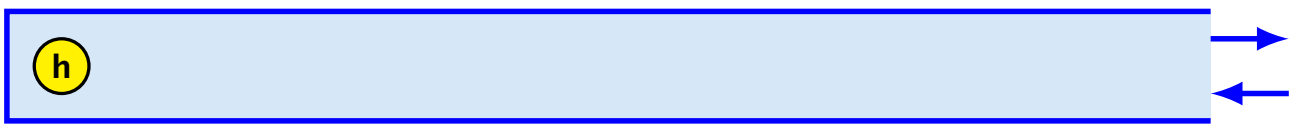
Kreuzkante



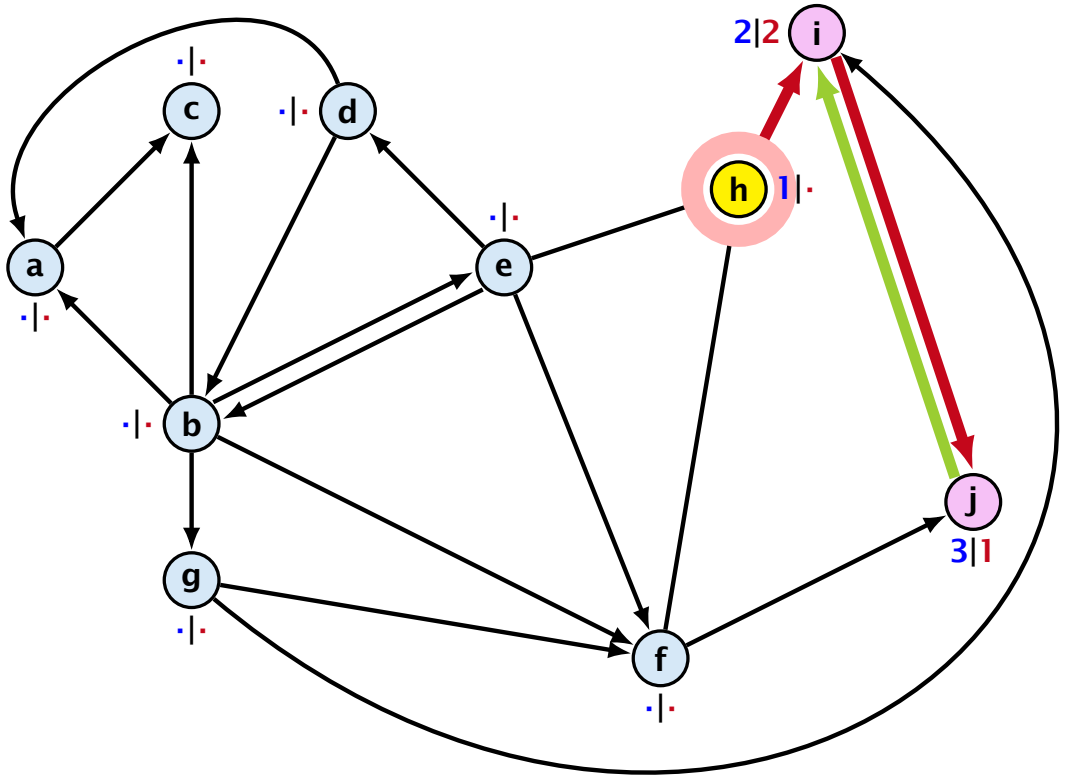
# DFS


2|2
i
h
1|·
c
d
a
e
b
g
f
j
3|1
·|·

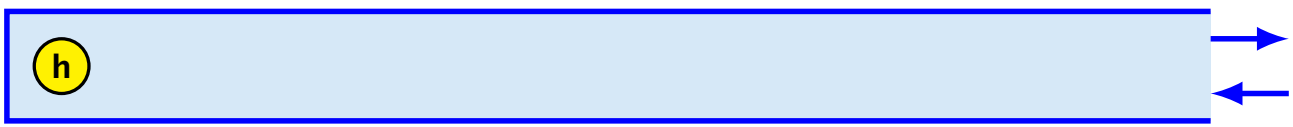
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h

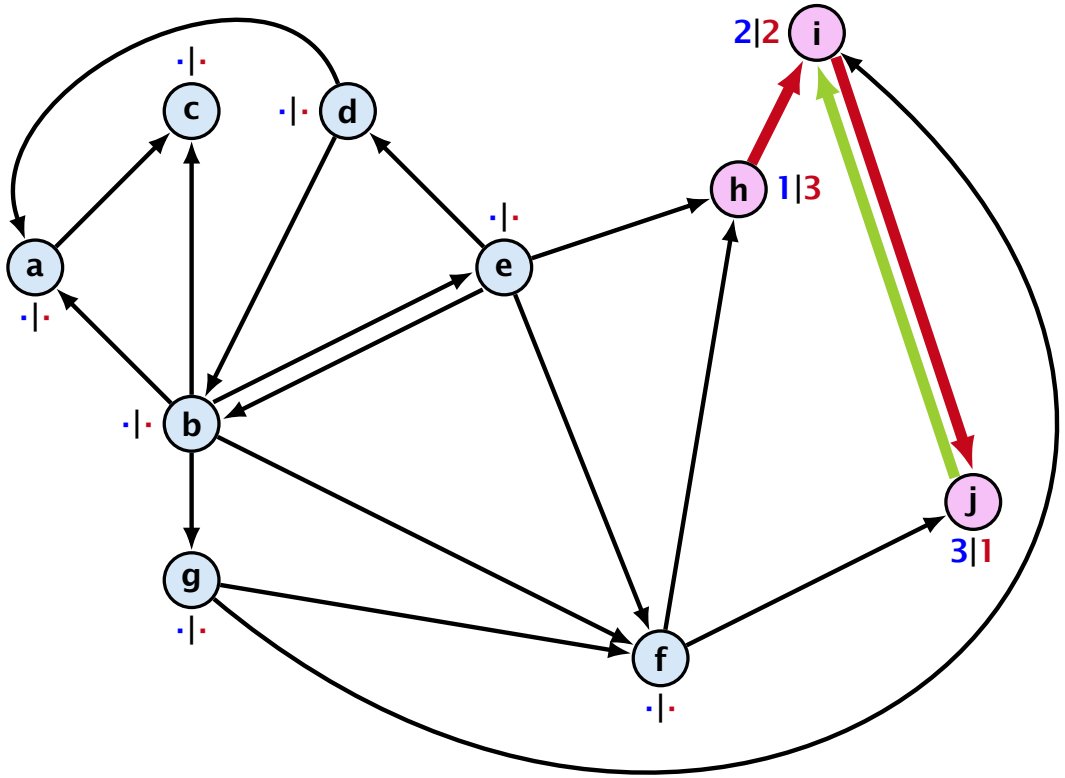
# DFS


2|2
i
h
1|·
c
d
a
e
b
g
f
j
3|1

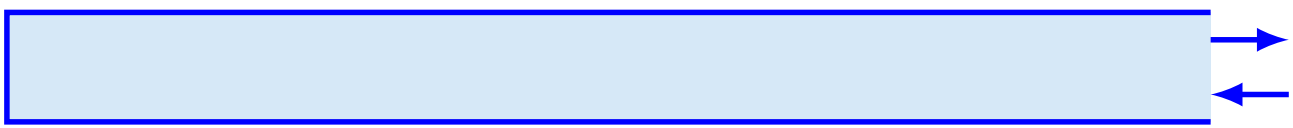
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


h

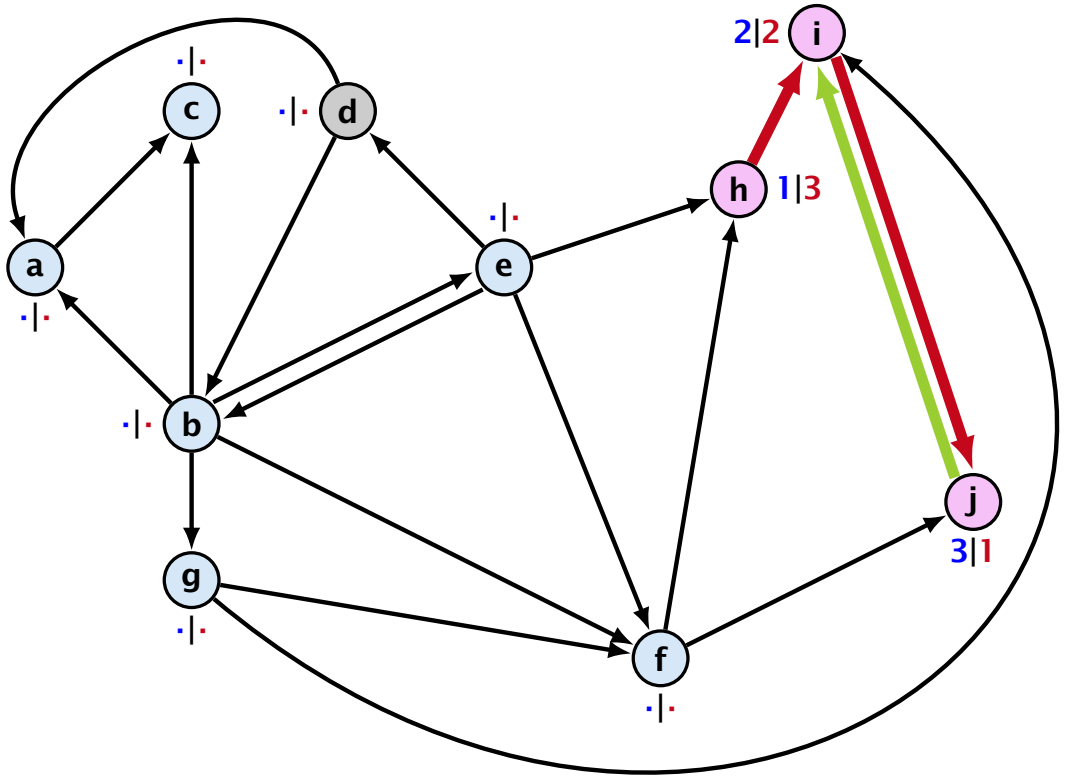
# DFS


2|2
i
h
1|3
c
d
a
e
b
j
3|1
g
f

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante



# DFS

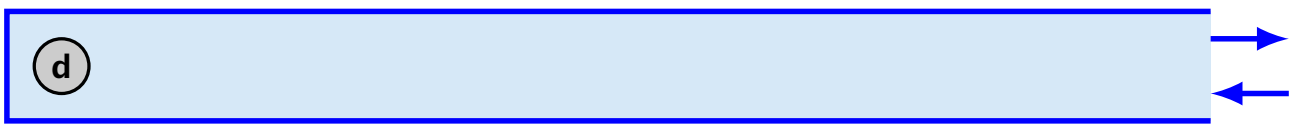



Baumkante

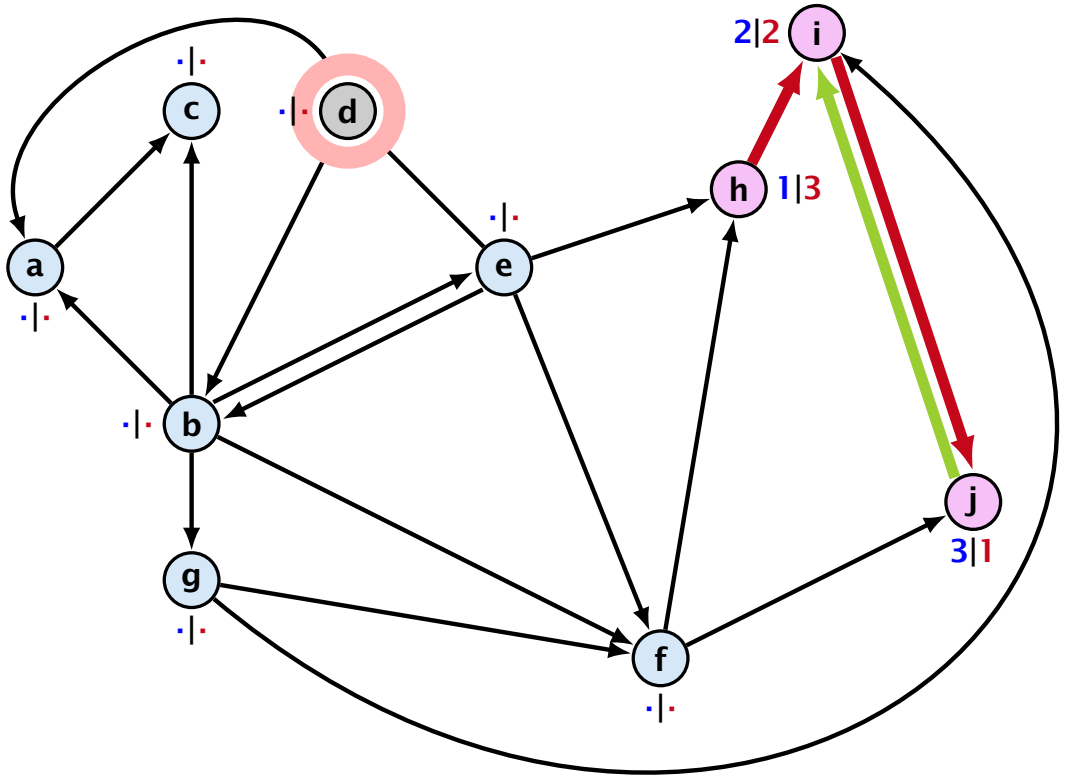
Vorwärtskante

Rückwärtskante

Kreuzkante



# DFS

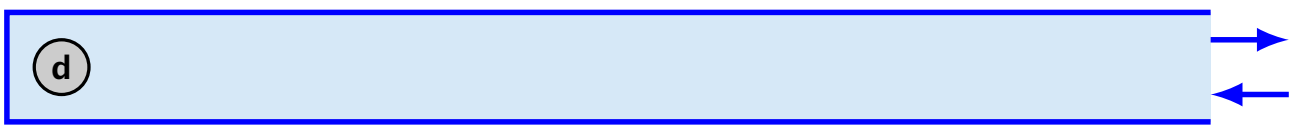



Baumkante

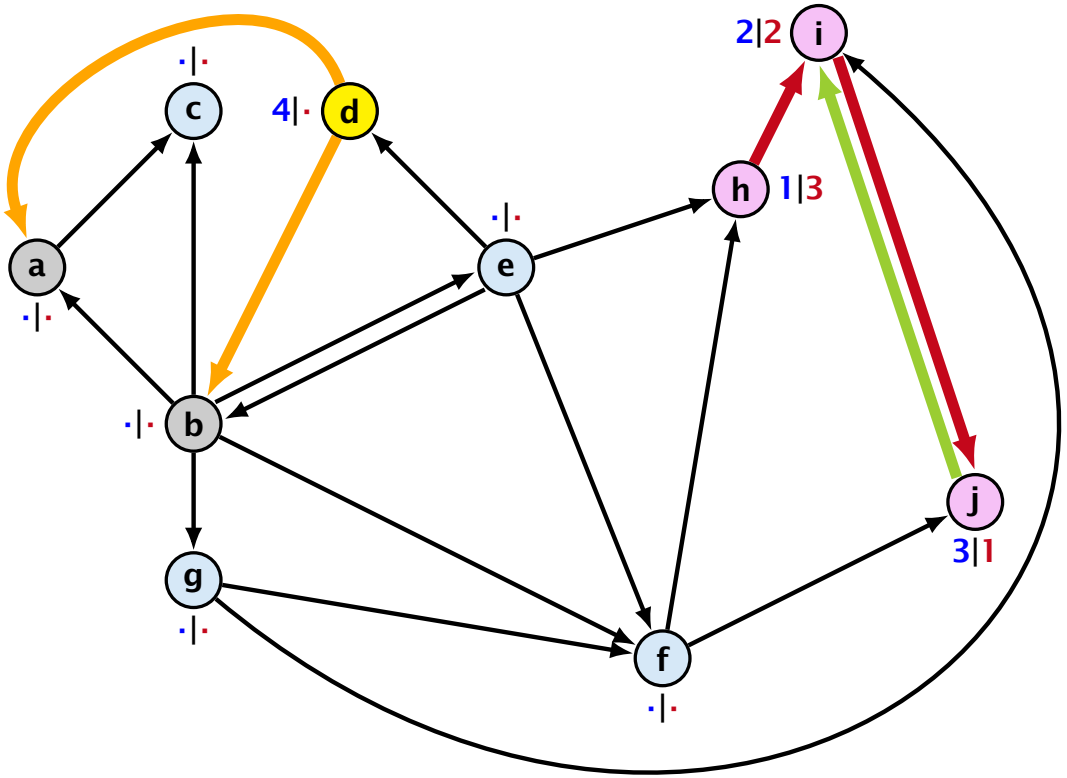
Vorwärtskante

Rückwärtskante

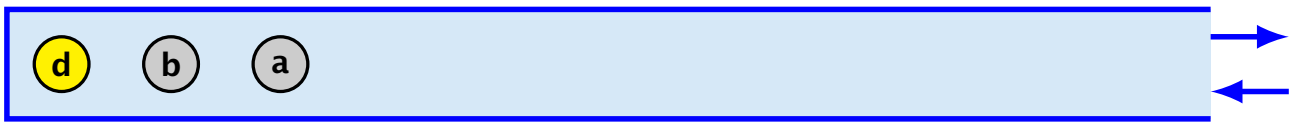
Kreuzkante



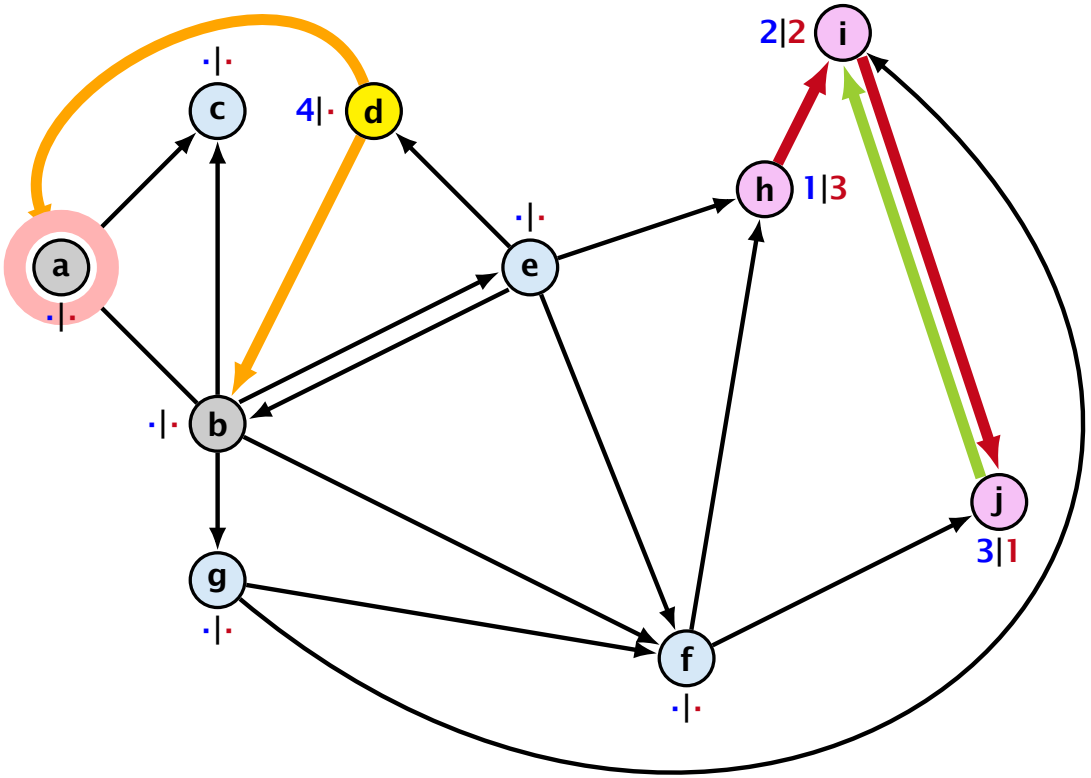
# DFS


2|2
i
·|·
c
4|·
d
h
1|3
·|·
a
·|·
e
·|·
b
j
3|1
g
·|·
f
·|·

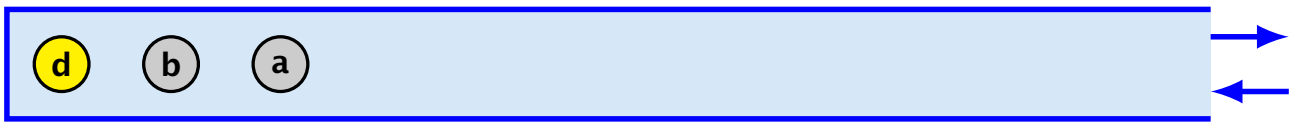
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a

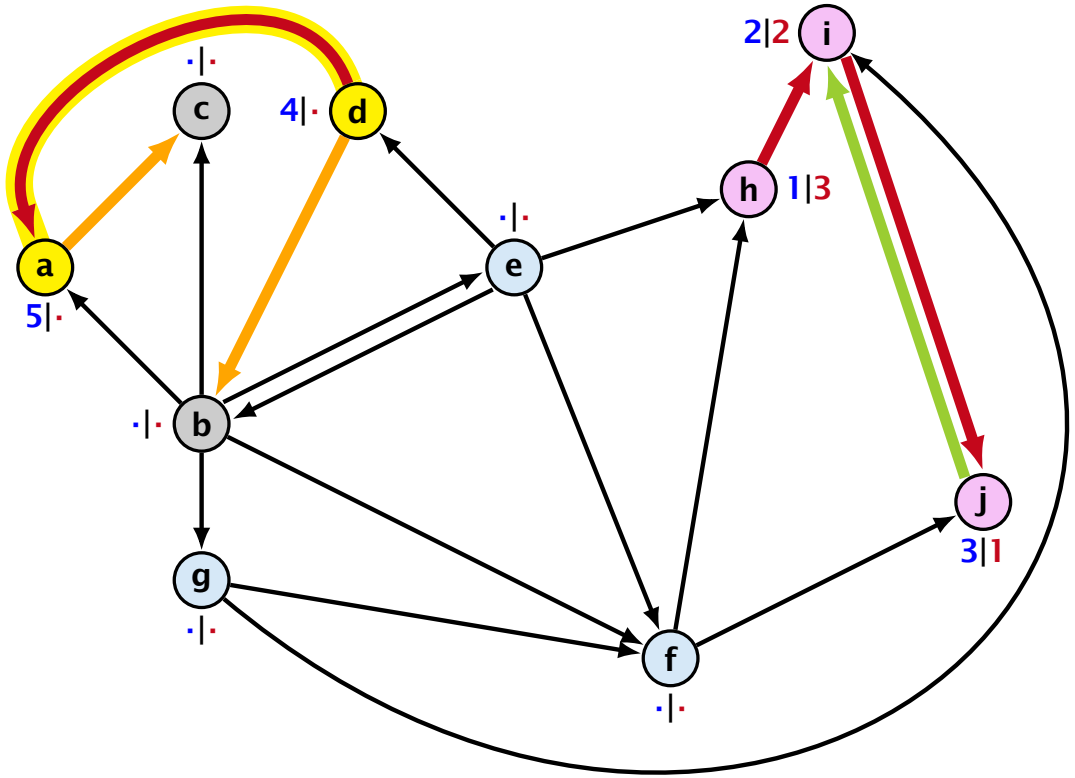
# DFS


2|2
i
·|·
c
4|·
d
h
1|3
·|·
a
·|·
e
·|·
b
j
3|1
g
·|·
f
·|·

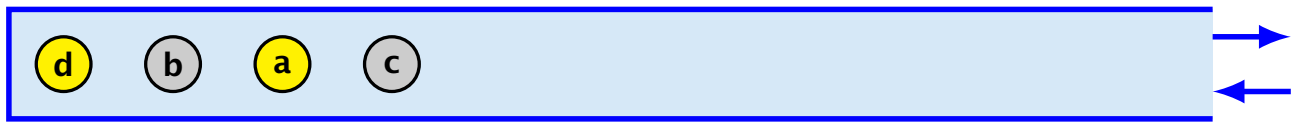
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a

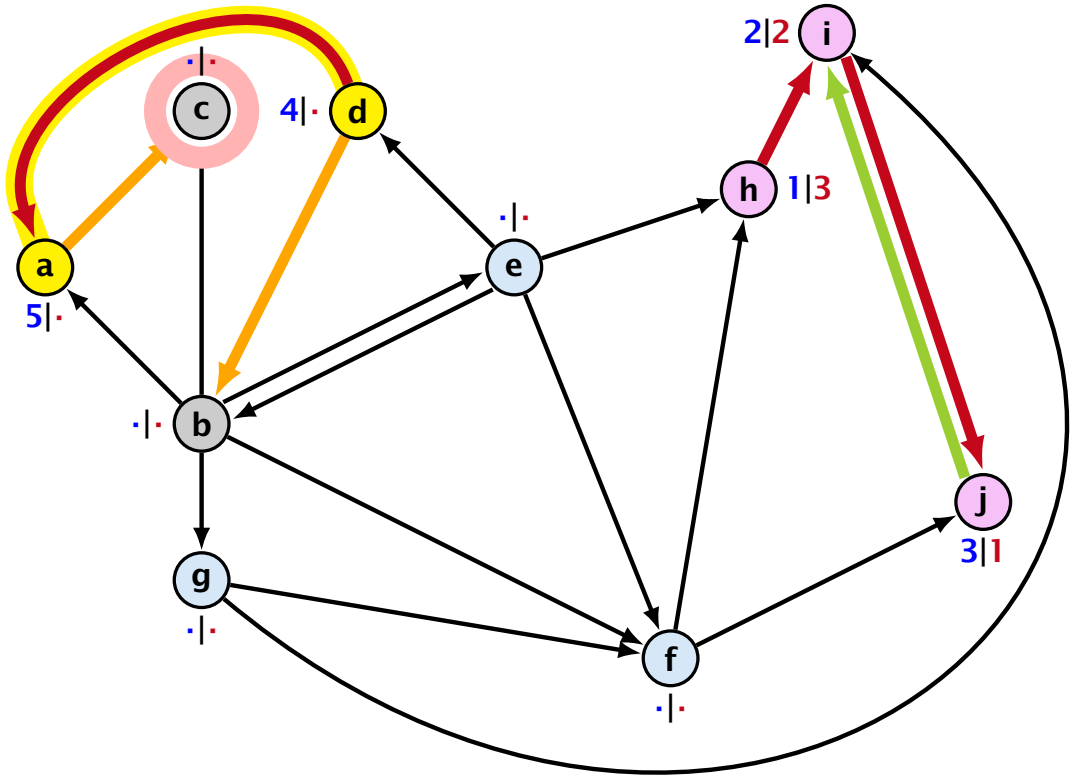
# DFS


2|2
i
·|·
c
4|·
d
h
1|3
·|·
a
e
5|·
·|·
b
j
3|1
g
·|·
f
·|·

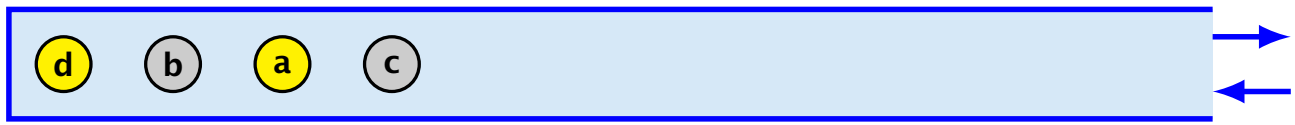
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a
c

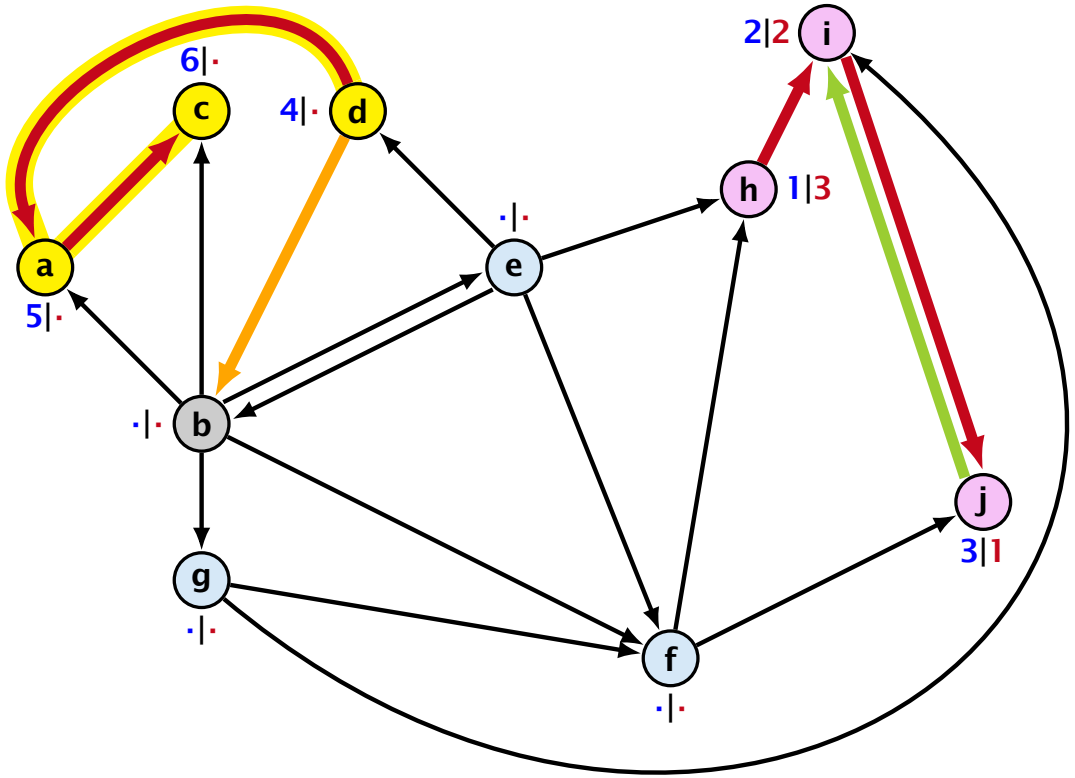
# DFS


2|2
i
·|·
c
4|·
d
h
1|3
·|·
a
e
5|·
·|·
b
j
3|1
g
·|·
f
·|·

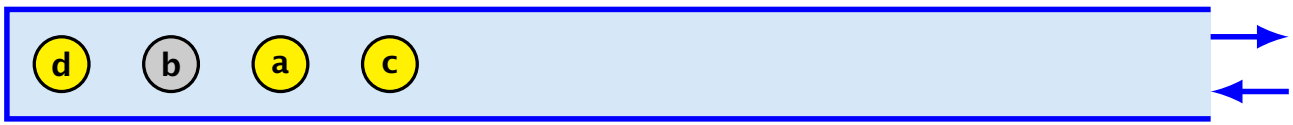
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a
c

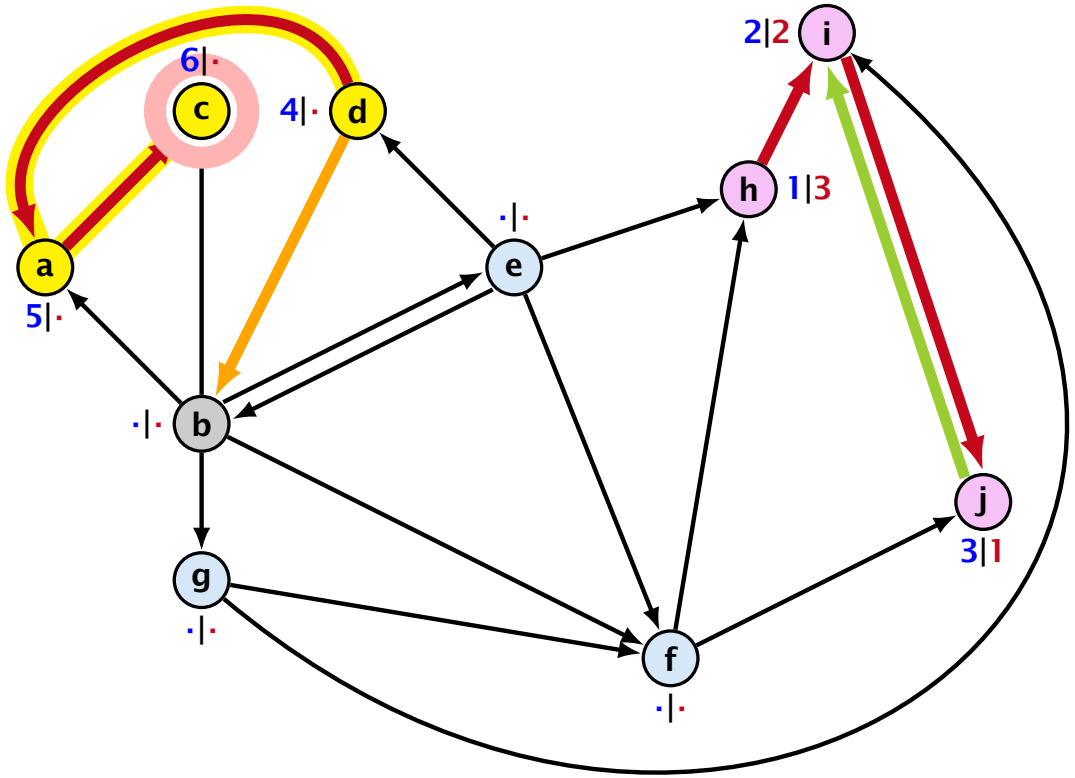
# DFS


6|·
c
4|·
d
2|2
i
h
1|3
·|·
a
e
5|·
·|·
b
j
3|1
g
·|·
f
·|·

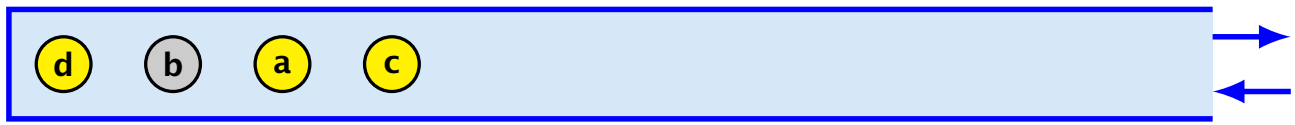
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a
c

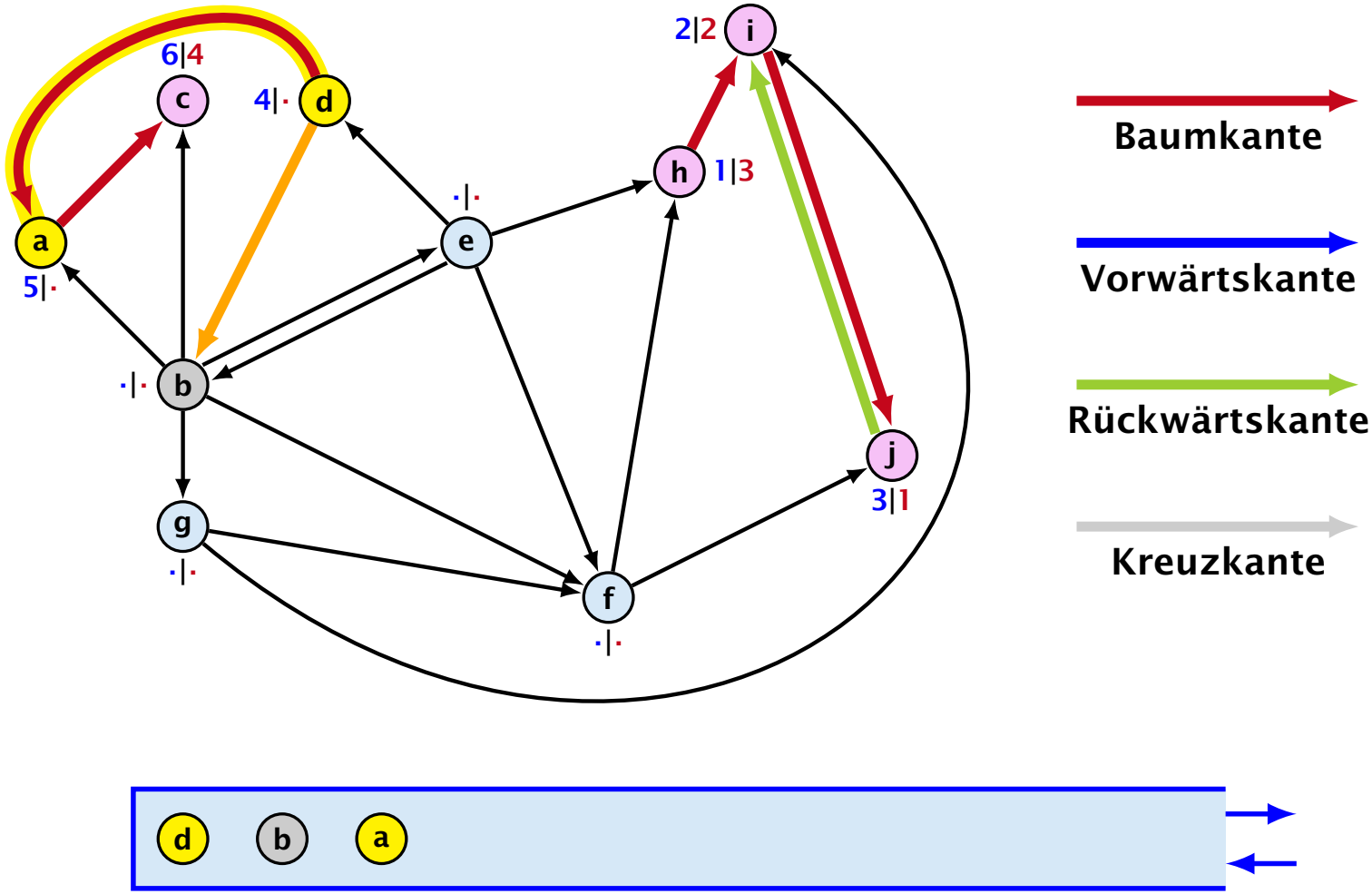
# DFS


6|·
c
4|·
d
2|2
i
h
1|3
·|·
e
a
5|·
·|·
b
g
·|·
f
·|·
j
3|1

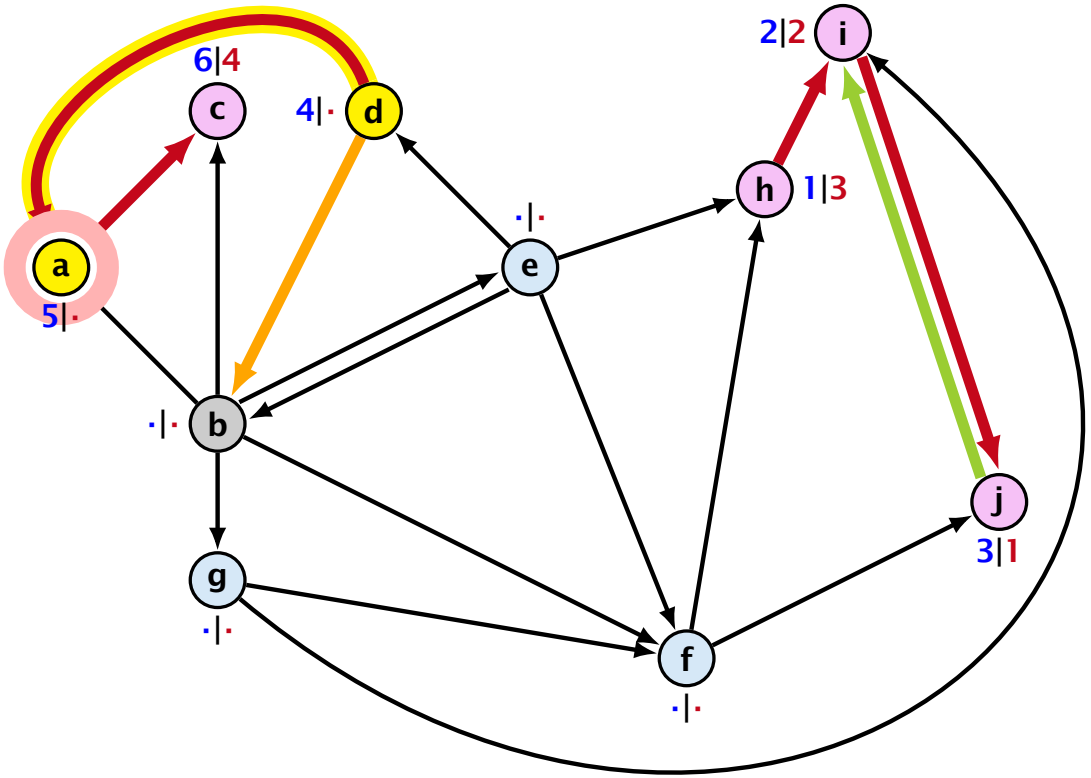
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a
c

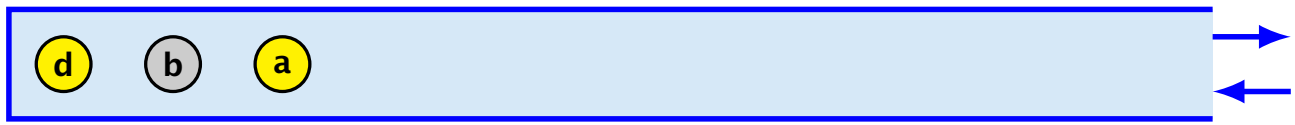
# DFS


6|4
c
4|·
d
2|2
i
h
1|3
·|·
a
5|·
e
·|·
b
j
3|1
g
·|·
f
·|·
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
d
b
a

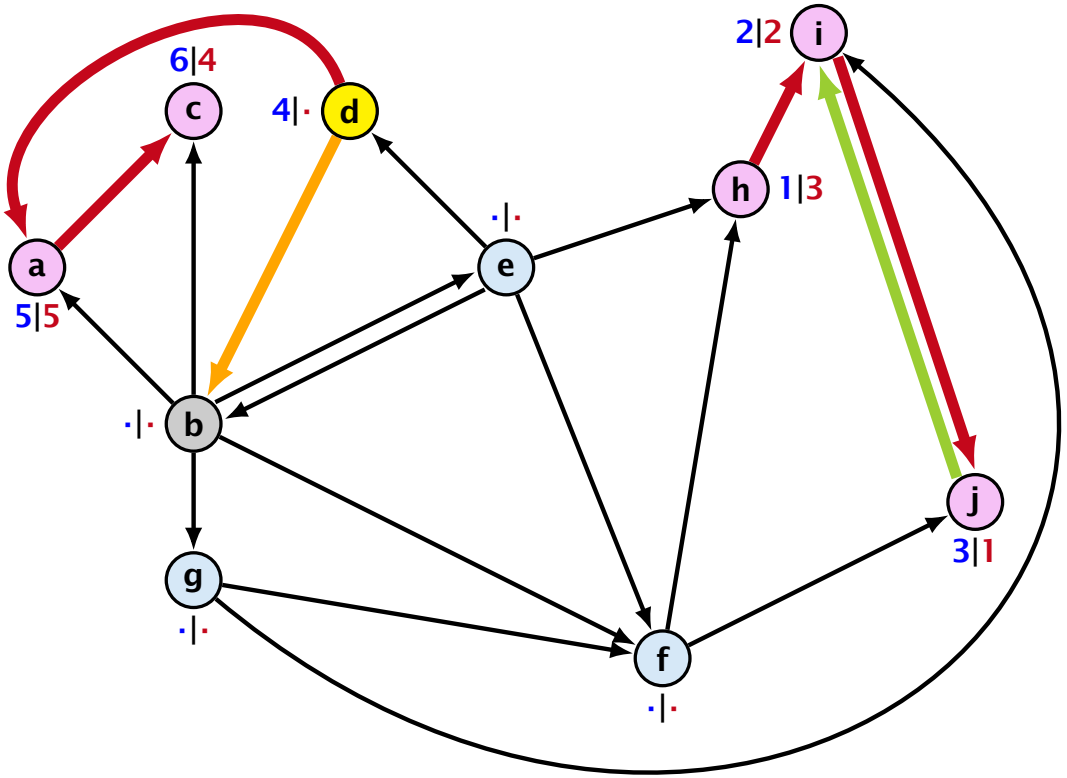
# DFS


6|4
c
4|·
d
2|2
i
h
1|3
·|·
e
a
5|·
·|·
b
g
·|·
f
·|·
j
3|1

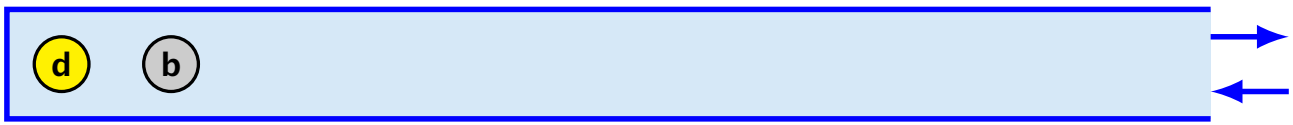
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
a

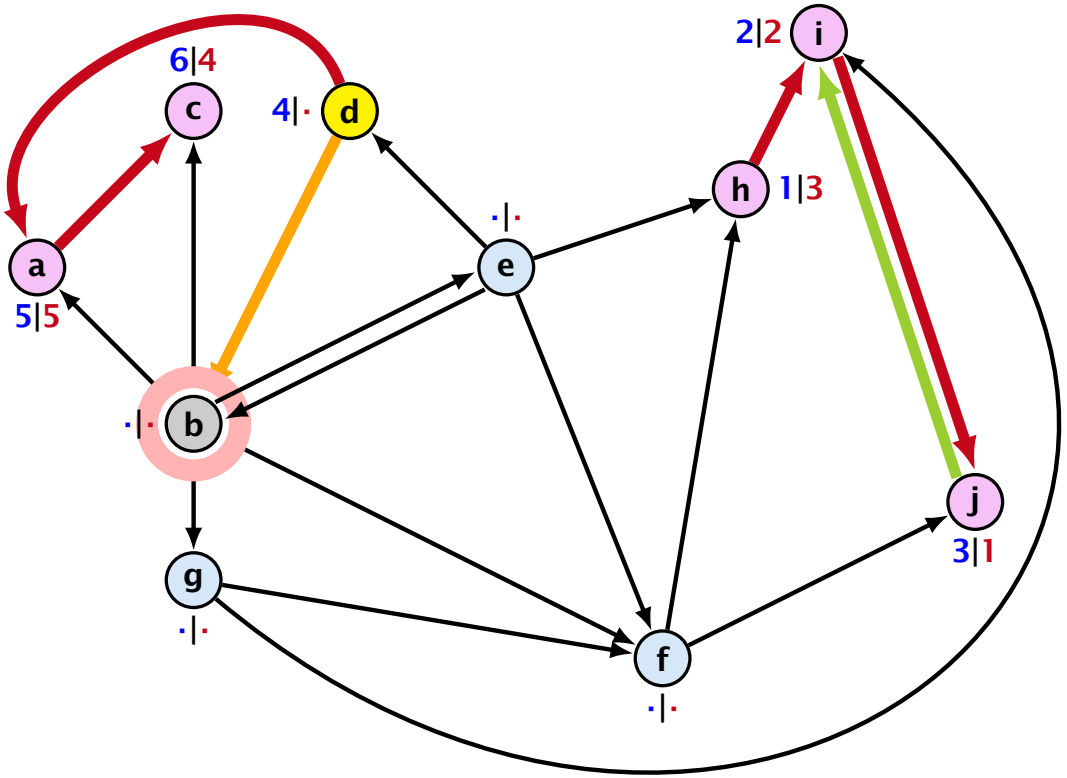
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
·|·
e
h
1|3
·|·
b
g
·|·
f
·|·
j
3|1

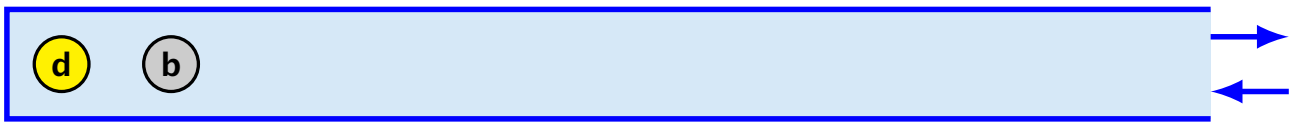
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b

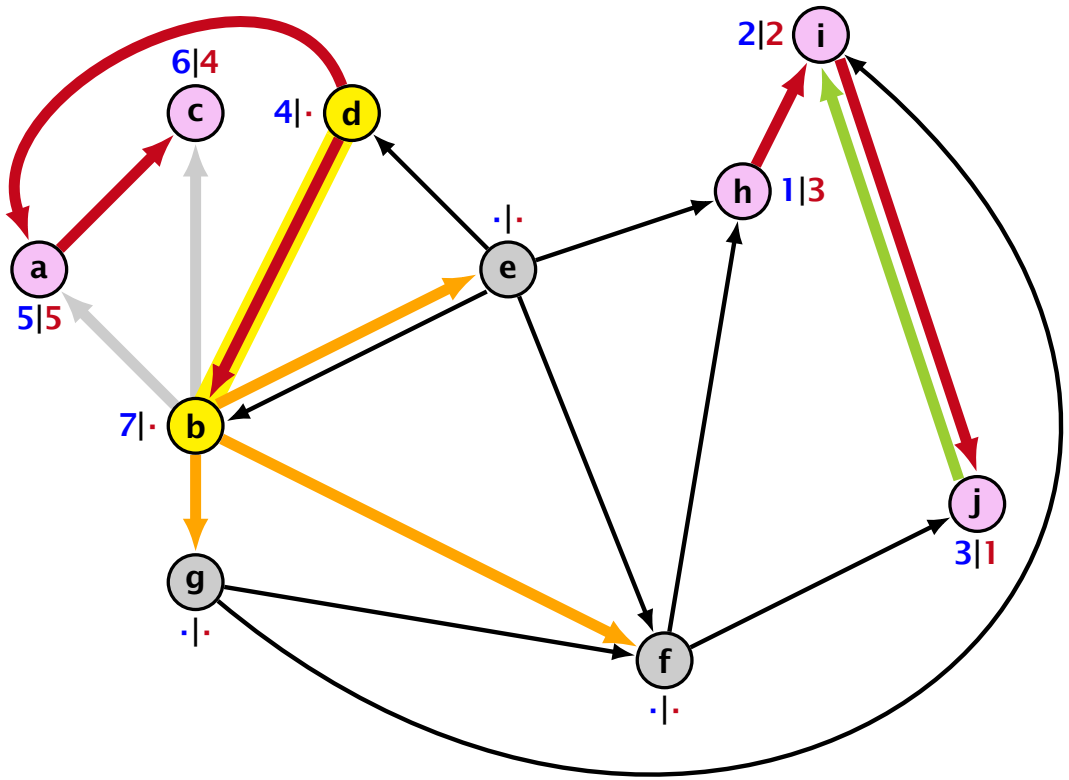
# DFS


6|4
c
4|·
d
2|2
i
h
1|3
·|·
a
e
5|5
·|·
b
g
·|·
f
·|·
j
3|1

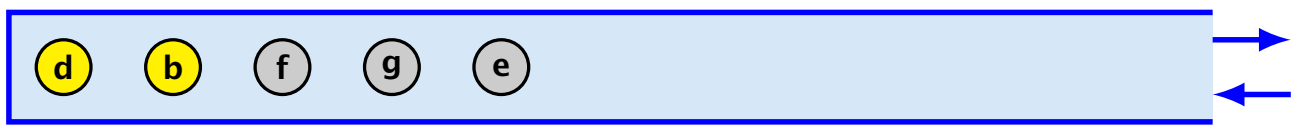
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b

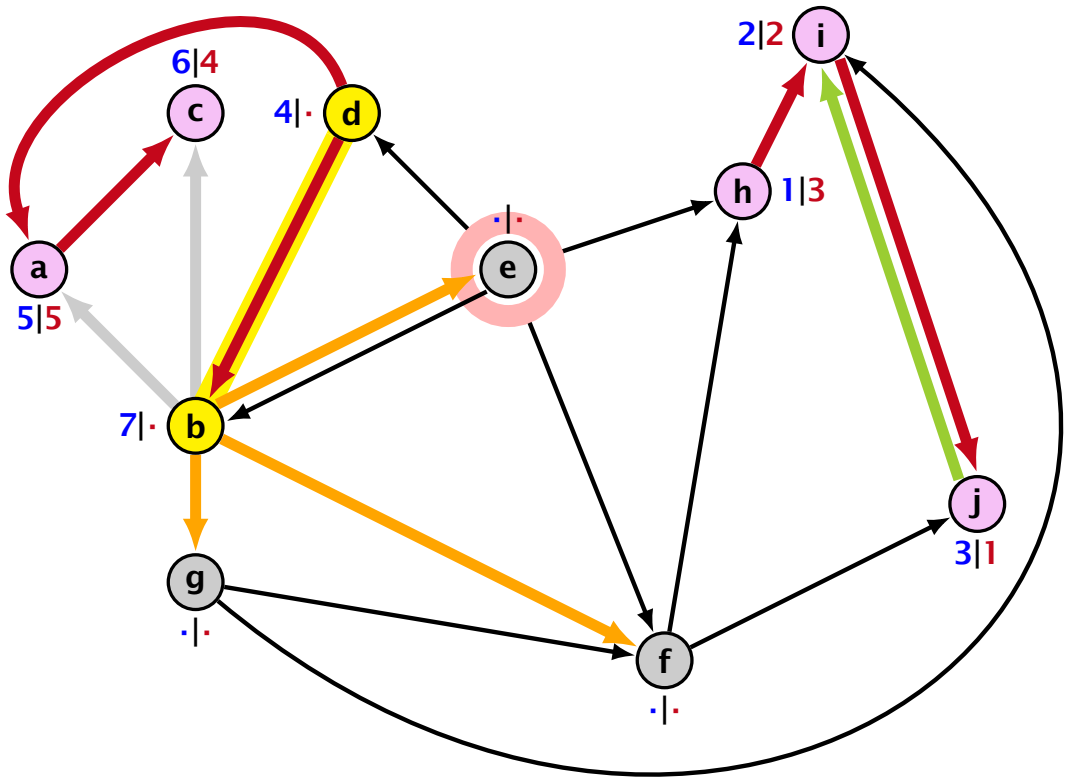
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
·|·
e
h
1|3
7|·
b
j
3|1
g
·|·
f
·|·

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g
e

# DFS

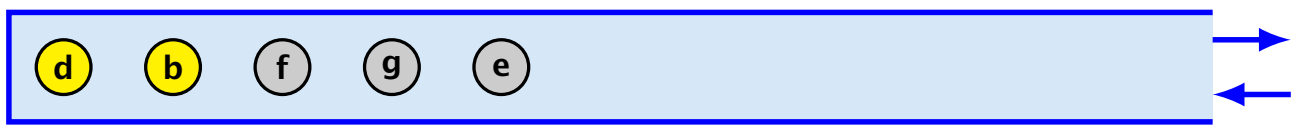



Baumkante

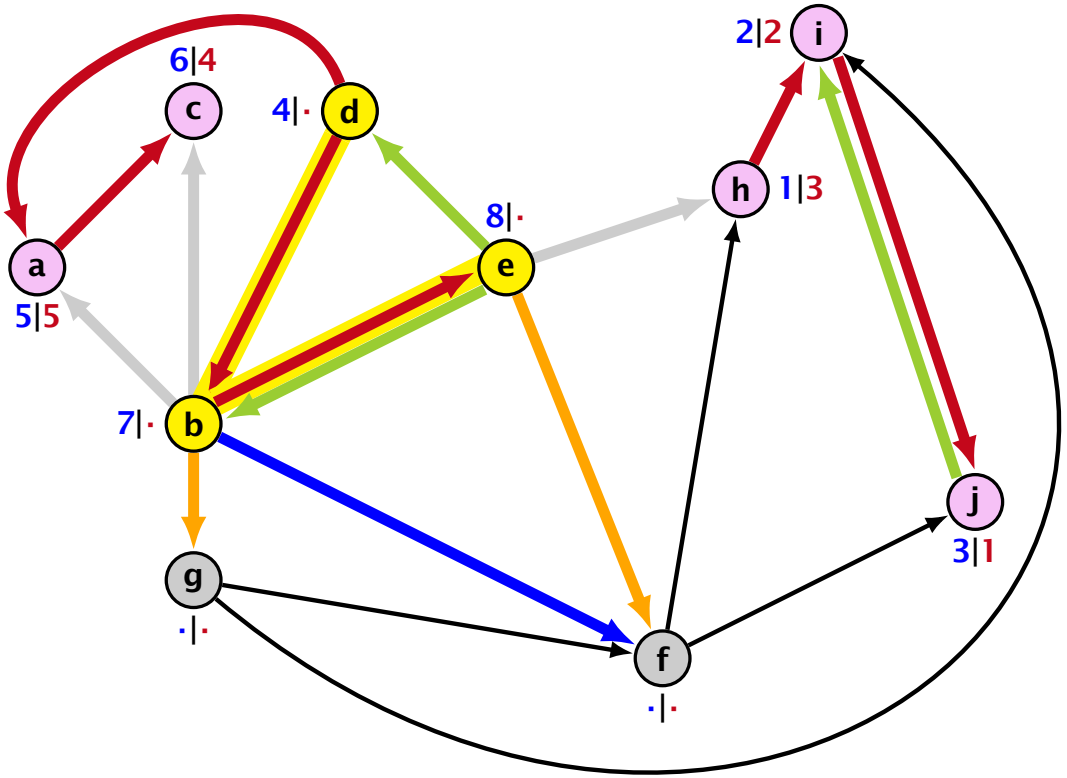
Vorwärtskante

Rückwärtskante

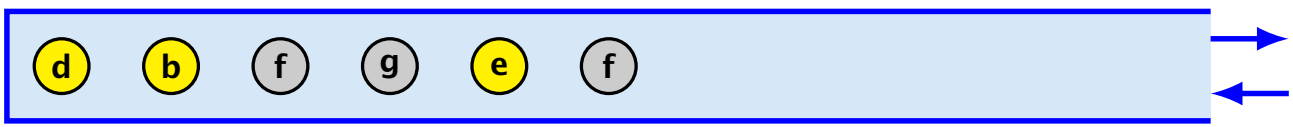
Kreuzkante



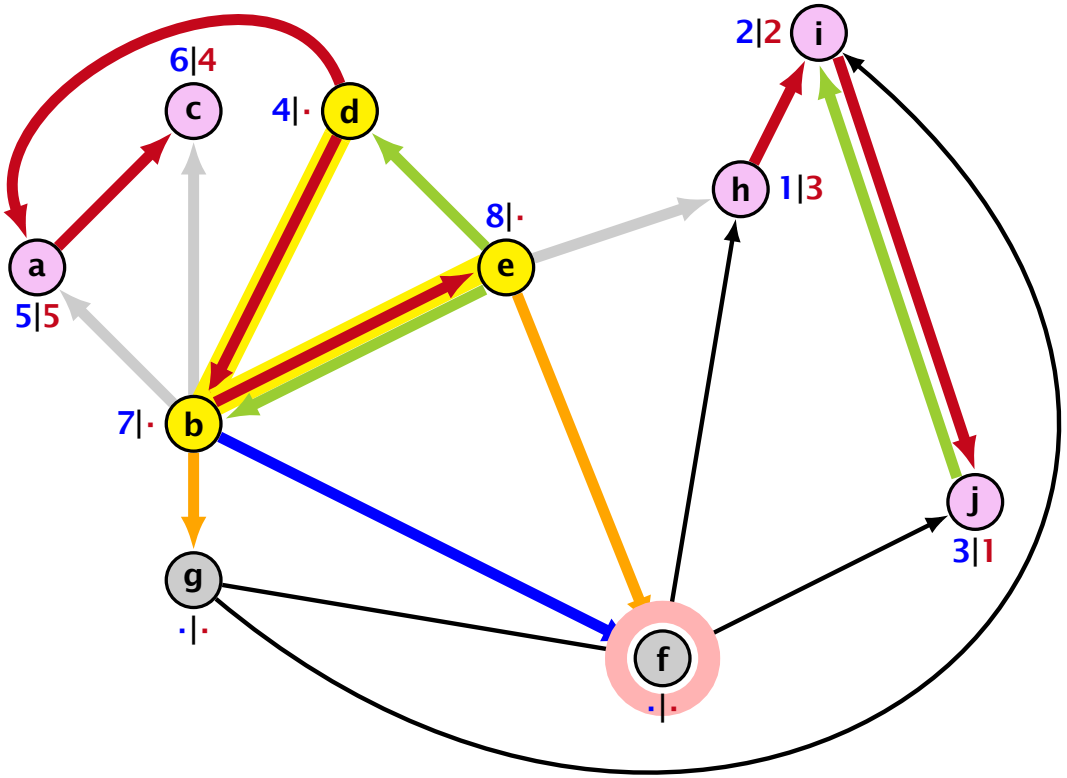
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|·
e
h
1|3
7|·
b
j
3|1
g
·|·
f
·|·

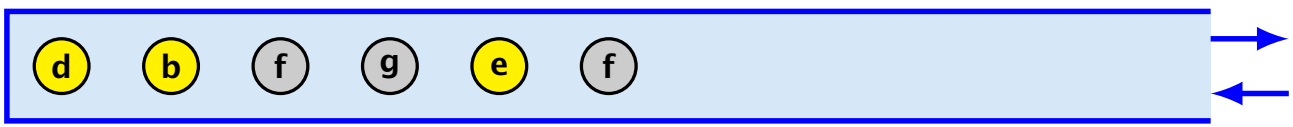
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g
e
f

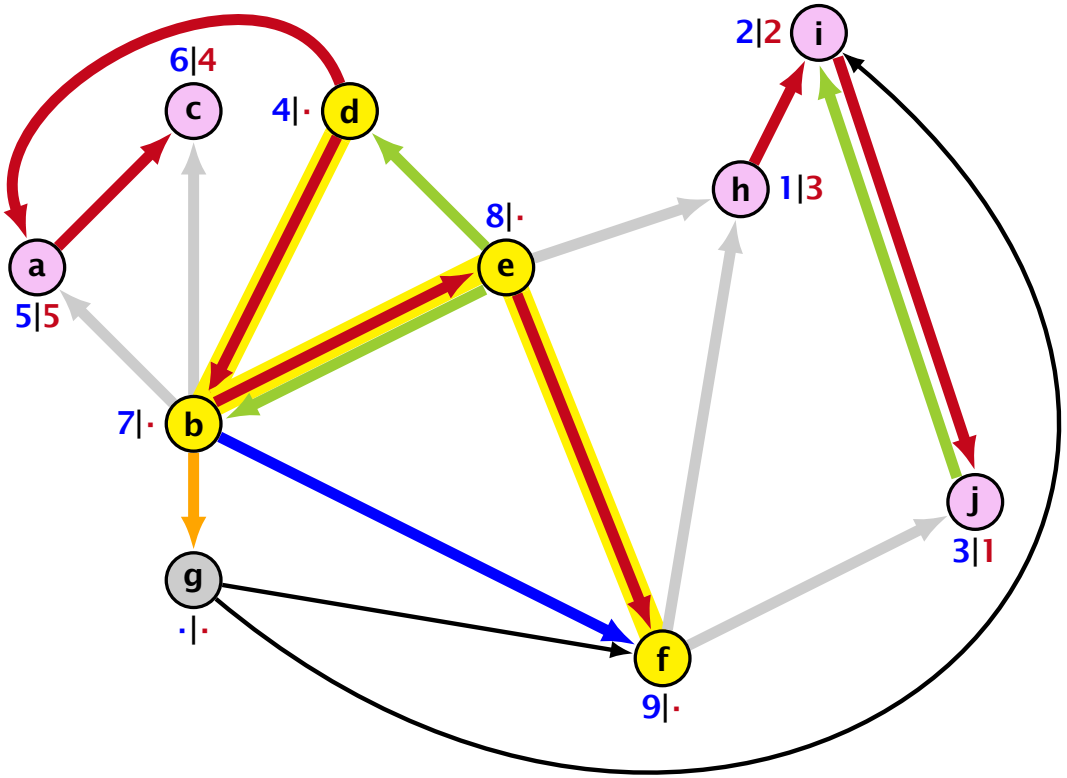
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|·
e
h
1|3
7|·
b
j
3|1
g
·|·
f
·|·

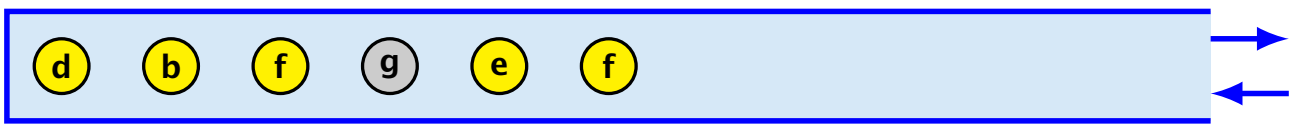
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g
e
f

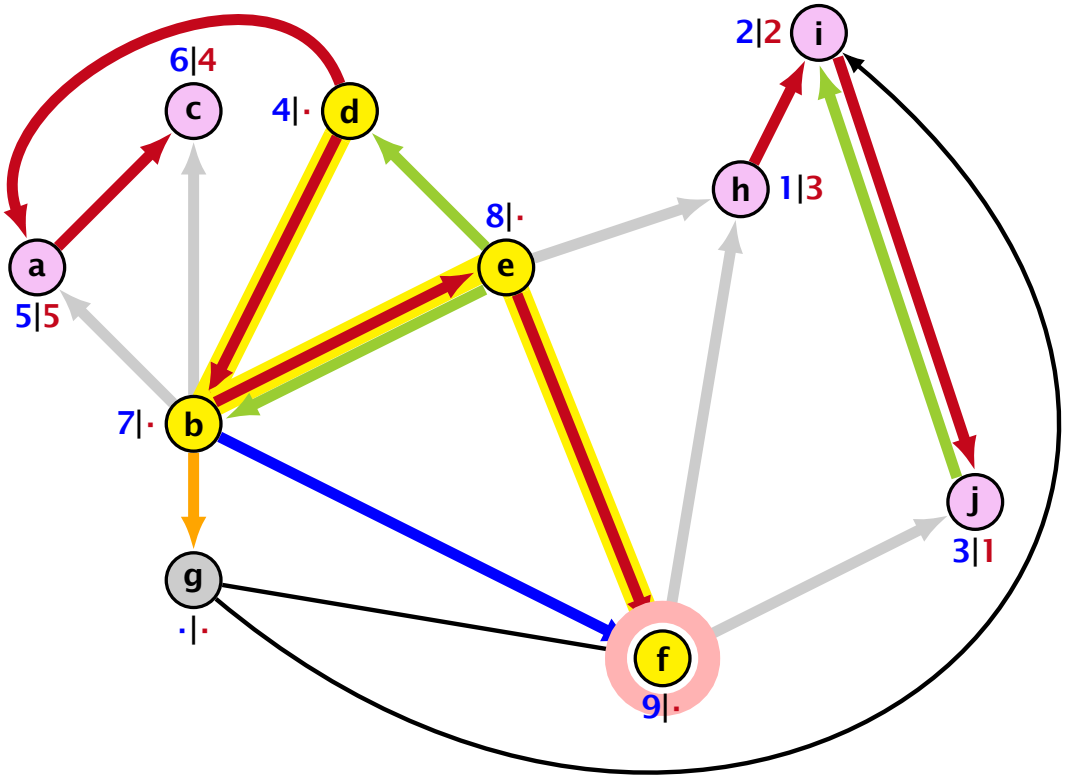
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|·
e
h
1|3
7|·
b
j
3|1
g
·|·
f
9|·

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g
e
f

# DFS

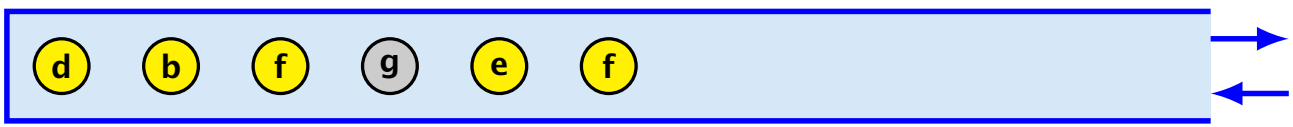



Baumkante

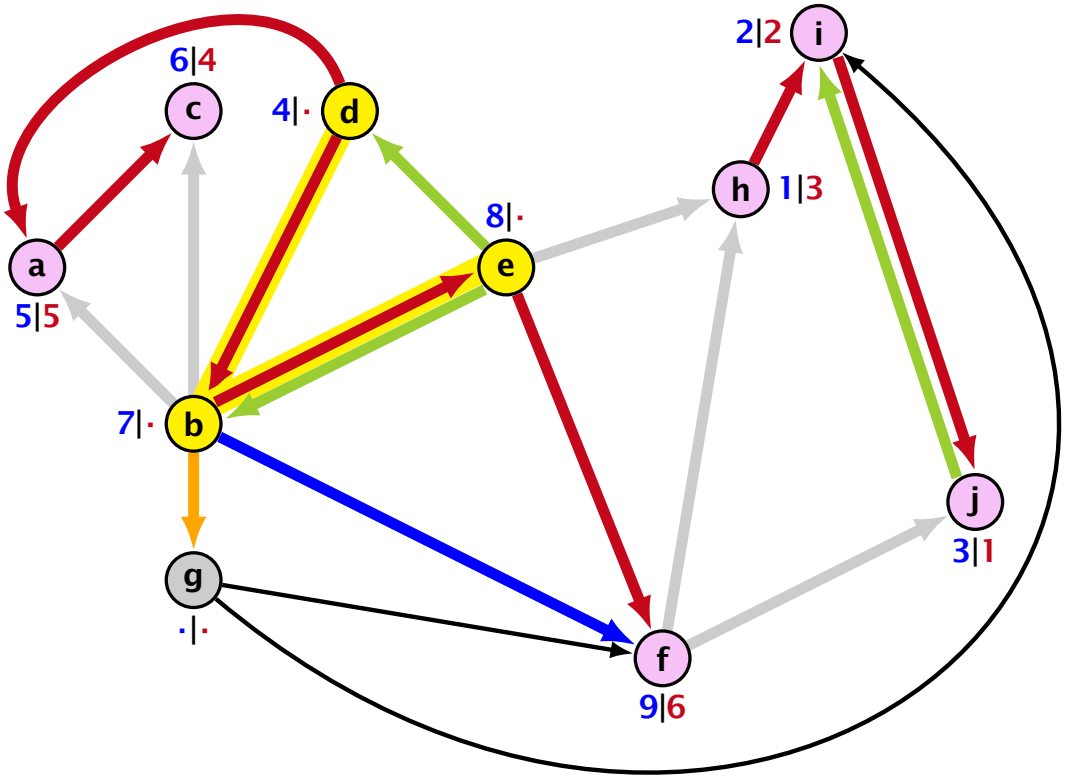
Vorwärtskante

Rückwärtskante

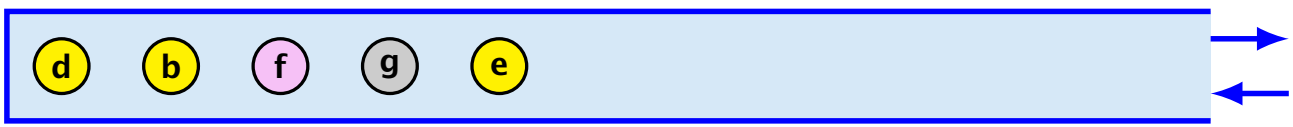
Kreuzkante



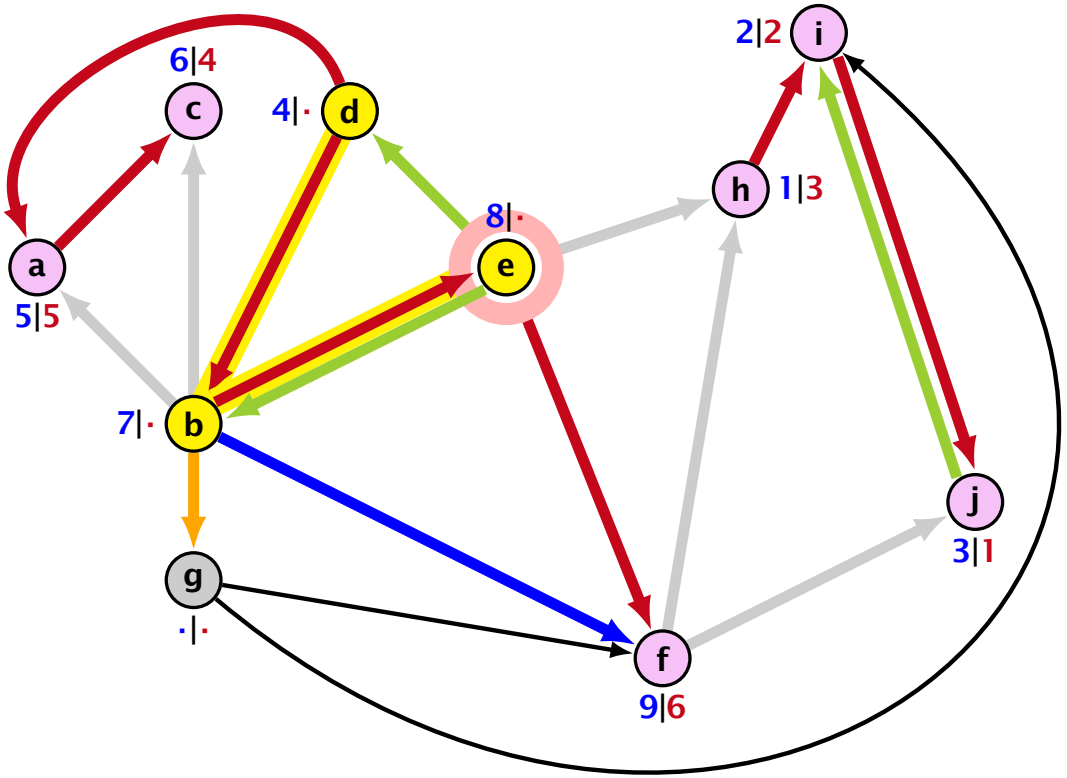
# DFS


6|4
c
4|·
d
2|2
i
h
1|3
8|·
e
a
5|5
7|·
b
j
3|1
g
·|·
f
9|6

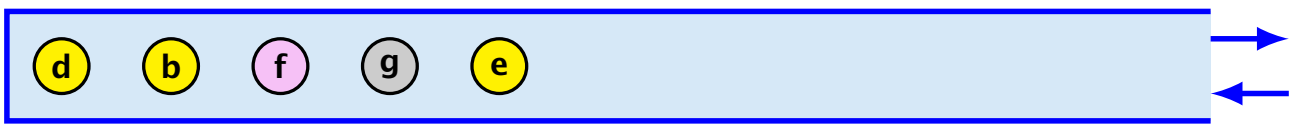
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g
e

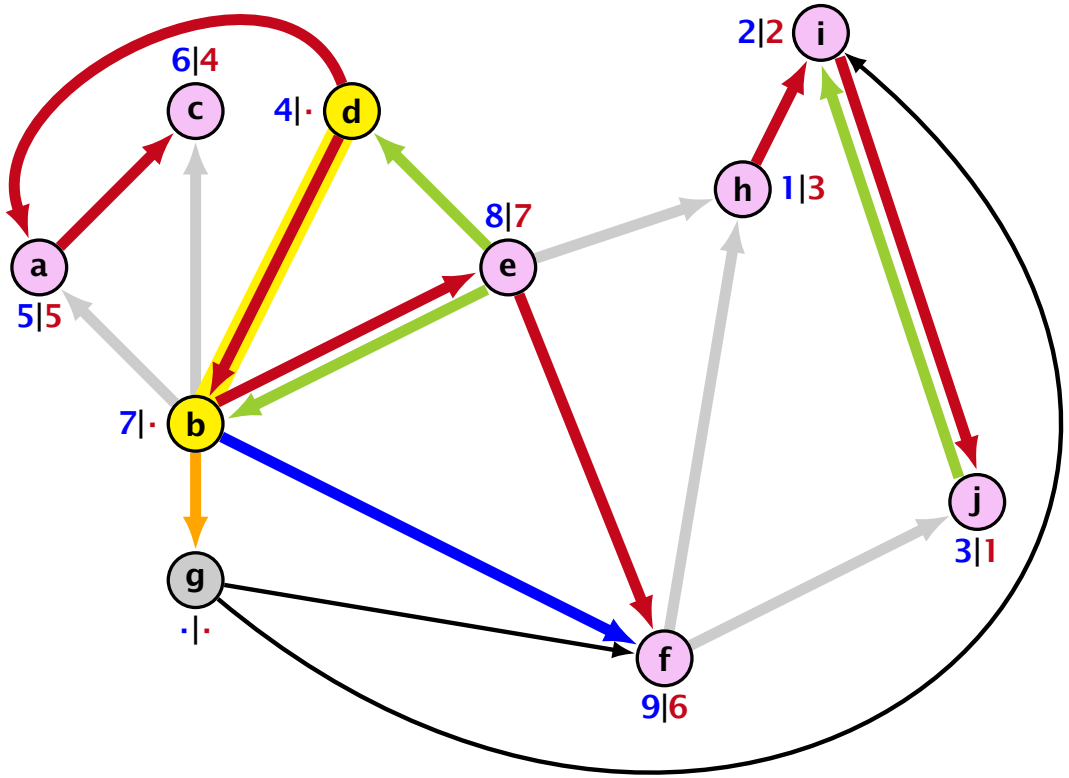
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|·
e
h
1|3
7|·
b
j
3|1
g
·|·
f
9|6

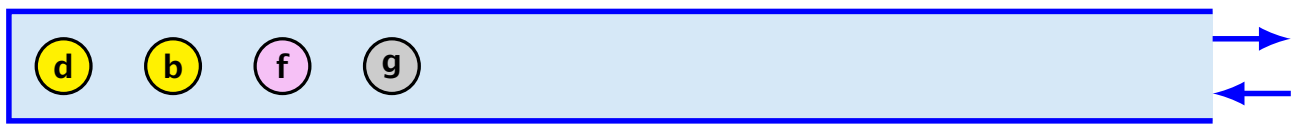
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g
e

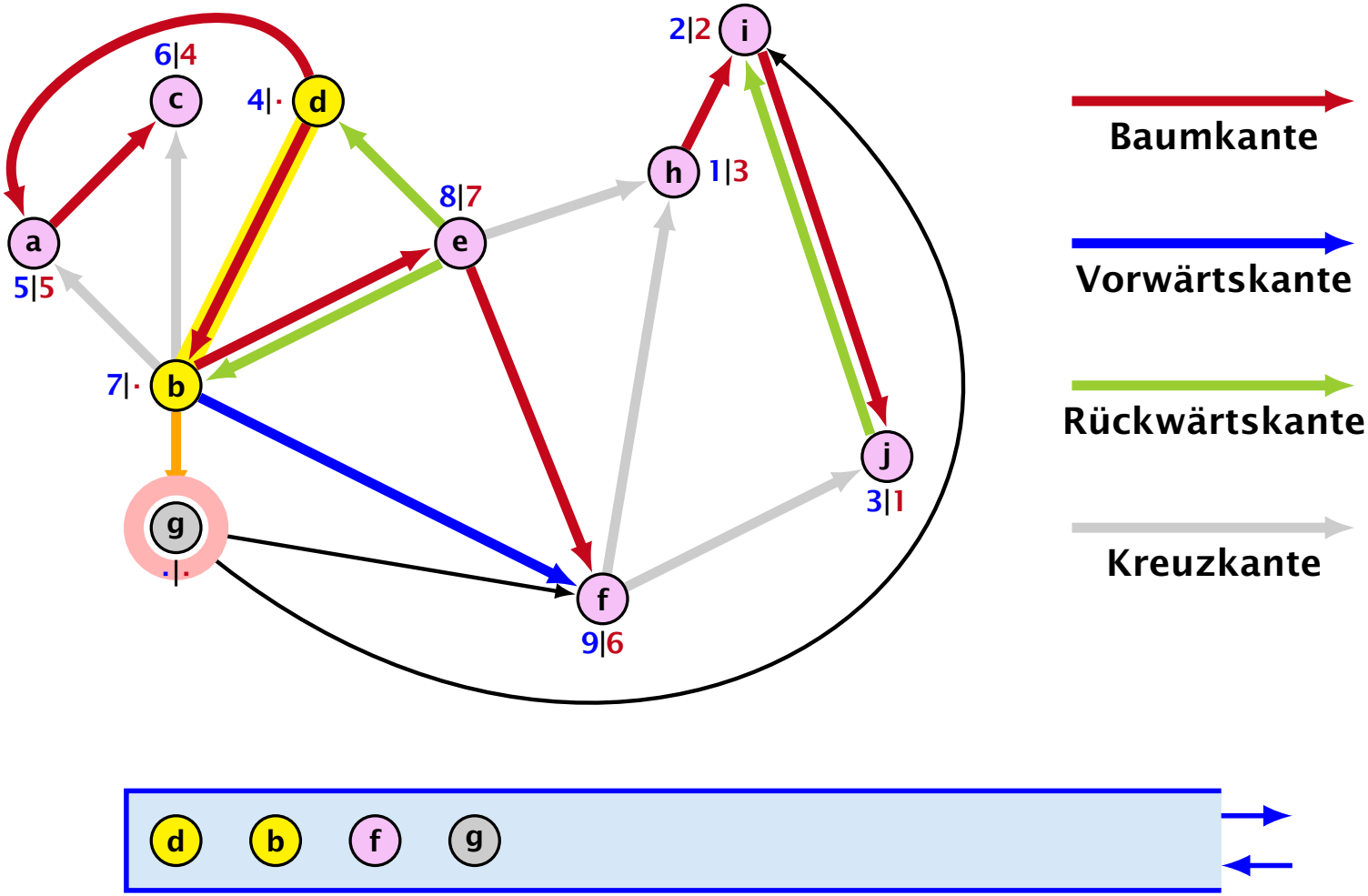
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
·|·
f
9|6

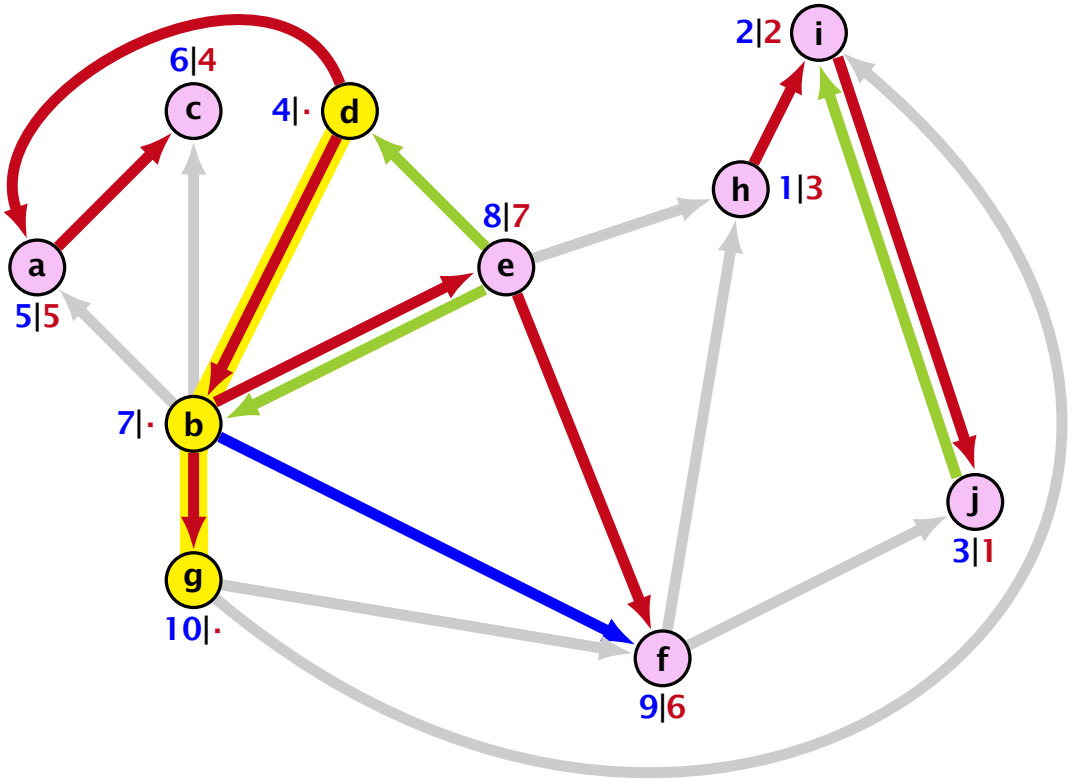
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g

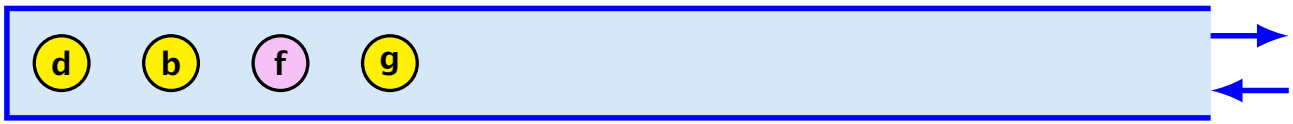
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
·|·
f
9|6
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
d
b
f
g

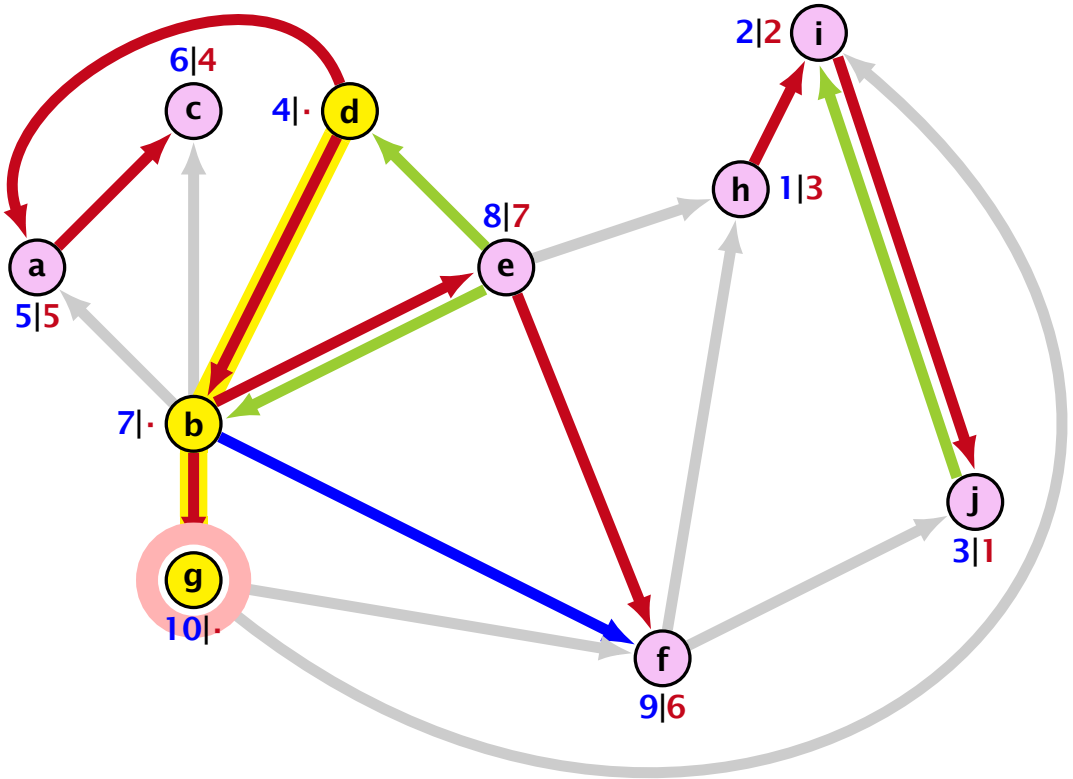
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
10|·
f
9|6

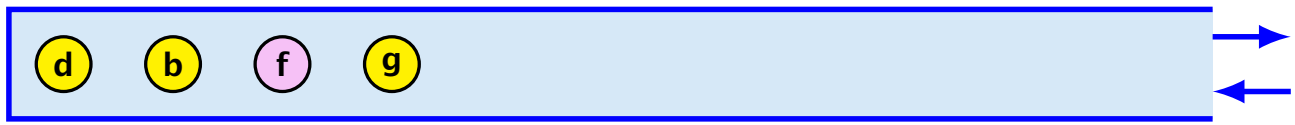
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g

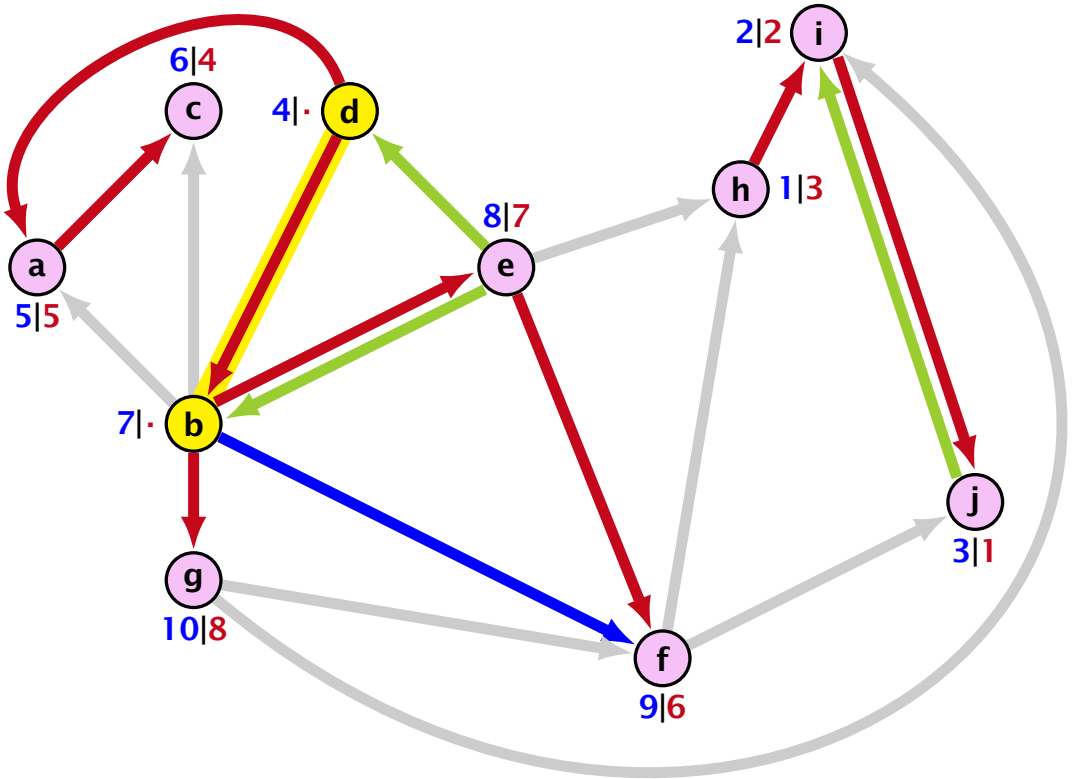
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
10|·
f
9|6

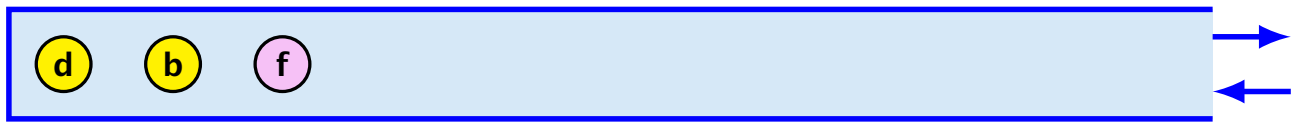
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f
g

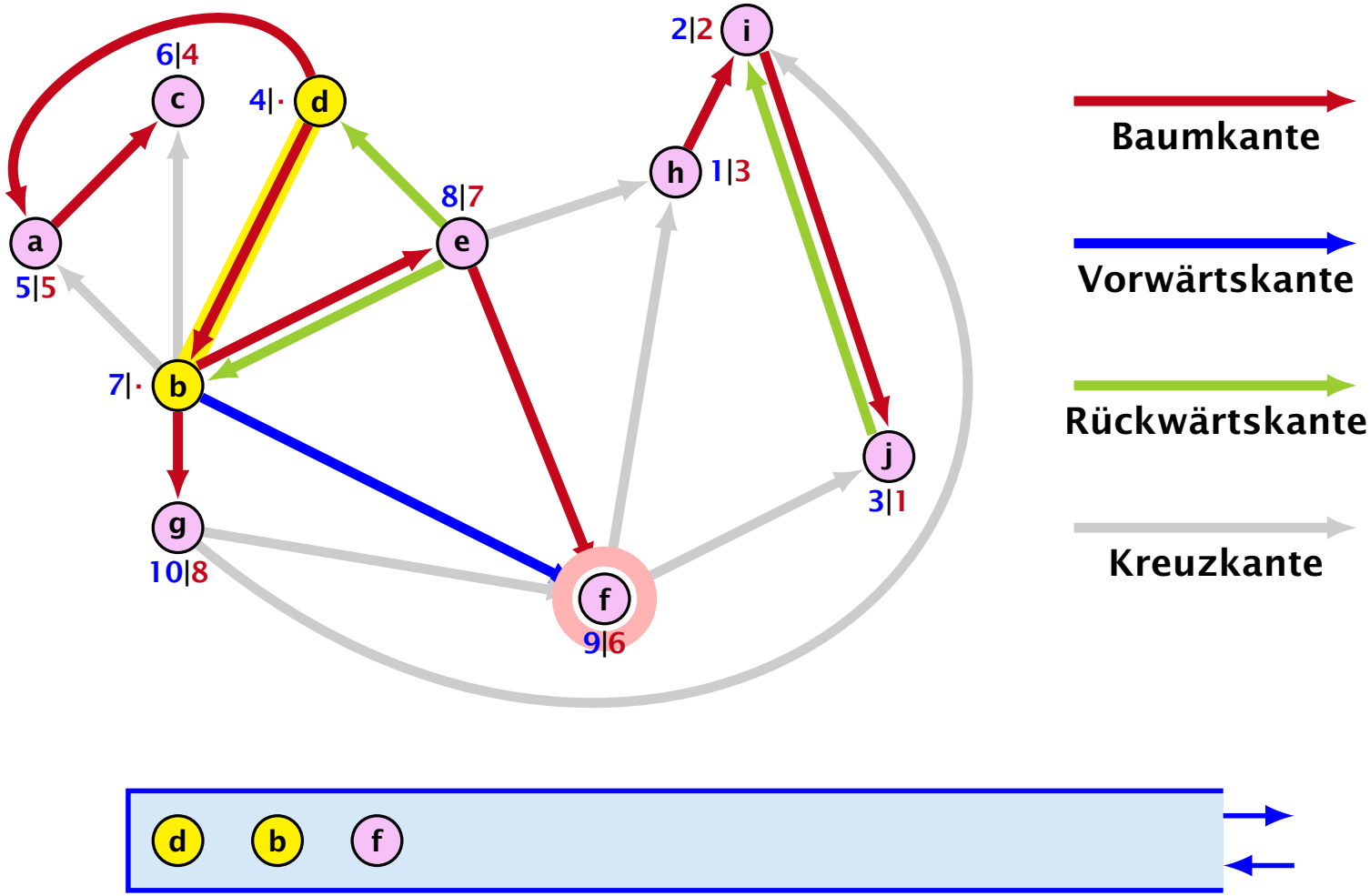
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
10|8
f
9|6

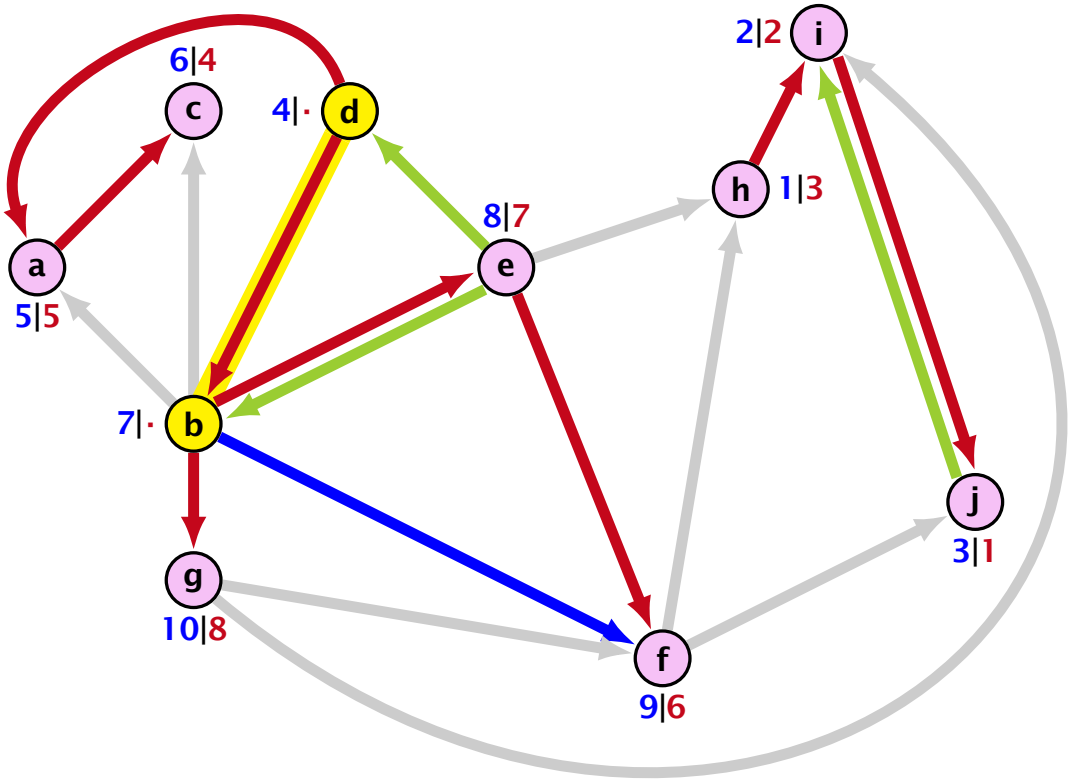
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b
f

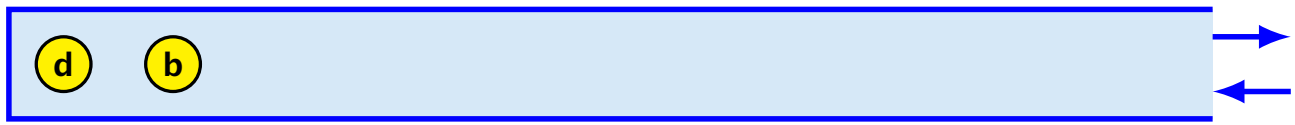
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
10|8
f
9|6
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
d
b
f

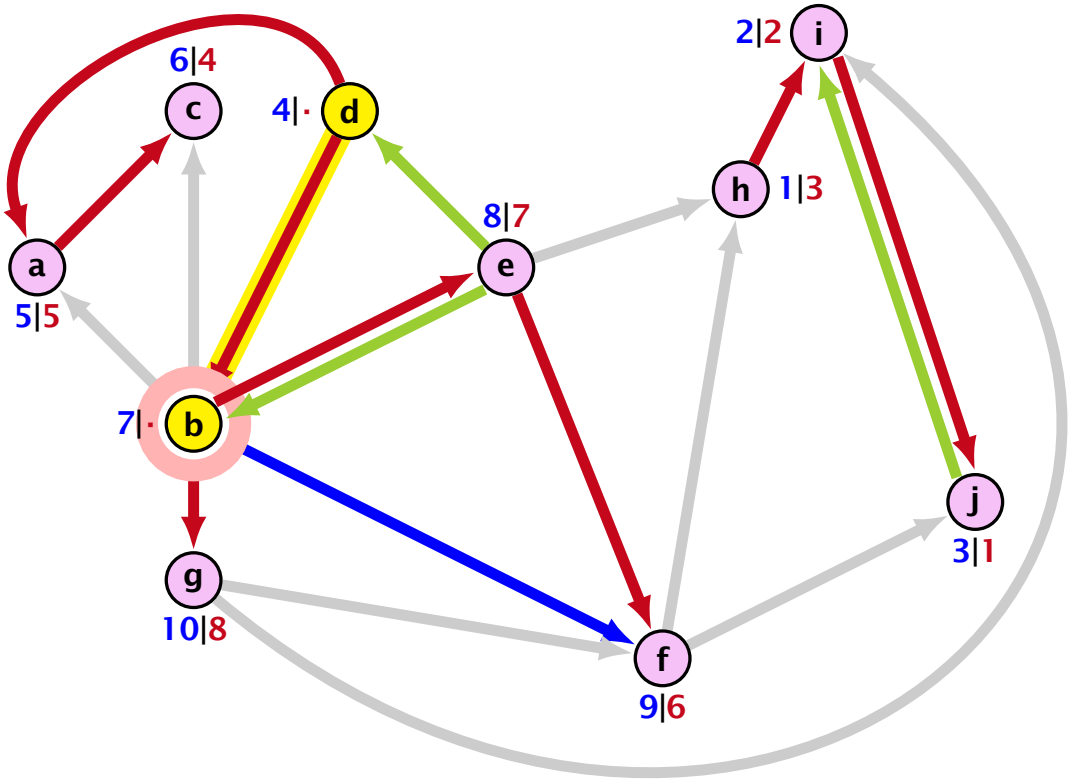
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
10|8
f
9|6

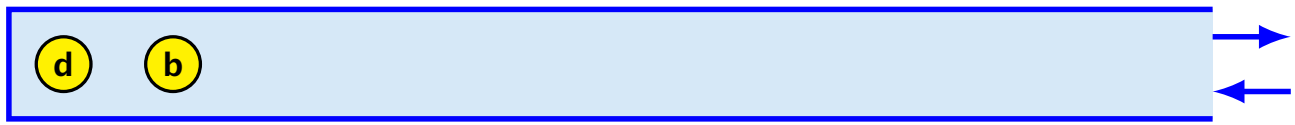
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
b

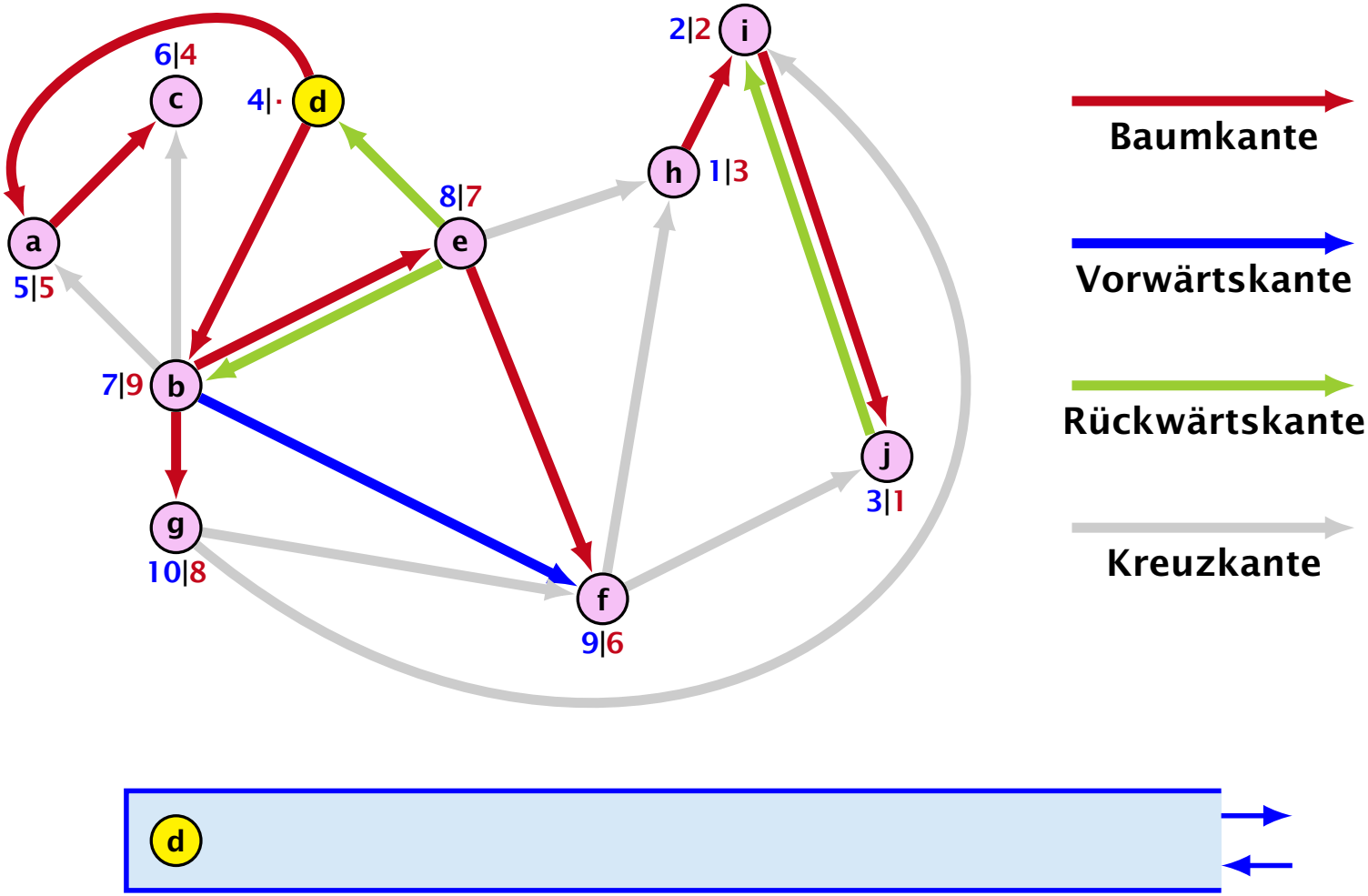
# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|·
b
j
3|1
g
10|8
f
9|6

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

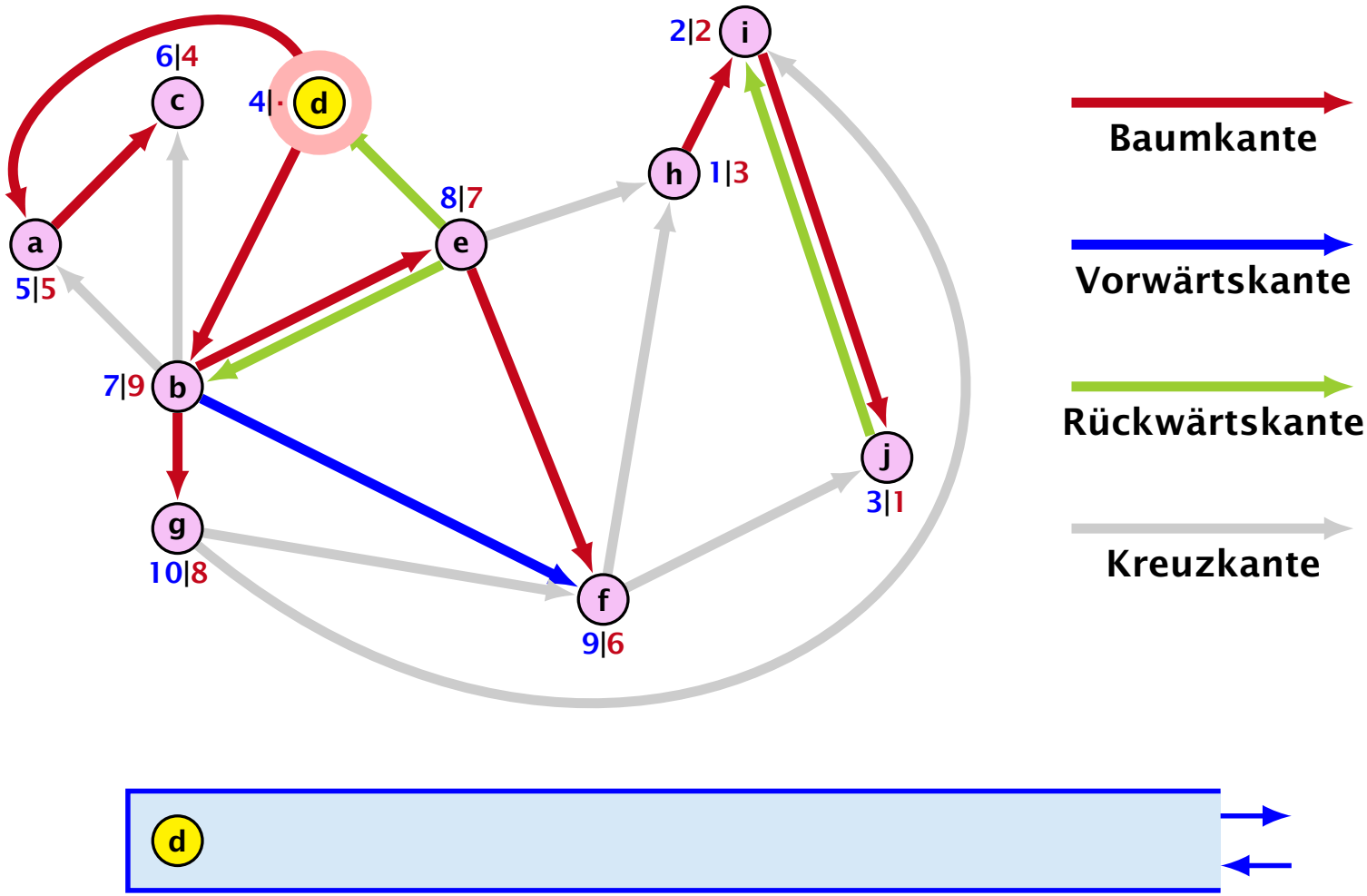

d
b

# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|9
b
j
3|1
g
10|8
f
9|6
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

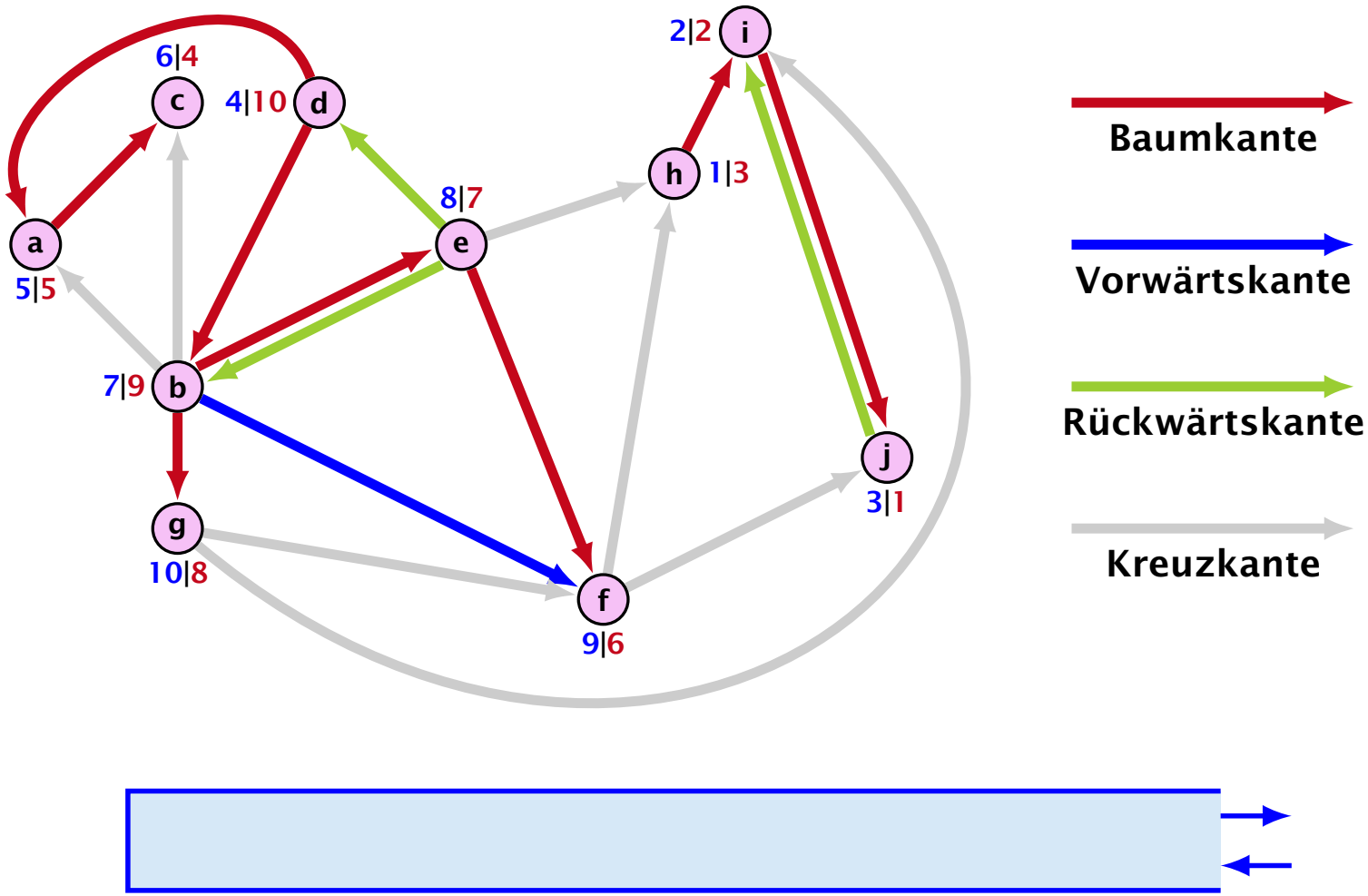

d

# DFS


6|4
c
4|·
d
2|2
i
a
5|5
8|7
e
h
1|3
7|9
b
j
3|1
g
10|8
f
9|6
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d

# DFS


a
5|5
b
7|9
c
6|4
d
4|10
e
8|7
f
9|6
g
10|8
h
1|3
i
2|2
j
3|1
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

# Tiefensuche

## Kantentypen

- **Baumkanten**
  Für jeden Nichtwurzelknoten $v$, speichert man den Vorgänger über den $v$ „betreten“ wurde. Diese Kanten erzeugen einen Wald.
  Die Bäume sind von der Wurzel zu den Blättern gerichtet.
- **Vorwärtskanten**
  Falls für Kante $(x, y)$ ein Pfad von $x$ nach $y$ in einem Baum existiert (und $(x, y)$ nicht Baumkante).
- **Rückwärtskanten**
  Falls für Kante $(x, y)$ ein Pfad von $y$ nach $x$ in einem Baum existiert.
- **Kreuzkanten**
  Alle anderen Kanten.

# Tiefensuche

Eigenschaften der DFS-Nummerierung einer Kante `(x,y)`:

**Baumkante** oder **Vorwärtskante**

`x->dfsNum < y->dfsNum` und `x->finNum > y->finNum`

**Rückwärtskante**

`x->dfsNum > y->dfsNum` und `x->finNum < y->finNum`

**Kreuzkante**

`x->dfsNum > y->dfsNum` und `x->finNum > y->finNum`

Es kann keine Kanten mit `x->dfsNum < y->dfsNum` und `x->finNum < y->finNum` geben.

# Komplexität der Tiefensuche

**Rekursive Implementierung**

- erste Schleife in DFS wird genau $|V|$ mal aufgerufen
- DFSvisit(v) wird für jeden Knoten genau einmal aufgerufen
- Schleife in DFSvisit(v) wird $|N[v]|$ mal aufgerufen;

$$\sum_{v \in V} |N[v]| \leq 2|E| = \Theta(|E|)$$

- Gesamtlaufzeit: $\Theta(|V| + |E|)$

Komplexität der Stackimplementierung ist asymptotisch gleich.

# Tiefensuche – Anwendungen

**Test auf (starken) Zusammenhang in Graphen.**

- rufe `DFSvisit(v)` nur für einen Startknoten auf
- falls danach nicht alle Knoten besucht, ist Graph nicht (stark) zusammenhängend

**Test auf Zyklenfreiheit.**

- Graph hat Zyklus genau dann wenn die Tiefensuche eine Rückwärtskante enthält
- $\Leftarrow$ wenn man eine Rückwärtskante hat wir mit dieser Rückwärtskante über die Baumkanten ein Zyklus geschlossen
- $\Longrightarrow$ angenommen man hat einen Zyklus und die Tiefensuche liefert keine Rückwärtskante; dann ist `finNum` entlang des Zyklus steigend, da dieser nur aus Vorwärtskanten, Baumkanten, und Kreuzkanten besteht (Widerspruch)

# Breitensuche

**Anwendung:**
Bestimmung der Entfernung zum Startknoten in einem ungewichteten Graphen.

# Breitensuche

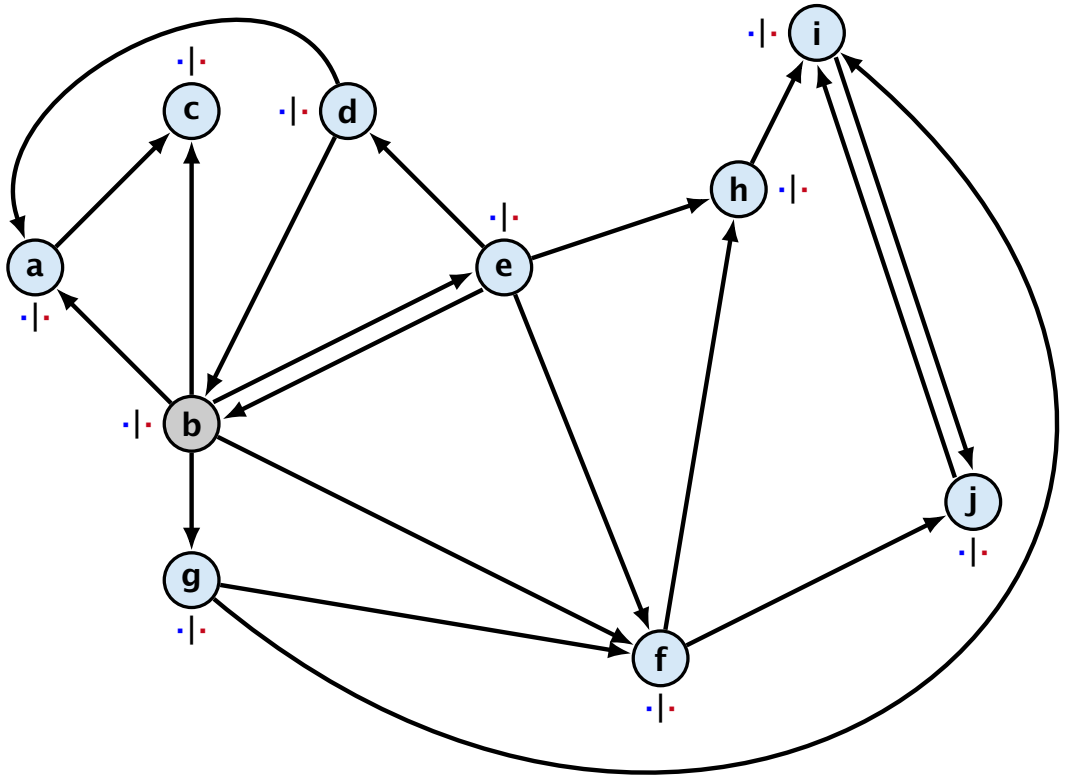
```
1 BFS()
2     bfsCount = 1;
3     finCount = 1;
4     foreach s ∈ V
5         v->state  = initial;
6         v->parent = NULL;
7     foreach (s ∈ V)
8         if s->state == initial
9             BFSvisit(s)
```

# Breitensuche – mit Queue

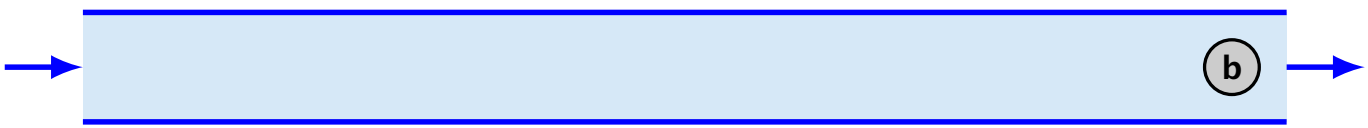
```
1  BFSvisit(s)
2      Queue Q; Q.enqueue(s);
3      while (!Q.empty()) {
4          v = Q.front();
5          if (v->state == initial)
6              v->state  = active;
7              v->bfsNum = bfsCount++;
8              foreach x ∈ N(v)
9                  // v->parent = x; removed statement
10                 Q.enqueue(x);
11         else
12             v->state  = finished;
13             v->finNum = finCount++;
14             Q.dequeue();
15     }
```

Wir haben im wesentlichen nur den Stack durch eine Queue ersetzt!

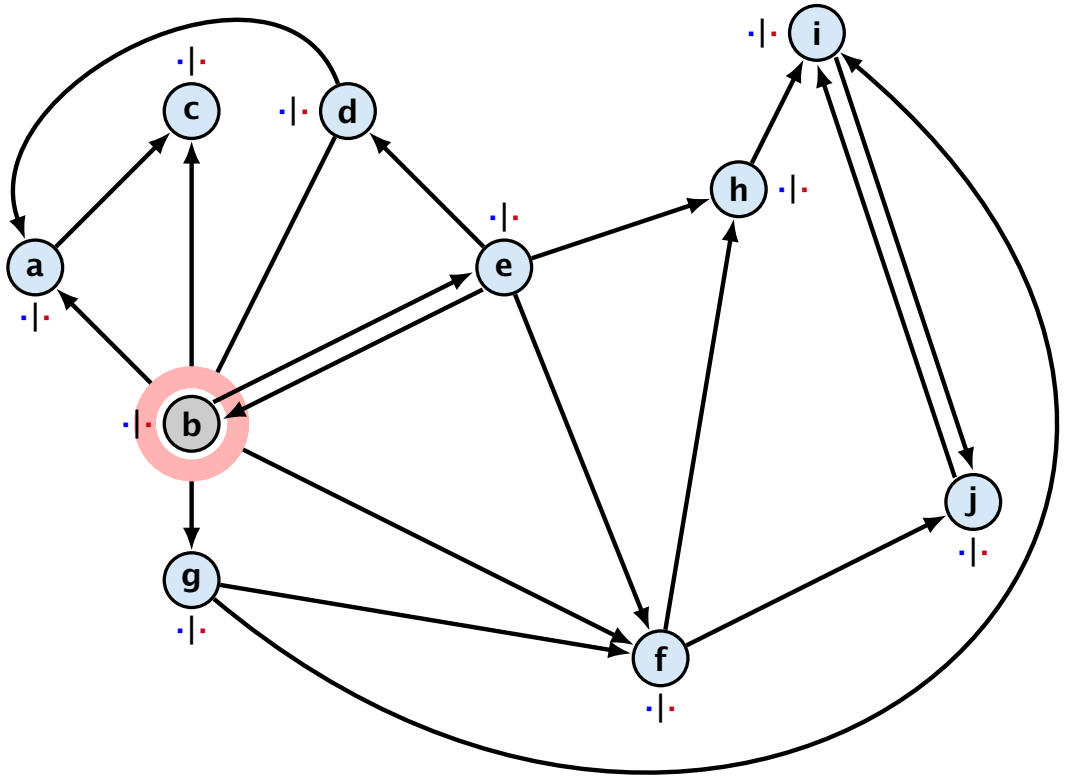
# BFS


a
b
c
d
e
f
g
h
i
j

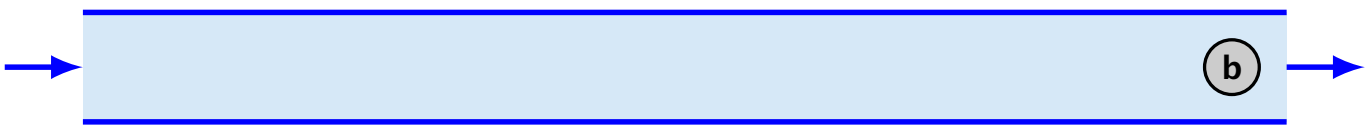
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b

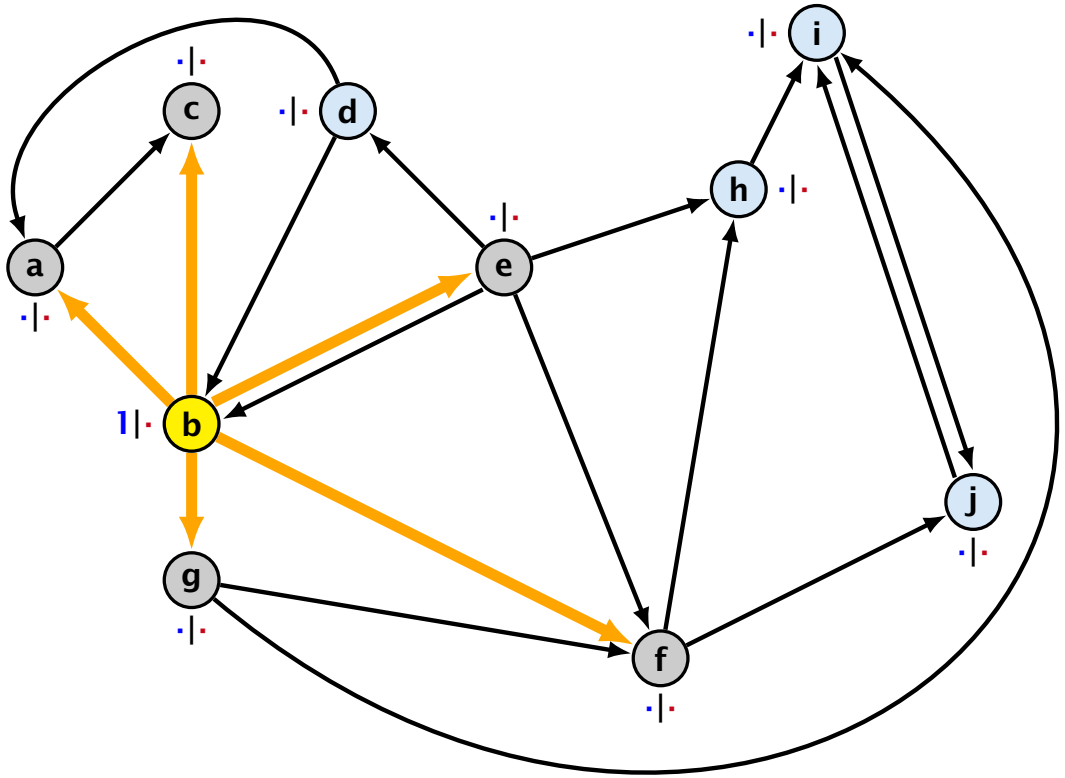
# BFS


a
c
d
e
h
i
b
g
f
j

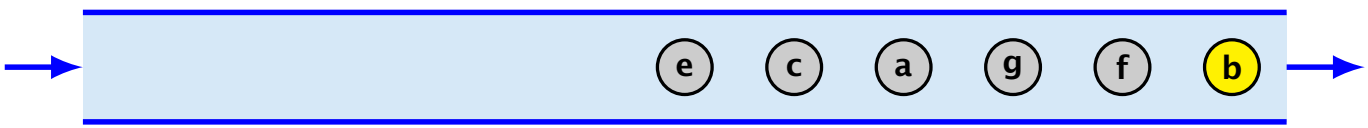
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


b

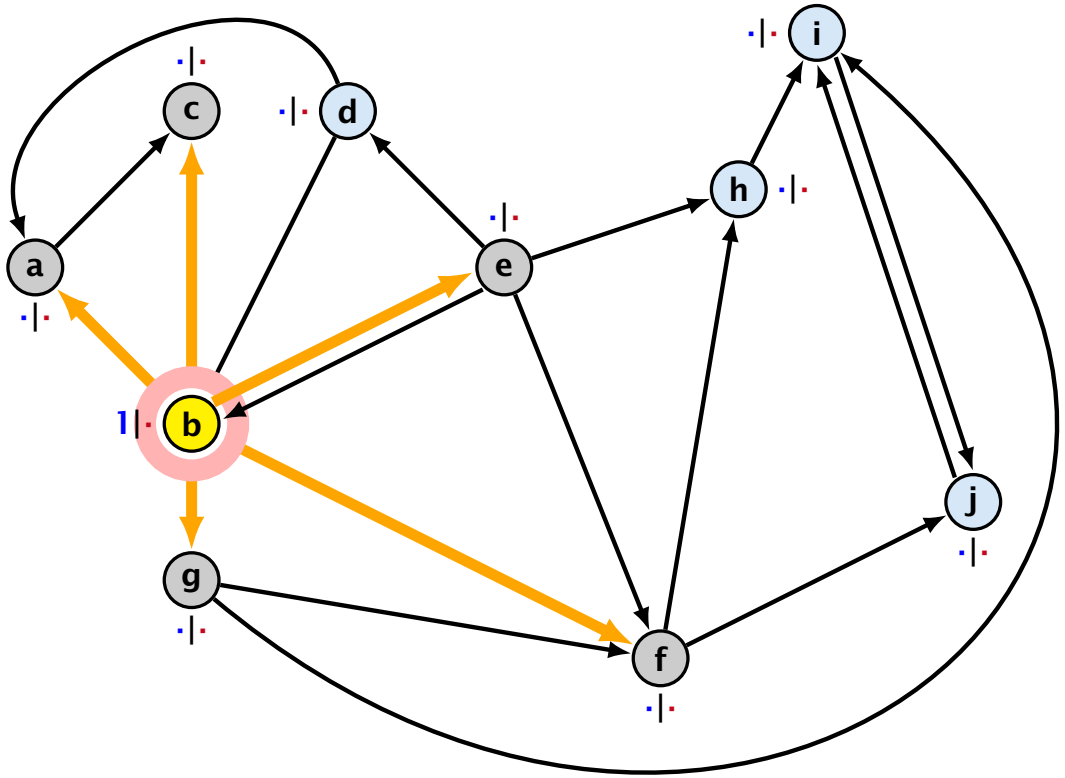
# BFS


i
c
d
h
a
e
1
b
j
g
f

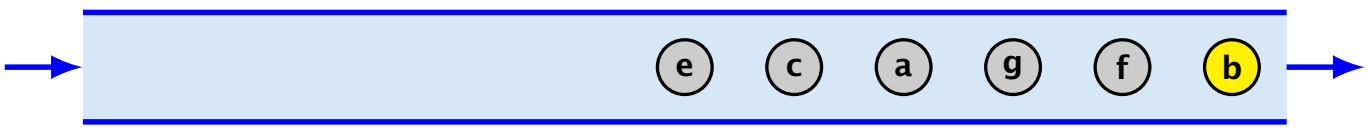
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


e
c
a
g
f
b

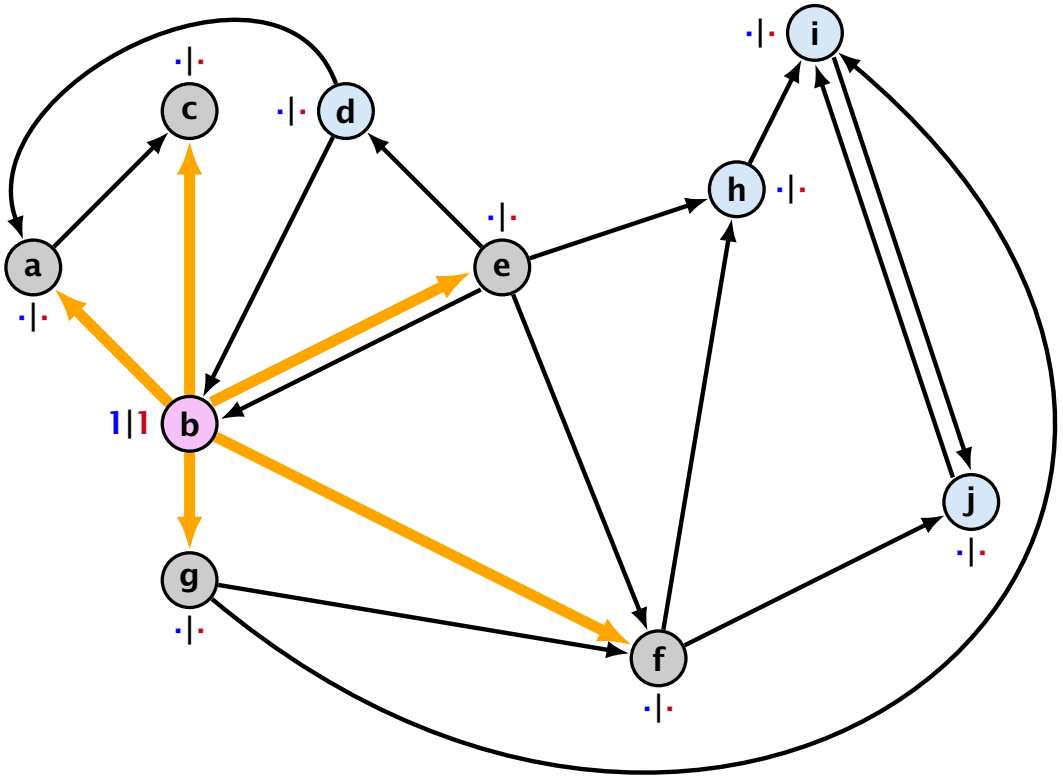
# BFS


i
c
d
h
a
e
1
b
j
g
f

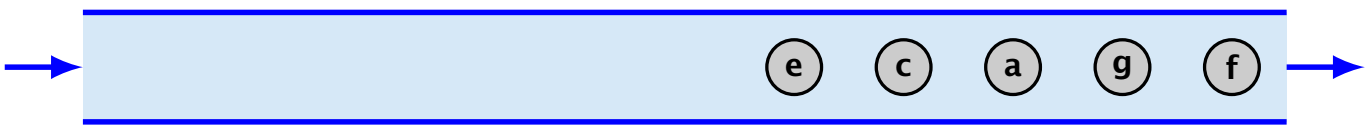
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


e
c
a
g
f
b

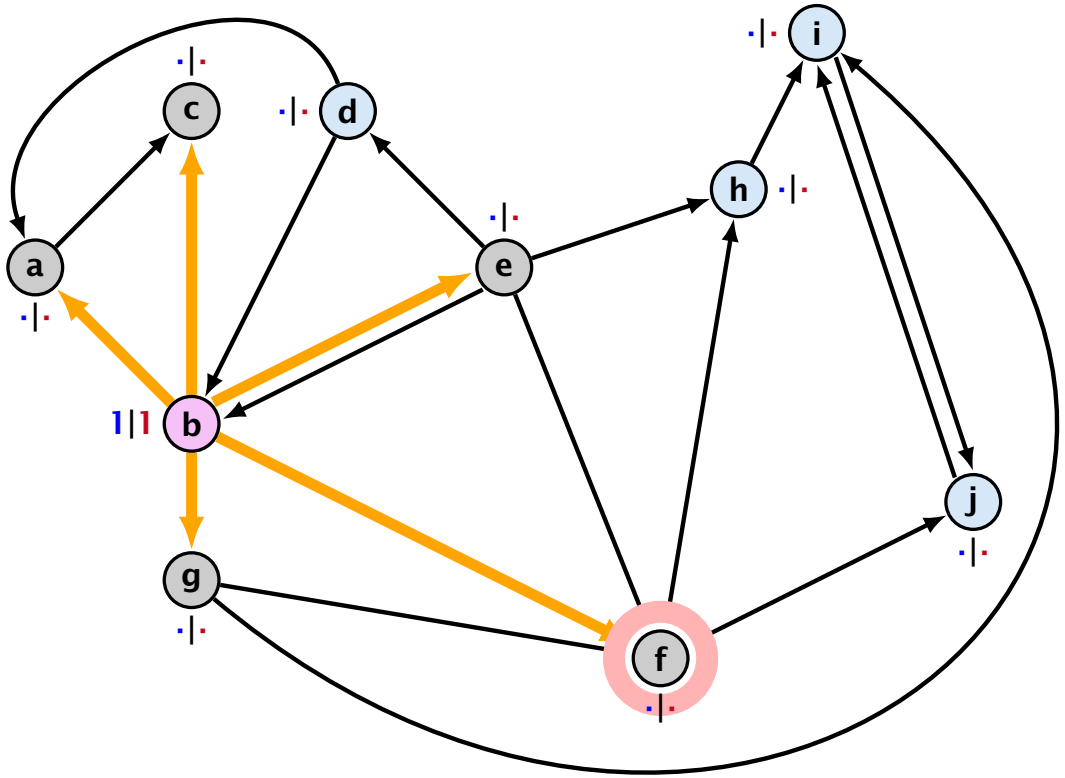
# BFS


i
c
d
h
a
e
1|1
b
j
g
f

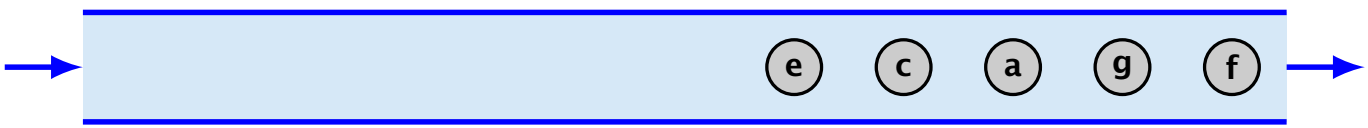
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


e
c
a
g
f

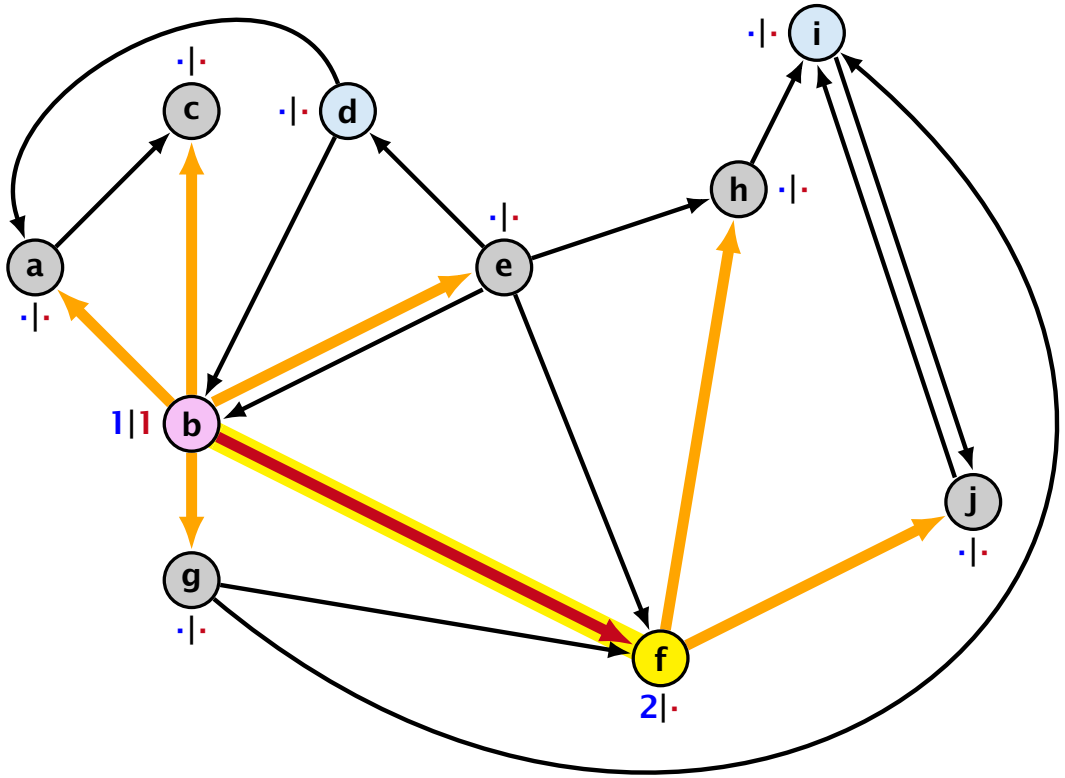
# BFS


i
c
d
h
a
e
1|1
b
j
g
f

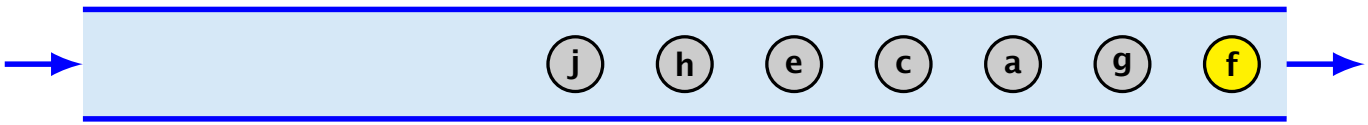
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


e
c
a
g
f

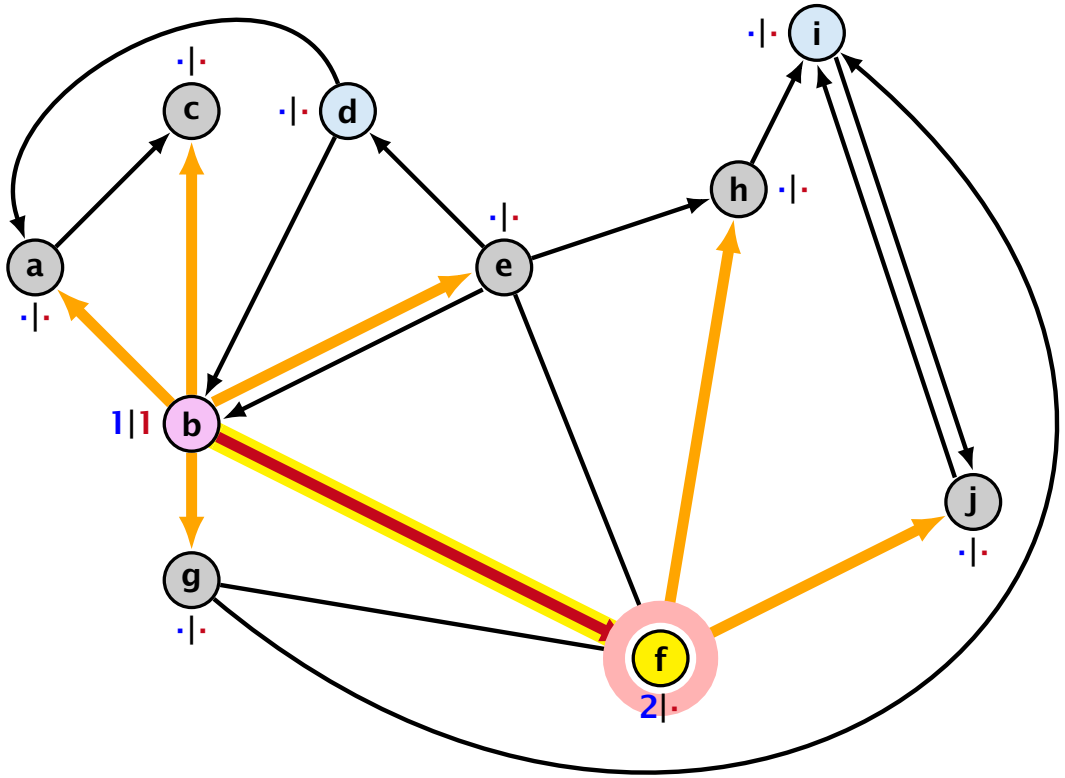
# BFS


i
c
d
h
a
e
1|1
b
j
g
f
2|·

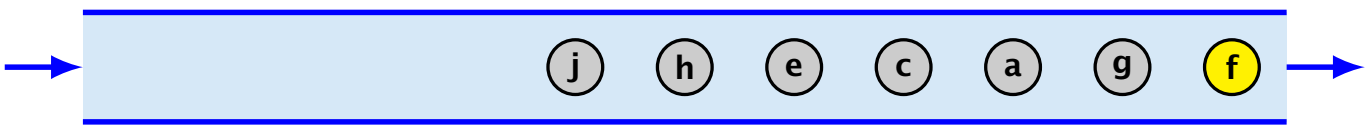
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


j
h
e
c
a
g
f

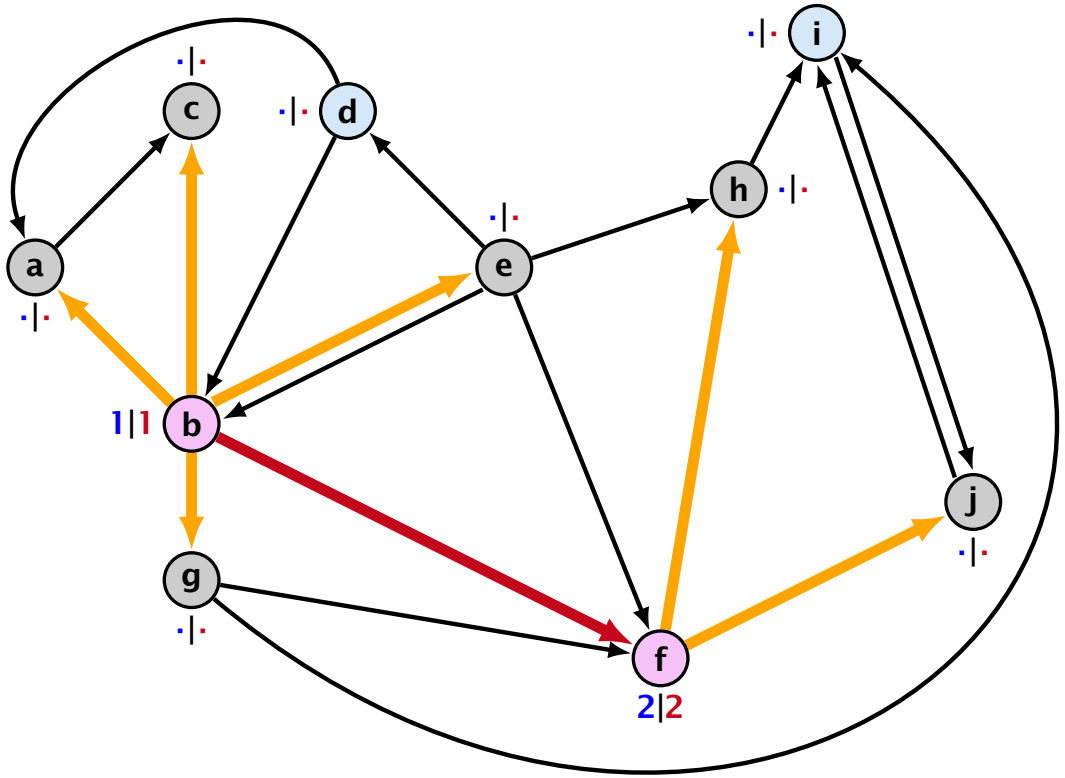
# BFS


i
c
d
h
a
e
1|1
b
j
g
f
2|

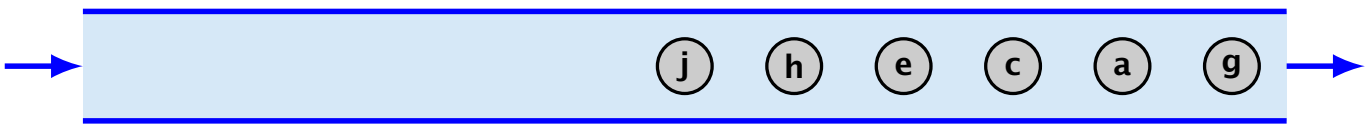
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


j
h
e
c
a
g
f

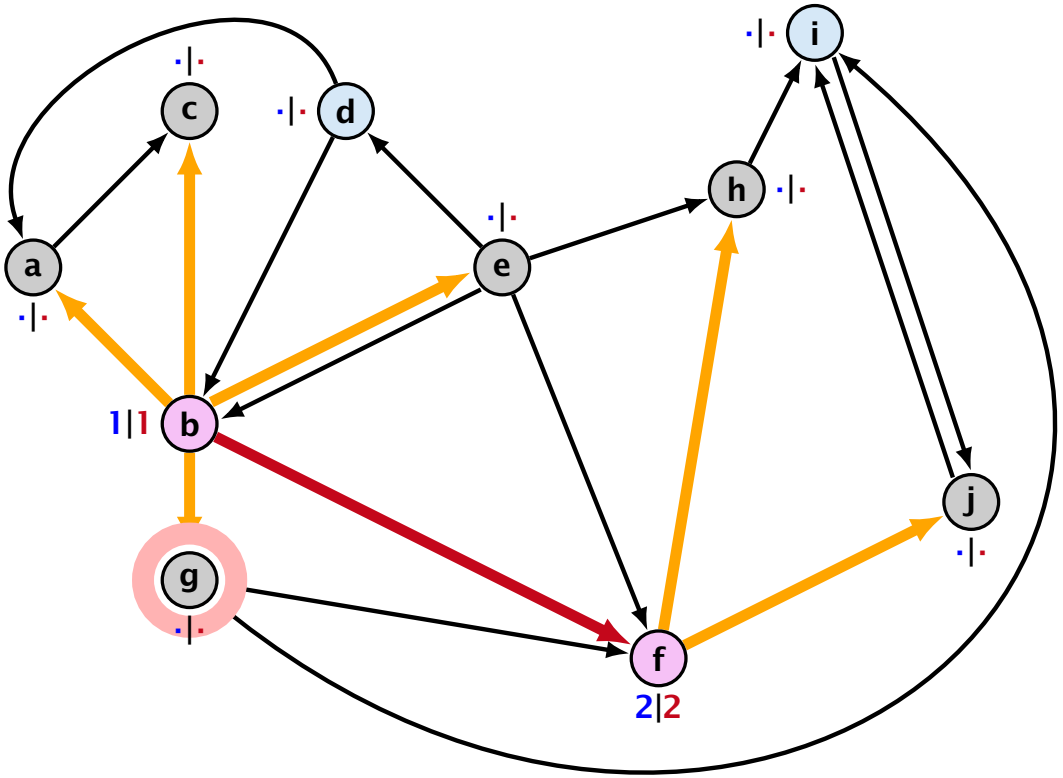
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
a
·|·
e
·|·
1|1
b
j
·|·
g
·|·
f
2|2

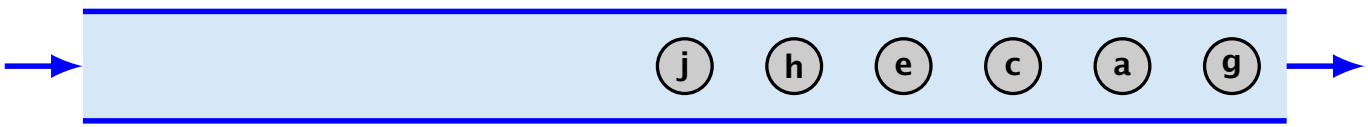
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


j
h
e
c
a
g

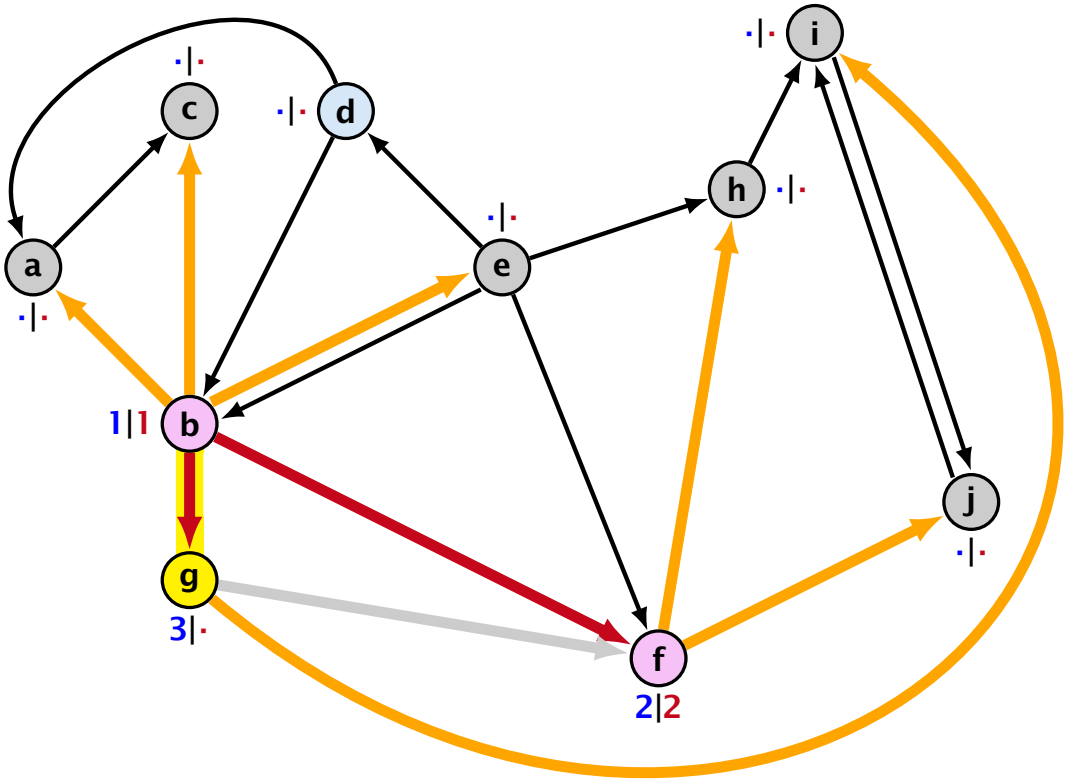
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
a
·|·
e
·|·
1|1
b
j
·|·
g
·|·
f
2|2

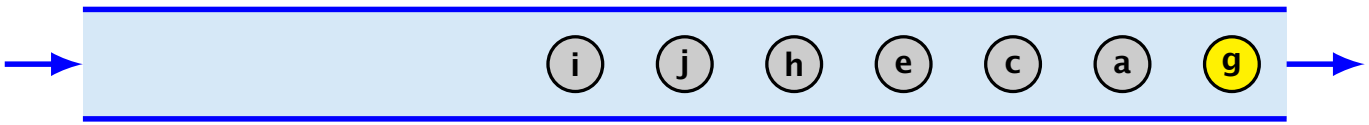
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


j
h
e
c
a
g

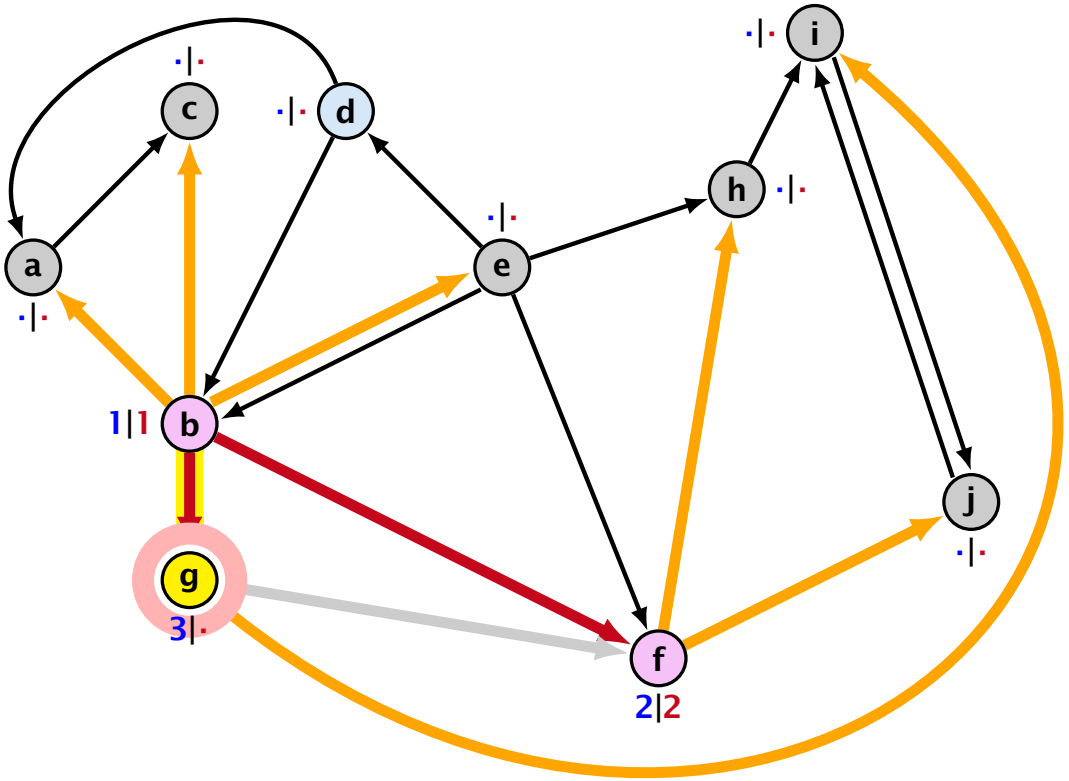
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
e
·|·
a
·|·
1|1
b
j
·|·
g
3|·
f
2|2

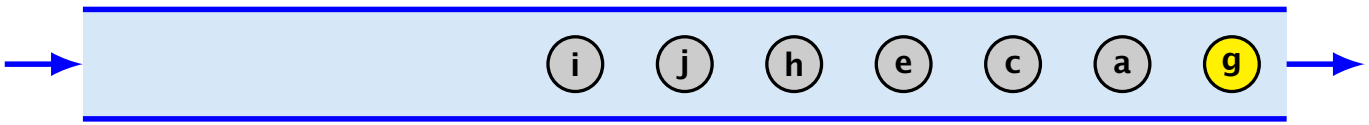
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
j
h
e
c
a
g

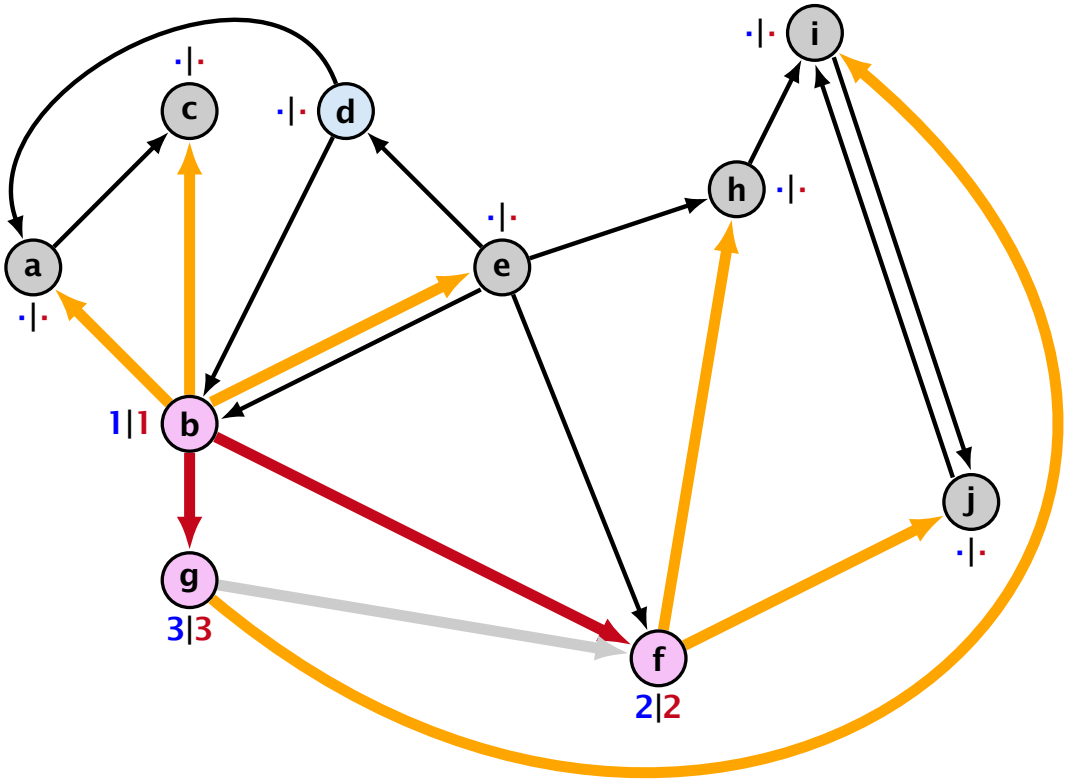
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
a
·|·
e
·|·
1|1
b
j
·|·
g
3|·
f
2|2

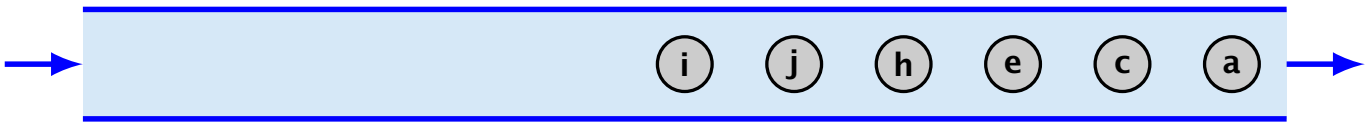
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
j
h
e
c
a
g

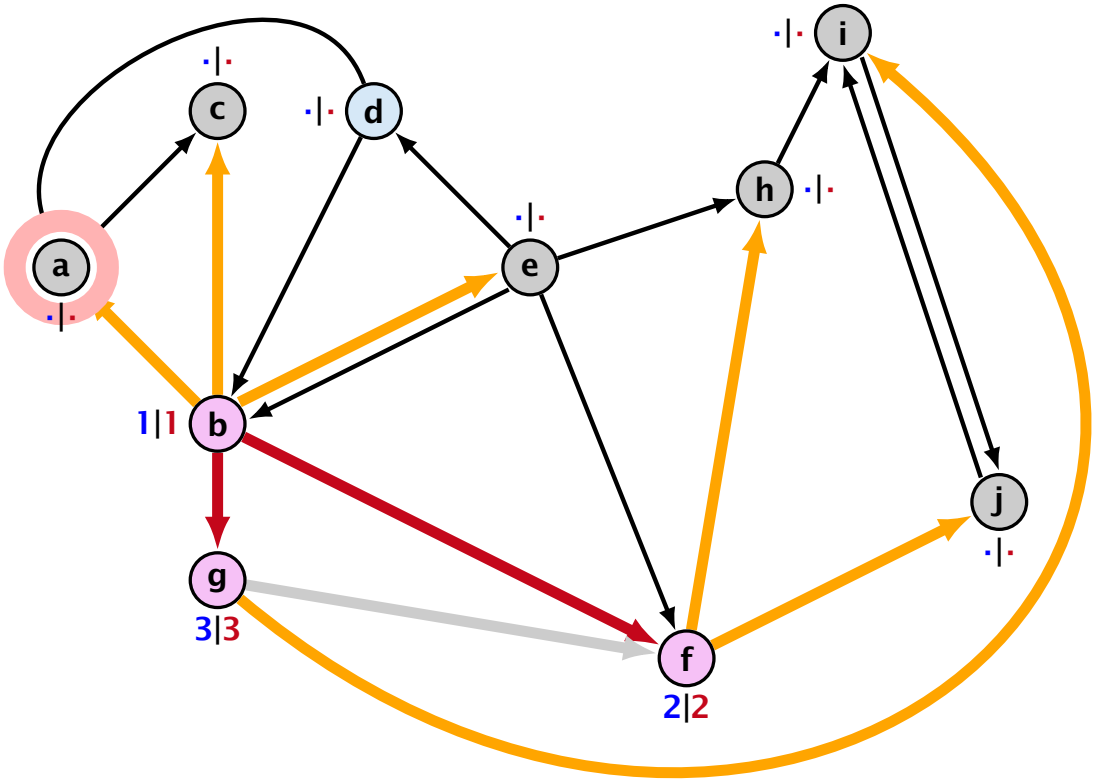
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
e
·|·
a
·|·
1|1
b
j
·|·
g
3|3
f
2|2

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
j
h
e
c
a

# BFS

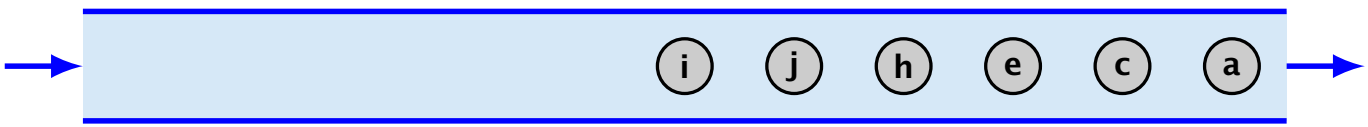



Baumkante

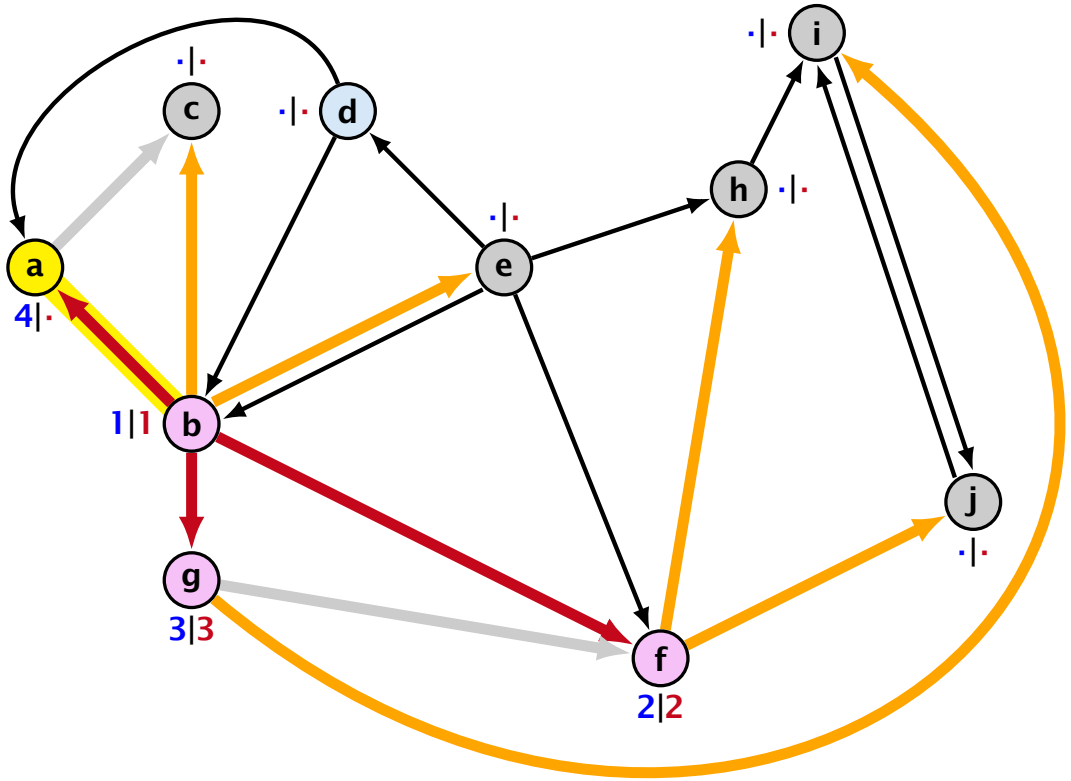
Vorwärtskante

Rückwärtskante

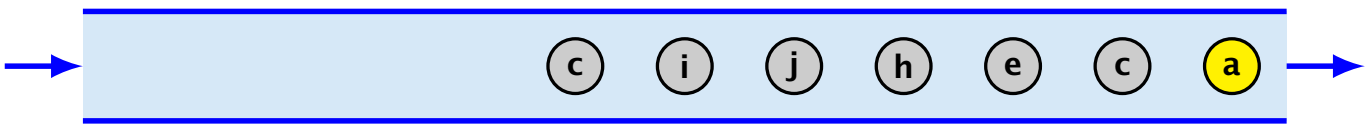
Kreuzkante



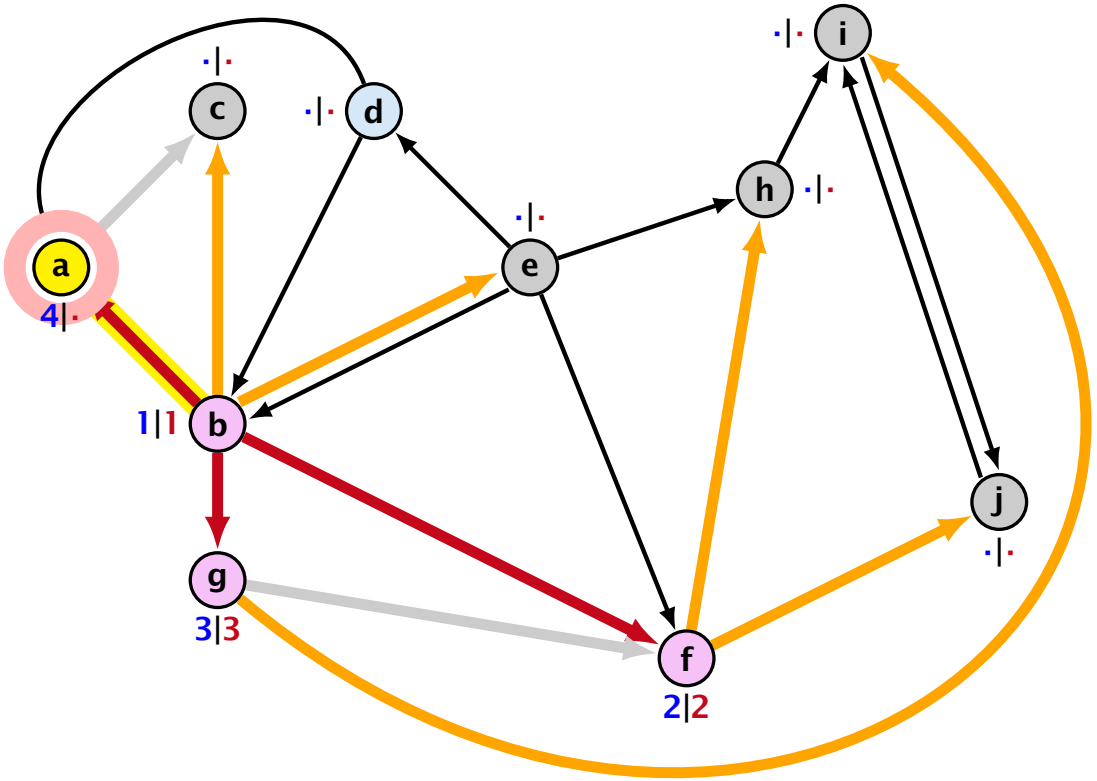
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
e
·|·
a
4|·
1|1
b
g
3|3
f
2|2
j
·|·

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


c
i
j
h
e
c
a

# BFS

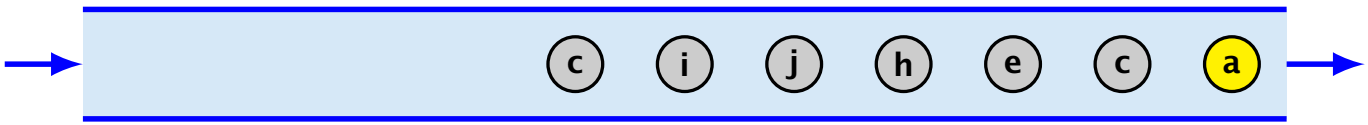



Baumkante

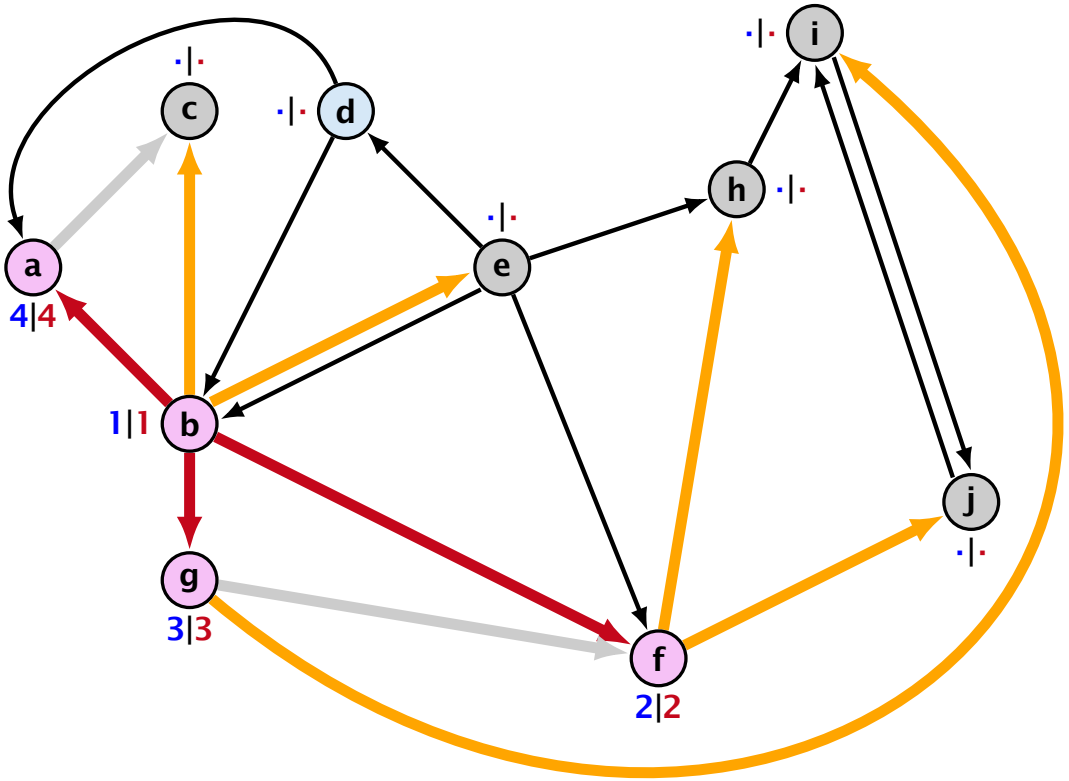
Vorwärtskante

Rückwärtskante

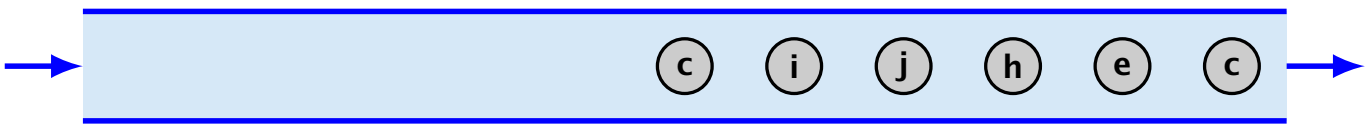
Kreuzkante



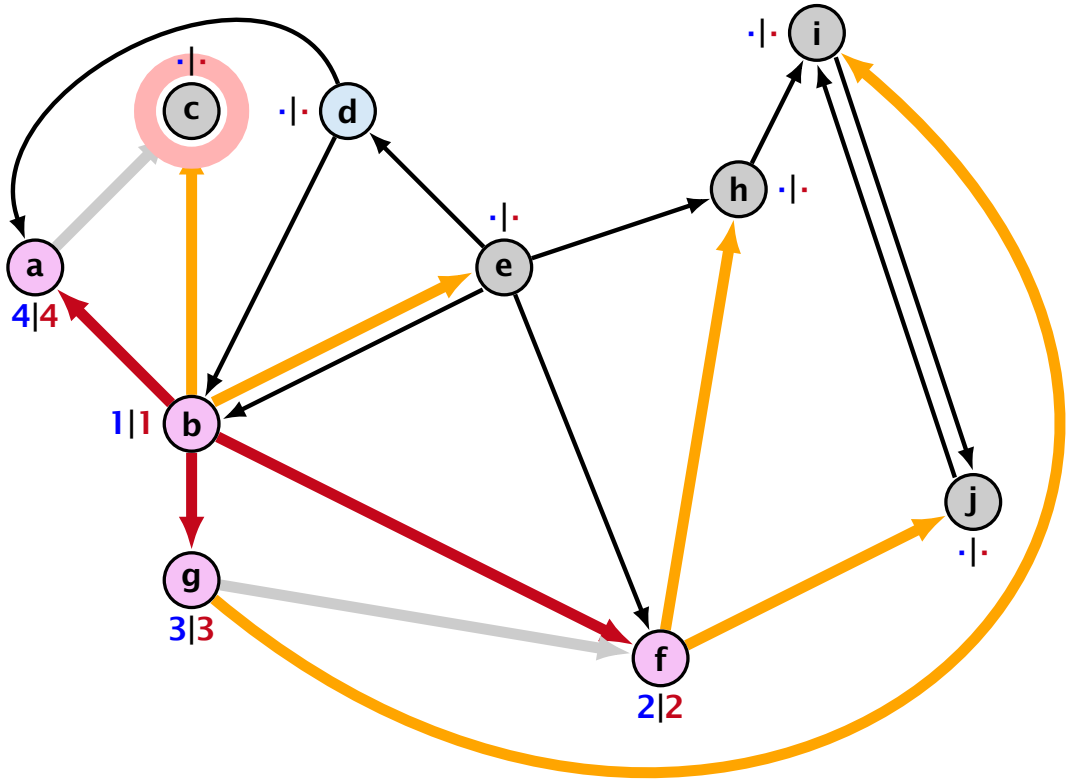
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
·|·
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

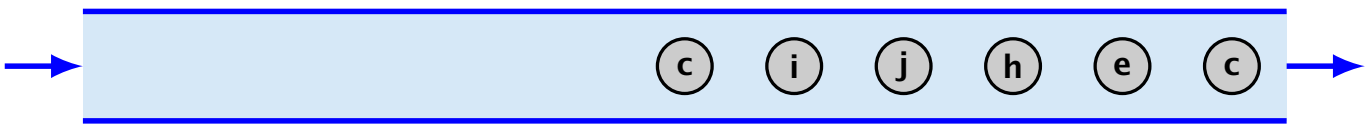
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


c
i
j
h
e
c

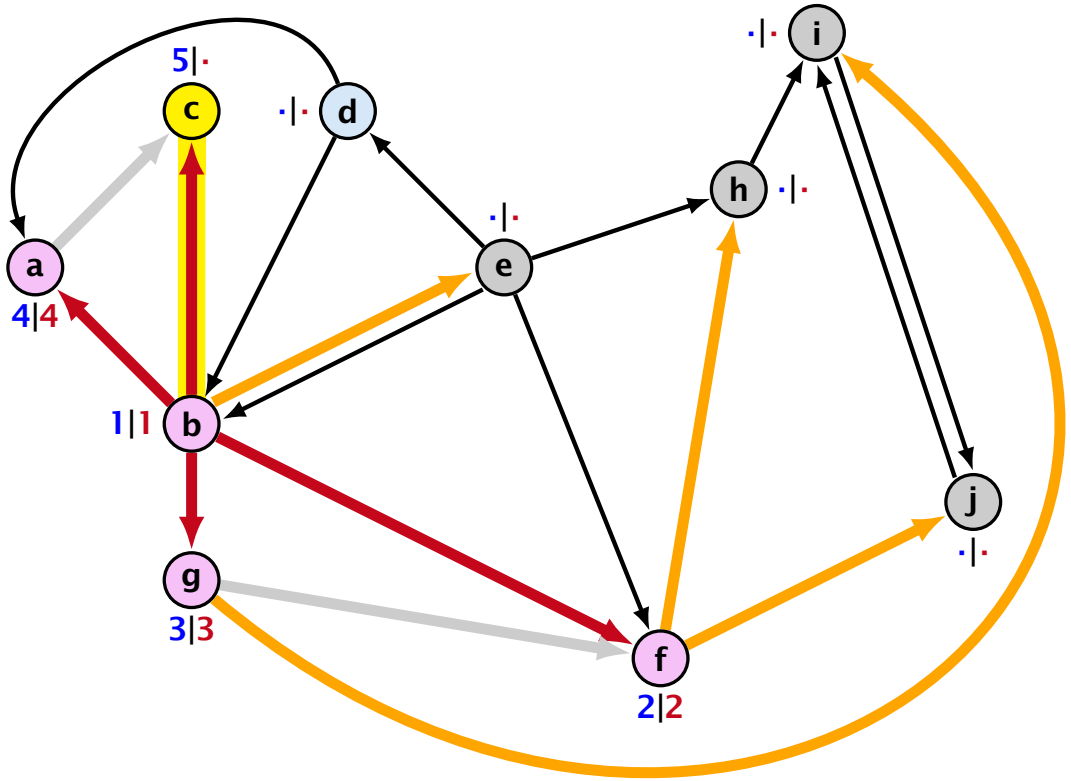
# BFS


·|·
c
·|·
d
·|·
i
h
·|·
·|·
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


c
i
j
h
e
c

# BFS

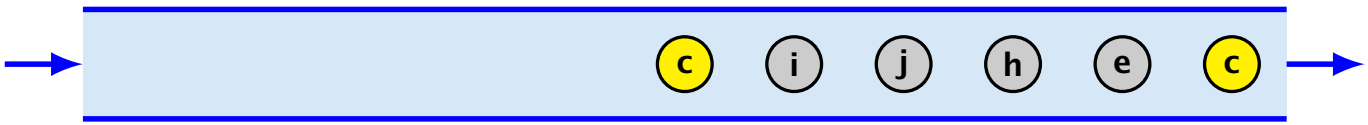



Baumkante

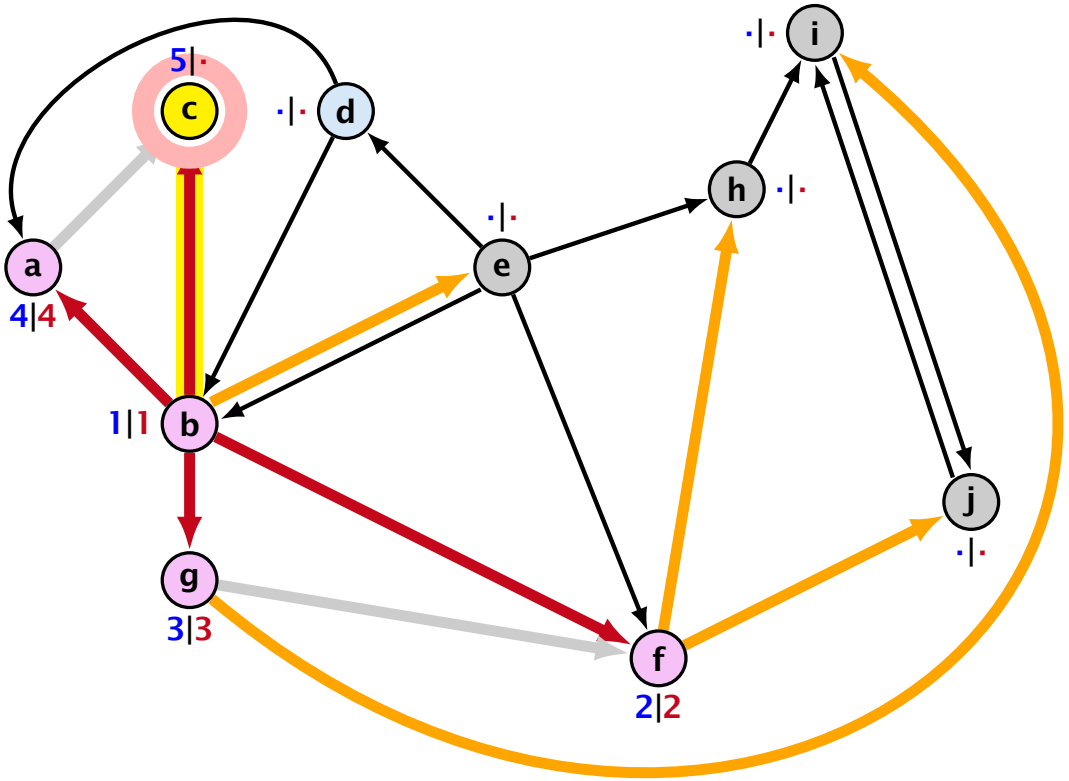
Vorwärtskante

Rückwärtskante

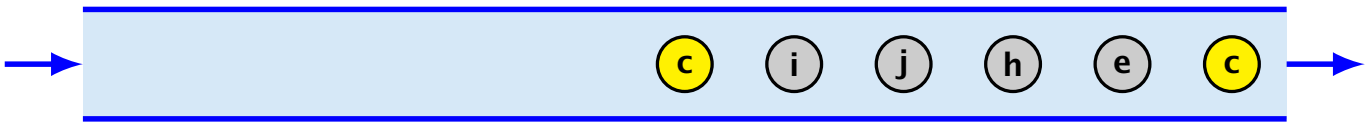
Kreuzkante



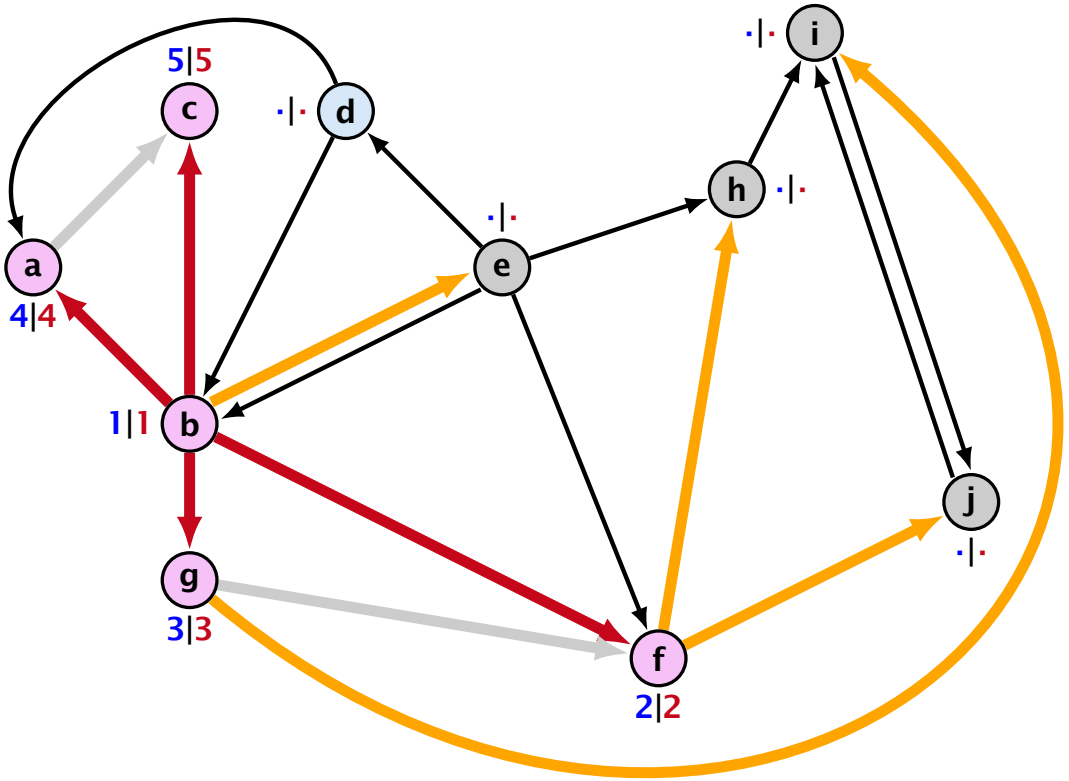
# BFS


5|·
c
·|·
d
·|·
i
h
·|·
·|·
e
a
4|4
1|1
b
j
·|·
g
3|3
f
2|2

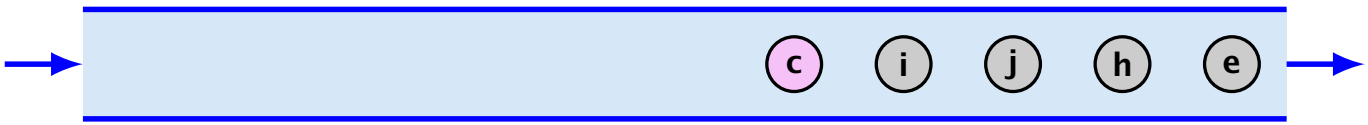
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


c
i
j
h
e
c

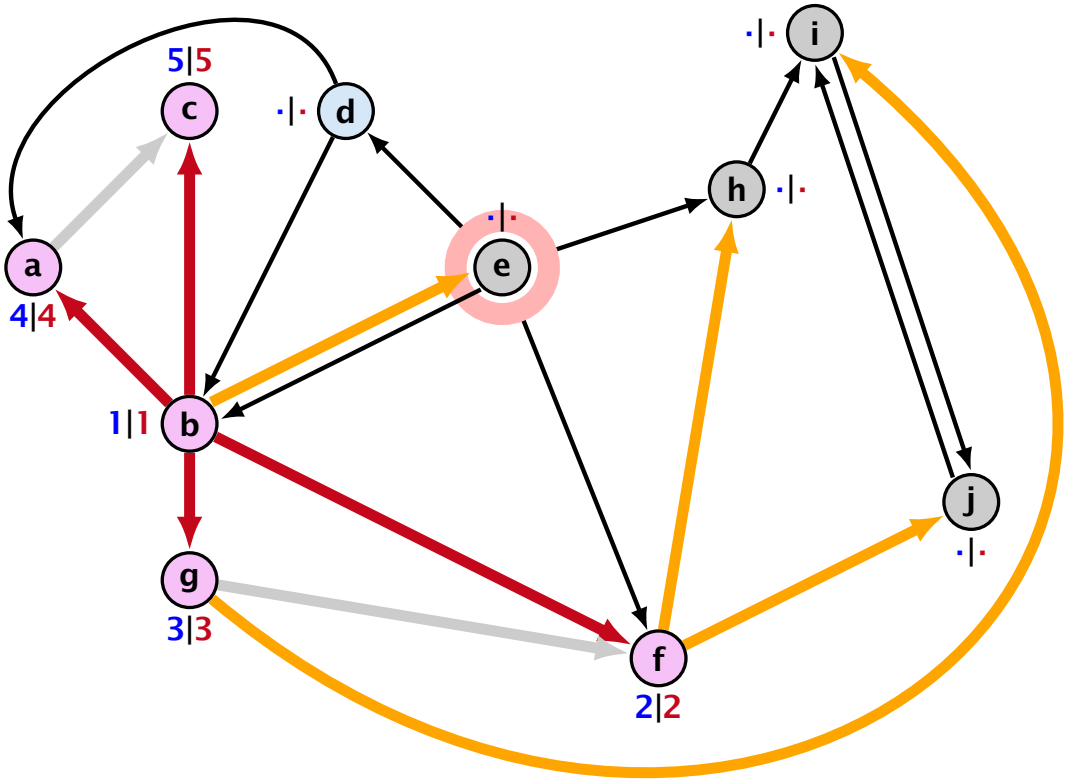
# BFS


5|5
c
·|·
d
·|·
i
h
·|·
·|·
e
a
4|4
1|1
b
j
·|·
g
3|3
f
2|2

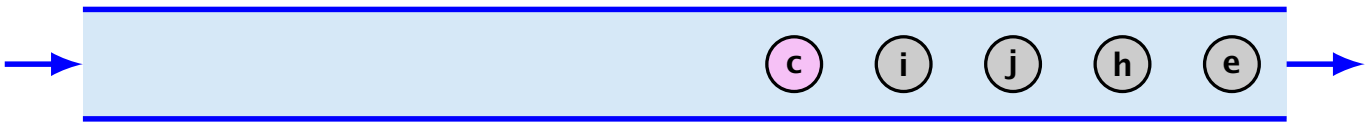
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


c
i
j
h
e

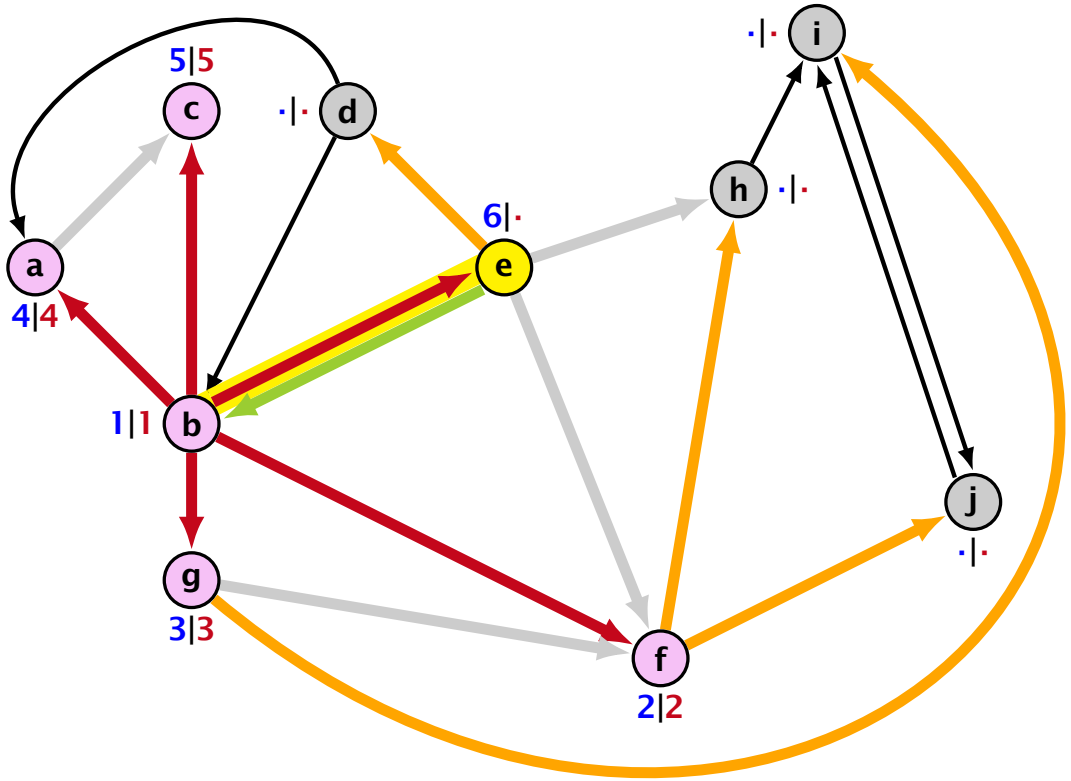
# BFS


5|5
c
·|·
d
·|·
i
h
·|·
·|·
e
a
4|4
1|1
b
j
·|·
g
3|3
f
2|2

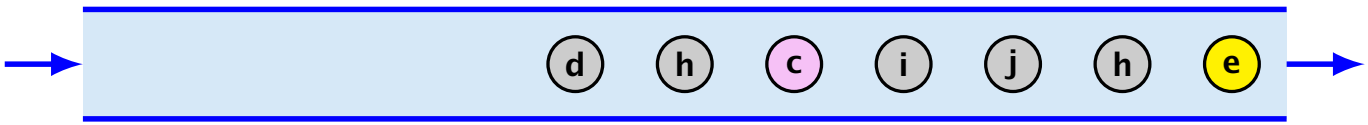
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


c
i
j
h
e

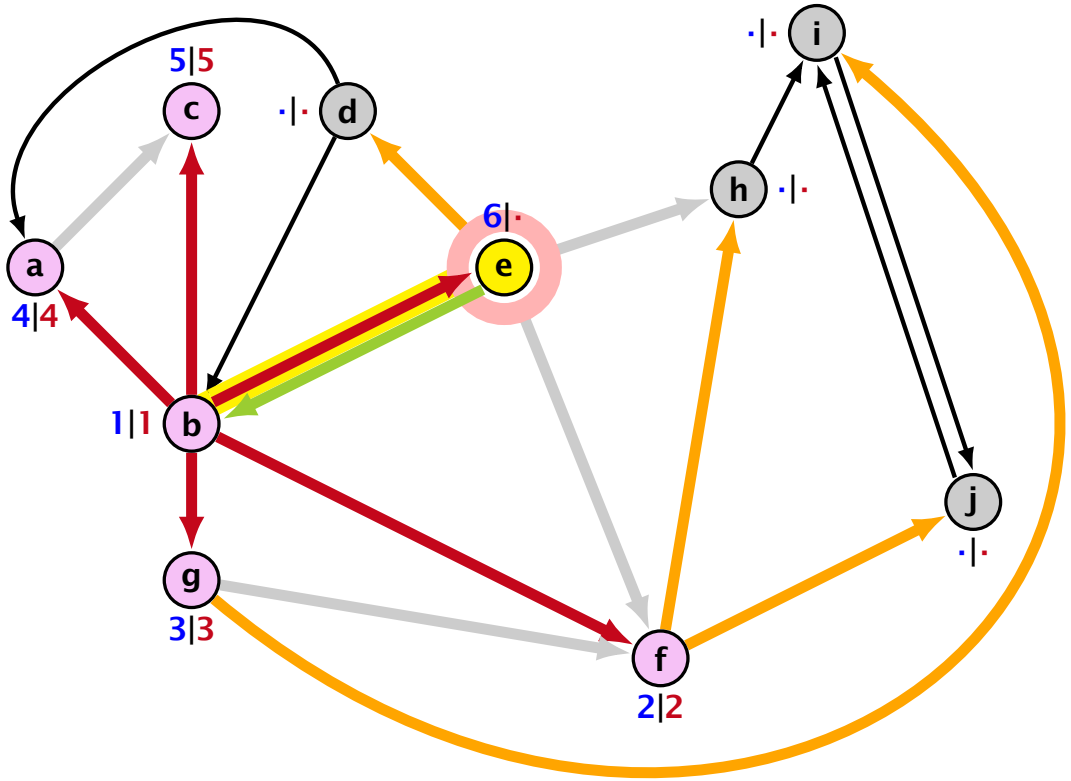
# BFS


5|5
c
·|·
d
·|·
i
h
·|·
6|·
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

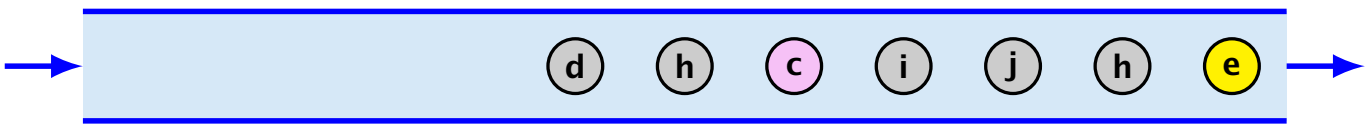
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
h
c
i
j
h
e

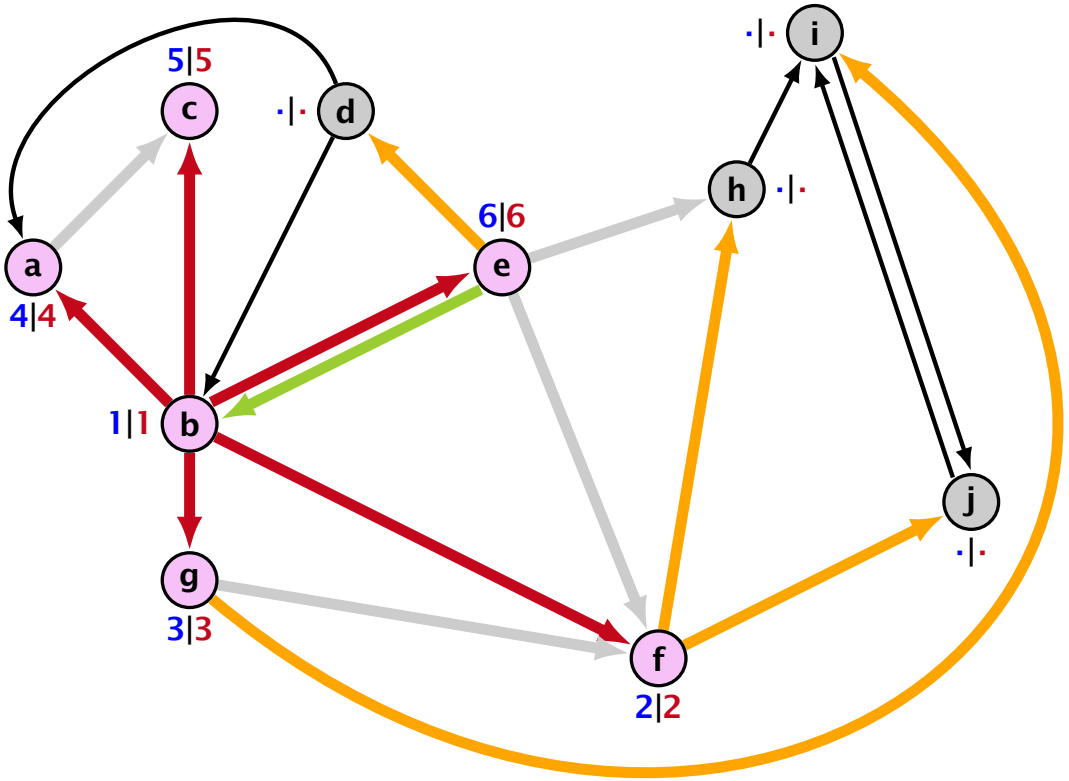
# BFS


5|5
c
·|·
d
·|·
i
h
·|·
6|·
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

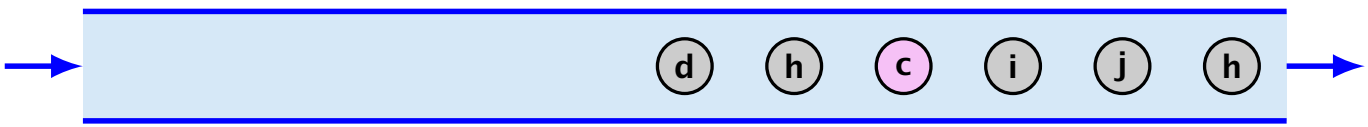
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
h
c
i
j
h
e

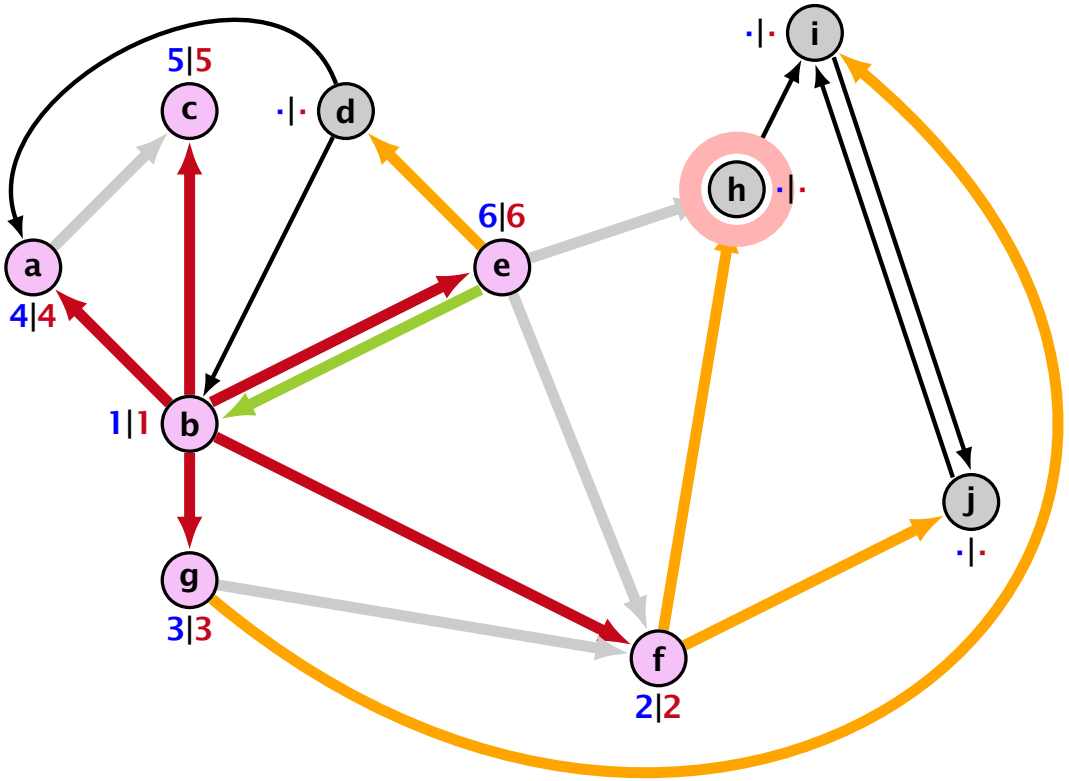
# BFS


5|5
c
·|·
d
·|·
i
h
·|·
6|6
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

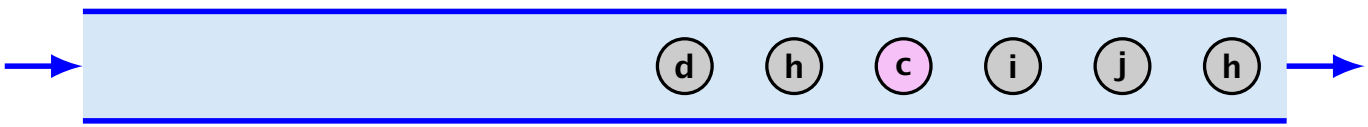
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
h
c
i
j
h

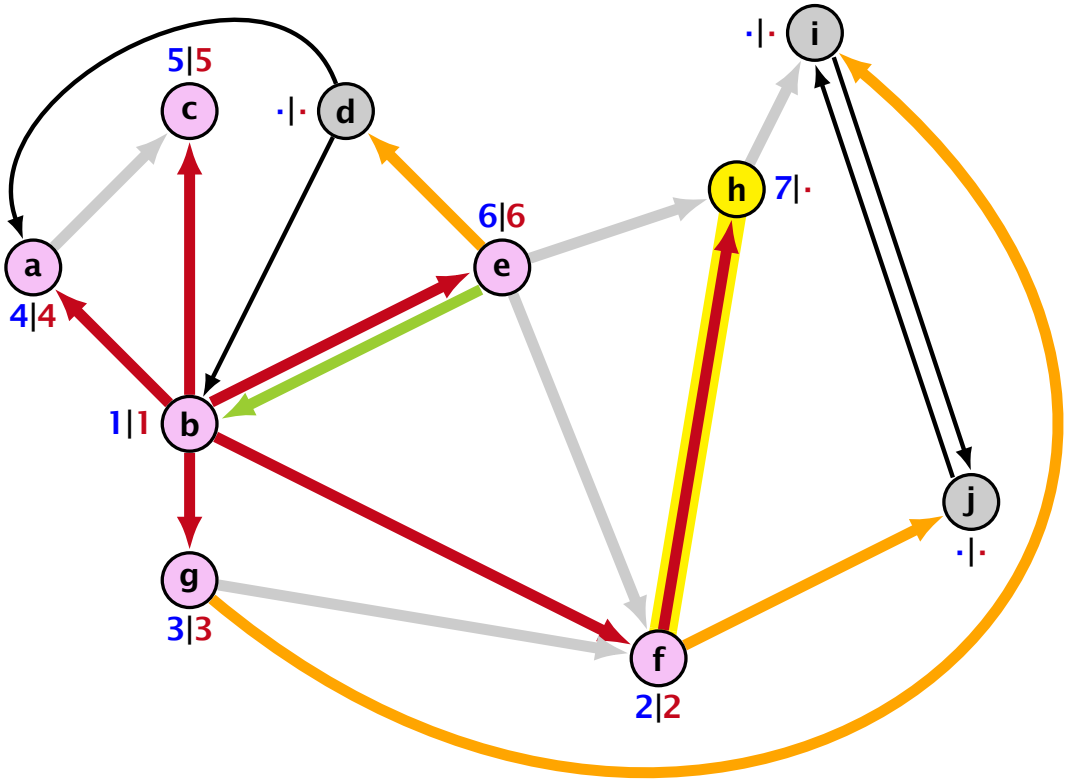
# BFS


5|5
c
·|·
d
·|·
i
h
·|·
6|6
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

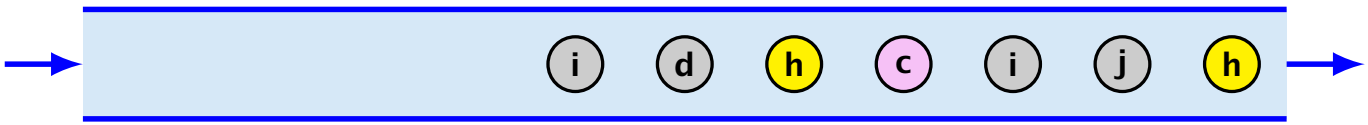
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


d
h
c
i
j
h

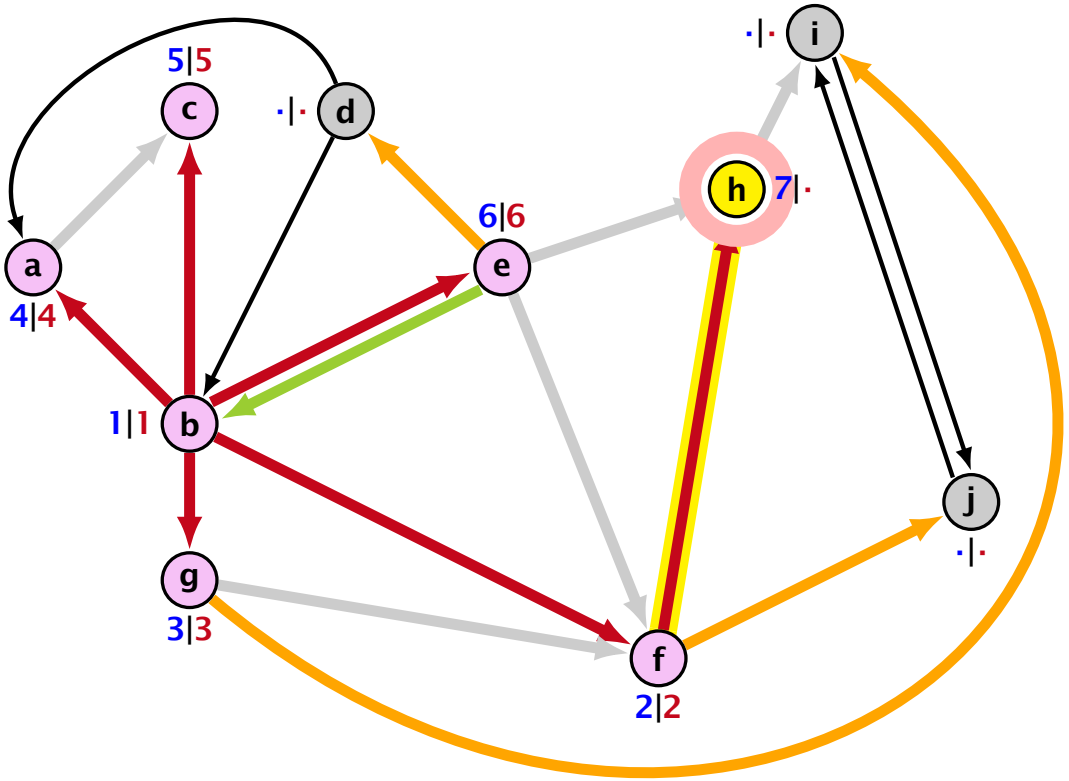
# BFS


5|5
c
·|·
d
·|·
i
h
7|·
6|6
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

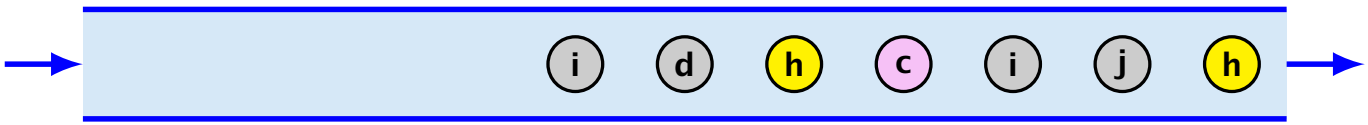
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
d
h
c
i
j
h

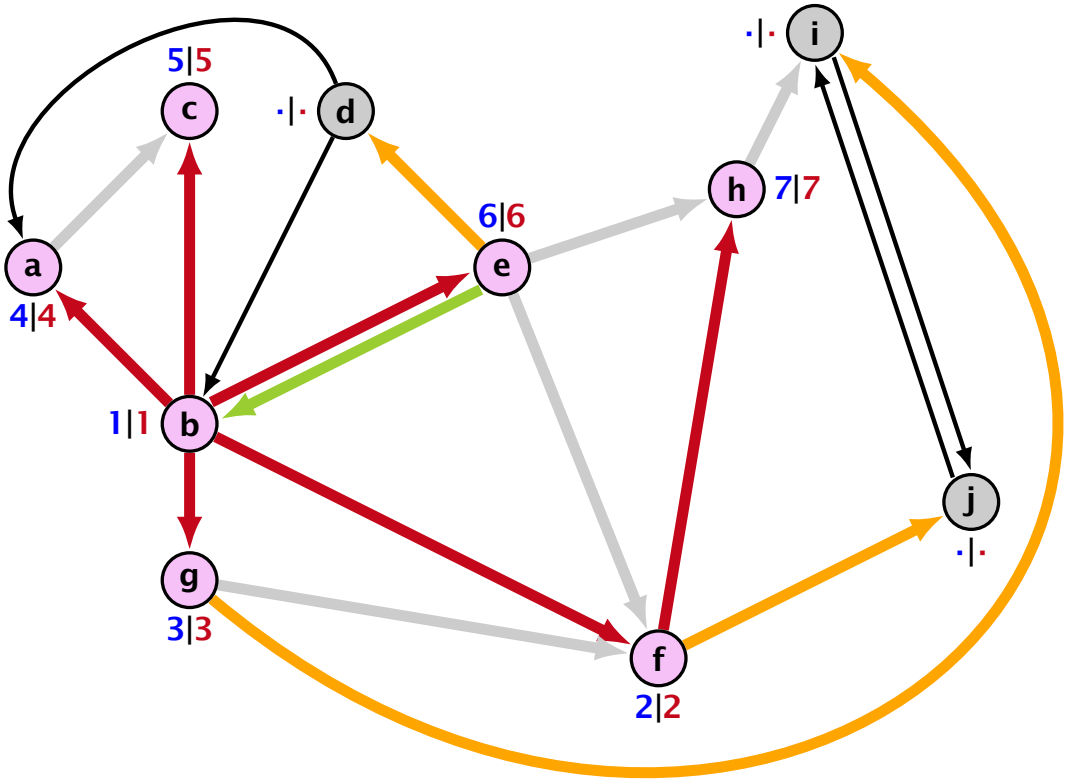
# BFS


5|5
c
·|·
d
·|·
i
h
7|·
6|6
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

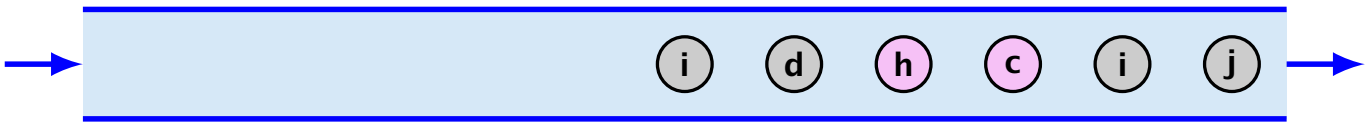
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
d
h
c
i
j
h

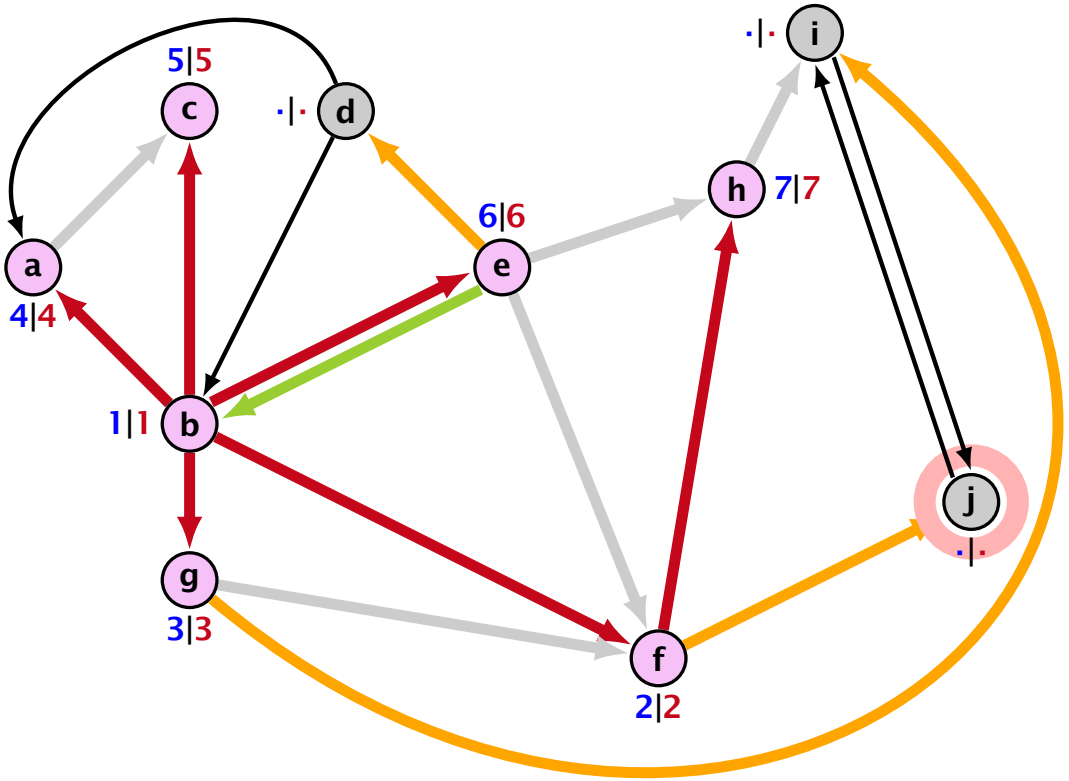
# BFS


5|5
c
·|·
d
·|·
i
h
7|7
6|6
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

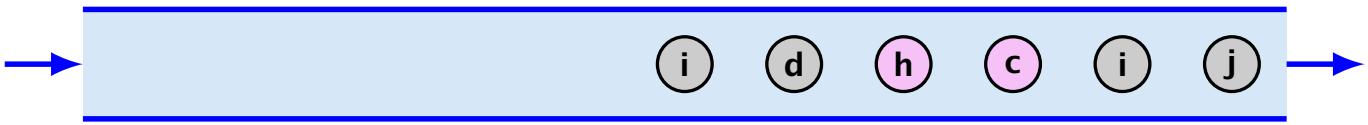
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
d
h
c
i
j

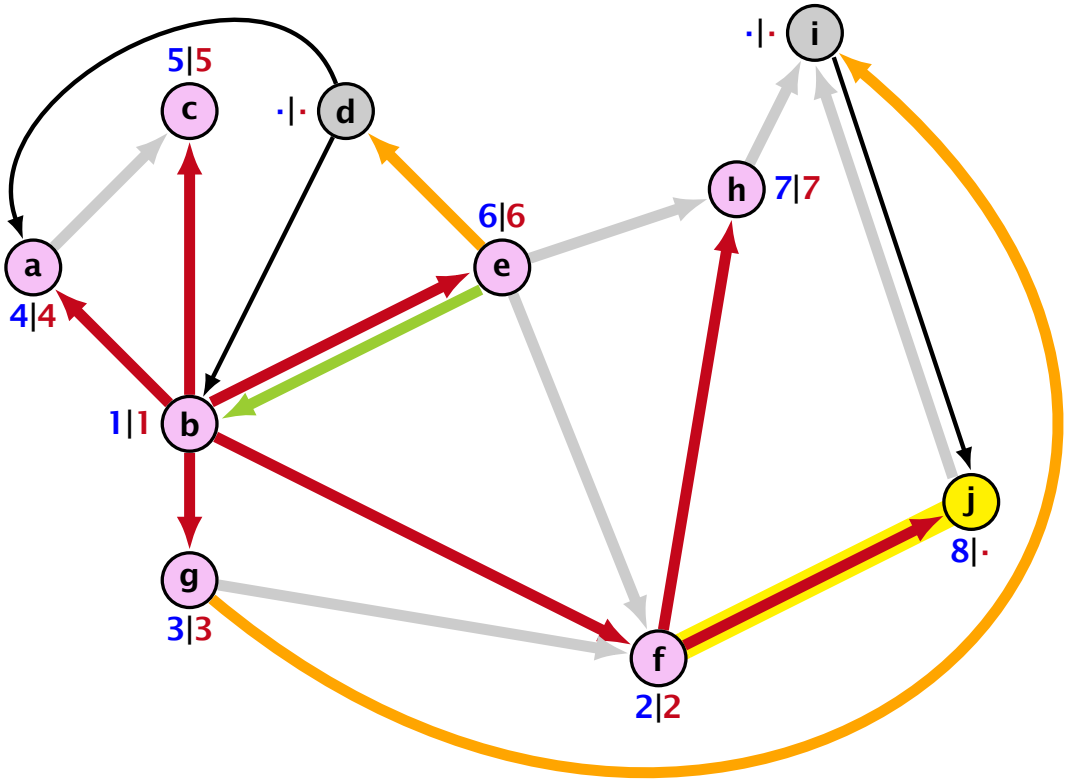
# BFS


5|5
c
·|·
d
·|·
i
h
7|7
6|6
e
a
4|4
1|1
b
g
3|3
f
2|2
j
·|·

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
d
h
c
i
j

# BFS

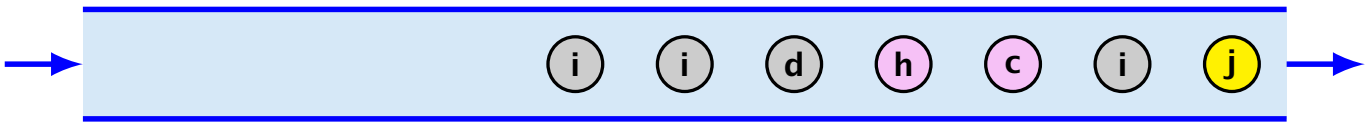



Baumkante

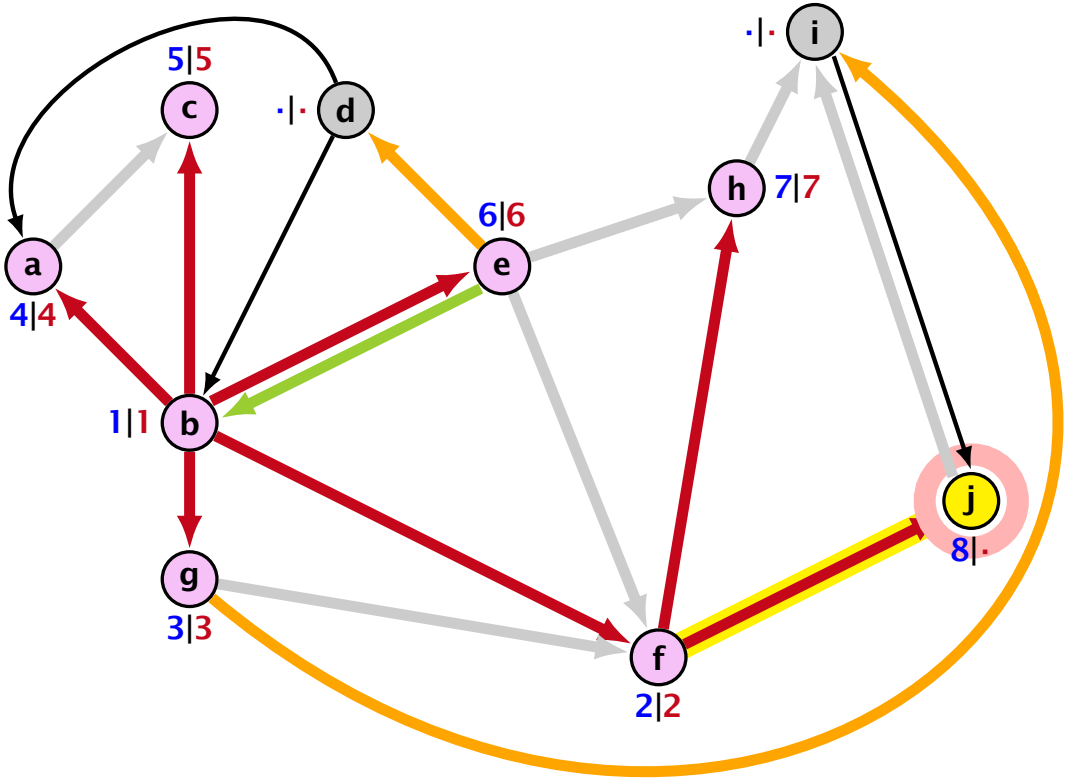
Vorwärtskante

Rückwärtskante

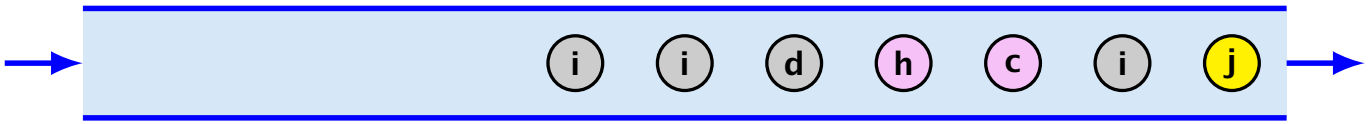
Kreuzkante



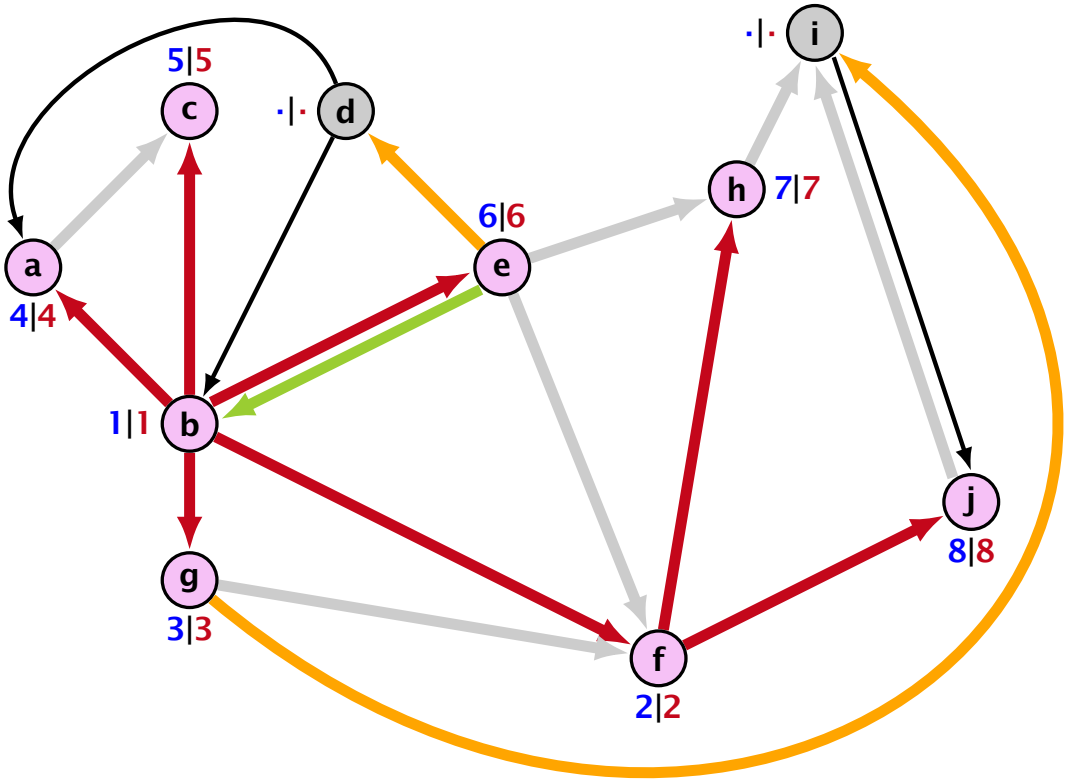
# BFS


5|5
c
·|·
d
·|·
i
h
7|7
6|6
e
a
4|4
1|1
b
j
8|·
g
3|3
f
2|2

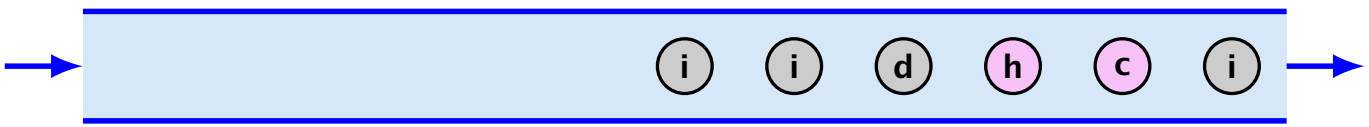
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d
h
c
i
j

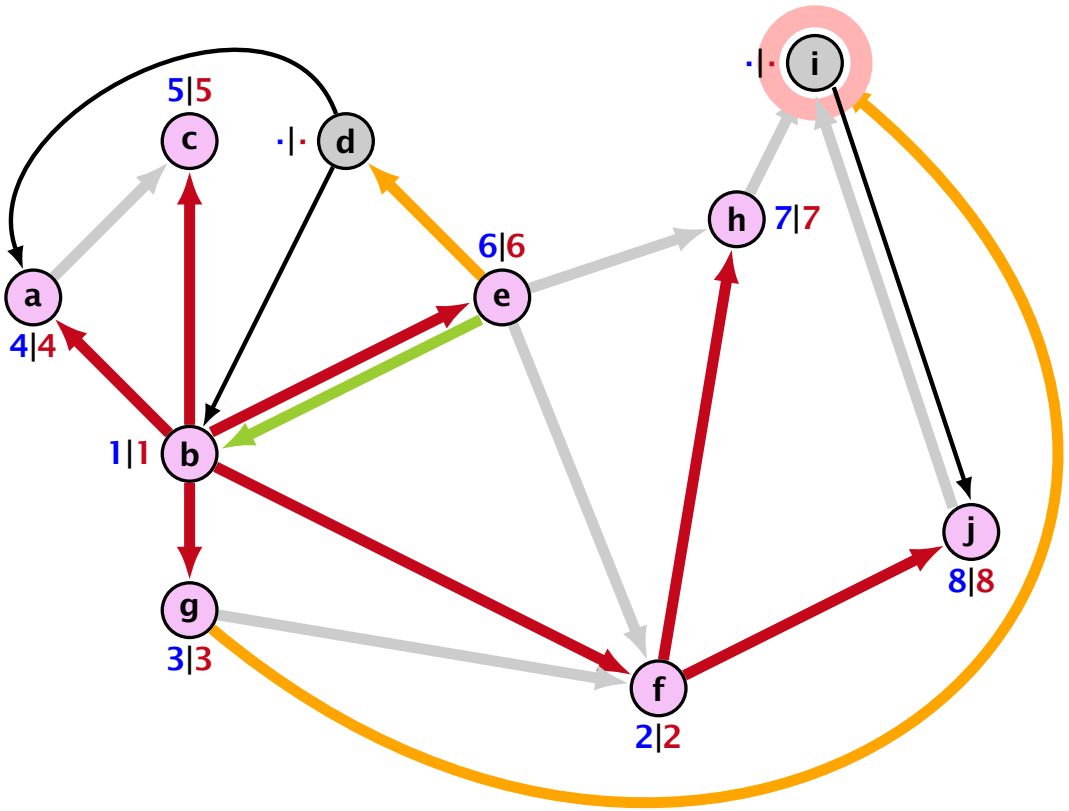
# BFS


·|·
i
5|5
c
·|·
d
6|6
h
7|7
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

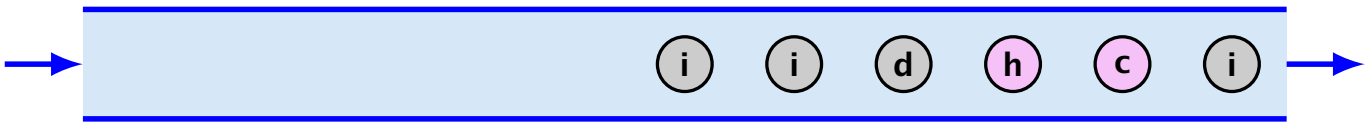
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d
h
c
i

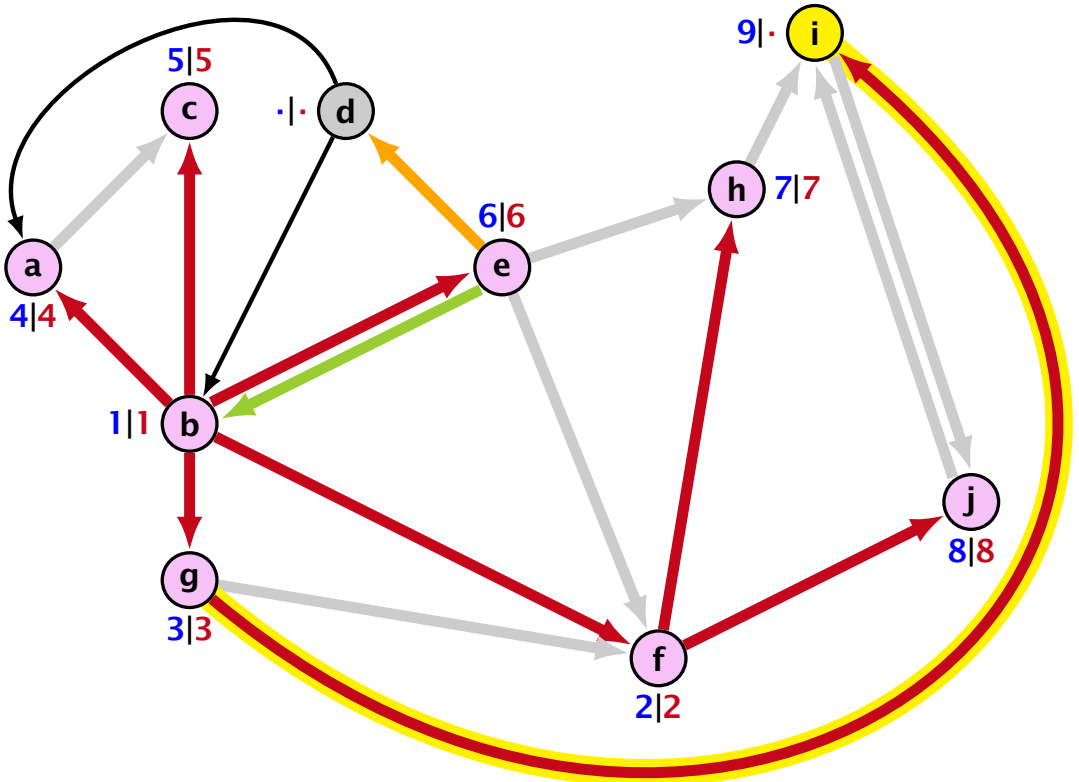
# BFS


·|·
i
5|5
c
·|·
d
6|6
h
7|7
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

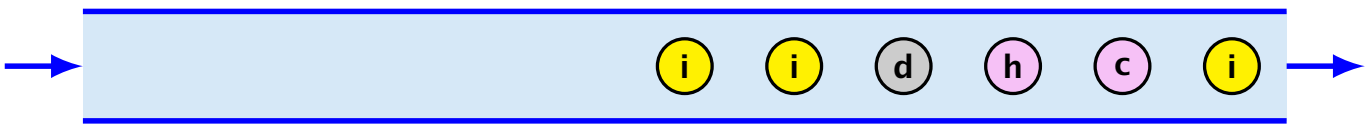
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d
h
c
i

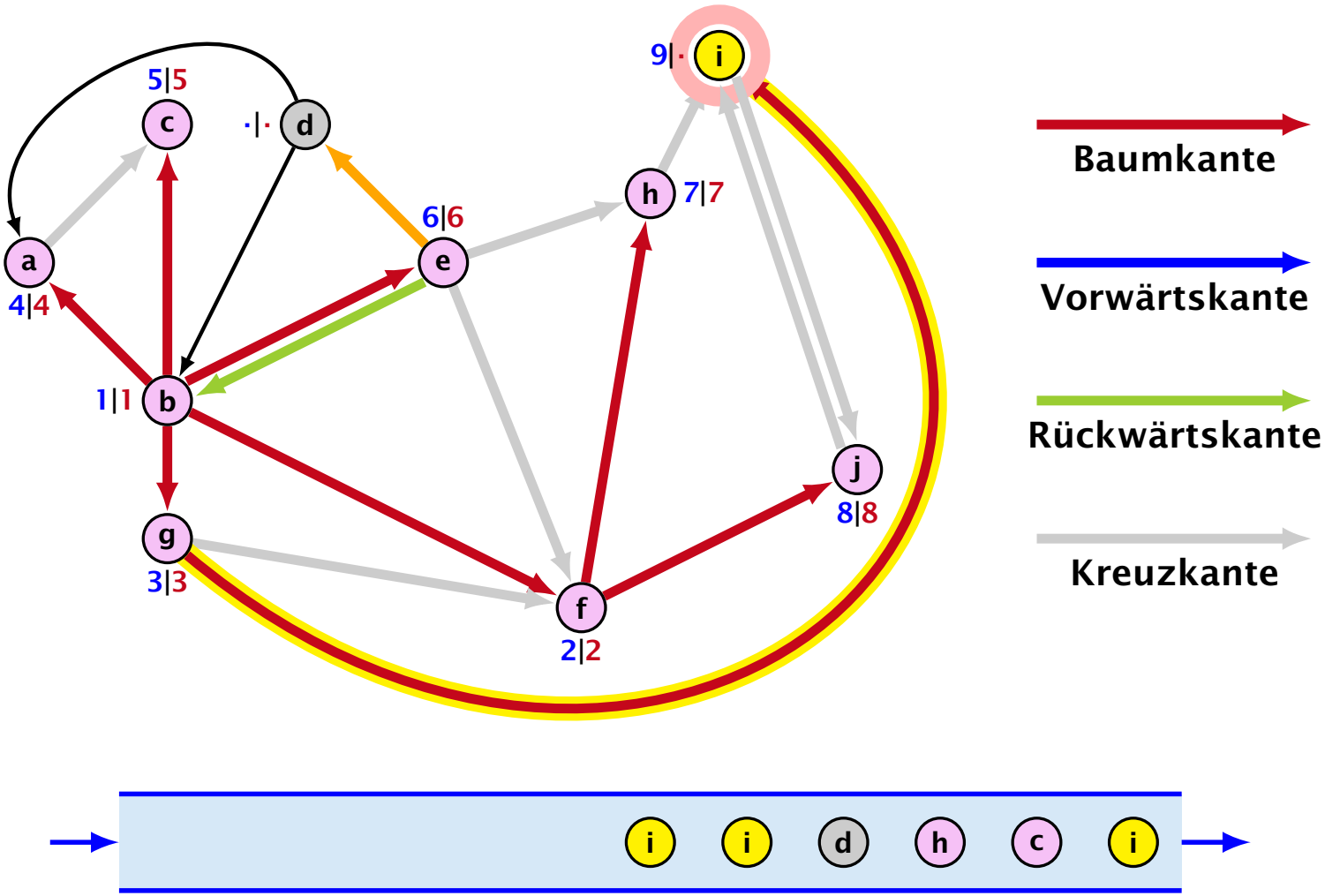
# BFS


9|·
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

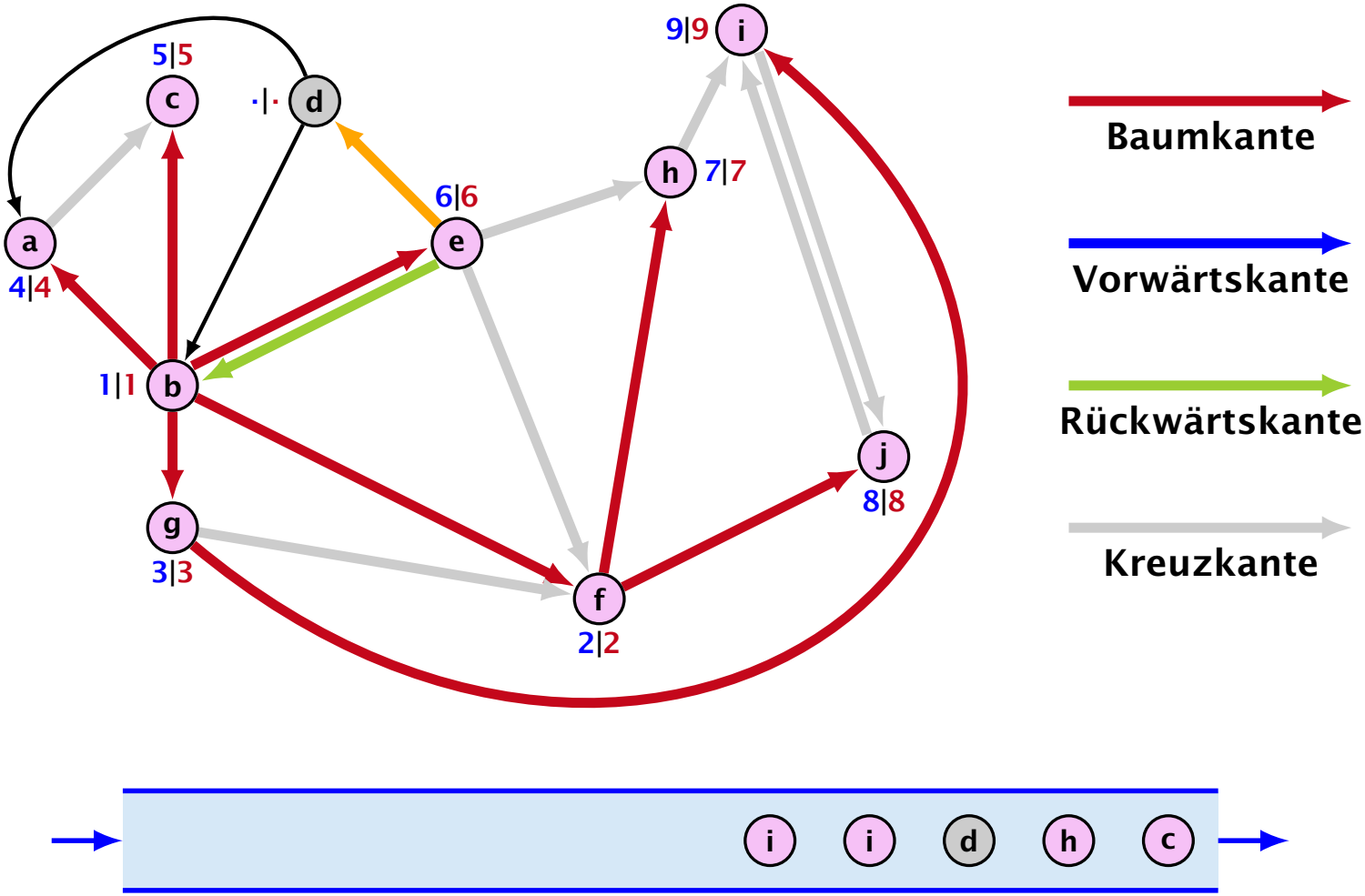
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d
h
c
i

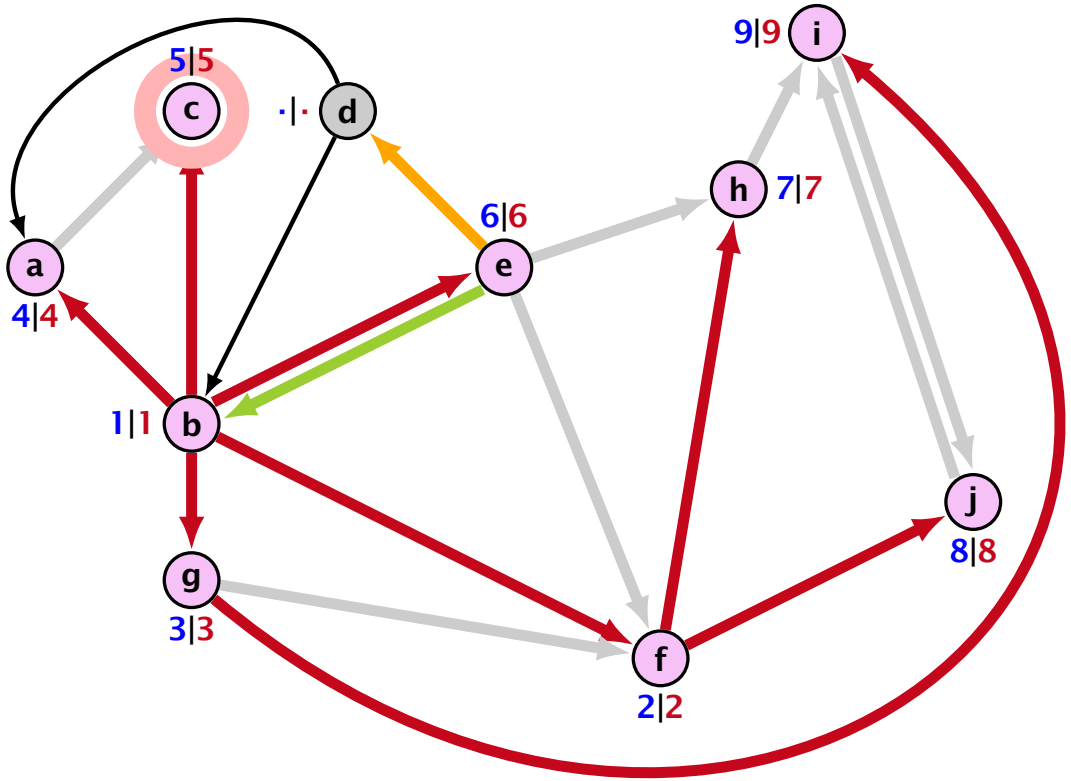
# BFS


9|·
i
5|5
c
·|·
d
h
7|7
6|6
e
a
4|4
1|1
b
j
8|8
g
3|3
f
2|2
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
i
i
d
h
c
i

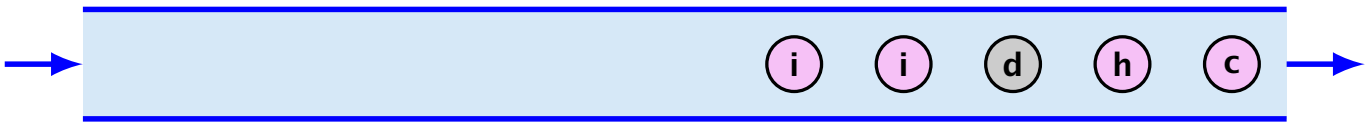
# BFS


9|9
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
i
i
d
h
c

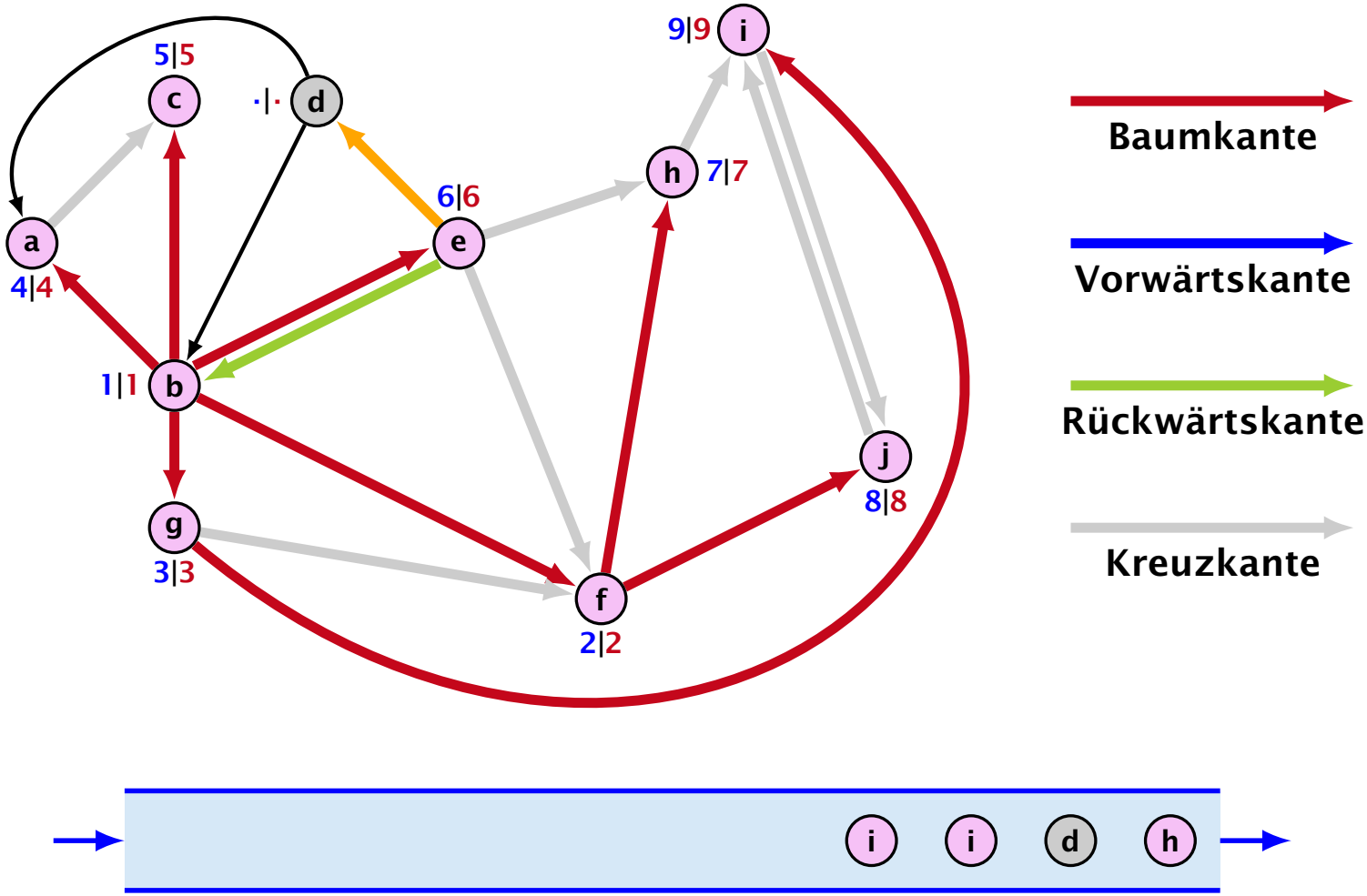
# BFS


9|9
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

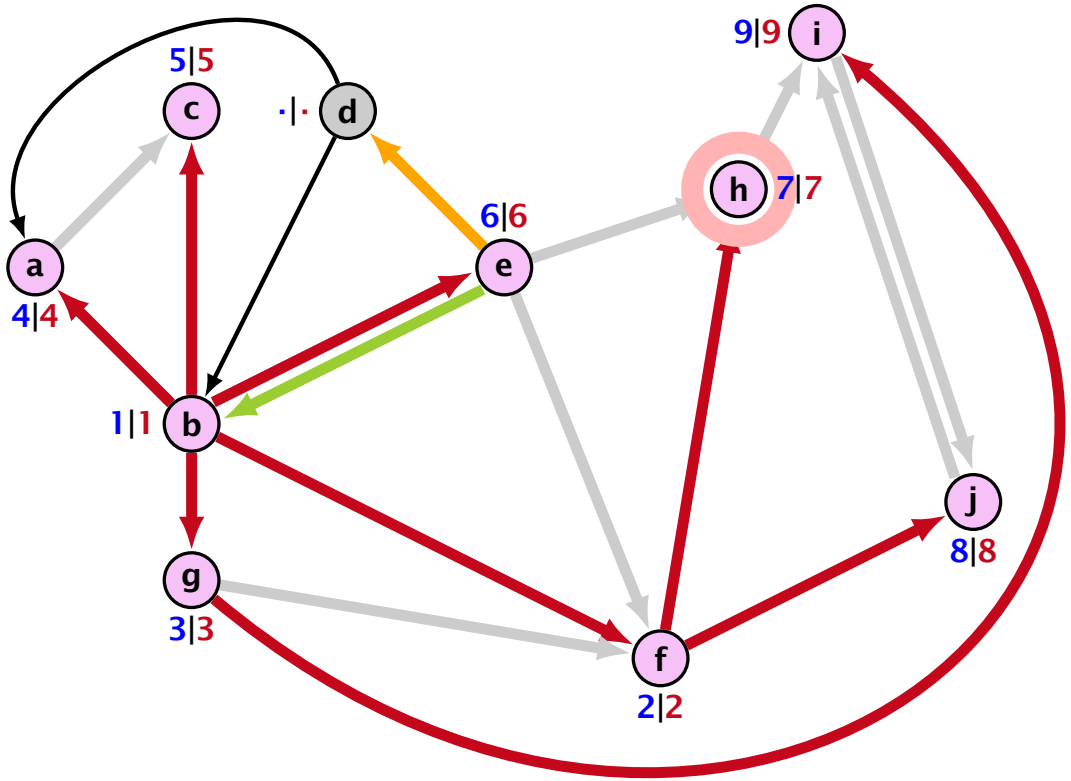
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d
h
c

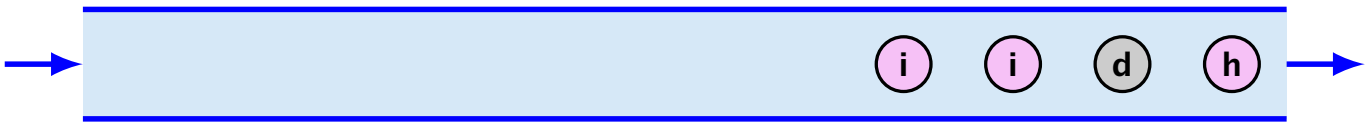
# BFS


9|9
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante
i
i
d
h

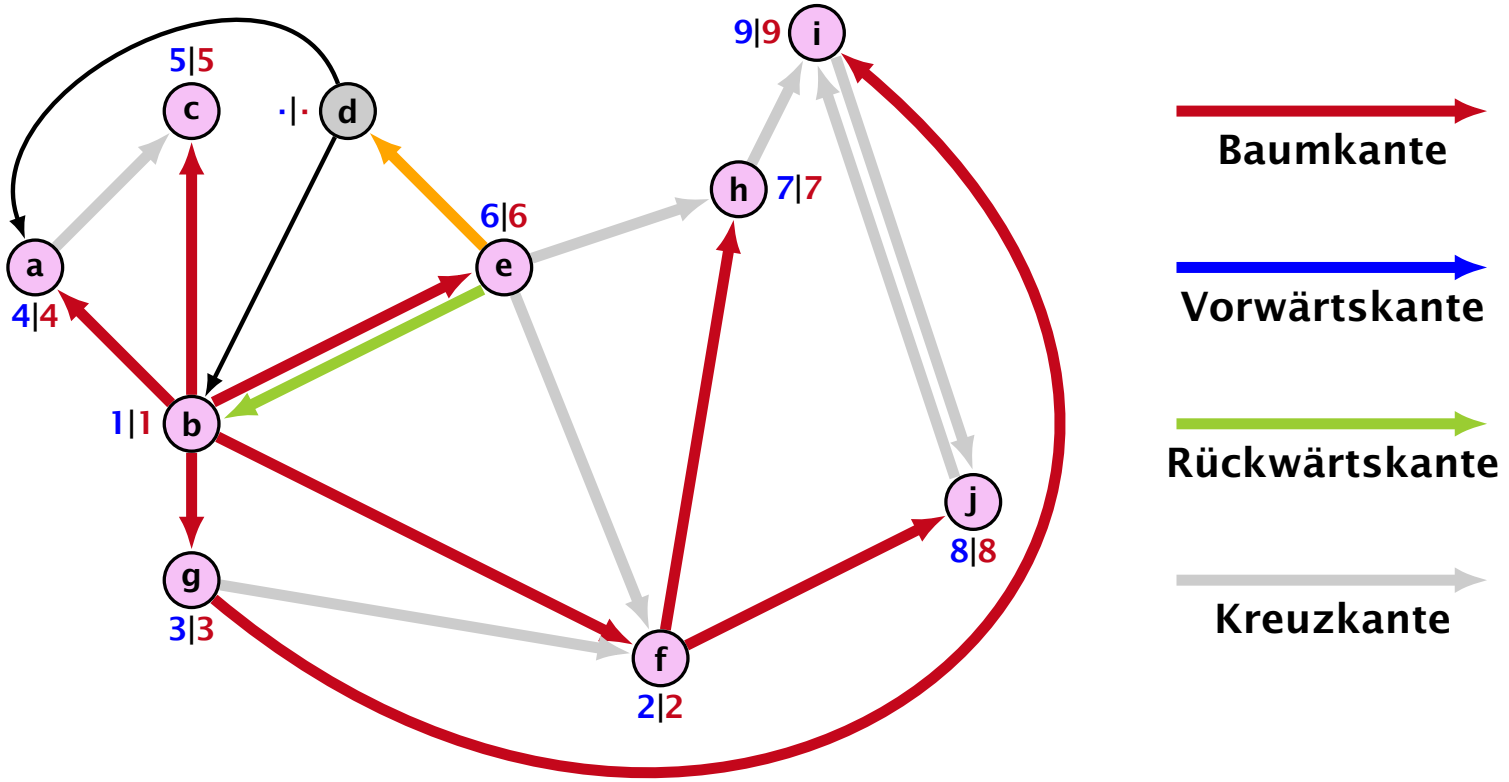
# BFS

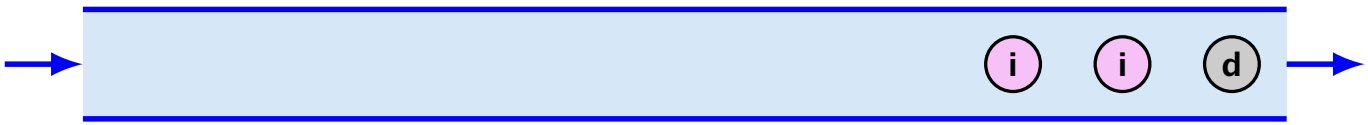

9|9
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

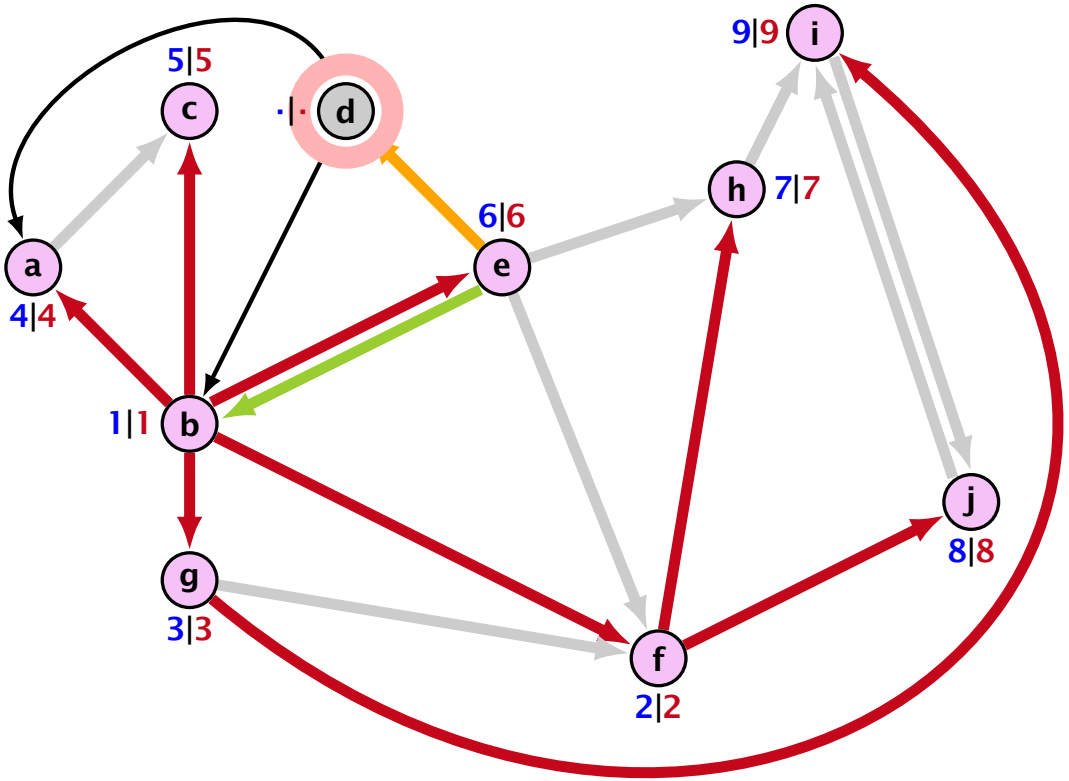

i
i
d
h

# BFS

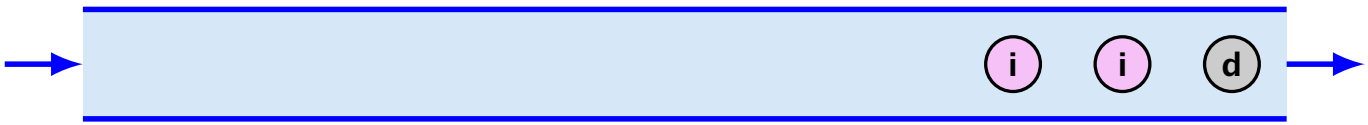

9|9
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d

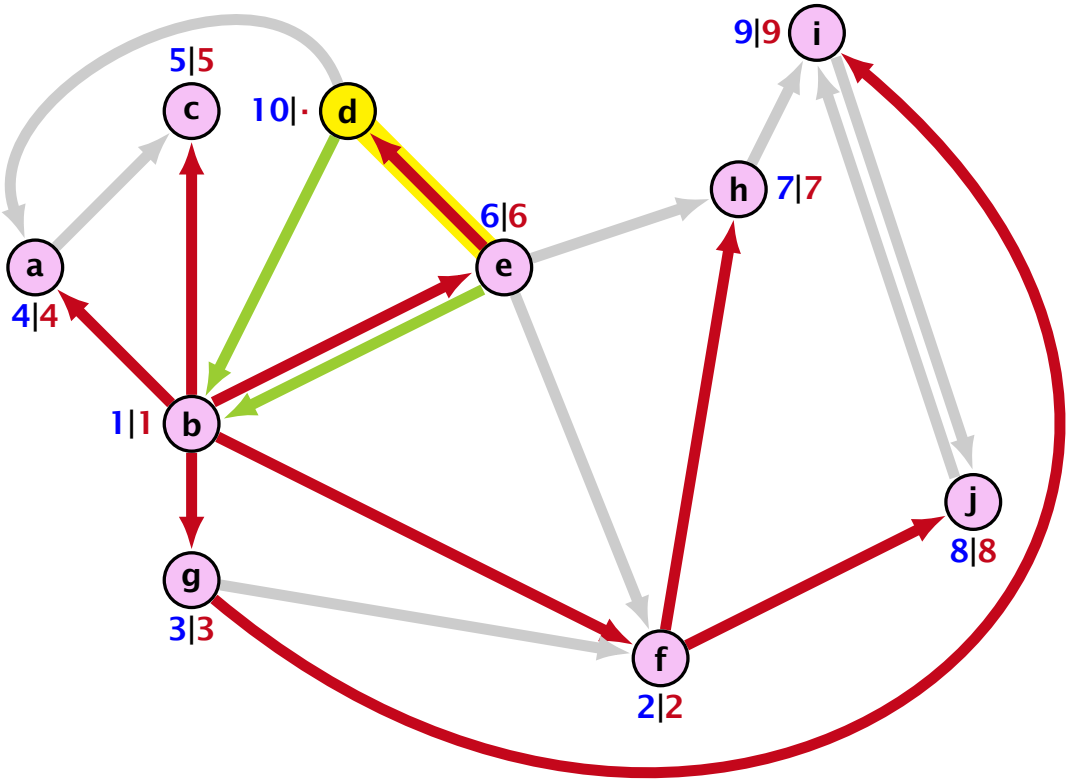
# BFS


9|9
i
5|5
c
·|·
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d

# BFS

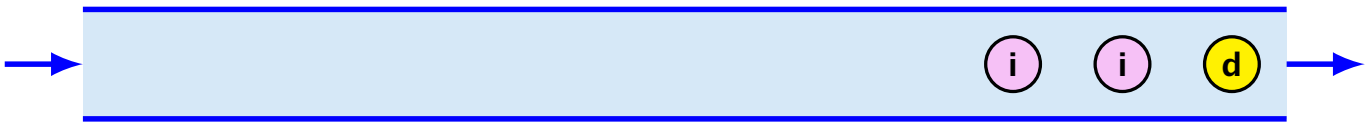



Baumkante

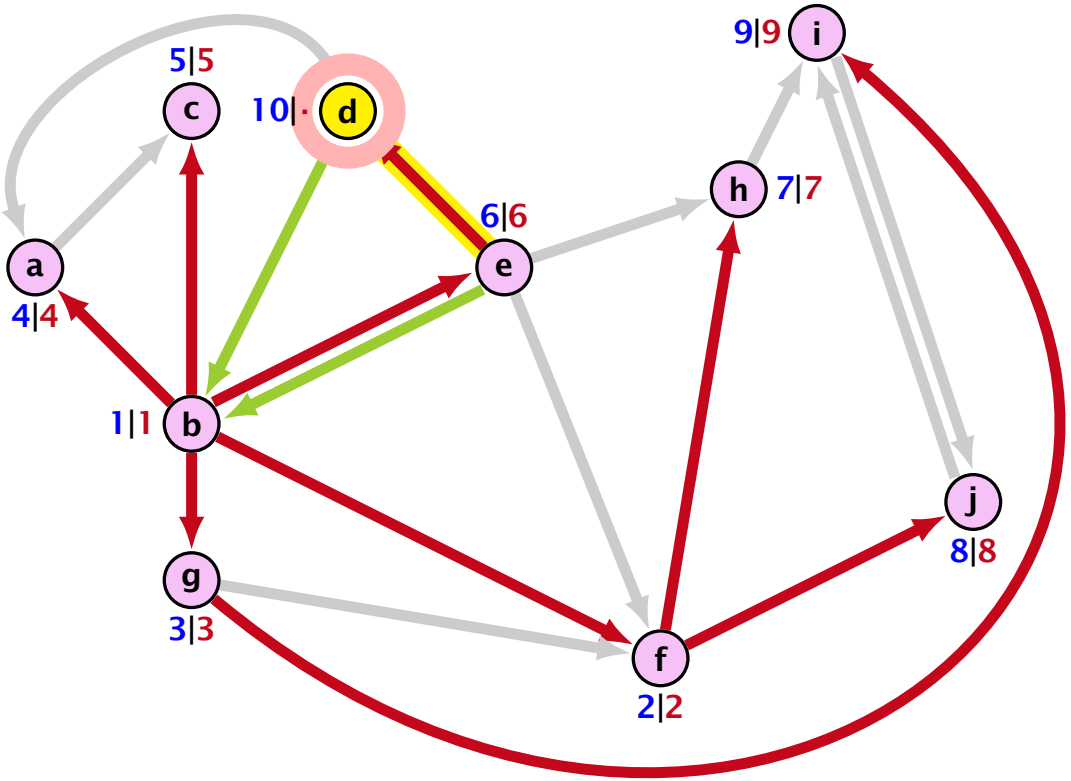
Vorwärtskante

Rückwärtskante

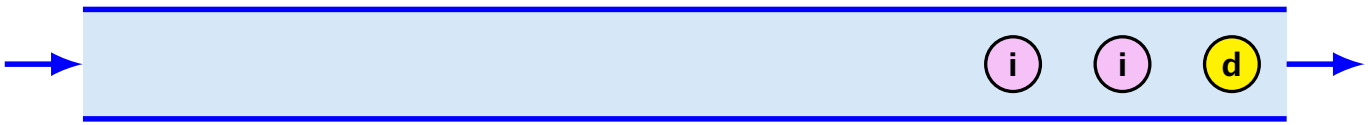
Kreuzkante



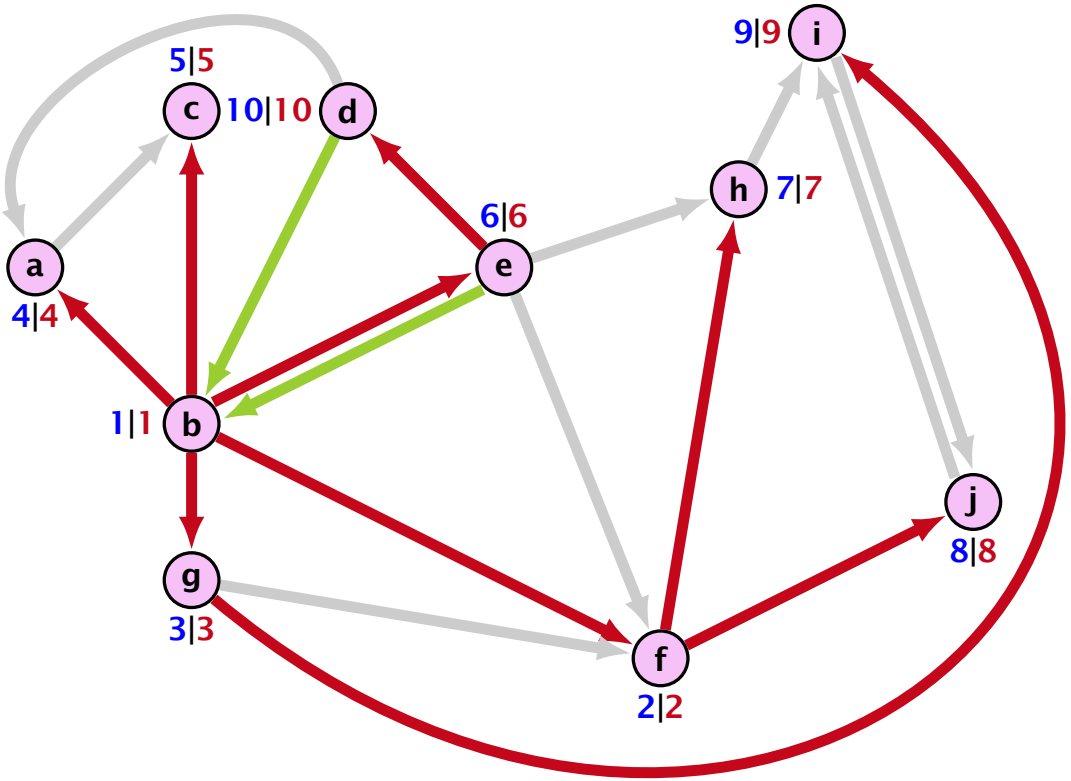
# BFS


a
4|4
b
1|1
c
5|5
d
10|·
e
6|6
f
2|2
g
3|3
h
7|7
i
9|9
j
8|8

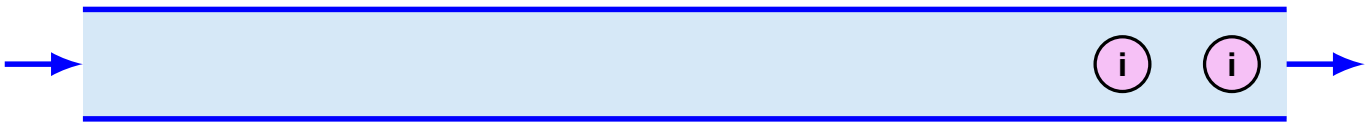
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i
d

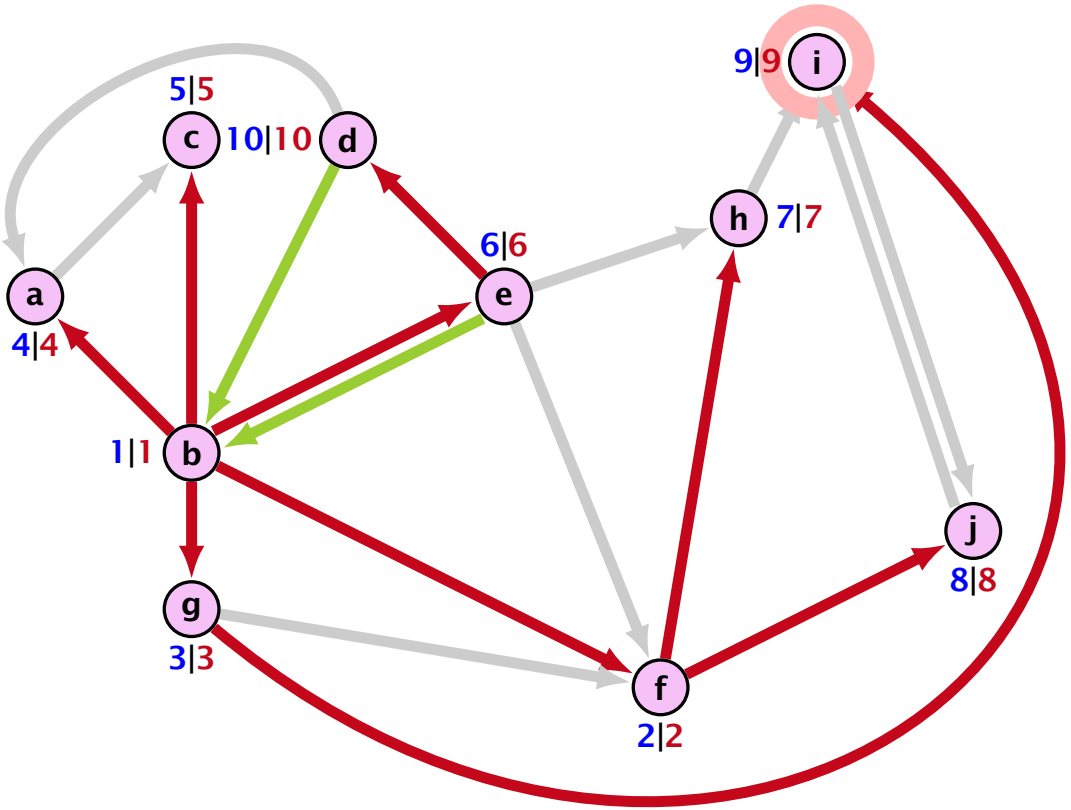
# BFS


9|9
i
5|5
c
10|10
d
h
7|7
6|6
e
a
4|4
1|1
b
j
8|8
g
3|3
f
2|2

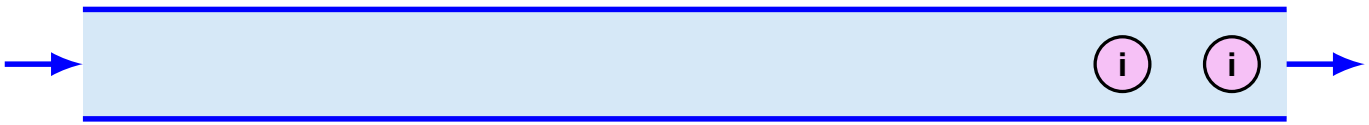
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i

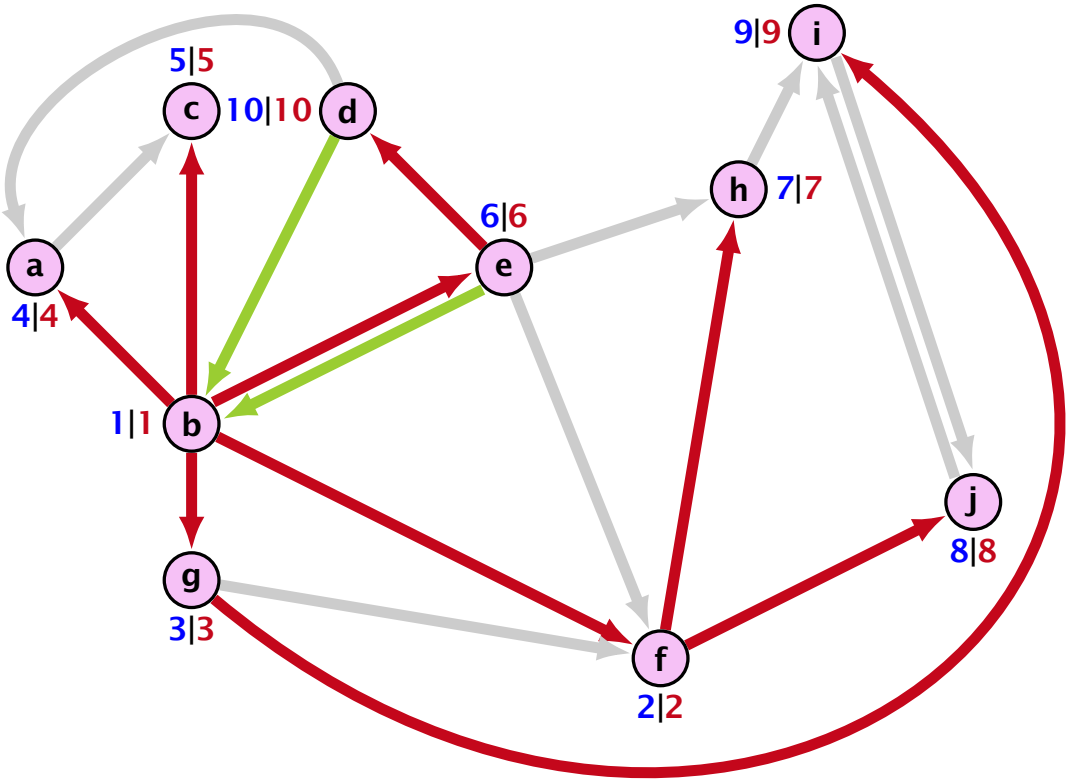
# BFS


9|9
i
5|5
c
10|10
d
h
7|7
6|6
e
a
4|4
1|1
b
j
8|8
g
3|3
f
2|2

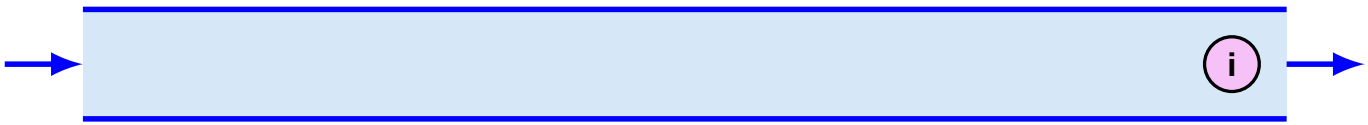
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i
i

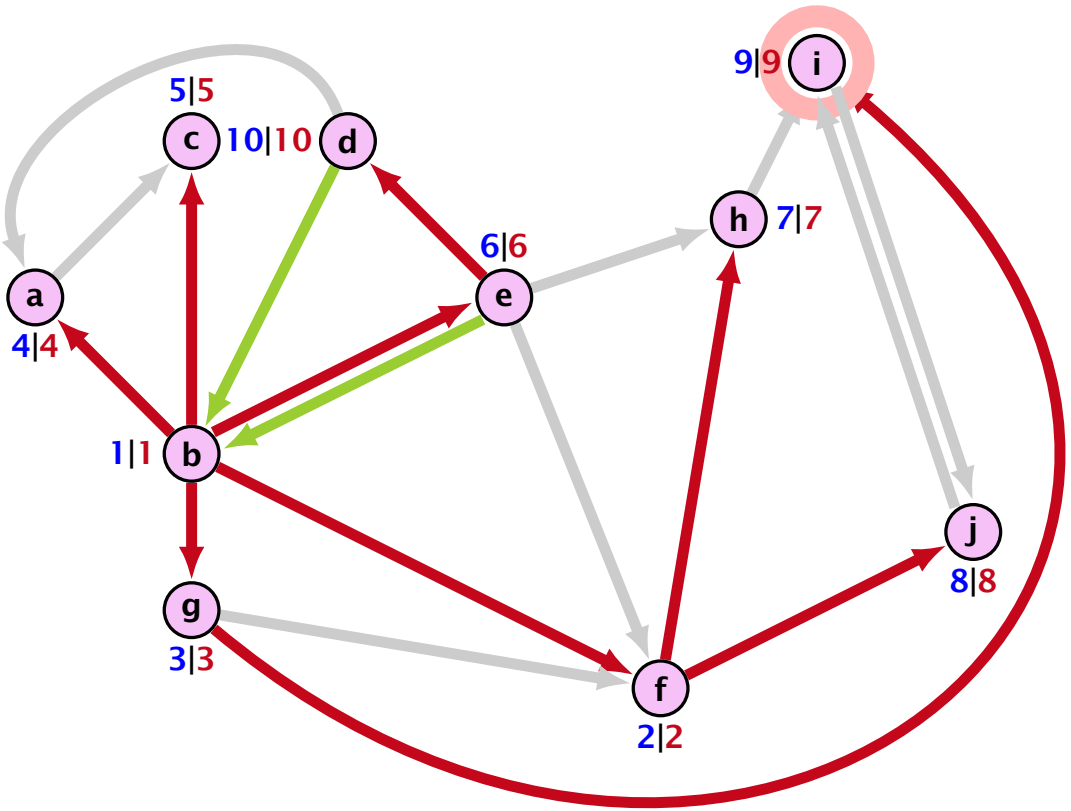
# BFS


9|9
i
5|5
c
10|10
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

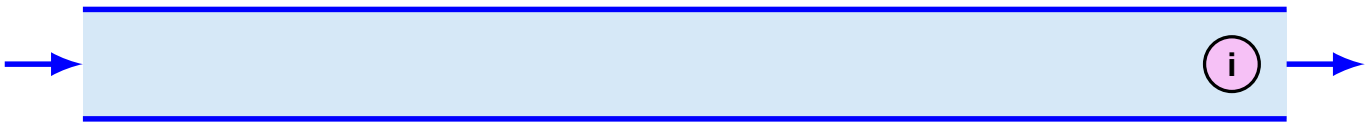
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i

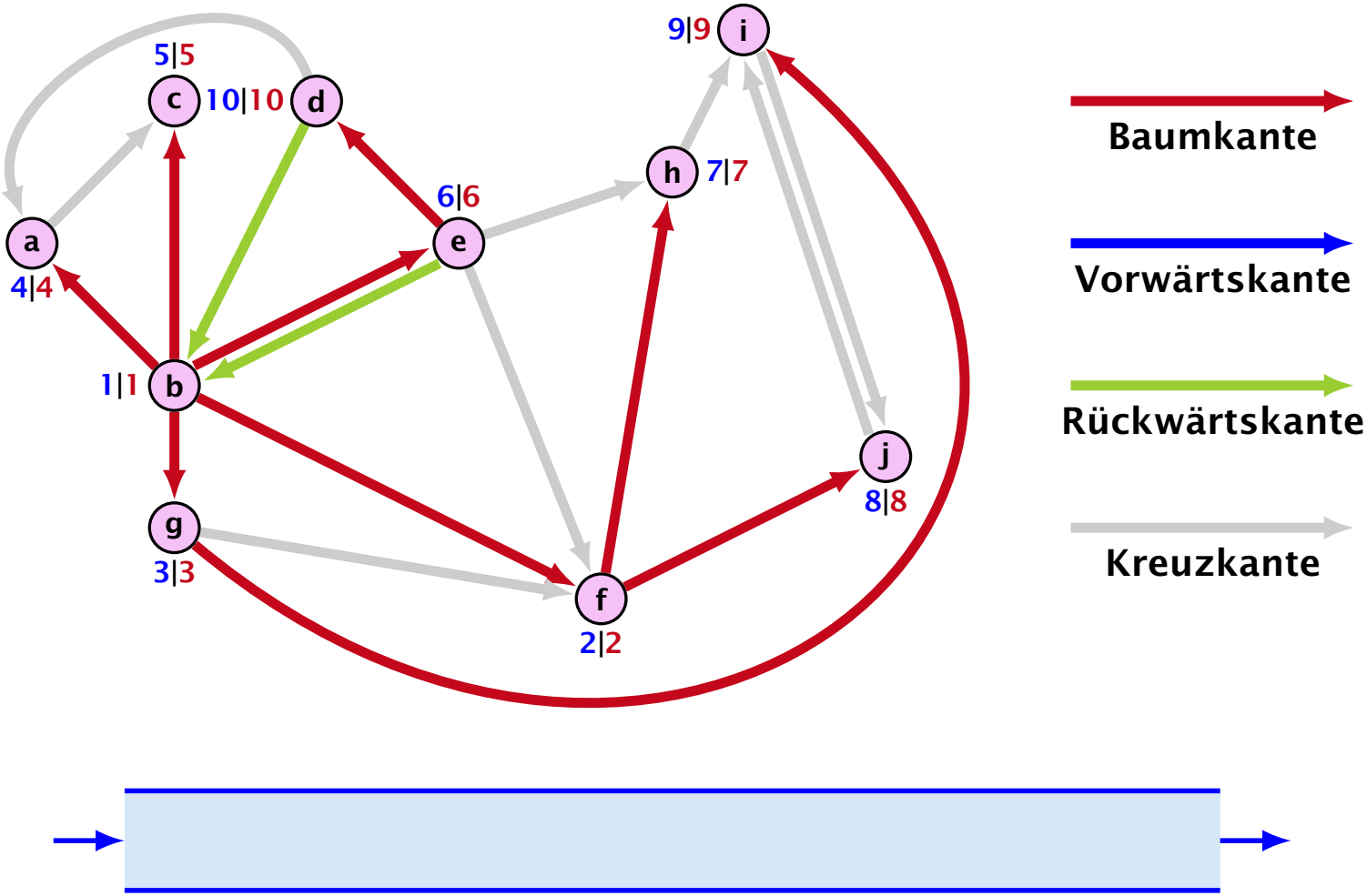
# BFS


9|9
i
5|5
c
10|10
d
h
7|7
6|6
a
e
4|4
1|1
b
j
8|8
g
3|3
f
2|2

Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante


i

# BFS


a
4|4
b
1|1
c
5|5
d
10|10
e
6|6
f
2|2
g
3|3
h
7|7
i
9|9
j
8|8
Baumkante
Vorwärtskante
Rückwärtskante
Kreuzkante

**Beobachtungen**

- `finNum` ist immer gleich `bfsNum` (also überflüssig)
- keine Vorwärtskanten
- Klassifizierung von Kreuzkanten und Rückwärtskanten ist nicht so einfach möglich; wird normalerweise bei BFS auch nicht gemacht
- Komplexität die gleiche wie DFS, da nur ein Stack gegen eine Queue getauscht wurde.

# Breitensuche – mit Queue

```
1  BFSvisit(s)
2      Queue Q; Q.enqueue(s); s->dist = 0;
3      while (!Q.empty()) {
4          v = Q.front();
5          if (v->parent) v->dist = v->parent->dist+1;
6          if (v->state == initial)
7              v->state  = active;
8              v->bfsNum = bfsCount++;
9              foreach x ∈ N(v)
10                 Q.enqueue(x);
11         else
12             v->state  = finished;
13             v->finNum = finCount++;
14             Q.dequeue();
15     }
```

Berechnet die Distanz zu $s$ im Baum. Für einen BFS-Baum ist dies auch die Distanz zu $s$ im ungewichteten Graphen.

**Beweis:**

- Sei $\mathrm{dist}(x)$ der berechnete Wert, und $\mathrm{rdist}(x)$ die wirkliche Distanz zu $s$.
- $\mathrm{rdist}(x) \leq \mathrm{dist}(x)$ folgt, da der Baum einen Pfad zu $s$ der Länge $\mathrm{dist}$ codiert.
- Angenommen es gibt einen Knoten $v$ mit $\mathrm{rdist}(v) < \mathrm{dist}(v)$; sei $v$ ein solcher Knoten mit kleinstem Wert von $\mathrm{dist}(v)$

  Die Knoten werden gemäß der Größe ihrer dist-Werte besucht. (warum?)

  Sei $p_v$, der vom Algorithmus gefundene Vorgänger von $v$ (`v->parent`) und sei $p'_v$ der Vorgänger von $v$ auf einem kürzesten $s$-$v$ Pfad.

  - $\mathrm{dist}(p_v) = \mathrm{rdist}(p_v) = \mathrm{dist}(v) - 1$
  - $\mathrm{dist}(p'_v) = \mathrm{rdist}(p'_v) - 1 < \mathrm{dist}(v) - 1$
  - dann wird aber $p'_v$ zuerst besucht; und dabei würden wir `v->parent` auf $p'_v$ setzen. Widerspruch.

# Kürzeste Wege

**Wie kommt man am schnellsten von A nach B?**

# 9 Kürzeste Wege

Gegeben gewichteter, gerichteter graph $G = (V, E)$, $w : E \to \mathbb{R}$.

- SSSP (single source shortest paths):
  finde kürzesten Weg von Quelle $s$ zu allen anderen Knoten

- APSP (all pairs shortest paths):
  finde kürzesten Weg zwischen allen Knotenpaaren

  Manchmal wird auch von SSSP/APSP geredet wenn man nur die Länge der entsprechenden Wege berechnen möchte.

# 9 Kürzeste Wege

Die Distanz $d(x,y)$ zwischen Knoten $x$ und $y$ ist die Länge eines kürzesten Weges.

Formal:

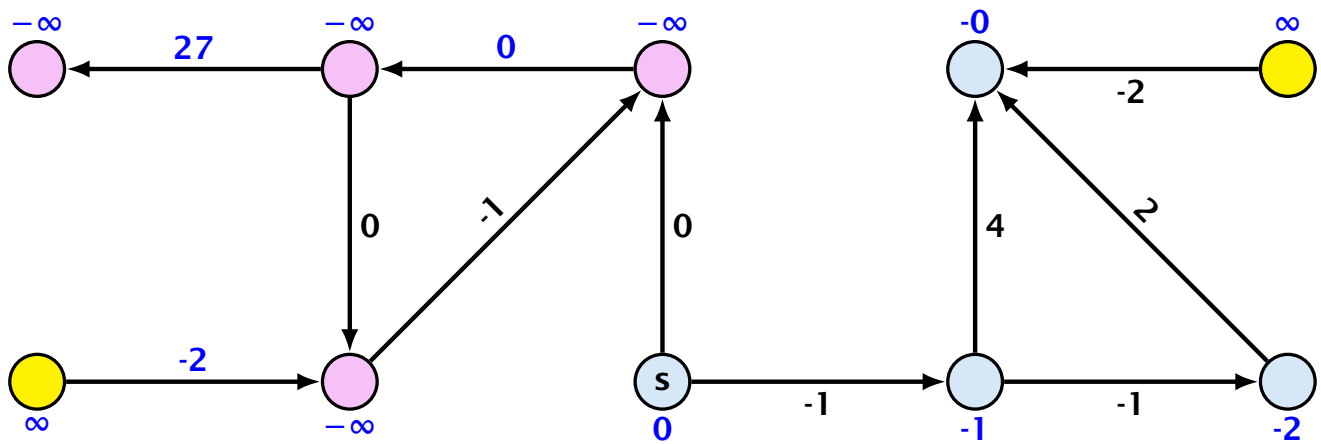
$$d(s,v) = \begin{cases} +\infty & \text{kein Pfad von } s \text{ nach } v \\ -\infty & \text{kein kürzester Pfad von } s \text{ nach } v \\ \min_{p \text{ path from } s \text{ to } v} w(p) & \text{sonst} \end{cases}$$

# 9 Kürzeste Wege

Die Distanz $d(x,y)$ zwischen Knoten $x$ und $y$ ist die Länge eines kürzesten Weges.

Formal:

$$d(s,v) = \begin{cases} +\infty & \text{kein Pfad von } s \text{ nach } v \\ -\infty & \text{kein kürzester Pfad von } s \text{ nach } v \\ \min_{p \text{ path from } s \text{ to } v} w(p) & \text{sonst} \end{cases}$$



# 9 Kürzeste Wege

Varianten des Problems:

- uniform Gewichte (alle 1)
  BFS, Laufzeit $\mathcal{O}(m+n)$
- beliebige Gewichte in einem DAG (**D**irected **A**cyclic **G**raph)
  Topologische Sortierung + Kantenrelaxierung, Laufzeit $\mathcal{O}(m+n)$
- beliebiger Graph mit nichtnegativen Gewichten
  Dijkstras Algorithmus,
  Laufzeit $\mathcal{O}((m+n)\log n)$ oder $\mathcal{O}(m+n\log n)$
- beliebiger Graph mit beliebigen Gewichten
  a) ohne negative Zyklen
  b) mit negativen Zyklen

# Kürzeste Wege

**Idee SSSP:**

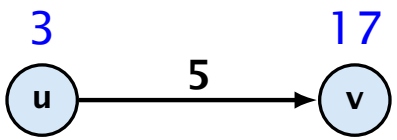
- Jeder Knoten $v$ hat Distanzlabel $\mathrm{dist}(v)$.
- Anfangs ist $\mathrm{dist}(s) = 0$ und $\mathrm{dist}(v) = \infty$, d.h. $\mathrm{dist}(x) \geq d(s,x)$ für alle Knoten $x$.
- Führe Kantenrelaxierungen durch...

# Kürzeste Wege

**Idee SSSP:**

- Jeder Knoten $v$ hat Distanzlabel $\text{dist}(v)$.
- Anfangs ist $\text{dist}(s) = 0$ und $\text{dist}(v) = \infty$, d.h. $\text{dist}(x) \geq d(s,x)$ für alle Knoten $x$.
- Führe Kantenrelaxierungen durch...

**Kantenrelaxierung:**



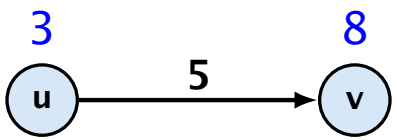

$$\text{dist}(v) := \min\{\text{dist}(v), \text{dist}(u) + w(u,v)\}$$

# Kürzeste Wege

**Idee SSSP:**

- Jeder Knoten $v$ hat Distanzlabel $\text{dist}(v)$.
- Anfangs ist $\text{dist}(s) = 0$ und $\text{dist}(v) = \infty$, d.h. $\text{dist}(x) \geq d(s,x)$ für alle Knoten $x$.
- Führe Kantenrelaxierungen durch...

**Kantenrelaxierung:**




$$\text{dist}(v) := \min\{\text{dist}(v), \text{dist}(u) + w(u,v)\}$$

es gilt immer noch $\text{dist}(v) \geq d(s,v)$...

# Kürzeste Wege

**Beobachtung:**

- Es gilt immer $\mathrm{dist}(v) \geq d(s,v)$ für alle Knoten.

**Hoffnung:**
Wenn wir genug Relaxierungen durchführen haben wir die richtigen Distanzen, da Distanzlabel nur kleiner werden.

Bei negativen Kreisen funktioniert das nicht.

Annahme: keine negativen Kreise.
$\Longrightarrow$ die kürzesten Pfade haben endlich viele „hops“.

# Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., $e_5$ und $e_{10}$ könnten z.B. die gleiche Kante bezeichnen.

Sei $\mathcal{R} = R_1, R_2, R_3, \ldots$ eine Folge von Kantenrelaxierungen, wobei die $i$-te Relaxierung $R_i$ auf Kante $e_i$ operiert.

Sei $p = (s = v_0, v_1, \ldots, v_k = x)$ ein kürzester Pfad von $s$ nach $x$, und sei $a_i = (v_{i-1}, v_i)$ die $i$-te Kante dieses Pfades ($1 \leq i \leq k$).

**Beobachtung**

Falls ein $k$-elementige Teilfolge $S = S_1, S_2, \ldots$ von $\mathcal{R}$ existiert bei der $S_i$ auf Kante $a_i$ operiert hat der Knoten $x$ nach Abarbeitung von $\mathcal{R}$ das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem $S_i$ durchgeführt ist hat $v_i$ das korrekte Distanzlabel;

# Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., $e_5$ und $e_{10}$ könnten z.B. die gleiche Kante bezeichnen.

Sei $\mathcal{R} = R_1, R_2, R_3, \ldots$ eine Folge von Kantenrelaxierungen, wobei die $i$-te Relaxierung $R_i$ auf Kante $e_i$ operiert.

Sei $p = (s = v_0, v_1, \ldots, v_k = x)$ ein kürzester Pfad von $s$ nach $x$, und sei $a_i = (v_{i-1}, v_i)$ die $i$-te Kante dieses Pfades ($1 \leq i \leq k$).

**Beobachtung**
Falls ein $k$-elementige Teilfolge $S = S_1, S_2, \ldots$ von $\mathcal{R}$ existiert bei der $S_i$ auf Kante $a_i$ operiert hat der Knoten $x$ nach Abarbeitung von $\mathcal{R}$ das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem $S_i$ durchgeführt ist hat $v_i$ das korrekte Distanzlabel;

# Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., $e_5$ und $e_{10}$ könnten z.B. die gleiche Kante bezeichnen.

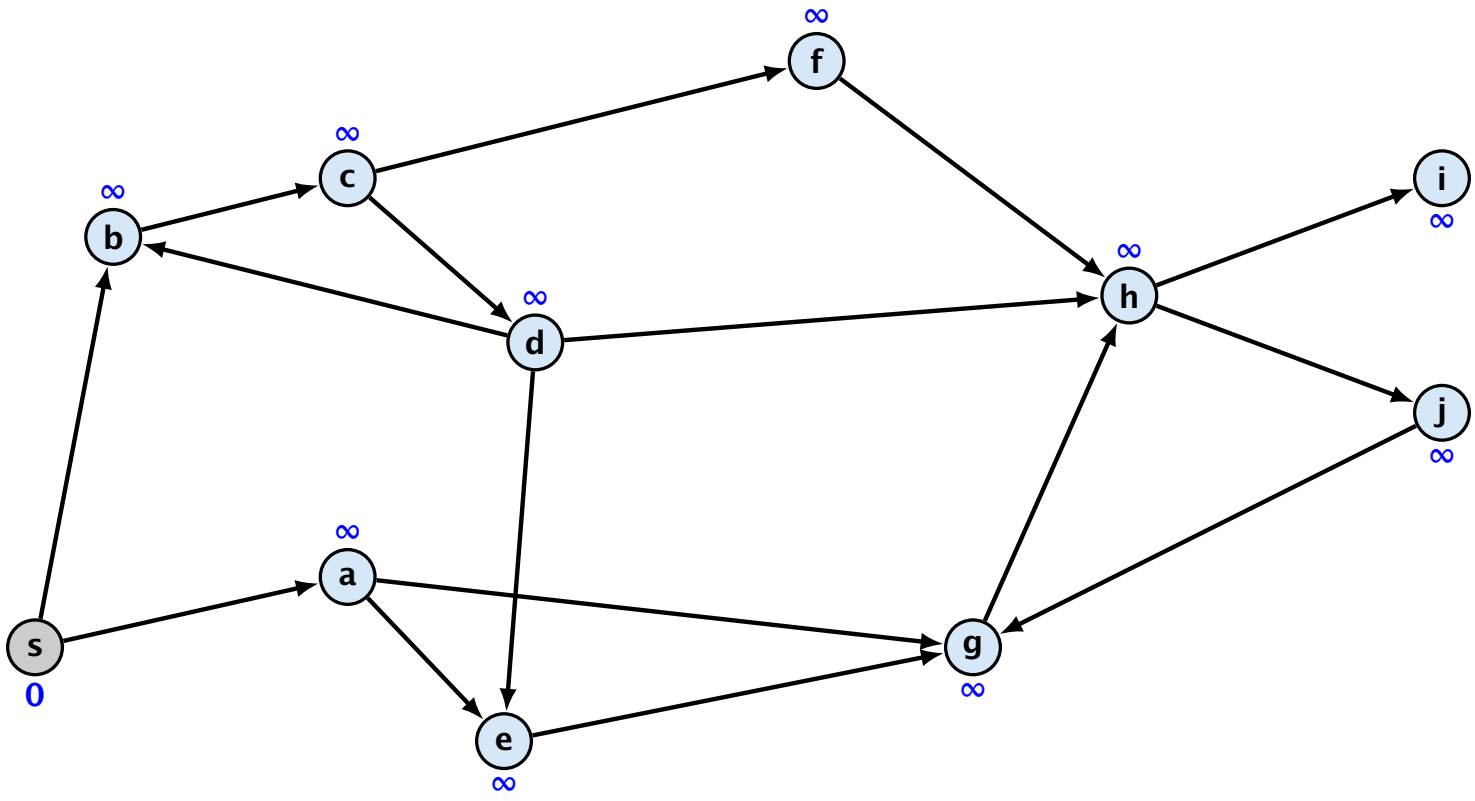
Sei $\mathcal{R} = R_1, R_2, R_3, \ldots$ eine Folge von Kantenrelaxierungen, wobei die $i$-te Relaxierung $R_i$ auf Kante $e_i$ operiert.

Sei $p = (s = v_0, v_1, \ldots, v_k = x)$ ein kürzester Pfad von $s$ nach $x$, und sei $a_i = (v_{i-1}, v_i)$ die $i$-te Kante dieses Pfades ($1 \leq i \leq k$).

**Beobachtung**
Falls ein $k$-elementige Teilfolge $S = S_1, S_2, \ldots$ von $\mathcal{R}$ existiert bei der $S_i$ auf Kante $a_i$ operiert hat der Knoten $x$ nach Abarbeitung von $\mathcal{R}$ das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem $S_i$ durchgeführt ist hat $v_i$ das korrekte Distanzlabel;

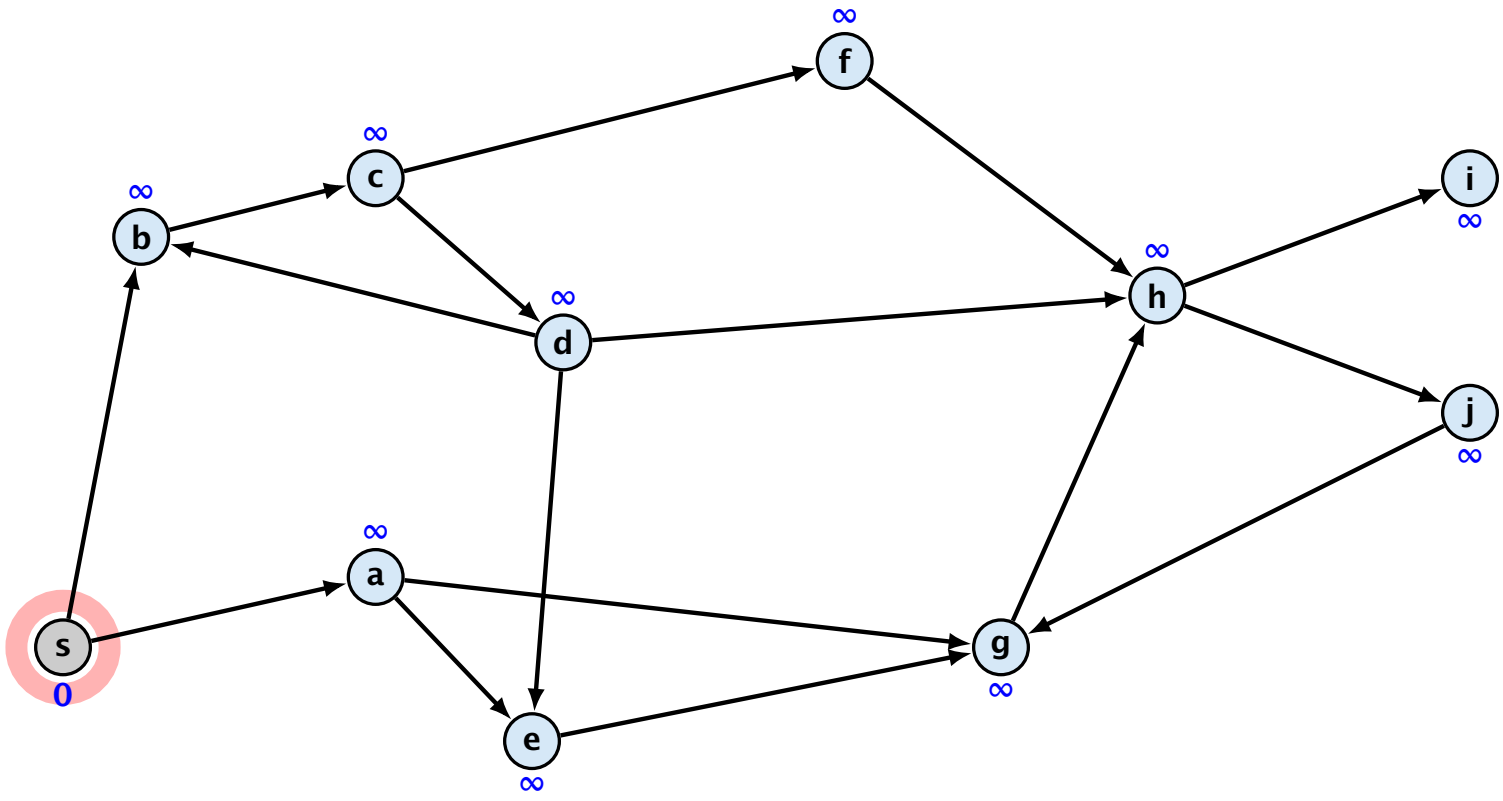
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

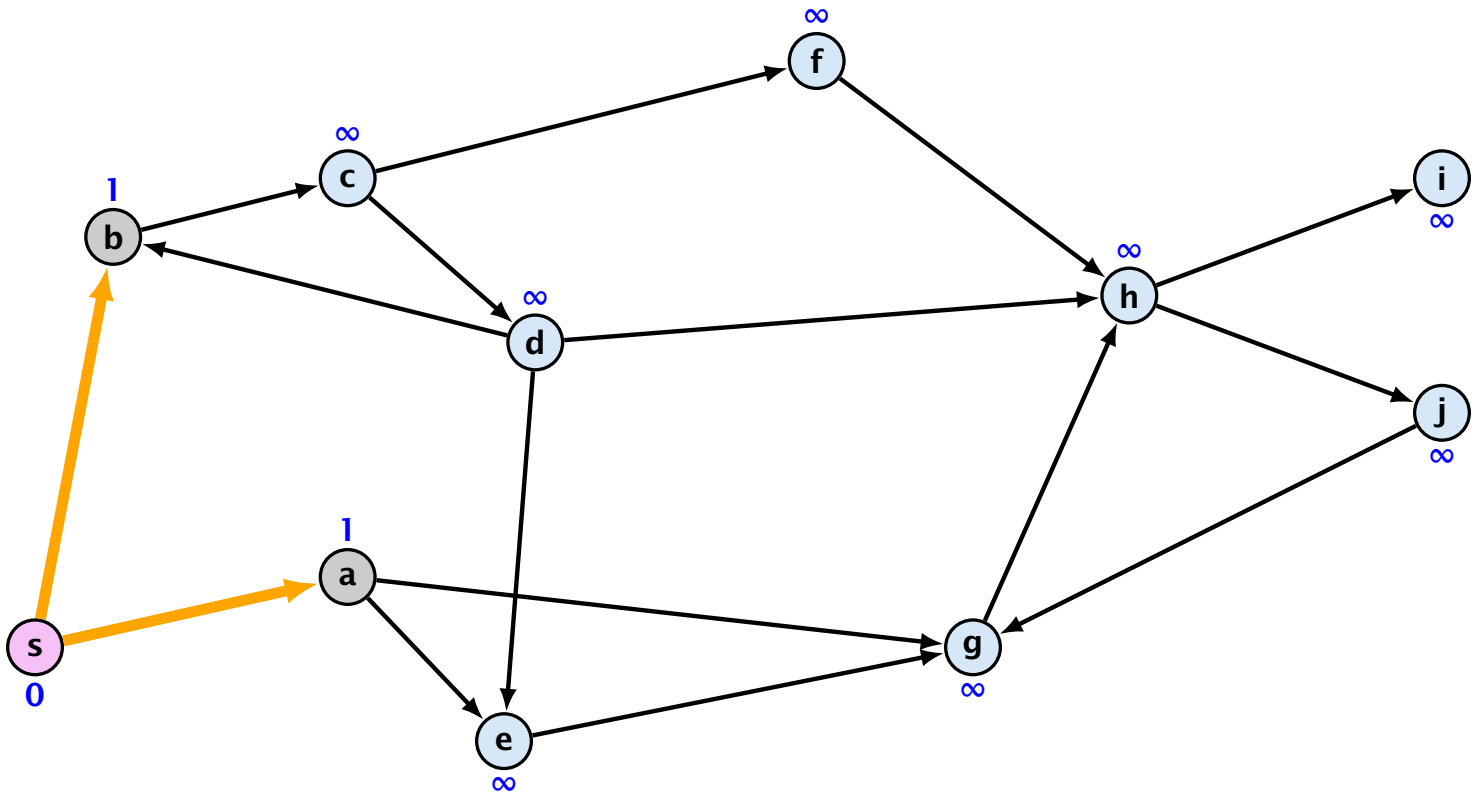
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

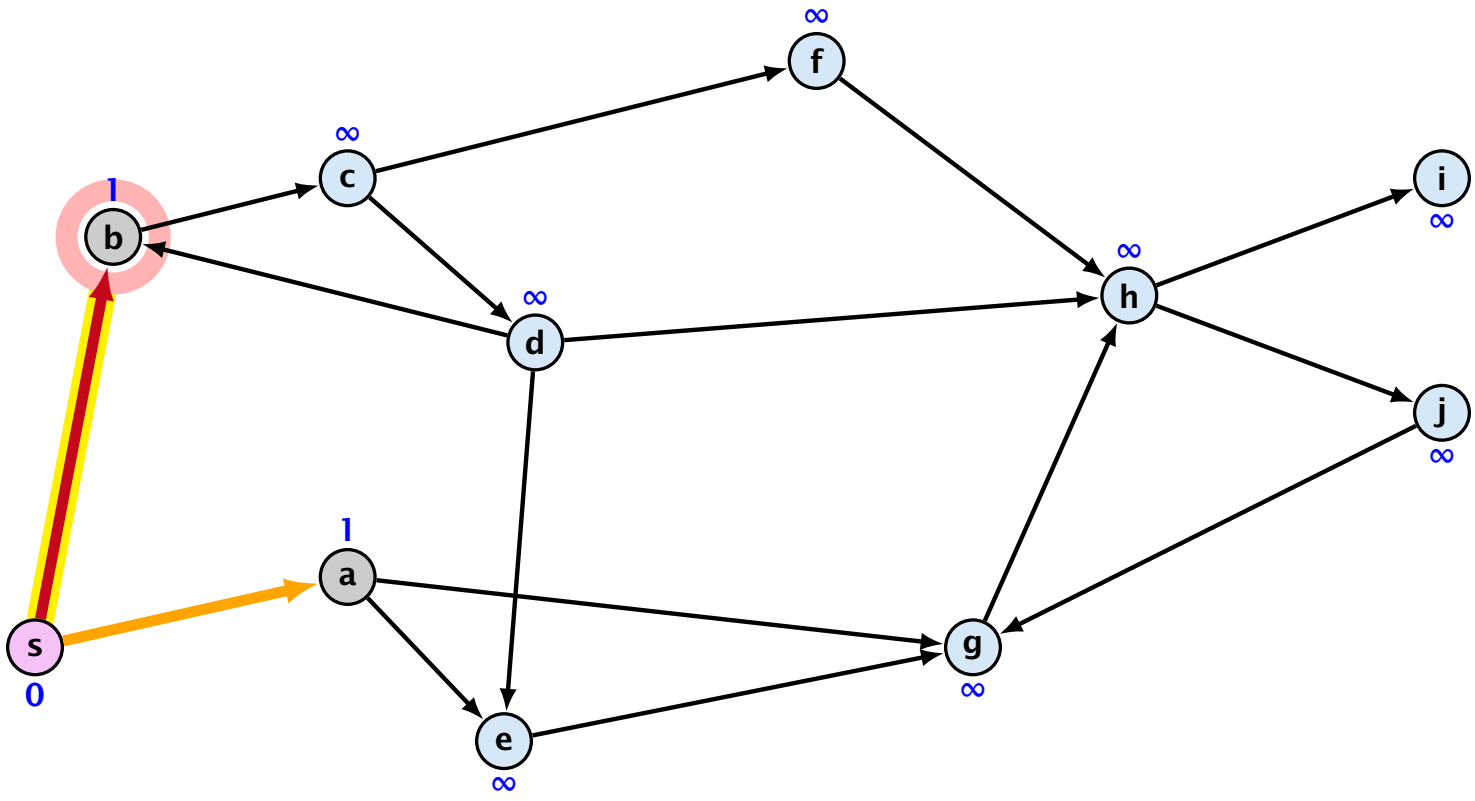
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

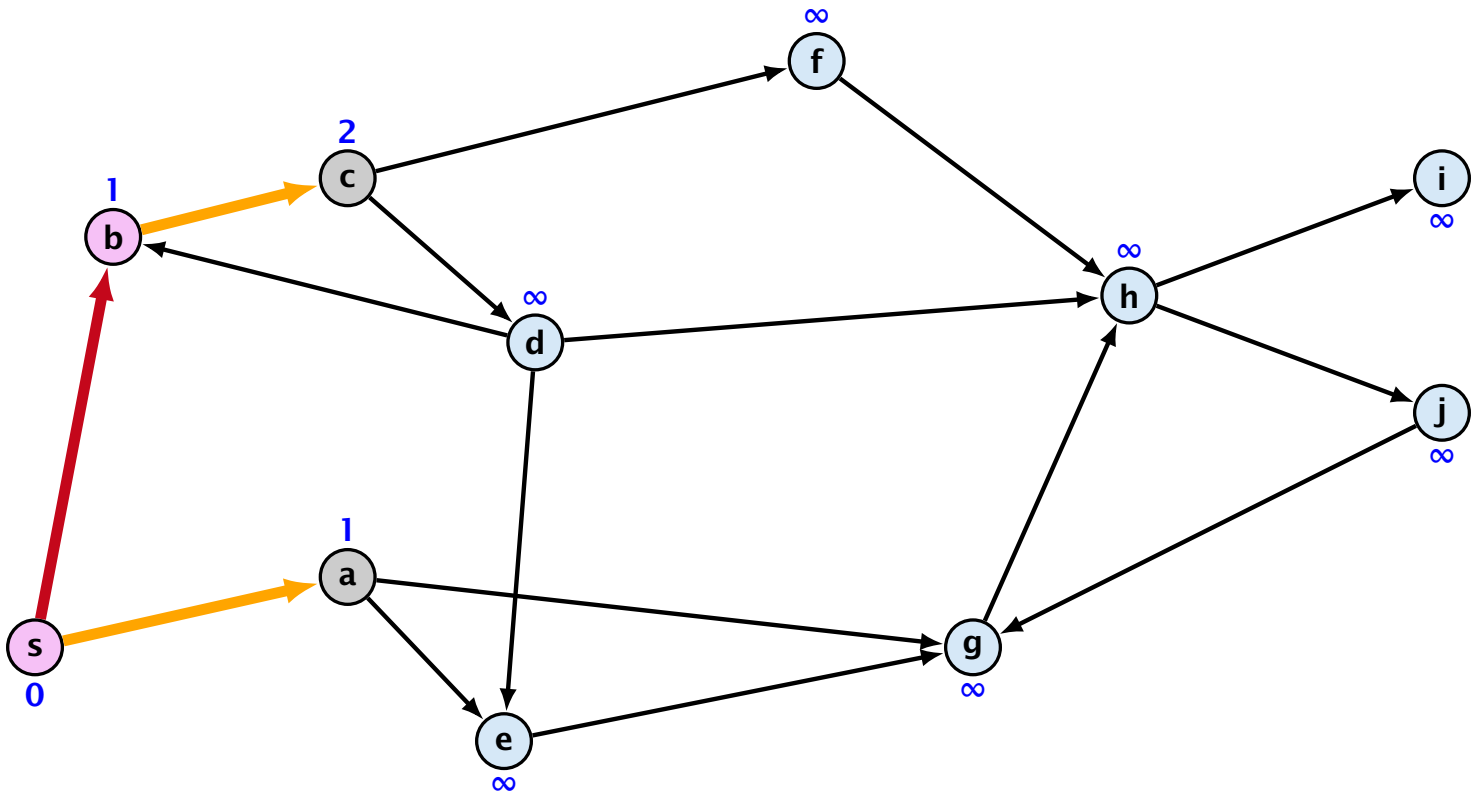
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

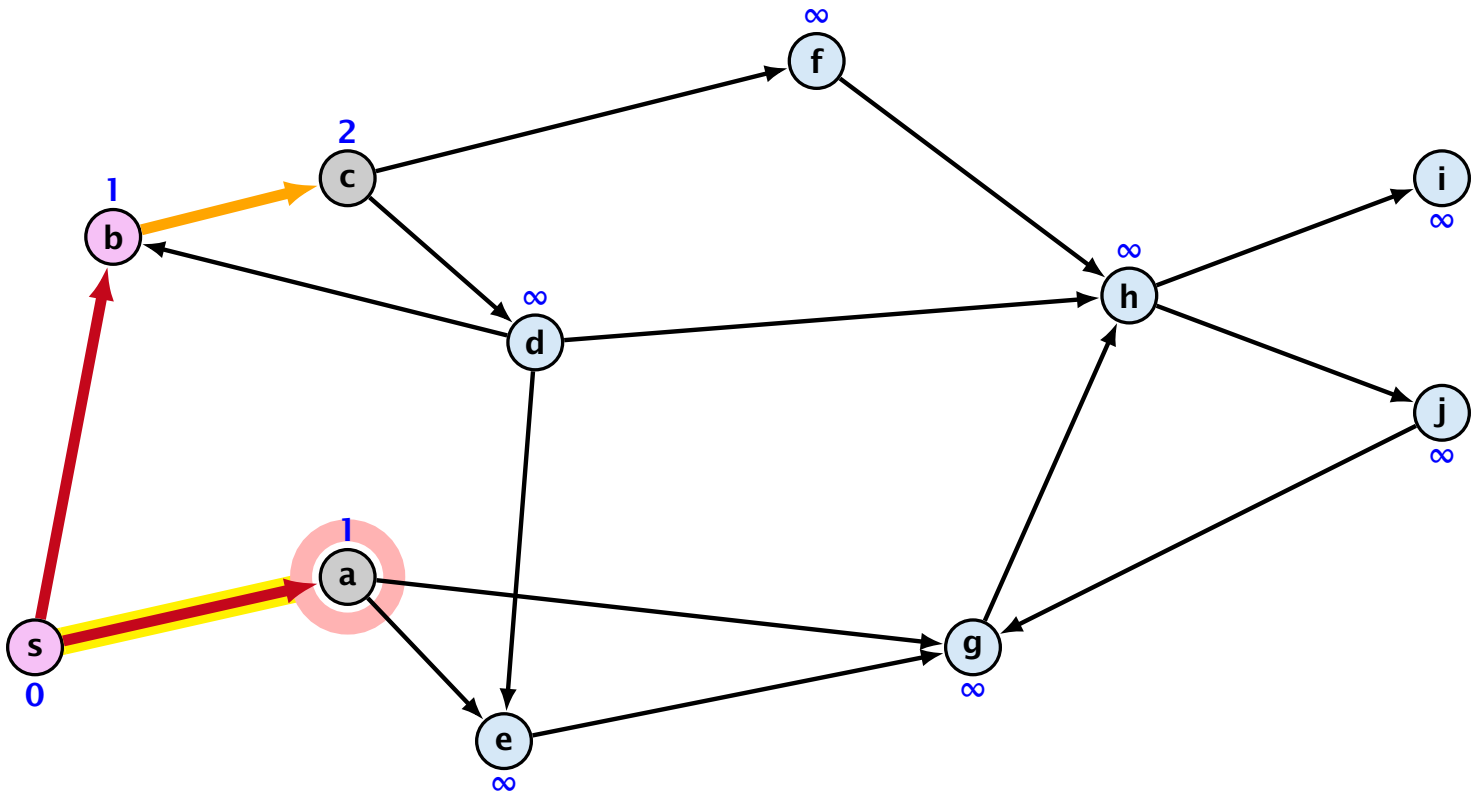
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

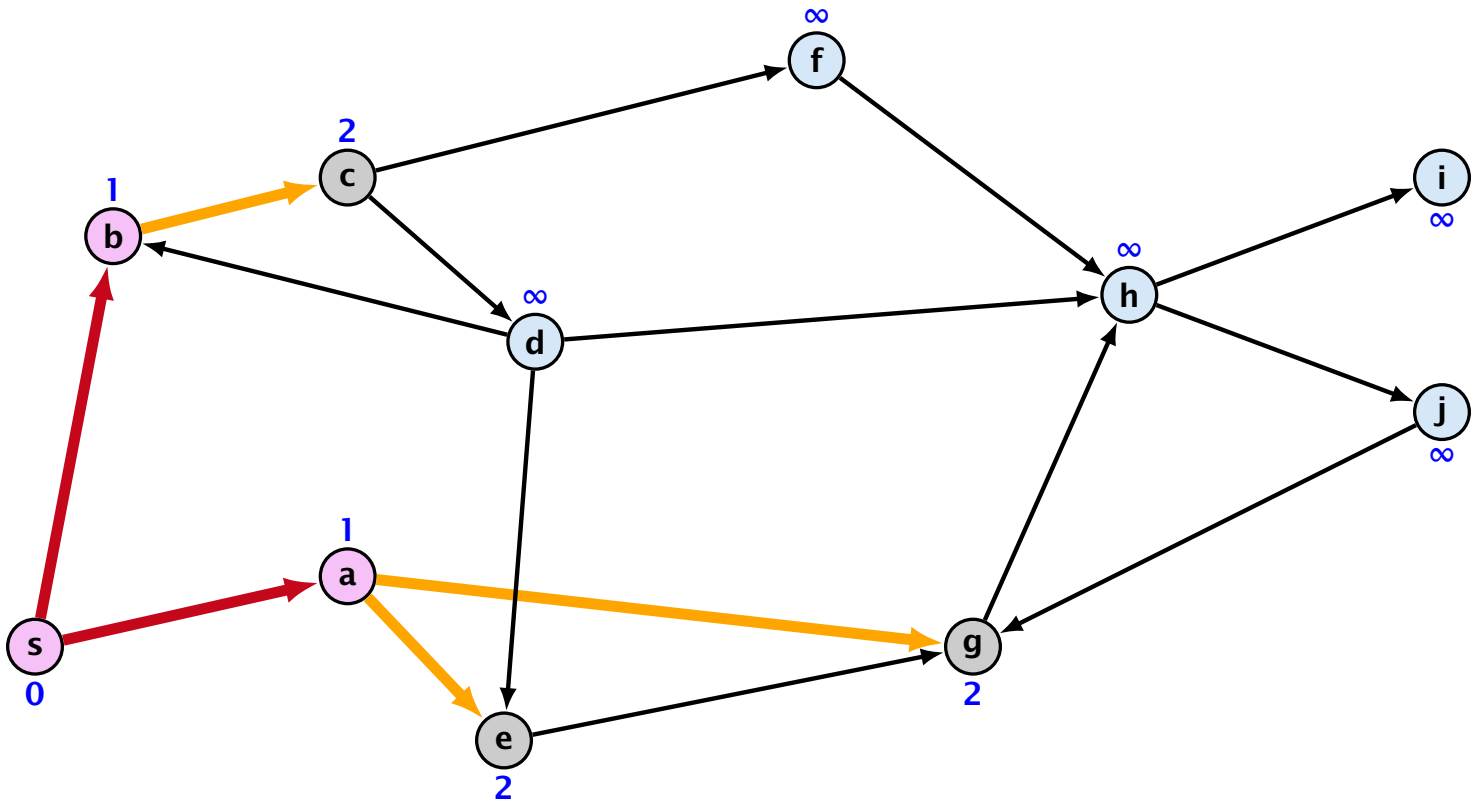
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

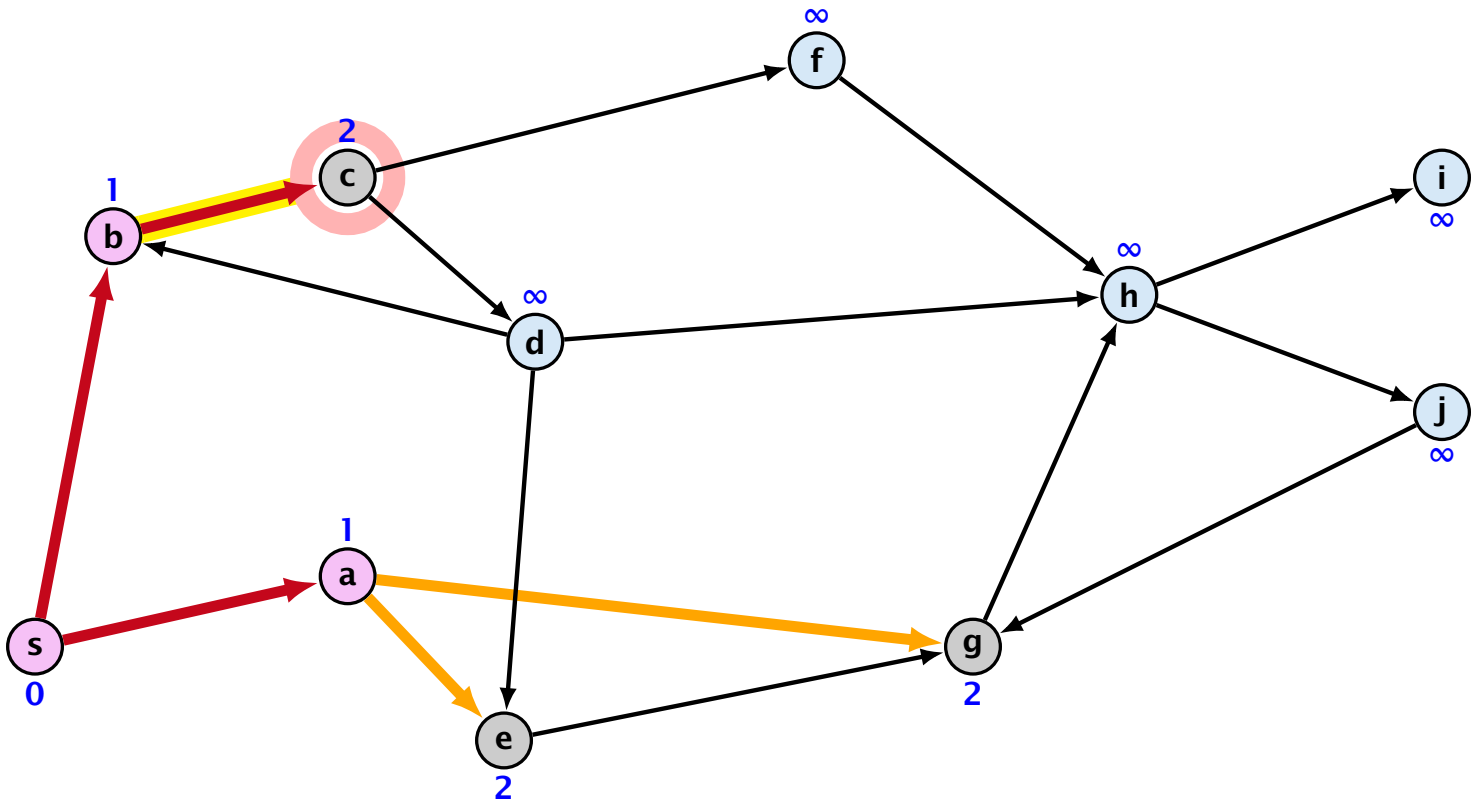
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

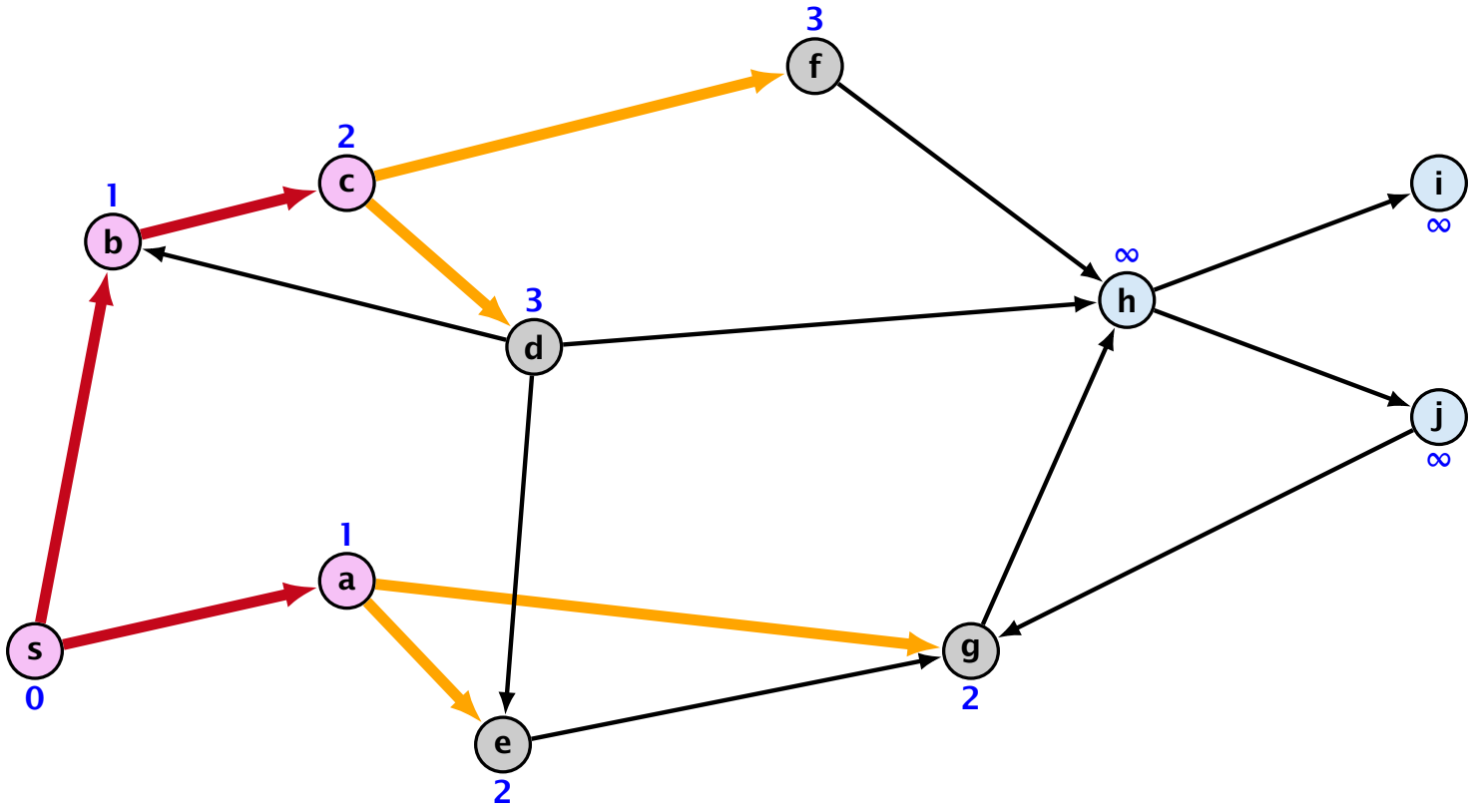
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

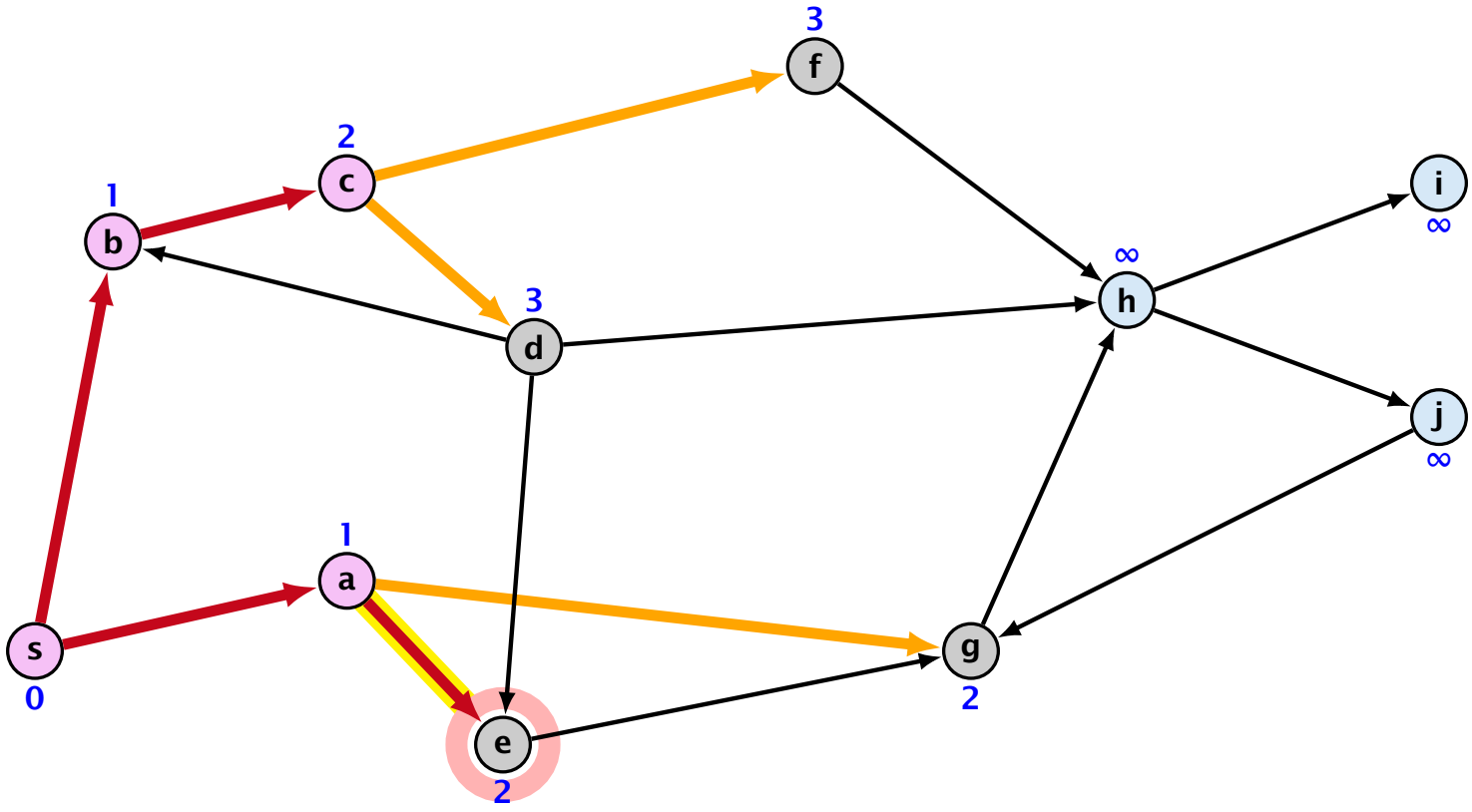
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

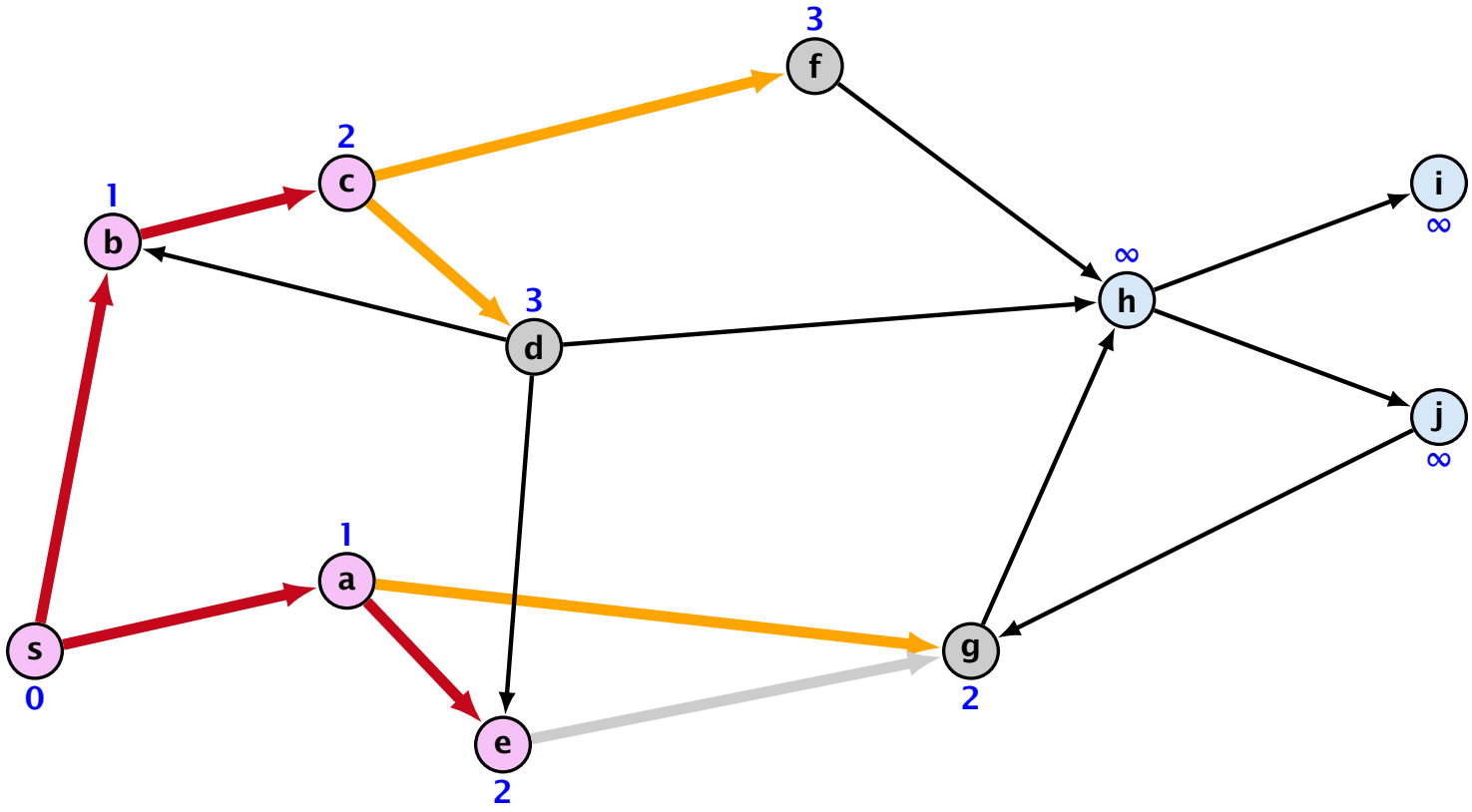
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

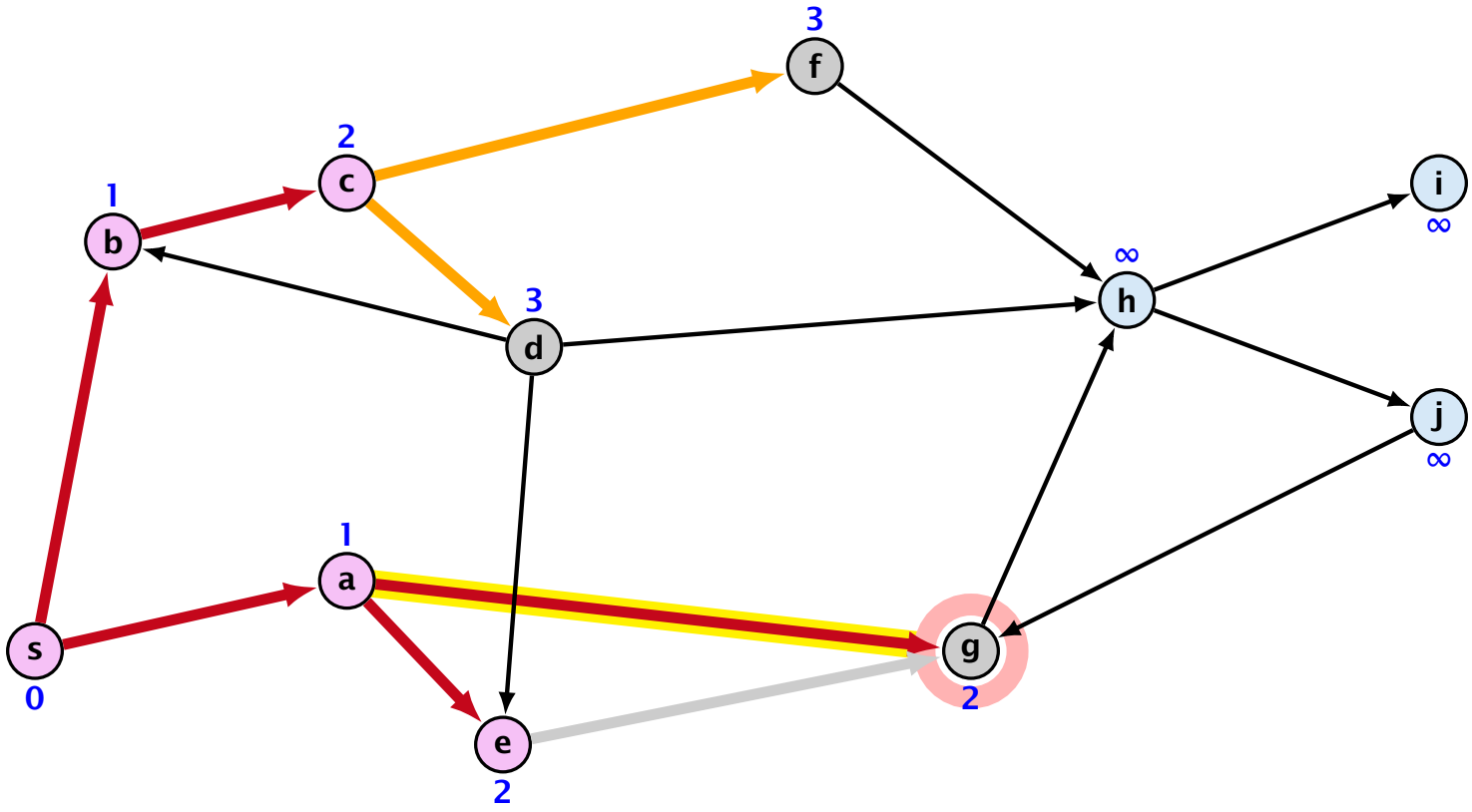
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

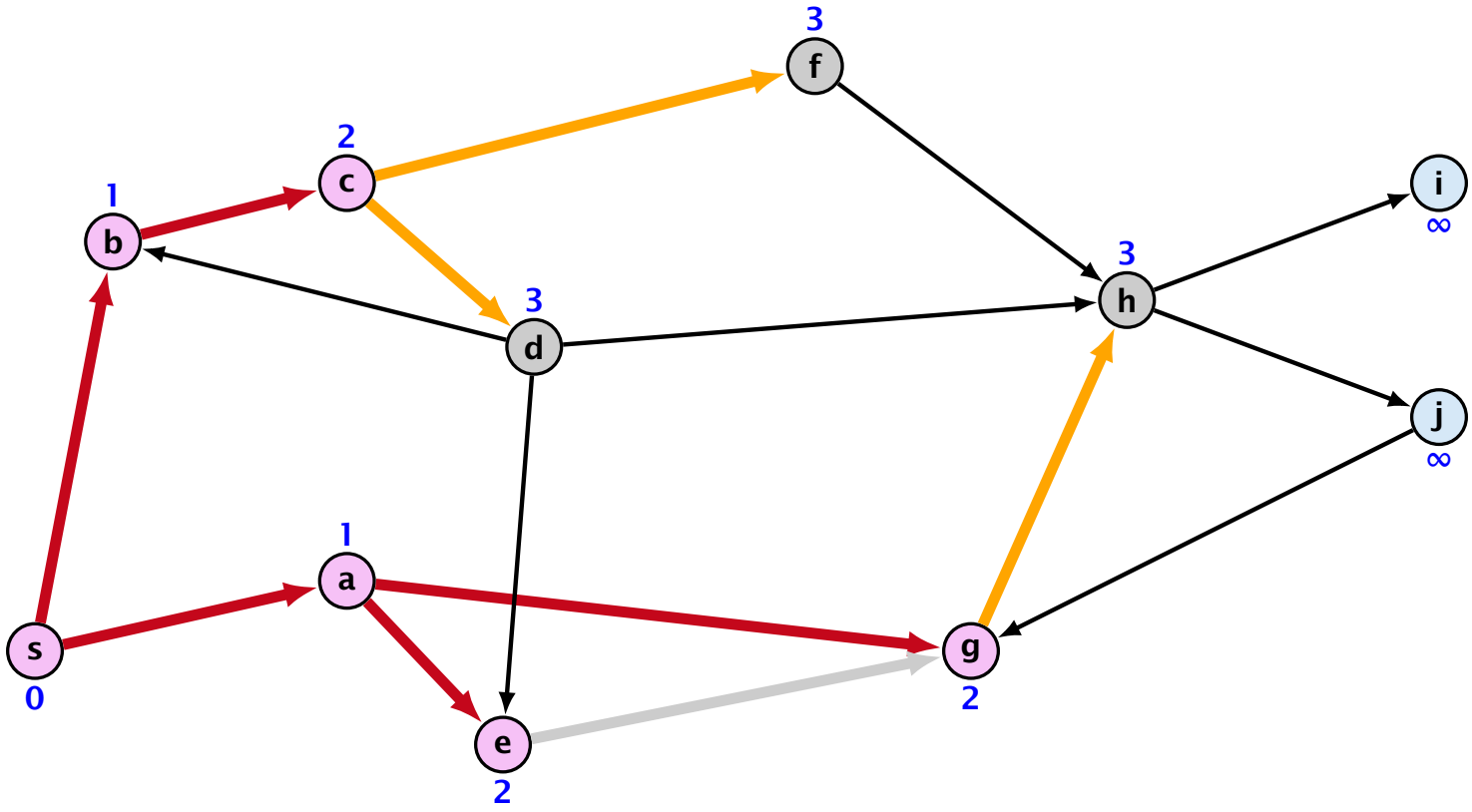
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

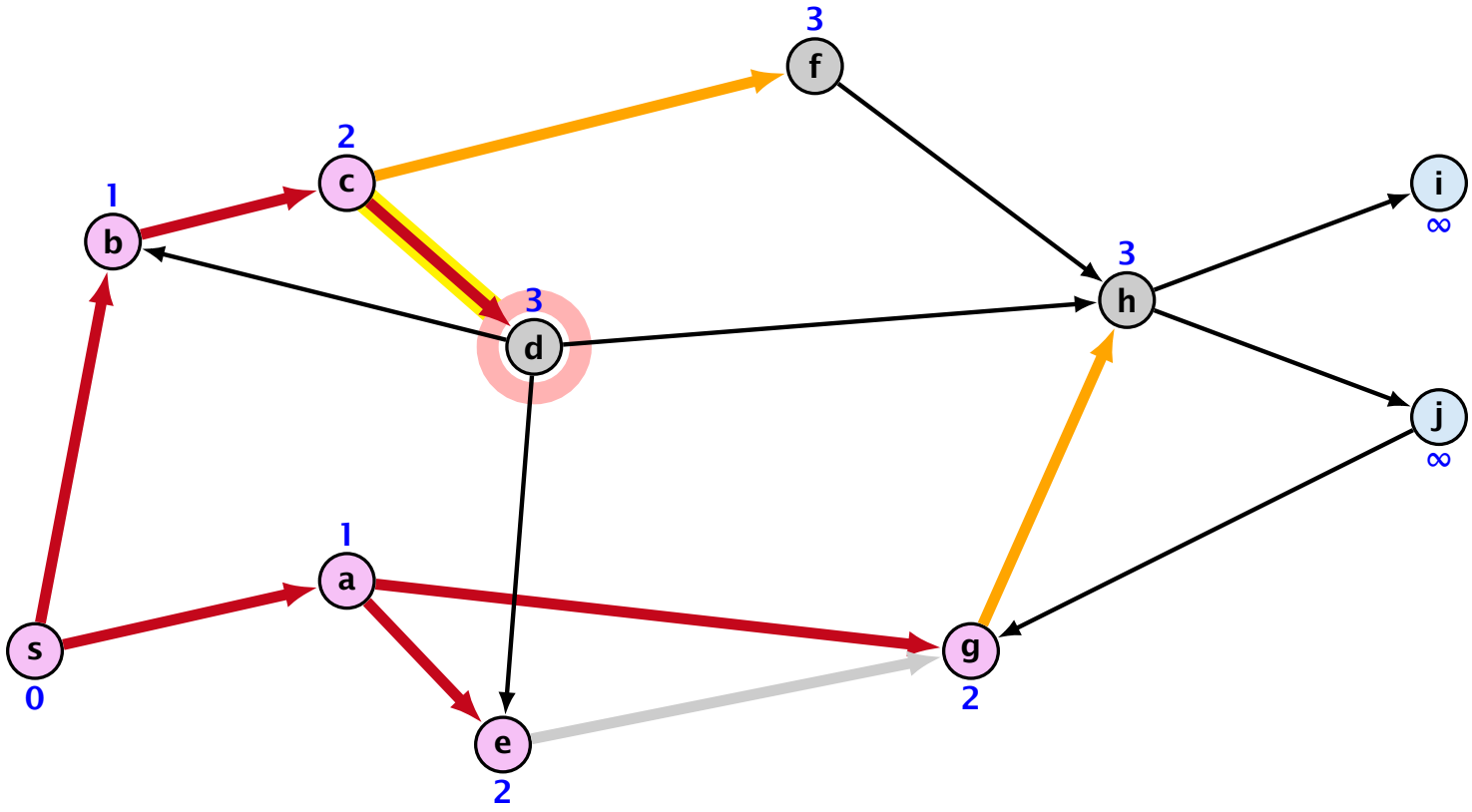
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

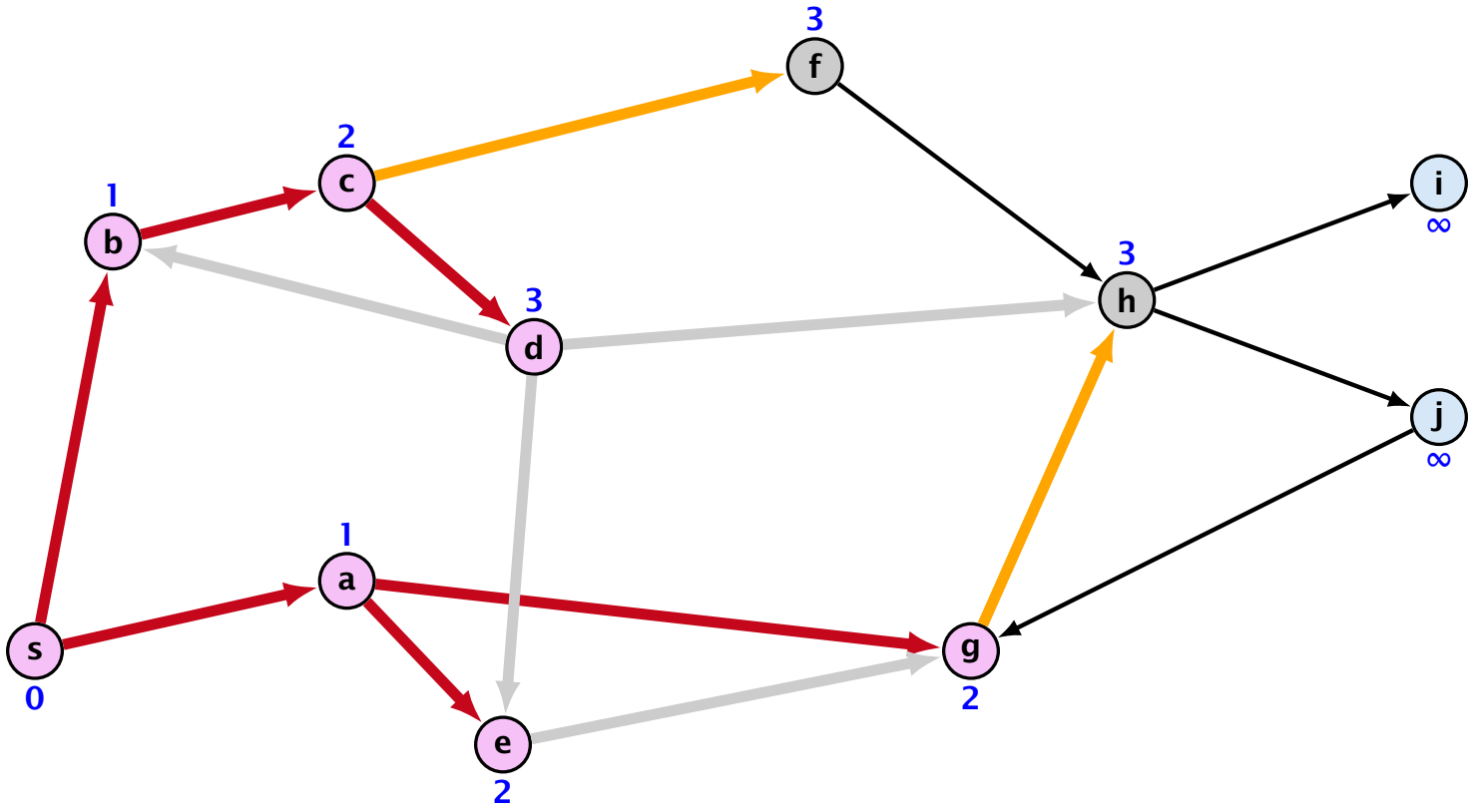
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

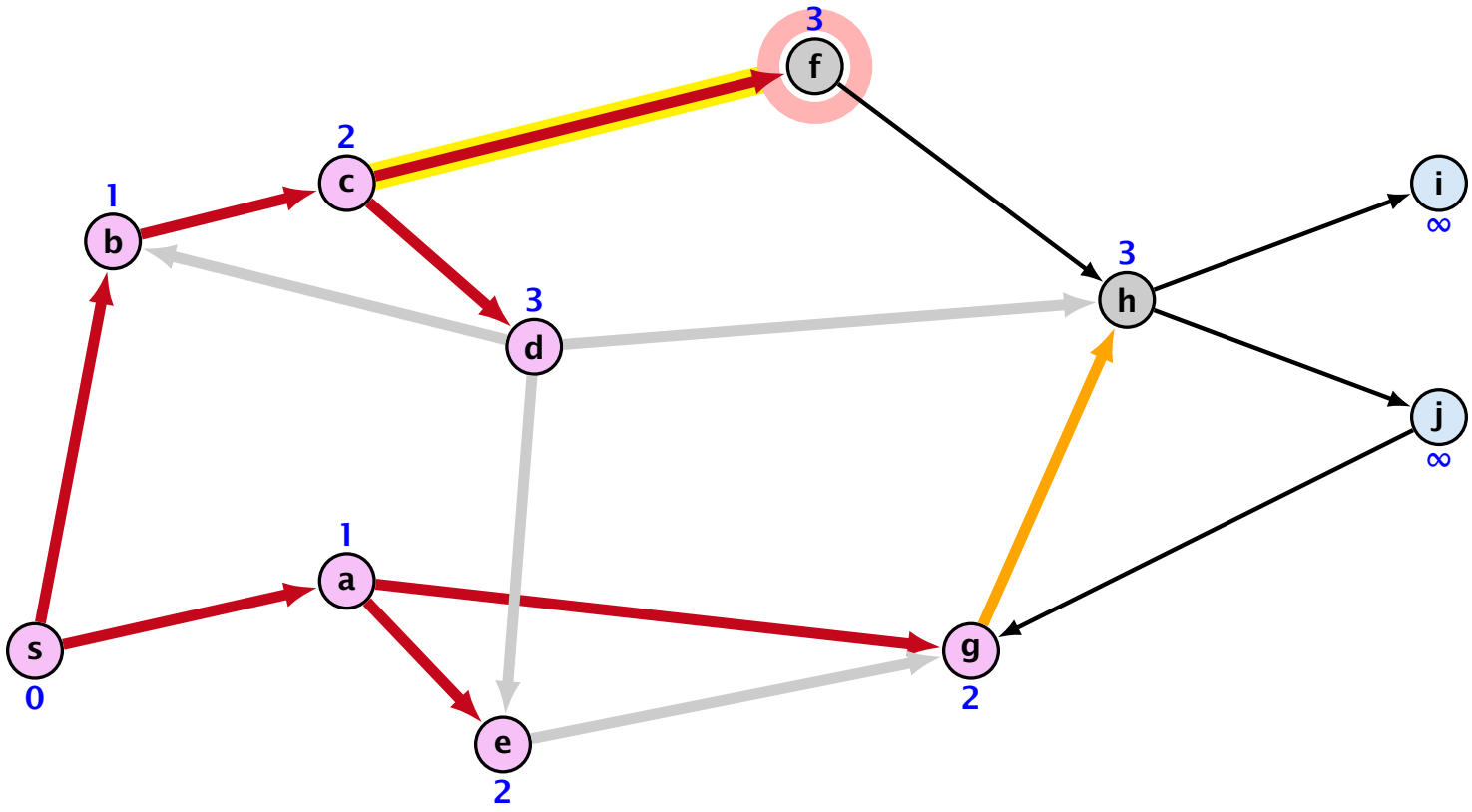
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

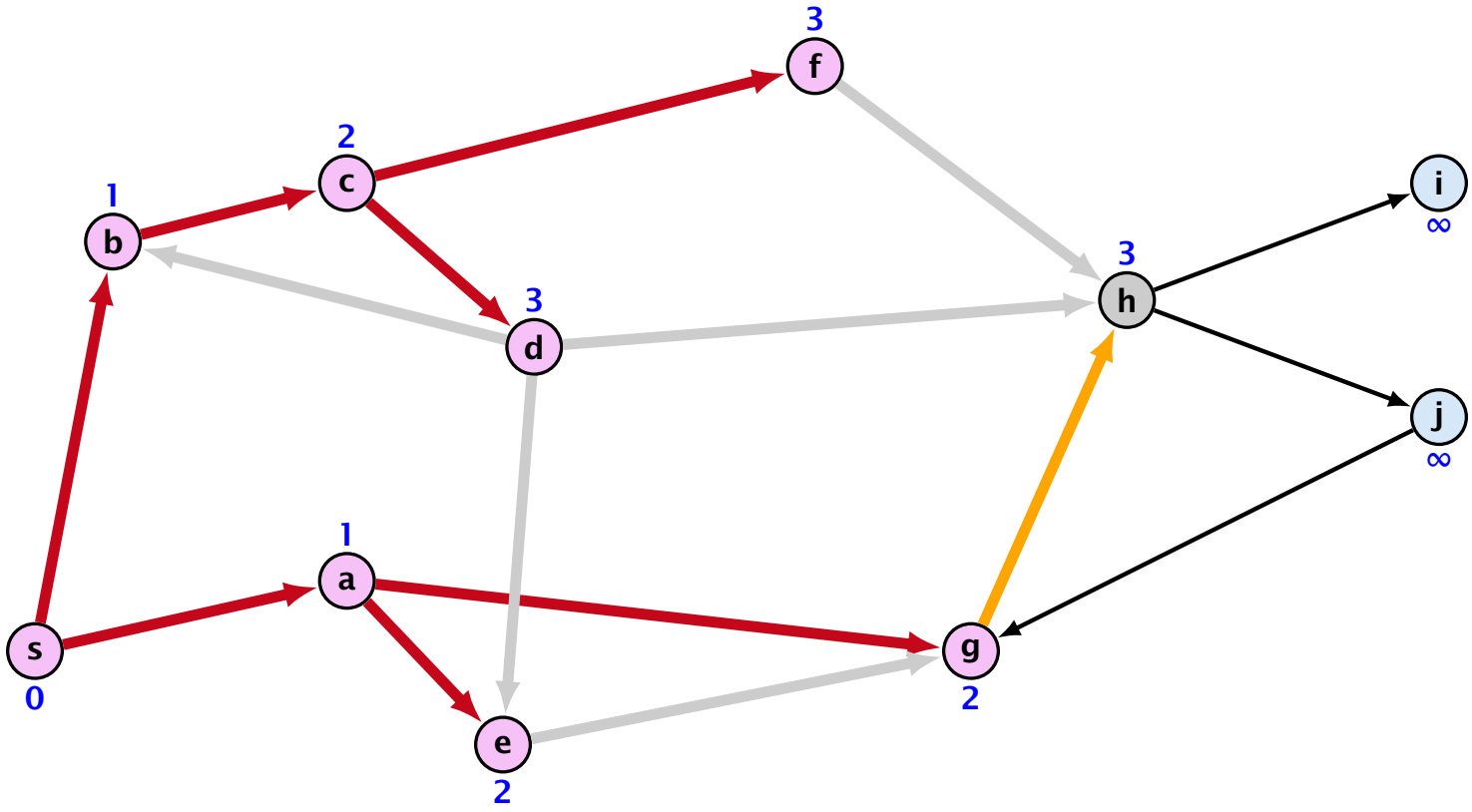
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

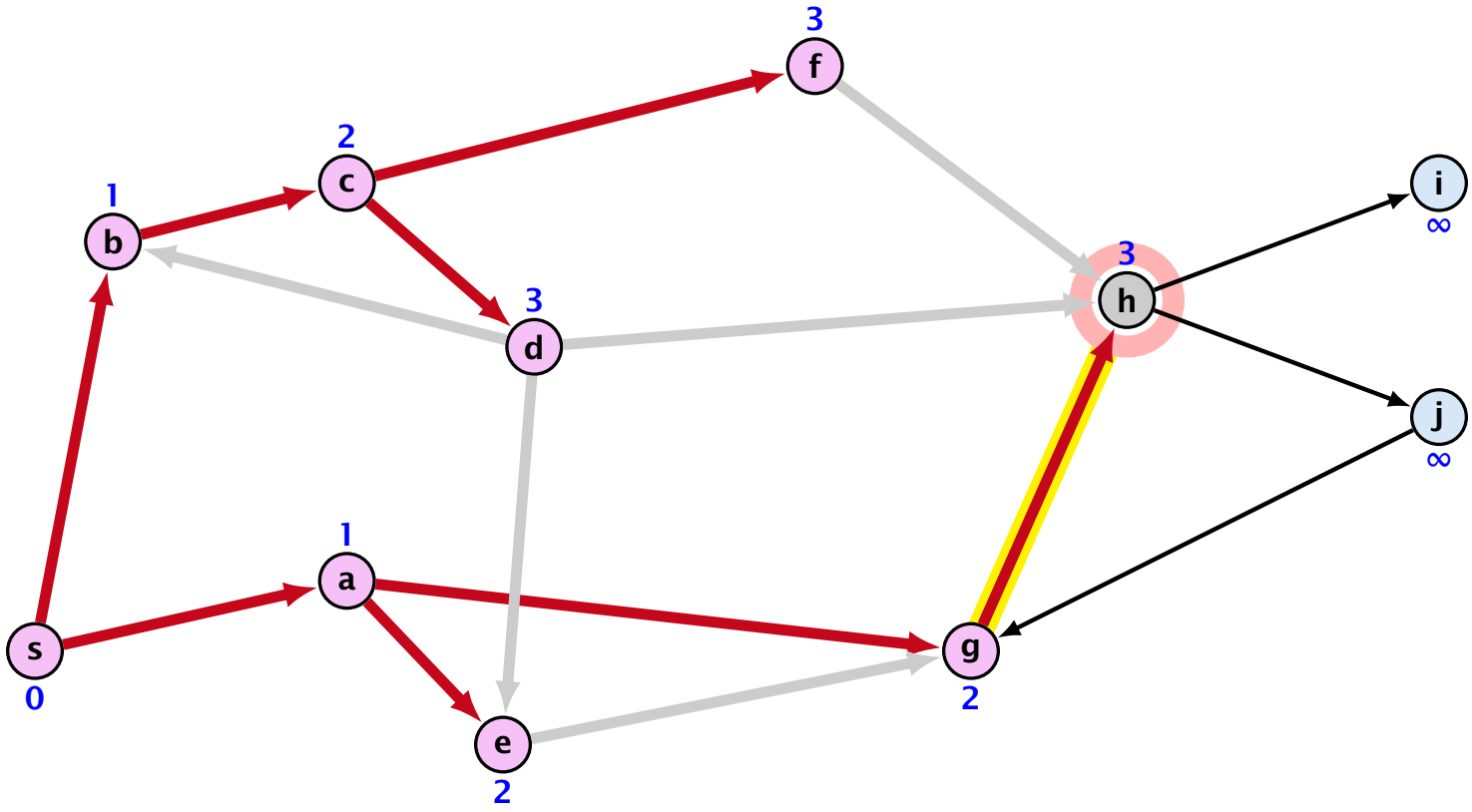
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

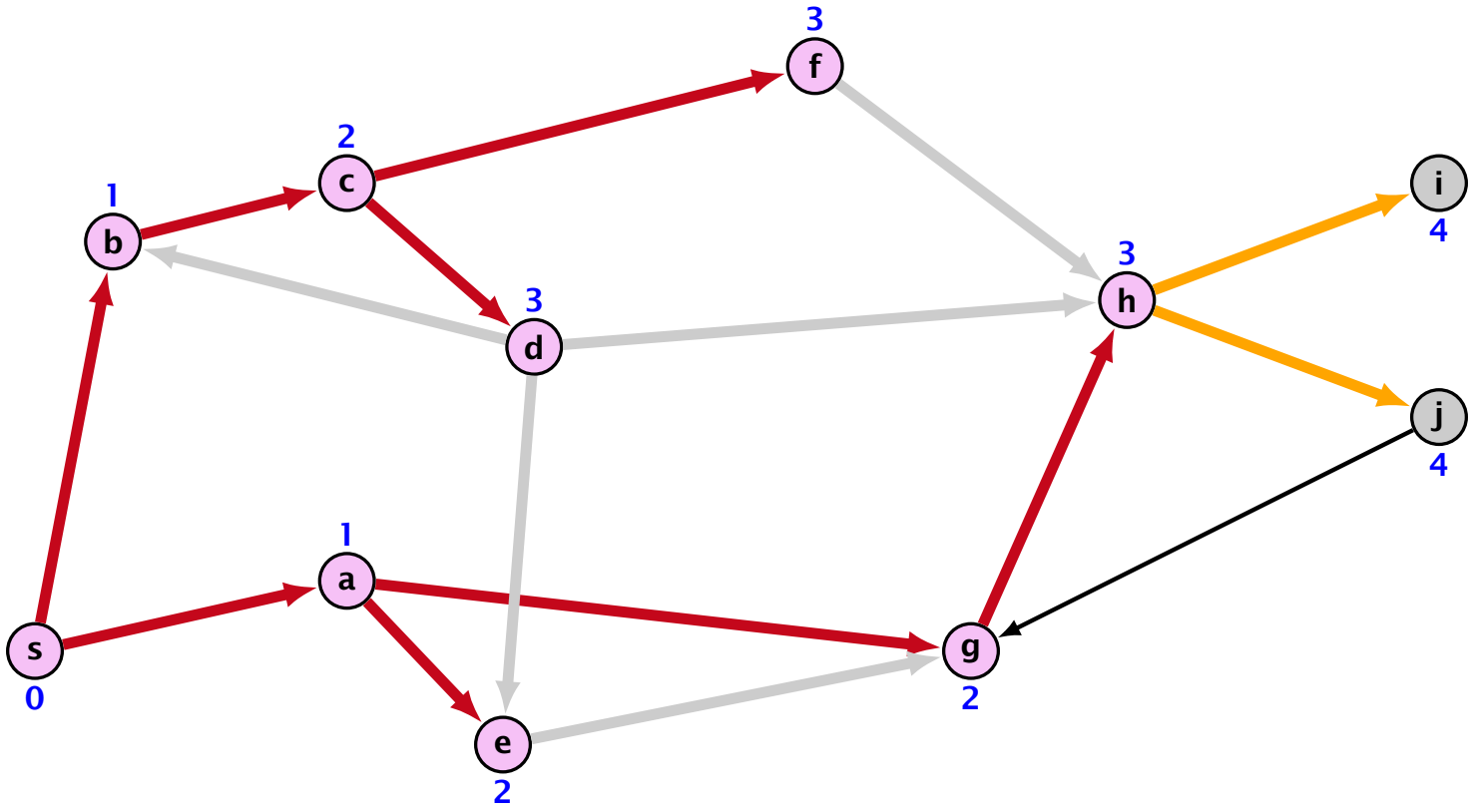
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

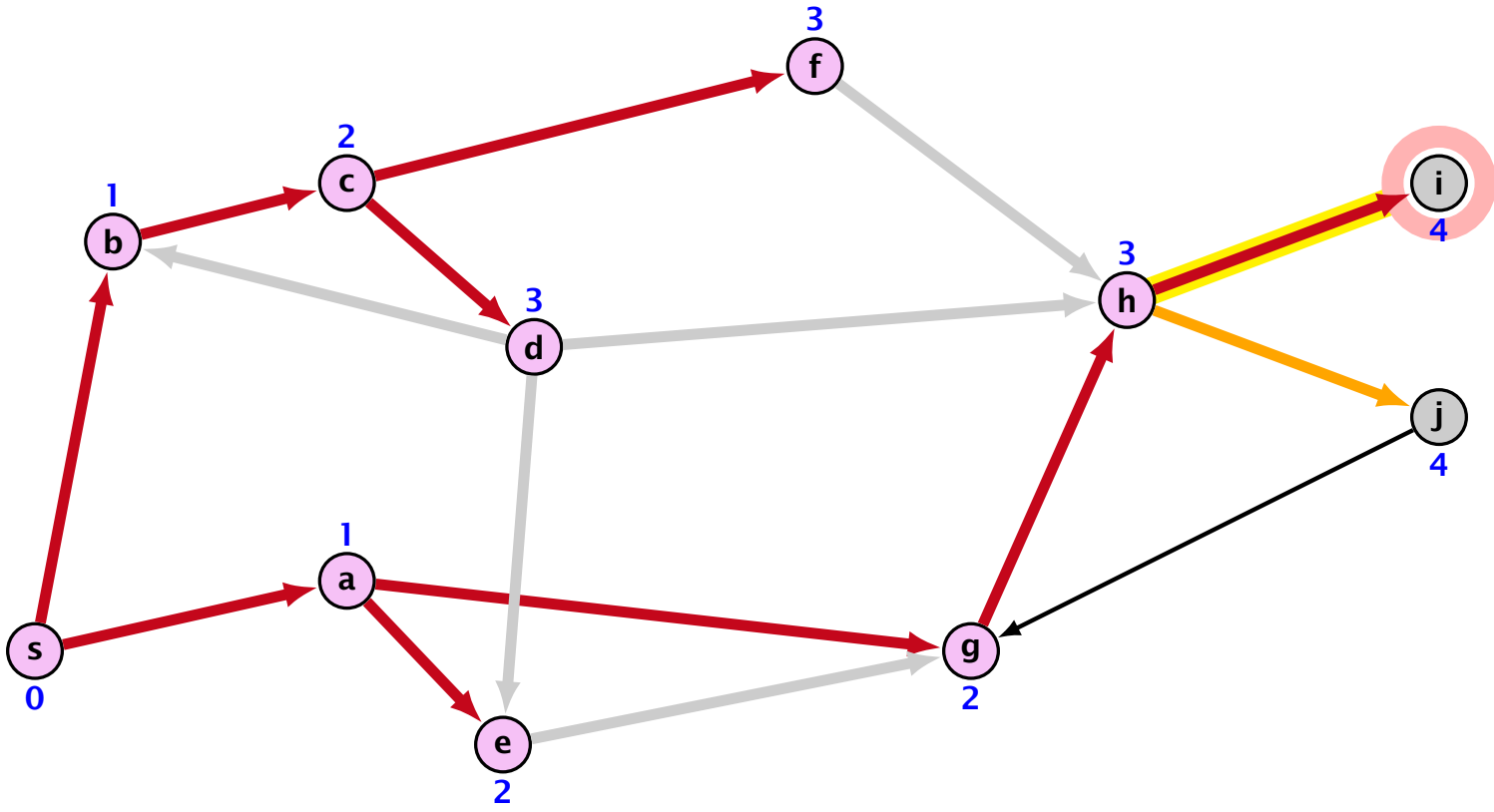
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

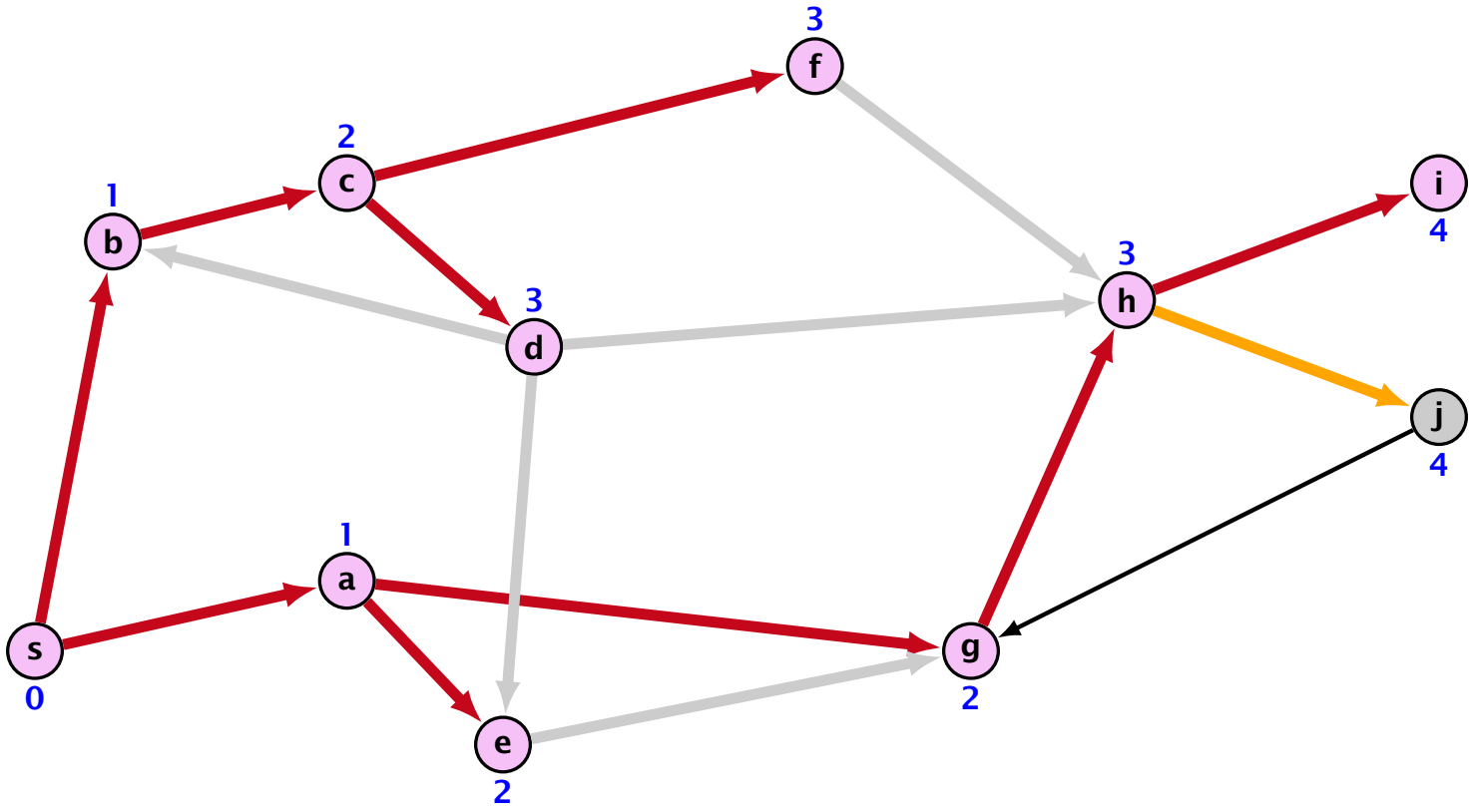
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

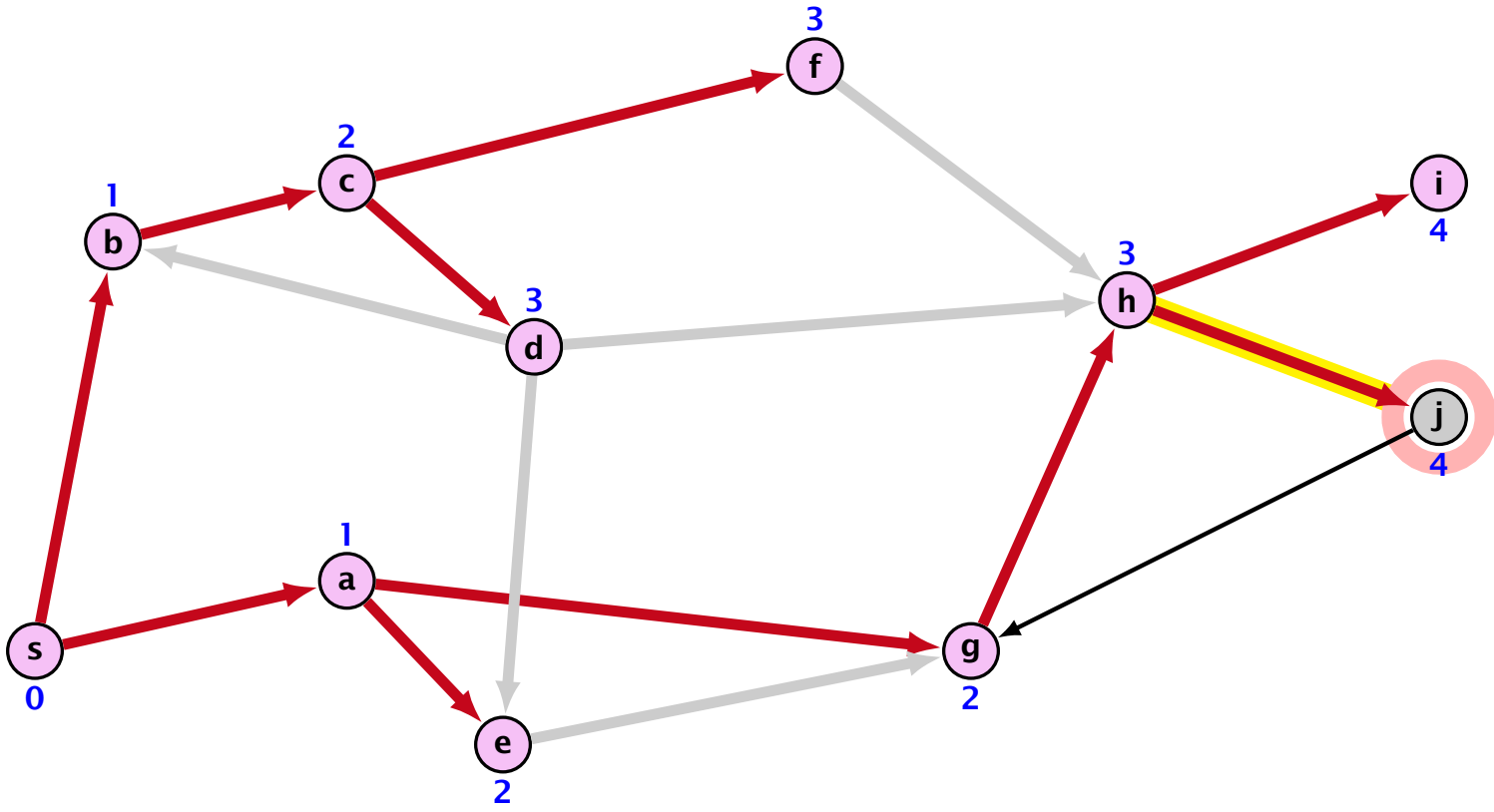
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

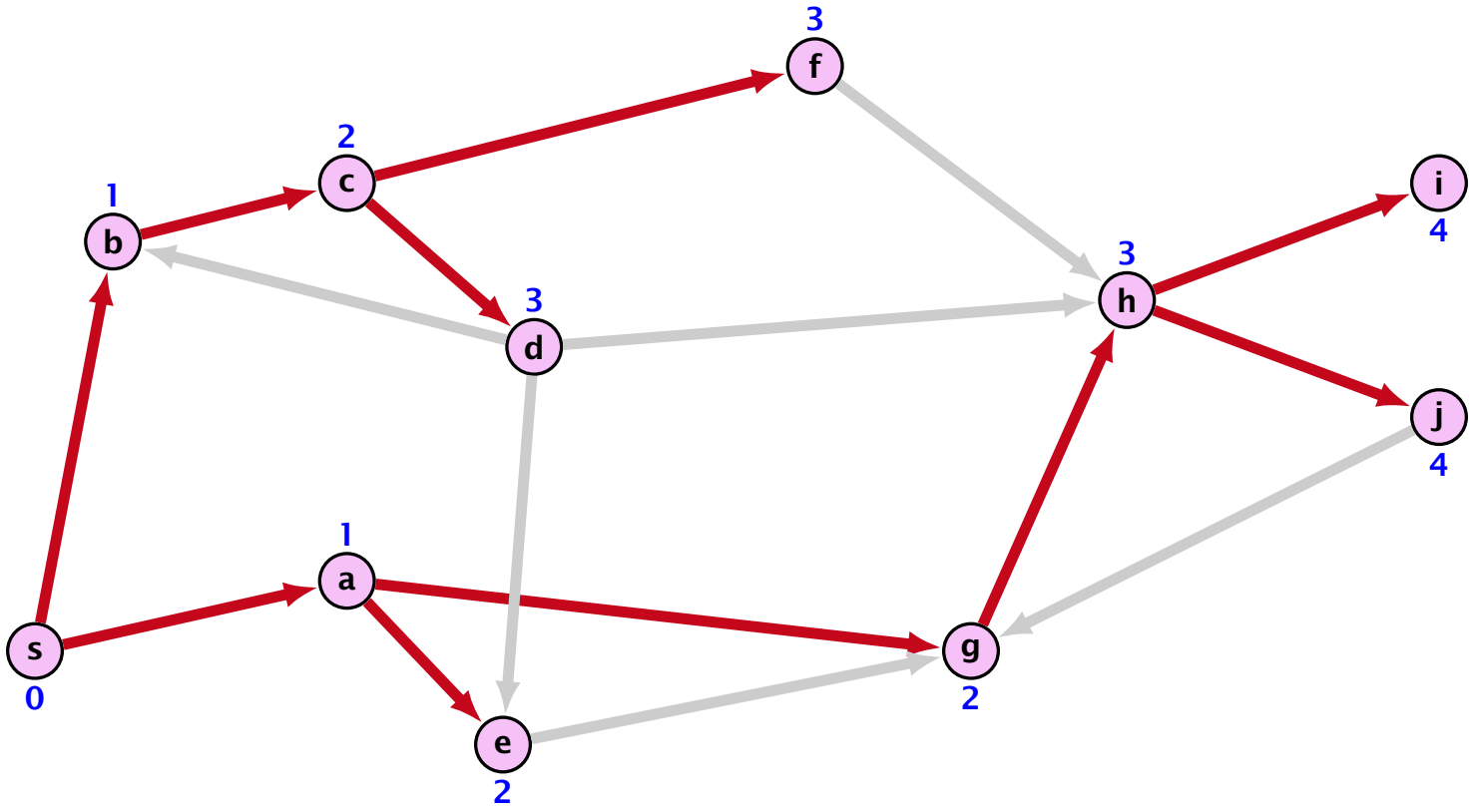
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

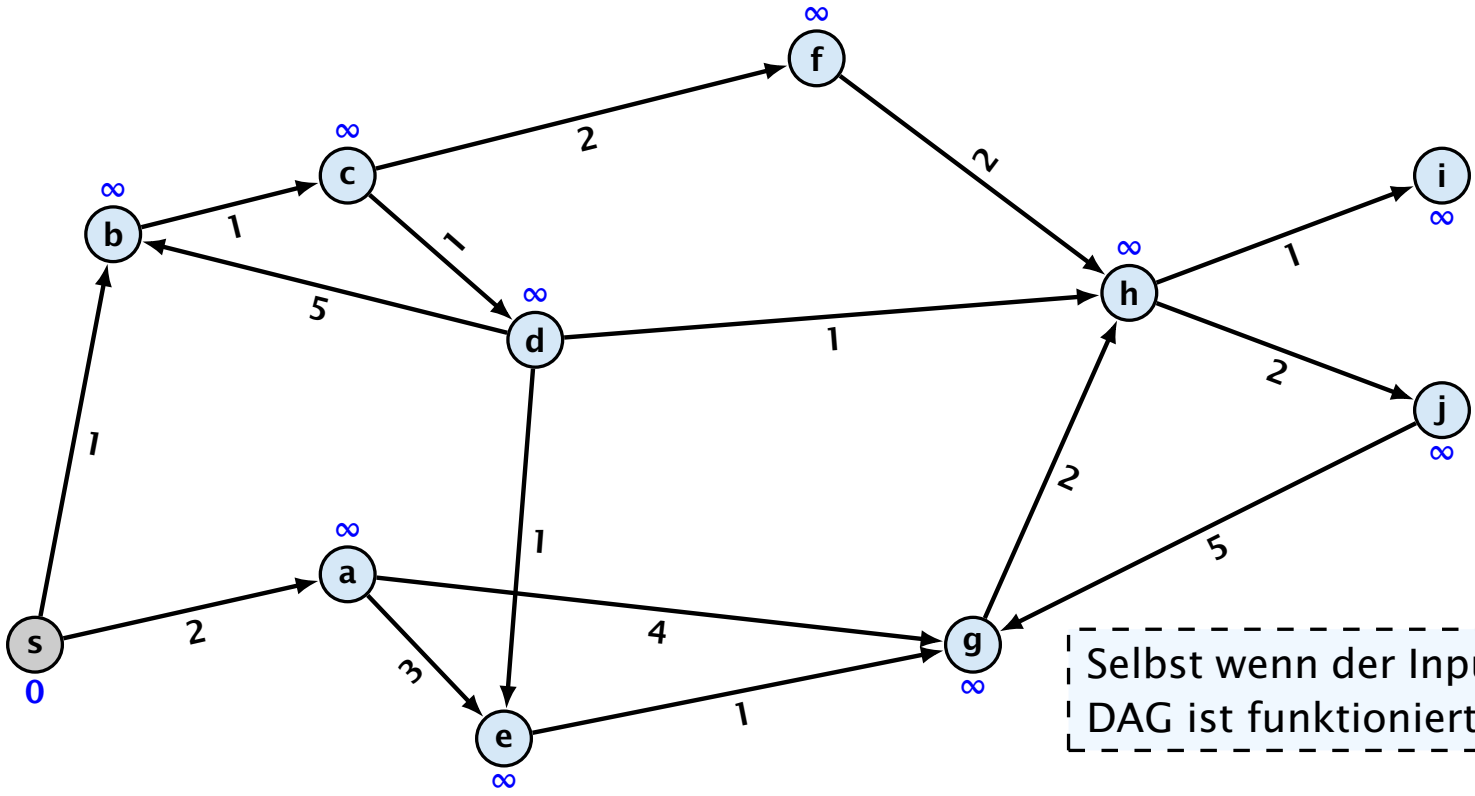
# Kürzeste Wege

Der Level ist die hop-Distanz von $s$ zu $v$; bei uniformen Kantengewichten entspricht dies der tatsächlichen Distanz.

- in der ersten Runde wird für jeden Level 1 Knoten eine eingehende Kante von Level 0 relaxiert
- in der zweiten Runde wird für jeden Level 2 Knoten eine eingehende Kante von Level 1 relaxiert
- ...

Bei einem kürzesten Pfad werden Relaxierungen in der Reihenfolge des Pfades durchgeführt.

# Beliebige Gewichte – BFS funktioniert nicht



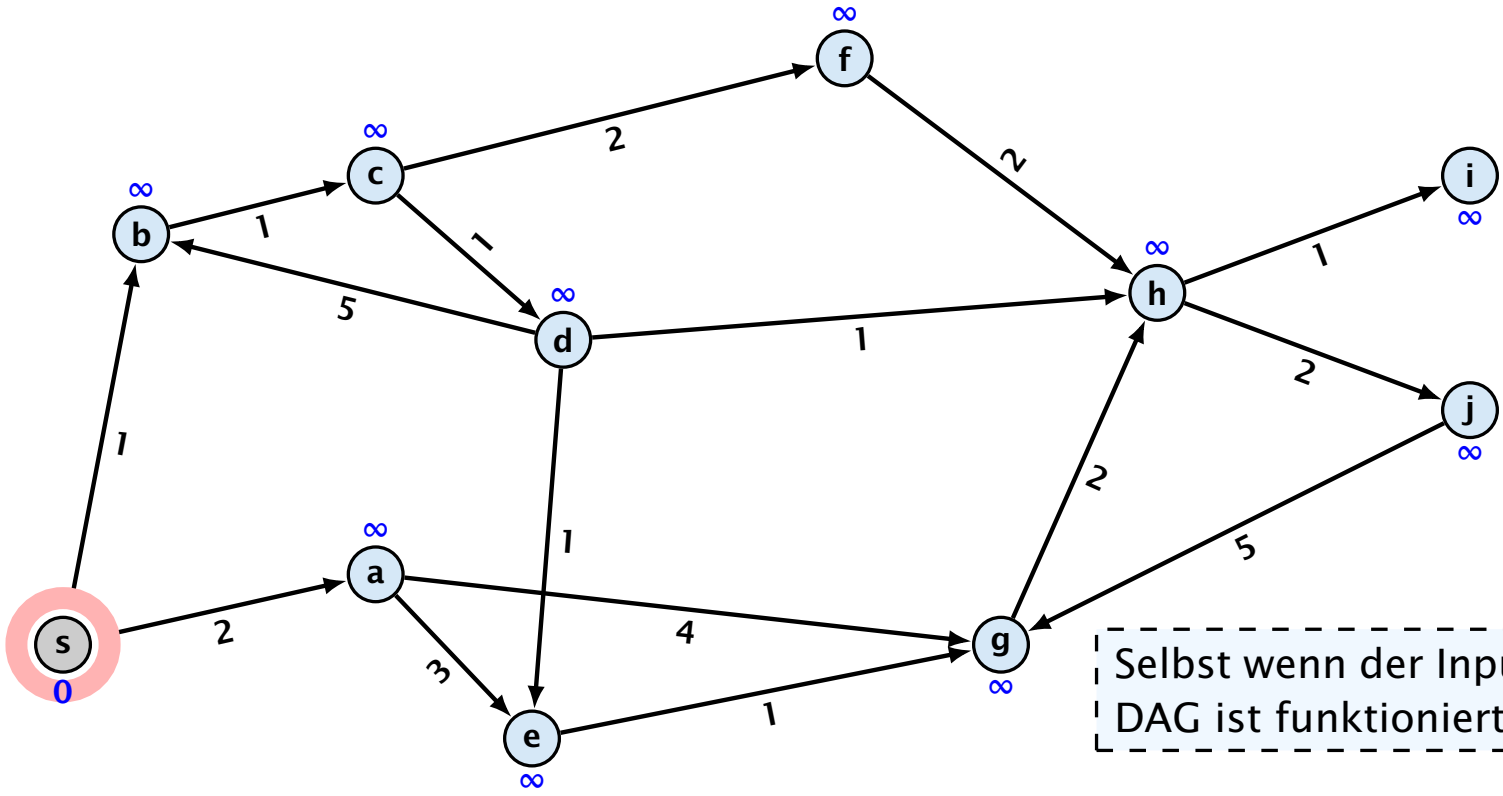

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



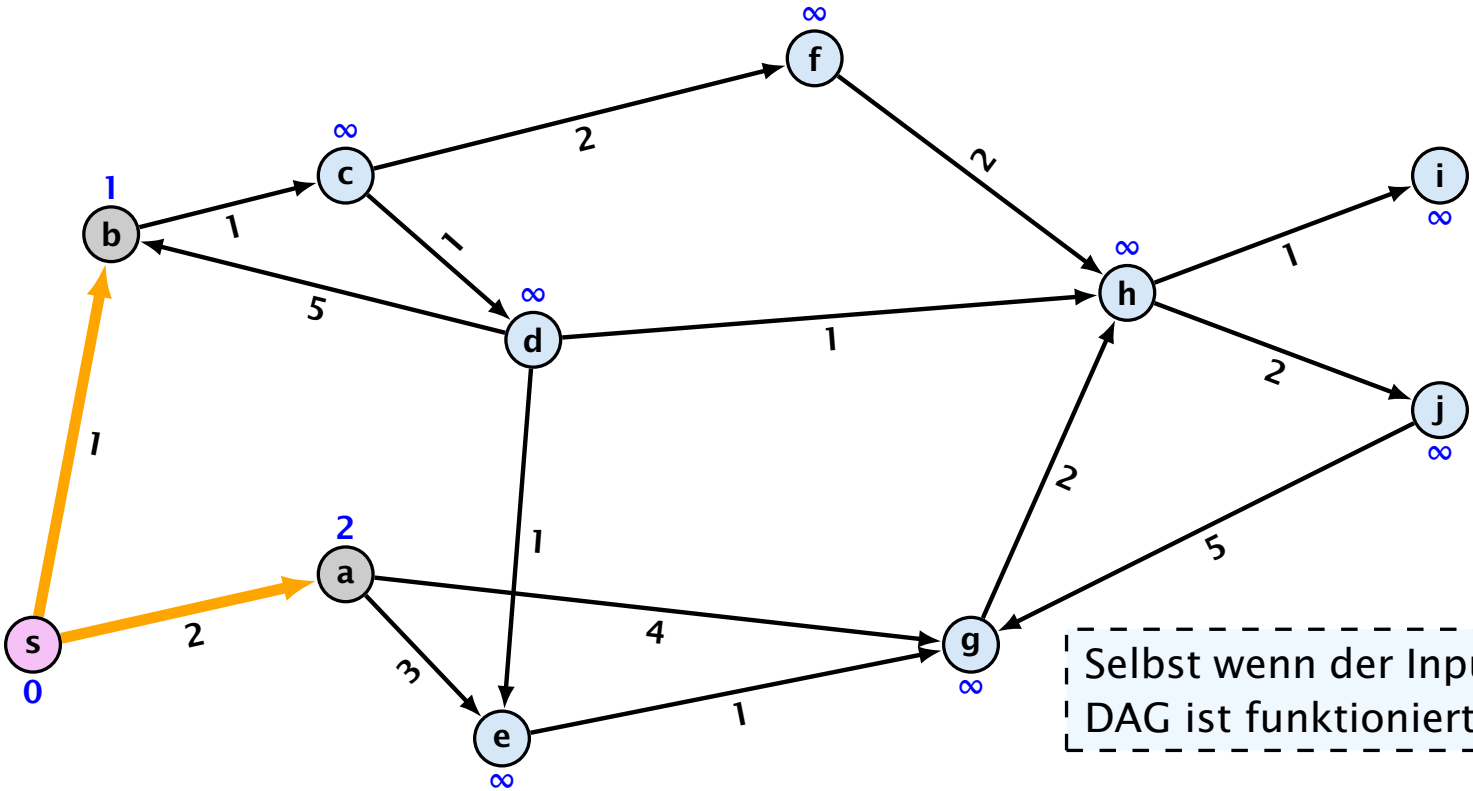

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



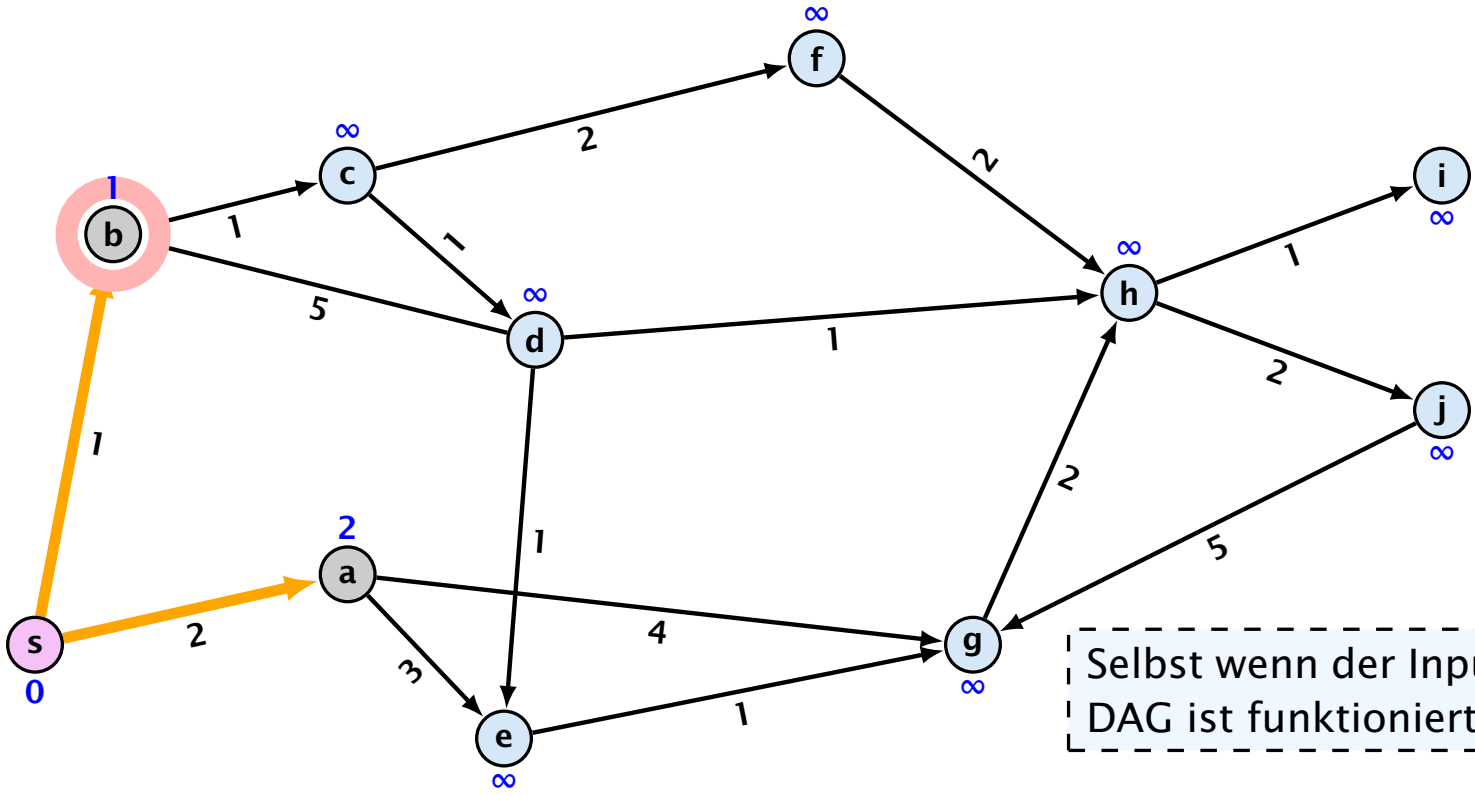

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



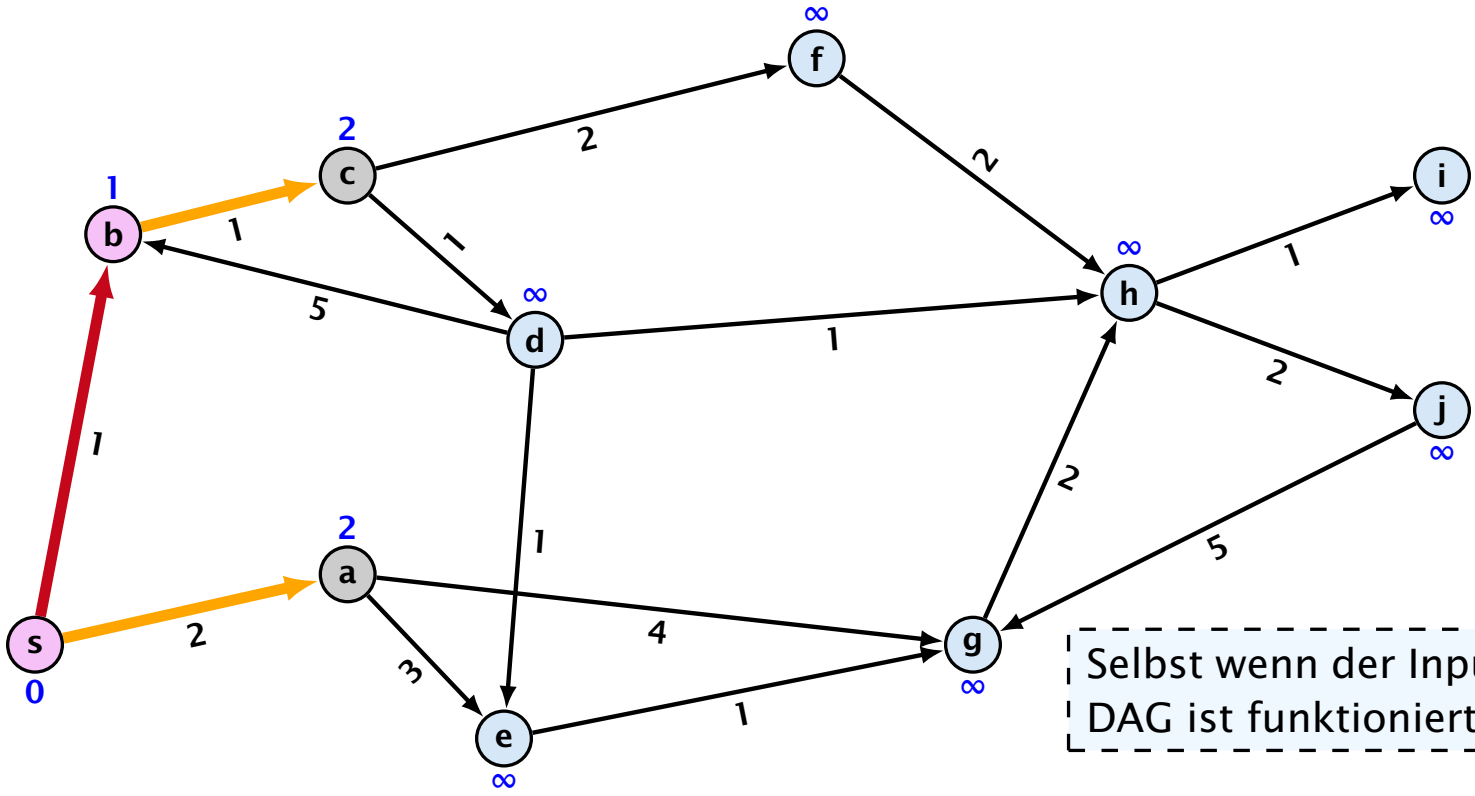

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



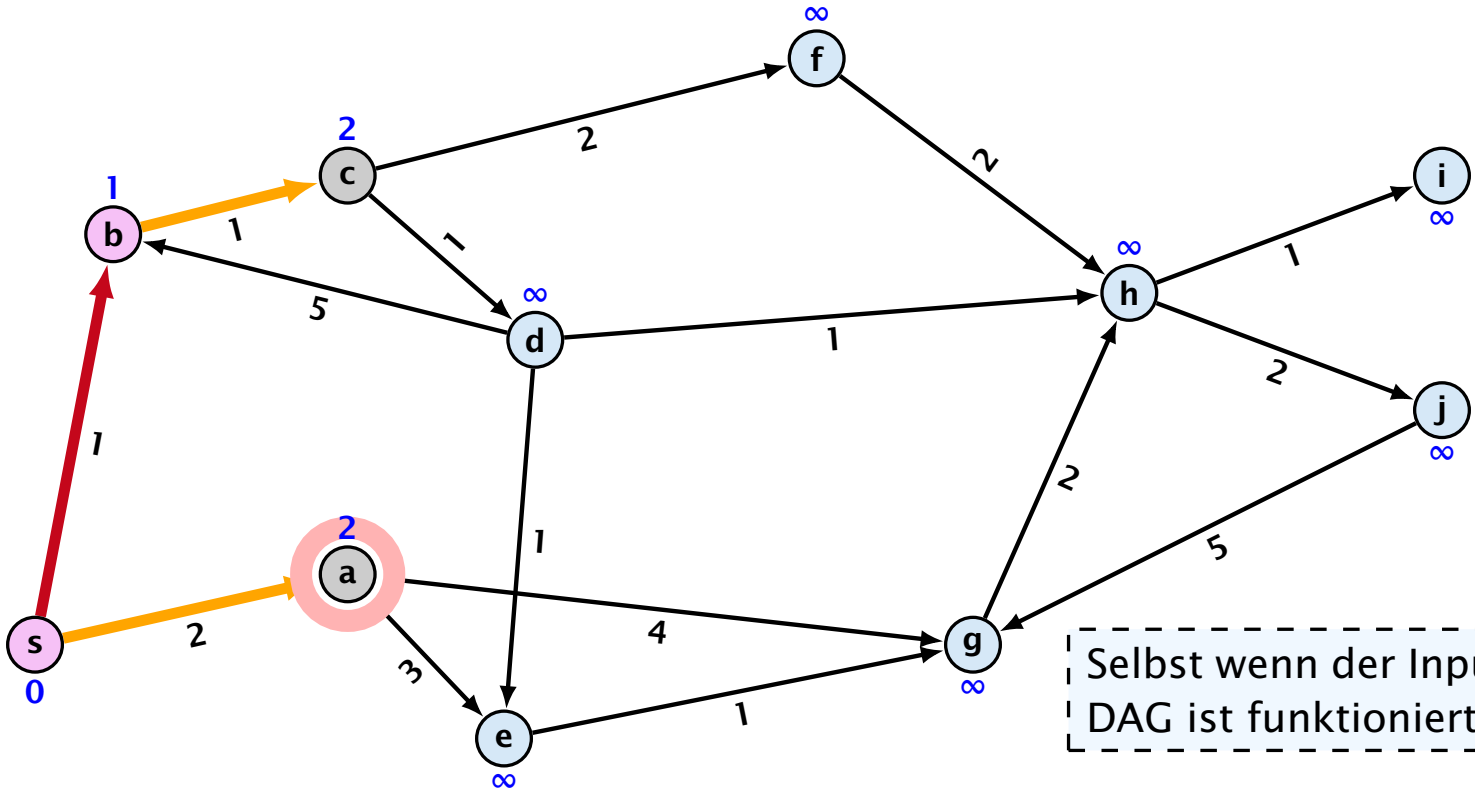

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



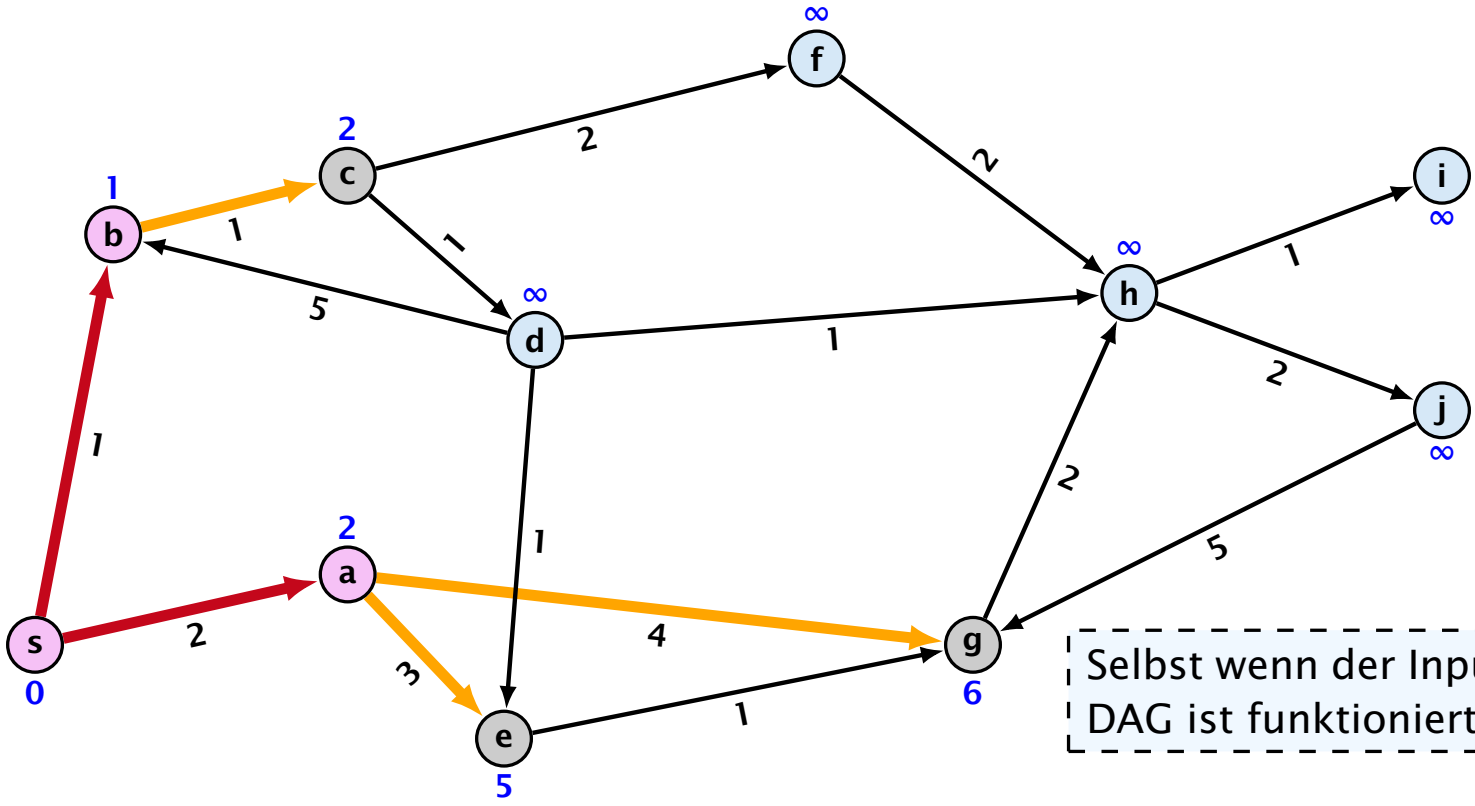

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



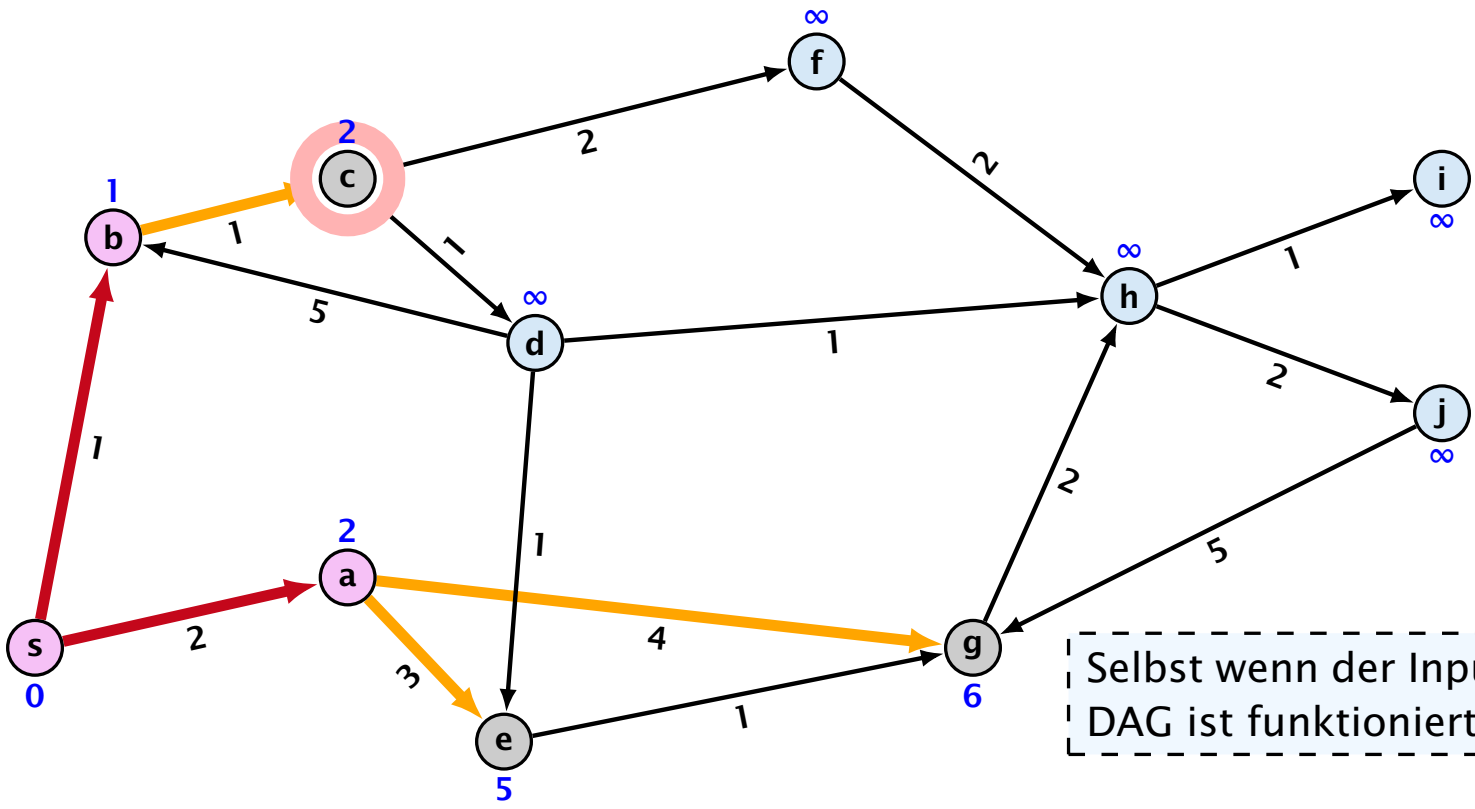

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



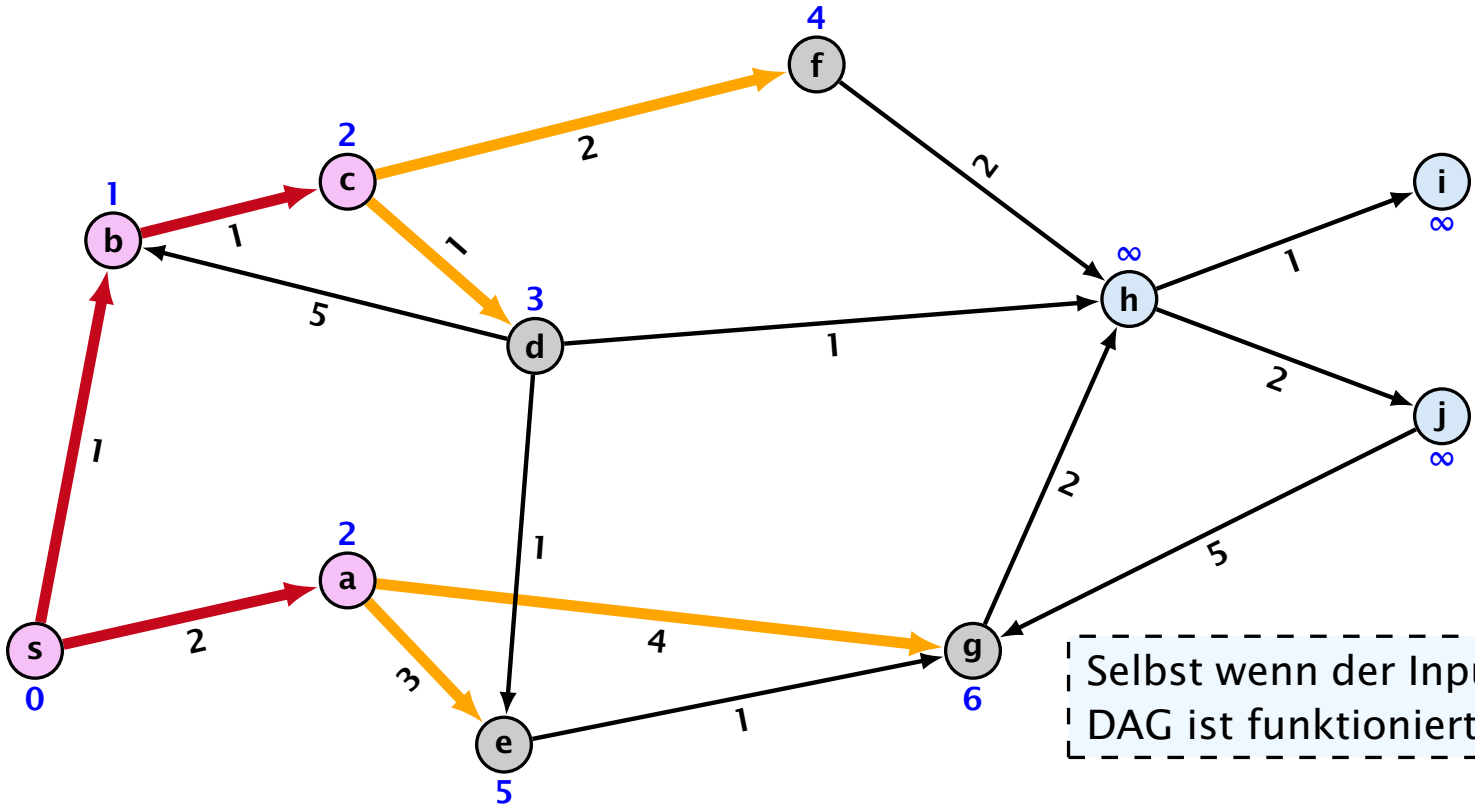

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



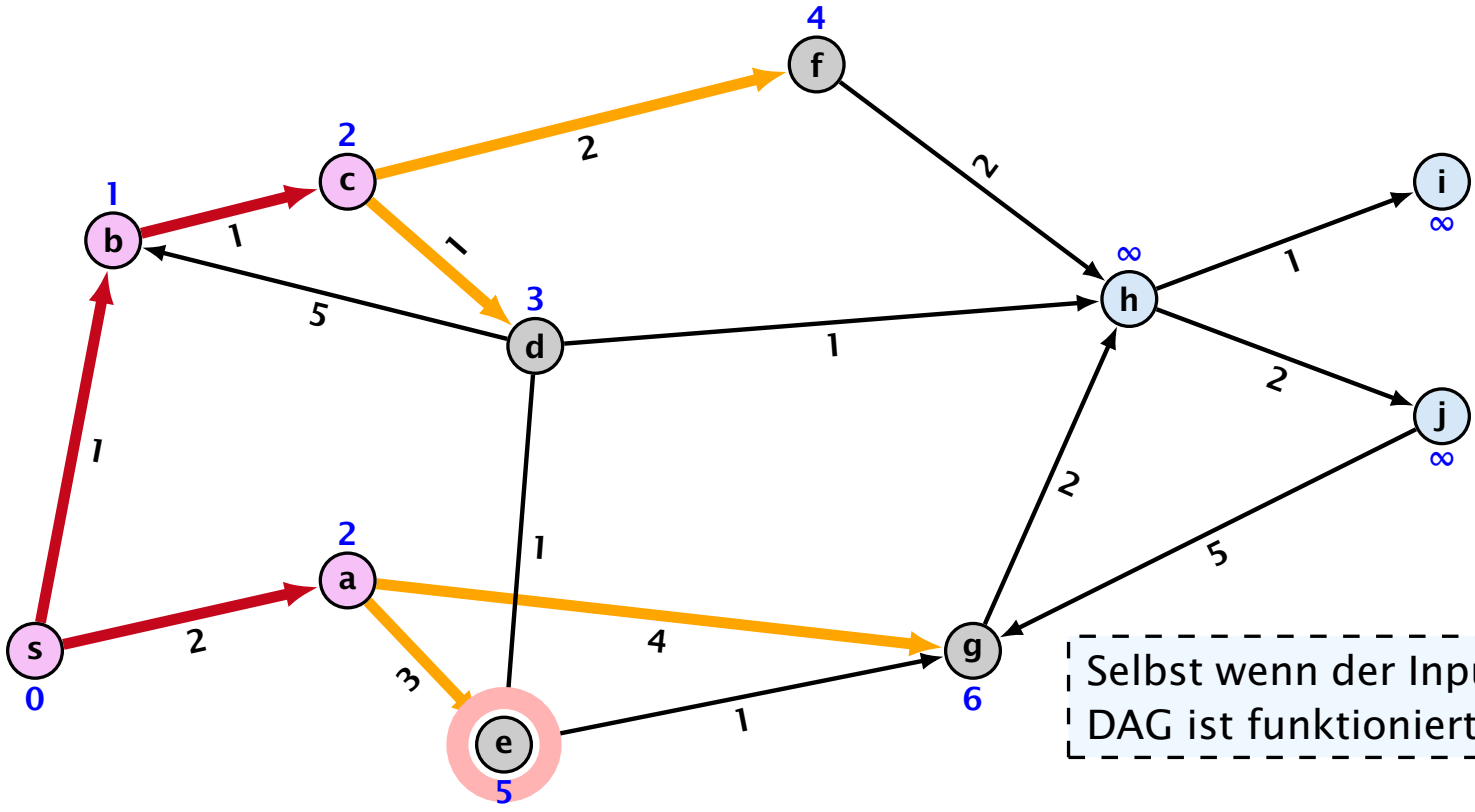

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



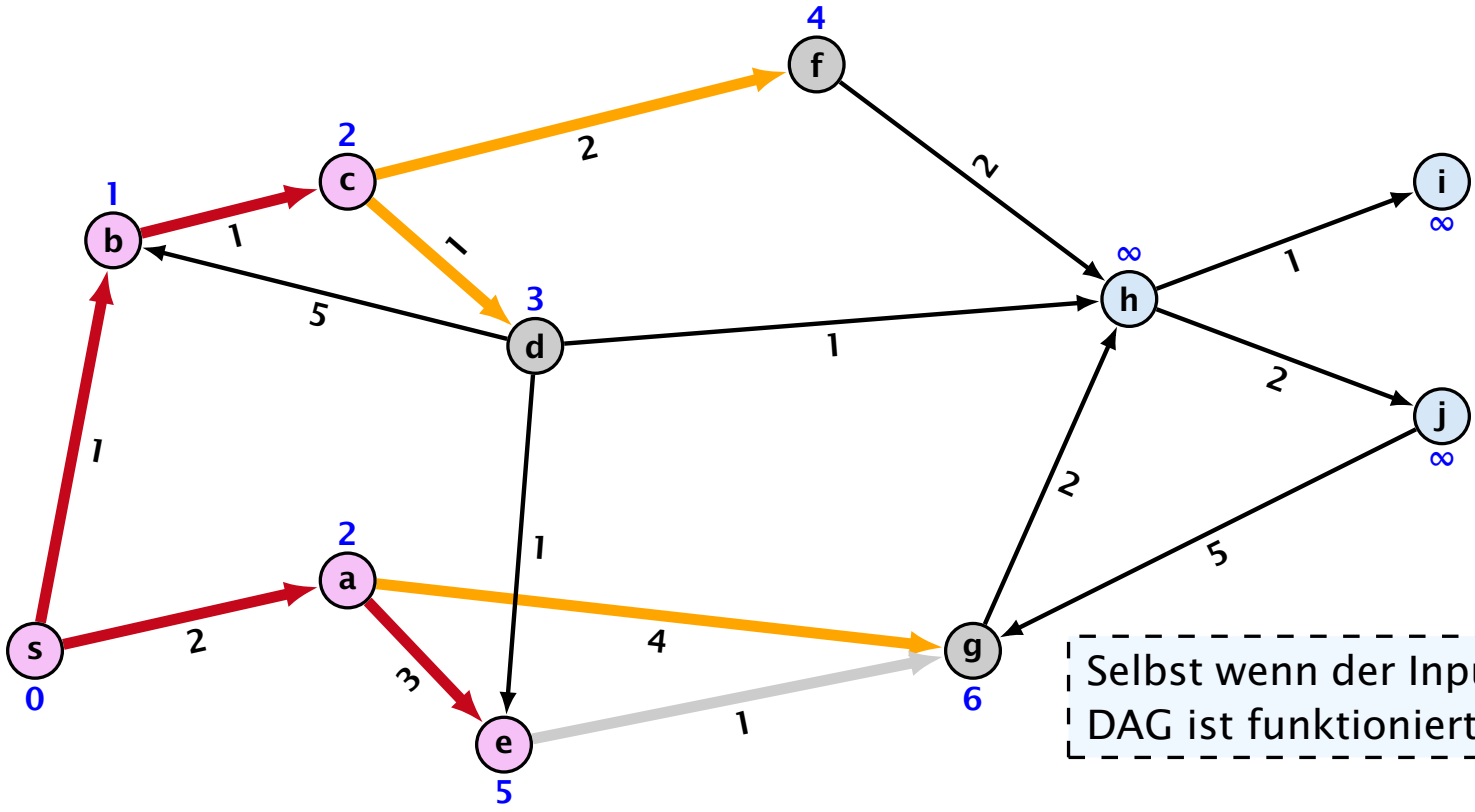

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



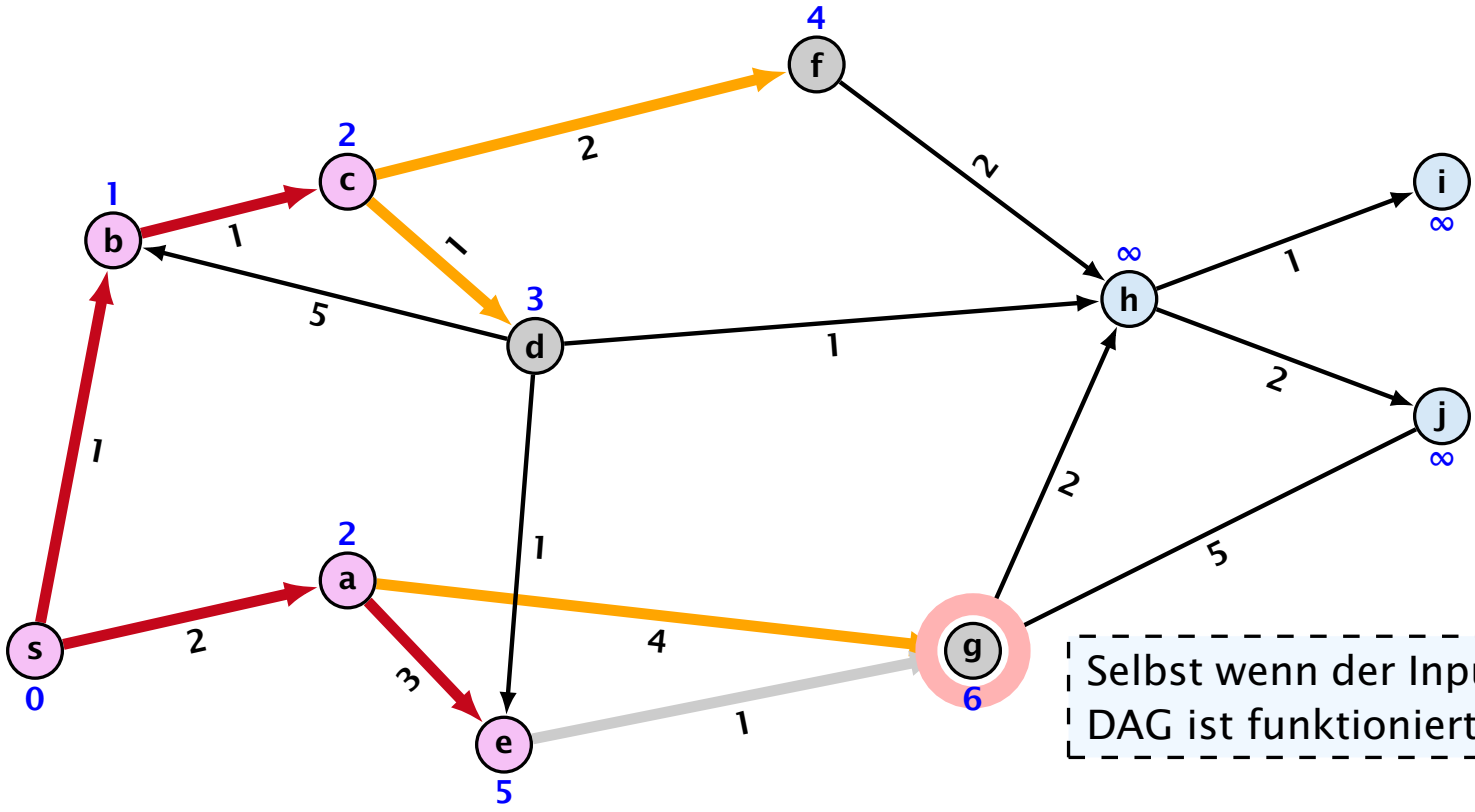

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



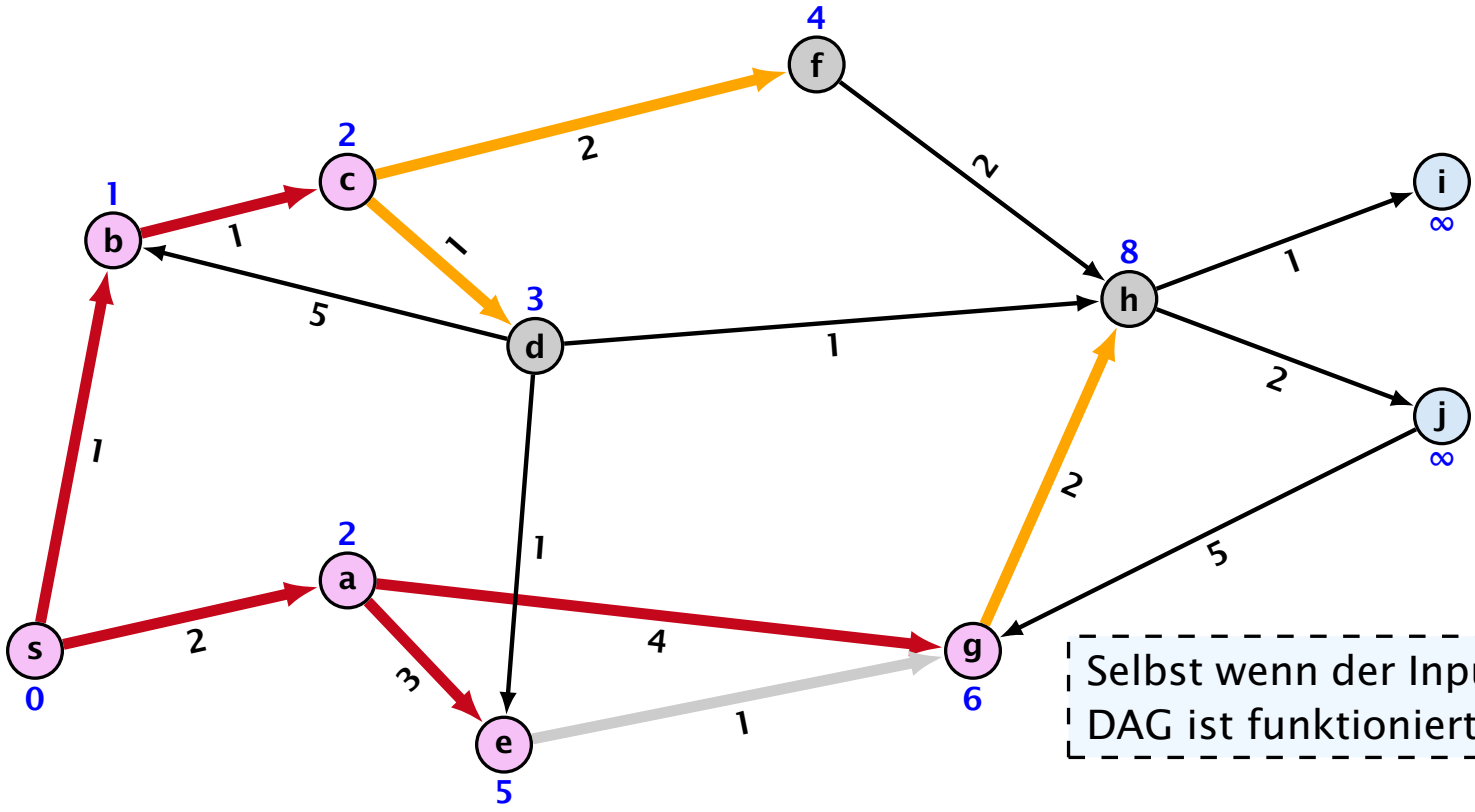

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



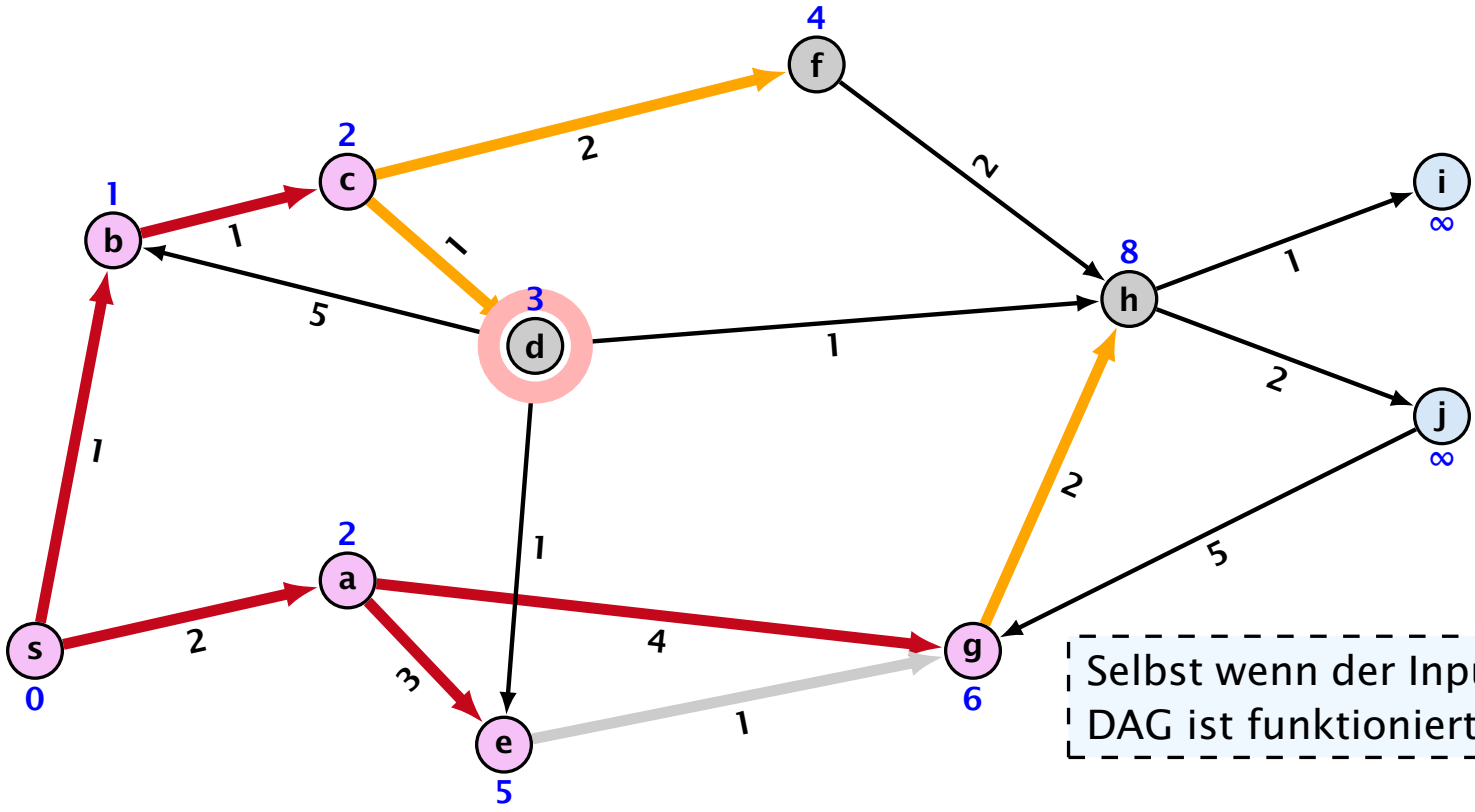

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



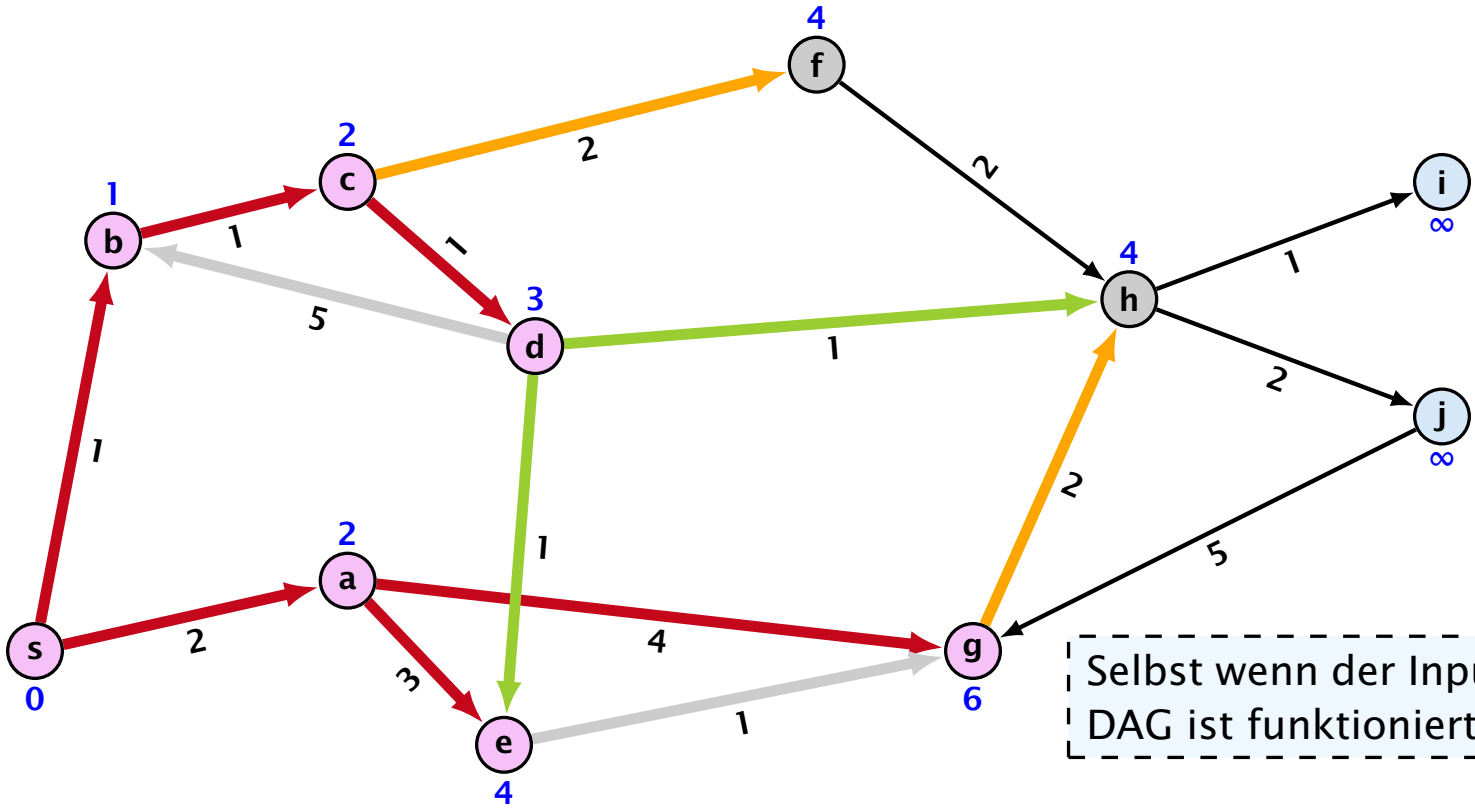

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



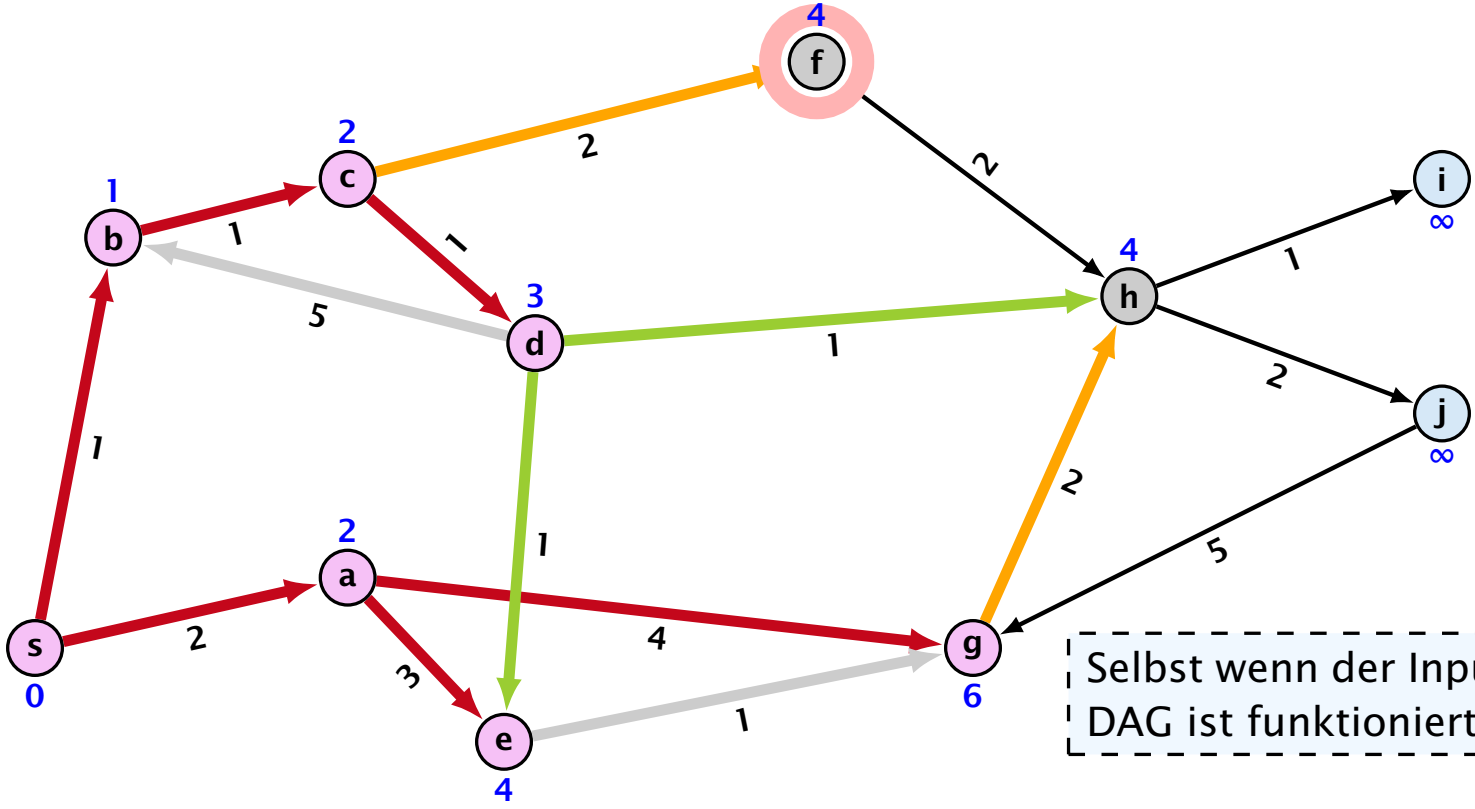

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



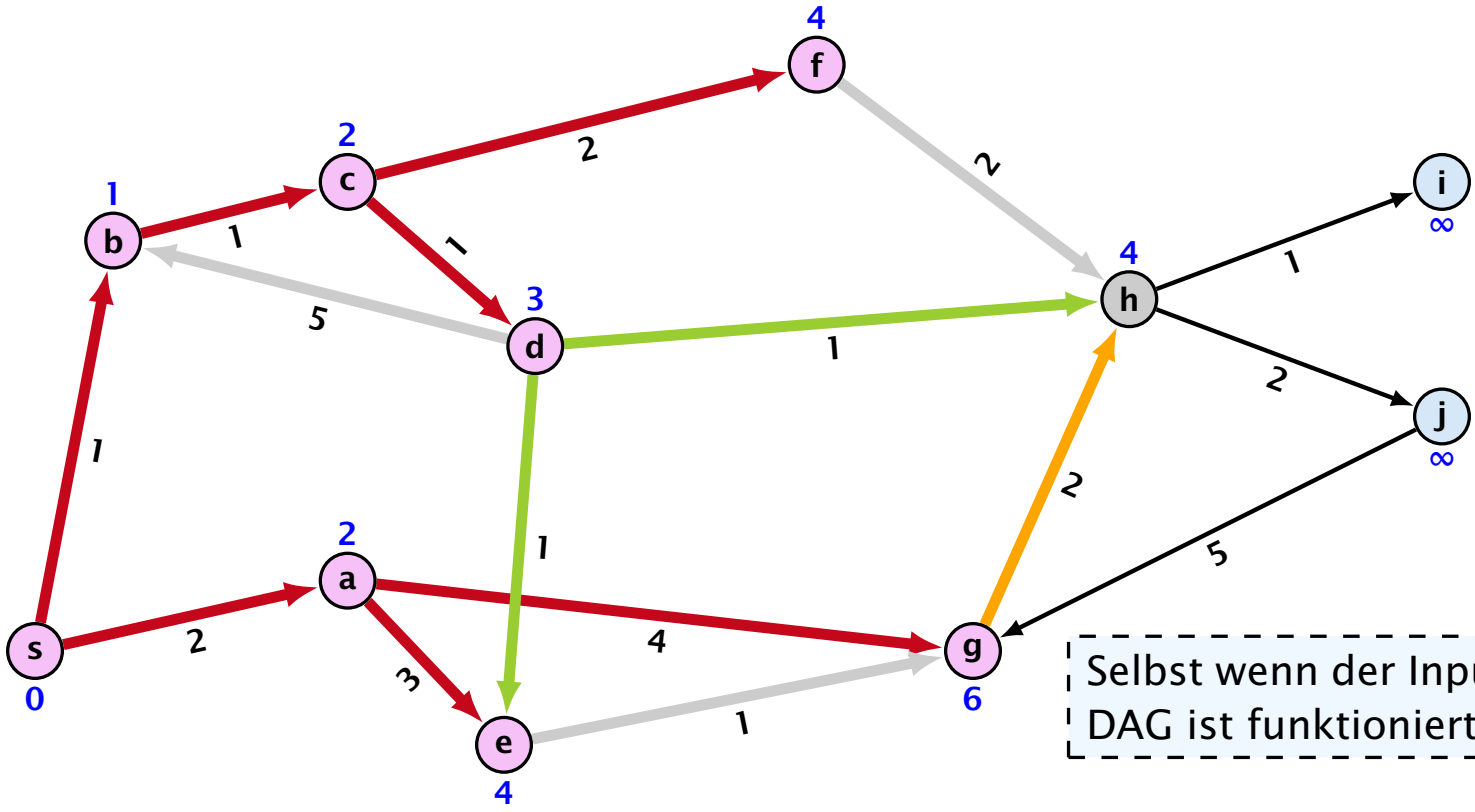

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



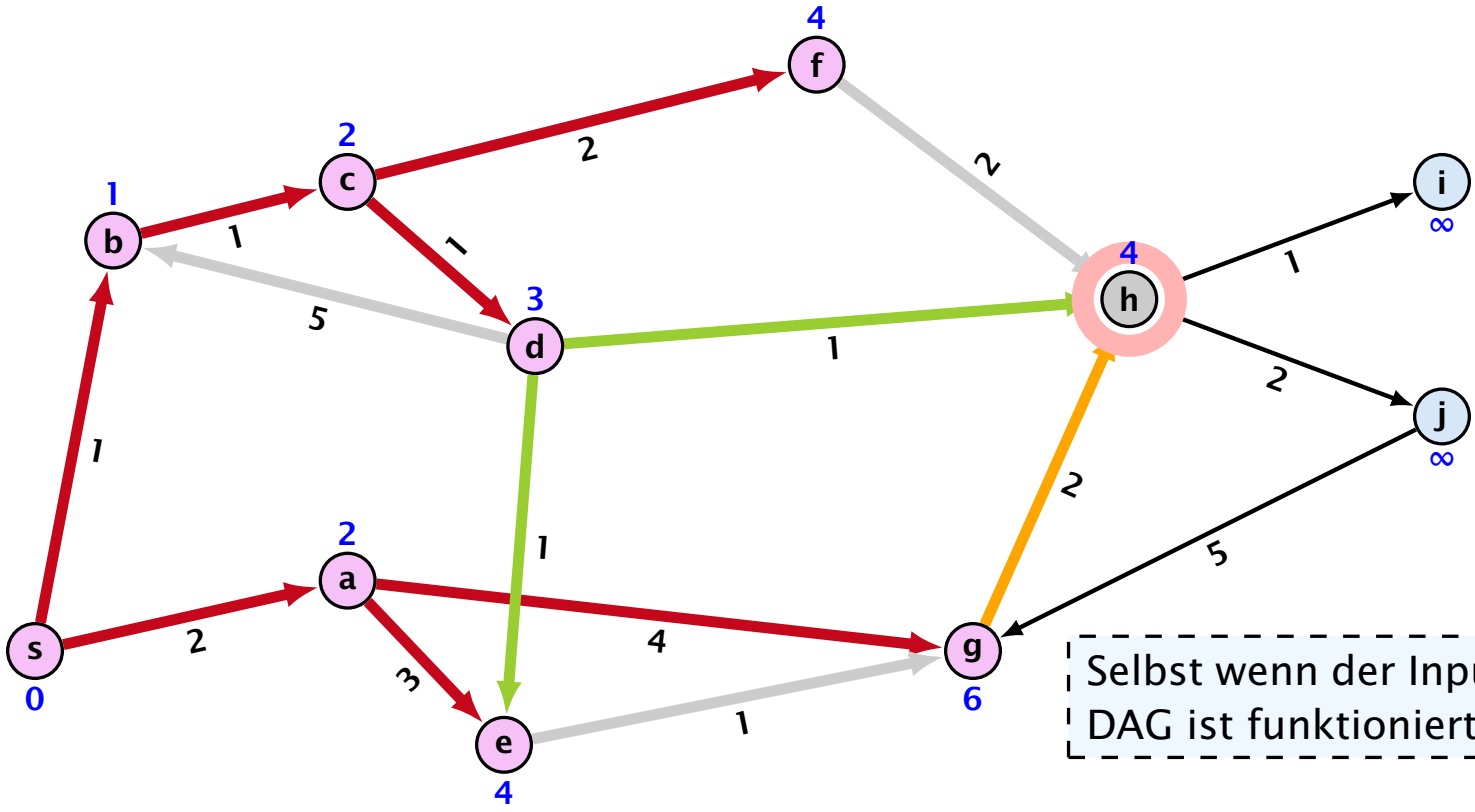

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



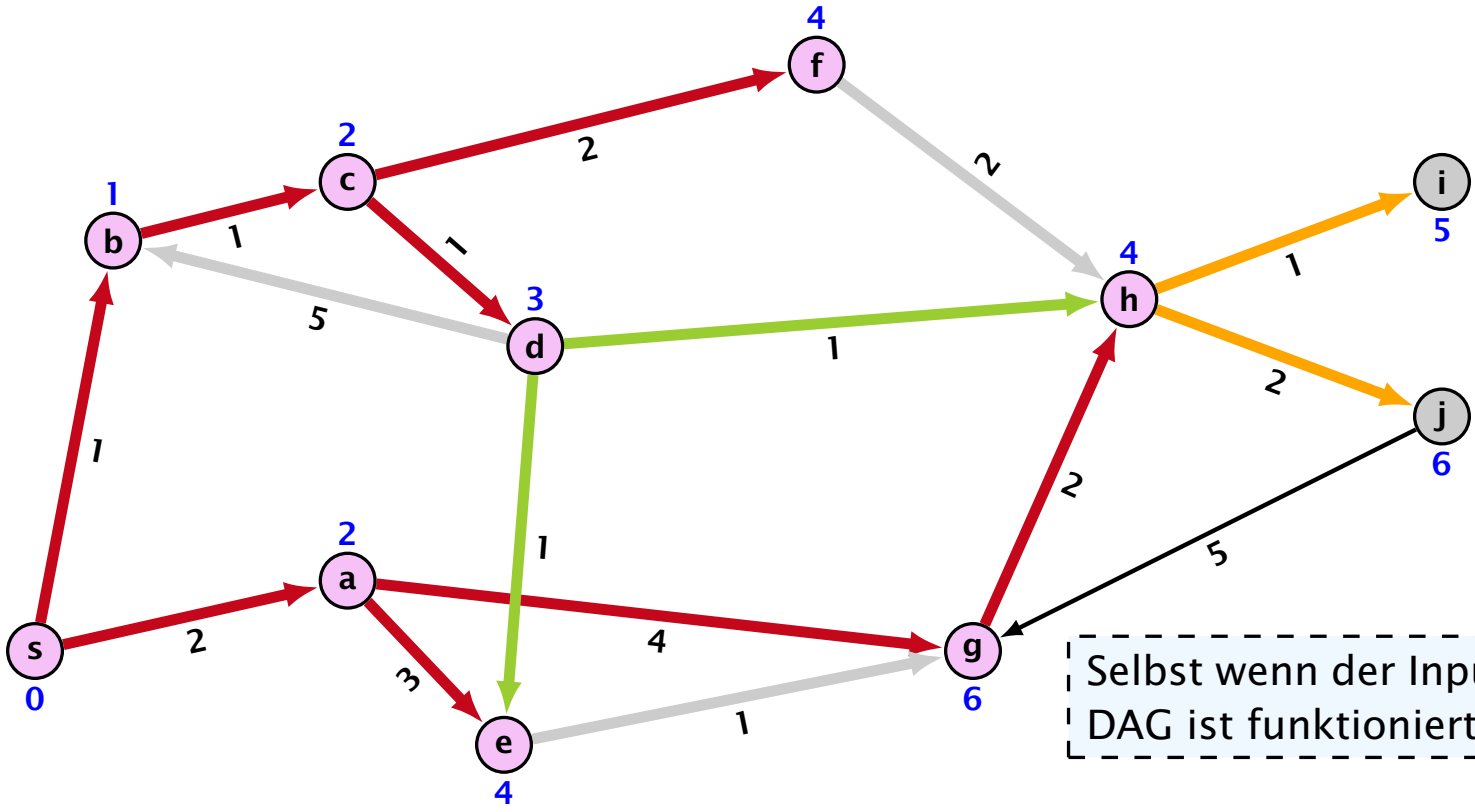

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



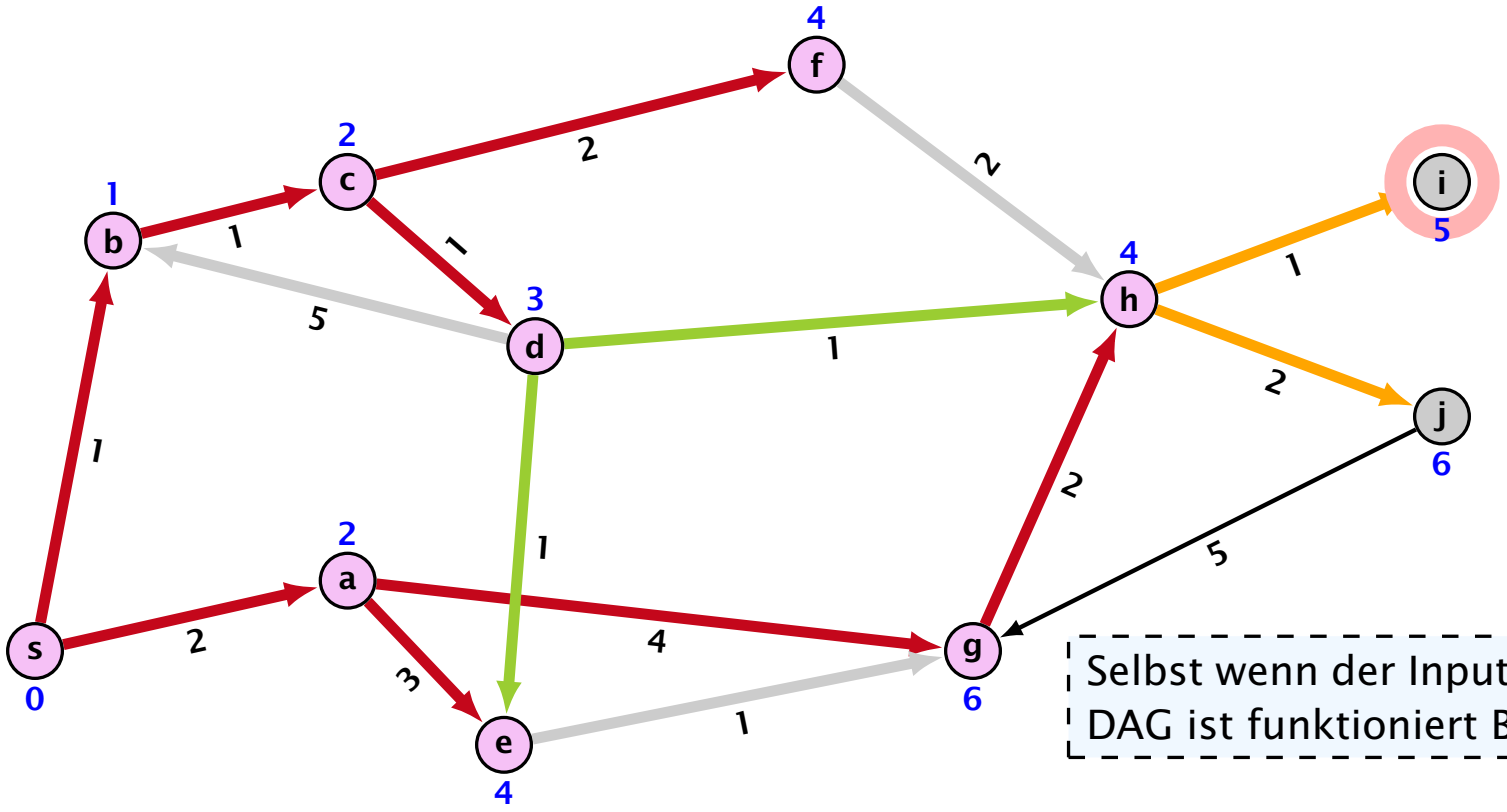

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



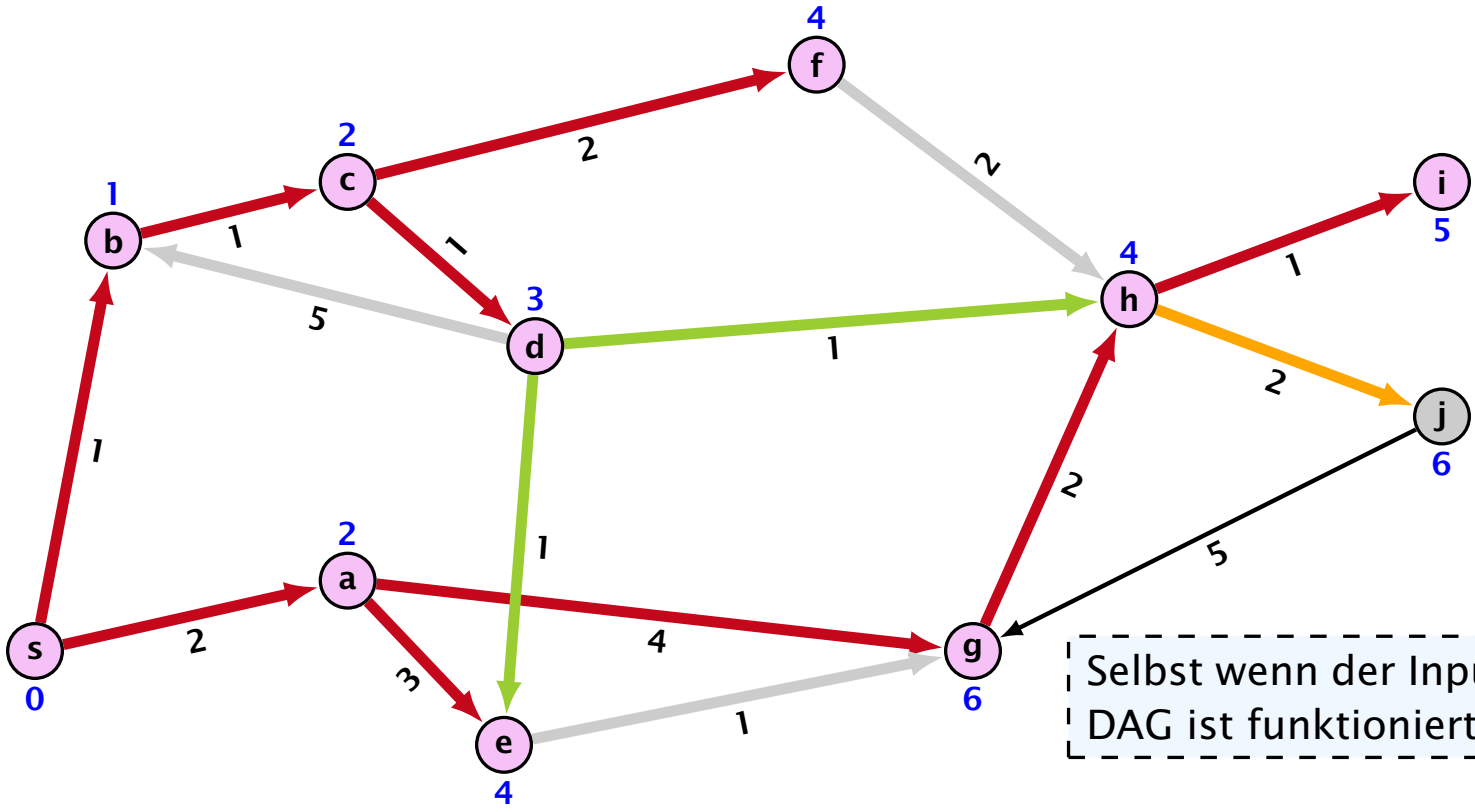

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



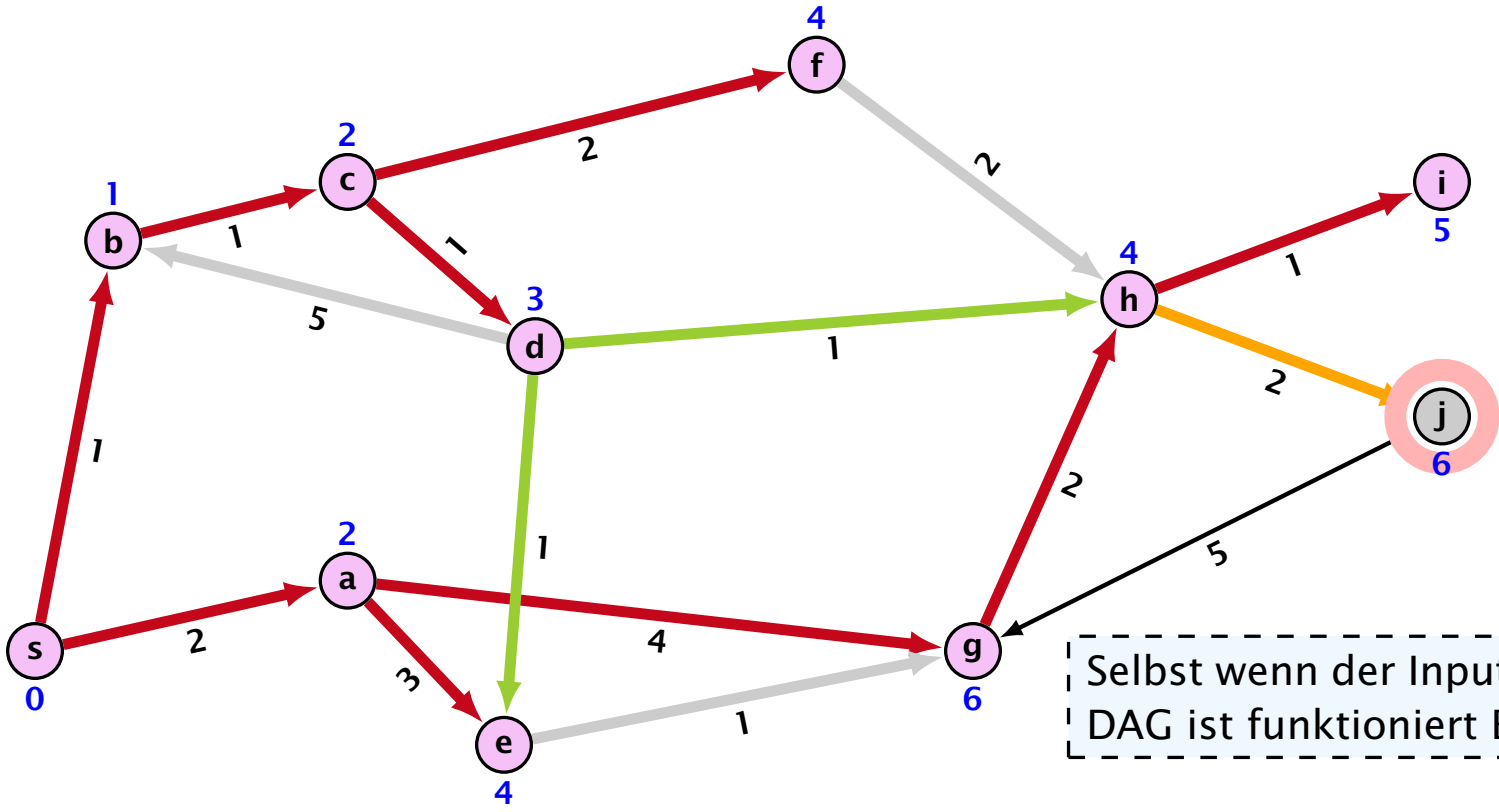

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



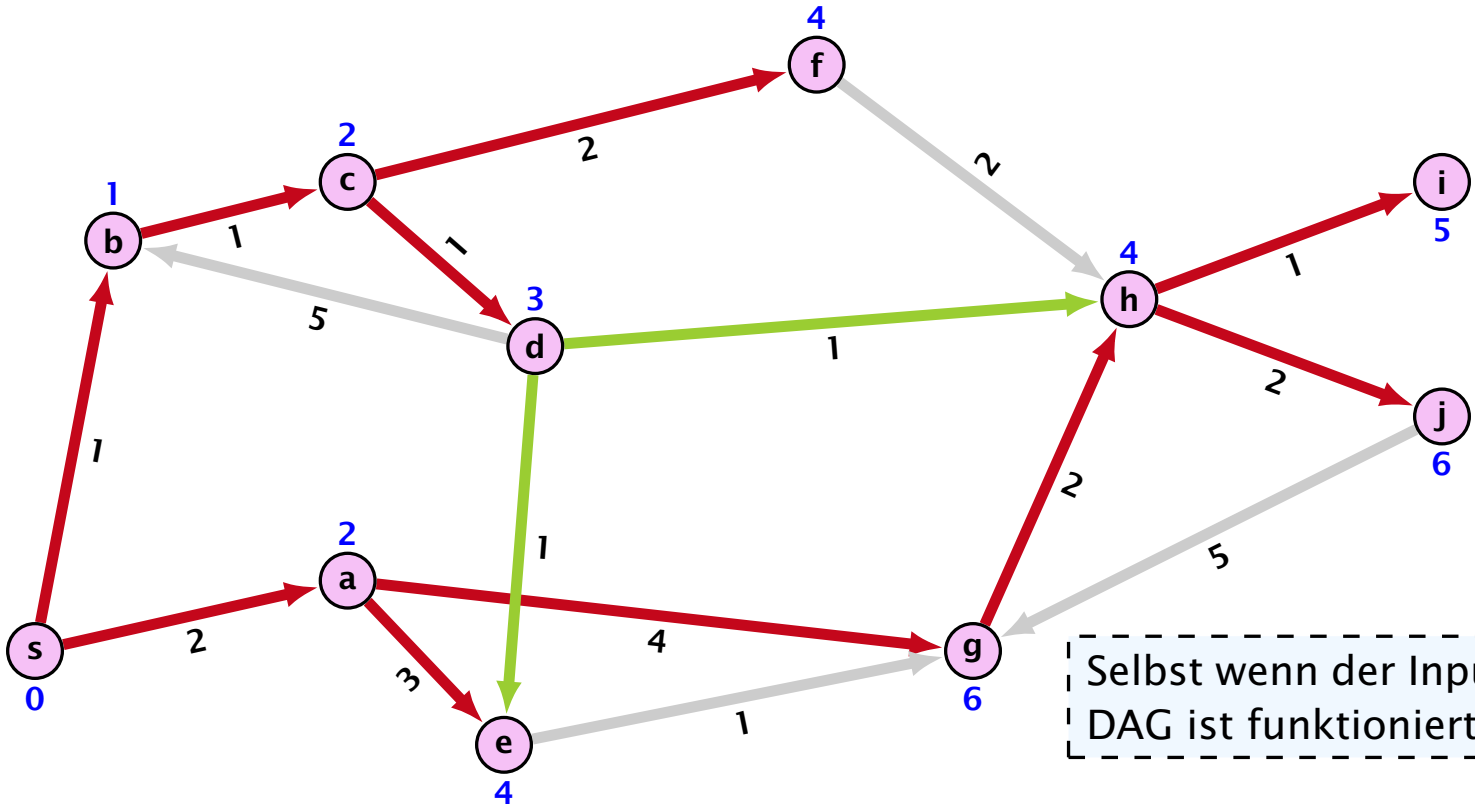

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht




Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Kürzeste Wege in DAGs

finNum(v) ist die bei DFS berechnete Reihenfolge, in der die rekursiven Aufrufe von dfsVisit(v) enden.

In DAGs existiert topologische Sortierung der Knoten:

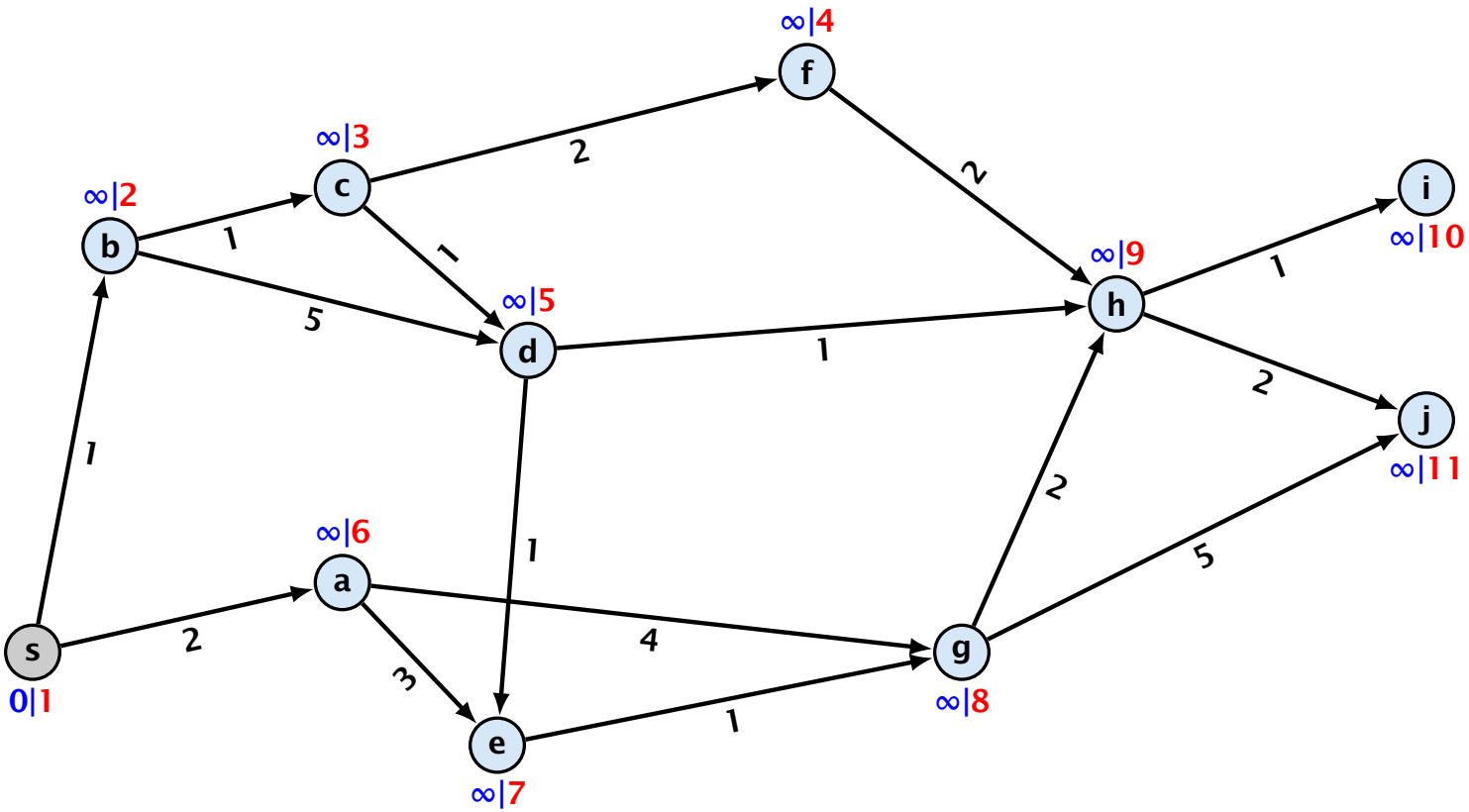
- $\text{top} : V \to \{1, \ldots, n\}$, bijektiv
- für alle $(x, y) \in E$: $\text{top}(x) < \text{top}(y)$

Zum Beispiel: $\text{top}(v) = n - \text{finNum}(v) + 1$.

```
1  Input: weighted DAG G=(V,E,w); start vertex s;
2         array top mit top[i] i-ter Knoten in TopSort
3  Output: key-field of node contains distance from s;
4
5  ShortestPathDag(s,top)
6      foreach v ∈ V
7          v->key = ∞;
8      s->key = 0;
9      for i = 1 to n do
10         v = top[i];
11         foreach x ∈ N[v]
12             x->key = min(v->key + w(v,x),x->key);
```

# Kürzeste Wege in DAGs

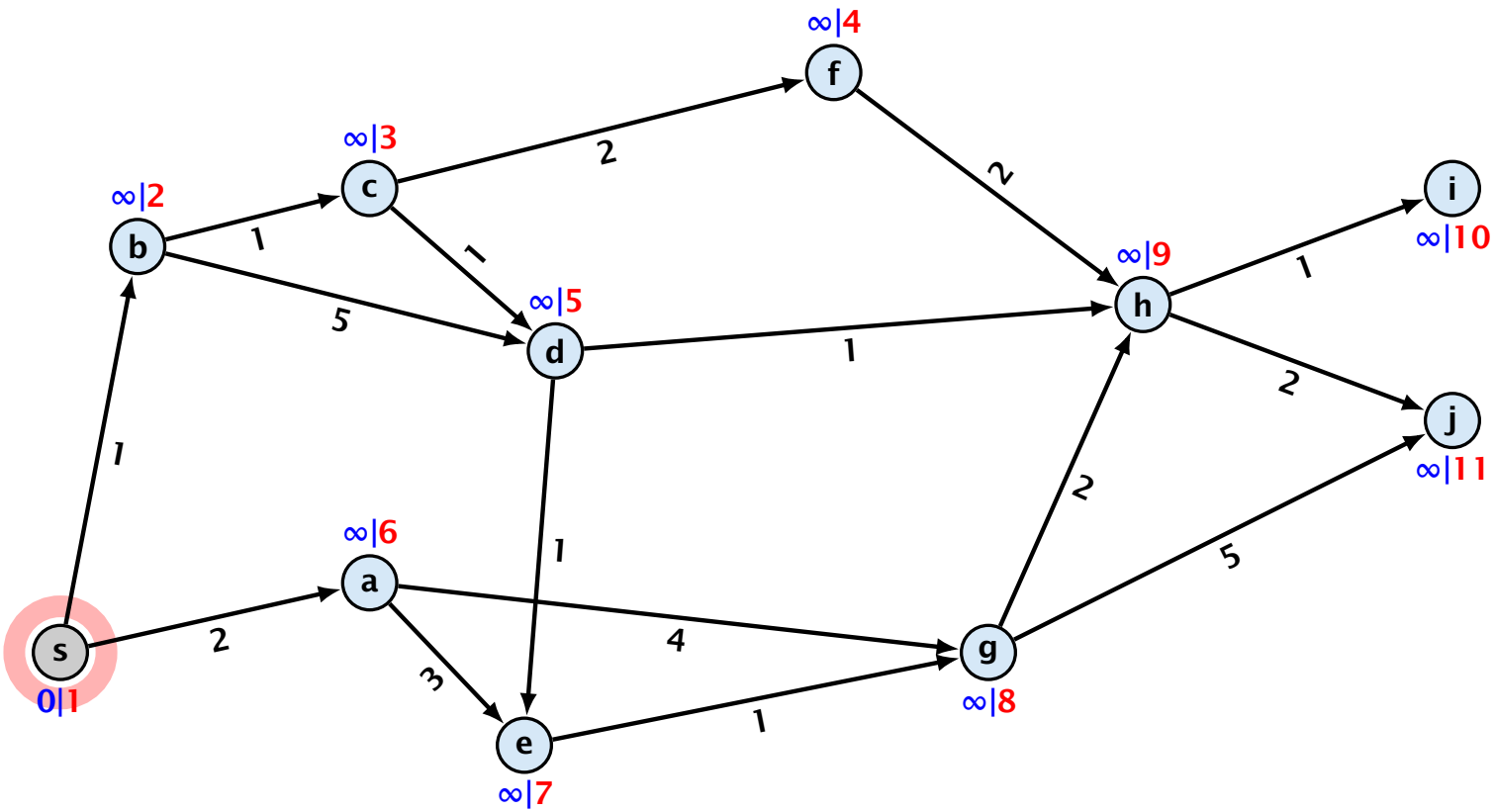
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

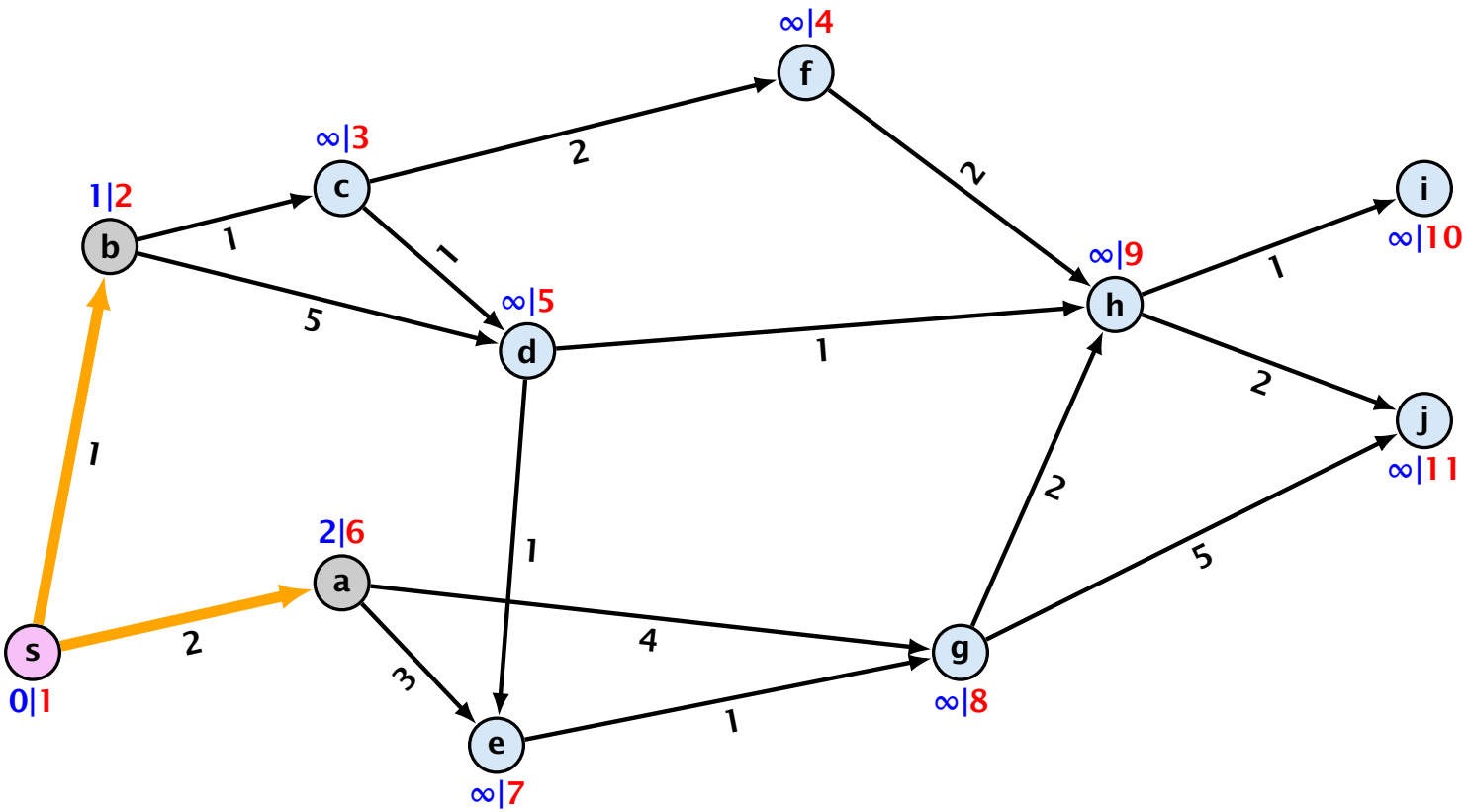
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

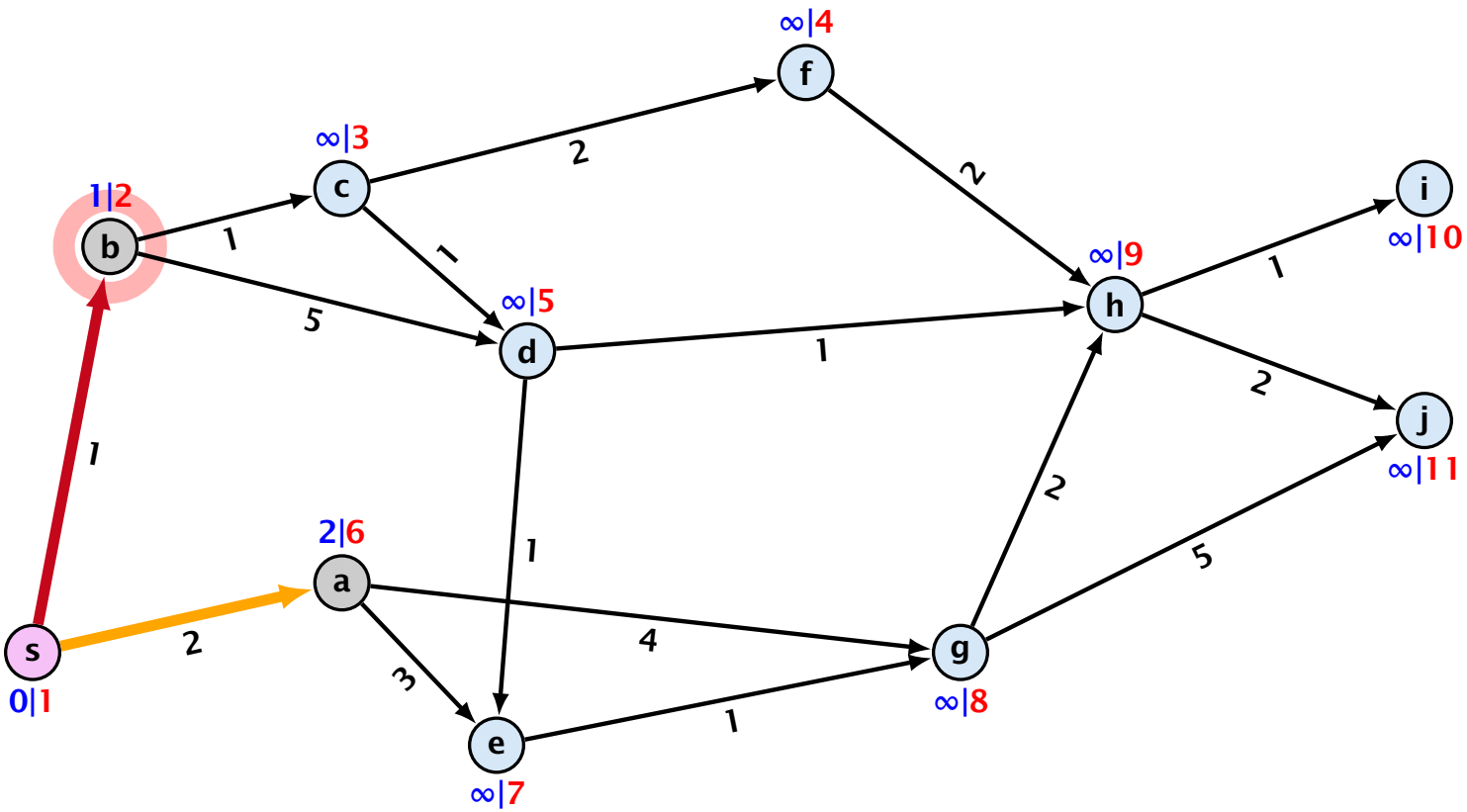
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

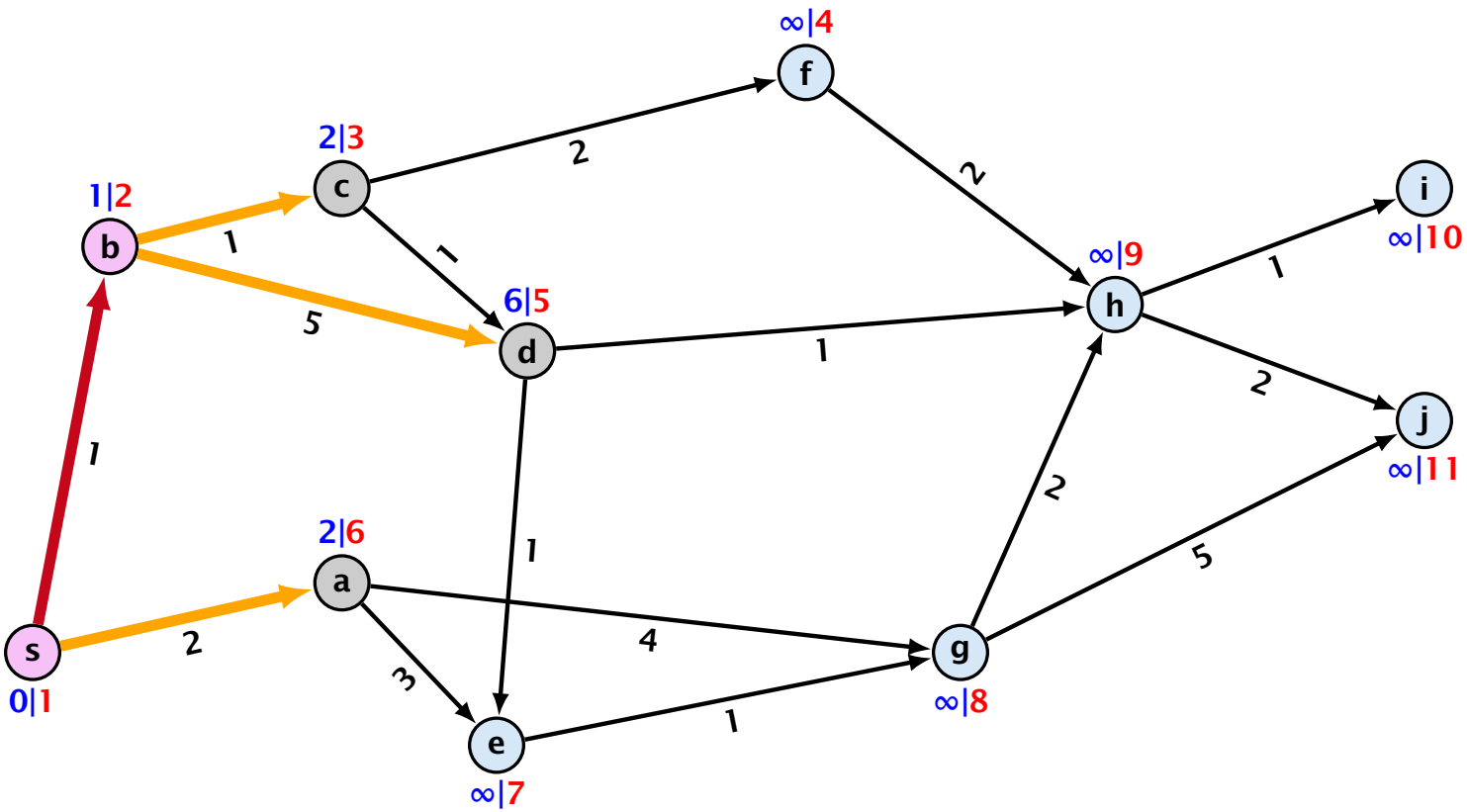
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

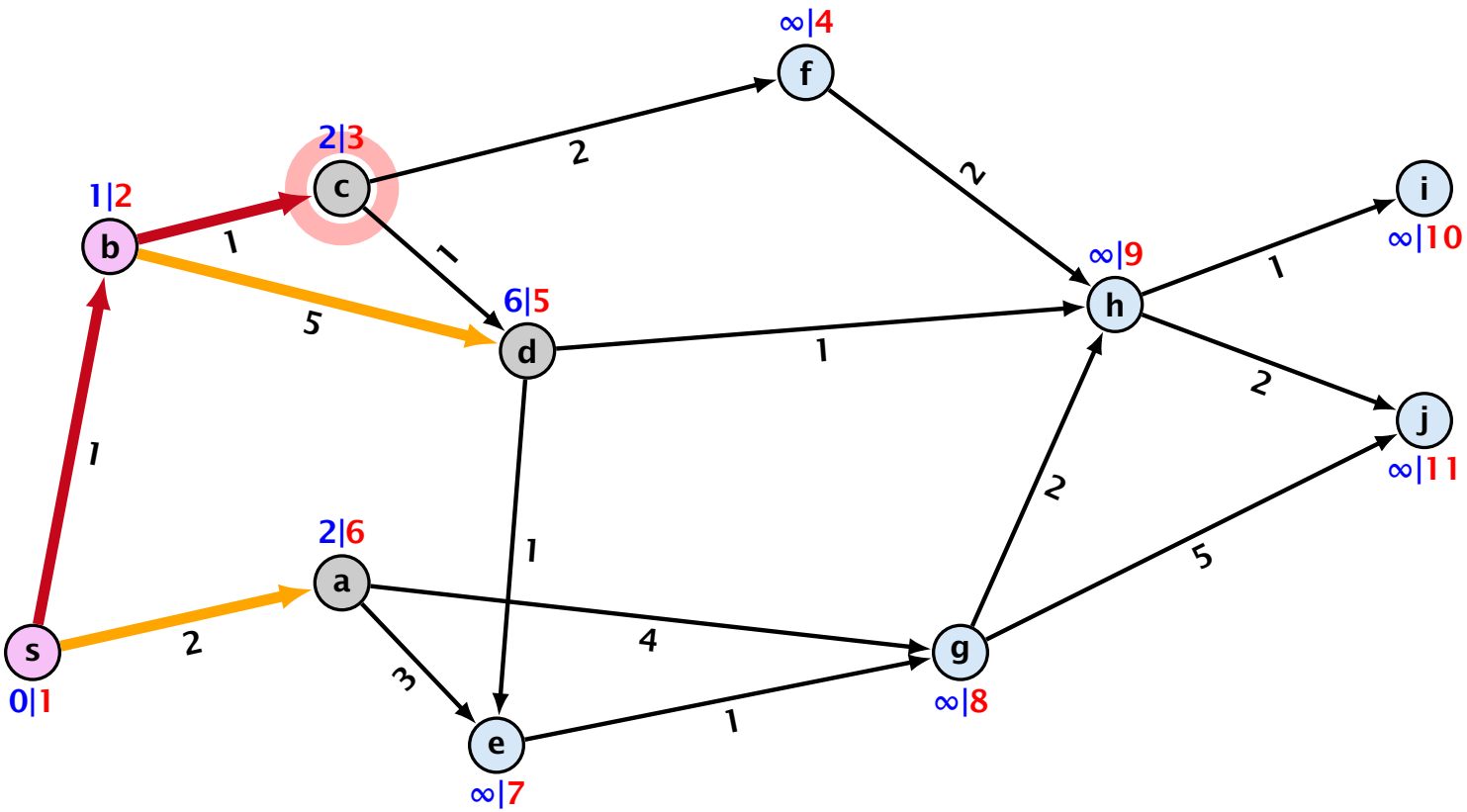
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

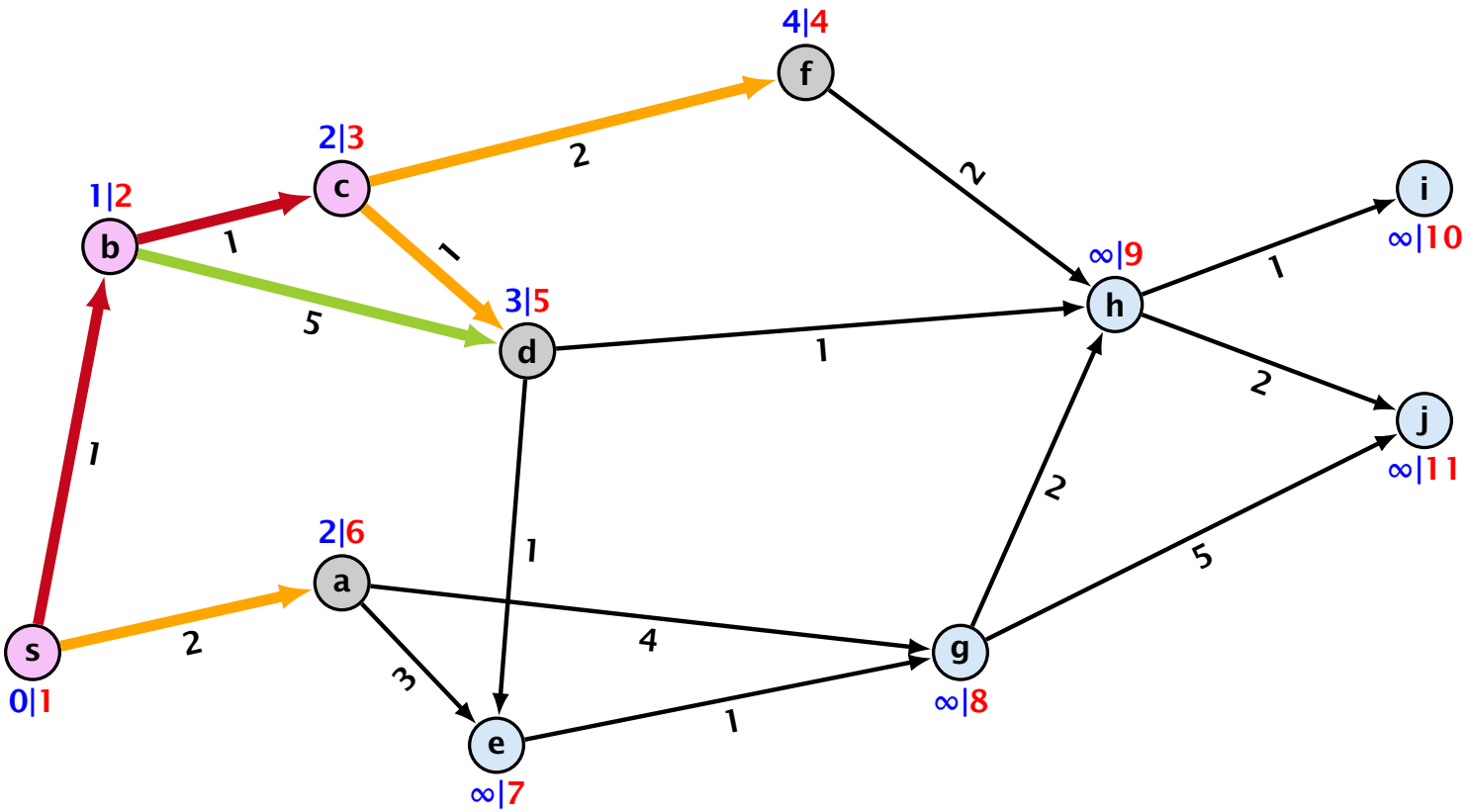
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

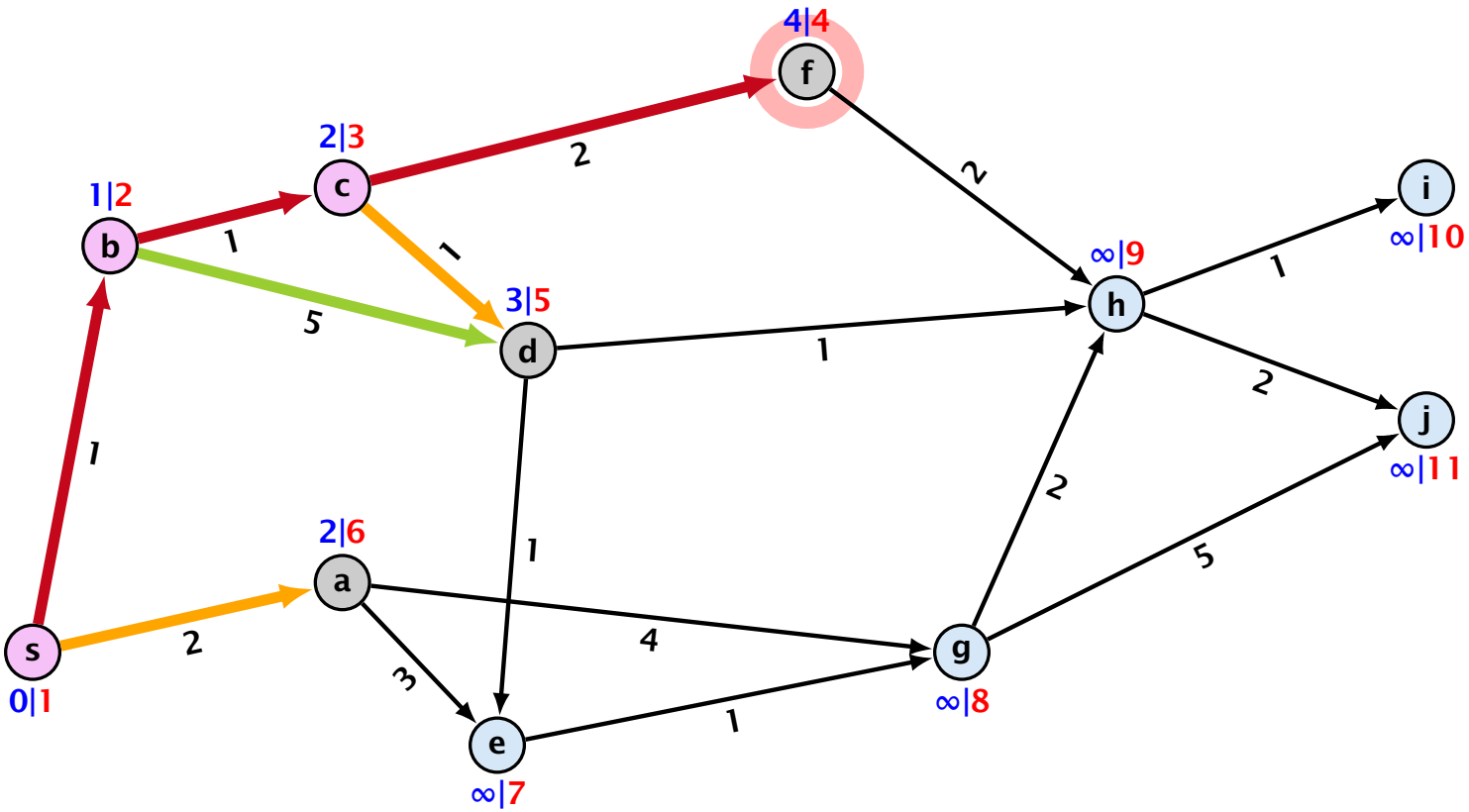
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

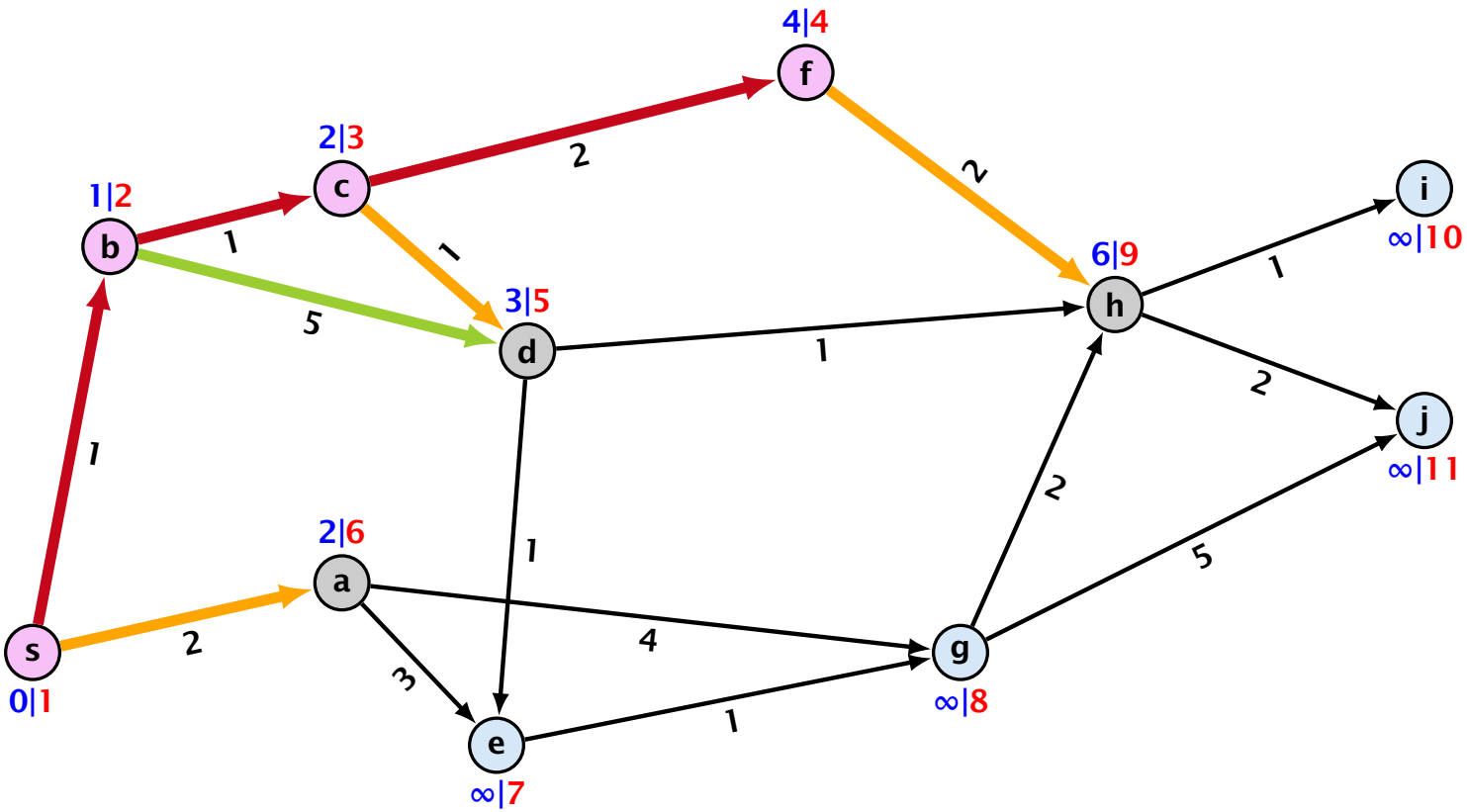
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

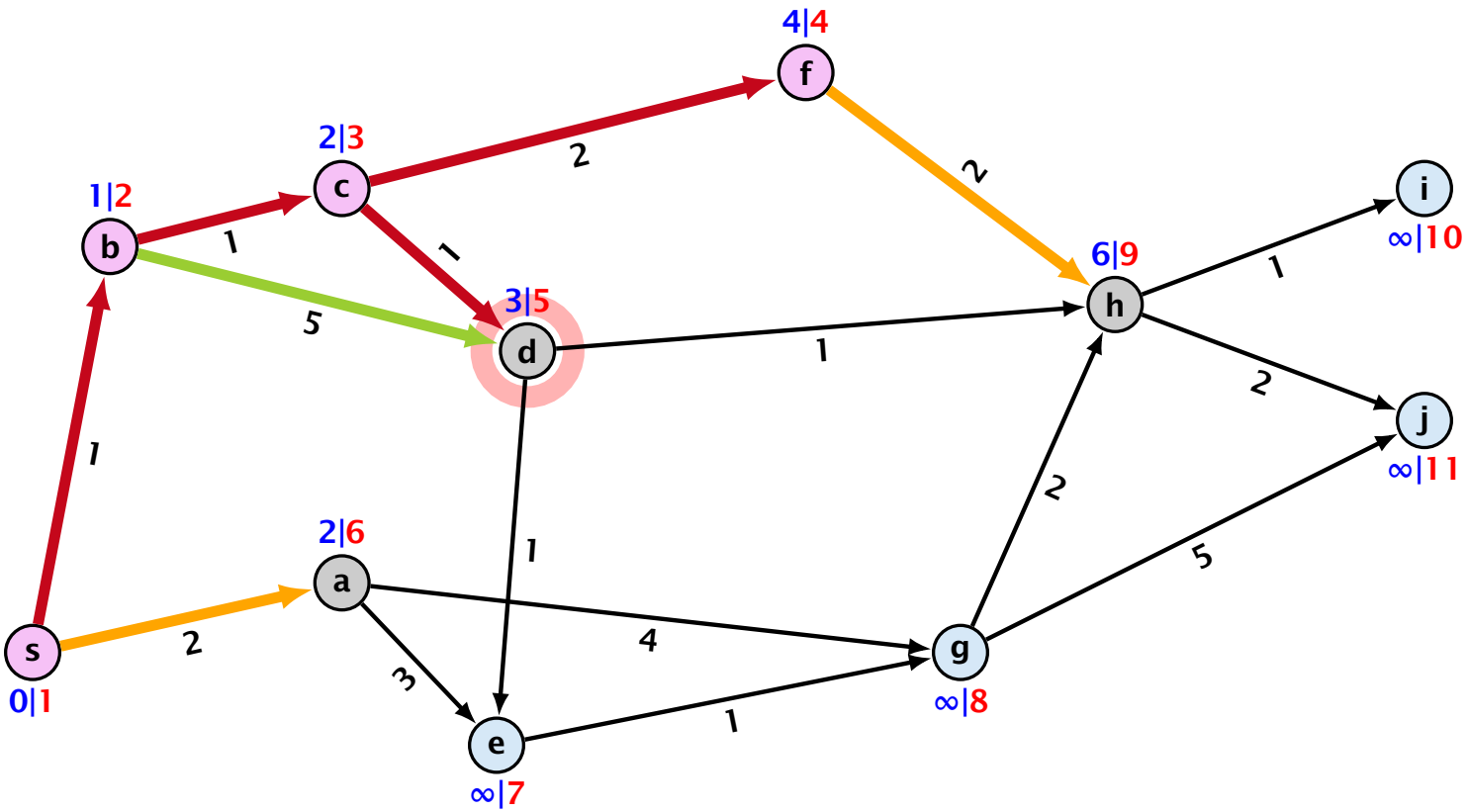
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

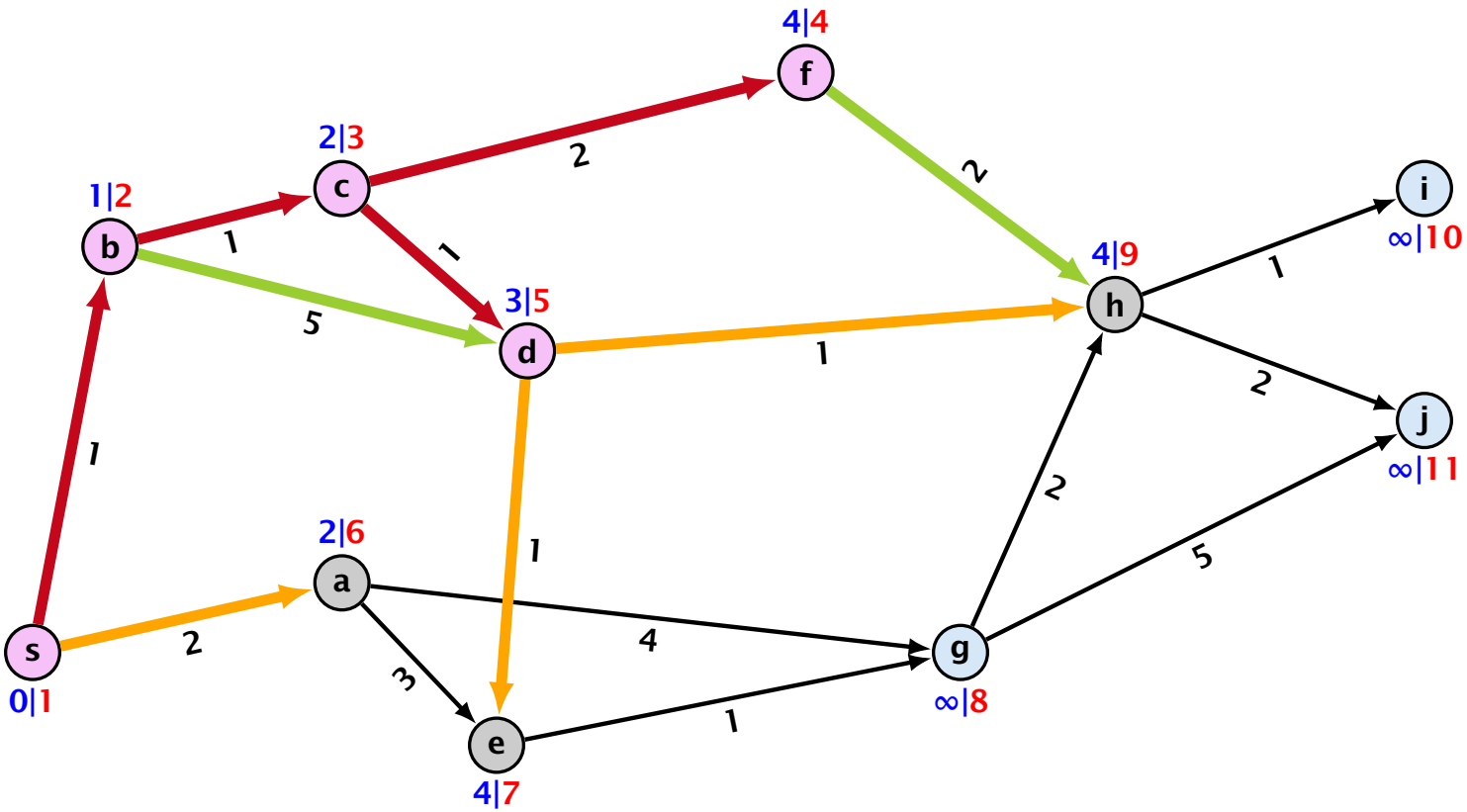
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

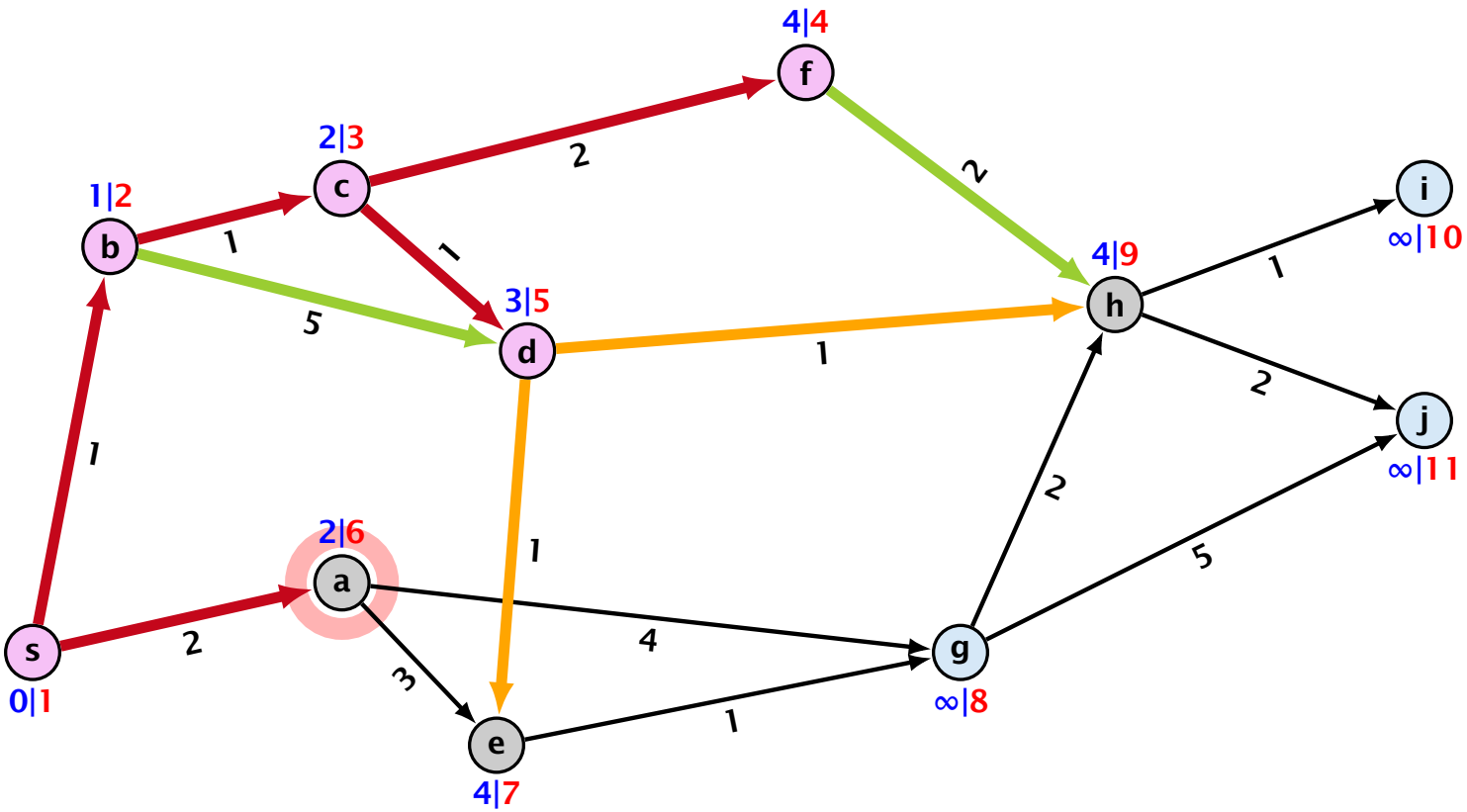
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

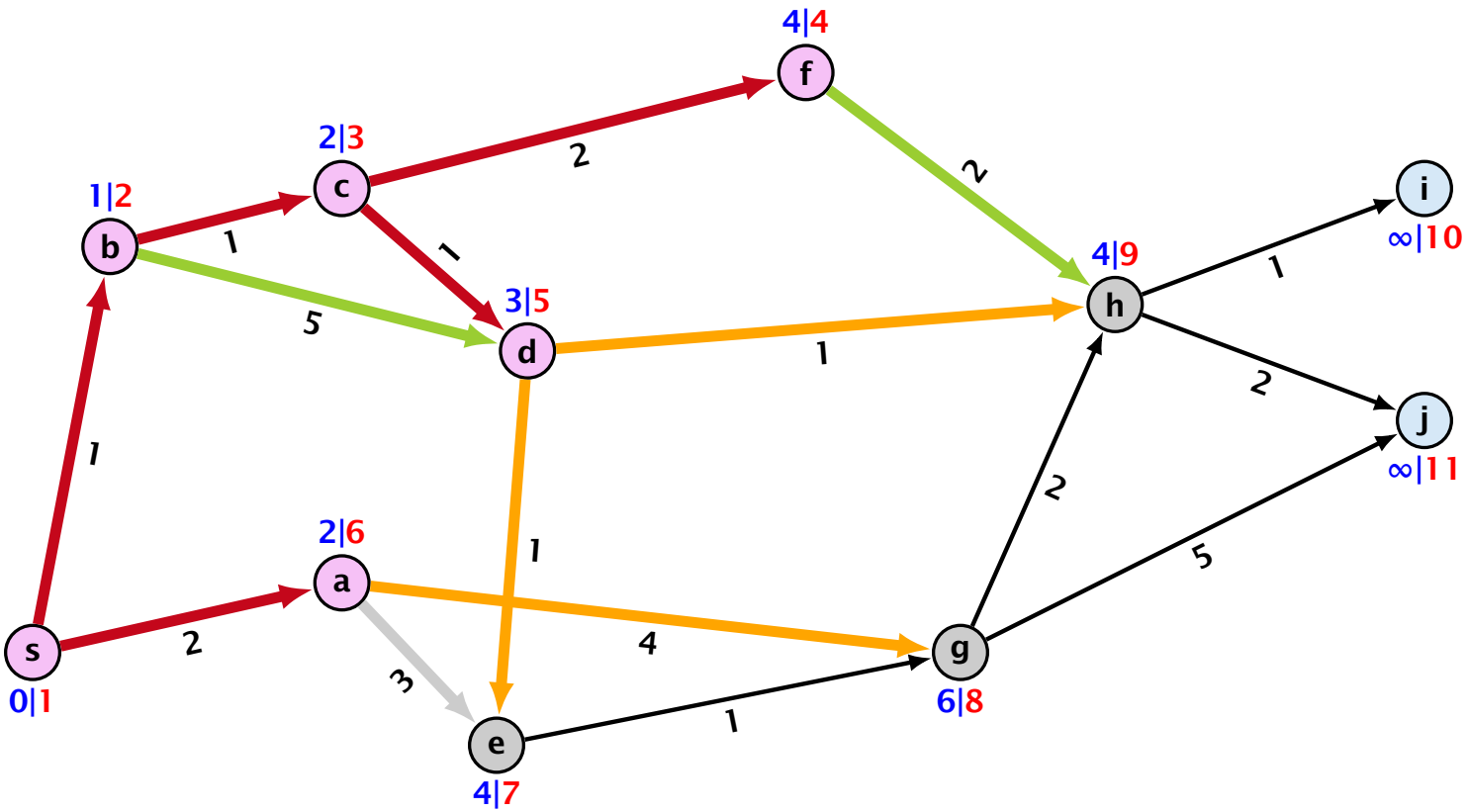
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

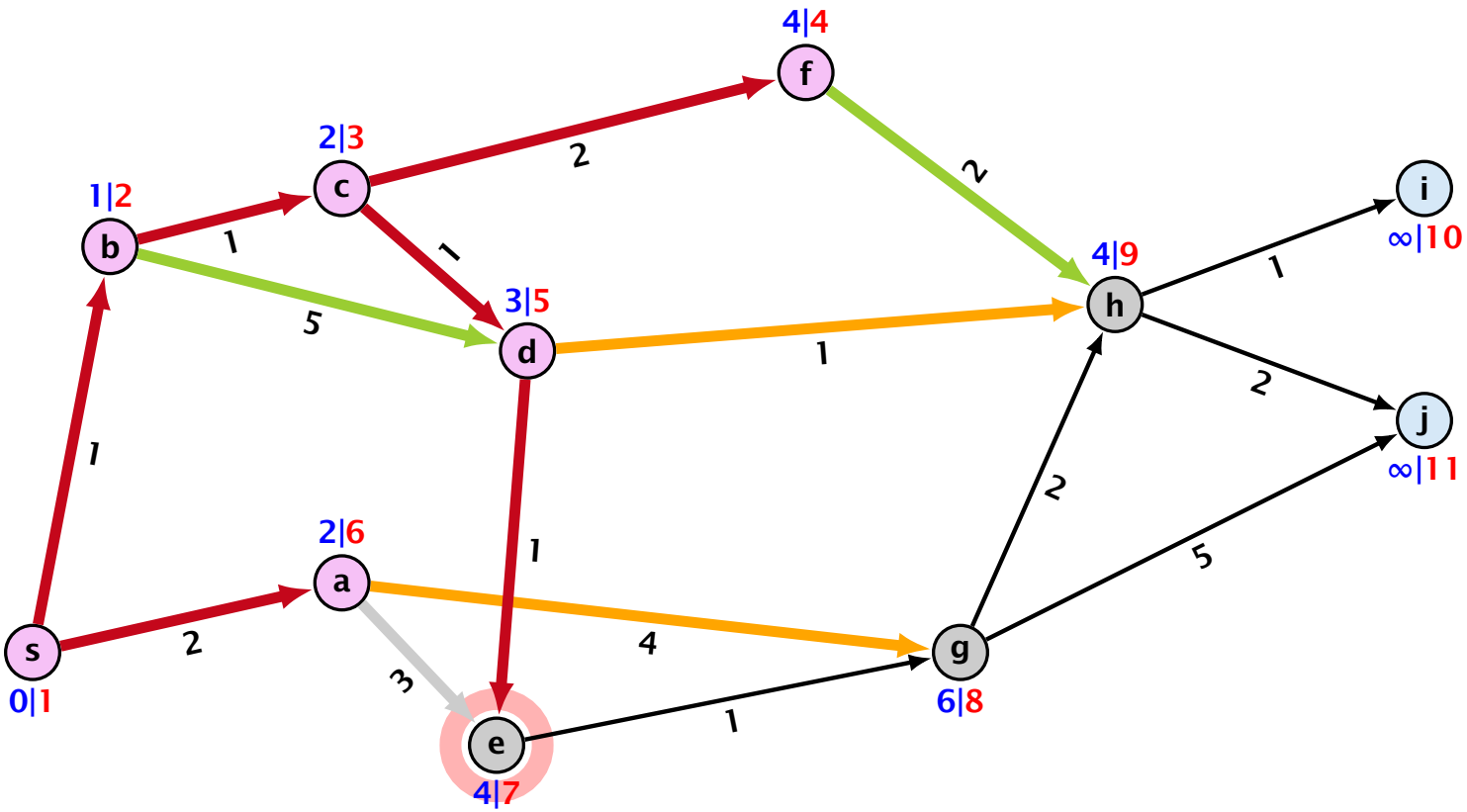
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

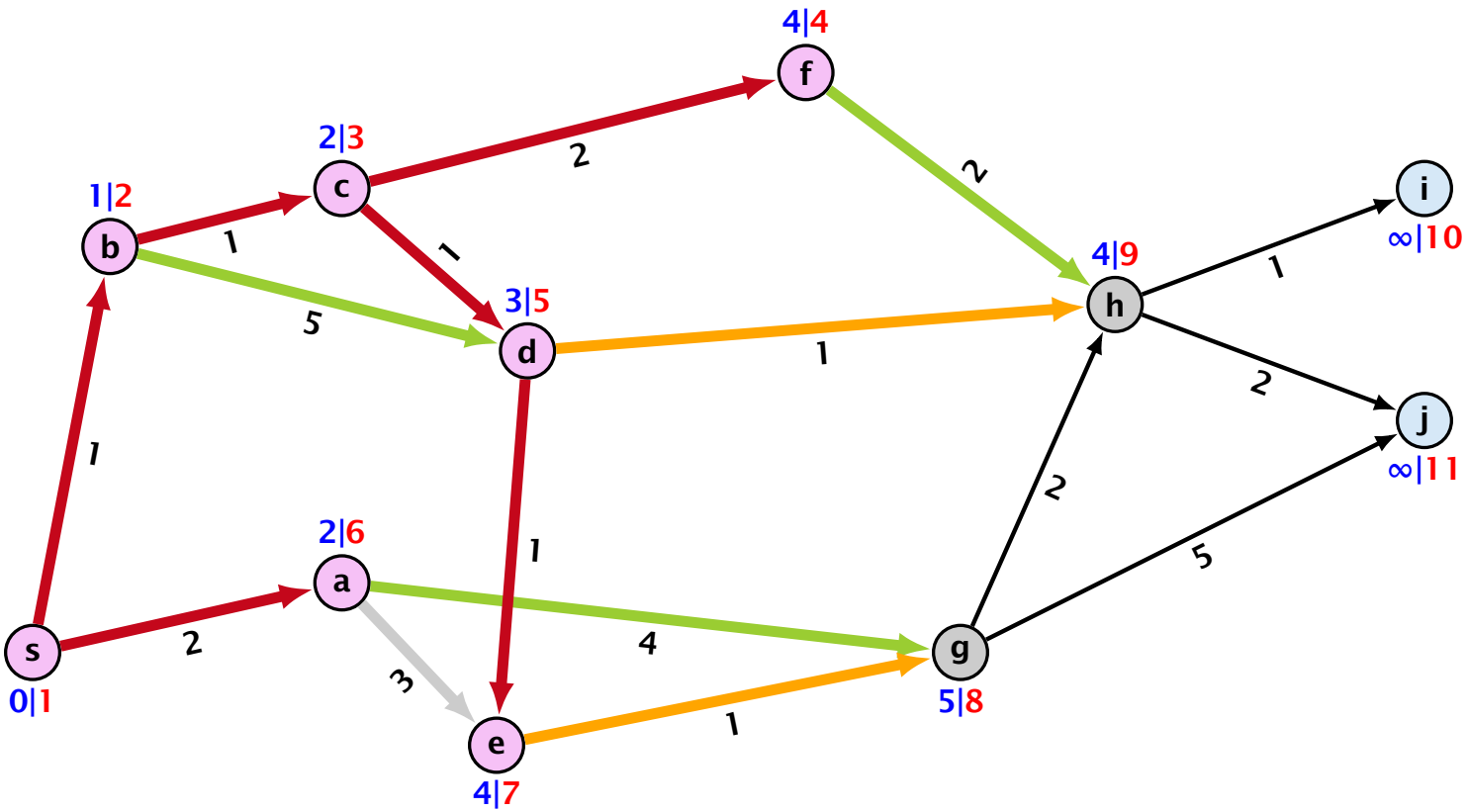
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

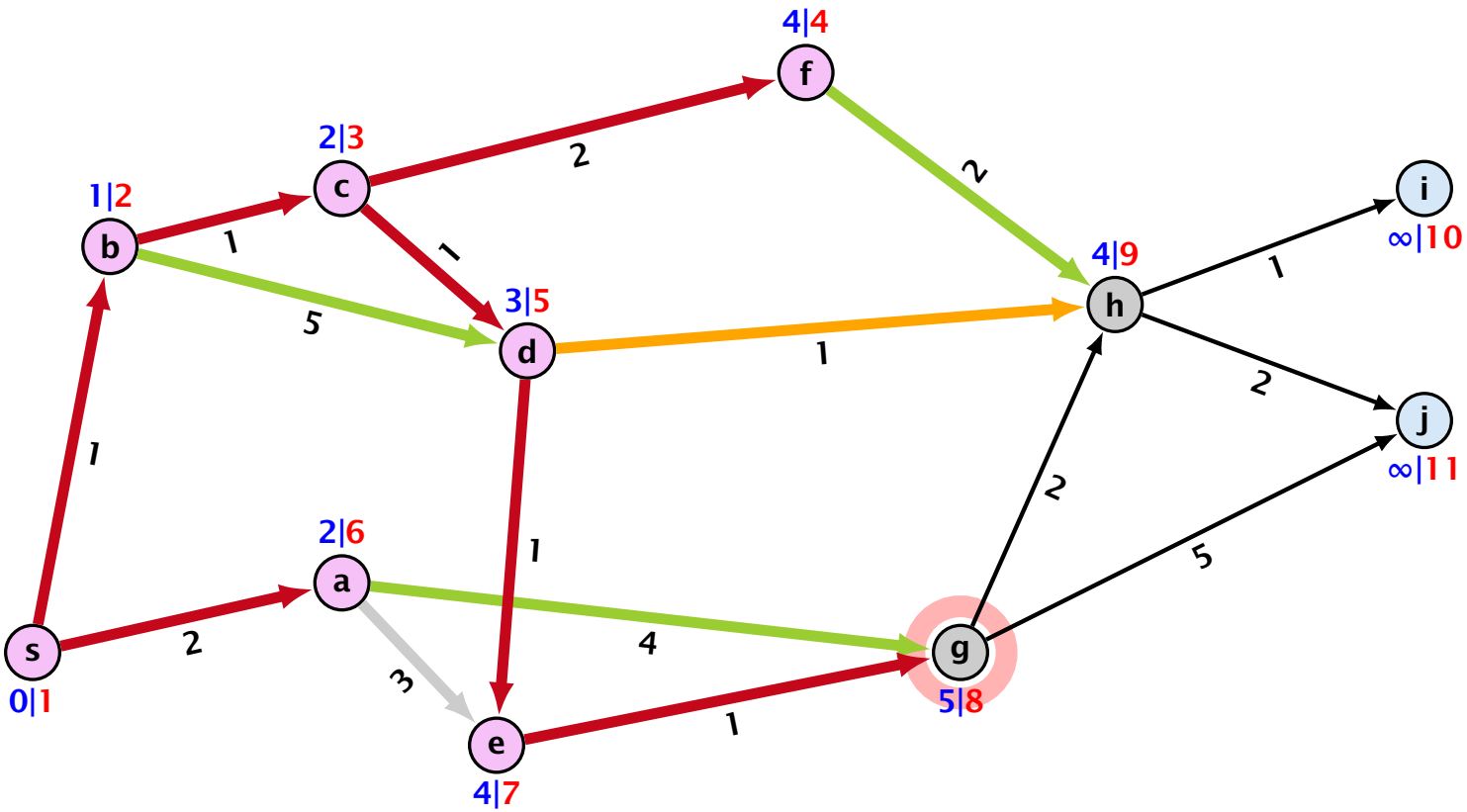
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

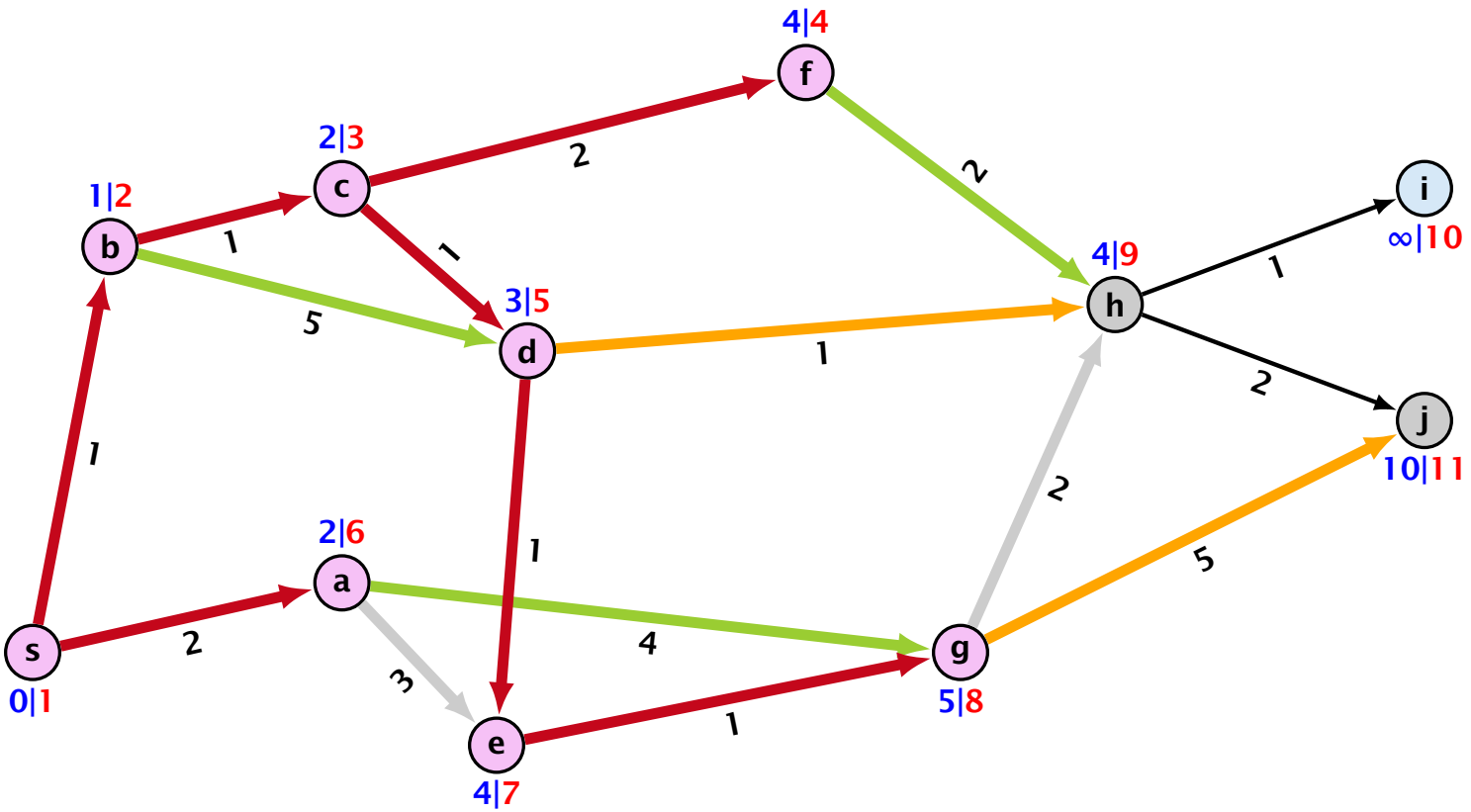
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

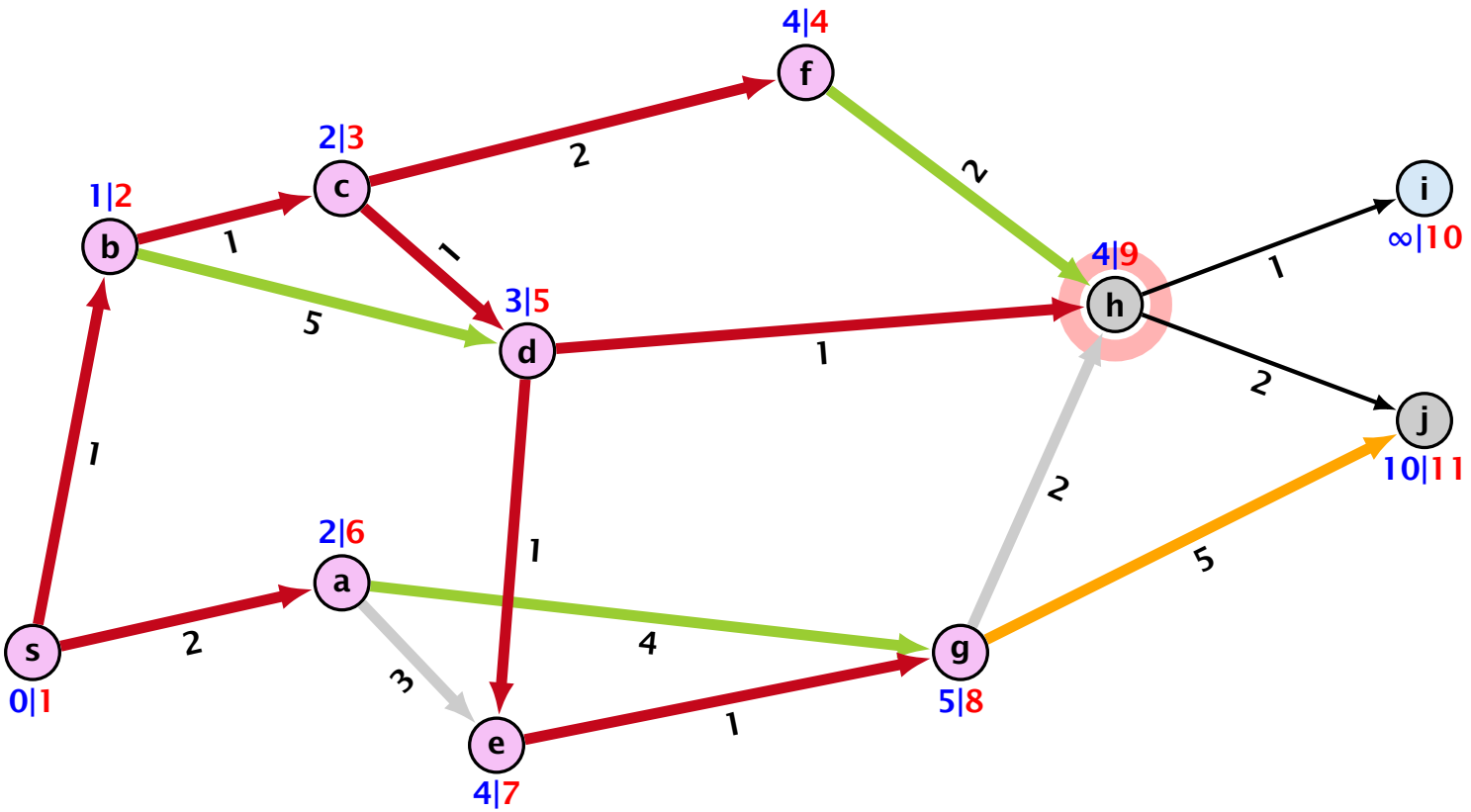
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

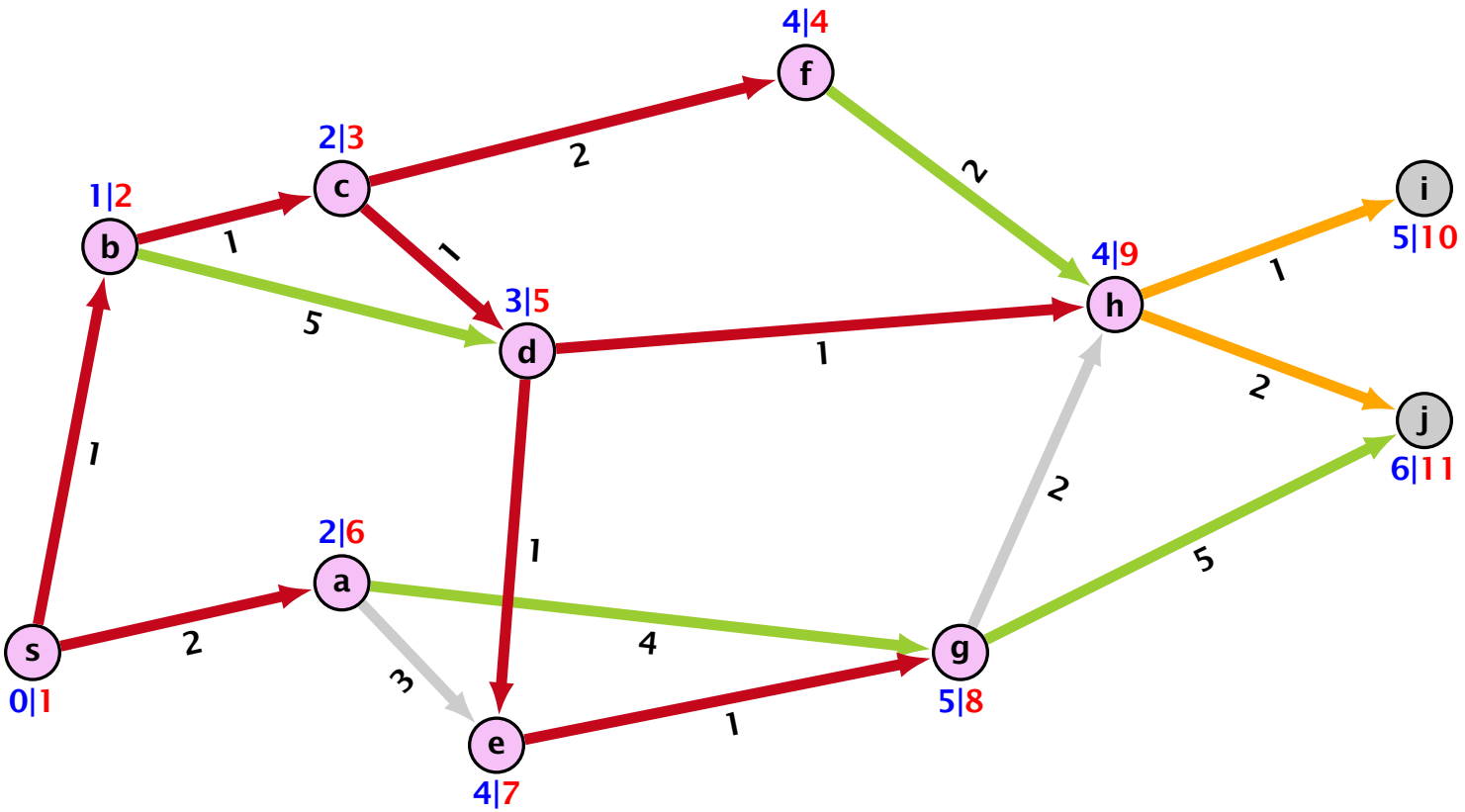
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

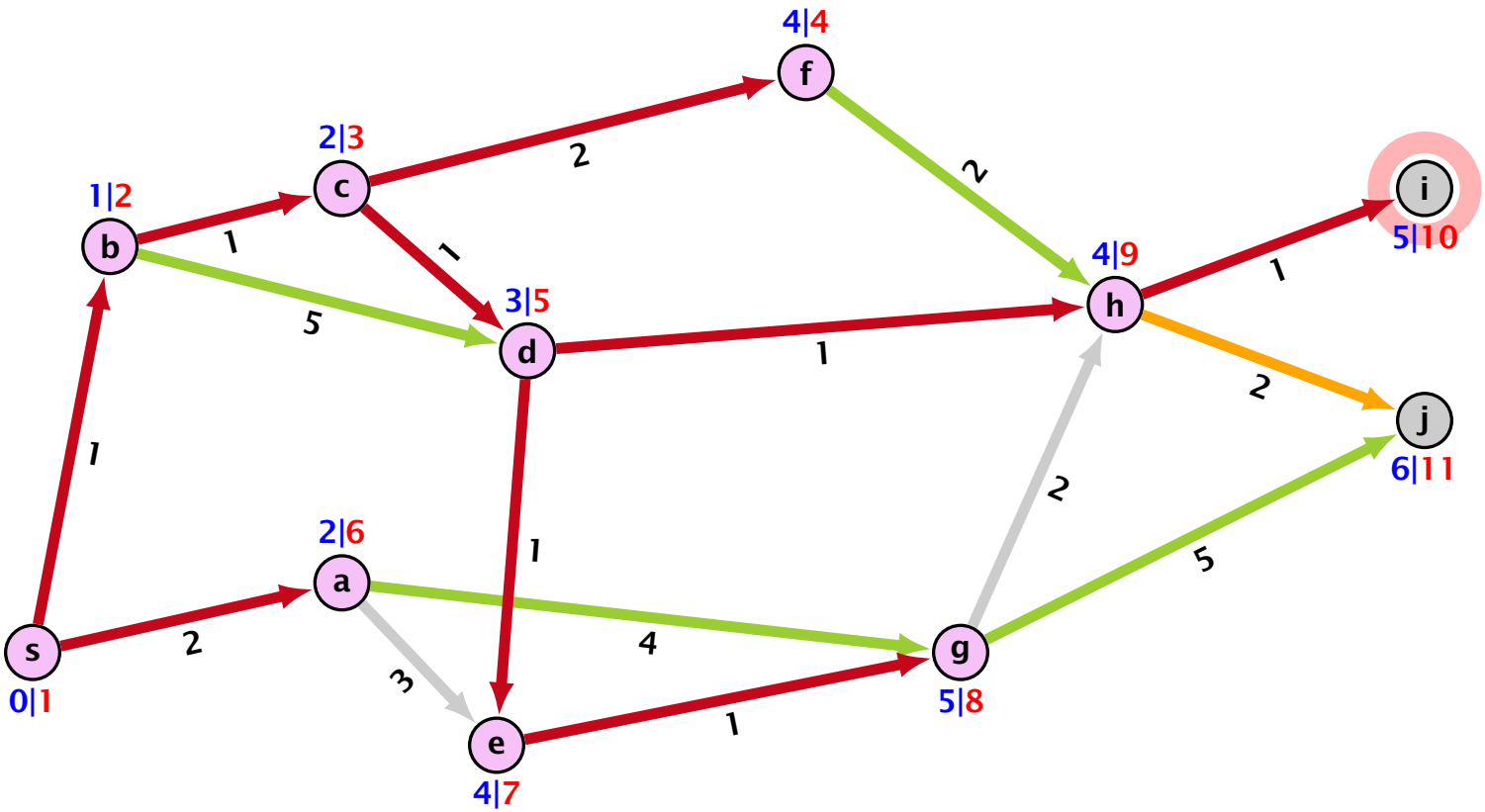
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

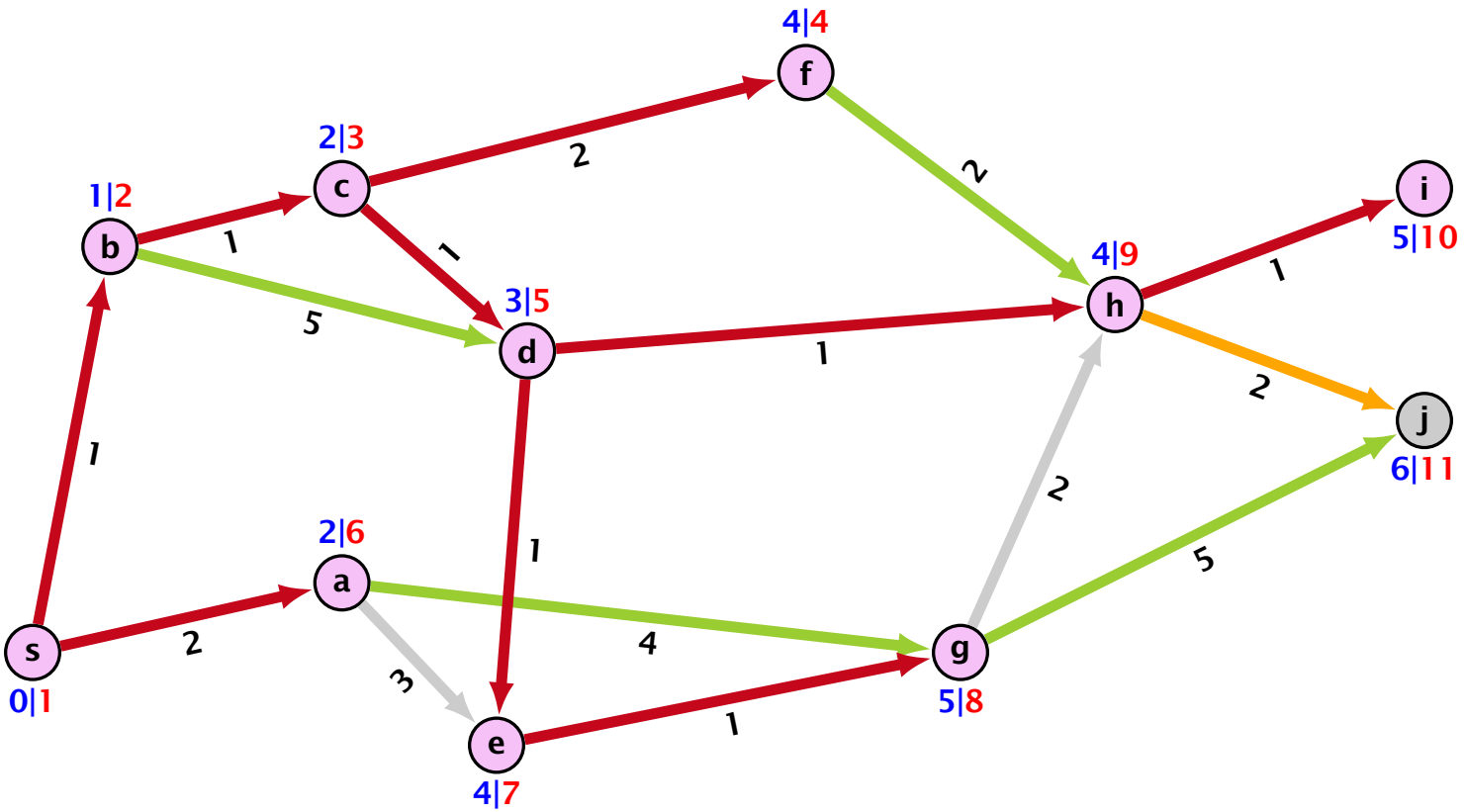
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

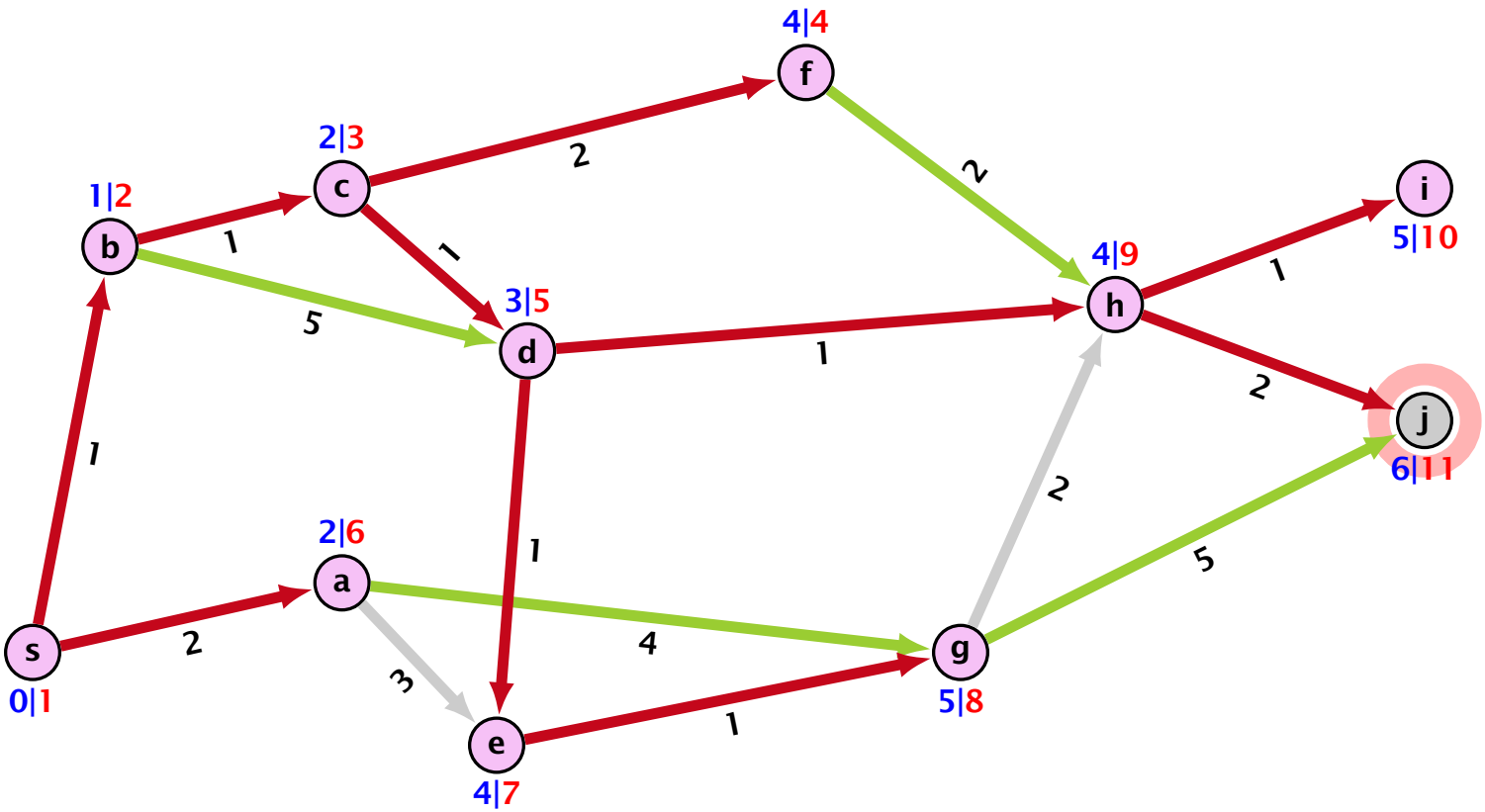
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

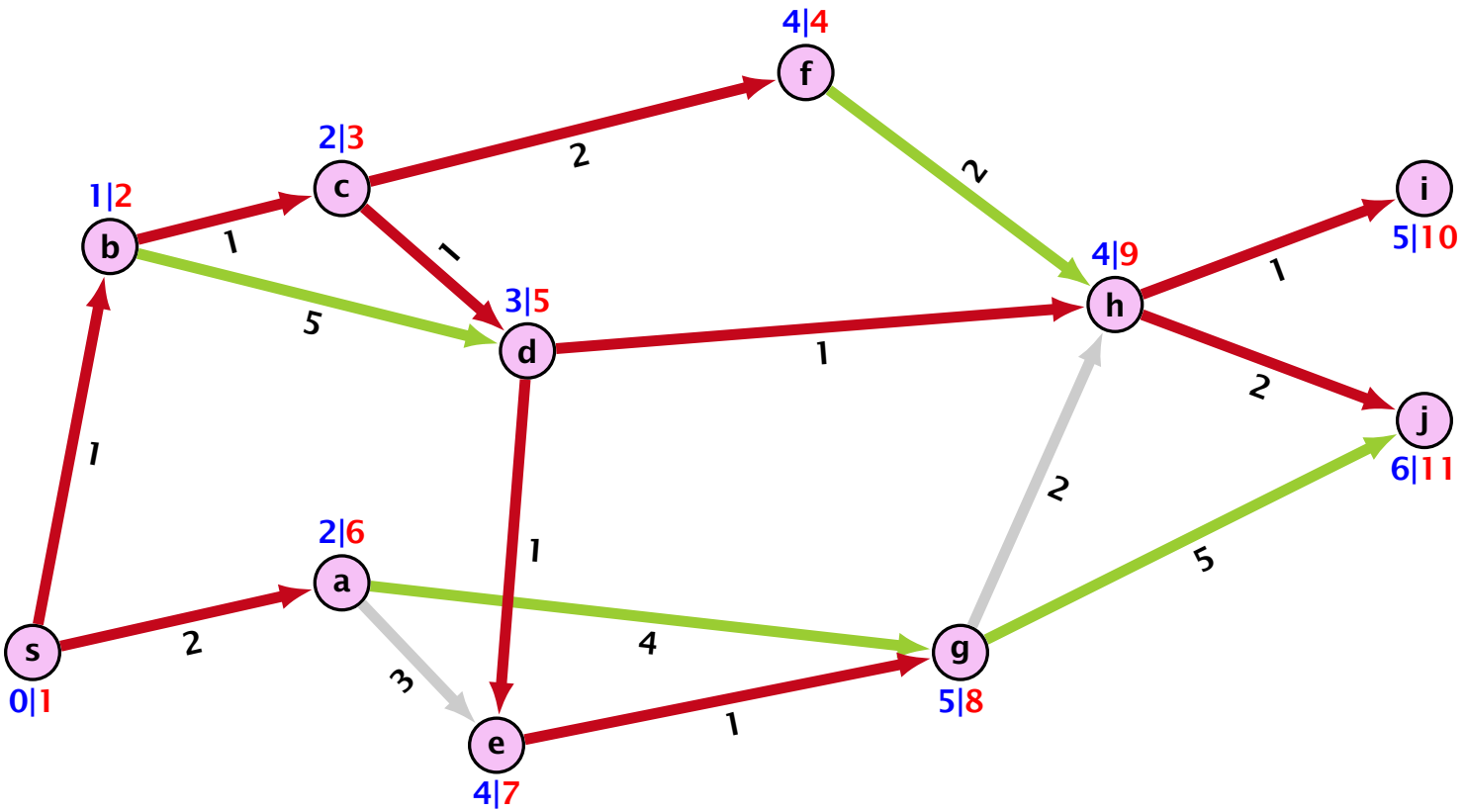
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

**Korrektheit:**
Für **jeden** Pfad werden die Relaxierungen in der Reihenfolge des Pfades durchgeführt.

# Kürzeste Wege in DAGs

**Laufzeit:**

DFS für topologische Sortierung

- Laufzeit $\mathcal{O}(n+m)$

Danach besucht der Algorithmus jede Kante genau einmal.
Zusätzlich wird jeder Knoten besucht:

- Laufzeit $\mathcal{O}(n+m)$

Also, Gesamtlaufzeit $\mathcal{O}(n+m)$.

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

Idee:

- Distanzrelaxierungen entlang eines kürzesten $s$-$x$ für alle $x \in V$, $x \neq s$.

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

Idee:

- Distanzrelaxierungen entlang eines kürzesten $s$-$x$ für alle $x \in V$, $x \neq s$.

Problem:

- führe Relaxierungen in der richtigen Reihenfolge durch

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

Idee:

- Distanzrelaxierungen entlang eines kürzesten $s$-$x$ für alle $x \in V$, $x \neq s$.

Problem:

- führe Relaxierungen in der richtigen Reihenfolge durch

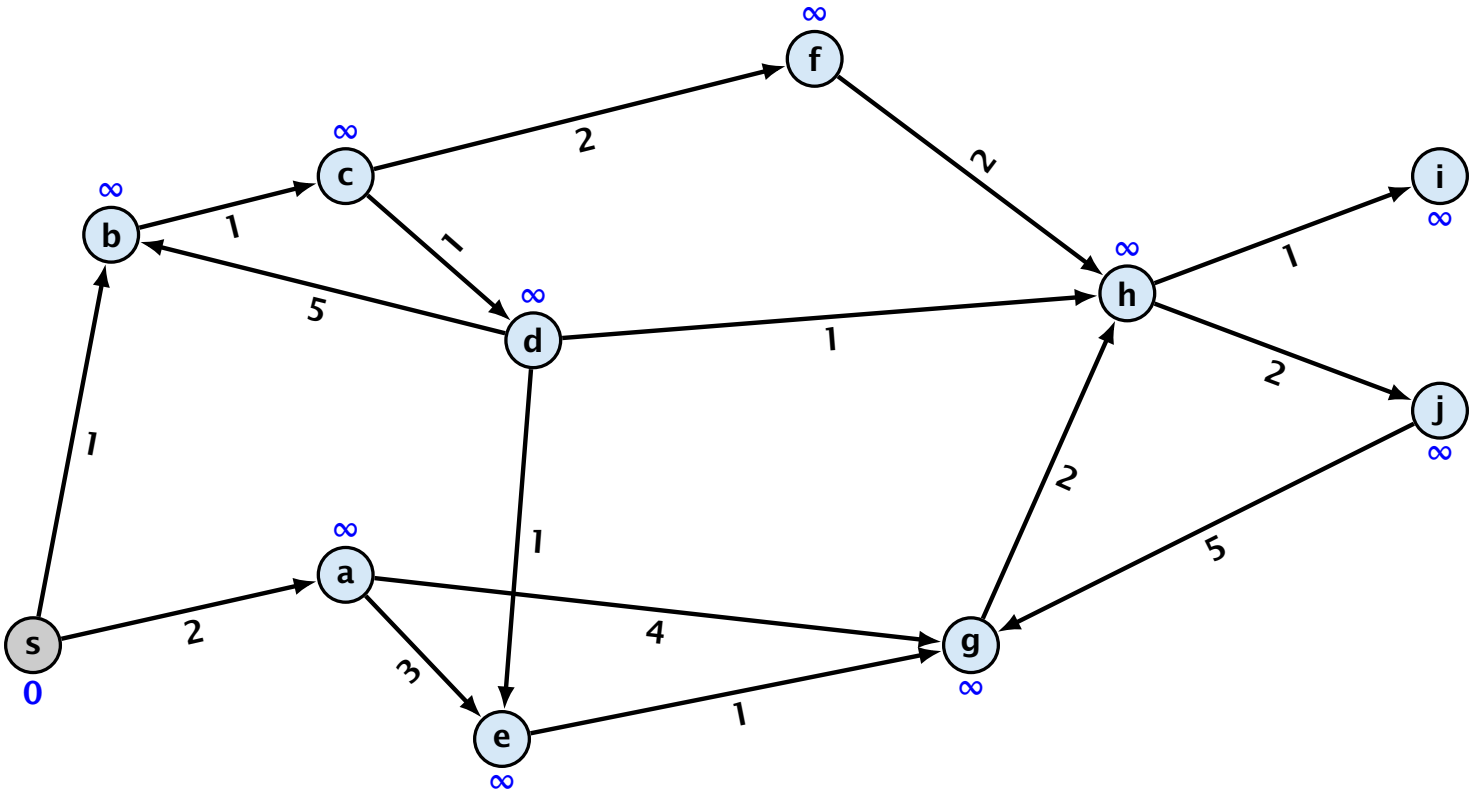
Lösung(?):

- führe Relaxierungen gemäß der Distanz von $s$ durch

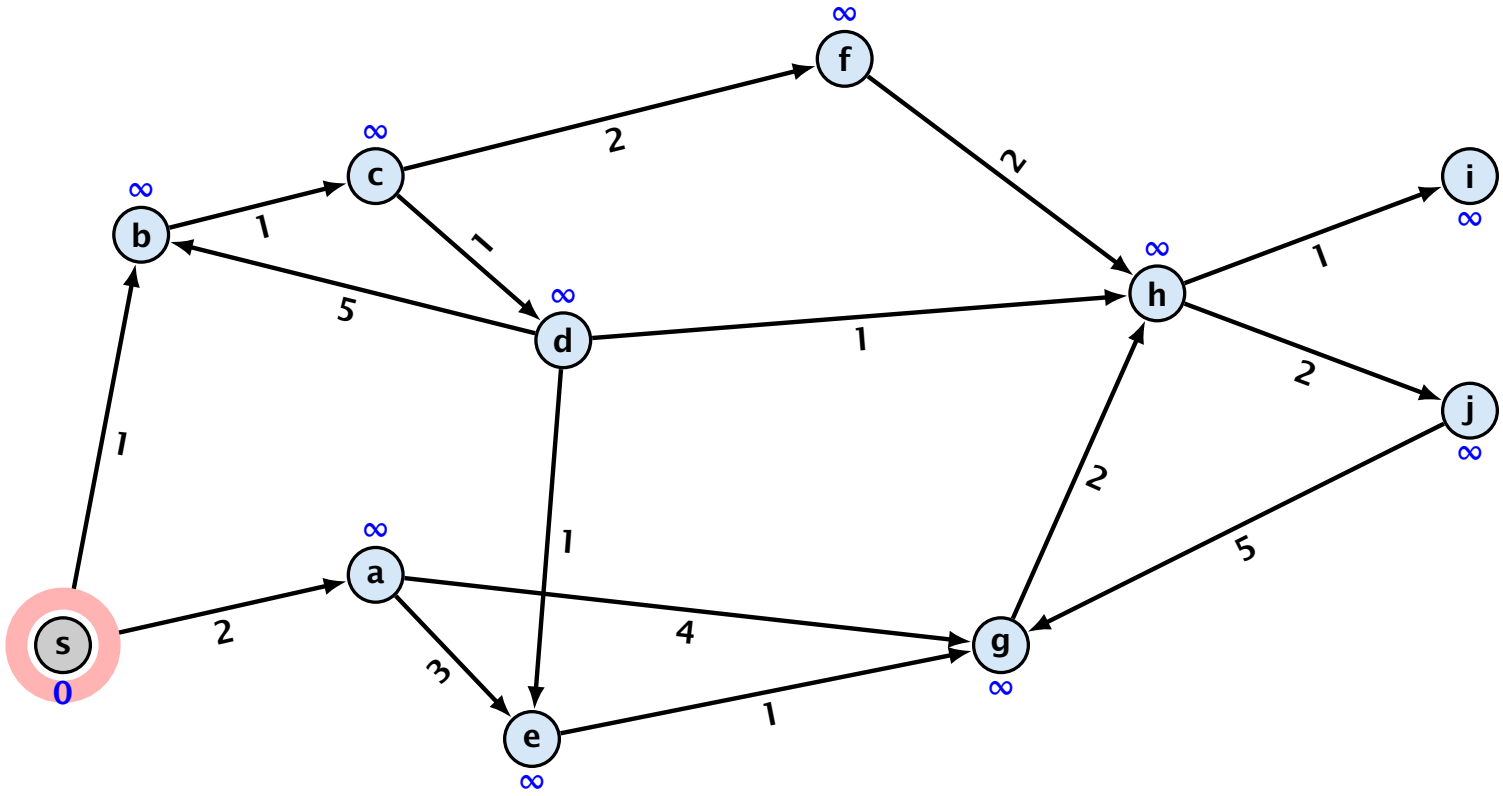
# Dijkstra's Algorithmus

```
1  Input: weighted graph G=(V,E,w); start vertex s;
2  Output: key-field of node contains distance from s;
3
4  Dijkstra(s)
5      S = new PriorityQueue()
6      foreach v ∈ V
7          v->key = ∞;
8          v->par = NULL;
9          v->han = S->insert(v);
10     s->key = 0; S->decreaseKey(s->han,0);
11     while (!S->empty()) {
12         v = S->extractMin();
13         foreach x ∈ N[v]
14             if (x->key > v->key + w(v,x))
15                 S->decreaseKey(x->han,v->key+w(v,x));
16                 x->key = v->key + w(v,x);
17                 x->par = v;
18     }
```

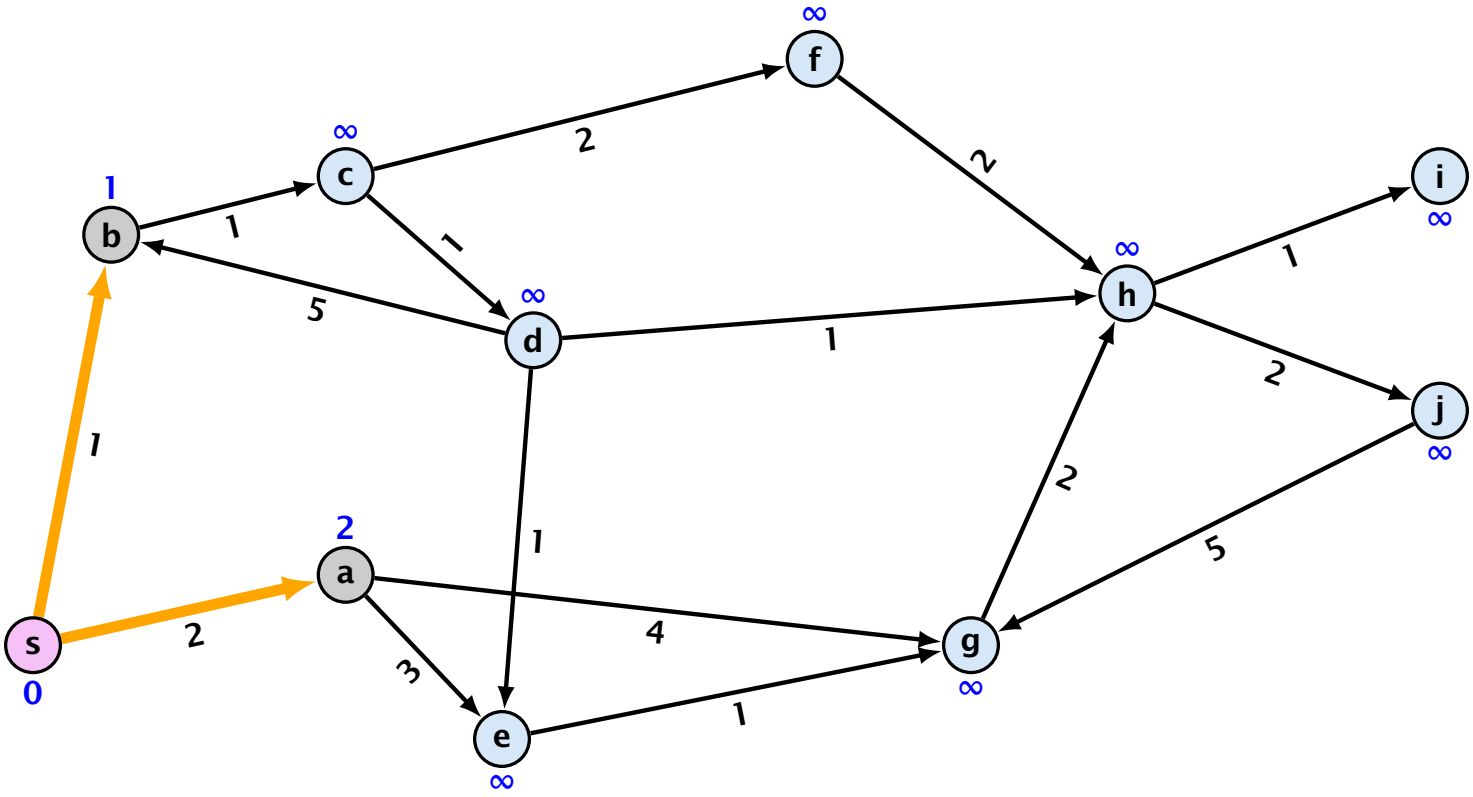
# Beliebige Graphen — nichtnegative Gewichte



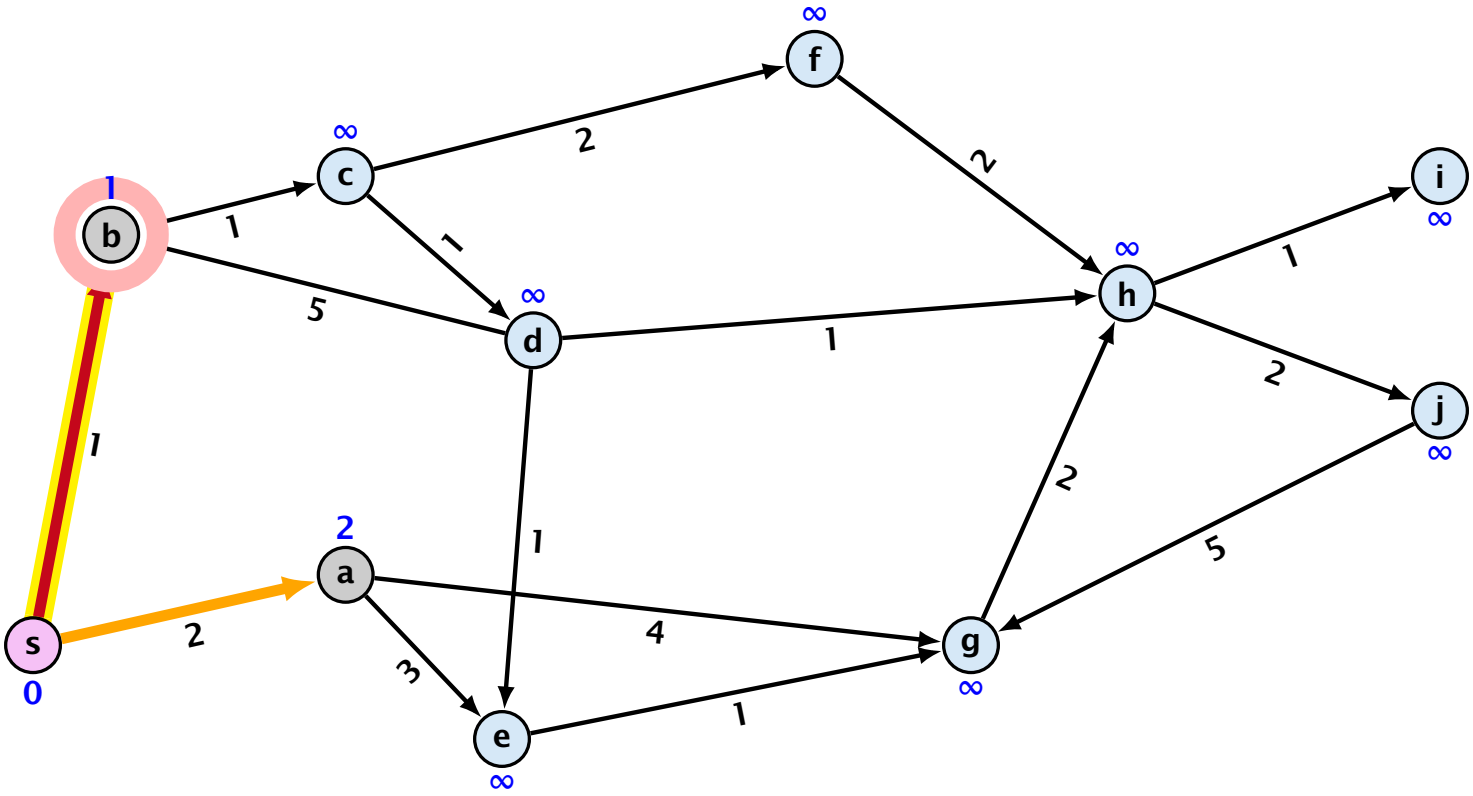
# Beliebige Graphen — nichtnegative Gewichte



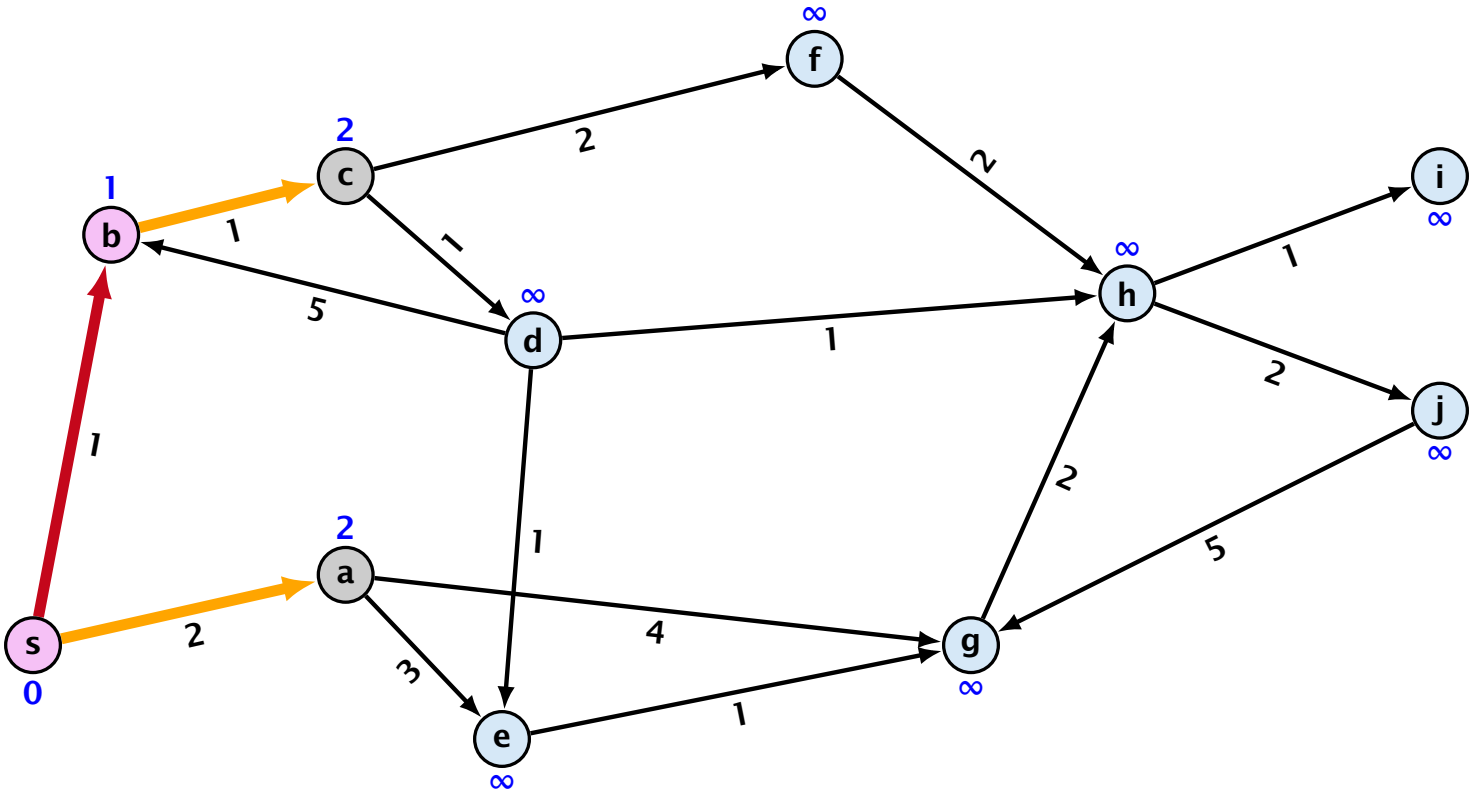
# Beliebige Graphen — nichtnegative Gewichte



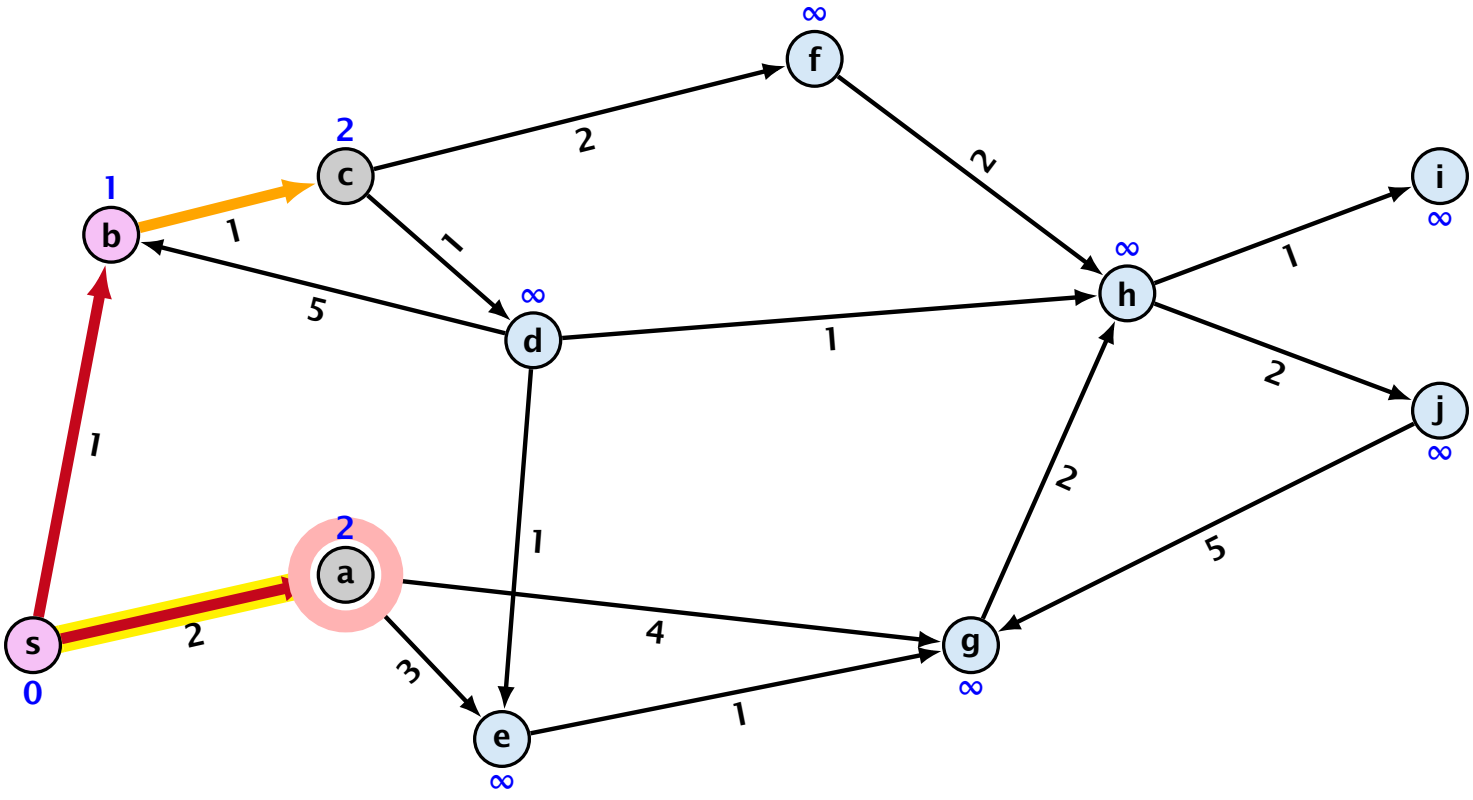
# Beliebige Graphen — nichtnegative Gewichte



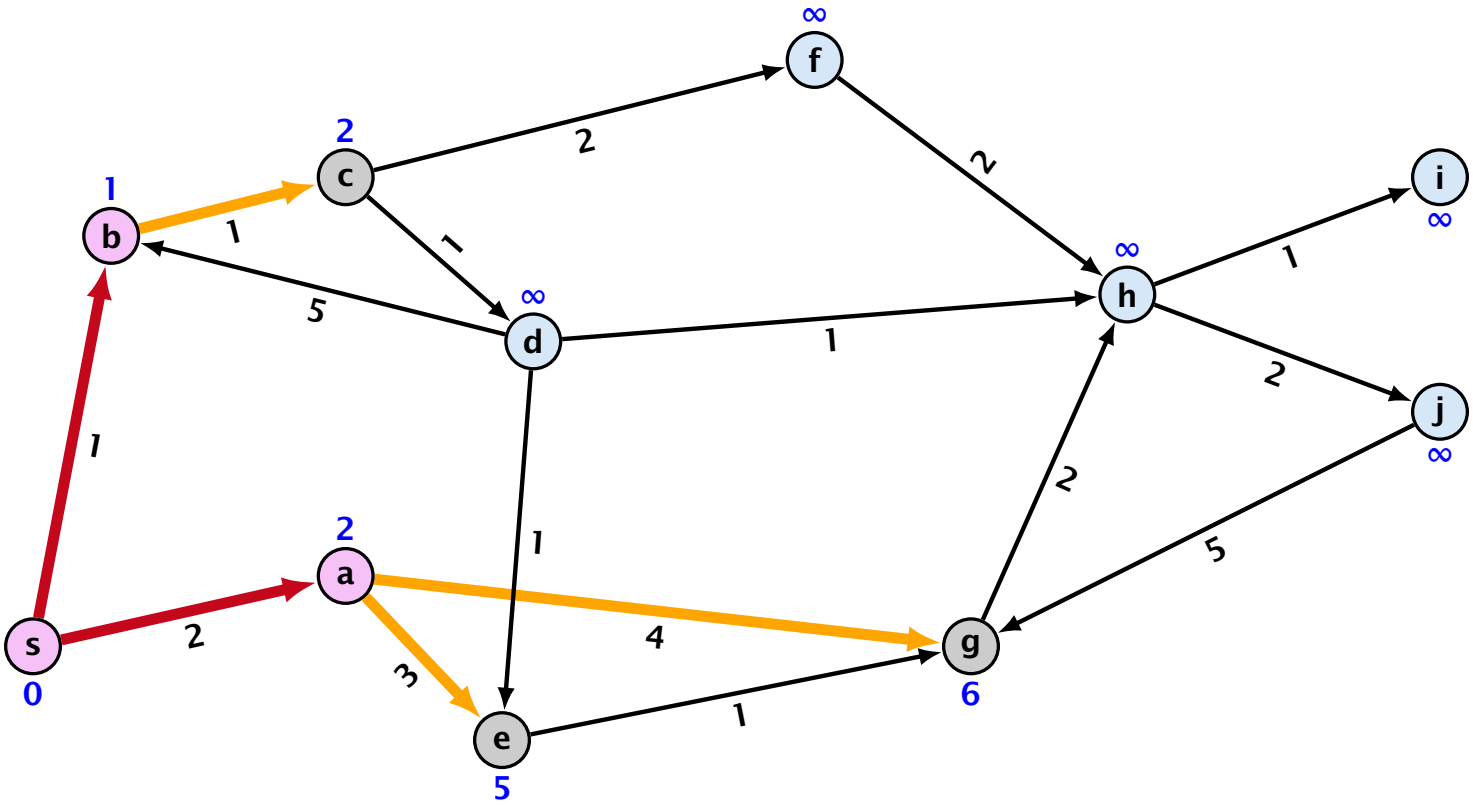
# Beliebige Graphen — nichtnegative Gewichte



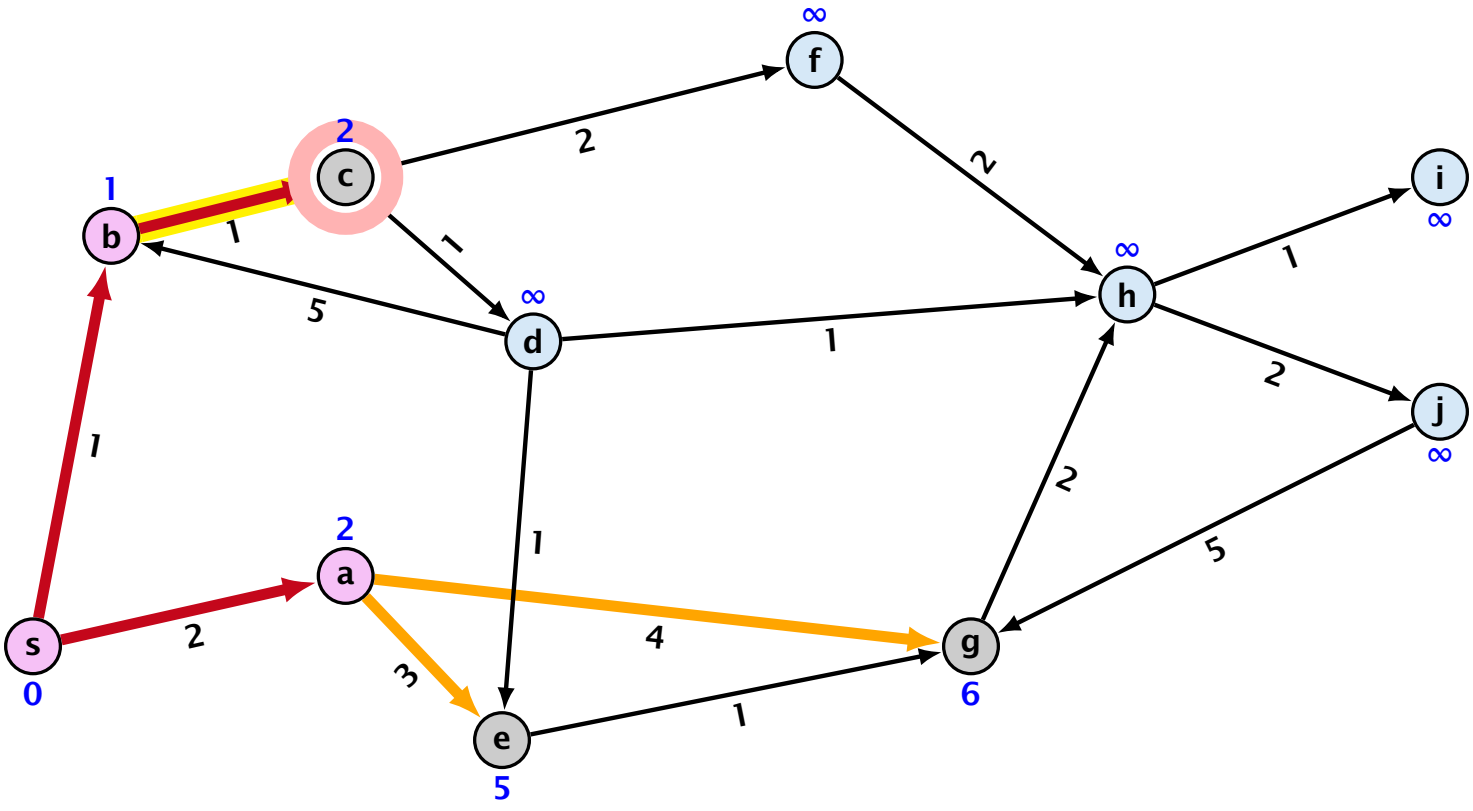
# Beliebige Graphen — nichtnegative Gewichte



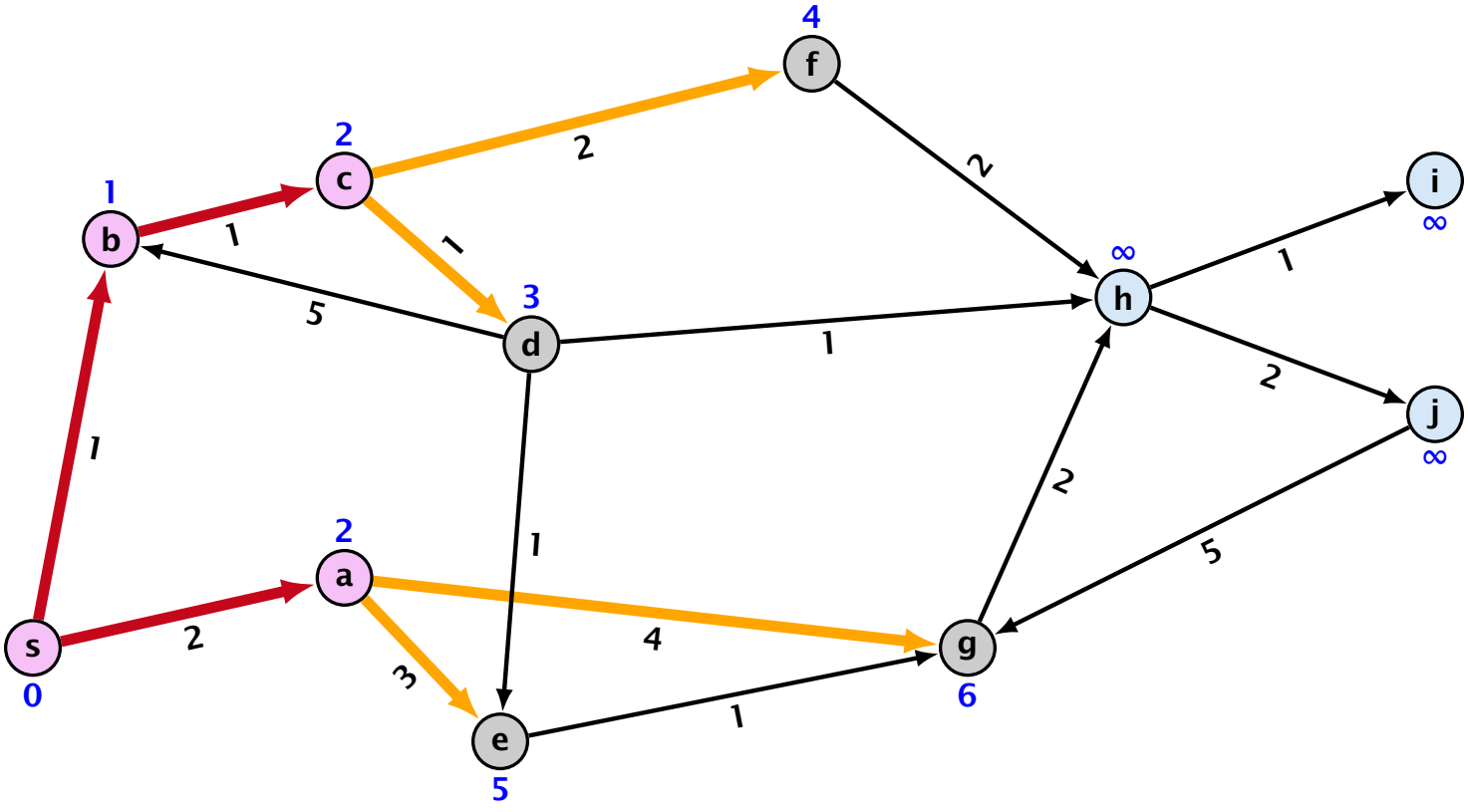
# Beliebige Graphen — nichtnegative Gewichte



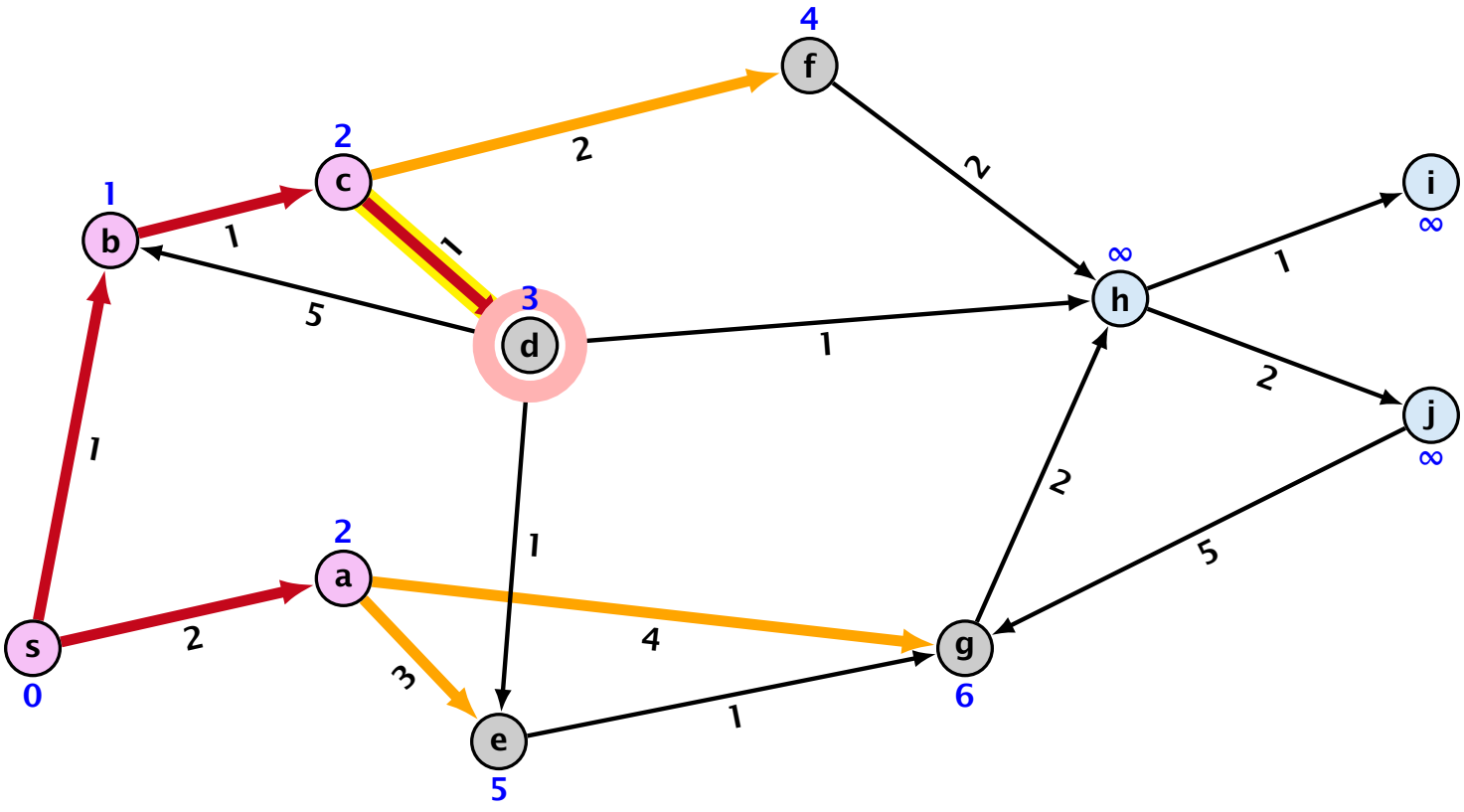
# Beliebige Graphen — nichtnegative Gewichte



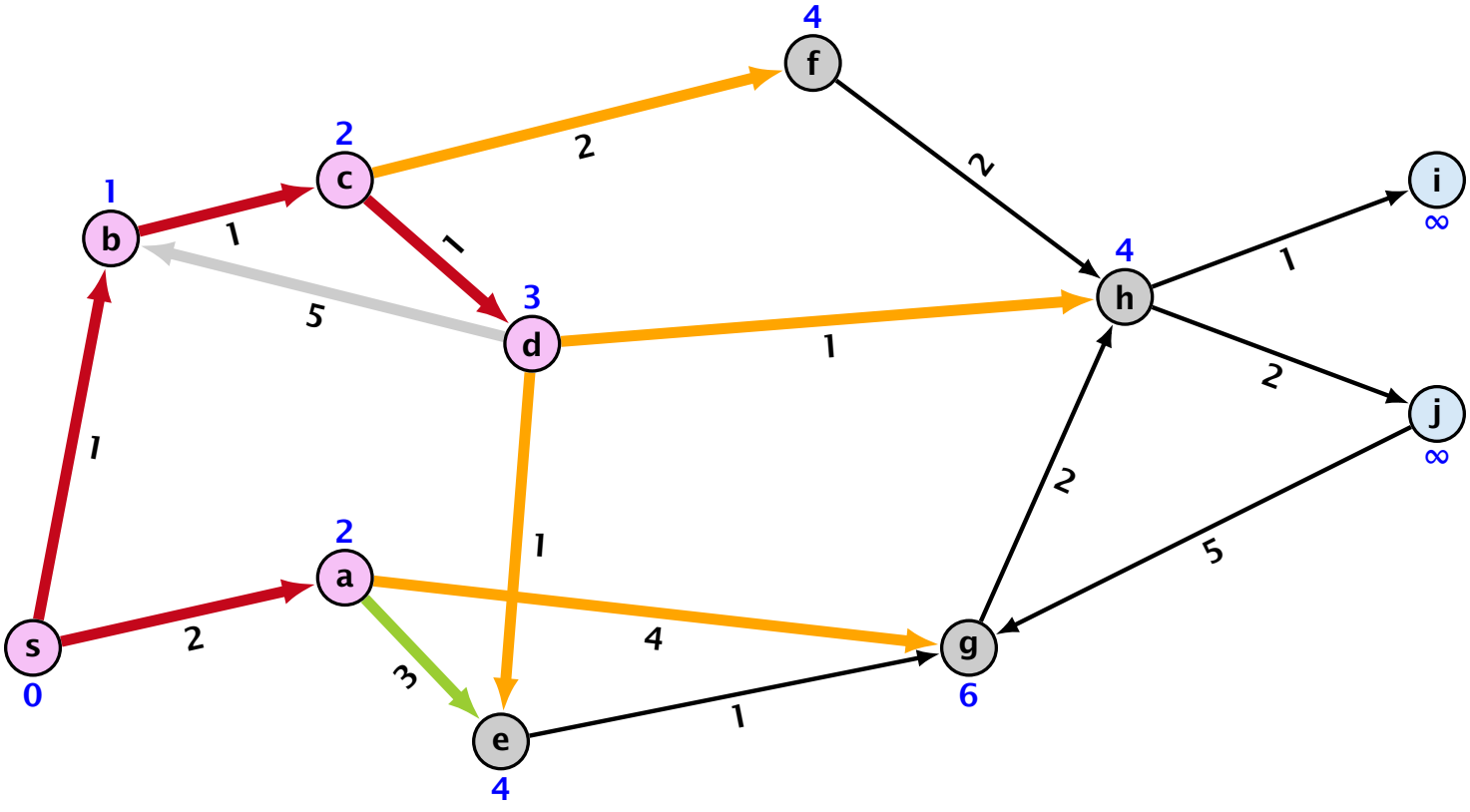
# Beliebige Graphen — nichtnegative Gewichte



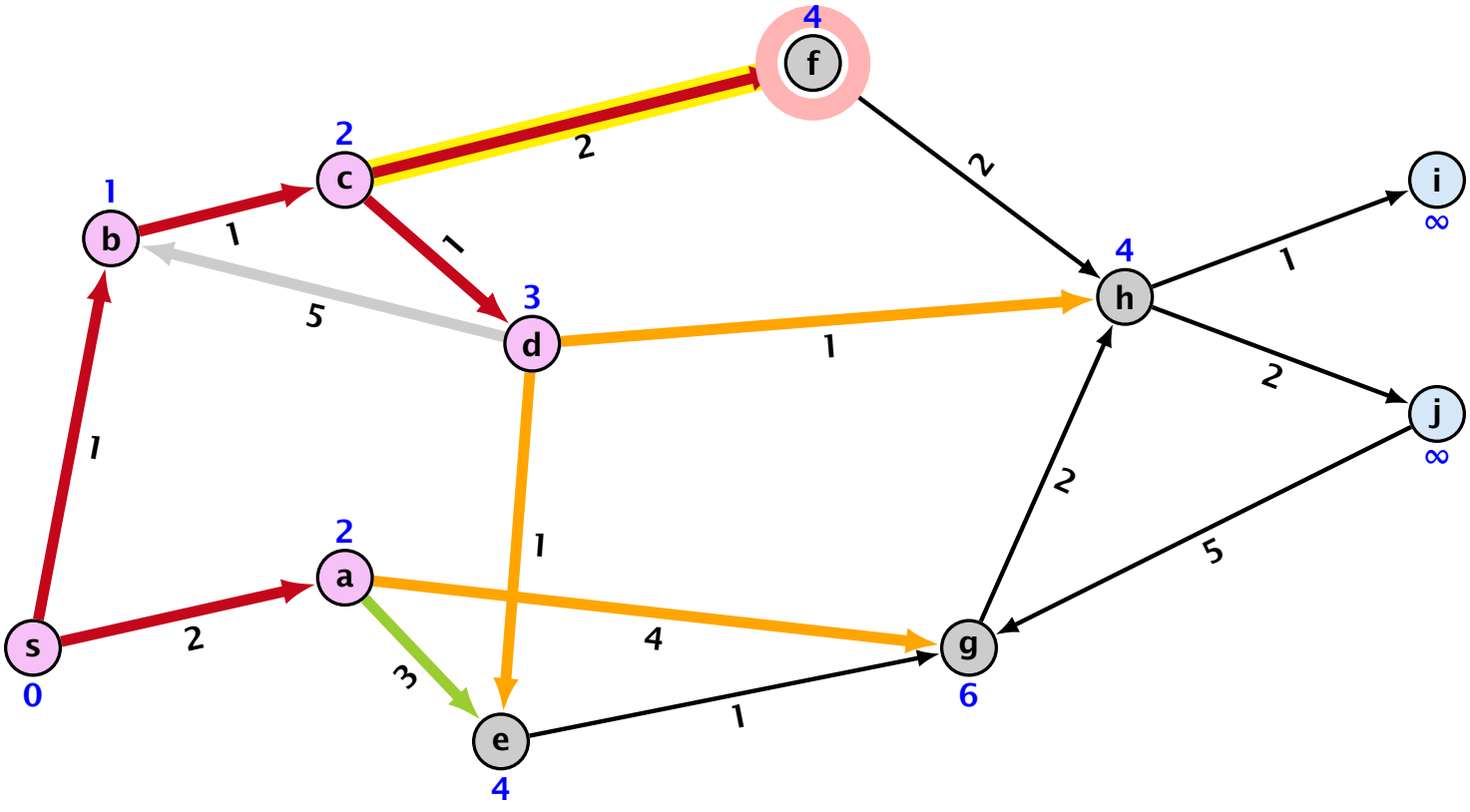
# Beliebige Graphen — nichtnegative Gewichte



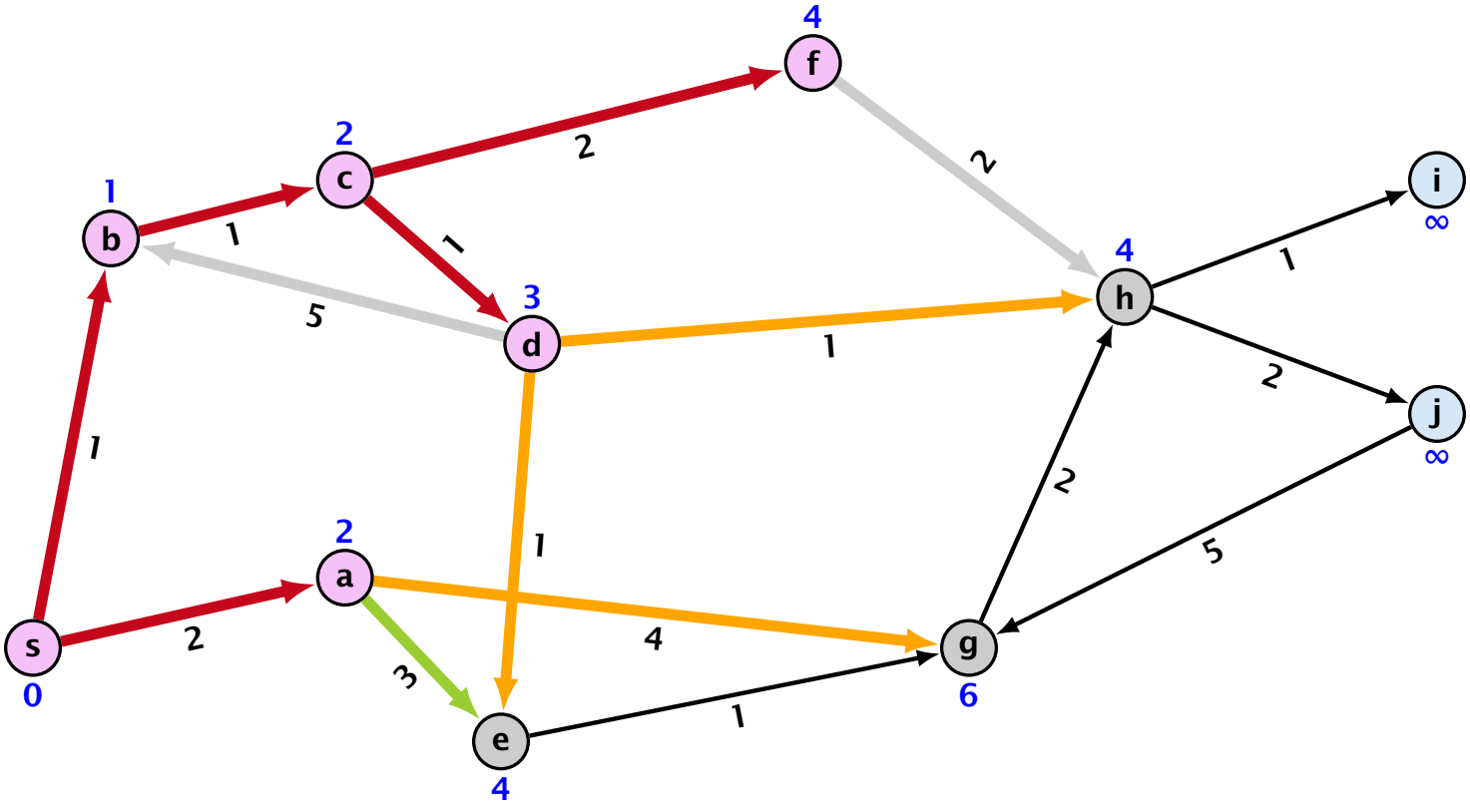
# Beliebige Graphen — nichtnegative Gewichte



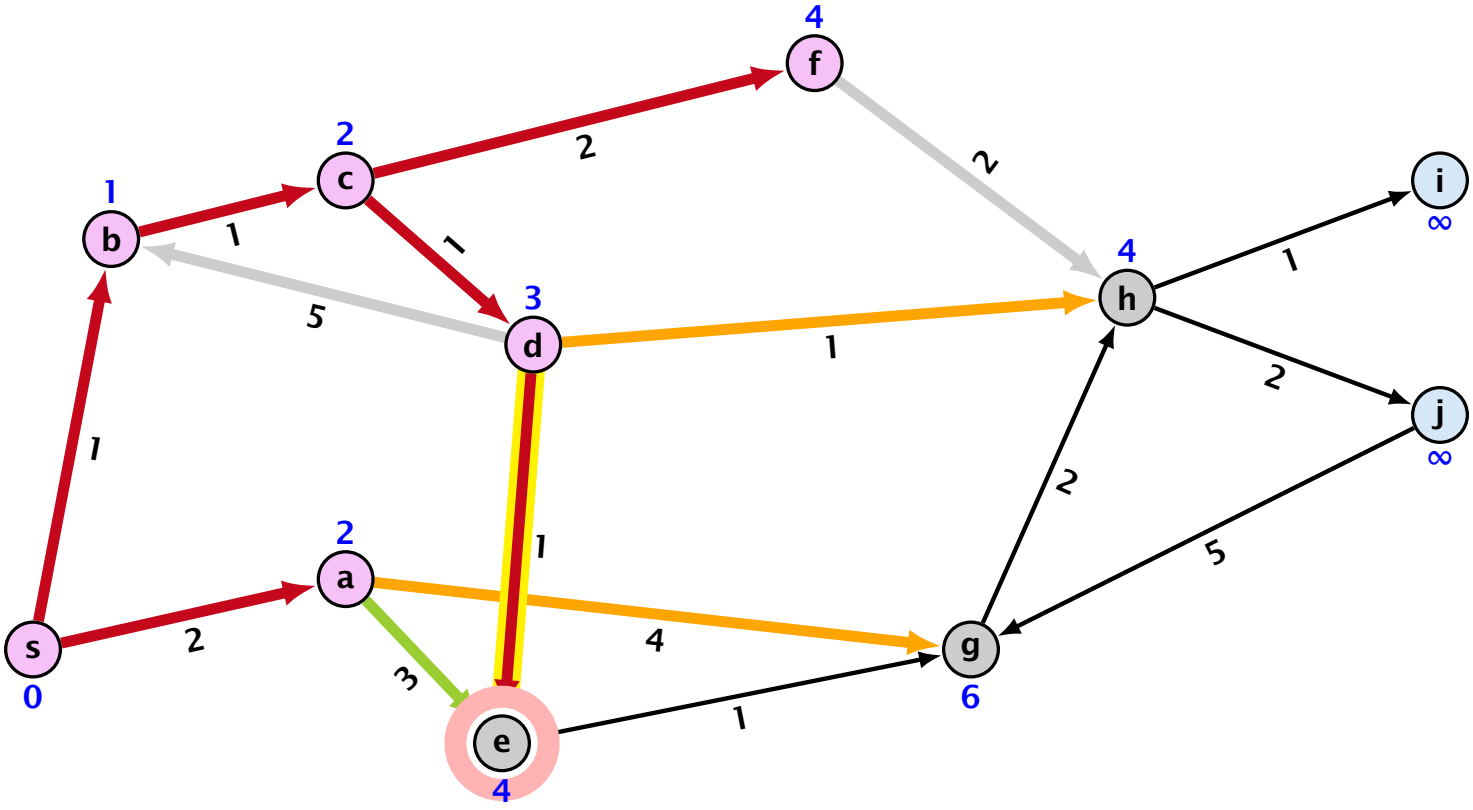
# Beliebige Graphen — nichtnegative Gewichte



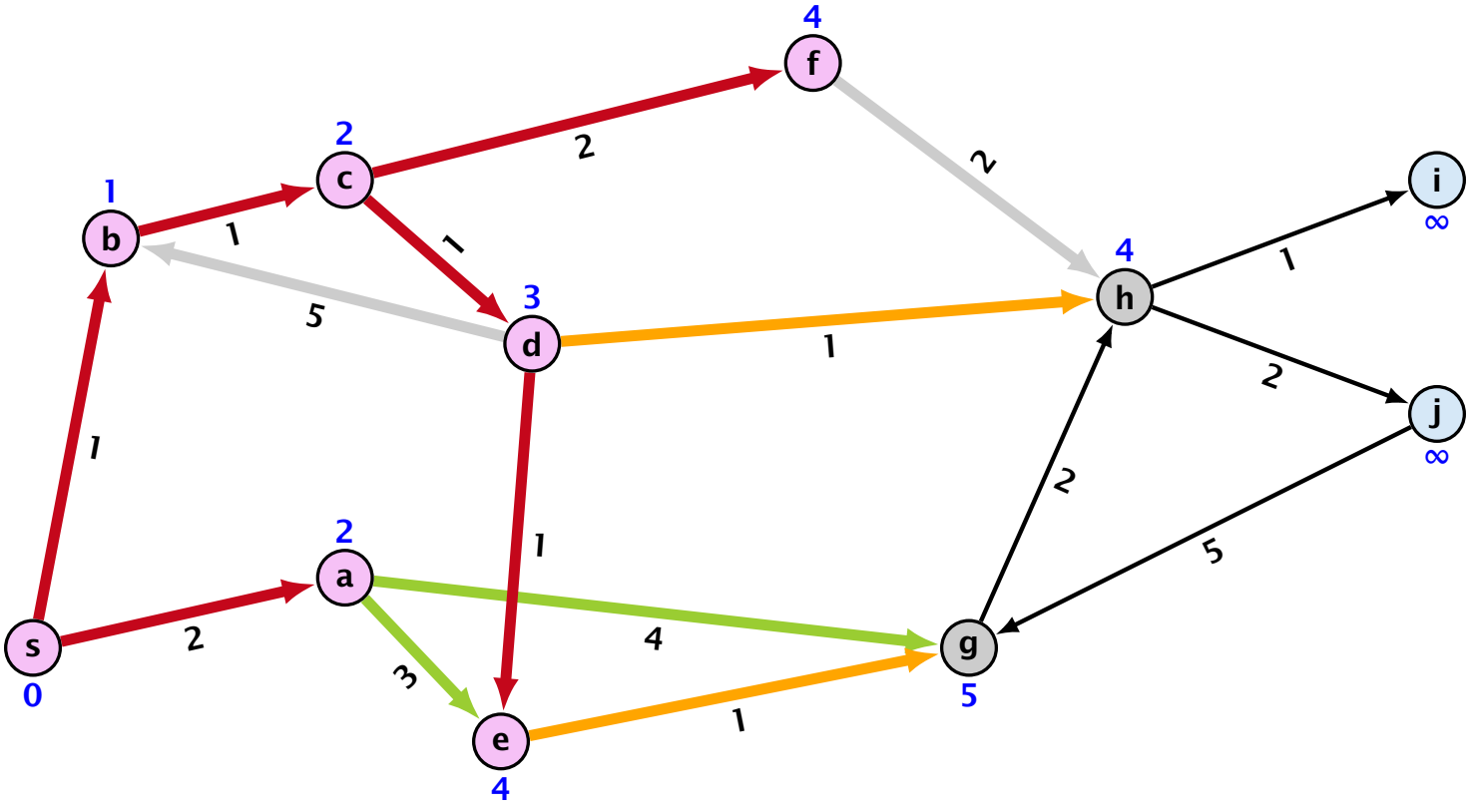
# Beliebige Graphen — nichtnegative Gewichte



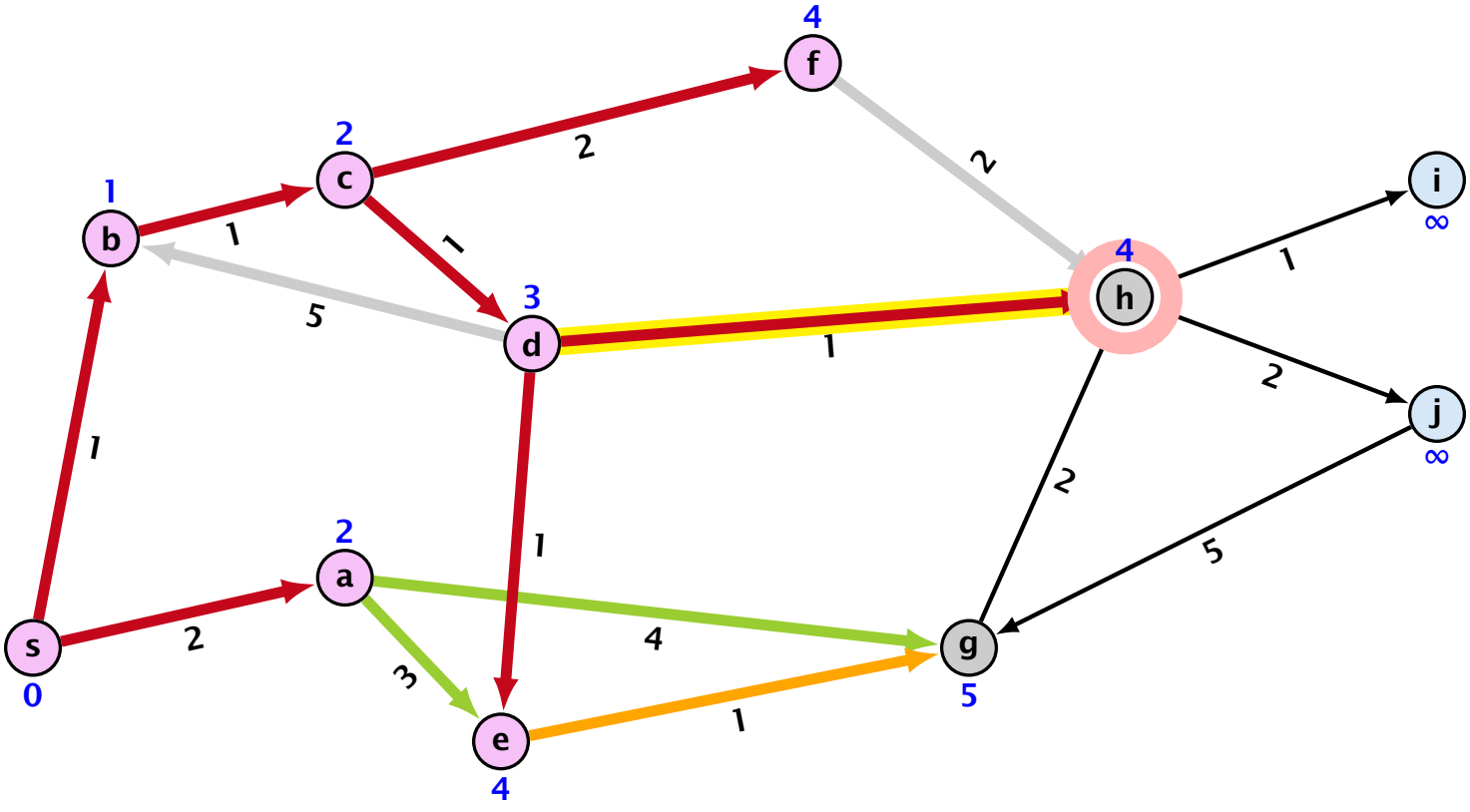
# Beliebige Graphen — nichtnegative Gewichte



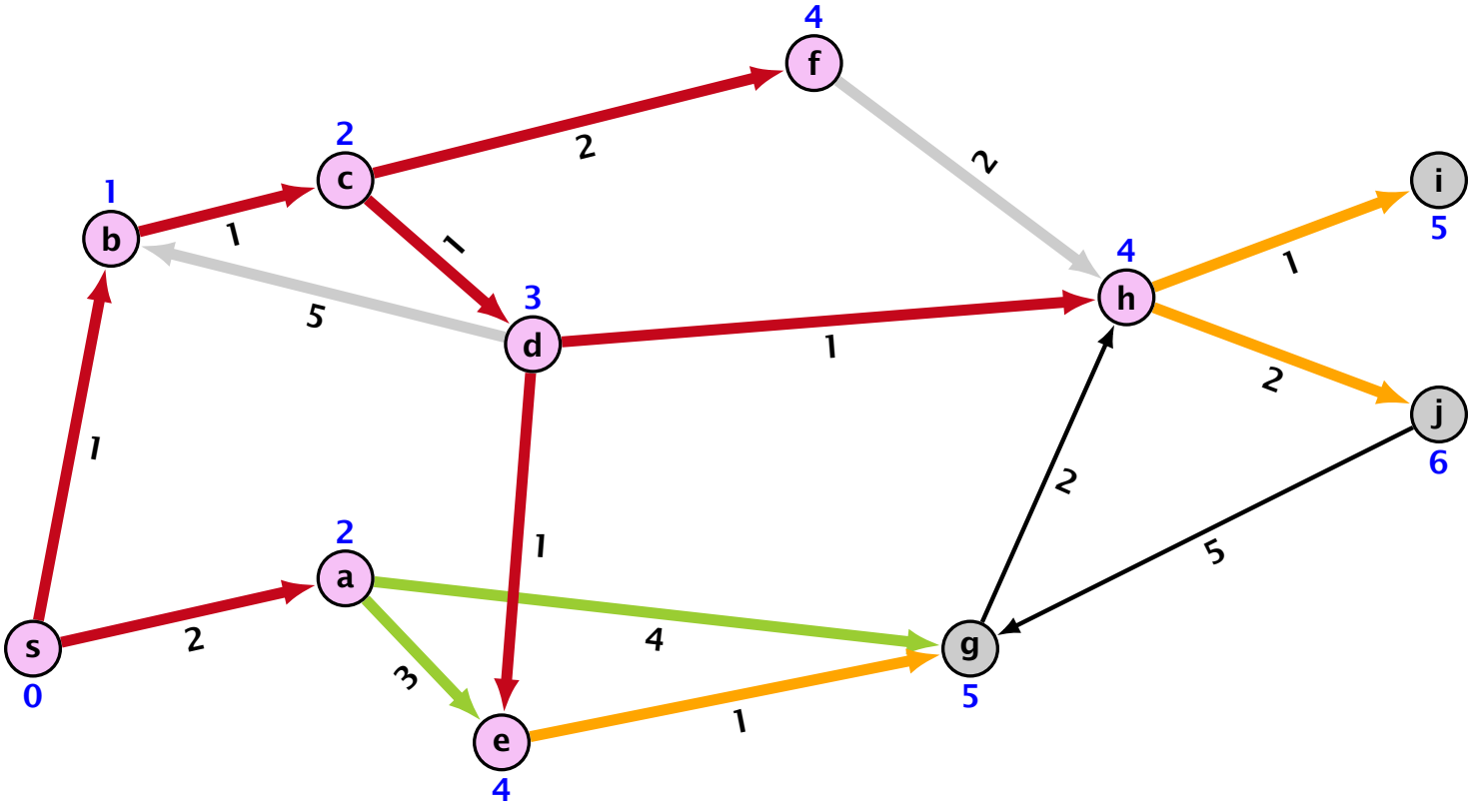
# Beliebige Graphen — nichtnegative Gewichte



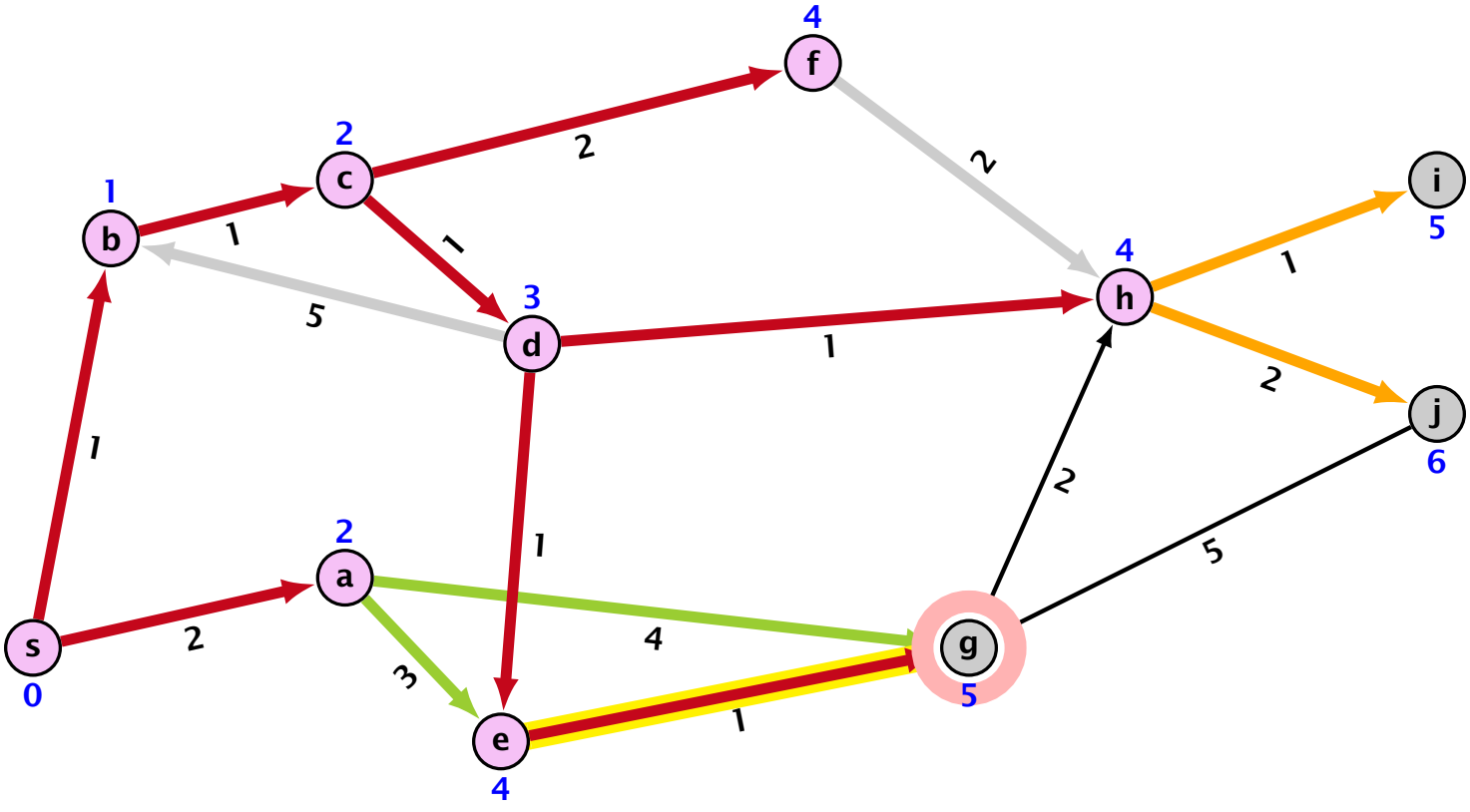
# Beliebige Graphen — nichtnegative Gewichte



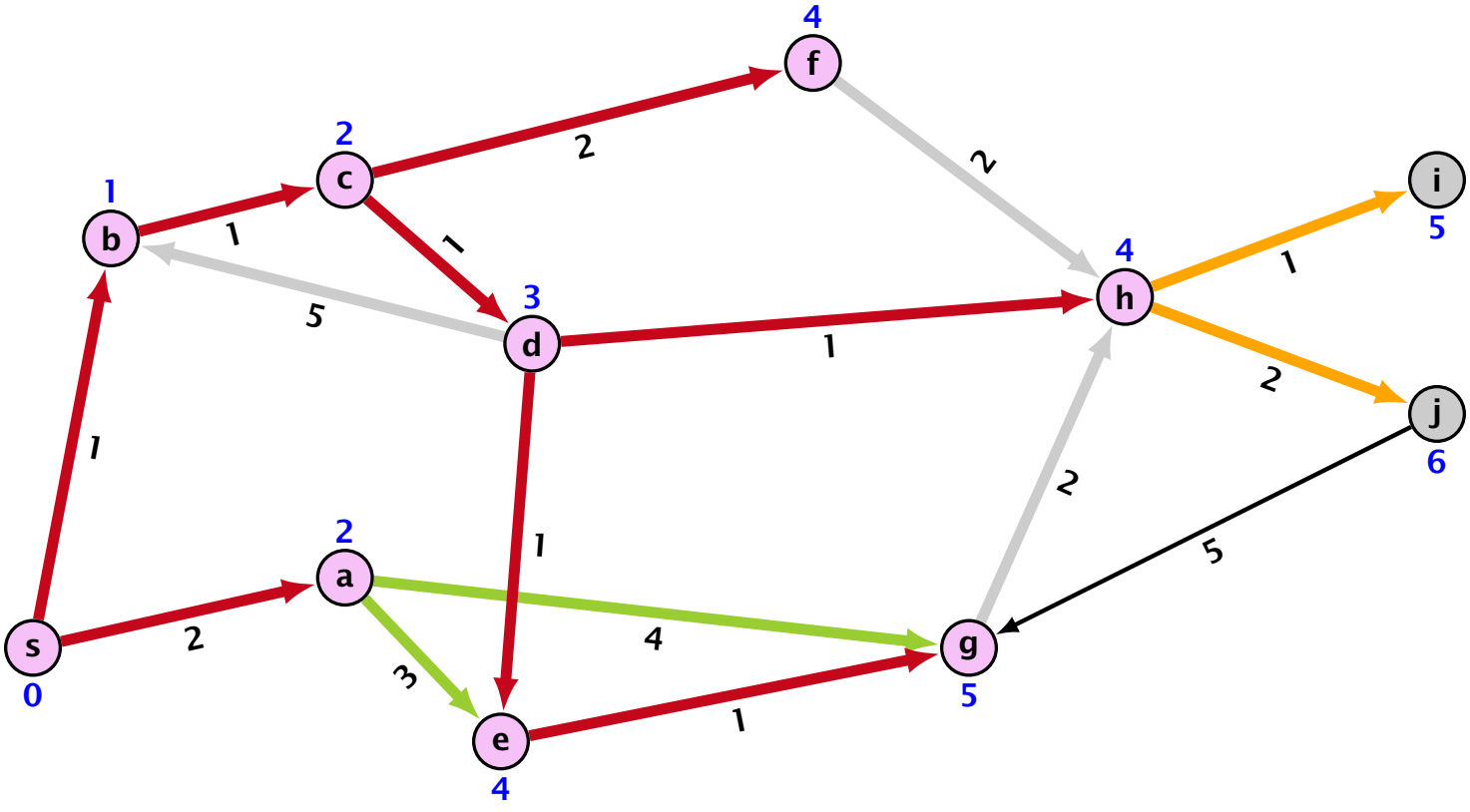
# Beliebige Graphen — nichtnegative Gewichte



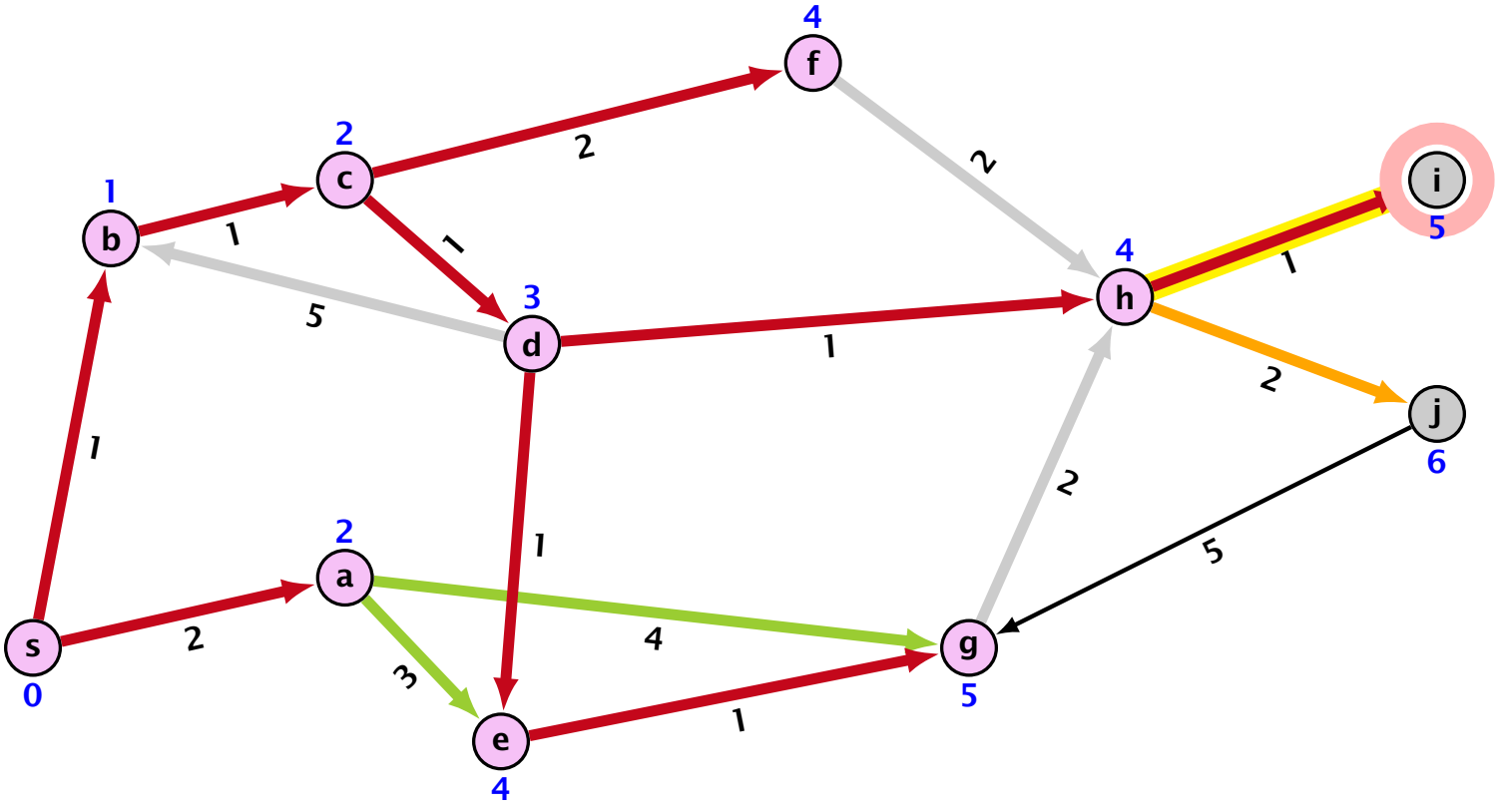
# Beliebige Graphen — nichtnegative Gewichte



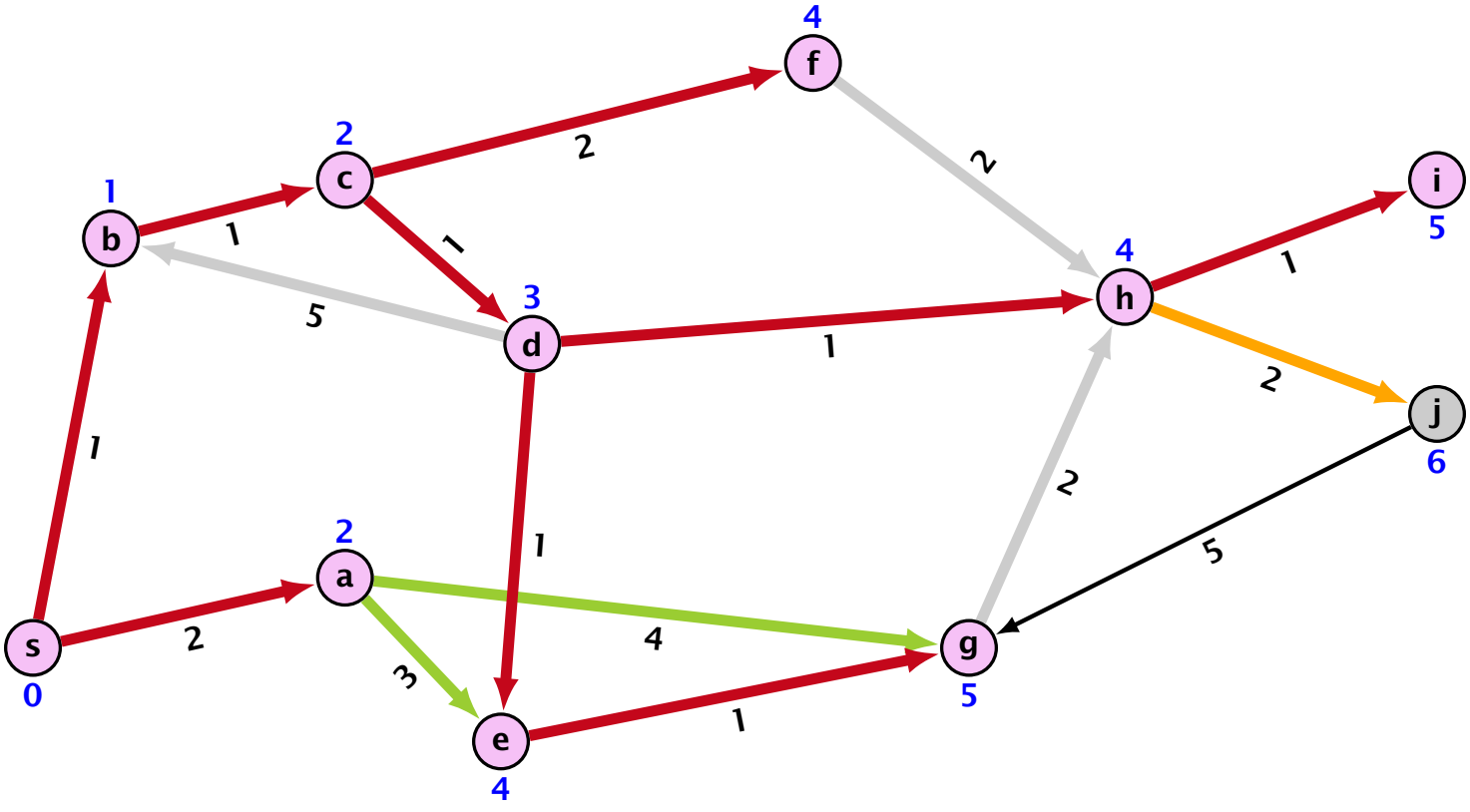
# Beliebige Graphen — nichtnegative Gewichte



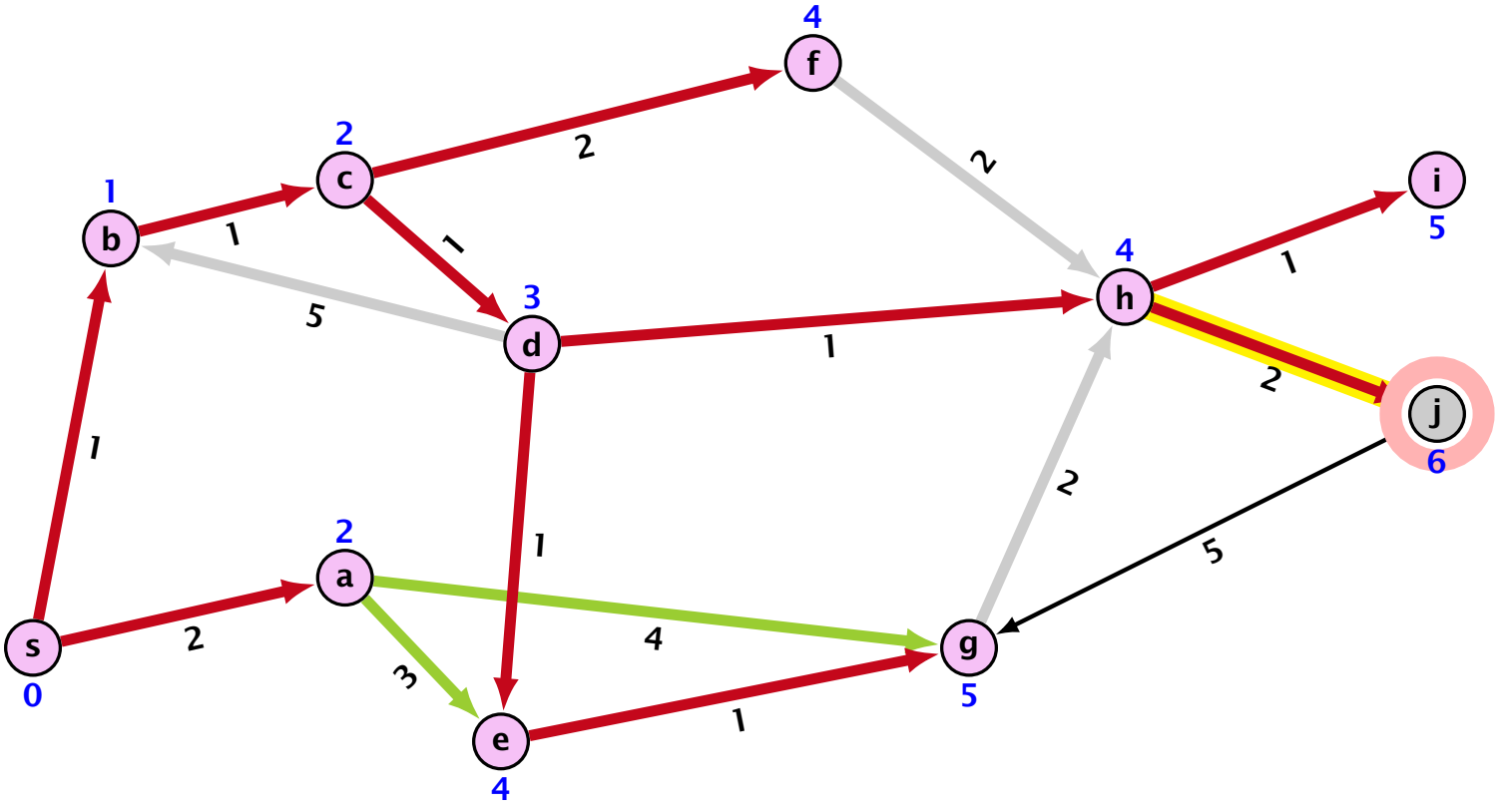
# Beliebige Graphen — nichtnegative Gewichte



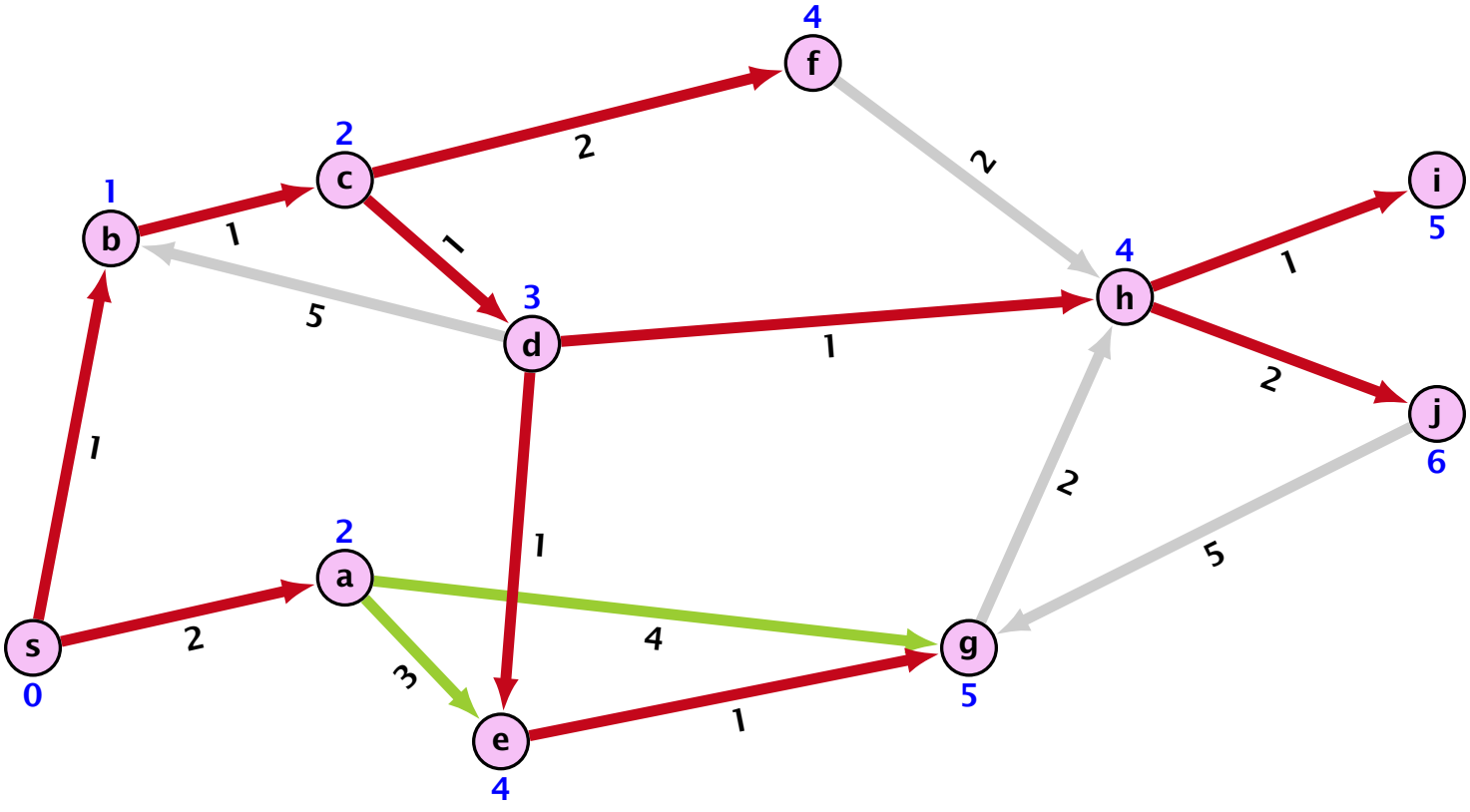
# Beliebige Graphen — nichtnegative Gewichte



# Beliebige Graphen — nichtnegative Gewichte



# Beliebige Graphen — nichtnegative Gewichte



# Warum funktioniert das?

**Invariante:**
Sei $A$ die Menge der Knoten, die schon aus der PQ entfernt wurden.

- **A** Das Distanzlabel $x.\text{key}$ erfüllt $x.\text{key} \leq w(P)$ für alle $s$-$x$ Pfade $P$ die **entweder** mit Kante $(a,x)$, $a \in A$ enden **oder** leer sind (nur aus Knoten $s$ bestehen).
- **B** Knoten in $A$ haben ihre korrekte Distanz.

Außerdem nutzen wir, dass $x.\text{key}$ eine obere Schranke an die Länge eines kürzesten $s$-$x$ Pfades ist (folgt da wir nur Relaxierungen durchführen).

**Initialisierung:**

A Initial ist $A$ leer; d.h. $s$ ist der einzig erlaubte Pfad. Deshalb ist Invariante erfüllt wenn man $s$ Distanz $0$ gibt and allen anderen Knoten Distanz $\infty$.

B Gilt, da $A$ leer ist.

**Initialisierung:**

**A** Initial ist $A$ leer; d.h. $s$ ist der einzig erlaubte Pfad. Deshalb ist Invariante erfüllt wenn man $s$ Distanz $0$ gibt and allen anderen Knoten Distanz $\infty$.

**B** Gilt, da $A$ leer ist.

**Beibehaltung der Invariante:**

**B** Wir nehmen den Knoten $v$ mit kleinstem key-Wert. Ein Pfad $P$ von $s$ nach $v$ muss Länge mindestens $v.\mathrm{key}$ haben.

Dies gilt, da der Pfad wenn er $A$ verläßt (zum Knoten $x$) schon Länge mindestens $x.\mathrm{key} \geq v.\mathrm{key}$ hat (durch Invariante A für $x.\mathrm{key}$). Er kann nicht kürzer werden da Kantengewichte nichtnegativ sind.

Deshalb, ist das Distanzlabel für $v$ korrekt (es ist eine obere Schranke und kein Pfad ist kürzer)

D.h. Invariante B gilt für $A' = A \cup \{v\}$.

**A** Da $v$ korrekte Distanz hat gilt $x.\text{key} \leq w(P)$ für alle Pfade die in Kante $(v,x)$ enden.

Da durch die Invariante gilt, dass $x.\text{key} \leq w(P)$ für Pfade, die in $(a,x)$, $a \in A$ enden (oder leer sind), gilt Invariante A für $A' = A \cup \{v\}$.

**Laufzeit:** $m$ decreaseKey Operationen, $n$ Einfügeoperationen, $n$ extractMin() Operationen

Laufzeit: $\mathcal{O}((m+n)\log n)$.

Eine Laufzeit von $\mathcal{O}(m + n\log n)$ kann man durch Implementierung der Prioritätswarteschlange mit Fibonacciheaps erreichen.

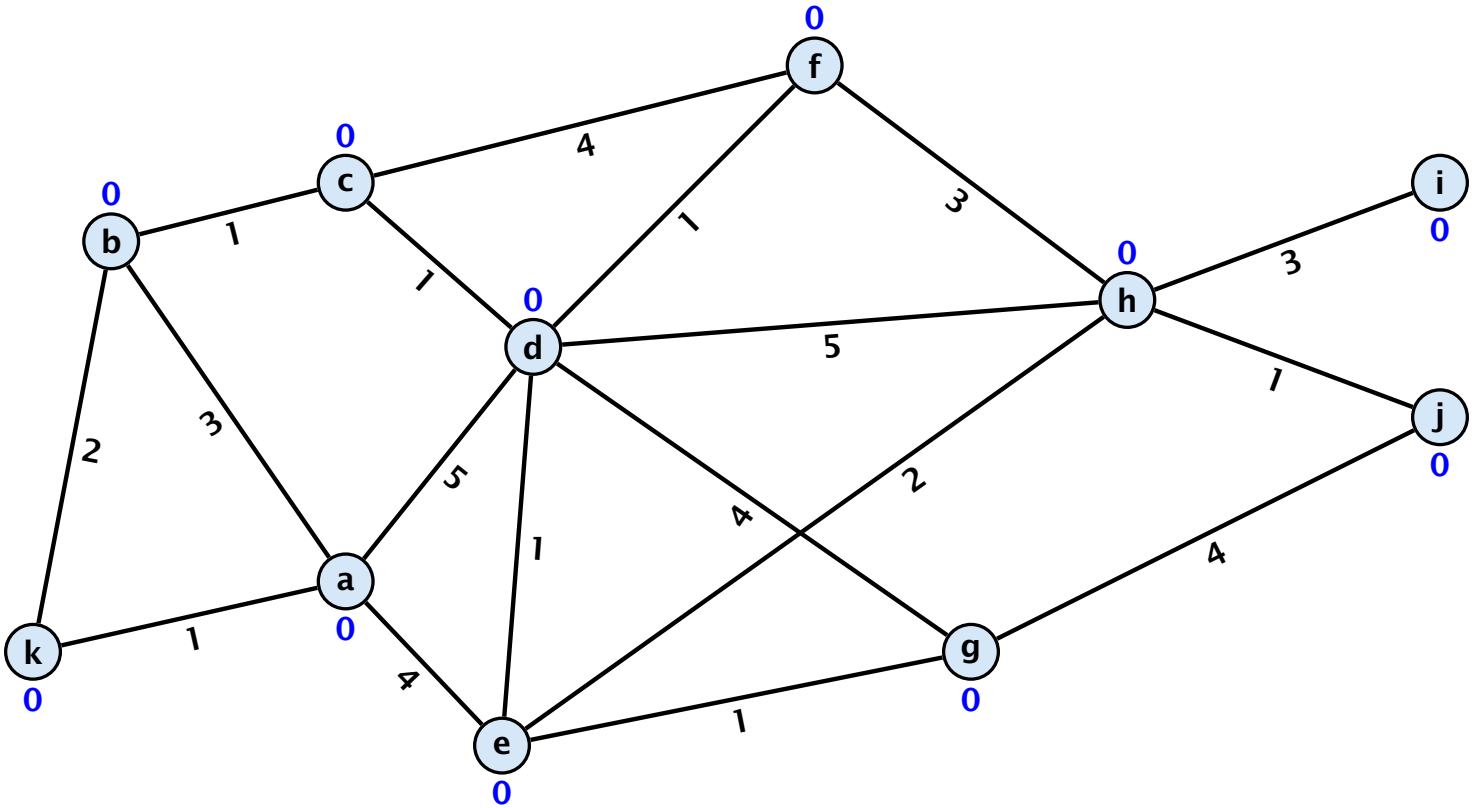
**Laufzeit:** $m$ decreaseKey Operationen, $n$ Einfügeoperationen, $n$ extractMin() Operationen

Laufzeit: $\mathcal{O}((m+n)\log n)$.

Eine Laufzeit von $\mathcal{O}(m + n\log n)$ kann man durch Implementierung der Prioritätswarteschlange mit Fibonacciheaps erreichen.

# Minimaler Spannbaum

Welche Kanten sollte man wählen um alle Knoten zu verbinden?




Minimiere Kosten!!!

# Minimaler Spannbaum

**Eingabe:**

- ungerichteter zusammenhängender graph $G = (V, E)$
- positive Kantengewichte (Kosten) $w : E \to \mathbb{R}^+$

**Ausgabe:**

- Teilmenge $T \subseteq E$ s.t. $G = (V, T)$ verbunden und $w(T) = \sum_{e \in T} w(e)$ minimal

**Beobachtung:**

- da Kantengewichte positiv ist $T$ ein Baum

# Minimaler Spannbaum

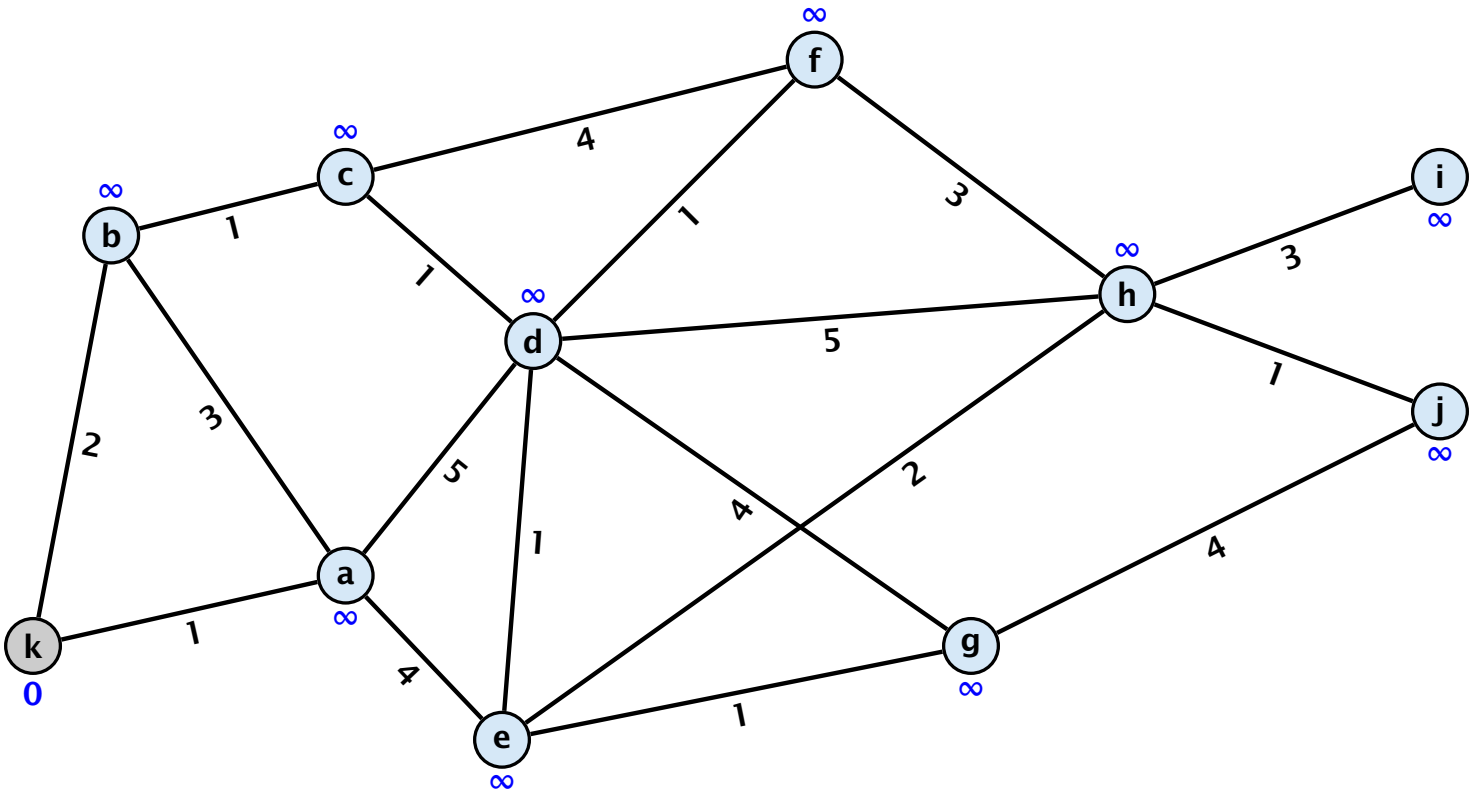
**Lemma 6**

*Sei $X, Y$ eine Partionierung von $V$ (d.h., $X \cup Y = V$ and $X \cap V = \varnothing$), und sei $e = (x, y)$ eine billigste Kante zwischen $X$ und $Y$ Dann existiert ein Minimaler Spannbaum der $e$ enthält.*

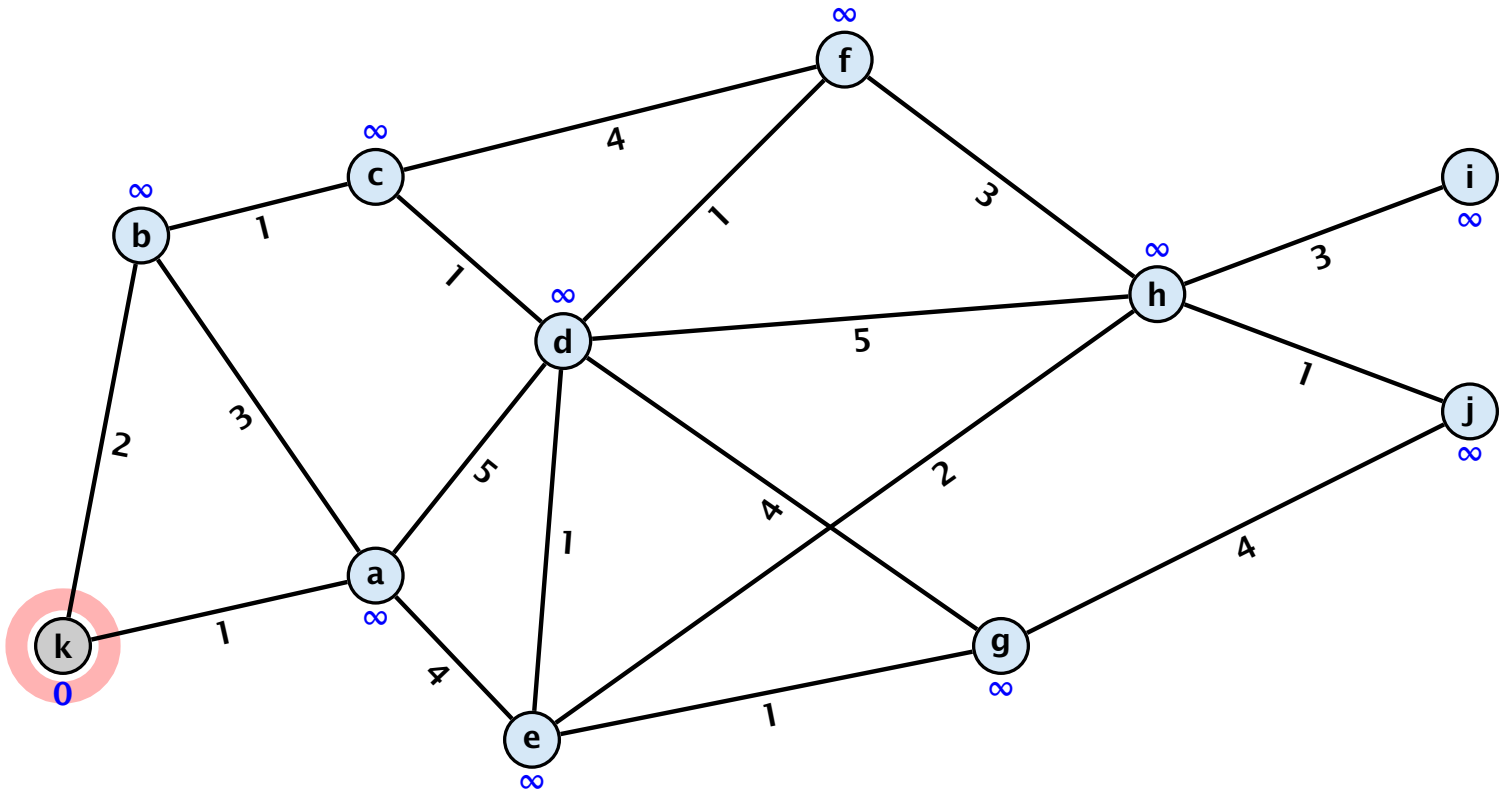
# Prims MST-Algorithmus

```
1  Input: weighted graph G=(V,E,w); start vertex s;
2  Output: parent-fields of nodes encode MST;
3
4  PrimMST(s)
5      S = new PriorityQueue()
6      foreach v ∈ V
7          v->key = ∞;
8          v->par = NULL;
9          v->han = S->insert(v);
10     s->key = 0; S->decreaseKey(s->han,0);
11     while (!S->empty()) {
12         v = S->extractMin();
13         foreach x ∈ N[v]
14             if (x->key > w(v,x))
15                 S->decreaseKey(x->han,w(v,x));
16                 x->key = w(v,x);
17                 x->par = v;
18     }
```

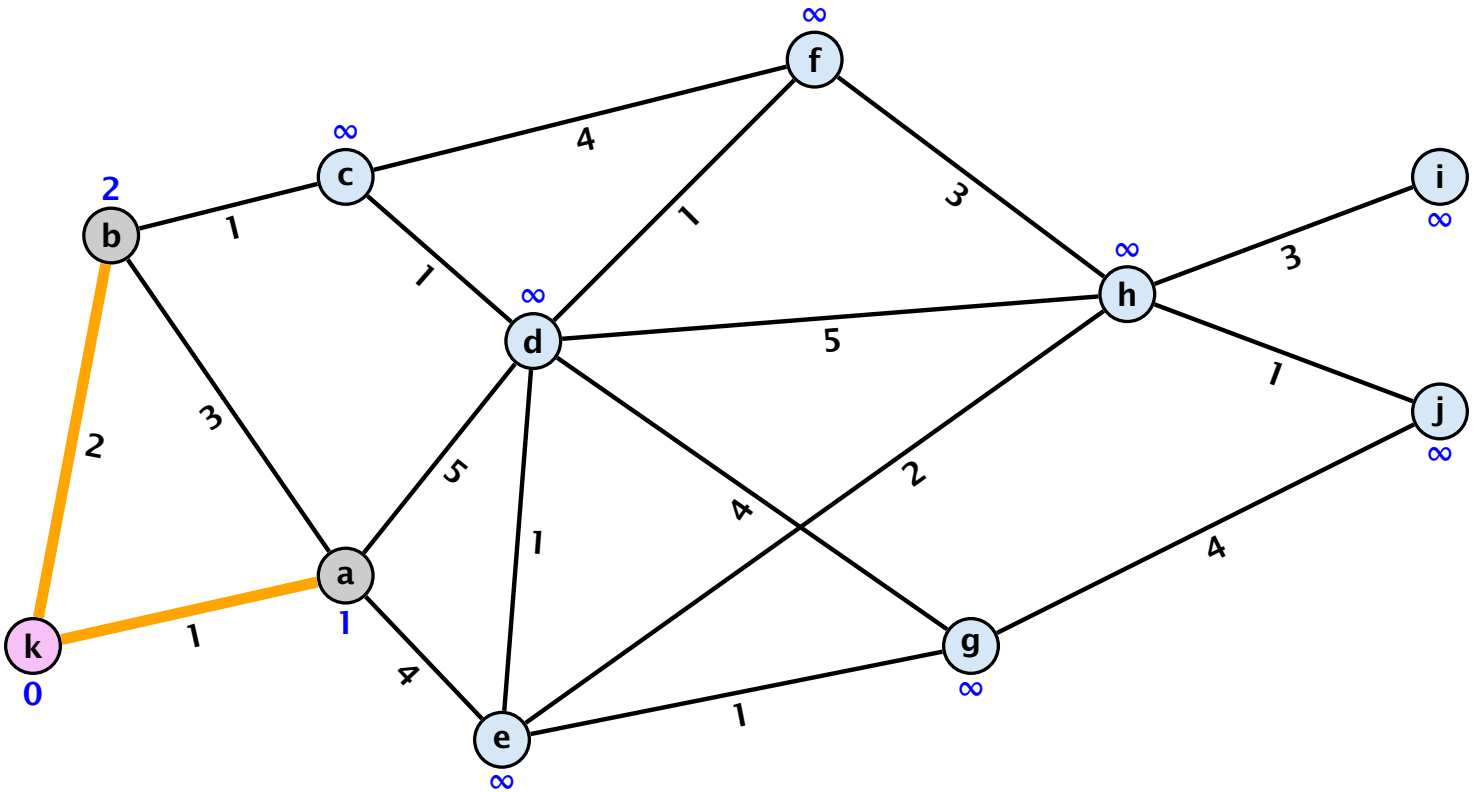
# Minimaler Spannbaum (Prim)



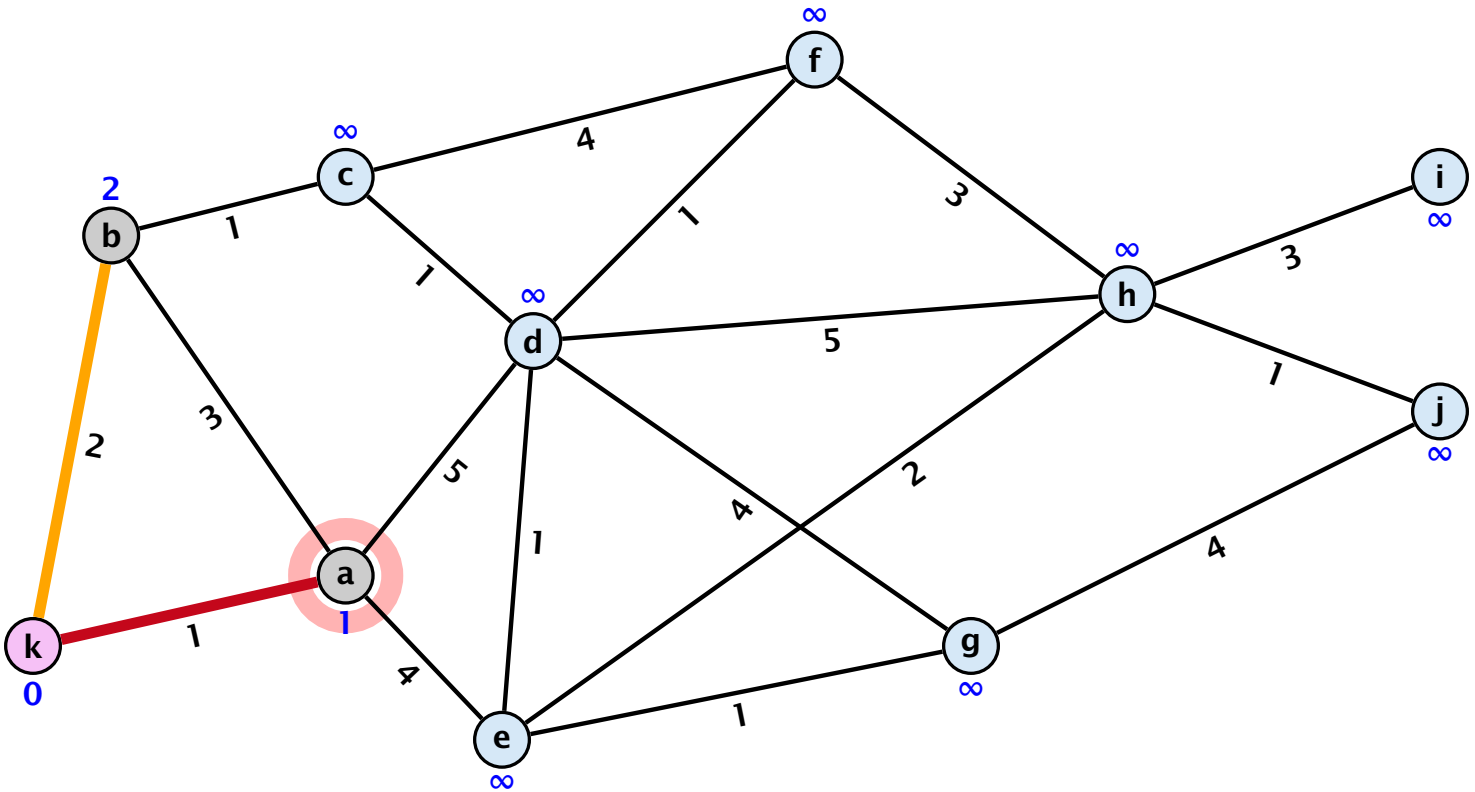
# Minimaler Spannbaum (Prim)



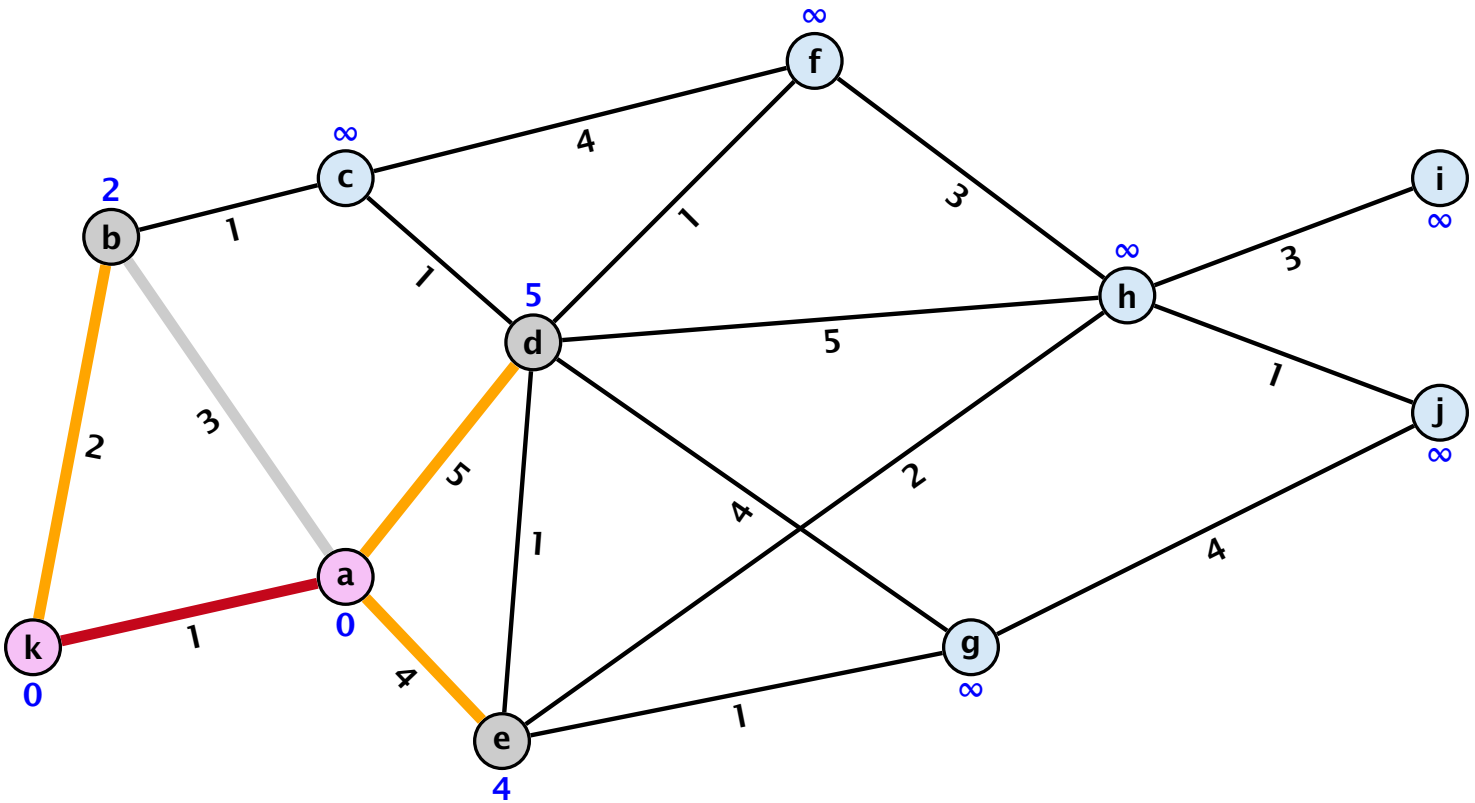
# Minimaler Spannbaum (Prim)



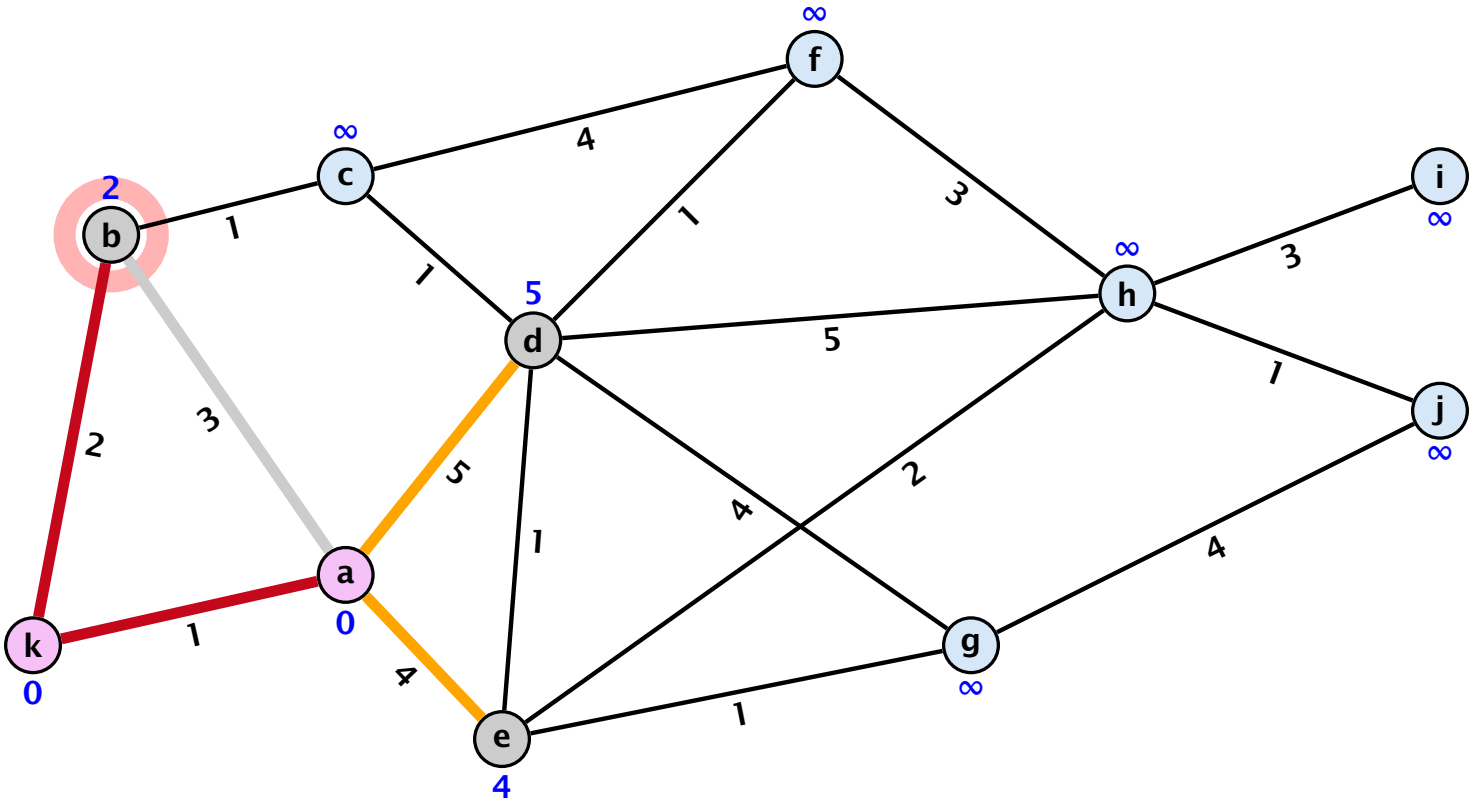
# Minimaler Spannbaum (Prim)



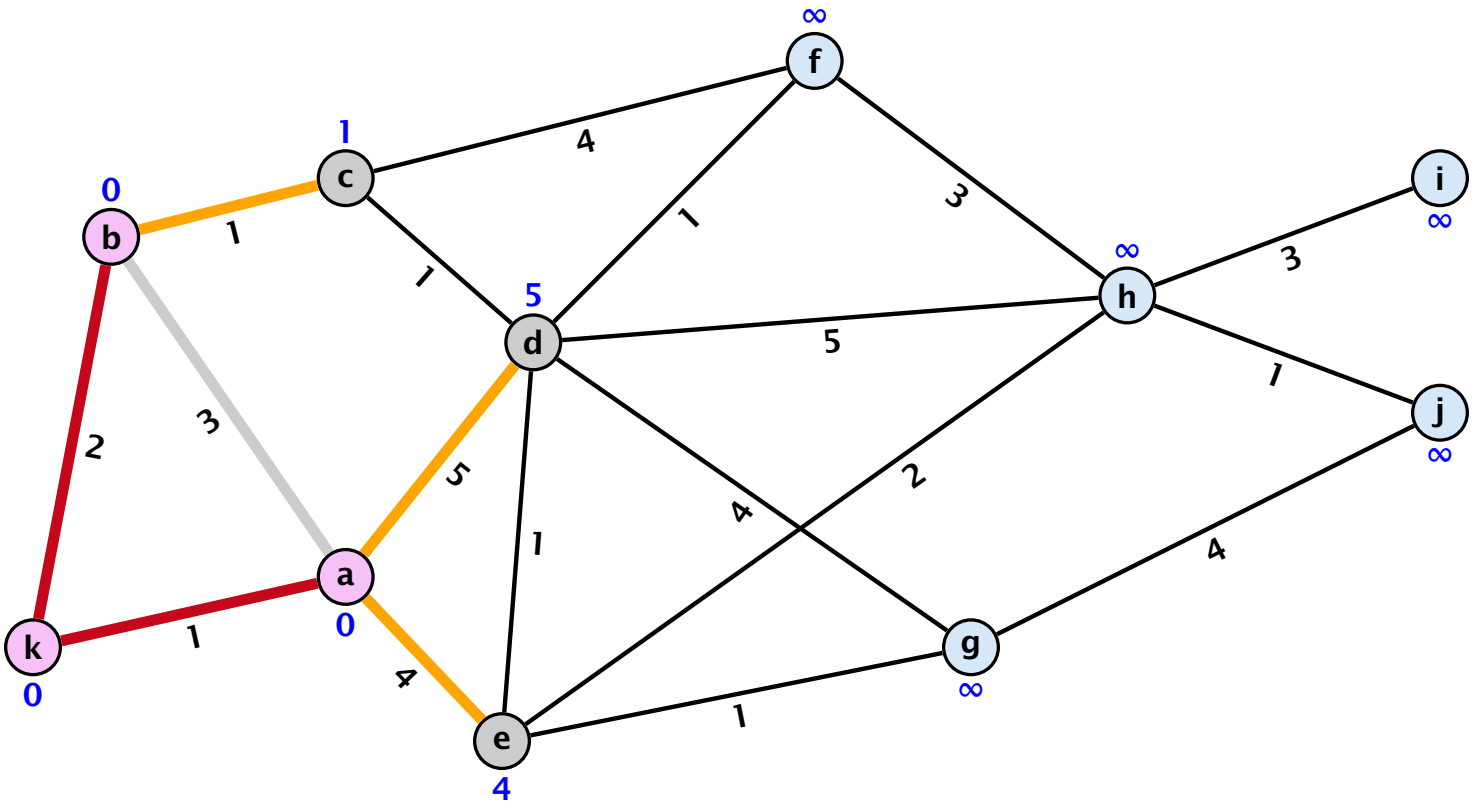
# Minimaler Spannbaum (Prim)



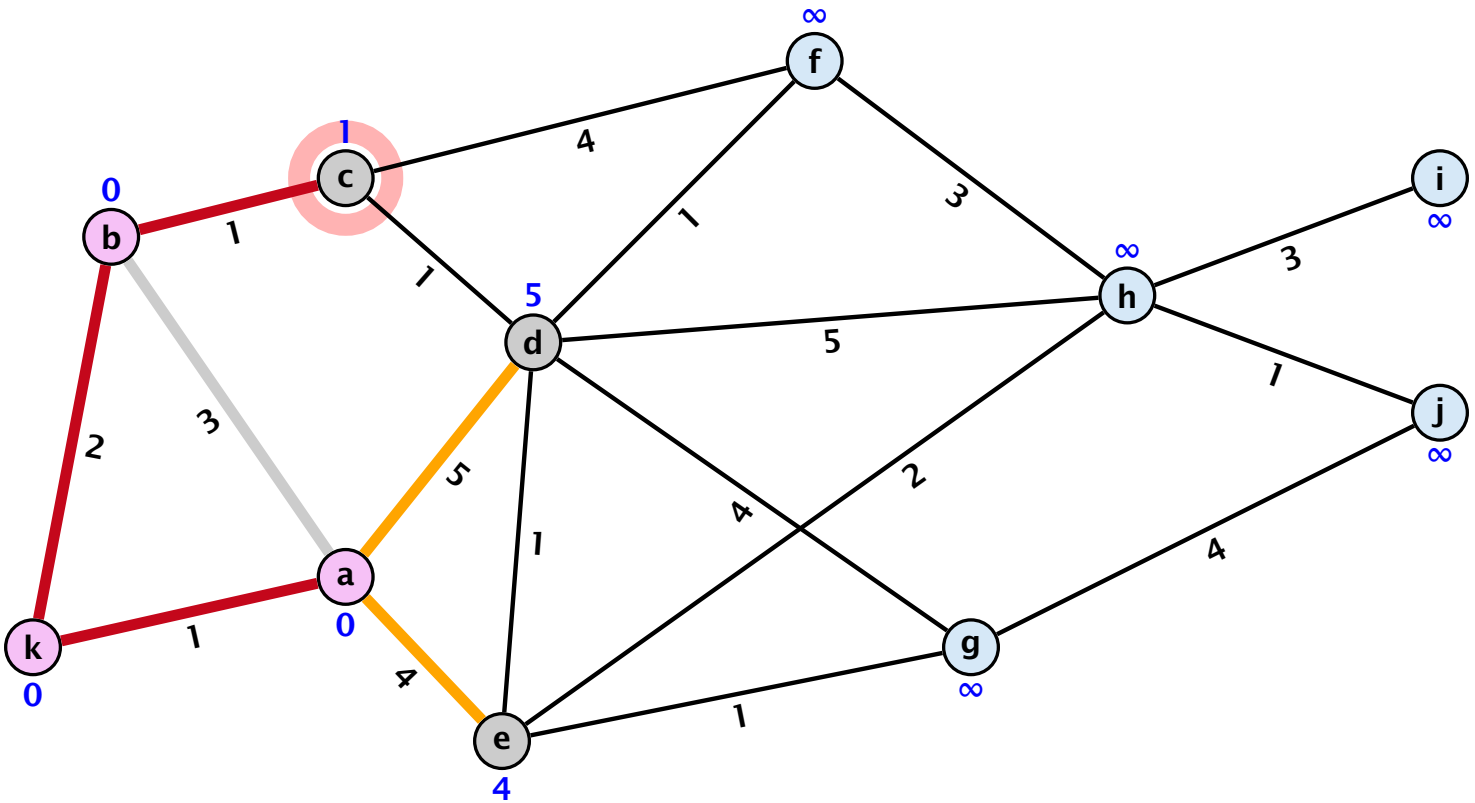
# Minimaler Spannbaum (Prim)



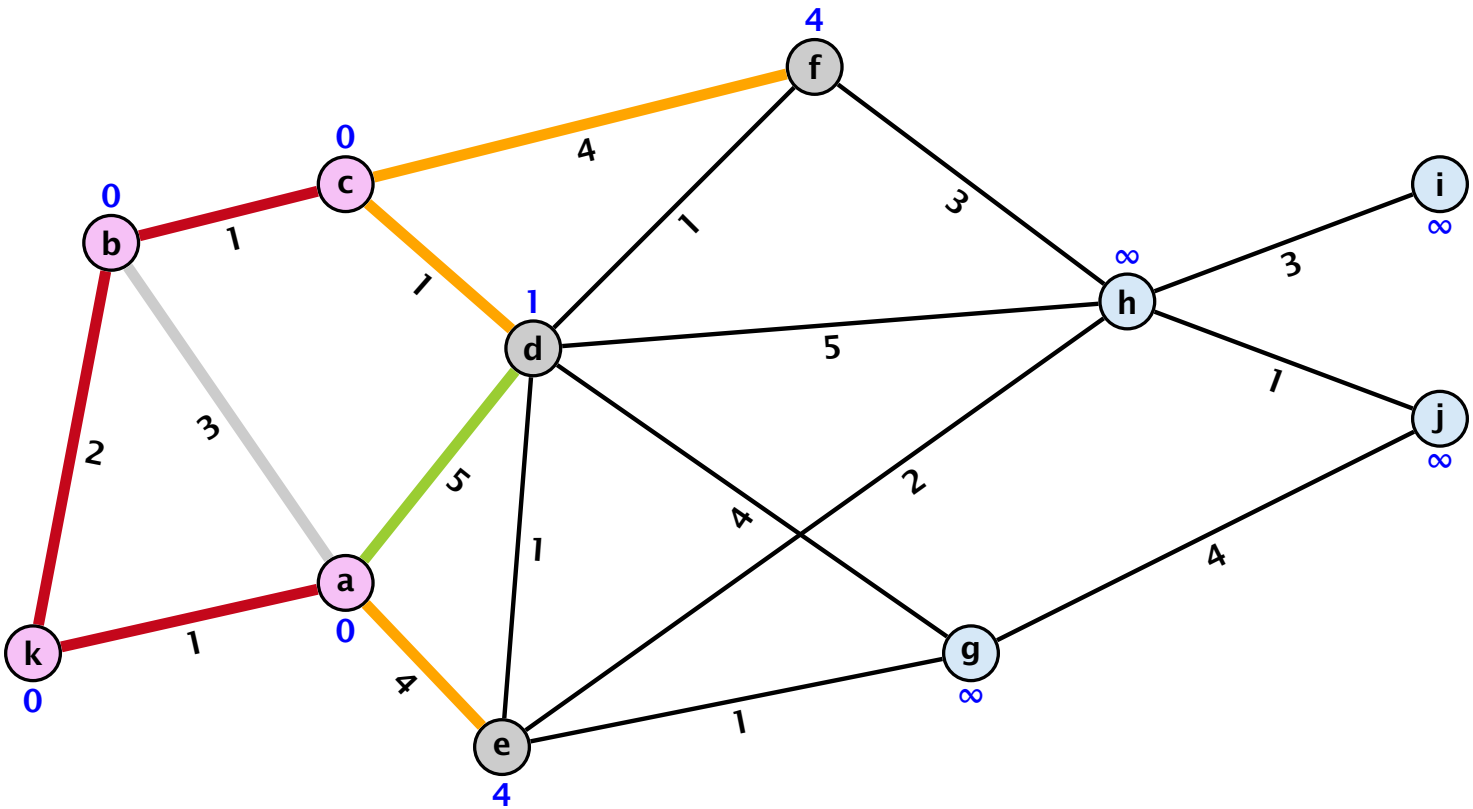
# Minimaler Spannbaum (Prim)



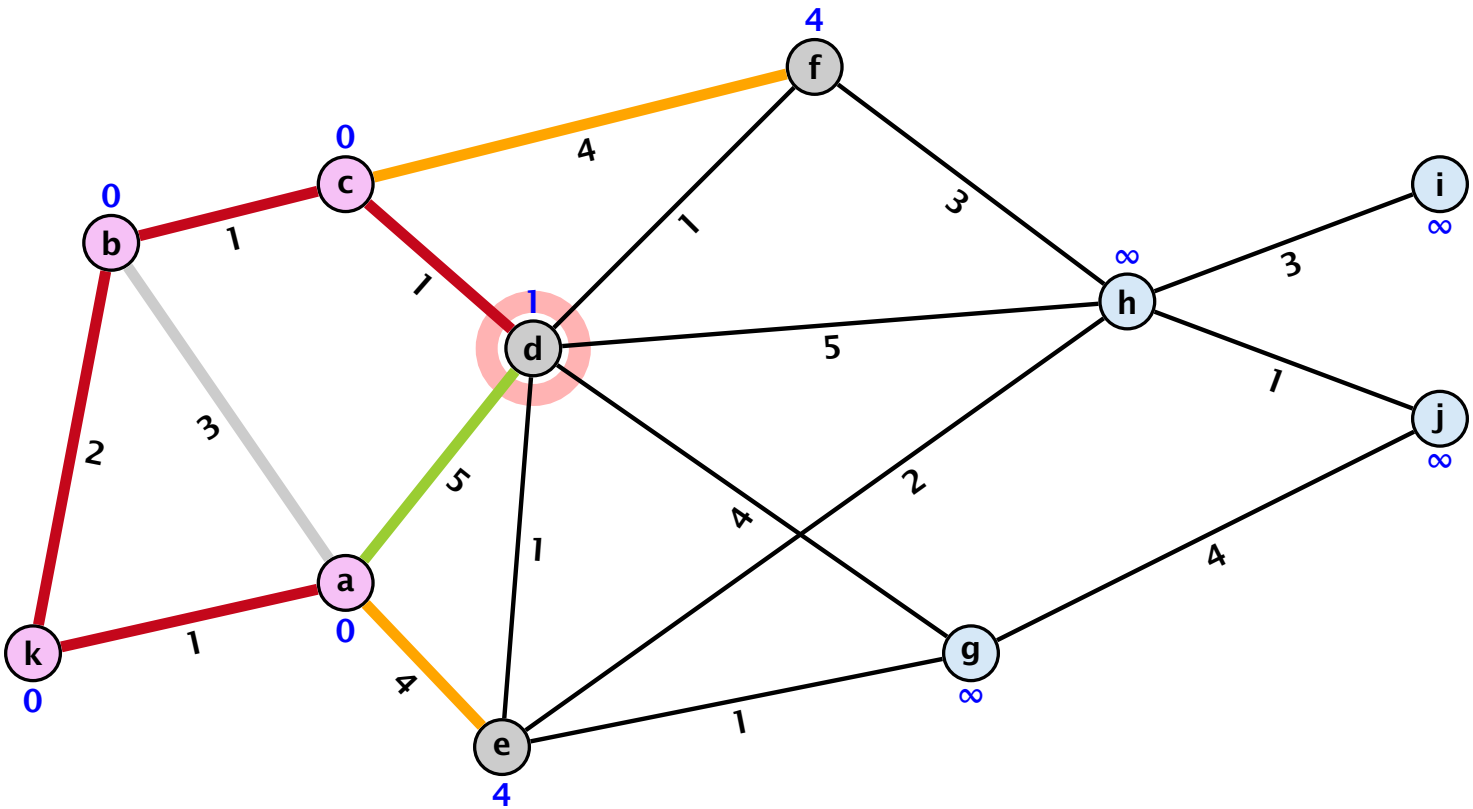
# Minimaler Spannbaum (Prim)



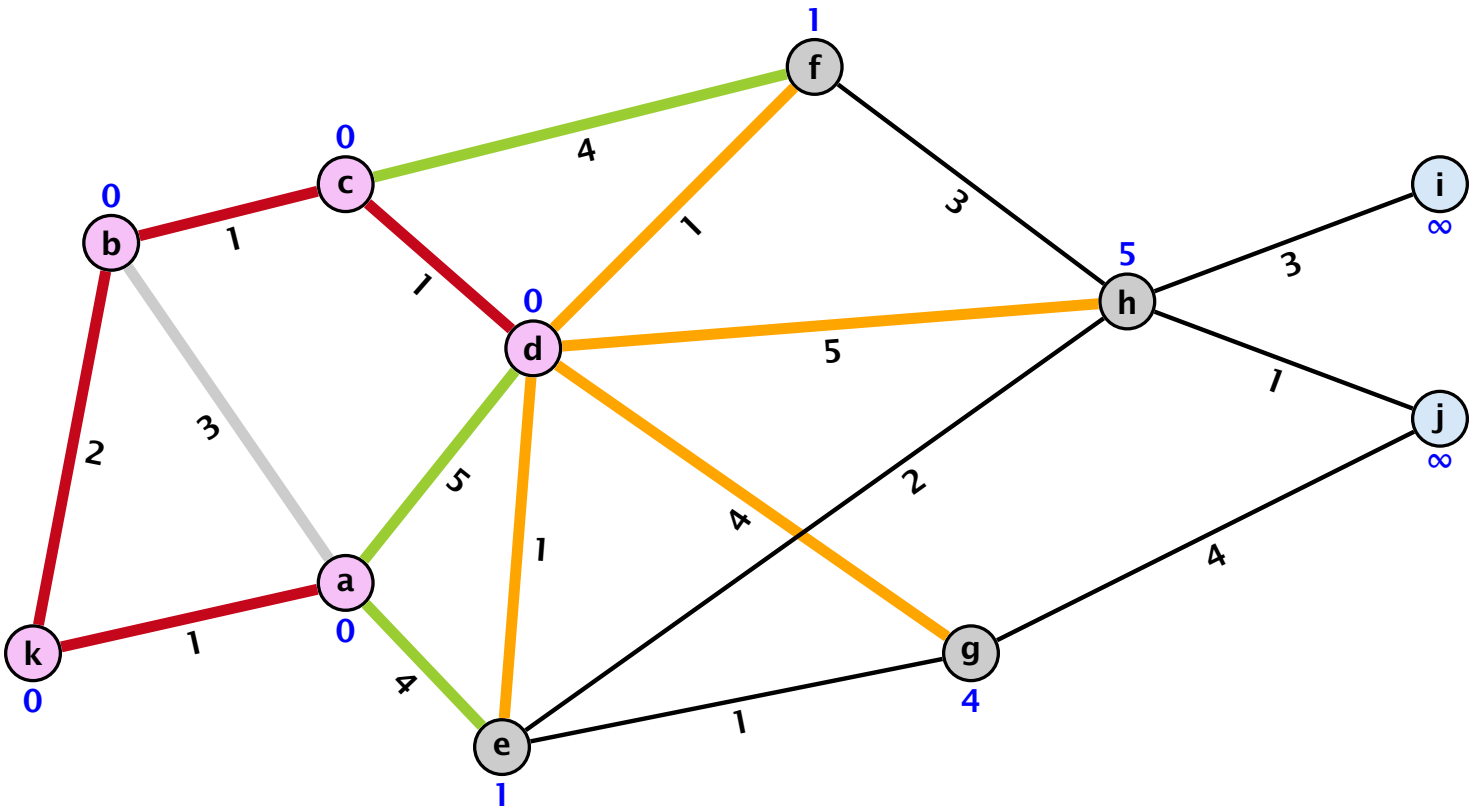
# Minimaler Spannbaum (Prim)



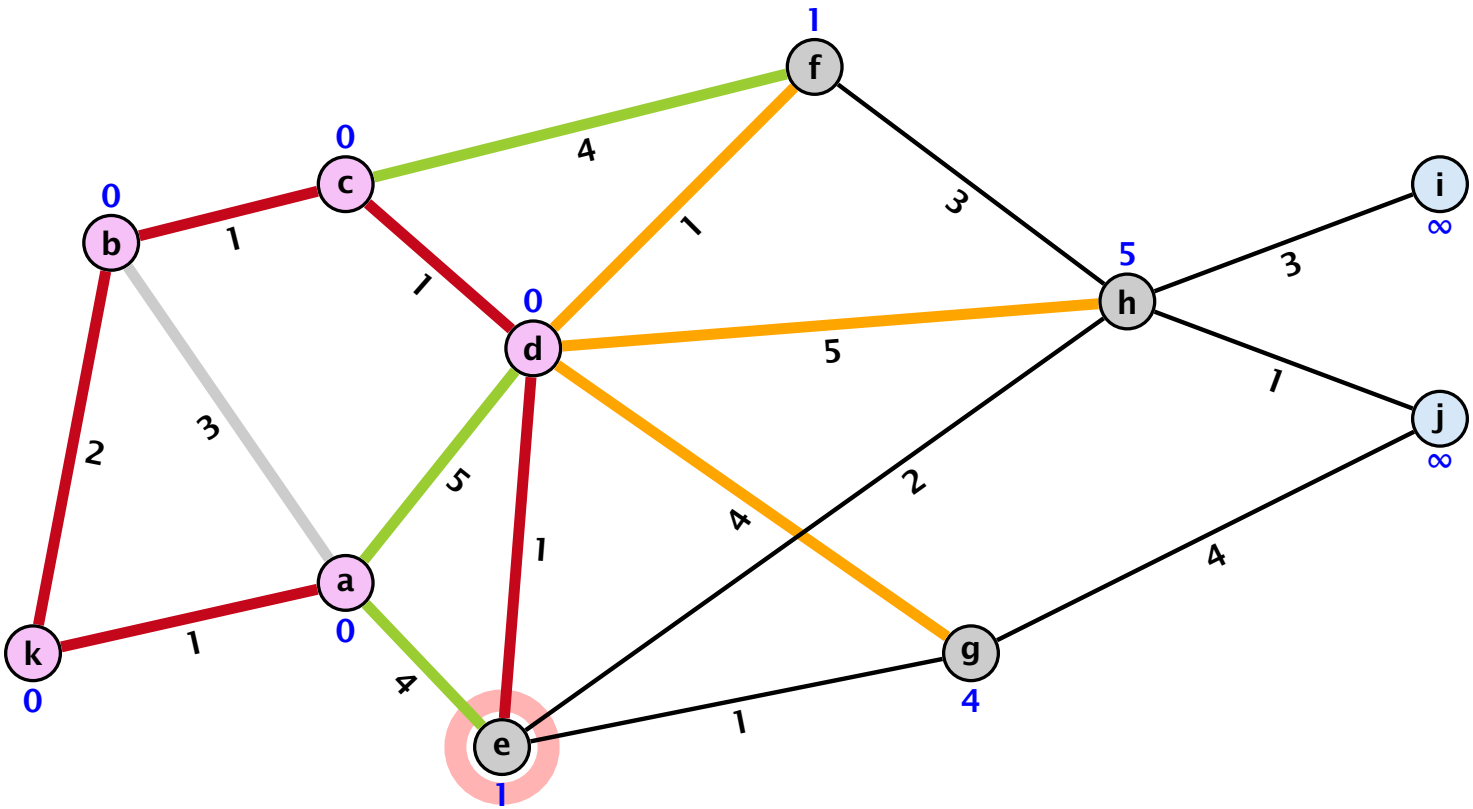
# Minimaler Spannbaum (Prim)



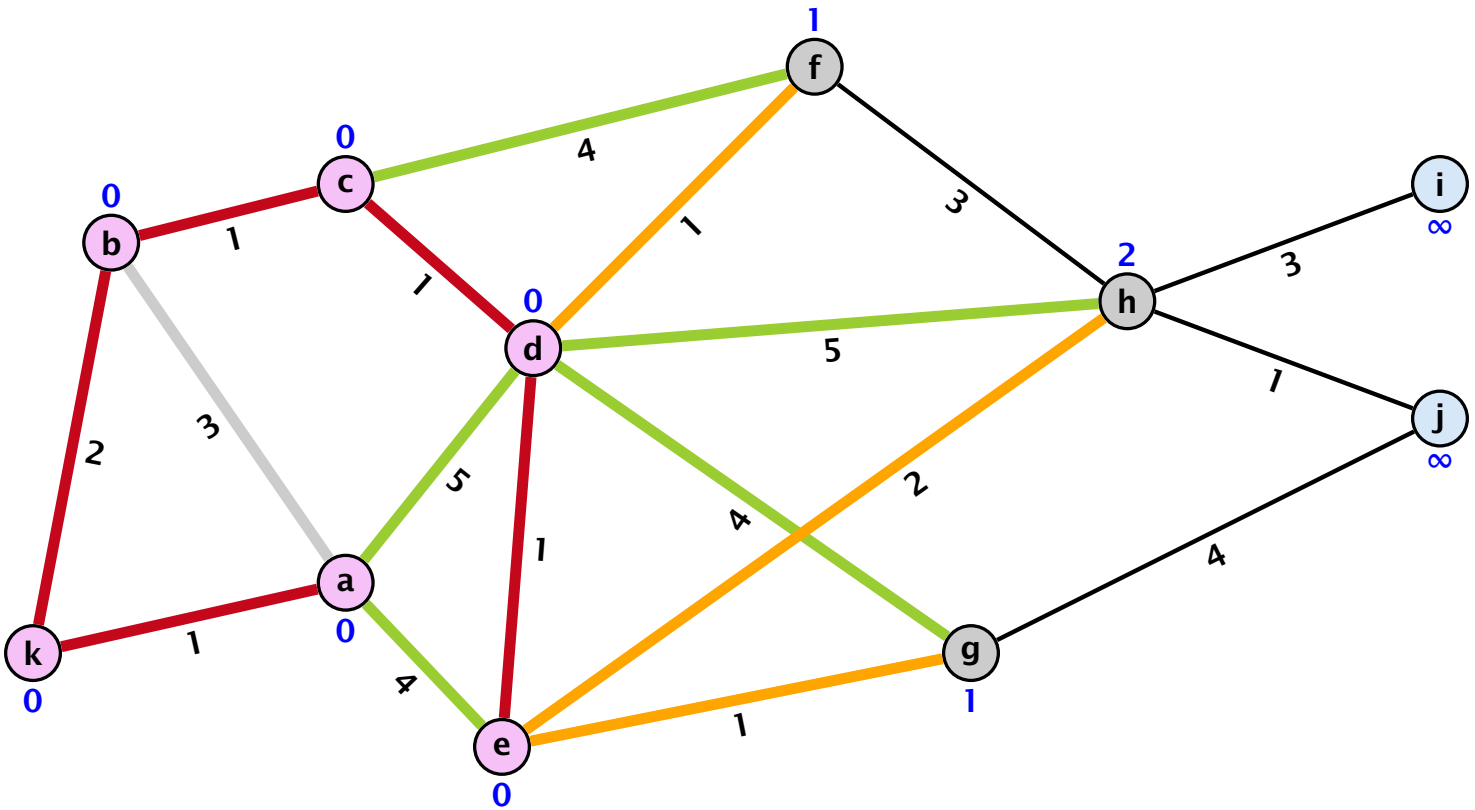
# Minimaler Spannbaum (Prim)



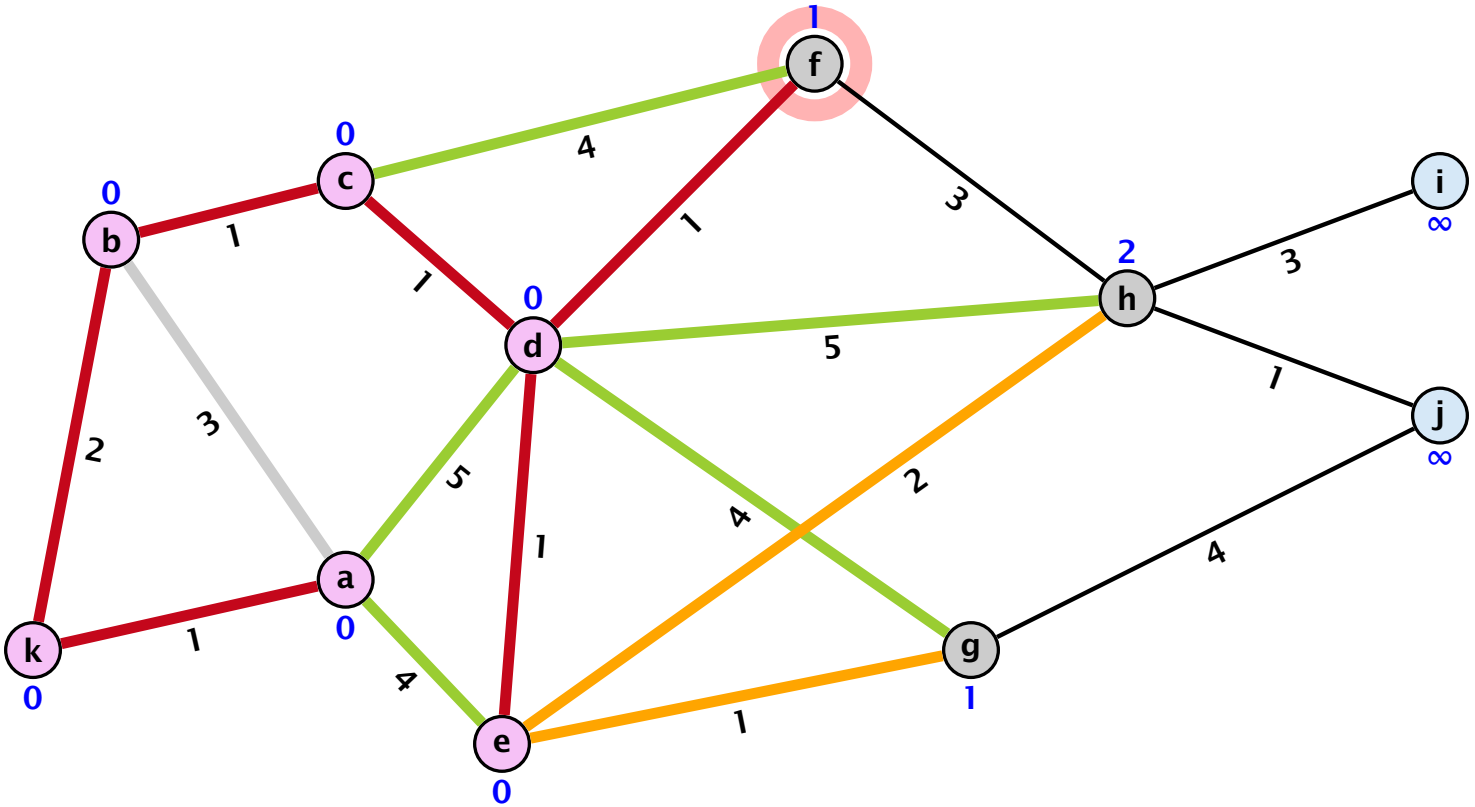
# Minimaler Spannbaum (Prim)



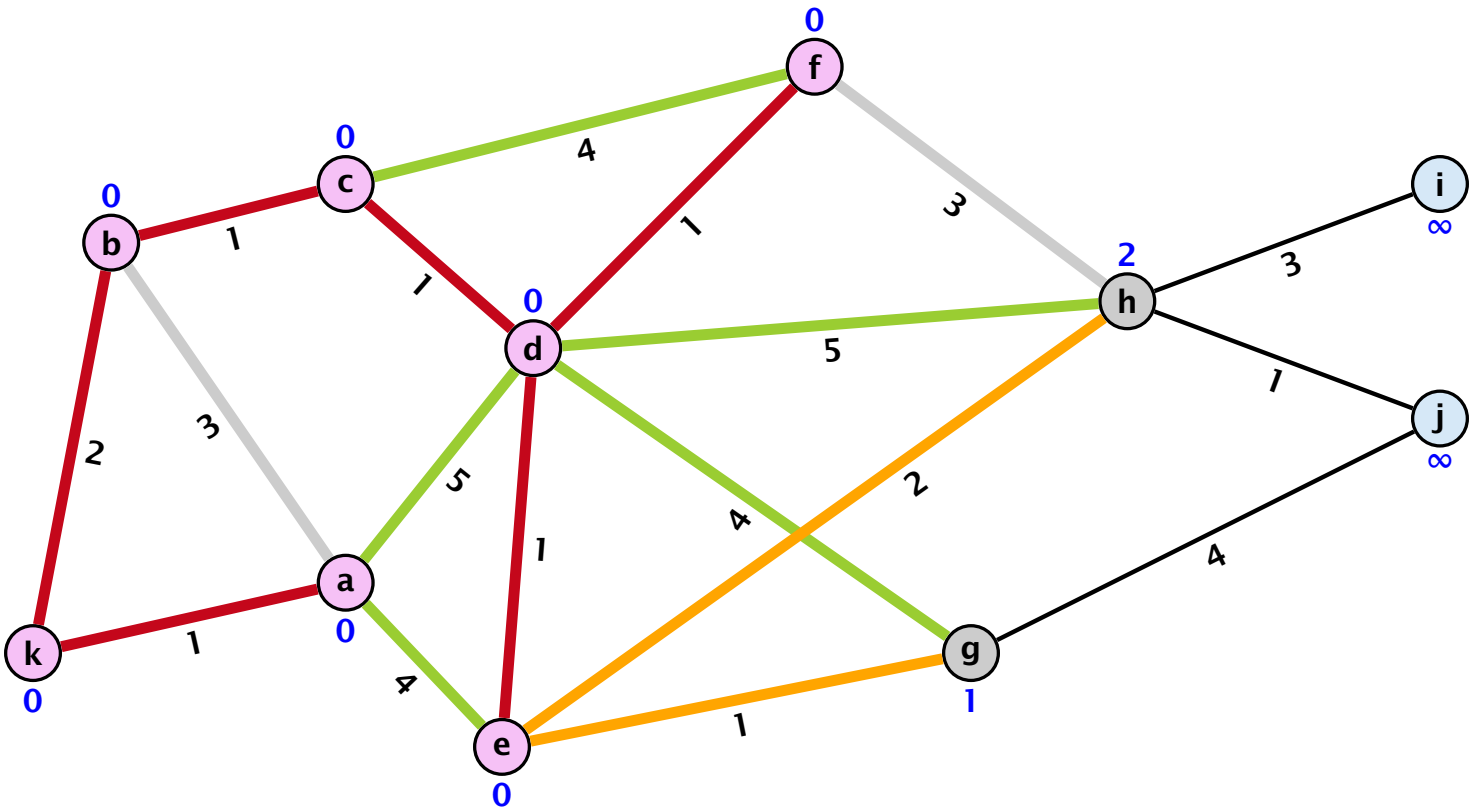
# Minimaler Spannbaum (Prim)



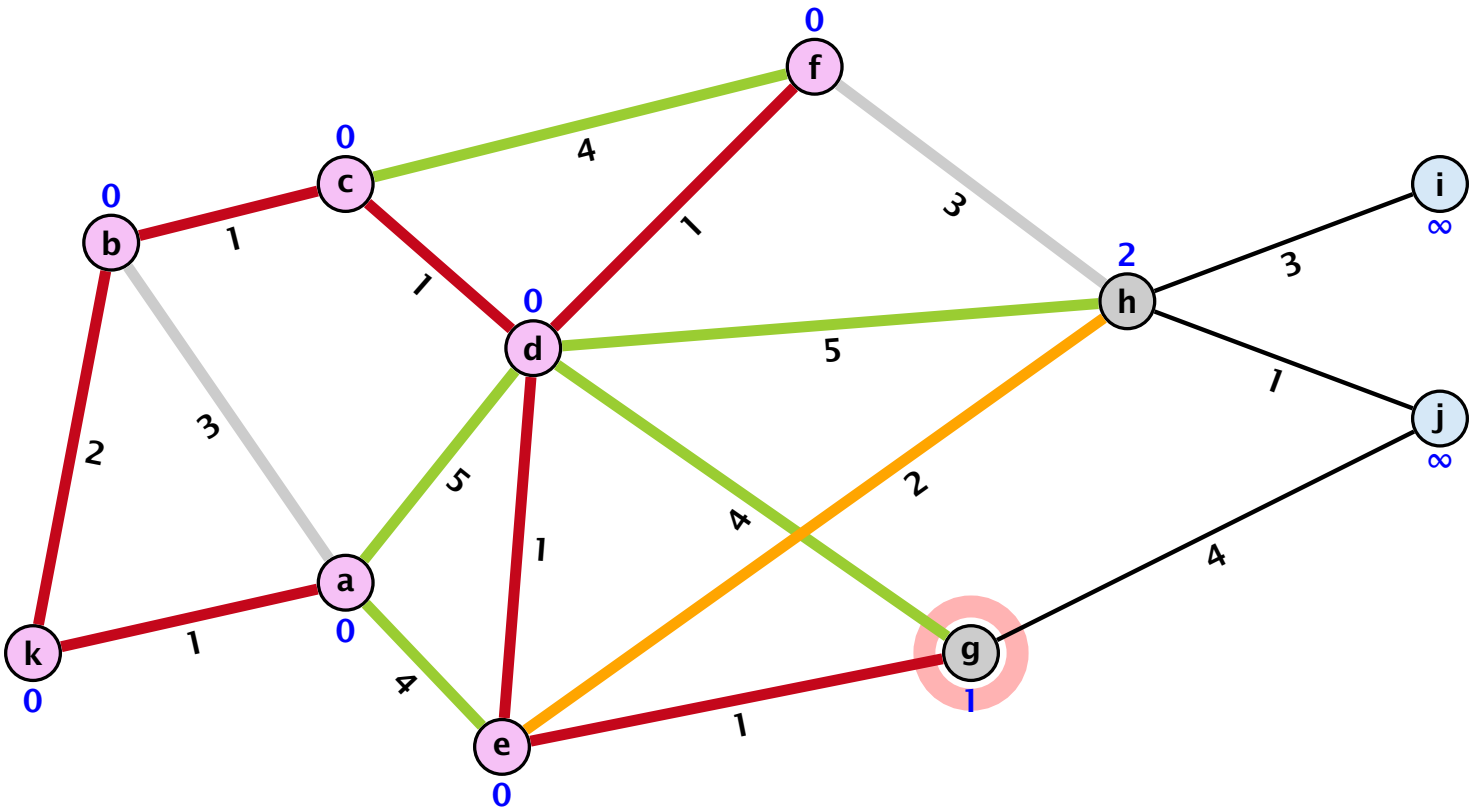
# Minimaler Spannbaum (Prim)



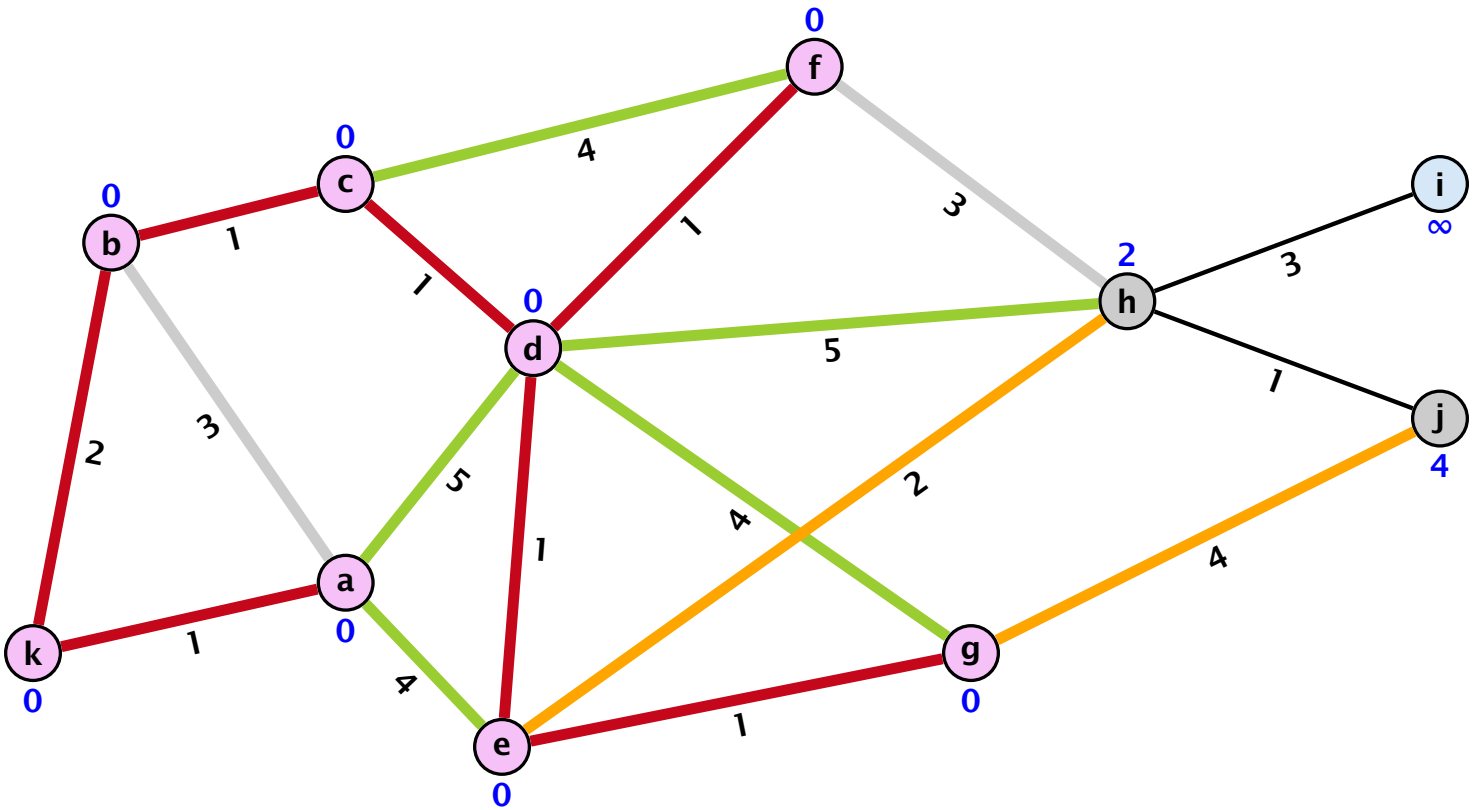
# Minimaler Spannbaum (Prim)



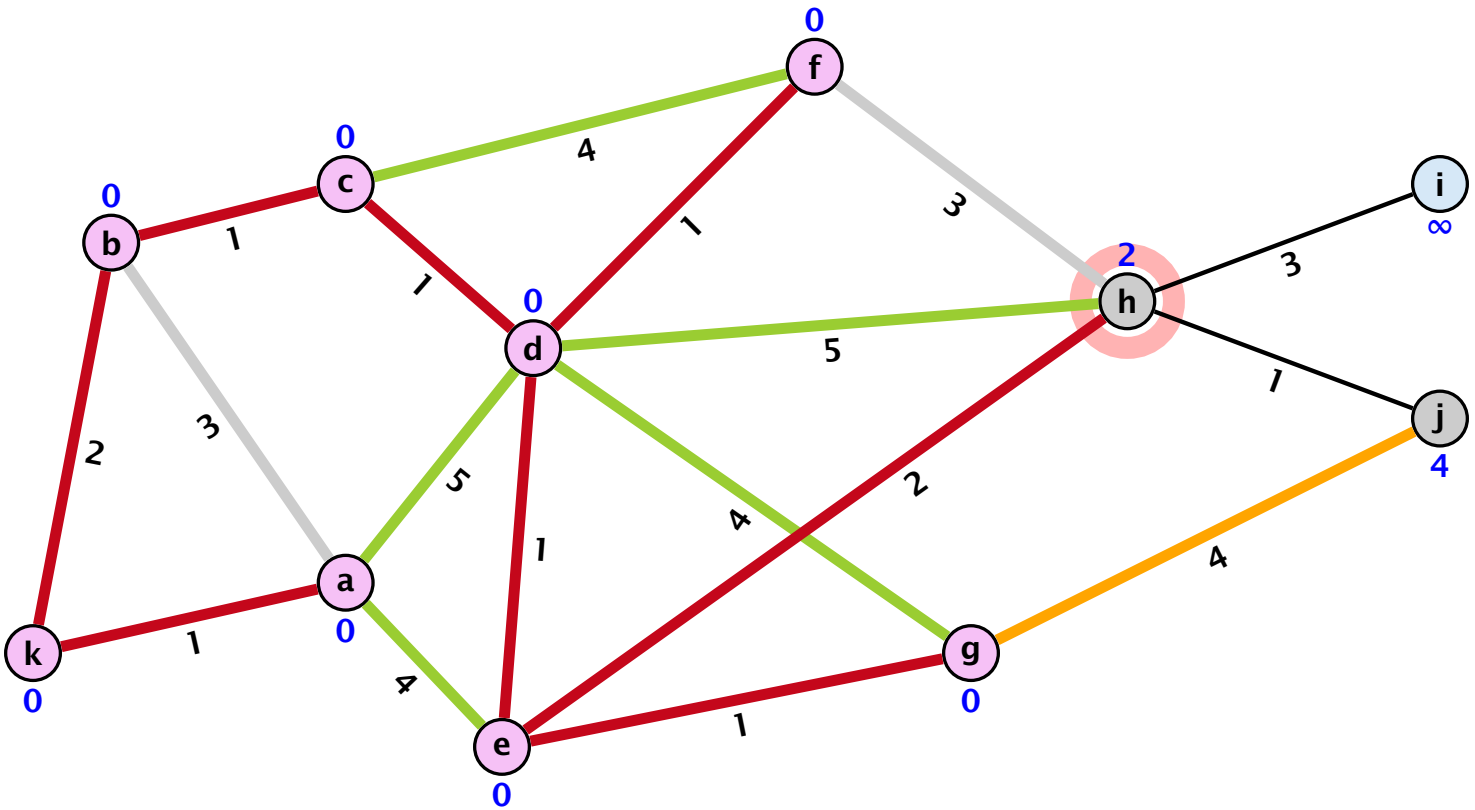
# Minimaler Spannbaum (Prim)



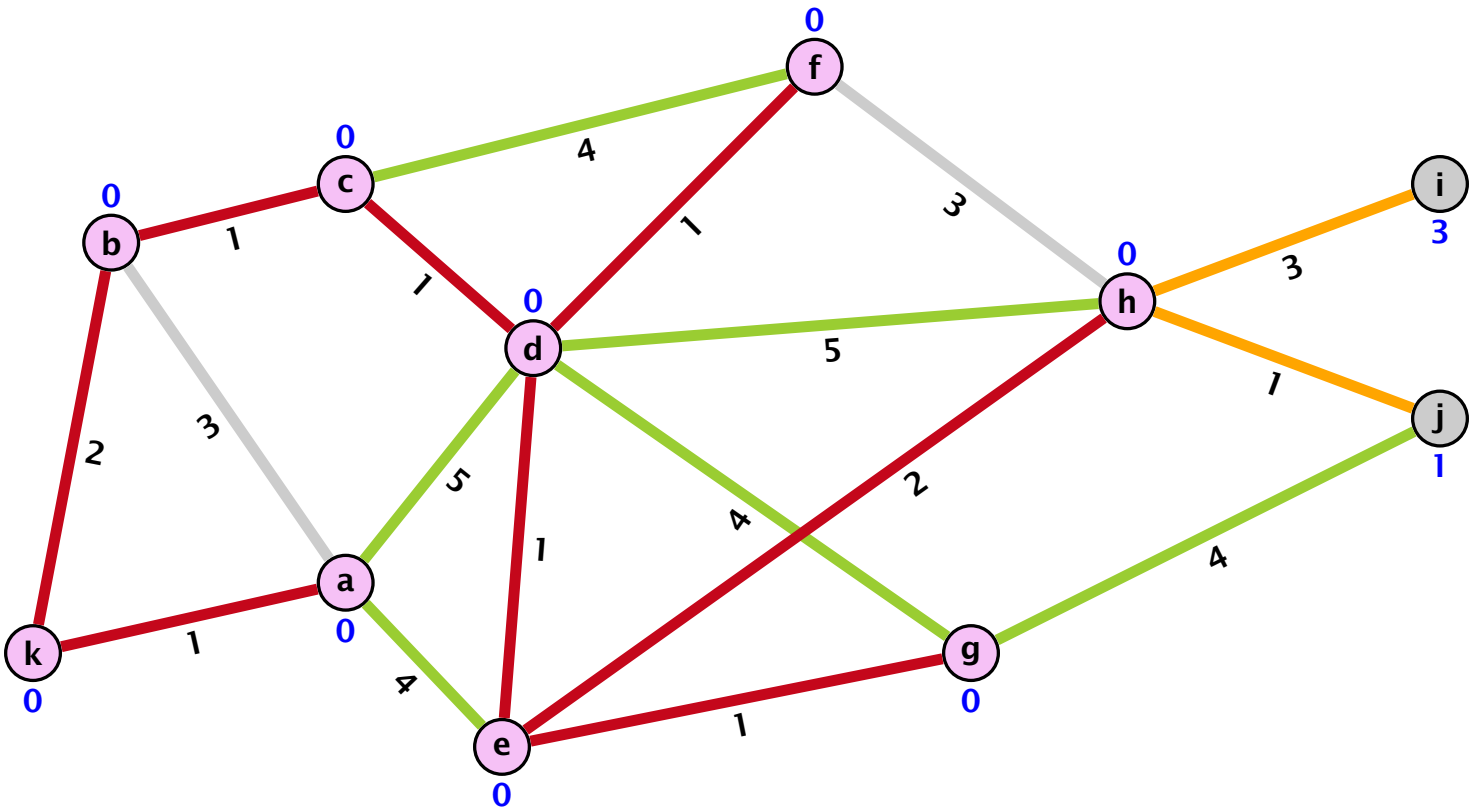
# Minimaler Spannbaum (Prim)



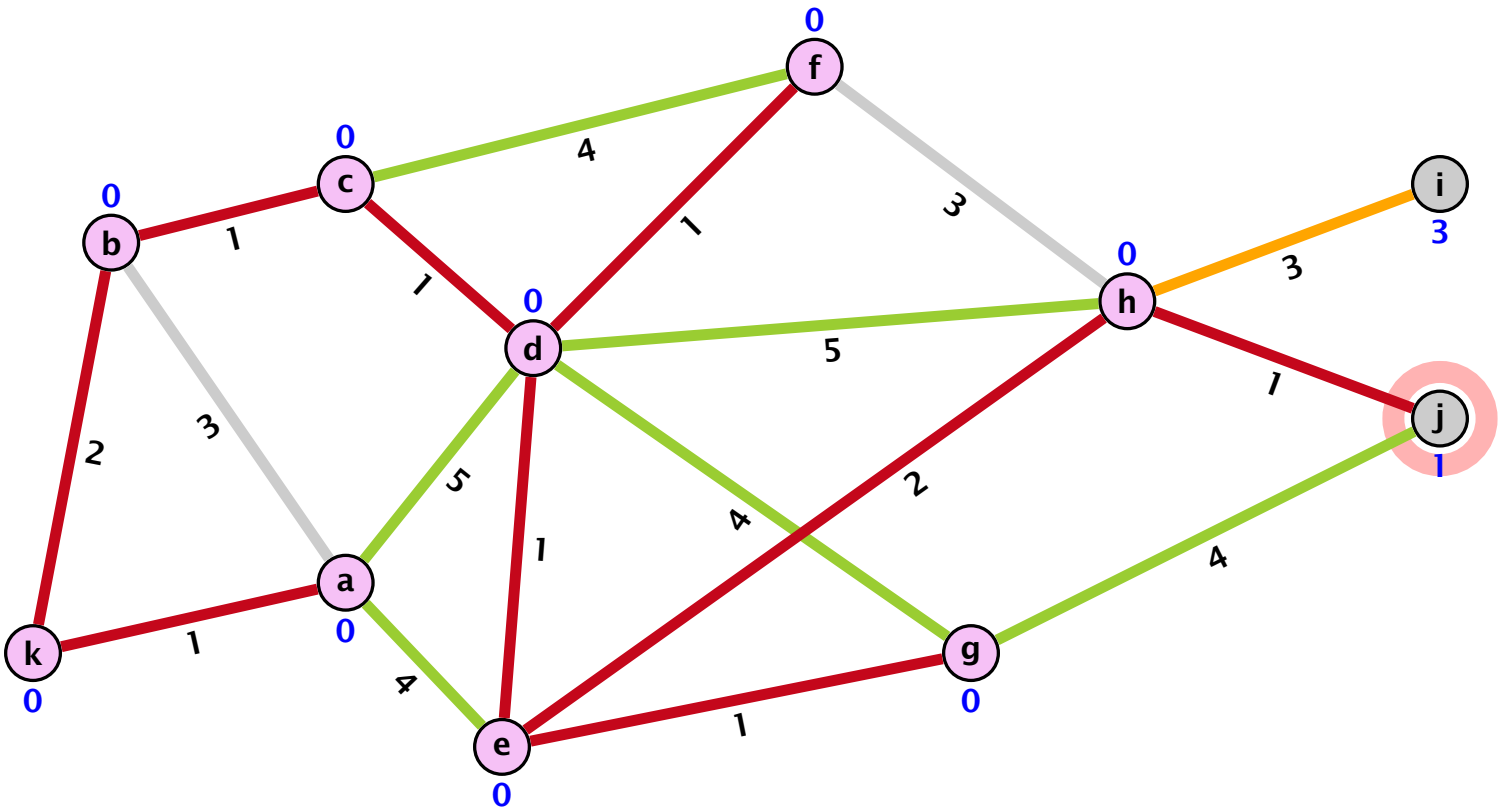
# Minimaler Spannbaum (Prim)



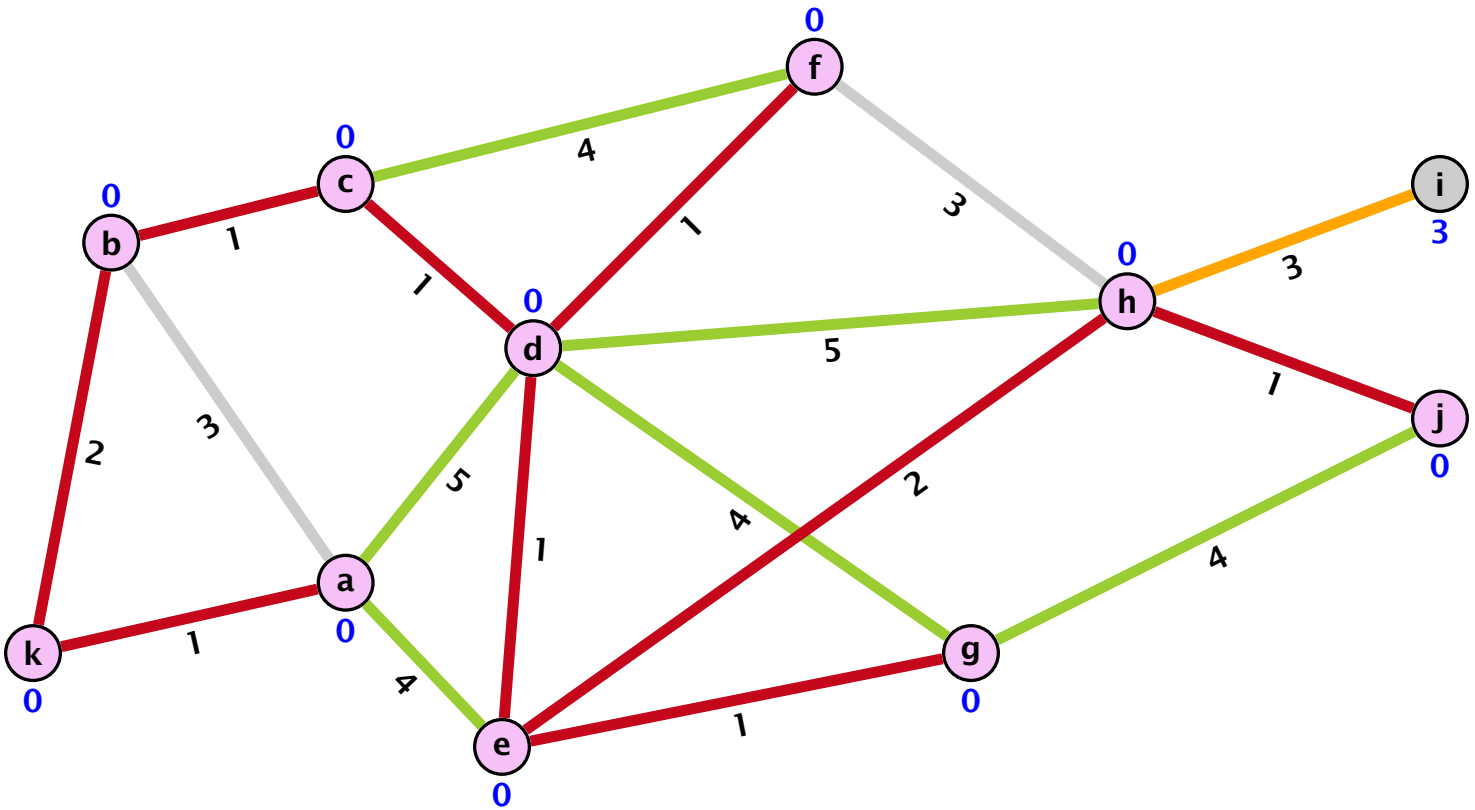
# Minimaler Spannbaum (Prim)



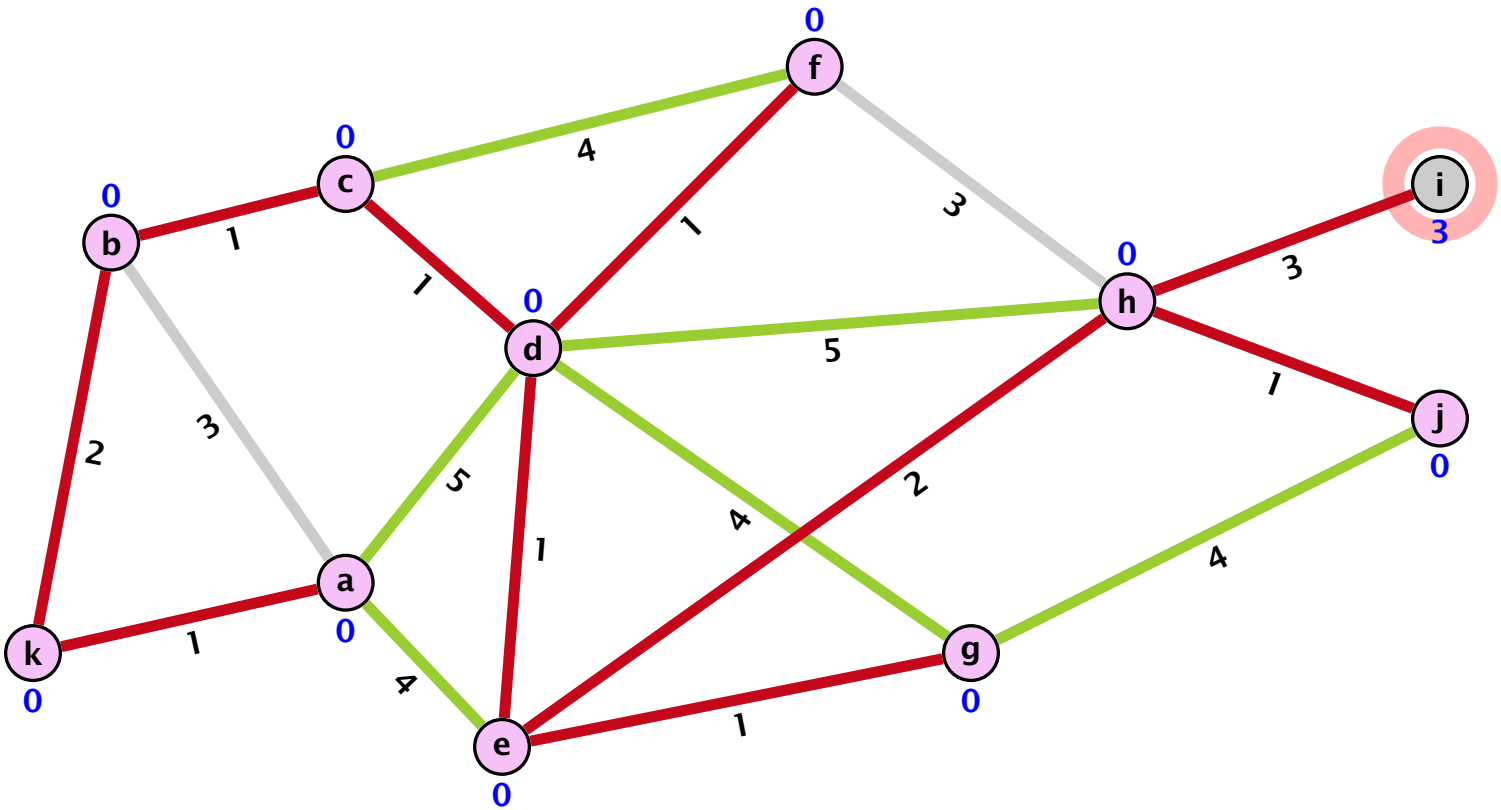
# Minimaler Spannbaum (Prim)



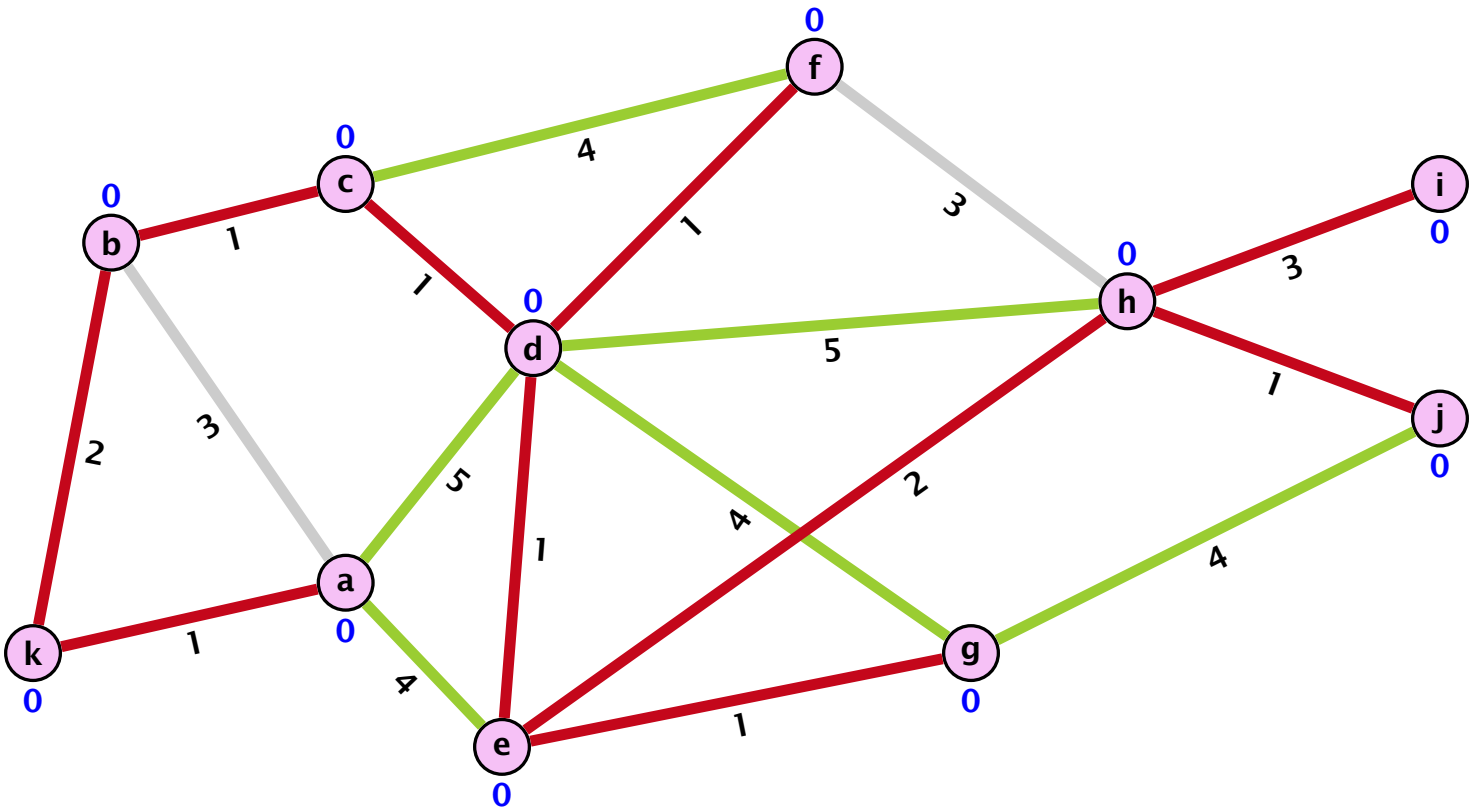
# Minimaler Spannbaum (Prim)



# Minimaler Spannbaum (Prim)

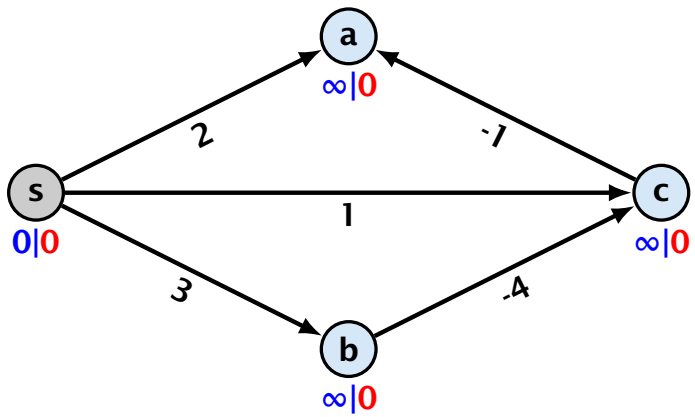


# Minimaler Spannbaum (Prim)



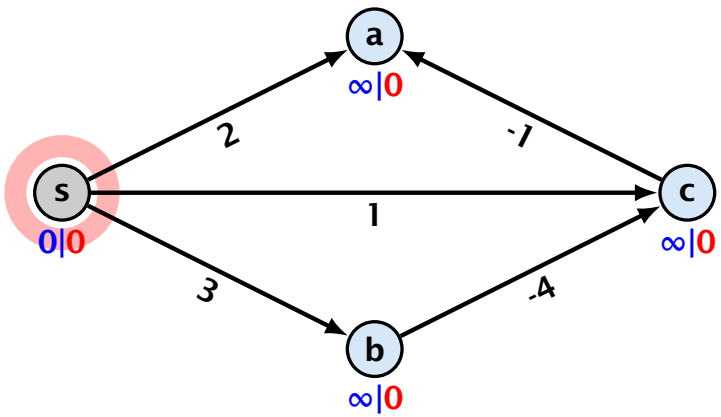
# Negative Kantengewichte

Dijkstra funktioniert nicht:



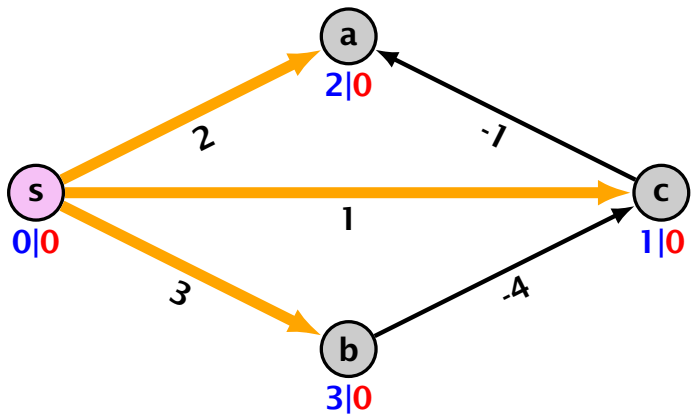
# Negative Kantengewichte

Dijkstra funktioniert nicht:



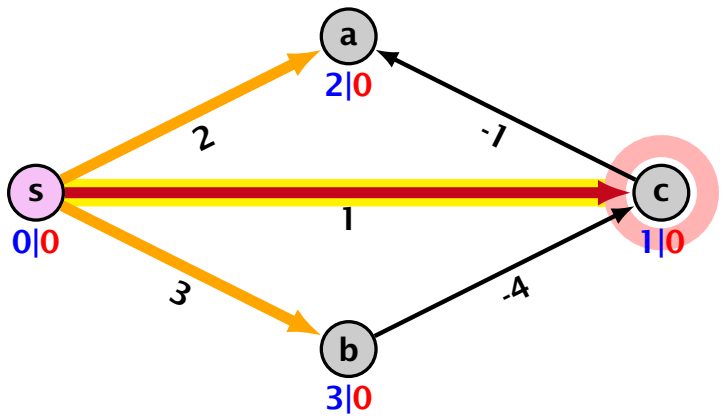
# Negative Kantengewichte

Dijkstra funktioniert nicht:



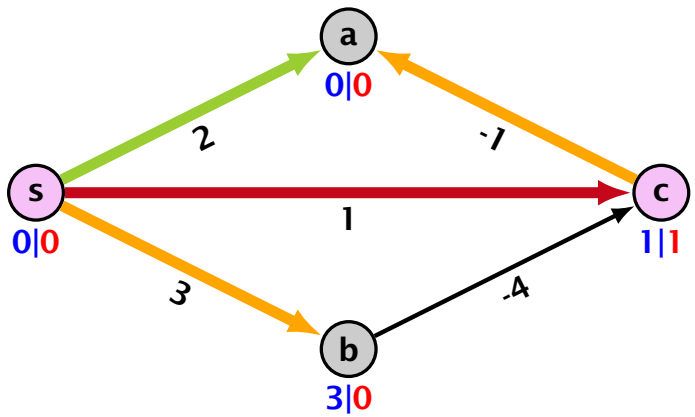
# Negative Kantengewichte

Dijkstra funktioniert nicht:



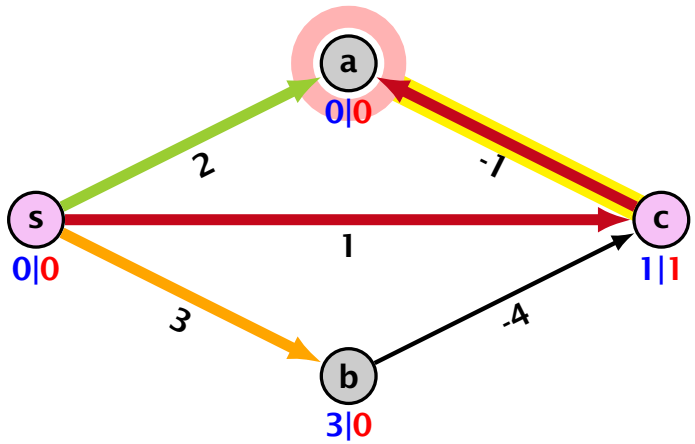
# Negative Kantengewichte

Dijkstra funktioniert nicht:



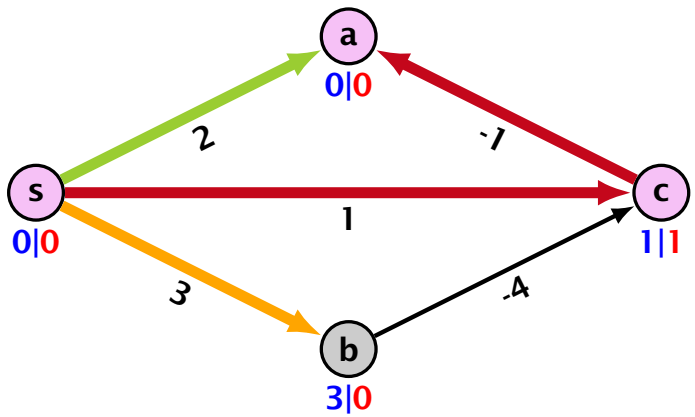
# Negative Kantengewichte

Dijkstra funktioniert nicht:



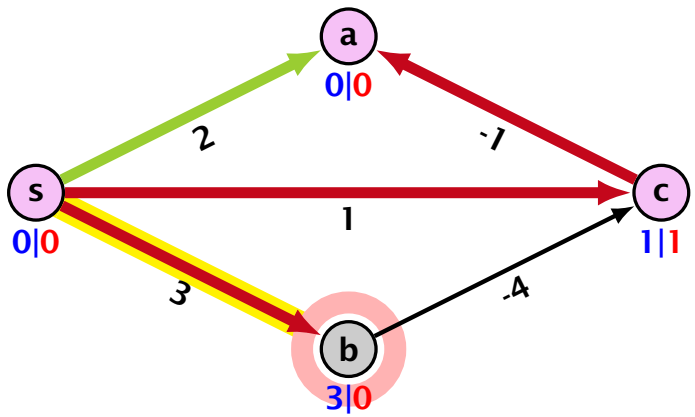
# Negative Kantengewichte

Dijkstra funktioniert nicht:



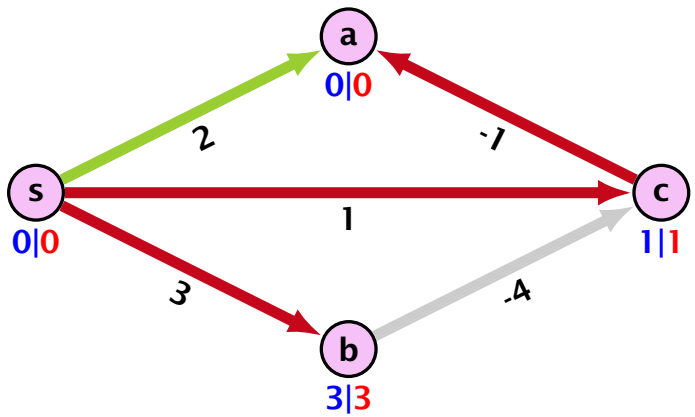
# Negative Kantengewichte

Dijkstra funktioniert nicht:



# Negative Kantengewichte

Dijkstra funktioniert nicht:



# Negative Kantengewichte

**Lemma 7**

*Für jeden Knoten $x$ mit $d(s,x) \neq \pm\infty$, existiert ein kürzester Weg mit höchstens $n-1$ Kanten zwischen $s$ und $x$.*

**Beweis:**

- sei $x$ ein Knoten mit $d(s,x) \neq \pm\infty$; dann existiert kürzester Pfad; sei $P$ solch ein Pfad mit wenigsten Kanten
- angenommen, $P$ hat $\geq n$ Kanten (d.h. $\geq n+1$ Knoten)
- dann enthält $P$ einen gerichteten Zyklus $Z$
- wenn $Z$ negatives Gewicht hat gilt $d(s,x) = -\infty$ (↯)
- andernfalls entfernen wir $Z$ aus $P$ und erhalten ein Pfad der Länge höchstens $w(P)$ hat mit weniger hops (↯)

# Bellman-Ford

Für $n-1$ Phasen relaxiert man alle Kanten.

```
1  Input: weighted graph G=(V,E,w); start vertex s;
2  Output: key-field of very node contains distance from s
3
4  NaiveBellmanFord(s)
5      foreach v ∈ V
6          v->key = ∞;
7      s->key = 0;
8      for i = 1 to n-1
9          foreach (x,y) ∈ E
10             y->key = min(y->key, x->key + w(x,y))
```

Für hop-minimalen kürzesten $s$-$x$ Pfad $P$ existiert Teilfolge von Relaxierungen, die $P$ von $s$ nach $x$ abarbeitet.

$\Longrightarrow$ alle Distanzen sind am Ende korrekt, falls man keine negativen Kreise hat

# Bellman-Ford

Wie finden wir Knoten, die von $s$ aus über einen Pfad erreichbar sind der einen Knoten eines negativen Kreises enthält? (Das sind genau die Knoten die Distanz $-\infty$ haben)

# Bellman-Ford

Wir fügen eine weitere Phase hinzu.

**Lemma 8**

*Jeder von $s$ erreichbare negative Zyklus enthält einen Knoten, der in Phase $n$ sein Distanzlabel ändert.*

# Bellman-Ford

**Beweis:**

- Sei $Z$ negativer Zyklus (erreichbar von $s$). Nach Phase $n-1$ haben alle Knoten des Zyklus endliches Label.
- Angenommen kein Knoten in $Z$ ändert sein Label in Phase $n$. Das bedeutet

$$v_{i+1} \rightarrow \text{key} \leq v_i \rightarrow \text{key} + w(v_i, v_{i+1})$$

# Bellman-Ford

D.h. wir müssen nur all Knoten markieren die in Runde $n$ ihr Label ändern (und alle Knoten, die von diesen erreichbar sind).

Dies geschieht durch folgende Routine:

```
1 mark(x)
2       if (x->key != −∞)
3           x->key = −∞;
4           for ((x,y) ∈ E) mark(y)
```

Falls wir $\mathrm{mark}(x)$ für alle Knoten $x \in S \subseteq V$ (für beliebige Teilmenge $S$) aufrufen benötigt dies Zeit $\mathcal{O}(|S| + m)$, da jede Kante nur einmal geprüft wird.

# Bellman-Ford

```
1  BellmanFord(s)
2      foreach v ∈ V
3          v->key = ∞;
4          v->par = NULL;
5      s->key = 0;
6      for i = 1 to n-1
7          foreach (x,y) ∈ E
8              if (y->key > x->key + w(x,y))
9                  y->key = x->key + w(x,y);
10                 y->par = x;
11     foreach (x,y) ∈ E
12         if (y->key > x->key + w(x,y))
13             y->par = x;
14             mark(y)
```

Laufzeit: $\mathcal{O}(mn)$

# Bellman-Ford

**Invariante:**

Nach der $\ell$-ten Runde zeigt parent-Zeiger von $x$ auf den Vorgänger von $x$ auf einem kürzesten $s$-$x$ Pfad mit höchstens $\ell$ Kanten.

$x.\mathrm{parent} = \varnothing$ bedeutet **entweder**, dass es keinen $s$-$x$ Pfad mit höchstens $\ell$ Kanten gibt **oder** dass der kürzeste dieser Pfade leer ist ($s = x$).

**Nach Terminierung:**

Für Knoten $v$ mit $d(s,v) \neq \pm\infty$ führen die parent pointer von $x$ nach $s$ (rückwärts entlang kürzestem $s$-$v$ Pfad).

Wenn man den parent-Zeigern von Knoten $v$ mit $d(s,v) = -\infty$ folgt gelangt man zu einem negativen Kreis (dies kann man nutzen um einen negativen Kreis zu finden).

# Bellman-Ford

## Verbesserung

- Benutze Queue, die Knoten speichert zu denen ein kürzerer Pfad gefunden wurde (und deren ausgehende Kanten deshalb relaxiert werden müssen).
- Wiederhole:
  Entferne Element $x$ aus der Queue und relaxiere ausgehende Kanten.

  Falls Relaxierung entlang Kante $(x, y)$ erfolgreich wird $y$ in die Queue aufgenommen (falls noch nicht enthalten).

  Runde/Phase endet wenn wenn die zu Anfang der Phase in der Queue enthaltenen Knoten bearbeitet sind.

# APSP – Floyd Warshall

gegeben:

- Graph mit beliebigen Kantengewichten; keine negativen Kreise

gesucht:

- Distanzen/kürzeste Weg zwischen allen Knotenpaaren.

Naive Strategie

- $n$-mal Bellman-Ford. Laufzeit $\mathcal{O}(m \cdot n^2)$.

# APSP – Floyd Warshall

**Beobachtung:**
geht der kürzeste $u$-$w$ Pfad über $v$ dann sind auch die Teile von $u$ nach $v$ und $v$ nach $w$ kürzeste Pfade.

**Dynamisches Programmieren:**

- Berechne kürzeste Wege $P_A(u,w)$ von $u$ nach $w$, die nur Zwischenknoten aus Menge $A$ benutzen (initial ist Menge $A$ leer).
- Wenn man alle Pfade $P_A(u,w)$ kennt, kann man einfach die Pfade $P_{A\cup\{v\}}(u,w)$ berechnen.
- Am Ende möchten wir $P_V(u,w)$ für alle Paare $u,w$ kennen.

# APSP – Floyd Warshall

```
1  Input: weighted graph G=(V,E,w);
2  Output: d[u,v] is distance from u to v; pred[u,v]
3          is predecessor of v on shortest path tree from u
4
5  FloydWarshall(G)
6      for u,v ∈ V
7          d[u,v]    = ∞;
8          pred[u,v] = NULL;
9      for v ∈ V
10         d[u,v]    = 0;
11     for (u,v) ∈ E
12         d[u,v]    = w(u,v);
13         pred[u,v] = u;
14     for v ∈ V
15         for {u,w} ∈ V × V
16             if (d[u,w] > d[u,v] + d[v,w])
17                 d(u,w)    = d(u,v) + d(v,w);
18                 pred(u,w) = pred(v,w);
```

# APSP – Floyd Warshall

- Komplexität $\mathcal{O}(n^3)$
- funktioniert auch bei Kanten mit negativem Gewicht
- Kreise negativer Länge verfälschen das Ergebnis; können aber durch negative Diagonaleinträge in der Ergebnismatrix erkannt werden.

# Starker Zusammenhang

Wir können BFS oder DFS benutzen um Zusammenhangskomponenten in einem ungerichteten Graphen zu finden.

Die einzelnen Bäume des berechneten Waldes sind die Zhks.

**Wie berechnen wir starke Zusammenhangskomponenten in einem gerichteten Graphen?**

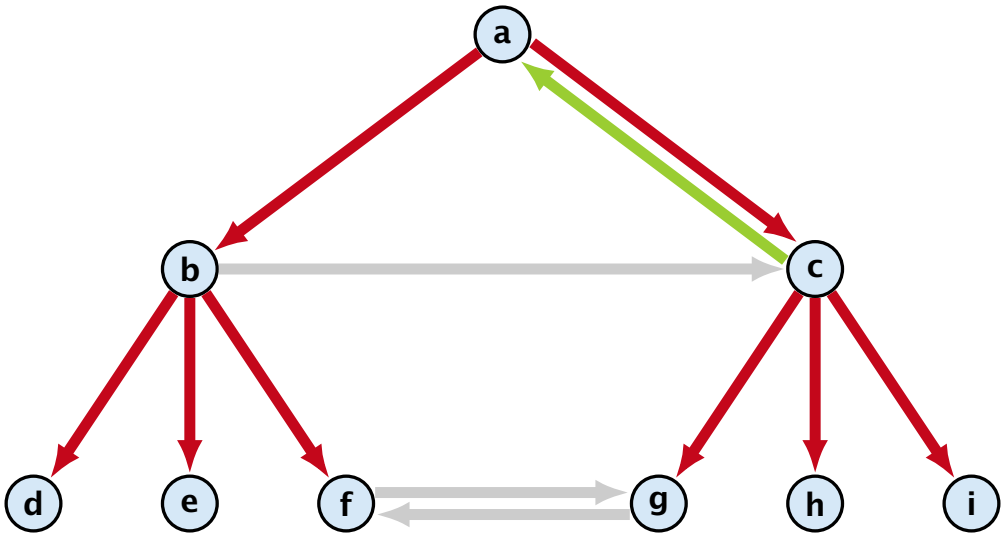
# Starker Zusammenhang

**Beobachtung**
Die starken Zhks müssen innerhalb der durch die Tiefen- oder Breitensuche berechneten Bäume liegen, d.h. eine Zhk kann nicht Knoten von verschiedenen Bäumen enthalten.

Wir müssen nur die einzelnen Bäume in (starke) Zhks zerlegen.

# Starker Zusammenhang

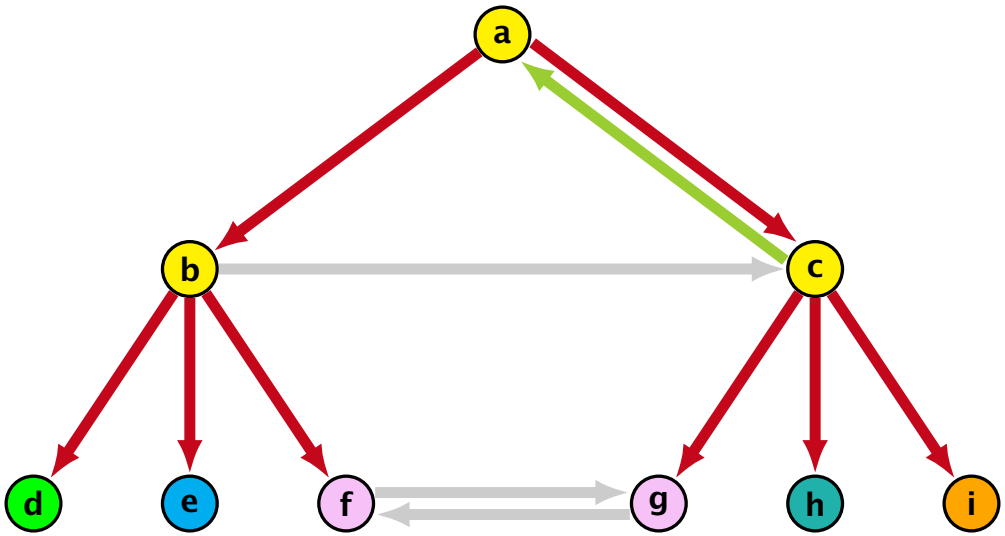
**BFS-Baum:**




Die (starken) Zhks sind nicht über Baumkanten verbunden.

# Starker Zusammenhang

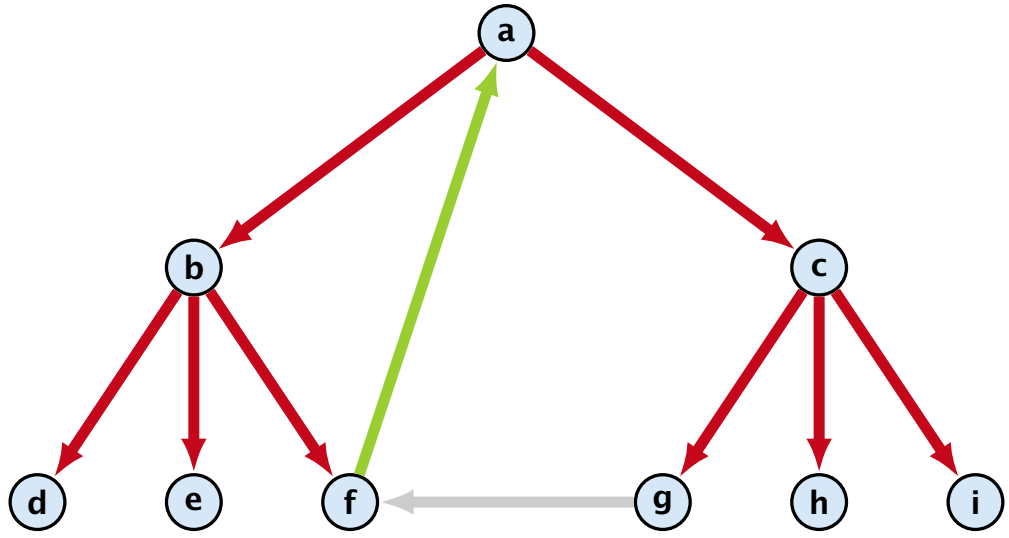
**BFS-Baum:**




Die (starken) Zhks sind nicht über Baumkanten verbunden.

# Starker Zusammenhang

**DFS-Baum:**

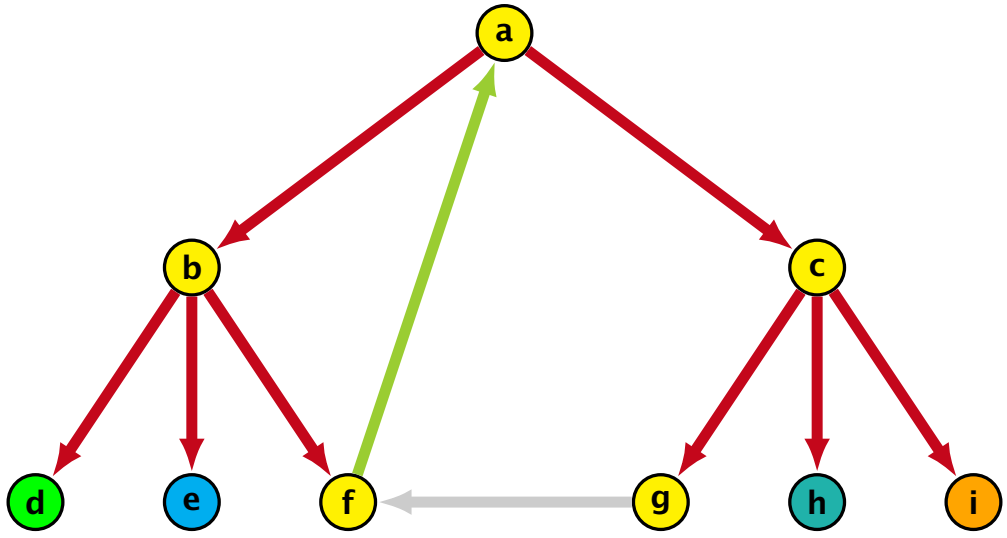



In einem DFS-Baum sind die starken Zusammenhangskomponenten über die Baumkanten verbunden (wenn man diese als ungerichtet annimmt).

D.h. für 2 Knoten $a$ und $b$, die in der gleichen Zhk sind, ist auch jeder Knoten auf dem (ungerichteten) Weg zwischen $a$ und $b$ im DFS-Baum in der gleichen Zhk.

# Starker Zusammenhang

**DFS-Baum:**




In einem DFS-Baum sind die starken Zusammenhangskomponenten über die Baumkanten verbunden (wenn man diese als ungerichtet annimmt).

D.h. für 2 Knoten $a$ und $b$, die in der gleichen Zhk sind, ist auch jeder Knoten auf dem (ungerichteten) Weg zwischen $a$ und $b$ im DFS-Baum in der gleichen Zhk.

# Starker Zusammenhang

**Beweis**

Sei $a, b \in Z$, wobei $Z$ starke Zhk ist. Wir zeigen zunächst $\text{lca}(a, b) \in Z$ wobei $\text{lca}(a, b)$, der kleinste gemeinsame Vorgänger von $a$ und $b$ im Baum ist.

**Annahme:**
$a$ und $b$ ist kleinstes Gegenbeispiel, d.h., Gegenbeispiel mit $\text{dfsNum}(a) + \text{dfsNum}(b)$ minimal. Sei $\text{dfsNum}(a) < \text{dfsNum}(b)$.

Wenn wir $a$ besuchen ist $b$ noch nicht besucht. Auf dem Pfad von $a$ nach $b$ muss es einen Knoten $x$ geben, der entweder aktiv ist oder schon besucht wurde (sonst würde $b$ im Teilbaum von $a$ enden).

# Starker Zusammenhang

**Beweis**

Sei $a, b \in Z$, wobei $Z$ starke Zhk ist. Wir zeigen zunächst $\text{lca}(a, b) \in Z$ wobei $\text{lca}(a, b)$, der kleinste gemeinsame Vorgänger von $a$ und $b$ im Baum ist.

**Annahme:**

$a$ und $b$ ist kleinstes Gegenbeispiel, d.h., Gegenbeispiel mit $\text{dfsNum}(a) + \text{dfsNum}(b)$ minimal. Sei $\text{dfsNum}(a) < \text{dfsNum}(b)$.

Wenn wir $a$ besuchen ist $b$ noch nicht besucht. Auf dem Pfad von $a$ nach $b$ muss es einen Knoten $x$ geben, der entweder aktiv ist oder schon besucht wurde (sonst würde $b$ im Teilbaum von $a$ enden).

# Starker Zusammenhang

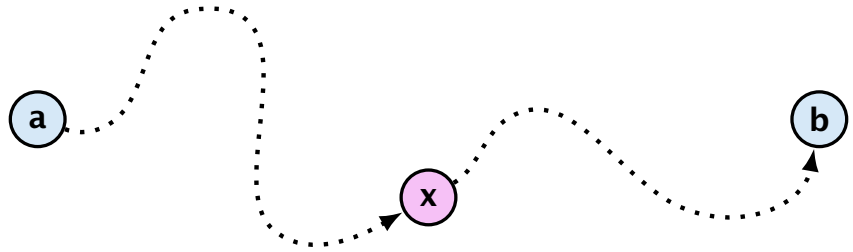
**Beweis**

Sei $a, b \in Z$, wobei $Z$ starke Zhk ist. Wir zeigen zunächst $\text{lca}(a, b) \in Z$ wobei $\text{lca}(a, b)$, der kleinste gemeinsame Vorgänger von $a$ und $b$ im Baum ist.

**Annahme:**

$a$ und $b$ ist kleinstes Gegenbeispiel, d.h., Gegenbeispiel mit $\text{dfsNum}(a) + \text{dfsNum}(b)$ minimal. Sei $\text{dfsNum}(a) < \text{dfsNum}(b)$.

Wenn wir $a$ besuchen ist $b$ noch nicht besucht. Auf dem Pfad von $a$ nach $b$ muss es einen Knoten $x$ geben, der entweder aktiv ist oder schon besucht wurde (sonst würde $b$ im Teilbaum von $a$ enden).



# Starker Zusammenhang

- $x \in Z$, da er von $a$ erreichbar ist und $b$ erreichen kann.
- $\ell := \mathrm{lca}(a,x) \in Z$, da wir sonst ein kleineres Gegenbeispiel hätten.
- Falls $\mathrm{lca}(a,b)$ Vorgänger von $\ell$ folgt, dass $\ell$ und $b$ ein kleineres Gegenbeispiel bilden (beachte $\ell \neq a$). ($\lightning$)
- Falls $\ell$ Vorgänger von $\mathrm{lca}(a,b)$ folgt, $\mathrm{lca}(a,b) \in Z$, da der Knoten von $\ell$ erreichbar ist und $a$ erreichen kann. ($\lightning$)

Das zeigt dass $\mathrm{lca}(a,b) \in Z$. Damit gilt dies auch für Knoten zwischen $\mathrm{lca}(a,b)$ und $a$ bzw. $b$, da diese von einem Knoten aus $Z$ erreichbar sind, und einen Knoten aus $Z$ erreichen können.

# Starker Zusammenhang

- $x \in Z$, da er von $a$ erreichbar ist und $b$ erreichen kann.
- $\ell := \mathrm{lca}(a,x) \in Z$, da wir sonst ein kleineres Gegenbeispiel hätten.
- Falls $\mathrm{lca}(a,b)$ Vorgänger von $\ell$ folgt, dass $\ell$ und $b$ ein kleineres Gegenbeispiel bilden (beachte $\ell \neq a$). ($\lightning$)
- Falls $\ell$ Vorgänger von $\mathrm{lca}(a,b)$ folgt, $\mathrm{lca}(a,b) \in Z$, da der Knoten von $\ell$ erreichbar ist und $a$ erreichen kann. ($\lightning$)

Das zeigt dass $\mathrm{lca}(a,b) \in Z$. Damit gilt dies auch für Knoten zwischen $\mathrm{lca}(a,b)$ und $a$ bzw. $b$, da diese von einem Knoten aus $Z$ erreichbar sind, und einen Knoten aus $Z$ erreichen können.

# Starker Zusammenhang

- $x \in Z$, da er von $a$ erreichbar ist und $b$ erreichen kann.
- $\ell := \mathrm{lca}(a,x) \in Z$, da wir sonst ein kleineres Gegenbeispiel hätten.
- Falls $\mathrm{lca}(a,b)$ Vorgänger von $\ell$ folgt, dass $\ell$ und $b$ ein kleineres Gegenbeispiel bilden (beachte $\ell \neq a$). (↯)
- Falls $\ell$ Vorgänger von $\mathrm{lca}(a,b)$ folgt, $\mathrm{lca}(a,b) \in Z$, da der Knoten von $\ell$ erreichbar ist und $a$ erreichen kann. (↯)

Das zeigt dass $\mathrm{lca}(a,b) \in Z$. Damit gilt dies auch für Knoten zwischen $\mathrm{lca}(a,b)$ und $a$ bzw. $b$, da diese von einem Knoten aus $Z$ erreichbar sind, und einen Knoten aus $Z$ erreichen können.

# Starker Zusammenhang

- $x \in Z$, da er von $a$ erreichbar ist und $b$ erreichen kann.
- $\ell := \mathrm{lca}(a,x) \in Z$, da wir sonst ein kleineres Gegenbeispiel hätten.
- Falls $\mathrm{lca}(a,b)$ Vorgänger von $\ell$ folgt, dass $\ell$ und $b$ ein kleineres Gegenbeispiel bilden (beachte $\ell \neq a$). (↯)
- Falls $\ell$ Vorgänger von $\mathrm{lca}(a,b)$ folgt, $\mathrm{lca}(a,b) \in Z$, da der Knoten von $\ell$ erreichbar ist und $a$ erreichen kann. (↯)

Das zeigt dass $\mathrm{lca}(a,b) \in Z$. Damit gilt dies auch für Knoten zwischen $\mathrm{lca}(a,b)$ und $a$ bzw. $b$, da diese von einem Knoten aus $Z$ erreichbar sind, und einen Knoten aus $Z$ erreichen können.

# Starker Zusammenhang

- $x \in Z$, da er von $a$ erreichbar ist und $b$ erreichen kann.
- $\ell := \mathrm{lca}(a,x) \in Z$, da wir sonst ein kleineres Gegenbeispiel hätten.
- Falls $\mathrm{lca}(a,b)$ Vorgänger von $\ell$ folgt, dass $\ell$ und $b$ ein kleineres Gegenbeispiel bilden (beachte $\ell \neq a$). (↯)
- Falls $\ell$ Vorgänger von $\mathrm{lca}(a,b)$ folgt, $\mathrm{lca}(a,b) \in Z$, da der Knoten von $\ell$ erreichbar ist und $a$ erreichen kann. (↯)

Das zeigt dass $\mathrm{lca}(a,b) \in Z$. Damit gilt dies auch für Knoten zwischen $\mathrm{lca}(a,b)$ und $a$ bzw. $b$, da diese von einem Knoten aus $Z$ erreichbar sind, und einen Knoten aus $Z$ erreichen können.

# Starker Zusammenhang

Wir müssen nur Kanten aus dem DFS-Baum entfernen um die starken Zhks zu bestimmen. Kindknoten einer entfernten Knoten ist Wurzel von Zhk.

**Welche Kanten werden entfernt?**

**Wie bestimmen wir Zhks?**

# Starker Zusammenhang

Wir müssen nur Kanten aus dem DFS-Baum entfernen um die starken Zhks zu bestimmen. Kindknoten einer entfernten Knoten ist Wurzel von Zhk.

**Welche Kanten werden entfernt?**

**Wie bestimmen wir Zhks?**

# Starker Zusammenhang

Wir müssen nur Kanten aus dem DFS-Baum entfernen um die starken Zhks zu bestimmen. Kindknoten einer entfernten Knoten ist Wurzel von Zhk.

**Welche Kanten werden entfernt?**

**Wie bestimmen wir Zhks?**

- Immer wenn wir einen rekursiven Aufruf starten (einen Teilbaum betreten) legen wir den zugehörigen Knoten auf einen (separaten) Stack.
- Wenn wir einen Teilbaum (mit Wurzel $v$) verlassen und die zugehörige Kante, die in den Teilbaum führt entfernen wollen, gehören alle Knoten, die nach $v$ auf dem Stack liegen zu Zhk. (nur dann werden sie entfernt).

# Starker Zusammenhang

Wir müssen nur Kanten aus dem DFS-Baum entfernen um die starken Zhks zu bestimmen. Kindknoten einer entfernten Knoten ist Wurzel von Zhk.

**Welche Kanten werden entfernt?**

**Wie bestimmen wir Zhks?**

- Immer wenn wir einen rekursiven Aufruf starten (einen Teilbaum betreten) legen wir den zugehörigen Knoten auf einen (separaten) Stack.
- Wenn wir einen Teilbaum (mit Wurzel $v$) verlassen und die zugehörige Kante, die in den Teilbaum führt entfernen wollen, gehören alle Knoten, die nach $v$ auf dem Stack liegen zu Zhk. (nur dann werden sie entfernt).

# Starker Zusammenhang

Wir müssen nur Kanten aus dem DFS-Baum entfernen um die starken Zhks zu bestimmen. Kindknoten einer entfernten Knoten ist Wurzel von Zhk.

**Welche Kanten werden entfernt?**

**Wie bestimmen wir Zhks?**

- Immer wenn wir einen rekursiven Aufruf starten (einen Teilbaum betreten) legen wir den zugehörigen Knoten auf einen (separaten) Stack.
- Wenn wir einen Teilbaum (mit Wurzel $v$) verlassen und die zugehörige Kante, die in den Teilbaum führt entfernen wollen, gehören alle Knoten, die nach $v$ auf dem Stack liegen zu Zhk. (nur dann werden sie entfernt).

# Starker Zusammenhang

```
1 dfsTarjan()
2     dfsCount = 1;
3     foreach v ∈ V
4         v->state  = initial;
5         v->parent = NULL;
6     foreach (s ∈ V)
7         if s->state == initial
8             dfsVisitTarjan(s)
```

# Starker Zusammenhang

```
1  Input: directed graph G = (V,E);
2  Output: prints connected components of G
3
4  dfsVisitTarjan(v)
5      v->state = active;
6      v->num   = dfsCount++;
7      S.push(v);
8      foreach (x ∈ N[v])
9          if (x->state == initial)
10             x->parent = v;
11             dfsVisitTarjan(x);
12     if (cut parent edge of v)
13         repeat
14             x = S.pop();
15             // add x to current component
16         until (v == x)
17         // print current component
```

# Starker Zusammenhang

In der rekursiven Variante wird normalerweise beim Start eines rekursiven Aufrufs $v$ auf den (impliziten) Stack gelegt, und bei Beendigung wieder entfernt.

Wir entfernen $v$ jetzt nur wenn die gesamte zugehörige Zusammenhangskomponente entfernt wird. D.h. wenn wir bei einem Vorgänger von $v$ die zugehörige Baumkante trennen.

**Invariante/Ziel**

Alle Knoten auf dem Stack können einen aktiven Knoten im Baum erreichen (deshalb ist die Zhk für diese Knoten noch nicht bestimmt).

# Starker Zusammenhang

In der rekursiven Variante wird normalerweise beim Start eines rekursiven Aufrufs $v$ auf den (impliziten) Stack gelegt, und bei Beendigung wieder entfernt.

Wir entfernen $v$ jetzt nur wenn die gesamte zugehörige Zusammenhangskomponente entfernt wird. D.h. wenn wir bei einem Vorgänger von $v$ die zugehörige Baumkante trennen.

**Invariante/Ziel**

Alle Knoten auf dem Stack können einen aktiven Knoten im Baum erreichen (deshalb ist die Zhk für diese Knoten noch nicht bestimmt).

# Starker Zusammenhang

In der rekursiven Variante wird normalerweise beim Start eines rekursiven Aufrufs $v$ auf den (impliziten) Stack gelegt, und bei Beendigung wieder entfernt.

Wir entfernen $v$ jetzt nur wenn die gesamte zugehörige Zusammenhangskomponente entfernt wird. D.h. wenn wir bei einem Vorgänger von $v$ die zugehörige Baumkante trennen.

**Invariante/Ziel**
Alle Knoten auf dem Stack können einen aktiven Knoten im Baum erreichen (deshalb ist die Zhk für diese Knoten noch nicht bestimmt).

# Starker Zusammenhang

Um zu entscheiden ob wir eine Kante schneiden müssen, berechnen wir für jeden Knoten den Wert v->low.

v->low
Minimale dfsNum eines Knotens, der in der gleichen Zhk wie v liegt und über eine Folge von Baumkanten gefolgt von maximal einer Kreuzkante oder einer Rückwärtskante erreichbar ist.

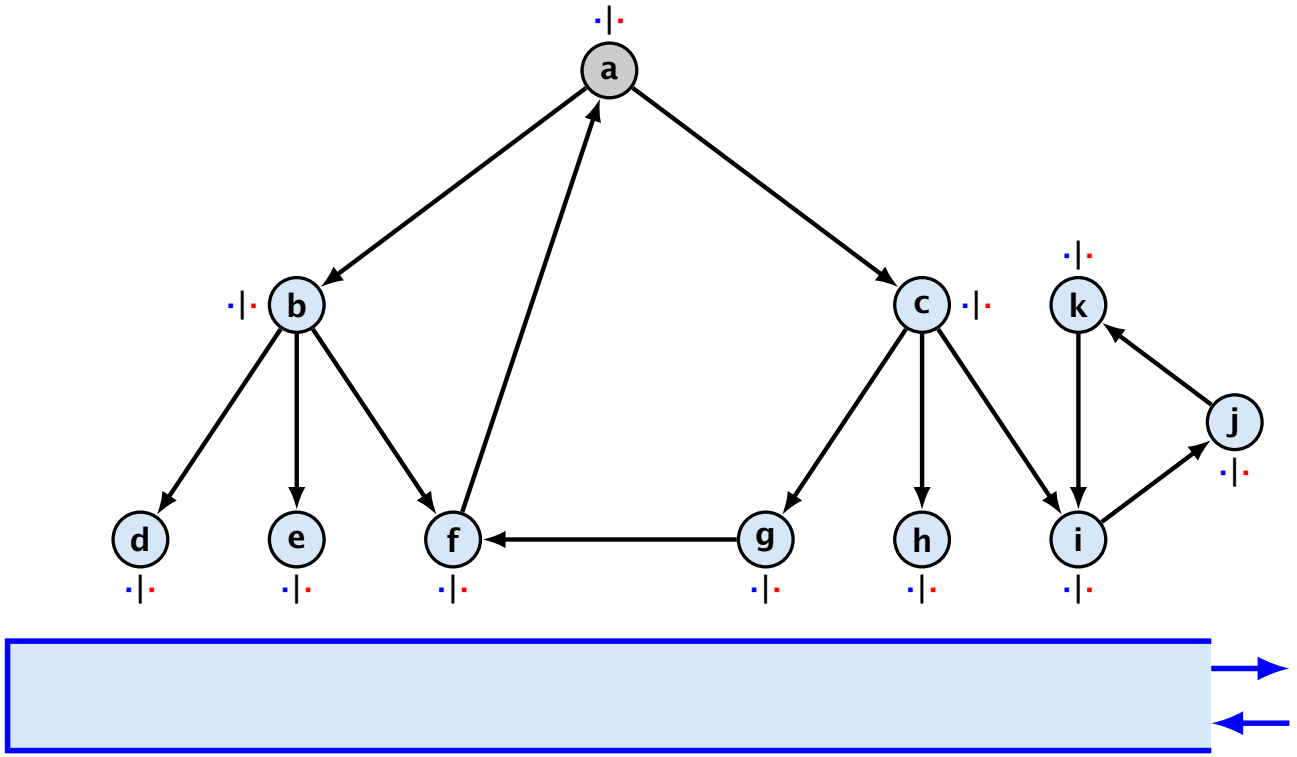
Wenn wir das können, dann sind Knoten für die v->low==v->num ist, die Knoten an denen wir schneiden müssen.

Irgendwie scheint diese Definition nicht hilfreich, da wir die Zusammenhangskomponenten ja gerade berechnen wollen...

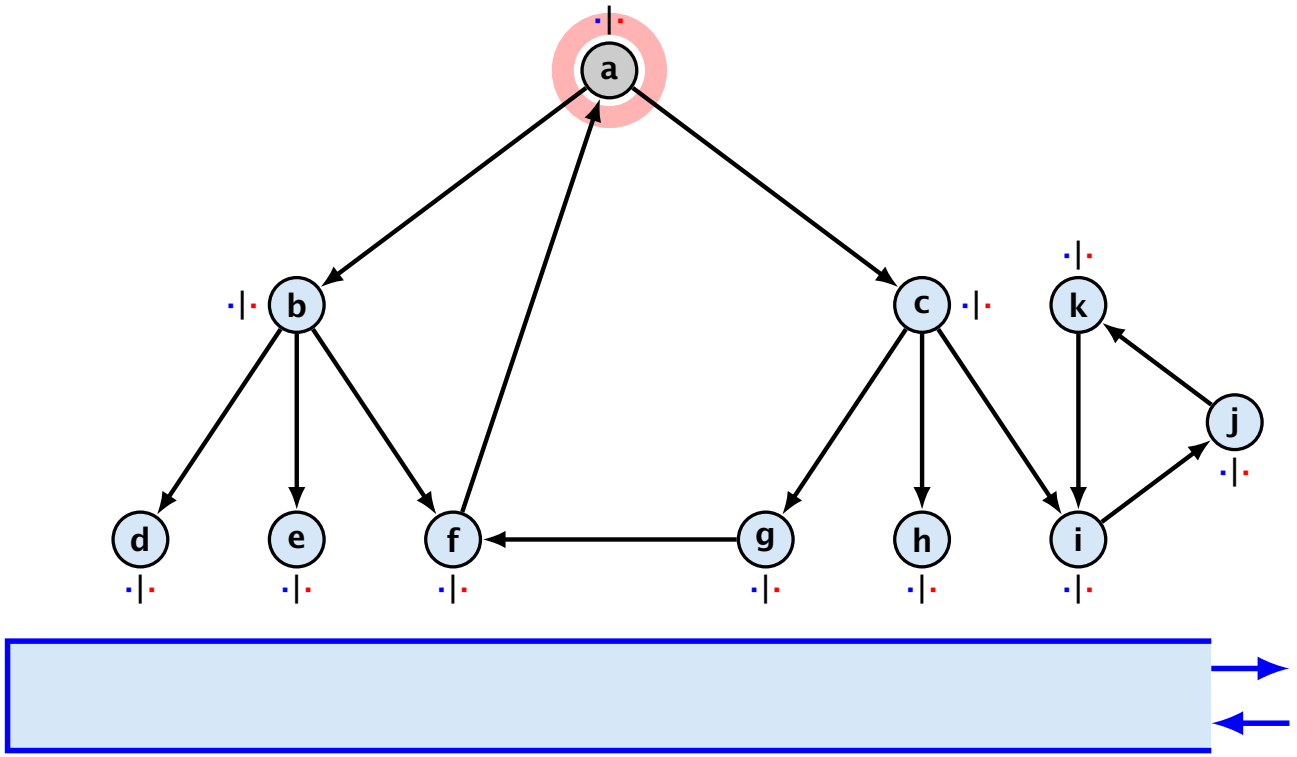
# Starker Zusammenhang

```
1  Input: directed graph G = (V,E);
2  Output: prints connected components of G
3
4  dfsVisitTarjan(v)
5      v->state = active;
6      v->num   = v->low = dfsCount++;
7      S.push(v); // setzt v->onStack
8      foreach (x ∈ N[v])
9          if (x->state == initial)
10             x->parent = v;
11             dfsVisitTarjan(x);
12             v->low = min(v->low,x->low)
13         else if (x->onStack)
14             v->low = min(v->low,x->num)
15     if (v->num == v->low)
16         repeat
17             x = S.pop(); // loescht x->onStack
18             // add x to current component
19         until (v == x)
20         // print current component
```

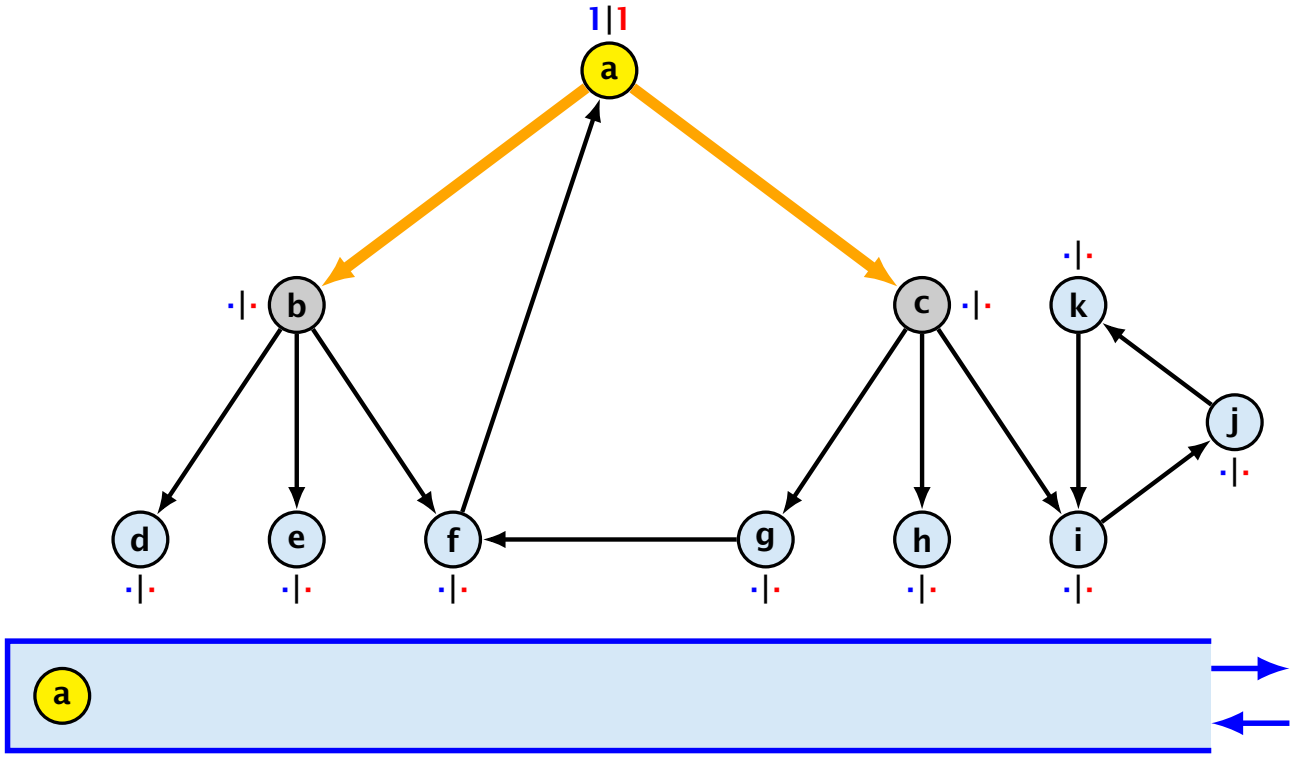
# Starker Zusammenhang



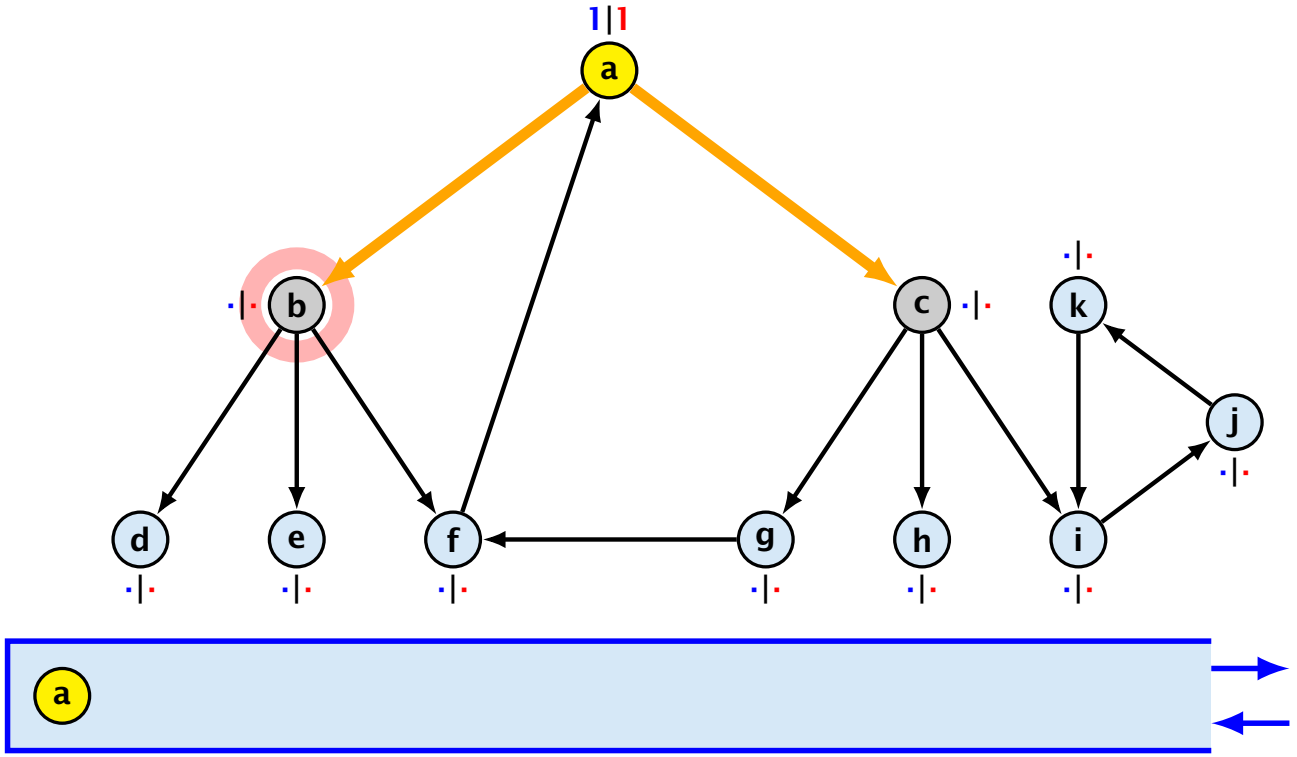
# Starker Zusammenhang



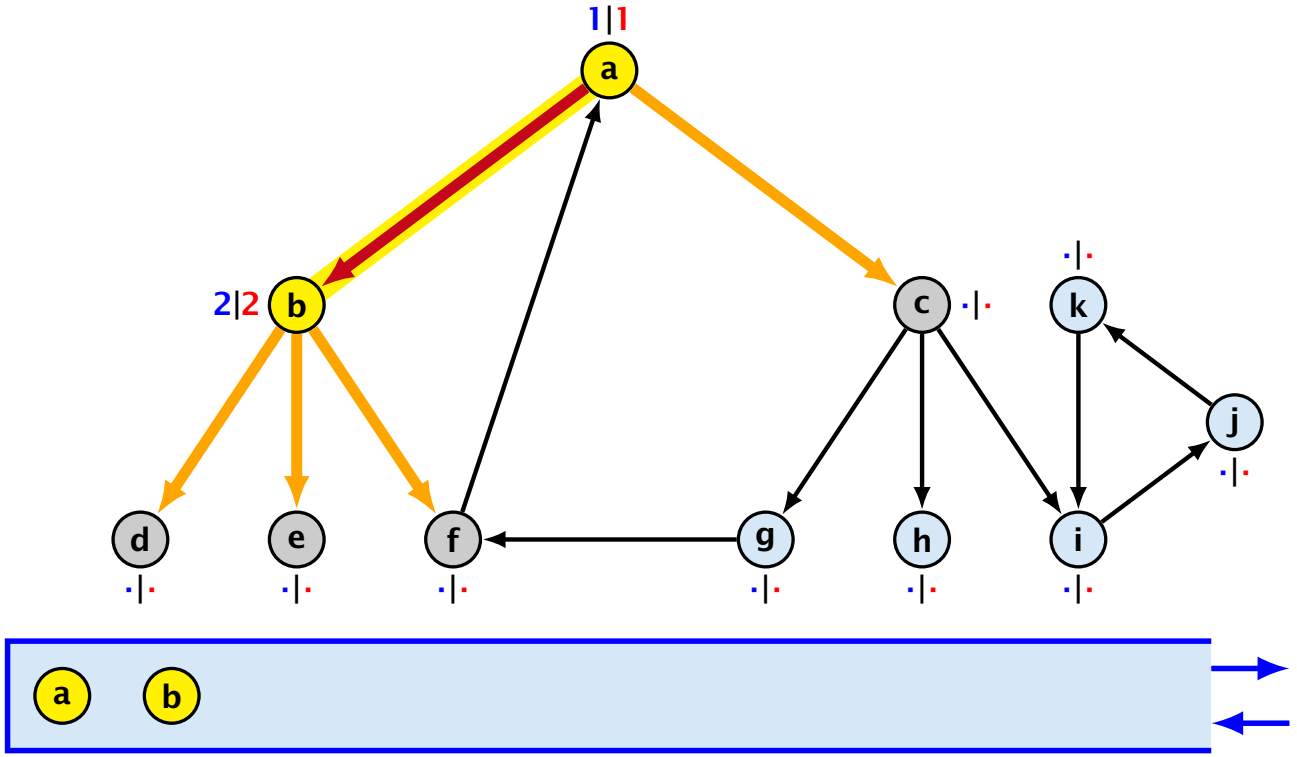
# Starker Zusammenhang



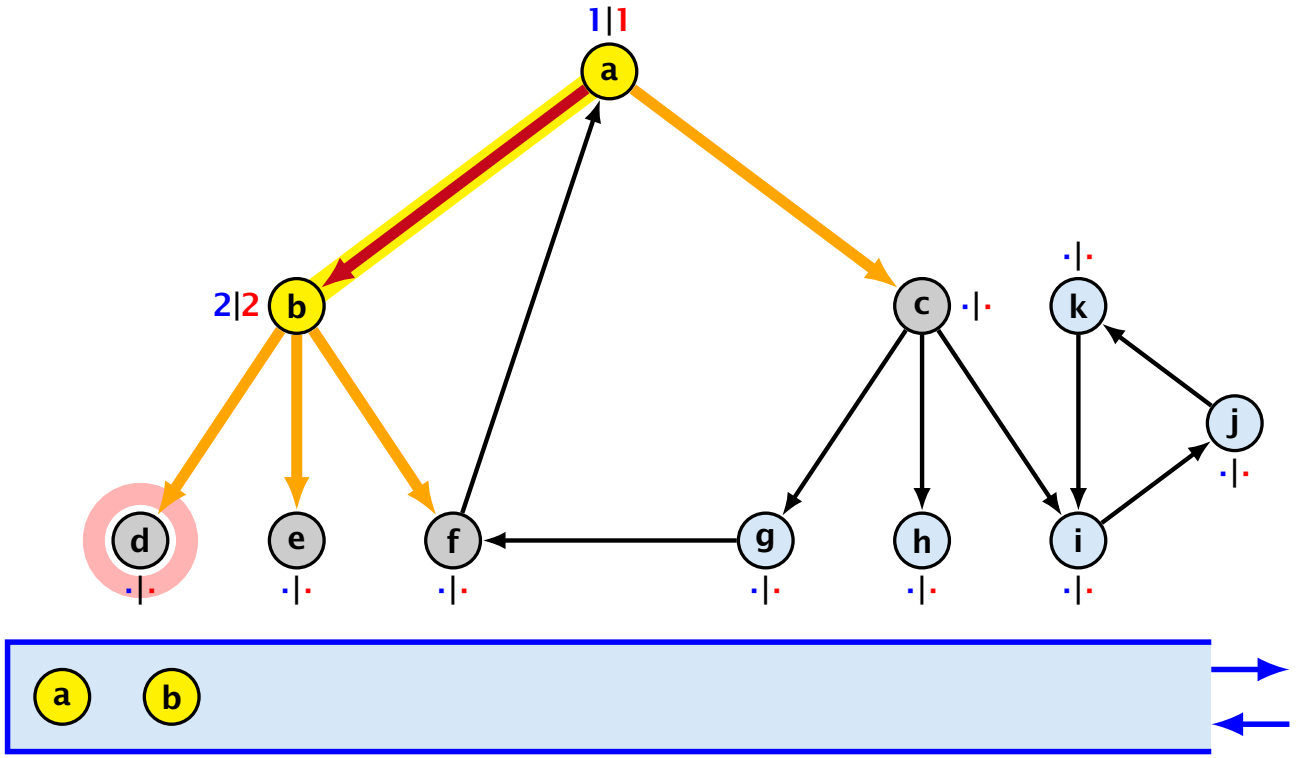
# Starker Zusammenhang



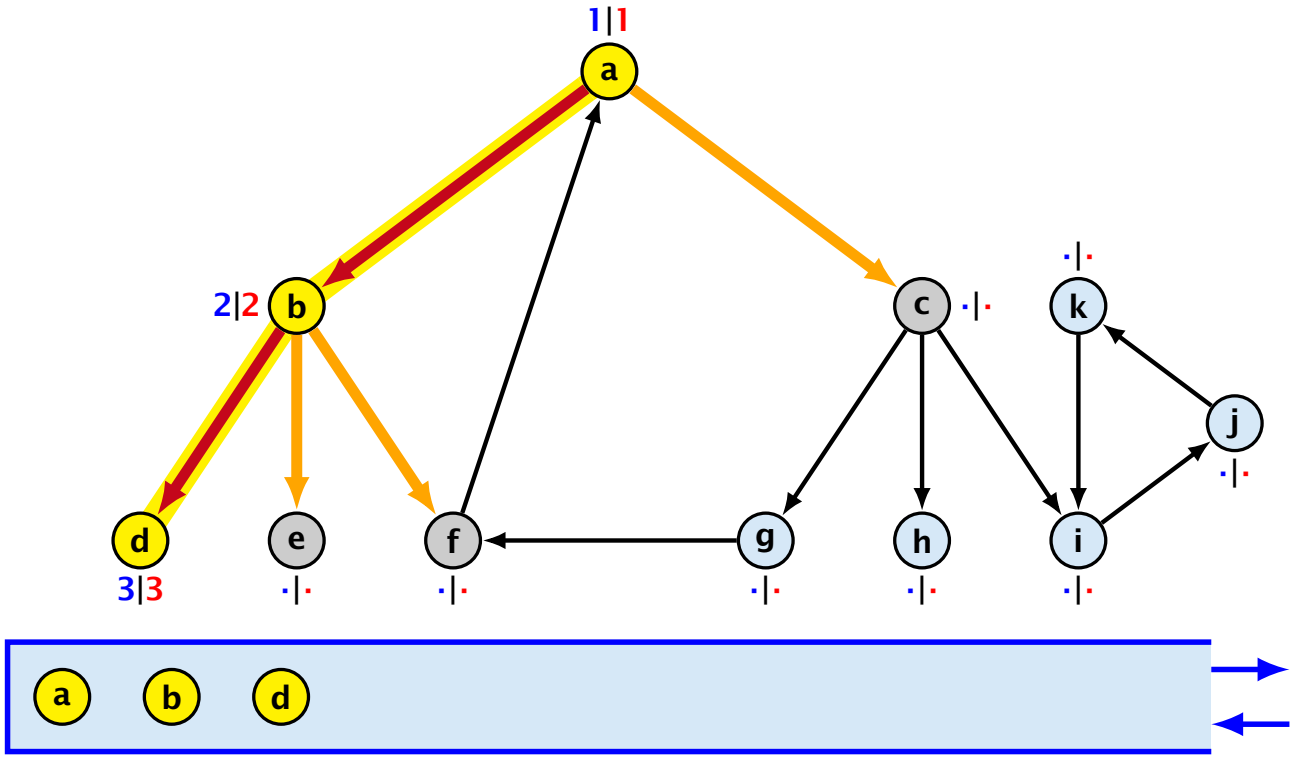
# Starker Zusammenhang



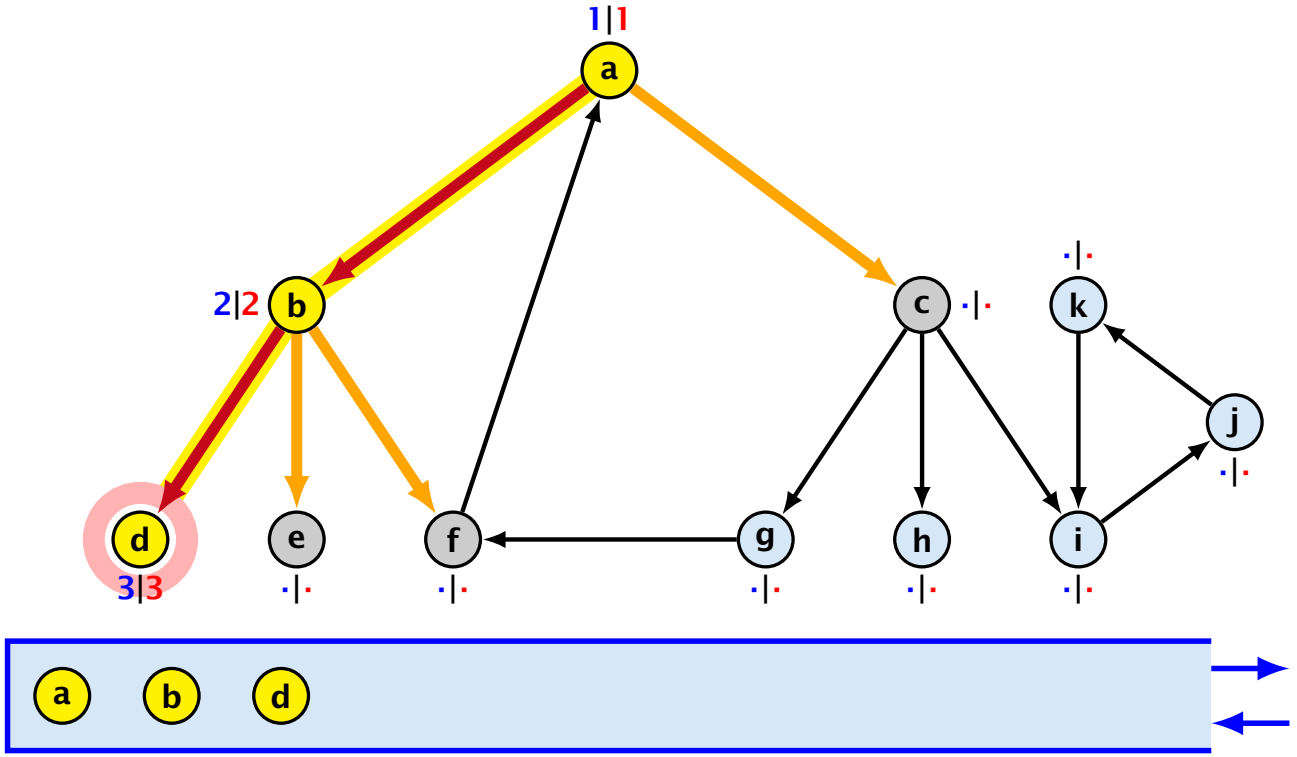
# Starker Zusammenhang



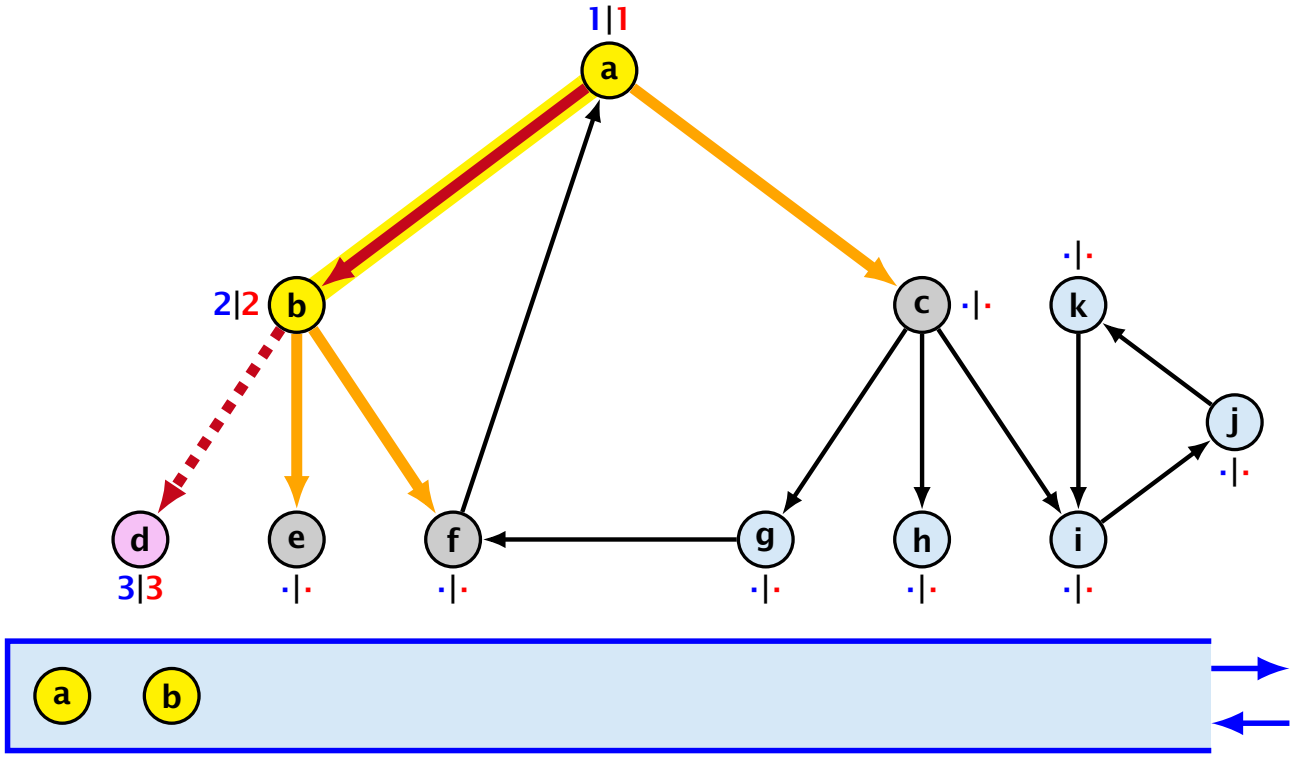
# Starker Zusammenhang



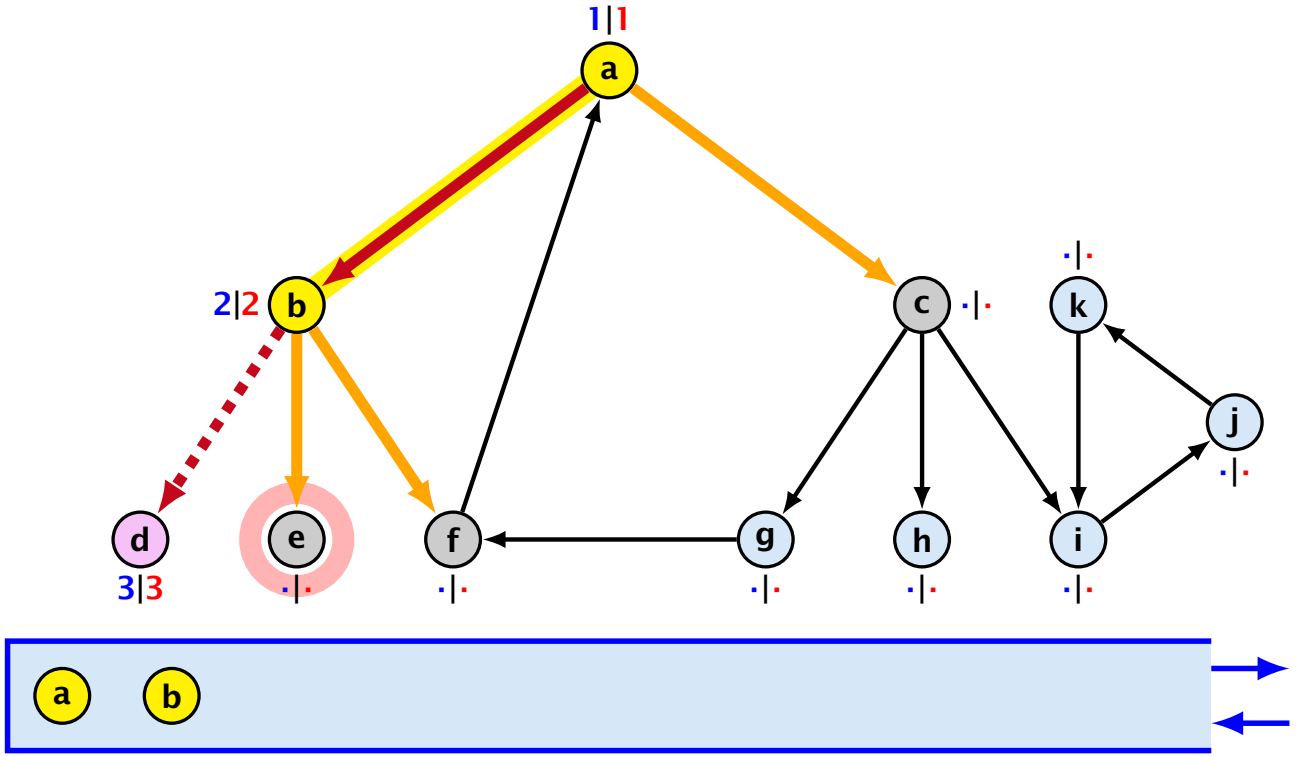
# Starker Zusammenhang



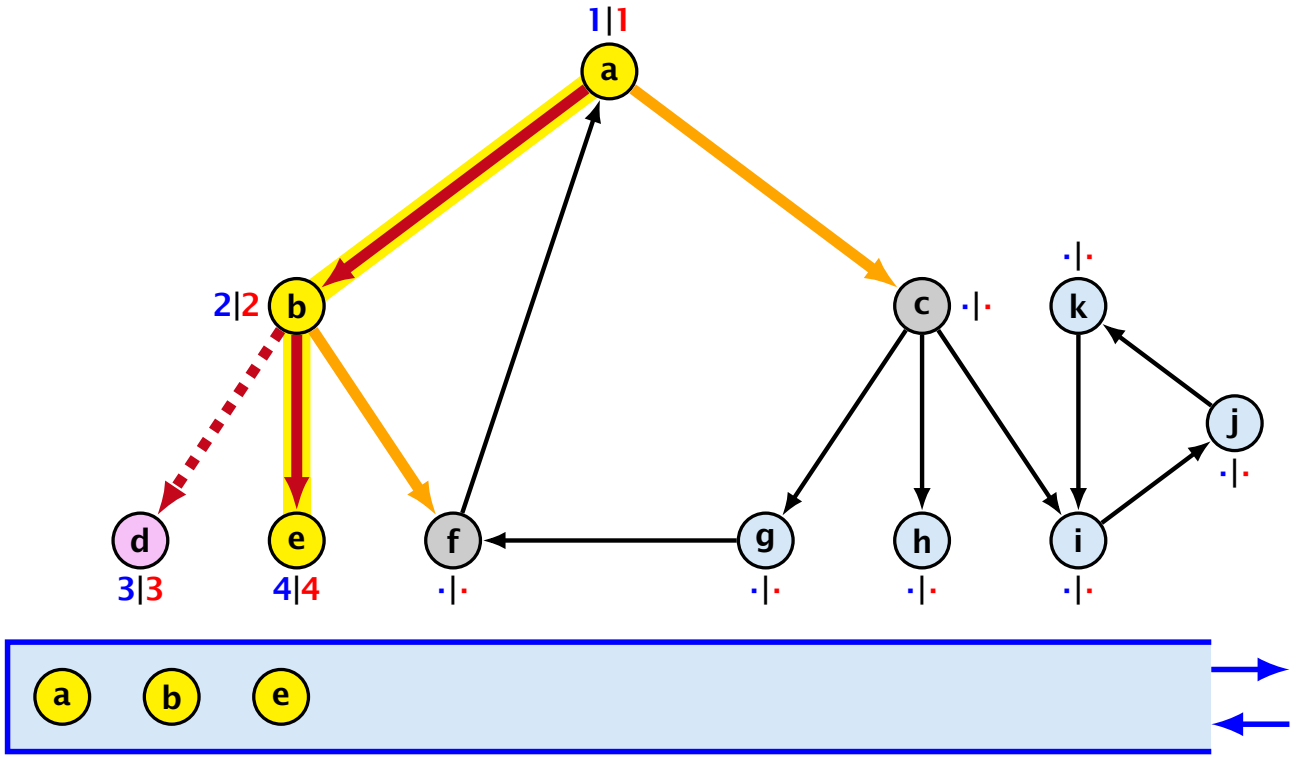
# Starker Zusammenhang



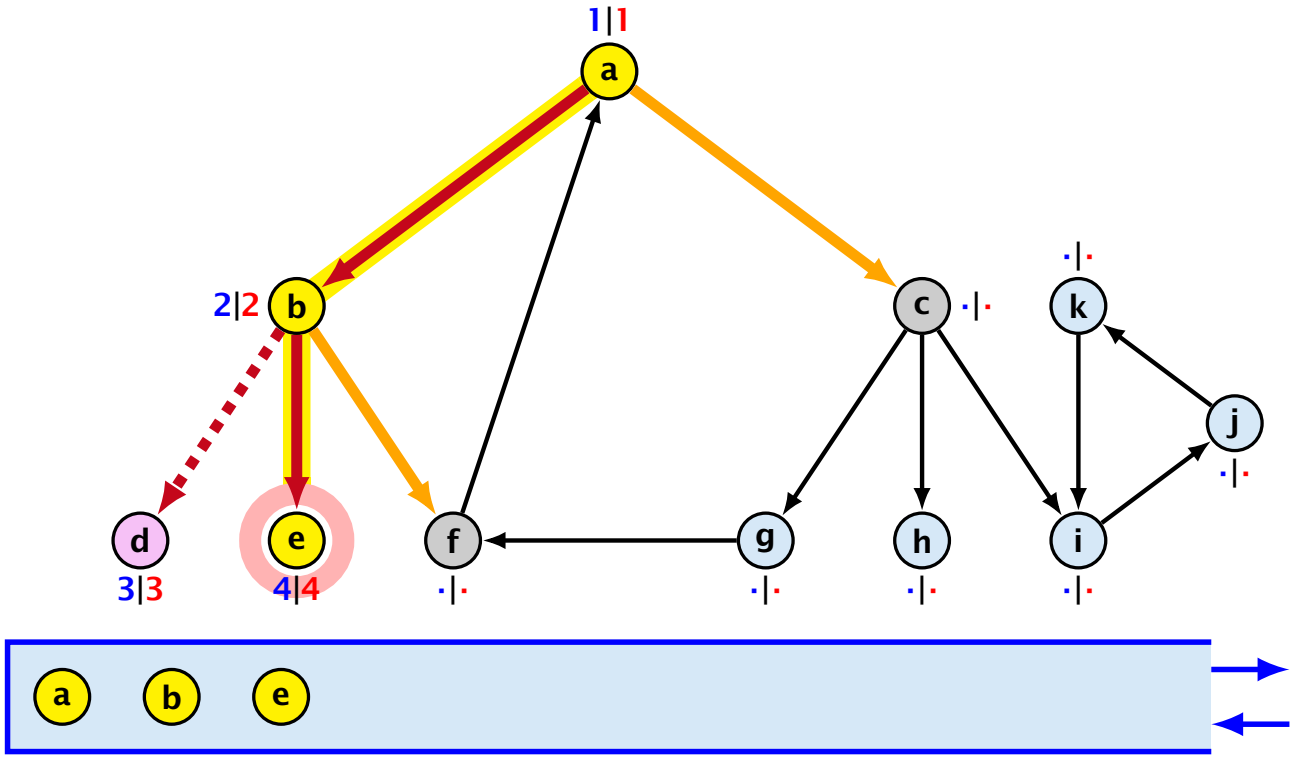
# Starker Zusammenhang



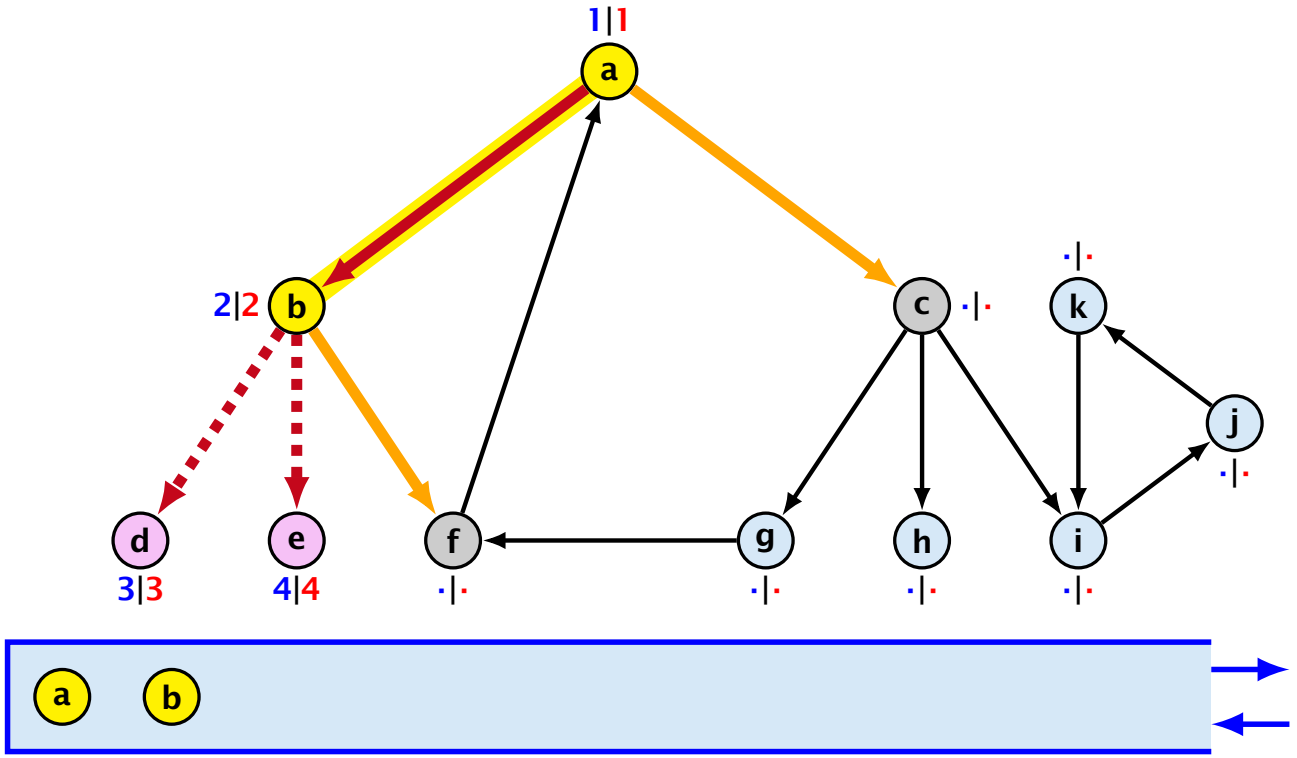
# Starker Zusammenhang



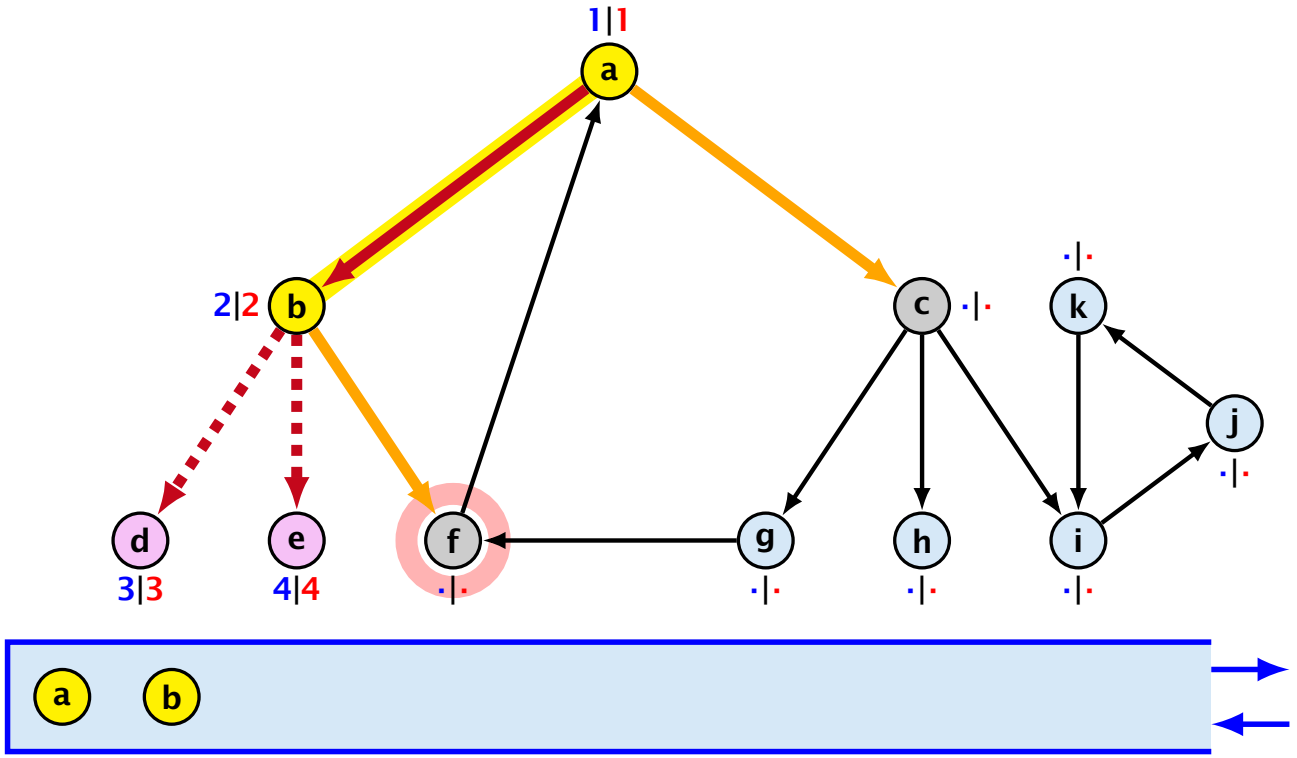
# Starker Zusammenhang



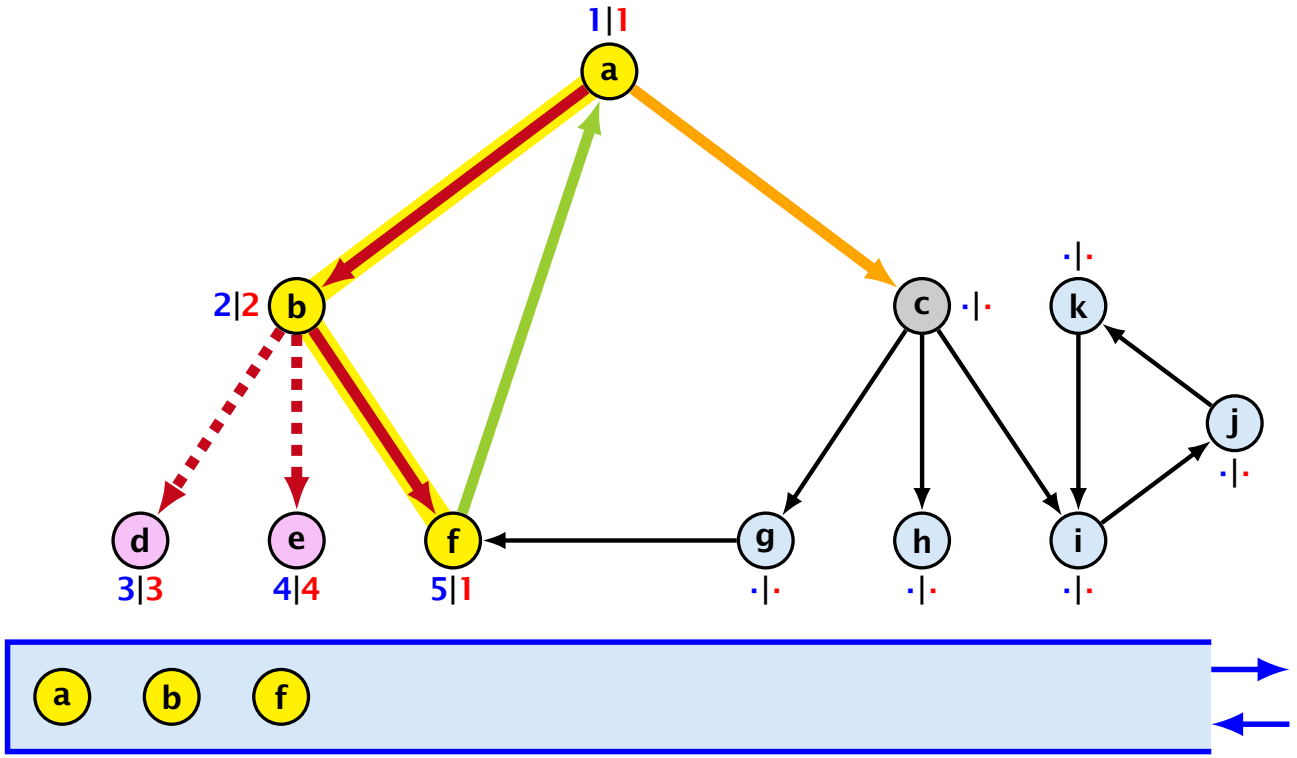
# Starker Zusammenhang



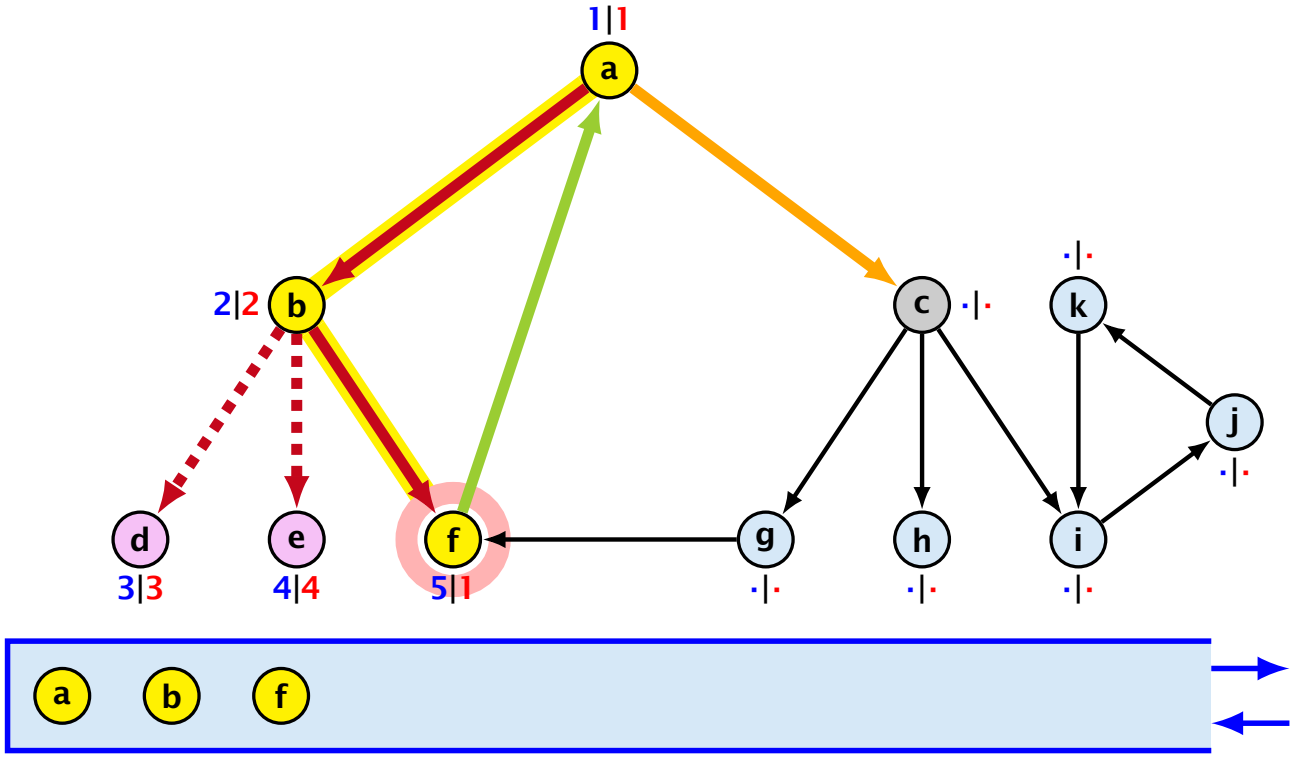
# Starker Zusammenhang



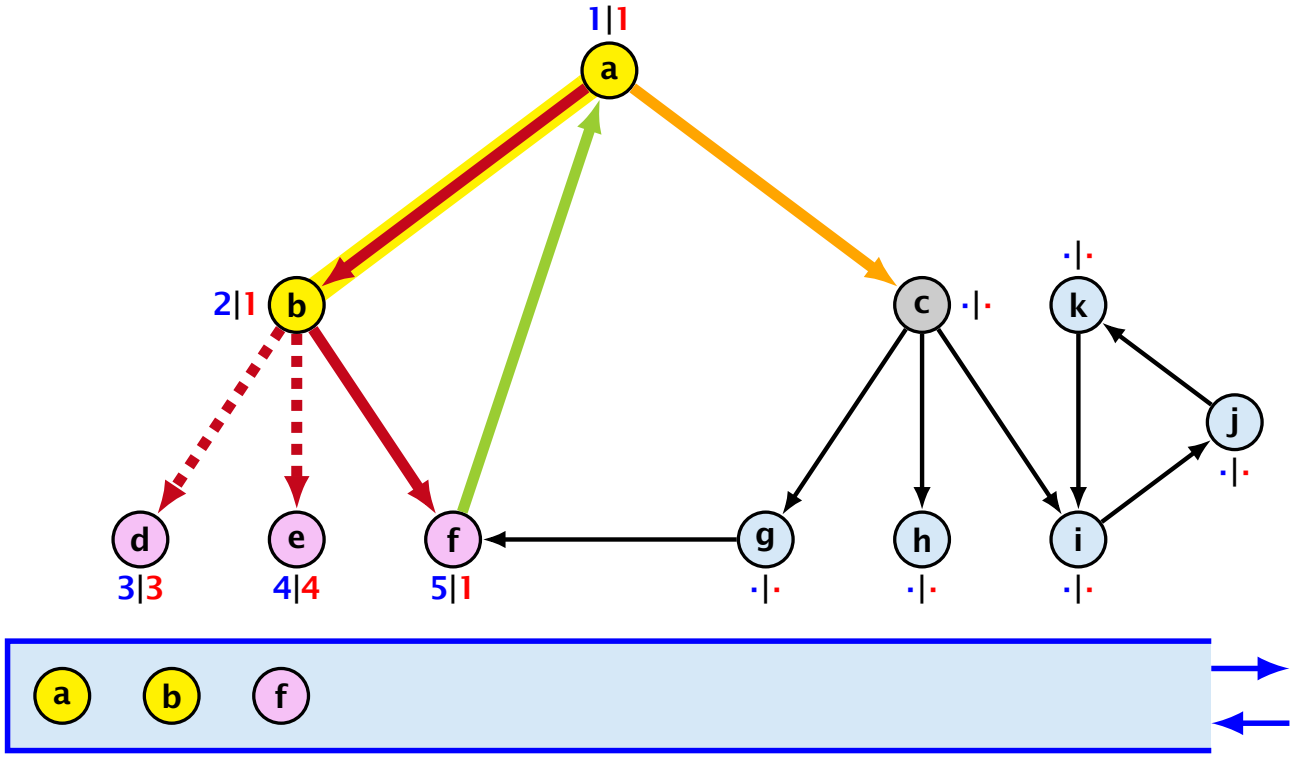
# Starker Zusammenhang



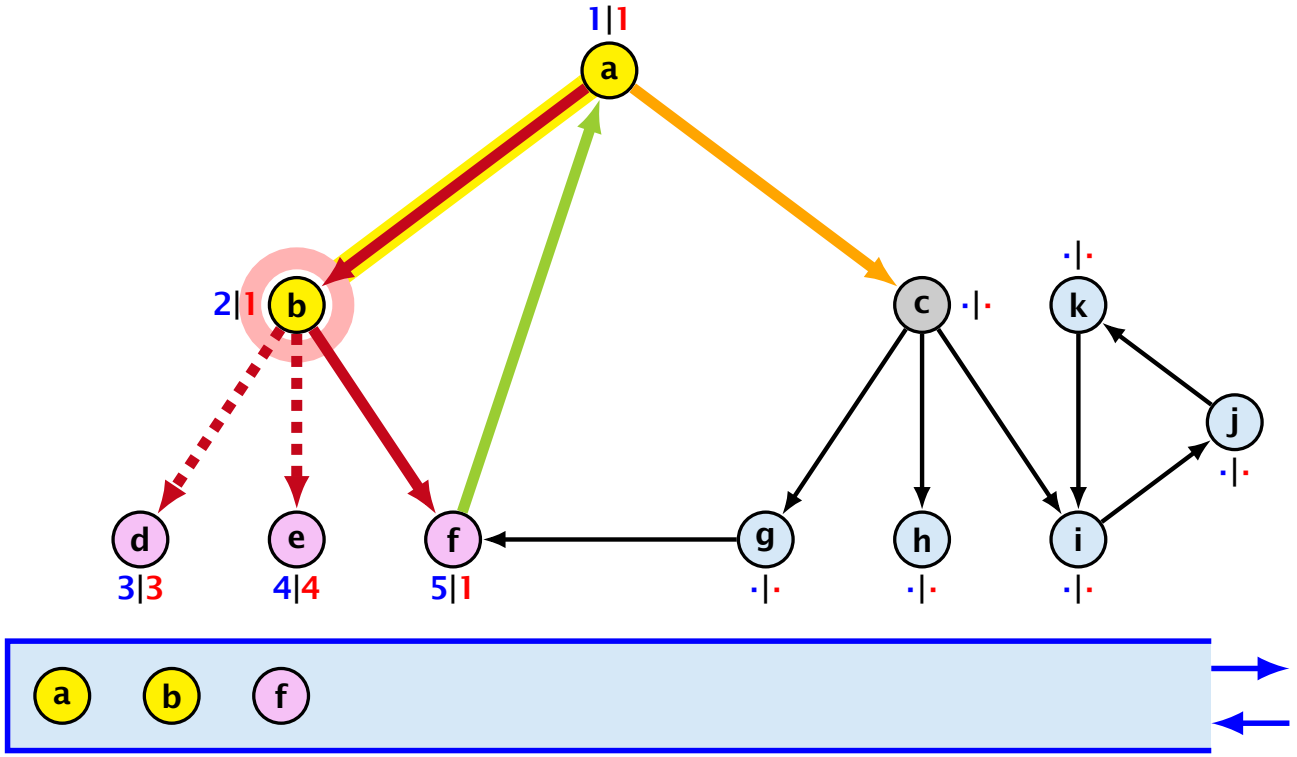
# Starker Zusammenhang



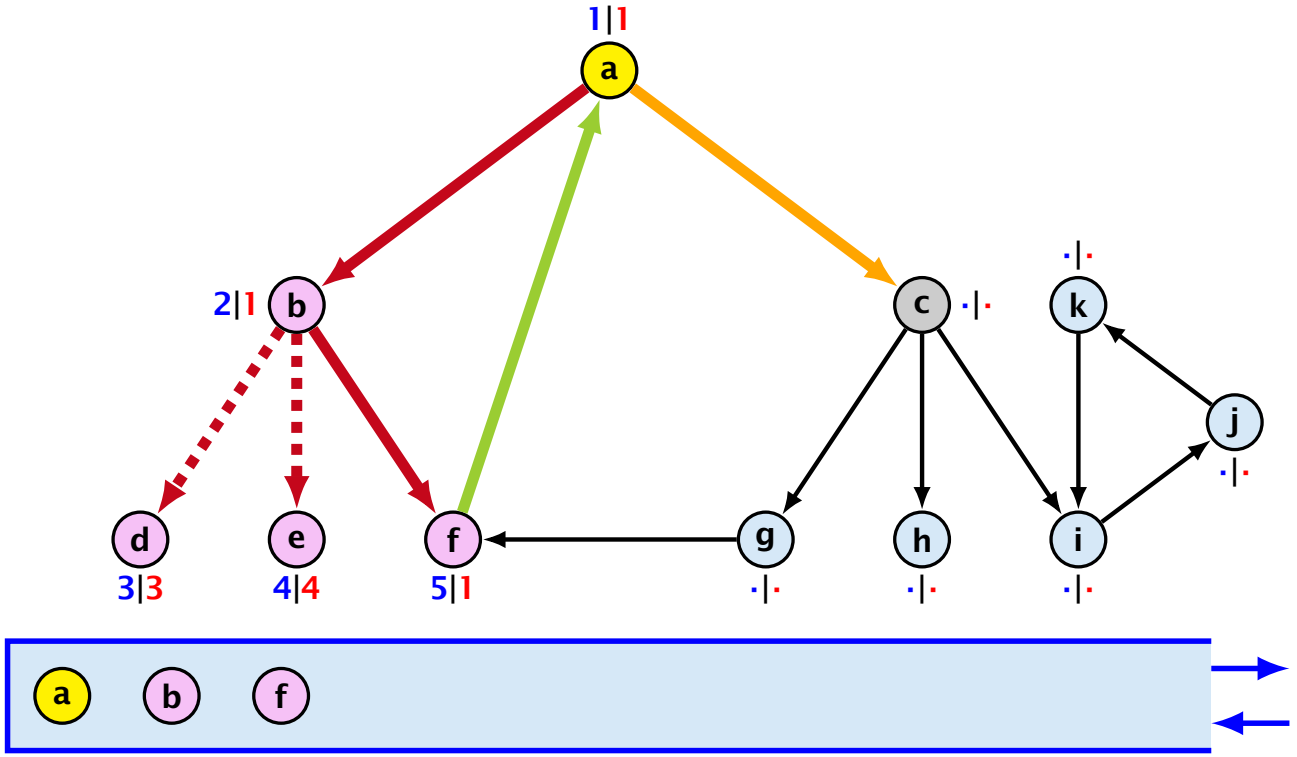
# Starker Zusammenhang



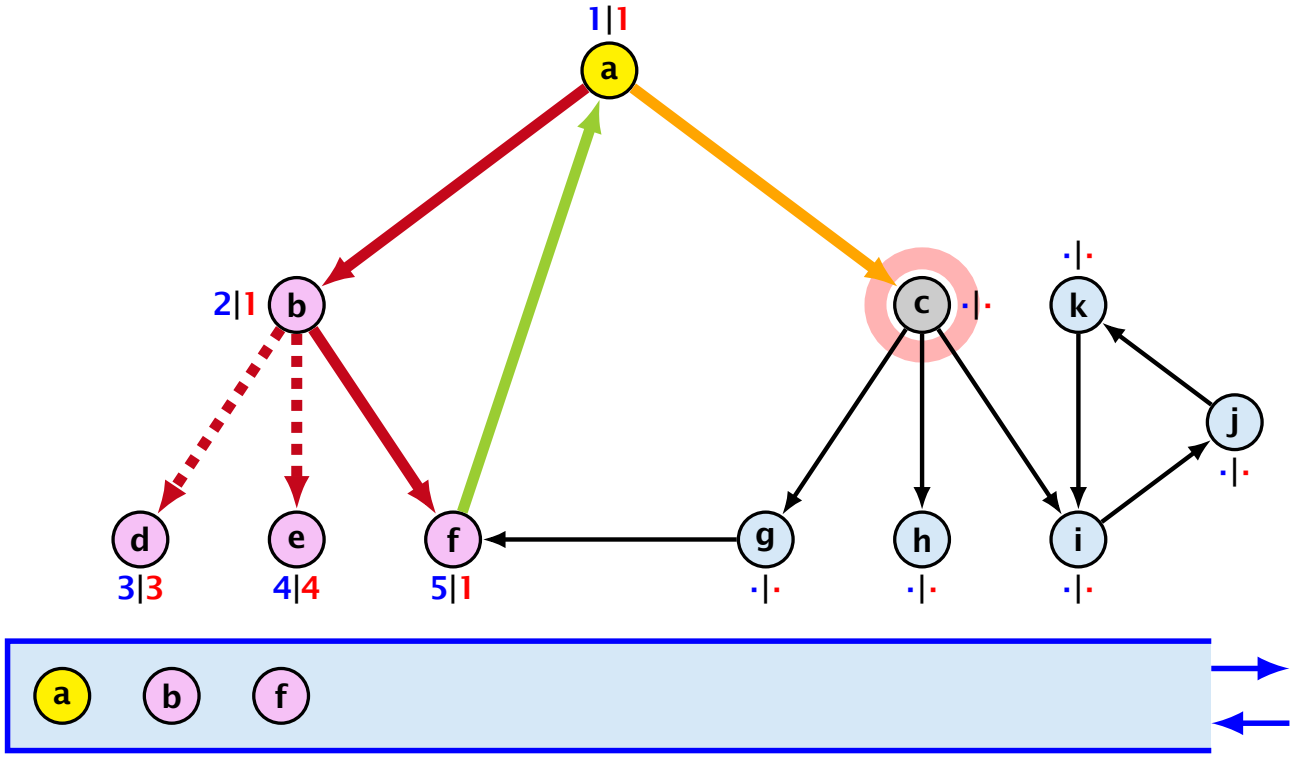
# Starker Zusammenhang



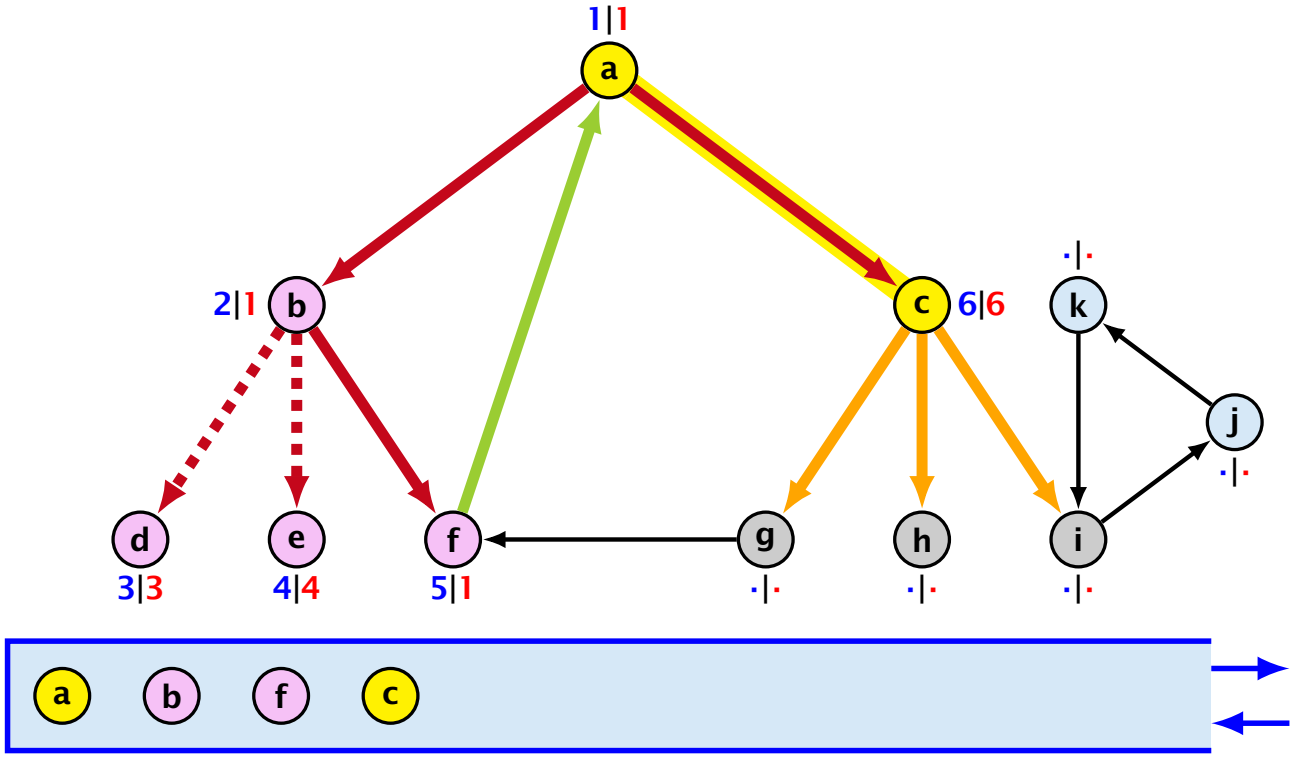
# Starker Zusammenhang



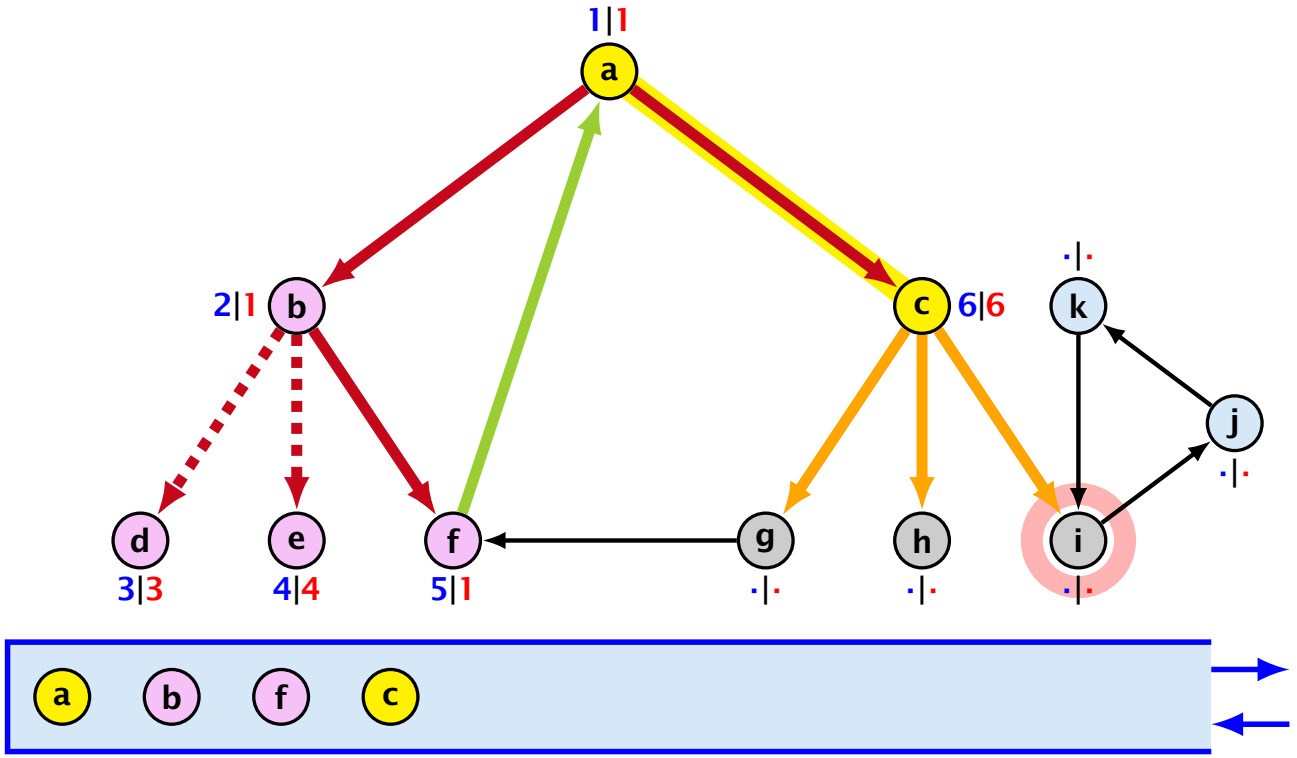
# Starker Zusammenhang



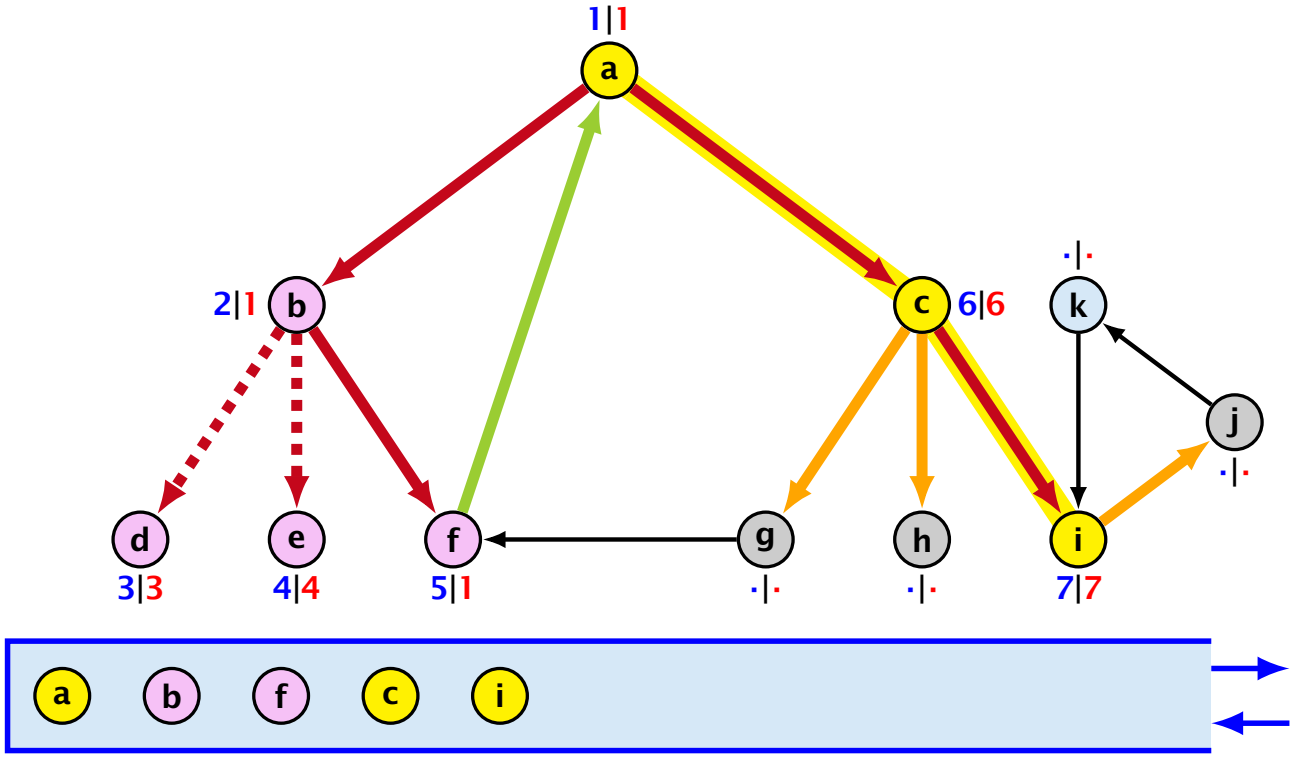
# Starker Zusammenhang



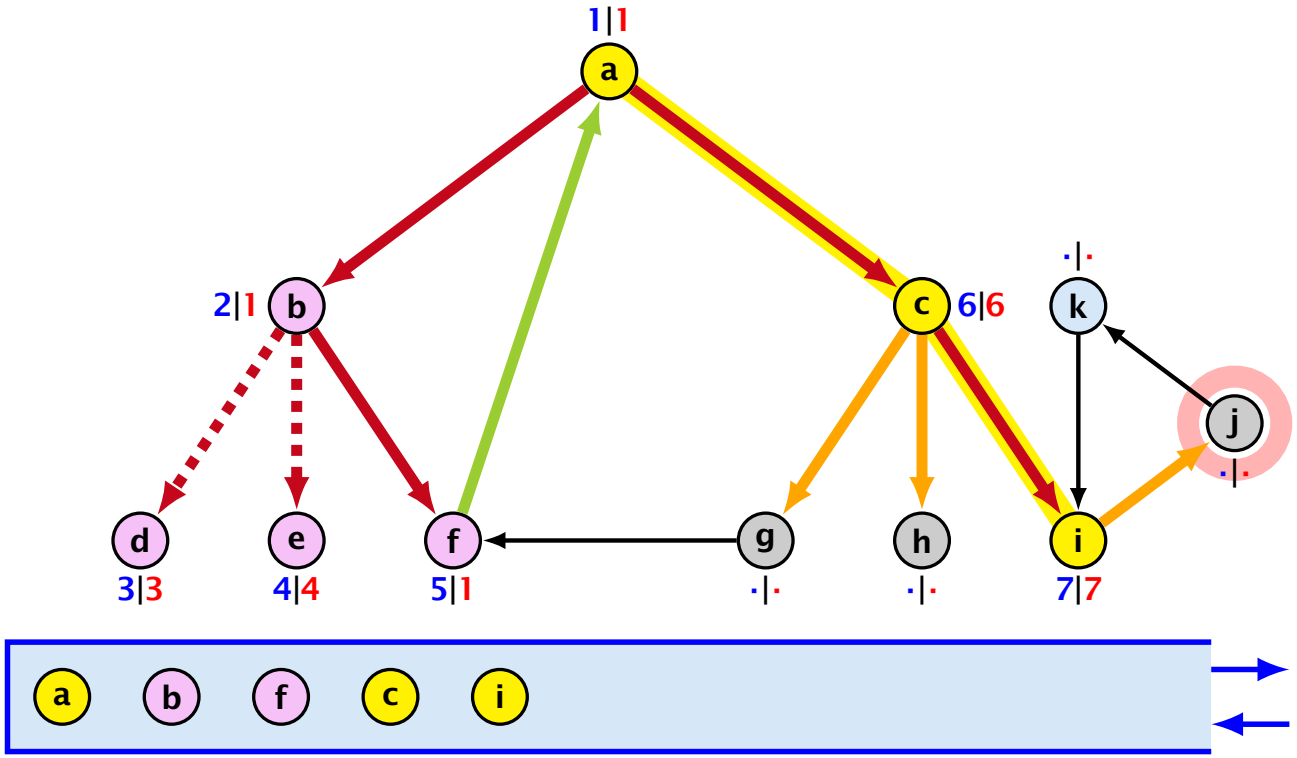
# Starker Zusammenhang



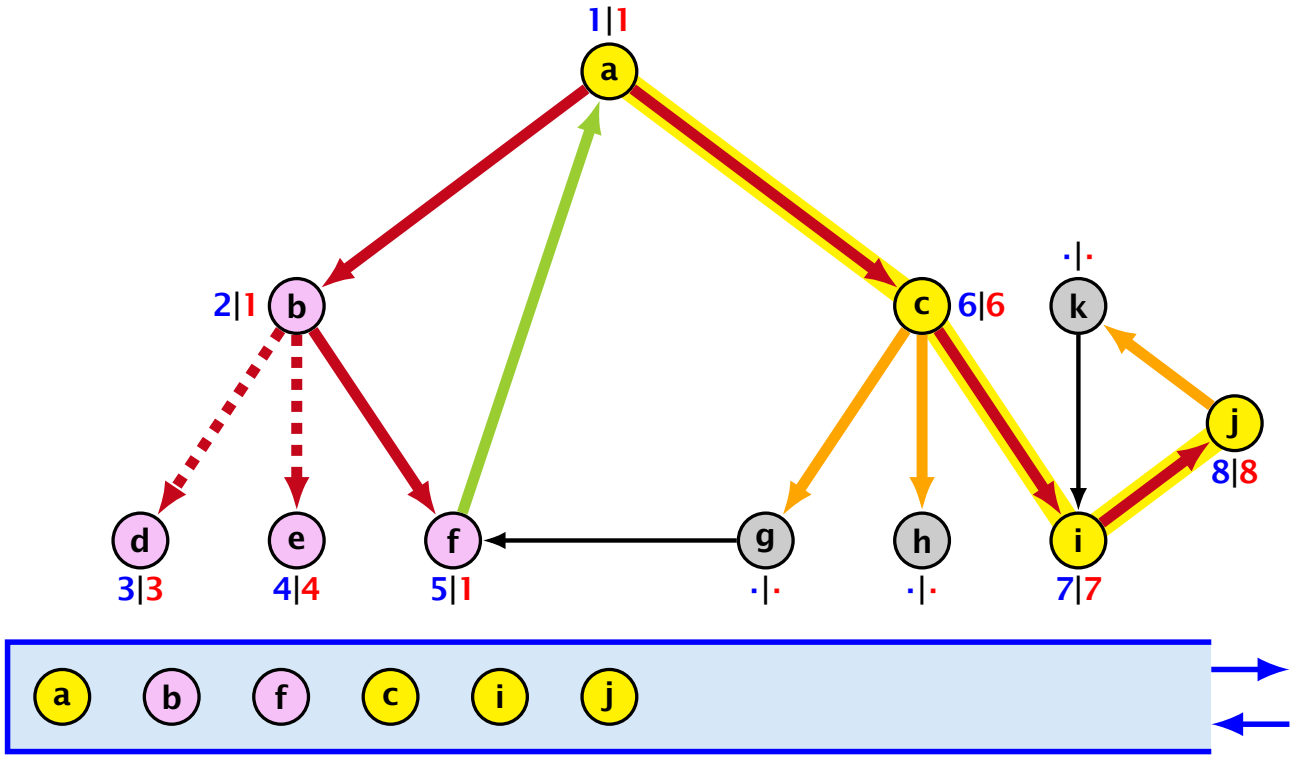
# Starker Zusammenhang



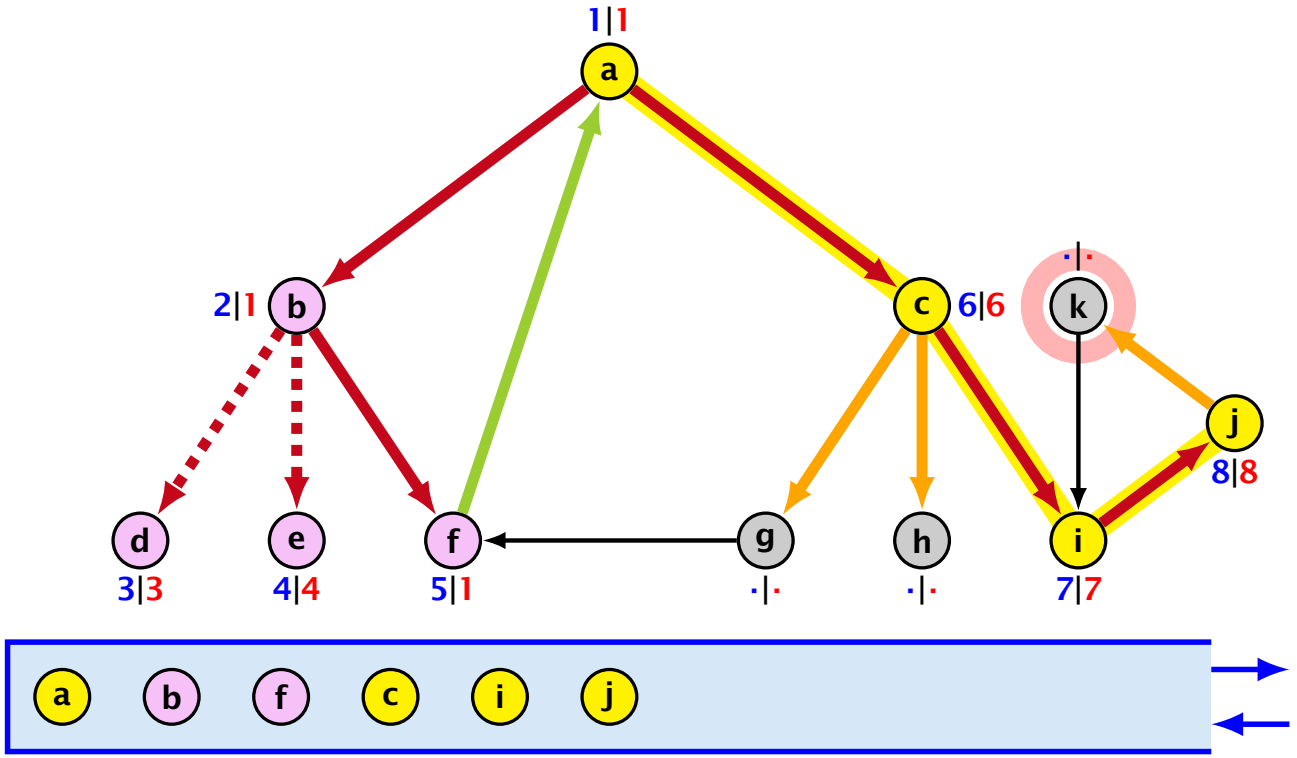
# Starker Zusammenhang



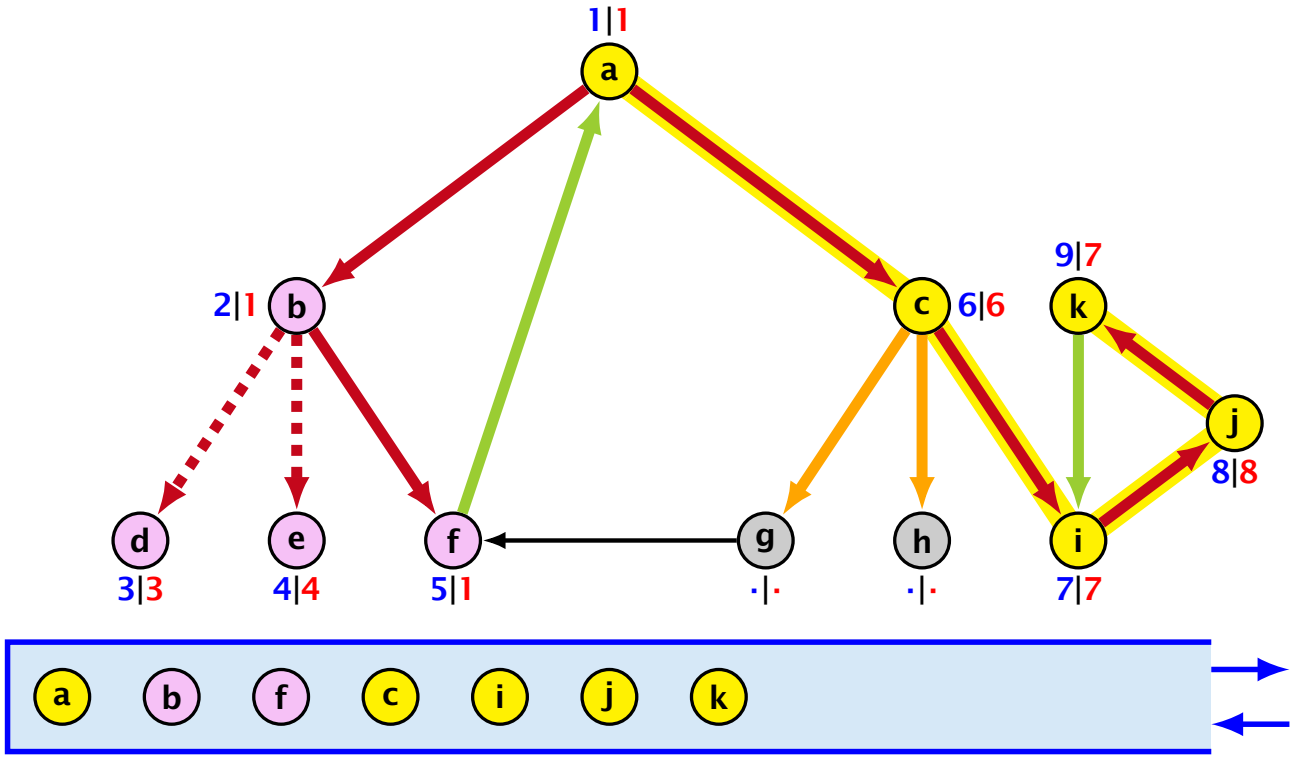
# Starker Zusammenhang



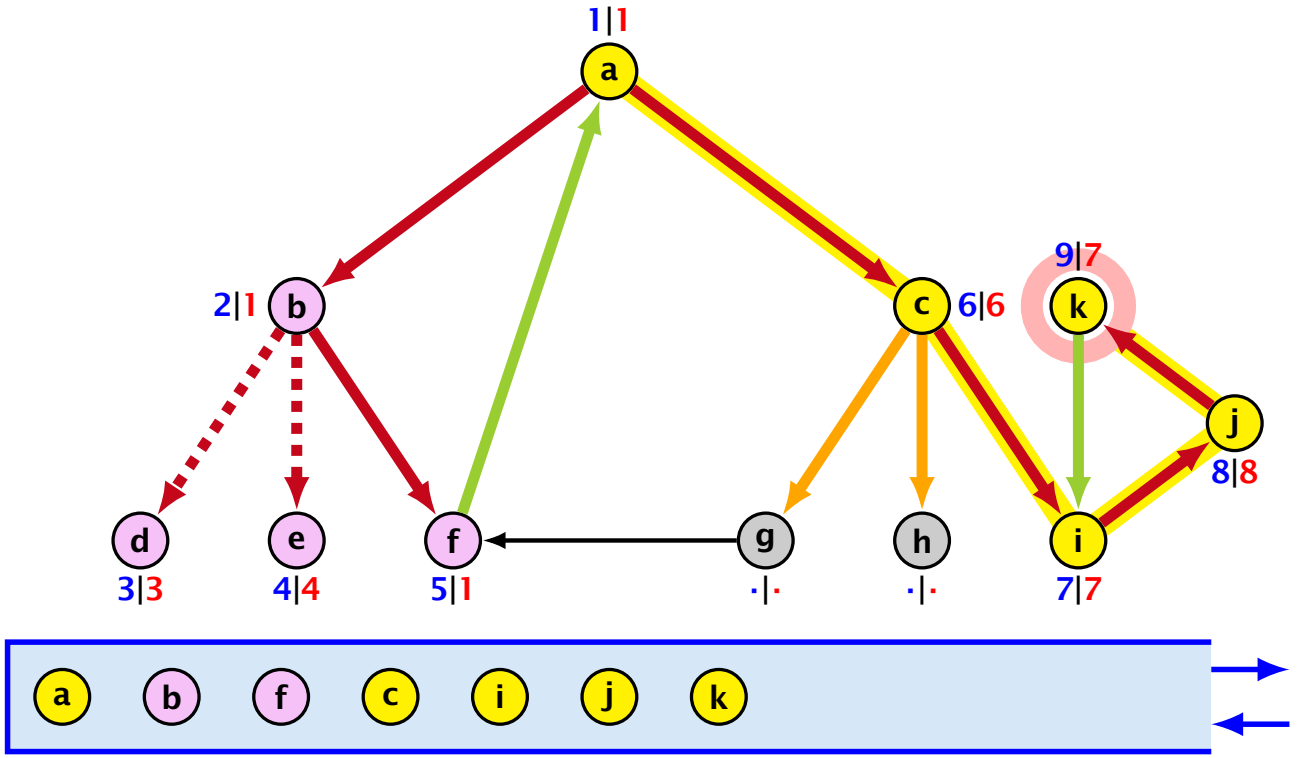
# Starker Zusammenhang



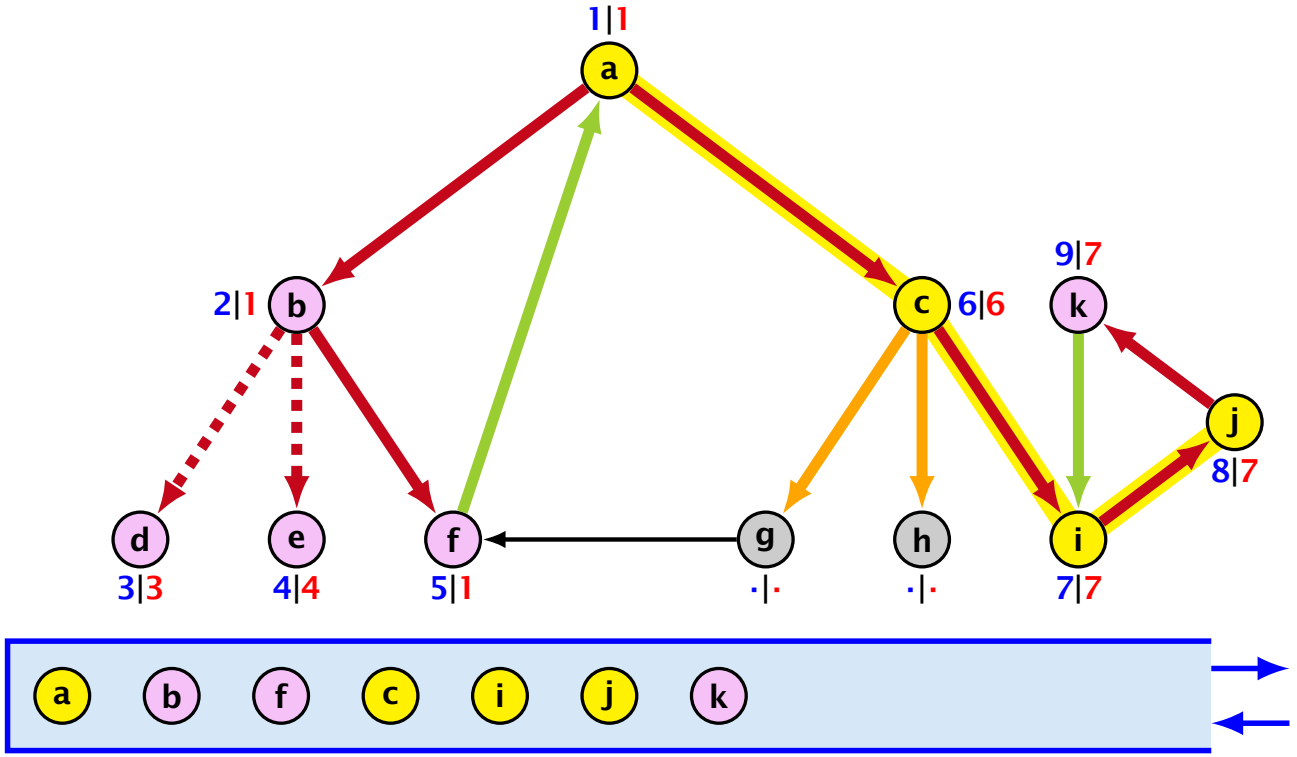
# Starker Zusammenhang



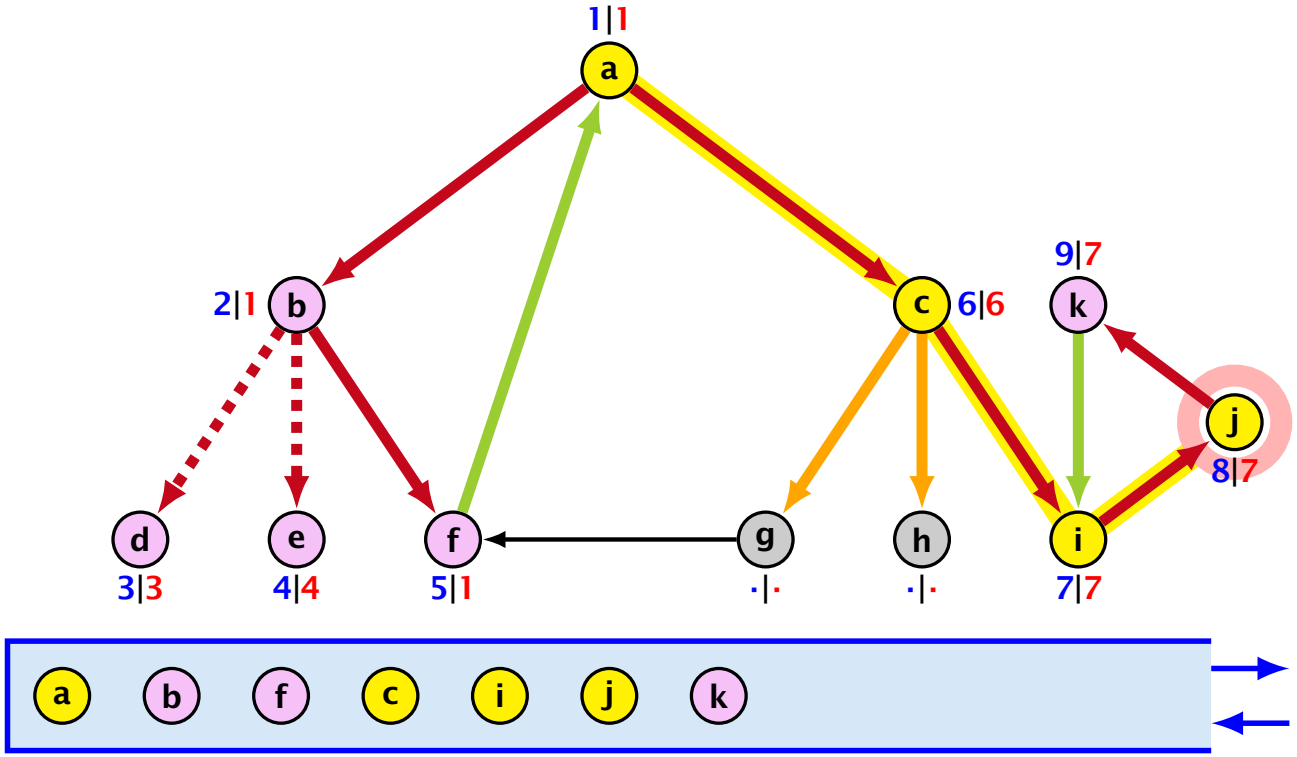
# Starker Zusammenhang



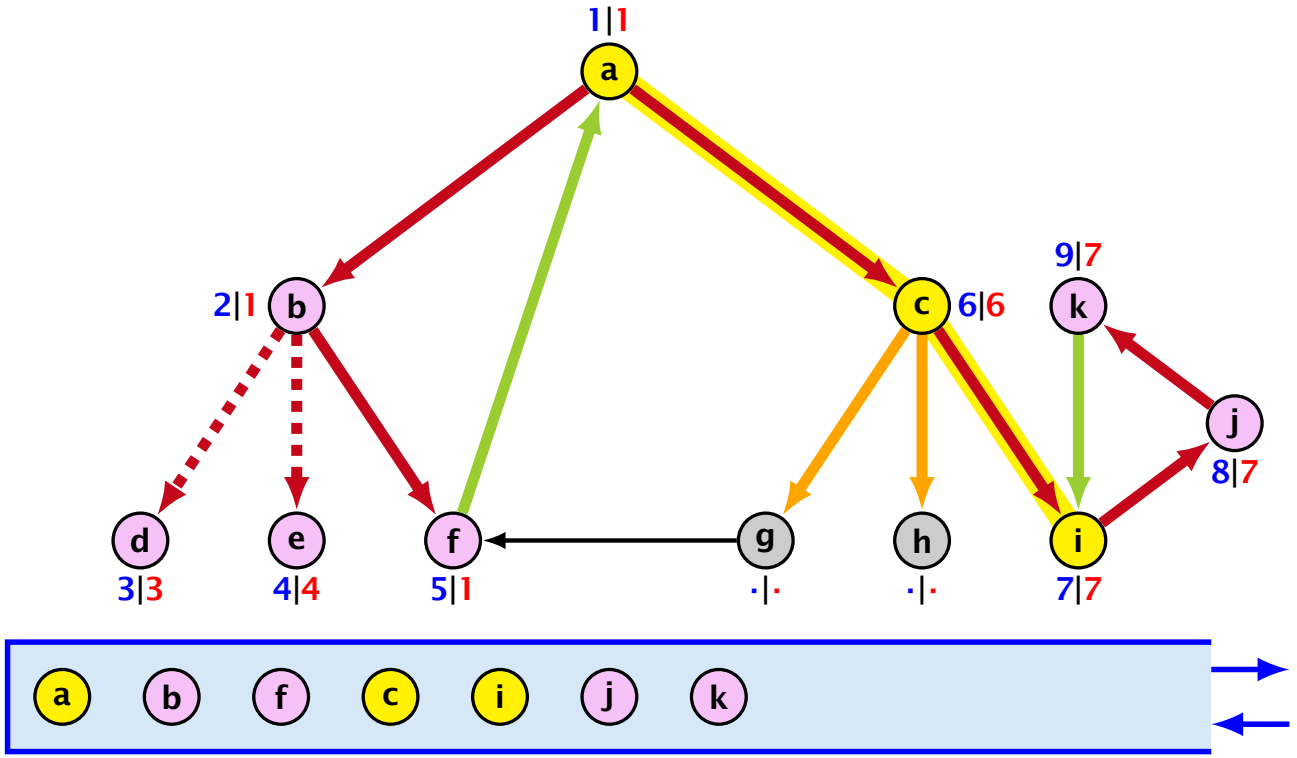
# Starker Zusammenhang



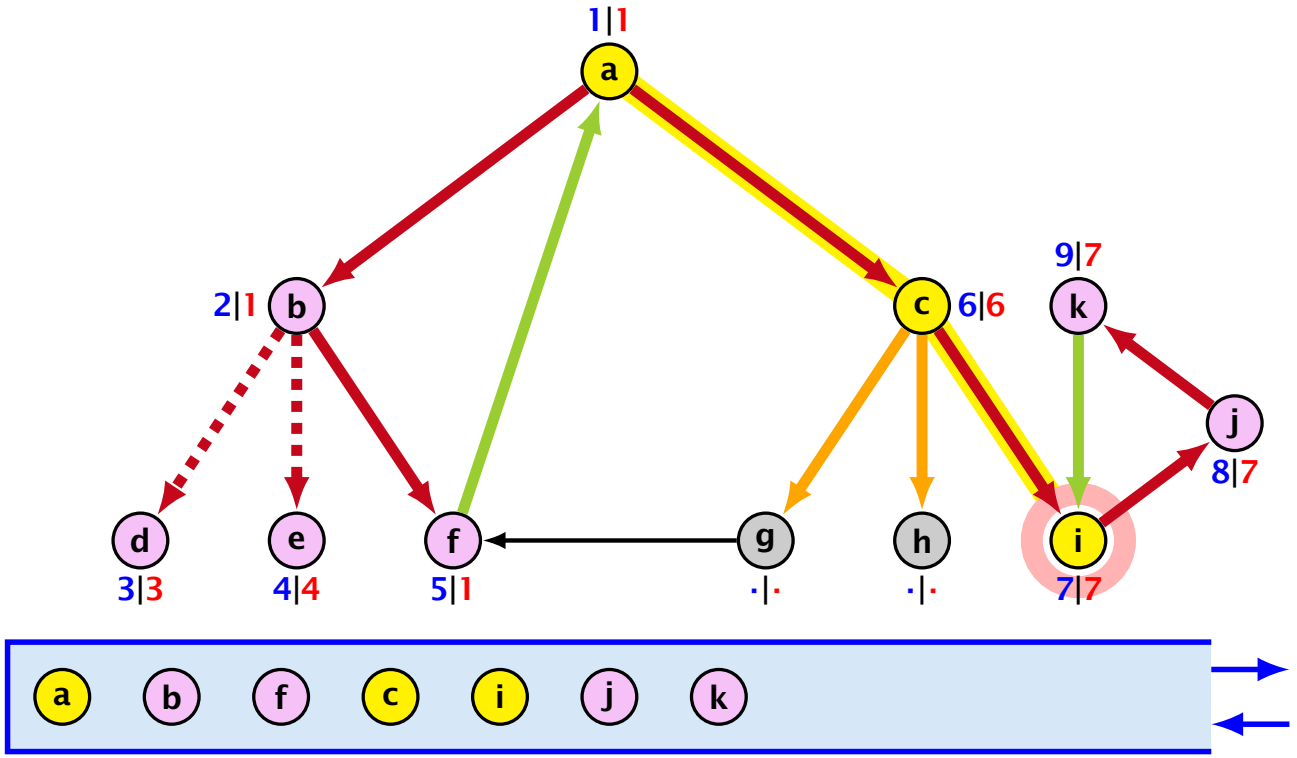
# Starker Zusammenhang



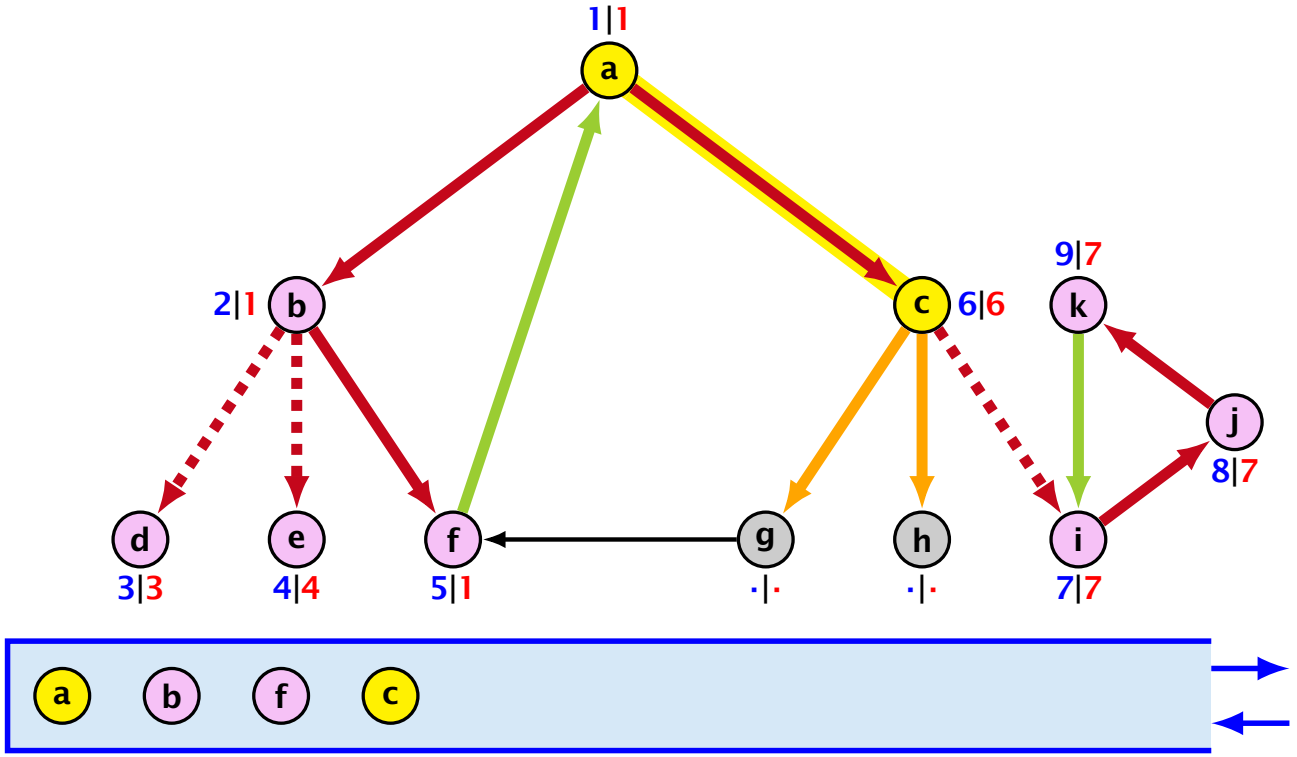
# Starker Zusammenhang



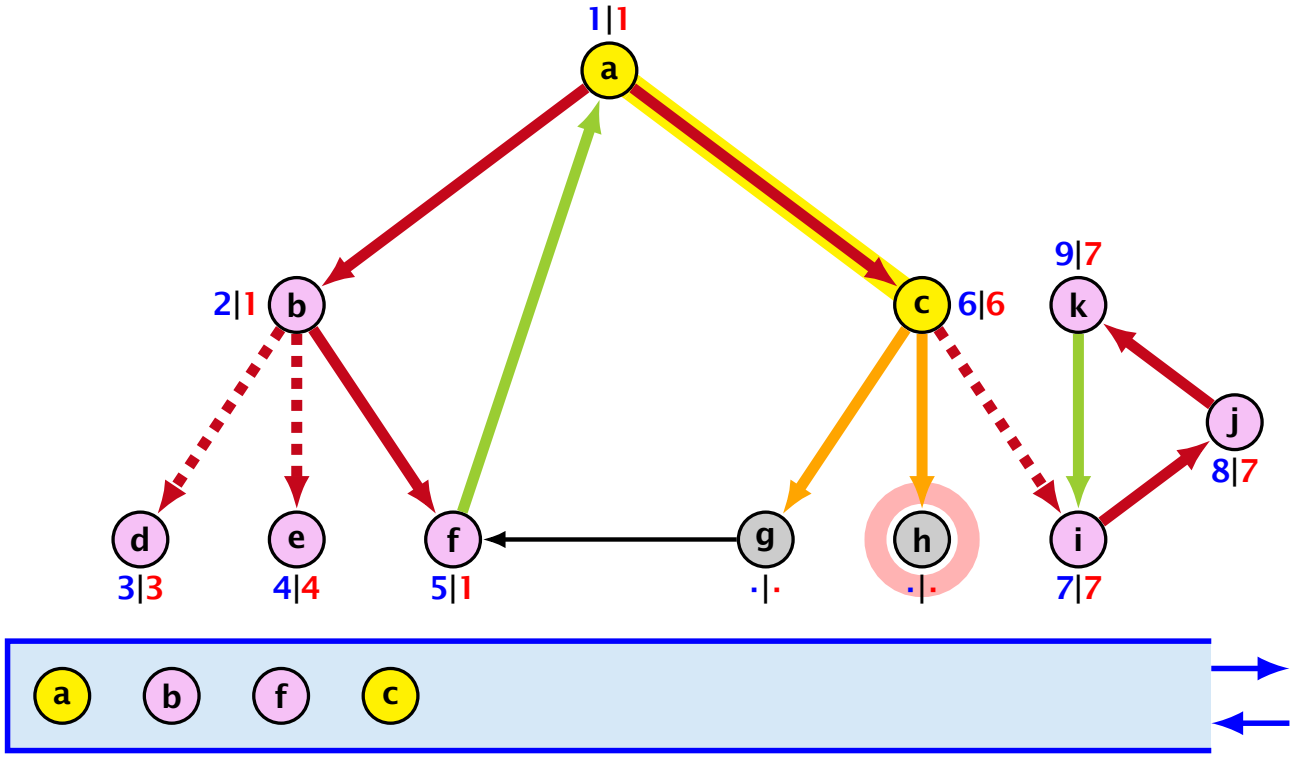
# Starker Zusammenhang



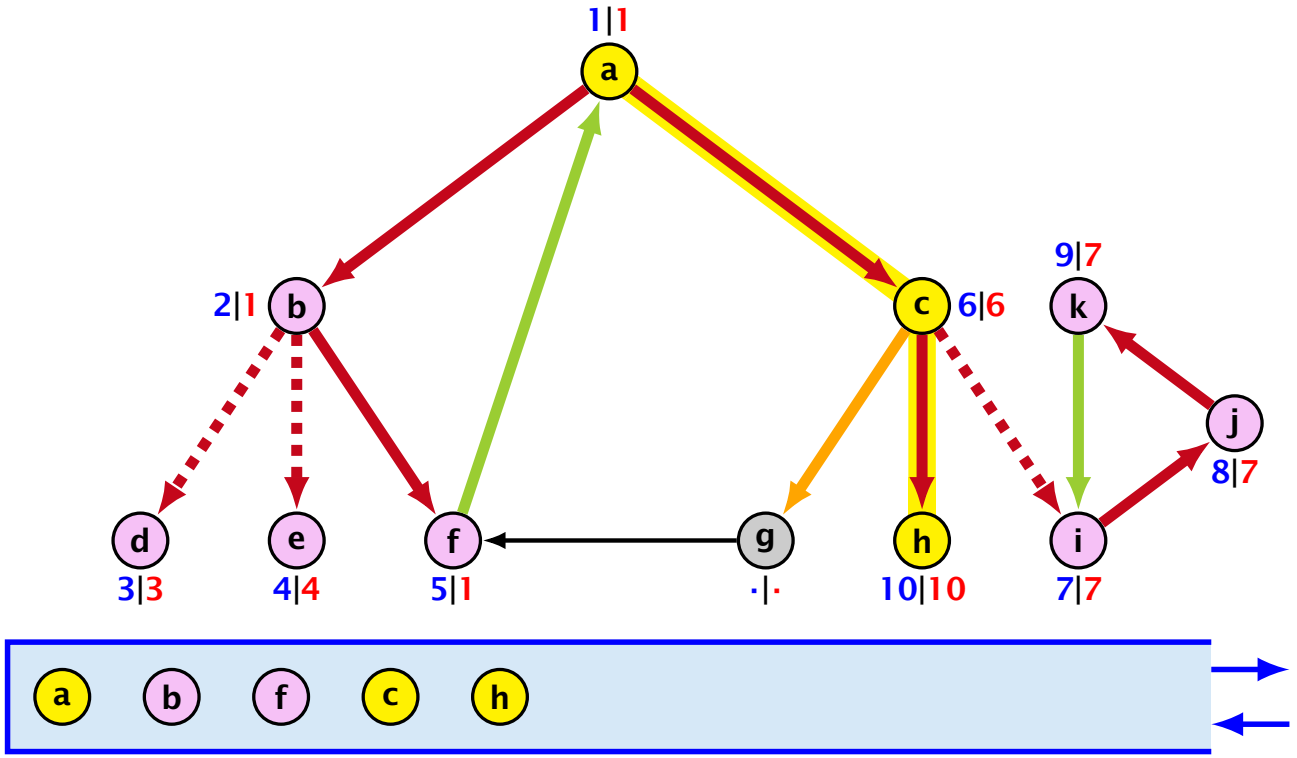
# Starker Zusammenhang



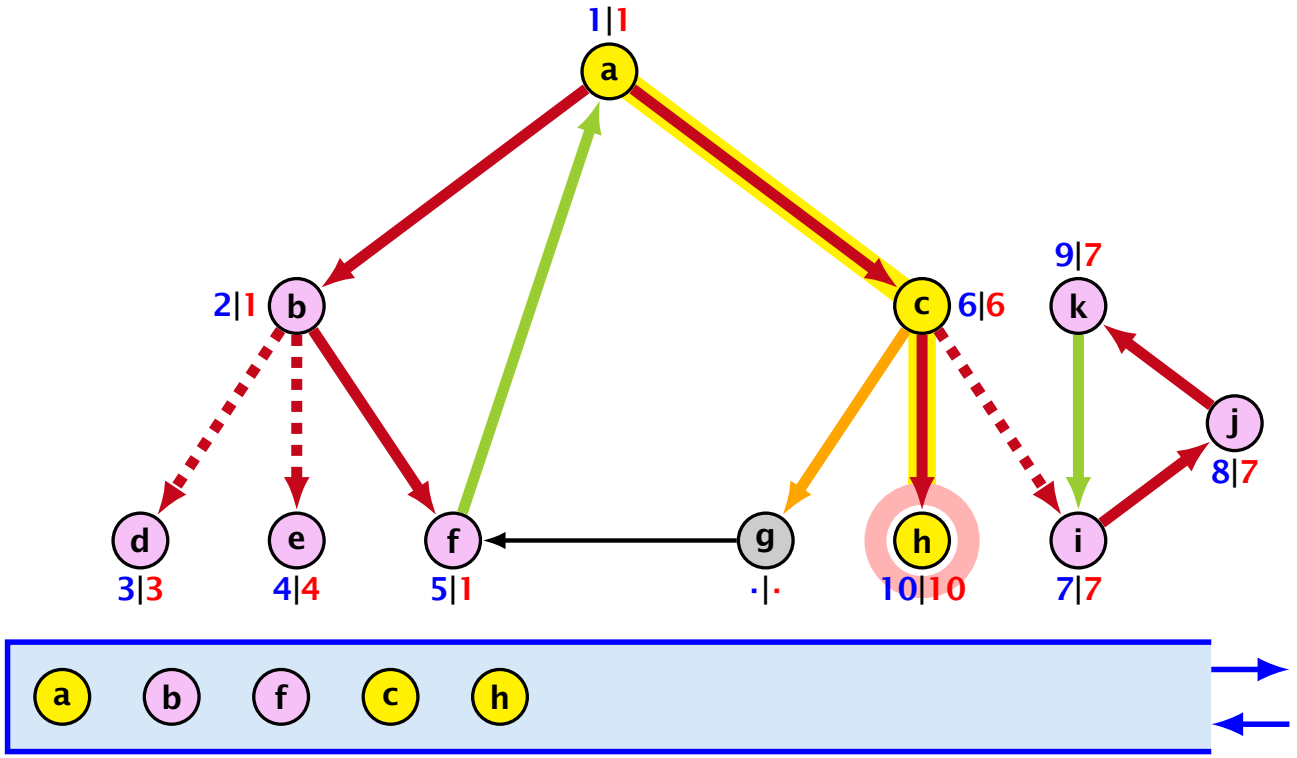
# Starker Zusammenhang



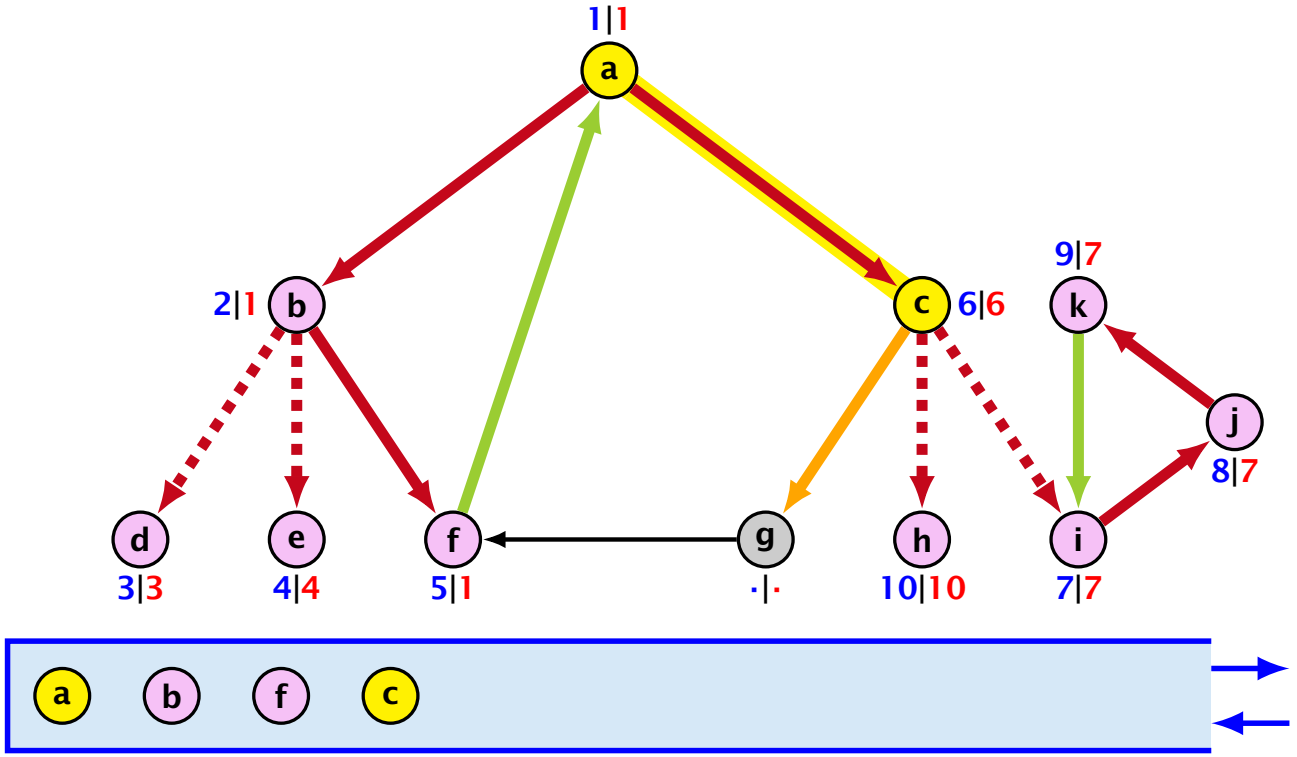
# Starker Zusammenhang



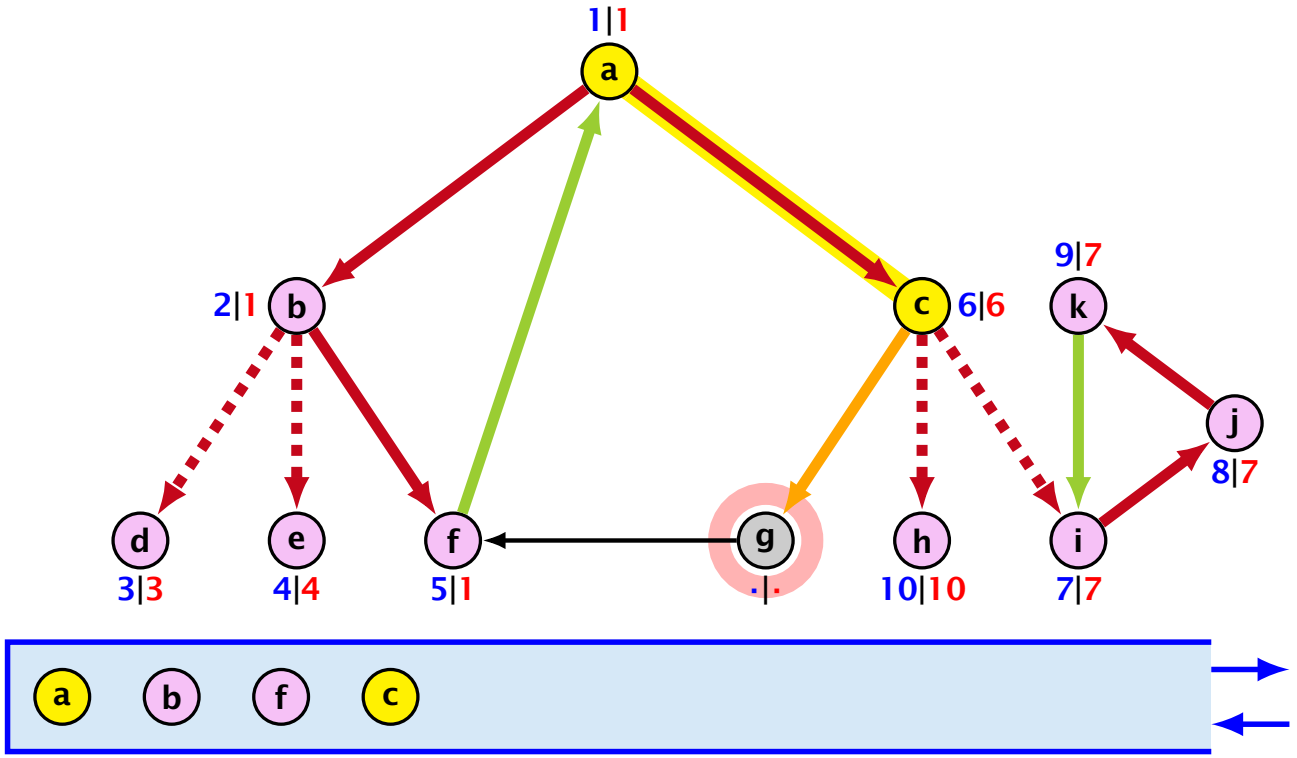
# Starker Zusammenhang



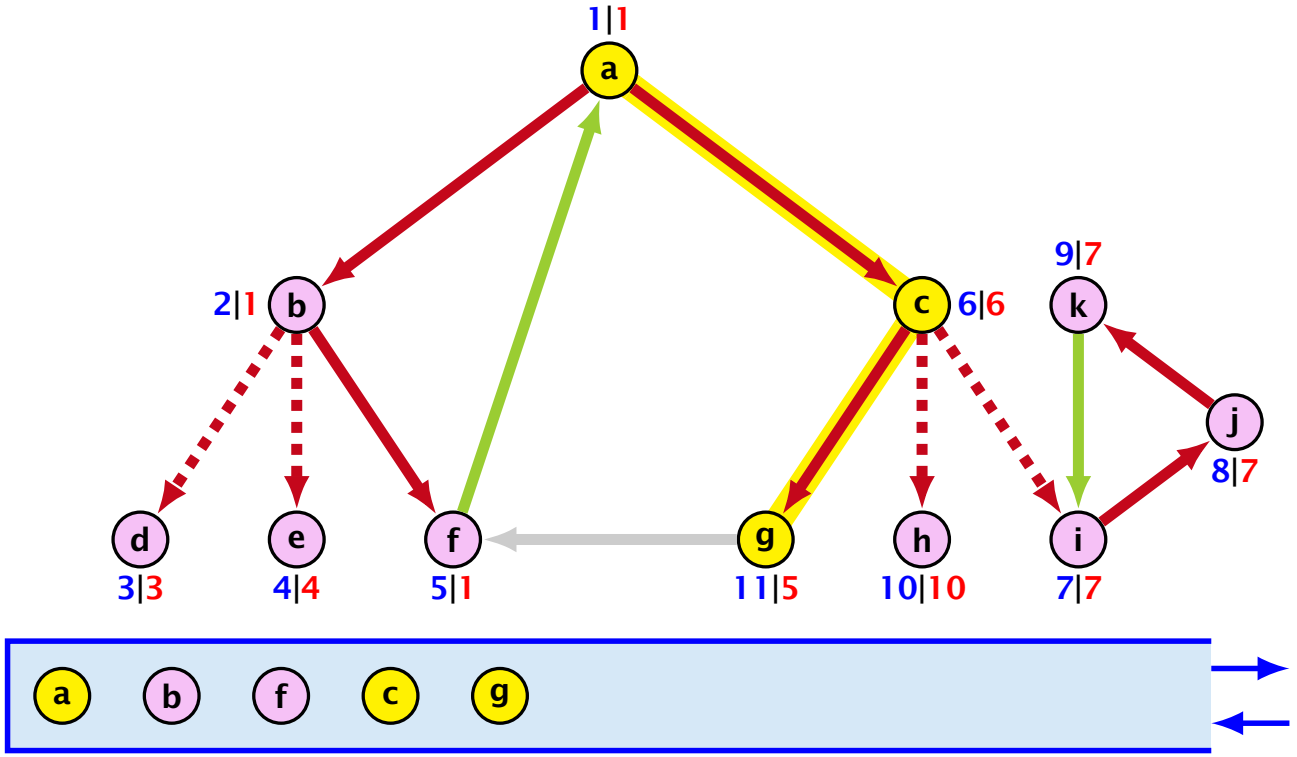
# Starker Zusammenhang



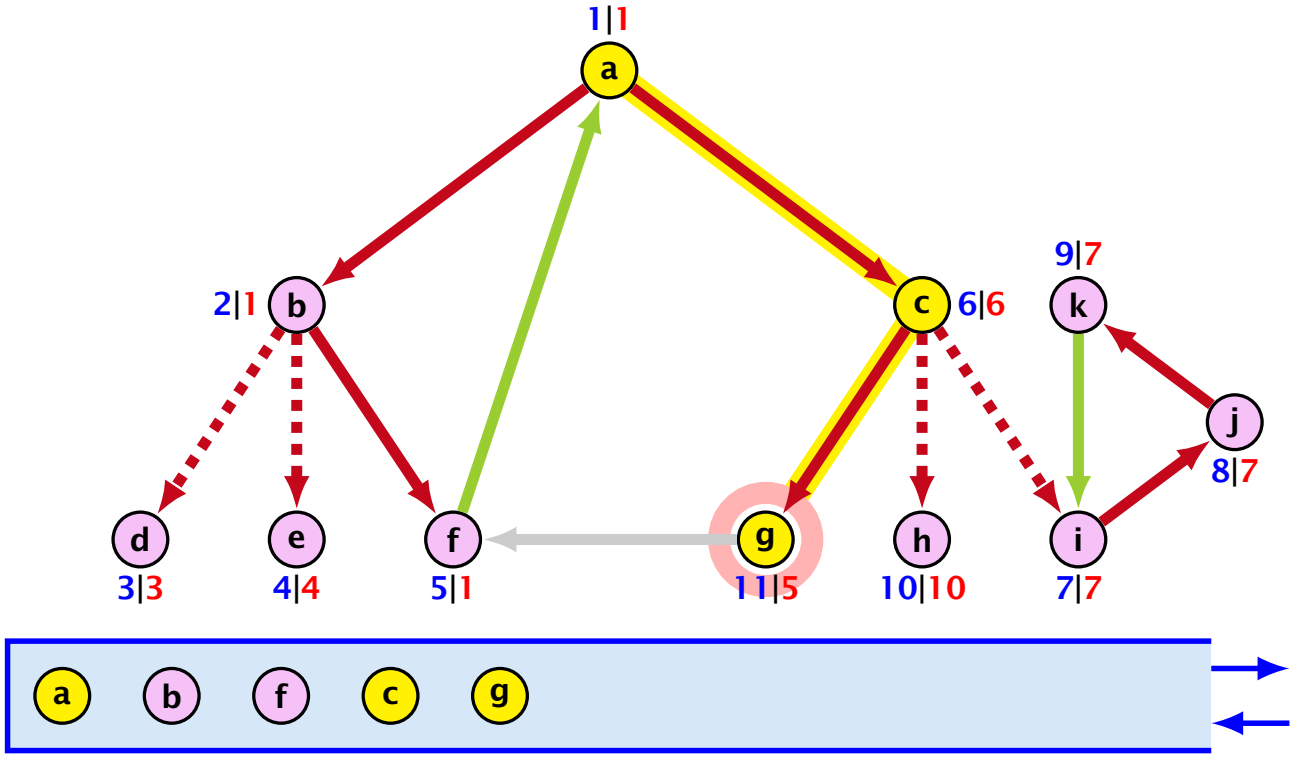
# Starker Zusammenhang



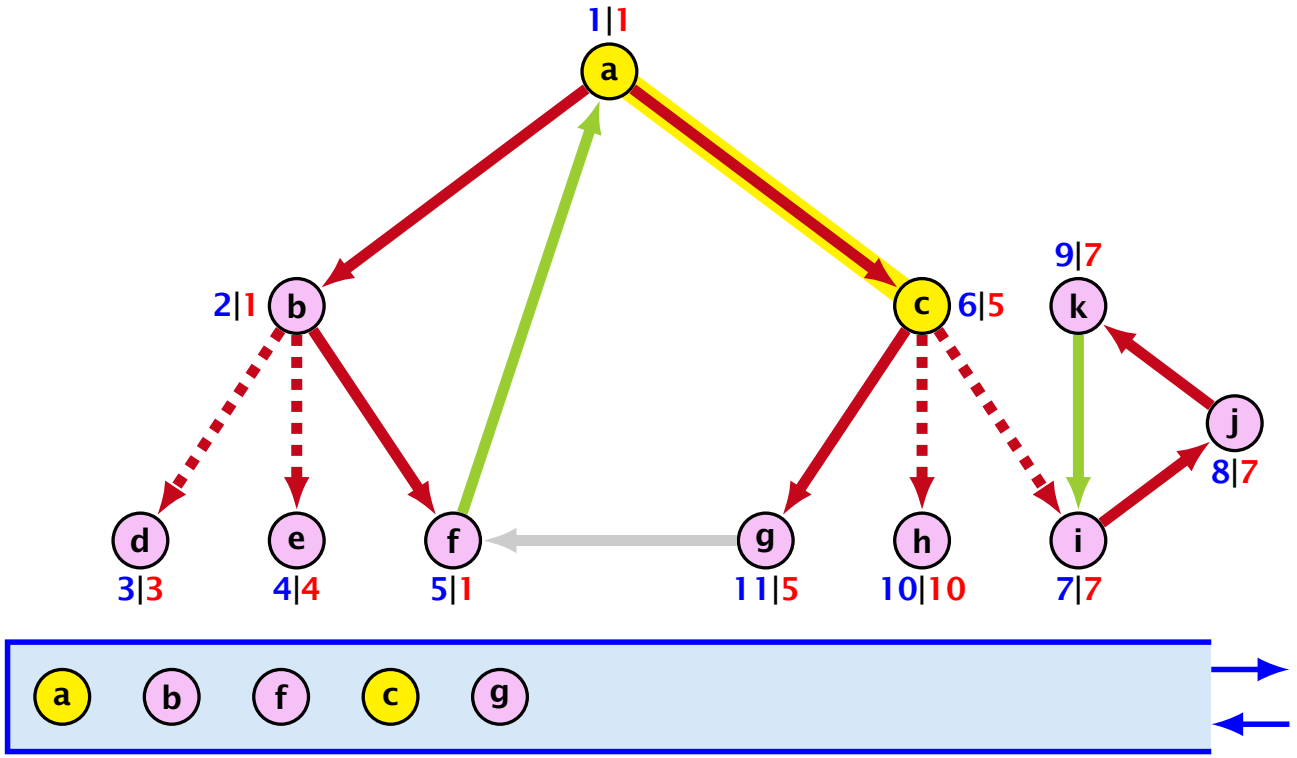
# Starker Zusammenhang



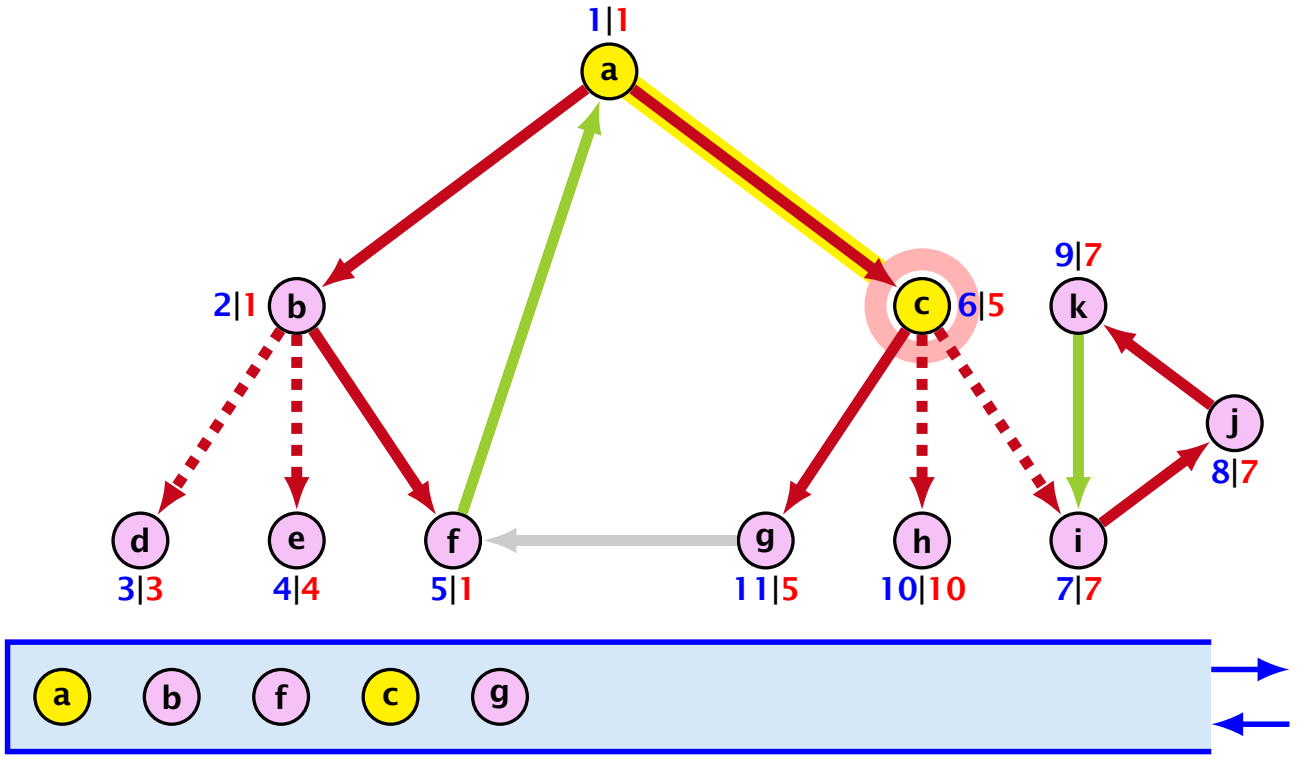
# Starker Zusammenhang



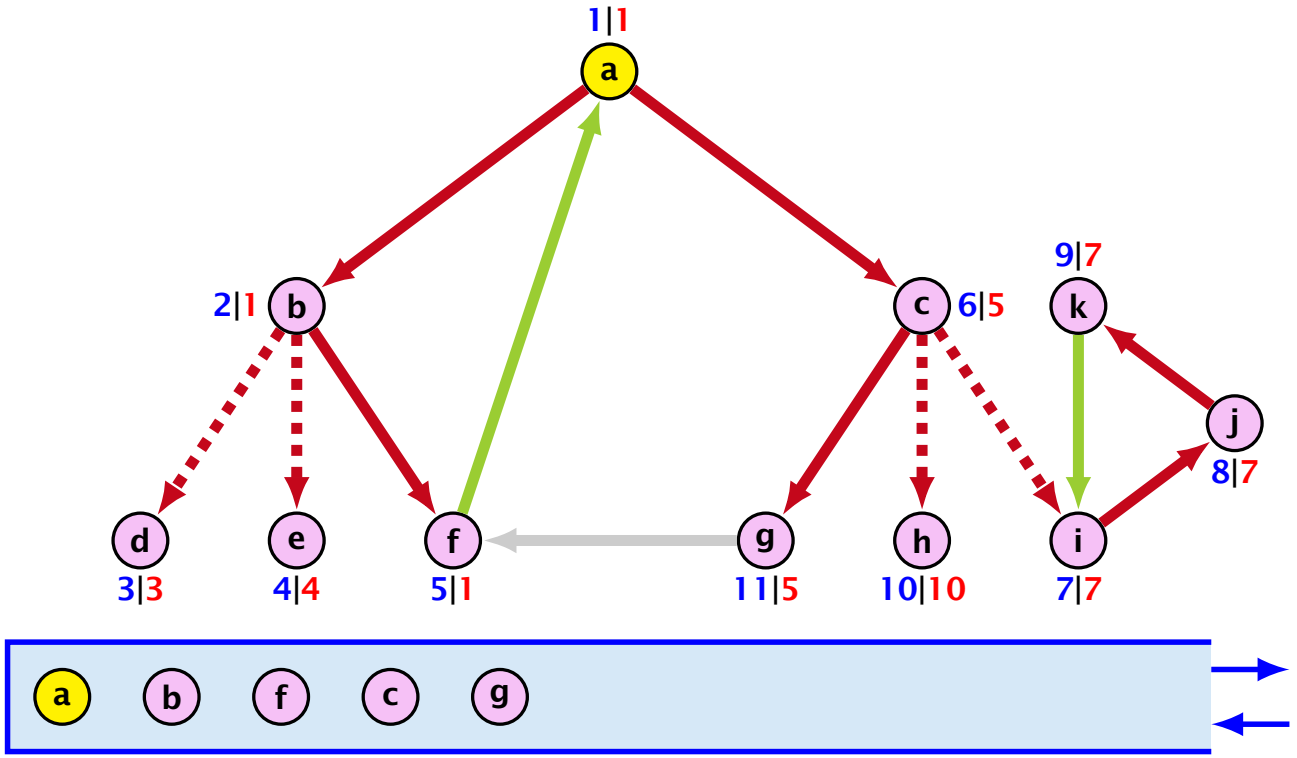
# Starker Zusammenhang



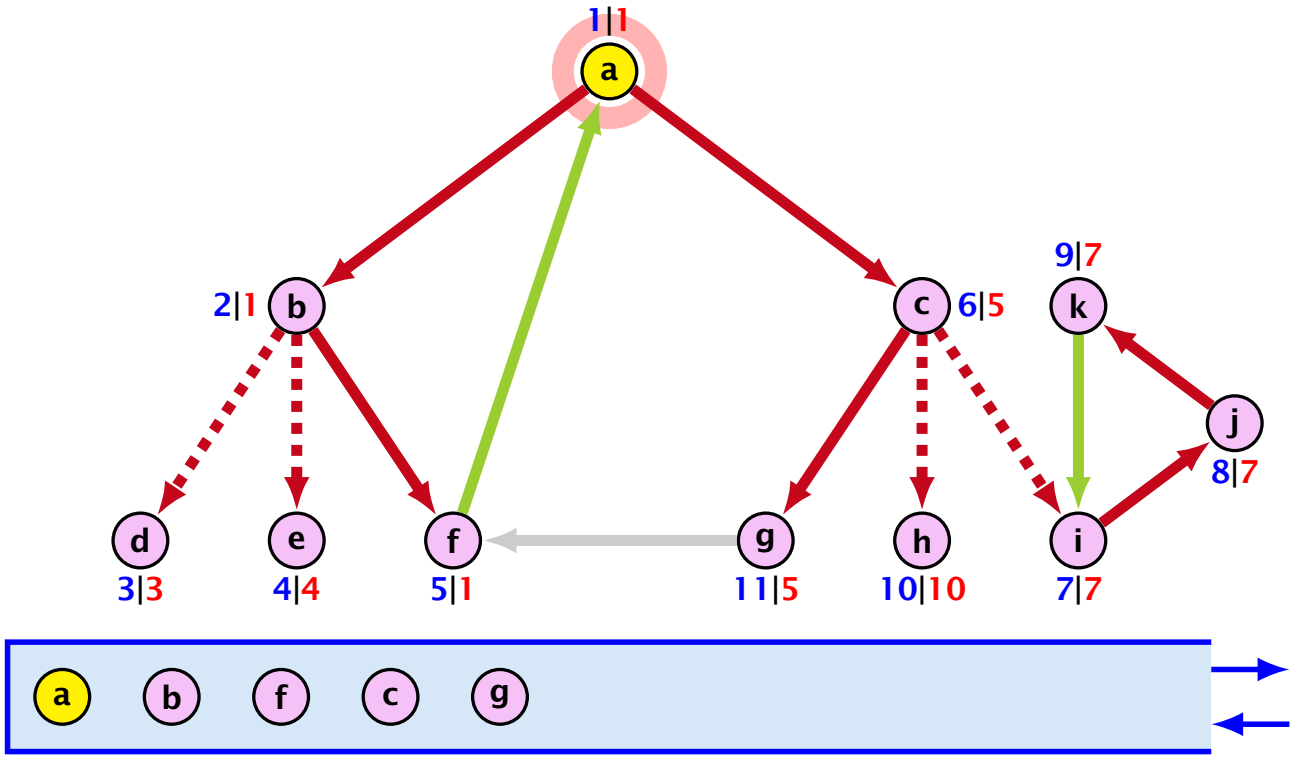
# Starker Zusammenhang



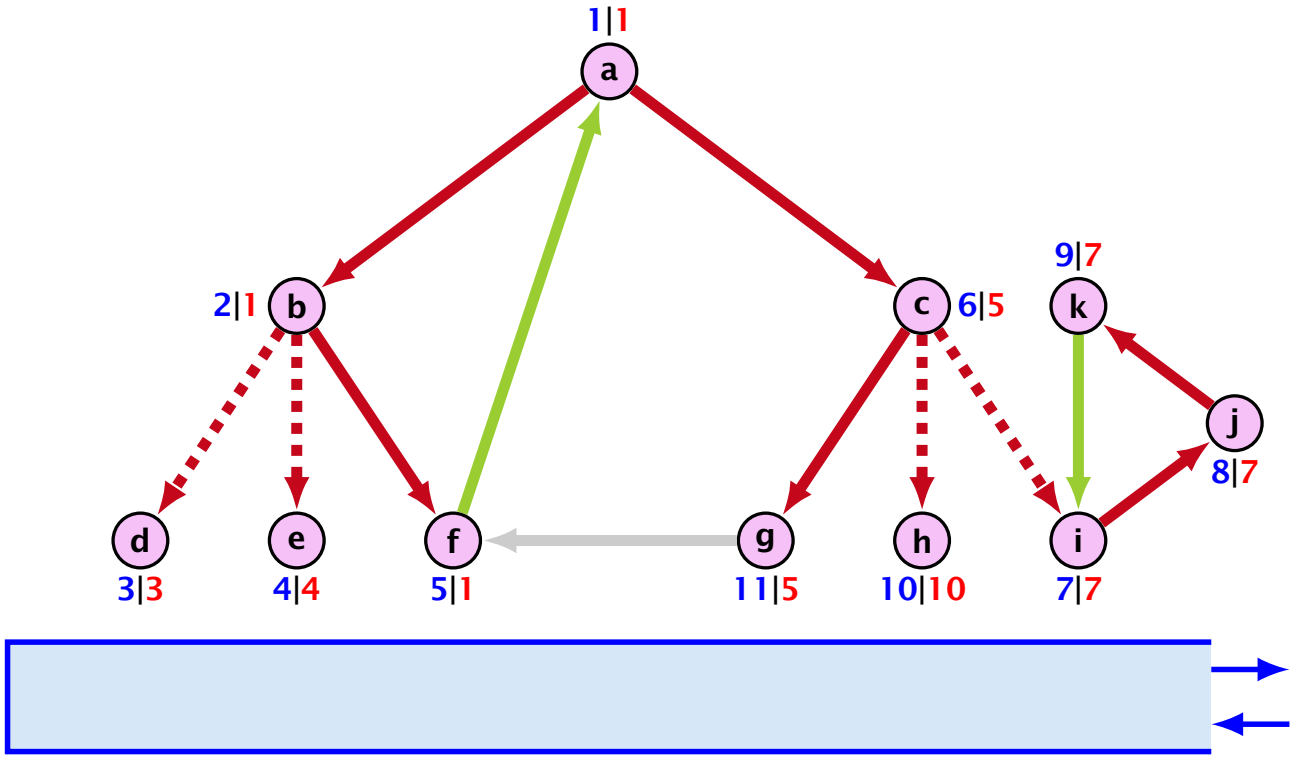
# Starker Zusammenhang



# Starker Zusammenhang



# Starker Zusammenhang



# 11 Augmentierung von Datenstrukturen

**Ziel: entwickle Datenstruktur mit Operationen:**

- **Insert($x$)**: füge $x$ ein.
- **Search($k$)**: suche nach Element mit Schlüssel $k$.
- **Delete($x$)**: Lösche Element, das durch pointer/handle $x$ referenziert wird.
- **find-by-rank($\ell$)**: gibt das $\ell$-kleinste Element zurück. NULL falls die Datenstruktur weniger als $\ell$ Elemente enthält.

Abwandlung/Augmentierung einer existierenden Datenstruktur anstatt eine neue zu entwickeln.

# 11 Augmentierung von Datenstrukturen

**Ziel: entwickle Datenstruktur mit Operationen:**

- **Insert($x$)**: füge $x$ ein.
- **Search($k$)**: suche nach Element mit Schlüssel $k$.
- **Delete($x$)**: Lösche Element, das durch pointer/handle $x$ referenziert wird.
- **find-by-rank($\ell$)**: gibt das $\ell$-kleinste Element zurück. NULL falls die Datenstruktur weniger als $\ell$ Elemente enthält.

**Abwandlung/Augmentierung einer existierenden Datenstruktur anstatt eine neue zu entwickeln.**

# 11 Augmentierung von Datenstrukturen

**Wie augmentiert man eine Datenstruktur?**

1. wähle eine zugrundeliegende Datenstruktur.
2. entscheide welche zusätzliche Information die Struktur speichern soll.
3. zeige, dass man diese zusätzlichen Informationen effizient updaten kann wenn sich die Datenstruktur ändert (z.B. beim Einfügen/Löschen)
4. entwickle die zusätzlichen Operationen

- Natürlich hängen diese Schritte voneinander ab. Es macht z.B. keinen Sinn Informationen zu speichern die man entweder nicht effizient updaten kann, oder nicht genügend Informationen um die neuen Operationen zu unterstützen.
- Allerdings kann man die obige Reihenfolge benutzen um die neue Datenstruktur zu dokumentieren.

# 11 Augmentierung von Datenstrukturen

**Wie augmentiert man eine Datenstruktur?**

1. wähle eine zugrundeliegende Datenstruktur.
2. entscheide welche zusätzliche Information die Struktur speichern soll.
3. zeige, dass man diese zusätzlichen Informationen effizient updaten kann wenn sich die Datenstruktur ändert (z.B. beim Einfügen/Löschen)
4. entwickle die zusätzlichen Operationen

- Natürlich hängen diese Schritte voneinander ab. Es macht z.B. keinen Sinn Informationen zu speichern die man entweder nicht effizient updaten kann, oder nicht genügend Informationen um die neuen Operationen zu unterstützen.
- Allerdings kann man die obige Reihenfolge benutzen um die neue Datenstruktur zu dokumentieren.

# 11 Augmentierung von Datenstrukturen

**Wie augmentiert man eine Datenstruktur?**

1. wähle eine zugrundeliegende Datenstruktur.
2. entscheide welche zusätzliche Information die Struktur speichern soll.
3. zeige, dass man diese zusätzlichen Informationen effizient updaten kann wenn sich die Datenstruktur ändert (z.B. beim Einfügen/Löschen)
4. entwickle die zusätzlichen Operationen

- Natürlich hängen diese Schritte voneinander ab. Es macht z.B. keinen Sinn Informationen zu speichern die man entweder nicht effizient updaten kann, oder nicht genügend Informationen um die neuen Operationen zu unterstützen.
- Allerdings kann man die obige Reihenfolge benutzen um die neue Datenstruktur zu dokumentieren.

# 11 Augmentierung von Datenstrukturen

**Wie augmentiert man eine Datenstruktur?**

1. wähle eine zugrundeliegende Datenstruktur.
2. entscheide welche zusätzliche Information die Struktur speichern soll.
3. zeige, dass man diese zusätzlichen Informationen effizient updaten kann wenn sich die Datenstruktur ändert (z.B. beim Einfügen/Löschen)
4. entwickle die zusätzlichen Operationen

- Natürlich hängen diese Schritte voneinander ab. Es macht z.B. keinen Sinn Informationen zu speichern die man entweder nicht effizient updaten kann, oder nicht genügend Informationen um die neuen Operationen zu unterstützen.
- Allerdings kann man die obige Reihenfolge benutzen um die neue Datenstruktur zu dokumentieren.

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

1. Wir wählen einen AVL-Baum als zugrundeliegende Datenstruktur.
2. Wir speichern in jedem Knoten $v$ die Größe des zugehörigen Teilbaums.
3. Wir müssen in der Lage sein, die Größenfelder in jedem Knoten zu aktualisieren ohne die asymptotische Laufzeit von insert, delete, und search zu beeinträchtigen. Später...

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

1. Wir wählen einen AVL-Baum als zugrundeliegende Datenstruktur.
2. Wir speichern in jedem Knoten $v$ die Größe des zugehörigen Teilbaums.
3. Wir müssen in der Lage sein, die Größenfelder in jedem Knoten zu aktualisieren ohne die asymptotische Laufzeit von insert, delete, und search zu beeinträchtigen. Später...

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

1. Wir wählen einen AVL-Baum als zugrundeliegende Datenstruktur.
2. Wir speichern in jedem Knoten $v$ die Größe des zugehörigen Teilbaums.
3. Wir müssen in der Lage sein, die Größenfelder in jedem Knoten zu aktualisieren ohne die asymptotische Laufzeit von insert, delete, und search zu beeinträchtigen. Später...

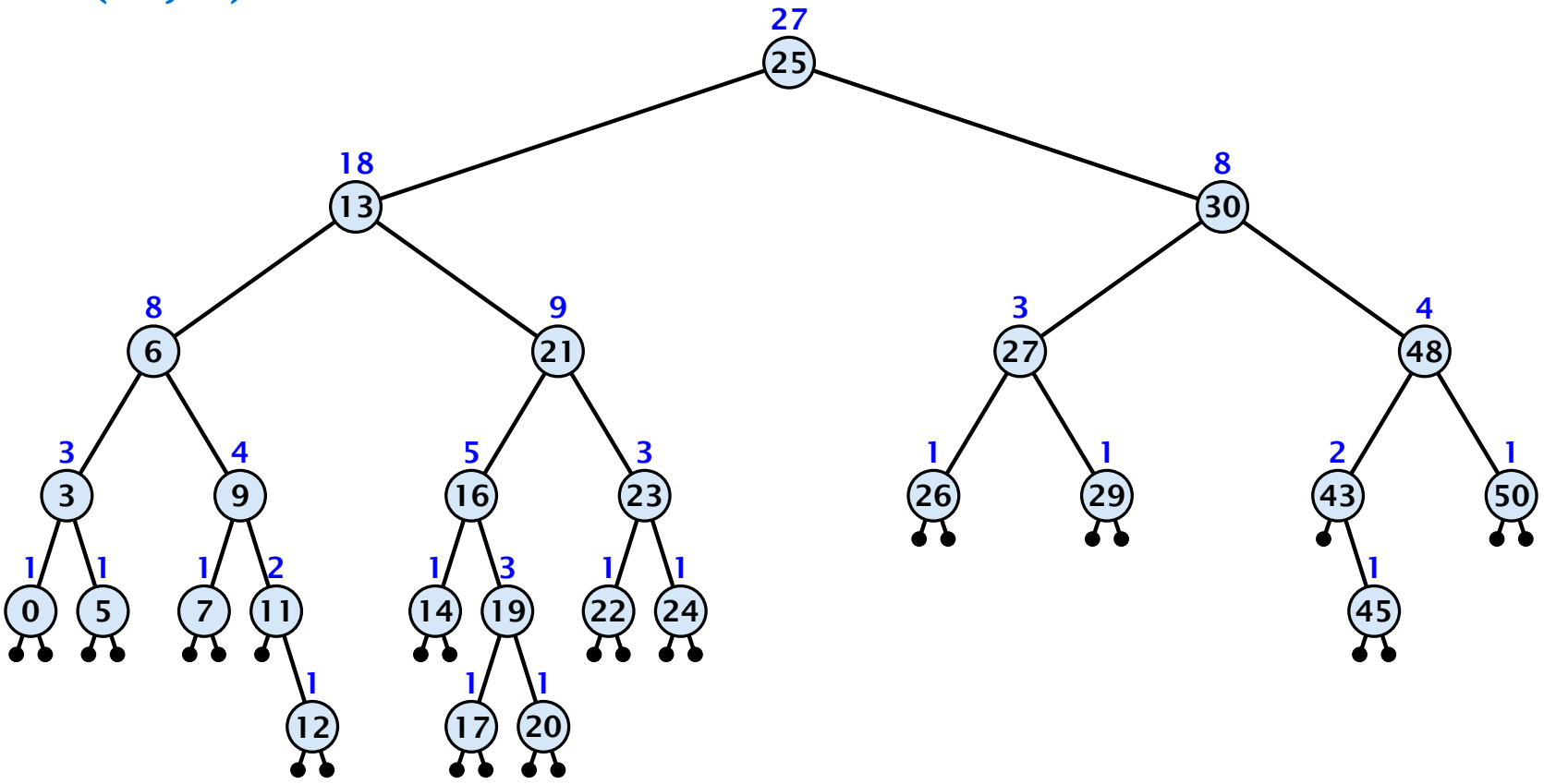
# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

4. Wie funktioniert find-by-rank?
   Find-by-rank($k$) := Select(root,$k$) mit

```
1  Input: Zeiger auf Wurzel eines Teilbaums
2         gib i-kleinstes Element des Teilbaums aus
3
4  Select(x,i)
5      if (x == NULL) return NULL;
6      if (x->left != NULL) r = x->left->size+1; else r = 1;
7      if (i == r) return x;
8      if (i < r)
9          return Select(x->left,i);
10     else
11         return Select(x->right,i-r);
```

# Select$(x, i)$

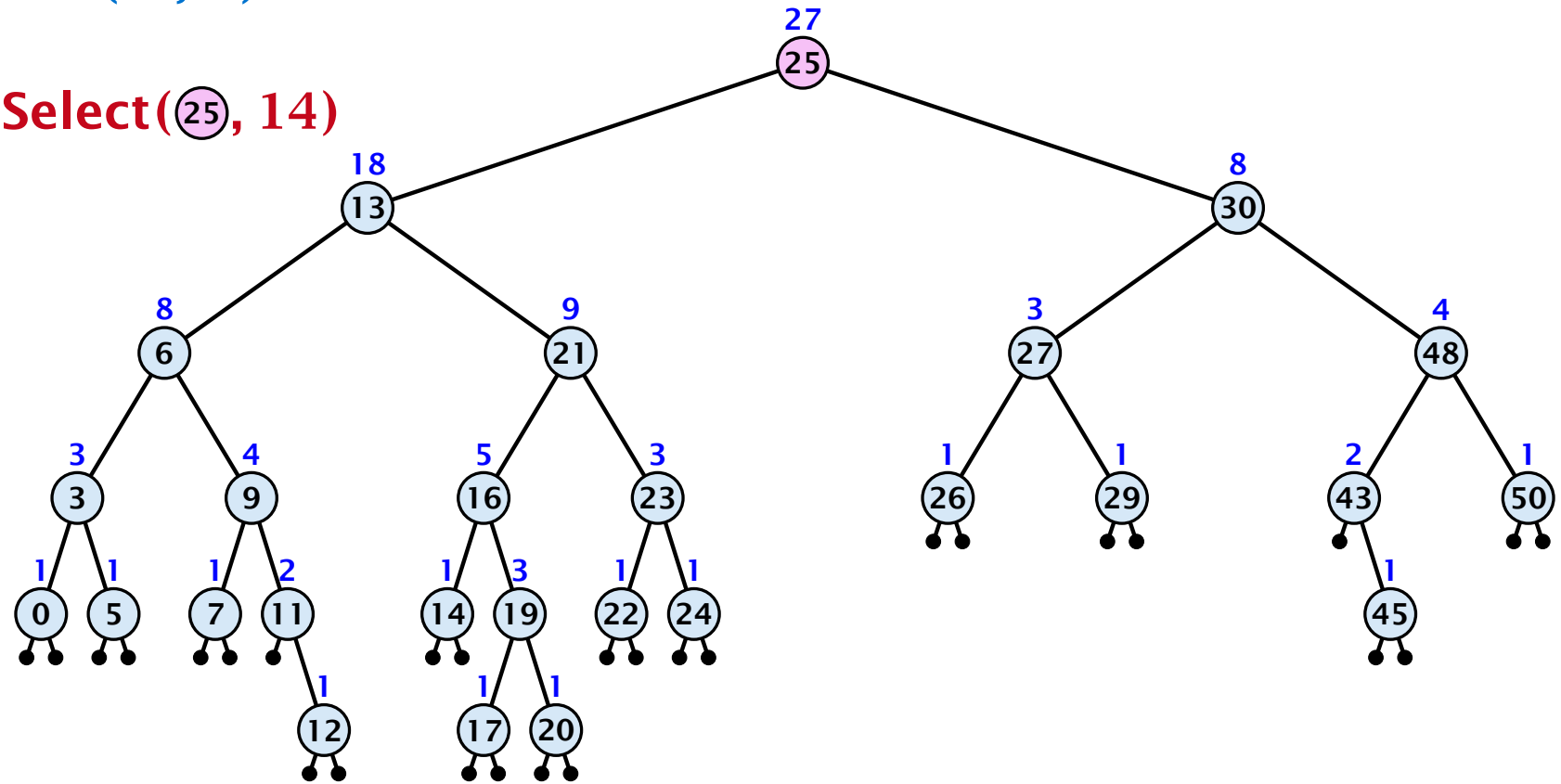



**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# Select$(x, i)$

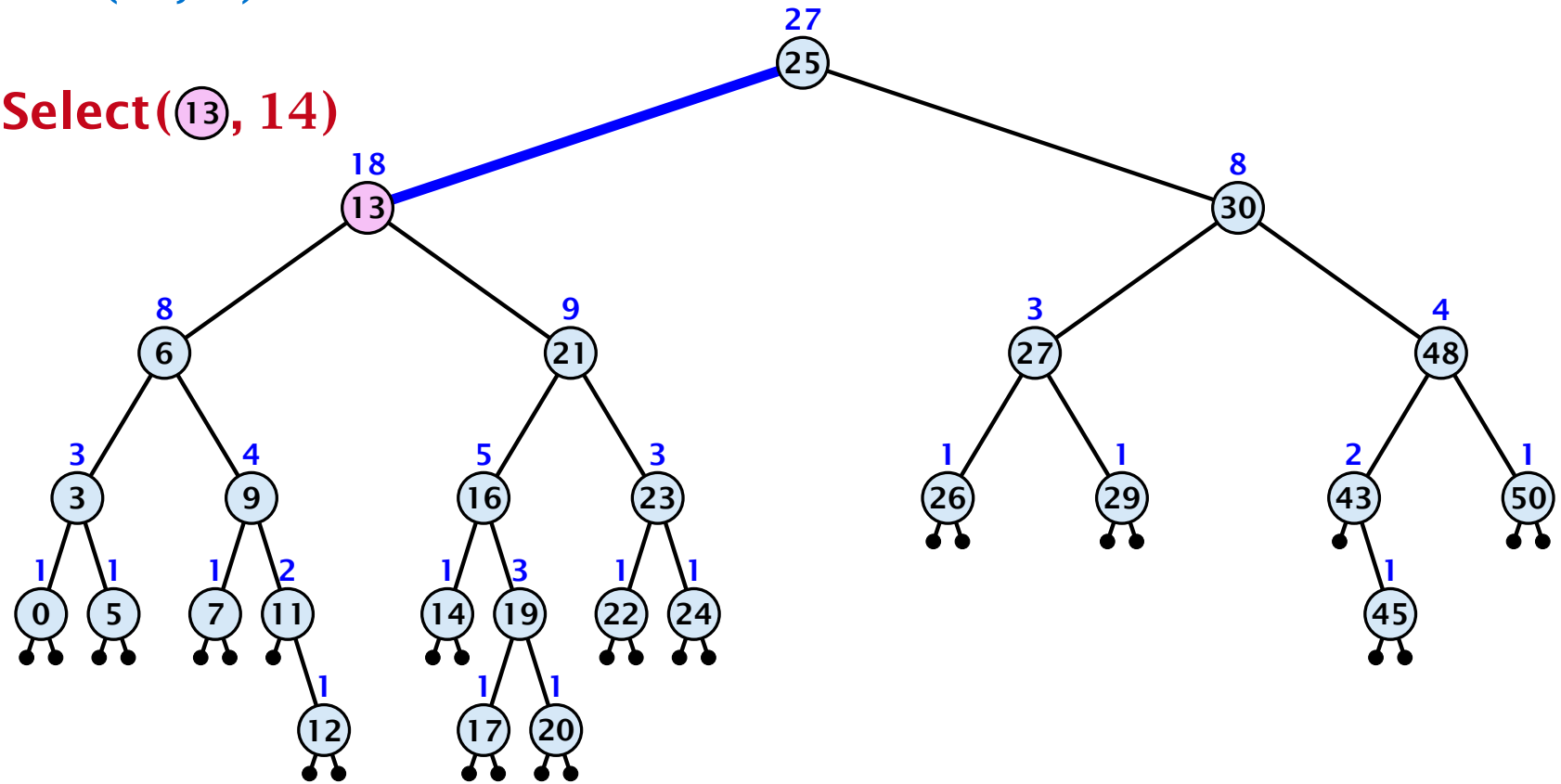
**Select(25, 14)**




**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# Select$(x, i)$

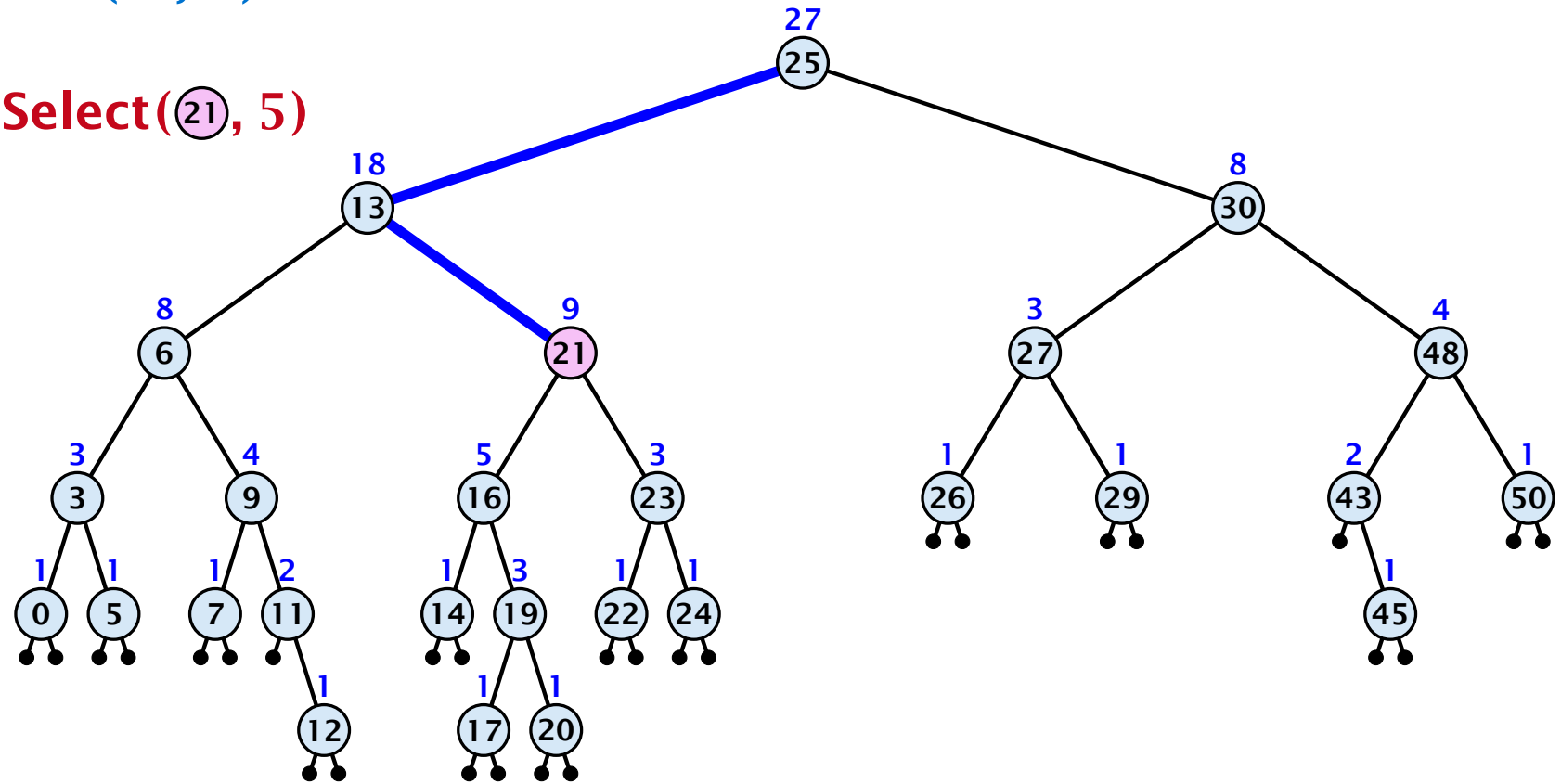



**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# Select$(x, i)$

Select(21, 5)

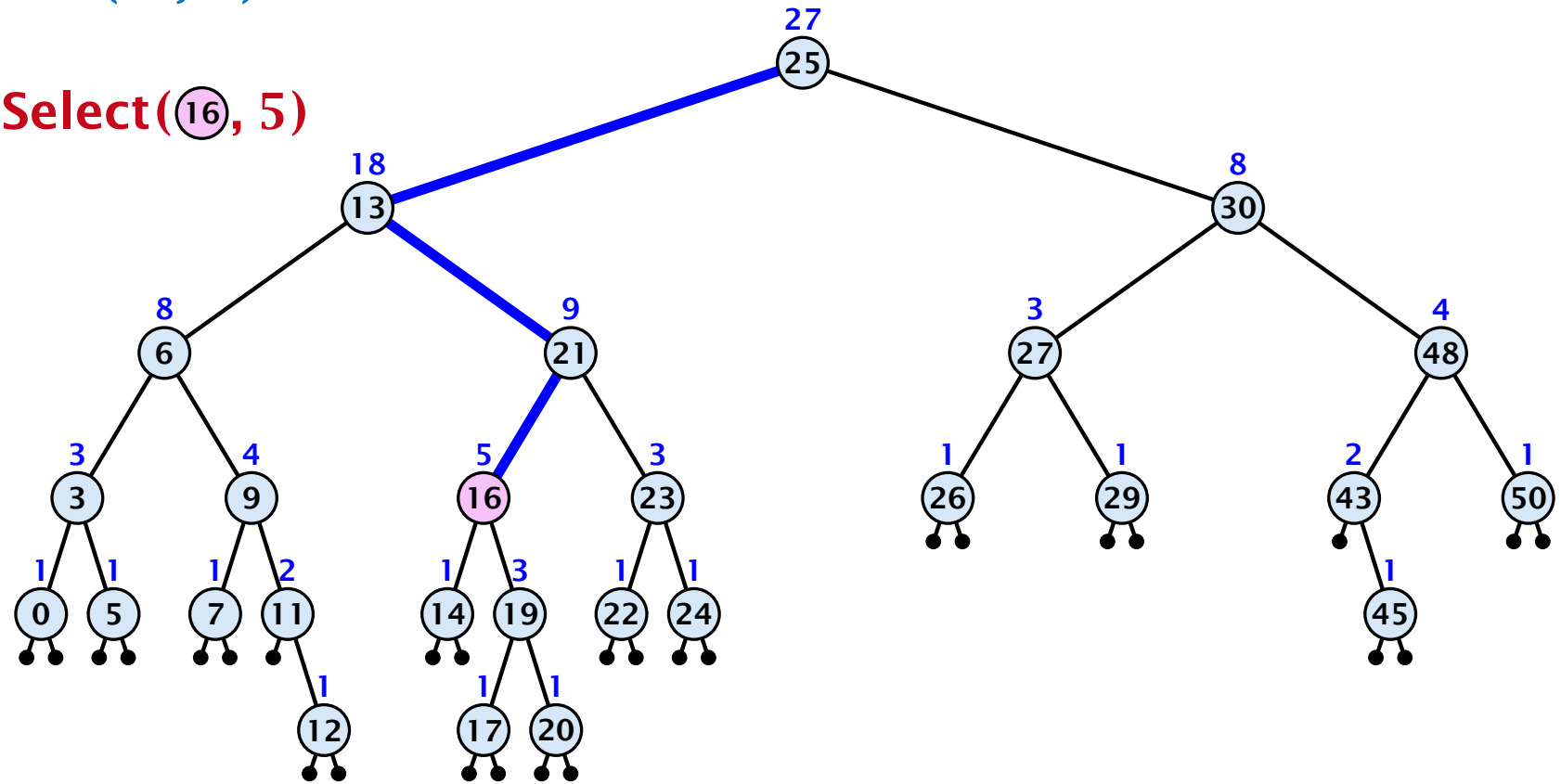



**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# Select$(x, i)$

Select(16, 5)

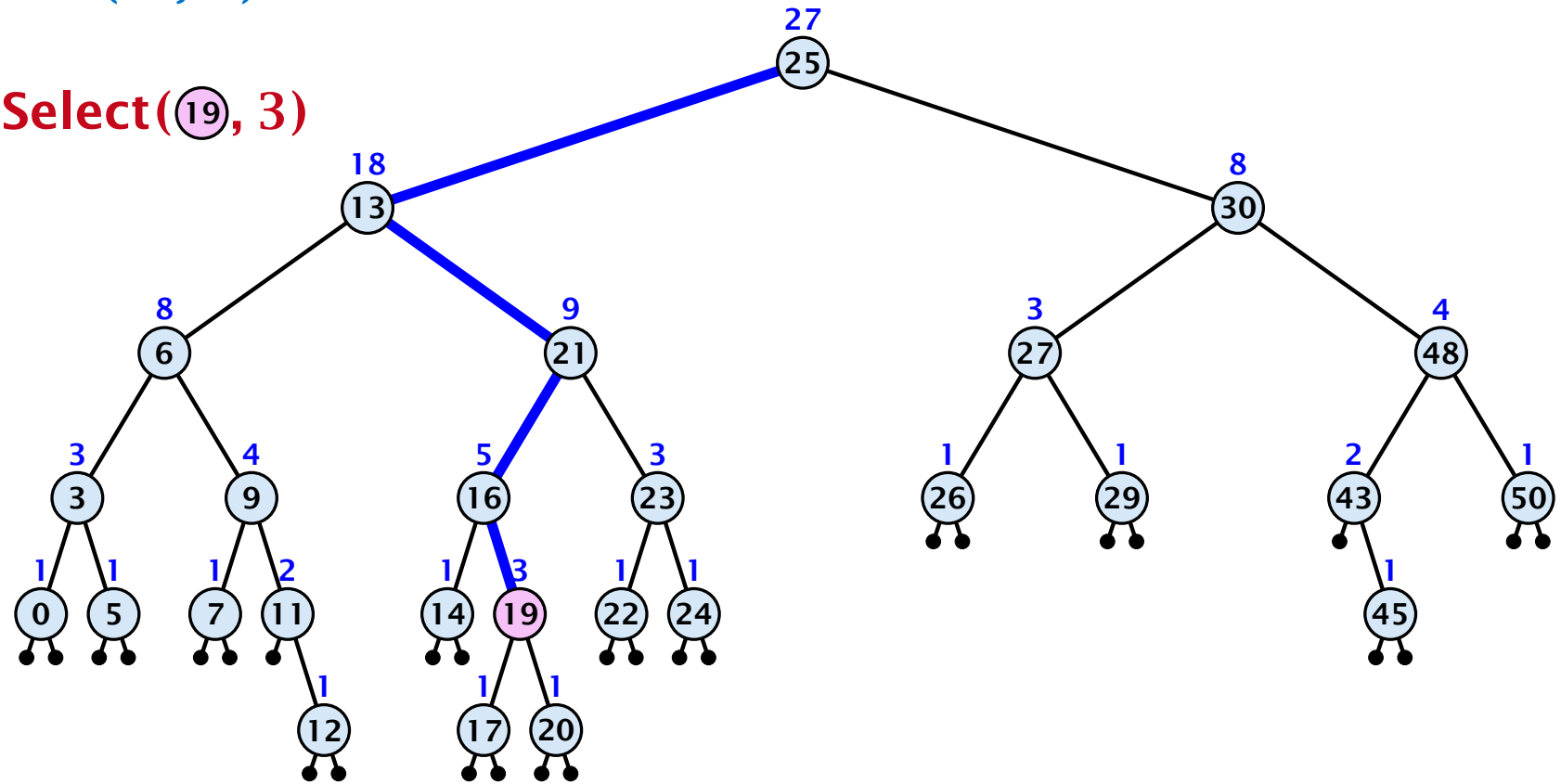



**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# Select$(x, i)$

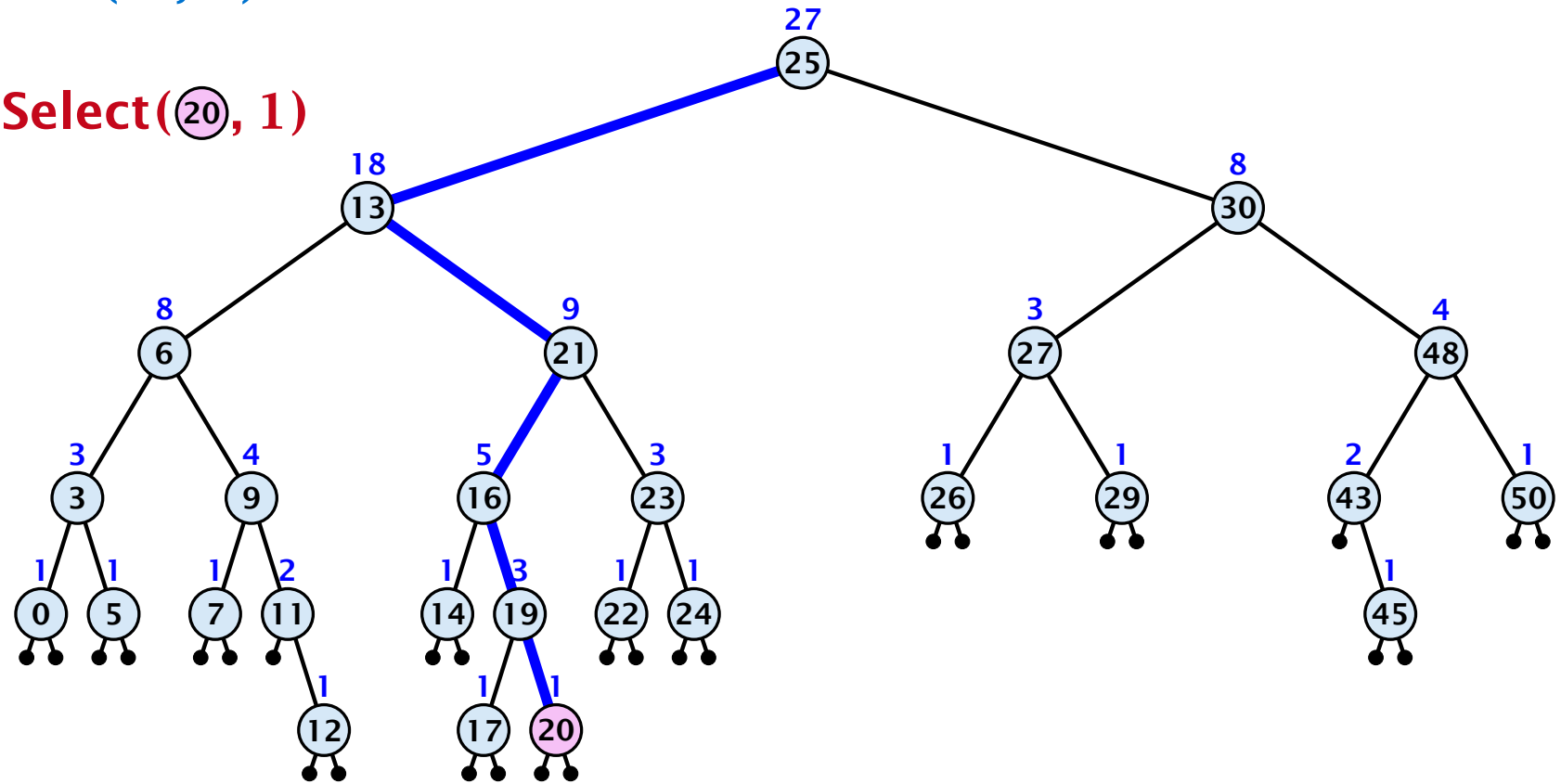
Select(19, 3)




**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# Select$(x, i)$




**Find-by-rank:**

- entscheide ob man in dem linken oder rechten Teilbaum weitersuchen muss
- passe den index des Elementes an nach dem gesucht werden muss falls man rechts verzweigt.

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

3. Wie wird die Information aktualisiert?

**Search($k$):** Nichts zu tun.

**Insert($x$):** Während man nach der Einfügeposition sucht erhöht man das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

**Delete($x$):** Wenn man ein Knoten überbrückt läuft man vom überbrückten Knoten aufwärts im Baum und reduziert das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

3. Wie wird die Information aktualisiert?

**Search($k$)**: Nichts zu tun.

**Insert($x$)**: Während man nach der Einfügeposition sucht erhöht man das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

**Delete($x$)**: Wenn man ein Knoten überbrückt läuft man vom überbrückten Knoten aufwärts im Baum und reduziert das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

3. Wie wird die Information aktualisiert?

**Search($k$)**: Nichts zu tun.

**Insert($x$)**: Während man nach der Einfügeposition sucht erhöht man das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

**Delete($x$)**: Wenn man ein Knoten überbrückt läuft man vom überbrückten Knoten aufwärts im Baum und reduziert das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

# 11 Augmentierung von Datenstrukturen

**Ziel: Datenstruktur mit insert, delete, search, und find-by-rank mit Laufzeit $\mathcal{O}(\log n)$.**

3. Wie wird die Information aktualisiert?

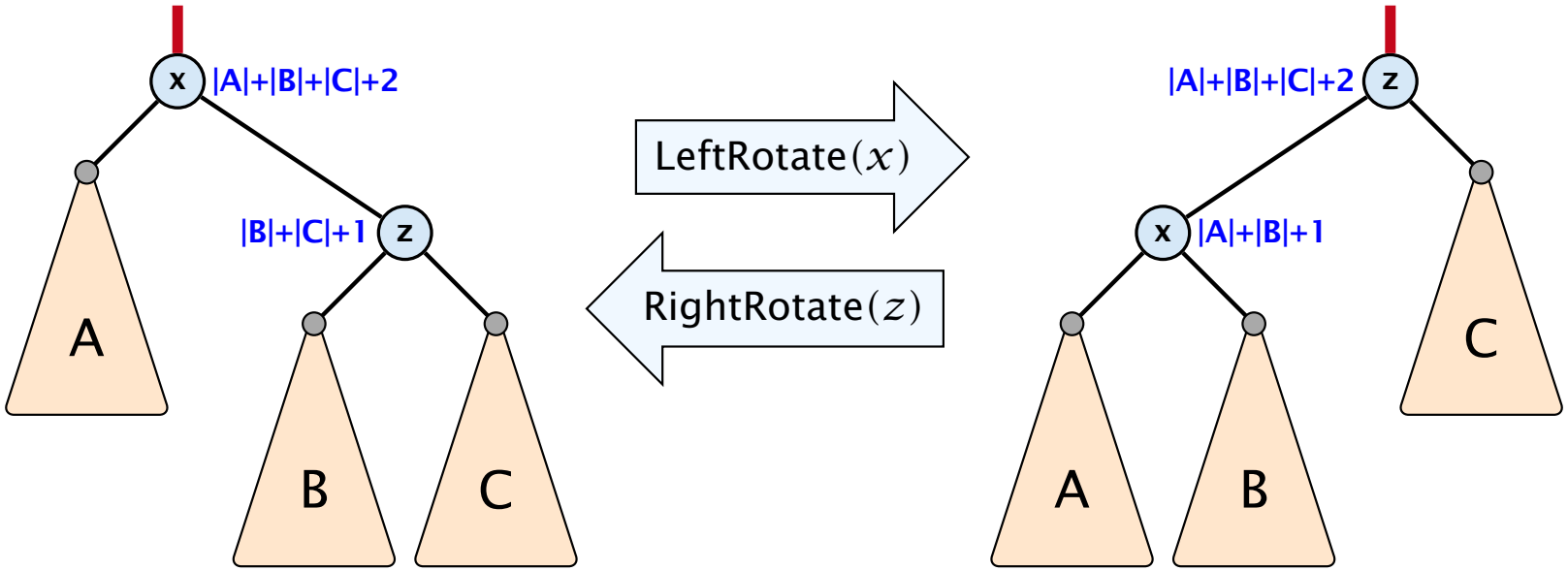
**Search($k$):** Nichts zu tun.

**Insert($x$):** Während man nach der Einfügeposition sucht erhöht man das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

**Delete($x$):** Wenn man ein Knoten überbrückt läuft man vom überbrückten Knoten aufwärts im Baum und reduziert das Größenfeld jedes besuchten Knotens um 1. Größenfeld bei Rotationen aktualisieren!

# Rotationen

Die einzige Operationen während der fix-up Routinen, die den Baum verändert und eine Aktualisierung der Größenfelder benötigt.




Die Knoten $x$ und $z$ sind die einzigen Knoten die ihre Größenfelder ändern.

Die neuen Größenfelder können lokal aus den Größenfelder der Kinder berechnet werden.

# 12 Union Find

**Union-Find Datenstruktur $\mathcal{P}$:** Verwaltet die Partionierung einer Menge $M$, d.h., die Zerlegung der Menge in disjunkte Teilmengen.

**Operationen**

- $\mathcal{P}.\mathbf{makeset}(x)$: Fügt ein Element $x$ zu $M$ hinzu und fügt es in eine Teilmenge $\{x\}$ ein, d.h., erzeugt eine Teilmenge, die nur dieses Element enthält. Gibt ein handle für $x$ zurück.
- $\mathcal{P}.\mathbf{find}(x)$: Input: handle für ein Element $x$; findet die Menge, die $x$ enthält; gibt einen Repräsentanten/Identifier für die Menge zurück.
- $\mathcal{P}.\mathbf{union}(x,y)$: Input: handles von zwei Elementen $x$ und $y$, die momentan in Mengen $S_x$ bzw. $S_y$ sind. Die Funktion ersetzt $S_x$ und $S_y$ durch $S_x \cup S_y$ und gibt den Repräsentanten/Identifier der neuen Menge zurück.

# 12 Union Find

**Union-Find Datenstruktur $\mathcal{P}$:** Verwaltet die Partionierung einer Menge $M$, d.h., die Zerlegung der Menge in disjunkte Teilmengen.

**Operationen**

- $\mathcal{P}.\mathbf{makeset}(x)$**:** Fügt ein Element $x$ zu $M$ hinzu und fügt es in eine Teilmenge $\{x\}$ ein, d.h., erzeugt eine Teilmenge, die nur dieses Element enthält. Gibt ein handle für $x$ zurück.
- $\mathcal{P}.\mathbf{find}(x)$**:** Input: handle für ein Element $x$; findet die Menge, die $x$ enthält; gibt einen Repräsentanten/Identifier für die Menge zurück.
- $\mathcal{P}.\mathbf{union}(x,y)$**:** Input: handles von zwei Elementen $x$ und $y$, die momentan in Mengen $S_x$ bzw. $S_y$ sind. Die Funktion ersetzt $S_x$ und $S_y$ durch $S_x \cup S_y$ und gibt den Repräsentanten/Identifier der neuen Menge zurück.

# 12 Union Find

**Union-Find Datenstruktur $\mathcal{P}$:** Verwaltet die Partionierung einer Menge $M$, d.h., die Zerlegung der Menge in disjunkte Teilmengen.

**Operationen**

- $\mathcal{P}.\mathbf{makeset}(x)$**:** Fügt ein Element $x$ zu $M$ hinzu und fügt es in eine Teilmenge $\{x\}$ ein, d.h., erzeugt eine Teilmenge, die nur dieses Element enthält. Gibt ein handle für $x$ zurück.
- $\mathcal{P}.\mathbf{find}(x)$**:** Input: handle für ein Element $x$; findet die Menge, die $x$ enthält; gibt einen Repräsentanten/Identifier für die Menge zurück.
- $\mathcal{P}.\mathbf{union}(x,y)$**:** Input: handles von zwei Elementen $x$ und $y$, die momentan in Mengen $S_x$ bzw. $S_y$ sind. Die Funktion ersetzt $S_x$ und $S_y$ durch $S_x \cup S_y$ und gibt den Repräsentanten/Identifier der neuen Menge zurück.

# 12 Union Find

**Union-Find Datenstruktur $\mathcal{P}$:** Verwaltet die Partionierung einer Menge $M$, d.h., die Zerlegung der Menge in disjunkte Teilmengen.

**Operationen**

- $\mathcal{P}.\mathbf{makeset}(x)$**:** Fügt ein Element $x$ zu $M$ hinzu und fügt es in eine Teilmenge $\{x\}$ ein, d.h., erzeugt eine Teilmenge, die nur dieses Element enthält. Gibt ein handle für $x$ zurück.
- $\mathcal{P}.\mathbf{find}(x)$**:** Input: handle für ein Element $x$; findet die Menge, die $x$ enthält; gibt einen Repräsentanten/Identifier für die Menge zurück.
- $\mathcal{P}.\mathbf{union}(x,y)$**:** Input: handles von zwei Elementen $x$ und $y$, die momentan in Mengen $S_x$ bzw. $S_y$ sind. Die Funktion ersetzt $S_x$ und $S_y$ durch $S_x \cup S_y$ und gibt den Repräsentanten/Identifier der neuen Menge zurück.

# 12 Union Find

**Anwendungen:**

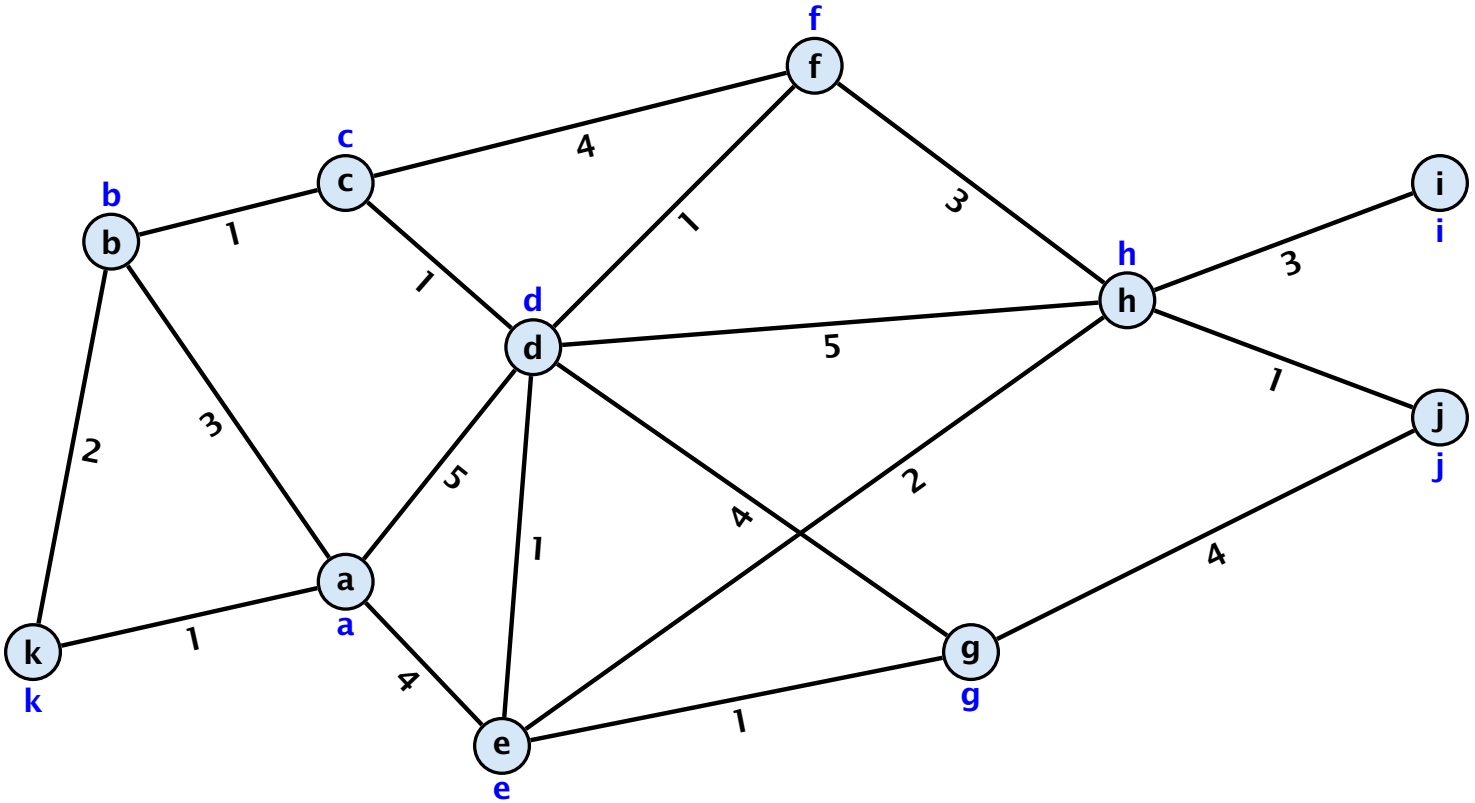
- Verwaltung von Zusammenhangskomponenten in einem dynamischen Graphen, der sich durch das Einfügen von Knoten und Kanten ändert.
- Kruskals Algorithmus für minimale Spannbäume.

# 12 Union Find

**Anwendungen:**

- Verwaltung von Zusammenhangskomponenten in einem dynamischen Graphen, der sich durch das Einfügen von Knoten und Kanten ändert.
- Kruskals Algorithmus für minimale Spannbäume.

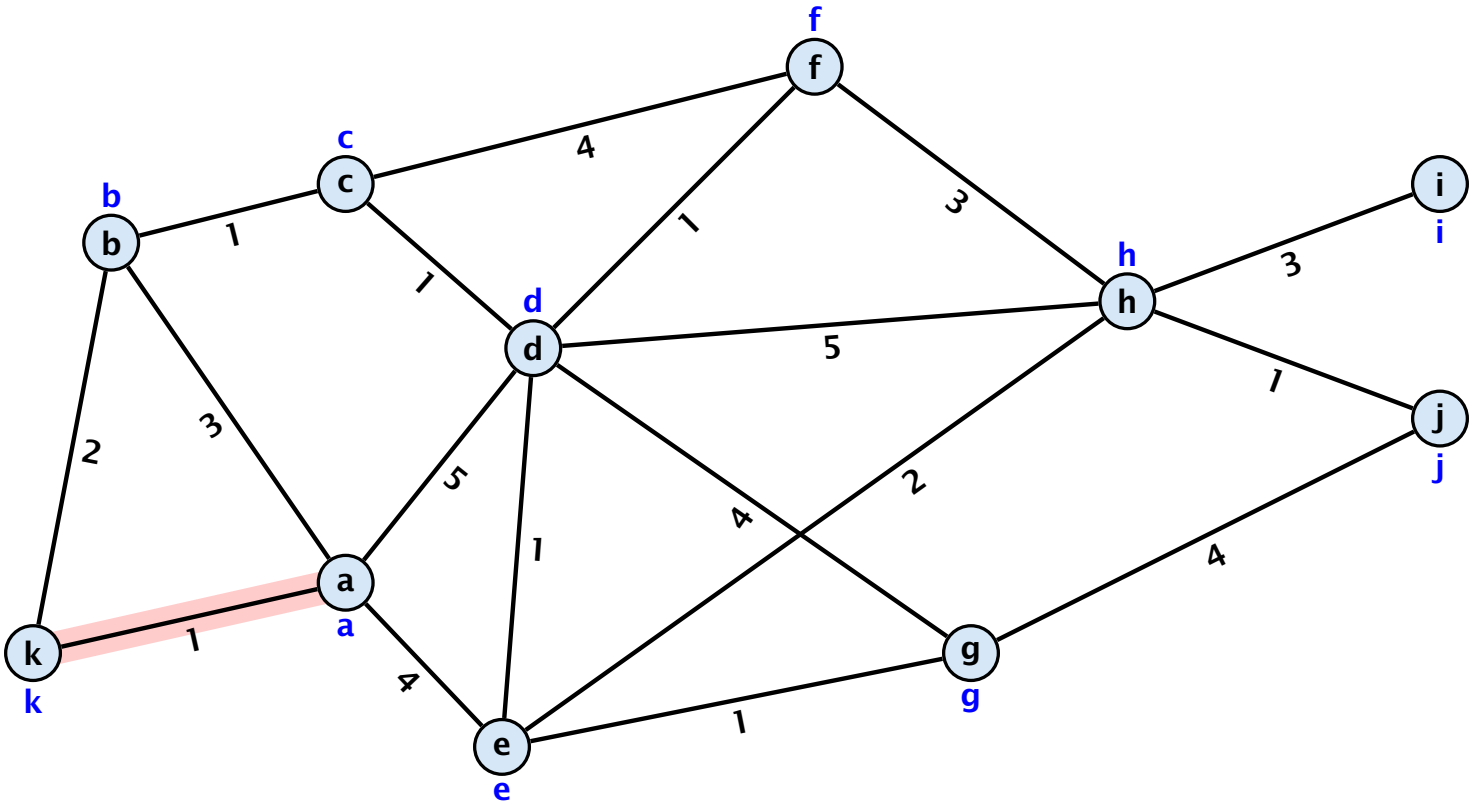
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

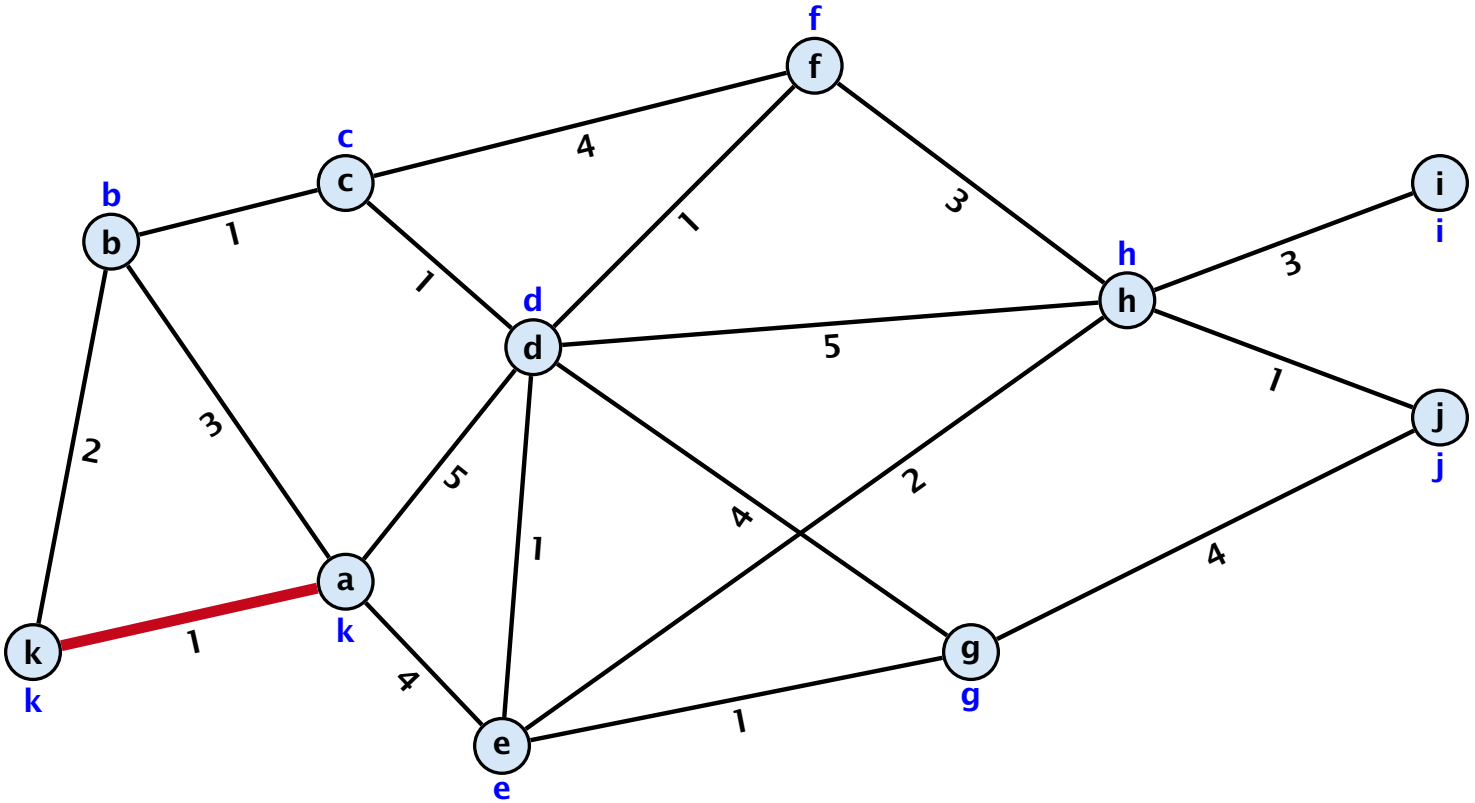
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

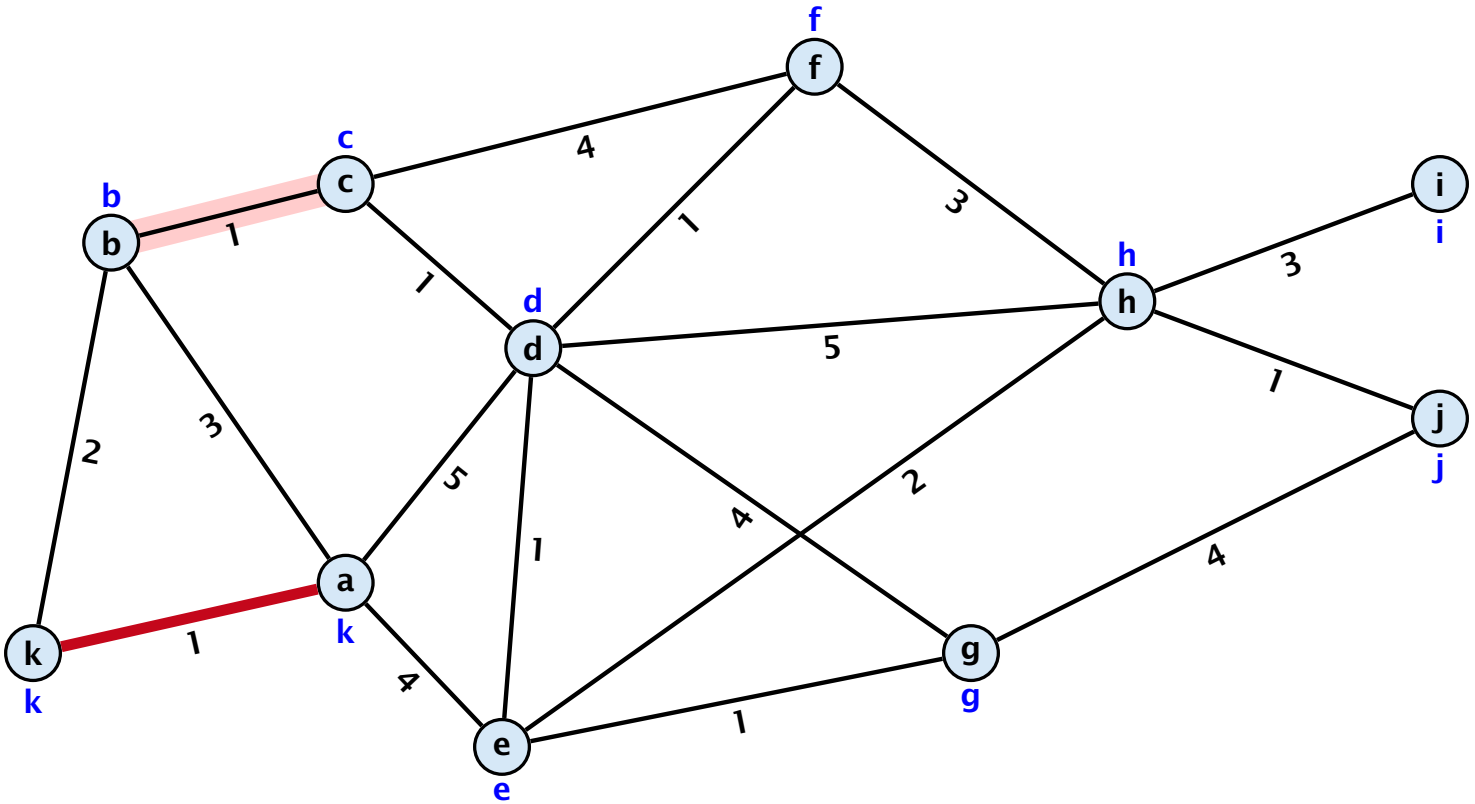
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

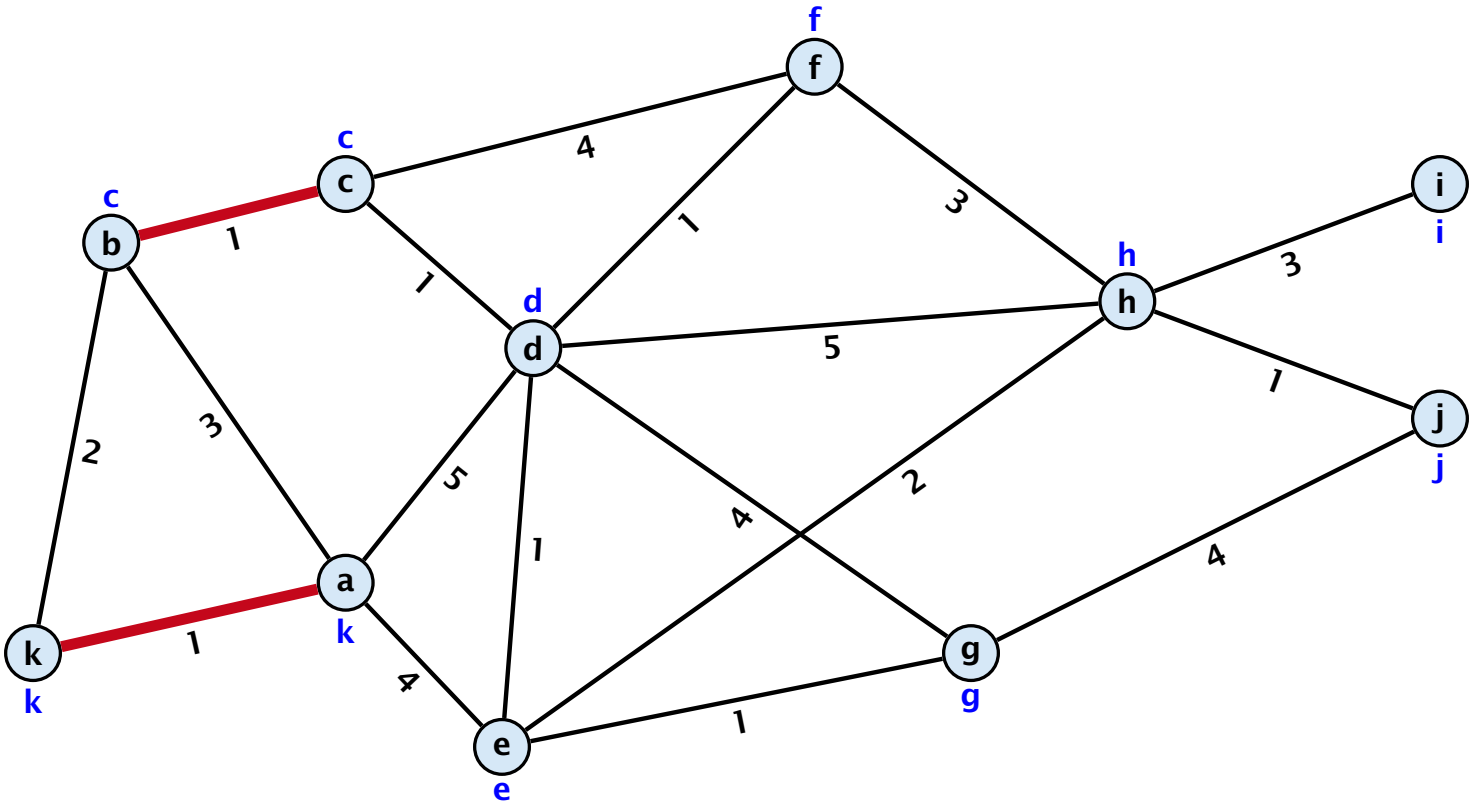
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

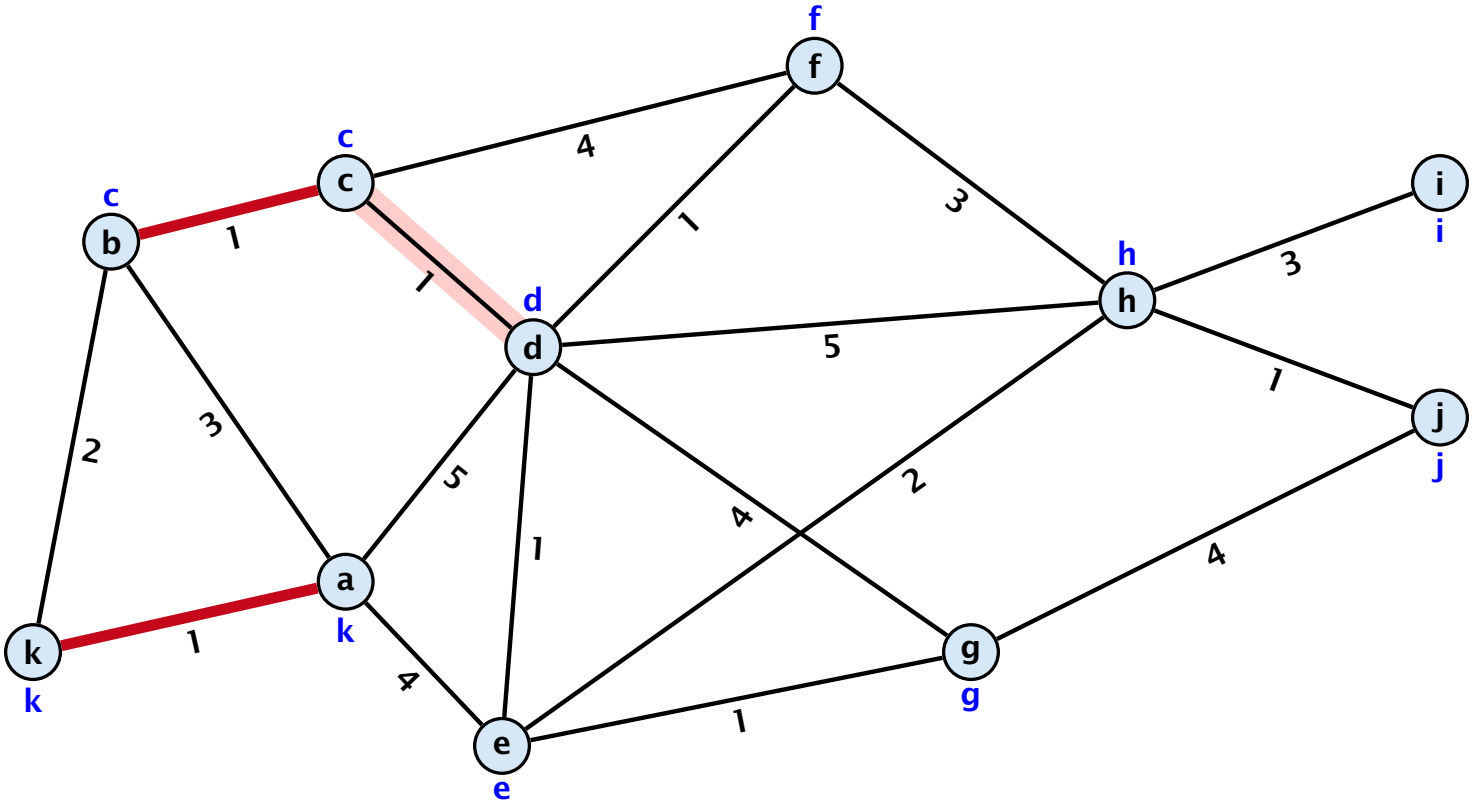
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

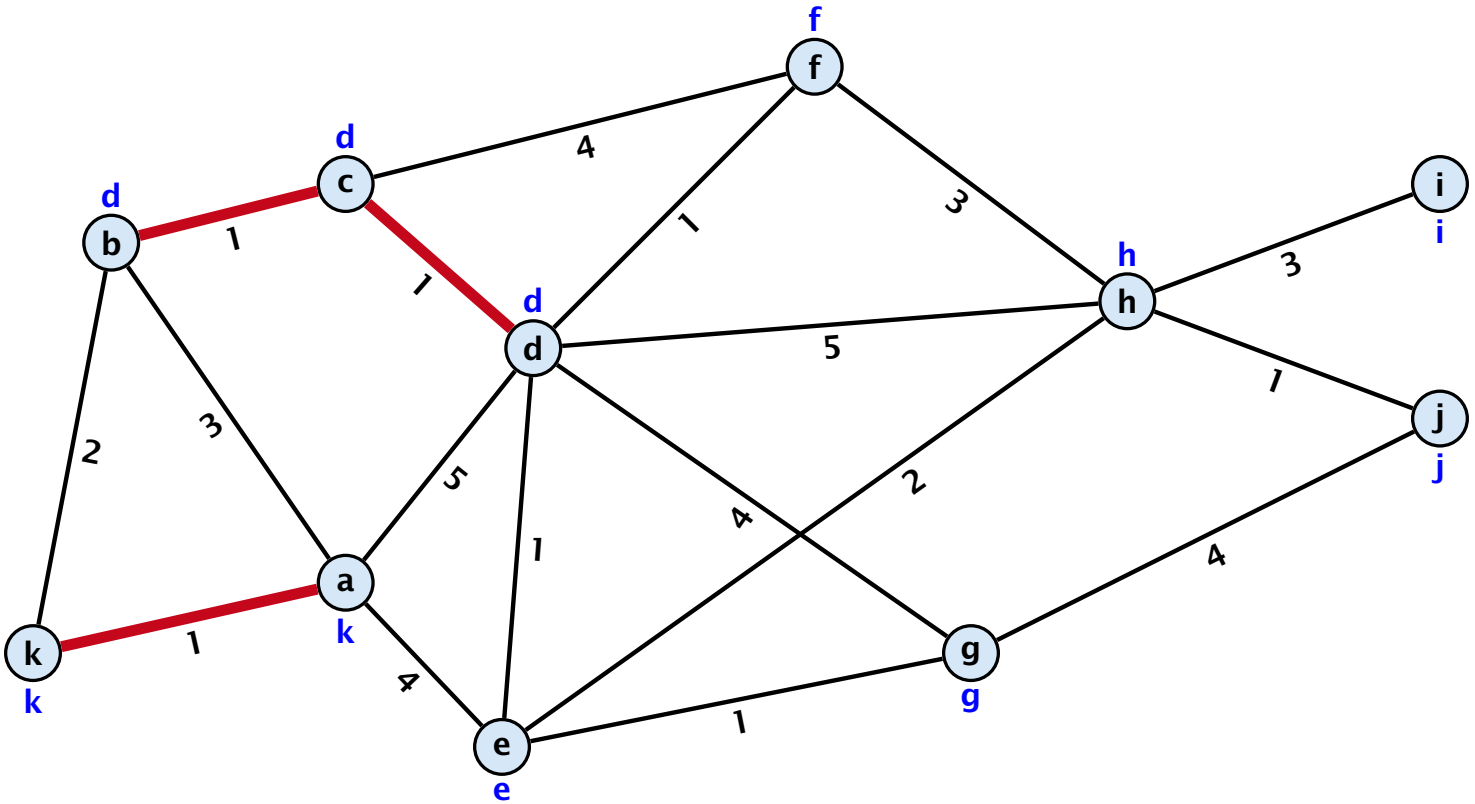
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

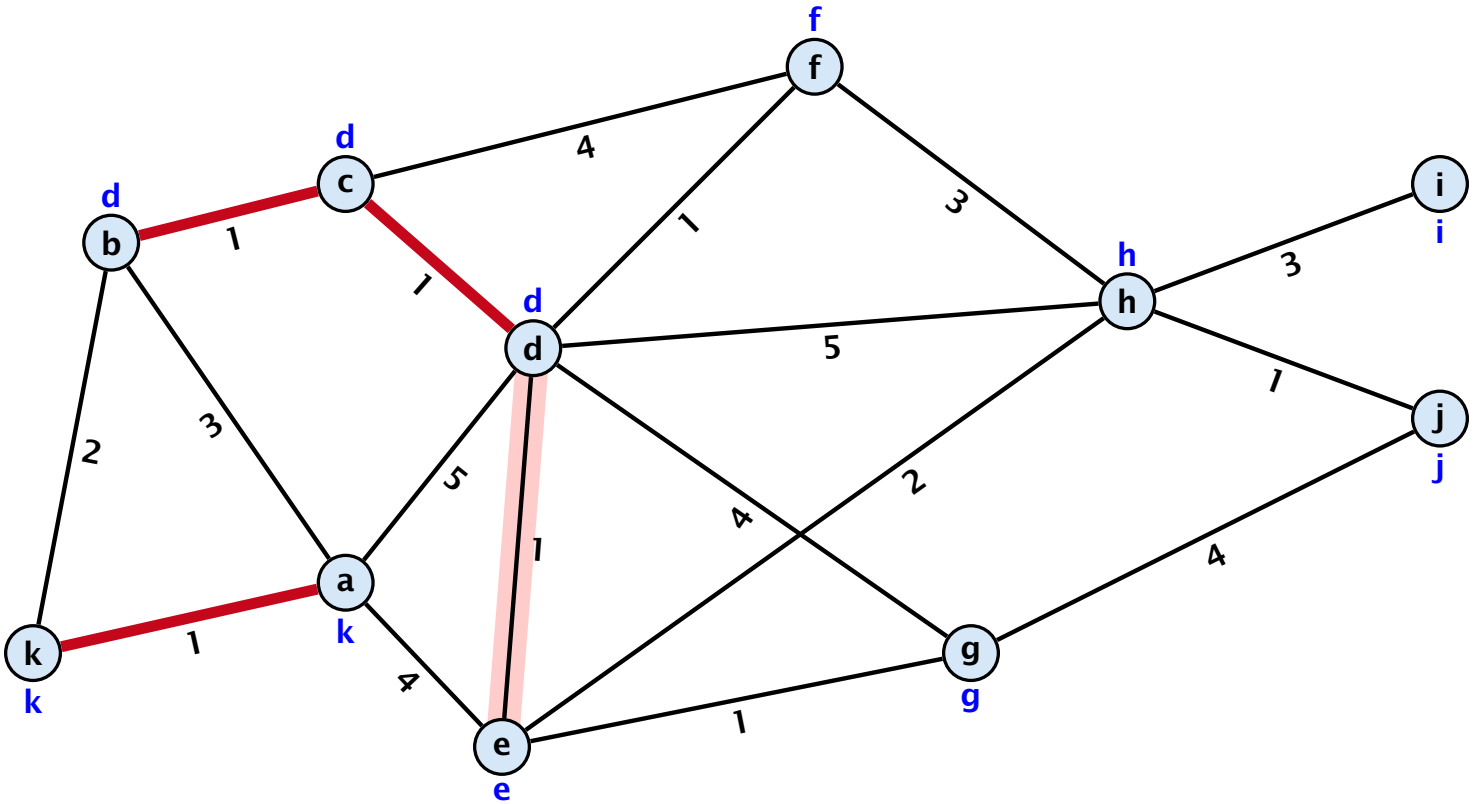
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

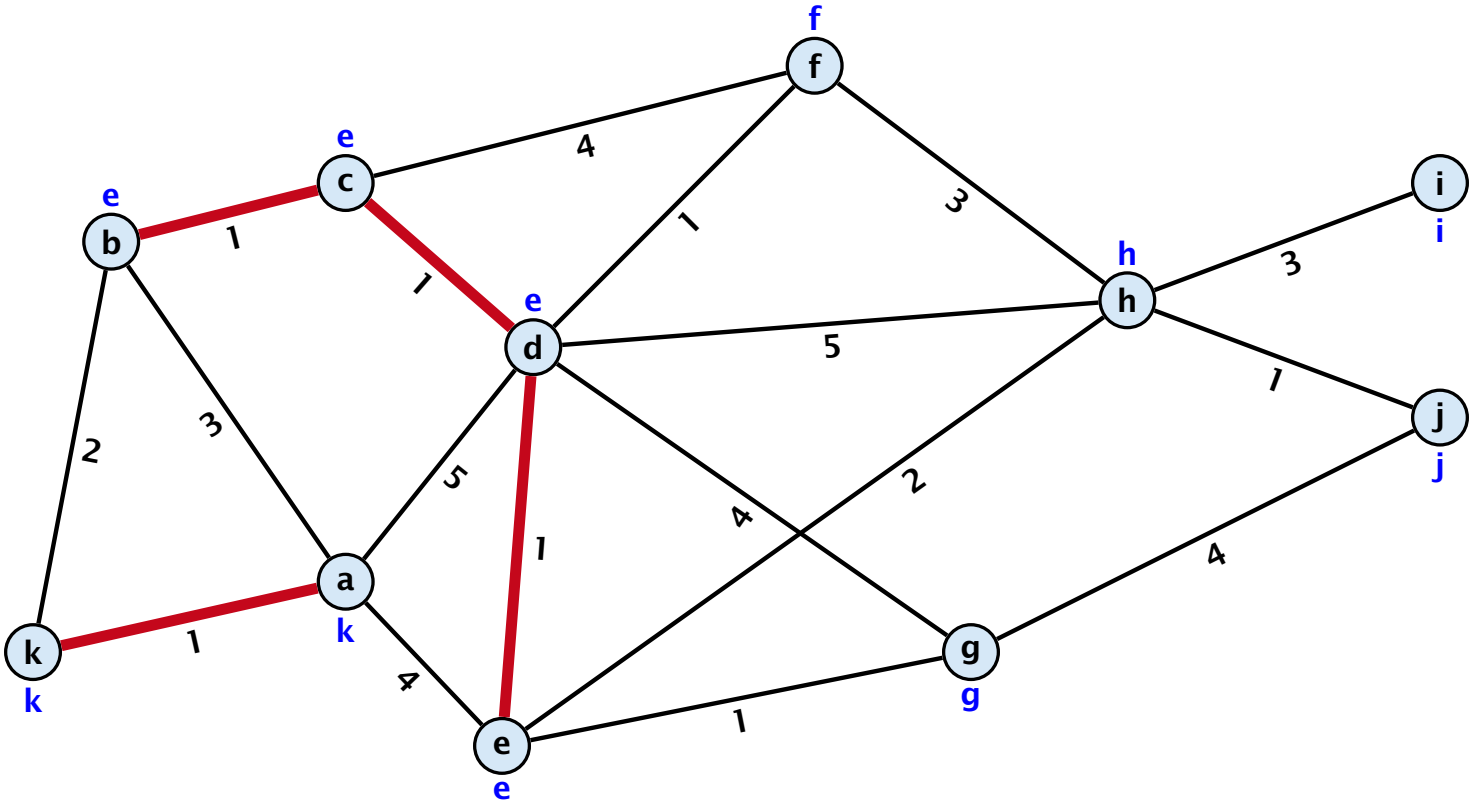
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

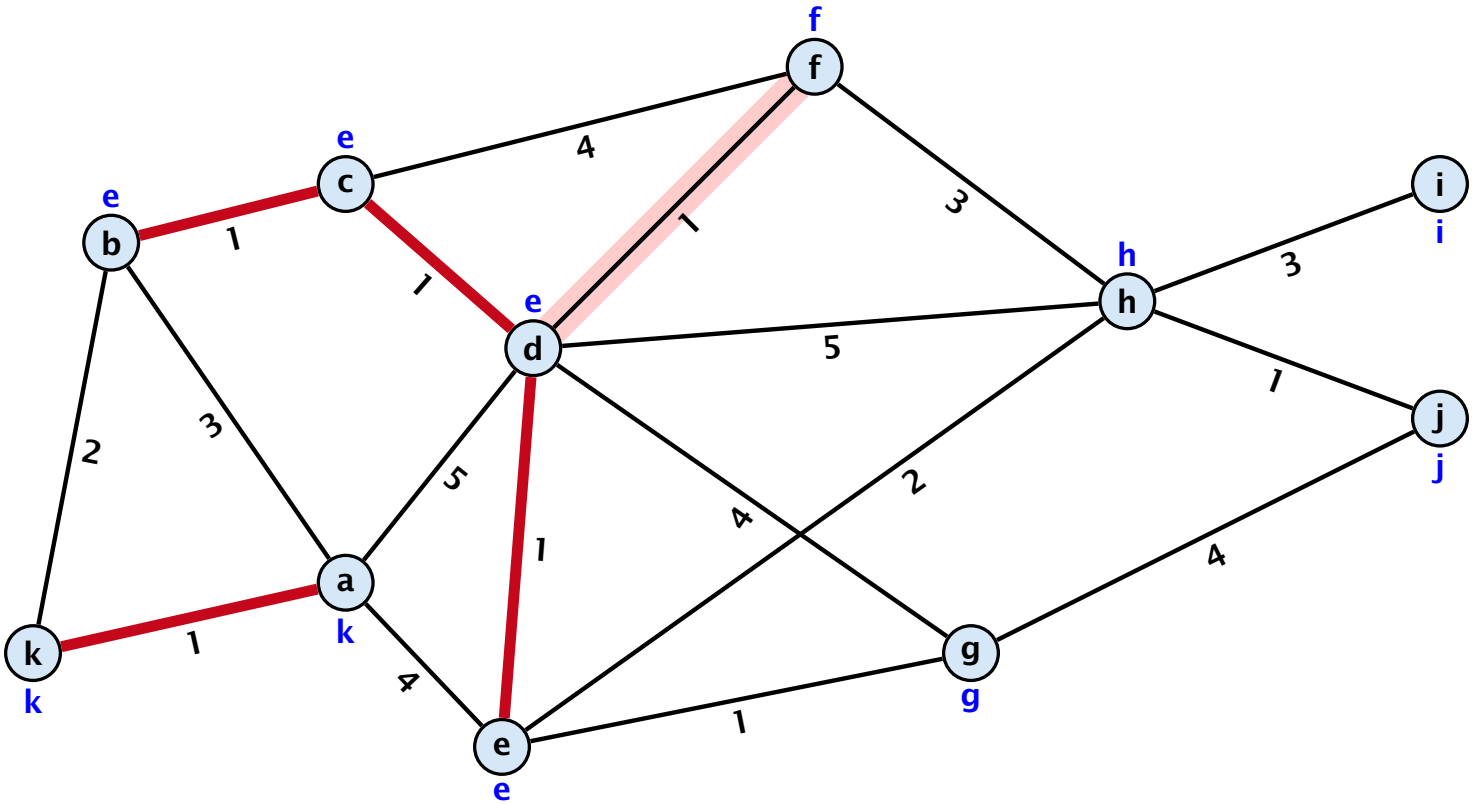
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

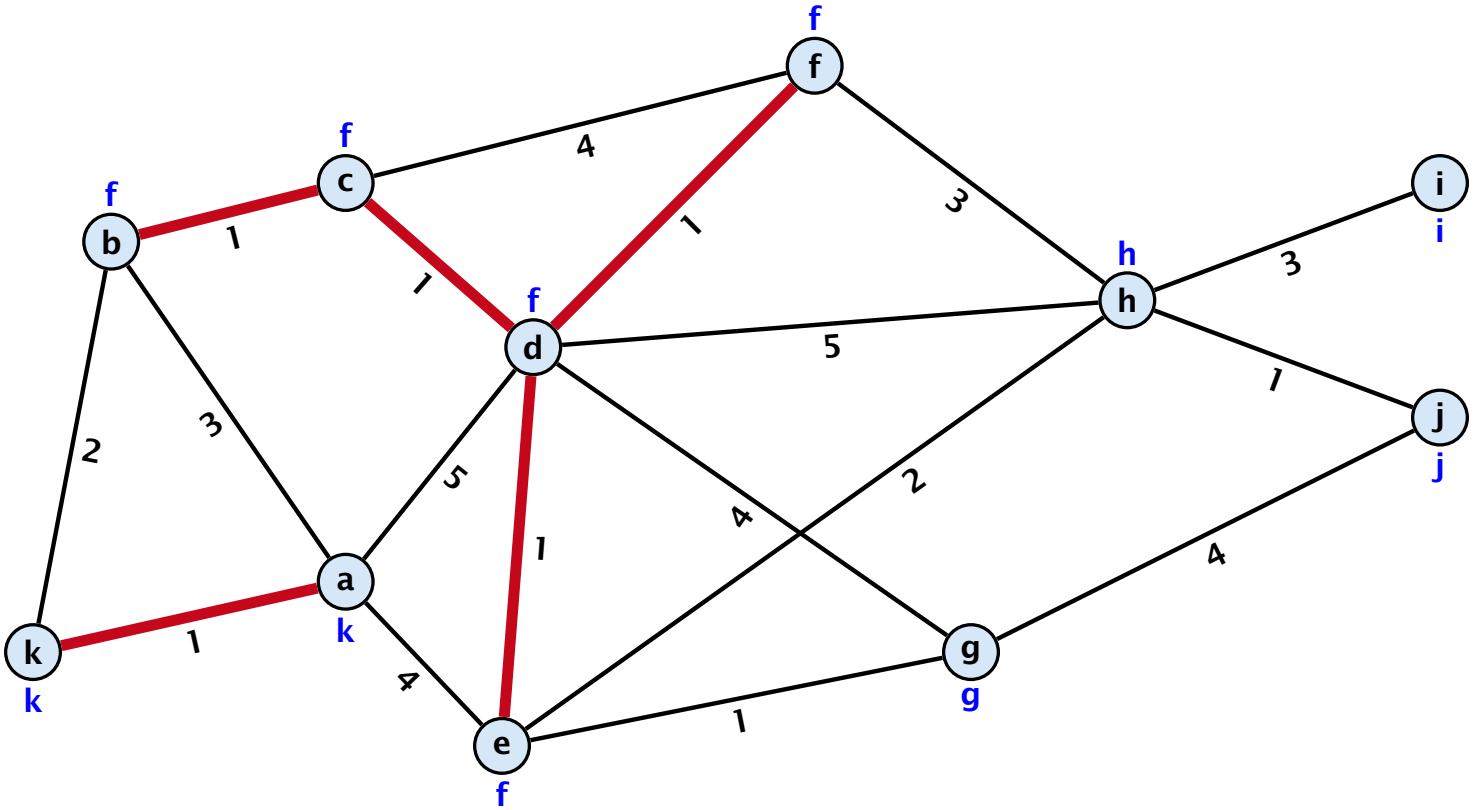
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

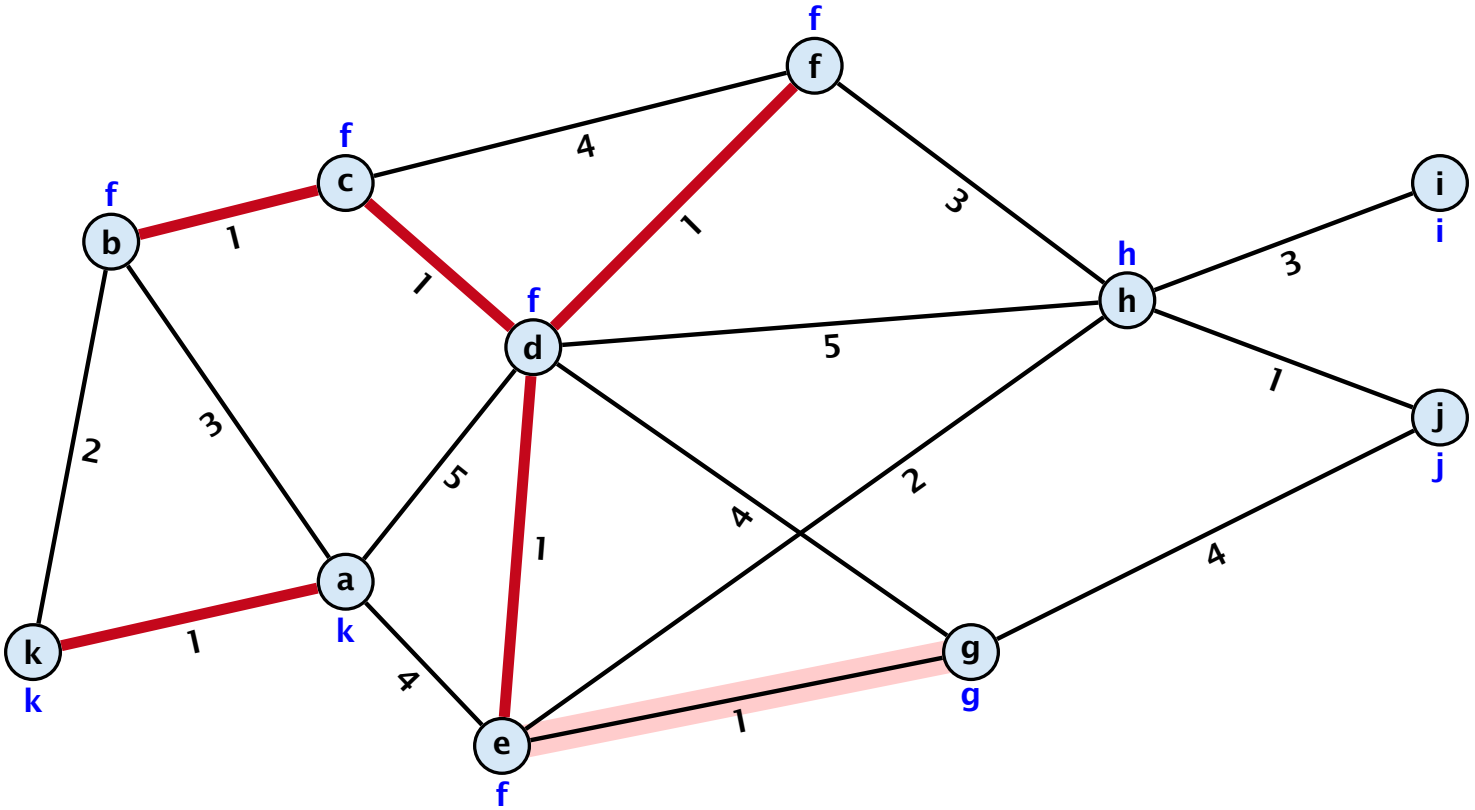
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

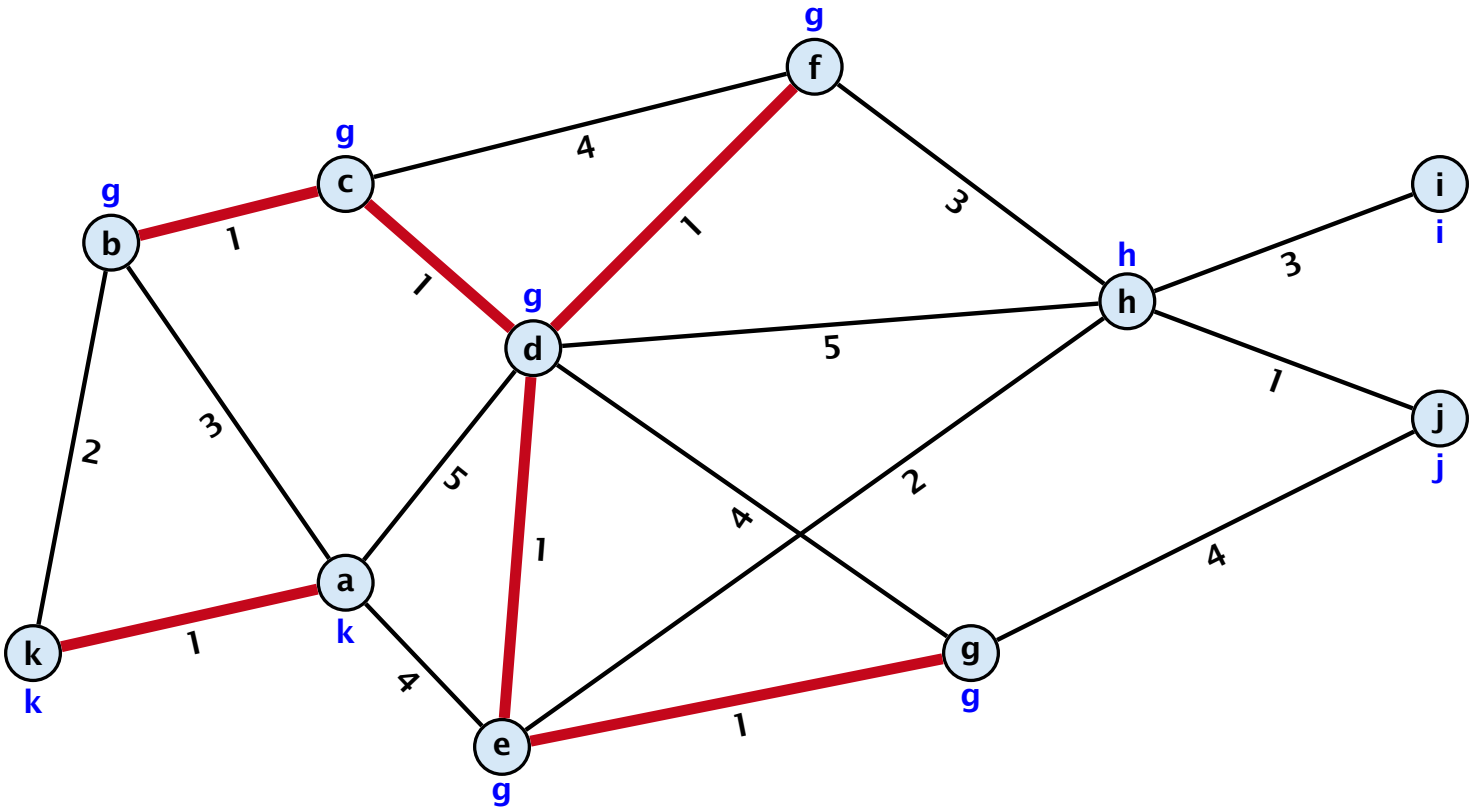
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

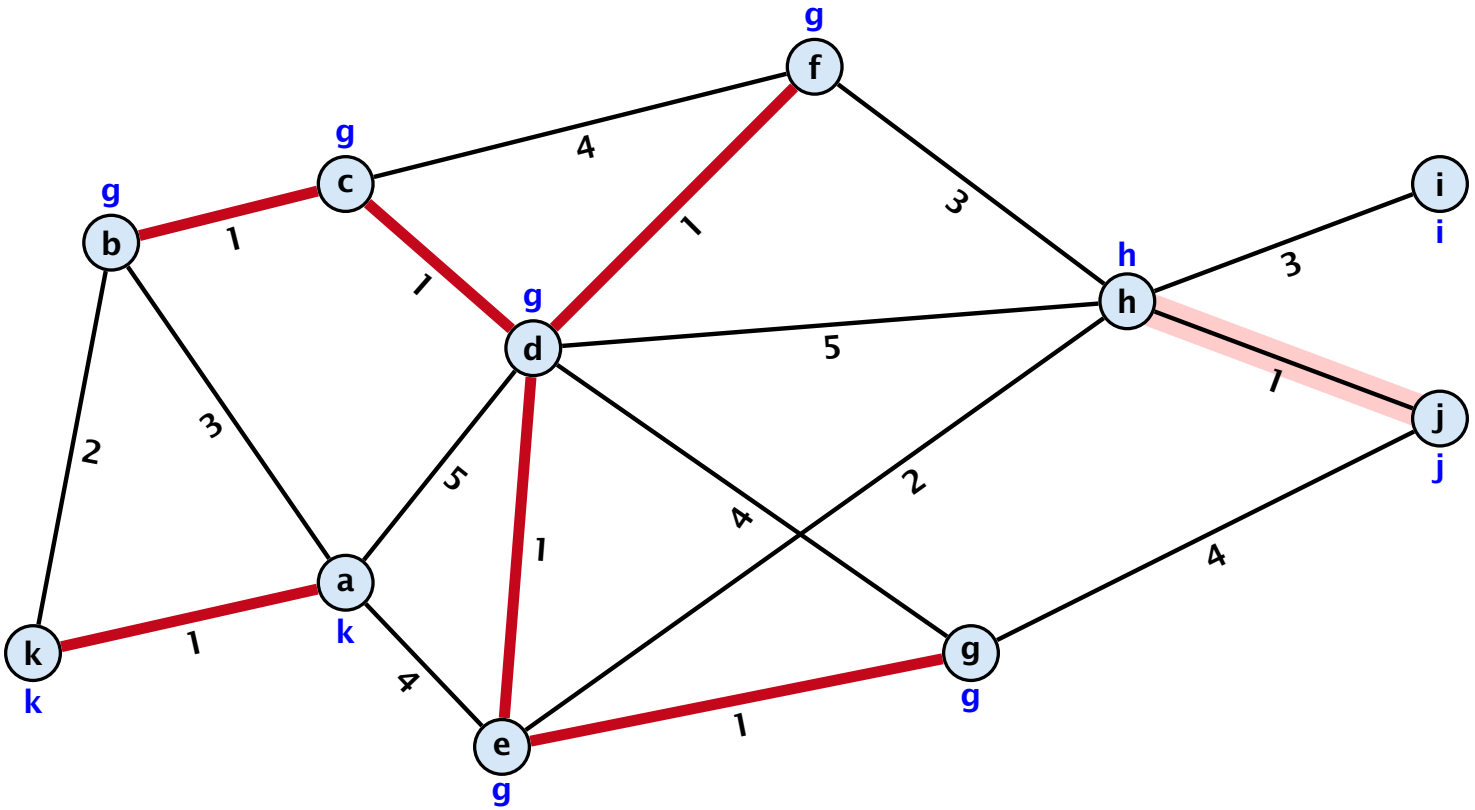
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

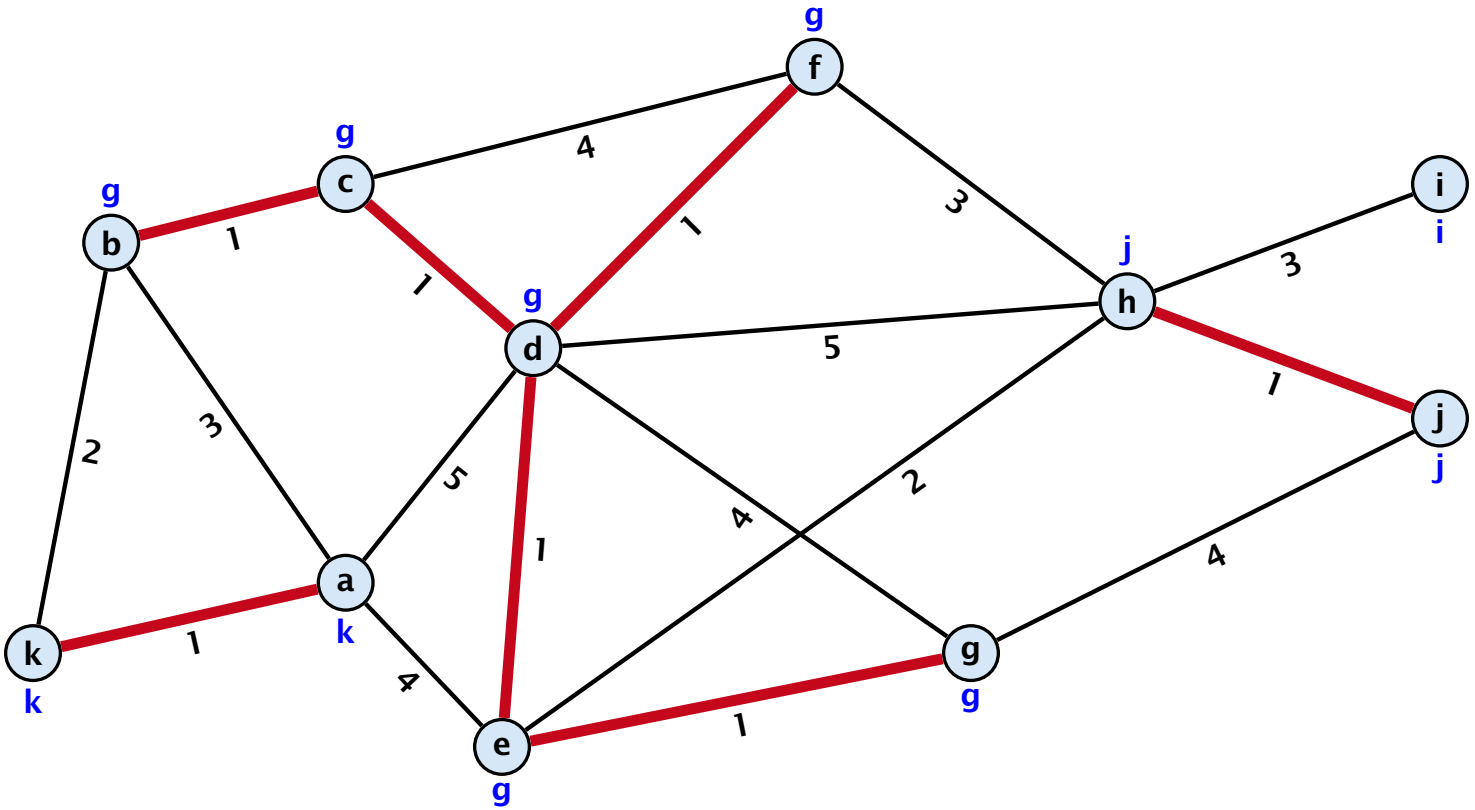
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

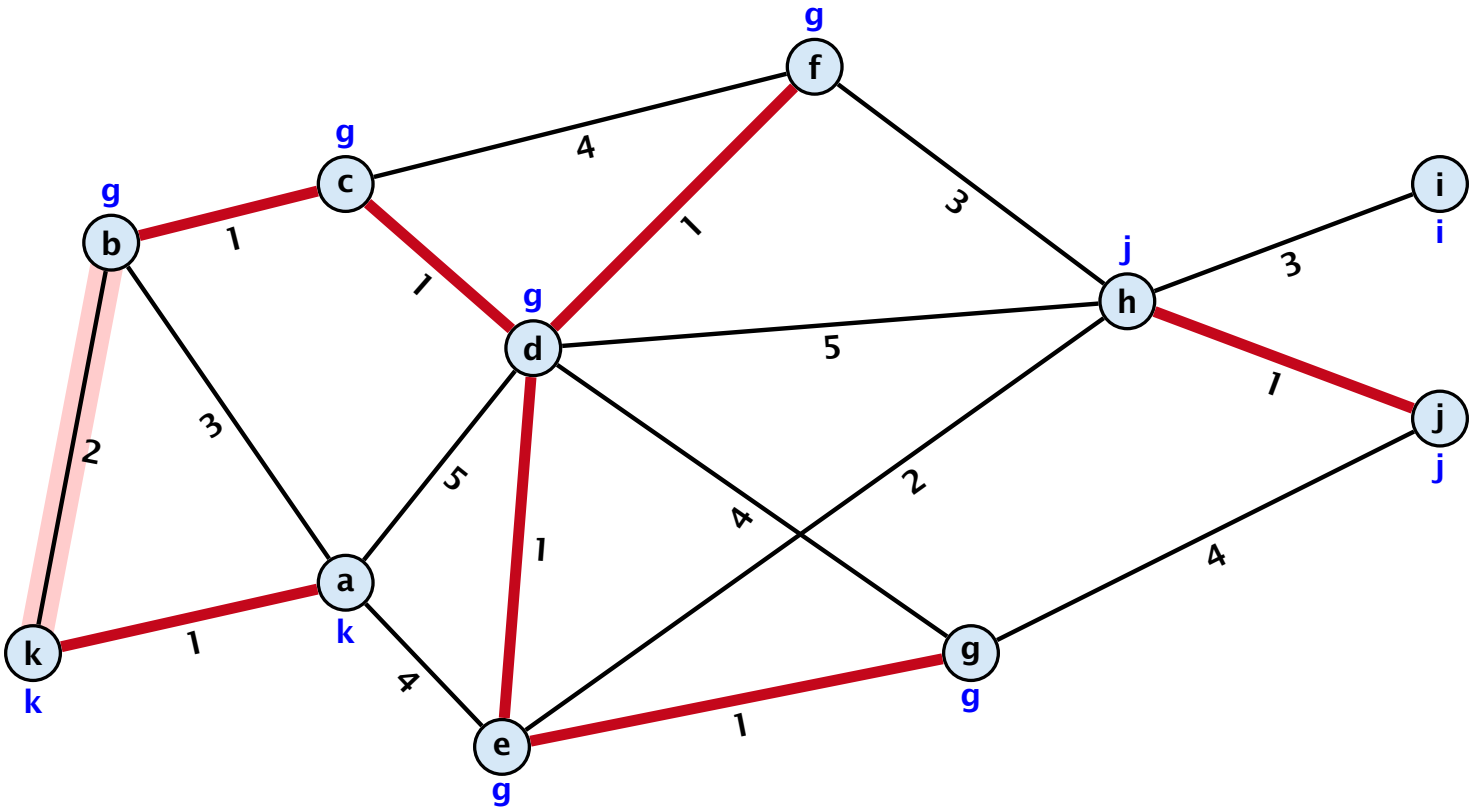
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

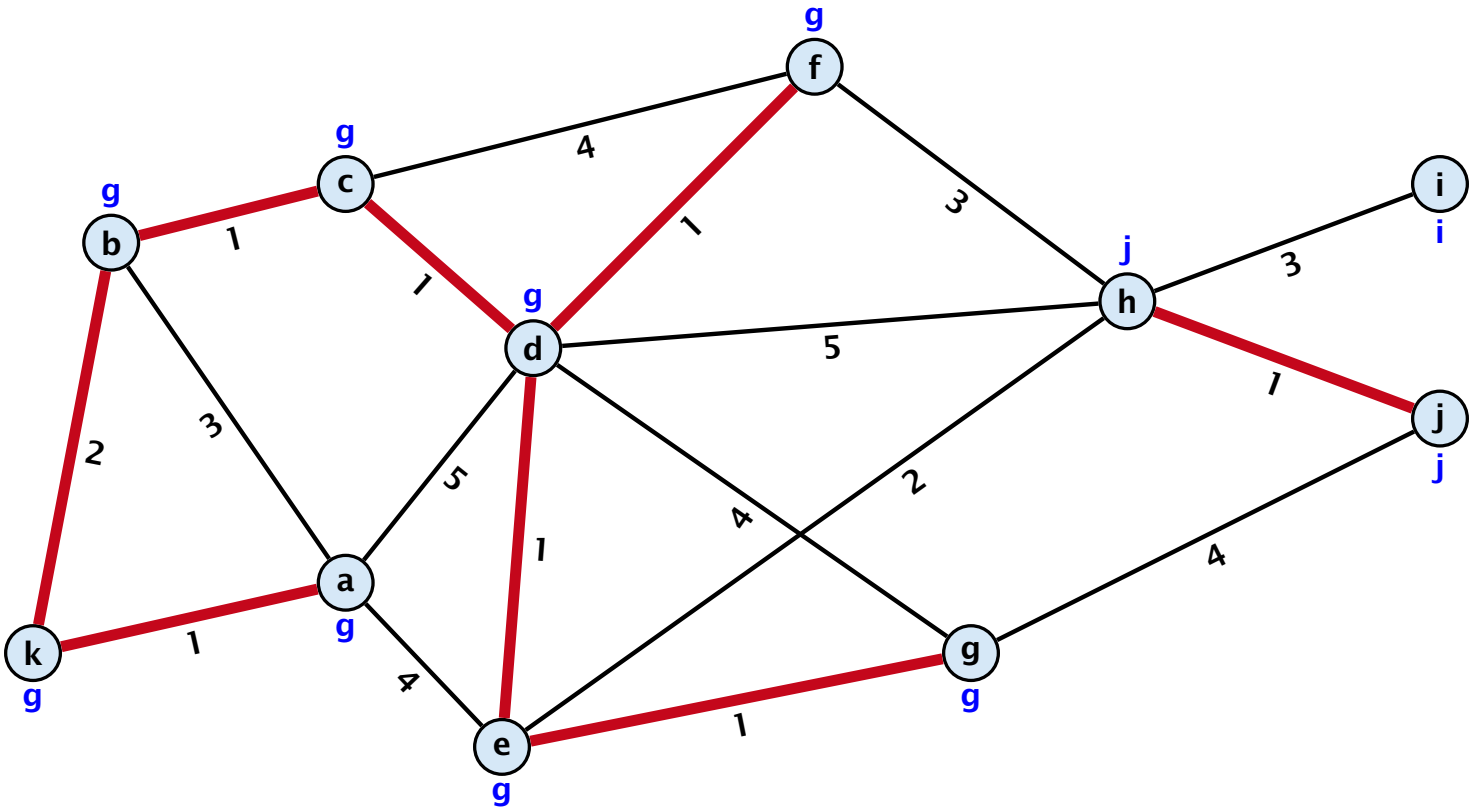
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

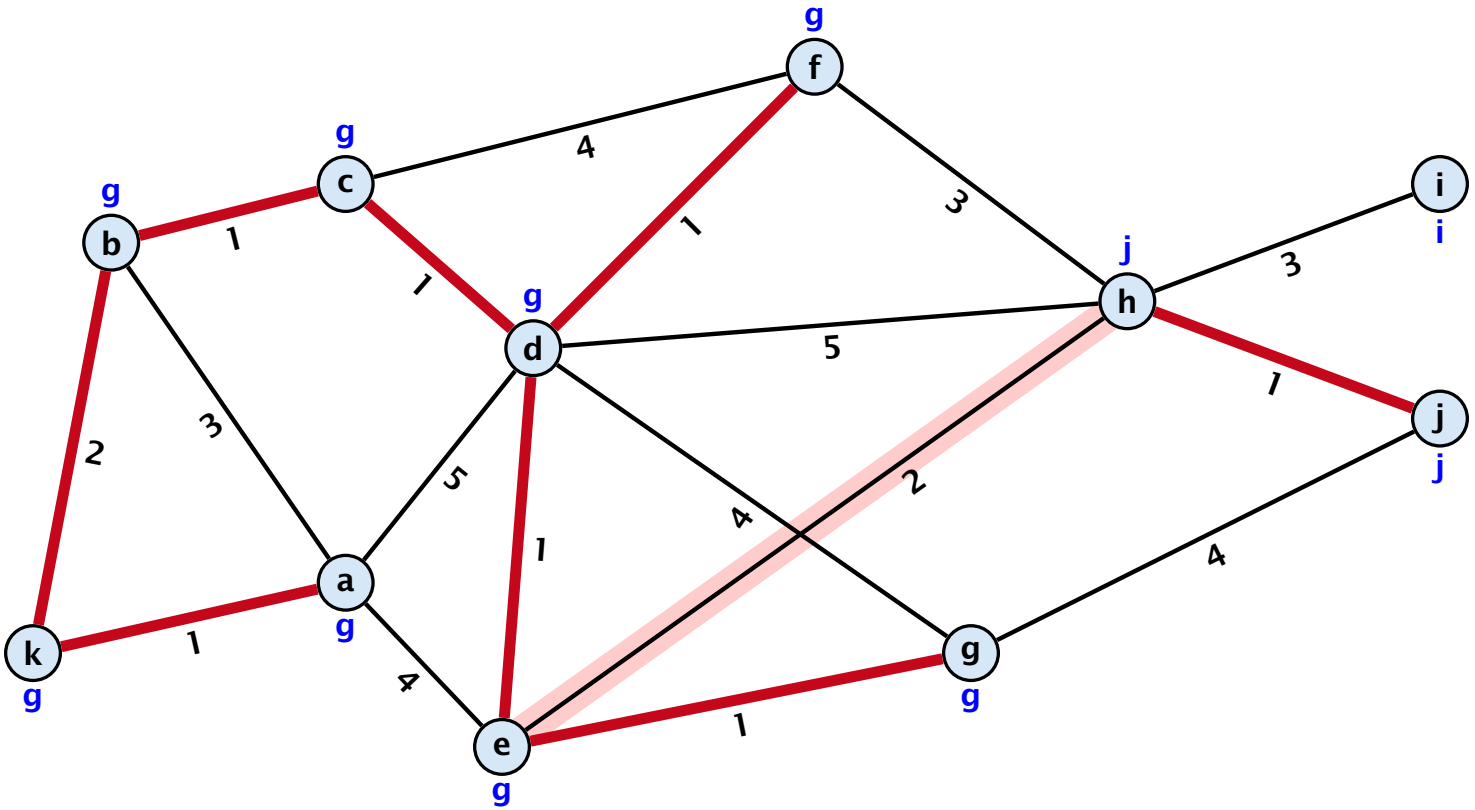
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

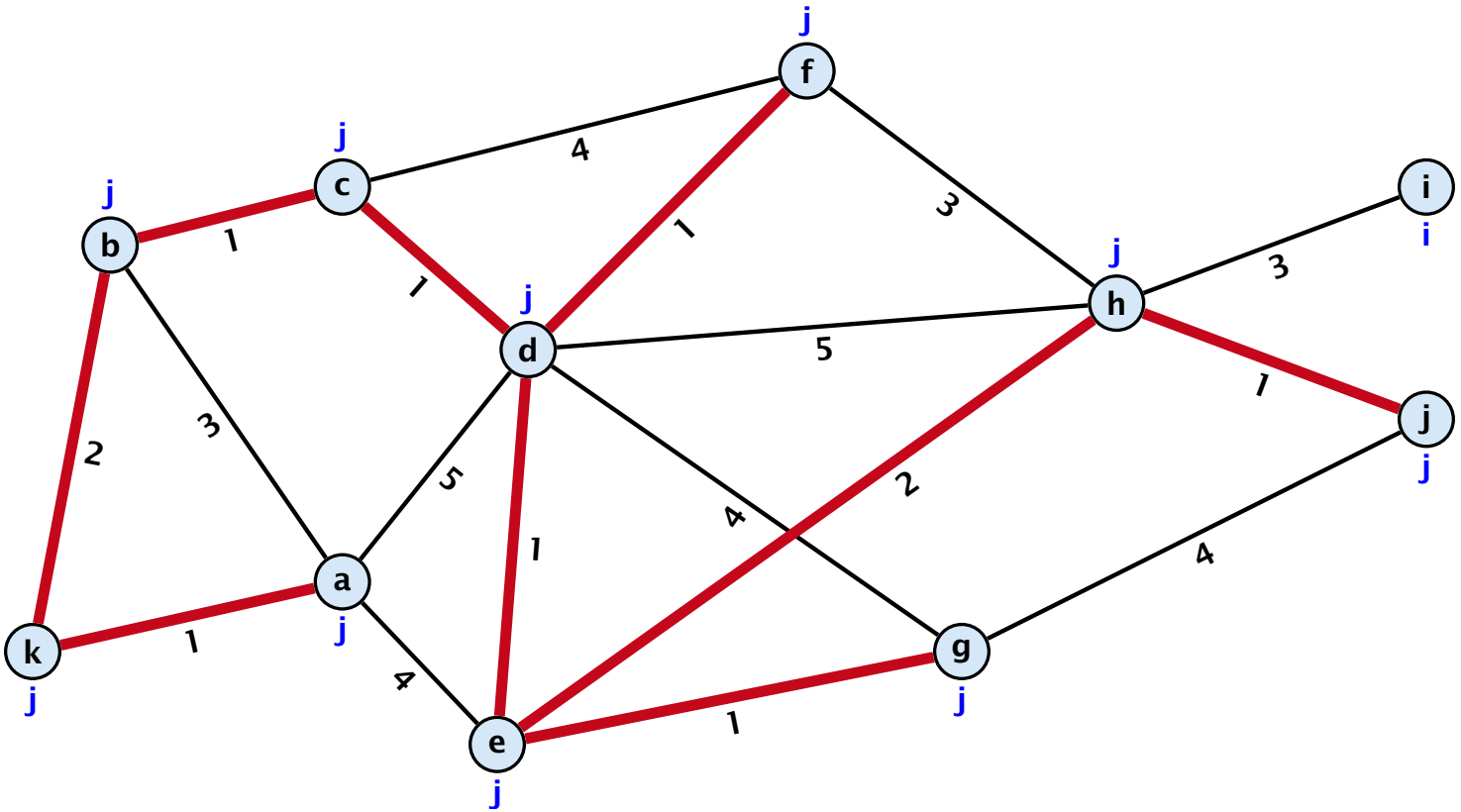
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

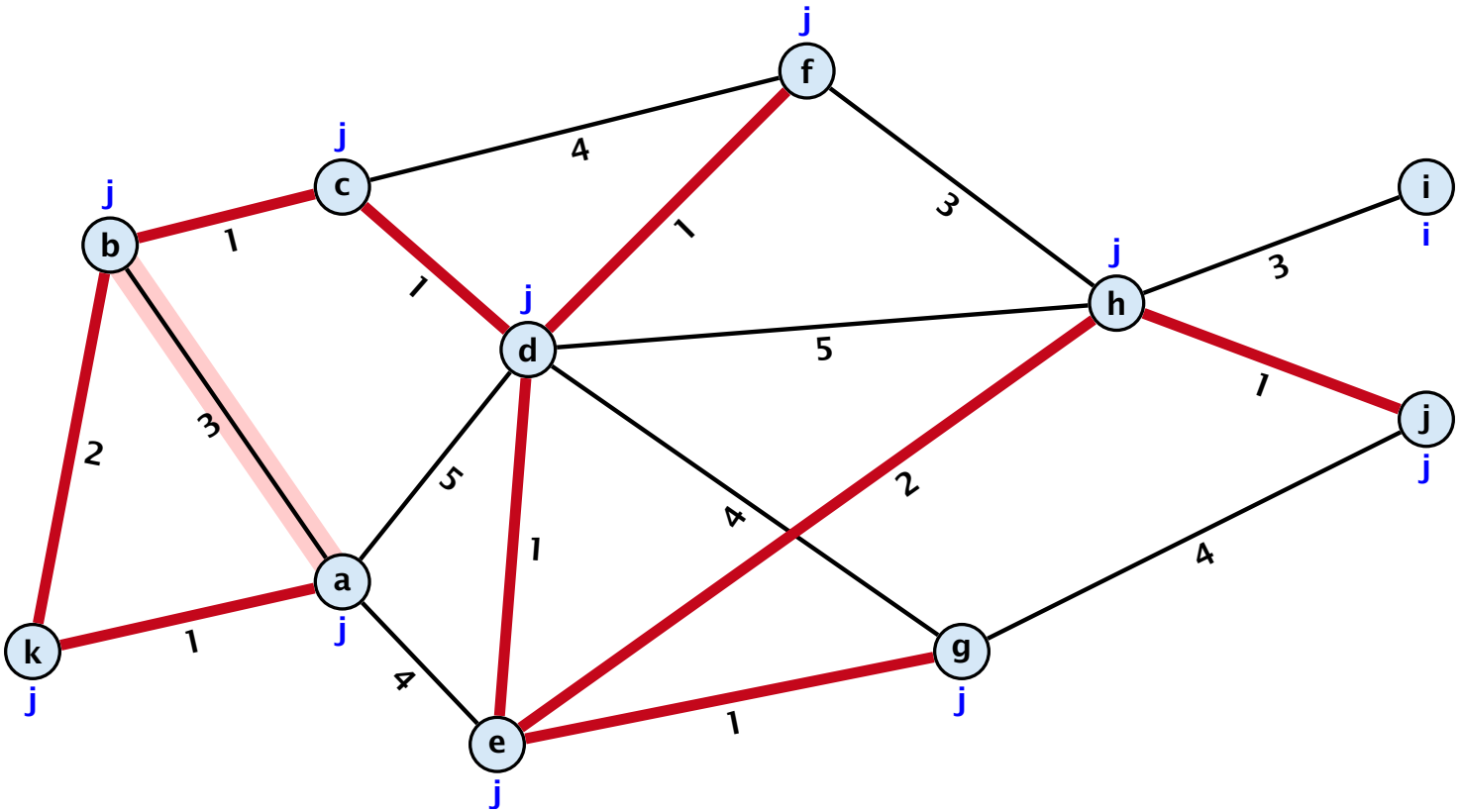
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

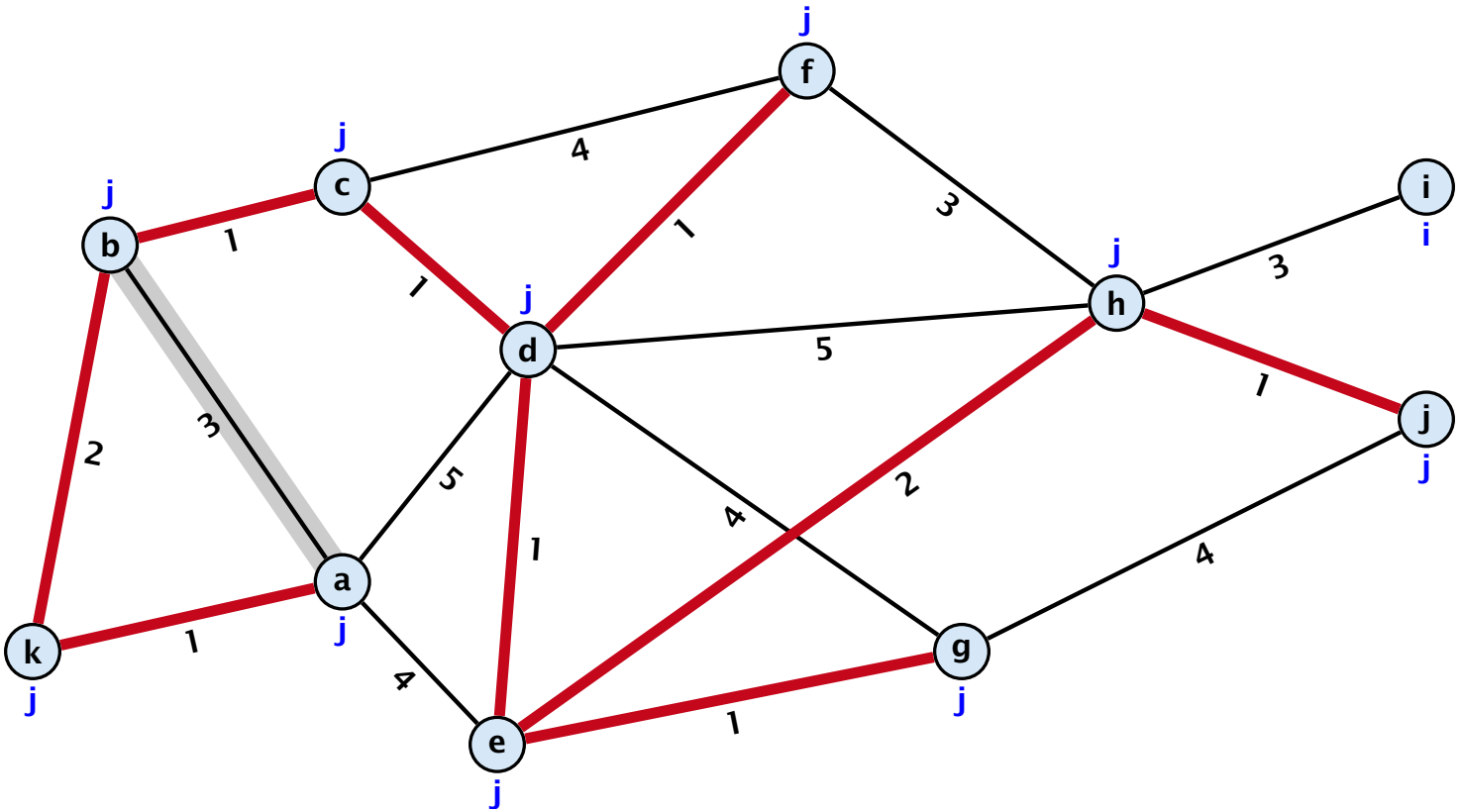
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

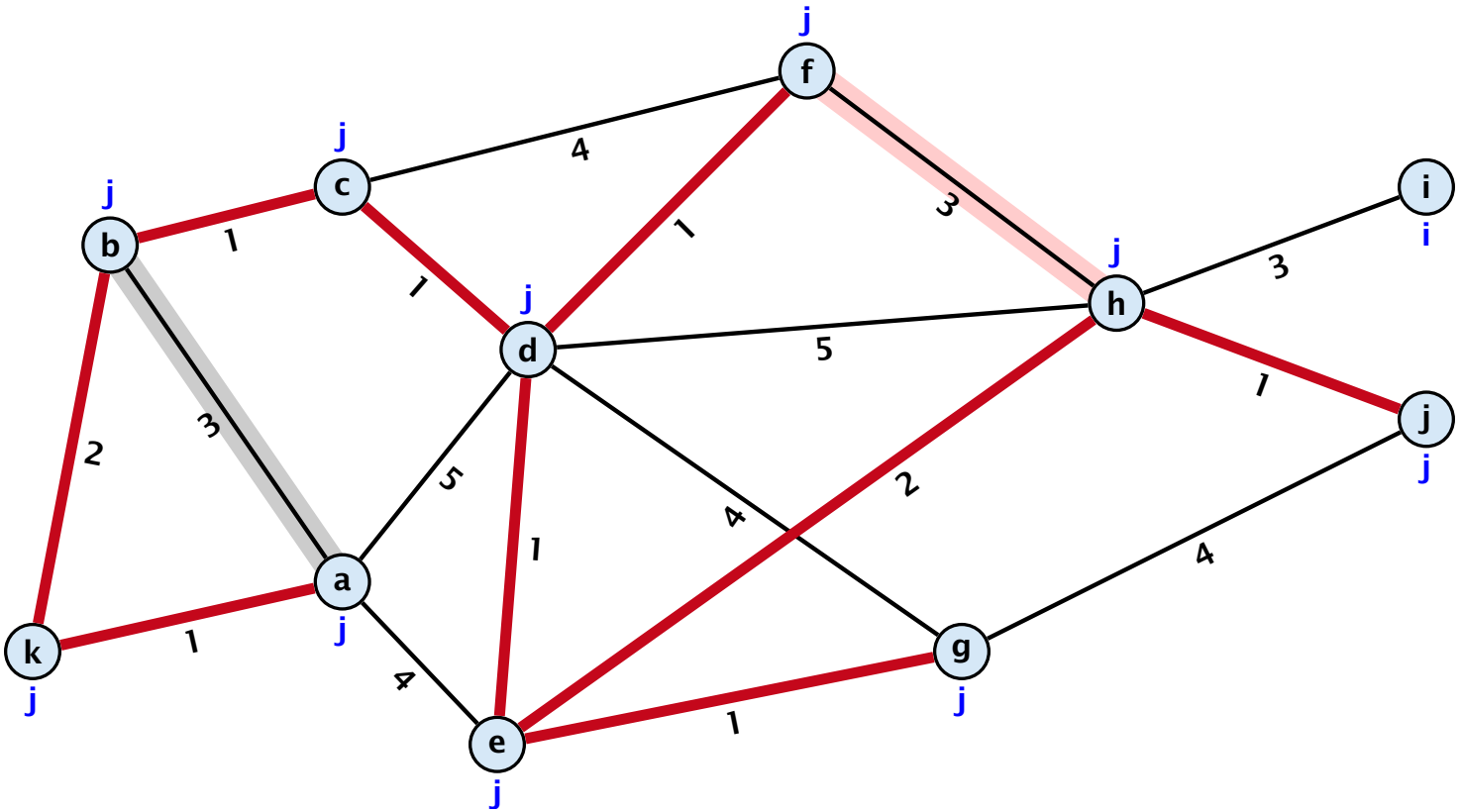
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

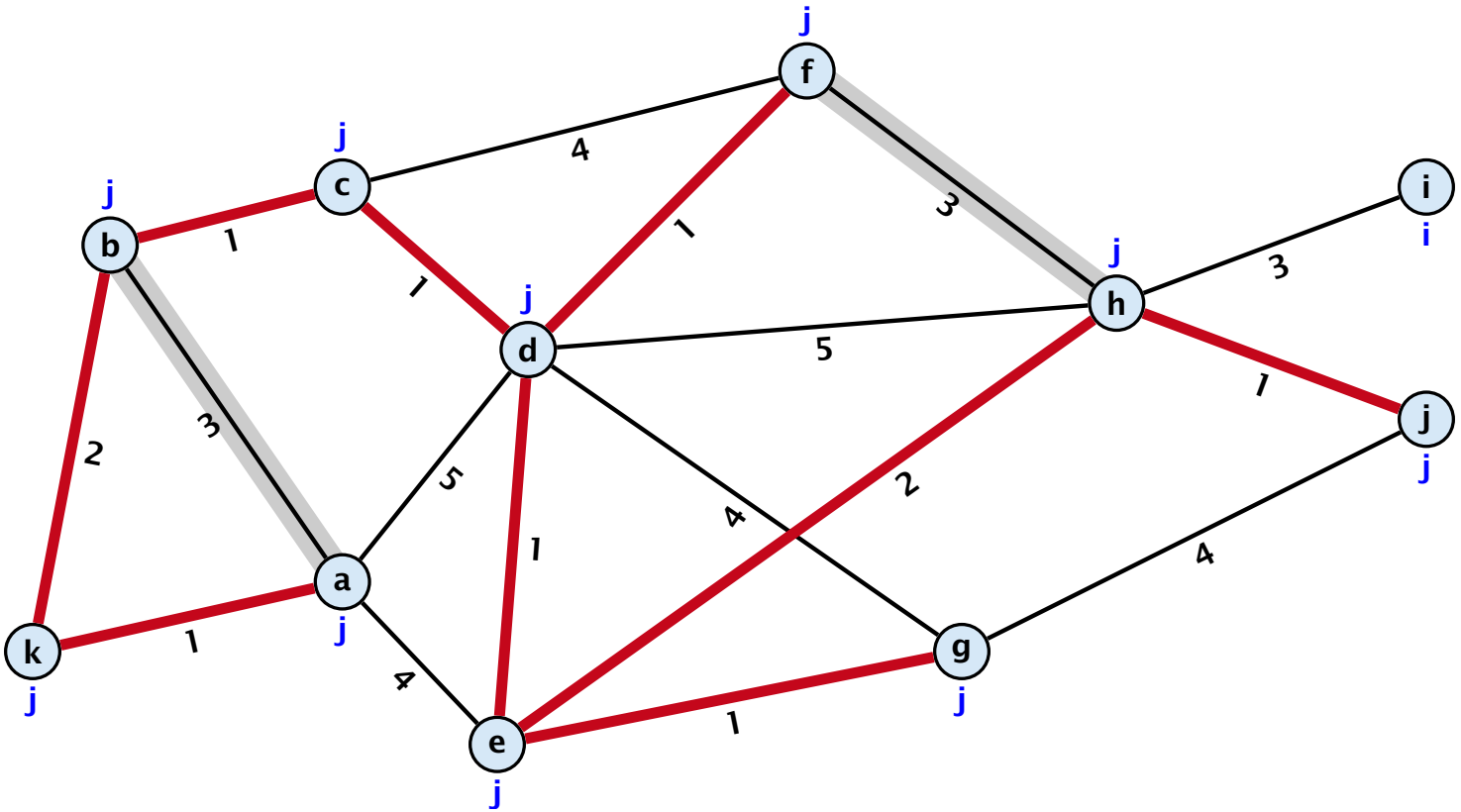
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

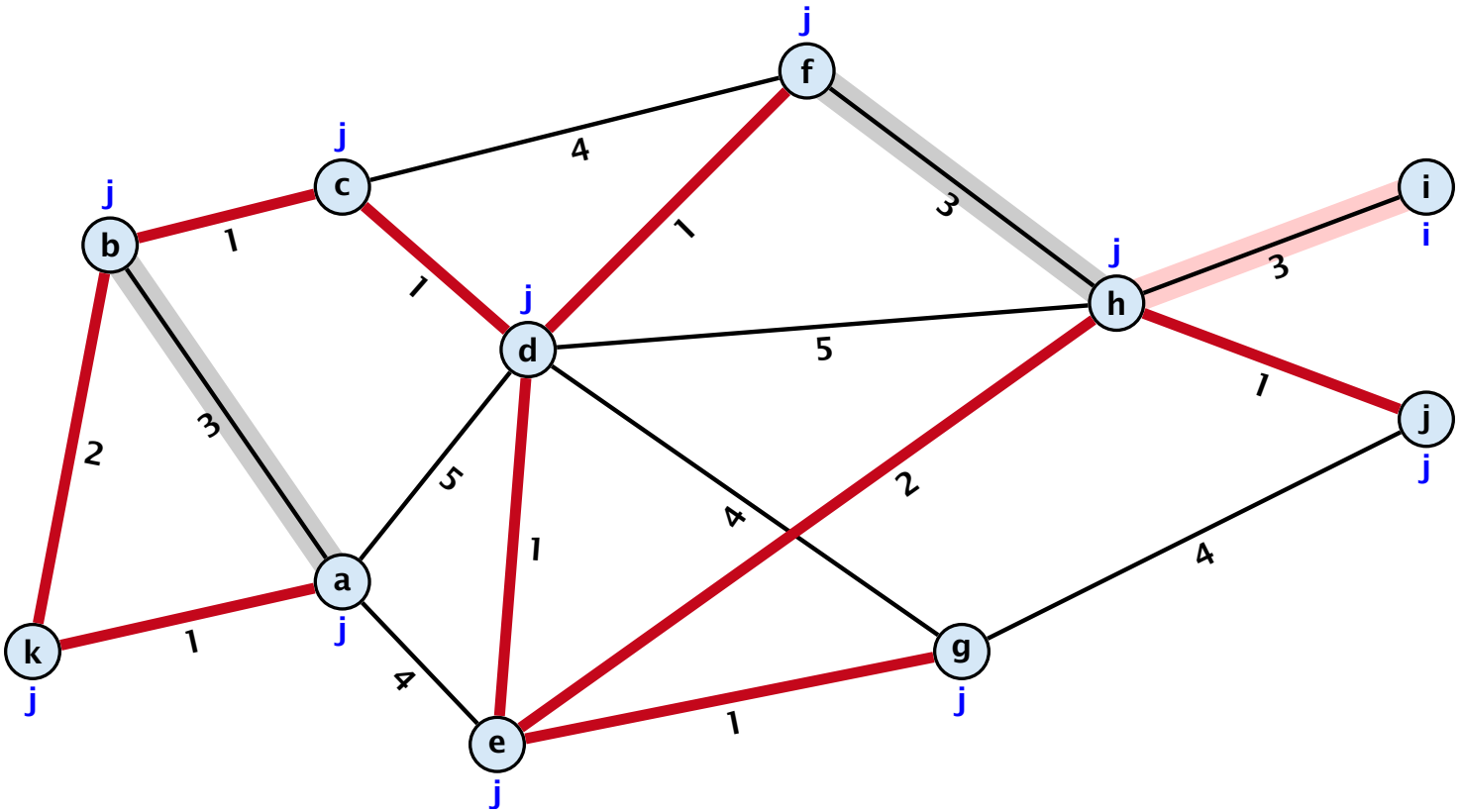
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

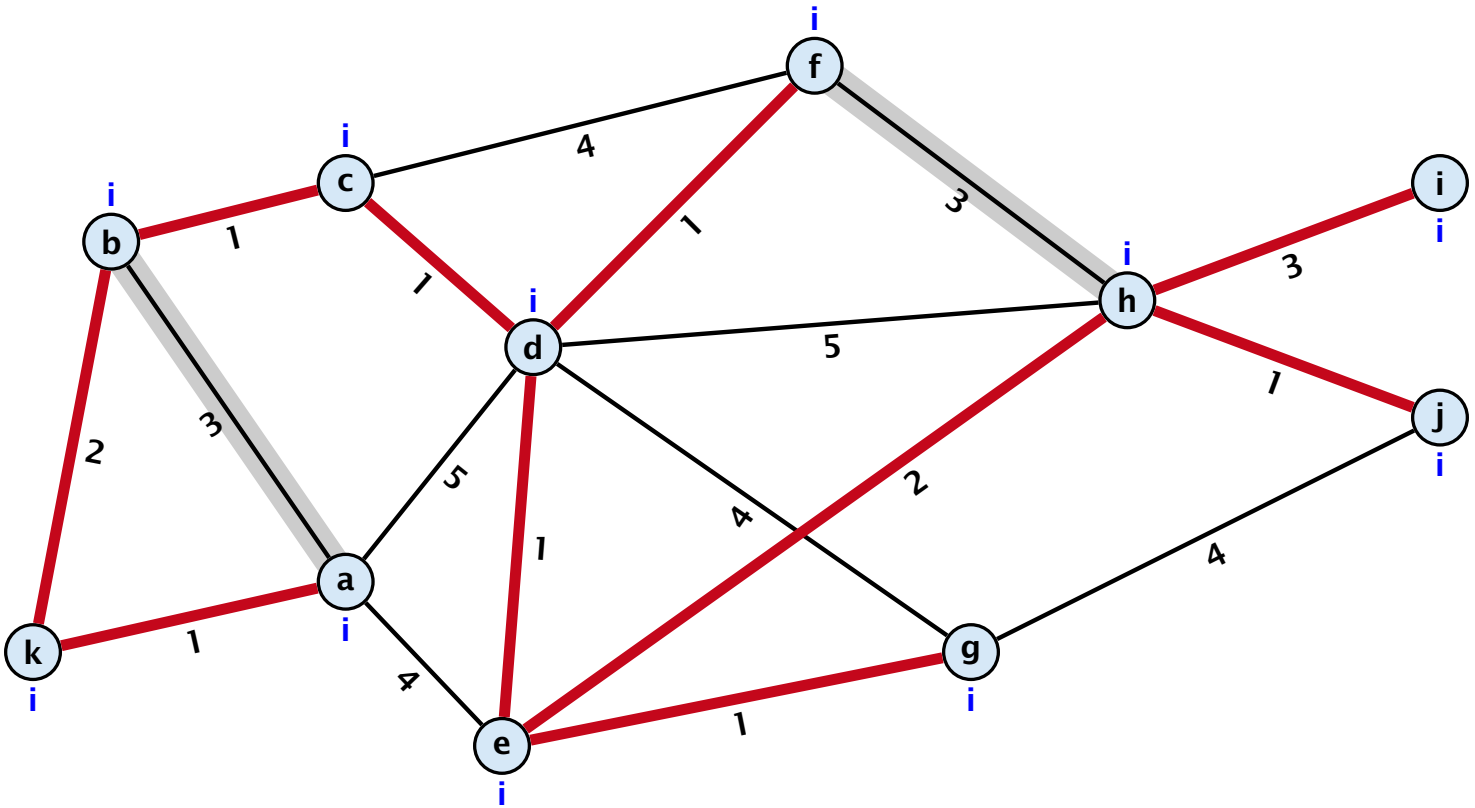
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

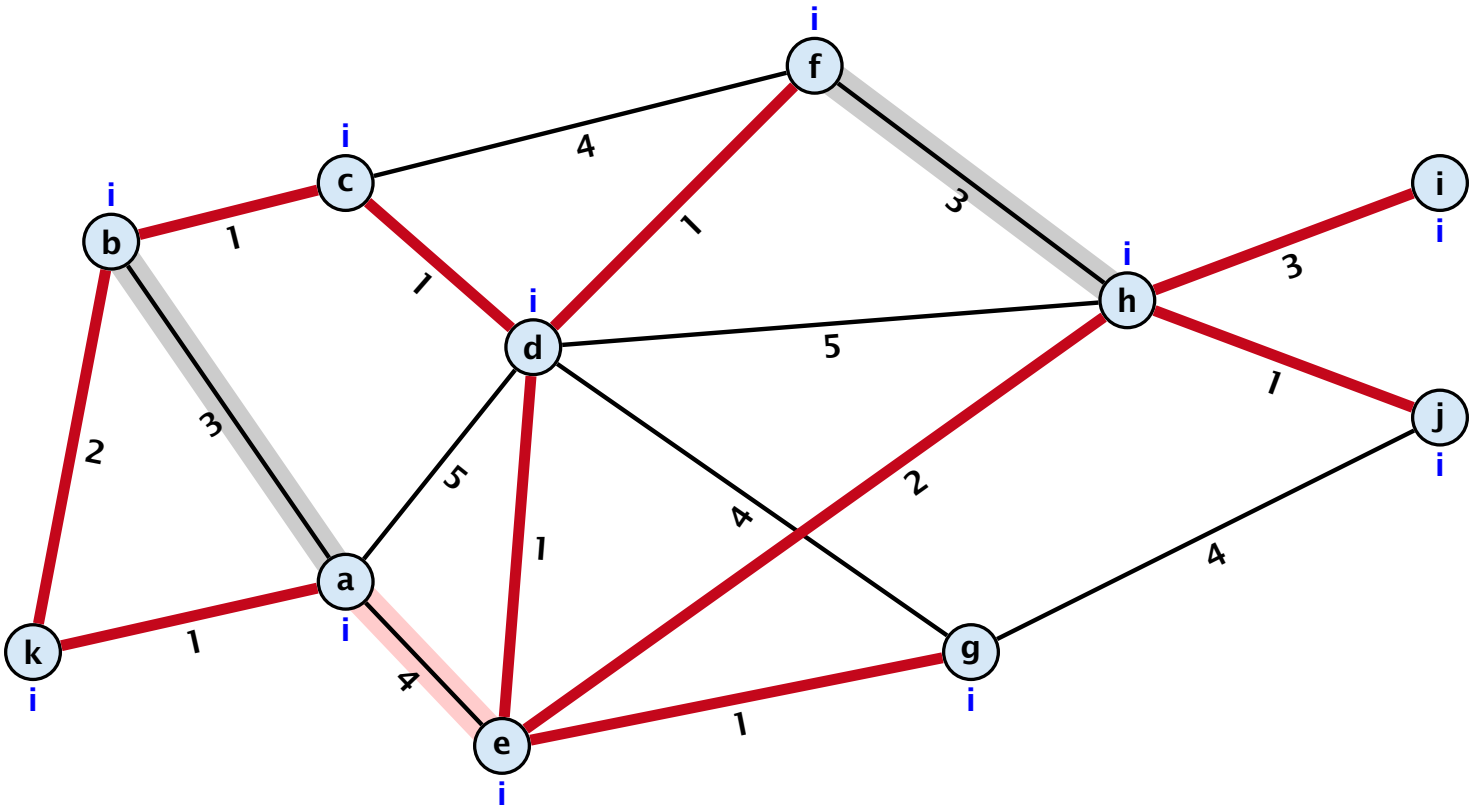
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

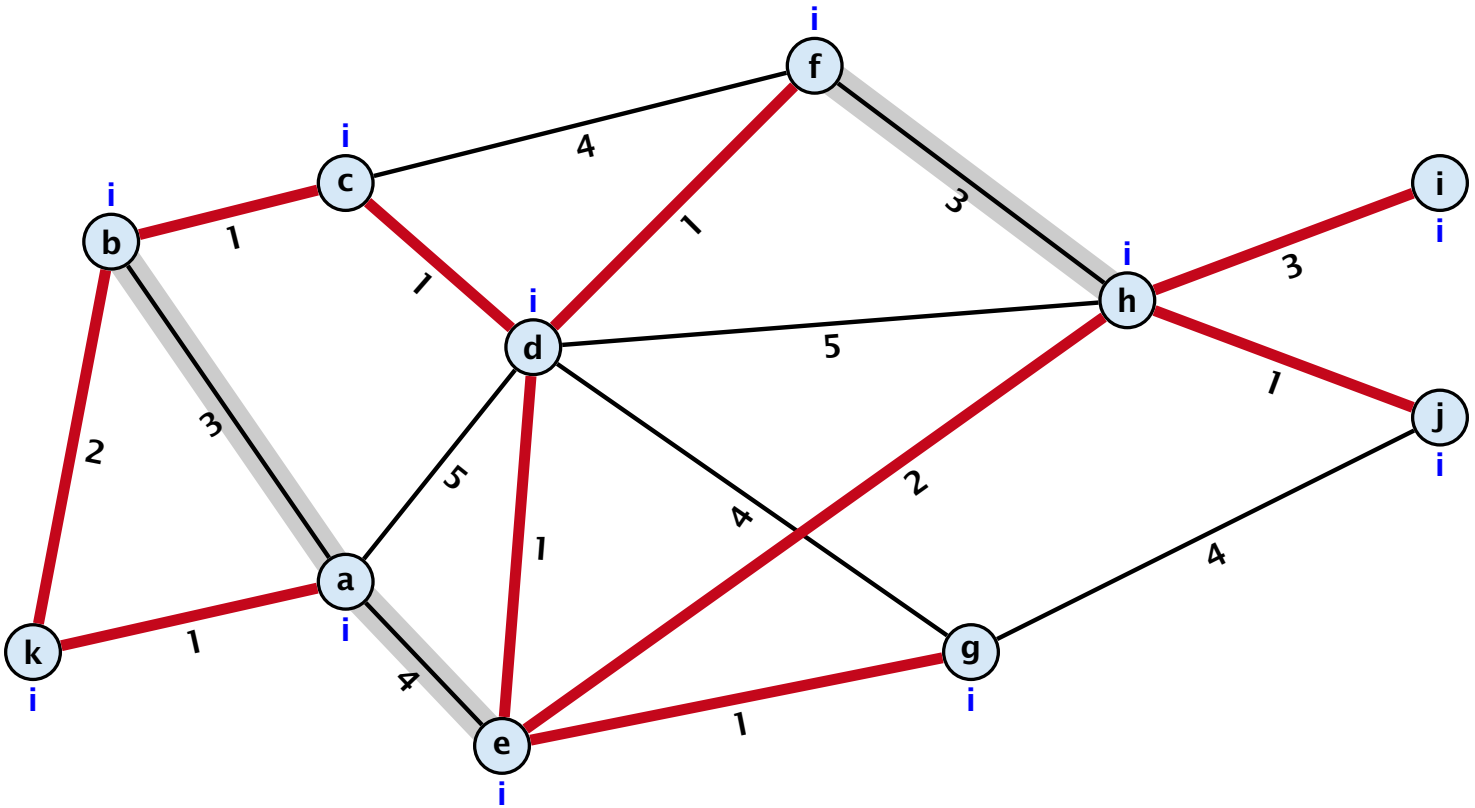
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

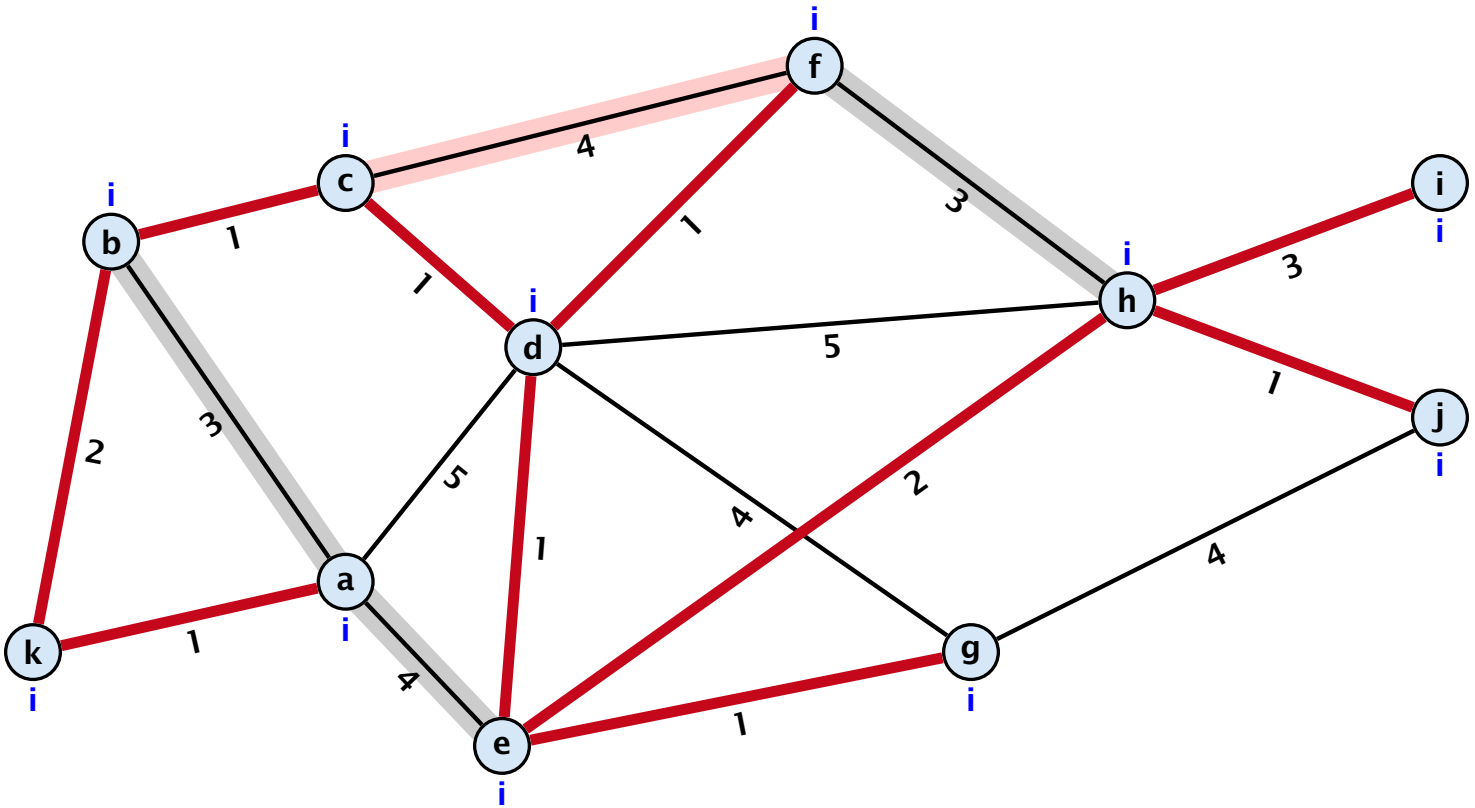
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

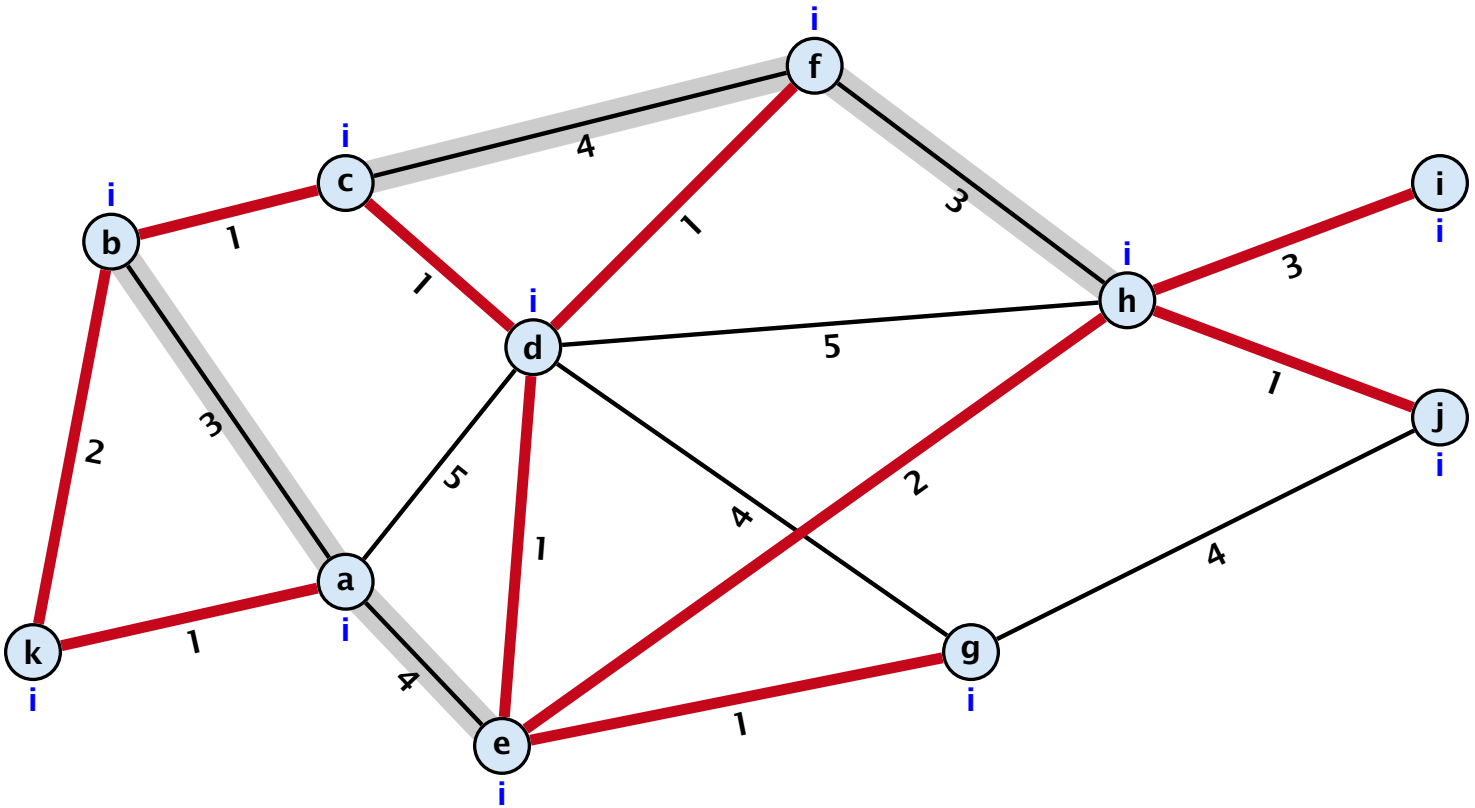
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

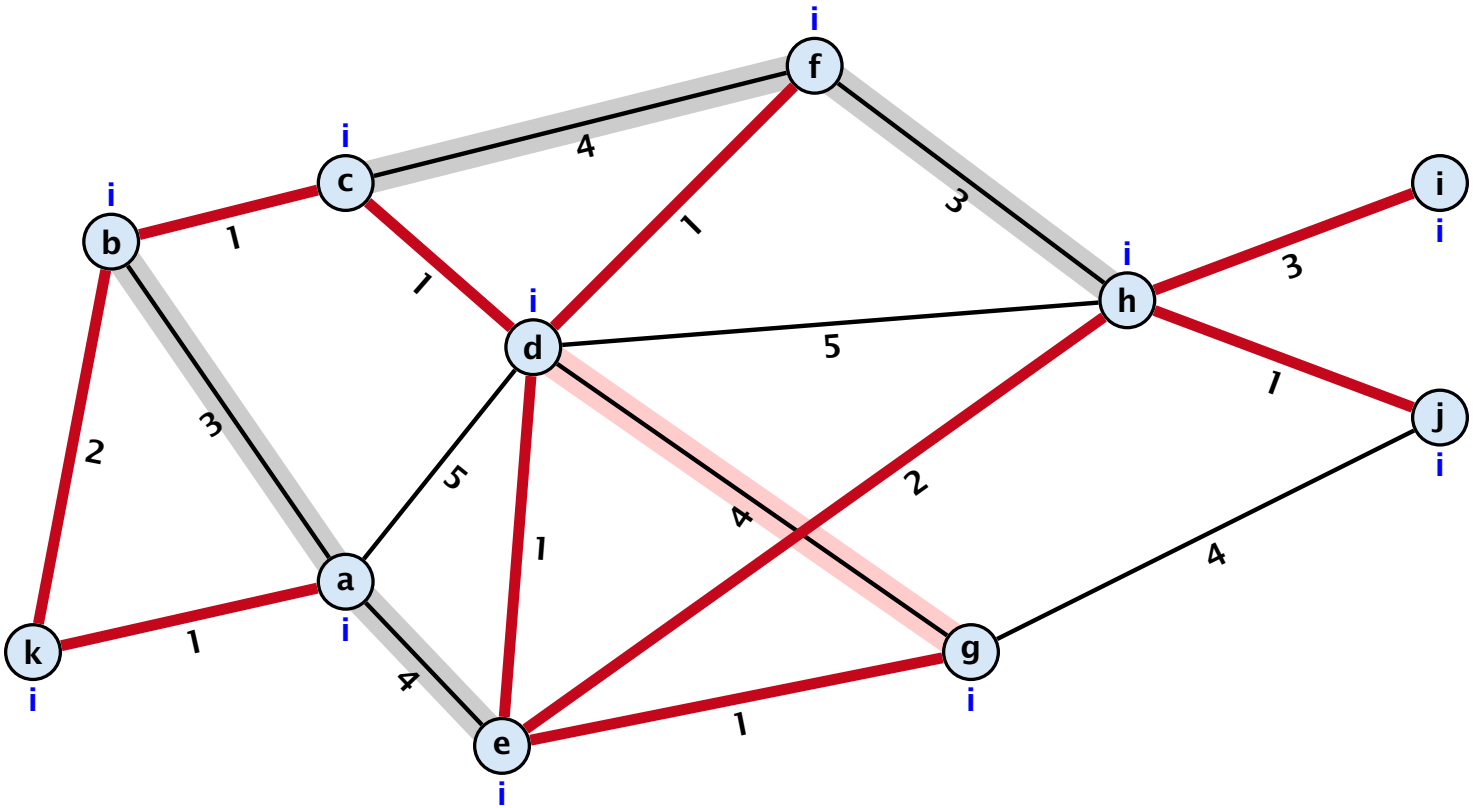
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

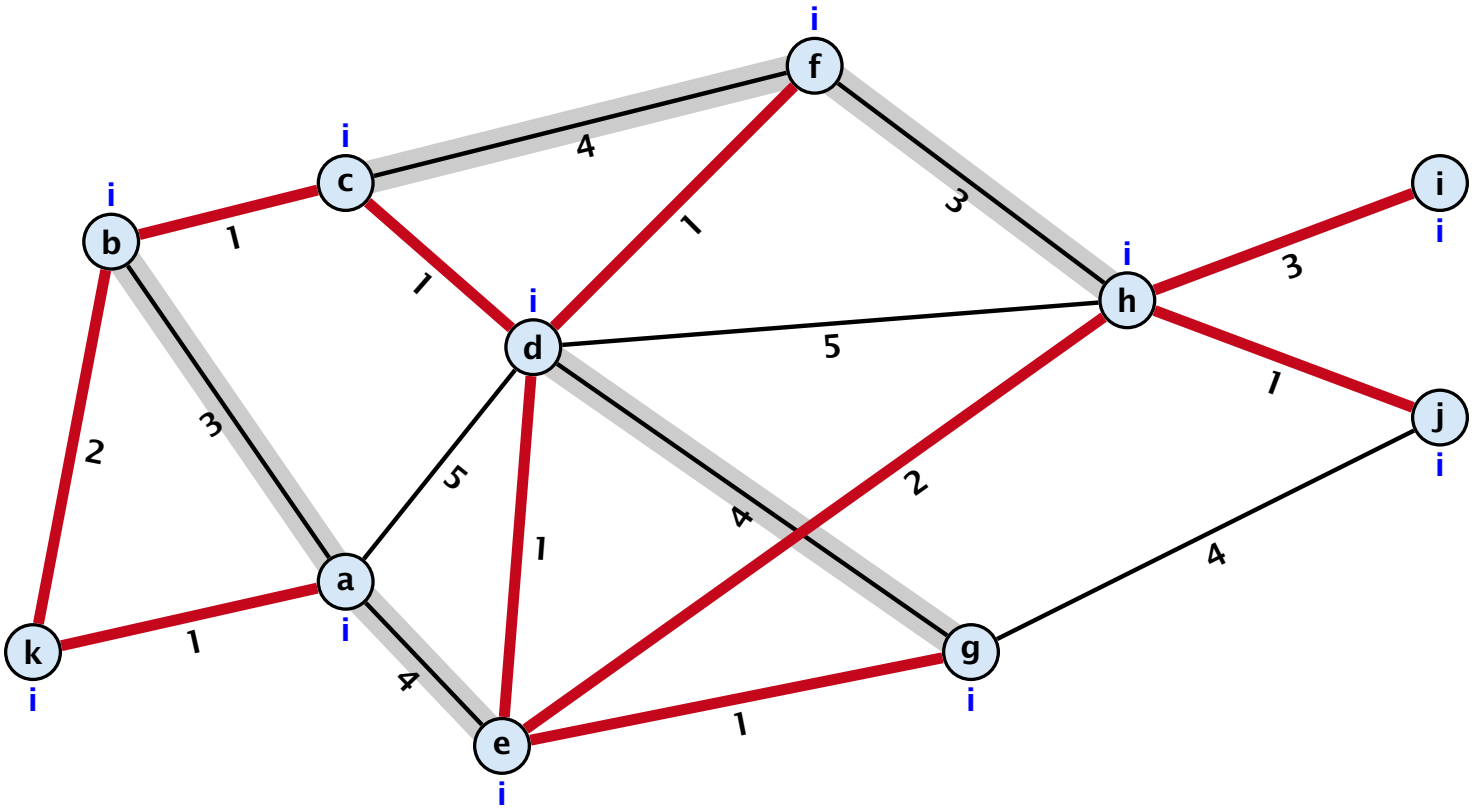
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

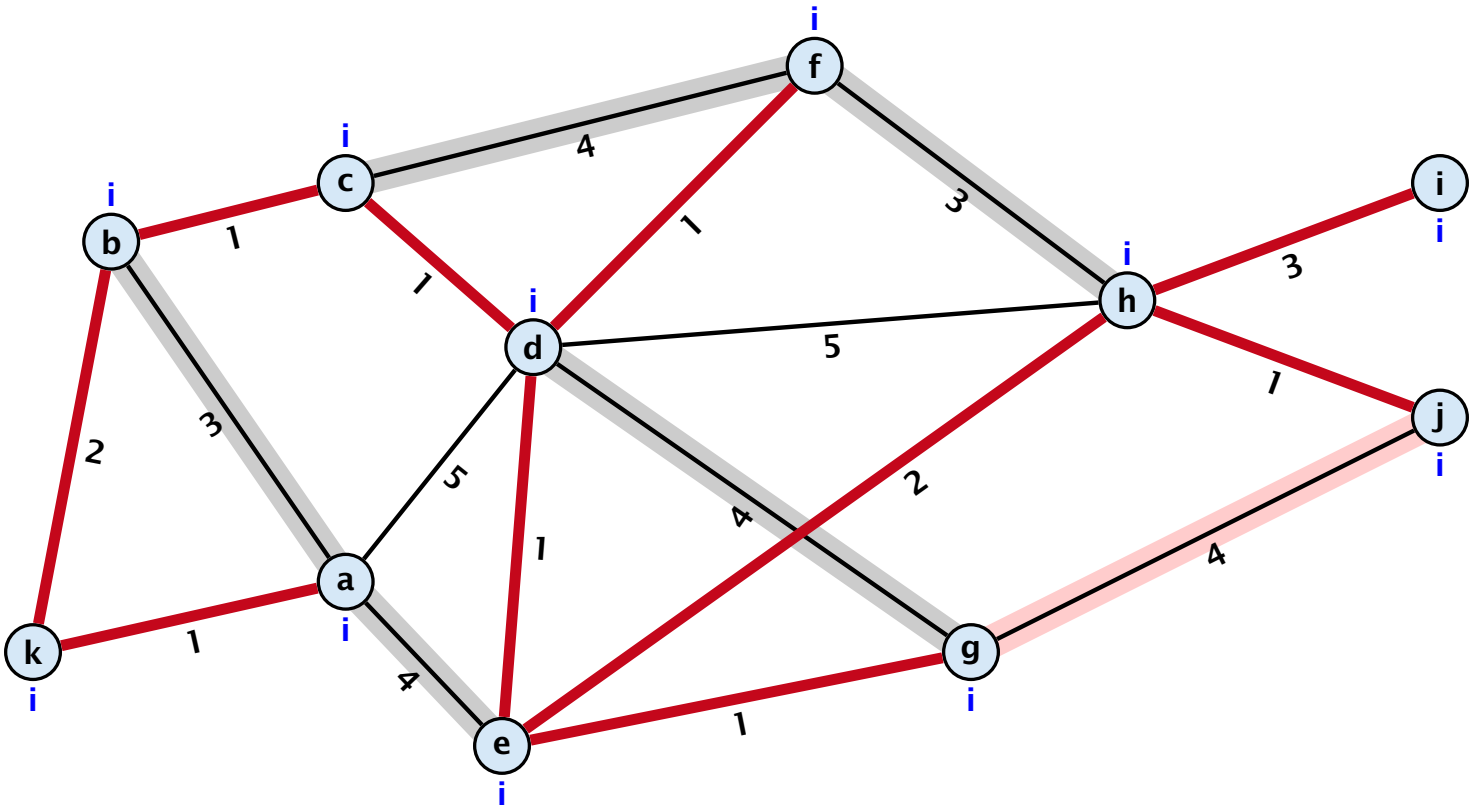
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

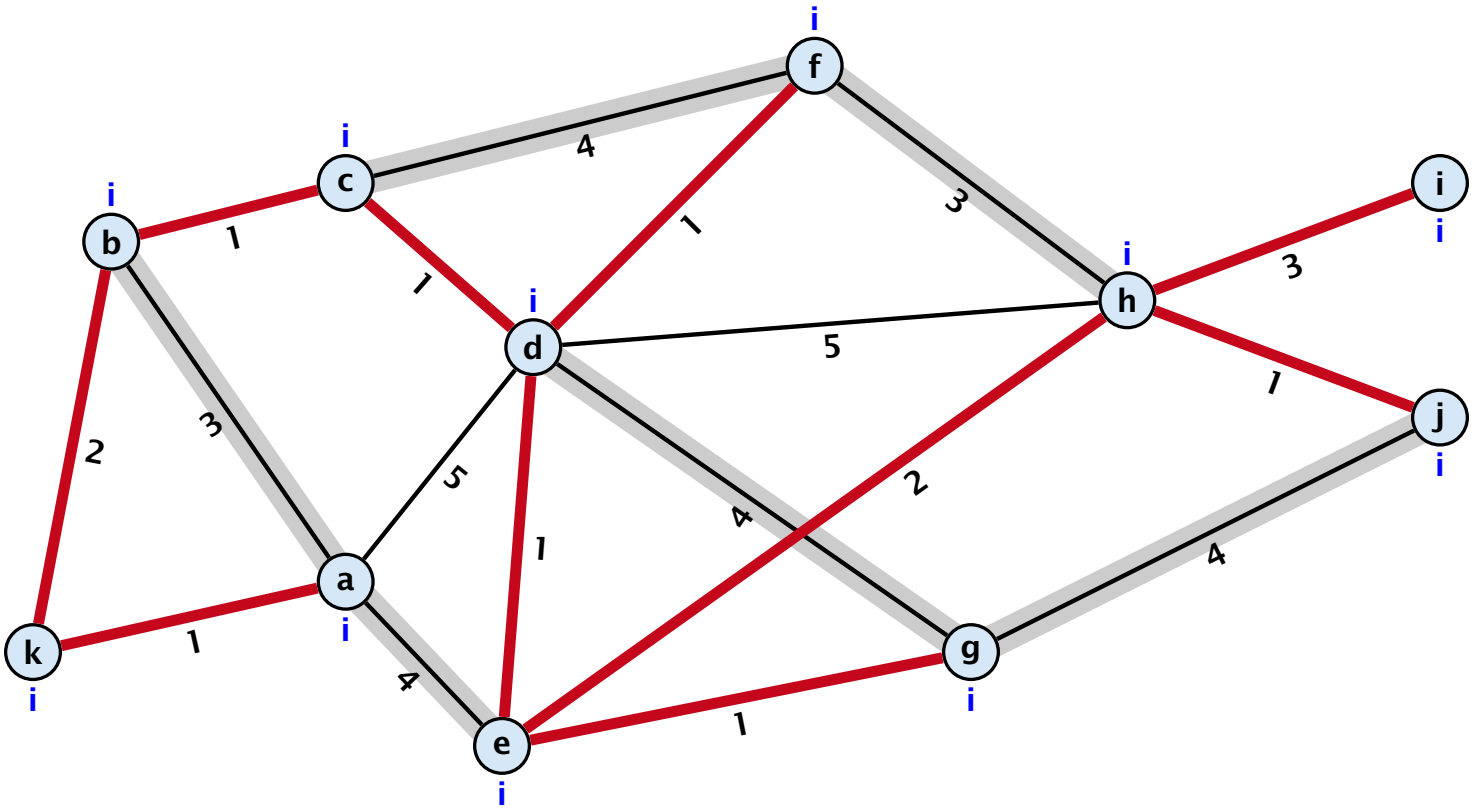
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

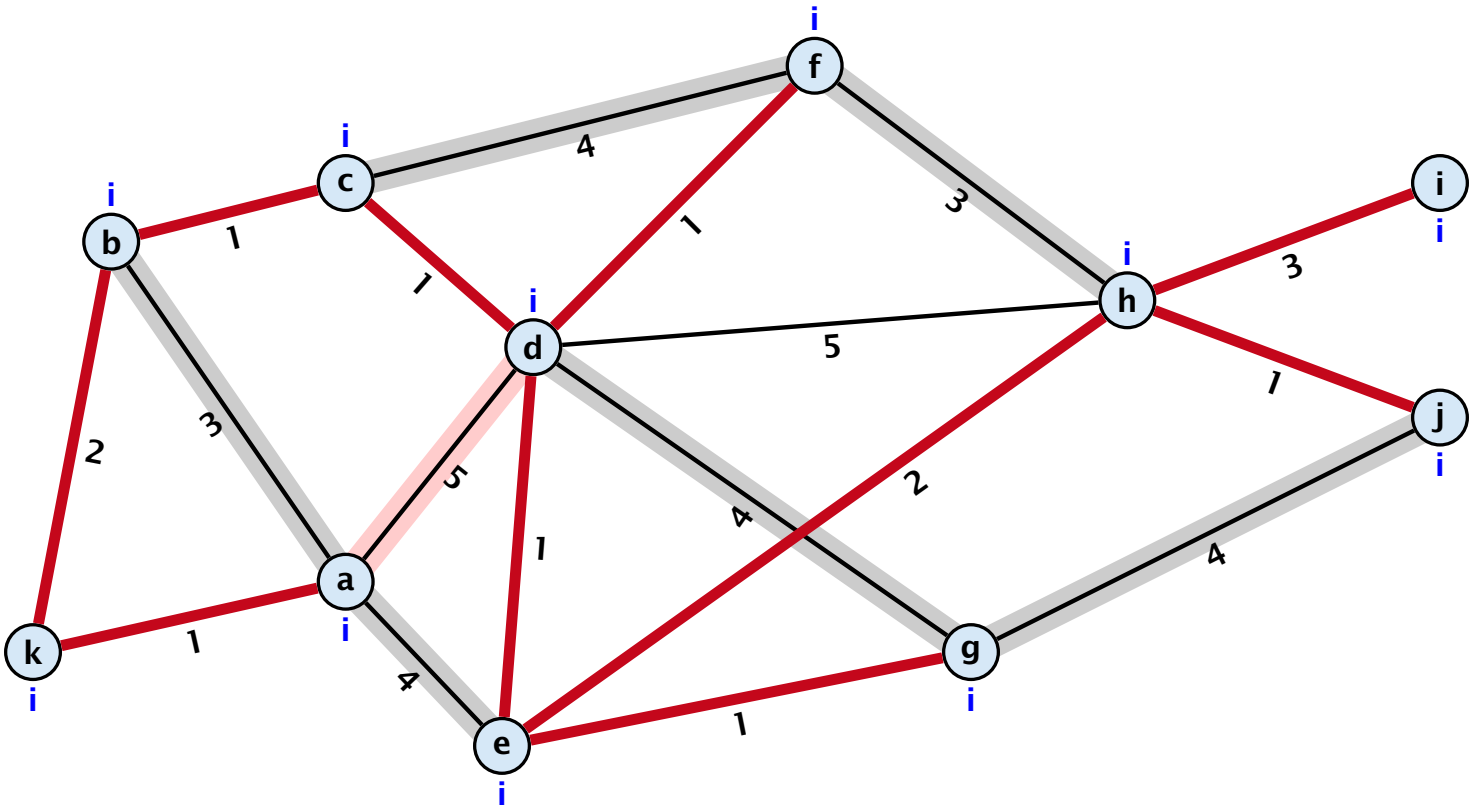
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

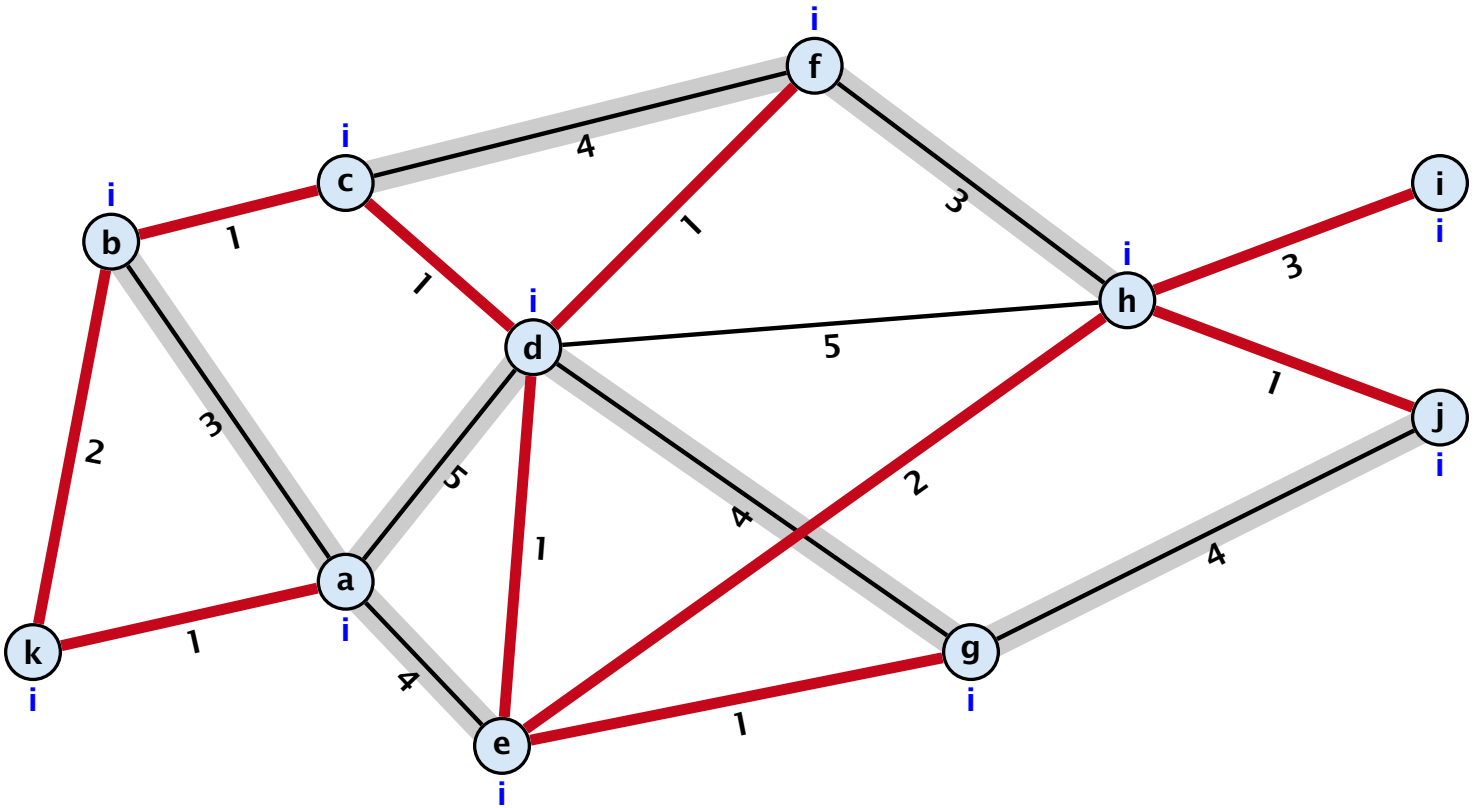
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

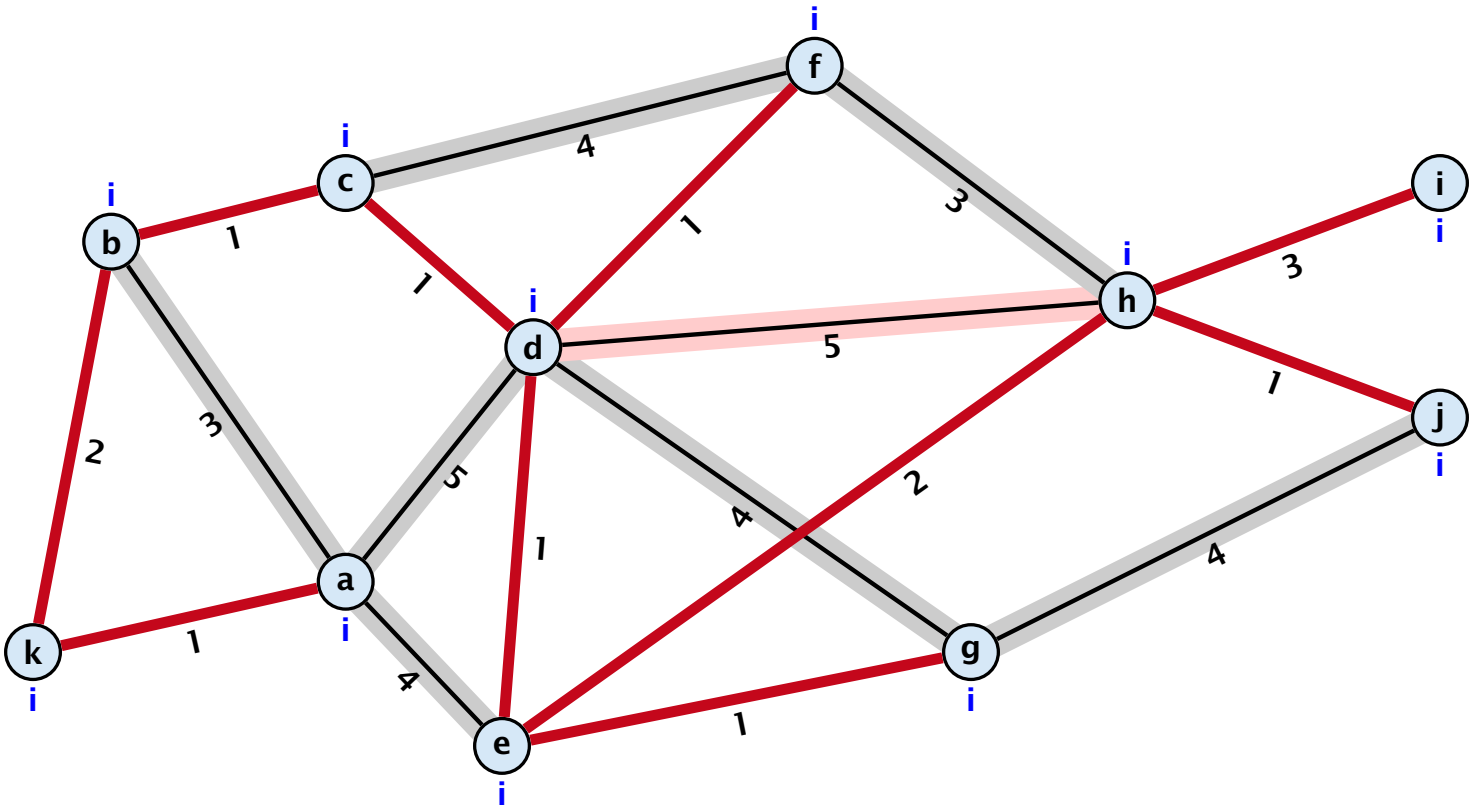
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

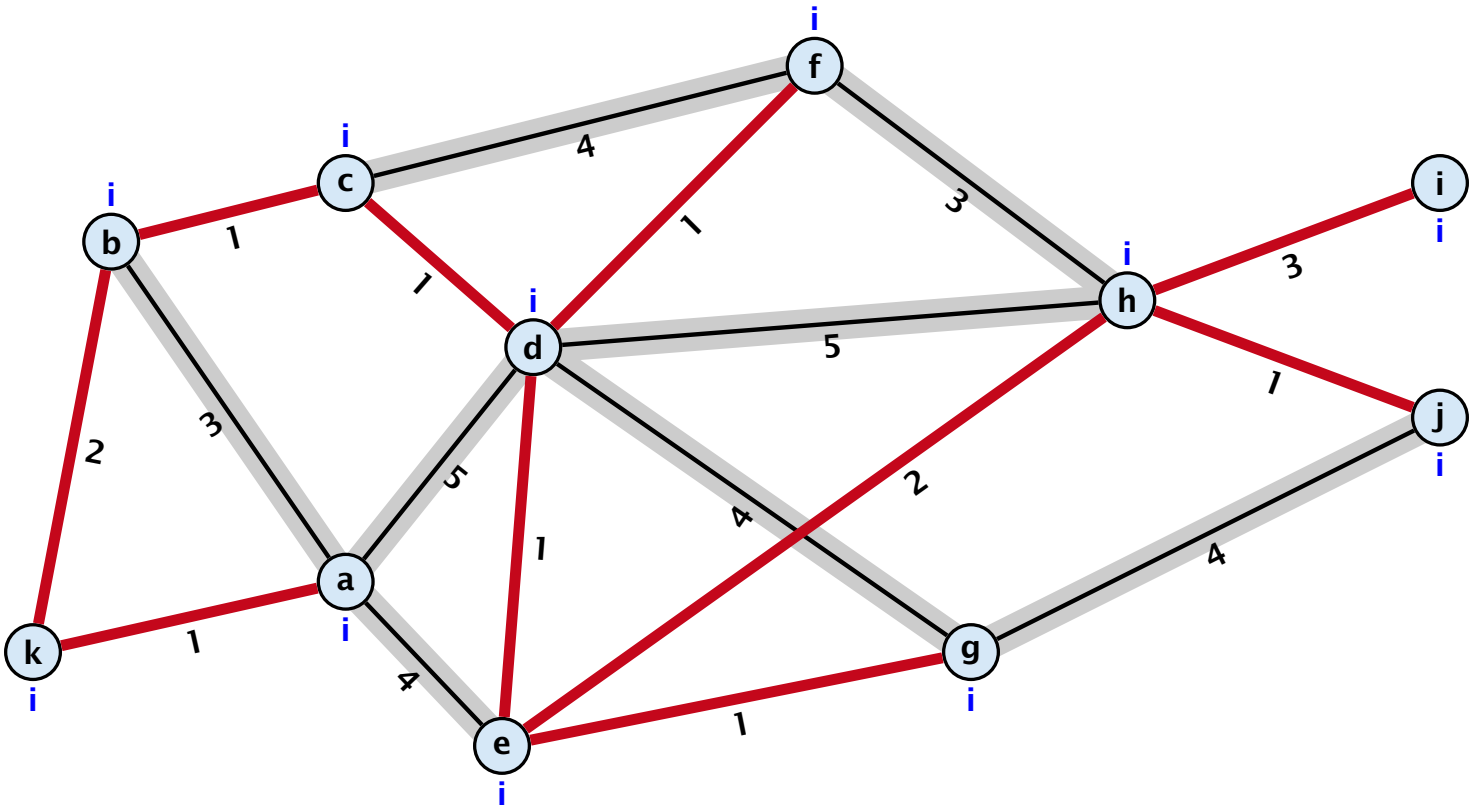
# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

# Minimaler Spannbaum (Kruskal)




**Beobachtung:**

Sei $E'$ eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus $E'$, die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der $e$ enthält.

# 12 Union Find

```
1  Input: ungerichteter Graph G=(V,E,w)
2  Output: Minimaler Spannbaum von G
3
4  Kruskal(G)
5      A = ∅
6      foreach v ∈ V
7          v->set = P->makeset(v->label)
8      sort edges according to weight w;
9      foreach (u,v) ∈ E in increasing order
10         if (P->find(u->set) != P->find(v->set))
11             A = A ∪ {(u,v)}
12             P->union(u->set,v->set);
```

# Listenimplementierung

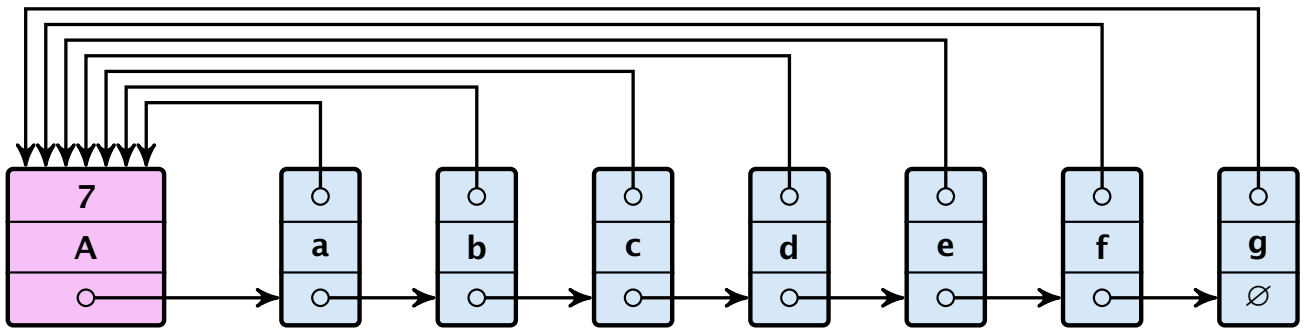
- Die Elemente einer Teilmenge werden in verketteter Liste gespeichert; jedes Listenelement bekommt zusätzlich einen Zeiger auf den Listenkopf.
- Der Listenkopf enthält den Identifier der Teilmenge und die Größe der Menge




- makeset($x$) kann in konstanter Zeit ausgeführt werden.
- find($x$) kann in konstanter Zeit ausgeführt werden.

# Listenimplementierung

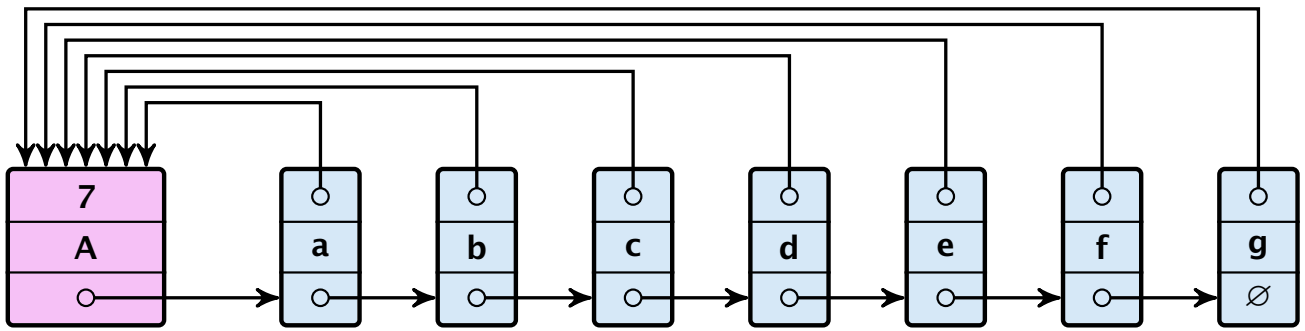
- Die Elemente einer Teilmenge werden in verketteter Liste gespeichert; jedes Listenelement bekommt zusätzlich einen Zeiger auf den Listenkopf.
- Der Listenkopf enthält den Identifier der Teilmenge und die Größe der Menge




- makeset($x$) kann in konstanter Zeit ausgeführt werden.
- find($x$) kann in konstanter Zeit ausgeführt werden.

# Listenimplementierung

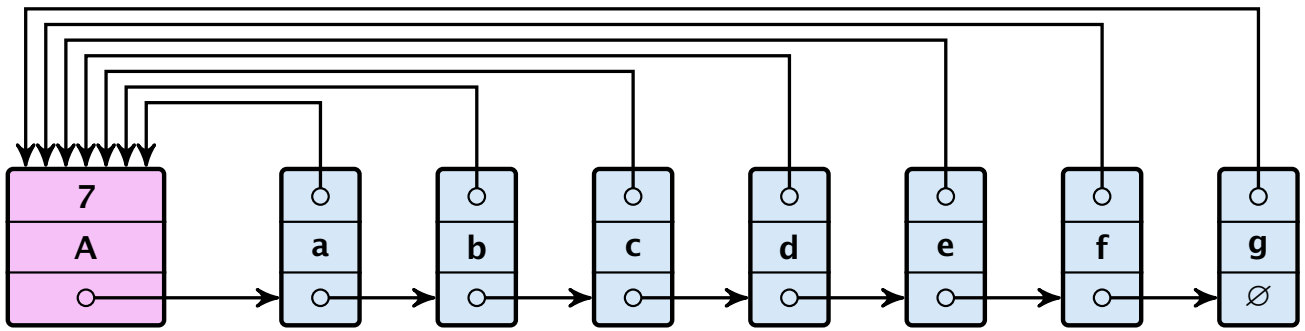
- Die Elemente einer Teilmenge werden in verketteter Liste gespeichert; jedes Listenelement bekommt zusätzlich einen Zeiger auf den Listenkopf.
- Der Listenkopf enthält den Identifier der Teilmenge und die Größe der Menge




- $\mathrm{makeset}(x)$ kann in konstanter Zeit ausgeführt werden.
- $\mathrm{find}(x)$ kann in konstanter Zeit ausgeführt werden.

# Listenimplementierung

- Die Elemente einer Teilmenge werden in verketteter Liste gespeichert; jedes Listenelement bekommt zusätzlich einen Zeiger auf den Listenkopf.
- Der Listenkopf enthält den Identifier der Teilmenge und die Größe der Menge




- $\mathrm{makeset}(x)$ kann in konstanter Zeit ausgeführt werden.
- $\mathrm{find}(x)$ kann in konstanter Zeit ausgeführt werden.

# Listenimplementierung

**union**$(x, y)$

- Bestimme Mengen $S_x$ und $S_y$.
- Durchlaufe die kleinere Liste (z.B. $S_y$), und ändere alle Zeiger auf den Listenkopf, so dass sie auf den Listenkopf von $S_x$ zeigen.
- Füge $S_y$ am Anfang von $S_x$ ein.
- Passe den Größeneintrag von $S_x$ an.
- Laufzeit: $\min\{|S_x|, |S_y|\}$.

# Listenimplementierung

**union**$(x, y)$

- Bestimme Mengen $S_x$ und $S_y$.
- Durchlaufe die kleinere Liste (z.B. $S_y$), und ändere alle Zeiger auf den Listenkopf, so dass sie auf den Listenkopf von $S_x$ zeigen.
- Füge $S_y$ am Anfang von $S_x$ ein.
- Passe den Größeneintrag von $S_x$ an.
- Laufzeit: $\min\{|S_x|, |S_y|\}$.

# Listenimplementierung

**union**$(x, y)$

- Bestimme Mengen $S_x$ und $S_y$.
- Durchlaufe die kleinere Liste (z.B. $S_y$), und ändere alle Zeiger auf den Listenkopf, so dass sie auf den Listenkopf von $S_x$ zeigen.
- Füge $S_y$ am Anfang von $S_x$ ein.
- Passe den Größeneintrag von $S_x$ an.
- Laufzeit: $\min\{|S_x|, |S_y|\}$.

# Listenimplementierung

**union**$(x, y)$

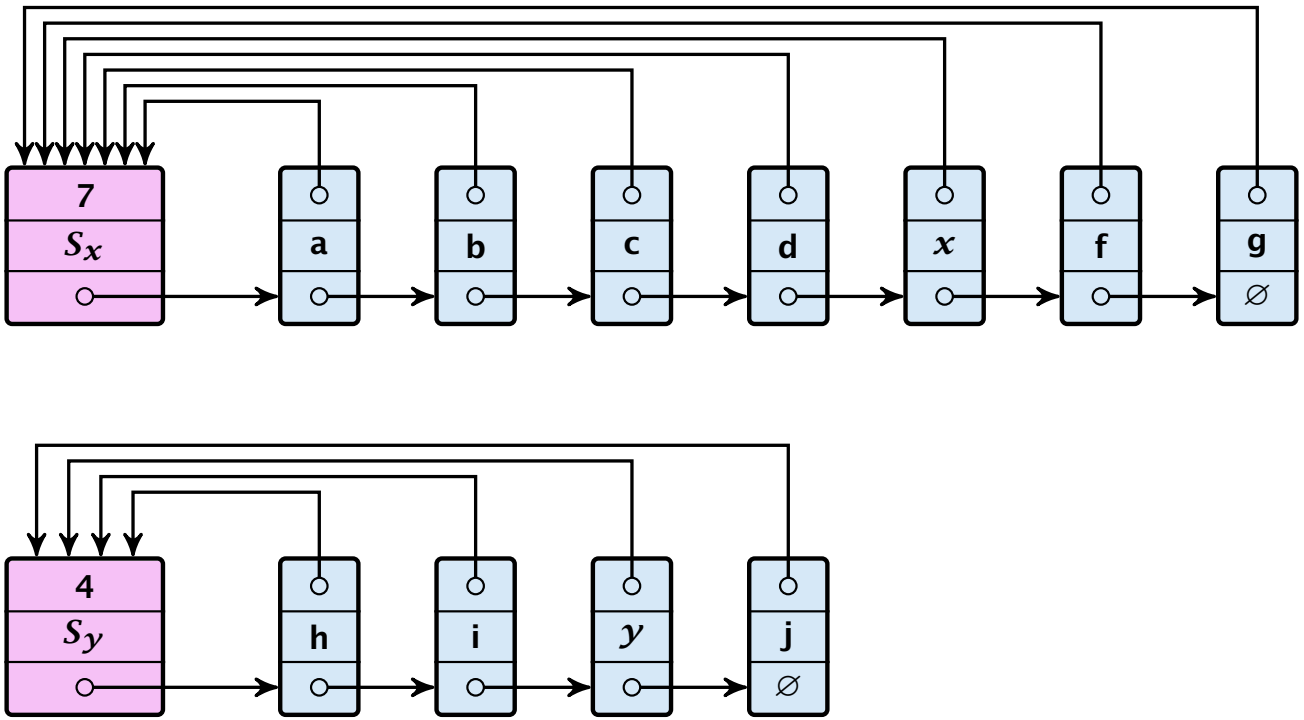
- Bestimme Mengen $S_x$ und $S_y$.
- Durchlaufe die kleinere Liste (z.B. $S_y$), und ändere alle Zeiger auf den Listenkopf, so dass sie auf den Listenkopf von $S_x$ zeigen.
- Füge $S_y$ am Anfang von $S_x$ ein.
- Passe den Größeneintrag von $S_x$ an.
- Laufzeit: $\min\{|S_x|, |S_y|\}$.

# Listenimplementierung

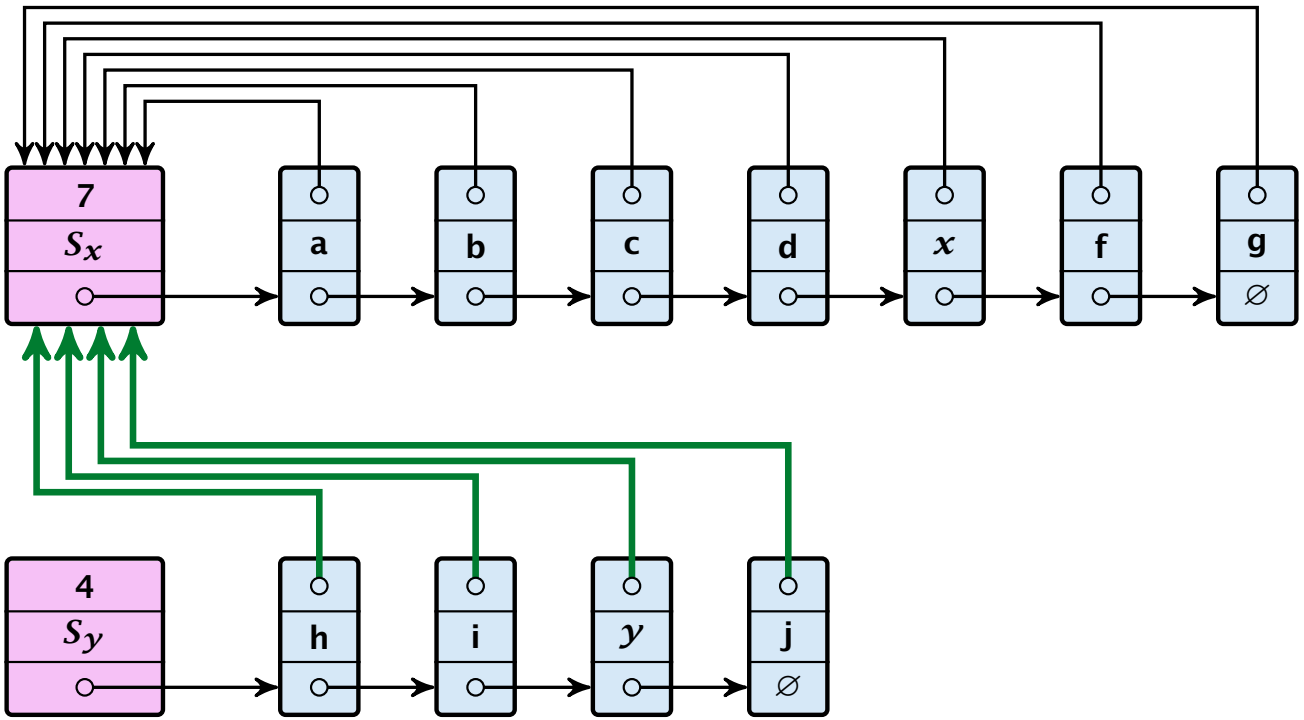
**union**$(x, y)$

- Bestimme Mengen $S_x$ und $S_y$.
- Durchlaufe die kleinere Liste (z.B. $S_y$), und ändere alle Zeiger auf den Listenkopf, so dass sie auf den Listenkopf von $S_x$ zeigen.
- Füge $S_y$ am Anfang von $S_x$ ein.
- Passe den Größeneintrag von $S_x$ an.
- Laufzeit: $\min\{|S_x|, |S_y|\}$.

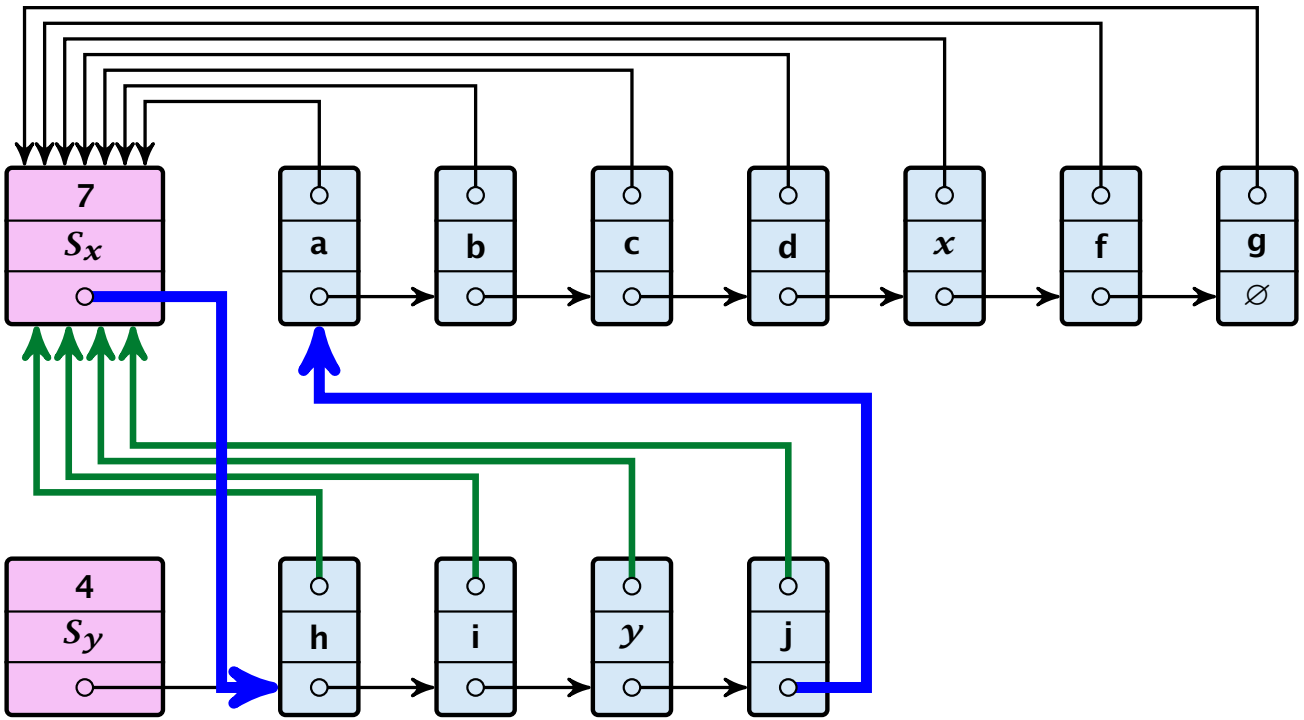
# List Implementation



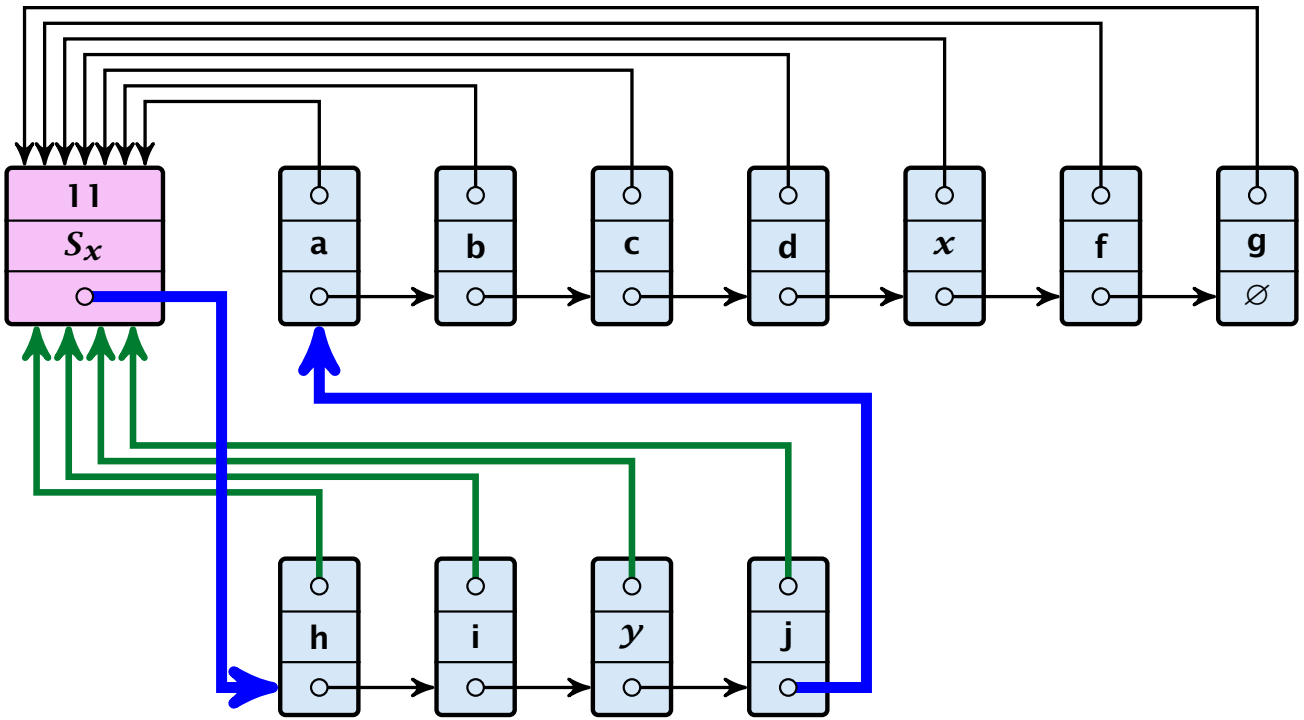
# List Implementation



# List Implementation



# List Implementation



# List Implementation

**Laufzeiten:**

- $\text{find}(x)$: konstant
- $\text{makeset}(x)$: konstant
- $\text{union}(x,y)$: $\mathcal{O}(n)$.

# Baumimplementierung

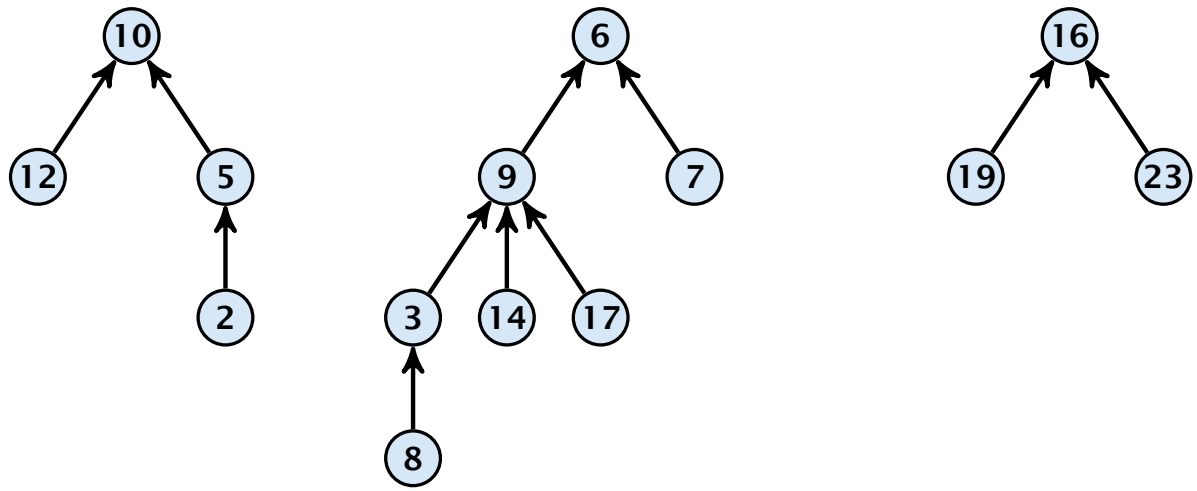
- Verwalte Elemente in Bäumen.
- Die Wurzel eines Baumes dient als Identifier der jeweiligen Teilmenge.
- Nur Elternzeiger existieren; wir können den Baum nicht traversieren und z.B. all Elemente einer Teilmenge ausgeben.
- Beispiel:




Mengensystem $\{2,5,10,12\}$, $\{3,6,7,8,9,14,17\}$, $\{16,19,23\}$.

# Baumimplementierung

- Verwalte Elemente in Bäumen.
- Die Wurzel eines Baumes dient als Identifier der jeweiligen Teilmenge.
- Nur Elternzeiger existieren; wir können den Baum nicht traversieren und z.B. all Elemente einer Teilmenge ausgeben.
- Beispiel:




Mengensystem $\{2,5,10,12\}$, $\{3,6,7,8,9,14,17\}$, $\{16,19,23\}$.

# Baumimplementierung

**makeset($x$)**

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

**find($x$)**

# Baumimplementierung

**makeset($x$)**

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

**find($x$)**

# Baumimplementierung

**makeset($x$)**

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

**find($x$)**

- Handle ist Zeiger auf Element;
- Starte beim Element $x$ im Baum. Gehe aufwärts bis zu Wurzel.
- Gib Identifier der Wurzel zurück.
- Zeit: $\mathcal{O}(\text{level}(x))$, wobei $\text{level}(x)$ die Distanz von $x$ zu Wurzel des jeweiligen Baumes ist. nicht konstant.

# Baumimplementierung

**makeset($x$)**

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

**find($x$)**

- Handle ist Zeiger auf Element;
- Starte beim Element $x$ im Baum. Gehe aufwärts bis zu Wurzel.
- Gib Identifier der Wurzel zurück.
- Zeit: $\mathcal{O}(\text{level}(x))$, wobei $\text{level}(x)$ die Distanz von $x$ zu Wurzel des jeweiligen Baumes ist. nicht konstant.

# Baumimplementierung

**makeset($x$)**

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

**find($x$)**

- Handle ist Zeiger auf Element;
- Starte beim Element $x$ im Baum. Gehe aufwärts bis zu Wurzel.
- Gib Identifier der Wurzel zurück.
- Zeit: $\mathcal{O}(\text{level}(x))$, wobei $\text{level}(x)$ die Distanz von $x$ zu Wurzel des jeweiligen Baumes ist. nicht konstant.

# Baumimplementierung

**makeset($x$)**

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

**find($x$)**

- Handle ist Zeiger auf Element;
- Starte beim Element $x$ im Baum. Gehe aufwärts bis zu Wurzel.
- Gib Identifier der Wurzel zurück.
- Zeit: $\mathcal{O}(\text{level}(x))$, wobei $\text{level}(x)$ die Distanz von $x$ zu Wurzel des jeweiligen Baumes ist. nicht konstant.

# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

**union**$(x, y)$

- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.

# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

**union**$(x, y)$

- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.
- $\text{link}(a, b)$ fügt den kleineren Teilbaum als Kind an den größeren an.

# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

**union**$(x, y)$

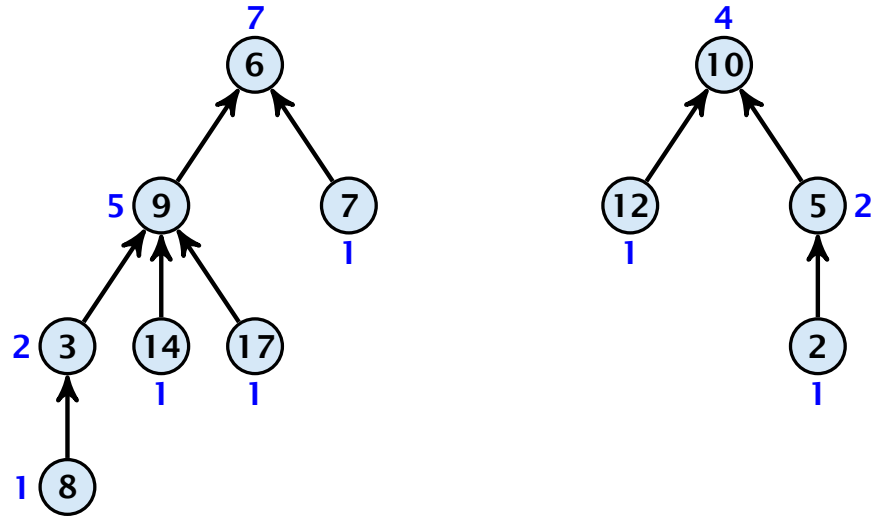
- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.
- $\text{link}(a, b)$ fügt den kleineren Teilbaum als Kind an den größeren an.
- Zusätzlich wird dabei das Größenfeld der neuen Wurzel aktualisert.

# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

**union**$(x, y)$

- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.
- $\text{link}(a, b)$ fügt den kleineren Teilbaum als Kind an den größeren an.
- Zusätzlich wird dabei das Größenfeld der neuen Wurzel aktualisert.

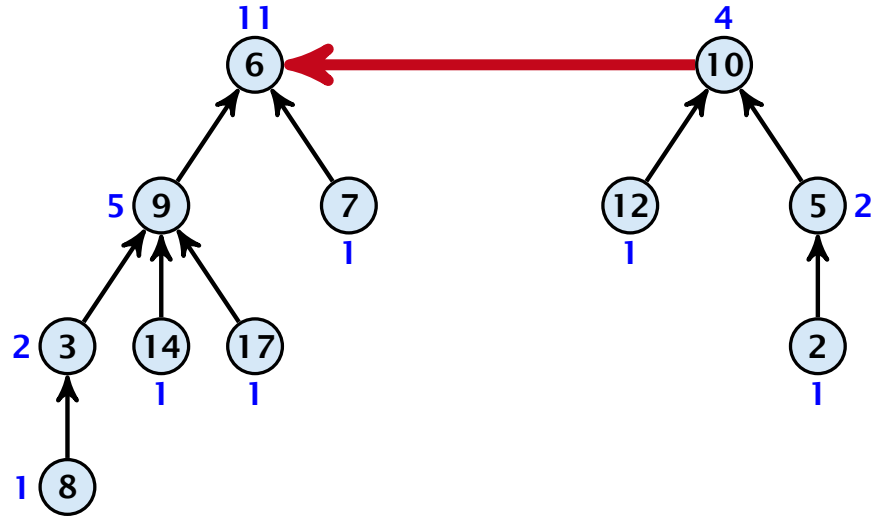


# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

**union**$(x, y)$

- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.
- $\text{link}(a, b)$ fügt den kleineren Teilbaum als Kind an den größeren an.
- Zusätzlich wird dabei das Größenfeld der neuen Wurzel aktualisert.

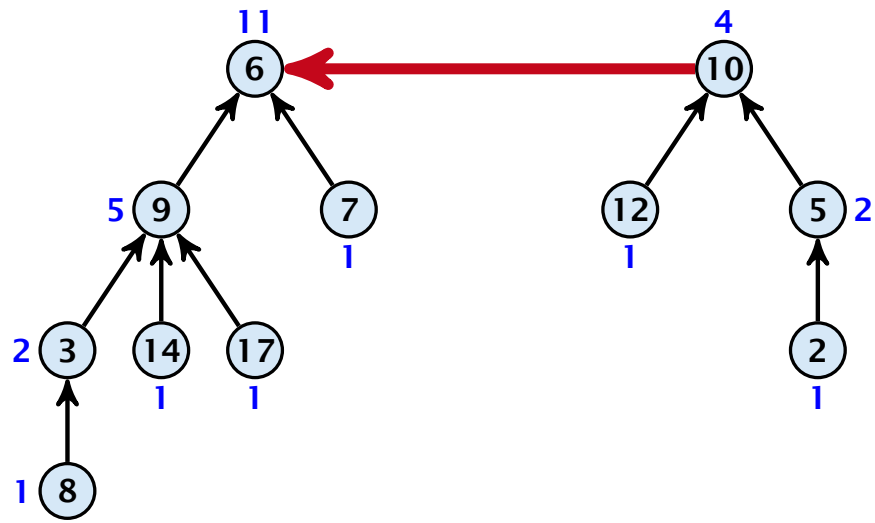


# Baumimplementierung

**Trick:** wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

**union**$(x, y)$

- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.
- $\text{link}(a, b)$ fügt den kleineren Teilbaum als Kind an den größeren an.
- Zusätzlich wird dabei das Größenfeld der neuen Wurzel aktualisert.




- Laufzeit: konstant für $\text{link}(a, b)$ + zwei find-Operationen.

# Baumimplementierung

**Lemma 9**

*Laufzeit für* $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

**Beweis.**

# Baumimplementierung

**Lemma 9**

*Laufzeit für* $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

**Beweis.**

- Wenn wir einen Teilbaum mit Wurzel $c$ an einen Teilbaum mit Wurzel $p$ anfügen, gilt nachher $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$.
- Danach kann sich $\mathrm{size}(c)$ nicht mehr ändern, während $\mathrm{size}(p)$ noch weiter steigen kann (durch weitere union-Operationen).
- D.h. jeder Baum erfüllt immer $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, für eine Kante $(p, c)$, wobei $p$ der Elternknoten von $c$ ist.
- Deshalb kann die Höhe des Baumes nur $\mathcal{O}(\log n)$ sein.

□

# Baumimplementierung

**Lemma 9**

*Laufzeit für* $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

**Beweis.**

- Wenn wir einen Teilbaum mit Wurzel $c$ an einen Teilbaum mit Wurzel $p$ anfügen, gilt nachher $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$.
- Danach kann sich $\mathrm{size}(c)$ nicht mehr ändern, während $\mathrm{size}(p)$ noch weiter steigen kann (durch weitere $\mathrm{union}$-Operationen).
- D.h. jeder Baum erfüllt immer $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, für eine Kante $(p, c)$, wobei $p$ der Elternknoten von $c$ ist.
- Deshalb kann die Höhe des Baumes nur $\mathcal{O}(\log n)$ sein.

□

# Baumimplementierung

**Lemma 9**

*Laufzeit für* $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

**Beweis.**

- Wenn wir einen Teilbaum mit Wurzel $c$ an einen Teilbaum mit Wurzel $p$ anfügen, gilt nachher $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$.
- Danach kann sich $\mathrm{size}(c)$ nicht mehr ändern, während $\mathrm{size}(p)$ noch weiter steigen kann (durch weitere $\mathrm{union}$-Operationen).
- D.h. jeder Baum erfüllt immer $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, für eine Kante $(p, c)$, wobei $p$ der Elternknoten von $c$ ist.
- Deshalb kann die Höhe des Baumes nur $\mathcal{O}(\log n)$ sein.

□

# Baumimplementierung

**Lemma 9**

*Laufzeit für* $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

**Beweis.**

- Wenn wir einen Teilbaum mit Wurzel $c$ an einen Teilbaum mit Wurzel $p$ anfügen, gilt nachher $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$.
- Danach kann sich $\mathrm{size}(c)$ nicht mehr ändern, während $\mathrm{size}(p)$ noch weiter steigen kann (durch weitere $\mathrm{union}$-Operationen).
- D.h. jeder Baum erfüllt immer $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, für eine Kante $(p, c)$, wobei $p$ der Elternknoten von $c$ ist.
- Deshalb kann die Höhe des Baumes nur $\mathcal{O}(\log n)$ sein.

□

# Baumimplementierung

**Lemma 9**

*Laufzeit für* $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

**Beweis.**

- Wenn wir einen Teilbaum mit Wurzel $c$ an einen Teilbaum mit Wurzel $p$ anfügen, gilt nachher $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$.
- Danach kann sich $\mathrm{size}(c)$ nicht mehr ändern, während $\mathrm{size}(p)$ noch weiter steigen kann (durch weitere $\mathrm{union}$-Operationen).
- D.h. jeder Baum erfüllt immer $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, für eine Kante $(p, c)$, wobei $p$ der Elternknoten von $c$ ist.
- Deshalb kann die Höhe des Baumes nur $\mathcal{O}(\log n)$ sein.

□

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

# Pfadkomprimierung

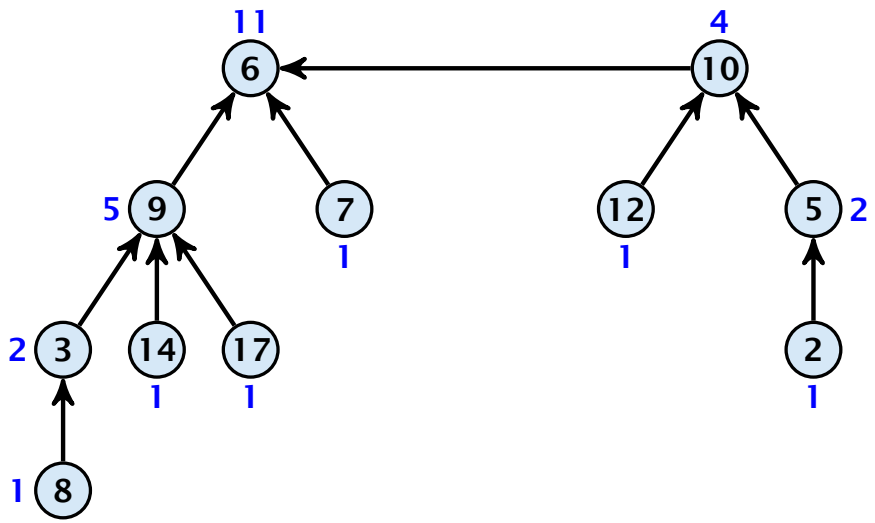
**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

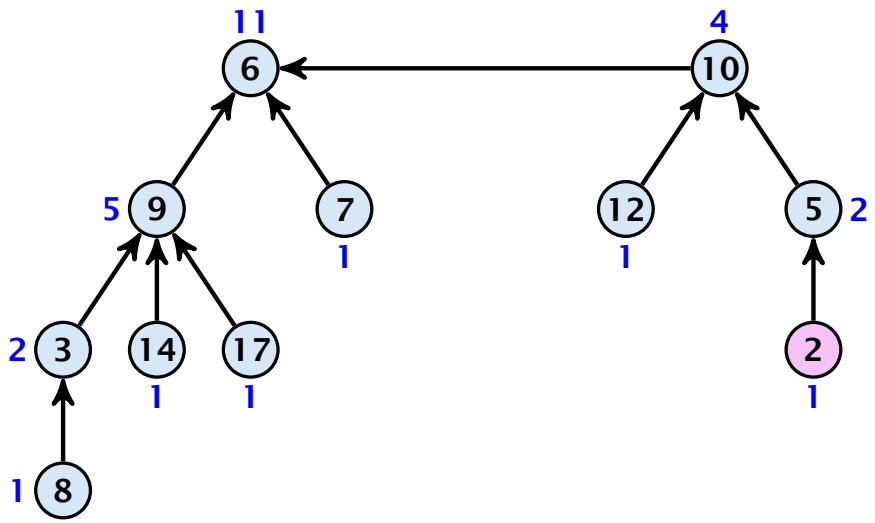



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

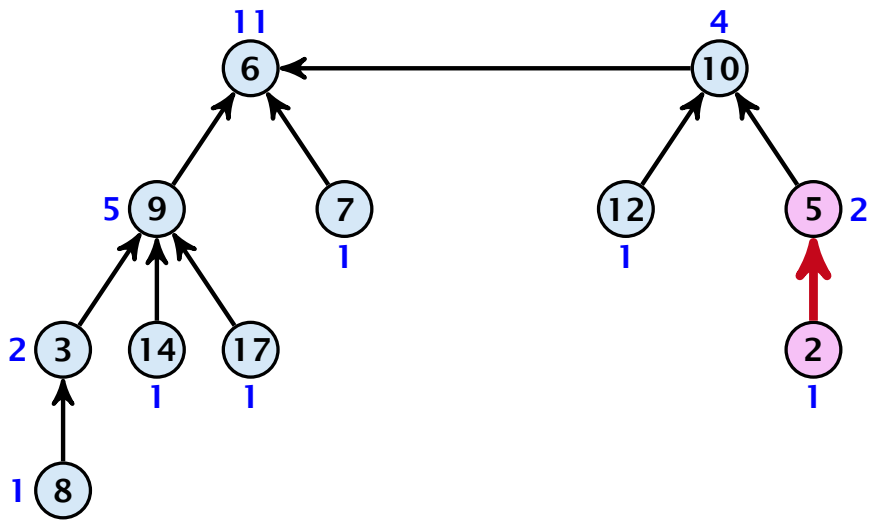



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

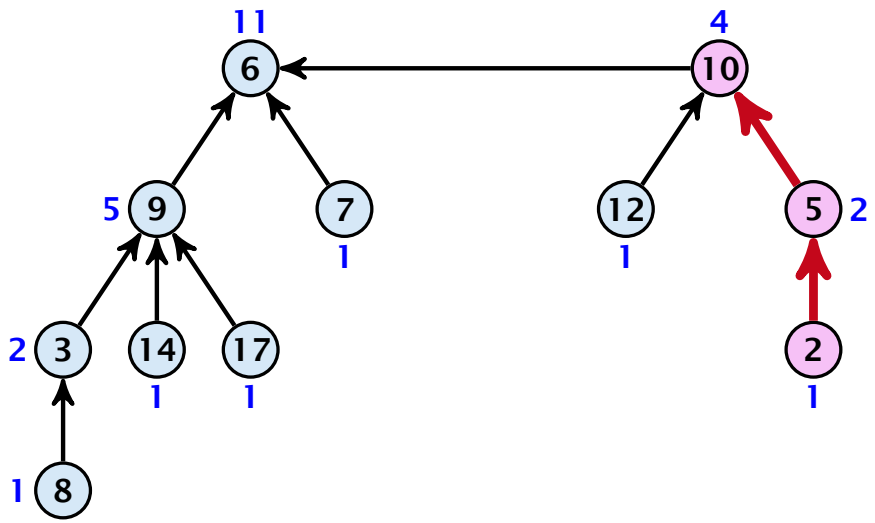



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

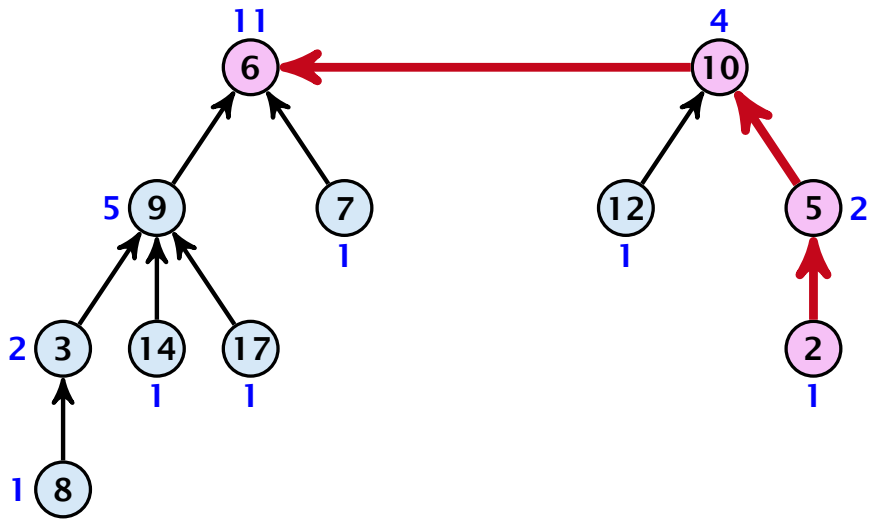



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

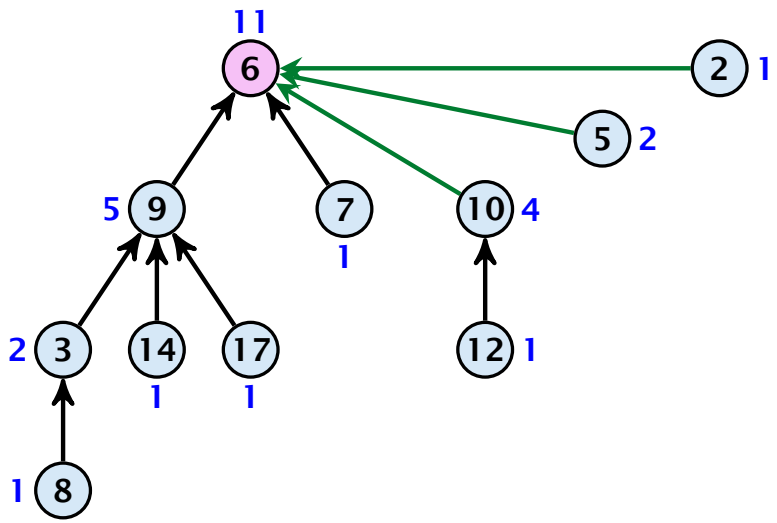



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

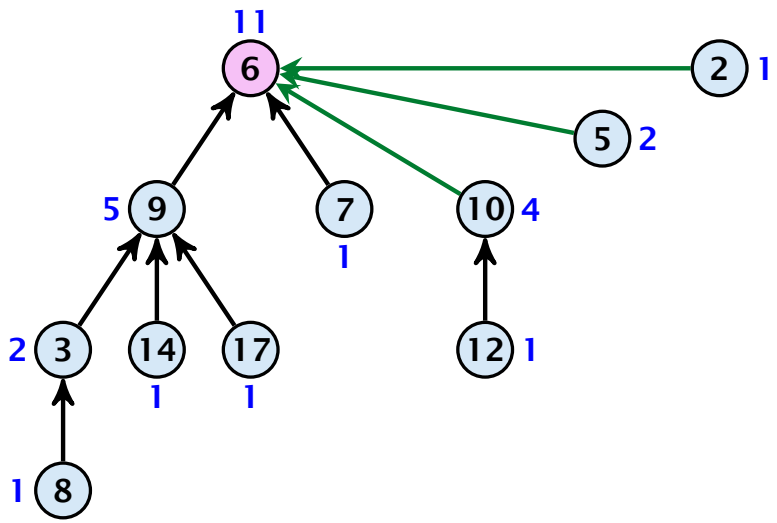



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

**find($x$):**

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.




- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

# Pfadkomprimierung

Die asymptotische Laufzeit ändert sich durch das Hinzufügen der Pfadkomprimierung nicht.

Die worst-case Laufzeit ist immer noch nur $\Theta(\log n)$.

# Pfadkomprimierung

Die asymptotische Laufzeit ändert sich durch das Hinzufügen der Pfadkomprimierung nicht.

Die worst-case Laufzeit ist immer noch nur $\mathcal{O}(\log n)$.

# Amortisierte Analyse

Bei der amortisierten Analyse betrachtet man nicht die Laufzeit für eine Operation, sondern die durchschnittliche Laufzeit über eine Folge von Operationen (auf höchstens $n$ Elementen beginnend mit leerer Menge).

- Die Listenimplementierung hat eine amortisierte Laufzeit von $\mathcal{O}(\log n)$ für `union`, und $\mathcal{O}(1)$ für `makeset`, und `find`.
- Die Baumimplementierung hat eine amortisierte Laufzeit von $\mathcal{O}(\log^* n)$ für alle Operationen.

$\log^* n$ kann man sich folgendermaßen vorstellen. Man gibt $n$ in den Taschenrechner ein, und zählt wie oft man die log-Taste drücken muß damit man eine Zahl $\leq 1$ erhält. Falls $n$ die Anzahl der Atome im (beobachtbaren) Universum ist dann ist $\log^* n = 5$.

# Laufzeit Kruskal

- Der Algorithmus sortiert die Kanten gemäß Gewicht; Laufzeit $\mathcal{O}(m \log m)$
- Dann läuft er über alle Kanten; für jede Kante führt er 2 find-Operationen aus und ggf. eine union-Operation. $\mathcal{O}(m \log^* m)$

Insgesamt: $\mathcal{O}(m \log m)$ (dominiert durch das Sortieren der Kanten)

# 13 Hashing

**Wörterbuchoperationen:**

- $S.\,\mathbf{insert}(x)$: Füge Element $x$.
- $S.\,\mathbf{delete}(x)$: Lösche das durch $x$ referenzierte Element.
- $S.\,\mathbf{search}(k)$: Gib eine Referenz auf Element $e$ zurück mit $\mathrm{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

# 13 Hashing

**Wörterbuchoperationen:**

- $S.\,\mathbf{insert}(x)$: Füge Element $x$.
- $S.\,\mathbf{delete}(x)$: Lösche das durch $x$ referenzierte Element.
- $S.\,\mathbf{search}(k)$: Gib eine Referenz auf Element $e$ zurück mit $\mathrm{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

# 13 Hashing

**Wörterbuchoperationen:**

- $S.\,\mathbf{insert}(x)$: Füge Element $x$.
- $S.\,\mathbf{delete}(x)$: Lösche das durch $x$ referenzierte Element.
- $S.\,\mathbf{search}(k)$: Gib eine Referenz auf Element $e$ zurück mit $\mathrm{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

# 13 Hashing

**Wörterbuchoperationen:**

- $S.\,\mathbf{insert}(x)$: Füge Element $x$.
- $S.\,\mathbf{delete}(x)$: Lösche das durch $x$ referenzierte Element.
- $S.\,\mathbf{search}(k)$: Gib eine Referenz auf Element $e$ zurück mit $\mathrm{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

# 13 Hashing

**Definitionen:**

- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

# 13 Hashing

**Definitionen:**

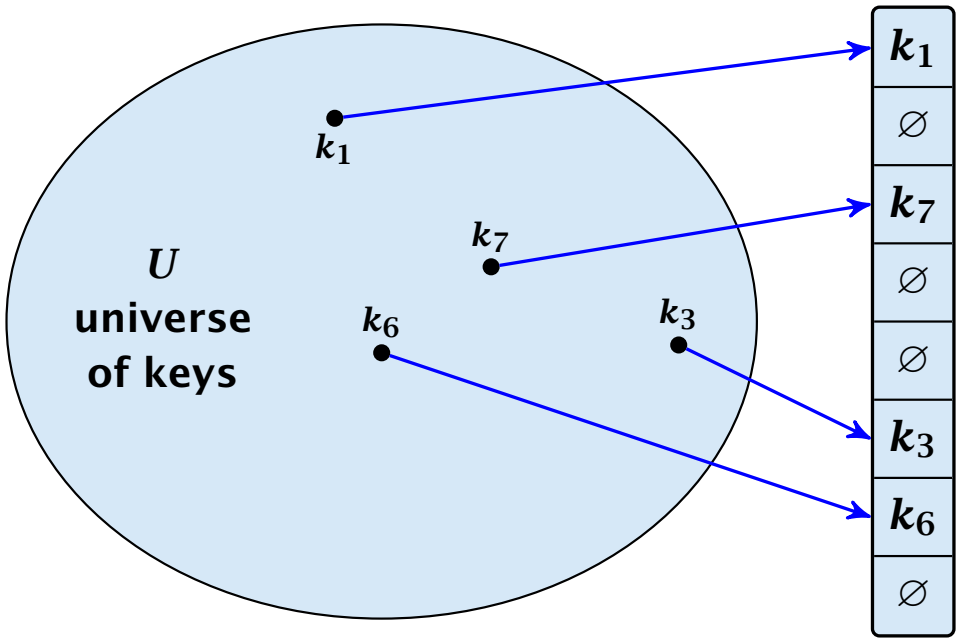
- Universum $U$ von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. $U$ sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

**Die Hashfunktion $h$ sollte:**

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

# Direkte Addressierung

Idealerweise bildet die Hashfunktion alle Elemente auf unterschiedliche Tabelleneinträge ab.




Diesen Spezialfall nennt man Direkte Adressierung. Selten möglich, da das Universum üblicherweise zu groß ist.

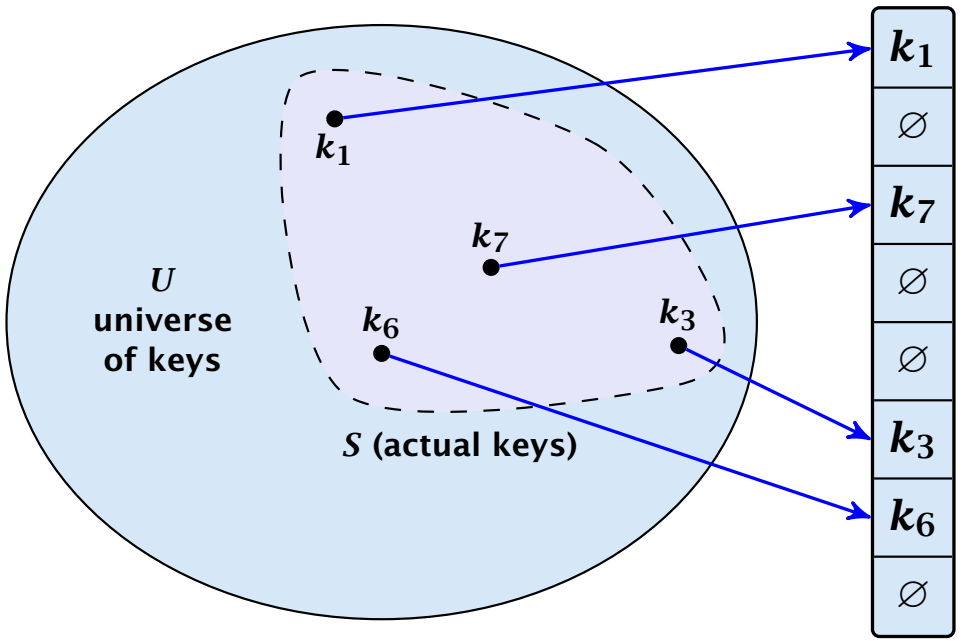
# Direkte Adressierung

**Operationen**

- `insert(x)`
  `A[x->key]=x`
- `search(k)`
  `return A[k]`
- `delete(x)`
  `A[x->key]=NULL`

# Perfektes Hashing

Angenommen wir kennen die Menge $S$ der auftretenen Schlüssel (kein Löschen/ kein Einfügen). Dann möchte man eine einfache Hashfunktion finden, die alle Schlüssel auf unterschiedliche Positionen abbildet.




Solche eine Hashfunktion $h$ nennt man eine perfekte Hashfunktion für Menge $S$.

# Perfektes Hashing

**Operationen**

- `insert(x)`
  `A[h(x->key)]=x`
- `search(k)`
  `return A[h(k)]`
- `delete(x)`
  `A[h(x->key)]=NULL`

# Kollisionen

Falls wir die Schlüssel nicht kennen, ist das beste, dass die Hashfunktion diese gleichmäßig über die Tabelle verteilt.

**Problem: Kollisionen**
Üblicherweise ist das Universum $U$ viele größer als die Tabellengröße $n$.

Zwei Elemente $k_1, k_2$ aus $S$ können auf den gleichen Speicherort abbilden (d.h., $h(k_1) = h(k_2)$). Dies nennt man eine Hashkollision.

# Kollisionen

Falls wir die Schlüssel nicht kennen, ist das beste, dass die Hashfunktion diese gleichmäßig über die Tabelle verteilt.

**Problem: Kollisionen**
Üblicherweise ist das Universum $U$ viele größer als die Tabellengröße $n$.

Zwei Elemente $k_1, k_2$ aus $S$ können auf den gleichen Speicherort abbilden (d.h., $h(k_1) = h(k_2)$). Dies nennt man eine Hashkollision.

# Kollisionen

Falls wir die Schlüssel nicht kennen, ist das beste, dass die Hashfunktion diese gleichmäßig über die Tabelle verteilt.

**Problem: Kollisionen**
Üblicherweise ist das Universum $U$ viele größer als die Tabellengröße $n$.

Zwei Elemente $k_1, k_2$ aus $S$ können auf den gleichen Speicherort abbilden (d.h., $h(k_1) = h(k_2)$). Dies nennt man eine Hashkollision.

# Kollisionen

Typischerweise treten Kollisionen auf wenn die Anzahl der Elemente $S$ sich $\Theta(\sqrt{n})$ nähert.

**Lemma 10**

*Die Wahrscheinlichkeit einer Kollision bei uniformem Hashing wenn $m$ Elemente in eine Tabelle der Größe $n$ abgebildet werden ist mindestens*

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} .$$

**Uniformes Hashing**:
Die Hashfunktion ist eine zufällige Funktion aus der Menge aller Funktionen $f : U \to [0, \ldots, n-1]$.

Uniformes Hashing ist nicht praktikabel, da solch eine Hashfunktion sehr viel Speicherplatz benötigt.

# Kollisionen

Typischerweise treten Kollisionen auf wenn die Anzahl der Elemente $S$ sich $\Theta(\sqrt{n})$ nähert.

**Lemma 10**

*Die Wahrscheinlichkeit einer Kollision bei uniformem Hashing wenn $m$ Elemente in eine Tabelle der Größe $n$ abgebildet werden ist mindestens*

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} \quad .$$

**Uniformes Hashing:**

Die Hashfunktion ist eine zufällige Funktion aus der Menge aller Funktionen $f : U \to [0, \ldots, n-1]$.

Uniformes Hashing ist nicht praktikabel, da solch eine Hashfunktion sehr viel Speicherplatz benötigt.

# Kollisionen

Typischerweise treten Kollisionen auf wenn die Anzahl der Elemente $S$ sich $\Theta(\sqrt{n})$ nähert.

**Lemma 10**

*Die Wahrscheinlichkeit einer Kollision bei uniformem Hashing wenn $m$ Elemente in eine Tabelle der Größe $n$ abgebildet werden ist mindestens*

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} \quad .$$

**Uniformes Hashing**:

Die Hashfunktion ist eine zufällige Funktion aus der Menge aller Funktionen $f : U \to [0, \ldots, n-1]$.

Uniformes Hashing ist nicht praktikabel, da solch eine Hashfunktion sehr viel Speicherplatz benötigt.

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}]$$

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n}$$

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right)$$

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1-\frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n}$$

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1-\frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}}$$

# Kollisionen

**Beweis.**

Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1-\frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

# Kollisionen

**Beweis.**

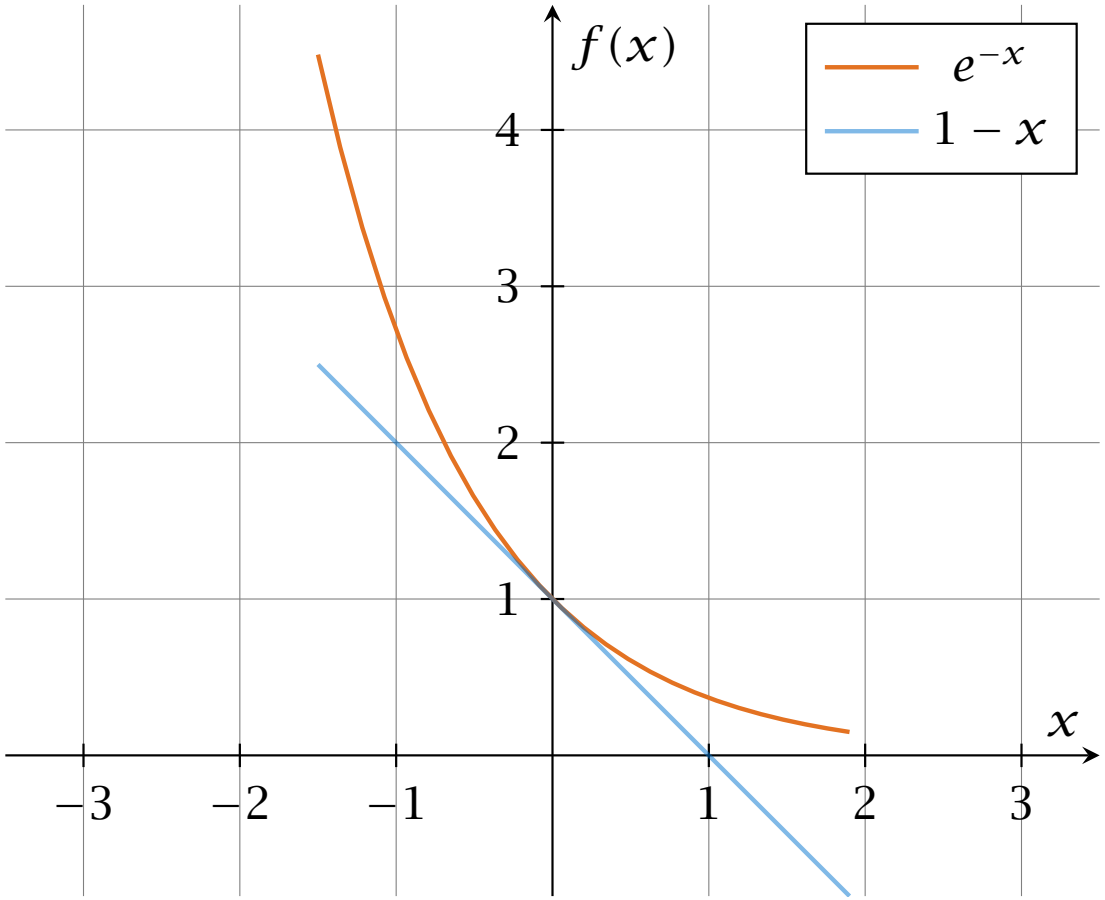
Sei $A_{m,n}$ das Ereignis dass das Einfügen von $m$ Schlüsseln in eine Tabelle der Größe $n$ keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1-\frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

Die erste Gleichung folgt, da das $\ell$-te Element das gehashed wird mit Wahrscheinlichkeit $\frac{n-\ell+1}{n}$ keine Kollision verursacht (unter der Bedingung, das die vorherigen Elemente nicht kollidiert sind). □

# Kollisionen




Die Ungleichung $1 - x \leq e^{-x}$ erhält man wenn man die Taylorexpansion von $e^{-x}$ nach dem 2. Term abbricht.

# Kollisionsauflösung

Es gibt zwei Hauptarten der Kollisionsauflösung

- Offen Adressierung (auch bekannt unter „open addressing“, „closed hashing“)
- Hashing mit Verkettung (auch bekannt unter „hashing with chaining“, „closed addressing“, „open hashing“).

Es gibt Anwendungen (z.B. Computerschach) wo Kollisionen nicht aufgelöst werden.

# Kollisionsauflösung

Es gibt zwei Hauptarten der Kollisionsauflösung

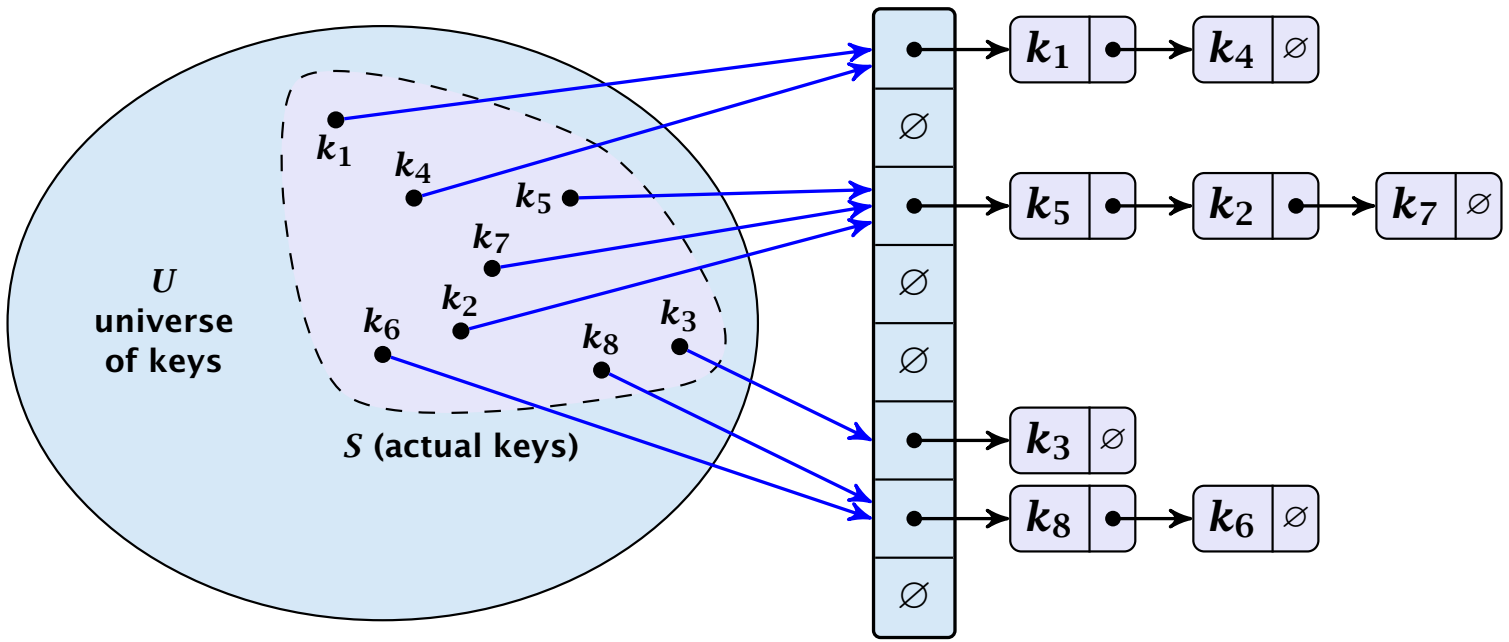
- Offen Adressierung (auch bekannt unter „open addressing“, „closed hashing“)
- Hashing mit Verkettung (auch bekannt unter „hashing with chaining“, „closed addressing“, „open hashing“).

Es gibt Anwendungen (z.B. Computerschach) wo Kollisionen nicht aufgelöst werden.

# Hashing mit Verkettung

Füge Elemente, die auf die gleiche Position abgebildet werden in verkettete Liste ein.

- Search: berechne $h(x)$ und durchsuche Liste nach x->key.
- Insert: Einfügen am Anfang der Liste; $\mathcal{O}(1)$
- Delete: doppelt verkette Liste: $\mathcal{O}(1)$
  einfach verkette Liste: suchen + $\mathcal{O}(1)$



# Hashing mit Verkettung

Im worst-case werden alle Elemente auf eine Liste gemapped.

Laufzeit dann $\Omega(n)$ für search.

**sehr schlecht**

**Auswege:**

- average-case Analyse
- Randomisierung; bestimme erwartete Laufzeit für die Wahl einer zufälligen Hashfunktion aus einer Menge von Hashfunktionen.

# Hashing mit Verkettung

Uniformes Hashing: wähle zufällige Hashfunktion aus Menge aller Funktionen

**Lemma**
Falls $m$ Elemente mittels uniformem Hashing in Hashtabelle der Größe $n$ gespeichert werden dann ist die erwartete Laufzeit einer search-Operation $\mathcal{O}(1 + m/n)$.

**Speicherverbrauch für Hashfunktion zu groß!**

# Beweis

- führe search(k) aus;
- erwartete Laufzeit ist $\mathcal{O}(1 + E[X])$, wobei $X$ Zufallsvariable für Länge der Liste $A[h(k)]$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes Element; $X_e = 1 \Leftrightarrow h(\text{e->key}) = h(k)$
- Listenlänge $X = \sum_e X_e$
- erwartete Listenlänge:

$$E[\textstyle\sum_{e \in S} X_e] = \displaystyle\sum_{e \in S} E[X_e] = \sum_{e \in S} \Pr[X_e = 1] = \sum_{e \in S} 1/n = m/n$$

# c-universelles Hashing

**Definition**

Eine Familie $\mathcal{H}$ von Hashfunktionen auf $\{0,\ldots,n-1\}$ heißt $c$-universell falls für jedes Schlüsselpaar $k_1, k_2$ gilt, dass

$$|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}| \leq \frac{c}{n}|\mathcal{H}|$$

D.h. bei zufälliger Wahl der Hashfunktion gilt

$$\Pr[h(k_1) = h(k_2)] \leq \frac{c}{n}$$

# c-universelles Hashing

**Lemma**

Falls $m$ Elemente mittels einer zufälligen Hashfunktion aus einer $c$-universellen Klasse in Hashtabelle der Größe $n$ gespeichert werden dann ist die erwartete Laufzeit einer search-Operation $\mathcal{O}(1 + cm/n)$.

# Beweis

- $X_e \in \{0,1\}$; $X_e = 1 \Leftrightarrow h(\text{e->key} = h(k))$
- Listenlänge für Schlüssel $k$ ist: $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge

$$E[X] = \sum[\sum_{e \in S} X_e] = \sum_{e \in S} E[X_e]$$

$$= \sum_{e \in S} \Pr[X_e = 1] \leq \sum_{e \in S} c/n = cm/n$$

# Hashing mit Verkettung

**Nachteile:**

- Zeiger erhöhen Speicherverbrauch
- schlechte Cache-Effizienz

**Vorteile:**

- kein festes Limit für die Anzahl der Elemente
- Löschen kann effizient implementiert werden

# Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im $j$-ten Schritt untersucht wird. Werte $h(k,0),\ldots,h(k,n-1)$ muss Permutation von $0,\ldots,n-1$ sein.

**Search($k$):** Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ....

**Insert($x$):** Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

# Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im $j$-ten Schritt untersucht wird. Werte $h(k,0),\ldots,h(k,n-1)$ muss Permutation von $0,\ldots,n-1$ sein.

**Search($k$):** Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ....

**Insert($x$):** Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

# Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im $j$-ten Schritt untersucht wird. Werte $h(k,0),\ldots,h(k,n-1)$ muss Permutation von $0,\ldots,n-1$ sein.

**Search($k$):** Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ....

**Insert($x$):** Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

# Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im $j$-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von $0$,...,$n-1$ sein.

**Search($k$):** Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ....

**Insert($x$):** Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

# Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im $j$-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von $0$,...,$n-1$ sein.

**Search($k$):** Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ....

**Insert($x$):** Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

# Offene Adressierung

Möglichkeiten für $h(k,j)$:

- Lineares Sondieren:
  $h(k,i) = h(k) + i \mod n$
  (manchmal: $h(k,i) = h(k) + ci \mod n$).
- Quadratisches Sondieren:
  $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Doppeltes Hashing:
  $h(k,i) = h_1(k) + i h_2(k) \mod n$.

# Offene Adressierung

Möglichkeiten für $h(k,j)$:

- Lineares Sondieren:
  $h(k,i) = h(k) + i \mod n$
  (manchmal: $h(k,i) = h(k) + ci \mod n$).
- Quadratisches Sondieren:
  $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Doppeltes Hashing:
  $h(k,i) = h_1(k) + i h_2(k) \mod n$.

# Offene Adressierung

Möglichkeiten für $h(k,j)$:

- Lineares Sondieren:
  $h(k,i) = h(k) + i \mod n$
  (manchmal: $h(k,i) = h(k) + ci \mod n$).
- Quadratisches Sondieren:
  $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Doppeltes Hashing:
  $h(k,i) = h_1(k) + i h_2(k) \mod n$.

# Offen Adressierung

**Nachteile:**

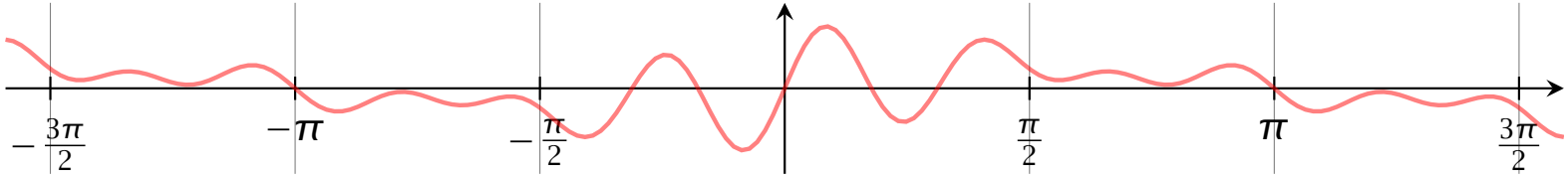
- nachträgliche Änderung der Tabellengröße sehr aufwändig
- Löschen schwierig

**Vorteile:**

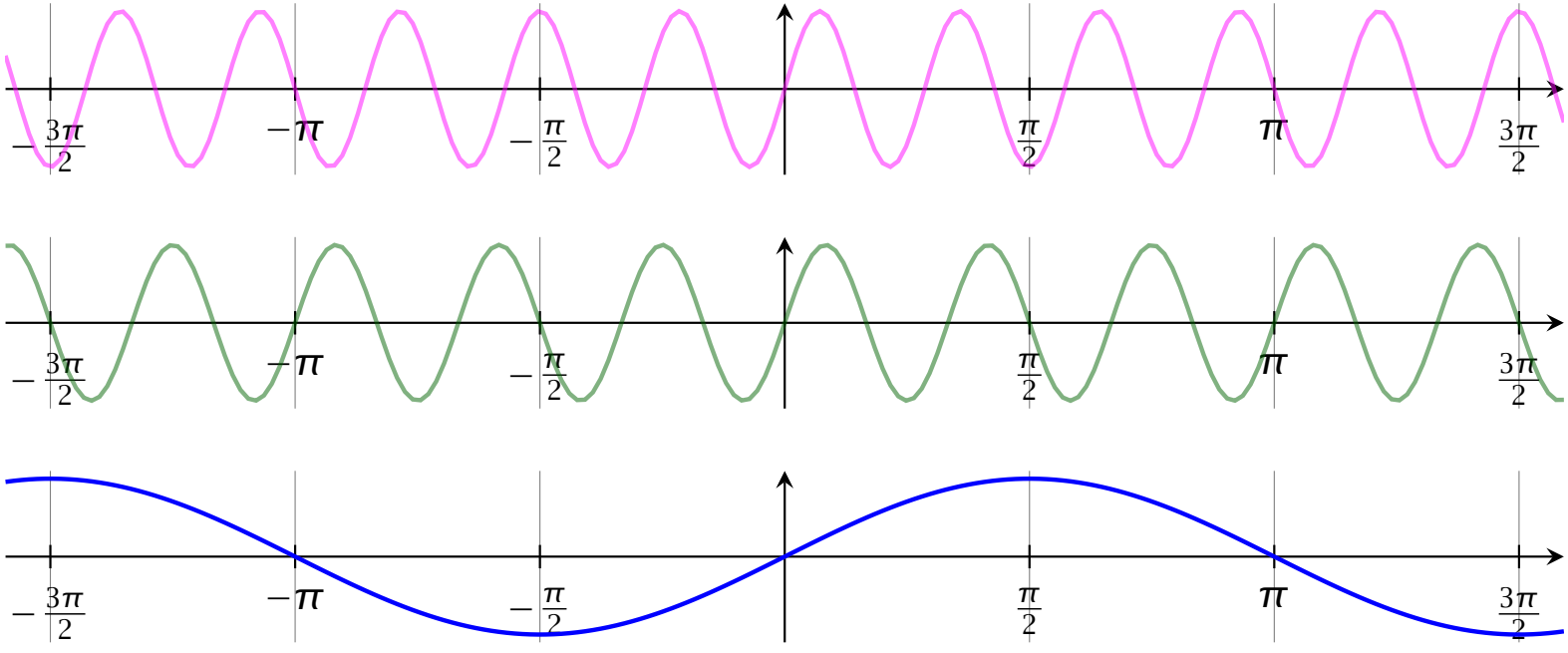
- Lineares Sondierung ist Cache-effizient;
- keine Zeiger; geringerer Speicherverbrauch

# Fourieranalyse

## Eingabe: Signal (periodisch)




## Ausgabe: Zerlegung in Teilfrequenzen



# Fourieranalyse

Im folgenden bezeichnet $j$ die imaginäre Einheit.

## Eingabe

- $x(t)$, periodisches Signal
- $x(t)$ ist Überlagerung von endlich vielen skalierten Sinus- und Kosinusfunktionen unterschiedlicher Frequenzen $\omega_s \geq 0$

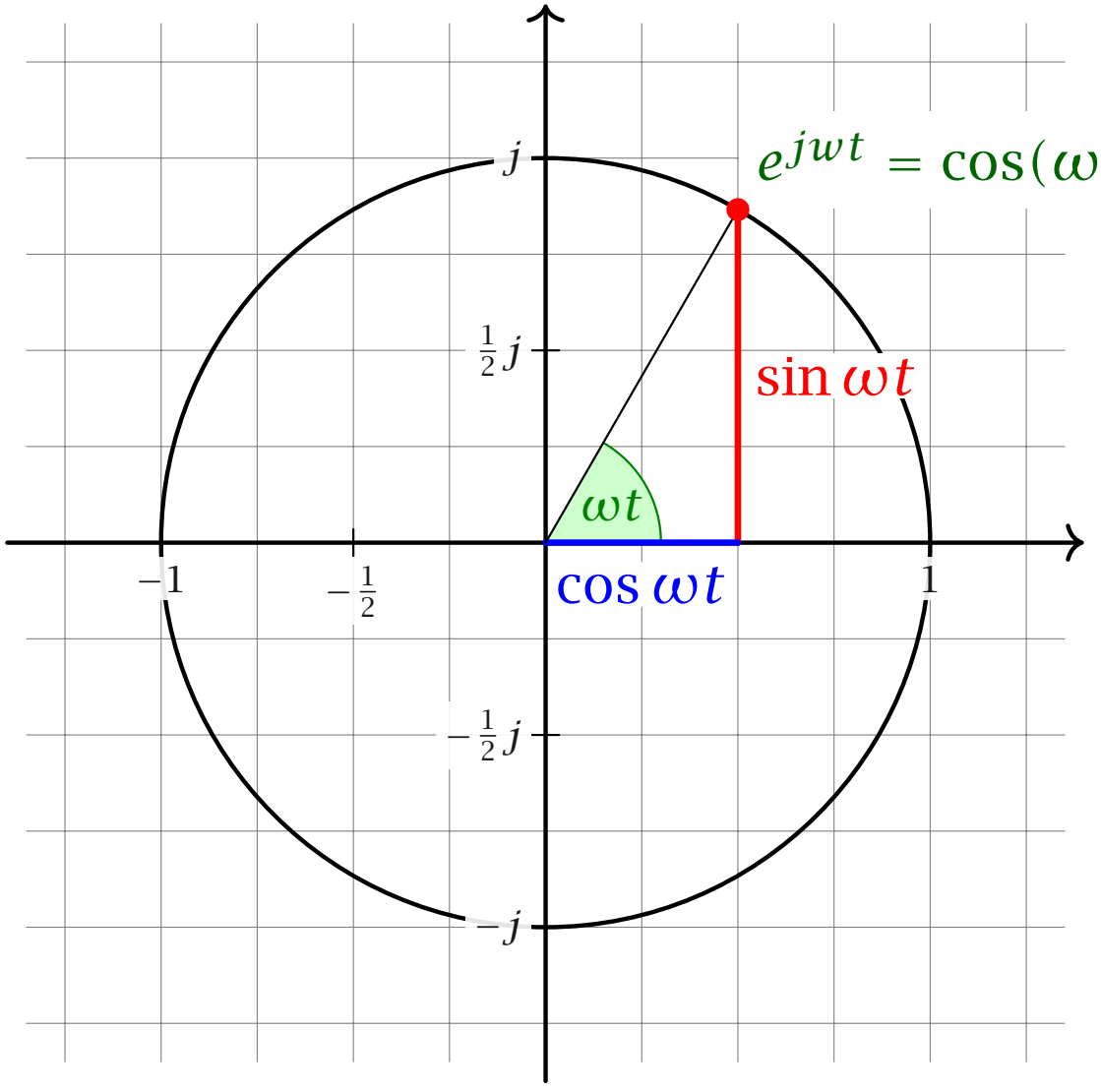
## Ausgabe

- schreibe $x(t)$ als

$$x(t) = \sum_i \hat{x}[\omega_i] \cdot e^{j\omega_i t}$$

- Beachte: für jede Frequenz $\omega_s$, die in $x(t)$ vorkommt gibt es in der obigen Formel zwei „Frequenzen“, $\omega_{i_1} = \omega_s$ und $\omega_{i_2} = -\omega_s$

# Euler's Formula



# Euler's Formula

Es gilt

$$e^{jwt} = \cos(\omega t) + j\sin(\omega t)$$

Damit auch

$$\cos(\omega t) = \frac{1}{2}\left(e^{j\omega t} + e^{-j\omega t}\right)$$

$$\sin(\omega t) = \frac{1}{2j}\left(e^{j\omega t} - e^{-j\omega t}\right)$$

D.h. wenn mein Signal aus Überlagerung von Sinus- und Kosinusfunktionen besteht, kann ich es in die geforderte Form bringen.

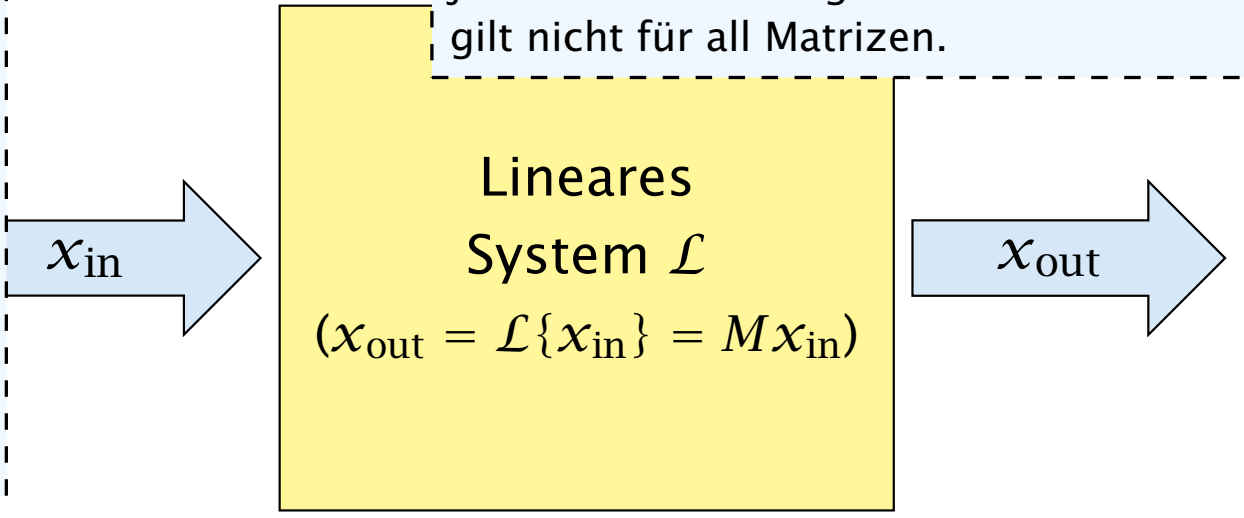
# Warum sollte ich das tun?

Komplexe Exponentialfunktionen sind Eigenfunktionen von linear zeitinvarianten Systemen.

# Vektoren

Wenn wir Vektoren haben ($x_{\text{in}} \in \mathbb{R}^n$ und $x_{\text{out}} \in \mathbb{R}^n$) ist ein lineares System einfach eine Matrixmultiplikation mit einer $n \times n$ Matrix $M$.

Dieses Verfahren ist natürlich nur möglich wenn die Matrix $M$ eine Basis von Eigenvektoren besitzt, so dass man jeden Vektor in die gewünschte Form bringen kann. Dies gilt nicht für all Matrizen.




- linear: $\mathcal{L}\{\alpha x_{\text{in}}\} = \alpha\mathcal{L}\{x_{\text{in}}\}$; $\mathcal{L}\{x_{\text{in}} + y_{\text{in}}\} = \mathcal{L}\{x_{\text{in}}\} + \mathcal{L}\{y_{\text{in}}\}$
- zerlege $x_{\text{in}}$ in Eigenvektoren, d.h., schreibe $x_{\text{in}} = \sum_i \alpha_i v_i$, wobei $v_i$ Eigenvektoren von $M$ sind.
- dann gilt:

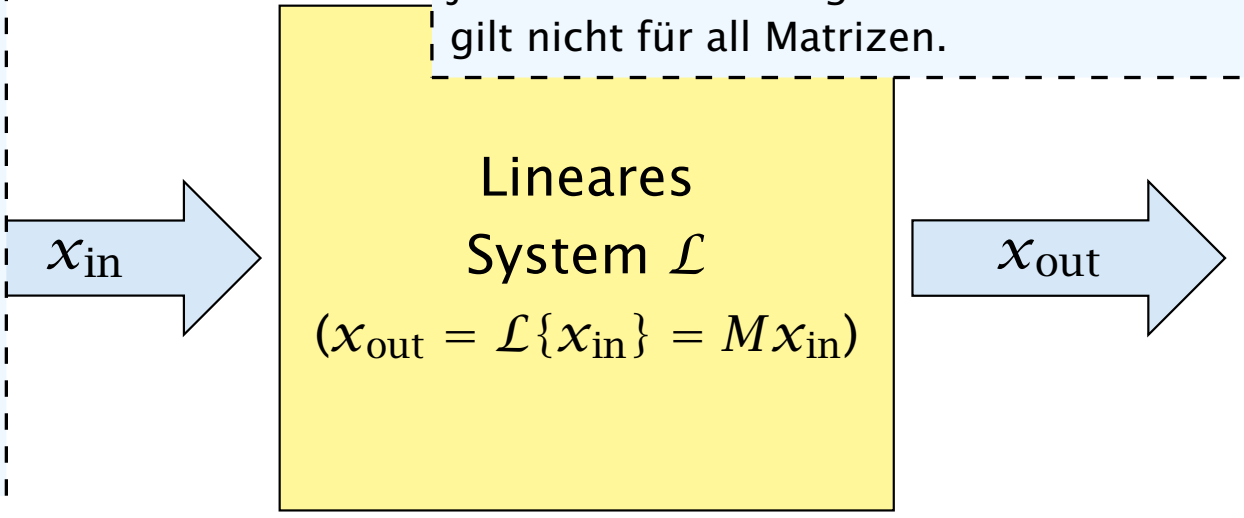
$$x_{\text{out}} = \mathcal{L}\left\{\sum_i \alpha_i v_i\right\} = \sum_i \alpha_i \mathcal{L}\{v_i\} = \sum_i \alpha_i \lambda_i v_i$$

wobei $\lambda_i$ Eigenwert zum Eigenvektor $v_i$.

# Vektoren

Wenn wir Vektoren haben ($x_{\text{in}} \in \mathbb{R}^n$ und $x_{\text{out}} \in \mathbb{R}^n$) ist ein lineares System einfach eine Matrixmultiplikation mit einer $n \times n$ Matrix $M$.

Dieses Verfahren ist natürlich nur möglich wenn die Matrix $M$ eine Basis von Eigenvektoren besitzt, so dass man jeden Vektor in die gewünschte Form bringen kann. Dies gilt nicht für all Matrizen.




- linear: $\mathcal{L}\{\alpha x_{\text{in}}\} = \alpha\mathcal{L}\{x_{\text{in}}\}$; $\mathcal{L}\{x_{\text{in}} + y_{\text{in}}\} = \mathcal{L}\{x_{\text{in}}\} + \mathcal{L}\{y_{\text{in}}\}$
- zerlege $x_{\text{in}}$ in Eigenvektoren, d.h., schreibe $x_{\text{in}} = \sum_i \alpha_i v_i$, wobei $v_i$ Eigenvektoren von $M$ sind.
- dann gilt:

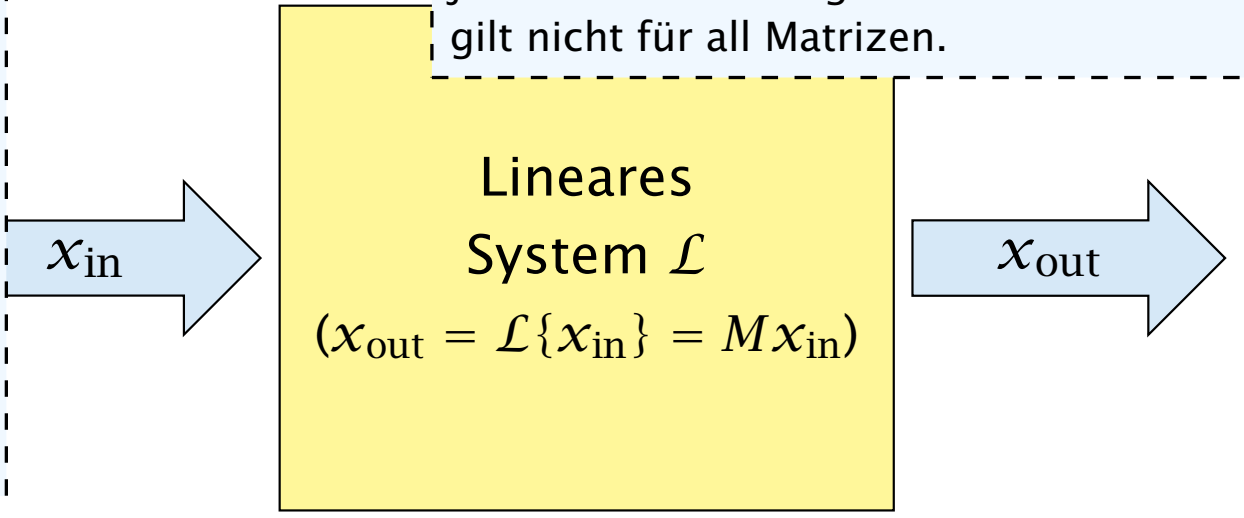
$$x_{\text{out}} = \mathcal{L}\left\{\sum_i \alpha_i v_i\right\} = \sum_i \alpha_i \mathcal{L}\{v_i\} = \sum_i \alpha_i \lambda_i v_i$$

wobei $\lambda_i$ Eigenwert zum Eigenvektor $v_i$.

# Vektoren

Wenn wir Vektoren haben ($x_{\text{in}} \in \mathbb{R}^n$ und $x_{\text{out}} \in \mathbb{R}^n$) ist ein lineares System einfach eine Matrixmultiplikation mit einer $n \times n$ Matrix $M$.

Dieses Verfahren ist natürlich nur möglich wenn die Matrix $M$ eine Basis von Eigenvektoren besitzt, so dass man jeden Vektor in die gewünschte Form bringen kann. Dies gilt nicht für all Matrizen.

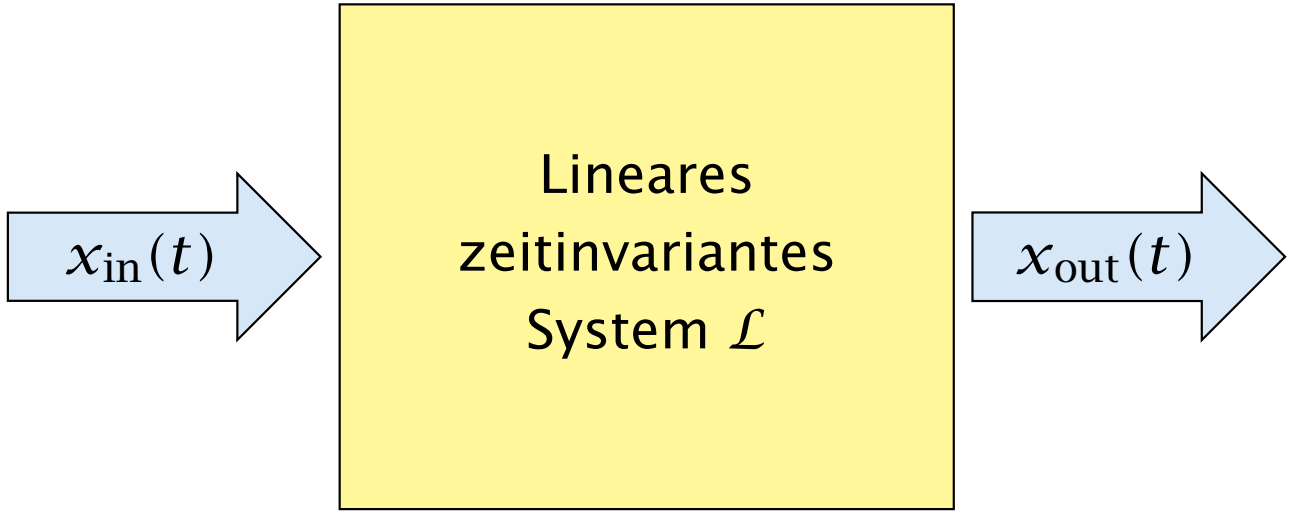



- linear: $\mathcal{L}\{\alpha x_{\text{in}}\} = \alpha\mathcal{L}\{x_{\text{in}}\}$; $\mathcal{L}\{x_{\text{in}} + y_{\text{in}}\} = \mathcal{L}\{x_{\text{in}}\} + \mathcal{L}\{y_{\text{in}}\}$
- zerlege $x_{\text{in}}$ in Eigenvektoren, d.h., schreibe $x_{\text{in}} = \sum_i \alpha_i v_i$, wobei $v_i$ Eigenvektoren von $M$ sind.
- dann gilt:

$$x_{\text{out}} = \mathcal{L}\left\{\sum_i \alpha_i v_i\right\} = \sum_i \alpha_i \mathcal{L}\{v_i\} = \sum_i \alpha_i \lambda_i v_i$$

wobei $\lambda_i$ Eigenwert zum Eigenvektor $v_i$.

# Periodische Funktionen




- zerlege $x_{\text{in}}(t)$ in Eigenfunktionen, d.h., schreibe $x_{\text{in}}(t) = \sum_i \alpha_i f_i(t)$, wobei $f_i$ Eigenfunktionen von $\mathcal{L}$ sind.
- dann gilt:

$$x_{\text{out}}(t) = \mathcal{L}\left\{\sum_i \alpha_i f_i(t)\right\} = \sum_i \alpha_i \mathcal{L}\{f_i(t)\} = \sum_i \alpha_i \lambda_i f_i(t)$$

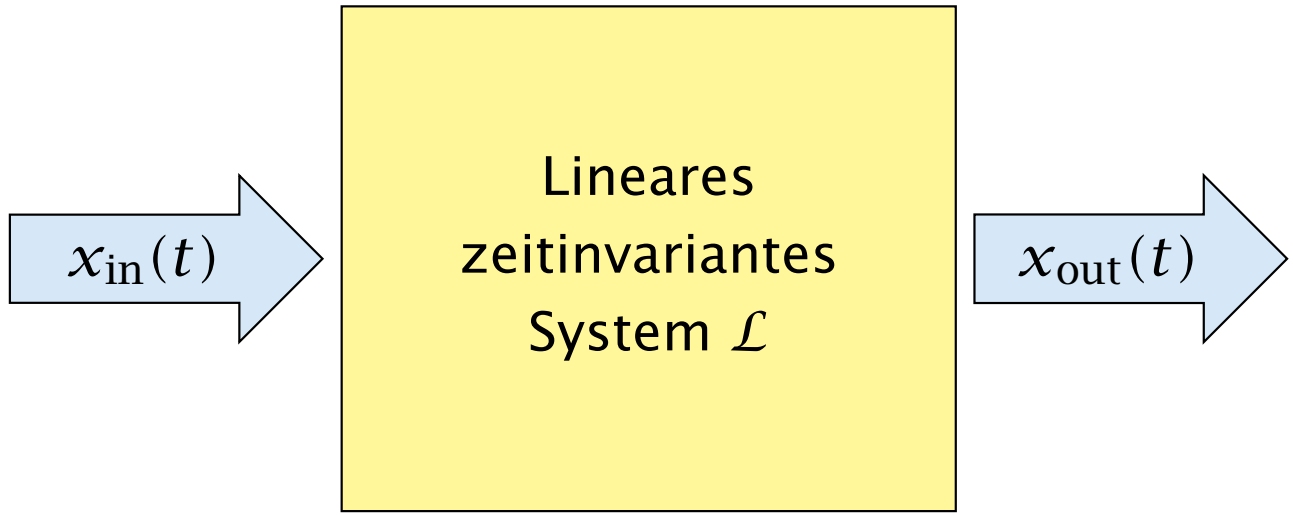
wobei $\lambda_i$ Eigenwert zur Eigenfunktion $f_i(t)$ ist.

$f(t)$ ist Eigenfunktion zum Eigenwert $\lambda$ falls $\mathcal{L}\{f(t)\} = \lambda f(t)$.

Das ist natürlich nur möglich, wenn wir die Funktion $x_{\text{in}}(t)$ als Summe von Eigenfunktionen von $\mathcal{L}$ ausdrücken können.

# Periodische Funktionen

$f(t)$ ist Eigenfunktion zum Eigenwert $\lambda$ falls $\mathcal{L}\{f(t)\} = \lambda f(t)$.




- zerlege $x_{\text{in}}(t)$ in Eigenfunktionen, d.h., schreibe $x_{\text{in}}(t) = \sum_i \alpha_i f_i(t)$, wobei $f_i$ Eigenfunktionen von $\mathcal{L}$ sind.
- dann gilt:

$$x_{\text{out}}(t) = \mathcal{L}\left\{\sum_i \alpha_i f_i(t)\right\} = \sum_i \alpha_i \mathcal{L}\{f_i(t)\} = \sum_i \alpha_i \lambda_i f_i(t)$$

wobei $\lambda_i$ Eigenwert zur Eigenfunktion $f_i(t)$ ist.

Das ist natürlich nur möglich, wenn wir die Funktion $x_{\text{in}}(t)$ als Summe von Eigenfunktionen von $\mathcal{L}$ ausdrücken können.

# Periodische Funktionen

Das heißt wenn wir periodische Signale haben benötigen wir nur eine Teilmenge der Eigenfunktionen. Insbesondere ist der Exponent rein imaginär.

**Theorem**

Komplexe Exponentialfunktionen ($f(t) = e^{st}$) sind Eigenfunktionen von linear zeitinvarianten Systemen.

**Theorem**

Jede periodische Funktion $x(t)$ mit Periode $T$ (d.h., $x(t+T) = x(t)$ für alle $t$) läßt sich als

$$x(t) = \sum_{i=-\infty}^{\infty} \alpha_i e^{j\frac{2\pi i}{T}t}$$

schreiben. Im folgenden $w := 2\pi/T$ und $f_i(t) = e^{j\omega it}$.

Die periodischen Signale müssen hinreichend gutartig sein. Stetig und differenzierbar reicht z.B. aus.

# Periodische Funktionen

Das heißt wenn wir periodische Signale haben benötigen wir nur eine Teilmenge der Eigenfunktionen. Insbesondere ist der Exponent rein imaginär.

**Theorem**

Komplexe Exponentialfunktionen ($f(t) = e^{st}$) sind Eigenfunktionen von linear zeitinvarianten Systemen.

**Theorem**

Jede periodische Funktion $x(t)$ mit Periode $T$ (d.h., $x(t+T) = x(t)$ für alle $t$) läßt sich als

$$x(t) = \sum_{i=-\infty}^{\infty} \alpha_i e^{j\frac{2\pi i}{T}t}$$

schreiben. Im folgenden $w := 2\pi/T$ und $f_i(t) = e^{j\omega it}$.

Die periodischen Signale müssen hinreichend gutartig sein. Stetig und differenzierbar reicht z.B. aus.

# Bestimmung der $\alpha_i$ — Vektoren

Wie schreibe ich einen Vektor $x$ als $\sum_i \alpha_i v_i$?

Bestimme eine Orthonormalbasis von Eigenvektoren, $v_1, \ldots, v_n$.

- Vektoren sind auf 1 normiert: $\|v_i\| = \langle v_i, v_i \rangle = 1$, wobei $\langle \cdot, \cdot \rangle$ das Standardskalarprodukt ist.
- Vektoren sind orthogonal, d.h. für $i \neq j$ gilt $\langle v_i, v_j \rangle = 0$.

dann gilt $\alpha_i = \langle v_i, x \rangle$, da

$$\langle v_i, x \rangle = \langle v_i, \sum_j \alpha_j v_j \rangle = \sum_j \alpha_j \langle v_i, v_j \rangle = \alpha_i$$

Das Standardskalarprodukt für einen Vektorraum über $\mathbb{C}$ ist $\langle x, y \rangle = \sum_i \bar{x}_i y_i$.

# Bestimmung der $\alpha_i$ — Vektoren

Wie schreibe ich einen Vektor $x$ als $\sum_i \alpha_i v_i$?

Bestimme eine Orthonormalbasis von Eigenvektoren, $v_1, \ldots, v_n$.

- Vektoren sind auf 1 normiert: $\|v_i\| = \langle v_i, v_i \rangle = 1$, wobei $\langle \cdot, \cdot \rangle$ das Standardskalarprodukt ist.
- Vektoren sind orthogonal, d.h. für $i \neq j$ gilt $\langle v_i, v_j \rangle = 0$.

dann gilt $\alpha_i = \langle v_i, x \rangle$, da

$$\langle v_i, x \rangle = \langle v_i, \sum_j \alpha_j v_j \rangle = \sum_j \alpha_j \langle v_i, v_j \rangle = \alpha_i$$

Das Standardskalarprodukt für einen Vektorraum über $\mathbb{C}$ ist $\langle x, y \rangle = \sum_i \bar{x}_i y_i$.

# Bestimmung der $\alpha_i$ — Vektoren

Wie schreibe ich einen Vektor $x$ als $\sum_i \alpha_i v_i$?

Bestimme eine Orthonormalbasis von Eigenvektoren, $v_1, \ldots, v_n$.

- Vektoren sind auf 1 normiert: $\|v_i\| = \langle v_i, v_i \rangle = 1$, wobei $\langle \cdot, \cdot \rangle$ das Standardskalarprodukt ist.
- Vektoren sind orthogonal, d.h. für $i \neq j$ gilt $\langle v_i, v_j \rangle = 0$.

dann gilt $\alpha_i = \langle v_i, x \rangle$, da

$$\langle v_i, x \rangle = \langle v_i, \sum_j \alpha_j v_j \rangle = \sum_j \alpha_j \langle v_i, v_j \rangle = \alpha_i$$

Das Standardskalarprodukt für einen Vektorraum über $\mathbb{C}$ ist $\langle x, y \rangle = \sum_i \overline{x}_i y_i$.

# Bestimmung der $\alpha_i$ — Funktionen

Wie schreibe ich ein Signal $x(t)$ als $\sum_i \alpha_i f_i(t)$?

Definiere ein Skalarprodukt, so dass $f_i(t)$'s orthonormal sind.

Auf der Menge der $T$-periodischen Funktionen definiert

$$\langle a(t), b(t) \rangle := \frac{1}{T} \int_0^T \overline{a(t)} b(t) \mathrm{d}t$$

ein Skalarprodukt.

# Bestimmung der $\alpha_i$ — Funktionen

Wie schreibe ich ein Signal $x(t)$ als $\sum_i \alpha_i f_i(t)$?

Definiere ein Skalarprodukt, so dass $f_i(t)$'s orthonormal sind.

Auf der Menge der $T$-periodischen Funktionen definiert

$$\langle a(t), b(t) \rangle := \frac{1}{T} \int_0^T \overline{a(t)} b(t) \mathrm{d}t$$

ein Skalarprodukt.

# Bestimmung der $\alpha_i$ — Funktionen

Wie schreibe ich ein Signal $x(t)$ als $\sum_i \alpha_i f_i(t)$?

Definiere ein Skalarprodukt, so dass $f_i(t)$'s orthonormal sind.

Auf der Menge der $T$-periodischen Funktionen definiert

$$\langle a(t), b(t) \rangle := \frac{1}{T} \int_0^T \overline{a(t)} b(t) \mathrm{d}t$$

ein Skalarprodukt.

# Bestimmung der $\alpha_i$ — Funktionen

**Skalarprodukt**

1. $\langle x(t), x(t)\rangle \geq 0$
2. $\langle x(t), x(t)\rangle = 0 \Longleftrightarrow x(t) = 0$
3. $\langle x(t), y(t)\rangle = \overline{\langle y(t), x(t)\rangle}$
4. $\langle x(t), \alpha y(t)\rangle = \alpha\langle x(t), y(t)\rangle$
   $\langle x(t), y(t) + z(t)\rangle = \langle x(t), y(t)\rangle + \langle x(t), z(t)\rangle$

1., 3., und 4. folgend direkt aus Integraleigenschaften. Für 2. muß man den Funktionenraum geeignet einschränken.

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle = \frac{1}{T} \int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t} \mathrm{d}t$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\begin{aligned}\langle f_i(t), f_\ell(t)\rangle &= \frac{1}{T}\int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t}\,\mathrm{d}t \\ &= \frac{1}{T}\int_0^T e^{j\omega(\ell-i)t}\,\mathrm{d}t\end{aligned}$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle = \frac{1}{T} \int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t} \mathrm{d}t$$

$$= \frac{1}{T} \int_0^T e^{j\omega(\ell - i)t} \mathrm{d}t$$

**$i = \ell$:**

$$= \frac{1}{T} \int_0^T 1 \mathrm{d}t = 1$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle = \frac{1}{T} \int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t} \mathrm{d}t$$

$$= \frac{1}{T} \int_0^T e^{j\omega(\ell - i)t} \mathrm{d}t$$

**$i = \ell$:**

$$= \frac{1}{T} \int_0^T 1 \mathrm{d}t = 1$$

**$i \neq \ell$:** $(\beta := j\omega(\ell - i) \neq 0)$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle = \frac{1}{T} \int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t} \mathrm{d}t$$

$$= \frac{1}{T} \int_0^T e^{j\omega(\ell - i)t} \mathrm{d}t$$

**$i = \ell$:**

$$= \frac{1}{T} \int_0^T 1 \mathrm{d}t = 1$$

**$i \neq \ell$:** $(\beta := j\omega(\ell - i) \neq 0)$

$$= \frac{1}{T} \int_0^T e^{\beta t}$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle = \frac{1}{T} \int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t} \mathrm{d}t$$
$$= \frac{1}{T} \int_0^T e^{j\omega(\ell - i)t} \mathrm{d}t$$

**$i = \ell$:**

$$= \frac{1}{T} \int_0^T 1 \mathrm{d}t = 1$$

**$i \neq \ell$:** $(\beta := j\omega(\ell - i) \neq 0)$

$$= \frac{1}{T} \int_0^T e^{\beta t} = \frac{1}{T} \left[ \frac{1}{\beta} e^{\beta t} \right]_0^T$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t) \rangle = \frac{1}{T} \int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t} \mathrm{d}t$$
$$= \frac{1}{T} \int_0^T e^{j\omega(\ell - i)t} \mathrm{d}t$$

**$i = \ell$:**

$$= \frac{1}{T} \int_0^T 1 \mathrm{d}t = 1$$

**$i \neq \ell$:** $(\beta := j\omega(\ell - i) \neq 0)$

$$= \frac{1}{T} \int_0^T e^{\beta t} = \frac{1}{T} \left[ \frac{1}{\beta} e^{\beta t} \right]_0^T = \frac{1}{T\beta} - \frac{1}{T\beta} = 0$$

# Bestimmung der $\alpha_i$ — Funktionen

$$\langle f_i(t), f_\ell(t)\rangle = \frac{1}{T}\int_0^T \overline{e^{j\omega it}} e^{j\omega \ell t}\,\mathrm{d}t$$
$$= \frac{1}{T}\int_0^T e^{j\omega(\ell-i)t}\,\mathrm{d}t$$

$\boldsymbol{i = \ell}$:

$$= \frac{1}{T}\int_0^T 1\,\mathrm{d}t = 1$$

$\boldsymbol{i \neq \ell}$: $\quad(\beta := j\omega(\ell - i) \neq 0)$

$$= \frac{1}{T}\int_0^T e^{\beta t} = \frac{1}{T}\left[\frac{1}{\beta}e^{\beta t}\right]_0^T = \frac{1}{T\beta} - \frac{1}{T\beta} = 0$$

da $e^{\beta T} = e^{j2\pi(\ell-i)} = 1$.

# Bestimmung der $\alpha_i$ — Funktionen

Damit gilt $\alpha_i = \langle f_i(t), x(t) \rangle$, d.h.

$$\alpha_i = \frac{1}{T} \int_0^T x(t) e^{-j\omega it} \mathrm{d}t$$

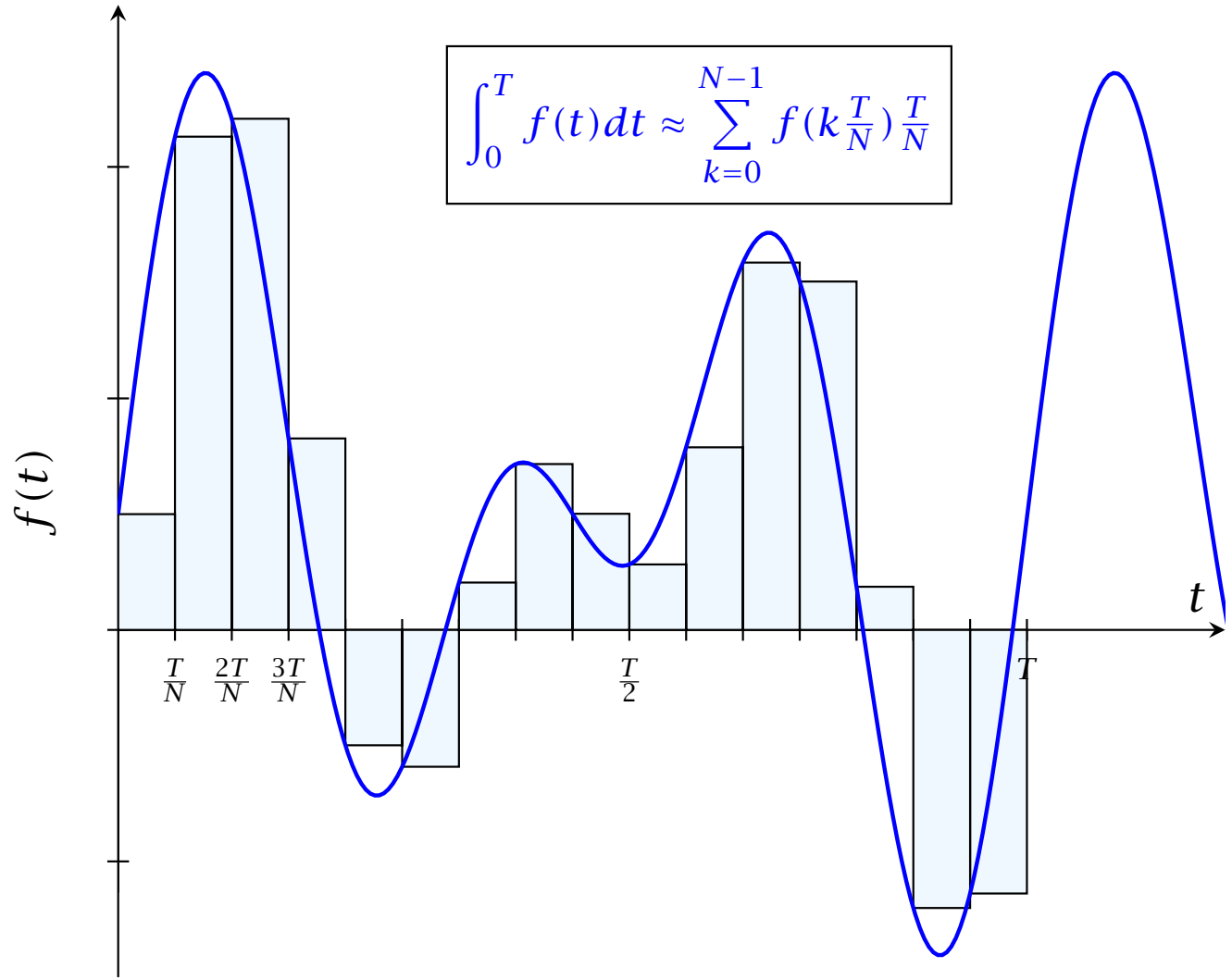
da

$$\langle f_i(t), x(t) \rangle = \langle f_i(t), \sum_\ell \alpha_\ell f_\ell(t) \rangle = \sum_\ell \alpha_\ell \langle f_i(t), f_\ell(t) \rangle = \alpha_i$$

**leider muß man hier integrieren...**

Wenn mein Signal $x(t)$ aus nur wenigen Frequenzen besteht, dann ist der eigentliche „Informationsinhalt“ des Signals sehr gering. Hier muß ich aber trotzdem das Signal über dem gesamten Intervall $[0, T]$ auswerten um $\alpha_i$ zu bestimmen. Das ist sehr ineffizient.

# Approximation des Integrals



# Bestimmung der $\alpha_i$

Angenommen mein Signal ist

$$x(t) = \sum_{\ell} \alpha_\ell e^{j\omega\ell t}$$

aber ich sehe nur das Gesamtsignal.

Bilde

# Bestimmung der $\alpha_i$

Angenommen mein Signal ist

$$x(t) = \sum_{\ell} \alpha_\ell e^{j\omega\ell t}$$

aber ich sehe nur das Gesamtsignal.

Bilde

$$\frac{1}{T}\sum_{k=0}^{N-1} x(\tfrac{T}{N}k)e^{-j\omega i\frac{T}{N}k}\frac{T}{N} = \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\omega\ell\frac{T}{N}k}e^{-j\omega i\frac{T}{N}k}\frac{T}{N}$$

$$= \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\frac{2\pi}{T}\ell\frac{T}{N}k}e^{-j\frac{2\pi}{T}i\frac{T}{N}k}\frac{T}{N}$$

$$= \sum_{i}\alpha_i\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} \stackrel{?}{=} \alpha_i$$

# Bestimmung der $\alpha_i$

Angenommen mein Signal ist

$$x(t) = \sum_{\ell} \alpha_\ell e^{j\omega\ell t}$$

aber ich sehe nur das Gesamtsignal.

Bilde

$$\frac{1}{T}\sum_{k=0}^{N-1} x(\tfrac{T}{N}k)e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N} = \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\omega\ell\frac{T}{N}k}e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N}$$

$$= \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\frac{2\pi}{T}\ell\frac{T}{N}k}e^{-j\frac{2\pi}{T}i\frac{T}{N}k}\tfrac{T}{N}$$

$$= \sum_{i}\alpha_i\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} \stackrel{?}{=} \alpha_i$$

# Bestimmung der $\alpha_i$

Angenommen mein Signal ist

$$x(t) = \sum_{\ell} \alpha_\ell e^{j\omega\ell t}$$

aber ich sehe nur das Gesamtsignal.

Bilde

$$\frac{1}{T}\sum_{k=0}^{N-1} x(\tfrac{T}{N}k)e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N} = \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\omega\ell\frac{T}{N}k}e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N}$$

$$= \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\frac{2\pi}{T}\ell\frac{T}{N}k}e^{-j\frac{2\pi}{T}i\frac{T}{N}k}\tfrac{T}{N}$$

$$= \sum_{i}\alpha_i\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} \stackrel{?}{=} \alpha_i$$

# Bestimmung der $\alpha_i$

Angenommen mein Signal ist

$$x(t) = \sum_{\ell} \alpha_\ell e^{j\omega\ell t}$$

aber ich sehe nur das Gesamtsignal.

Bilde

$$\frac{1}{T}\sum_{k=0}^{N-1} x(\tfrac{T}{N}k)e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N} = \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\omega\ell\frac{T}{N}k}e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N}$$

$$= \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\frac{2\pi}{T}\ell\frac{T}{N}k}e^{-j\frac{2\pi}{T}i\frac{T}{N}k}\tfrac{T}{N}$$

$$= \sum_{i}\alpha_i\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} \stackrel{?}{=} \alpha_i$$

# Bestimmung der $\alpha_i$

Angenommen mein Signal ist

$$x(t) = \sum_{\ell} \alpha_\ell e^{j\omega\ell t}$$

aber ich sehe nur das Gesamtsignal.

Bilde

$$\begin{aligned}
\frac{1}{T}\sum_{k=0}^{N-1} x(\tfrac{T}{N}k)e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N} &= \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\omega\ell\frac{T}{N}k}e^{-j\omega i\frac{T}{N}k}\tfrac{T}{N} \\
&= \frac{1}{T}\sum_{k=0}^{N-1}\sum_{\ell}\alpha_\ell e^{j\frac{2\pi}{T}\ell\frac{T}{N}k}e^{-j\frac{2\pi}{T}i\frac{T}{N}k}\tfrac{T}{N} \\
&= \sum_{i}\alpha_i\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} \stackrel{?}{=} \alpha_i
\end{aligned}$$

# Wie finde ich den Koeffizienten für Frequenz $\omega_\ell$?

**Für $i \neq \ell \mod N$ gilt**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} q^k \qquad \text{mit } q = e^{j\frac{2\pi}{N}(\ell-i)} \neq 1$$

$$= \frac{1}{N}\frac{q^N-1}{q-1} = 0 \qquad \text{da } q^N = 1$$

**Für $i = \ell \mod N$**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} 1 = 1$$

Sei $\frac{2\pi}{N}s$ größte Frequenz die vorkommt. Wir wählen $N \geq 2s+1$.

**Dann bekommt man $\alpha_i$ exakt!!!**

# Wie finde ich den Koeffizienten für Frequenz $\omega_\ell$?

**Für $i \neq \ell \mod N$ gilt**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} q^k \qquad \text{mit } q = e^{j\frac{2\pi}{N}(\ell-i)} \neq 1$$

$$= \frac{1}{N}\frac{q^N-1}{q-1} = 0 \qquad \text{da } q^N = 1$$

**Für $i = \ell \mod N$**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} 1 = 1$$

Sei $\frac{2\pi}{N}s$ größte Frequenz die vorkommt. Wir wählen $N \geq 2s+1$.

**Dann bekommt man $\alpha_i$ exakt!!!**

# Wie finde ich den Koeffizienten für Frequenz $\omega_\ell$?

**Für $i \neq \ell \mod N$ gilt**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} q^k \qquad \text{mit } q = e^{j\frac{2\pi}{N}(\ell-i)} \neq 1$$

$$= \frac{1}{N}\frac{q^N-1}{q-1} = 0 \qquad \text{da } q^N = 1$$

**Für $i = \ell \mod N$**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} 1 = 1$$

Sei $\frac{2\pi}{N}s$ größte Frequenz die vorkommt. Wir wählen $N \geq 2s+1$.

**Dann bekommt man $\alpha_i$ exakt!!!**

# Wie finde ich den Koeffizienten für Frequenz $\omega_\ell$?

**Für $i \neq \ell \mod N$ gilt**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} q^k \qquad \text{mit } q = e^{j\frac{2\pi}{N}(\ell-i)} \neq 1$$

$$= \frac{1}{N}\frac{q^N-1}{q-1} = 0 \qquad \text{da } q^N = 1$$

**Für $i = \ell \mod N$**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} 1 = 1$$

Sei $\frac{2\pi}{N}s$ größte Frequenz die vorkommt. Wir wählen $N \geq 2s+1$.
**Dann bekommt man $\alpha_i$ exakt!!!**

# Wie finde ich den Koeffizienten für Frequenz $\omega_\ell$?

**Für $i \neq \ell \mod N$ gilt**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} q^k \qquad \text{mit } q = e^{j\frac{2\pi}{N}(\ell-i)} \neq 1$$

$$= \frac{1}{N}\frac{q^N-1}{q-1} = 0 \qquad \text{da } q^N = 1$$

**Für $i = \ell \mod N$**

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}(\ell-i)k} = \frac{1}{N}\sum_{k=0}^{N-1} 1 = 1$$

Sei $\frac{2\pi}{N}s$ größte Frequenz die vorkommt. Wir wählen $N \geq 2s+1$.

**Dann bekommt man $\alpha_i$ exakt!!!**

# Diskrete Fouriertransformation

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\frac{1}{N}\sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}\ell k} \quad \text{für } \ell \in \{-s, \ldots, s\}$$

Anstatt negativer Werte für $\ell$ können wir für negatives $\ell$, $\ell$ durch $\ell' := N + \ell$ ersetzen.

**Diskrete Fouriertransformation (DFT)**

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\hat{x}_\ell = \frac{1}{N}\sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}\ell k} \quad \text{für } \ell \in \{0, \ldots, N-1\}$$

Wenn wir $N > 2s + 1$ gewählt haben berechnen wir eventuell mehr Werte als wir brauchen. Dies ist aber kein Problem.

# Diskrete Fouriertransformation

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\frac{1}{N}\sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}\ell k} \quad \text{für } \ell \in \{-s, \ldots, s\}$$

Anstatt negativer Werte für $\ell$ können wir für negatives $\ell$, $\ell$ durch $\ell' := N + \ell$ ersetzen.

**Diskrete Fouriertransformation (DFT)**

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\hat{x}_\ell = \frac{1}{N}\sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}\ell k} \quad \text{für } \ell \in \{0, \ldots, N-1\}$$

Wenn wir $N > 2s + 1$ gewählt haben berechnen wir eventuell mehr Werte als wir brauchen. Dies ist aber kein Problem.

# Diskrete Fouriertransformation

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\frac{1}{N}\sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}\ell k} \quad \text{für } \ell \in \{-s, \ldots, s\}$$

Anstatt negativer Werte für $\ell$ können wir für negatives $\ell$, $\ell$ durch $\ell' := N + \ell$ ersetzen.

**Diskrete Fouriertransformation (DFT)**

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\hat{x}_\ell = \frac{1}{N}\sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}\ell k} \quad \text{für } \ell \in \{0, \ldots, N-1\}$$

Wenn wir $N > 2s+1$ gewählt haben berechnen wir eventuell mehr Werte als wir brauchen. Dies ist aber kein Problem.

# $N$-te Einheitswurzeln

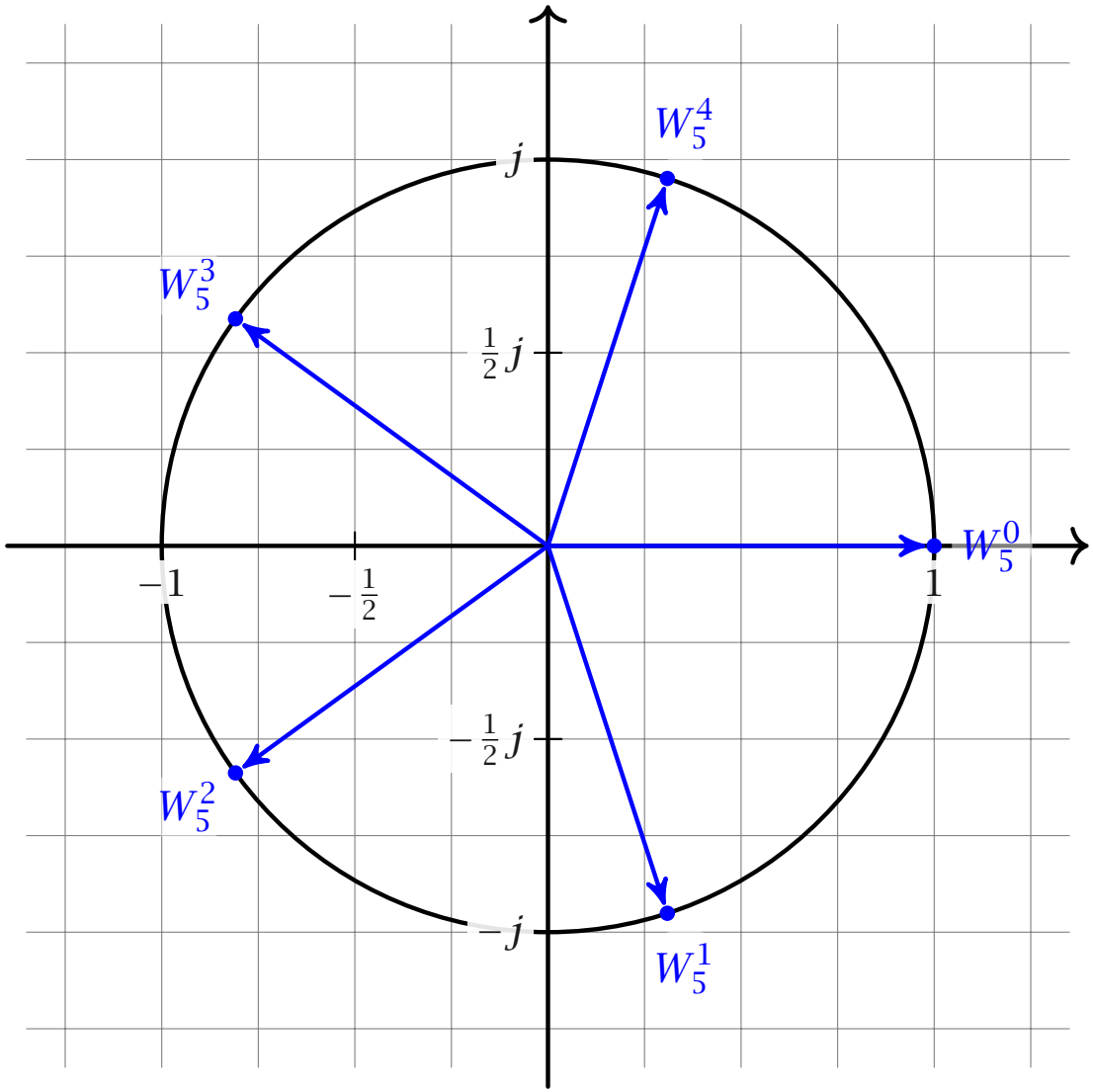
**Definition:** Einheitswurzel
$q \in \mathbb{C}$ ist $N$-te Einheitswurzel, gdw., $q^N = 1$.

**Definition:** primitive Einheitswurzel
Einheitswurzel $q \in \mathbb{C}$ ist primitiv falls sich jede Einheitswurzel $q'$ als $q' = q^i$ mit $i \in \mathbb{N}$ schreiben läßt.

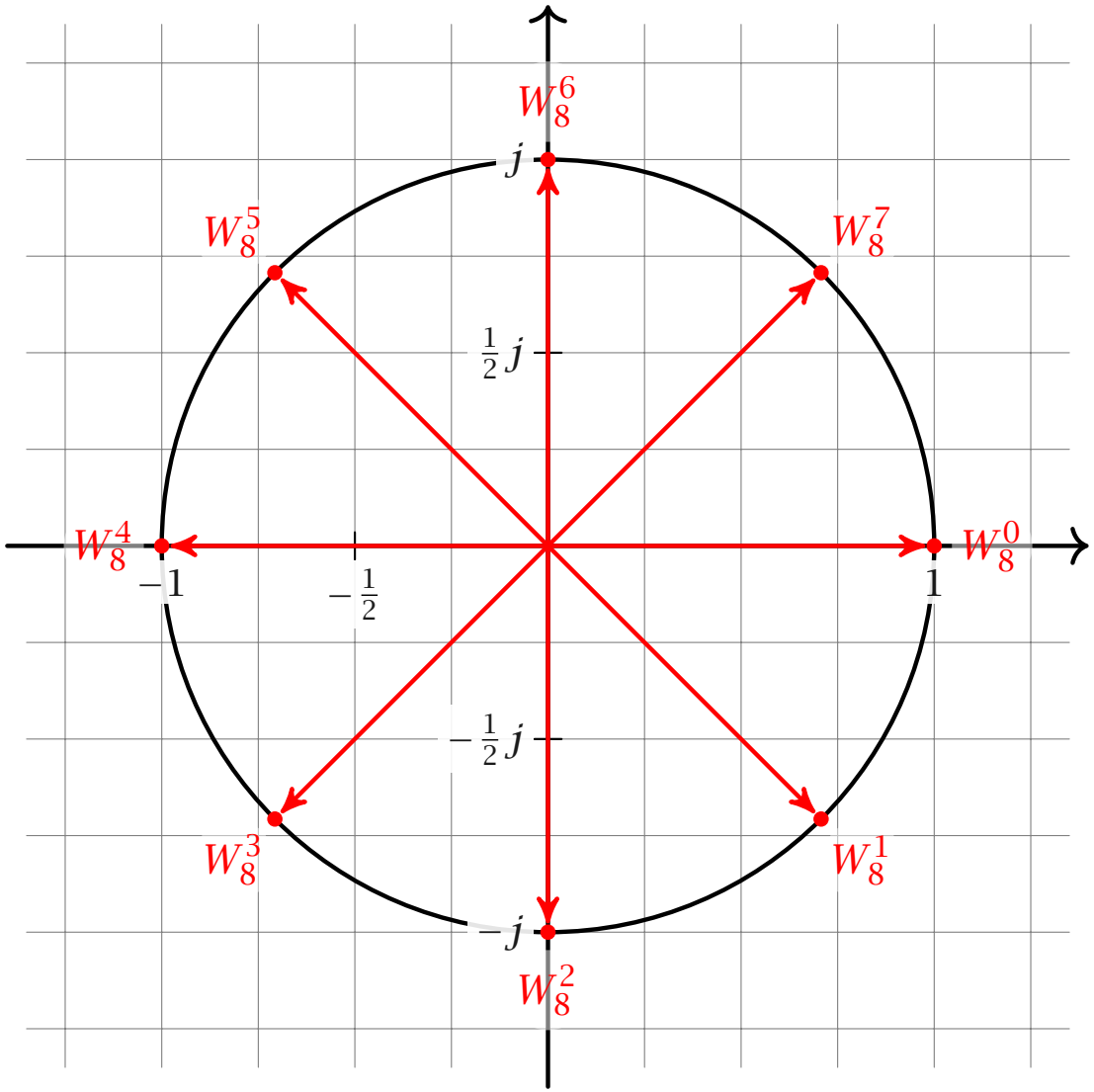
**Beispiel:**

$W_N := e^{-j\frac{2\pi}{N}}$ ist primitive $N$-te Einheitswurzel.

# $N$-te Einheitswurzeln



# $N$-te Einheitswurzeln



# Diskrete Fouriertransformation

**Diskrete Fouriertransformation (DFT)**

Gegeben $x_0, \ldots, x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\hat{x}_\ell = \frac{1}{N} \sum_{k=0}^{N-1} x_k W_N^{\ell k} \quad \text{für } \ell \in \{0, \ldots, N-1\}$$

Das heißt wir möchten im wesentlichen das durch Koeffizienten $x_0, \ldots, x_{N-1}$ definierte Polynom an den Stellen $W_N^\ell$ auswerten ($\ell \in \{0, \ldots, N-1\}$).

Ein Polynom vom Grad $N-1$ hat die Form $\sum_{i=0}^{N-1} c_i X^i$. Hier sind die Koeffizienten $c_i = x_i$ und die Unbestimmte (Variable) ist $X$. Wir möchten dieses Polynom nun an den Stellen $W_N^0, \ldots, W_N^{N-1}$ auswerten. Danach müssen wir noch all Ergebnisse durch $N$ dividieren.

# Polynomauswertung

Gegeben Koeffizienten $c_0, \ldots, c_{N-1}$, die ein Polynom

$$P[X] = \sum_{i=0}^{N-1} c_i X^i$$

definieren. Wir können $P[z]$ in Zeit $\mathcal{O}(N)$ berechnen.

**Horner-Schema:**

$$P[X] = c_0 + X(c_1 + X(c_2 + X(\cdots X(c_{N-2} + X(c_{N-1}))\cdots)))$$

```
1 evaluateHorner(C,z) // C is array of coefficients
2     res = C[N-1];
3     for i = N-2 to 0
4         res *= z;
5         res += C[i];
6     return res;
```

# Diskrete Fouriertransformation

Damit können wir eine DFT in Zeit $\mathcal{O}(N^2)$ berechnen.

Geht das schneller?

JA: Polynomauswertung an $N$-ten Einheitswurzeln kann sehr effizient gemacht werden.

Im folgenden beschreiben wir ein Verfahren, dass ein gegebenes Polynom $P$ vom Grad $N-1$ (gegeben durch Koeffizienten $c_0,\ldots,c_{N-1}$) an den $N$-ten Einheitswurzeln, d.h. an den Stellen $W_N^\ell$, $\ell \in \{0,\ldots,N-1\}$, auswertet, wobei $W_N$ eine beliebige primitive $N$-te Einheitswurzel ist. Das heißt im folgenden gehen wir nicht davon aus, dass $W_N = e^{-j2\pi/N}$ ist. Um eine DFT zu erhalten muß man das folgende Verfahren zur Auswertung von Polynomen mit $W_N = e^{-j2\pi/N}$ anwenden und danach mit $1/N$ multiplizieren.

# $N$-te Einheitswurzeln

**Halbierungslemma**

Sei $N \in \mathbb{N}$ gerade. Dann gilt $W_N^2$ ist $N/2$-te primitive Einheitswurzel.

**Beweis**

# $N$-te Einheitswurzeln

**Halbierungslemma**

Sei $N \in \mathbb{N}$ gerade. Dann gilt $W_N^2$ ist $N/2$-te primitive Einheitswurzel.

**Beweis**

- $(W_N^2)^{N/2} = W_N^N = 1$, d.h. $W_N^2$ ist $N/2$-te Einheitswurzel.
- Wir wollen zeigen, dass $W_N^2$ primitive $N/2$-te Einheitswurzel ist. Dafür zeigen wir, dass $(W_N^2)^\ell$ für $\ell \in \{0, \ldots, N/2-1\}$ insgesamt $N/2$ verschiedene Werte ergibt.

# $N$-te Einheitswurzeln

**Halbierungslemma**

Sei $N \in \mathbb{N}$ gerade. Dann gilt $W_N^2$ ist $N/2$-te primitive Einheitswurzel.

**Beweis**

- $(W_N^2)^{N/2} = W_N^N = 1$, d.h. $W_N^2$ ist $N/2$-te Einheitswurzel.
- Wir wollen zeigen, dass $W_N^2$ primitive $N/2$-te Einheitswurzel ist. Dafür zeigen wir, dass $(W_N^2)^\ell$ für $\ell \in \{0, \ldots, N/2-1\}$ insgesamt $N/2$ verschiedene Werte ergibt.

  Angenommen es existiert $\ell_1, \ell_2 \in \{0, \ldots, N/2-1\}$ mit $(W_N^2)^{\ell_1} = (W_N^2)^{\ell_2}$ aber $\ell_1 \neq \ell_2$ (o.B.d.A. $\ell_1 < \ell_2$).

  Dann folgt $W_N^{2(\ell_2-\ell_1)} = 1$. Da $0 < 2(\ell_2 - \ell_1) \leq N-1$ kann $W_N$ damit keine primitive $N$-te Einheitswurzel sein ($\lightning$).

# $N$-te Einheitswurzeln

**Halbierungslemma**

Sei $N \in \mathbb{N}$ gerade. Dann gilt $W_N^2$ ist $N/2$-te primitive Einheitswurzel.

**Beweis**

- $(W_N^2)^{N/2} = W_N^N = 1$, d.h. $W_N^2$ ist $N/2$-te Einheitswurzel.
- Wir wollen zeigen, dass $W_N^2$ primitive $N/2$-te Einheitswurzel ist. Dafür zeigen wir, dass $(W_N^2)^\ell$ für $\ell \in \{0, \ldots, N/2-1\}$ insgesamt $N/2$ verschiedene Werte ergibt.

  Angenommen es existiert $\ell_1, \ell_2 \in \{0, \ldots, N/2-1\}$ mit $(W_N^2)^{\ell_1} = (W_N^2)^{\ell_2}$ aber $\ell_1 \neq \ell_2$ (o.B.d.A. $\ell_1 < \ell_2$).

  Dann folgt $W_N^{2(\ell_2-\ell_1)} = 1$. Da $0 < 2(\ell_2 - \ell_1) \leq N-1$ kann $W_N$ damit keine primitive $N$-te Einheitswurzel sein ($\lightning$).

# $N$-te Einheitswurzeln

**Halbierungslemma**

Sei $N \in \mathbb{N}$ gerade. Dann gilt $W_N^2$ ist $N/2$-te primitive Einheitswurzel.

**Beweis**

- $(W_N^2)^{N/2} = W_N^N = 1$, d.h. $W_N^2$ ist $N/2$-te Einheitswurzel.
- Wir wollen zeigen, dass $W_N^2$ primitive $N/2$-te Einheitswurzel ist. Dafür zeigen wir, dass $(W_N^2)^\ell$ für $\ell \in \{0, \ldots, N/2-1\}$ insgesamt $N/2$ verschiedene Werte ergibt.

  Angenommen es existiert $\ell_1, \ell_2 \in \{0, \ldots, N/2-1\}$ mit $(W_N^2)^{\ell_1} = (W_N^2)^{\ell_2}$ aber $\ell_1 \neq \ell_2$ (o.B.d.A. $\ell_1 < \ell_2$).

  Dann folgt $W_N^{2(\ell_2-\ell_1)} = 1$. Da $0 < 2(\ell_2 - \ell_1) \leq N-1$ kann $W_N$ damit keine primitive $N$-te Einheitswurzel sein (↯).

# Fast Fourier Transform (FFT)

**Aufgabe:**

Werte Polynom $P$ an Stellen $W_N^0, \ldots, W_N^{N-1}$ aus.

**Annahme: $N$ ist 2er-Potenz**

definiere:

$$P_{\text{odd}} = c_1 + c_3 X + c_5 X^2 + \ldots + c_{N-1} X^{\frac{N-2}{2}}$$

$$P_{\text{even}} = c_0 + c_2 X + c_4 X^2 + \ldots + c_{N-2} X^{\frac{N-2}{2}}$$

dann gilt:

$$P[a] = P_{\text{even}}[a^2] + a P_{\text{odd}}[a^2]$$

Wir müssen z.B. $P_{\text{odd}}$ für alle $a^2$ mit $a \in \{W_N^0, \ldots, W_N^{N-1}\}$ berechnen, d.h. für Menge $\{(W_N^0)^2, \ldots, (W_N^{N-1})^2\}$, das ist aber die Menge $\{W_{N/2}^0, \ldots, W_{N/2}^{N/2-1}\}$ (Halbierungslemma).

# Fast Fourier Transform (FFT)

**Aufgabe:**

Werte Polynom $P$ an Stellen $W_N^0, \ldots, W_N^{N-1}$ aus.

**Annahme: $N$ ist 2er-Potenz**

definiere:

$$P_{\text{odd}} = c_1 + c_3 X + c_5 X^2 + \ldots + c_{N-1} X^{\frac{N-2}{2}}$$

$$P_{\text{even}} = c_0 + c_2 X + c_4 X^2 + \ldots + c_{N-2} X^{\frac{N-2}{2}}$$

dann gilt:

$$P[a] = P_{\text{even}}[a^2] + a P_{\text{odd}}[a^2]$$

Wir müssen z.B. $P_{\text{odd}}$ für alle $a^2$ mit $a \in \{W_N^0, \ldots, W_N^{N-1}\}$ berechnen, d.h. für Menge $\{(W_N^0)^2, \ldots, (W_N^{N-1})^2\}$, das ist aber die Menge $\{W_{N/2}^0, \ldots, W_{N/2}^{N/2-1}\}$ (Halbierungslemma).

# Fast Fourier Transform (FFT)

**Aufgabe:**

Werte Polynom $P$ an Stellen $W_N^0, \ldots, W_N^{N-1}$ aus.

**Annahme: $N$ ist 2er-Potenz**

definiere:

$$P_{\text{odd}} = c_1 + c_3 X + c_5 X^2 + \ldots + c_{N-1} X^{\frac{N-2}{2}}$$

$$P_{\text{even}} = c_0 + c_2 X + c_4 X^2 + \ldots + c_{N-2} X^{\frac{N-2}{2}}$$

dann gilt:

$$P[a] = P_{\text{even}}[a^2] + a P_{\text{odd}}[a^2]$$

Wir müssen z.B. $P_{\text{odd}}$ für alle $a^2$ mit $a \in \{W_N^0, \ldots, W_N^{N-1}\}$ berechnen, d.h. für Menge $\{(W_N^0)^2, \ldots, (W_N^{N-1})^2\}$, das ist aber die Menge $\{W_{N/2}^0, \ldots, W_{N/2}^{N/2-1}\}$ (Halbierungslemma).

# Fast Fourier Transform (FFT)

**Aufgabe:**
Werte Polynom $P$ an Stellen $W_N^0, \ldots, W_N^{N-1}$ aus.
**Annahme: $N$ ist 2er-Potenz**

definiere:

$$P_{\text{odd}} = c_1 + c_3 X + c_5 X^2 + \ldots + c_{N-1} X^{\frac{N-2}{2}}$$

$$P_{\text{even}} = c_0 + c_2 X + c_4 X^2 + \ldots + c_{N-2} X^{\frac{N-2}{2}}$$

dann gilt:

$$P[a] = P_{\text{even}}[a^2] + a P_{\text{odd}}[a^2]$$

Wir müssen z.B. $P_{\text{odd}}$ für alle $a^2$ mit $a \in \{W_N^0, \ldots, W_N^{N-1}\}$ berechnen, d.h. für Menge $\{(W_N^0)^2, \ldots, (W_N^{N-1})^2\}$, das ist aber die Menge $\{W_{N/2}^0, \ldots, W_{N/2}^{N/2-1}\}$ (Halbierungslemma).

# Implementierung

```
1  Input:  Polynom P of degree N-1, by coefficient array C
2  Output: R[ℓ] contains P[W_N^ℓ]
3
4  FFT(C,N,W_N) // C is array of coefficients
5      if (N == 1) return C;
6      C_odd  = [C[1],C[3],...,C[N-1]];
7      C_even = [C[0],C[2],...,C[N-2]];
8      R_odd  = FFT(C_odd ,N/2,W_N^2);
9      R_even = FFT(C_even,N/2,W_N^2);
10     W = 1; // W = W_N^0
11     for ℓ = 0 to N/2-1
12         R[ℓ]       = R_even[ℓ] + W * R_odd [ℓ];
13         R[ℓ + N/2] = R_even[ℓ] - W * R_odd [ℓ];
14         W *= W_N; // W = W_N^ℓ
15     return R;
```

Um die DFT zu erhalten müssen wir $W_N = e^{-j2\pi/N}$ setzen, und das Ergebnis durch $N$ dividieren.

# Korrektheit

Für $N = 1$ besteht das Polynom nur aus Koeffizient $c_0$, d.h., bei der Auswertung ist nichts zu tun.

Nach Zeile 5/6 gilt

$$\mathsf{R}_{\mathsf{odd}}[\ell] = P_{\mathsf{odd}}[W_{N/2}^{\ell}] = P_{\mathsf{odd}}[W_N^{2\ell}] \quad \text{für } \ell = 0, \ldots, N/2 - 1$$

$$\mathsf{R}_{\mathsf{even}}[\ell] = P_{\mathsf{even}}[W_{N/2}^{\ell}] = P_{\mathsf{even}}[W_N^{2\ell}] \quad \text{für } \ell = 0, \ldots, N/2 - 1$$

Nach Zeile 9 gilt

$$\mathsf{R}[\ell] = P_{\mathsf{even}}[W_N^{2\ell}] + W_N^{\ell} \cdot P_{\mathsf{odd}}[W_N^{2\ell}] = P[W_N^{\ell}]$$

Nach Zeile 10 gilt

$$\begin{aligned}
\mathsf{R}[\ell + N/2] &= P_{\mathsf{even}}[W_N^{2\ell}] - W_N^{\ell} \cdot P_{\mathsf{odd}}[W_N^{2\ell}] \\
&= P_{\mathsf{even}}[(W_N^{\ell+N/2})^2] + W_N^{\ell+N/2} \cdot P_{\mathsf{odd}}[(W_N^{\ell+N/2})^2] \\
&= P[W_N^{\ell+N/2}]
\end{aligned}$$

Beachte, dass $W_N^{N/2} = -1$.

# Fast Fourier Transform (FFT)

Laufzeit:

- Rekurrenzgleichung: $T(n) = 2T(n/2) + \mathcal{O}(n)$
- Mastertheorem ergibt: $T(n) = \mathcal{O}(n \log n)$

# Diskrete Fouriertransformation

Hier vewenden wir wieder $W_N = e^{-j2\pi/N}$, d.h., $W_N$ ist keine beliebige Einheitswurzel.

Eine DFT berechnet das folgende Matrix-Vektor Produkt.

$$\frac{1}{N}\underbrace{\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N & W_N^2 & \cdots & W_N^{N-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-2)} & \cdots & W_N^{(N-1)^2} \end{pmatrix}}_{V_N} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} \hat{x}_0 \\ \hat{x}_1 \\ \vdots \\ \hat{x}_{N-1} \end{pmatrix}$$

$V_N$ heißt Vandermonde matrix. Die Inverse Matrix $V_N^{-1}$ ist gegeben durch

$$V_N^{-1} = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N^{-1} & W_N^{-2} & \cdots & W_N^{-(N-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & W_N^{-(N-1)} & W_N^{-2(N-2)} & \cdots & W_N^{-(N-1)^2} \end{pmatrix}$$

# Diskrete Fouriertransformation

**Beweis:**

Das Element in Zeile $i$ und Spalte $j$ in $V_N \cdot V_N^{-1}$ ist gegeben durch

$$\frac{1}{N}\sum_{k=0}^{N-1}(W_N^i)^k \cdot (W_N^{-j})^k = \frac{1}{N}\sum_{k=0}^{N-1}(W_N^{i-j})^k = \begin{cases} 1 & i = j \mod N \\ 0 & i \neq j \mod N \end{cases}$$

# Polynominterpolation

Angenommen wir haben ein Polynom an Stellen $W_N^0, \ldots, W_N^{N-1}$ ausgewertet. Mit Hilfe von $V_N^{-1}$ können wir die Koeffizienten des Polynoms herausfinden:

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N^{-1} & W_N^{-2} & \cdots & W_N^{-(N-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & W_N^{-(N-1)} & W_N^{-2(N-2)} & \cdots & W_N^{-(N-1)^2} \end{pmatrix}}_{N \cdot V_N^{-1}} \begin{pmatrix} P[W_N^0] \\ P[W_N^1] \\ \vdots \\ P[W_N^{N-1}] \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix}$$

Um das Matrix-Vektor-Produkt auszurechen müssen wir den FFT-Algorithmus mit $-W_N$ benutzen; wobei $P[W_N^0], \ldots, P[W_N^{N-1}]$ die Koeffizienten des auszuwertenden Polynoms sind.

# Polynome

Man kann Polynome als Funktionen auffassen; z.B. $f : \mathbb{R} \to \mathbb{R}, f(x) \mapsto P[x]$. Allgemeiner ist es aber Polynome als algebraische Objekte aufzufassen. Auf diesen Objekten gibt es Operationen wie z.B. Auswertung, Multiplikation, Addition, oder Division.

Ein formaler Ausdruck der Form

$$P[X] = c_0 + c_1 X + c_2 X^2 + \cdots + c_{n-1} X^{n-1}$$

mit $c_i \in \mathbb{R}$ und $c_{n-1} \neq 0$ heißt Polynom vom Grad $n-1$. Die $c_i$ heißen Koeffizienten des Polynoms.

- $\mathrm{grad}(P)$ bezeichnet den Grad des Polynoms;

# Operationen auf Polynomen

Seien $A[X], B[X]$ Polynome

- Auswertung: berechne für $x_0 \in \mathbb{R}$

$$A[x_0]$$

- Addition: berechne Polynom $C[X]$, so daß für alle $x \in \mathbb{R}$

$$C[x] = A[x] \cdot B[x]$$

- Multiplikation: berechne Polynom $C[X]$, so daß für alle $x \in \mathbb{R}$

$$C[x] = A[x] + B[x]$$

# Repräsentation von Polynomen

**Koeffizientdarstellung**

Wir repräsentieren das Polynom durch die Koeffizienten

$c_0, \ldots, c_{n-1}$.

# Repräsentation von Polynomen

**Theorem:**
Ein Polynom vom Grad $n-1$ ist durch Funktionswerte an $n$ paarweise verschiedenen Stellen eindeutig definiert.

**Stützstellendarstellung**
Wir wählen $n$ paarweise verschiedene Stützstellen $x_0, \ldots, x_{n-1}$.

Wir repräsentieren das Polynom $P$ durch die Funktionswerte $P[x_0], \ldots, P[x_{n-1}]$.

Die Stützstellendarstellung ist nicht eindeutig. Jede Menge von verschiedenen Punkten $x_0, \ldots, x_{n-1}$ kann verwendet werden.

# Auswertung von Polynomen

**Koeffizientendarstellung**

Mit dem Hornerschema können wir $P[z]$ in Zeit $\mathcal{O}(n)$ berechnen.

**Stützstellendarstellung bzgl. $x_0, \ldots, x_{n-1}$**

nicht so einfach...

# Auswertung von Polynomen

**Koeffizientendarstellung**

Mit dem Hornerschema können wir $P[z]$ in Zeit $\mathcal{O}(n)$ berechnen.

**Stützstellendarstellung bzgl. $x_0, \ldots, x_{n-1}$**

nicht so einfach...

# Addition von Polynomen

**Koeffizientendarstellung**

Seien $a_0, \ldots, a_{n-1}$ Koeffizienten von $A[X]$ und $b_0, \ldots, b_{n-1}$ Koeffizienten von $B[X]$.

Koeffizienten von $C[X] = A[X] + B[X]$ sind gegeben durch $c_i = a_i + b_i$, d.h. wir benötigen $n$ Additionen.

**Stützstellendarstellung bzgl. $x_0, \ldots, x_{n-1}$**

Sei $A[x_i] = a_i$ und $B[x_i] = b_i$. $C[x_i] = a_i + b_i$ (direkt aus der Definition). D.h. wir benötigen $n$ Additionen.

# Addition von Polynomen

**Koeffizientendarstellung**

Seien $a_0, \ldots, a_{n-1}$ Koeffizienten von $A[X]$ und $b_0, \ldots, b_{n-1}$ Koeffizienten von $B[X]$.

Koeffizienten von $C[X] = A[X] + B[X]$ sind gegeben durch $c_i = a_i + b_i$, d.h. wir benötigen $n$ Additionen.

**Stützstellendarstellung bzgl. $x_0, \ldots, x_{n-1}$**

Sei $A[x_i] = a_i$ und $B[x_i] = b_i$. $C[x_i] = a_i + b_i$ (direkt aus der Definition). D.h. wir benötigen $n$ Additionen.

# Multiplikation von Polynomen

**Stützstellendarstellung bzgl. $x_0, \ldots, x_{n-1}$**

Sei $A[x_i] = a_i$ und $B[x_i] = b_i$. $C[x_i] = a_i \cdot b_i$ (direkt aus der Definition). D.h. wir benötigen $n$ Multiplikationen.

**Koeffizientendarstellung**

Seien $a_0, \ldots, a_{n-1}$ Koeffizienten von $A[X]$ und $b_0, \ldots, b_{n-1}$ Koeffizienten von $B[X]$.

Koeffizienten von $C[X] = A[X] \cdot B[X]$ sind gegeben durch

$$c_i = \sum_{k=0}^{i} a_k b_{i-k}$$

Für Element $c_i$ benötigt man $i+1$ Multiplikationen und $i$ Additionen. Insgesamt Aufwand $\mathcal{O}(n^2)$.

# Multiplikation von Polynomen

**Stützstellendarstellung bzgl. $x_0, \ldots, x_{n-1}$**

Sei $A[x_i] = a_i$ und $B[x_i] = b_i$. $C[x_i] = a_i \cdot b_i$ (direkt aus der Definition). D.h. wir benötigen $n$ Multiplikationen.

**Koeffizientendarstellung**

Seien $a_0, \ldots, a_{n-1}$ Koeffizienten von $A[X]$ und $b_0, \ldots, b_{n-1}$ Koeffizienten von $B[X]$.

Koeffizienten von $C[X] = A[X] \cdot B[X]$ sind gegeben durch

$$c_i = \sum_{k=0}^{i} a_k b_{i-k}$$

Für Element $c_i$ benötigt man $i+1$ Multiplikationen und $i$ Additionen. Insgesamt Aufwand $\mathcal{O}(n^2)$.

# Interpolation

Transformation eines Polynoms von Stützstellendarstellung in Koeffizientendarstellung heißt Polynominterpolation.

# Interpolation

Stützstellen $x_0, \ldots, x_{n-1}$ beliebig. Sei $y_k = A[x_k]$.

**Lagrange-Interpolationsformel:**

$$A[X] = \sum_{k=0}^{n-1} y_k \frac{\Pi_{j \neq k}(X - x_j)}{\Pi_{j \neq k}(x_k - x_j)}$$

**Aufwand:** $\mathcal{O}(n^2)$ durch geschickte Reihenfolge der Operationen.

Es ist klar, dass die Formel ein Polynom vom Grad höchstens $n-1$ liefert. Außerdem sieht man leicht, dass $A[x_k] = y_k$ ist. Also ergibt die Formel das gesuchte Polynom.

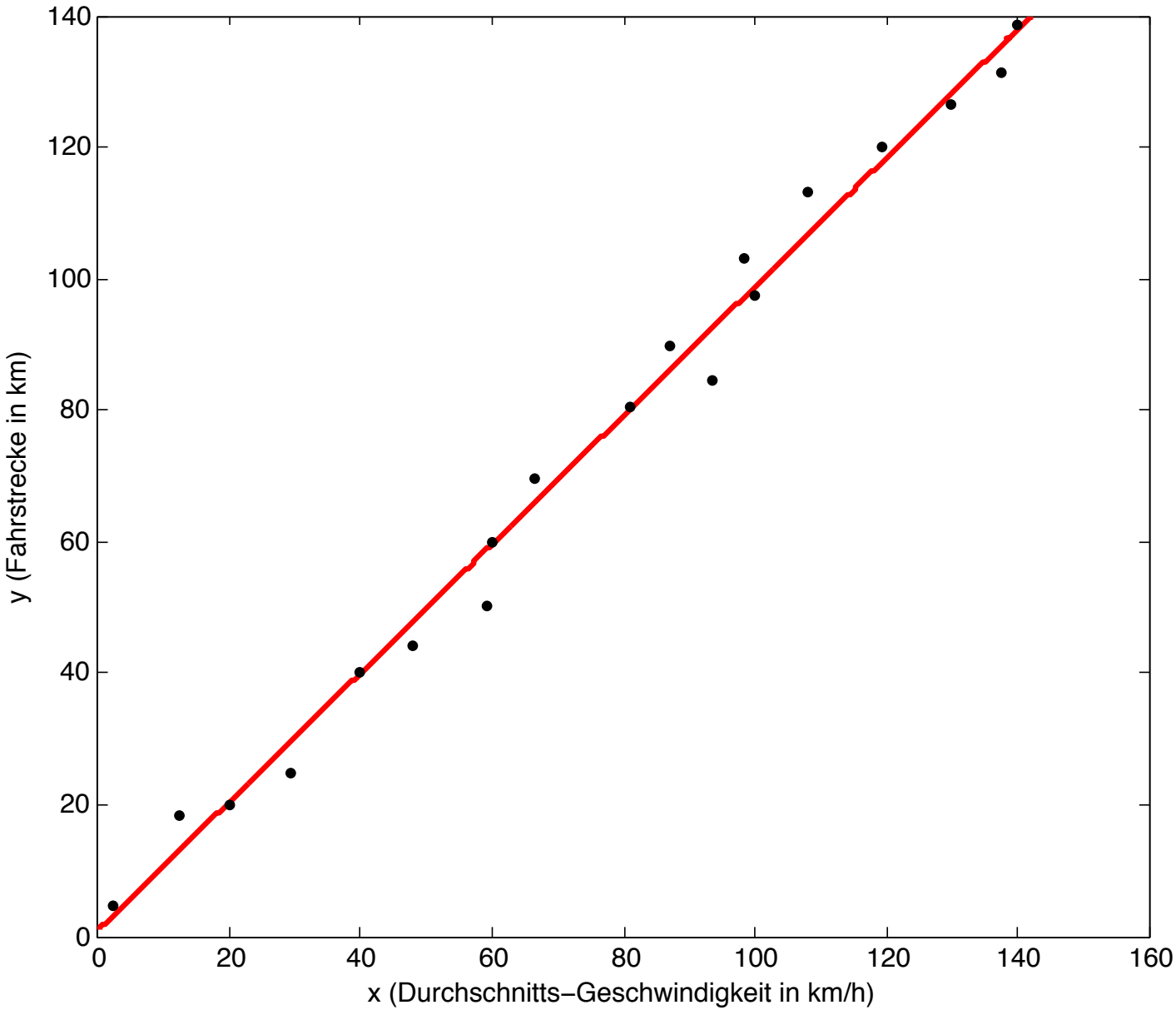
**Beobachtung**

Wenn wir komplexe Einheitswurzeln als Stützstellen verwenden können wir in Zeit $\mathcal{O}(n \log n)$ zwischen Koeffizientendarstellung und Stützstellendarstellung wechseln!!!!!!!!!!

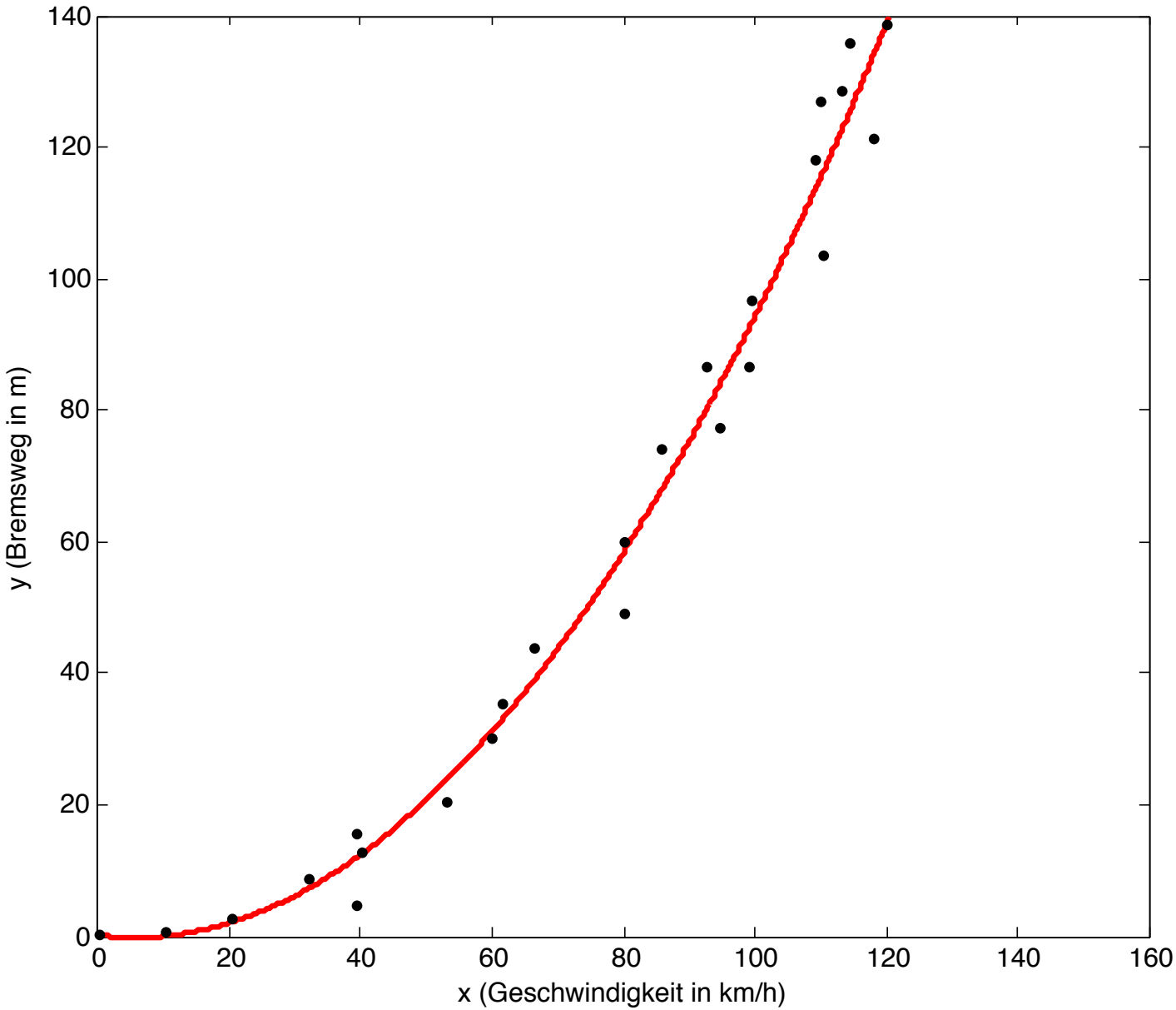
Wir können Polynome in Koeffizientendarstellung in Zeit $\mathcal{O}(n \log n)$ multiplizieren.

- berechne Stützstellendarstellung bzgl. Einheitswurzeln (Zeit $\mathcal{O}(n \log n)$ via FFT)
- berechne Produkt in Stützstellendarstellung: Zeit $(n)$
- berechne Koeffizientendarstellung aus Stützstellendarstellung (Zeit $\mathcal{O}(n \log n)$ via inverser FFT)

# Beispiel-Problem: Geschwindigkeit vs. Fahrstrecke



# Beispiel-Problem: Geschwindigkeit vs. Bremsweg



# Problemstellung Least Squares

- **Gegeben:** Datenreihe mit $m$ Datenpunkten

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$$

mit $x_j, y_j \in \mathbb{R}$ für $j = 1, \ldots, m$

- **Erwartung:** $y_j$ enthalten Messfehler ($j = 1, \ldots, m$)
- **Gesucht:** Funktion $F : \mathbb{R} \to \mathbb{R}$ so dass Approximationsfehler

$$\eta_j = F(x_j) - y_j$$

für alle $j = 1, \ldots, m$ möglichst gering

- **Annahme:** $F$ lässt sich darstellen als Summe von Basisfunktionen $f_i$,

$$F(x) = \sum_{i=1}^{n} c_i f_i(x)$$

# Problemstellung Least Squares

- **Gegeben:** Datenreihe mit $m$ Datenpunkten

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$$

mit $x_j, y_j \in \mathbb{R}$ für $j = 1, \ldots, m$

- **Erwartung:** $y_j$ enthalten Messfehler ($j = 1, \ldots, m$)
- **Gesucht:** Funktion $F : \mathbb{R} \to \mathbb{R}$ so dass Approximationsfehler

$$\eta_j = F(x_j) - y_j$$

für alle $j = 1, \ldots, m$ möglichst gering

- **Annahme:** $F$ lässt sich darstellen als Summe von Basisfunktionen $f_i$,

$$F(x) = \sum_{i=1}^{n} c_i f_i(x)$$

# Wahl der Basisfunktionen

**Wahl der Basisfunktionen $f_i$ für $F$:**

- typische Wahl: $f_i(x) = x^{i-1}$, dann

$$F(x) = c_1 + c_2 x + c_3 x^2 + \ldots + c_n x^{n-1}$$

(Polynom in $x$ vom Grad $n-1$)

- auch oft mit $n = 2$, dann

$$F(x) = c_1 + c_2 x$$

(Gerade) auch genannt lineare Regression

- im Fall von Tomographie: Pixel- oder Voxel-Basisfunktionen

# Matrix-Notation

- für die verschiedenen Datenpunkte $x_j$, $j = 1, \ldots, m$, kann $F$ geschrieben werden als:

$$\begin{pmatrix} F(x_1) \\ \vdots \\ F(x_m) \end{pmatrix} = \underbrace{\begin{pmatrix} f_1(x_1) & \cdots & f_n(x_1) \\ \vdots & & \vdots \\ f_1(x_m) & \cdots & f_n(x_m) \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}}_{=:c}$$

  mit $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$.

- untersucht wird dann der Approximationsfehler $\eta$

$$\eta = Ac - y$$

  mit $\eta = (\eta_1, \ldots, \eta_m) \in \mathbb{R}^m$, $y = (y_1, \ldots, y_m) \in \mathbb{R}^m$.

# Daten mit Meßfehler

ist $m = n$ und $A$ invertierbar, so kann direkt

$$Ac - y = 0$$

gelöst werden.

**Problem:**

- selbst wenn $A$ invertierbar ist, $y$ enthält Meßfehler
- Lösung $F$ ist dann meist nicht die **gewünschte** Lösung
- passt sich zu sehr an "Ausreißer" an

**besser:** mehr Datenpunkte, $m >> n$

# Minimierung des Approximationsfehlers

- zur Minimierung des Approximationsfehlers kann z.B. die Norm $\|\eta\|$ betrachtet werden

$$\|\eta\| = \left(\sum_{j=1}^{m} \eta_j^2\right)^{\frac{1}{2}} = \sqrt{\sum_{j=1}^{m} \eta_j^2}$$

- zur Vereinfachung betrachtet man üblicherweise

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{j=1}^{m} \left(\sum_{i=1}^{n} a_{ji} c_i - y_j\right)^2$$

- daher der Name: Methode der kleinsten Quadrate oder Least squares

# Least-squares Lösung

**Least-squares Lösung**

Sei $A \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$ mit $m \geq n$. Eine Lösung $c \in \mathbb{R}^n$ des Minimierungs-Problems

$$\min_c \|Ac - y\| \qquad \text{or} \qquad \min_c \|Ac - y\|^2$$

heißt Least-squares Lösung.

**Anwendungs-Beispiele:**

- Tracking von Objekten mit Kameras
- Kalibrierung von Kameras, Robotern, etc.
- Iterative Rekonstruktion für Tomographie

# Normalengleichung

- Berechnung der Least-squares Lösung mit Standard-Technik "Ableitung gleich null setzen"
- hier: partielle Ableitungen für $k = 1, \ldots, n$

$$\frac{\partial \|\eta\|^2}{\partial c_k} = \sum_{j=1}^{m} 2 \left( \sum_{i=1}^{n} a_{ji} c_i - y_j \right) a_{jk} = 0$$

- daraus folgt

$$\nabla \|\eta\|^2 = A^T (Ac - y) = 0$$

- umgeformt ergibt sich

$$A^T A c = A^T y$$

auch genannt die Normalengleichung.

# Normalengleichung

**Normalengleichung**

Sei $A \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$ mit $m \geq n$. $c \in \mathbb{R}^n$ ist eine Least-squares Lösung von $\min_c \|Ac - y\|$ genau dann, wenn die Normalengleichung gilt:

$$A^T A c = A^T y$$

Insbesondere ist die Normalengleichung lösbar, falls $\text{rank}(A) = n$.

# Normalengleichung

**Normalengleichung**

Sei $A \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$ mit $m \geq n$. $c \in \mathbb{R}^n$ ist eine Least-squares Lösung von $\min_c \|Ac - y\|$ genau dann, wenn die Normalengleichung gilt:

$$A^T A c = A^T y$$

Insbesondere ist die Normalengleichung lösbar, falls $\text{rank}(A) = n$.

- Die Matrix $A^T A$ ist immer symmetrisch.
- Falls $\text{rank}(A) = n$ ist $A^T A$ auch positiv definit und damit invertierbar
- Die Lösung der Normalengleichung ist dann

$$c = ((A^T A)^{-1} A^T) y.$$

# Pseudoinverse

**Pseudoinverse**

Sei $A \in \mathbb{R}^{m \times n}$ mit $m \geq n$ und $\text{rank}(A) = n$. Die Matrix

$$A^+ := (A^T A)^{-1} A^T$$

heißt Pseudoinverse von $A$ (auch Moore-Penrose Inverse genannt).

- Die Pseudoinverse verallgemeinert das Konzept der Inversen für nicht-quadratische Matrizen.
- Ist $A$ invertierbar, dann gilt $A^+ = A^{-1}$.

# Pseudoinverse

Für $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ löse

$$A^T A c = A^T y$$

Umbennung der Variablen:

$$Mx = b$$

- $x = c$
- $M = A^T A$; Berechnung in Zeit $\mathcal{O}(n^2 m)$; (trivialer Algorithmus)
- $b = A^T y$; Berechnung in Zeit $\mathcal{O}(mn)$; (trivialer Algorithmus)

# Lösen von linearen Systemen

**Inverse Matrix**

Sei $M \in \mathbb{R}^{n \times n}$ quadratische Matrix. Falls ein $M' \in \mathbb{R}^{n \times n}$ existiert mit

$$MM' = M'M = I_n$$

(wobei $I_n$ die $n \times n$ Einheitsmatrix ist), dann heißt $M$ invertierbar und wir nennen $M' =: M^{-1}$ das Inverse von $M$.

- Ist $Mx = b$ lineares System mit $M \in \mathbb{R}^{n \times n}$ invertierbar, dann ist

  $$x = M^{-1}b$$

  die Lösung des linearen Systems. Diese Lösung ist eindeutig.

# Invertierbarkeit von Matrizen

**Problem 1:** wann ist eine Matrix $M \in \mathbb{R}^{n \times n}$ invertierbar?

- $Mx = 0$ hat nur die triviale Lösung $x = 0$

# Invertierbarkeit von Matrizen

**Problem 1:** wann ist eine Matrix $M \in \mathbb{R}^{n \times n}$ invertierbar?

- $Mx = 0$ hat nur die triviale Lösung $x = 0$

**Problem 2:** falls $M$ invertierbar, wie findet man das Inverse $M^{-1}$?

- Berechnung mit Gauss-Jordan Elimination
- Umweg über Zerlegungen von $M$ (z.B. LU-Zerlegung)
- **generell:** Berechnung der Inversen meist numerisch nicht stabil

# Günstige Matrix-Formen

- $M$ diagonal: Lösung kann abgelesen werden

$$\begin{pmatrix} M_{11} & & 0 \\ & \ddots & \\ 0 & & M_{nn} \end{pmatrix} x = b$$

- $M$ obere Dreiecksmatrix: Rückwärtssubstitution

$$\begin{pmatrix} M_{11} & \cdots & M_{1n} \\ & \ddots & \vdots \\ 0 & & M_{nn} \end{pmatrix} x = b$$

- $M$ untere Dreiecksmatrix: Vorwärtssubstitution

$$\begin{pmatrix} M_{11} & & 0 \\ \vdots & \ddots & \\ M_{n1} & \cdots & M_{nn} \end{pmatrix} x = b$$

# Rückwärtssubstitution

Lineares System in oberer Dreiecksform:

$$
\begin{aligned}
M_{11}x_1 + M_{21}x_2 + \quad \ldots + M_{1,n-1}x_{n-1} \quad + \quad M_{1n}x_n &= b_1 \\
M_{22}x_2 + \quad \ldots + M_{2,n-1}x_{n-1} \quad + \quad M_{2n}x_n &= b_2 \\
&\vdots \\
M_{n-1,n-1}x_{n-1} + \quad M_{n-1,n}x_n &= b_{n-1} \\
M_{n,n}x_n &= b_n
\end{aligned}
$$

Rückwärts nacheinander nach $x_n, x_{n-1}, \ldots, x_1$ auflösen:

$$
x_i = \left( b_i - \sum_{j=i+1}^{n} M_{ij}x_j \right) / M_{ii}
$$

**Aufwand:** $\mathcal{O}(n^2)$ arithmetische Operationen (FLOPs)

# Vorwärtssubstitution

Lineares System in unterer Dreiecksform:

$$
\begin{aligned}
M_{11}x_1 &= b_1 \\
M_{21}x_1 + M_{22}x_2 &= b_2 \\
&\vdots \\
M_{n-1,1}x_1 + M_{n-1,2}x_2 + \ldots + M_{n-1,n-1}x_{n-1} &= b_{n-1} \\
M_{n1}x_1 + M_{n2}x_2 + \ldots + M_{n,n-1}x_{n-1} + M_{nn}x_n &= b_n
\end{aligned}
$$

Vorwärts nacheinander nach $x_1, x_2, \ldots, x_n$ auflösen:

$$x_i = \left( b_i - \sum_{j=1}^{i-1} M_{ij}x_j \right) / M_{ii}$$

**Aufwand:** $\mathcal{O}(n^2)$ FLOPs

# Gauss Elimination

**Gauss-Elimination:** Überführung der Matrix $A$ in Dreiecksform durch elementare Zeilenumformungen

- Komplexität: $\Theta(n^3)$

**Gauss-Jordan-Elimination:** Überführung der Matrix $A$ in Diagonalform durch elementare Zeilenumformungen

- Komplexität: $\Theta(n^3)$

**Elementare Zeilenumformungen:**

- Vielfaches einer Zeile zu einer anderen Zeile addieren
- zwei Zeilen vertauschen
- Vielfaches einer Zeile berechnen

für lineare Systeme: Gauss-Elimination auf erweiterter Matrix

$$(A|b) = \left(\begin{array}{ccc|c} M_{11} & \cdots & M_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ M_{n1} & \cdots & M_{nn} & b_n \end{array}\right)$$

# Gauss Elimination

**Gauss-Elimination:** Überführung der Matrix $A$ in Dreiecksform durch elementare Zeilenumformungen

- Komplexität: $\Theta(n^3)$

**Gauss-Jordan-Elimination:** Überführung der Matrix $A$ in Diagonalform durch elementare Zeilenumformungen

- Komplexität: $\Theta(n^3)$

**Elementare Zeilenumformungen:**

- Vielfaches einer Zeile zu einer anderen Zeile addieren
- zwei Zeilen vertauschen
- Vielfaches einer Zeile berechnen

für lineare Systeme: Gauss-Elimination auf erweiterter Matrix

$$(A|b) = \left(\begin{array}{ccc|c} M_{11} & \cdots & M_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ M_{n1} & \cdots & M_{nn} & b_n \end{array}\right)$$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1)\\
-3x & - & y & + & 2z & = & -11 & (G_2)\\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 \mathrel{+}= \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 \mathrel{+}= G_1}{\Longrightarrow}
\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 \mathrel{+}= -4 G_2}{\Longrightarrow}
\left(\begin{array}{ccc|c} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{ccc|c} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1)\\
-3x & - & y & + & 2z & = & -11 & (G_2)\\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 \mathrel{+}= \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 \mathrel{+}= G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 \mathrel{+}= -4 G_2}{\Longrightarrow}
\left(\begin{array}{rrr|r} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{rrr|r} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1) \\
-3x & - & y & + & 2z & = & -11 & (G_2) \\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 \mathrel{+}= \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 \mathrel{+}= G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 \mathrel{+}= -4 G_2}{\Longrightarrow}
\left(\begin{array}{rrr|r} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{rrr|r} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1) \\
-3x & - & y & + & 2z & = & -11 & (G_2) \\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 \mathrel{+}= \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 \mathrel{+}= G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 \mathrel{+}= -4 G_2}{\Longrightarrow}
\left(\begin{array}{rrr|r} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{rrr|r} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1) \\
-3x & - & y & + & 2z & = & -11 & (G_2) \\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 \mathrel{+}= \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 \mathrel{+}= G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 \mathrel{+}= -4 G_2}{\Longrightarrow}
\left(\begin{array}{rrr|r} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{rrr|r} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1)\\
-3x & - & y & + & 2z & = & -11 & (G_2)\\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 \mathrel{+}= \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 \mathrel{+}= G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 \mathrel{+}= -4 G_2}{\Longrightarrow}
\left(\begin{array}{rrr|r} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{rrr|r} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Beispiel

$$
\begin{array}{rcrcrcrl}
2x & + & y & - & z & = & 8 & (G_1)\\
-3x & - & y & + & 2z & = & -11 & (G_2)\\
-2x & + & y & + & 2z & = & -3 & (G_3)
\end{array}
$$

$$
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)
\overset{G_2 += \frac{3}{2} G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ -2 & 1 & 2 & -3 \end{array}\right)
$$

$$
\overset{G_3 += G_1}{\Longrightarrow}
\left(\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)
\overset{G_3 += -4 G_2}{\Longrightarrow}
\left(\begin{array}{rrr|r} -2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)
$$

$$
\overset{G_1/2,\ G_2 \cdot 2,\ G_3 \cdot (-1)}{\Longrightarrow}
\left(\begin{array}{rrr|r} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right)
$$

Lösung nun durch Rücksubstitution: $z = -1$, $y = 3$ und $x = 2$

# Gauss Elimination: Algorithmus

```
1  Gauss(M)
2      for (j=0; j<n-1; j++)
3          foundNonZero = false;
4          for (i=j; i<n-1; i++)
5              if (M_ij != 0)
6                  swap(i,j);
7                  foundNonZero = true;
8                  break;
9          if (!foundNonZero)
10             printf("Matrix not invertible");
11             return;
12         for (i=j+1; i<n-1; i++)
13             M_i += -(M_ij/M_jj)*M_j;
```

- Laufzeit: Die äußere Schleife wird $n$ mal durchlaufen; die inneren Schleifen $\leq n$ mal. D.h. jede Operation wird höchstens $n^2$ mal ausgeführt.
- Die swap-Operation in Zeile 5 und die Operation in Zeile 11 sind aber nicht elementar und benötigen Laufzeit $\mathcal{O}(n)$.
- $\mathcal{O}(n^3)$

# Erklärung

- $j$ geht über die Spalten; und $i$ über die Zeilen;
- Zu Beginn der von Iteration $j$, haben alle Zeilen $i \geq j$ Nullen in Spalten $\ell$ mit $\ell < j$.
- In Iteration $j$ bekommen die Zeilen $i > j$ Nullen in Spalte $\ell$;
- Die Zeile $j$ soll aber in Spalte $j$ einen Wert $M_{jj} \neq 0$ haben; dazu wird erst nach einer geeigneten Zeile gesucht (in den Zeilen mit $i \geq j$) und ggfs. eine Vertauschung vorgenommen. Falls dies fehlschlägt haben alle Zeilen $i \geq j$ Nullen in Spalte $\ell$ mit $1 \leq \ell \leq j$. D.h., diese insgesamt $n - j$ Zeilen haben höchstens $n - j - 1$ Spalten, die nicht Null sind. Damit sind diese Zeilen linear abhängig und die Matrix ist nicht invertierbar.
- In einer Iteration der for-Schleife in Zeile 11 wird die $j$-te Zeile mit einem geeigneten Vielfachen multipliziert und zu der $i$-ten Zeile addiert, so daß letztere einen Nulleintrag in Spalte $j$ bekommt.

# Inverse via Gauss-Jordan Elimination

Sei $A \in \mathbb{R}^{n \times n}$.

- Bilde erweiterte Matrix

$$(A|I_n) = \left(\begin{array}{ccc|ccc} a_{11} & \cdots & a_{n1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} & 0 & \cdots & 1 \end{array}\right)$$

  wobei $I_n$ die $n \times n$ Identitätsmatrix ist.

- Führe Gauss-Jordan Elimination für $A$ aus (linker Teil) bis auf $I_n$ reduziert, repliziere elementare Zeilenumformungen für $I_n$ (rechter Teil).

- Am Ende entspricht rechter Teil $A^{-1}$.

# Pseudoinverse

Falls $A^TA$ nicht invertierbar ist die Pseudoinverse gegeben durch die Matrix $A^+$, die folgende Bedinungen erfüllt:

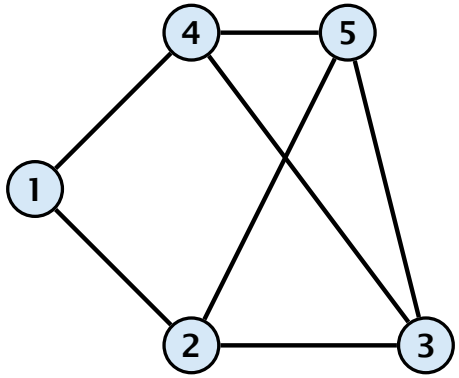
- $AA^+A = A$
- $A^+AA^+ = A+$
- $(AA^+)^T = AA^+$
- $(A^+A)^T = A^+A$

**Theorem**

Diese Matrix existiert immer.

# Anwendung

**gegeben:** ungerichteter Graph $G = (V, E)$



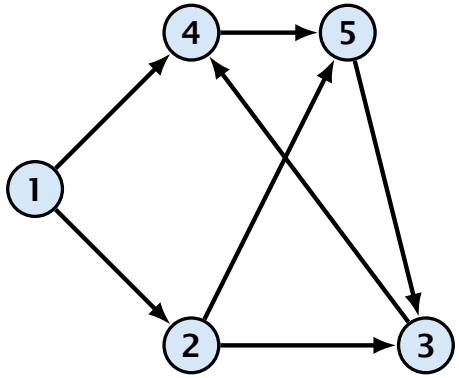

weise Kanten beliebige Richtungen zu (Zählpfeile)

**Definition:** Die Knoten-Kanten Inzidenzmatrix $B$ ist wie folgt definiert:

- eine Zeile für jede Kante; eine Spalte für jeden Knoten
- die Zeile für Kante $e = (u, v)$ enthält ein $1$ in Spalte $u$ und eine $-1$ in Spalte $v$

# Anwendung

**gegeben:** ungerichteter Graph $G = (V, E)$

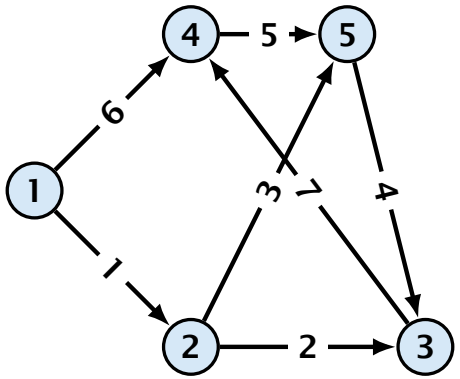



weise Kanten beliebige Richtungen zu (Zählpfeile)

**Definition:** Die Knoten-Kanten Inzidenzmatrix $B$ ist wie folgt definiert:

- eine Zeile für jede Kante; eine Spalte für jeden Knoten
- die Zeile für Kante $e = (u, v)$ enthält ein $1$ in Spalte $u$ und eine $-1$ in Spalte $v$

# Anwendung




$$\underbrace{\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 \end{pmatrix}}_{B}$$

Angenommen Vektor $x$ ordnet jedem Knoten ein Potential zu und die Kanten des Graphen haben Widerstand 1.

$Bx$ = Vektor von Strömen entlang der Kanten

# Anwendung

$Bx$ = Vektor von Strömen entlang der Kanten

$\underbrace{B^T B}_{L} x$ = Vektor von Strömen aus Knoten

Gegeben: Vektor $b \in \mathbb{R}^n$ der Ein- und Ausgangsströme für alle Knoten angibt (z.B. einfach +1 an Knoten $v$ und -1 an Knoten $u$ bei einer Stromquelle zwischen diesen Knoten).

Löse

$$Lx = b$$

$L^+ b$ ist die Lösung der Gleichung die $\|x\|_2$ minimiert.