

Algorithmen und Datenstrukturen SS'23

# Kapitel 1: Appetithäppchen

Marvin Künnemann

AG Algorithmen & Komplexität

# Überblick

---

**Heute:** Appetithäppchen für das Themengebiet **Algorithmik**

# Überblick

---

Heute: Appetithäppchen für das Themengebiet **Algorithmik**  
Algorithmes Problem mit zahlreichen Anwendungen: Langzahlarithmetik

# Überblick

---

Heute: Appetithäppchen für das Themengebiet **Algorithmik**  
Algorithmes Problem mit zahlreichen Anwendungen: Langzahlarithmetik

$$\begin{array}{c} x + y \\ \swarrow \quad \searrow \\ x \cdot y \end{array}$$

# Überblick

---

Heute: Appetithäppchen für das Themengebiet **Algorithmik**  
Algorithmes Problem mit zahlreichen Anwendungen: Langzahlarithmetik

$$x + y \quad x \cdot y$$


Anhand dieses Beispiels besprechen wir grundlegende Fragen:

# Überblick

---

Heute: Appetithäppchen für das Themengebiet **Algorithmik**  
Algorithmes Problem mit zahlreichen Anwendungen: Langzahlarithmetik

$$\begin{array}{c} & \nearrow \\ x + y & & x \cdot y \end{array}$$

Anhand dieses Beispiels besprechen wir grundlegende Fragen:

- Was ist ein **algorithmisches Problem**?
- Was ist ein **Algorithmus**?
- Wie bewerten wir die "Qualität" eines Algorithmus? → **Laufzeit**

# Überblick

---

Heute: Appetithäppchen für das Themengebiet **Algorithmik**  
Algorithmes Problem mit zahlreichen Anwendungen: Langzahlarithmetik

$$x + y \quad x \cdot y$$


Anhand dieses Beispiels besprechen wir grundlegende Fragen:

- Was ist ein **algorithmisches Problem**?
- Was ist ein **Algorithmus**?
- Wie bewerten wir die "Qualität" eines Algorithmus? → **Laufzeit**

**Highlight:** wir verbessern die Schulmethode für die Multiplikation!

# Algorithmische Probleme

---

Eingabe



Ausgabe

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:

Ausgabe:

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

# Algorithmische Probleme: Eingabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

# Algorithmische Probleme: Eingabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind  $x$  und  $y$  dargestellt?

# Algorithmische Probleme: Eingabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind  $x$  und  $y$  dargestellt? **Viele Möglichkeiten!**

# Algorithmische Probleme: Eingabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind  $x$  und  $y$  dargestellt? **Viele Möglichkeiten!**

· 27

# Algorithmische Probleme: Eingabe



## 1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind x und y dargestellt?

# Viele Möglichkeiten!

- 27
  - 

# Algorithmische Probleme: Eingabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind  $x$  und  $y$  dargestellt?

**Viele Möglichkeiten!**

- 27
- 
- 11011

# Algorithmische Probleme: Eingabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind  $x$  und  $y$  dargestellt?

**Viele Möglichkeiten!**

- 27
- 
- 11011
- 1B

# Algorithmische Probleme: Eingabe



## 1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind x und y dargestellt?

# Viele Möglichkeiten!

- 27
  - 
  - 11011
  - 1B
  - $\sqrt[3]{19683}$

# Algorithmische Probleme: Eingabe



## 1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Wie sind x und y dargestellt?

# Viele Möglichkeiten!

- 27
  - 
  - 11011
  - 1B
  - $\sqrt[3]{19683}$

Typischerweise benutzen wir die Darstellung in einem **Positionssystem**

# Exkurs: Positionssystem

---

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B - 1\}$ .

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B - 1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B - 1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

Übliche Wahlen für  $B$ :

## Binärsystem:

$B = 2$       → Ziffern: 0,1

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B-1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

Übliche Wahlen für  $B$ :

Beispiel: 27 ist

## Binärsystem:

$B = 2$

→ Ziffern: 0,1

$$11011_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1$$

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B-1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

Übliche Wahlen für  $B$ :

Beispiel: 27 ist

## Binärsystem:

$B = 2$  → Ziffern: 0,1

$$11011_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1$$

## Dezimalsystem:

$B = 10$  → Ziffern: 0,1,...,9

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B-1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

Übliche Wahlen für  $B$ :

## Binärsystem:

$B = 2$  → Ziffern: 0,1

$$11011_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1$$

## Dezimalsystem:

$B = 10$  → Ziffern: 0,1,...,9

Beispiel: 27 ist

$$27_{10} = 2 \cdot 10^1 + 7$$

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B-1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

Übliche Wahlen für  $B$ :

### Binärsystem:

$B = 2$  → Ziffern: 0, 1

$$11011_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1$$

### Dezimalsystem:

$B = 10$  → Ziffern: 0, 1, ..., 9

$$27_{10} = 2 \cdot 10^1 + 7$$

### Hexadezimalsystem:

$B = 16$  → Ziffern: 0, 1, ..., 9, A, B, C, D, E, F

Beispiel: 27 ist

# Exkurs: Positionssystem

---

## Definition

Eine Zahl zur Basis  $B$  ist eine Zeichenkette  $a_{n-1} \dots a_1 a_0$  mit  $a_i \in \{0, \dots, B-1\}$ .

Die Zeichenkette  $a_{n-1} \dots a_1 a_0$  stellt die folgende Zahl dar:

$$\sum_{i=0}^{n-1} a_i \cdot B^i = a_{n-1}B^{n-1} + \dots + a_2B^2 + a_1B + a_0$$

Übliche Wahlen für  $B$ :

### Binärsystem:

$B = 2$  → Ziffern: 0, 1

$$11011_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1$$

### Dezimalsystem:

$B = 10$  → Ziffern: 0, 1, ..., 9

$$27_{10} = 2 \cdot 10^1 + 7$$

### Hexadezimalsystem:

$B = 16$  → Ziffern: 0, 1, ..., 9, A, B, C, D, E, F

$$1B_{16} = 1 \cdot 16^1 + 11$$

# Algorithmische Probleme: Ausgabe

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

# Algorithmische Probleme: Ausgabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Was für eine Ausgabe ist gewünscht?

# Algorithmische Probleme: Ausgabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Was für eine Ausgabe ist gewünscht?

Gegeben Zahlen  $x = x_{n-1} \dots x_0$  und  $y = y_{n-1} \dots y_0$  zur Basis  $B$ ,  
gebe  $x + y$  als Zahl zur Basis  $B$  aus

# Algorithmische Probleme: Ausgabe

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Was für eine Ausgabe ist gewünscht?

Gegeben Zahlen  $x = x_{n-1} \dots x_0$  und  $y = y_{n-1} \dots y_0$  zur Basis  $B$ ,  
gebe  $x + y$  als Zahl zur Basis  $B$  aus

Bsp.: Gegeben 101 und 052, gebe 153 aus.

# Algorithmische Probleme: Algorithmus

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$x$	1	4	5	8	5	3	6
$y$	6	3	1	7	5	1	

---

$$x + y$$

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$$\begin{array}{r} x \\ y \\ \hline \end{array} \quad \begin{array}{r} 1 & 4 & 5 & 8 & 5 & 3 & 6 \\ 6 & 3 & 1 & 7 & 5 & 1 \\ \hline \end{array}$$

$$x + y$$

$$7$$

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

x	1	4	5	8	5	3	6
y	6	3	1	7	5	1	

$$x + y$$

$$\underline{\quad\quad\quad}$$

8 7

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$$\begin{array}{r} x \\ y \\ \hline \end{array} \quad \begin{array}{r} 1 & 4 & 5 & 8 & 5 & 3 & 6 \\ 6 & 3 & 1 & 7 & 5 & 1 \\ \hline \end{array} \quad \begin{array}{r} \text{Übertrag (carry)} \\ 1 \\ \hline x + y \\ 2 & 8 & 7 \end{array}$$

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$$\begin{array}{r} x \\ y \\ \text{Übertrag (carry)} \\ \hline x + y \end{array} \quad \begin{array}{ccccccc} 1 & 4 & 5 & 8 & 5 & 3 & 6 \\ 6 & 3 & 1 & 7 & 5 & 1 & \\ 1 & 1 & & & & & \\ \hline 0 & 2 & 8 & 7 & & & \end{array}$$

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$$\begin{array}{r} x \\ y \\ \text{Übertrag (carry)} \\ \hline x + y \end{array} \quad \begin{array}{ccccccc} 1 & 4 & 5 & 8 & 5 & 3 & 6 \\ 6 & 3 & 1 & 7 & 5 & 1 & \\ 1 & 1 & & & & & \\ \hline 9 & 0 & 2 & 8 & 7 & & \end{array}$$

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$x$	1	4	5	8	5	3	6
$y$		6	3	1	7	5	1
Übertrag (carry)		1	1	1			
<hr/>							
$x + y$		0	9	0	2	8	7

# Algorithmische Probleme: Algorithmus

---



1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

Ein **algorithmisches Problem** ist durch Eingabe & Ausgabe definiert. → **Was?**

Zentraler Inhalt dieser Vorlesung: das Verfahren zum Erreichen der Ausgabe. → **Wie?**

Aus der Schule kennen wir bereits ein Verfahren:

$x$	1	4	5	8	5	3	6
$y$		6	3	1	7	5	1
Übertrag (carry)		1	1	1			
<hr/>							
$x + y$	2	0	9	0	2	8	7

# **Exkurs: Algorithmen beschreiben**

---

Wie können wir einen Algorithmus **beschreiben**?

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

x	$x_{n-1}$	...	$x_1$	$x_0$
y	$y_{n-1}$	...	$y_1$	$y_0$

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \end{array} \quad \begin{array}{cccc} x_{n-1} & \dots & x_1 & x_0 \\ y_{n-1} & \dots & y_1 & y_0 \end{array}$$

---

$$z = x + y$$

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \\ \hline c_n & c_{n-1} & \dots & c_1 & c_0 \end{array}$$

---

$$z = x + y \quad z_n \quad z_{n-1} \quad \dots \quad z_1 \quad z_0$$

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \\ \hline c_n & c_{n-1} & \dots & c_1 & c_0 \end{array}$$
$$z = x + y \quad z_n \quad z_{n-1} \quad \dots \quad z_1 \quad z_0$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \\ \hline c_n & c_{n-1} & \dots & c_1 & c_0 \end{array}$$
$$z = x + y \quad z_n \quad z_{n-1} \quad \dots \quad z_1 \quad z_0$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \\ \hline c_n & c_{n-1} & \dots & c_1 & c_0 \end{array}$$
$$z = x + y \quad z_n \quad z_{n-1} \quad \dots \quad z_1 \quad z_0$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

für alle  $i = 0, \dots, n - 1$ : bestimme  $z_i$  und  $c_{i+1}$  aus  $x_i + y_i + c_i$ :

# Exkurs: Algorithmen beschreiben

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \\ \hline c_n & c_{n-1} & \dots & c_1 & c_0 \\ \hline z = x + y & z_n & z_{n-1} & \dots & z_1 & z_0 \end{array}$$

Teilaufgabe:

$$\begin{array}{r} x_i \\ y_i \\ \hline c_{i+1} & z_i \end{array}$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

für alle  $i = 0, \dots, n - 1$ : bestimme  $z_i$  und  $c_{i+1}$  aus  $x_i + y_i + c_i$ :

# Exkurs: Algorithmen beschreiben

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$x$	$x_{n-1}$	$\dots$	$x_1$	$x_0$
$y$	$y_{n-1}$	$\dots$	$y_1$	$y_0$
	$c_n$	$c_{n-1}$	$\dots$	$c_1$
<hr/>				
$z = x + y$	$z_n$	$z_{n-1}$	$\dots$	$z_1$
				$z_0$

Teilaufgabe:

$$\begin{array}{r} x_i \\ y_i \\ c_i \\ \hline c_{i+1} & z_i \end{array} \quad \begin{array}{r} 8 \\ 9 \\ 1 \\ \hline 18 \end{array}$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

für alle  $i = 0, \dots, n - 1$ : bestimme  $z_i$  und  $c_{i+1}$  aus  $x_i + y_i + c_i$ :

# Exkurs: Algorithmen beschreiben

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$$\begin{array}{r} x \\ y \\ \hline c_n & c_{n-1} & \dots & c_1 & c_0 \\ z = x + y & z_n & z_{n-1} & \dots & z_1 & z_0 \end{array}$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

für alle  $i = 0, \dots, n - 1$ : bestimme  $z_i$  und  $c_{i+1}$  aus  $x_i + y_i + c_i$ :

Teilaufgabe:

$$\begin{array}{r} x_i \\ y_i \\ c_i \\ \hline c_{i+1} & z_i \end{array} \quad \begin{array}{r} 8 \\ 9 \\ 1 \\ \hline 18 \end{array}$$

**Elementaroperation:**  
3 Ziffern aufaddieren

# Exkurs: Algorithmen beschreiben

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$x$		$x_{n-1}$	$\dots$	$x_1$	$x_0$
$y$		$y_{n-1}$	$\dots$	$y_1$	$y_0$
	$c_n$	$c_{n-1}$	$\dots$	$c_1$	$c_0$
<hr/>					
$z = x + y$	$z_n$	$z_{n-1}$	$\dots$	$z_1$	$z_0$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

für alle  $i = 0, \dots, n - 1$ : bestimme  $z_i$  und  $c_{i+1}$  aus  $x_i + y_i + c_i$ :

$$c_{i+1} \cdot B + z_i = x_i + y_i + c_i$$

Teilaufgabe:

$$\begin{array}{r} x_i & & & 8 \\ y_i & & & 9 \\ c_i & & & 1 \\ \hline c_{i+1} & z_i & & 18 \end{array}$$

**Elementaroperation:**  
3 Ziffern aufaddieren

# Exkurs: Algorithmen beschreiben

Wie können wir einen Algorithmus **beschreiben**?

Versuchen wir zunächst, ihn mathematisch zu beschreiben:

$x$		$x_{n-1}$	...	$x_1$	$x_0$
$y$		$y_{n-1}$	...	$y_1$	$y_0$
	$c_n$	$c_{n-1}$	...	$c_1$	$c_0$
<hr/>					
$z = x + y$	$z_n$	$z_{n-1}$	...	$z_1$	$z_0$

Teilaufgabe:

$$\begin{array}{r} x_i \\ y_i \\ c_i \\ \hline c_{i+1} & z_i \end{array} \quad \begin{array}{r} 8 \\ 9 \\ 1 \\ \hline 18 \end{array}$$

wobei sich die  $c_i$ 's und  $z_i$ 's wie folgt ergeben:

setze  $c_0 = 0$

für alle  $i = 0, \dots, n - 1$ : bestimme  $z_i$  und  $c_{i+1}$  aus  $x_i + y_i + c_i$ :

$$c_{i+1} \cdot B + z_i = x_i + y_i + c_i$$

wir schreiben:  $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$

**Elementaroperation:**  
3 Ziffern aufaddieren

# **Exkurs: Algorithmen beschreiben**

---

Wie können wir einen Algorithmus **beschreiben**?

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

`add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):`

`$c_0 \leftarrow 0$`

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
     $c_0 \leftarrow 0$ 
```

```
    for  $i = 0, \dots, n - 1$  do:
```

```
        |
```

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
     $c_0 \leftarrow 0$ 
```

```
    for  $i = 0, \dots, n - 1$  do:
```

```
        // elementare Operation
```

```
         $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
     $c_0 \leftarrow 0$ 
```

```
    for  $i = 0, \dots, n - 1$  do:
```

```
        // elementare Operation
```

```
         $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
     $z_n \leftarrow c_n$ 
```

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
     $c_0 \leftarrow 0$ 
```

```
    for  $i = 0, \dots, n - 1$  do:
```

```
        // elementare Operation
```

```
         $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
     $z_n \leftarrow c_n$ 
```

```
    return  $z_n \dots z_0$ 
```

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
 $c_0 \leftarrow 0$ 
```

```
for  $i = 0, \dots, n - 1$  do:
```

```
    // elementare Operation
```

```
     $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
 $z_n \leftarrow c_n$ 
```

```
return  $z_n \dots z_0$ 
```

**Pseudocode:**

# Exkurs: Algorithmen beschreiben

---

Wie können wir einen Algorithmus **beschreiben**?

im Wesentlichen ist ein Algorithmus eine Sequenz an (elementaren) Anweisungen  
(Vergleich: ein Rezept ist eine Sequenz an Kochanweisungen)

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
 $c_0 \leftarrow 0$ 
```

```
for  $i = 0, \dots, n - 1$  do:
```

```
    // elementare Operation
```

```
     $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
 $z_n \leftarrow c_n$ 
```

```
return  $z_n \dots z_0$ 
```

## Pseudocode:

- einfache, semi-formale Beschreibung
- lässt sich einfach in beliebige Programmiersprache übersetzen

# "Qualität" eines Algorithmus'?

---

Wie **gut** ist ein Algorithmus?

# "Qualität" eines Algorithmus?

---

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

# "Qualität" eines Algorithmus'?

---

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

Fokus in der Vorlesung: Zeitbedarf, **Laufzeit**

# "Qualität" eines Algorithmus'?

---

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

Fokus in der Vorlesung: Zeitbedarf, **Laufzeit**

Wie schnell läuft ein Algorithmus?

# "Qualität" eines Algorithmus?

---

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

Fokus in der Vorlesung: Zeitbedarf, **Laufzeit**

Wie schnell läuft ein Algorithmus?      Ansatz: Zählen der elementaren Operationen!

# "Qualität" eines Algorithmus'?

---

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

Fokus in der Vorlesung: Zeitbedarf, **Laufzeit**

Wie schnell läuft ein Algorithmus?      Ansatz: Zählen der elementaren Operationen!

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
     $c_0 \leftarrow 0$ 
```

```
    for  $i = 0, \dots, n - 1$  do:
```

```
        // elementare Operation
```

```
         $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
     $z_n \leftarrow c_n$ 
```

```
    return  $z_n \dots z_0$ 
```

# "Qualität" eines Algorithmus'?

---

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

Fokus in der Vorlesung: Zeitbedarf, **Laufzeit**

Wie schnell läuft ein Algorithmus?      Ansatz: Zählen der elementaren Operationen!

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
 $c_0 \leftarrow 0$ 
```

```
for  $i = 0, \dots, n - 1$  do:
```

```
    // elementare Operation
```

```
     $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
 $z_n \leftarrow c_n$ 
```

```
return  $z_n \dots z_0$ 
```

**Frage:**

Wie viele elementaren Operationen benötigt **add**?

# "Qualität" eines Algorithmus'?

Wie **gut** ist ein Algorithmus?

benutzt wenig Ressourcen (z.B.: Zeitbedarf, Speicherplatz, Energie, ...)

Fokus in der Vorlesung: Zeitbedarf, **Laufzeit**

Wie schnell läuft ein Algorithmus?

Ansatz: Zählen der elementaren Operationen!

```
add( $x_{n-1} \dots x_0, y_{n-1} \dots y_0$ ):
```

```
 $c_0 \leftarrow 0$ 
```

```
for  $i = 0, \dots, n - 1$  do:
```

```
    // elementare Operation  
     $(c_{i+1}, z_i) \leftarrow x_i + y_i + c_i$ 
```

```
 $z_n \leftarrow c_n$ 
```

```
return  $z_n \dots z_0$ 
```

**Frage:**

Wie viele elementaren Operationen benötigt **add**?

**Theorem**

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $n$  elementaren Operationen zu einer  $(n + 1)$ -stelligen Zahl  $z = x + y$  addieren.

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

2. Beispiel: **Multiplikation** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x \cdot y$

# Algorithmische Probleme

---

Eingabe → ? → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

2. Beispiel: **Multiplikation** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x \cdot y$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

3 1 2 . 2 1 4 1

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} \\ \cdot \quad \underline{2 \quad 1 \quad 4 \quad 1} \\ \hline \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} 3 \ 1 \ 2 \\ \times \ 2 \ 1 \ 4 \ 1 \\ \hline 3 \ 1 \ 2 \end{array}$$

---

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} 3 \ 1 \ 2 \\ \times \ 2 \ 1 \ 4 \ 1 \\ \hline & 3 \ 1 \ 2 \\ 1 \ 2 \ 4 \ 8 \\ \hline \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & 3 \quad 1 \quad 2 \\ & & 1 \quad 2 \quad 4 \quad 8 \\ & & 3 \quad 1 \quad 2 \\ \hline \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} \\ \cdot \quad \underline{2 \quad 1 \quad 4 \quad 1} \\ \hphantom{3 \quad 1 \quad 2} \quad \underline{\quad 3 \quad 1 \quad 2} \\ \hphantom{3 \quad 1 \quad 2} \quad \underline{1 \quad 2 \quad 4 \quad 8} \\ \hphantom{3 \quad 1 \quad 2} \quad \underline{3 \quad 1 \quad 2} \\ \hphantom{3 \quad 1 \quad 2} \quad \underline{6 \quad 2 \quad 4} \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad 3 \quad 1 \quad 2 \\ \hline & & 6 \quad 2 \quad 4 \\ & & \hline & 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$4 \quad 4 \quad 2 \quad \cdot \quad 3$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{\quad 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ & & \hline \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ & & \underline{6} \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ & & \underline{1} \\ & & \underline{2 \quad 6} \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \underline{2 \quad 2 \quad 6} \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \begin{array}{r} 3 \quad 1 \quad 2 \\ 1 \quad 2 \quad 4 \quad 8 \\ 3 \quad 1 \quad 2 \\ \hline 6 \quad 2 \quad 4 \end{array} \\ & & \hline \\ & & \begin{array}{r} 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2 \end{array} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ \hline 6 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \underline{2 \quad 2 \quad 6} \\ 2 \quad 6 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \begin{array}{r} 3 \quad 1 \quad 2 \\ 1 \quad 2 \quad 4 \quad 8 \\ 3 \quad 1 \quad 2 \\ \hline 6 \quad 2 \quad 4 \end{array} \\ & & \hline \\ & & \begin{array}{r} 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2 \end{array} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ \hline 3 \quad 2 \quad 6 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \quad \quad \quad \underline{3 \quad 1 \quad 2} \\ & & \quad \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \underline{2 \quad 2 \quad 6} \\ 1 \quad 3 \quad 2 \quad 6 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \begin{array}{r} 3 \quad 1 \quad 2 \\ 1 \quad 2 \quad 4 \quad 8 \\ 3 \quad 1 \quad 2 \\ \hline 6 \quad 2 \quad 4 \end{array} \\ & & \hline \\ & & \begin{array}{r} 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2 \end{array} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ \hline 1 \quad 3 \quad 2 \quad 6 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \begin{array}{r} 3 \quad 1 \quad 2 \\ 1 \quad 2 \quad 4 \quad 8 \\ 3 \quad 1 \quad 2 \\ \hline 6 \quad 2 \quad 4 \end{array} \\ & & \hline \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

Beispiel:

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ \hline 1 \quad 3 \quad 2 \quad 6 \end{array}$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} 3 \quad 1 \quad 2 \\ \cdot \quad \quad \quad 2 \quad 1 \quad 4 \quad 1 \\ \hline & & 3 \quad 1 \quad 2 \\ & 1 \quad 2 \quad 4 \quad 8 \\ & 3 \quad 1 \quad 2 \\ \hline 6 \quad 2 \quad 4 \\ \hline 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2 \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

Beispiel:

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} 4 \quad 4 \quad 2 \\ \cdot \quad \quad \quad 3 \\ \hline 1 \quad 1 \\ 2 \quad 2 \quad 6 \\ \hline 1 \quad 3 \quad 2 \quad 6 \end{array}$$

$$\begin{array}{r} a_i \\ \cdot \quad \beta \\ \hline c_i \quad d_i \end{array} \qquad a_i \cdot \beta = c_i \cdot B + d_i$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \underline{1 \quad 2 \quad 4 \quad 8} \\ & & \underline{3 \quad 1 \quad 2} \\ & & \underline{6 \quad 2 \quad 4} \\ \hline & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

Beispiel:

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ 1 \quad 3 \quad 2 \quad 6 \end{array}$$

$$\begin{array}{r} a_i & & 3 \\ \cdot & \beta & \cdot 4 \\ \hline c_i \ d_i & & 1 \ 2 \end{array} \qquad a_i \cdot \beta = c_i \cdot B + d_i$$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \quad \quad 1 \quad 2 \quad 4 \quad 8 \\ & & \quad \quad 3 \quad 1 \quad 2 \\ & & \underline{6 \quad 2 \quad 4} \\ & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

Beispiel:

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \underline{2 \quad 2 \quad 6} \\ 1 \quad 3 \quad 2 \quad 6 \end{array}$$

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} a_i & & 3 \\ \cdot \beta & & \cdot 4 \\ \hline c_i \ d_i & & 1 \ 2 \end{array} \qquad a_i \cdot \beta = c_i \cdot B + d_i$$

2. Wir berechnen die Summe von  $c_{n-1} \dots c_0 0$  und  $d_{n-1} \dots d_0$

# Schulmethode zur Multiplikation

---

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \underline{1 \quad 2 \quad 4 \quad 8} \\ & & \underline{3 \quad 1 \quad 2} \\ & & \underline{6 \quad 2 \quad 4} \\ \hline & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

---

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

Beispiel:

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ 1 \quad 3 \quad 2 \quad 6 \end{array}$$

$$\begin{array}{r} a_i & & 3 \\ \cdot & \beta & \cdot 4 \\ \hline c_i \ d_i & & 1 \ 2 \end{array} \quad a_i \cdot \beta = c_i \cdot B + d_i$$

2. Wir berechnen die Summe von  $c_{n-1} \dots c_0 0$  und  $d_{n-1} \dots d_0$

$$\begin{array}{r} c_{n-1} \ c_{n-2} \ \dots \ c_0 \ 0 \\ + \ d_{n-1} \ \dots \ d_1 \ d_0 \\ \hline p_n \ p_{n-1} \ \dots \ p_1 \ p_0 \end{array}$$

# Schulmethode zur Multiplikation

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \quad \underline{1 \quad 2 \quad 4 \quad 8} \\ & & \quad \underline{3 \quad 1 \quad 2} \\ & & \underline{6 \quad 2 \quad 4} \\ \hline & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

Beispiel:

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} \underline{4 \quad 4 \quad 2} & \cdot & 3 \\ 1 \quad 1 \\ \hline 2 \quad 2 \quad 6 \\ 1 \quad 3 \quad 2 \quad 6 \end{array}$$

$$\begin{array}{r} a_i & & 3 \\ \cdot & \beta & \cdot 4 \\ \hline c_i \ d_i & & 1 \ 2 \end{array} \quad a_i \cdot \beta = c_i \cdot B + d_i$$

2. Wir berechnen die Summe von  $c_{n-1} \dots c_0 0$  und  $d_{n-1} \dots d_0$

$$\begin{array}{r} c_{n-1} \ c_{n-2} \ \dots \ c_0 \ 0 \\ + \ d_{n-1} \ \dots \ d_1 \ d_0 \\ \hline p_n \ p_{n-1} \ \dots \ p_1 \ p_0 \end{array} \rightarrow \text{unser gewünschtes Ergebnis}$$

# Schulmethode zur Multiplikation

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \underline{1 \quad 2 \quad 4 \quad 8} \\ & & \underline{3 \quad 1 \quad 2} \\ & & \underline{6 \quad 2 \quad 4} \\ \hline & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

$n$  Elem.-operationen

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} a_i & \quad 3 \\ \cdot \beta & \cdot 4 \\ \hline c_i \ d_i & \underline{1 \ 2} \end{array} \quad a_i \cdot \beta = c_i \cdot B + d_i$$

2. Wir berechnen die Summe von  $c_{n-1} \dots c_0 0$  und  $d_{n-1} \dots d_0$

$$\begin{array}{r} c_{n-1} \ c_{n-2} \ \dots \ c_0 \ 0 \\ + \ \underline{d_{n-1} \ \dots \ d_1 \ d_0} \\ \hline p_n \ p_{n-1} \ \dots \ p_1 \ p_0 \end{array} \rightarrow \text{unser gewünschtes Ergebnis}$$

# Schulmethode zur Multiplikation

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \underline{1 \quad 2 \quad 4 \quad 8} \\ & & \underline{3 \quad 1 \quad 2} \\ & & \underline{6 \quad 2 \quad 4} \\ \hline & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} a_i & \underline{3} \\ \cdot \beta & \cdot 4 \\ \hline c_i d_i & \underline{1 \quad 2} \end{array} \qquad a_i \cdot \beta = c_i \cdot B + d_i$$

2. Wir berechnen die Summe von  $c_{n-1} \dots c_0 0$  und  $d_{n-1} \dots d_0$

$$\begin{array}{r} c_{n-1} c_{n-2} \dots c_0 0 \\ + \underline{d_{n-1} \dots d_1 d_0} \\ \hline p_n p_{n-1} \dots p_1 p_0 \end{array} \rightarrow \text{unser gewünschtes Ergebnis}$$

$n$  Elem.-operationen

$n$  Elem.-operationen  
→ letztes Theorem

# Schulmethode zur Multiplikation

Kleine Erinnerung:

$$\begin{array}{r} \underline{3 \quad 1 \quad 2} & \cdot & \underline{2 \quad 1 \quad 4 \quad 1} \\ & & \underline{\quad \quad \quad 3 \quad 1 \quad 2} \\ & & \underline{1 \quad 2 \quad 4 \quad 8} \\ & & \underline{3 \quad 1 \quad 2} \\ & & \underline{6 \quad 2 \quad 4} \\ \hline & & \underline{6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2} \end{array}$$

Wichtige Subroutine: Berechne  $p = a_{n-1} \dots a_0 \cdot \beta$  für eine gegebene Ziffer  $\beta$

$n$  Elem.-operationen

1. Wir berechnen  $a_i \cdot \beta$  für alle  $i = 0, \dots, n - 1$

**Elementaroperation:** Zwei Zwiffern multiplizieren

$$\begin{array}{r} a_i & \quad 3 \\ \cdot \beta & \quad \cdot 4 \\ \hline c_i \ d_i & \quad 1 \ 2 \end{array} \quad a_i \cdot \beta = c_i \cdot B + d_i$$

$n$  Elem.-operationen  
→ letztes Theorem

2. Wir berechnen die Summe von  $c_{n-1} \dots c_0 0$  und  $d_{n-1} \dots d_0$

$$\begin{array}{r} c_{n-1} \ c_{n-2} \ \dots \ c_0 \ 0 \\ + \ d_{n-1} \ \dots \ d_1 \ d_0 \\ \hline p_n \ p_{n-1} \ \dots \ p_1 \ p_0 \end{array} \rightarrow \text{unser gewünschtes Ergebnis}$$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$$\begin{array}{r} 3 & 1 & 2 & \cdot & 2 & 1 & 4 & 1 \\ \times & & & & 3 & 1 & 2 \\ \hline & & & & 1 & 2 & 4 & 8 \\ & & & & 3 & 1 & 2 \\ & & & & 6 & 2 & 4 \\ \hline & 6 & 6 & 7 & 9 & 9 & 2 \end{array}$$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r} 3 & 1 & 2 & \cdot & 2 & 1 & 4 & 1 \\ \times & & & & 3 & 1 & 2 \\ \hline & & & & 1 & 2 & 4 & 8 \\ & & & & 3 & 1 & 2 & \\ \hline & & & & 6 & 2 & 4 & \\ & & & & 6 & 6 & 7 & 9 & 9 & 2 \end{array}$$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r} a_{n-1} \dots a_0 \quad . \\ \hline b_{n-1} \quad \dots \quad b_1 \quad b_0 \end{array}$$

$$\begin{array}{r} 3 \quad 1 \quad 2 \quad . \quad 2 \quad 1 \quad 4 \quad 1 \\ \hline & & & & 3 \quad 1 \quad 2 \\ & & & & 1 \quad 2 \quad 4 \quad 8 \\ & & & & 3 \quad 1 \quad 2 \\ & & & & 6 \quad 2 \quad 4 \\ \hline & 6 \quad 6 \quad 7 \quad 9 \quad 9 \quad 2 \end{array}$$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r} 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\ \hline & & & & 3 & 1 & 2 \\ & & & & 1 & 2 & 4 & 8 \\ & & & & 3 & 1 & 2 \\ & & & & 6 & 2 & 4 \\ \hline & 6 & 6 & 7 & 9 & 9 & 2 \end{array}$$

$a_{n-1} \dots a_0 \cdot .$	$b_{n-1}$	$\dots$	$b_1$	$b_0$
	$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$
				$p_0^{(0)}$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r} 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\ \hline & & & & 3 & 1 & 2 \\ & & & & 1 & 2 & 4 & 8 \\ & & & & 3 & 1 & 2 \\ & & & & 6 & 2 & 4 \\ \hline & 6 & 6 & 7 & 9 & 9 & 2 \end{array}$$

$a_{n-1} \dots a_0 \cdot .$	$b_{n-1}$	$\dots$	$b_1$	$b_0$
$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$	$p_0^{(0)}$
$p_n^{(1)}$	$p_{n-1}^{(1)}$	$\dots$	$p_1^{(1)}$	$p_0^{(1)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r} 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\ \hline & & & & 3 & 1 & 2 \\ & & & & 1 & 2 & 4 & 8 \\ & & & & 3 & 1 & 2 \\ & & & & 6 & 2 & 4 \\ \hline & 6 & 6 & 7 & 9 & 9 & 2 \end{array}$$

$a_{n-1} \dots a_0 \cdot .$	$b_{n-1}$	$\dots$	$b_1$	$b_0$
$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$	$p_0^{(0)}$
$p_n^{(1)}$	$p_{n-1}^{(1)}$	$\dots$	$p_1^{(1)}$	$p_0^{(1)}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 & 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & & 3 & 1 & 2 \\
 & & & & & 1 & 2 & 4 & 8 \\
 & & & & & 3 & 1 & 2 \\
 & & & & & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

$a_{n-1} \dots a_0 \cdot .$	$b_{n-1}$	$\dots$	$b_1$	$b_0$
$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$	$p_0^{(0)}$
$p_n^{(1)}$	$p_{n-1}^{(1)}$	$\dots$	$p_1^{(1)}$	$p_0^{(1)}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$p_n^{(n-1)}$	$\dots$	$p_1^{(n-1)}$	$p_0^{(n-1)}$	

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 & 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & & 3 & 1 & 2 \\
 & & & & & 1 & 2 & 4 & 8 \\
 & & & & & 3 & 1 & 2 \\
 & & & & & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

$a_{n-1} \dots a_0 \cdot .$	$b_{n-1}$	$\dots$	$b_1$	$b_0$
$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$	$p_0^{(0)}$
$p_n^{(1)}$	$p_{n-1}^{(1)}$	$\dots$	$p_1^{(1)}$	$p_0^{(1)}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$p_n^{(n-1)}$	$\dots$	$p_1^{(n-1)}$	$p_0^{(n-1)}$	
$c_{2n-1} \dots$	$c_{n+1}$	$c_n$	$c_{n-1}$	$c_1 \dots c_0$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 & 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & & 3 & 1 & 2 \\
 & & & & & 1 & 2 & 4 & 8 \\
 & & & & & 3 & 1 & 2 \\
 & & & & & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{ccccc}
 a_{n-1} \dots a_0 & . & b_{n-1} & \dots & b_1 & b_0 \\
 \hline
 p^{(0)}B^0 & & p_n^{(0)} & p_{n-1}^{(0)} & \dots & p_1^{(0)} & p_0^{(0)} \\
 p_n^{(1)} & p_{n-1}^{(1)} & \dots & p_1^{(1)} & p_0^{(1)} & & \\
 & & \dots & & & & \\
 p_n^{(n-1)} & \dots & p_1^{(n-1)} & p_0^{(n-1)} & & & \\
 \hline
 c_{2n-1} \dots & c_{n+1} & c_n & c_{n-1} & \dots & c_1 & c_0
 \end{array}
 \end{array}$$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & 3 & 1 & 2 \\
 & & & & 1 & 2 & 4 & 8 \\
 & & & & 3 & 1 & 2 \\
 & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

	$a_{n-1} \dots a_0$	.		$b_{n-1}$	$\dots$	$b_1$	$b_0$
$p^{(0)}B^0$			$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$	$p_0^{(0)}$
$p^{(1)}B^1$		$p_n^{(1)}$	$p_{n-1}^{(1)}$	$\dots$	$p_1^{(1)}$	$p_0^{(1)}$	
				$\dots$			
	$p_n^{(n-1)}$	$\dots$	$p_1^{(n-1)}$	$p_0^{(n-1)}$			
	$c_{2n-1} \dots c_{n+1}$	$c_n$	.	$c_{n-1}$	$\dots$	$c_1$	$c_0$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & 3 & 1 & 2 \\
 & & & & 1 & 2 & 4 & 8 \\
 & & & & 3 & 1 & 2 \\
 & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 & 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & & 3 & 1 & 2 \\
 & & & & & 1 & 2 & 4 & 8 \\
 & & & & & 3 & 1 & 2 \\
 & & & & & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

$a_{n-1} \dots a_0 \cdot .$	$b_{n-1}$	$\dots$	$b_1$	$b_0$
$p^{(0)}B^0$	$p_n^{(0)}$	$p_{n-1}^{(0)}$	$\dots$	$p_1^{(0)}$
$p^{(1)}B^1$	$p_n^{(1)}$	$p_{n-1}^{(1)}$	$\dots$	$p_0^{(1)}$
			$\dots$	
$p^{(n-1)}B^{n-1}$	$p_n^{(n-1)}$	$\dots$	$p_1^{(n-1)}$	$p_0^{(n-1)}$
	$c_{2n-1} \dots$	$c_{n+1}$	$c_n$	$c_{n-1} \dots c_1 c_0$

2. Wir berechnen die Summe aller  $p^{(j)}B^j$  für  $j = 0, \dots, n - 1$

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 & 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & & 3 & 1 & 2 \\
 & & & & & 1 & 2 & 4 & 8 \\
 & & & & & 3 & 1 & 2 \\
 & & & & & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{ccccc}
 a_{n-1} \dots a_0 & . & b_{n-1} & \dots & b_1 & b_0 \\
 \hline
 p^{(0)}B^0 & & p_n^{(0)} & p_{n-1}^{(0)} & \dots & p_1^{(0)} & p_0^{(0)} \\
 p^{(1)}B^1 & & p_n^{(1)} & p_{n-1}^{(1)} & \dots & p_1^{(1)} & p_0^{(1)} \\
 & & & \dots & & & \\
 & & & & & & \\
 p^{(n-1)}B^{n-1} & & p_n^{(n-1)} & p_1^{(n-1)} & p_0^{(n-1)} & & \\
 \hline
 c_{2n-1} \dots & c_{n+1} & c_n & c_{n-1} & \dots & c_1 & c_0
 \end{array}
 \end{array}$$

2. Wir berechnen die Summe aller  $p^{(j)}B^j$  für  $j = 0, \dots, n - 1$

$\rightarrow n$  Additionen von  $\leq 2n$ -stelligen Zahlen

# Schulmethode zur Multiplikation II

Wollen berechnen:  $a_{n-1} \dots a_0 \cdot b_{n-1} \dots b_0$

1. Wir berechnen  $p^{(j)} = a_{n-1} \dots a_0 \cdot b_j$  für alle  $j = 0, \dots, n - 1$

$\rightarrow p^{(j)}$  ist eine  $(n + 1)$ -stellige Zahl  $p_n^{(j)} \dots p_0^{(j)}$

$$\begin{array}{r}
 & 3 & 1 & 2 & . & 2 & 1 & 4 & 1 \\
 \hline
 & & & & & 3 & 1 & 2 \\
 & & & & & 1 & 2 & 4 & 8 \\
 & & & & & 3 & 1 & 2 \\
 & & & & & 6 & 2 & 4 \\
 \hline
 & 6 & 6 & 7 & 9 & 9 & 2
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{ccccc}
 a_{n-1} \dots a_0 & . & b_{n-1} & \dots & b_1 & b_0 \\
 \hline
 p^{(0)}B^0 & & p_n^{(0)} & p_{n-1}^{(0)} & \dots & p_1^{(0)} & p_0^{(0)} \\
 p^{(1)}B^1 & & p_n^{(1)} & p_{n-1}^{(1)} & \dots & p_1^{(1)} & p_0^{(1)} \\
 & & & \dots & & & \\
 & & & & & & \\
 p^{(n-1)}B^{n-1} & & p_n^{(n-1)} & p_1^{(n-1)} & p_0^{(n-1)} & & \\
 \hline
 c_{2n-1} \dots & c_{n+1} & c_n & c_{n-1} & \dots & c_1 & c_0
 \end{array}
 \end{array}$$

2. Wir berechnen die Summe aller  $p^{(j)}B^j$  für  $j = 0, \dots, n - 1$

$\rightarrow n$  Additionen von  $\leq 2n$ -stelligen Zahlen

$\rightarrow$  Ergebnis ist  $2n$ -stellige Zahl  $c = c_{2n-1} \dots c_0$

# Schulmethode zur Multiplikation: Pseudocode

---

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
    s  $\leftarrow$  0  
    for  $j = 0, \dots, n - 1$  do:  
        |  $p \leftarrow \text{multZiffer}(a, b_j)$   
        |  $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return s
```

# Schulmethode zur Multiplikation: Pseudocode

---

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
    s  $\leftarrow$  0  
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow$  multZiffer( $a, b_j$ )  
        s  $\leftarrow$  add(s,  $p_n \dots p_0$   $\underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return s
```

berechnet  $a_{n-1} \dots a_0 \cdot b_j$  mit  
2n Elementaroperationen  
(↗ Folie 12)

# Schulmethode zur Multiplikation: Pseudocode

---

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
    s  $\leftarrow$  0  
    for  $j = 0, \dots, n - 1$  do:  
        |  $p \leftarrow \text{multZiffer}(a, b_j)$   
        |  $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return s
```

berechnet  $a_{n-1} \dots a_0 \cdot b_j$  mit  
2n Elementaroperationen  
(↗ Folie 12)

berechnet Summe (↗ Folie 10)

# Schulmethode zur Multiplikation: Pseudocode

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):
```

```
     $s \leftarrow 0$ 
```

```
    for  $j = 0, \dots, n - 1$  do:
```

```
         $p \leftarrow \text{multZiffer}(a, b_j)$ 
```

```
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$ 
```

```
    return  $s$ 
```

berechnet  $a_{n-1} \dots a_0 \cdot b_j$  mit  
2n Elementaroperationen  
(↗ Folie 12)

berechnet Summe (↗ Folie 10)

# Schulmethode zur Multiplikation: Pseudocode

---

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):
```

```
     $s \leftarrow 0$ 
```

```
    for  $j = 0, \dots, n - 1$  do:
```

```
         $p \leftarrow \text{multZiffer}(a, b_j)$ 
```

```
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$ 
```

```
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

---

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

In jeder Iteration benötigen wir:

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):
```

```
     $s \leftarrow 0$ 
```

```
    for  $j = 0, \dots, n - 1$  do:
```

```
         $p \leftarrow \text{multZiffer}(a, b_j)$ 
```

```
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$ 
```

```
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

---

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):
```

```
     $s \leftarrow 0$ 
```

```
    for  $j = 0, \dots, n - 1$  do:
```

```
         $p \leftarrow \text{multZiffer}(a, b_j)$ 
```

```
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$ 
```

```
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

---

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):
```

```
     $s \leftarrow 0$ 
```

```
    for  $j = 0, \dots, n - 1$  do:
```

```
         $p \leftarrow \text{multZiffer}(a, b_j)$ 
```

```
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$ 
```

```
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

---

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

---

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig
  - $n + j + 1 \leq 2n$

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig
  - $n + j + 1 \leq 2n$
  - $\text{add}(\cdot, \cdot)$  benötigt  $N$  Elementaroperationen für  $N$ -stellige Zahlen

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

# Schulmethode zur Multiplikation: Pseudocode

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig
  - $n + j + 1 \leq 2n$
  - $\text{add}(\cdot, \cdot)$  benötigt  $N$  Elementaroperationen für  $N$ -stellige Zahlen

Über alle  $n$  Iterationen sind das höchstens

# Schulmethode zur Multiplikation: Pseudocode

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig
  - $n + j + 1 \leq 2n$
- $\text{add}(\cdot, \cdot)$  benötigt  $N$  Elementaroperationen für  $N$ -stellige Zahlen

Über alle  $n$  Iterationen sind das höchstens

$$n \cdot (4n) = 4n^2 \text{ Elementaroperationen. } \square$$

# Schulmethode zur Multiplikation: Pseudocode

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig
  - $n + j + 1 \leq 2n$
- $\text{add}(\cdot, \cdot)$  benötigt  $N$  Elementaroperationen für  $N$ -stellige Zahlen

Über alle  $n$  Iterationen sind das höchstens

$$n \cdot (4n) = 4n^2 \text{ Elementaroperationen. } \square$$

Tatsächlich bräuchten wir sogar nur  $\approx 3n^2$  Elem.-operationen.

# Schulmethode zur Multiplikation: Pseudocode

## Theorem

Wir können zwei  $n$ -stellige Zahlen  $x, y$  mit  $4n^2$  elementaren Operationen zu einer  $2n$ -stelligen Zahl  $x \cdot y$  multiplizieren.

```
mult( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ ):  
     $s \leftarrow 0$   
    for  $j = 0, \dots, n - 1$  do:  
         $p \leftarrow \text{multZiffer}(a, b_j)$   
         $s \leftarrow \text{add}(s, p_n \dots p_0 \underbrace{0 \dots 0}_{j \text{ Nullen}})$   
    return  $s$ 
```

## Beweis

In jeder Iteration benötigen wir:

- $2n$  Elementaroperationen für  $\text{multZiffer}(\cdot, \cdot)$
- $\leq 2n$  Elementaroperationen für  $\text{add}(\cdot, \cdot)$ , denn:
  - $s$  und  $pB^j$  sind höchstens  $(n + j + 1)$ -stellig
  - $n + j + 1 \leq 2n$
  - $\text{add}(\cdot, \cdot)$  benötigt  $N$  Elementaroperationen für  $N$ -stellige Zahlen

Über alle  $n$  Iterationen sind das höchstens

$$n \cdot (4n) = 4n^2 \text{ Elementaroperationen. } \square$$

Tatsächlich bräuchten wir sogar nur  $\approx 3n^2$  Elem.-operationen.  
Warum?

# Exkurs: Quadratische vs. Lineare Laufzeit

---

## Addition

Laufzeit (#Elem.-operationen):  $n$

# Exkurs: Quadratische vs. Lineare Laufzeit

---

## Addition

Laufzeit (#Elem.-operationen):  $n$

*lineare Laufzeit*

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	<i>quadratische Laufzeit</i>

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	<i>quadratische Laufzeit</i>
Wenn wir die Anzahl Stellen um den Faktor 2 vergrößern, dann vergrößert sich die Laufzeit um den Faktor...		

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	<i>quadratische Laufzeit</i>
Wenn wir die Anzahl Stellen um den Faktor 2 vergrößern, dann vergrößert sich die Laufzeit um den Faktor...		2

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	<i>quadratische Laufzeit</i>
Wenn wir die Anzahl Stellen um den Faktor 2 vergrößern, dann vergrößert sich die Laufzeit um den Faktor...	2	4

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	<i>quadratische Laufzeit</i>
Wenn wir die Anzahl Stellen um den Faktor 2 vergrößern, dann vergrößert sich die Laufzeit um den Faktor...	2	4

→ je größer die Eingabe, desto größer der Unterschied

# Exkurs: Quadratische vs. Lineare Laufzeit

---

	Addition	Multiplikation
Laufzeit (#Elem.-operationen):	$n$	$\leq 4n^2$
	<i>lineare Laufzeit</i>	<i>quadratische Laufzeit</i>
Wenn wir die Anzahl Stellen um den Faktor 2 vergrößern, dann vergrößert sich die Laufzeit um den Faktor...	2	4

- je größer die Eingabe, desto größer der Unterschied
- wir interessieren uns hauptsächlich für das **asymptotische Wachstum**

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

# Algorithmische Probleme

---

Eingabe →  → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

2. Beispiel: **Multiplikation** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x \cdot y$

# Algorithmische Probleme

---

Eingabe → ? → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

2. Beispiel: **Multiplikation** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x \cdot y$

→  $4n^2$  elementare Operationen ✓

# Algorithmische Probleme

---

Eingabe → ? → Ausgabe

1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

2. Beispiel: **Multiplikation** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x \cdot y$

→  $4n^2$  elementare Operationen ✓

Geht das noch besser?

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**:

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$a_{n-1}$

...

$a_0$

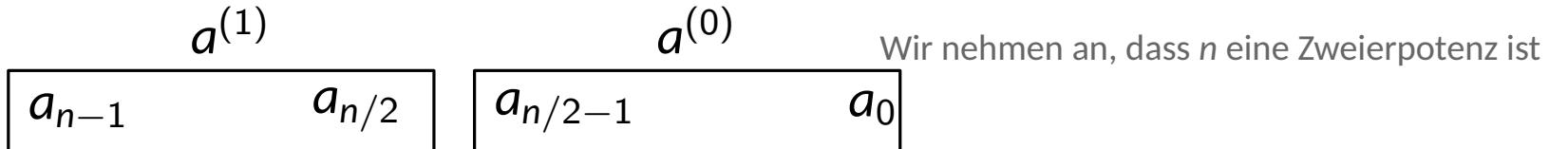
# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:



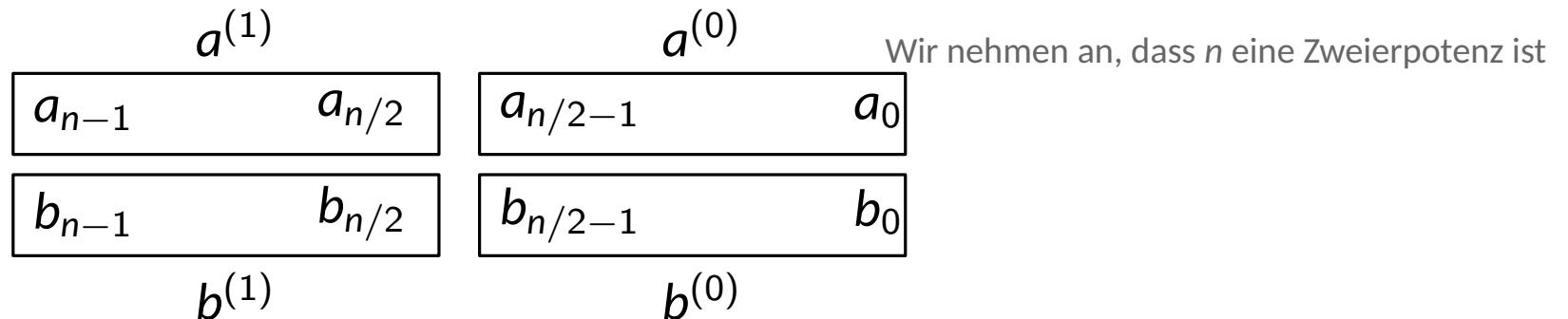
# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:



Wir nehmen an, dass  $n$  eine Zweierpotenz ist

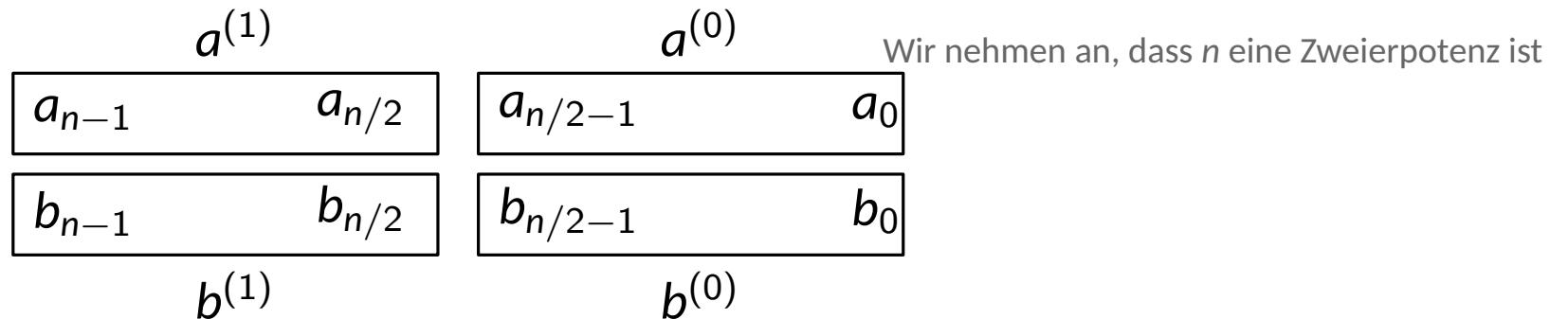
# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:



1. Teile: Bestimme die  $\frac{n}{2}$ -steligen Teile  $a^{(1)}, a^{(0)}$  und  $b^{(1)}, b^{(0)}$
- $$a = a^{(1)}B^{n/2} + a^{(0)}$$
- $$b = b^{(1)}B^{n/2} + b^{(0)}$$

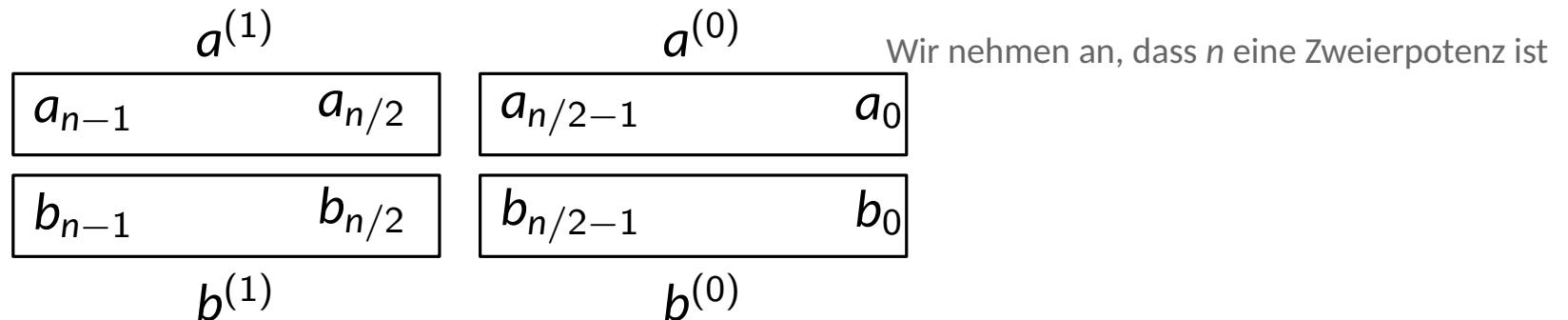
# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:



1. Teile: Bestimme die  $\frac{n}{2}$ -steligen Teile  $a^{(1)}, a^{(0)}$  und  $b^{(1)}, b^{(0)}$   
$$a = a^{(1)}B^{n/2} + a^{(0)}$$
$$b = b^{(1)}B^{n/2} + b^{(0)}$$

Bestimme  $a^{(0)} \cdot b^{(0)}, a^{(0)} \cdot b^{(1)}, a^{(1)} \cdot b^{(0)}$  und  $a^{(1)} \cdot b^{(1)}$  **rekursiv**

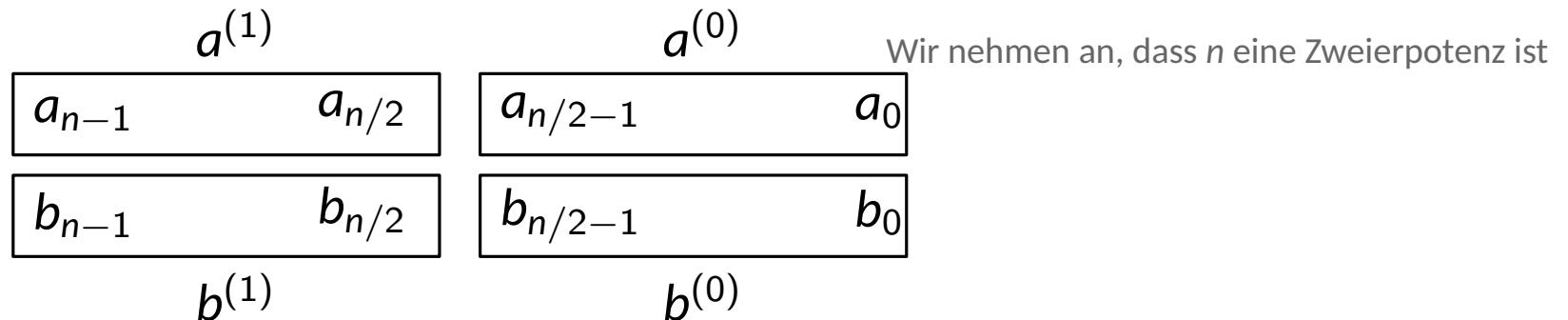
# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:



1. Teile: Bestimme die  $\frac{n}{2}$ -steligen Teile  $a^{(1)}, a^{(0)}$  und  $b^{(1)}, b^{(0)}$   
$$a = a^{(1)}B^{n/2} + a^{(0)}$$
$$b = b^{(1)}B^{n/2} + b^{(0)}$$

Bestimme  $a^{(0)} \cdot b^{(0)}, a^{(0)} \cdot b^{(1)}, a^{(1)} \cdot b^{(0)}$  und  $a^{(1)} \cdot b^{(1)}$  **rekursiv**

2. Herrsche: Bestimme das Ergebnis  $a \cdot b$ :

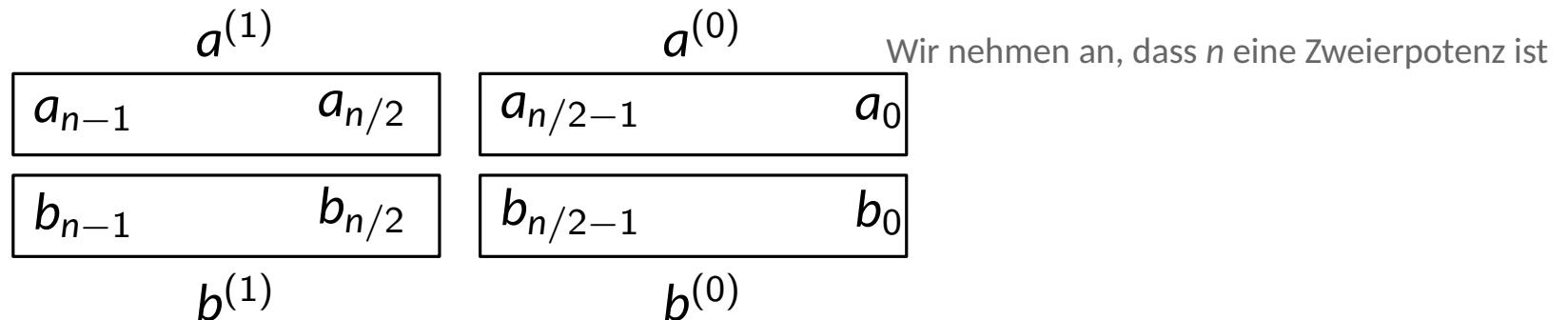
# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:



1. Teile: Bestimme die  $\frac{n}{2}$ -stelligen Teile  $a^{(1)}, a^{(0)}$  und  $b^{(1)}, b^{(0)}$   
$$a = a^{(1)}B^{n/2} + a^{(0)}$$
$$b = b^{(1)}B^{n/2} + b^{(0)}$$

Bestimme  $a^{(0)} \cdot b^{(0)}, a^{(0)} \cdot b^{(1)}, a^{(1)} \cdot b^{(0)}$  und  $a^{(1)} \cdot b^{(1)}$  **rekursiv**

2. Herrsche: Bestimme das Ergebnis  $a \cdot b$ :

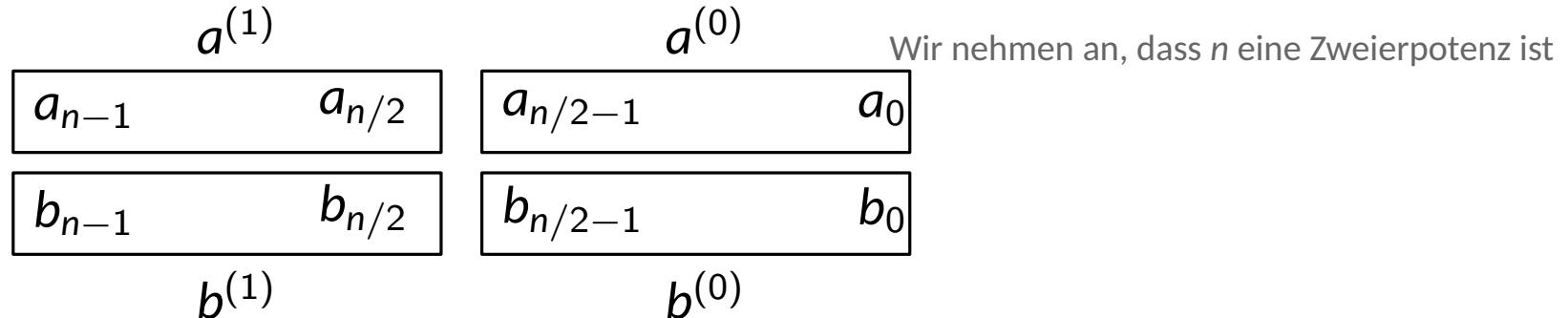
$$a \cdot b = a^{(1)} \cdot b^{(1)}B^n + a^{(1)} \cdot b^{(0)}B^{n/2} + a^{(0)} \cdot b^{(1)}B^{n/2} + a^{(0)} \cdot b^{(0)}$$

# Karatsuba's Algorithmus

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

**Kurzeinführung zu Karatsuba's Algorithmus:** Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

## Idee:



1. Teile: Bestimme die  $\frac{n}{2}$ -st gigen Teile  $a^{(1)}, a^{(0)}$  und  $b^{(1)}, b^{(0)}$

Bestimme  $a^{(0)} \cdot b^{(0)}$ ,  $a^{(0)} \cdot b^{(1)}$ ,  $a^{(1)} \cdot b^{(0)}$  und  $a^{(1)} \cdot b^{(1)}$  rekursiv

2. Herrsche: Bestimme das Ergebnis  $a \cdot b$ :

$$a \cdot b = a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)}$$

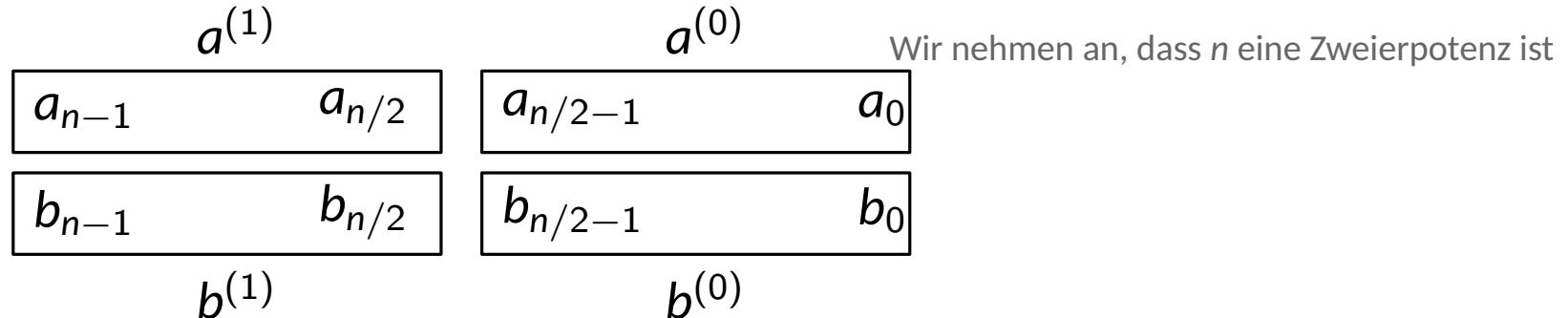
→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +

# Karatsuba's Algorithmus

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

**Kurzeinführung zu Karatsuba's Algorithmus:** Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

## Idee:



1. Teile: Bestimme die  $\frac{n}{2}$ -steligen Teile  $a^{(1)}, a^{(0)}$  und  $b^{(1)}, b^{(0)}$

Bestimme  $a^{(0)} \cdot b^{(0)}$ ,  $a^{(0)} \cdot b^{(1)}$ ,  $a^{(1)} \cdot b^{(0)}$  und  $a^{(1)} \cdot b^{(1)}$  rekursiv

2. Herrsche: Bestimme das Ergebnis  $a \cdot b$ :

$$a \cdot b = a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$a \cdot b = a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$a \cdot b = a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$a \cdot b = a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= a^{(1)} \cdot b^{(1)} B^n + (a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}) B^{n/2} + a^{(0)} \cdot b^{(0)} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= a^{(1)} \cdot b^{(1)} B^n + (a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}) B^{n/2} + a^{(0)} \cdot b^{(0)} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

$$(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)}) = a^{(1)}b^{(1)} + a^{(1)}b^{(0)} + a^{(0)}b^{(1)} + a^{(0)}b^{(0)}$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= a^{(1)} \cdot b^{(1)} B^n + (a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}) B^{n/2} + \underline{\underline{a^{(0)} \cdot b^{(0)}}} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

$$(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)}) = a^{(1)}b^{(1)} + \underline{\underline{a^{(1)}b^{(0)} + a^{(0)}b^{(1)} + a^{(0)}b^{(0)}}}$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= \underbrace{a^{(1)} \cdot b^{(1)} B^n}_{\beta} + (a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}) B^{n/2} + \underbrace{a^{(0)} \cdot b^{(0)}}_{\alpha} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

$$(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)}) = \underbrace{a^{(1)}b^{(1)}}_{\beta} + \underbrace{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}_{\alpha} + \underbrace{a^{(0)}b^{(0)}}_{\alpha}$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= \underbrace{a^{(1)} \cdot b^{(1)} B^n}_{\beta} + (a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}) B^{n/2} + \underbrace{a^{(0)} \cdot b^{(0)}}_{\alpha} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \underbrace{a^{(1)}b^{(1)}}_{\beta} + \color{orange}{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}} + \underbrace{a^{(0)}b^{(0)}}_{\alpha}$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= \underbrace{a^{(1)} \cdot b^{(1)}}_{\beta} B^n + \underbrace{(a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)})}_{\delta} B^{n/2} + \underbrace{a^{(0)} \cdot b^{(0)}}_{\alpha} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \underbrace{a^{(1)}b^{(1)}}_{\beta} + \underbrace{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}_{\delta} + \underbrace{a^{(0)}b^{(0)}}_{\alpha}$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**: Hinweis: Dies ist ein Appetithäppchen für Inhalte, die wir lernen werden.

Idee:

$$\begin{aligned} a \cdot b &= a^{(1)} \cdot b^{(1)} B^n + a^{(1)} \cdot b^{(0)} B^{n/2} + a^{(0)} \cdot b^{(1)} B^{n/2} + a^{(0)} \cdot b^{(0)} \\ &= \underbrace{a^{(1)} \cdot b^{(1)}}_{\beta} B^n + \underbrace{(a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)})}_{\delta} B^{n/2} + \underbrace{a^{(0)} \cdot b^{(0)}}_{\alpha} \end{aligned}$$

→ 4 (rekursive) Multiplikationen von  $n/2$ -steligen Zahlen +  
3 Additionen von  $n$ -steligen Zahlen

Wir werden sehen: die 4 rekursiven Multiplikationen sind **teuer**

Erstaunlicher Trick: 3 rekursive Multiplikationen reichen!

$$\begin{aligned} \underbrace{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}_{\gamma} &= \underbrace{a^{(1)}b^{(1)}}_{\beta} + \underbrace{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}_{\delta} + \underbrace{a^{(0)}b^{(0)}}_{\alpha} \\ \rightarrow \delta &= \gamma - \alpha - \beta \end{aligned}$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**:

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \underline{\overline{a^{(1)}b^{(1)}}} + \underline{\overline{a^{(1)}b^{(0)}}} + \underline{\overline{a^{(0)}b^{(1)}}} + \underline{\overline{a^{(0)}b^{(0)}}}$$
$$\rightarrow \delta = \gamma - \alpha - \beta$$

# Karatsuba's Algorithmus

---

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu **Karatsuba's Algorithmus**:

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \frac{a^{(1)}b^{(1)}}{\beta} + \frac{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}{\delta} + \frac{a^{(0)}b^{(0)}}{\alpha}$$
$$\rightarrow \delta = \gamma - \alpha - \beta$$

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$  //Elementaroperation

$a^{(1)} \leftarrow a_{n-1}, \dots, a_{n/2}, a^{(0)} \leftarrow a_{n/2-1}, \dots, a_0$

$b^{(1)} \leftarrow b_{n-1}, \dots, b_{n/2}, b^{(0)} \leftarrow b_{n/2-1}, \dots, b_0$

$\alpha \leftarrow \text{Karatsuba}(a^{(0)}, b^{(0)})$

$\beta \leftarrow \text{Karatsuba}(a^{(1)}, b^{(1)})$

$\gamma \leftarrow \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$

$\delta \leftarrow \gamma - \alpha - \beta$

**return**  $\beta \cdot B^n + \delta B^{n/2} + \alpha$

letzte Folie:  $a \cdot b = a^{(1)}b^{(1)}B^n + (a^{(1)}b^{(0)} + a^{(0)}b^{(1)})B^{n/2} + a^{(0)}b^{(0)}$

# Karatsuba's Algorithmus

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu Karatsuba's Algorithmus:

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \frac{a^{(1)}b^{(1)}}{\beta} + \frac{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}{\delta} + \frac{a^{(0)}b^{(0)}}{\alpha}$$
$$\rightarrow \delta = \gamma - \alpha - \beta$$

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$  //Elementaroperation

$a^{(1)} \leftarrow a_{n-1}, \dots, a_{n/2}, a^{(0)} \leftarrow a_{n/2-1}, \dots, a_0$

$b^{(1)} \leftarrow b_{n-1}, \dots, b_{n/2}, b^{(0)} \leftarrow b_{n/2-1}, \dots, b_0$

$\alpha \leftarrow \text{Karatsuba}(a^{(0)}, b^{(0)})$

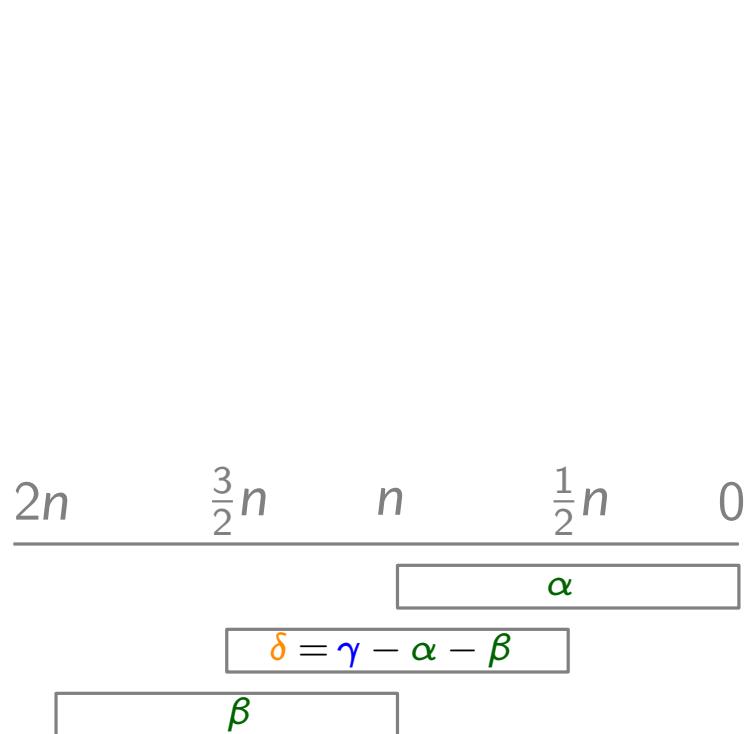
$\beta \leftarrow \text{Karatsuba}(a^{(1)}, b^{(1)})$

$\gamma \leftarrow \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$

$\delta \leftarrow \gamma - \alpha - \beta$

**return**  $\beta \cdot B^n + \delta B^{n/2} + \alpha$

letzte Folie:  $a \cdot b = a^{(1)}b^{(1)}B^n + (a^{(1)}b^{(0)} + a^{(0)}b^{(1)})B^{n/2} + a^{(0)}b^{(0)}$



# Karatsuba's Algorithmus

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu Karatsuba's Algorithmus:

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \frac{a^{(1)}b^{(1)}}{\beta} + \frac{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}{\delta} + \frac{a^{(0)}b^{(0)}}{\alpha}$$
$$\rightarrow \delta = \gamma - \alpha - \beta$$

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$  //Elementaroperation

$a^{(1)} \leftarrow a_{n-1}, \dots, a_{n/2}, a^{(0)} \leftarrow a_{n/2-1}, \dots, a_0$

$b^{(1)} \leftarrow b_{n-1}, \dots, b_{n/2}, b^{(0)} \leftarrow b_{n/2-1}, \dots, b_0$

$\alpha \leftarrow \text{Karatsuba}(a^{(0)}, b^{(0)})$

$\beta \leftarrow \text{Karatsuba}(a^{(1)}, b^{(1)})$

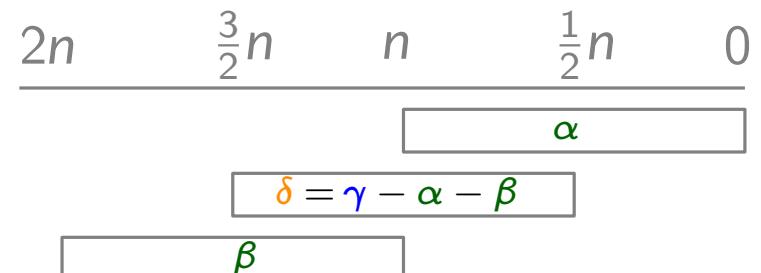
$\gamma \leftarrow \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$

$\delta \leftarrow \gamma - \alpha - \beta$

**return**  $\beta \cdot B^n + \delta B^{n/2} + \alpha$

letzte Folie:  $a \cdot b = a^{(1)}b^{(1)}B^n + (a^{(1)}b^{(0)} + a^{(0)}b^{(1)})B^{n/2} + a^{(0)}b^{(0)}$

Wir werden lernen, solche Algorithmen zu analysieren



# Karatsuba's Algorithmus

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu Karatsuba's Algorithmus:

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \frac{a^{(1)}b^{(1)}}{\beta} + \frac{a^{(1)}b^{(0)} + a^{(0)}b^{(1)}}{\delta} + \frac{a^{(0)}b^{(0)}}{\alpha}$$
$$\rightarrow \delta = \gamma - \alpha - \beta$$

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$  //Elementaroperation

$a^{(1)} \leftarrow a_{n-1}, \dots, a_{n/2}, a^{(0)} \leftarrow a_{n/2-1}, \dots, a_0$

$b^{(1)} \leftarrow b_{n-1}, \dots, b_{n/2}, b^{(0)} \leftarrow b_{n/2-1}, \dots, b_0$

$\alpha \leftarrow \text{Karatsuba}(a^{(0)}, b^{(0)})$

$\beta \leftarrow \text{Karatsuba}(a^{(1)}, b^{(1)})$

$\gamma \leftarrow \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$

$\delta \leftarrow \gamma - \alpha - \beta$

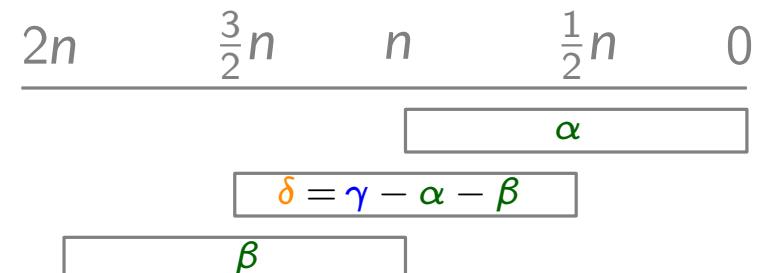
**return**  $\beta \cdot B^n + \delta B^{n/2} + \alpha$

letzte Folie:  $a \cdot b = a^{(1)}b^{(1)}B^n + (a^{(1)}b^{(0)} + a^{(0)}b^{(1)})B^{n/2} + a^{(0)}b^{(0)}$

Wir werden lernen, solche Algorithmen zu analysieren

## Theorem

Karatsuba() multipliziert zwei  $n$ -stellige Zahlen mit höchstens  $153n^{\log 3}$  Elementaroperationen.



# Karatsuba's Algorithmus

Ja, wir können einen wesentlich schnelleren Algorithmus entwickeln!

Kurzeinführung zu Karatsuba's Algorithmus:

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

$$\frac{(a^{(1)} + a^{(0)})(b^{(1)} + b^{(0)})}{\gamma} = \underline{a^{(1)}b^{(1)}} + \underline{\color{orange}a^{(1)}b^{(0)} + a^{(0)}b^{(1)}} + \underline{a^{(0)}b^{(0)}} \quad \rightarrow \delta = \gamma - \alpha - \beta$$

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$  //Elementaroperation

$a^{(1)} \leftarrow a_{n-1}, \dots, a_{n/2}, a^{(0)} \leftarrow a_{n/2-1}, \dots, a_0$

$b^{(1)} \leftarrow b_{n-1}, \dots, b_{n/2}, b^{(0)} \leftarrow b_{n/2-1}, \dots, b_0$

$\alpha \leftarrow \text{Karatsuba}(a^{(0)}, b^{(0)})$

$\beta \leftarrow \text{Karatsuba}(a^{(1)}, b^{(1)})$

$\gamma \leftarrow \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$

$\delta \leftarrow \gamma - \alpha - \beta$

**return**  $\beta \cdot B^n + \delta B^{n/2} + \alpha$

letzte Folie:  $a \cdot b = a^{(1)}b^{(1)}B^n + (a^{(1)}b^{(0)} + a^{(0)}b^{(1)})B^{n/2} + a^{(0)}b^{(0)}$

Wir werden lernen, solche Algorithmen zu analysieren

## Theorem

Karatsuba() multipliziert zwei  $n$ -stellige Zahlen mit höchstens  $153n^{\log 3}$  Elementaroperationen.

$$n^{\log 3} \approx n^{1.585}$$

$2n$	$\frac{3}{2}n$	$n$	$\frac{1}{2}n$	$0$
------	----------------	-----	----------------	-----

$\alpha$

$$\delta = \gamma - \alpha - \beta$$

$\beta$

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

**Grundbegriffe:**

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

**Grundbegriffe:**

- Was ist ein algorithmisches Problem?

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

**Grundbegriffe:**

- Was ist ein algorithmisches Problem?
- Was ist ein Algorithmus und wie beschreibt man ihn?

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

**Grundbegriffe:**

- Was ist ein algorithmisches Problem?
- Was ist ein Algorithmus und wie beschreibt man ihn?
- Was ist die Laufzeit und wie analysiert man sie?

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

## **Grundbegriffe:**

- Was ist ein algorithmisches Problem?
- Was ist ein Algorithmus und wie beschreibt man ihn?
- Was ist die Laufzeit und wie analysiert man sie?
- Mithilfe einfacher Bausteine lassen sich komplexe Algorithmen entwerfen

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

## **Grundbegriffe:**

- Was ist ein algorithmisches Problem?
- Was ist ein Algorithmus und wie beschreibt man ihn?
- Was ist die Laufzeit und wie analysiert man sie?
- Mithilfe einfacher Bausteine lassen sich komplexe Algorithmen entwerfen
- Es gibt nützliche Entwurfstechniken (z.B. "Teile-und-Herrsche")

# Zusammenfassung

---

Wir haben gesehen (im Schnelldurchlauf):

## **Grundbegriffe:**

- Was ist ein algorithmisches Problem?
- Was ist ein Algorithmus und wie beschreibt man ihn?
- Was ist die Laufzeit und wie analysiert man sie?
- Mithilfe einfacher Bausteine lassen sich komplexe Algorithmen entwerfen
- Es gibt nützliche Entwurfstechniken (z.B. "Teile-und-Herrsche")

## **Fragen?**

Algorithmen und Datenstrukturen SS'23

# Kapitel 2: Grundbegriffe

Marvin Künemann

AG Algorithmen & Komplexität

# Eine kurze Bitte vorab

---

Bitte überprüfen Sie, ob Sie alle möglichen Übungstermine angegeben haben!

**Anmeldungsfrist ist heute!**

Eingabe → ? → Ausgabe

### 1. Beispiel: **Addition** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x + y$

→  $n$  elementare Operationen ✓

### 2. Beispiel: **Multiplikation** natürlicher Zahlen

Eingabe:  $x, y \in \mathbb{N}$

Ausgabe:  $x \cdot y$

→  $4n^2$  elementare Operationen ✓

Geht das noch besser?

**Karatsubas Algorithmus:**

(ohne Analyse)

→ höchstens  $153 \cdot n^{1.585}$  elementare Operationen ✓

# Kapitelüberblick

---

Letztes Kapitel: Beispielhafte Einführung in die Algorithmitik

Jetzt: Gründliche Definition der Grundbegriffe

Insbesondere klären wir in diesem Kapitel:

- Lernziele
- Einführung der **Grundbegriffe**:
  - Algorithmus
  - Rechenmodell
  - Pseudocode
  - Laufzeit
  - Korrektheit

# Algorithmus: Definition

---

## Algorithmenbegriff

Ein Algorithmus ist eine **eindeutige Handlungsvorschrift** zur Lösung eines algorithmischen Problems.

Vergleichbare Handlungsvorschriften finden sich in vielen Bereichen des Lebens:

Bereich	Beschreibung durch...	"gelöstes Problem"
Kochen	Rezept	gegeben Zutaten, produziere Gericht
Musik	Noten	spiele Musikstück
Zusammensetzung demokratisches Parlament	Gesetzestext	gegeben Stimmen für jede Partei, ermittle Sitzverteilung

# Warum dieser Kurs wichtig ist...

---

## Bundeswahlgesetz

### §6 Wahl nach Landeslisten

- (1) Für die Verteilung der nach Landeslisten zu besetzenden Sitze werden die für jede Landesliste abgegebenen Zweitstimmen zusammengezählt. Nicht berücksichtigt werden dabei die Zweitstimmen derjenigen Wähler, die ihre Erststimme für einen im Wahlkreis erfolgreichen Bewerber abgegeben haben, der gemäß §20 Absatz 3 oder von einer Partei vorgeschlagen ist, die nach Absatz 3 bei der Sitzverteilung nicht berücksichtigt wird oder für die in dem betreffenden Land keine Landesliste zugelassen ist. Von der Gesamtzahl der Abgeordneten (§1 Absatz 1) wird die Zahl der erfolgreichen Wahlkreisbewerber abgezogen, die in Satz 2 genannt sind.
- (2) In einer ersten Verteilung wird zunächst die Gesamtzahl der Sitze (§1 Absatz 1) in dem in Satz 2 bis 7 beschriebenen Berechnungsverfahren den Ländern nach deren Bevölkerungsanteil (§3 Absatz 1) und sodann in jedem Land die Zahl der dort nach Absatz 1 Satz 3 verbleibenden Sitze auf der Grundlage der zu berücksichtigenden Zweitstimmen den Landeslisten zugeordnet. Jede Landesliste erhält so viele Sitze, wie sich nach Teilung der Summe ihrer erhaltenen Zweitstimmen durch einen Zuteilungsdivisor ergeben. Zahlenbruchteile unter 0,5 werden auf die darunter liegende ganze Zahl abgerundet, solche über 0,5 werden auf die darüber liegende ganze Zahl aufgerundet. Zahlenbruchteile, die gleich 0,5 sind, werden so aufgerundet oder abgerundet, dass die Zahl der zu vergebenden Sitze eingehalten wird; ergeben sich dabei mehrere mögliche Sitzzuweisungen, so entscheidet das vom Bundeswahlleiter zu ziehende Los. Der Zuteilungsdivisor ist so zu bestimmen, dass insgesamt so viele Sitze auf die Landeslisten entfallen, wie Sitze zu vergeben sind. Dazu wird zunächst die Gesamtzahl der Zweitstimmen aller zu berücksichtigenden Landeslisten durch die Zahl der jeweils nach Absatz 1 Satz 3 verbleibenden Sitze geteilt. Entfallen danach mehr Sitze auf die Landeslisten, als Sitze zu vergeben sind, ist der Zuteilungsdivisor so heraufzusetzen, dass sich bei der Berechnung die zu vergebende Sitzzahl ergibt; entfallen zu wenig Sitze auf die Landeslisten, ist der Zuteilungsdivisor entsprechend herunterzusetzen.
- (3) Bei Verteilung der Sitze auf die Landeslisten werden nur Parteien berücksichtigt, die mindestens 5 Prozent der im Wahlgebiet abgegebenen gültigen Zweitstimmen erhalten oder in mindestens drei Wahlkreisen einen Sitz errungen haben. Satz 1 findet auf die von Parteien nationaler Minderheiten eingereichten Listen keine Anwendung.
- (4) Von der für jede Landesliste so ermittelten Sitzzahl wird die Zahl der von der Partei in den Wahlkreisen des Landes errungenen Sitze (§5) abgerechnet. In den Wahlkreisen errungene Sitze verbleiben einer Partei auch dann, wenn sie die nach den Absätzen 2 und 3 ermittelte Zahl übersteigen.

# Algorithmus: Definition

---

## Algorithmenbegriff

Ein Algorithmus ist eine **eindeutige Handlungsvorschrift** zur Lösung eines algorithmischen Problems.

Vergleichbare Handlungsvorschriften finden sich in vielen Bereichen des Lebens:

Bereich	Beschreibung durch...	"gelöstes Problem"
Kochen	Rezept	gegeben Zutaten, produziere Gericht
Musik	Noten	spiele Musikstück
Zusammensetzung demokratisches Parlament	Gesetzestext	gegeben Stimmen für jede Partei, ermittle Sitzverteilung

In dieser Vorlesung interessieren wir uns für Probleme, die von einem Computer gelöst werden können.

# Lernziele

---



## Ziele der Vorlesung sind:

- Kenntis & Verständnis grundlegender Algorithmen und Datenstrukturen
- Algorithmen beschreiben, analysieren und bewerten können
- Neue, korrekte, gute Algorithmen für vorher unbekannte Probleme entwerfen können
- Formal korrekt argumentieren können
- Mit anderen über Algorithmen reden können

## Warum ist das wichtig?

- nötig, um Algorithmen entwickeln und auch anwenden zu können
- Verständnis dafür, welche Probleme/Anwendungen wir effizient lösen können
- Vorlesung fördert strukturiertes Problemlösen

# Thematische Schwerpunkte

---

Eingabe →  → Ausgabe

## Grundbegriffe und Handwerkszeug

- Was ist ein Algorithmus und wie analysiere ich ihn?
- Korrektheit und asymptotische Laufzeitschranken
- formale Methoden und Beweistechniken

## Algorithmen und Datenstrukturen

- Elementare Datenstrukturen
- Suchen, Sortieren, Wörterbücher
- Spezielle Strukturen: Listen, Graphen

## Entwurfsmethoden

- Komplexe Algorithmen aus einfachen Algorithmen aufbauen
- Standardtechniken für bestimmte Problemklassen

## Theorie

- Komplexitätstheorie: Grenzen der effizienten Berechenbarkeit

# Berechnungsmodelle

---

Wir müssen klären, was genau unser Berechnungsmodell ist

→ Welche Anweisungen sind erlaubt? Wie geben wir diese an?

Mit einem eindeutig definierten Berechnungsmodell können wir **formal genau und korrekt** arbeiten.

Es existieren verschiedene Berechnungsmodelle:

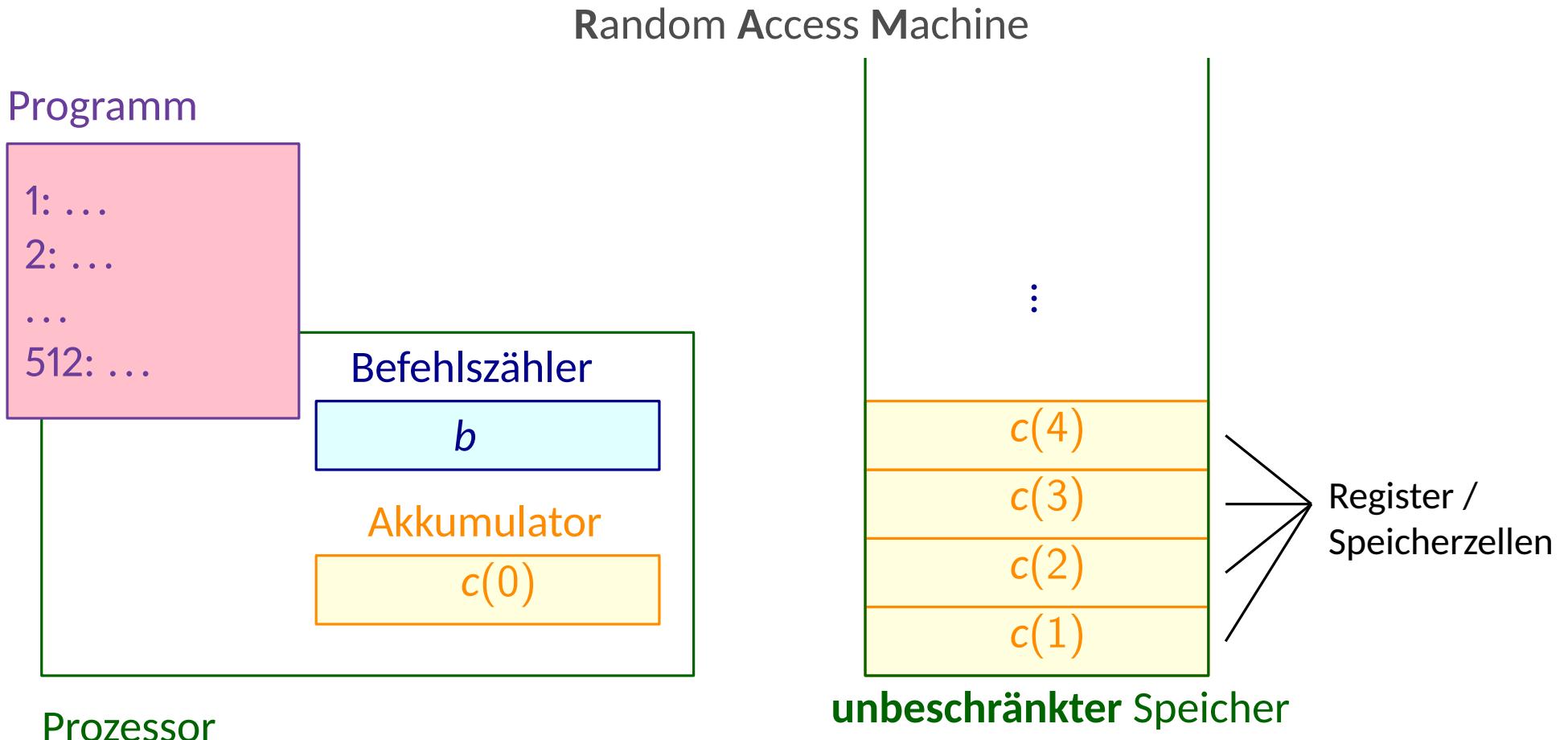
- Registermaschinen (RAM) ← in dieser VL standardmäßig verwendet
- Turingmaschinen ← am Ende dieser VL verwendet
- $\lambda$ -Kalkül
- While-Programme (und Loop-Programme)
- Grammatiken
- ...

Man könnte auch ein Berechnungsmodell mittels üblicher Programmiersprachen definieren

- Java, C++, Pascal, etc.
- Haskell, etc.

# Registermaschine (RAM)

Hinweis: in der Literatur finden sich unterschiedliche Definitionen



## Prozessor

- Prozessor führt Instruktionen des Programms aus
- Befehlszähler gibt aktuelle Instruktion im Programm an
- Daten werden im Akkumulator  $c(0)$  sowie den Registern  $c(1), c(2), \dots$  gespeichert
- Akkumulator und Register speichern jeweils eine (beschränkte) natürliche Zahl  $c(i) \in \{0, \dots, 2^w - 1\}$
- Instruktionen können Akkumulator, Befehlszähler und Register verändern

realistisch?

Nein, aber wichtig als theoretisches Modell!

# Instruktionen der Registermaschine (RAM)

---

- Laden/Speichern

LOAD $i$ :	$c(0) \leftarrow c(i)$	$b \leftarrow b + 1$
STORE $i$ :	$c(i) \leftarrow c(0)$	$b \leftarrow b + 1$
CLOAD $x$ :	$c(0) \leftarrow x$	$b \leftarrow b + 1$

- Einfache Operationen:

ADD $i$ :	$c(0) \leftarrow c(0) + c(i)$	$b \leftarrow b + 1$
CADD $x$ :	$c(0) \leftarrow c(0) + x$	$b \leftarrow b + 1$
SUB $i$ :	$c(0) \leftarrow \max\{c(0) - c(i), 0\}$	$b \leftarrow b + 1$

...

ähnlich für Multiplikationen (MULT) und ganzzahlige Division (DIV)

- (bedingte) Sprünge:

GOTO $j$ :	$b \leftarrow j$
------------	------------------

IF $c(0) = x$ GOTO $j$ :
--------------------------

$$b \leftarrow \begin{cases} j & \text{wenn } c(0) = x \\ b + 1 & \text{ansonsten} \end{cases}$$

- Indirekte Adressierung:

Für die meisten obigen Operationen gibt es eine Variante mit indirekter Adressierung

Hierbei gibt man statt dem gewünschten Register  $i$  ein Register  $\alpha$  an, dass die gewünschte Registernummer  $i$  enthält

INDLOAD $\alpha$ :	$c(0) \leftarrow c(c(\alpha))$	$b \leftarrow b + 1$
--------------------	--------------------------------	----------------------

INDADD $\alpha$ :	$c(0) \leftarrow c(0) + c(c(\alpha))$	$b \leftarrow b + 1$
-------------------	---------------------------------------	----------------------

... ähnlich für INDSTORE, INDMULT, INDSUB, etc.

# Algorithmische Probleme auf der RAM

Wie berechnet nun eine Registermaschine ein algorithmisches Problem?

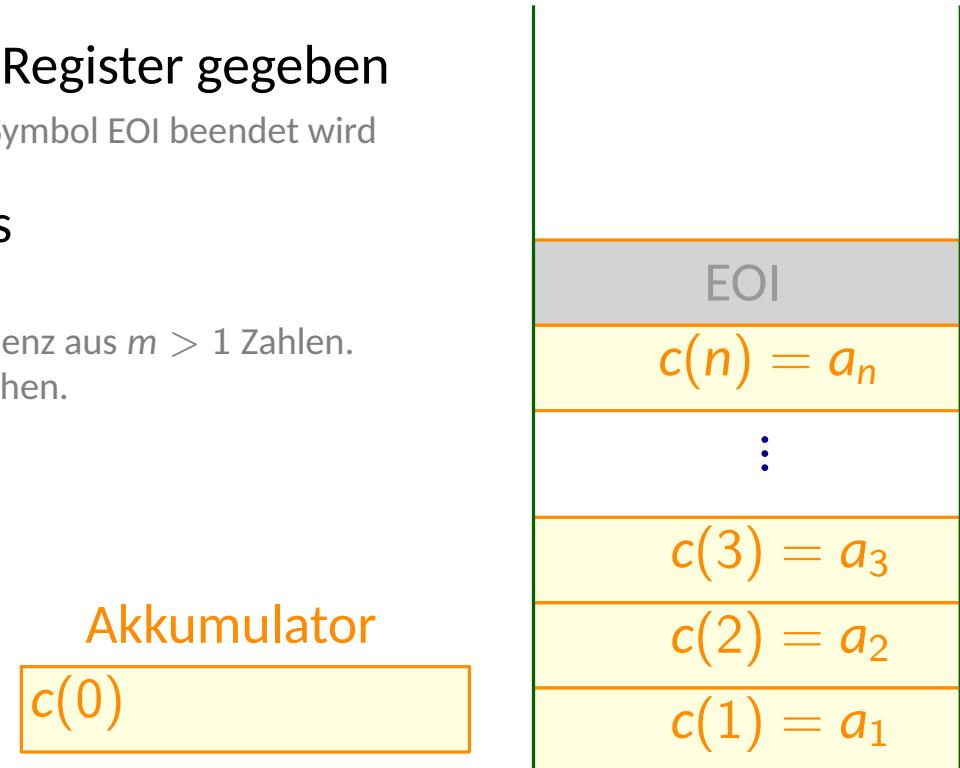
Die Eingabe  $a_1 \dots a_n$  ist als Inhalt der ersten Register gegeben

Man darf sich vorstellen, dass die Eingabe durch ein spezielles Symbol EOI beendet wird

Die Ausgabe ist der Inhalt des Akkumulators  
nach Ende des Programms.

Manche algorithmischen Problem haben als Ergebnis eine Sequenz aus  $m > 1$  Zahlen.  
In diesem Fall soll die Ausgabe in den Registern  $c(1) \dots c(m)$  stehen.

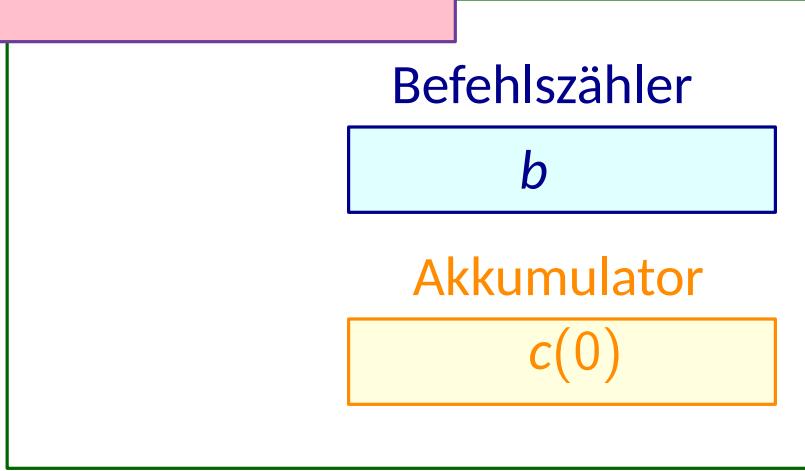
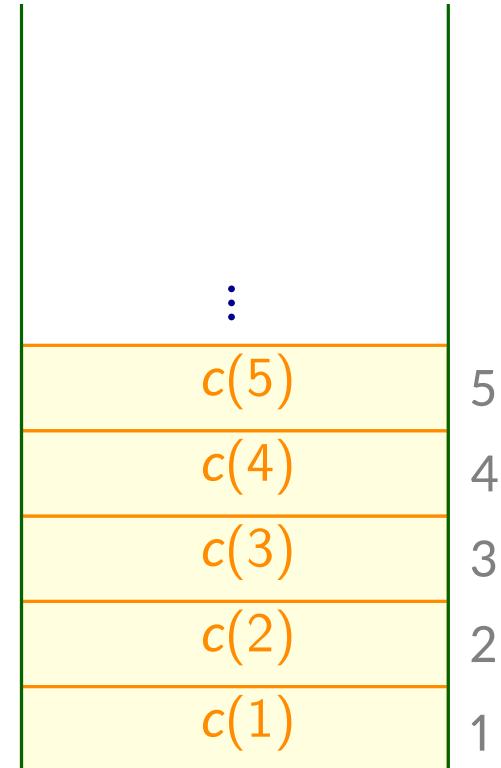
Wir nennen  $n$  die **Eingabegröße**.



# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

:	
0	5
0	4
0	3
3	2
2	1

Befehlszähler

Akkumulator

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

Eingabe: 2,3

:		
0	5	
0	4	
0	3	
3	2	
2	1	

Befehlszähler

Akkumulator

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

:	
0	5
0	4
0	3
3	2
2	1

Befehlszähler

Akkumulator

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

:	
0	5
0	4
0	3
3	2
2	1

Befehlszähler

Akkumulator

0

# Beispiel einer RAM-Berechnung

Programm



```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

:	
0	5
0	4
0	3
3	2
2	1

Befehlszähler

1

Akkumulator

0

# Beispiel einer RAM-Berechnung

Programm



```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

⋮		
0	5	
0	4	
0	3	
3	2	
2	1	

Befehlszähler

1

Akkumulator

1

1

# Beispiel einer RAM-Berechnung

Programm



```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

:	
0	5
0	4
0	3
3	2
2	1

Befehlszähler

2

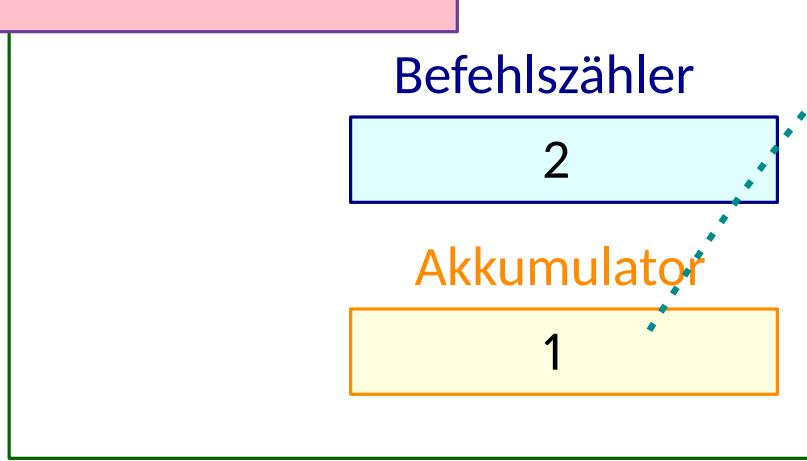
Akkumulator

1

# Beispiel einer RAM-Berechnung

Programm

```
→ 1: CLOAD 1
    2: STORE 3
    3: LOAD 2
    4: If c(0) = 0 THEN GOTO 11
    5: CSUB 1
    6: STORE 2
    7: LOAD 3
    8: MULT 1
    9: STORE 3
   10: GOTO 3
   11: LOAD 3
   12: END
```



⋮		
0	5	
0	4	
1	3	
3	2	
2	1	

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
1	3
3	2
2	1

Befehlszähler

3

Akkumulator

1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



⋮		
0	5	
0	4	
1	3	
3	2	
2	1	

Befehlszähler

3

Akkumulator

3

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
1	2
3	1
2	1

Befehlszähler

4

Akkumulator

3

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
1	2
3	1
2	1

Befehlszähler

4

Akkumulator

3

? = 0

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
1	2
3	1
2	1

Befehlszähler

4

Akkumulator

3

? = 0 Nein!

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
1	3
3	2
2	1

Befehlszähler

5

Akkumulator

3

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
1	3
3	2
2	1

Befehlszähler

5

Akkumulator

x 2

- 1



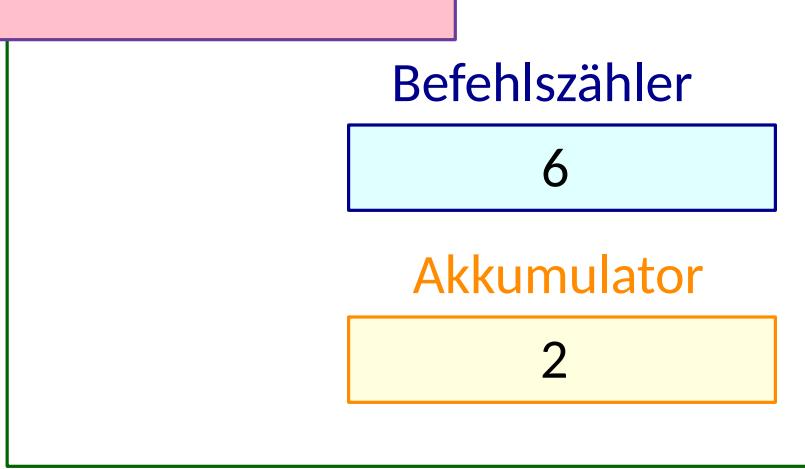
# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```

→

⋮	5
0	4
0	3
1	2
3	1
2	1



# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



Befehlszähler

6

Akkumulator

2

⋮		
0		5
0		4
1		3
2		2
2		1

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:		
0		5
0		4
1		3
2		2
2		1

Befehlszähler

7

Akkumulator

2

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
1	2
2	2
2	1

Befehlszähler

7

Akkumulator

1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
1	3
2	2
2	1

Befehlszähler

8

Akkumulator

1

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0	5	
0	4	
1	3	
2	2	
2	1	

Befehlszähler

8

Akkumulator

× 2

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
1	3
2	2
2	1

Befehlszähler

9

Akkumulator

2

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

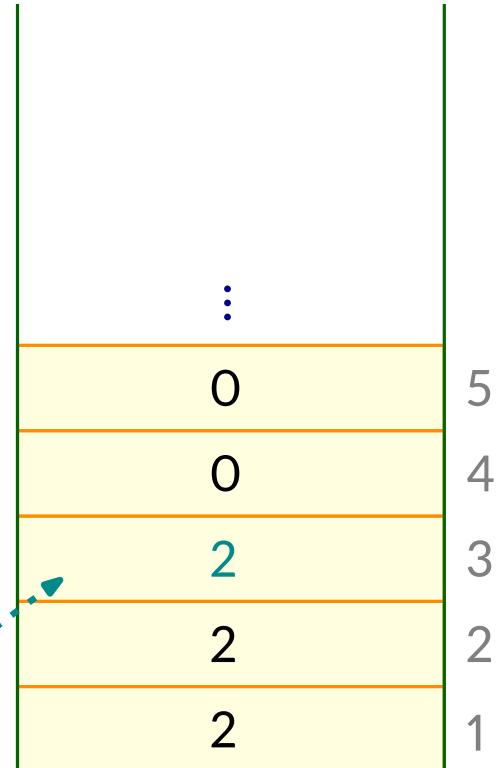


Befehlszähler

9

Akkumulator

2



# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:		
0	5	
0	4	
2	3	
2	2	
2	1	

Befehlszähler

10

Akkumulator

2

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0	5	
0	4	
2	3	
2	2	
2	1	

Befehlszähler

3

Akkumulator

2

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



⋮		
0	5	
0	4	
2	3	
2	2	
2	1	

Befehlszähler

3

Akkumulator

2

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
2	2
2	1

Befehlszähler

4

Akkumulator

2

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
2	2
2	1

Befehlszähler

4

Akkumulator

2

?

=

0

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
2	2
2	1

Befehlszähler

4

Akkumulator

2

? = 0 Nein!

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
2	3
2	2
2	1

Befehlszähler

5

Akkumulator

2

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
2	3
2	2
2	1

Befehlszähler

5

Akkumulator

$\times$  1

- 1

# Beispiel einer RAM-Berechnung

## Programm



```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

⋮		
0	5	
0	4	
2	3	
2	2	
2	1	

Befehlszähler

6

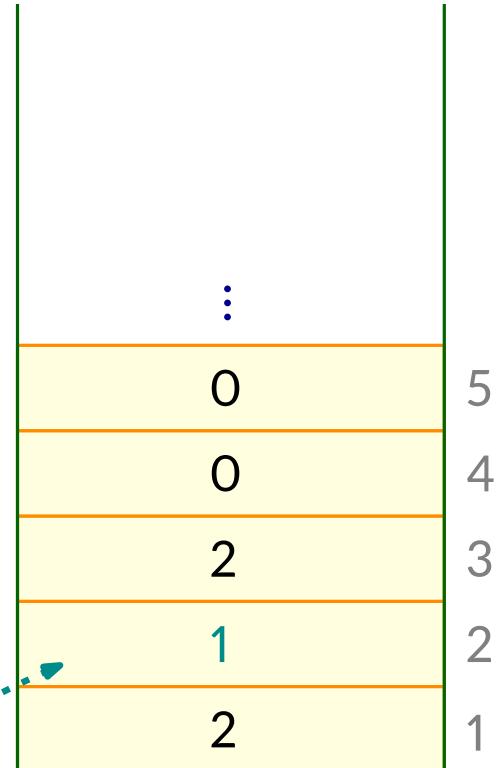
Akkumulator

1

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
2	3
1	2
2	1

Befehlszähler

7

Akkumulator

1

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



Befehlszähler

7

Akkumulator

2

⋮	5
0	4
0	3
2	2
1	2
2	1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
2	3
1	2
2	1

Befehlszähler

8

Akkumulator

2

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0		5
0		4
2		3
1		2
2		1

Befehlszähler

8

Akkumulator

4

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:		
0	5	5
0	4	4
2	3	3
1	2	2
2	1	1

Befehlszähler

9

Akkumulator

4

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

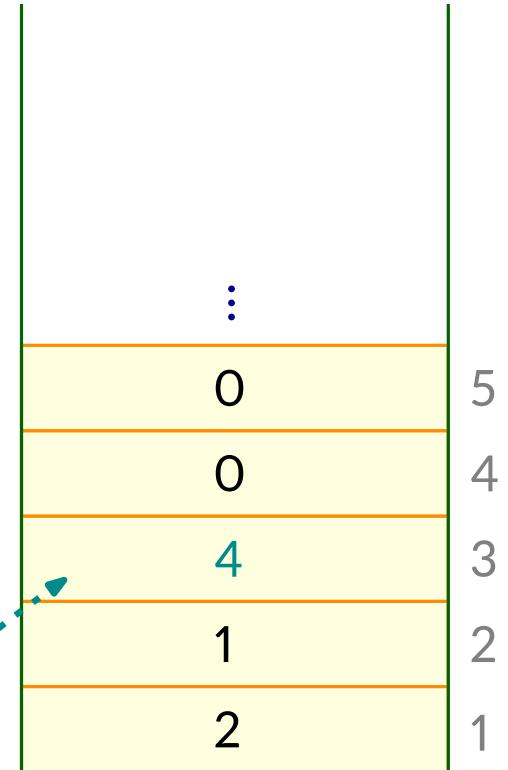


Befehlszähler

9

Akkumulator

4



# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

→

⋮	5
0	4
0	3
4	2
1	2
2	1

Befehlszähler

10

Akkumulator

4

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
4	3
1	2
2	1

Befehlszähler

3

Akkumulator

4

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



⋮		
0		5
0		4
4		3
1		2
2		1

Befehlszähler

3

Akkumulator

1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:		
0		5
0		4
4		3
1		2
2		1

Befehlszähler

4

Akkumulator

1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
4	2
1	2
2	1

Befehlszähler

4

Akkumulator

1

? = 0

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
4	2
1	2
2	1

Befehlszähler

4

Akkumulator

1

? = 0 Nein!

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
4	3
1	2
2	1

Befehlszähler

5

Akkumulator

1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
4	3
1	2
2	1

Befehlszähler

5

Akkumulator

x 0

-1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



⋮		
0		5
0		4
4		3
1		2
2		1

Befehlszähler

6

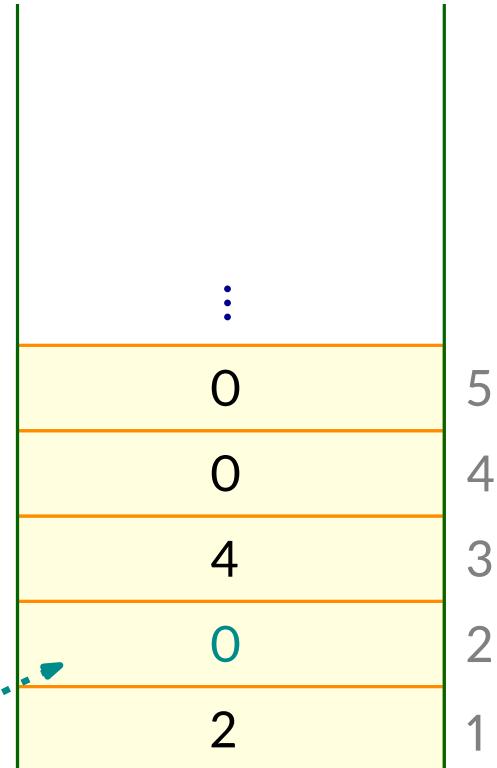
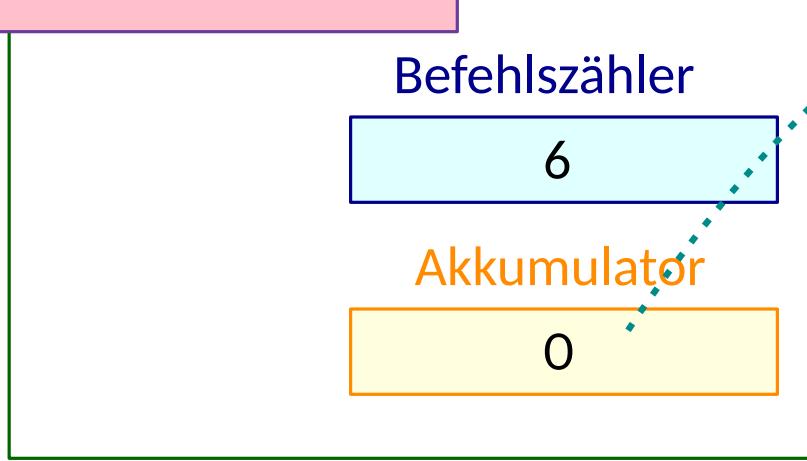
Akkumulator

0

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
4	3
0	2
2	1

Befehlszähler

7

Akkumulator

0

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0		5
0		4
4		3
0		2
2		1

Befehlszähler

7

Akkumulator

4

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0		5
0		4
4		3
0		2
2		1

Befehlszähler

8

Akkumulator

4

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



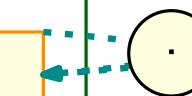
⋮		
0		5
0		4
4		3
0		2
2		1

Befehlszähler

8

Akkumulator

4 8



# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0		5
0		4
4		3
0		2
2		1

Befehlszähler

9

Akkumulator

8

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



Befehlszähler

9

Akkumulator

8

⋮		
0		5
0		4
8		3
0		2
2		1

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

→

⋮	5
0	4
0	3
8	2
0	1
2	

Befehlszähler

10

Akkumulator

8

# Beispiel einer RAM-Berechnung

Programm



```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

⋮		
0	5	
0	4	
8	3	
0	2	
2	1	

Befehlszähler

3

Akkumulator

8

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1  
2: STORE 3  
3: LOAD 2  
4: If c(0) = 0 THEN GOTO 11  
5: CSUB 1  
6: STORE 2  
7: LOAD 3  
8: MULT 1  
9: STORE 3  
10: GOTO 3  
11: LOAD 3  
12: END
```



⋮		
0		5
0		4
8		3
0		2
2		1

Befehlszähler

3

Akkumulator

0

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
8	2
0	1
2	

Befehlszähler

4

Akkumulator

0

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
8	2
0	1
2	

Befehlszähler

4

Akkumulator

0

? = 0

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮	5
0	4
0	3
8	2
0	1
2	

Befehlszähler

4

Akkumulator

0

? = 0 Ja!

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:		
0		5
0		4
8		3
0		2
2		1

Befehlszähler

11

Akkumulator

0

# Beispiel einer RAM-Berechnung

## Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



Befehlszähler

11

Akkumulator

8

⋮		
0		5
0		4
8		3
0		2
2		1

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



⋮		
0		5
0		4
8		3
0		2
2		1

Befehlszähler

12

Akkumulator

8

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



:	
0	5
0	4
8	3
0	2
2	1

Befehlszähler

12

Akkumulator

8

→ Ausgabe der Berechnung!

# Beispiel einer RAM-Berechnung

Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: If c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```



**Frage:**  
Was berechnet  
das Programm?

:	
0	5
0	4
8	3
0	2
2	1

Befehlszähler

12

Akkumulator

8

→ Ausgabe der Berechnung!

# Def. Berechnete Funktion

Eine RAM berechnet eine Funktion  $f : \mathbb{N}^* \rightarrow \mathbb{N} \cup \{\perp\}$

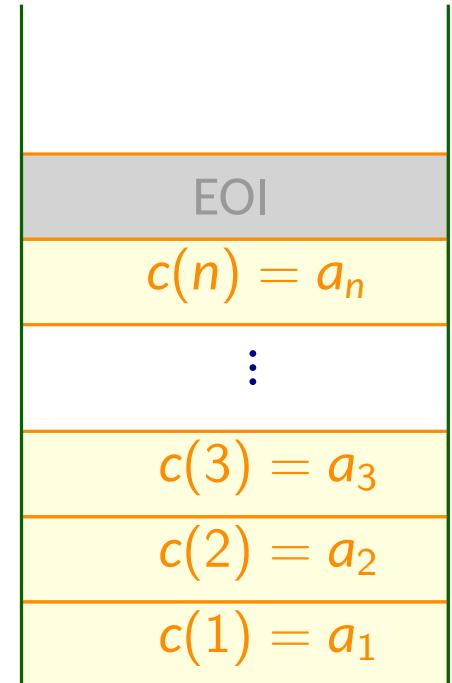
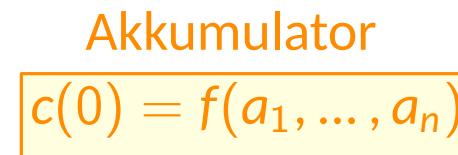
endliche Folgen natürlicher Zahlen  
 $\hat{=}$  Eingaben  $(a_1, \dots, a_n)$

Ausgabe, wobei  $\perp$  bedeutet, dass die Registermaschine nicht hält.

## Definition

Für eine RAM A ist die von A berechnete Funktion  $f : \mathbb{N}^* \rightarrow \mathbb{N} \cup \{\perp\}$  gegeben durch

$$f(a_1, \dots, a_n) = \begin{cases} \perp & \text{wenn } A \text{ auf } a_1, \dots, a_n \text{ nicht hält,} \\ y & \text{wenn } A \text{ mit Inhalt } y \text{ im Akkumulator hält.} \end{cases}$$



## Bemerkung:

Die gleiche Funktion  $f$  kann durch viele verschiedene Algorithmen (= RAM-Programme) berechnet werden.

Notiz: Das Beispielprogramm berechnet  $f : (a_1, a_2, \dots, a_n) \mapsto a_1^{a_2}$

# Def. Laufzeit auf der RAM

---

Prinzipieller Ansatz: Wir zählen einfach die Berechnungsschritte der RAM

Im Detail unterscheidet man aber zwischen:

→ **Uniformes Kostenmaß**: Jede ausgeführte Instruktion kostet eine Zeiteinheit.

Sehr hilfreiches Modell, kann aber unrealistisch werden, wenn die Zahlen in den Registern riesig sind

→ **Logarithmisches Kostenmaß**: Jede ausgeführte Instruktion kostet  $\log_2 x$ , wobei  $x$  die größte Zahl der durch die Instruktion verwendeten Register ist.

Dieses Modell betrachtet auch die Bitlänge der Registerinhalte. Beispiel: Hier kostet "ADD 3" 10 Zeitschritte.  
In manchen Fällen bietet dieses ein realistischeres Modell.

$$c(0) = 2^{10} = 1024$$

$1000000000_2$

$$c(3) = 2^5 = 32$$

$100000_2$  :

Unser **Standardmodell**:

Typischerweise benutzen wir das uniforme Kostenmaß, mit einer zusätzlichen Annahme:  
jedes Register kann nur Zahlen bis  $n^c$  speichern

Hierbei ist  $n$  die Eingabegröße und  $c \geq 1$  eine Zahl, die als beliebig große Konstante gesetzt werden kann.

**Hinweis:** Selbst mit dieser Annahme können wir alle Eingaberegister indirekt adressieren.

# Diskussion: Registermaschine

---

Ist die Registermaschine ein "gutes" theoretisches Modell?

## Pro:

Die Laufzeit auf der RAM ähnelt häufig der tatsächlich gemessenen Laufzeit von Implementierungen

Registermaschinen sind genauso "mächtig" wie die wichtigsten Berechnungsmodelle (insbesondere Turingmaschinen)

## Contra:

zu ungenau (einfach)

zu genau (komplex)

wichtige Merkmale moderner Computer sind nicht abgebildet:

- Speicherhierarchie (verschiedene Cache-Level etc.)
- Parallelisierung
- Randomisierung

Man kann die RAM um solche Elemente erweitern!

↗ diese VL  
+ weiterführende VL'en

# Unser Laufzeitmaß als Gütekriterium

---

Wie bemessen wir die **Laufzeit** zur Bewertung eines Algorithmus'?

Wir haben die Laufzeit als Zeitkosten auf der RAM definiert.

Warum benutzen wir nicht die gemessene Laufzeit (z.B. in ms) auf einem realen Computer?

**Viele Nachteile:**

- Laufzeit stark plattformabhängig
- Laufzeit kann selbst auf der gleichen Plattform stark schwanken  
(Prozessortemperatur, andere gleichzeitig ausgeführte Prozesse, etc.)
- schwierig zu beschreiben
- müsste experimentell bestimmt werden
- experimentelle Evaluation von Algorithmen ist ein Gebiet für sich

Unsere Definition erlaubt eine mathematisch saubere **Theorie** von Algorithmen

Diese Theorie ist ein zusätzliches Werkzeug, neben experimenteller Evaluation

# Def. Laufzeit, Teil II

## Definition

RAM-Programm

Berechnungsschritte auf der RAM (nach gewähltem Kostenmaß)

Für einen Algorithmus A bezeichnet  $t_A(x)$  die Laufzeit von A auf Eingabe x.

Manchmal zählen wir nur bestimmte elementare Operationen (↗ VL1: Langzahlarithmetik)

Beispiel: In unserem Beispielprogramm mit Eingabe  $x = (a_1, a_2, \dots, a_n)$  ergibt sich:

$$t_A(x) = 8a_2 + 5$$

Erläuterung: Wenn  $a_2 = 0$ , so führt A genau 5 Instruktionen aus

Jeder Schleifendurchlauf besteht aus 8 Instruktionen

Es gibt  $a_2$  Schleifendurchläufe

Wir werden die Laufzeit in Abhängigkeit der **Eingabegröße** betrachten

Intuitiv: Je größer die Eingabe, desto länger die Berechnung.

Die interessante Frage ist: **Wieviel** länger?

RAM-Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: IF c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10:GOTO 3
11: LOAD 3
12:END
```

# Laufzeit in Abhängigkeit der Eingabegröße

---

**Erinnerung:** Wir haben die Eingabegröße definiert als Anzahl an (Eingabe-)Registern

Wichtiger Hinweis: Hierzu gibt es auch Alternativen, z.B. die gesamte Bitlänge der Eingabe.

Die tatsächliche Laufzeit hängt aber auch vom genauen Inhalt ab

**Definition:**

Für  $n \in \mathbb{N}$  sei  $I_n$  die Menge der **Eingaben (Inputs)** der Größe  $n$ .

**Definition:**

Für einen Algorithmus A ist die

- **worst-case-Laufzeit** die Funktion  $T$  mit  $T(n) = \max\{t_A(x) \mid x \in I_n\}$
- **best-case-Laufzeit** die Funktion  $T$  mit  $T(n) = \min\{t_A(x) \mid x \in I_n\}$
- **uniforme average-case-Laufzeit** die Funktion  $T$  mit  $T(n) = \frac{\sum_{x \in I_n} t_A(x)}{|I_n|}$

Die average-case-Laufzeit ist viel allgemeiner definiert.

Mit Abstand am Wichtigsten ist die worst-case-Laufzeit!

Wir haben bereits Schranken für den worst-case gesehen:

Schulmethode für die Addition:

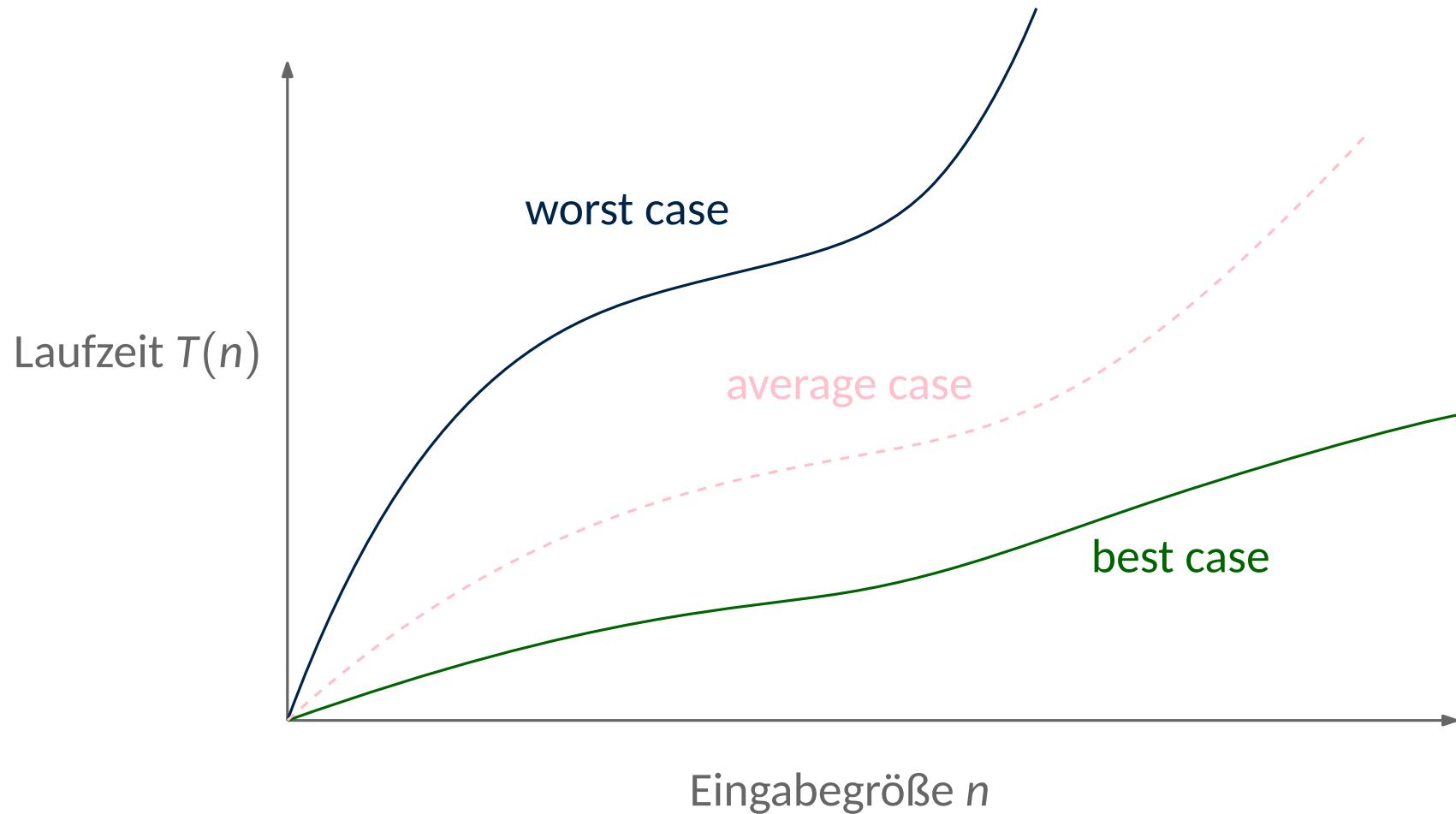
$n$  Elementaroperationen

Schulmethode für die Multiplikation:

$\leq 4n^2$  Elementaroperationen

# Visualisierung: worst/best/average case

---

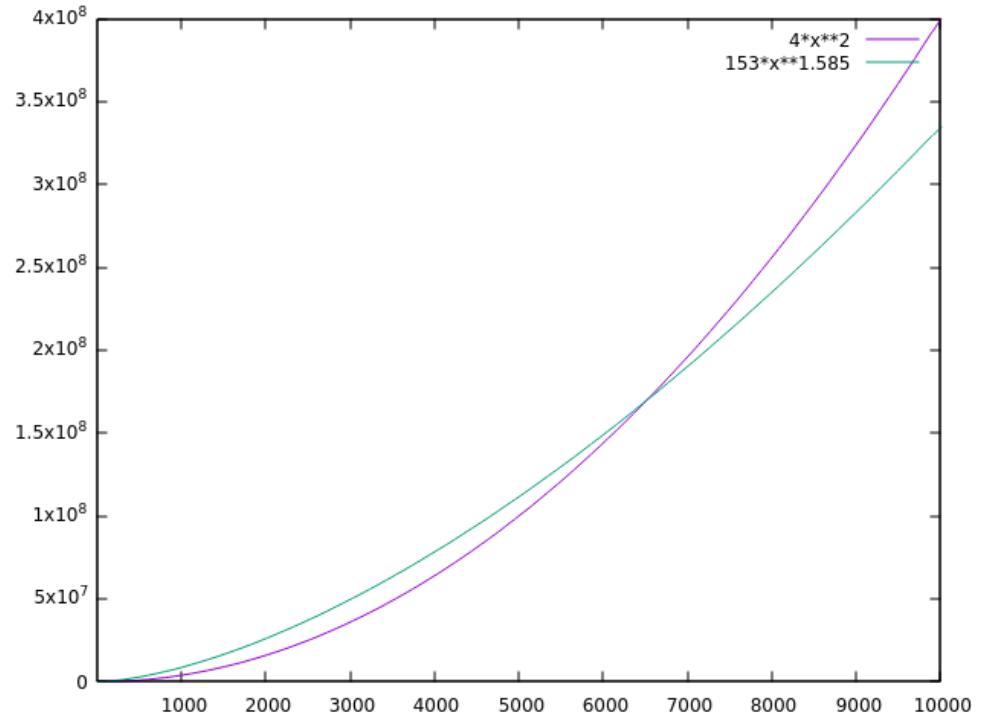


# Asymptotische Laufzeiten

Welche Laufzeit ist besser?

$$4n^2$$

$$153n^{1.585}$$



- für  $n \geq 6510$  ist  $153n^{1.585}$  kleiner      unendlich viele  $n$  ("fast alle")  
→ für  $n \leq 6509$  ist  $4n^2$  kleiner      nur kleine  $n$

**Grundsätzlich** interessieren wir uns vor allem für das Verhalten auf großen Eingaben

Das ist in der Praxis **nicht immer** genau das Gewünschte, ist aber als grundsätzlicher Ansatz sehr wertvoll

von besonderem Interesse: das **asymptotische Wachstum** der Laufzeiten

# Achtung:

Es folgt **sehr wichtiger** Inhalt!

# O-Notation

# O-Notation (Landau-Notation)

---

↗ siehe Tafelpräsentation

## Definition

Für jedes  $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ , definieren wir

1.  $O(f) := \{g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \exists c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
2.  $\Omega(f) := \{g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \exists c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$
3.  $\Theta(f) := O(f) \cap \Omega(f)$
4.  $o(f) := \{g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \forall c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
5.  $\omega(f) := \{g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \forall c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

# O-Notation (Landau-Notation)

---

$g \in O(f)$ bedeutet:	"ab einer gewissen Eingabegröße ist $g$ höchstens um einen konstanten Faktor größer als $f$ "	" $\leq$ "
$g \in \Omega(f)$ bedeutet:	$g$ wächst asymptotisch mindestens so schnell wie $f$	" $\geq$ "
$g \in \Theta(f)$ bedeutet:	$g$ wächst asymptotisch genau so schnell wie $f$	" $=$ "
$g \in o(f)$ bedeutet:	$g$ wächst asymptotisch langsamer als $f$	" $<$ "
$g \in \omega(f)$ bedeutet:	$g$ wächst asymptotisch schneller als $f$	" $>$ "

**Hinweis:** In der Literatur betreibt man **Notationsmissbrauch**:

man schreibt in der Regel  $g = O(f), g = \Theta(f), g = \omega(f)$ , etc.

↗ siehe spätere Folie

# O-Notation: Beispiele

---

$$10n \in O(n)$$

$$5n^2 \in O(n^2)$$

$$1000000n \in O(n)$$

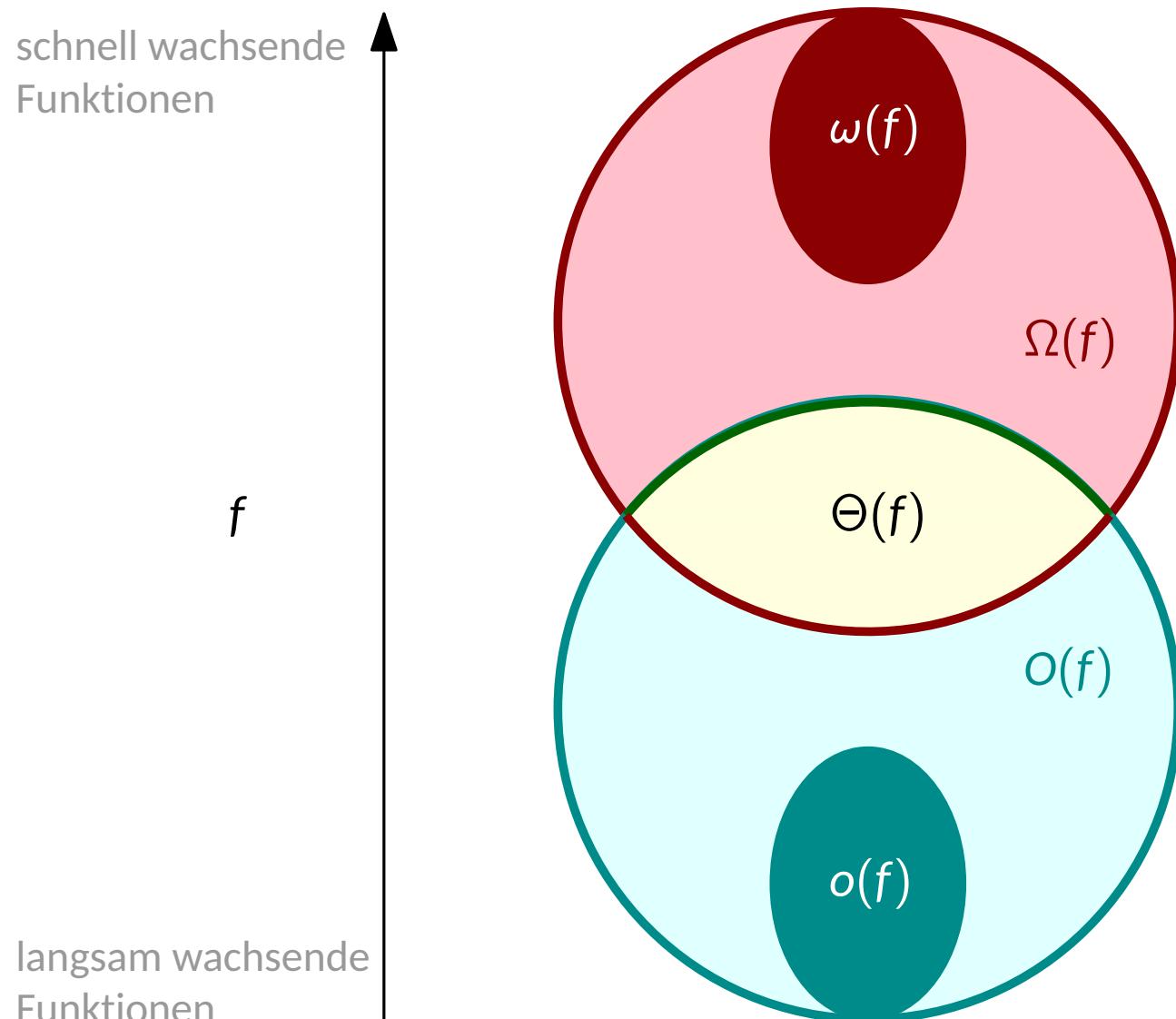
$$n^2 - 3n \in O(n^2)$$

$$0.05n \in O(n)$$

$$n \in O(n^2)$$

# O-Notation: Illustration

---



# Grenzwert-Lemma

→ siehe Tafelpräsentation

## Lemma

**Wenn** der Grenzwert  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)}$  existiert und Wert  $c \in \mathbb{R}_{\geq 0} \cup \{\infty\}$  hat, dann gilt

1. Wenn  $c = 0$ , dann ist  $g \in o(f)$
  2. Wenn  $c = \infty$ , dann ist  $g \in \omega(f)$
  3. Wenn  $0 < c < \infty$ , dann ist  $g \in \Theta(f)$

Aus der Kombination der vorherigen Fälle ergibt sich:

4. Wenn  $0 \leq c < \infty$ , dann ist  $g \in O(f)$   
 5. Wenn  $0 < c \leq \infty$ , dann ist  $g \in \Omega(f)$

Achtung: Um dieses Lemma anzuwenden, muss sichergestellt werden, dass  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)}$  existiert.

# Weitere Bemerkungen

---

Man verwendet die O-Notation auch häufig in komplexeren Ausdrücken

Für eine Funktion  $h : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ , sei

$$h(O(f)) := \bigcup_{g \in O(f)} \{h \circ g\}$$
$$\rightarrow 2 \cdot O(n^2) = O(n^2)$$
$$\rightarrow 2^{O(n)} = \bigcup_{c=1}^{\infty} O(c^n)$$

Analoge Definitionen gelten für Summe, Multiplikationen, etc. von  $O(\cdot)$ -Termen:

$$O(f) + O(h) = \{g_f + g_h \mid g_f \in O(f), g_h \in O(h)\}$$

analog für  $O(f) \cdot O(g)$ , etc.

Weiterhin werden wir O-Notation auch für mehrere Variablen verwenden:

z.B.  $T = O(n \log n + m)$  bedeutet:

Es gibt eine Konstante  $c$  sodass für genügend große Eingaben  $T$  durch  $c \cdot (n \log n + m)$  beschränkt ist

Gelegentlich schreiben wir auch für eine Menge  $\mathcal{F}$  von Funktionen:

$$O(\mathcal{F}) := \bigcup_{f \in \mathcal{F}} O(f) \quad \rightarrow \quad O(O(f)) = O(f)$$

# Eigenschaften der O-Notation

---

**Lemma** Für alle Funktionen  $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$  gilt:

1.  $f \in O(f), f \in \Theta(f), f \in \Omega(f)$  (Reflexivität von  $O, \Theta, \Omega$ )
  2.  $f \notin o(f), f \notin \omega(f)$  (Irreflexivität von  $o, \omega$ )
  3. wenn  $f \in O(g)$  und  $g \in O(h)$ , dann  $f \in O(h)$  (Transitivität von  $O$ )
- Es gilt auch die Transitivität von  $\Omega, \Theta, o$  und  $\omega$
4.  $f \in O(g) \Leftrightarrow g \in \Omega(f)$  (Schiefsymmetrie von  $O, \Omega$ )
  - $f \in o(g) \Leftrightarrow g \in \omega(f)$  (Schiefsymmetrie von  $o, \omega$ )
  - $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$  (Symmetrie von  $\Theta$ )
5. Für jede Konstante  $c \in \mathbb{R}_{>0}$  ist  
 $O(c \cdot f) = O(f)$
  6.  $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$
  7.  $O(f) \cdot O(g) = O(f \cdot g)$

# Notationsmissbrauch

Die meisten AlgorithmikerInnen schreiben " $=$ " anstelle von " $\in$ " bei der Benutzung der O-Notation:

z.B.  $n^2 + 200n = O(n^2)$  anstelle von  $n^2 + 200n \in O(n^2)$

Missverständnisse können insbesondere entstehen, wenn  $\subseteq$  durch  $=$  ersetzt wird

z.B.  $n^2 + 200n = O(n^2) = O(n^3)$  anstelle von  $n^2 + 200n \in O(n^2) \subseteq O(n^3)$

Dieses " $=$ " muss "von links nach rechts" gelesen werden  
Es gilt schließlich nicht, dass  $O(n^3) \subseteq O(n^2)$ .

Gefährlich ist auch die Nutzung von O-Notation außerhalb geschlossener Formen:

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

**Interpretation 1:**

$$\{\sum_{i=1}^n g(n) \mid g \in O(n)\}$$

$$\subseteq O(n^2)$$

**Interpretation 2:**

$$\underbrace{O(n) + O(n) + \dots + O(n)}_{n \text{ mal}}$$

→ konstante Faktoren in jedem Term könnten sich unterscheiden?

**nicht zu empfehlen**

Missverständliche Notation werden wir vermeiden.

# Pseudocode

# Pseudocode

## RAM-Programm

```
1: CLOAD 1
2: STORE 3
3: LOAD 2
4: IF c(0) = 0 THEN GOTO 11
5: CSUB 1
6: STORE 2
7: LOAD 3
8: MULT 1
9: STORE 3
10: GOTO 3
11: LOAD 3
12: END
```

Man kann Algorithmen in unterschiedlichem Detailgrad angeben  
→ Wir benutzen hauptsächlich einen sehr abstrakten Pseudocode

## Java-ähnlicher Pseudocode

```
public static int power(int a , int b) {
    int r = 1;
    while (b > 0) {
        r = a * r ;
        b--; // b wird dekrementiert
    }
    return r;
}
```

## etwas abstrakterer Pseudocode

```
power(a, b)
r = 1
for i = 1, ..., b do
    r = a · r
return r
```

# Pseudocode II

Wir benutzen in dieser Vorlesung in der Regel den Java-ähnlichen oder den abstrakteren Pseudocode

→ Zweck: übersichtliche Beschreibung von Algorithmen

Details zum Java-ähnlichen Pseudocode finden sich im ↗ OLAT Materialordner (erstellt von Prof. Garth)

- wenig feste Konventionen
  - benutzt “typische” Schleifenkonstrukte und Operationen
  - werden Notation bei Bedarf klären
- Beschreibung sollte immer eindeutig sein und sich leicht in beliebiger Programmiersprache umsetzen lassen

## Java-ähnlicher Pseudocode

```
public static int power(int a , int b) {  
    int r = 1;  
    while (b > 0) {  
        r = a * r ;  
        b--; // b wird dekrementiert  
    }  
    return r ;  
}
```

## etwas abstrakterer Pseudocode

```
power(a, b)  
|   r = 1  
|   for i = 1, ..., b do  
|       |   r = a · r  
|   return r
```

# Grundlegende Strategien zur Laufzeitanalyse

---

## Sequentielle Ausführung

P:

1 P1  
2 P2

$$T(P1) + T(P2)$$

## If-Bedingungen

P:

1 **if** (C) **then**  
2 | P1  
3 **else**  
4 | P2

$$O(T(C) + \max\{T(P1), T(P2)\})$$

## einfache For-Schleife

P:

1 **for**  $i = a, \dots, b$  **do**  
2 | P1

Überprüfung der Abbruchbedingung  
/

$$O(1) + \sum_{x=a}^b T(x)$$

P:  
1 **for** ( $i=a; i \leq b; i++$ ) {  
2 | P1  
3 }

wobei  $T(x)$  die Laufzeit von P1 für den Fall  $i = x$  bezeichnet

# Korrektheit

# Korrektheitsbeweise

---

Ein wichtiger Teil der Vorlesung ist es, Korrektheit von Algorithmen zu beweisen

**Grund:** Das ist bei weitem nicht immer offensichtlich!

```
power(a, b)
|   r = 1
|   for i = 1, ..., b do
|       |   r = a · r
|   return r
```

```
fastPower(a, b)
|   r = 1, p = a
|   while (b > 0) do
|       |   if (b mod 2 ≠ 0) do
|           |       |   b = b - 1
|           |       |   r = r · p
|       |   else
|           |       |   b = b/2
|           |       |   p = p · p
|   return r
```

Klar: berechnet  $a^b$

Weniger offensichtlich: berechnet auch  $a^b$

Es ist nicht einmal sofort offensichtlich, dass fastPower immer terminiert!

**Korrektheitsbeweis:** Beweis, dass der angegebene Algorithmus immer die gewünschte Ausgabe liefert

- Terminierung
- gewünschtes Ergebnis

# Rückblick

---

Letztes Kapitel: Beispielhafte Einführung in die Algorithmitik

Jetzt: Gründliche Definition der Grundbegriffe

Insbesondere klären wir in diesem Kapitel:

- Lernziele
- Einführung der **Grundbegriffe**:
  - Algorithmus
  - Rechenmodell      ← heute im Detail besprochen, in der Zukunft viel impliziter
  - Pseudocode        ← ab jetzt unser Standard
  - Laufzeit           ← wir werden von jetzt an O-Notation ausgiebig verwenden
  - Korrektheit

## Algorithmen und Datenstrukturen SS'23

# Kapitel 3: Sortieren

Marvin Künemann

AG Algorithmen & Komplexität

# Quiz: O-Notation

---

Wahr oder falsch?

1.  $n^2 + 10^6n \in O(n^2)$  ?

2.  $n^2 + 10^6n \in O(n^3)$  ?

3.  $n^2 + 10^6n \in \Theta(n^3)$  ?

4.  $(n + 1)(n - 1) \in O(n)$  ?

5.  $(n + 1)(n - 1) \in O(n^2)$  ?

Es sei  $p(n) = p_0 + p_1n + \dots + p_dn^d$  ein Polynom vom Grad  $d$  mit  $p_i > 0$  für alle  $1 \leq i \leq d$ .

Was ist das kleinste  $c$  sodass  $p(n) \in O(n^c)$ ?  $\nearrow$  Übungsblatt 2

# Kapitelüberblick

---

Letztes Kapitel: Grundbegriffe der Algorithmik

Jetzt: Erstes Themenfeld, Sortieren

Insbesondere klären wir in diesem Kapitel:

- 3 verschiedene Sortieralgorithmen
- erste Korrektheitsanalyse (mithilfe einer Schleifeninvariante)
- erste Analyse eines Divide-and-Conquer-Algorithmus

# Sortieren

## Problem: Sortieren

gegeben: Sequenz von nat. Zahlen  $A[1], \dots, A[n]$

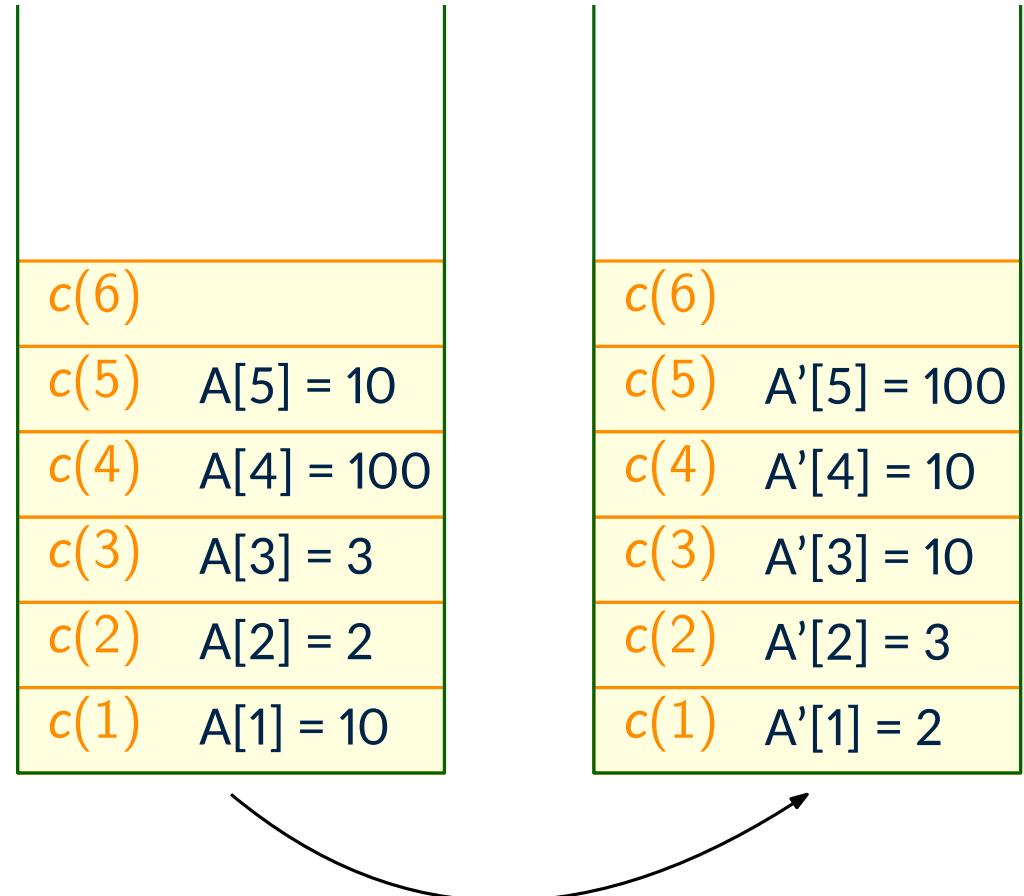
gesucht: sortierte Reihenfolge von  $A[1], \dots, A[n]$

Das heißt: finde eine Permutation  $A'[1], \dots, A'[n]$ , sodass

$$A'[1] \leq A'[2] \leq \dots \leq A'[n]$$

## Warum ist Sortieren interessant?

- grundlegendes Problem
- taucht in komplexen Anwendungen auf
- ein gutes Beispiel für das Entwerfen und Analysieren schneller Algorithmen



# Bubble Sort

# Bubble Sort

Einer der primitivsten Sortieralgorithmen

**Idee:** Solange falsch sortierte Nachbarn vertauschen, bis die Liste sortiert ist

**Pseudocode:**

```
Array A = (A[1], ..., A[n])  
  
bubbleSort(A[1 ... n]):  
    done = false  
    while (not done) do // bis A sortiert ist  
        done = true  
        for i = 1, ..., n - 1 do  
            if A[i] > A[i + 1] then  
                swap(A[i], A[i + 1])  
                done = false  
    return A[1], ..., A[n]
```

```
swap(A[i], A[j]): // vertausche Inhalt von A[i] und A[j]  
    x = A[i]  
    A[i] = A[j]  
    A[j] = x
```

// betrachte alle Nachbarn (A[1], A[2]), (A[2], A[3]), ..., (A[n - 1], A[n])

// vertausche die Nachbarn, wenn notwendig

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :

2	6	1	4	3	7
---	---	---	---	---	---

Pseudocode:

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :

2	6	1	4	3	7
---	---	---	---	---	---

Pseudocode:

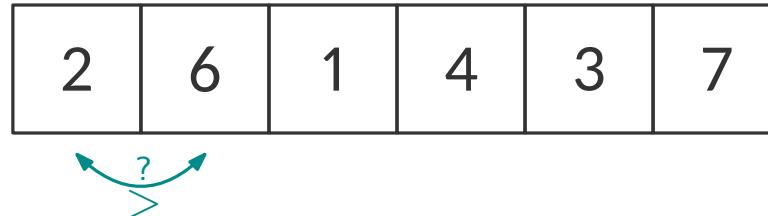
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

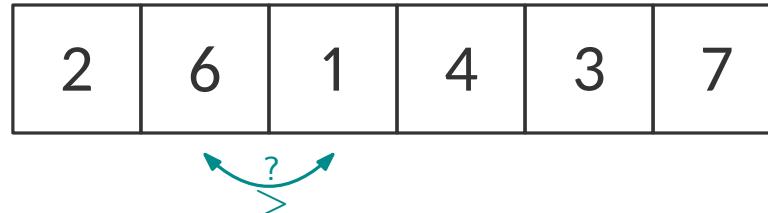
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

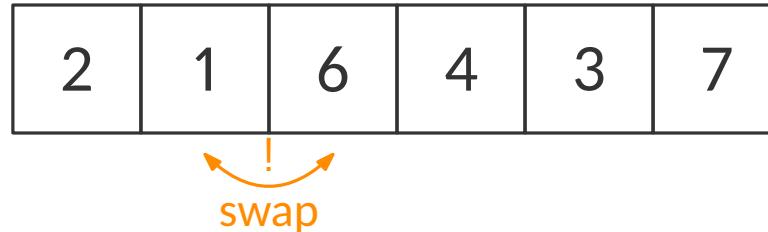
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

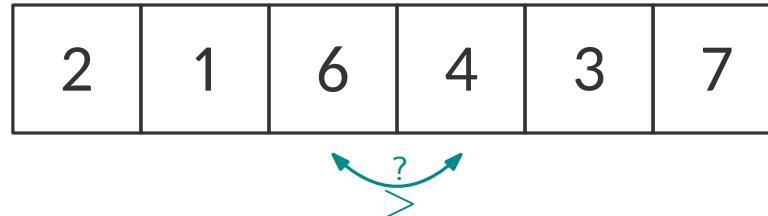
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

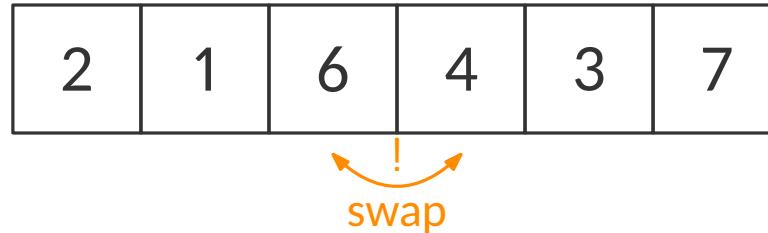
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

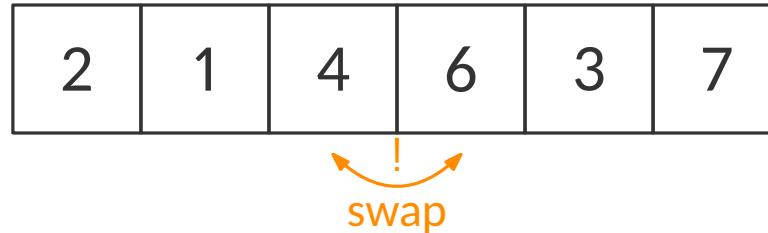
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

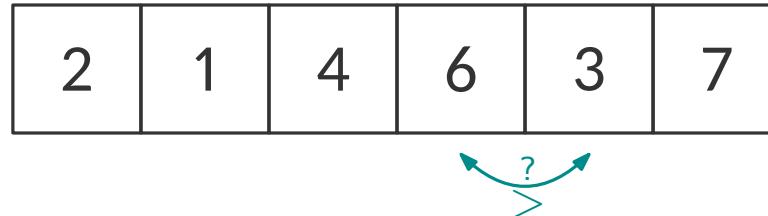
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

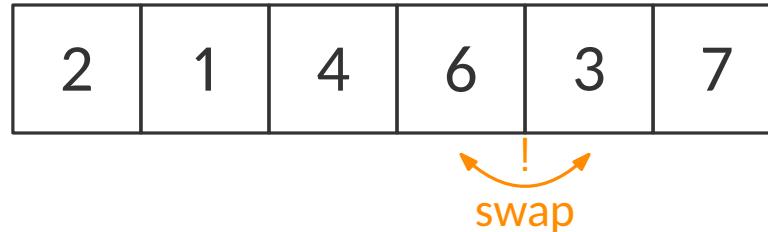
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

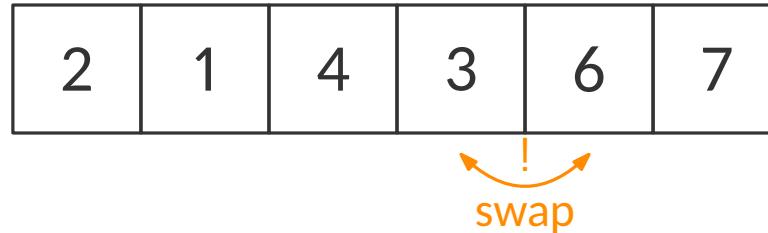
1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

1. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

A[1 ... 6]:

2	1	4	3	6	7
---	---	---	---	---	---

Pseudocode:

1. Iteration der while-Schleife

```
bubbleSort(A[1 ... n]):  
    done = false  
    while (not done) do  
        done = true  
        for i = 1, ..., n - 1 do  
            if A[i] > A[i + 1] then  
                swap(A[i], A[i + 1])  
                done = false  
    return A[1], ..., A[n]
```

# Bubble Sort: Beispiel

---

A[1 ... 6]:

2	1	4	3	6	7
---	---	---	---	---	---

Pseudocode:

2. Iteration der while-Schleife

```
bubbleSort(A[1 ... n]):  
    done = false  
    while (not done) do  
        done = true  
        for i = 1, ..., n - 1 do  
            if A[i] > A[i + 1] then  
                swap(A[i], A[i + 1])  
                done = false  
    return A[1], ..., A[n]
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

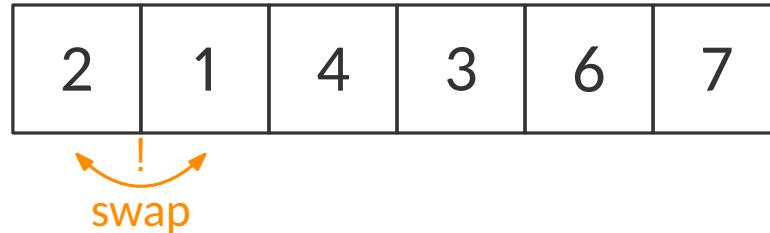
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

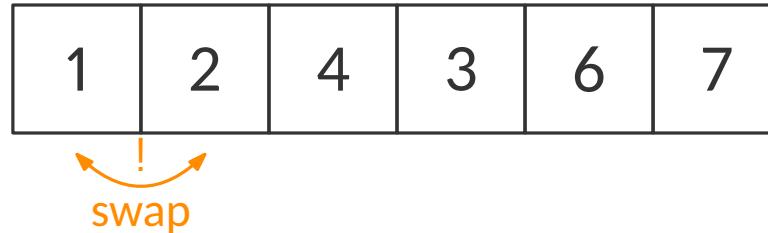
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

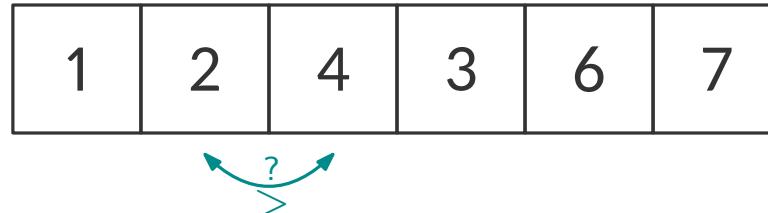
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

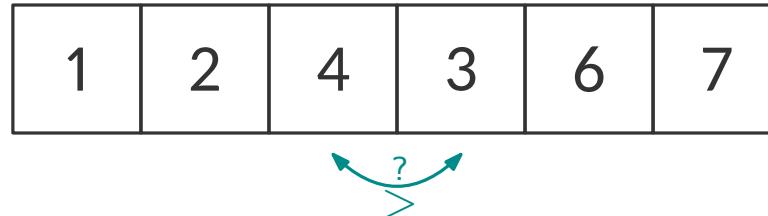
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

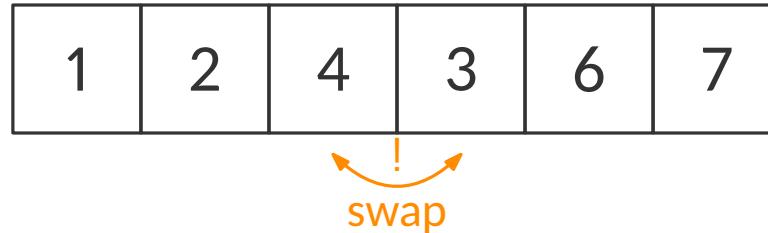
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

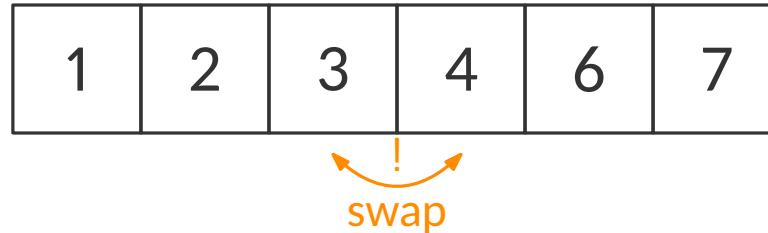
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

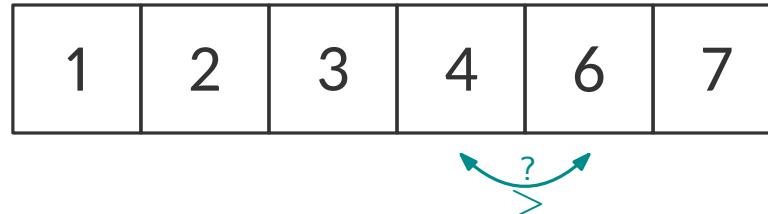
2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :

1	2	3	4	6	7
---	---	---	---	---	---

Pseudocode:

2. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :

1	2	3	4	6	7
---	---	---	---	---	---

Pseudocode:

3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

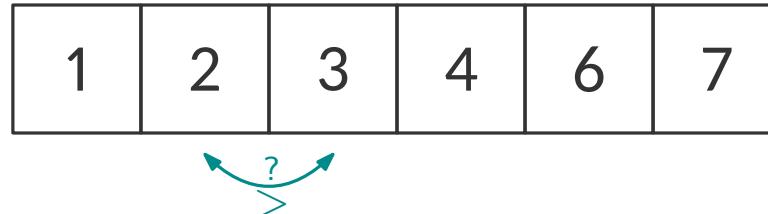
3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

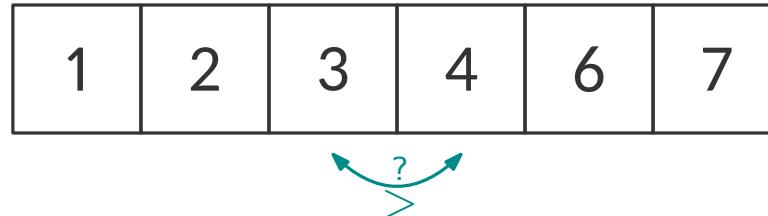
3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :



Pseudocode:

3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :

1	2	3	4	6	7
---	---	---	---	---	---

Pseudocode:

3. Iteration der while-Schleife

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Bubble Sort: Beispiel

---

$A[1 \dots 6]$ :

1	2	3	4	6	7
---	---	---	---	---	---

Pseudocode:

Ausgabe: 1,2,3,4,6,7

```
bubbleSort( $A[1 \dots n]$ ):  
    done = false  
    while (not done) do  
        done = true  
        for  $i = 1, \dots, n - 1$  do  
            if  $A[i] > A[i + 1]$  then  
                swap( $A[i], A[i + 1]$ )  
                done = false  
    return  $A[1], \dots, A[n]$ 
```

# Analyse von Bubblesort

---

## Theorem

Bubble Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Wir müssen folgendes beweisen:

- Bubble Sort terminiert
- das Ergebnis ist die sortierte Reihenfolge von  $A[1 \dots n]$
- Bubble Sort benötigt höchstens  $O(n^2)$  Berechnungsschritte

`bubbleSort( $A[1 \dots n]$ ):`

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if  $A[i] > A[i + 1]$  then
            swap( $A[i], A[i + 1]$ )
            done = false
return  $A[1], \dots, A[n]$ 
```

Für den Beweis führen wir ein hilfreiches Prinzip ein:

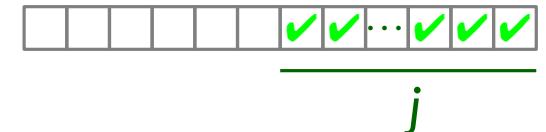
- Argumentation mithilfe einer **Schleifenvarianten**
  - Aussage, die erhalten bleibt durch Schleifenausführung
  - wird induktiv bewiesen

# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if  $A[i] > A[i + 1]$  then
            swap( $A[i], A[i + 1]$ )
            done = false
    return  $A[1], \dots, A[n]$ 
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if  $A[i] > A[i + 1]$  then
            swap( $A[i], A[i + 1]$ )
            done = false
    return  $A[1], \dots, A[n]$ 
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:



$$\overline{j} = 0$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:



$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

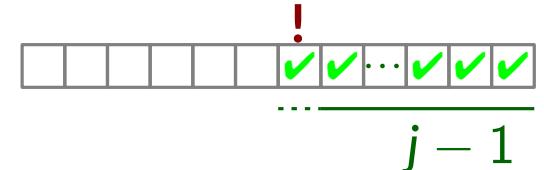


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
    return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

bubbleSort( $A[1 \dots n]$ ):

```
done = false  
  
while (not done) do  
    done = true  
    for i = 1, ..., n - 1 do  
        if  $A[i] > A[i + 1]$  then  
            swap( $A[i], A[i + 1]$ )  
            done = false  
  
return  $A[1], \dots, A[n]$ 
```

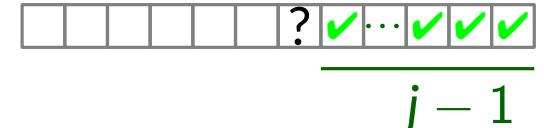


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

bubbleSort( $A[1 \dots n]$ ):

```
done = false  
while (not done) do  
    done = true  
    for i = 1, ..., n - 1 do  
        if  $A[i] > A[i + 1]$  then  
            swap( $A[i], A[i + 1]$ )  
            done = false  
    return  $A[1], \dots, A[n]$ 
```

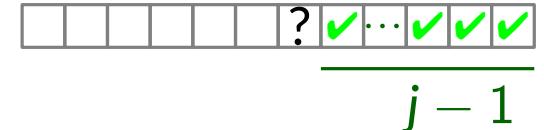


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

bubbleSort( $A[1 \dots n]$ ):

```
done = false  
while (not done) do  
    done = true  
    for i = 1, ..., n - 1 do  
        if A[i] > A[i + 1] then  
            swap(A[i], A[i + 1])  
            done = false  
    return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

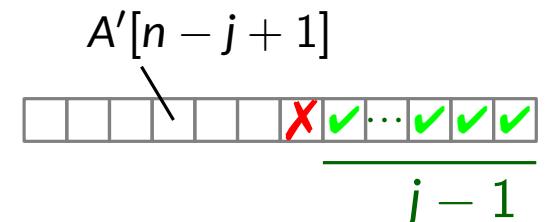
Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

bubbleSort( $A[1 \dots n]$ ):

```
done = false  
  
while (not done) do  
    done = true  
    for i = 1, ..., n - 1 do  
        if A[i] > A[i + 1] then  
            swap(A[i], A[i + 1])  
            done = false  
  
return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

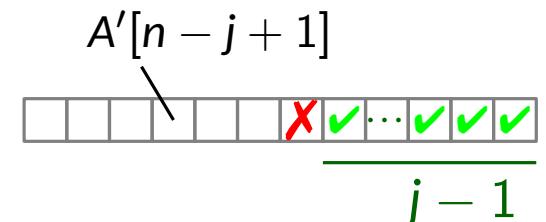
Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

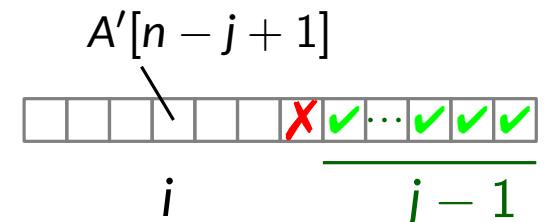
Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

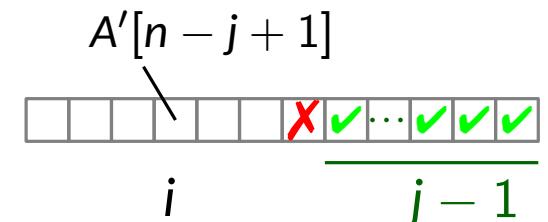
Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$

**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

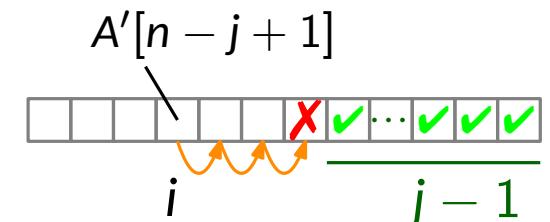
Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

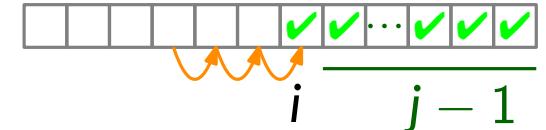


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1$

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

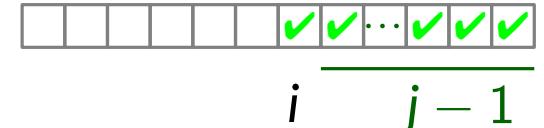


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
    return A[1], ..., A[n]
```

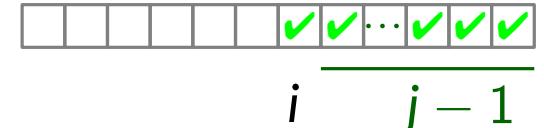


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
    return A[1], ..., A[n]
```

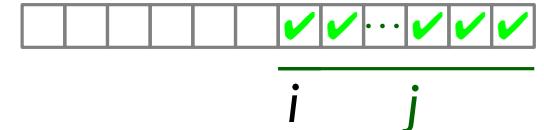


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:

$$A[n - j + 1] \leq A[n - j + 2] \leq \dots A[n]$$

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
    return A[1], ..., A[n]
```

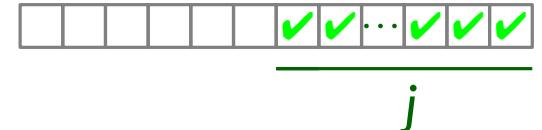


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:

$$A[n - j + 1] \leq A[n - j + 2] \leq \dots A[n]$$

→ wir vertauschen nichts mehr

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
    return A[1], ..., A[n]
```

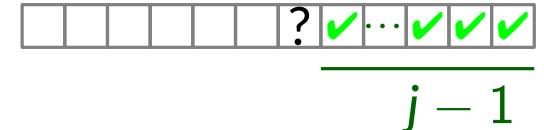


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:

sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:

$$A[n - j + 1] \leq A[n - j + 2] \leq \dots A[n]$$

→ wir vertauschen nichts mehr

2. Wenn  $A[n - j + 1] = A'[n - j + 1]$ , dann:

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

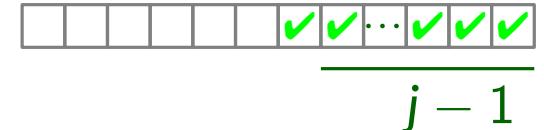


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:  
sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:  
$$A[n - j + 1] \leq A[n - j + 2] \leq \dots A[n]$$
  
→ wir vertauschen nichts mehr
2. Wenn  $A[n - j + 1] = A'[n - j + 1]$ , dann:

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

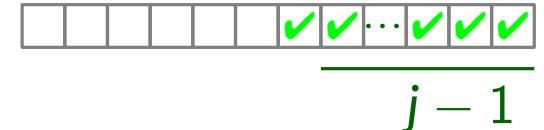


# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:  
sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:  
$$A[n - j + 1] \leq A[n - j + 2] \leq \dots A[n]$$
  
→ wir vertauschen nichts mehr
2. Wenn  $A[n - j + 1] = A'[n - j + 1]$ , dann:  
wir vertauschen nichts ab  $i = n - j + 1$

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
    return A[1], ..., A[n]
```



# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:  
sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:  
$$A[n - j + 1] \leq A[n - j + 2] \leq \dots \leq A[n]$$
  
→ wir vertauschen nichts mehr
2. Wenn  $A[n - j + 1] = A'[n - j + 1]$ , dann:  
wir vertauschen nichts ab  $i = n - j + 1$   
→ die **Aussage** gilt am Ende der  $j$ -ten Schleife.

bubbleSort( $A[1 \dots n]$ ):

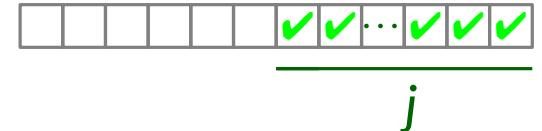
```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

# Schleifeninvariante für Bubblesort

Für  $A[1], \dots, A[n]$  sei  $A'[1], \dots, A'[n]$  die sortierte Reihenfolge

**Aussage** Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:

$$A[k] = A'[k] \text{ für alle } k \in \{n - j + 1, \dots, n\}$$



**Beweis:** Vereinfachende Annahme: Alle  $A[i]$  sind unterschiedlich. Gilt jedoch allgemein!

**Induktionsanfang:**

Die **Aussage** ist trivial erfüllt für  $j = 0$ .  
(nichts zu zeigen)

**Induktionsschritt:**

Die **Aussage** sei erfüllt für  $j - 1$ .

Im  $j$ -ten Durchlauf der Schleife passiert folgendes:

1. Wenn  $A[n - j + 1] \neq A'[n - j + 1]$ , dann:  
sobald  $i$  so ist, dass  $A[i] = A'[n - j + 1]$ ,  
dann vertauschen wir immer Nachbarn  
bis  $i = n - j + 1 \rightarrow A[n - j + 1] = A'[n - j + 1]$   
ab dann gilt:  
$$A[n - j + 1] \leq A[n - j + 2] \leq \dots A[n]$$
  
→ wir vertauschen nichts mehr
2. Wenn  $A[n - j + 1] = A'[n - j + 1]$ , dann:  
wir vertauschen nichts ab  $i = n - j + 1$   
→ die **Aussage** gilt am Ende der  $j$ -ten Schleife.  $\square$

bubbleSort( $A[1 \dots n]$ ):

```
done = false
while (not done) do
    done = true
    for i = 1, ..., n - 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            done = false
return A[1], ..., A[n]
```

# Analyse von Bubblesort

## Theorem

Bubble Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Wir müssen folgendes beweisen:

- Bubble Sort terminiert
- das Ergebnis ist die sortierte Reihenfolge von  $A[1 \dots n]$
- Bubble Sort benötigt höchstens  $O(n^2)$  Berechnungsschritte

## Beweis vom Theorem (1):

```
bubbleSort(A[1 ... n])  
  
done = false  
  
while (not done) do  
  
    done = true  
  
    for i = 1, ..., n - 1 do  
  
        if A[i] > A[i + 1] then  
            swap(A[i], A[i + 1])  
            done = false  
  
    return A[1], ..., A[n]
```

Unsere bewiesene **Schleifeninvariante**:

Nach dem  $j$ -ten Durchlauf der while-Schleife gilt:  
 $A[k] = A'[k]$  für alle  $k \in \{n - j + 1, \dots, n\}$

$A'[1], \dots, A'[n]$  bezeichnet die sortierte Reihenfolge von  $A[1], \dots, A[n]$

Schleifeninvariante zeigt:

spätestens wenn  $j = n$  gilt  $A[1] \leq A[2] \leq \dots \leq A[n]$

→ spätestens der  $(n + 1)$ -ste Schleifendurchlauf behält  $\text{done}=\text{true}$

Terminierung ✓

→ das Ergebnis ist dann die sortierte Permutation des Eingabearrays

Korrektheit ✓

→ **Fakt:** es gibt höchstens  $n + 1$  Iterationen der while-Schleife

# Analyse von Bubblesort

---

## Theorem

Bubble Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Wir müssen folgendes beweisen:

- Bubble Sort terminiert ✓
- das Ergebnis ist die sortierte Reihenfolge von  $A[1 \dots n]$  ✓
- Bubble Sort benötigt höchstens  $O(n^2)$  Berechnungsschritte

Beweis vom Theorem (2):

Fakt: es gibt  $\leq n + 1$  Iterationen der while-Schleife

```
bubbleSort(A[1 ... n])
```

```
    done = false
```

```
    while (not done) do
```

```
        done = true
```

```
        for i = 1, ..., n - 1 do
```

```
            if A[i] > A[i + 1] then
```

```
                swap(A[i], A[i + 1])
```

```
                done = false
```

```
    return A[1], ..., A[n]
```

# Analyse von Bubblesort

## Theorem

Bubble Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Wir müssen folgendes beweisen:

- Bubble Sort terminiert ✓
- das Ergebnis ist die sortierte Reihenfolge von  $A[1 \dots n]$  ✓
- Bubble Sort benötigt höchstens  $O(n^2)$  Berechnungsschritte

Beweis vom Theorem (2):

**Fakt:** es gibt  $\leq n + 1$  Iterationen der while-Schleife

```
bubbleSort(A[1 ... n])  
    done = false  
    while (not done) do  
        done = true  
        for i = 1, ..., n - 1 do  
            if A[i] > A[i + 1] then  
                swap(A[i], A[i + 1])  
                done = false  
    return A[1], ..., A[n]
```

## Laufzeitanalyse:

Es sei  $T(n)$  die Worst-Case-Laufzeit von Bubble Sort

1. ein Durchlauf des If-Blocks benötigt  $\leq c_1$  Schritte für eine Konstante  $c_1$
2. die for-Schleife benötigt höchstens  $c_2 + \sum_{i=1}^{n-1} c_1 = c_2 + (n - 1)c_1$  Schritte für eine Konstante  $c_2$
3. ein Durchlauf der while-Schleife benötigt höchstens  $c_3 + (n - 1)c_1 \leq c_3 + nc_1$  Schritte für eine Konstante  $c_3$
4. aus dem **Fakt** folgt:  
$$T(n) \leq c_4 + (n + 1)(c_3 + nc_1)$$
 für eine Konstante  $c_4$

# Analyse von Bubblesort

## Theorem

Bubble Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Wir müssen folgendes beweisen:

- Bubble Sort terminiert ✓
- das Ergebnis ist die sortierte Reihenfolge von  $A[1 \dots n]$  ✓
- Bubble Sort benötigt höchstens  $O(n^2)$  Berechnungsschritte ✓

Beweis vom Theorem (2):

Fakt: es gibt  $\leq n + 1$  Iterationen der while-Schleife

```
bubbleSort(A[1 ... n])  
    done = false  
    while (not done) do  
        done = true  
        for i = 1, ..., n - 1 do  
            if A[i] > A[i + 1] then  
                swap(A[i], A[i + 1])  
                done = false  
    return A[1], ..., A[n]
```

Laufzeitanalyse (2):

$$\begin{aligned} T(n) &\leq c_4 + (n + 1)(c_3 + nc_1) \\ &= (c_4 + c_3) + n \cdot (c_3 + c_1) + n^2 \cdot c_1 \\ &\Rightarrow T(n) \in O(n^2) \end{aligned}$$

□

# Selection Sort

# Selection Sort

---

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

```
selectMinimum(A[1 ... n], a, b)           // gib Position des kleinsten Elements in A[a], ..., A[b] zurück
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

5	2	6	1	4	3
---	---	---	---	---	---

```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

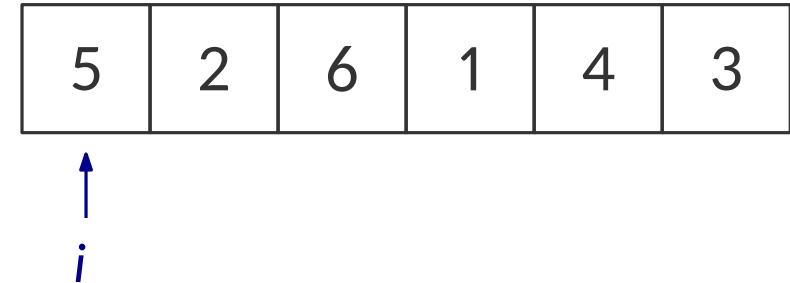
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

# Selection Sort

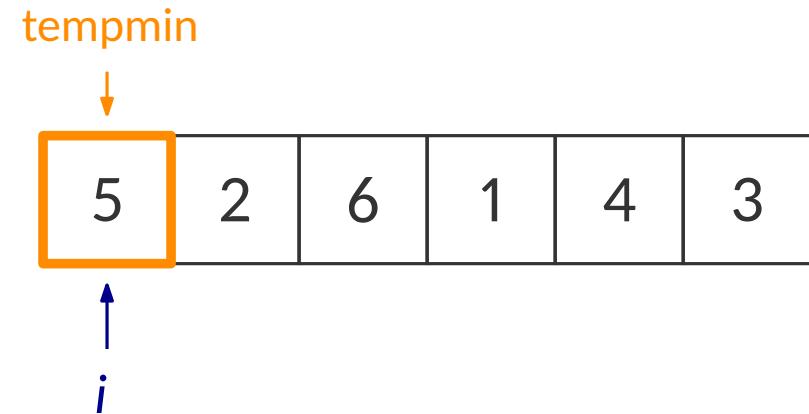
Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element

→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

```
tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

# Selection Sort

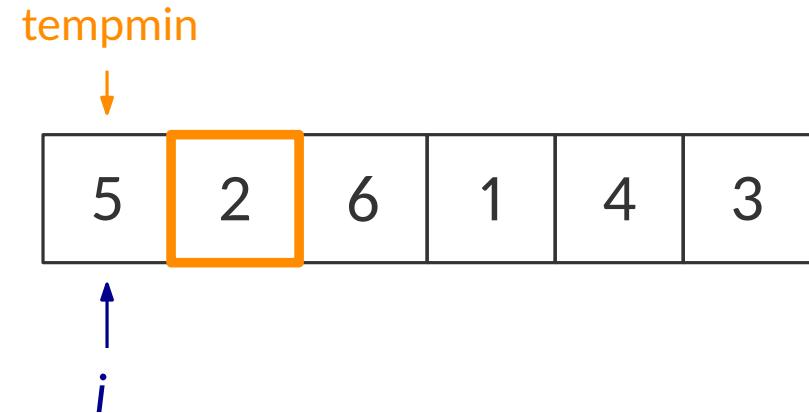
Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element

→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

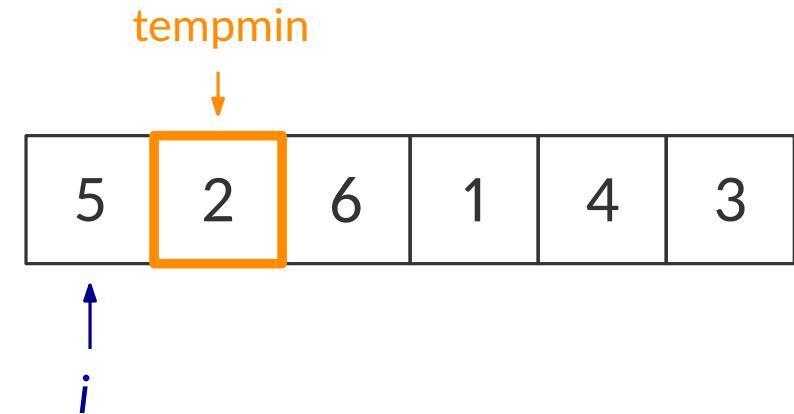
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

```
tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

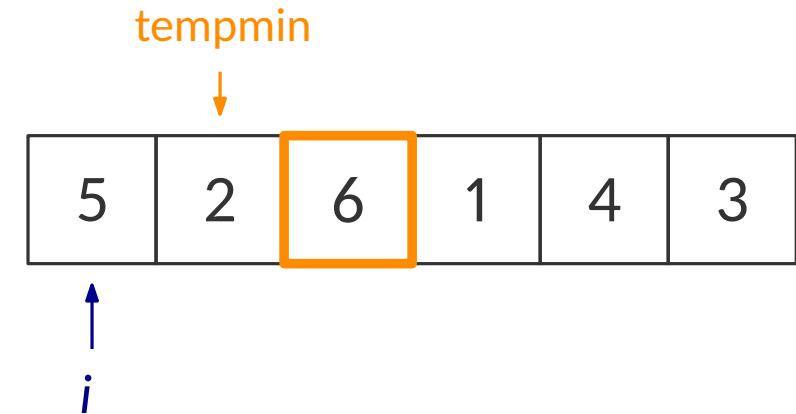
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

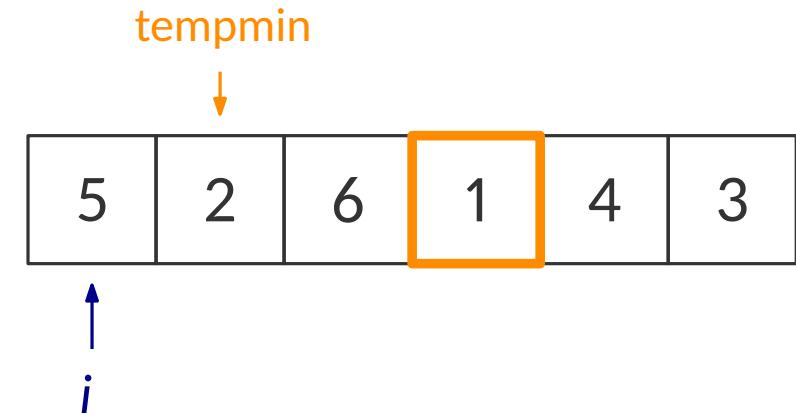
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

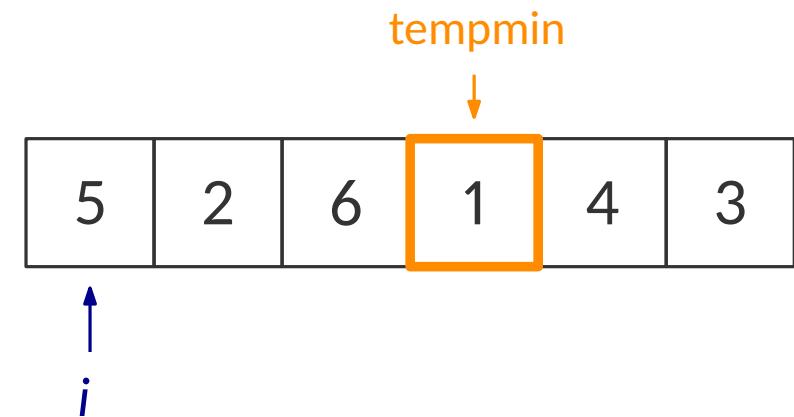
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

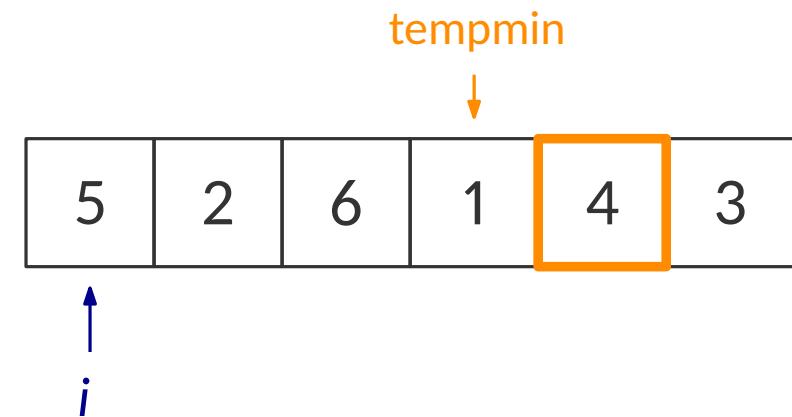
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

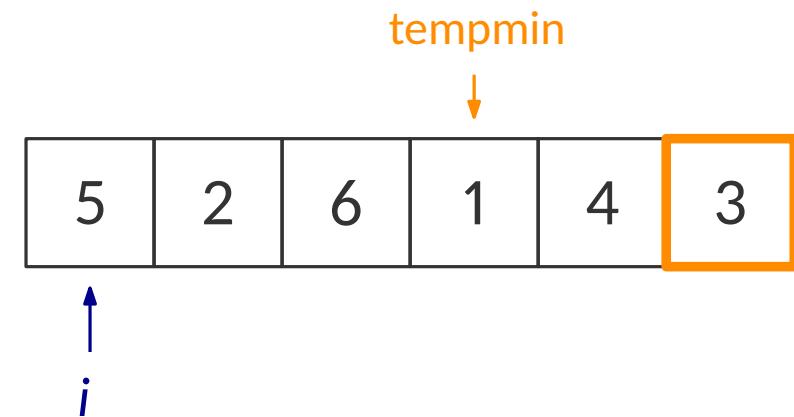
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

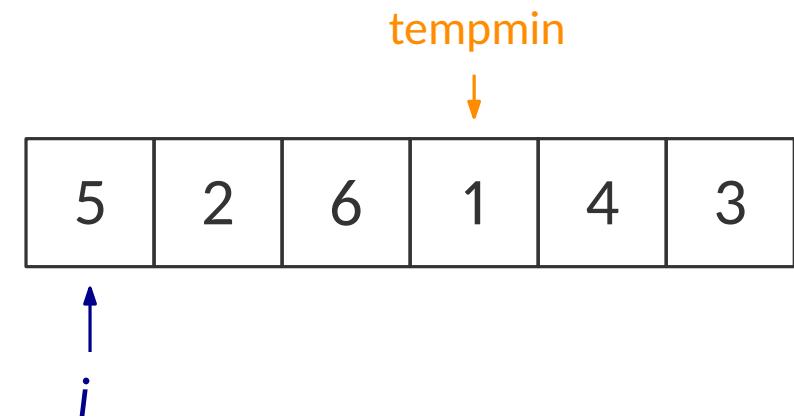
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

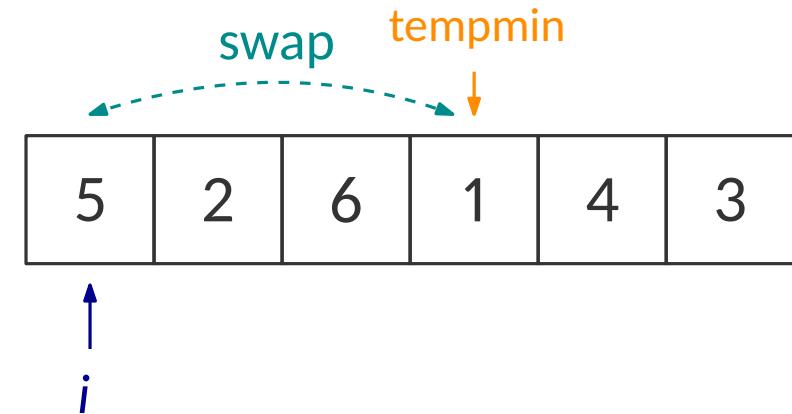
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

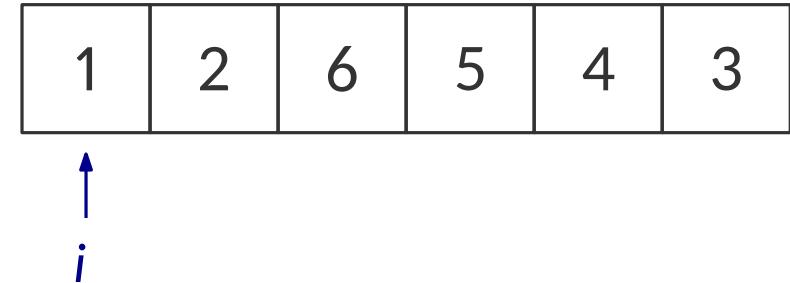
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

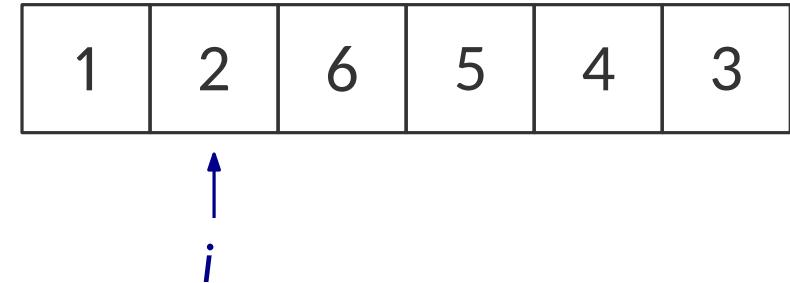
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

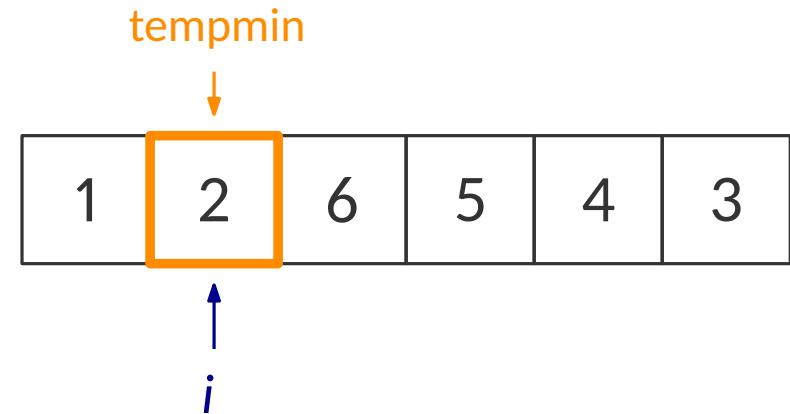
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

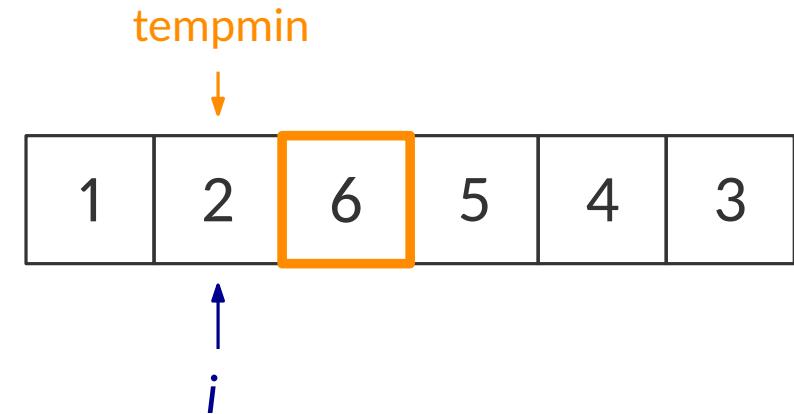
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

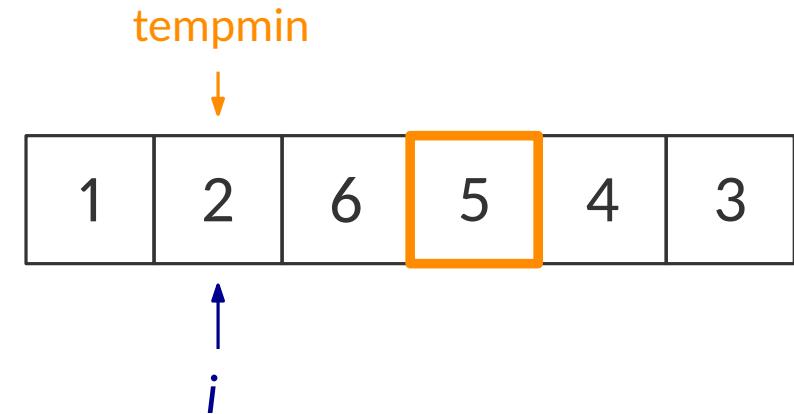
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

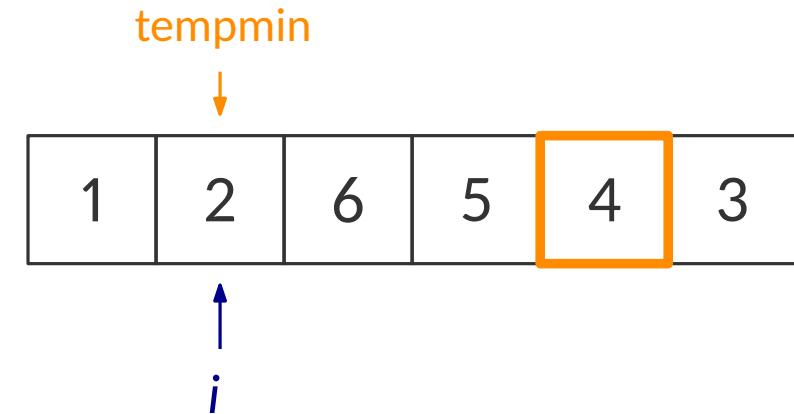
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

# Selection Sort

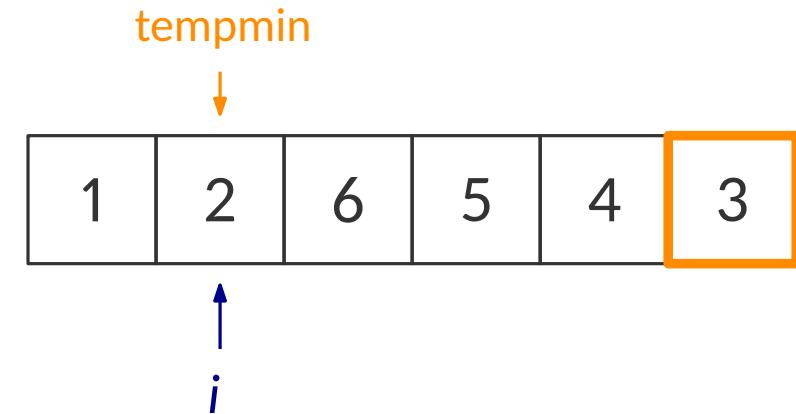
Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element

→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

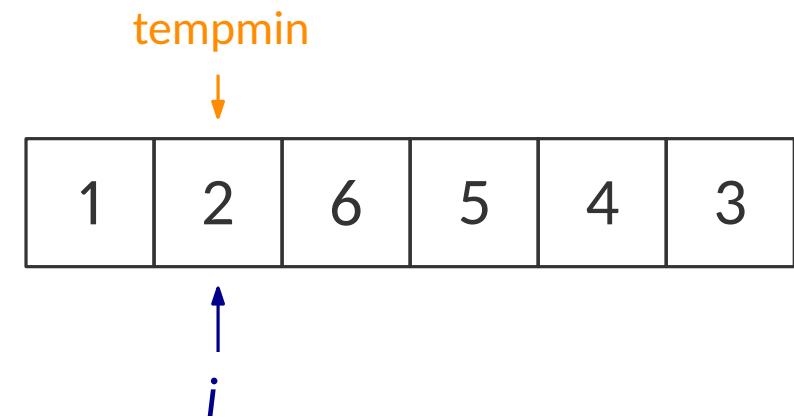
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element

→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
```

```
  for  $i = 1, \dots, n - 1$  do
```

```
    |    $j = \text{selectMinimum}(A, i, n)$ 
```

```
    |   swap(A[i], A[j])
```

```
  return A[1], \dots, A[n]
```

```
selectMinimum(A[1 ... n], a, b)
```

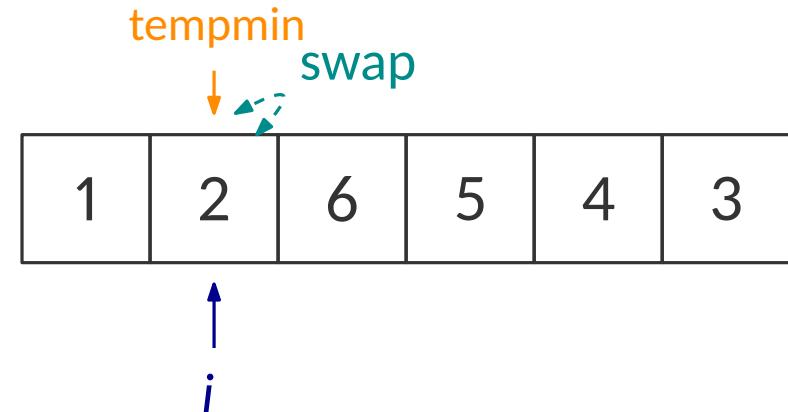
```
  tempmin = a
```

```
  for  $i = a + 1, \dots, b$  do
```

```
    |   if  $A[i] < A[\text{tempmin}]$  then
```

```
    |       |   tempmin = i
```

```
  return tempmin
```



//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

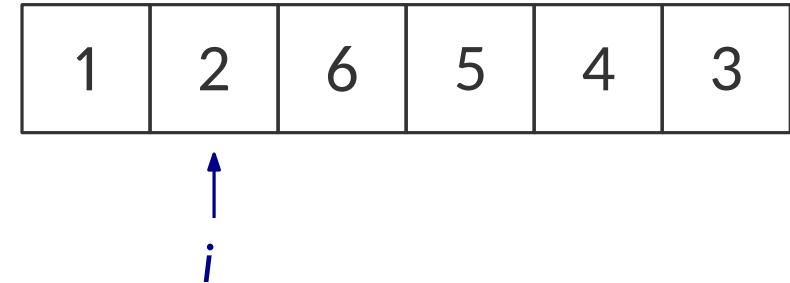
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

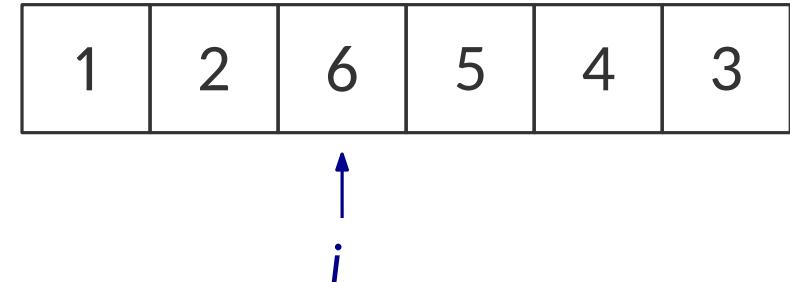
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

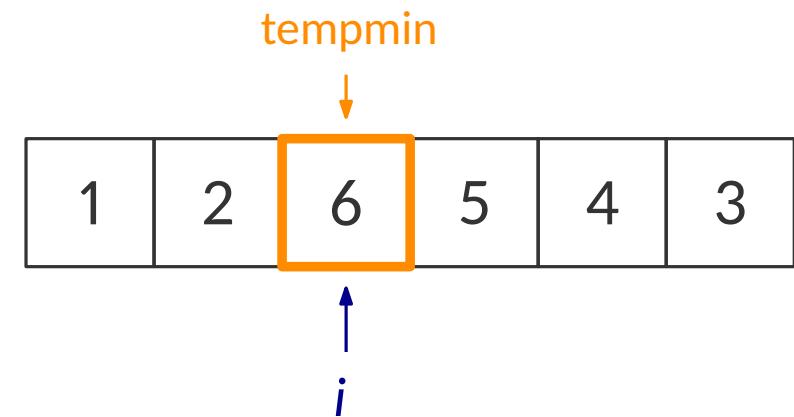
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

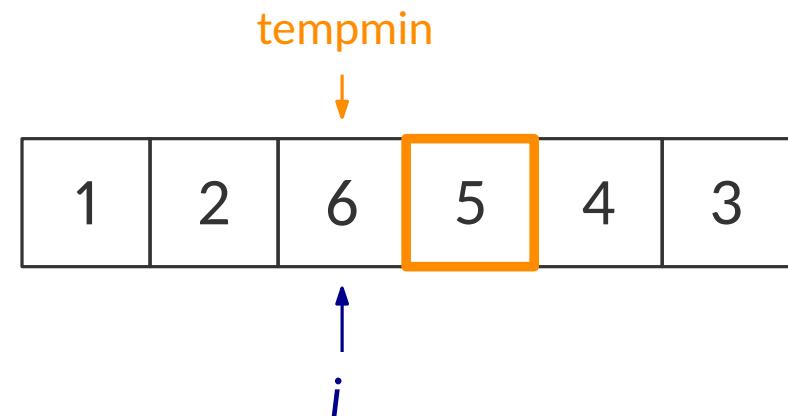
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

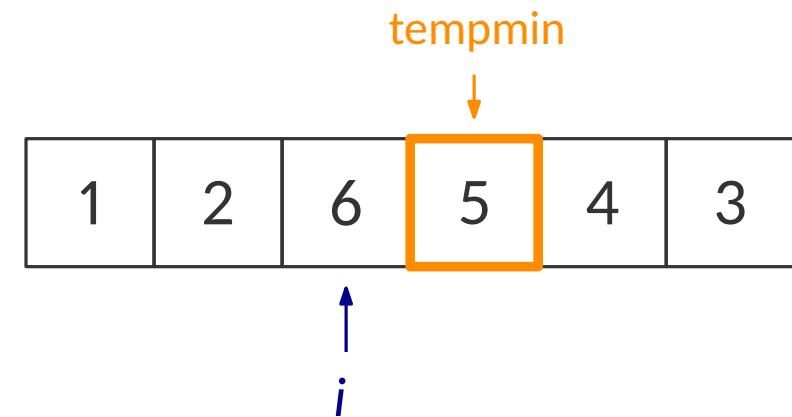
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

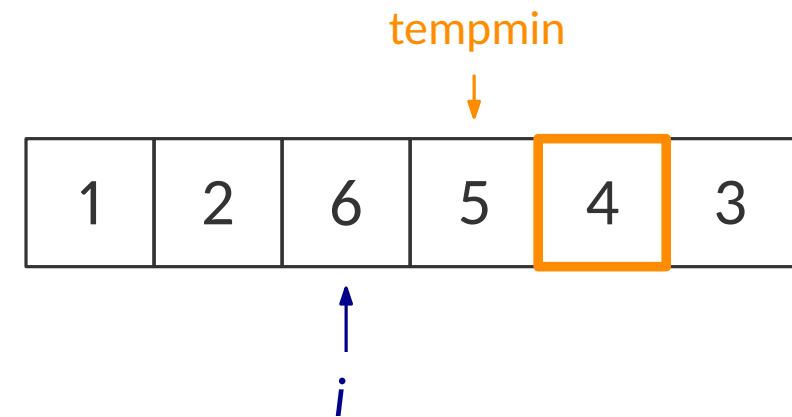
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

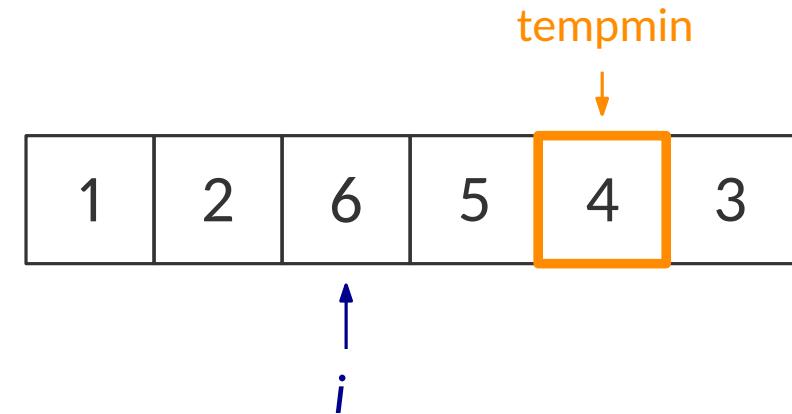
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

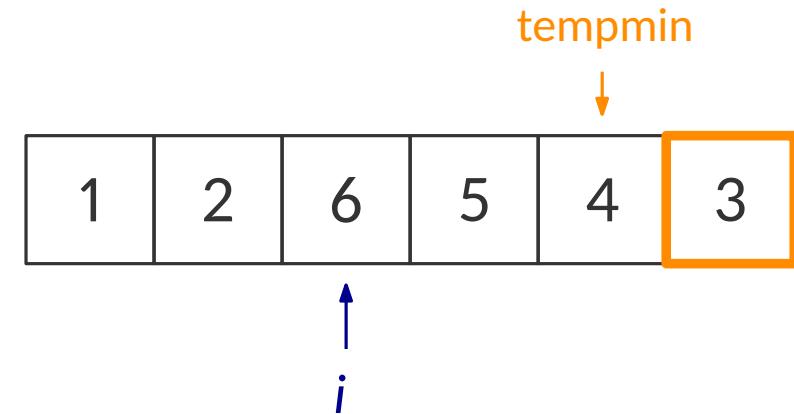
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

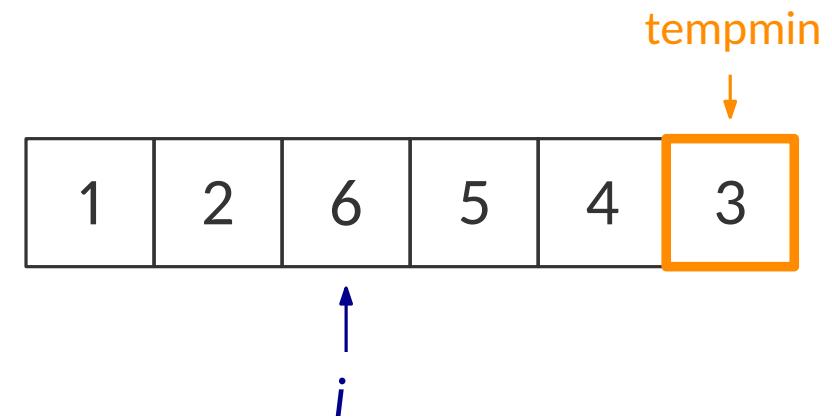
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

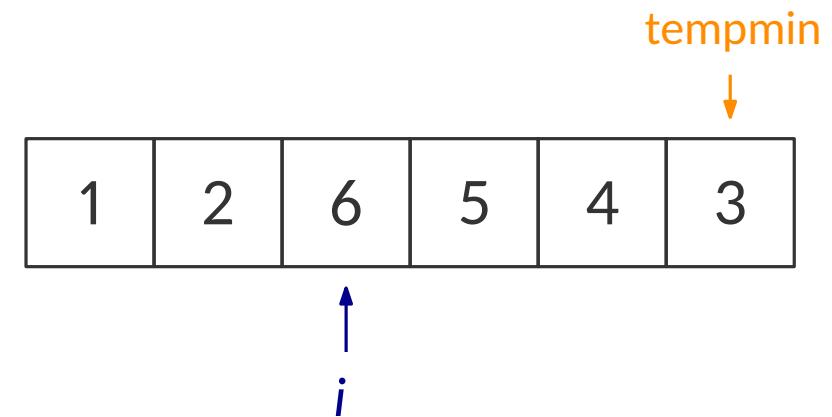
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

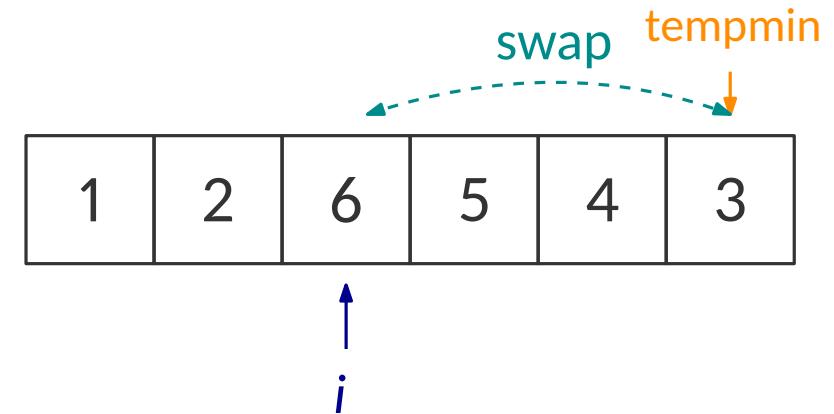
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

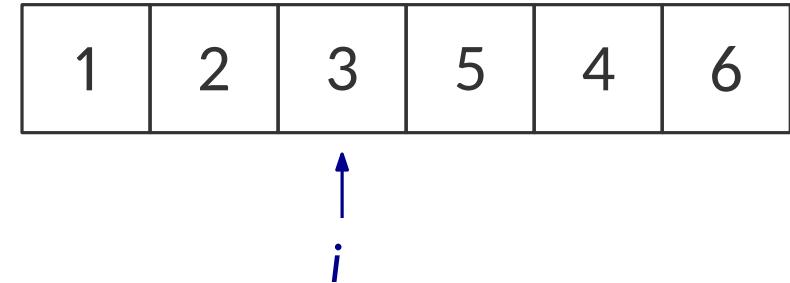
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

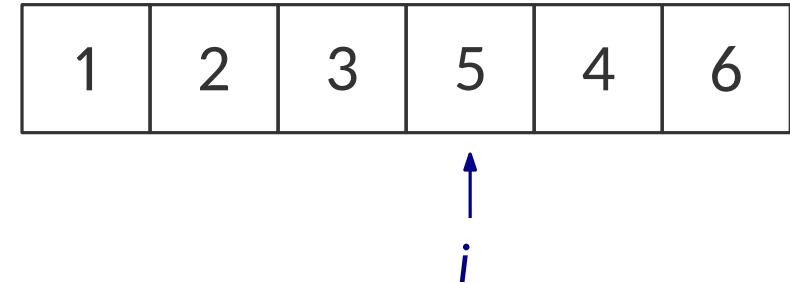
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

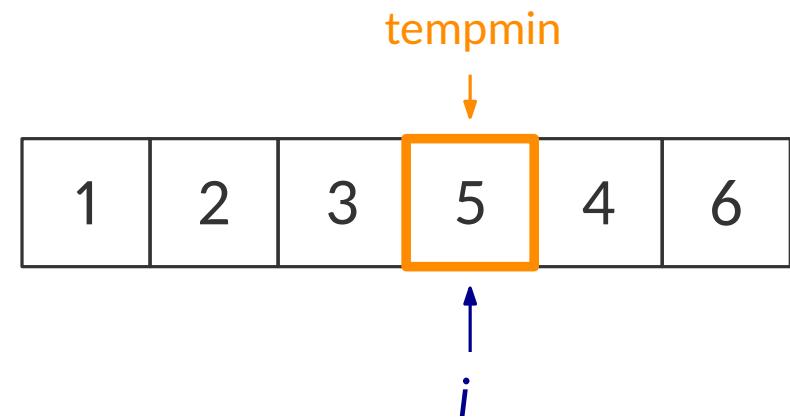
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

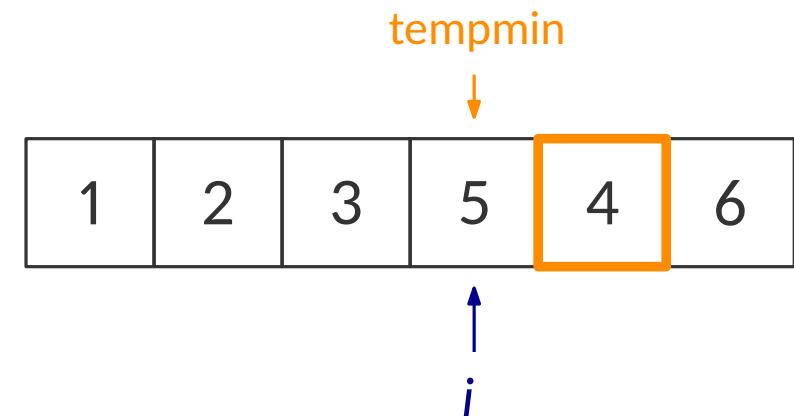
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

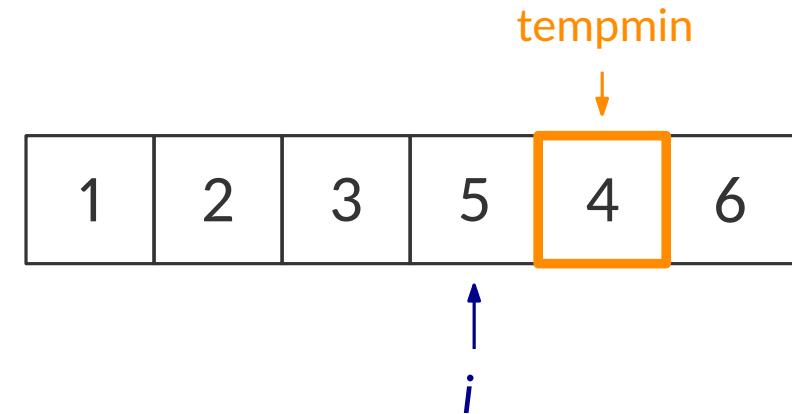
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

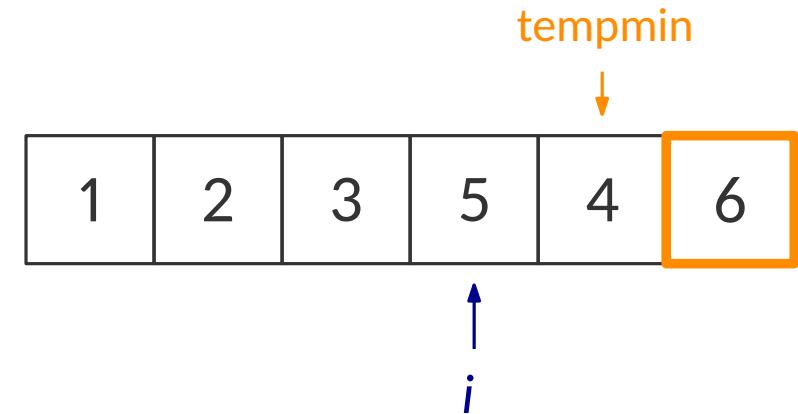
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

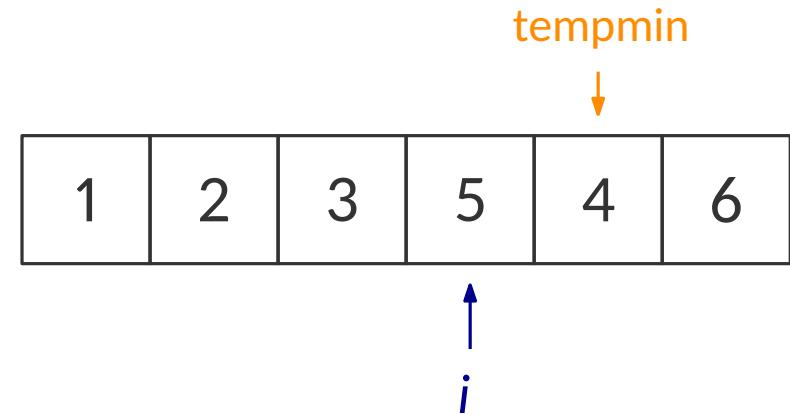
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

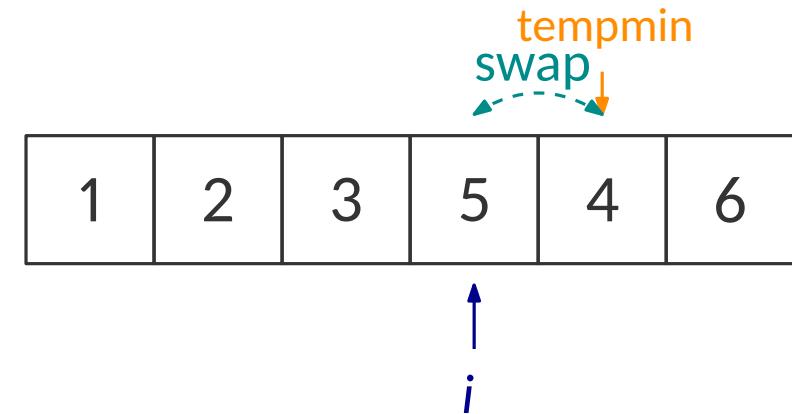
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

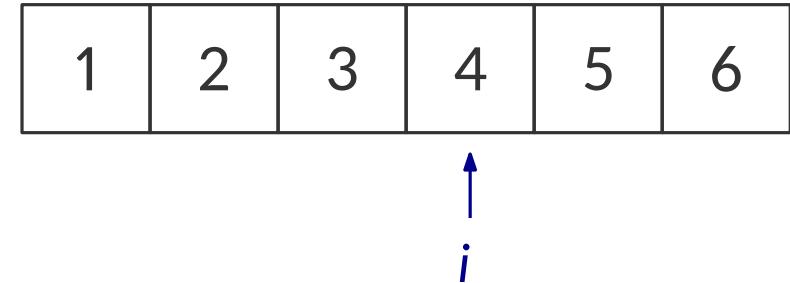
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

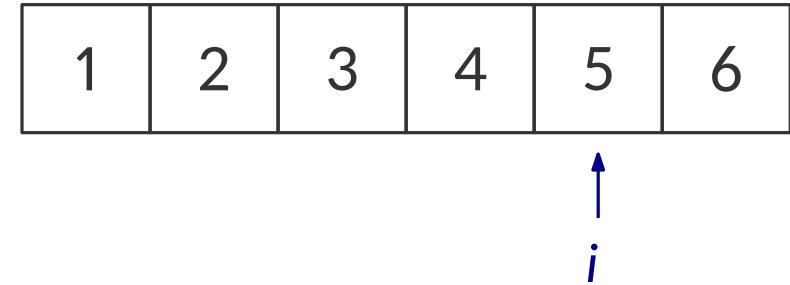
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

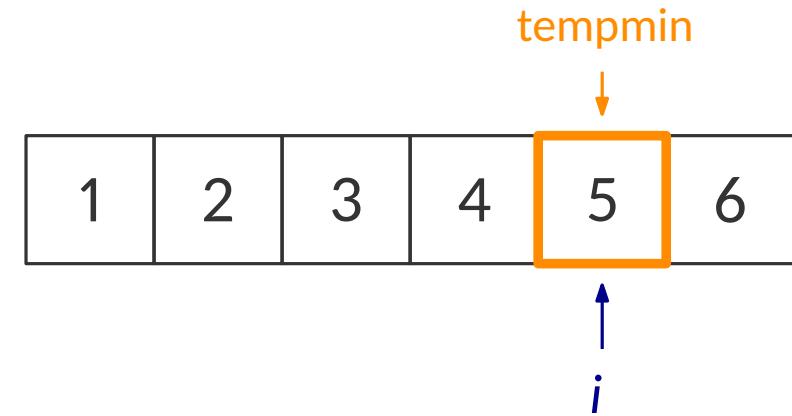
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

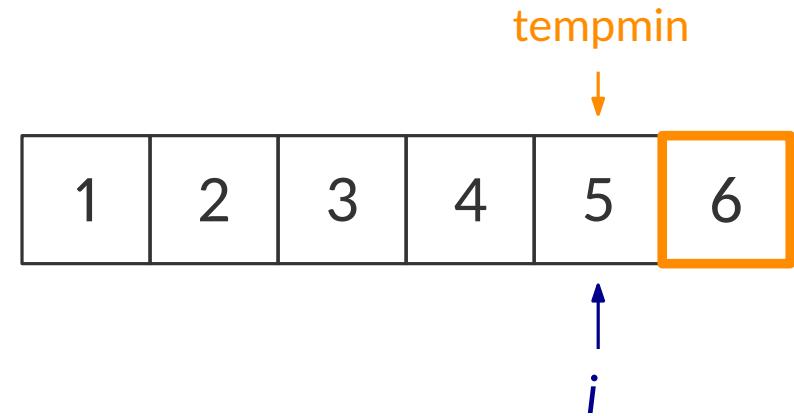
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

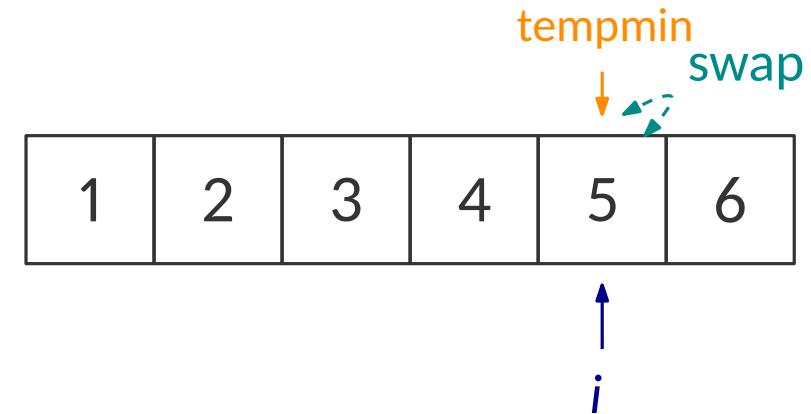
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

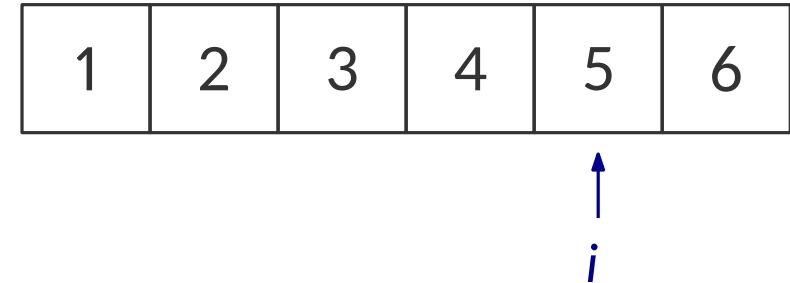
# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```



```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in A[a], ..., A[b] zurück

# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

1	2	3	4	5	6
---	---	---	---	---	---

```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

//gib Position des kleinsten Elements in  $A[a], \dots, A[b]$  zurück

# Selection Sort

---

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

## Theorem

Selection Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

## Terminierung ✓

```
selectMinimum(A[1 ... n], a, b)
```

```
tempmin = a
for i = a + 1, ..., b do
  if A[i] < A[tempmin] then
    | tempmin = i
return tempmin
```

# Selection Sort

---

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

## Theorem

Selection Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

**Terminierung** ✓  
**Korrektheit:**

```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

# Selection Sort

---

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

**Idee:** Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

```
selectMinimum(A[1 ... n], a, b)
```

```
tempmin = a
for i = a + 1, ..., b do
  if A[i] < A[tempmin] then
    | tempmin = i
return tempmin
```

## Theorem

Selection Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

**Terminierung** ✓

**Korrektheit:**

**Frage:**

Was für eine Schleifeninvariante sollten wir benutzen?

# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    |   j = selectMinimum(A, i, n)
    |   swap(A[i], A[j])
  return A[1], ..., A[n]
```

```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    |   if A[i] < A[tempmin] then
    |     |   tempmin = i
  return tempmin
```

## Theorem

Selection Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Terminierung ✓

Korrektheit: (Details ausgelassen) ✓

Laufzeit:

Behauptung: Es gibt Konstanten  $c_1, c_2, c_3$  sodass:

$$T(n) \leq c_1 + \sum_{i=1}^{n-1} (c_2 + \sum_{j=i}^n c_3)$$

Frage: Warum?

# Selection Sort

Ein vielleicht etwas strukturierteres Vorgehen als Bubble Sort

Idee: Wir gehen rundenweise vor:

In der  $i$ . Runde suchen wir das  $i$ . kleinste Element  
→ setzen es auf die  $i$ . Stelle

```
selectionSort(A[1 ... n])
  for i = 1, ..., n - 1 do
    | j = selectMinimum(A, i, n)
    | swap(A[i], A[j])
  return A[1], ..., A[n]
```

```
selectMinimum(A[1 ... n], a, b)
  tempmin = a
  for i = a + 1, ..., b do
    | if A[i] < A[tempmin] then
    |   | tempmin = i
  return tempmin
```

## Theorem

Selection Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n^2)$ .

Terminierung ✓

Korrektheit: (Details ausgelassen) ✓

Laufzeit: ✓

Behauptung: Es gibt Konstanten  $c_1, c_2, c_3$  sodass:

$$\begin{aligned} T(n) &\leq c_1 + \sum_{i=1}^{n-1} (c_2 + \sum_{j=i}^n c_3) \\ &\leq c_1 + \sum_{i=1}^n (c_2 + \sum_{j=1}^n c_3) \\ &= c_1 + \sum_{i=1}^n c_2 + \sum_{i=1}^n \sum_{j=1}^n c_3 \\ &= c_1 + c_2 n + c_3 n^2 \\ &\Rightarrow T(n) \in O(n^2) \end{aligned}$$



# Bubble Sort vs. Selection Sort

---

Bubble Sort und Selection Sort haben die gleiche asymptotische worst-case Schranke von  $O(n^2)$ .

Sind beide Algorithmen gleich gut?

- in O-Notation versteckte Konstanten könnten sich unterscheiden  
→ kann sich in der Praxis bemerkbar machen
- Für sortierte oder nahezu sortierte Laufzeiten ist Bubble Sort wesentlich schneller

Benötigen beide Algorithmen im worst-case tatsächlich Zeit  $\Omega(n^2)$ ?

- klar für Selection Sort:  
 $n - 1$  Runden  
in Runde  $i$  führen wir  $\geq c \cdot (n - i)$  Operationen aus (für eine Konstante  $c$ )  
$$\sum_{i=1}^{n-1} c \cdot (n - i) = \sum_{j=1}^{n-1} c \cdot j = c \cdot \sum_{j=1}^{n-1} j = c \cdot \frac{(n-1)n}{2} \in \Omega(n^2)$$
- gilt auch für Bubble Sort:  
auf Eingabe  $n, n - 1, \dots, 1$  machen wir genau  $\sum_{i=1}^n (n - i) \in \Omega(n^2)$  Vertauschungen  
⇒ Selection Sort und Bubble Sort haben Worst-Case-Laufzeit in  $\Theta(n^2)$   
Können wir schneller sortieren als in Zeit  $\Theta(n^2)$ ?

# Merge Sort

# Merge Sort: Eine rekursiver Sortieralgorithmus

Ja, wir können in Zeit  $o(n^2)$  sortieren!

Merge Sort ist ein solches Verfahren

Dieses folgt dem Algorithmenentwurfsprinzip **Teile-und-Herrsche** (Divide & Conquer)

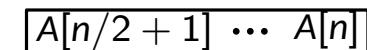
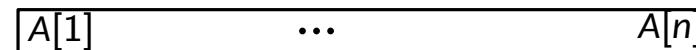
prinzipieller Ansatz:

- Teile das Problem in kleinere **Teilprobleme**
- Löse die **Teilprobleme** rekursiv
- Kombiniere die Lösungen der **Teilprobleme** zu einer Gesamtlösung

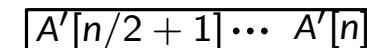
"Divide"  
"Conquer"

Idee von Merge Sort:

1. Teile  $A[1], \dots, A[n]$  in zwei Hälften

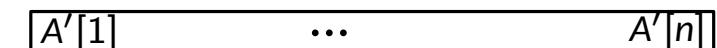


2. Sortiere jede Hälfte getrennt **Magie der Rekursion!**



3. Merge: "Mische" die sortierten Hälften zusammen

Achtung: "Mische" bedeutet hier "sortiert zusammenfügen"!



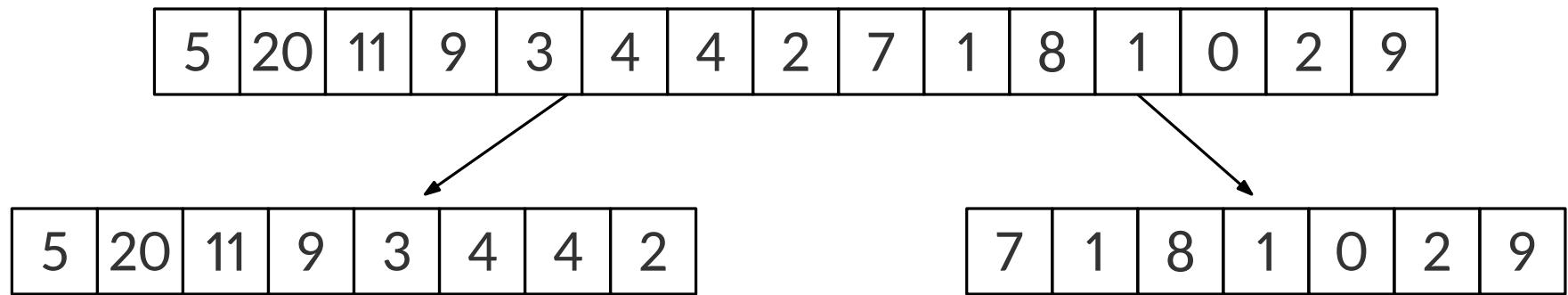
# Merge-Schritt: Idee/Beispiel

---

5	20	11	9	3	4	4	2	7	1	8	1	0	2	9
---	----	----	---	---	---	---	---	---	---	---	---	---	---	---

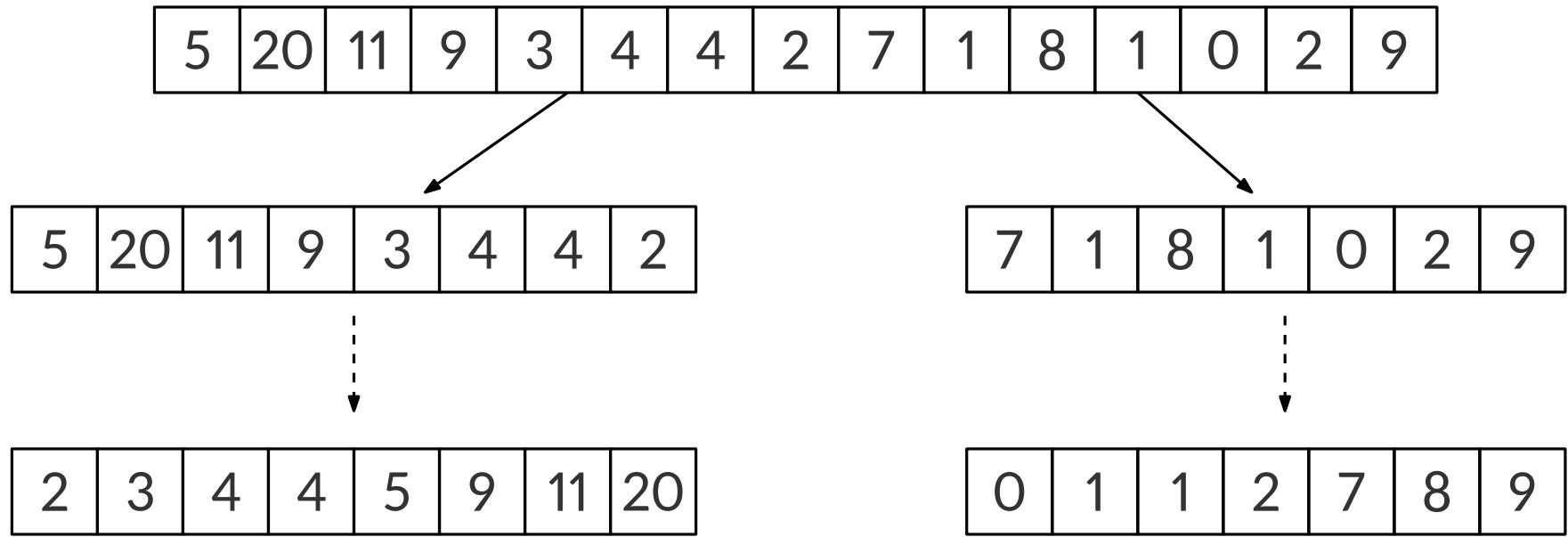
# Merge-Schritt: Idee/Beispiel

---

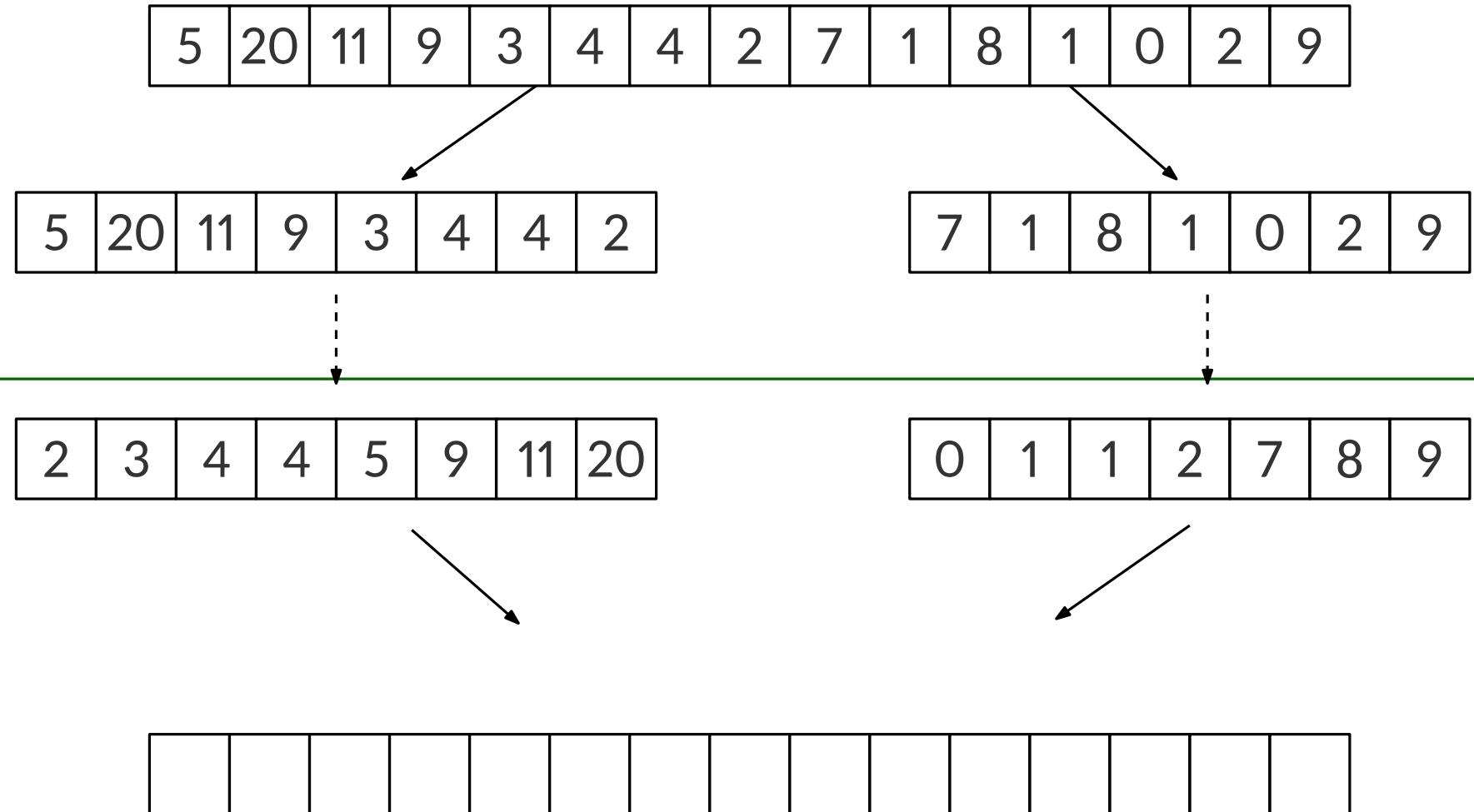


# Merge-Schritt: Idee/Beispiel

---

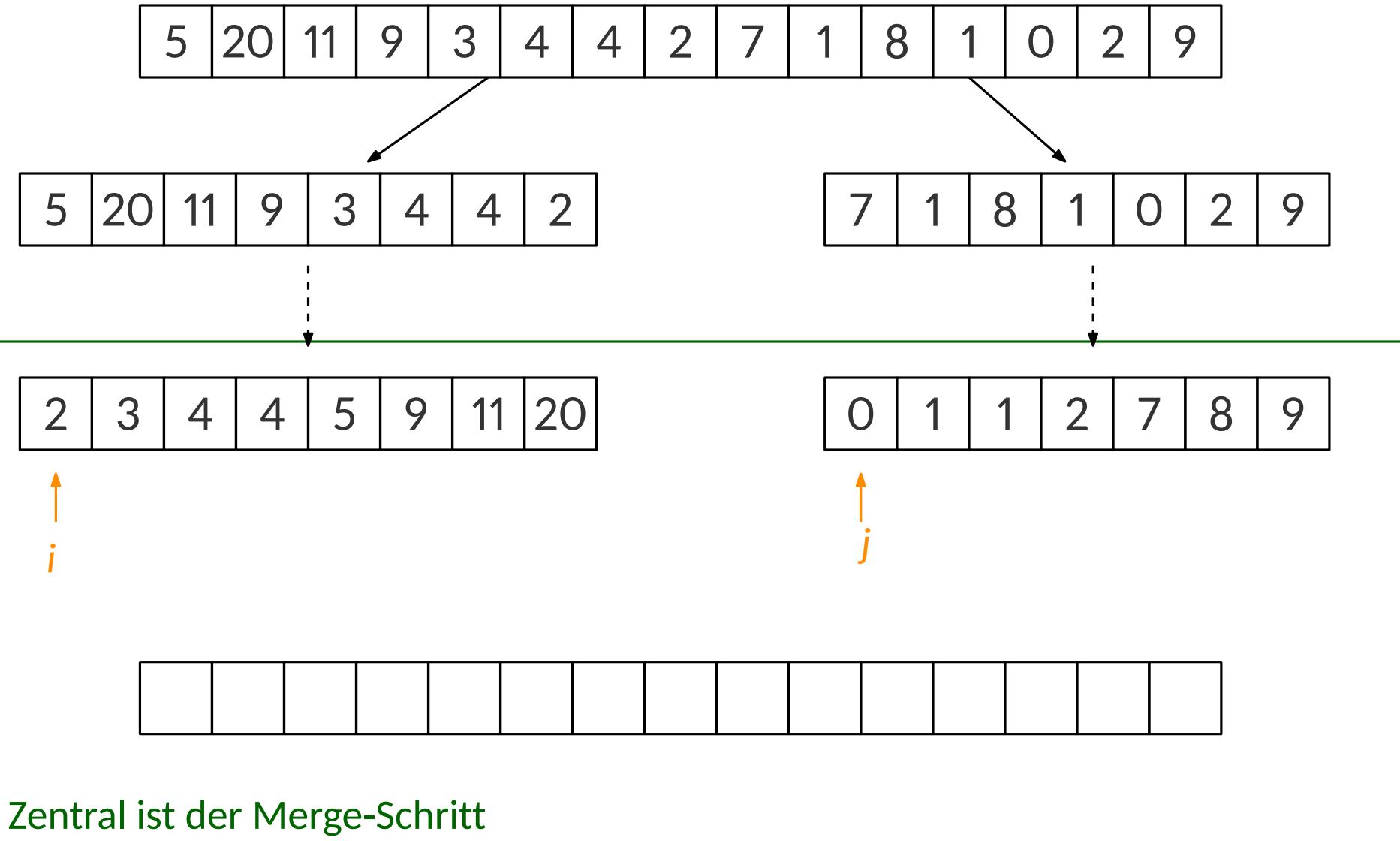


# Merge-Schritt: Idee/Beispiel

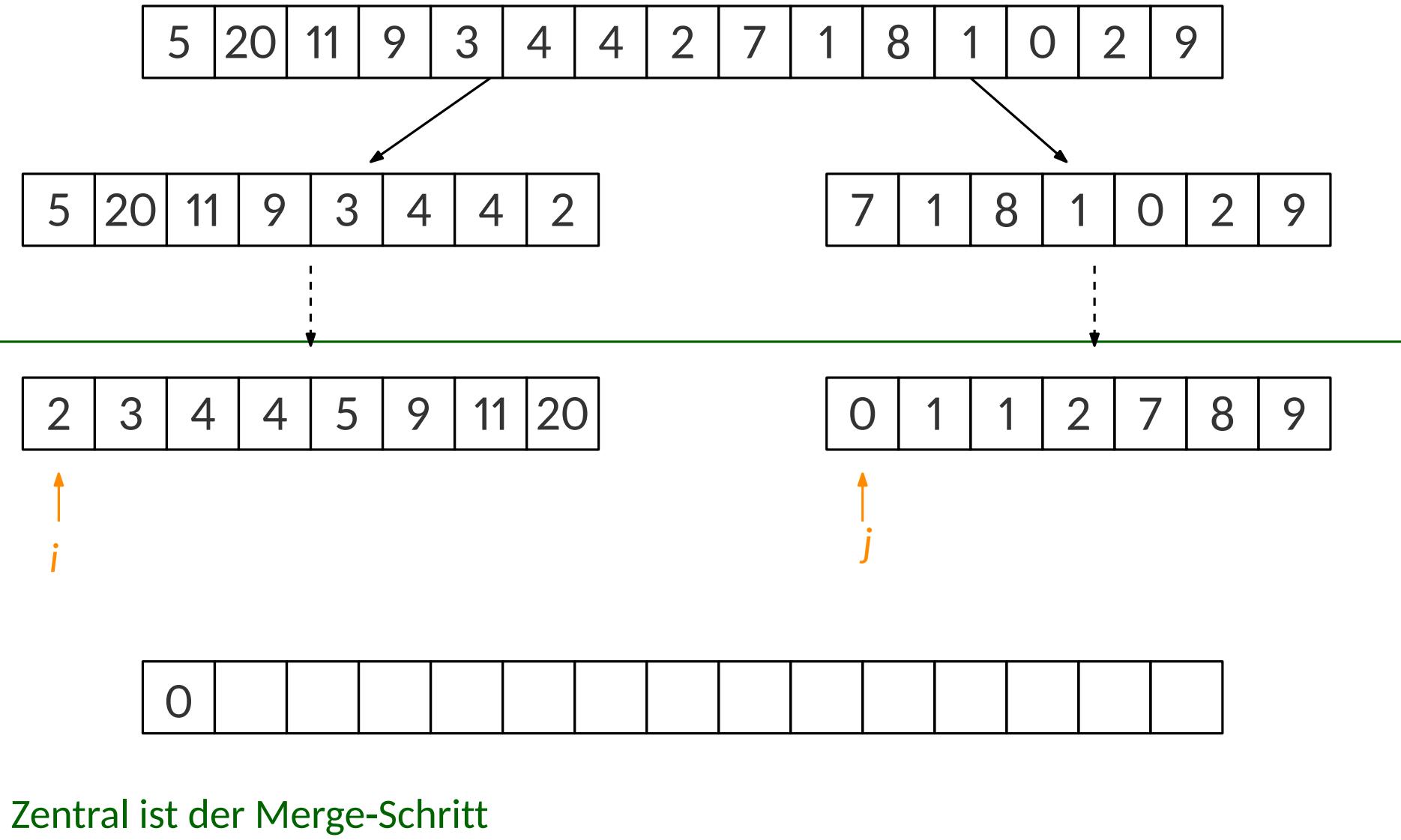


Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel

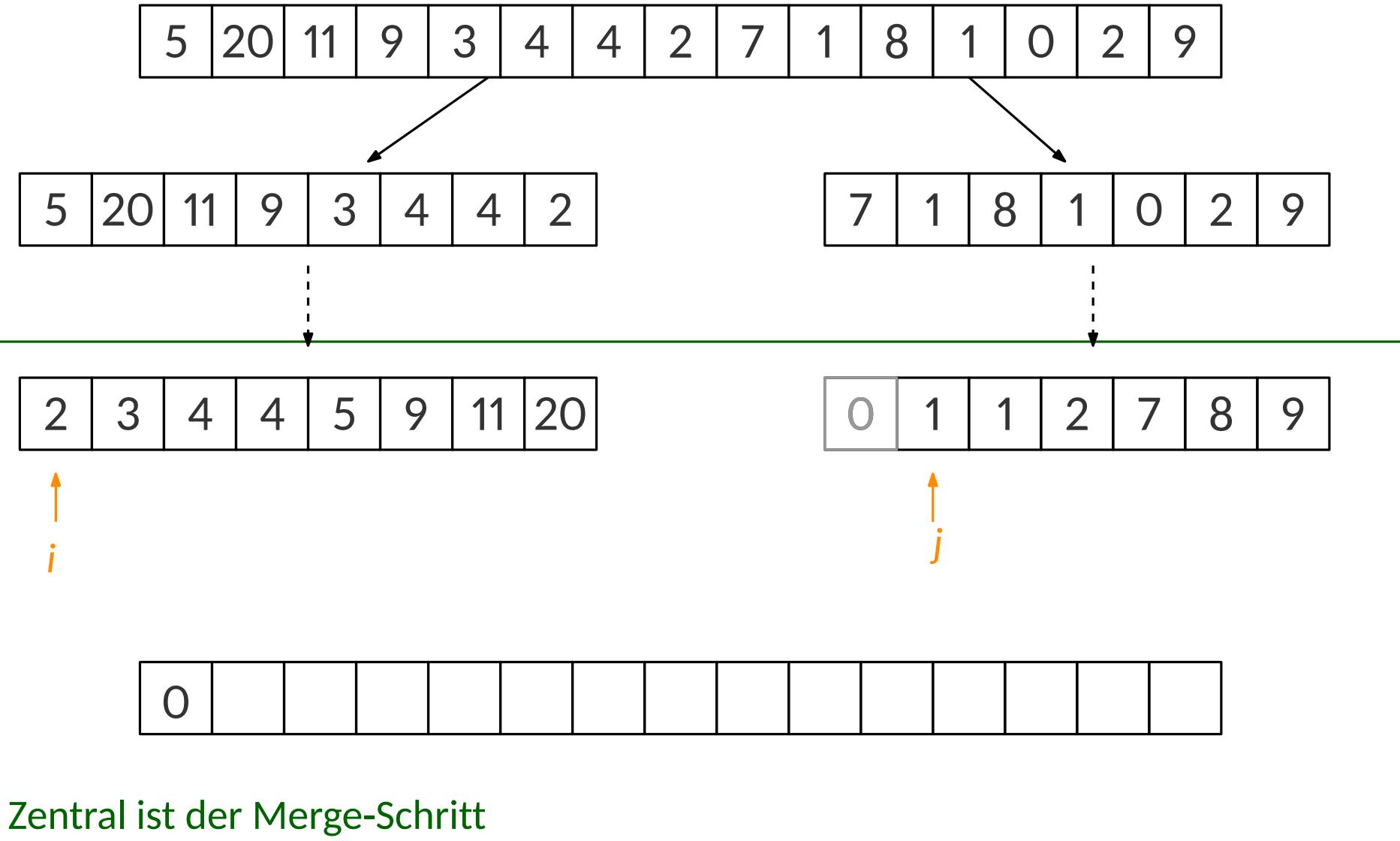


# Merge-Schritt: Idee/Beispiel

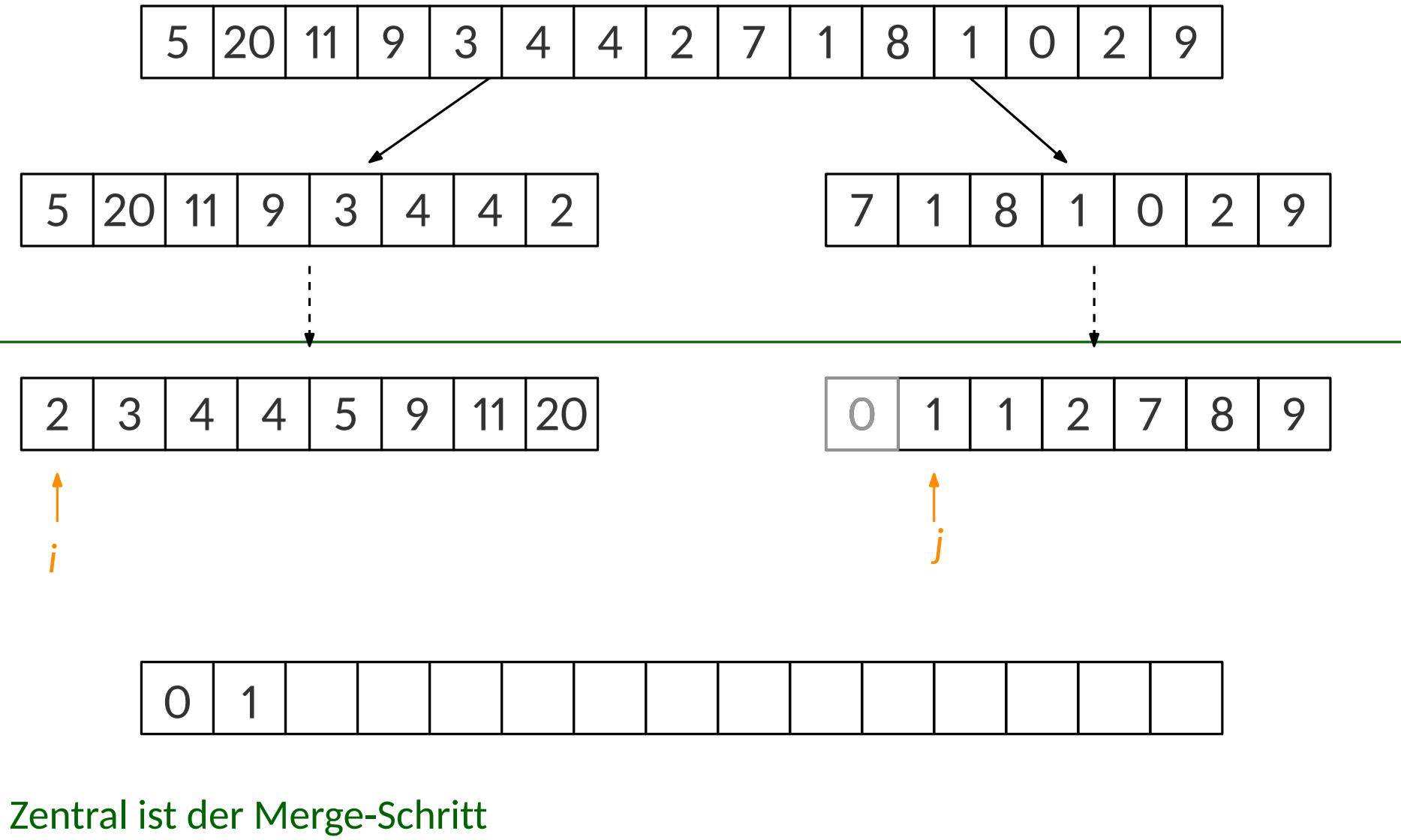


Zentral ist der Merge-Schritt

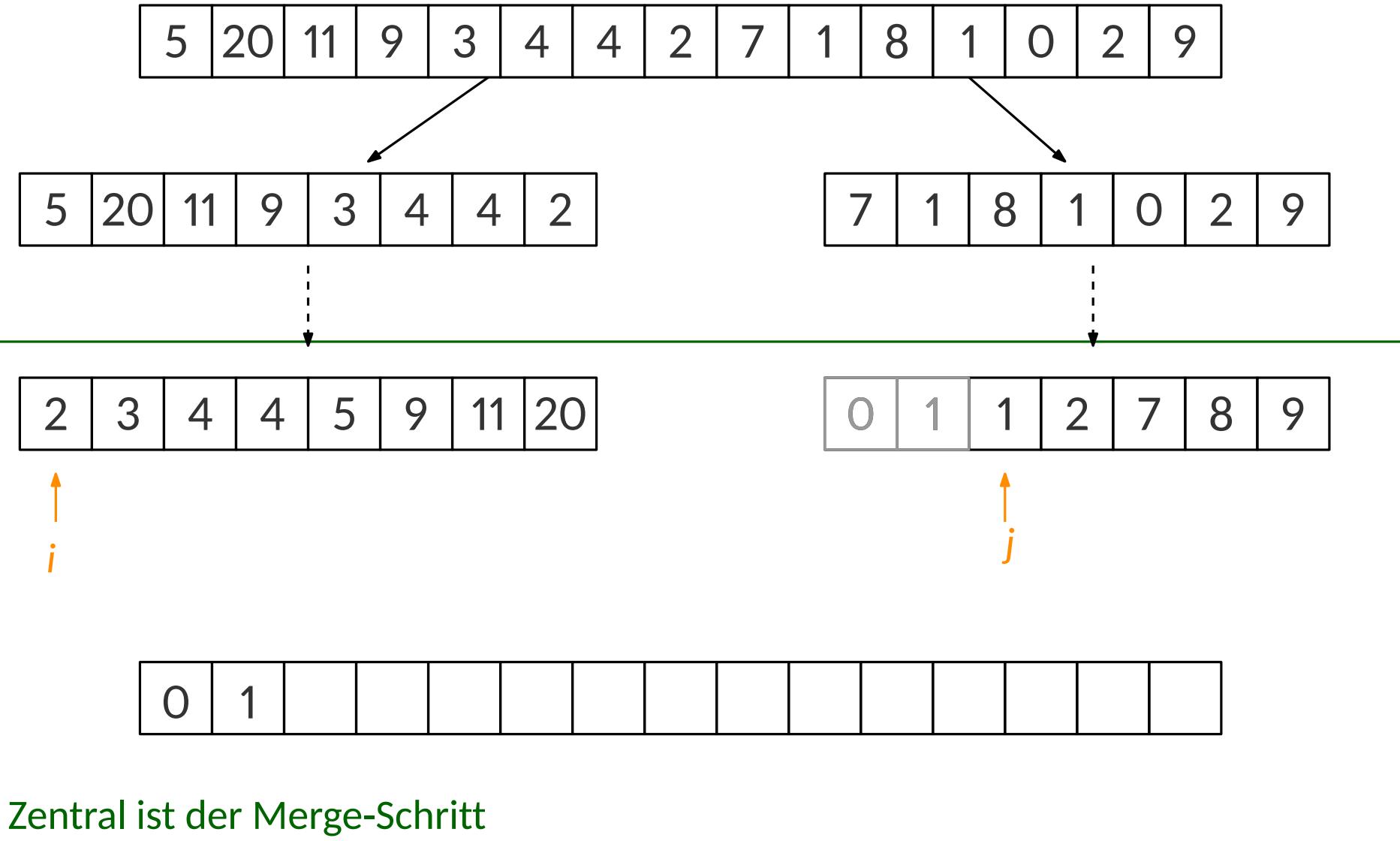
# Merge-Schritt: Idee/Beispiel



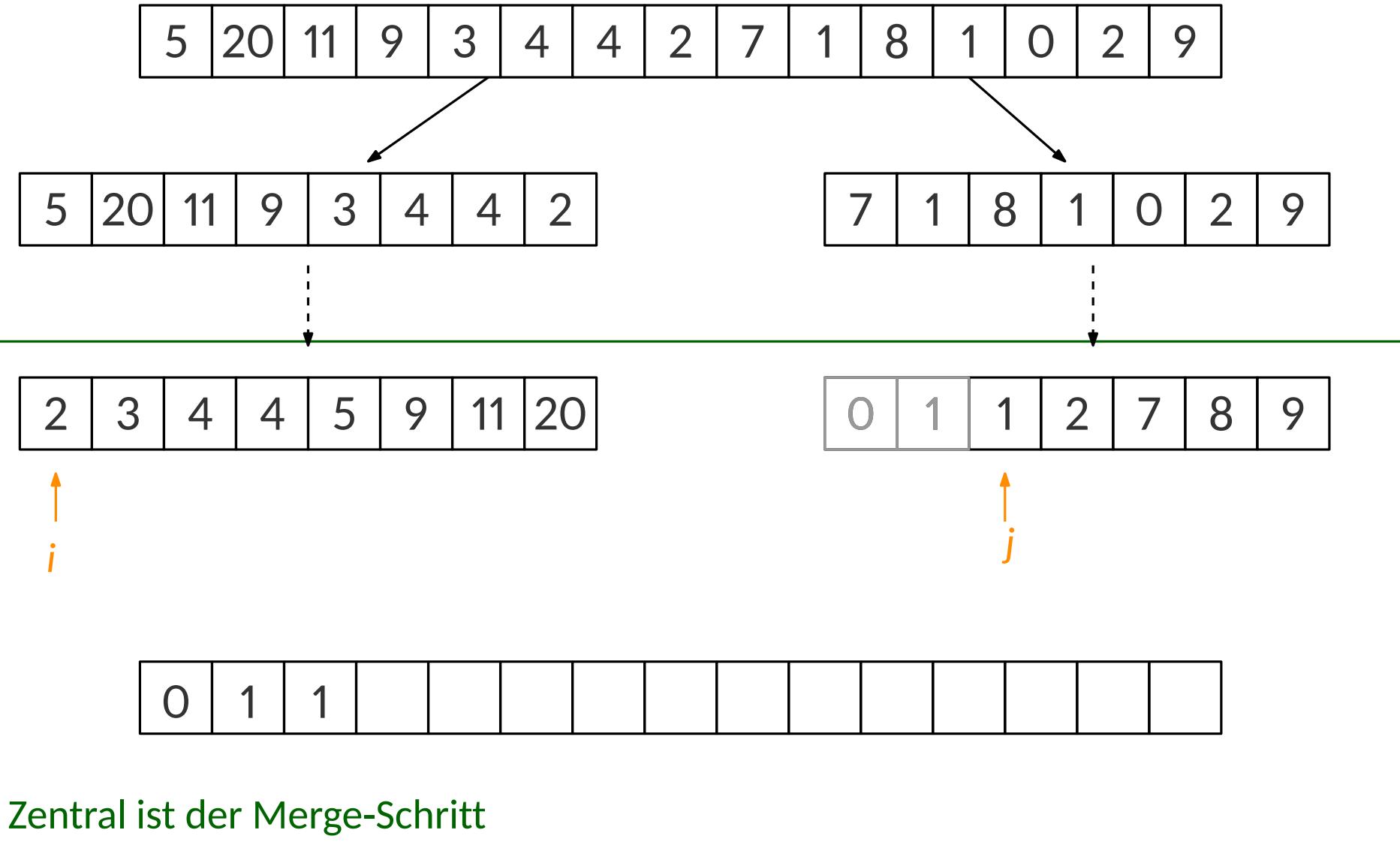
# Merge-Schritt: Idee/Beispiel



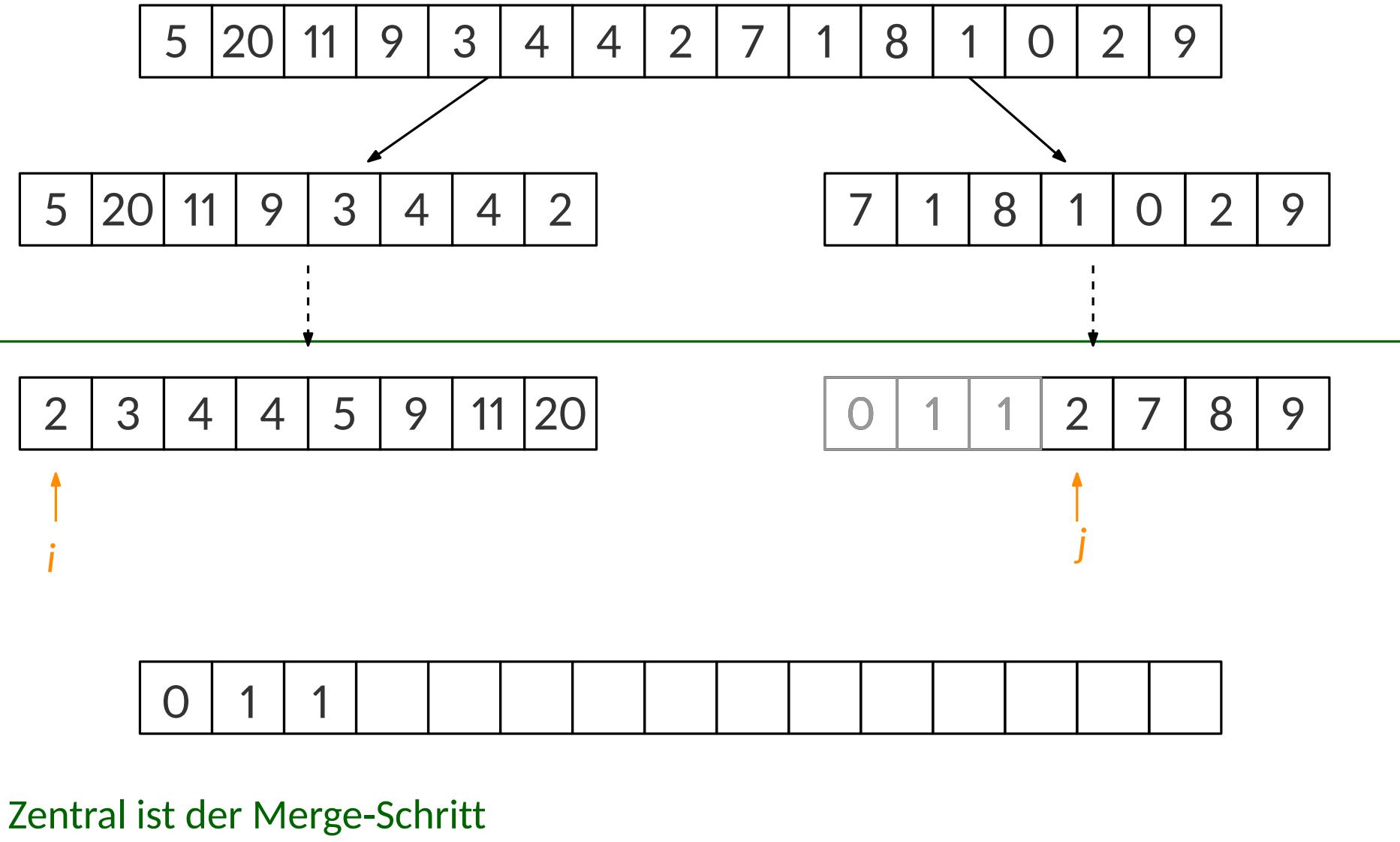
# Merge-Schritt: Idee/Beispiel



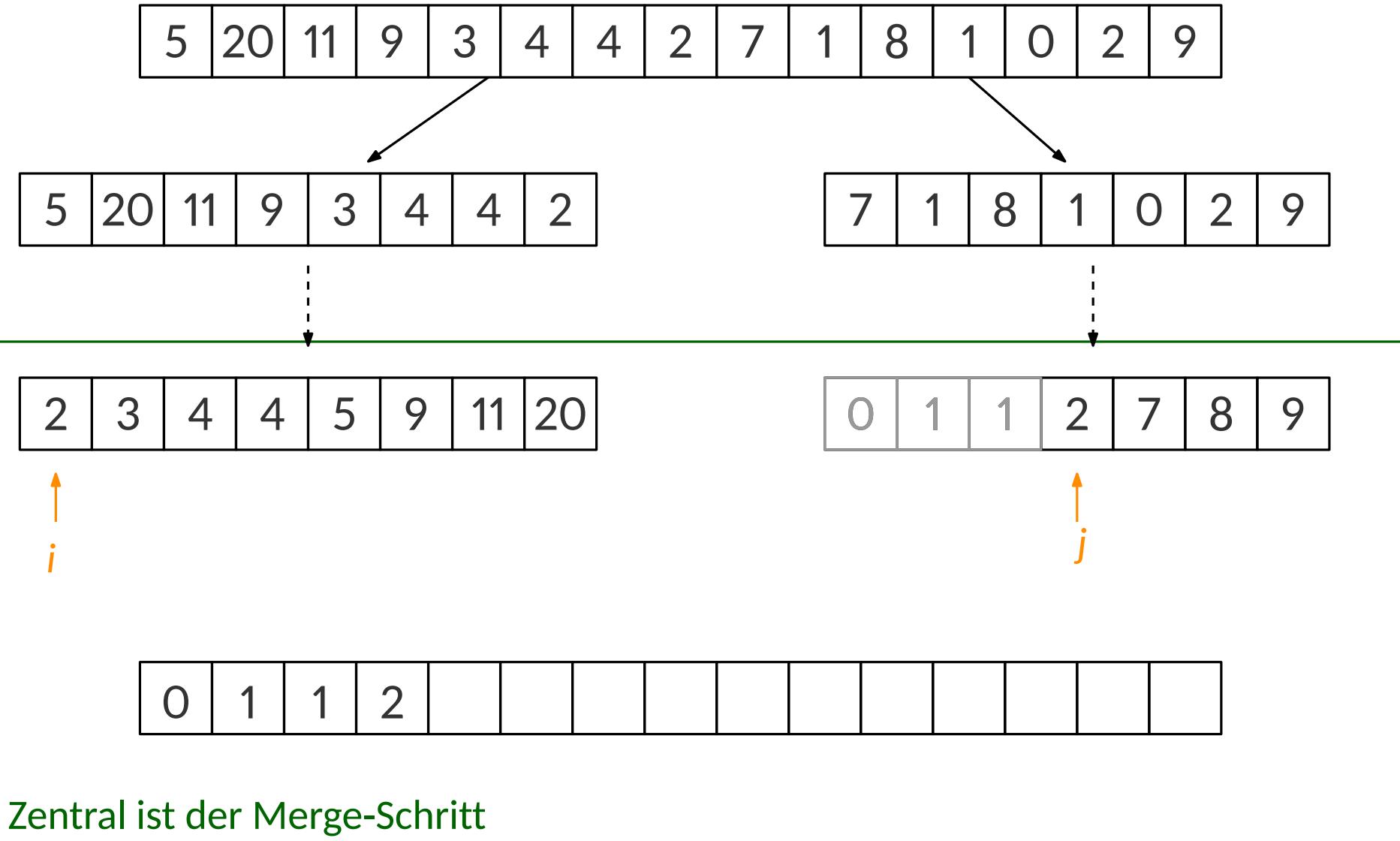
# Merge-Schritt: Idee/Beispiel



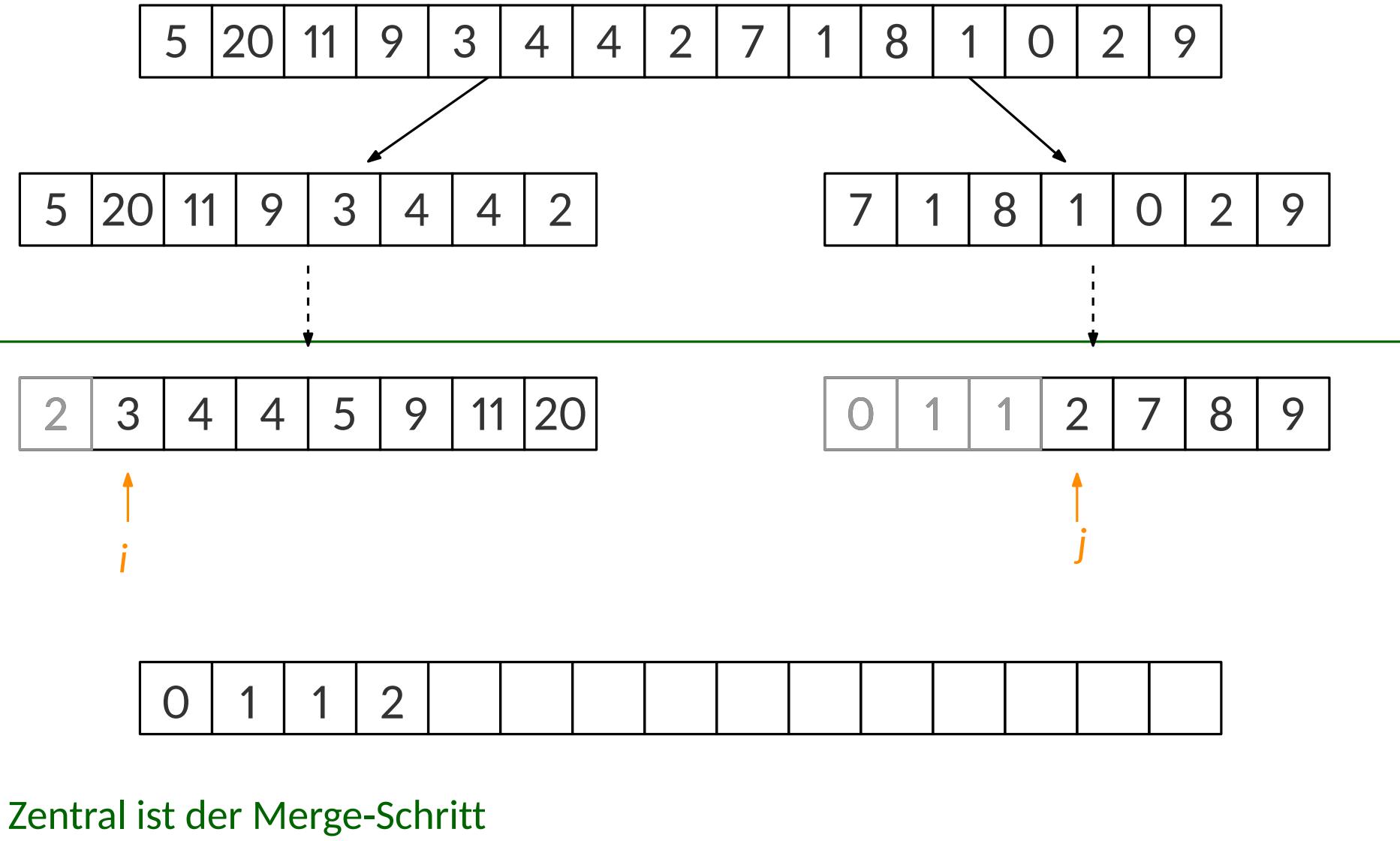
# Merge-Schritt: Idee/Beispiel



# Merge-Schritt: Idee/Beispiel

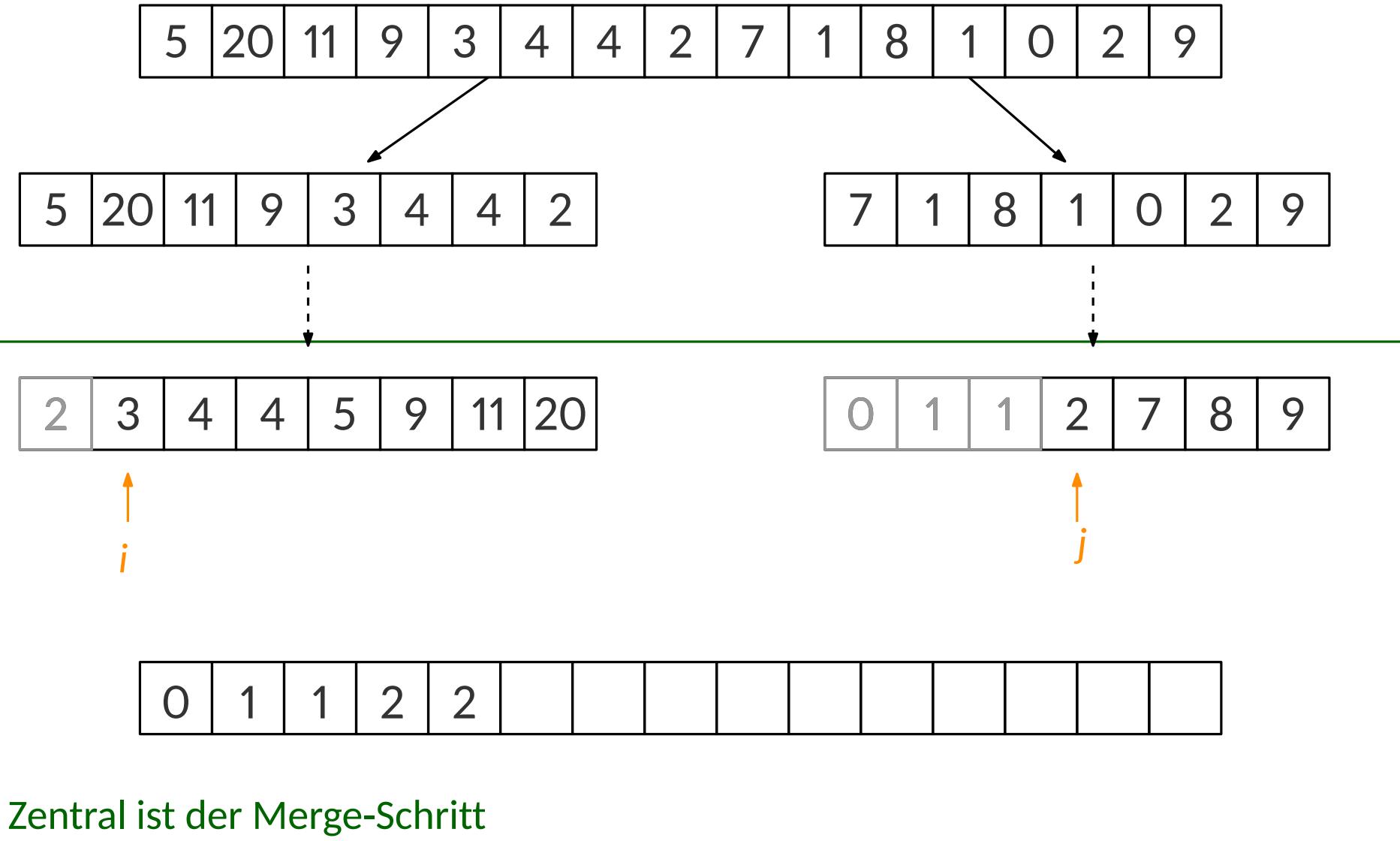


# Merge-Schritt: Idee/Beispiel

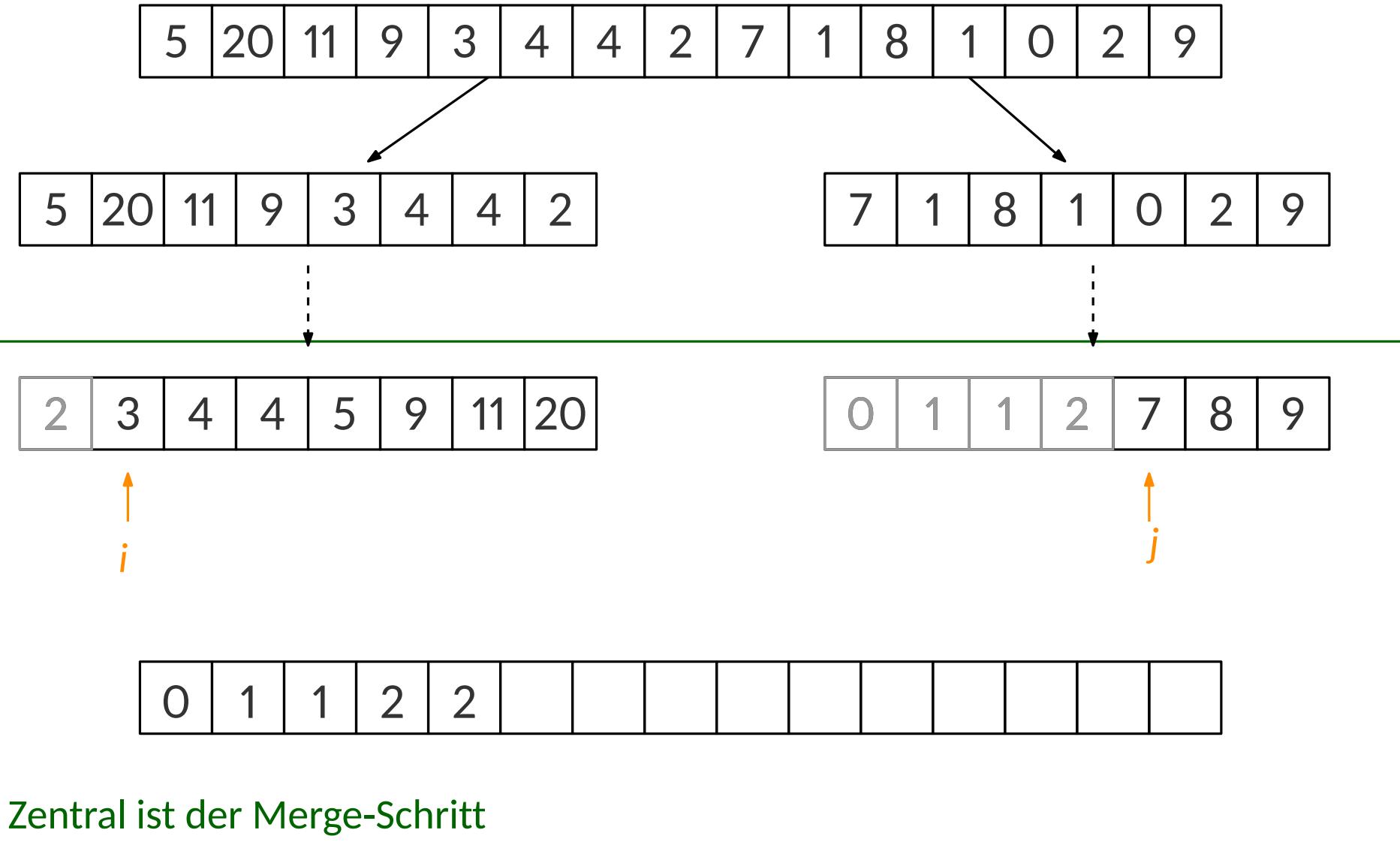


Zentral ist der Merge-Schritt

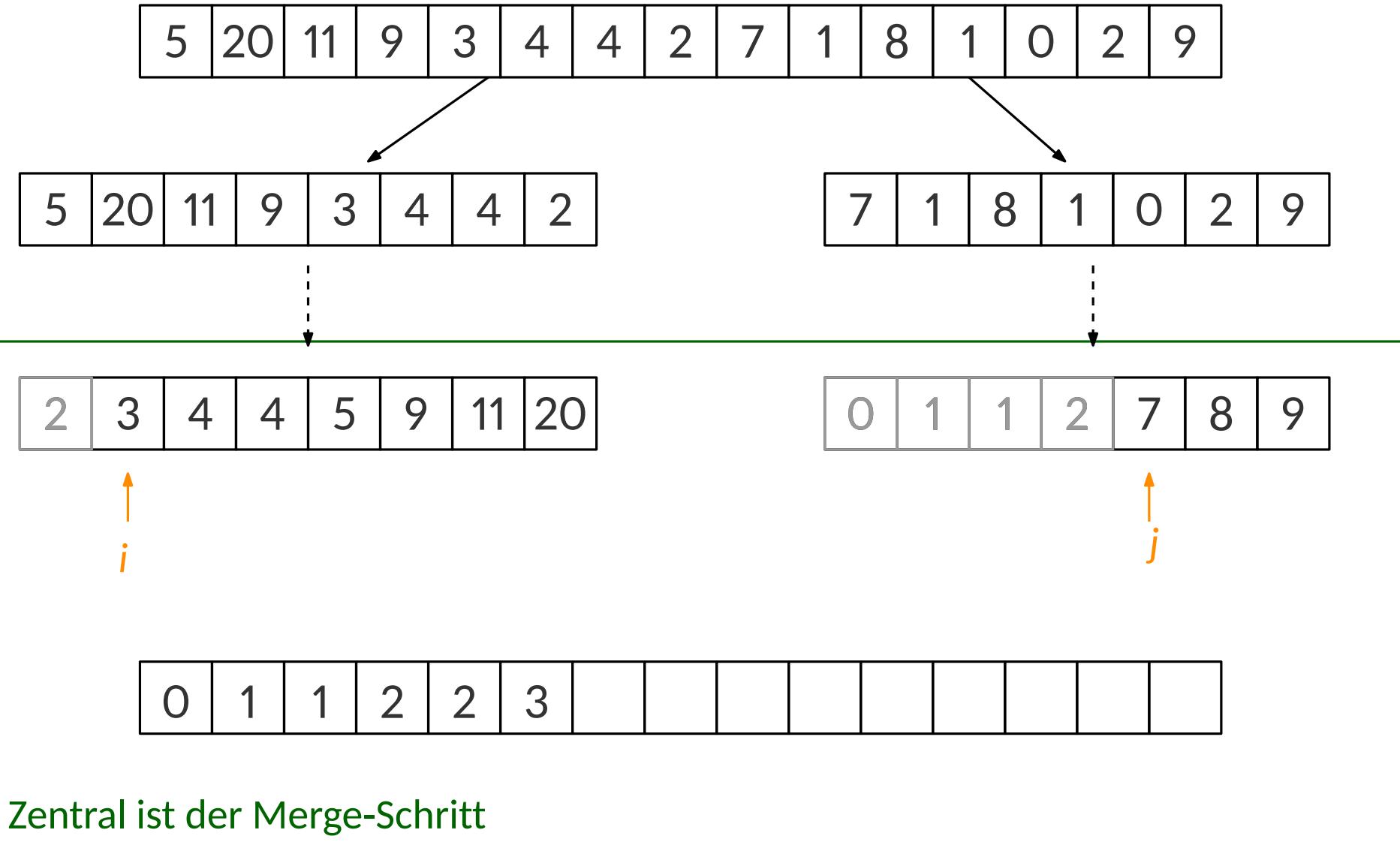
# Merge-Schritt: Idee/Beispiel



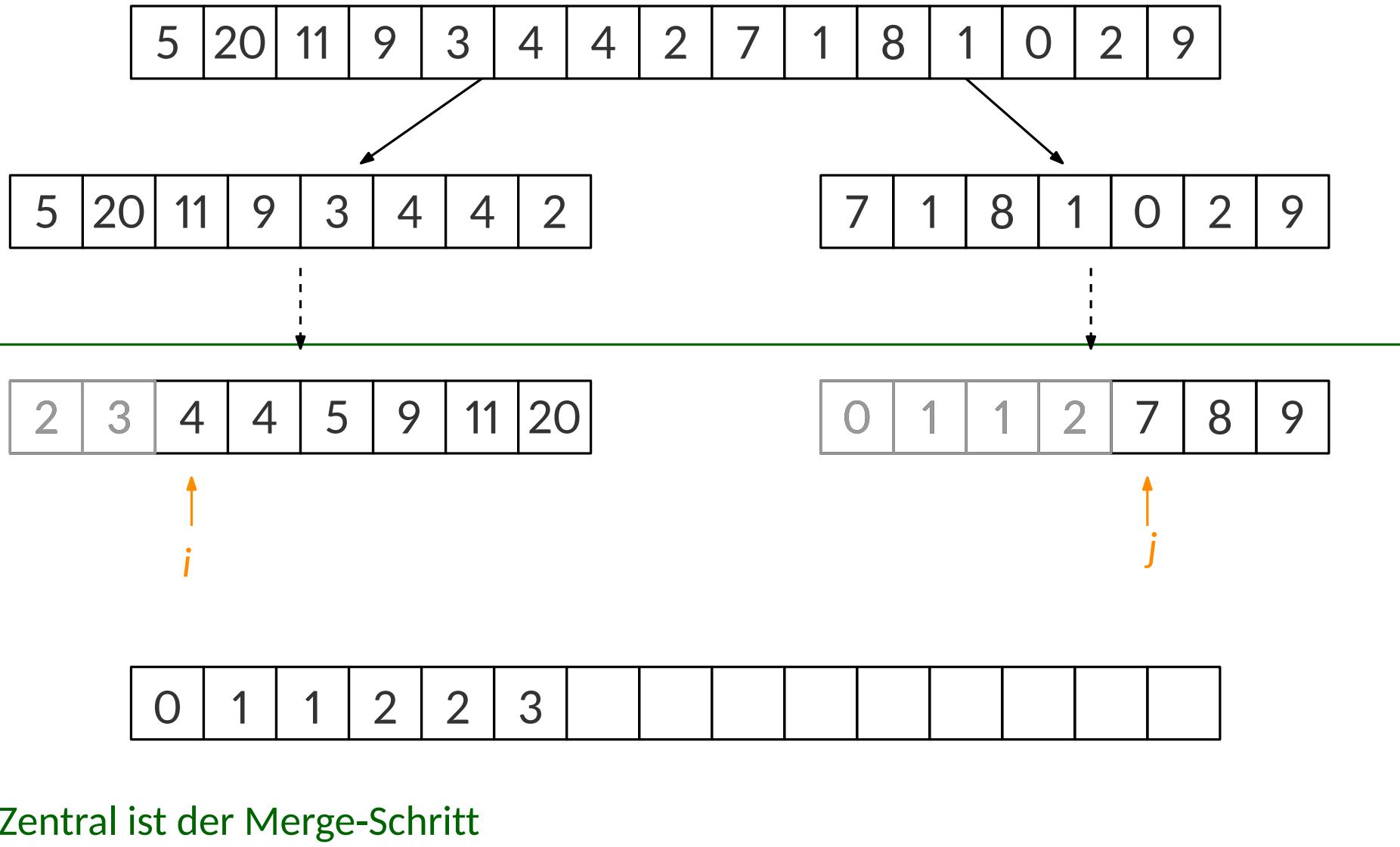
# Merge-Schritt: Idee/Beispiel



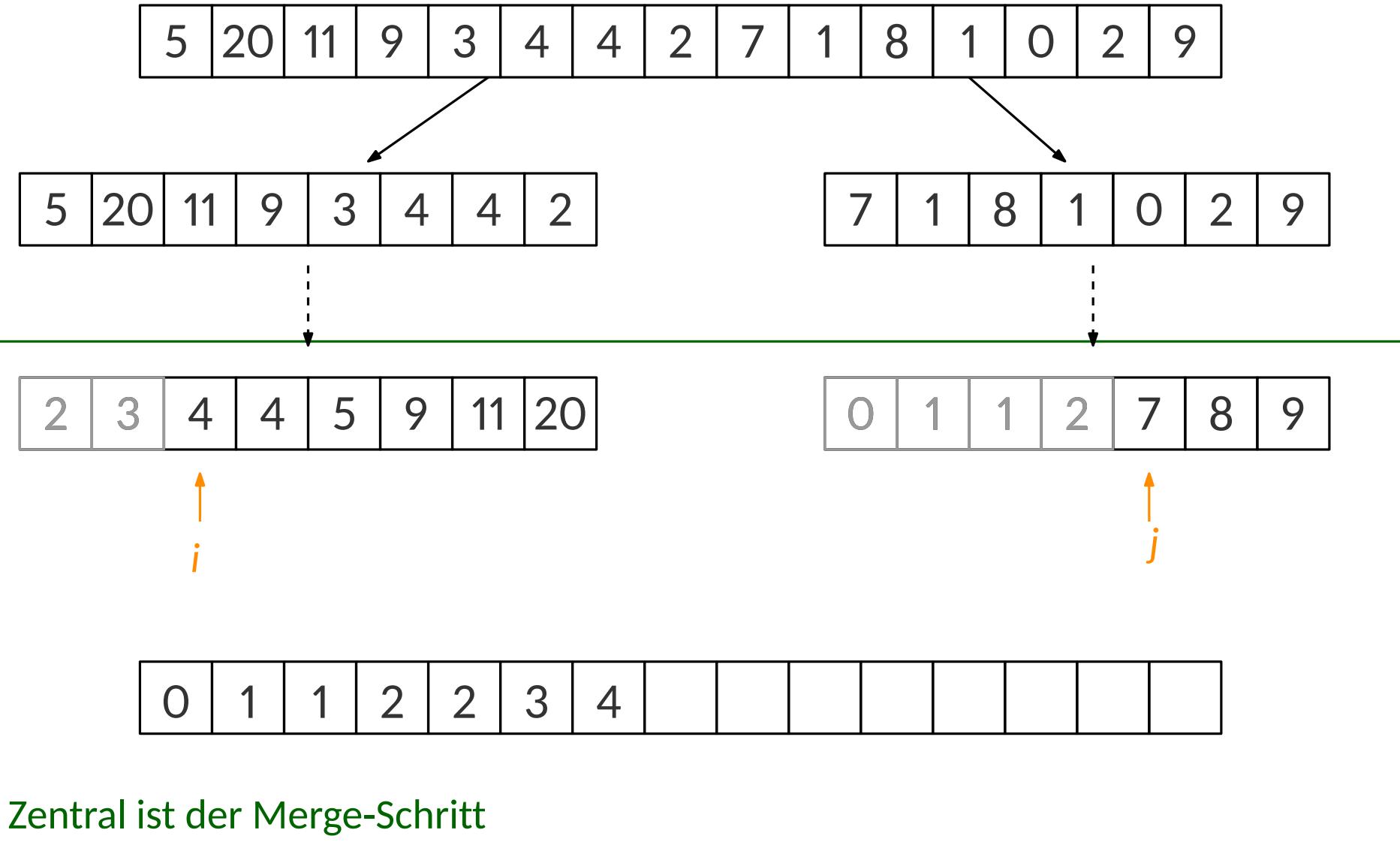
# Merge-Schritt: Idee/Beispiel



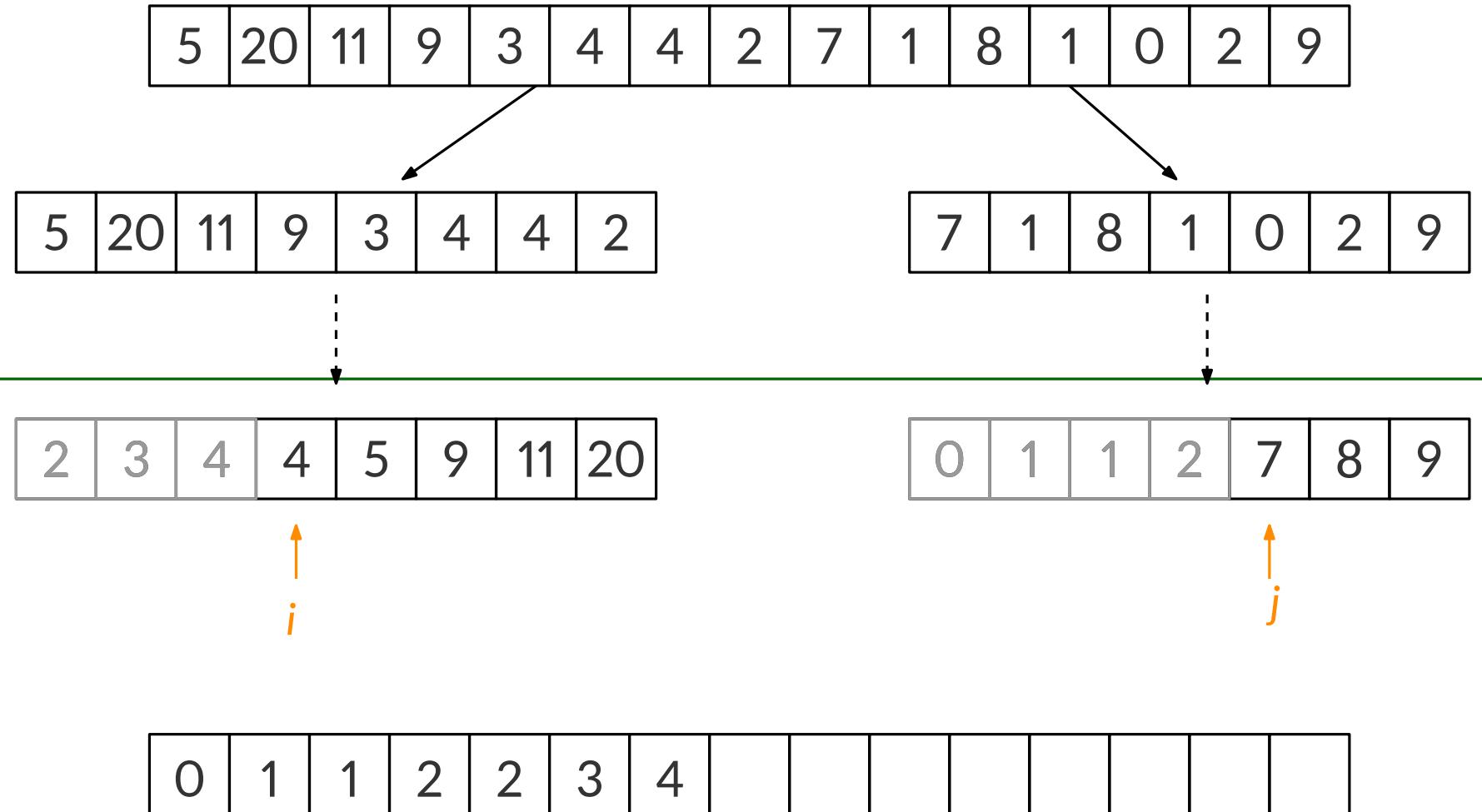
# Merge-Schritt: Idee/Beispiel



# Merge-Schritt: Idee/Beispiel

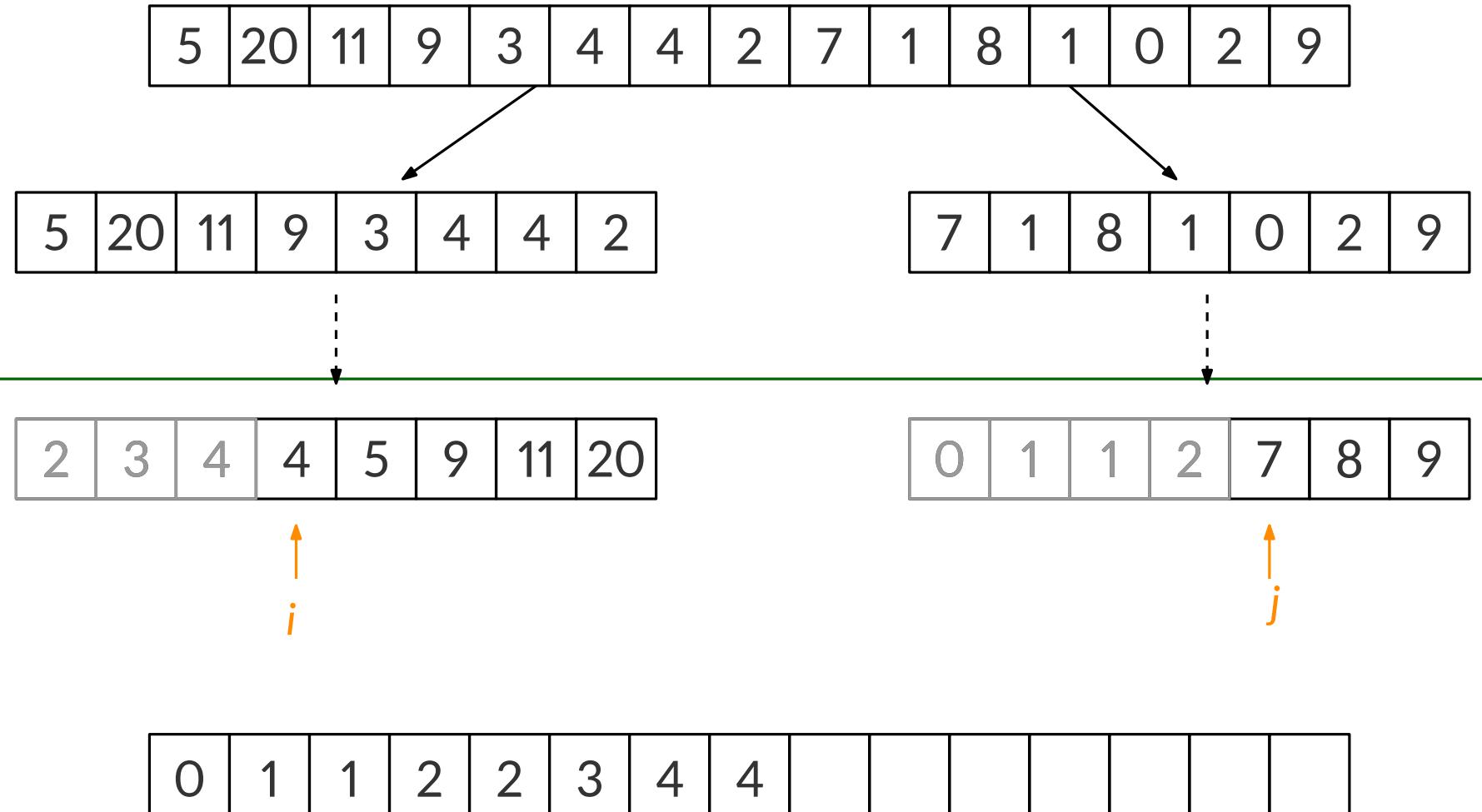


# Merge-Schritt: Idee/Beispiel



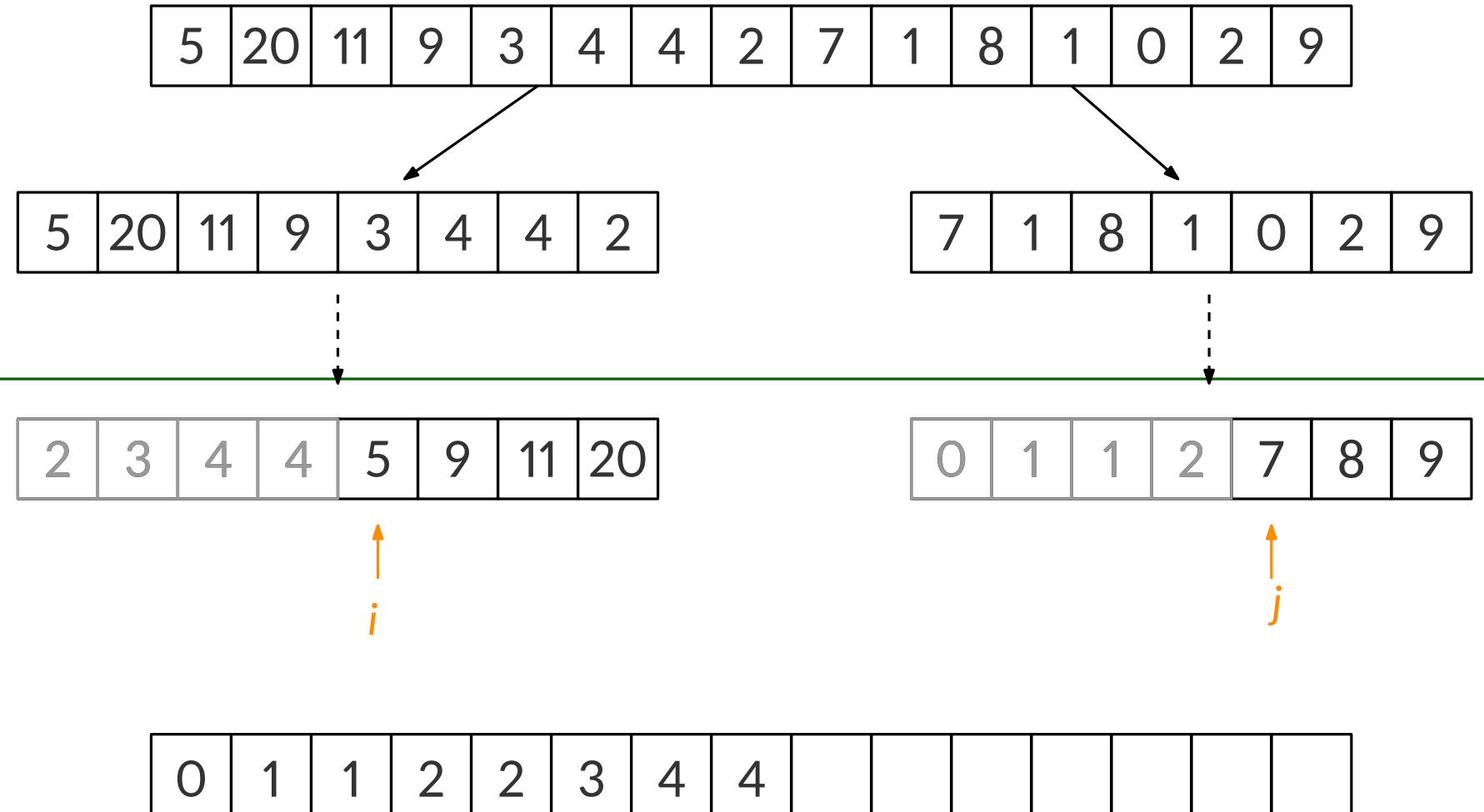
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



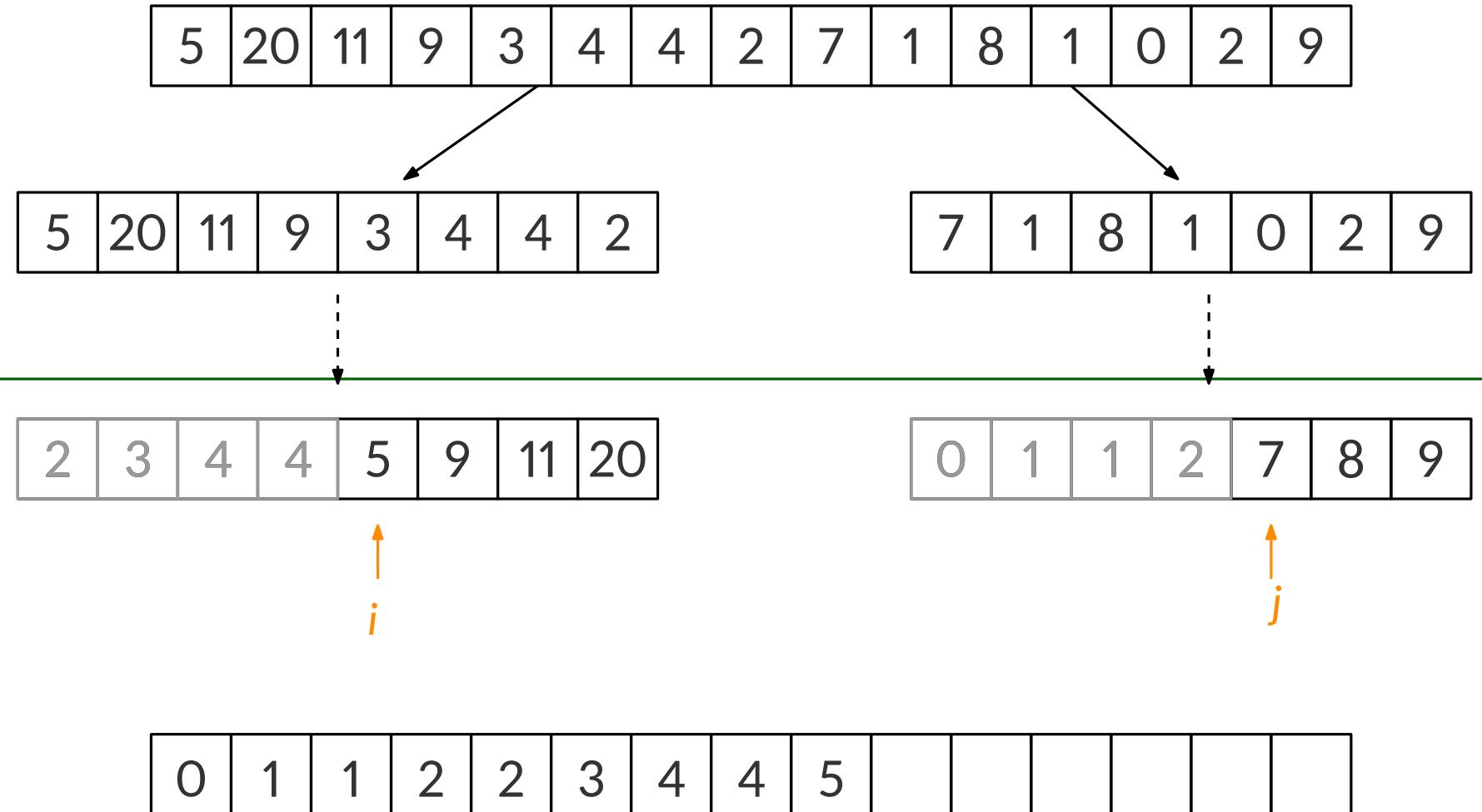
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



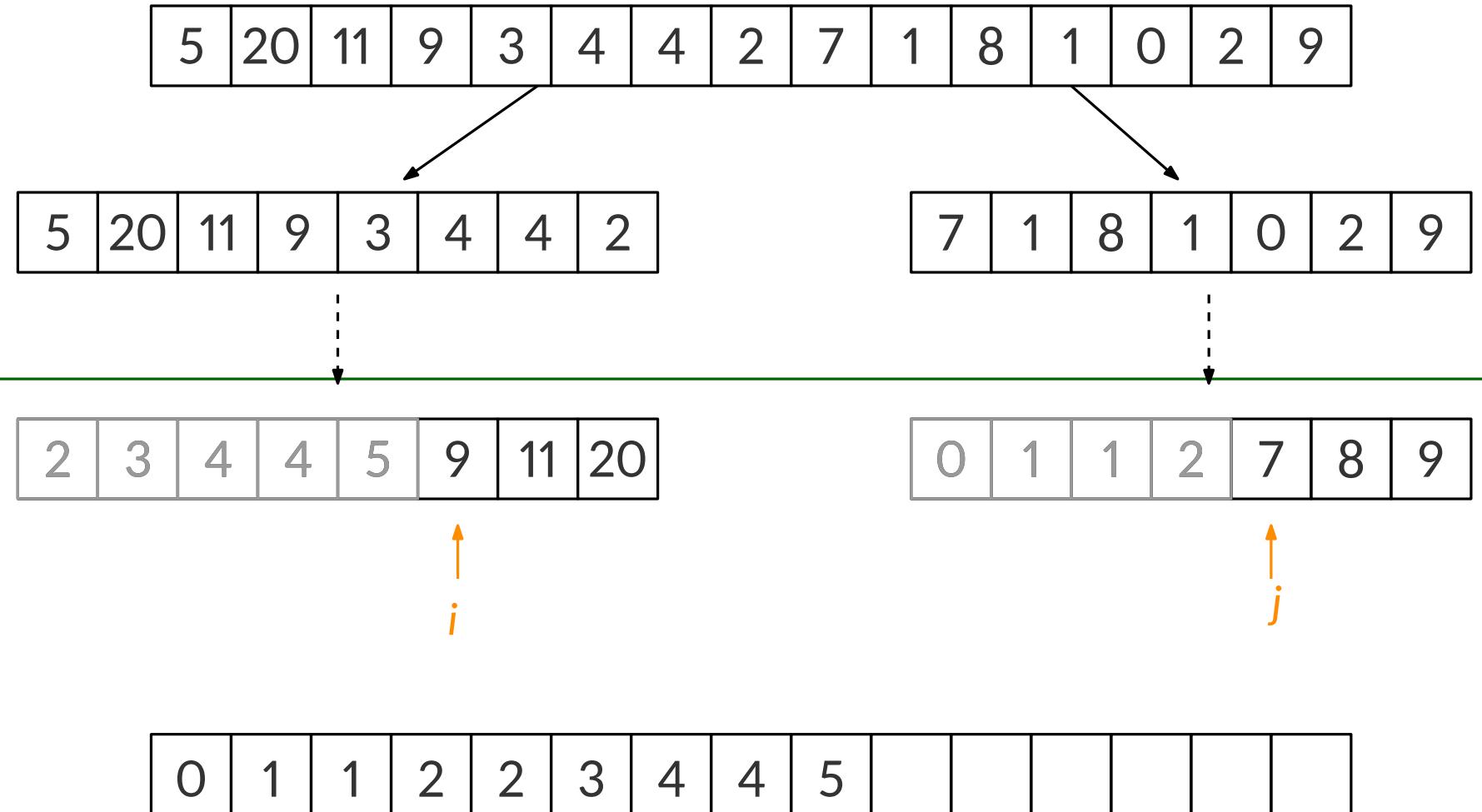
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



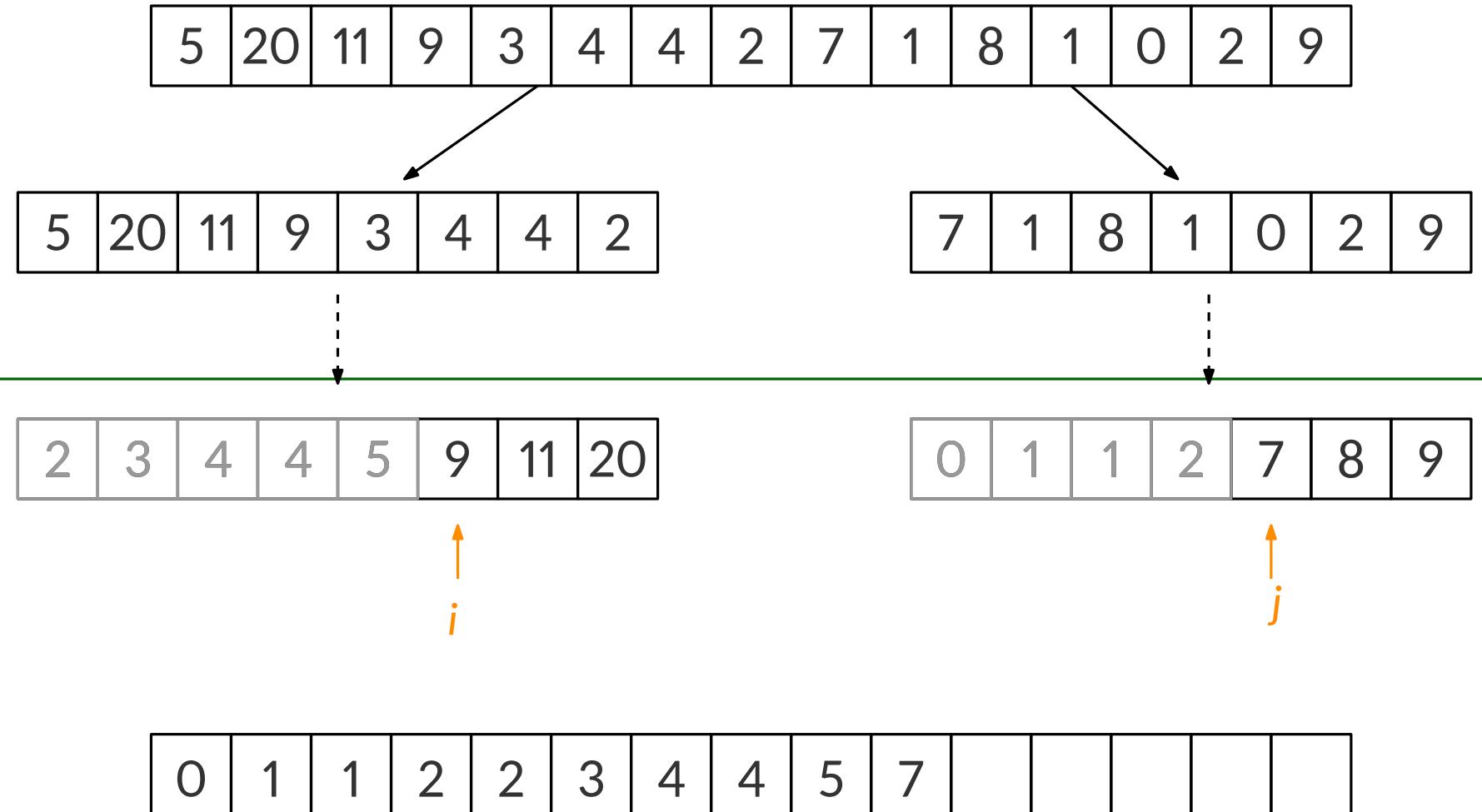
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



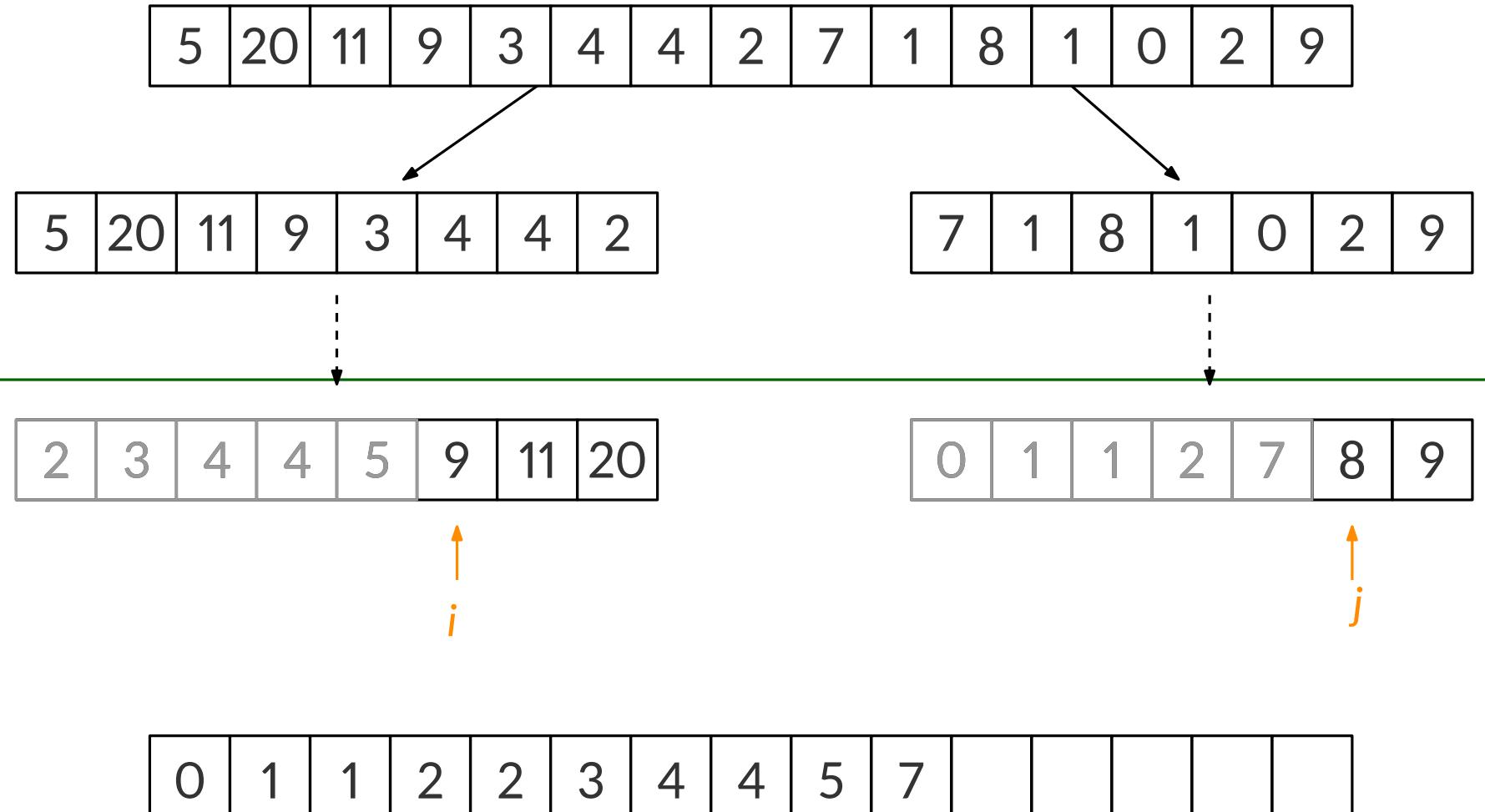
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



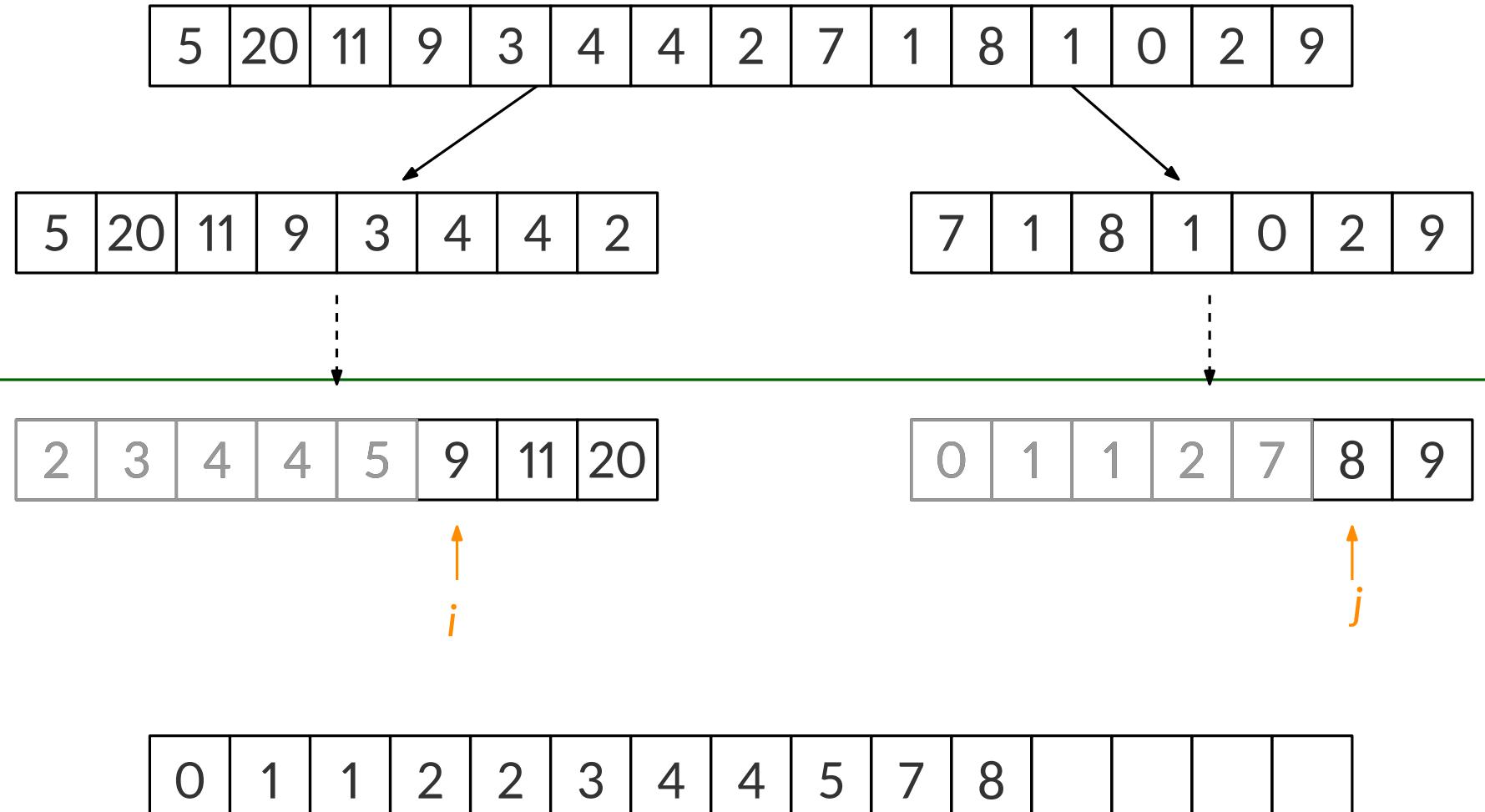
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



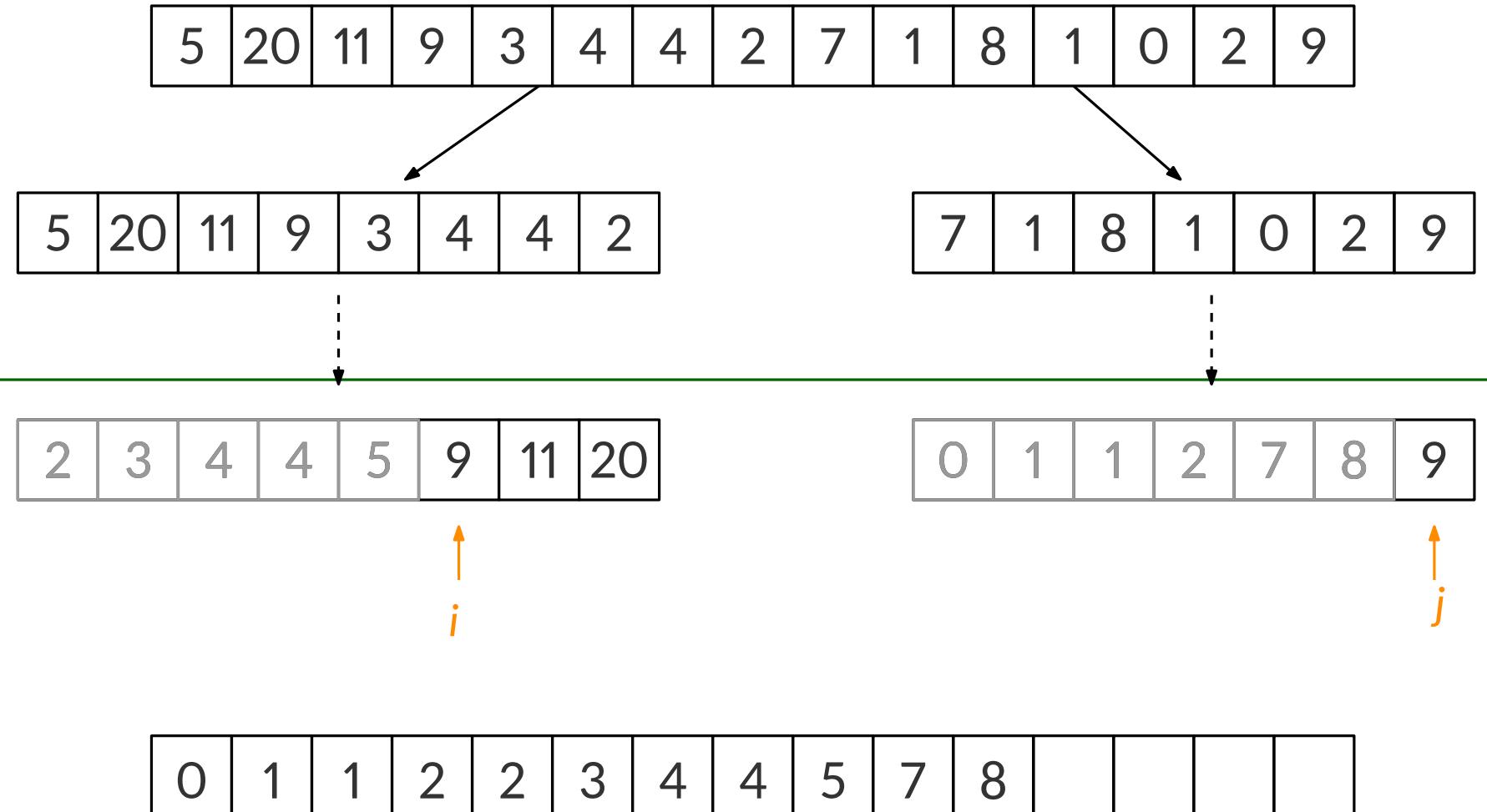
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



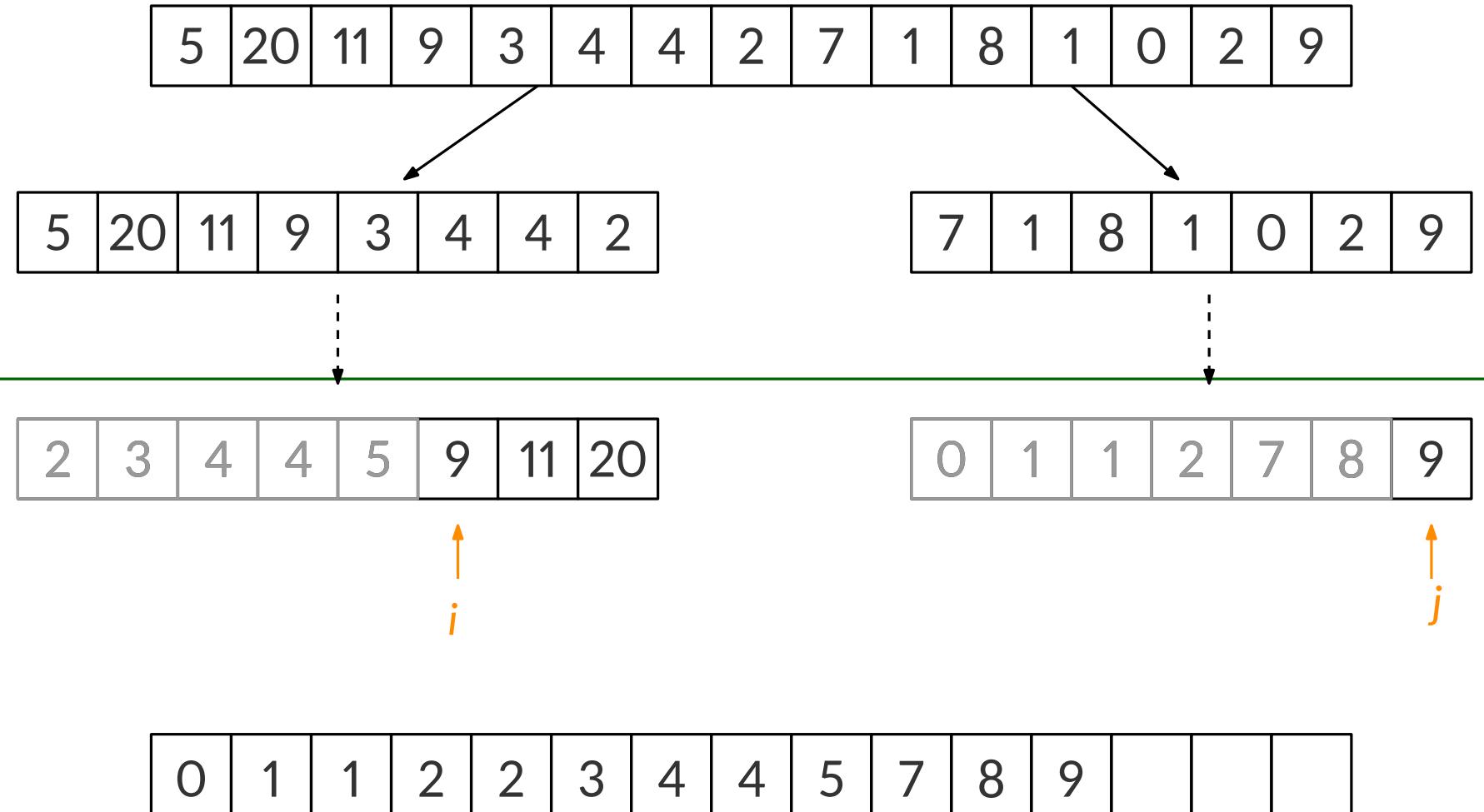
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



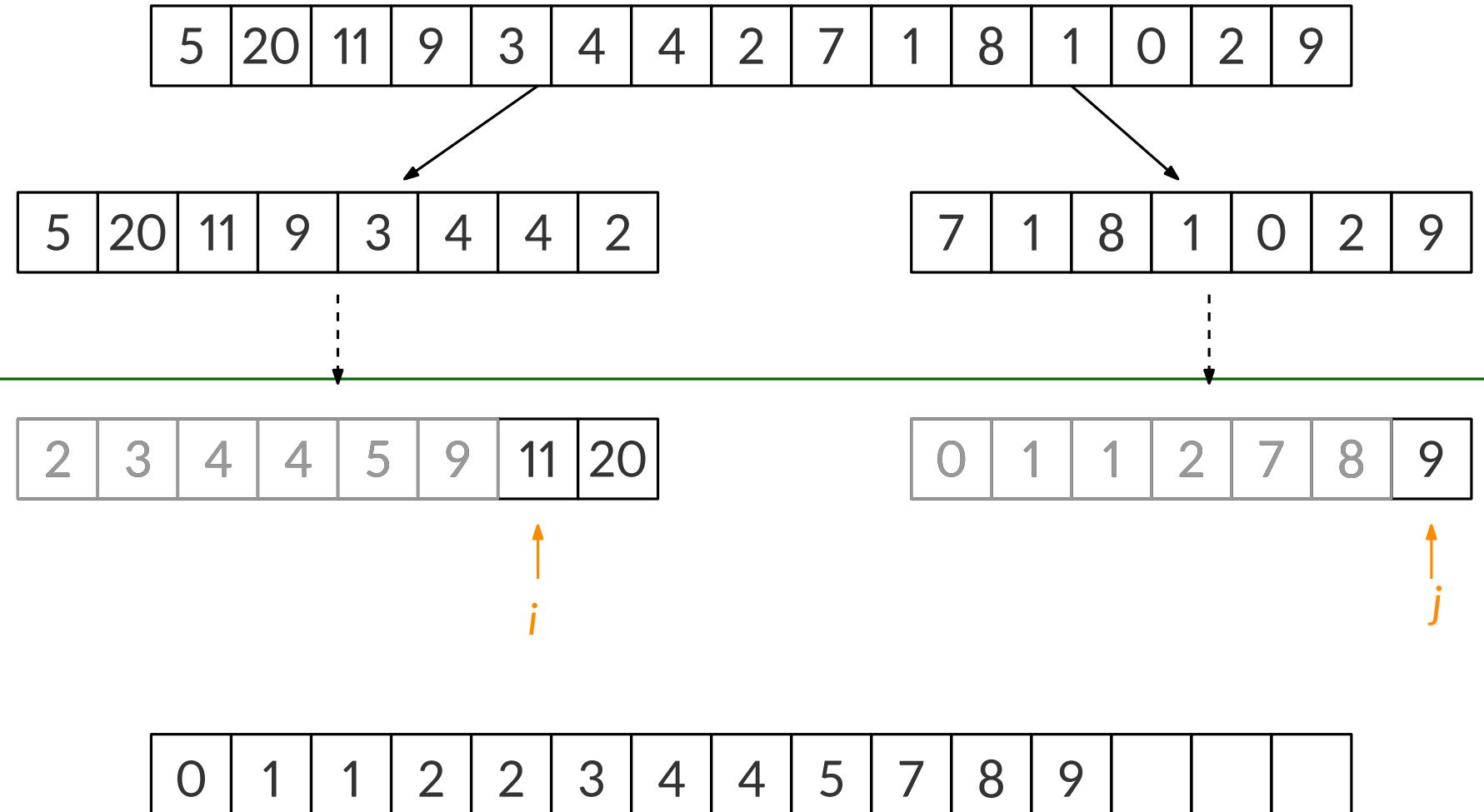
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



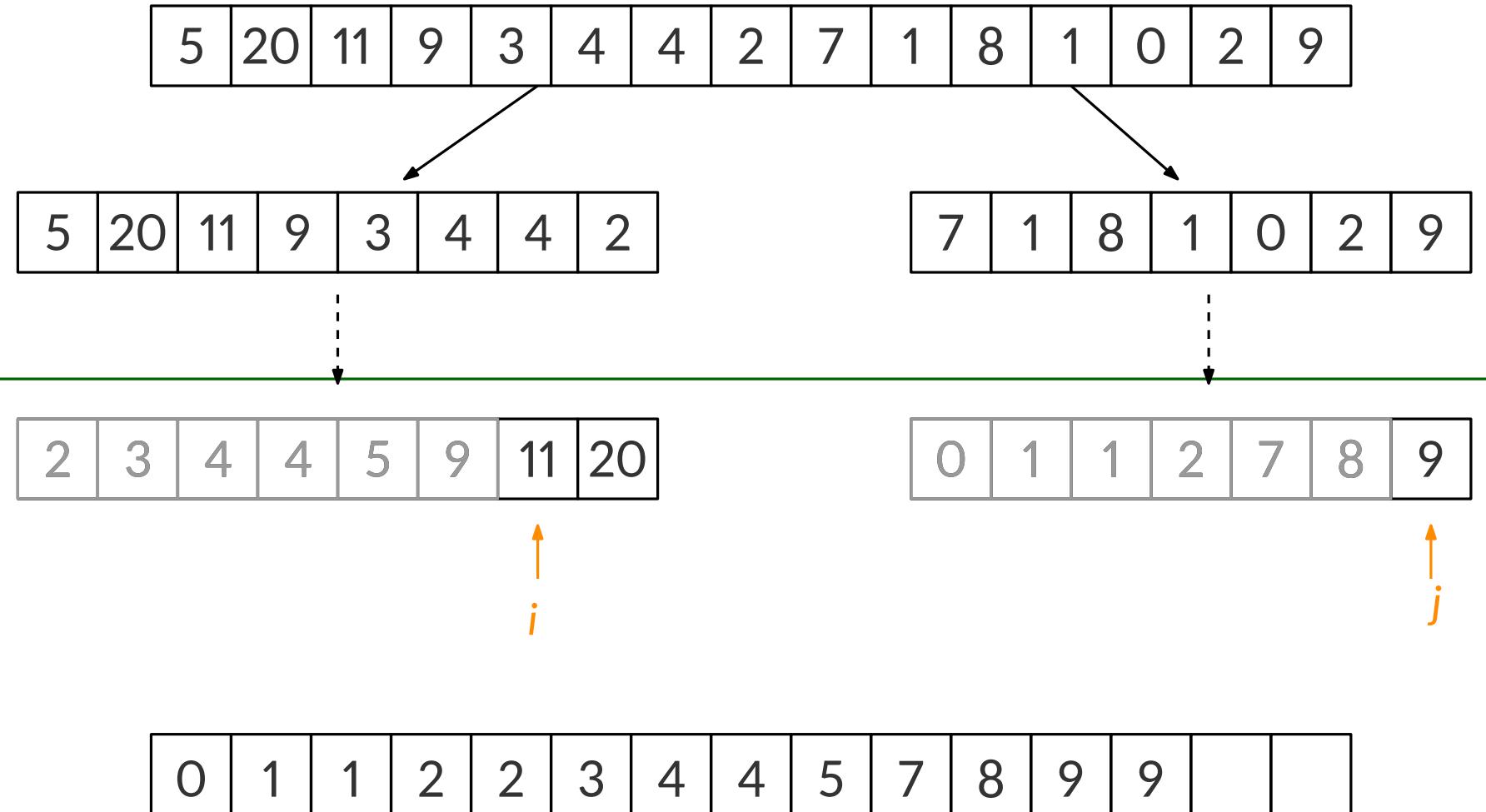
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



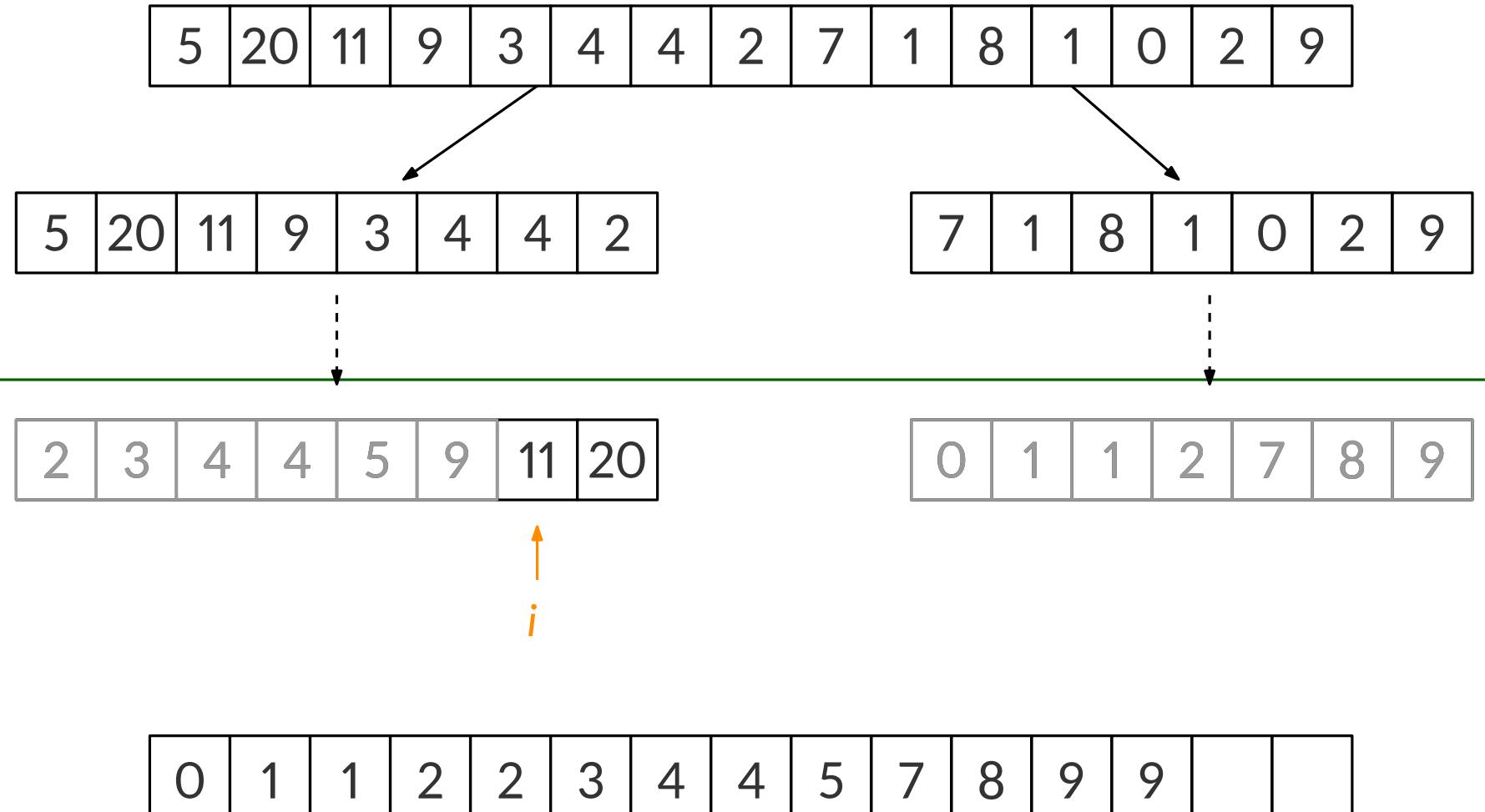
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



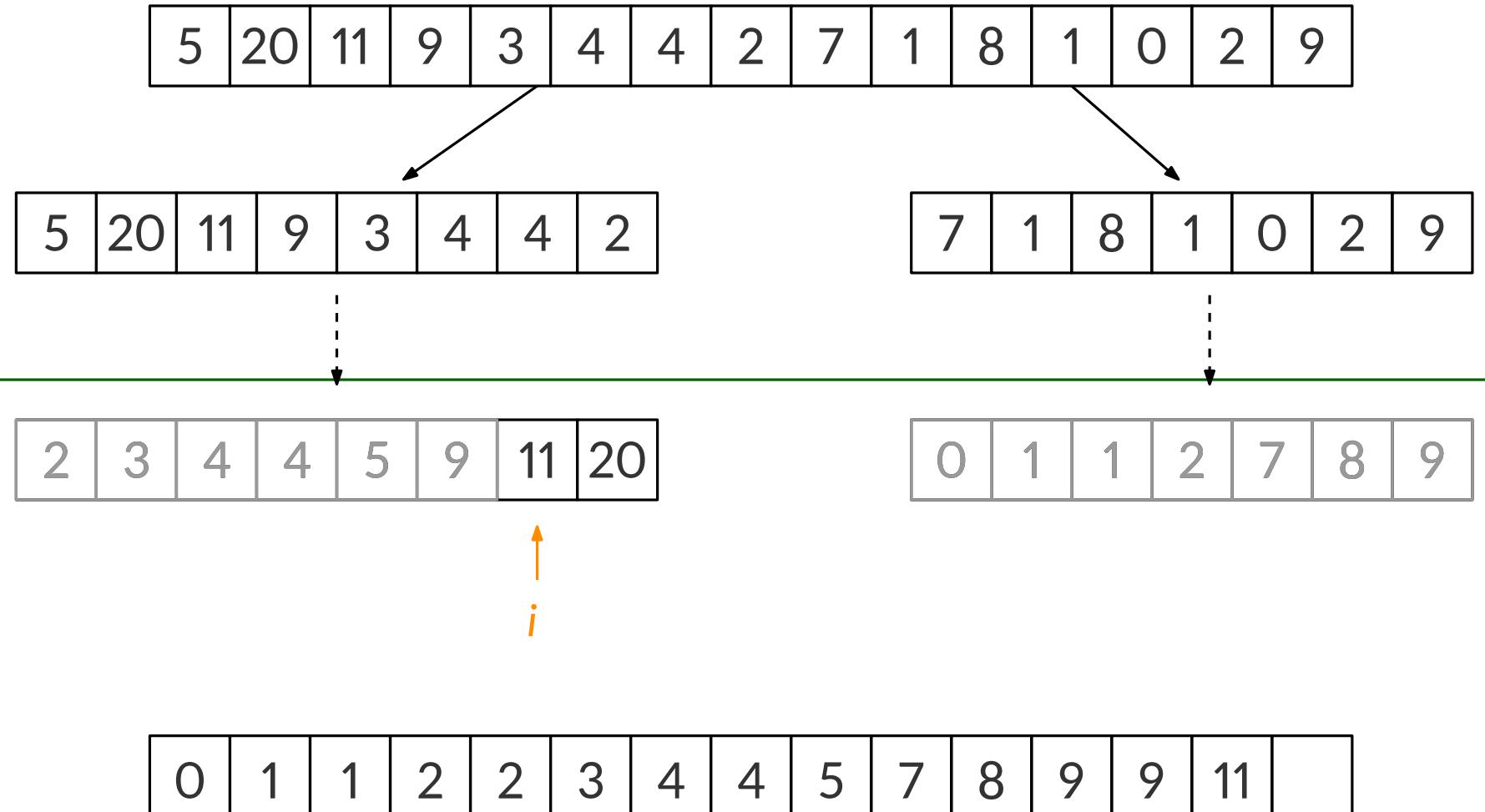
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



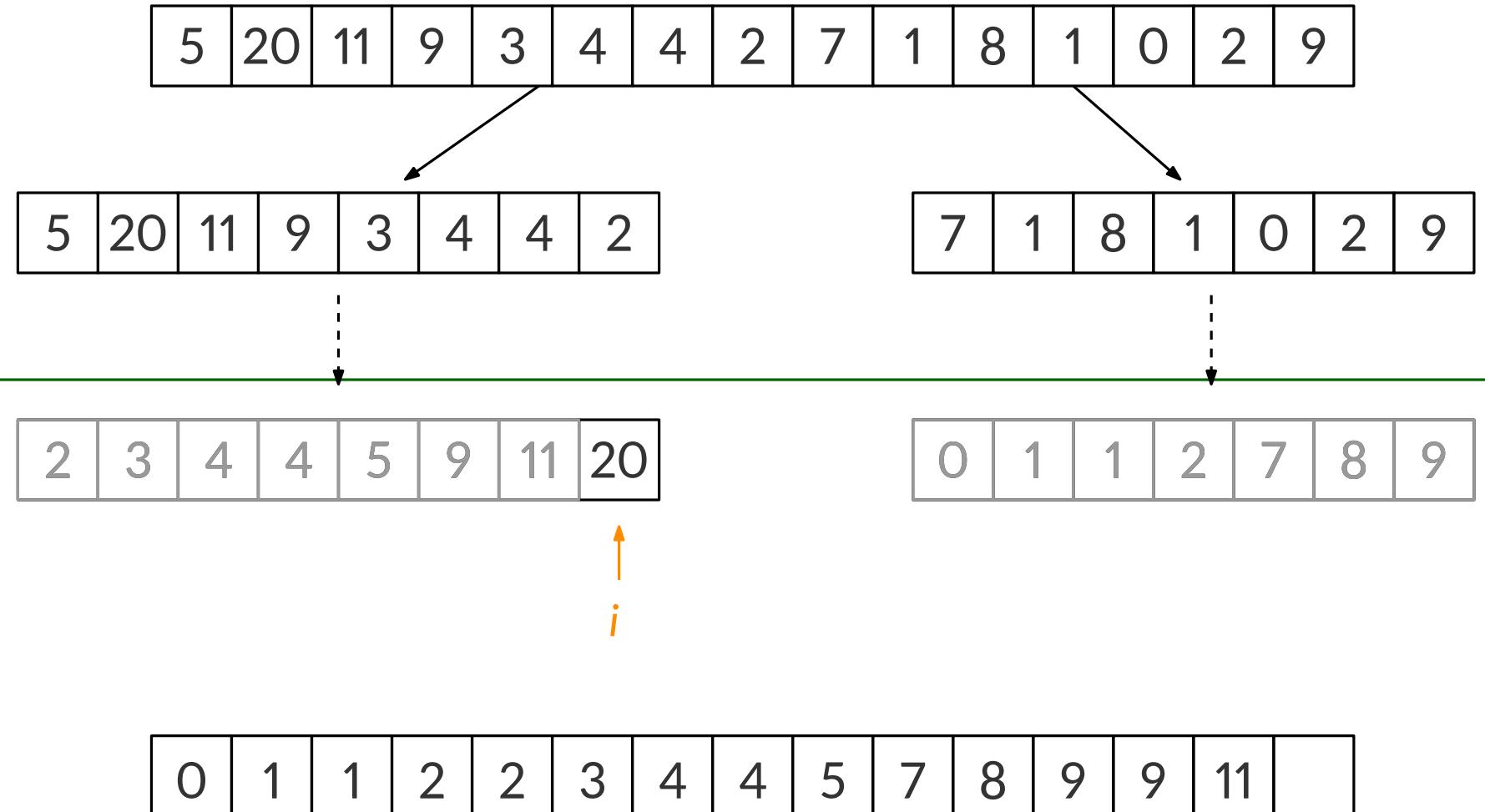
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



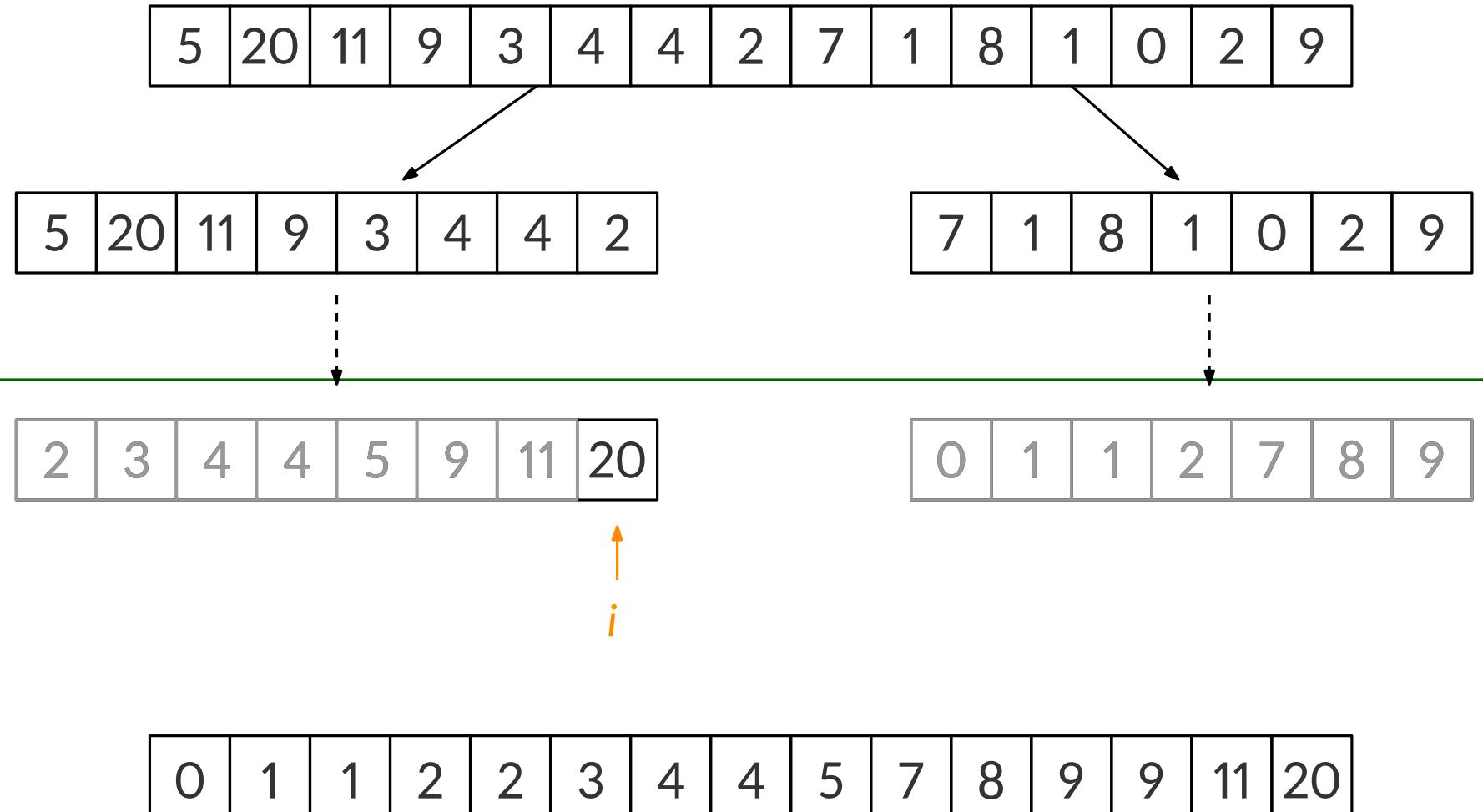
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



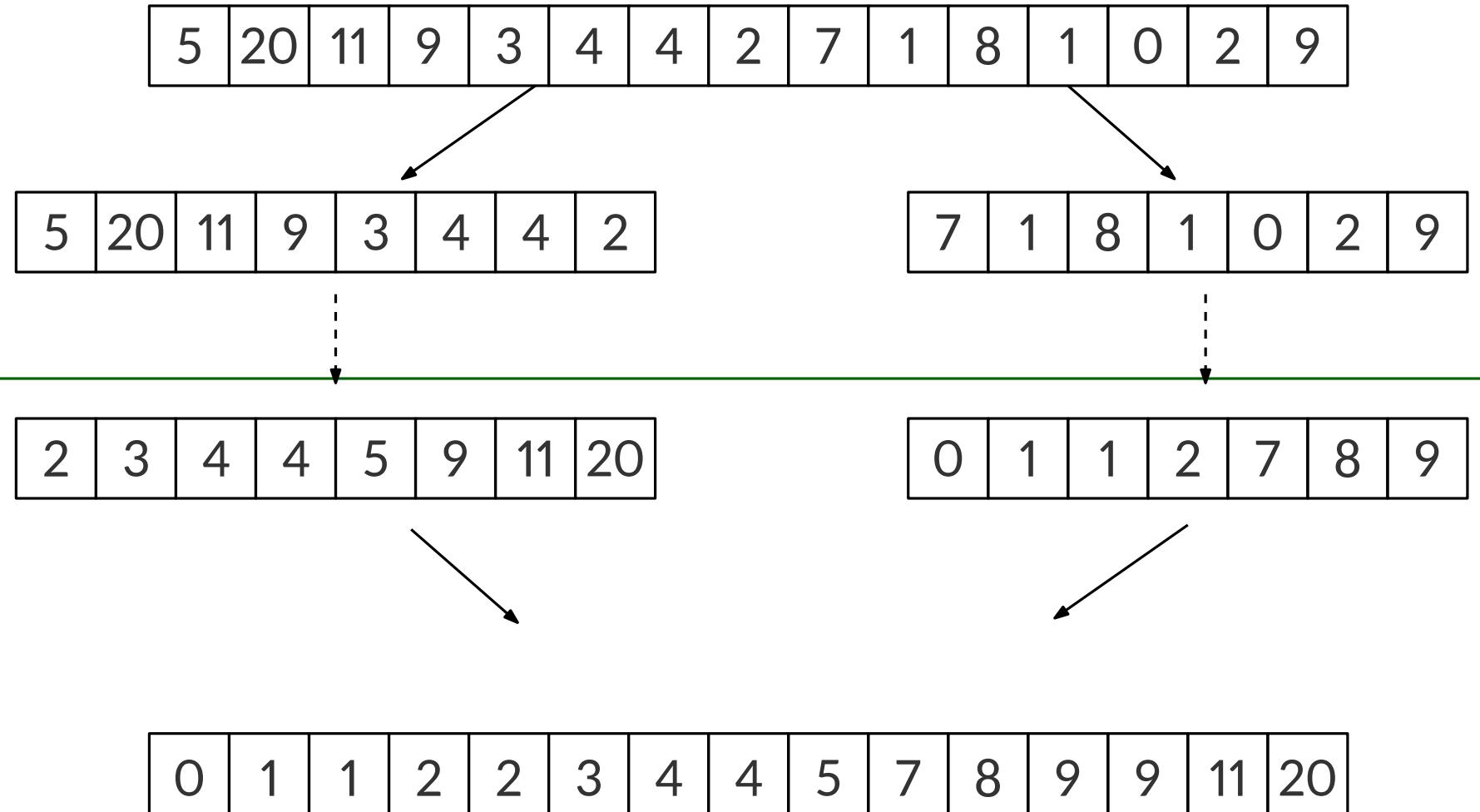
Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



Zentral ist der Merge-Schritt

# Merge-Schritt: Idee/Beispiel



Zentral ist der Merge-Schritt

# Merge Sort: Pseudocode

---

**Merge(A[1 ... n], B[1 ... m]):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array C[1 ... n + m] //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

**return**  $C[1], \dots, C[n + m]$

**MergeSort(A[1 ... n]):**

**if**  $n == 1$  **then**

        | **return**  $A[1]$

$m = \lfloor n/2 \rfloor$

$A'[1 \dots m] = \text{MergeSort}(A[1 \dots m])$

$A'[m + 1 \dots n] = \text{MergeSort}(A[m + 1 \dots n])$

**return**  $\text{Merge}(A'[1 \dots m], A'[m + 1 \dots n])$

# Merge Sort: Korrektheit

---

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

## Korrektheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
- kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

**return**  $C[1], \dots, C[n + m]$

# Merge Sort: Korrektheit

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

## Korrektheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
- kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

**return**  $C[1], \dots, C[n + m]$

## Initialisierung:

Nach dem ersten Schleifendurchlauf gilt die Schleifeninvariante:

- $C[1]$  is die sortierte Reihenfolge von  $A[1..i - 1], B[1..j - 1]$
- kein Element in  $A[i..n], B[j..n]$  ist kleiner als  $C[1]$

# Merge Sort: Korrektheit

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

## Korrektheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
- kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

**return**  $C[1], \dots, C[n + m]$

## Initialisierung:

Nach dem ersten Schleifendurchlauf gilt die Schleifeninvariante:

- $C[1]$  is die sortierte Reihenfolge von  $A[1..i - 1], B[1..j - 1]$
- kein Element in  $A[i..n], B[j..n]$  ist kleiner als  $C[1]$

## Aufrechterhaltung:

$C[1] \leq \dots \leq C[k]$

$\wedge \quad A[i] \leq A[i + 1] \leq \dots \leq A[n]$

$\wedge \quad B[j] \leq B[j + 1] \leq \dots \leq B[m]$

# Merge Sort: Korrektheit

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

## Korrektheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
- kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

**return**  $C[1], \dots, C[n + m]$

## Initialisierung:

Nach dem ersten Schleifendurchlauf gilt die Schleifeninvariante:

- $C[1]$  is die sortierte Reihenfolge von  $A[1..i - 1], B[1..j - 1]$
- kein Element in  $A[i..n], B[j..n]$  ist kleiner als  $C[1]$

## Aufrechterhaltung:

$C[1] \leq \dots \leq C[k]$

$\swarrow A[i] \leq A[i + 1] \leq \dots \leq A[n]$

|  $\wedge \sim$

$\nwarrow B[j] \leq B[j + 1] \leq \dots \leq B[m]$

# Merge Sort: Korrektheit

**Merge( $A[1 \dots n]$ ,  $B[1 \dots m]$ ):** //Eingabe: sortierte Arrays  $A, B$ , also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array C[1 ... n + m] //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$$i = 1, j = 1$$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$$C[k] = A[i]$$

$$i = i + 1$$

else

$$C[k] = B[i]$$

$$i \equiv i + 1$$

# Korrekttheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
  - kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

Nach dem ersten Schleifendurchlauf gilt die Schleifeninvariante:

- $C[1]$  ist die sortierte Reihenfolge von  $A[1..i - 1], B[1..j - 1]$
  - kein Element in  $A[i..n], B[j..n]$  ist kleiner als  $C[1]$

### Aufrechterhaltung:

$$C[1] \leq \dots \leq C[k-1] \quad \swarrow \quad C[k] \leq A[i] \leq \dots \leq A[n]$$

$$\quad\quad\quad \downarrow \wedge$$

$$B[j] \leq \dots \leq B[m]$$

# Merge Sort: Korrektheit

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

## Korrektheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
- kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

**return**  $C[1], \dots, C[n + m]$

## Initialisierung:

Nach dem ersten Schleifendurchlauf gilt die Schleifeninvariante:

- $C[1]$  is die sortierte Reihenfolge von  $A[1..i - 1], B[1..j - 1]$
- kein Element in  $A[i..n], B[j..n]$  ist kleiner als  $C[1]$

## Aufrechterhaltung:

$C[1] \leq \dots \leq C[k - 1]$

$A[i] \leq A[i + 1] \leq \dots \leq A[n]$

$\wedge \quad \forall j \quad C[k] \leq B[j] \leq \dots \leq B[m]$

# Merge Sort: Korrektheit

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

## Korrektheitsanalyse

**Schleifeninvariante:** Am Ende jeder for-Schleife gilt:

- $C[1] \leq \dots \leq C[k]$  ist sortierte Reihenfolge von  $A[1 \dots i - 1], B[1 \dots j - 1]$  und
- kein Element in  $A[i \dots n], B[j \dots m]$  ist kleiner als  $C[k]$

→ am Ende ist  $C[1], \dots, C[n + m]$  die gewünschte sortierte Reihenfolge von A und B

**return**  $C[1], \dots, C[n + m]$

**MergeSort( $A[1 \dots n]$ ):**

**if**  $n == 1$  **then**

**return**  $A[1]$

$m = \lfloor n/2 \rfloor$

$A'[1 \dots m] = \text{MergeSort}(A[1 \dots m])$

$A'[m + 1 \dots n] = \text{MergeSort}(A[m + 1 \dots n])$

**return**  $\text{Merge}(A'[1 \dots m], A'[m + 1 \dots n])$

Korrektheit von MergeSort folgt  
direkt aus der Korrektheit von Merge

# Merge Sort: Anzahl Vergleiche

---

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

**return**  $C[1], \dots, C[n + m]$

## Laufzeitanalyse

Wir analysieren zunächst die Anzahl an **Vergleichen**  $A[i] \leq B[j]$ .

## Lemma

$\text{Merge}(A[1 \dots n], B[1 \dots m])$  macht höchstens  $n + m$  Vergleiche

## Beweis:

- In jeder Iteration gibt es höchstens einen Vergleich  $A[i] \leq B[j]$
- Es gibt  $n + m$  Iterationen der for-Schleife

□

# Merge Sort: Anzahl Vergleiche II

---

## Bemerkung

Sei  $C_{MS}(n)$  die Anzahl an Vergleichen von MergeSort( $A[1 \dots n]$ ).

Es gilt:

$$C_{MS}(1) = 0$$

$$C_{MS}(n) \leq C_{MS}(\lfloor \frac{n}{2} \rfloor) + C_{MS}(\lceil \frac{n}{2} \rceil) + n \quad \text{für } n \geq 2$$

↗ nächstes Kapitel:  
grundlegendes Werkzeuge zur Analyse  
rekursiver Algorithmen

## Behauptung

$C_{MS}(n) \leq C \cdot n \log_2 n$  für eine Konstante  $C$

**Beweis** ↗ Tafelpräsentation

Hinweis:

Man kann auch zeigen, dass  $C_{MS} \in \Omega(n \log_2 n)$ .

⇒ MergeSort macht  $\Theta(n \log n)$  Vergleiche.

MergeSort( $A[1 \dots n]$ ):

**if**  $n == 1$  **then**  
**return**  $A[1]$

$$m = \lfloor n/2 \rfloor$$

$A'[1 \dots m] = \text{MergeSort}(A[1 \dots m])$

$A'[m + 1 \dots n] = \text{MergeSort}(A[m + 1 \dots n])$

**return**  $\text{Merge}(A'[1 \dots m], A'[m + 1 \dots n])$

# Merge Sort: Anzahl Vergleiche Laufzeit

**Merge( $A[1 \dots n], B[1 \dots m]$ ):** //Eingabe: sortierte Arrays A, B, also:  $A[1] \leq \dots \leq A[n]$  sowie  $B[1] \leq \dots \leq B[m]$

Initialisiere Array  $C[1 \dots n + m]$  //gewünschte Ausgabe: Array C, die sortierte Reihenfolge von A, B

$i = 1, j = 1$

**for**  $k = 1, \dots, n + m$  **do**

**if** ( $j == m + 1$  oder ( $i \leq n$  und  $A[i] \leq B[j]$ )) **then**

$C[k] = A[i]$

$i = i + 1$

**else**

$C[k] = B[j]$

$j = j + 1$

**return**  $C[1], \dots, C[n + m]$

## Laufzeitanalyse

Wir analysieren zunächst die Anzahl an ~~Vergleichen  $A[i] \leq B[j]$~~  die Laufzeit

## Lemma

$\text{Merge}(A[1 \dots n], B[1 \dots m])$  macht höchstens  ~~$n + m$  Vergleiche~~ läuft in Zeit  $O(n + m)$ .

## Beweis:

$O(1)$  Operationen.

- In jeder Iteration gibt es höchstens einen Vergleich  $A[i] \leq B[j]$
- Es gibt  $n + m$  Iterationen der for-Schleife

□

# Merge Sort: Anzahl Vergleiche II

---

## Laufzeit

**Bemerkung**  $T_{MS}(n)$  die Laufzeit

Sei  $C_{MS}(n)$  die Anzahl an Vergleichen von MergeSort( $A[1 \dots n]$ ).

~~Es gilt:~~ Es gibt Konstanten  $d, e$ , sodass

$$\underline{C_{MS}(1) = 0} \quad T_{MS}(1) = d$$

$$\underline{C_{MS}(n) \leq C_{MS}(\lfloor \frac{n}{2} \rfloor) + C_{MS}(\lceil \frac{n}{2} \rceil) + n} \quad \text{für } n \geq 2$$

$$T_{MS}(n) \leq T_{MS}(\lfloor \frac{n}{2} \rfloor) + T_{MS}(\lceil \frac{n}{2} \rceil) + e \cdot n$$

**Behauptung**

$$\underline{C_{MS}(n) \leq C \cdot n \log_2 n} \text{ für eine Konstante } C$$

$$T_{MS}(n) \leq C \cdot n \log_2 n \text{ für eine Konstante } C$$

**Beweis** ↗ Tafelpräsentation

⇒ Merge Sort läuft in Zeit  $\Theta(n \log n)$ .

Hinweis:

$$T_{MS}(n) \in \Omega(n \log_2 n)$$

Man kann auch zeigen, dass  $C_{MS} \in \Omega(n \log_2 n)$ .

⇒ MergeSort macht  $\Theta(n \log n)$  Vergleiche.

↗ nächstes Kapitel:

grundlegendes Werkzeuge zur Analyse  
rekursiver Algorithmen

MergeSort( $A[1 \dots n]$ ):

```
if  $n == 1$  then
    return  $A[1]$ 
```

$$m = \lfloor n/2 \rfloor$$

```
 $A'[1 \dots m] = \text{MergeSort}(A[1 \dots m])$ 
```

```
 $A'[m + 1 \dots n] = \text{MergeSort}(A[m + 1 \dots n])$ 
```

```
return  $\text{Merge}(A'[1 \dots m], A'[m + 1 \dots n])$ 
```

# Ausblick & Zusammenfassung

# Generalisiertes Sortieren

## Problem: Sortieren

gegeben: Sequenz  $E$  von Elementen  $e_1, \dots, e_n$  mit **Schlüsseln** (keys)  $k_i = \text{key}(e_i) \in K$

gesucht: Permutation  $e'_1, \dots, e'_n$  von  $E$ , sortiert nach den Schlüsseln

Das heißt:  $\text{key}(e'_1) \leq \text{key}(e'_2) \leq \dots \leq \text{key}(e'_n)$

**Annahme:** Auf der Schlüsselmenge  $K$  ist eine **totale Ordnung** definiert

Es gibt einen Vergleichsoperator  $\leq_K$  auf den Schlüsseln, sodass:

Für alle  $k, k', k'' \in K$  gilt

- $k \leq_K k$  (Reflexivität)
- wenn  $k \leq_K k'$  und  $k' \leq_K k$ , dann  $k = k'$  (Antisymmetrie)
- wenn  $k \leq_K k'$  und  $k' \leq_K k''$ , dann  $k \leq_K k''$  (Transitivität)
- $k \leq_K k'$  oder  $k' \leq_K k$  (Totalität)

→ es gibt es immer eine wohldefinierte sortierte Reihenfolge

Name	alternativer Name
Samweis	Perhael
Aragorn	Elessar
Saruman	Curunír
Arwen	Undómiel
Olórin	Mithrandir

$\xrightarrow{\leq_{\text{Name}}^{\text{lex}}}$

Name	alternativer Name
Aragorn	Elessar
Arwen	Undómiel
Olórin	Mithrandir
Samweis	Perhael
Saruman	Curunír

# Zusammenfassung

---

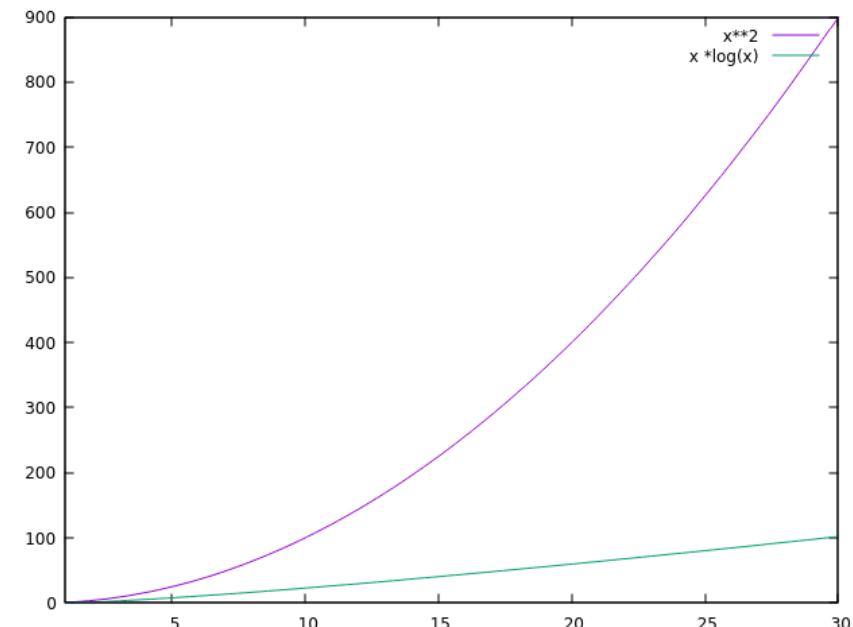
	Bubble Sort	Selection Sort	Merge Sort
worst-case Laufzeit	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$
Idee	Nachbarn vertauschen, wenn nötig	$n$ -maliges Bestimmen des Minimums	Divide-and-Conquer-Methode

Hinweise

schneller für nahezu sortierte Eingaben

**Interessante Frage für ein späteres Kapitel:**

Geht es noch schneller als  $\Theta(n \log n)$ ?



Algorithmen und Datenstrukturen SS'23

# Kapitel 4: Analyse rekursiver Algorithmen

Marvin Künemann

AG Algorithmen & Komplexität

# Quiz: geometrische Reihen

---

1.  $\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = \dots?$

a) 2

b) 100 000

c)  $\infty$

2.  $\sum_{i=0}^{\infty} 2^i = \dots?$

a) 2

b) 100 000

c)  $\infty$

3.  $\sum_{i=0}^{\infty} 0.99999^i = \dots?$

a) 2

b) 100 000

c)  $\infty$

4.  $\sum_{i=0}^{\infty} 1^i = \dots?$

a) 2

b) 100 000

c)  $\infty$

**Fakt:** Wenn  $q \neq 1$ , gilt:  $\sum_{i=0}^k q^i = \frac{1-q^{k+1}}{1-q}$

↗ TCS Cheat Sheet

Wenn  $|q| < 1$ , gilt:  $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$

# Kapitelüberblick

---

Letztes Kapitel: erste Sortierverfahren, insbesondere Merge Sort

Jetzt: Exkurs zur Analyse rekursiver Algorithmen

Insbesondere klären wir in diesem Kapitel:

- Analyse typischer Divide-and-Conquer-Algorithmen in zwei Schritten:

**Schritt 1: Rekursions(un)gleichung aufstellen**

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

**Schritt 2: Rekursions(un)gleichung lösen**

$$\Rightarrow T(n) \in \Theta(\dots)$$

# Rekursive Algorithmen: Übersicht

Wir haben bereits Beispiele für rekursive Algorithmen gesehen

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$

$$a^{(1)} = a_{n-1} \dots a_{\lfloor n/2 \rfloor}, a^{(0)} = a_{\lfloor n/2 \rfloor - 1} \dots a_0 \\ b^{(1)} = b_{n-1} \dots b_{\lfloor n/2 \rfloor}, b^{(0)} = b_{\lfloor n/2 \rfloor - 1} \dots b_0$$

$\alpha = \text{Karatsuba}(a^{(0)}, b^{(0)})$

$\beta = \text{Karatsuba}(a^{(1)}, b^{(1)})$

$\gamma = \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$

$$\delta = \gamma - \alpha - \beta$$

**return**  $\beta \cdot B^n + \delta B^{\lfloor n/2 \rfloor} + \alpha$

$$\Theta(n^{\log_2 3})$$

$$\log_2 3 \approx 1.585$$

Base Case

Aufteilen

Rekursive Aufrufe

Zusammenfügen

MergeSort( $A[1 \dots n]$ ):

**if**  $n = 1$  **then return**  $A[1]$

$$m = \lfloor n/2 \rfloor$$

$A'[1 \dots m] = \text{MergeSort}(A[1 \dots m])$

$A'[m+1 \dots n] = \text{MergeSort}(A[m+1 \dots n])$

**return** Merge( $A'[1 \dots m], A'[m+1 \dots n]$ )

## Schritt 1: **Rekursions(un)gleichung aufstellen**

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

# Rekurrenzgleichung erstellen

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

**if**  $n = 1$ , **then return**  $a_0 \cdot b_0$

$$a^{(1)} = a_{n-1} \dots a_{\lfloor n/2 \rfloor}, a^{(0)} = a_{\lfloor n/2 \rfloor - 1} \dots a_0$$

$$b^{(1)} = b_{n-1} \dots b_{\lfloor n/2 \rfloor}, b^{(0)} = b_{\lfloor n/2 \rfloor - 1} \dots b_0$$

$$\alpha = \text{Karatsuba}(a^{(0)}, b^{(0)})$$

$$\beta = \text{Karatsuba}(a^{(1)}, b^{(1)})$$

$$\gamma = \text{Karatsuba}(a^{(1)} + a^{(0)}, b^{(1)} + b^{(0)})$$

$$\delta = \gamma - \alpha - \beta$$

$$\text{return } \beta \cdot B^n + \delta B^{n/2} + \alpha$$

Es gibt Konstanten  $C, D$ , sodass

$$T(1) = C$$

$$\begin{aligned} T(n) &\leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + T(\frac{n}{2} + 1) + Dn \quad n \geq 2 \\ &\leq 3 \cdot T(\frac{n}{2} + 1) + Dn \end{aligned}$$

MergeSort( $A[1 \dots n]$ ):

**if**  $n = 1$  **then return**  $A[1]$

$$m = \lfloor n/2 \rfloor$$

$$A'[1 \dots m] = \text{MergeSort}(A[1 \dots m])$$

$$A'[m+1 \dots n] = \text{MergeSort}(A[m+1 \dots n])$$

**return** Merge( $A'[1 \dots m], A'[m+1 \dots n]$ )

Es gibt Konstanten  $C, D$ , sodass

$$T(1) = C$$

$$\begin{aligned} T(n) &\leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + Dn \\ &\leq 2 \cdot T(\frac{n}{2} + 1) + Dn \end{aligned}$$

# Rekurrenzgleichung erstellen

---

Karatsuba( $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ )

Es gibt Konstanten  $c, d$ , sodass

$$\begin{aligned} T(1) &= c \\ T(n) &\leq 3 \cdot T\left(\frac{n}{2} + 1\right) + dn \quad \text{für } n \geq 2 \end{aligned}$$

MergeSort( $A[1 \dots n]$ ):

Es gibt Konstanten  $c, d$ , sodass

$$\begin{aligned} T(1) &= c \\ T(n) &\leq 2 \cdot T\left(\frac{n}{2} + 1\right) + dn \quad \text{für } n \geq 2 \end{aligned}$$

## Rekurrenz

Typische Formen einer Rekursions(un)gleichung:

**Einfache Form:** Es gibt Konstanten  $C, D, n_0, a, b, s$  sodass

$$T(n) \leq \begin{cases} C & \text{für alle } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

**Allgemeinere Form:** Es gibt Konstanten  $C, D, n_0, a, b, e, s$  sodass

$$T(n) \leq \begin{cases} C & \text{für alle } n \leq n_0 \\ a \cdot T\left(\lceil \frac{n}{b} \rceil + e\right) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

## Schritt 2: Rekurrenz(un)gleichung lösen

$$T(n) \in \Theta(\dots)$$

# Methode 1: Raten/Wissen und induktiv Beweisen

---

Bereits ein Beispiel gesehen:

$$C(1) = 0$$

$$C(n) = 2 \cdot C\left(\frac{n}{2}\right) + n \quad \text{für } n \geq 2$$

Wenn man weiß oder rät, dass  $C(n) = n \log_2 n$ , ist die Aussage leicht per Induktion zu beweisen!  
(für Zweierpotenzen)

↗ Kapitel 3

Leider kommt es bei dieser Methode auch auf kleine Details an:

Für  $T(n) = \begin{cases} 0 & \text{wenn } n = 1, \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1 & \text{wenn } n \geq 2 \end{cases}$

lässt sich  $T(n) \in O(n)$  leicht induktiv zeigen, wenn man rät, dass  $T(n) \leq n - 1$ .

Der Induktionsschritt für die Aussage  $T(n) \leq n$  erzeugt leider Probleme.

↗ Tafelpräsentation

# Methode 2: Rekursionsbaummethode

---

Angenommen, wir wissen für eine Laufzeitfunktion  $T$ , dass es Konstanten  $c, d$  gibt sodass

$$T(n) \leq \begin{cases} c & \text{wenn } n < 4, \\ 3T\left(\frac{n}{4}\right) + d \cdot n^2 & \text{wenn } n \geq 4 \end{cases}$$

Wir nehmen hier der Einfachheit halber an, dass  $n = 4^k$  für ein  $k \in \mathbb{N}$ , um Rundungseffekte zu ignorieren.

Wiederholtes Einsetzen bzw. Rekursionsbaum ( $\nearrow$  Tafelpräsentation)

$$\begin{aligned} T(n) &\leq 3T\left(\frac{n}{4}\right) + dn^2 \\ &\leq 3 \left( 3T\left(\frac{n}{16}\right) + d \left(\frac{n}{4}\right)^2 \right) + dn^2 \\ &\leq 3 \left( 3 \left( 3T\left(\frac{n}{64}\right) + d \left(\frac{n}{16}\right)^2 \right) + d \left(\frac{n}{4}\right)^2 \right) + dn^2 \\ &\dots \\ &\leq 3^{\log_4(n)} T(1) + 3^{\log_4(n)-1} \cdot \frac{dn^2}{16^{\log_4(n)-1}} + \dots + 3^2 \cdot \frac{dn^2}{16^2} + 3 \cdot \frac{dn^2}{16} + dn^2 \\ &= n^{\log_4(3)} T(1) + \sum_{i=0}^{\log_4(n)-1} \left(\frac{3}{16}\right)^i dn^2 \leq n^{\log_4(3)} c + \left(\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i\right) dn^2 \\ &\in O(n^{\log_4(3)} + n^2) = O(n^2). \end{aligned}$$

# Achtung:

Es folgt **sehr nützlicher** Inhalt!

# Master-Theorem

# Methode 3: Master-Theorem

---

## Master-Theorem (einfache Form)

Sei  $T$  eine Laufzeitfunktion, die die folgende Rekursionsungleichung erfüllt:

Einfache Form: Es gibt Konstanten  $C, D, n_0, a, b, s$  mit  $a \geq 1, b > 1$  sodass

$$T(n) \leq \begin{cases} C & \text{für alle } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

Dann gilt:

1. Wenn  $a > b^s$  (d.h.  $\log_b a > s$ ), dann ist  $T(n) \in O(n^{\log_b a})$

2. Wenn  $a = b^s$  (d.h.  $\log_b a = s$ ), dann ist  $T(n) \in O(n^s \log n)$

3. Wenn  $a < b^s$  (d.h.  $\log_b a < s$ ), dann ist  $T(n) \in O(n^s)$

**Hinweis:** Diese Schranke ist scharf.

# Methode 3: Master-Theorem II

---

## Master-Theorem (allgemeinere Form)

Sei  $T$  eine Laufzeitfunktion, die die folgende Rekursionsungleichung erfüllt:

Allgemeinere Form: Es gibt Konstanten  $C, D, n_0, a, b, e, s$  mit  $a \geq 1, b > 1$  sodass

$$T(n) \leq \begin{cases} C & \text{für alle } n \leq n_0 \\ a \cdot T(\lceil \frac{n}{b} \rceil + e) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

Dann gilt:

1. Wenn  $a > b^s$  (d.h.  $\log_b a > s$ ), dann ist  $T(n) \in O(n^{\log_b a})$

2. Wenn  $a = b^s$  (d.h.  $\log_b a = s$ ), dann ist  $T(n) \in O(n^s \log n)$

3. Wenn  $a < b^s$  (d.h.  $\log_b a < s$ ), dann ist  $T(n) \in O(n^s)$

**Hinweis:** Diese Schranke ist scharf.

# Methode 3: Master-Theorem III

## Beispiele

Laufzeit von Merge Sort:

$$T(1) = c$$

$$T(n) \leq 2 \cdot T\left(\frac{n}{2} + 1\right) + dn \quad \text{für } n \geq 2$$

Master-Thm. (Erinnerung)

$$T(n) \leq \begin{cases} C & \text{für alle } n \leq n_0 \\ a \cdot T\left(\lceil \frac{n}{b} \rceil + e\right) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

$a \geq 1, b > 1$

für alle  $n \leq n_0$

1. Wenn  $a > b^s$ , dann  $T \in O(n^{\log_b a})$

2. Wenn  $a = b^s$ , dann  $T \in O(n^s \log n)$

3. Wenn  $a < b^s$ , dann  $T \in O(n^s)$

nach Mastertheorem, Fall 2:

$$T(n) \in O(n \log n)$$

Laufzeit von Karatsubas Algorithmus:

$$T(1) = c$$

$$T(n) \leq 3 \cdot T\left(\frac{n}{2} + 1\right) + dn \quad \text{für } n \geq 2$$

nach Mastertheorem, Fall 1:

$$T(n) \in O(n^{\log_2 3})$$

Was wäre, wenn Karatsubas Algorithmus Zeit  $O(n^2)$  für den nicht-rekursiven Teil benötigte?

$$T(1) = c$$

$$T(n) \leq 3 \cdot T\left(\frac{n}{2} + 1\right) + dn^2 \quad \text{für } n \geq 2$$

nach Mastertheorem, Fall 3:

$$T(n) \in O(n^2)$$

Bemerkungen:

· es gibt viele verschiedene Varianten des Mastertheorems:

· z.B. Aufteilen des Problems in 2 Subprobleme der Größe  $\frac{n}{3}$  und ein Subproblem der Größe  $\frac{n}{3}$ , etc.

→ diese lassen sich für gewöhnlich durch eine detaillierte Analyse durch die Rekursionsbaummethode beweisen

# Methode 3: Master-Theorem III, untere Schranke

---

## Master-Theorem (allgemeinere Form, untere Schranke)

Sei  $T$  eine Laufzeitfunktion, die die folgende Rekursionsungleichung erfüllt:

Allgemeinere Form: Es gibt Konstanten  $C, D, n_0, a, b, e, s$  mit  $a \geq 1, b > 1$  sodass

$$T(n) \geq \begin{cases} C & \text{für alle } n \leq n_0 \\ a \cdot T(\lfloor \frac{n}{b} \rfloor - e) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

Dann gilt:

1. Wenn  $a > b^s$  (d.h.  $\log_b a > s$ ), dann ist  $T(n) \in \Omega(n^{\log_b a})$

2. Wenn  $a = b^s$  (d.h.  $\log_b a = s$ ), dann ist  $T(n) \in \Omega(n^s \log n)$

3. Wenn  $a < b^s$  (d.h.  $\log_b a < s$ ), dann ist  $T(n) \in \Omega(n^s)$

# Zusammenfassung

---

Grundlegende Methodik zur Analyse rekursiver Algorithmen

**Schritt 1: Rekursions(un)gleichung aufstellen**

$$T(n) = \begin{cases} C & \text{wenn } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & \text{wenn } n > n_0 \end{cases}$$

**Schritt 2: Rekursions(un)gleichung lösen**

$$\Rightarrow T(n) \in \Theta(\dots)$$

Methode 1: richtig raten und beweisen → Details sind wichtig

Methode 2: Rekursionsbaum aufschreiben und zusammenrechnen

Methode 3: Kopf ausschalten und Master-Theorem anwenden

## Algorithmen und Datenstrukturen SS'23

# Kapitel 5: Sortieren in $o(n \log n)$ ?

Marvin Künemann

AG Algorithmen & Komplexität

# Kapitelüberblick

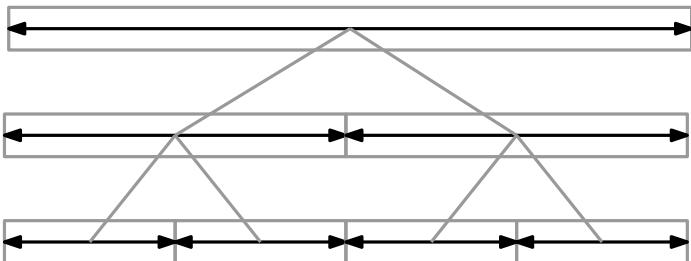
---

Letztes Kapitel: Exkurs zur Analyse rekursiver Algorithmen

# Die Fälle des Mastertheorems: Intuition (Fall 2)

MergeSort:

$$T(n) \leq \begin{cases} c & \text{für alle } n \leq 1 \\ 2 \cdot T(\lceil \frac{n}{2} \rceil) + Dn & \text{für alle } n > 1 \end{cases}$$



$$a \geq 1, b > 1$$

$$T(n) \leq \begin{cases} c & \text{für alle } n \leq n_0 \\ a \cdot T(\lceil \frac{n}{b} \rceil + e) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

1. Wenn  $a > b^s$ , dann  $T \in O(n^{\log_b a})$
2. Wenn  $a = b^s$ , dann  $T \in O(n^s \log n)$
3. Wenn  $a < b^s$ , dann  $T \in O(n^s)$

$$\begin{aligned} \frac{Dn}{2} + \frac{Dn}{2} &= Dn \\ \frac{Dn}{4} + \dots + \frac{Dn}{4} &= Dn \\ &\vdots \\ \underbrace{C + \dots + C}_{n \text{ Mal}} &= Cn \end{aligned}$$

$Dn$   
 $\log_2(n) \text{ mal}$

jedes Level der Rekursion hat (etwa) den gleichen Beitrag  $O(n)$  zur Laufzeit  $\Rightarrow$  Laufzeit  $O(n \log n)$

# Die Fälle des Mastertheorems: Intuition (Fall 1)

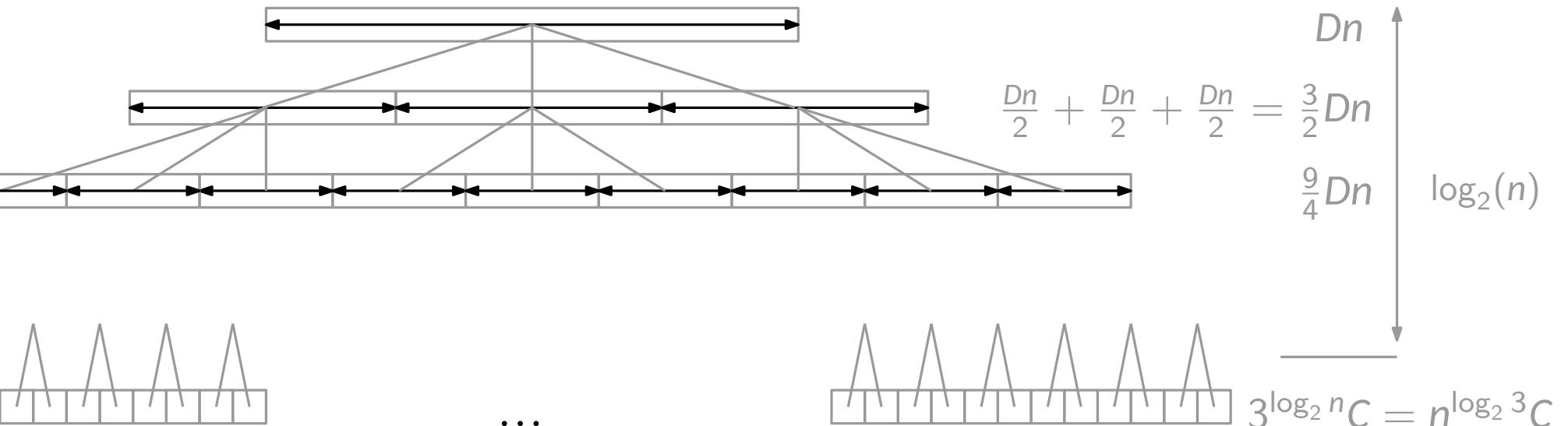
$$a \geq 1, b > 1$$

$$T(n) \leq \begin{cases} c & \text{für alle } n \leq n_0 \\ a \cdot T(\lceil \frac{n}{b} \rceil + e) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

Karatsuba:

$$T(n) \leq \begin{cases} c & \text{für alle } n \leq 1 \\ 3 \cdot T(\lceil \frac{n}{2} \rceil + 1) + Dn & \text{für alle } n > 1 \end{cases}$$

1. Wenn  $a > b^s$ , dann  $T \in O(n^{\log_b a})$
2. Wenn  $a = b^s$ , dann  $T \in O(n^s \log n)$
3. Wenn  $a < b^s$ , dann  $T \in O(n^s)$

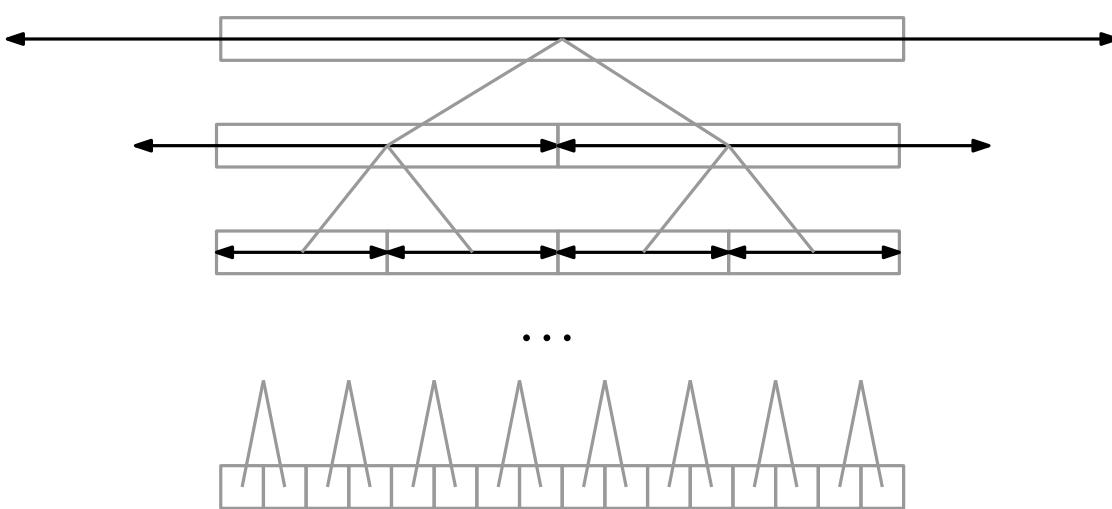


je tiefer das Level, desto höher der Beitrag zur Laufzeit

$\Rightarrow$  Laufzeit dominiert von der Arbeit für die Blätter

# Die Fälle des Mastertheorems: Intuition (Fall 3)

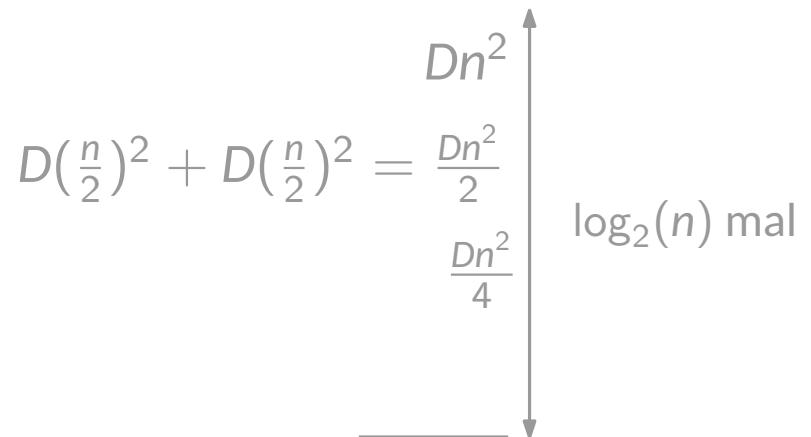
$$T(n) \leq \begin{cases} c & \text{für alle } n \leq 1 \\ 2 \cdot T(\lceil \frac{n}{2} \rceil + 1) + Dn^2 & \text{für alle } n > 1 \end{cases}$$



$$a \geq 1, b > 1$$

$$T(n) \leq \begin{cases} c & \text{für alle } n \leq n_0 \\ a \cdot T(\lceil \frac{n}{b} \rceil + e) + Dn^s & \text{für alle } n > n_0 \end{cases}$$

1. Wenn  $a > b^s$ , dann  $T \in O(n^{\log_b a})$
2. Wenn  $a = b^s$ , dann  $T \in O(n^s \log n)$
3. Wenn  $a < b^s$ , dann  $T \in O(n^s)$



den größten Beitrag hat das erste Level, danach geometrisch abfallend

$\Rightarrow$  Laufzeit dominiert von der "initialen" Arbeit  $O(n^s)$

# Kapitelüberblick

---

Letztes Kapitel: Exkurs zur Analyse rekursiver Algorithmen

Jetzt: Können wir in Zeit  $o(n \log n)$  sortieren?

Insbesondere besprechen wir in diesem Kapitel:

- eine untere Schranke für vergleichsbasierte Sortieralgorithmen
- nicht vergleichsbasierte Algorithmen Counting Sort und Radix Sort

# Motivierende Frage

---

Ist  $\Theta(n \log n)$  Zeit optimal für das Sortieren?

Wir beweisen: Ja, für **vergleichsbasierte Algorithmen!**

# Vergleichsbasierter Algorithmus

---

Wir definieren ein **eingeschränktes** Modell für Sortieralgorithmen:

Dazu definieren wir eine **Vergleichsoperation**  $\stackrel{?}{\leq}$ , die den Wahrheitswert von  $A[i] \leq A[j]$  berechnet:

$$A[i] \stackrel{?}{\leq} A[j] = \begin{cases} \text{true} & \text{wenn } A[i] \leq A[j] \\ \text{false} & \text{ansonsten.} \end{cases} \quad i, j \in \{1, \dots, n\}$$

## Definition

Ein Sortieralgorithmus ist **vergleichsbasiert**, wenn er auf das Eingabearray nur mittels der Vergleichsoperation  $\stackrel{?}{\leq}$  zugreift.

Also: kein "Rechnen" mit Eingabewerten, kein Zugriff auf konkreten Wert!

Das heißt unser Problem ist wie folgt:

Gegeben  $A[1], \dots, A[n]$ , die wir nur miteinander vergleichen können, bestimme eine Permutation  $(i_1, \dots, i_n)$  von  $(1, \dots, n)$  sodass

$$A[i_1] \leq A[i_2] \leq \dots \leq A[i_n]$$

**Wichtiger Hinweis:** Bubble Sort, Selection Sort und Merge Sort sind vergleichsbasierte Algorithmen!

# Untere Schranke für vergleichsbasierte Algorithmen

---

Wir werden zeigen:

Jeder vergleichsbasierte Algorithmus benötigt  $\Omega(n \log n)$  Vergleiche im worst-case.

# Bäume

(Gerichteter) Baum  $T$

Level 0

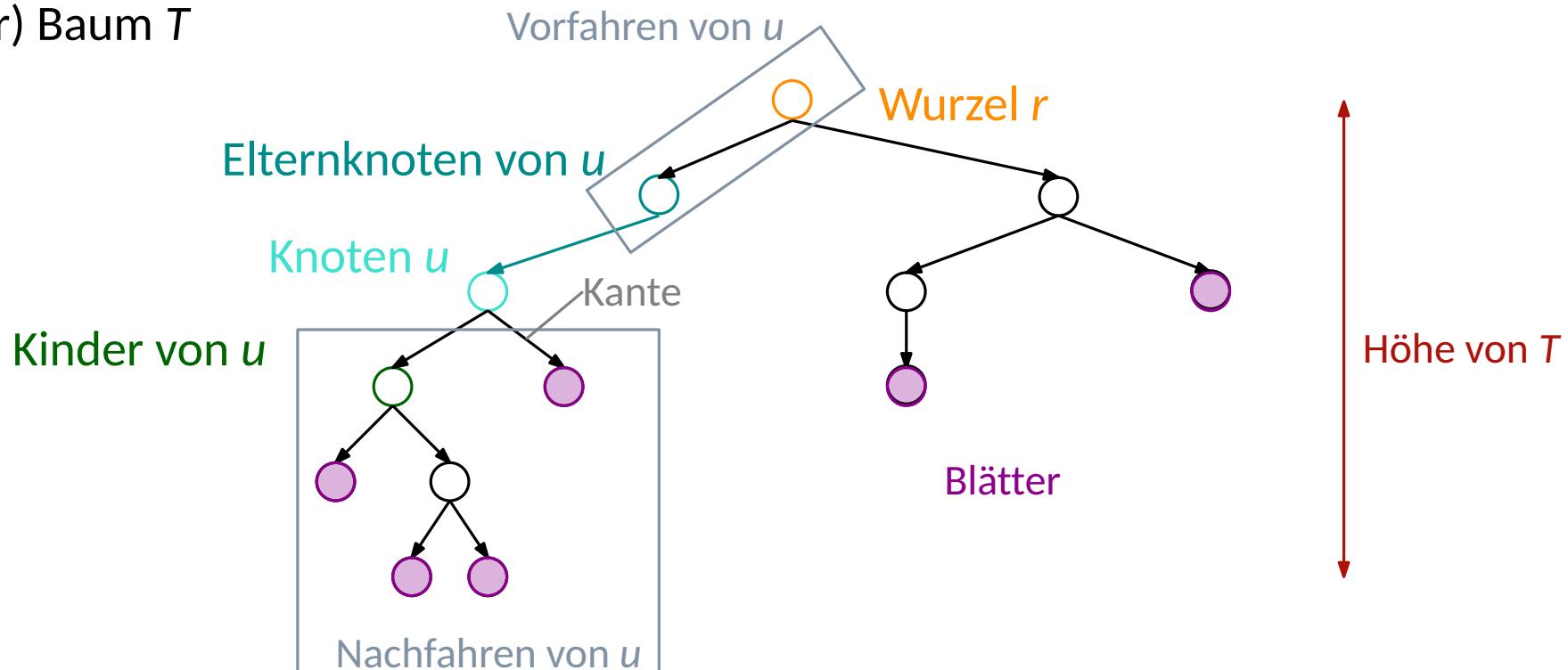
Level 1

Level 2

Level 3

Level 4

Level 5



Wurzel: Knoten ohne Vorfahren

Blätter: Knoten ohne Nachfahren

innere Knoten: Knoten mit Nachfahren

Höhe von  $T$ : das maximale Level eines Knotens in  $T$

$T$  ist Binärbaum, wenn jeder Knoten höchstens zwei Kinder hat

$T$  ist voller Binärbaum, wenn jeder Knoten entweder kein Kind oder zwei Kinder hat

**Pfad:** Eine Knotensequenz  $v_0, \dots, v_t$ , sodass für alle  $i$  die Kante  $(v_i, v_{i+1})$  im Baum existiert

Bemerke: Für jeden Knoten in  $T$  gibt es einen eindeutigen Pfad von der Wurzel

**Fakt** Jeder Binärbaum der Höhe  $d$  hat höchstens  $2^d$  Blätter

$\Rightarrow$  Ein Binärbaum mit  $n$  Blättern hat Höhe mindestens  $\log_2 n$

# Entscheidungsbaum für das Sortieren

Ein Entscheidungsbaum beschreibt den Entscheidungsprozess, um die sortierte Reihenfolge aus Vergleichen zu bestimmen

## Definition

Ein Entscheidungsbaum für das Sortieren ist ein **voller Binärbaum**  $T$ , sodass

- Jedem inneren Knoten ist ein Vergleich der Form  $A[i] \stackrel{?}{\leq} A[j]$  zugewiesen
- Für jeden inneren Knoten ist eine ausgehende Kante mit **true**, die andere mit **false** beschriftet
- Jedem Blatt  $b$  von  $T$  ist eine Permutation  $\pi_b = (i_1, \dots, i_n)$  von  $(1, \dots, n)$  zugewiesen
- Für jedes Blatt  $b$  gilt:

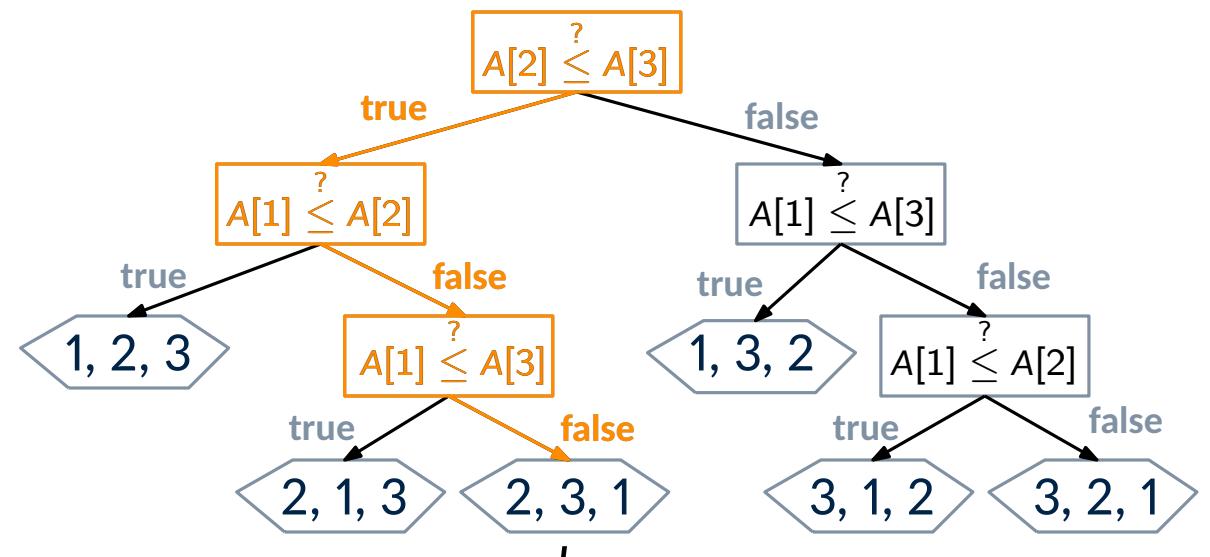
Der Pfad von der Wurzel zu  $b$  korrespondiert zu Vergleichen  $v_1, \dots, v_t$  mit Wahrheitswerten  $\alpha_1, \dots, \alpha_t$   
Die Permutation  $\pi_b = (i_1, \dots, i_n)$  erfüllt  $A[i_1] \leq A[i_2] \leq \dots \leq A[i_n]$ , wenn  $v_i = \alpha_i$  für alle  $i \in \{1, \dots, t\}$

$$v_1 = \alpha_1: A[2] \leq A[3]$$

$$v_2 = \alpha_2: A[1] > A[2]$$

$$v_3 = \alpha_3: A[1] > A[3]$$

$$\Rightarrow A[2] \leq A[3] < A[1]$$



# Untere Schranke für vergleichsbasierte Algorithmen

---

Wir werden zeigen:

Jeder vergleichsbasierte Algorithmus benötigt  $\Omega(n \log n)$  Vergleiche im worst-case.

**Beweis** Sei  $A$  ein vergleichsbasierter Sortieralgorithmus, der höchstens  $t(n)$  Vergleiche macht.

1. Es gibt einen Entscheidungsbaum  $T_A$  für das Sortieren mit Höhe höchstens  $t(n)$ .

↗ Tafelpräsentation

2. Ein Binärbaum mit mindestens  $b$  Blättern hat Höhe mindestens  $\log_2 b$

↗ Folie 10

3. Ein Entscheidungsbaum für das Sortieren hat mindestens  $n!$  Blätter.

↗ Tafelpräsentation

**Zusammensetzen:** Für jeden vergleichsbasierten Algorithmus gilt:

$$t(n) \geq \text{Höhe von } T_A \geq \log_2(\text{Anzahl Blätter von } T_A) \geq \log_2(n!) \in \Omega(n \log n).$$

1.                    2.                    3.                    ↗ Tafelpräsentation      □

Tatsächlich haben wir sogar bewiesen, dass  $t(n) \in n \log_2(n) - O(n)$

Wir haben auch bewiesen, dass Merge Sort höchstens  $n \log_2(n)$  Vergleiche macht (wenn  $n = 2^k, k \in \mathbb{N}$ )

12 - 15 ⇒ Optimal bis auf linearen Term (ein Term niedrigerer Ordnung)

# Motivierende Frage

---

Ist  $\Theta(n \log n)$  Zeit optimal für das Sortieren?

Ja, für **vergleichsbasierte Algorithmen!**

$\Omega(n \log n)$  untere Schranke im Entscheidungsbaummodell

Gibt es schnelle Sortieralgorithmen, die nicht vergleichsbasiert sind?

# Sortieren kleiner Zahlen

## Problem: Sortieren

gegeben: Liste von nat. Zahlen  $A[1], \dots, A[n] \in \{1, \dots, U\}$

gesucht: sortierte Reihenfolge von  $A[1], \dots, A[n]$

Das heißt: finde eine Permutation  $A'[1], \dots, A'[n]$ , sodass

$$A'[1] \leq A'[2] \leq \dots \leq A'[n]$$

Frage:

Gilt die  $\Omega(n \log n)$  Schranke für vergleichsbasierte Algorithmen auch für dieses Problem?

Genauer: für welche Werte von  $U$  gilt sie?

$c(6)$	
$c(5)$	$A[5] = 10$
$c(4)$	$A[4] = 100$
$c(3)$	$A[3] = 3$
$c(2)$	$A[2] = 2$
$c(1)$	$A[1] = 10$

$c(6)$	
$c(5)$	$A'[5] = 100$
$c(4)$	$A'[4] = 10$
$c(3)$	$A'[3] = 10$
$c(2)$	$A'[2] = 3$
$c(1)$	$A'[1] = 2$

Wir beschreiben einen Algorithmus Counting Sort

## Theorem

Counting Sort sortiert  $A[1 \dots n]$  in Zeit  $O(n + U)$ .

Das heißt, wir können z.B.  $n$  Zahlen zwischen 1 und  $100n$  in Zeit  $\Theta(n)$  sortieren.

# Counting Sort

---

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt  
→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$



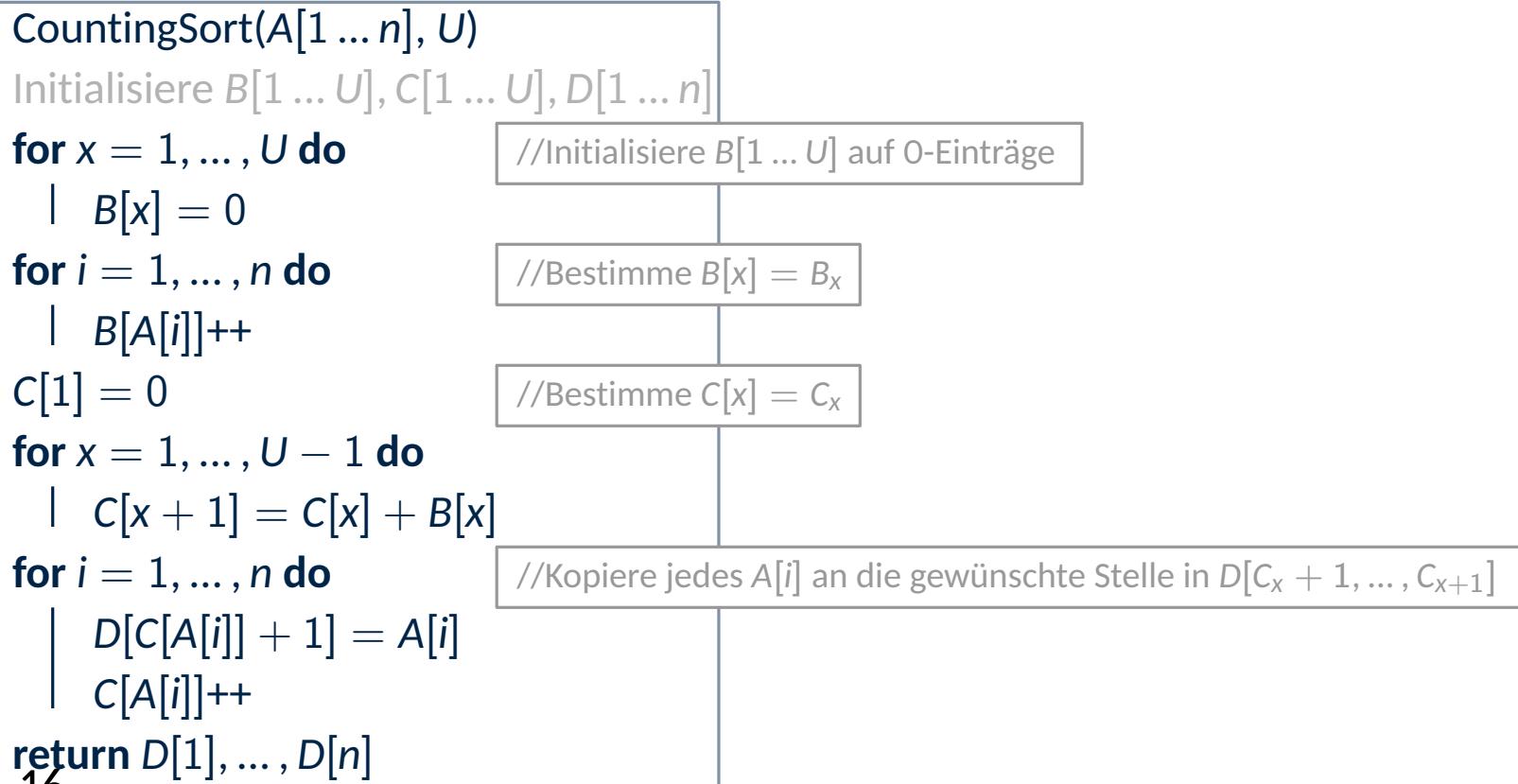
# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$



# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
for i = 1, ..., n do
    | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort(A[1..9], 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

1 2 3 4 5

$C$ 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

$D$ 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

0	0	0	0	0
---	---	---	---	---

1 2 3 4 5

$C$ 

1	1	1	1	1
---	---	---	---	---

$D$ 

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

B 

0	0	0	0	0
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
	↑								
B	0	0	0	0	0				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
	↑								
B	0	0	0	1	0				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
							↑		
B	0	0	0	1	0				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	2	3	4	5	6	7	8	9
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
							↑		
B	1	0	0	1	0				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	2	3	4	5	6	7	8	9
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

```
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
```

```
for x = 1, ..., U do
```

```
    | B[x] = 0
```

```
    for i = 1, ..., n do
```

```
        | B[A[i]]++
```

```
C[1] = 0
```

```
for x = 1, ..., U - 1 do
```

```
    | C[x + 1] = C[x] + B[x]
```

```
for i = 1, ..., n do
```

```
    | D[C[A[i]] + 1] = A[i]  
    | C[A[i]]++
```

```
return D[1], ..., D[n]
```

**Beispiel:** CountingSort(A[1..9], 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

1	0	0	1	0
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9], 5$ )

A	4	1	1	3	5	5	1	4	4
					↑				
B	2	0	0	1	0				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9], 5$ )

A	4	1	1	3	5	5	1	4	4
B	2	0	0	1	0				
	1	2	3	4	5				
C	1	1	1	1	1				
D	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
B	2	0	1	1	0				
	1	2	3	4	5				
C	1	1	1	1	1				
D	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

```
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
```

```
for x = 1, ..., U do
```

```
    | B[x] = 0
```

```
    for i = 1, ..., n do
```

```
        | B[A[i]]++
```

```
C[1] = 0
```

```
for x = 1, ..., U - 1 do
```

```
    | C[x + 1] = C[x] + B[x]
```

```
for i = 1, ..., n do
```

```
    | D[C[A[i]] + 1] = A[i]  
    | C[A[i]]++
```

```
return D[1], ..., D[n]
```

**Beispiel:** CountingSort(A[1..9], 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

2	0	1	1	0
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

2	0	1	1	1
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

```
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
```

```
for x = 1, ..., U do
```

```
    | B[x] = 0
```

```
    for i = 1, ..., n do
```

```
        | B[A[i]]++
```

```
C[1] = 0
```

```
for x = 1, ..., U - 1 do
```

```
    | C[x + 1] = C[x] + B[x]
```

```
for i = 1, ..., n do
```

```
    | D[C[A[i]] + 1] = A[i]  
    | C[A[i]]++
```

```
return D[1], ..., D[n]
```

**Beispiel:** CountingSort(A[1..9], 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

2	0	1	1	1
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

```
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
```

```
for x = 1, ..., U do
```

```
    | B[x] = 0
```

```
    for i = 1, ..., n do
```

```
        | B[A[i]]++
```

```
C[1] = 0
```

```
for x = 1, ..., U - 1 do
```

```
    | C[x + 1] = C[x] + B[x]
```

```
for i = 1, ..., n do
```

```
    | D[C[A[i]] + 1] = A[i]  
    | C[A[i]]++
```

```
return D[1], ..., D[n]
```

**Beispiel:** CountingSort(A[1..9], 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

2	0	1	1	2
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
									↑
B	2	0	1	1	2				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9], 5$ )

A	4	1	1	3	5	5	1	4	4
									↑
B	3	0	1	1	2				
		1	2	3	4	5			
C	1	1	1	1	1				
D	1	1	1	1	1	1	1	1	1
	2	3	4	5	6	7	8	9	

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

```
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
```

```
for x = 1, ..., U do
```

```
    | B[x] = 0
```

```
    for i = 1, ..., n do
```

```
        | B[A[i]]++
```

```
C[1] = 0
```

```
for x = 1, ..., U - 1 do
```

```
    | C[x + 1] = C[x] + B[x]
```

```
for i = 1, ..., n do
```

```
    | D[C[A[i]] + 1] = A[i]  
    | C[A[i]]++
```

```
return D[1], ..., D[n]
```

**Beispiel:** CountingSort(A[1..9], 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	1	2
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4
									↑
B	3	0	1	2	2				
	1	2	3	4	5				
C	1	1	1	1	1				
	1	2	3	4	5				
D	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4	↑
B	3	0	1	2	2					
	1	2	3	4	5					

C	⊥	⊥	⊥	⊥	⊥					
D	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	

	1	2	3	4	5	6	7	8	9	
--	---	---	---	---	---	---	---	---	---	--

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A	4	1	1	3	5	5	1	4	4	↑
B	3	0	1	3	2					
	1	2	3	4	5					

C	⊥	⊥	⊥	⊥	⊥					
D	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	

	1	2	3	4	5	6	7	8	9	
--	---	---	---	---	---	---	---	---	---	--

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
Initialisiere B[1 ... U], C[1 ... U], D[1 ... n]
for x = 1, ..., U do
    | B[x] = 0
    for i = 1, ..., n do
        | B[A[i]]++
C[1] = 0
for x = 1, ..., U - 1 do
    | C[x + 1] = C[x] + B[x]
for i = 1, ..., n do
    | D[C[A[i]] + 1] = A[i]
    | C[A[i]]++
return D[1], ..., D[n]
```

**Beispiel:** CountingSort( $A[1..9], 5$ )

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

⊥	⊥	⊥	⊥	⊥
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

1	1	1	1	1
---	---	---	---	---

$D$ 

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

0	1	1	1	1
---	---	---	---	---

$D$ 

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

0	3	1	1	1
---	---	---	---	---

$D$ 

1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

0	3	3	1	1
---	---	---	---	---

$D$ 

1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

0	3	3	4	1
---	---	---	---	---

$D$ 

1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

0	3	3	4	7
---	---	---	---	---

$D$ 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

0	3	3	4	7
---	---	---	---	---

$D$ 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

0	3	3	4	7
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

0	3	3	4	7
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

0	3	3	5	7
---	---	---	---	---

D 

⊥	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A    

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

  
     ↑

B    

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C    

0	3	3	5	7
---	---	---	---	---

D    

⊥	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A    

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

  
      ↑

B    

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C    

0	3	3	5	7
---	---	---	---	---

D    

1	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

1	3	3	5	7
---	---	---	---	---

D 

1	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

1	3	3	5	7
---	---	---	---	---

D 

1	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

1	3	3	5	7
---	---	---	---	---

D 

1	1	⊥	⊥	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

2	3	3	5	7
---	---	---	---	---

$D$ 

1	1	1	1	4	1	1	1	1
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

2	3	3	5	7
---	---	---	---	---

$D$ 

1	1	1	1	4	1	1	1	1
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	3	5	7
---	---	---	---	---

D 

1	1	⊥	3	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

2	3	4	5	7
---	---	---	---	---

$D$ 

1	1	1	3	4	1	1	1	1
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	4	5	7
---	---	---	---	---

D 

1	1	⊥	3	4	⊥	⊥	⊥	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

2	3	4	5	7
---	---	---	---	---

$D$ 

1	1	⊥	3	4	⊥	⊥	5	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

2	3	4	5	8
---	---	---	---	---

$D$ 

1	1	⊥	3	4	⊥	⊥	5	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	4	5	8
---	---	---	---	---

D 

1	1	⊥	3	4	⊥	⊥	5	⊥
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	4	5	8
---	---	---	---	---

D 

1	1	⊥	3	4	⊥	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	4	5	9
---	---	---	---	---

D 

1	1	⊥	3	4	⊥	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	4	5	9
---	---	---	---	---

D 

1	1	⊥	3	4	⊥	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

2	3	4	5	9
---	---	---	---	---

D 

1	1	1	3	4	⊥	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	5	9
---	---	---	---	---

D 

1	1	1	3	4	⊥	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	5	9
---	---	---	---	---

D 

1	1	1	3	4	⊥	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	5	9
---	---	---	---	---

D 

1	1	1	3	4	4	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	6	9
---	---	---	---	---

D 

1	1	1	3	4	4	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	6	9
---	---	---	---	---

D 

1	1	1	3	4	4	⊥	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	6	9
---	---	---	---	---

D 

1	1	1	3	4	4	4	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

A 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

 ↑

B 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

C 

3	3	4	7	9
---	---	---	---	---

D 

1	1	1	3	4	4	4	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

```
CountingSort(A[1 ... n], U)
```

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$   
  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

**Beispiel:** CountingSort( $A[1..9]$ , 5)

$A$ 

4	1	1	3	5	5	1	4	4
---	---	---	---	---	---	---	---	---

$B$ 

3	0	1	3	2
---	---	---	---	---

1 2 3 4 5

$C$ 

3	3	4	7	9
---	---	---	---	---

$D$ 

1	1	1	3	4	4	4	5	5
---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8 9

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt

→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

CountingSort( $A[1 \dots n]$ ,  $U$ )

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

|  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

|  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

|  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

|  $D[C[A[i]] + 1] = A[i]$

|  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

//Initialisiere  $B[1 \dots U]$  auf 0-Einträge

//Bestimme  $B[x] = B_x$

//Bestimme  $C[x] = C_x$

//Kopiere jedes  $A[i]$  an die gewünschte Stelle in  $D[C_x + 1, \dots, C_{x+1}]$

**Korrektheit:**

Invariante für die letzte for-Schleife: Für alle  $x$  gilt, dass  $C[x] + 1$  die nächste freie Position für ein Element mit Wert  $x$  ist.

# Counting Sort

**Idee:** Wir zählen für jedes  $x \in \{1, \dots, U\}$  wie häufig es vorkommt  
→ daraus lässt sich leicht bestimmen, wo jedes Element im sortierten Array  $A'$  hingehört

Sei  $B_x = |\{1 \leq i \leq n \mid A[i] = x\}|$  - Anzahl an Elementen mit Wert  $x$   
und  $C_x = \sum_{y=1}^{x-1} B_y$  - Anzahl an Elementen mit Wert  $< x$

Dann finden sich alle Elemente mit Wert  $x$  in  $A'[C_x + 1 \dots C_{x+1}]$

CountingSort( $A[1 \dots n]$ ,  $U$ )

Initialisiere  $B[1 \dots U]$ ,  $C[1 \dots U]$ ,  $D[1 \dots n]$

**for**  $x = 1, \dots, U$  **do**

  |  $B[x] = 0$

**for**  $i = 1, \dots, n$  **do**

  |  $B[A[i]]++$

$C[1] = 0$

**for**  $x = 1, \dots, U - 1$  **do**

  |  $C[x + 1] = C[x] + B[x]$

**for**  $i = 1, \dots, n$  **do**

  |  $D[C[A[i]] + 1] = A[i]$

  |  $C[A[i]]++$

**return**  $D[1], \dots, D[n]$

15

83

**Frage:**

Warum ist Counting Sort **nicht** vergleichsbasiert?

**Laufzeit:**

Jeder der vier Abschnitte läuft in Zeit  $O(n)$  oder  $O(U)$ .

⇒ Gesamlaufzeit  $O(n + U)$ .

# Stabiles Sortieren

---

## Problem: Generalisiertes Sortieren

gegeben: Liste  $E$  von Elementen  $e_1, \dots, e_n$  mit **Schlüsseln** (keys)  $k_i = \text{key}(e_i)$

gesucht: Permutation  $e'_1, \dots, e'_n$  von  $E$ , sortiert nach den Schlüsseln

Das heißt:  $\text{key}(e'_1) \leq \text{key}(e'_2) \leq \dots \leq \text{key}(e'_n)$

Im Folgenden sei die Schlüsselmenge  $K$  die natürlichen Zahlen

**Hinweis:** Wir können die bisherigen Sortieralgorithmen verwenden, wenn wir  $\text{key}(e_i)$  für  $A[i]$  verwenden.

## Definition

Ein Sortieralgorithmus ist **stabil**, wenn Elemente mit gleichem Schlüssel relativ zueinander die gleiche Reihenfolge wie in der Eingabe haben

D.h., wenn  $\text{key}(e_i) = \text{key}(e_j)$  für  $i < j$ , dann kommt  $e_i$  vor  $e_j$  in der sortierten Reihenfolge

## Bemerkung:

Counting Sort liefert ein **stabiles** Sortierverfahren.

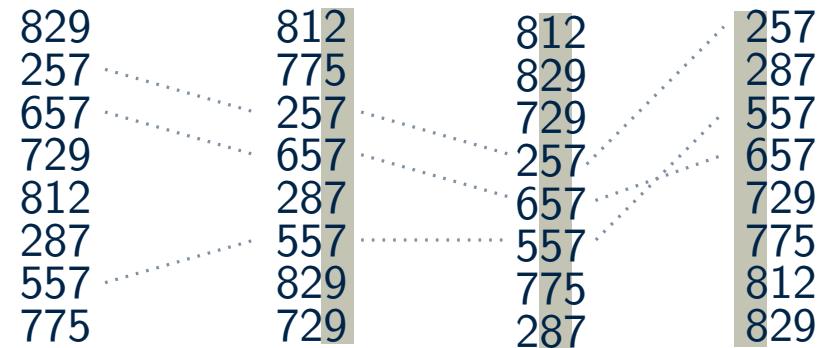
Frage: Warum?

# (LSD-)Radix Sort

**Problem:** Sortieren

gegeben: Liste von  $d$ -stelligen Zahlen  $A[1], \dots, A[n]$  zur Basis  $B$

gesucht: sortierte Reihenfolge von  $A[1], \dots, A[n]$



LSDRadixSort( $A[1 \dots n]$ ):

**for**  $i = 1, \dots, d$  **do**

sortiere  $A$  mit **stabilem** Sortierverfahren mit der  $i$ . niedrigsten Stelle als Schlüssel

**Theorem**

RadixSort (mithilfe CountingSort) sortiert  $d$ -stellige Zahlen  $A[1 \dots n]$  zur Basis  $B$  in Zeit  $O(d(n + B))$ .

# (LSD-)Radix Sort

---

**Problem:** Sortieren

gegeben: Liste von  $d$ -stelligen Zahlen  $A[1], \dots, A[n]$  zur Basis  $B$

gesucht: sortierte Reihenfolge von  $A[1], \dots, A[n]$

829	812	812	257
257	775	829	287
657	257	729	557
729	657	257	657
812	287	657	729
287	557	557	775
557	829	775	812
775	729	287	829

LSDRadixSort( $A[1 \dots n]$ ):

**for**  $i = 1, \dots, d$  **do**

sortiere  $A$  mit **stabilem** Sortierverfahren mit der  $i$ . niedrigsten Stelle als Schlüssel

**Theorem**

RadixSort (mithilfe CountingSort) sortiert  $d$ -stellige Zahlen  $A[1 \dots n]$  zur Basis  $B$  in Zeit  $O(d(n + B))$ .

Frage: Was für eine Schleifeninvariante kann man verwenden um Korrektheit zu beweisen?

# Zusammenfassung

---

Können wir also in Zeit  $o(n \log n)$  sortieren?

- für vergleichsbasierte Sortieralgorithmen (z.B. MergeSort) ist  $\Theta(n \log n)$  optimal  
Grund:  $\Omega(n \log n)$  untere Schranke im Entscheidungsbaummodell
- für besondere Eingaben können wir schneller sortieren:
  - Counting Sort sortiert Zahlen in  $\{1, \dots, U\}$  in Zeit  $O(n + U)$  und ist **stabil**.
  - Radix Sort sortiert  $d$ -stellige Zahlen zur Basis  $B$  in Zeit  $O(d(n + B))$

**Hinweis:** Diese Algorithmen sind nicht vergleichsbasiert

## Ausblick:

Schnellste Sortierverfahren nach wie vor Forschungsgegenstand:

Es gibt RAM-Algorithmen mit Laufzeit  $O(n \log \log n)$  [Andersson, Hagerup, Nilsson, Raman 1998]  
sowie randomisiert in  $O(n \sqrt{\log \log n})$  [Han, Thorup 2002]

Genauer gesagt, eine Version unseres RAM-Modells mit Registern, die  
w-bit Zahlen speichern und auch bitweise Operationen unterstützen  
"word size"  $w \in \Omega(\log n)$



## Algorithmen und Datenstrukturen SS'23

# Kapitel 6: Quicksort

Marvin Künemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

**Letztes Kapitel:** Können wir in  $o(n \log n)$  Zeit sortieren?

**Jetzt:** Yet another  $O(n^2)$  Sortierverfahren: Quicksort



Insbesondere besprechen wir in diesem Kapitel:

- Quicksort, ein in der Praxis häufig verwendetes Sortierverfahren
- aber: Quicksort benötigt  $\Omega(n^2)$ -Zeit im worst-case!
- es stellt sich allerdings heraus, dass:
  - es eine randomisierte Variante mit geringerer erwarteter Laufzeit im worst-case hat
  - es eine schnelle deterministische Variante hat
    - nicht randomisiert!
- um die schnelle deterministische Variante zu erhalten, besprechen wir "en route":
  - **Rangselektion** (Auswahlproblem) in Zeit  $O(n)$   
→ Bestimmung von Median, Quartilen, etc.

# Quicksort: Idee

Idee: · Wir bestimmen ein **Pivotelement**  $p$

· Wir **teilen** das Problem in zwei Teile:

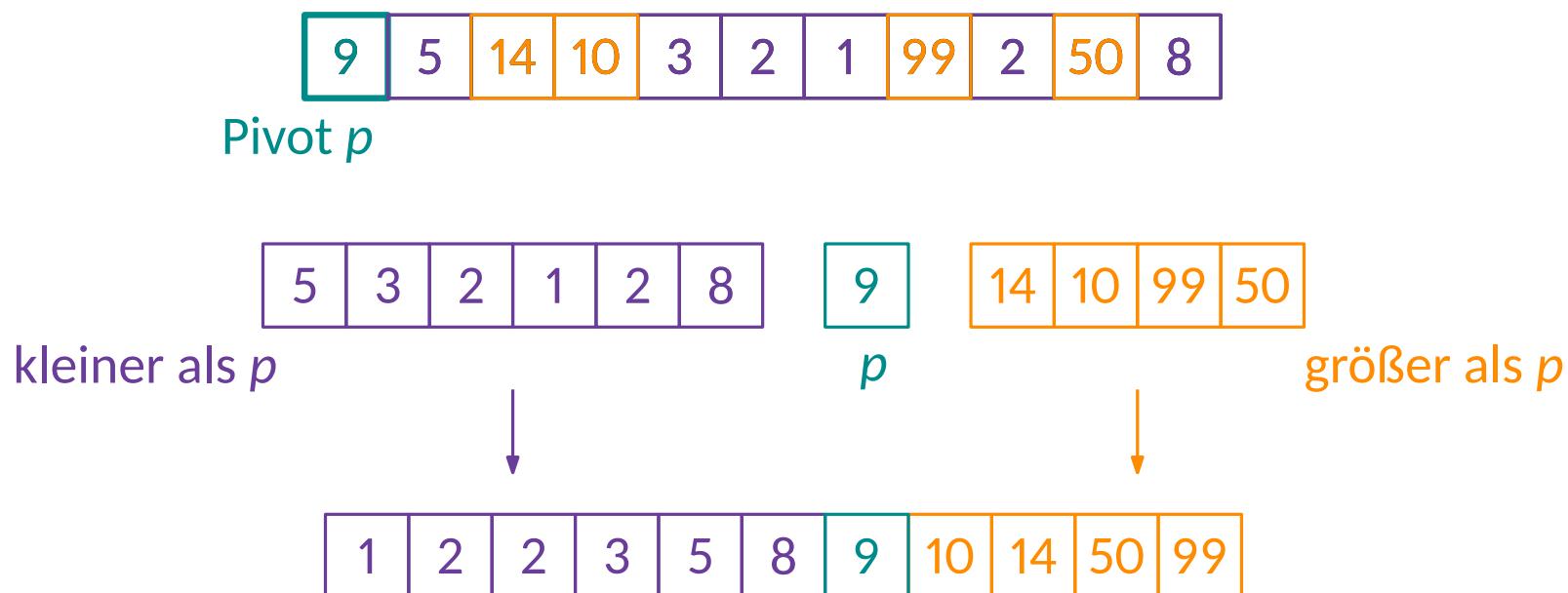
- in einem sind alle Elemente  $\leq p$
- im anderen sind alle Elemente  $\geq p$

· Wir sortieren diese Teile getrennt voneinander

· Wir fügen das Ergebnis zusammen

wichtig zu klären: Wie wählen wir das Pivotelement?

Wie bestimmen wir die Teilprobleme?



# Quicksort: Pseudocode

Hinweis: Übergabe des Arrays als "Referenz" → wir verändern A

```
QuickSort(A[1 ... n], ℓ, r)
```

```
  if (ℓ < r) then
```

```
    q = Partition(A, ℓ, r)
```

```
    QuickSort(A, ℓ, q - 1)
```

```
    QuickSort(A, q + 1, r)
```

Ausgabe: Hinterher ist  $A[\ell \dots r]$  sortiert

// permutiert A sodass ab jetzt gilt:  $A[i] \leq A[q]$  für alle  $\ell \leq i < q$  und  
 $A[q] \leq a[j]$  für alle  $q < j \leq r$

**Korrektheit (Beweisskizze):** Induktion über  $r - \ell$

Quicksort permutiert die Elemente in  $A[\ell \dots r]$  sodass:

Korrektheit von  $\text{Partition}(A[\ell \dots r])$

$$A[\ell] \leq \cdots \leq \underline{A[q-1]} \leq \underline{A[q]} \leq \underline{A[q+1]} \leq \cdots \leq A[r]$$

Korrektheit von  $\text{QuickSort}(A[\ell, q-1])$

Korrektheit von  $\text{QuickSort}(A[q+1, r])$

# Partition: Pseudocode

```
Partition(A[1, ..., n], ℓ, r)
```

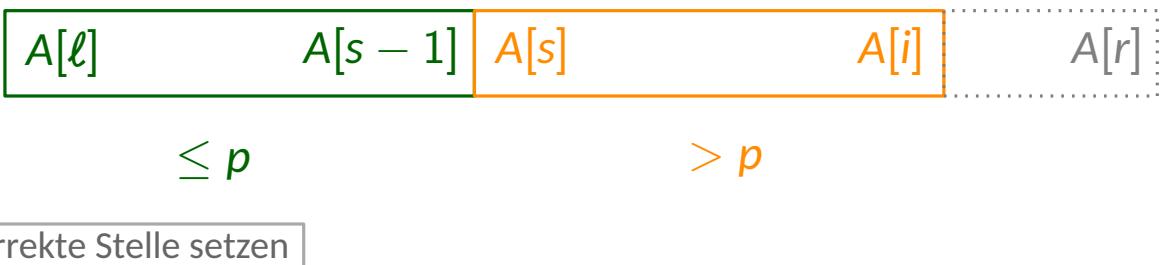
```
    k = pickPivot(A, ℓ, r)      // Wahl des Pivotelements  
    p = A[k]  
    swap(A[k], A[r])          // temporär ans Ende setzen
```

```
    s = ℓ  
    for (i = ℓ, ..., r - 1) do  
        if (A[i] ≤ p) do  
            swap(A[i], A[s])  
            s = s + 1  
    swap(A[s], A[r])          // Pivot an die korrekte Stelle setzen  
  
    return s
```

Ausgabe: Index  $\ell \leq q \leq r$  sodass:

- alle Elemente in  $A[\ell \dots q - 1]$  sind höchstens  $A[q]$ , und
- alle Elemente in  $A[q + 1 \dots r]$  sind mindestens  $A[q]$

**Schleifeninvariante:** Am Ende jeder Iteration gilt:



# Partition: Pseudocode

```
Partition(A[1, ..., n], ℓ, r)
```

```
    k = pickPivot(A, ℓ, r)      // Wahl des Pivotelements  
    p = A[k]  
    swap(A[k], A[r])          // temporär ans Ende setzen
```

```
    s = ℓ  
    for (i = ℓ, ..., r - 1) do  
        if (A[i] ≤ p) do  
            swap(A[i], A[s])  
            s = s + 1  
    swap(A[s], A[r])          // Pivot an die korrekte Stelle setzen  
    return s
```

Ausgabe: Index  $\ell \leq q \leq r$  sodass:

- alle Elemente in  $A[\ell \dots q - 1]$  sind höchstens  $A[q]$ , und
- alle Elemente in  $A[q + 1 \dots r]$  sind mindestens  $A[q]$

**Schleifeninvariante:** Am Ende der for-Schleife gilt:



**Frage:** Was ist die Laufzeit?

Für den Anfang betrachten wir folgende Pivotwahl:

```
pickPivot(A, ℓ, r)  
return ℓ
```

// Wähle das erste Element des Arrays als Pivot

# Quicksort: Eigenschaften

```
QuickSort(A[1 ... n], ℓ, r)
```

```
  if ( $\ell < r$ ) then
```

```
    q = Partition(A, ℓ, r)
```

```
    QuickSort(A, ℓ, q - 1)
```

```
    QuickSort(A, q + 1, r)
```

```
pickPivot(A, ℓ, r)
```

```
  return ℓ
```

```
Partition(A[1, ..., n], ℓ, r)
```

```
  k = pickPivot(A, ℓ, r)
```

```
  p = A[k]
```

```
  swap(A[k], A[r])
```

```
  s = ℓ
```

```
  for ( $i = \ell, \dots, r - 1$ ) do
```

```
    if ( $A[i] \leq p$ ) do
```

```
      swap(A[i], A[s])  
      s = s + 1
```

```
  swap(A[s], A[r])
```

```
  return s
```

## Eigenschaften

- Quicksort arbeitet **in place**

benötigt nur  $O(1)$  zusätzlichen Speicher

→ kein zusätzliches Array nötig, nur Vertauschungen

- Laufzeit hängt von der **Wahl des Pivotelementes** ab!

Sei  $T(n)$  die Laufzeit von  $\text{Quicksort}(A, \ell, r)$  mit  $n = r - \ell + 1$ . Dann gilt:

$$T(n) \leq T(q - \ell) + T(r - q) + Cn$$

- Unsere bisherige Wahl resultiert in einer Worst-Case Laufzeit von  $\Omega(n^2)$

Wähle das erste Element der Liste als Pivot

Frage: Warum?

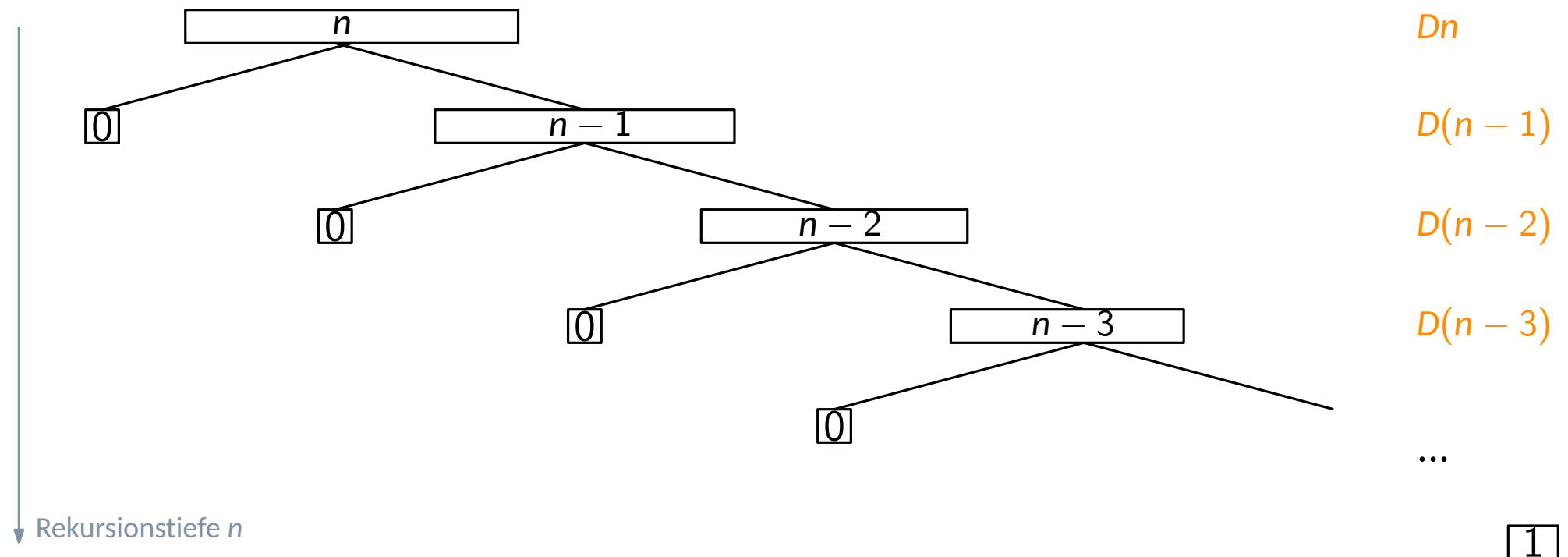
↗ Tafelpräsentation

# Quicksort: Tatsächlich langsam?

Ist Quicksort tatsächlich so langsam?

- auf vielen unsortierten Eingaben ist Quicksort erstaunlich effizient

Wann tritt der **schlechteste Fall** ein?



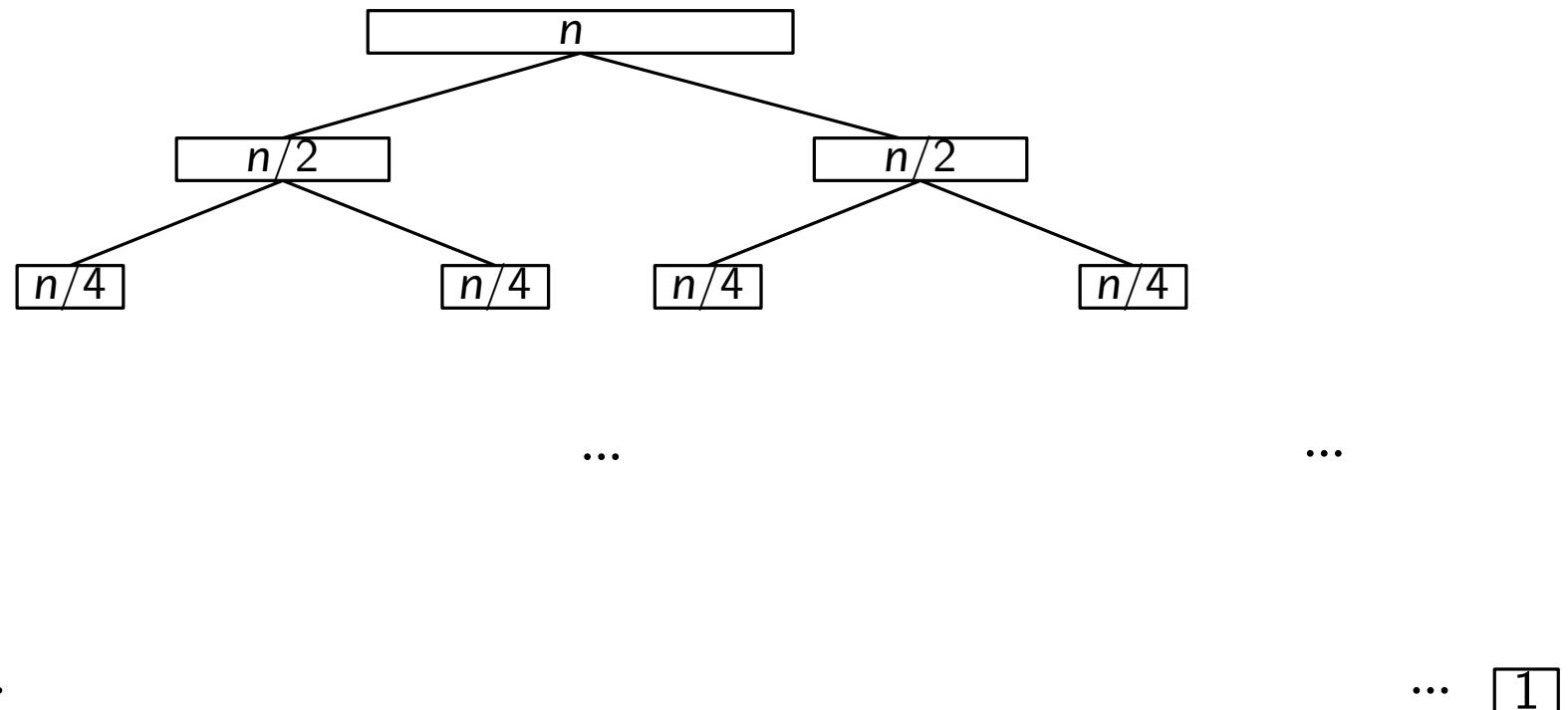
$$\Rightarrow T(n) \geq \sum_{i=1}^n Di = D \sum_{i=1}^n i = D \cdot \frac{n(n+1)}{2} \in \Omega(n^2)$$

# Quicksort: Tatsächlich langsam?

Ist Quicksort tatsächlich so langsam?

- auf vielen unsortierten Eingaben ist Quicksort erstaunlich effizient

Wann tritt der **beste Fall** ein?



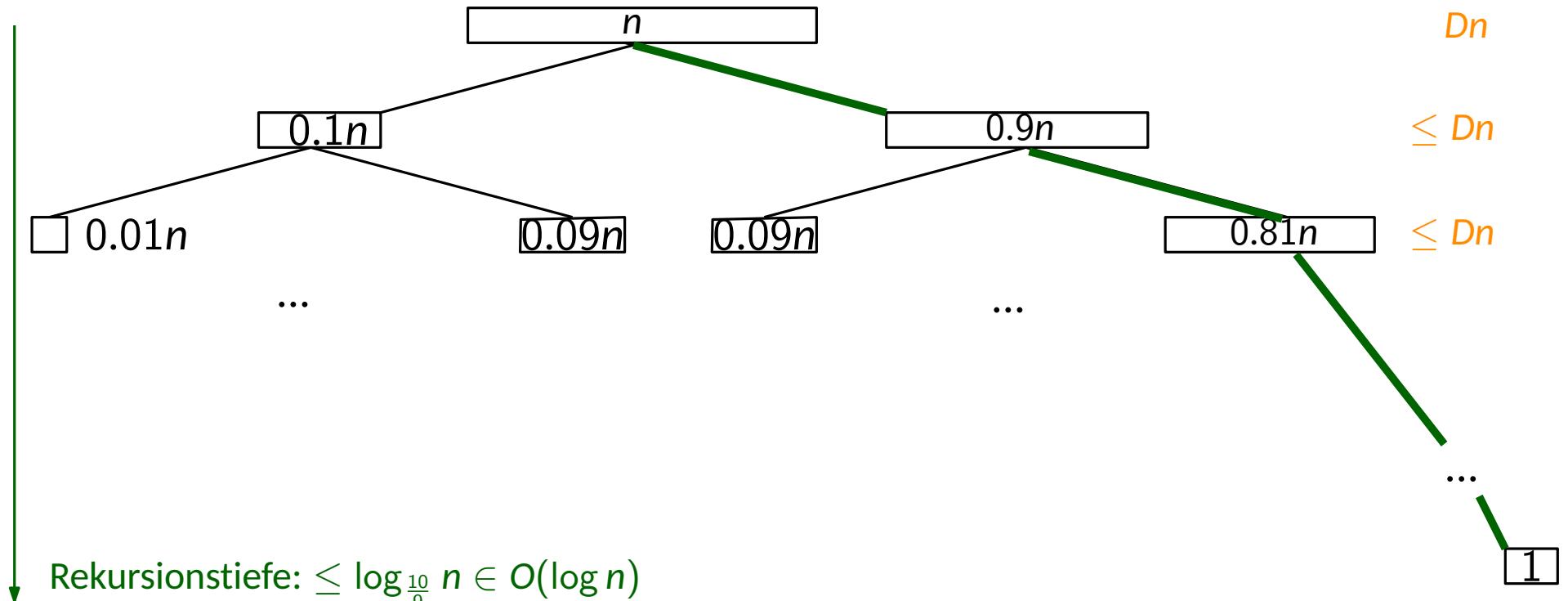
$$\begin{aligned}T(n) &\leq 2 \cdot T(n/2) + Dn \\ \Rightarrow T(n) &\in O(n \log n)\end{aligned}$$

# Quicksort: Tatsächlich langsam?

Ist Quicksort tatsächlich so langsam?

Was wäre, wenn wir jedes Mal "halbwegs" balanciert aufteilen?

z.B. jedes Subproblem der Größe  $n$  teilt sich in eins der Größe  $0.1n$  und eins der Größe  $0.9n$  auf



$$\Rightarrow T(n) \leq \log_{\frac{10}{9}}(n) \cdot Dn \in O(n \log n)$$

Hinweis:

Die  $O(n \log n)$ -Schranke würde auch folgen, wenn wir jedes Mal aufteilen würden in  $0.000001n$  und  $0.999999n$ .

# Pivotwahl

---

Können wir ein Pivot wählen, um möglichst balanciert aufteilen?

Wir benutzen ein hilfreiches Prinzip: **Randomisierung** (Zufall)

# Exkurs: Randomisierte Algorithmen (Las Vegas)

Randomisierte Algorithmen passen nicht in unser bisheriges Berechnungsmodell der RAM  
→ die Ausführung war bisher immer eindeutig vorgegeben (**deterministisch**), unabhängig von weiteren Einflüssen wie Zufall

Für randomisierte Algorithmen erweitern wir also unser Modell um eine weitere Instruktion:

**RAND C:** erzeugt eine uniform zufällige Zahl in  $\{1, \dots, C\}$  und schreibt sie in den Akkumulator  
+ zusätzlich Varianten mit (indirekter) Registeraddressierung

Beispiele: RAND 2 entspricht dem Ergebnis eines Münzwurfs  
RAND 6 entspricht dem Ergebnis eines 6-seitigen Würfels  
RAND 20 entspricht dem Ergebnis eines D20

RAND( $\ell, r$ ):

**return**  $(\ell - 1) + \text{RAND } (r - \ell + 1)$

// liefert zufällige Zahl in  $\{\ell, \dots, r\}$

## Definition

Ein randomisierter Algorithmus A ist ein **Las-Vegas-Algorithmus** für eine Funktion f, wenn für jede Eingabe x gilt, dass sobald A terminiert, das Ergebnis der Berechnung  $f(x)$  ist.

Wir nennen  $T_A(n) = \max_{x \in I_n} E[t_A(x)]$  die **erwartete Laufzeit** von A.

Hinweis: die erwartete Laufzeit im schlimmsten Fall!

# Randomisierte Variante von Quicksort

---

Idee: Pivotelement jedes Mal uniform zufällig wählen

```
pickPivot( $A, \ell, r$ )
| return RAND( $\ell, r$ )
```

//Wähle eine uniform zufällige Position aus  $\{\ell, \dots, r\}$

→ erweist sich in der Praxis als schnell

## Theorem

Die erwartete Laufzeit von Quicksort mit uniform zufälligem Pivot ist in  $O(n \log n)$ .

# Exkurs: Wahrscheinlichkeitstheorie (Grundlagen)

Ergebnisraum  $S$

$$S = \{1, \dots, 6\} \times \{1, \dots, 6\}$$

Was kann eintreten?

Wahrscheinlichkeitstraum  $(S, \Pr)$

$$S \text{ mit } \Pr[s] = \frac{1}{36} \text{ für jedes } s \in S$$

Ergebnisraum

+ Wahrscheinlichkeiten für jedes Elementarereignis

Ereignis  $E \subseteq S$  z.B.  $E = \{(a, b) \in S \mid a + b = 4\}$

Menge von Elementarereignissen

Wahrscheinlichkeit von  $E$ :  $\Pr[E] = \frac{3}{36}$

$$\Pr[E] = \sum_{s \in E} \Pr[s]$$

Zufallsvariable  $X : S \rightarrow \mathbb{N}$

$$\text{z.B. } X((a, b)) = a + b$$

Erwartungswert von  $X$ :

$$E[X] = \sum_{s \in S} \Pr[s]X(s) = \sum_{n \in \mathbb{N}} \Pr[X = n] \cdot n$$

Linearität des Erwartungswertes:

$$E[X + Y] = E[X] + E[Y]$$

Beispiel:

```
a = rand(6)  
b = rand(6)  
return a+b
```

$$A((a, b)) = a, \quad B((a, b)) = b$$

$$E[A] = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = 3.5,$$

$$E[B] = E[A] = 3.5,$$

$$E[X] = E[A + B] = E[A] + E[B] = 7$$

# Exkurs: Wahrscheinlichkeitstheorie (Grundlagen)

Ergebnisraum  $S$

$$S = \{1, \dots, 6\} \times \{1, \dots, 6\}$$

Was kann eintreten?

Wahrscheinlichkeitstraum  $(S, \Pr)$

$$S \text{ mit } \Pr[s] = \frac{1}{36} \text{ für jedes } s \in S$$

Ergebnisraum

+ Wahrscheinlichkeiten für jedes Elementarereignis

Ereignis  $E \subseteq S$  z.B.  $E = \{(a, b) \in S \mid a + b = 4\}$

Menge von Elementarereignissen

Wahrscheinlichkeit von  $E$ :  $\Pr[E] = \frac{3}{36}$

$$\Pr[E] = \sum_{s \in E} \Pr[s]$$

Zufallsvariable  $X : S \rightarrow \mathbb{N}$

$$\text{z.B. } X((a, b)) = a + b$$

Erwartungswert von  $X$ :

$$\mathbb{E}[X] = \sum_{s \in S} \Pr[s]X(s) = \sum_{n \in \mathbb{N}} \Pr[X = n] \cdot n$$

Linearität des Erwartungswertes:

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

Beispiel:

```
a = rand(6)  
b = rand(6)  
return a+b
```

$$A((a, b)) = a, \quad B((a, b)) = b$$

$$\mathbb{E}[A] = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = 3.5,$$

$$\mathbb{E}[B] = \mathbb{E}[A] = 3.5,$$

$$\mathbb{E}[X] = \mathbb{E}[A + B] = \mathbb{E}[A] + \mathbb{E}[B] = 7$$

Unabhängigkeit:

Zufallsvariablen  $X, Y$  sind **unabhängig**, wenn für alle  $x, y$  gilt:

$$\Pr[X = x \text{ und } Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$$

Allgemeiner:  $X_1, \dots, X_k$  unabhängig wenn für alle  $x_1, \dots, x_k$  gilt, dass:

$$\Pr[\bigwedge_{i=1}^k X_i = x_i] = \prod_{i=1}^k \Pr[X_i = x_i]$$

# Analyse: Quicksort mit uniform zufälligem Pivot

```
pickPivot( $A, \ell, r$ )
| return RAND( $\ell, r$ )
```

//Wähle eine uniform zufällige Position aus  $\{\ell, \dots, r\}$

## Theorem

Die erwartete Laufzeit von Quicksort mit uniform zufälligem Pivot ist in  $O(n \log n)$ .

**Beweis:** Es sei  $X$  die Anzahl an Vergleichen von Quicksort

Es sei  $A'[1] \leq \dots \leq A'[n]$  die sortierte Reihenfolge des Eingabe-Arrays

Wir definieren  $X_{i,j} = \begin{cases} 1 & \text{wenn } A'[i] \text{ irgendwann gegen } A'[j] \text{ verglichen wird} \\ 0 & \text{ansonsten.} \end{cases}$   
Indikatorvariable

**Lemma:** Für  $i < j$  gilt:  $\Pr[X_{i,j} = 1] = \frac{2}{j-i+1}$

**Beweis:** Betrachte  $I = \{A'[i], \dots, A'[j]\}$

Solange Quicksort ein Pivot  $p \notin I$  wählt:

- werden  $A'[i]$  und  $A'[j]$  momentan nicht verglichen
- befinden sich alle Elemente in  $I$  in genau einem der rekursiven Aufrufe

Sobald Quicksort ein Pivot  $p \in I$  wählt, gibt es drei Fälle:

- $p = A'[i]$ . Es folgt  $X_{i,j} = 1$
- $p = A'[j]$ . Es folgt  $X_{i,j} = 1$
- $p = A'[k]$  für  $i < k < j$ . Es folgt  $X_{i,j} = 0$

$$\Pr[X_{i,j} = 1] = \frac{2}{|I|} = \frac{2}{j-i+1} \quad \square$$

# Analyse: Quicksort mit uniform zufälligem Pivot

```
pickPivot( $A, \ell, r$ )
| return RAND( $\ell, r$ )
```

//Wähle eine uniform zufällige Position aus  $\{\ell, \dots, r\}$

## Theorem

Die erwartete Laufzeit von Quicksort mit uniform zufälligem Pivot ist in  $O(n \log n)$ .

**Beweis:** Es sei  $X$  die Anzahl an Vergleichen von Quicksort

Es sei  $A'[1] \leq \dots \leq A'[n]$  die sortierte Reihenfolge des Eingabe-Arrays

Wir definieren  $X_{i,j} = \begin{cases} 1 & \text{wenn } A'[i] \text{ irgendwann gegen } A'[j] \text{ verglichen wird} \\ 0 & \text{ansonsten.} \end{cases}$   
Indikatorvariable

**Lemma:** Für  $i < j$  gilt:  $\Pr[X_{i,j} = 1] = \frac{2}{j-i+1}$

$$\begin{aligned} E[X] &= E \left[ \sum_{i=1}^n \sum_{j=i+1}^n X_{i,j} \right] = \sum_{i=1}^n \sum_{j=i+1}^n E[X_{i,j}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \Pr[X_{i,j} = 1] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &\leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} = 2n \sum_{k=2}^n \frac{1}{k} \end{aligned}$$

# Analyse: Quicksort mit uniform zufälligem Pivot

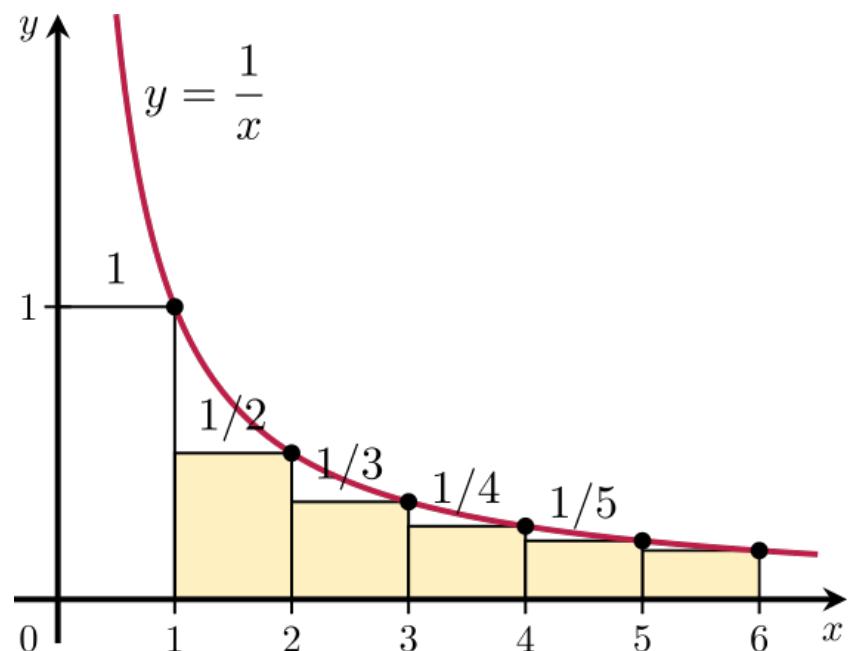
```
pickPivot( $A, \ell, r$ )
| return RAND( $\ell, r$ )
```

//Wähle eine uniform zufällige Position aus  $\{\ell, \dots, r\}$

## Theorem

Die erwartete Laufzeit von Quicksort mit uniform zufälligem Pivot ist in  $O(n \log n)$ .

## Beweis:



$$\sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx = \ln n$$

$$\leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} = 2n \sum_{k=2}^n \frac{1}{k} \leq 2n \ln n$$

# Analyse: Quicksort mit uniform zufälligem Pivot

```
pickPivot( $A, \ell, r$ )
| return RAND( $\ell, r$ )
```

//Wähle eine uniform zufällige Position aus  $\{\ell, \dots, r\}$

## Theorem

Die erwartete Laufzeit von Quicksort mit uniform zufälligem Pivot ist in  $O(n \log n)$ .

**Beweis:** Es sei  $X$  die Anzahl an Vergleichen von Quicksort

Es sei  $A'[1] \leq \dots \leq A'[n]$  die sortierte Reihenfolge des Eingabe-Arrays

Wir definieren  $X_{i,j} = \begin{cases} 1 & \text{wenn } A'[i] \text{ irgendwann gegen } A'[j] \text{ verglichen wird} \\ 0 & \text{ansonsten.} \end{cases}$   
Indikatorvariable

**Lemma:** Für  $i < j$  gilt:  $\Pr[X_{i,j} = 1] = \frac{2}{j-i+1}$

$$\begin{aligned} E[X] &= E \left[ \sum_{i=1}^n \sum_{j=i+1}^n X_{i,j} \right] = \sum_{i=1}^n \sum_{j=i+1}^n E[X_{i,j}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \Pr[X_{i,j} = 1] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &\leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} = 2n \sum_{k=2}^n \frac{1}{k} \leq 2n \ln n \end{aligned}$$

# Illustration: Quicksort

---

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quicksort

---

Let's Part...

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quicksort

---

Let's Partition:

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quicksort

---

Let's Partition:

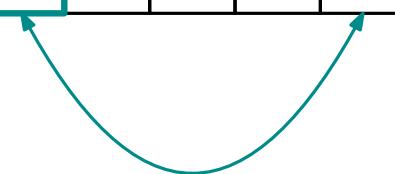
9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quicksort

---

Let's Partition:

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---



# Illustration: Quicksort

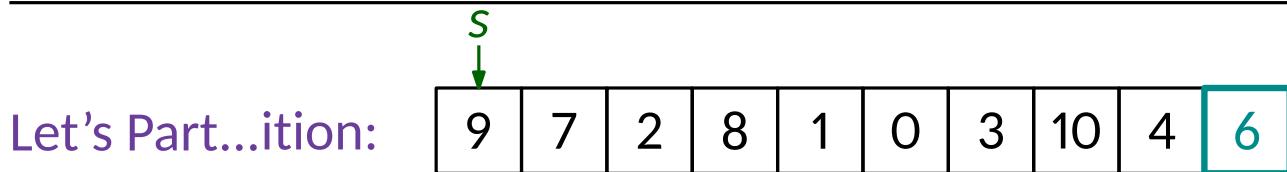
---

Let's Partition:

9	7	2	8	1	0	3	10	4	6
---	---	---	---	---	---	---	----	---	---

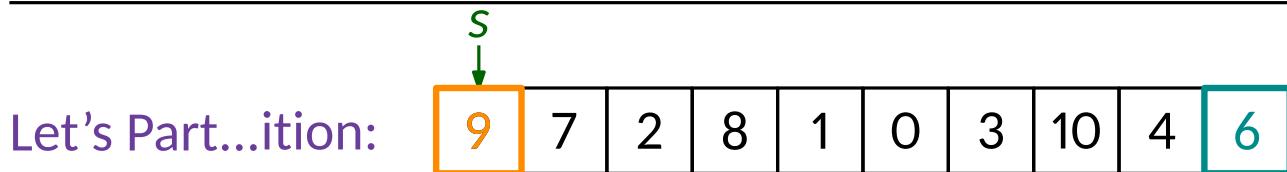
# Illustration: Quicksort

---



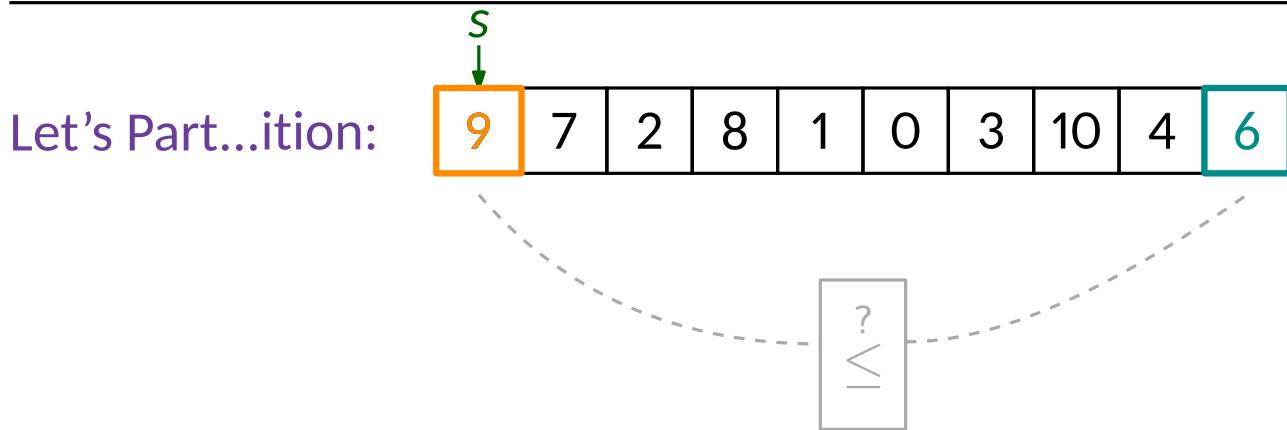
# Illustration: Quicksort

---



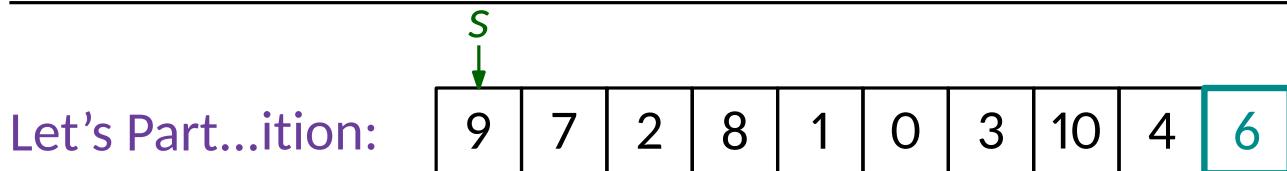
# Illustration: Quicksort

---



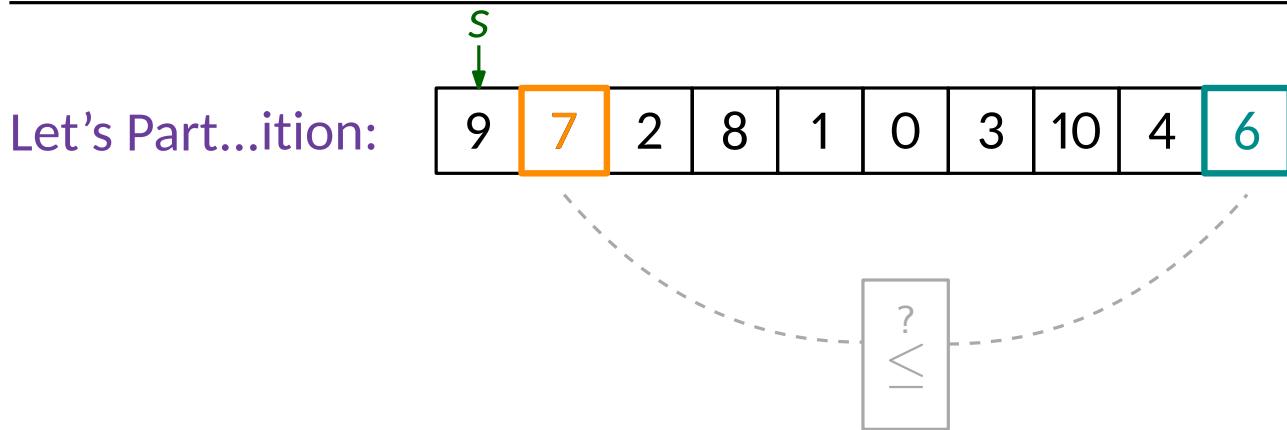
# Illustration: Quicksort

---



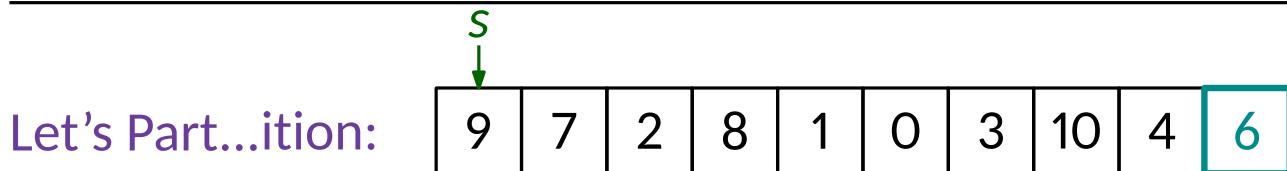
# Illustration: Quicksort

---



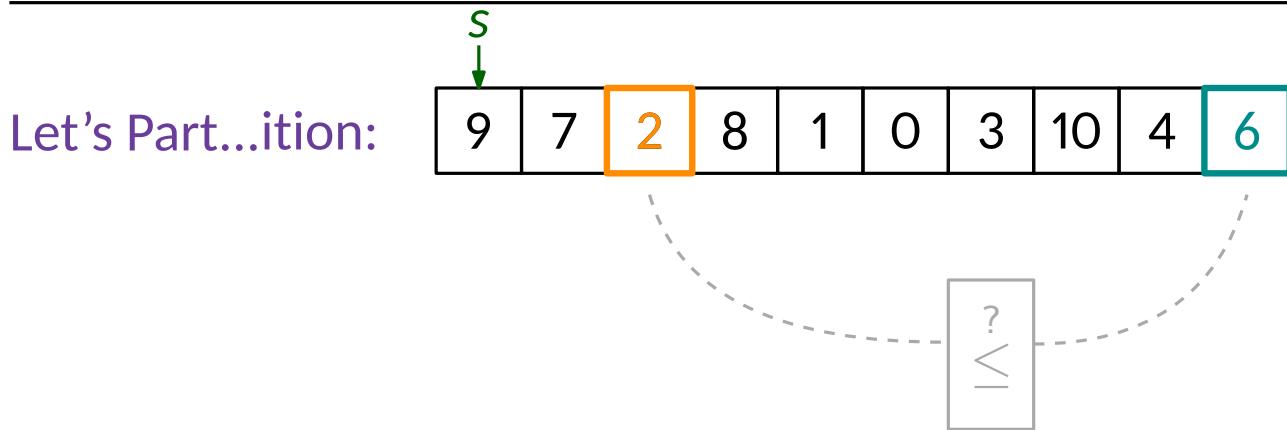
# Illustration: Quicksort

---



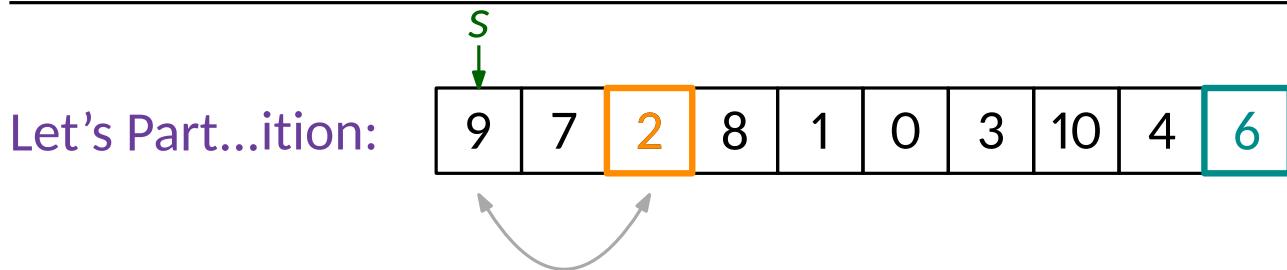
# Illustration: Quicksort

---



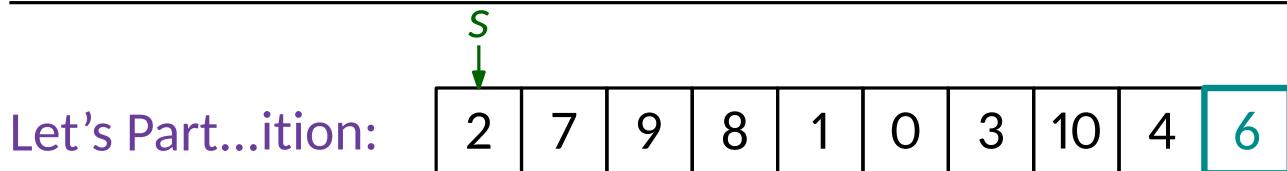
# Illustration: Quicksort

---



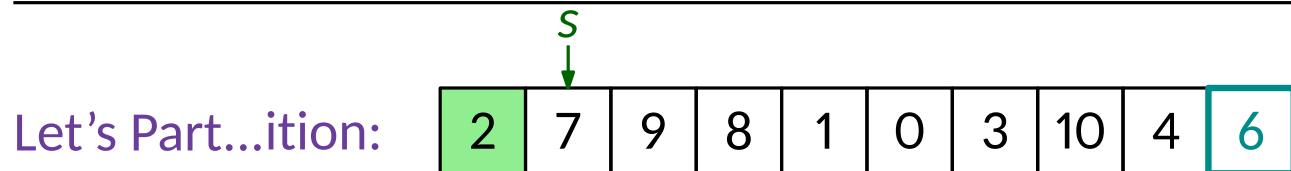
# Illustration: Quicksort

---



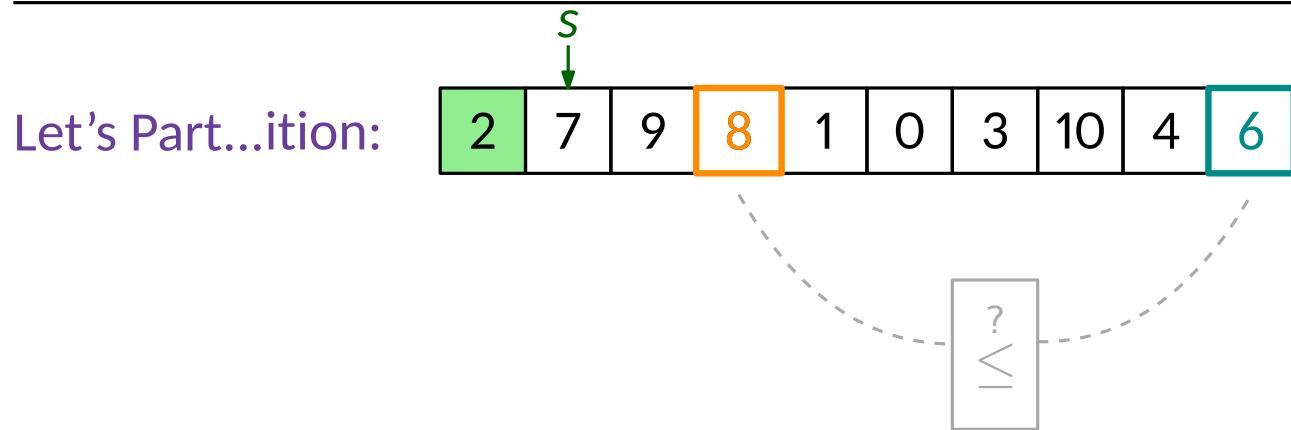
# Illustration: Quicksort

---



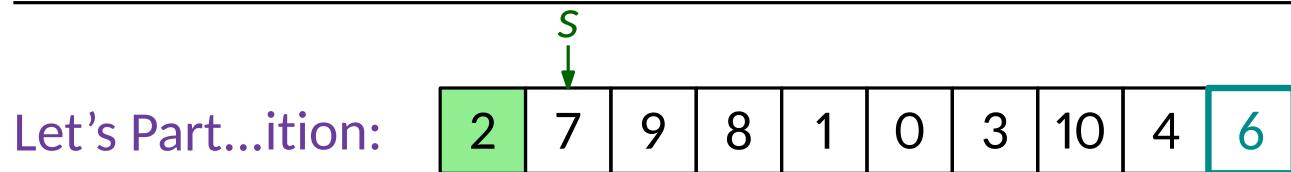
# Illustration: Quicksort

---



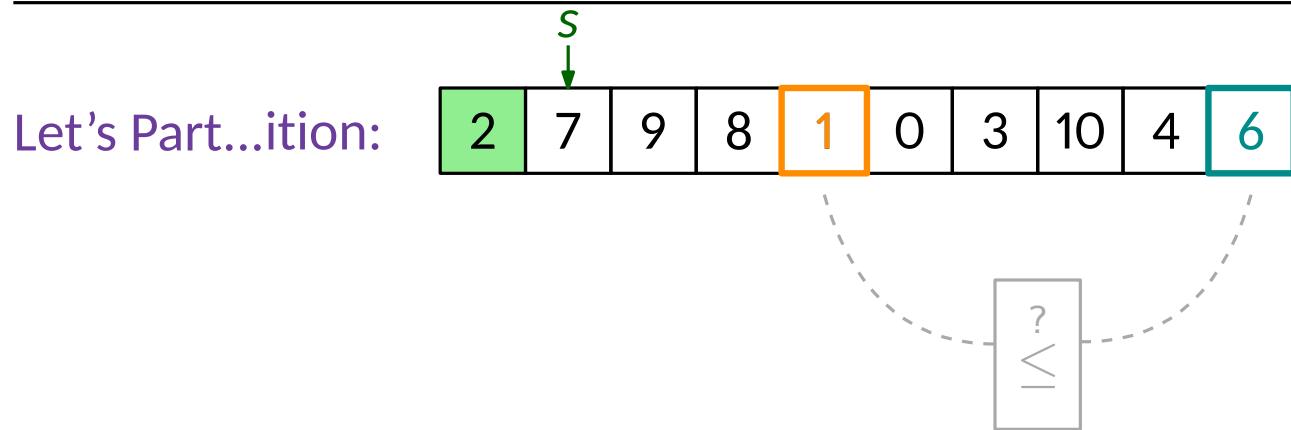
# Illustration: Quicksort

---



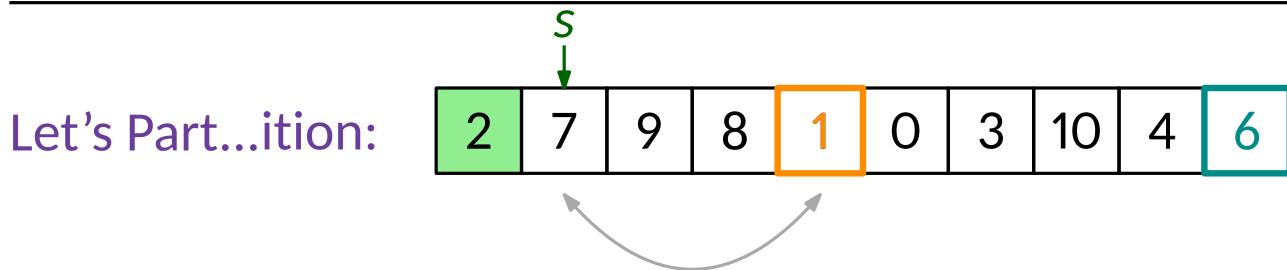
# Illustration: Quicksort

---



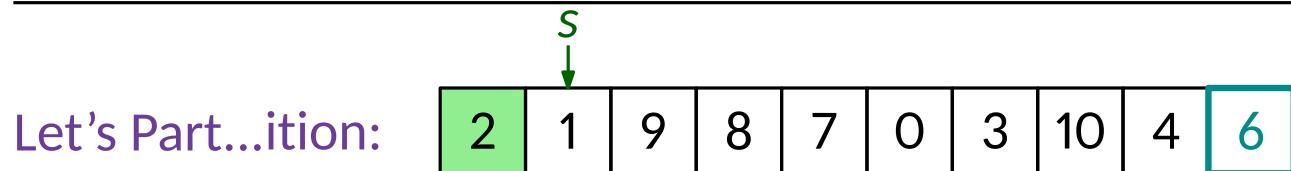
# Illustration: Quicksort

---



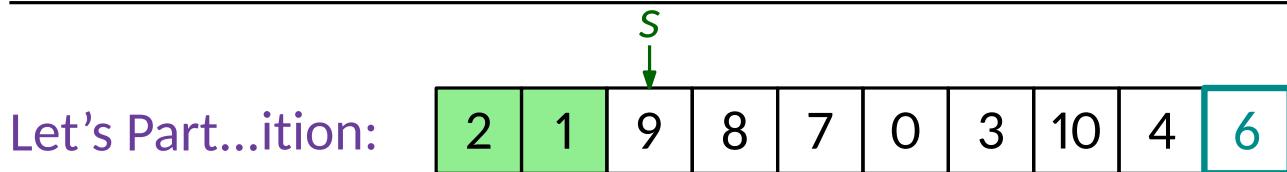
# Illustration: Quicksort

---



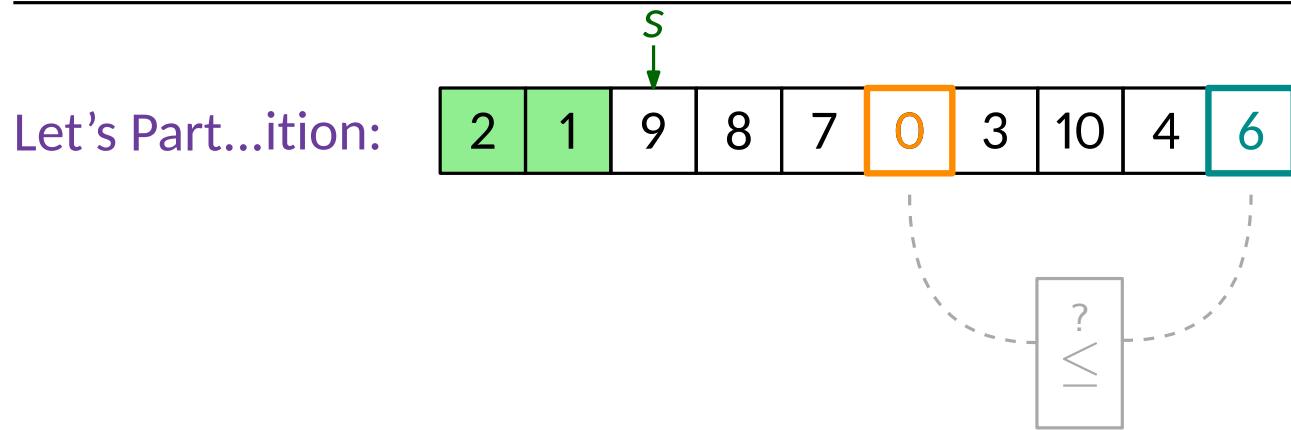
# Illustration: Quicksort

---



# Illustration: Quicksort

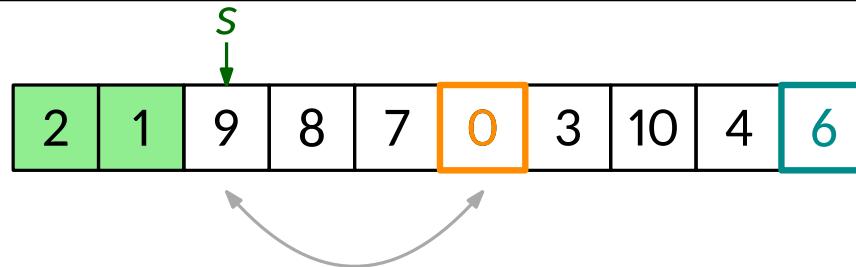
---



# Illustration: Quicksort

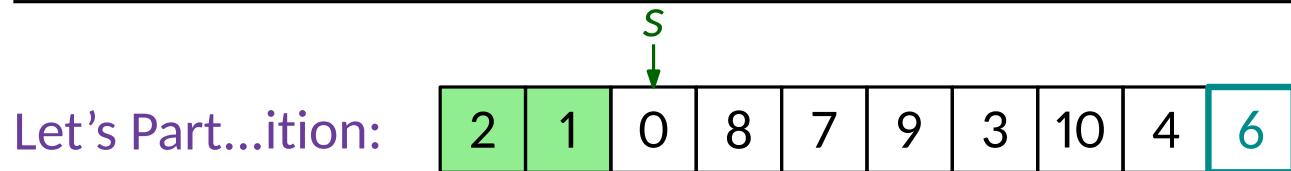
---

Let's Partition:



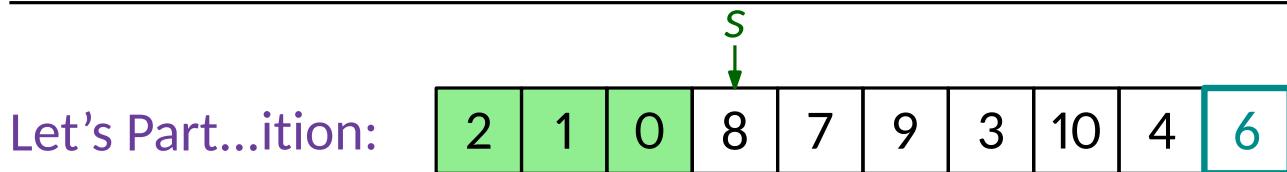
# Illustration: Quicksort

---



# Illustration: Quicksort

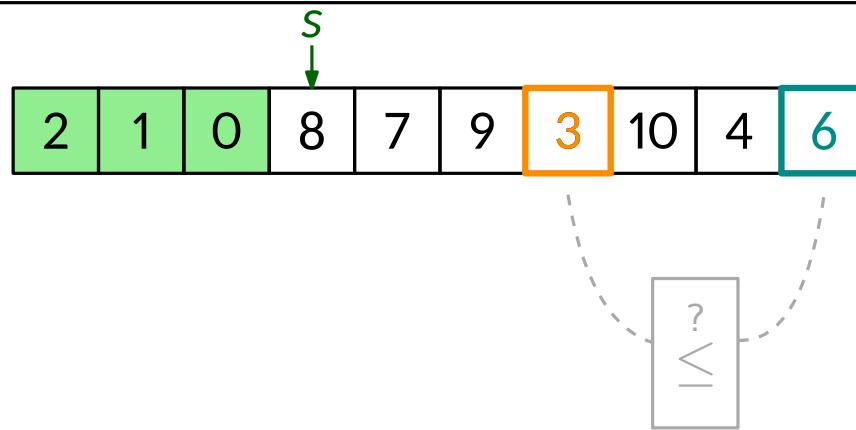
---



# Illustration: Quicksort

---

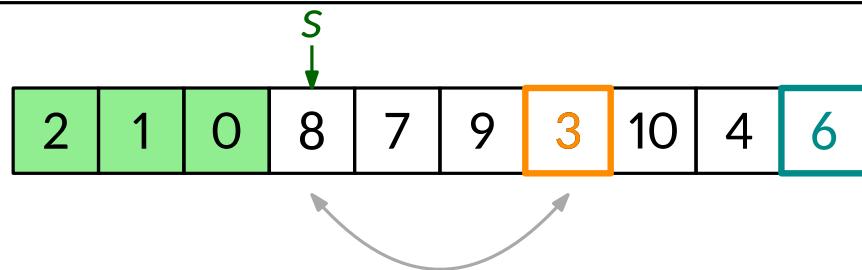
Let's Partition:



# Illustration: Quicksort

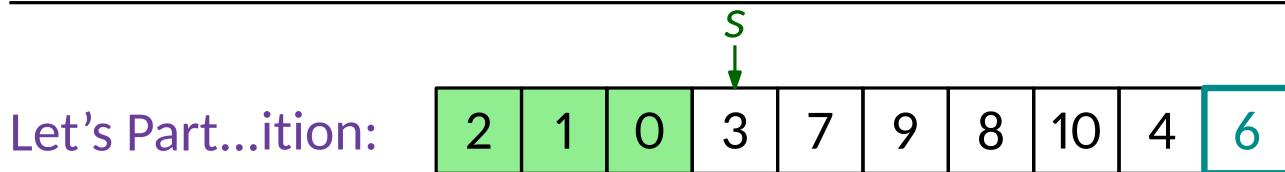
---

Let's Partition:



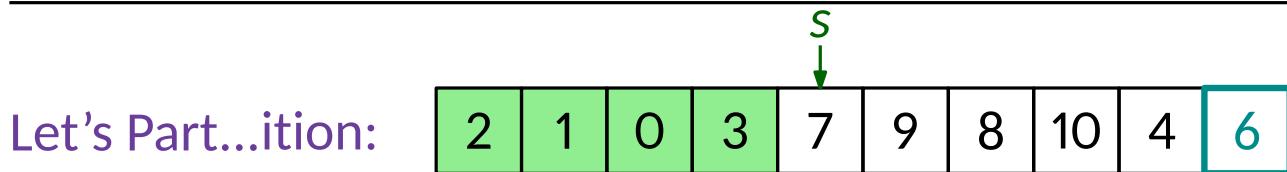
# Illustration: Quicksort

---



# Illustration: Quicksort

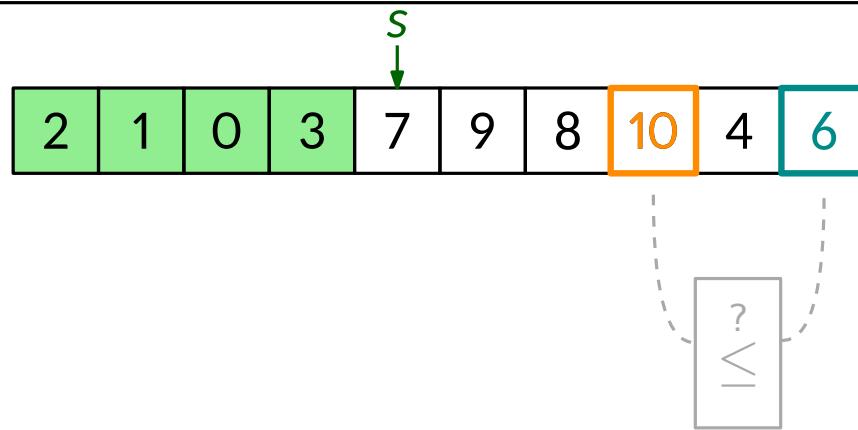
---



# Illustration: Quicksort

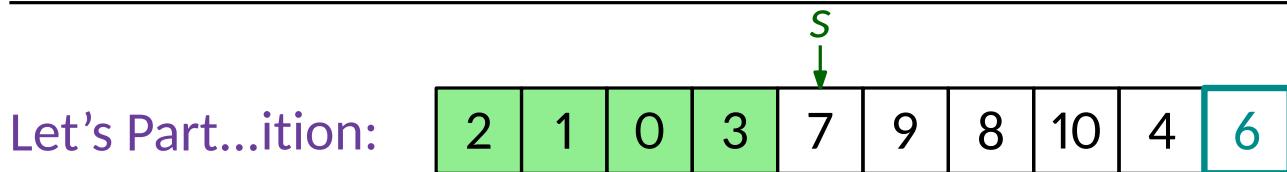
---

Let's Partition:



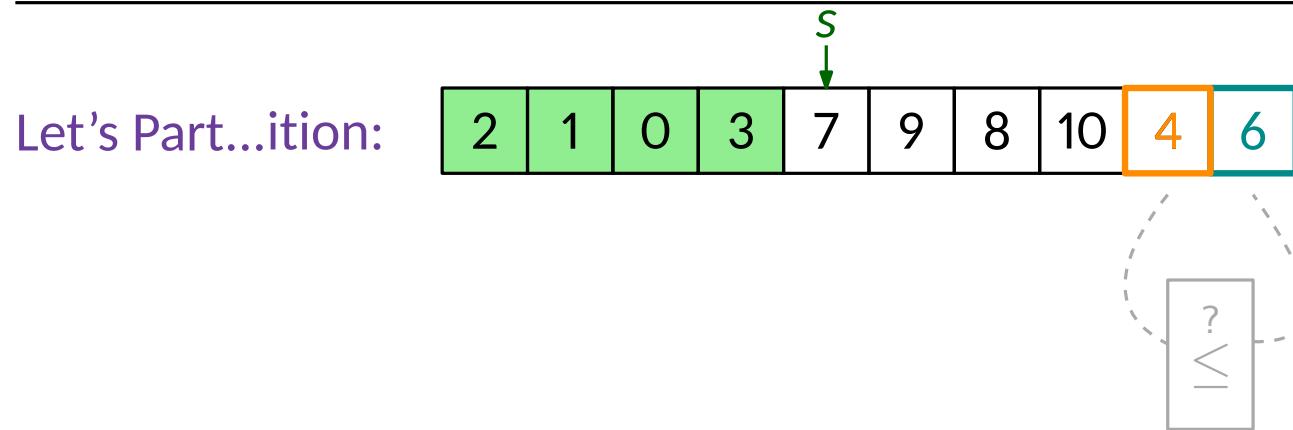
# Illustration: Quicksort

---



# Illustration: Quicksort

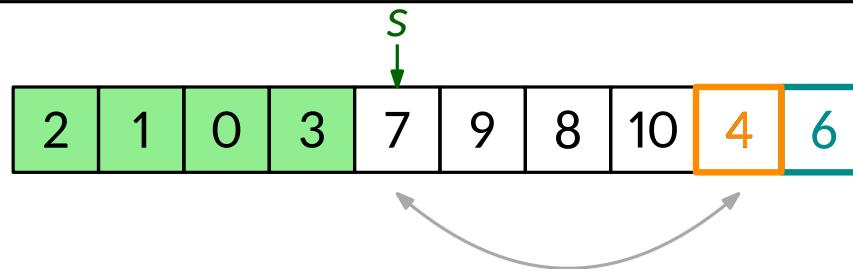
---



# Illustration: Quicksort

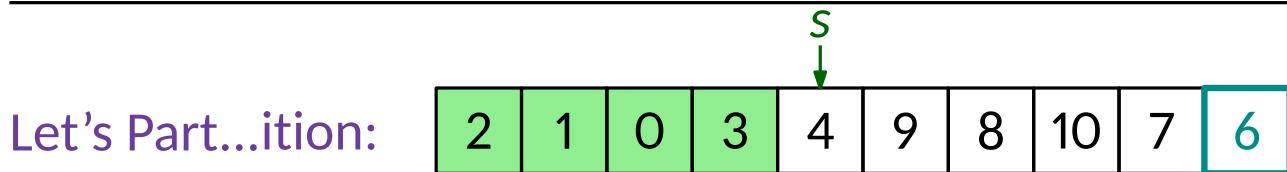
---

Let's Partition:



# Illustration: Quicksort

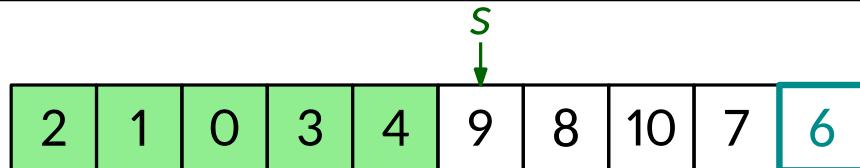
---



# Illustration: Quicksort

---

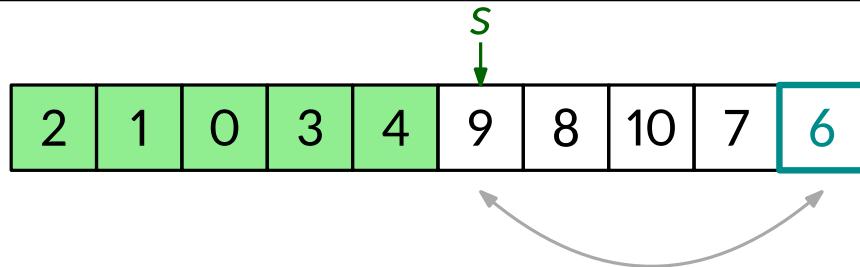
Let's Partition:



# Illustration: Quicksort

---

Let's Partition:



# Illustration: Quicksort

---

Let's Partition:

2	1	0	3	4	6	8	10	7	9
---	---	---	---	---	---	---	----	---	---

# Illustration: Quicksort

---

Let's Partition:



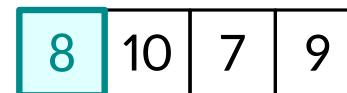
2	1	0	3	4
---	---	---	---	---

8	10	7	9
---	----	---	---

# Illustration: Quicksort

---

Let's Partition:



# Illustration: Quicksort

---

Let's Partition:



Ergebnis von Partition

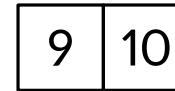
# Illustration: Quicksort

---

Let's Partition:



Ergebnis von Partition



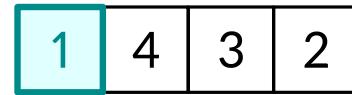
# Illustration: Quicksort

---

Let's Partition:



Ergebnis von Partition



# Illustration: Quicksort

---

Let's Partition:



Ergebnis von Partition



Ergebnis von Partition

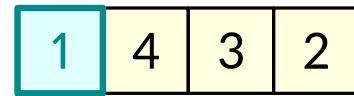
# Illustration: Quicksort

---

Let's Partition:



Ergebnis von Partition



Ergebnis von Partition



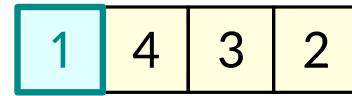
# Illustration: Quicksort

---

Let's Partition:



Ergebnis von Partition



Ergebnis von Partition



Ergebnis von Partition

# Illustration: Quicksort

Let's Partition:



Ergebnis von Partition



Ergebnis von Partition



Ergebnis von Partition



# Zwischenstand: Einfluss der Pivotwahl

---

Pivotwahl		Laufzeit
beliebiges festes Element (z.B. erstes)	deterministisch	Laufzeit $\Theta(n^2)$
<pre>pickPivot(A, ℓ, r)   return ℓ</pre>		
uniform zufälliges Element	randomisiert	erwartete Laufzeit $\Theta(n \log n)$
<pre>pickPivot(A, ℓ, r)   return RAND(ℓ, r)</pre>		

**Frage:** Was zeichnet eine gute Pivotwahl aus?

# Zwischenstand: Einfluss der Pivotwahl

---

## Pivotwahl

beliebiges festes Element (z.B. erstes)

```
pickPivot(A, ℓ, r)
| return ℓ
```

deterministisch

## Laufzeit

Laufzeit  $\Theta(n^2)$

uniform zufälliges Element

```
pickPivot(A, ℓ, r)
| return RAND(ℓ, r)
```

randomisiert

erwartete Laufzeit  $\Theta(n \log n)$

## Median

Intuitiv: Das Element, sodass die Hälfte der Elemente kleiner (und die andere Hälfte größer) ist

```
pickPivot(A, ℓ, r)
sort(A, ℓ, r)
return A[ $\lfloor \frac{\ell+r}{2} \rfloor$ ]
```

zu langsam!

?

deterministisch

Laufzeit  $\Theta(n \log n)$

# Median und das Auswahlproblem

**Definition** Für eine Folge  $(a_1, \dots, a_n)$  heißt  $a$  ein *Median* der Folge, wenn

$$|\{1 \leq i \leq n \mid a_i \leq a\}| \geq \lceil \frac{n}{2} \rceil, \text{ sowie}$$

$$|\{1 \leq i \leq n \mid a_i \geq a\}| \geq \lceil \frac{n}{2} \rceil$$

Das heißt: wenn  $n$  ungerade und  $a_1 \leq a_2 \leq \dots \leq a_n$ , dann ist  $a_{\frac{n+1}{2}}$  Median der Folge.

wenn  $n$  gerade und  $a_1 \leq a_2 \leq \dots \leq a_n$ , dann ist ein beliebiger Wert in  $[a_{\lfloor \frac{n+1}{2} \rfloor}, a_{\lceil \frac{n+1}{2} \rceil}]$  Median der Folge.

Achtung: Manchmal wird der Median stattdessen als  $a_{\lfloor \frac{n+1}{2} \rfloor}$  oder  $\frac{1}{2}(a_{\lfloor \frac{n+1}{2} \rfloor} + a_{\lceil \frac{n+1}{2} \rceil})$  oder  $a_{\lceil \frac{n+1}{2} \rceil}$  definiert.

## Medianproblem

**Gegeben:** Array  $A[1 \dots n]$

**Gesucht:** Median  $a$  von  $A[1], \dots, A[n]$

**Baseline:** Lösbar in Zeit  $O(n \log n)$  via Sortieren  
→ Können wir es in Zeit  $O(n)$  lösen?

Es wird uns **helfen**, sogar ein allgemeineres Problem schnell zu lösen:

## Rangselektion (Auswahlproblem)

**Gegeben:** Array  $A[1 \dots n]$ , Rang  $1 \leq k \leq n$

**Gesucht:** das  $k$ -kleinste Element von  $A[1], \dots, A[n]$

d.h.  $A'[k]$ , wobei  $A'[1] \leq \dots \leq A'[n]$   
die sortierte Reihenfolge von  $A[1], \dots, A[n]$  ist  
sehr schnell in der Praxis

Wir werden sehen: Wir können das Auswahlproblem in Zeit  $O(n)$  lösen, sowohl randomisiert als auch **deterministisch**  
→ wir können viele statistisch wichtige Größen schneller als durch Sortieren bestimmen: Mediane, Quartile, etc. (Quantile)

# Divide & Conquer für das Auswahlproblem

---

```
QuickSort(A[1 ... n], ℓ, r)
    assert( $1 \leq \ell \leq r \leq n$ )
    if ( $\ell < r$ ) then
        q = Partition(A, ℓ, r)

        QuickSort(A, ℓ, q - 1)
        QuickSort(A, q + 1, r)
```

**Frage:**

Wie sollten wir Quicksort für das Auswahlproblem anpassen?

# Illustration: Quickselect

---

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):

9	7	2	8	1	6	3	10	4	0
---	---	---	---	---	---	---	----	---	---

# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

$$L = 6$$

$$\Rightarrow k < L$$

quickselect( $A, 1, 5; 2$ ):



# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

$$L = 6$$

$$\Rightarrow k < L$$

quickselect( $A, 1, 5; 2$ ):



Ergebnis von Partition

# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

$$L = 6$$

$$\Rightarrow k < L$$

quickselect( $A, 1, 5; 2$ ):

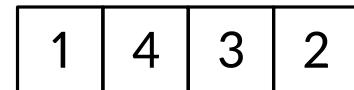


Ergebnis von Partition

$$L = 1$$

$$\Rightarrow k > L$$

quickselect( $A, 2, 5; 1$ ):



# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

$$L = 6$$

$$\Rightarrow k < L$$

quickselect( $A, 1, 5; 2$ ):



Ergebnis von Partition

$$L = 1$$

$$\Rightarrow k > L$$

quickselect( $A, 2, 5; 1$ ):



# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

$$L = 6$$

$$\Rightarrow k < L$$

quickselect( $A, 1, 5; 2$ ):



Ergebnis von Partition

$$L = 1$$

$$\Rightarrow k > L$$

quickselect( $A, 2, 5; 1$ ):



Ergebnis von Partition

# Illustration: Quickselect

---

quickselect( $A, 1, n; 2$ ):



Ergebnis von Partition

$$L = 6$$

$$\Rightarrow k < L$$

quickselect( $A, 1, 5; 2$ ):



Ergebnis von Partition

$$L = 1$$

$$\Rightarrow k > L$$

quickselect( $A, 2, 5; 1$ ):

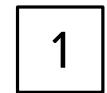


Ergebnis von Partition

$$L = 1$$

$$\Rightarrow k = L$$

return



# Divide & Conquer für das Auswahlproblem

```
QuickSort(A[1 ... n], ℓ, r)
```

```
    assert( $1 \leq \ell \leq r \leq n$ )
```

```
    if ( $\ell < r$ ) then
```

```
        q = Partition(A, ℓ, r)
```

```
        QuickSort(A, ℓ, q - 1)
```

```
        QuickSort(A, q + 1, r)
```



```
QuickSelect(A[1 ... n], ℓ, r; k)
```

```
    assert( $1 \leq \ell \leq r \leq n$  and  $1 \leq k \leq r - \ell + 1$ )
```

```
    if ( $\ell < r$ ) then
```

```
        q = Partition(A, ℓ, r)
```

//jetzt gilt:  $A[i] \leq A[q]$  für alle  $\ell \leq i < q$   
 $A[j] \geq A[q]$  für alle  $q < j \leq r$

```
        L = q - ℓ + 1 //Anzahl an Elementen in A[ℓ ... q]
```

```
        if ( $k < L$ )
```

```
            | return QuickSelect(A, ℓ, q - 1; k)
```

```
        else if ( $k > L$ )
```

```
            | return QuickSelect(A, q + 1, r; k - L)
```

```
        else
```

```
            | return A[q]
```

```
    else // $\ell = r$ 
```

```
        | return A[ℓ]
```

```
pickPivot(A, ℓ, r)  
| return RAND(ℓ, r)
```

Wichtiger Unterschied:

Quickselect macht  $\leq 1$  rekursiven Aufruf!

Theorem

Quickselect mit uniform zufälligen Pivotelement ist ein Las-Vegas-Algorithmus für Rangselektion mit erwarteter Laufzeit  $O(n)$ .

Wir beweisen diese Aussage nicht. Sie lässt sich mit ähnlichen Argumenten wie für Quicksort beweisen

# Divide & Conquer für das Auswahlproblem II

Können wir das auch **deterministisch** (ohne Zufall)?

- Wir wollen also wieder eine gute Pivotwahl,  
ideal: Median
- Henne-Ei-Problem?

## Beobachtung:

Wir benötigen nicht unbedingt den Median

Es reicht schon, wenn beide Subprobleme  $\text{Größe} \leq \alpha n$  haben,  
für eine Konstante  $\alpha < 1$ .

- Dazu reicht uns ein **approximativer Median**

```
pickPivot(A, ℓ, r)
| return RAND(ℓ, r) ?
```

```
QuickSelect(A[1 ... n], ℓ, r; k)
```

```
assert(1 ≤ ℓ ≤ r ≤ n and 1 ≤ k ≤ r − ℓ + 1)
if (ℓ < r) then
    q = Partition(A, ℓ, r)
    L = q − ℓ + 1 //Anzahl an Elementen in A[ℓ ... q]
    if (k < L)
        | return QuickSelect(A, ℓ, q − 1; k)
    else if (k > L)
        | return QuickSelect(A, q + 1, r; k − L)
    else
        | return A[q]
else //ℓ = r
    | return A[ℓ]
```

# Median der Mediane: Idee

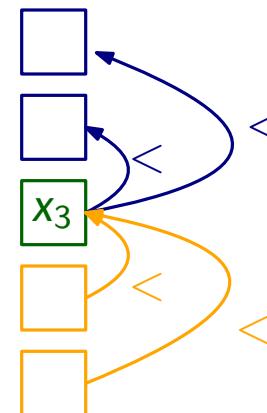
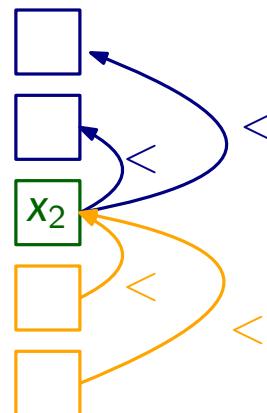
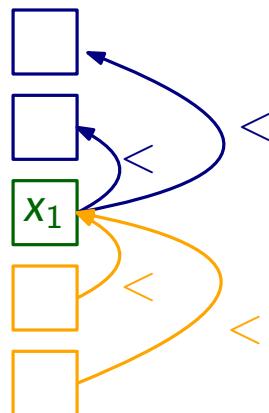
Hinweis: Wir nehmen vereinfachend an, dass alle Elemente unterschiedlich sind.

A[1]



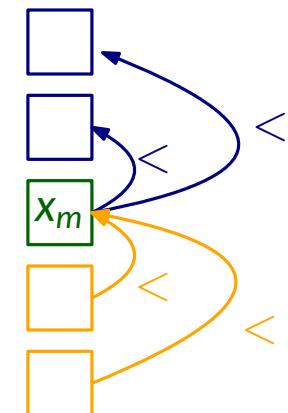
• • •

A[n]



• • •

$$m = \frac{n}{5}$$



Teilen die Elemente in 5er-Gruppen auf.  
Bestimmen den Median  $x_i$  der  $i$ -ten 5er-Gruppe.  
Bestimmen  $\tilde{x}$  als Median von  $x_1, \dots, x_m$

## Definition

Das so gewählte Pivotelement  $\tilde{x}$  nennen wir **Median der Mediane**.

## Eigenschaft:

Der Median der Mediane ist:

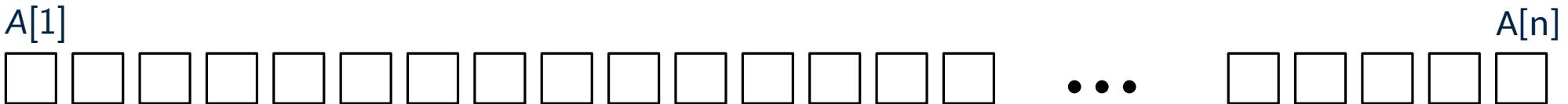
- größer als mindestens 30% der Elemente ( $\pm O(1)$ )
- kleiner als mindestens 30% der Elemente ( $\pm O(1)$ )

## Hinweis:

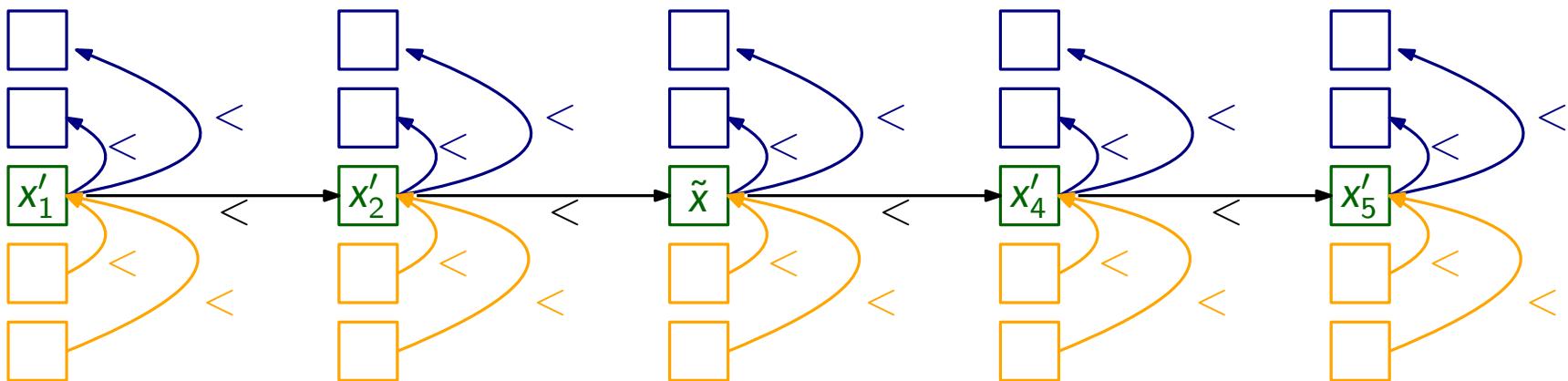
Wir müssen nicht unbedingt  $x_1, \dots, x_m$  sortieren um  $\tilde{x}$  zu bestimmen!

# Median der Mediane: Idee

Hinweis: Wir nehmen vereinfachend an, dass alle Elemente unterschiedlich sind.



Nur für die Analyse (!) schauen wir uns die Gruppen in der sortierten Reihenfolge  $x'_1, \dots, x'_m$  der Mediane an



Teilen die Elemente in 5er-Gruppen auf.  
Bestimmen den Median  $x_i$  der  $i$ -ten 5er-Gruppe.  
Bestimmen  $\tilde{x}$  als Median von  $x_1, \dots, x_m$

**Definition**  
Das so gewählte Pivotelement  $\tilde{x}$  nennen wir **Median der Mediane**.

## Eigenschaft:

Der Median der Mediane ist:

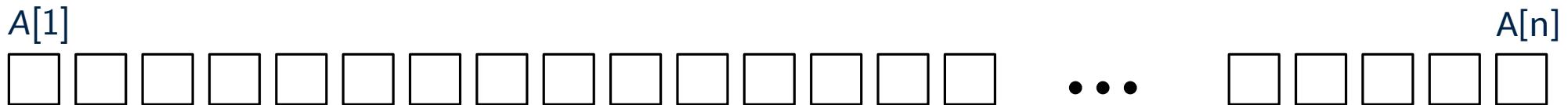
- größer als mindestens 30% der Elemente ( $\pm O(1)$ )
- kleiner als mindestens 30% der Elemente ( $\pm O(1)$ )

## Hinweis:

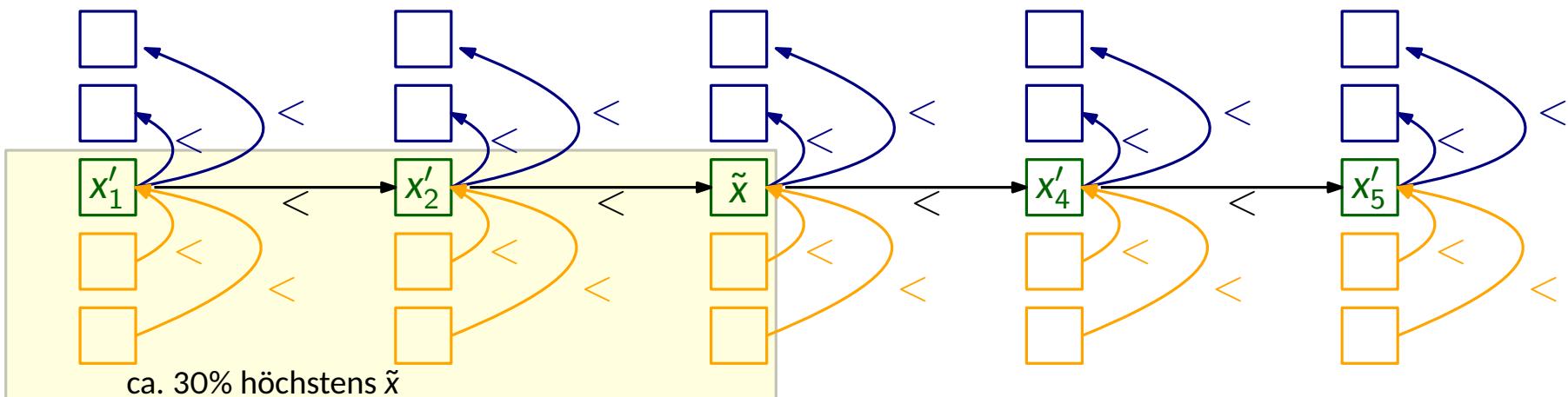
Wir müssen nicht unbedingt  $x_1, \dots, x_m$  sortieren um  $\tilde{x}$  zu bestimmen!

# Median der Mediane: Idee

Hinweis: Wir nehmen vereinfachend an, dass alle Elemente unterschiedlich sind.



Nur für die Analyse (!) schauen wir uns die Gruppen in der sortierten Reihenfolge  $x'_1, \dots, x'_m$  der Mediane an



Teilen die Elemente in 5er-Gruppen auf.  
Bestimmen den Median  $x_i$  der  $i$ -ten 5er-Gruppe  
Bestimmen  $\tilde{x}$  als Median von  $x_1, \dots, x_m$

## Definition

Das so gewählte Pivotelement  $\tilde{x}$  nennen wir **Median der Mediane**.

## Eigenschaft:

Der Median der Mediane ist:

- größer als mindestens 30% der Elemente ( $\pm O(1)$ )
  - kleiner als mindestens 30% der Elemente ( $\pm O(1)$ )

## Hinweis:

Wir müssen nicht unbedingt  $x_1, \dots, x_m$  sortieren um  $\tilde{x}$  zu bestimmen!

# Median der Mediane: Idee

Hinweis: Wir nehmen vereinfachend an, dass alle Elemente unterschiedlich sind.

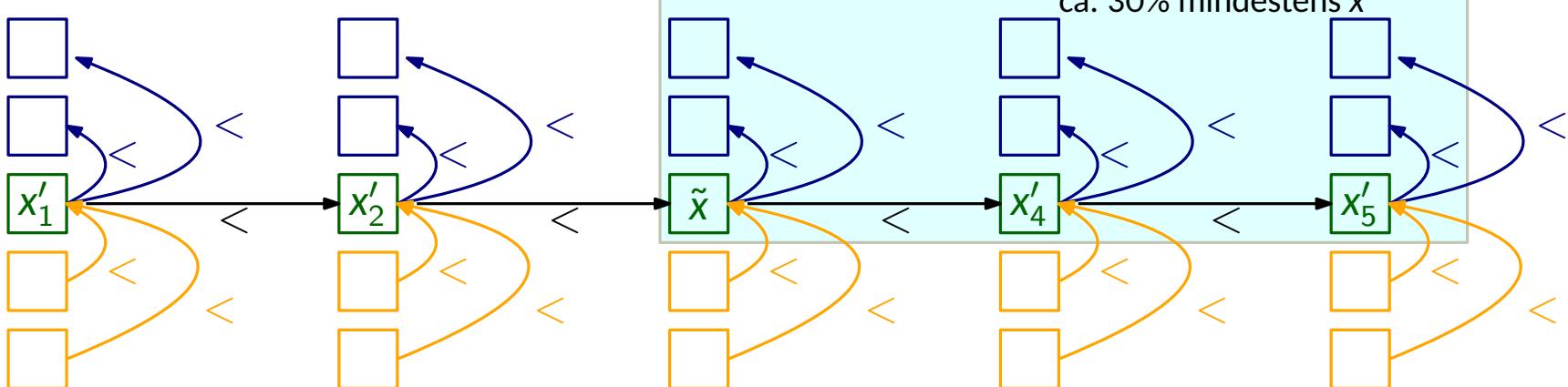
A[1]



A[n]



Nur für die Analyse (!) schauen wir uns die Gruppen in der sortierten Reihenfolge  $x'_1, \dots, x'_m$  der Mediane an



Teilen die Elemente in 5er-Gruppen auf.  
Bestimmen den Median  $x_i$  der  $i$ -ten 5er-Gruppe.  
Bestimmen  $\tilde{x}$  als Median von  $x_1, \dots, x_m$

## Definition

Das so gewählte Pivotelement  $\tilde{x}$  nennen wir **Median der Mediane**.

## Eigenschaft:

Der Median der Mediane ist:

- größer als mindestens 30% der Elemente ( $\pm O(1)$ )
- kleiner als mindestens 30% der Elemente ( $\pm O(1)$ )

## Hinweis:

Wir müssen nicht unbedingt  $x_1, \dots, x_m$  sortieren um  $\tilde{x}$  zu bestimmen!

# Deterministische Rangselektion: Pseudocode

```
QuickSelect( $A[1 \dots n]$ ,  $\ell, r; k$ )
```

```
assert( $1 \leq \ell \leq r \leq n$  and  $1 \leq k \leq r - \ell + 1$ )
if ( $\ell < r$ ) then
     $q = \text{Partition}(A, \ell, r)$ 
     $L = q - \ell + 1$  //Anzahl an Elementen in  $A[\ell \dots q]$ 
    if ( $k < L$ )
        | return QuickSelect( $A, \ell, q - 1; k$ )
    else if ( $k > L$ )
        | return QuickSelect( $A, q + 1, r; k - L$ )
    else
        | return  $A[q]$ 
else // $\ell = r$ 
    | return  $A[\ell]$ 
```

```
pickPivot( $A, \ell, r$ ) //bestimmt den Median der Mediane
```

```
 $n = r - \ell + 1$ 
 $m = \lfloor n/5 \rfloor$ 
//Initialisiere  $M[1 \dots m]$ 
for  $i = 1, \dots, m$  do
     $M[i] = \text{MedianOfFive}(A[\ell + 5i - 4 \dots \ell + 5i])$ 
return QuickSelect( $M, 1, m; \lfloor \frac{m}{2} \rfloor$ )
```

MedianOfFive( $A[1 \dots 5]$ )

- bestimmt den Median von  $A[1 \dots 5]$   
(auf beliebige Art und Weise)

Achtung!

Jeder Aufruf von Quickselect hat bis zu **zwei** rekursive Aufrufe:

- zur Bestimmung des Pivotelements
- Ergebnisbestimmung in kleinerem Subproblem

# Deterministische Rangselektion: Laufzeit

```
QuickSelect(A[1 ... n], ℓ, r; k)
```

```
assert( $1 \leq \ell \leq r \leq n$  and  $1 \leq k \leq r - \ell + 1$ )
if ( $\ell < r$ ) then
    q = Partition(A, ℓ, r)
    L = q - ℓ + 1 //Anzahl an Elementen in A[ℓ ... q]
    if (k < L)
        | return QuickSelect(A, ℓ, q - 1; k)
    else if (k > L)
        | return QuickSelect(A, q + 1, r; k - L)
    else
        | return A[q]
else // $\ell = r$ 
    | return A[ℓ]
```

```
pickPivot(A, ℓ, r) //bestimmt den Median der Mediane
```

```
n = r - ℓ + 1
m = ⌊n/5⌋
//Initialisiere M[1 ... m]
for i = 1, ..., m do
    M[i] = MedianOfFive(A[ℓ + 5i - 4 ... ℓ + 5i])
return QuickSelect(M, 1, m; ⌊m/2⌋)
```

MedianOfFive(A[1 ... 5])

- bestimmt den Median von A[1 ... 5]  
(auf beliebige Art und Weise)

**Behauptung:** Es gibt Konstanten  $C, D, n_0$  sodass:

$$T(n) \leq \begin{cases} C & \text{für alle } n < n_0 \\ T(\lfloor \frac{1}{5}n \rfloor) + T(\lfloor \frac{7}{10}n \rfloor + 2) + Dn & \text{für alle } n \geq n_0 \end{cases}$$

**Schlussfolgerung:**  $T(n) \in O(n)$ . ↗ Tafelpräsentation

## Theorem

Quickselect mit Median der Mediane als Pivotwahl löst das Rangselektionsproblem in Zeit  $O(n)$ .

# Einfluss der Pivotwahl: Quickselect

---

## Quickselect

### Pivotwahl

beliebiges festes Element (z.B. erstes)

```
pickPivot(A, ℓ, r)
    return ℓ
```

### Laufzeit

Laufzeit  $O(n^2)$

uniform zufälliges Element

randomisiert

erwartete Laufzeit  $O(n)$

Median der Mediane

deterministisch

Laufzeit  $O(n)$

```
pickPivot(A, ℓ, r)
    n = r - ℓ + 1
    m = ⌊n/5⌋
    //Initialisiere M[1 ... m]
    for i = 1, ..., m do
        M[i] = MedianOfFive(A[ℓ + 5i - 4 ... ℓ + 5i])
    return QuickSelect(M, 1, m; ⌊m/2⌋)
```

# Einfluss der Pivotwahl: Quicksort

## Quicksort

Pivotwahl	Laufzeit
beliebiges festes Element (z.B. erstes)	deterministisch <b>Laufzeit <math>O(n^2)</math></b>
<pre>pickPivot(A, ℓ, r)   return ℓ</pre>	
uniform zufälliges Element	randomisiert <b>erwartete Laufzeit <math>O(n \log n)</math></b>
<pre>pickPivot(A, ℓ, r)   return RAND(ℓ, r)</pre>	
Median berechnet via Quickselect mit Median der Mediane	deterministisch <b>Laufzeit <math>O(n \log n)</math></b>
<pre>pickPivot(A, ℓ, r)   return quickselect(A, ℓ, r; ⌊(r-ℓ+1)/2⌋)</pre>	
Median vom ersten, mittleren und letzten Element	deterministisch <b>Laufzeit <math>O(n^2)</math></b>
<pre>pickPivot(A, ℓ, r)   return Median von A[ℓ], A[⌊(ℓ+r)/2⌋], A[r]</pre>	
Median von $k$ zufälligen Elementen	randomisiert <b>erwartete Laufzeit <math>O(n \log n)</math></b>
...	

# Praktische Implementierungen

```
QuickSort( $A[1 \dots n]$ ,  $\ell, r$ )
```

```
  if ( $r - \ell + 1 > n_0$ ) then
     $k = \text{pickPivot}(A, \ell, r)$  // Wahl des Pivotelements
     $p = A[k]$ 
     $\text{swap}(A[k], A[r])$       // ans Ende setzen
```

```
   $i = \ell, j = r$ 
```

```
  repeat
```

```
    while ( $A[i] < p$ ) do  $i++$ 
    while ( $A[j] > p$ ) do  $j--$ 
    if ( $i \leq j$ ) then
       $\text{swap}(A[i], A[j])$ 
       $i++, j--$ 
```

```
  until  $i > j$ 
```

```
  QuickSort( $A, \ell, j$ )
```

```
  QuickSort( $A, i, r$ )
```

```
else
```

```
  | Benutze schnelles Sortierverfahren für Eingaben  $\leq n_0$ 
```

Die schnellsten Implementierungen benutzen bestimmte Optimierungen:

- anderes Partitionierungsverfahren (nach Hoare)
- benutzen anderes Sortierverfahren für kleine Eingaben
- achten auf Endrekursion (wird iterativ statt rekursiv verarbeitet)  
hier nicht gezeigt!
- ...

# Zusammenfassung

---

Wir haben gesehen:

- Quicksort, ein in der Praxis häufig verwendetes Sortierverfahren
- aber: quicksort benötigt  $\Omega(n^2)$ -Zeit im worst-case!
- es stellt sich allerdings heraus, dass:
  - es eine randomisierte Variante mit geringerer erwarteter Laufzeit im worst-case hat
  - es eine schnelle deterministische Variante hat
    - nicht randomisiert!
- um die schnelle deterministische Variante zu erhalten, besprachen wir "en route":
  - **Rangselektion** (Auswahlproblem) in Zeit  $O(n)$   
→ Bestimmung von Median, Quartilen, etc.

Analyse: Rekurrenz der Form

$$T(n) \leq \begin{cases} C & \text{für alle } n < n_0 \\ \left( \sum_{i=1}^{\ell} T(\lceil \alpha_i n \rceil + e_i) \right) + Dn & \text{für alle } n \geq n_0 \end{cases}$$

mit  $\sum_{i=1}^{\ell} \alpha_i < 1$  erfüllt  $T(n) \in O(n)$

Algorithmen und Datenstrukturen SS'23

# Kapitel 7: Elementare Datenstrukturen

Marvin Künemann

AG Algorithmen & Komplexität

# Quiz: Rekursionsungleichungen

---

Was ist die **schärfste** Laufzeitschranke an  $T(n)$  für folgende Rekursionsgleichungen?

Für alle  $n \geq n_0$  gelte:

1.  $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + Dn$

$O(n)$ ?       $O(n \log n)$ ?       $O(n^2)$ ?

2.  $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + Dn$

$O(n)$ ?       $O(n \log n)$ ?       $O(n^2)$ ?

3.  $T(n) = T(n - 2) + T(1) + Dn$

$O(n)$ ?       $O(n \log n)$ ?       $O(n^2)$ ?

4.  $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{3}\right) + Dn$

$O(n)$ ?       $O(n \log n)$ ?       $O(n^2)$ ?

# Kapitelüberblick

---

Letzte Kapitel: viele verschiedene Algorithmen für ein bestimmtes Problem (Sortieren)

Jetzt: grundlegende **Datenstrukturen**

Insbesondere behandeln wir in diesem Kapitel:

- Arrays als Datenstruktur
- (Einfach/Doppelt) Verkettete Listen
- Dynamische Arrays
- Analysekonzept: **amortisierte Analyse**

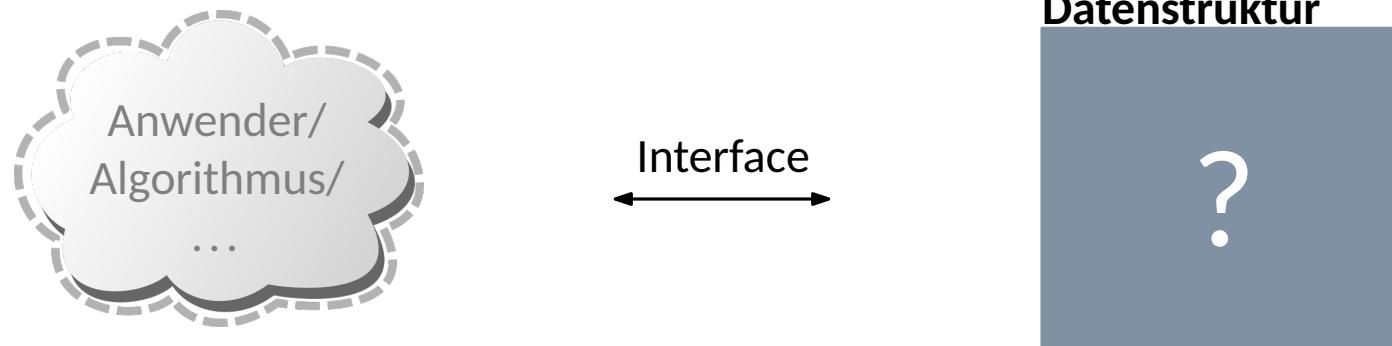
# (Abstrakte) Datenstrukturen

Fokus bisher: Lösen eines **algorithmischen Problems**



Neues Konzept: Entwickeln einer **Datenstruktur**

→ Organisation von Daten in einer Form, die bestimmte Zugriffe ermöglicht

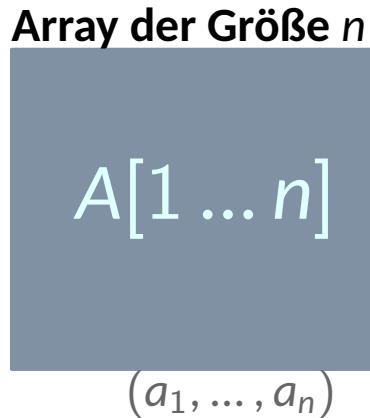


# Arrays

# Bereits bekanntes Beispiel: Arrays



Interface  
↔



Implementierung:

⋮	
$A[n]$	$c(k + n)$
⋮	
$A[2]$	$c(k + 2)$
$A[1]$	$c(k + 1)$
⋮	

## Interface (erlaubte Operationen)

Auf ein Array A der Größe  $n$  können wir wie folgt zugreifen:

**Auswertung:** Gegeben  $1 \leq i \leq n$ , gib das  $i$ -te Element zurück

$A[i]$  → Der Aufruf  $A[i]$  gibt  $a_i$  zurück

$A.get(i)$  **Laufzeit:**  $O(1)$

Kann als Klasse beschrieben werden:

```
class Array {  
    ...  
    int get(int i);  
    int set(int i, int x);  
}
```

**Zuweisung:** Gegeben  $1 \leq i \leq n$  und  $x$ , setze das  $i$ -te Element auf  $x$

$A[i] = x$  → Der Aufruf  $A[i] = x$  ersetzt bisheriges  $a_i$  durch  $x$

$A.set(i,x)$  Aus  $(a_1, \dots, a_n)$  wird  $(a_1, \dots, a_{i-1}, x, a_{i+1}, \dots, a_n)$

**Laufzeit:**  $O(1)$

# Arrays: Initialisierung

Den Speicherbereich für ein Array zu reservieren (**Allokation**)  
ist in konstanter Zeit möglich

`type A[100];`

(Steht bereits zur Compilezeit fest)

`n = 12314;`

`type* A = new type[n];`

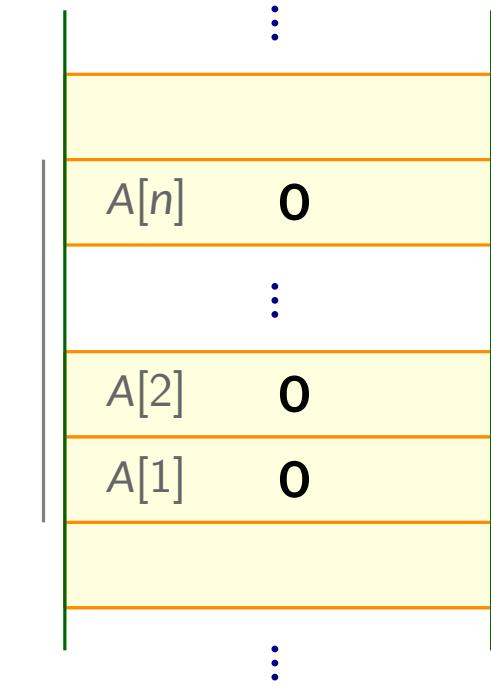
(während Ausführung allokiert)

**Achtung:** Ohne besondere Initialisierung können wir keine  
Annahmen über den Inhalt machen!

**Beispiel:** Wenn wir wollen, dass  $A[i] = 0$  für  $1 \leq i \leq n$ ,  
müssen wir ausführen:

`for i = 1, ..., n do`  
  | `A[i] = 0`

Laufzeit  $O(n)$



↗ Übungsaufgabe: Man kann **simulieren**, dass alle Elemente anfangs 0 sind  
→ "Arrays", die man in Zeit  $O(1)$  auf  $A[i] = 0$  für alle  $1 \leq i \leq n$  initialisieren kann

In Programmiersprachen ohne Garbage-Collection sollte nicht benutzter Speicher wieder freigegeben werden:

`delete[] A;`

# Verkettete Listen

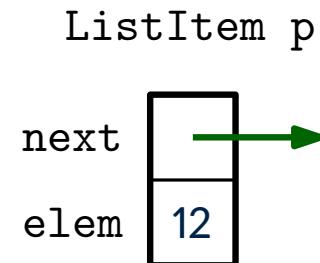
# Verkettete Listen

Verkettete Listen (Linked Lists) bilden eine Alternative zu Arrays, um eine Folge  $a_1, \dots, a_n$  zu speichern

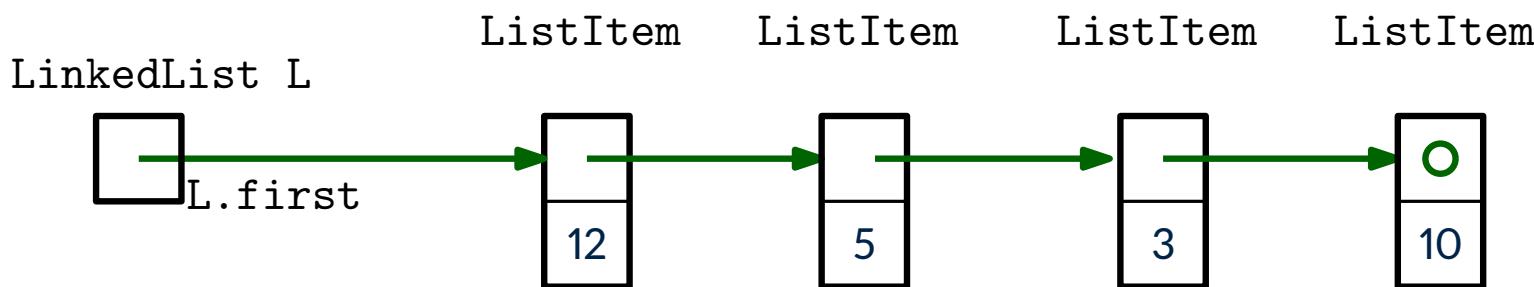
Idee: wir bilden eine Kette bestehend aus Zeigern (Pointer) auf das nachfolgende Element

Grundbaustein: Item, bestehend aus einem Element und einem Zeiger auf das nächste Item

```
struct ListItem {  
    ListItem* next; //Zeiger auf das nächste Item  
    type elem;  
};
```



Daraus bildet sich eine Liste an Elementen (12, 5, 3, 10) wie folgt:



```
class LinkedList {  
    ListItem* first;  
    ...  
};
```

Das Ende der Liste wird gekennzeichnet durch  
○ nullptr

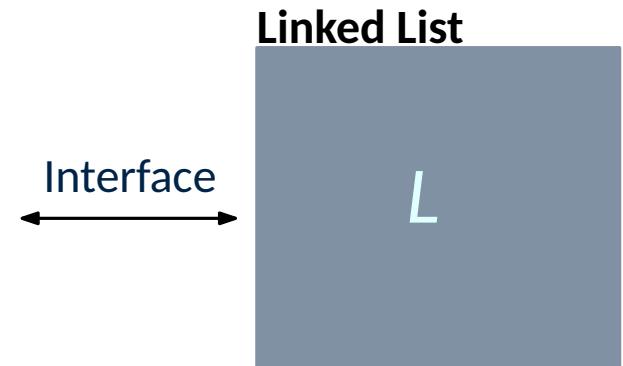
Frage: Lässt sich eine leere Liste als LinkedList repräsentieren?

# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

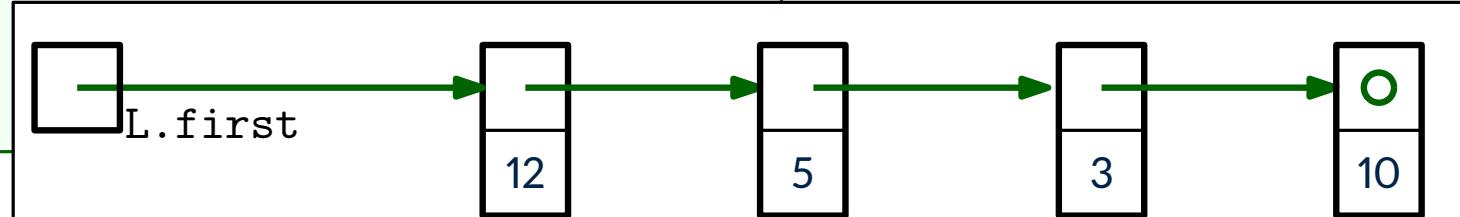
Wie effizient können wir sie ausführen?



`L.find(x)` Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;  
  
    if (p->elem != x)  
        p = nullptr;  
  
    return p;
```

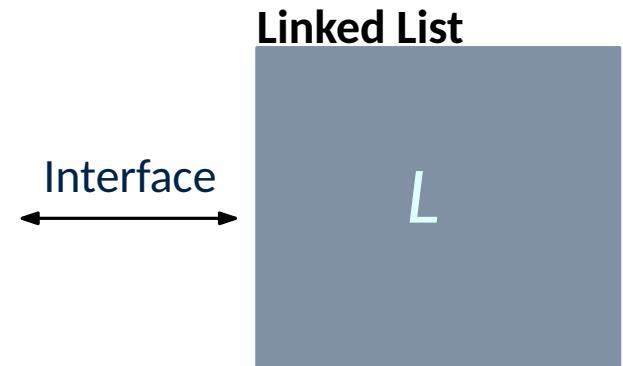


# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

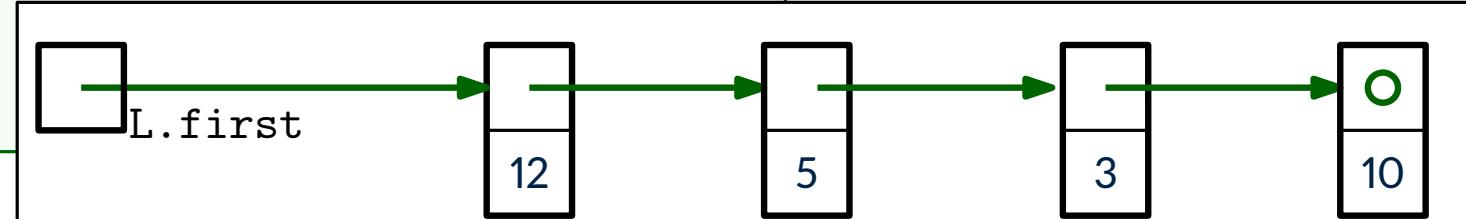
```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;
```

`find(4, L)`

```
        if (p->elem != x)  
            p = nullptr;
```

```
    return p;
```

```
}
```

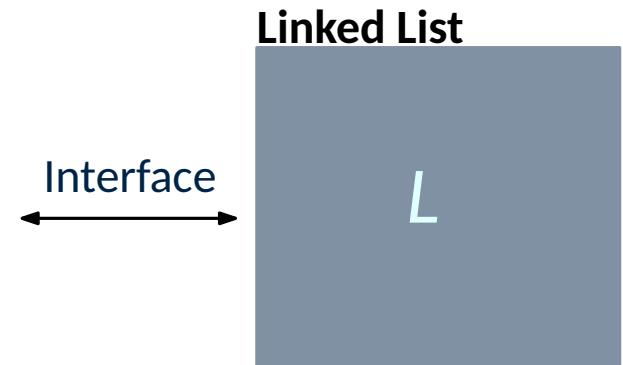


# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;
```

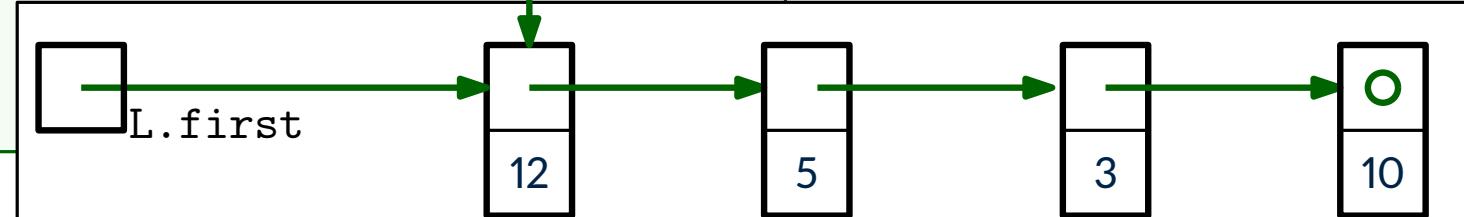
`find(4, L)`

```
        if (p->elem != x)  
            p = nullptr;
```

$p$

```
    return p;
```

}

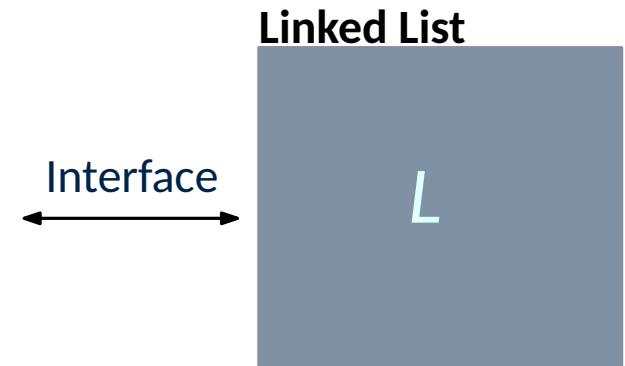


# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;  
  
    if (p->elem != x)  
        p = nullptr;  
  
    return p;
```

find(4, L)

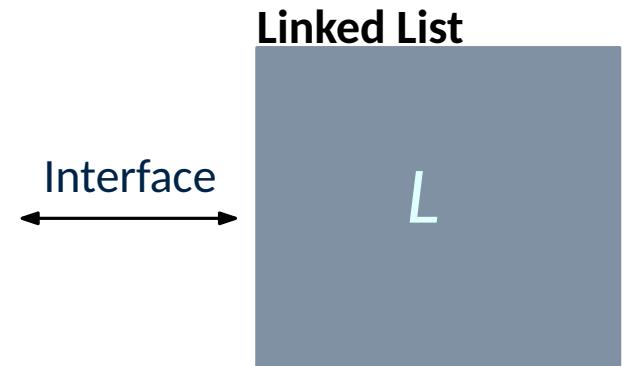
p

# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



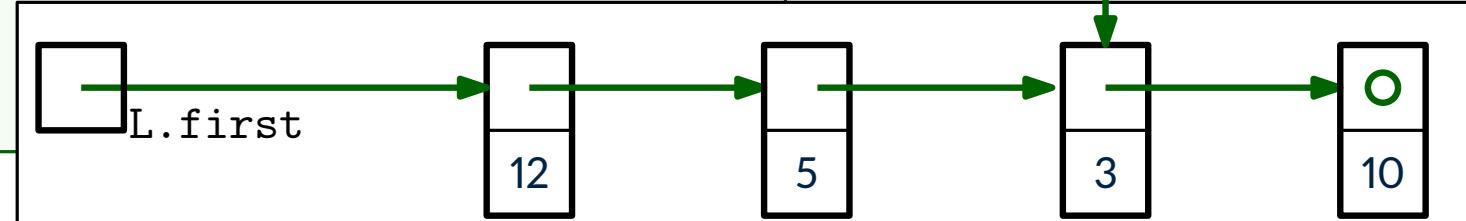
$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;  
  
    if (p->elem != x)  
        p = nullptr;  
  
    return p;
```

find(4, L)

p

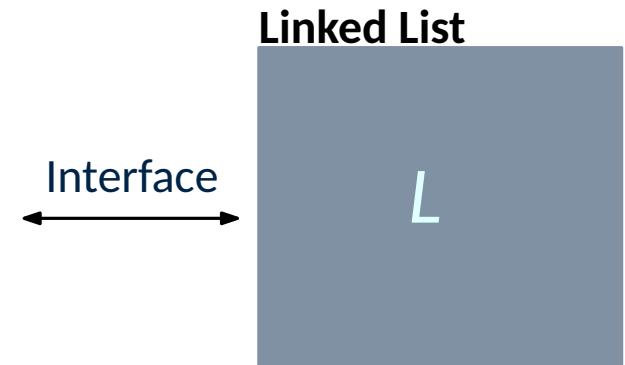


# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

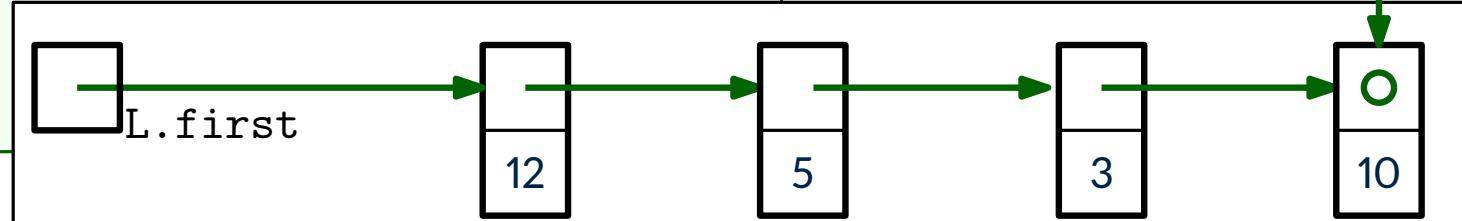


$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;  
  
    if (p->elem != x)  
        p = nullptr;  
  
    return p;
```

find(4, L)

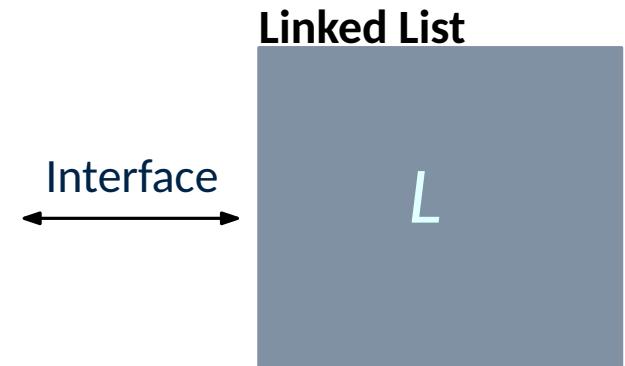


# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

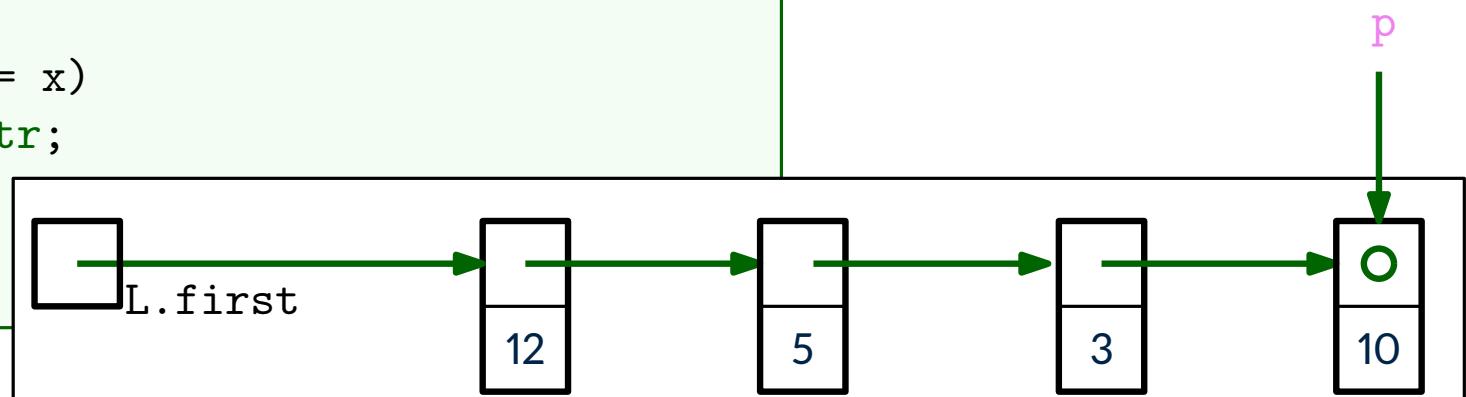


$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;  
  
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;  
  
    if (p->elem != x)  
        p = nullptr;  
  
    return p;
```

find(4, L) : return nullptr

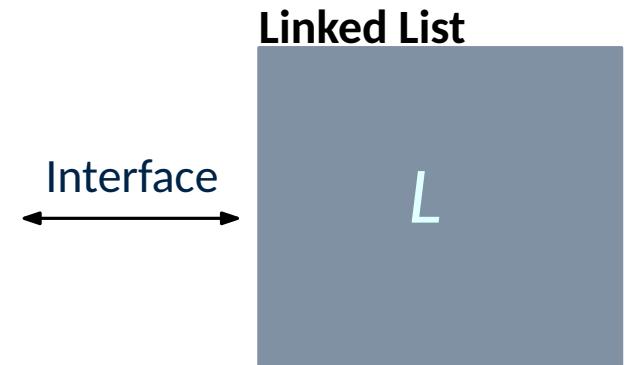


# Operationen auf Listen: Suchen (Find)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



$L.\text{find}(x)$  Gibt es ein  $i$  mit  $a_i = x$ ?

Wenn ja, gib einen Zeiger auf das entsprechende Item zurück  
Ansonsten, gib `nullptr` zurück

```
ListItem* find(int x, LinkedList L )  
    if (L.first == nullptr)  
        return nullptr;
```

Frage: Was ist die Laufzeit von  $L.\text{find}(x)$ ?

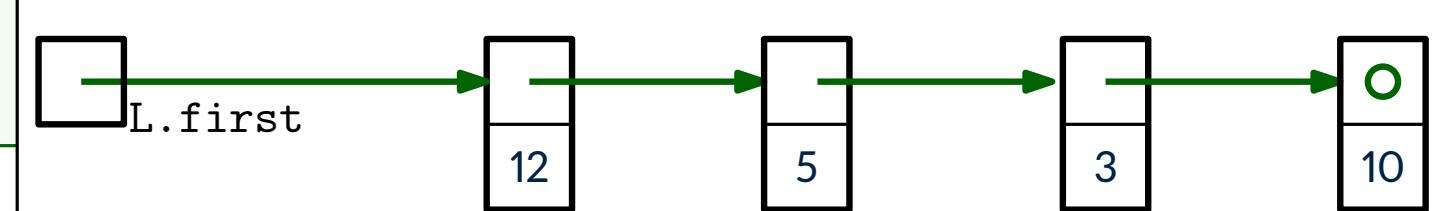
```
    ListItem* p = L.first;  
    while (p->elem != x and p->next != nullptr)  
        p = p->next;
```

$\text{find}(4, L) : \text{return } nullptr$

```
    if (p->elem != x)  
        p = nullptr;
```

```
    return p;
```

```
}
```

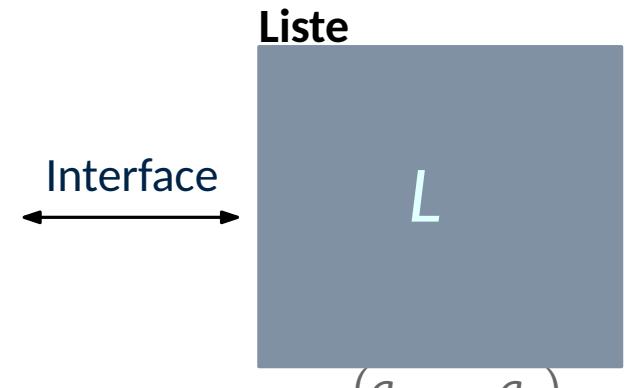


# Operationen auf Listen: Einfügen (Insert)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



`L.insert_after(p, x)`  $p$  ist Zeiger auf ein Item  $a_i$  der Liste

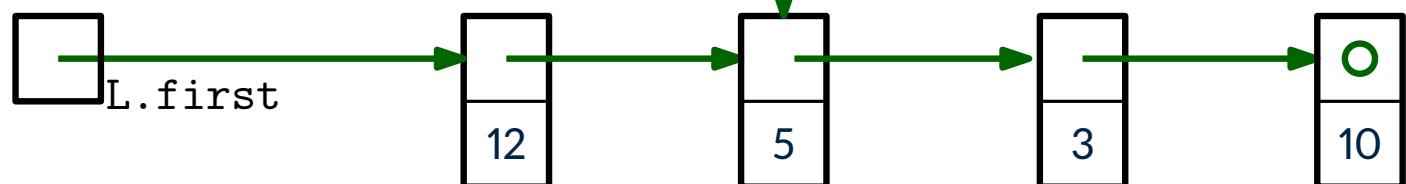
Füge  $x$  nach  $a_i$  ein

$\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, x, a_{i+1}, \dots, a_n)$

```
void insert_after(ListItem* p, int x) {  
    ListItem* q = new ListItem;  
    q->elem = x;  
    q->next = p->next;  
  
    p->next = q;  
}
```

`insert_after(p, 10)`

$p$

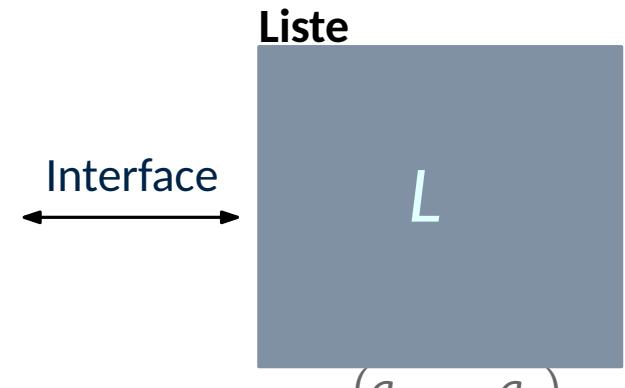


# Operationen auf Listen: Einfügen (Insert)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



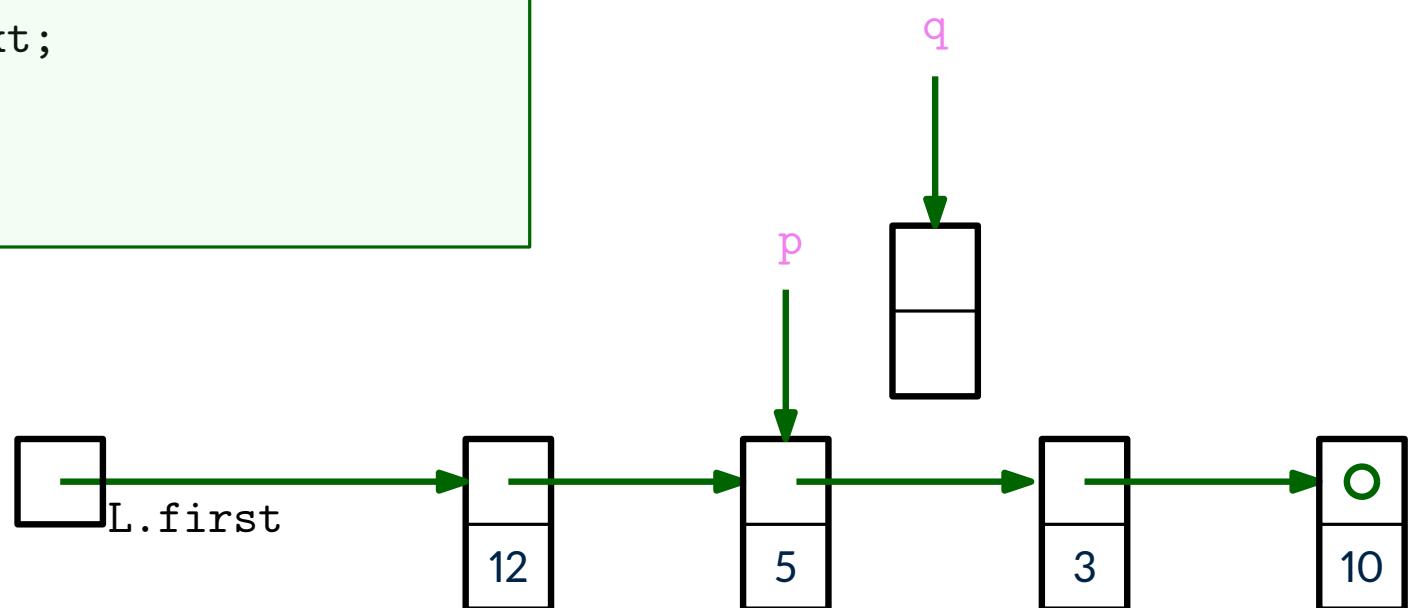
`L.insert_after(p, x)`  $p$  ist Zeiger auf ein Item  $a_i$  der Liste

Füge  $x$  nach  $a_i$  ein

$\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, x, a_{i+1}, \dots, a_n)$

```
void insert_after(ListItem* p, int x) {  
    ListItem* q = new ListItem;  
    q->elem = x;  
    q->next = p->next;  
  
    p->next = q;  
}
```

`insert_after(p, 10)`

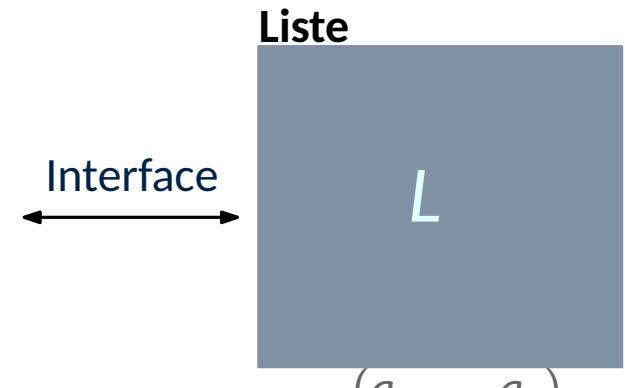


# Operationen auf Listen: Einfügen (Insert)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



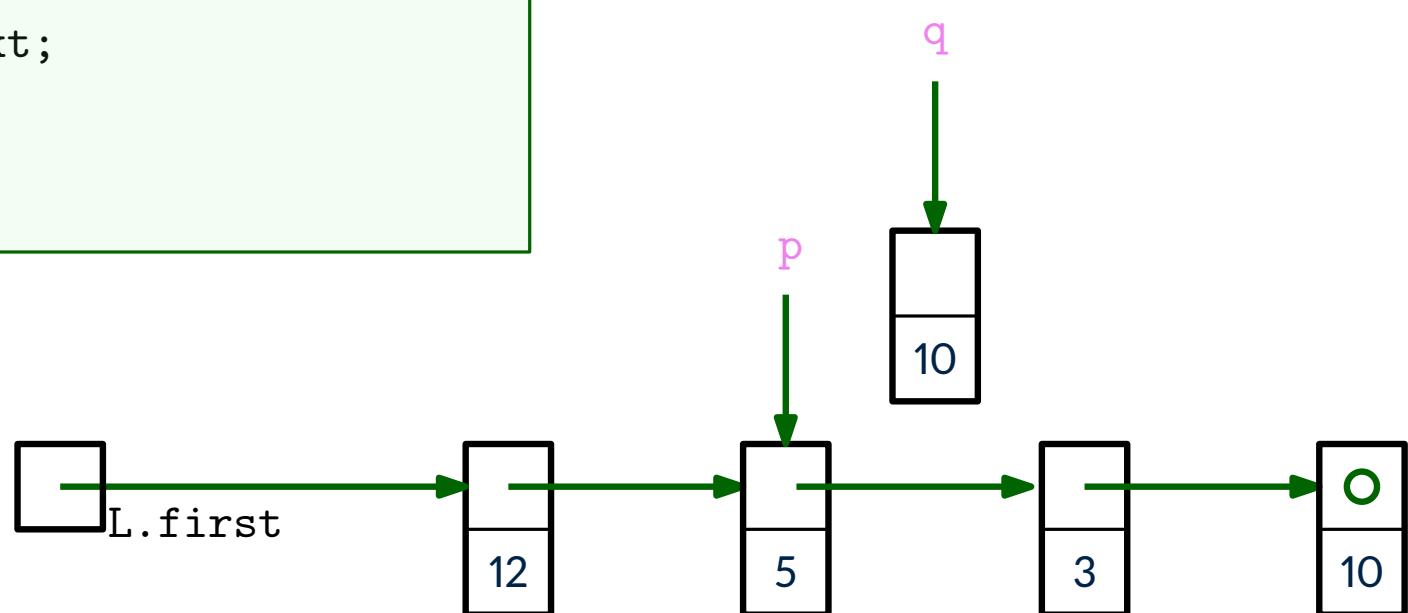
`L.insert_after(p, x)`  $p$  ist Zeiger auf ein Item  $a_i$  der Liste

Füge  $x$  nach  $a_i$  ein

$\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, x, a_{i+1}, \dots, a_n)$

```
void insert_after(ListItem* p, int x) {  
    ListItem* q = new ListItem;  
    q->elem = x;  
    q->next = p->next;  
  
    p->next = q;  
}
```

`insert_after(p, 10)`

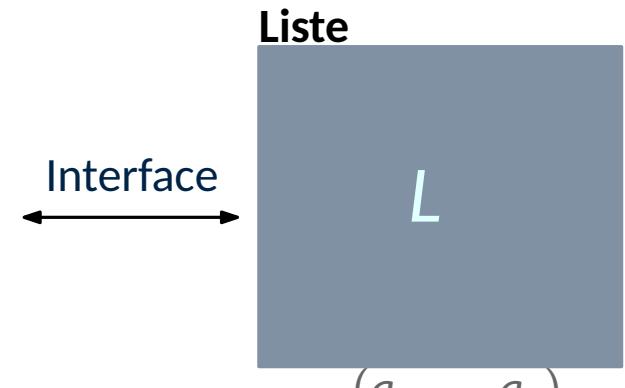


# Operationen auf Listen: Einfügen (Insert)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



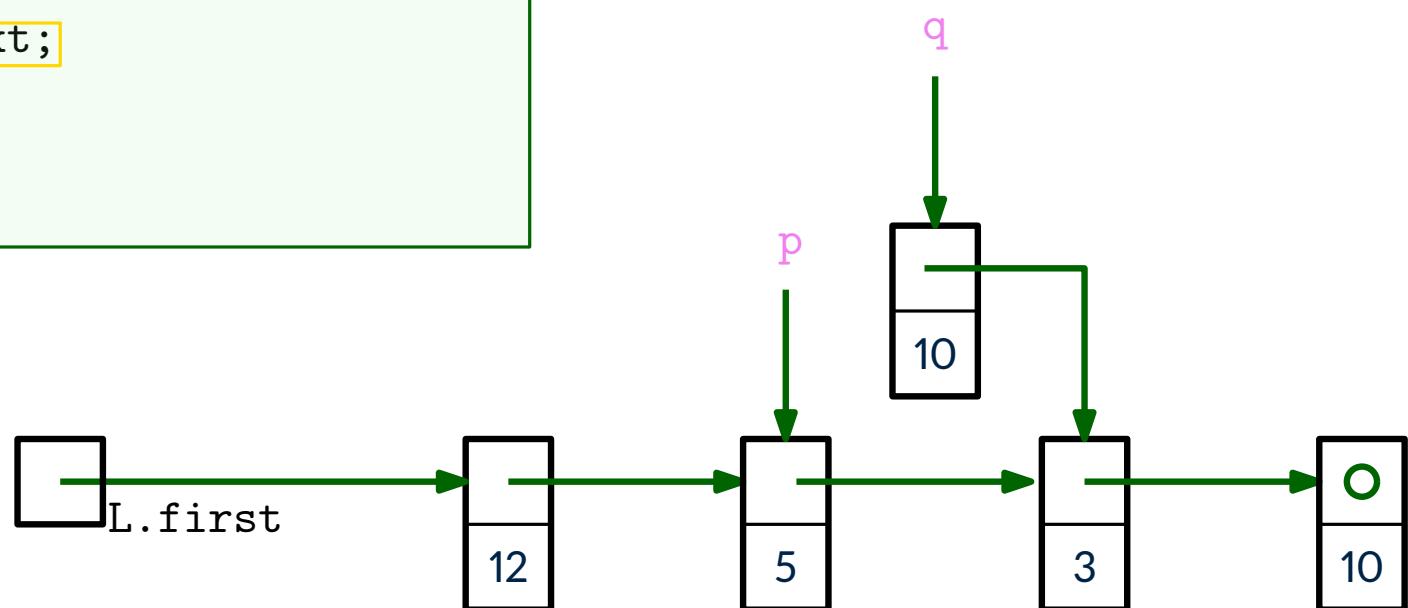
`L.insert_after(p, x)`  $p$  ist Zeiger auf ein Item  $a_i$  der Liste

Füge  $x$  nach  $a_i$  ein

$\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, x, a_{i+1}, \dots, a_n)$

```
void insert_after(ListItem* p, int x) {  
    ListItem* q = new ListItem;  
    q->elem = x;  
    q->next = p->next;  
  
    p->next = q;  
}
```

`insert_after(p, 10)`

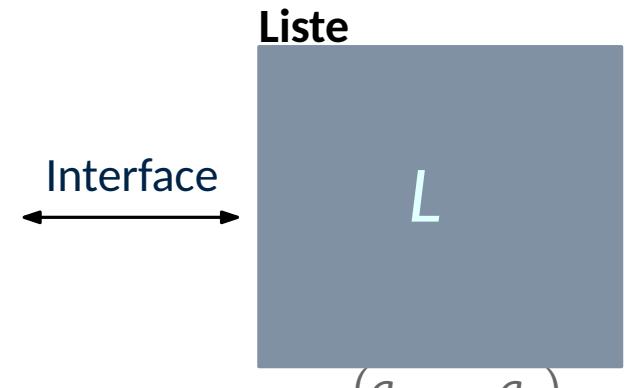


# Operationen auf Listen: Einfügen (Insert)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



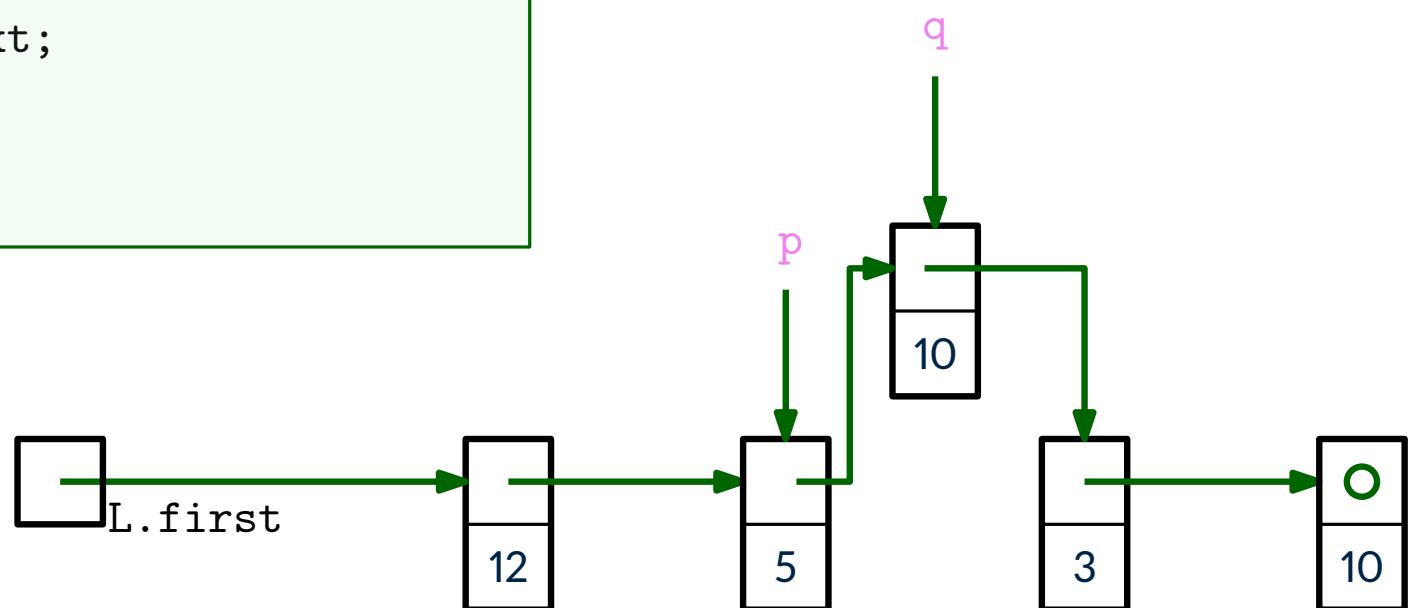
`L.insert_after(p, x)`  $p$  ist Zeiger auf ein Item  $a_i$  der Liste

Füge  $x$  nach  $a_i$  ein

$\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, x, a_{i+1}, \dots, a_n)$

```
void insert_after(ListItem* p, int x) {  
    ListItem* q = new ListItem;  
    q->elem = x;  
    q->next = p->next;  
  
    p->next = q;  
}
```

`insert_after(p, 10)`

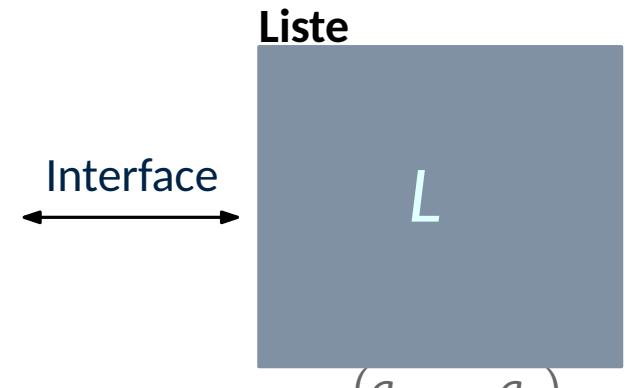


# Operationen auf Listen: Einfügen (Insert)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



`L.insert_after(p, x)`  $p$  ist Zeiger auf ein Item  $a_i$  der Liste

Füge  $x$  nach  $a_i$  ein

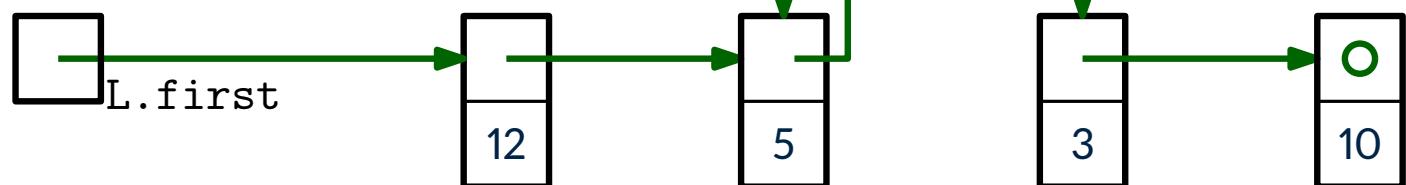
$\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, x, a_{i+1}, \dots, a_n)$

```
void insert_after(ListItem* p, int x) {  
    ListItem* q = new ListItem;  
    q->elem = x;  
    q->next = p->next;  
  
    p->next = q;  
}
```

`insert_after(p, 10)`

**Laufzeit:** `L.insert_after(p, x)` läuft in Zeit  $O(1)$ .

**Frage:** Können wir auch direkt vor  $p$  in Laufzeit  $O(1)$  einfügen?

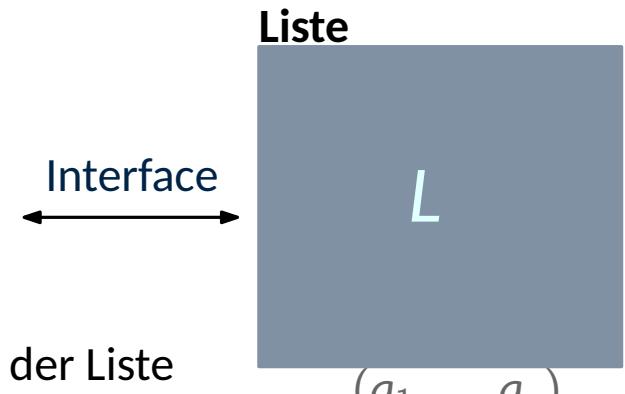


# Operationen auf Listen: Löschen (Delete)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

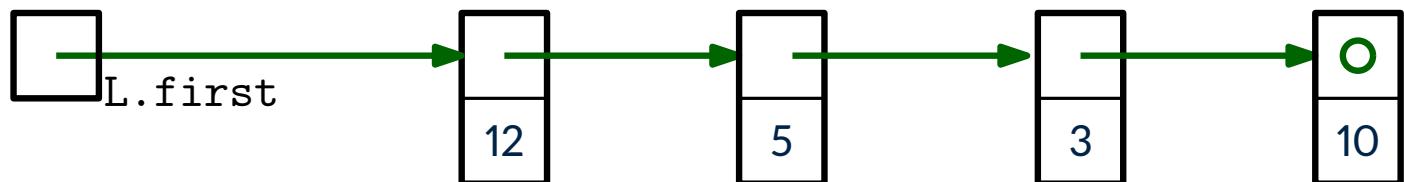
Wie effizient können wir sie ausführen?



`L.erase_after(p)`

$p$  ist Zeiger auf ein Element  $a_i$  (oder den Anfang) der Liste  
Lösche das Element  $a_{i+1}$  (bzw.  $a_1$ )  
 $\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, a_{i+2}, \dots, a_n)$

```
void erase_after(ListItem* p) {  
    ListItem* q = p->next;  
  
    p->next = q->next;  
    delete q;  
}
```

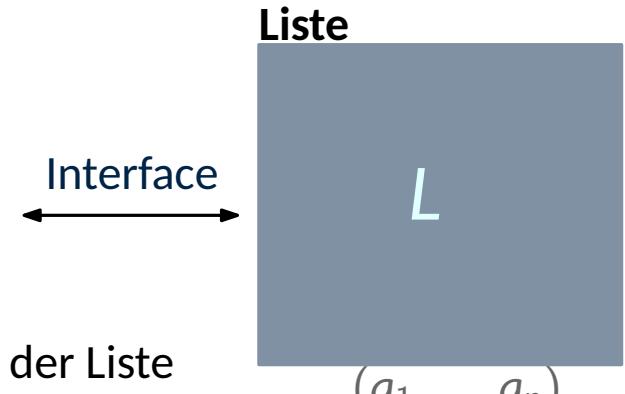


# Operationen auf Listen: Löschen (Delete)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

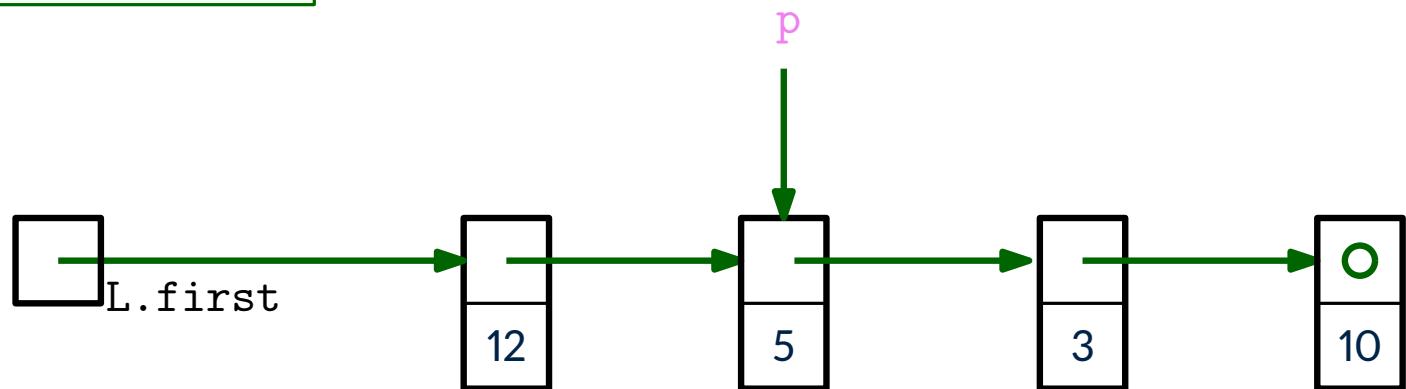


`L.erase_after(p)`

$p$  ist Zeiger auf ein Element  $a_i$  (oder den Anfang) der Liste  
Lösche das Element  $a_{i+1}$  (bzw.  $a_1$ )  
 $\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, a_{i+2}, \dots, a_n)$

```
void erase_after(ListItem* p) {  
    ListItem* q = p->next;  
  
    p->next = q->next;  
    delete q;  
}
```

`delete_after(p)`

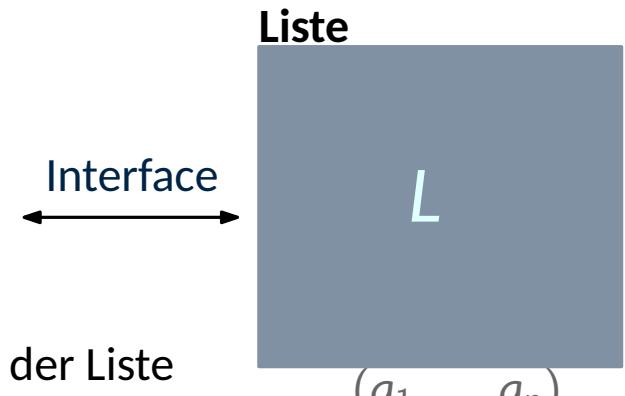


# Operationen auf Listen: Löschen (Delete)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

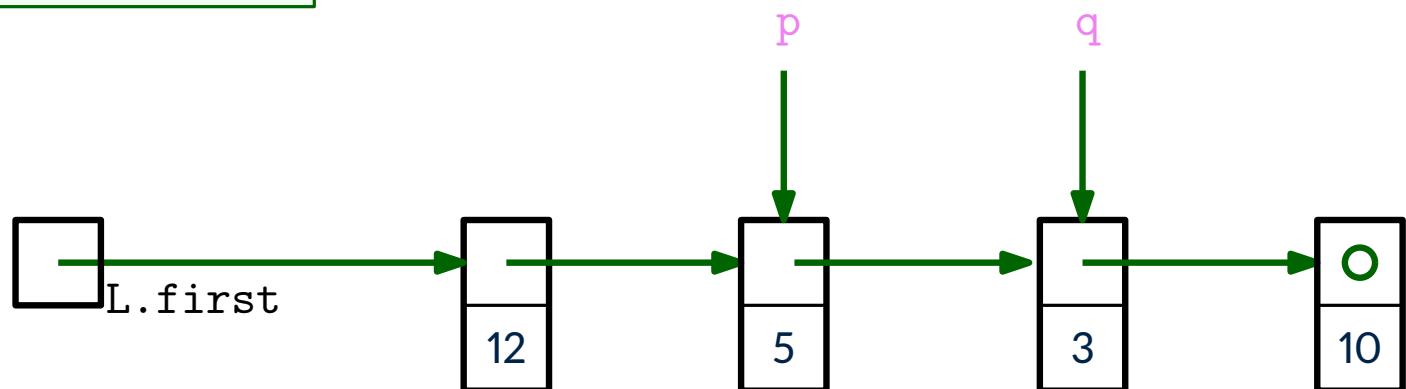


`L.erase_after(p)`

$p$  ist Zeiger auf ein Element  $a_i$  (oder den Anfang) der Liste  
Lösche das Element  $a_{i+1}$  (bzw.  $a_1$ )  
 $\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, a_{i+2}, \dots, a_n)$

```
void erase_after(ListItem* p) {  
    ListItem* q = p->next;  
  
    p->next = q->next;  
    delete q;  
}
```

`delete_after(p)`

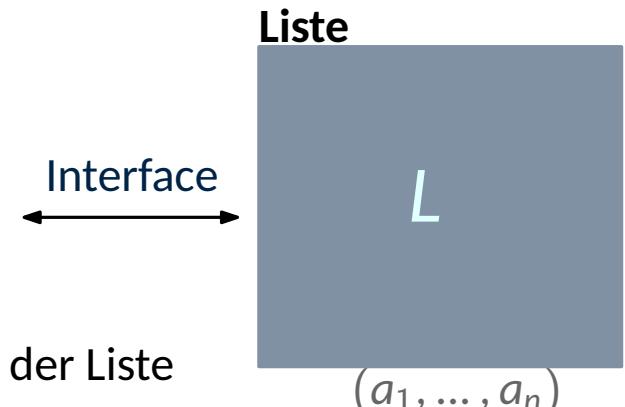


# Operationen auf Listen: Löschen (Delete)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

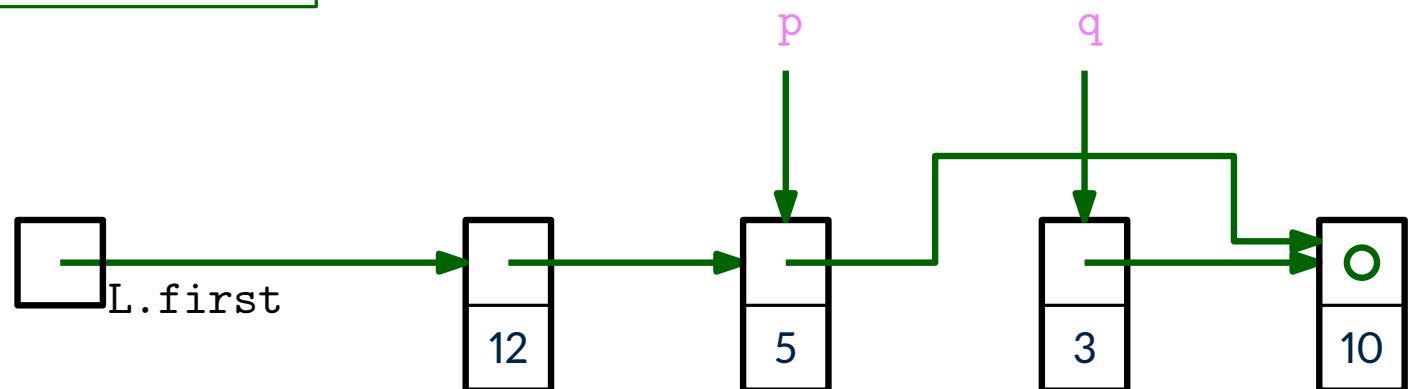


`L.erase_after(p)`

$p$  ist Zeiger auf ein Element  $a_i$  (oder den Anfang) der Liste  
Lösche das Element  $a_{i+1}$  (bzw.  $a_1$ )  
 $\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, a_{i+2}, \dots, a_n)$

```
void erase_after(ListItem* p) {  
    ListItem* q = p->next;  
  
    p->next = q->next;  
    delete q;  
}
```

`delete_after(p)`

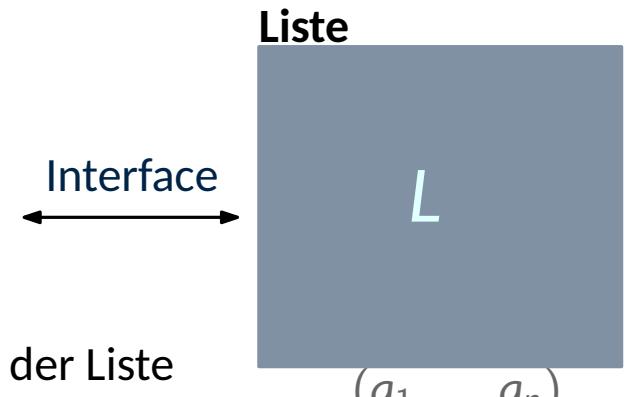


# Operationen auf Listen: Löschen (Delete)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

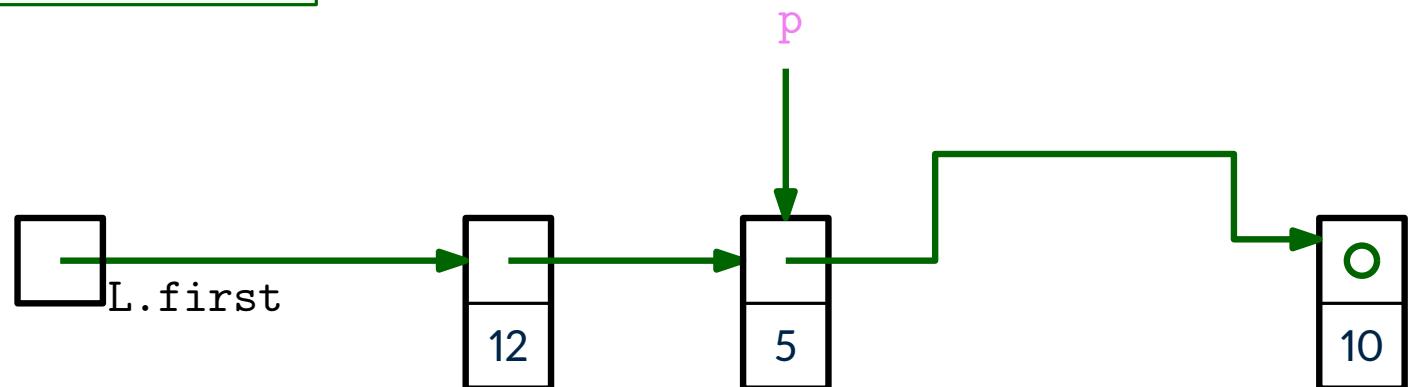


`L.erase_after(p)`

$p$  ist Zeiger auf ein Element  $a_i$  (oder den Anfang) der Liste  
Lösche das Element  $a_{i+1}$  (bzw.  $a_1$ )  
 $\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, a_{i+2}, \dots, a_n)$

```
void erase_after(ListItem* p) {  
    ListItem* q = p->next;  
  
    p->next = q->next;  
    delete q;  
}
```

`delete_after(p)`

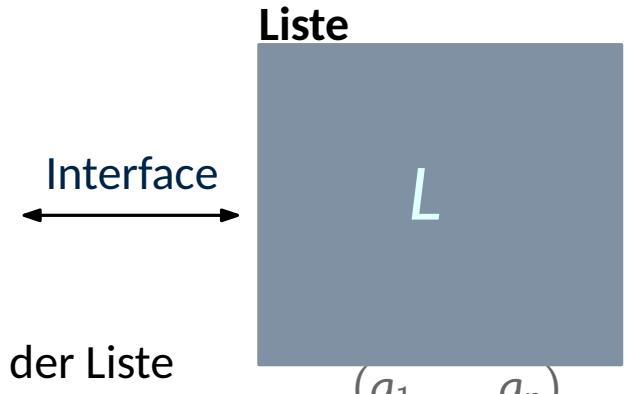


# Operationen auf Listen: Löschen (Delete)

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



`L.erase_after(p)`

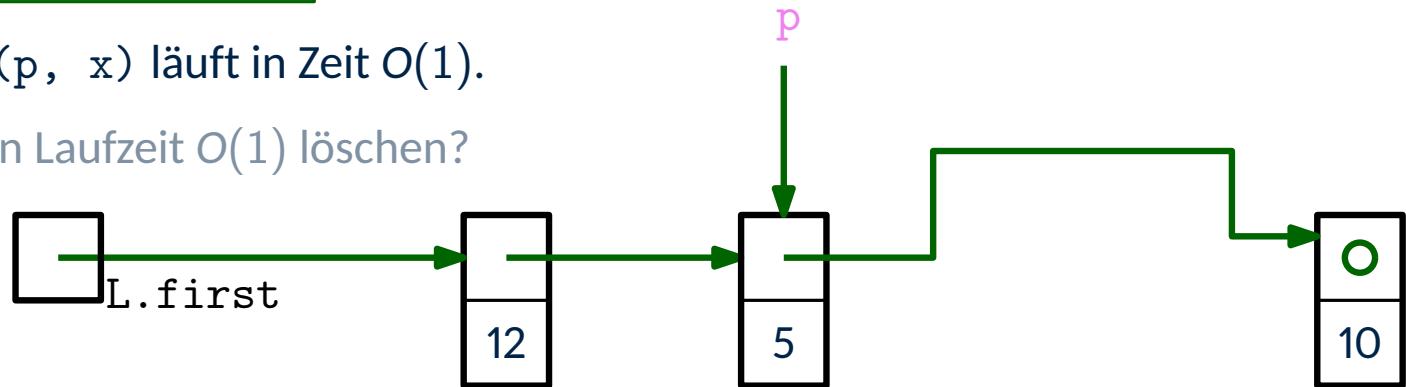
$p$  ist Zeiger auf ein Element  $a_i$  (oder den Anfang) der Liste  
Lösche das Element  $a_{i+1}$  (bzw.  $a_1$ )  
 $\rightarrow (a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_i, a_{i+2}, \dots, a_n)$

```
void erase_after(ListItem* p) {  
    ListItem* q = p->next;  
  
    p->next = q->next;  
    delete q;  
}
```

`delete_after(p)`

**Laufzeit:** `L.erase_after(p, x)` läuft in Zeit  $O(1)$ .

**Frage:** Können wir auch  $p$  in Laufzeit  $O(1)$  löschen?

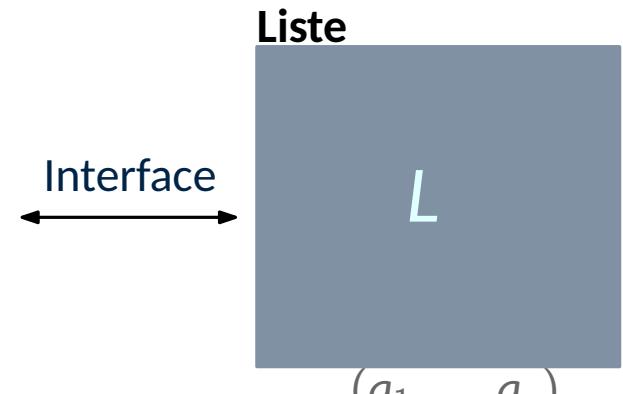


# Weitere Operationen auf Listen

Eine verkettete Liste repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

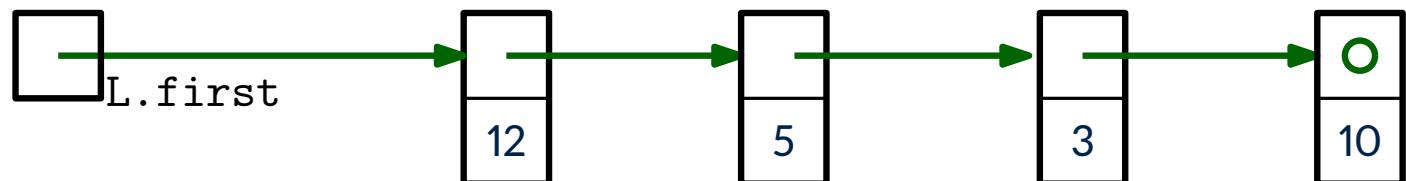


L.find(x)	$O(n)$
L.insert_after(p, x)	$O(1)$
L.erase_after(p)	$O(1)$
L.insert(p, x)	$O(1)$
L.erase(p)	$O(n)$ Problem: Löschen des letzten Elements
L.push_front(x)	$O(1)$
L.push_back(x)	$O(n)$ Hinweis: Man kann einen Zeiger auf das letzte Element speichern für $O(1)$

Merke: Die Laufzeit wird in Abhängigkeit der gespeicherten Elemente angegeben

Linked Lists erlauben auch Operationen of (langen) Teillisten in Zeit  $O(1)$

↗ siehe kommende Folien zur splice-Operation auf doppelt verketteten Listen



# Doppelt verkettete Listen

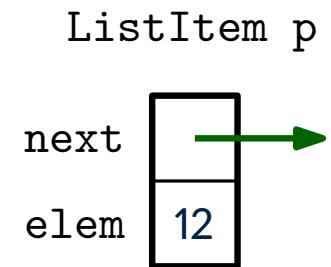
# Doppelt verkettete Listen

**Doppelt verkettete Listen (Doubly Linked Lists)** sind flexibler als Linked Lists

Idee: wir benutzen zusätzliche Zeiger auf das vorhergehende Element

**Grundbaustein:** Item, bestehend aus einem Element und einem Zeiger auf das nächste Item

```
struct ListItem {  
    ListItem* next; //Zeiger auf das nächste Item  
  
    type elem;  
};
```



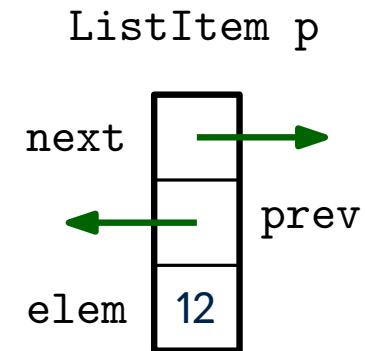
# Doppelt verkettete Listen

**Doppelt verkettete Listen (Doubly Linked Lists)** sind flexibler als Linked Lists

Idee: wir benutzen zusätzliche Zeiger auf das vorhergehende Element

**Grundbaustein:** Item, bestehend aus einem Element und einem Zeiger auf das nächste Item und einem Zeiger auf das vorhergehende Item

```
struct DListItem {  
    DListItem* next; //Zeiger auf das nächste Item  
    DListItem* prev; //Zeiger auf das vorhergehende Item  
    type elem;  
};
```



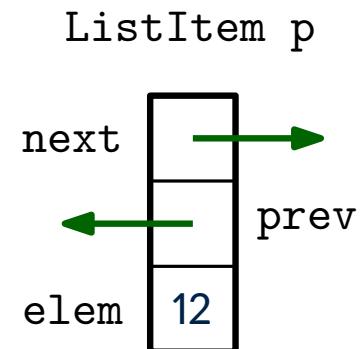
# Doppelt verkettete Listen

Doppelt verkettete Listen (Doubly Linked Lists) sind flexibler als Linked Lists

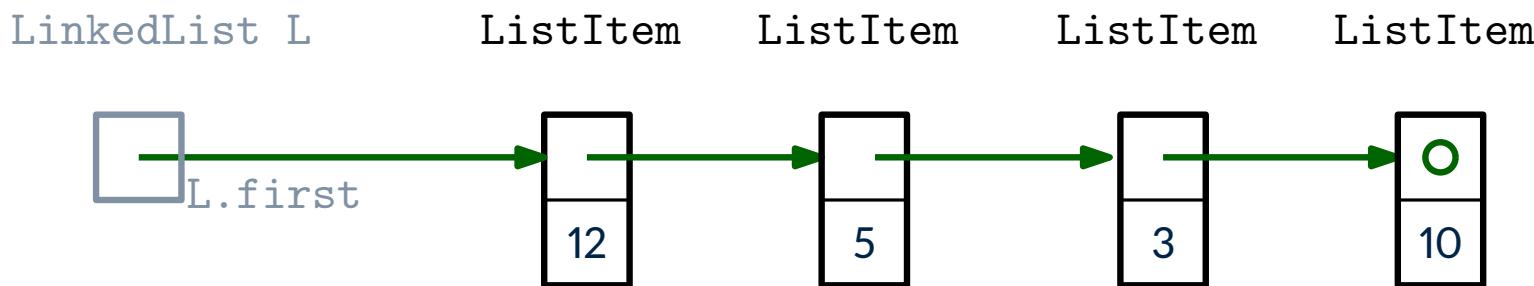
Idee: wir benutzen zusätzliche Zeiger auf das vorhergehende Element

**Grundbaustein:** Item, bestehend aus einem Element und einem Zeiger auf das nächste Item und einem Zeiger auf das vorhergehende Item

```
struct DListItem {  
    DListItem* next; //Zeiger auf das nächste Item  
    DListItem* prev; //Zeiger auf das vorhergehende Item  
    type elem;  
};
```



Daraus bildet sich eine Liste an Elementen (12, 5, 3, 10) wie folgt:



```
class LinkedList {  
    ListItem* first;  
  
    ...  
};
```

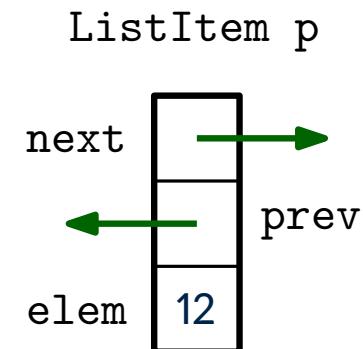
# Doppelt verkettete Listen

Doppelt verkettete Listen (Doubly Linked Lists) sind flexibler als Linked Lists

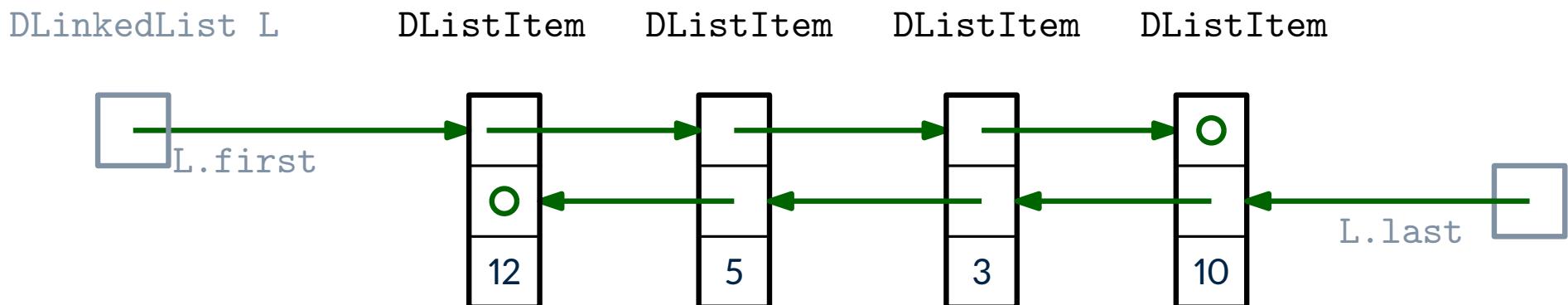
Idee: wir benutzen zusätzliche Zeiger auf das vorhergehende Element

**Grundbaustein:** Item, bestehend aus einem Element und einem Zeiger auf das nächste Item und einem Zeiger auf das vorhergehende Item

```
struct DListItem {  
    DListItem* next; //Zeiger auf das nächste Item  
    DListItem* prev; //Zeiger auf das vorhergehende Item  
    type elem;  
};
```



Daraus bildet sich eine Liste an Elementen (12, 5, 3, 10) wie folgt:



VARIANTE 1

15 - 6

```
class DLinkedList {  
    DListItem* first;  
    DListItem* last;  
    ...  
};
```

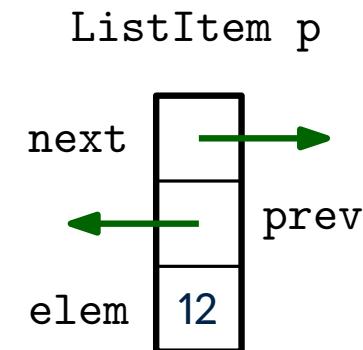
# Doppelt verkettete Listen

Doppelt verkettete Listen (Doubly Linked Lists) sind flexibler als Linked Lists

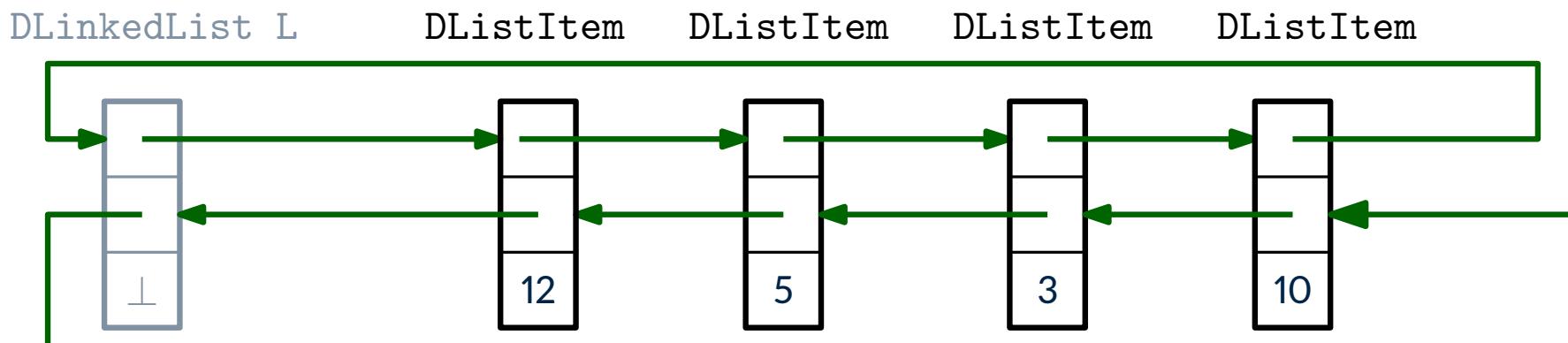
Idee: wir benutzen zusätzliche Zeiger auf das vorhergehende Element

**Grundbaustein:** Item, bestehend aus einem Element und einem Zeiger auf das nächste Item und einem Zeiger auf das vorhergehende Item

```
struct DListItem {  
    DListItem* next; //Zeiger auf das nächste Item  
    DListItem* prev; //Zeiger auf das vorhergehende Item  
    type elem;  
};
```



Daraus bildet sich eine Liste an Elementen (12, 5, 3, 10) wie folgt:



**VARIANTE 2**

15 - 8

```
class DLinkedList {  
    DListItem* h;  
    ...  
};
```

**Invariante:** Für alle `DListItem*` p gilt:  
 $p->next->prev == p->prev->next == p$

# Doppelt verkettete Listen: Splice

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

find, insert, erase lassen sich ähnlich zu Linked Lists implementieren

**Vorteil:** insert und erase direkt unterstützt  
nicht nur insert\_after und erase\_after

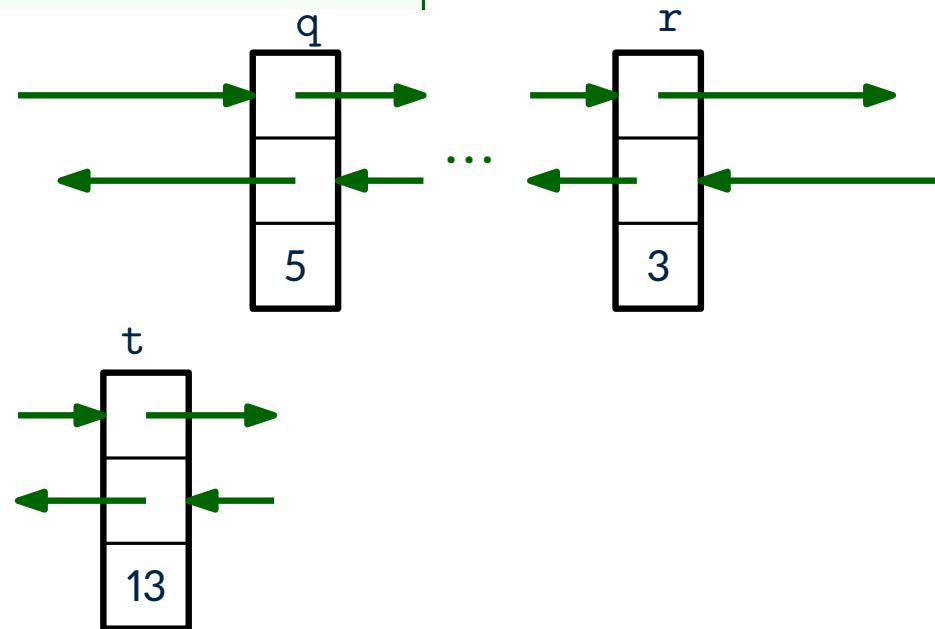
Doppelt verkettete Liste



Schauen uns nun Operationen auf ganzen Teillisten an:

splice(q, r, t)     q, r sind Zeiger auf das erste und letzte Item einer Teilliste ( $a_i, \dots, a_j$ )  
                      t ist Zeiger auf ein Item b nach dem wir einfügen wollen  
 $(\dots, a_{i-1}, a_i, \dots, a_j, a_{j+1}, \dots)$  und  $(\dots, b, b', \dots) \rightarrow (\dots, a_{i-1}, a_{j+1}, \dots)$  und  $(\dots, b, a_i, \dots, a_j, b', \dots)$

```
void splice(DListItem* q, DListItem* r, DListItem* t) {  
    ListItem* p = q->prev;  
    ListItem* s = r->next;  
    p->next = s;  
    s->prev = p;  
  
    ListItem* u = t->next;  
    r->next = u;  
    u->prev = r;  
    t->next = q;  
    q->prev = t;  
}
```



# Doppelt verkettete Listen: Splice

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

find, insert, erase lassen sich ähnlich zu Linked Lists implementieren

**Vorteil:** insert und erase direkt unterstützt

nicht nur insert\_after und erase\_after

Doppelt verkettete Liste

Interface

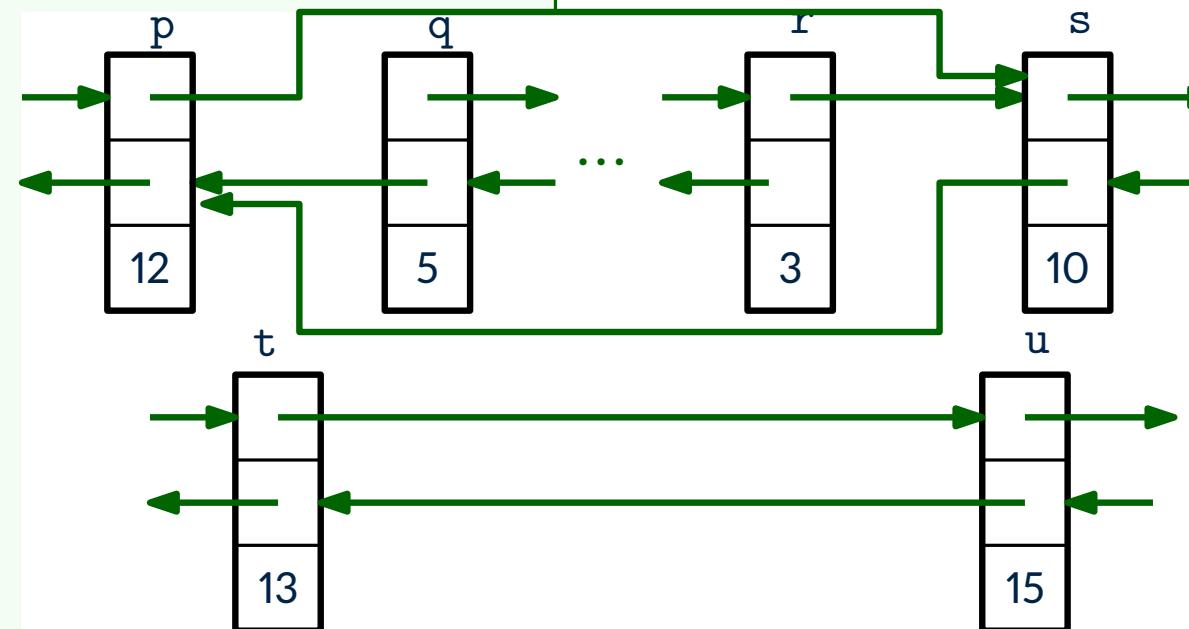
L

$(a_1, \dots, a_n)$

Schauen uns nun Operationen auf ganzen Teillisten an:

splice(q, r, t)     q, r sind Zeiger auf das erste und letzte Item einer Teilliste  $(a_i, \dots, a_j)$   
                      t ist Zeiger auf ein Item b nach dem wir einfügen wollen  
 $(\dots, a_{i-1}, a_i, \dots, a_j, a_{j+1}, \dots) \text{ und } (\dots, b, b', \dots) \rightarrow (\dots, a_{i-1}, a_{j+1}, \dots) \text{ und } (\dots, b, a_i, \dots, a_j, b', \dots)$

```
void splice(DListItem* q, DListItem* r, DListItem* t) {  
    ListItem* p = q->prev;  
    ListItem* s = r->next;  
    p->next = s;  
    s->prev = p;  
  
    ListItem* u = t->next;  
    r->next = u;  
    u->prev = r;  
    t->next = q;  
    q->prev = t;
```



# Doppelt verkettete Listen: Splice

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?

find, insert, erase lassen sich ähnlich zu Linked Lists implementieren

**Vorteil:** insert und erase direkt unterstützt

nicht nur insert\_after und erase\_after

Doppelt verkettete Liste

Interface

L

$(a_1, \dots, a_n)$

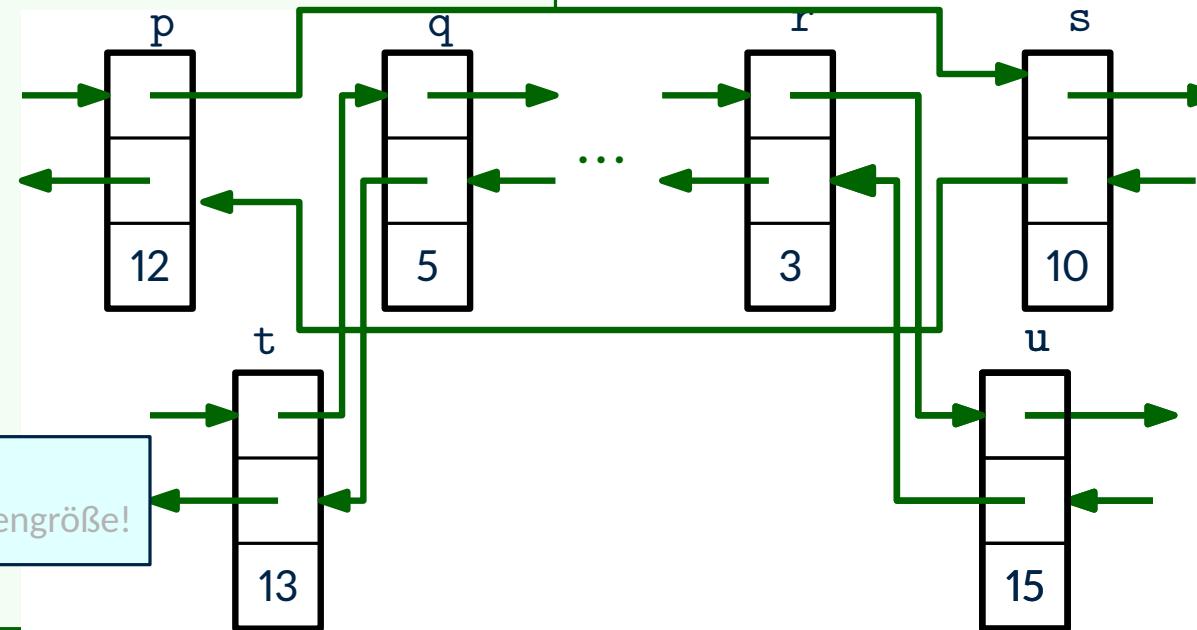
Schauen uns nun Operationen auf ganzen Teillisten an:

splice(q, r, t)     q, r sind Zeiger auf das erste und letzte Item einer Teilliste  $(a_i, \dots, a_j)$   
                      t ist Zeiger auf ein Item b nach dem wir einfügen wollen  
 $(\dots, a_{i-1}, a_i, \dots, a_j, a_{j+1}, \dots) \text{ und } (\dots, b, b', \dots) \rightarrow (\dots, a_{i-1}, a_{j+1}, \dots) \text{ und } (\dots, b, a_i, \dots, a_j, b', \dots)$

```
void splice(DListItem* q, DListItem* r, DListItem* t) {
    ListItem* p = q->prev;
    ListItem* s = r->next;
    p->next = s;
    s->prev = p;

    ListItem* u = t->next;
    r->next = u;
    u->prev = r;
    t->next = q;
    q->prev = t;
```

Laufzeit  $O(1)$   
unabhängig von Teillistengröße!



# Dynamische Arrays

# Dynamische Arrays

---

Listen können um weitere Elemente erweitert werden.

Geht das auch für Arrays?

Wir betrachten **dynamische Arrays** (d.h., ihre Größe ist veränderbar) mit folgenden Operationen

- Auswertung  $A[i]$  und Zuweisung  $A[i] = x$  wie zuvor
- $A.push\_back(x)$  - fügt  $x$  als zusätzliches Element ans Ende ein
- $A.pop\_back()$  - löscht das letzte Element des Arrays
- $A.size()$  - gibt die aktuelle Anzahl an Elementen zurück

Wie können wir diese effizient implementieren?

- Wenn wir ein Array fester Größe  $N$  zu Verfügung haben, können wir es benutzen solange  $A.size() \leq N$ .
- Sobald  $A.size() > N$  wird, müssen wir ein neues Arrays für mindestens  $N + 1$  Elemente erschaffen
  - **Idee:** reservieren Speicherbereich für ein Arrays der Größe  $N + 1$  (Allokation) und kopieren alle Elemente vom alten Array in das neue Array

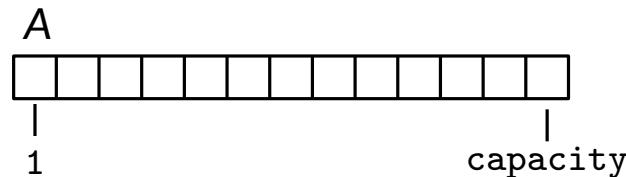
**Frage:** Ist das eine gute Idee?

- Wir betrachten folgende Lösung:
  - Sobald die Kapazität überschritten wird, benutzen wir ein Array **doppelter** Größe

# Dynamische Arrays: Pseudocode (push\_back)

---

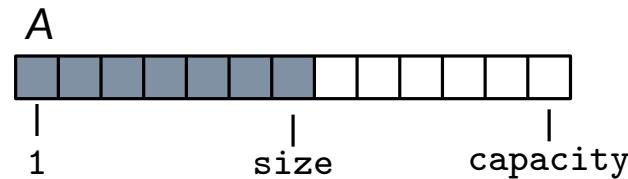
```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
  
    }  
    ...  
}
```



# Dynamische Arrays: Pseudocode (push\_back)

---

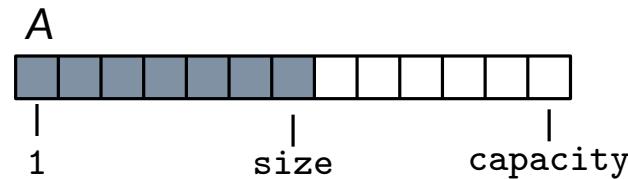
```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
  
    }  
    ...  
}
```



# Dynamische Arrays: Pseudocode (push\_back)

---

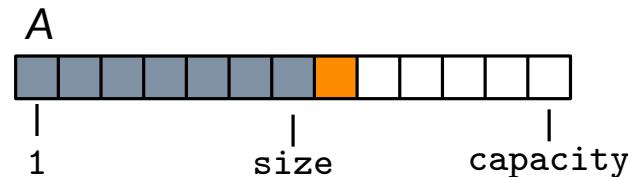
```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
  
            A[size+1] = x;  
            size++;  
        }  
        ...  
}
```



# Dynamische Arrays: Pseudocode (push\_back)

---

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
  
            A[size+1] = x;  
            size++;  
        }  
        ...  
}
```

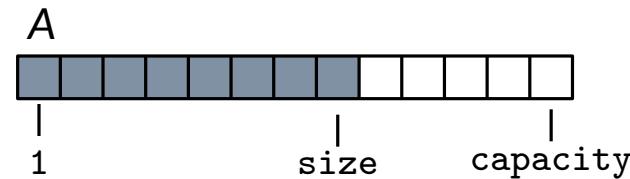


# Dynamische Arrays: Pseudocode (push\_back)

---

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity
```

```
    public:  
        int size() { return size; }  
  
        void push_back(type x) {
```



```
            A[size+1] = x;  
            size++;
```

```
}
```

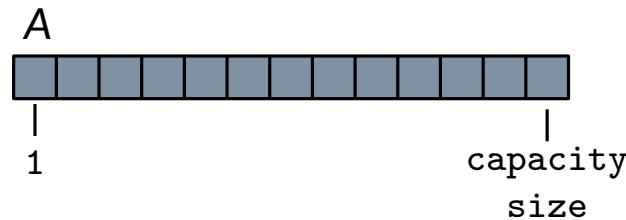
```
...
```

```
}
```

# Dynamische Arrays: Pseudocode (push\_back)

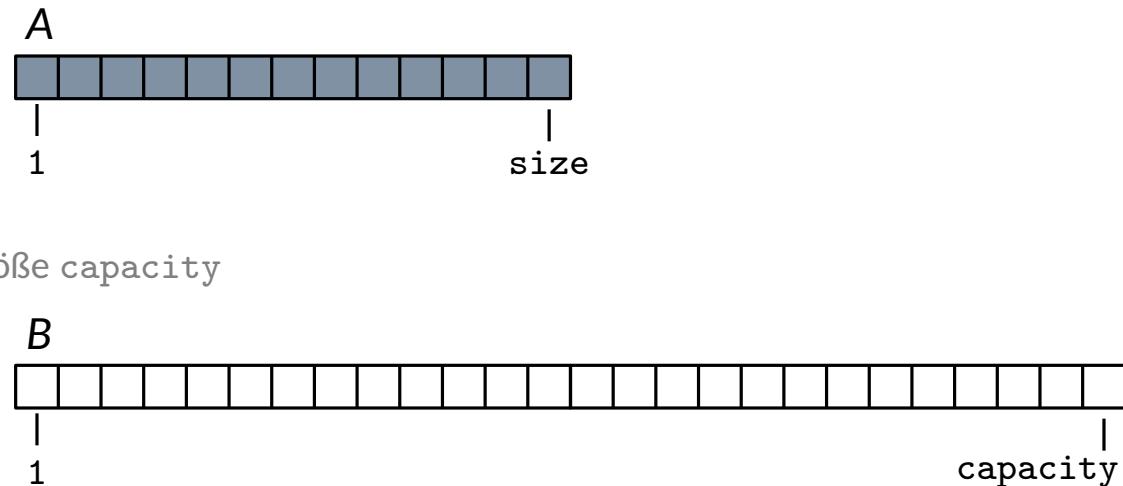
---

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
            if (size == capacity) {  
                ...  
            }  
            A[size+1] = x;  
            size++;  
        }  
    }  
}
```



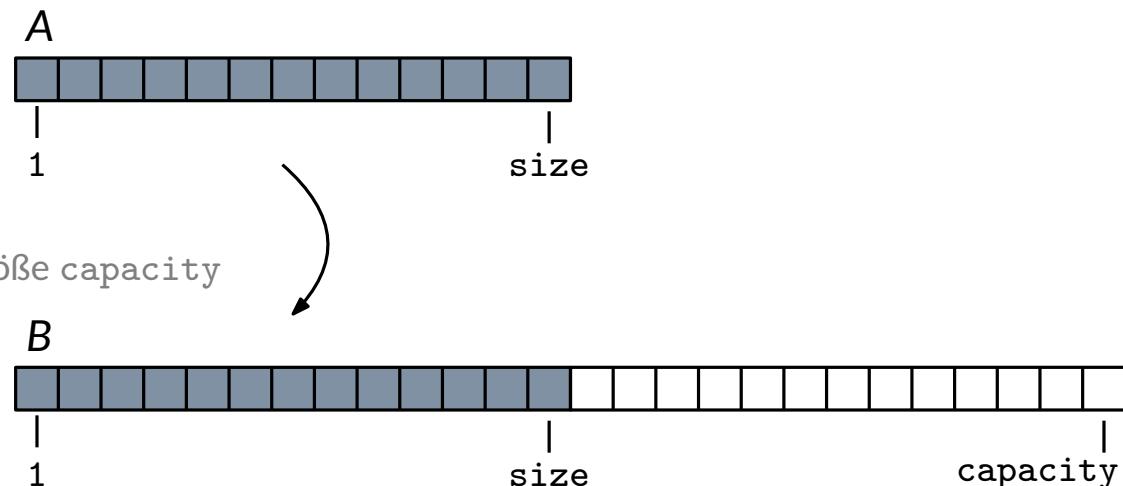
# Dynamische Arrays: Pseudocode (push\_back)

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
            if (size == capacity) {  
                capacity = 2*capacity;  
                type* B = new type[capacity];  
  
                copy A[1..size] to B[1..size]  
  
                delete[] A;  
                A = B;  
            }  
            A[size+1] = x;  
            size++;  
        }  
        ...  
}
```



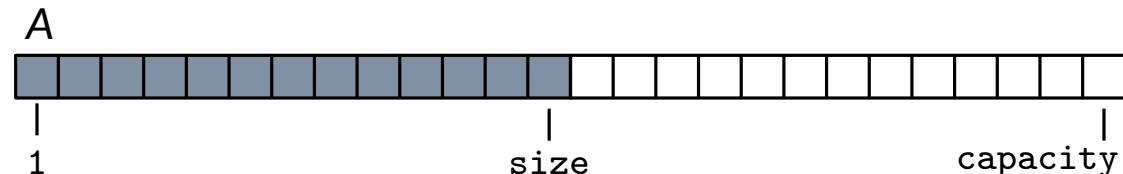
# Dynamische Arrays: Pseudocode (push\_back)

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
  
        void push_back(type x) {  
            if (size == capacity) {  
                capacity = 2*capacity;  
                type* B = new type[capacity];  
  
                copy A[1..size] to B[1..size]  
  
                delete[] A;  
                A = B;  
            }  
            A[size+1] = x;  
            size++;  
        }  
        ...  
}
```



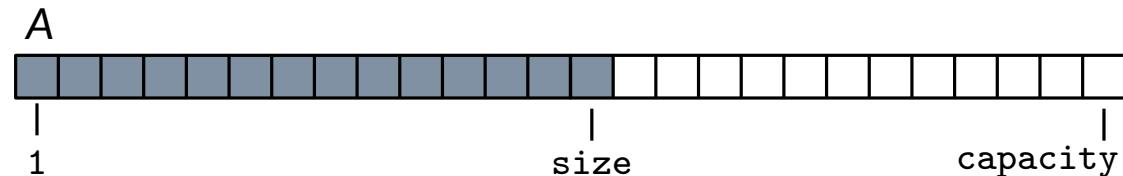
# Dynamische Arrays: Pseudocode (push\_back)

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
        void push_back(type x) {  
            if (size == capacity) {  
                capacity = 2*capacity;  
                type* B = new type[capacity];  
  
                copy A[1..size] to B[1..size]  
  
                delete[] A;  
                A = B;  
            }  
            A[size+1] = x;  
            size++;  
        }  
        ...  
}
```



# Dynamische Arrays: Pseudocode (push\_back)

```
class DynArray {  
    private:  
        int size; //tatsächliche Größe des Arrays  
        int capacity;  
        type* A; //aktuell allokiert Speicher der Größe capacity  
  
    public:  
        int size() { return size; }  
        void push_back(type x) {  
            if (size == capacity) {  
                capacity = 2*capacity;  
                type* B = new type[capacity];  
  
                copy A[1..size] to B[1..size]  
  
                delete[] A;  
                A = B;  
            }  
            A[size+1] = x;  
            size++;  
        }  
        ...  
}
```



# Dynamische Arrays: Pseudocode (pop\_back)

---

Wir könnten das letzten Element des Arrays wie folgt löschen:

```
void pop_back() { size--; }
```

Frage: Was ist das Problem dieser Lösung?

Wir schrumpfen das allokierte Array auf die Hälfte, **sobald**  
die tatsächliche Größe ein Viertel der allokierten Größe unterschreitet

Warum ein Viertel? ↗ Übungsaufgabe

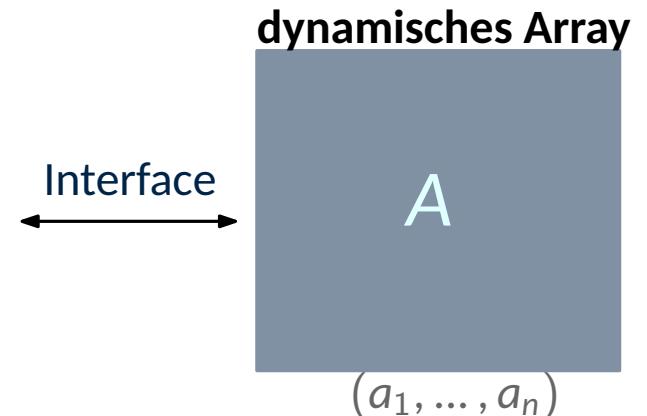
```
void pop_back() {
    size--;
    if (4*size <= capacity) {
        capacity = capacity/2;
        type* B = new type[capacity];
        copy A[1..size] to B[1..size]
        delete[] A;
        A = B;
    }
}
```

# Operationen auf dynamischen Arrays

Ein dynamisches Array repräsentiert eine Folge  $(a_1, \dots, a_n)$ .

Welche Operationen werden unterstützt?

Wie effizient können wir sie ausführen?



$A[i]$	$O(1)$
$A[i] = x$	$O(1)$
$A.size()$	$O(1)$
$A.push\_back(x)$	$O(n)$
$A.pop\_back()$	$O(n)$

Ist die Laufzeit von `push_back(x)` und `pop_back()` wirklich so schlecht?

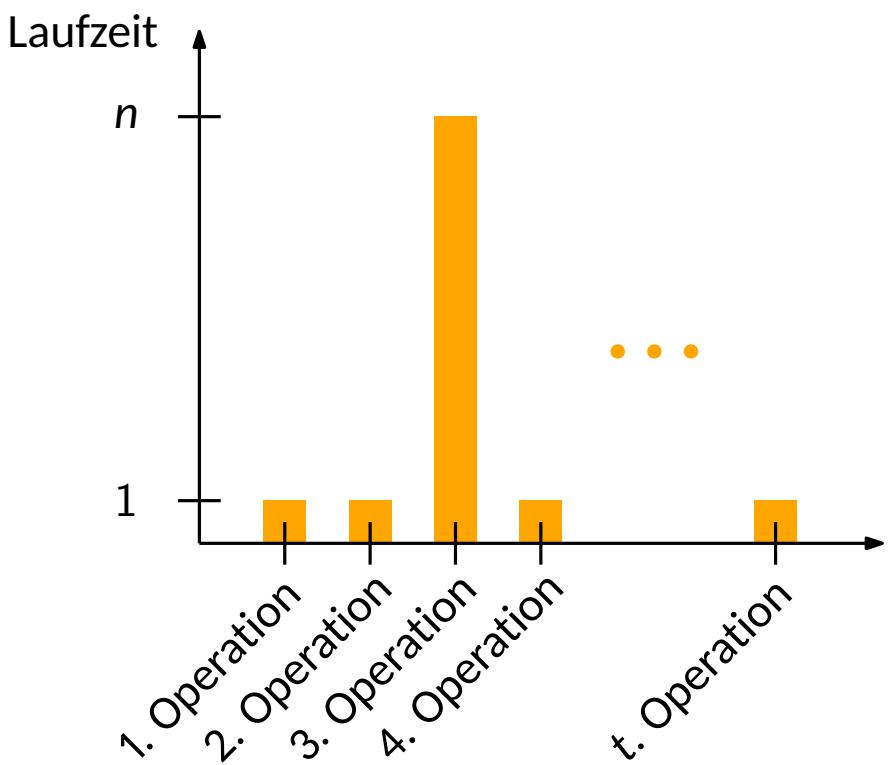
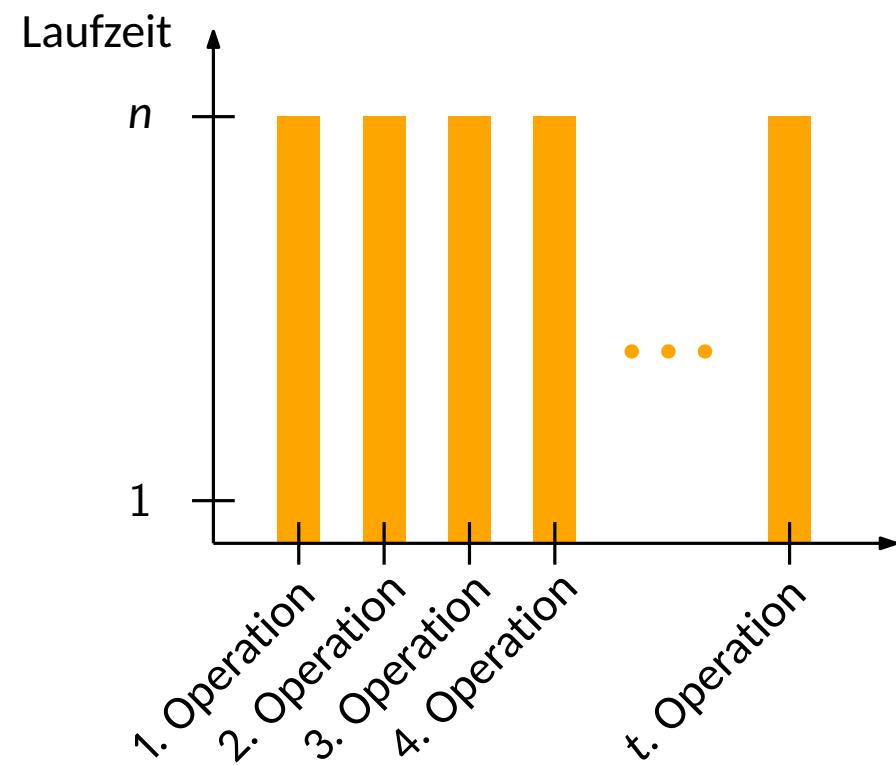
Wie häufig tritt der schlechteste Fall ein?

**Intuition:** wir haben nur **selten** mehr als  $O(1)$  Aufwand

# Amortisierte Analyse

# Amortisierte Analyse

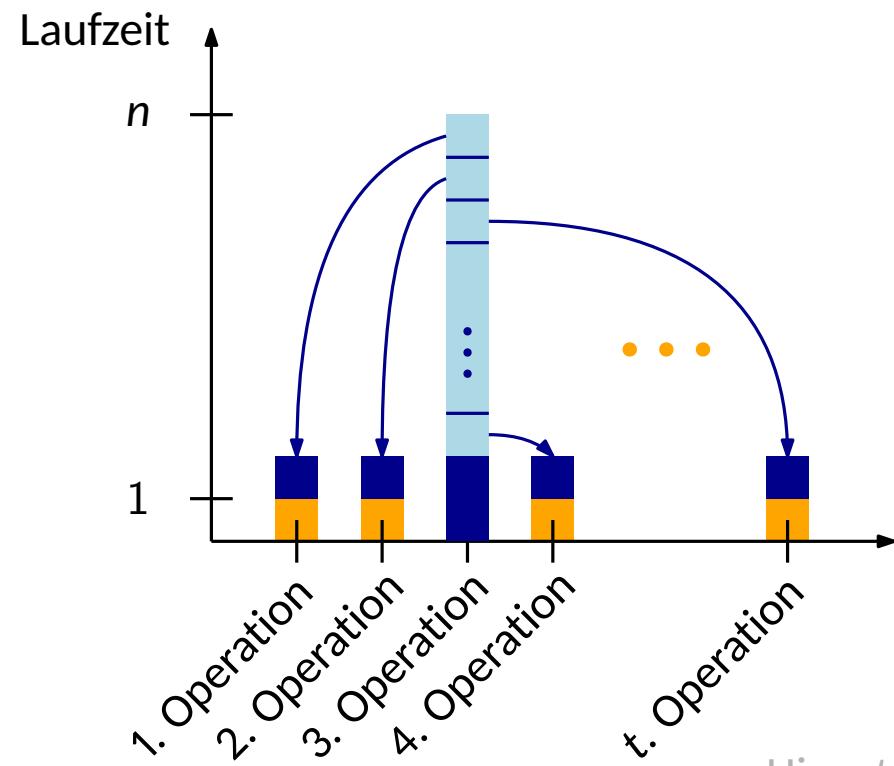
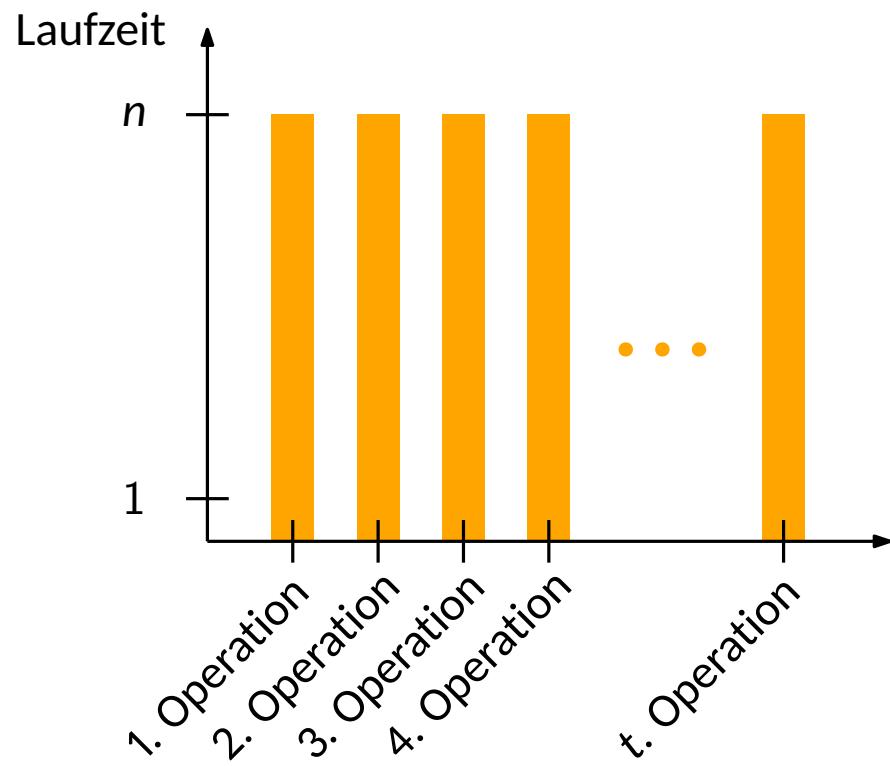
Zwei Szenarien mit Worst-case-Laufzeit  $O(n)$  pro Operation:



Sind beide Szenarien gleich schlecht? **Nein!**

# Amortisierte Analyse

Zwei Szenarien mit Worst-case-Laufzeit  $O(n)$  pro Operation:



Hier:  $t = n - 1$

Sind beide Szenarien gleich schlecht? **Nein!**

Durchschnittliche Laufzeit über  $t$  Operationen:  $n$

Durchschnittliche Laufzeit über  $t$  Operationen: 2

→ **amortisierte Laufzeit 2**

# Amortisierte Laufzeitanalyse

---

## Theorem

Sei A ein dynamisches Array, das anfangs leer ist.

Sei  $\sigma_1, \dots, \sigma_t$  eine beliebige Folge von `push_back(x)`- oder `pop_back()`-Operationen.

Die Gesamtlaufzeit von  $\sigma_1, \dots, \sigma_t$  ist in  $O(t)$ .

→ asymptotisch bestmögliche Gesamtlaufzeit

→ **amortisierte Laufzeit** einer Operation ist in  $O(1)$ . Die "durchschnittliche" Laufzeit über  $t$  Operationen ist  $O(1)$

## Korollar

Dynamische Arrays implementieren die Operationen Auswertung, Zuweisung und `size()` in Worst-Case-Laufzeit  $O(1)$  und die Operationen `push_back(x)`, `pop_back()` in amortisierter Laufzeit  $O(1)$ .

Beweis vom Theorem: ↗ Tafelpräsentation

# Quiz: Arrays vs Linked Lists

---

	Dynamisches Array	Doubly Linked List
Auswertung/Zuweisung $A[i] / A[i] = x$		
<code>size()</code>		
<code>push_back(x)</code>		
<code>pop_back()</code>		
<code>push_front(x)</code>		
<code>pop_front()</code>		

# Weiteres Beispiel: Zahl inkrementieren

Zähler

$a$



Interface:

Initialisierung - setzt  $a$  auf 0

increment() - erhöht  $a$  um 1

$$a \leftarrow a + 1 \pmod{2^n}$$

$n$ -stellige Zahl  $a_{n-1} \dots a_0$  zur Basis 2

**Worst-Case-Laufzeit:  $O(n)$**

**Amortisierte Laufzeitanalyse:**

- Laufzeit von increment() ist  $\leq C \cdot (\text{Anzahl geflippter Bits})$
- Wir analysieren die amortisierte Anzahl geflippter Bits

geflippte Bits      Gesamtzahl geflippter Bits

0 0 0 0 0		
0 0 0 0 1	1	1
0 0 0 1 0	2	3
0 0 0 1 1	1	4
0 0 1 0 0	3	7
0 0 1 0 1	1	8
0 0 1 1 0	2	10
0 0 1 1 1	1	11
0 1 0 0 0	4	15
0 1 0 0 1	1	16
0 1 0 1 0	2	18
0 1 0 1 1	1	19
0 1 1 0 0	3	22
0 1 1 0 1	1	23

$$\begin{array}{r} 10010111 \\ + 1 \\ \hline 10011000 \end{array}$$

increment() //  $a = (a[n-1] \dots a[0])$

```
i = 0
while (i < n and a[i] = 1) do
    | a[i] = 0
    | i = i + 1
if (i < n) then
    | a[i] = 1
```

**Lemma**

Die amortisierte Anzahl durch increment() geflippter Bits ist  $\leq 2$  Nach  $t$  increment()'s haben wir  $\leq 2t$  Bits geflippt.

**Konsequenz:**

increment() hat amortisierte Laufzeit  $O(1)$ .

Beweis vom Lemma → Tafelpräsentation

# Zusammenfassung

---

- Arrays als Datenstruktur
- (Einfach/Doppelt) Verkettete Listen
- Dynamische Arrays
- Analysekonzept: **amortisierte Analyse**

Algorithmen und Datenstrukturen SS'23

# Kapitel 8: Elementare Datenstrukturen II - Stacks und (Priority-)Queues

Marvin Künemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letztes Kapitel: grundlegende Datenstrukturen für Sequenzen

Jetzt: spezialisierte Datenstrukturen (Stacks, Queues, Priorityqueues)

Insbesondere behandeln wir in diesem Kapitel:

- **Stacks, Queues and Deques** als spezialisierte Datenstrukturen
- **Prioritätswarteschlangen**
  - erste baumbasierte Datenstruktur: **Heaps**
  - HeapSort

# Stacks

Ein **Stack S** repräsentiert eine Folge  $(a_1, \dots, a_n)$

Wir haben nur eingeschränkte Zugriffsmöglichkeiten:

		Interface	Stack
<code>S.last()</code>	- gibt das letzte Element der Folge zurück	Laufzeit $O(1)$	
<code>S.top()</code>	gibt $a_n$ zurück		
<code>S.push_back(x)</code>	- fügt $x$ an das Ende der Folge ein	Laufzeit $O(1)$	
<code>S.push(x)</code>	$(a_1, \dots, a_n)$ wird zu $(a_1, \dots, a_n, x)$		
<code>S.pop_back()</code>	- löscht das letzte Element der Folge	Laufzeit $O(1)$	
<code>S.pop()</code>	$(a_1, \dots, a_{n-1}, a_n)$ wird zu $(a_1, \dots, a_{n-1})$		

**Prinzip:** LIFO - "Last In, First Out"

Wir können immer nur auf das zuletzt hinzugefügte Element zugreifen

Wie kann man Stacks implementieren?

1. Möglichkeit: als (dynamisches) Array A

```
S.last(): return A[A.size()]
S.push_back(x): A.push_back(x)
S.pop_back(): A.pop_back()
```

2. Möglichkeit: als einfach verkettete Liste L

```
S.last(): return L.first->elem
S.push_back(x): L.push_front(x)
S.pop_back(): L.pop_front()
```

# Queues

Eine **Queue (Warteschlange)**  $Q$  repräsentiert eine Folge  $a_1, \dots, a_n$

Wir haben nur eingeschränkte Zugriffsmöglichkeiten:

`Q.first()`  
`Q.front()`

- gibt das erste Element der Folge zurück  
gibt  $a_1$  zurück

Interface  
Laufzeit  $O(1)$

`Q.push_back(x)`  
`Q.enqueue(x)`

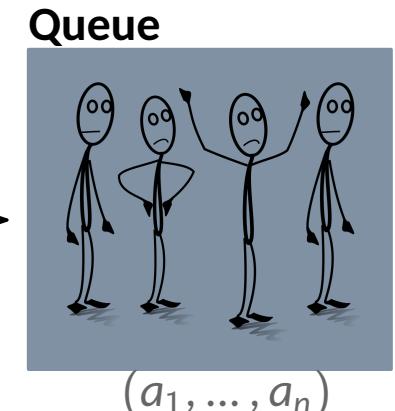
- fügt  $x$  an das Ende der Folge ein  
 $(a_1, \dots, a_n)$  wird zu  $(a_1, \dots, a_n, x)$

Laufzeit  $O(1)$

`Q.pop_front()`  
`Q.dequeue()`

- löscht das erste Element der Folge  
 $(a_1, a_2, \dots, a_n)$  wird zu  $(a_2, \dots, a_n)$

Laufzeit  $O(1)$



**Prinzip:** FIFO - "First In, First Out"

Wir können immer nur auf das frühest eingefügte (nicht herausgenommene) Element zugreifen

Wie kann man Queues implementieren?

1. Möglichkeit: als doppelt verkettete Liste  $L$

`Q.first(): return L.first->elem`  
`Q.push_back(x): L.push_back(x)`  
`Q.pop_front(): L.pop_front()`

2. Möglichkeit: als einfach verkettete Liste  $L$

möglich, wenn wir immer einen Zeiger auf das Listenende speichern!

3. Möglichkeit: als Array      Annahme: wir haben eine Kapazitätsgrenze  $C \rightarrow Q$  wird nie mehr als  $C$  Elemente enthalten

# Implementierung von Queues als Arrays

---



```
class Queue {  
    type A[C];  
    int head; //Kopf der Schlange; anfangs 1  
    int tail; //Schwanz der Schlange; anfangs 1
```

# Implementierung von Queues als Arrays

---

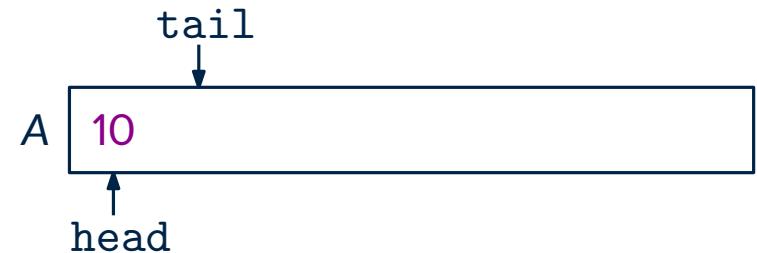
```
class Queue {  
    type A[C];  
    int head; //Kopf der Schlange; anfangs 1  
    int tail; //Schwanz der Schlange; anfangs 1  
    type front() {  
        if (head==tail) {  
            error: Schlange ist leer!  
        }  
        return A[head];  
    }  
    void push_back(type x) {  
        if (head==tail+1 or (head==1 and tail==C)) {  
            error: Queue hat Kapazitätsgrenze überschritten  
        }  
        A[tail] = x;  
        if (tail == C) {  
            tail = 1;  
        } else { tail = tail + 1; }  
    }  
}
```



# Implementierung von Queues als Arrays

push\_back(10)

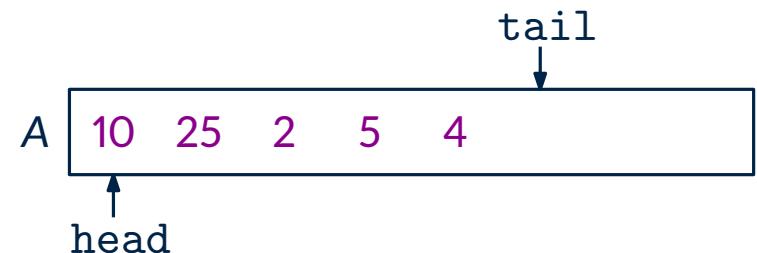
```
class Queue {  
    type A[C];  
    int head; //Kopf der Schlange; anfangs 1  
    int tail; //Schwanz der Schlange; anfangs 1  
    type front() {  
        if (head==tail) {  
            error: Schlange ist leer!  
        }  
        return A[head];  
    }  
    void push_back(type x) {  
        if (head==tail+1 or (head==1 and tail==C)) {  
            error: Queue hat Kapazitätsgrenze überschritten  
        }  
        A[tail] = x;  
        if (tail == C) {  
            tail = 1;  
        } else { tail = tail + 1; }  
    }  
}
```



# Implementierung von Queues als Arrays

```
push_back(10)
front()
push_back(25)
push_back(2)
push_back(5)
push_back(4)
```

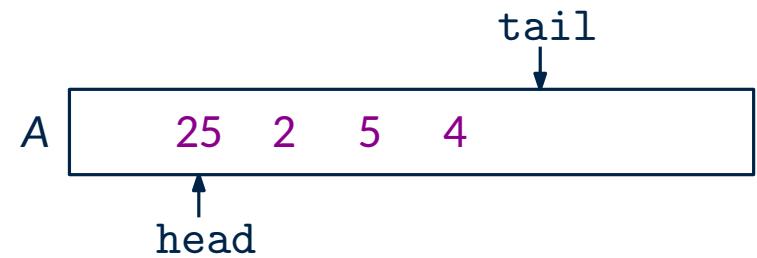
```
class Queue {
    type A[C];
    int head; //Kopf der Schlange; anfangs 1
    int tail; //Schwanz der Schlange; anfangs 1
    type front() {
        if (head==tail) {
            error: Schlange ist leer!
        }
        return A[head];
    }
    void push_back(type x) {
        if (head==tail+1 or (head==1 and tail==C)) {
            error: Queue hat Kapazitätsgrenze überschritten
        }
        A[tail] = x;
        if (tail == C) {
            tail = 1;
        } else { tail = tail + 1; }
    }
}
```



# Implementierung von Queues als Arrays

```
push_back(10)
front()
push_back(25)
push_back(2)
push_back(5)
push_back(4)
pop_front()
```

```
class Queue {
    type A[C];
    int head; //Kopf der Schlange; anfangs 1
    int tail; //Schwanz der Schlange; anfangs 1
    type front() {
        if (head==tail) {
            error: Schlange ist leer!
        }
        return A[head];
    }
    void push_back(type x) {
        if (head==tail+1 or (head==1 and tail==C)) {
            error: Queue hat Kapazitätsgrenze überschritten
        }
        A[tail] = x;
        if (tail == C) {
            tail = 1;
        } else { tail = tail + 1; }
    }
}
```

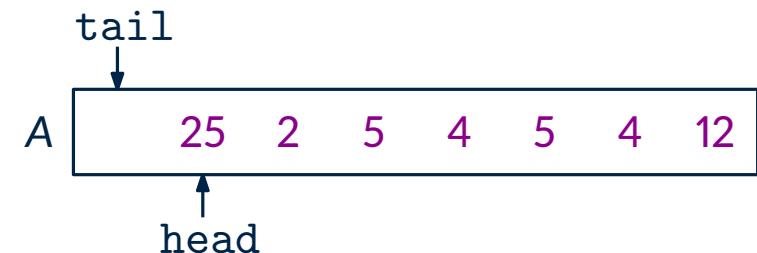


```
type pop_front() {
    if (head==tail) {
        error: Schlange ist leer!
    }
    if (head == C) {
        head = 1;
    } else { head = head + 1; }
}
```

# Implementierung von Queues als Arrays

```
push_back(10)      push_back(12)
front()
push_back(25)
push_back(2)
push_back(5)
push_back(4)
pop_front()
push_back(5)
push_back(4)

class Queue {
    type A[C];
    int head; //Kopf der Schlange; anfangs 1
    int tail; //Schwanz der Schlange; anfangs 1
    type front() {
        if (head==tail) {
            error: Schlange ist leer!
        }
        return A[head];
    }
    void push_back(type x) {
        if (head==tail+1 or (head==1 and tail==C)) {
            error: Queue hat Kapazitätsgrenze überschritten
        }
        A[tail] = x;
        if (tail == C) {
            tail = 1;
        } else { tail = tail + 1; }
    }
}
```

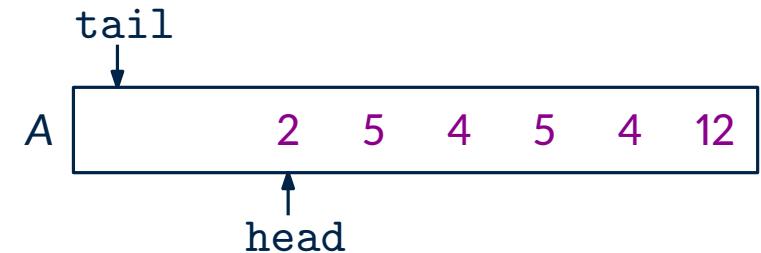


```
type pop_front() {
    if (head==tail) {
        error: Schlange ist leer!
    }
    if (head == C) {
        head = 1;
    } else { head = head + 1; }
}
```

# Implementierung von Queues als Arrays

```
push_back(10)      push_back(12)
front()           pop_front()
push_back(25)     push_back(2)
push_back(2)      push_back(5)
push_back(5)      push_back(4)
push_back(4)      pop_front()
push_back(5)      push_back(5)
push_back(4)

class Queue {
    type A[C];
    int head; //Kopf der Schlange; anfangs 1
    int tail; //Schwanz der Schlange; anfangs 1
    type front() {
        if (head==tail) {
            error: Schlange ist leer!
        }
        return A[head];
    }
    void push_back(type x) {
        if (head==tail+1 or (head==1 and tail==C)) {
            error: Queue hat Kapazitätsgrenze überschritten
        }
        A[tail] = x;
        if (tail == C) {
            tail = 1;
        } else { tail = tail + 1; }
    }
}
```

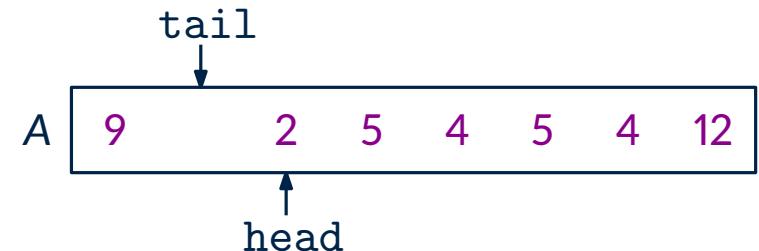


```
type pop_front() {
    if (head==tail) {
        error: Schlange ist leer!
    }
    if (head == C) {
        head = 1;
    } else { head = head + 1; }
}
```

# Implementierung von Queues als Arrays

```
push_back(10)    push_back(12)
front()          pop_front()
push_back(25)    push_back(9)
push_back(2)     push_back(13): error
push_back(5)
push_back(4)
pop_front()
push_back(5)
push_back(4)
```

```
class Queue {
    type A[C];
    int head; //Kopf der Schlange; anfangs 1
    int tail; //Schwanz der Schlange; anfangs 1
    type front() {
        if (head==tail) {
            error: Schlange ist leer!
        }
        return A[head];
    }
    void push_back(type x) {
        if (head==tail+1 or (head==1 and tail==C)) {
            error: Queue hat Kapazitätsgrenze überschritten
        }
        A[tail] = x;
        if (tail == C) {
            tail = 1;
        } else { tail = tail + 1; }
    }
}
```

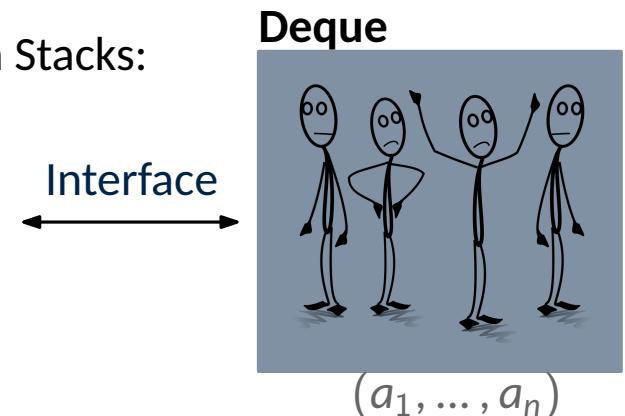


```
type pop_front() {
    if (head==tail) {
        error: Schlange ist leer!
    }
    if (head == C) {
        head = 1;
    } else { head = head + 1; }
}
```

# Double-ended Queues (Deques)

Eine double-ended Queue  $D$  verallgemeinert sowohl Queues als auch Stacks:

- `D.first()`
- `D.last()`
- `D.push_front(x)`
- `D.push_back(x)`
- `D.pop_front()`
- `D.pop_back()`



Wie kann man Deques effizient implementieren?

Frage 1: Als doppelt verkettete Liste?

Frage 2: Als einfach verkettete Liste?

Frage 3: Als Array?

Frage 4: Als Array, auch wenn wir keine Kapazitätsgrenze  $C$  kennen?

Wir können Deques so implementieren, dass alle Operationen Laufzeit  $O(1)$  haben

# Stacks, Queues, Deques: Zusammenfassung

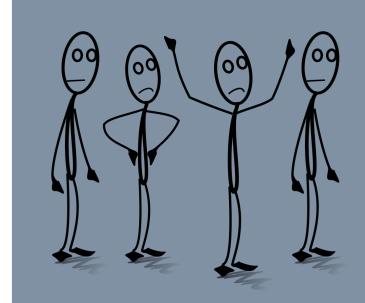
---

Stack



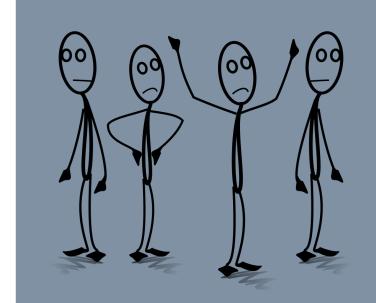
$(a_1, \dots, a_n)$

Queue



$(a_1, \dots, a_n)$

Deque



$(a_1, \dots, a_n)$

`S.last()`

`Q.first()`

`D.first()`

`D.last()`

`S.push_back(x)`

`Q.push_back(x)`

`D.push_front(x)`

`D.push_back(x)`

`S.pop_back()`

`Q.pop_front()`

`D.pop_front()`

`D.pop_back()`

## Implementierungsmöglichkeiten:

1. Verkettete Listen      → alle Operationen  $O(1)$

2. Dynamische Arrays      → alle Operationen amortisiert  $O(1)$

# Stacks, Queues, Deques... Warum?

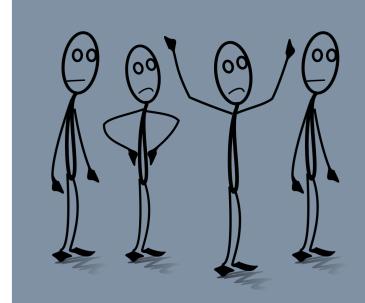
---

Stack



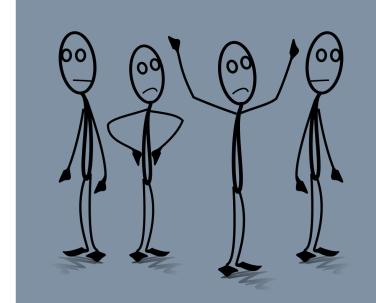
$(a_1, \dots, a_n)$

Queue



$(a_1, \dots, a_n)$

Deque



$(a_1, \dots, a_n)$

Wir können diese Datenstrukturen z.B. mit doppelt verketteten Listen implementieren.

→ Warum betrachten wir diese **Spezialfälle** und benutzen nicht immer doppelt verkettete Listen?

Wir wollen immer die **sparsamste** Datenstruktur wählen, die notwendige Operationen unterstützt

Interface, Speicherbedarf, Implementierungsaufwand, etc.

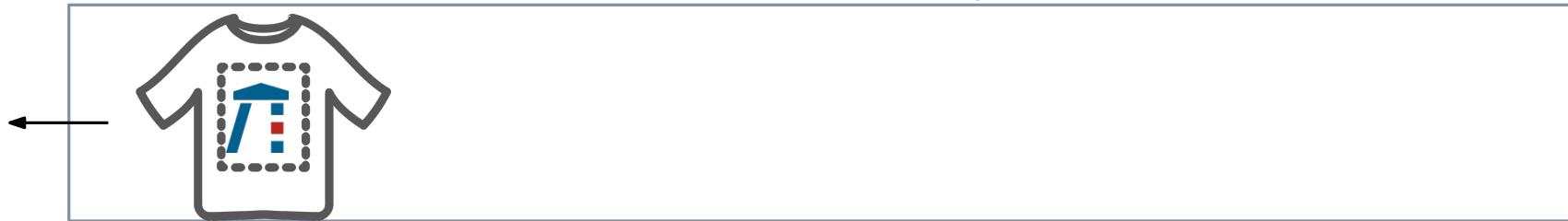
- Arrays benutzen weniger zusätzlichen Speicher als verkettete Listen
- einfache verkettete Listen benutzen weniger zusätzlichen Speicher als doppelt verkettete Listen
- "kleines" Interface bedeutet: weniger Aufwand beim Debuggen
- ...

# Prioritätswarteschlangen

---

**Szenario:** Wir benutzen eine Schlange um eine **Aufgabenliste** zu führen

Queue

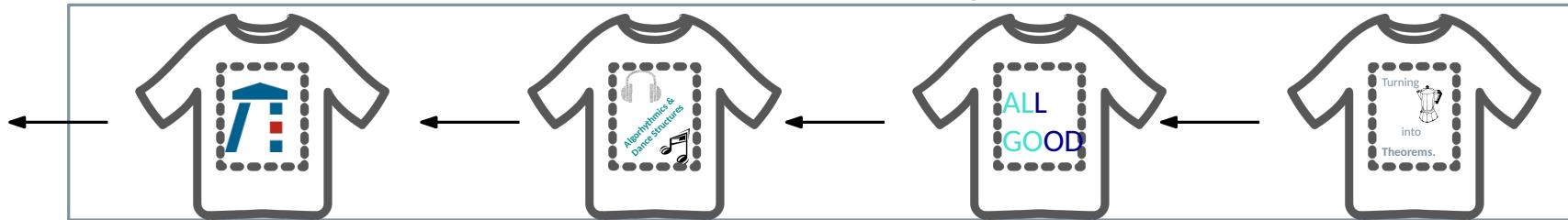


Beispiel: Produktion individualisierter T-Shirts

# Prioritätswarteschlangen

Szenario: Wir benutzen eine Schlange um eine Aufgabenliste zu führen

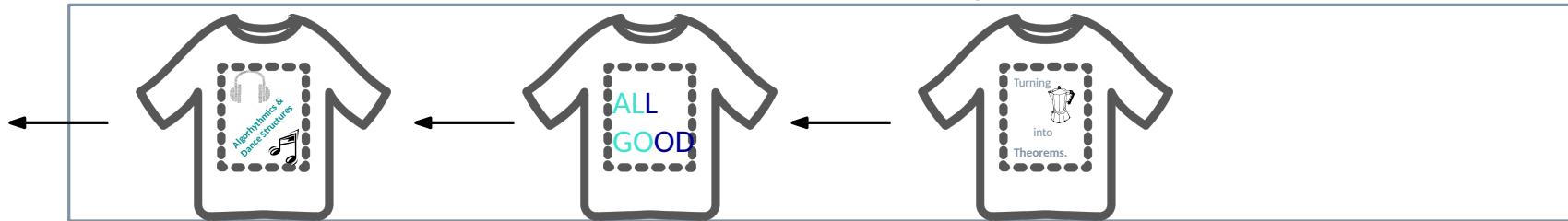
Queue



# Prioritätswarteschlangen

**Szenario:** Wir benutzen eine Schlange um eine **Aufgabenliste** zu führen

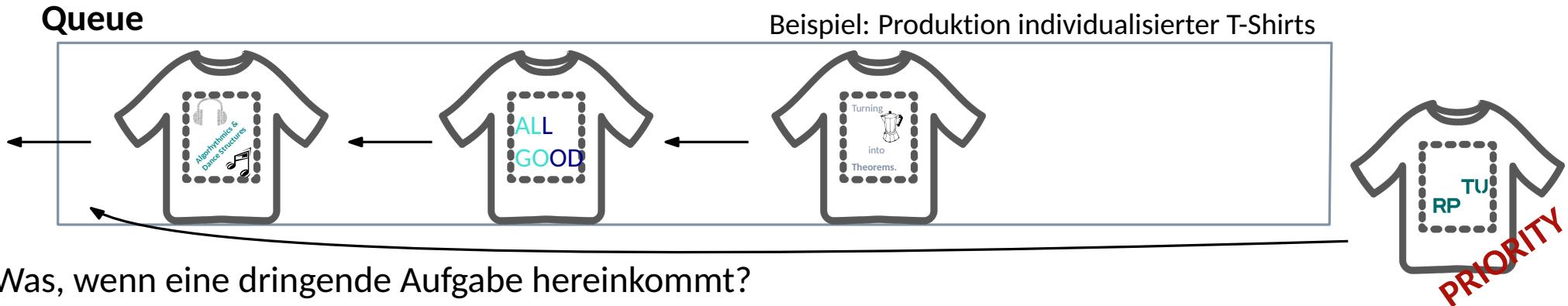
Queue



Beispiel: Produktion individualisierter T-Shirts

# Prioritätswarteschlangen

Szenario: Wir benutzen eine Schlange um eine **Aufgabenliste** zu führen

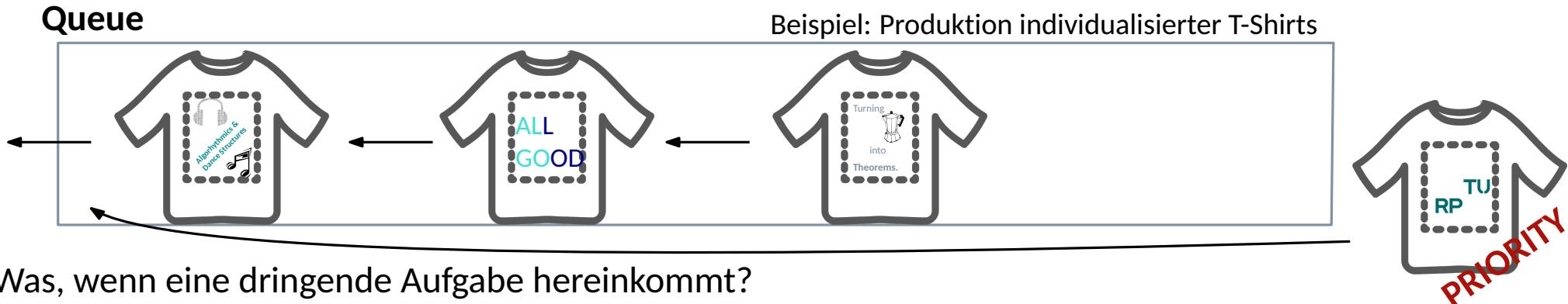


Was, wenn eine dringende Aufgabe hereinkommt?

→ Können wir **Prioritäten** einführen, sodass wir schnell auf die Aufgabe mit **höchster Priorität** zugreifen können?

# Prioritätswarteschlangen

**Szenario:** Wir benutzen eine Schlange um eine **Aufgabenliste** zu führen



Wir betrachten eine Erweiterung von Schlangen, die **Prioritätswartseschlange (Priority Queue)**:

Eine **Priority Queue** speichert eine Menge  $M$  an Schlüsseln unter folgenden Operationen:

haben totale Ordnung

`PQ.build({ $e_1, \dots, e_n$ })`

initialisiert  $M = \{e_1, \dots, e_n\}$

`PQ.insert( $e$ )`

fügt  $e$  in  $M$  ein, d.h.  $M \leftarrow M \cup \{e\}$

`PQ.max()`

**return** max  $M$

`PQ.deleteMax()`

löscht max  $M$ , d.h.  $M \leftarrow M \setminus \{\text{max } M\}$

Priority Queue

Interface

Hinweis: Statt dem entsprechenden Key könnte man auch (ein Zeiger auf) das entsprechende Element zurückgeben ↗ später  
Hinweis 2: Statt mit dem Maximum könnte man Priority Queues mit dem Minimum definieren.

# Bäume

(Gerichteter) Baum  $T$

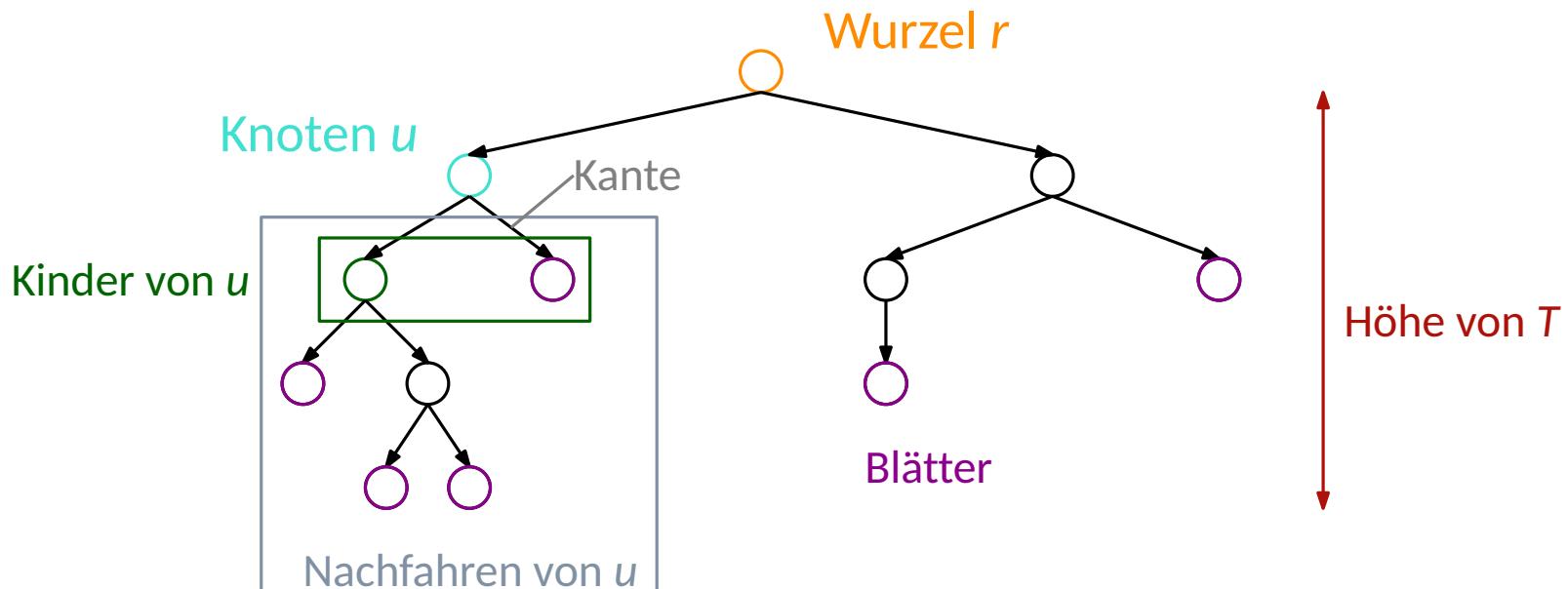
Level 0

Level 1

Level 2

Level 3

Level 4



Wurzel: Knoten ohne Verfahren

Blätter: Knoten ohne Nachfahren

Höhe von  $T$ : das maximale Level eines Knotens in  $T$

$T$  ist Binärbaum, wenn jeder Knoten höchstens zwei Kinder hat

$T$  ist voller Binärbaum, wenn jeder Knoten entweder kein Kind oder zwei Kinder hat

Ein **vollständiger Binärbaum** ist ein voller Binärbaum, in dem alle Blätter das gleiche Level haben

**Teilbaum** an Knoten  $u$ :

- lösche alle Knoten bis auf  $u$  und alle Nachfahren von  $u$   
→ der entstandene Baum mit Wurzel  $u$  heißt **Teilbaum** von  $T$  an Knoten  $u$ .

# Heaps

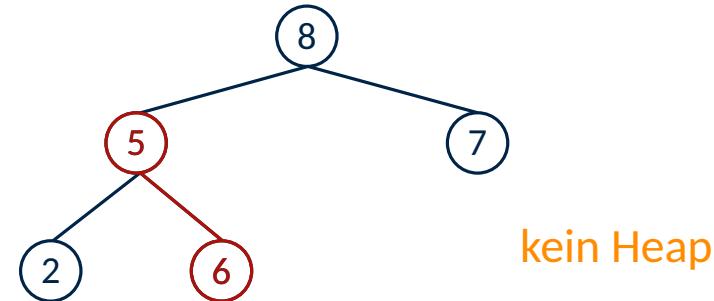
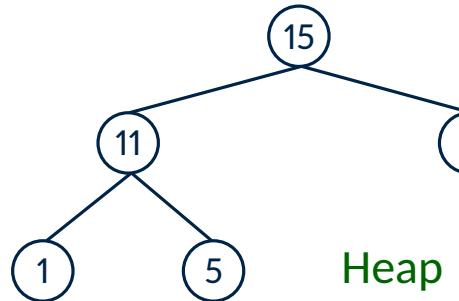
Hier definiert als max-Heap. Alternative ist min-Heap.

Unsere bisherige Datenstrukturen haben Folgen (Sequenzen) dargestellt.  
**Heaps** basieren auf einer **Baumstruktur**.

## Definition

Ein **Heap** ist ein (gerichteter) Baum, sodass:

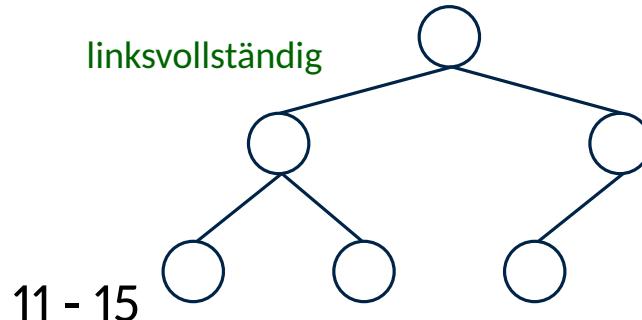
- jedem Knoten ist ein Schlüssel zugeordnet
- für jeden Kindknoten ist der Schlüssel nicht größer als der des Elternknotens



## Definition

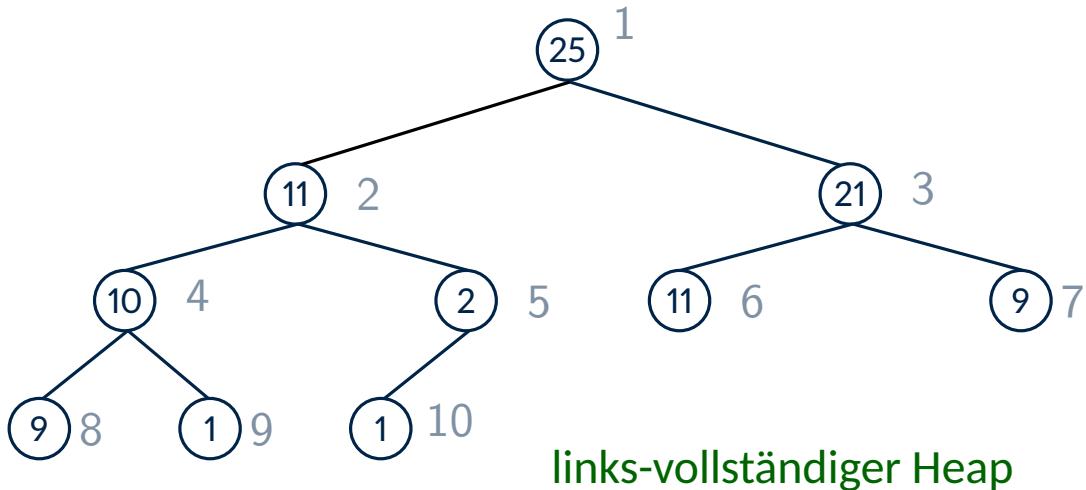
Ein geordneter binärer Baum der Höhe  $h$  heißt **links-vollständiger Baum**, wenn:

- die Knoten der Level 0 bis  $h - 1$  einen vollständigen Binärbaum bilden
- die Knoten des Level  $h$  soweit links wie möglich sind



# Implementierung links-vollständiger Heaps

Warum interessieren uns **links-vollständige Heaps**?  
Sie erlauben eine besonders einfache Darstellung als Array!



Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$

**Heap-Eigenschaft:**

$$A[\text{parent}(i)] \geq A[i] \text{ für alle } 2 \leq i \leq n$$

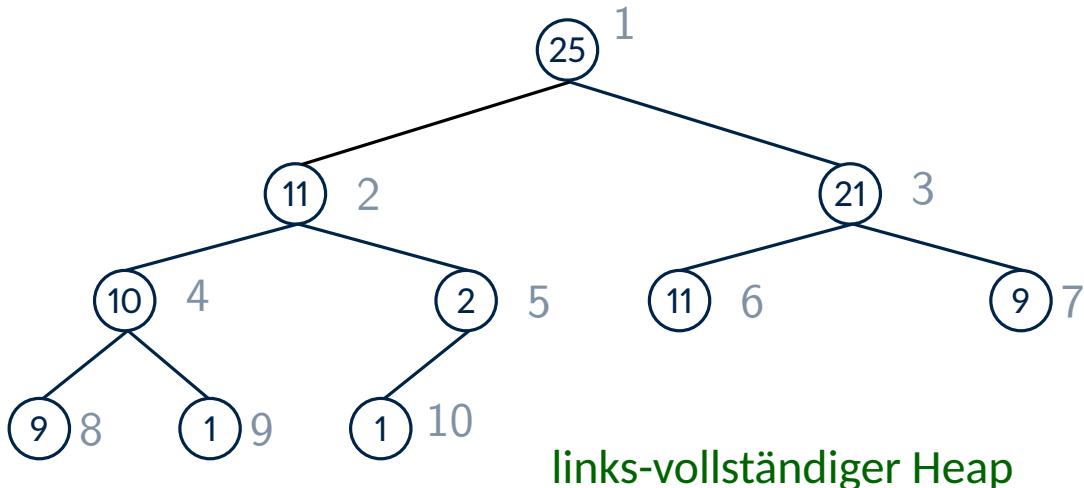
25	11	21	10	2	11	9	9	1	1
1	2	3	4	5	6	7	8	9	10

Die Knoten eines links-vollständigen Binärbaums bilden zusammenhängendes Anfangsstück des Arrays!

**Wir zeigen jetzt:**

links-vollständige Heaps in Array-Darstellung erlauben effiziente Implementierung einer Priority Queue

# Links-vollständige Heaps: Fakten



Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$

Heap-Eigenschaft:

$$A[\text{parent}(i)] \geq A[i] \text{ für alle } 2 \leq i \leq n$$

25	11	21	10	2	11	9	9	1	1
1	2	3	4	5	6	7	8	9	10

- Wichtige Fakten:**
1. Das größte Element befindet sich in der Wurzel
  2. Die Höhe des Baumes ist höchstens  $\lceil \log_2 n \rceil = O(\log n)$  **Frage:** Warum?

# Heap-Operationen I: max

## Priority Queue



Menge M von Schlüsseln

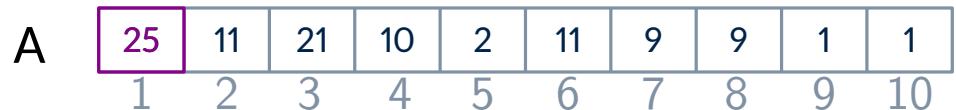
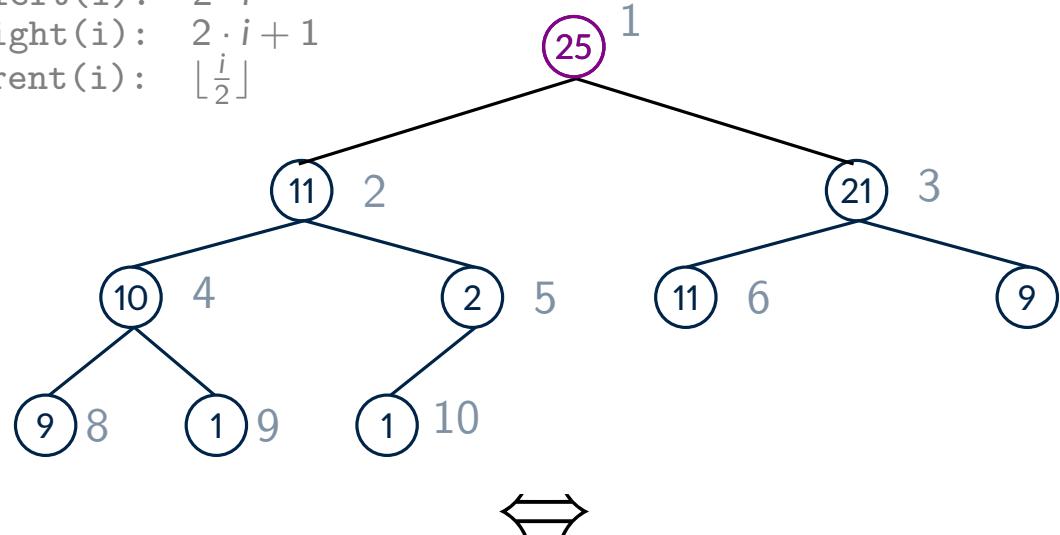
PQ.max():  
return max M

```
type max() {  
    return A[1];  
}
```

max(): 25

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



Laufzeit:  $O(1)$

Die Bestimmung des größten Elements eines linksvollständigen Heaps ist trivial.

→ Die wesentliche Aufgabe ist es also, die "linksvollständiger Heap"-Eigenschaft beizubehalten!

# Heap-Operationen II: insert

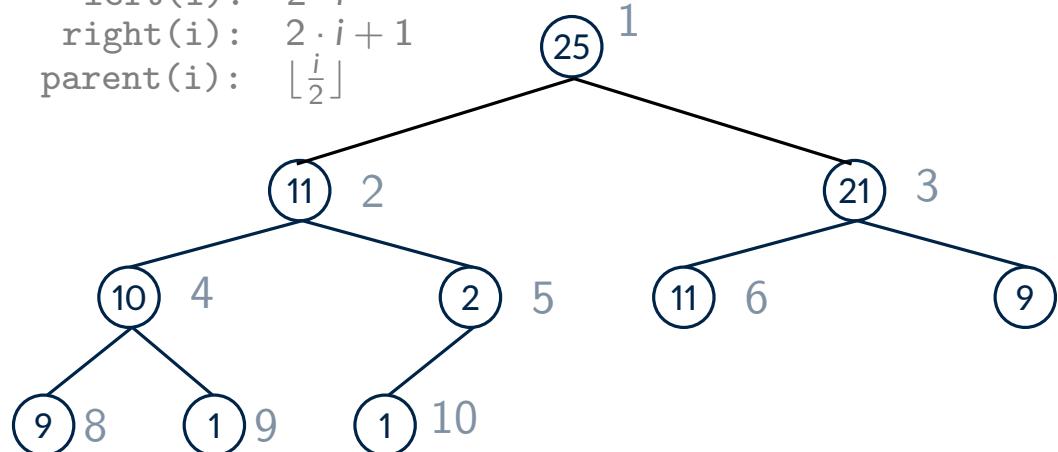
## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



insert(20)

A	25	11	21	10	2	11	9	9	1	1
	1	2	3	4	5	6	7	8	9	10

# Heap-Operationen II: insert

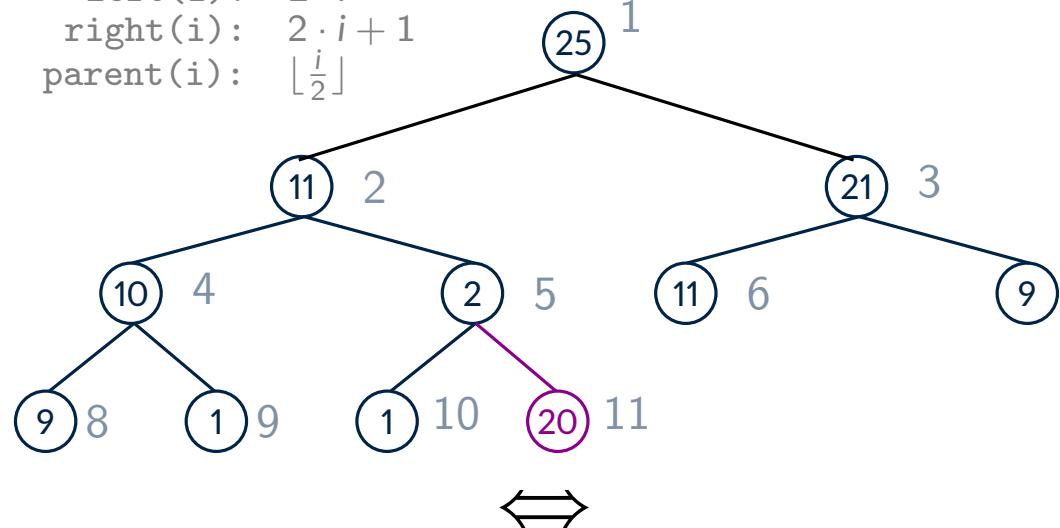
## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



insert(20)

A	25	11	21	10	2	11	9	9	1	1	20
	1	2	3	4	5	6	7	8	9	10	11

## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen II: insert

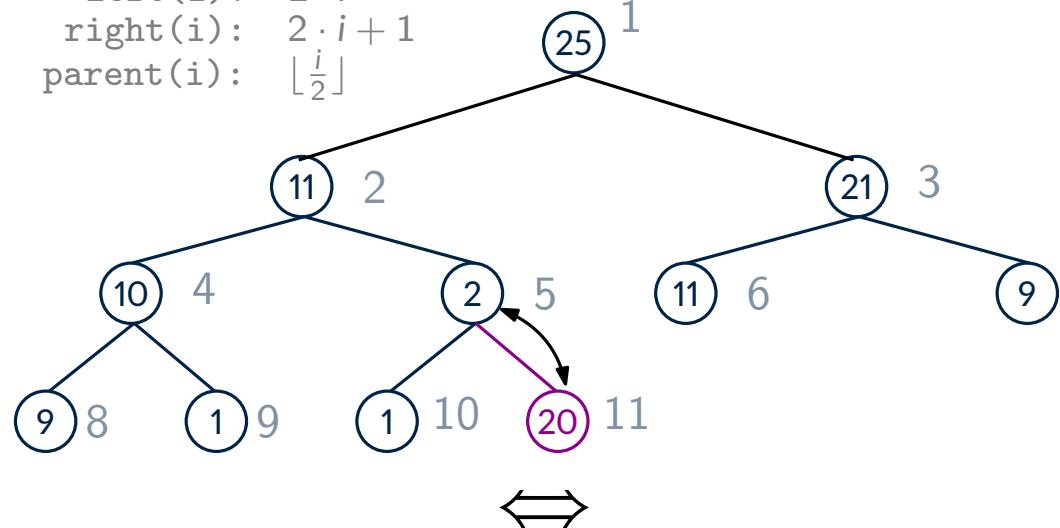
## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



insert(20)

A	25	11	21	10	2	11	9	9	1	1	20
	1	2	3	4	5	6	7	8	9	10	11

## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen II: insert

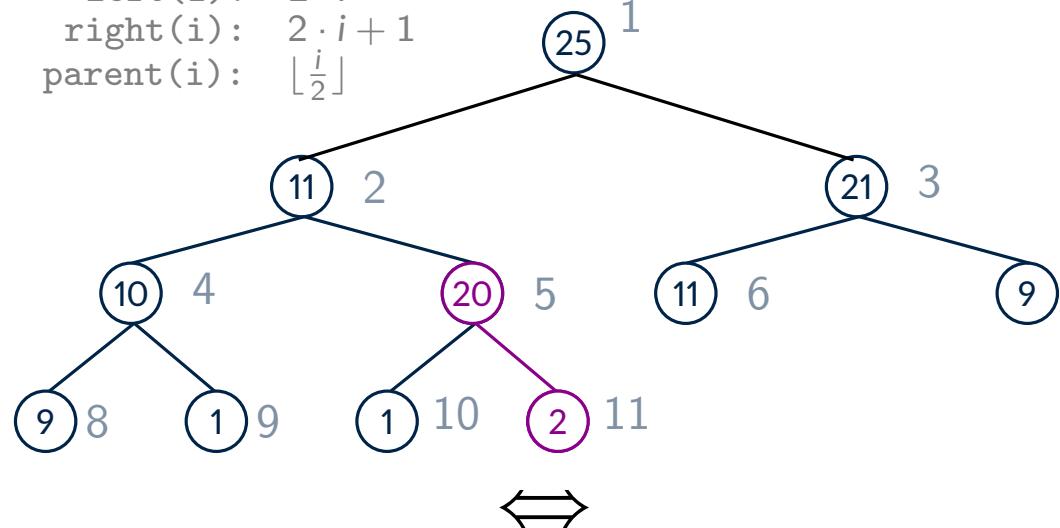
## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein
- **siebe ("sifte") das Element nach oben:**

Solange die Heap-Eigenschaft an  $i$  verletzt ist,  
vertausche Knoten  $i$  mit seinem Elternknoten

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen II: insert

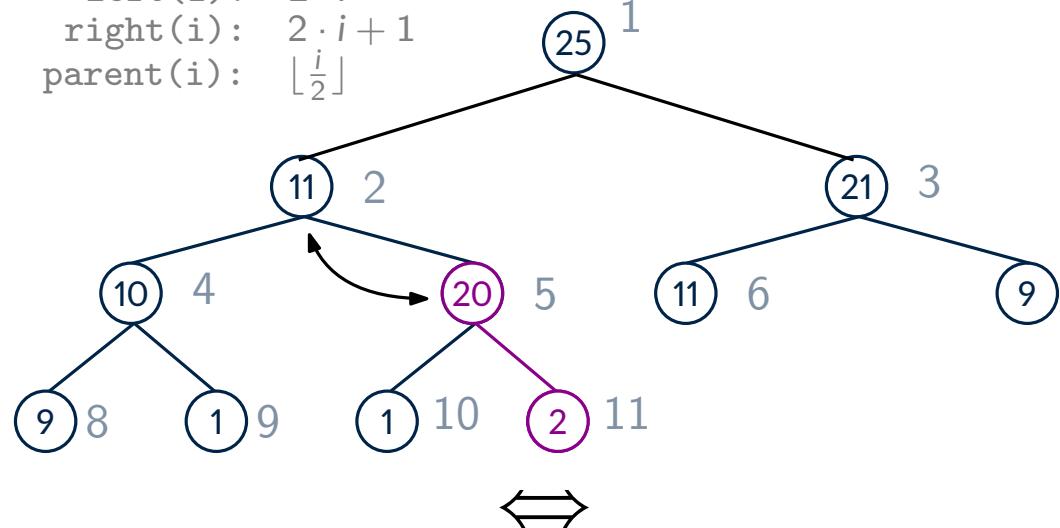
## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



A	25	11	21	10	20	11	9	9	1	1	2
	1	2	3	4	5	6	7	8	9	10	11

## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein
- **siebe ("sifte") das Element nach oben:**

Solange die Heap-Eigenschaft an  $i$  verletzt ist, vertausche Knoten  $i$  mit seinem Elternknoten und ersetze  $i$  durch den Index seines Elternknotens

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen II: insert

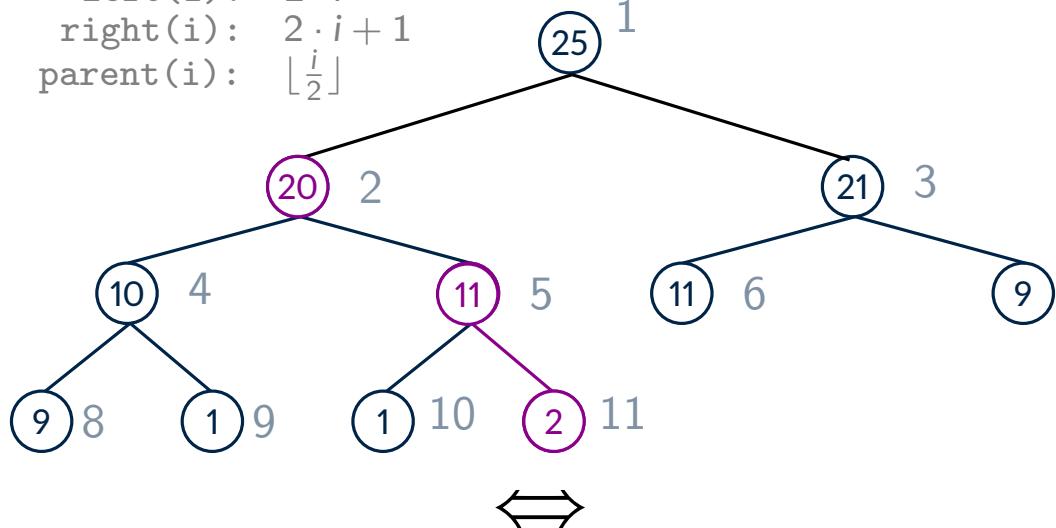
## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



A	25	20	21	10	11	11	9	9	1	1	2
	1	2	3	4	5	6	7	8	9	10	11

## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein
- **siebe ("sifte") das Element nach oben:**

Solange die Heap-Eigenschaft an  $i$  verletzt ist, vertausche Knoten  $i$  mit seinem Elternknoten und ersetze  $i$  durch den Index seines Elternknotens

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen II: insert

## Priority Queue

Menge M von Schlüsseln

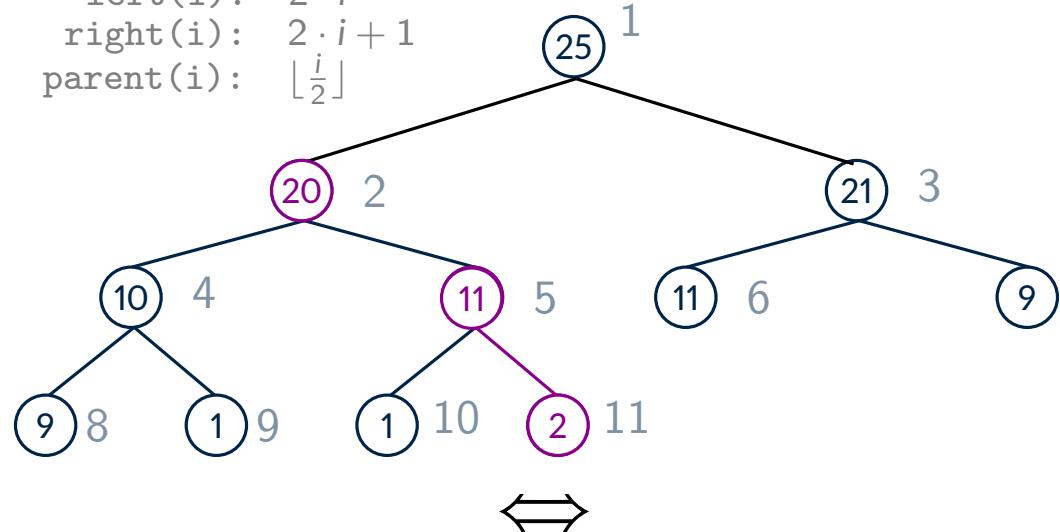
PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Laufzeit?

- die Höhe des Baumes ist höchstens  $\lceil \log_2 n \rceil$
- es gibt höchstens  $\lceil \log_2 n \rceil$  Vertauschungen
- ⇒ Laufzeit  $O(\log n)$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein
- siebe ("sifte") das Element nach oben:**  
Solange die Heap-Eigenschaft an  $i$  verletzt ist, vertausche Knoten  $i$  mit seinem Elternknoten und ersetze  $i$  durch den Index seines Elternknotens

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen II: insert

## Priority Queue

Menge M von Schlüsseln

PQ.insert( $e$ )  
 $M \leftarrow M \cup \{e\}$

Laufzeit?  $O(\log n)$  ✓

## Korrektheit?

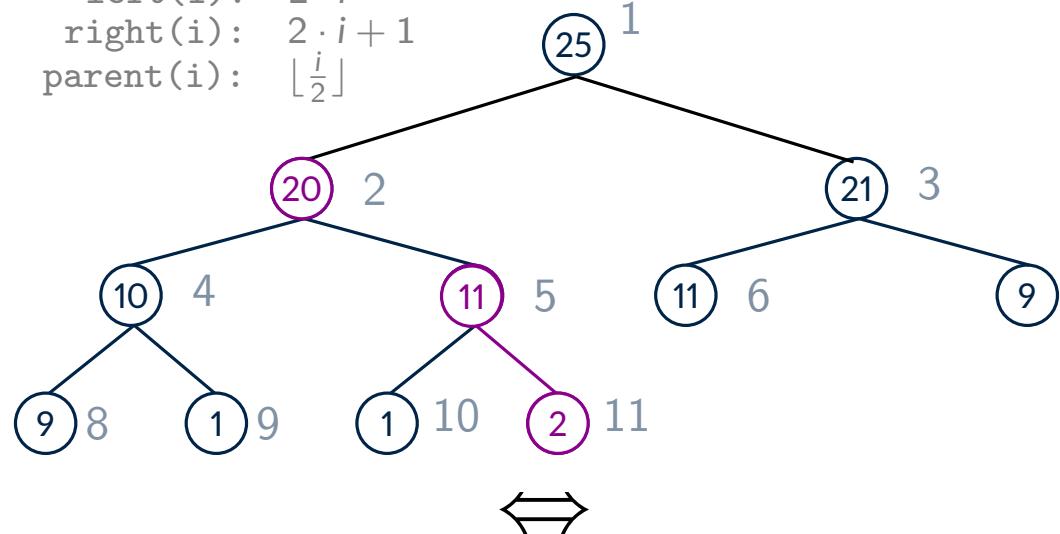
- **Invariante:** Am Anfang jeder Schleifeniteration gilt:  
Heap-Eigenschaft kann höchstens für  $i$  verletzt sein  
→ wird in der Iteration repariert, wenn nötig  
Spätestens wenn  $i$  auf die Wurzel gesetzt wird,  
gilt Heap-Eigenschaft ✓

## Algorithmenbeschreibung:

- füge  $e$  an die nächste freie Stelle  $i$  ein
- **siebe ("sifte") das Element nach oben:**  
Solange die Heap-Eigenschaft an  $i$  verletzt ist,  
vertausche Knoten  $i$  mit seinem Elternknoten  
und ersetze  $i$  durch den Index seines Elternknotens

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



A	25	20	21	10	11	11	9	9	1	1	11
	1	2	3	4	5	6	7	8	9	10	11

```
type insert(type e) {
    A.push_back(e);
    i = A.size();
    while (i>1 and A[parent(i)] < A[i]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

# Heap-Operationen III: deleteMax

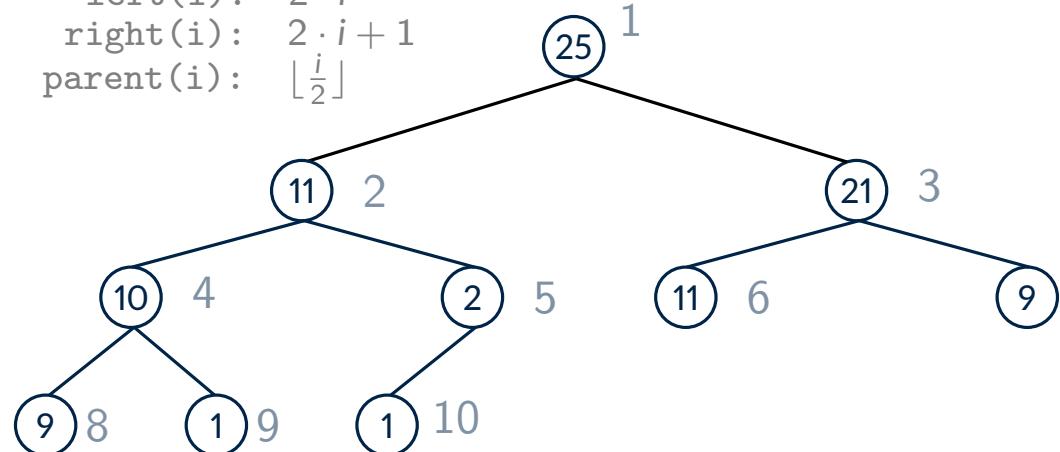
## Priority Queue

Menge M von Schlüsseln

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



deleteMax()

A	25	11	21	10	2	11	9	9	1	1
	1	2	3	4	5	6	7	8	9	10

# Heap-Operationen III: deleteMax

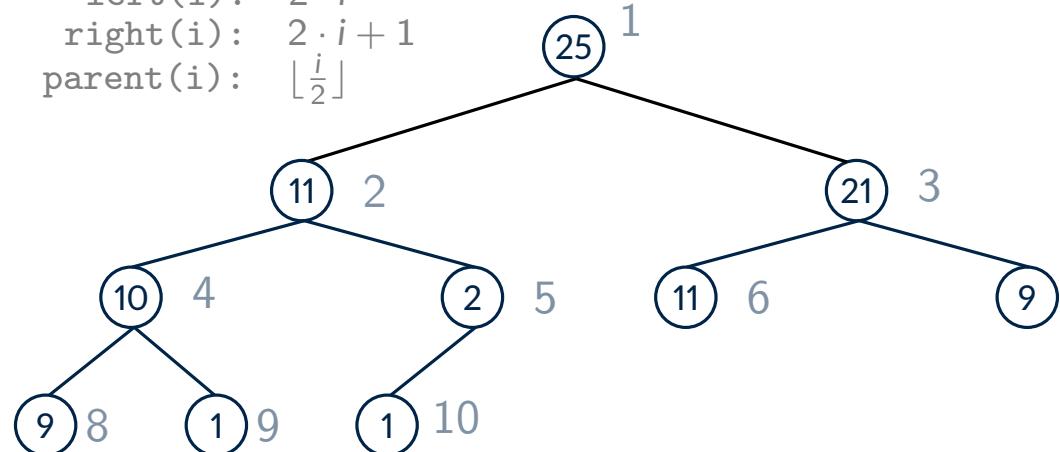
## Priority Queue

Menge M von Schlüsseln

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



A	25	11	21	10	2	11	9	9	1	1
	1	2	3	4	5	6	7	8	9	10

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
}
```

# Heap-Operationen III: deleteMax

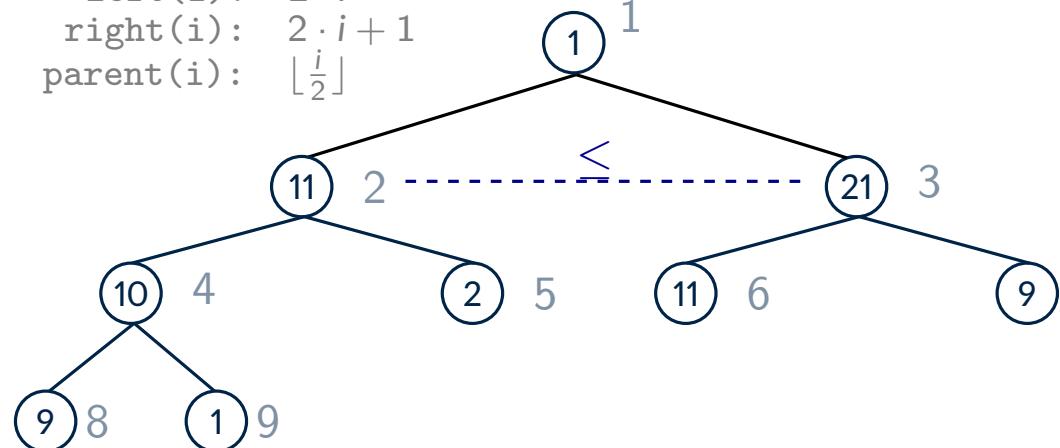
## Priority Queue

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Menge M von Schlüsseln

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



deleteMax()

A	1	11	21	10	2	11	9	9	1
	1	2	3	4	5	6	7	8	9

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:**

→ `siftDown(int i)` Pseudocode: → nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist,  
vertausche Knoten  $i$  mit dem größten Kindknoten

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

# Heap-Operationen III: deleteMax

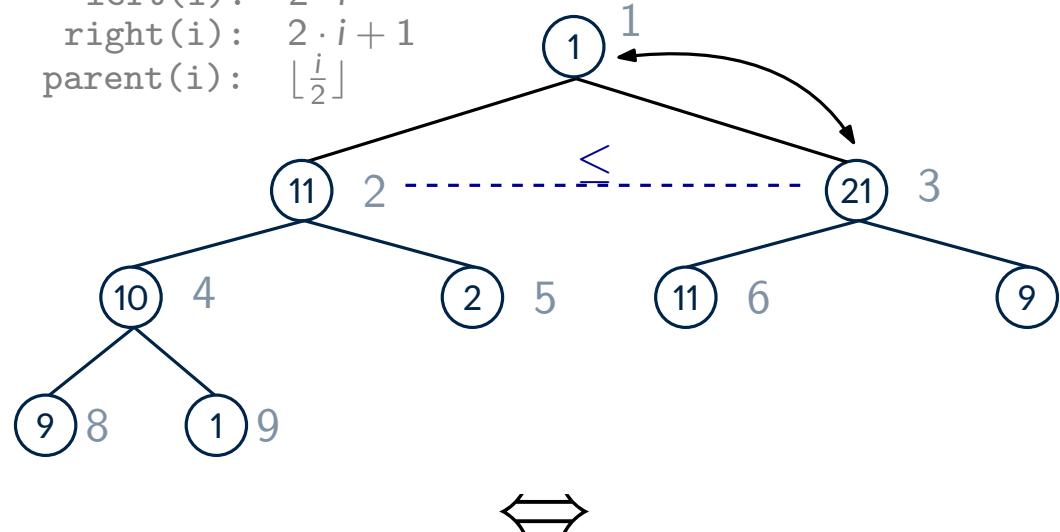
## Priority Queue

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Menge M von Schlüsseln

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



A	1	11	21	10	2	11	9	9	1
	1	2	3	4	5	6	7	8	9

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:**

→ `siftDown(int i)` Pseudocode: → nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist,  
vertausche Knoten  $i$  mit dem größten Kindknoten

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

# Heap-Operationen III: deleteMax

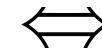
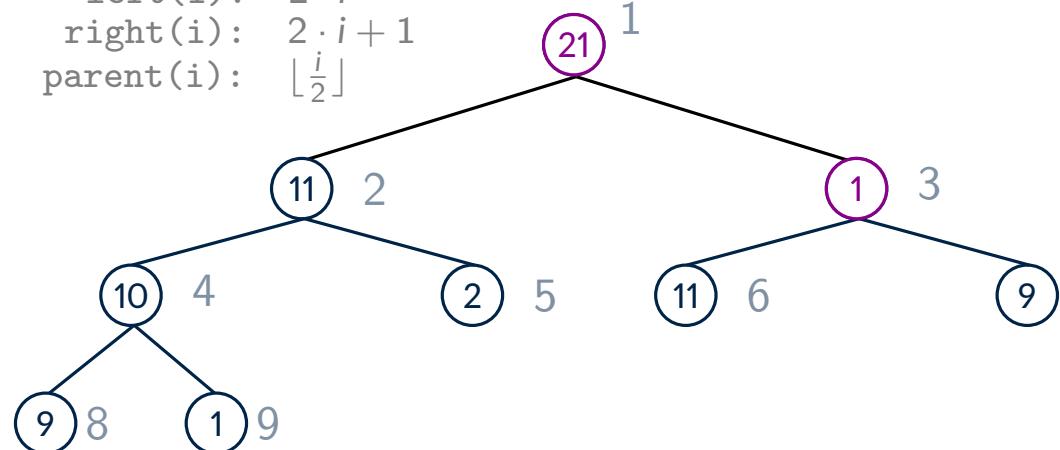
## Priority Queue

Menge M von Schlüsseln

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



deleteMax()

A	21	11	1	10	2	11	9	9	1
	1	2	3	4	5	6	7	8	9

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:**  
→ `siftDown(int i)` Pseudocode: → nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist, vertausche Knoten  $i$  mit dem größten Kindknoten und ersetze  $i$  durch den Index dieses Kindknotens

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

# Heap-Operationen III: deleteMax

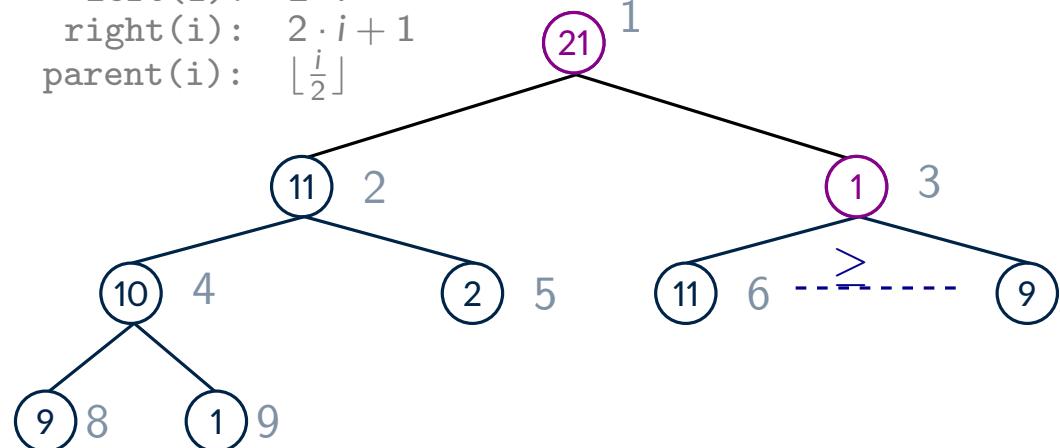
## Priority Queue

Menge M von Schlüsseln

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



deleteMax()

A	21	11	1	10	2	11	9	9	1
	1	2	3	4	5	6	7	8	9

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:**

→ `siftDown(int i)` Pseudocode: → nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist, vertausche Knoten  $i$  mit dem größten Kindknoten und ersetze  $i$  durch den Index dieses Kindknotens

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

# Heap-Operationen III: deleteMax

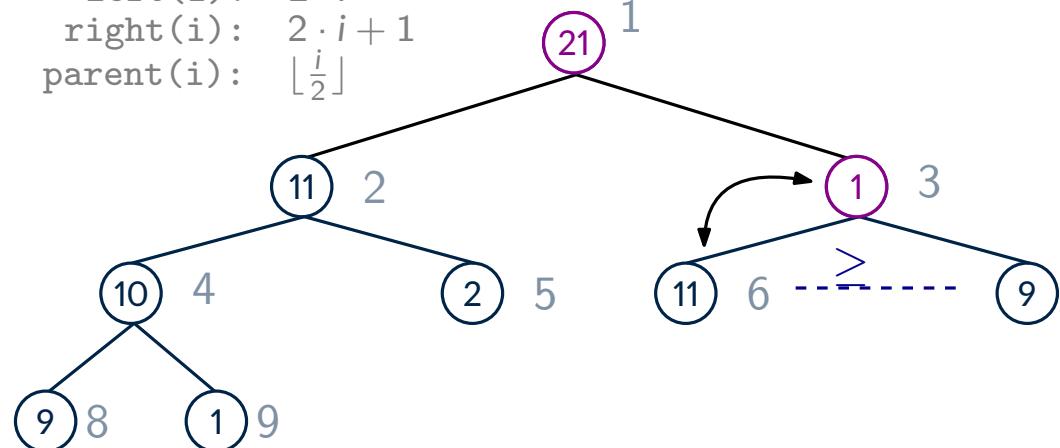
## Priority Queue

Menge M von Schlüsseln

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



deleteMax()

A	21	11	1	10	2	11	9	9	1
	1	2	3	4	5	6	7	8	9

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:**  
→ `siftDown(int i)` Pseudocode: → nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist, vertausche Knoten  $i$  mit dem größten Kindknoten und ersetze  $i$  durch den Index dieses Kindknotens

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

# Heap-Operationen III: deleteMax

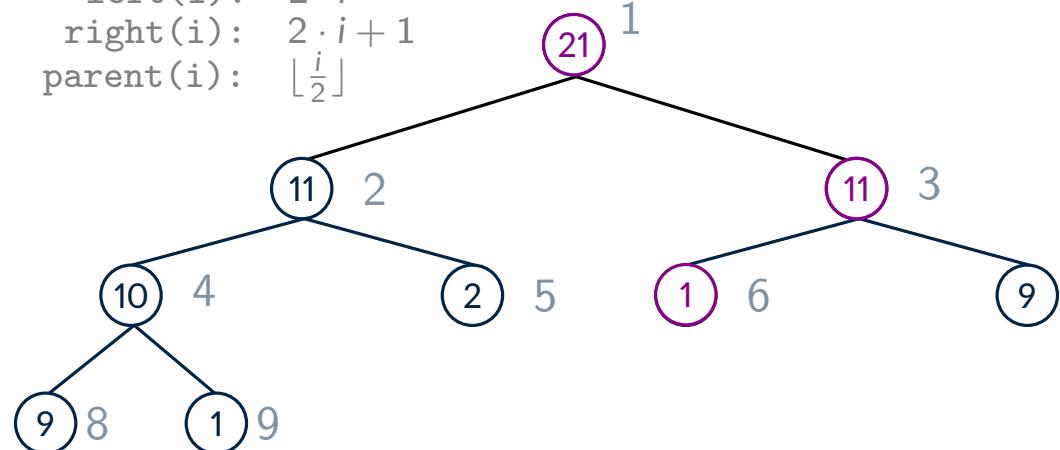
## Priority Queue

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Menge M von Schlüsseln

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned}\text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor\end{aligned}$$



deleteMax()

A	21	11	11	10	2	1	9	9	1
	1	2	3	4	5	6	7	8	9

## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:

→ `siftDown(int i)` Pseudocode: → nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist, vertausche Knoten  $i$  mit dem größten Kindknoten und ersetze  $i$  durch den Index dieses Kindknotens

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

# Heap-Operationen III: deleteMax

## Priority Queue

PQ.deleteMax()  
 $M \leftarrow M \setminus \{\max M\}$

Laufzeit?

- die Höhe des Baumes ist höchstens  $\lceil \log_2 n \rceil$
- es gibt höchstens  $\lceil \log_2 n \rceil$  Vertauschungen
- ⇒ Laufzeit  $O(\log n)$

Korrektheit? ↗ nächste Folie

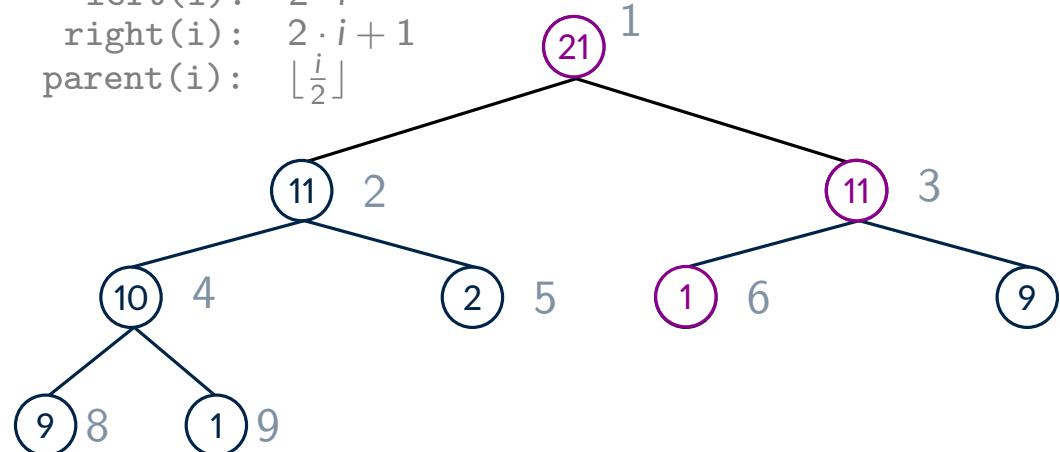
## Algorithmenbeschreibung:

- ersetze die Wurzel durch letztes Element  $A.last()$
- siebe ("sifte") das Element nach unten:**  
→ `siftDown(int i)` Pseudocode: ↗ nächste Folie

Solange die Heap-Eigenschaft an Kind von  $i$  verletzt ist, vertausche Knoten  $i$  mit dem größten Kindknoten und ersetze  $i$  durch den Index dieses Kindknotens

Für den Knoten mit Index  $i$  haben wir:

$$\begin{aligned} \text{left}(i) &: 2 \cdot i \\ \text{right}(i) &: 2 \cdot i + 1 \\ \text{parent}(i) &: \lfloor \frac{i}{2} \rfloor \end{aligned}$$



A

21	11	11	10	2	1	9	9	1
1	2	3	4	5	6	7	8	9

```
type deleteMax() {
    A[1] = A[size]
    size = size-1;
    siftdown(1);
}
```

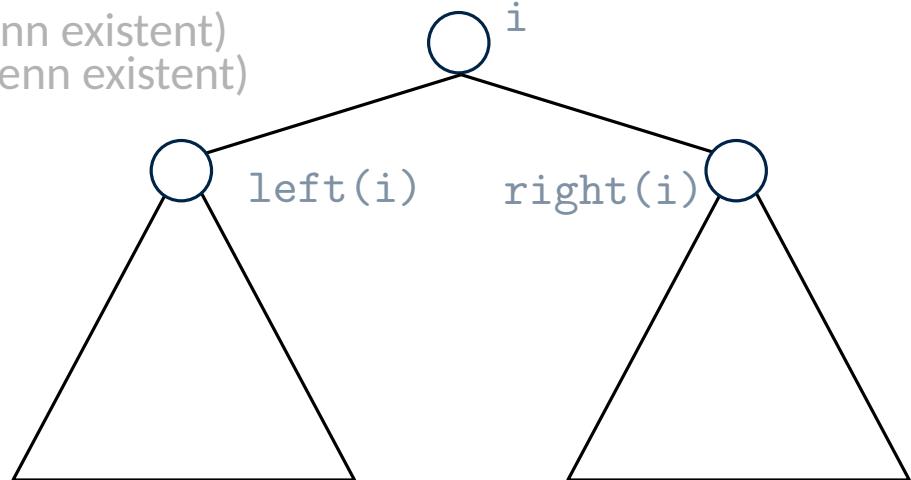
# Heap-Operationen: siftDown

## Korrektheit:

Angenommen, dass:

- der Teilbaum an  $\text{left}(i)$  erfüllt Heap-Eigenschaft (wenn existent)
- der Teilbaum an  $\text{right}(i)$  erfüllt Heap-Eigenschaft (wenn existent)

Dann erfüllt der Teilbaum an  $i$  die Heap-Eigenschaft nach Aufruf von  $\text{siftDown}(i)$ .



## Laufzeit:

$\text{siftDown}(i)$  läuft in Zeit  $O(h)$ , wobei  $h$  die Höhe des Teilbaums an Knoten  $i$  ist.

```
type siftDown(int i) {
    if (left(i) <= size) {
        if (right(i) > size or A[left(i)] >= A[right(i)]) {
            m = left(i);
        } else {
            m = right(i);
        }
        if (A[m] > A[i]) {
            swap(A[i], A[m]);
            siftDown(m);
        }
    }
}
```

**Korrektheitsbeweis** per Induktion über Höhe des Baums:

Induktionsanfang für Höhe 0 trivial.

**Induktionsschritt:**

Wenn Heap-Eigenschaft an Kind von  $i$  verletzt ist, dann vertauschen wir  $A[i]$  mit dem größten Kindschlüssel  $A[m]$

- Heap-Eigenschaft an  $\text{left}(i)$ ,  $\text{right}(i)$  erfüllt ✓  
weil Vertauschung mit größtem Kind!
- Heap-Eigenschaft kann nur an  $\text{left}(m)$ ,  $\text{right}(m)$  verletzt sein  
→ Heap-Eigenschaft wird durch  $\text{siftDown}(m)$  hergestellt □

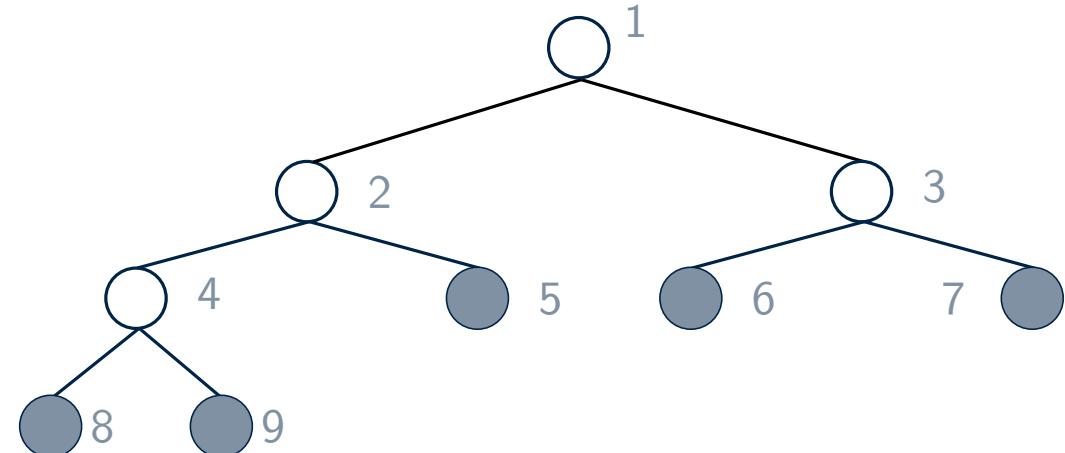
# Heap-Operationen IV: build

## Priority Queue



PQ.build( $\{e_1, \dots, e_n\}$ )  
initialisiert  $M = \{e_1, \dots, e_n\}$

Baseline:    Initialisiere leeren Heap PQ  
              for  $i = 1, \dots, n$  do  
                  PQ.insert( $e_i$ )



→ Baseline-Lösung hat Laufzeit  $O(n \log n)$ .

Können wir das besser?    Ja, mittels "Bottom-up-Ansatz"

Wir können die Heap-Eigenschaft von unten nach oben durch `siftDown(·)` herstellen.

```
void build(type A[]) {  
    for (int i = A.size() / 2; i >= 1; i--) {  
        siftDown(i);  
    }  
}
```

**Hinweis:** Indizes  $i > \lfloor \frac{n}{2} \rfloor$  sind Blätter

→ Teilbäume bestehend aus einem Blatt sind trivial Heaps

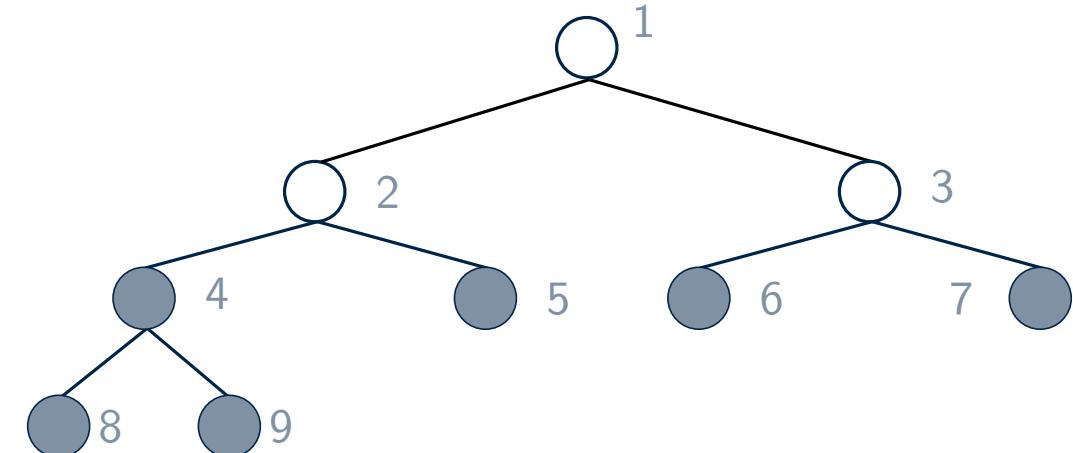
# Heap-Operationen IV: build

## Priority Queue



PQ.build( $\{e_1, \dots, e_n\}$ )  
initialisiert  $M = \{e_1, \dots, e_n\}$

Baseline:    Initialisiere leeren Heap PQ  
              for  $i = 1, \dots, n$  do  
                  PQ.insert( $e_i$ )



→ Baseline-Lösung hat Laufzeit  $O(n \log n)$ .

Können wir das besser?    Ja, mittels "Bottom-up-Ansatz"

Wir können die Heap-Eigenschaft von unten nach oben durch `siftDown(·)` herstellen.

```
void build(type A[]) {  
    for (int i = A.size() / 2; i >= 1; i--) {  
        siftDown(i);  
    }  
}
```

**Hinweis:** Indizes  $i > \lfloor \frac{n}{2} \rfloor$  sind Blätter

→ Teilbäume bestehend aus einem Blatt sind trivial Heaps

# Heap-Operationen IV: build

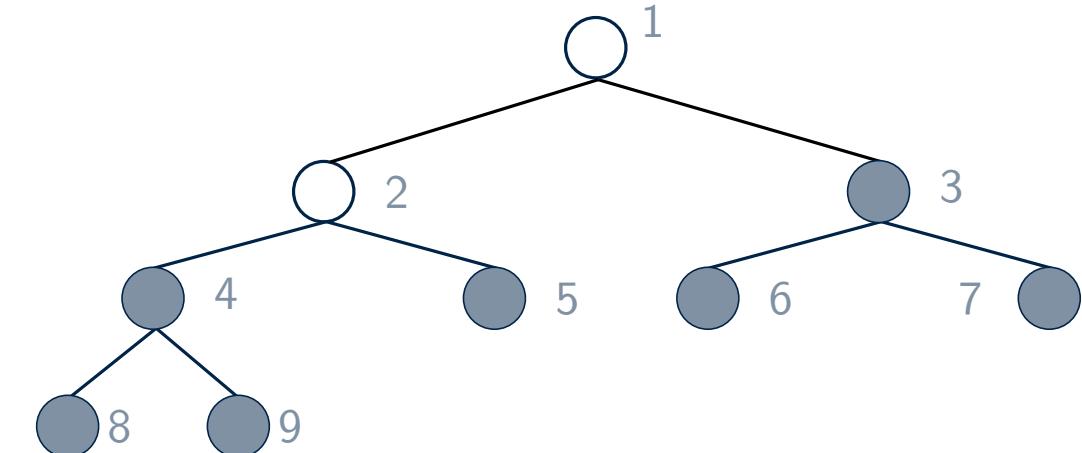
## Priority Queue



Menge  $M$  von Schlüsseln

PQ.build( $\{e_1, \dots, e_n\}$ )  
initialisiert  $M = \{e_1, \dots, e_n\}$

Baseline:    Initialisiere leeren Heap PQ  
              for  $i = 1, \dots, n$  do  
                  PQ.insert( $e_i$ )



→ Baseline-Lösung hat Laufzeit  $O(n \log n)$ .

Können wir das besser?    Ja, mittels "Bottom-up-Ansatz"

Wir können die Heap-Eigenschaft von unten nach oben durch `siftDown(·)` herstellen.

```
void build(type A[]) {  
    for (int i = A.size() / 2; i >= 1; i--) {  
        siftDown(i);  
    }  
}
```

**Hinweis:** Indizes  $i > \lfloor \frac{n}{2} \rfloor$  sind Blätter

→ Teilbäume bestehend aus einem Blatt sind trivial Heaps

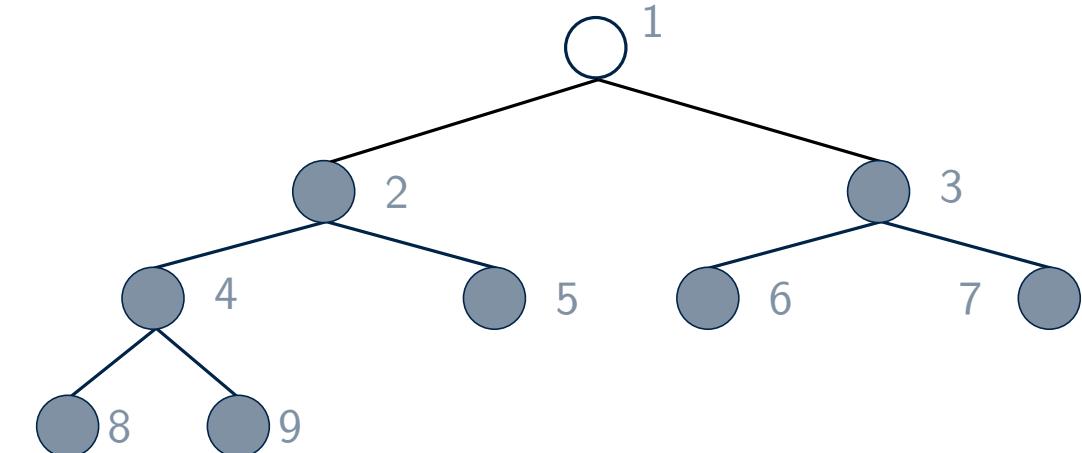
# Heap-Operationen IV: build

## Priority Queue



PQ.build( $\{e_1, \dots, e_n\}$ )  
initialisiert  $M = \{e_1, \dots, e_n\}$

Baseline:    Initialisiere leeren Heap PQ  
              for  $i = 1, \dots, n$  do  
                  PQ.insert( $e_i$ )



→ Baseline-Lösung hat Laufzeit  $O(n \log n)$ .

Können wir das besser?    Ja, mittels "Bottom-up-Ansatz"

Wir können die Heap-Eigenschaft von unten nach oben durch `siftDown(·)` herstellen.

```
void build(type A[]) {  
    for (int i = A.size() / 2; i >= 1; i--) {  
        siftDown(i);  
    }  
}
```

**Hinweis:** Indizes  $i > \lfloor \frac{n}{2} \rfloor$  sind Blätter

→ Teilbäume bestehend aus einem Blatt sind trivial Heaps

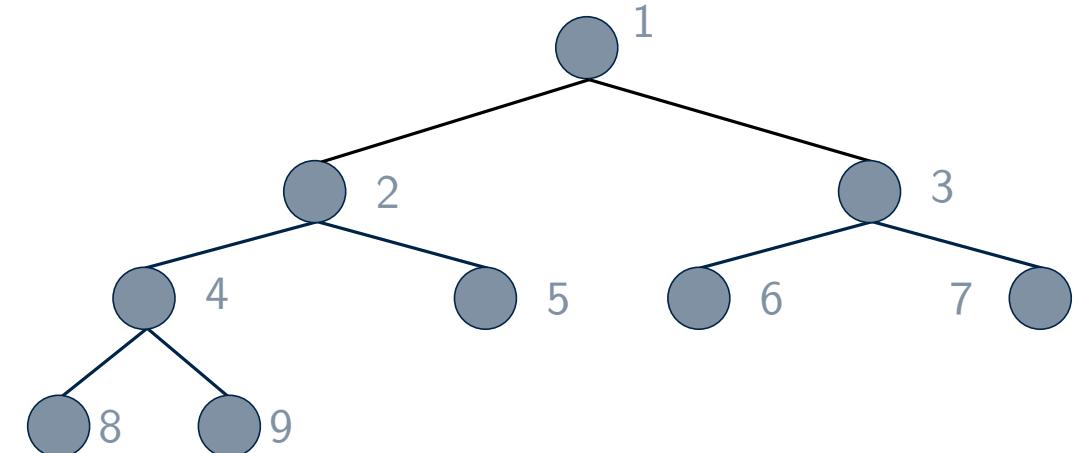
# Heap-Operationen IV: build

## Priority Queue



PQ.build( $\{e_1, \dots, e_n\}$ )  
initialisiert  $M = \{e_1, \dots, e_n\}$

Baseline:    Initialisiere leeren Heap PQ  
              for  $i = 1, \dots, n$  do  
                  PQ.insert( $e_i$ )



→ Baseline-Lösung hat Laufzeit  $O(n \log n)$ .

Können wir das besser?    Ja, mittels "Bottom-up-Ansatz"

Wir können die Heap-Eigenschaft von unten nach oben durch `siftDown(·)` herstellen.

```
void build(type A[]) {  
    for (int i = A.size() / 2; i >= 1; i--) {  
        siftDown(i);  
    }  
}
```

**Hinweis:** Indizes  $i > \lfloor \frac{n}{2} \rfloor$  sind Blätter

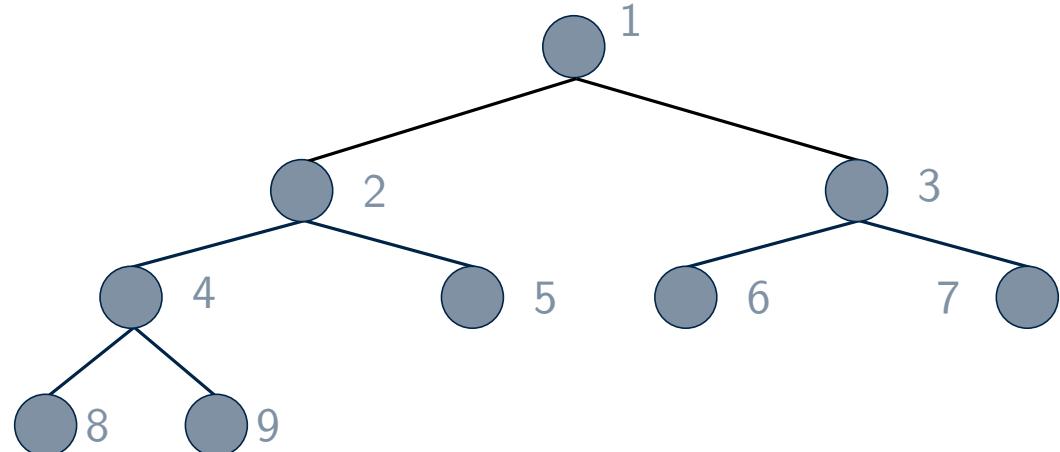
→ Teilbäume bestehend aus einem Blatt sind trivial Heaps

# Heap-Operationen IV: build

## Priority Queue



PQ.build( $\{e_1, \dots, e_n\}$ )  
initialisiert  $M = \{e_1, \dots, e_n\}$



Baseline:    Initialisiere leeren Heap PQ  
for  $i = 1, \dots, n$  do  
    PQ.insert( $e_i$ )

→ Baseline-Lösung hat Laufzeit  $O(n \log n)$ .

Können wir das besser?    Ja, mittels "Bottom-up-Ansatz"

Wir können die Heap-Eigenschaft von unten nach oben durch `siftDown(·)` herstellen.

```
void build(type A[]) {  
    for (int i = A.size() / 2; i >= 1; i--) {  
        siftDown(i);  
    }  
}
```

**Hinweis:** Indizes  $i > \lfloor \frac{n}{2} \rfloor$  sind Blätter

→ Teilbäume bestehend aus einem Blatt sind trivial Heaps

Laufzeitanalyse ist analog zu amortisierter Analyse von Zahl inkrementieren!

· für  $h = 0, \dots, \lfloor \log_2 n \rfloor$  gibt es  $\leq \lceil \frac{n}{2^{h+1}} \rceil$  Teilbäume der Höhe  $h$

⇒ Gesamlaufzeit ist höchstens  $O(\sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot h) \subseteq O(n(\sum_{h=0}^{\infty} \frac{h}{2^{h+1}})) \subseteq O(n)$

# Priority Queues via Heaps

---

## Theorem

Unsere Heaps implementieren eine Priority Queue mit den folgenden Zugriffszeiten:

- `PQ.build({e1, ..., en})` läuft in Zeit  $O(n)$ .
- `PQ.insert(e)` läuft in Zeit  $O(\log n)$ .
- `PQ.max()` läuft in Zeit  $O(1)$ .
- `PQ.deleteMax()` läuft in Zeit  $O(\log n)$ .

F: Könnten wir Heaps als Datenstruktur verbessern?  
↗ Übung

## Beispielanwendung: HeapSort

`HeapSort(A[1 ... n]):`

```
    Priority Queue PQ = build(A)
    Initialisiere A'[1 ... n]
    for i = n, ..., 1 do
        | A'[i] = PQ.max()
        | PQ.deleteMax()
    return A'[1 ... n]
```

F: Wie beweist man die Korrektheit?

F: Was ist die Laufzeit?

→ Ein weiteres vergleichsbasiertes Sortierverfahren

# Adressierbare Priority Queues

---

Wir betrachten Erweiterung zu einer **adressierbaren Priority Queue**:

## adressierbare Priority Queue:

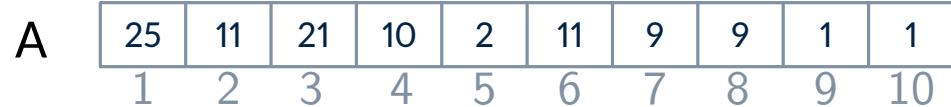
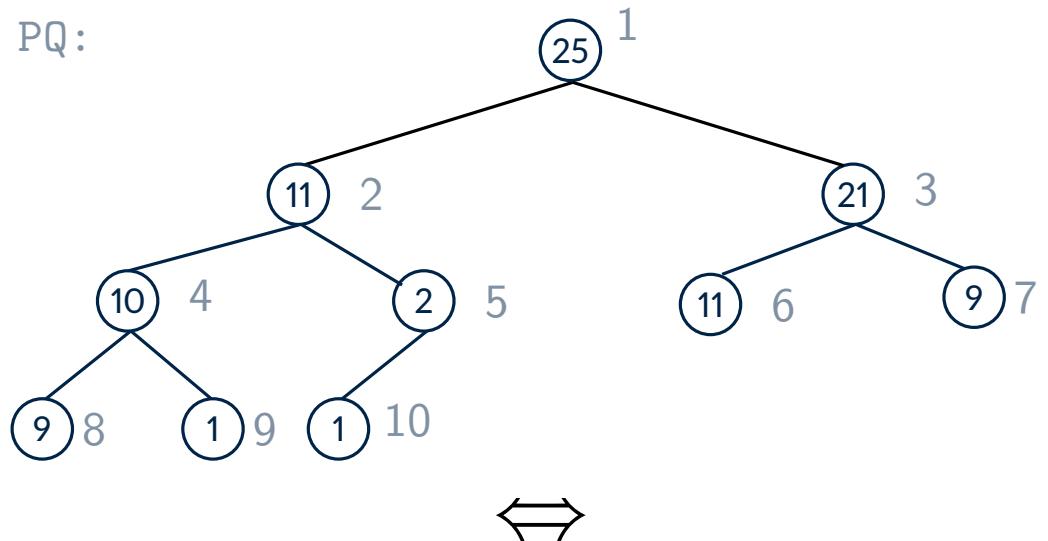
PQ.build( $\{e_1, \dots, e_n\}$ )	initialisiert $M = \{e_1, \dots, e_n\}$
PQ.insert( $e$ )	fügt $e$ in $M$ ein, d.h. $M \leftarrow M \cup \{e\}$ und gibt Zeiger $h$ auf das eingefügte Element zurück
PQ.max()	return max $M$
PQ.deleteMax()	löscht max $M$ , d.h. $M \leftarrow M \setminus \{\max M\}$
PQ.remove( $h$ )	löscht das Element aus $M$ , auf das der Zeiger $h$ zeigt
PQ.increaseKey( $h, k$ )	vergrößert den Schlüssel des Elements, auf das der Zeiger $h$ zeigt, auf Wert $k$
PQ.merge(PQ2)	vereinigt Priority Queues PQ und PQ2 $M \leftarrow M \cup M'$ , $M' \leftarrow \emptyset$ $M'$ ist die von PQ2 repräsentierte Schlüsselmenge

Wie schnell können wir diese Operationen implementieren? ↗ Übung + spätere VL

# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

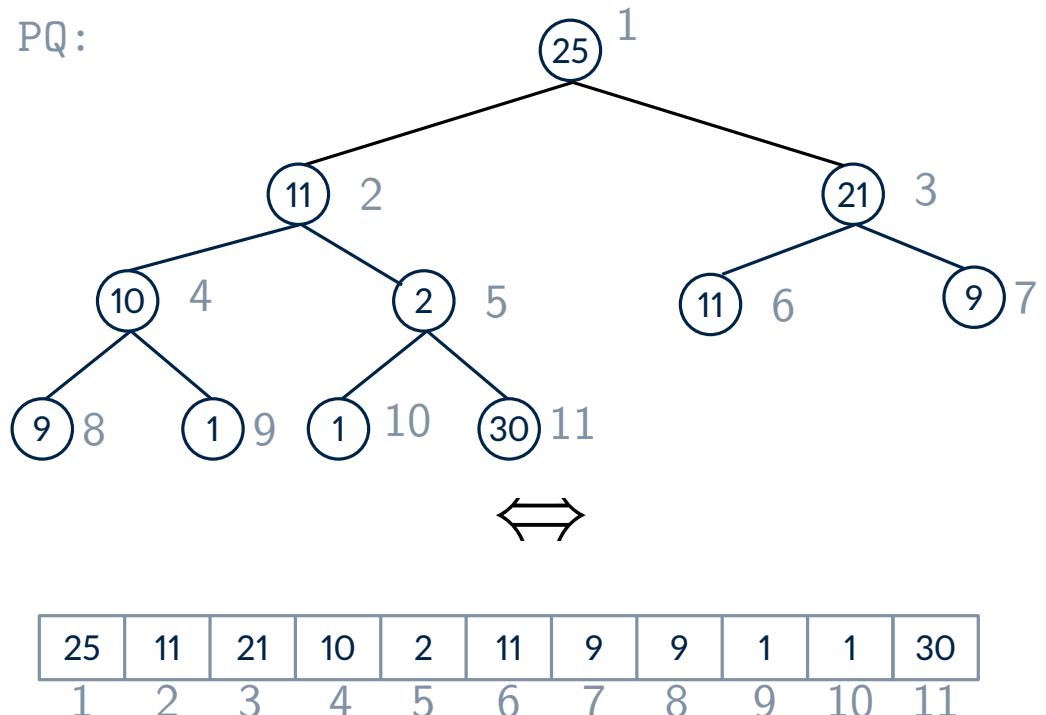
h1 = PQ.insert(30)



# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

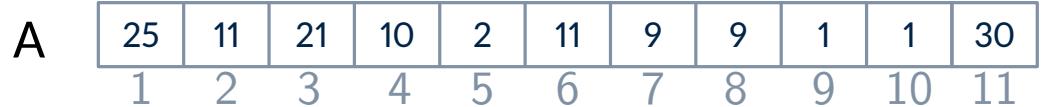
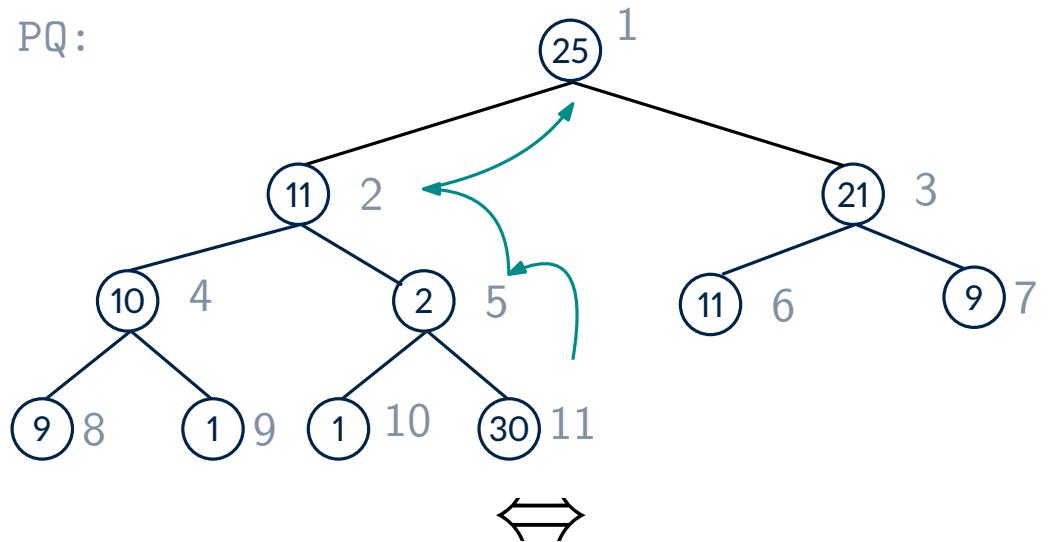
h1 = PQ.insert(30)



# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

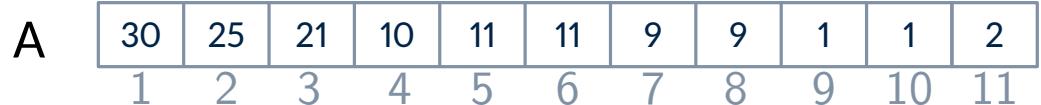
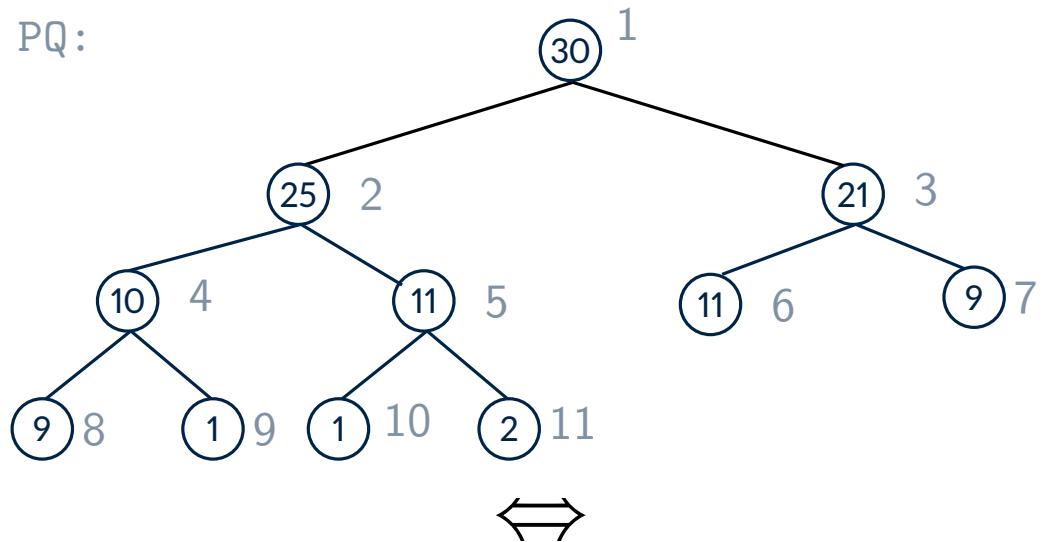
h1 = PQ.insert(30)



# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

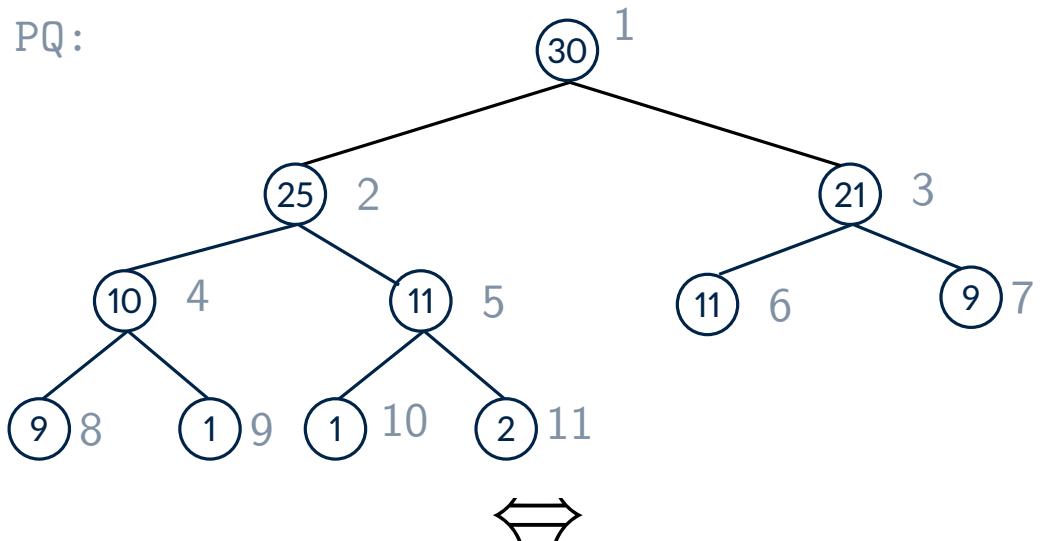
h1 = PQ.insert(30)



# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

h1 = PQ.insert(30)



A

30	25	21	10	11	11	9	9	1	1	2
1	2	3	4	5	6	7	8	9	10	11

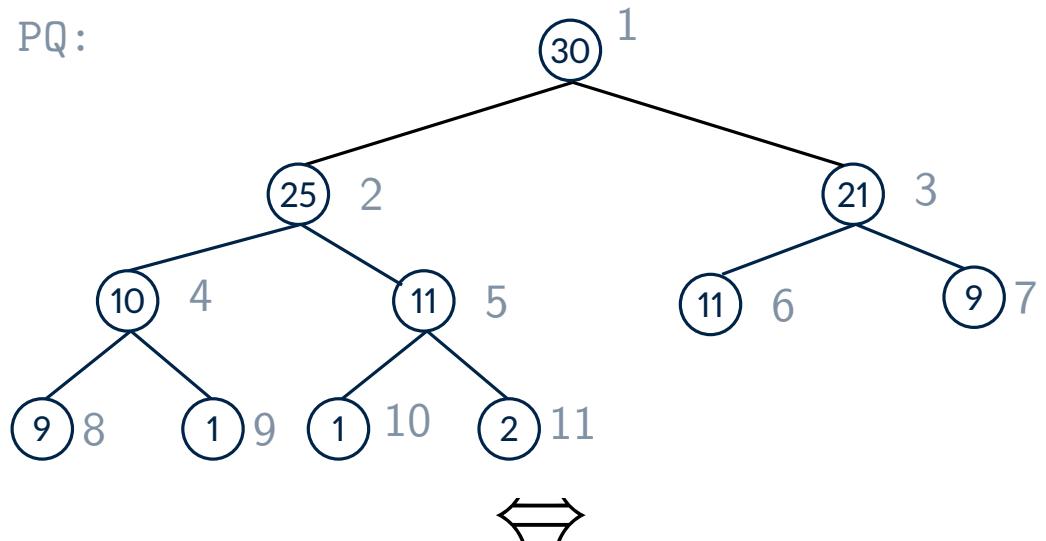
h1: 1

# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```



A

	30	25	21	10	11	11	9	9	1	1	2
1	2	3	4	5	6	7	8	9	10	11	

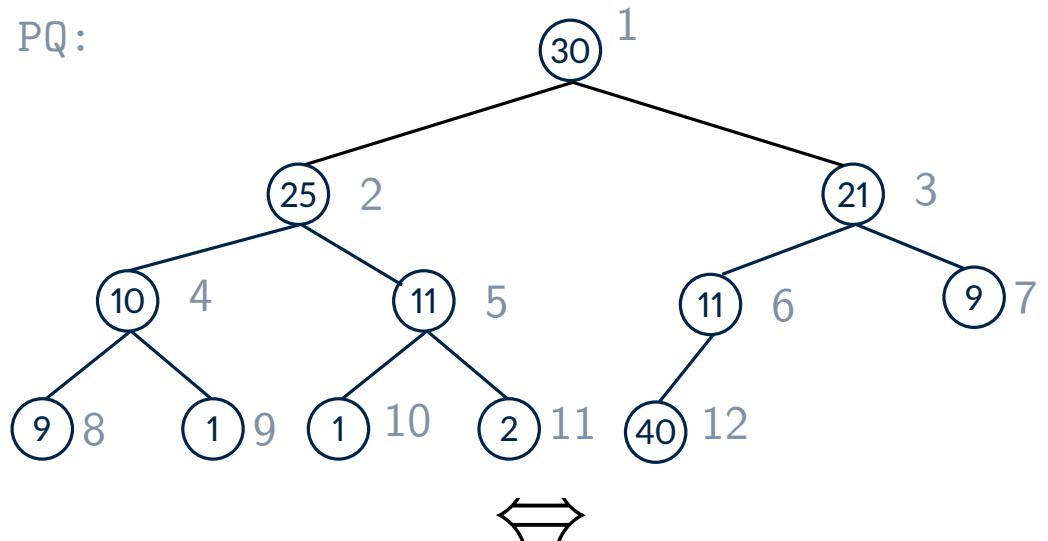
h1: 1

# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```



A

30	25	21	10	11	11	9	9	1	1	2	40
1	2	3	4	5	6	7	8	9	10	11	12

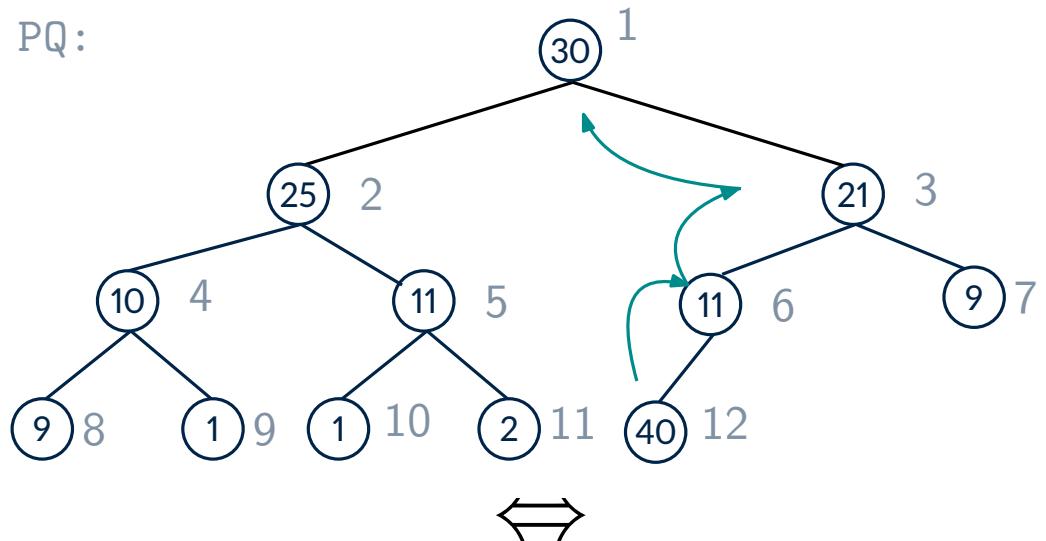
h1: 1

# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```



A

30	25	21	10	11	11	9	9	1	1	2	10	11	40
1	2	3	4	5	6	7	8	9	10	11	12		

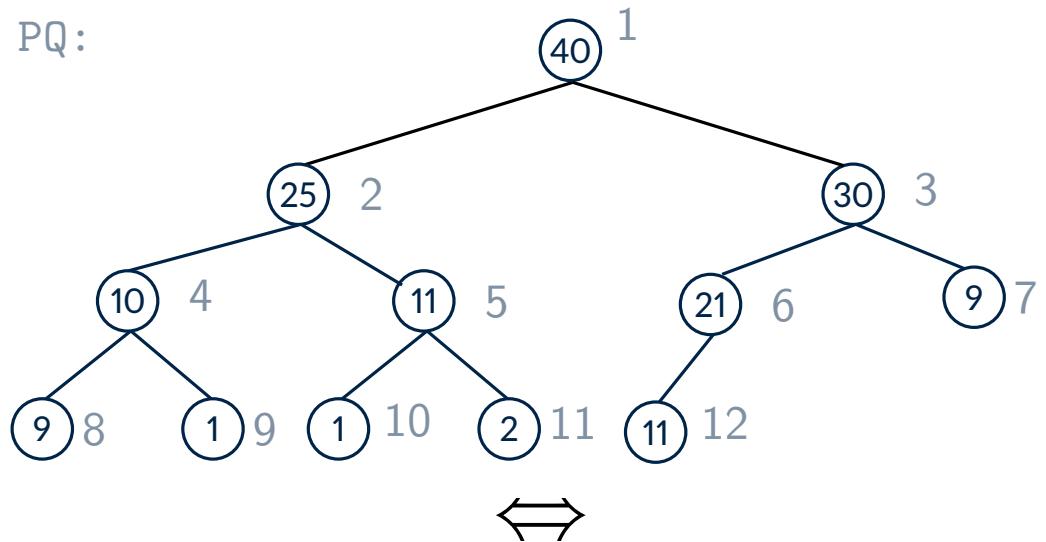
h1: 1

# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```



A

	40	25	30	10	11	21	9	9	1	1	2	11	12
1	2	3	4	5	6	7	8	9	10	11	12		

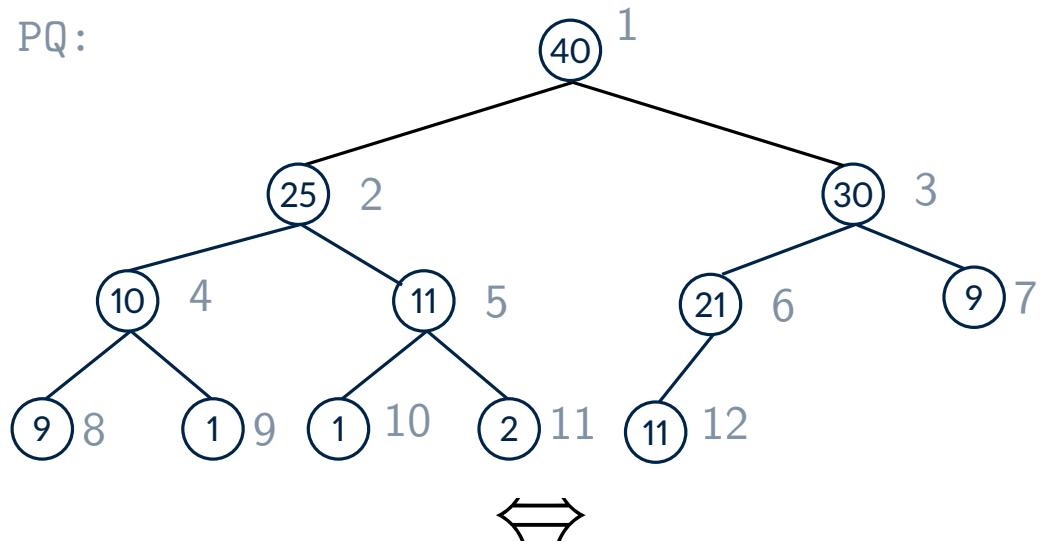
h1: 1

# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```



A

40	25	30	10	11	21	9	9	1	1	2	11	12
1	2	3	4	5	6	7	8	9	10	11	12	

h1: 1

h2: 1

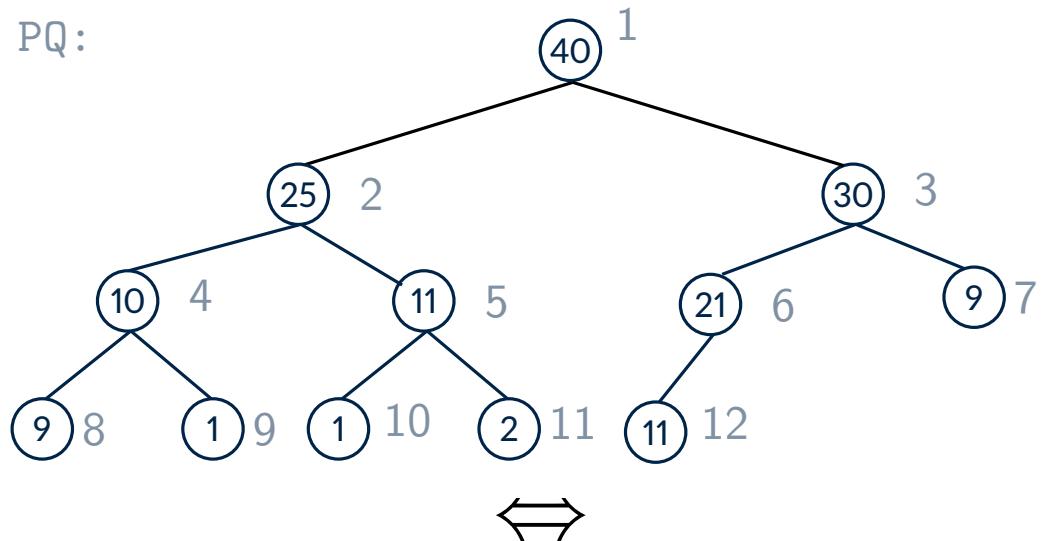
# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```

```
PQ.remove(h1)
```



A	40	25	30	10	11	21	9	9	1	1	2	11	12
	1	2	3	4	5	6	7	8	9	10	11	12	

h1: 1

h2: 1

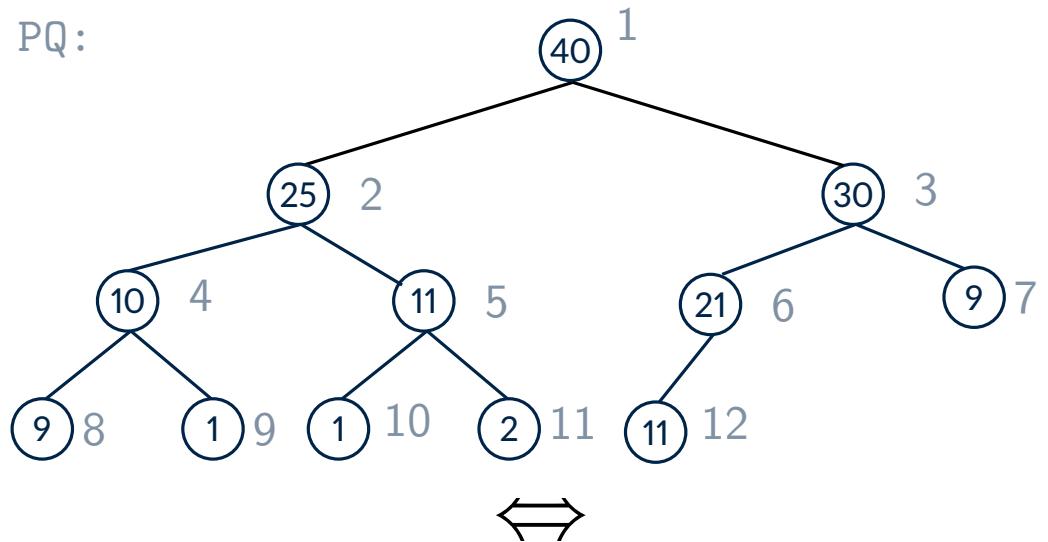
# Das Problem "ungültiger" Adressen...

Frage: Warum reicht es nicht, den Index im Array als Adresse zu nehmen?

```
h1 = PQ.insert(30)
```

```
h2 = PQ.insert(40)
```

```
PQ.remove(h1)
```



A	40	25	30	10	11	21	9	9	1	1	2	11	12
	1	2	3	4	5	6	7	8	9	10	11	12	

h1: 1 → tatsächlicher Index vom Schlüssel 30 wäre inzwischen 3

h2: 1

**Problem:** remove(h1) löscht nicht den Schlüssel 30, sondern 40

→ Adresse h1 ist zwischendurch ungültig geworden...

# Adressierbare Priority Queues

---

Wir betrachten Erweiterung zu einer **adressierbaren Priority Queue**:

## adressierbare Priority Queue:

PQ.build( $\{e_1, \dots, e_n\}$ )	initialisiert $M = \{e_1, \dots, e_n\}$
PQ.insert( $e$ )	fügt $e$ in $M$ ein, d.h. $M \leftarrow M \cup \{e\}$ und gibt Zeiger $h$ auf das eingefügte Element zurück
PQ.max()	return max $M$
PQ.deleteMax()	löscht max $M$ , d.h. $M \leftarrow M \setminus \{\max M\}$
PQ.remove( $h$ )	löscht das Element aus $M$ , auf das der Zeiger $h$ zeigt
PQ.increaseKey( $h, k$ )	vergrößert den Schlüssel des Elements, auf das der Zeiger $h$ zeigt, auf Wert $k$
PQ.merge(PQ2)	vereinigt Priority Queues PQ und PQ2 $M \leftarrow M \cup M'$ , $M' \leftarrow \emptyset$ $M'$ ist die von PQ2 repräsentierte Schlüsselmenge

Wie schnell können wir diese Operationen implementieren? ↗ Übung + spätere VL

# Adressierbare Priority Queues

Wir betrachten Erweiterung zu einer **adressierbaren Priority Queue**:

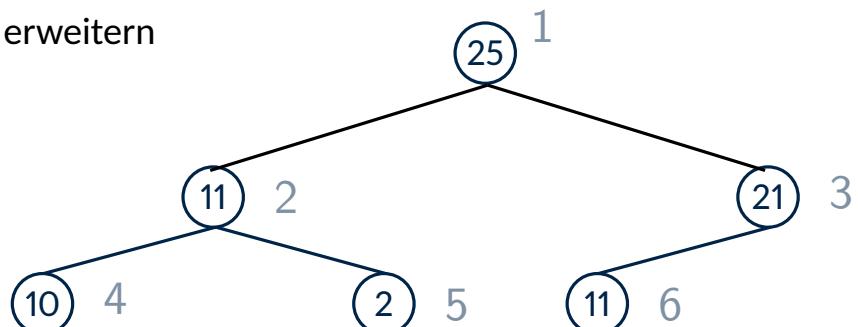
## adressierbare Priority Queue:

PQ.build( $\{e_1, \dots, e_n\}$ )	initialisiert $M = \{e_1, \dots, e_n\}$
PQ.insert( $e$ )	fügt $e$ in $M$ ein, d.h. $M \leftarrow M \cup \{e\}$ und gibt Zeiger $h$ auf das eingefügte Element zurück
PQ.max()	return max $M$
PQ.deleteMax()	löscht max $M$ , d.h. $M \leftarrow M \setminus \{\max M\}$
PQ.remove( $h$ )	löscht das Element aus $M$ , auf das der Zeiger $h$ zeigt
PQ.increaseKey( $h, k$ )	vergrößert den Schlüssel des Elements, auf das der Zeiger $h$ zeigt, auf Wert $k$
PQ.merge(PQ2)	vereinigt Priority Queues PQ und PQ2 $M \leftarrow M \cup M'$ , $M' \leftarrow \emptyset$ $M'$ ist die von PQ2 repräsentierte Schlüsselmenge

Wie schnell können wir diese Operationen implementieren? ↗ Übung + spätere VL

**Tipp:** Man kann unsere Priority Queue um weitere Daten erweitern

A	25	11	21	10	2	11
	1	2	3	4	5	6



# Adressierbare Priority Queues

Wir betrachten Erweiterung zu einer **adressierbaren Priority Queue**:

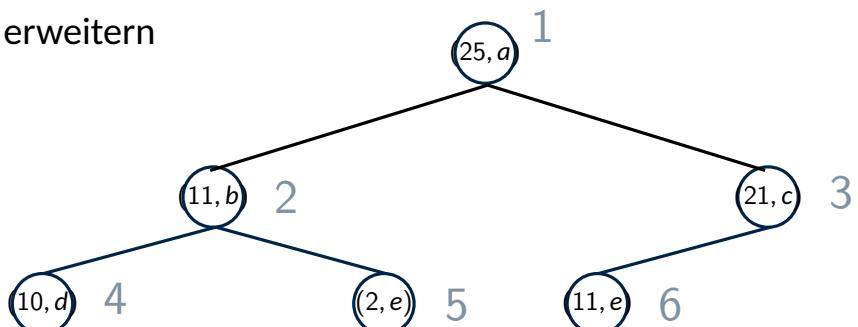
## adressierbare Priority Queue:

PQ.build( $\{e_1, \dots, e_n\}$ )	initialisiert $M = \{e_1, \dots, e_n\}$
PQ.insert( $e$ )	fügt $e$ in $M$ ein, d.h. $M \leftarrow M \cup \{e\}$ und gibt Zeiger $h$ auf das eingefügte Element zurück
PQ.max()	return max $M$
PQ.deleteMax()	löscht max $M$ , d.h. $M \leftarrow M \setminus \{\max M\}$
PQ.remove( $h$ )	löscht das Element aus $M$ , auf das der Zeiger $h$ zeigt
PQ.increaseKey( $h, k$ )	vergrößert den Schlüssel des Elements, auf das der Zeiger $h$ zeigt, auf Wert $k$
PQ.merge(PQ2)	vereinigt Priority Queues PQ und PQ2 $M \leftarrow M \cup M'$ , $M' \leftarrow \emptyset$ $M'$ ist die von PQ2 repräsentierte Schlüsselmenge

Wie schnell können wir diese Operationen implementieren? ↗ Übung + spätere VL

**Tipp:** Man kann unsere Priority Queue um weitere Daten erweitern

A	(25, a)	(11, b)	(21, c)	(10, d)	(2, e)	(11, e)
	1	2	3	4	5	6



# Adressierbare Priority Queues

Wir betrachten Erweiterung zu einer **adressierbaren Priority Queue**:

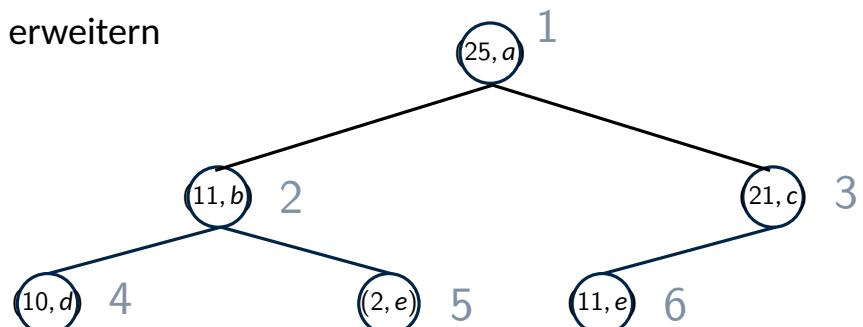
## adressierbare Priority Queue:

PQ.build( $\{e_1, \dots, e_n\}$ )	initialisiert $M = \{e_1, \dots, e_n\}$
PQ.insert( $e$ )	fügt $e$ in $M$ ein, d.h. $M \leftarrow M \cup \{e\}$ und gibt Zeiger $h$ auf das eingefügte Element zurück
PQ.max()	return max $M$
PQ.deleteMax()	löscht max $M$ , d.h. $M \leftarrow M \setminus \{\max M\}$
PQ.remove( $h$ )	löscht das Element aus $M$ , auf das der Zeiger $h$ zeigt
PQ.increaseKey( $h, k$ )	vergrößert den Schlüssel des Elements, auf das der Zeiger $h$ zeigt, auf Wert $k$
PQ.merge(PQ2)	vereinigt Priority Queues PQ und PQ2 $M \leftarrow M \cup M'$ , $M' \leftarrow \emptyset$ $M'$ ist die von PQ2 repräsentierte Schlüsselmenge

Wie schnell können wir diese Operationen implementieren? ↗ Übung + spätere VL

**Tipp:** Man kann unsere Priority Queue um weitere Daten erweitern

A	<table border="1"><tr><td>25</td><td>11</td><td>21</td><td>10</td><td>2</td><td>11</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	25	11	21	10	2	11	1	2	3	4	5	6
25	11	21	10	2	11								
1	2	3	4	5	6								
$A'$	<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	a	b	c	d	e	f	1	2	3	4	5	6
a	b	c	d	e	f								
1	2	3	4	5	6								



# Zusammenfassung

---

- **Stacks, Queues and Deques** als spezialisierte Datenstrukturen
  - naheliegende Implementierungen als verkettete Listen
  - simple und effiziente Implementierung als Array
- **Prioritätswarteschlangen**
  - erste baumbasierte Datenstruktur: **Heaps**
    - elegante Implementierung als Array
  - **HeapSort**
    - eine clevere Umsetzung der SelectionSort-Idee

Algorithmen und Datenstrukturen SS'23

# Kapitel 9: Binäre Suchbäume

Marvin Künemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letzte Kapitel: Elementare Datenstrukturen

Jetzt: Effiziente Darstellung von Wörterbüchern – Binäre Suchbäume

Insbesondere besprechen wir in diesem Kapitel:

- **Wörterbuchproblem (Dictionary problem)**
- Binäre Suche
- Binäre Suchbäume
  - grundlegende Operationen (Suche, Einfügen, Entfernen)
  - Balanciertheit
  - **Rot-Schwarz-Bäume** (red-black trees)

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen  
→ Lookup: Namen zu gegebener Matrikelnummer finden

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza

Wie heißt StudentIn mit Matrikelnummer 404040?

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen  
→ Lookup: Namen zu gegebener Matrikelnummer finden

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza

Wie heißt StudentIn mit Matrikelnummer 404040?  
→ Tahani Al-Jamil

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza

Wie heißt StudentIn mit Matrikelnummer 404040?

→ Tahani Al-Jamil

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza
666666:	Michael Realman

Wie heißt StudentIn mit Matrikelnummer 404040?

→ Tahani Al-Jamil

Füge "Michael Realman" mit Matrikelnummer 666666 ein

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen
  - Löschen: exmatrikulierte Studierende entfernen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza
666666:	Michael Realman

Wie heißt StudentIn mit Matrikelnummer 404040?

→ Tahani Al-Jamil

Füge "Michael Realman" mit Matrikelnummer 666666 ein

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen
  - Löschen: exmatrikulierte Studierende entfernen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jason Mendoza
666666:	Michael Realman

Wie heißt StudentIn mit Matrikelnummer 404040?

→ Tahani Al-Jamil

Füge "Michael Realman" mit Matrikelnummer 666666 ein

Lösche StudentIn mit Matrikelnummer 401337

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen
  - Löschen: exmatrikulierte Studierende entfernen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
666666:	Michael Realman

Wie heißt StudentIn mit Matrikelnummer 404040?

→ Tahani Al-Jamil

Füge "Michael Realman" mit Matrikelnummer 666666 ein

Lösche StudentIn mit Matrikelnummer 401337

# Ein Anwendungsszenario

---

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen
  - Löschen: exmatrikulierte Studierende entfernen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jianyu
666666:	Michael Realman

Wie heißt StudentIn mit Matrikelnummer 404040?

→ Tahani Al-Jamil

Füge "Michael Realman" mit Matrikelnummer 666666 ein

Lösche StudentIn mit Matrikelnummer 401337

Füge "Jianyu" mit Matrikelnummer 401337 ein

# Wörterbücher (Dictionaries, Associative Array)

---

Wir haben mehrere Möglichkeiten gesehen, **Sequenzen**  $a_1, \dots, a_n$  darzustellen

Wir betrachten nun besonders effiziente Repräsentationen von **Wörterbüchern**

**Wörterbuch (Dictionary, Assoziatives Array)**

Unsere Daten sind Elemente der Form  $e = (k, v)$

$\text{key}(e) = k \in K$

$\text{val}(e) = v \in V$

Schlüsselmenge  
mit totaler Ordnung

Wir wollen eine Menge  $S$  von Elementen darstellen, sodass folgende Operationen unterstützt werden:

$S.\text{find}(k)$

- gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

Wir nehmen an, dass es für jeden key  $k$  höchstens ein Element  $(k, v) \in S$  geben soll.

Wenn ja, dann gib den Wert  $v$  zurück, sodass  $(k, v) \in S$

Wenn nein, gib  $\perp$  zurück

$S.\text{insert}(e)$

- füge  $e$  in  $S$  ein

$S.\text{insert}(k, v)$

Wenn es bereits ein Element mit Schlüssel  $\text{key}(e)$  in  $S$  gibt, ersetze dieses durch  $e$

$S.\text{remove}(k)$

- lösche das Element mit Schlüssel  $k$  aus  $S$

# Zurück zum Anwendungsszenario

Angenommen, wir wollen aktuell **immatrikulierte Studierende** verwalten

Benötigte Funktionalität:

- Zuordnung von Matrikelnummer zu Namen
  - Lookup: Namen zu gegebener Matrikelnummer finden
  - Einfügen: neu immatrikulierte Studierende einfügen
  - Löschen: exmatrikulierte Studierende entfernen

417777:	Eleanor Shellstrop
404040:	Tahani Al-Jamil
411414:	Chidi Anagonye
401337:	Jianyu
666666:	Michael Realman

Dictionaries sind in praktisch allen Programmiersprachen implementiert:

## Dictionaries, Maps, Associative Arrays, ...

Bsp.:

- std::map bzw. std::unordered\_map (C++, STL)
- TreeMap bzw. HashMap (Java)
- dictionaries (Python)

```
map<int, string> dict;                                C++
//insert(666666, 'Michael Realman')
dict[666666] = 'Michael Realman';
if (dict.contains(401337)) { //find(401337) ≠ ⊥?
    dict.erase(401337);      //remove(401337)
}
```

## Spezialfall: Mengen (Sets)

Manchmal interessieren uns nur die Schlüssel  
→ Element  $e$  trägt keinen Wert  $\text{val}(e)$ , nur  $\text{key}(e)$

```
set<int> my_set;                                C++
my_set.insert(666666);
if (my_set.contains(401337)) {
    my_set.erase(401337);
}
```

# Wörterbücher (Dictionaries, Associative Array)

## Wörterbuch (Dictionary, Assoziatives Array)

Unsere Daten sind Elemente der Form  $e = (k, v)$

$\text{key}(e) = k \in K$  Schlüsselmenge  
 $\text{val}(e) = v \in V$  mit totaler Ordnung

Wir wollen eine Menge  $S$  von Elementen darstellen, sodass folgende Operationen unterstützt werden:

$S.\text{find}(k)$

- gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

Wir nehmen an, dass es für jeden key  $k$  höchstens ein Element  $(k, v) \in S$  geben soll.

Wenn ja, dann gib den Wert  $v$  zurück, sodass  $(k, v) \in S$   
Wenn nein, gib  $\perp$  zurück

$S.\text{insert}(e)$

- füge  $e$  in  $S$  ein

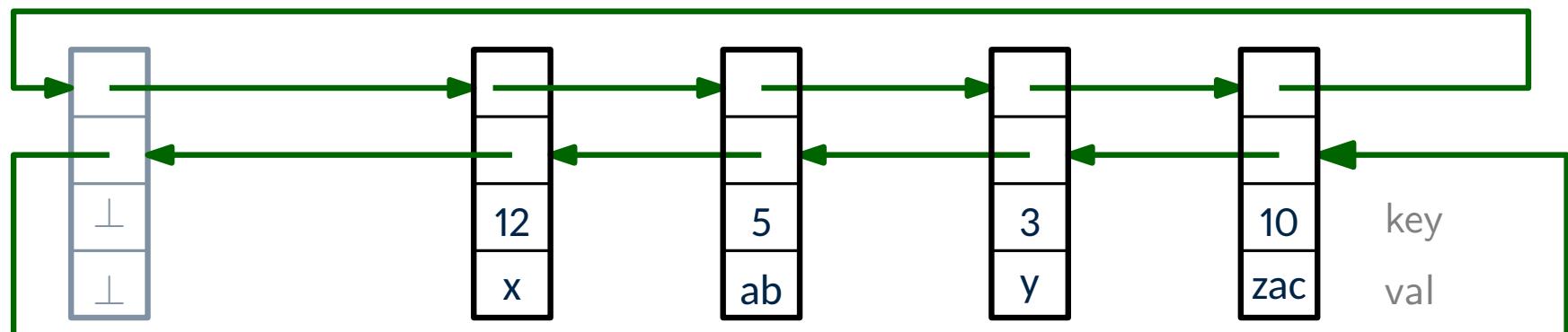
$S.\text{insert}(k, v)$

Wenn es bereits ein Element mit Schlüssel  $\text{key}(e)$  in  $S$  gibt, ersetze dieses durch  $e$

$S.\text{remove}(k)$

- lösche das Element mit Schlüssel  $k$  aus  $S$

Wir könnten  $S$  mithilfe von Arrays oder verkettete Listen implementieren.



# Binäre Suche

---

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

$A$

2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

`find( $A[1, \dots, 14]$ , 9)`

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$

`find( $A[1, \dots, 14]$ , 9)`

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$

find( $A[1, \dots, 14], 9$ )

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$

find( $A[1, \dots, 14], 9$ )

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

find( $A[1, \dots, 14], 9$ )

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2   3   5   6   9   14   16   17   18   20   24   25   31   49
---	--

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

```
find(A[1, ..., 14], 9)    return 5
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

```
find(A[1, ..., 14], 9)    return 5
```

```
find(A[1, ..., 14], 10)
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

```
find(A[1, ..., 14], 9)    return 5
```

```
find(A[1, ..., 14], 10)
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

```
find(A[1, ..., 14], 9)    return 5
```

```
find(A[1, ..., 14], 10)
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2   3   5   6   9   14   16   17   18   20   24   25   31   49
---	--

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

```
find(A[1, ..., 14], 9)    return 5
```

```
find(A[1, ..., 14], 10)
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

```
find(A[1, ..., 14], 9)    return 5
```

```
find(A[1, ..., 14], 10)
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

Wenn wir  $x$  in leerem Intervall  $A[\ell \dots r]$  mit  $\ell > r$  suchen,  
gib ungültigen Index  $-1$  zurück

```
find(A[1, ..., 14], 9)  return 5
```

```
find(A[1, ..., 14], 10) return -1
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

$A$	$-\infty$	2	3	5	6	9	14	16	17	18	20	24	25	31	49	$\infty$
-----	-----------	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----------

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

Wenn wir  $x$  in leerem Intervall  $A[\ell \dots r]$  mit  $\ell > r$  suchen,  
gib ungültigen Index  $-1$  zurück

Korrektheit? Frage: angemessene Schleifeninvariante?

$$A[\ell - 1] < x < A[r + 1]$$

`find(A[1, ..., 14], 9) return 5`

`find(A[1, ..., 14], 10) return -1`

```
int find(type A[], int l, int r, type key) {  
    while (l <= r) {  
        int m = (l+r)/2;  
        if (A[m] == key) {  
            return m;  
        } else if (A[m] < key) {  
            l = m+1;  
        } else {  
            r = m-1;  
        }  
    }  
    return -1;  
}
```

# Binäre Suche

## Bemerkung:

Wir können einen gegebenen Schlüssel schnell suchen,  
wenn wir eine sortiertes Array  $A[1 \dots n]$  der Schlüssel gegeben haben

$$A[1] < A[2] < \dots < A[n]$$

A	2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

## rekursiver Divide-&-Conquer-Ansatz:

Um  $x$  in  $A[\ell \dots r]$  mit  $\ell \leq r$  zu suchen,

- Vergleiche  $x$  mit  $A[m]$ , wobei  $m = \lfloor \frac{\ell+r}{2} \rfloor$
- Ist  $x = A[m]$ , gib  $m$  zurück
- Ist  $x < A[m]$ , suche  $x$  in  $A[\ell \dots m - 1]$
- Ist  $x > A[m]$ , suche  $x$  in  $A[m + 1 \dots r]$

Wenn wir  $x$  in leerem Intervall  $A[\ell \dots r]$  mit  $\ell > r$  suchen,  
gib ungültigen Index  $-1$  zurück

```
find(A[1, ..., 14], 9)  return 5
```

```
find(A[1, ..., 14], 10) return -1
```

```
int find(type A[], int l, int r, type key) {  
    while (l <= r) {  
        int m = (l+r)/2;  
        if (A[m] == key) {  
            return m;  
        } else if (A[m] < key) {  
            l = m+1;  
        } else {  
            r = m-1;  
        }  
    }  
    return -1;  
}
```

## Korrektheit? ✓

## Laufzeit:

Sei  $T(n)$  die Laufzeit von `find` auf  $A[\ell \dots r]$  mit  $n = r - \ell + 1$

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + C$$

$$\Rightarrow T(n) = O(\log n)$$

# Dictionaries durch sortiertes Array von Schlüsseln?

**Ansatz:** Wir könnten  $S$  mithilfe eines sortierten Arrays von Schlüsseln implementieren

K	2	3	5	6	9	14	16	17	18	20	24	25	31	49
V	Mindy	Tahani	Michael	Shawn	Eleanor	Simone	Glenn	Jason	John	Vicky	Chidi	Derek	Brent	Janet

F: Laufzeiten dieser Lösung?

Wörterbuch (Dictionary, Assoziatives Array)

`S.find(k)` - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

`S.insert(e)` - füge  $e$  in  $S$  ein

`S.remove(k)` - lösche das Element mit Schlüssel  $k$  aus  $S$

Wie können wir das Einfügen und Löschen verbessern?

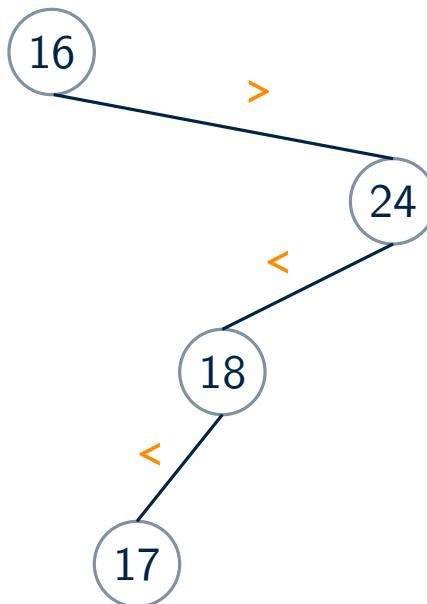
# Binäre Suche als Entscheidungsbaum

---

A

2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	----	----	----	----	----	----	----	----	----

find(A[1 ... n], 17)

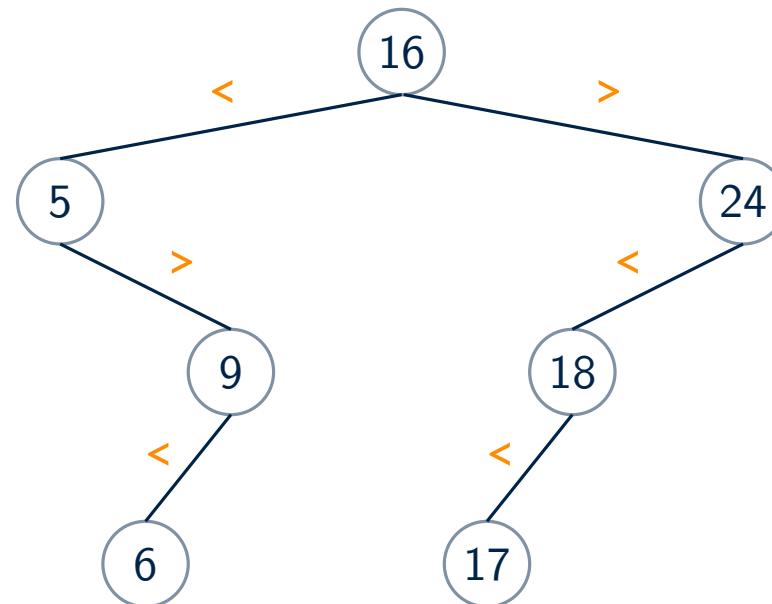


# Binäre Suche als Entscheidungsbaum

A

2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	----	----	----	----	----	----	----	----	----

find( $A[1 \dots n]$ , 17)  
find( $A[1 \dots n]$ , 8)

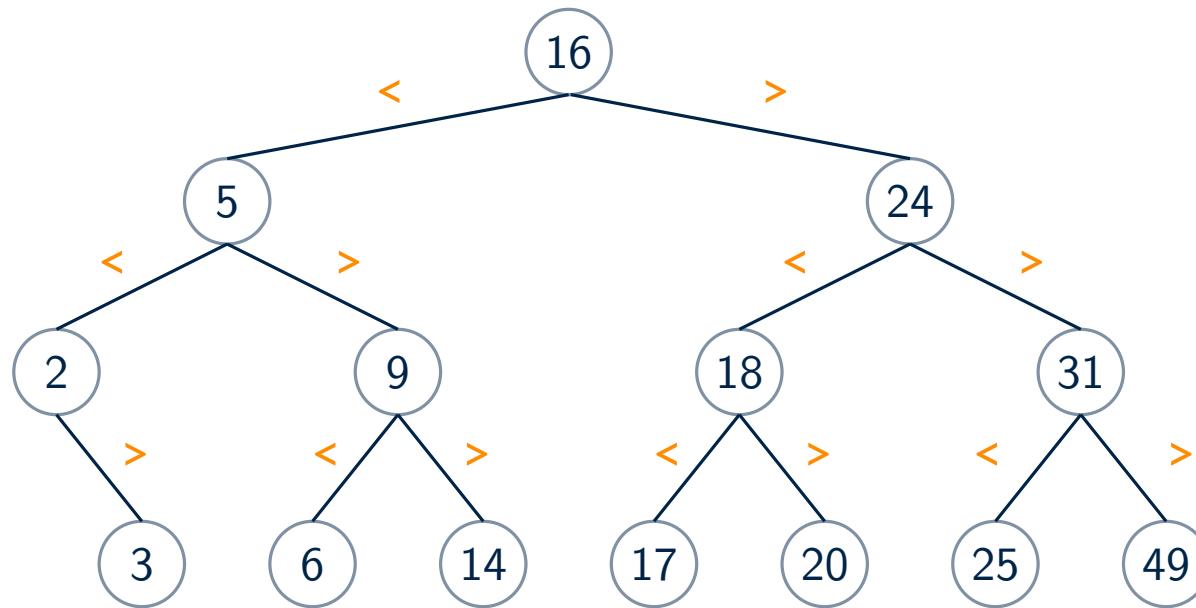


# Binäre Suche als Entscheidungsbaum

A

2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	----	----	----	----	----	----	----	----	----

find(A[1 ... n], 17)  
find(A[1 ... n], 8)  
...

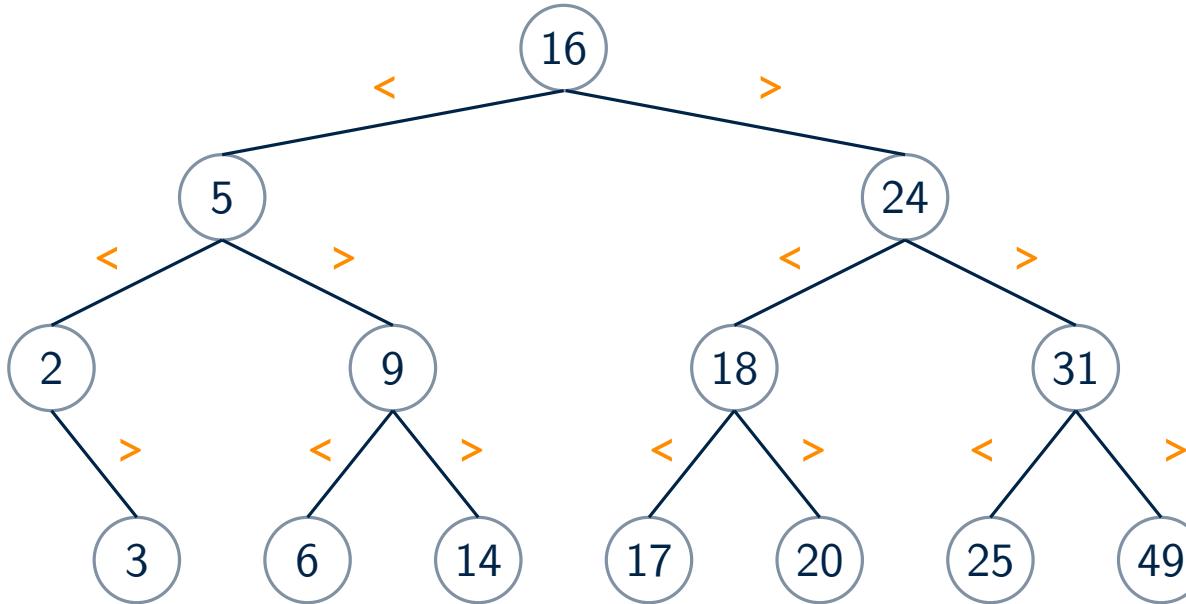


# Binäre Suche als Entscheidungsbaum

A

2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	----	----	----	----	----	----	----	----	----

find(A[1 ... n], 17)  
find(A[1 ... n], 8)  
...

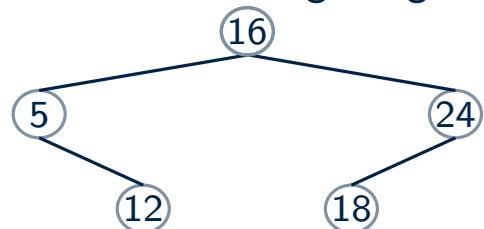


**Definition.**

Jeder Knoten hat höchstens einen linken Kindknoten und höchstens einen rechten Kindknoten.

Ein binärer, geordneter Baum heißt **binärer Suchbaum**, wenn:

- jedem Knoten  $u$  ist ein Schlüssel  $k_u$  aus einer total geordneten Schlüsselmenge zugeordnet
- für jeden Knoten  $u$  gilt:
  - alle Schlüssel im linken Teilbaum von  $u$  sind  $< k_u$
  - alle Schlüssel im rechten Teilbaum von  $u$  sind  $> k_u$

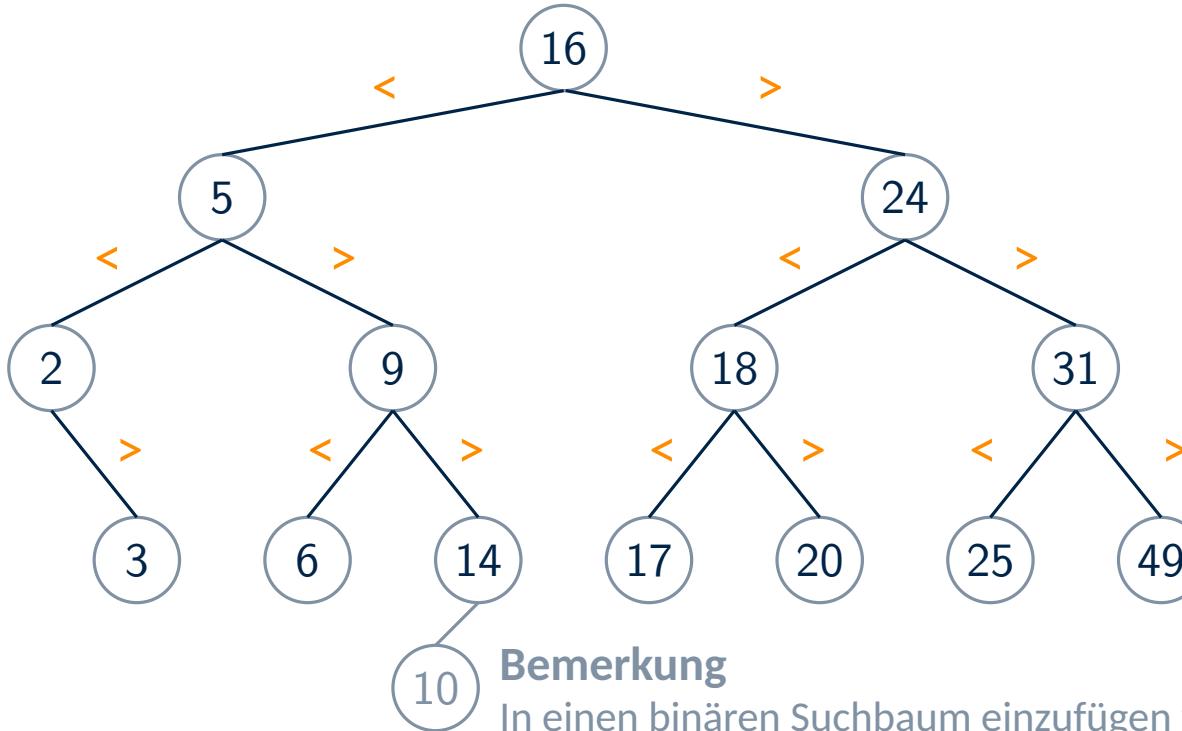


# Binäre Suche als Entscheidungsbaum

A

2	3	5	6	9	14	16	17	18	20	24	25	31	49
---	---	---	---	---	----	----	----	----	----	----	----	----	----

find(A[1 ... n], 17)  
find(A[1 ... n], 8)  
...

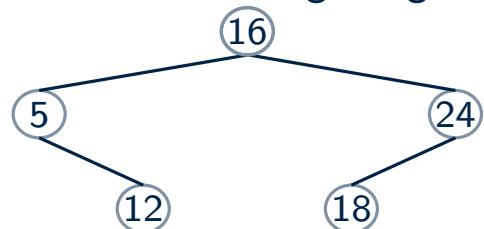


**Definition.**

Jeder Knoten hat höchstens einen linken Kindknoten und höchstens einen rechten Kindknoten.

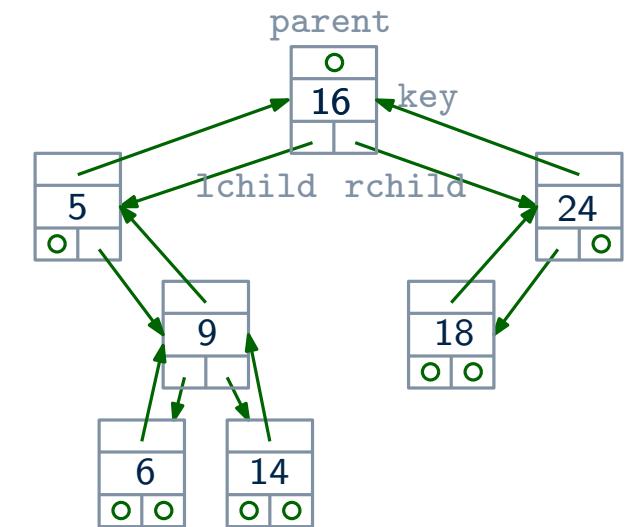
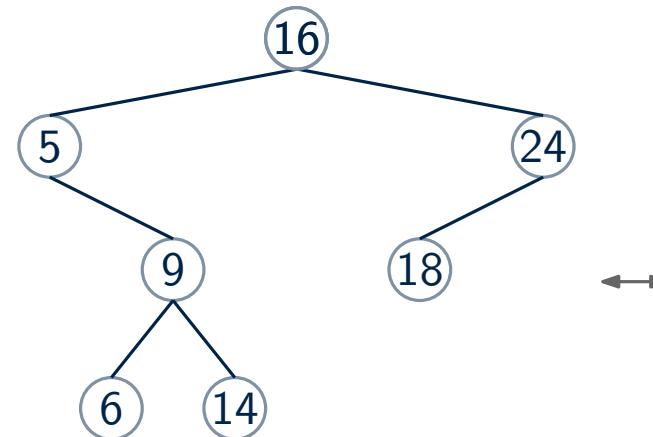
Ein **binärer, geordneter Baum** heißt **binärer Suchbaum**, wenn:

- jedem Knoten  $u$  ist ein Schlüssel  $k_u$  aus einer total geordneten Schlüsselmenge zugeordnet
- für jeden Knoten  $u$  gilt:
  - alle Schlüssel im linken Teilbaum von  $u$  sind  $< k_u$
  - alle Schlüssel im rechten Teilbaum von  $u$  sind  $> k_u$



# Implementierung eines binären Suchbaums

```
struct Node {  
    Node* parent;  
    Node* lchild;  
    Node* rchild;  
    type key;  
    type val;  
    ...  
}
```



Hinweis: val nicht dargestellt

Der Baum ist dann gegeben als Zeiger auf den Wurzelknoten:

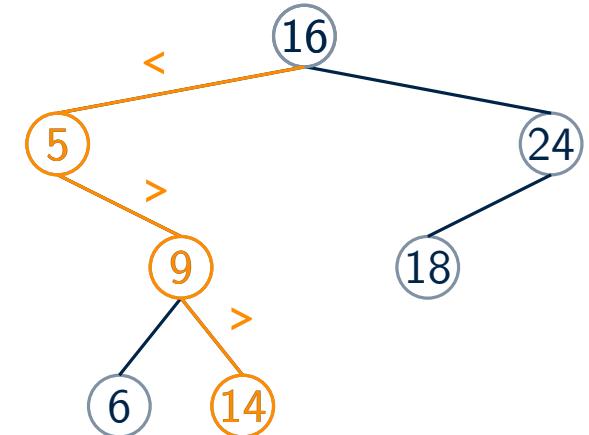
```
struct Tree {  
    Node* root;  
    ...  
}
```

Hinweis: Diese Art der Implementierung lässt sich einfach auf beliebige Bäume erweitern

Im folgenden sei `Tree T` ein solcher Baum.

# Suchbäume: find

```
Node* find(Node* x, type k) {  
    if (x == nullptr or x->key == k) {  
        return x;  
    }  
    if (k < x->key) {  
        return find(x->lchild, k);  
    } else {  
        return find(x->rchild, k);  
    }  
}
```



find(T.root, 12): return nullptr

- Laufzeit:**
- in jedem rekursiven Aufruf erhöht sich das Level des betrachteten Knoten x
  - die Laufzeit ergibt sich durch  $T(h) = T(h - 1) + C$ , wobei  $h$  die Höhe des Baumes T ist.
  - Laufzeit ist  $\leq C \cdot h = O(h)$

## Zusätzliche Operationen:

`min(u)` - findet den Knoten mit dem kleinsten Schlüssel im Teilbaum an Knoten u

F: Wie implementiert man `min(u)`? → möglich in Laufzeit  $O(h)$ . ✓ [nächste Folie](#)

Der **Nachfolger von u** sei der Knoten mit dem nächstgrößeren Schlüssel zu u

`successor(u)` - findet den Nachfolger von u

F: Wie implementiert man `successor(u)`? → möglich in Laufzeit  $O(h)$ . ✓ [nächste Folie](#)

11 - 18 `max(u)` und `predecessor(u)` sind natürlich analog definiert und implementiert.

# Suchbäume: min/max; succ/pred

Finde Element mit kleinstem Schlüssel im Teilbaum an  $u$ :

```
Node* min(Node* u) {  
    if (u->lchild != nullptr) {  
        return min(u->lchild);  
    }  
    return u;  
}
```

//Element mit kleinstem Schlüssel im linken Teilbaum, wenn existent

Laufzeit  $O(h)$

Gib Nachfolger von  $u$  zurück (bzw. `nullptr` wenn  $u$  größten Schlüssel trägt):

```
Node* successor(Node* u) {  
    if (u->rchild != nullptr) {  
        return min(u->rchild);  
    }  
    Node* p = u->parent;  
    while (p != nullptr and u != p->lchild) {  
        u = p;  
        p = p->parent;  
    }  
    return p;  
}
```

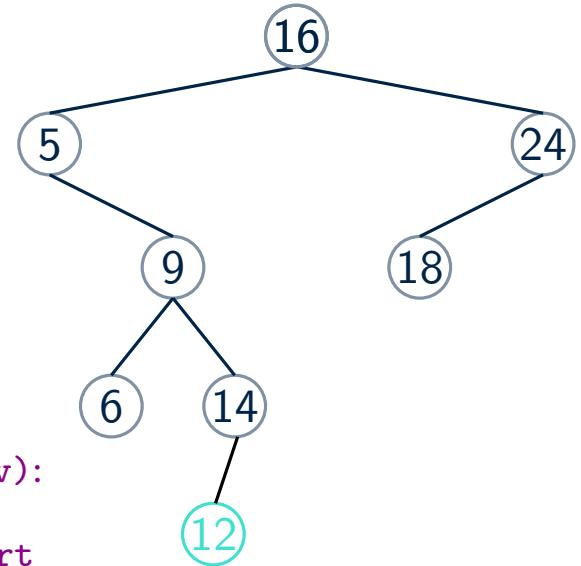
//Element mit kleinstem Schlüssel im rechten Teilbaum, wenn existent

//ansonsten: der erste Vorfahre  $p$ , sodass  $u$  im linken Teilbaum von  $p$  ist

Laufzeit  $O(h)$

# Suchbäume: insert

---



T.insert(12, v):  
T.find(12)  
update/insert

Wie implementieren wir insert(k, v)?

Ein Aufruf von `find(T.root, k)` findet entweder ein Element mit Schlüssel  $k$  oder die Stelle, an der ein solches Element eingefügt werden muss.

Wir passen also `find` dementsprechend an:

Wir können den entsprechenden Knoten aktualisieren bzw.  
einen neuen Knoten an der entsprechenden Stelle einfügen

→ nur  $O(1)$  zusätzliche Laufzeit verglichen zu `find`

**Laufzeit:**  $O(h)$ , wobei  $h$  die Höhe des Baumes ist.

# Suchbäume: remove

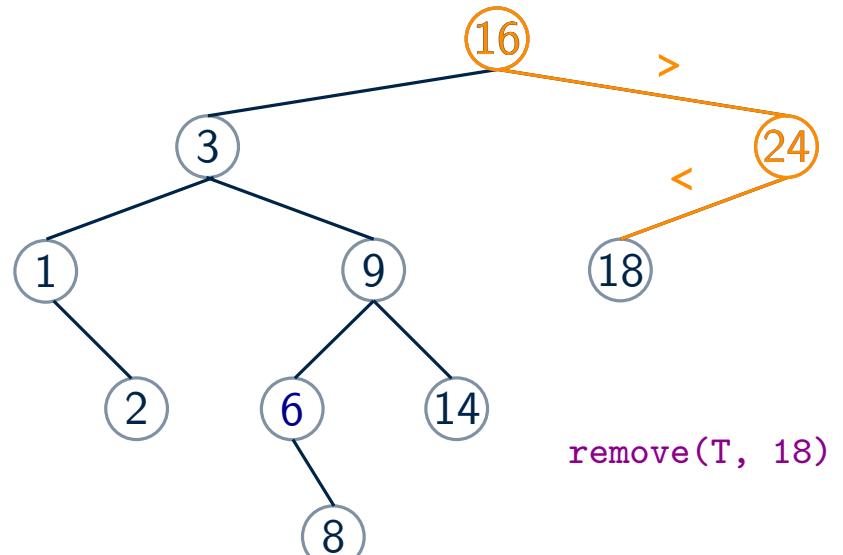
Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

**Fall 1:**  $u$  ist ein Blatt



# Suchbäume: remove

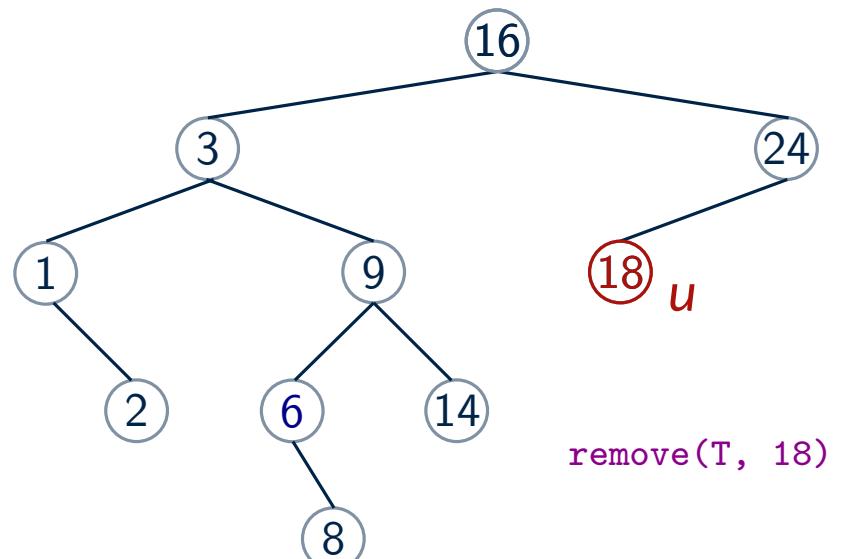
Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

**Fall 1:**  $u$  ist ein Blatt



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

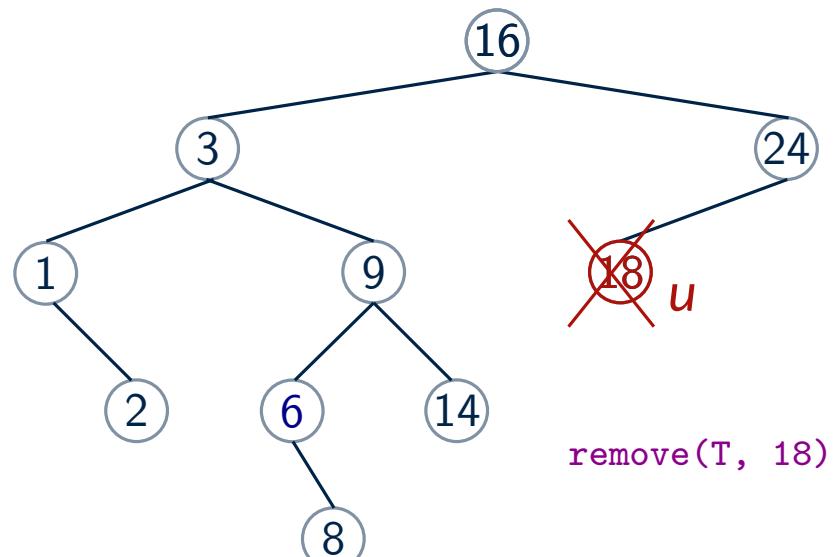
## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

### Fall 1: $u$ ist ein Blatt

Es reicht dieses Blatt zu löschen:

`u.parent->lchild = nullptr` bzw.  
`u.parent->rchild = nullptr`



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

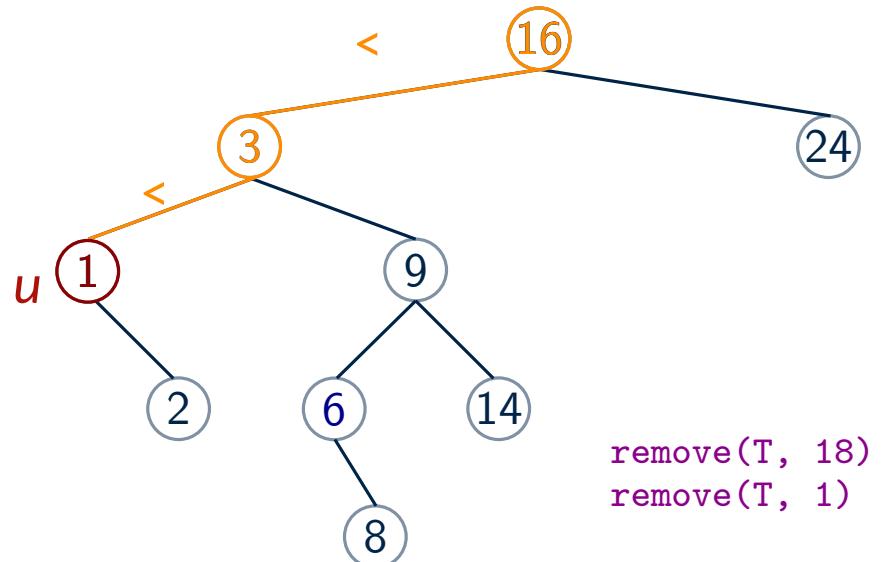
## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

### Fall 1: $u$ ist ein Blatt

Es reicht dieses Blatt zu löschen:      `u.parent->lchild = nullptr` bzw.  
`u.parent->rchild = nullptr`

### Fall 2: $u$ hat genau ein Kind



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

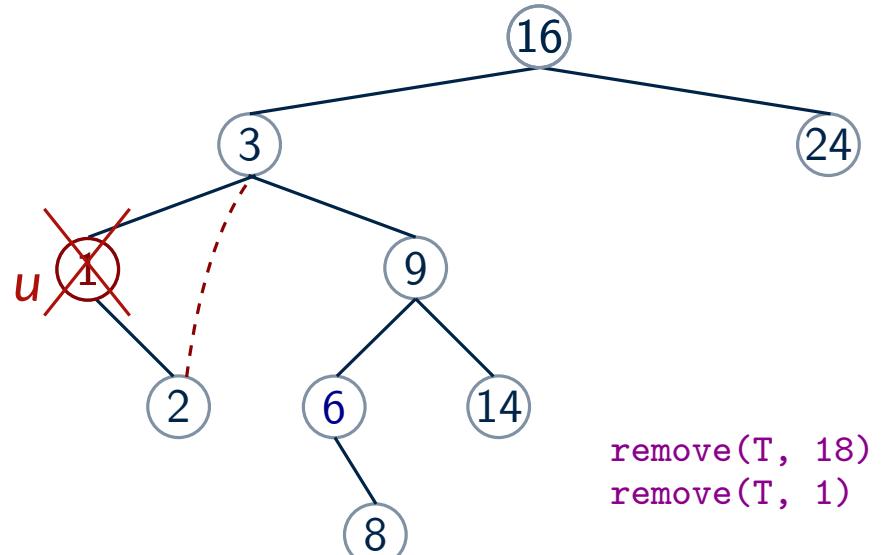
Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

### Fall 1: $u$ ist ein Blatt

Es reicht dieses Blatt zu löschen:     `u.parent->lchild = nullptr` bzw.  
   `u.parent->rchild = nullptr`

### Fall 2: $u$ hat genau ein Kind

Es reicht,  $u$  zu löschen und das Kind von  $u$  zum entsprechenden Kind des Elternknotens von  $u$  zu machen



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

S.remove(k) - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

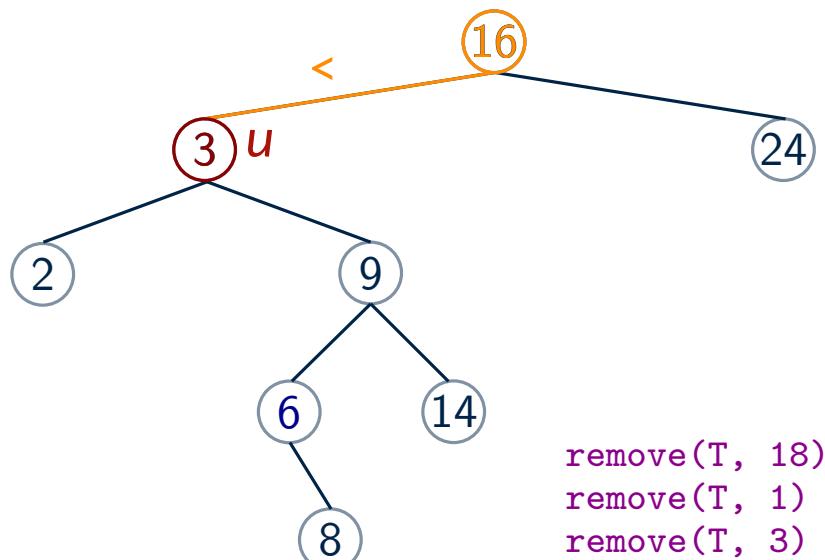
### Fall 1: $u$ ist ein Blatt

Es reicht dieses Blatt zu löschen:  $u.parent \rightarrow lchild = \text{nullptr}$  bzw.  
 $u.parent \rightarrow rchild = \text{nullptr}$

### Fall 2: $u$ hat genau ein Kind

Es reicht,  $u$  zu löschen und das Kind von  $u$  zum entsprechenden Kind des Elternknotens von  $u$  zu machen

### Fall 3: $u$ hat genau zwei Kinder



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

### Fall 1: $u$ ist ein Blatt

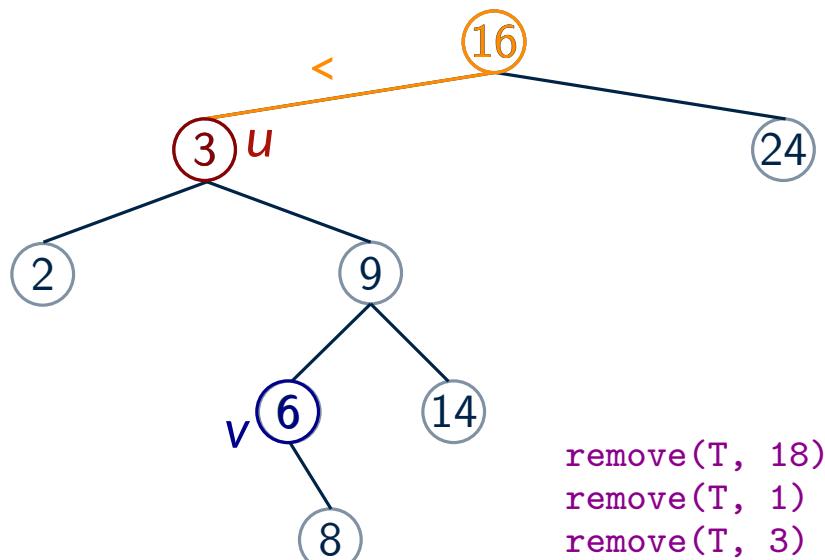
Es reicht dieses Blatt zu löschen:      `u.parent->lchild = nullptr` bzw.  
`u.parent->rchild = nullptr`

### Fall 2: $u$ hat genau ein Kind

Es reicht,  $u$  zu löschen und das Kind von  $u$  zum entsprechenden Kind des Elternknotens von  $u$  zu machen

### Fall 3: $u$ hat genau zwei Kinder

Wir finden den Nachfolger  $v$  von  $u$



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

### Fall 1: $u$ ist ein Blatt

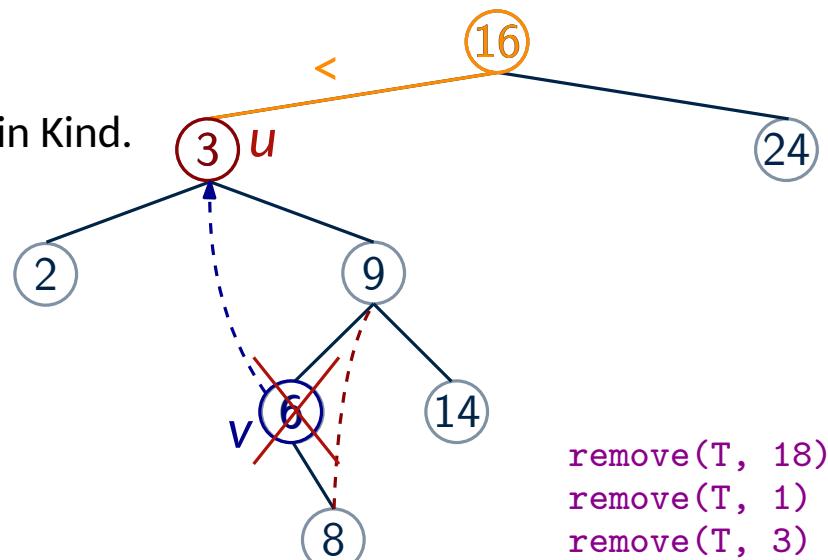
Es reicht dieses Blatt zu löschen:  $u.parent \rightarrow lchild = \text{nullptr}$  bzw.  
 $u.parent \rightarrow rchild = \text{nullptr}$

### Fall 2: $u$ hat genau ein Kind

Es reicht,  $u$  zu löschen und das Kind von  $u$  zum entsprechenden Kind des Elternknotens von  $u$  zu machen

### Fall 3: $u$ hat genau zwei Kinder

Wir finden den Nachfolger  $v$  von  $u$   
→ Beobachtung:  $v$  hat kein linkes Kind, also höchstens ein Kind.  
Wir löschen  $v$  von dort (Fall 1 od. 2)  
und speichern  $v$  an der Stelle von  $u$



# Suchbäume: remove

Die Löschoperation ist leicht aufwändiger.

`S.remove(k)` - lösche das Element mit Schlüssel k aus S

## Algorithmenbeschreibung:

Wir finden zunächst den Knoten  $u$ , der den Schlüssel  $k$  repräsentiert.  
Wenn dieser nicht existiert, bleibt nichts zu tun.

### Fall 1: $u$ ist ein Blatt

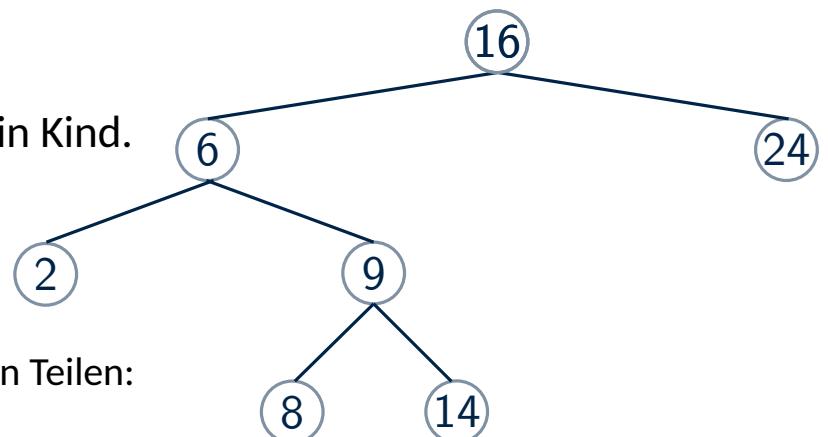
Es reicht dieses Blatt zu löschen:  $u.parent \rightarrow lchild = \text{nullptr}$  bzw.  
 $u.parent \rightarrow rchild = \text{nullptr}$

### Fall 2: $u$ hat genau ein Kind

Es reicht,  $u$  zu löschen und das Kind von  $u$  zum entsprechenden Kind des Elternknotens von  $u$  zu machen

### Fall 3: $u$ hat genau zwei Kinder

Wir finden den Nachfolger  $v$  von  $u$   
→ Beobachtung:  $v$  hat kein linkes Kind, also höchstens ein Kind.  
Wir löschen  $v$  von dort (↗ Fall 1 od. 2)  
und speichern  $v$  an der Stelle von  $u$



**Laufzeit:** Ein Aufruf von `remove` besteht aus höchstens den folgenden Teilen:

- Finden von  $u$
- Finden des Nachfolgers  $v$  von  $u$
- $O(1)$  Zusatzaufwand (Kopieren, etc.)

`remove(T, 18)`  
`remove(T, 1)`  
`remove(T, 3)`

14 - 22 → Laufzeit  $O(h)$ , wobei  $h$  die Höhe des Baumes ist.

# Zwischenstand

---

Wörterbuch (Dictionary, Assoziatives Array) via Binären Suchbäumen

$h$  ist Höhe des Suchbaumes

`S.find(k)` - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k?$   $O(h)$

`S.insert(e)` - füge  $e$  in  $S$  ein  $O(h)$

`S.remove(k)` - lösche das Element mit Schlüssel  $k$  aus  $S$   $O(h)$

Zusätzliche Operationen:

`S.min()` - gib das Element mit dem kleinsten Schlüssel zurück  $O(h)$   
`S.max()` - gib das Element mit dem größten Schlüssel zurück  $O(h)$

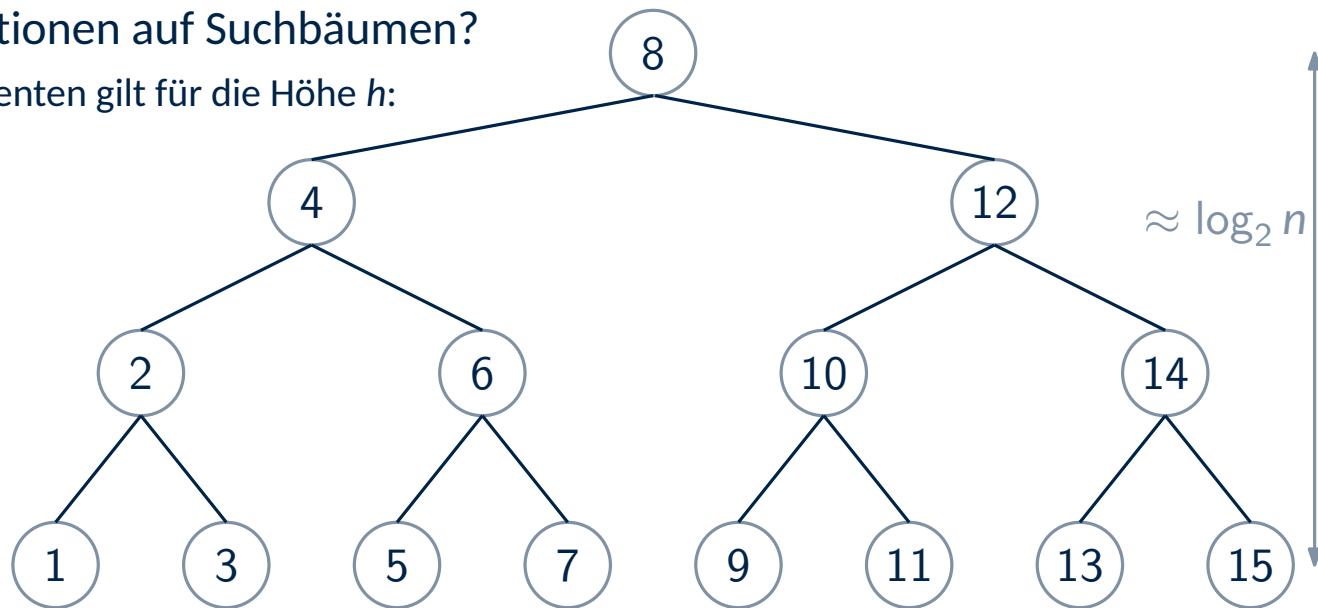
`S.successor(k)` - gib das Element mit dem nächstgrößeren Schlüssel zu  $k$  zurück  $O(h)$   
`S.predecessor(k)` - gib das Element mit dem nächstkleineren Schlüssel zu  $k$  zurück  $O(h)$

# Effizienz der Suchbäume: Balanciertheit

Wie effizient sind die  $O(h)$ -Operationen auf Suchbäumen?

In jedem binären Suchbaum von  $n$  Elementen gilt für die Höhe  $h$ :

$$\log_2 n \leq h + 1 \leq n$$



Beide Suchbäume repräsentieren die gleiche Schlüsselmenge!

**Aber:** Höhe des Baumes unterscheidet sich erheblich

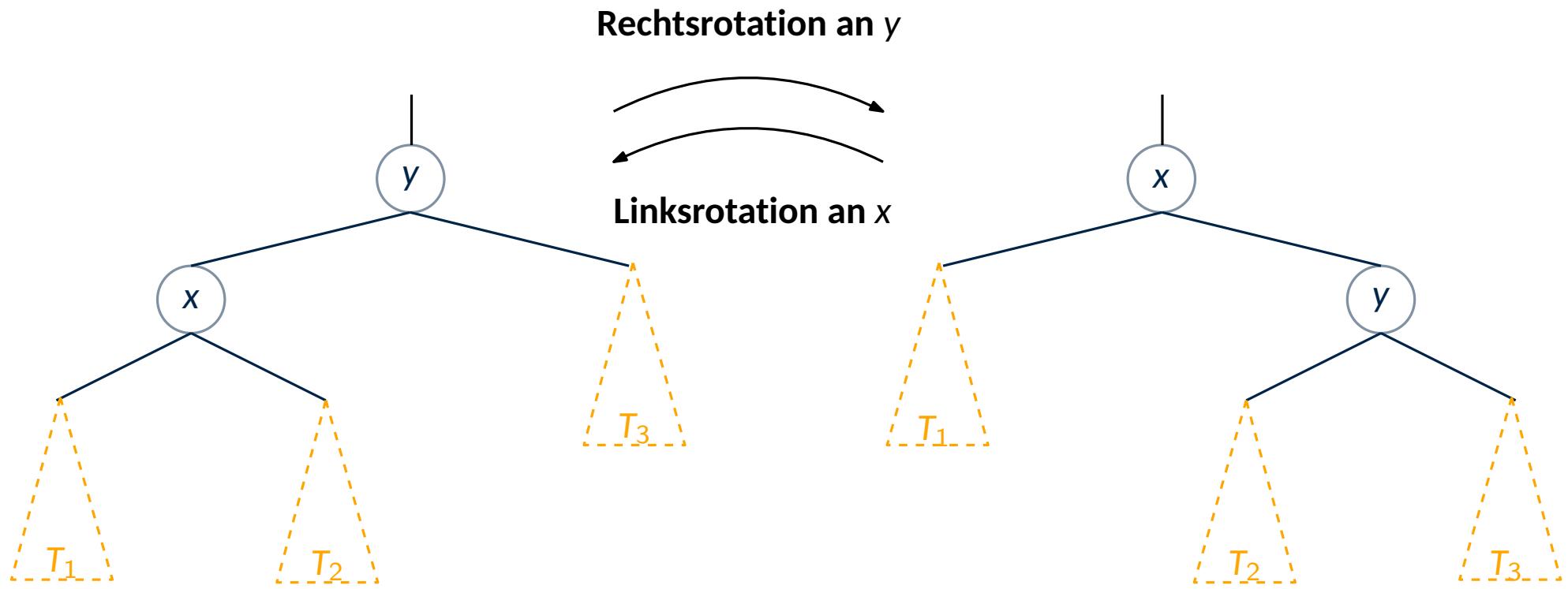
→ Dramatischer Unterschied in der Laufzeit der Operationen

**Konsequenz:**

Wir möchten immer einen  
möglichst balancierten Suchbaum erhalten

# Rotationen

---



## Lemma

Links- und Rechtsrotationen erhalten die Binäre-Suchbaum-Eigenschaft.

Frage: Wie implementiert man eine Links- oder Rechtsrotation? Laufzeit?

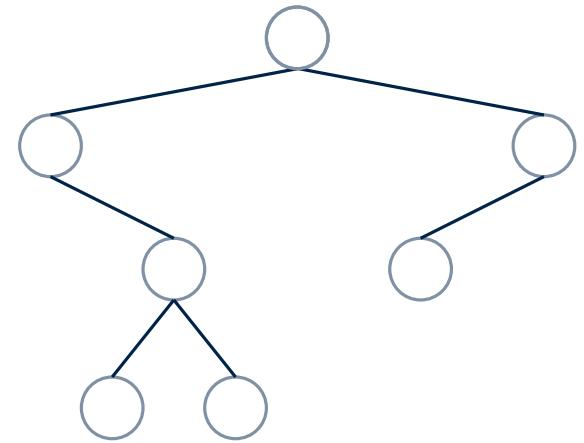
# Die Macht von Rotationen

---

## Eigenschaften von binären Suchbäumen:

- gegeben binärer, geordneter Baum  $T$  mit  $n$  Knoten sowie Schlüsselmenge  $S$  mit  $|S| = n$ ,

$$S = \{5, 6, 9, 14, 16, 18, 24\}$$



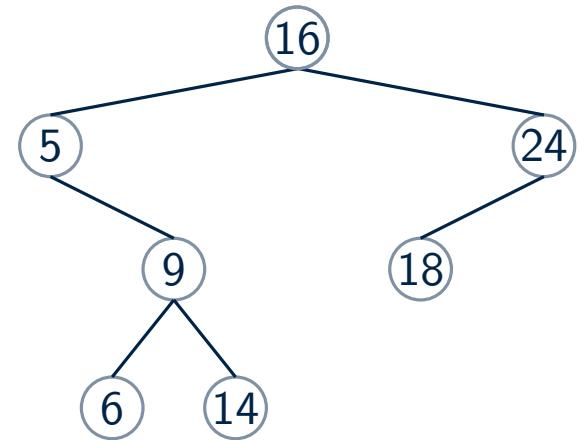
# Die Macht von Rotationen

---

## Eigenschaften von binären Suchbäumen:

- gegeben binärer, geordneter Baum  $T$  mit  $n$  Knoten sowie Schlüsselmenge  $S$  mit  $|S| = n$ , gibt es eindeutige bijektive Zuordnung von  $S$  zu Knoten in  $T$ , sodass  $T$  ein binärer Suchbaum ist

$$S = \{5, 6, 9, 14, 16, 18, 24\}$$



# Die Macht von Rotationen

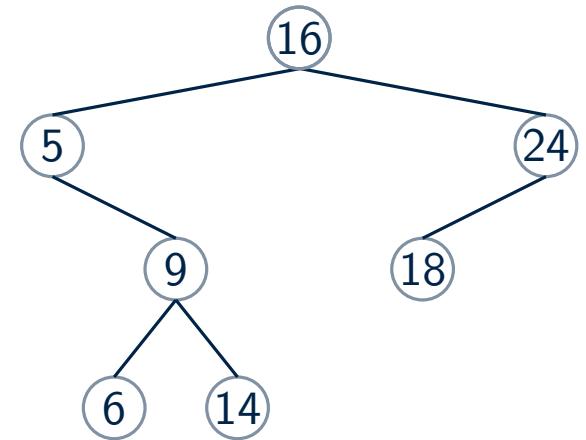
---

## Eigenschaften von binären Suchbäumen:

- gegeben binärer, geordneter Baum  $T$  mit  $n$  Knoten sowie Schlüsselmenge  $S$  mit  $|S| = n$ , gibt es eindeutige bijektive Zuordnung von  $S$  zu Knoten in  $T$ , sodass  $T$  ein binärer Suchbaum ist

$$S = \{5, 6, 9, 14, 16, 18, 24\}$$

- für beliebige binäre, geordnete Bäume  $T_1, T_2$  mit gleicher Knotenzahl können wir  $T_1$  durch Rotationen zu  $T_2$  umformen  
allerdings: im schlimmsten Fall benötigen wir dazu  $\Omega(n)$  Rotationen



Wir werden zeigen:

Man kann binäre Suchbäume so implementieren, dass

- jedes Einfügen und Löschen höchstens  $O(1)$  Rotationen benötigt, und trotzdem
- die Höhe in  $O(\log n)$  bleibt.

# Selbstbalancierende Suchbäume

Es gibt viele verschiedene Datenstrukturen, die **balancierte** Suchbäume aufrechterhalten.

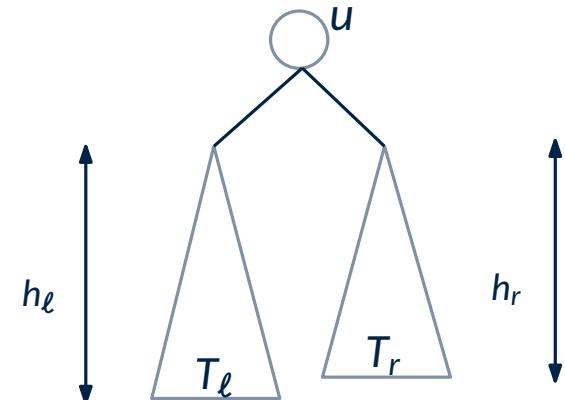
Für diese Datenstrukturen gilt:

- die Höhe bleibt durch  $O(\log n)$  beschränkt. "balanciert" → `find, minimum, successor, etc.` laufen in Zeit  $O(\log n)$
  - der zusätzliche Aufwand zur Aufrechterhaltung der Balanciertheit ist in  $O(\log n)$
- Hinweis: keine **perfekte** Balancierung nötig  
(↗ Vergleich: Median der Mediane)
- betrifft nur `insert` und `remove`

U.a. die folgenden Datenstrukturen implementieren balancierte Suchbäume:

- AVL-Bäume  
Adelson-Velski, Landis '62

**Idee von AVL-Bäumen:**  
· stelle sicher, dass für jeden Knoten  $u$  gilt:



$$|h_\ell - h_r| \leq 1$$

Höhen des linken und rechten Teilbaums fast gleich ( $\pm 1$ )

# Selbstbalancierende Suchbäume

---

Es gibt viele verschiedene Datenstrukturen, die **balancierte** Suchbäume aufrechterhalten.

Für diese Datenstrukturen gilt:

- die Höhe bleibt durch  $O(\log n)$  beschränkt. "balanciert" → `find, minimum, successor, etc.` laufen in Zeit  $O(\log n)$
  - der zusätzliche Aufwand zur Aufrechterhaltung der Balanciertheit ist in  $O(\log n)$
- Hinweis: keine **perfekte** Balancierung nötig  
(↗ Vergleich: Median der Mediane)
- betrifft nur `insert` und `remove`

U.a. die folgenden Datenstrukturen implementieren balancierte Suchbäume:

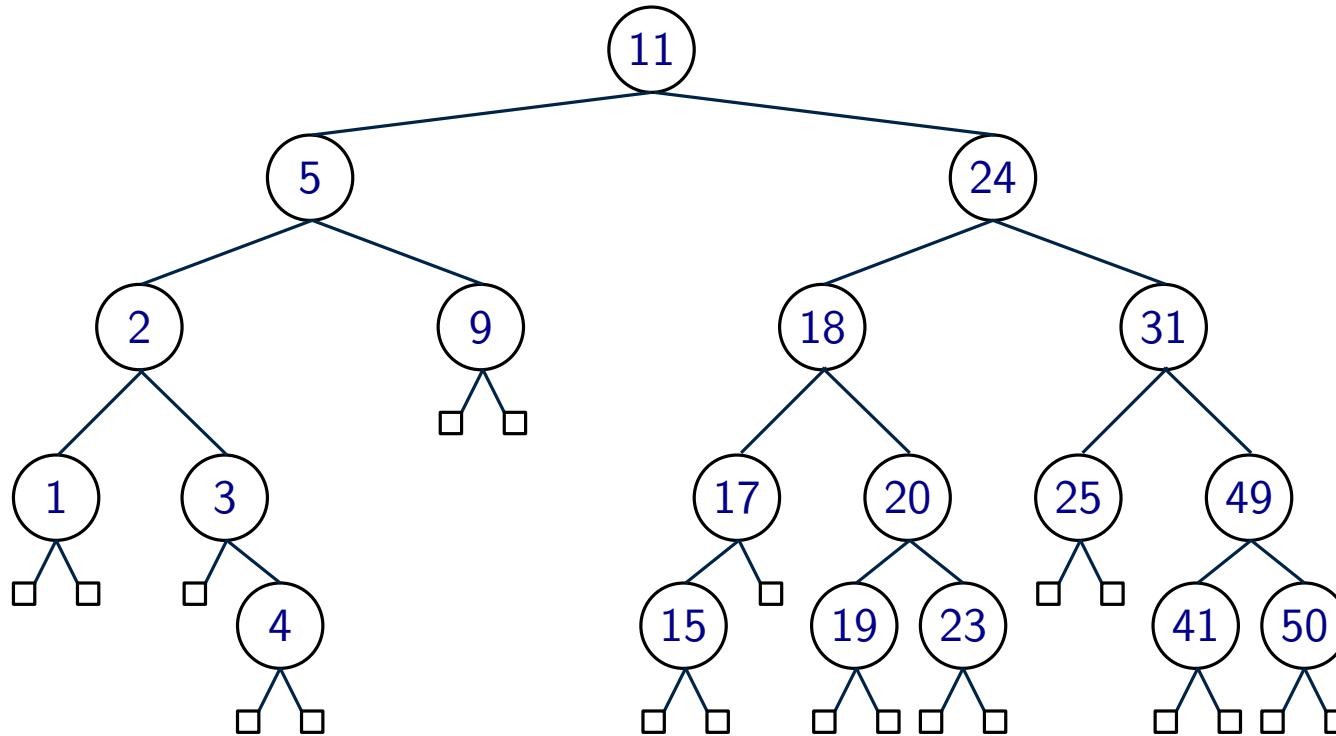
- AVL-Bäume
- **Rot-Schwarz**-Bäume (red-black trees)
- $(a, b)$ -Bäume (B-Bäume):  
Spezialfälle: 2-3-Bäume, 2-3-4-Bäume
- Treaps

Wir betrachten zunächst **Rot-Schwarz-Bäume**

- gehören zu den meistgenutzten balancierten Suchbäumen
- effiziente Implementierungen in vielen Bibliotheken
- z.B. `std::map` in C++, `TreeMap` in Java

# Kleine Definitionshilfe: virtuelle Blätter

---

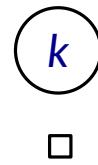


Jeden fehlenden linken oder rechten Kindknoten, d.h. jedes Vorkommen von `nullptr`, ersetzen wir durch ein **virtuelles Blatt**.

In dem neuen Baum haben wir jetzt:

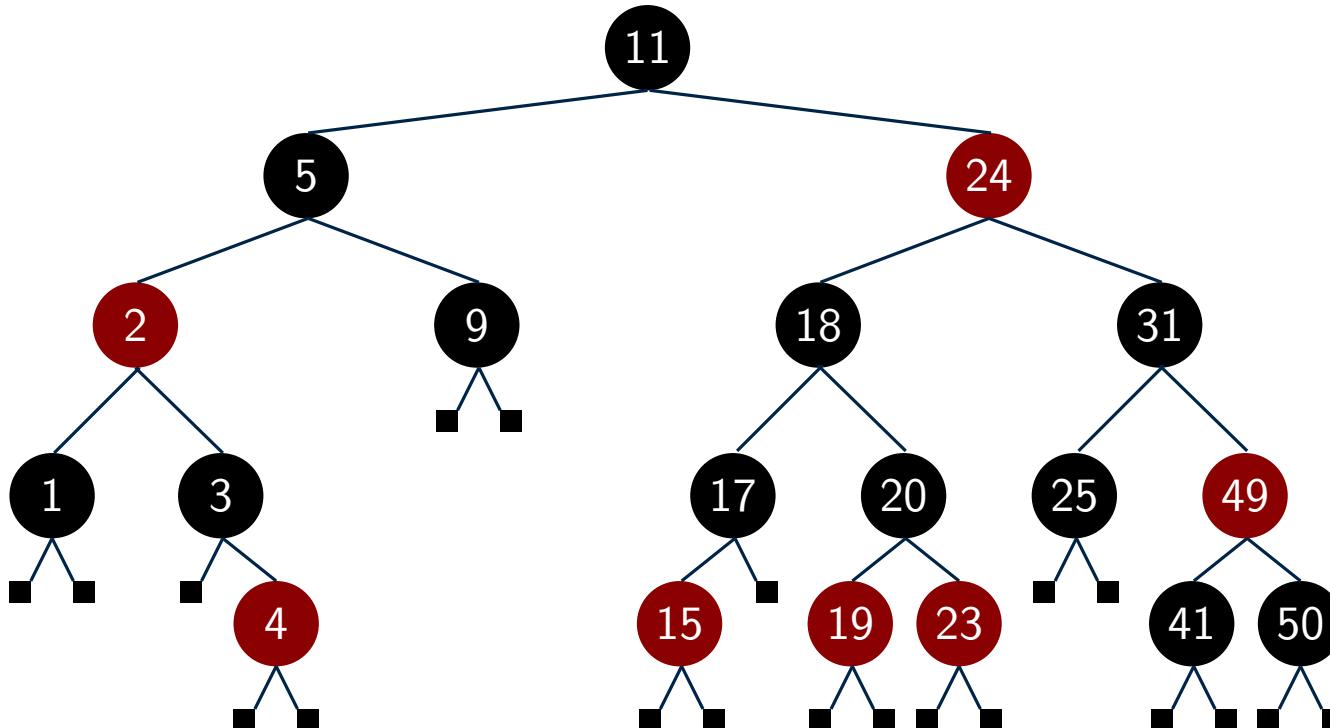
- **n innere Knoten** mit je genau zwei Kindern
- **$n + 1$  virtuelle Blätter**

**Schlüsselknoten**  
**Blatt**



# Rot-Schwarz-Bäume: Definition

Idee: Jeder Knoten bekommt eine Farbe (**rot**/schwarz), die wir nur zur Balancierung nutzen



## Definition

Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum, in dem die Schlüsselknoten **rot** oder **schwarz** gefärbt sind, sodass:

Wurzelbedingung: · Die Wurzel ist **schwarz** gefärbt

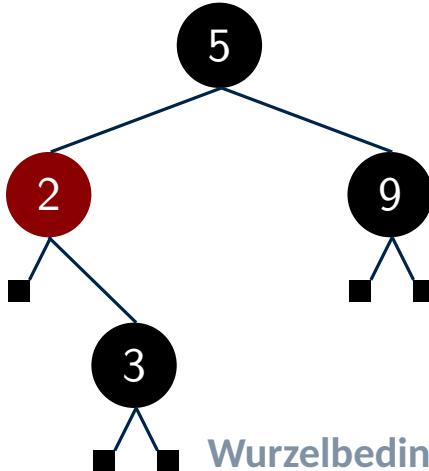
Rotbedingung: · Für jeden **roten Knoten** muss der Elternknoten **schwarz** gefärbt sein.  
→ kein Doppelrot von Eltern- und Kindknoten

Tiefenbedingung: · Die **Schwarztiefe** jedes Blättes muss gleich sein.

Die **Schwarztiefe** von  $b$  ist definiert als Anzahl **schwarzer Knoten** auf dem Pfad von der Wurzel zu  $b$ .

Hierbei sind die virtuellen Blätter immer **schwarz** gefärbt.

# Rot-Schwarz-Baum oder kein Rot-Schwarzbaum?



Wurzelbedingung:

- Die Wurzel ist **schwarz**.

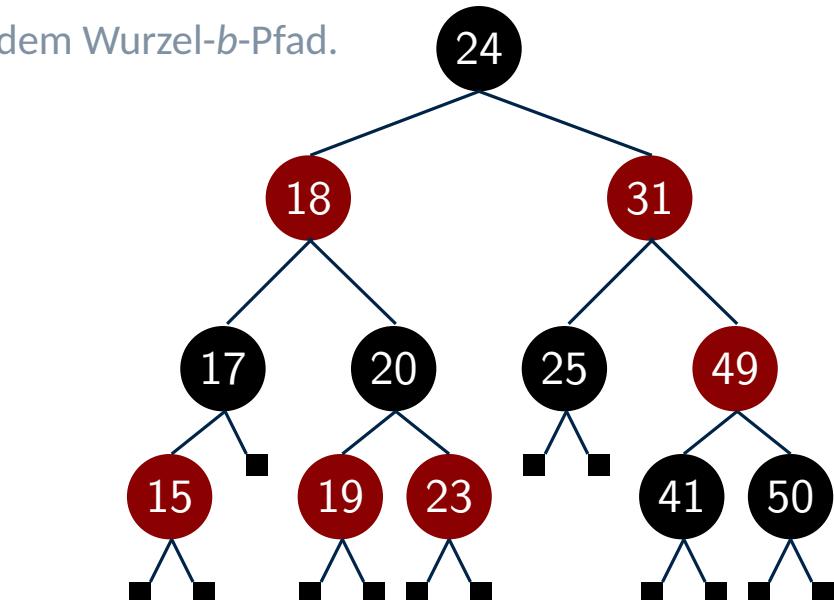
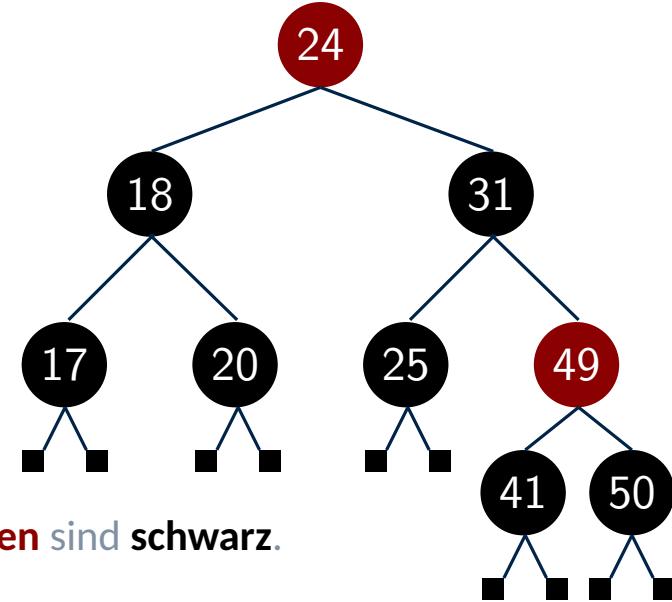
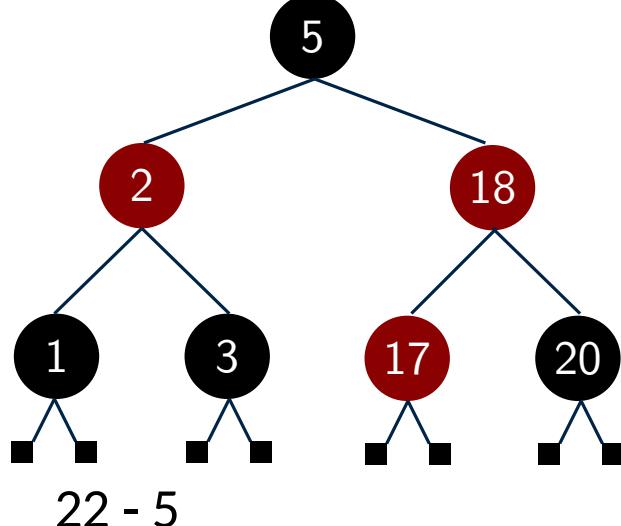
Rotbedingung:

- Elternknoten von **roten Knoten** sind **schwarz**.

Tiefenbedingung:

- Die **Schwarztiefe** jedes Blattes  $b$  muss gleich sein.

Schwarztiefe von  $b$ : Anzahl **schwarzer Knoten** auf dem Wurzel- $b$ -Pfad.



# Höhe

## Lemma

Für die Höhe  $h$  eines Rot-Schwarz-Baums mit  $n$  Schlüsselknoten gilt:

$$\log_2(n + 1) \leq h \leq 2 \log_2(n + 1)$$

## Beweis:

Unser Binärbaum hat  $n + 1$  Blätter, also gilt:

$$h \geq \log_2(n + 1)$$

Interessanter ist, dass  $h \leq 2 \log_2(n + 1)$ :

Sei  $s$  die Schwarztiefe der Blätter.

Jeder Pfad zu einem Blatt hat:

- $s$  schwarze Knoten      **Schwarztiefe**
- $\leq s - 1$  rote Knoten      **Rotbedingung!**

$\Rightarrow$  Höhe ist beschränkt durch  $h \leq 2(s - 1)$

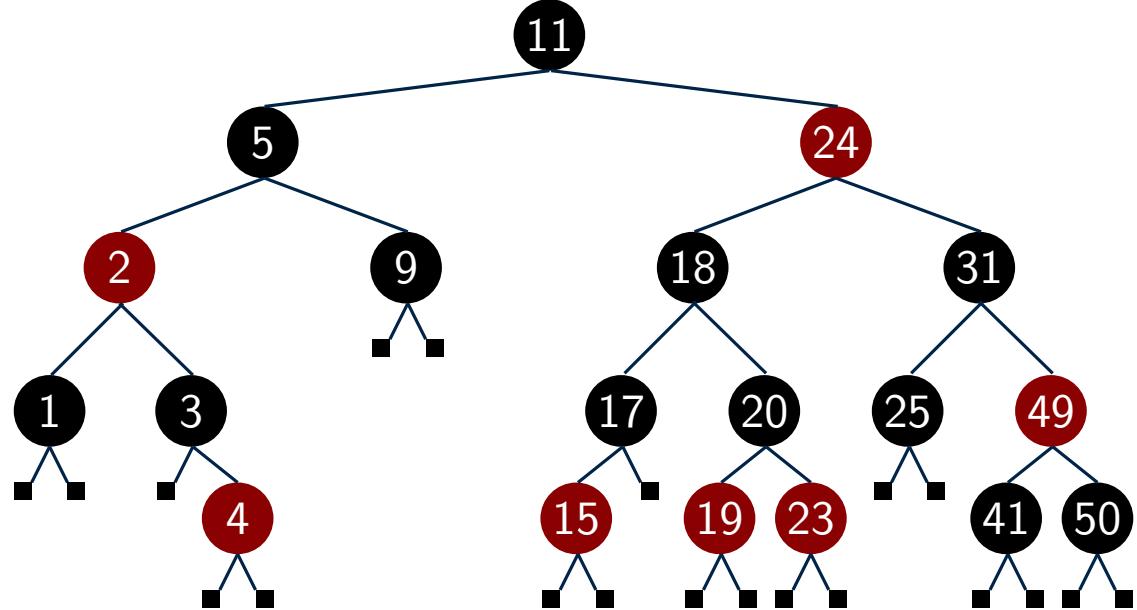
Aufgrund der **Tiefenbedingung** gilt:

Die ersten  $s - 1$  Level des Baumes sind vollständig mit Schlüsselknoten besetzt.

$$n \geq \sum_{i=0}^{s-2} 2^i = 2^{s-1} - 1 \quad \Rightarrow 2^{s-1} \leq n + 1$$

$$\Rightarrow h \leq 2(s - 1) \leq 2 \log_2(n + 1)$$

□



**Wurzelbedingung:**

- Die Wurzel ist **schwarz**.

**Rotbedingung:**

- Elternknoten von **roten Knoten** sind **schwarz**.

**Tiefenbedingung:**

- Die **Schwarztiefe** jedes Blattes muss gleich sein.

**Schwarztiefe** von  $b$ : Anzahl **schwarzer Knoten** auf dem Wurzel- $b$ -Pfad.

Fakt 1: Binärbaum der Höhe  $h$  hat höchstens  $2^h$  Blätter.

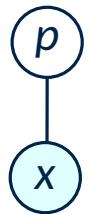
Fakt 2:  $h = \max. \#Knoten$  auf Wurzel-Blatt-Pfad – 1

Fakt 3: die ersten  $k$  Level eines vollständigen Binärbaums haben

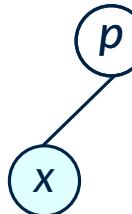
$$\sum_{i=0}^{k-1} 2^i = 2^k - 1 \text{ Knoten.}$$

# Etwas Notation und Begrifflichkeit

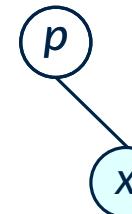
---



steht für



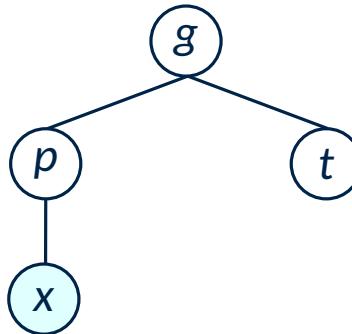
oder



Kleine Familienkunde:

*g* - ist Großelter von x

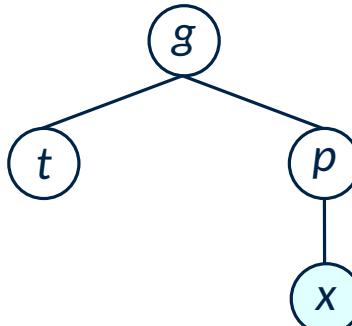
*p* - ist Elter von x



*t* - ist ... Tante? Onkel? **Tankell!** von x

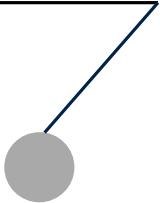
Mit Dank an Prof. Schweitzer für diese Begrifflichkeit

symmetrische Situation:



# Einfügen in Rot-Schwarz-Baum

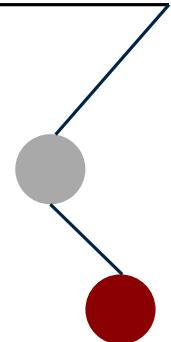
---



# Einfügen in Rot-Schwarz-Baum

---

**Grundprinzip:** · Wenn wir einen neuen Knoten einfügen, wird er **rot** gefärbt

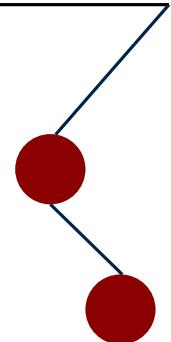


# Einfügen in Rot-Schwarz-Baum

---

**Grundprinzip:**

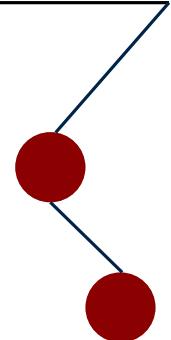
- Wenn wir einen neuen Knoten einfügen, wird er **rot** gefärbt
- Dabei kann es eine **Rotverletzung** entstehen.



# Einfügen in Rot-Schwarz-Baum

---

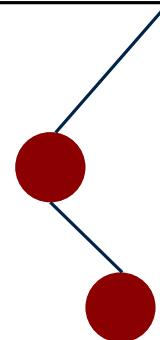
- Grundprinzip:**
- Wenn wir einen neuen Knoten einfügen, wird er **rot** gefärbt
  - Dabei kann es eine **Rotverletzung** entstehen.
  - Wir versuchen folgendes:
    - die **Rotverletzung** durch **Umfärbung** an den Großelternknoten abzugeben und rekursiv mit dem Großelternknoten fortzufahren



# Einfügen in Rot-Schwarz-Baum

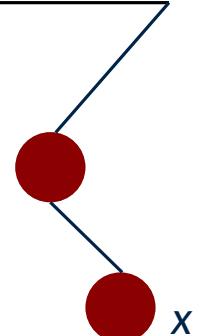
---

- Grundprinzip:**
- Wenn wir einen neuen Knoten einfügen, wird er **rot** gefärbt
  - Dabei kann es eine **Rotverletzung** entstehen.
  - Wir versuchen folgendes:
    - die **Rotverletzung** durch **Umfärbung** an den Großelternknoten abzugeben und rekursiv mit dem Großelternknoten fortzufahren
    - oder
    - die **Rotverletzung** durch Rotation(en) + Umfärbungen aufzulösen



# Einfügen in Rot-Schwarz-Baum

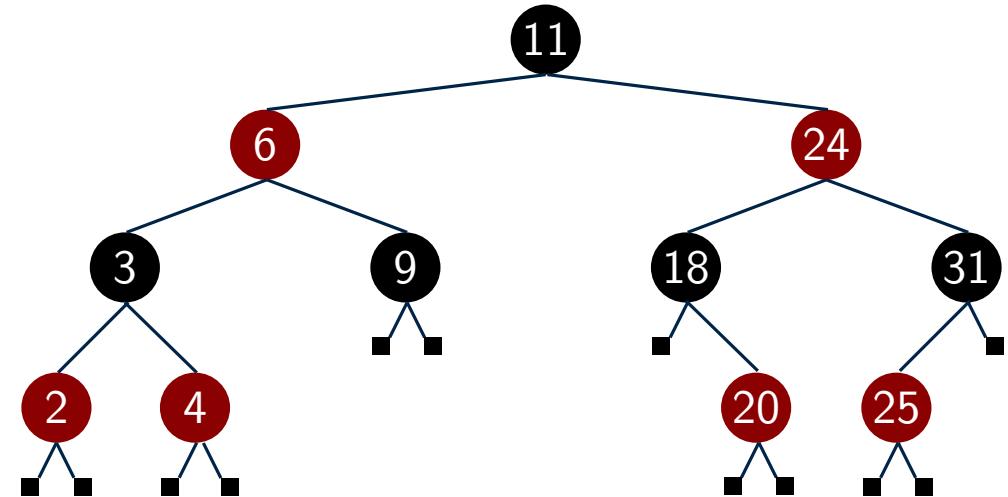
- Grundprinzip:**
- Wenn wir einen neuen Knoten einfügen, wird er **rot** gefärbt
  - Dabei kann es eine **Rotverletzung** entstehen.
  - Wir versuchen folgendes:
    - die **Rotverletzung** durch **Umfärbung** an den Großelternknoten abzugeben und rekursiv mit dem Großelternknoten fortzufahren
    - oder
    - die **Rotverletzung** durch Rotation(en) + Umfärbungen aufzulösen



```
insert(e): Füge Schlüssel  $k = \text{key}(e)$  an die entsprechende Stelle im Baum ein  $\rightsquigarrow$  Knoten x  
Färbe x rot.  
while (Rotverletzung bei x) do  
    behebeRotverletzung(x) // darf x verändern! Insbesondere: Auf Großelter setzen  
if (x ist die Wurzel) then  
    färbe x nötigenfalls schwarz
```

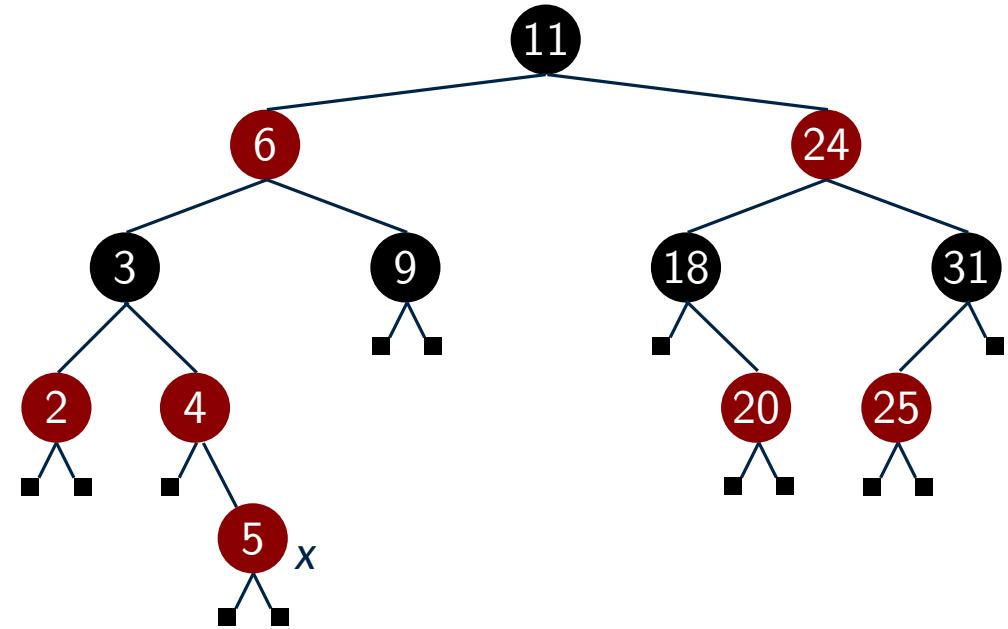
# behebeRotverletzung(x): Fall 1 - Umfärben

---



# behebeRotverletzung(x): Fall 1 - Umfärben

---



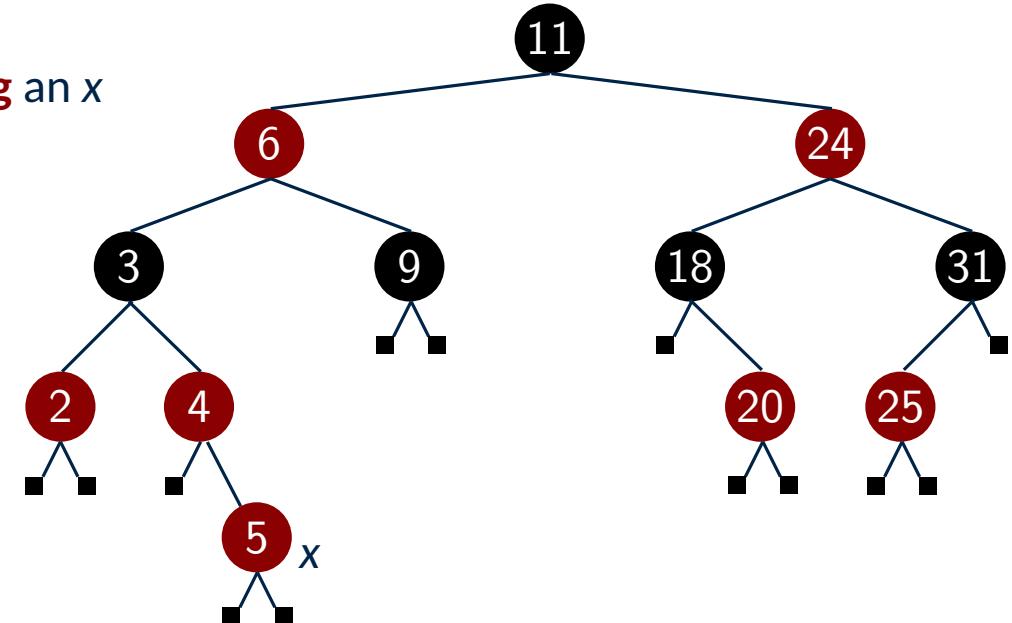
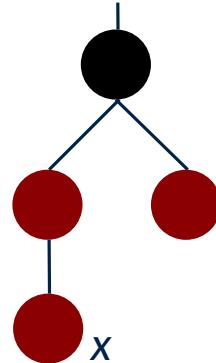
# behebeRotverletzung(x): Fall 1 - Umfärbben

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Vater t von x ist **rot**



# behebeRotverletzung(x): Fall 1 - Umfärbchen

Ausgangssituation:

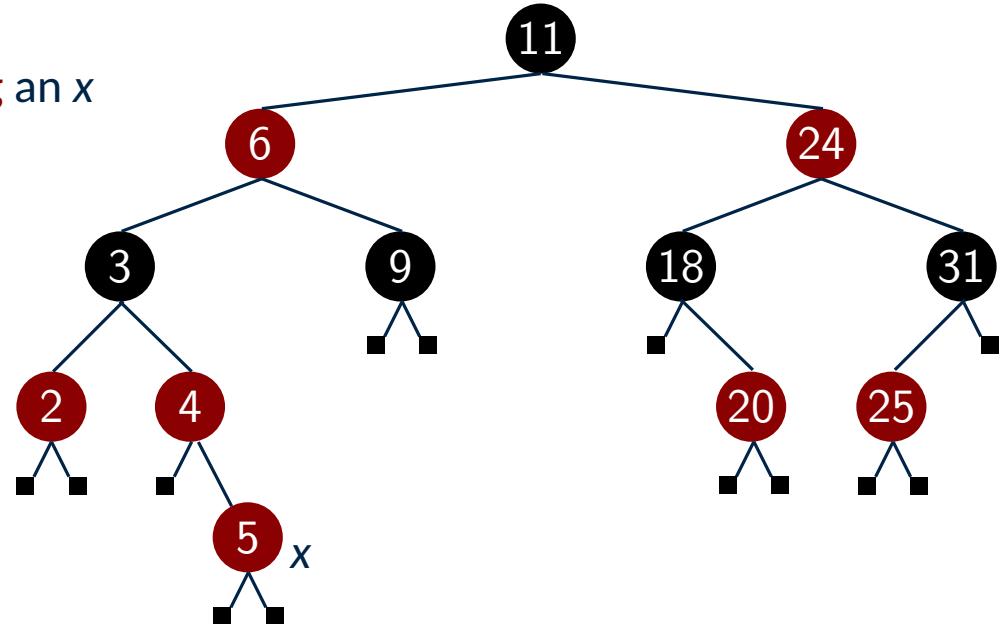
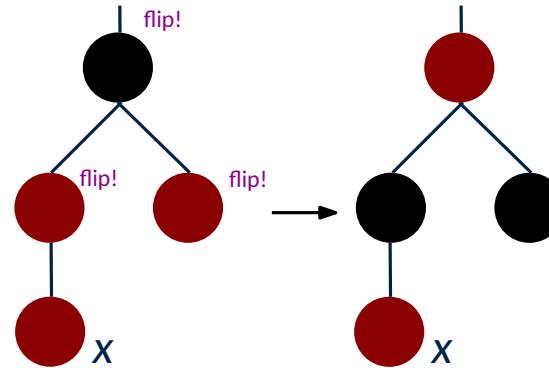
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



# behebeRotverletzung(x): Fall 1 - Umfärbung

Ausgangssituation:

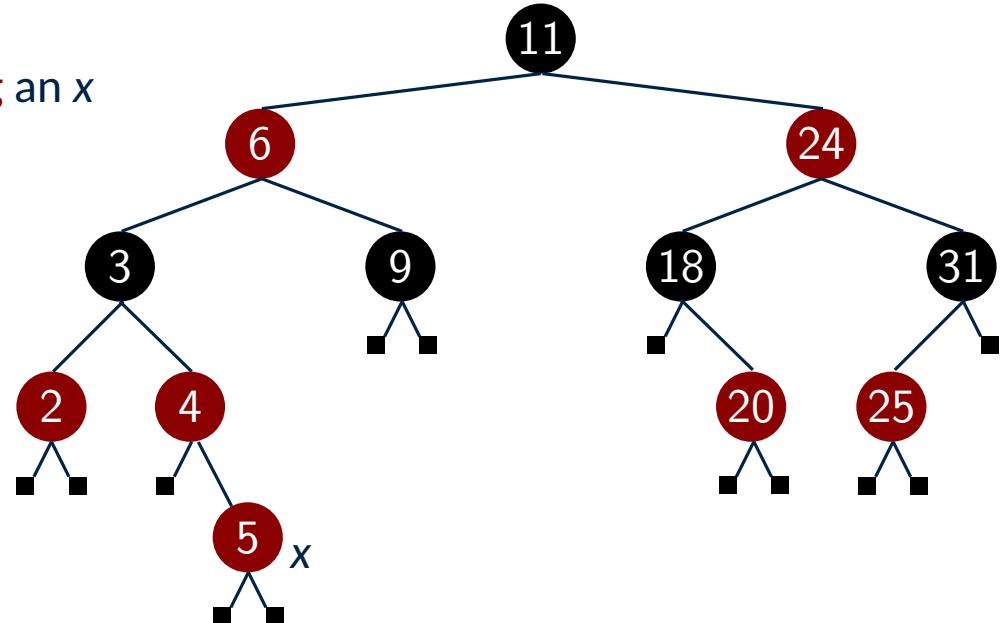
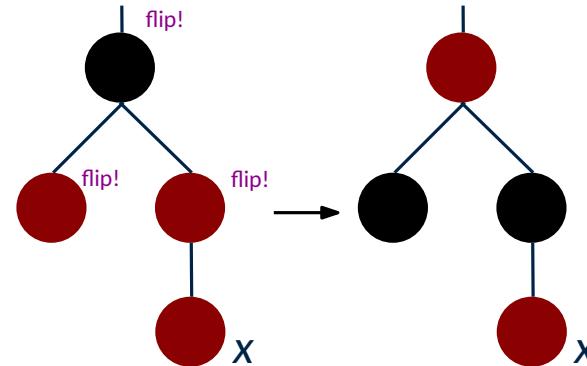
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



Der Fälle "x rechter Enkelknoten" und "x linker Enkelknoten" sind symmetrisch

# behebeRotverletzung(x): Fall 1 - Umfärbben

Ausgangssituation:

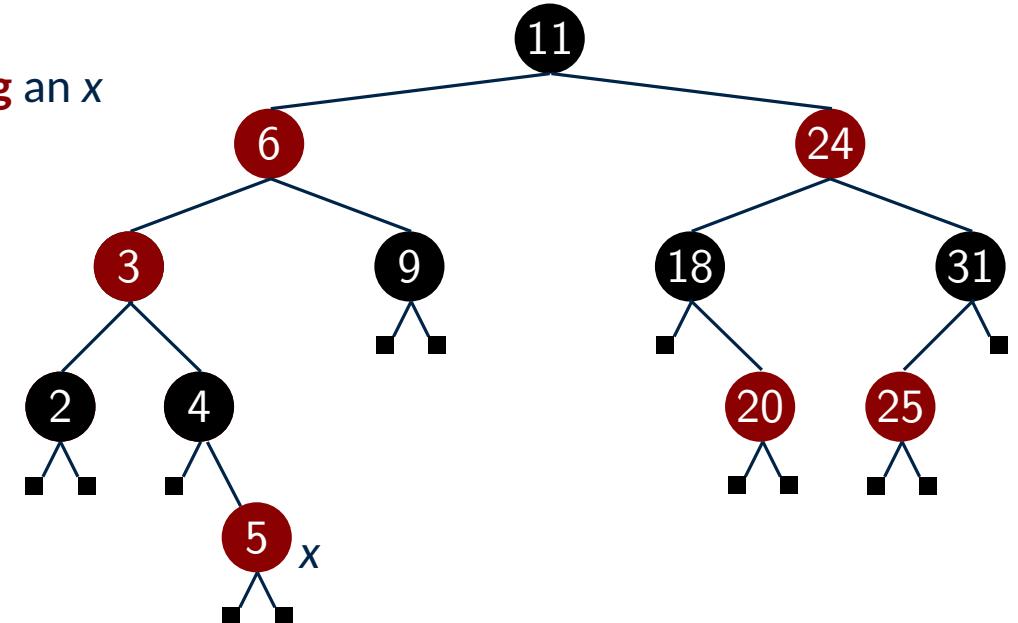
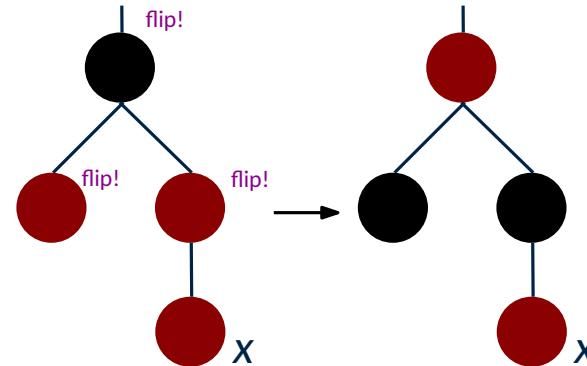
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



Der Fälle "x rechter Enkelknoten" und "x linker Enkelknoten" sind symmetrisch

# behebeRotverletzung(x): Fall 1 - Umfärbken

Ausgangssituation:

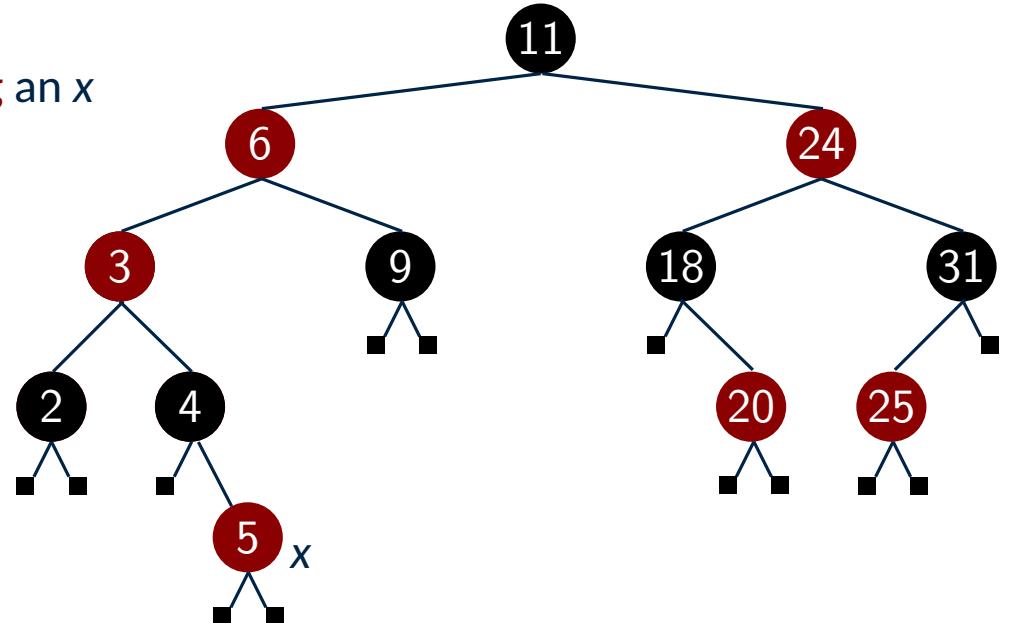
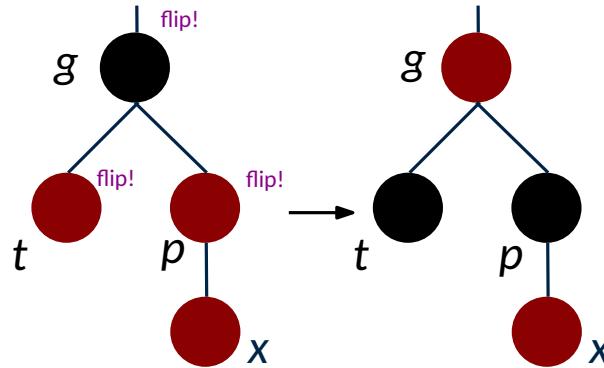
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



Der Fälle "x rechter Enkelknoten" und "x linker Enkelknoten" sind symmetrisch

Bemerkung: 1. Nach dem Umfärbken kann die einzige **Rotverletzung** am Großelter g von x sein.

- Rotverletzung** an x aufgelöst.
- Elter und Tankel **schwarz** umgefärbt → trivial keine **Rotverletzung**
- nur der **rot** umgefärzte Großelter kann neue **Rotverletzung** erzeugen (mit Urgroßelter).

2. Wir haben keine **Tiefenverletzung** erzeugt.

- einige neue **Tiefenverletzung** müsste in einem Pfad durch g entstehen
- aber: jeder Wurzel-Blatt-Pfad durch g sieht die gleiche Anzahl **schwarzer Knoten** (vorher: g, nachher: p/t)

# behebeRotverletzung(x): Fall 1 - Umfärbken

Ausgangssituation:

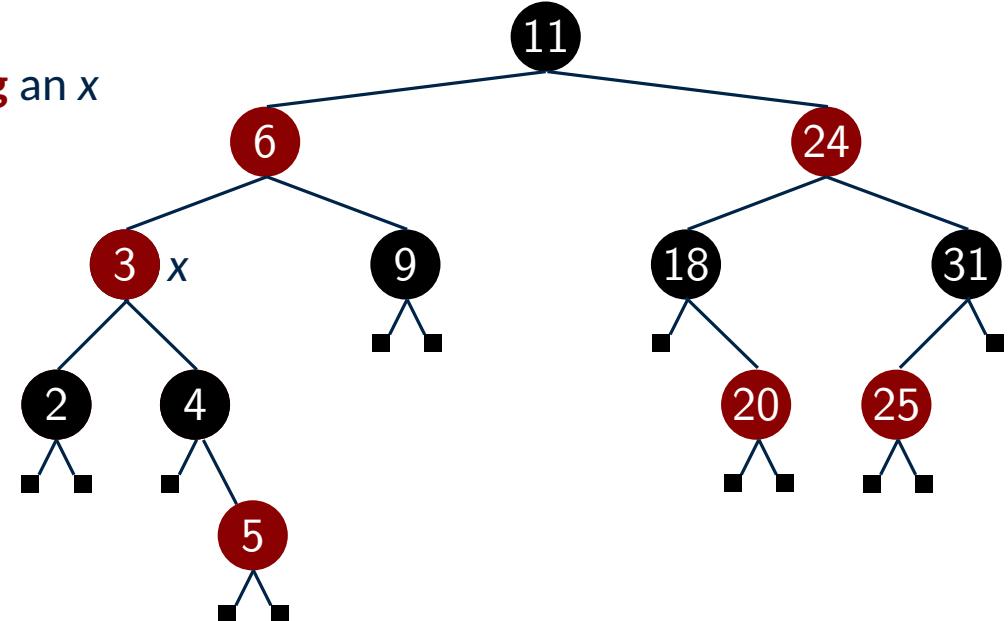
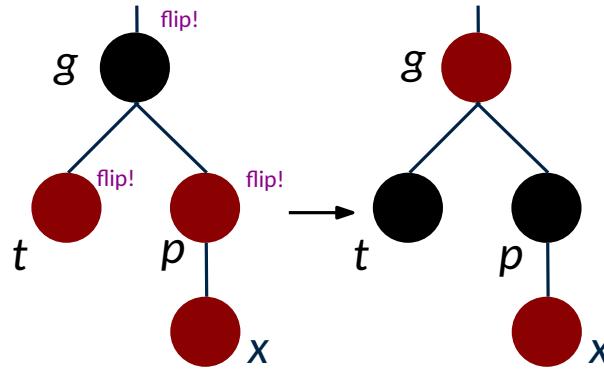
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



Der Fälle "x rechter Enkelknoten" und "x linker Enkelknoten" sind symmetrisch

Wir setzen x um auf g.

Bemerkung: 1. Nach dem Umfärbken kann die einzige **Rotverletzung** am Großelter g von x sein.

- Rotverletzung** an x aufgelöst.
- Elter und Tankel **schwarz** umgefärbt → trivial keine **Rotverletzung**
- nur der **rot** umgefärzte Großelter kann neue **Rotverletzung** erzeugen (mit Urgroßelter).

2. Wir haben keine **Tiefenverletzung** erzeugt.

- einige neue **Tiefenverletzung** müsste in einem Pfad durch g entstehen
- aber: jeder Wurzel-Blatt-Pfad durch g sieht die gleiche Anzahl **schwarzer Knoten** (vorher: g, nachher: p/t)

# behebeRotverletzung(x): Fall 1 - Umfärbken

Ausgangssituation:

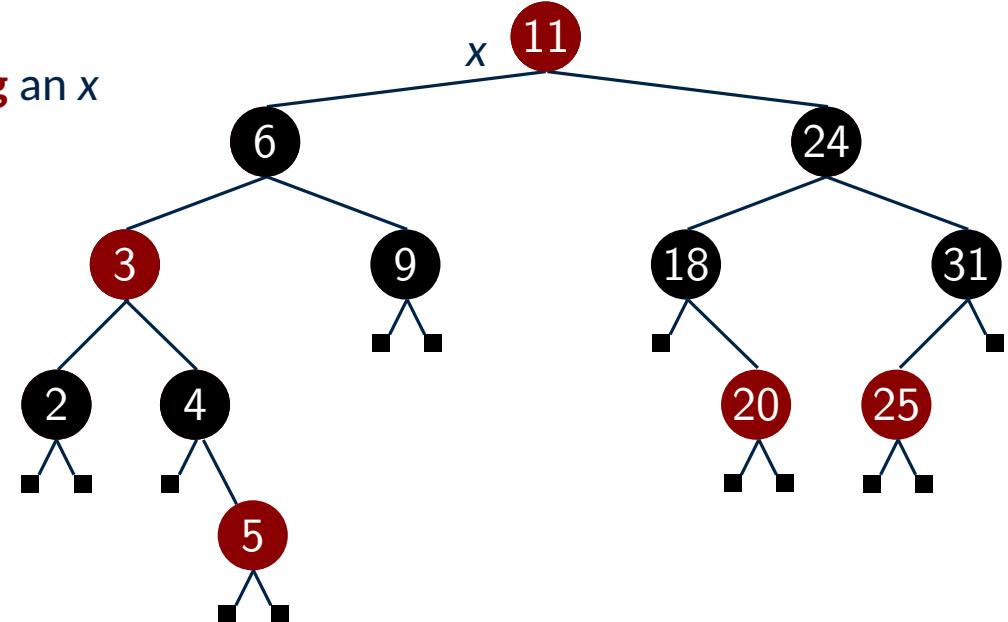
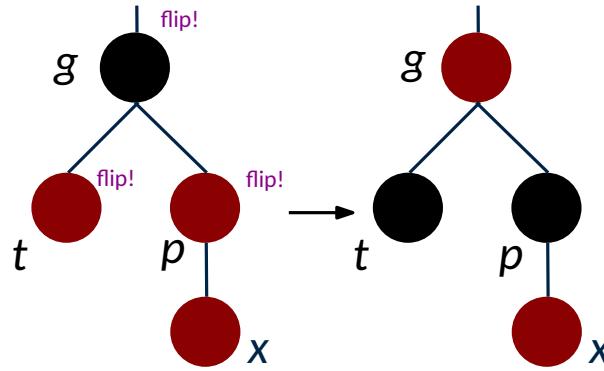
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



Der Fälle "x rechter Enkelknoten" und "x linker Enkelknoten" sind symmetrisch

Wir setzen x um auf g.

Bemerkung: 1. Nach dem Umfärbken kann die einzige **Rotverletzung** am Großelter g von x sein.

- Rotverletzung** an x aufgelöst.
- Elter und Tankel **schwarz** umgefärbt → trivial keine **Rotverletzung**
- nur der **rot** umgefärzte Großelter kann neue **Rotverletzung** erzeugen (mit Urgroßelter).

2. Wir haben keine **Tiefenverletzung** erzeugt.

- einige neue **Tiefenverletzung** müsste in einem Pfad durch g entstehen
- aber: jeder Wurzel-Blatt-Pfad durch g sieht die gleiche Anzahl **schwarzer Knoten** (vorher: g, nachher: p/t)

# behebeRotverletzung(x): Fall 1 - Umfärbken

Ausgangssituation:

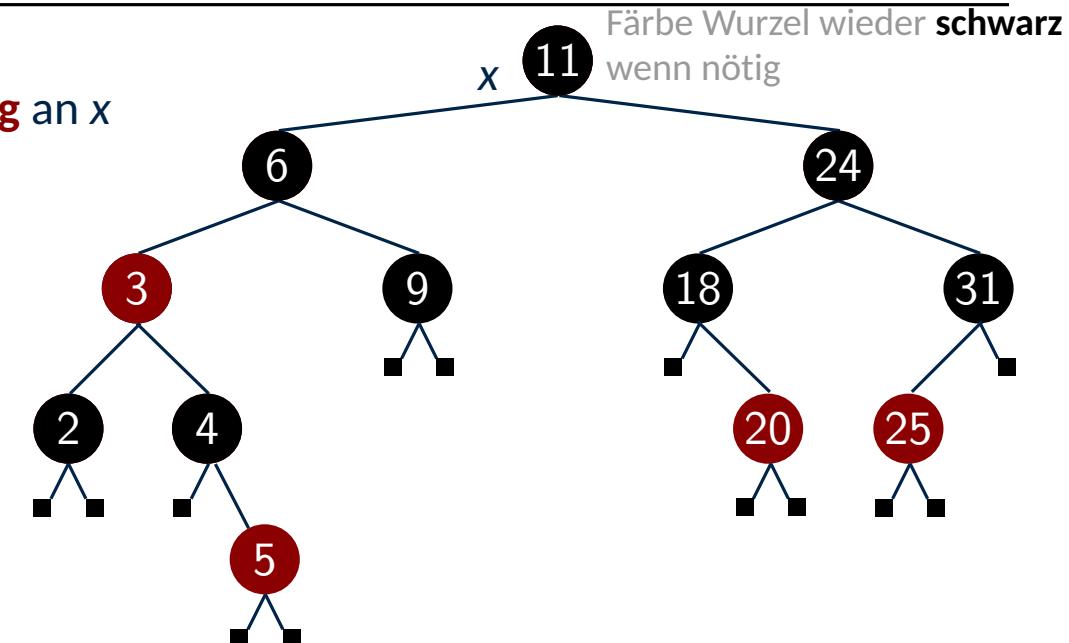
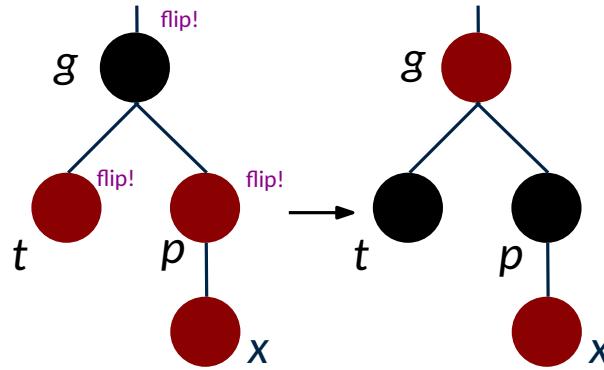
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 1: Tankel t von x ist **rot**

Wir färben um:

- Elter und Tankel: **rot** → **schwarz**
- Großelter: **schwarz** → **rot**



Der Fälle "x rechter Enkelknoten" und "x linker Enkelknoten" sind symmetrisch

Wir setzen x um auf g.

Bemerkung: 1. Nach dem Umfärbken kann die einzige **Rotverletzung** am Großelter g von x sein.

- Rotverletzung** an x aufgelöst.
- Elter und Tankel **schwarz** umgefärbt → trivial keine **Rotverletzung**
- nur der **rot** umgefärzte Großelter kann neue **Rotverletzung** erzeugen (mit Urgroßelter).

2. Wir haben keine **Tiefenverletzung** erzeugt.

- einige neue **Tiefenverletzung** müsste in einem Pfad durch g entstehen
- aber: jeder Wurzel-Blatt-Pfad durch g sieht die gleiche Anzahl **schwarzer Knoten** (vorher: g, nachher: p/t)

# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

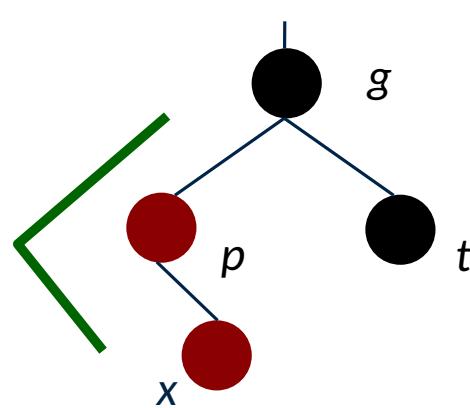
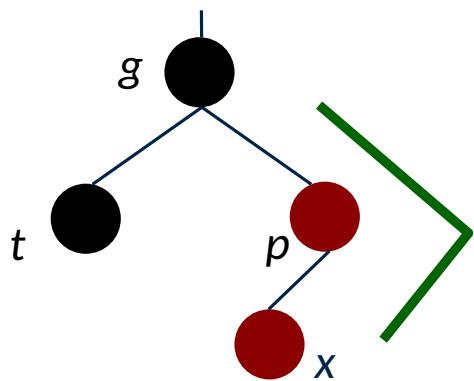
**Fall 2: Tankel t von x ist schwarz**

Wir unterscheiden in zwei Unterfälle:

**Fall 2a: x ist eckiges Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

**Fall 2b: x ist gerades Enkelkind von g**



# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist schwarz

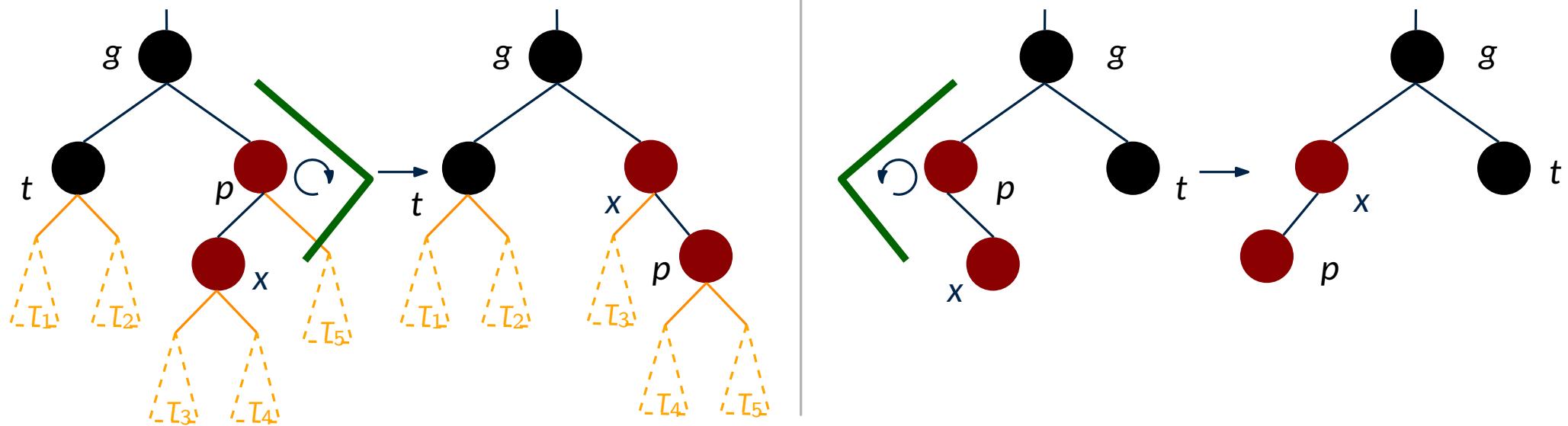
Wir unterscheiden in zwei Unterfälle:

Fall 2a: x ist **eckiges Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

Fall 2b: x ist **gerades Enkelkind von g**

Wir rotieren p in entgegengesetzte Richtung zu x ("bringen x nach oben").



# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist schwarz

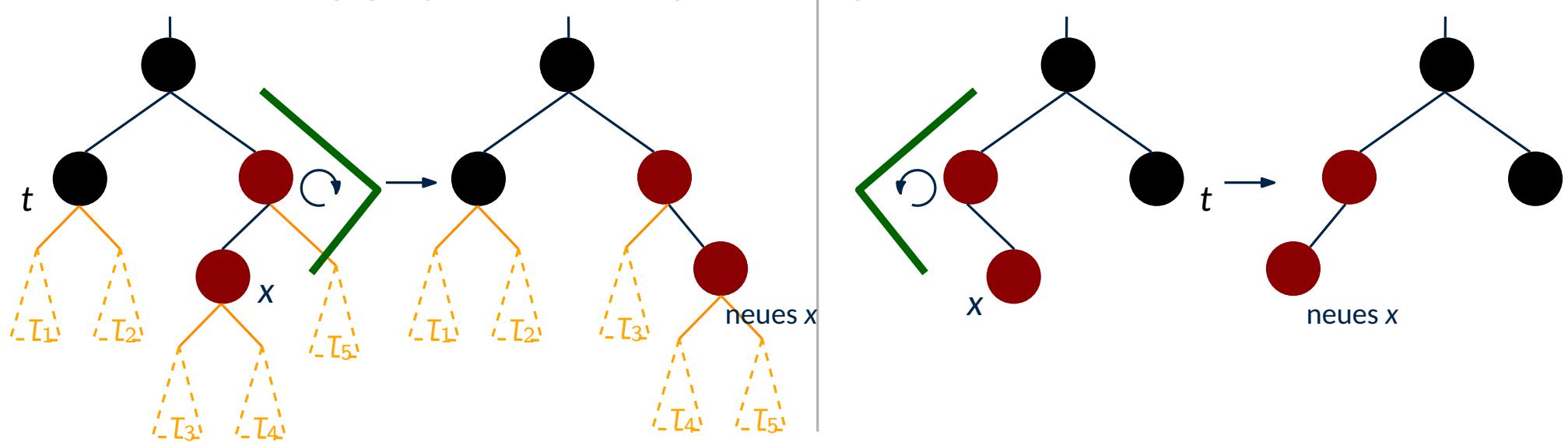
Wir unterscheiden in zwei Unterfälle:

Fall 2a: x ist **eckiges Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

Fall 2b: x ist **gerades Enkelkind von g**

Wir rotieren p in entgegengesetzte Richtung zu x ("bringen x nach oben").



Achtung: Danach wird p das neue x!

Insbesondere: Danach landen wir in Fall 2b.

# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist schwarz

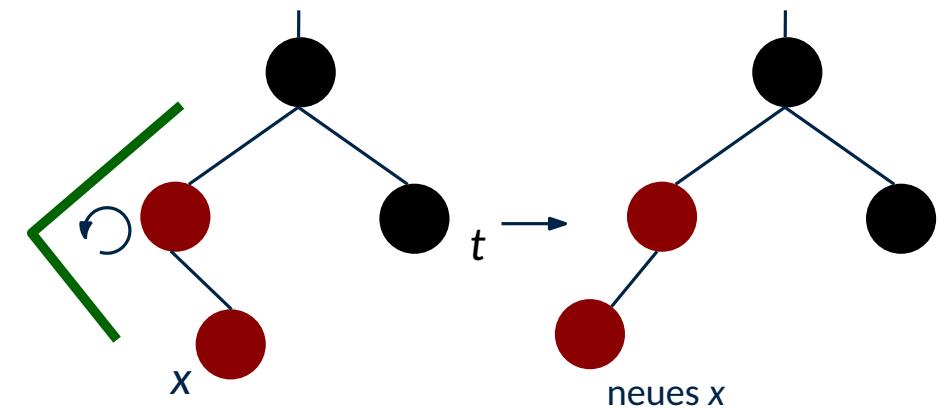
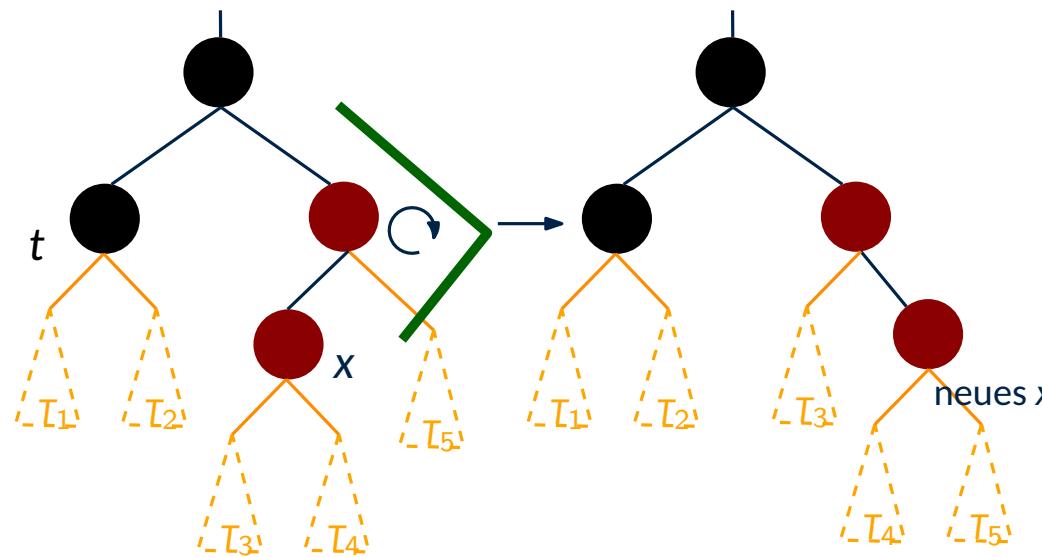
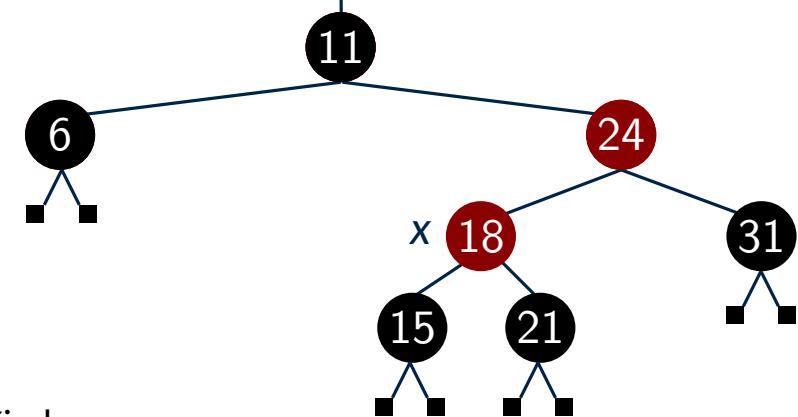
Wir unterscheiden in zwei Unterfälle:

Fall 2a: x ist **eckiges Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

Fall 2b: x ist **gerades Enkelkind von g**

Wir rotieren p in entgegengesetzte Richtung zu x ("bringen x nach oben").



Achtung: Danach wird p das neue x!

Insbesondere: Danach landen wir in Fall 2b.

# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist schwarz

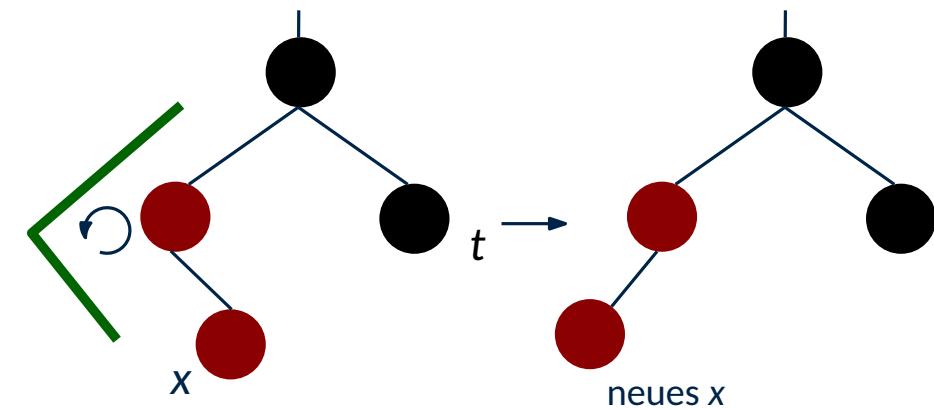
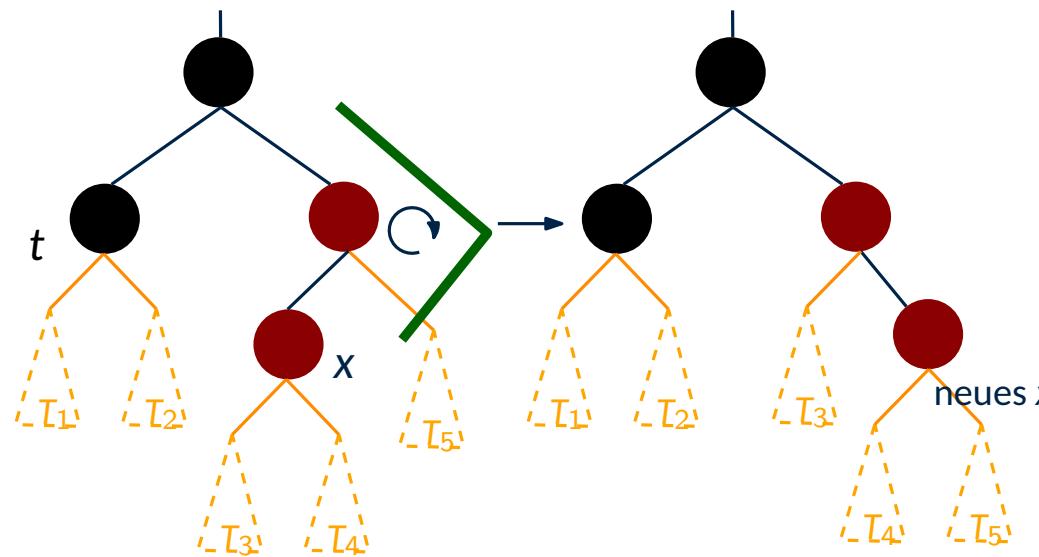
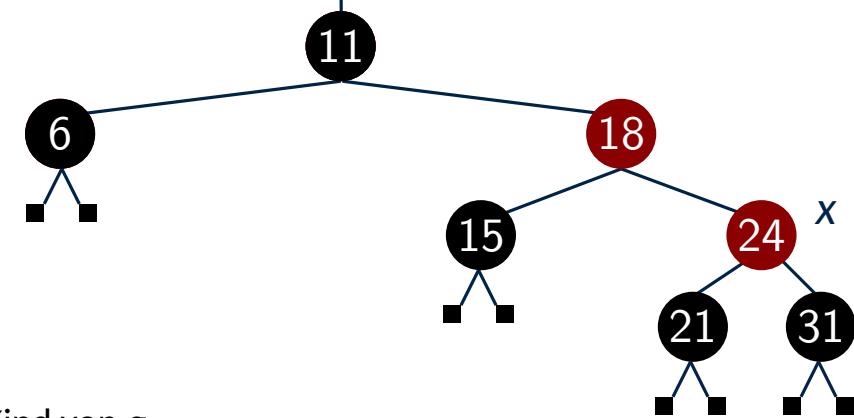
Wir unterscheiden in zwei Unterfälle:

Fall 2a: x ist **eckiges Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

Fall 2b: x ist **gerades Enkelkind von g**

Wir rotieren p in entgegengesetzte Richtung zu x ("bringen x nach oben").



Achtung: Danach wird p das neue x!

Insbesondere: Danach landen wir in Fall 2b.

# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

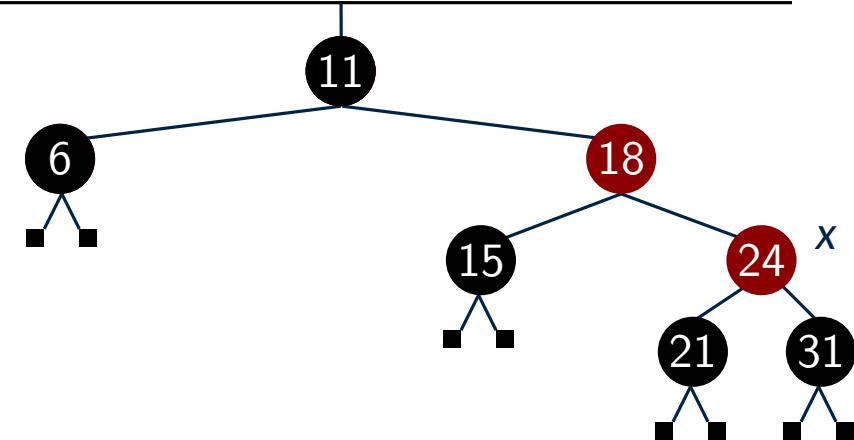
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist schwarz

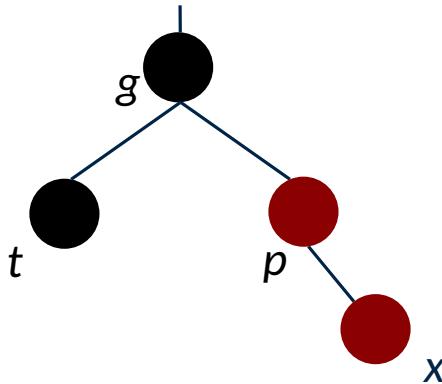
Wir unterscheiden in zwei Unterfälle:

Fall 2a: x ist eckiges Enkelkind von g



Fall 2b: x ist gerades Enkelkind von g

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g



# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

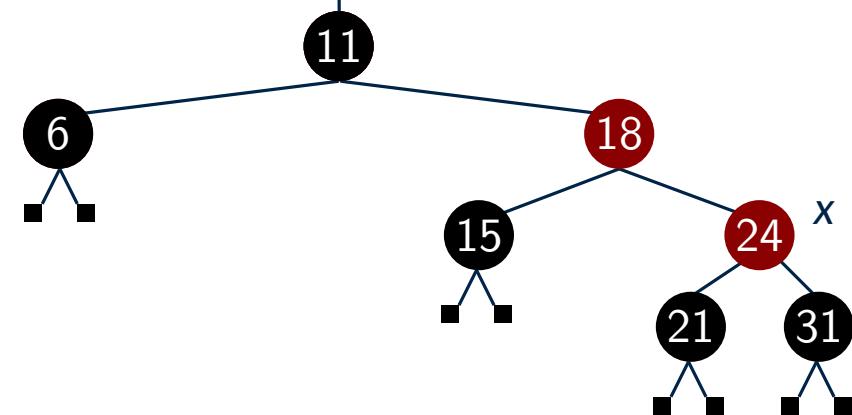
einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist **schwarz**

Wir unterscheiden in zwei Unterfälle:

Fall 2a: x ist eckiges Enkelkind von g

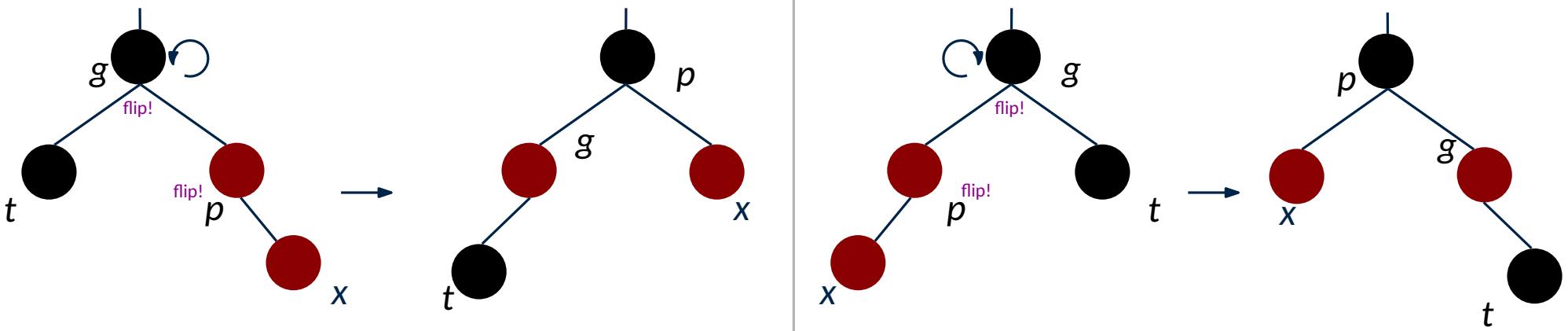


Fall 2b: x ist **gerades Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

Wir rotieren g in entgegengesetzte Richtung zu p und x ("bringen p nach oben").

Wir färben um: Großelter g wird **rot**, Elter p wird **schwarz**



Bemerkung: Die Anwendung von Fall 2b (oder Fall 2a gefolgt von Fall 2b) löst **Rotverletzung** auf.

Keine **Tiefenverletzung** eingeführt, da jedes Blatt im Teilbaum an g gleiche **Schwarztiefe** behält.

# behebeRotverletzung(x): Fall 2 - Rotation(en)

Ausgangssituation:

einige verletzte Bedingung ist eine **Rotverletzung** an x

D.h. x und Elter von x sind **rot**.

Fall 2: Tankel t von x ist **schwarz**

Wir unterscheiden in zwei Unterfälle:

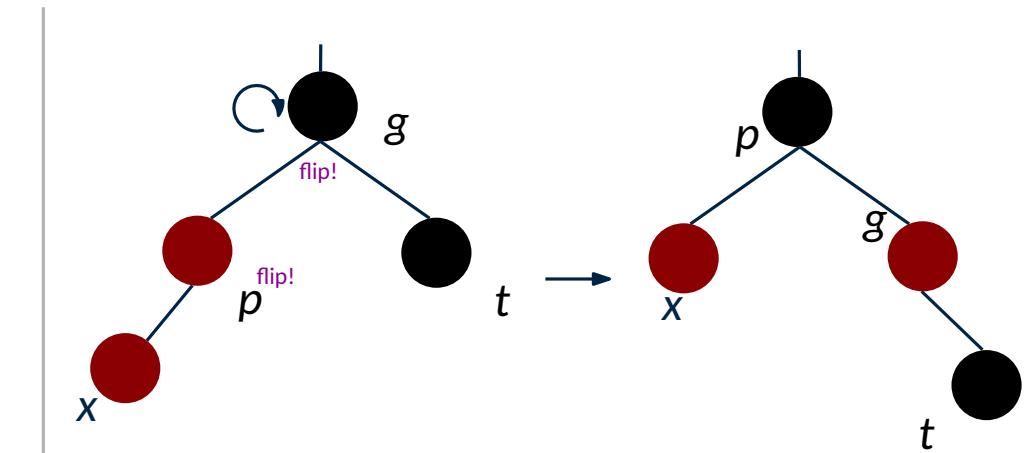
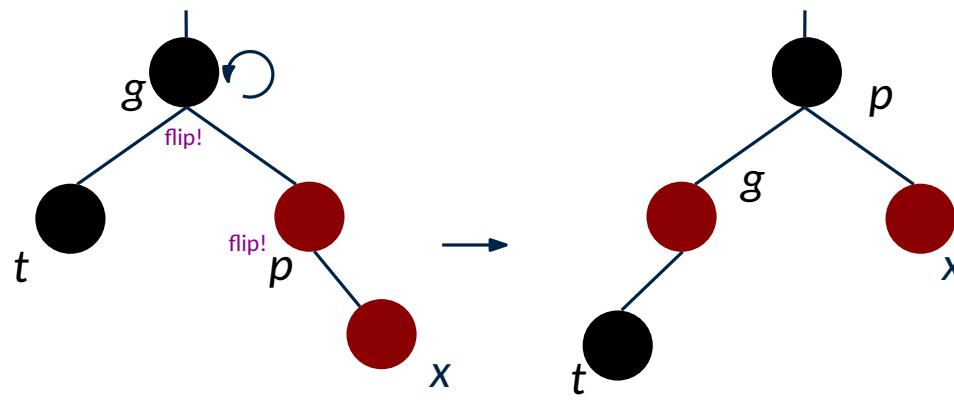
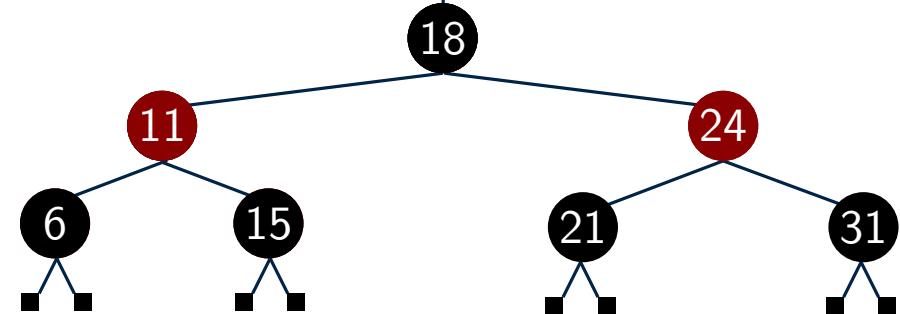
Fall 2a: x ist eckiges Enkelkind von g

Fall 2b: x ist **gerades Enkelkind von g**

D.h. x ist linkes Kind vom rechten Kind von g oder rechtes Kind vom linken Kind von g

Wir rotieren g in entgegengesetzte Richtung zu p und x ("bringen p nach oben").

Wir färben um: Großelter g wird **rot**, Elter p wird **schwarz**

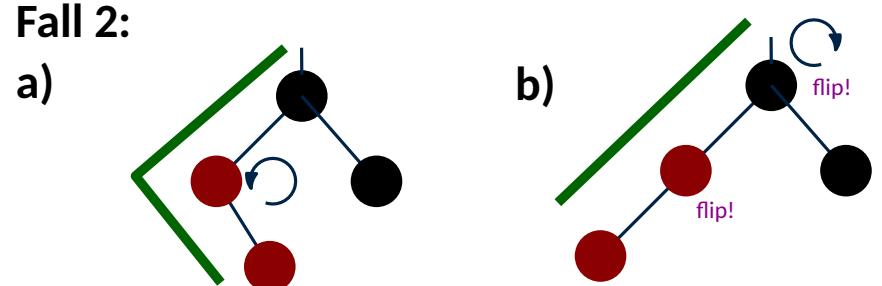
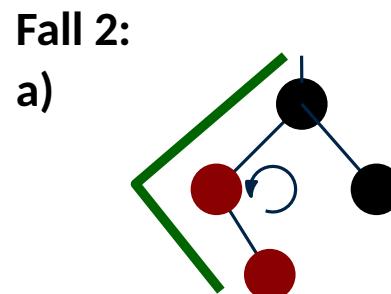
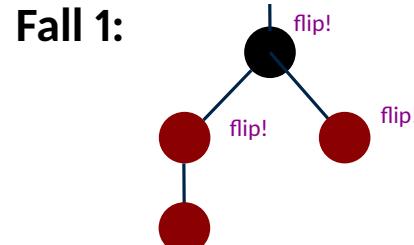


Bemerkung: Die Anwendung von Fall 2b (oder Fall 2a gefolgt von Fall 2b) löst **Rotverletzung** auf.

Keine **Tiefenverletzung** eingeführt, da jedes Blatt im Teilbaum an g gleiche **Schwarztiefe** behält.

# Beispiel zum Einfügen

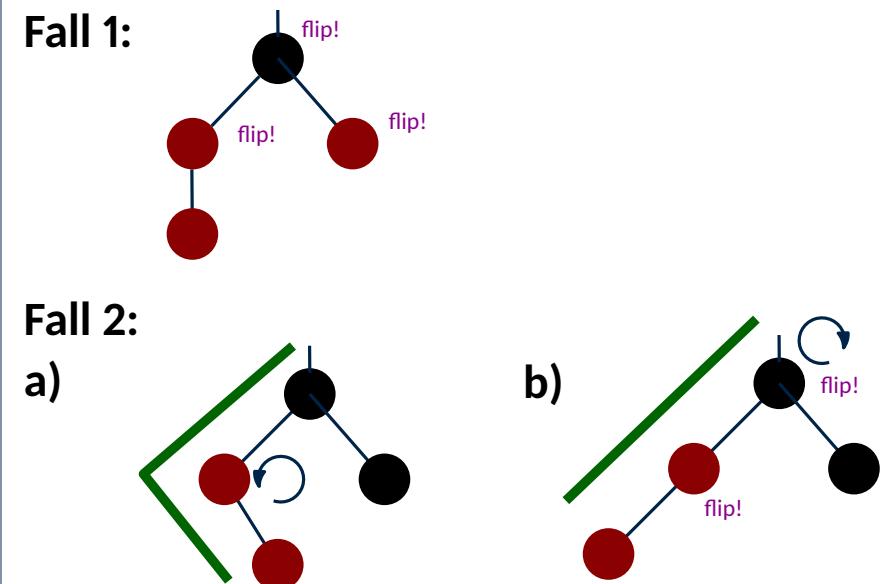
---



# Beispiel zum Einfügen

---

insert(10)

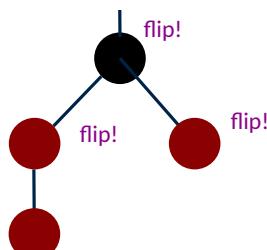


# Beispiel zum Einfügen

insert(10)

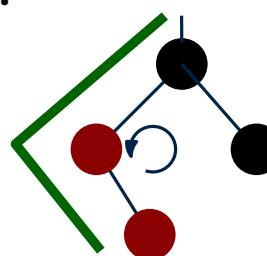


Fall 1:

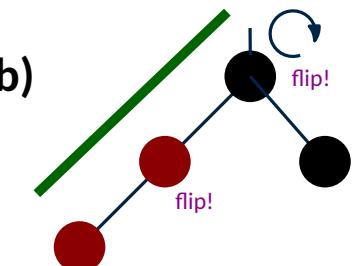


Fall 2:

a)

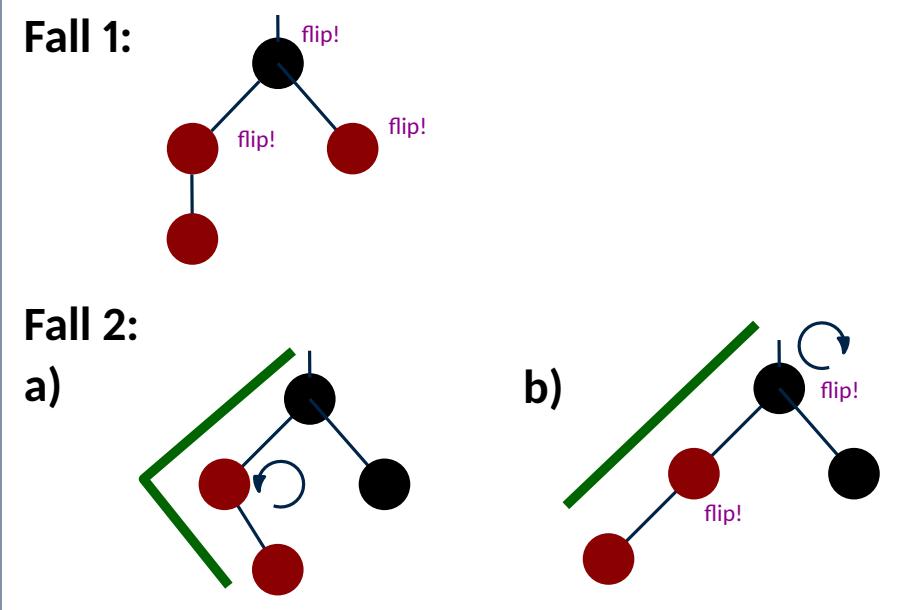


b)



# Beispiel zum Einfügen

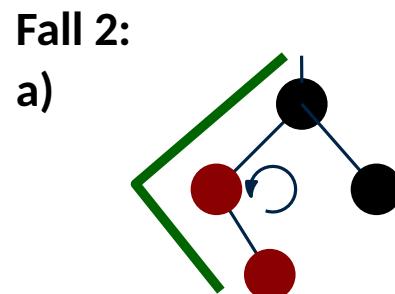
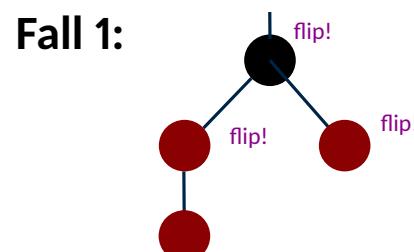
insert(10)  
→ Wurzel rot



# Beispiel zum Einfügen

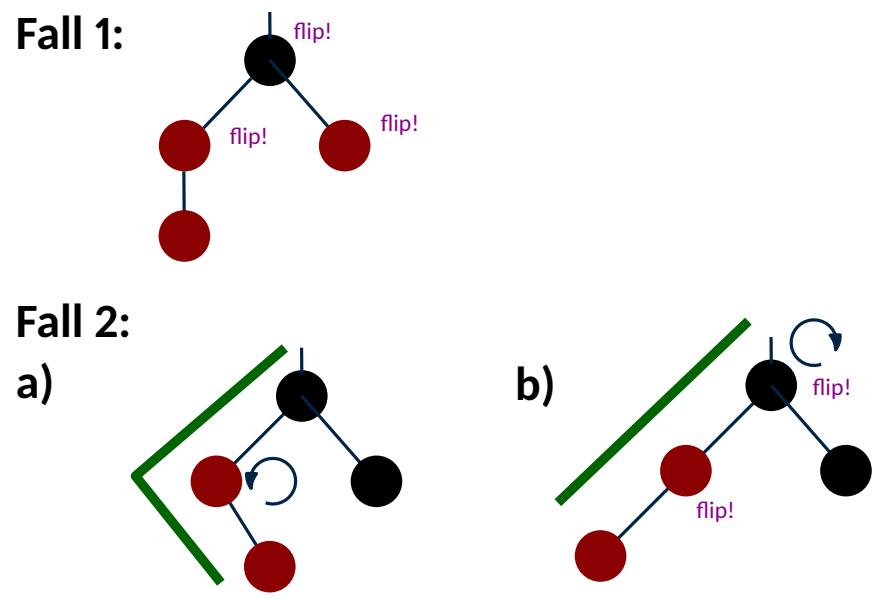
insert(10)

→ Wurzel rot



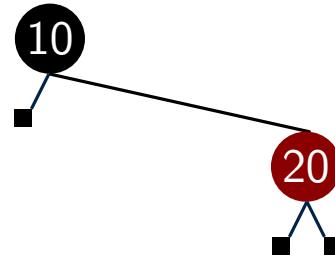
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)
```

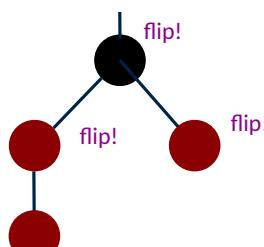


# Beispiel zum Einfügen

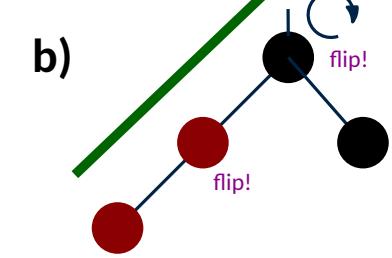
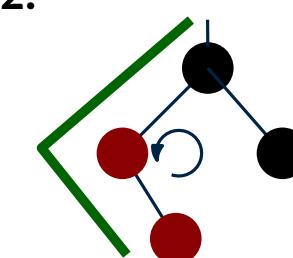
insert(10)  
→ Wurzel rot  
insert(20)



Fall 1:

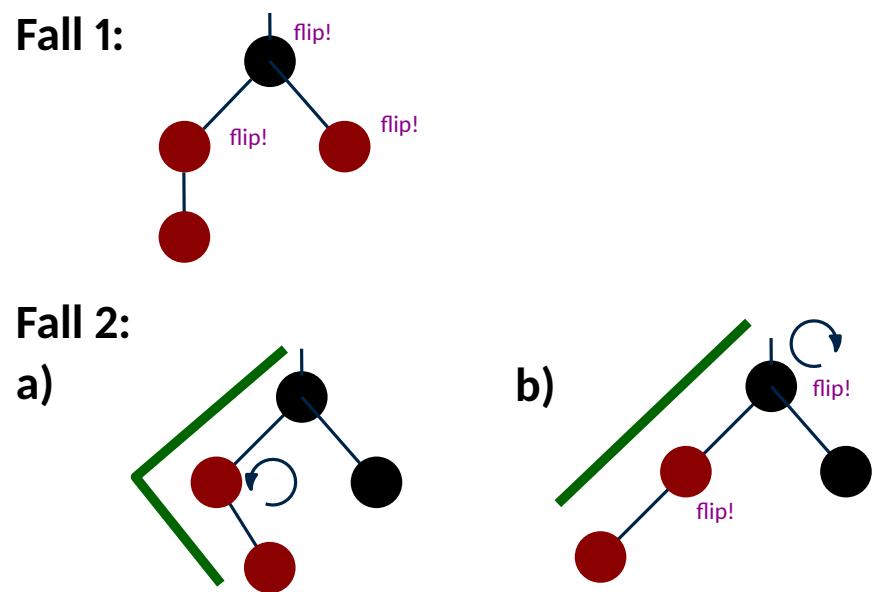
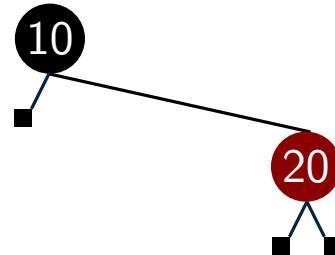


Fall 2:



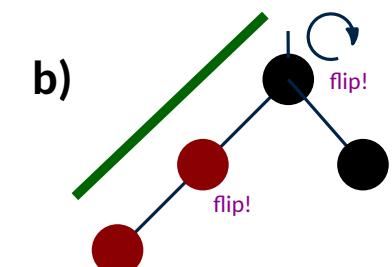
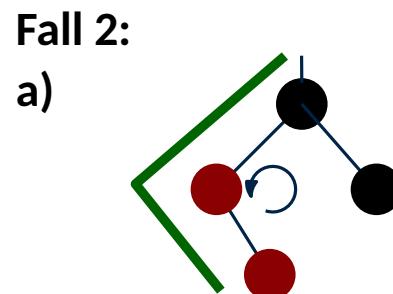
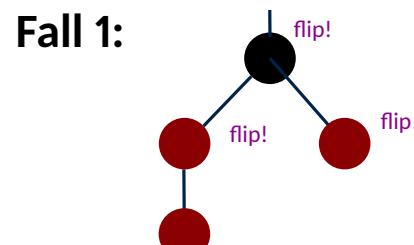
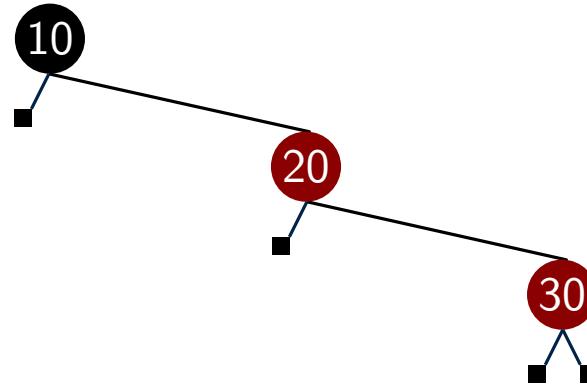
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)
```



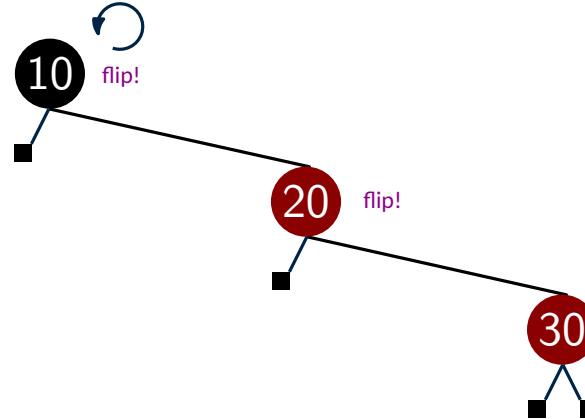
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)
```

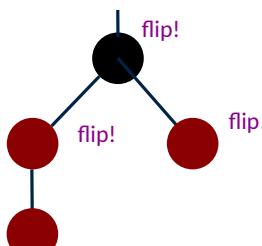


# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)
```

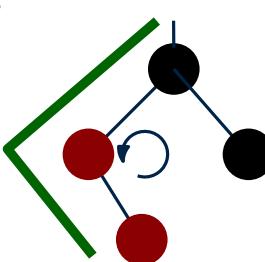


Fall 1:

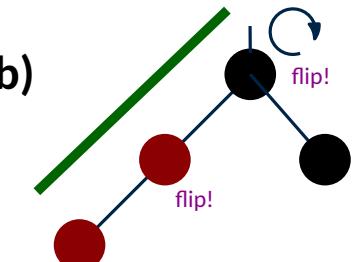


Fall 2:

a)

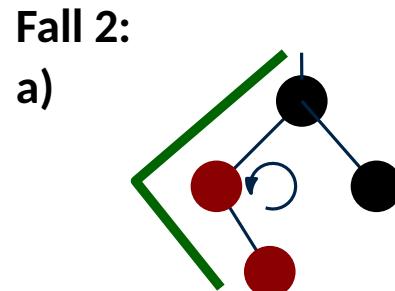
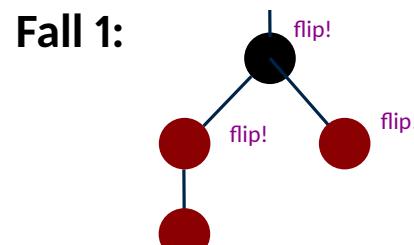
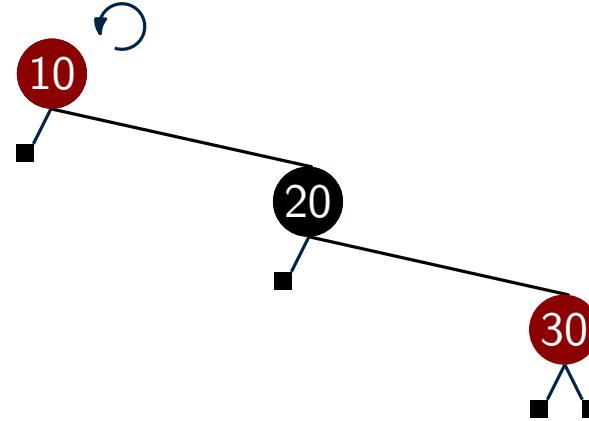


b)



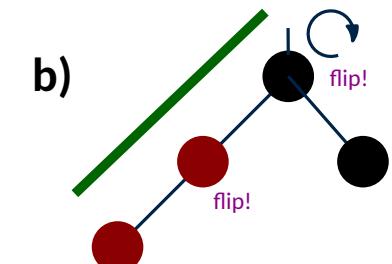
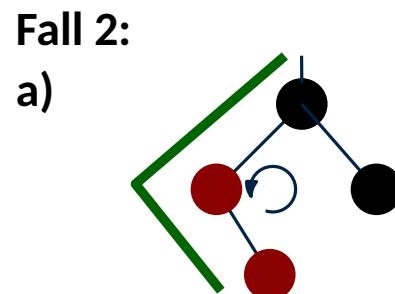
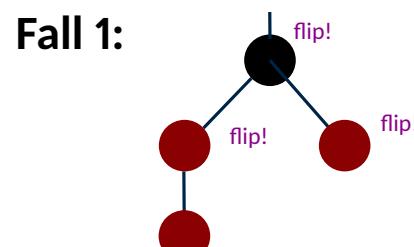
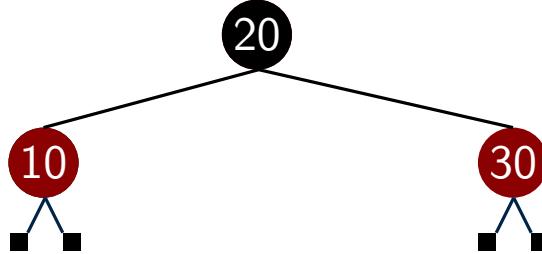
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)
```



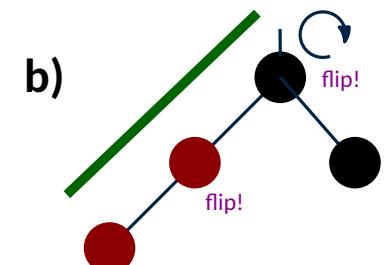
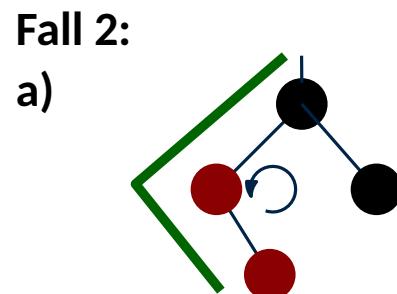
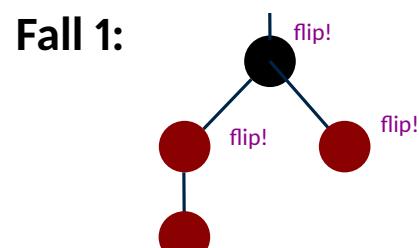
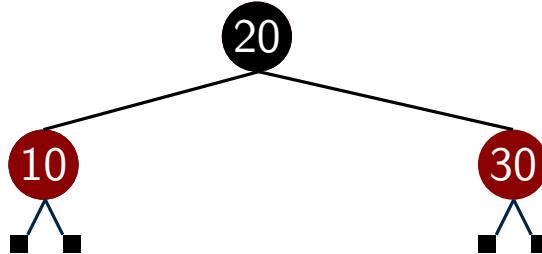
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)
```



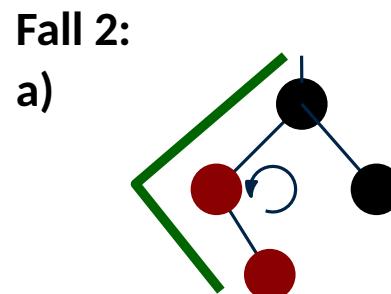
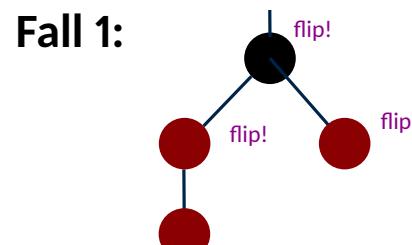
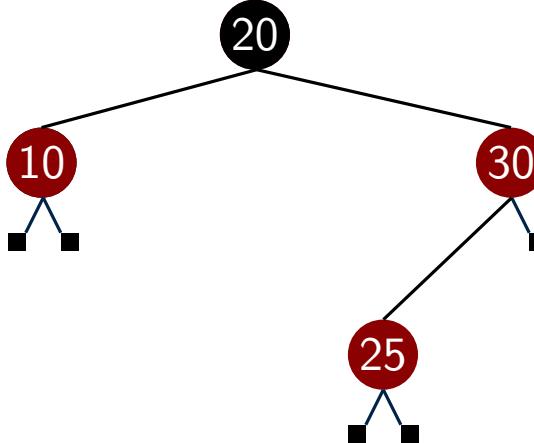
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)  
insert(25)
```



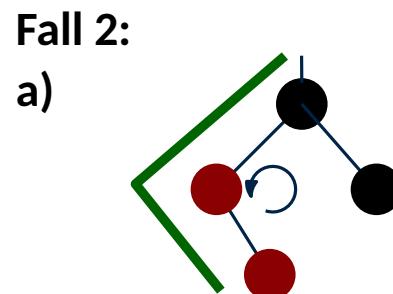
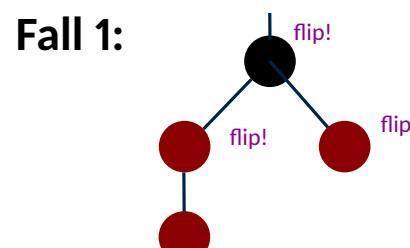
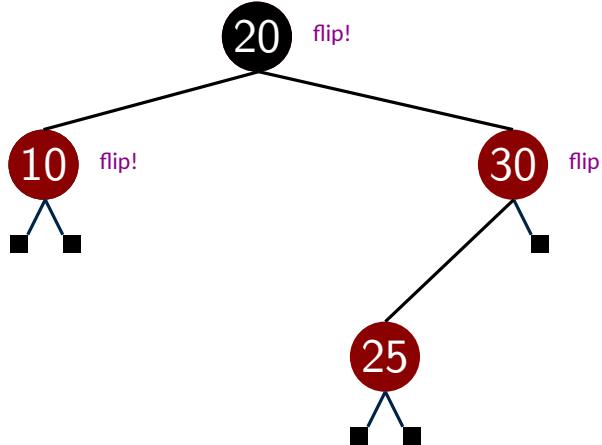
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)  
insert(25)
```



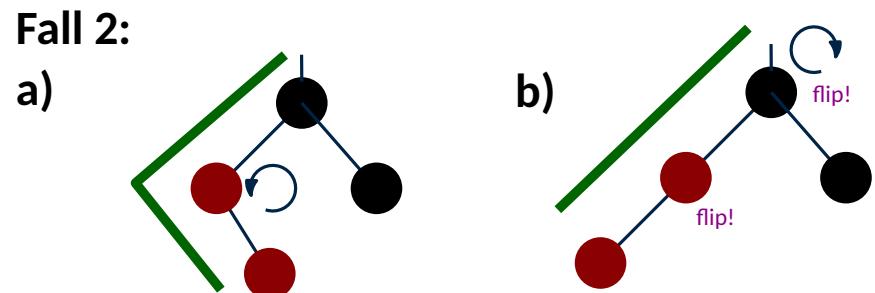
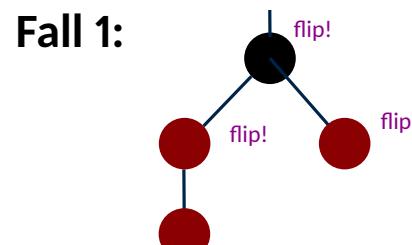
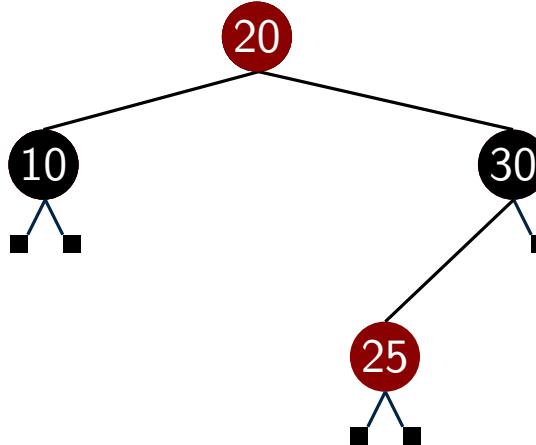
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)  
insert(25)  
→ Fall 1
```



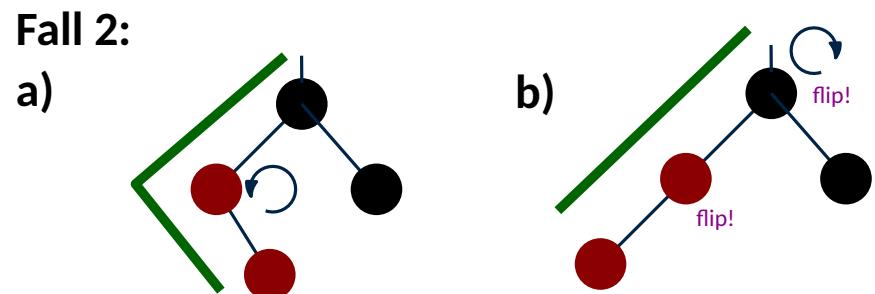
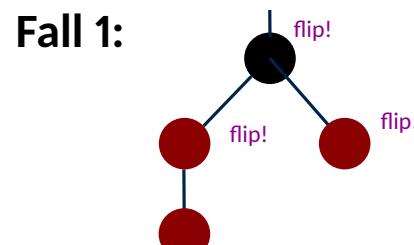
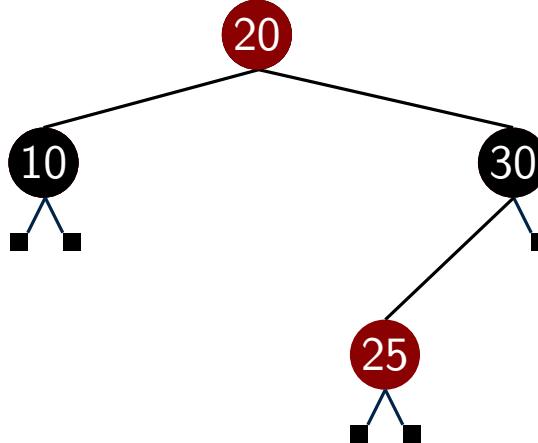
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)  
insert(25)  
→ Fall 1
```



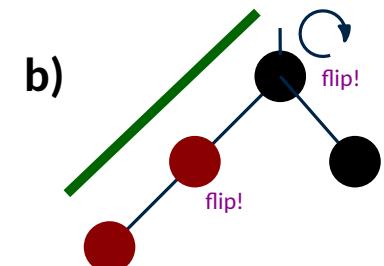
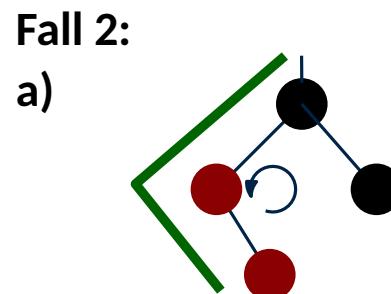
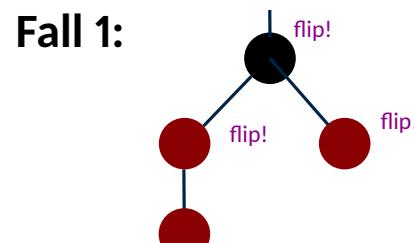
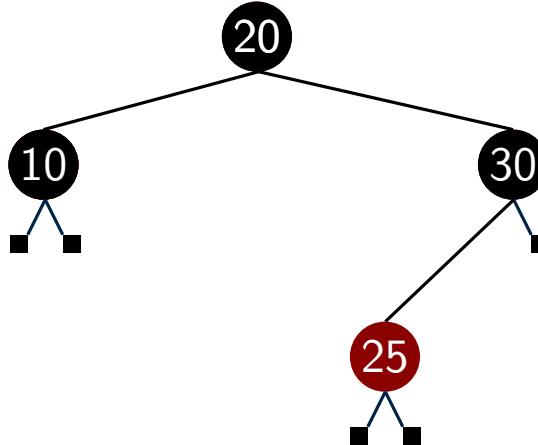
# Beispiel zum Einfügen

```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
```



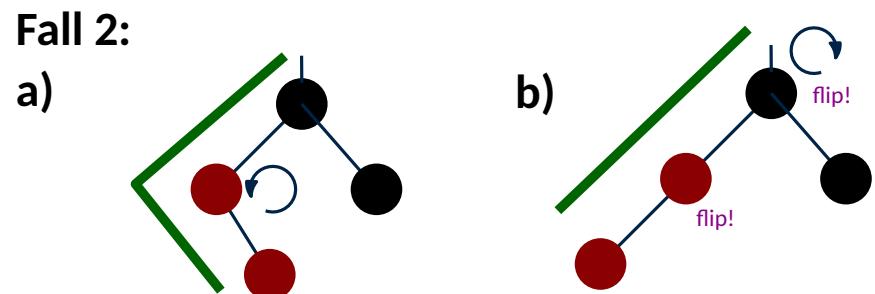
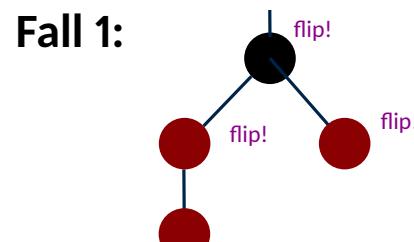
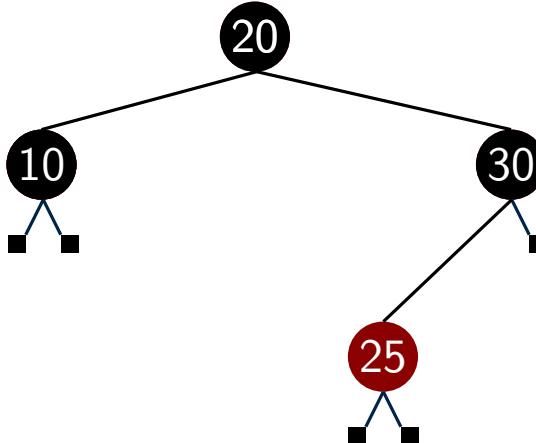
# Beispiel zum Einfügen

```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
```



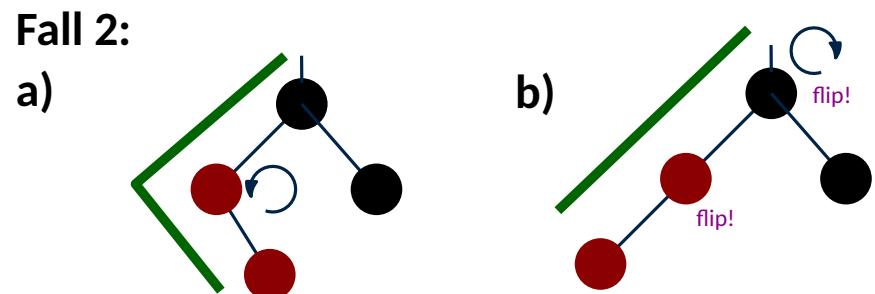
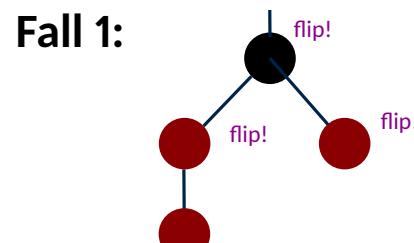
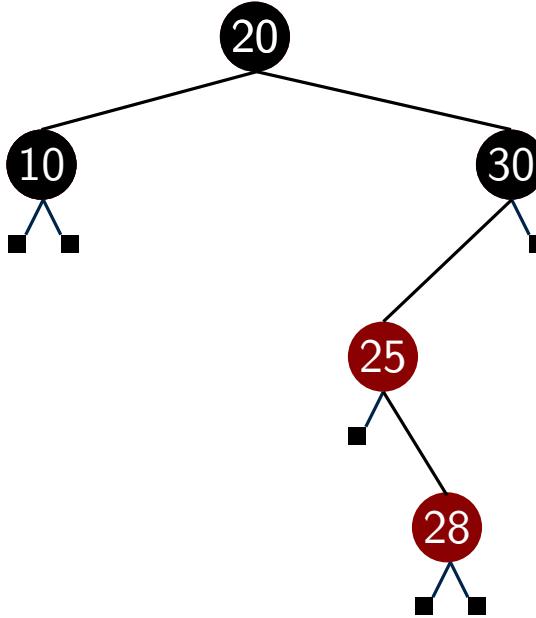
# Beispiel zum Einfügen

```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
```



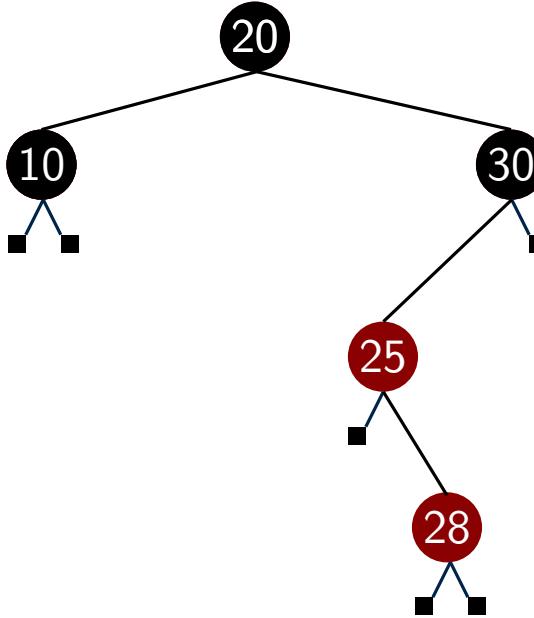
# Beispiel zum Einfügen

```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)  
insert(25)  
→ Fall 1  
→ Wurzel rot  
insert(28)
```

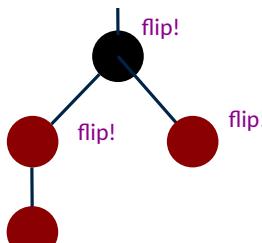


# Beispiel zum Einfügen

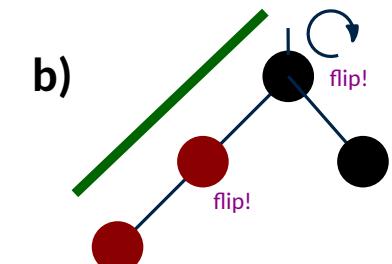
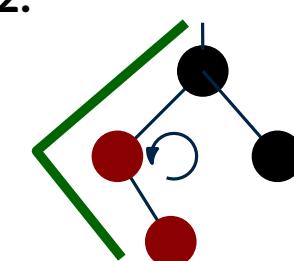
```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
  → Fall 2a)
```



Fall 1:

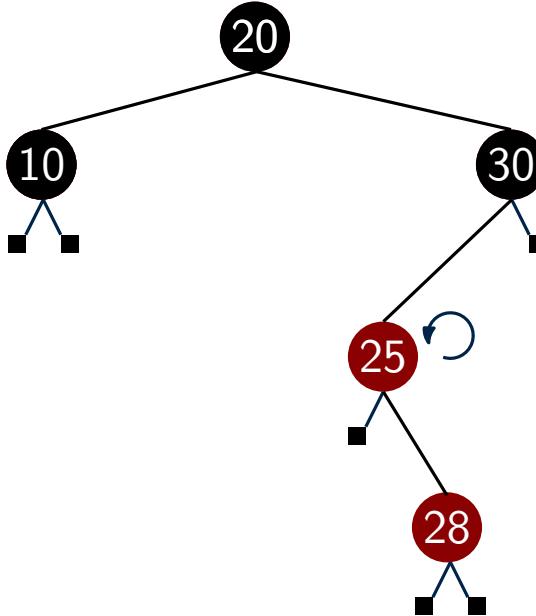


Fall 2:

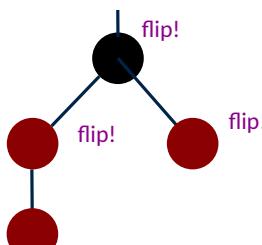


# Beispiel zum Einfügen

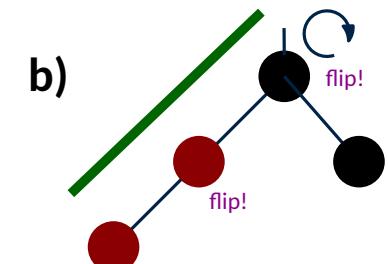
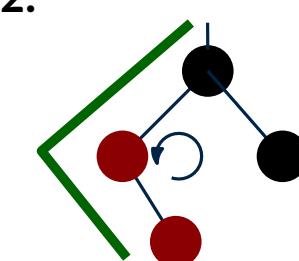
```
insert(10)  
→ Wurzel rot  
insert(20)  
insert(30)  
→ Fall 2b)  
insert(25)  
→ Fall 1  
→ Wurzel rot  
insert(28)  
→ Fall 2a)
```



Fall 1:

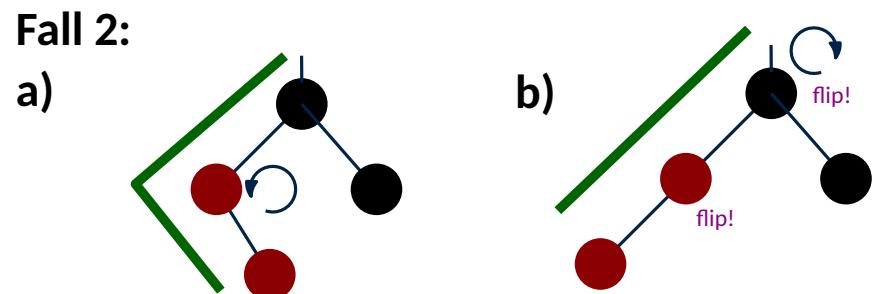
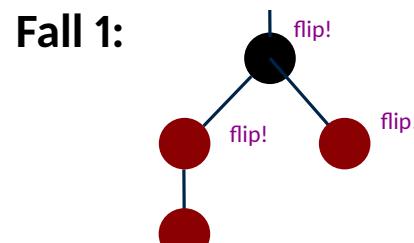
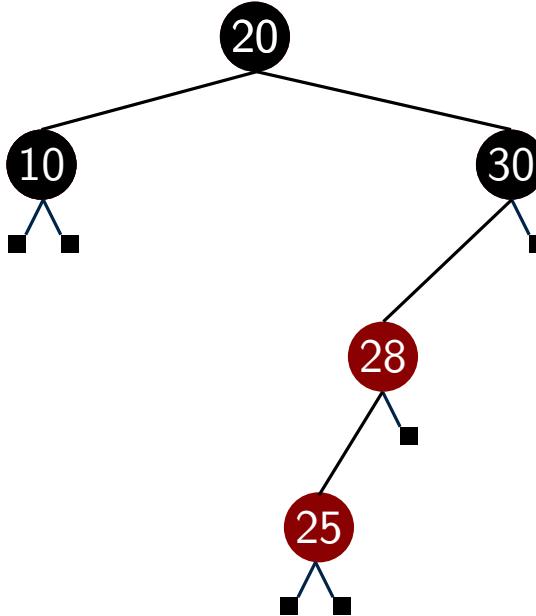


Fall 2:



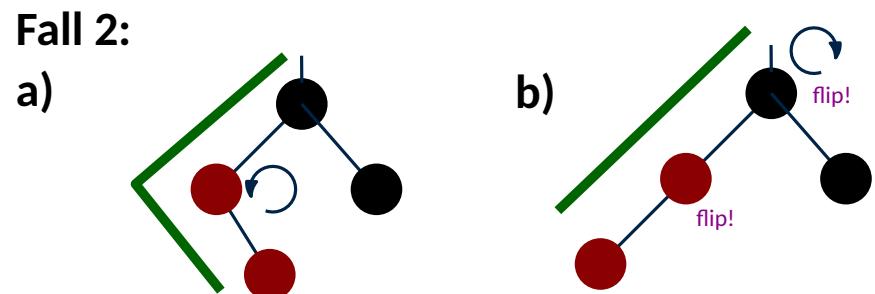
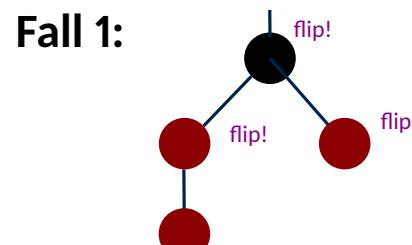
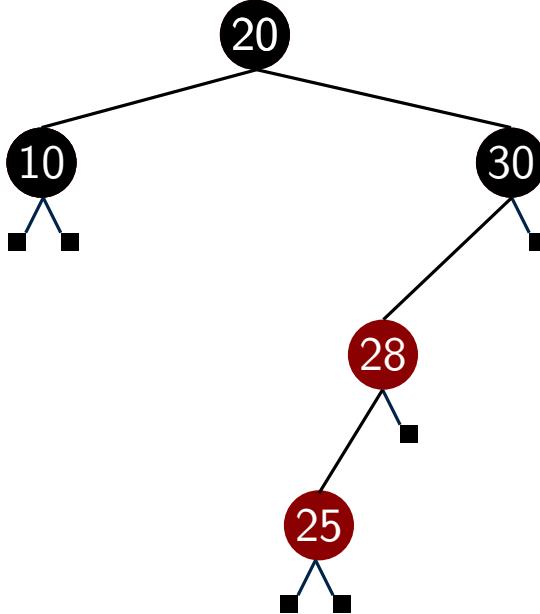
# Beispiel zum Einfügen

```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
  → Fall 2a)
```



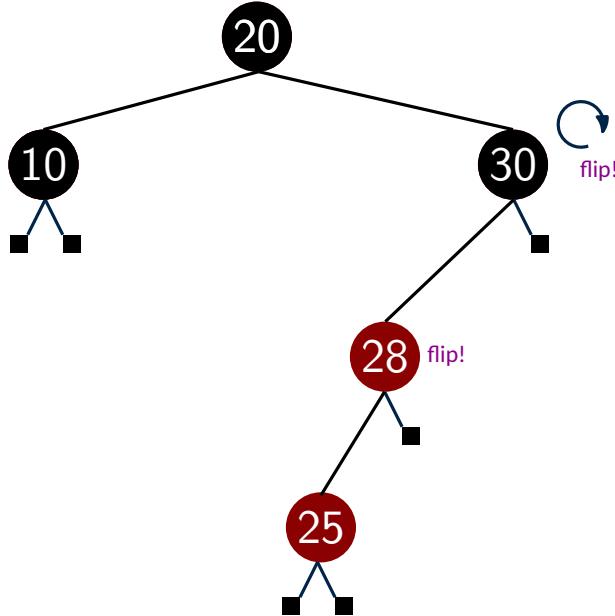
# Beispiel zum Einfügen

```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
  → Fall 2a)
  → Fall 2b)
```

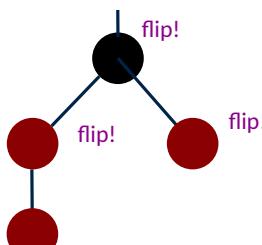


# Beispiel zum Einfügen

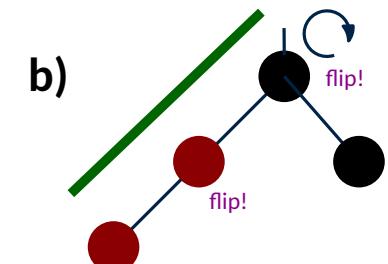
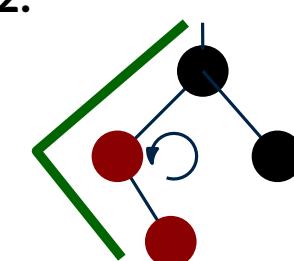
```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
  → Fall 2a)
  → Fall 2b)
```



Fall 1:

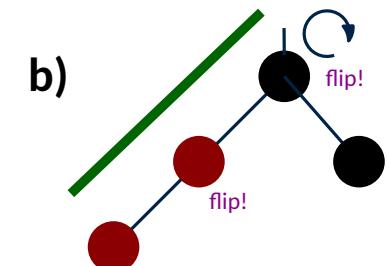
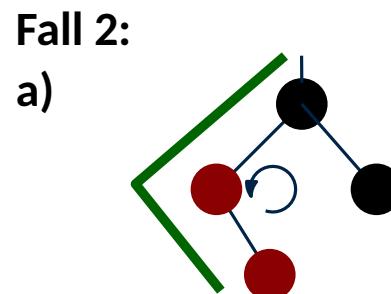
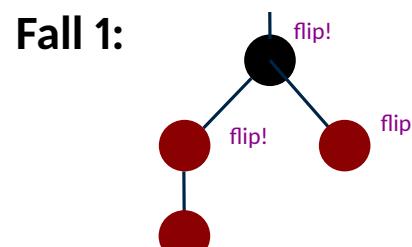
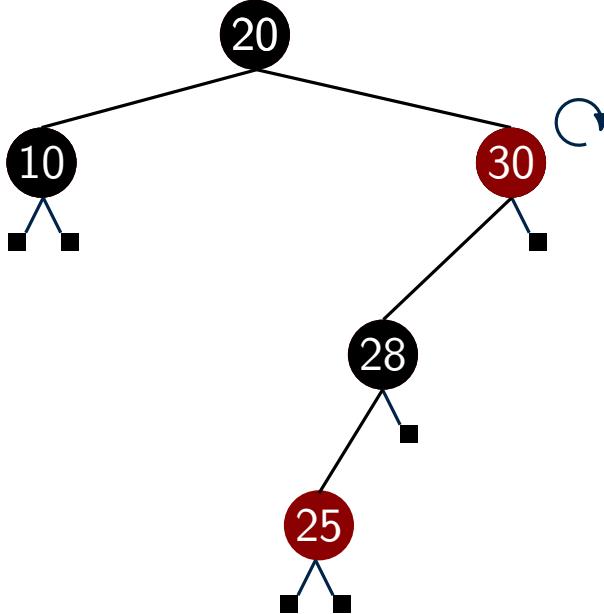


Fall 2:



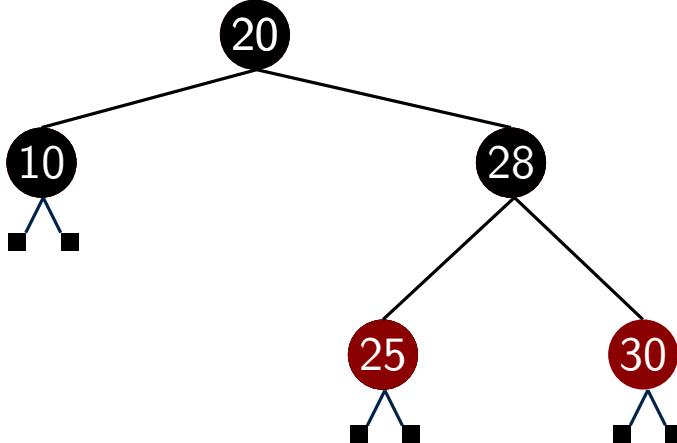
# Beispiel zum Einfügen

```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
  → Fall 2a)
  → Fall 2b)
```

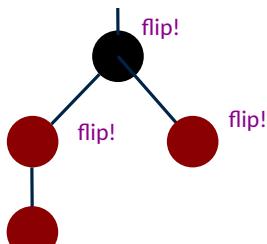


# Beispiel zum Einfügen

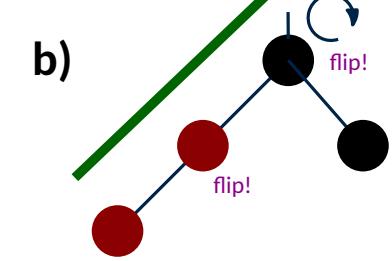
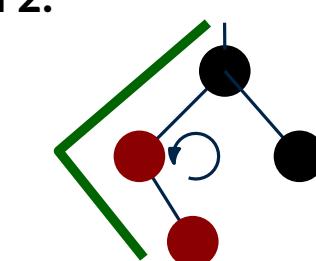
```
insert(10)
  → Wurzel rot
insert(20)
insert(30)
  → Fall 2b)
insert(25)
  → Fall 1
  → Wurzel rot
insert(28)
  → Fall 2a)
  → Fall 2b)
```



Fall 1:



Fall 2:



# Beispiel

---

↗ Tafelpräsentation

# Korrektheit und Laufzeit von insert()

```
insert(e): Füge Schlüssel  $k = \text{key}(e)$  an die entsprechende Stelle im Baum ein  $\rightsquigarrow$  Knoten x  
Färbe x rot.  
while (Rotverletzung bei x) do  
    behebeRotverletzung(x) // darf x verändern! Insbesondere: Auf Großelter setzen  
if (x ist die Wurzel) then  
    färbe x nötigenfalls schwarz
```

**Korrektheit:** behebeRotverletzung(x) hat folgende Eigenschaften:

- es fügt keine **Tiefenverletzung** ein
- es löst die **Rotverletzung** an x auf
- danach ist die einzige mögliche **Rotverletzung** am Großelter von x → ersetzt x durch den Großelter von x | **Fall 1**

spätestens wenn x die Wurzel ist (& ggf. **schwarz** gefärbt wird), ist Rot-Schwarz-Baum-Eigenschaft erfüllt

**Lemma.**

insert(e) fügt einen neuen Element e in Zeit  $O(\log n)$  ein und benutzt höchstens 2 Rotationen.

**Beweis:** Die Höhe des ursprünglichen Baumes ist  $O(\log n)$  (nach Rot-Schwarz-Baum-Eigenschaft).

⇒ Die while-Schleife terminiert nach  $O(\log n)$  Iterationen.

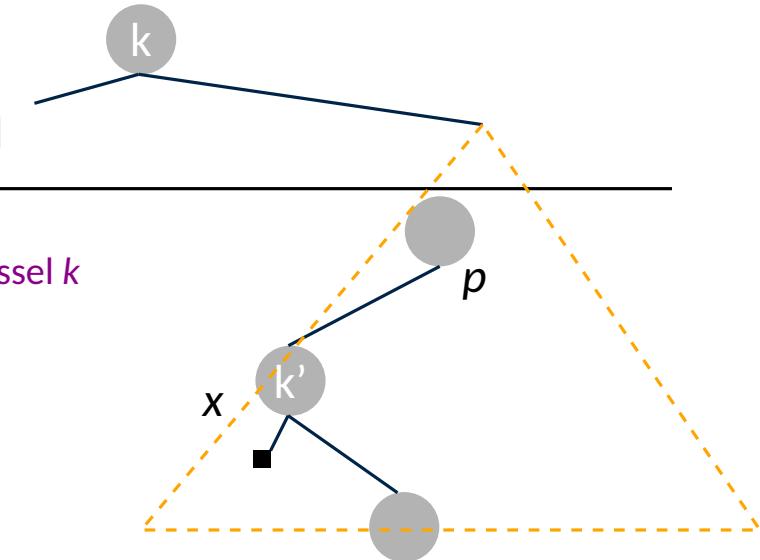
Wir führen nur Umfärbungen durch, bis Rotverletzung bereits behoben ist oder wir Fall 2 erreichen.

Nur, wenn wir Fall 2 erreichen, erfolgen Rotationen: entweder eine (Fall 2b) oder zwei (Fall 2a gefolgt von 2b).  
→ danach terminieren wir sofort. □

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$



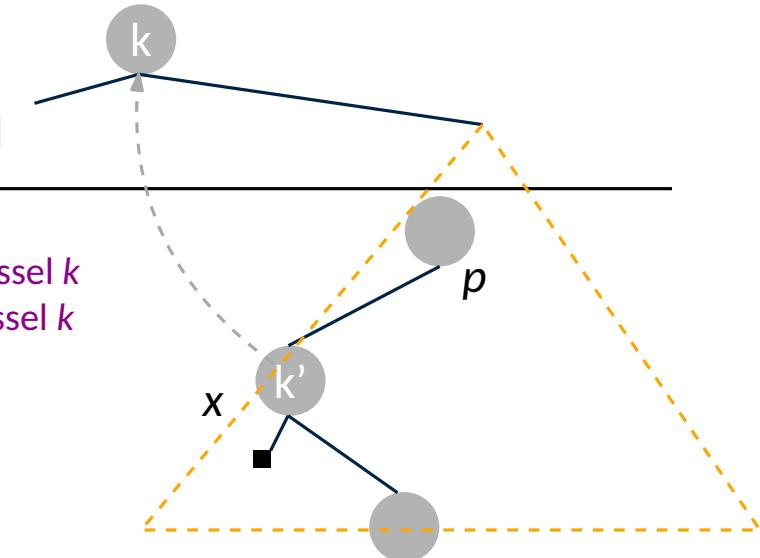
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$



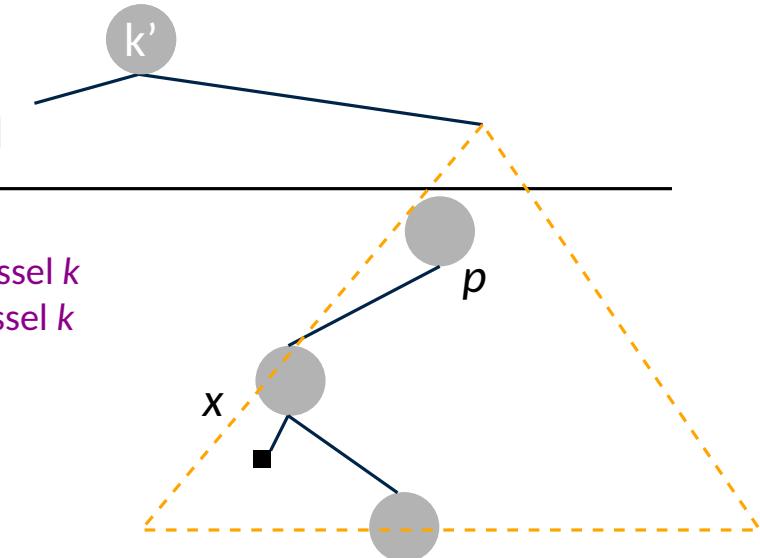
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$



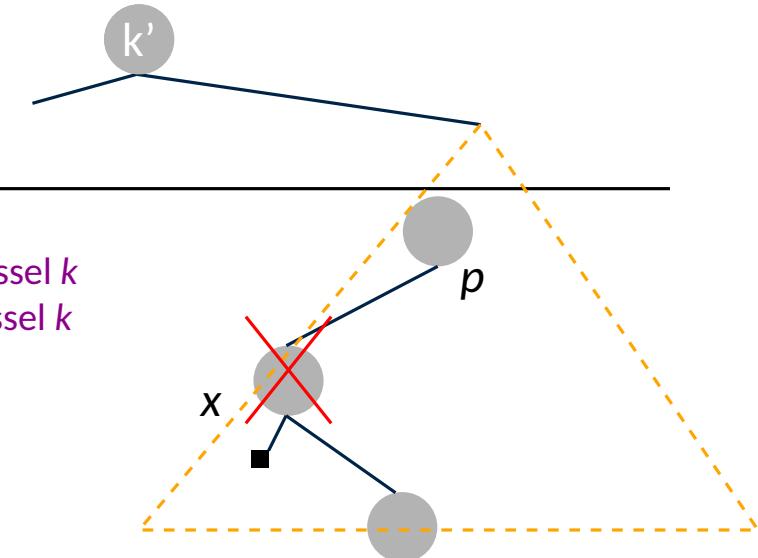
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



## Grundprinzip:

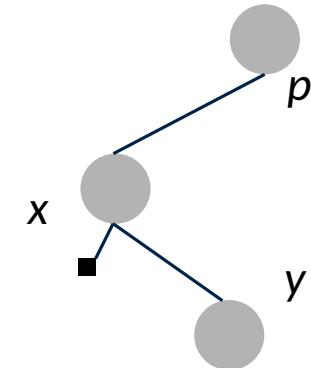
- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen

# Entfernen im Rot-Schwarz-Baum

---

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



## Grundprinzip:

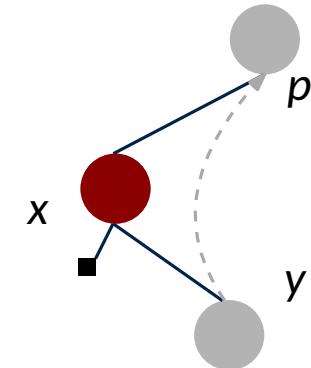
- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt

# Entfernen im Rot-Schwarz-Baum

---

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



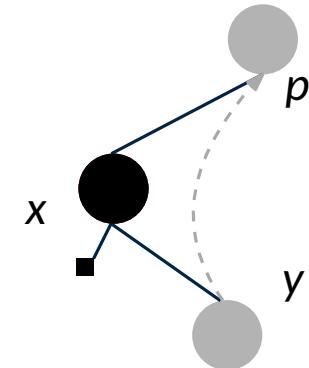
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



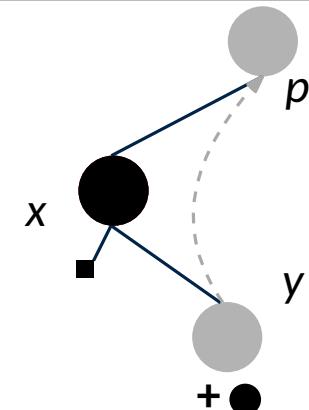
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



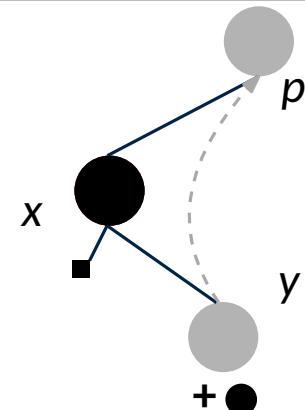
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



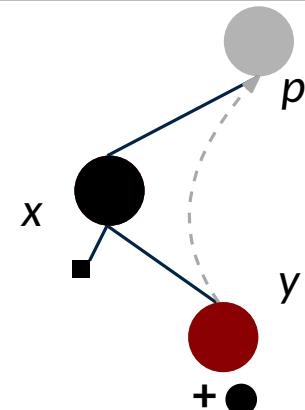
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



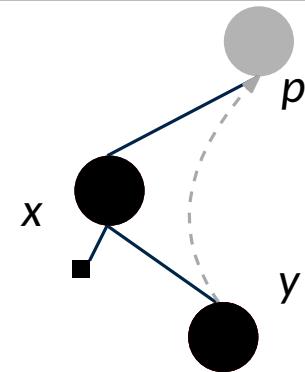
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
  - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz aufgelöst**

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



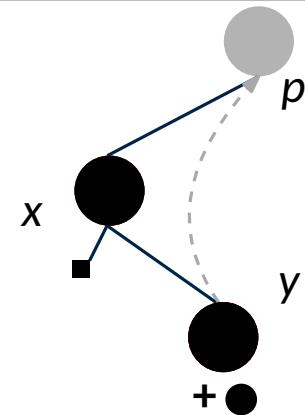
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
  - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz aufgelöst**

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



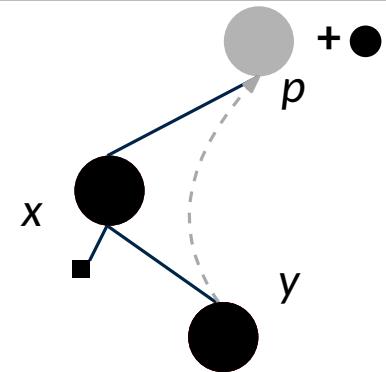
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
  - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz** aufgelöst
  - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



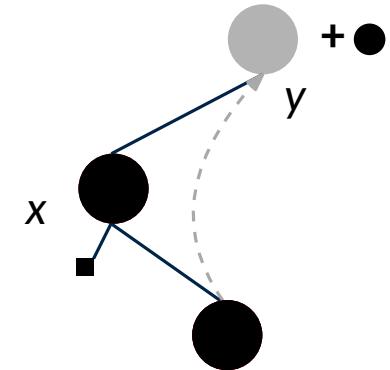
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
  - Wenn  $x$  **rot** ist, können wir dies einfach tun.
  - Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**
    - Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
    - wir sagen:  $y$  benötigt ein **Extra-Schwarz**
  - **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
    - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz** aufgelöst
    - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



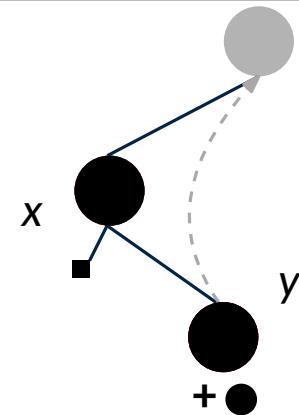
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**
  - Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
  - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz** aufgelöst
  - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben  
→ fahre rekursiv mit Elter  $p$  als  $y$  fort.

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



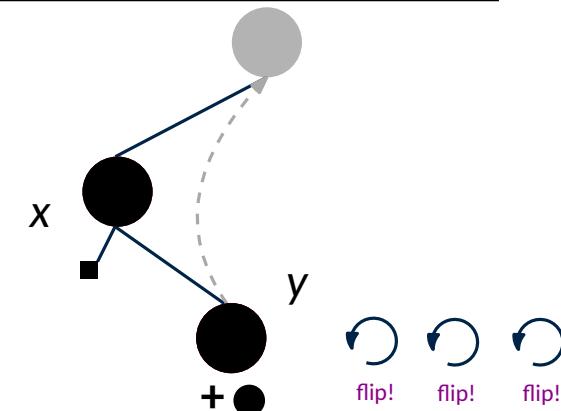
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
  - Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
  - Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
  - wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
  - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz aufgelöst**
  - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben  
→ fahre rekursiv mit Elter  $p$  als  $y$  fort.
  - wenn das nicht möglich ist, löse das **Extra-Schwarz** durch bis zu drei Rotationen + Umfärbung auf

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



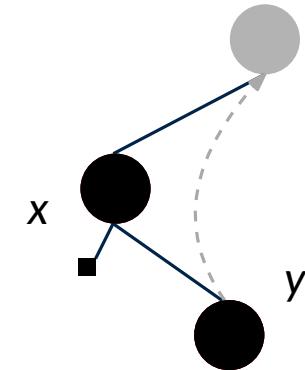
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
  - Wenn  $x$  **rot** ist, können wir dies einfach tun.
  - Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**
    - Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
    - wir sagen:  $y$  benötigt ein **Extra-Schwarz**
  - **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
    - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz** aufgelöst
    - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben
      - fahre rekursiv mit Elter  $p$  als  $y$  fort.
    - wenn das nicht möglich ist, löse das **Extra-Schwarz** durch bis zu drei Rotationen + Umfärbung auf

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



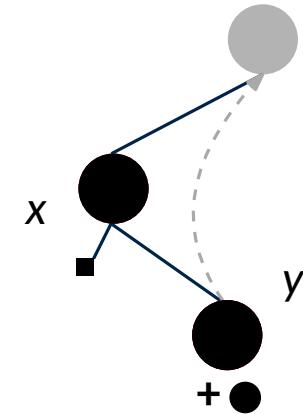
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
  - Wenn  $x$  **rot** ist, können wir dies einfach tun.
  - Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**
    - Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
    - wir sagen:  $y$  benötigt ein **Extra-Schwarz**
  - **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
    - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz aufgelöst**
    - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben
      - fahre rekursiv mit Elter  $p$  als  $y$  fort.
    - wenn das nicht möglich ist, löse das **Extra-Schwarz** durch bis zu drei Rotationen + Umfärbung auf

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



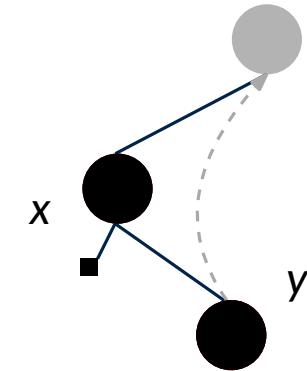
## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
  - Wenn  $x$  **rot** ist, können wir dies einfach tun.  
· Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
    → Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig  
· wir sagen:  $y$  benötigt ein **Extra-Schwarz**
  - **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**
    - wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↪ **Extra-Schwarz aufgelöst**
    - ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben  
    → fahre rekursiv mit Elter  $p$  als  $y$  fort.
    - wenn das nicht möglich ist, löse das **Extra-Schwarz** durch bis zu drei Rotationen + Umfärbung auf
    - **Sonderfall:** wenn  $y$  die Wurzel ist, dann löst sich das **Extra-Schwarz** von alleine auf

# Entfernen im Rot-Schwarz-Baum

`remove(k):`

finde Nachfolger-Schlüsselknoten  $x$  zum Schlüssel  $k$   
verschiebe Inhalt von  $x$  zum Knoten mit Schlüssel  $k$   
lösche  $x$  aus dem Baum



## Grundprinzip:

- o.B.d.A. müssen wir einen Schlüsselknoten  $x$  mit höchstens einem Schlüsselknoten  $y$  als Kind löschen
- Wenn  $x$  **rot** ist, können wir dies einfach tun.  
wenn  $x$  nur Blätter als Kinder hat, dann sei  $y$  ein Blatt
- Wenn  $x$  **schwarz** ist, entsteht eine **Tiefenverletzung!**  
→ Sobald  $y$  an die Stelle von  $x$  tritt, haben alle Wurzel-Blatt-Pfade durch  $y$  einen **schwarzen Knoten** zu wenig
- wir sagen:  $y$  benötigt ein **Extra-Schwarz**
- **Strategie zur Auflösung von Extra-Schwarz an  $y$ :**

- wenn  $y$  **rot** ist, färbe  $y$  **schwarz** ↵ **Extra-Schwarz** aufgelöst
- ansonsten wollen wir möglichst das **Extra-Schwarz** durch Umfärbung an den Elter abgeben  
→ fahre rekursiv mit Elter  $p$  als  $y$  fort.
- wenn das nicht möglich ist, löse das **Extra-Schwarz** durch bis zu drei Rotationen + Umfärbung auf
- **Sonderfall:** wenn  $y$  die Wurzel ist, dann löst sich das **Extra-Schwarz** von alleine auf

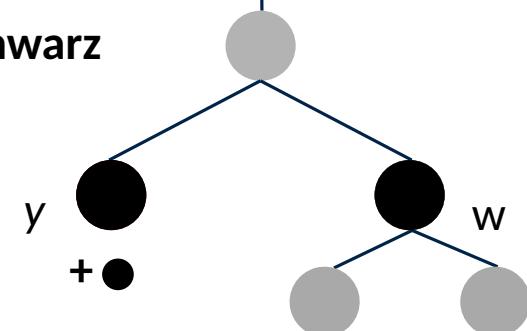
Im gesamten Prozess wollen wir keine **Rotverletzung** einführen

# Entfernen, Fall 1: schwarzer Geschwisterknoten

---

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

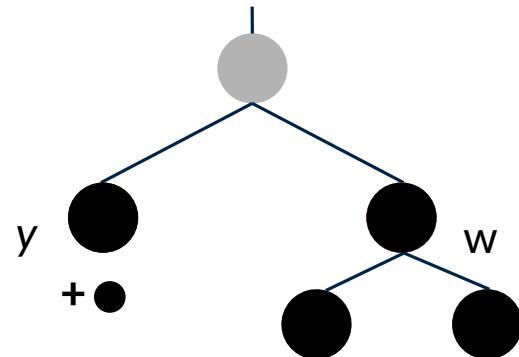
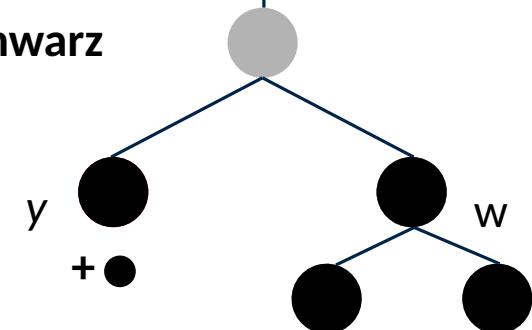


# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

**Fall 1a: Beide Kinder von  $w$  sind schwarz**



(symmetrischer Fall weggelassen)

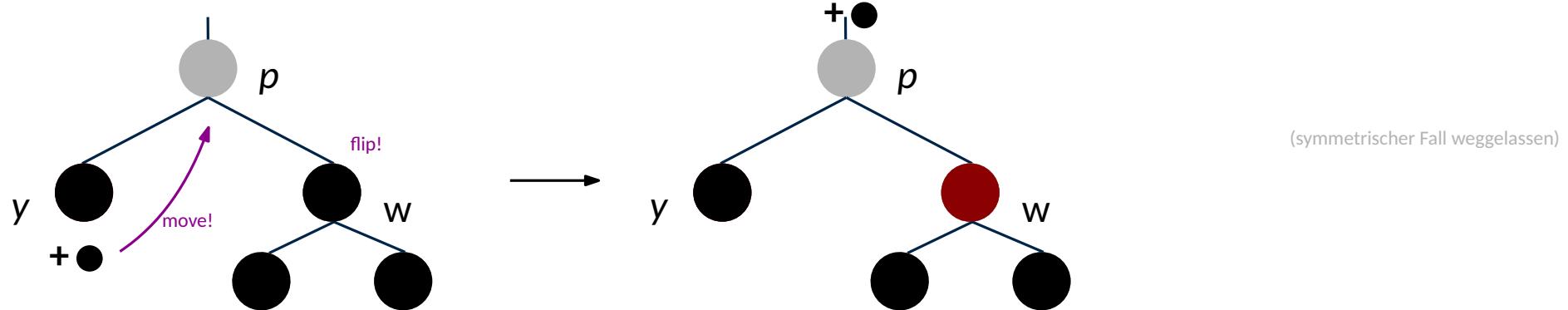
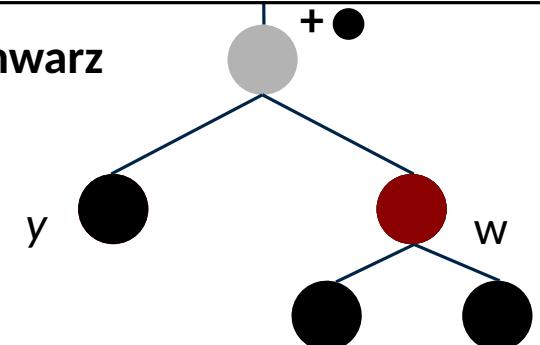
# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

**Fall 1a: Beide Kinder von  $w$  sind schwarz**

Wir färben  $w$  von **schwarz** nach **rot** und  
schieben das **Extra-Schwarz** zum Elter  $p$  von  $y$



**Bemerkung:** Nach der Umfärbung hat jeder Wurzel-Blatt-Pfad durch  $p$  (nicht nur  $y$ ) einen **schwarzen Knoten** zu wenig  
→ von nun an benötigt  $p$  (statt  $y$ ) das **Extra-Schwarz**

**Frage:** Kann eine **Rotverletzung** an  $w$  eintreten?

Ja, wenn  $p$  **rot** ist. Dann wird allerdings  $p$  im nächsten Schritt **schwarz** gefärbt!

Danach fahren wir mit dem Elter  $p$  als  $y$  fort

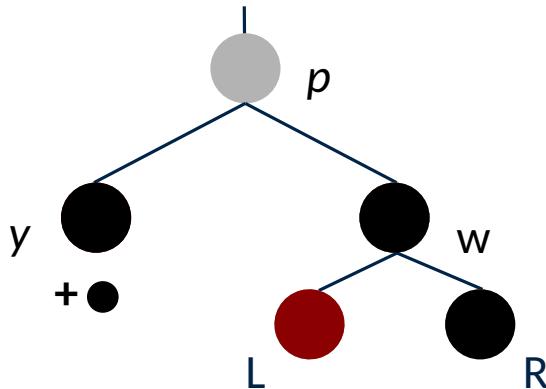
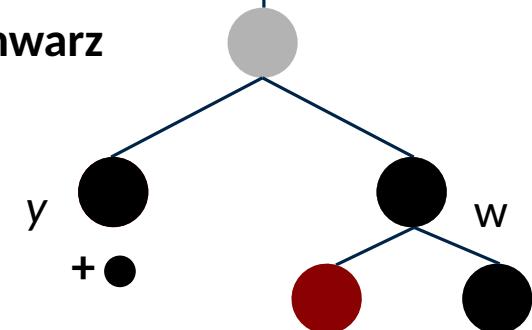
# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

**Fall 1a:** Beide Kinder von  $w$  sind schwarz

**Fall 1b:** Nur das linke Kind von  $w$  ist rot



# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

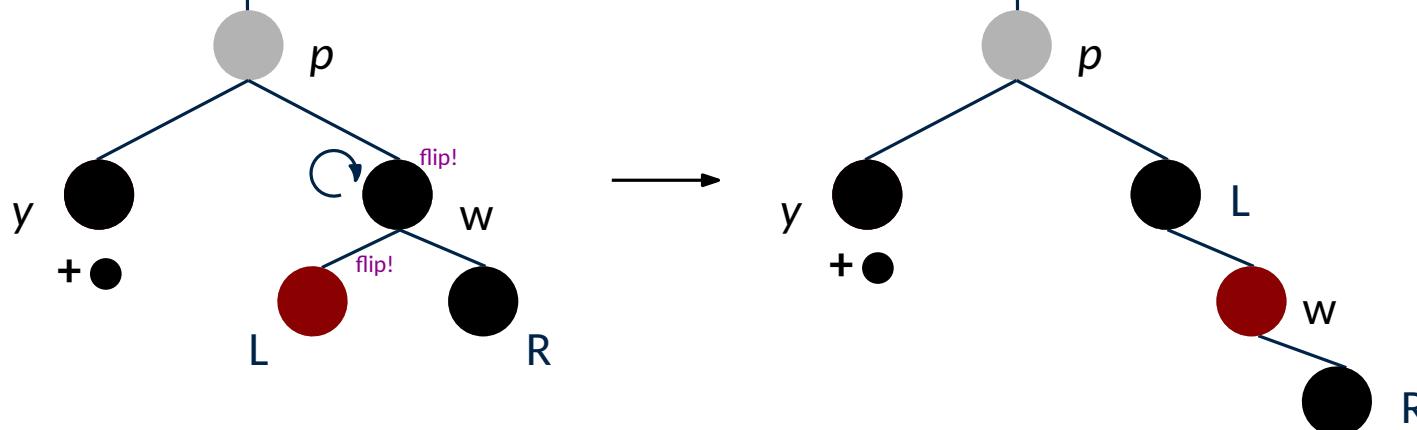
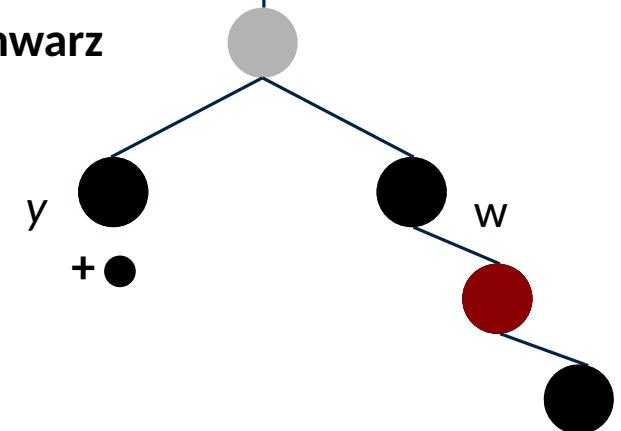
**Fall 1: Geschwisterknoten  $w$  ist schwarz**

**Fall 1a: Beide Kinder von  $w$  sind schwarz**

**Fall 1b: Nur das linke Kind von  $w$  ist rot**

Wir rotieren  $w$  in die entgegengesetzte Richtung von  $y$

Wir färben  $w$  **rot** und das linke Kind  $L$  von  $w$  **schwarz**.



**Bemerkung:** Es entsteht keine **Rotverletzung**

# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

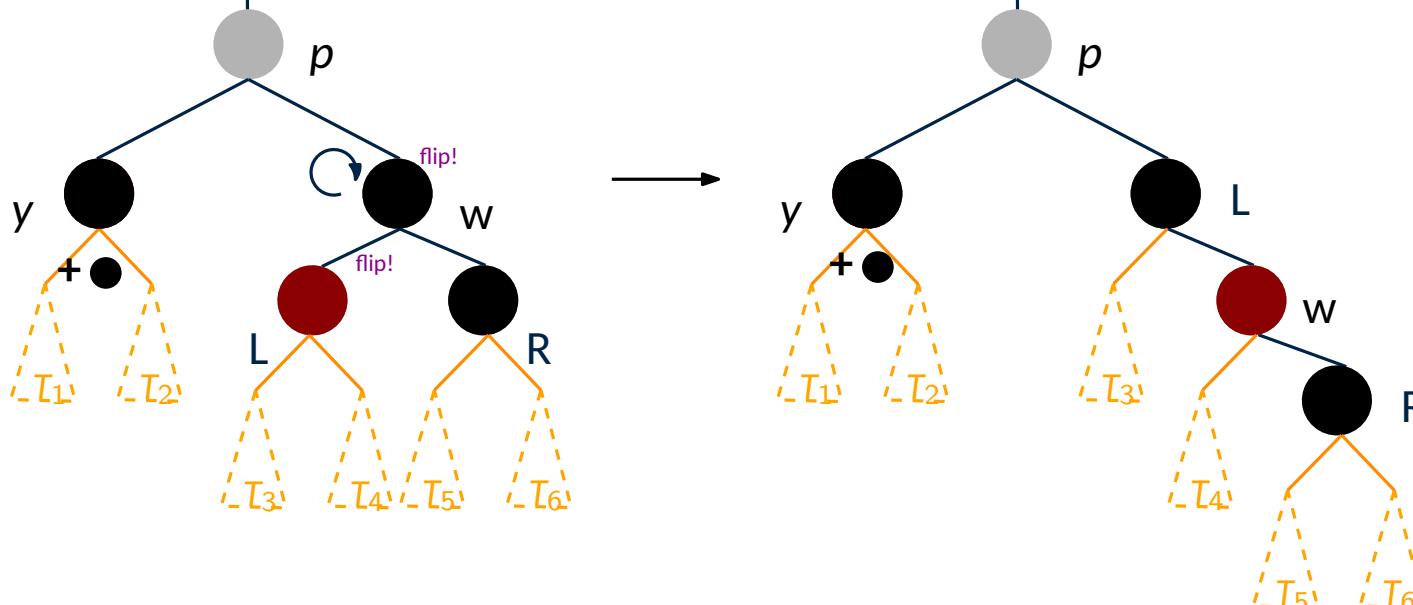
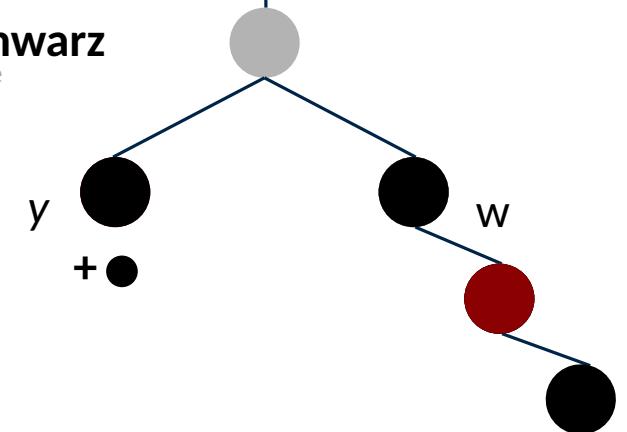
**Fall 1: Geschwisterknoten  $w$  ist schwarz**

**Fall 1a: Beide Kinder von  $w$  sind schwarz**

**Fall 1b: Nur das linke Kind von  $w$  ist rot**

Wir rotieren  $w$  in die entgegengesetzte Richtung von  $y$

Wir färben  $w$  **rot** und das linke Kind  $L$  von  $w$  **schwarz**.



**Bemerkung:** Es entsteht keine **Rotverletzung**

Wir verändern keine **Schwarztiefe**

Wir transformieren die Situation lediglich in den **Fall 1c!**

# Entfernen, Fall 1: schwarzer Geschwisterknoten

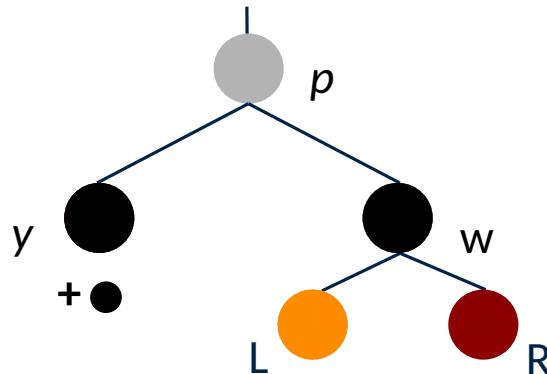
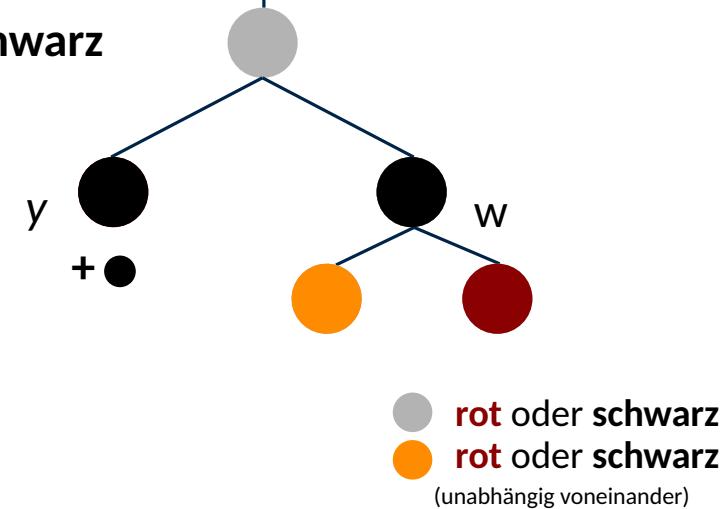
**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

Fall 1a: Beide Kinder von  $w$  sind schwarz

Fall 1b: Nur das linke Kind von  $w$  ist rot

Fall 1c: mindestens das rechte Kind von  $w$  ist **rot**



# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

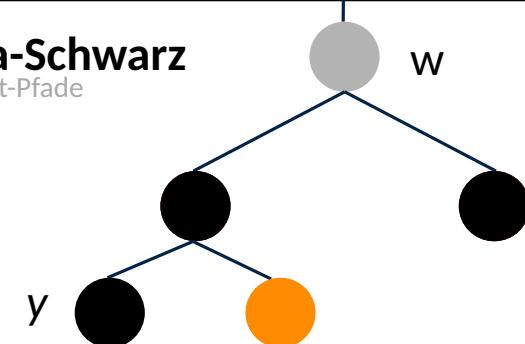
Fall 1a: Beide Kinder von  $w$  sind schwarz

Fall 1b: Nur das linke Kind von  $w$  ist rot

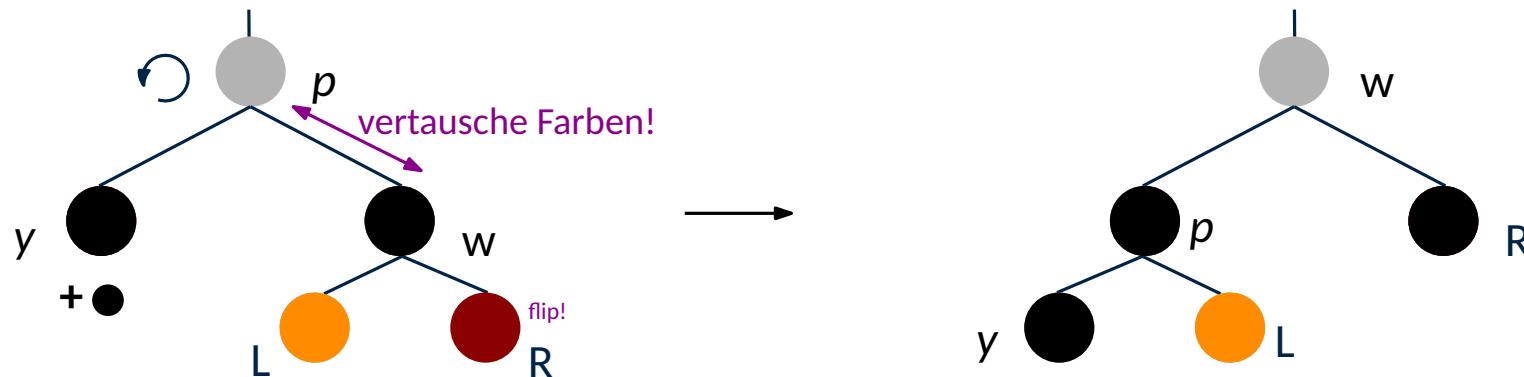
**Fall 1c: mindestens das rechte Kind von  $w$  ist rot**

Wir rotieren  $p$  in die Richtung von  $y$

Wir **vertauschen** die Farben von  $p$  und  $w$  und färben das rechte Kind  $R$  von  $w$  **schwarz**



rot oder schwarz  
rot oder schwarz  
(unabhängig voneinander)



**Bemerkung:** Es entsteht keine **Rotverletzung**

# Entfernen, Fall 1: schwarzer Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 1: Geschwisterknoten  $w$  ist schwarz**

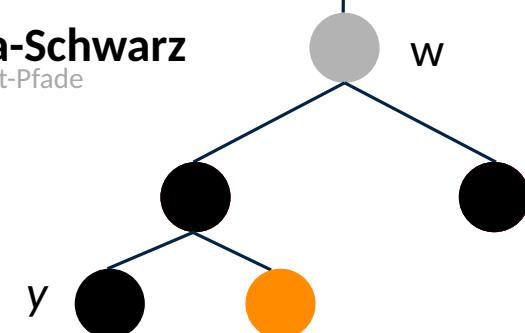
Fall 1a: Beide Kinder von  $w$  sind schwarz

Fall 1b: Nur das linke Kind von  $w$  ist rot

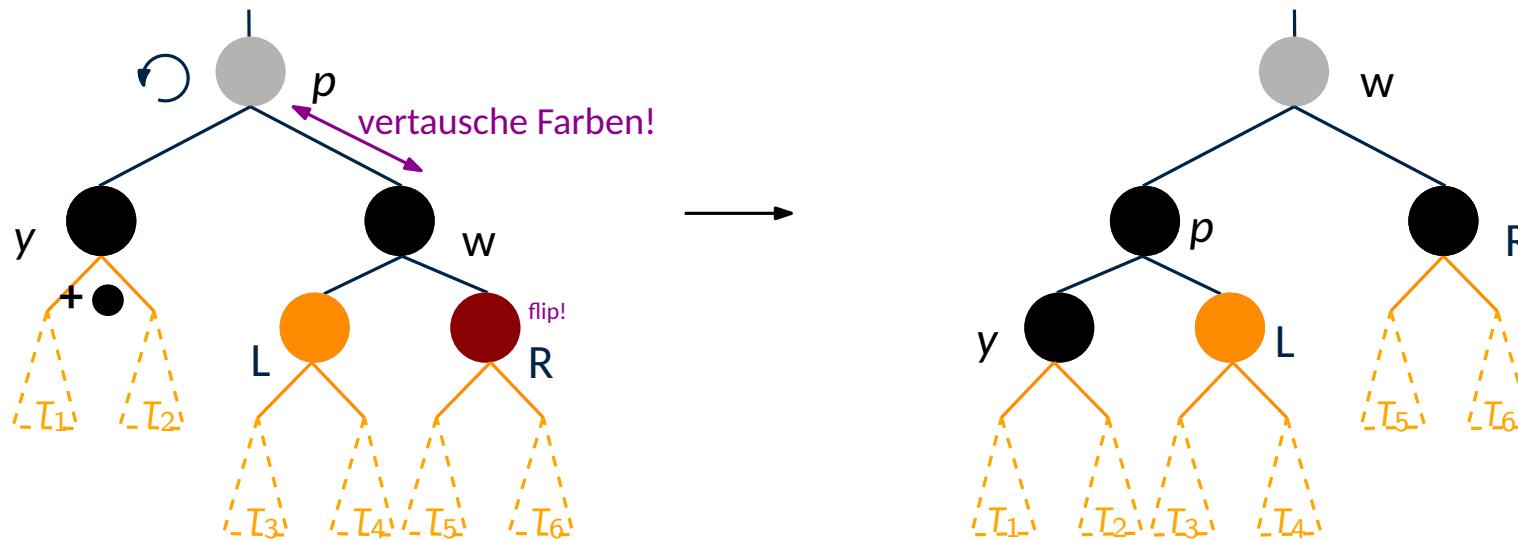
**Fall 1c: mindestens das rechte Kind von  $w$  ist rot**

Wir rotieren  $p$  in die Richtung von  $y$

Wir **vertauschen** die Farben von  $p$  und  $w$  und färben das rechte Kind  $R$  von  $w$  **schwarz**



rot oder schwarz  
rot oder schwarz  
(unabhängig voneinander)



**Bemerkung:** Es entsteht keine **Rotverletzung**

Alle Blätter im Teilbaum haben danach die gleiche **Schwarztiefe**!

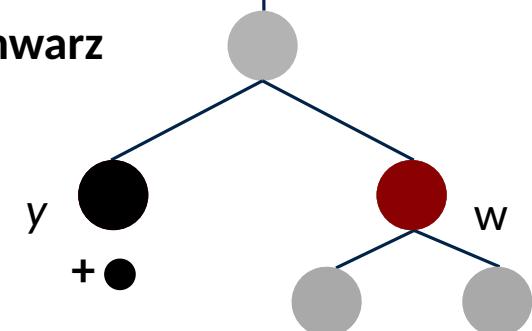
→ **Tiefenverletzung** aufgelöst!

# Entfernen, Fall 2: roter Geschwisterknoten

---

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

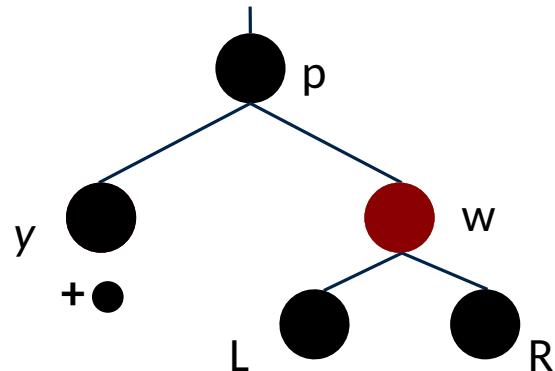
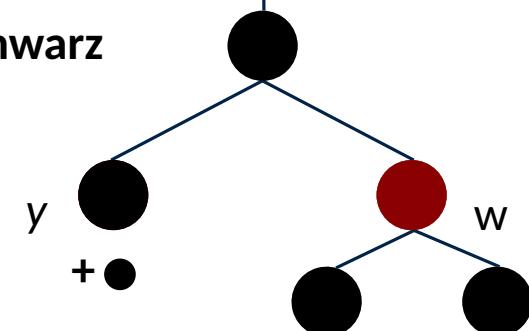
**Fall 2: Geschwisterknoten  $w$  ist rot**



# Entfernen, Fall 2: roter Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 2: Geschwisterknoten  $w$  ist rot**



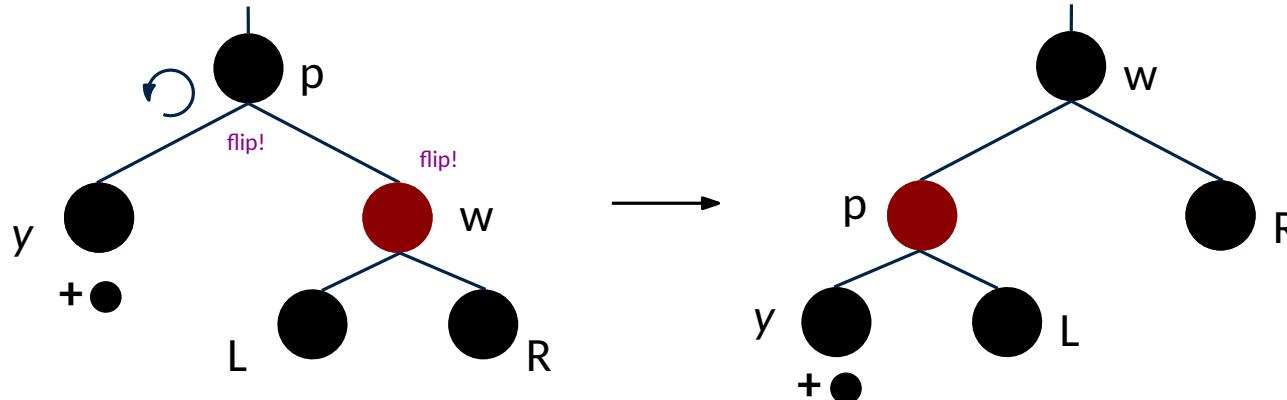
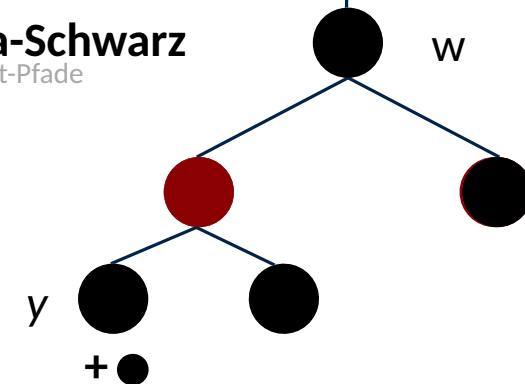
# Entfernen, Fall 2: roter Geschwisterknoten

**Ausgangssituation:**  $y$  ist **schwarzer Knoten** und benötigt **Extra-Schwarz**  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen **schwarzen Knoten** weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 2: Geschwisterknoten  $w$  ist rot**

Wir rotieren  $p$  in die Richtung von  $y$

Wir färben  $p$  **rot** und  $w$  **schwarz**.



**Bemerkung:** Es entsteht keine **Rotverletzung**

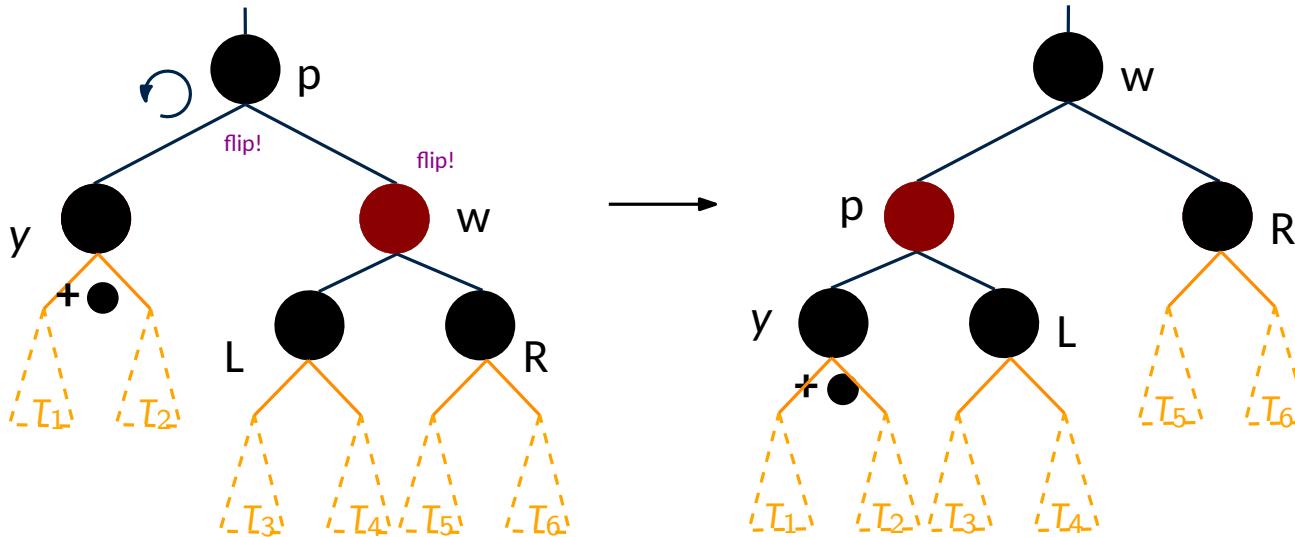
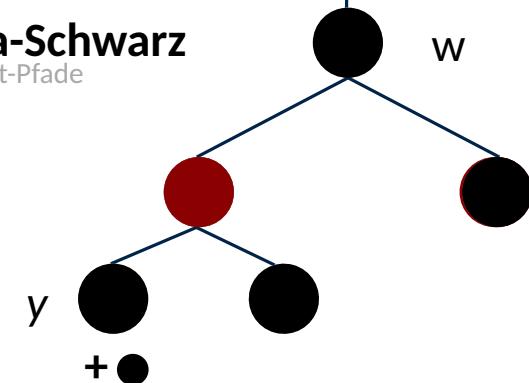
# Entfernen, Fall 2: roter Geschwisterknoten

**Ausgangssituation:**  $y$  ist schwarzer Knoten und benötigt Extra-Schwarz  
d.h. jeder Wurzel-Blatt-Pfad durch  $y$  hat einen schwarzen Knoten weniger als alle anderen Wurzel-Blatt-Pfade

**Fall 2: Geschwisterknoten  $w$  ist rot**

Wir rotieren  $p$  in die Richtung von  $y$

Wir färben  $p$  rot und  $w$  schwarz.



**Bemerkung:** Es entsteht keine Rotverletzung

Wir verändern keine Schwarztiefe

Wir transformieren die Situation lediglich in den Fall 1!

# Beispiel

---

↗ Tafelpräsentation

# Korrektheit und Laufzeit von remove(k)

---

**Korrektheit:** folgt aus unseren Bemerkungen zu den einzelnen Fällen

wir übertragen das **Extraschwarz** an den jeweiligen Elternknoten bis:

- wir auf einen **roten Knoten**  $y$  treffen, den wir **schwarz** färben, oder
- wir auf die Wurzel treffen, sodass das **Extraschwarz** automatisch aufgelöst ist, oder
- durch geeignete Rotation(en) das **Extraschwarz** auflösen konnten

**Lemma.**

`remove(k)` entfernt das Element mit Schlüssel  $k$  in Zeit  $O(\log n)$  und benutzt höchstens 3 Rotationen.

# Zusammenfassung

---

Wörterbuch (Dictionary, Assoziatives Array) via Rot-Schwarz-Bäumen

`S.find(k)` - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k?$   $O(\log n)$

`S.insert(e)` - füge  $e$  in  $S$  ein  $O(\log n)$

`S.remove(k)` - lösche das Element mit Schlüssel  $k$  aus  $S$   $O(\log n)$

Zusätzliche Operationen:

`S.min()` - gib das Element mit dem kleinsten Schlüssel zurück  $O(\log n)$   
`S.max()` - gib das Element mit dem größten Schlüssel zurück

`S.successor(k)` - gib das Element mit dem nächstgrößeren Schlüssel zu  $k$  zurück  
`S.predecessor(k)` - gib das Element mit dem nächstkleineren Schlüssel zu  $k$  zurück  $O(\log n)$

Dramatische Verbesserungen gegenüber  $O(n)$ -Laufzeiten!

## Algorithmen und Datenstrukturen SS'23

# Kapitel 10: $(a, b)$ -Bäume

Marvin Künnemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letztes Kapitel: Binäre Suchbäume, insbesondere: **Rot-Schwarz-Bäume**

Jetzt: weiterer balancierter, aber nicht binärer Suchbaum

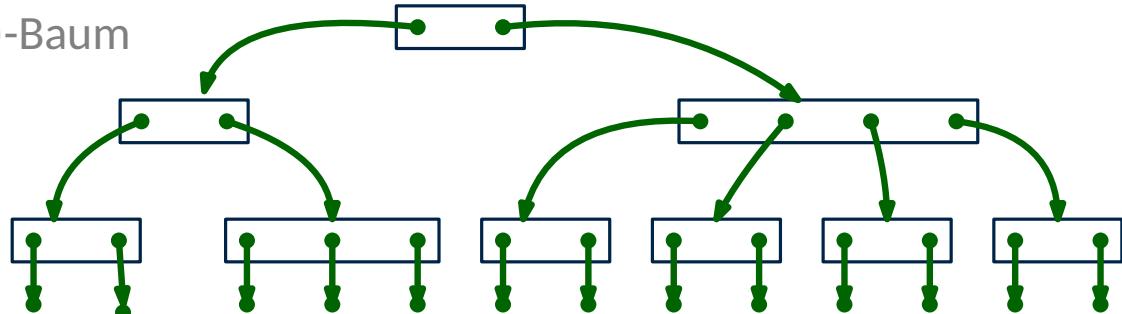
Insbesondere besprechen wir in diesem Kapitel:

- Definition von  $(a, b)$ -Bäumen
- effiziente Wörterbuchoperationen auf  $(a, b)$ -Bäumen
- Antwort auf "Wie kommt man auf die Idee von Rot-Schwarz-Bäumen?"
- amortisierte Analyse der Änderungs-Operationen

# $(a, b)$ -Bäume: Definition

Für einen Knoten  $u$  sei der **Grad**  $\deg(u)$  definiert als die Anzahl Kinder von  $u$ .

Beispiel:  $(2, 4)$ -Baum



## Definition

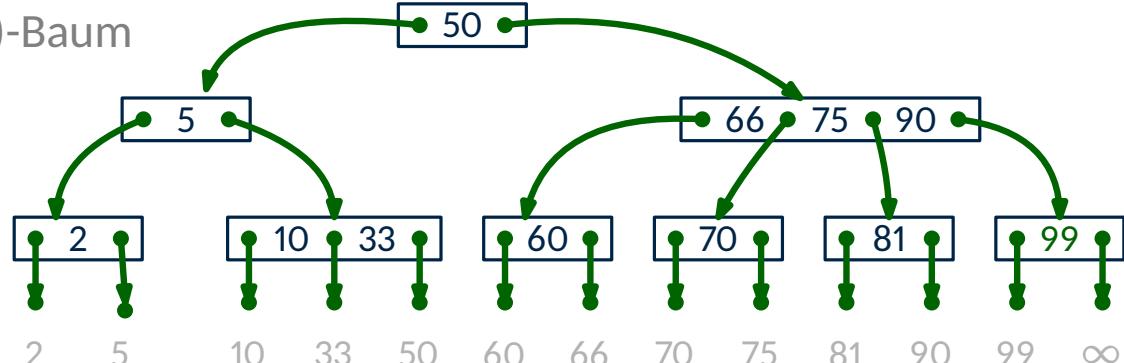
Ein  $(a, b)$ -Baum ist ein geordneter Baum  $T$  mit folgenden Eigenschaften:

- **Gradbedingung:** alle inneren Knoten außer die Wurzel  $r$  haben mindestens  $a$  und höchstens  $b$  Kinder  
$$a \leq \deg(u) \leq b \quad \text{für alle inneren Knoten } u \text{ mit } u \neq r$$
- **Wurzelbedingung:** die Wurzel  $r$  hat mindestens 2 und höchstens  $b$  Kinder  
$$2 \leq \deg(r) \leq b$$
- **Blattbedingung:** alle Blätter befinden sich auf dem gleichen Level

# $(a, b)$ -Bäume: Definition

Für einen Knoten  $u$  sei der **Grad**  $\deg(u)$  definiert als die Anzahl Kinder von  $u$ .

Beispiel:  $(2, 4)$ -Baum



## Definition

Ein  $(a, b)$ -Baum ist ein geordneter Baum  $T$  mit folgenden Eigenschaften:

- **Gradbedingung:** alle inneren Knoten außer die Wurzel  $r$  haben mindestens  $a$  und höchstens  $b$  Kinder  
$$a \leq \deg(u) \leq b \quad \text{für alle inneren Knoten } u \text{ mit } u \neq r$$
- **Wurzelbedingung:** die Wurzel  $r$  hat mindestens 2 und höchstens  $b$  Kinder  
$$2 \leq \deg(r) \leq b$$
- **Blattbedingung:** alle Blätter befinden sich auf dem gleichen Level

Jedem Knoten  $u$  sind  $\deg(u) - 1$  Schlüssel als **Splitter**  $s_1 \leq s_2 \leq \dots \leq s_{\deg(u)-1}$  zugeordnet.

Wir setzen  $s_0 := -\infty$  und  $s_{\deg(u)} := \infty$

- **Suchbaumeigenschaft:** die Schlüssel im Teilbaum des  $i$ . Kindes sind in  $(s_{i-1}, s_i]$

Wir verlangen, dass  $a \geq 2$  und  $b \geq 2a - 1$

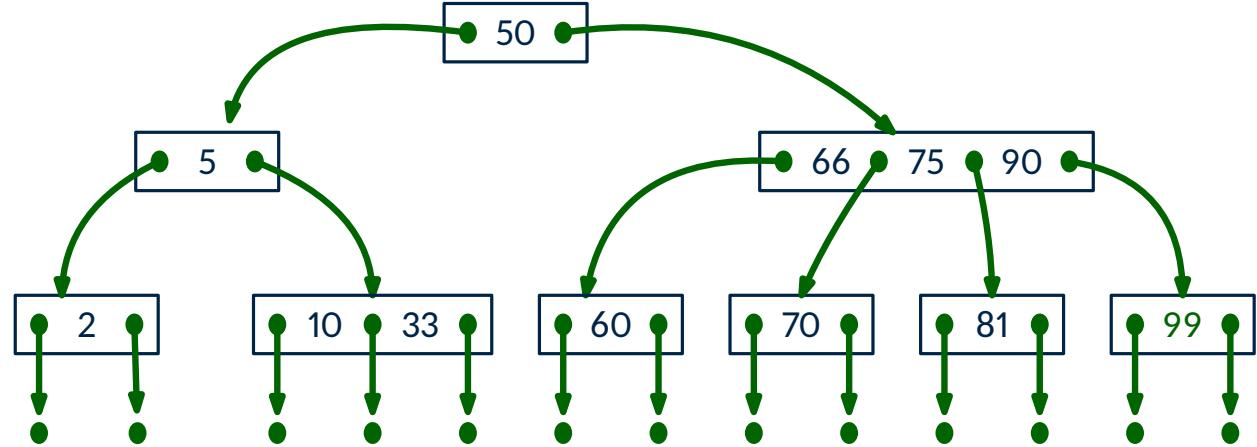
Typischerweise sind  $a$  und  $b$  Konstanten, wie z.B.  $a = 2$  und  $b = 4$

# Implementierung von $(a, b)$ -Bäumen

Die bis zu  $b - 1$  Splitter werden in einem sortierten Array gespeichert.

```
struct BNode {  
    int deg; //Grad des Knotens  
    type s[b-1]; //Splitter  
    BNode* children[b];  
}
```

```
struct BTREE {  
    BNode* root;  
    int h; //Höhe des Baumes  
}
```

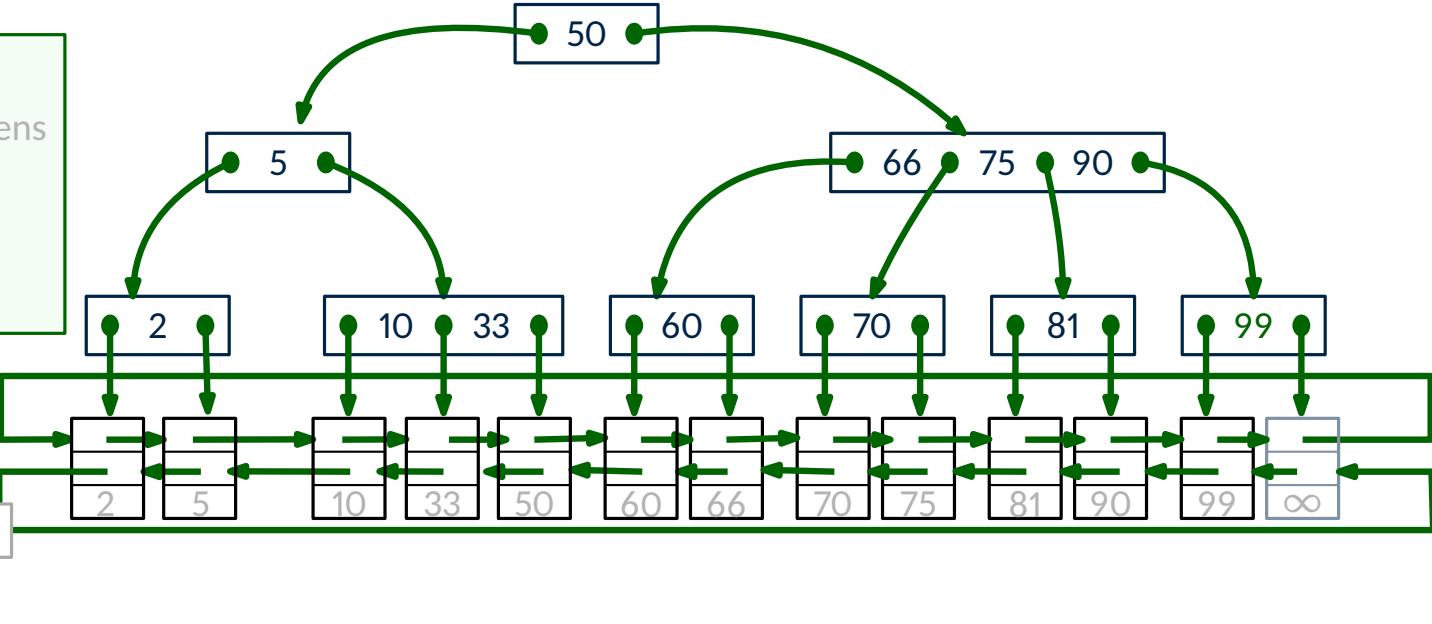


# Implementierung von $(a, b)$ -Bäumen

Die bis zu  $b - 1$  Splitter werden in einem sortierten Array gespeichert.

```
struct BNode {  
    int deg; //Grad des Knotens  
    type s[b-1]; //Splitter  
    BNode* children[b];  
}
```

```
struct BTREE {  
    BNode* root;  
    int h; //Höhe des Baumes  
}
```



Typischerweise werden die eigentlichen Elemente in den Blättern gespeichert.

Hierbei eignet sich besonders ein doppelt verkettete Liste!

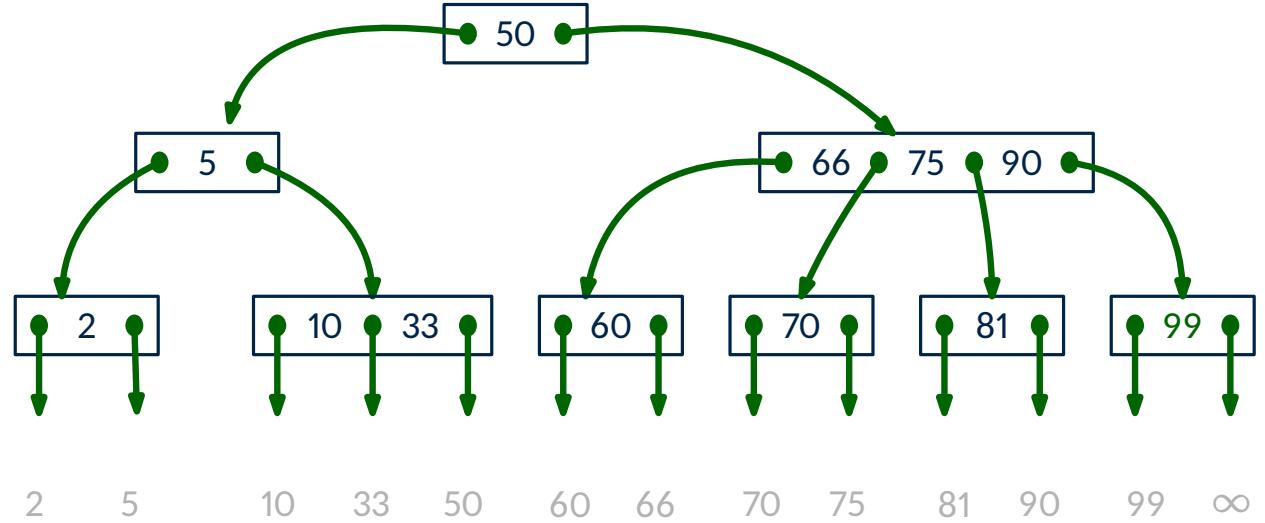
**Hinweis:** Die successor-Funktion läuft damit sogar in Zeit  $O(1)$

# Implementierung von $(a, b)$ -Bäumen

Die bis zu  $b - 1$  Splitter werden in einem sortierten Array gespeichert.

```
struct BNode {  
    int deg; //Grad des Knotens  
    type s[b-1]; //Splitter  
    BNode* children[b];  
}
```

```
struct BTREE {  
    BNode* root;  
    int h; //Höhe des Baumes  
}
```



Typischerweise werden die eigentlichen Elemente in den Blättern gespeichert.

Hierbei eignet sich besonders ein doppelt verkettete Liste!

**Hinweis:** Die successor-Funktion läuft damit sogar in Zeit  $O(1)$

# Höhe von $(a, b)$ -Bäumen

## Lemma

Für die Höhe eines  $(a, b)$ -Baums mit  $n$  Schlüsseln gilt:

$$\log_b(n + 1) \leq h \leq 1 + \log_a(n + 1)$$

## Beweis

Der Baum hat genau  $n + 1$  Blätter.

Ein Baum der Höhe  $h$ , in dem jeder innere Knoten höchstens  $b$  Kinder hat, hat höchstens  $b^h$  Blätter.

$$\Rightarrow b^h \geq n + 1$$

$$\Rightarrow h \geq \log_b(n + 1)$$

Ein Baum der Höhe  $h$ , in dem

- die Wurzel mindestens 2 Kinder hat und
- alle anderen inneren Knoten mindestens  $a$  Kinder haben,

hat mindestens  $2a^{h-1} \geq a^{h-1}$  Blätter.

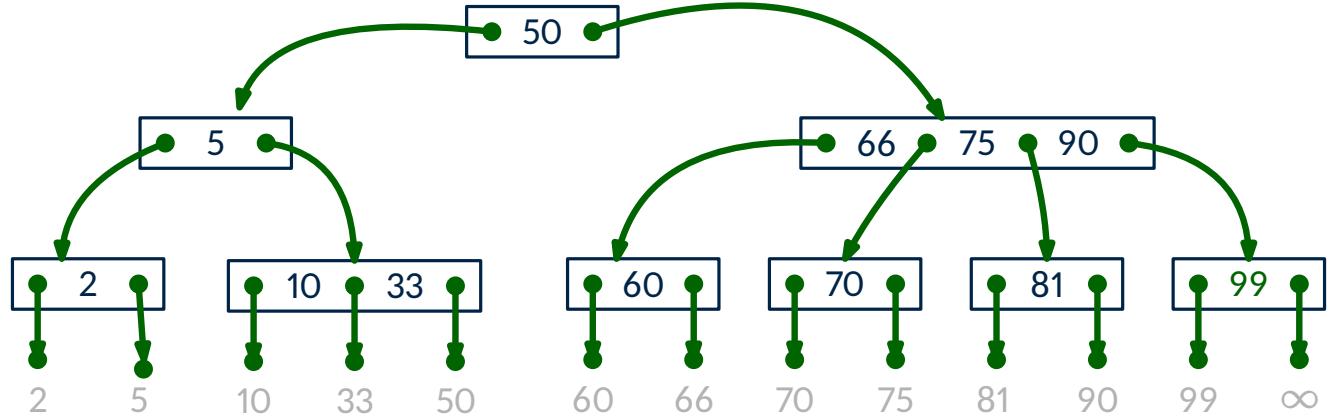
$$\Rightarrow a^{h-1} \leq n + 1$$

$$\Rightarrow h - 1 \leq \log_a(n + 1)$$

□

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

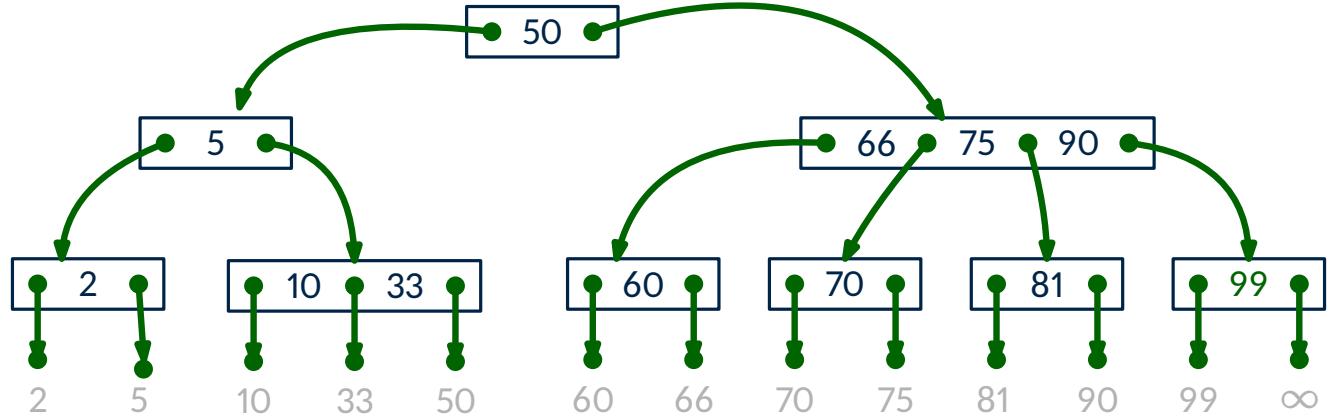
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 70, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

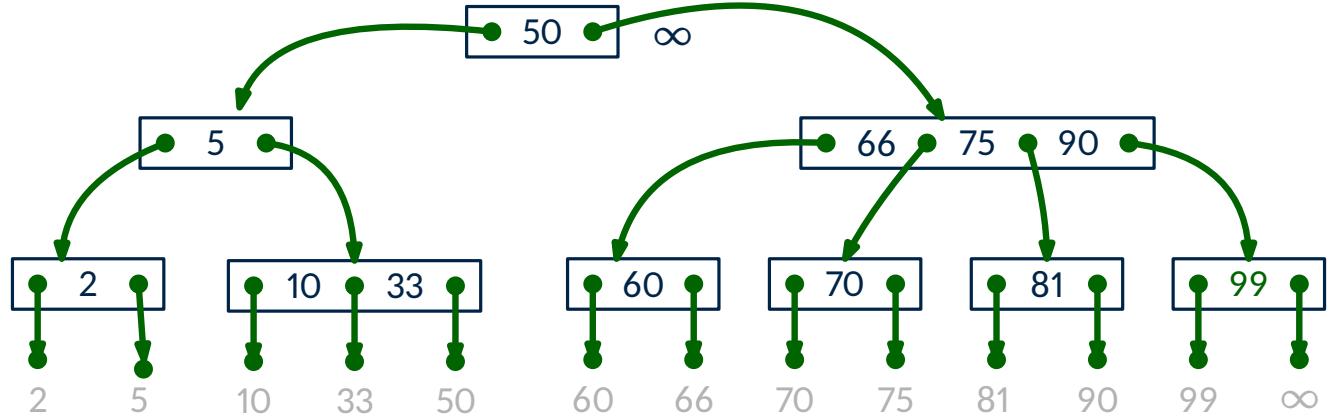
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 70, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

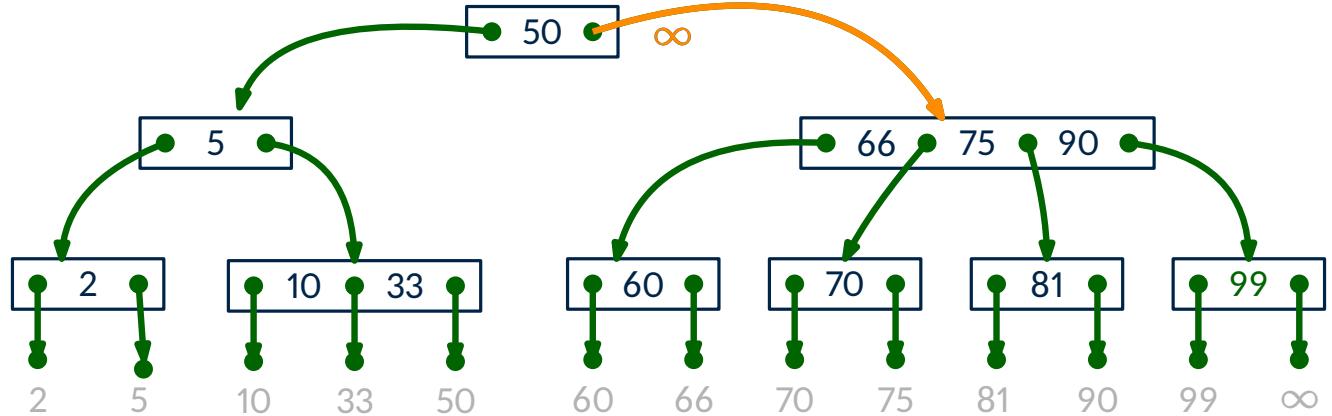
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 70, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

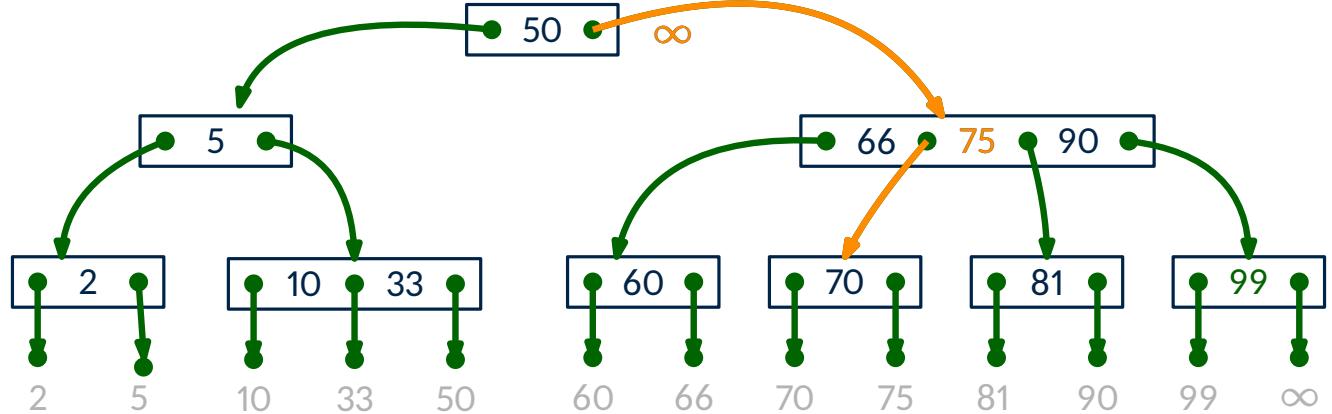
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 70, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

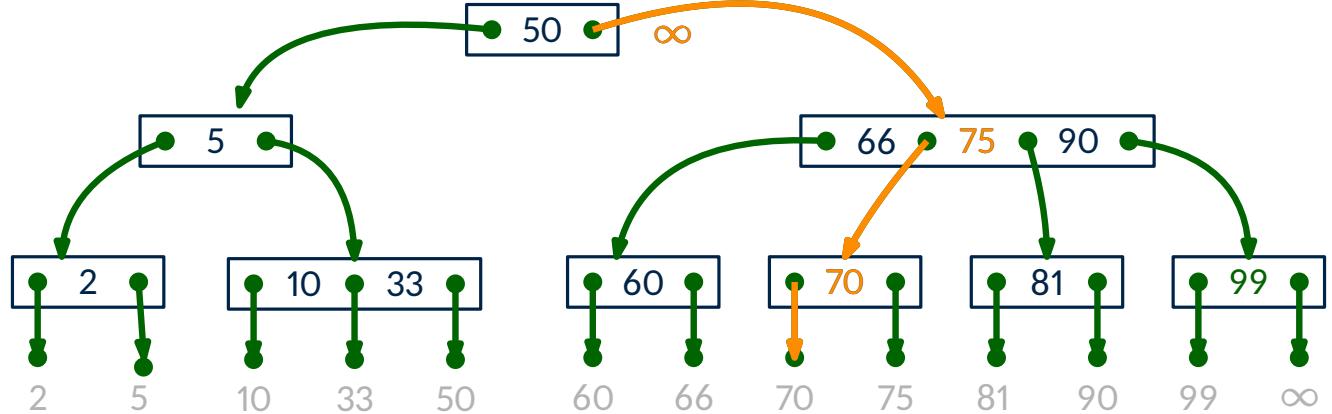
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 70, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

**Frage:**

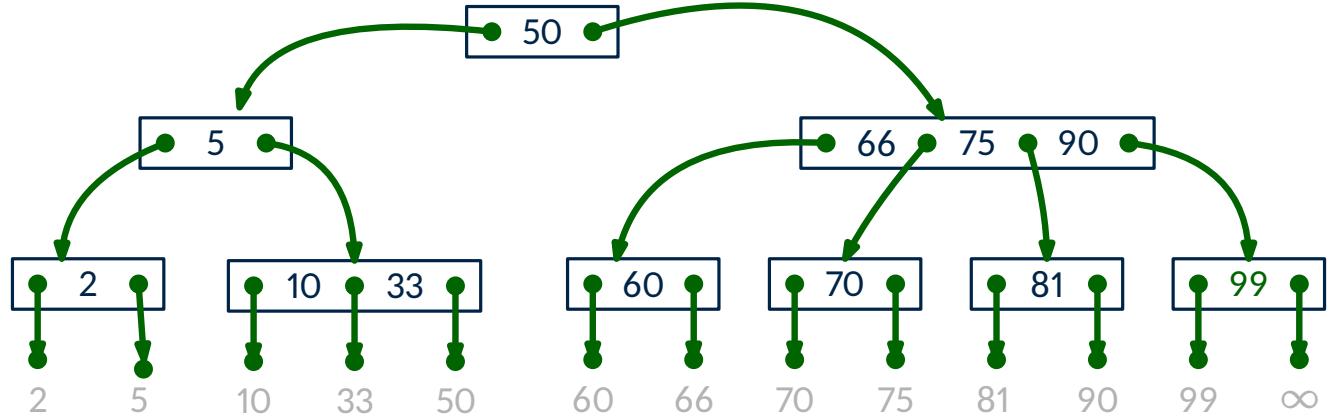
**Antwort:**

Wie bestimmen wir  $i$  am besten?

(Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

```
find(Node* x, type k, int h): h ist Höhe vom Teilbaum an x
```

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

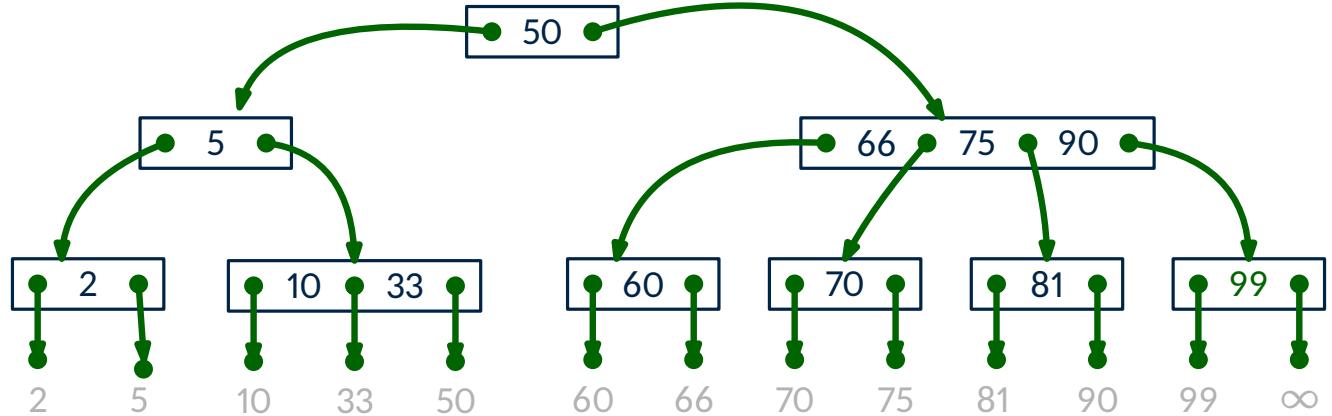
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 49, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

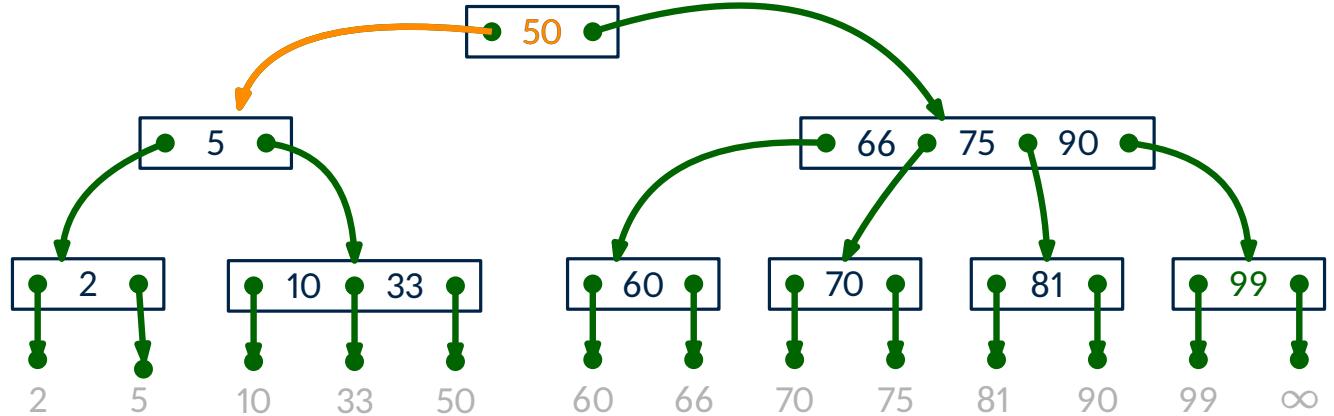
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 49, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

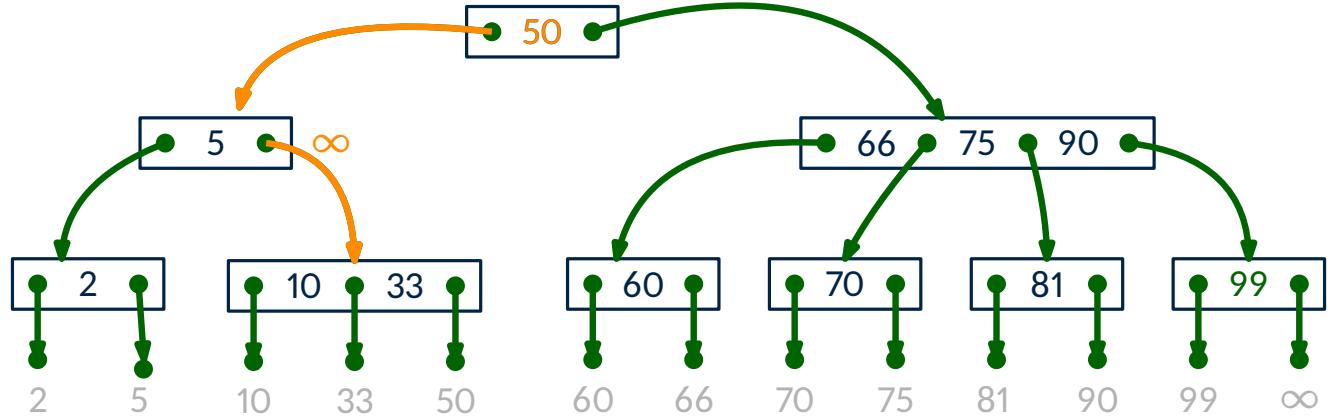
**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?

`find(T.root, 49, T.h)`



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

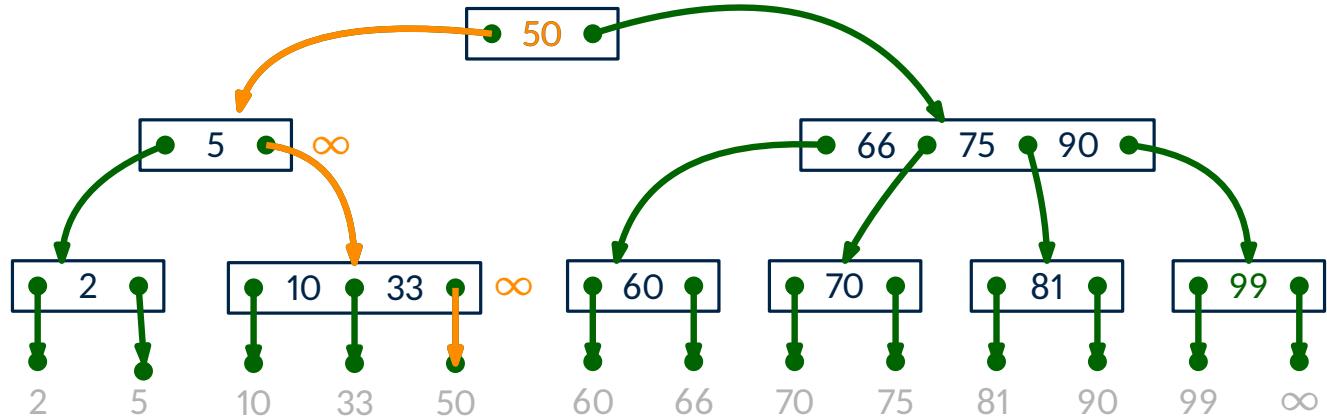
gib `find(x->children[i], k, h-1)` zurück.

**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

# Wie finden wir ein Element mit Schlüssel $k$ ?



**Achtung:** Diesmal gibt `find` das Element mit dem **kleinsten** Schlüssel  $k' \geq k$  zurück!

`find(Node* x, type k, int h):`  $h$  ist Höhe vom Teilbaum an  $x$

bestimme  $i$  als kleinsten Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
 $\rightarrow i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

Wenn  $h = 1$ :

gib x->children[i] zurück.

## Ansonsten:

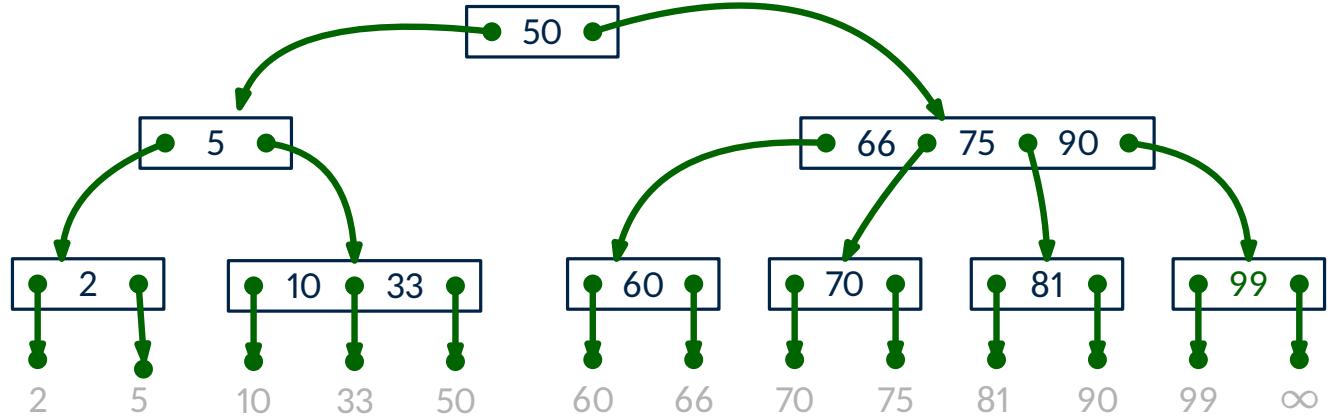
gib `find(x->children[i], k, h-1)` zurück.

## Frage: Wie bestimmen wir *i* am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots s[\deg]$   
 $\rightarrow O(\log b)$  Laufzeit

# Suche im $(a, b)$ -Baum

Wie finden wir ein Element mit Schlüssel  $k$ ?



Achtung: Diesmal gibt `find` das Element mit dem kleinsten Schlüssel  $k' \geq k$  zurück

```
find(Node* x, type k, int h): h ist Höhe vom Teilbaum an x
```

bestimme  $i$  als kleinstein Wert  $1 \leq j \leq \deg$  sodass  $k \leq s[j]$

Hierbei ist  $s[\deg] = \infty$   
→  $i$  ist wohldefiniert und in  $\{1, \dots, \deg\}$

**Wenn  $h = 1$ :**

gib `x->children[i]` zurück.

**Ansonsten:**

gib `find(x->children[i], k, h-1)` zurück.

**Frage:** Wie bestimmen wir  $i$  am besten?

**Antwort:** (Angepasste) binäre Suche in  $s[1], \dots, s[\deg]$   
→  $O(\log b)$  Laufzeit

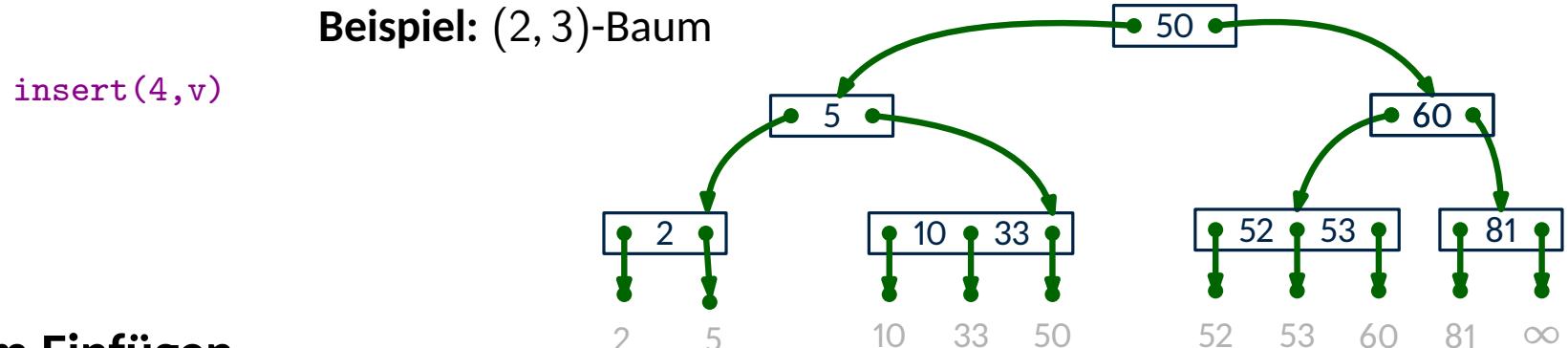
**Laufzeit:**  $O(h \cdot \log b)$ .

→ Wenn  $b$  eine feste Konstante ist, ist das in  $O(h) = O(\log_a n) = O(\log n)$ .

letztes Lemma

$b$  konstant

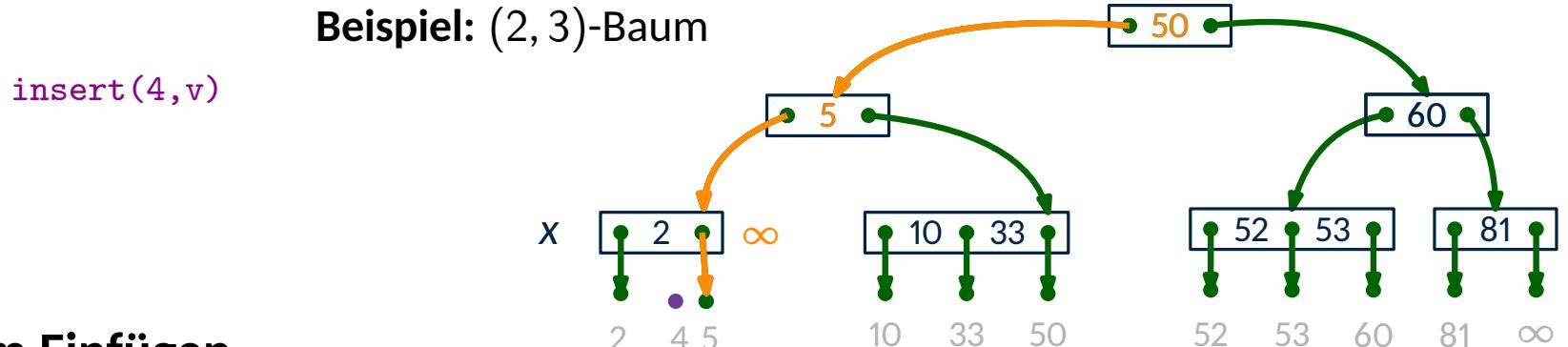
# Einfügen im $(a, b)$ -Baum



## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss

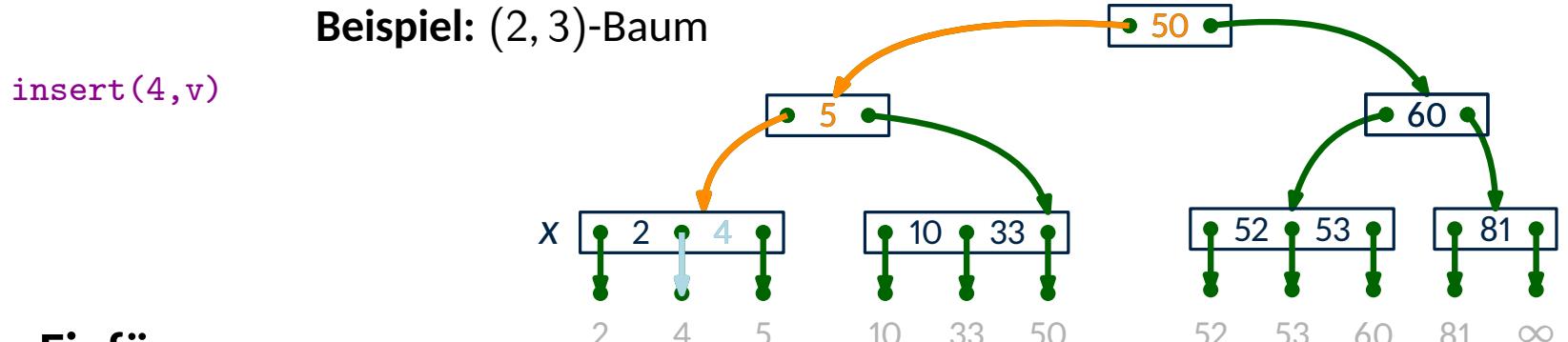
# Einfügen im $(a, b)$ -Baum



## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$
- der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einführen.

# Einfügen im $(a, b)$ -Baum



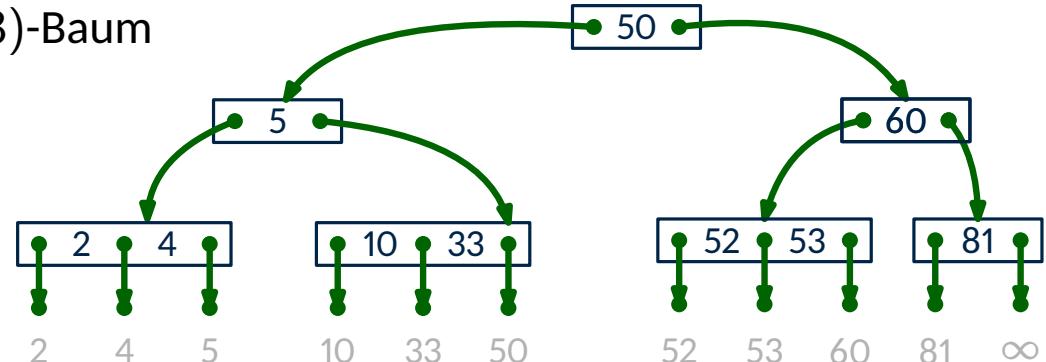
## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einfügen.
- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.

# Einfügen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

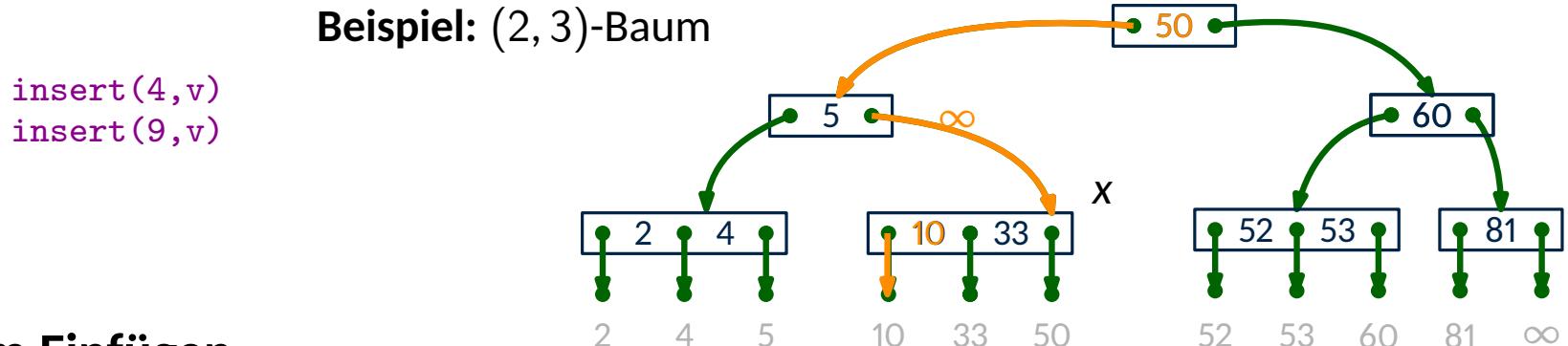
`insert(4, v)  
insert(9, v)`



## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einführen.
- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.

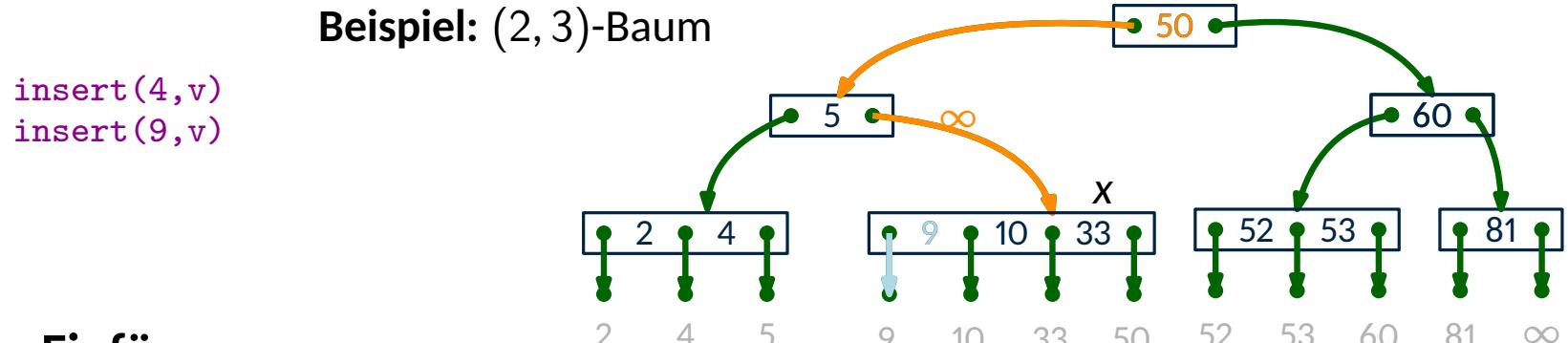
# Einfügen im $(a, b)$ -Baum



## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einführen.
- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.

# Einfügen im $(a, b)$ -Baum

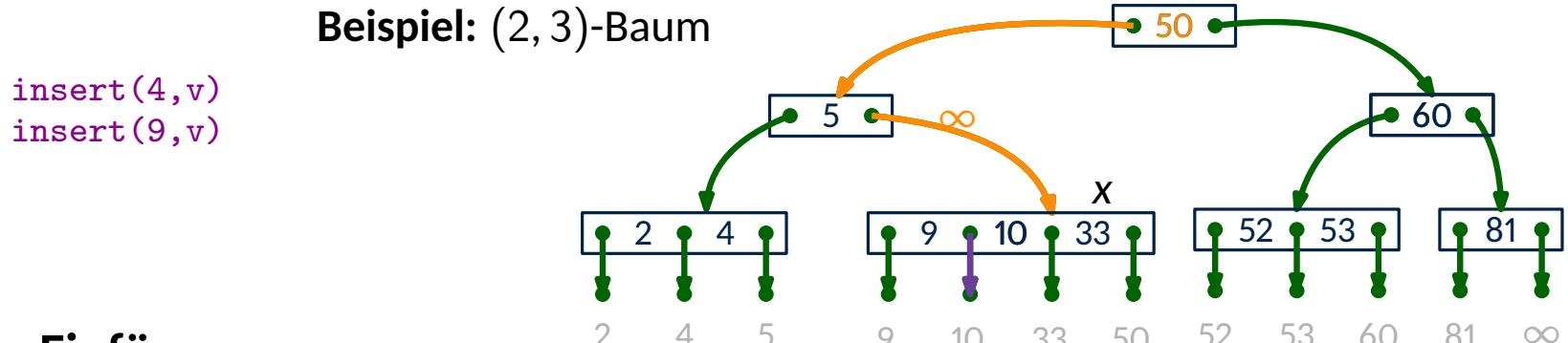


## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einführen.

- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.
- **Ansonsten** ist der neue Grad  $\deg(x) = b + 1$  und wir machen eine **Spaltung (Split)** von  $x$ :

# Einfügen im $(a, b)$ -Baum

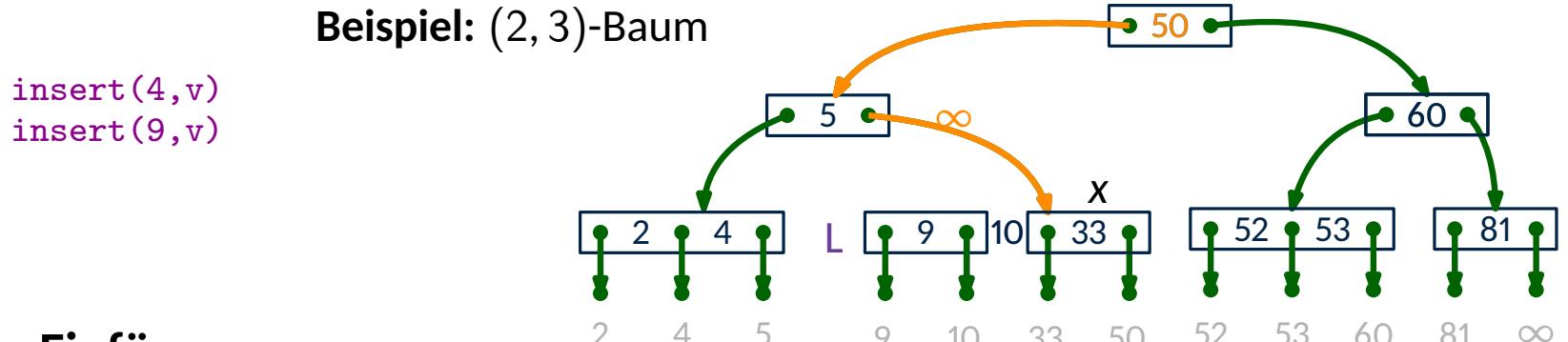


## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einführen.

- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.
- **Ansonsten** ist der neue Grad  $\deg(x) = b + 1$  und wir machen eine **Spaltung (Split)** von  $x$  :
  - Wir wählen den mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$  zur Aufteilung (Median)

# Einfügen im $(a, b)$ -Baum



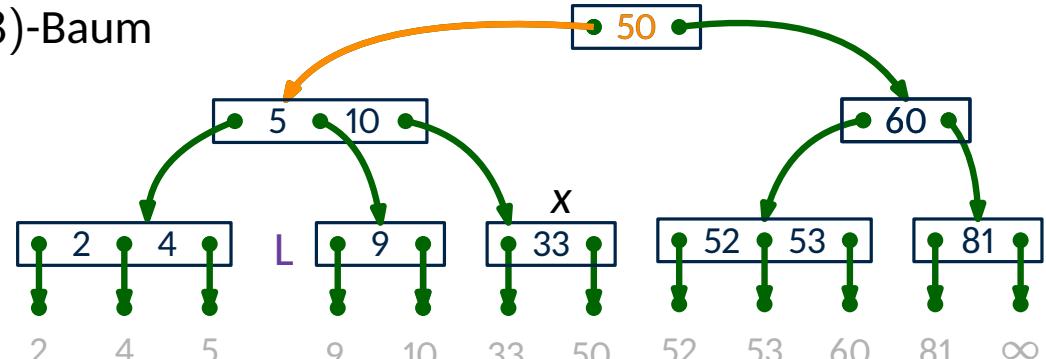
## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einfügen.
- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.
- **Ansonsten** ist der neue Grad  $\deg(x) = b + 1$  und wir machen eine **Spaltung (Split)** von  $x$  :
  - Wir wählen den mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$  zur Aufteilung (Median)
  - Wir erzeugen einen neuen Knoten  $L$  links von  $x$ , der alle kleineren Splitter als  $s[m]$  und das Kind  $c[m]$  enthält
  - $x$  behält alle verbleibenden Splitter und Kinder (diese enthalten nur Schlüssel  $> s[m]$ )
  - Jetzt müssen wir den neuen Knoten  $L$  mit dem Splitter  $s[m]$  an den Elternknoten von  $x$  einfügen  
→ wir setzen  $c = L$ ,  $s = s[m]$  und  $x$  auf den Elternknoten von  $x$

# Einfügen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

`insert(4, v)  
insert(9, v)`



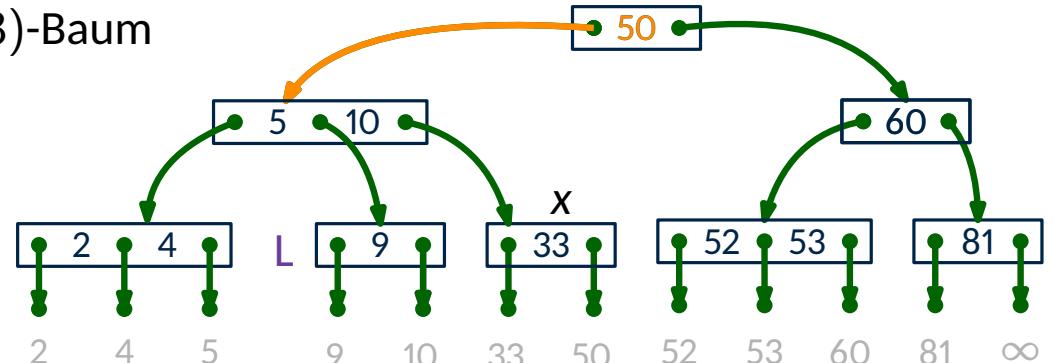
## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$   
→ der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einfügen.
- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- **Wenn**  $\deg(x) \leq b$  sind wir fertig.
- **Ansonsten** ist der neue Grad  $\deg(x) = b + 1$  und wir machen eine **Spaltung (Split)** von  $x$  :
  - Wir wählen den mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$  zur Aufteilung (Median)
  - Wir erzeugen einen neuen Knoten  $L$  links von  $x$ , der alle kleineren Splitter als  $s[m]$  und das Kind  $c[m]$  enthält
  - $x$  behält alle verbleibenden Splitter und Kinder (diese enthalten nur Schlüssel  $> s[m]$ )
  - Jetzt müssen wir den neuen Knoten  $L$  mit dem Splitter  $s[m]$  an den Elternknoten von  $x$  einfügen  
→ wir setzen  $c = L$ ,  $s = s[m]$  und  $x$  auf den Elternknoten von  $x$

# Einfügen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

`insert(4, v)  
insert(9, v)`



## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$
- der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einfügen.

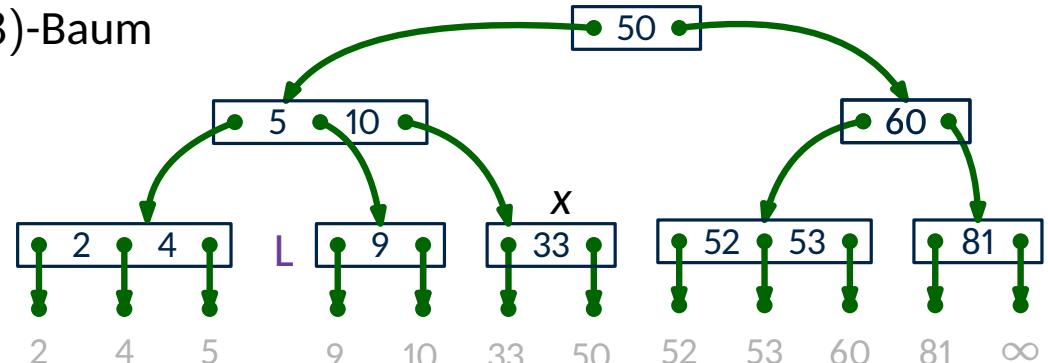
Wir wiederholen:

- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- Wenn**  $\deg(x) \leq b$  sind wir fertig.
- Ansonsten** ist der neue Grad  $\deg(x) = b + 1$  und wir machen eine **Spaltung (Split)** von  $x$ :
  - Wir wählen den mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$  zur Aufteilung (Median)
  - Wir erzeugen einen neuen Knoten  $L$  links von  $x$ , der alle kleineren Splitter als  $s[m]$  und das Kind  $c[m]$  enthält
  - $x$  behält alle verbleibenden Splitter und Kinder (diese enthalten nur Schlüssel  $> s[m]$ )
  - Jetzt müssen wir den neuen Knoten  $L$  mit dem Splitter  $s[m]$  an den Elternknoten von  $x$  einfügen  
→ wir setzen  $c = L$ ,  $s = s[m]$  und  $x$  auf den Elternknoten von  $x$

# Einfügen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

`insert(4, v)  
insert(9, v)`



## Strategie zum Einfügen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist oder eingefügt werden muss  
→ wir sind entweder fertig oder erzeugen ein neues Blatt  $b$  mit Schlüssel  $k$
- der Elternknoten  $x$  muss einen neuen Splitter  $s = k$  und Kind  $c = b$  einfügen.

Wir wiederholen:

- füge Splitter  $s$  und Kind  $c$  in das Splitterarray und Kindarray von  $x$  ein
- Wenn**  $\deg(x) \leq b$  sind wir fertig.
- Ansonsten** ist der neue Grad  $\deg(x) = b + 1$  und wir machen eine **Spaltung (Split)** von  $x$ :
  - Wir wählen den mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$  zur Aufteilung (Median)
  - Wir erzeugen einen neuen Knoten  $L$  links von  $x$ , der alle kleineren Splitter als  $s[m]$  und das Kind  $c[m]$  enthält
  - $x$  behält alle verbleibenden Splitter und Kinder (diese enthalten nur Schlüssel  $> s[m]$ )
  - Jetzt müssen wir den neuen Knoten  $L$  mit dem Splitter  $s[m]$  an den Elternknoten von  $x$  einfügen  
→ wir setzen  $c = L$ ,  $s = s[m]$  und  $x$  auf den Elternknoten von  $x$

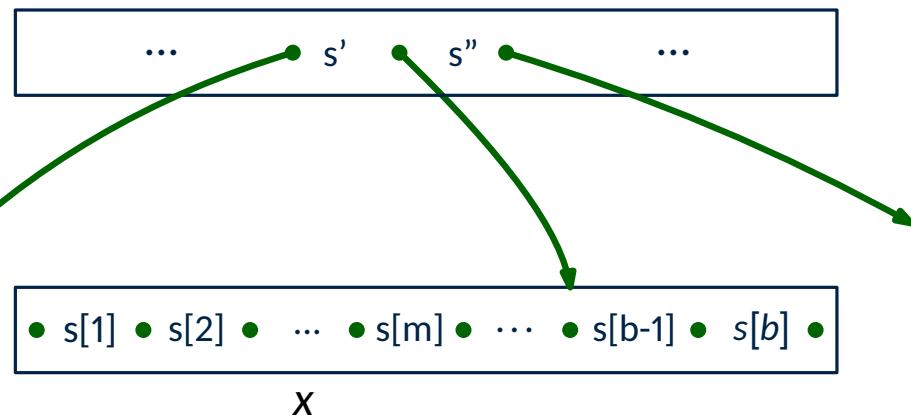
Frage: Was machen wir, wenn die Wurzel mehr als  $b$  Kinder bekommen soll?

Wir **spalten** die Wurzel in zwei Knoten ( $L$  und  $x$ ) und erzeugen eine neue Wurzel mit Kindern  $L$  und  $x$ .  
→ die Höhe des Baumes erhöht sich

# Spaltung eines Knotens

---

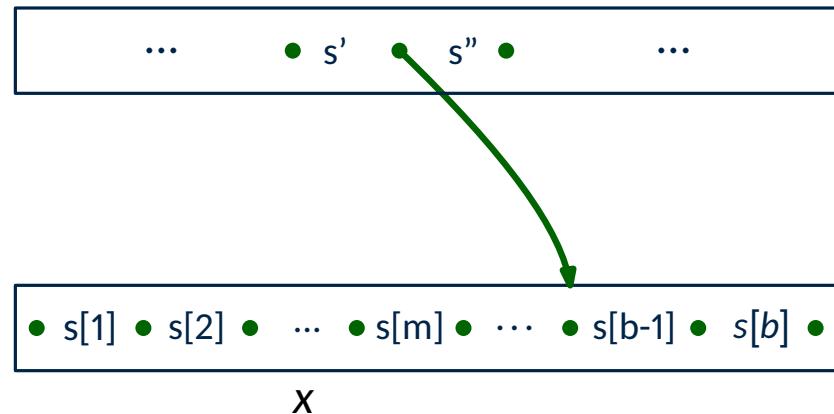
Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:



# Spaltung eines Knotens

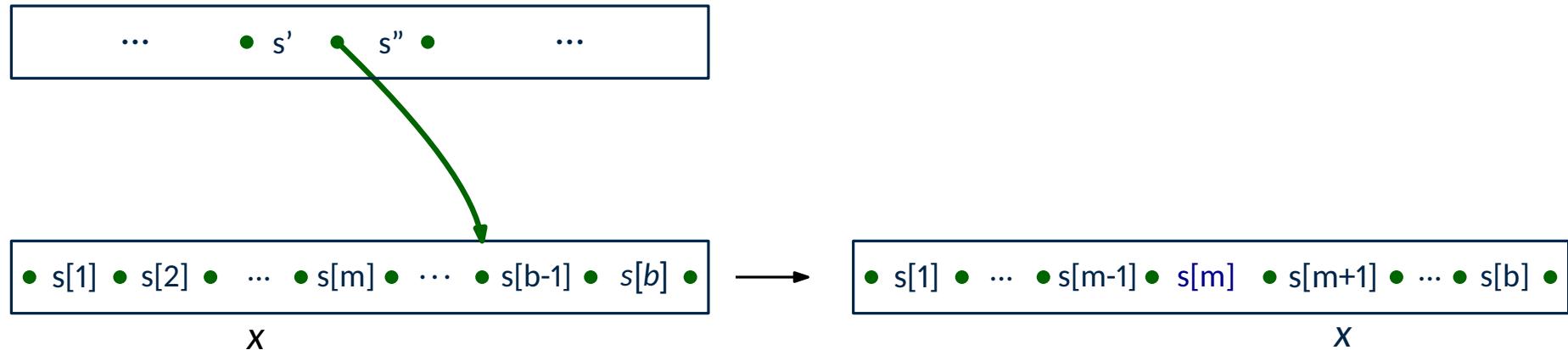
---

Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:



# Spaltung eines Knotens

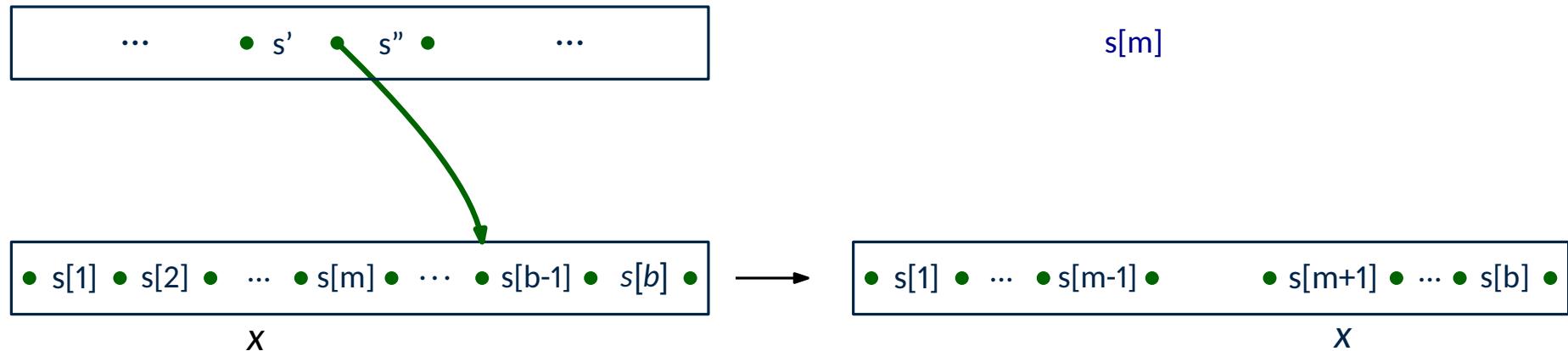
Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:



Wir spalten einen Knoten  $x$  mit  $b + 1$  Kindern am mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$

# Spaltung eines Knotens

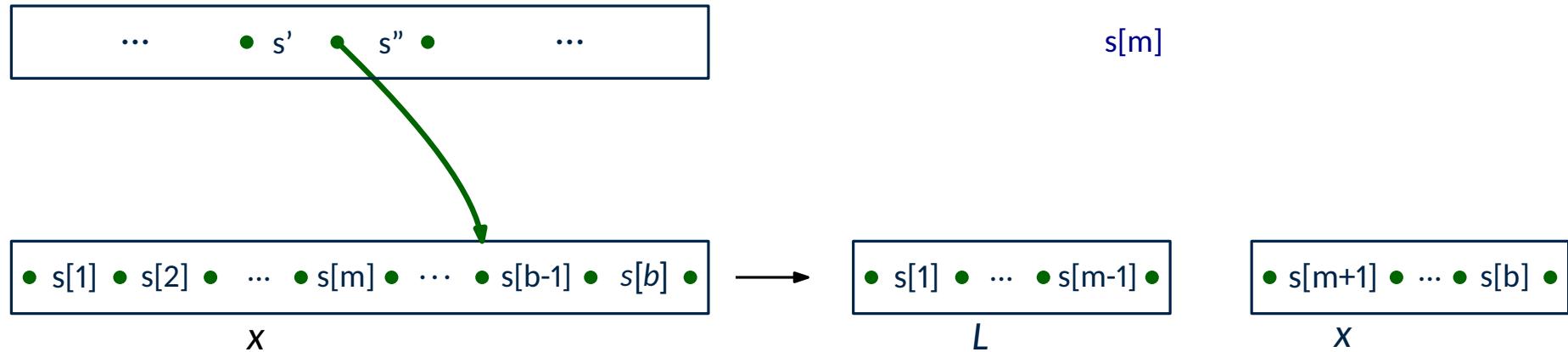
Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:



Wir spalten einen Knoten  $x$  mit  $b + 1$  Kindern am mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$

# Spaltung eines Knotens

Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:

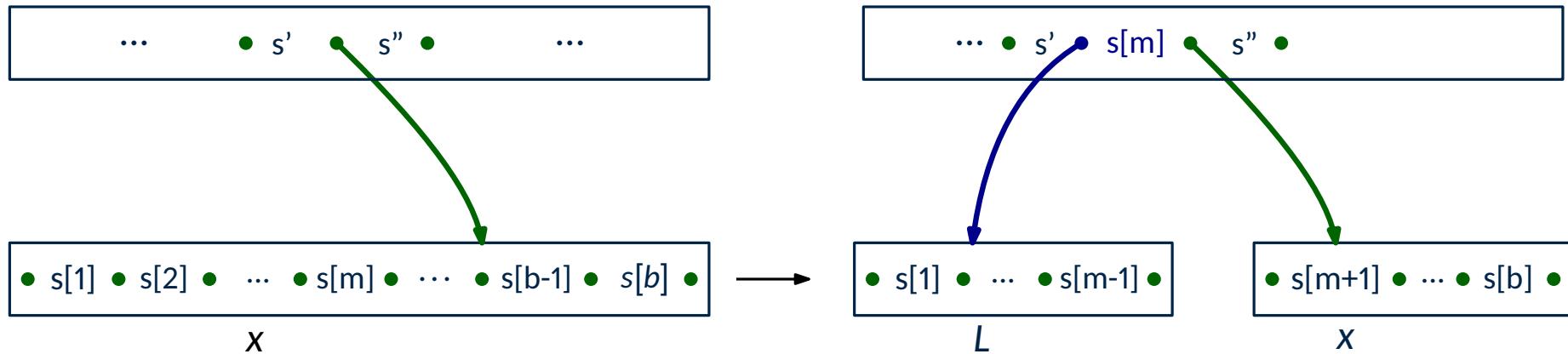


Wir spalten einen Knoten  $x$  mit  $b + 1$  Kindern am mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$

→  $L$  bekommt die kleineren  $m = \lceil \frac{b}{2} \rceil$  Kinder

# Spaltung eines Knotens

Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:



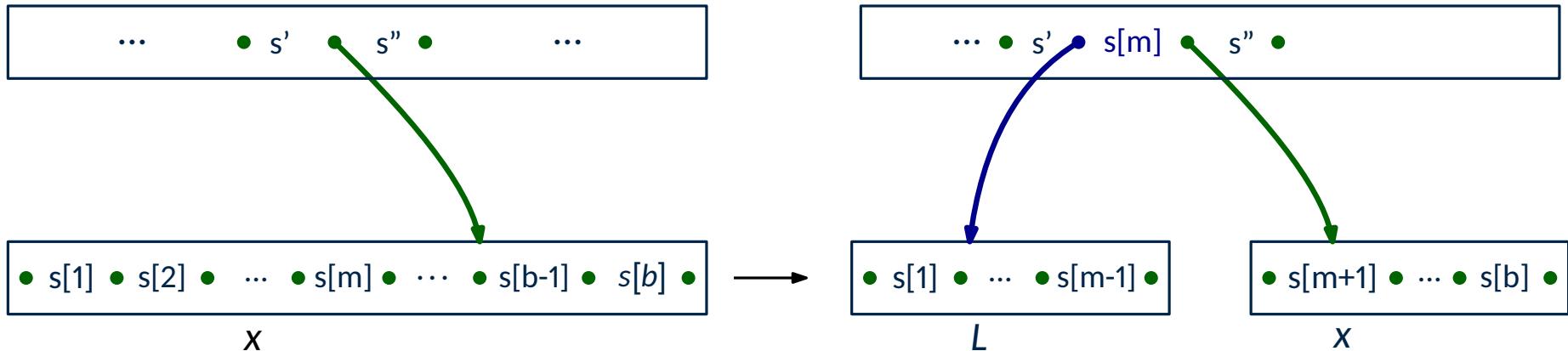
Wir spalten einen Knoten  $x$  mit  $b + 1$  Kindern am mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$

→  $L$  bekommt die kleineren  $m = \lceil \frac{b}{2} \rceil$  Kinder

→  $x$  behält die größeren  $(b + 1) - m$  Kinder

# Spaltung eines Knotens

Spaltung eines Knoten  $x$  mit  $b + 1$  Kindern:



Wir spalten einen Knoten  $x$  mit  $b + 1$  Kindern am mittleren Splitter  $s[m]$  mit  $m = \lceil \frac{b}{2} \rceil$

→  $L$  bekommt die kleineren  $m = \lceil \frac{b}{2} \rceil$  Kinder

→  $x$  behält die größeren  $(b + 1) - m$  Kinder

Warum gilt immer, dass  $x$  und  $L$  je mindestens  $a$  Kinder haben?

Weil  $b \geq 2a - 1$  ↗ Übung

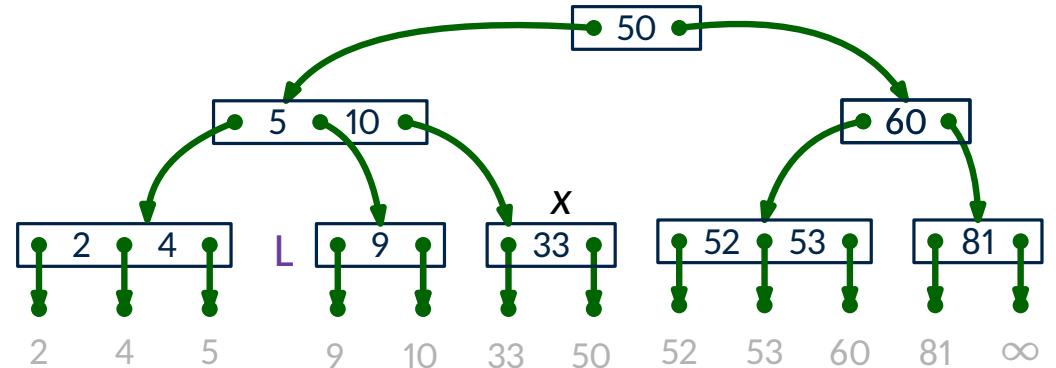
**Laufzeit** der Spaltung von  $x$ : · Verschieben von  $m$  Zeiger in ein neues Kinderarray für  $L$

· Verschieben von  $m - 1$  Splitter in ein neues Splitterarray für  $L$

· Einfügen von  $s[m]$  und den Zeiger auf  $L$  in den Elternknoten von  $x$

**Gesamlaufzeit:**  $O(b)$  → also  $O(1)$  wenn  $b$  eine Konstante ist.

# Laufzeit der Einfügeoperation



**Lemma.**

Wir können in Zeit  $O(\log_a(n) \cdot b)$  in einen  $(a, b)$ -Baum einfügen

**Beweis:** Wir suchen die Position für das neue Blatt in Zeit  $O(\log(b) \cdot \log_a n)$ .

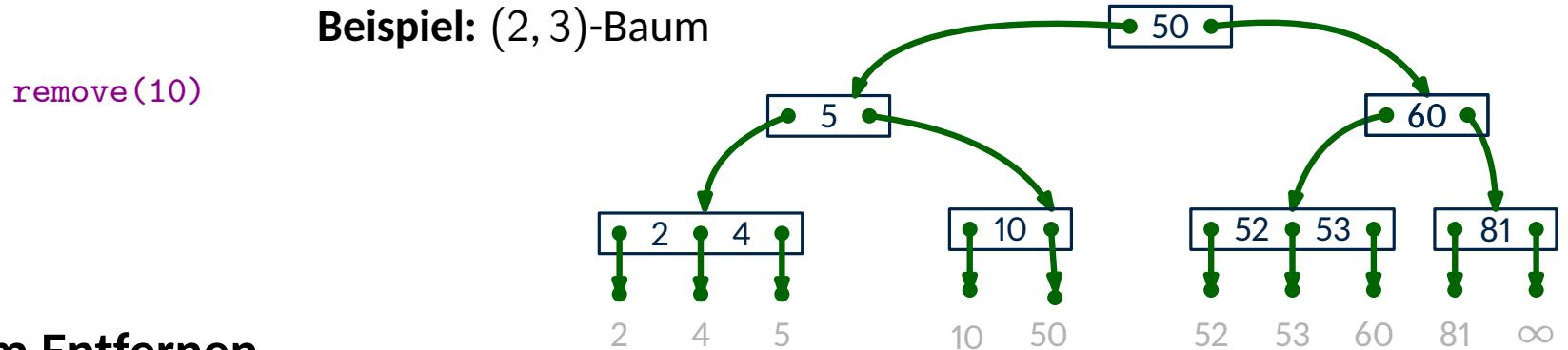
Wir fügen das neue Blatt in Zeit  $O(1)$  ein.

Jeder Vorfahre des neuen Blattes wird höchstens einmal aufgespalten.

Da die Höhe des Baumes  $O(\log_a n)$  ist, führen wir höchstens  $O(\log_a n)$  Spalteoperationen aus.

Da jede Spalteoperation  $O(b)$  benötigt, erhalten wir die angegebene Gesamtaufzeit. □

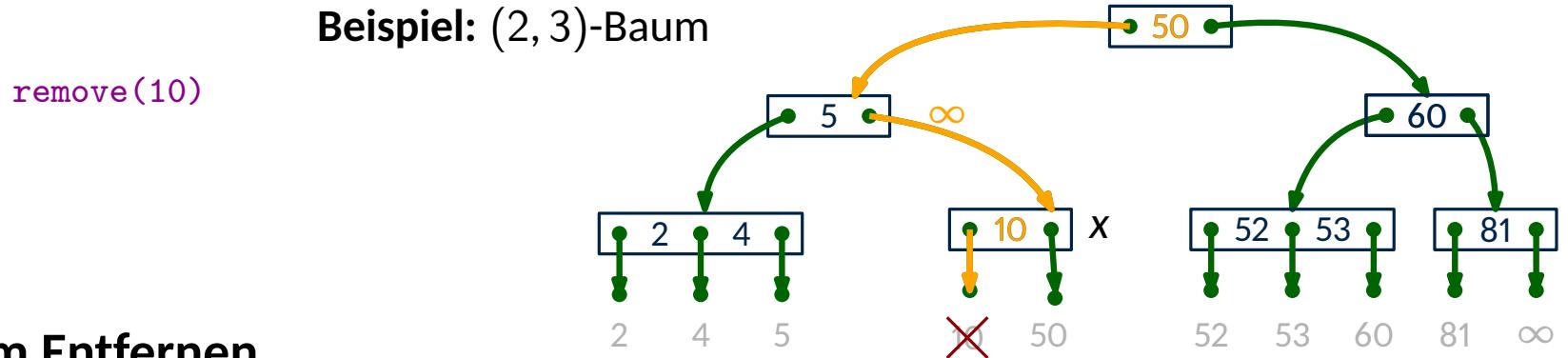
# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss

# Entfernen im $(a, b)$ -Baum

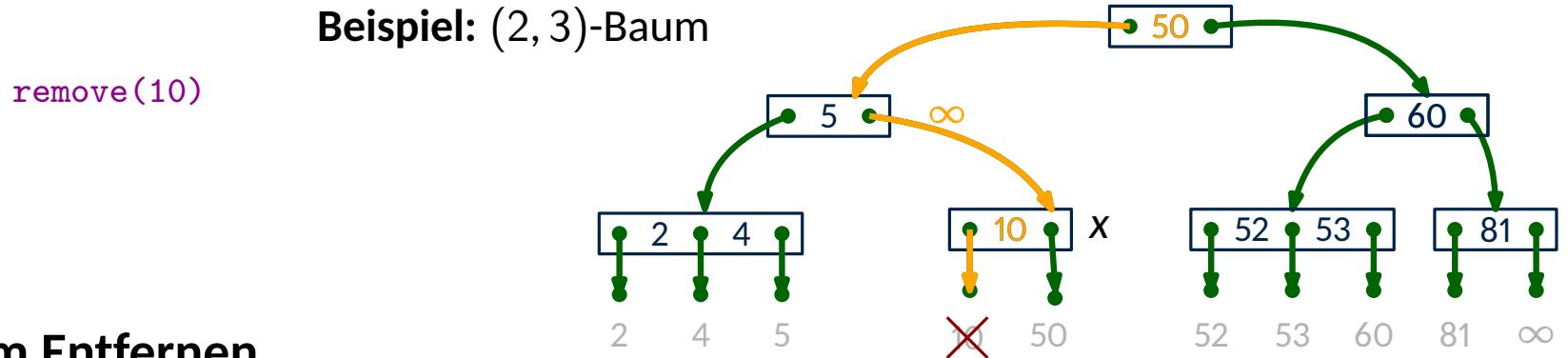


## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

(Sonderfall letztes Kind ↗ Folie 13)

# Entfernen im $(a, b)$ -Baum



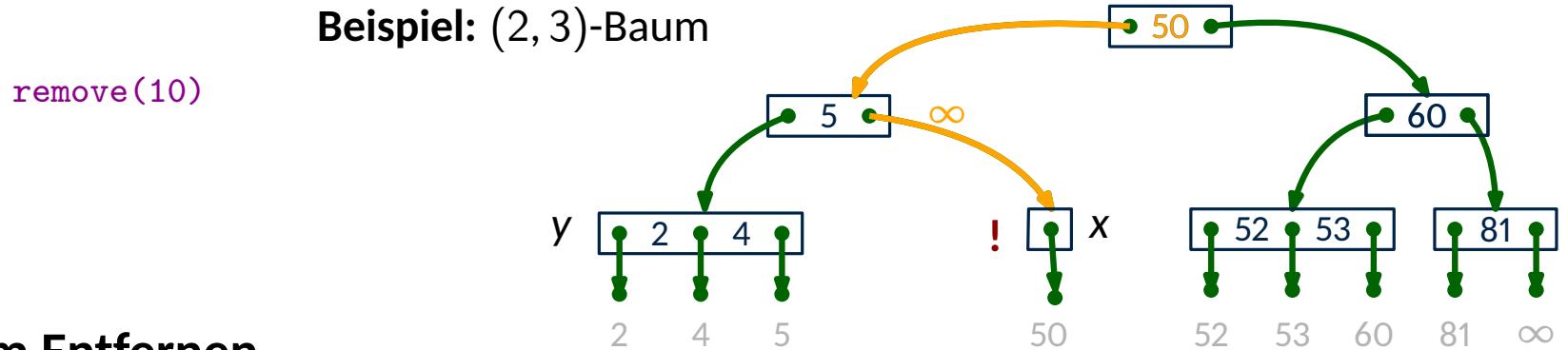
## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

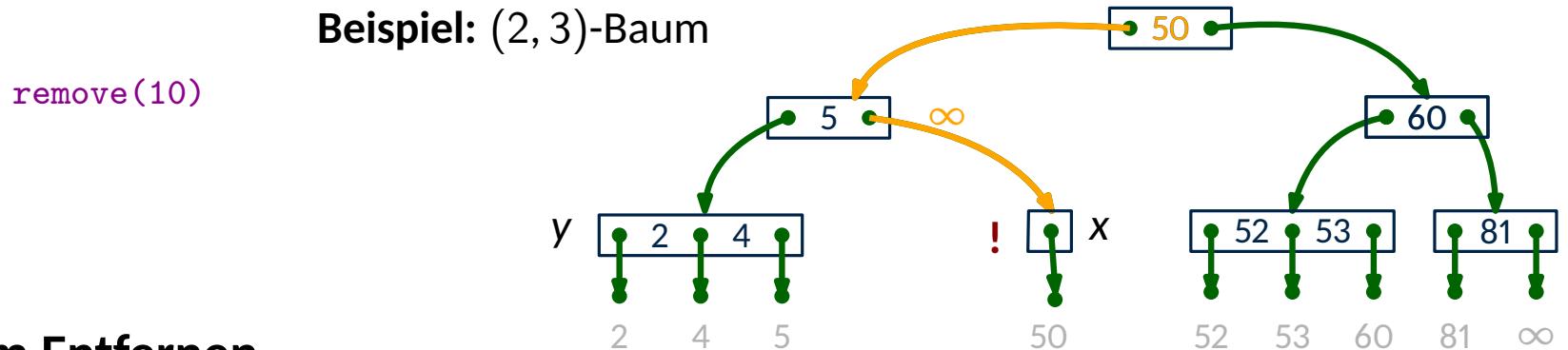
Wir wiederholen:

- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

Fall 1:

Fall 2:

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

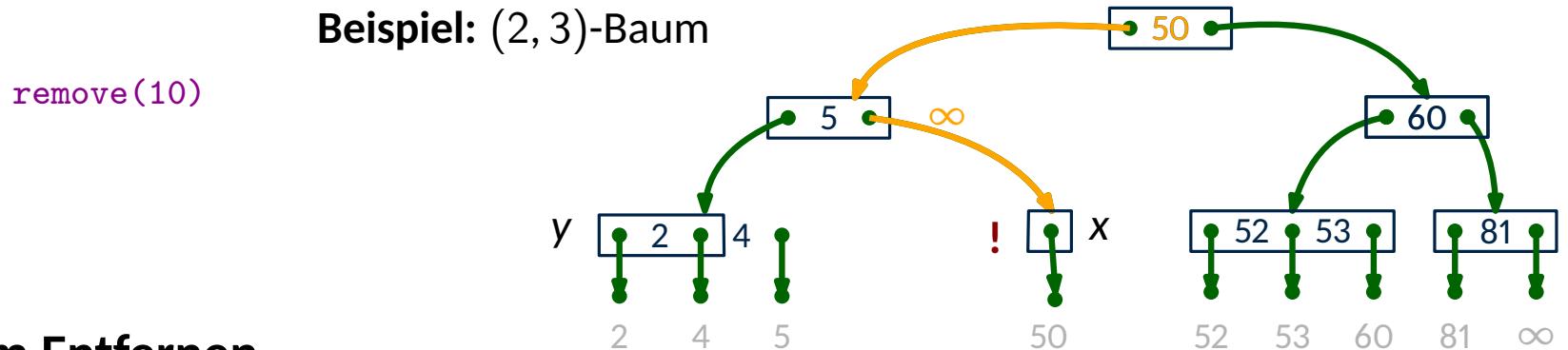
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

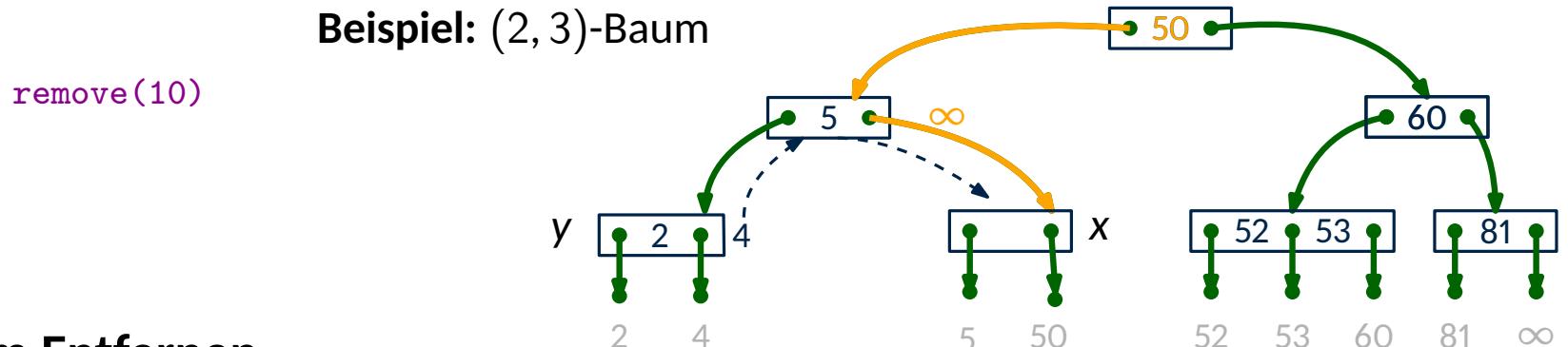
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

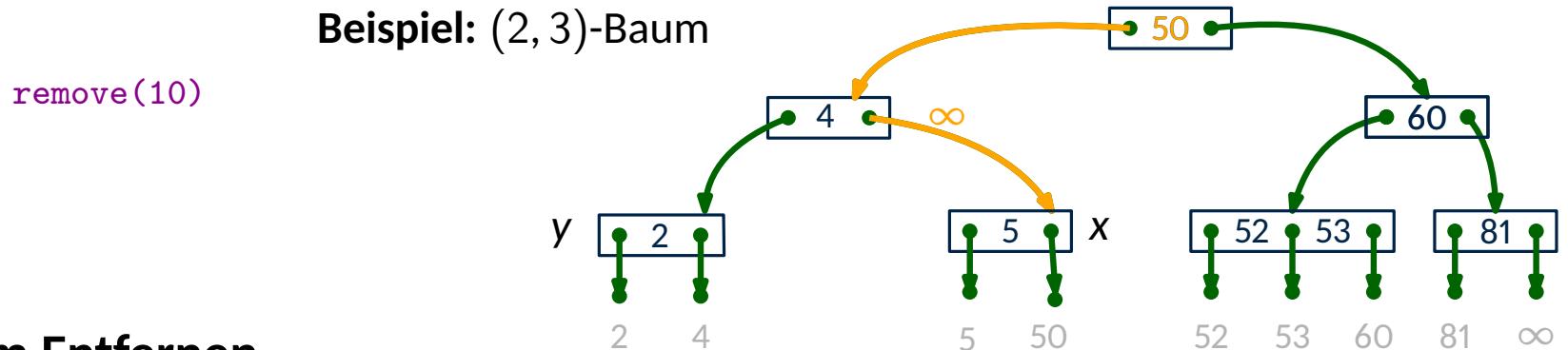
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

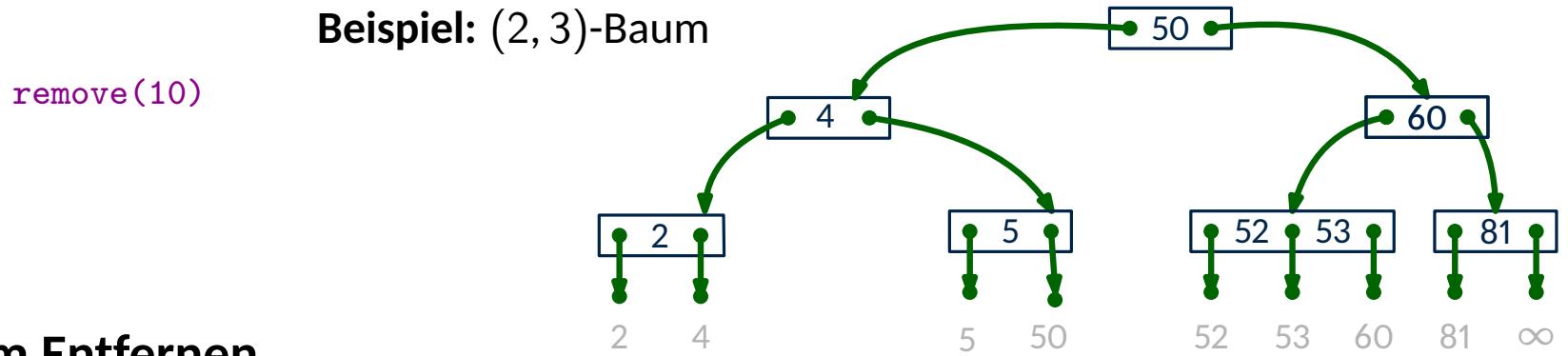
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

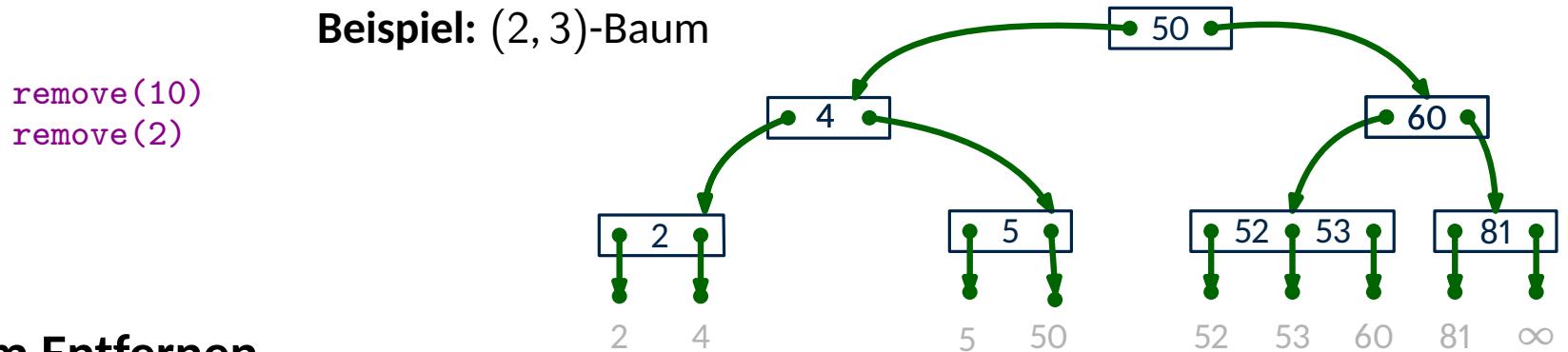
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

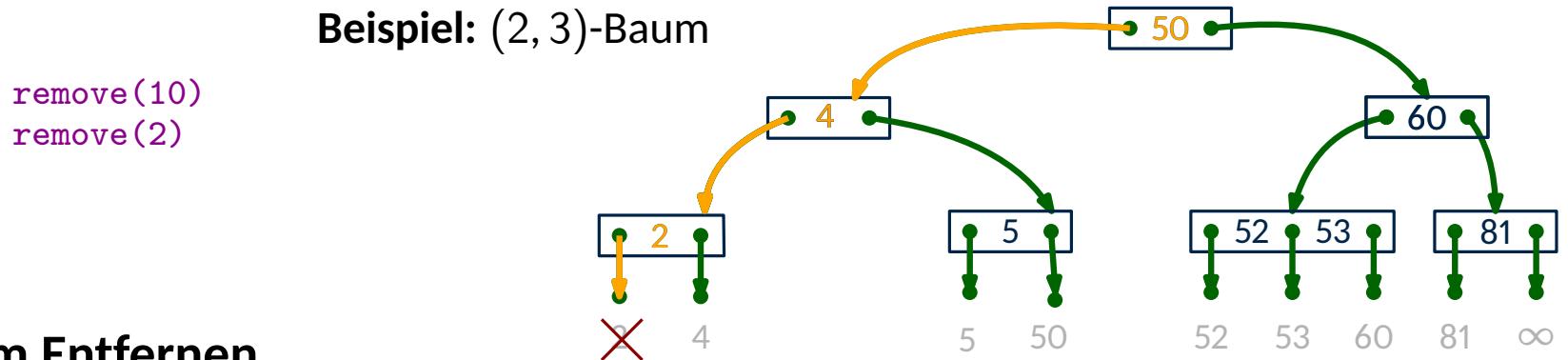
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

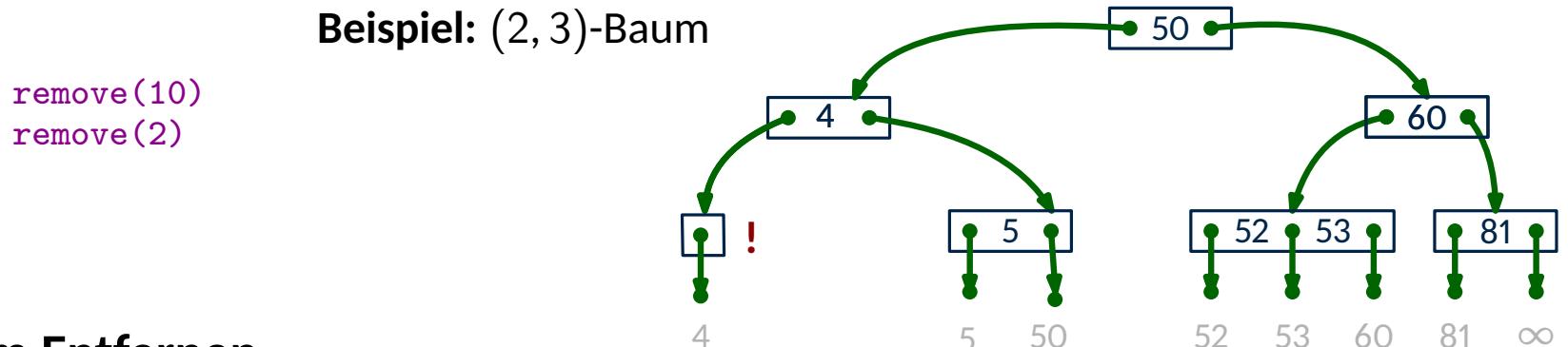
- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

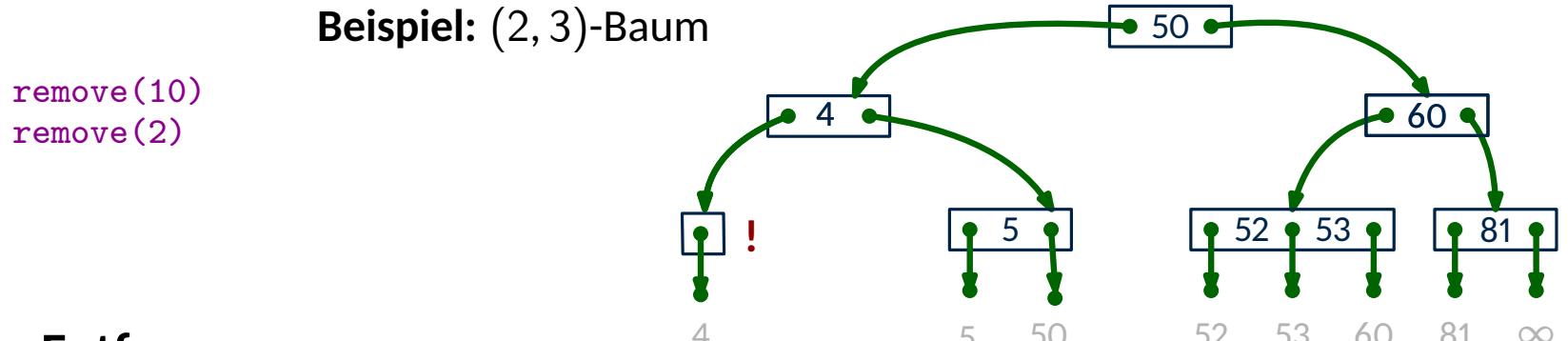
- Wenn  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**

# Entfernen im $(a, b)$ -Baum



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

- Wenn  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

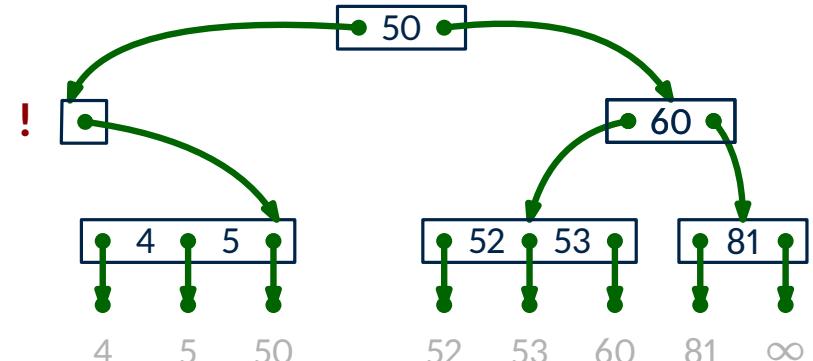
**Fall 2:**  $y$  hat  $a$  Kinder

- wir **fusionieren**  $x$  und  $y$ :  $x$  gibt alle Kinder und Schlüssel an  $y$  ab.
- dabei gibt Elter  $p$  den entsprechenden Splitter an  $y$  ab und verliert ein Kind.

# Entfernen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

`remove(10)  
remove(2)`



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

- **Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- **Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

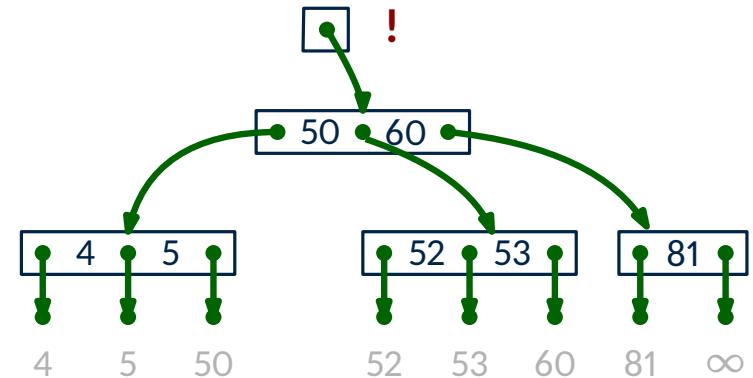
**Fall 2:**  $y$  hat  $a$  Kinder

- wir **fusionieren**  $x$  und  $y$ :  $x$  gibt alle Kinder und Schlüssel an  $y$  ab.
- dabei gibt Elter  $p$  den entsprechenden Splitter an  $y$  ab und verliert ein Kind.

# Entfernen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

`remove(10)`  
`remove(2)`



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

- **Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- **Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

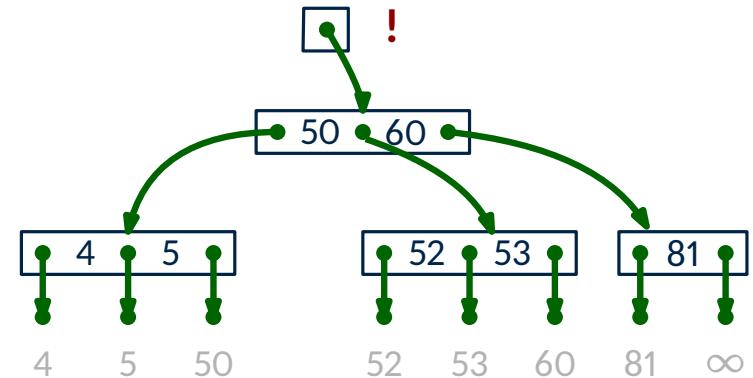
**Fall 2:**  $y$  hat  $a$  Kinder

- wir **fusionieren**  $x$  und  $y$ :  $x$  gibt alle Kinder und Schlüssel an  $y$  ab.
- dabei gibt Elter  $p$  den entsprechenden Splitter an  $y$  ab und verliert ein Kind.  
→ wir betrachten nun den Elternknoten  $p$  als Knoten  $x$

# Entfernen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

`remove(10)`  
`remove(2)`



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

- Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**  $y$  hat  $a$  Kinder

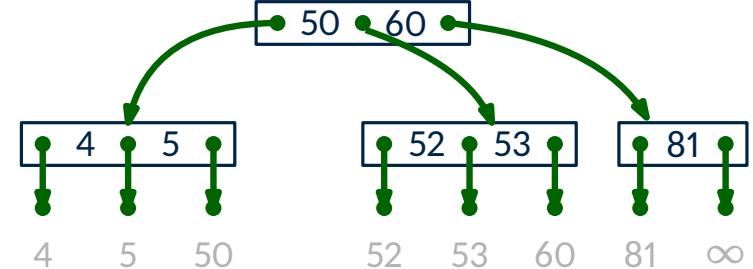
- wir **fusionieren**  $x$  und  $y$ :  $x$  gibt alle Kinder und Schlüssel an  $y$  ab.
- dabei gibt Elter  $p$  den entsprechenden Splitter an  $y$  ab und verliert ein Kind.  
→ wir betrachten nun den Elternknoten  $p$  als Knoten  $x$

**Sonderfall Wurzel:** Wenn die Wurzel nur noch ein Kind hat, wird ihr Kind die neue Wurzel (oder der Baum leer)

# Entfernen im $(a, b)$ -Baum

Beispiel:  $(2, 3)$ -Baum

remove(10)  
remove(2)



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter  
(Sonderfall letztes Kind ↗ Folie 13)

Wir wiederholen:

- **Wenn**  $\deg(x) \geq a$ , dann beenden wir (Sonderfall Wurzel: später)
- **Ansonsten** ist  $\deg(x) = a - 1$ . Wir betrachten den linken oder rechten Geschwisterknoten  $y$  von  $x$

**Fall 1:**  $y$  hat  $a + t$  Kinder mit  $t \geq 1$

- wir **gleichen** die Grade von  $x$  und  $y$  an:  $y$  gibt  $\lceil \frac{t}{2} \rceil \geq 1$  Kinder an  $x$  ab.
- dabei gibt  $y$  einen Splitter an seinen Elter  $p$  und  $p$  einen Splitter an  $x$  ab.

**Fall 2:**  $y$  hat  $a$  Kinder

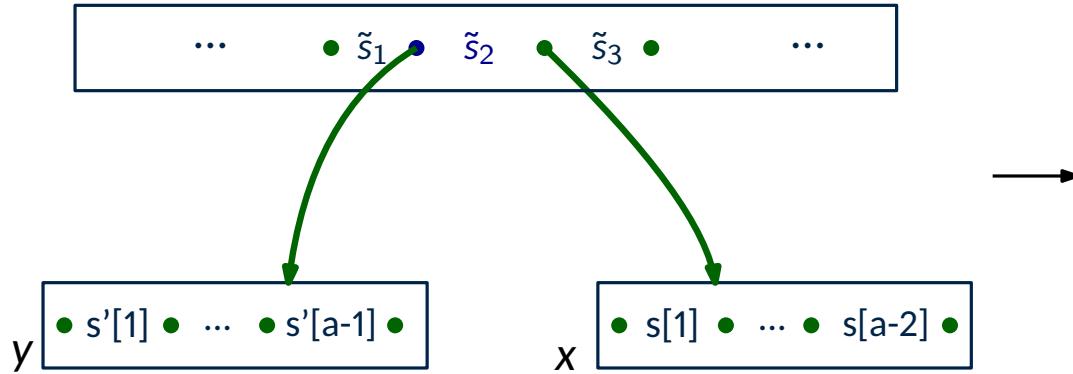
- wir **fusionieren**  $x$  und  $y$ :  $x$  gibt alle Kinder und Schlüssel an  $y$  ab.
- dabei gibt Elter  $p$  den entsprechenden Splitter an  $y$  ab und verliert ein Kind.  
→ wir betrachten nun den Elternknoten  $p$  als Knoten  $x$

**Sonderfall Wurzel:** Wenn die Wurzel nur noch ein Kind hat, wird ihr Kind die neue Wurzel (oder der Baum leer)  
→ Höhe des Baumes verringert sich

# Fall 2: Fusionieren

Fusionieren eines Knotens  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar  $y$  mit  $a$  Kindern

Fusion mit einem rechten Nachbar ist symmetrisch

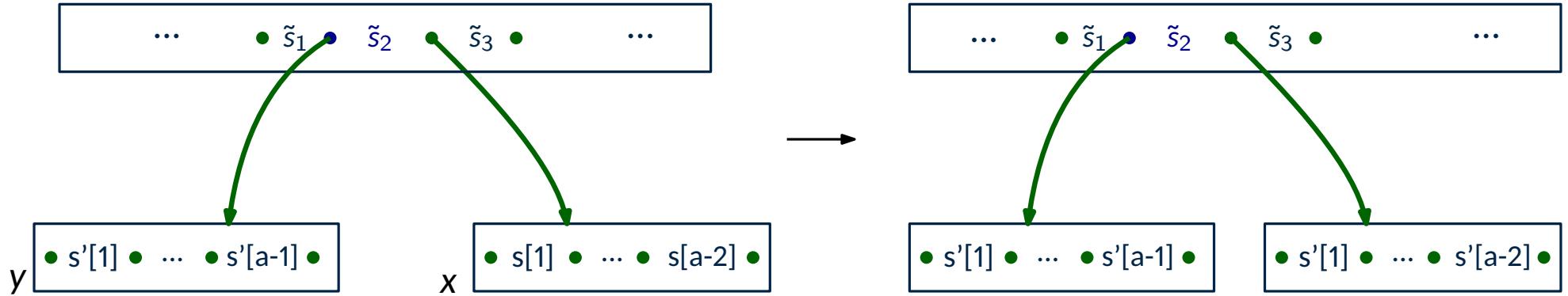


Wir fusionieren einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn mit  $a$  Kindern.

# Fall 2: Fusionieren

Fusionieren eines Knotens  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar  $y$  mit  $a$  Kindern

Fusion mit einem rechten Nachbar ist symmetrisch

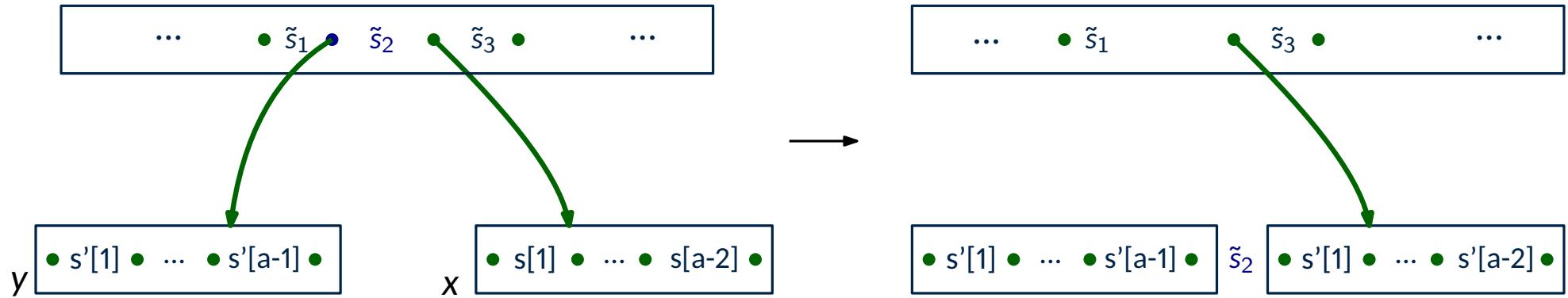


Wir fusionieren einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn mit  $a$  Kindern.

# Fall 2: Fusionieren

Fusionieren eines Knotens  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar  $y$  mit  $a$  Kindern

Fusion mit einem rechten Nachbar ist symmetrisch



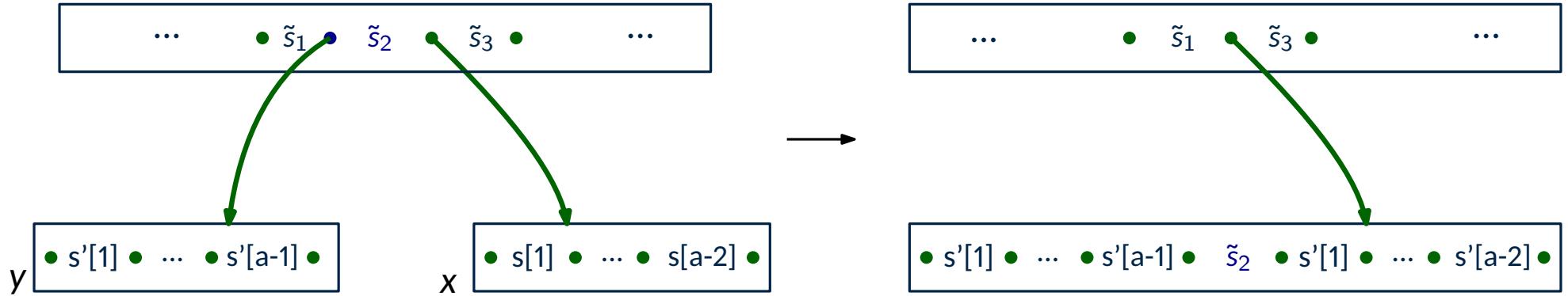
Wir fusionieren einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn mit  $a$  Kindern.

- der Elternknoten verliert ein Kind und gibt den dazugehörigen Schlüssel an den fusionierten Knoten ab

# Fall 2: Fusionieren

Fusionieren eines Knotens  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar  $y$  mit  $a$  Kindern

Fusion mit einem rechten Nachbar ist symmetrisch



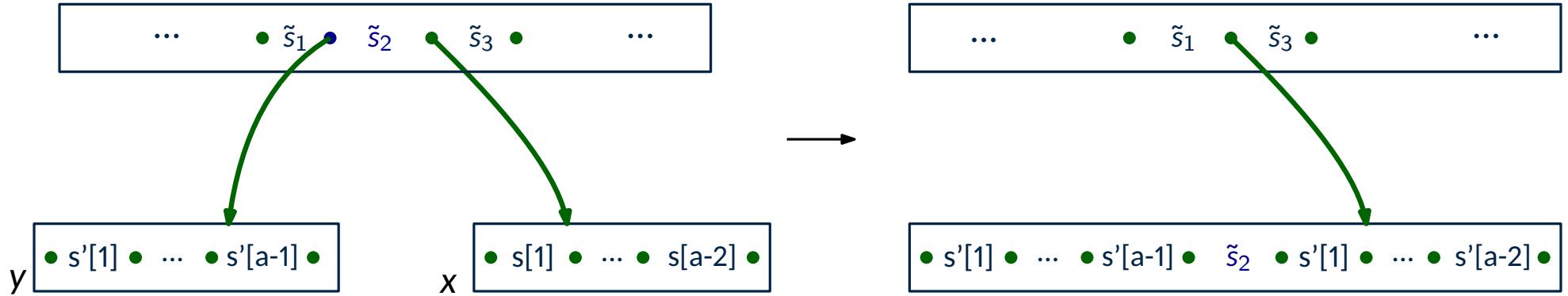
Wir fusionieren einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn mit  $a$  Kindern.

- der Elternknoten verliert ein Kind und gibt den dazugehörigen Schlüssel an den fusionierten Knoten ab

# Fall 2: Fusionieren

Fusionieren eines Knotens  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar  $y$  mit  $a$  Kindern

Fusion mit einem rechten Nachbar ist symmetrisch



Wir fusionieren einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn mit  $a$  Kindern.

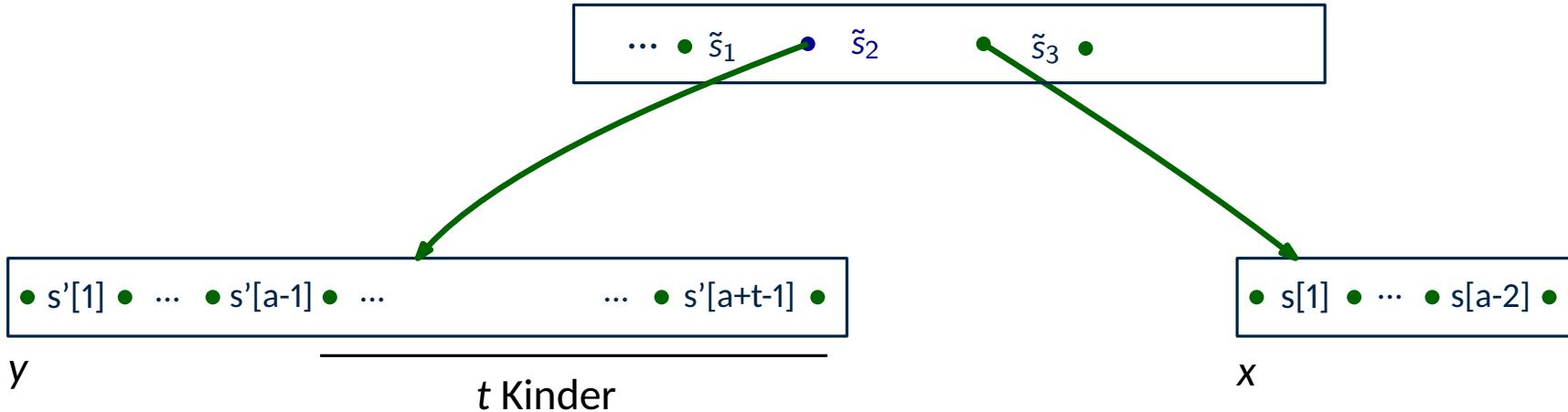
- der Elternknoten verliert ein Kind und gibt den dazugehörigen Schlüssel an den fusionierten Knoten ab
- der neue Knoten hat  $2a - 1 \leq b$  Kinder

Laufzeit einer Fusion:  $O(b)$  → also  $O(1)$ , wenn  $b$  eine Konstante ist.

# Fall 1: Angleichen

Angleichen eines Knoten  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar mit  $a + t$  Kindern

Angleichen mit einem rechten Nachbar ist symmetrisch

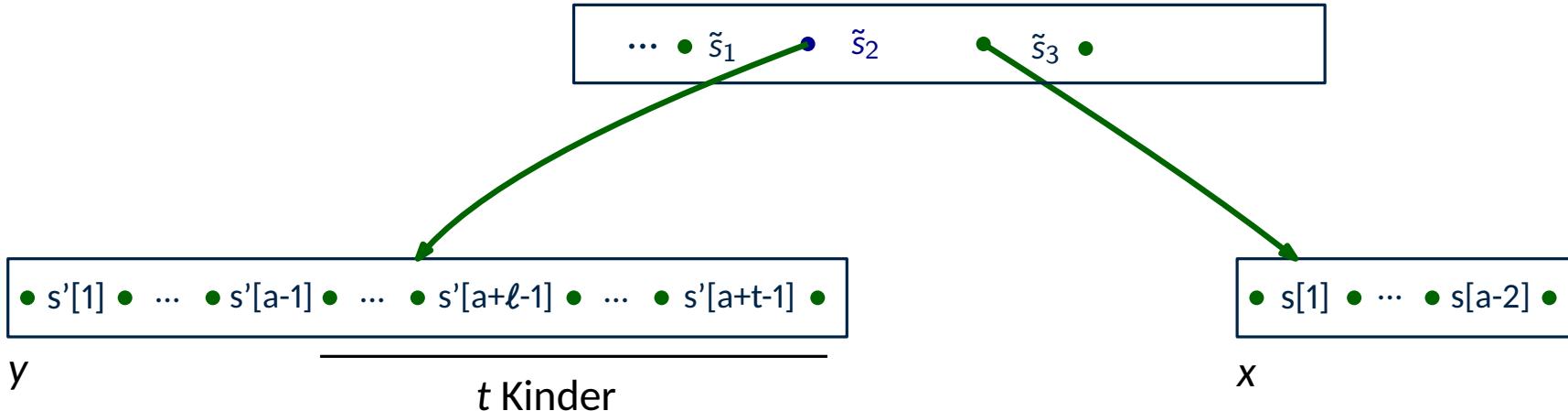


Wir gleichen einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn  $y$  mit  $a + t$  Kindern an:

# Fall 1: Angleichen

Angleichen eines Knoten  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar mit  $a + t$  Kindern

Angleichen mit einem rechten Nachbar ist symmetrisch



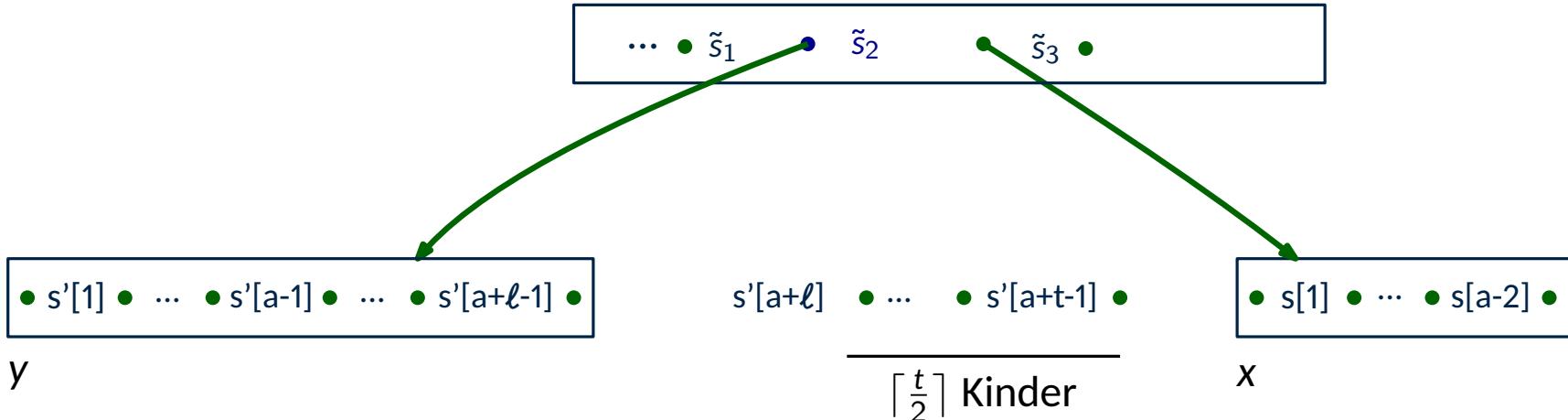
Wir gleichen einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn  $y$  mit  $a + t$  Kindern an:

Wir setzen  $\ell = \lfloor \frac{t}{2} \rfloor$  und lassen den Nachbarn  $a + \ell$  Kinder behalten

# Fall 1: Angleichen

Angleichen eines Knoten  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar mit  $a + t$  Kindern

Angleichen mit einem rechten Nachbar ist symmetrisch



Wir gleichen einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn  $y$  mit  $a + t$  Kindern an:

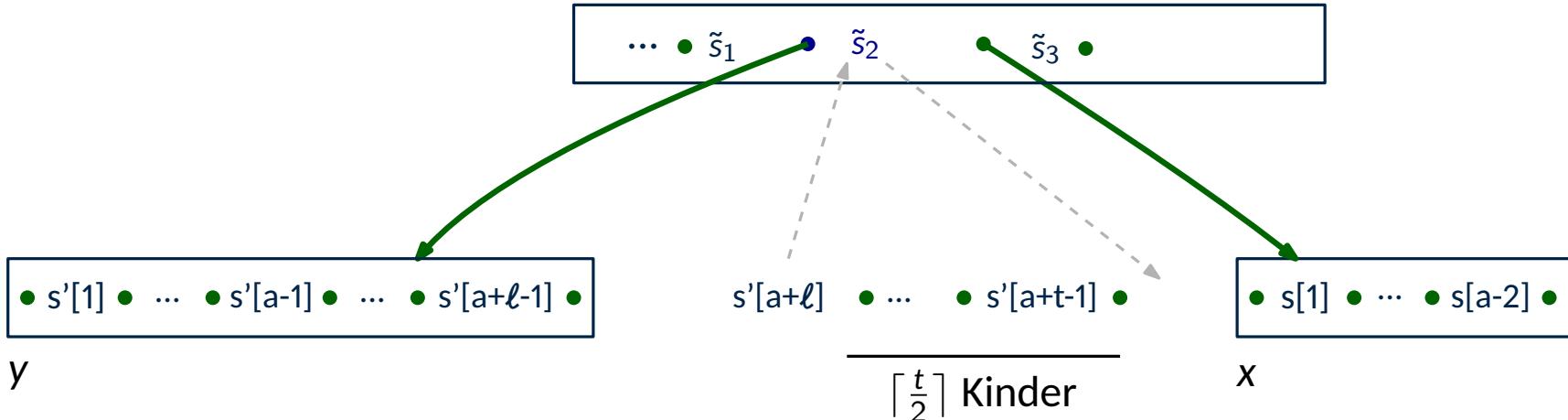
Wir setzen  $\ell = \lfloor \frac{t}{2} \rfloor$  und lassen den Nachbarn  $a + \ell$  Kinder behalten

$\rightarrow x$  bekommt  $t - \ell = \lceil \frac{t}{2} \rceil \geq 1$  zusätzliche Kinder

# Fall 1: Angleichen

Angleichen eines Knoten  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar mit  $a + t$  Kindern

Angleichen mit einem rechten Nachbar ist symmetrisch



Wir gleichen einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn  $y$  mit  $a + t$  Kindern an:

Wir setzen  $\ell = \lfloor \frac{t}{2} \rfloor$  und lassen den Nachbarn  $a + \ell$  Kinder behalten

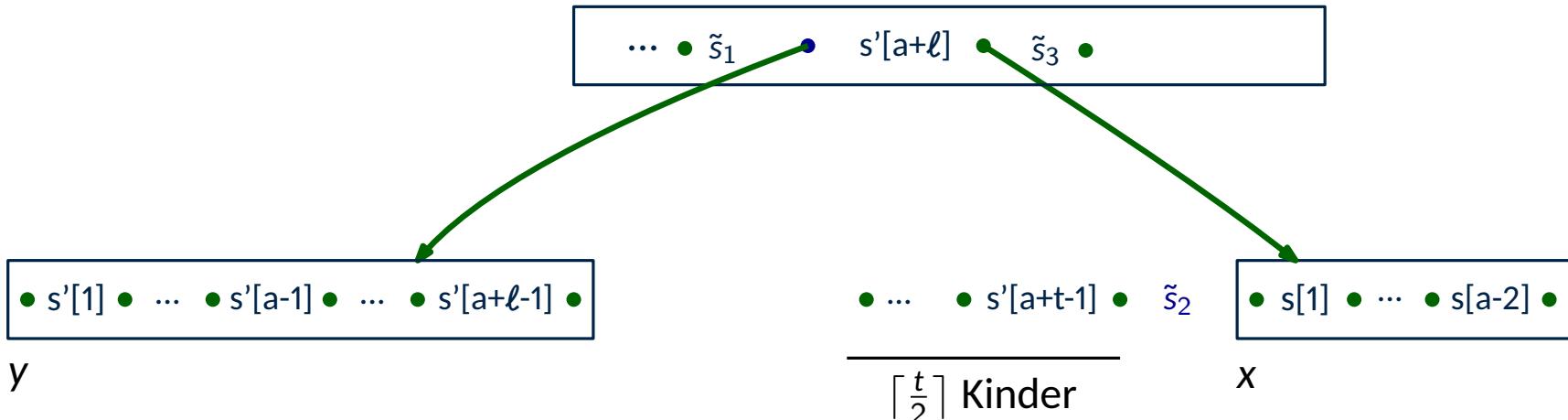
$\rightarrow x$  bekommt  $t - \ell = \lceil \frac{t}{2} \rceil \geq 1$  zusätzliche Kinder

Dabei gibt  $y$  einen Schlüssel an den Elternknoten ab und der Elternknoten einen Schlüssel an  $x$  ab.

# Fall 1: Angleichen

Angleichen eines Knoten  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar mit  $a + t$  Kindern

Angleichen mit einem rechten Nachbar ist symmetrisch



Wir gleichen einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn  $y$  mit  $a + t$  Kindern an:

Wir setzen  $\ell = \lfloor \frac{t}{2} \rfloor$  und lassen den Nachbarn  $a + \ell$  Kinder behalten

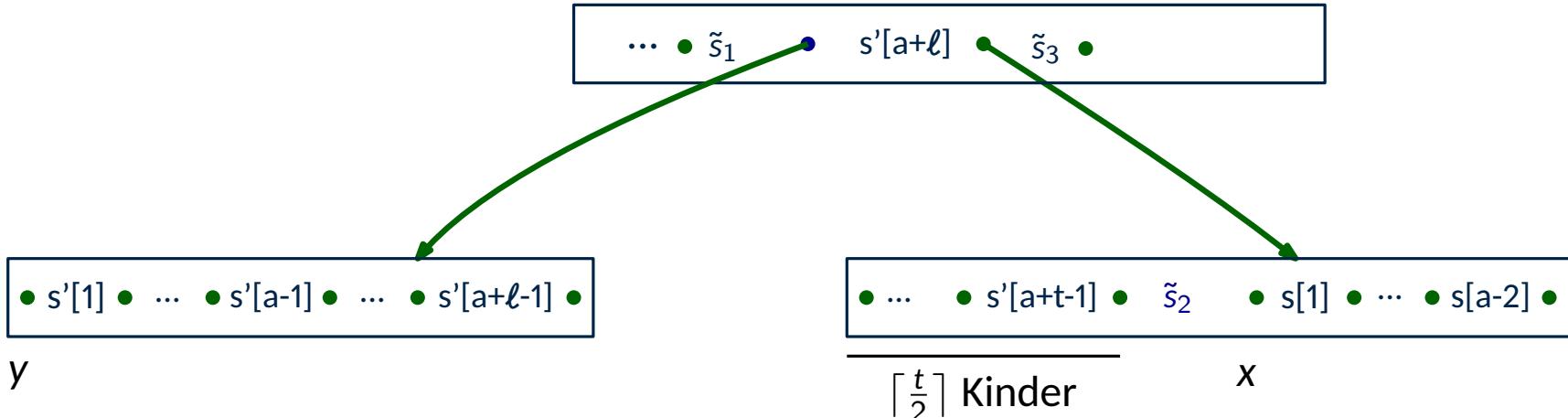
$\rightarrow x$  bekommt  $t - \ell = \lceil \frac{t}{2} \rceil \geq 1$  zusätzliche Kinder

Dabei gibt  $y$  einen Schlüssel an den Elternknoten ab und der Elternknoten einen Schlüssel an  $x$  ab.

# Fall 1: Angleichen

Angleichen eines Knoten  $x$  mit  $a - 1$  Kindern mit einem linken Nachbar mit  $a + t$  Kindern

Angleichen mit einem rechten Nachbar ist symmetrisch



Wir gleichen einen Knoten  $x$  mit  $a - 1$  Kindern mit einem Nachbarn  $y$  mit  $a + t$  Kindern an:

Wir setzen  $\ell = \lfloor \frac{t}{2} \rfloor$  und lassen den Nachbarn  $a + \ell$  Kinder behalten

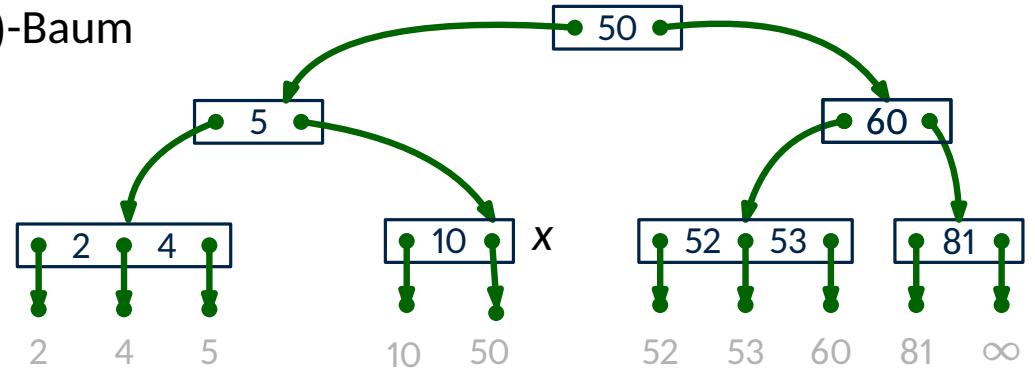
$\rightarrow x$  bekommt  $t - \ell = \lceil \frac{t}{2} \rceil \geq 1$  zusätzliche Kinder

Dabei gibt  $y$  einen Schlüssel an den Elternknoten ab und der Elternknoten einen Schlüssel an  $x$  ab.

**Laufzeit** einer Angleichung:  $O(b)$   $\rightarrow$  also  $O(1)$ , wenn  $b$  eine Konstante ist.

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind

Beispiel:  $(2, 3)$ -Baum

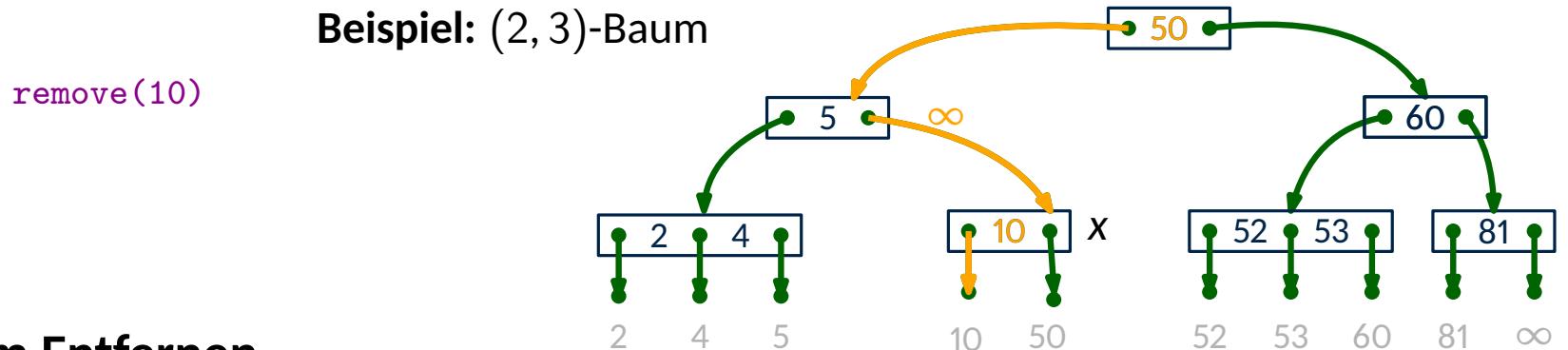


## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letztes Kind vom Elternknoten  $x$  ist.

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind

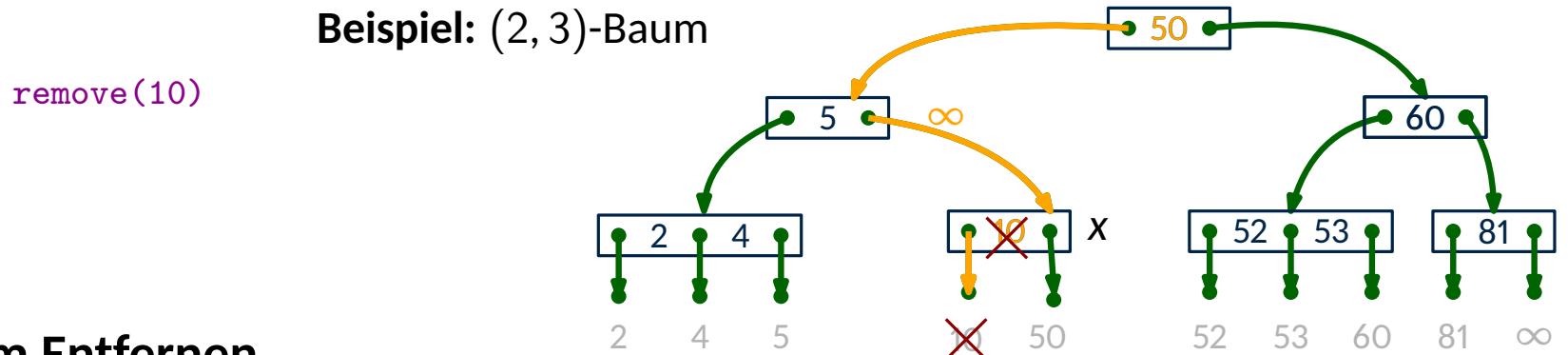


## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letztes Kind vom Elternknoten  $x$  ist.

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



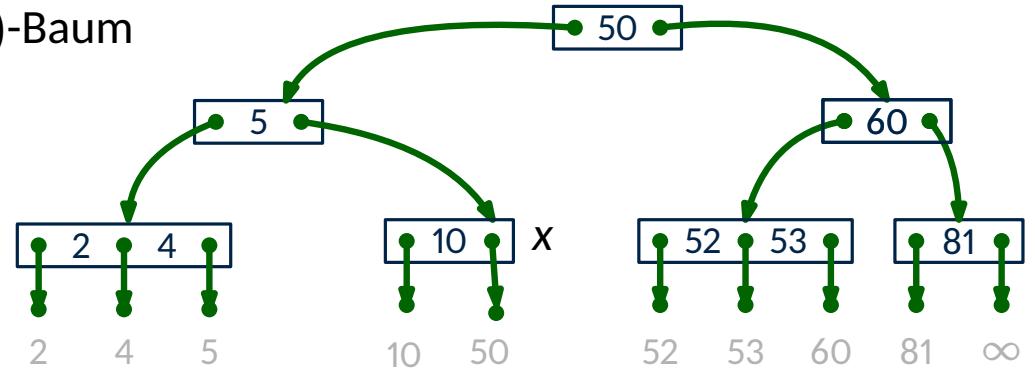
## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letztes Kind vom Elternknoten  $x$  ist.

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind

Beispiel:  $(2, 3)$ -Baum

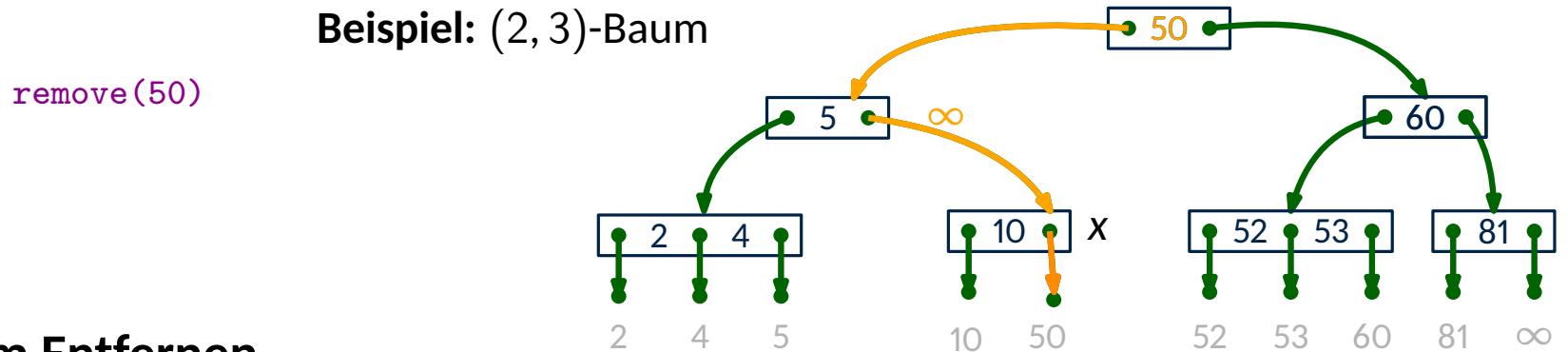


## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letztes Kind vom Elternknoten  $x$  ist.

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind

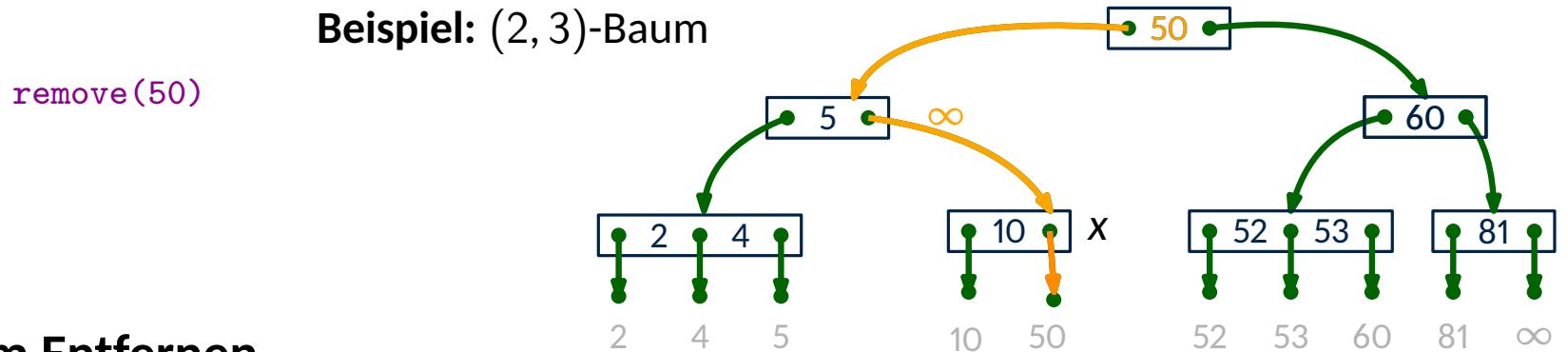


## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letztes Kind vom Elternknoten  $x$  ist.

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



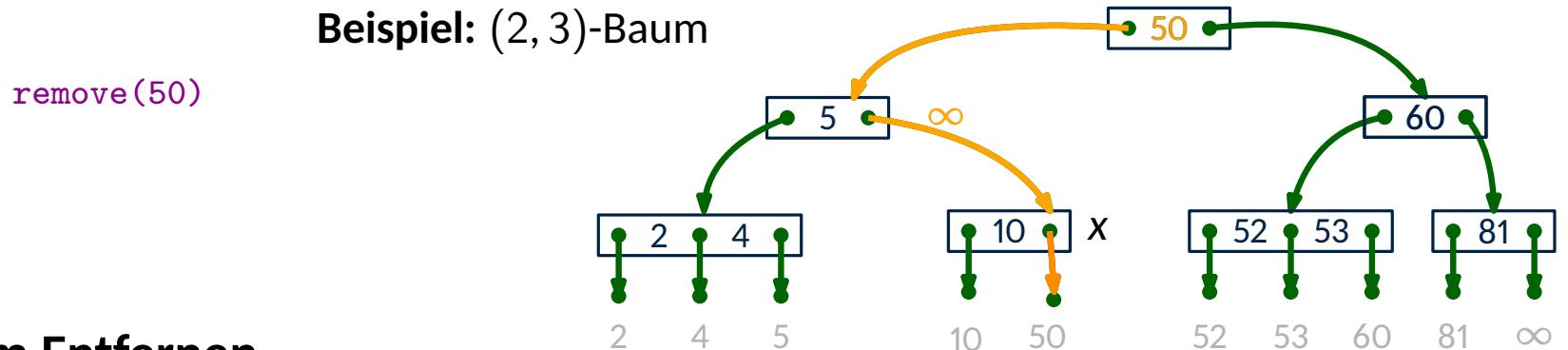
## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letzte Kind vom Elternknoten  $x$  ist.

Frage: Wie können wir den zugehörigen Splitter entfernen?

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



## Strategie zum Entfernen

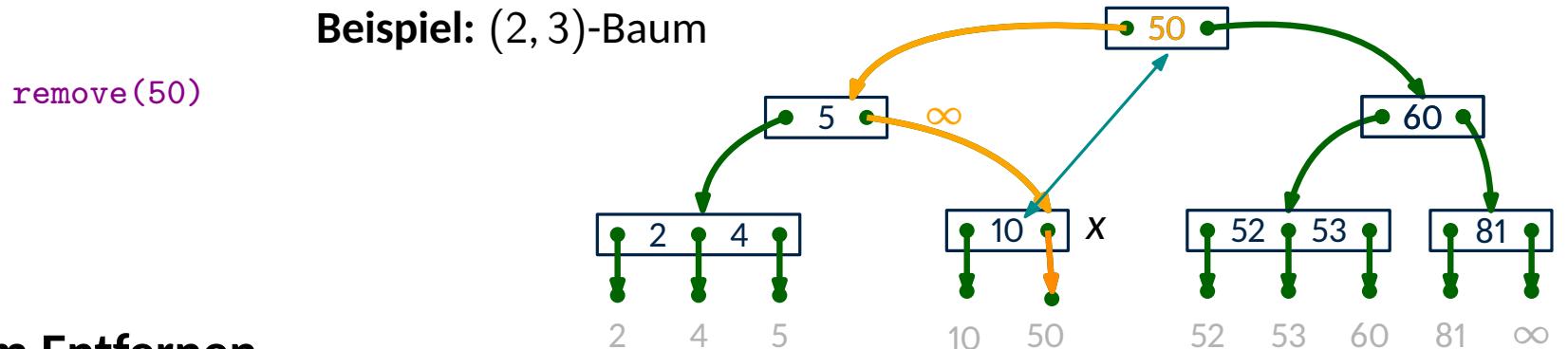
- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letztes Kind vom Elternknoten  $x$  ist.

Frage: Wie können wir den zugehörigen Splitter entfernen?

Wenn Blatt das letzte Kind des Elternknoten ist, vertausche Splitter mit Wert  $k$  durch Vorgänger-Splitter

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



## Strategie zum Entfernen

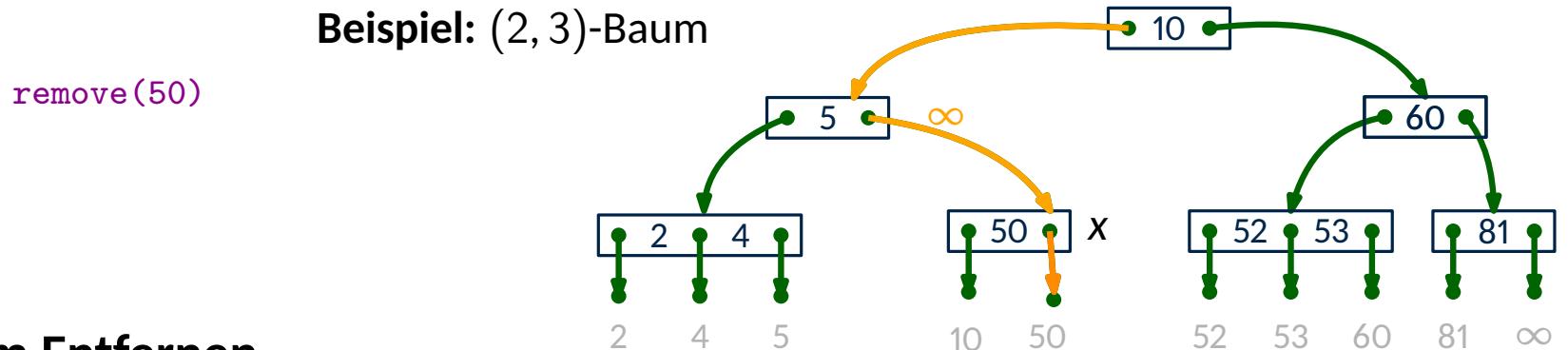
- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letzte Kind vom Elternknoten  $x$  ist.

**Frage:** Wie können wir den zugehörigen Splitter entfernen?

Wenn Blatt das letzte Kind des Elternknoten ist, vertausche Splitter mit Wert  $k$  durch Vorgänger-Splitter

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



## Strategie zum Entfernen

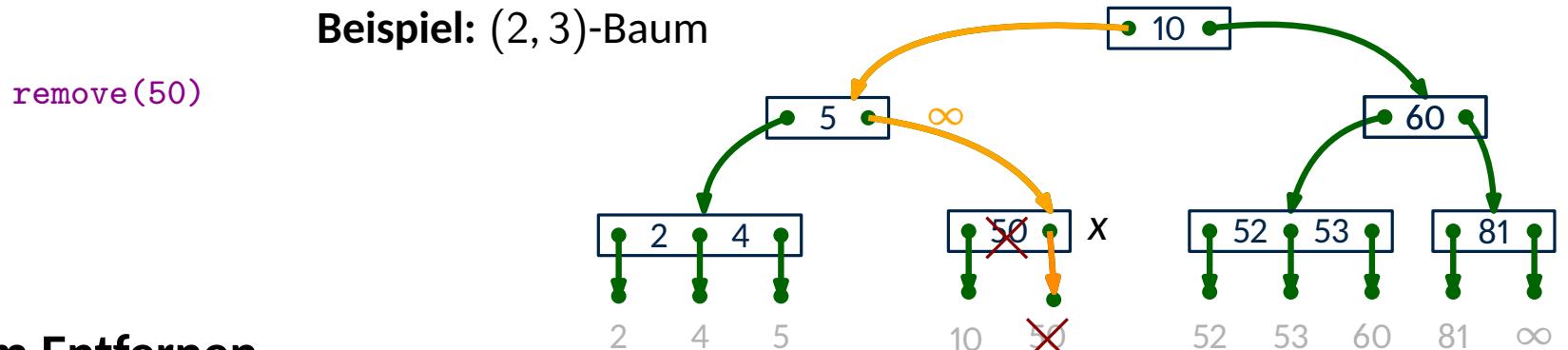
- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letzte Kind vom Elternknoten  $x$  ist.

**Frage:** Wie können wir den zugehörigen Splitter entfernen?

Wenn Blatt das letzte Kind des Elternknoten ist, vertausche Splitter mit Wert  $k$  durch Vorgänger-Splitter

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



## Strategie zum Entfernen

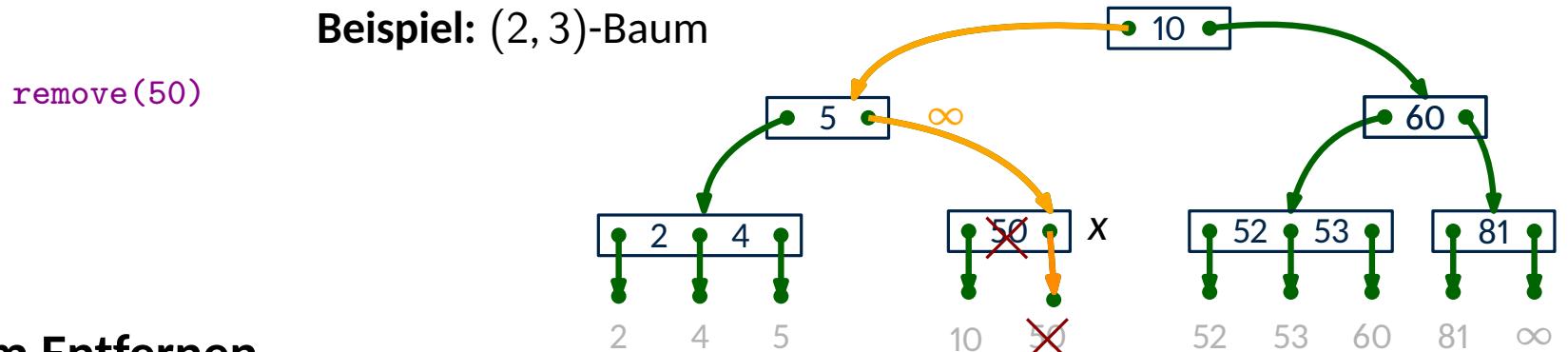
- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

Betrachten nun den Sonderfall, dass das Blatt das letzte Kind vom Elternknoten  $x$  ist.

**Frage:** Wie können wir den zugehörigen Splitter entfernen?

Wenn Blatt das letzte Kind des Elternknoten ist, vertausche Splitter mit Wert  $k$  durch Vorgänger-Splitter  
→ können nun das Blatt und Splitter  $k$  entfernen

# Entfernen im $(a, b)$ -Baum: Sonderfall letztes Kind



## Strategie zum Entfernen

- suche die Stelle, an der der Schlüssel  $k$  bereits gespeichert ist und gelöscht werden muss  
→ wir entfernen das Blatt. Dessen Elternknoten  $x$  verliert also ein Kind  $c$  und einen Splitter

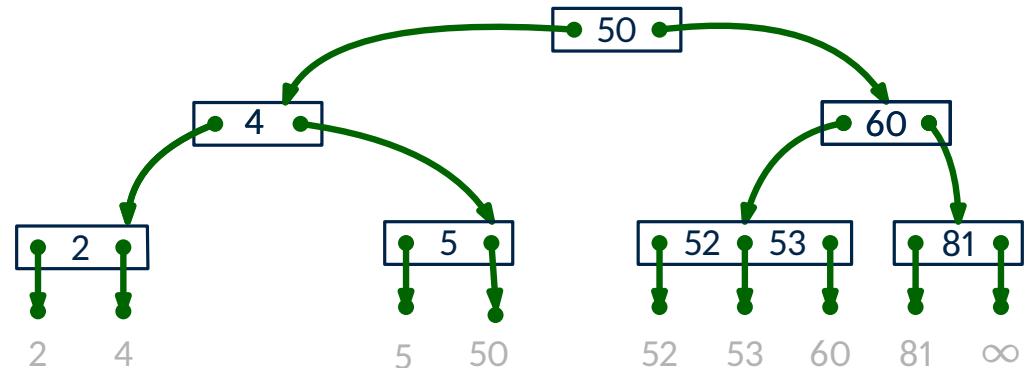
Betrachten nun den Sonderfall, dass das Blatt das letzte Kind vom Elternknoten  $x$  ist.

**Frage:** Wie können wir den zugehörigen Splitter entfernen?

Wenn Blatt das letzte Kind des Elternknoten ist, vertausche Splitter mit Wert  $k$  durch Vorgänger-Splitter  
→ können nun das Blatt und Splitter  $k$  entfernen

Der Rest funktioniert wie bisher.

# Laufzeit der Löschoperation



**Lemma.**

Wir können in Zeit  $O(\log_a(n) \cdot b)$  aus einem  $(a, b)$ -Baum entfernen.

**Beweis:** Wir suchen die Position für das zu entfernende Blatt in Zeit  $O(\log(b) \cdot \log_a n)$ .

Wir entfernen das Blatt in Zeit  $O(1)$ .

Für jeden Vorfahren des entfernten Blattes haben wir höchstens eine Fusions-Operation.

Weiterhin kann es höchstens eine Angleich-Operation (Fall 1) sowie eine Wurzellösung geben.

Da die Höhe des Baumes  $O(\log_a n)$  ist, führen wir höchstens  $O(\log_a n)$  dieser Operationen aus.

Da jede dieser Operationen in Zeit  $O(b)$  läuft, erhalten wir die angegebene Gesamlaufzeit. □

# Zusammenfassung

---

Seien  $a, b$  Konstanten mit  $a \geq 2, b \geq 2a - 1$ .

**Wörterbuch (Dictionary, Assoziatives Array)** via  $(a, b)$ -Bäumen

`S.find(k)` - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?  $O(\log n)$

`S.insert(e)` - füge  $e$  in  $S$  ein  $O(\log n)$

`S.remove(k)` - lösche das Element mit Schlüssel  $k$  aus  $S$   $O(\log n)$

Zusätzliche Operationen:

`S.min()` - gib das Element mit dem kleinsten Schlüssel zurück  $O(1)$   
`S.max()` - gib das Element mit dem größten Schlüssel zurück

Frage: Wie?

`S.successor(x)` - gib das Element mit dem nächstgrößeren Schlüssel zu Element  $x$  zurück  
`S.predecessor(x)` - gib das Element mit dem nächstkleineren Schlüssel zu Element  $x$  zurück  $O(1)$   
( $x$  ist durch Zeiger auf entsprechendes Blatt gegeben)

# Amortisierte Analyse der Änderungsoperationen

---

Bestandteile einer Einfüge- oder Löschoperation in  $(a, b)$ -Baum:

- $O(\log(b) \cdot \log_a n)$  Laufzeit für die Suche
- $O(b \log_a n)$  Laufzeit für Änderungen im Baum

Benötigen wir fast immer  $\Omega(\log n)$  viele Änderungen im Baum pro Einfügen oder Löschen?

# Amortisierte Analyse der Änderungsoperationen

---

Bestandteile einer Einfüge- oder Löschoperation in  $(a, b)$ -Baum:

- $O(\log(b) \cdot \log_a n)$  Laufzeit für die Suche
- $O(b \log_a n)$  Laufzeit für Änderungen im Baum      amortisiert nur  $O(1)$  Änderungen!

Benötigen wir fast immer  $\Omega(\log n)$  viele Änderungen im Baum pro Einfügen oder Löschen?

**Theorem.**

$T$  sei ein  $(a, b)$ -Baum, wobei:

- $a, b$  sind feste Konstanten mit  $a \geq 2$  und  $b \geq 2a$ .
- $T$  ist anfangs leer.

Für jede Sequenz von  $t$  Einfüge- oder Lösch-Operationen ist  
die Gesamtanzahl an Spaltungs- oder Fusions-Operationen in  $O(t)$ .

→ amortisierte Anzahl an Spaltungs- oder Fusions-Operationen nur  $O(1)$ .

**Beweis:** ↗ Tafelpräsentation

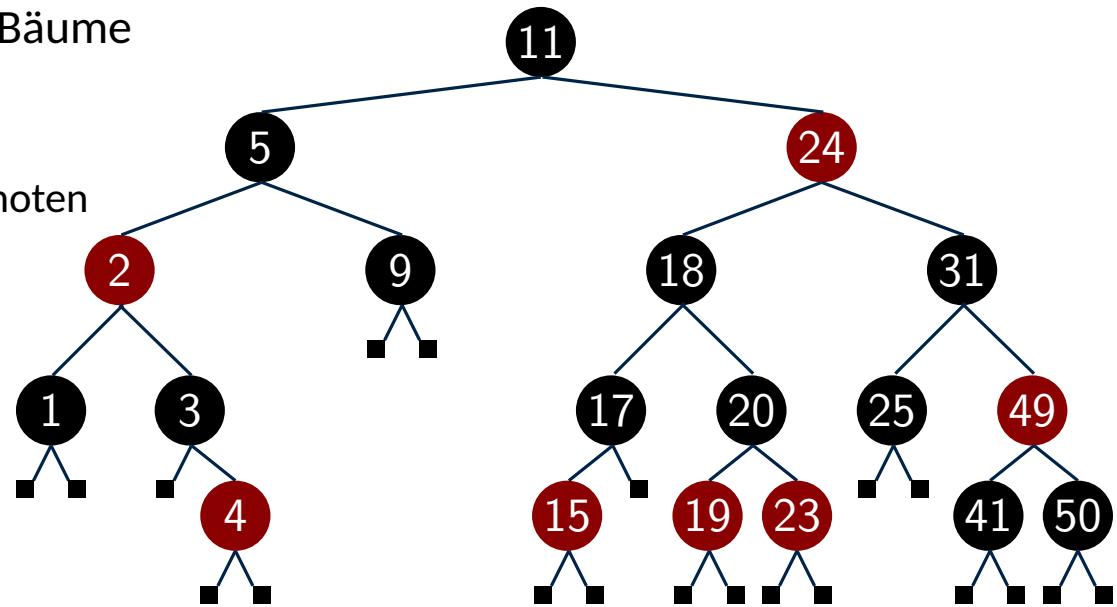
# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

---

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum  $\rightarrow$  (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter  $\rightsquigarrow$  Superknoten

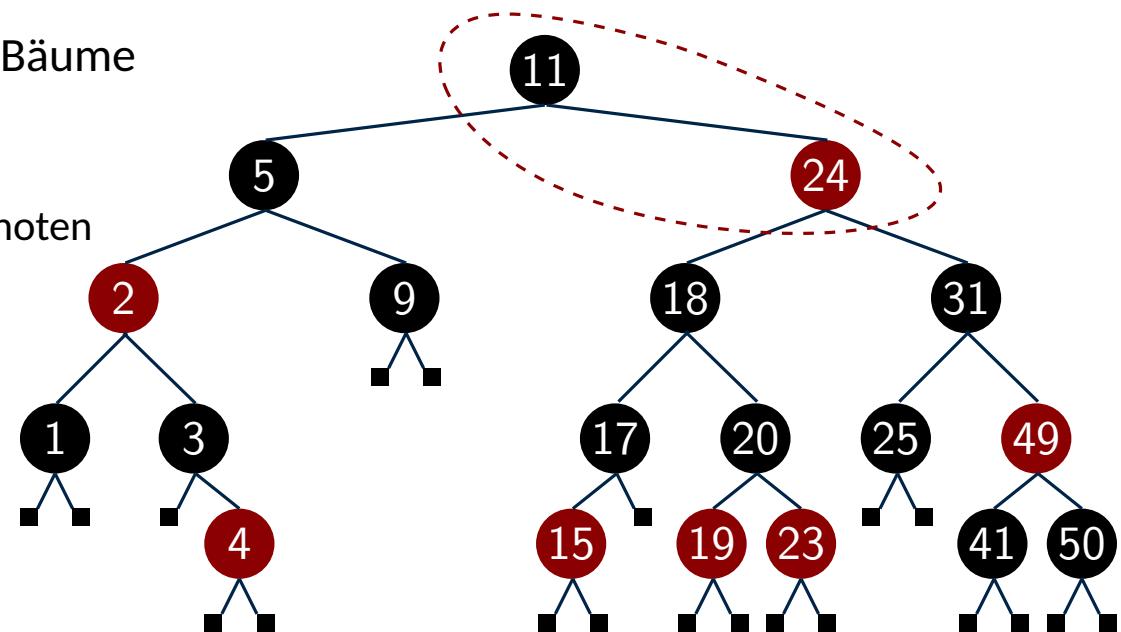


# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum → (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter ↵ Superknoten



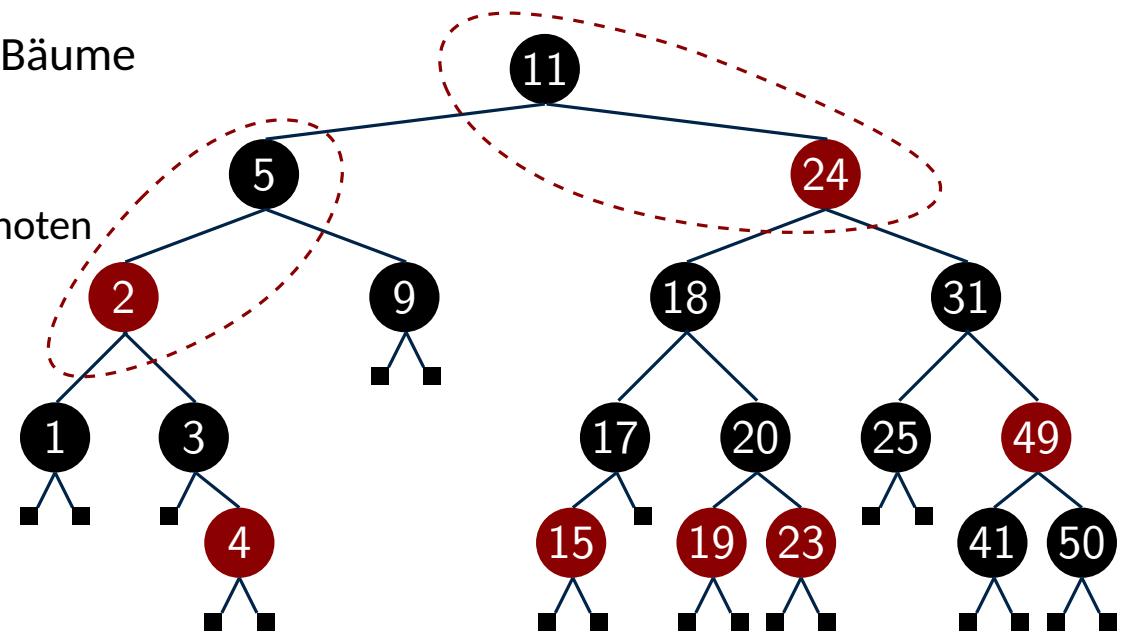
• 11 • 24 •

# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum → (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter ↵ Superknoten



• 11 • 24 •

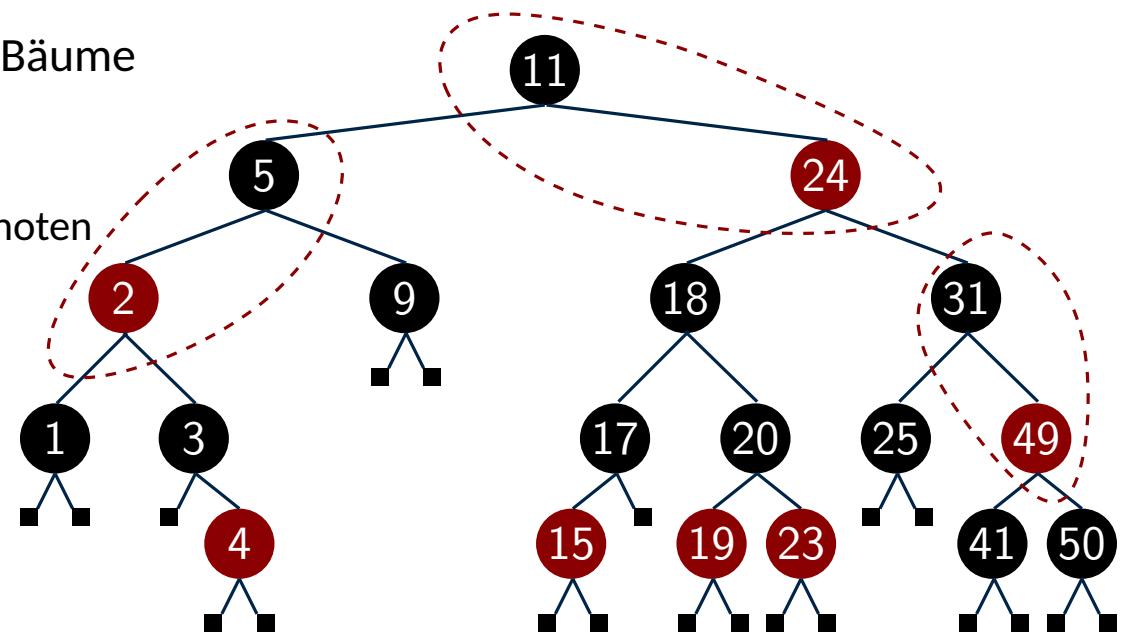
• 2 • 5 •

# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum → (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter ↵ Superknoten



• 11 • 24 •

• 2 • 5 •

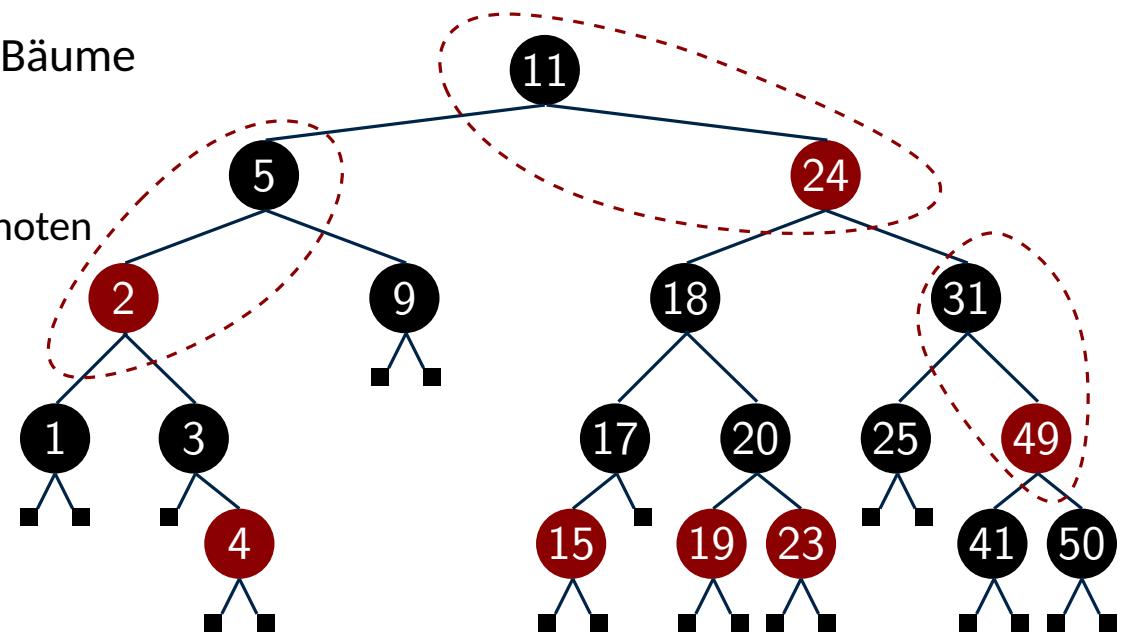
• 31 • 49 •

# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum → (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter ↵ Superknoten



• 11 • 24 •

• 2 • 5 •

• 18 •

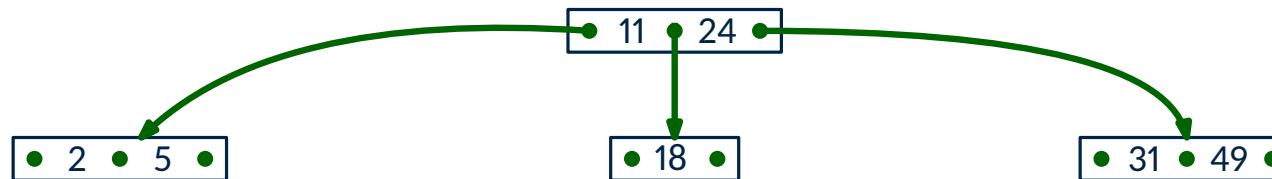
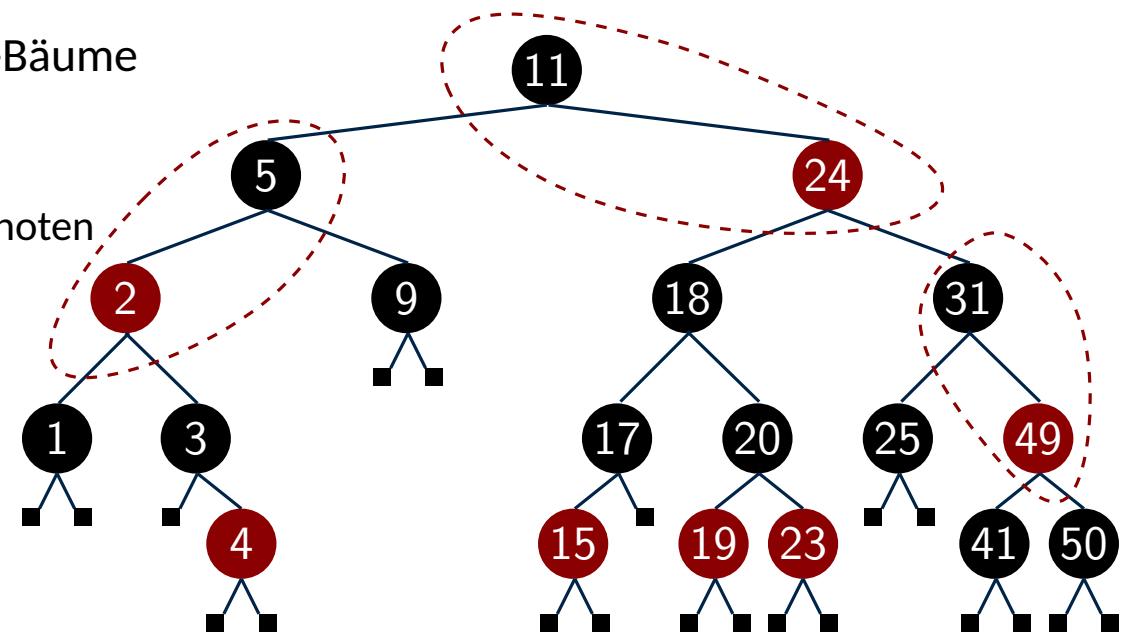
• 31 • 49 •

# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum → (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter ↵ Superknoten

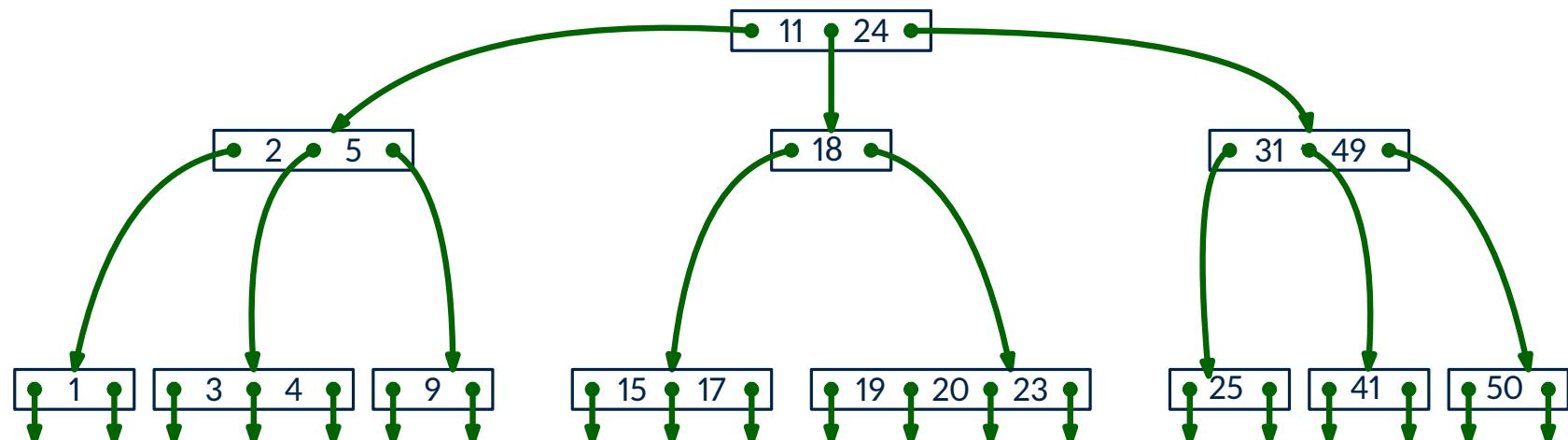
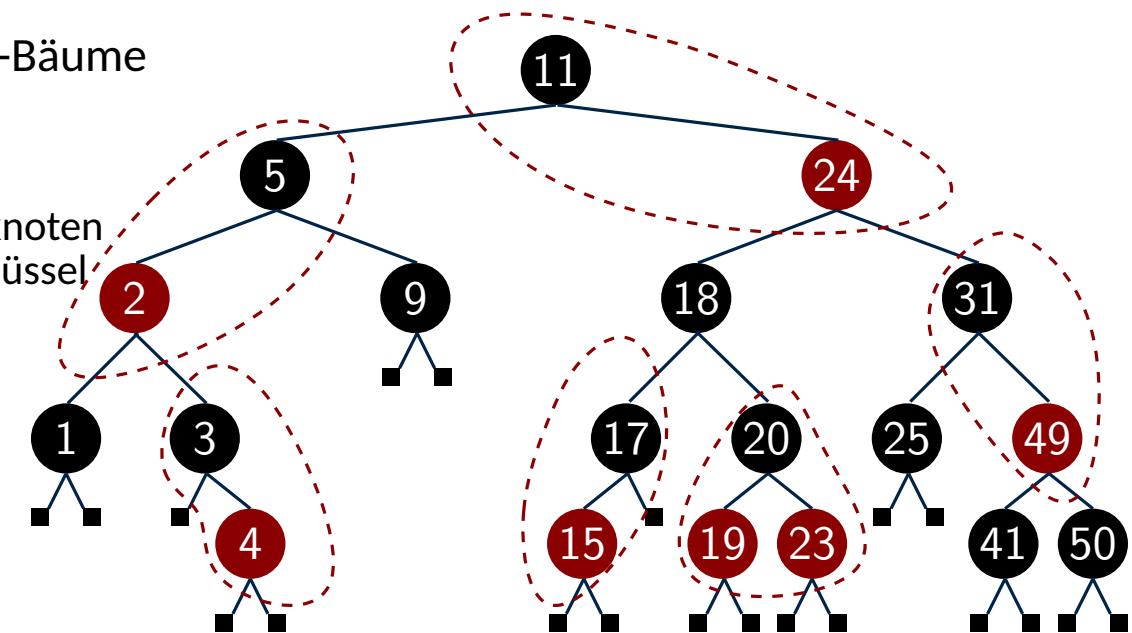


# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

Rot-Schwarz-Baum → (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter ↵ Superknoten  
→ wg. **Rotbedingung**: jeder Superknoten hat 1-3 Schlüssel



# Rot-Schwarz-Bäume vs. (2, 4)-Bäume

Rot-Schwarz-Bäume sind inspiriert durch (2, 4)-Bäume

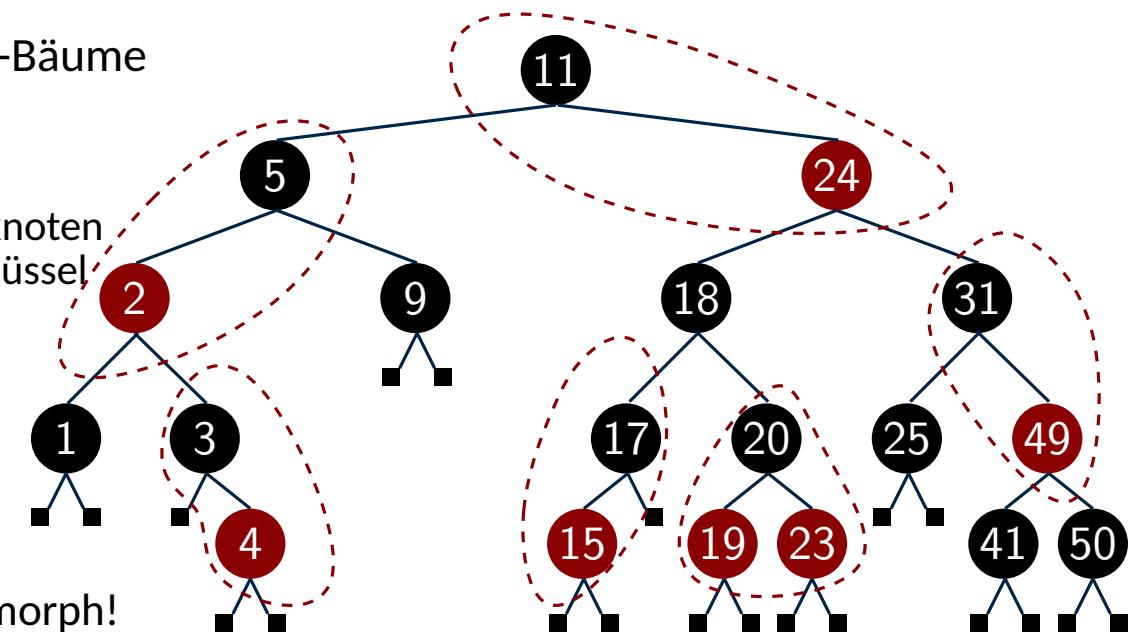
Rot-Schwarz-Baum  $\rightarrow$  (2, 4)-Baum:

- verschmelze jeden **roten** Knoten mit Elter  $\rightsquigarrow$  Superknoten  
 $\rightarrow$  wg. **Rotbedingung**: jeder Superknoten hat 1-3 Schlüssel

(2, 4)-Baum  $\rightarrow$  Rot-Schwarz-Baum?

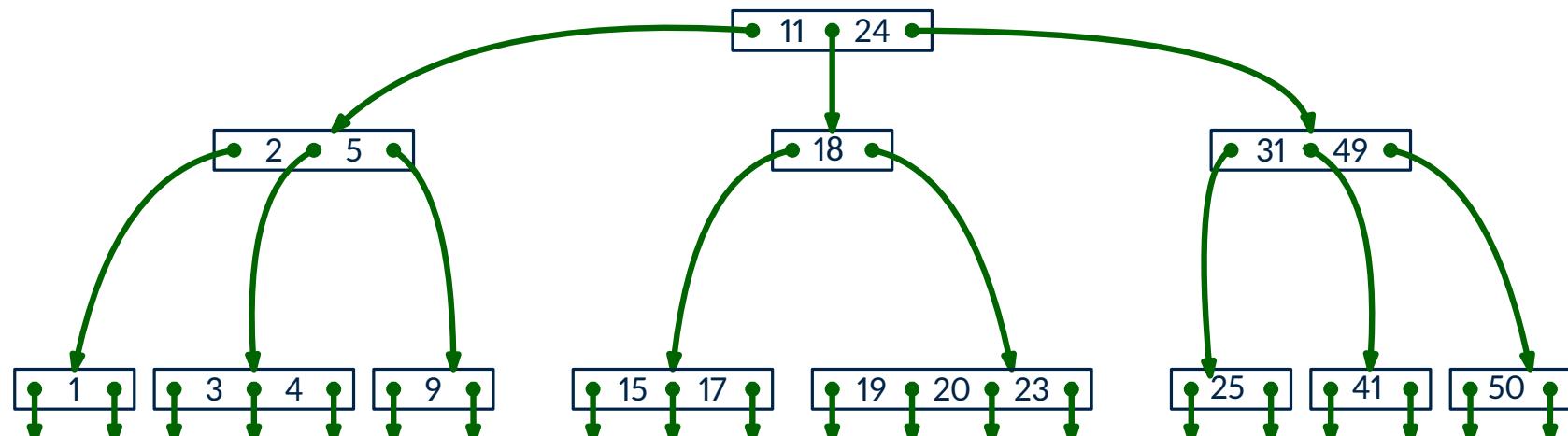
- auch möglich (eindeutig bis auf Symmetrie)

$\Rightarrow$  (2, 4)-Bäume und Rot-Schwarz-Bäume sind isomorph!



## Guter Verständnistest:

- Wie überträgt sich das Vorgehen beim Einfügen/Löschen in Rot-Schwarz-Bäumen auf (2, 4)-Bäume?
- Wie überträgt sich unser Beweis der Balanciertheit von einem zum anderen Baum?



# Zusammenfassung

---

- $(a, b)$ -Bäumen liefern eine elegante Umsetzung für das Wörterbuchproblem
  - in der Praxis insbesondere wegen Cache-Effizienz sehr beliebt
- effiziente Wörterbuchoperationen auf  $(a, b)$ -Bäumen
  - insbesondere auch besonders effiziente Anfrage nach Maximum/Nachfolger/etc.
  - wie auch Rot-Schwarz-Bäume erweiterbar für noch allgemeinere Aufgaben ↗ Übung
- Antwort auf "Wie kommt man auf die Idee von Rot-Schwarz-Bäumen?"
  - **Rot-Schwarz**-Bäume sind isomorph zu  $(2, 4)$ -Bäumen
- amortisierte Analyse der Wörterbuchoperationen
  - wenn  $b \geq 2a$ , amortisiert nur  $O(1)$  Änderungen pro Einfüge-/Löschoperation

Algorithmen und Datenstrukturen SS'23

# Kapitel 11: Hashing

Marvin Künemann

AG Algorithmen & Komplexität

# Bemerkungen zu randomisierten Algorithmen

---

## Erinnerung:

Wenn ein Algorithmus A randomisiert arbeitet,  
(d.h. die RAND C-Funktion verwendet)

analysieren wir seine **erwartete Laufzeit**:

$$T(n) = \max_{x \in I_n} E[t_A(x)]$$

**Intuition:** Wenn wir A häufig aufrufen, wird die durchschnittliche Laufzeit  $\leq T(n)$  sein.

(Häufig kann man sogar zeigen, dass A mit Wahrscheinlichkeit >99% höchstens  $2T(n)$  Berechnungsschritte macht.)

↗ weiterführende Vorlesungen zu randomisierten Algorithmen

## Hinweis zum Erwartungshorizont:

- am Ende der Vorlesung wird **nicht** erwartet:
  - dass Sie erwartete Laufzeiten komplizierter randomisierter Algorithmen analysieren können
    - ↗ Beispiel: Aufgabe 5.4 würde **nicht** in der Klausur gestellt werden
  - dass Sie die Analyse randomisierter Algorithmen aus der Vorlesung reproduzieren können
- allerdings **wird** erwartet:
  - dass Sie die randomisierten Algorithmen aus der Vorlesung und ihre Eigenschaften kennen
  - dass Sie die randomisierten Algorithmen aus der Vorlesung anwenden können
    - ↗ Beispiel:  $n$  Wiederholungen vom randomisierten Quicksort hat erwartete Laufzeit  $O(n^2 \log n)$

# Kapitelüberblick

---

Letzte Kapitel: Suchbaumlösungen für das Wörterbuchproblem

Jetzt: eine weitere Lösung via **Hashing**

Insbesondere behandeln wir in diesem Kapitel:

- Grundideen des Hashing
- Was macht eine gute Hashfunktion aus?
  - $c$ -universelle Hashfamilien: Definition + Beispiele
- Strategien, mit Kollisionen umzugehen
- Anwendungen: Wörterbuchproblem und mehr

# Eine einfache Lösung des Wörterbuchproblems?

Wörterbuch (Dictionary, Assoziatives Array):

Speichert eine Menge  $S$  an Elementen der Form  $(k, v) \in K \times V$  unter folgenden Operationen:

$S.\text{find}(k)$  - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

$S.\text{insert}(e)$  - füge  $e$  in  $S$  ein

$S.\text{remove}(k)$  - lösche das Element mit Schlüssel  $k$  aus  $S$

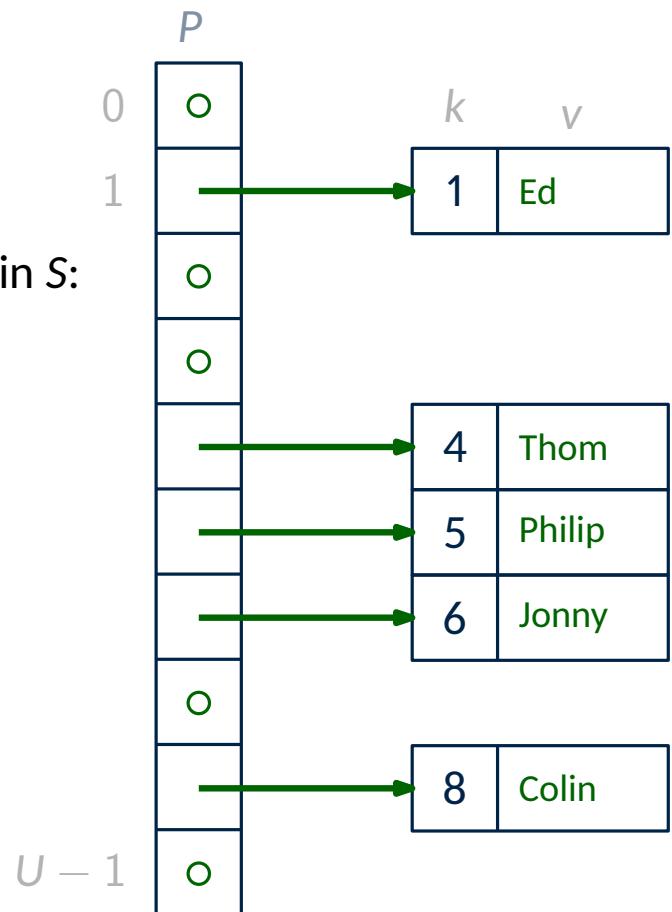
Hierbei beschreibt  $K$  die Schlüsselmenge.

Angenommen  $K = \{0, \dots, U - 1\}$  für ein  $U \in \mathbb{N}$ .

**Einfacher Lösungsansatz: "Direkte Adressierung"**

· verwalten Array  $P[0 \dots U - 1]$  mit Zeigern auf die Elemente in  $S$ :

$P[k]$  ist Zeiger auf Element mit Schlüssel  $k$  in  $S$



# Eine einfache Lösung des Wörterbuchproblems?

Wörterbuch (Dictionary, Assoziatives Array):

Speichert eine Menge  $S$  an Elementen der Form  $(k, v) \in K \times V$  unter folgenden Operationen:

$S.\text{find}(k)$  - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

$S.\text{insert}(e)$  - füge  $e$  in  $S$  ein

$S.\text{remove}(k)$  - lösche das Element mit Schlüssel  $k$  aus  $S$

Hierbei beschreibt  $K$  die Schlüsselmenge.

Angenommen  $K = \{0, \dots, U - 1\}$  für ein  $U \in \mathbb{N}$ .

**Einfacher Lösungsansatz: "Direkte Adressierung"**

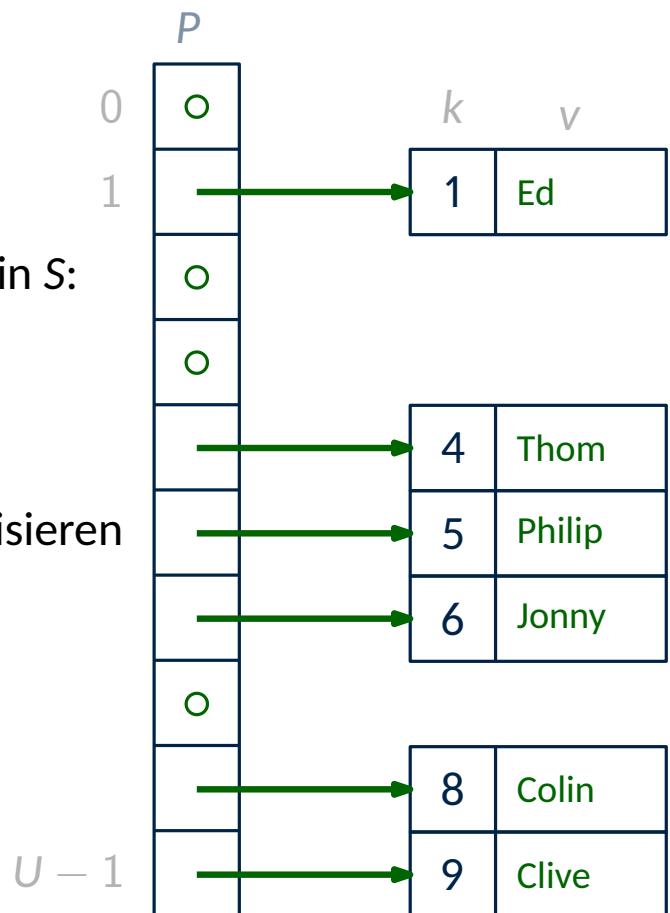
· verwalten Array  $P[0 \dots U - 1]$  mit Zeigern auf die Elemente in  $S$ :

$P[k]$  ist Zeiger auf Element mit Schlüssel  $k$  in  $S$

→ wir können:

· in Zeit  $O(1)$  auf Element mit Schlüssel  $k$  zugreifen

· in Zeit  $O(1)$  Element mit Schlüssel  $k$  hinzufügen/aktualisieren



# Eine einfache Lösung des Wörterbuchproblems?

## Wörterbuch (Dictionary, Assoziatives Array):

Speichert eine Menge  $S$  an Elementen der Form  $(k, v) \in K \times V$  unter folgenden Operationen:

$S.\text{find}(k)$  - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

$S.\text{insert}(e)$  - füge  $e$  in  $S$  ein

$S.\text{remove}(k)$  - lösche das Element mit Schlüssel  $k$  aus  $S$

Hierbei beschreibt  $K$  die Schlüsselmenge.

Angenommen  $K = \{0, \dots, U - 1\}$  für ein  $U \in \mathbb{N}$ .

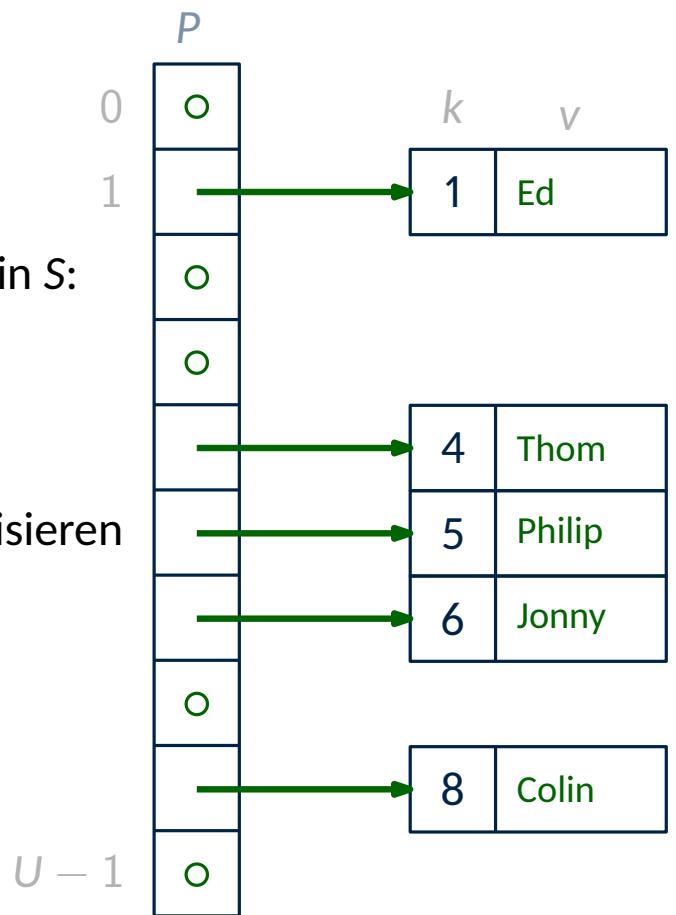
**Einfacher Lösungsansatz: "Direkte Adressierung"**

· verwalten Array  $P[0 \dots U - 1]$  mit Zeigern auf die Elemente in  $S$ :

$P[k]$  ist Zeiger auf Element mit Schlüssel  $k$  in  $S$

→ wir können:

- in Zeit  $O(1)$  auf Element mit Schlüssel  $k$  zugreifen
- in Zeit  $O(1)$  Element mit Schlüssel  $k$  hinzufügen/aktualisieren
- in Zeit  $O(1)$  Element mit Schlüssel  $k$  löschen



# Eine einfache Lösung des Wörterbuchproblems?

## Wörterbuch (Dictionary, Assoziatives Array):

Speichert eine Menge  $S$  an Elementen der Form  $(k, v) \in K \times V$  unter folgenden Operationen:

$S.\text{find}(k)$  - gibt es ein Element  $e \in S$  mit  $\text{key}(e) = k$ ?

$S.\text{insert}(e)$  - füge  $e$  in  $S$  ein

$S.\text{remove}(k)$  - lösche das Element mit Schlüssel  $k$  aus  $S$

Hierbei beschreibt  $K$  die Schlüsselmenge.

Angenommen  $K = \{0, \dots, U - 1\}$  für ein  $U \in \mathbb{N}$ .

**Einfacher Lösungsansatz: "Direkte Adressierung"**

· verwalten Array  $P[0 \dots U - 1]$  mit Zeigern auf die Elemente in  $S$ :

$P[k]$  ist Zeiger auf Element mit Schlüssel  $k$  in  $S$

→ wir können:

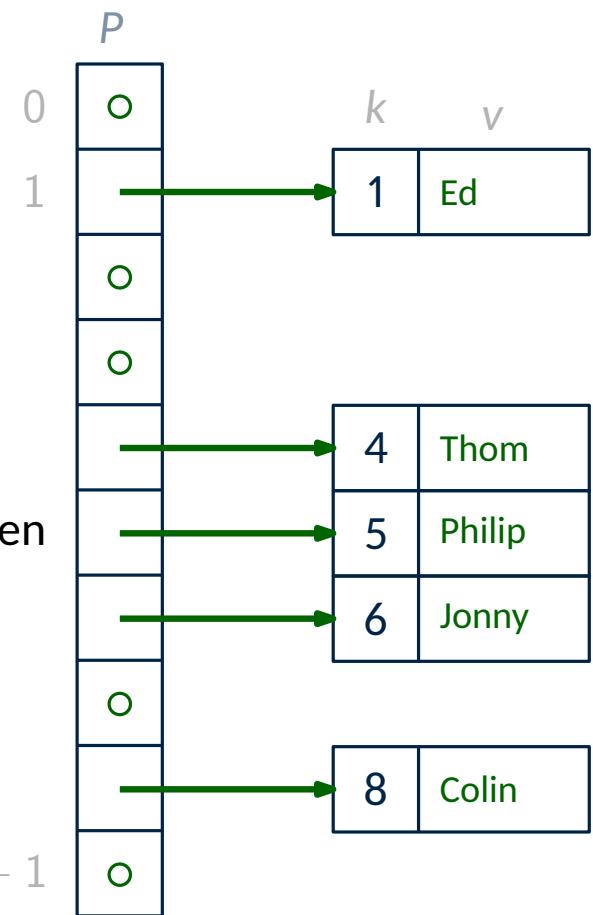
- in Zeit  $O(1)$  auf Element mit Schlüssel  $k$  zugreifen
- in Zeit  $O(1)$  Element mit Schlüssel  $k$  hinzufügen/aktualisieren
- in Zeit  $O(1)$  Element mit Schlüssel  $k$  löschen

**Was ist der Nachteil dieses Ansatzes?**

Speicherbedarf:  $\Theta(U)$

→ in sehr vielen Anwendungen ist  $U$  inpraktikabel groß!

aber: sehr sinnvolle Lösung, wenn  $U$  klein ist (z.B.  $U = O(n)$ )



# Anwendungsszenario: IP-Adressen-Lookup

---

**Szenario:** Internet-Provider, lokaler Router o.Ä. speichert Bezeichnungen für besondere IP-Adressen

IP-Adresse	Bezeichnung
192.168.0.1	Home
192.168.0.5	Media-Server
192.168.1.1	Drucker
...	...

**IPv4-Adresse:** 32-Bit-Adresse      192      168      0      1  
                                  11000000 10101000 00000000 00000001

$$\rightarrow K = \{0, \dots, U - 1\} \text{ mit } U = 2^{32} = 4,294,967,296$$

Schon allein einen 32-Bit-Zeiger für jeden Schlüssel  $k \in \{0, \dots, U - 1\}$  zu speichern, benötigt:

$\approx 17 \text{ GB Speicherplatz}$

$\Rightarrow$  Unnötig ineffizient, wenn wir nur eine kleine Anzahl IP-Adressen speichern wollen

**IPv6-Adresse:** 128-Bit-Adresse

$$\rightarrow K = \{0, \dots, U - 1\} \text{ mit}$$

$$U = 2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456$$

Dieser Ansatz wäre hoffnungslos!

# Magie des Hashing

Sagen wir, wir haben ein Array  $T$  der Größe  $m \ll U$  zur Verfügung

Idee:

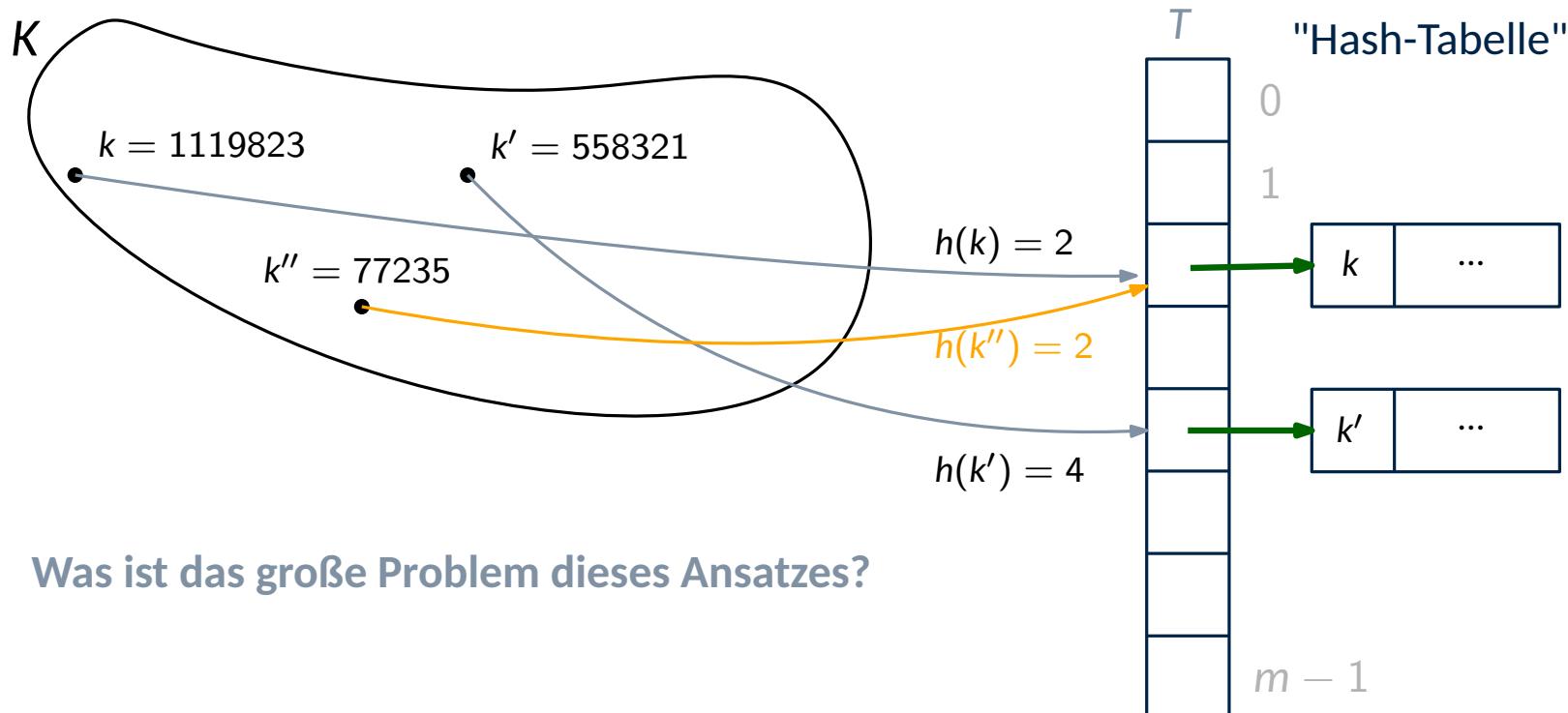
Gegeben  $k$ , "errechnen" wir einen Index  $h(k)$  im Array  $T$

Hashfunktion  $h$ :

$$h : K = \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$$

potentiell riesige Menge!

Hinweis: Dieser Ansatz lässt sich auf andere Schlüsselmengen (wie z.B. Strings) verallgemeinern



Was ist das große Problem dieses Ansatzes?

# Magie des Hashing

Sagen wir, wir haben ein Array  $T$  der Größe  $m \ll U$  zur Verfügung

Idee:

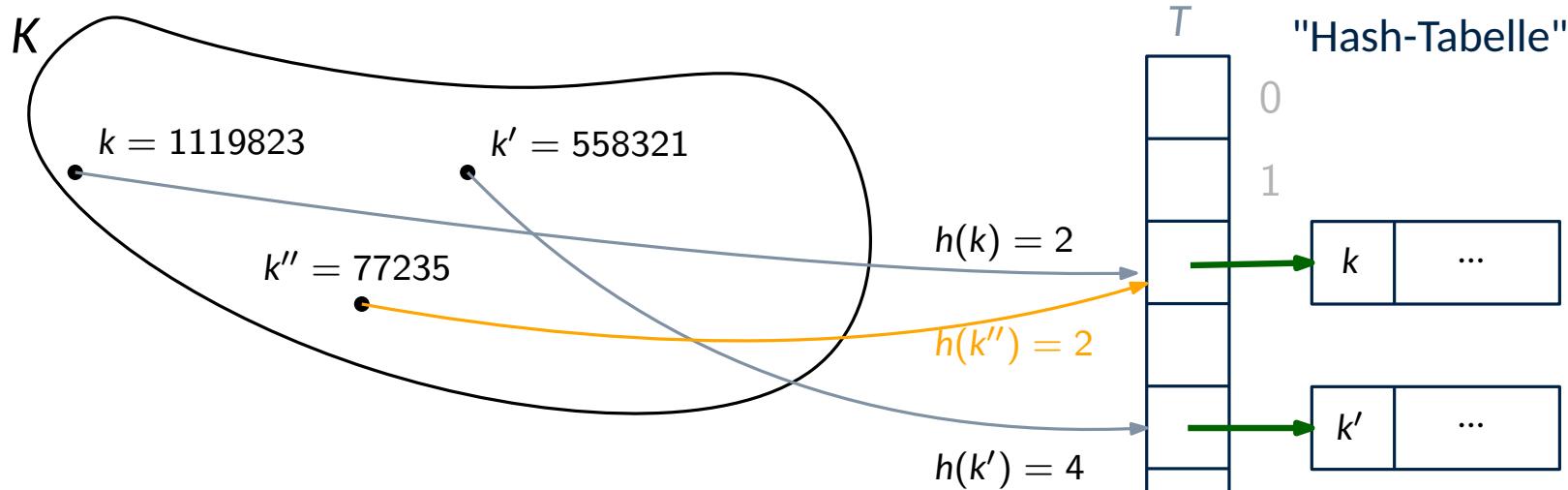
Gegeben  $k$ , "errechnen" wir einen Index  $h(k)$  im Array  $T$

Hashfunktion  $h$ :

$$h : K = \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$$

potentiell riesige Menge!

Hinweis: Dieser Ansatz lässt sich auf andere Schlüsselmengen (wie z.B. Strings) verallgemeinern



Was ist das große Problem dieses Ansatzes?

**Kollisionen:** Für verschiedene Schlüssel  $k, k' \in S$  kann gelten:

$$h(k) = h(k')$$

# Magie des Hashing

Sagen wir, wir haben ein Array  $T$  der Größe  $m \ll U$  zur Verfügung

Idee:

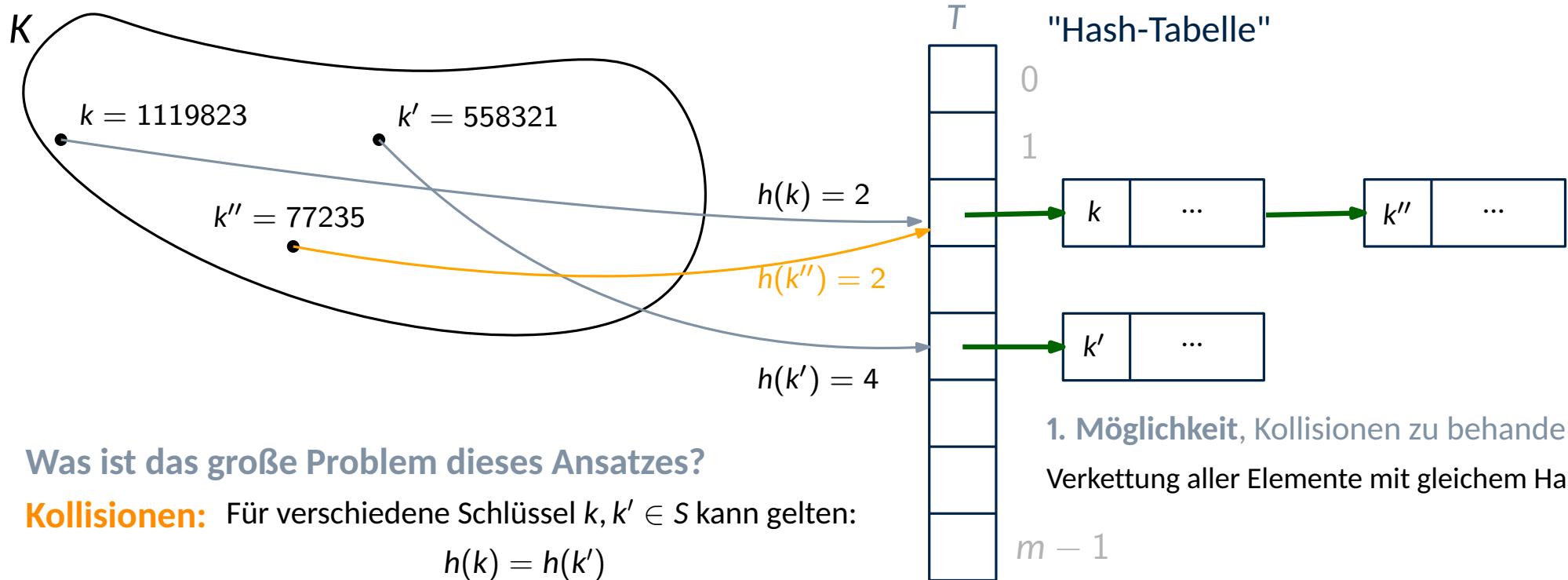
Gegeben  $k$ , "errechnen" wir einen Index  $h(k)$  im Array  $T$

Hashfunktion  $h$ :

$$h : K = \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$$

potentiell riesige Menge!

Hinweis: Dieser Ansatz lässt sich auf andere Schlüsselmengen (wie z.B. Strings) verallgemeinern



Was ist das große Problem dieses Ansatzes?

**Kollisionen:** Für verschiedene Schlüssel  $k, k' \in S$  kann gelten:

$$h(k) = h(k')$$

1. Möglichkeit, Kollisionen zu behandeln:  
Verkettung aller Elemente mit gleichem Hash

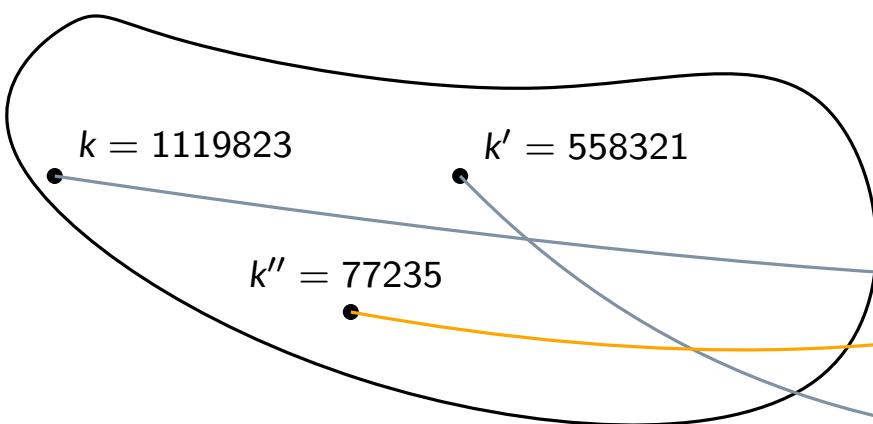
# Hashing mit Verkettung

**Ansatz:** Für jede Position  $j$  der Hashtabelle speichern wir eine verkettete Liste  $L_j$

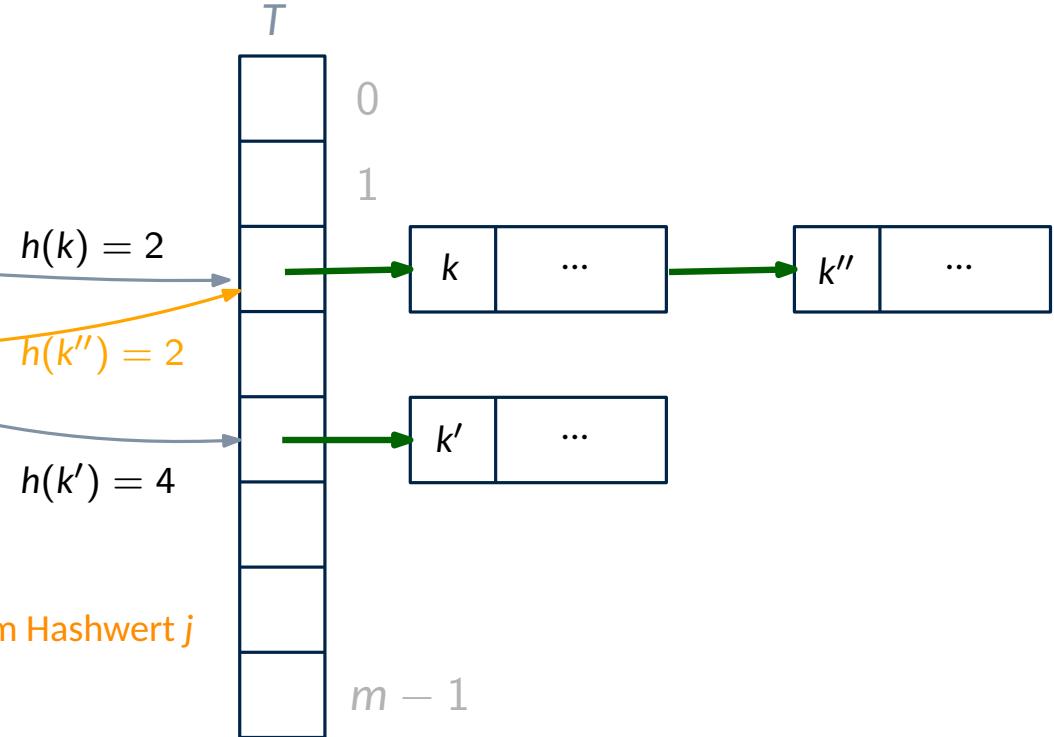
→  $L_j$  speichert alle Elemente  $e \in S$  mit  $h(\text{key}(e)) = j$

**Ausgangspunkt:** Wir haben eine "gute" Hashfunktion  $h$ , die wir in Zeit  $O(1)$  auswerten können

- $\text{find}(k)$ : durchsuche  $L_{h(k)}$  nach Element mit Schlüssel  $k$   $O(1 + |L_{h(k)}|)$
- $\text{insert}(k, v)$ : durchsuche  $L_{h(k)}$  nach Element mit Schlüssel  $k$   
wenn es existiert, aktualisiere das Element, sonst füge  $(k, v)$  zu  $L_{h(k)}$  hinzu.  $O(1 + |L_{h(k)}|)$
- $\text{remove}(k)$ : durchsuche  $L_{h(k)}$  nach Element mit Schlüssel  $k$  und entferne es aus  $L_{h(k)}$   $O(1 + |L_{h(k)}|)$



**Laufzeit?**  $O(1 + |L_{h(k)}|)$  für alle Operationen



**Definition.**

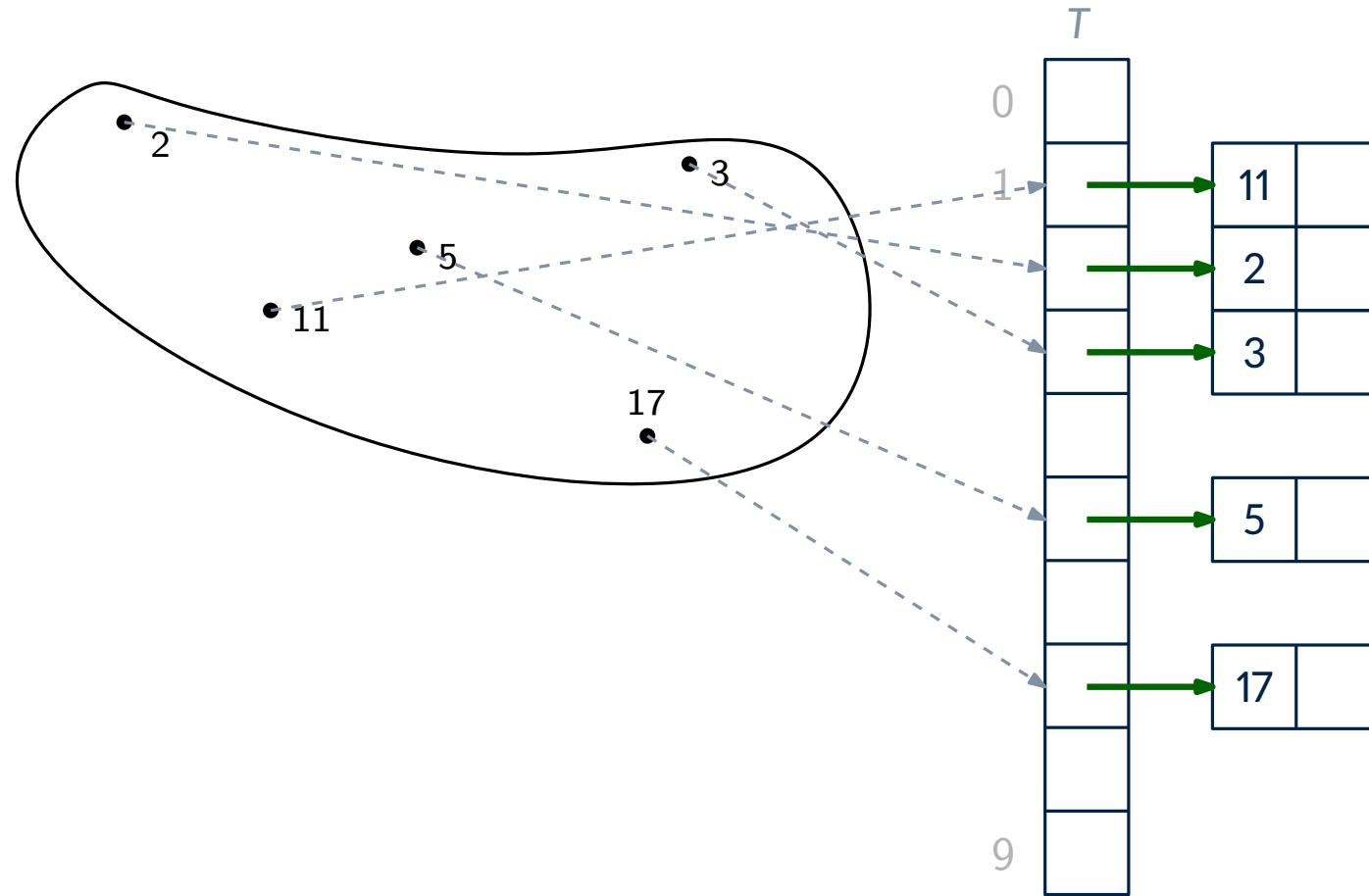
$C_j := |\{(k, v) \in S \mid h(k) = j\}|$  sei die Anzahl Kollisionen am Hashwert  $j$

Dann ist die Laufzeit in  $O(1 + C_{h(k)})$

# Beispiel

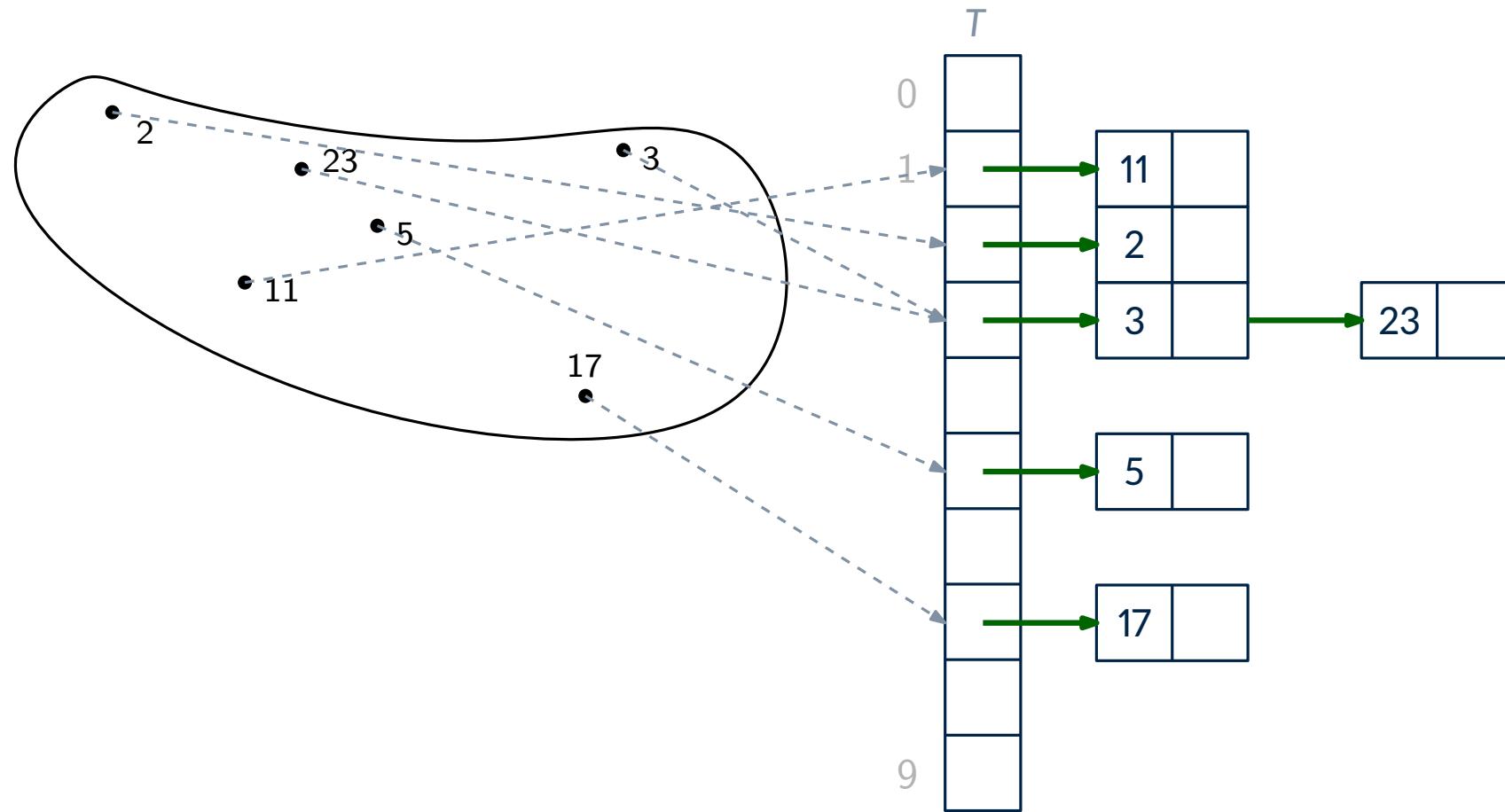
---

Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"



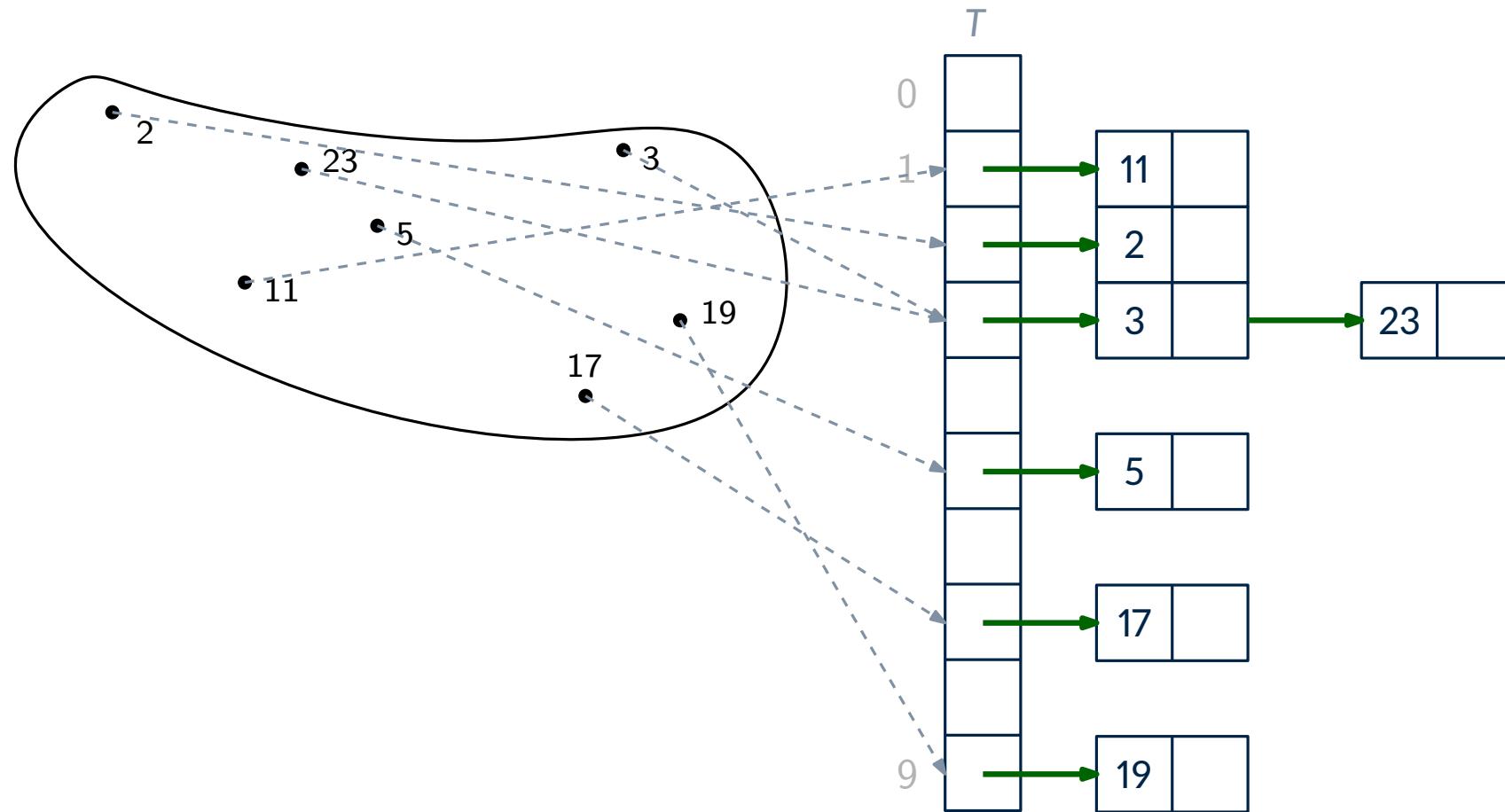
# Beispiel

Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"



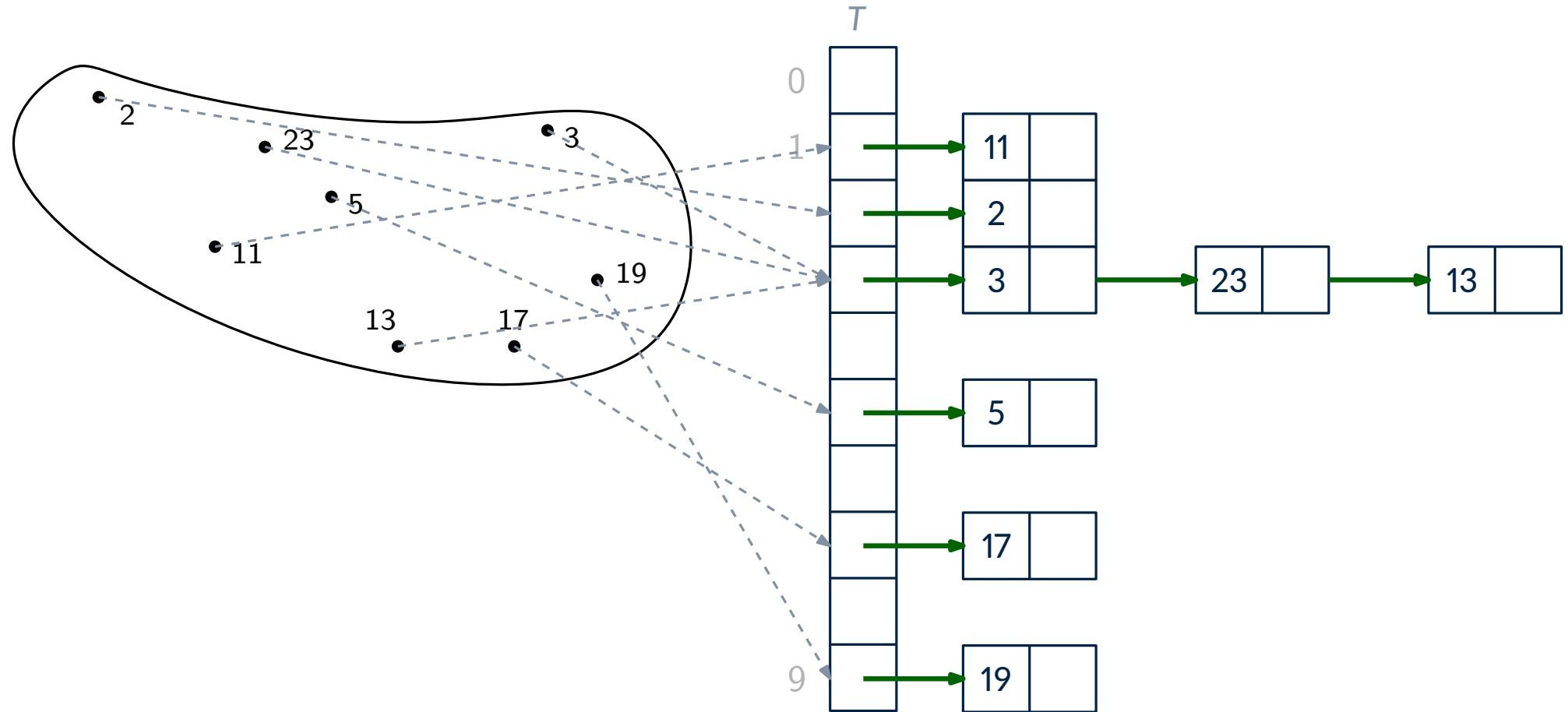
# Beispiel

Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"



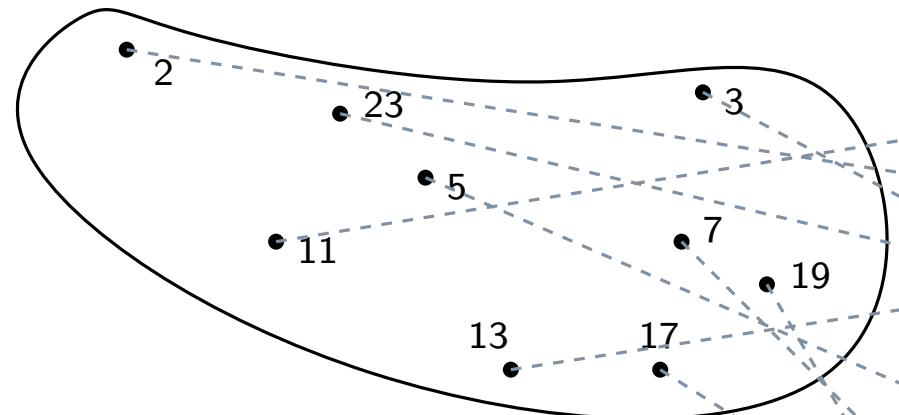
# Beispiel

Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

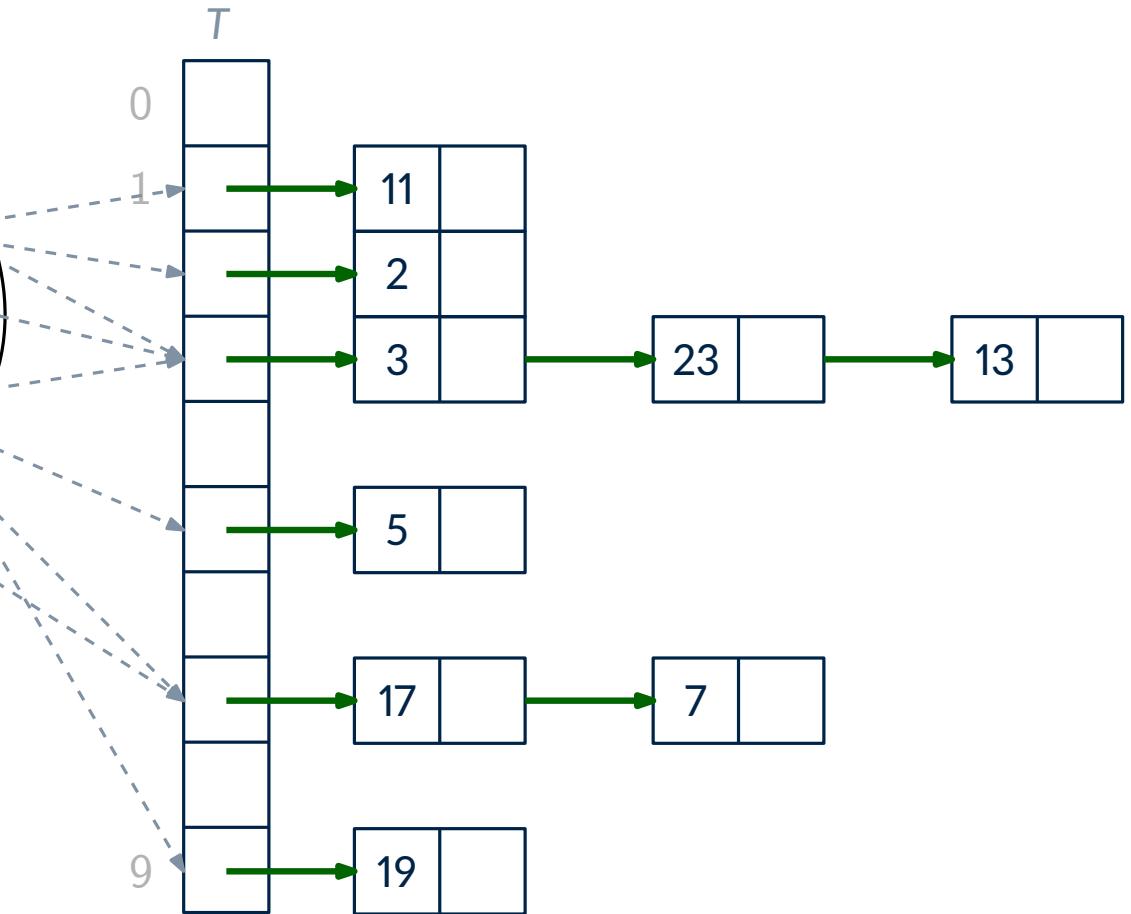


# Beispiel

Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

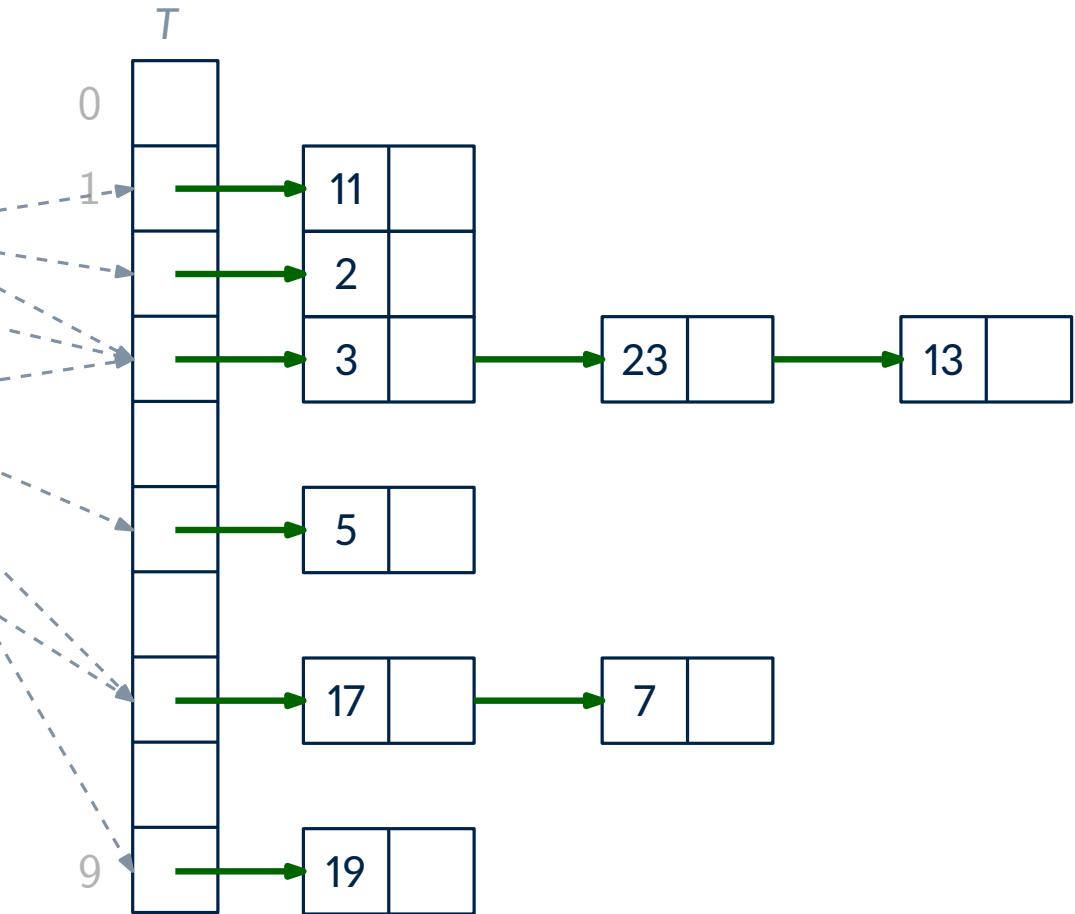
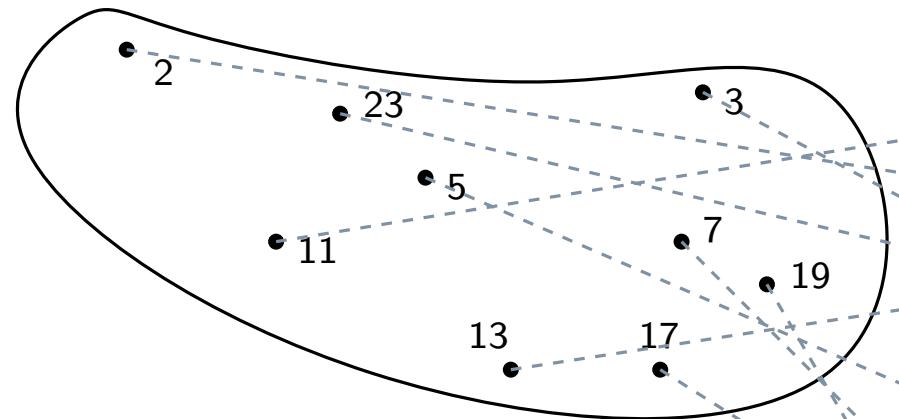


Ist  $h$  eine gute Hashfunktion?



# Beispiel

Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"



Ist  $h$  eine gute Hashfunktion?

Nicht unbedingt!

**Beispiel:**  $\text{key}(S) = \{10, 20, \dots, 10n\}$   
 $\rightarrow h(k) = 0$  für alle  $(k, v) \in S$

z.B. wenn  $h(k) = h(k')$  für alle  $k, k'$  haben **find**, **insert**, **remove** Worst-Case-Laufzeit  $\Theta(n)$

# Was macht eine gute Hash-Funktion aus?

---

Betrachten weiterhin Schlüsselmenge  $K = \{0, \dots, U - 1\}$ .

Für eine Hashing-Lösung des Wörterbuchproblems haben wir u.a. die folgenden Anforderungen:

$$h : \quad K = \{0, \dots, U - 1\} \quad \rightarrow \quad \{0, \dots, m - 1\}$$

**1.  $h$  kann schnell ausgewertet werden**

**2.  $h$  erzeugt möglichst wenig Kollisionen**

**Behauptung:** Für jedes solche  $h$  und  $S$  mit  $|S| = n$  gibt es einen Hashwert  $j$  mit  $\geq \frac{n}{m}$  Kollisionen.

**Frage:** Warum?

**Ziel:** Ein Hashwert  $h$  hat erwartet höchstens  $O(1 + \frac{n}{m})$  Kollisionen

# 1. Versuch: zufällige Funktion

---

Idee: Wir nehmen eine **uniform zufällige** Funktion  $h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$

$k$	$h(k)$
0	RAND(0, $m - 1$ )
1	RAND(0, $m - 1$ )
2	RAND(0, $m - 1$ )
3	RAND(0, $m - 1$ )
:	:
$U - 1$	RAND(0, $m - 1$ )

# 1. Versuch: zufällige Funktion

---

Idee: Wir nehmen eine **uniform zufällige** Funktion  $h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$

$k$	$h(k)$
0	3
1	4
2	3
3	1
:	:
$U - 1$	3

# 1. Versuch: zufällige Funktion

---

Idee: Wir nehmen eine **uniform zufällige** Funktion  $h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$

Das heißt für jedes  $k \in \{0, \dots, U - 1\}$  und  $j \in \{0, \dots, m - 1\}$  gilt:  $\Pr[h(k) = j] = \frac{1}{m}$   
 $\rightarrow h(k)$  ist uniform zufällig aus  $\{0, \dots, m - 1\}$  gewählt

$k$	$h(k)$
0	3
1	4
2	3
3	1
:	:
$U - 1$	3

# 1. Versuch: zufällige Funktion

**Idee:** Wir nehmen eine **uniform zufällige** Funktion  $h : \{0, \dots, U-1\} \rightarrow \{0, \dots, m-1\}$

Das heißt für jedes  $k \in \{0, \dots, U-1\}$  und  $j \in \{0, \dots, m-1\}$  gilt:  $\Pr[h(k) = j] = \frac{1}{m}$   
 $\rightarrow h(k)$  ist uniform zufällig aus  $\{0, \dots, m-1\}$  gewählt

## Theorem.

Wenn wir eine uniform zufällige Hashfunktion  $h : \{0, \dots, U-1\} \rightarrow \{0, \dots, m-1\}$  wählen, gilt:

Die erwartete Laufzeit von `find`, `insert` und `remove` für Hashing mit Verkettung ist  $O(1 + \frac{n}{m})$ .

**Beweis:** Wir wissen, dass `find(k)`, `insert(k, ·)`, `remove(k)` in Zeit  $O(1 + C_{h(k)})$  laufen, wobei

$$C_j = |\{(k', v) \in S \mid h(k') = j\}|$$

Definiere, für beliebiges  $0 \leq k' < U$ ,  $0 \leq j < m$  die **Indikatorvariable**

$$X_{k',j} = \begin{cases} 1 & \text{wenn } h(k') = j, \\ 0 & \text{ansonsten} \end{cases}$$

Für beliebiges  $j$  gilt:  $C_j = \sum_{(k',v) \in S} X_{k',j}$

Also gilt:

$$\begin{aligned} E[C_{h(k)}] &= \sum_{(k',v) \in S} E[X_{k',h(k)}] = \sum_{(k',v) \in S} \Pr[X_{k',h(k)} = 1] \\ &= \sum_{(k',v) \in S} \Pr[h(k') = h(k)] \\ &\leq 1 + |S| \cdot \frac{1}{m} = 1 + \frac{n}{m} \quad (\text{Fall 1 tritt höchstens einmal auf}) \end{aligned}$$

**Fall 1:**  $k' = k$

$$\Pr[h(k') = h(k)] = \Pr[h(k) = h(k)] = 1$$

**Fall 2:**  $k' \neq k$

$h(k')$  ist unabhängig von  $h(k)$  verteilt mit  
 $\Pr[h(k') = x] = \frac{1}{m}$  für  $x \in \{0, \dots, m-1\}$

$$\text{Daher: } \Pr[h(k') = h(k)] = \frac{1}{m}$$



# Problem mit 1. Versuch

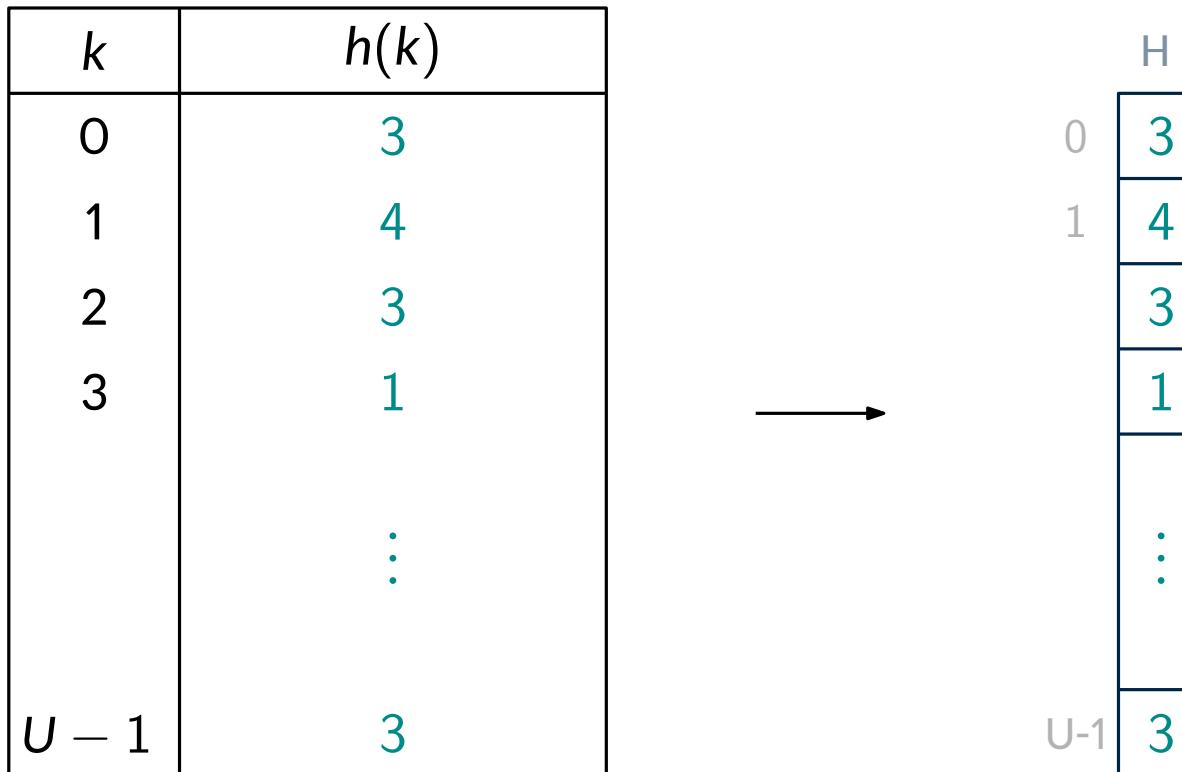
Wir haben bewiesen:

Mit **uniform zufälliger** Hashfunktion  $h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$  erzeugen wir wenige Kollisionen  
Genauer: alle drei Operationen laufen in erwarteter Zeit  $O(1 + \frac{n}{m})$ .

Wir haben dabei implizit angenommen, dass wir  $h$  in Zeit  $O(1)$  auswerten können.

Können wir das überhaupt?

Ja, wir können ein Array  $H$  erstellen, sodass  $H[k]$  der Hashwert von  $k$  ist.



# Problem mit 1. Versuch

---

Wir haben bewiesen:

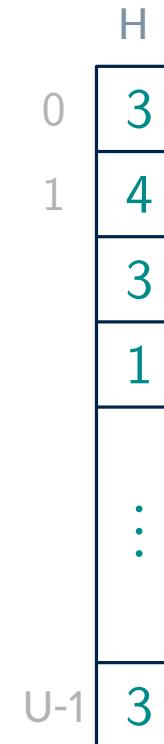
Mit **uniform zufälliger** Hashfunktion  $h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$  erzeugen wir wenige Kollisionen  
Genauer: alle drei Operationen laufen in erwarteter Zeit  $O(1 + \frac{n}{m})$ .

Wir haben dabei implizit angenommen, dass wir  $h$  in Zeit  $O(1)$  auswerten können.

**Können wir das überhaupt?**

Ja, wir können ein Array  $H$  erstellen, sodass  $H[k]$  der Hashwert von  $k$  ist.

**Was ist der große Nachteil des Ansatzes?** Speicherbedarf  $\Omega(U)$



# Problem mit 1. Versuch

---

Wir haben bewiesen:

Mit **uniform zufälliger** Hashfunktion  $h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$  erzeugen wir wenige Kollisionen  
Genauer: alle drei Operationen laufen in erwarteter Zeit  $O(1 + \frac{n}{m})$ .

Wir haben dabei implizit angenommen, dass wir  $h$  in Zeit  $O(1)$  auswerten können.

**Können wir das überhaupt?**

Ja, wir können ein Array  $H$  erstellen, sodass  $H[k]$  der Hashwert von  $k$  ist.

**Was ist der große Nachteil des Ansatzes? Speicherbedarf  $\Omega(U)$**

Im Allgemeinen gilt: wir benötigen erwartet  $\Omega(U \log m)$  Bits, um ein uniform zufälliges  $h$  zu speichern  
→ dieser Ansatz hat größeren Speicherbedarf als direkte Adressierung!

Gibt es gute Hashfunktionen, die geringen Speicherbedarf haben?

# c-universelle Hashfamilie

---

Die folgende Definition abstrahiert, was für eine Eigenschaft wir im Beweis des letzten Theorems benötigt haben.

**Definition**      Synonym für Menge!

$\mathcal{H}$  sei eine Familie von Hashfunktionen der Form  $\{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$ .

$h$  sei eine Funktion die uniform zufällig aus  $\mathcal{H}$  gezogen wird.

Wir sagen  $\mathcal{H}$  ist **c-universell**, wenn für alle  $k, k' \in \{0, \dots, U - 1\}$  mit  $k \neq k'$  gilt, dass:

$$\Pr_h[h(k) = h(k')] \leq \frac{c}{m}$$

Äquivalent:  $|\{h \in \mathcal{H} \mid h(k) = h(k')\}| \leq \frac{c}{m} \cdot |\mathcal{H}|$

**Bedeutung:** Zwei gegebene Schlüssel kollidieren unter  $h$  aus  $\mathcal{H}$  mit geringer Wahrscheinlichkeit ( $\leq \frac{c}{m}$ )

**Frage:**  $\mathcal{H}$  sei die Menge aller Funktionen  $\{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}$ .

Ist  $\mathcal{H}$  c-universell für eine Konstante  $c$ ?

# Hashing mit $c$ -universeller Hashfamilie

$\mathcal{H}$  sei  $c$ -universelle Hashfamilie

für alle  $k \neq k'$  gilt:  $\Pr_h[h(k) = h(k')] \leq \frac{c}{m}$

## Theorem.

Wenn wir eine uniform zufällige Hashfunktion  $h$  aus  $c$ -universeller Hashfamilie  $\mathcal{H}$  wählen, gilt:

Die erwartete Laufzeit von `find`, `insert` und `remove` für Hashing mit Verkettung ist  $O(1 + \frac{cn}{m})$ .

**Beweis:** Wir wissen, dass `find`( $k$ ), `insert`( $k, \cdot$ ), `remove`( $k$ ) in Zeit  $O(1 + C_{h(k)})$  laufen, wobei

$$C_j = |\{(k', v) \in S \mid h(k') = j\}|$$

Definiere, für beliebiges  $0 \leq k' < U$ ,  $0 \leq j < m$  die **Indikatorvariable**

$$X_{k',j} = \begin{cases} 1 & \text{wenn } h(k') = j, \\ 0 & \text{ansonsten} \end{cases}$$

Für beliebiges  $j$  gilt:  $C_j = \sum_{(k',v) \in S} X_{k',j}$

Also gilt:

$$\begin{aligned} E[C_{h(k)}] &= \sum_{(k',v) \in S} E[X_{k',h(k)}] = \sum_{(k',v) \in S} \Pr[X_{k',h(k)} = 1] \\ &= \sum_{(k',v) \in S} \Pr[h(k') = h(k)] \\ &\leq 1 + |S| \cdot \frac{c}{m} = 1 + \frac{cn}{m} \quad (\text{Fall 1 tritt höchstens einmal auf}) \end{aligned}$$

**Fall 1:**  $k' = k$

$$\Pr[h(k') = h(k)] = \Pr[h(k) = h(k)] = 1$$

**Fall 2:**  $k' \neq k$

$$\mathcal{H} \text{ ist } c\text{-universell: } \Pr[h(k') = h(k)] \leq \frac{c}{m}$$



# Eine 1-universelle Hashfamilie

Wir konstruieren eine 1-universelle Hashfamilie  $\mathcal{H}$  mit Funktionen

$$h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}.$$

**Annahme:**  $m$  ist Primzahl

**Ansatz:**

wir zerteilen einen Schlüssel in "Stücke" (jeweils aus  $\{0, \dots, m - 1\}$ )

Sei  $b = \lfloor \log_2 m \rfloor$ . Dann ist jede  $b$ -Bit-Zahl in  $\{0, \dots, m - 1\}$ .

$$\rightarrow t = \lceil \frac{\log_2 U}{b} \rceil \text{ Stücke}$$

Jede Zahl  $x \in \{0, \dots, U - 1\}$  lässt sich als

Folge von  $b$ -Bit-Zahlen  $(x_0, \dots, x_{t-1})$  schreiben:

$$x = \sum_{i=0}^{t-1} x_i 2^{b \cdot i}$$

Für eine Zahl  $x$  bezeichne  $\mathbf{x} = (x_0, \dots, x_{t-1})$  die Folge der Stücke.

**Definition**

Für jede Wahl von  $\mathbf{a} = (a_0, \dots, a_{t-1}) \in \{0, \dots, m - 1\}^t$ , definieren wir

$$\begin{aligned} h_{\mathbf{a}}(k) &= \mathbf{a} \cdot \mathbf{k} \bmod m \\ &= (\sum_{i=0}^{t-1} a_i \cdot k_i) \bmod m \end{aligned}$$

**Theorem.**

Die Hashfamilie  $\mathcal{H}_{SP} = \{h_{\mathbf{a}} \mid \mathbf{a} \in \{0, \dots, m - 1\}^t\}$  ist **1-universell**.

**Beweis:** ↗ Tafelpräsentation

**Beispiel:**

$$U = 1024$$

$$m = 13$$

$$\rightarrow b = 3$$

$$\rightarrow t = \lceil \frac{10}{3} \rceil = 4$$

$$\text{z.B. } x = 696$$

$$\begin{aligned} &= \underline{00} \underline{10} \underline{10} \underline{11} \underline{000}_2 \\ &= \quad 1 \quad 2 \quad 7 \quad 0_{2^3} \end{aligned}$$

$$\rightarrow \mathbf{x} = (0, 7, 2, 1)$$

Für  $\mathbf{a} = (5, 2, 0, 3)$  ist

$$h_{\mathbf{a}}(696)$$

$$= (5, 2, 0, 3) \cdot (0, 7, 2, 1) \bmod 13$$

$$= (0 + 14 + 0 + 3) \bmod 13$$

$$= 4$$

# Nur für Primzahlen - eine Einschränkung?

---

Die 1-Universalität von  $\mathcal{H}_{SP}$  fußte darauf, dass  $m$  als Primzahl gewählt wurde.  
Ist das eine starke Einschränkung?

**Fakt 1** Für jede natürliche Zahl  $x$  findet sich eine Primzahl im Intervall  $\{x, x + 1, \dots, 2x\}$

→ Für jede gewünschte Hashtabellengröße  $m$  kann man eine Primzahl  $m \leq p \leq 2m$  finden.  
~~ 1-universelle Hashfamilie für eine Hashtabelle der Größe  $\Theta(m)$ .

In der Praxis könnte man eine Tabelle möglicher Primzahlen bereithalten.

Mit etwas mehr zahlentheoretischen Hilfsmitteln kann man sogar folgendes zeigen:

## Lemma

Für beliebiges  $m \in \mathbb{N}$  lässt sich eine Primzahl  $m \leq p \leq 2m$  in Zeit  $o(m)$  finden.

→ Schneller als  $\Theta(m)$ -Zeit-Initialisierung der Hash-Tabelle

**Beweisskizze\***: · wähle  $k$  so, dass  $(k - 1)^3 \leq m \leq k^3$

- **Zahlentheorie:** Es gibt eine Primzahl in  $I = \{k^3, k^3 + 1, \dots, (k + 1)^3\}$
- suche nach Primzahlen nach dem Prinzip des **Sieb des Eratosthenes**
  - gehe durch alle  $2 \leq d \leq (k + 1)^{3/2}$  und streiche alle Vielfachen von  $d$  aus  $I$
  - alle verbleibenden Zahlen in  $I$  sind nun Primzahlen;  $I$  ist nicht leer
- eine genaue Analyse ergibt eine Laufzeit von  $O(k^2 \log n) = o(m)$ . □

# Zwischenstand

---

Angenommen, wir wollen ein Wörterbuch mit  $K = \{0, \dots, U - 1\}$  und Kapazität  $C$  implementieren.

Hashing mit Verkettung mit  $\mathcal{H}_{SP}$  mit Hashtabellengröße  $m = \Theta(C)$  erreicht folgende Laufzeit:

## Beschränktes Wörterbuch (Dictionary, Assoziatives Array):

Speichert eine Menge  $S$  von  $\leq C$  Elementen der Form  $(k, v) \in \{0, \dots, U - 1\} \times V$  unter folgenden Operationen:

- |                          |   |
|--------------------------|---|
| <code>S.init(C)</code>   | - initialisiert die Datenstruktur mit Kapazität $C$       |
| <code>S.find(k)</code>   | - gibt es ein Element $e \in S$ mit $\text{key}(e) = k$ ? |
| <code>S.insert(e)</code> | - füge $e$ in $S$ ein                                     |
| <code>S.remove(k)</code> | - lösche das Element mit Schlüssel $k$ aus $S$            |

`S.init(C)` läuft in Zeit  $O(C)$ , alle anderen Operationen haben erwartete Laufzeit  $O(1)$ .

## Erweiterungen:

- (unbeschränktes) Wörterbuch:

wir können `find`, `insert`, `remove` mit amortisierter erwarteter Laufzeit  $O(1)$  implementieren. ↗ Übung

- andere Schlüsselmengen, z.B. Strings

$\mathcal{H}_{SP}$  lässt sich für die Nutzung auf Strings anpassen. ↗ Übung

# Anwendungen des Hashings

---

Universelle Hashfamilien haben zahlreiche Anwendungen neben dem Wörterbuchproblem

Schneller randomisierter Vergleich großer Objekte

**Beispiel:** Sind zwei gegebene Arrays  $A_1[1 \dots n], A_2[1 \dots n]$  gleich?

**Ansatz:** Angenommen wir haben eine 1-universelle Hashfamilie  $\mathcal{H}$  mit Funktionen der Form:

$$h : \text{Arrays der Größe } n \rightarrow \{0, \dots, m-1\} \quad \text{mit } m \geq n$$

und kennen bereits  $h(A_1)$  und  $h(A_2)$

Wenn  $h(A_1) \neq h(A_2)$ , dann wissen wir, dass  $A_1$  und  $A_2$  verschieden sind.

Wenn  $h(A_1) = h(A_2)$ , dann könnten  $A_1$  und  $A_2$  trotzdem verschieden sein:

→ in diesem Fall iterieren wir über  $i = 1, \dots, n$  und testen, ob  $A_1[i] = A_2[i]$  immer gilt

**Erwartete Laufzeit dieses Tests:**

- wenn  $A_1 = A_2$ , dann landen wir immer im zweiten Fall. → Laufzeit  $\Theta(n)$
- wenn  $A_1 \neq A_2$ , dann landen wir immer im zweiten Fall nur mit Wahrscheinlichkeit  $\leq \frac{1}{m}$ .  
⇒ erwartete Laufzeit ist  $O(1) + \frac{1}{m} \cdot O(n) = O(1 + \frac{n}{m}) = O(1)$ .

⇒ Mit bekannten Hashes von zwei unterschiedlichen Arrays  $A_1 \neq A_2$  können wir ihre Ungleichheit in erwarteter Zeit  $O(1)$  feststellen!  
(Vgl: md5sum)

**Hinweis:** Hashing wird auch ausgiebig für **kryptografische Anwendungen** benutzt.

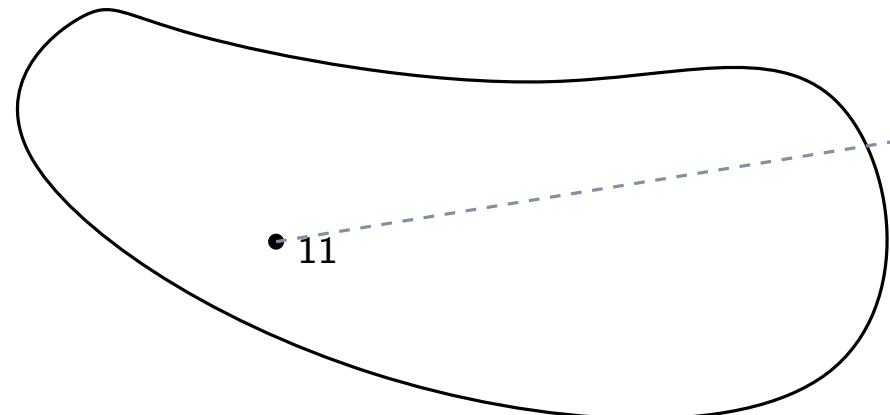
Für diese Anwendungen werden meist noch stärkere Eigenschaften benötigt.

z.B.: Gegeben  $k$  soll es nicht einfach sein, einen Schlüssel  $k' \neq k$  zu finden, sodass

$$h(k) = h(k')$$

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"



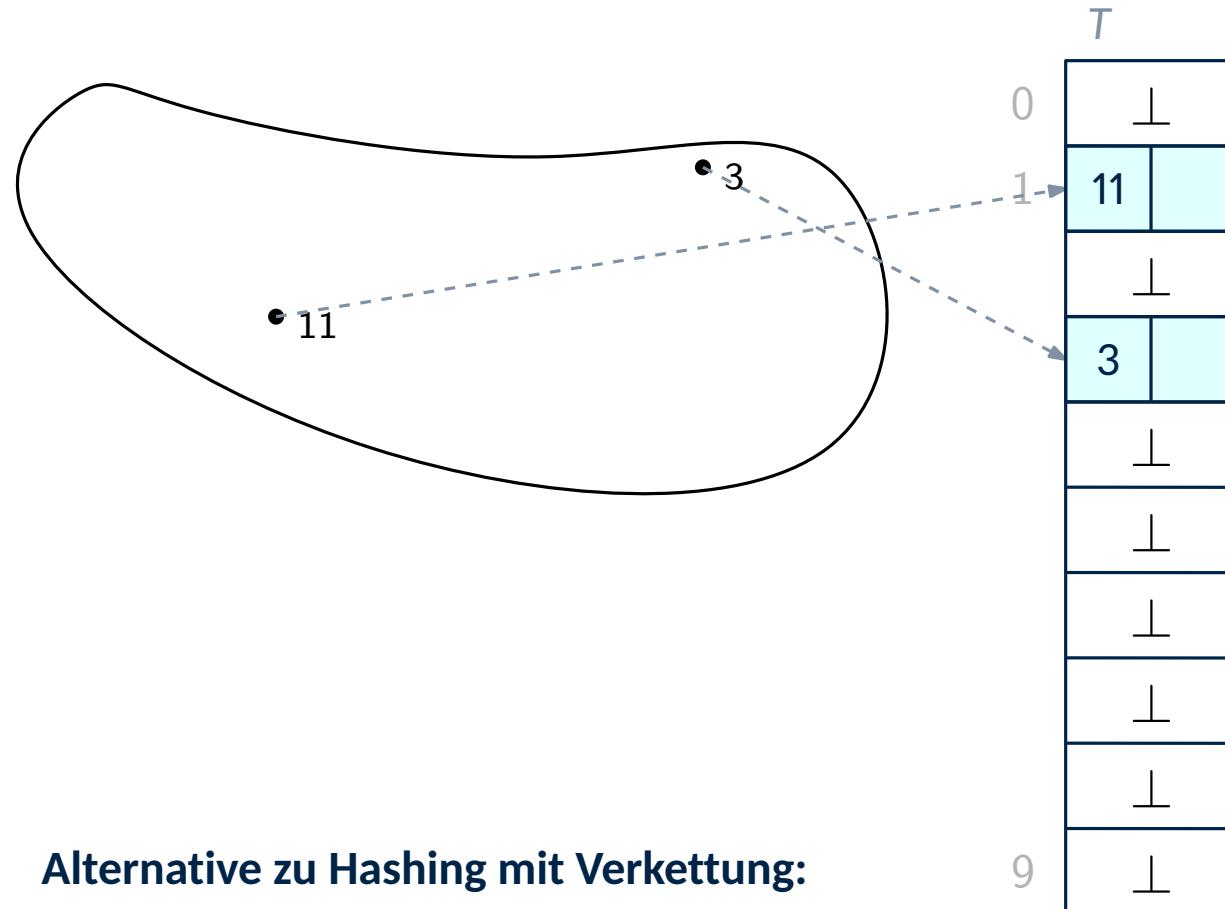
$T$
0
11
1
⊥
⊥
⊥
⊥
⊥
⊥
9
⊥

## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

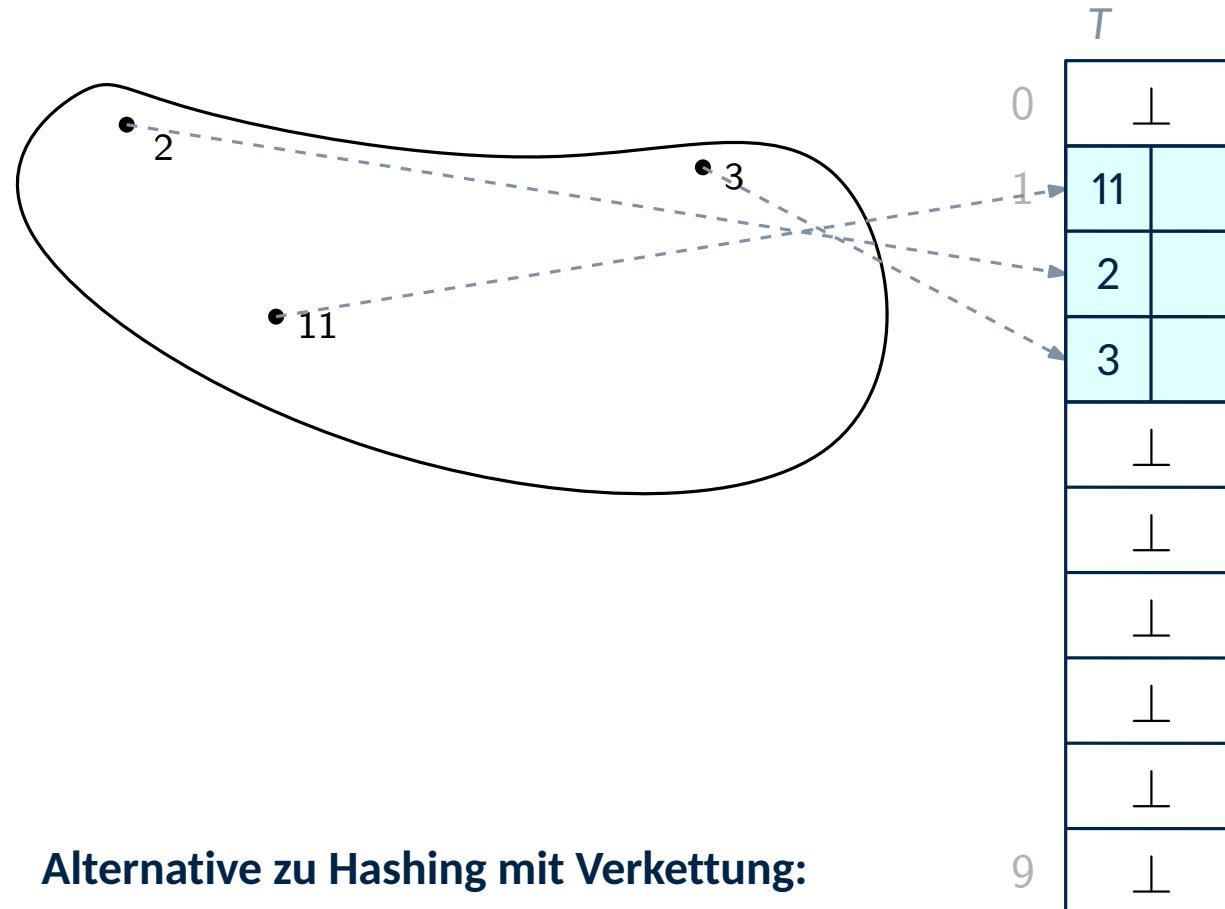


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

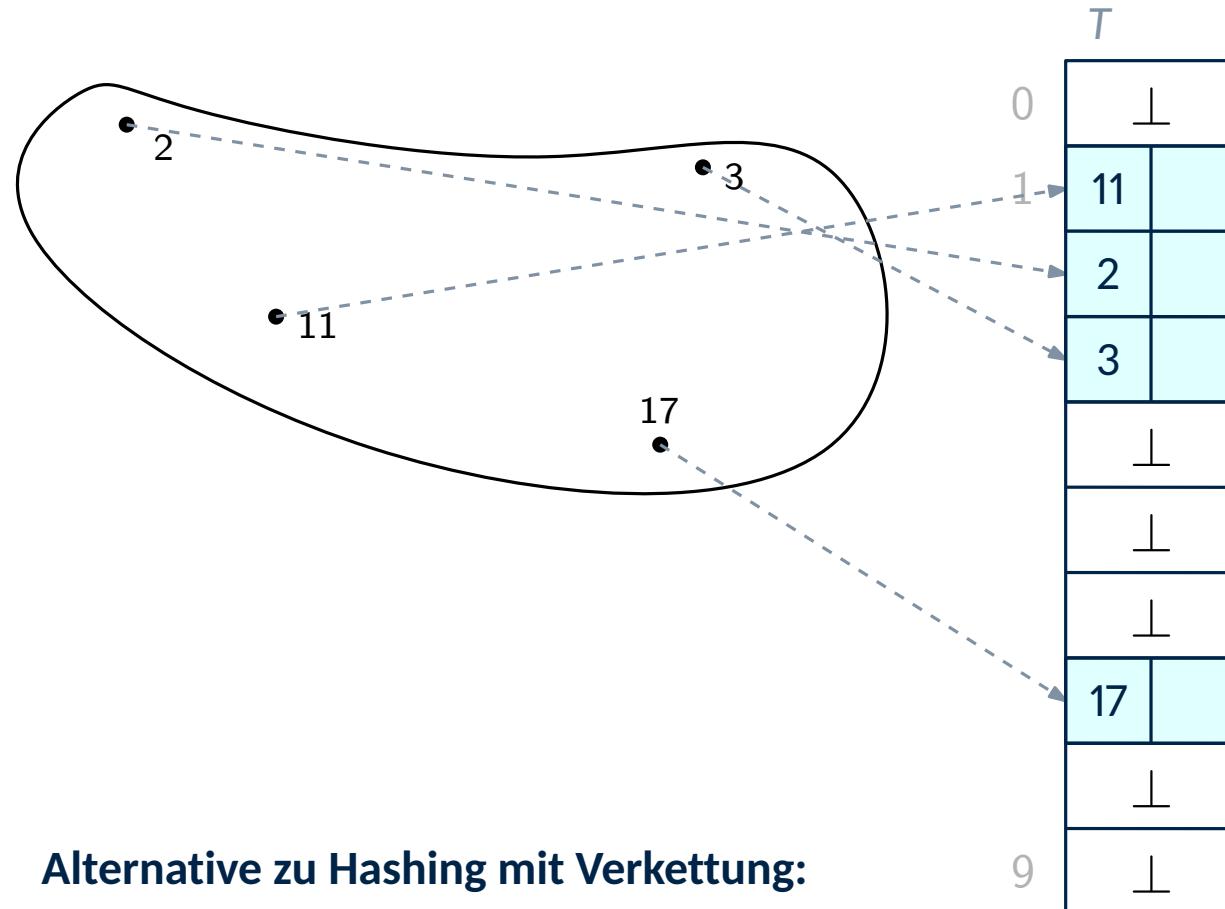


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

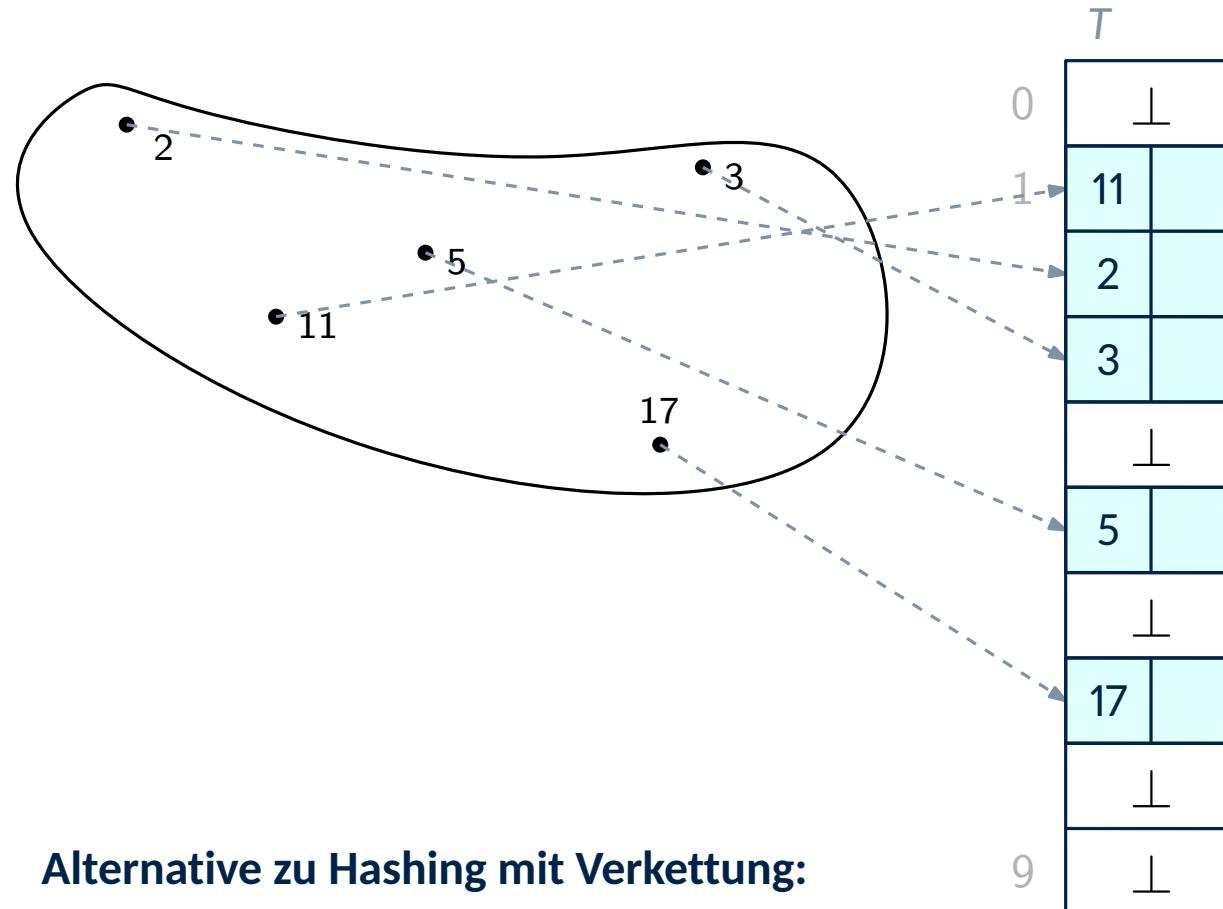


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

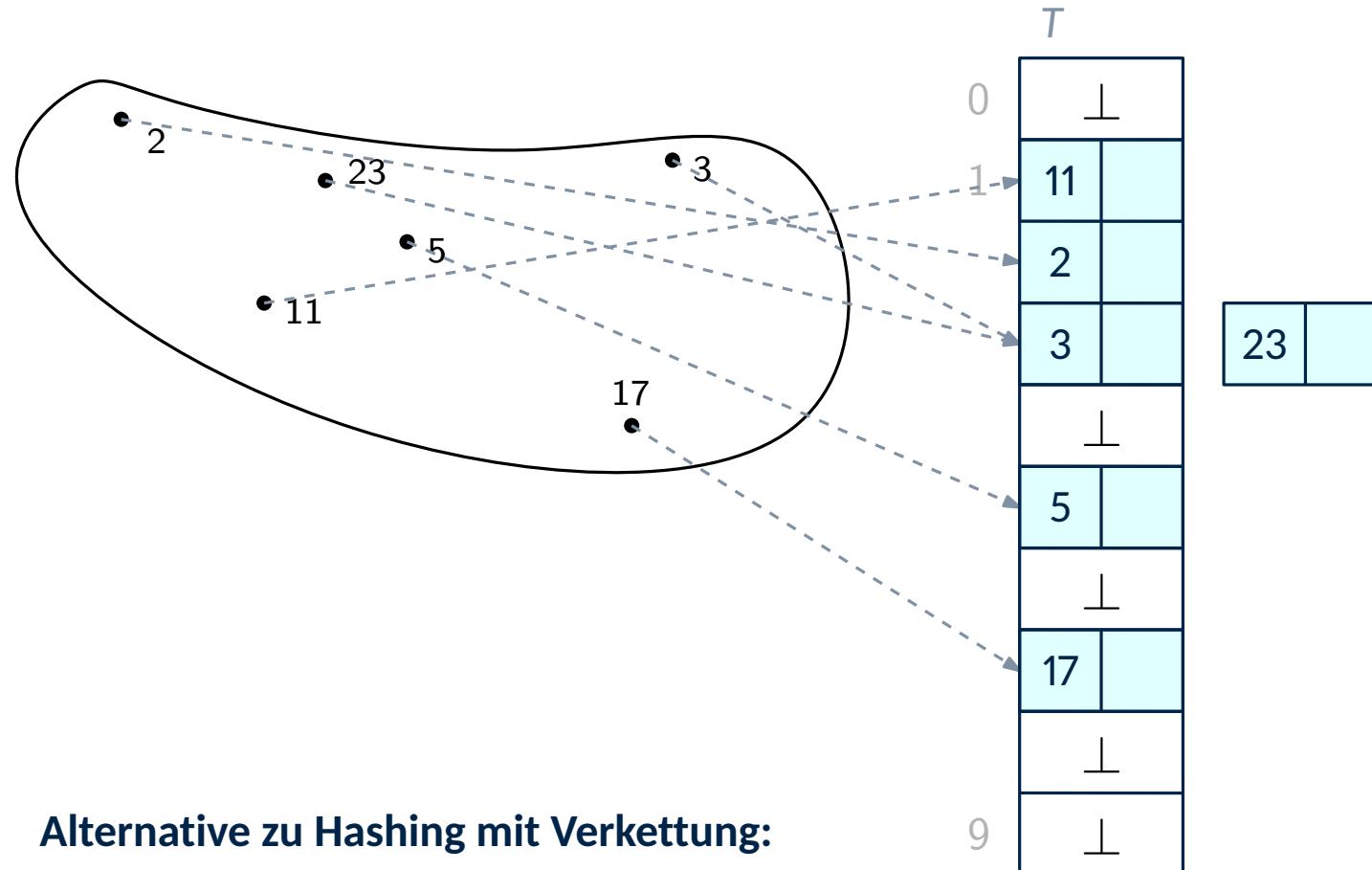


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

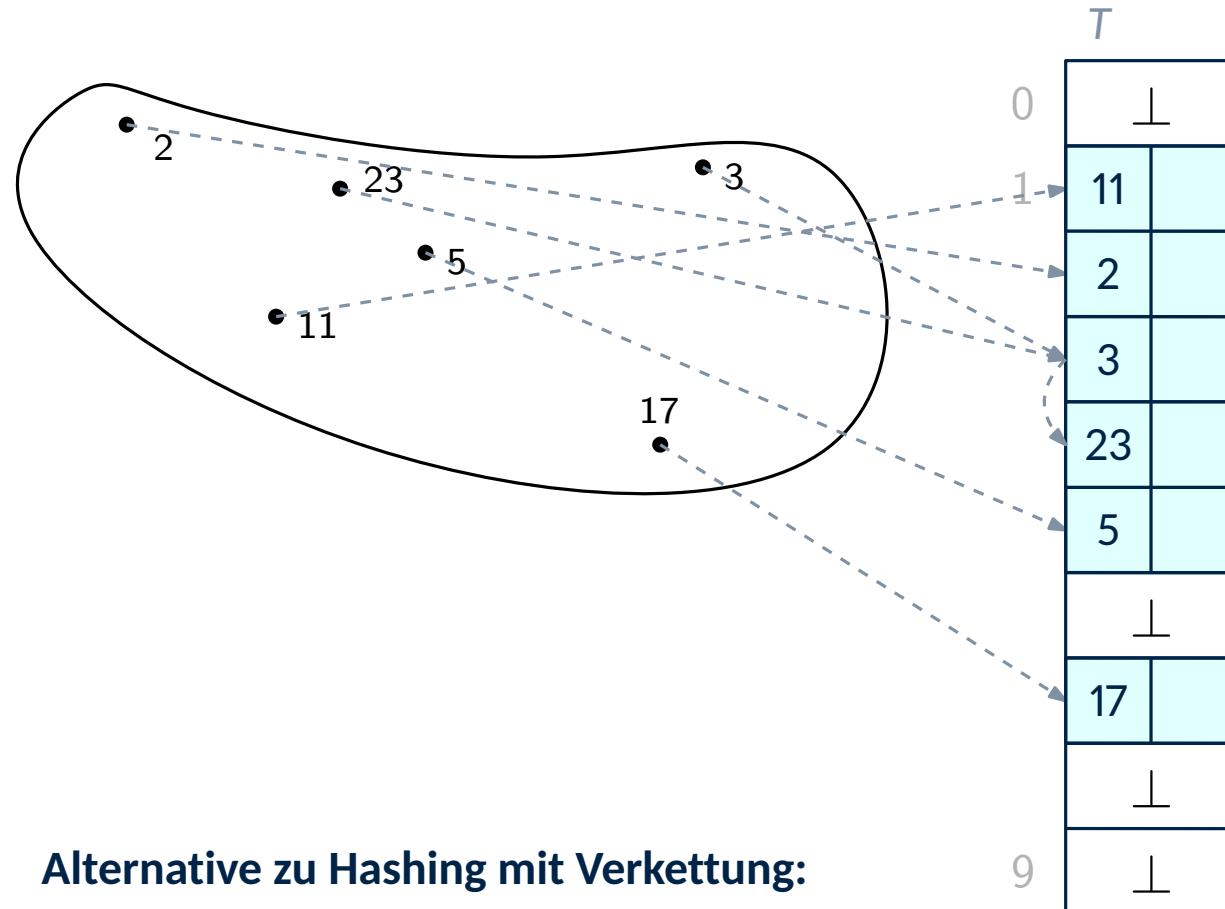


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

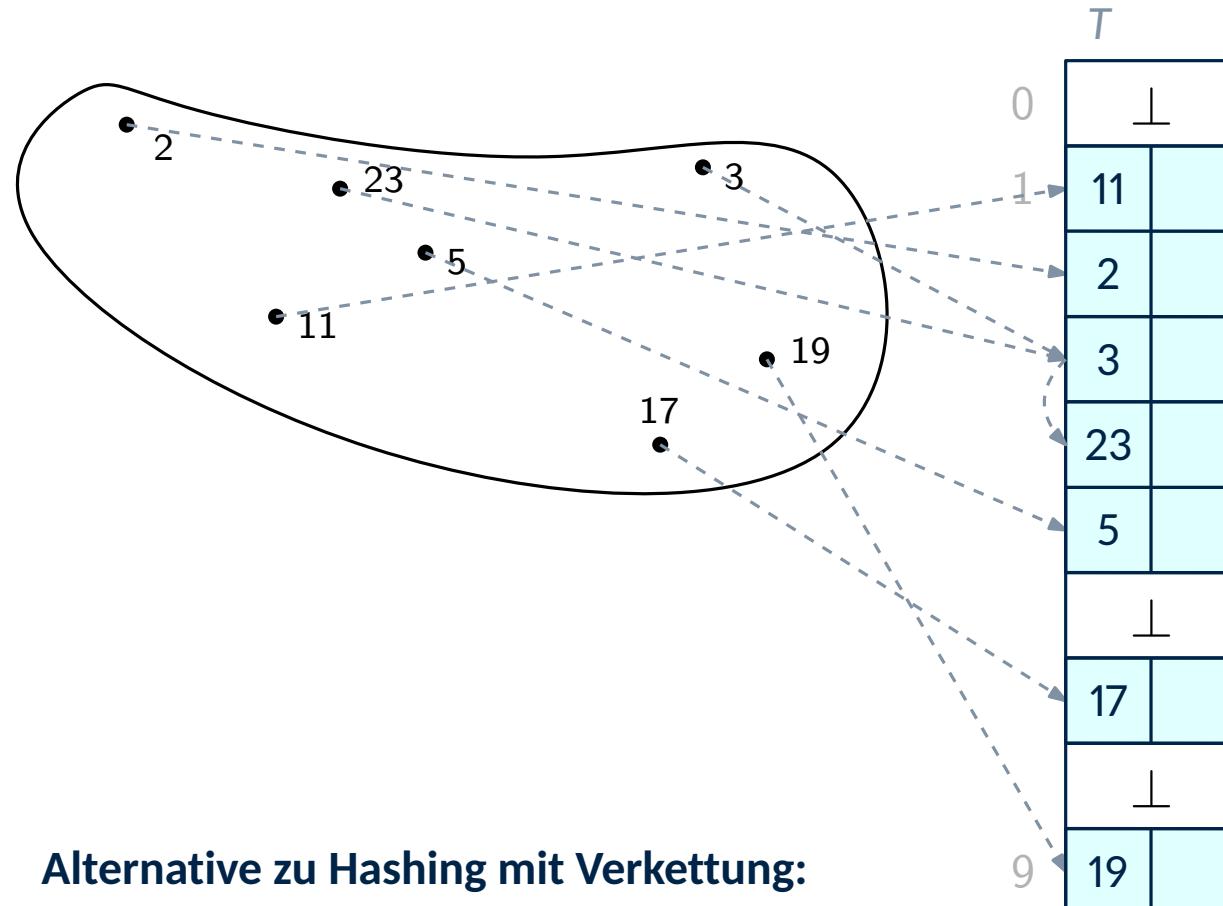


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

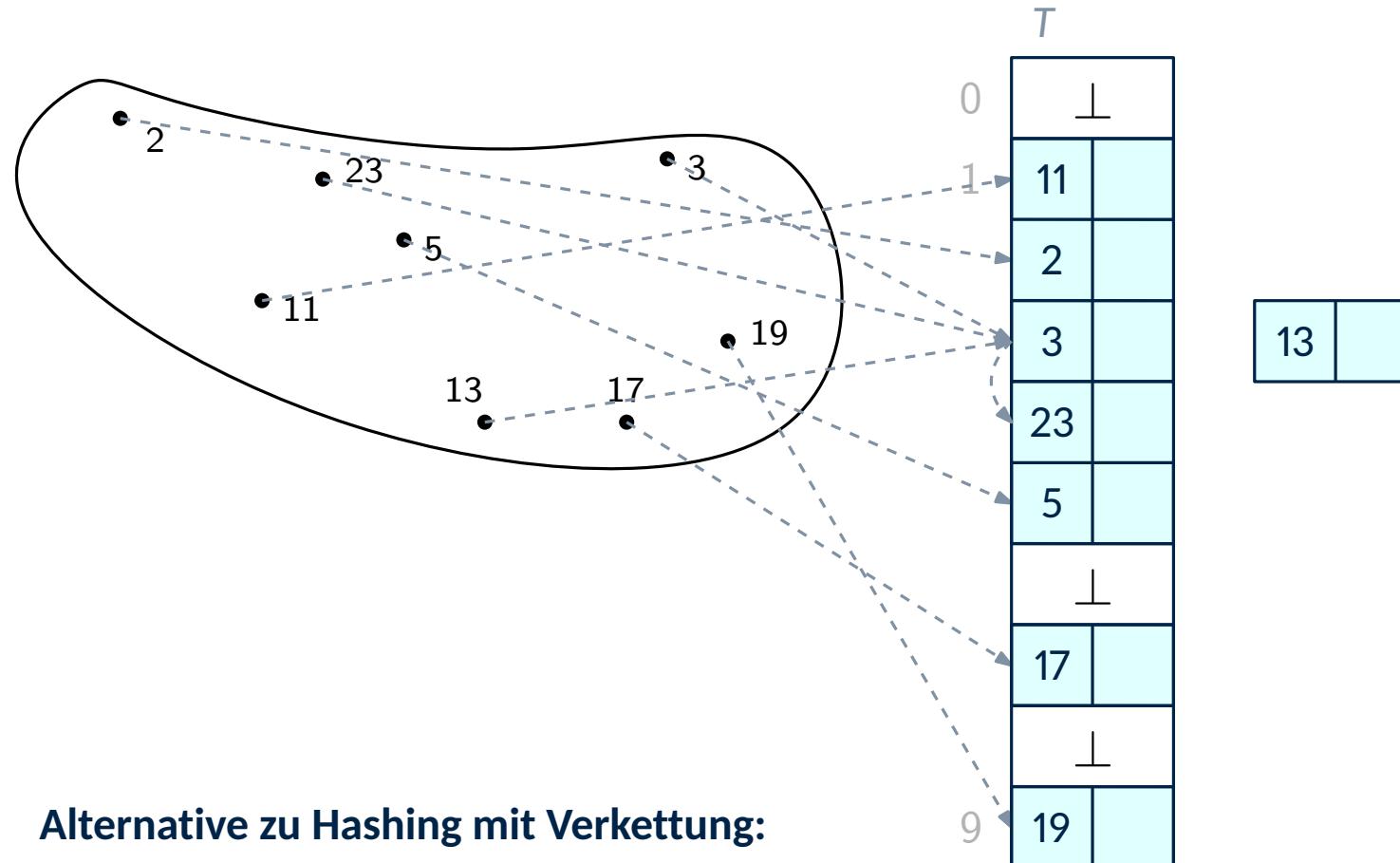


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

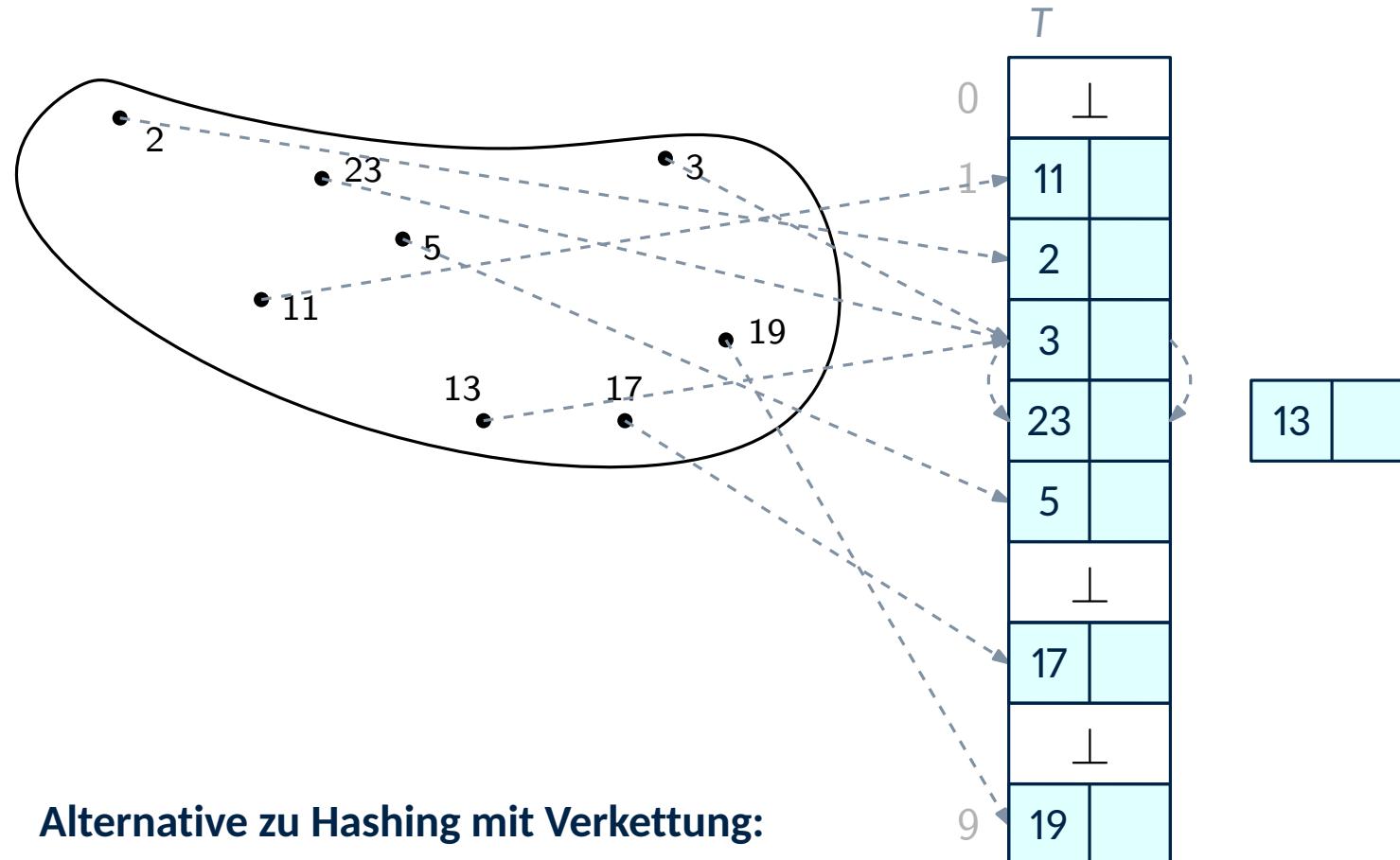


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

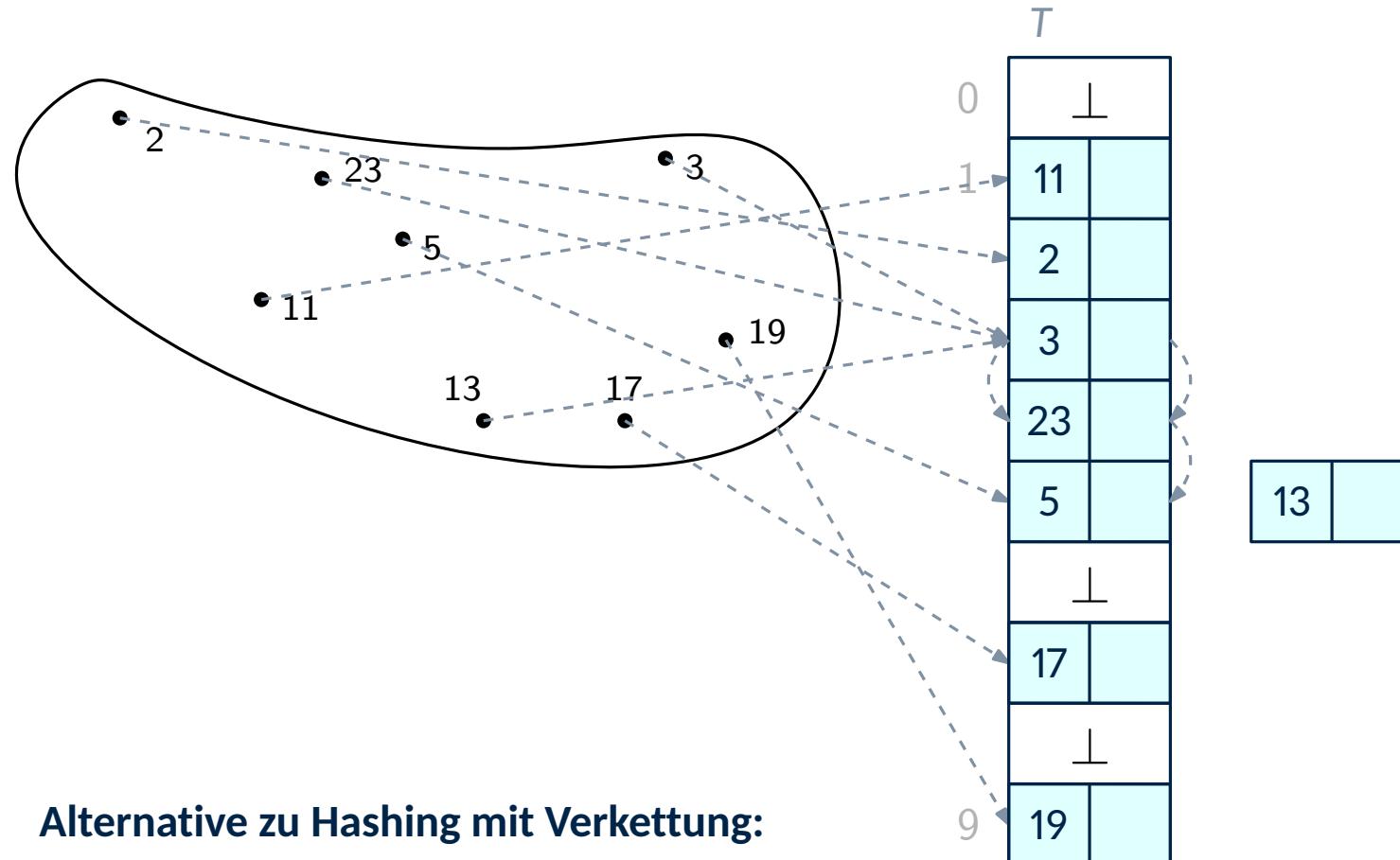


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

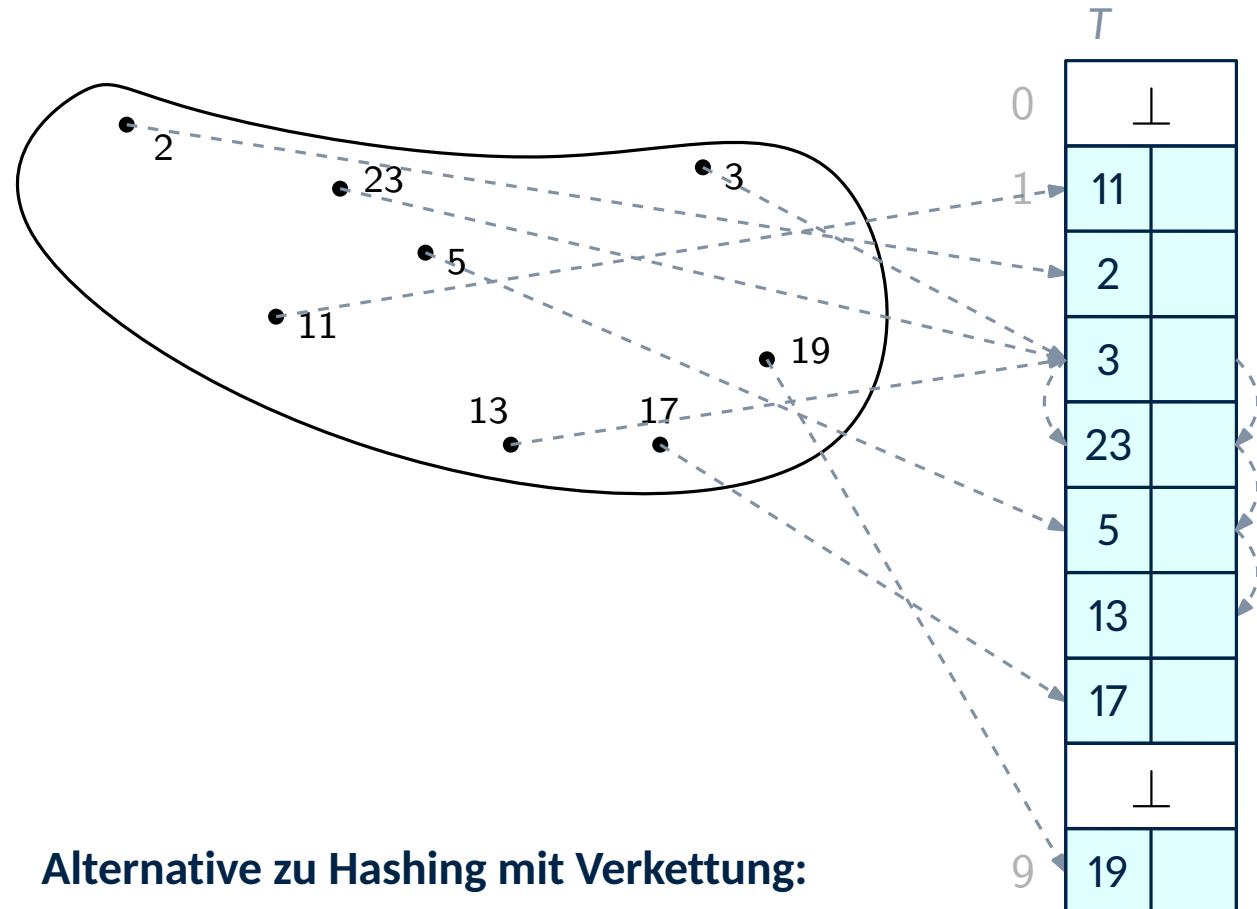


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

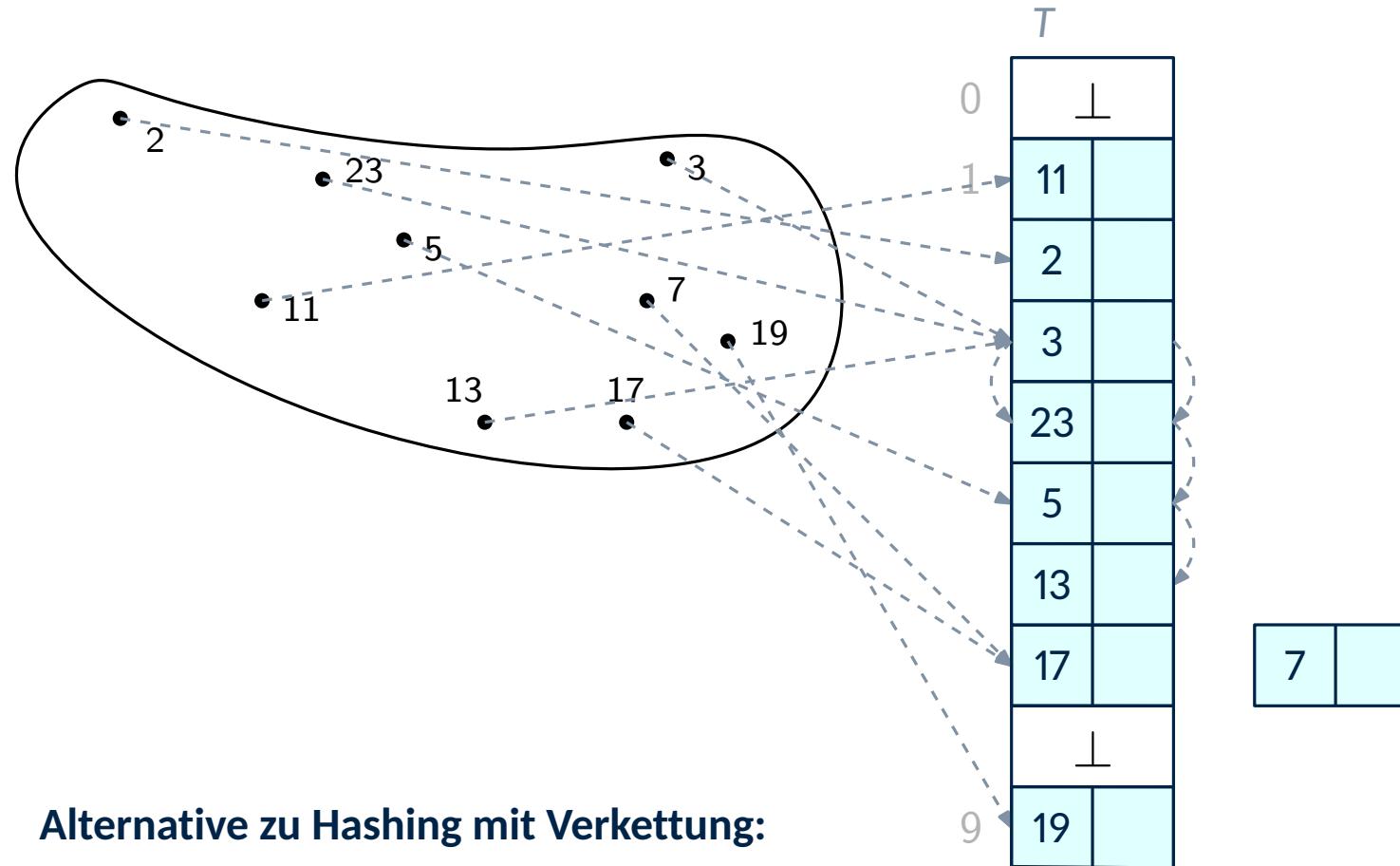


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

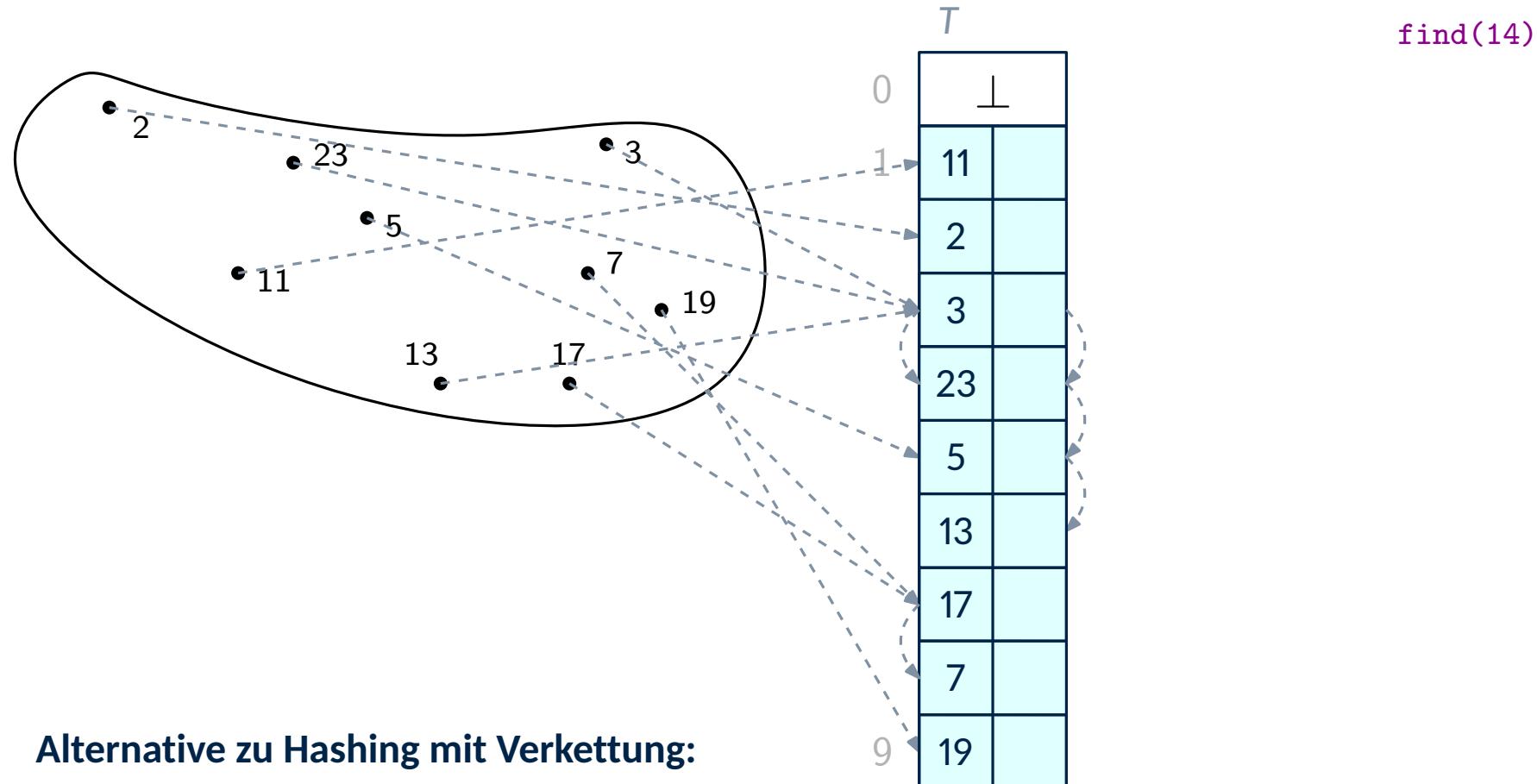


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

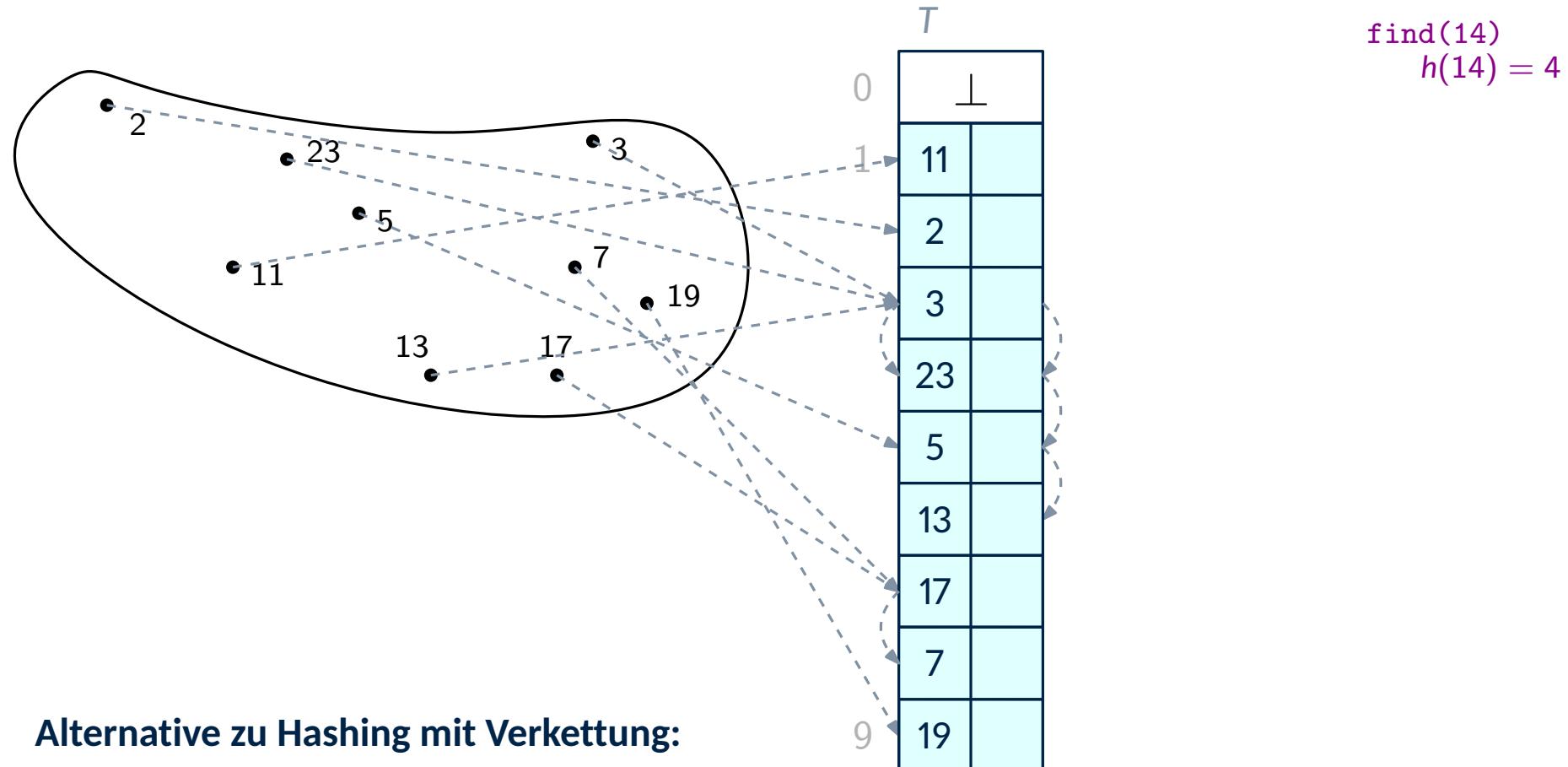


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

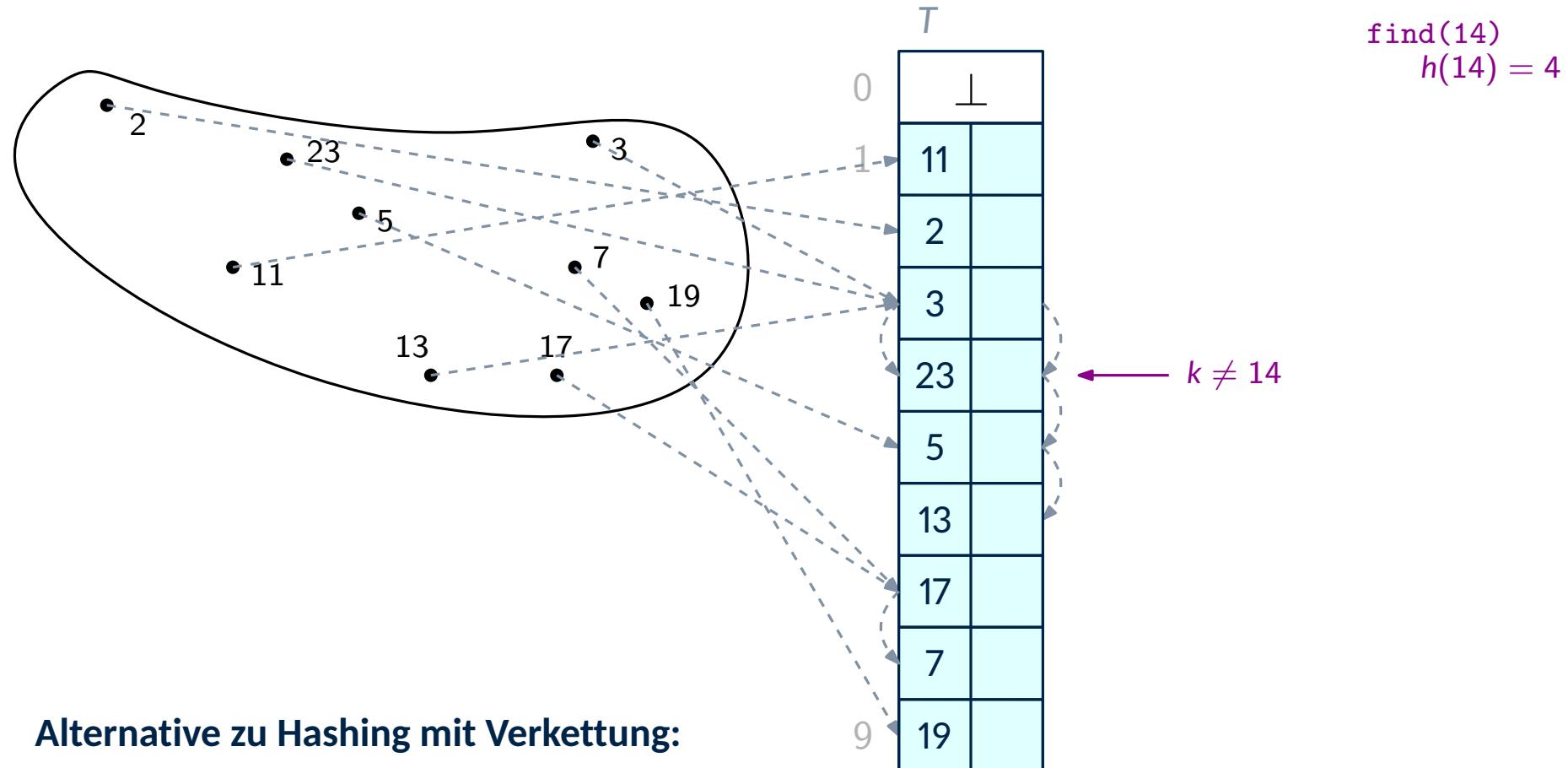


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

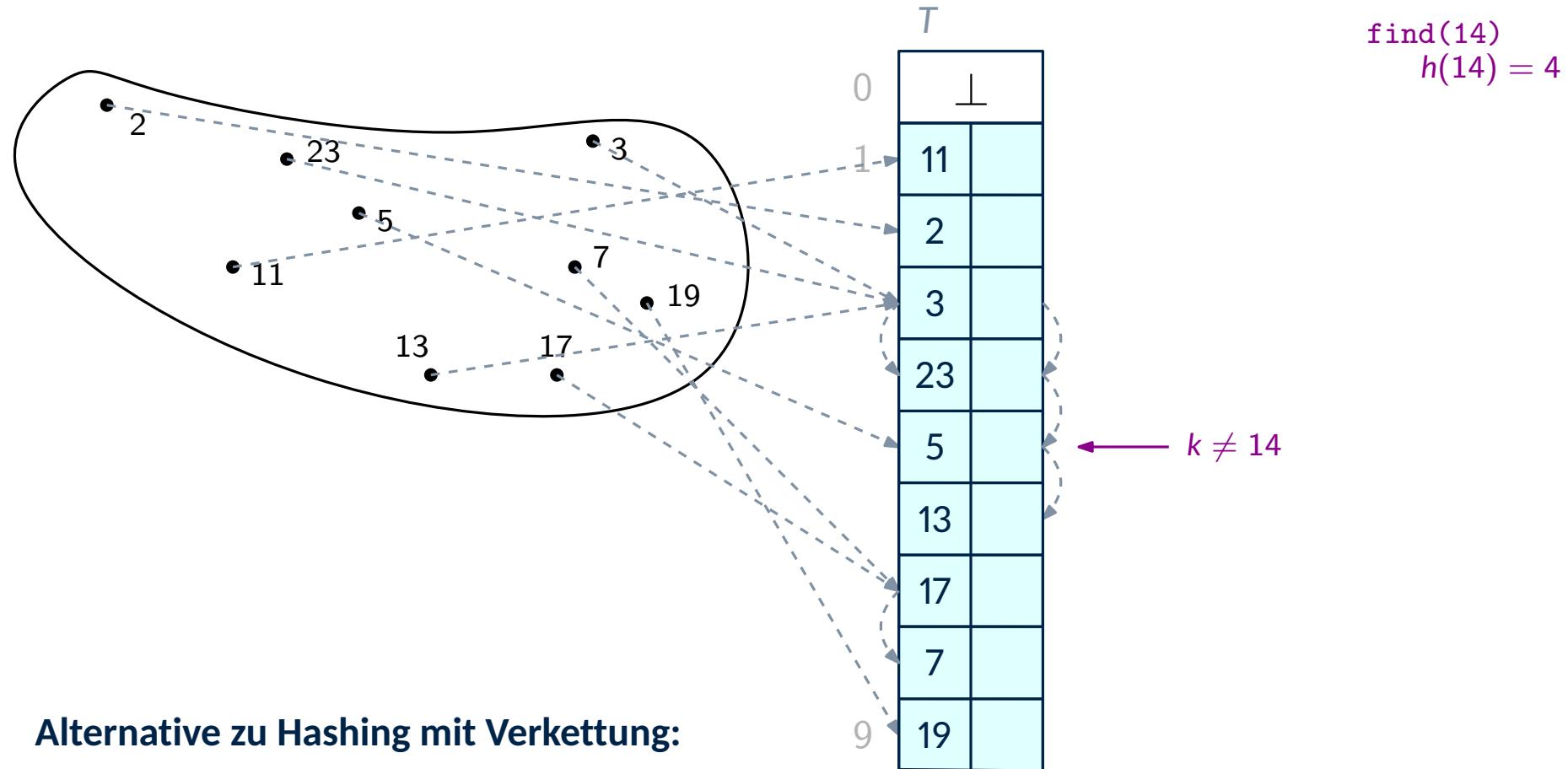


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

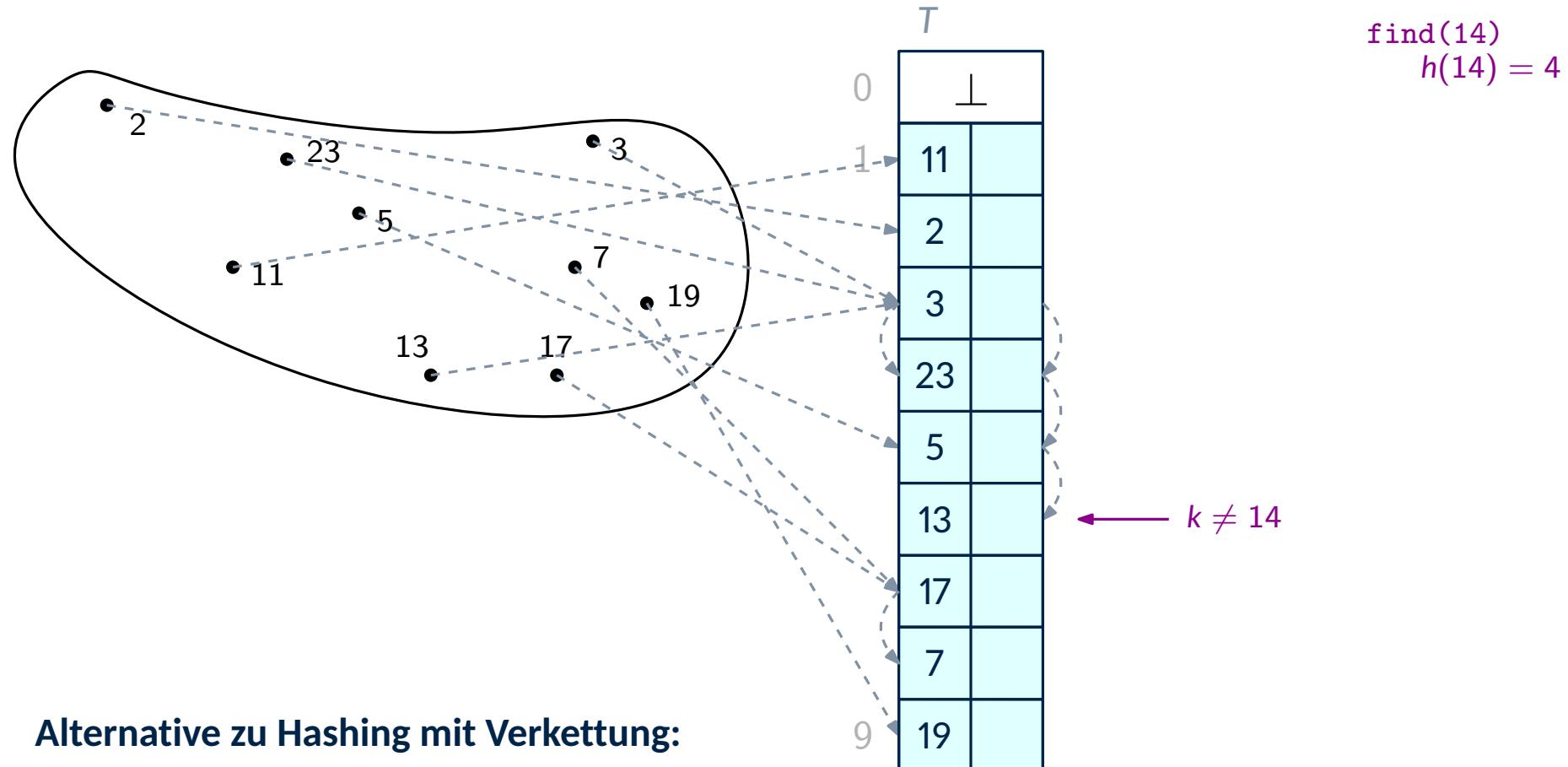


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

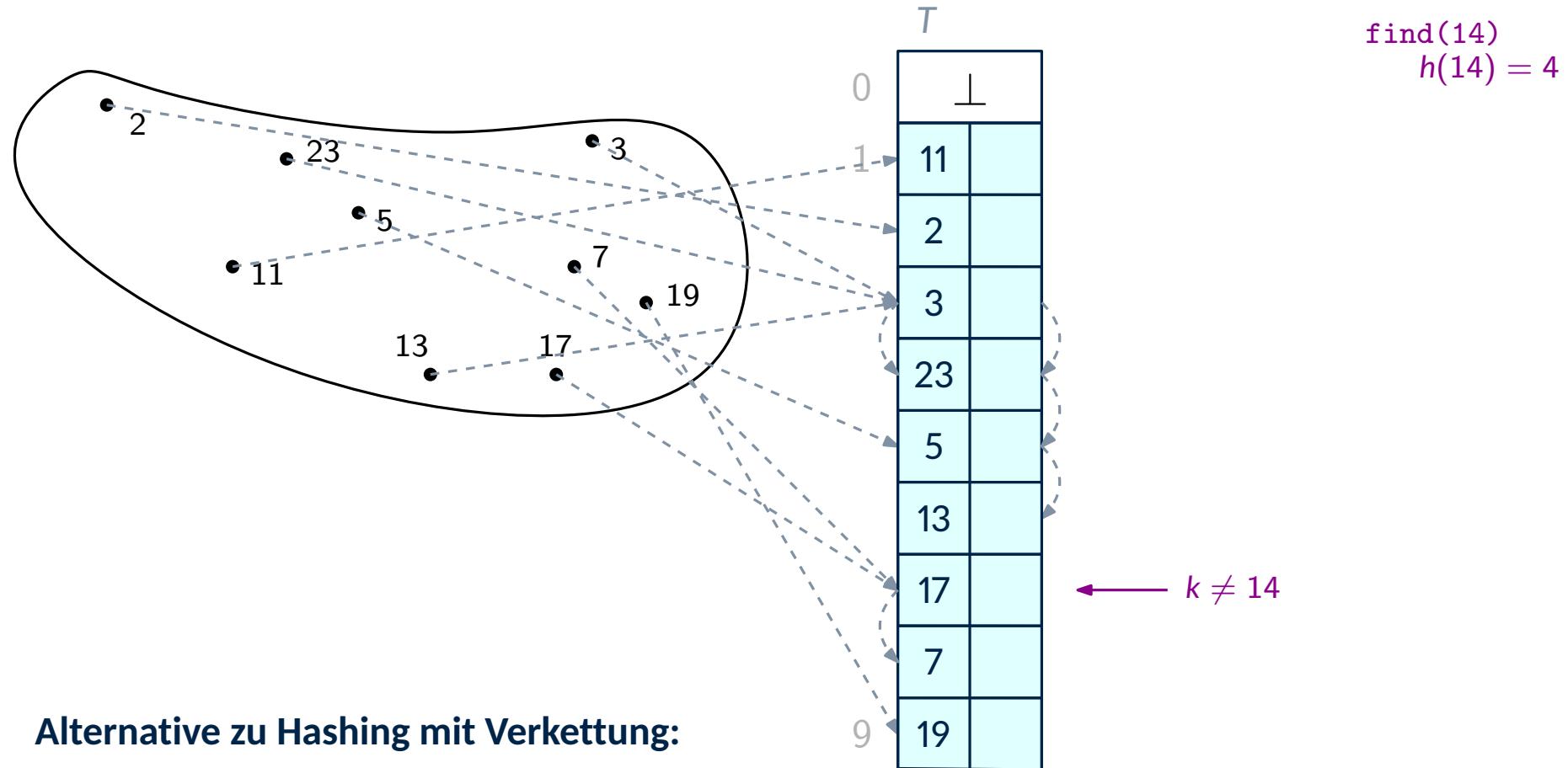


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

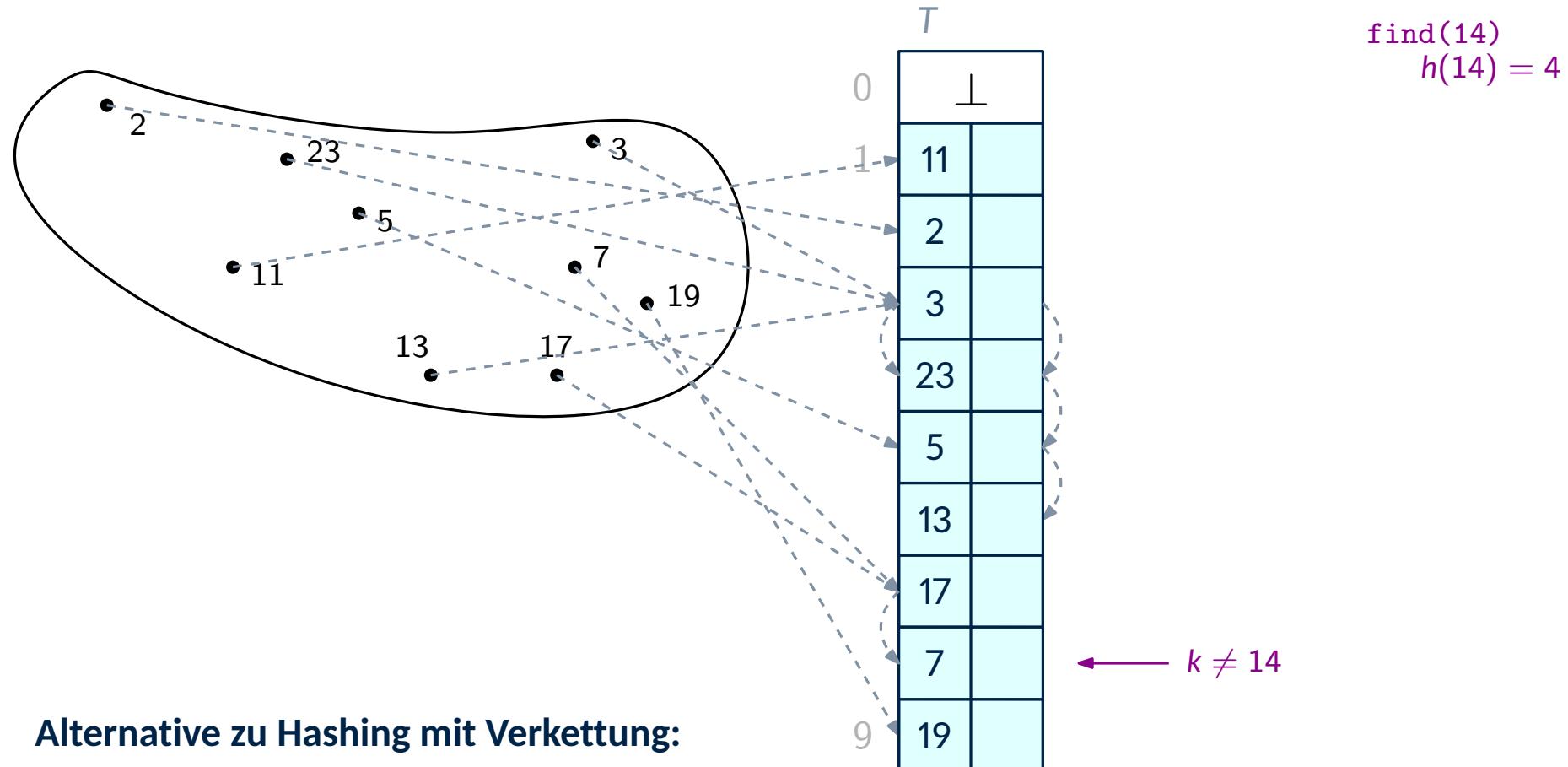


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

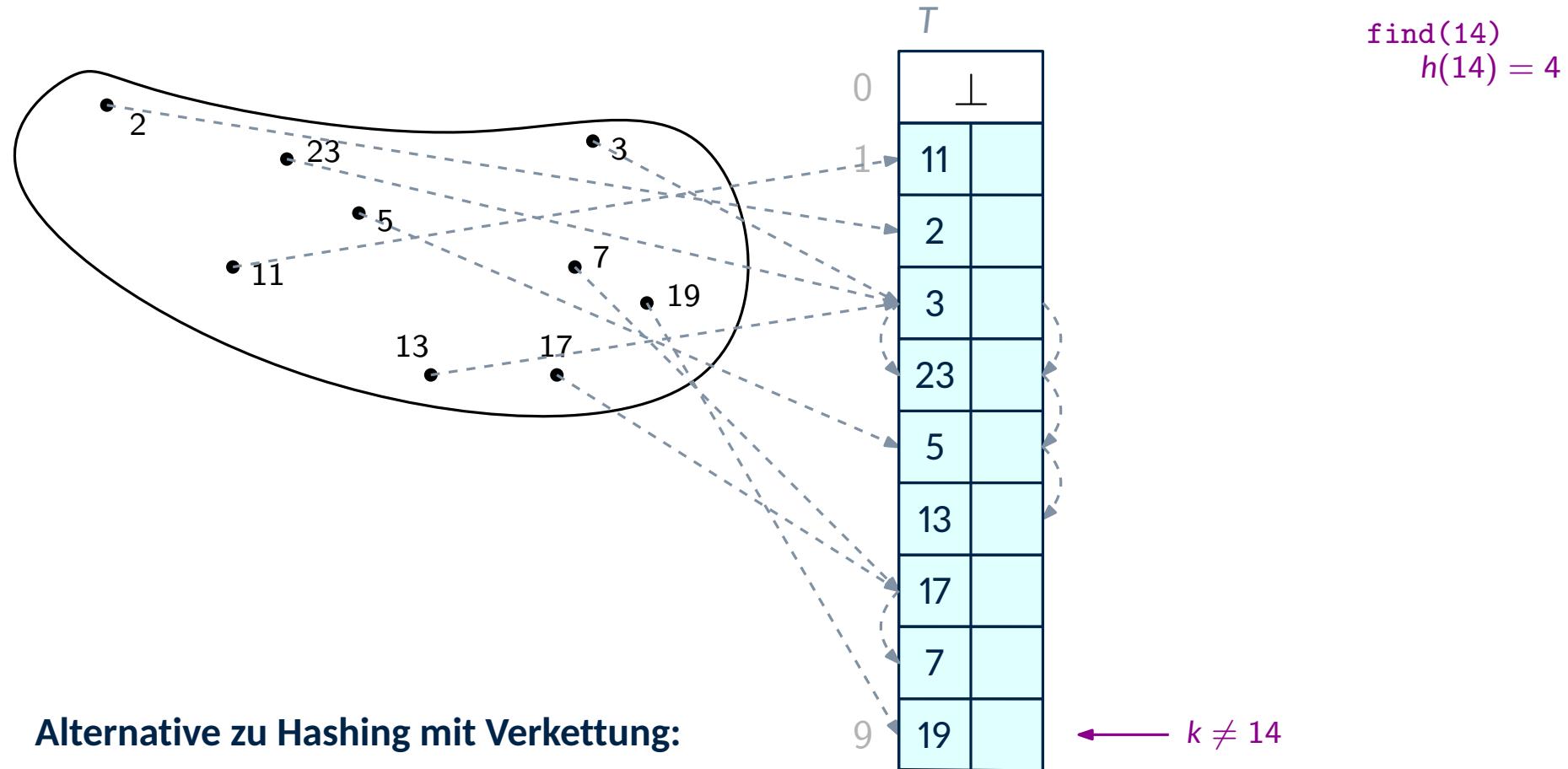


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

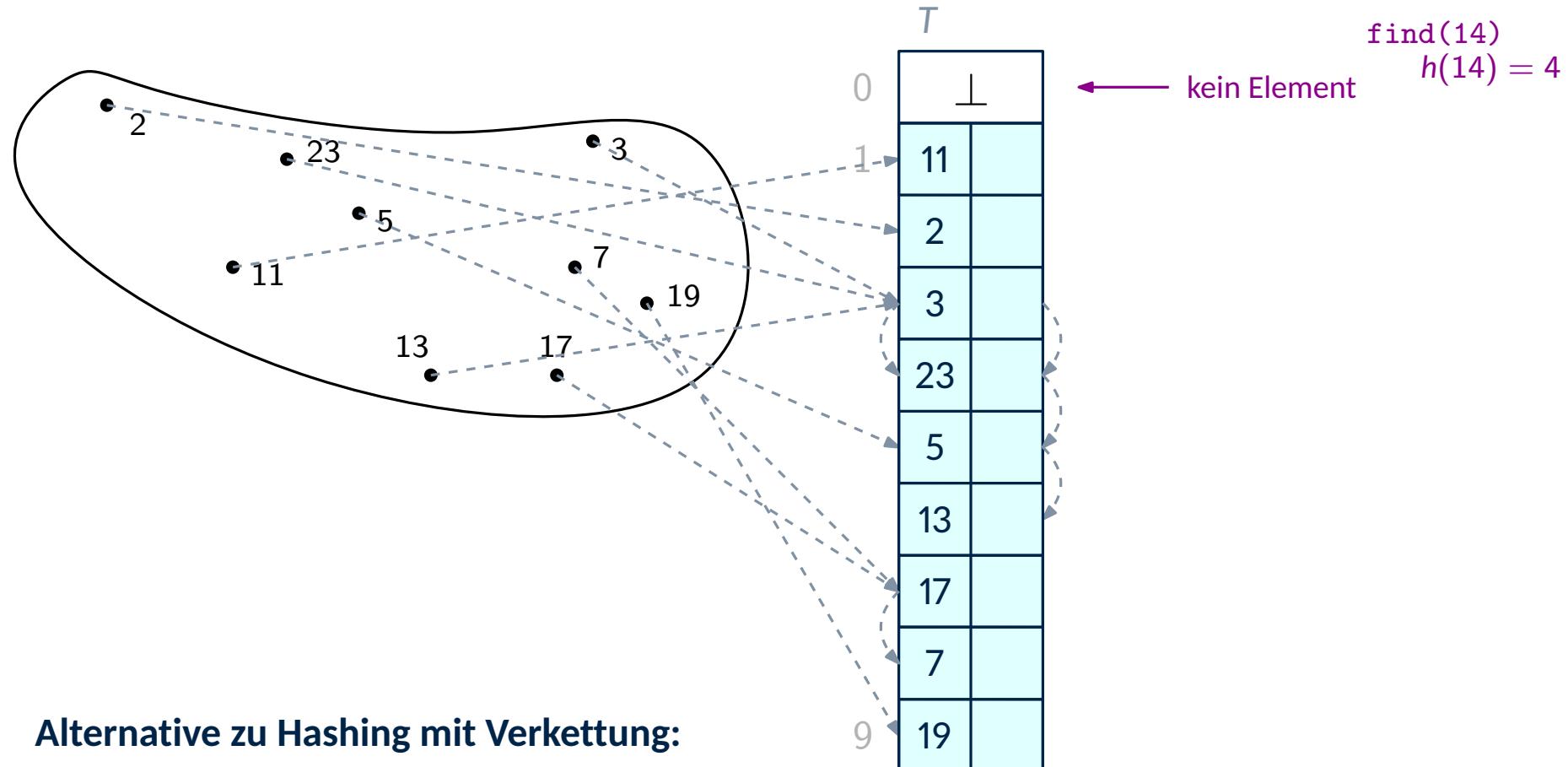


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"

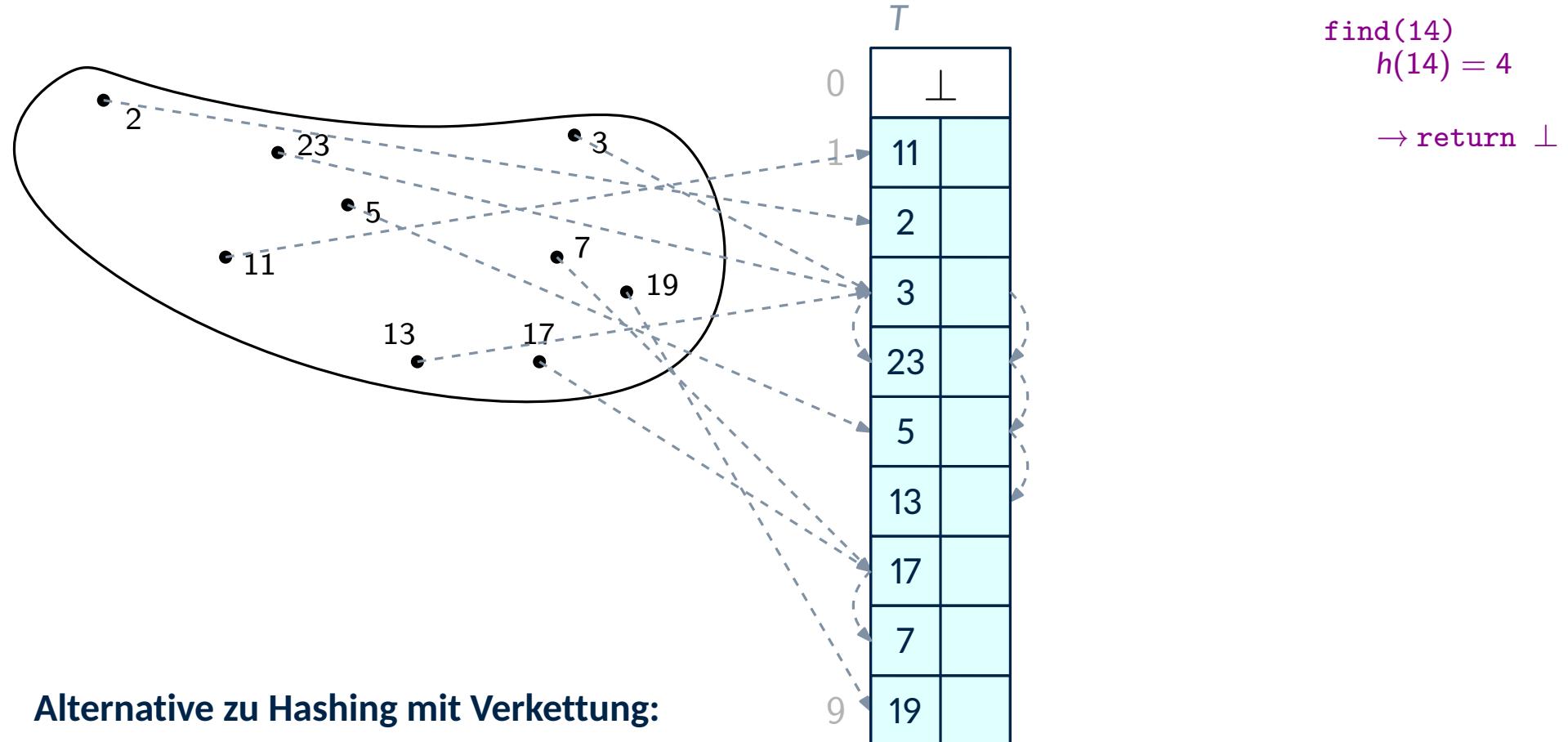


## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

# Alternative: Hashing mit Offener Adressierung

**Beispiel:** Es sei  $m = 10$  und  $h(x) := x \bmod 10$ . "Hashwert ist letzte Ziffer"



## Alternative zu Hashing mit Verkettung:

- Wir speichern die Elemente direkt in der Tabelle
- Bei Kollisionen suchen wir systematisch einen freien Platz
- einfachster Ansatz ist **Lineares Sondieren**: erhöhe den Index bis ein freier Platz gefunden wurde

⇒ Finden & Entfernen müssen Lineares Sondieren berücksichtigen!

# Hashing mit Linearem Sondieren

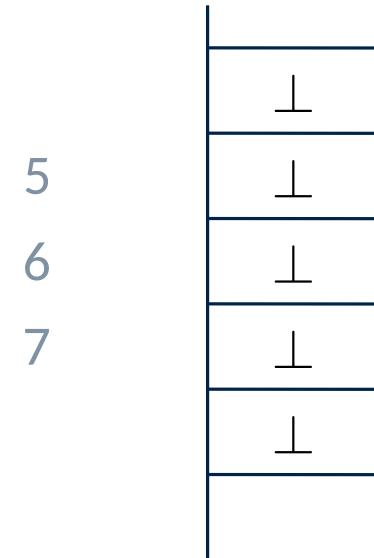
---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$
  - gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?
- `insert(k, v):`

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

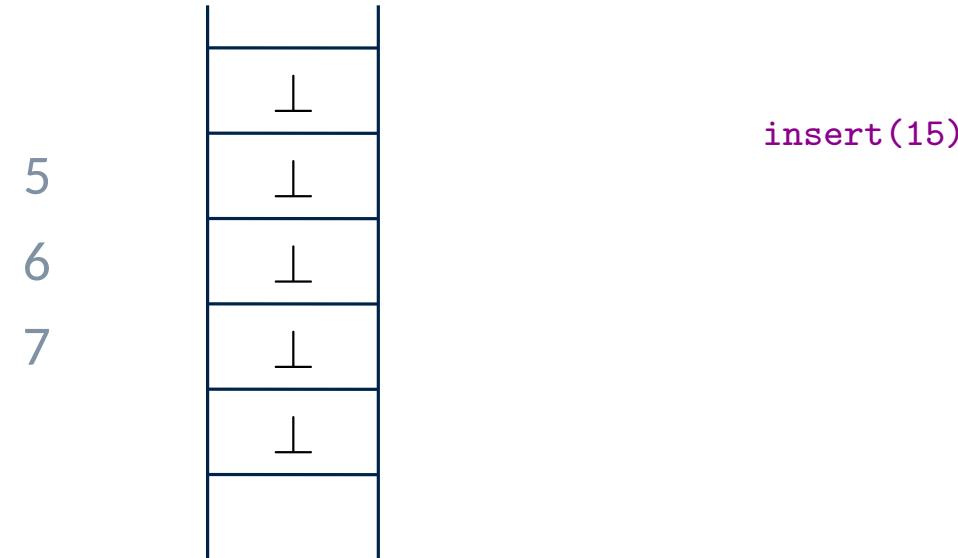


- `insert(k, v):`  
19 - 8

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

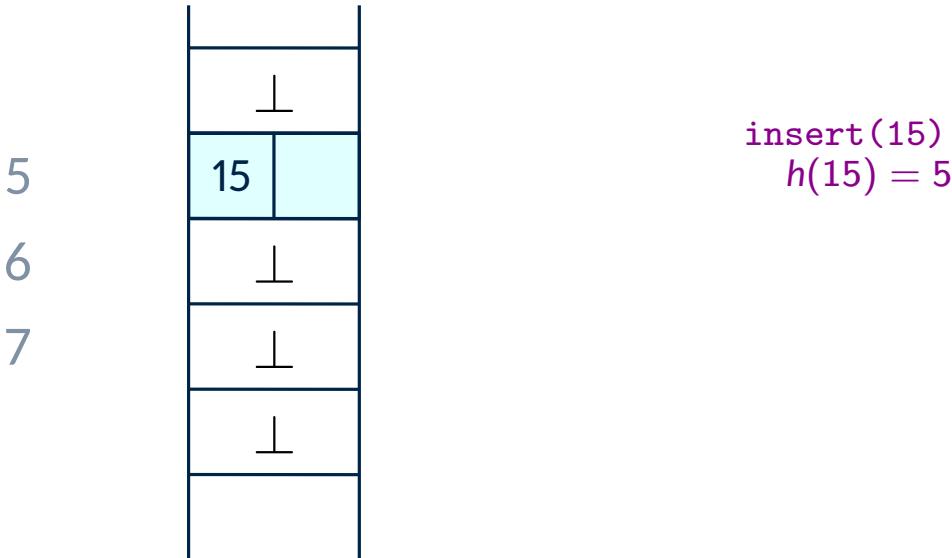


- `insert(k, v):`  
19 - 9

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

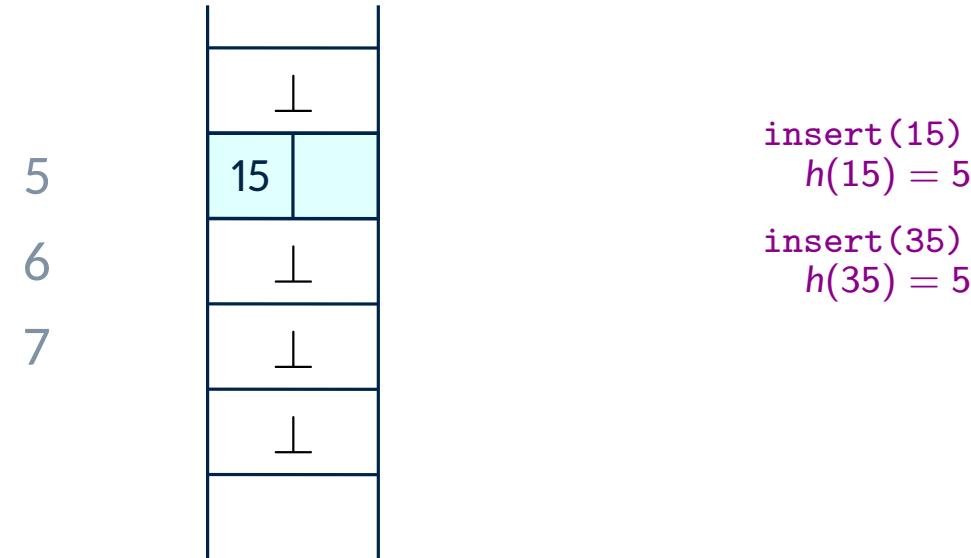


- `insert(k, v):`  
19 - 10

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

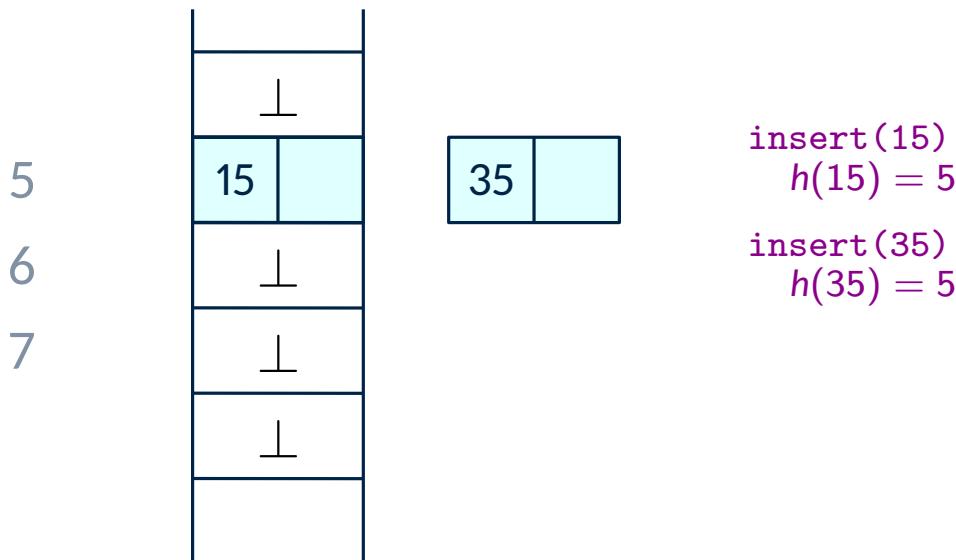


- `insert(k, v):`  
19 - 11

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

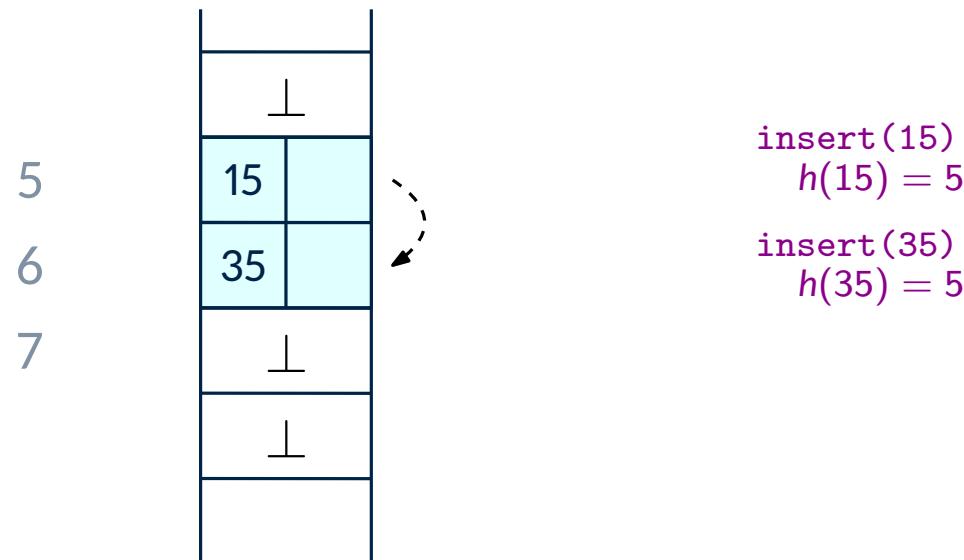


- `insert(k, v):`  
19 - 12

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

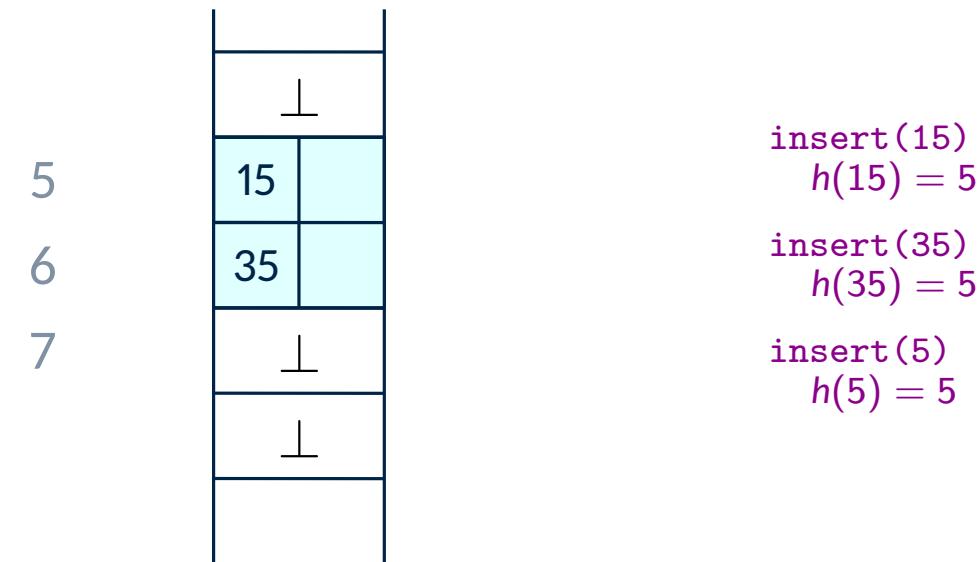


- `insert(k, v):`  
19 - 13

# Hashing mit Linearem Sondieren

---

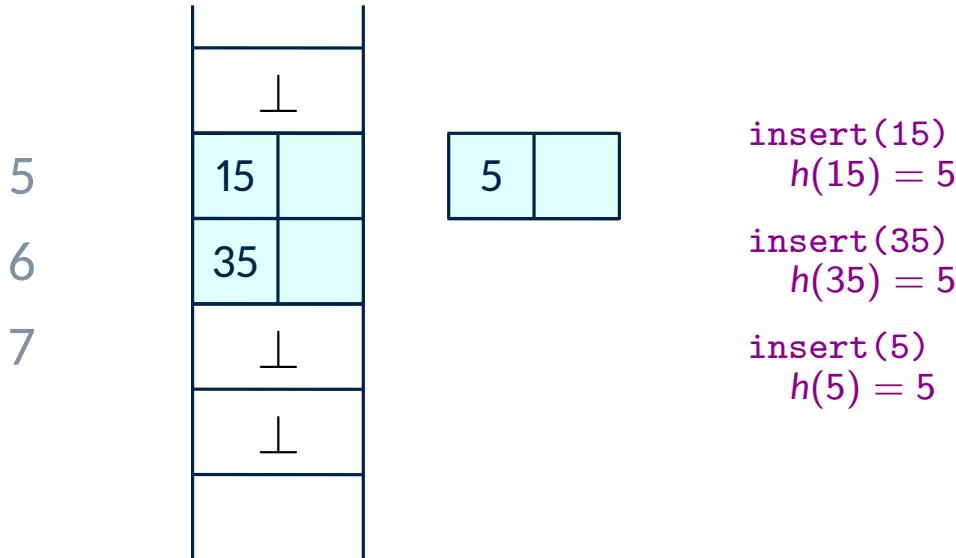
- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



- `insert(k, v):`  
19 - 14

# Hashing mit Linearem Sondieren

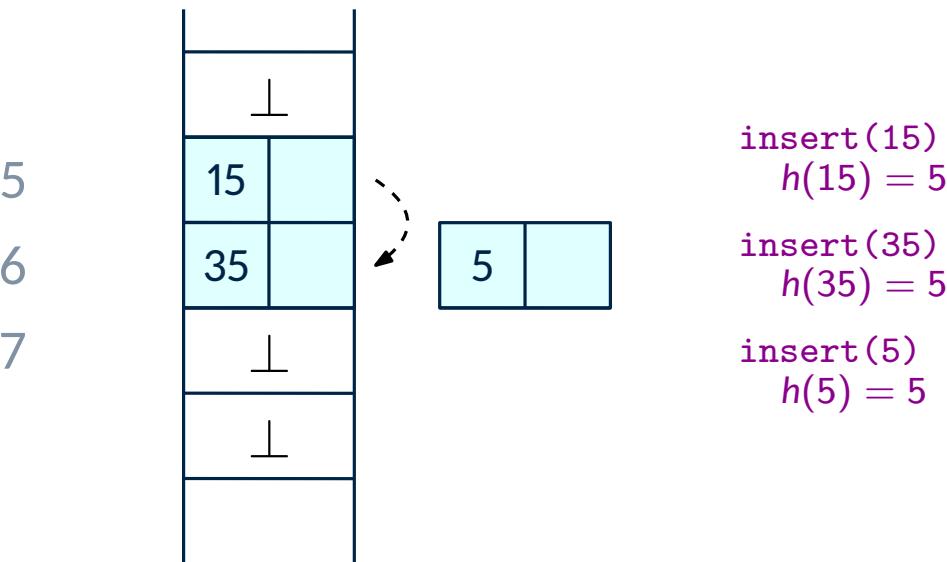
- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



- `insert(k, v):`  
19 - 15

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



- `insert(k, v):`  
19 - 16

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

5		15	
6		35	
7		5	
			$\perp$

insert(15)  
 $h(15) = 5$

insert(35)  
 $h(35) = 5$

insert(5)  
 $h(5) = 5$

- `insert(k, v):`

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

5	15	
6	35	
7	5	
	$\perp$	

`insert(15)`  
 $h(15) = 5$

`insert(35)`  
 $h(35) = 5$

`insert(5)`  
 $h(5) = 5$

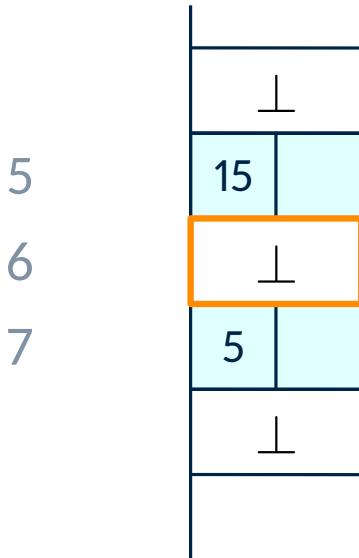
`remove(35)`

- `insert(k, v):`  
19 - 18

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



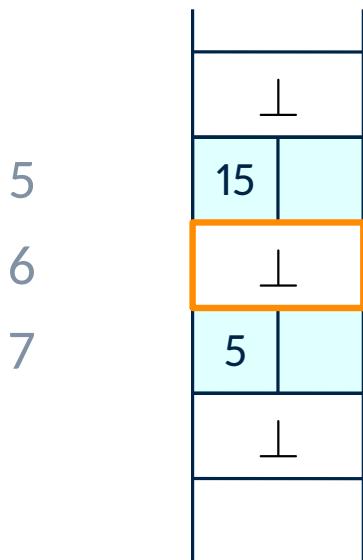
`insert(15)`  
 $h(15) = 5$   
`insert(35)`  
 $h(35) = 5$   
`insert(5)`  
 $h(5) = 5$   
`remove(35)`

- `insert(k, v):`  
19 - 19

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:

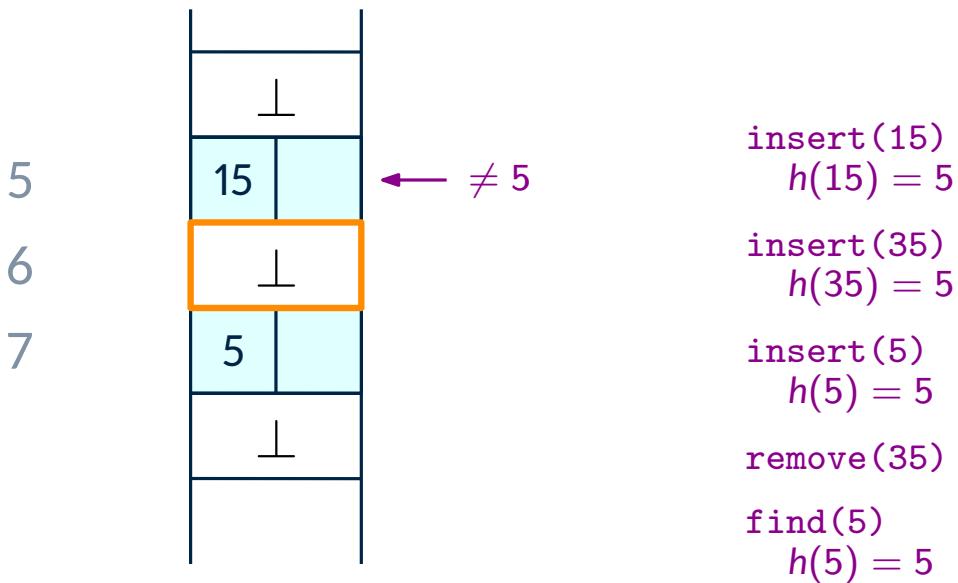


`insert(15)`  
 $h(15) = 5$   
`insert(35)`  
 $h(35) = 5$   
`insert(5)`  
 $h(5) = 5$   
`remove(35)`  
`find(5)`

- `insert(k, v):`  
19 - 20

# Hashing mit Linearem Sondieren

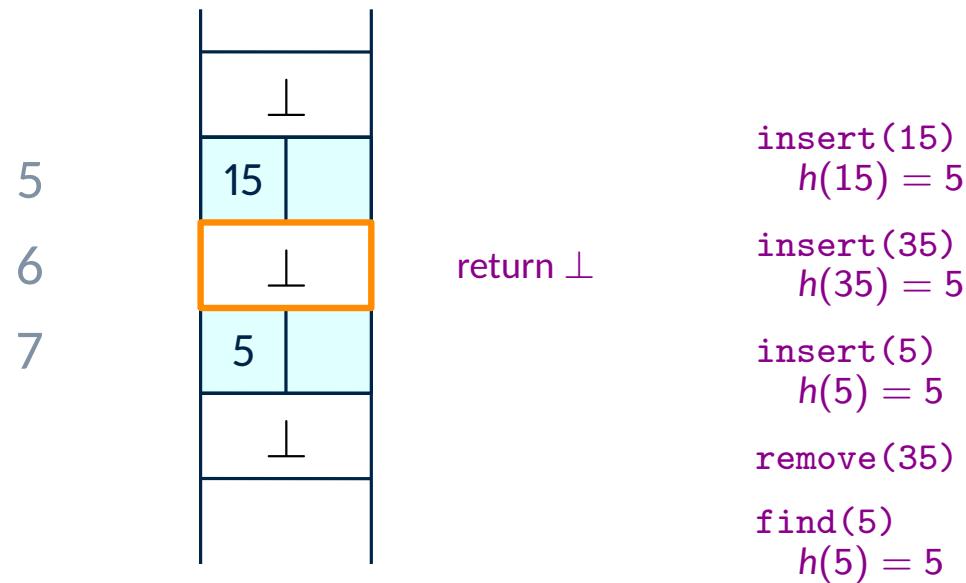
- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



- `insert(k, v):`  
19 - 21

# Hashing mit Linearem Sondieren

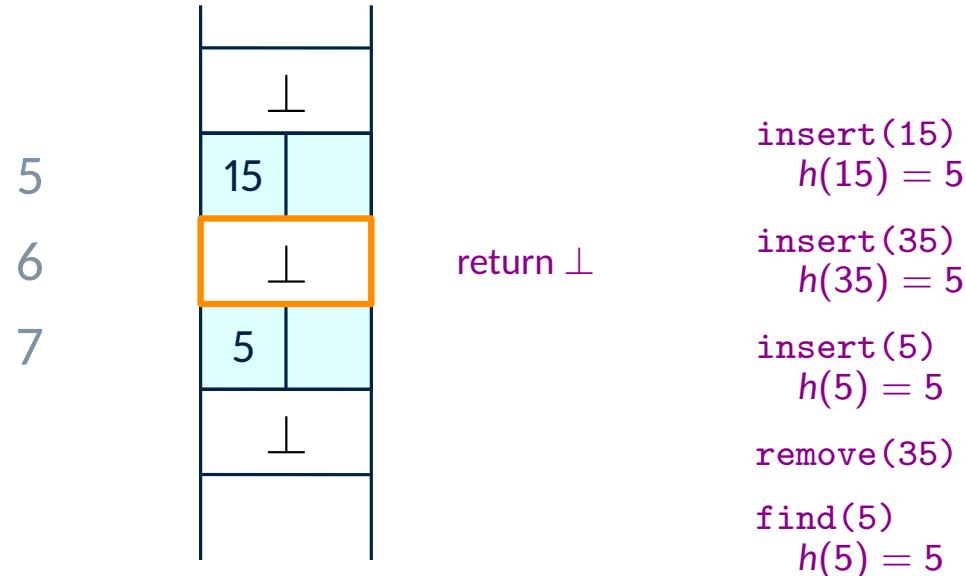
- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



- `insert(k, v):`  
19 - 22

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, denn dann:



**Problem:**

zwischen  $h(k)$  und tatsächlichem Fundort von  $k$  befindet sich  $\perp$

- `insert(k, v):`

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?

Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :

Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$

"Wraparound"  $(\bmod m)$  beachten

## Möglicher Umgang mit dem Problem:

1. wir verbieten `remove`

→ inkrementelles Wörterbuch

2. gelöschte Elemente **markieren**

wir speichern zu jedem Element,  
ob es als gelöscht markiert ist

- `insert(k, v):`

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :
  - | **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?

Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :

Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$

"Wraparound"  $(\bmod m)$  beachten

## Möglicher Umgang mit dem Problem:

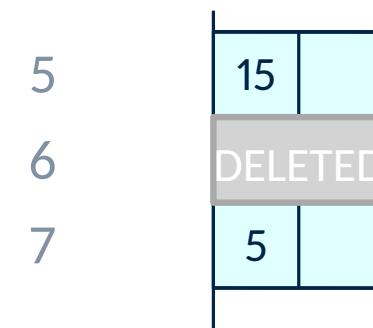
1. wir verbieten remove

→ inkrementelles Wörterbuch

2. gelöschte Elemente **markieren**

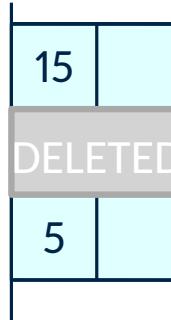
wir speichern zu jedem Element,  
ob es als gelöscht markiert ist

- `insert(k, v):`



# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp:$  und  $T[j]$  ist nicht als gelöscht markiert
    - | wenn  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
    - |  $j = (j + 1) \bmod m$
    - gib  $\perp$  zurück
  - `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :  
Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$   
"Wraparound"  $(\bmod m)$  beachten
  - **Möglicher Umgang mit dem Problem:**

1. wir verbieten <code>remove</code> → inkrementelles Wörterbuch	2. gelöschte Elemente <b>markieren</b> wir speichern zu jedem Element, ob es als gelöscht markiert ist
---	--
  - `insert(k, v):`
- 19 - 30
- 

# Hashing mit Linearem Sondieren

---

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :  
  |   **wenn**  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück  
  |    $j = (j + 1) \bmod m$   
  gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?

Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :

Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$

"Wraparound"  $(\bmod m)$  beachten

## Möglicher Umgang mit dem Problem:

1. wir verbieten `remove`

→ inkrementelles Wörterbuch

2. gelöschte Elemente **markieren**

wir speichern zu jedem Element,  
ob es als gelöscht markiert ist

### Nachteil:

gelöschte Elemente beanspruchen Platz!

Ggf. sollte Tabelle reorganisiert werden,  
wenn es zu viele gelöschte Elemente gibt

- `insert(k, v):`

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp:$  und  $T[j]$  ist nicht als gelöscht markiert
  - | wenn  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$
  - gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?

Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :

Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$

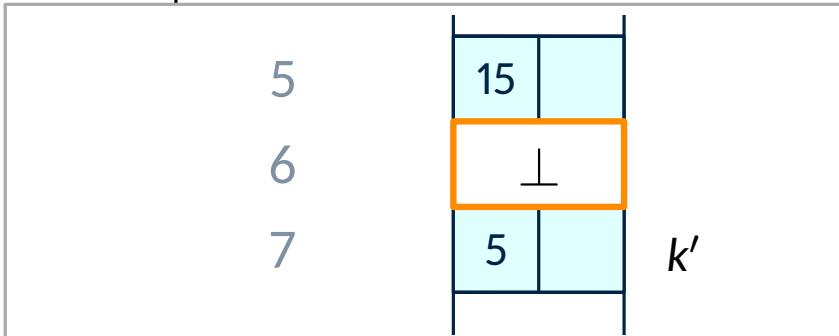
"Wraparound"  $(\bmod m)$  beachten

## Möglicher Umgang mit dem Problem:

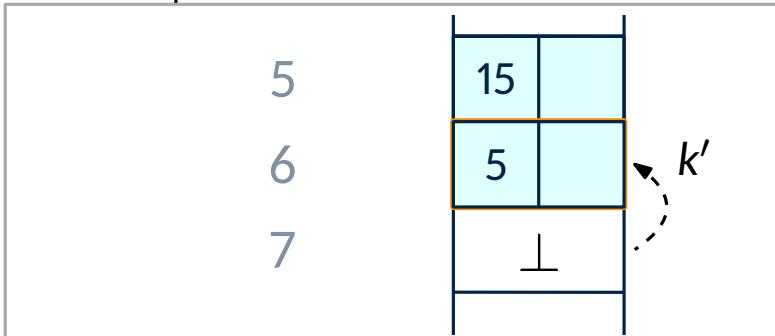
- |   |   |  |
|---|---|--|
| 1. wir verbieten <code>remove</code><br>→ inkrementelles Wörterbuch | 2. gelöschte Elemente <b>markieren</b><br>wir speichern zu jedem Element,<br>ob es als gelöscht markiert ist<br><br><b>Nachteil:</b><br>gelöschte Elemente beanspruchen Platz!<br><br>Ggf. sollte Tabelle reorganisiert werden,<br>wenn es zu viele gelöschte Elemente gibt | 3. Invariante reparieren <ul style="list-style-type: none"><li>· wir löschen das Element</li><li>· Invariante für <math>k' \neq k</math> verletzt:<br/>verschiebe das Element ins "Loch"</li><li>→ u.U. wiederholtes Verschieben</li></ul> |
|---|---|--|

- `insert(k, v):`

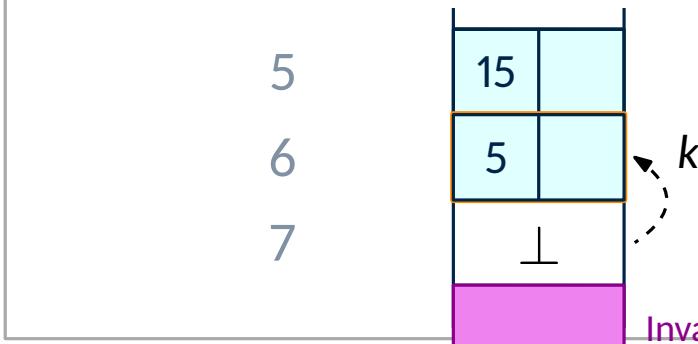
# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp:$  und  $T[j]$  ist nicht als gelöscht markiert
    - wenn  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
    - $j = (j + 1) \bmod m$gib  $\perp$  zurück
  - `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :  
Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$   
"Wraparound"  $(\bmod m)$  beachten
  - **Möglicher Umgang mit dem Problem:**
    - 1. wir verbieten `remove`  
→ inkrementelles Wörterbuch
    - 2. gelöschte Elemente **markieren**  
wir speichern zu jedem Element, ob es als gelöscht markiert ist
    - 3. Invariante reparieren
      - wir löschen das Element
      - Invariante für  $k' \neq k$  verletzt:  
verschiebe das Element ins "Loch"  
→ u.U. wiederholtes Verschieben
  - `insert(k, v):`
- 19 - 38
- 

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp$ :  
    | wenn  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück  
    |  $j = (j + 1) \bmod m$   
    gib  $\perp$  zurück
  - `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
  
Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :  
    Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$   
        "Wraparound"  $(\bmod m)$  beachten
  - **Möglicher Umgang mit dem Problem:**
    - 1. wir verbieten `remove`  
    → inkrementelles Wörterbuch
    - 2. gelöschte Elemente **markieren**  
    wir speichern zu jedem Element,  
    ob es als gelöscht markiert ist
    - 3. Invariante reparieren
      - wir löschen das Element
      - Invariante für  $k' \neq k$  verletzt:  
        verschiebe das Element ins "Loch"  
    → u.U. wiederholtes Verschieben
  - `insert(k, v):`
- 19 - 39
- 

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp:$  und  $T[j]$  ist nicht als gelöscht markiert
    - wenn  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
    - $j = (j + 1) \bmod m$gib  $\perp$  zurück
  - `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :  
Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$   
"Wraparound"  $(\bmod m)$  beachten
  - **Möglicher Umgang mit dem Problem:**
    - 1. wir verbieten `remove`  
→ inkrementelles Wörterbuch
    - 2. gelöschte Elemente **markieren**  
wir speichern zu jedem Element, ob es als gelöscht markiert ist
    - 3. Invariante reparieren
      - wir löschen das Element
      - Invariante für  $k' \neq k$  verletzt:  
verschiebe das Element ins "Loch"  
→ u.U. wiederholtes Verschieben
  - `insert(k, v):`
- 
- Invariante weiter unten verletzt?
- 19 - 40

# Hashing mit Linearem Sondieren

- `find(k):` Setze  $j := h(k)$   
**Solange**  $T[j] \neq \perp:$  und  $T[j]$  ist nicht als gelöscht markiert
  - | wenn  $\text{key}(T[j]) = k$ , gib  $T[j]$  zurück
  - |  $j = (j + 1) \bmod m$
  - gib  $\perp$  zurück
- `remove(k):` **Frage:** Können wir einfach das Element  $T[j]$  mit  $\text{key}(T[j]) = k$  entfernen?  
Nein, wir benötigen die folgende **Invariante** für jeden vorhandenen Schlüssel  $k$ :  
Sei  $j^*$  so, dass  $\text{key}(T[j^*]) = k$ . Dann ist  $T[j] \neq \perp$  für alle  $j$  zwischen  $h(k)$  und  $j^*$   
"Wraparound"  $(\bmod m)$  beachten
- **Möglicher Umgang mit dem Problem:**

1. wir verbieten <code>remove</code> → inkrementelles Wörterbuch	2. gelöschte Elemente <b>markieren</b> wir speichern zu jedem Element, ob es als gelöscht markiert ist <b>Nachteil:</b> gelöschte Elemente beanspruchen Platz!  Ggf. sollte Tabelle reorganisiert werden, wenn es zu viele gelöschte Elemente gibt	3. Invariante reparieren <ul style="list-style-type: none"><li>· wir löschen das Element</li><li>· Invariante für <math>k' \neq k</math> verletzt: verschiebe das Element ins "Loch" → u.U. wiederholtes Verschieben</li></ul>
---	---	--
- `insert(k, v):` ähnlich zu `find`, Details von `remove` abhängig.

# Hashing mit Linearem Sondieren: Laufzeit

---

Um eine Menge der Größe  $n$  zu speichern, benötigen wir eine Hashtabelle  $T[0 \dots m - 1]$  mit  $m \geq n$

Worst-Case-Laufzeit von `find`, `insert`, `remove` ist  $\Theta(n)$

Mit geeigneten Familien von Hashfunktionen sind die erwarteten Zugriffszeiten beweisbar schneller

Allerdings benötigt die Analyse des Linearen Sondierens stärkere Eigenschaften als c-Universalität

# Weitere $c$ -universelle Hashfamilien

---

Wir konstruieren eine weitere 1-universelle Hashfamilie  $\mathcal{H}$  mit Funktionen

$$h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}. \quad U \geq m$$

**Annahme:** Wir haben eine Primzahl  $p \geq U$

## Definition

Für jede Wahl von  $a \in \{1, \dots, p - 1\}, b \in \{0, \dots, p - 1\}$ , definieren wir

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

## Theorem.

Die Hashfamilie  $\mathcal{H}_{lin} = \{h_{a,b} \mid a \in \{1, \dots, p - 1\}, b \in \{0, \dots, p - 1\}\}$  ist 1-universell.

## Hinweis:

Die Hashfamilie  $\mathcal{H}_{lin2} = \{h_{a,0} \mid a \in \{1, \dots, p - 1\}\}$  ist 2-universell.

# Tabellenhashing

Wir konstruieren eine 1-universelle Hashfamilie  $\mathcal{H}$  mit Funktionen

$$h : \{0, \dots, U - 1\} \rightarrow \{0, \dots, m - 1\}.$$

**Annahme:**  $m = 2^a$  für ein  $a$

Wähle ein  $b < \log_2 U$ .

Wir zerteilen einen Schlüssel wieder in  $b$ -Bit-Zahlen ("Stücke")  
 $\rightarrow t = \lceil \frac{\log_2 U}{b} \rceil$  Stücke

Jede Zahl  $x \in \{0, \dots, U - 1\}$  lässt sich als

Folge von  $b$ -Bit-Zahlen  $(x_0, \dots, x_{t-1})$  schreiben:

$$x = \sum_{i=0}^{t-1} x_i 2^{b \cdot i}$$

Für eine Zahl  $x$  bezeichne  $\mathbf{x} = (x_0, \dots, x_{t-1})$  die Folge dieser Zahlen.

Wir erstellen Tabellen  $T_0, \dots, T_{t-1}$  von  $2^b$  uniform zufälligen  $a$ -Bit-Zahlen.

Für jede solche Wahl von Tabellen definieren wir

$$h_{\oplus(T_0, \dots, T_{t-1})}(k) = T_0[k_0] \oplus T_1[k_1] \oplus \dots \oplus T_{t-1}[k_{t-1}]$$

bitweises exklusives Oder

**Theorem.**

Die Hashfamilie  $\mathcal{H}_{Tab} = \{h_{\oplus(T_0, \dots, T_{t-1})} \mid T_0, \dots, T_{t-1}\}$  ist 1-universell.

Tabellenhashing hat sogar noch bessere Eigenschaften.

**Beispiel:**

$$U = 1024$$

$$m = 32 \rightarrow a = 5$$

$$b = 3$$

$$\rightarrow t = \lceil \frac{10}{3} \rceil = 4$$

$$\text{z.B. } x = 696$$

$$\begin{aligned} &= \underline{00} \underline{10} \underline{10} \underline{11} \underline{000}_2 \\ &= \quad 1 \quad 2 \quad 7 \quad 0_{2^3} \end{aligned}$$

$$\rightarrow \mathbf{x} = (0, 7, 2, 1)$$

$T_0$	$T_1$	$T_2$	$T_3$
00100	11101	01101	11100
10110	00110	11111	00101
11110	01010	00110	01111
01010	11011	11000	01101
11000	10010	11001	11011
01110	01011	11110	01110
00000	00100	11100	11000
01001	11010	01111	00101

$$h_{\oplus(T_0, T_1, T_2, T_3)}(696)$$

$$= T_0[0] \oplus T_1[7] \oplus T_2[2] \oplus T_3[1]$$

$$= 00100 \oplus 11010 \oplus 00110 \oplus 00101$$

$$= 11101 \rightsquigarrow h_{\oplus(T_0, T_1, T_2, T_3)}(696) = 29$$

# Zusammenfassung

---

## Balancierte Suchbäume

find  
insert  
remove

$O(\log n)$

Speicherbedarf

$O(n)$

minimum/maximum  
predecessor/successor

$O(\log n)$

Java

`java.util.TreeMap`

## Hashing mit Verkettung mit $O(1)$ -universeller Hashfamilie

erwartet  $O(1)$     randomisiert!

$O(n)$

$O(n)$

C++

`std::map`

`java.util.HashMap`

`std::unordered_map`

Algorithmen und Datenstrukturen SS'23

# Kapitel 12: Graphen (Grundbegriffe)

Marvin Künemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letzte Kapitel: wichtige Datenstrukturen zur Verwaltung von Sequenzen und Mengen  
Jetzt: Algorithmen auf Graphen

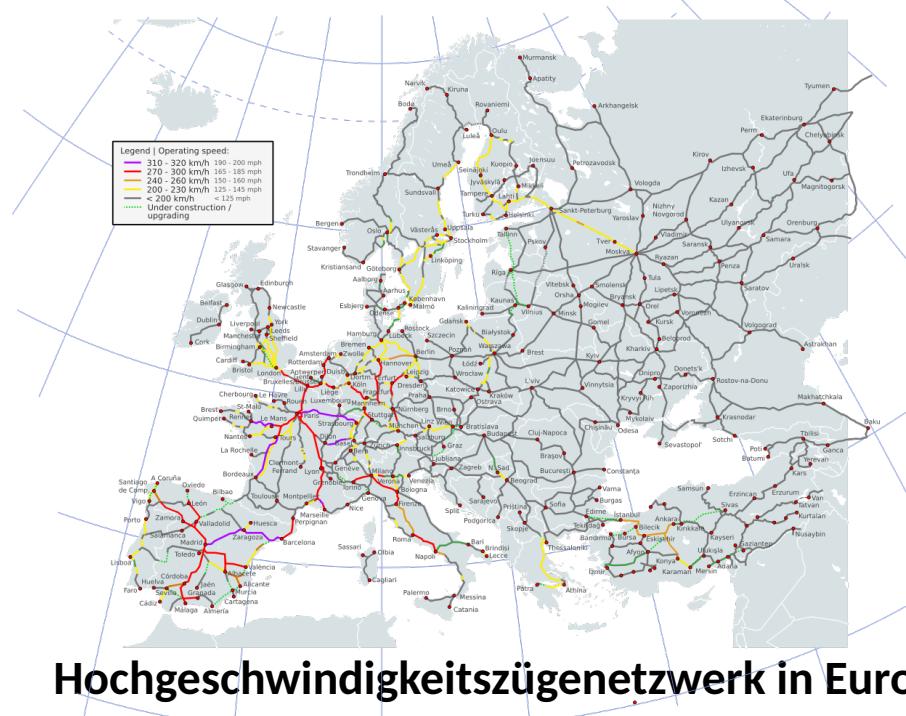
Insbesondere klären wir in diesem Kapitel:

- grundlegende Definition
- Repräsentationen von Graphen
- Traversierungen von Graphen

# Graphen: Motivation

In vielen Anwendungen wollen wir  
Objekte und ihre Verbindungen beschreiben  
→ Graphen

Beispiele:

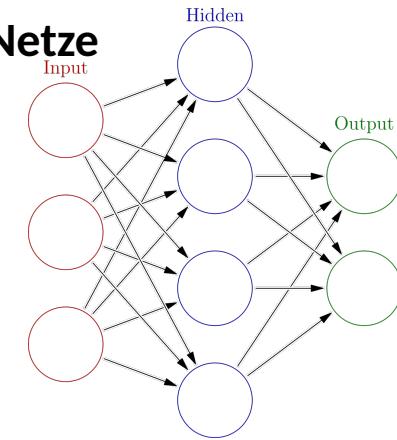


Hochgeschwindigkeitszügenetzwerk in Europa

By Original PNG : User:Bernese media, User:BIL2011 SVG version: User:Akwa and others,  
CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=49950794>

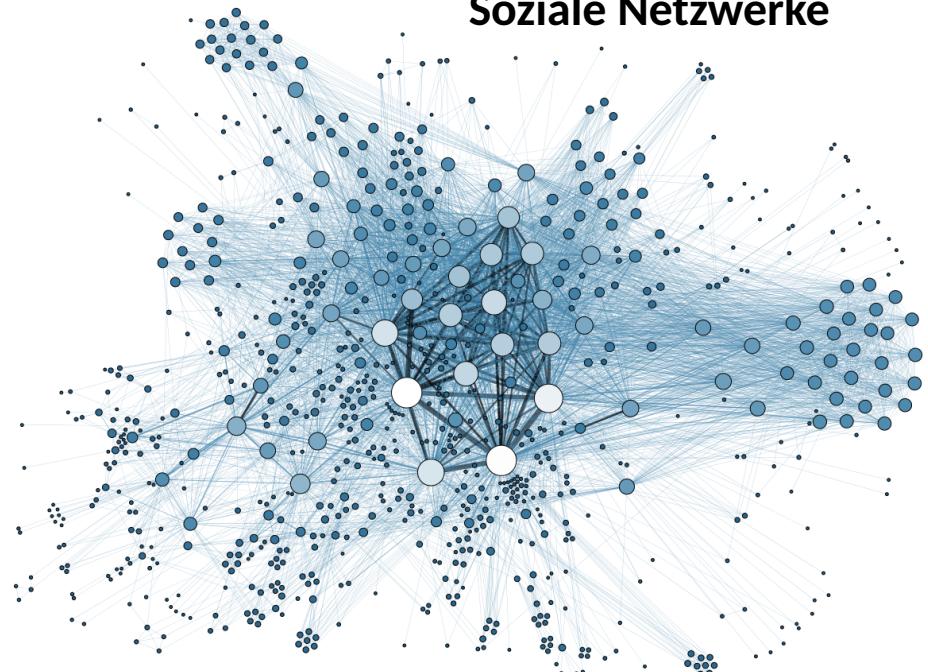
Unzählige algorithmische Aufgaben auf Graphen:  
Routing, Mustererkennung, Planung, etc.

## Neuronale Netze



By Glosser.ca - Own work, Derivative of File:Artificial neural network.svg,  
CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=24913461>

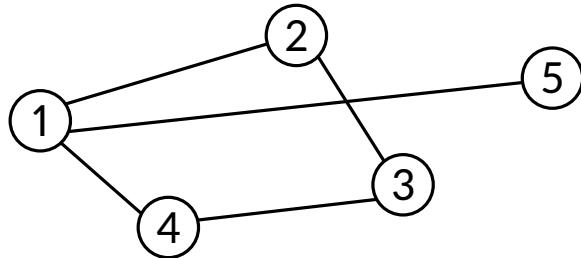
## Soziale Netzwerke



By Martin Grandjean - Grandjean, Martin (2014). "La connaissance est un réseau".  
Les Cahiers du Numérique 10 (3): 37-54. DOI:10.3166/LCN.10.3.37-54.,  
CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=29364647>

# **Was sind Graphen?**

# Ungerichtete Graphen: Definition



$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{\{1, 2\}, \{1, 5\}, \{1, 4\}, \{2, 3\}, \{4, 3\}\}$$

## Definition

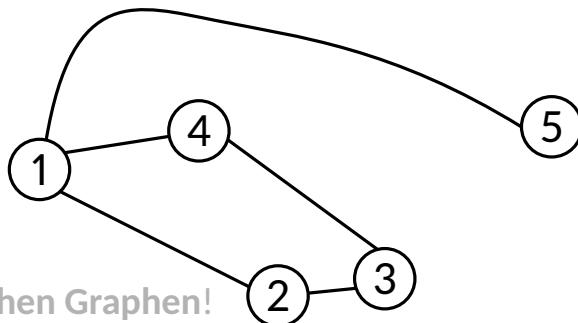
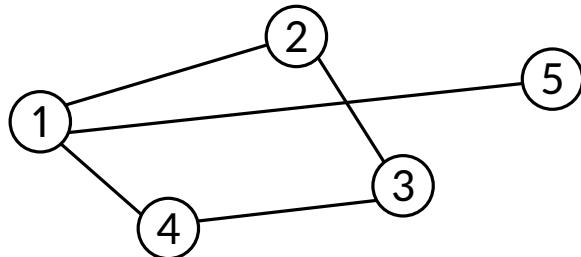
Ein ungerichteter Graph  $G$  ist ein Paar  $(V, E)$ , wobei:

- $V$  eine endliche Menge ist und
- $E$  eine Menge 2-elementiger Teilmengen von  $V$  ist.

~~> Knoten (vertices/nodes)  
~~> Kanten (edges)

$e = \{u, v\} \in E$  bedeutet:  $u$  und  $v$  sind durch Kante  $e$  verbunden;  $u$  und  $v$  sind **adjazent** (Nachbarn)  
 $u$  und  $e$  sind **inzident** (symmetrisch:  $v$  und  $e$  sind inzident)

# Ungerichtete Graphen: Definition



unterschiedliche Darstellung des gleichen Graphen!

$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{\{1, 2\}, \{1, 5\}, \{1, 4\}, \{2, 3\}, \{4, 3\}\}$$

## Definition

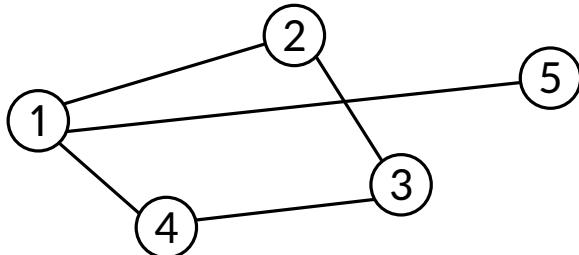
Ein ungerichteter Graph  $G$  ist ein Paar  $(V, E)$ , wobei:

- $V$  eine endliche Menge ist und
- $E$  eine Menge 2-elementiger Teilmengen von  $V$  ist.

~~> Knoten (vertices/nodes)  
~~> Kanten (edges)

$e = \{u, v\} \in E$  bedeutet:  $u$  und  $v$  sind durch Kante  $e$  verbunden;  $u$  und  $v$  sind **adjazent** (Nachbarn)  
 $u$  und  $e$  sind **inzident** (symmetrisch:  $v$  und  $e$  sind inzident)

# Ungerichtete Graphen: Definition



$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{\{1, 2\}, \{1, 5\}, \{1, 4\}, \{2, 3\}, \{4, 3\}\}$$

## Definition

Ein ungerichteter Graph  $G$  ist ein Paar  $(V, E)$ , wobei:

- $V$  eine endliche Menge ist und  $\rightsquigarrow$  Knoten (vertices/nodes)
- $E$  eine Menge 2-elementiger Teilmengen von  $V$  ist.  $\rightsquigarrow$  Kanten (edges)

$e = \{u, v\} \in E$  bedeutet:  $u$  und  $v$  sind durch Kante  $e$  verbunden;  $u$  und  $v$  sind **adjazent** (Nachbarn)  
 $u$  und  $e$  sind **inzident** (symmetrisch:  $v$  und  $e$  sind inzident)

Wir schreiben gelegentlich  $\binom{V}{2}$  für die Menge 2-elementiger Teilmengen von  $V$

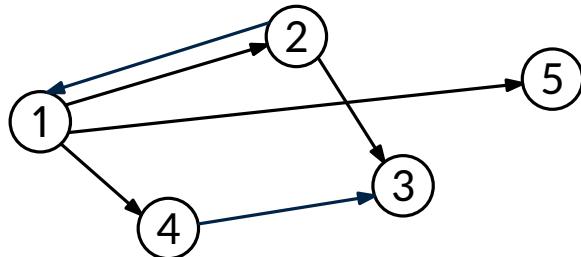
Beispiel:  $\binom{\{v_1, v_2, v_3, v_4\}}{2} = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$

Dann können wir die zweite Bedingung schreiben als  $E \subseteq \binom{V}{2}$

**Konvention:** Wenn nicht anders angegeben, sei  $n = |V|$  und  $m = |E|$

In der Regel benutzen wir  $V = \{1, \dots, n\}$

# Gerichtete Graphen: Definition



$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{(1, 2), (1, 5), (1, 4), (2, 1), (2, 3), (4, 3)\}$$

## Definition

Ein gerichteter Graph  $G$  ist ein Paar  $(V, E)$ , wobei:

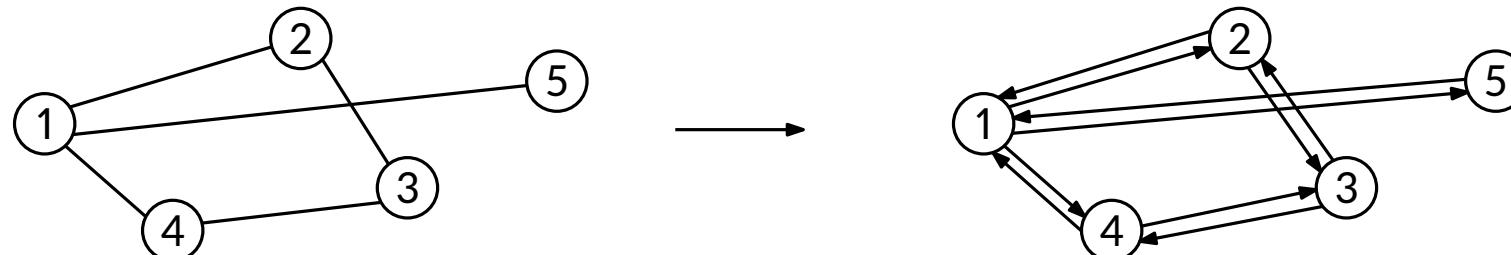
- $V$  eine endliche Menge ist und
- $E$  eine Menge geordneter Paare von  $V$  ist.

↔ Knoten (vertices/nodes)  
↔ Kanten (edges)

$e = (u, v) \in E$  bedeutet: die Kante  $e$  geht von  $u$  nach  $v$ ;  $u$  und  $v$  sind adjazent  
 $u$  und  $e$  sind inzident;  $v$  und  $e$  sind inzident

Wir können die zweite Bedingung auch schreiben als  $E \subseteq V \times V$

Wir können einen ungerichteten Graphen als gerichteten Graphen darstellen:



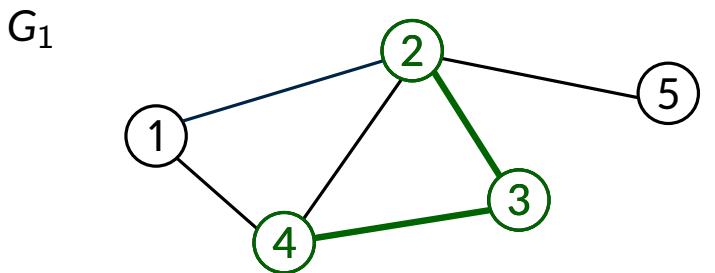
# Wege und Pfade

## Definition

Wir nennen eine Sequenz  $v_0, \dots, v_k$  mit  $k \geq 0$  einen **Weg** (walk) von  $v_0$  nach  $v_k$  in  $G = (V, E)$ , wenn:

- $v_i \in V$  für alle  $0 \leq i \leq k$ , sowie
- wenn  $G$  ungerichtet ist:  
 $\{v_i, v_{i+1}\} \in E$  für alle  $0 \leq i < k$

2, 3, 4 ist ein Weg in  $G_1$



# Wege und Pfade

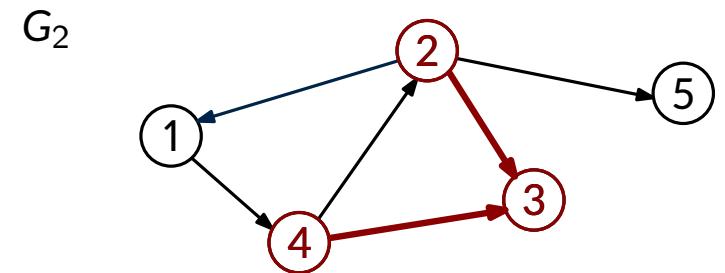
## Definition

Wir nennen eine Sequenz  $v_0, \dots, v_k$  mit  $k \geq 0$  einen **Weg** (walk) von  $v_0$  nach  $v_k$  in  $G = (V, E)$ , wenn:

- $v_i \in V$  für alle  $0 \leq i \leq k$ , sowie
- wenn  $G$  ungerichtet ist:  
 $\{v_i, v_{i+1}\} \in E$  für alle  $0 \leq i < k$

- wenn  $G$  gerichtet ist:  
 $(v_i, v_{i+1}) \in E$  für alle  $0 \leq i < k$   
"alle Kanten gehen in die gleiche Richtung"

2, 3, 4 ist kein Weg in  $G_2$



# Wege und Pfade

## Definition

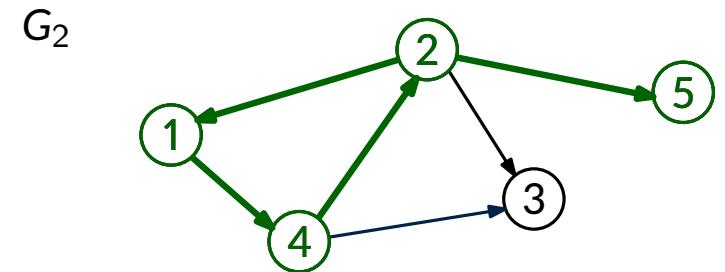
Wir nennen eine Sequenz  $v_0, \dots, v_k$  mit  $k \geq 0$  einen **Weg** (walk) von  $v_0$  nach  $v_k$  in  $G = (V, E)$ , wenn:

- $v_i \in V$  für alle  $0 \leq i \leq k$ , sowie
- wenn  $G$  ungerichtet ist:  
 $\{v_i, v_{i+1}\} \in E$  für alle  $0 \leq i < k$

- wenn  $G$  gerichtet ist:  
 $(v_i, v_{i+1}) \in E$  für alle  $0 \leq i < k$   
"alle Kanten gehen in die gleiche Richtung"

2, 3, 4 ist kein Weg in  $G_2$

2, 1, 4, 2, 5 ist ein Weg in  $G_2$



# Wege und Pfade

## Definition

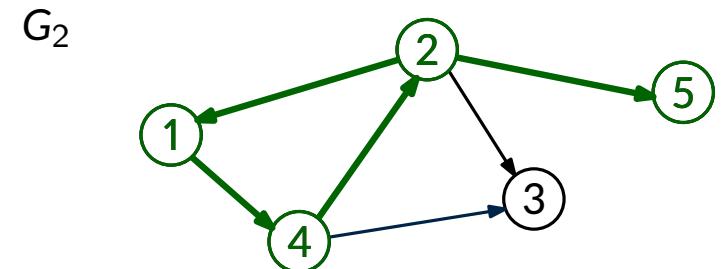
Wir nennen eine Sequenz  $v_0, \dots, v_k$  mit  $k \geq 0$  einen **Weg** (walk) von  $v_0$  nach  $v_k$  in  $G = (V, E)$ , wenn:

- $v_i \in V$  für alle  $0 \leq i \leq k$ , sowie
- wenn  $G$  ungerichtet ist:  
 $\{v_i, v_{i+1}\} \in E$  für alle  $0 \leq i < k$

- wenn  $G$  gerichtet ist:  
 $(v_i, v_{i+1}) \in E$  für alle  $0 \leq i < k$   
"alle Kanten gehen in die gleiche Richtung"

2, 3, 4 ist kein Weg in  $G_2$

2, 1, 4, 2, 5 ist ein Weg in  $G_2$ , aber kein Pfad



Wir nennen  $k$  die **Länge** des Weges

Ein Weg  $v_0, \dots, v_k$  ist **einfach**, auch genannt **Pfad** (path), wenn gilt:

alle Knoten  $v_i$  sind **paarweise verschieden**, bis auf möglicherweise Anfangs- und Endknoten

Wenn  $v_i = v_j$  für  $0 \leq i, j \leq k$ , dann gilt  $i = j$  oder  $\{i, j\} = \{0, k\}$

# Zyklus

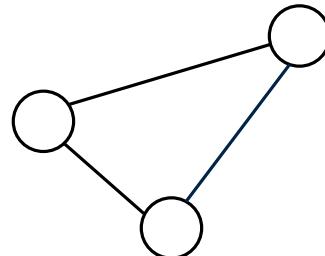
---

Ein Weg  $v_0, \dots, v_k$  heißt **geschlossen**, wenn  $v_0 = v_k$

## Definition

engl: cycle

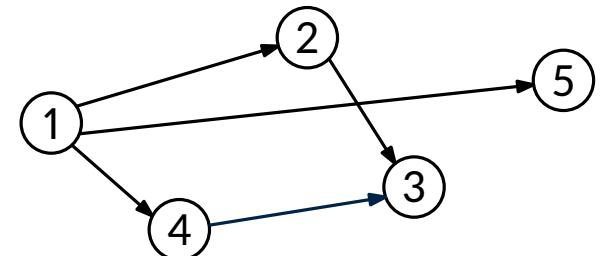
Wenn  $G$  ungerichtet ist, ist ein **Zyklus** (Kreis) ein geschlossener Pfad der Länge  $k \geq 3$ .



Wenn  $G$  gerichtet ist, ist ein **Zyklus** (Kreis) ein geschlossener Pfad der Länge  $k \geq 2$ .



Ein Graph  $G$  heißt **azyklisch**, wenn er keinen Zyklus enthält.



# Erreichbarkeit: ungerichtete Graphen

Sei  $G$  ein gegebener Graph.

Ein Knoten  $v$  heißt **erreichbar** von  $u$ , wenn es einen Weg von  $u$  nach  $v$  in  $G$  gibt.

Wir schreiben:  $u \rightsquigarrow_G v$  Wenn der Graph  $G$  klar ist, schreiben wir gelegentlich nur  $u \rightsquigarrow v$

**Bemerkung:** Wenn es einen Weg von  $u$  nach  $v$  gibt, dann gibt es auch einen Pfad von  $u$  nach  $v$ .

↗ Übung

Es folgt:  $v$  erreichbar von  $u \Leftrightarrow$  es gibt einen Pfad von  $u$  nach  $v$ .

**Fakt:** In einem **ungerichteten** Graphen  $G$  ist  $\rightsquigarrow_G$  eine **Äquivalenzrelation**.

D. h. für alle  $u, v, w \in V$  gilt:

- $v \rightsquigarrow_G v$
- wenn  $u \rightsquigarrow_G v$ , dann  $v \rightsquigarrow_G u$
- wenn  $u \rightsquigarrow_G v$  und  $v \rightsquigarrow_G w$ , dann  $u \rightsquigarrow_G w$

**Reflexivität**  
**Symmetrie**  
**Transitivität**

**Beweis:** ↗ Übung

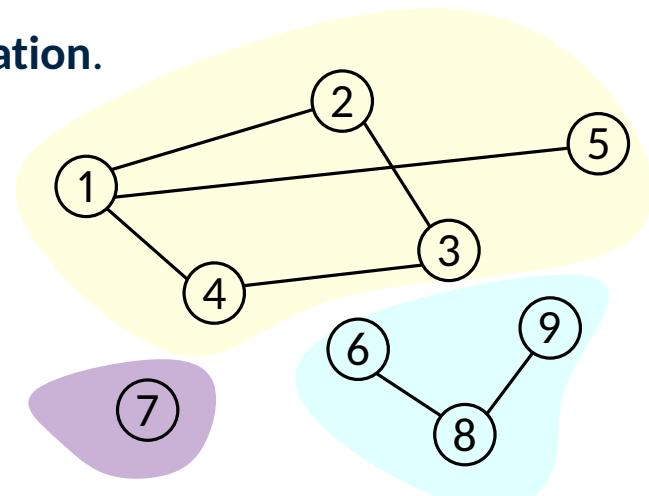
## Definition

Jede Äquivalenzklasse der Erreichbarkeitsrelation  $\rightsquigarrow_G$  heißt **Zusammenhangskomponente**.

Insbesondere ist die **Zusammenhangskomponente von  $u$**  die Menge  $\{v \in V \mid u \rightsquigarrow_G v\}$

Ein Graph  $G$  heißt **zusammenhängend**, wenn es nur eine Zusammenhangskomponente gibt.

d.h.  $u \rightsquigarrow_G v$  für alle  $u, v \in V$



# Erreichbarkeit: ungerichtete Graphen

Sei  $G$  ein gegebener Graph.

Ein Knoten  $v$  heißt **erreichbar** von  $u$ , wenn es einen Weg von  $u$  nach  $v$  in  $G$  gibt.

Wir schreiben:  $u \rightsquigarrow_G v$  Wenn der Graph  $G$  klar ist, schreiben wir gelegentlich nur  $u \rightsquigarrow v$

**Bemerkung:** Wenn es einen Weg von  $u$  nach  $v$  gibt, dann gibt es auch einen Pfad von  $u$  nach  $v$ .

↗ Übung

Es folgt:  $v$  erreichbar von  $u \Leftrightarrow$  es gibt einen Pfad von  $u$  nach  $v$ .

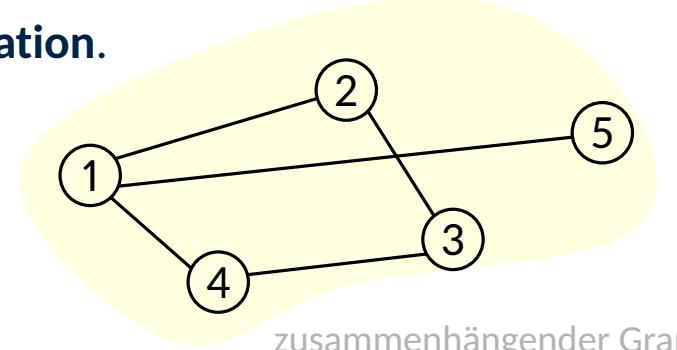
**Fakt:** In einem **ungerichteten** Graphen  $G$  ist  $\rightsquigarrow_G$  eine **Äquivalenzrelation**.

D. h. für alle  $u, v, w \in V$  gilt:

- $v \rightsquigarrow_G v$
- wenn  $u \rightsquigarrow_G v$ , dann  $v \rightsquigarrow_G u$
- wenn  $u \rightsquigarrow_G v$  und  $v \rightsquigarrow_G w$ , dann  $u \rightsquigarrow_G w$

**Reflexivität**  
**Symmetrie**  
**Transitivität**

**Beweis:** ↗ Übung



## Definition

Jede Äquivalenzklasse der Erreichbarkeitsrelation  $\rightsquigarrow_G$  heißt **Zusammenhangskomponente**.

Insbesondere ist die **Zusammenhangskomponente von  $u$**  die Menge  $\{v \in V \mid u \rightsquigarrow_G v\}$

Ein Graph  $G$  heißt **zusammenhängend**, wenn es nur eine Zusammenhangskomponente gibt.

d.h.  $u \rightsquigarrow_G v$  für alle  $u, v \in V$

# Erreichbarkeit: gerichtete Graphen

---

Sei  $G$  ein gegebener Graph.

Ein Knoten  $v$  heißt **erreichbar** von  $u$ , wenn es einen Weg von  $u$  nach  $v$  in  $G$  gibt.

Wir schreiben:  $u \rightsquigarrow_G v$  Wenn der Graph  $G$  klar ist, schreiben wir gelegentlich nur  $u \rightsquigarrow v$

**Bemerkung:** Wenn es einen Weg von  $u$  nach  $v$  gibt, dann gibt es auch einen Pfad von  $u$  nach  $v$ .

↗ Übung

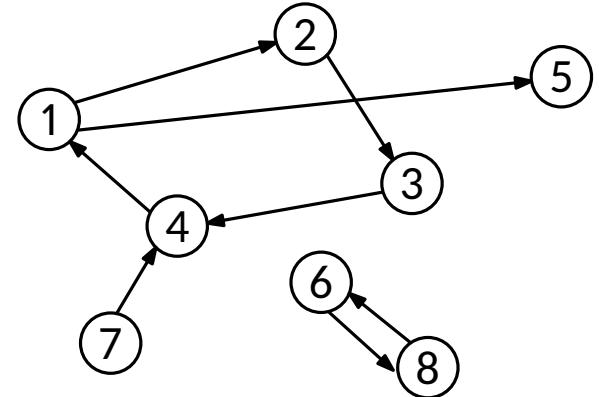
Es folgt:  $u$  erreichbar von  $v \Leftrightarrow$  es gibt einen Pfad von  $u$  nach  $v$

---

**Achtung:** In einem **gerichteten Graphen** ist  $\rightsquigarrow_G$  keine Äquivalenzrelation!

Beispiel: 5 ist erreichbar von 4, aber nicht umgekehrt!

⇒ Es gibt verschiedene Definitionen von Zusammenhang in gerichteten Graphen:



# Erreichbarkeit: gerichtete Graphen

---

Sei  $G$  ein gegebener Graph.

Ein Knoten  $v$  heißt **erreichbar** von  $u$ , wenn es einen Weg von  $u$  nach  $v$  in  $G$  gibt.

Wir schreiben:  $u \rightsquigarrow_G v$  Wenn der Graph  $G$  klar ist, schreiben wir gelegentlich nur  $u \rightsquigarrow v$

**Bemerkung:** Wenn es einen Weg von  $u$  nach  $v$  gibt, dann gibt es auch einen Pfad von  $u$  nach  $v$ .

↗ Übung

Es folgt:  $u$  erreichbar von  $v \Leftrightarrow$  es gibt einen Pfad von  $u$  nach  $v$

---

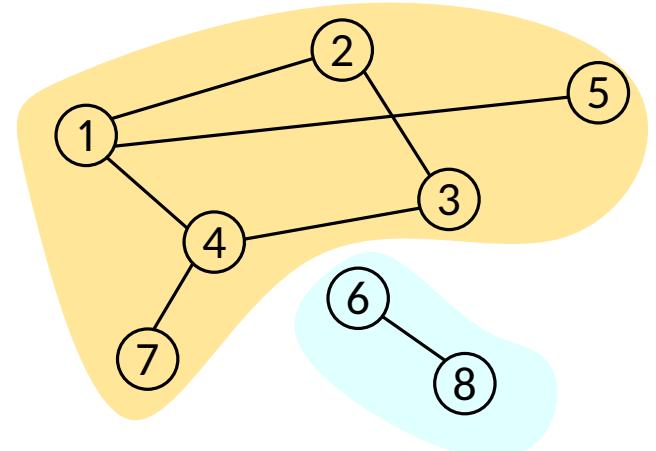
**Achtung:** In einem **gerichteten Graphen** ist  $\rightsquigarrow_G$  keine Äquivalenzrelation!

Beispiel: 5 ist erreichbar von 4, aber nicht umgekehrt!

⇒ Es gibt verschiedene Definitionen von Zusammenhang in gerichteten Graphen:

**(Schwacher) Zusammenhang:**

wir vergessen die Richtung der Kanten von  $G$



# Erreichbarkeit: gerichtete Graphen

Sei  $G$  ein gegebener Graph.

Ein Knoten  $v$  heißt **erreichbar** von  $u$ , wenn es einen Weg von  $u$  nach  $v$  in  $G$  gibt.

Wir schreiben:  $u \rightsquigarrow_G v$  Wenn der Graph  $G$  klar ist, schreiben wir gelegentlich nur  $u \rightsquigarrow v$

**Bemerkung:** Wenn es einen Weg von  $u$  nach  $v$  gibt, dann gibt es auch einen Pfad von  $u$  nach  $v$ .

↗ Übung

Es folgt:  $u$  erreichbar von  $v \Leftrightarrow$  es gibt einen Pfad von  $u$  nach  $v$

**Achtung:** In einem **gerichteten Graphen** ist  $\rightsquigarrow_G$  keine Äquivalenzrelation!

Beispiel: 5 ist erreichbar von 4, aber nicht umgekehrt!

⇒ Es gibt verschiedene Definitionen von Zusammenhang in gerichteten Graphen:

**(Schwacher) Zusammenhang:**

wir vergessen die Richtung der Kanten von  $G$

**Starker Zusammenhang:**

↗ Übung

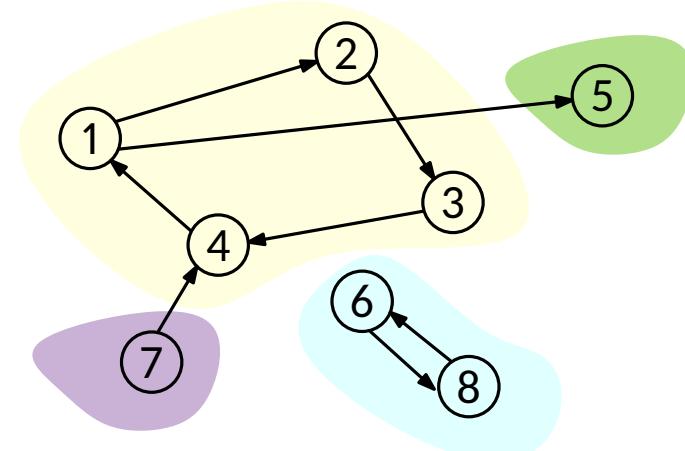
wir definieren eine neue Äquivalenzrelation

$u \rightsquigarrow_G v$  genau dann, wenn  $u \rightsquigarrow_G v$  und  $v \rightsquigarrow_G u$

**Definition**

Jede Äquivalenzklasse der Relation  $\rightsquigarrow_G$  heißt **starke Zusammenhangskomponente**.

Die **starke Zusammenhangskomponente** von  $u$  ist die Menge  $\{v \in V \mid u \rightsquigarrow_G v \text{ und } v \rightsquigarrow_G u\}$



# Erreichbarkeit: gerichtete Graphen

Sei  $G$  ein gegebener Graph.

Ein Knoten  $v$  heißt **erreichbar** von  $u$ , wenn es einen Weg von  $u$  nach  $v$  in  $G$  gibt.

Wir schreiben:  $u \rightsquigarrow_G v$  Wenn der Graph  $G$  klar ist, schreiben wir gelegentlich nur  $u \rightsquigarrow v$

**Bemerkung:** Wenn es einen Weg von  $u$  nach  $v$  gibt, dann gibt es auch einen Pfad von  $u$  nach  $v$ .

↗ Übung

Es folgt:  $u$  erreichbar von  $v \Leftrightarrow$  es gibt einen Pfad von  $u$  nach  $v$

**Achtung:** In einem **gerichteten Graphen** ist  $\rightsquigarrow_G$  keine Äquivalenzrelation!

Beispiel: 5 ist erreichbar von 4, aber nicht umgekehrt!

⇒ Es gibt verschiedene Definitionen von Zusammenhang in gerichteten Graphen:

**(Schwacher) Zusammenhang:**

wir vergessen die Richtung der Kanten von  $G$

**Starker Zusammenhang:**

↗ Übung

wir definieren eine neue Äquivalenzrelation

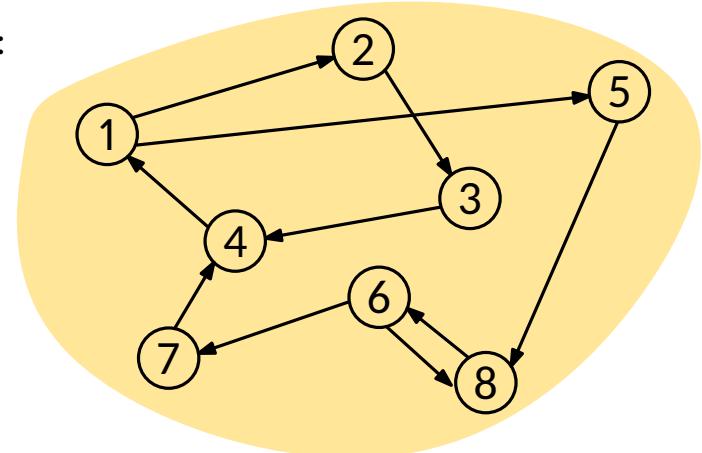
$u \rightsquigarrow_G v$  genau dann, wenn  $u \rightsquigarrow_G v$  und  $v \rightsquigarrow_G u$

**Definition**

Jede Äquivalenzklasse der Relation  $\rightsquigarrow_G$  heißt **starke Zusammenhangskomponente**.

Die **starke Zusammenhangskomponente** von  $u$  ist die Menge  $\{v \in V \mid u \rightsquigarrow_G v \text{ und } v \rightsquigarrow_G u\}$

Ein Graph  $G$  heißt **stark zusammenhängend**, wenn es nur eine starke Zusammenhangskomponente gibt.



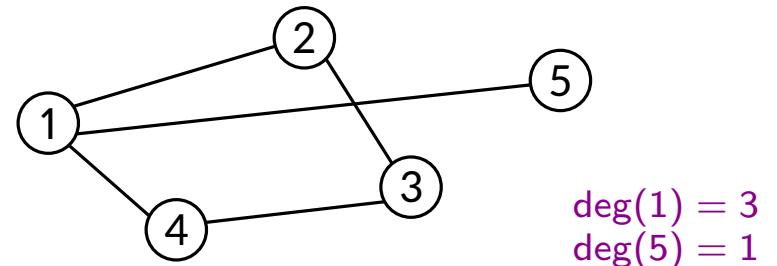
stark zusammenhängend

# Grad eines Knotens

## Definition

In einem ungerichteten Graphen G ist der **Grad**  $\deg(u)$  eines Knoten  $u$  die Anzahl zu  $u$  inzidenter Kanten:

$$\deg(u) = |\{v \in V \mid \{u, v\} \in E\}|$$



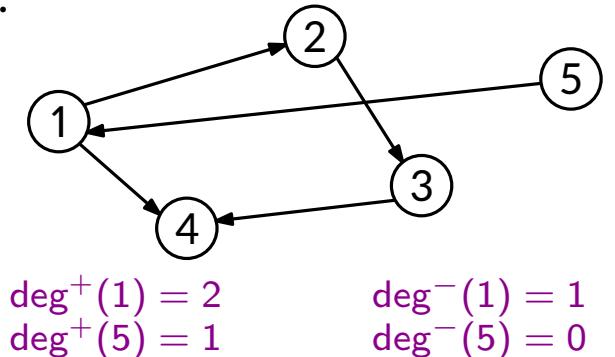
In einem gerichteten Graphen G unterscheiden wir zwischen **Eingangs-** und **Ausgangsgrad**:

Der **Ausgangsgrad**  $\deg^+(u)$  ist die Anzahl von  $u$  ausgehender Kanten:

$$\deg^+(u) = |\{v \in V \mid (u, v) \in E\}|$$

Der **Eingangsgrad**  $\deg^-(u)$  ist die Anzahl in  $u$  eingehender Kanten:

$$\deg^-(u) = |\{v \in V \mid (v, u) \in E\}|$$



## Fakt

In einem ungerichteten Graphen G gilt:

- $2m = \sum_{v \in V} \deg(v)$
- $m \leq \binom{n}{2} \leq n^2$

In einem gerichteten Graphen G gilt:

- $m = \sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v)$
- $m \leq n^2$

Wir nennen einen Graphen "dicht besetzt", wenn  $m = \Theta(n^2)$

Wir nennen einen Graphen "dünn besetzt", wenn  $m = O(n)$

Achtung: "dünn besetzt" wird manchmal allgemeiner definiert.

# Einschränkungen und Erweiterungen

---

**Schleifen** sind Kanten der Form  $(v, v)$



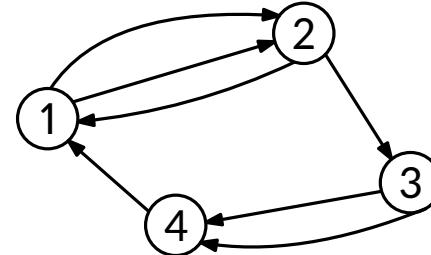
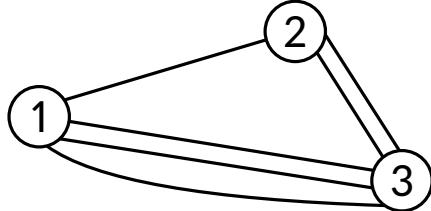
In der Regel betrachten wir Graphen ohne Schleifen (**schleifenloser Graph**)

Gelegentlich betrachtet man allgemeinere Formen von Graphen, z.B.:

- ungerichtete Graphen, die Schleifen enthalten dürfen



- **Multigraphen**: Graphen, in denen Kanten mehrfach auftauchen dürfen:



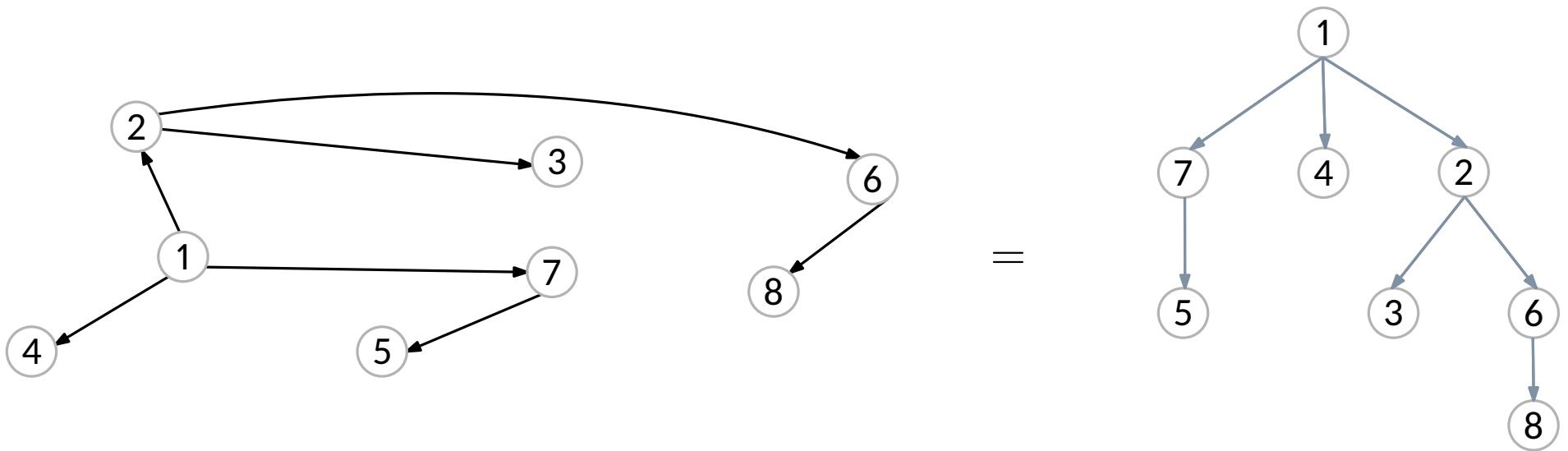
# Spezialfall: Bäume

Definition (gerichteter Baum).

$(u, v) \in E \Leftrightarrow u$  ist Elternknoten von  $v$

Ein gewurzelter gerichteter **Baum**  $T$  ist ein gerichteter Graph  $(V, E)$  mit folgenden Bedingungen:

- es gibt keine Schleifen, d.h.  $(v, v) \notin E$  für alle  $v \in V$
- es gibt genau einen Knoten  $r \in V$  mit  $\deg^-(r) = 0$      $r$  ist **Wurzel** von  $T$
- für alle  $u \in V \setminus \{r\}$  gilt  $\deg^-(u) = 1$                         alle Knoten außer  $r$  haben genau einen Elter
- für alle  $u \in V$  gilt  $r \rightsquigarrow_T u$                                 es gibt einen Wurzel- $u$ -Pfad für alle Knoten  $u$



Definition (ungerichteter Baum).

Ein ungerichteter Baum  $T$  ist ein ungerichteter, azyklischer, zusammenhängender Graph.

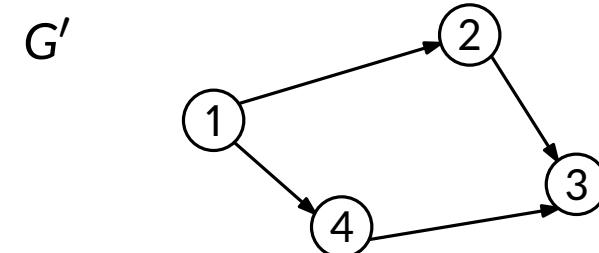
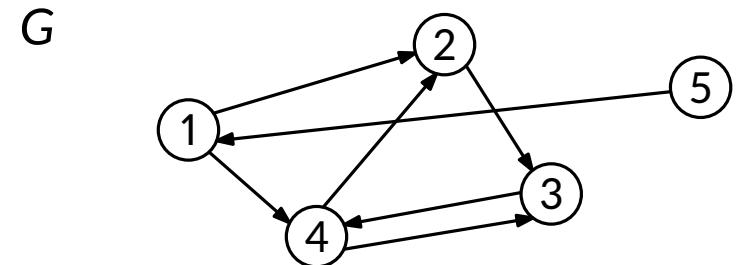
# Teilgraph

---

## Definition.

$G' = (V', E')$  ist **Teilgraph** von  $G = (V, E)$ , wenn

- $V' \subseteq V$
- $E' \subseteq E$



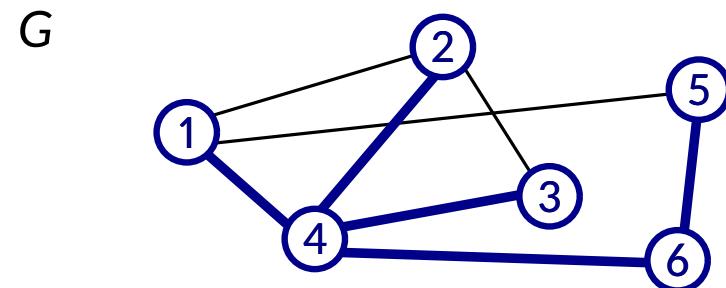
## Definition (Spannbaum).

Sei G ein ungerichteter Graph.

Ein Baum  $T = (V', E')$  sodass:

- $T$  ist Teilgraph von G und
- $V' = V$

heißt **Spannbaum** von G.



# **Wie repräsentieren wir Graphen?**

# Anforderungen

---

Was wollen wir mit Graphen tun?

Zu den möglichen, interessanten Grundoperationen gehören:

**Kantenabfragen** - Sind zwei Knoten  $u$  und  $v$  adjazent?

**Navigation** - Welche Kanten sind inzident zu Knoten  $u$ ?

Häufig wollen wir bestimmte Wege durch einen Graphen bestimmen.

Wichtige Operation: gib alle ausgehenden (oder eingehenden) Kanten von  $u$  aus

**Zugriff auf Zusatzinformation** - Welche Zusatzinformation haben wir zu Knoten  $u$  oder Kante  $e$ ?

Häufig wollen wir neben Informationen zu den Knoten und/oder Kanten abspeichern

Beispiel: jede Kante hat ein Gewicht, jeder Knoten eine Beschriftung, etc.

**Knoten oder Kanten einfügen/löschen**

Gelegentlich kann sich ein Graph verändern.

Beispiel: füge Kante von  $u$  nach  $v$  ein, oder lösche Knoten  $w$  und alle inzidenten Kanten

# Überblick

Es gibt verschiedene Möglichkeiten, Graphen im Computer darzustellen:  
Wir besprechen:

1. Unsortierte Kantenfolge
2. Adjazenzlisten (bzw. -arrays)
3. Adjazenzmatrix

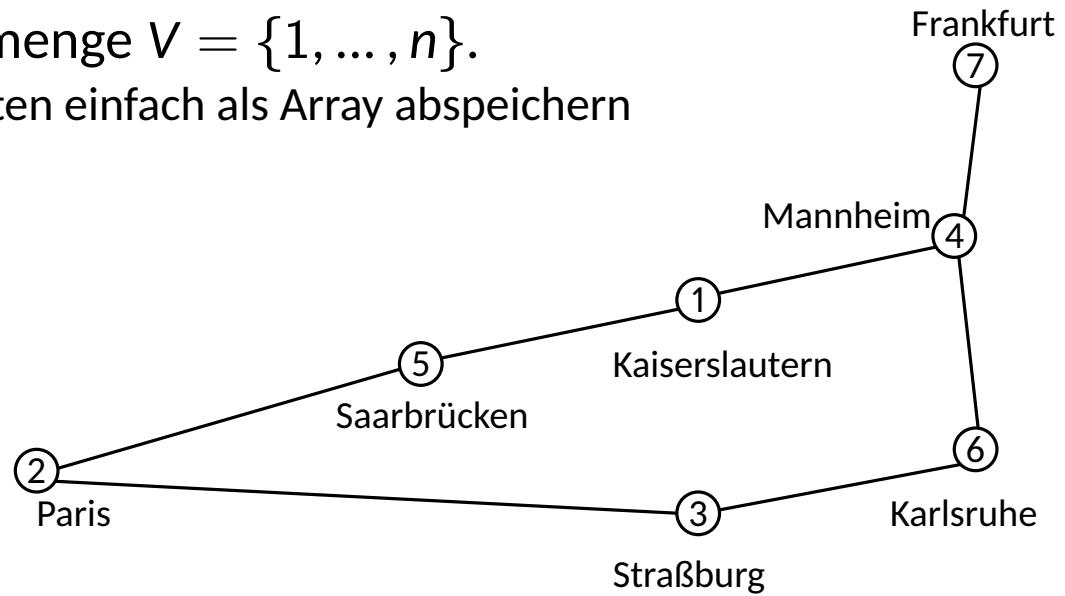
Jede Darstellung hat verschiedene **Vor- und Nachteile!**

## Konvention:

Typischerweise benutzen wir als Knotenmenge  $V = \{1, \dots, n\}$ .

⇒ wir können Zusatzinformationen zu den Knoten einfach als Array abspeichern

Name[1..7]  
Kaiserslautern  
Paris  
Straßburg  
Mannheim  
Saarbrücken  
Karlsruhe  
Frankfurt



**Frage:** Welche Darstellung eignet sich, um zu einem gegebenen Ortsnamen die Knotennummer schnell zu bestimmen?

# 1. Möglichkeit: Unsortierte Kantenfolge

## Einfachste Möglichkeit:

Wir speichern eine Folge aller Kanten als Array oder Liste

$$(u_1, v_1), \dots, (u_m, v_m)$$

**Speicherbedarf:**  $\Theta(m)$  Speicherzellen

2m Speicherzellen, wenn Array genutzt wird

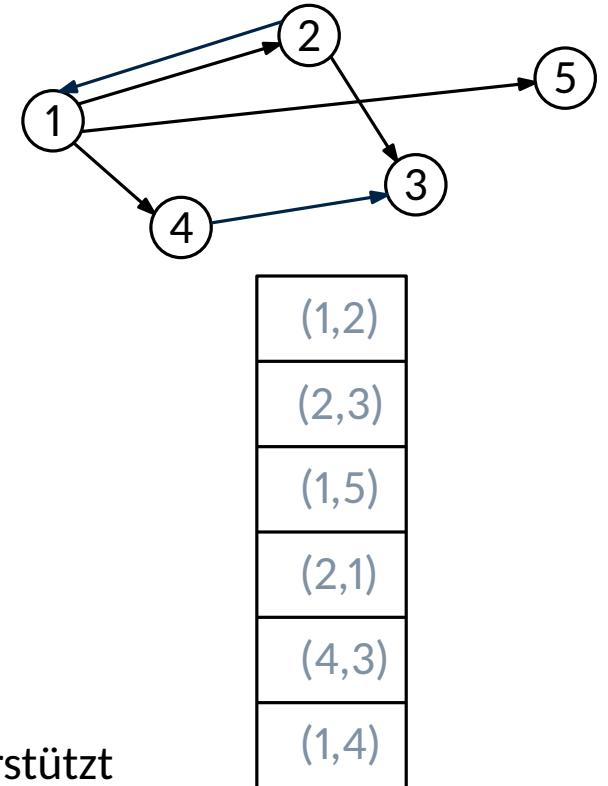
### Vorteile:

- simple Repräsentation
- kleiner Speicherbedarf
- einfach um zusätzliche Kanten erweiterbar

### Nachteile:

- selbst viele einfache Operationen werden nicht schnell unterstützt

**Beispiel:** Abfrage, ob  $u$  adjazent zu  $v$  ist, benötigt  $\Theta(m)$  Zeit



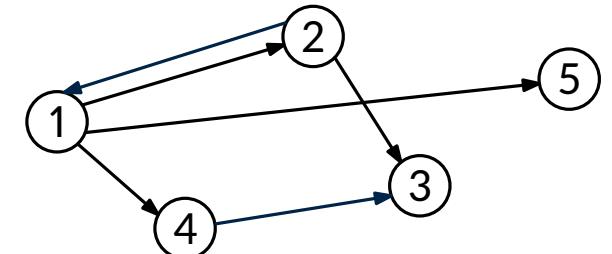
Wird in der Regel vor der Verwendung in andere Repräsentation überführt

## 2. Möglichkeit: Adjazenzlisten (bzw. -arrays)

Für jeden Knoten  $u$  speichern wir eine Liste  $L_u$  (bzw. Array  $A_u$ )

Für jede Kante  $(u, v)$  fügen wir  $v$  in  $L_u$  (bzw.  $A_u$ ) ein.

$\{u, v\}$  in ungerichteten Graphen



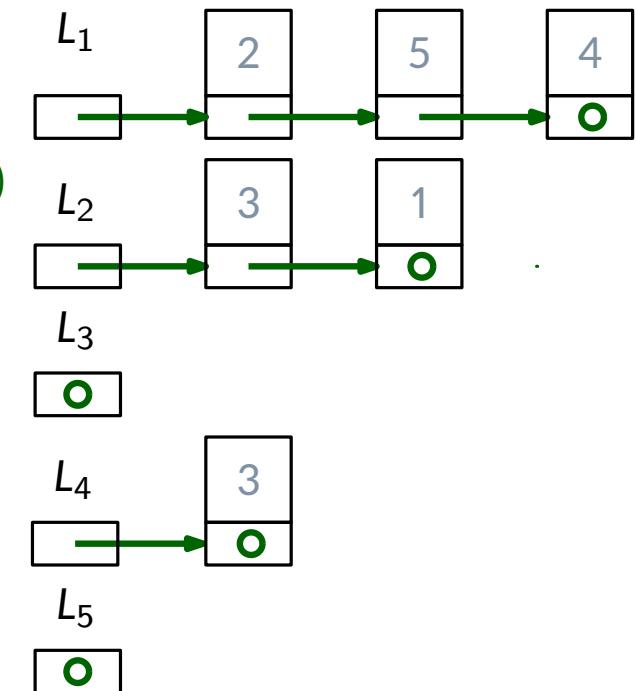
**Speicherbedarf:**  $\Theta(n + m)$  Speicherzellen

$n + m$  Speicherzellen, wenn Arrays verwendet werden

Wir können schnell alle von  $u$  ausgehenden Kanten ausgeben  $O(\deg^+(u))$

Frage: Können wir auch schnell die eingehenden Kanten bestimmen?

Existenz einer Kante von  $u$  nach  $v$  kann in  $O(\deg^+(u))$  beantwortet werden



**Vorteile:**

- sehr kleiner Speicherbedarf
- erlaubt gute Navigation im Graphen

**Nachteile:**

- Abfrage, Einfügen oder Löschen einer gegebenen Kante  $(u, v)$  benötigt  $\Theta(\deg^+(u))$  Zeit im Worst-Case

Frage: kann das vermieden werden?

Adjazenzlisten eignen sich besonders für "dünn besetzte" Graphen  
19 - 15

# 3. Möglichkeit: Adjazenzmatrix

Wir speichern eine **Matrix** (zweidimensionales Array)  $A[1 \dots n, 1 \dots n]$

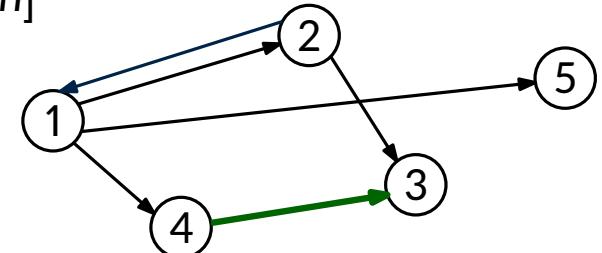
$$A[u, v] = \begin{cases} 1 & \text{wenn } (u, v) \in E \\ 0 & \text{ansonsten} \end{cases}$$

$\{u, v\} \in E$  (in ungerichteten Graphen)

**Speicherbedarf:**  $n^2$  Bits

Wir können in Zeit  $O(1)$  abfragen, ob  $u$  und  $v$  adjazent sind!

Wir können in Zeit  $O(n)$  alle ausgehenden Kanten von  $u$  bestimmen  
bzw. eingehenden



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

## Vorteile:

- Abfrage, Einfügen und Löschen einer Kante  $(u, v)$  läuft in Zeit  $O(1)$

## Nachteile:

- selbst für "dünn besetzte" Graphen benötigen wir  $\Theta(n^2)$  Speicherplatz
- das Bestimmen ausgehender Kanten benötigt  $\Omega(n)$  im Worst-Case, auch wenn  $\deg^+(u)$  klein ist

Adjazenzmatrizen eignen sich besonders für "dicht besetzte" Graphen

# Kanteninformationen

Gelegentlich wollen wir **Zusatzinformationen** zu den Kanten abspeichern

**Beispiele:** 1. Ein Graph repräsentiert ein Straßennetzwerk

→ jede Kante  $e$  repräsentiert einen Straßenabschnitt und hat eine Länge  $\ell(e)$

2. Ein Graph repräsentiert ein Kanalisationssystem

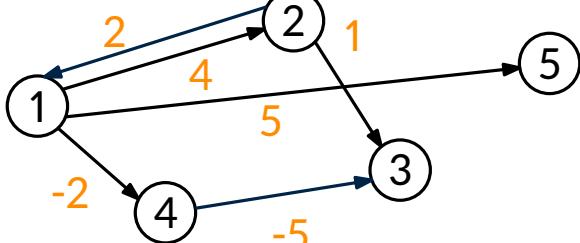
→ jede Kante  $e$  repräsentiert ein Rohr eines gewissen Durchmessers  $d(e)$

→ **Gewichteter Graph**  $G = (V, E, w)$ :

Ein Graph  $G = (V, E)$  mit zusätzlicher **Gewichtsfunktion** (Kantengewichtung)  $w : E \rightarrow \mathbb{Z}$

Alle besprochenen Darstellungen lassen sich gut zu Kantengewichtungen erweitern:

**Beispiele:**



(1,2,4)
(2,3,1)
(1,5,5)
(2,1,2)
(4,3,-5)
(1,4,-2)

$$A = \begin{pmatrix} 0 & 4 & 0 & -2 & 5 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

**Hinweis:**

Manchmal gibt man fehlenden Kanten andere Einträge als 0  
z.B.  $\infty$  oder  $-\infty$

# **Wie traversieren wir Graphen?**

# Traversierung eines Graphen

---

Einen Graph zu **traversieren** bedeutet, möglichst jeden Knoten eines Graphen zu besuchen.

Dabei wechseln wir von einem Knoten  $u$  zum nächsten Knoten  $v$  nur entlang einer Kante  $(u, v)$   
in ungerichteten Graphen:  $\{u, v\}$

## Abstraktes Grundprinzip:

Wir haben einen Startknoten  $s \in V$

Wir besuchen Knoten, die von  $s$  erreichbar sind, und entdecken dabei neue Knoten, die von  $s$  erreichbar sind.

Traversierungsalgorithmen unterscheiden sich in der Reihenfolge, in der "neu entdeckte" Knoten besucht werden

## Wichtigste Traversierungsalgorithmen:

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

## Hinweis:

Wir beschreiben BFS und DFS für **gerichtete Graphen**.

Man kann diese für ungerichtete Graphen  $G$  verwenden, in dem man jedes  $\{u, v\} \in E$  durch  $(u, v)$  und  $(v, u)$  ersetzt.  
→ dies liefert einen entsprechenden gerichteten Graphen.

# Breitensuche (BFS)

---

**Idee:** Wir besuchen Knoten, die von  $s$  erreichbar sind, "levelweise"

Der Startknoten bildet das Level 0

Die Nachbarn von  $s$ , also alle Knoten  $v$  mit  $(s, v) \in E$ , bilden das Level 1

Die Knoten des Level  $i \geq 2$  sind die Nachbarn von Level- $(i - 1)$ -Knoten, die nicht schon besucht sind.  
also nicht schon in einem niedrigeren Level sind

→ alle von  $s$  erreichbaren Knoten werden sich in einem Level finden

## Algorithmenbeschreibung:

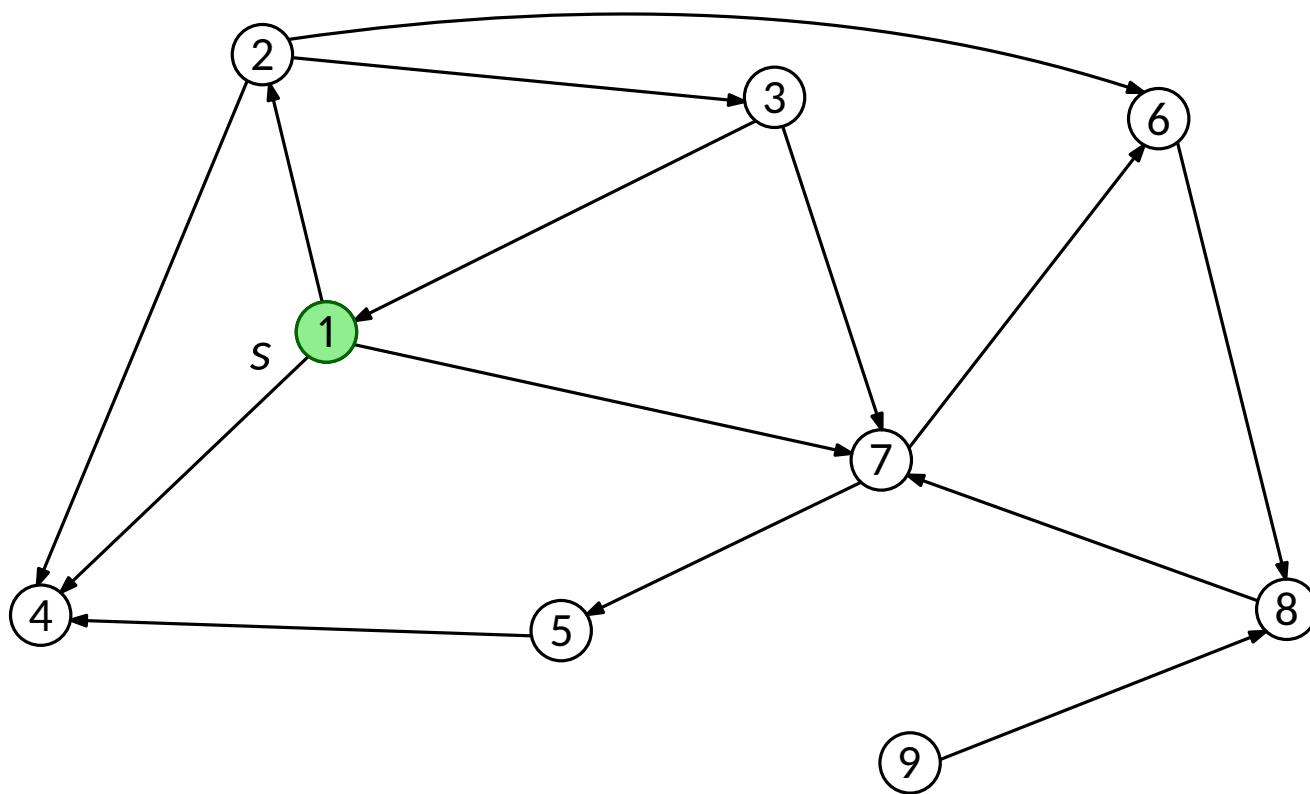
- Zu jedem Knoten speichern wir einen Status: **unentdeckt**, **entdeckt** oder **abgefertigt**
- anfangs ist nur  $s$  **entdeckt**, alle anderen Knoten sind **unentdeckt**
- wir speichern **entdeckte** Knoten in einer Queue  $Q$ , anfangs enthält sie nur  $s$
- **Solange**  $Q$  nicht leer ist, machen wir folgendes:
  - wir entnehmen einen Knoten  $u$  vom Anfang der Queue
  - **für** jeden **unentdeckten** Knoten  $v$  mit  $(u, v) \in E$  machen wir folgendes:
    - wir setzen  $v$ 's Status auf **entdeckt**
    - wir fügen  $v$  ans Ende der Queue  $Q$  ein
  - wir setzen  $u$ 's Status auf **abgefertigt**

→ Der Algorithmus endet, wenn  $Q$  leer ist. Alle **unentdeckten** Knoten sind von  $s$  nicht erreichbar.

# Beispiel für BFS

Startknoten:  $s = 1$

- unentdeckt
- entdeckt



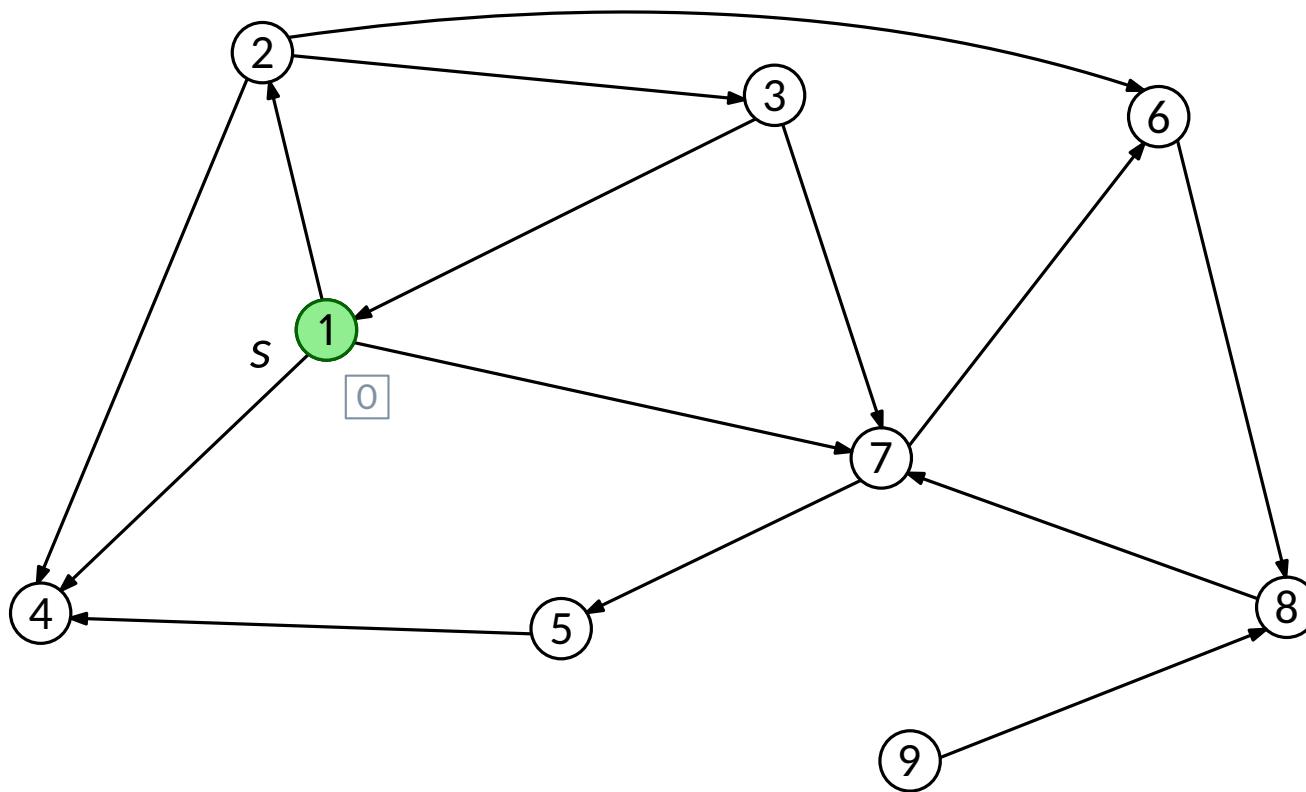
Queue Q:

# Beispiel für BFS

Startknoten:  $s = 1$

- unentdeckt (white circle)
- entdeckt (green circle)

$i$  Level  $i$

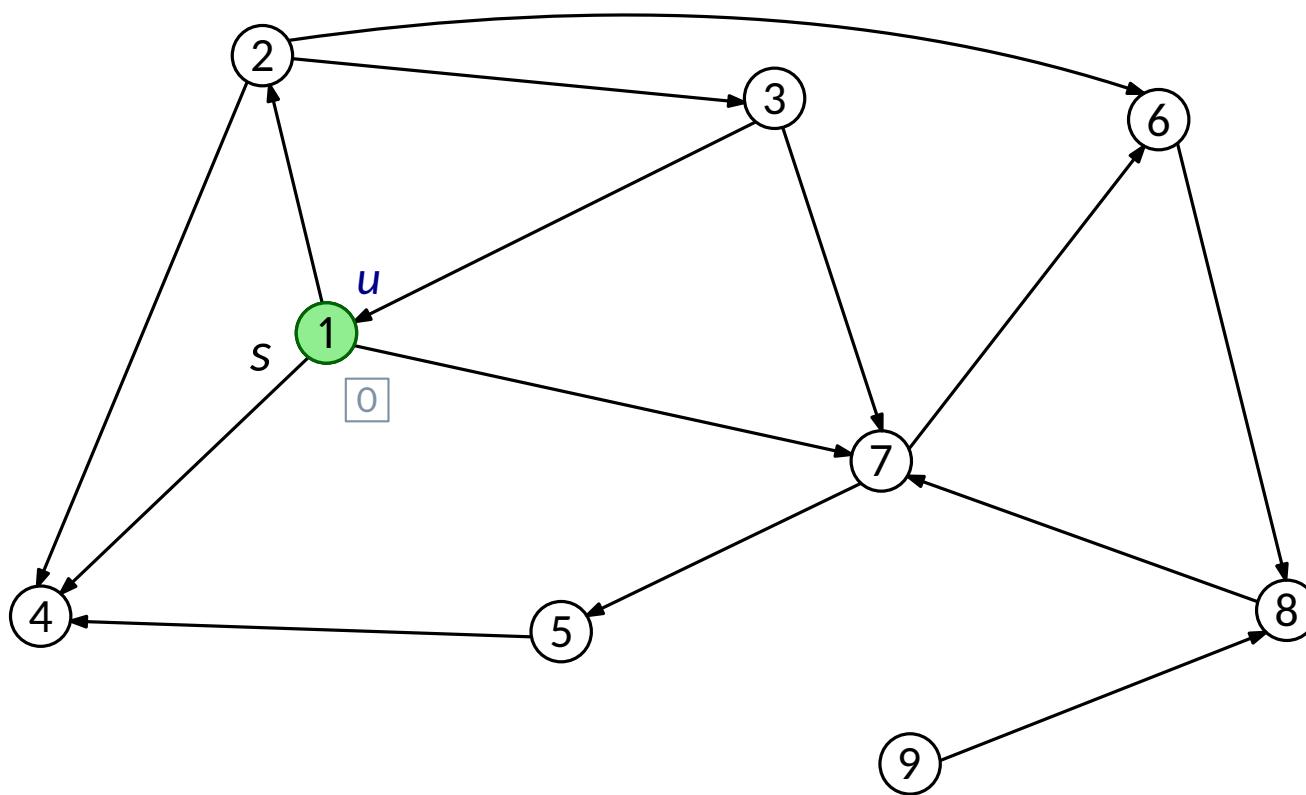


Queue Q:

1

# Beispiel für BFS

Startknoten:  $s = 1$



○ unentdeckt  
● entdeckt

□  $i$  Level  $i$

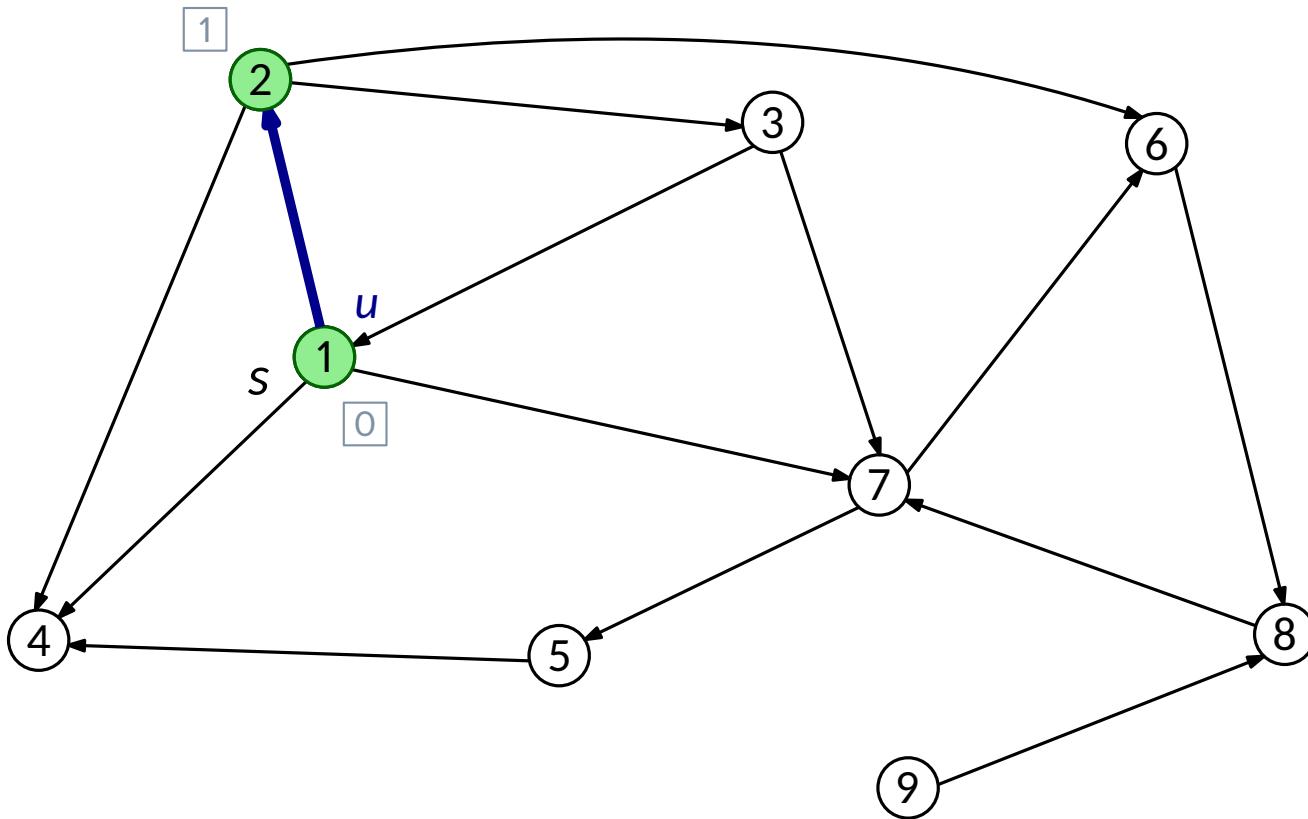
Queue Q:

# Beispiel für BFS

Startknoten:  $s = 1$

- unentdeckt
- entdeckt

$i$  Level  $i$



Queue Q:

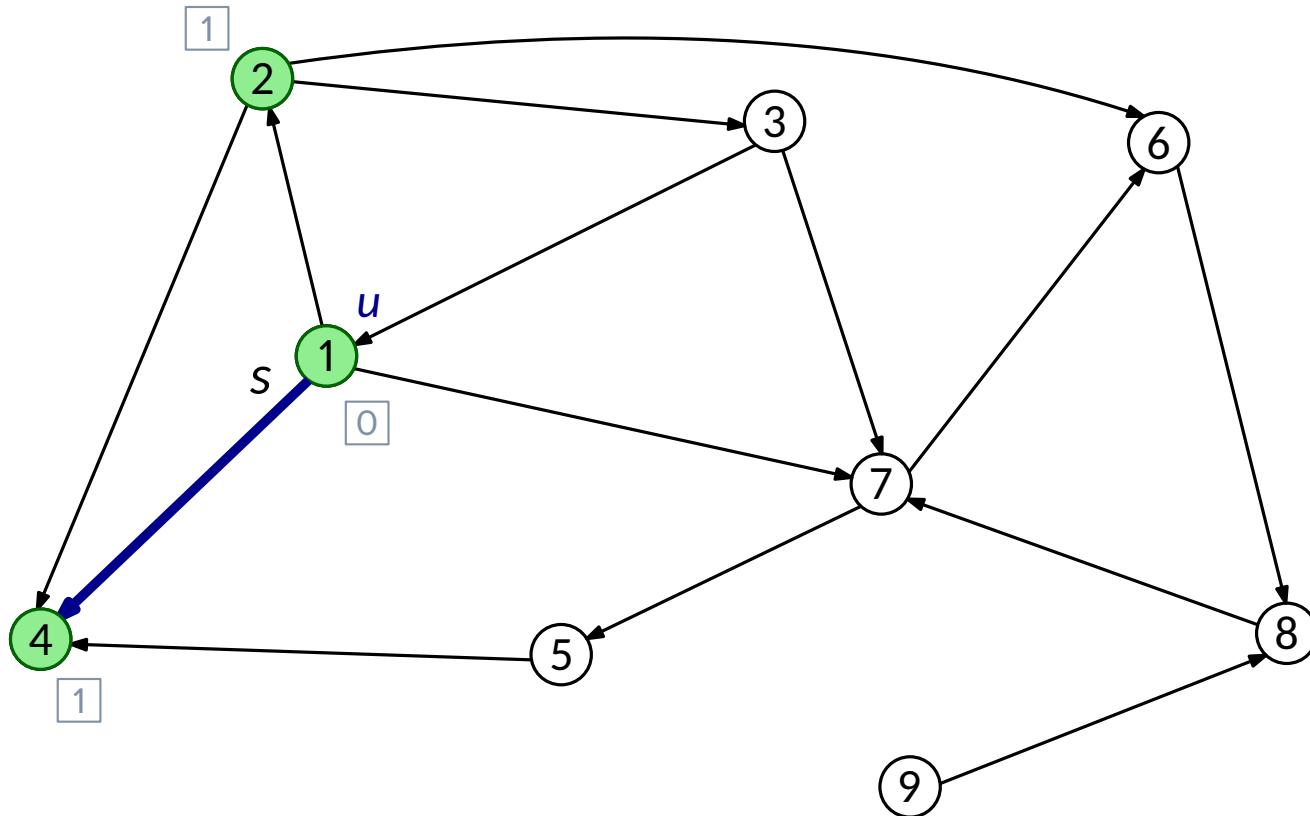
2

# Beispiel für BFS

Startknoten:  $s = 1$

- unentdeckt
- entdeckt

$i$  Level  $i$



Queue Q:

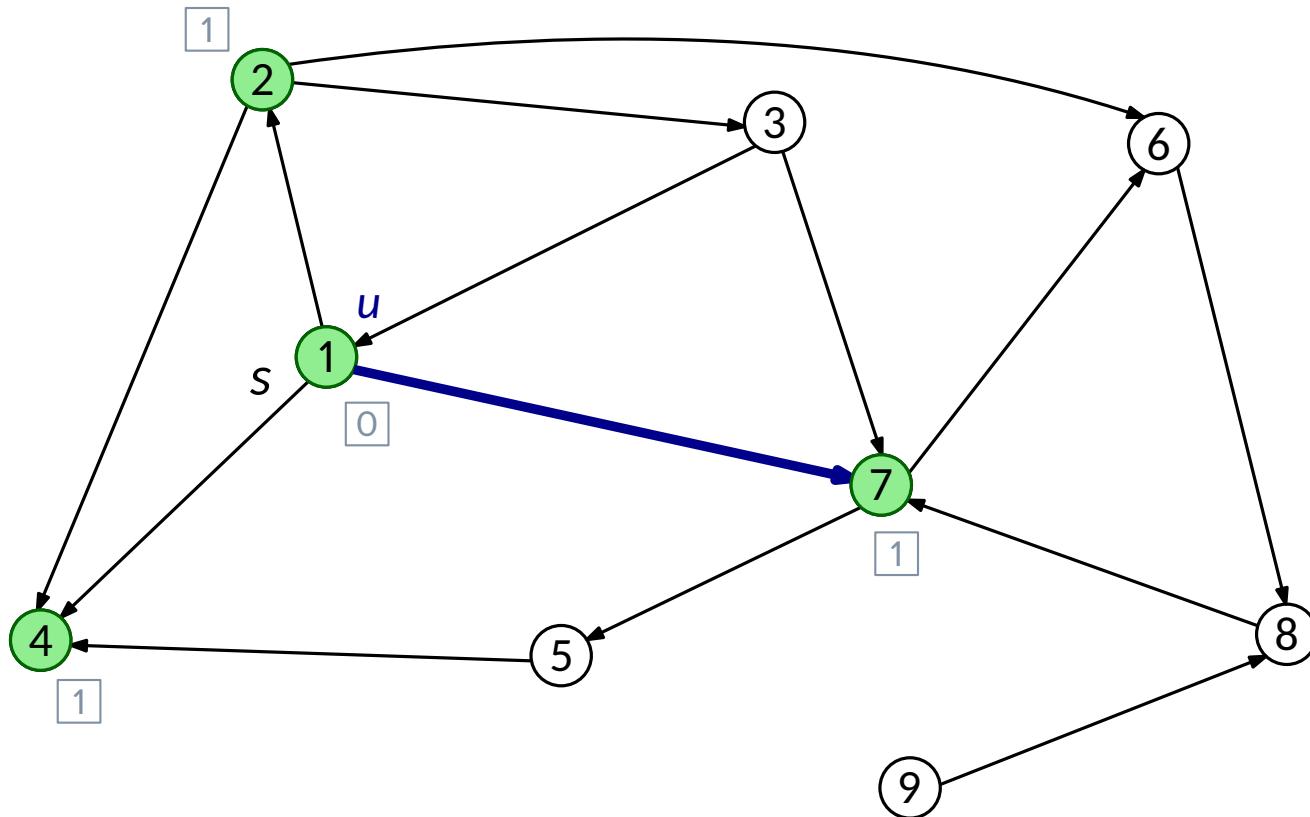
2 4

# Beispiel für BFS

Startknoten:  $s = 1$

- unentdeckt
- entdeckt

$i$  Level  $i$

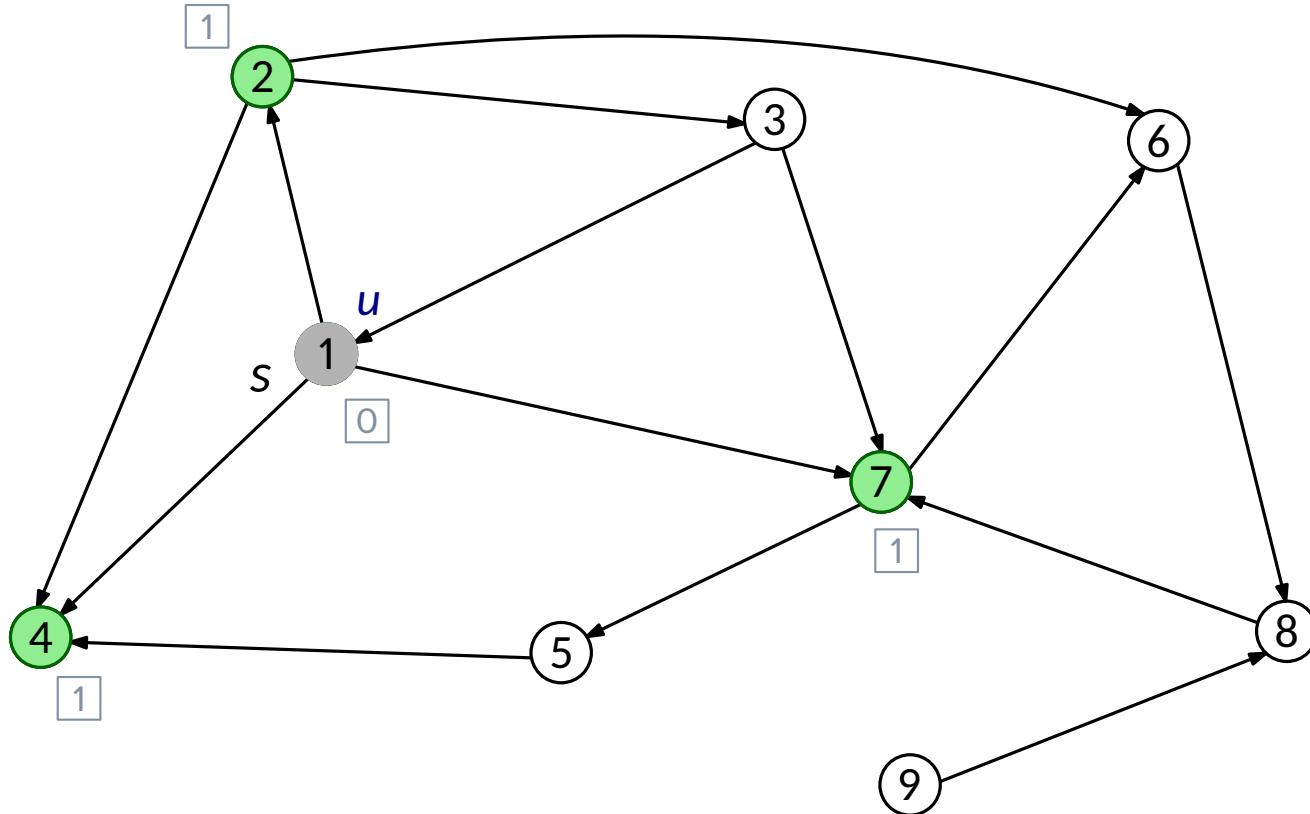


Queue Q:

2 4 7

# Beispiel für BFS

Startknoten:  $s = 1$



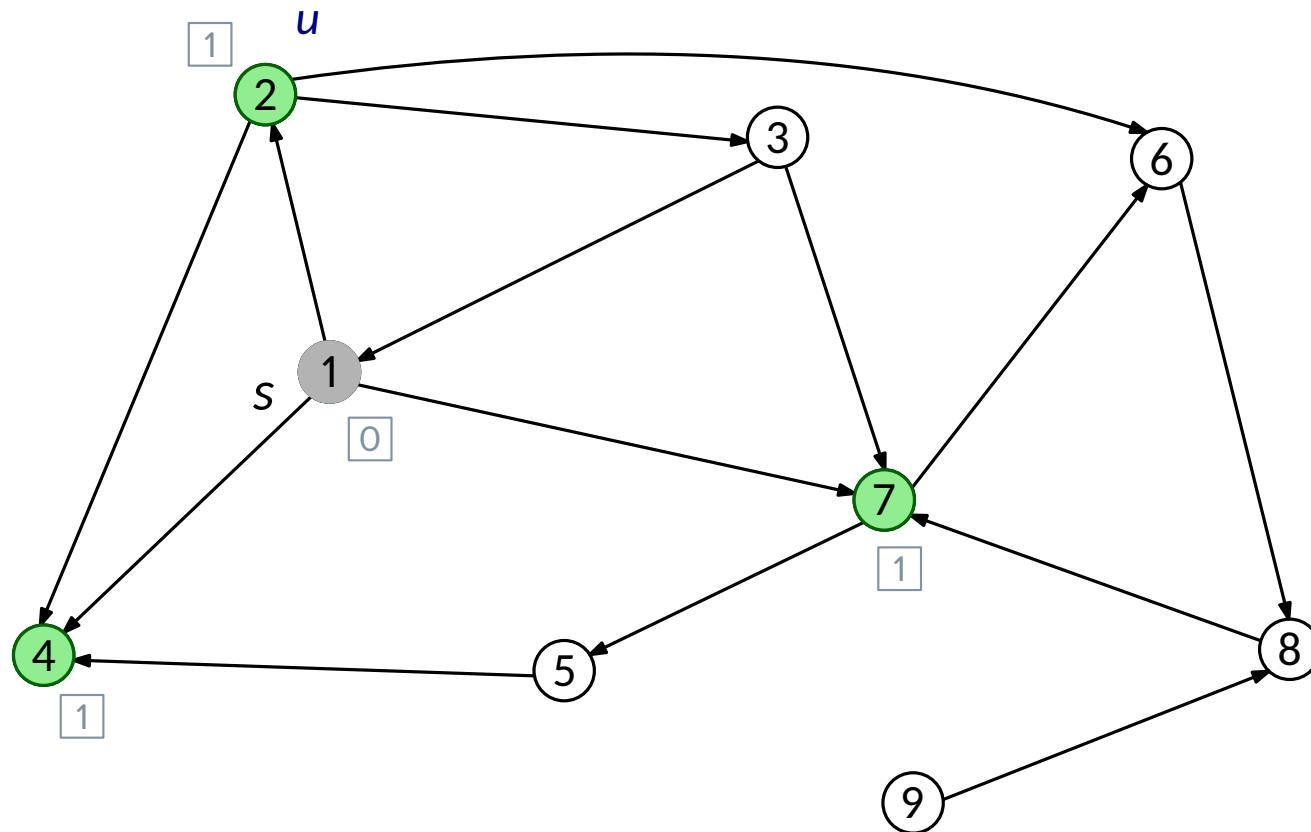
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

2 4 7

# Beispiel für BFS

Startknoten:  $s = 1$



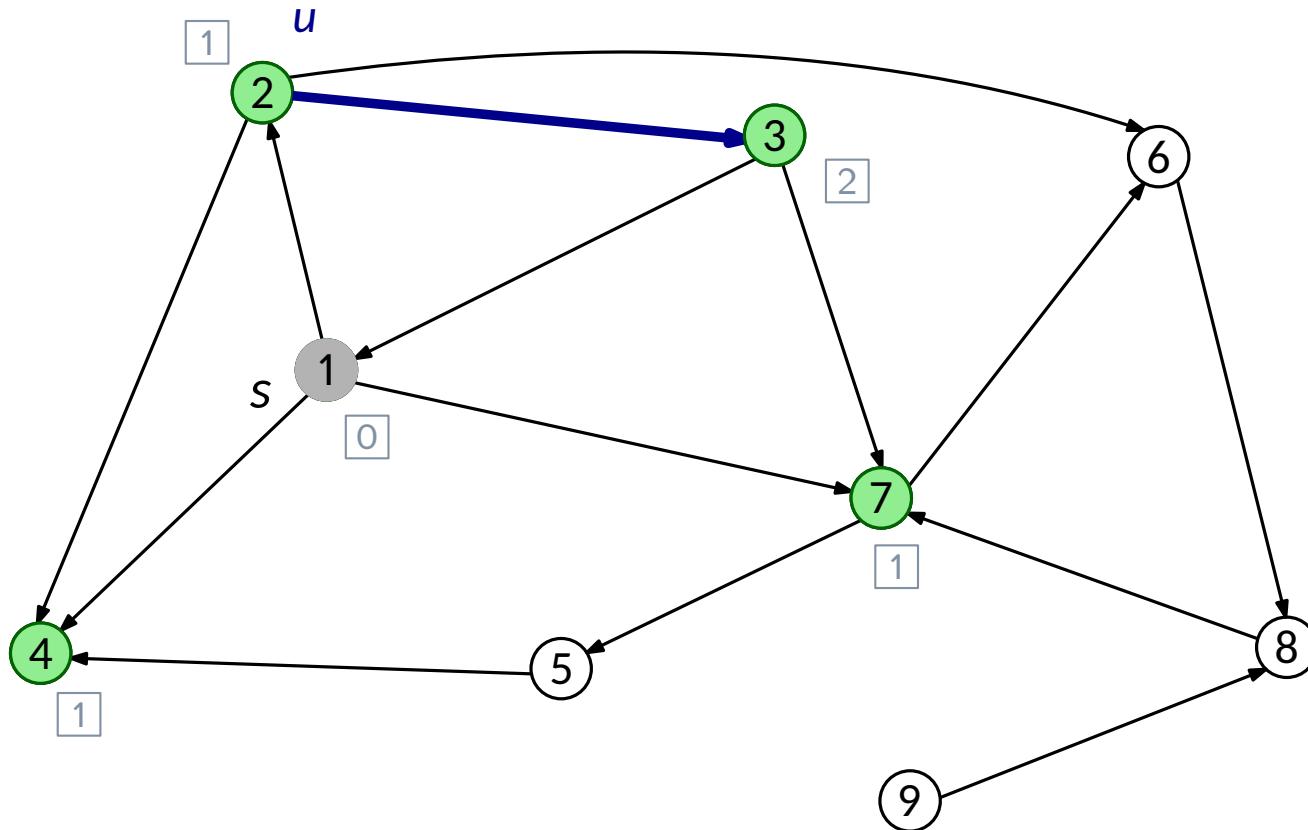
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

4 7

# Beispiel für BFS

Startknoten:  $s = 1$



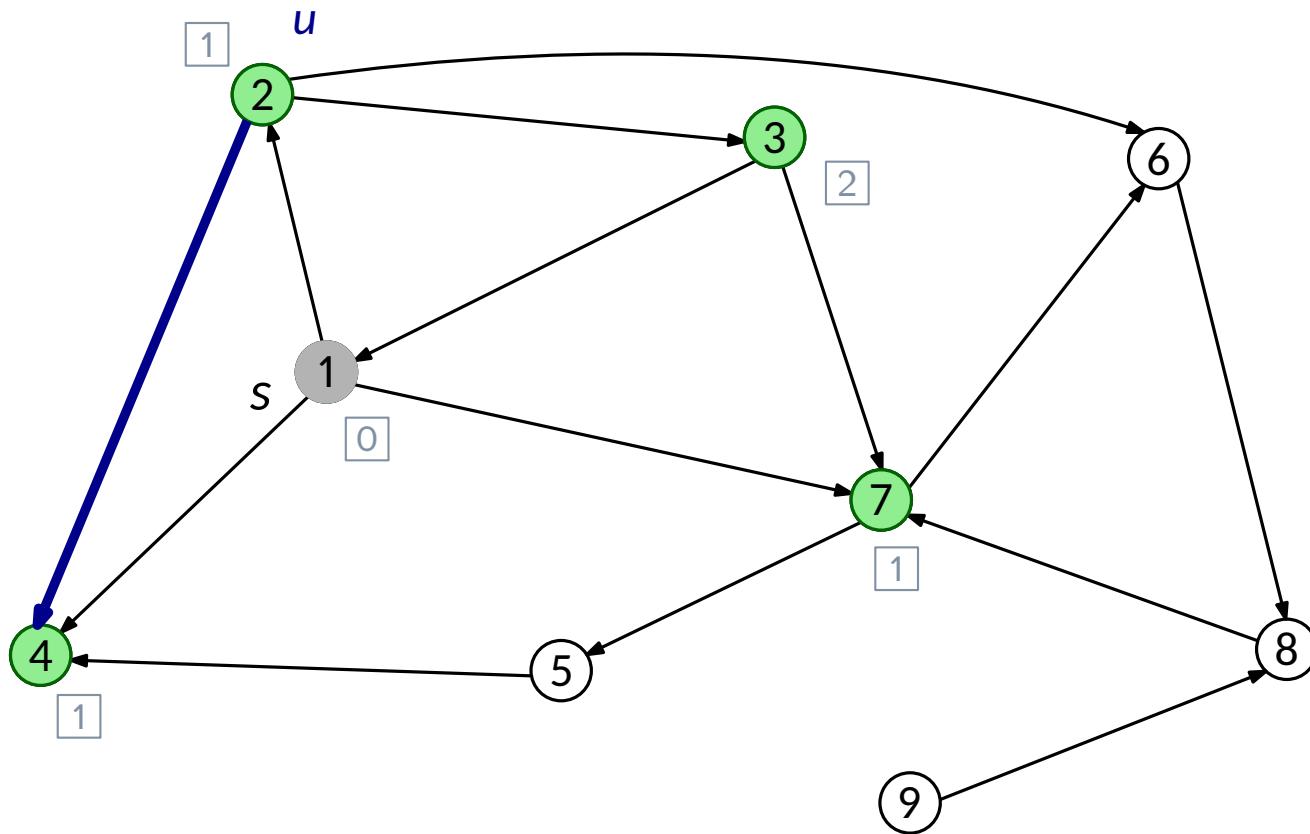
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

4   7   3

# Beispiel für BFS

Startknoten:  $s = 1$



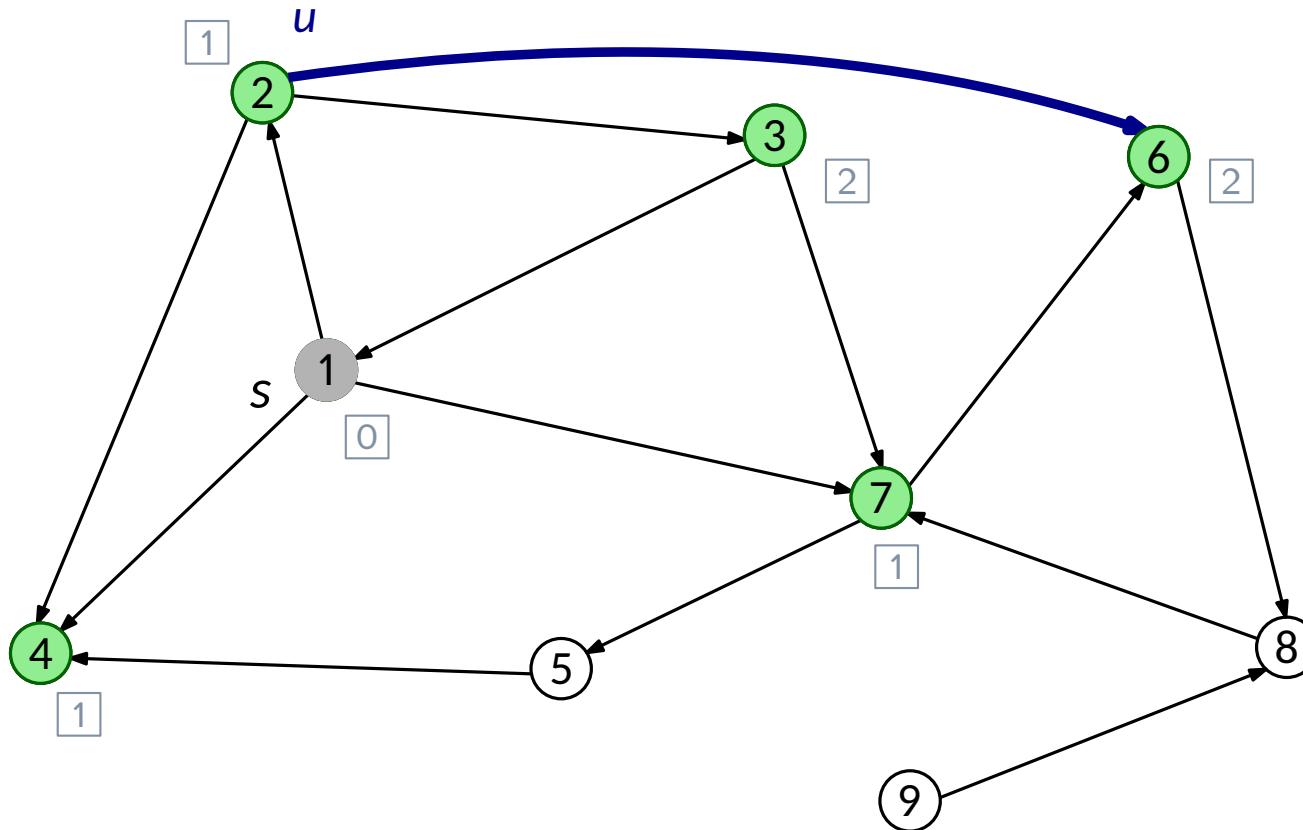
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

4   7   3

# Beispiel für BFS

Startknoten:  $s = 1$



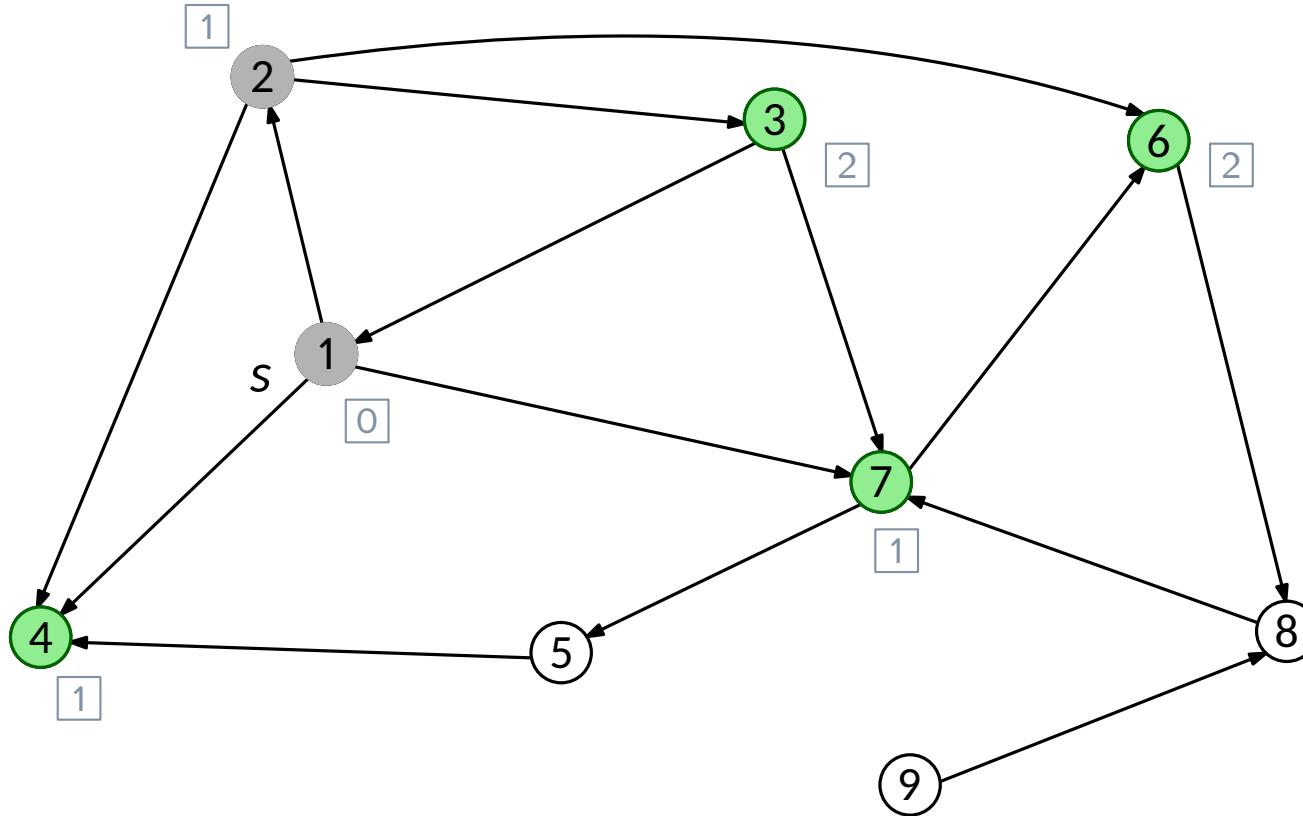
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

4 7 3 6

# Beispiel für BFS

Startknoten:  $s = 1$



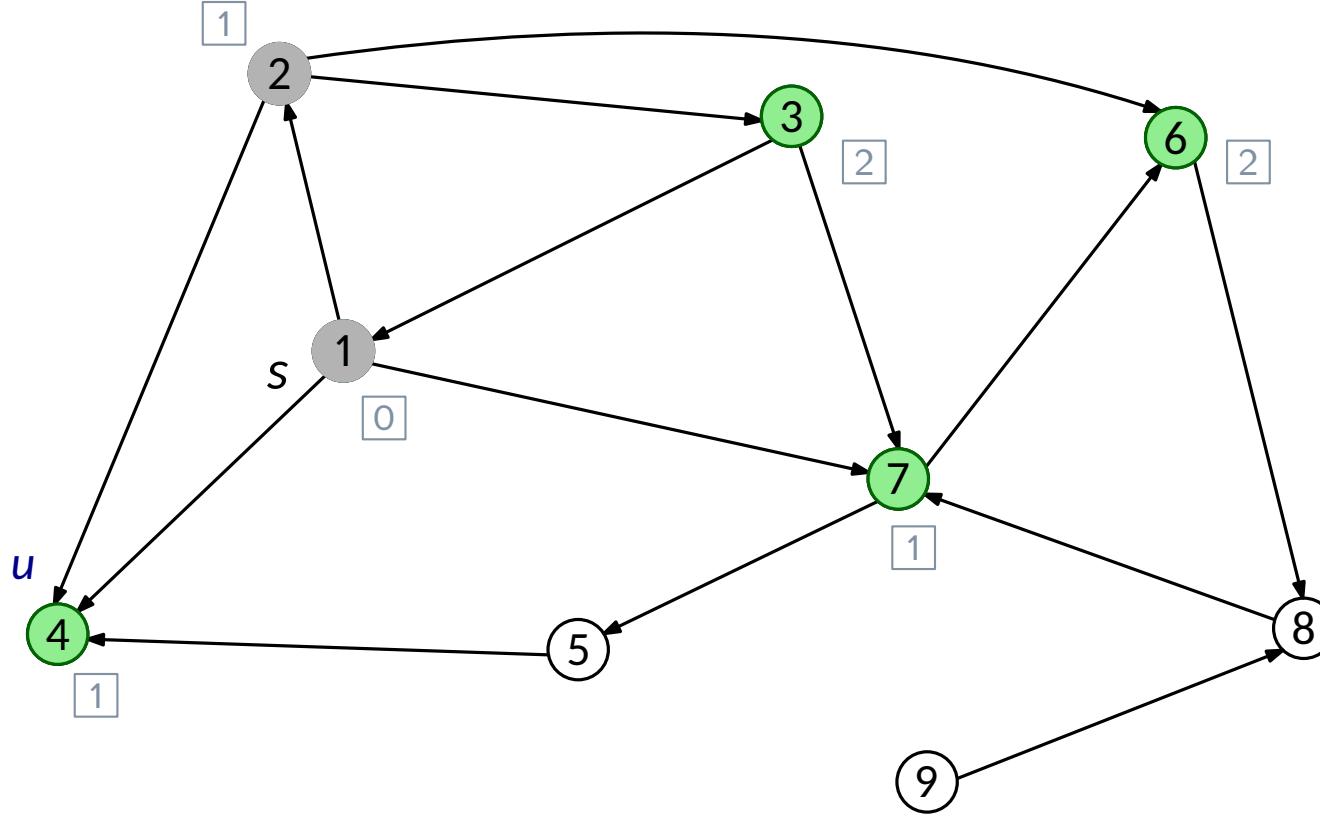
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

4 7 3 6

# Beispiel für BFS

Startknoten:  $s = 1$



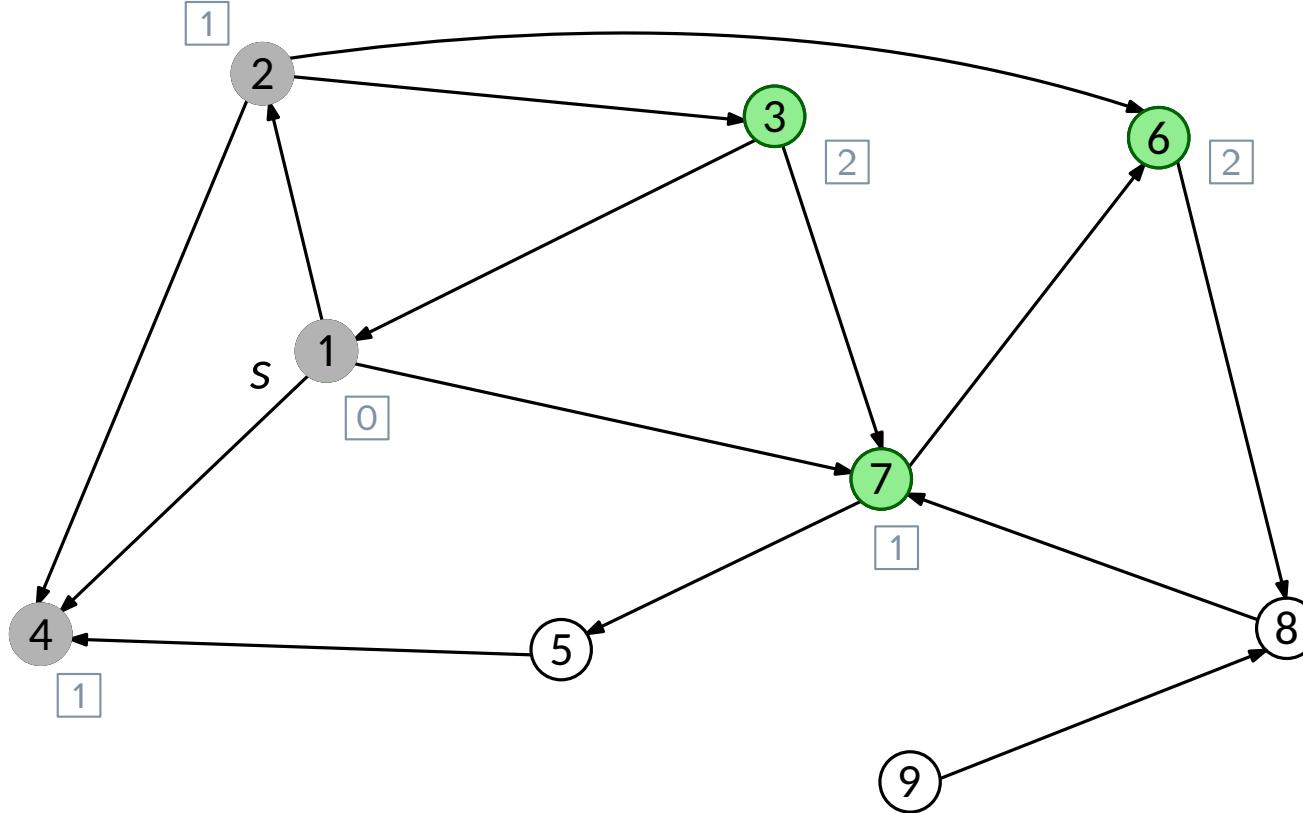
- unentdeckt
- entdeckt
- abgefertigt
- i Level i

Queue Q:

7 3 6

# Beispiel für BFS

Startknoten:  $s = 1$



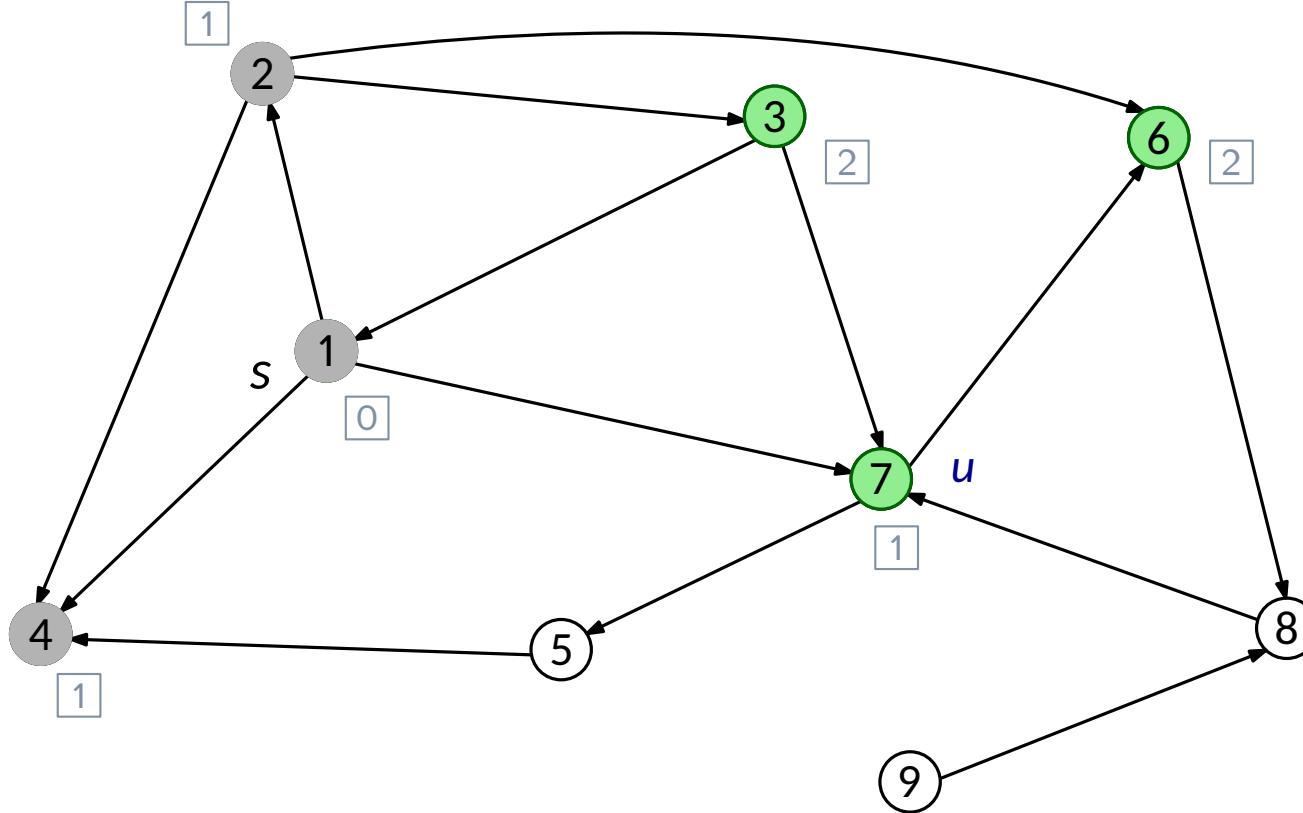
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

7 3 6

# Beispiel für BFS

Startknoten:  $s = 1$



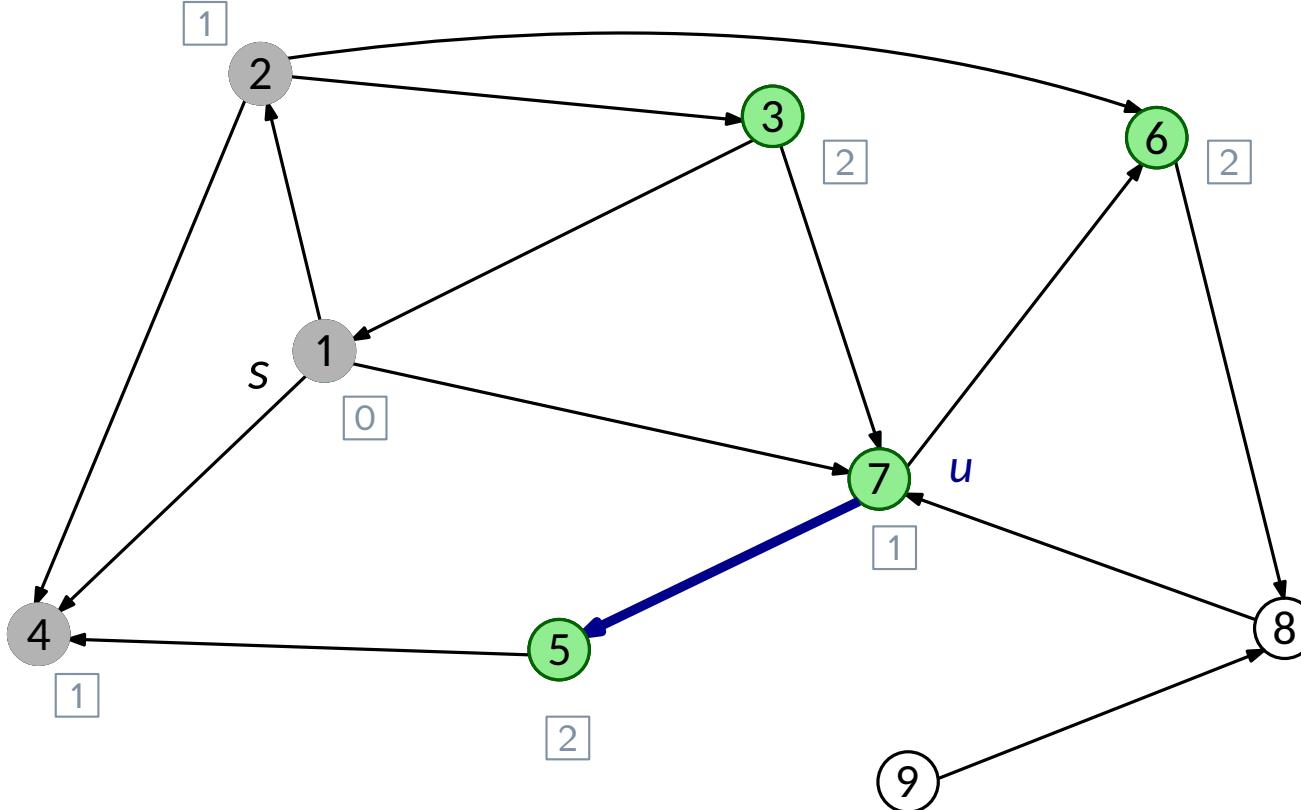
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

3 6

# Beispiel für BFS

Startknoten:  $s = 1$



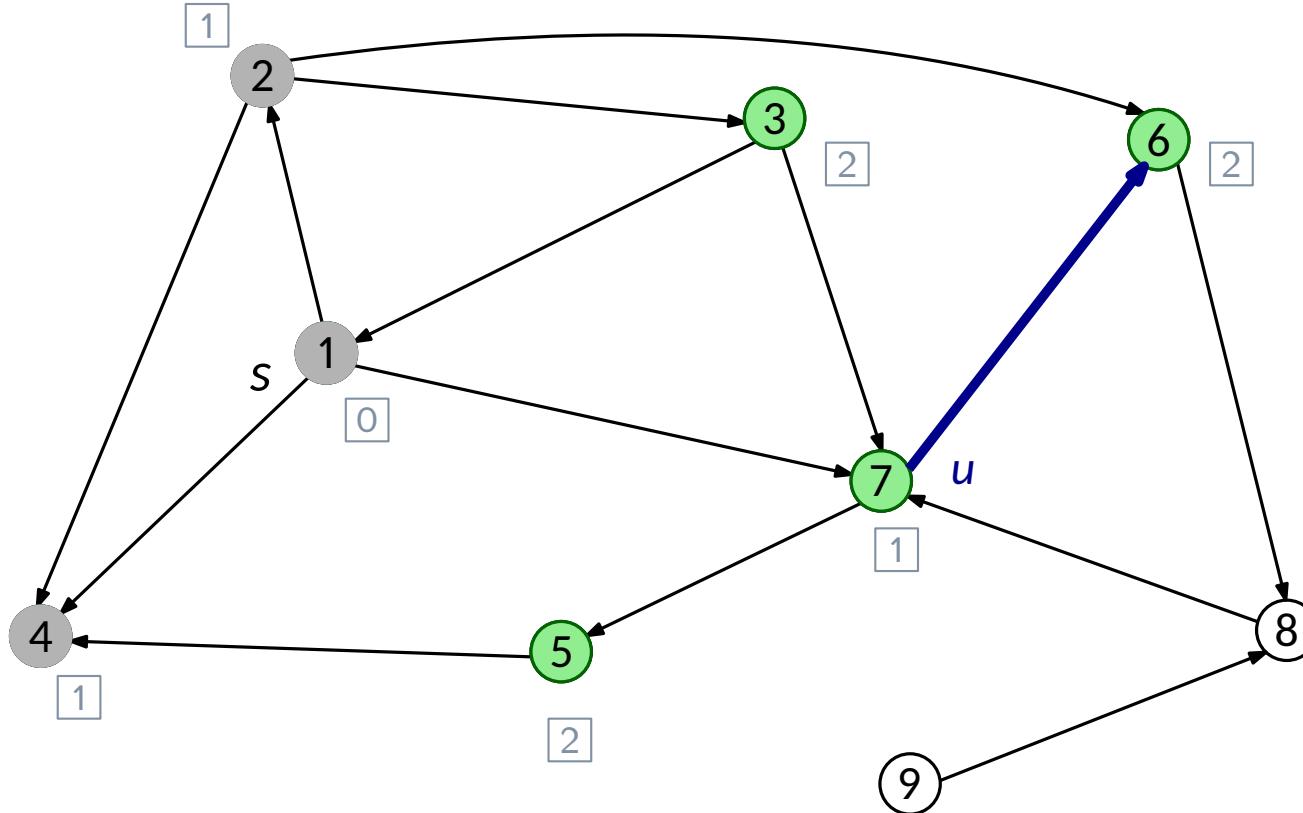
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

3 6 5

# Beispiel für BFS

Startknoten:  $s = 1$



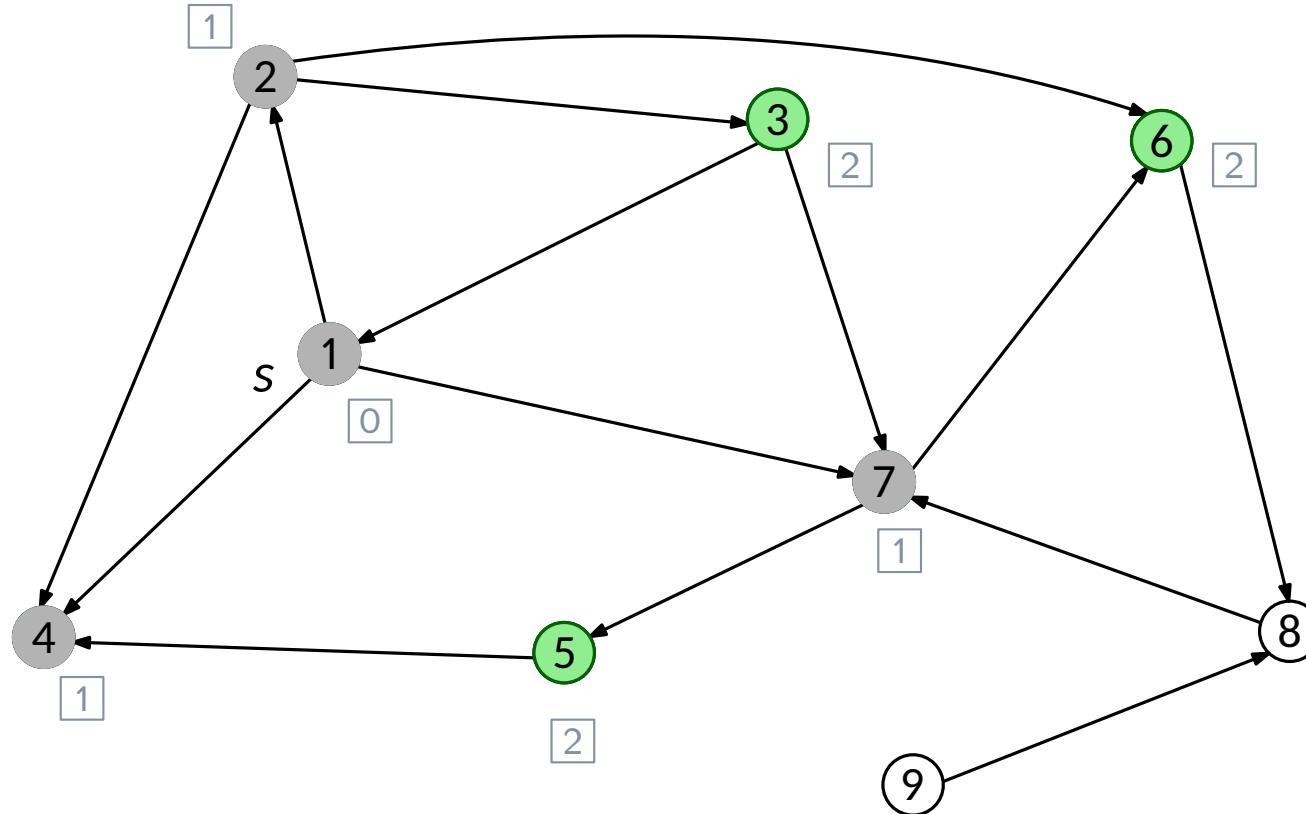
- unentdeckt
- entdeckt
- abgefertigt
- Level  $i$

Queue Q:

3 6 5

# Beispiel für BFS

Startknoten:  $s = 1$



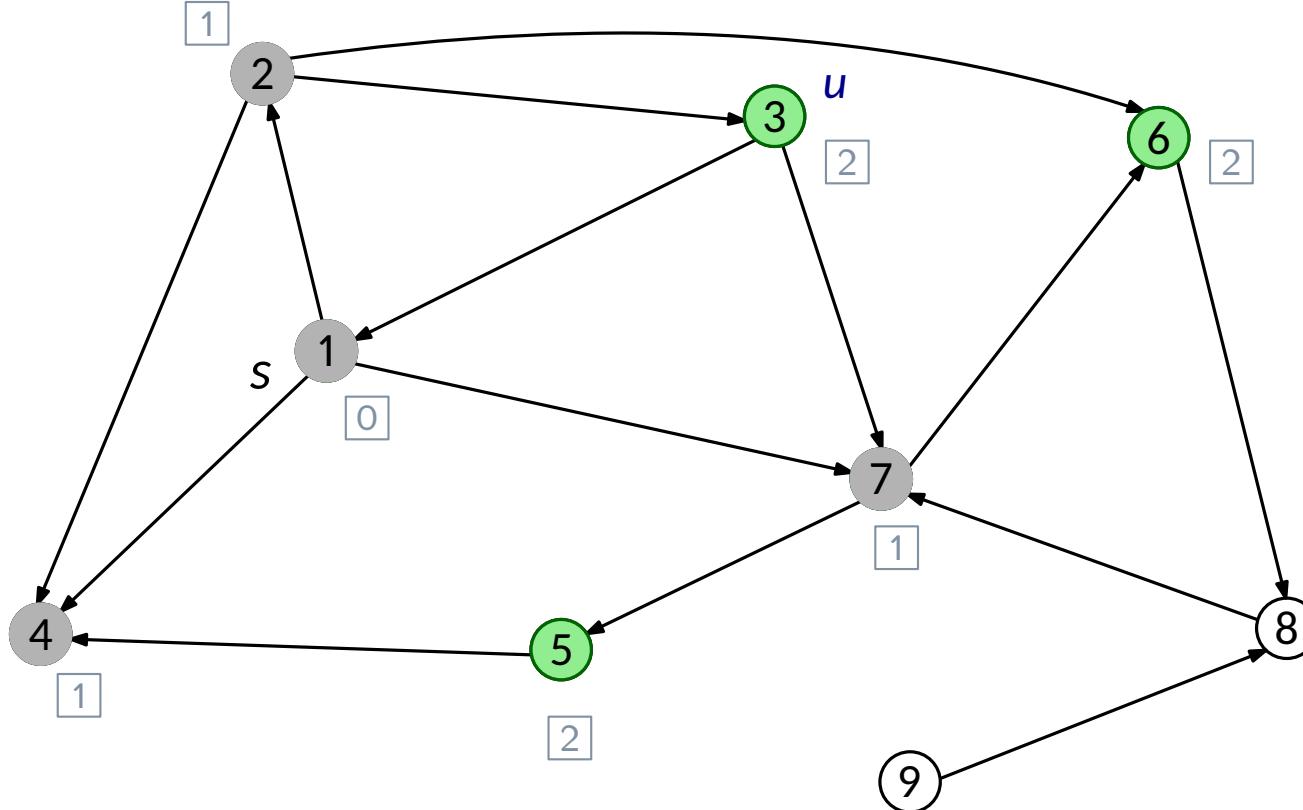
- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

Queue Q:

3 6 5

# Beispiel für BFS

Startknoten:  $s = 1$



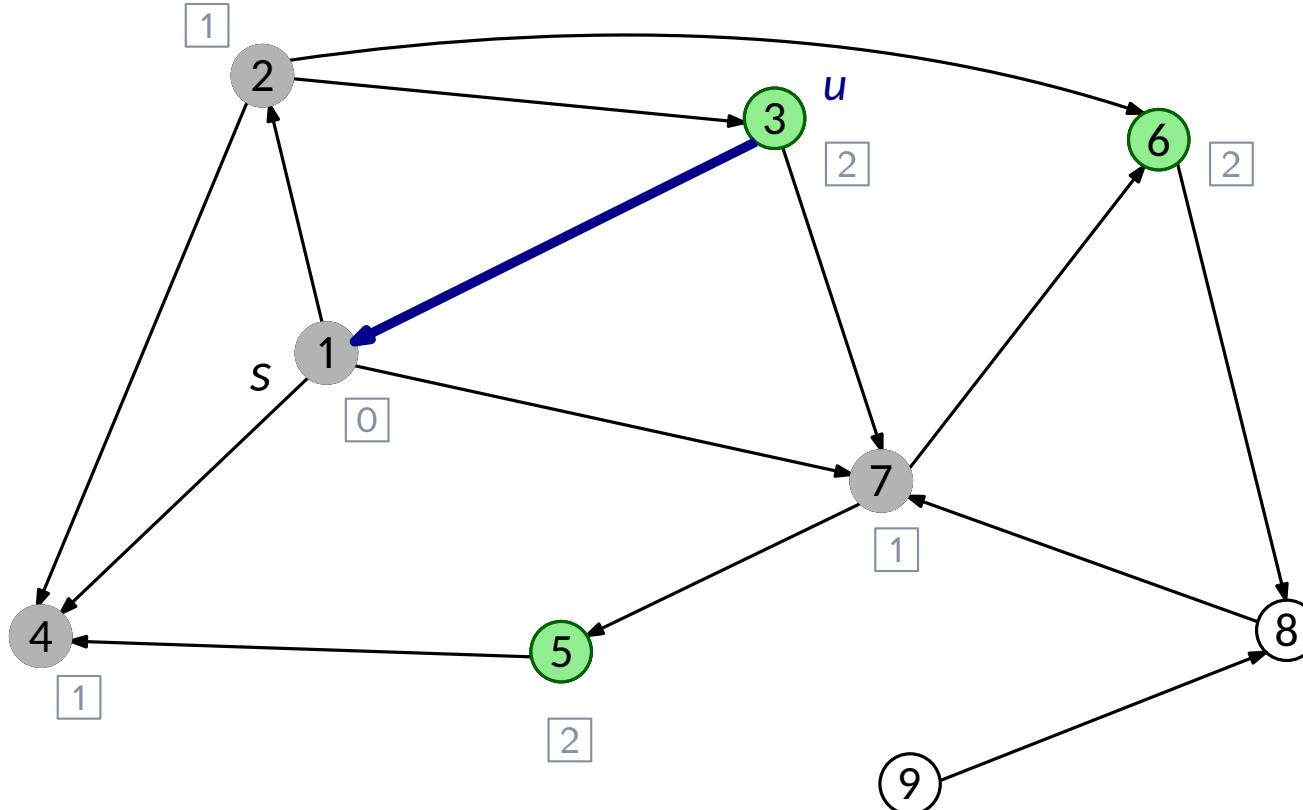
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

6 5

# Beispiel für BFS

Startknoten:  $s = 1$



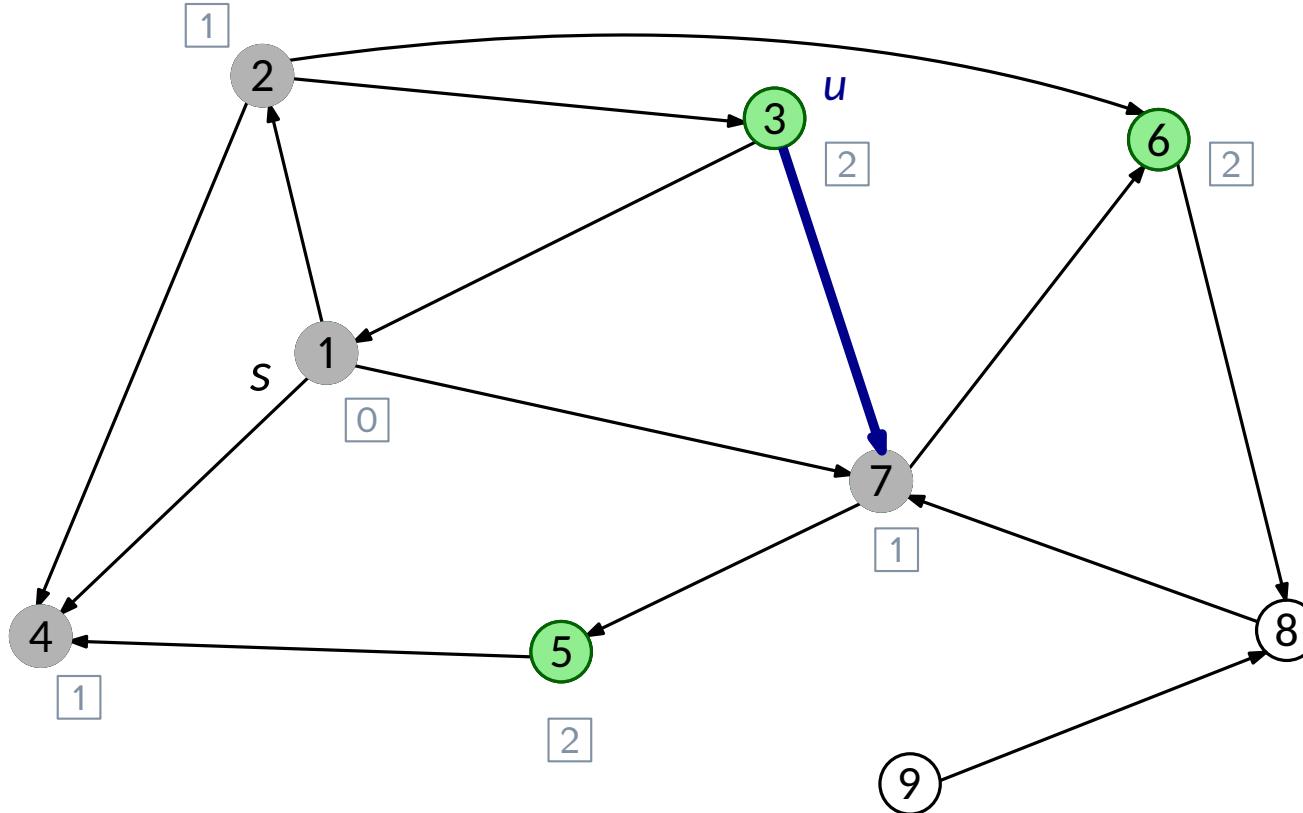
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

6 5

# Beispiel für BFS

Startknoten:  $s = 1$



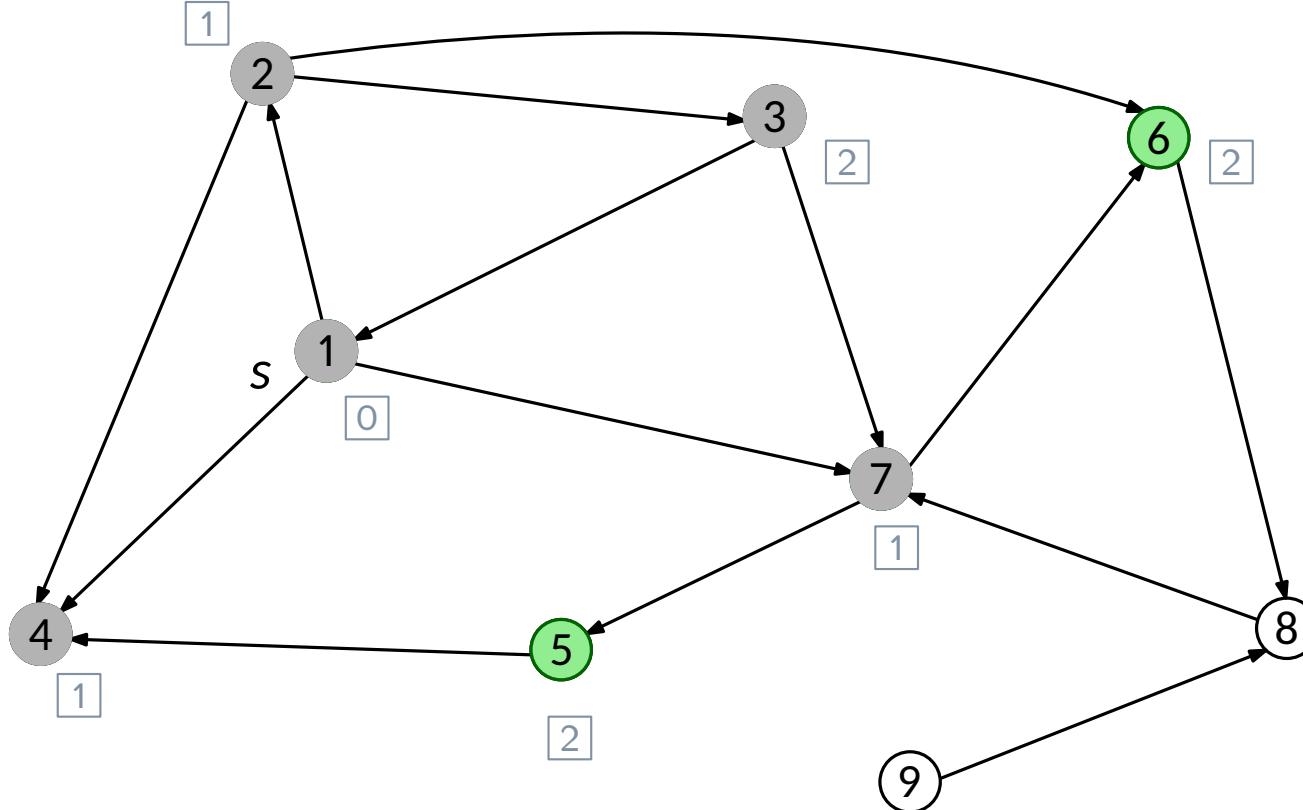
- unentdeckt
- entdeckt
- abgefertigt
- Level  $i$

Queue Q:

6 5

# Beispiel für BFS

Startknoten:  $s = 1$



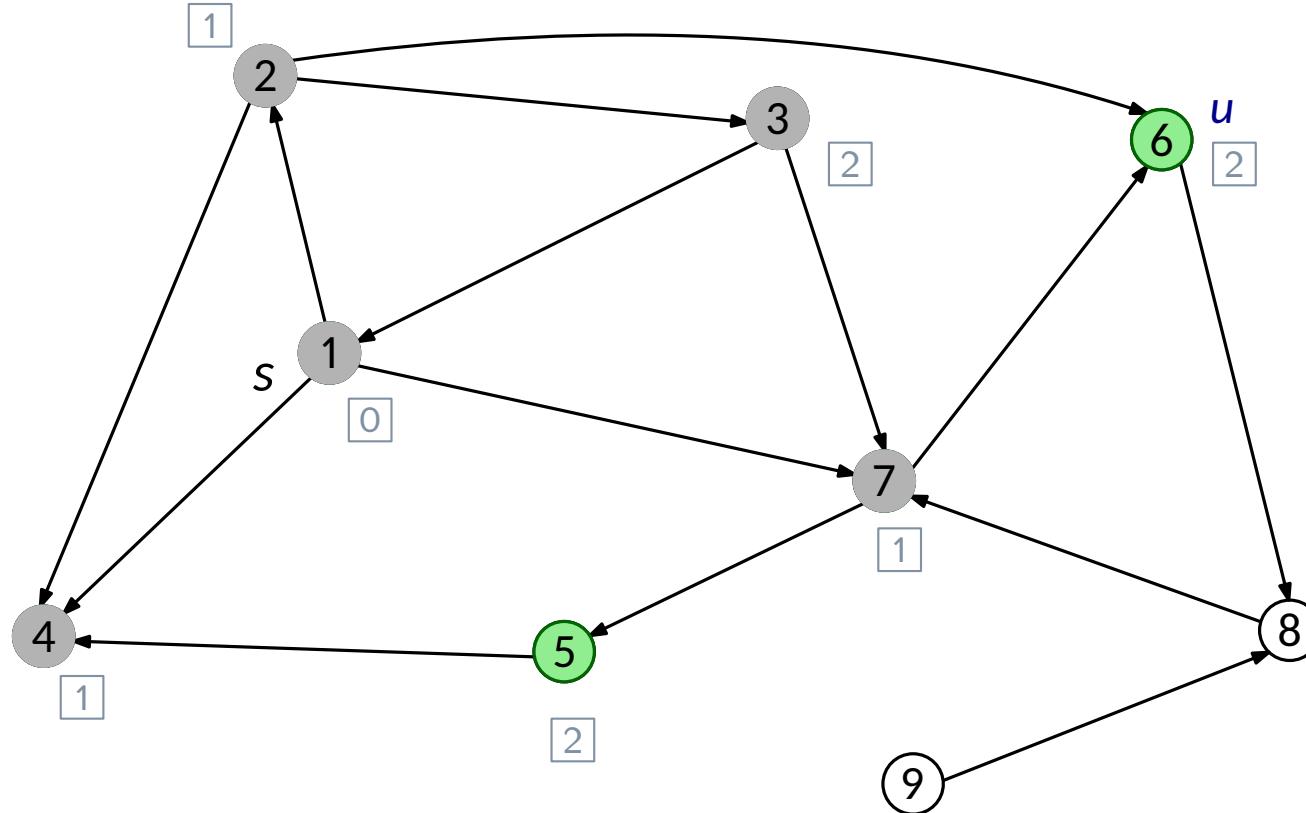
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

6 5

# Beispiel für BFS

Startknoten:  $s = 1$

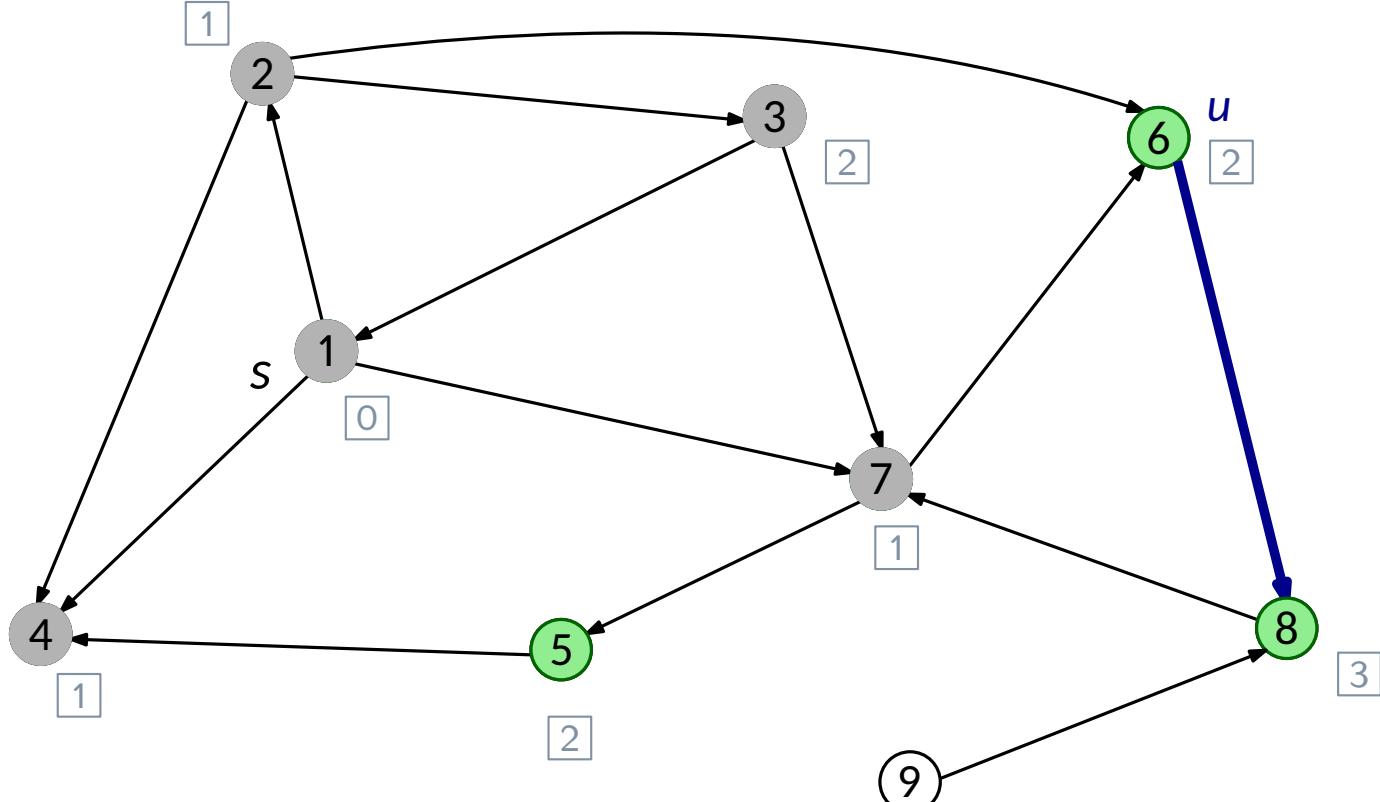


Queue Q:

5

# Beispiel für BFS

Startknoten:  $s = 1$



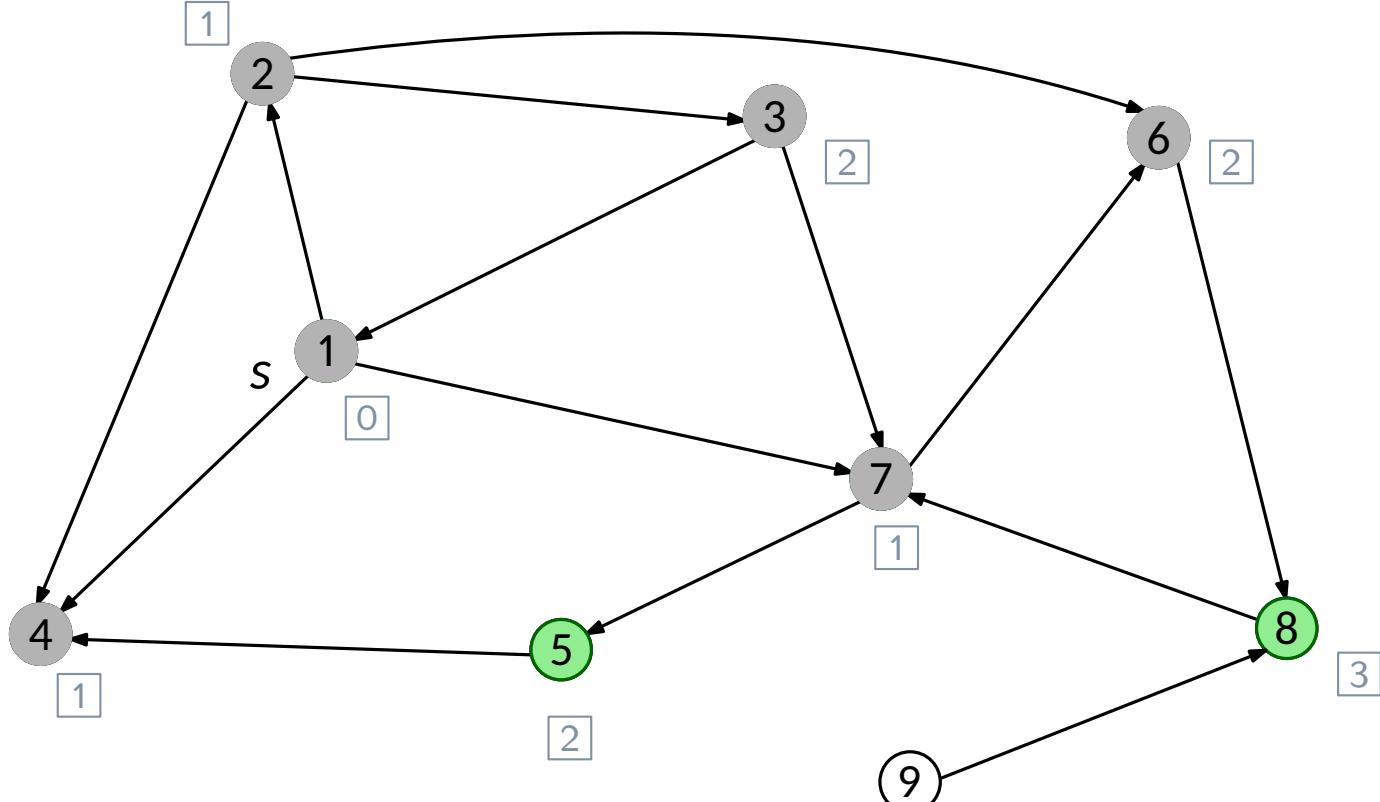
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

5 8

# Beispiel für BFS

Startknoten:  $s = 1$



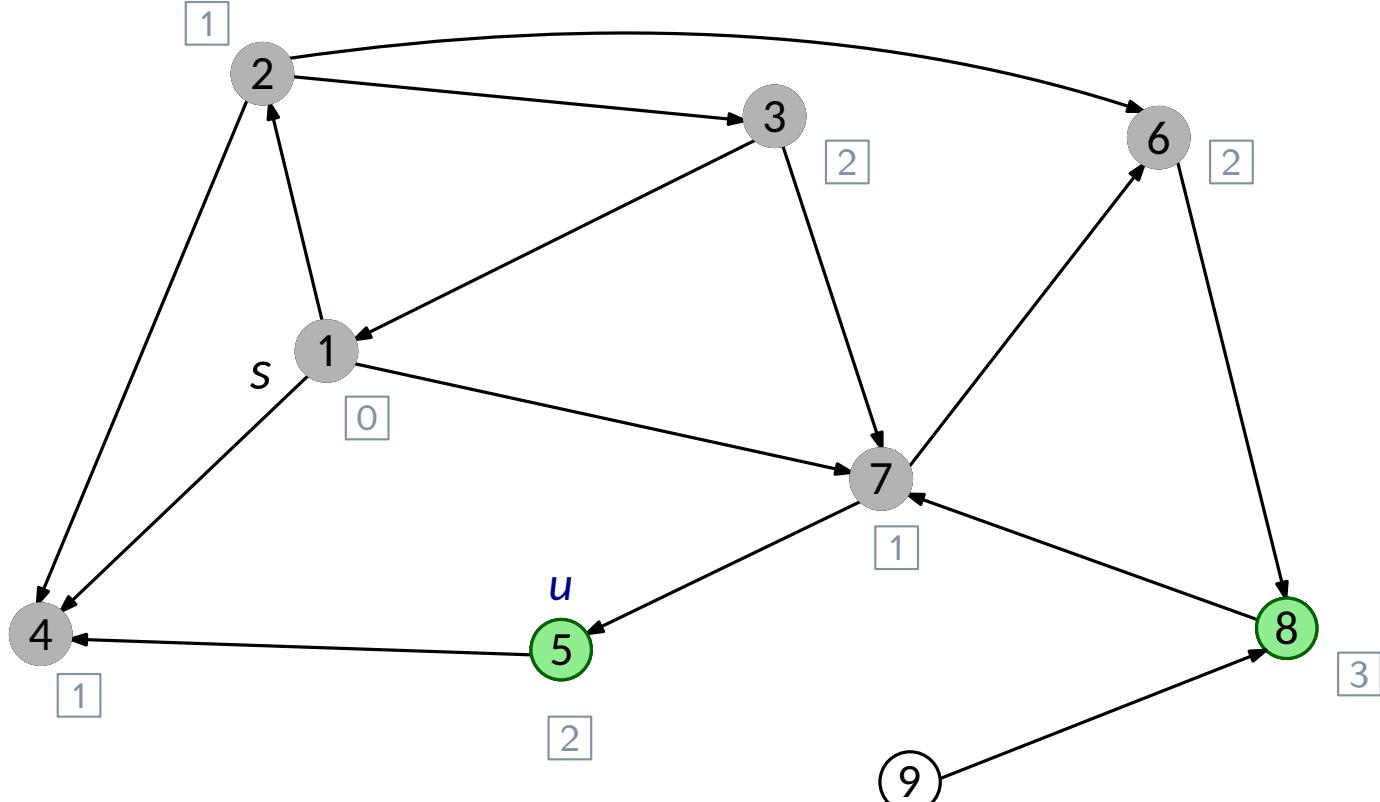
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

5 8

# Beispiel für BFS

Startknoten:  $s = 1$



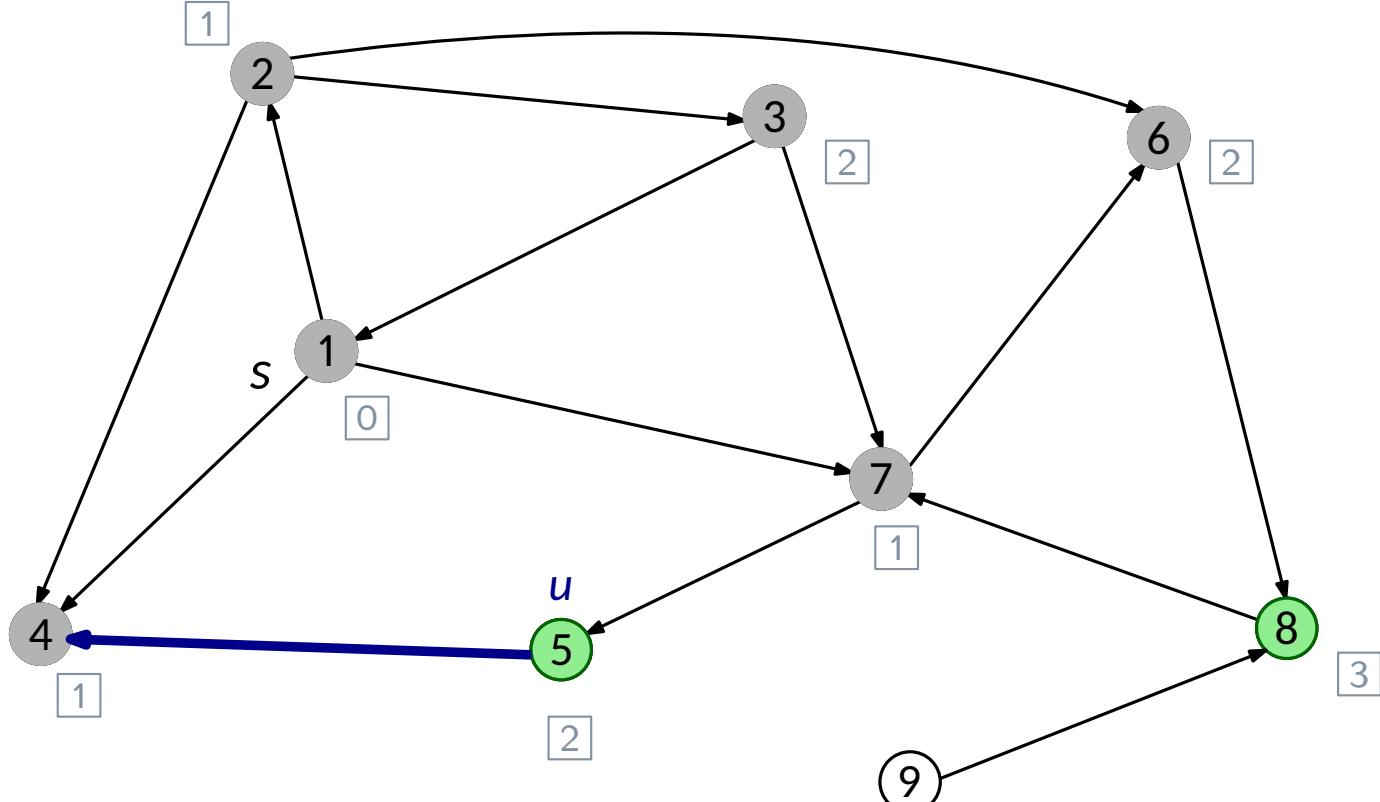
- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

8

# Beispiel für BFS

Startknoten:  $s = 1$



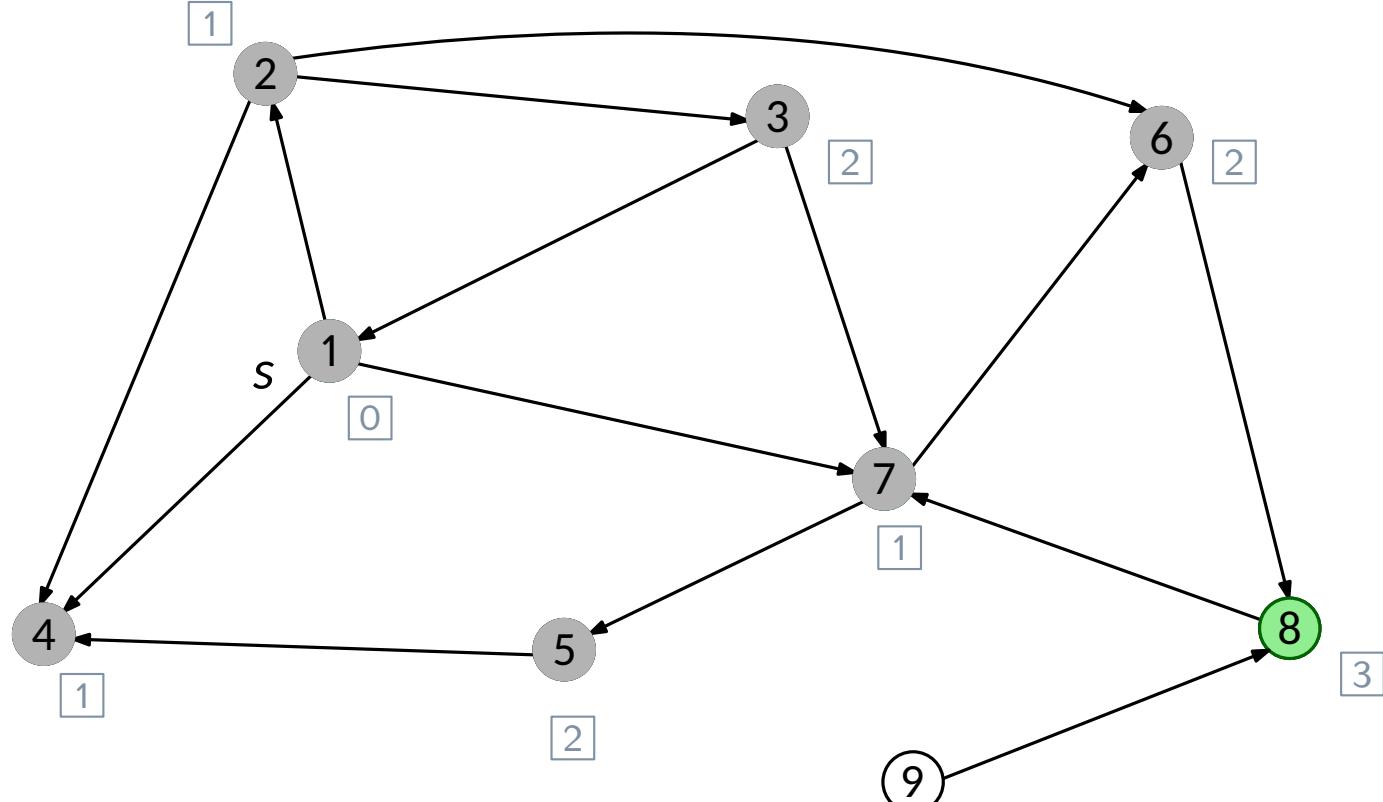
- unentdeckt
- entdeckt
- abgefertigt
- Level  $i$

Queue Q:

8

# Beispiel für BFS

Startknoten:  $s = 1$



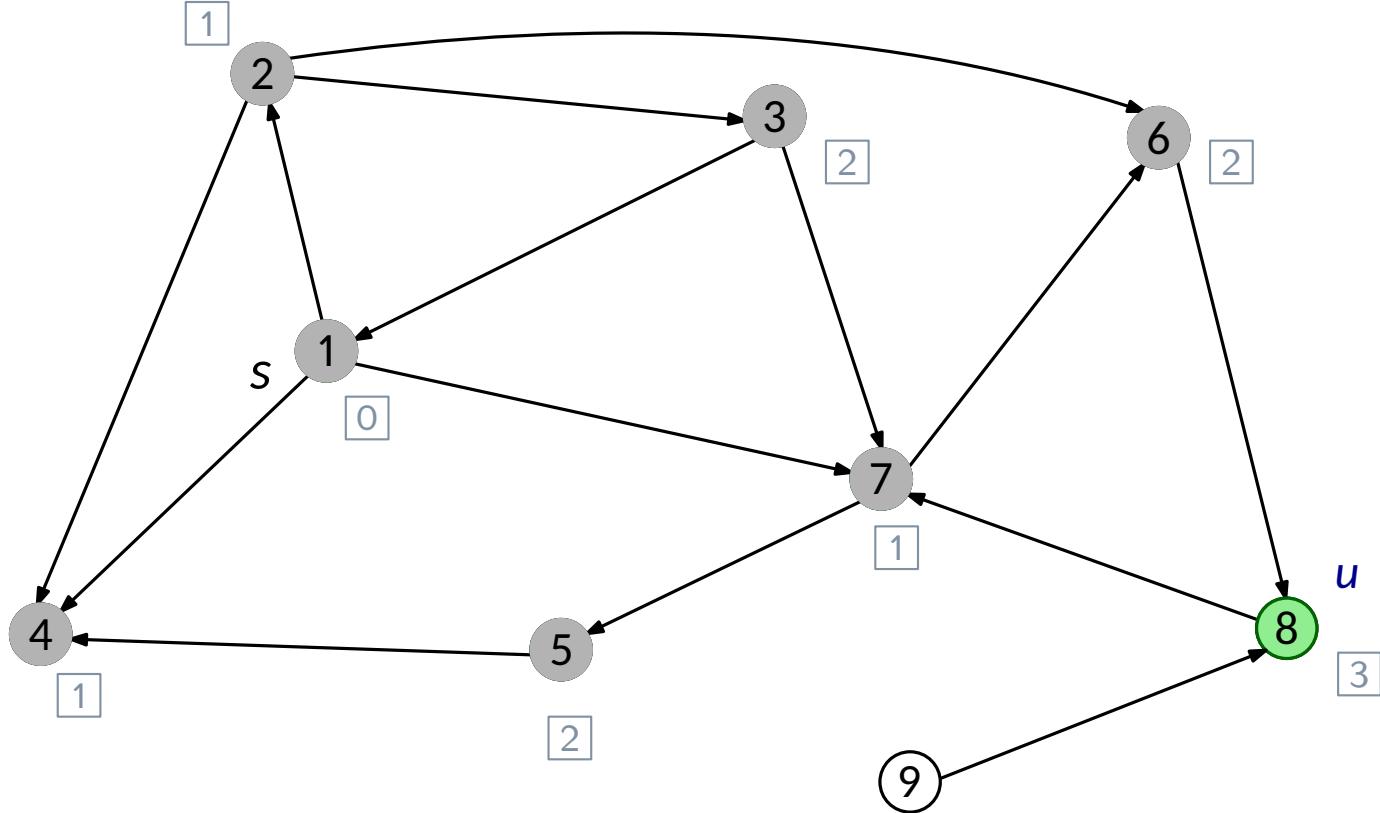
- unentdeckt
- entdeckt
- abgefertigt
- Level  $i$

Queue Q:

8

# Beispiel für BFS

Startknoten:  $s = 1$

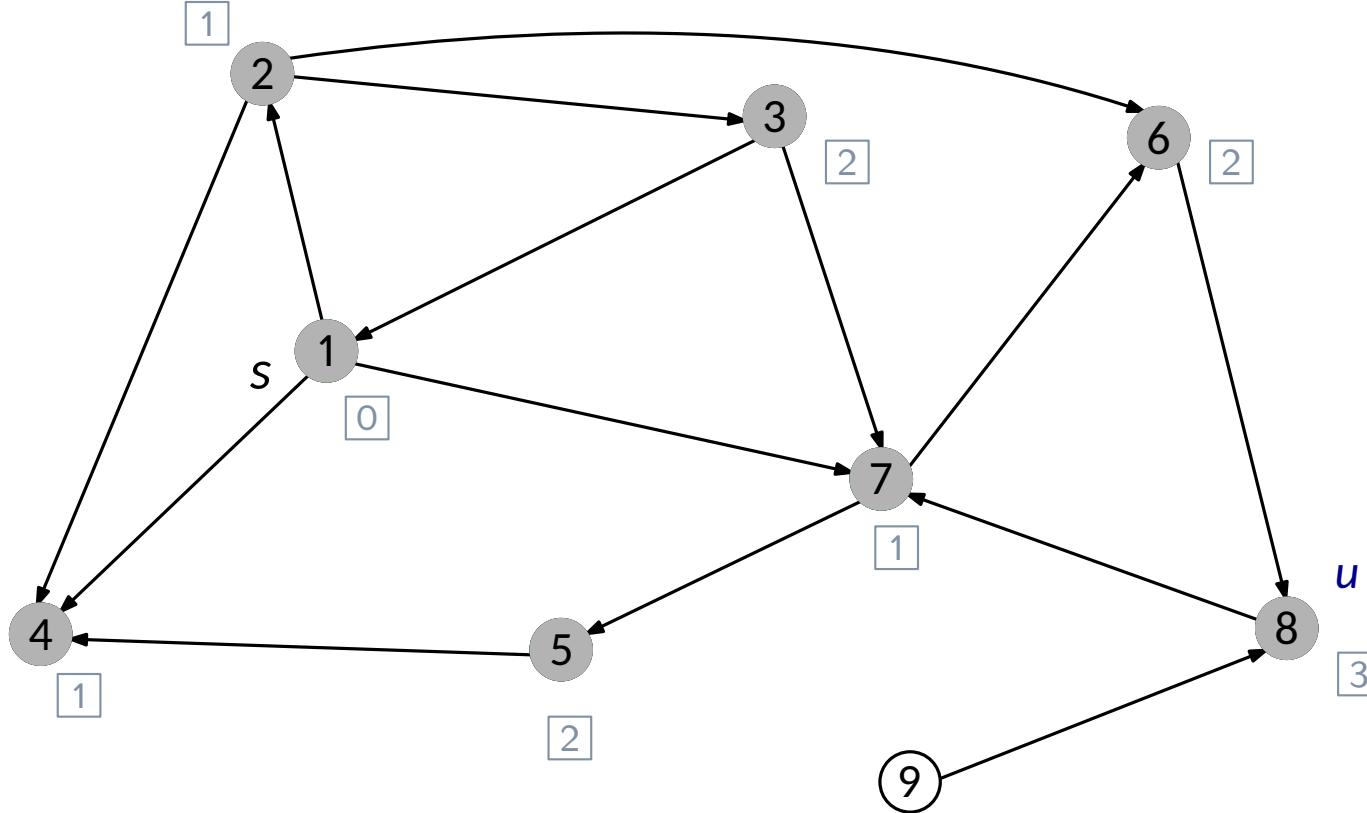


- unentdeckt (white circle)
- entdeckt (green circle)
- abgefertigt (gray circle)
- Level  $i$  (blue box)

Queue Q:

# Beispiel für BFS

Startknoten:  $s = 1$

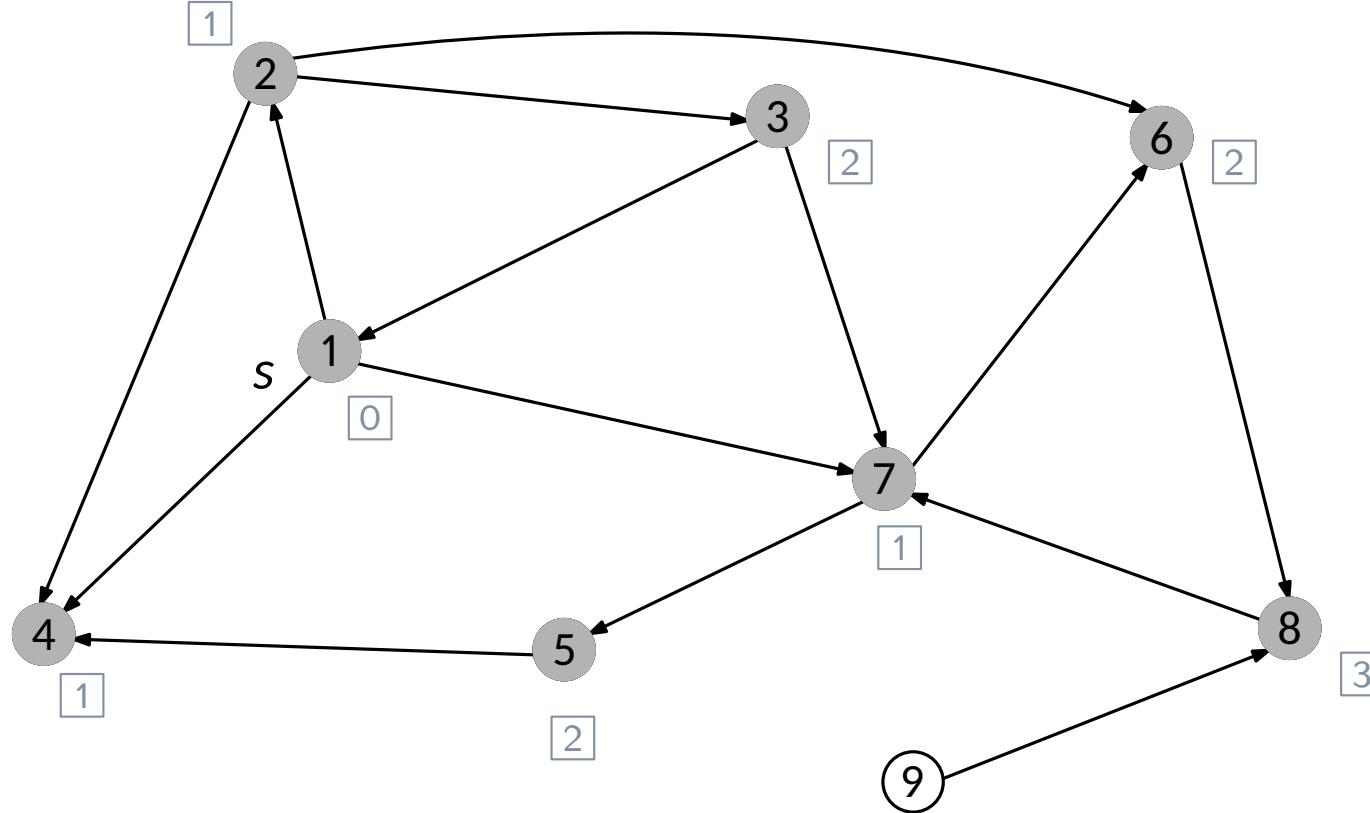


- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

# Beispiel für BFS

Startknoten:  $s = 1$

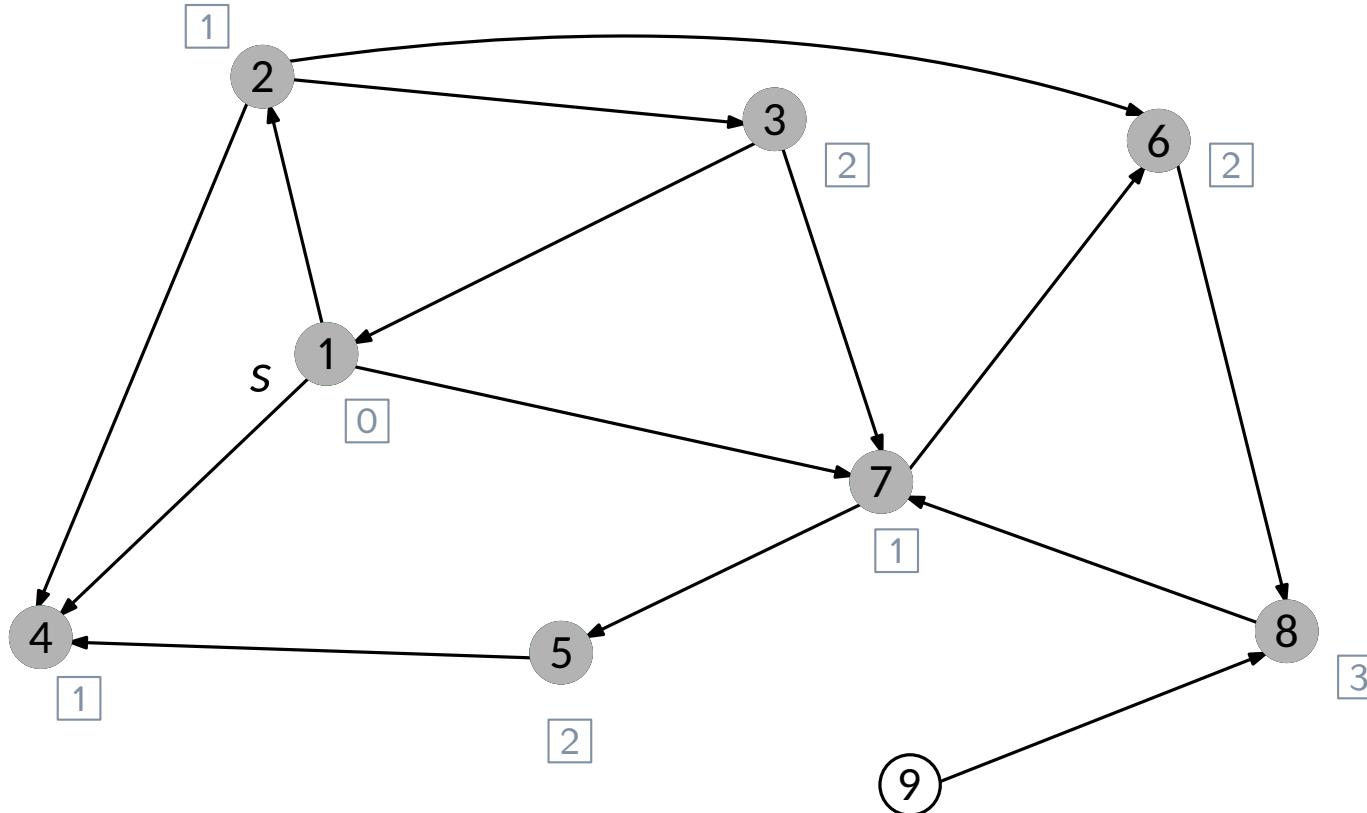


- unentdeckt
- entdeckt
- abgefertigt
- Level  $i$

Queue Q:

# Beispiel für BFS

Startknoten:  $s = 1$



- unentdeckt
- entdeckt
- abgefertigt
- i Level  $i$

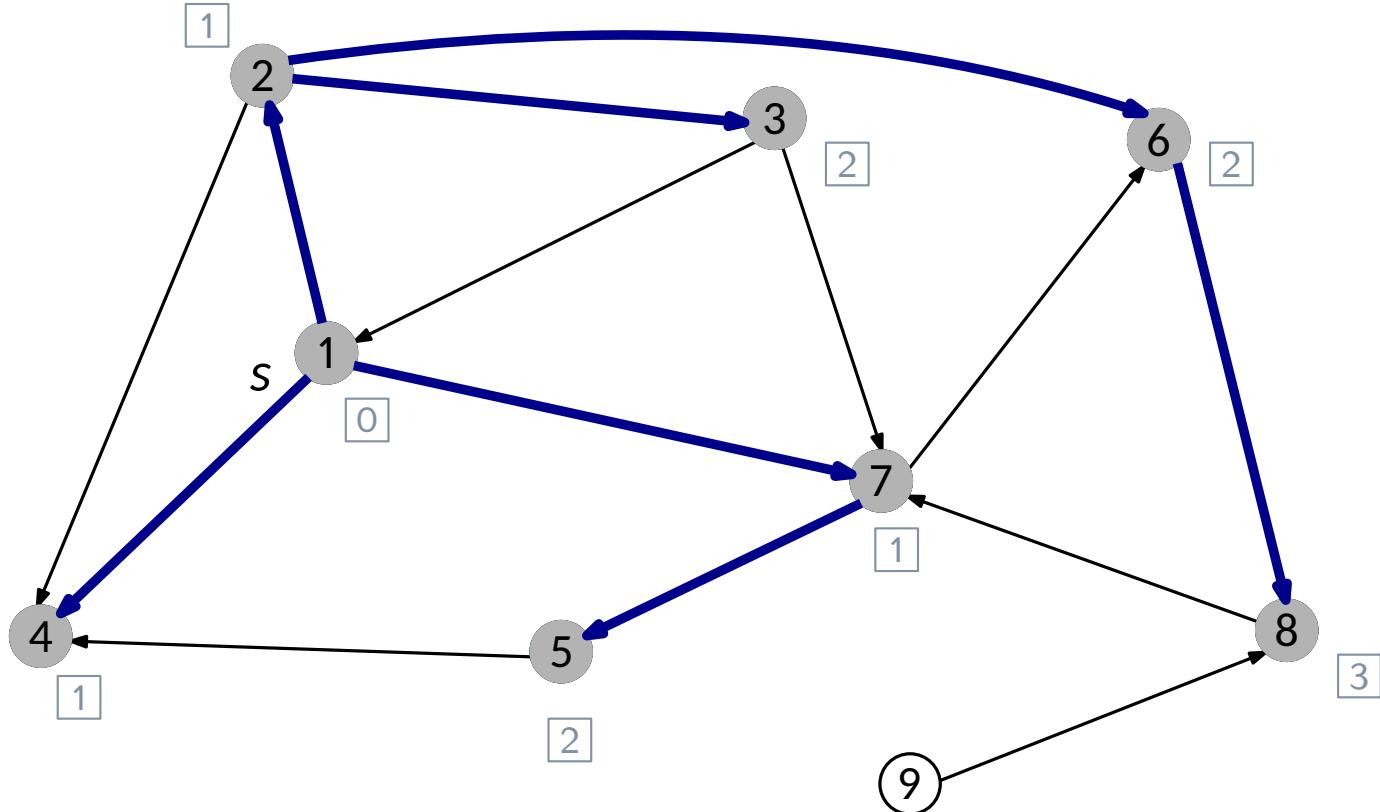
## Bemerkungen:

- Level-Nummer liefert **Abstand** von  $s$  ↗ kommende VL

Queue Q:

# Beispiel für BFS

Startknoten:  $s = 1$



- unentdeckt
- entdeckt
- abgefertigt
- $i$  Level  $i$

Queue Q:

## Bemerkungen:

- Level-Nummer liefert **Abstand** von  $s$  ↗ kommende VL
- "erste Entdeckung" der Knoten liefert einen **Suchbaum** der erreichbaren Knoten ↗ kommende Folien

# BFS: Details

BFS( $G, s$ ):

$n = \text{Anzahl an Knoten von } G$

```
for  $u = 1, \dots, n$  do  
    state[ $u$ ] = unentdeckt  
     $d[u] = \infty$   
     $p[u] = \perp$ 
```

$d[u]$  speichert das Level von Knoten  $u$

Die  $p[u]$ -Werte werden den Suchbaum beschreiben:  
·  $p[u] = v$  bedeutet  $v$  ist Parent von  $u$   
·  $s$  ist Wurzel mit  $p[s] = \perp$

Initialisiere Queue  $Q$

Frage: Kennen wir eine Kapazitätsgrenze für  $Q$ ?

```
 $d[s] = 0$   
state[ $s$ ] = entdeckt  
Q.push_back( $s$ )
```

while ( $Q$  ist nicht leer) do

```
     $u = Q.\text{front}()$   
    Q.pop_front()  
    for all  $v$  with  $(u, v) \in E$  do  
        if state[ $v$ ] = unentdeckt then  
            state[ $v$ ] = entdeckt  
             $d[v] = d[u] + 1$   
             $p[v] = u$   
            Q.push_back( $v$ )
```

genaue Implementierung von Repräsentation abhängig:

· Adjazenzlisten:  
- gehe durch alle Knoten  $v$  der Liste  $L_u$

· Adjazenzmatrix:  
- gehe durch alle  $v = 1, \dots, n$   
→ Wenn  $A[u, v] = 1$ , führe Schleifeninhalt aus

# Korrektheit und Laufzeit von BFS

## Lemma

Eine Breitensuche von  $s$  bestimmt alle von  $s$  erreichbaren Knoten in Zeit  $O(|V| + |E|) = O(n + m)$ .  
Annahme:  $G$  ist in Adjazenzlistenrepräsentation gegeben.

## Beweis:

### 1. Korrektheit

Jeden Knoten  $u$ , den wir besuchen, finden wir über einen Weg von  $s$ , also gilt  $s \rightsquigarrow_G u$

Frage: Wie kann man den Weg bestimmen, über den wir  $u$  erreicht haben?

Umgekehrt müssen wir zeigen, dass wir jeden Knoten  $u$  mit  $s \rightsquigarrow_G u$  während der BFS besuchen:

Sei  $u$  ein Knoten, der von  $s$  erreichbar ist.

Dann gibt es einen Weg  $v_0, \dots, v_k$  von  $v_0 = s$  nach  $v_k = u$ .

Jeder Knoten dieser  $v_i$  landet irgendwann in der Queue:

- $v_0 = s$  ist anfangs in der Queue
- wenn  $v_i$  in der Queue ist, dann wird  $v_{i+1}$  auch in der Queue landen:  
spätestens, wenn  $u = v_i$  werden wir  $v_{i+1}$  zur Queue hinzufügen, da  $(v_i, v_{i+1}) \in E$ .

→  $u$  landet in der Queue und wird besucht

### 2. Laufzeit

Initialisierung läuft in Zeit  $O(n)$

Jeder Knoten wird nur einmal in die Queue eingefügt und entnommen.

→  $\leq n$  Iterationen der while-Schleife

Jede Kante  $e = (x, y)$  wird nur einmal betrachtet (wenn  $u = x$ )

→  $\leq m$  Gesamtiterationen der for-all-Schleife (über alle Iterationen der while-Schleife)

□

# BFS: Alternative Laufzeitanalyse

BFS( $G, s$ ):

$n = \text{Anzahl an Knoten von } G$

```
for  $u = 1, \dots, n$  do
    state[ $u$ ] = unentdeckt
     $d[u] = \infty$ 
     $p[u] = \perp$ 
```

Initialisiere Queue Q

```
 $d[s] = 0$ 
state[ $s$ ] = entdeckt
Q.push_back( $s$ )
```

**while** (Q ist nicht leer) **do**

```
     $u = Q.\text{front}()$ 
    Q.pop_front()
    state[ $u$ ] = abgefertigt
```

**for all**  $v$  with  $(u, v) \in E$  **do**

```
        if state[ $v$ ] = unentdeckt then
```

```
            state[ $v$ ] = entdeckt
             $d[v] = d[u] + 1$ 
             $p[v] = u$ 
            Q.push_back( $v$ )
```

$O(n)$

$O(1)$

$O(\deg^+(u))$

Alternative Argumentation:

Gesamtaufzeit ist:

$$O(n + \sum_{u:s \sim u} (1 + \deg^+(u)))$$

Es gilt:

$$\begin{aligned} & \sum_{u:s \sim u} (1 + \deg^+(u)) \\ & \leq \sum_{u \in V} (1 + \deg^+(u)) \\ & = |V| + \sum_{u \in V} \deg^+(u) \\ & = n + m \end{aligned}$$

# Tiefensuche (DFS)

---

Idee:

Sobald wir einen von  $s$  erreichbaren Knoten  $u$  entdecken, durchlaufen wir alle von  $u$  erreichbare Knoten

Algorithmenbeschreibung:

- Zu jedem Knoten speichern wir einen Status: **unentdeckt**, **entdeckt** oder **abgefertigt**
- anfangs sind alle Knoten **unentdeckt**
- wir starten eine DFS von  $s$

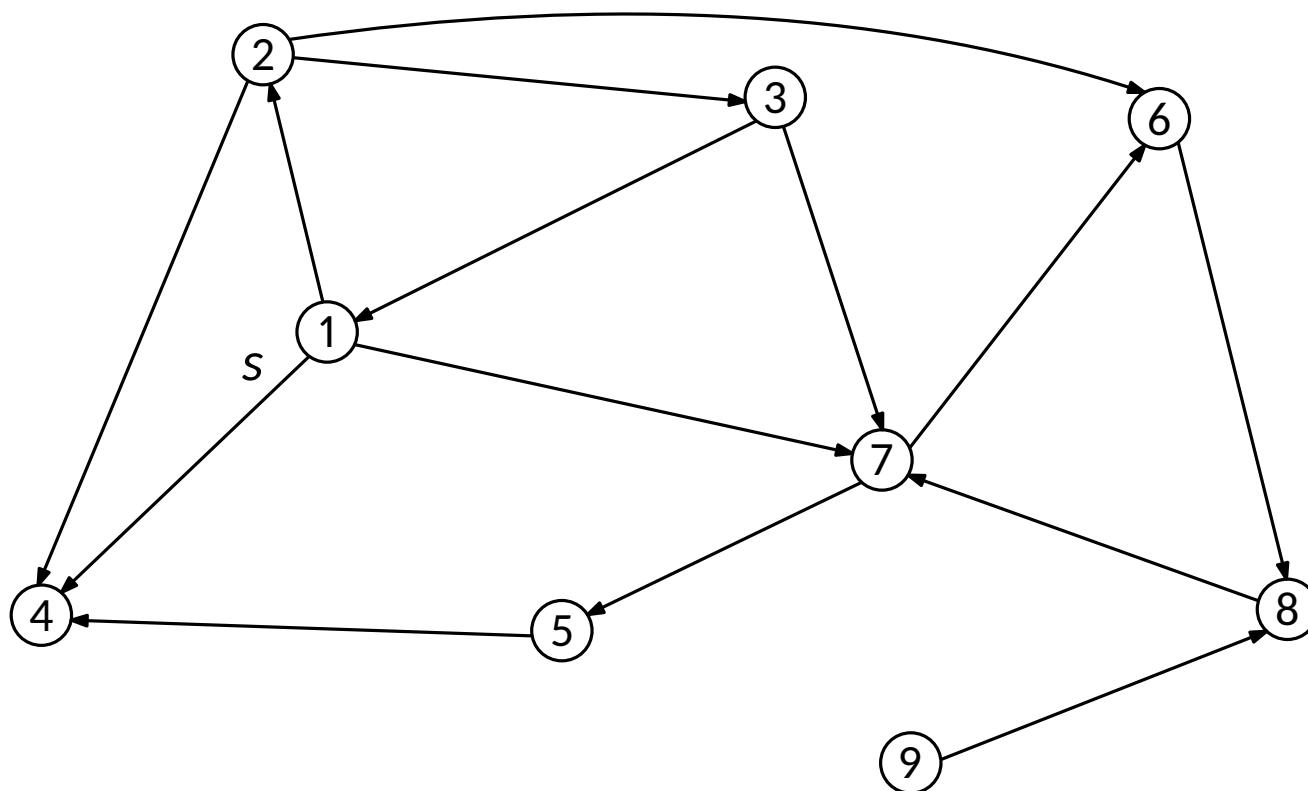
DFS von  $u$ : (**rekursives** Prinzip)

- wir setzen  $u$ 's Status auf **entdeckt**
- für jeden **unentdeckten** Knoten  $v$  mit  $(u, v) \in E$  machen wir folgendes:
  - wir starten **rekursiv** eine DFS von  $v$
- wir setzen  $u$ 's Status auf **abgefertigt**

**Hauptunterschied zu BFS:** BFS verfährt nach FIFO-Prinzip (first in, first out)  
DFS verfährt nach LIFO-Prinzip (last in, first out)

# Beispiel für DFS

Startknoten:  $s = 1$



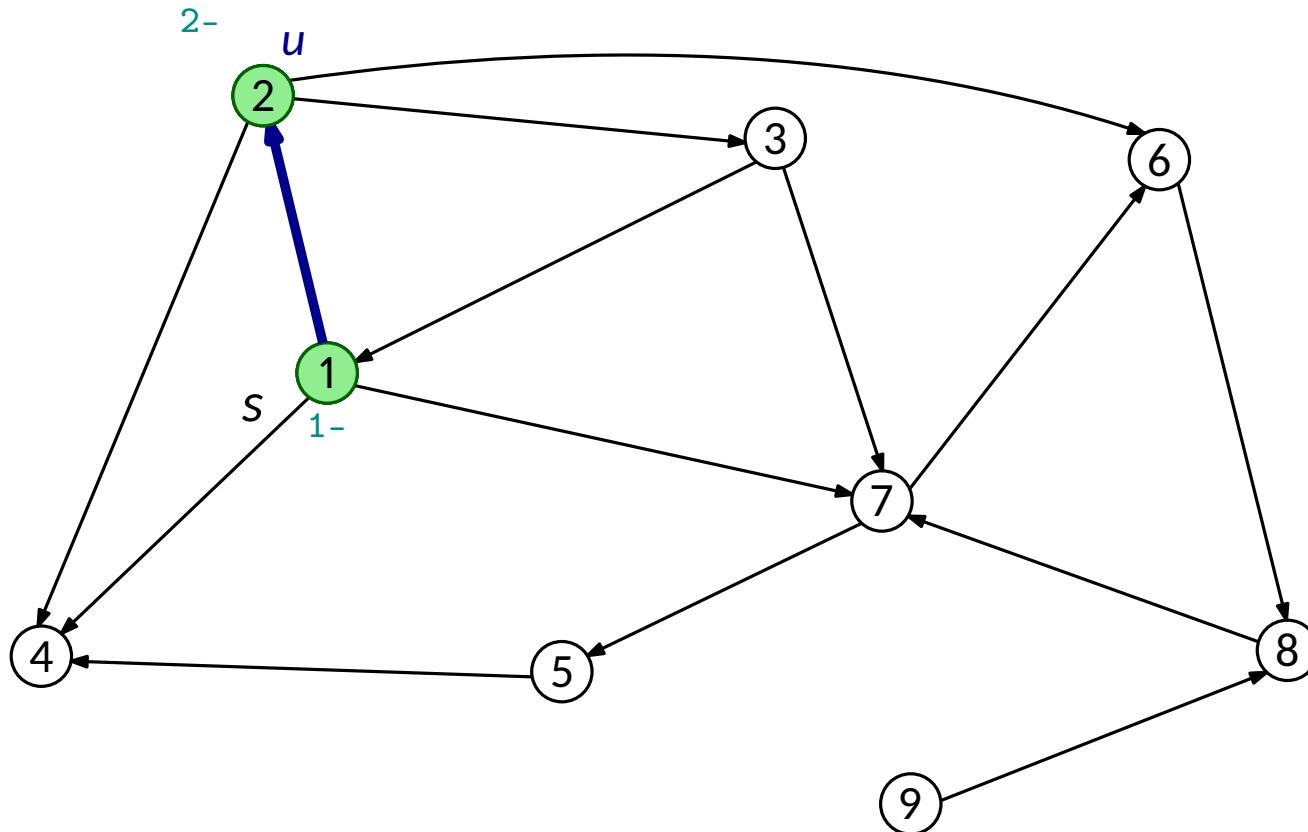
- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

1

# Beispiel für DFS

Startknoten:  $s = 1$

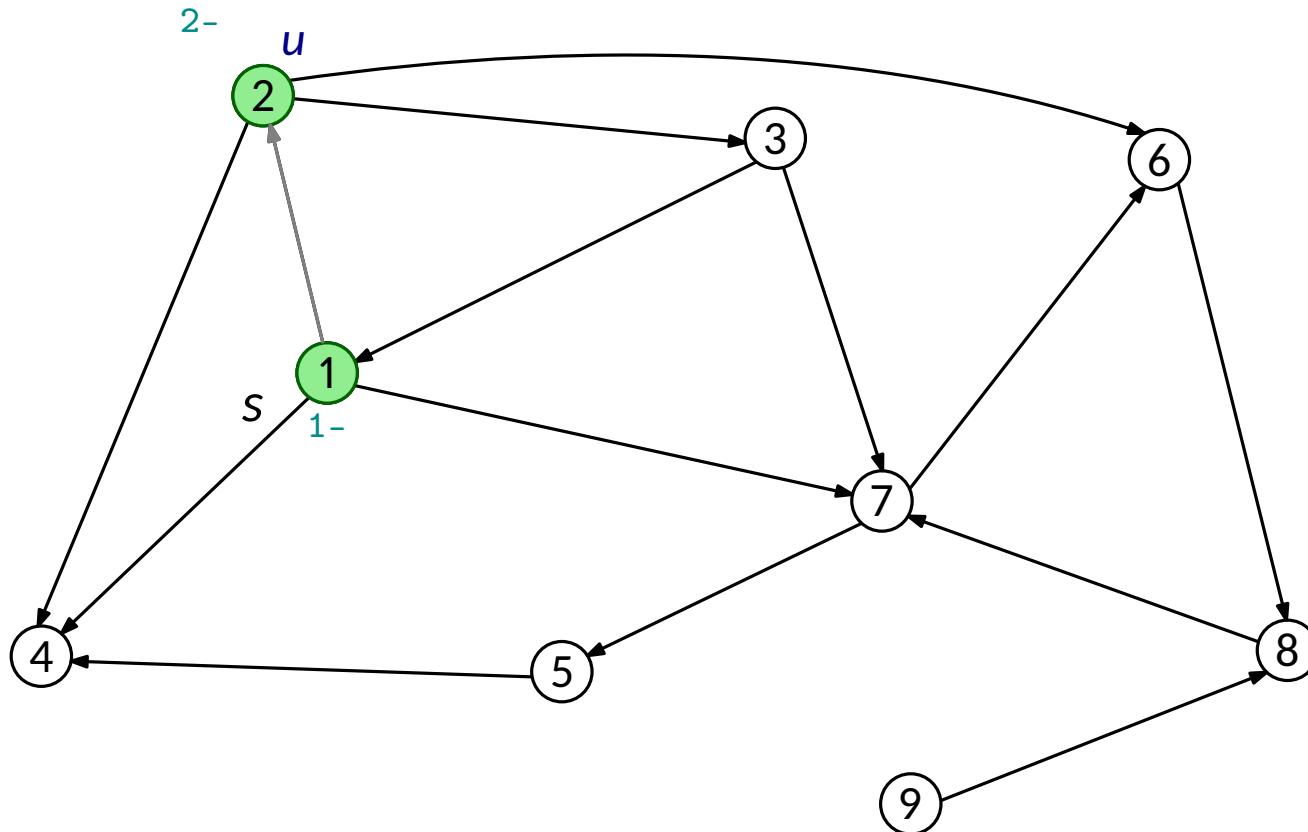


- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

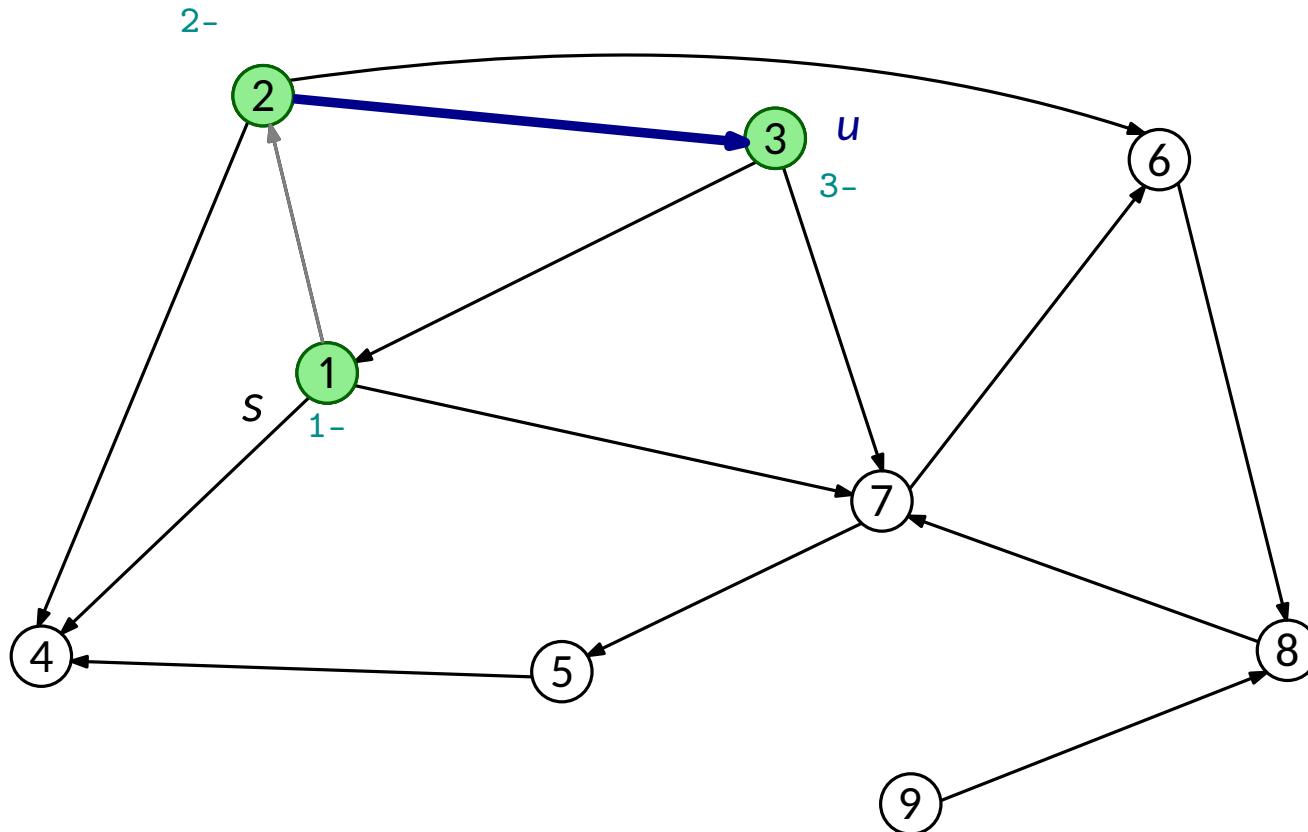


- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$



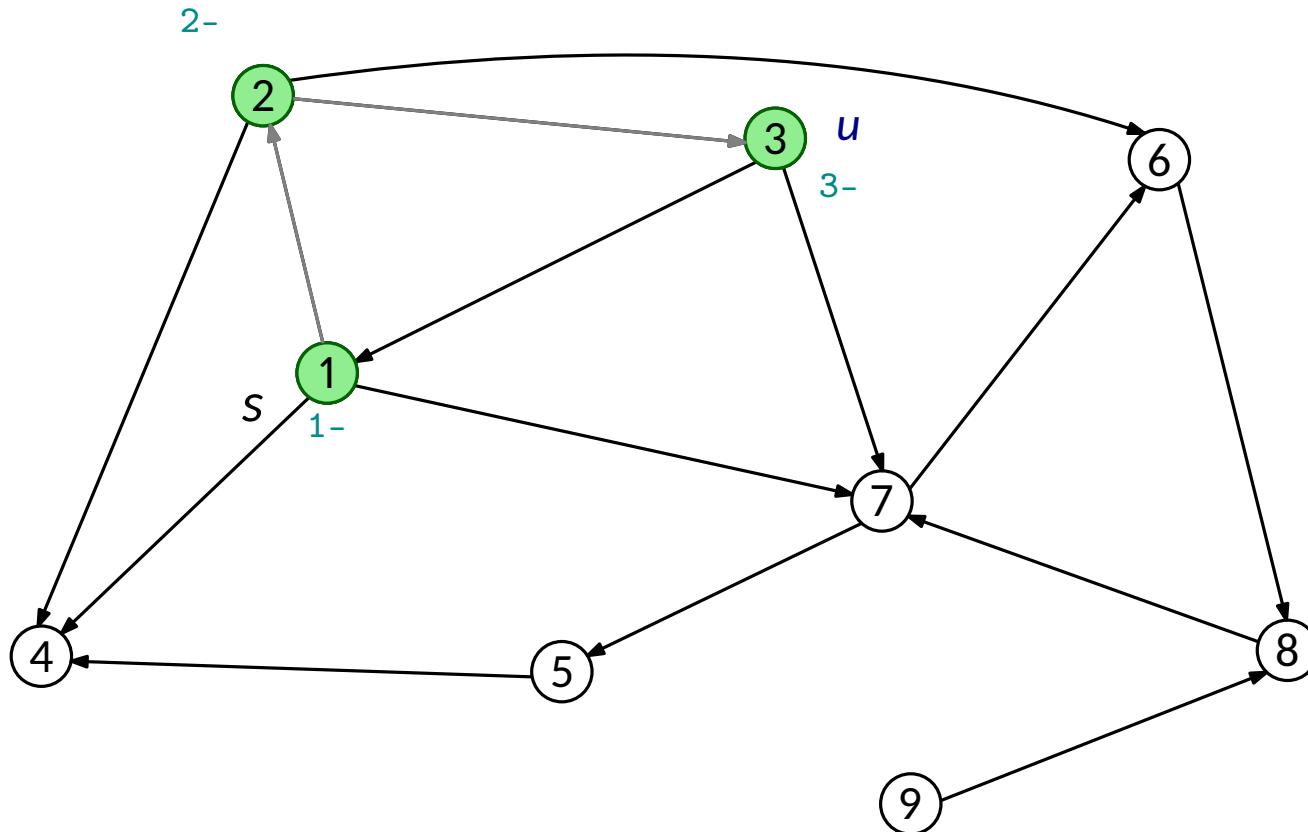
- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

1 2 3

# Beispiel für DFS

Startknoten:  $s = 1$

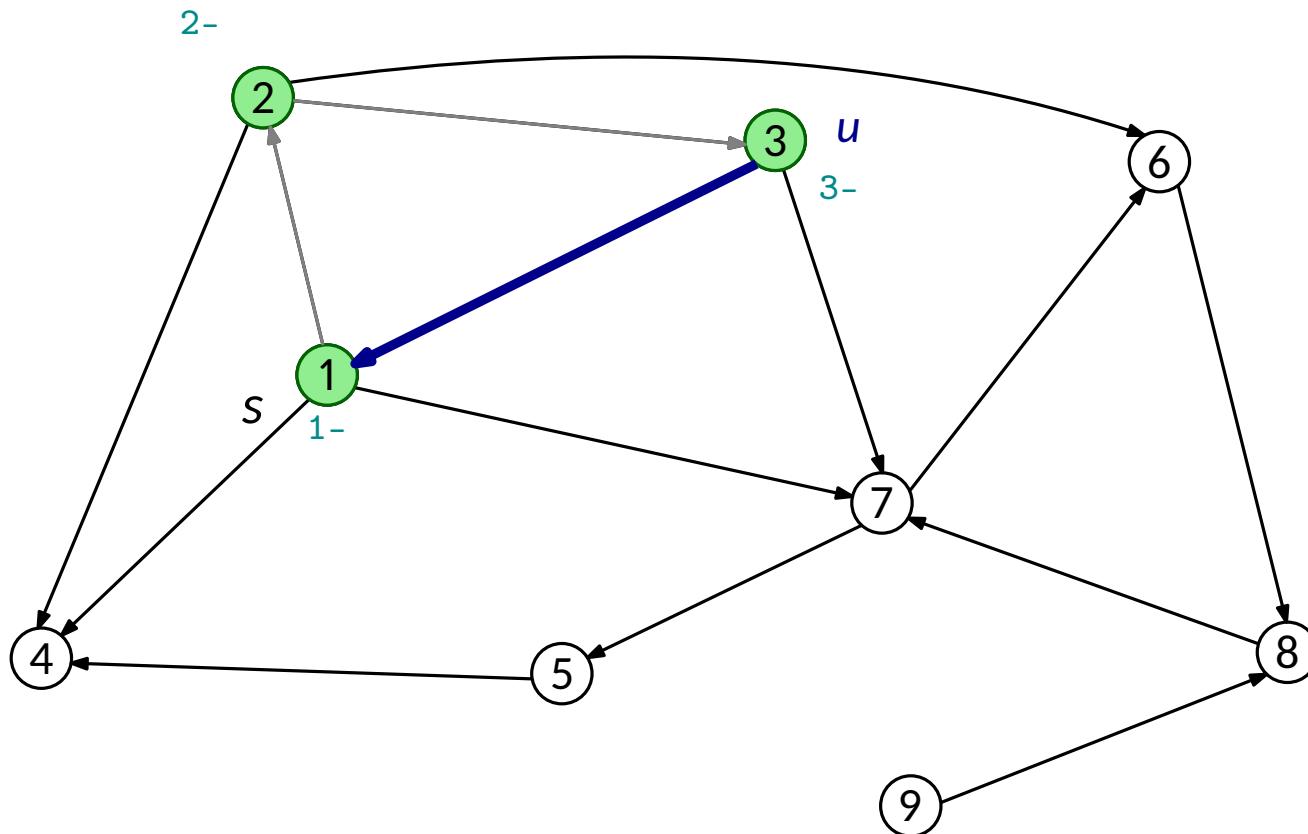


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2 3

# Beispiel für DFS

Startknoten:  $s = 1$



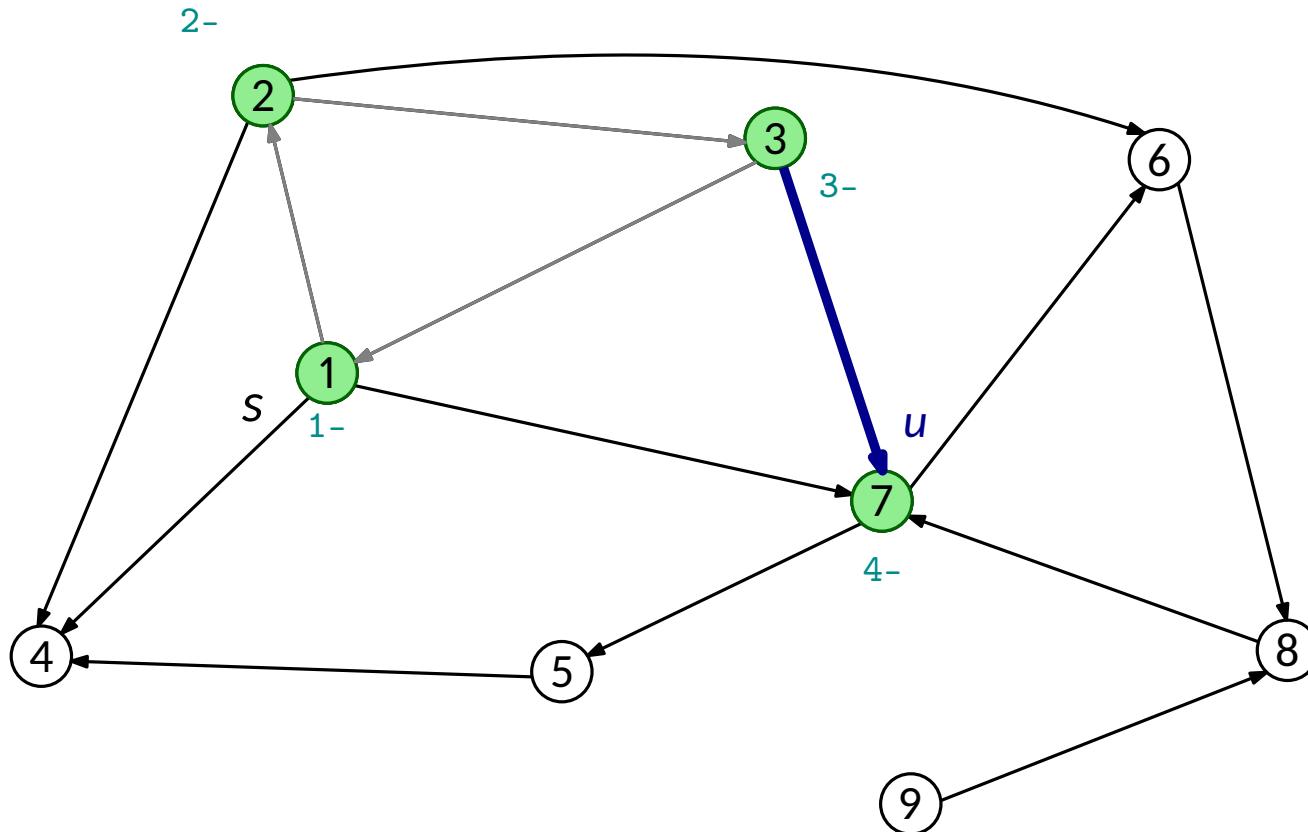
- unentdeckt
  - entdeckt
  - abgefertigt
- i-j Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

1 2 3

# Beispiel für DFS

Startknoten:  $s = 1$



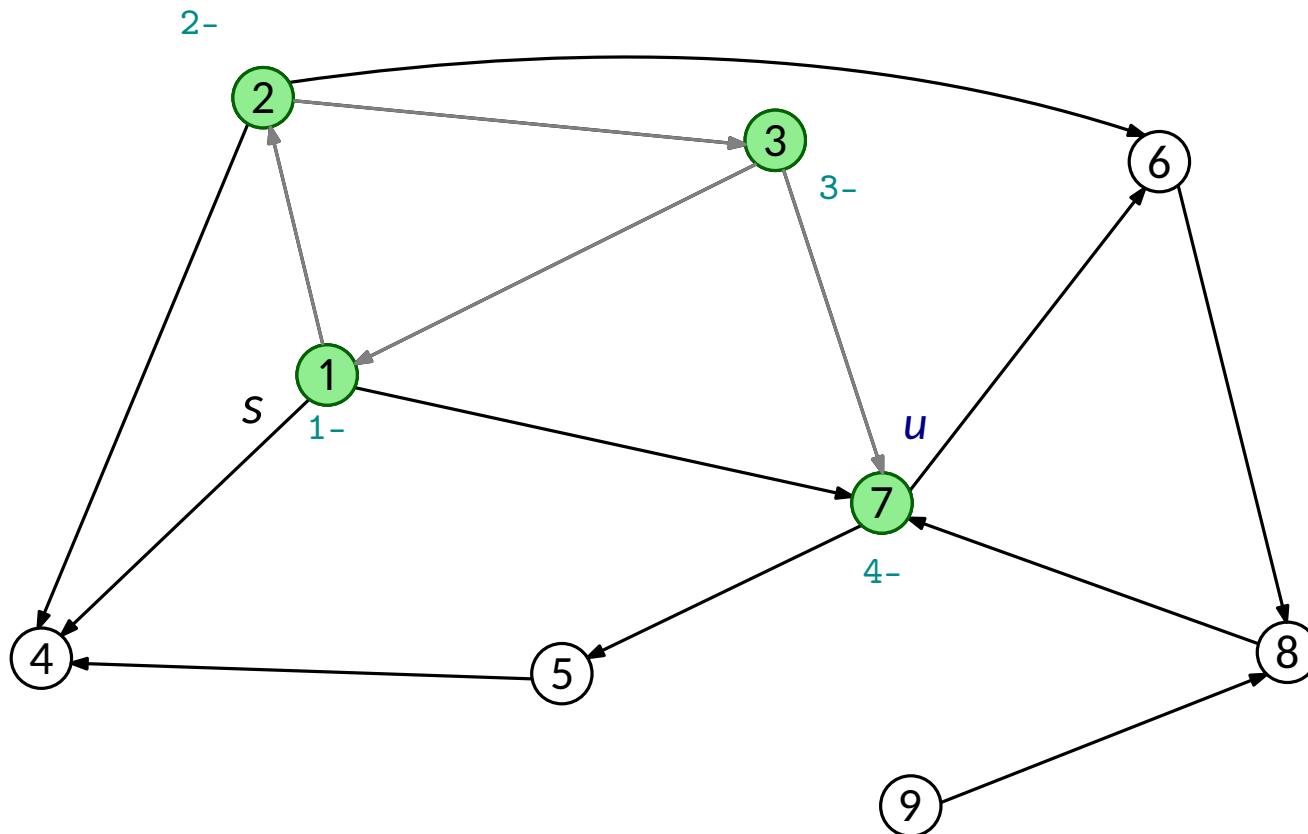
- unentdeckt
  - entdeckt
  - abgefertigt
- i-j Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

1 2 3 7

# Beispiel für DFS

Startknoten:  $s = 1$

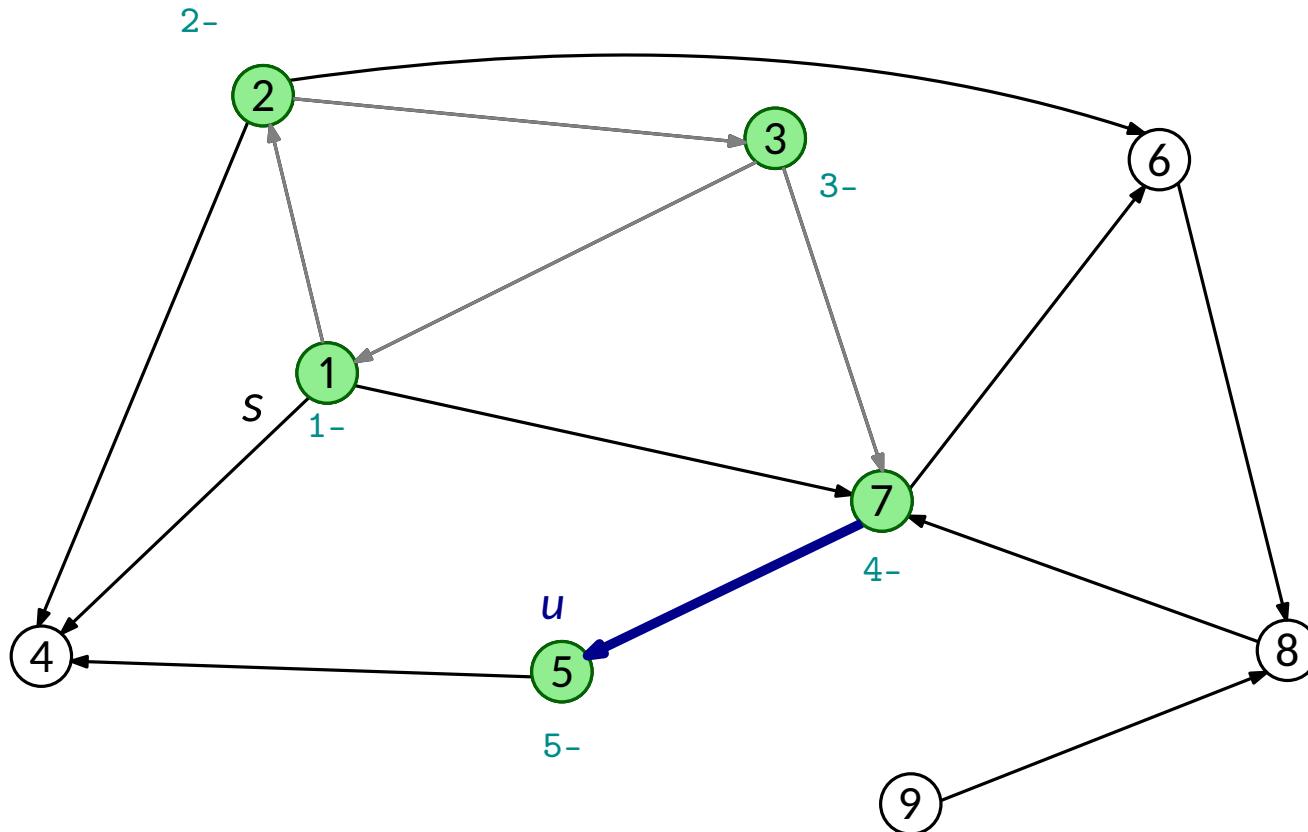


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2 3 7

# Beispiel für DFS

Startknoten:  $s = 1$



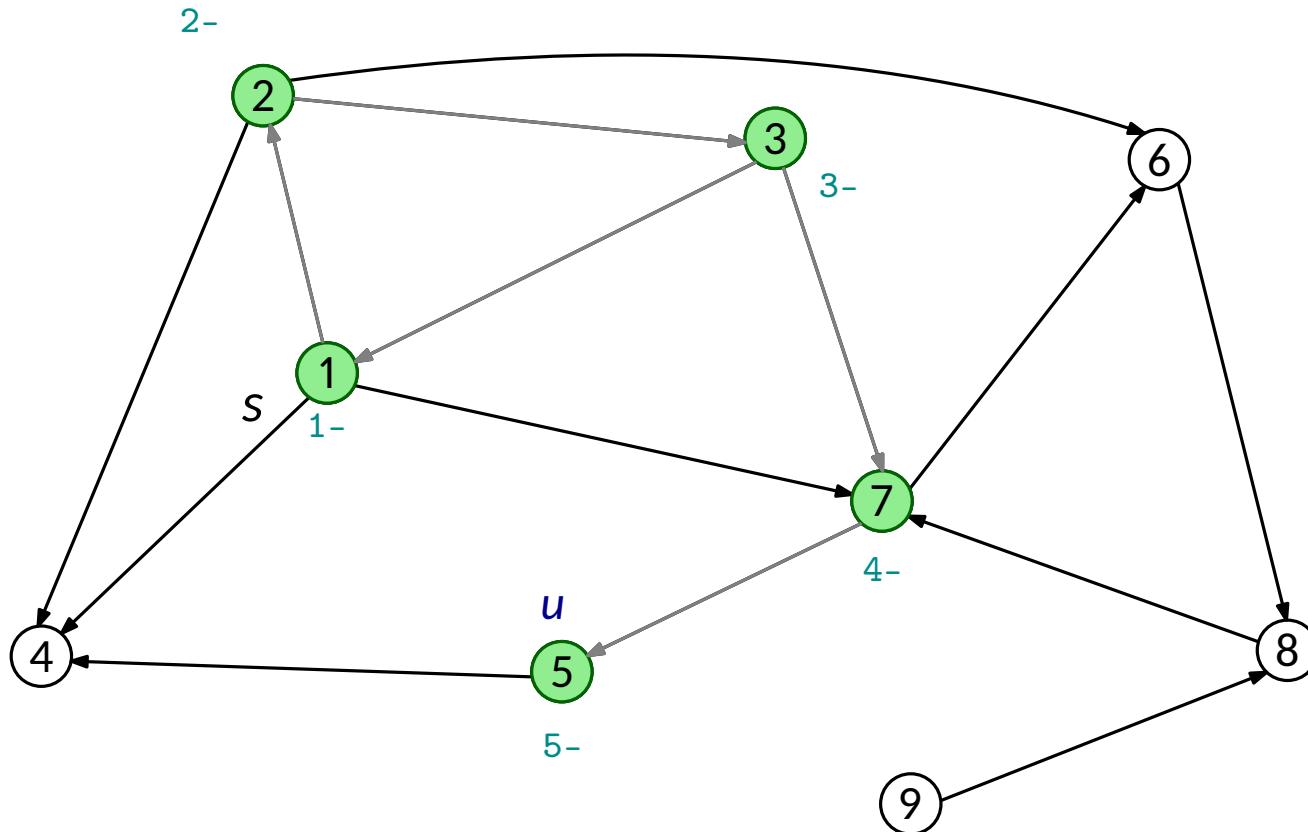
- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

1 2 3 7 5

# Beispiel für DFS

Startknoten:  $s = 1$



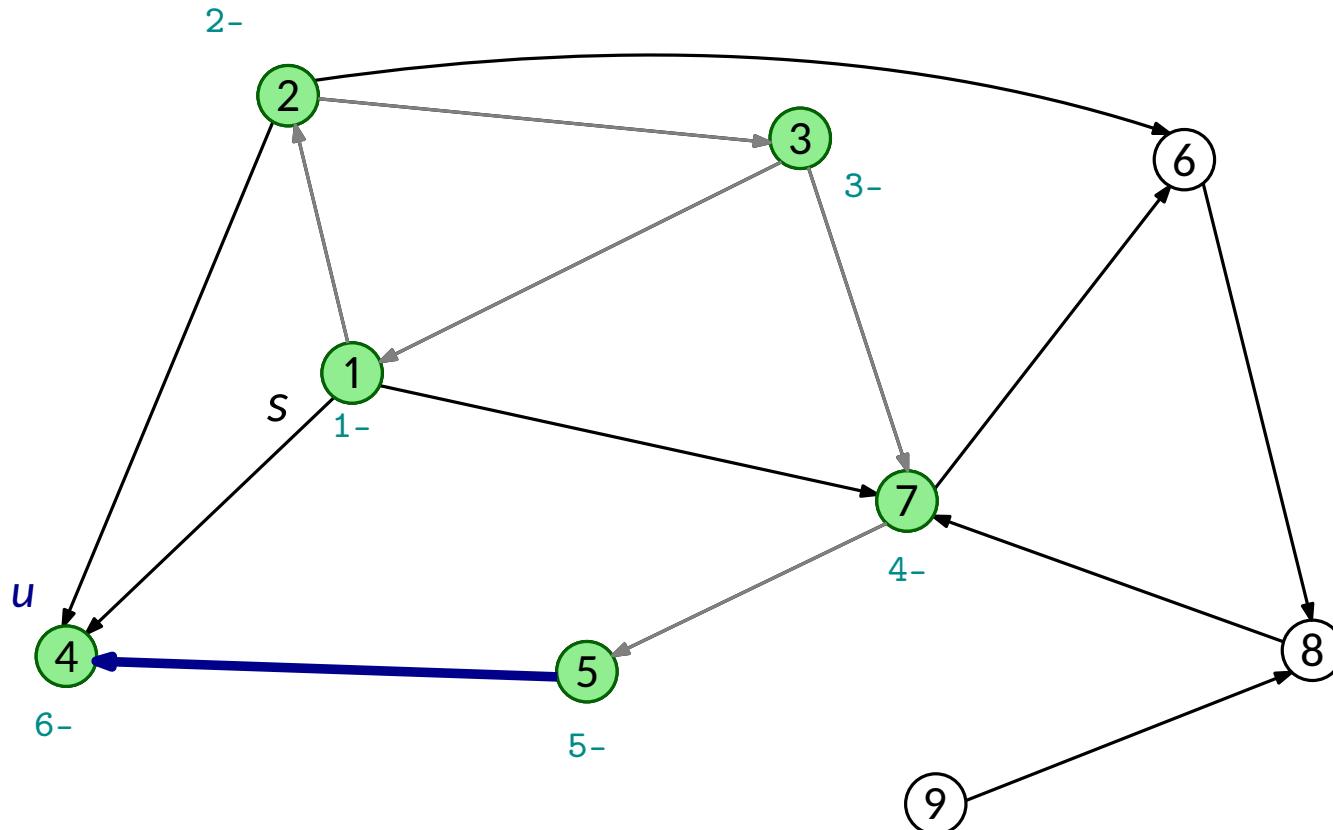
- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

1 2 3 7 5

# Beispiel für DFS

Startknoten:  $s = 1$

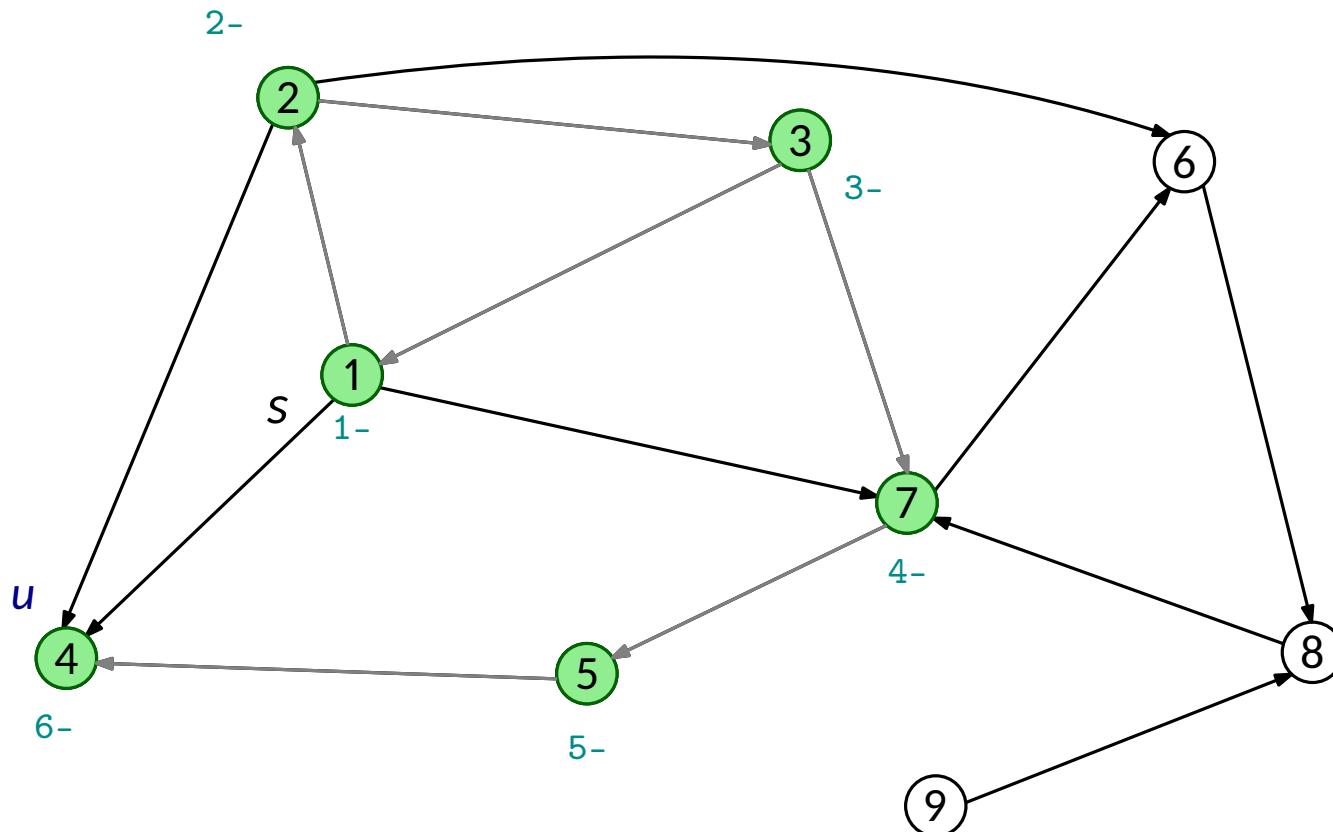


- unentdeckt (white circle)
  - entdeckt (light green circle)
  - abgefertigt (dark grey circle)
- i-j Lebenszeit von i bis j (cyan text)

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

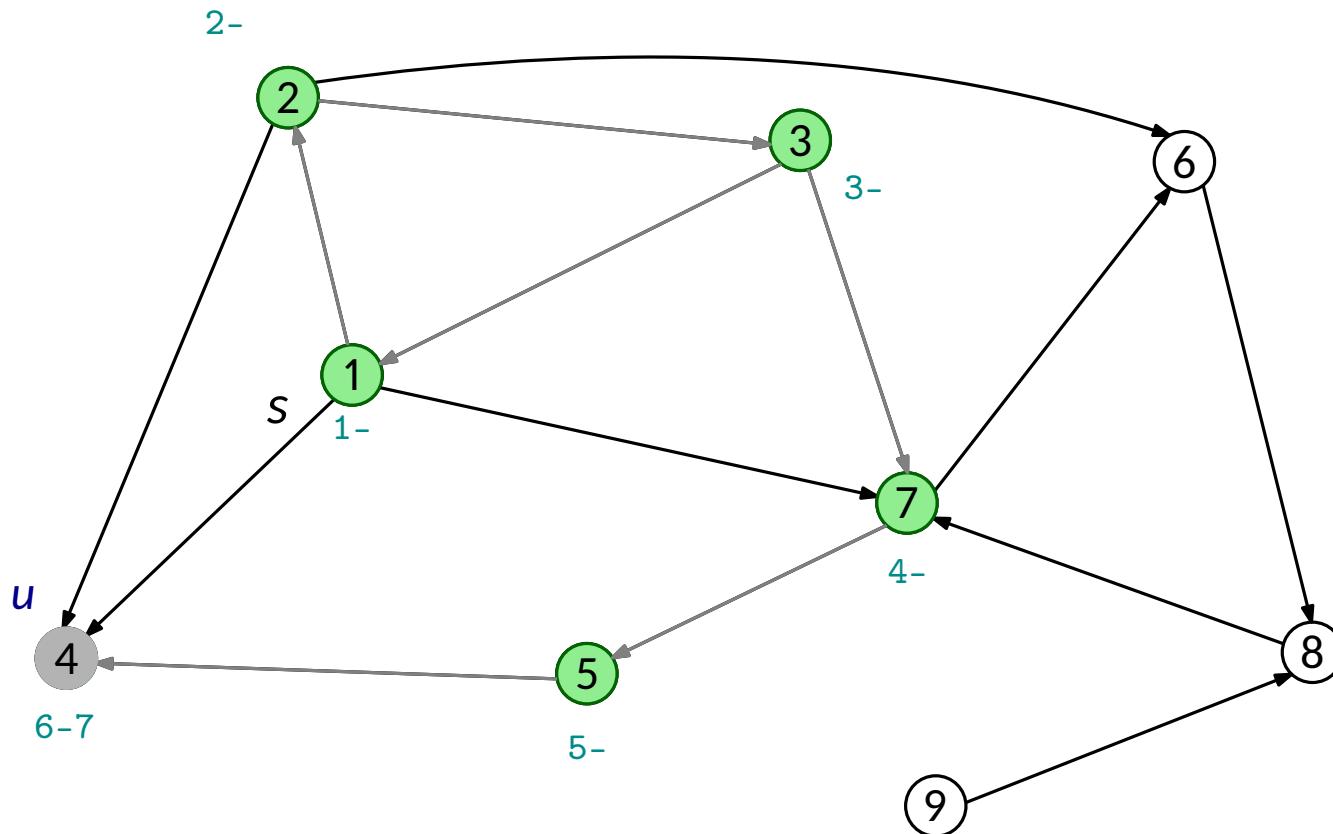


- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

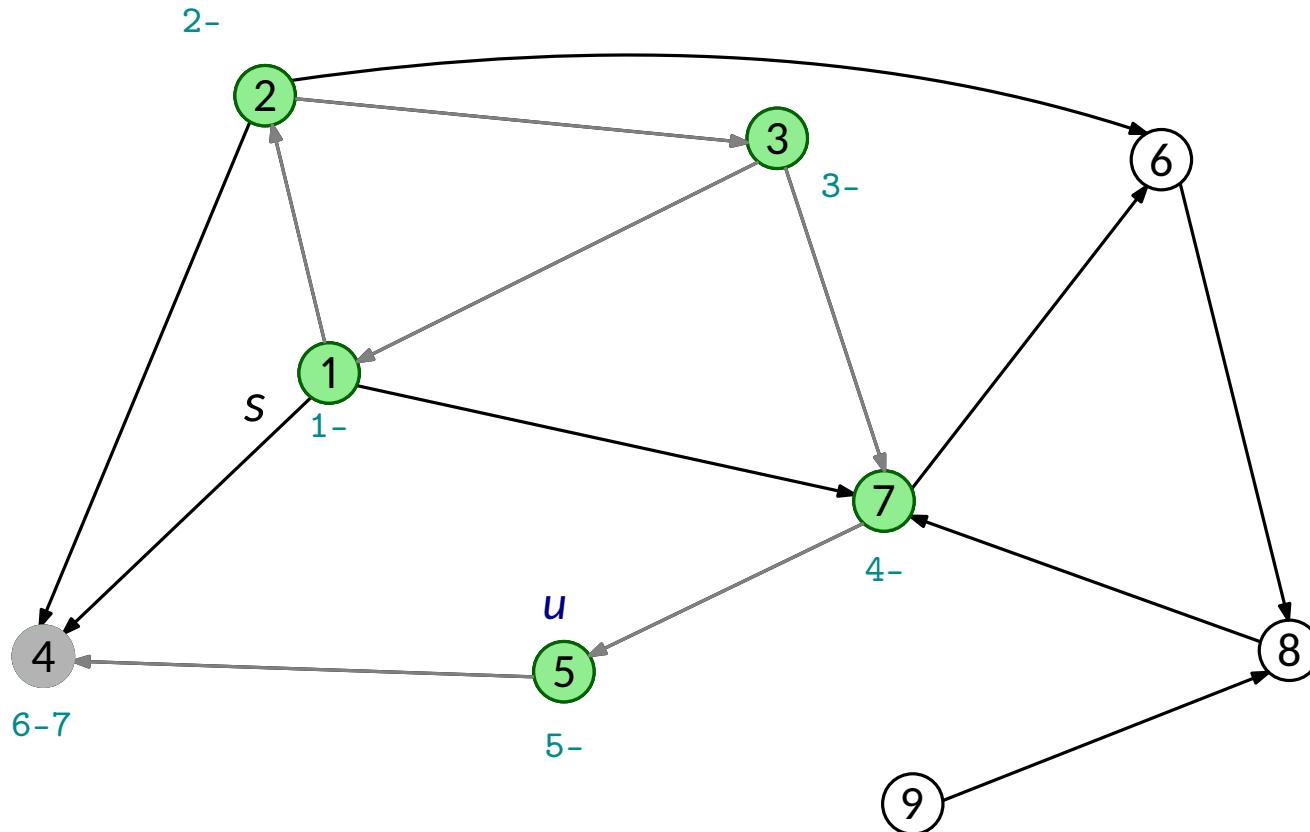


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- i-j Lebenszeit von i bis j (teal text)

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

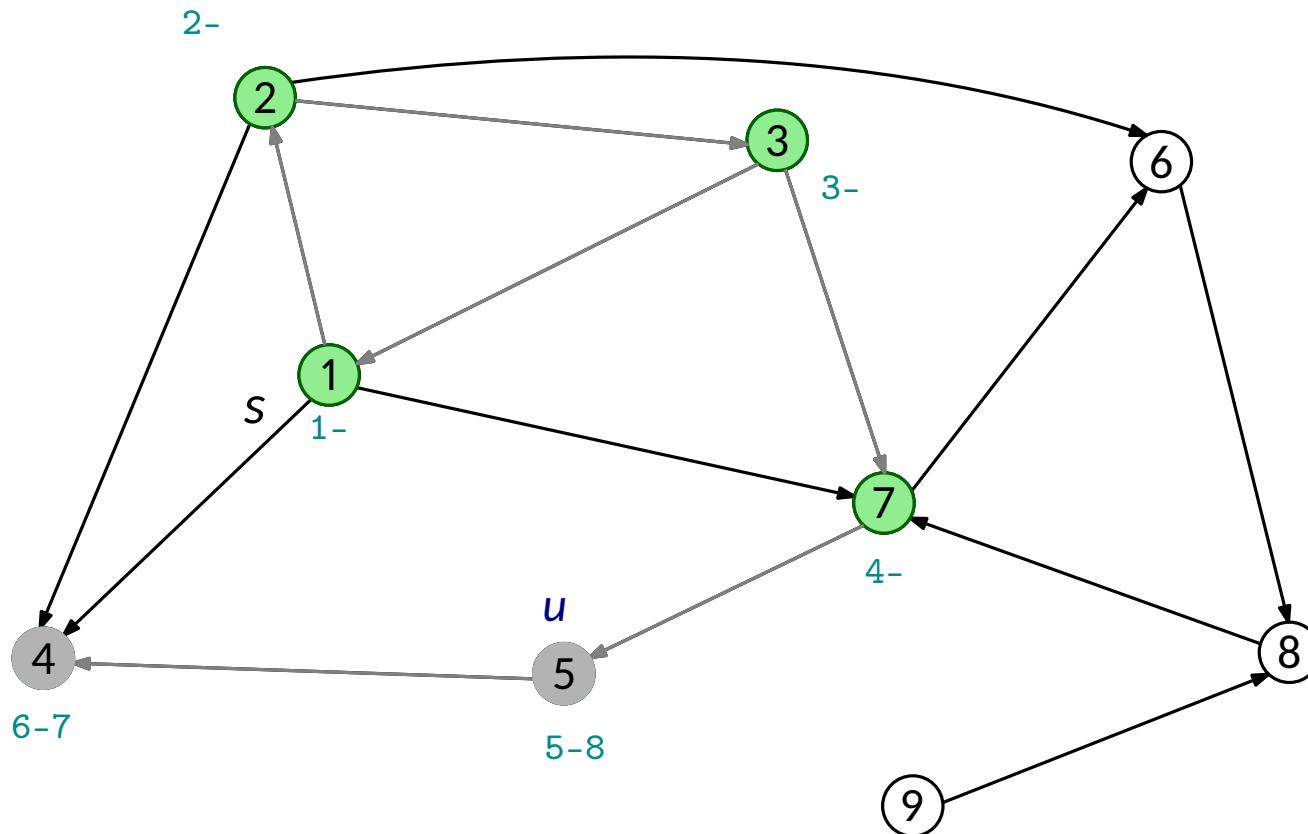


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$



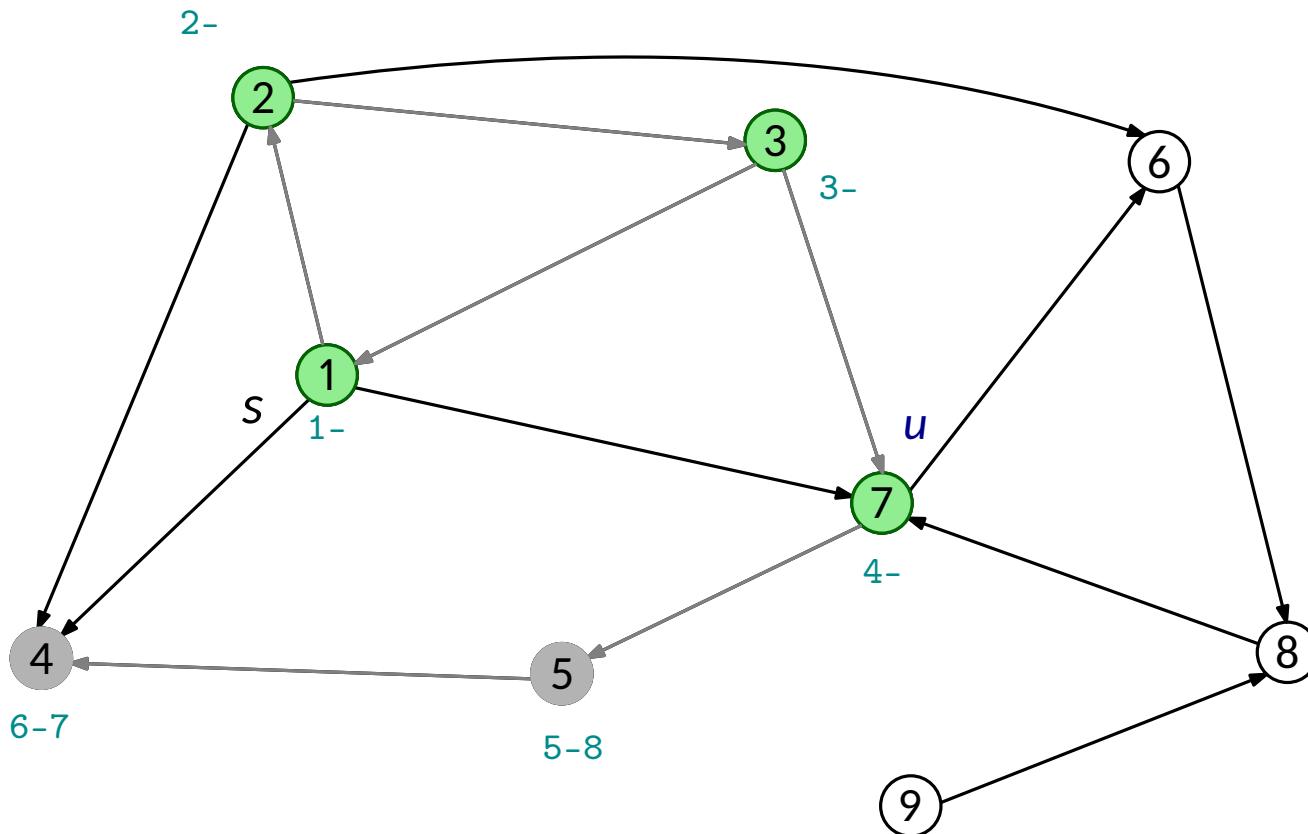
- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

1 2 3 7 5

# Beispiel für DFS

Startknoten:  $s = 1$

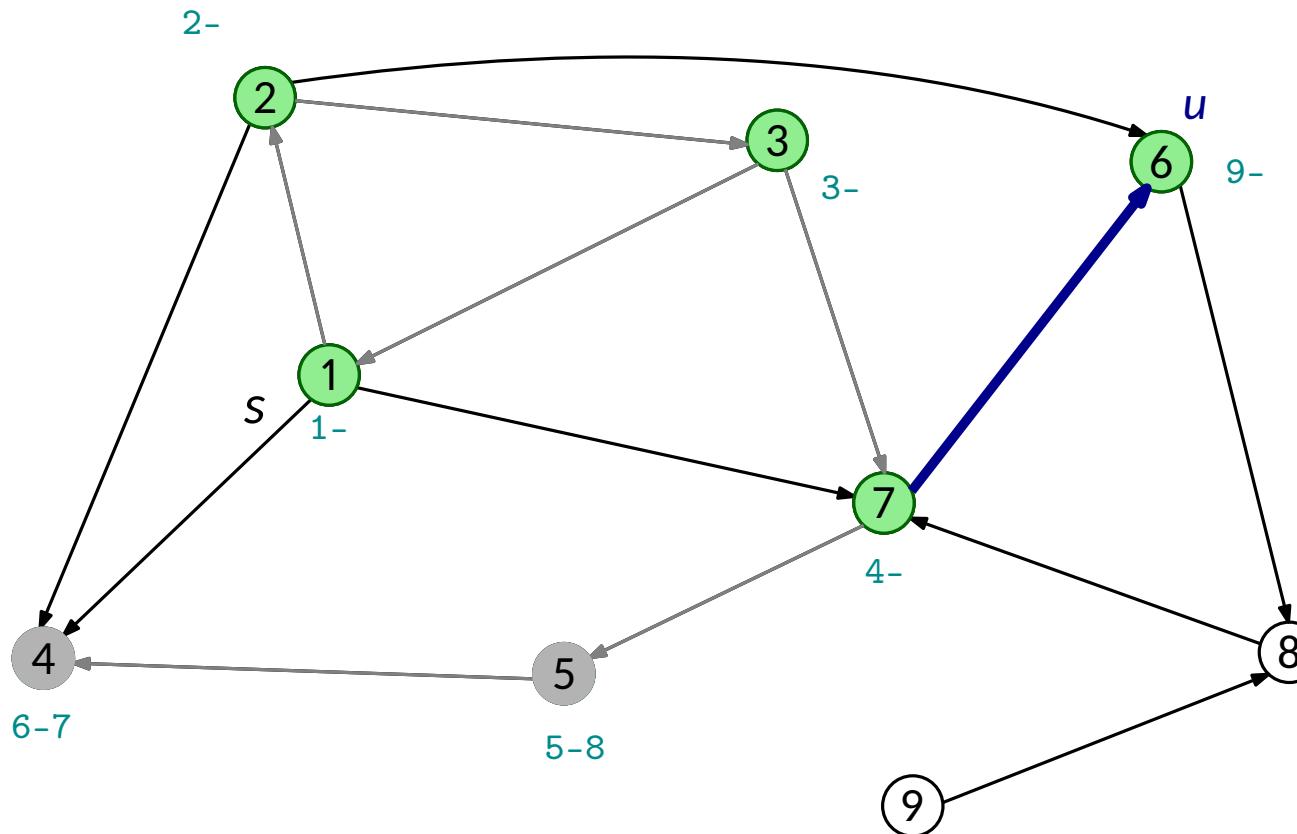


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2 3 7

# Beispiel für DFS

Startknoten:  $s = 1$



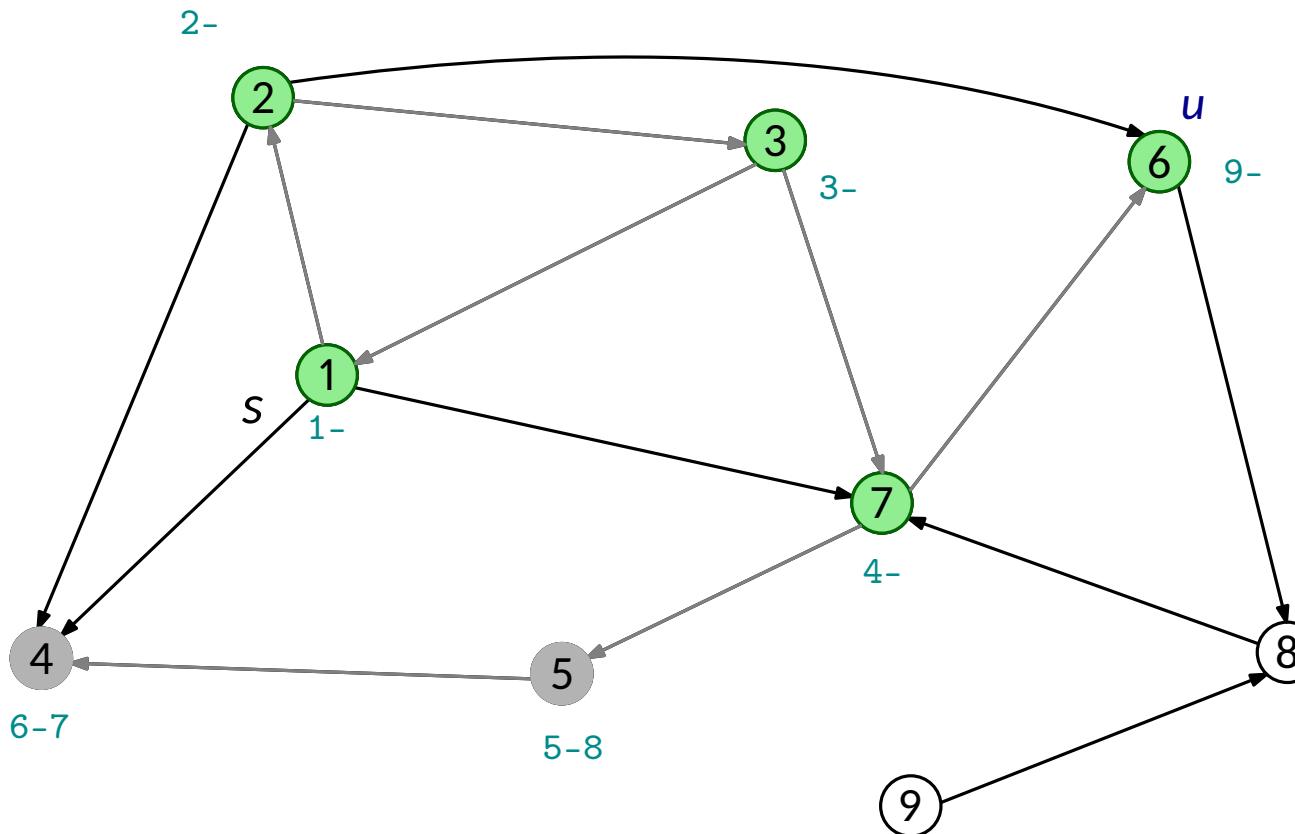
- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

1 2 3 7 6

# Beispiel für DFS

Startknoten:  $s = 1$

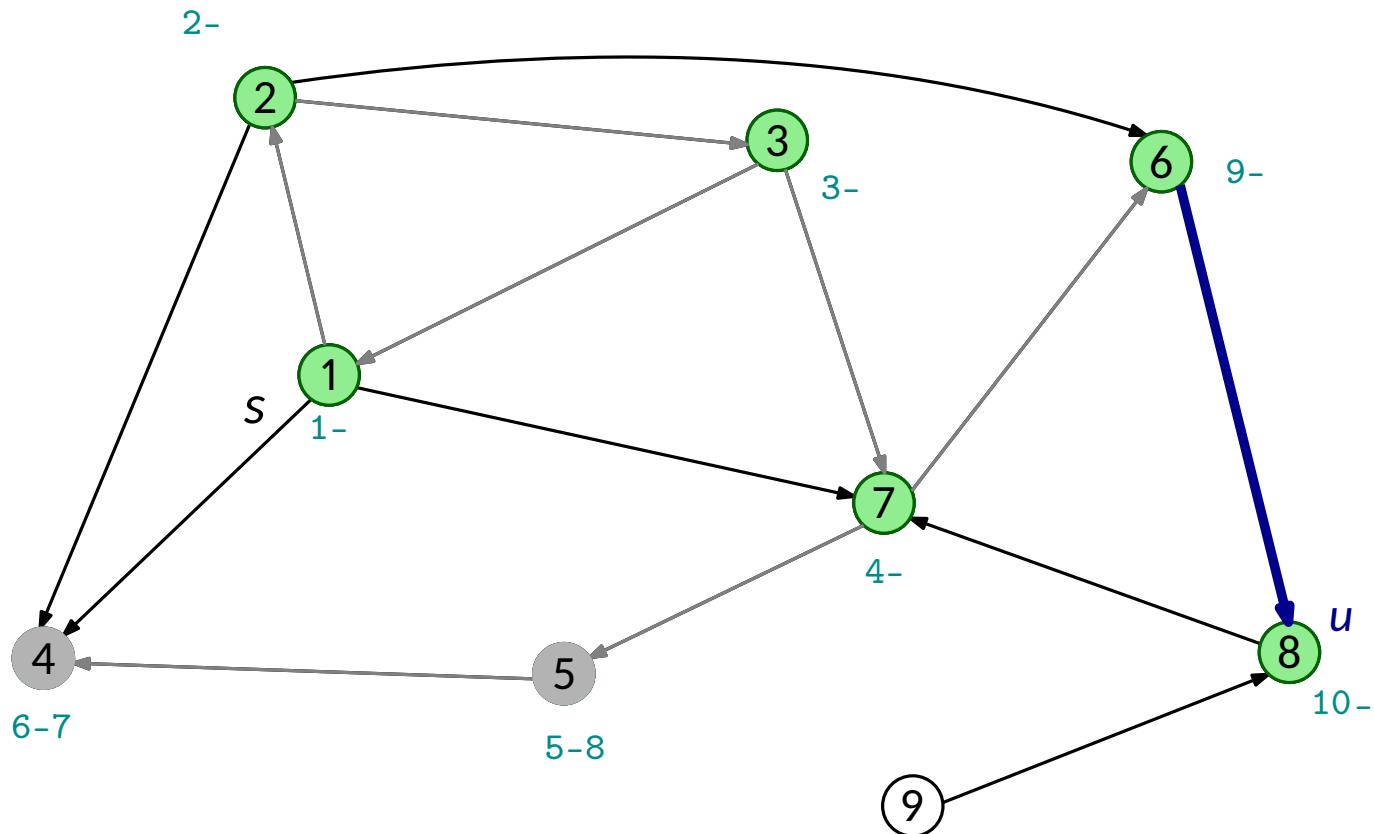


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit) 1 2 3 7 6

# Beispiel für DFS

Startknoten:  $s = 1$

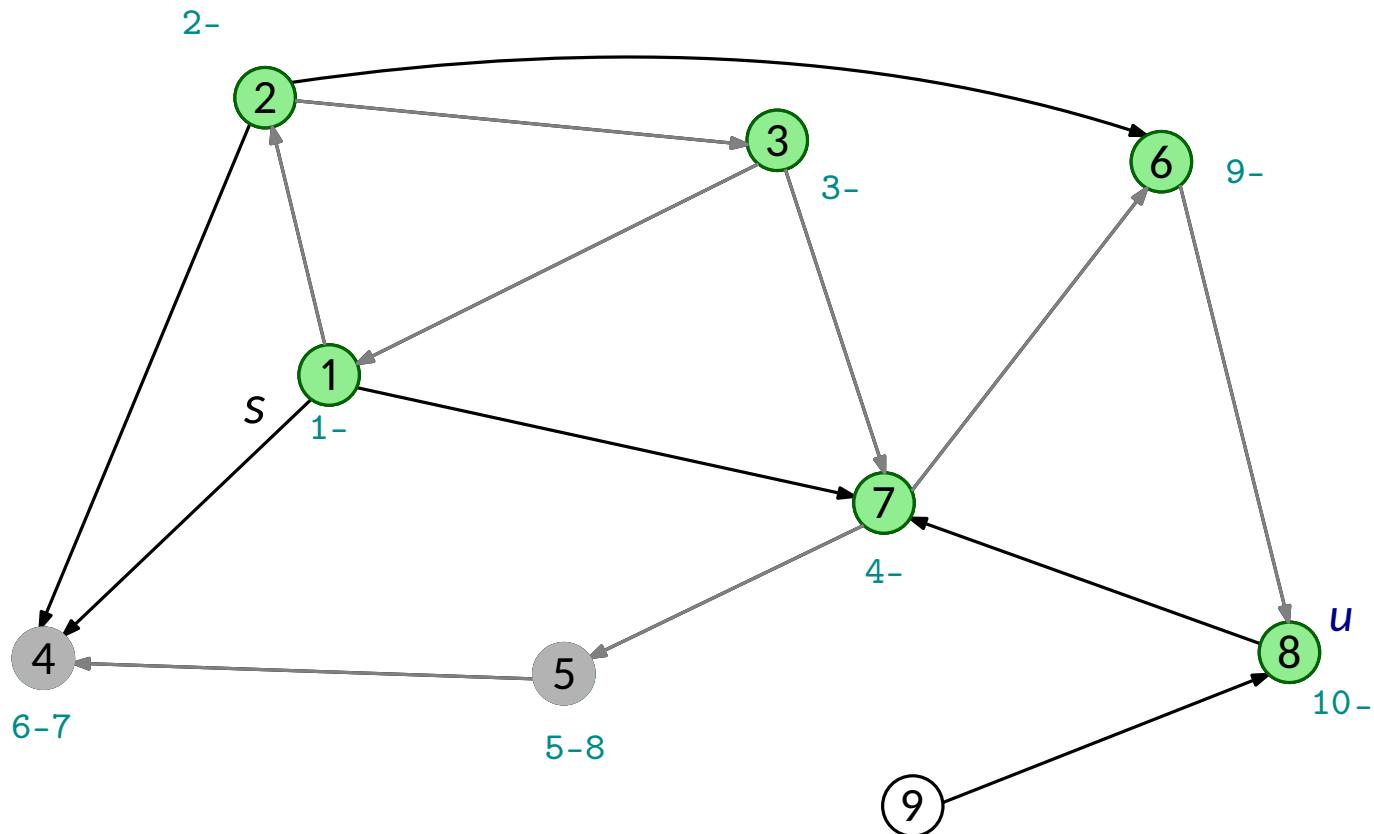


- unentdeckt (white circle)
  - entdeckt (light green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

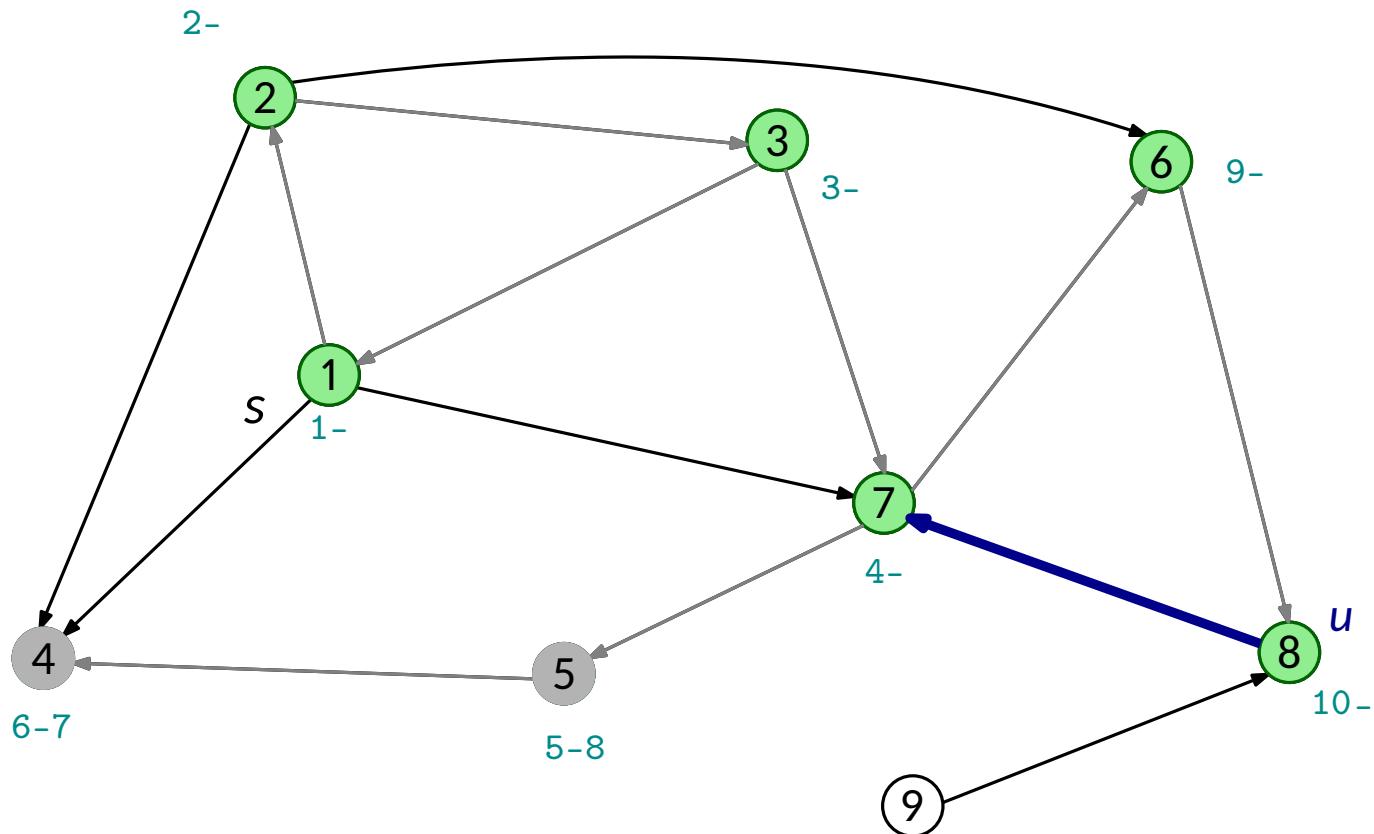


- unentdeckt (white circle)
  - entdeckt (light green circle)
  - abgefertigt (grey circle)
- i-j Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

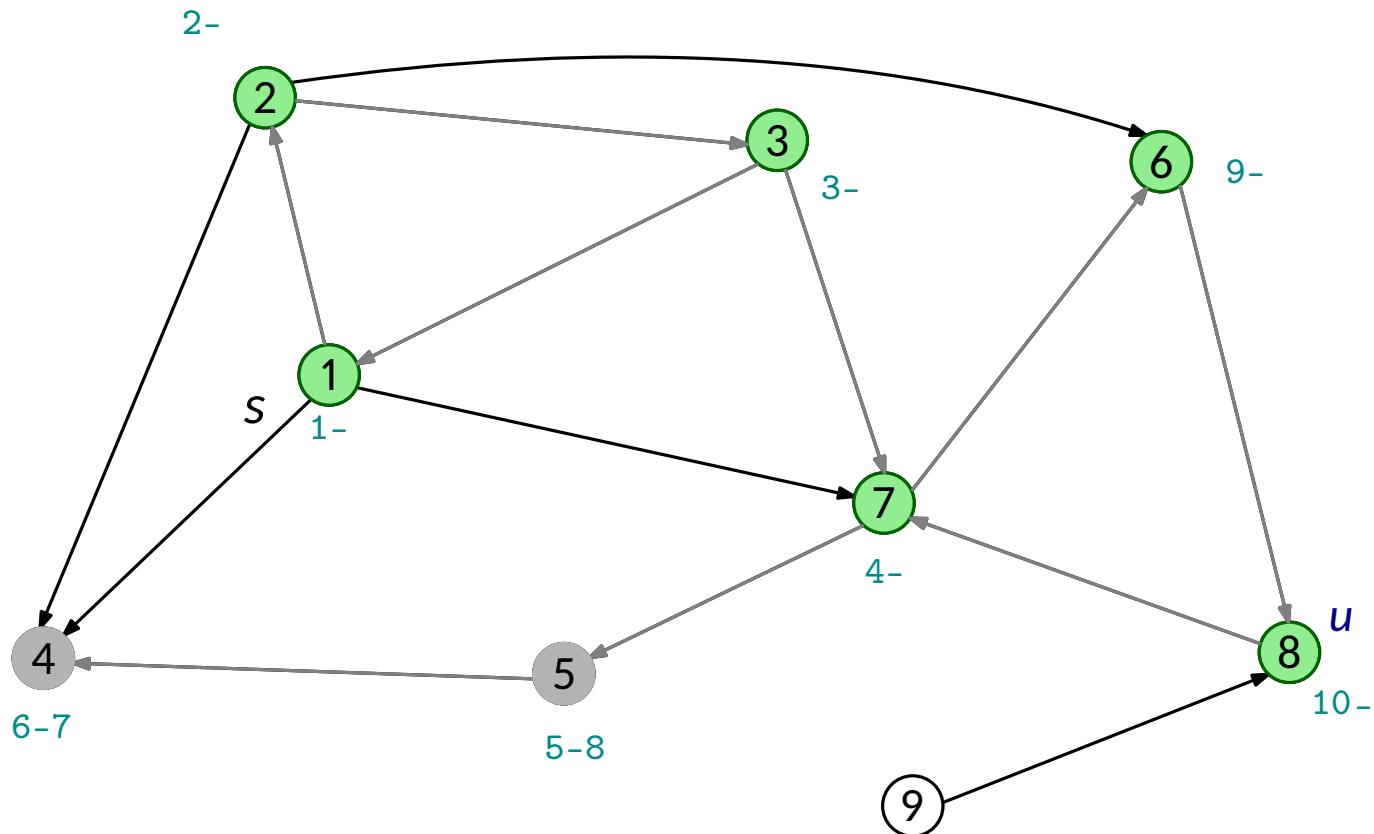


- unentdeckt (white circle)
  - entdeckt (light green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

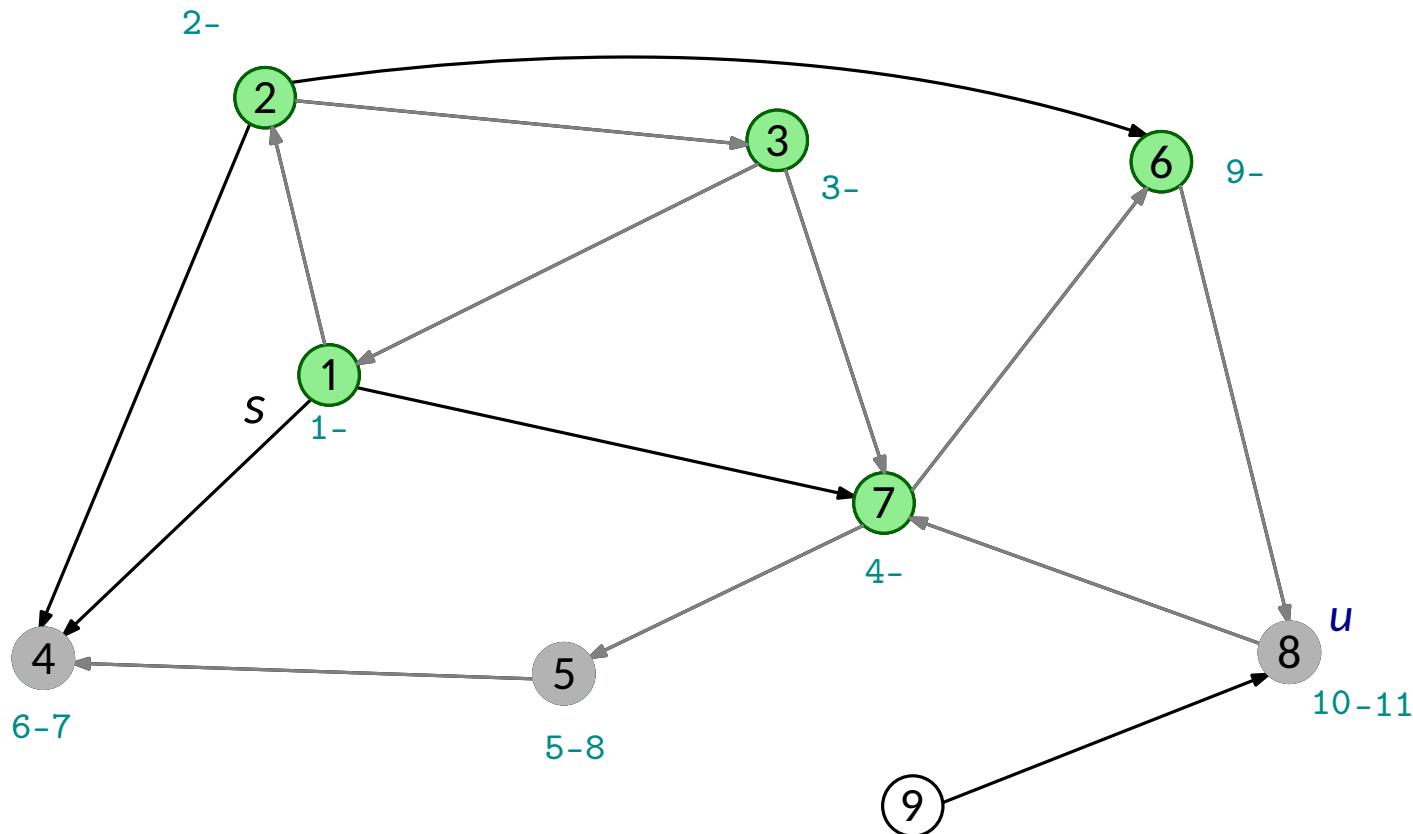


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

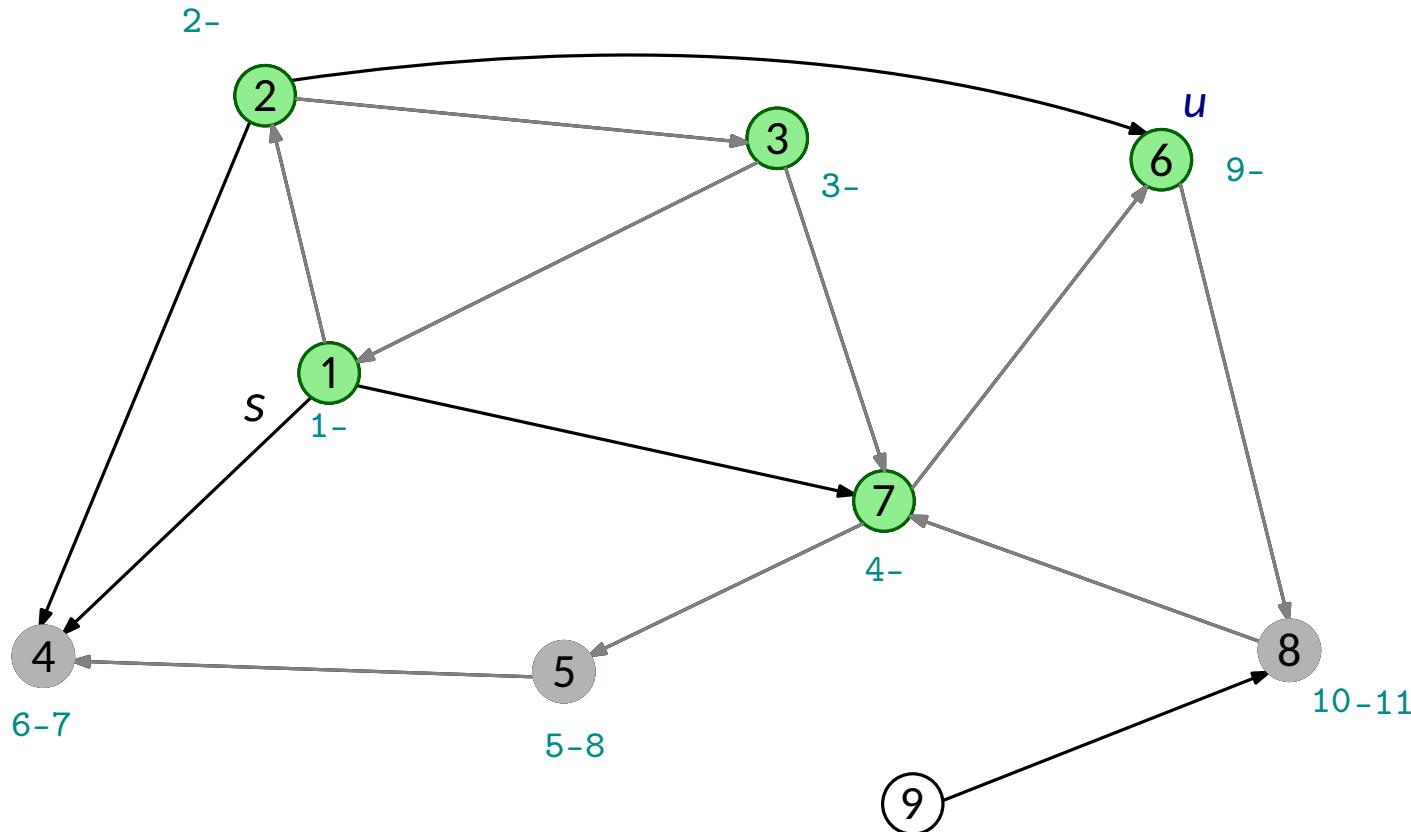


- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$



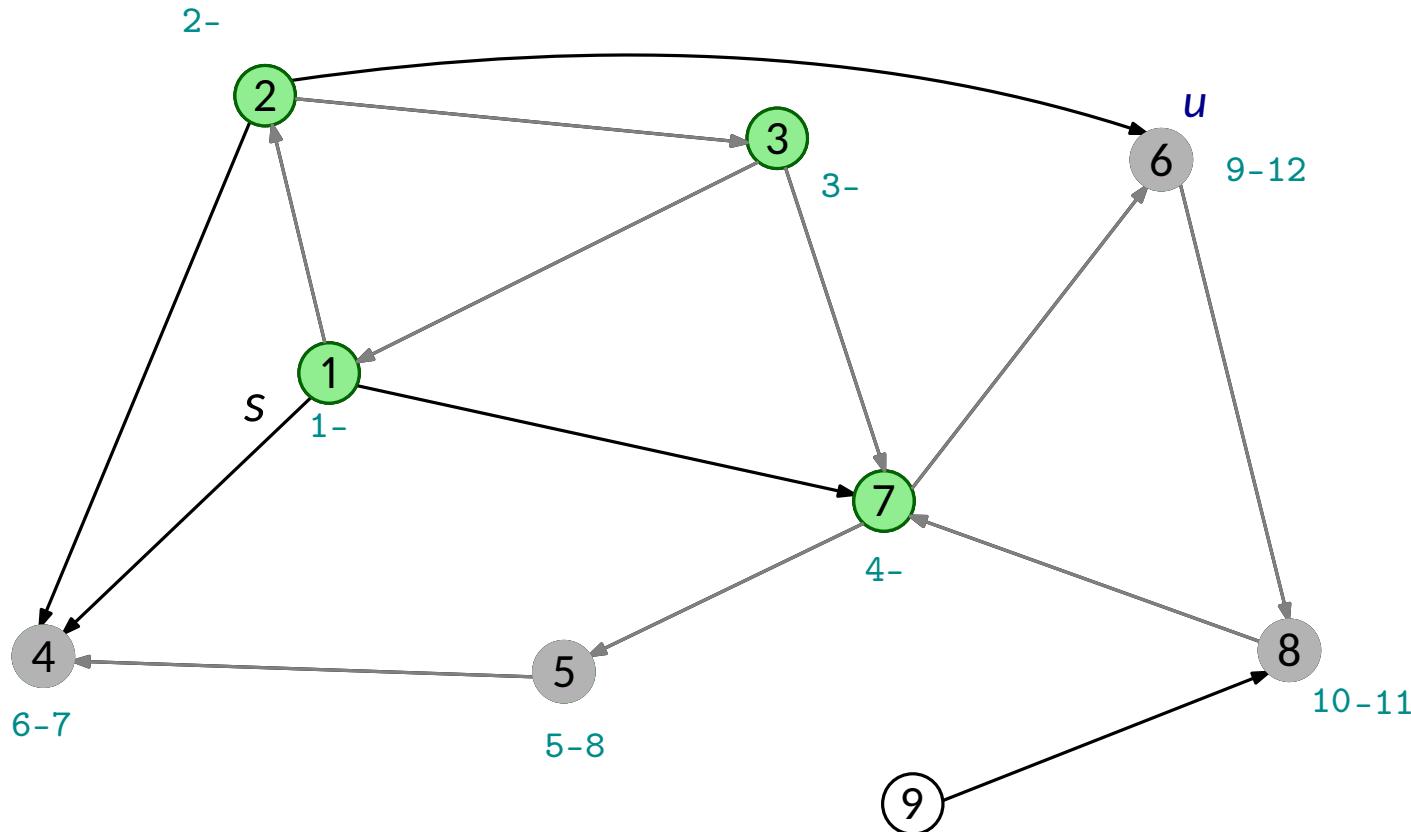
- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

1 2 3 7 6

# Beispiel für DFS

Startknoten:  $s = 1$

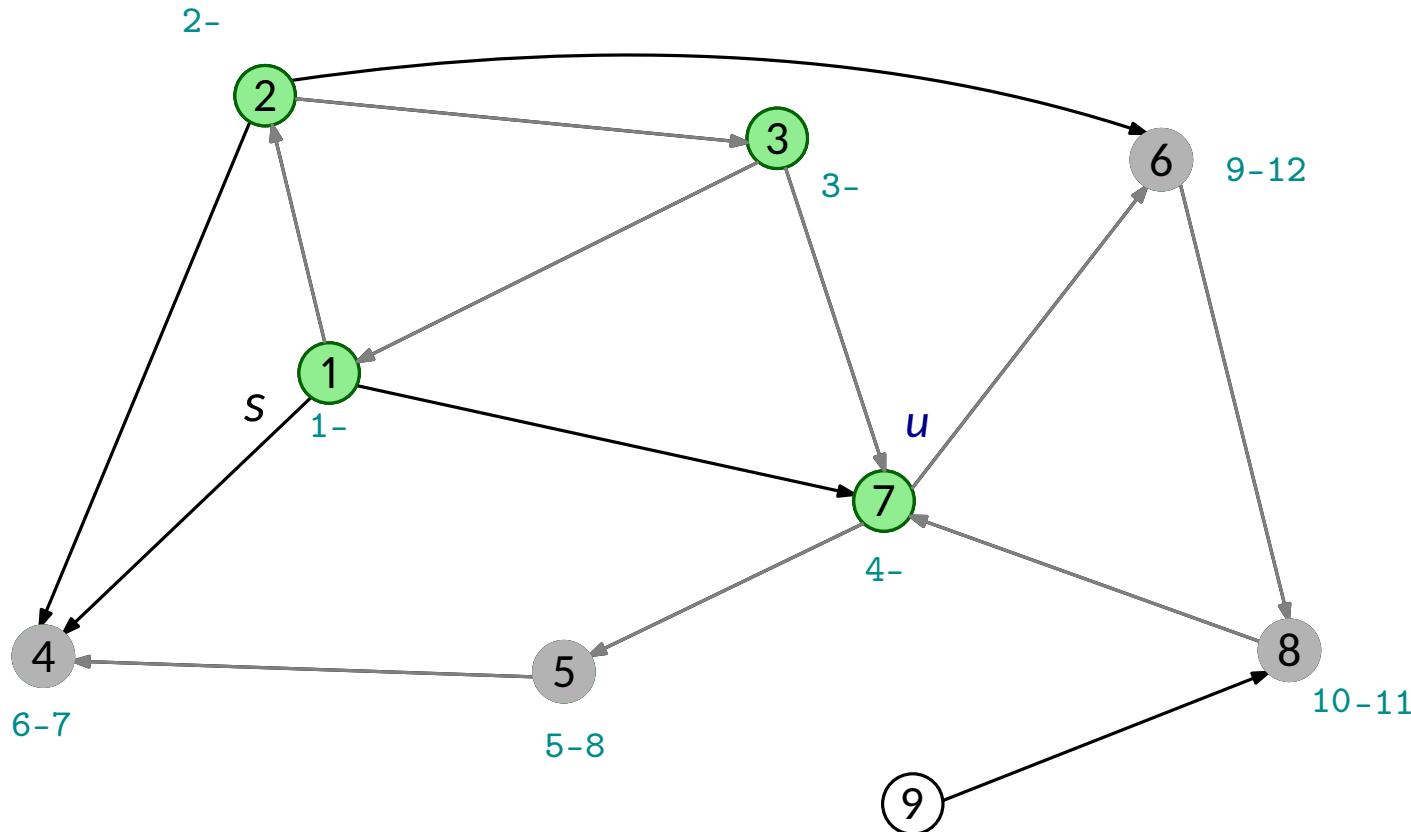


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

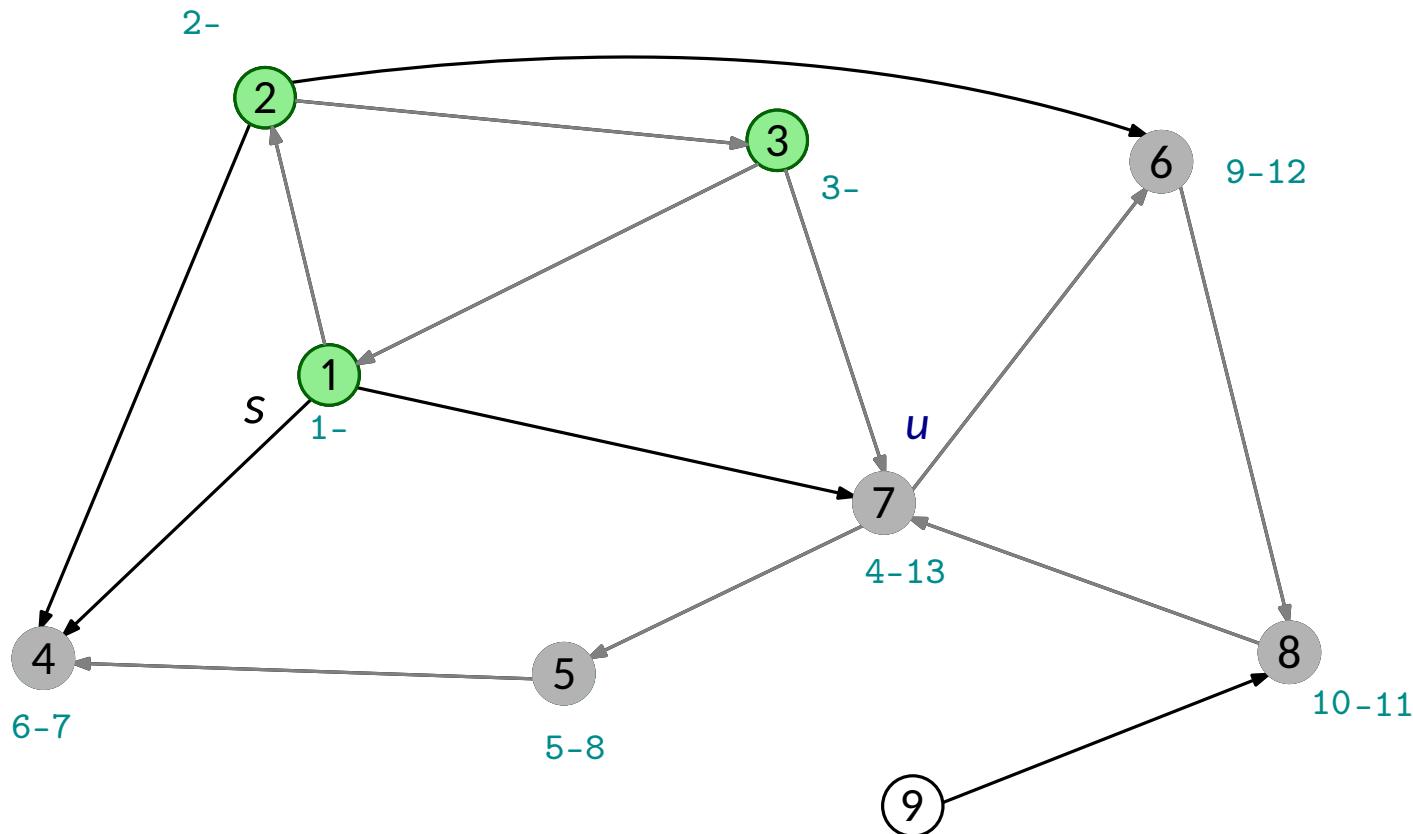


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2 3 7

# Beispiel für DFS

Startknoten:  $s = 1$

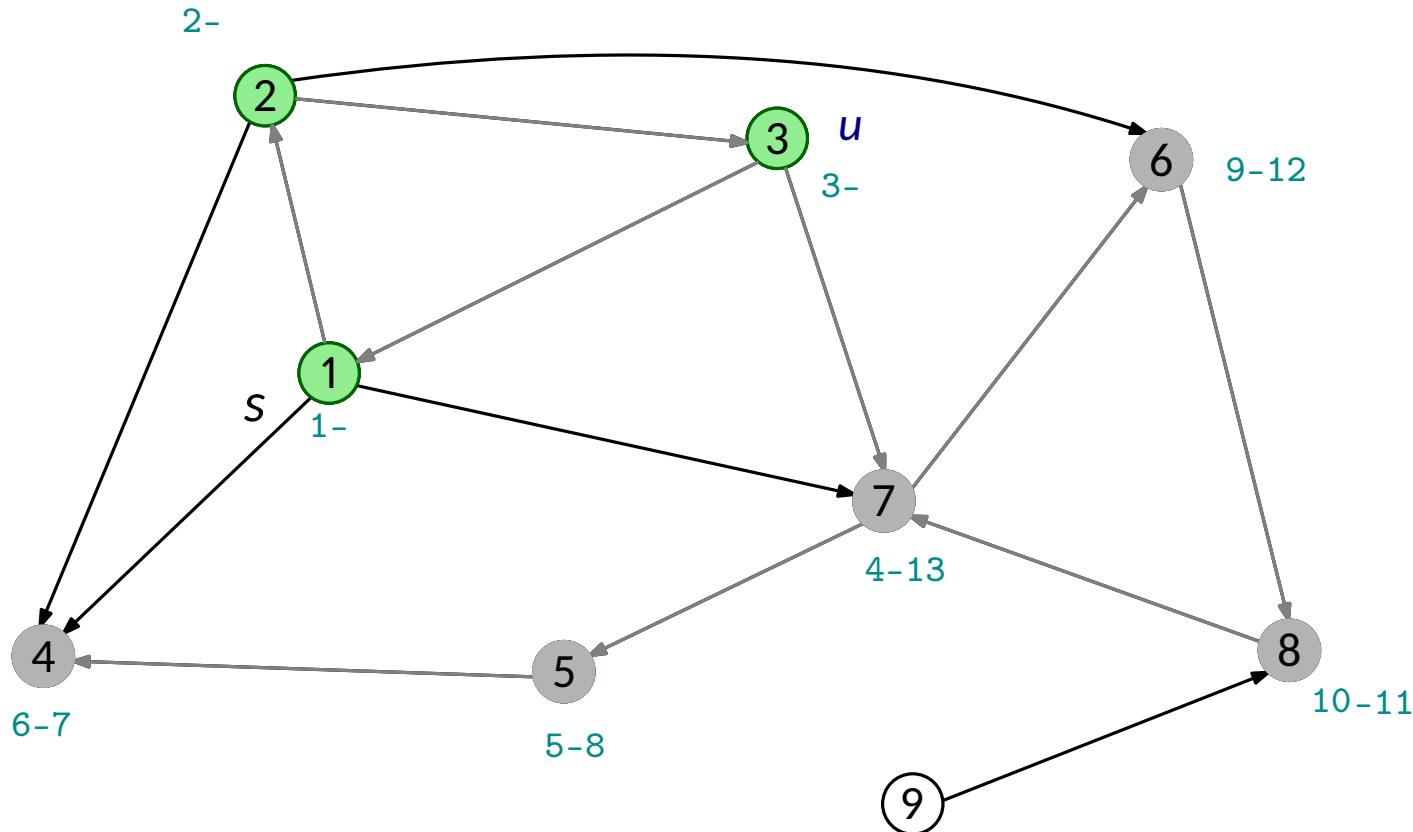


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2 3 7

# Beispiel für DFS

Startknoten:  $s = 1$

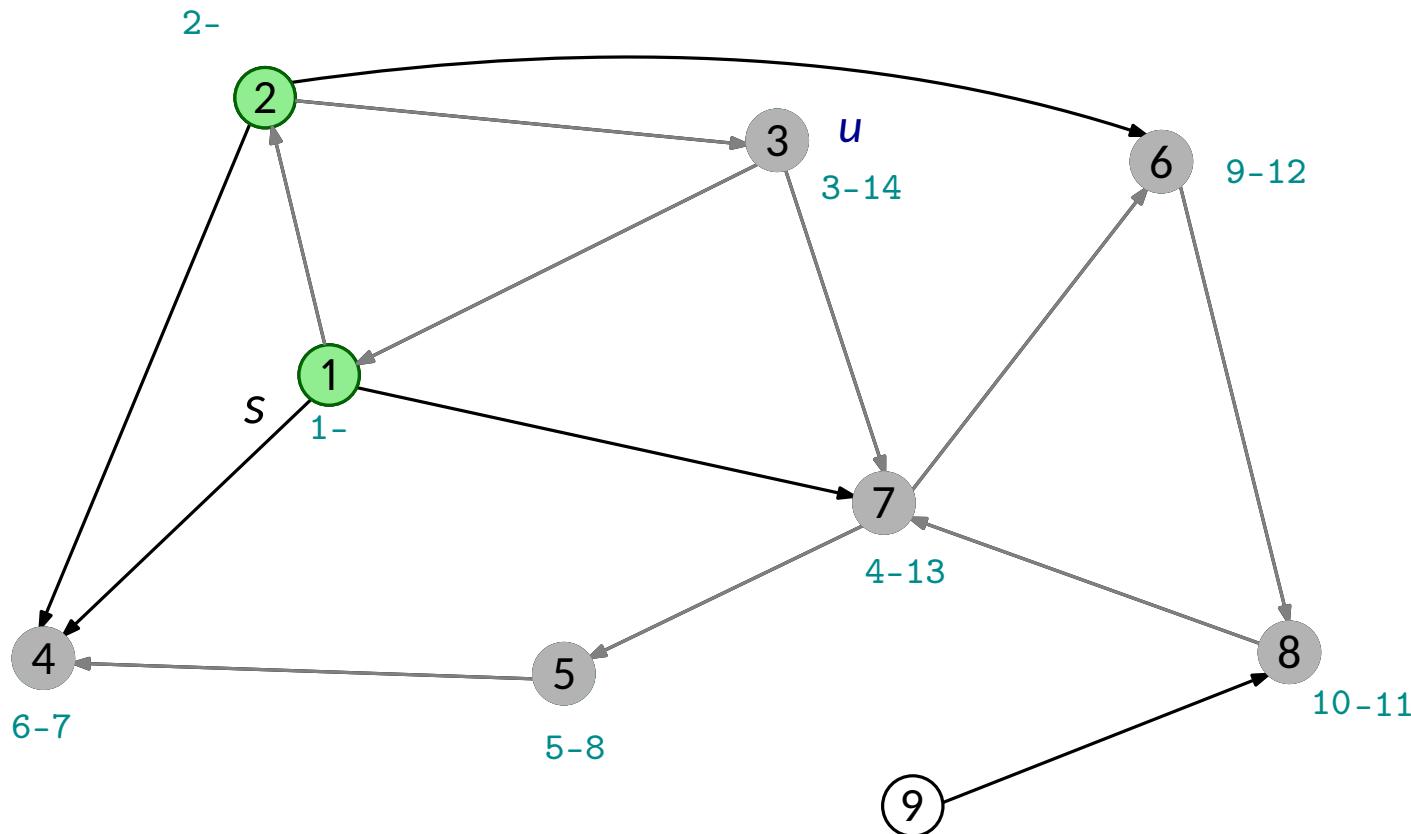


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2 3

# Beispiel für DFS

Startknoten:  $s = 1$

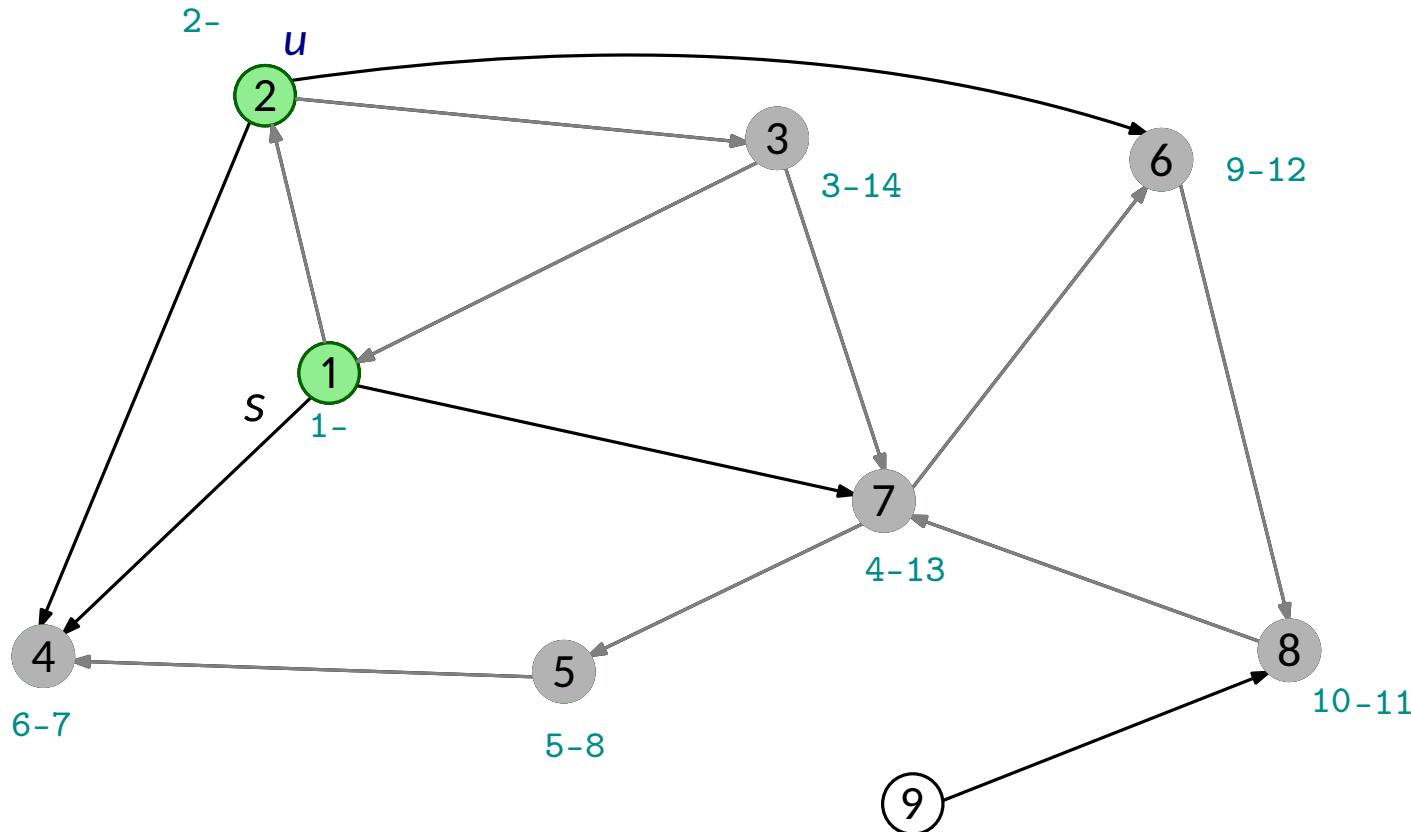


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit) 1 2 3

# Beispiel für DFS

Startknoten:  $s = 1$

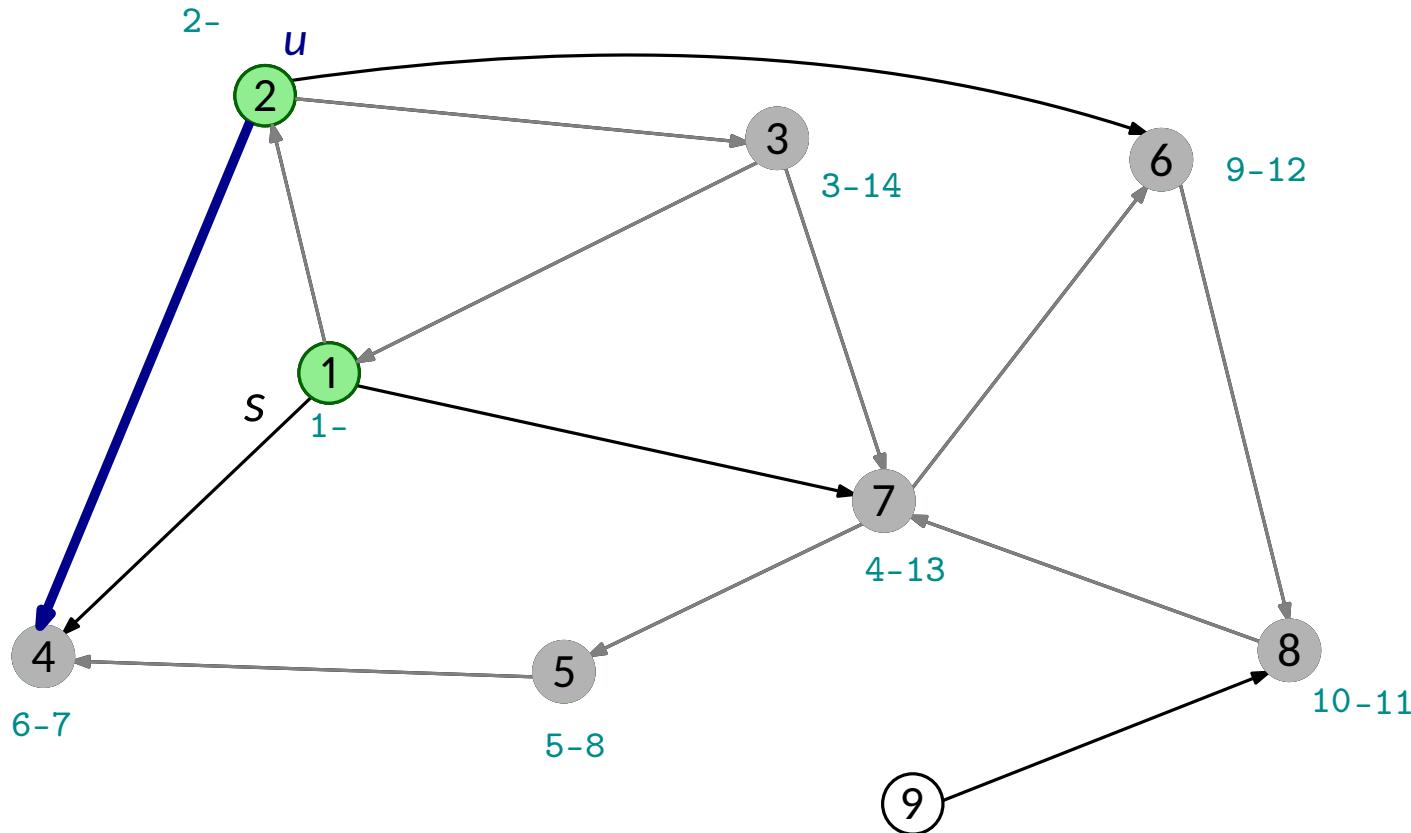


- unentdeckt
  - entdeckt
  - abgefertigt
- i-j Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

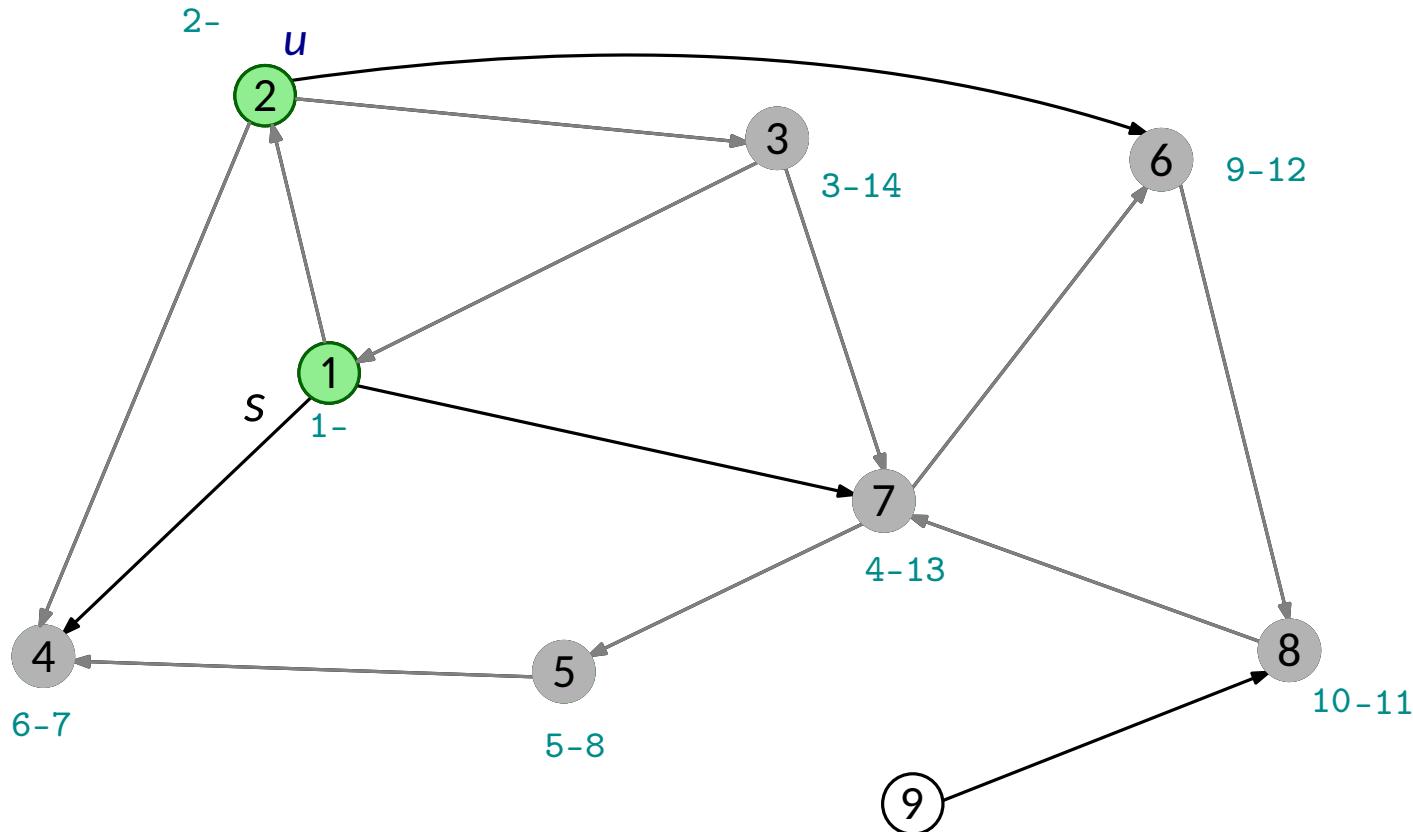


- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

# Beispiel für DFS

Startknoten:  $s = 1$

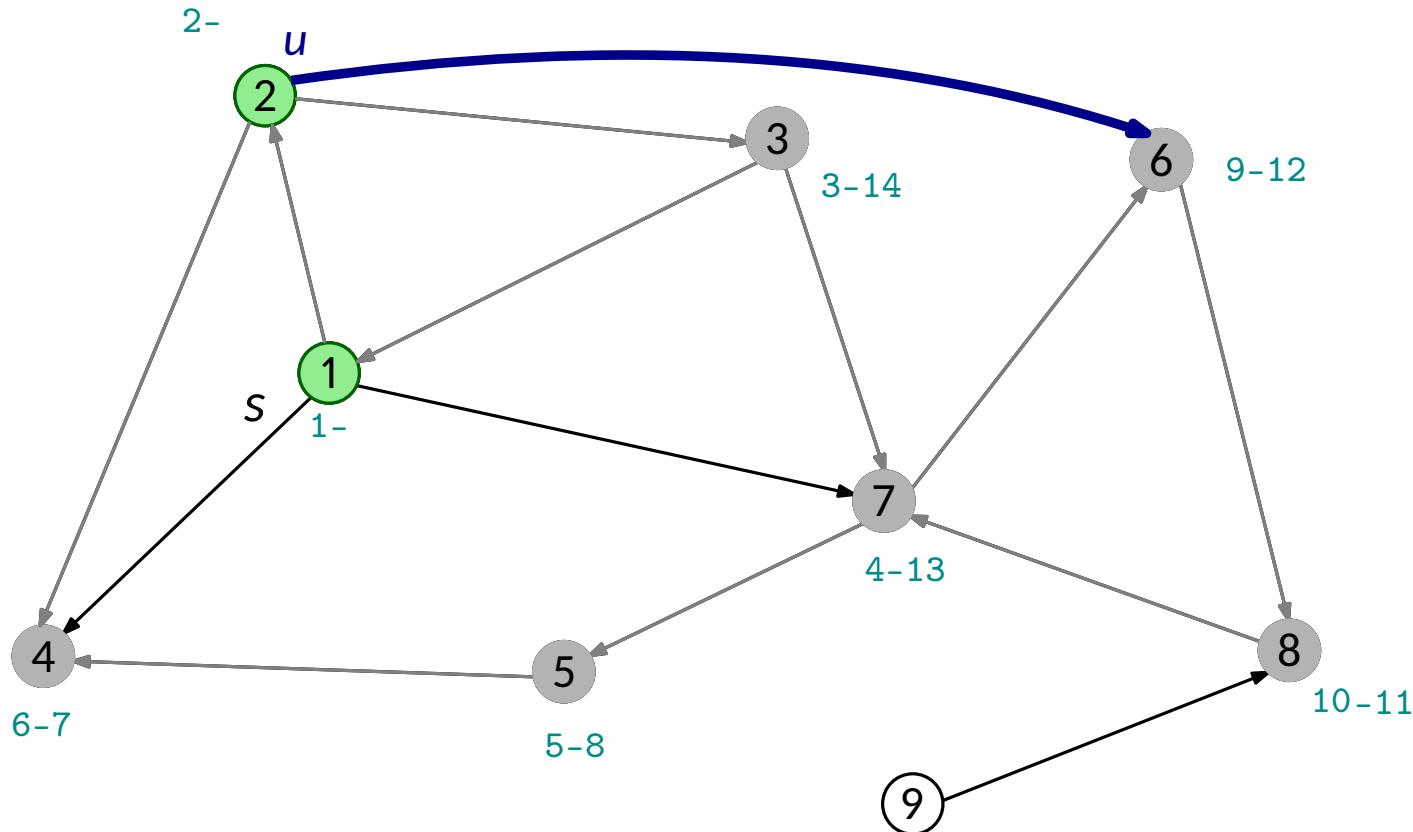


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von i bis j

Rekursionsstack:  
(implizit) 1 2

# Beispiel für DFS

Startknoten:  $s = 1$

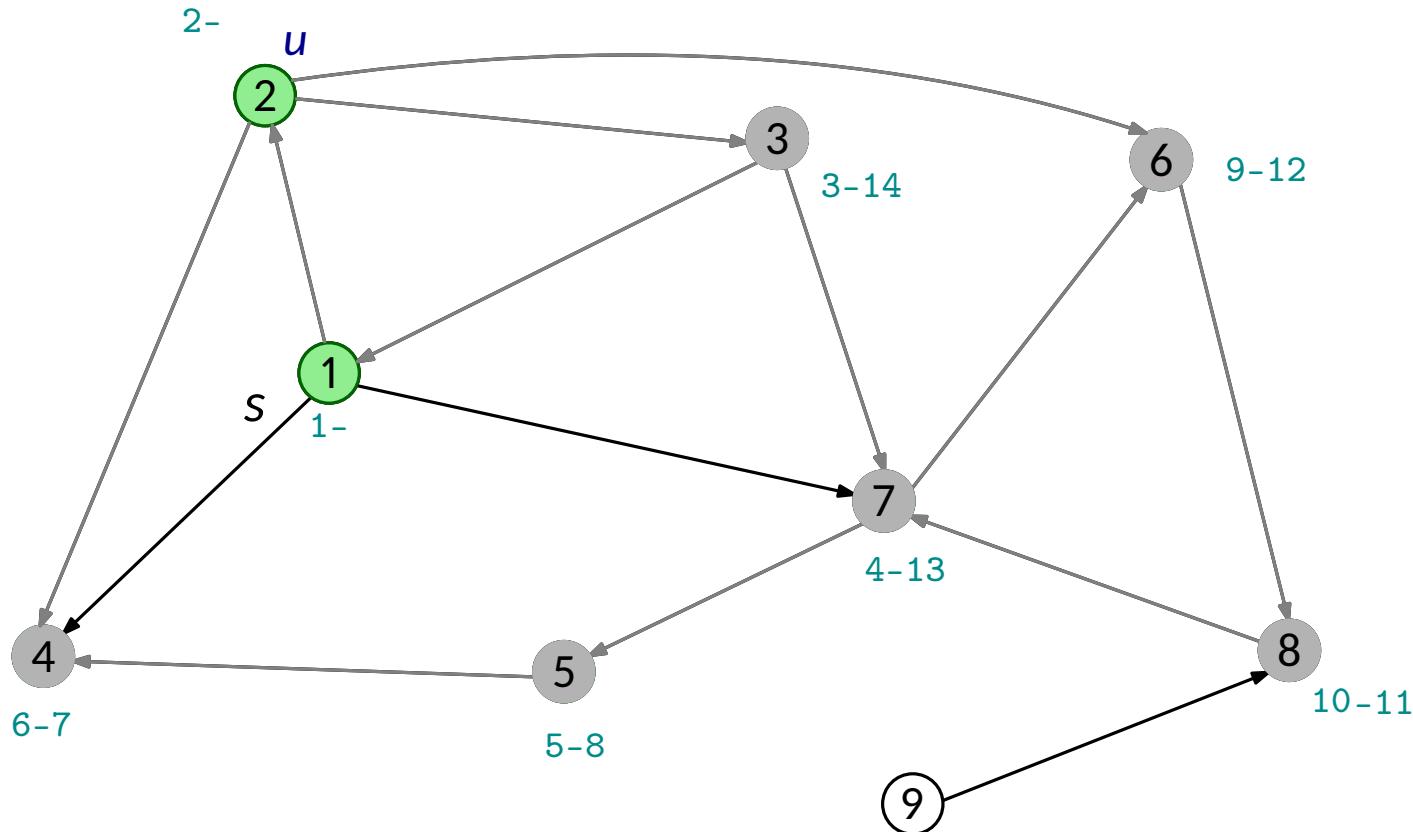


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2

# Beispiel für DFS

Startknoten:  $s = 1$



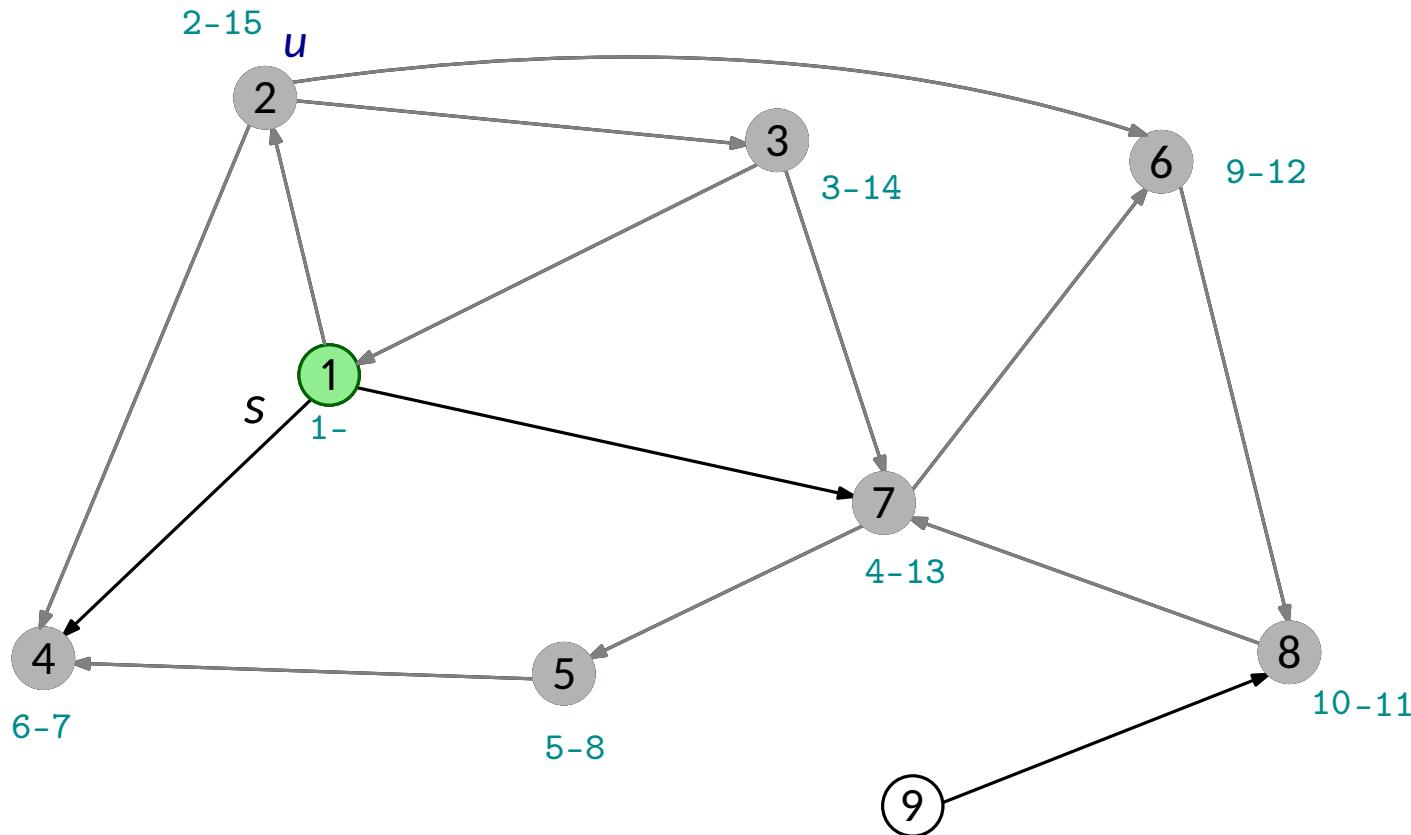
- unentdeckt
  - entdeckt
  - abgefertigt
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit)

1 2

# Beispiel für DFS

Startknoten:  $s = 1$

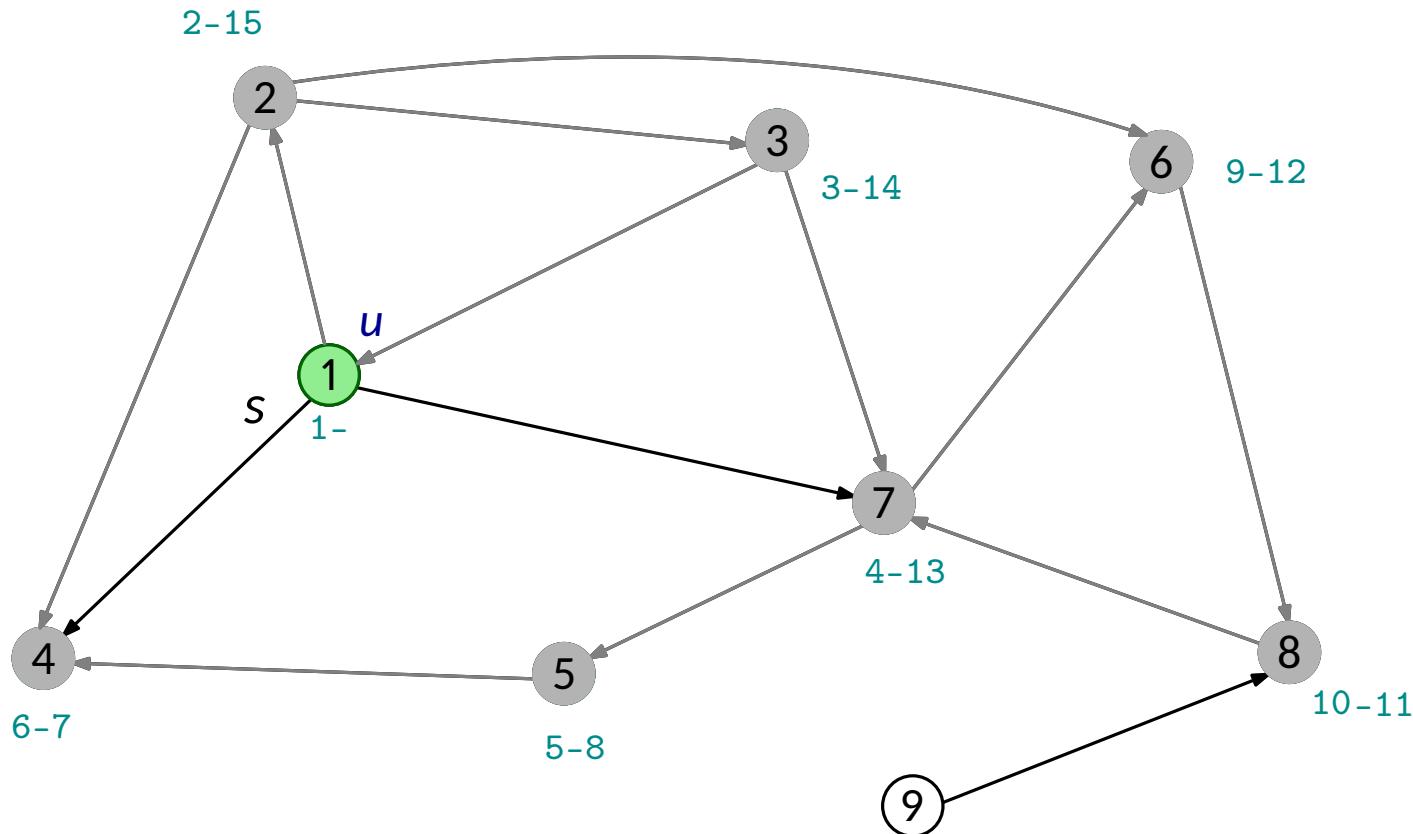


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1 2

# Beispiel für DFS

Startknoten:  $s = 1$

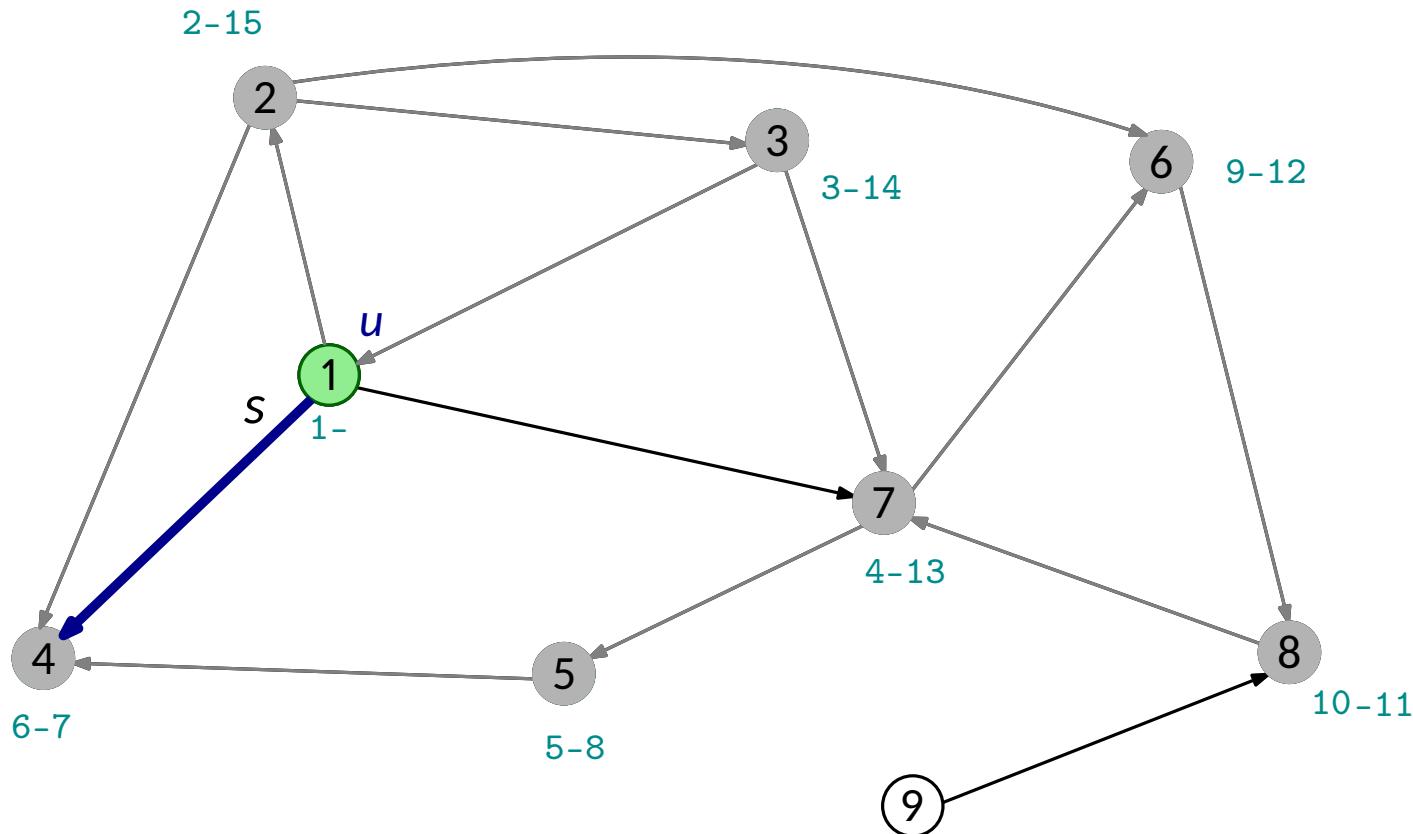


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1

# Beispiel für DFS

Startknoten:  $s = 1$

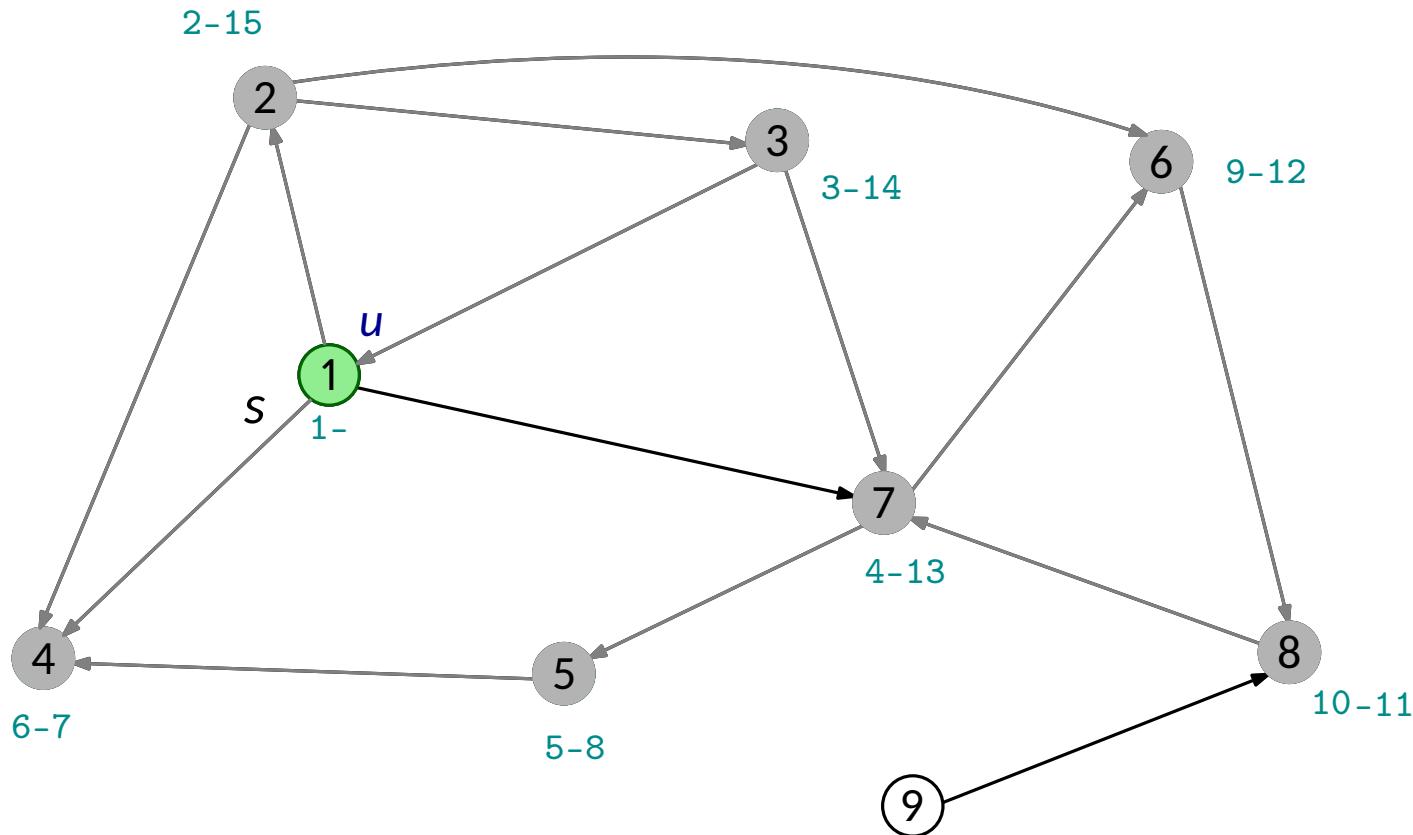


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1

# Beispiel für DFS

Startknoten:  $s = 1$

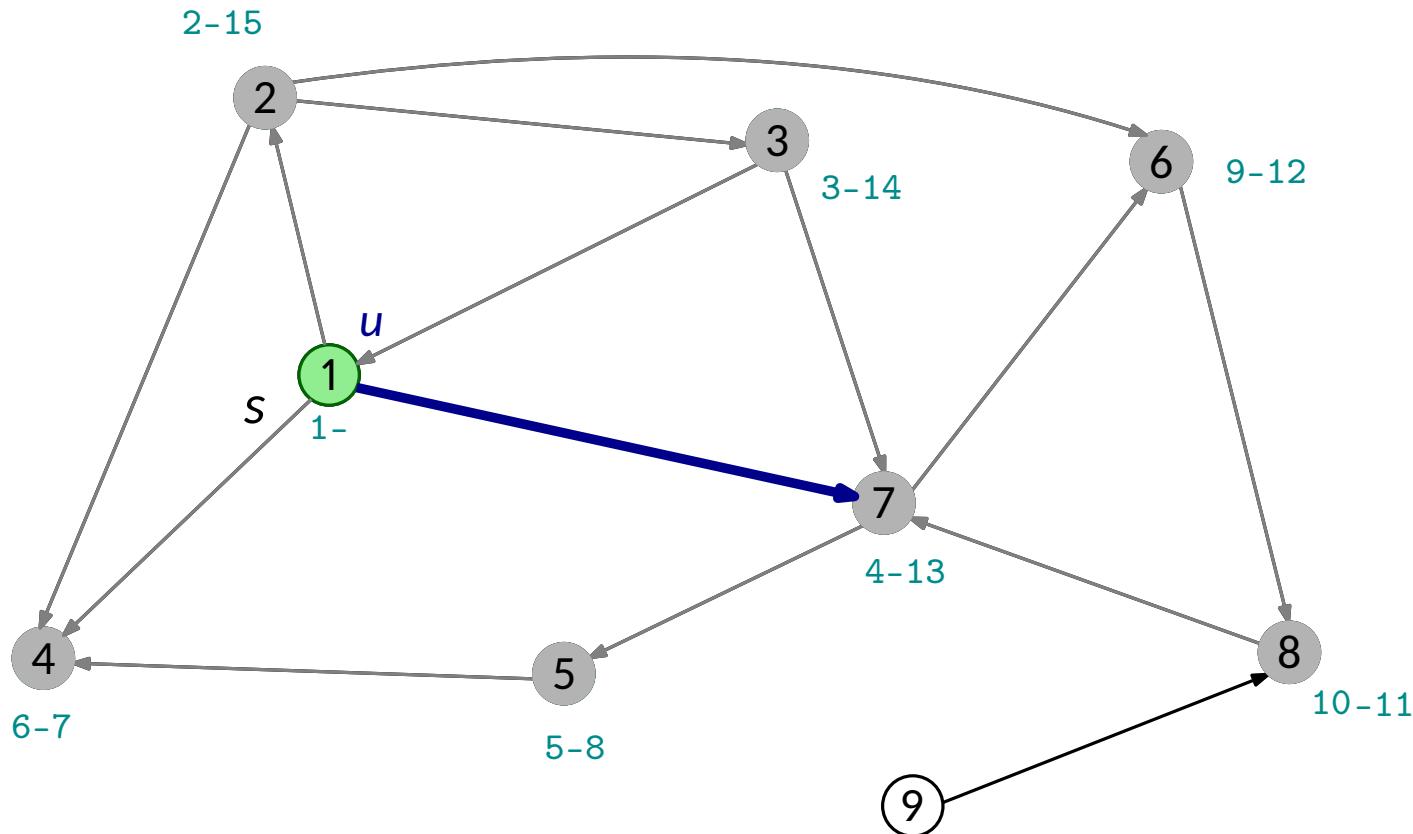


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1

# Beispiel für DFS

Startknoten:  $s = 1$



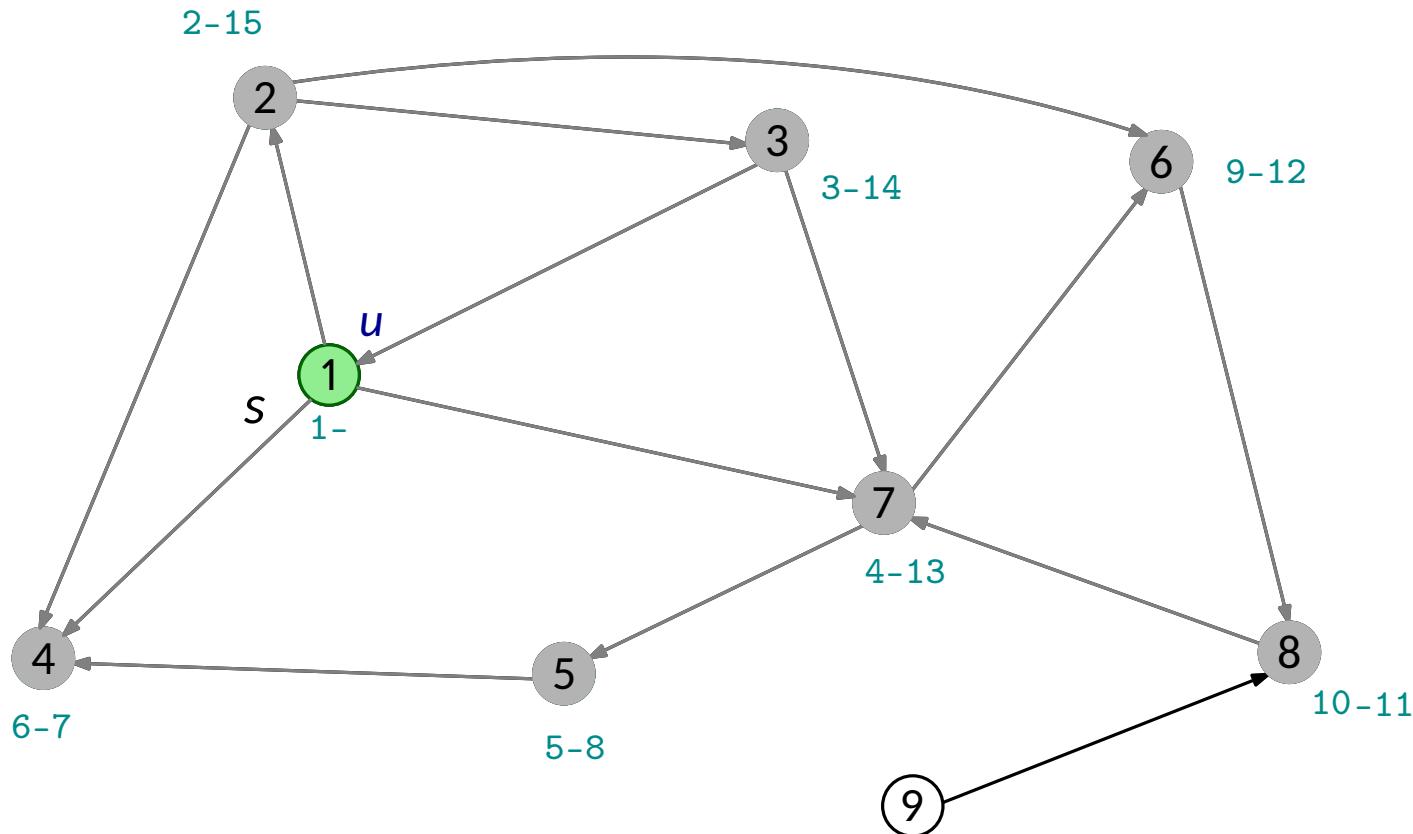
- unentdeckt
  - entdeckt
  - abgefertigt
- i-j Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

1

# Beispiel für DFS

Startknoten:  $s = 1$

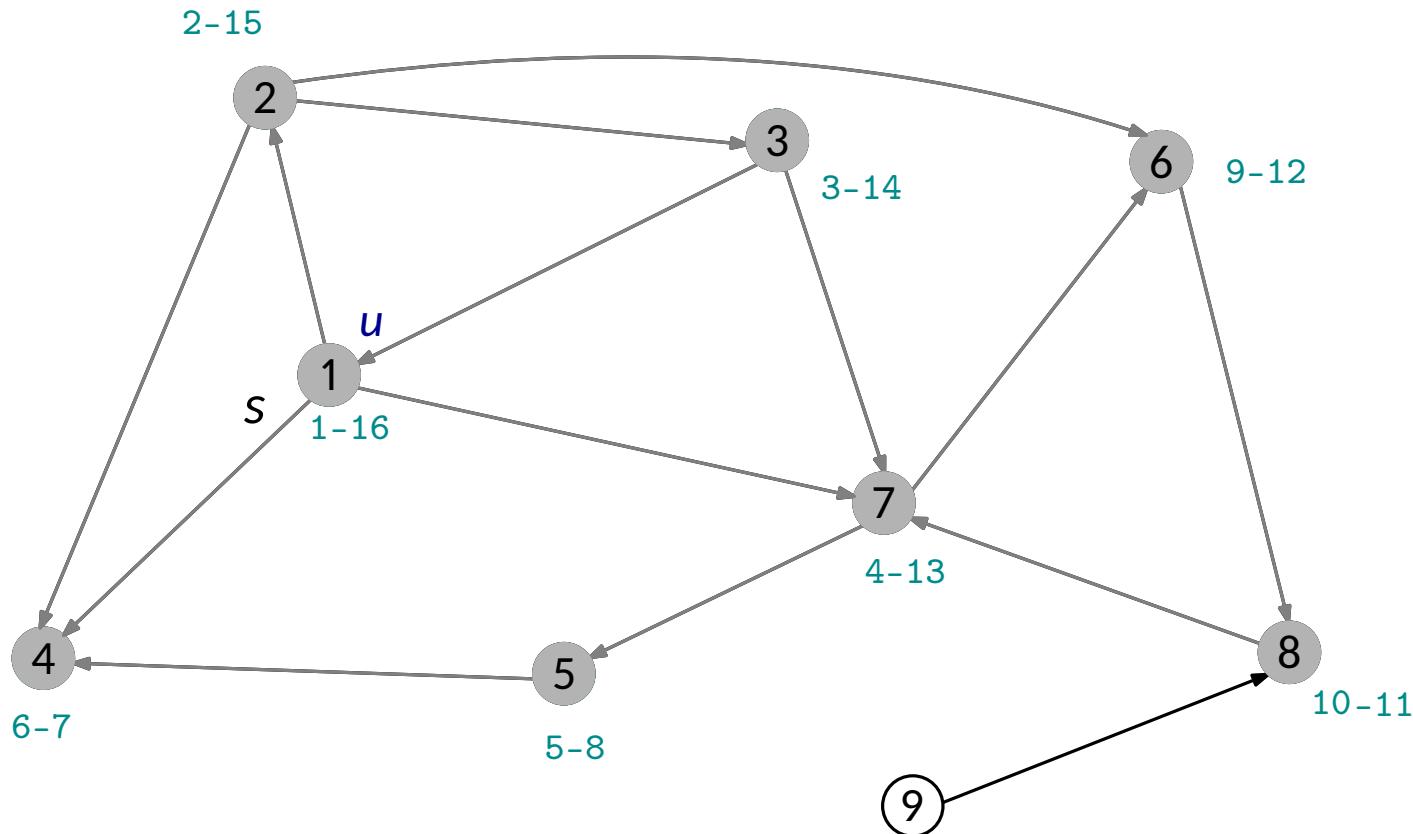


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (grey circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1

# Beispiel für DFS

Startknoten:  $s = 1$

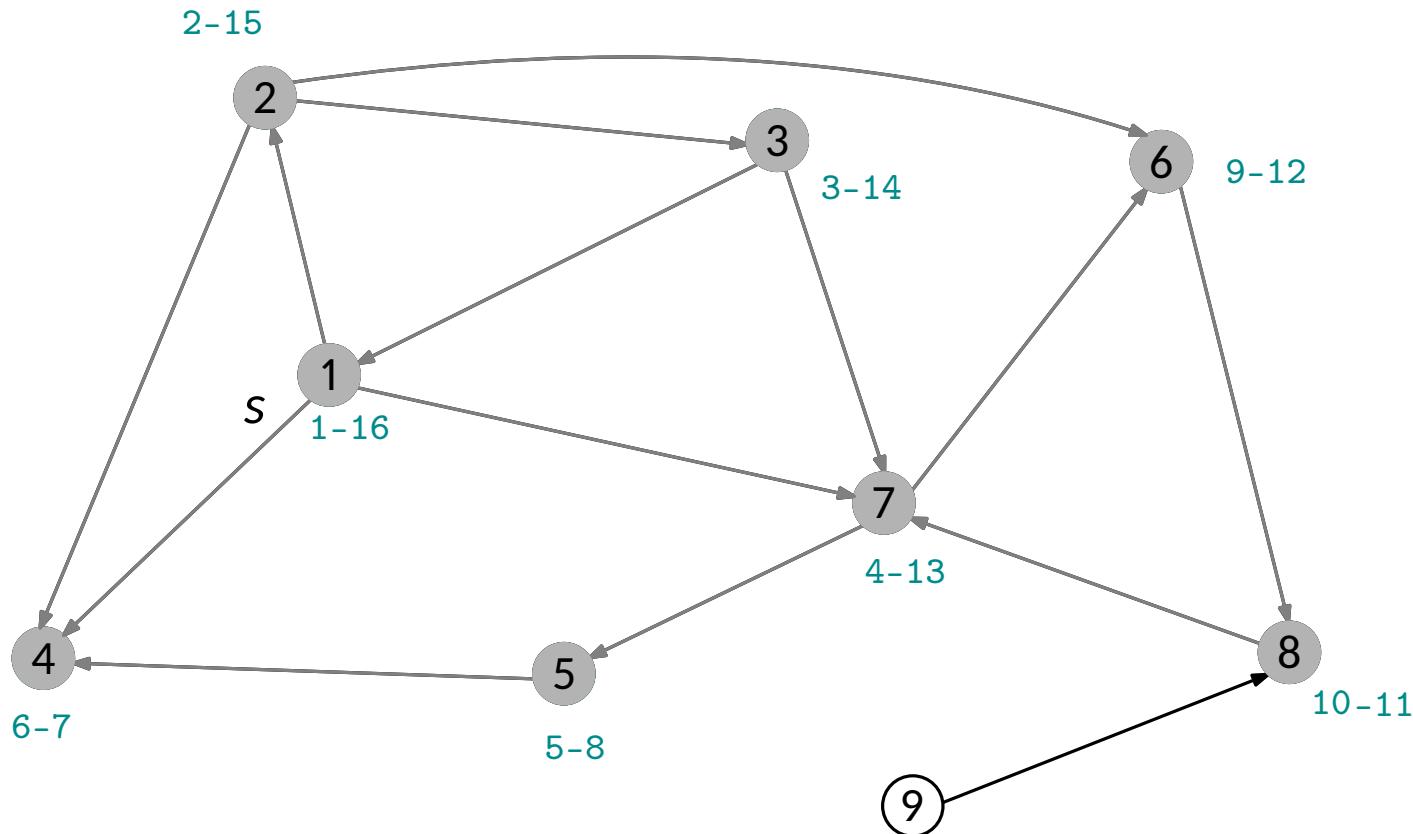


- unentdeckt (white circle)
  - entdeckt (green circle)
  - abgefertigt (gray circle)
- $i-j$  Lebenszeit von  $i$  bis  $j$

Rekursionsstack:  
(implizit) 1

# Beispiel für DFS

Startknoten:  $s = 1$



- unentdeckt
  - entdeckt
  - abgefertigt
- i-j Lebenszeit von i bis j

Rekursionsstack:  
(implizit)

# DFS: Details

DFS-explore( $G, s$ ):

$n = \text{Anzahl an Knoten von } G$

**for**  $u = 1, \dots, n$  **do**  
|  $\text{state}[u] = \text{unentdeckt}$   
|  $p[u] = \perp$

$t = 1$   
DFS( $G, s$ )

$p[u], u \in V$  beschreibt den **Suchbaum** ( $\nearrow$  wie bei der BFS)

$t$  ist ein globaler Zähler ("Zeit")  
 $b[u]$  - Entdeckungszeitpunkt von  $u$   
 $f[u]$  - Abfertigungsszeitpunkt von  $u$

$\rightarrow$  Lebenszeit von  $u$ :  $I_u = [b[u], f[u]]$

DFS( $G, u$ ):

$\text{state}[u] = \text{entdeckt}$

$b[u] = t; t = t + 1$

**for all**  $v$  with  $(u, v) \in E$  **do**  
| **if**  $\text{state}[v] = \text{unentdeckt}$  **then**  
| | DFS( $G, v$ )

$\text{state}[u] = \text{abgefertigt}$

$f[u] = t; t = t + 1$

# DFS: Korrektheit und Laufzeit

---

## Lemma

Eine Tiefensuche von  $s$  bestimmt alle von  $s$  erreichbaren Knoten in Zeit  $O(|V| + |E|) = O(n + m)$ .

Annahme:  $G$  ist in Adjazenzlistenrepräsentation gegeben.

Hinweis: Die Laufzeit ist sogar etwas besser, nämlich  $O(n + \sum_{u:s \leadsto u} \deg^+(u))$

Beweis vergleichbar zum Beweis für die Breitensuche.

# Anwendungen von Graphtraversierungen

# Zusammenhangskomponenten

---

## Lemma

Wir können die Zusammenhangskomponenten eines **ungerichteten Graphen**  $G$  in Zeit  $O(n + m)$  bestimmen.  
also auch die **schwachen** Zusammenhangskomponenten einer **gerichteten Graphen**

# Zusammenhangskomponenten

---

## Lemma

Wir können die Zusammenhangskomponenten eines **ungerichteten Graphen**  $G$  in Zeit  $O(n + m)$  bestimmen.  
also auch die **schwachen** Zusammenhangskomponenten einer **gerichteten Graphen**

- Wir starten mit einem beliebigen Startknoten (z.B.  $s = 1$ )
- Wir machen eine BFS oder DFS von  $s$   
→ liefert Zusammenhangskomponente von  $s$
- Solange es noch unentdeckte Knoten gibt:  
    wiederhole die Graphtraversierung mit einem unentdeckten Knoten  $s$   
→ liefert weitere Zusammenhangskomponente(n)

DFS-explore( $G, s$ ):

$n$  = Anzahl an Knoten von  $G$

**for**  $u = 1, \dots, n$  **do**  
     $state[u] = \text{unentdeckt}$   
     $p[u] = \perp$

$t = 0$   
DFS( $G, s$ )

# Zusammenhangskomponenten

## Lemma

Wir können die Zusammenhangskomponenten eines **ungerichteten Graphen**  $G$  in Zeit  $O(n + m)$  bestimmen.  
also auch die **schwachen** Zusammenhangskomponenten einer **gerichteten Graphen**

- Wir starten mit einem beliebigen Startknoten (z.B.  $s = 1$ )
- Wir machen eine BFS oder DFS von  $s$   
→ liefert Zusammenhangskomponente von  $s$
- Solange es noch unentdeckte Knoten gibt:  
    wiederhole die Graphtraversierung mit einem unentdeckten Knoten  $s$   
→ liefert weitere Zusammenhangskomponente(n)

bestimmt Zusammenhangskomponente von  $s$

```
DFS-explore-complete( $G$ ):  
 $n$  = Anzahl an Knoten von  $G$   
for  $u = 1, \dots, n$  do  
     $state[u] = \text{unentdeckt}$   
     $p[u] = \perp$   
  
 $t = 0$   
for  $s = 1, \dots, n$  do  
    if  $state[s] = \text{unentdeckt}$  then  
        DFS( $G, s$ )
```

# Zusammenhangskomponenten

## Lemma

Wir können die Zusammenhangskomponenten eines **ungerichteten Graphen**  $G$  in Zeit  $O(n + m)$  bestimmen.  
also auch die **schwachen** Zusammenhangskomponenten einer **gerichteten Graphen**

- Wir starten mit einem beliebigen Startknoten (z.B.  $s = 1$ )
- Wir machen eine BFS oder DFS von  $s$   
→ liefert Zusammenhangskomponente von  $s$
- Solange es noch unentdeckte Knoten gibt:  
    wiederhole die Graphtraversierung mit einem unentdeckten Knoten  $s$   
→ liefert weitere Zusammenhangskomponente(n)

Ein genauer Blick ergibt eine Laufzeit von  $O(n + m)$ .

bestimmt Zusammenhangskomponente von  $s$

```
DFS-explore-complete( $G$ ):  
 $n$  = Anzahl an Knoten von  $G$   
for  $u = 1, \dots, n$  do  
     $state[u] = \text{unentdeckt}$   
     $p[u] = \perp$   
  
 $t = 0$   
for  $s = 1, \dots, n$  do  
    if  $state[s] = \text{unentdeckt}$  then  
        | DFS( $G, s$ )
```

## Lemma

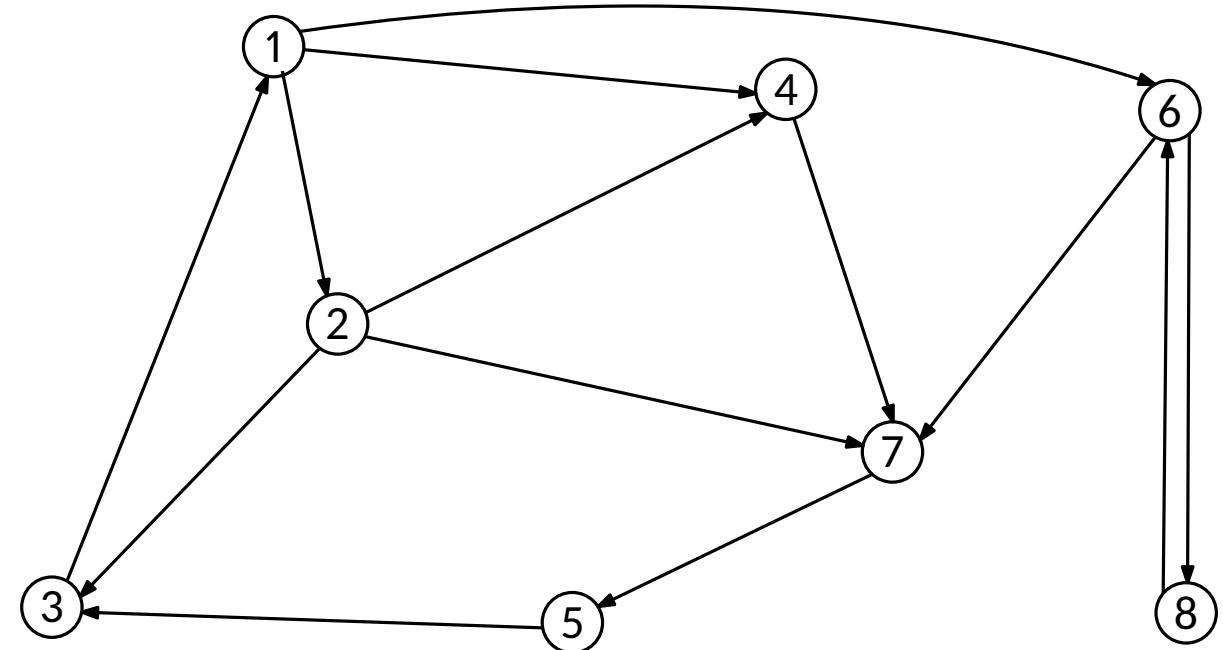
Wir können die **starken** Zusammenhangskomponenten eines **gerichteten Graphen**  $G$  in Zeit  $O(n + m)$  bestimmen.

verschiedene Algorithmen, i.d.R. nicht-triviale Anwendung einer (bzw. mehrerer) DFS  
(Kosarajus Algorithmus, Tarjans Algorithmus, Pfadbasierter Algorithmus)

→ In dieser Vorlesung nicht bewiesen

# Suchbäume: Kantenarten

---

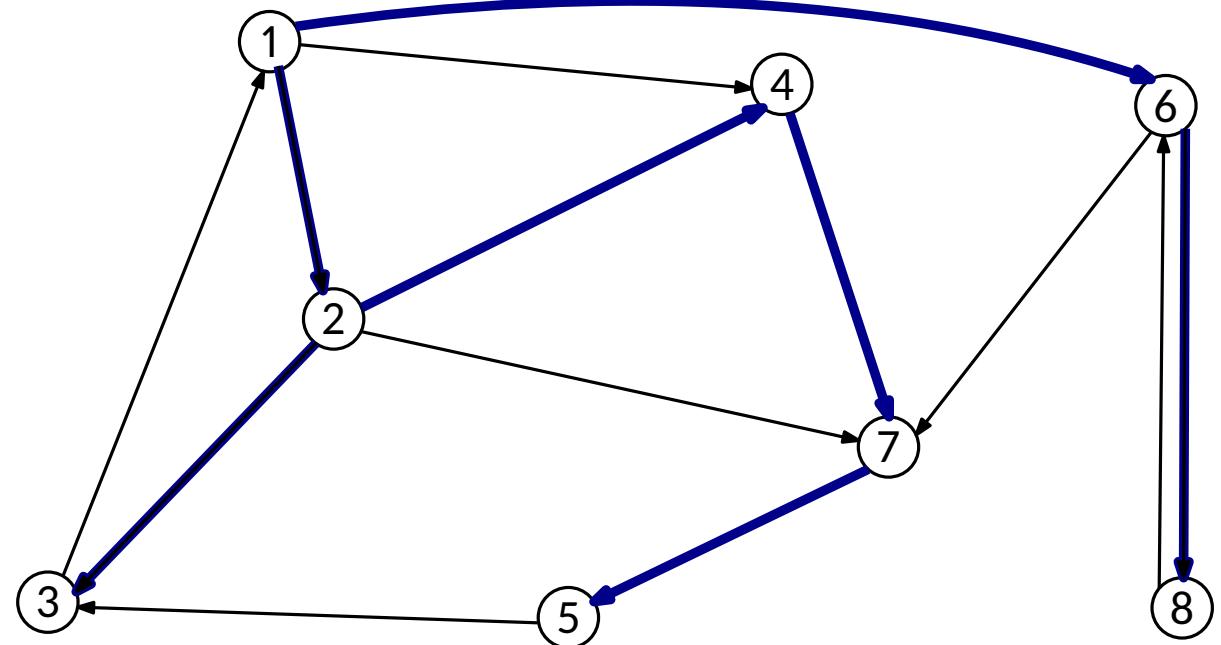


# Suchbäume: Kantenarten

---

Sowohl BFS als auch DFS von  $s$  liefert einen Suchbaum  $T$  mit Wurzel  $s$ .

→  $T = (V_T, E_T)$  ist ein gerichteter Baum, der **Teilgraph** von  $G = (V, E)$  ist



# Suchbäume: Kantenarten

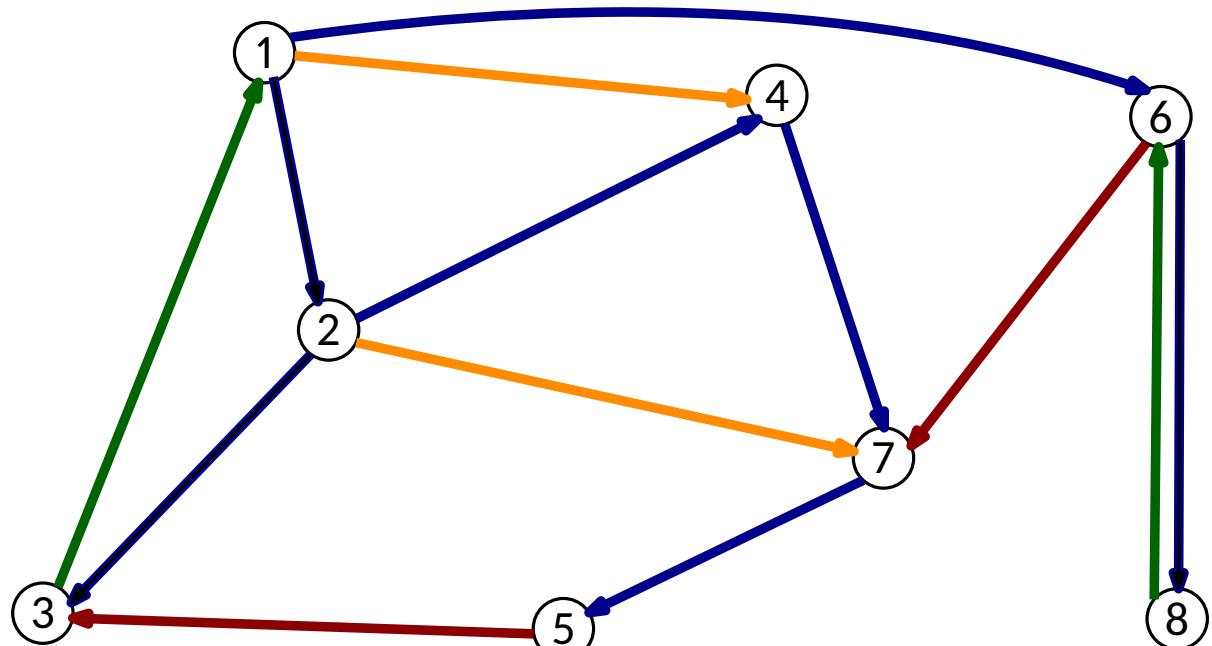
Sowohl BFS als auch DFS von  $s$  liefert einen Suchbaum  $T$  mit Wurzel  $s$ .

$\rightarrow T = (V_T, E_T)$  ist ein gerichteter Baum, der **Teilgraph** von  $G = (V, E)$  ist

Jede Kante  $(u, v) \in E$  hat einen der folgenden Typen:

- **Baumkante:**
  - $(u, v) \in E_T$
- **Vorwärtskante:**
  - $(u, v) \notin E_T$ , und
  - $v$  ist Nachfahre von  $u$  in  $T$
- **Rückwärtskante:**
  - $(u, v) \notin E_T$ , und
  - $v$  ist Vorfahre von  $u$  in  $T$

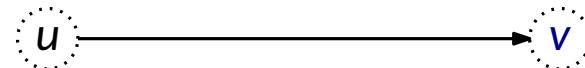
- **Querkante:**
  - alle anderen Kanten
  - (verbindet zwei verschiedene Teilbäume)



# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?



· **Baumkante:**

- $(u, v) \in E_T$

· **Vorwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Nachfahre von  $u$  in  $T$

· **Rückwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Vorfahre von  $u$  in  $T$

· **Querkante:**

- alle anderen Kanten

# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?

Klar ist:

- $u$ 's Status ist **entdeckt**
- $u$  hat eine Entdeckungszeit  $b(u)$

$b(u) -$



· **Baumkante:**

- $(u, v) \in E_T$

· **Vorwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Nachfahre von  $u$  in  $T$

· **Rückwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Vorfahre von  $u$  in  $T$

· **Querkante:**

- alle anderen Kanten

# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?

Klar ist:

- $u$ 's Status ist **entdeckt**
- $u$  hat eine Entdeckungszeit  $b(u)$

$b(u) -$



**Fall 1:**  $v$  ist **unentdeckt**  
 $(u, v)$  ist **Baumkante**

• **Baumkante:**

- $(u, v) \in E_T$

• **Vorwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Nachfahre von  $u$  in  $T$

• **Rückwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Vorfahre von  $u$  in  $T$

• **Querkante:**

- alle anderen Kanten

# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?

Klar ist:

- $u$ 's Status ist **entdeckt**
- $u$  hat eine Entdeckungszeit  $b(u)$



**Fall 1:**  $v$  ist **unentdeckt**  
 $(u, v)$  ist **Baumkante**

**Fall 2:**  $v$  ist **entdeckt**  
 $(u, v)$  ist **Rückwärtskante**

$v$  ist noch nicht **abgefertigt**.  $u$  muss also Nachfahre von  $v$  sein  
 $\Rightarrow (u, v)$  schließt einen **Zyklus!**

• **Baumkante:**

- $(u, v) \in E_T$

• **Vorwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist **Nachfahre** von  $u$  in  $T$

• **Rückwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist **Vorfahre** von  $u$  in  $T$

• **Querkante:**

- alle anderen Kanten

# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?

Klar ist:

- $u$ 's Status ist **entdeckt**
- $u$  hat eine Entdeckungszeit  $b(u)$



**Fall 1:**  $v$  ist **unentdeckt**  
 $(u, v)$  ist **Baumkante**

**Fall 2:**  $v$  ist **entdeckt**  
 $(u, v)$  ist **Rückwärtskante**

$v$  ist noch nicht **abgefertigt**.  $u$  muss also Nachfahre von  $v$  sein  
 $\Rightarrow (u, v)$  schließt einen **Zyklus!**

**Fall 3:**  $v$  ist **abgefertigt**  
Frage: Welchen Kantentyp hat  $e$ ?

• **Baumkante:**

- $(u, v) \in E_T$

• **Vorwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist **Nachfahre** von  $u$  in  $T$

• **Rückwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist **Vorfahre** von  $u$  in  $T$

• **Querkante:**

- alle anderen Kanten

# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?

Klar ist:

- $u$ 's Status ist **entdeckt**
- $u$  hat eine Entdeckungszeit  $b(u)$



**Fall 1:**  $v$  ist **unentdeckt**  
 $(u, v)$  ist **Baumkante**

**Fall 2:**  $v$  ist **entdeckt**  
 $(u, v)$  ist **Rückwärtskante**  
 $v$  ist noch nicht **abgefertigt**.  $u$  muss also Nachfahre von  $v$  sein  
 $\Rightarrow (u, v)$  schließt einen **Zyklus!**

- **Baumkante:**
  - $(u, v) \in E_T$
- **Vorwärtskante:**
  - $(u, v) \notin E_T$ , und
  - $v$  ist **Nachfahre** von  $u$  in  $T$
- **Rückwärtskante:**
  - $(u, v) \notin E_T$ , und
  - $v$  ist **Vorfahre** von  $u$  in  $T$
- **Querkante:**
  - alle anderen Kanten

**Fall 3:**  $v$  ist **abgefertigt**

**Fall 3a:**  $v$  ist **abgefertigt** und  $b(v) > b(u)$   
 $(u, v)$  ist **Vorwärtskante**  
 $v$  wurde erst nach  $u$  entdeckt  $\rightarrow v$  ist Nachfahre von  $u$

**Fall 3b:**  $v$  ist **abgefertigt** und  $b(v) < b(u)$

# Kantenarten während einer DFS

**Situation:** Während einer DFS betrachten wir eine Kante  $e = (u, v)$

Wie können wir den Typ von  $e$  bestimmen?

Klar ist:

- $u$ 's Status ist **entdeckt**
- $u$  hat eine Entdeckungszeit  $b(u)$



**Fall 1:**  $v$  ist **unentdeckt**  
 $(u, v)$  ist **Baumkante**

**Fall 2:**  $v$  ist **entdeckt**  
 $(u, v)$  ist **Rückwärtskante**  
 $v$  ist noch nicht **abgefertigt**.  $u$  muss also Nachfahre von  $v$  sein  
 $\Rightarrow (u, v)$  schließt einen **Zyklus!**

**Fall 3:**  $v$  ist **abgefertigt**

**Fall 3a:**  $v$  ist **abgefertigt** und  $b(v) > b(u)$   
 $(u, v)$  ist **Vorwärtskante**  
 $v$  wurde erst nach  $u$  entdeckt  $\rightarrow v$  ist Nachfahre von  $u$

**Fall 3b:**  $v$  ist **abgefertigt** und  $b(v) < b(u)$   
 $(u, v)$  ist **Querkante**

$v$  abgefertigt bevor  $u$  entdeckt wurde  $\rightarrow$  kein Vor- oder Nachfahre

• **Baumkante:**

- $(u, v) \in E_T$

• **Vorwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Nachfahre von  $u$  in  $T$

• **Rückwärtskante:**

- $(u, v) \notin E_T$ , und
- $v$  ist Vorfahre von  $u$  in  $T$

• **Querkante:**

- alle anderen Kanten

# Zyklus finden

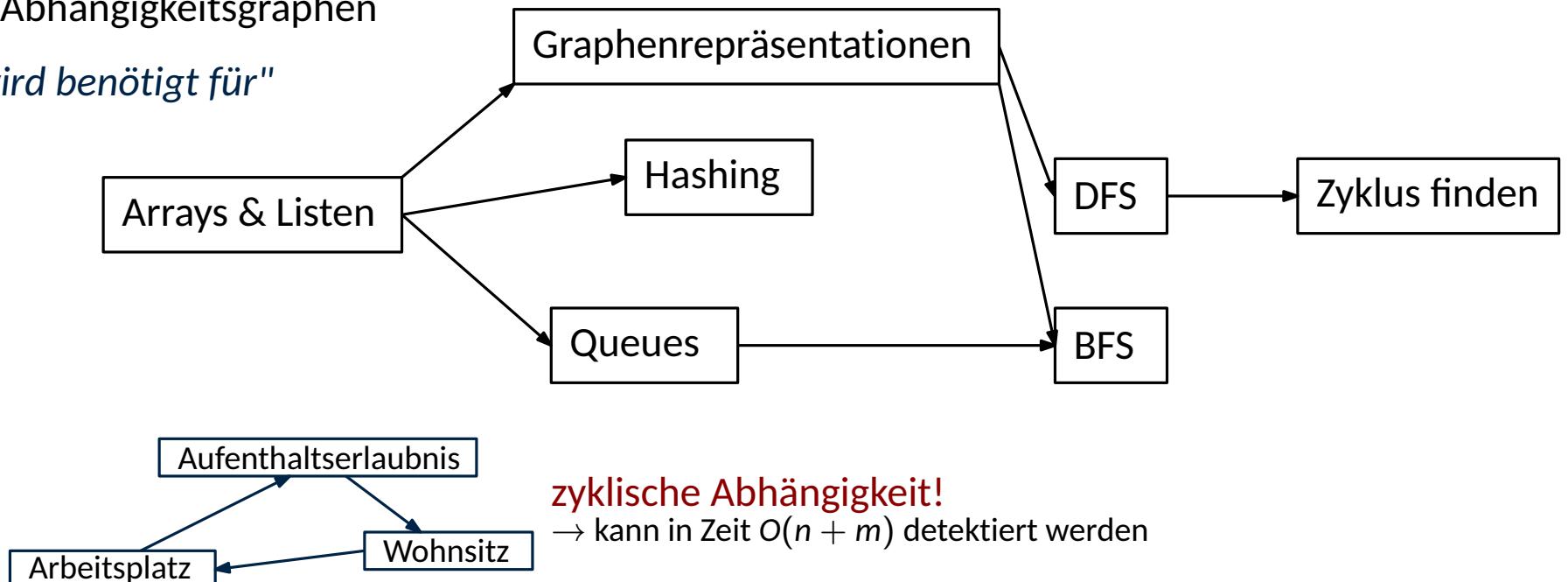
Bemerkung:  $G$  ist azyklisch  $\Leftrightarrow$  Tiefensuche auf  $G$  erzeugt keine Rückwärtskanten  
genauer:  $\text{DFS-explore-complete}(G)$

Wir können die Tiefensuche zu einem Algorithmus anpassen, der:

- in Zeit  $O(n + m)$  läuft
- wenn  $G$  einen Zyklus enthält, einen solchen Zyklus zurückgibt und
- ansonsten zurückgibt, dass  $G$  azyklisch ist

Beispiel: Abhängigkeitsgraphen

→ "wird benötigt für"

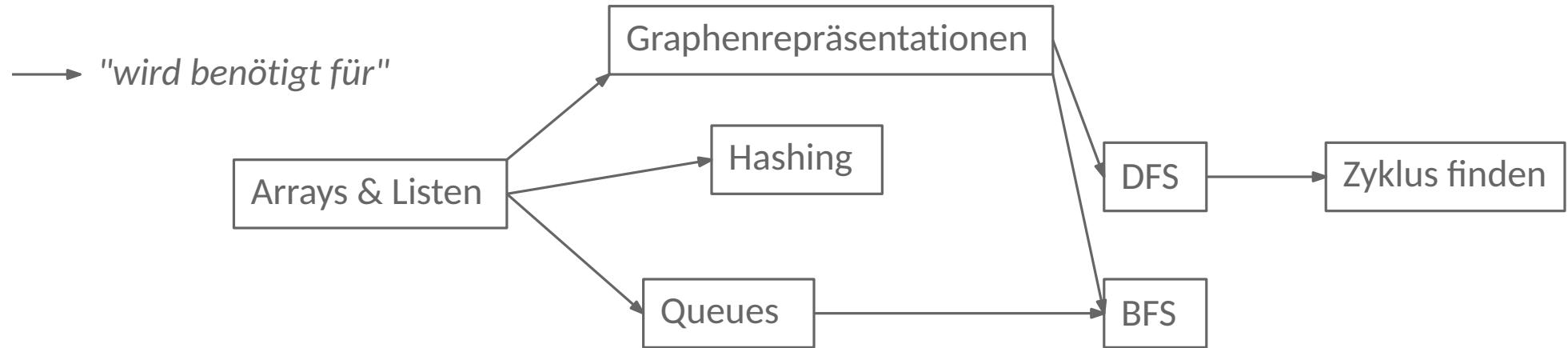


zyklische Abhängigkeit!  
→ kann in Zeit  $O(n + m)$  detektiert werden

# Topologische Sortierung

Gerichtete azyklische Graphen (**DAG** für **directed acyclic graph**) haben viele Anwendungen

Nochmal: Abhängigkeitsgraphen

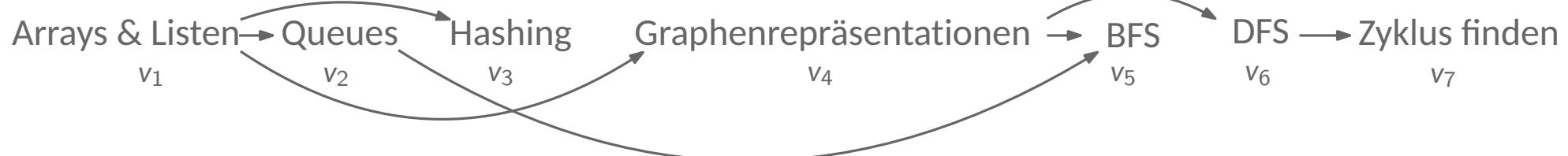


## Definition

Eine **topologische Sortierung** von  $G = (V, E)$  ist eine Permutation  $v_1, \dots, v_n$  von  $V$ , sodass:

wenn  $(v_i, v_j) \in E$ , dann gilt  $i < j$       Kanten gehen nur von "kleineren" zu "größeren" Knoten

z.B:



## Lemma

In einem **gerichteten, azyklischen Graphen**  $G$  existiert immer eine topologische Sortierung. Eine solche kann in Zeit  $O(n + m)$  berechnet werden.

# Zusammenfassung

---

- Graphen sind wichtige mathematische Hilfsmittel, um Netzwerke u.Ä. zu beschreiben
- verschiedene Möglichkeiten der Repräsentation, insbesondere:
  - Adjazenzlisten
  - Adjazenzmatrix
- Graphtraversierungen helfen, den Graphen zu erkunden:
  - BFS
  - DFS
  - **Anwendungen:**
    - Zusammenhang(skomponenten) bestimmen
    - Zyklen finden bzw. entscheiden, ob  $G$  azyklisch ist
    - topologische Sortierung

Laufzeit  $O(n + m)$

## Algorithmen und Datenstrukturen SS'23

# Kapitel 13: Kürzeste Wege

Marvin Künnemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letztes Kapitel: Grundbegriffe von Graphen, BFS, DFS

Jetzt: kürzeste Wege - Routing, etc.

Insbesondere besprechen wir in diesem Kapitel:

- ungewichtete Graphen: BFS
- nichtnegativ gewichtete Graphen: Dijkstra
- allgemein gewichtete Graphen: Bellman-Ford

# **Kürzeste Wege in ungewichteten Graphen**

# Kürzeste Wege - Definition

$$u = 1$$

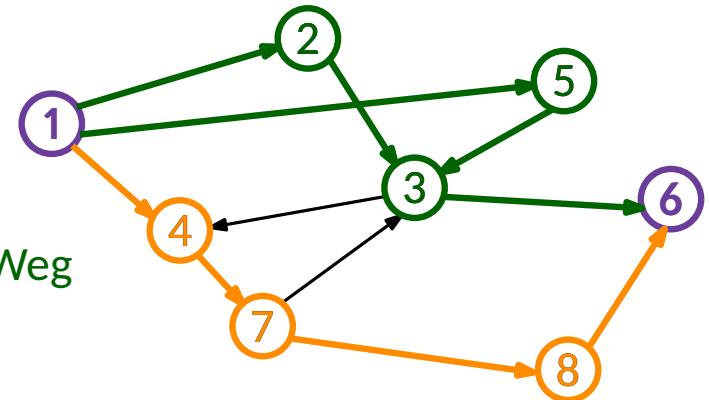
$$v = 6$$

1, 4, 7, 8, 6 - kein kürzester Weg

1, 2, 3, 6 - ein kürzester Weg

1, 5, 3, 6 - auch ein kürzester Weg

$$\rightarrow \delta(1, 6) = 3$$



## Definition

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

Wir bezeichnen einen Weg  $p_0, \dots, p_k$  von  $p_0 = u$  nach  $p_k = v$  als einen **kürzesten Weg**, wenn seine Länge  $k$  minimal ist.

Wir definieren die (**Kürzeste-Wege-Distanz**)  $\delta(u, v)$  durch

$$\delta(u, v) = \min\{k \mid \text{es existiert ein Weg } p_0, \dots, p_k \text{ von } u = p_0 \text{ nach } v = p_k\}$$

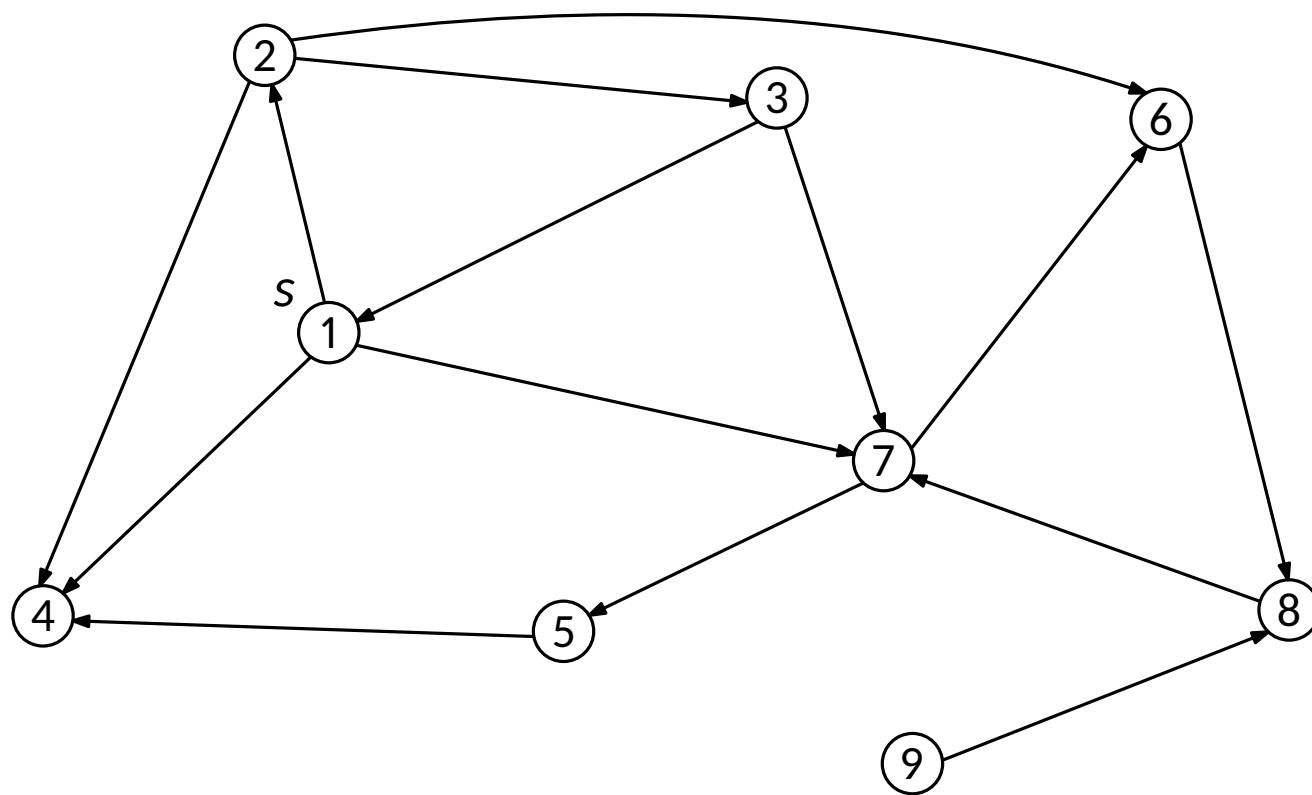
- Hinweise:**
- im Allgemeinen kann es mehr als einen kürzesten Weg geben
  - $\delta(u, v) = \infty$  genau dann, wenn  $v$  nicht von  $u$  erreichbar ist  
Beispiel:  $\delta(6, 1) = \infty$

## Behauptung:

Wir haben bereits einen Algorithmus kennengelernt, der den Wert  $\delta(u, v)$  berechnen kann.

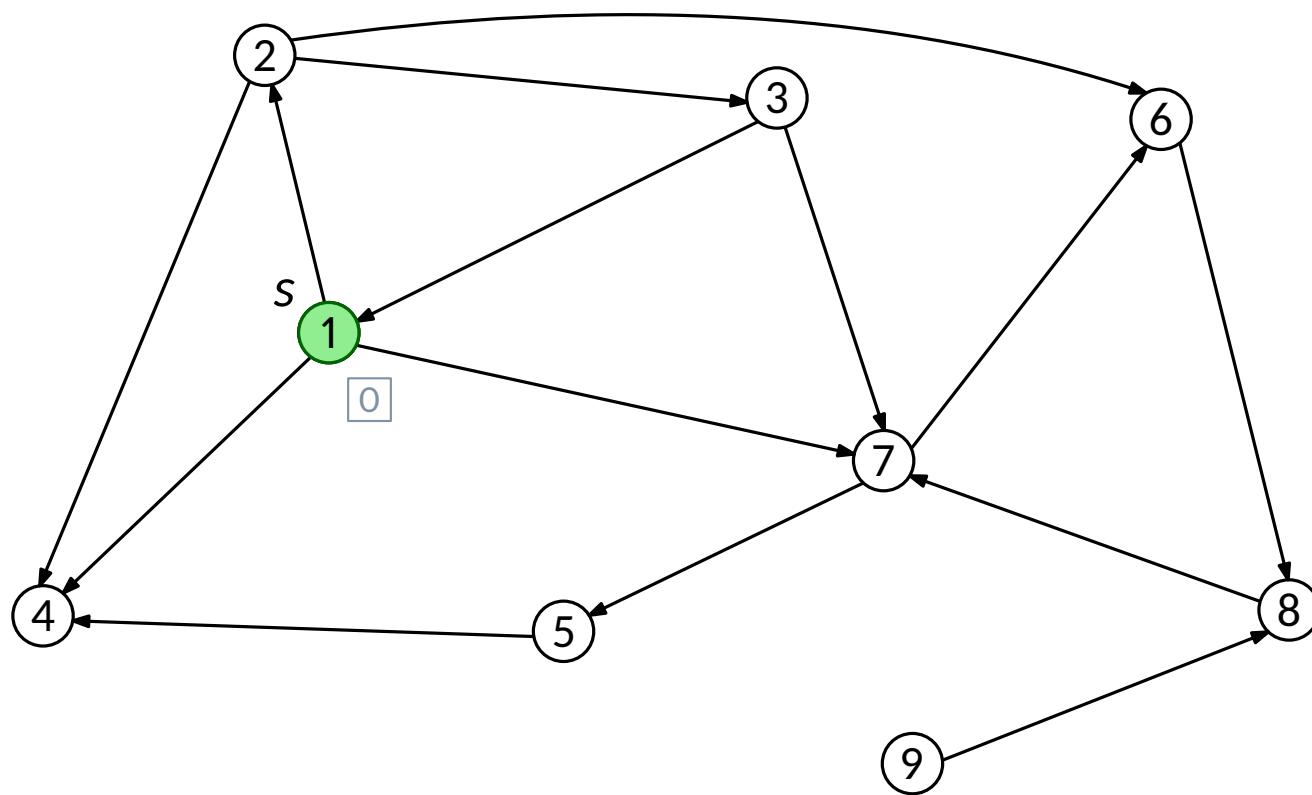
# Erinnerung: BFS

---



# Erinnerung: BFS

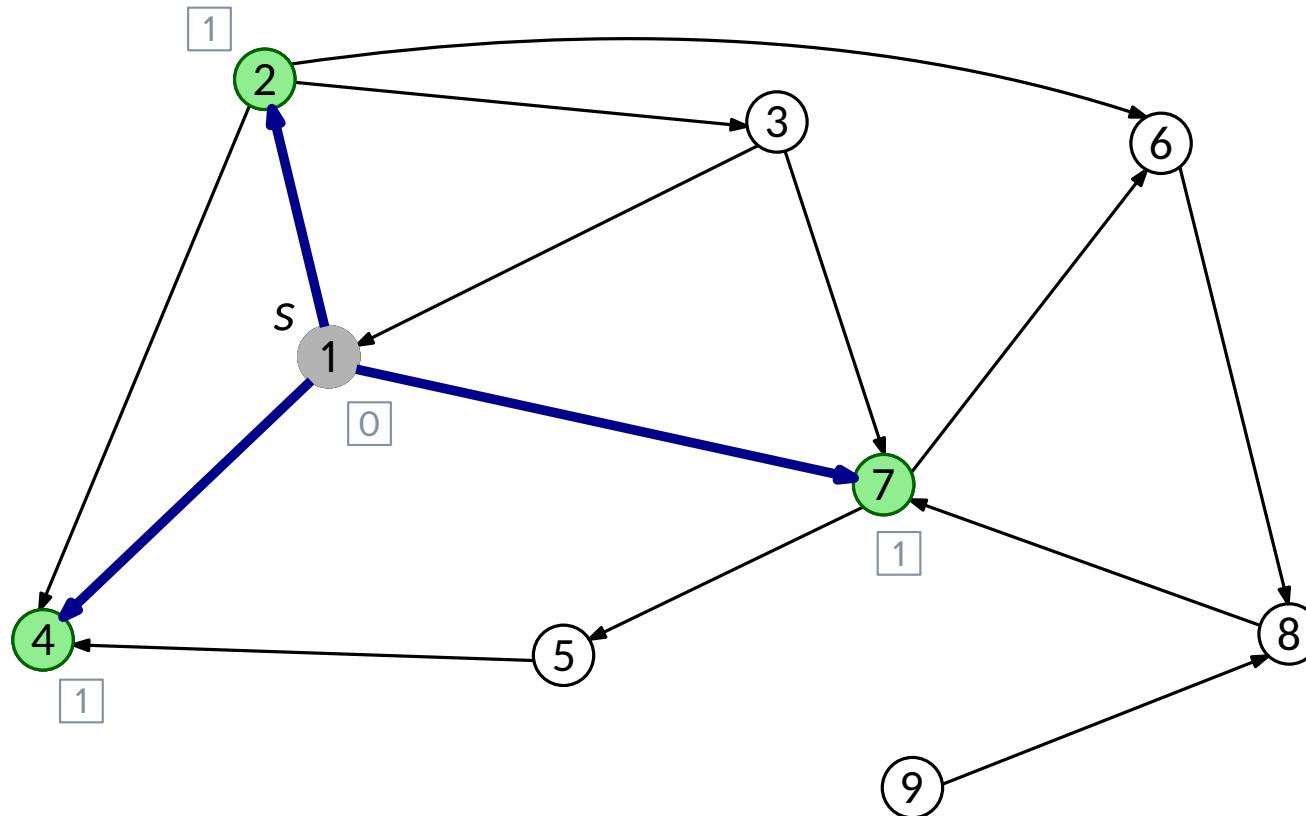
---



1

# Erinnerung: BFS

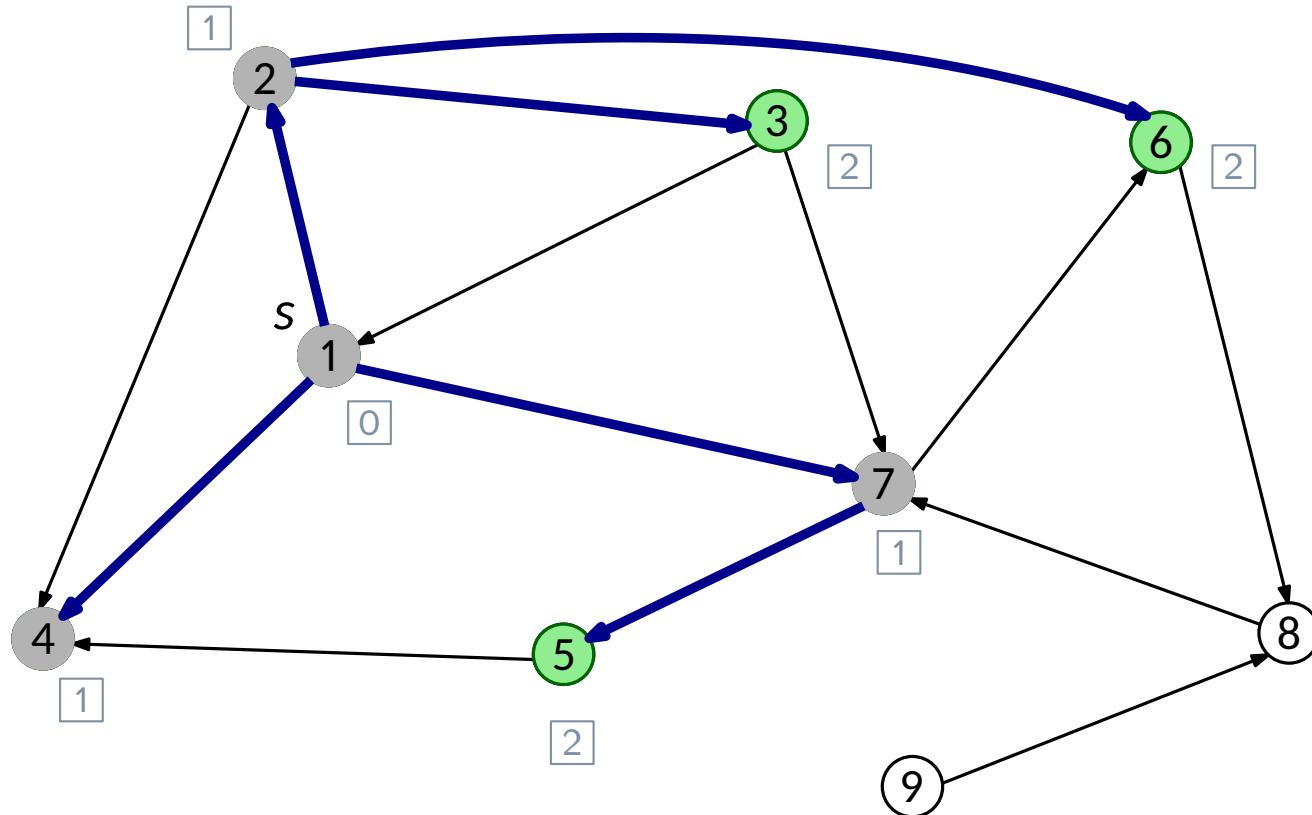
---



1 2 4 7

# Erinnerung: BFS

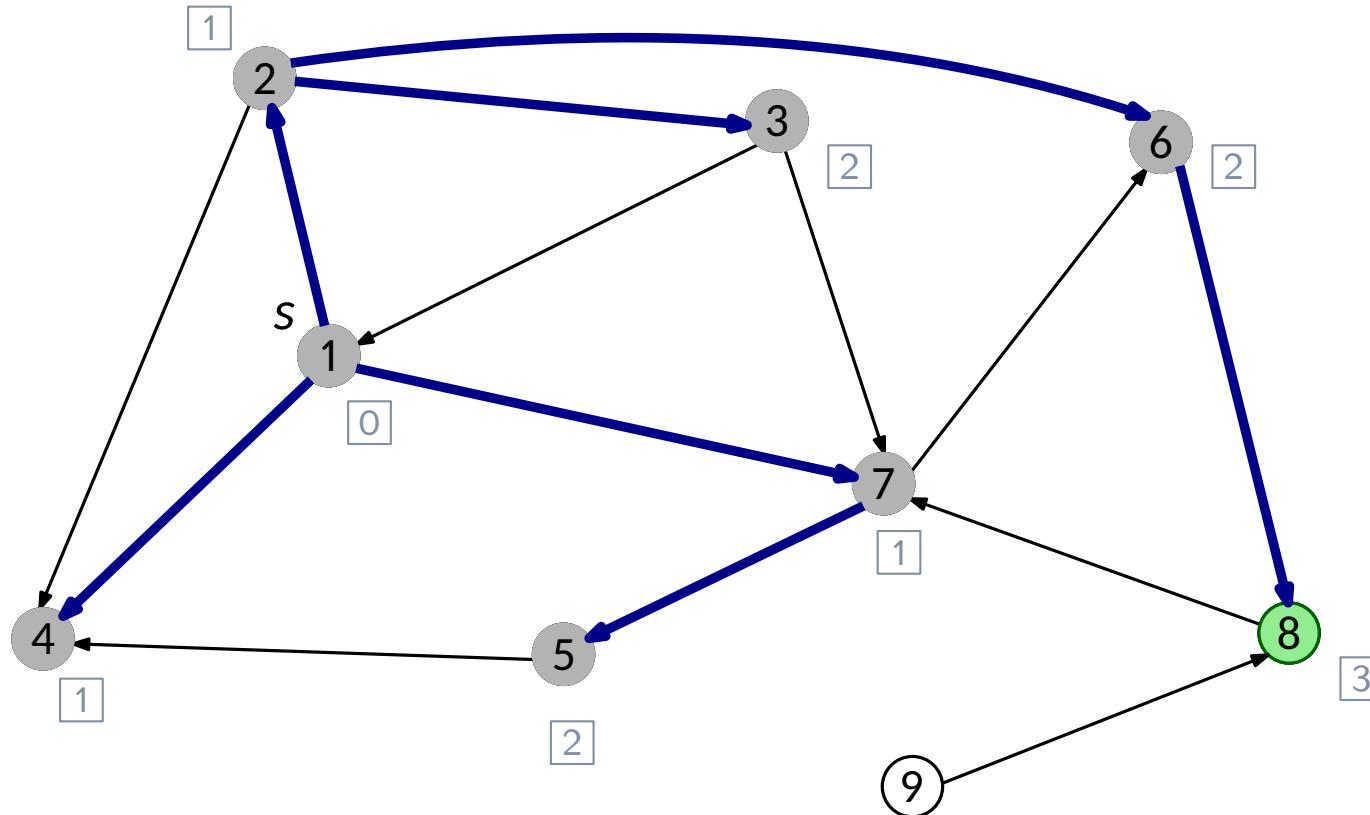
---



1 2 4 7 3 6 5

# Erinnerung: BFS

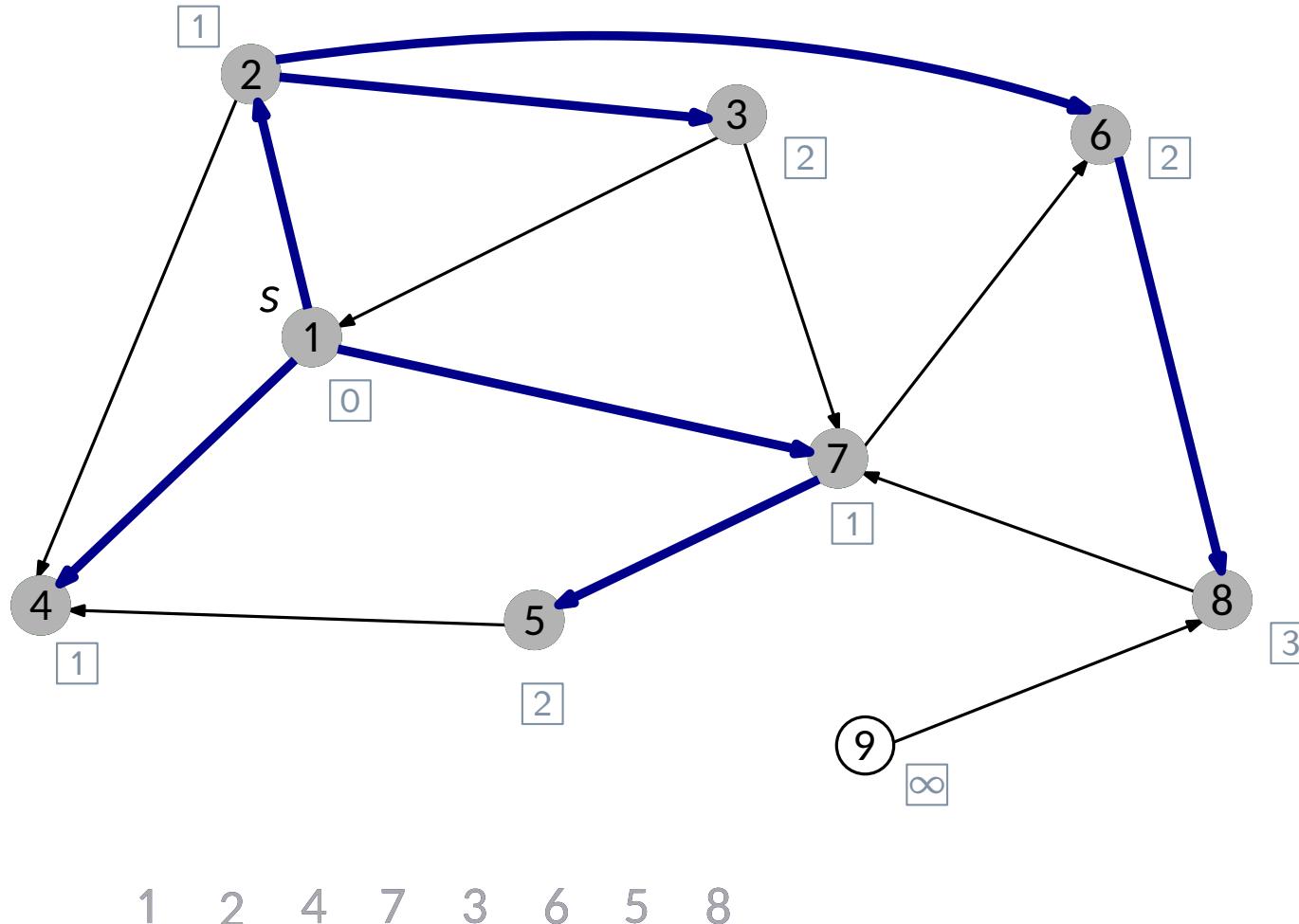
---



1 2 4 7 3 6 5 8

# Erinnerung: BFS

---



1 2 4 7 3 6 5 8

# BFS: kürzeste Wege in ungewichteten Graphen

---

## Theorem.

Sei  $G = (V, E)$  ein Graph und  $s \in V$ .

Eine BFS von  $s$  berechnet alle Distanzen  $\delta(s, v)$  mit  $v \in V$  in Zeit  $O(n + m)$ .

Wir lassen den Beweis weg, denn wir werden später ein allgemeineres Resultat beweisen.

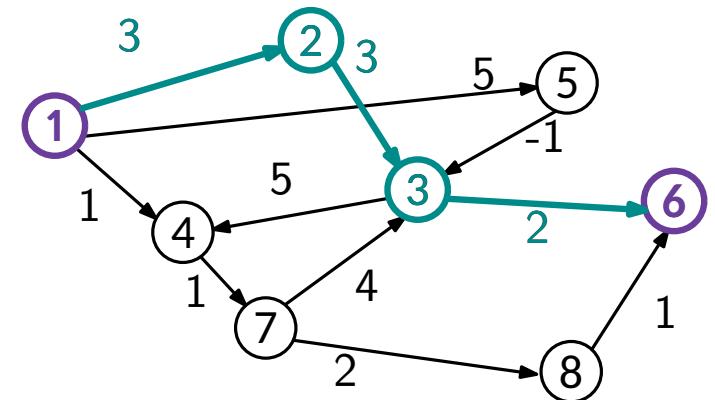
# **Kürzeste Wege in gewichteten Graphen**

# Allgemeinere Definition (gewichteten Graphen)

"Kosten"  $w(e)$ :  
Länge/Zeitbedarf/Preis  
Für  $e = (u, v)$  schreiben wir  
 $w(u, v) = w(e)$

$$\begin{array}{l} u = 1 \\ v = 6 \\ 1, 2, 3, 6 \end{array}$$

Kosten: 8



## Definition

Sei  $G = (V, E, w)$  ein gewichteter Graph und  $u, v \in V$ .  $w(e) \in \mathbb{Z}$  beschreibt **Kosten** (Länge) der Kante  $e$ . Wir bezeichnen einen Weg  $\vec{p} = (p_0, \dots, p_k)$  von  $p_0 = u$  nach  $p_k = v$  als einen **kürzesten Weg**, wenn seine **Kosten**  $w(\vec{p}) = \sum_{i=0}^{k-1} w(p_i, p_{i+1})$  minimal sind.

# Allgemeinere Definition (gewichteten Graphen)

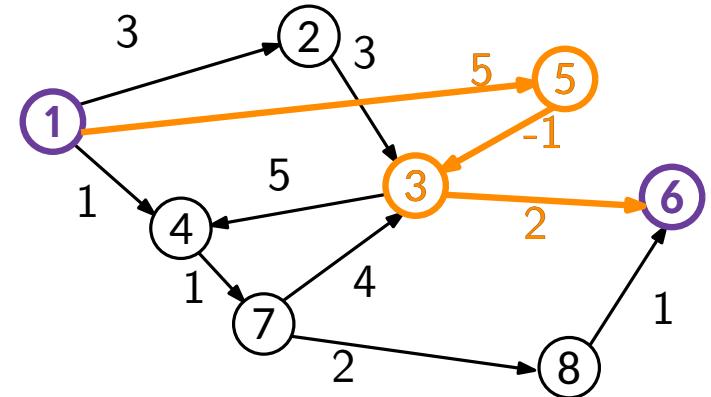
"Kosten"  $w(e)$ :  
Länge/Zeitbedarf/Preis  
Für  $e = (u, v)$  schreiben wir  
 $w(u, v) = w(e)$

$$u = 1$$

$$v = 6$$

$$\begin{matrix} 1, 2, 3, 6 \\ 1, 5, 3, 6 \end{matrix}$$

$$\begin{matrix} \text{Kosten: 8} \\ \text{Kosten: 6} \end{matrix}$$



## Definition

Sei  $G = (V, E, w)$  ein gewichteter Graph und  $u, v \in V$ .  $w(e) \in \mathbb{Z}$  beschreibt **Kosten** (Länge) der Kante  $e$ . Wir bezeichnen einen Weg  $\vec{p} = (p_0, \dots, p_k)$  von  $p_0 = u$  nach  $p_k = v$  als einen **kürzesten Weg**, wenn seine **Kosten**  $w(\vec{p}) = \sum_{i=0}^{k-1} w(p_i, p_{i+1})$  minimal sind.

# Allgemeinere Definition (gewichteten Graphen)

"Kosten"  $w(e)$ :

Länge/Zeitbedarf/Preis

Für  $e = (u, v)$  schreiben wir  
 $w(u, v) = w(e)$

$$u = 1$$

$$v = 6$$

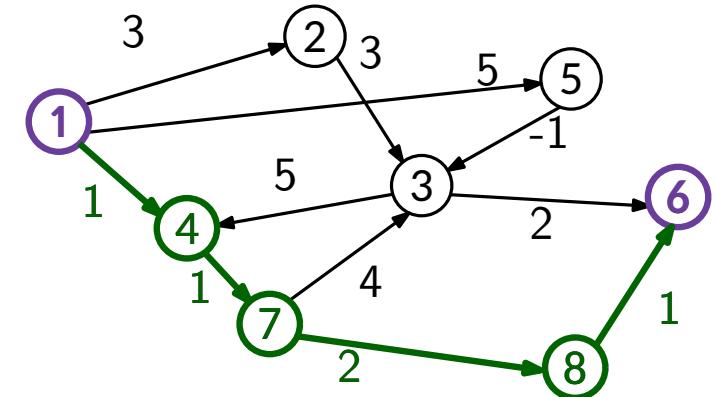
$$\begin{matrix} 1, 2, 3, 6 \\ 1, 5, 3, 6 \end{matrix}$$

$$\begin{matrix} \text{Kosten: 8} \\ \text{Kosten: 6} \end{matrix}$$

$$1, 2, 4, 8, 6$$

$$\begin{matrix} \text{Kosten: 5} \\ \rightarrow \text{kürzester Weg!} \end{matrix}$$

$$\rightarrow \delta(1, 6) = 5$$



## Definition

Sei  $G = (V, E, w)$  ein gewichteter Graph und  $u, v \in V$ .  $w(e) \in \mathbb{Z}$  beschreibt **Kosten** (Länge) der Kante  $e$ . Wir bezeichnen einen Weg  $\vec{p} = (p_0, \dots, p_k)$  von  $p_0 = u$  nach  $p_k = v$  als einen **kürzesten Weg**, wenn seine **Kosten**  $w(\vec{p}) = \sum_{i=0}^{k-1} w(p_i, p_{i+1})$  minimal sind.

Wir definieren die (**Kürzeste-Wege**-)Distanz  $\delta(u, v)$  durch

$$\delta(u, v) = \inf \left\{ \sum_{i=0}^{k-1} w(p_i, p_{i+1}) \mid \text{es existiert ein Weg } p_0, \dots, p_k \text{ von } u = p_0 \text{ nach } v = p_k \right\}$$

- Hinweise:**
- Verallgemeinerung von kürzesten Wegen in ungewichteten Graphen:  
 $\rightarrow$  wenn  $w(e) = 1$  für alle  $e \in E$  erhalten wir die Distanz in ungewichteten Graphen
  - **Frage:** Sei  $v$  erreichbar von  $u$ . Existiert auch immer ein **kürzester Weg** von  $u$  nach  $v$ ?

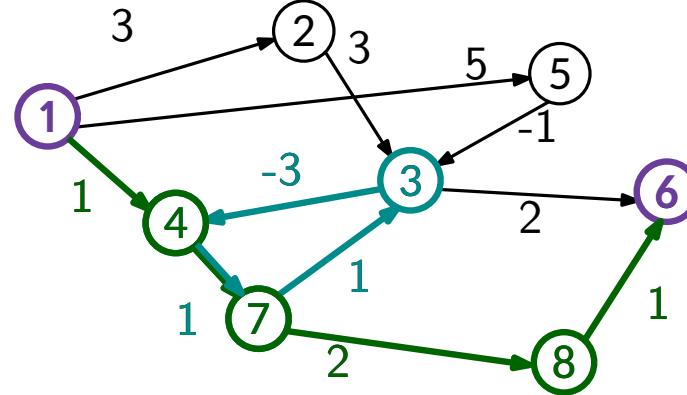
# Negative Zyklen

## Negativer Zyklus:

Ein Zyklus  $z_0, \dots, z_k$  (also  $z_0 = z_k = z$  für ein  $z$ ), sodass  $\sum_{i=0}^{k-1} w(z_i, z_{i+1}) < 0$

7, 3, 4, 7 ist negativer Zyklus (Kosten: -1)

Weg	Kosten
1, 4, 7, 8, 6	5
1, 4, 7, <u>3, 4, 7</u> , 8, 6	4
1, 4, 7, <u>3, 4, 7</u> , <u>3, 4, 7</u> , 8, 6	3
...	...



Wenn Zyklen mit negativen Kosten existieren:

- mehrfaches Durchlaufen des Zyklus' liefert beliebig kleine Kosten!

Wenn keine Zyklen mit negativen Kosten existieren:

- kann ein Zyklus die Kosten niemals verringern  
→ der kürzeste Weg ist ein **Pfad**!

## Lemma.

Es gilt:

$$\delta(u, v) = \begin{cases} \infty & \text{wenn es keinen Weg von } u \text{ nach } v \text{ gibt,} \\ -\infty & \text{wenn es negativen Zyklus } z, z_1, \dots, z_{k-1}, z \text{ gibt, sodass } u \rightsquigarrow z \rightsquigarrow v \\ w(\vec{p}) & \text{ansonsten, wobei } \vec{p} \text{ ein kürzester Weg von } u \text{ nach } v \text{ ist.} \end{cases}$$

$\vec{p}$  ist sogar ein Pfad

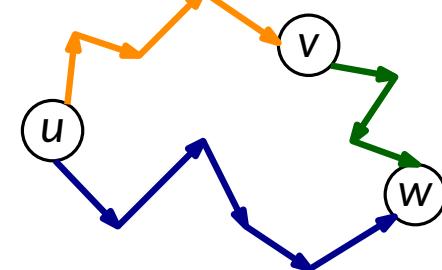
# Wichtige Eigenschaften

Besondere Eigenschaften der Kürzeste-Wege-Distanz  $\delta$ :

## 1. Dreiecksungleichung

Für alle  $u, v, w \in V$  gilt:

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



**Beweis:** Nimm einen kürzesten Weg von  $u$  nach  $v$  und hänge ihn an an einen kürzesten Pfad von  $v$  nach  $w$ .

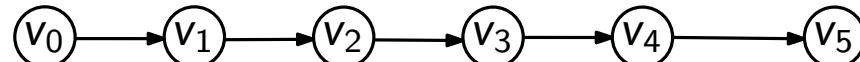
Wir erhalten einen Weg von  $u$  nach  $w$  der Länge  $\delta(u, v) + \delta(v, w)$ . □

## 2. Optimalität von Subpfaden

Sei  $v_0, \dots, v_k$  ein kürzester Weg.

Dann gilt für alle  $0 \leq a < b \leq k$ :

$v_a, v_{a+1}, \dots, v_b$  ist ein kürzester Weg.



### Beweis (durch Widerspruch):

Angenommen, es gibt einen Weg  $\vec{p} = (p_0, \dots, p_\ell)$  von  $p_0 = v_a$  nach  $p_\ell = v_b$  mit Kosten  $w(\vec{p}) < \sum_{i=a}^{b-1} w(v_i, v_{i+1})$

Dann liefert  $v_0, v_1, \dots, v_{a-1}, \underline{v_a, p_1, \dots, p_{\ell-1}, v_b, \dots, v_k}$  einen kürzeren Weg von  $v_0$  nach  $v_k$ , denn er hat Kosten:

$$\begin{aligned} \left( \sum_{i=0}^{a-1} w(v_i, v_{i+1}) \right) + w(\vec{p}) + \left( \sum_{i=b}^{k-1} w(v_i, v_{i+1}) \right) &< \left( \sum_{i=0}^{a-1} w(v_i, v_{i+1}) \right) + \left( \sum_{i=a}^{b-1} w(v_i, v_{i+1}) \right) + \left( \sum_{i=b}^{k-1} w(v_i, v_{i+1}) \right) \\ &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \quad \text{dazu, dass } v_0, \dots, v_k \text{ kürzester Weg ist.} \end{aligned}$$

# Single-Source Shortest Path Problem (SSSP)

---

Eines der wichtigsten algorithmischen Problemen auf Graphen:

## Single-Source Shortest Path (SSSP)

**Gegeben:** Graph  $G = (V, E, w)$ , Startknoten  $s \in V$  (auch genannt Quellknoten/source)

**Gesucht:** die Distanzen von  $s$  zu allen anderen Knoten, d.h.

$$\delta(s, v_1), \dots, \delta(s, v_n) \text{ wobei } V = \{v_1, \dots, v_n\}$$

In der Praxis ist man häufig nur an einer einzigen Distanz – also  $\delta(s, t)$  für ein bestimmtes Paar  $(s, t) \in V \times V$  – interessiert.  
→ um diese Distanz zu berechnen, werden wir i.d.R. ohnehin zusätzliche Distanzen berechnen.

## Viele Anwendungen:

- Routing: Wie komme ich am schnellsten von A nach B?
- wichtiger Baustein: komplexere Aufgabenstellungen benötigen Kürzeste-Wege-Informationen

# **SSSP für nichtnegative Kantengewichte: Dijkstra-Algorithmus**

# Dijkstra's Algorithmus

---

In vielen Anwendungen haben wir keine negativen Kantengewichte!

Beispiele: geographische Distanzen, Zeitbedarf um entlang einer Kante zu "reisen", etc.

→ keine negative Zyklen!  $\Rightarrow$  Es gibt keine  $u, v \in V$  mit  $\delta(u, v) = -\infty$ .

## Theorem (Dijkstra's Algorithmus).

Sei  $G = (V, E, w)$  ein gewichteter Graph mit  $w(e) \geq 0$  für alle  $e \in E$ .

Sei  $s \in V$  ein gegebener Startknoten.

Wir können alle Distanzen  $\delta(s, v)$  mit  $v \in V$  in Zeit  $O(n \log n + m)$  berechnen.

# Subroutine: Kante relaxieren

---

## Grundsätzliches Vorgehen:

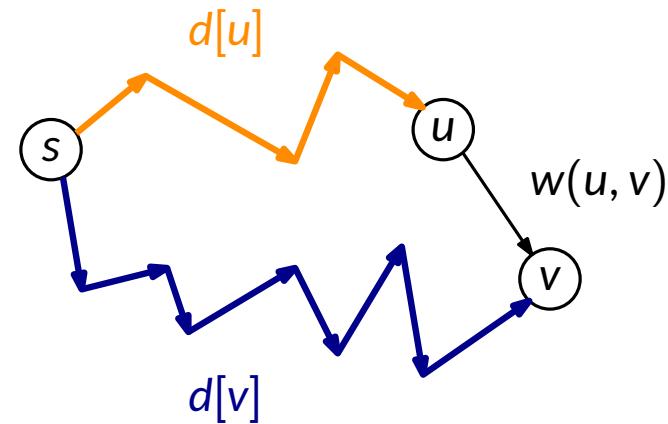
- wir werden zu jedem Knoten  $v$  eine valide **Schätzdistanz**  $d[v]$  aufrechterhalten:

$$\delta(s, v) \leq d[v]$$

- wir können anfangs  $d[v] = \infty$  setzen
- im Verlauf des Algorithmus wollen wir die Schätzdistanz verbessern ( $d[v]$  von oben an  $\delta(s, v)$  annähern)
- dazu werden wir **Kanten relaxieren**

```
relaxiere( $u, v$ ): //relaxiert Kante  $e = (u, v)$ 
```

```
if  $d[v] > d[u] + w(u, v)$  then  
     $d[v] = d[u] + w(u, v)$ 
```



# Subroutine: Kante relaxieren

---

## Grundsätzliches Vorgehen:

- wir werden zu jedem Knoten  $v$  eine valide **Schätzdistanz**  $d[v]$  aufrechterhalten:

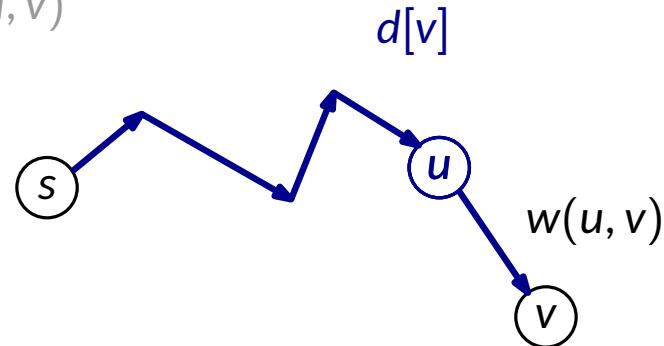
$$\delta(s, v) \leq d[v]$$

- wir können anfangs  $d[v] = \infty$  setzen
- im Verlauf des Algorithmus wollen wir die Schätzdistanz verbessern ( $d[v]$  von oben an  $\delta(s, v)$  annähern)
- dazu werden wir **Kanten relaxieren**

```
relaxiere( $u, v$ ): //relaxiert Kante  $e = (u, v)$ 
```

```
if  $d[v] > d[u] + w(u, v)$  then
```

```
     $d[v] = d[u] + w(u, v)$ 
```



# Subroutine: Kante relaxieren

## Grundsätzliches Vorgehen:

- wir werden zu jedem Knoten  $v$  eine valide **Schätzdistanz**  $d[v]$  aufrechterhalten:

$$\delta(s, v) \leq d[v]$$

- wir können anfangs  $d[v] = \infty$  setzen
- im Verlauf des Algorithmus wollen wir die Schätzdistanz verbessern ( $d[v]$  von oben an  $\delta(s, v)$  annähern)
- dazu werden wir **Kanten relaxieren**

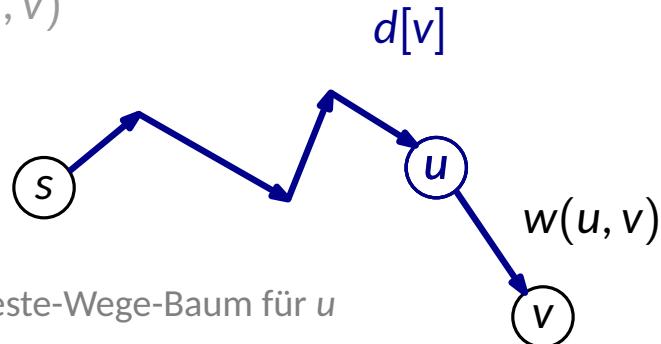
```
relaxiere( $u, v$ ): //relaxiert Kante  $e = (u, v)$ 
```

```
if  $d[v] > d[u] + w(u, v)$  then
```

```
     $d[v] = d[u] + w(u, v)$ 
```

```
     $p[v] = u$ 
```

speichert Vorgänger in einem Kürzeste-Wege-Baum für  $u$



## Schätzdistanzen bleiben valide!

Wenn  $\delta(s, u) \leq d[u]$ ,

dann bleibt auch  $\delta(s, v) \leq d[v]$  erhalten.

# Algorithmenbeschreibung von Dijkstra

---

## Algorithmenbeschreibung:

- wir initialisieren die Schätzdistanzen:

$$d[s] = 0 \quad d[v] = \infty \quad \text{für alle } v \in V \setminus \{s\}$$

- **Solange** es noch unbetrachtete Knoten gibt:

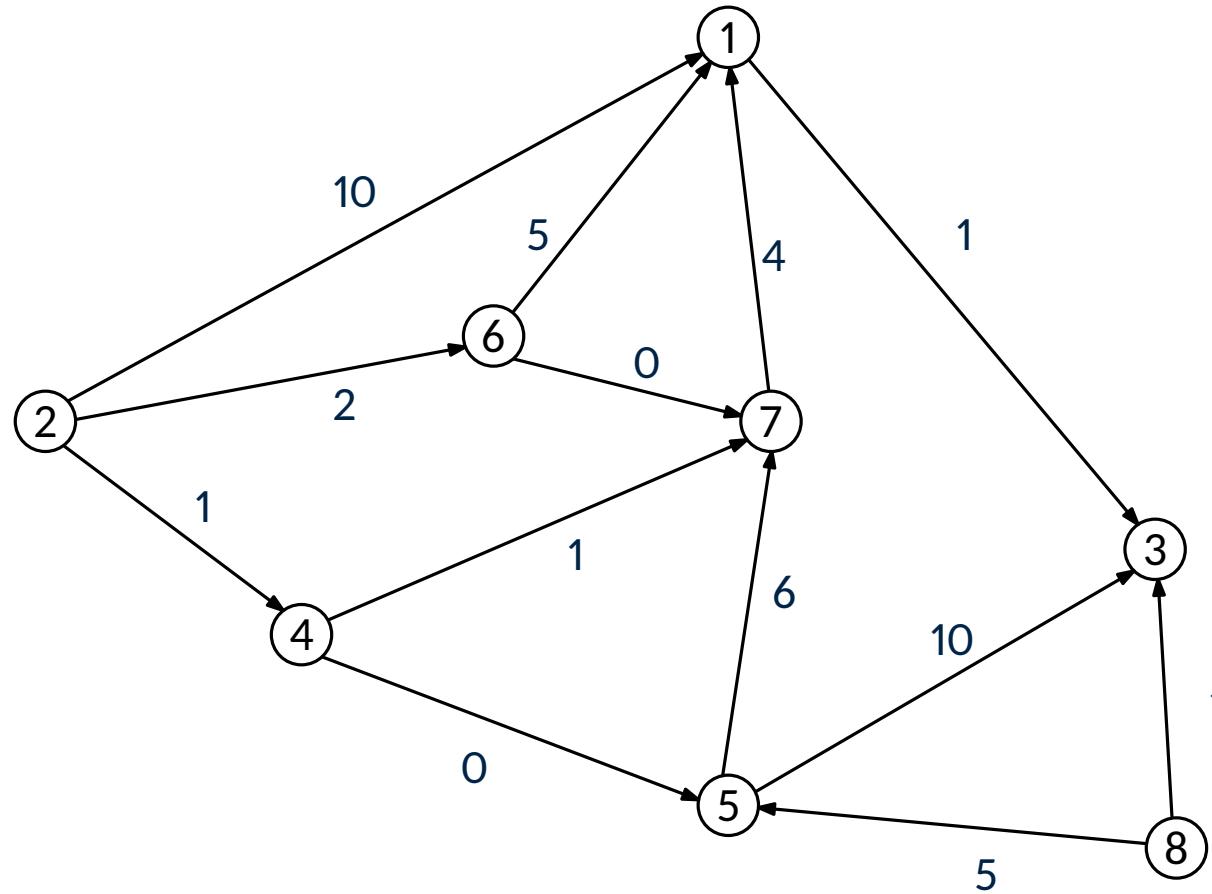
- Betrachte den Knoten  $u$  mit kleinster Schätzdistanz  $d[u]$

- **für jede** von  $u$  ausgehende Kante  $(u, v) \in E$ :

- **relaxiere**  $(u, v)$       //→ ggf. verbesserte Schätzdistanz für  $v$

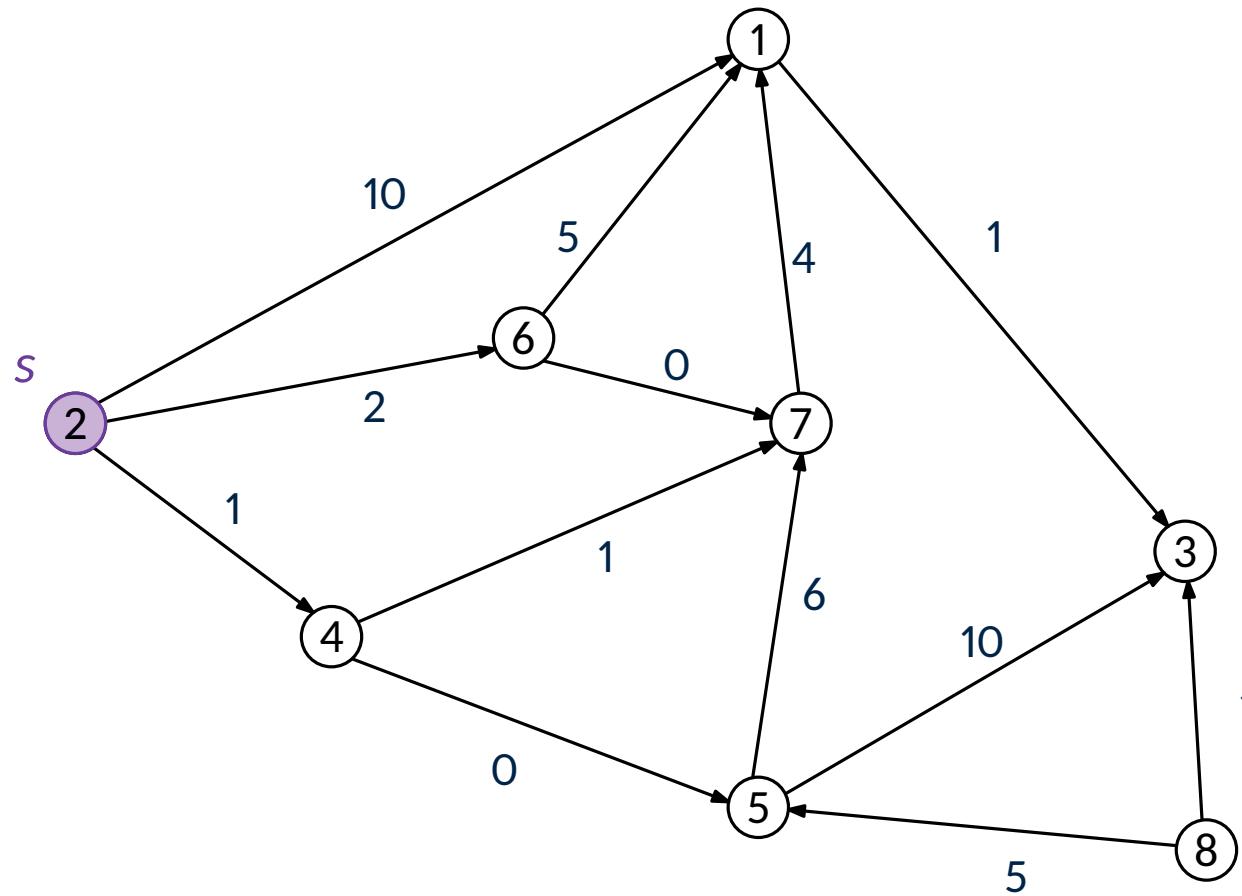
# Dijkstra-Algorithmus: Beispiel

---

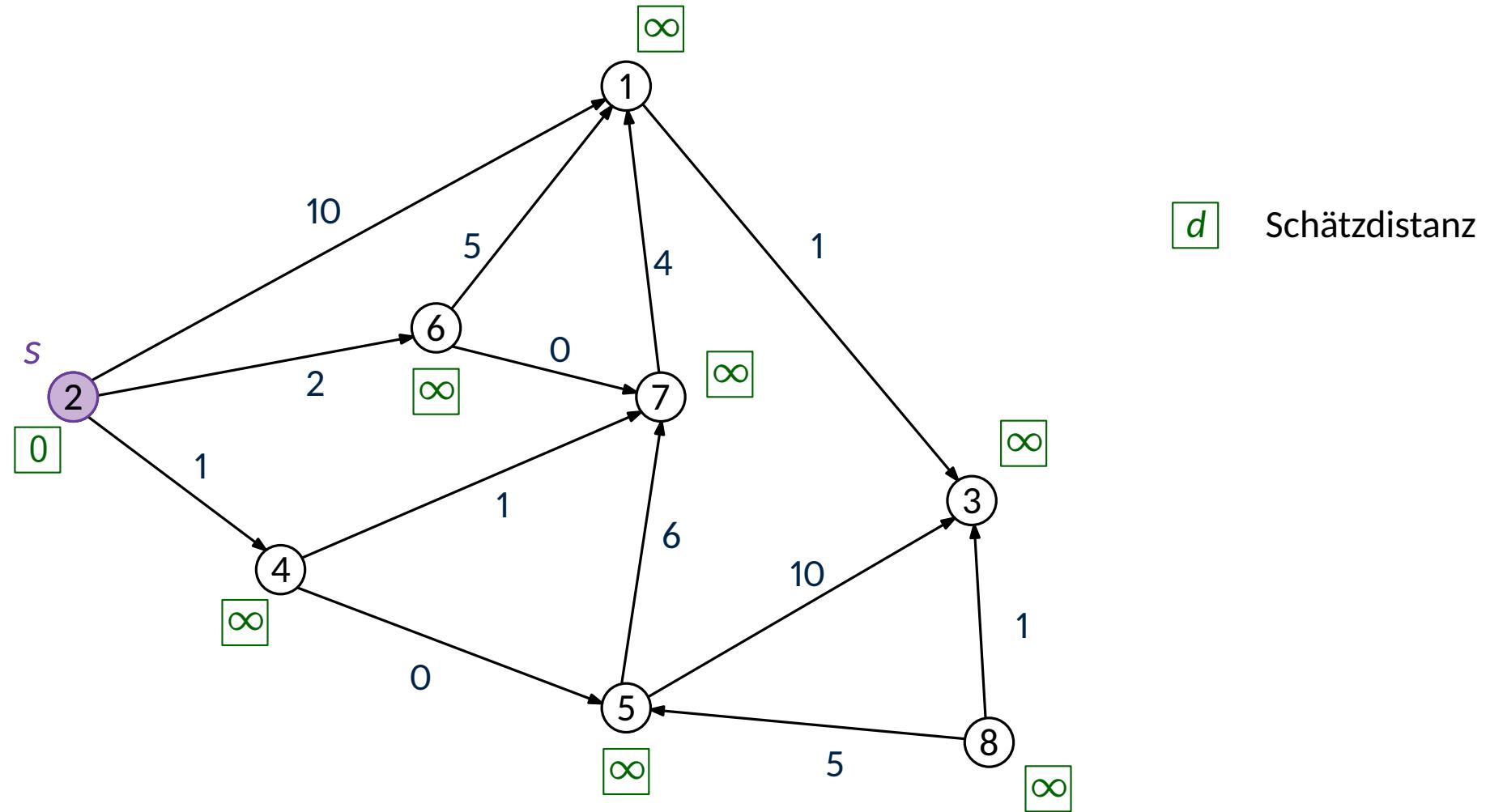


# Dijkstra-Algorithmus: Beispiel

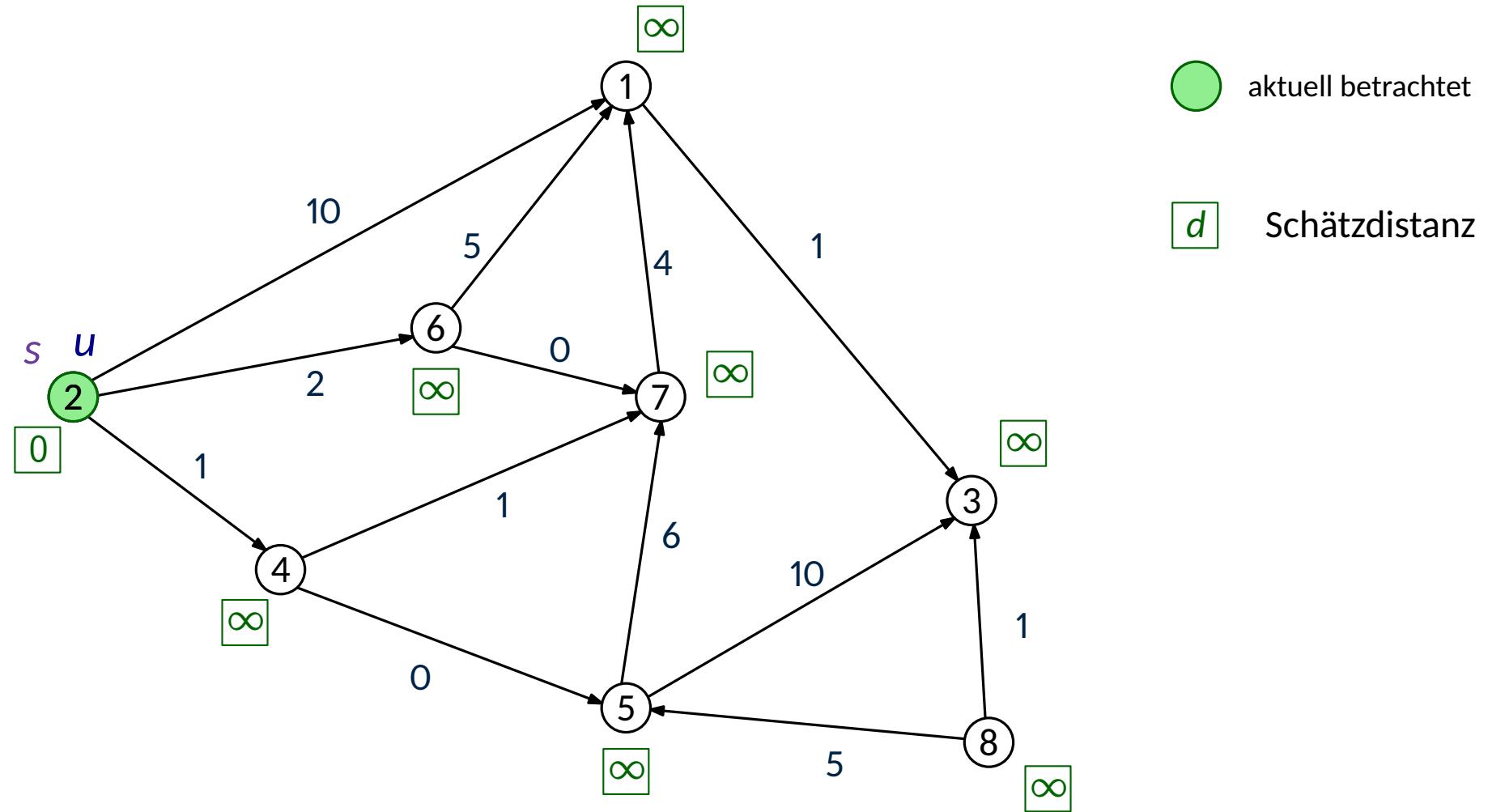
---



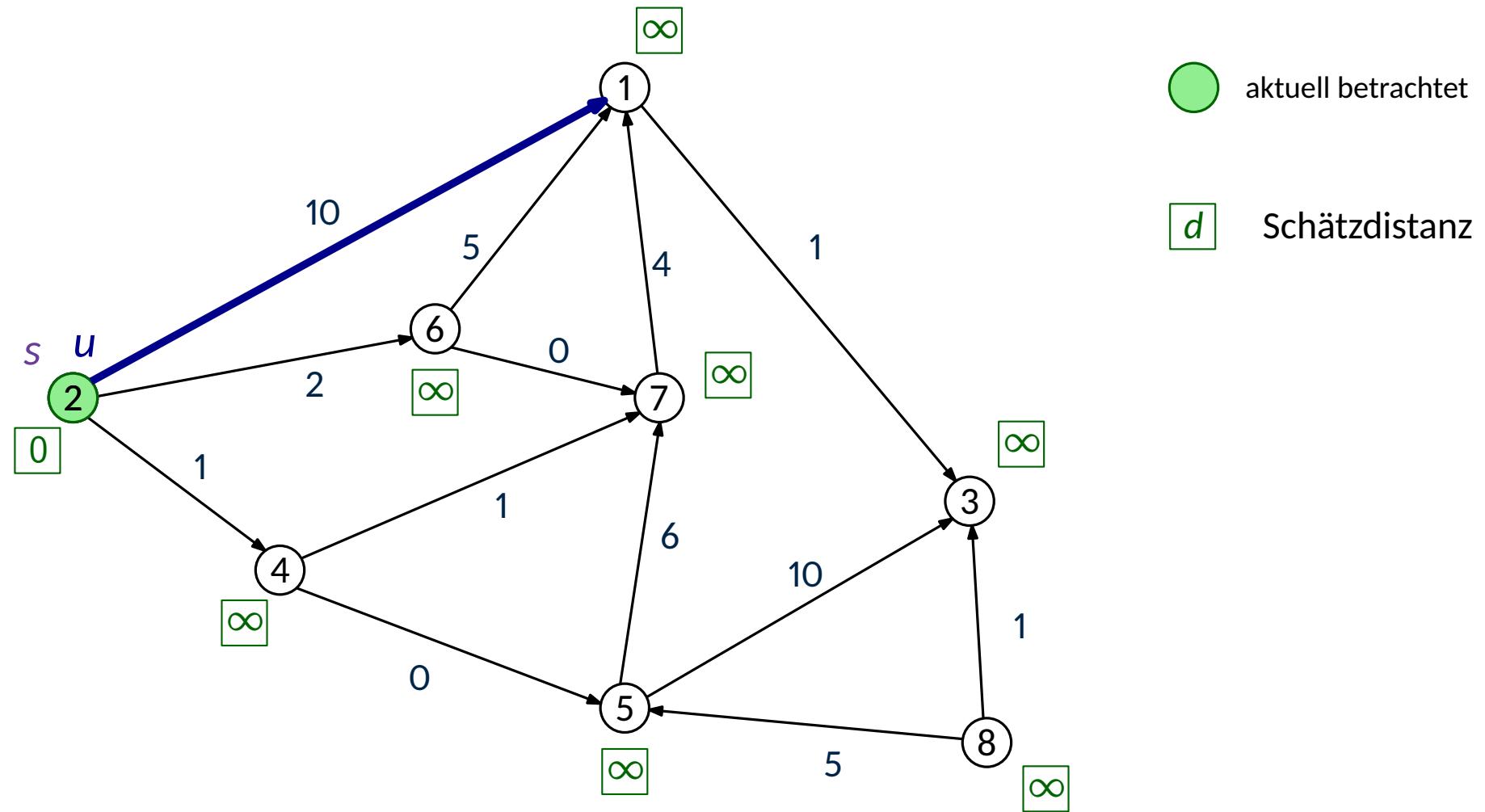
# Dijkstra-Algorithmus: Beispiel



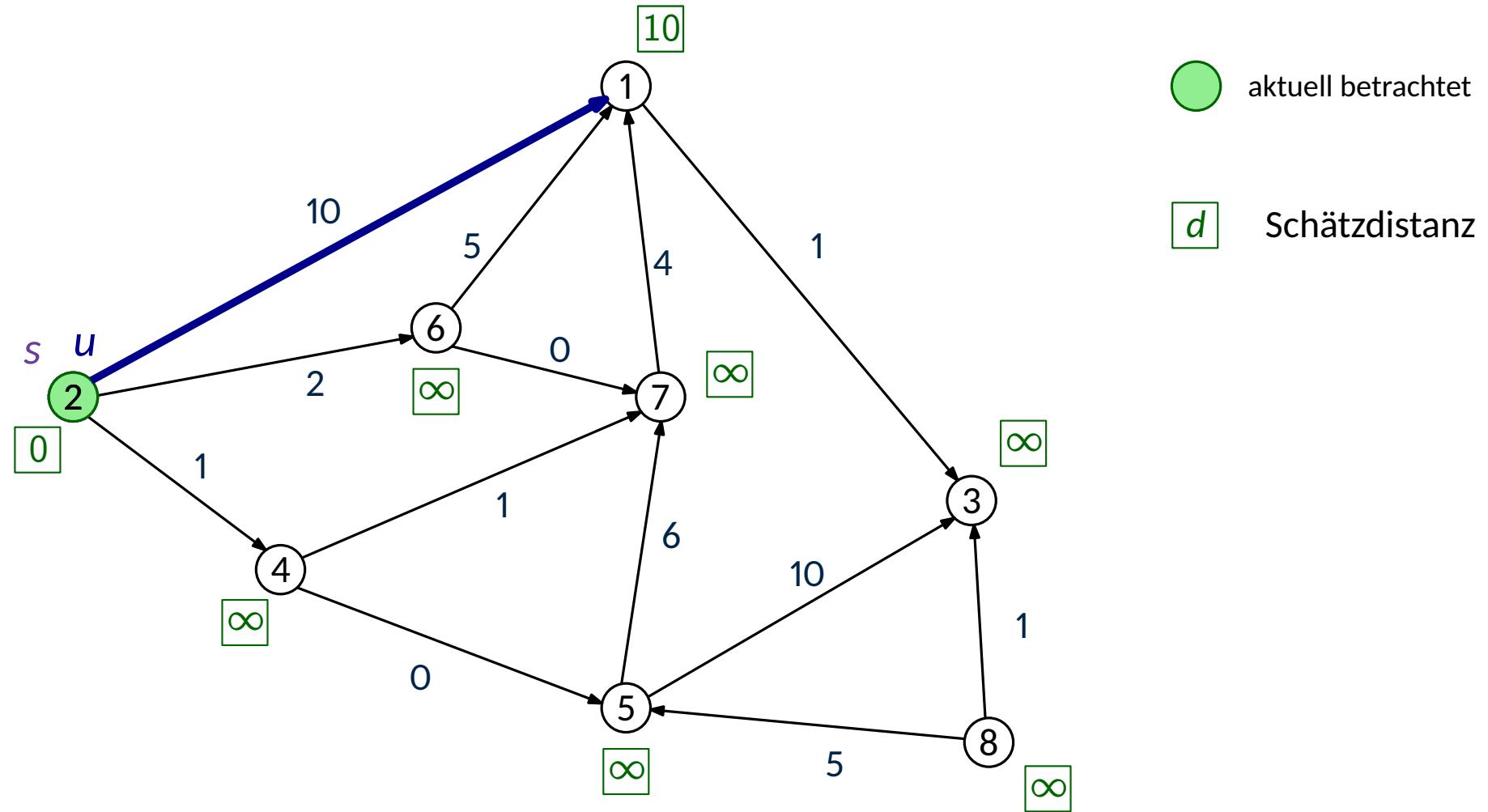
# Dijkstra-Algorithmus: Beispiel



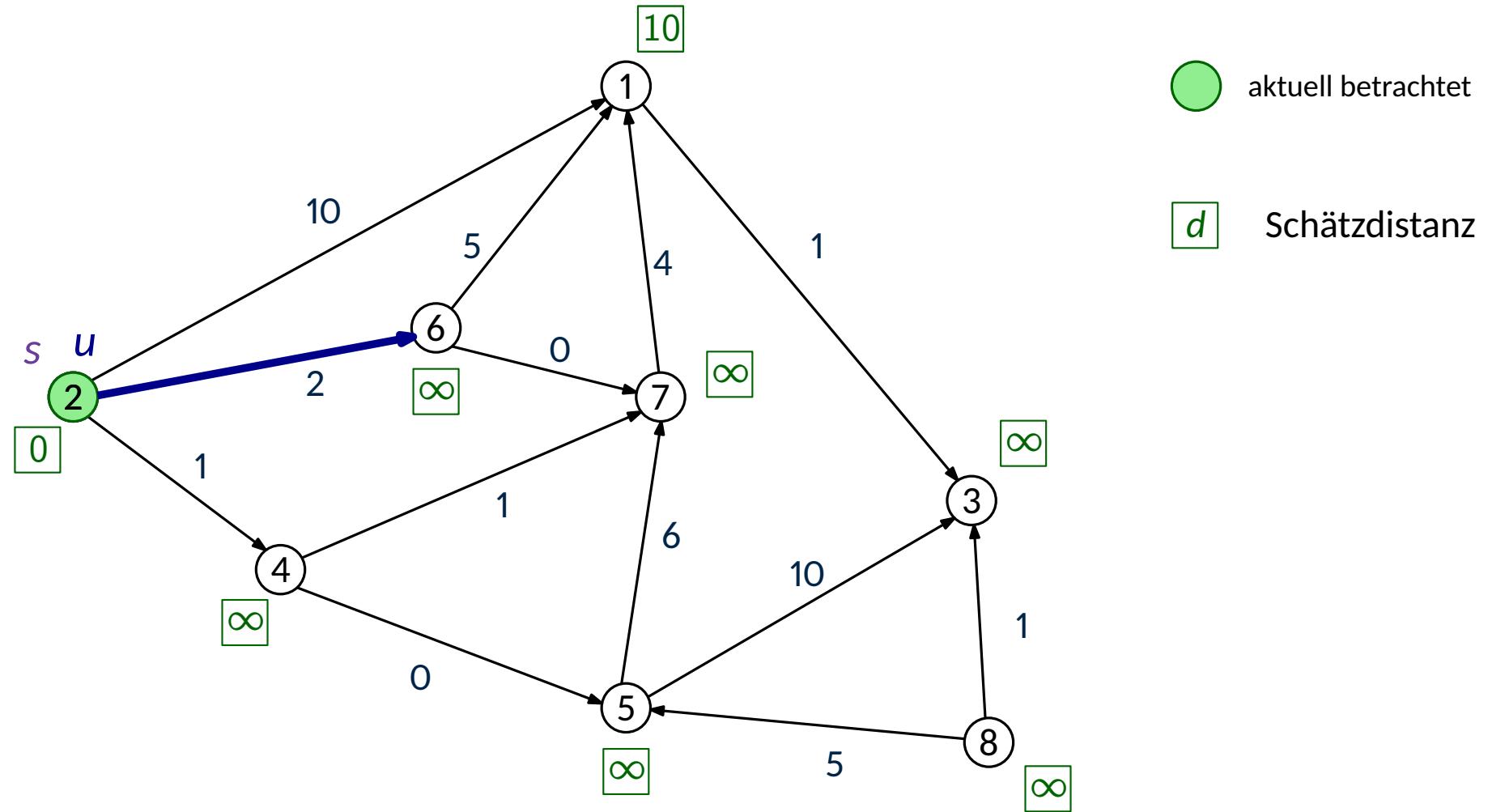
# Dijkstra-Algorithmus: Beispiel



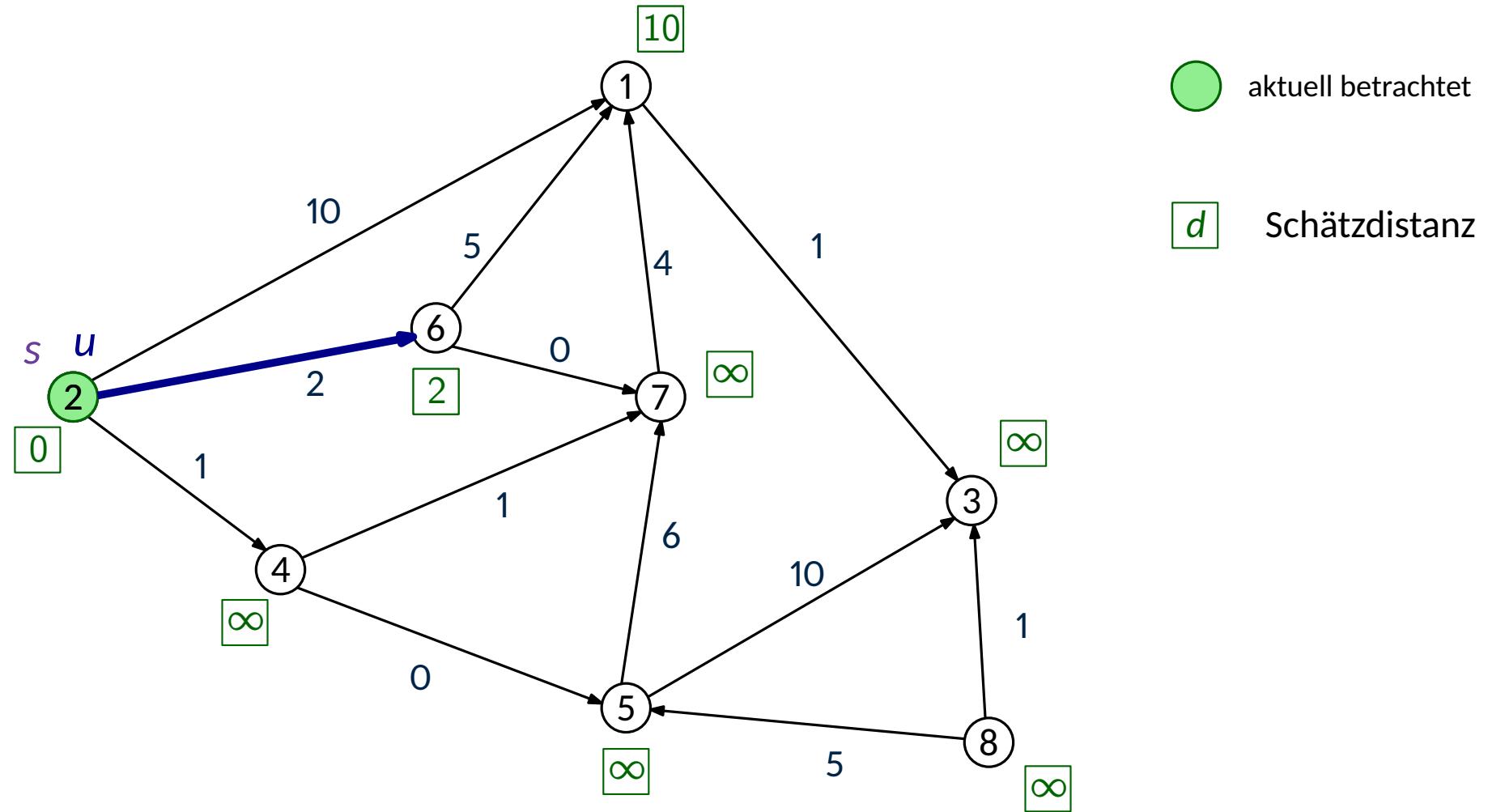
# Dijkstra-Algorithmus: Beispiel



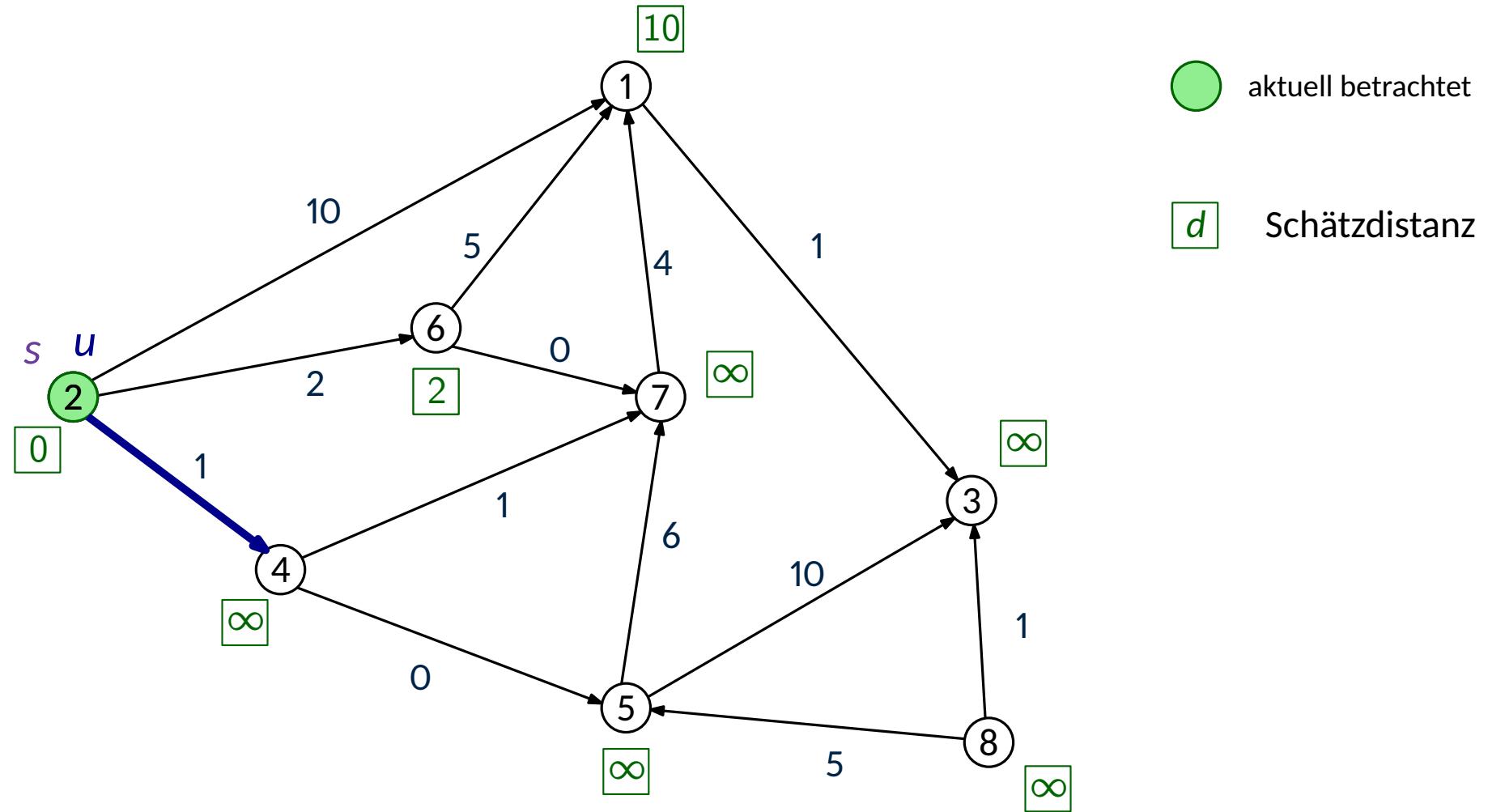
# Dijkstra-Algorithmus: Beispiel



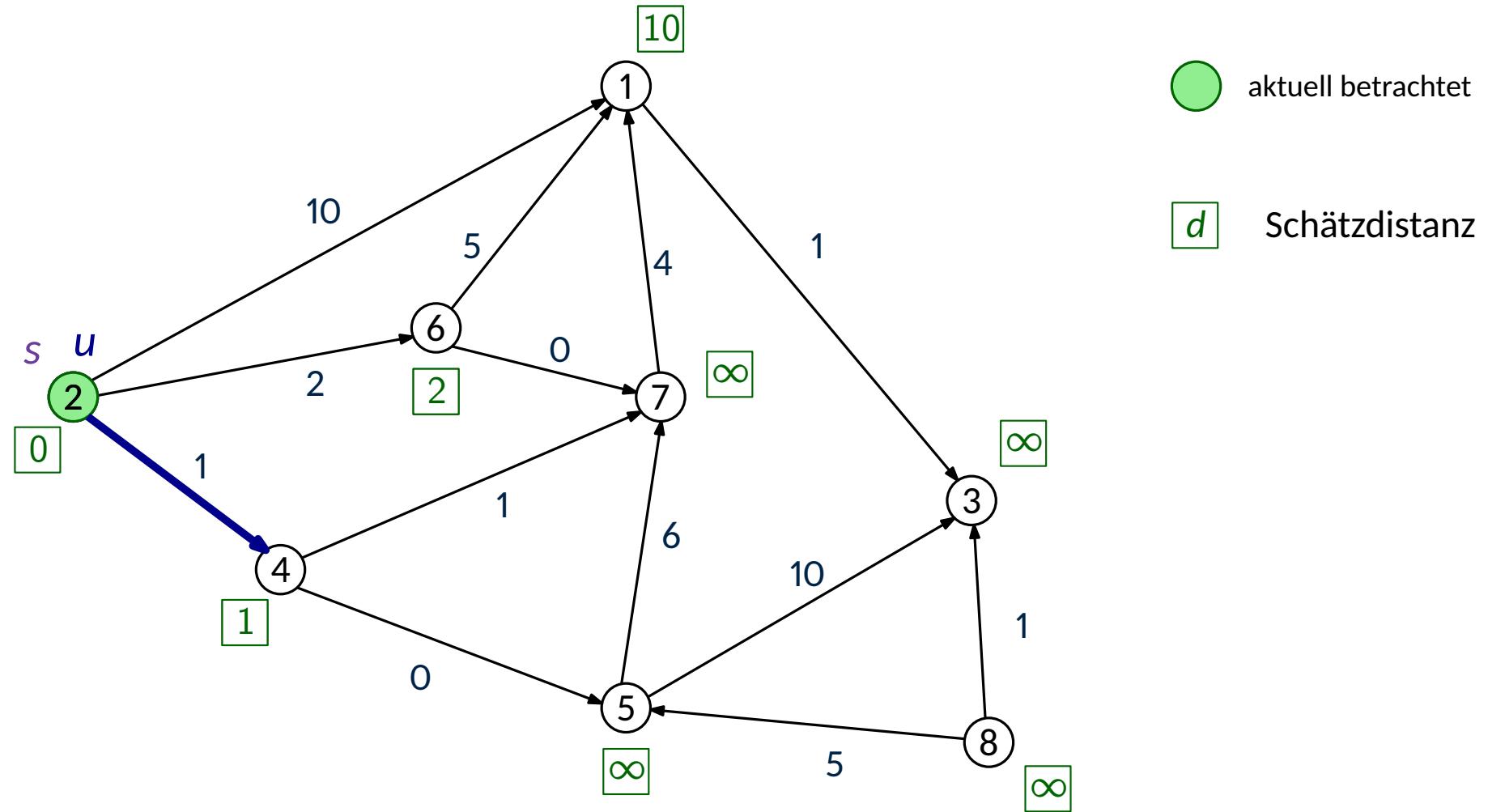
# Dijkstra-Algorithmus: Beispiel



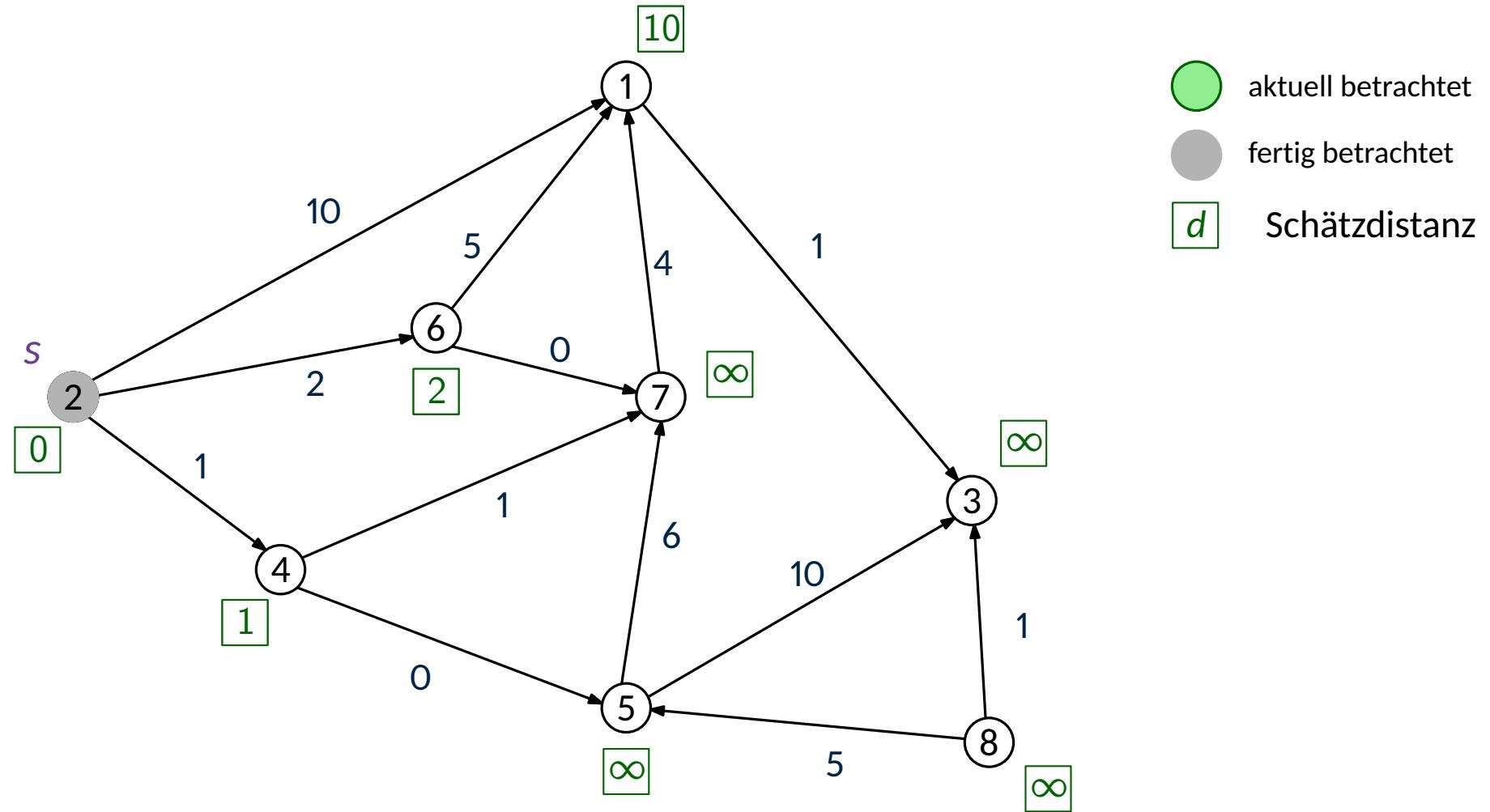
# Dijkstra-Algorithmus: Beispiel



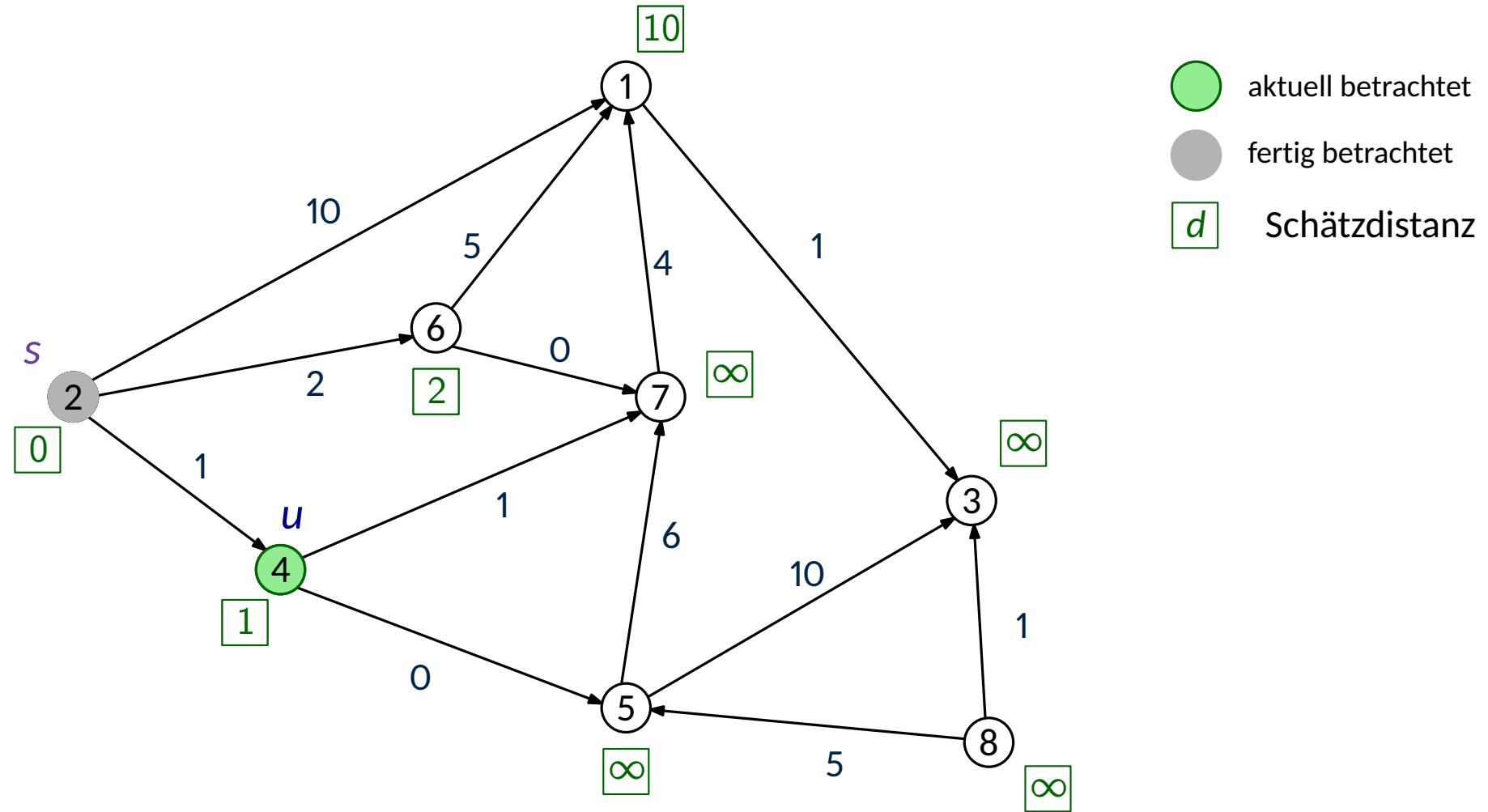
# Dijkstra-Algorithmus: Beispiel



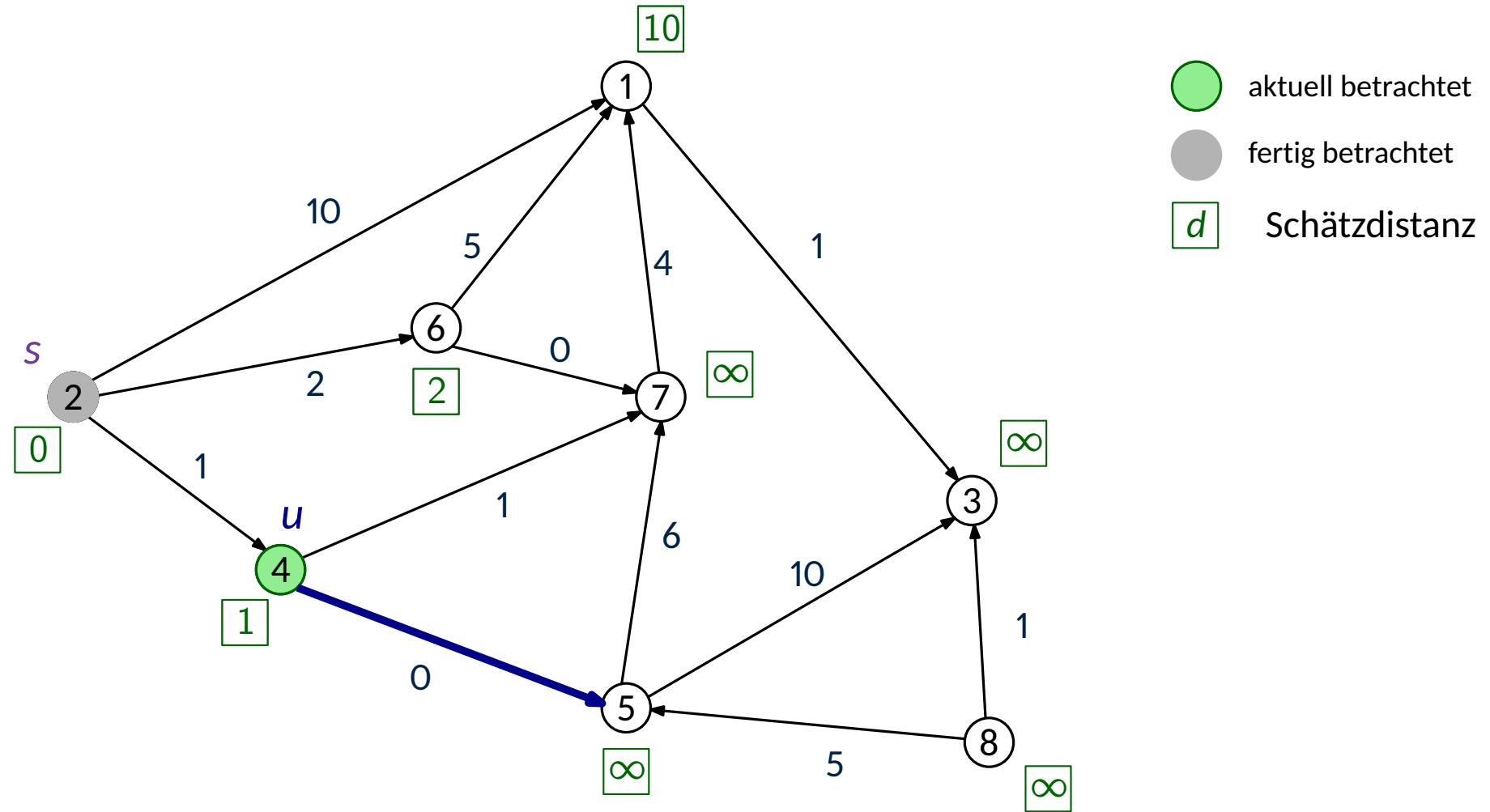
# Dijkstra-Algorithmus: Beispiel



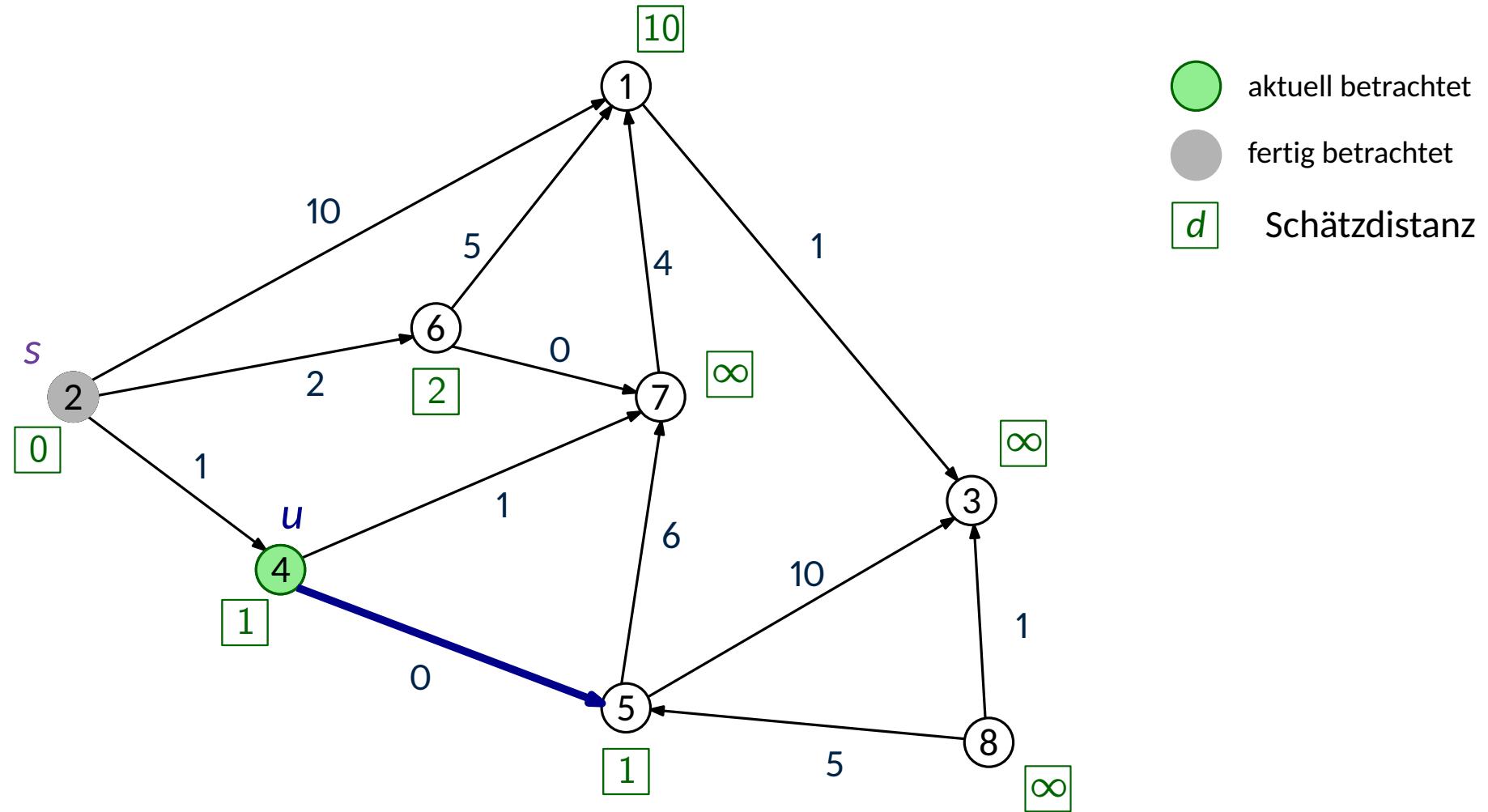
# Dijkstra-Algorithmus: Beispiel



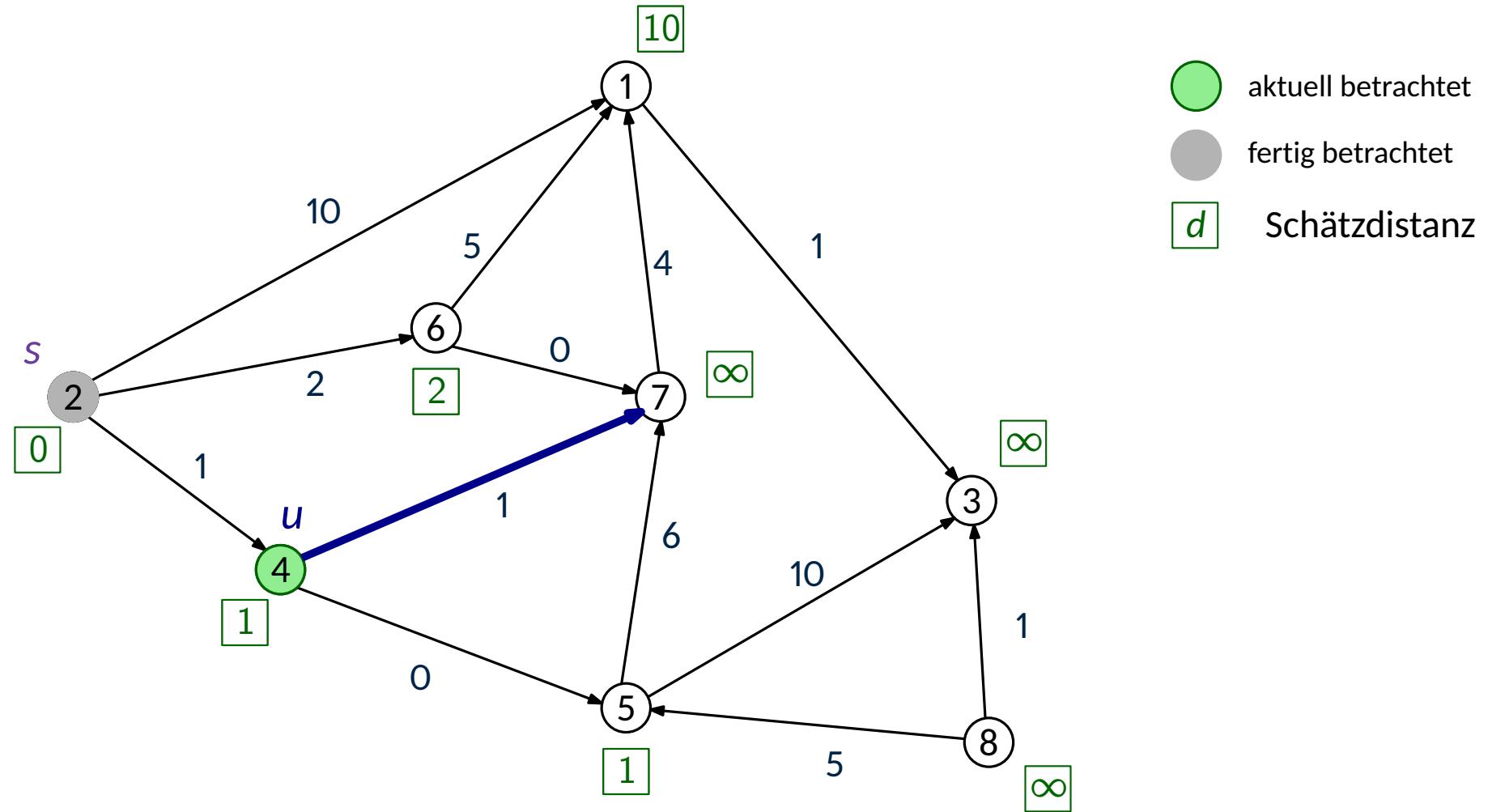
# Dijkstra-Algorithmus: Beispiel



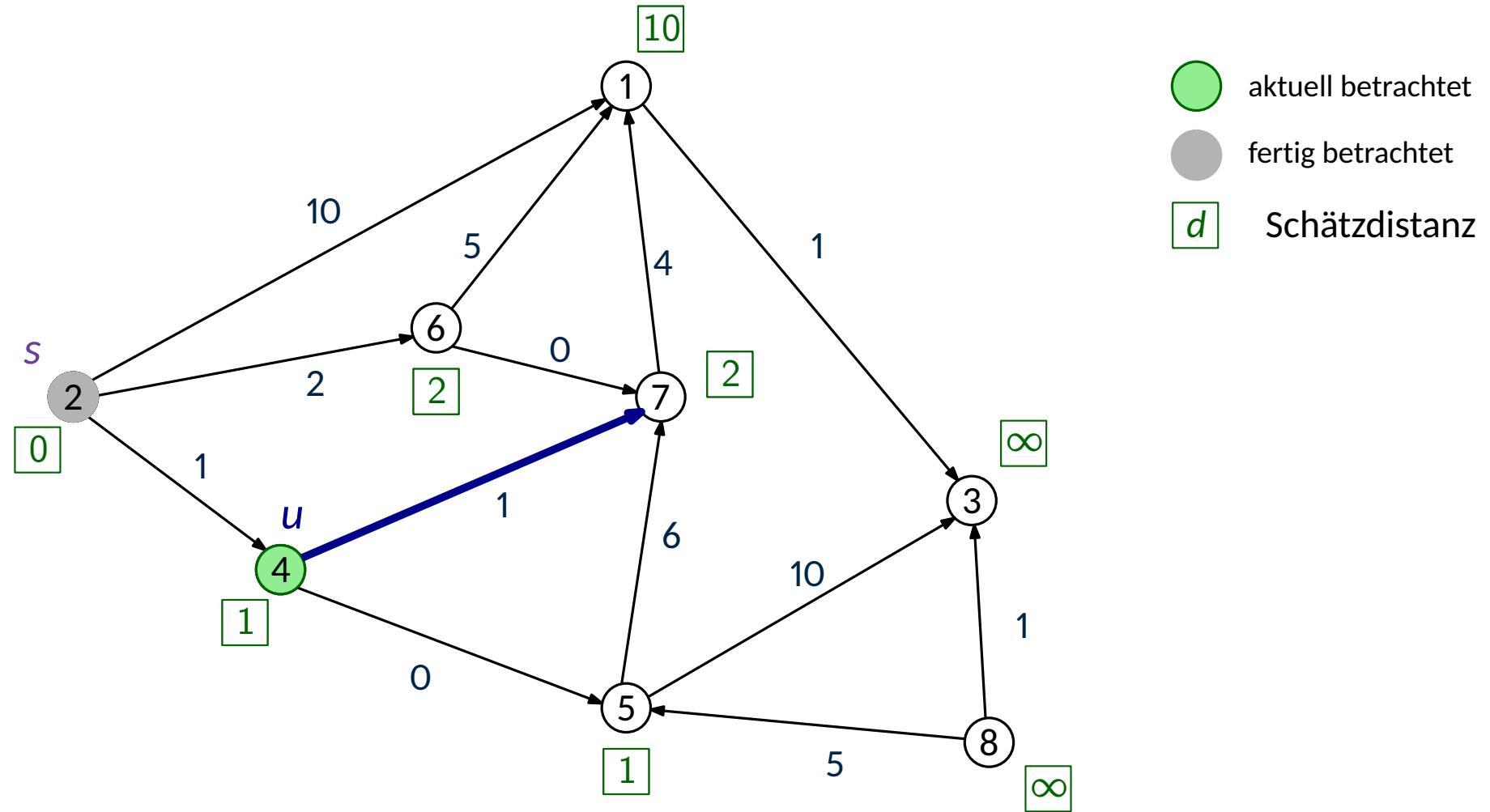
# Dijkstra-Algorithmus: Beispiel



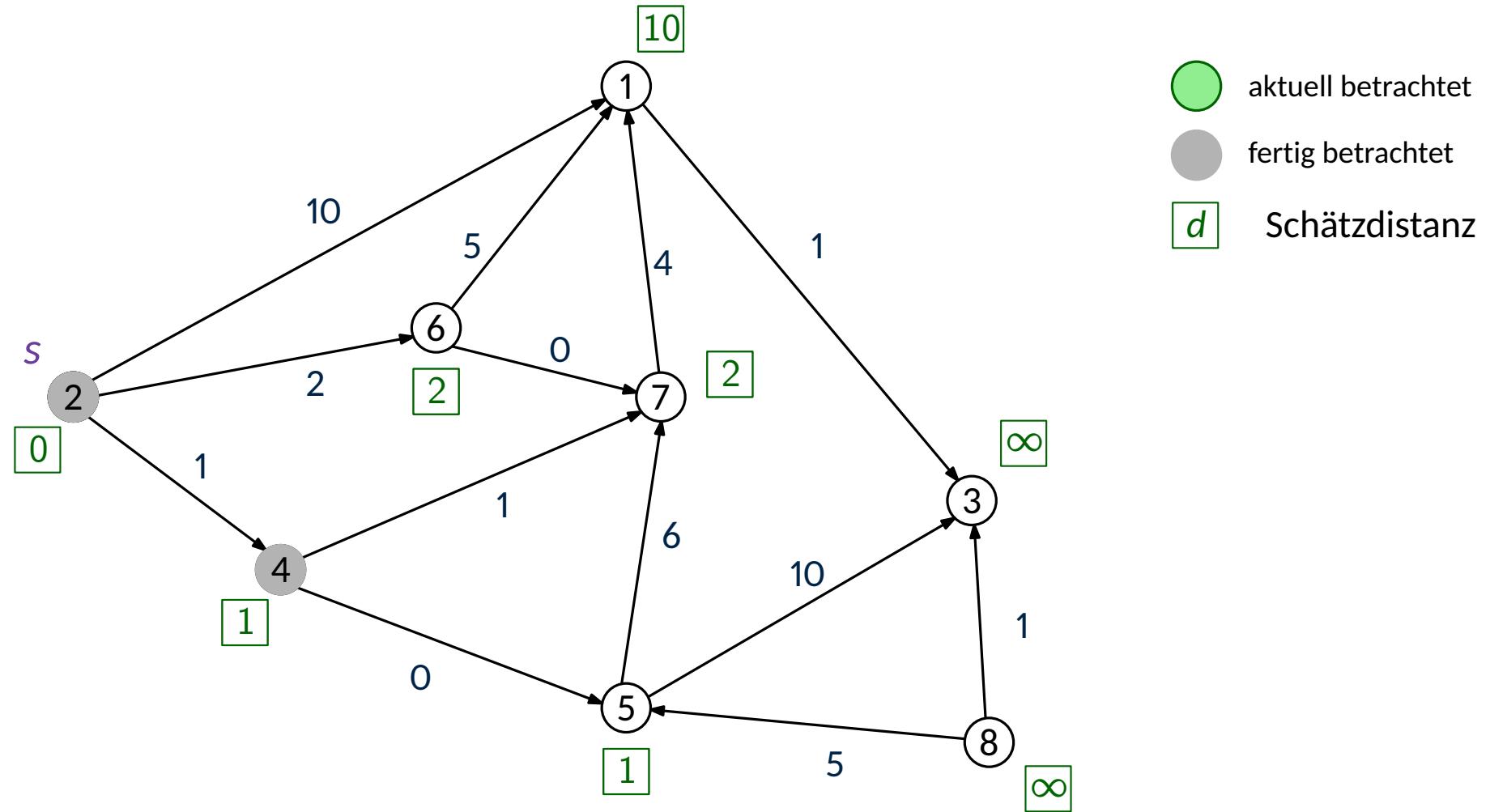
# Dijkstra-Algorithmus: Beispiel



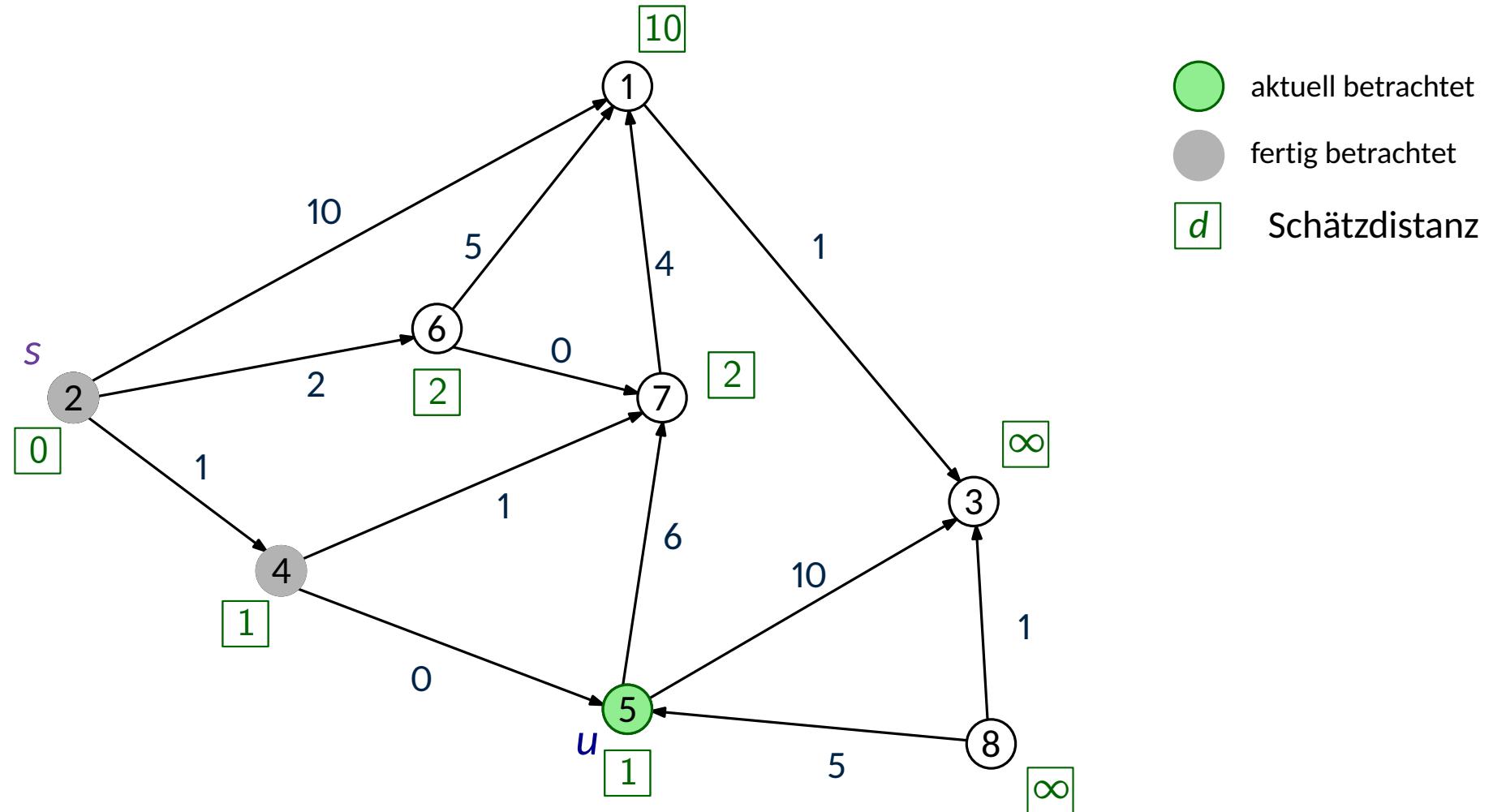
# Dijkstra-Algorithmus: Beispiel



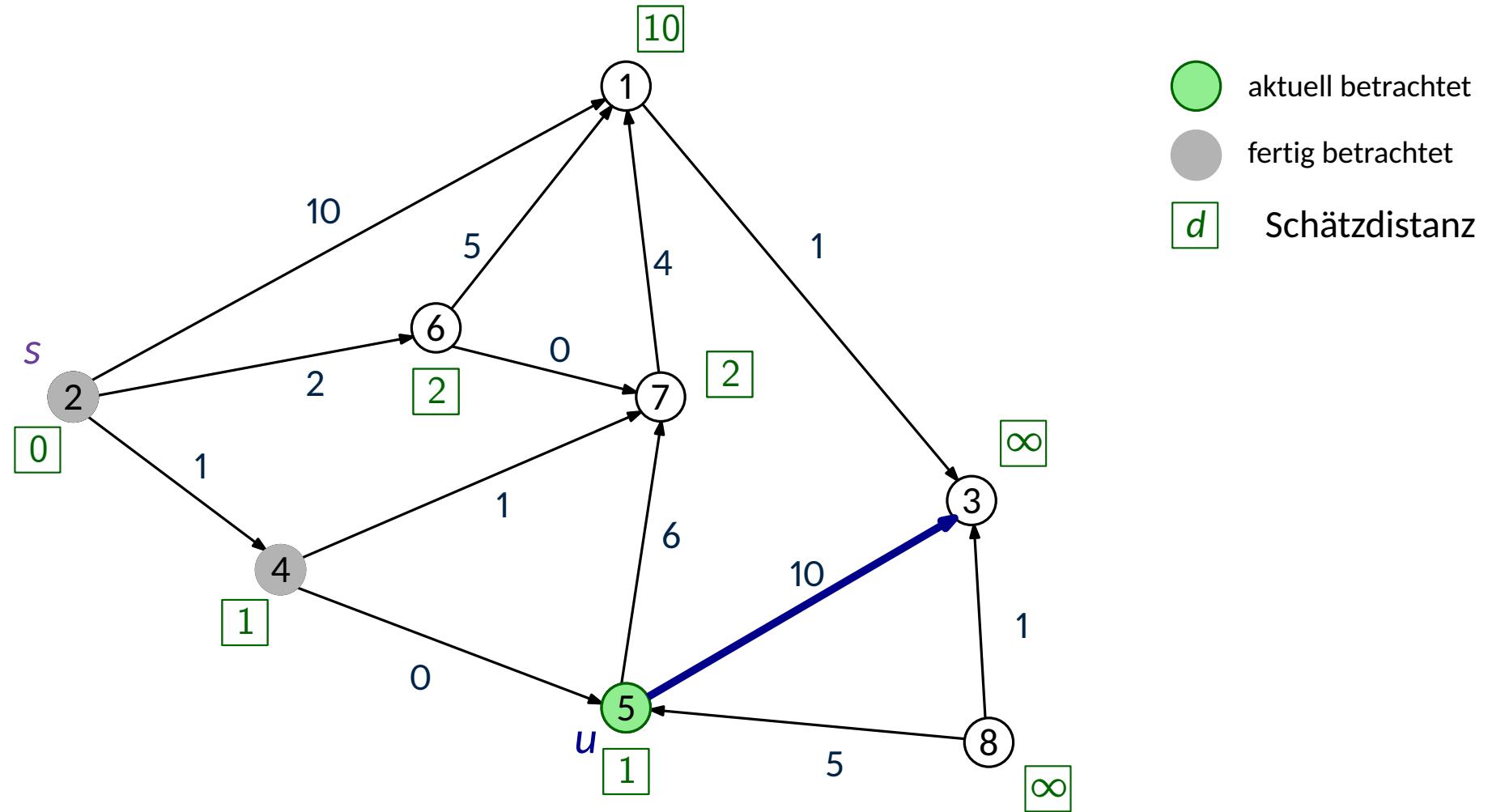
# Dijkstra-Algorithmus: Beispiel



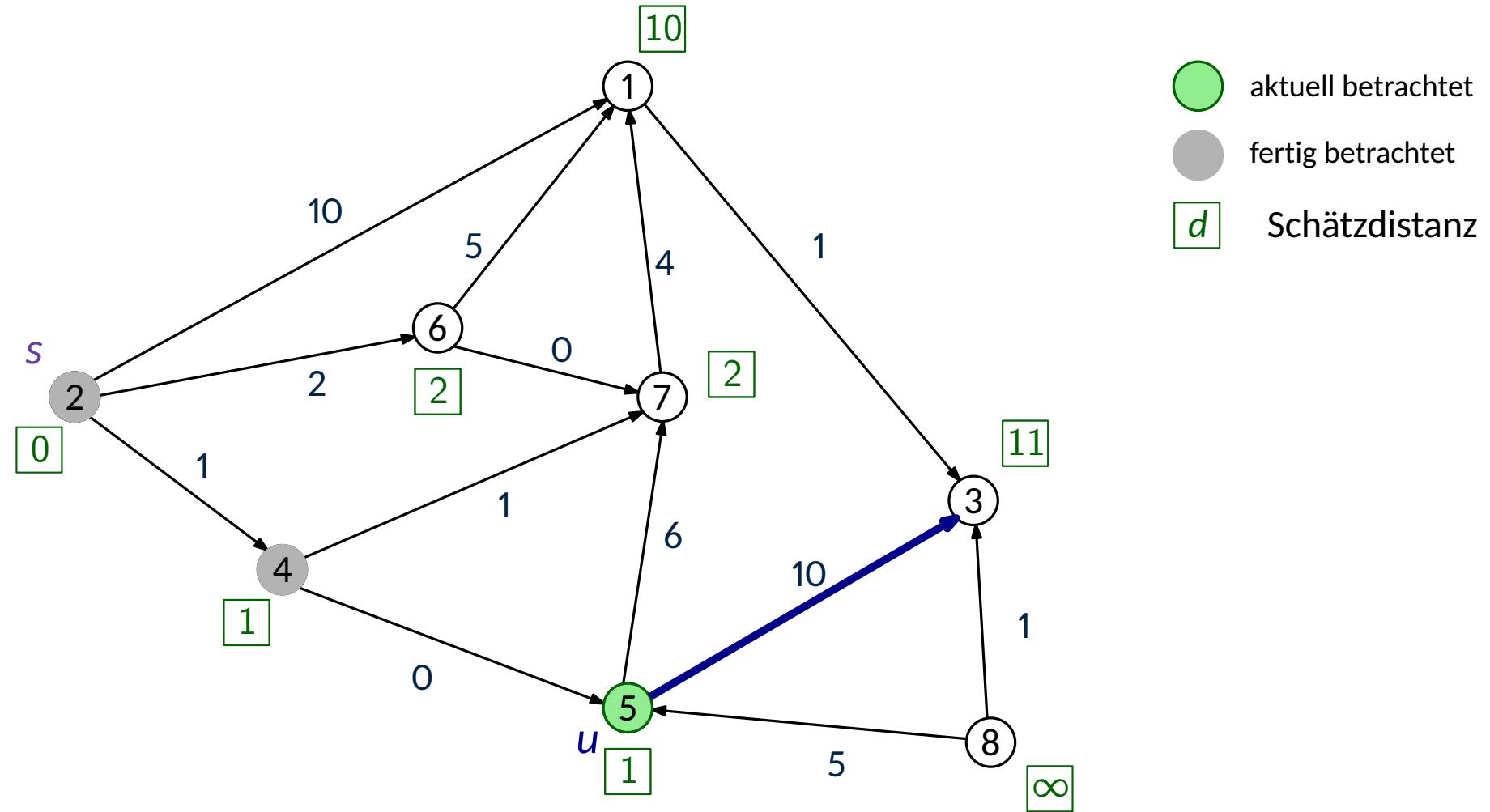
# Dijkstra-Algorithmus: Beispiel



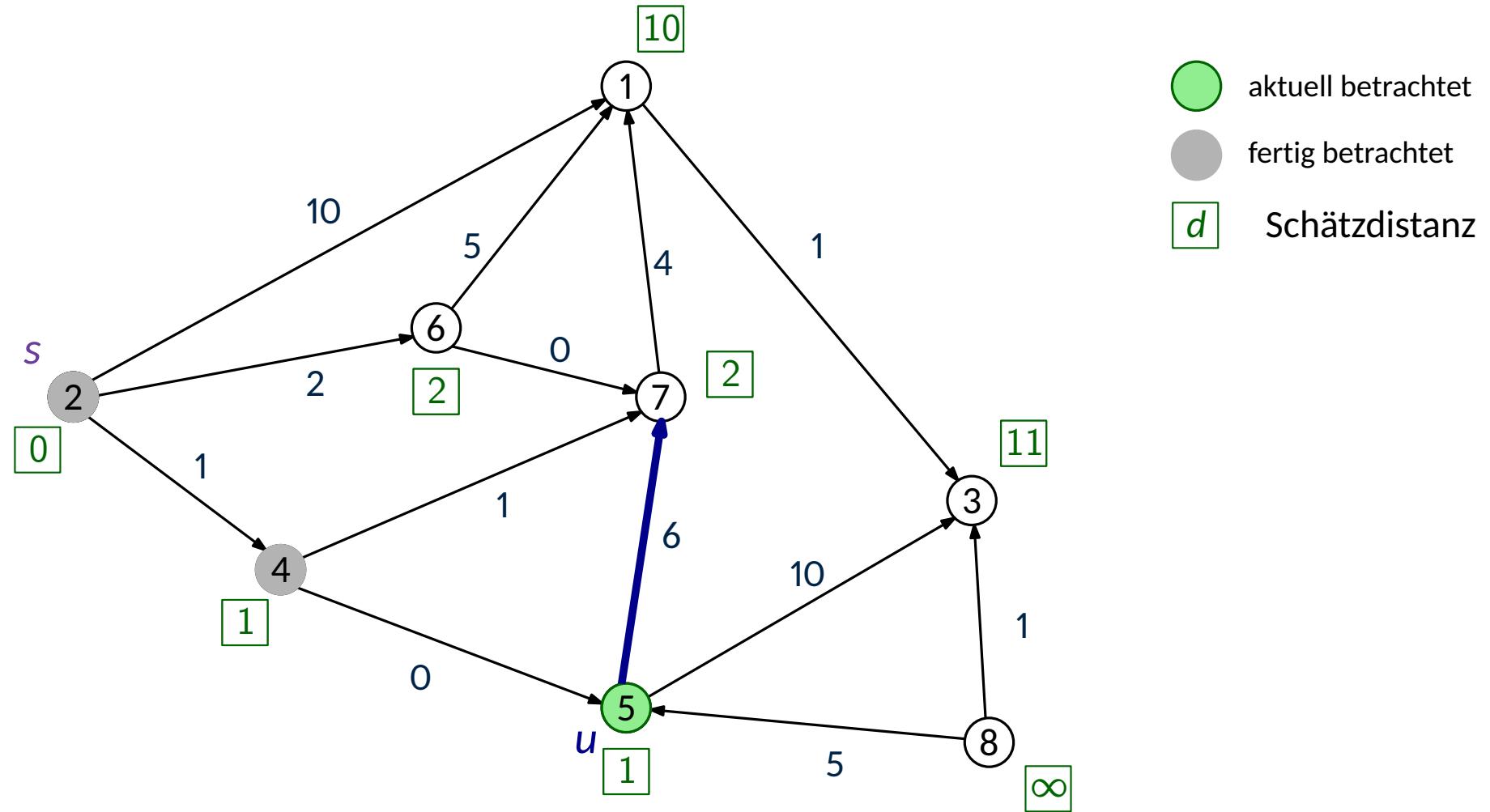
# Dijkstra-Algorithmus: Beispiel



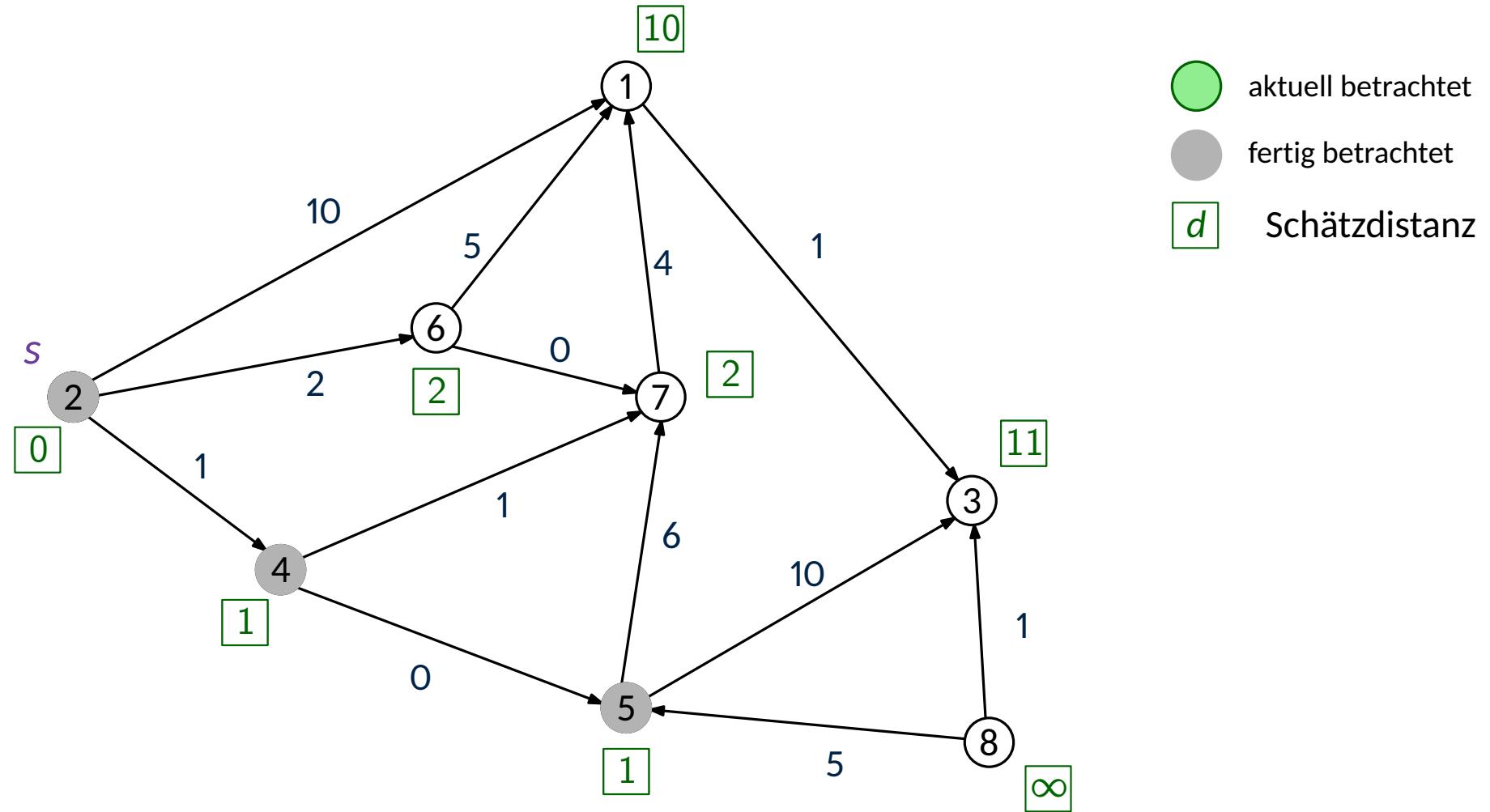
# Dijkstra-Algorithmus: Beispiel



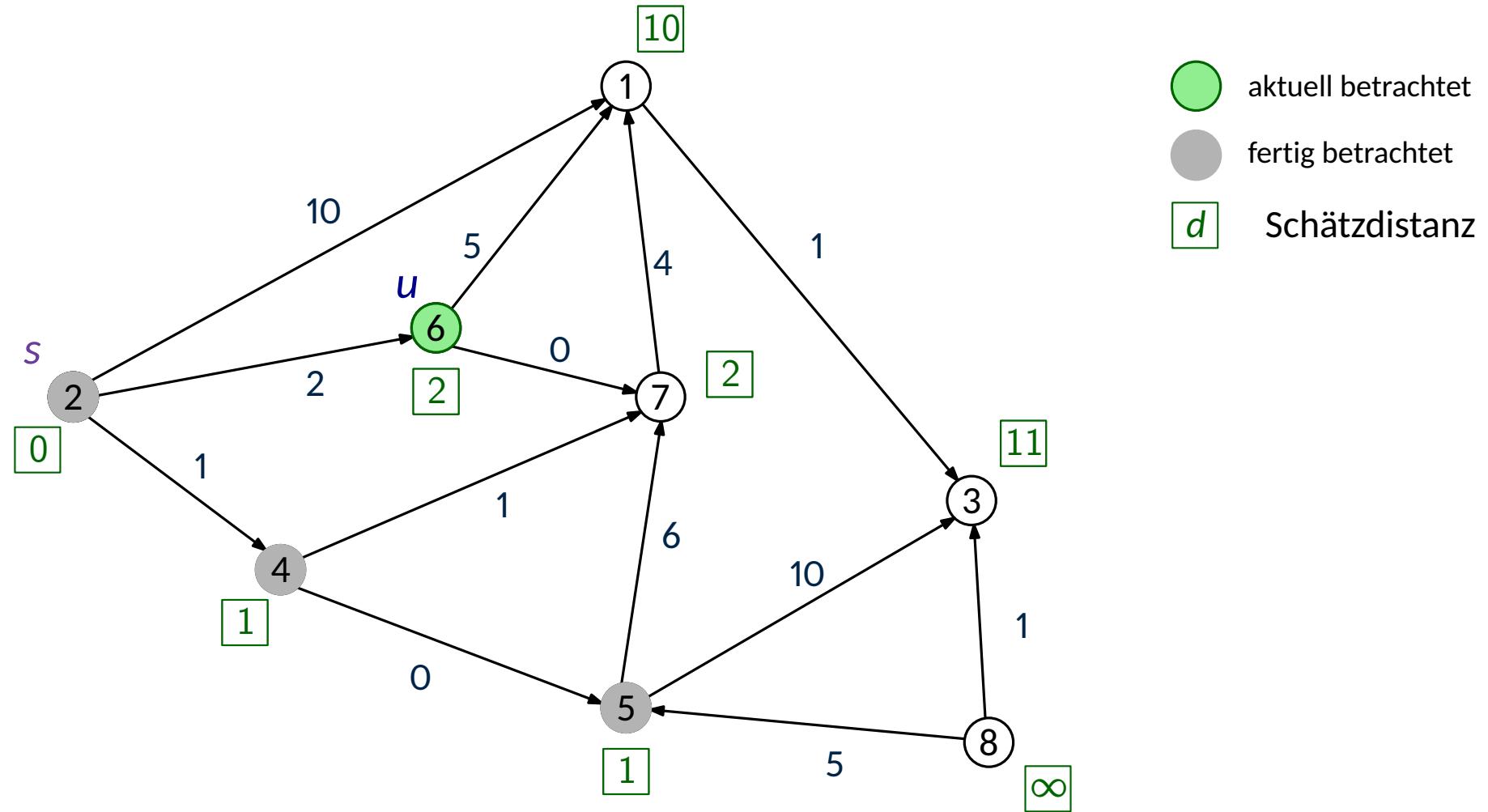
# Dijkstra-Algorithmus: Beispiel



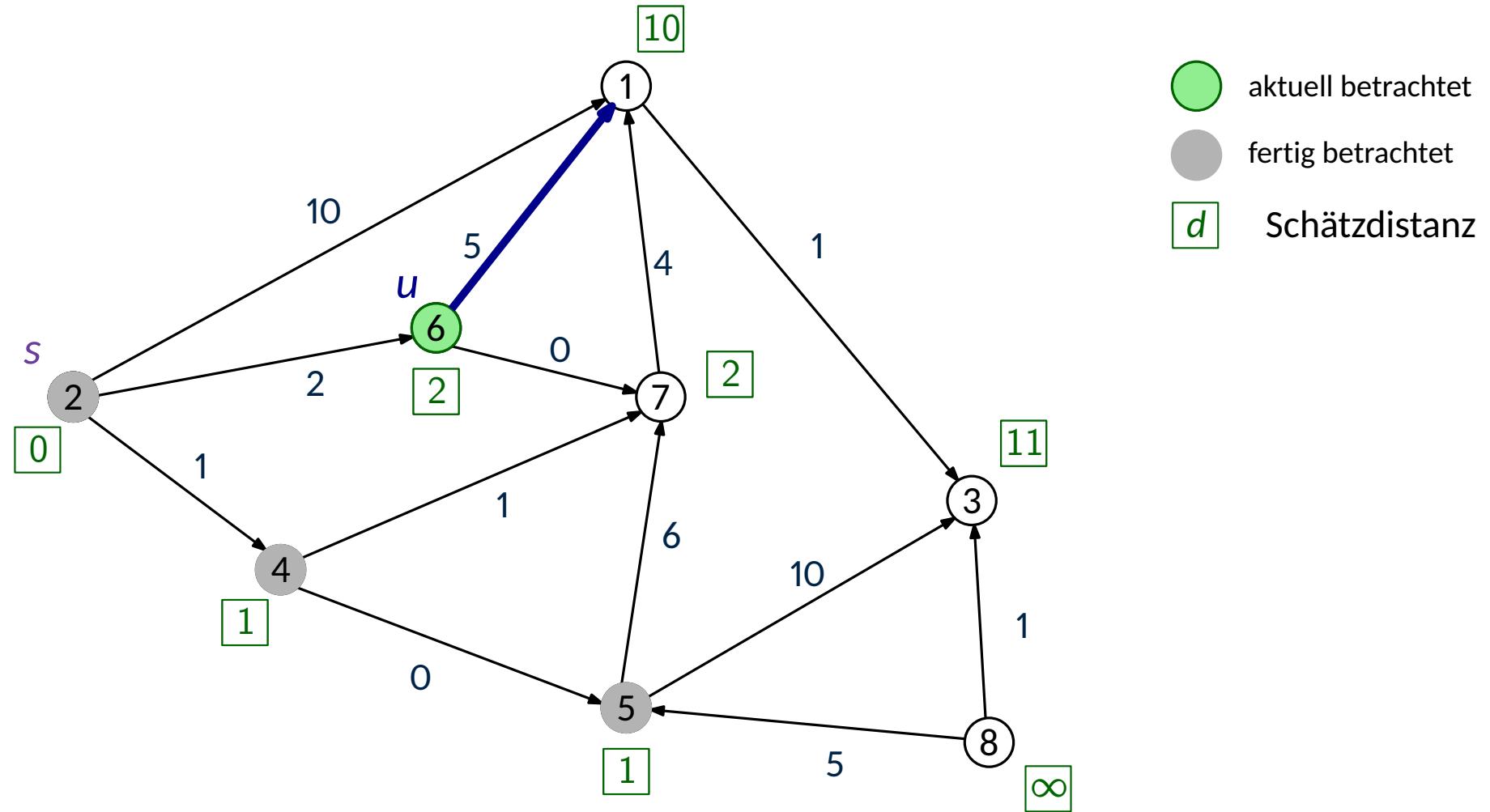
# Dijkstra-Algorithmus: Beispiel



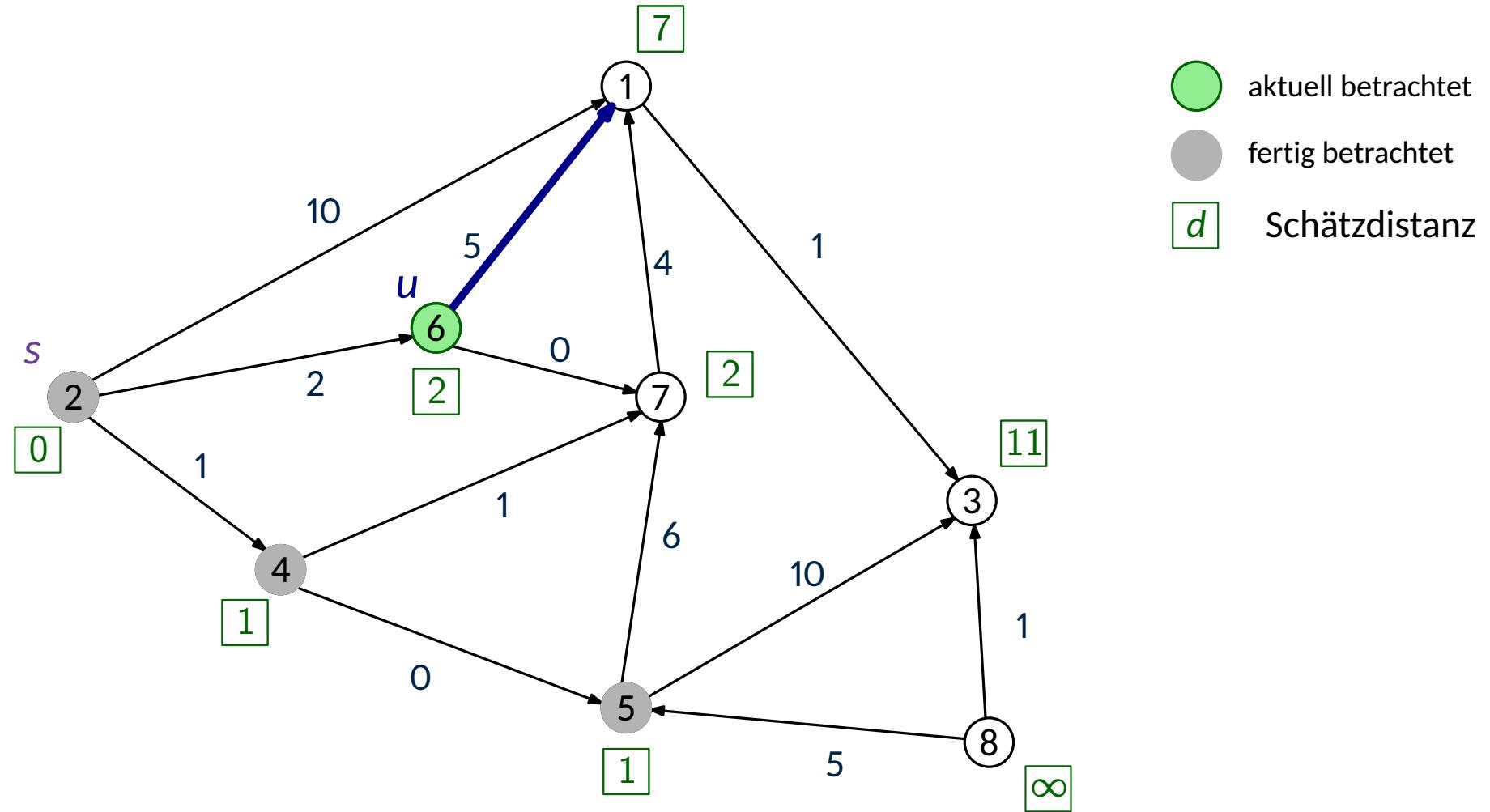
# Dijkstra-Algorithmus: Beispiel



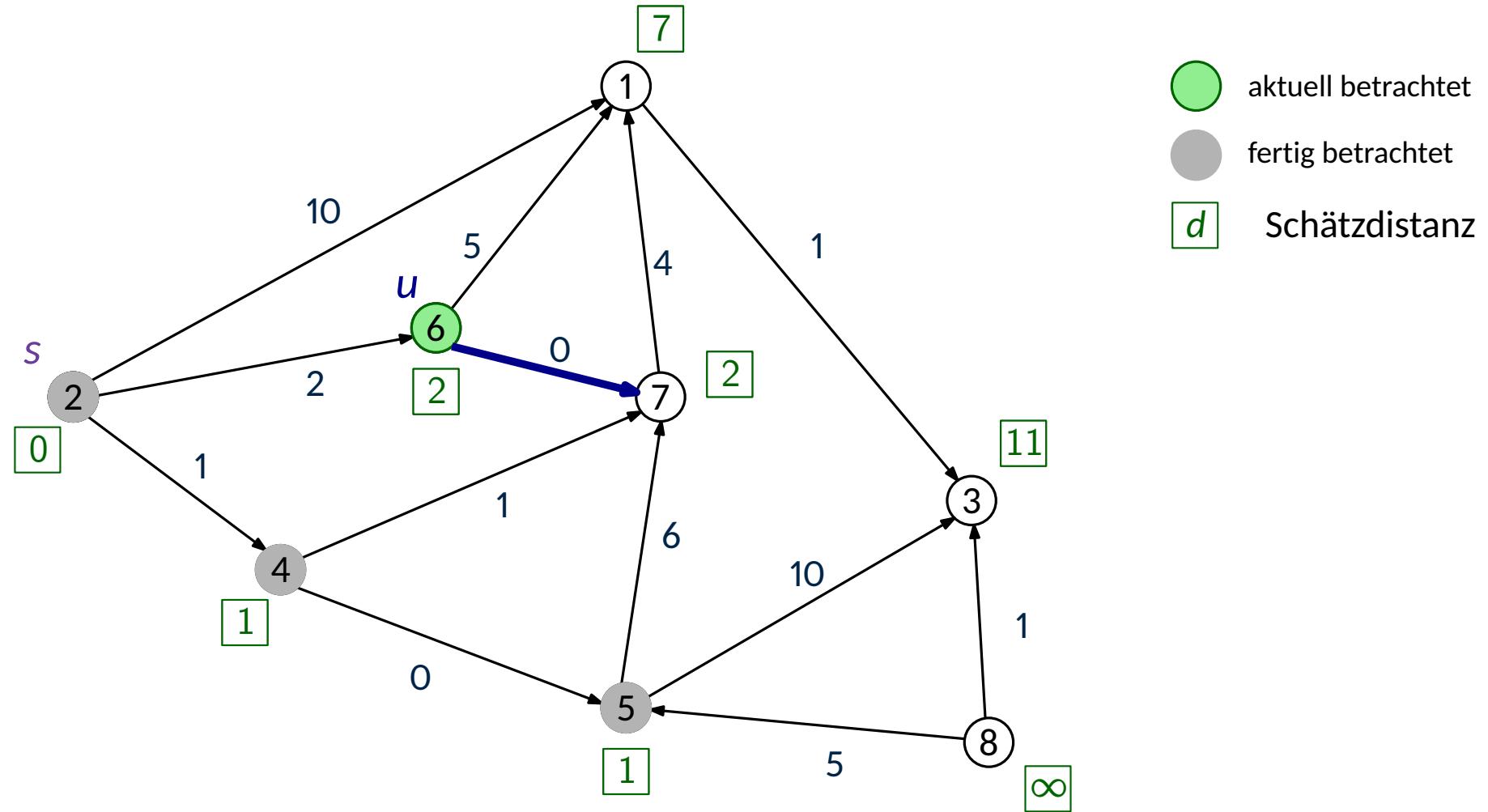
# Dijkstra-Algorithmus: Beispiel



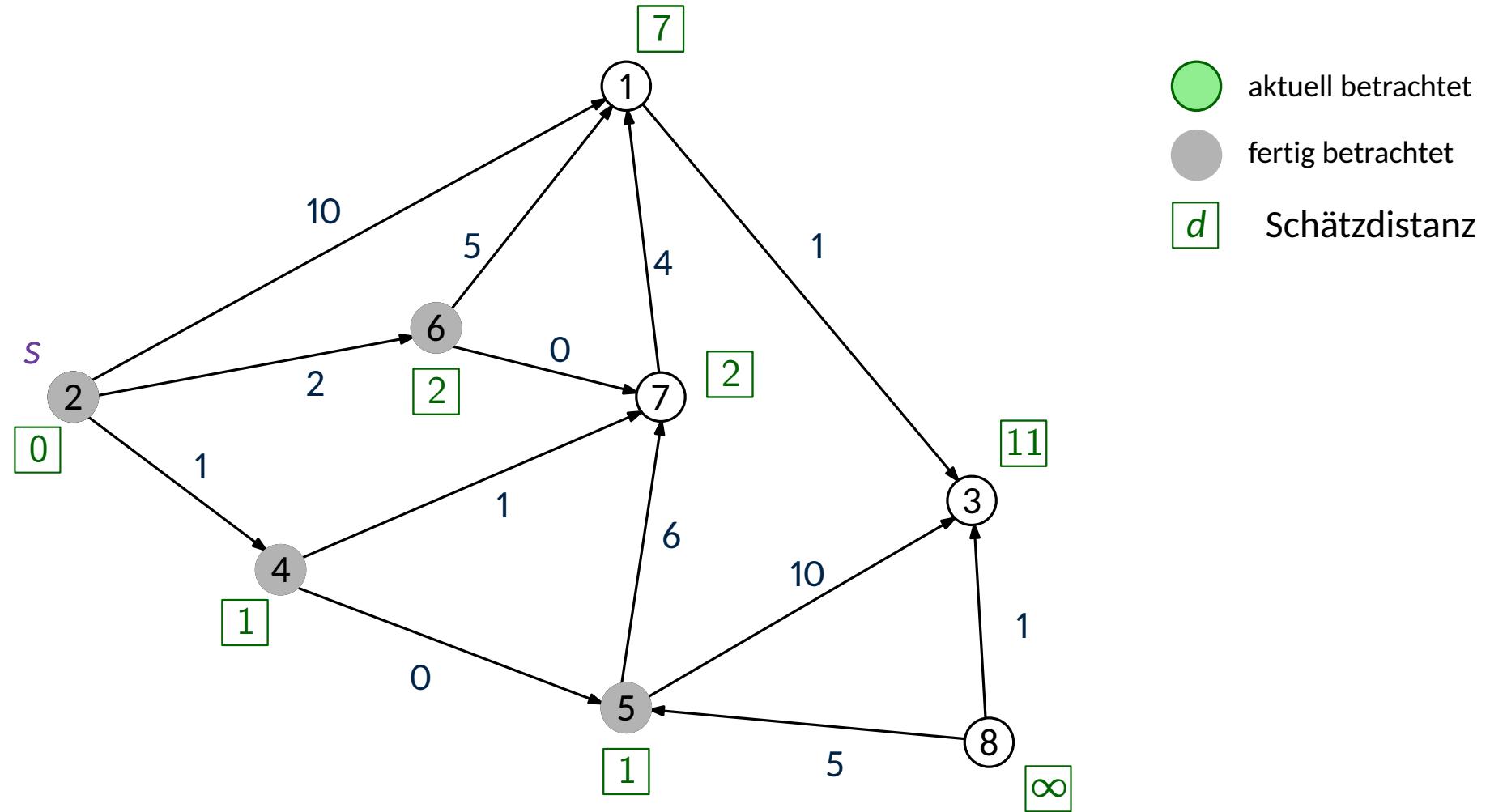
# Dijkstra-Algorithmus: Beispiel



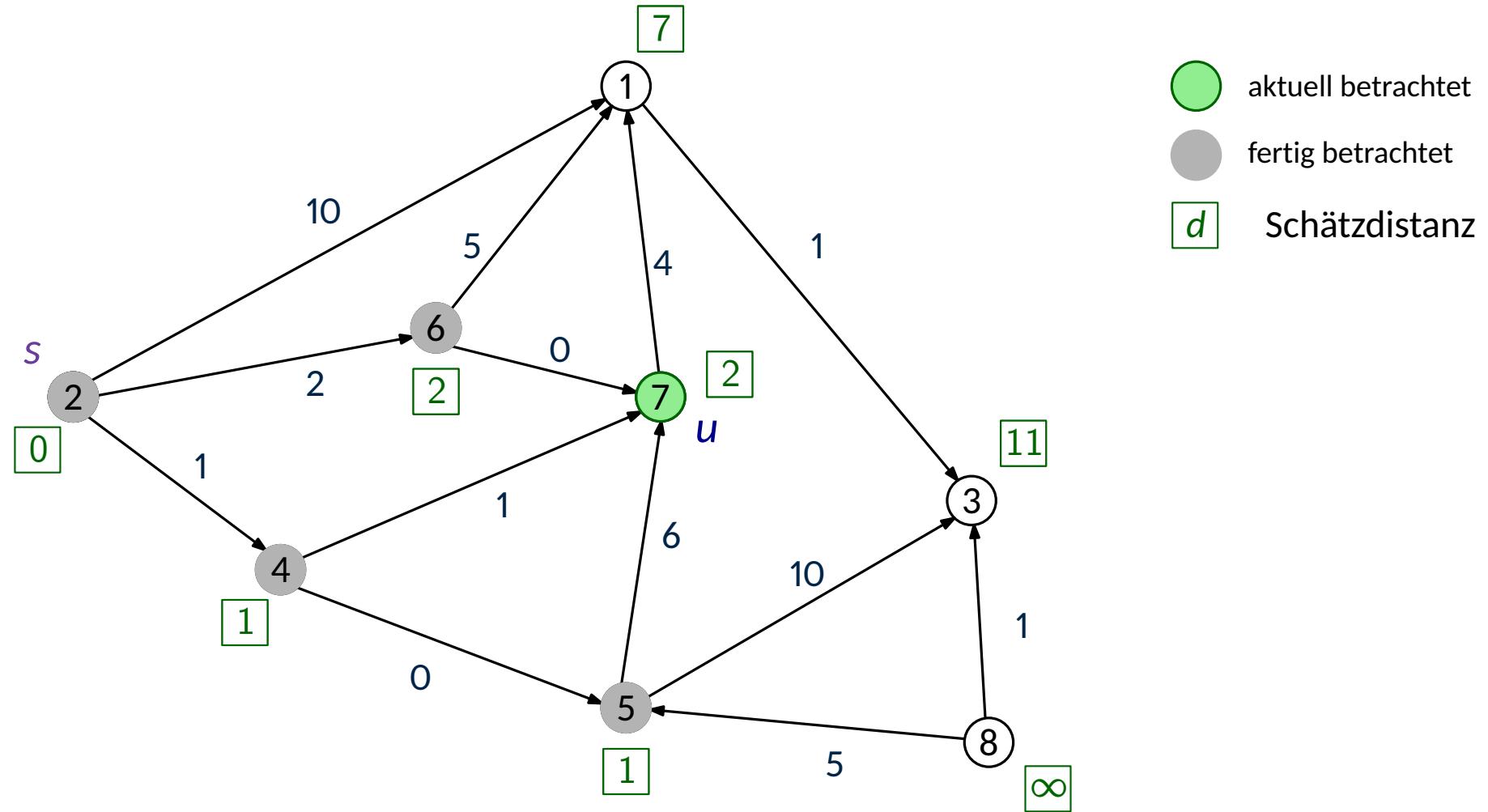
# Dijkstra-Algorithmus: Beispiel



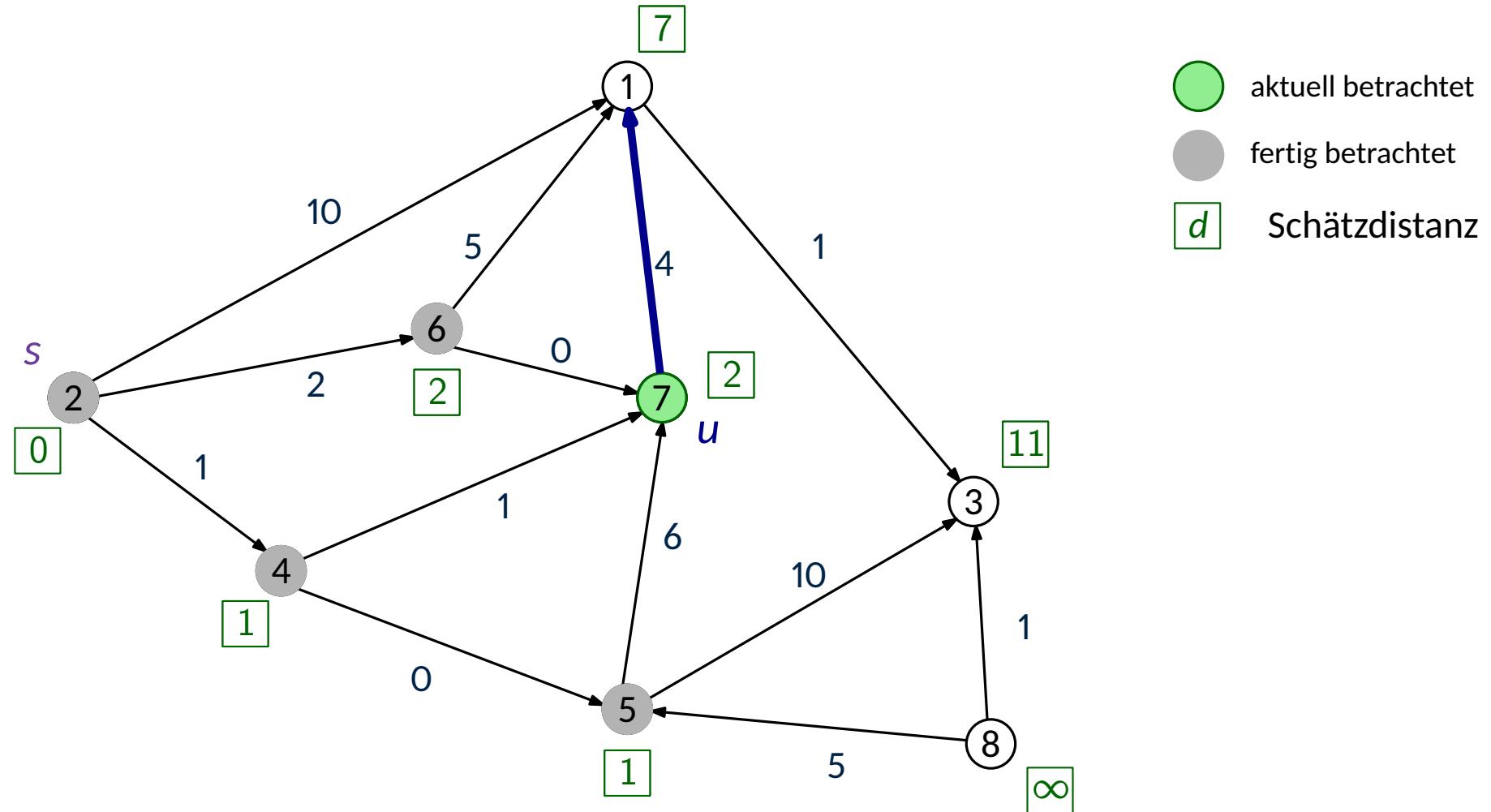
# Dijkstra-Algorithmus: Beispiel



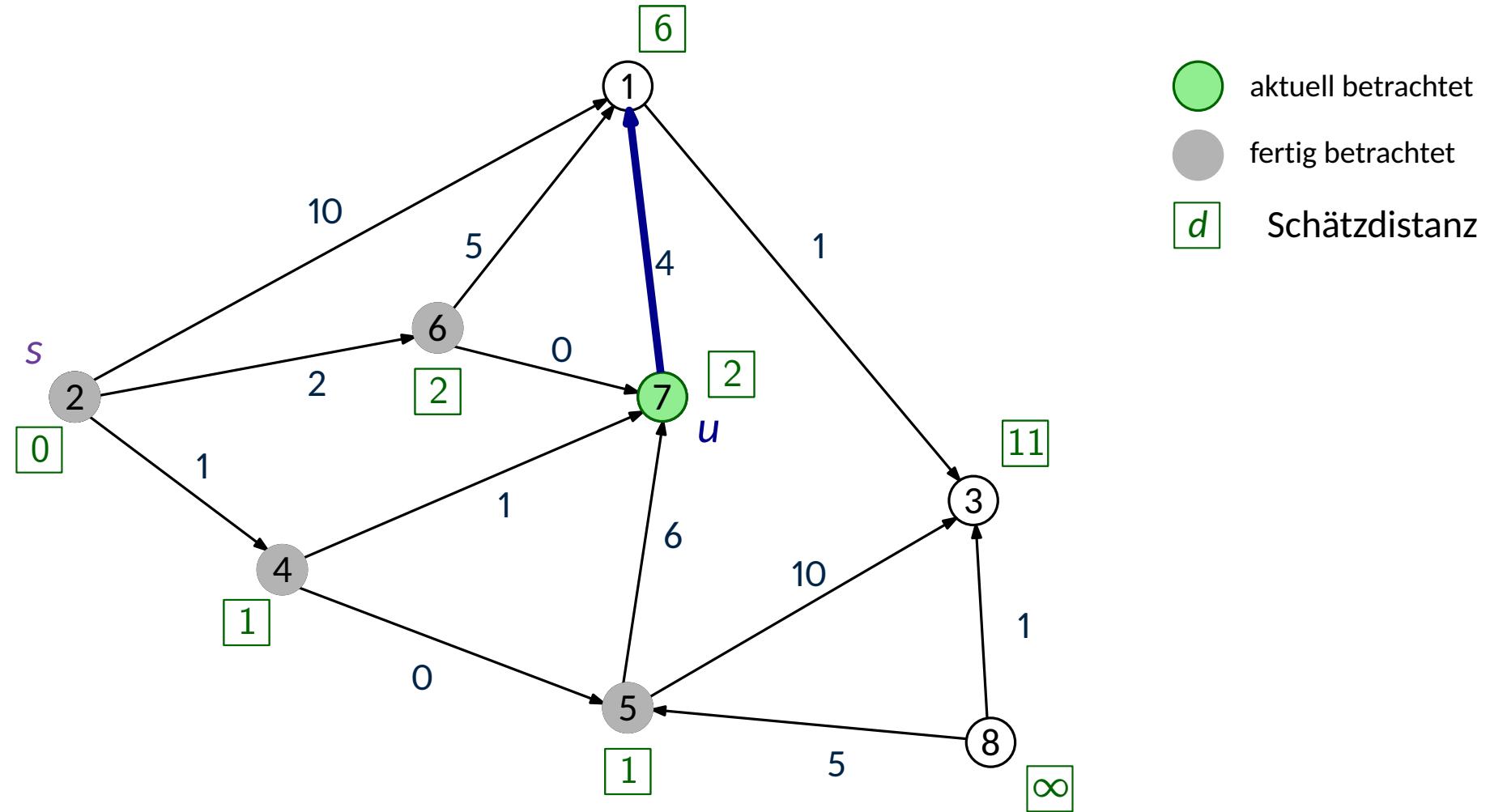
# Dijkstra-Algorithmus: Beispiel



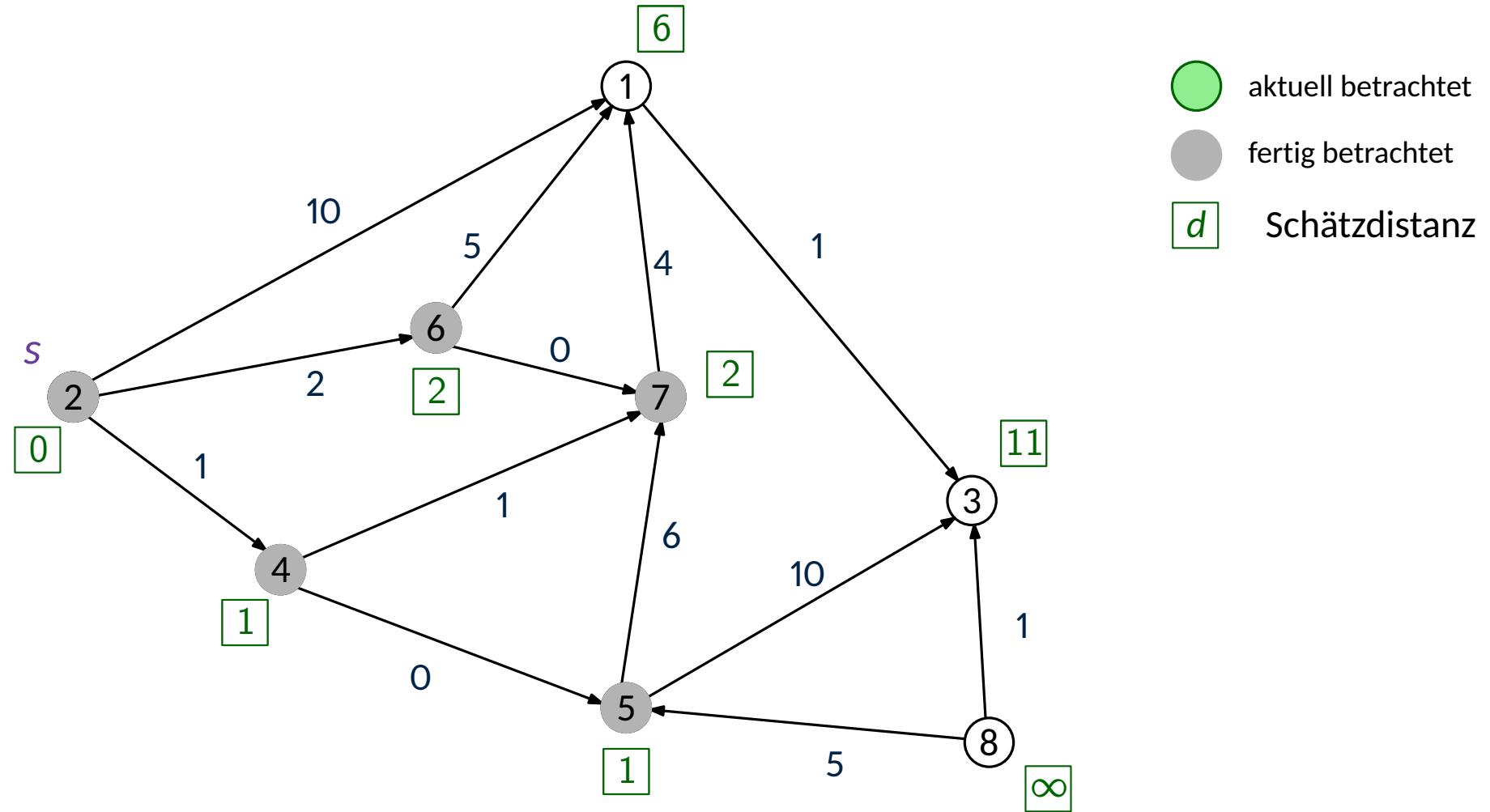
# Dijkstra-Algorithmus: Beispiel



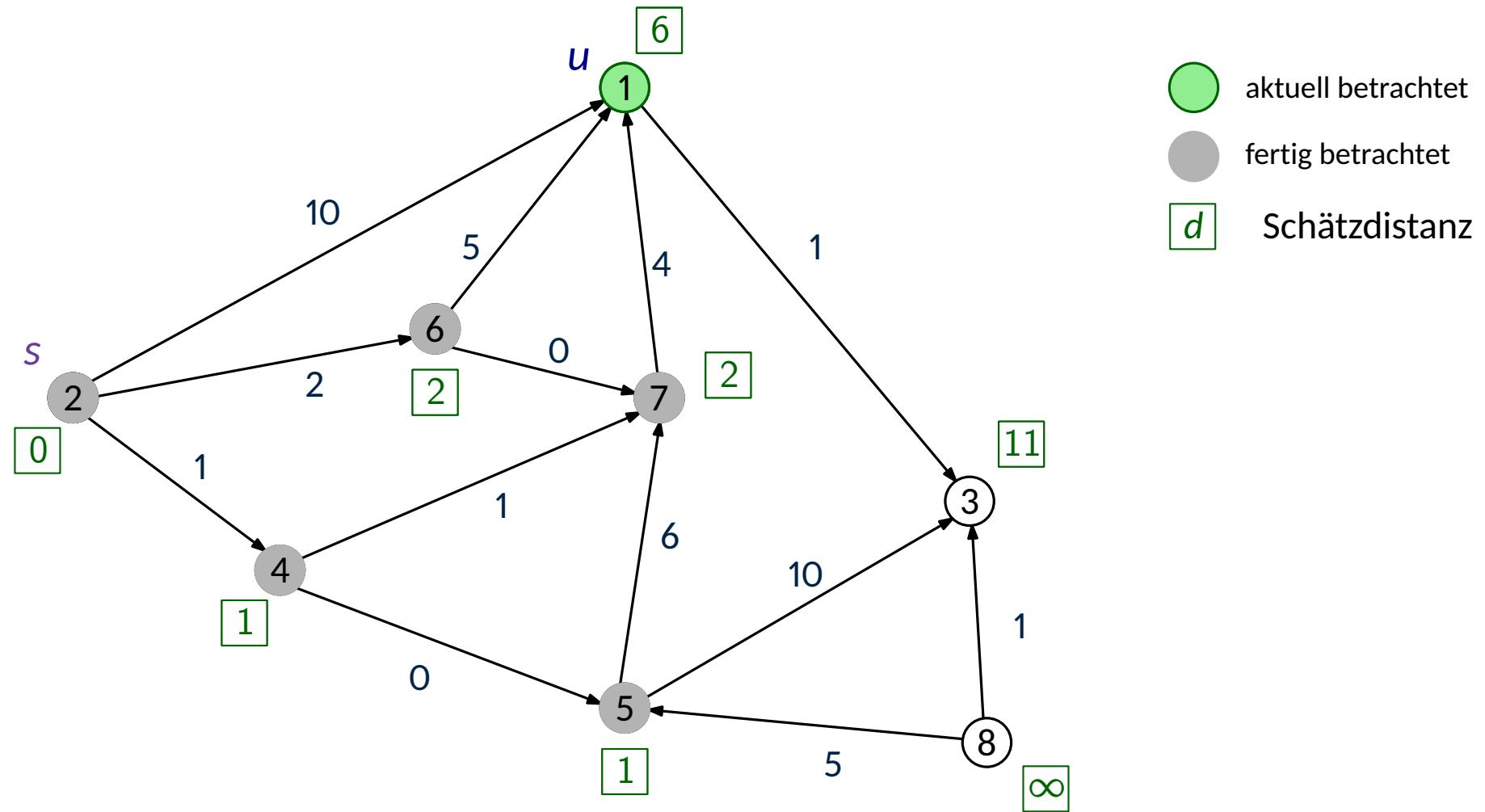
# Dijkstra-Algorithmus: Beispiel



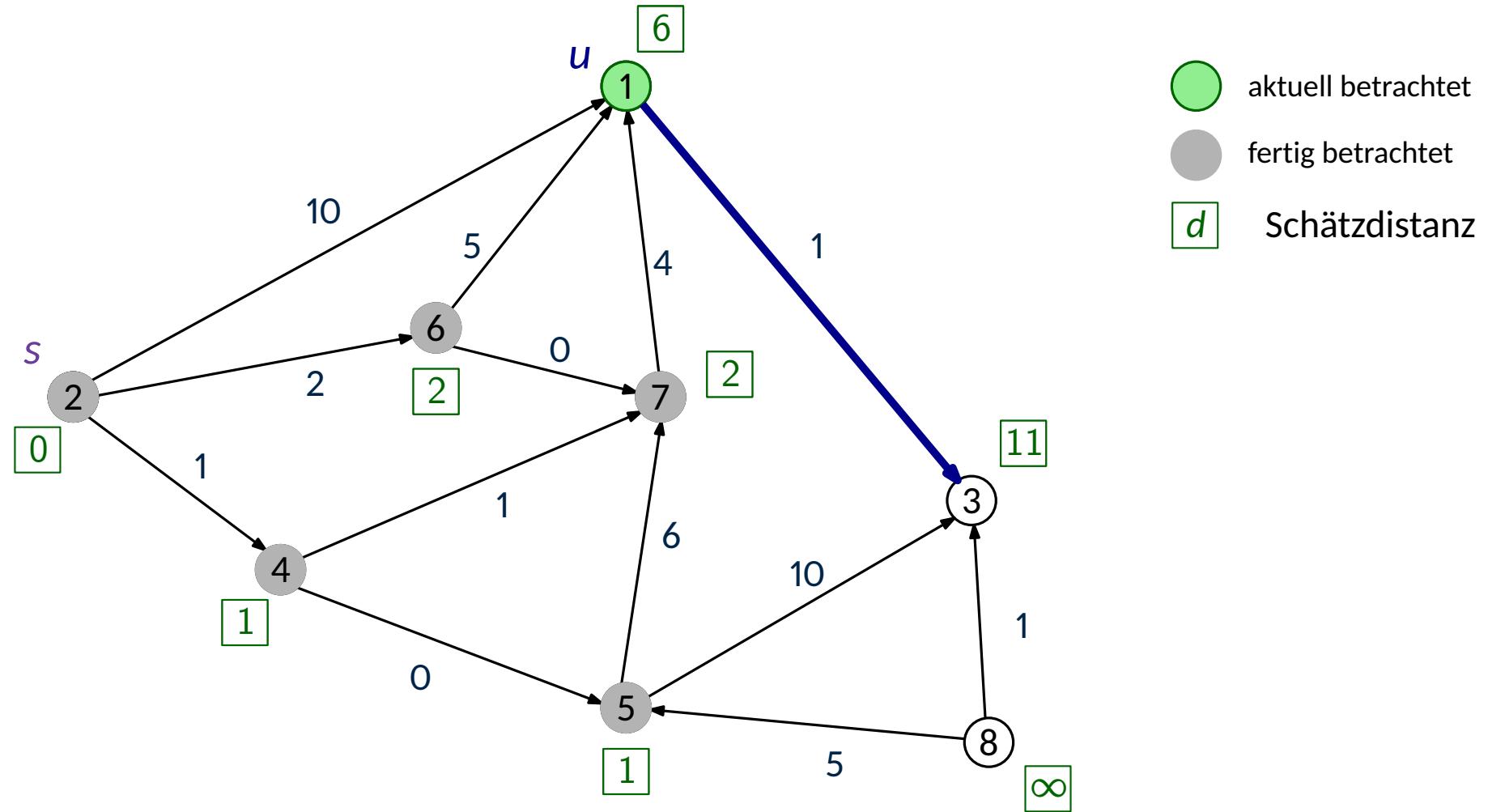
# Dijkstra-Algorithmus: Beispiel



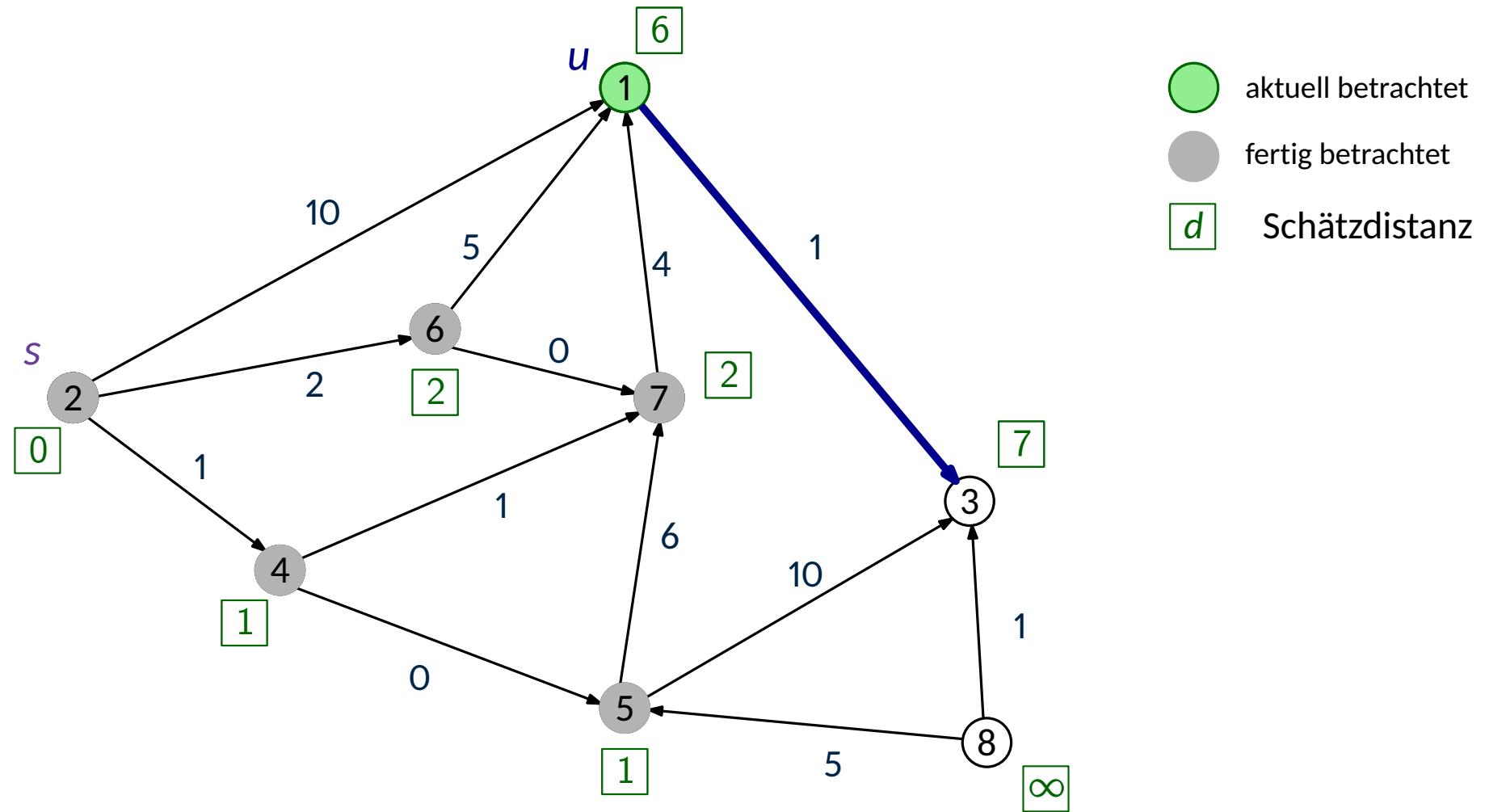
# Dijkstra-Algorithmus: Beispiel



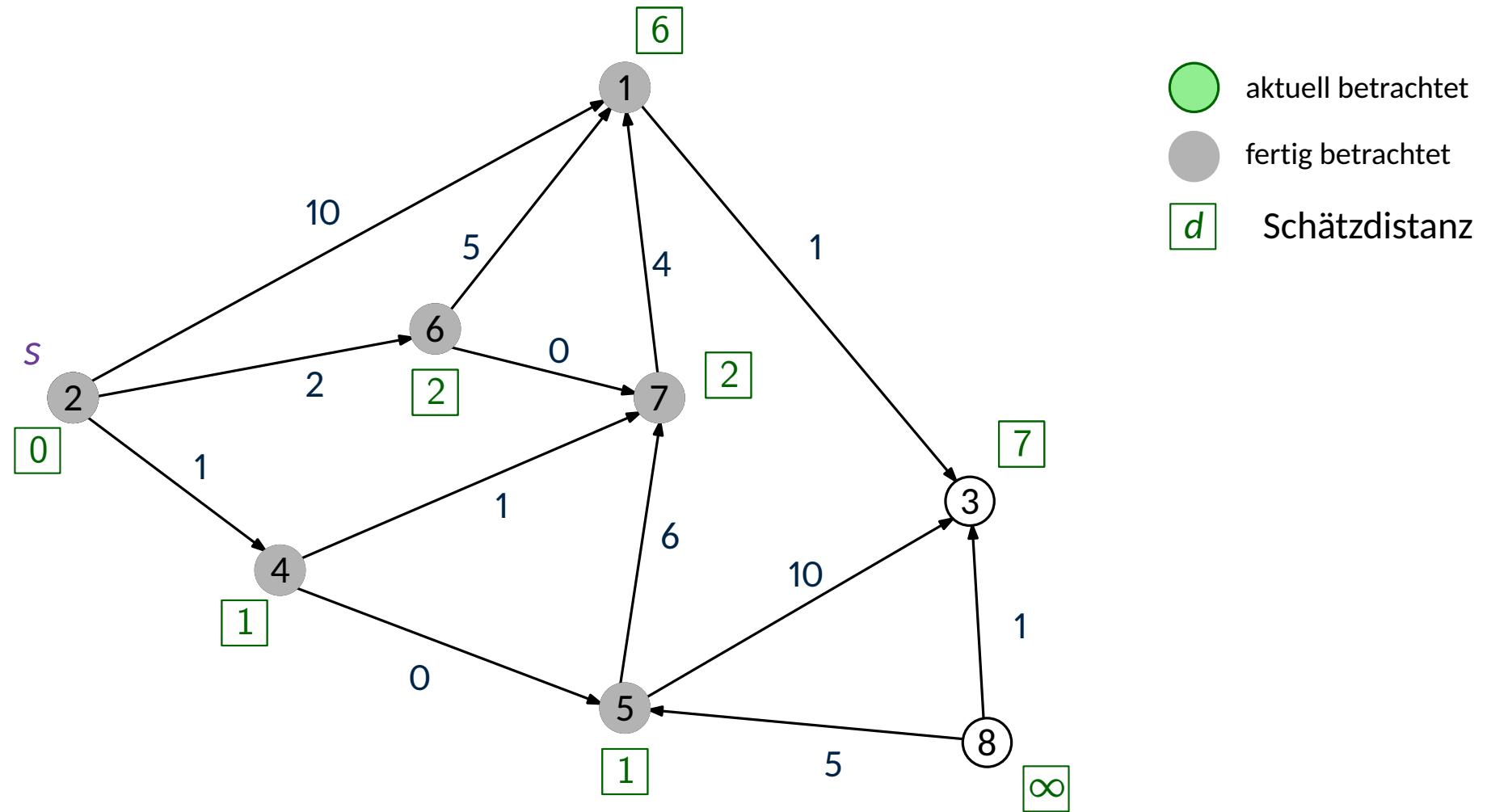
# Dijkstra-Algorithmus: Beispiel



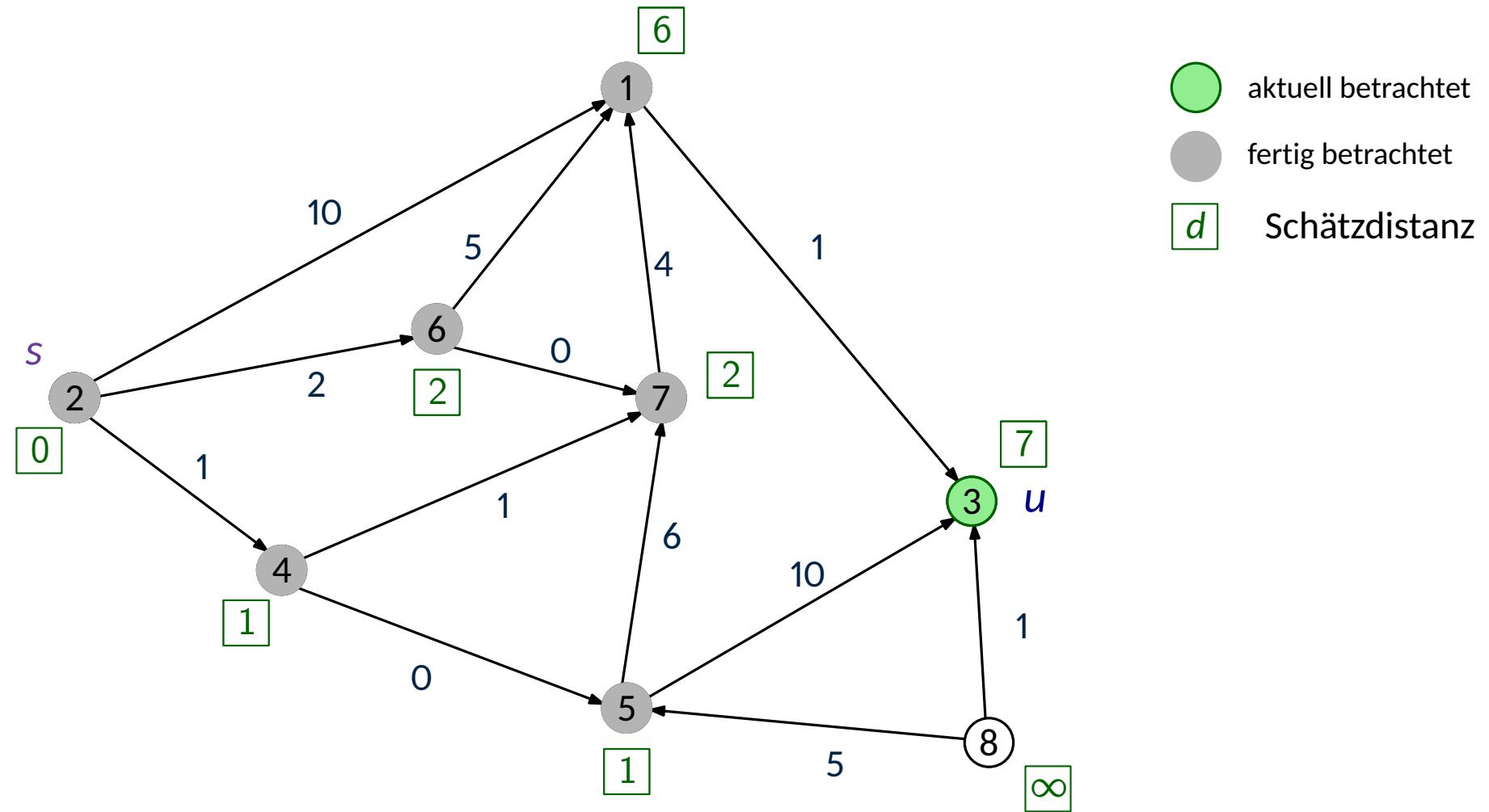
# Dijkstra-Algorithmus: Beispiel



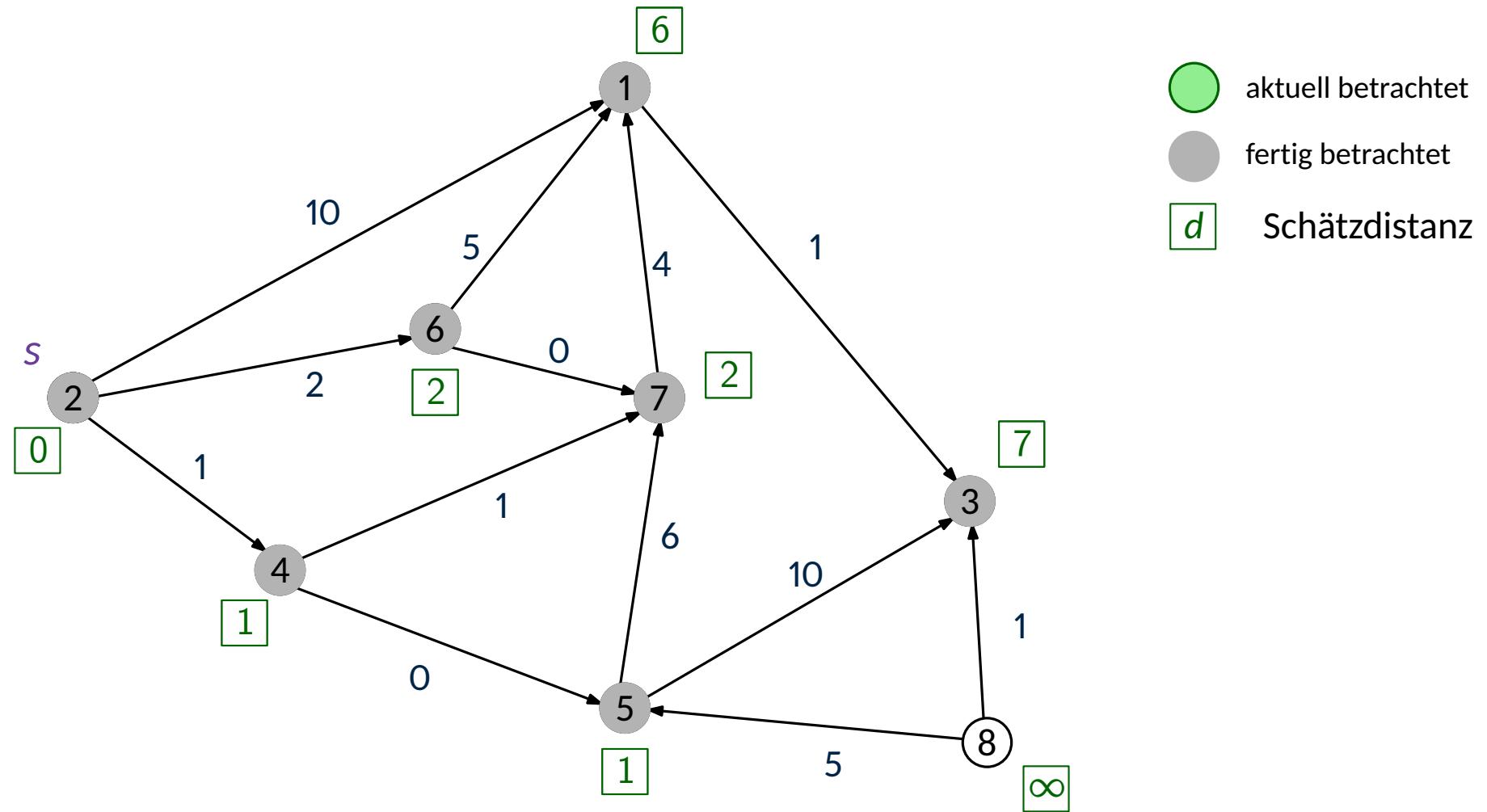
# Dijkstra-Algorithmus: Beispiel



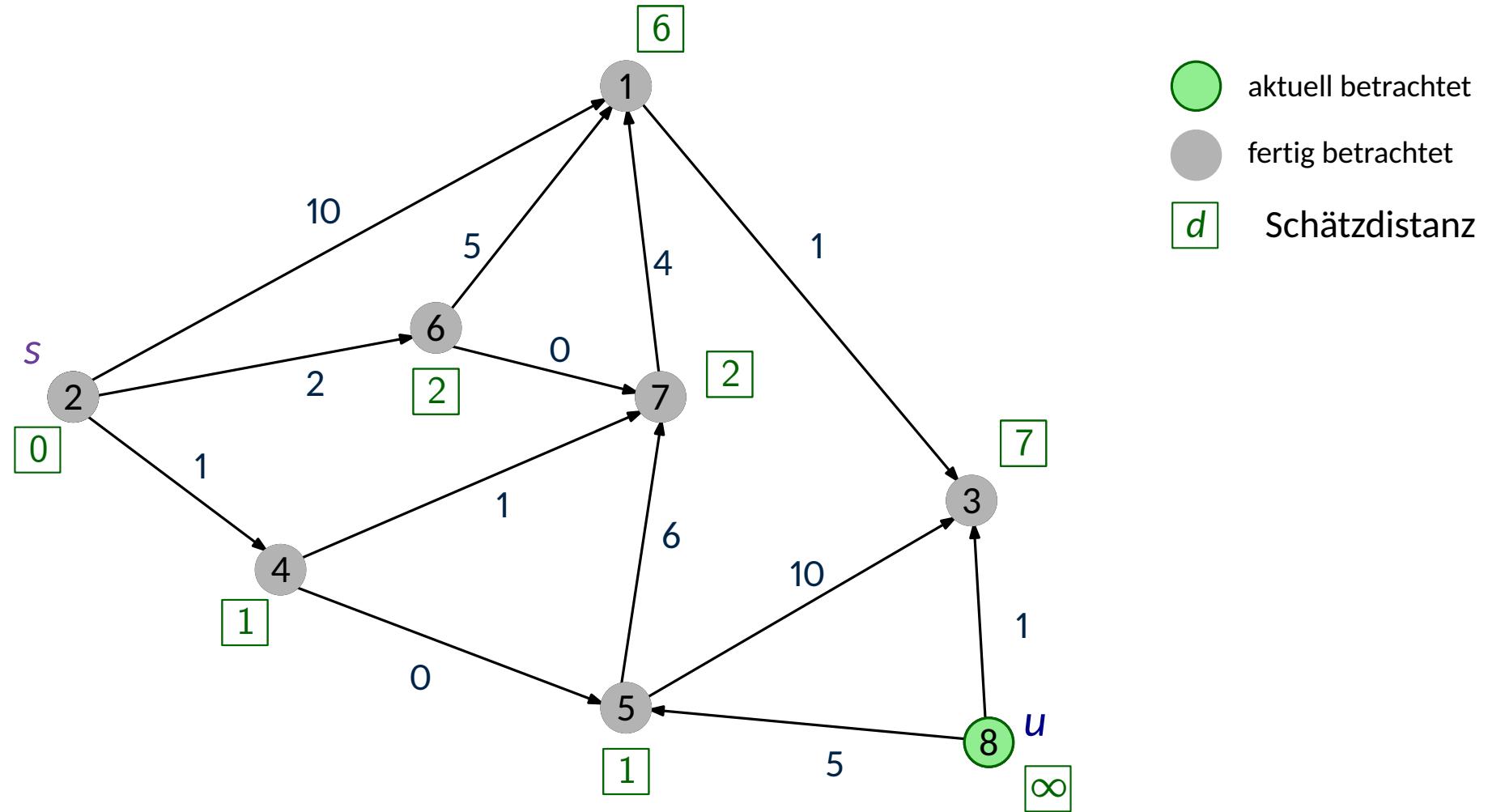
# Dijkstra-Algorithmus: Beispiel



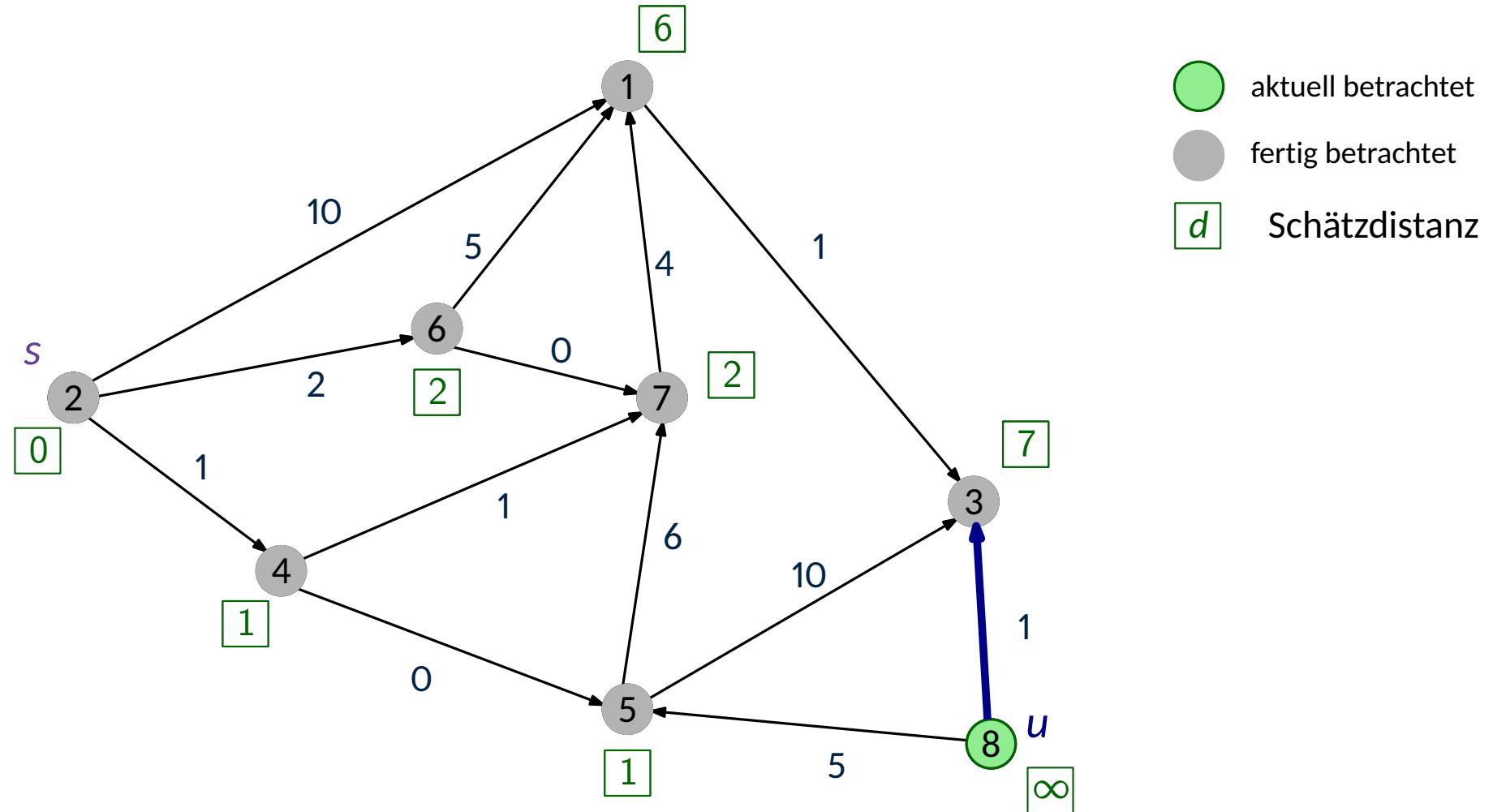
# Dijkstra-Algorithmus: Beispiel



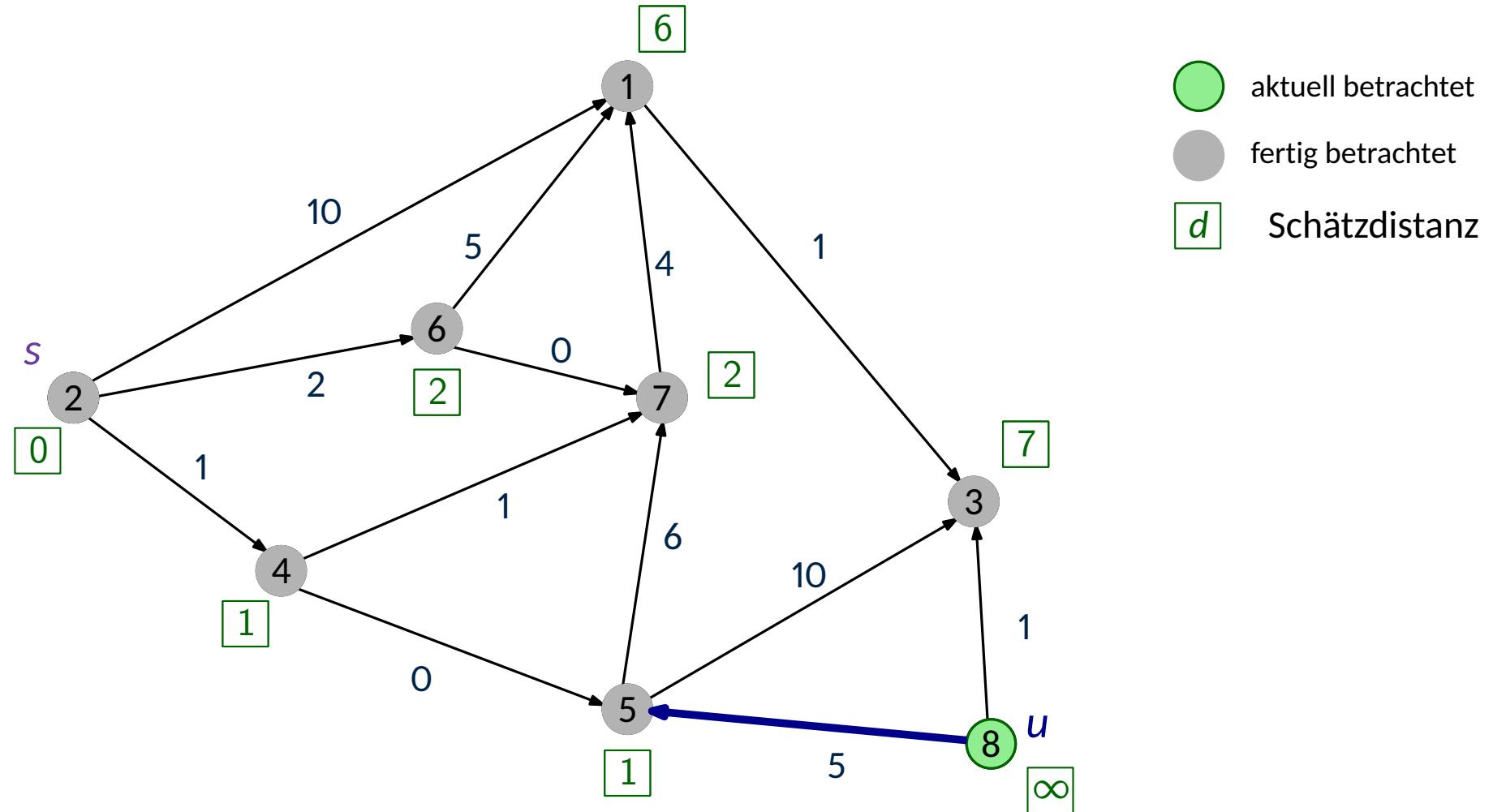
# Dijkstra-Algorithmus: Beispiel



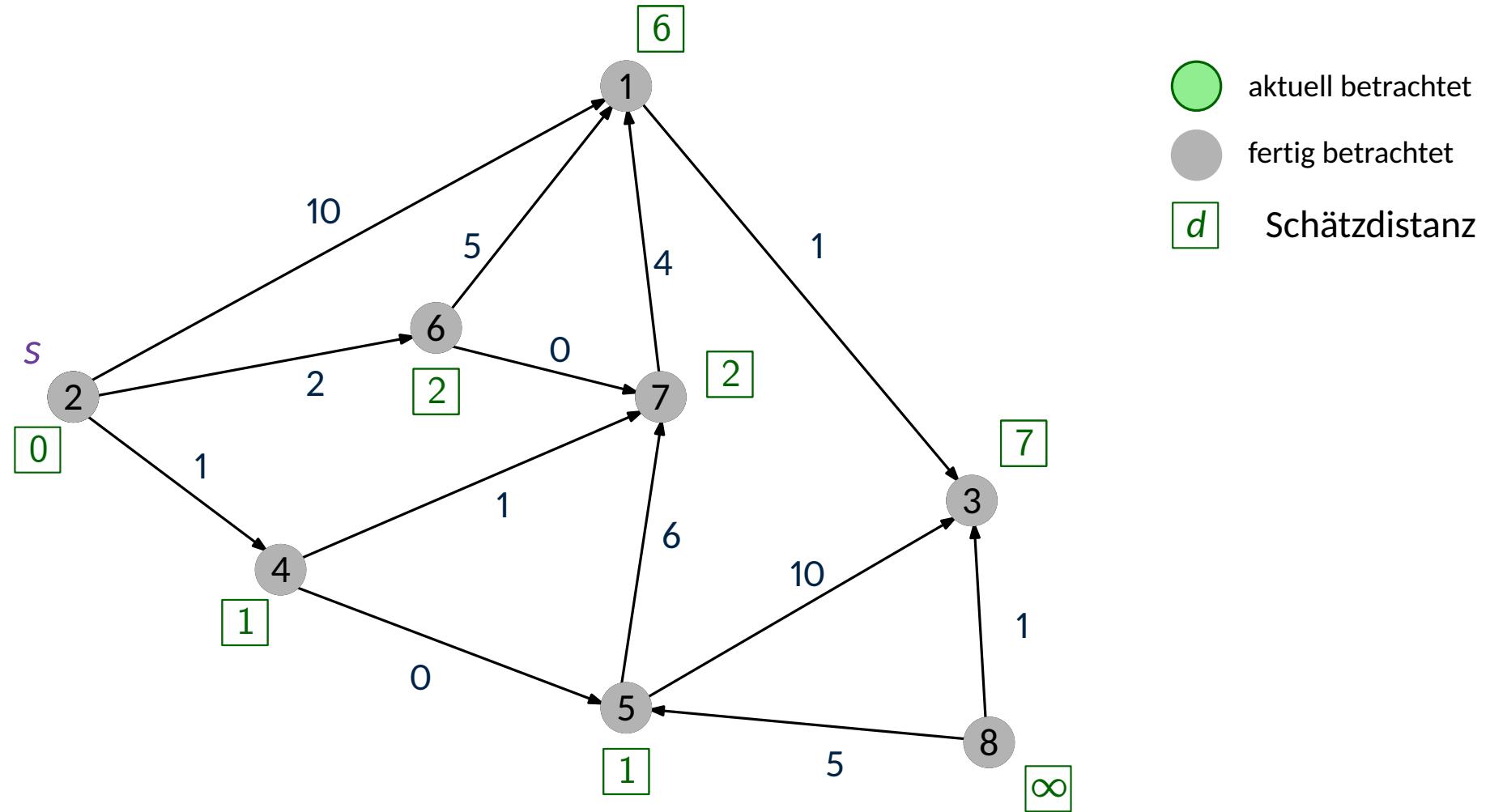
# Dijkstra-Algorithmus: Beispiel



# Dijkstra-Algorithmus: Beispiel



# Dijkstra-Algorithmus: Beispiel



# Korrektheit von Dijkstra's Algorithmus

**Lemma (Korrektheit von Dijkstra's Algorithmus).**

**Situation:**  $w(e) \geq 0$  für alle  $e \in E$ .

Nach Ausführung des Algorithmus gilt für alle  $v \in V$ :  $d[v] = \delta(s, v)$ .

**Beweis:**

Per Induktion. Wir zeigen: Sobald Knoten  $u$  betrachtet wird, gilt  $d[u] = \delta(s, u)$ .

**Anfang:**

Der erste Knoten  $u$ , der betrachtet wird, ist  $s$  und es gilt  $d[s] = 0 = \delta(s, s)$  nach Initialisierung.

**Schritt:**

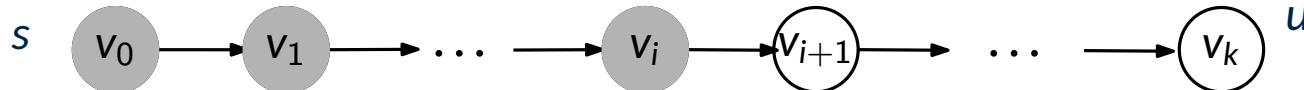
Angenommen, für alle bereits betrachteten Knoten  $u'$  gilt  $d[u'] = \delta(s, u')$ .

Sei  $u$  der nächste betrachtete Knoten.

Wir zeigen: Wenn  $d[u] \neq \delta(s, u)$  erhalten wir einen Widerspruch.

Da  $d[u]$  immer eine valide Schätzdistanz ( $\delta(s, u) \leq d[u]$ ) bleibt, muss gelten:  $\delta(s, u) < d[u]$ .

**Keine negative Kantengewichte**  $\Rightarrow$  es gibt einen kürzesten Weg  $v_0, \dots, v_k$  von  $v_0 = s$  nach  $v_k = u$



Sei  $0 \leq i < k$  maximal, so dass wir  $v_0, \dots, v_i$  vor  $u$  betrachtet haben. Es gilt:

·  $v_i$  wurde vor  $u$  betrachtet  $\rightarrow d[v_i] = \delta(s, v_i)$

· die Relaxierung von  $(v_i, v_{i+1}) \in E$  ergab also:

Optimalität von Subpfaden!

$$d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$$

Da  $\delta(s, v_{i+1}) \leq \delta(s, u)$ , gilt also  $d[v_{i+1}] \leq \delta(s, u) < d[u]$

Also:  $d[v_{i+1}] < d[u]$ . Widerspruch dazu, dass  $v_{i+1}$  noch nicht betrachtet wurde.  $\sharp$



# Genaue Umsetzung des Dijkstra-Algorithmus

---

Zentrale Frage:

Wir speichern wir die unbetrachteten Knoten, um den Knoten  $v$  mit kleinster Schätzdistanz  $d[v]$  bestimmen zu können?

Antwort: Mithilfe einer addressierbaren Min-Prioritätswarteschlange!

Dijkstra( $G, s$ ):

```
for  $v = 1, \dots, n$  do
    |  $d[v] = \infty$ 
    |  $p[v] = \perp$ 
 $d[s] = 0$ 
Initialisiere Min-Prioritätswarteschlange  $Q$  für  $v \in V$  mit Schlüssel  $d[v]$ 
while  $Q$  ist nicht leer do
    |  $u$  sei der Knoten aus  $Q$  mit minimaler Schätzdistanz  $d[u]$ ; entferne diesen aus  $Q$ 
    for all  $(u, v) \in E$  do
        | if  $(d[v] > d[u] + w(u, v))$  then //relaxiere Kante  $(u, v)$ 
        |     |  $d[v] = d[u] + w(u, v)$ 
        |     |  $p[v] = u$ 
        |     |  $Q.\text{decreaseKey}(v, d[v])$ 
```

Laufzeit: · je  $\leq n$  Operationen:  $Q.\text{min}()$ ,  $Q.\text{deleteMin}()$

·  $\leq m$  Operationen:  $Q.\text{decreaseKey}()$

18 - 15 ·  $O(n + m)$  für die restlichen Operationen (Vgl.  $\nearrow$  BFS)

# Laufzeit des Dijkstra-Algorithmus

Addressierbare Prioritätswarteschlange:	via <b>Binary Heaps</b>  frühere Übung	via <b>Fibonacci Heaps</b> nicht besprochen
Initialisierung für $n$ Elemente	$O(n)$	$O(n)$
min	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$ amortisiert
decreaseKey	$O(\log n)$	$O(1)$ amortisiert
Laufzeit von Dijkstra:	$\leq O(n + m) + n \text{ min/deleteMin's} + m \text{ decreaseKey's}$	
	$O((n + m) \log n)$	$O(n \log n + m)$

⇒ Mit ausgefeilter Implementierung von Heaps läuft Dijkstra in Zeit  $O(n \log n + m)$   
In der Praxis verwendet man häufig einfachere Implementierungen von Prioritätswarteschlangen.

# **SSSP für allgemeine Kantengewichte: Bellman-Ford-Algorithmus**

# Bellman-Ford

---

## Algorithmenbeschreibung:

- nach einer Initialisierung führen wir  $n - 1$  Phasen aus.
- in jeder Phase relaxieren wir **jede Kante**
  - wenn danach eine weitere Relaxierung erfolgreich ist, berichten wir, dass ein negativer Zyklus existiert.
  - ansonsten sehen wir unsere Schätzdistanzen als tatsächliche Distanzen an

`BellmanFord( $G, s$ ):`

```
for  $v = 1, \dots, n$  do
    |  $d[v] = \infty$ 
    |  $p[v] = \perp$ 
 $d[s] = 0$ 

for  $t = 1, \dots, n - 1$  do // $n - 1$  Phasen
    for all  $(u, v) \in E$  do //relaxiere Kante  $(u, v)$ 
        if  $(d[v] > d[u] + w(u, v))$  then
            |  $d[v] = d[u] + w(u, v)$ 
            |  $p[v] = u$ 

for all  $(u, v) \in E$  do
    if  $(d[v] > d[u] + w(u, v))$  then
        | error: "es existiert ein negativer Zyklus"
```

# Bellman-Ford: Analyse

## Theorem (Bellman-Ford).

Sei  $G = (V, E, w)$  ein gewichteter Graph und  $s \in V$ .

Der Bellman-Ford-Algorithmus läuft in Zeit  $O(nm)$  und bestimmt:

- $d[v] = \delta(s, v)$  für alle  $v \in V$  oder
- die Existenz eines negativen Zyklus, der von  $s$  erreichbar ist.

## Zwei mögliche Fälle:

1. es gibt keinen negativen Zyklus  $\vec{z} = (z_0, \dots, z_{\ell-1})$  mit  $z = z_0 = z_\ell$  sodass  $s \rightsquigarrow z$ :

Dann existiert für jedes  $v \in V$  mit  $s \rightsquigarrow v$  ein kürzester Weg  $\vec{p} = (p_0, \dots, p_k)$  von  $p_0 = s$  nach  $p_k = v$ .

Es gilt:  $\vec{p}$  ist ein **Pfad**!

$\Rightarrow \vec{p}$  hat Länge  $k \leq n - 1$ .

Nach der  $i$ .ten Phase gilt:  $d[p_i] = \delta(s, p_i)$

Am Ende des Algorithmus gilt  $d[v] = \delta(s, v)$  und keine weitere Relaxierung verbessert Schätzdistanz.

2. es existiert ein negativer Zyklus  $\vec{z} = (z_0, \dots, z_\ell)$  mit  $z_0 = z_\ell = z$  sodass  $s \rightsquigarrow z$ :

**Behauptung:** Es gibt ein  $0 \leq i < \ell$  sodass eine Relaxierung der Kante  $(z_i, z_{i+1})$  erfolgreich ist.

**Beweis:** eigene Übung

# Ausblick

Kürzeste Wege in Graphen sind Gegenstand intensiver Forschung:

## Algorithm Engineering

→ viele weitere Tricks und Kniffe führen zu hocheffizienten praktischen Implementierungen  
siehe z.B. Dietzfelbinger et al. für einen Überblick

## Jüngste Entwicklungen:

Im März 2022 wurden folgende Algorithmen für SSSP mit negativen Gewichten bekanntgegeben:  
 $m^{1+o(1)}$  Algorithmus [Chen, Kyng, Liu, Peng, Probst Gutenberg, Sachdeva 2022]  
 $O(m \log^9(n))$  Algorithmus [Bernstein, Nanongkai, Wulff-Nielsen 2022]

## Zentrales Problem: All-Pairs Shortest Paths (APSP)

**Gegeben:** Graph  $G = (V, E, w)$

**Gesucht:** alle Distanzen, d.h.

$\delta(u, v)$  für alle  $u, v \in V$

## Offene Forschungsfrage:

Kann APSP in Zeit  $O(n^{2.999})$  gelöst werden?

→ lösbar über  $n$  SSSP-Anfragen

→ lösbar mittels **Floyd-Warshall-Algorithmus**

```
Initialisiere  $d[i, j] = w(i, j)$  für alle  $i, j$ 
for  $k = 1, \dots, n$  do
  for  $i = 1, \dots, n$  do
    for  $j = 1, \dots, n$  do
      |  $d[i, j] = \min(d[i, j], d[i, k] + d[k, j])$ 
```

$O(n^3)$  Laufzeit

# Zusammenfassung

---

Bestimmen aller Kürzeste-Wege-Distanzen von einem Knoten  $s$  (SSSP):

- ungewichtete Graphen: BFS  $O(n + m)$
- nichtnegativ gewichtete Graphen: Dijkstra
  - via Binary Heaps  $O(n \log n + m \log n)$
  - via Fibonacci Heaps\*  $O(n \log n + m)$
- allgemein gewichtete Graphen: Bellman-Ford  $O(n \cdot m)$

Algorithmen und Datenstrukturen SS'23

# Kapitel 14: Grundbegriffe der Komplexitätstheorie

Marvin Künemann

AG Algorithmen & Komplexität

# VLU-Kommentare

---

Vielen Dank für bereits eingetroffene Kommentare!

Reaktion zu zwei ausgewählten Kommentaren:

1. "Die Übungsblätter sind extrem zeitintensiv, vor allem wenn man alle Aufgaben bearbeiten möchte. Manche Aufgaben waren leider auch kaum anspruchsvoll, aber dauerten trotzdem ewig zum Bearbeiten. Bei Aufgabe 9.5 beispielsweise mussten zuerst die grundlegenden Datenstrukturen komplett implementiert werden, damit man sie auch erweitern konnte. Die eigentliche Aufgabe hat bei uns maximal 10% der benötigten Zeit beansprucht und der Rest war nur stupides Implementieren."

**Wichtiger Hinweis: Sie dürfen immer Inhalte der Vorlesung benutzen.**

Im konkreten Fall reicht es, zu beschreiben, welche Operationen der Datenstruktur wie angepasst werden müssen

→ Beispiel Einfügen im Rot-Schwarz-Baum:

1. Nach dem Einfügen muss  $AS(t)$  von jedem Vorfahren  $t$  des neu eingefügten Knoten um eins erhöht werden  
→ beschreibbar durch kurzen Pseudocode
2. Ansonsten ändert sich die Baumstruktur nur durch Links- oder Rechtsrotation  
→ beschreibbar durch kurzen Zusatzpseudocode für Links- bzw. Rechtsrotation

(Selbstverständlich gilt auch hier: Korrektheit und Laufzeit der Anpassung nicht vergessen!)

Allerdings ist es auch eine gute Übung, die komplette Datenstruktur durchdacht und beschrieben zu haben

2. "Es gibt oft Übungen in denen die in der Vorlesung behandelten Algorithmen verbessert werden sollen. Dort fände ich es besser neue Algorithmen zu schreiben, statt bereits behandelte zu verbessern"

Diesen Kommentar verstehe ich leider nicht ganz.

Manche Übungen zielen auf das Verständnis eines vorgestellten Algorithmus/Datenstruktur ab  
wir üben sowohl Verständnis als auch Anwendbarkeit der vorgestellten Ideen

# Kapitelüberblick

---

Bisher: viele effiziente Algorithmen und Datenstrukturen

Jetzt: Gibt es **Grenzen** für effiziente Algorithmen und Datenstrukturen?

Insbesondere besprechen wir in diesem Kapitel:

- das Erfüllbarkeitsproblem
- die Klassen P und NP

# Thematische Schwerpunkte

---

Eingabe →  → Ausgabe

## Grundbegriffe und Handwerkszeug

- Was ist ein Algorithmus und wie analysiere ich ihn? ✓
- Korrektheit und asymptotische Laufzeitschranken ✓
- formale Methoden und Beweistechniken ✓

## Algorithmen und Datenstrukturen

- Elementare Datenstrukturen ✓
- Suchen, Sortieren, Wörterbücher ✓
- Spezielle Strukturen: Listen, Graphen ✓

## Entwurfsmethoden

- Komplexe Algorithmen aus einfachen Algorithmen aufbauen
- Standardtechniken für bestimmte Problemklassen

→ verschoben ans Ende der VL

vorgezogen

## Theorie

- Komplexitätstheorie: Grenzen der effizienten Berechenbarkeit

# Exkurs: Aussagenlogik

---

$$\begin{aligned}\phi(x_1, x_2, x_3) &= (x_1 \text{ ODER } x_2) \text{ UND } (x_2 \text{ ODER } x_3) \text{ UND } (\text{NICHT } x_2) \\ &= (x_1 \vee x_2) \quad \wedge \quad (x_2 \vee x_3) \quad \wedge \quad (\neg x_2) \\ &= (x_1 \vee x_2) \quad \wedge \quad (x_2 \vee x_3) \quad \wedge \quad (\bar{x}_2)\end{aligned}$$

$x_1, x_2, x_3$ : **Boolesche Variablen** Ihr Wert ist entweder **false** oder **true**.  
Wir benutzen immer 0 für **false** und 1 für **true**

$\vee$  logisches ODER:  $x \vee y = 1$  genau dann, wenn  $x = 1$  oder  $y = 1$

$\wedge$  logisches UND:  $x \wedge y = 1$  genau dann, wenn  $x = 1$  und  $y = 1$

$\neg$  logische Negation:  $\neg x = 1$  genau dann, wenn  $x = 0$   
 $\bar{x} = 1$  genau dann, wenn  $x = 0$

**Wahrheitstabelle:**

$x_1$	$x_2$	$x_3$	$\phi$
0	0	0	0
0	0	1	0
...			
1	0	1	1
...			
5 - 17	1	1	0

**Belegung:** Eine Zuweisung von 0/1 zu allen Variablen

**Frage:** Wieviele Zeilen hat die Tabelle?

# Konjunktive Normalform (KNF)

Wir werden immer Formeln einer bestimmten Art betrachten:

**Definition (Konjunktive Normalform, KNF).** auch: conjunctive normal form (CNF)

Eine Boolesche Formel in KNF ist eine Formel  $\phi$  der Form

$$\phi = \bigwedge_i \left( \bigvee_j \ell_{ij} \right) \quad \text{ein "großes UND von ODERn"}$$

wobei jedes **Literal**  $\ell_{ij}$  entweder eine Variable  $x_i$  oder ihre Negation  $\bar{x}_i$  ist.

Wir nennen die  $\bigvee_j \ell_{ij}$ -Terme die **Klauseln** von  $\phi$ .

Eine solche Formel ist in  $k$ -KNF, wenn jede Klausel aus höchstens  $k$  Literalen besteht.

Ist  $(x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3 \vee x_4) \wedge (\bar{x}_2)$  eine KNF-Formel?

Ist  $(x_1 \wedge x_2) \vee (\bar{x}_2)$  eine KNF-Formel?

**Behauptung:** Jede Boolesche Funktion  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  lässt sich in KNF darstellen.

**Beweis:** Betrachte eine Belegung  $x_1 = b_1, \dots, x_n = b_n$  gegeben durch  $b = (b_1, \dots, b_n) \in \{0, 1\}^n$ .

Es gibt eine Klausel  $C_b$  sodass  $C_b(b) = 0$  und  $C_b(b') = 1$  wenn  $b' \neq b$ :

z.B. für  $b = (1, 0, 1, 1)$  ist  $C_b = (\bar{x}_1 \vee x_2 \vee \bar{x}_3 \vee \bar{x}_4)$

Wir definieren  $\phi = \bigwedge_{b:f(b)=0} C_b$

Man kann prüfen, dass  $\phi$  äquivalent zu  $f$  ist. Weiterhin ist  $\phi$  in KNF. □

# Erfüllbarkeitsproblem (SAT)

---

$\phi : \{0, 1\}^n \rightarrow \{0, 1\}$  sei eine Boolesche Formel mit  $n$  Variablen

Wir sagen  $\phi$  ist **erfüllbar**, wenn es eine Belegung  $x_1 = b_1, \dots, x_n = b_n$  gibt, sodass  $\phi$  true (1) ergibt.

In Formeln:  $\exists (b_1, \dots, b_n) \in \{0, 1\}^n : \phi(b_1, \dots, b_n) = 1$

→ eine solche Belegung  $(b_1, \dots, b_n)$  heißt **erfüllende Belegung**

## Erfüllbarkeitsproblem (SAT).

**Gegeben:** Eine Formel  $\phi$  in KNF. ( $n$  sei die Anzahl an Variablen von  $\phi$ .)

**Gesucht:** Ist  $\phi$  erfüllbar?

Das heißt, bestimme ob  $\exists b_1, \dots, b_n \in \{0, 1\}$  sodass  $\phi(b_1, \dots, b_n) = 1$

## Erfüllbarkeitsproblem (**k**-SAT).

**Gegeben:** Eine Formel  $\phi$  in **k**-KNF. ( $n$  sei die Anzahl an Variablen von  $\phi$ .)

**Gesucht:** Ist  $\phi$  erfüllbar?

Das heißt, bestimme ob  $\exists b_1, \dots, b_n \in \{0, 1\}$  sodass  $\phi(b_1, \dots, b_n) = 1$

# Quiz: Erfüllbarkeitsproblem

---

Ist  $\phi$  erfüllbar?

$$1. \phi = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2})$$

$$2. \phi = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3})$$

$$3. \phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_4} \vee \overline{x_5}) \wedge \\ (x_4 \vee \overline{x_5}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee x_4)$$

Wirkt aufwändig zu bestimmen...

**Frage:** Können Sie einen Algorithmus für dieses Problem angeben?  
Er darf beliebig langsam sein.

# Motivation für Komplexitätstheorie

---

## 2-SAT:

Gegeben Boolesche Formel  $\phi$  in 2-KNF, bestimme ob  $\phi$  erfüllbar ist.

## 3-SAT:

Gegeben Boolesche Formel  $\phi$  in 3-KNF, bestimme ob  $\phi$  erfüllbar ist.

## Kürzester Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **kürzesten** Pfades von  $s$  nach  $t$ .

## Längster Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **längsten** Pfades von  $s$  nach  $t$ .

**Frage:** Kennen wir bereits Algorithmen für irgendeins dieser Probleme?

# Motivation für Komplexitätstheorie

---

## 2-SAT:

Gegeben Boolesche Formel  $\phi$  in 2-KNF, bestimme ob  $\phi$  erfüllbar ist.

## 3-SAT:

Gegeben Boolesche Formel  $\phi$  in 3-KNF, bestimme ob  $\phi$  erfüllbar ist.

## Kürzester Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **kürzesten** Pfades von  $s$  nach  $t$ .  $O(n + m)$   
(via BFS)

## Längster Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **längsten** Pfades von  $s$  nach  $t$ .

**Frage:** Kennen wir bereits Algorithmen für irgendeins dieser Probleme?

# Motivation für Komplexitätstheorie

---

## 2-SAT:

Gegeben Boolesche Formel  $\phi$  in 2-KNF, bestimme ob  $\phi$  erfüllbar ist.

alle Belegungen überprüfen:  $\Omega(2^n)$  Zeit

## 3-SAT:

Gegeben Boolesche Formel  $\phi$  in 3-KNF, bestimme ob  $\phi$  erfüllbar ist.

alle Belegungen überprüfen:  $\Omega(2^n)$  Zeit

## Kürzester Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **kürzesten** Pfades von  $s$  nach  $t$ .  $O(n + m)$   
(via BFS)

## Längster Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **längsten** Pfades von  $s$  nach  $t$ .

alle Pfade überprüfen:  $\Omega(n!)$  Zeit

alle Pfade überprüfen:  $\Omega(n!)$  Zeit

**Frage:** Kennen wir bereits Algorithmen für irgendeins dieser Probleme?

# Motivation für Komplexitätstheorie

---

## 2-SAT:

Gegeben Boolesche Formel  $\phi$  in 2-KNF, bestimme ob  $\phi$  erfüllbar ist.

alle Belegungen überprüfen:  $\Omega(2^n)$  Zeit

## 3-SAT:

Gegeben Boolesche Formel  $\phi$  in 3-KNF, bestimme ob  $\phi$  erfüllbar ist.

alle Belegungen überprüfen:  $\Omega(2^n)$  Zeit

## Kürzester Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **kürzesten** Pfades von  $s$  nach  $t$ .  $O(n + m)$   
(via BFS)

## Längster Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **längsten** Pfades von  $s$  nach  $t$ .

alle Pfade überprüfen:  $\Omega(n!)$  Zeit

alle Pfade überprüfen:  $\Omega(n!)$  Zeit

**Sind diese Probleme unterschiedlich schwierig zu lösen?**

# Motivation für Komplexitätstheorie

---

## 2-SAT:

Gegeben Boolesche Formel  $\phi$  in 2-KNF, bestimme ob  $\phi$  erfüllbar ist.

$O(n + m)$

alle Belegungen überprüfen:  $\Omega(2^n)$  Zeit

## 3-SAT:

Gegeben Boolesche Formel  $\phi$  in 3-KNF, bestimme ob  $\phi$  erfüllbar ist.

alle Belegungen überprüfen:  $\Omega(2^n)$  Zeit

## Kürzester Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **kürzesten** Pfades von  $s$  nach  $t$ .

$O(n + m)$   
(via BFS)

alle Pfade überprüfen:  $\Omega(n!)$  Zeit

## Längster Pfad:

Gegeben  $G = (V, E)$  und  $s, t \in V$ , bestimme die Länge des **längsten** Pfades von  $s$  nach  $t$ .

alle Pfade überprüfen:  $\Omega(n!)$  Zeit

**Sind diese Probleme unterschiedlich schwierig zu lösen?**

**Spoiler alert:** · 2-SAT ist ebenfalls in Zeit  $O(n + m)$  lösbar.

· Wenn Sie einen  $O(n^{100})$ -Algorithmus für 3-SAT oder Längster Pfad entwickeln, können Sie 1.000.000 USD erhalten!

# **Die Klassen P und NP**

# Eine Vorbemerkung

---

Unser Berechnungsmodell für die Vorlesung ist die **RAM**.

Es gibt allerdings verschiedene Möglichkeiten, die in einem gewissen Sinne **äquivalent** zueinander sind.

In der Komplexitätstheorie wird häufig die **Turingmaschine (TM)** verwendet  
→ diese ist besonders geeignet, die zentralen Aussagen zu beweisen

**Ziel:** Berechnungsmodell, das

- einfach zu beschreiben ist, aber
- trotzdem so mächtig wie unsere RAM

**Wir werden diese Turingmaschine erst später besprechen.**

Man geht davon aus, dass es kein "mächtigeres" Berechnungsmodell als RAM/TM gibt.  
Dieser Gedanke ist ausgedrückt in der folgenden (nicht wohldefinierten) These:

## Church-Turing-These

Die Funktionen, die "intuitiv berechenbar" sind,  
sind genau die von einer TM berechenbaren Funktionen.

# Entscheidungsprobleme, Sprachen

Wir werden von nun an hauptsächlich **Entscheidungsprobleme** betrachten:

→ Die gewünschte Ausgabe ist **Ja (1)** oder **Nein (0)**

**Beispiele:** 1. Gegeben eine Folge Zahlen  $x_1, \dots, x_n \in \mathbb{N}$ , ist die Folge sortiert, d.h.  $x_1 \leq x_2 \leq \dots \leq x_n$ ?  
2. Gegeben ein ungerichteter Graph  $G = (V, E)$ , ist  $G$  zusammenhängend?

Entscheidungsprobleme kann man als formale **Sprachen** formalisieren.

Menge aller endlichen Folgen über  $\Sigma$ :  $\Sigma^* = \bigcup_{k \in \mathbb{N}_0} \Sigma^k$

**Definition.**

$\Sigma$  sei ein endliches Alphabet. Eine Teilmenge  $L$  von  $\Sigma^*$  ist eine **Sprache**.

Wir benutzen vor allem  $\Sigma = \{0, 1, \#\}$ . Dabei ist  $\#$  ein Trennzeichen.

$L$  besteht genau aus den Eingaben, für die wir **Ja (1)** zurückgeben wollen: **Ja-Instanzen**

**Sprache für Beispiel 1:**

$$L_1 = \{0, 1, 10, 11, \dots, 0\#0, 0\#1, 0\#10, \dots, 1\#1, 1\#10, \dots, 0\#0\#0, 0\#0\#1, \dots\}$$

→  $L_1$  beschreibt sortierte Folgen von Zahlen (mit Trennungszeichen #)

$$0\#10\#101 \in L_1$$

$$1\#0 \notin L_1$$

Es sei  $\text{bin}(x) \in \{0, 1\}^*$  die Darstellung von  $x \in \mathbb{N}$  als Binärzahl.

Dann ist  $L_1 = \{\text{bin}(x_1)\# \dots \#\text{bin}(x_n) \mid x_1, \dots, x_n \in \mathbb{N} \text{ mit } x_1 \leq x_2 \leq \dots \leq x_n\}$

**Konvention:**  $n = |x|$  ist die Länge der Eingabe  $x$ .

Im Kontext der Komplexitätstheorie gilt für uns:  
Eingabegröße = Bitlänge der Eingabe

# Algorithmen und Sprachen

---

Für Entscheidungsprobleme interpretieren wir die Ausgabe eines Algorithmus A wie folgt:

- wenn A eine 1 zurückgibt, so **akzeptiert** A die Eingabe
- ansonsten **verwirft** A die Eingabe

Die von A akzeptierte Sprache  $L_A$  ist gegeben durch:

$$x \in L_A \Leftrightarrow A \text{ akzeptiert die Eingabe } x$$

# Die Klasse P

---

Eine Laufzeitfunktion  $t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$  heißt **polynomiell beschränkt**,  
wenn es eine Konstante  $c$  gibt sodass  $t(n) \in O(n^c)$ .

polynomiell beschränkt:  $n \log n, n^2, n^{100}/\log n$

nicht polynomiell beschränkt:  $2^n, e^{\sqrt{n}}, n!$

Einen Algorithmus, der eine polynomiell beschränkte Worst-Case-Laufzeit hat,  
nennen wir **Polynomialzeitalgorithmus**

## Definition (P).

Menge von Problemen

P bezeichnet die Klasse von Entscheidungsproblemen (Sprachen),  
für die es einen Polynomialzeitalgorithmus gibt.

# Beispiele für Probleme in P

---

## ZSHG

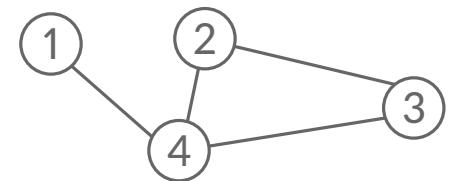
Gegeben ein ungerichteter Graph  $G = (V, E)$  als Adjazenzmatrix, ist  $G$  zusammenhängend?

→ Kennen RAM-Algorithmus mit **Laufzeit**:  $O(N)$ , wobei  $N = \Theta(|V|^2)$  die Eingabegröße ist

Beschreibung als Sprache  $L_{ZSHG}$ :

$$x = 0001\#0011\#0101\#1110 \in L_{ZSHG}$$

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



# Beispiele für Probleme in P

---

## ZSHG

Gegeben ein ungerichteter Graph  $G = (V, E)$  als Adjazenzmatrix, ist  $G$  zusammenhängend?

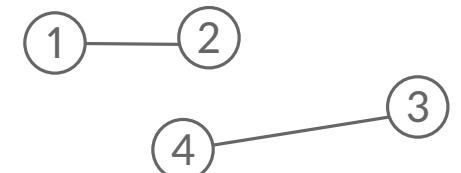
→ Kennen RAM-Algorithmus mit **Laufzeit**:  $O(N)$ , wobei  $N = \Theta(|V|^2)$  die Eingabegröße ist

Beschreibung als Sprache  $L_{ZSHG}$ :

$$x = 0001\#0011\#0101\#1110 \in L_{ZSHG}$$

$$x' = 0100\#1000\#0001\#0010 \notin L_{ZSHG}$$

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



# Beispiele für Probleme in P

---

## ZSHG

Gegeben ein ungerichteter Graph  $G = (V, E)$  als Adjazenzmatrix, ist  $G$  zusammenhängend?

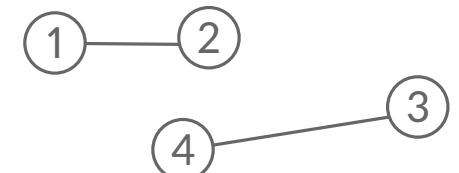
→ Kennen RAM-Algorithmus mit **Laufzeit**:  $O(N)$ , wobei  $N = \Theta(|V|^2)$  die Eingabegröße ist

Beschreibung als Sprache  $L_{ZSHG}$ :

$$x = 0001\#0011\#0101\#1110 \in L_{ZSHG}$$

$$x' = 0100\#1000\#0001\#0010 \notin L_{ZSHG}$$

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



**Hinweis:** Streng genommen erhalten wir unterschiedliche Probleme je nach gewählter Kodierung.  
(In bestimmten Fällen kann es einen Unterschied machen, wie genau die Kodierung gewählt ist.)

## DUPLIKAT

· Gegeben eine Folge  $x_1, \dots, x_n \in \mathbb{N}$ , taucht eine Zahl mehrfach auf?

→ Kennen RAM-Algorithmus mit **Laufzeit**:  $O(N \log N)$ , wobei  $N$  die Eingabegröße ist

Kodierung der Eingabe:

$$x = \text{bin}(x_1)\#\text{bin}(x_2)\#\dots\#\text{bin}(x_n)$$

## Erfüllbarkeitsproblem (**k**-SAT).

**Gegeben:** Eine Formel  $\phi$  in **k**-KNF. ( $n$  sei die Anzahl an Variablen von  $\phi$ .)

**Gesucht:** Ist  $\phi$  erfüllbar?

Das heißt, bestimme ob  $\exists b_1, \dots, b_n \in \{0, 1\}$  sodass  $\phi(b_1, \dots, b_n) = 1$

## Ist 3-SAT in **P**?

Das ist eine große offene Frage, die wir etwas genauer betrachten werden.

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_4} \vee \overline{x_5}) \wedge \\ (x_4 \vee \overline{x_5}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee x_4)$$

# Effiziente Verifizierbarkeit

---

$$\phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_4} \vee \overline{x_5}) \wedge \\ (x_4 \vee \overline{x_5}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee x_4)$$

Erfüllbarkeit von  $\phi$  scheint nicht direkt ablesbar.

Aber: Kennt man eine erfüllende Belegung, so kann man sich leicht von der Erfüllbarkeit überzeugen

**Behauptung:**  $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0$  erfüllt  $\phi$

**Nachweis:** trivial, werte die Formel aus

---

$$\phi' = (x_1 \vee \overline{x_2}) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_4} \vee \overline{x_5}) \wedge \\ (x_4 \vee \overline{x_5}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee x_4) \wedge (x_2 \vee x_5)$$

**Bemerkung:** Keine Belegung wird von der Erfüllbarkeit überzeugen können.

# Die Klasse NP

**Definition (Polynomiell beschränkte Verifiziererin).**

$L \subseteq \Sigma^*$  sei ein Entscheidungsproblem (Sprache).

Ein Algorithmus  $M$  ist eine **polynomiell beschränkte Verifiziererin** für  $L$ , wenn:

- $M$  hat polynomiell beschränkte Laufzeit
- Wir bezeichnen mit  $M(x, z)$  die Ausgabe von  $M$  auf Eingabe  $x \# z$ . Dann gelte:

$$x \in L \Leftrightarrow \exists z \in \{0, 1\}^{p(|x|)} : M(x, z) = 1$$

für ein Polynom  $p : \mathbb{N} \rightarrow \mathbb{N}$

Eine Verifiziererin  $M$  bekommt die Eingabe  $x$  für  $L$  sowie ein zusätzliches Zertifikat  $z$  polynomieller Länge

- $M$  akzeptiert  $x \# z$  nur, wenn  $x$  tatsächlich in der Sprache  $L$  ist ( $z$  ist ein gültiges Zertifikat)
- für jedes  $x \in L$  gibt es auch tatsächlich ein (kurzes) Zertifikat  $z$ , sodass  $M$  akzeptiert.  
( $z$  hat polynomielle Länge)

**Definition (NP).**

**NP** bezeichnet die Klasse von Entscheidungsproblemen (Sprachen), für die es eine polynomiell beschränkte Verifiziererin gibt.

# SAT ∈ NP

---

**Eigenschaft.** SAT ∈ NP

Damit gilt auch für jedes  $k$ , dass  $k$ -SAT ∈ NP

**Beweis:**

Wir müssen zeigen, dass es eine polynomiell beschränkte Verifiziererin  $M$  für SAT gibt.

$M$  sei wie folgt definiert:

- $M$  bekommt eine Eingabe  $\phi$  (Kodierung einer SAT Formel) und ein Zertifikat  $z \in \{0, 1\}^n$
- $M$  interpretiert  $z = (z_1, \dots, z_n)$  als Belegung für die Variablen  $x_1, \dots, x_n$  der Eingabeformel  $\phi$
- $M$  berechnet den Wahrheitswert der Eingabeformel unter der Belegung  $x_1 = z_1, \dots, x_n = z_n$ .  
→ wenn der berechnete Wert 1 ist, **akzeptiert**  $M$   
→ wenn der berechnete Wert 0 ist, **verwirft**  $M$

$M$  erfüllt die gewünschten Eigenschaften:

1.  $M$  hat polynomiell beschränkte Laufzeit  $O(n + |\phi|)$ .
2.  $\phi \in \text{SAT}$  genau dann, wenn  $\exists z \in \{0, 1\}^n : M(\phi, z) = 1$ :

Wenn  $\phi \in \text{SAT}$ , dann gibt es eine erfüllende Belegung  $b_1, \dots, b_n$ . Für  $z = (b_1, \dots, b_n)$  akzeptiert  $M$ .

Wenn  $\phi \notin \text{SAT}$ , dann gilt für jede Wahl von  $z = (b_1, \dots, b_n)$ , dass die Formel zu 0 evaluiert.  
→  $M$  verwirft für **jedes** Zertifikat  $z$ .

□

# NP: Weitere Bemerkungen

**Definition (Polynomiell beschränkte Verifiziererin).**

$L \subseteq \Sigma^*$  sei ein Entscheidungsproblem (Sprache).

Eine Algorithmus  $M$  ist eine **polynomiell beschränkte Verifiziererin** für  $L$ , wenn:

- $M$  hat polynomiell beschränkte Laufzeit
- Wir bezeichnen mit  $M(x, z)$  die Ausgabe von  $M$  auf Eingabe  $x \# z$ . Dann gelte:

$$x \in L \Leftrightarrow \exists z \in \{0, 1\}^{p(|x|)} : M(x, z) = 1$$

für ein Polynom  $p : \mathbb{N} \rightarrow \mathbb{N}$

**Definition (NP).**

**NP** bezeichnet die Klasse von Entscheidungsproblemen (Sprachen), für die es eine polynomiell beschränkte Verifiziererin gibt.

Es gilt:

$$P \subseteq NP$$

Frage: Warum?

Ist  $P \supseteq NP$ ?

offene Frage! Bekannt als "P vs NP"-Frage

Hinweis: Es gibt eine alternative Definition von NP mithilfe von **nichtdeterministischen Turingmaschinen**. ↗ spätere VL

# P vs NP

---

"Die" große Frage der theoretischen Informatik:

Ist  $P=NP?$

Es wird allgemein angenommen, dass  $P \neq NP$

Es ist eins der 7 Millennium Prize Problems  
→ eine Lösung dieser Frage würde mit 1.000.000 USD belohnt werden

Die Bedeutung ist allerdings weitreichender,  
besonders für die Kryptografie

Intuition, warum  $P=NP$  überraschend wäre:

Ich müsste lediglich Korrektheit einer Lösung schnell überprüfen können, schon könnte ich eine Lösung schnell finden...

## Algorithmen und Datenstrukturen SS'23

# Kapitel 15: Reduktionen und NP-Vollständigkeit

Marvin Künemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letztes Kapitel: die Klassen P und NP

Jetzt: NP-Vollständigkeit

Insbesondere besprechen wir in diesem Kapitel:

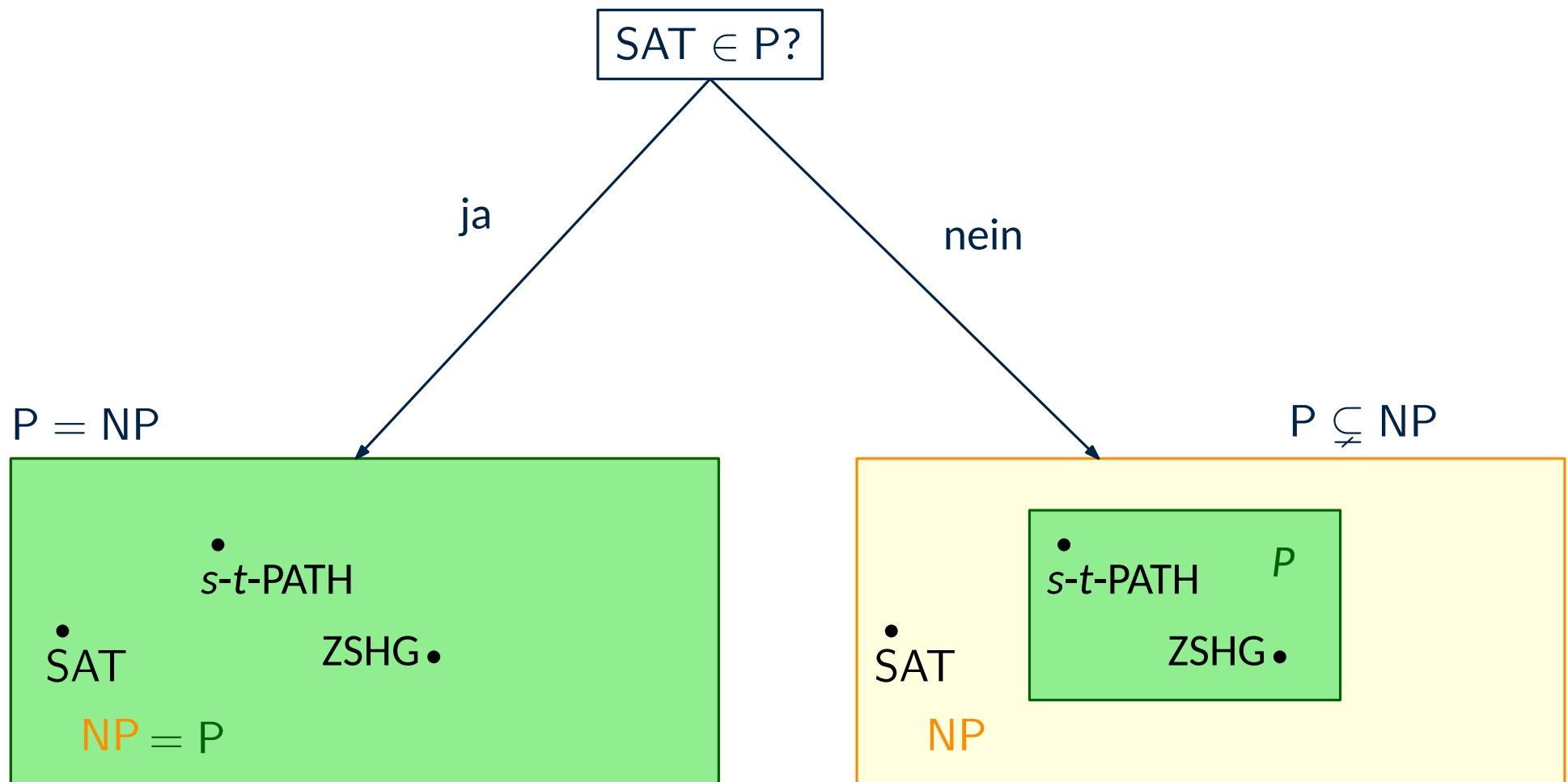
- Polynomialzeitreduktionen
- NP-Vollständigkeit
  - Definition
  - Cook's Theorem: 3SAT ist NP-vollständig
  - weitere Beispiele für NP-Vollständigkeit: Clique

# Thema dieses Kapitels

---

P: "effizient" lösbarer Probleme

NP: "effizient" verifizierbare Probleme

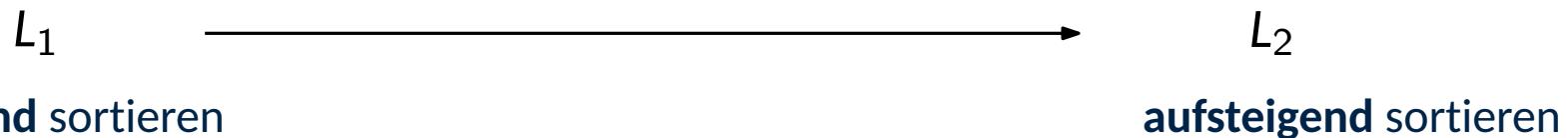


# **Polynomialzeitreduktionen**

# Reduktion: Intuition

## Reduktion von $L_1$ auf $L_2$ :

Ein Ansatz, Problem  $L_1$  zu lösen, indem man einen Algorithmus für  $L_2$  ausnutzt.



## Triviales Beispiel zur Veranschaulichung:

Angenommen, wir haben einen schnellen Algorithmus dafür, Zahlen  $x_1, \dots, x_n \in \mathbb{Z}$  **aufsteigend** zu sortieren

**Frage:** Können wir  $n$  Zahlen auch **absteigend** sortieren, ohne den Algorithmus A genau zu kennen?

## Eine Form des "Hackings"?

Wir tricksen einen Anbieter von Lösungen für  $L_2$  aus, um Problem  $L_1$  zu lösen!

# Polynomialzeitreduktion

**Definition (Polynomialzeitreduktion).**

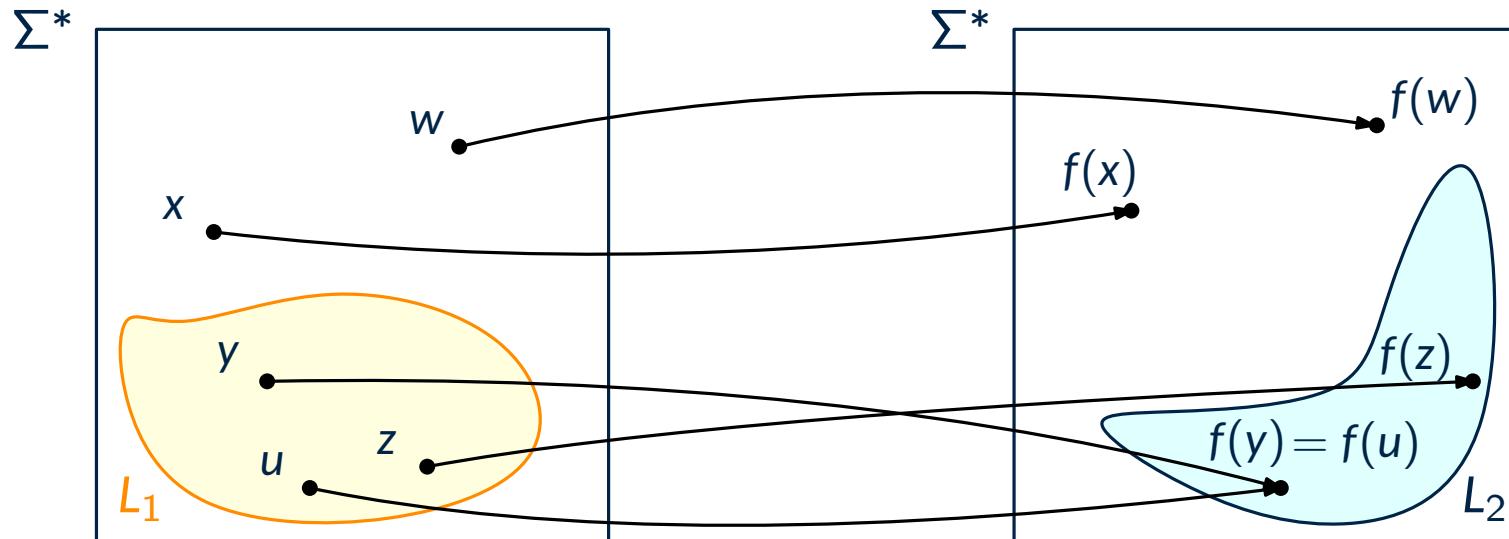
$L_1$  und  $L_2$  seien Sprachen über einem Alphabet  $\Sigma$ .

$L_1$  heißt **polynomiell reduzierbar auf  $L_2$** , wenn es eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  gibt, sodass:

- $f$  kann in Polynomialzeit berechnet werden
- für alle  $x \in \Sigma^*$  gilt:

$$x \in L_1 \quad \Leftrightarrow \quad f(x) \in L_2$$

Wir schreiben:  $L_1 \leq_P L_2$



# Wichtige Eigenschaften I

1. Wenn  $L_1 \leq_P L_2$  und  $L_2 \in P$ , so gilt  $L_1 \in P$ .

**Beweis:** Da  $L_1 \leq_P L_2$ , gibt es einen Algorithmus A, eine Funktion f und eine Konstante c, sodass:

- für alle x gilt:  $x \in L_1 \Leftrightarrow f(x) \in L_2$ . (1)
- A berechnet  $f(x)$  in Zeit  $O(|x|^c)$ .

Da  $L_2 \in P$ , gibt es einen Algorithmus  $A'$  und eine Konstante  $d \geq 1$ , sodass:

- $A'$  entscheidet  $y \in L_2$  in Zeit  $O(|y|^d)$

Betrachte folgenden Algorithmus  $A''$  für  $L_1$ :

- auf Eingabe x berechnen wir  $y := f(x)$  mithilfe von A
- wir entscheiden, ob  $y \in L_2$  mithilfe von  $A'$
- wenn ja, dann akzeptieren wir, ansonsten verwerfen wir

**Korrektheit:** Klar wegen (1)

**Laufzeit von  $A''$ :**

- der Laufzeitanteil von A ist  $O(|x|^c)$
  - der Laufzeitanteil von  $A'$  ist  $O(|y|^d) = O(|x|^{cd})$
- $\Rightarrow$  Gesamlaufzeit ist  $O(|x|^c) + O(|x|^{cd}) = O(|x|^{cd})$

$$|y| = O(|x|^c)$$

**Frage:** Warum?

$\Rightarrow A''$  entscheidet  $L_1$  in Polynomialzeit

□

# Wichtige Eigenschaften II

---

2. Wenn  $L_1 \leq_P L_2$  und  $L_2 \leq_P L_3$ , so gilt  $L_1 \leq_P L_3$ .

Transitivität von  $\leq_P$

Beweis: ↗ Übung

# **NP-Vollständigkeit**

# NP-Schwere

---

**Definition (NP-Schwere).** NP-hardness; manchmal auch in der Fehlübersetzung "NP-Härte"

Ein Problem  $L$  heißt **NP-schwer** (engl.: NP-hard), wenn gilt:

$$\forall L' \in \text{NP} : L' \leq_P L \quad \text{jedes Problem in NP lässt sich polynomiell auf } L \text{ reduzieren}$$

**Bemerkung:**

$L$  sei ein NP-schweres Problem. Wenn  $L \in P$ , dann ist  $P = \text{NP}$ .

**Beweis:** Sei  $L' \in \text{NP}$ . Dann gilt  $L' \leq_P L$  (NP-Schwere) und aus  $L \in P$  folgt  $L' \in P$ .  $\square$

Ein NP-schweres Problem effizient zu lösen ist so schwierig, wie **alle Probleme in NP** effizient zu lösen!

# NP-Vollständigkeit

---

**Definition (NP-Vollständigkeit).** NP-completeness

Ein Problem  $L$  heißt **NP-vollständig** (NP-complete), wenn gilt:

- $L \in \text{NP}$  und
- $L$  ist NP-schwer

Die Klasse der NP-vollständigen Probleme heißt auch NPC (für "NP-complete")

NP-vollständige Probleme bilden also die **schwierigsten Probleme innerhalb von NP!**

# Erstes NP-vollständiges Problem: SAT

---

Theorem (Satz von Cook-Levin).

SAT ist NP-vollständig.

Ein zentrales, wegbereitendes Resultat der theoretischen Informatik!

Konsequenz:

$$\text{SAT} \in \text{P} \iff \text{P} = \text{NP}$$

" $\Leftarrow$ ": trivial, da  $\text{SAT} \in \text{NP}$ .

" $\Rightarrow$ ": folgt aus NP-Schwere von SAT. ( $\nearrow$  Folie 10)

Beweis: kann aus Zeitgründen nicht besprochen werden.

---

Bedeutung:

Viele ForscherInnen vermuten, dass  $\text{P} \neq \text{NP}$  ist.

Wenn man also  $\text{P} \neq \text{NP}$  annimmt, kann es keinen Polynomialzeitalgorithmus für SAT geben.

# Nachweis von NP-Vollständigkeit

Wir werden zeigen: es gibt viele interessante Probleme, die NP-vollständig sind.

Wie weist man NP-Vollständigkeit eines gegebenen Problems  $L$  nach?

**Eigenschaft:** Wenn  $L'$  **NP-schwer** ist, und  $L' \leq_P L$ , dann ist auch  $L$  **NP-schwer**

**Beweis:** Sei  $L'' \in \text{NP}$ . Dann gilt  $L'' \leq_P L'$ . Wegen  $L' \leq_P L$  und Transitivität folgt  $L'' \leq_P L$ .

|  
NP-Schwere von  $L'$

□

Ein "**Rezept**" zum Nachweis von NP-Vollständigkeit eines Problems  $L$

1. Wir zeigen, dass  $L \in \text{NP}$ .
2. Wir wählen ein geeignetes **NP-vollständiges** Problems  $L'$ .
3. Wir überlegen uns, wie wir  $L'$  mithilfe von  $L$  lösen können  
→ Reduktionsfunktion  $f$ , die Eingaben für  $L'$  auf Eingaben für  $L$  abbildet
4. Wir zeigen, dass  $f$  in polynomieller Laufzeit berechnet werden kann
5. Wir zeigen **Korrektheit** der Reduktion:

Für alle Eingaben  $x$  gilt:  $x \in L' \iff f(x) \in L$

# **NP-Vollständigkeit von CLIQUE**

# CLIQUE: Definition

Cliquenproblem (CLIQUE).

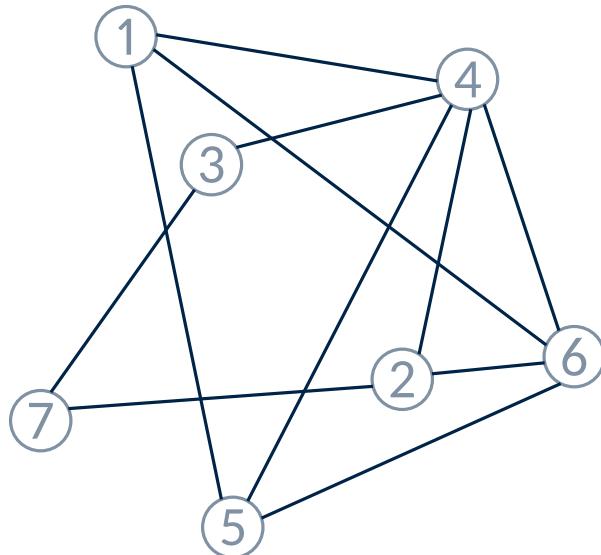
Gegeben: Ein ungerichteter Graph  $G = (V, E)$  und  $k \in \mathbb{N}$

Gesucht: Hat  $G$  eine  **$k$ -Clique**?

Eine Teilmenge  $S \subseteq V$  heißt  **$k$ -Clique**, wenn:

- $|S| = k$  und
- alle Paare von Knoten in  $S$  sind adjazent, d.h.  $\{u, v\} \in E$  für alle  $u, v \in S$  mit  $u \neq v$ .

$G$ :



Frage: Ist  $(G, 3)$  eine Ja-Instanz von CLIQUE?

# CLIQUE: Definition

Cliquenproblem (CLIQUE).

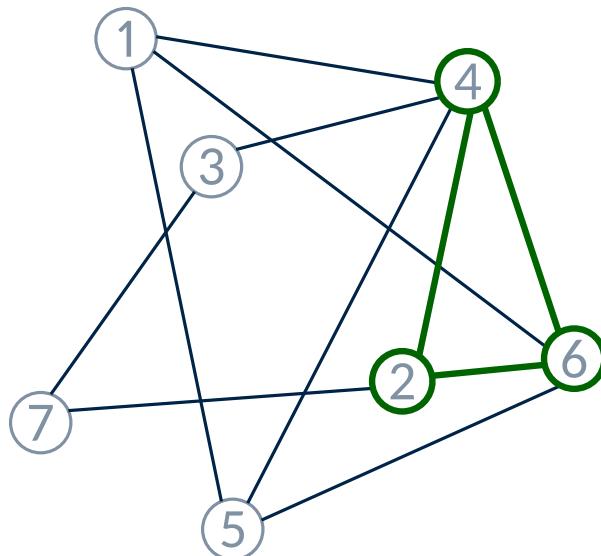
Gegeben: Ein ungerichteter Graph  $G = (V, E)$  und  $k \in \mathbb{N}$

Gesucht: Hat  $G$  eine  **$k$ -Clique**?

Eine Teilmenge  $S \subseteq V$  heißt  **$k$ -Clique**, wenn:

- $|S| = k$  und
- alle Paare von Knoten in  $S$  sind adjazent, d.h.  $\{u, v\} \in E$  für alle  $u, v \in S$  mit  $u \neq v$ .

$G$ :



Frage: Ist  $(G, 3)$  eine Ja-Instanz von CLIQUE?

# CLIQUE: Definition

Cliquenproblem (CLIQUE).

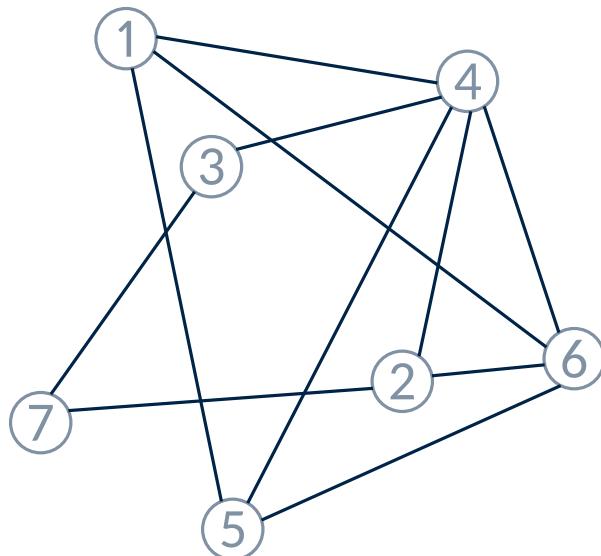
Gegeben: Ein ungerichteter Graph  $G = (V, E)$  und  $k \in \mathbb{N}$

Gesucht: Hat  $G$  eine  **$k$ -Clique**?

Eine Teilmenge  $S \subseteq V$  heißt  **$k$ -Clique**, wenn:

- $|S| = k$  und
- alle Paare von Knoten in  $S$  sind adjazent, d.h.  $\{u, v\} \in E$  für alle  $u, v \in S$  mit  $u \neq v$ .

$G$ :



Frage: Ist  $(G, 3)$  eine Ja-Instanz von CLIQUE?

Frage: Ist  $(G, 4)$  eine Ja-Instanz von CLIQUE?

# CLIQUE: Definition

Cliquenproblem (CLIQUE).

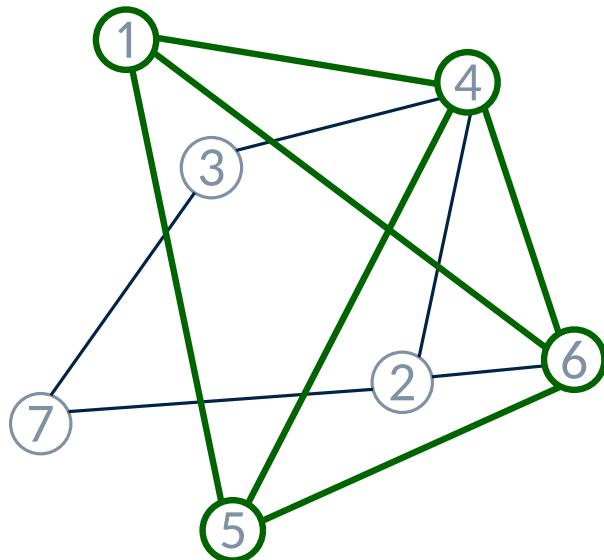
Gegeben: Ein ungerichteter Graph  $G = (V, E)$  und  $k \in \mathbb{N}$

Gesucht: Hat  $G$  eine  **$k$ -Clique**?

Eine Teilmenge  $S \subseteq V$  heißt  **$k$ -Clique**, wenn:

- $|S| = k$  und
- alle Paare von Knoten in  $S$  sind adjazent, d.h.  $\{u, v\} \in E$  für alle  $u, v \in S$  mit  $u \neq v$ .

$G$ :



Frage: Ist  $(G, 3)$  eine Ja-Instanz von CLIQUE?

Frage: Ist  $(G, 4)$  eine Ja-Instanz von CLIQUE?

# CLIQUE: Definition

Cliquenproblem (CLIQUE).

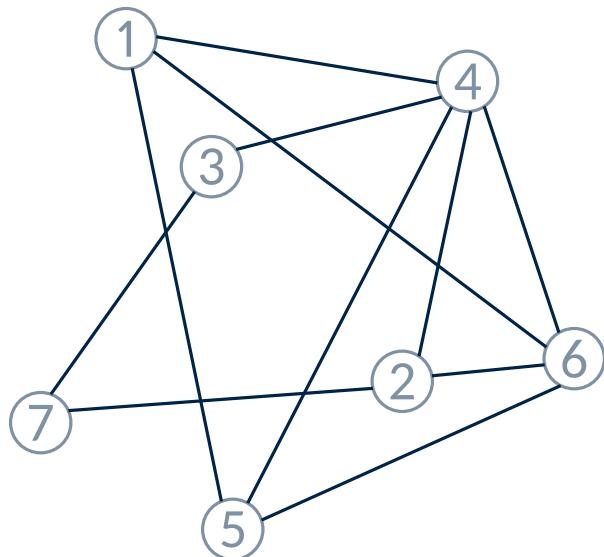
Gegeben: Ein ungerichteter Graph  $G = (V, E)$  und  $k \in \mathbb{N}$

Gesucht: Hat  $G$  eine  **$k$ -Clique**?

Eine Teilmenge  $S \subseteq V$  heißt  **$k$ -Clique**, wenn:

- $|S| = k$  und
- alle Paare von Knoten in  $S$  sind adjazent, d.h.  $\{u, v\} \in E$  für alle  $u, v \in S$  mit  $u \neq v$ .

$G$ :



Frage: Ist  $(G, 3)$  eine Ja-Instanz von CLIQUE?

Frage: Ist  $(G, 4)$  eine Ja-Instanz von CLIQUE?

Frage: Ist  $(G, 5)$  eine Ja-Instanz von CLIQUE?

# NP-Vollständigkeit von CLIQUE: Übersicht

---

**Theorem.**

CLIQUE ist NP-vollständig.

**Beweis.**

1. CLIQUE ist in NP:

Wir beschreiben eine polynomiell beschränkte Verifiziererin  $M$  für CLIQUE:

- $M$  bekommt als Eingabe einen Graphen  $G$ , eine Zahl  $k \in \mathbb{N}$  und ein **Zertifikat**  $z \in \{0, 1\}^n$ .
- $M$  interpretiert das Zertifikat  $z$  als Teilmenge  $S$  von  $V = \{1, \dots, n\}$ :

$$i \in S \Leftrightarrow z_i = 1$$

- $M$  überprüft, ob  $|S| = k$ . Wenn nein, verwirft  $M$ .
- $M$  überprüft für jedes  $\{u, v\} \in \binom{S}{2}$ , ob  $\{u, v\} \in E$ . Wenn nein, verwirft  $M$
- Wenn alle Überprüfungen erfolgreich waren, akzeptiert  $M$ .

**Laufzeit:**  $M$  läuft in Zeit  $O(n + k^2) = O(n^2)$ .  $\Rightarrow M$  ist polynomiell beschränkt.

**Korrektheit:** z.z.:  $G$  hat eine  $k$ -Clique  $S \Leftrightarrow \exists z \in \{0, 1\}^n$  sodass  $M$  die Eingabe  $G, k, z$  akzeptiert.

Betrachte die Bijektion zwischen  $S \subseteq V$  und  $z \in \{0, 1\}^n$  gegeben durch  $i \in S \Leftrightarrow z_i = 1$

Es gilt:  $S$  ist  $k$ -Clique in  $G \Leftrightarrow M$  akzeptiert die Eingabe  $G, k, z$ .

# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

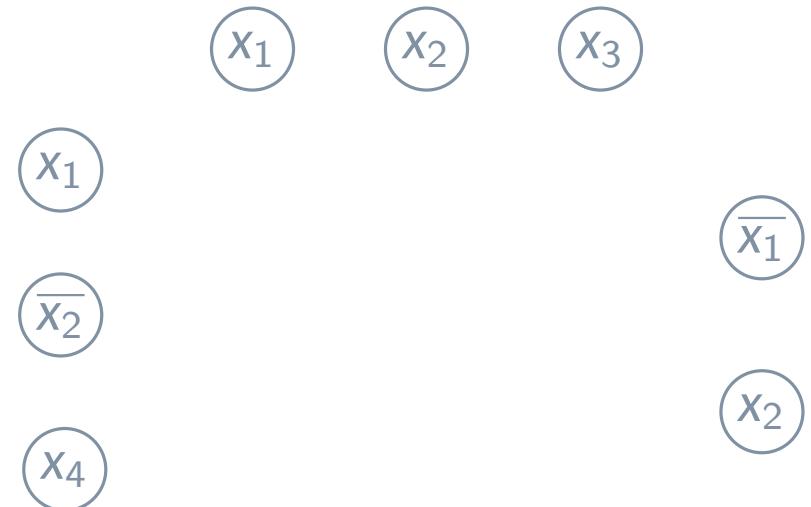
Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$



# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und
- $\ell$  und  $\ell'$  widersprechen sich nicht

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

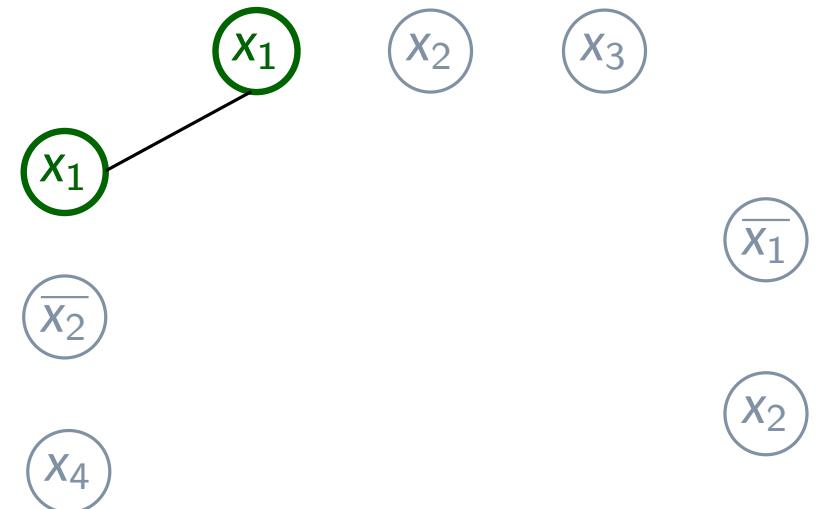
- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und

- $\ell$  und  $\ell'$  widersprechen sich nicht

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_p$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

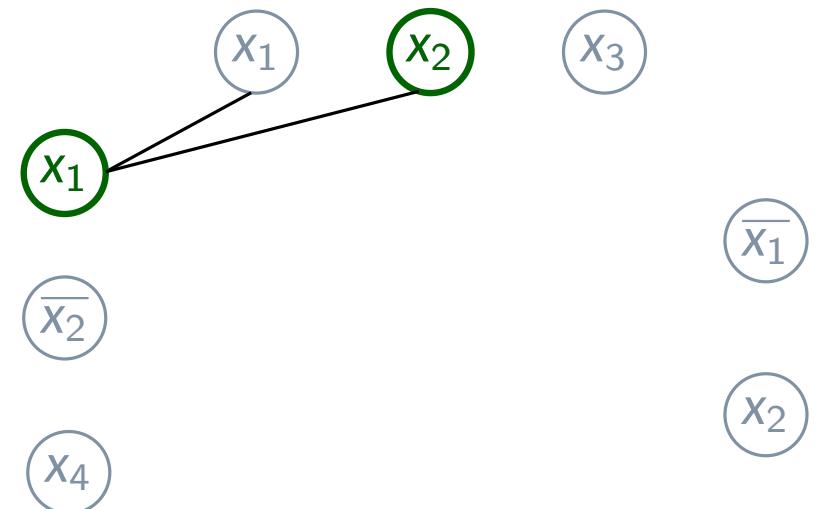
- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und

- $\ell$  und  $\ell'$  widersprechen sich nicht

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

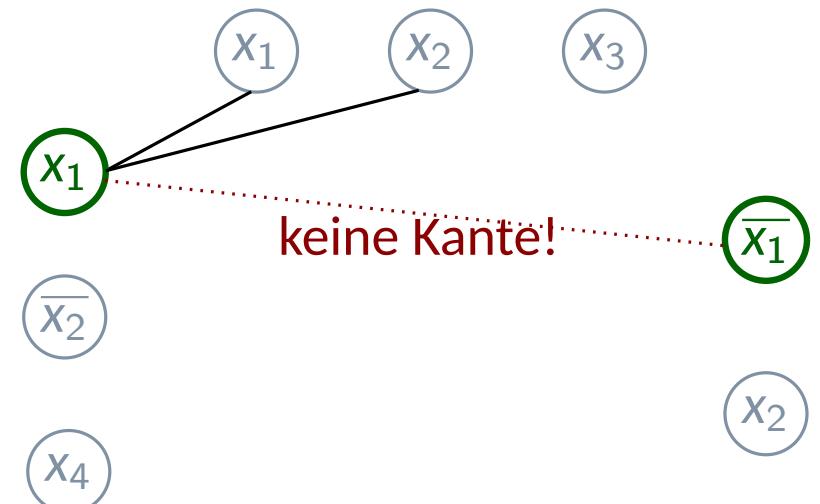
- $i \neq i'$  und

- $\ell$  und  $\ell'$  widersprechen sich nicht

$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und

- $\ell$  und  $\ell'$  widersprechen sich nicht

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_p$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und

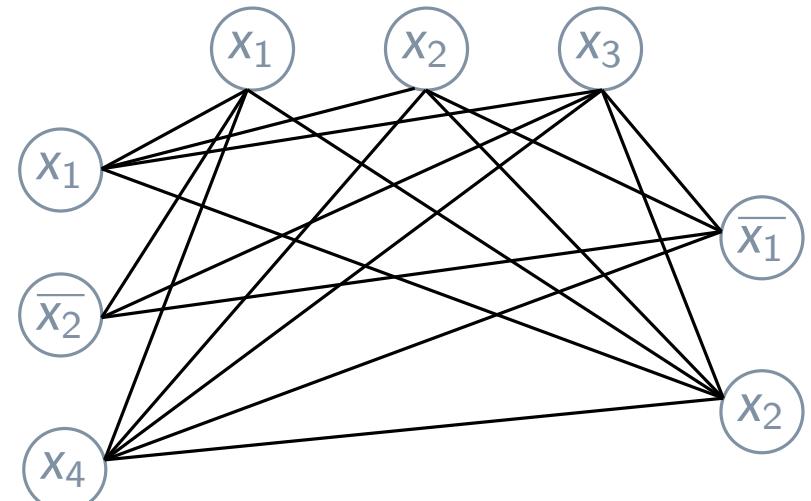
- $\ell$  und  $\ell'$  widersprechen sich nicht

→ dies bildet Graph G

$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



# NP-Vollständigkeit von CLIQUE: Reduktion

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_p$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \dots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und
- $\ell$  und  $\ell'$  widersprechen sich nicht

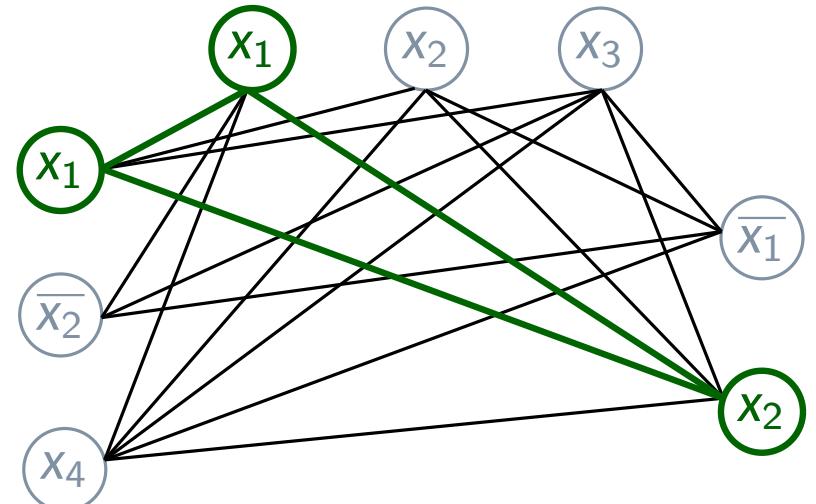
→ dies bildet Graph G

Definiere  $k = m$ .

$\ell$  und  $\ell'$  widersprechen sich,  
wenn  $\ell$  die Negation von  $\ell'$  ist

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$



Behauptung:

G hat  $m$ -Clique

$\Leftrightarrow$   
 $\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion II

---

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \cdots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:

- $i \neq i'$  und
- $\ell$  und  $\ell'$  widersprechen sich nicht

→ dies bildet Graph G

Definiere  $k = m$ .

# NP-Vollständigkeit von CLIQUE: Reduktion II

---

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

2. Wir wollen zeigen: SAT  $\leq_P$  CLIQUE.

3. Betrachte folgende Reduktion:

Es sei  $\phi = C_1 \wedge C_2 \cdots \wedge C_m$  eine SAT-Formel

Für jede Klausel  $C_i$  und jedes Literal  $\ell$  in  $C_i$ :

- führe einen Knoten  $u_{i,\ell}$  ein

Für jedes Knotenpaar  $u_{i,\ell}$  und  $u_{i',\ell'}$ :

- führe eine Kante  $\{u_{i,\ell}, u_{i',\ell'}\}$  ein, wenn:
  - $i \neq i'$  und
  - $\ell$  und  $\ell'$  widersprechen sich nicht

→ dies bildet Graph G

Definiere  $k = m$ .

4. Laufzeit der Reduktion:

· G hat höchstens  $|\phi|$  Knoten.

→ höchstens  $O(|\phi|^2)$  Knotenpaare

Können alle Kanten in Zeit  $O(|\phi|^2)$  bestimmen

→ können G, k in Polynomialzeit berechnen.

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

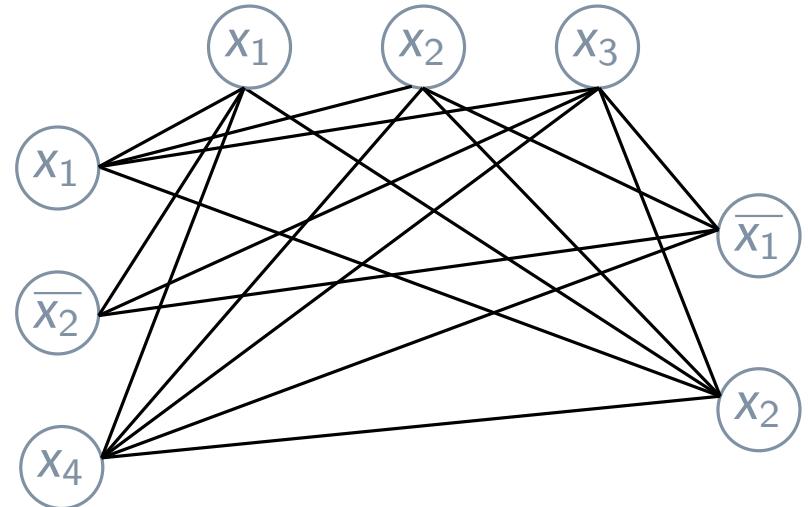
$\phi$  erfüllbar  $\Rightarrow G$  hat  $m$ -Clique:

Wähle eine erfüllende Belegung  $b$  für  $\phi$

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$

$$b = (0, 1, 0, 1)$$



Behauptung:

$G$  hat  $m$ -Clique

$\Leftrightarrow$

$\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

$\phi$  erfüllbar  $\Rightarrow G$  hat  $m$ -Clique:

Wähle eine erfüllende Belegung  $b$  für  $\phi$

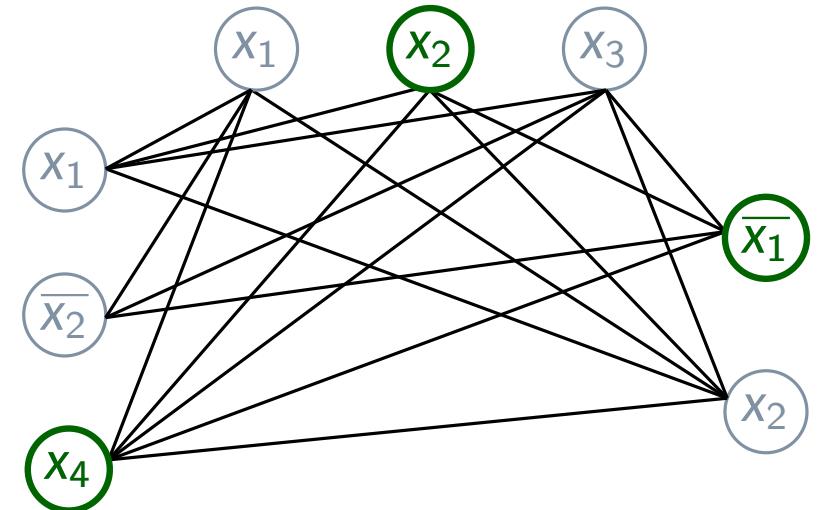
Für jede Klausel  $C_i$ :

- $b$  erfüllt mindestens ein Literal von  $C_i$   
 $\rightarrow$  es sei  $\ell_i$  ein solches erfülltes Literal

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$

$$b = (0, 1, 0, 1)$$



Behauptung:

$G$  hat  $m$ -Clique

$\Leftrightarrow$

$\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

$\phi$  erfüllbar  $\Rightarrow G$  hat  $m$ -Clique:

Wähle eine erfüllende Belegung  $b$  für  $\phi$

Für jede Klausel  $C_i$ :

- $b$  erfüllt mindestens ein Literal von  $C_i$   
 $\rightarrow$  es sei  $\ell_i$  ein solches erfülltes Literal

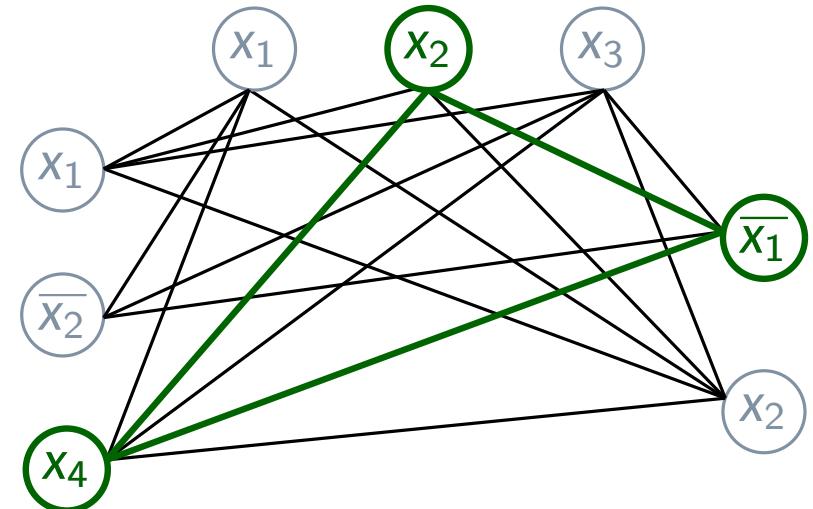
Dann ist  $\ell_1, \dots, \ell_m$  eine  $m$ -Clique:

- Kardinalität ist klar
- betrachte Knotenpaar  $\ell_i, \ell_j$  für  $i \neq j$ :
  - stammen von unterschiedlichen Klauseln
  - widersprechen sich nicht, da anhand  $b$  gewählt
  - $\rightarrow \ell_i$  und  $\ell_j$  in  $G$  adjazent

Beispiel:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$$

$$b = (0, 1, 0, 1)$$



Behauptung:

$G$  hat  $m$ -Clique

$\Leftrightarrow$

$\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

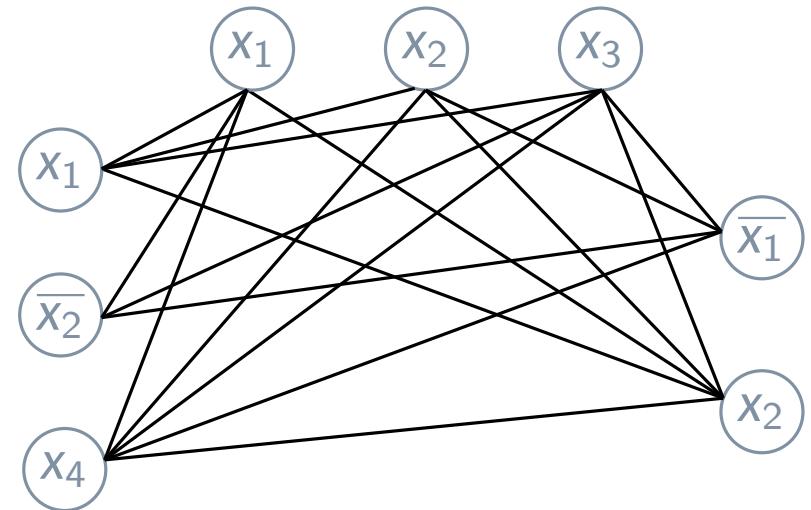
Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

$\phi$  erfüllbar  $\Rightarrow$  G hat m-Clique ✓

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$



Behauptung:

G hat m-Clique

$\Leftrightarrow$   
 $\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

$\phi$  erfüllbar  $\Rightarrow$  G hat m-Clique ✓

G hat m-Clique  $\Rightarrow$   $\phi$  erfüllbar

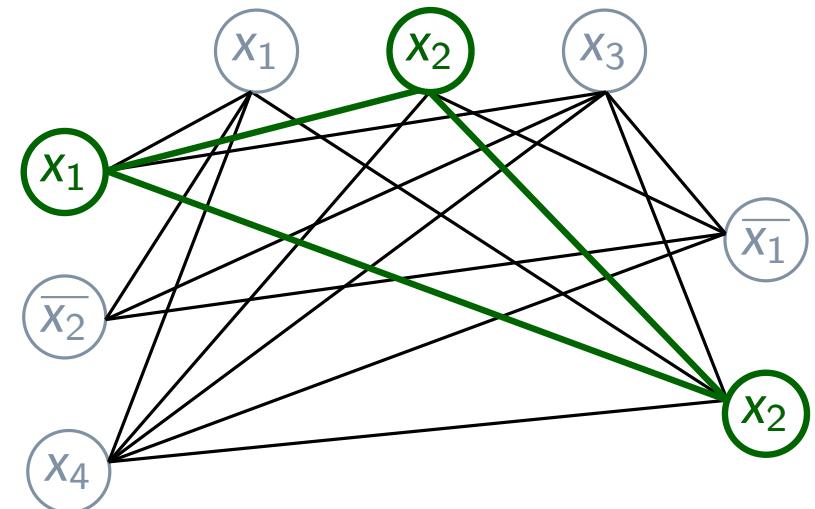
Es sei S eine m-Clique in G

Dann wählt S ein Literal  $\ell_i$  für jede Klausel  $C_i$ :

- würde S zwei oder mehr Literale aus  $C_i$  wählen, so wären diese nicht adjazent
- es gibt m Klauseln

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$



Behauptung:

G hat m-Clique

$\Leftrightarrow$

$\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

$\phi$  erfüllbar  $\Rightarrow$   $G$  hat  $m$ -Clique ✓

$G$  hat  $m$ -Clique  $\Rightarrow \phi$  erfüllbar

Es sei  $S$  eine  $m$ -Clique in  $G$

Dann wählt  $S$  ein Literal  $\ell_i$  für jede Klausel  $C_i$ :

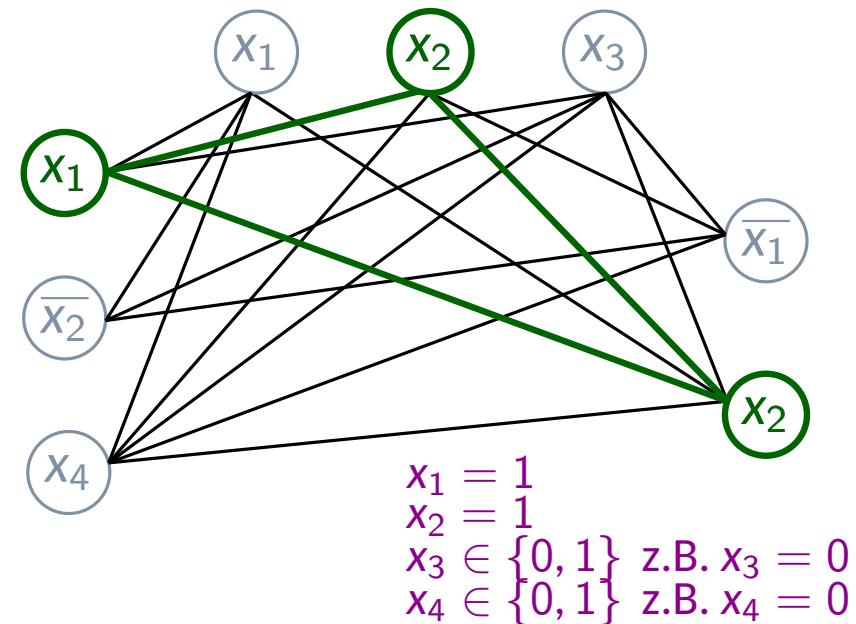
- würde  $S$  zwei oder mehr Literale aus  $C_i$  wählen, so wären diese nicht adjazent
- es gibt  $m$  Klauseln

Definieren eine Belegung  $b$ , die **konsistent** mit  $S$  ist:

- setze alle Literale aus  $S$  auf wahr
- verbleibende Variablen werden beliebig gesetzt

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$



Behauptung:

$G$  hat  $m$ -Clique

$\Leftrightarrow$   
 $\phi$  ist erfüllbar

# NP-Vollständigkeit von CLIQUE: Reduktion III

Theorem.

CLIQUE ist NP-vollständig.

Beweis (Fortsetzung).

## 5. Korrektheit der Reduktion:

$\phi$  erfüllbar  $\Rightarrow$   $G$  hat  $m$ -Clique ✓

$G$  hat  $m$ -Clique  $\Rightarrow \phi$  erfüllbar

Es sei  $S$  eine  $m$ -Clique in  $G$

Dann wählt  $S$  ein Literal  $\ell_i$  für jede Klausel  $C_i$ :

- würde  $S$  zwei oder mehr Literale aus  $C_i$  wählen, so wären diese nicht adjazent
- es gibt  $m$  Klauseln

Definieren eine Belegung  $b$ , die **konsistent** mit  $S$  ist:

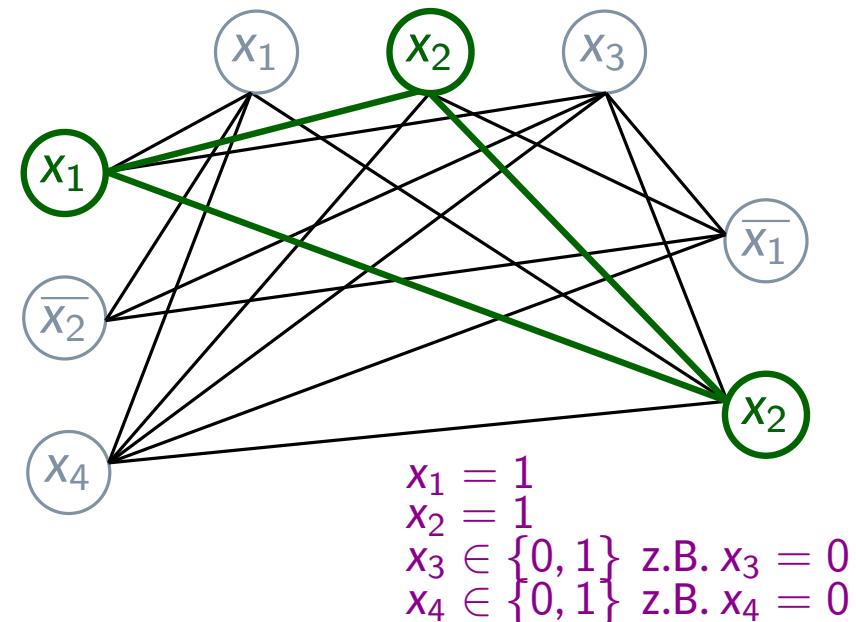
- setze alle Literale aus  $S$  auf wahr
- verbleibende Variablen werden beliebig gesetzt

Eine solche Belegung existiert, denn die Literale in  $S$  widersprechen sich nicht.

$b$  ist erfüllend: für jede Klausel ist min. 1 Literal erfüllt

Beispiel:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2)$$



Behauptung:

$G$  hat  $m$ -Clique

$\Leftrightarrow$

$\phi$  ist erfüllbar

# Karp's Liste mit 21 NP-vollständigen Problemen

---

Theorem (Karp '72).

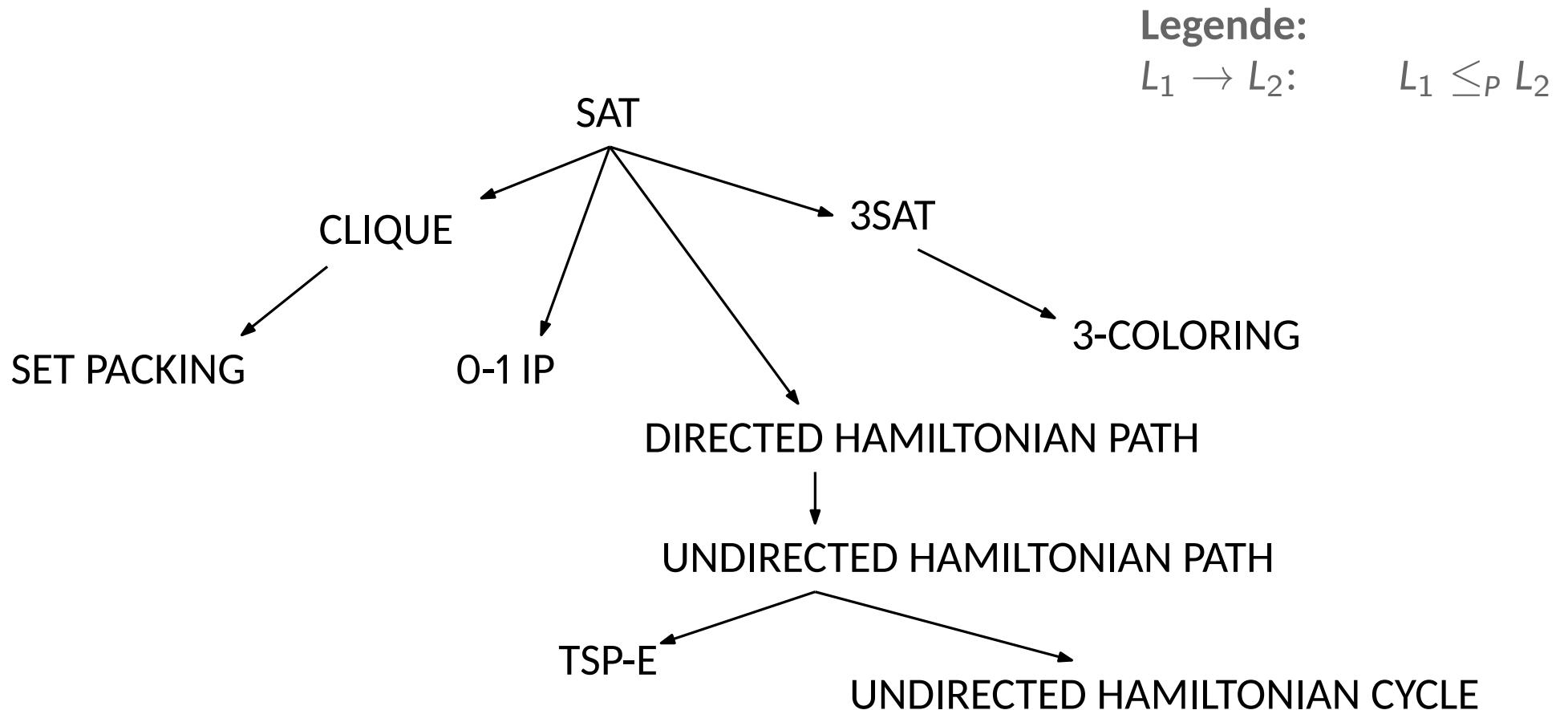
Die folgenden Probleme sind **NP-vollständig**:

- SAT
- 3-SAT
- CLIQUE
- SET PACKING
- VERTEX COVER
- SET COVERING
- HAMILTONIAN CYCLE
- DIRECTED HAMILTONIAN CYCLE
- FEEDBACK VERTEX SET
- FEEDBACK ARC SET
- 0-1 INTEGER PROGRAMMING
- CHROMATIC NUMBER
- CLIQUE COVER
- EXACT COVER
- STEINER TREE
- HITTING SET
- 3-DIMENSIONAL MATCHING
- KNAPSACK
- JOB SEQUENCING
- PARTITION
- MAX-CUT

Heutzutage kennen wir unzählige NP-vollständige Probleme  
→ wir werden ein paar wichtige näher besprechen

# Ein Netz von Reduktionen

---



# SAT $\leq_P$ 3SAT

---

**Lemma.** SAT  $\leq_P$  3SAT

Da wir bereits wissen, dass 3SAT  $\in$  NP, folgt daraus, dass 3SAT ein NP-vollständiges Problem ist.

## Vorbereitung des Beweises:

Sei  $\phi$  eine gegebene Formel in KNF.

Wollen eine 3-KNF-Formel  $\phi'$  konstruieren, sodass:  $\phi$  erfüllbar  $\Leftrightarrow \phi'$  erfüllbar

**Idee:** Wir benutzen eine **Hilfsvariable**, um eine "große" Klausel mit  $k > 3$  Literalen durch zwei kleinere Klauseln zu ersetzen!

**Beispiel:** Betrachte die Klausel  $C = (x_1 \vee \overline{x_2} \vee x_3 \vee \overline{x_4} \vee \overline{x_5})$

Wir führen eine neue Hilfsvariable  $h$  ein und ersetzen  $C$  durch:

$$C_1 = (x_1 \vee \overline{x_2} \vee x_3 \vee h) \text{ und } C_2 = (\overline{h} \vee \overline{x_4} \vee \overline{x_5})$$

C ist erfüllt  $\Leftrightarrow$  es existiert eine Belegung für  $h$  sodass  $C_1 \wedge C_2$  erfüllt ist.

# SAT $\leq_P$ 3SAT

---

**Lemma.** SAT  $\leq_P$  3SAT

Da wir bereits wissen, dass 3SAT  $\in$  NP, folgt daraus, dass 3SAT ein NP-vollständiges Problem ist.

## Vorbereitung des Beweises:

Sei  $\phi$  eine gegebene Formel in KNF.

Wollen eine 3-KNF-Formel  $\phi'$  konstruieren, sodass:  $\phi$  erfüllbar  $\Leftrightarrow \phi'$  erfüllbar

**Idee:** Wir benutzen eine **Hilfsvariable**, um eine "große" Klausel mit  $k > 3$  Literalen durch zwei kleinere Klauseln zu ersetzen!

**Beispiel:** Betrachte die Klausel  $C = (x_1 \vee \overline{x_2} \vee x_3 \vee \overline{x_4} \vee \overline{x_5})$

Wir führen eine neue Hilfsvariable  $h$  ein und ersetzen  $C$  durch:

$$C_1 = (x_1 \vee \overline{x_2} \vee x_3 \vee h) \text{ und } C_2 = (\overline{h} \vee \overline{x_4} \vee \overline{x_5})$$

C ist erfüllt  $\Leftrightarrow$  es existiert eine Belegung für  $h$  sodass  $C_1 \wedge C_2$  erfüllt ist.

C<sub>1</sub> ist leider immer noch zu groß...

# SAT $\leq_P$ 3SAT

---

**Lemma.** SAT  $\leq_P$  3SAT

Da wir bereits wissen, dass 3SAT  $\in$  NP, folgt daraus, dass 3SAT ein NP-vollständiges Problem ist.

## Vorbereitung des Beweises:

Sei  $\phi$  eine gegebene Formel in KNF.

Wollen eine 3-KNF-Formel  $\phi'$  konstruieren, sodass:  $\phi$  erfüllbar  $\Leftrightarrow \phi'$  erfüllbar

**Idee:** Wir benutzen eine **Hilfsvariable**, um eine "große" Klausel mit  $k > 3$  Literalen durch zwei kleinere Klauseln zu ersetzen!

**Beispiel:** Betrachte die Klausel  $C = (x_1 \vee \overline{x_2} \vee x_3 \vee \overline{x_4} \vee \overline{x_5})$

Wir führen eine neue Hilfsvariable  $h$  ein und ersetzen  $C$  durch:

$$C_1 = (x_1 \vee \overline{x_2} \vee x_3 \vee h) \text{ und } C_2 = (\overline{h} \vee \overline{x_4} \vee \overline{x_5})$$

$C$  ist erfüllt  $\Leftrightarrow$  es existiert eine Belegung für  $h$  sodass  $C_1 \wedge C_2$  erfüllt ist.

$C_1$  ist leider immer noch zu groß... wir wiederholen den Trick!

Wir führen weitere Hilfsvariable  $h_2$  ein und ersetzen  $C_1$  durch

$$C_{1,1} = (x_1 \vee \overline{x_2} \vee h_2) \text{ und } C_{1,2} = (\overline{h_2} \vee x_3 \vee h)$$

# Klauseltransformation

---

## Klauseltransformation:

Sei  $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$  eine Klausel von  $\phi$  mit  $k > 3$  Literalen.

Wir erzeugen eine **Hilfsvariable**  $h$  und ersetzen  $C$  durch die beiden Klauseln

$$(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_{k-2} \vee h) \text{ und } (\bar{h} \vee \ell_{k-1} \vee \ell_k)$$

## Eigenschaft:

Eine Klauseltransformation erzeugt eine Formel  $\phi'$ ,  
die erfüllbar ist genau dann, wenn die Ursprungsformel erfüllbar ist.

## Beweis von $SAT \leq_P 3SAT$ :

Sei  $\phi$  eine gegebene Formel in KNF.

Für jede Klausel  $C$  mit  $k > 3$  Literalen:

- wende  $(k - 3)$  Klauseltransformationen an, um  $C$  vollständig durch eine 3KNF-Formel zu ersetzen
- wir erhalten eine 3-KNF Formel  $\phi'$ , die erfüllbar ist genau dann wenn  $\phi$  erfüllbar ist.

Es sei  $N$  die Gesamtanzahl von Literalen von  $\phi$ . Dann machen wir  $\leq N - 3$  Klauseltransformationen.  
⇒  $\phi'$  lässt sich in Polynomialzeit berechnen. □

# Zusammenfassung

---

Wir besprachen in diesem Kapitel:

- Polynomialzeitreduktionen
- NP-Vollständigkeit
  - Definition
  - Cook's Theorem: 3SAT ist NP-vollständig
  - weitere Beispiele für NP-Vollständigkeit: Clique

Algorithmen und Datenstrukturen SS'23

# Kapitel 16: Weitere NP-vollständige Probleme

Marvin Künnemann

AG Algorithmen & Komplexität

# Kapitelüberblick

---

Letztes Kapitel: NP-Vollständigkeit von SAT, 3SAT, Clique

Jetzt: Weitere wichtige NP-vollständige Probleme

Insbesondere besprechen wir in diesem Kapitel:

- 3-Färbbarkeit (3-Coloring)
- Traveling Salesperson Problem (TSP)
- SubsetSum und das Rucksackproblem

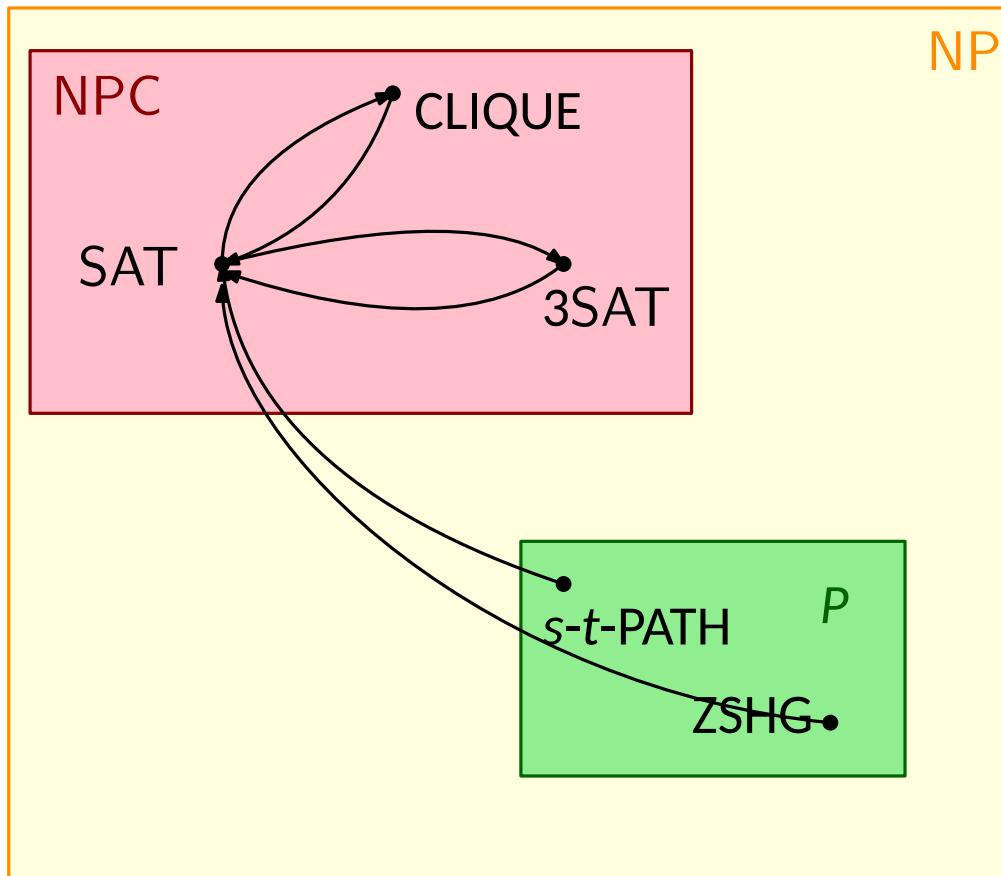
# **Wiederholung**

P: "effizient" lösbar Probleme

NP: "effizient" verifizierbare Probleme

NPC: die "schwierigsten" Probleme in NP

Für dieses Bild nehmen wir an, dass  $P \neq NP$ :



- Cook-Levin-Theorem:  
SAT ist NP-vollständig
- $SAT \leq_P CLIQUE$   
→ CLIQUE ist NP-vollständig
- $SAT \leq_P 3SAT$   
→ 3SAT ist NP-vollständig

## Definition (Polynomialzeitreduktion).

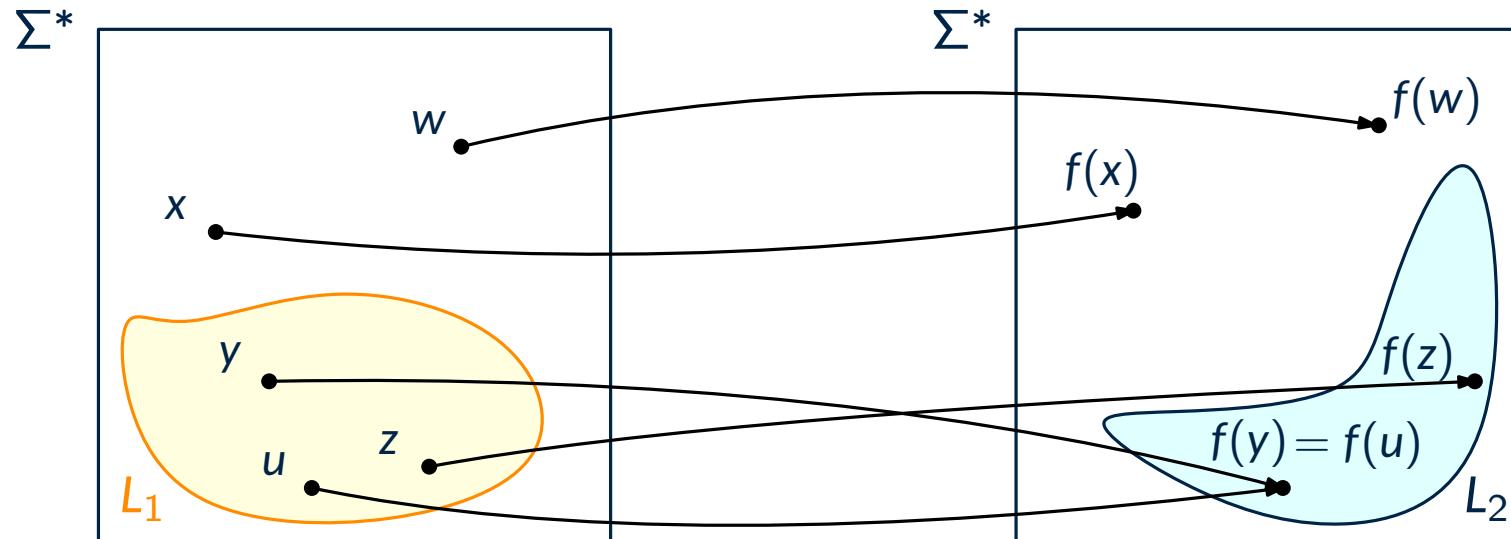
$L_1$  und  $L_2$  seien Sprachen über einem Alphabet  $\Sigma$ .

$L_1$  heißt **polynomiell reduzierbar auf  $L_2$** , wenn es eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  gibt, sodass:

- $f$  kann in Polynomialzeit berechnet werden
- für alle  $x \in \Sigma^*$  gilt:

$$x \in L_1 \quad \Leftrightarrow \quad f(x) \in L_2$$

Wir schreiben:  $L_1 \leq_P L_2$



# Nachweis von NP-Vollständigkeit

WIEDERHOLUNG

---

Ein "Rezept" zum Nachweis von NP-Vollständigkeit eines Problems  $L$

1. Wir zeigen, dass  $L \in NP$ .
2. Wir wählen ein geeignetes **NP-vollständiges** Problems  $L'$ .
3. Wir überlegen uns, wie wir  $L'$  mithilfe von  $L$  lösen können  
→ Reduktionsfunktion  $f$ , die Eingaben für  $L'$  auf Eingaben für  $L$  abbildet
4. Wir zeigen, dass  $f$  in polynomieller Laufzeit berechnet werden kann
5. Wir zeigen **Korrektheit** der Reduktion:

$$\text{Für alle Eingaben } x \text{ gilt: } x \in L' \iff f(x) \in L$$

# Färbbarkeitsprobleme

# **$k$ -Färbbarkeit**

---

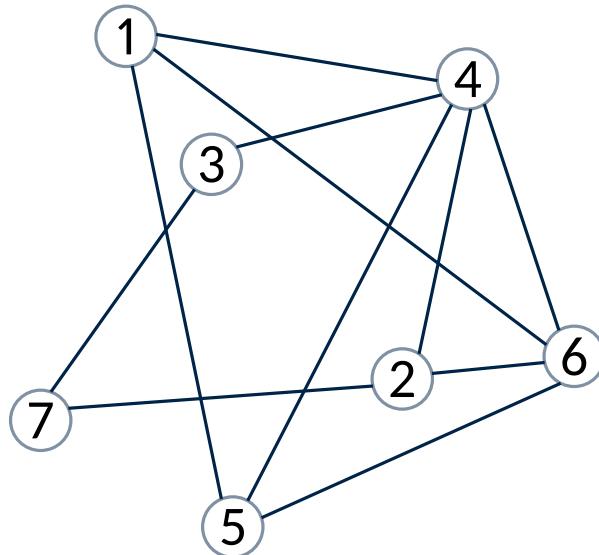
**$k$ -Färbbarkeitsproblem ( $k$ -COLORING).**

**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$

**Gesucht:** Ist  $G$   **$k$ -färbbar**?

$G$  heißt  **$k$ -färbbar**, wenn es eine Färbung  $c : V \rightarrow \{1, \dots, k\}$  gibt, sodass alle adjazente Knoten unterschiedliche Farben bekommen, d.h.:  
 $\forall \{u, v\} \in E: c(u) \neq c(v)$ .

$G$ :



# $k$ -Färbbarkeit

**$k$ -Färbbarkeitsproblem ( $k$ -COLORING).**

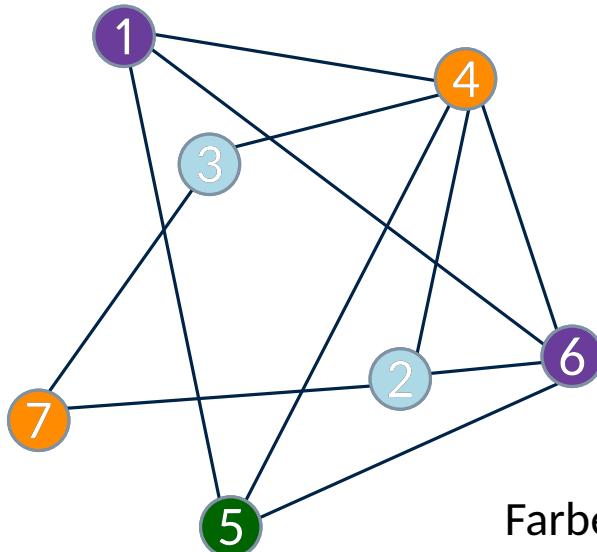
**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$

**Gesucht:** Ist  $G$   $k$ -färbbar?

$G$  heißt  $k$ -färbbar, wenn es eine Färbung  $c : V \rightarrow \{1, \dots, k\}$  gibt, sodass alle adjazente Knoten unterschiedliche Farben bekommen, d.h.:  
 $\forall \{u, v\} \in E: c(u) \neq c(v)$ .

Betrachte  $k = 4$

$G$ :



Frage: Ist  $c$  eine gültige 4-Färbung?

Farbe  $\leftrightarrow \{1, \dots, k\}$



1



2



3



4

# $k$ -Färbbarkeit

**$k$ -Färbbarkeitsproblem ( $k$ -COLORING).**

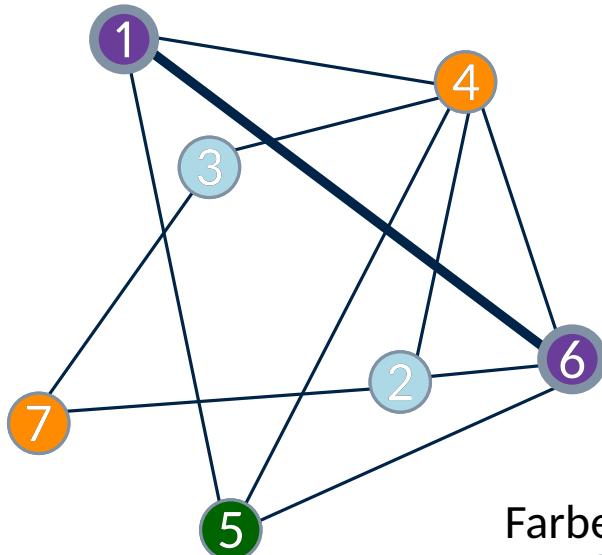
**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$

**Gesucht:** Ist  $G$   $k$ -färbbar?

$G$  heißt  $k$ -färbbar, wenn es eine Färbung  $c : V \rightarrow \{1, \dots, k\}$  gibt, sodass alle adjazente Knoten unterschiedliche Farben bekommen, d.h.:  
 $\forall \{u, v\} \in E: c(u) \neq c(v)$ .

Betrachte  $k = 4$

$G$ :



Frage: Ist  $c$  eine gültige 4-Färbung? Nein

Farbe  $\leftrightarrow \{1, \dots, k\}$



1



2



3



4

# $k$ -Färbbarkeit

**$k$ -Färbbarkeitsproblem ( $k$ -COLORING).**

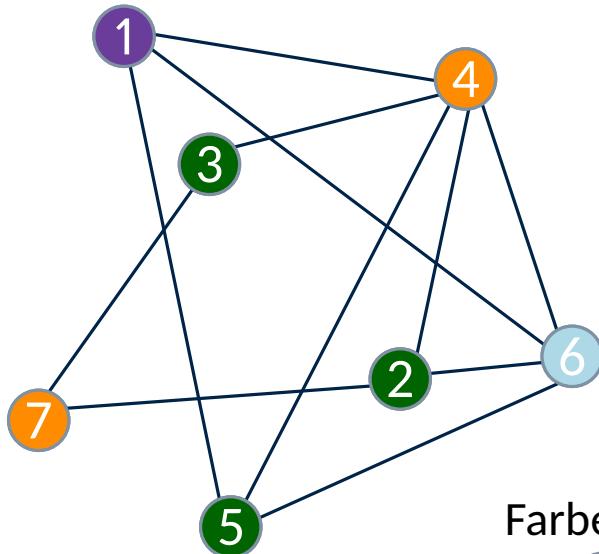
**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$

**Gesucht:** Ist  $G$   $k$ -färbbar?

$G$  heißt  $k$ -färbbar, wenn es eine Färbung  $c : V \rightarrow \{1, \dots, k\}$  gibt, sodass alle adjazente Knoten unterschiedliche Farben bekommen, d.h.:  
 $\forall \{u, v\} \in E: c(u) \neq c(v)$ .

Betrachte  $k = 4$

$G$ :



Frage: Ist  $c$  eine gültige 4-Färbung von  $G$ ?

Farbe  $\leftrightarrow \{1, \dots, k\}$



1



2



3



4

# $k$ -Färbbarkeit

**$k$ -Färbbarkeitsproblem ( $k$ -COLORING).**

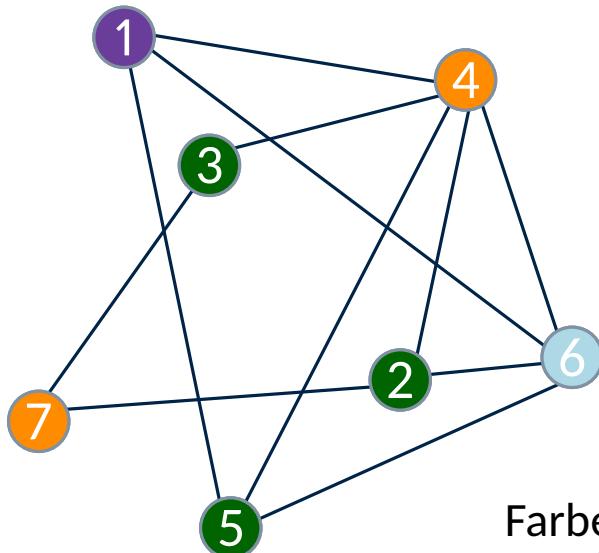
**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$

**Gesucht:** Ist  $G$   $k$ -färbbar?

$G$  heißt  $k$ -färbbar, wenn es eine Färbung  $c : V \rightarrow \{1, \dots, k\}$  gibt, sodass alle adjazente Knoten unterschiedliche Farben bekommen, d.h.:  
 $\forall \{u, v\} \in E: c(u) \neq c(v)$ .

Betrachte  $k = 4$

$G$ :



**Frage:** Ist  $c$  eine gültige 4-Färbung von  $G$ ? Ja!  
→  $G$  ist 4-färbbar

Farbe  $\leftrightarrow \{1, \dots, k\}$



1



2



3



4

# $k$ -Färbbarkeit

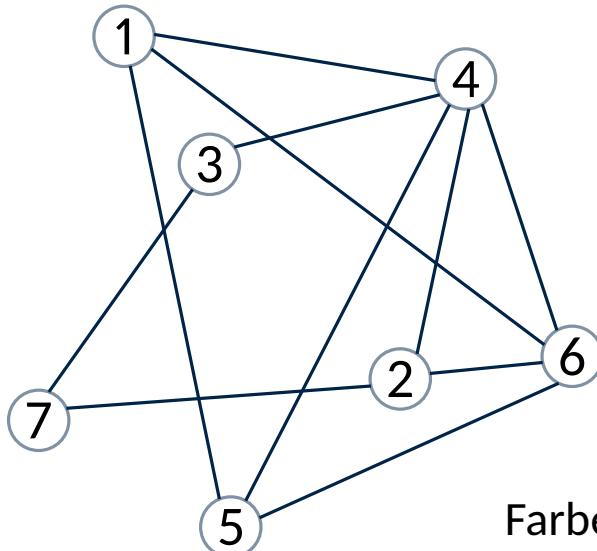
**$k$ -Färbbarkeitsproblem ( $k$ -COLORING).**

**Gegeben:** Ein ungerichteter Graph  $G = (V, E)$

**Gesucht:** Ist  $G$   $k$ -färbbar?

$G$  heißt  $k$ -färbbar, wenn es eine Färbung  $c : V \rightarrow \{1, \dots, k\}$  gibt, sodass alle adjazente Knoten unterschiedliche Farben bekommen, d.h.:  
 $\forall \{u, v\} \in E: c(u) \neq c(v)$ .

$G$ :



Frage: Ist  $G$  3-färbbar?

Farbe  $\leftrightarrow \{1, \dots, k\}$



1



2



3



4

# NP-Schwere der 3-Färbbarkeit

---

**Theorem.** 3-COLORING ist NP-schwer.

Mit diesem Resultat können wir NP-Vollständigkeit von  $k$ -Färbbarkeit für alle  $k \geq 3$  zeigen.  Übung

Wir reduzieren vom NP-vollständigen Problem 3SAT.

**Aufgabe:**

Für eine gegebene 3-KNF-Formel  $\phi$  wollen wir einen Graphen  $G$  konstruieren, sodass:

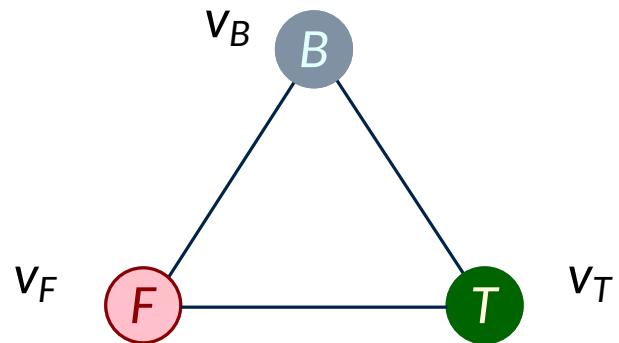
$$\phi \text{ ist erfüllbar} \iff G \text{ ist 3-färbbar}$$

Für dieses Ziel konstruieren wir bestimmte **Gadgets**  kleine Bausteine für die Reduktion

# 1. Gadget: Definition der Farben

---

Konstruiere ein Dreieck bestehend aus Knoten  $v_B, v_T, v_F$ :



In jeder gültigen 3-Färbung bekommen  $v_B, v_F, v_T$  unterschiedliche Farben  
→ wir bezeichnen die Farben von  $v_B, v_F$  bzw.  $v_T$  mit  $B, F$  bzw.  $T$

$T$ : "true"

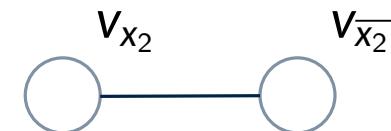
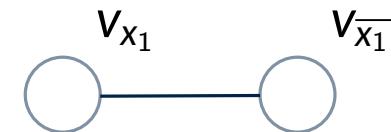
$F$ : "false"

$B$ : "base"

## 2. Gadget: Variablenbelegung

---

Für jede Variable  $x_i$  führen wir zwei adjazente **Literalknoten**  $v_{x_i}$  und  $v_{\overline{x}_i}$  ein



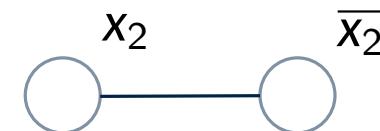
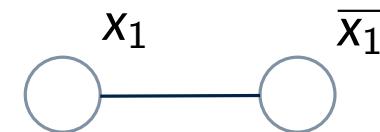
⋮



## 2. Gadget: Variablenbelegung

---

Für jede Variable  $x_i$  führen wir zwei adjazente **Literalknoten**  $v_{x_i}$  und  $v_{\bar{x}_i}$  ein

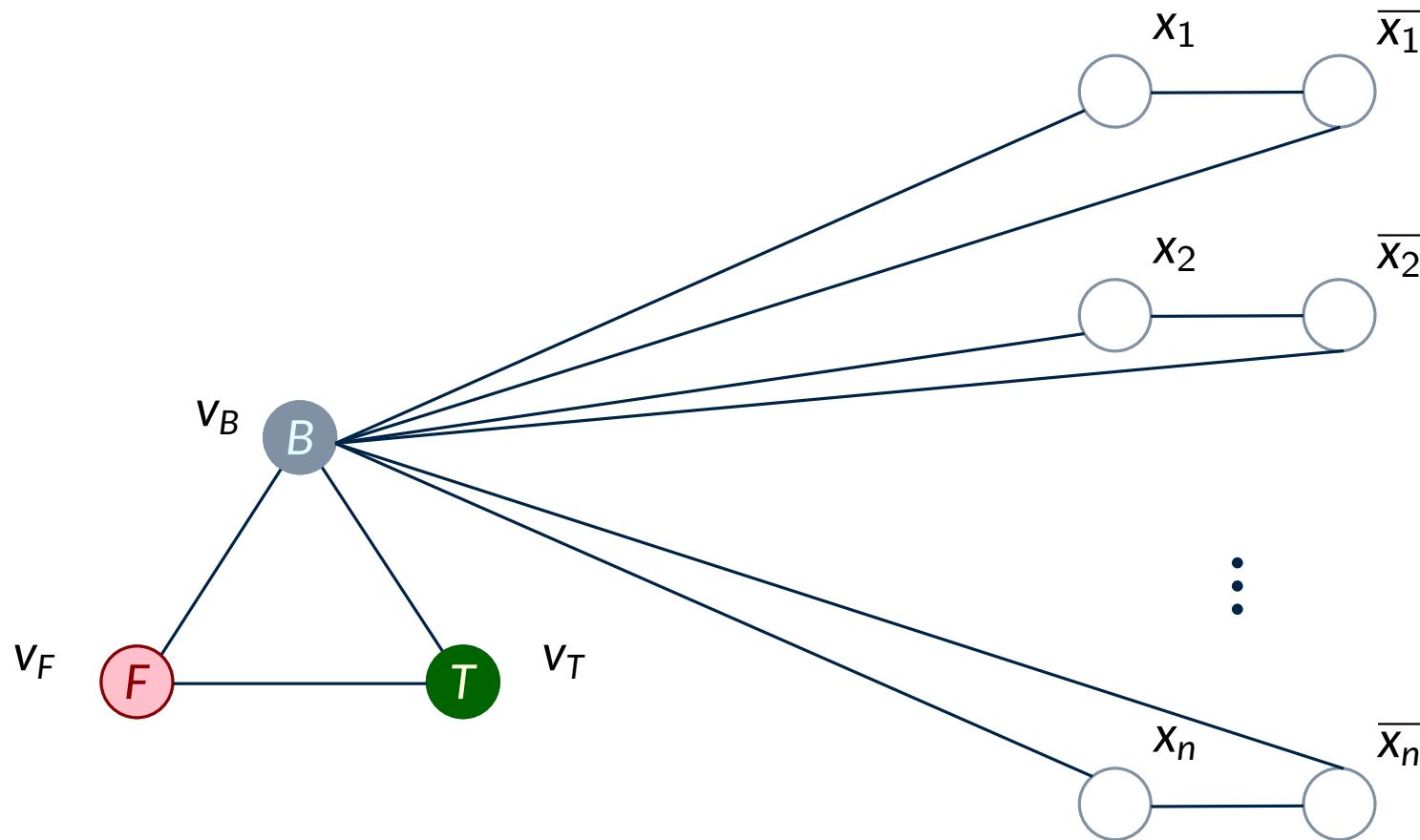


⋮



## 2. Gadget: Variablenbelegung

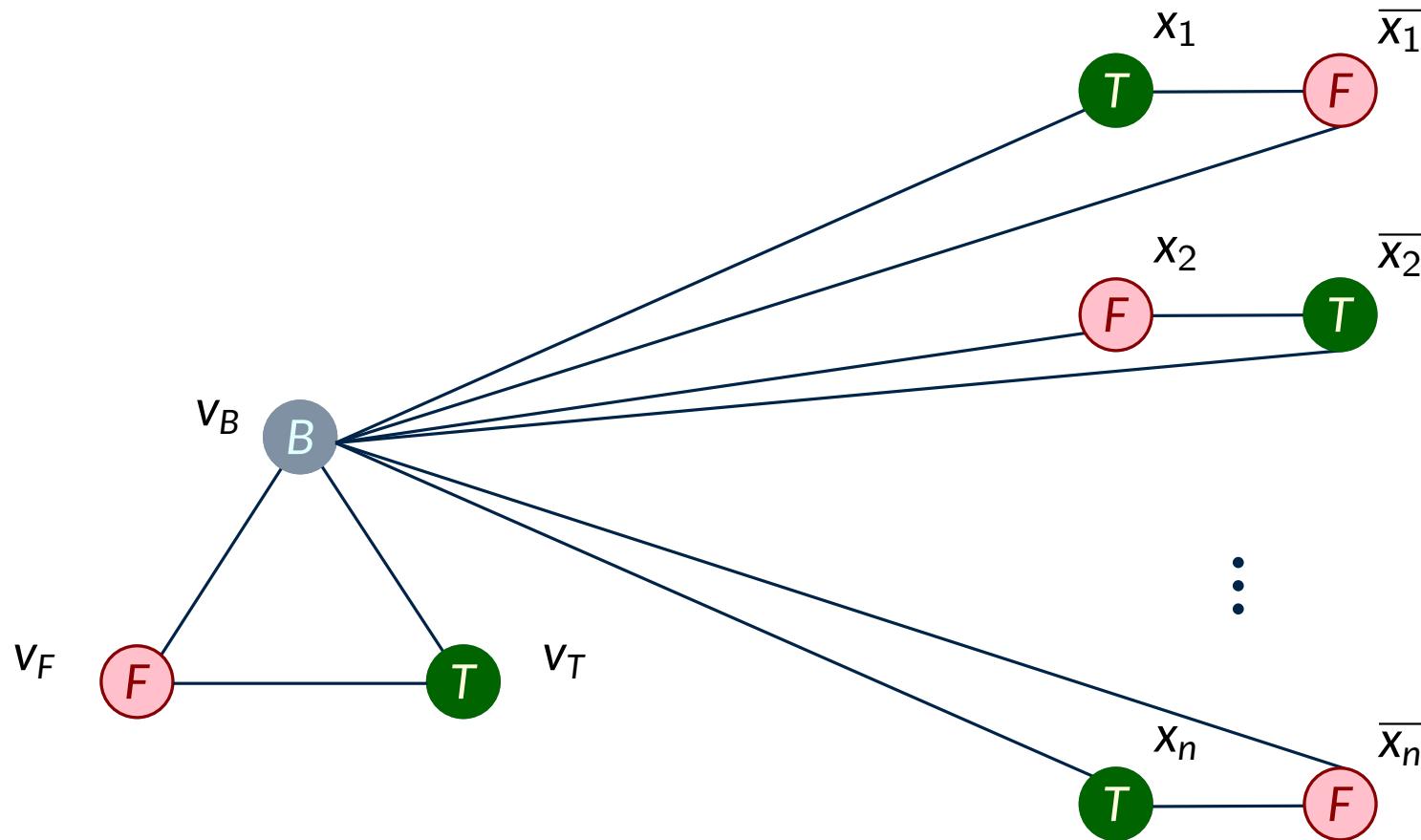
Für jede Variable  $x_i$  führen wir zwei adjazente **Literalknoten**  $v_{x_i}$  und  $v_{\bar{x}_i}$  ein  
Wir machen jeden Literalknoten  $v_\ell$  adjazent zu  $v_B$



Frage: Wie sehen gültige 3-Färbungen aus?

## 2. Gadget: Variablenbelegung

Für jede Variable  $x_i$  führen wir zwei adjazente **Literalknoten**  $v_{x_i}$  und  $v_{\bar{x}_i}$  ein  
Wir machen jeden Literalknoten  $v_\ell$  adjazent zu  $v_B$



Frage: Wie sehen gültige 3-Färbungen aus?

Eigenschaft: Jede gültige 3-Färbung färbt  $x_i$  mit **T** und  $\bar{x}_i$  mit **F** oder umgekehrt  
→ kann als **Variablenbelegung** interpretiert werden

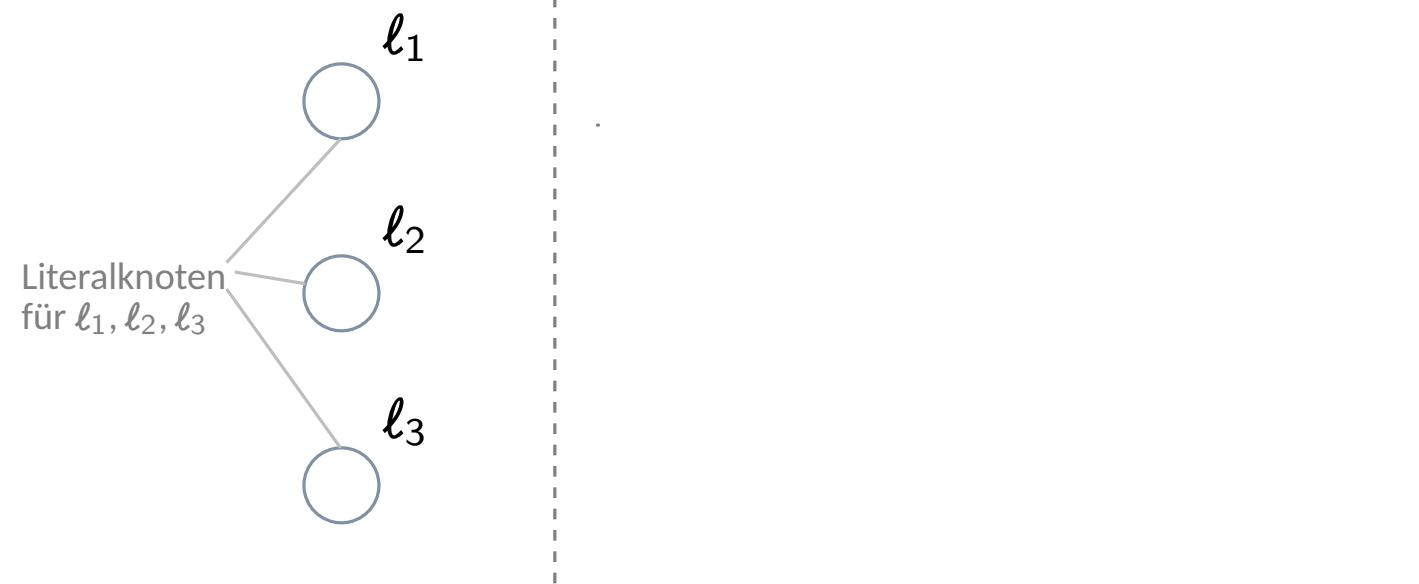
# 3. Gadget: Klauselgadget

---

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

**Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$**

"Input" des Gadgets



# 3. Gadget: Klauselgadget

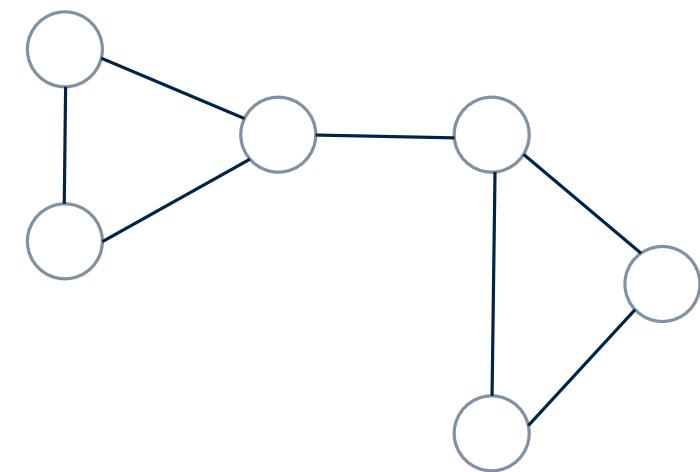
Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



neu eingeführte Klauselknoten:

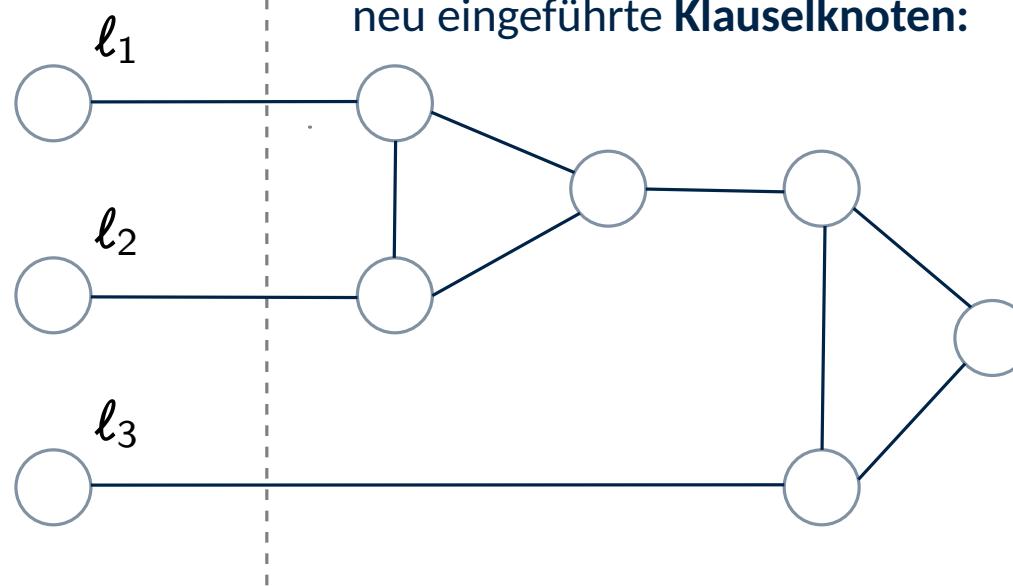


# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets

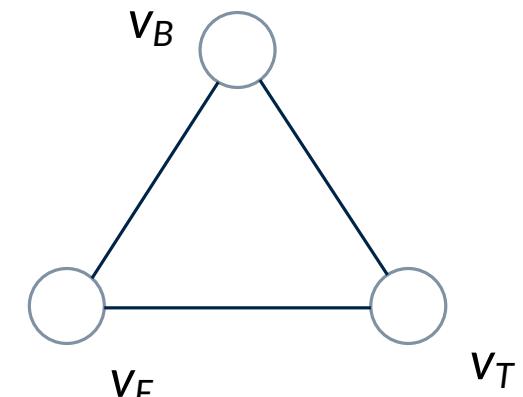
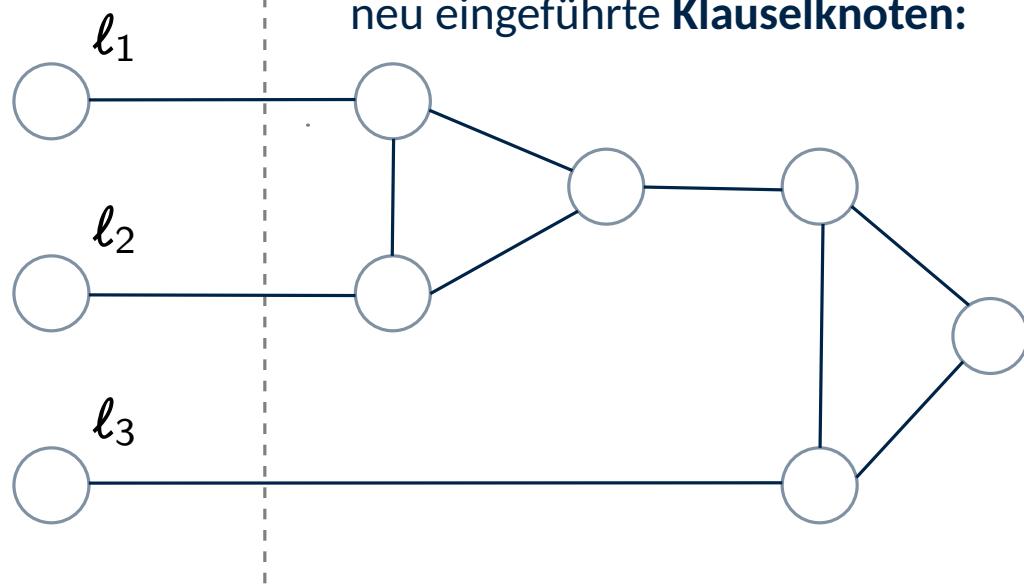


# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets

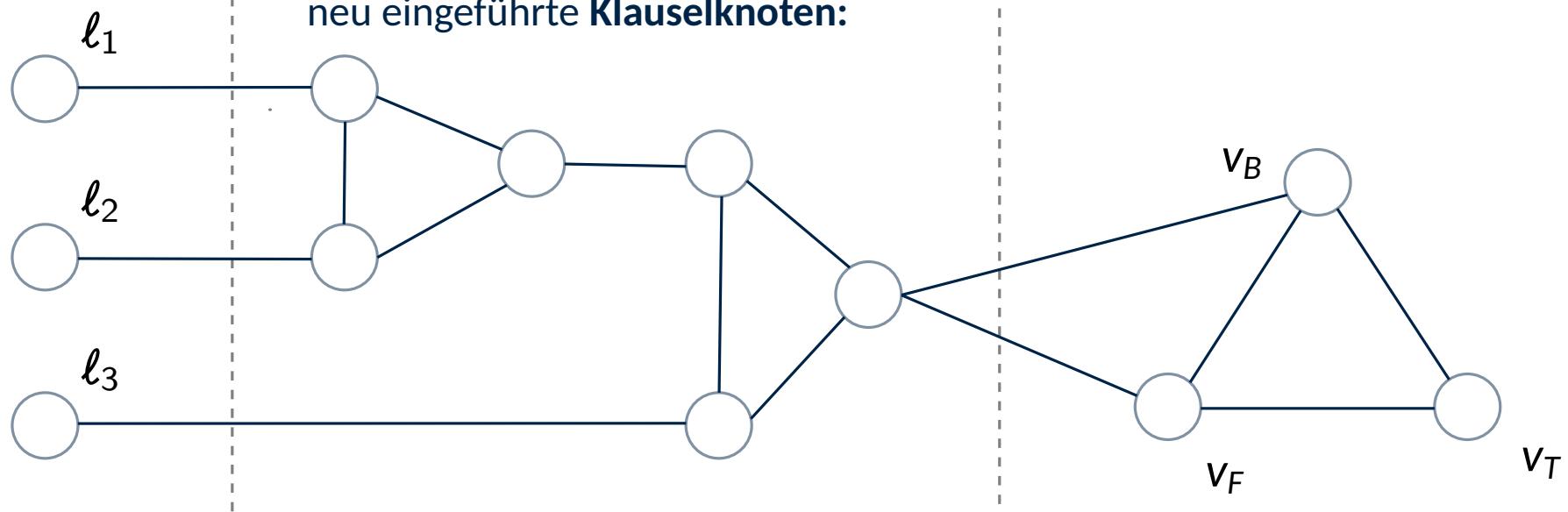


# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets

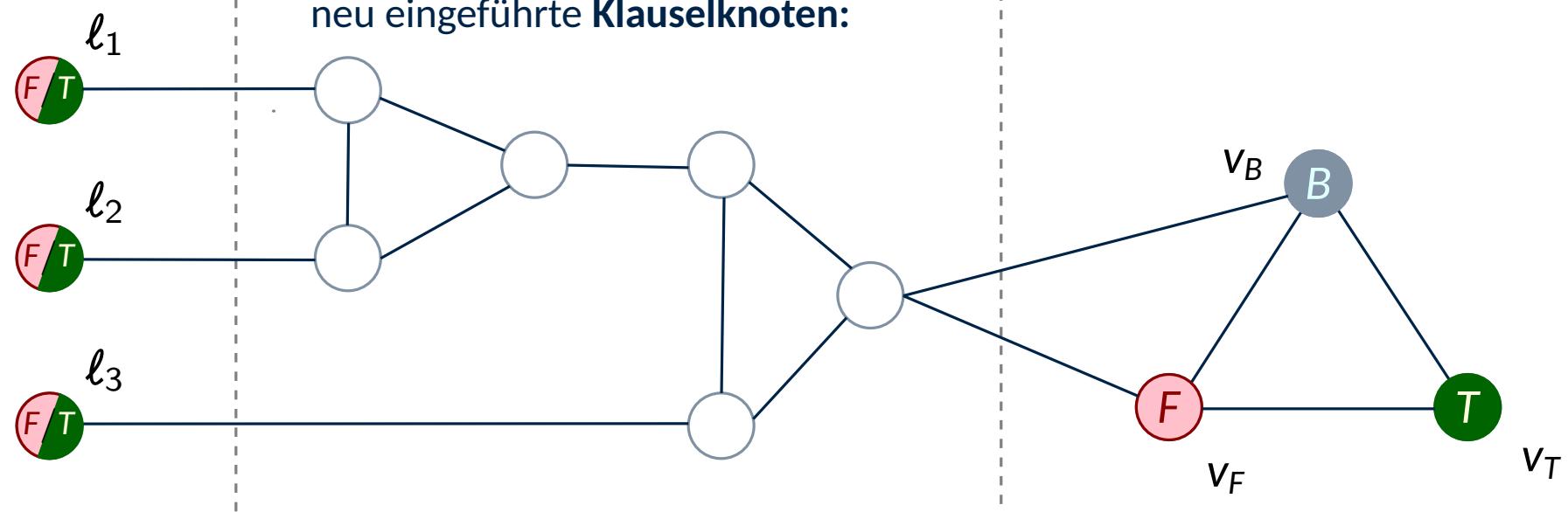


# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets

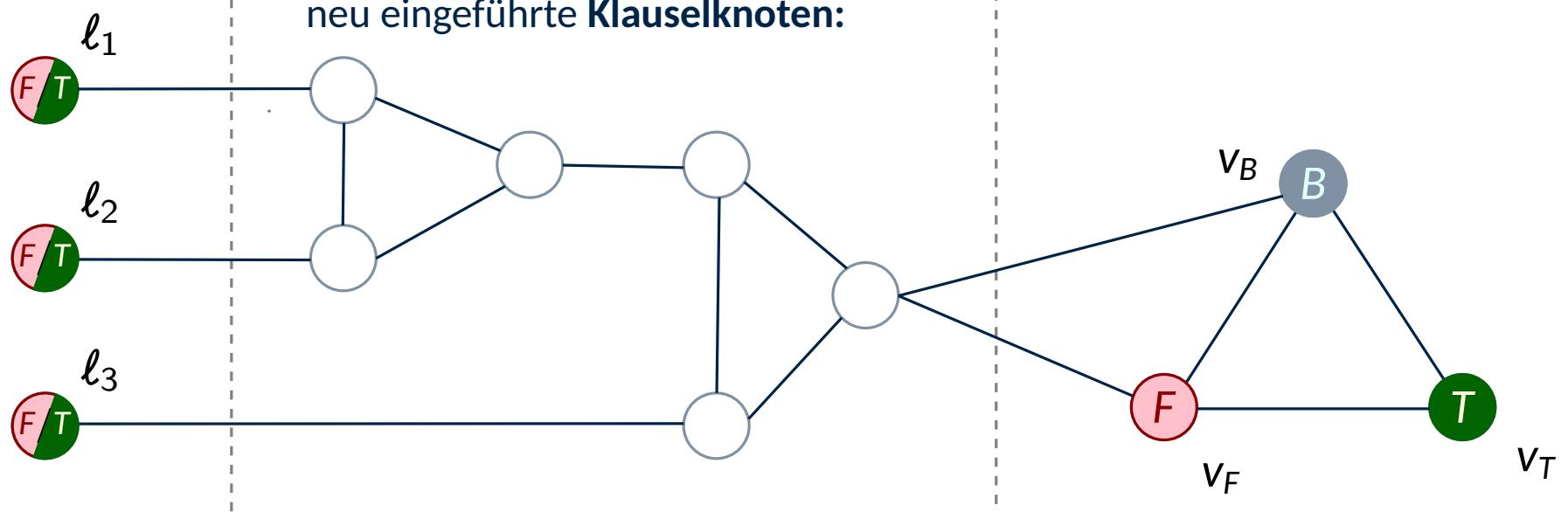


# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

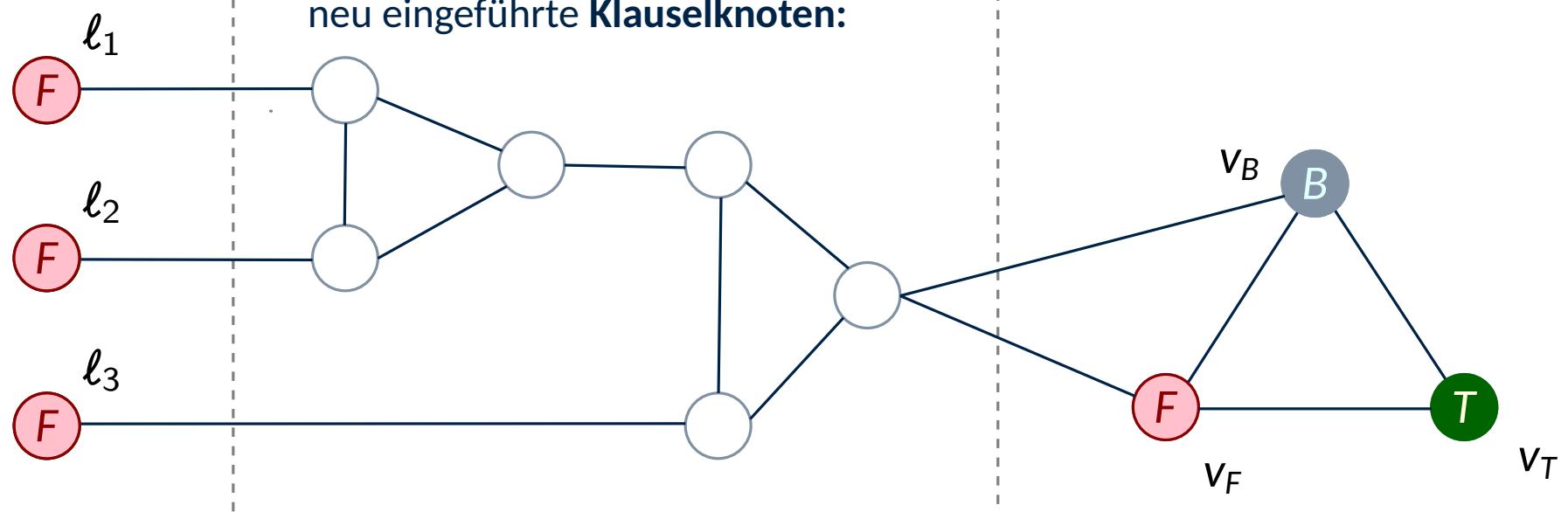
Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

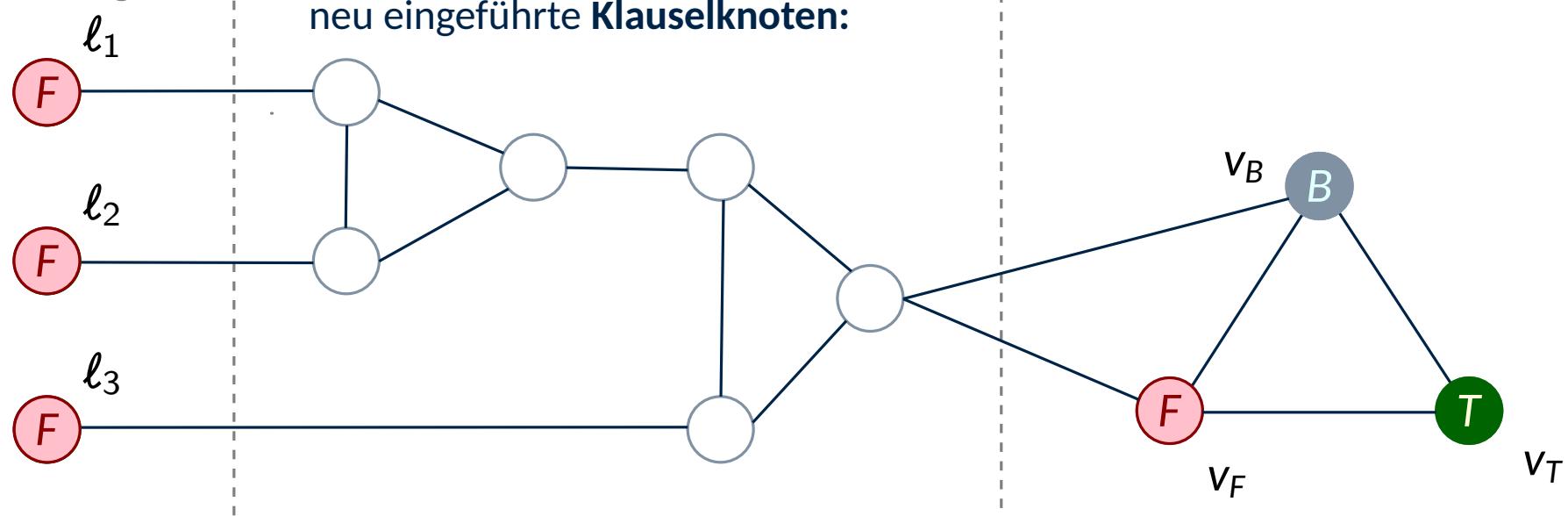
**Fall 1:** alle drei sind  $F$  gefärbt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

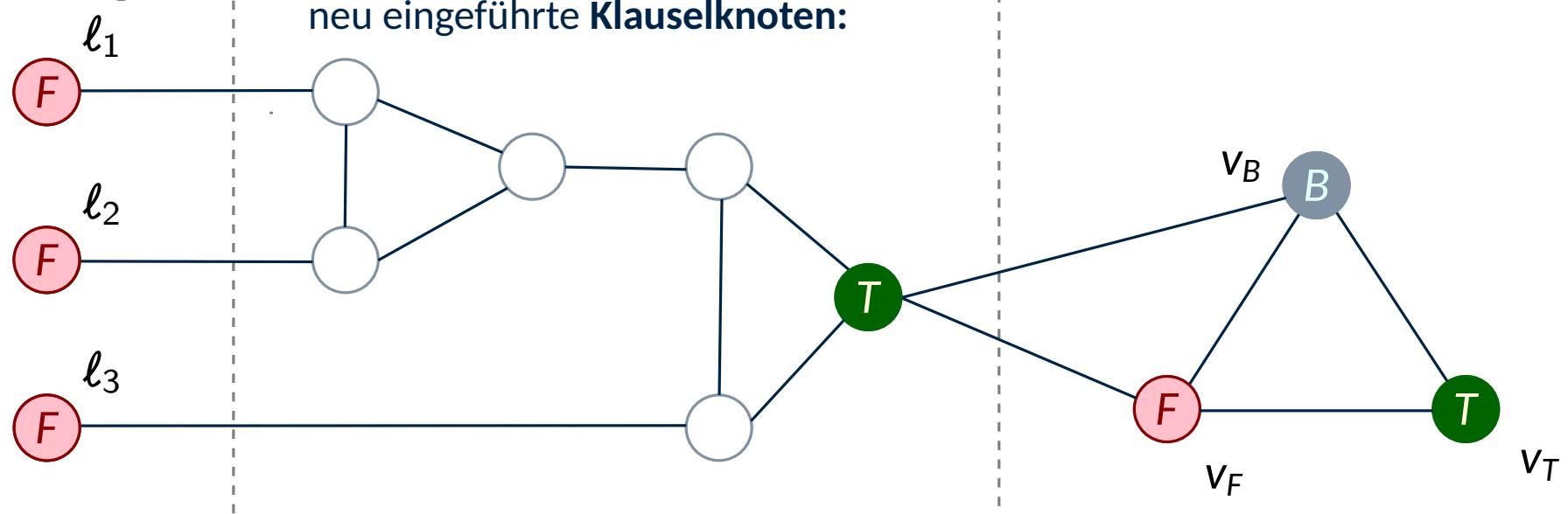
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

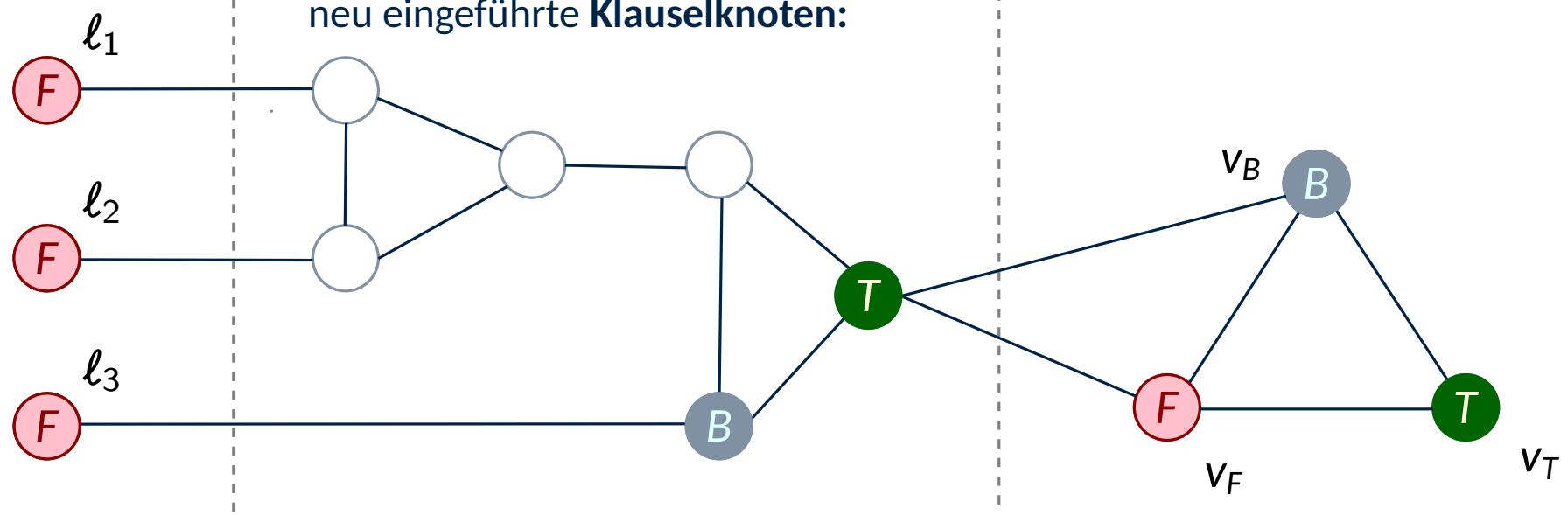
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

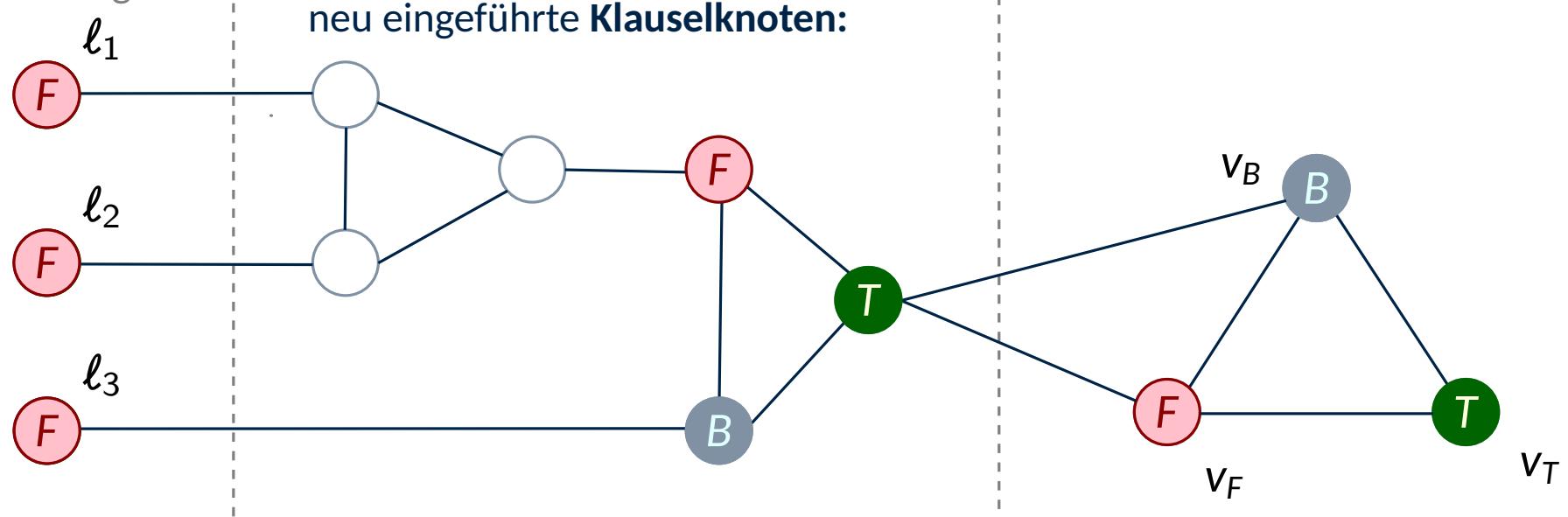
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

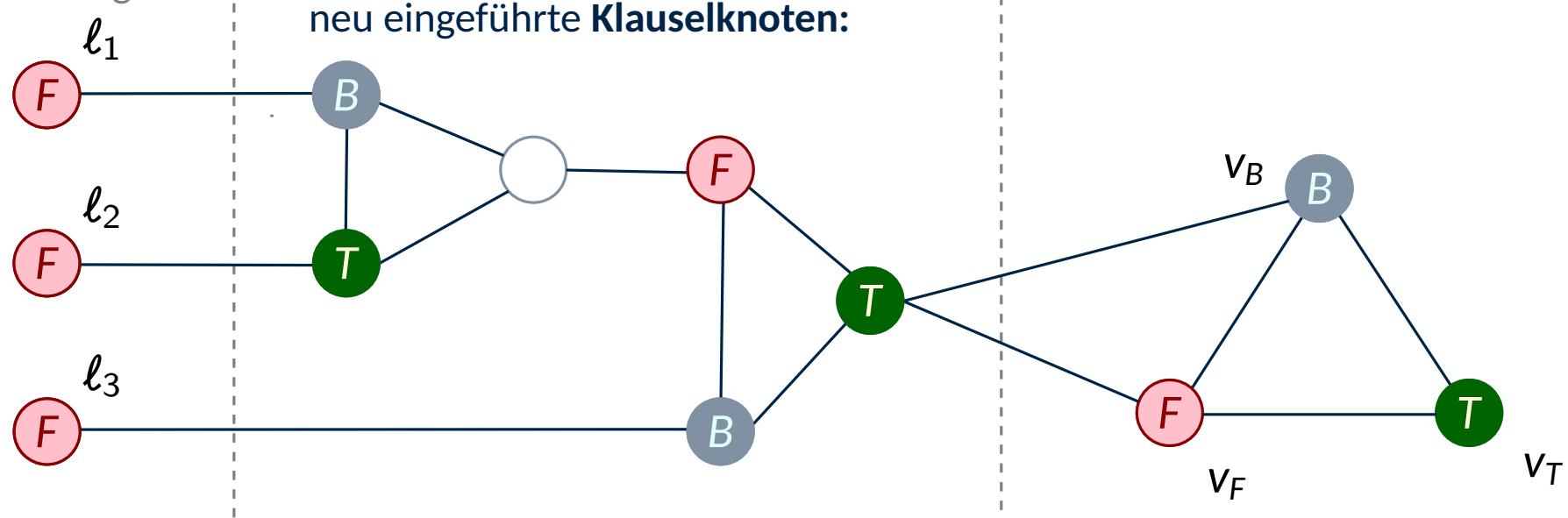
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

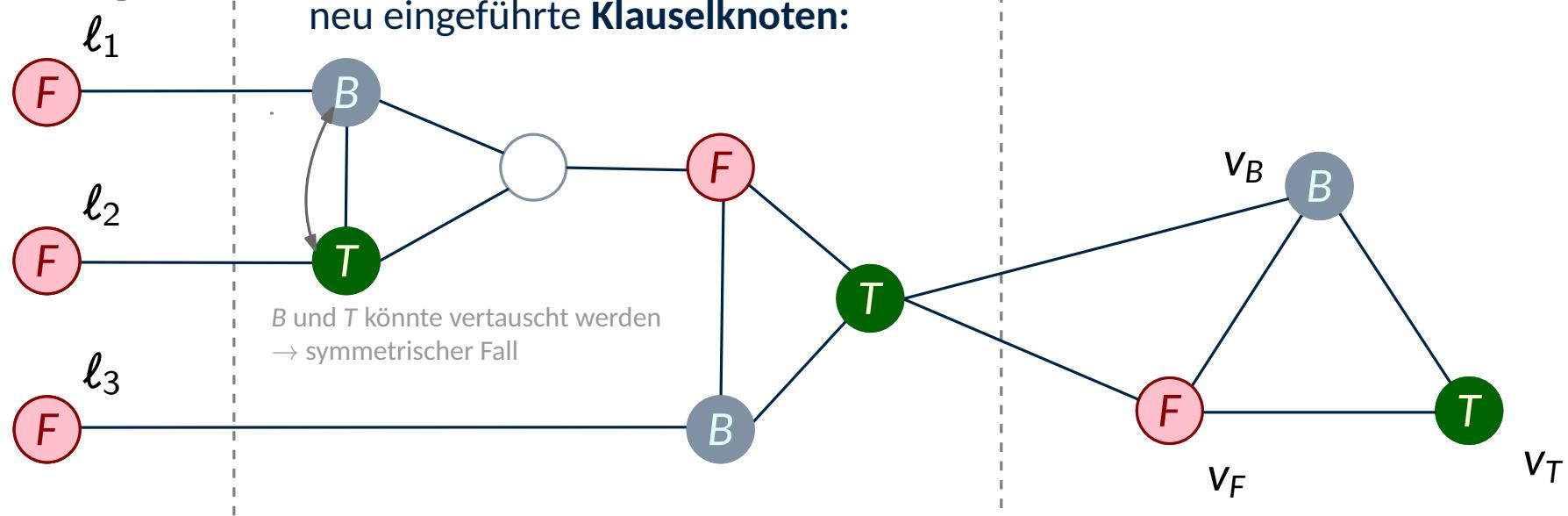
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

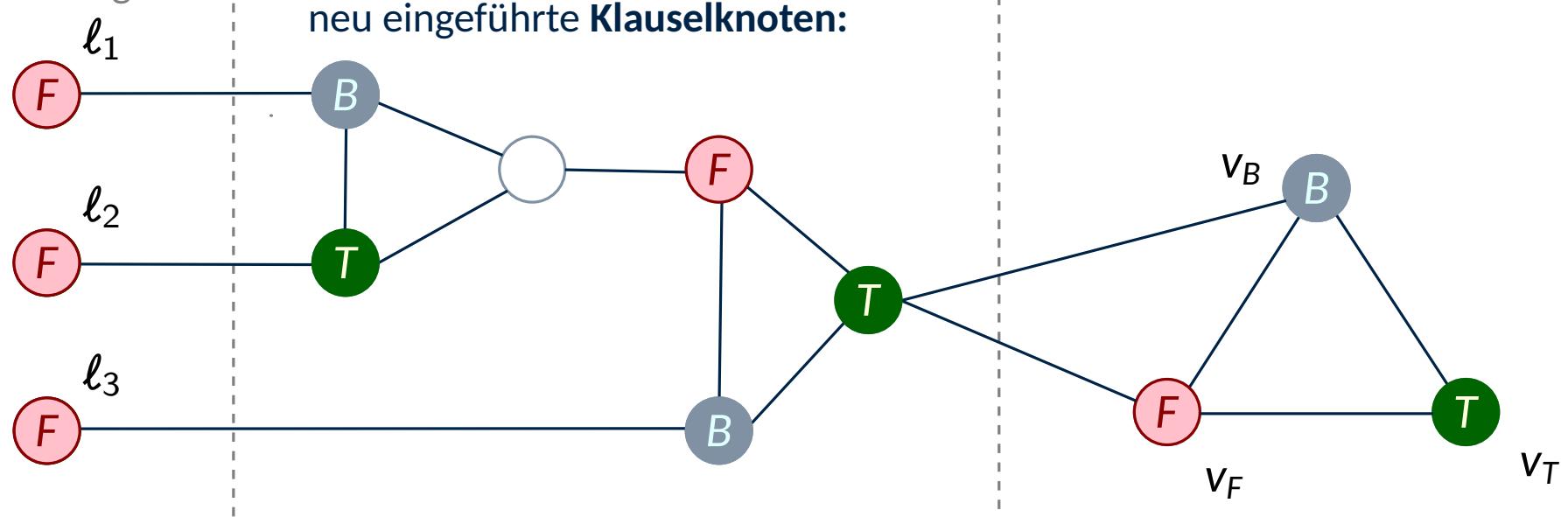
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

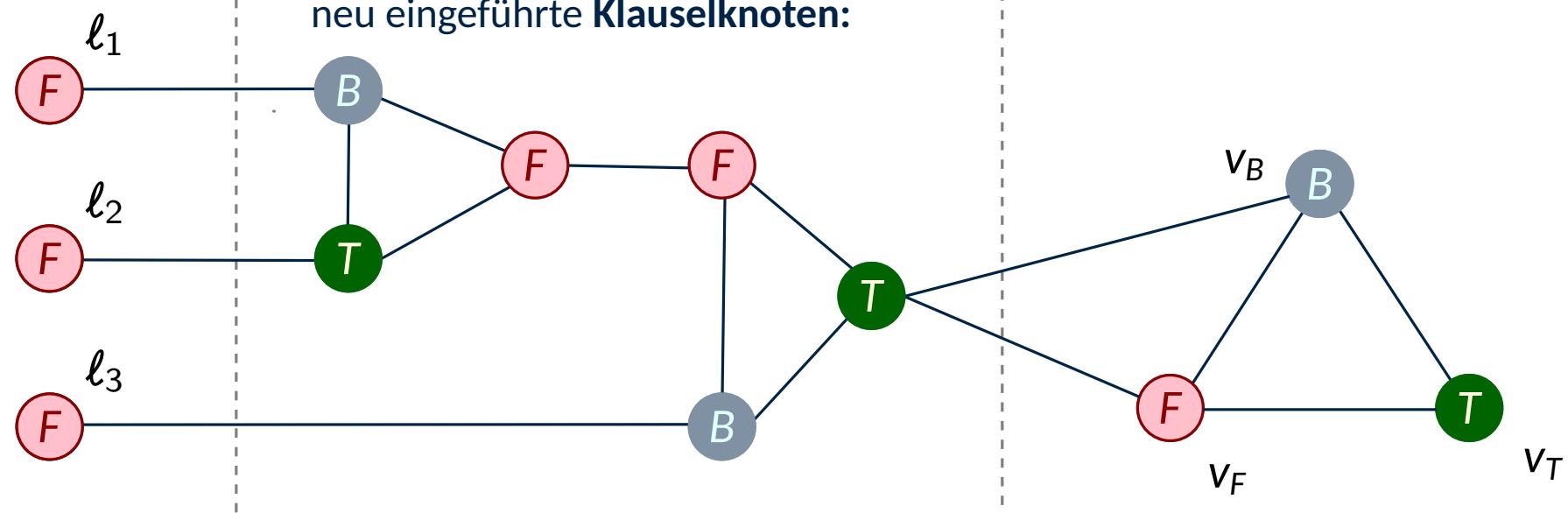
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

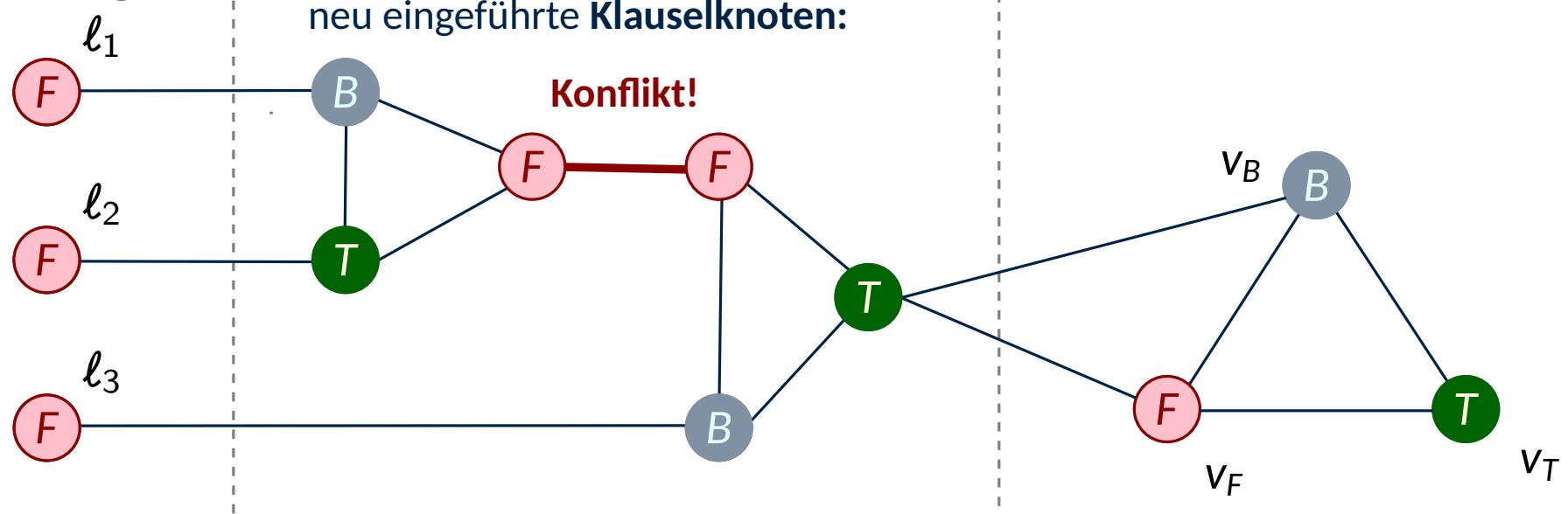
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}$ ,  $v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

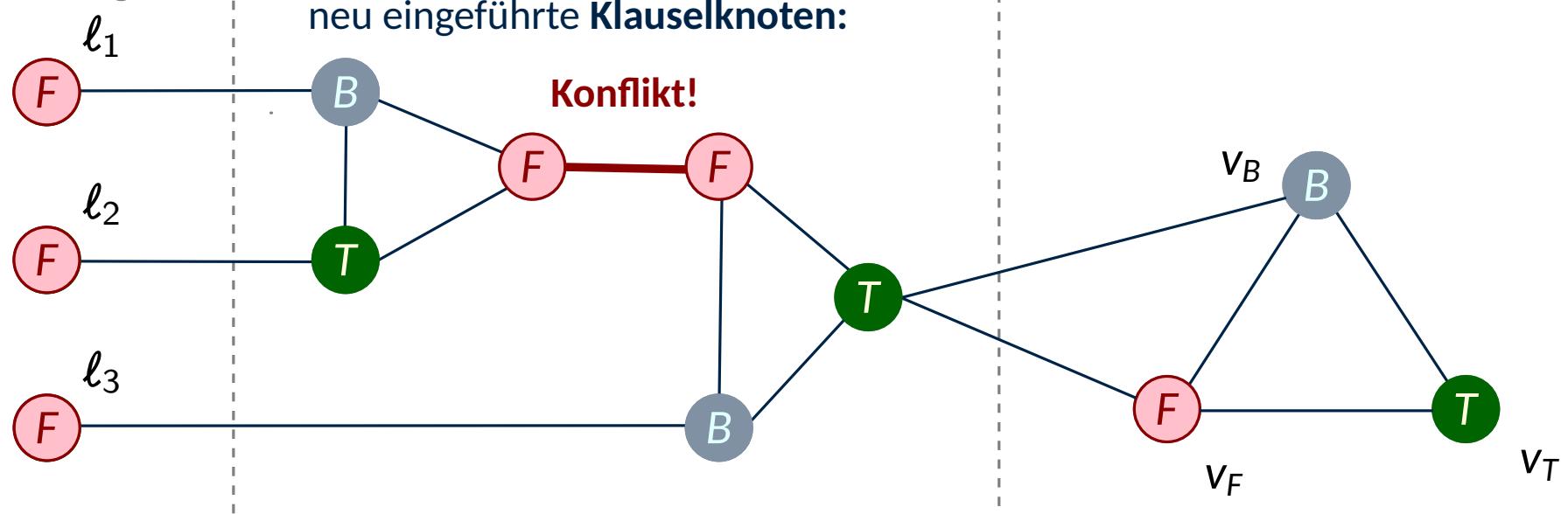
**Fall 1:** alle drei sind  $F$  gefärbt → wir färben Knoten ein, wenn es nur eine gültige Möglichkeit gibt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



**Eigenschaft:**  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

**Beweis:** Man betrachte alle Fälle

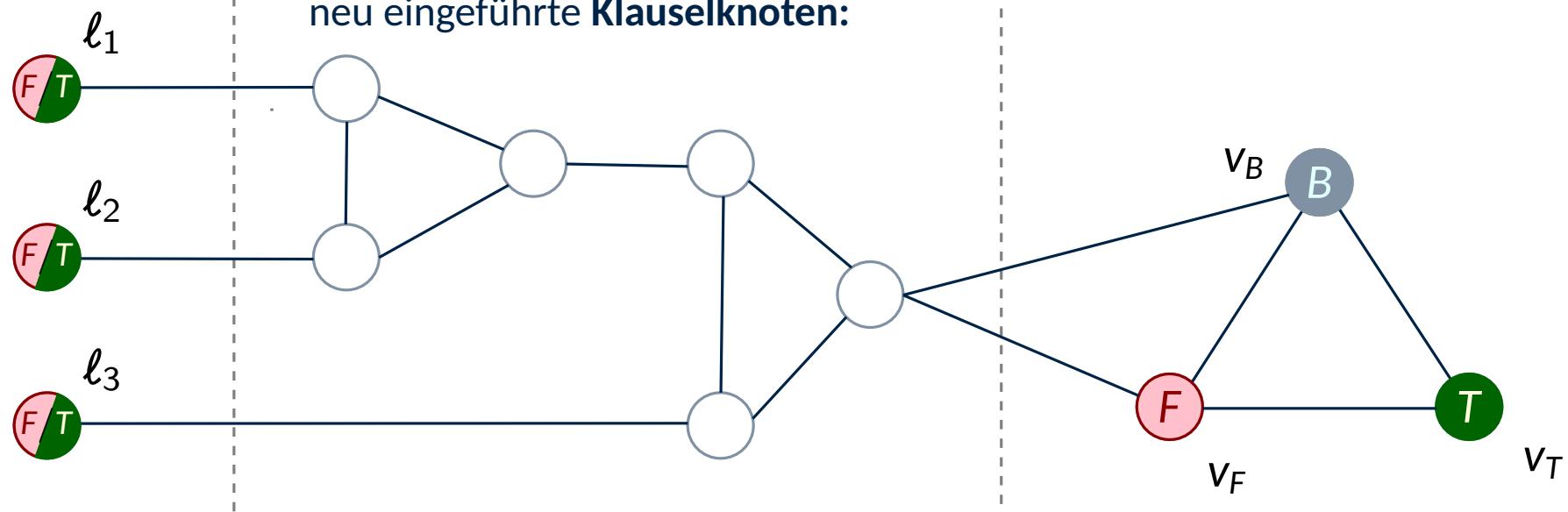
**Fall 1:** alle drei sind  $F$  gefärbt → keine gültige Färbung möglich!

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



Eigenschaft:  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist **T** gefärbt

Beweis: Man betrachte alle Fälle

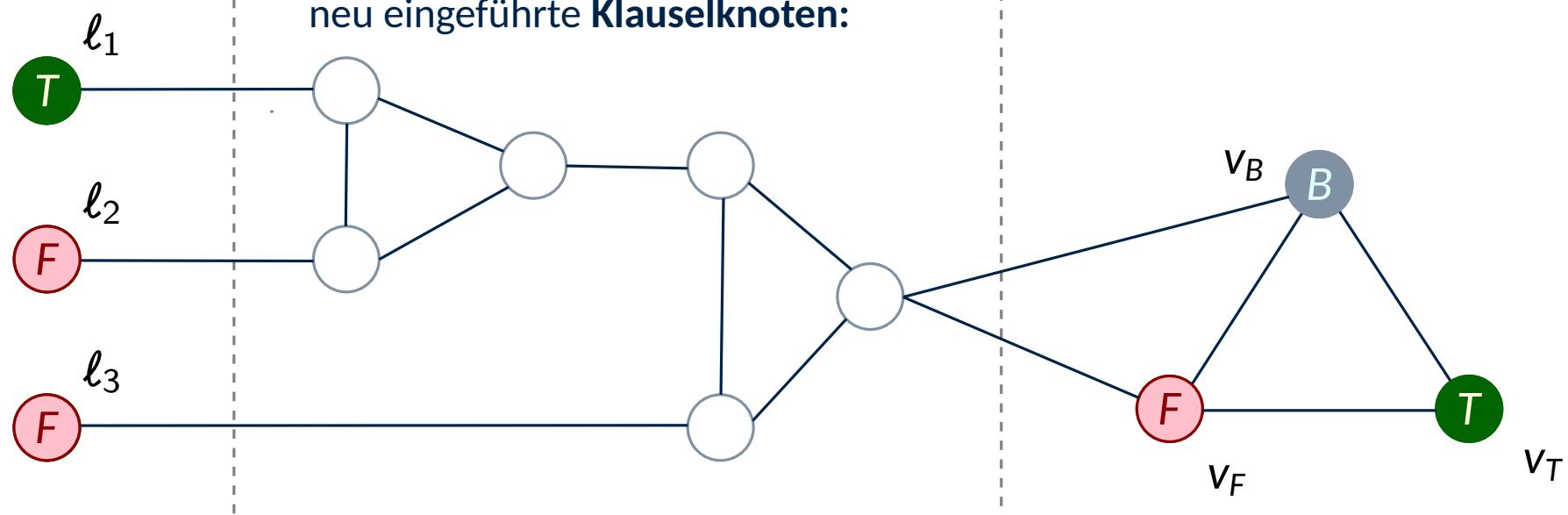
Fall 1: alle drei sind **F** gefärbt → keine gültige Färbung möglich!  
Fall 2:  $v_{\ell_1}$  ist alleinig **T** gefärbt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



Eigenschaft:  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist  $T$  gefärbt

Beweis: Man betrachte alle Fälle

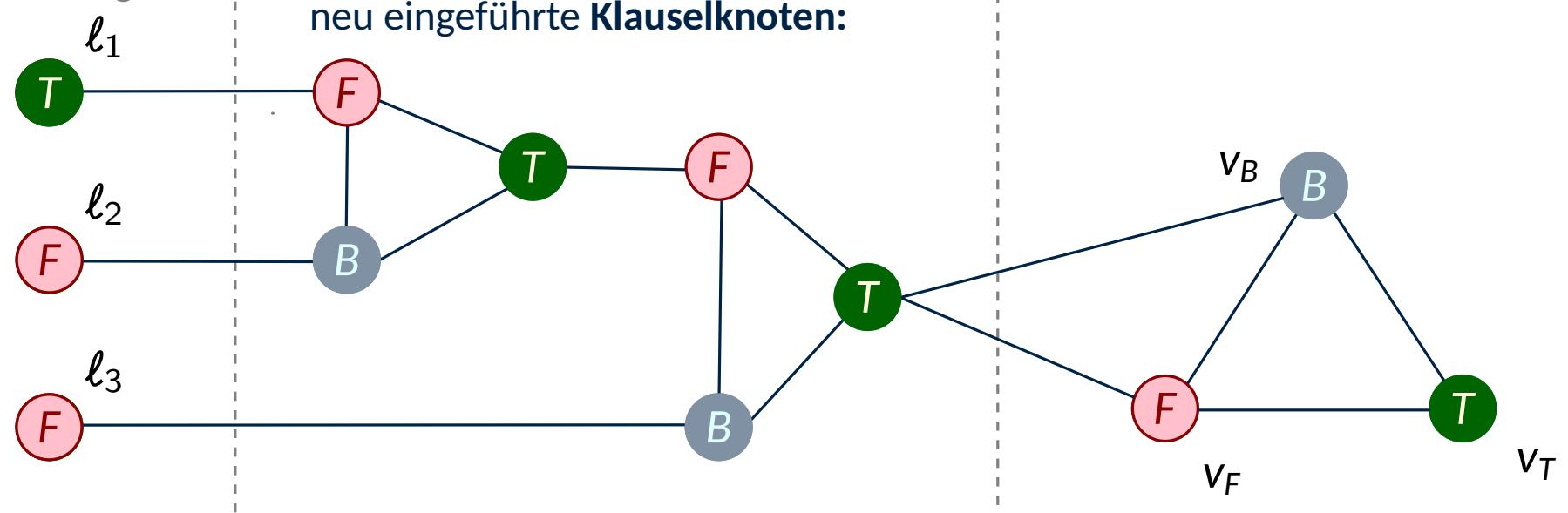
- Fall 1: alle drei sind  $F$  gefärbt → keine gültige Färbung möglich!
- Fall 2:  $v_{\ell_1}$  ist alleinig  $T$  gefärbt

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



Eigenschaft:  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist **T** gefärbt

Beweis: Man betrachte alle Fälle

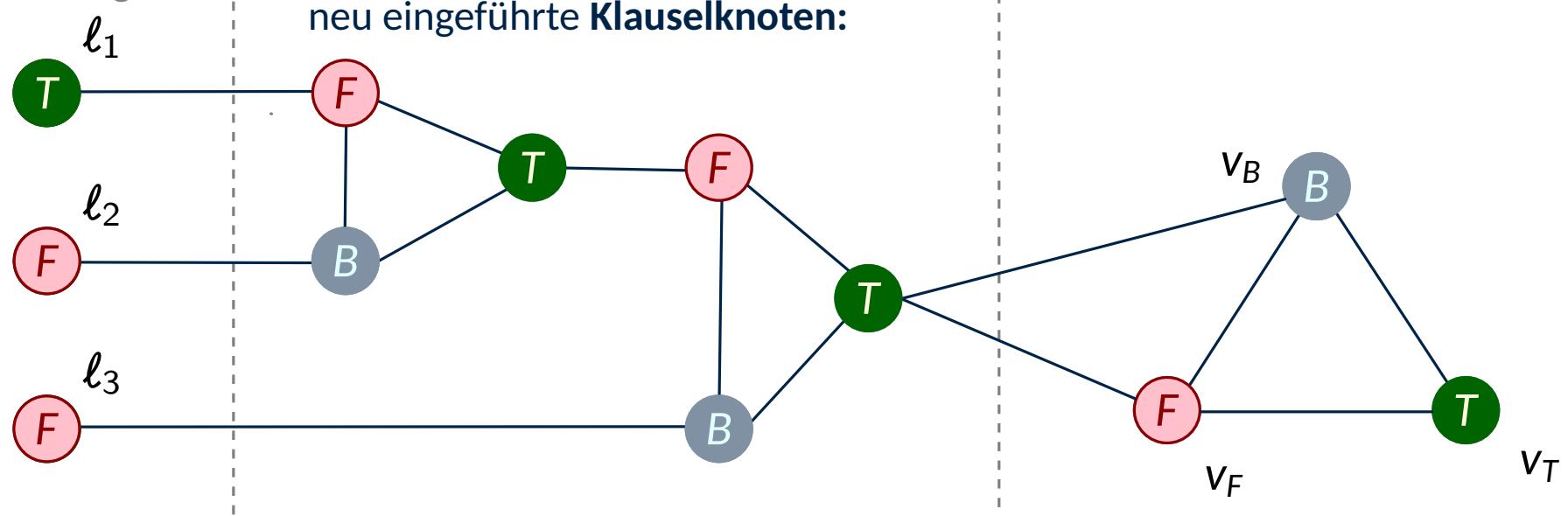
- Fall 1: alle drei sind **F** gefärbt → keine gültige Färbung möglich!  
Fall 2:  $v_{\ell_1}$  ist alleinig **T** gefärbt → gültige Färbung möglich!

# 3. Gadget: Klauselgadget

Wie können wir "überprüfen", ob Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  von der gewählten Belegung erfüllt wird?

Klauselgadget für  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$

"Input" des Gadgets



Eigenschaft:  
Es gilt:

Die Klauselknoten für  $C = (\ell_1 \vee \ell_2 \vee \ell_3)$  können gültig 3-gefärbt werden genau dann, wenn mindestens einer von  $v_{\ell_1}, v_{\ell_2}$  und  $v_{\ell_3}$  ist **T** gefärbt

Beweis: Man betrachte alle Fälle

Fall 1: alle drei sind **F** gefärbt → keine gültige Färbung möglich!

Fall 2:  $v_{\ell_1}$  ist alleinig **T** gefärbt → gültige Färbung möglich!

# Gesamte Konstruktion

---

**Reduktion:**

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

**Beispiel:**  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$

# Gesamte Konstruktion

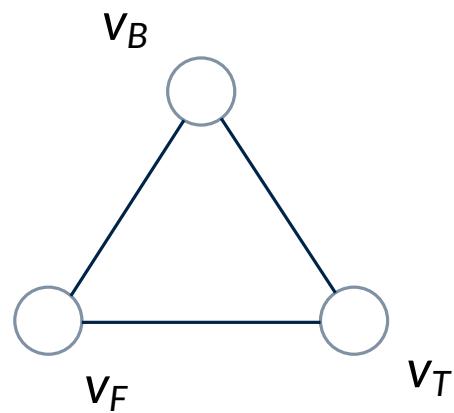
---

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein

**Beispiel:**  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$



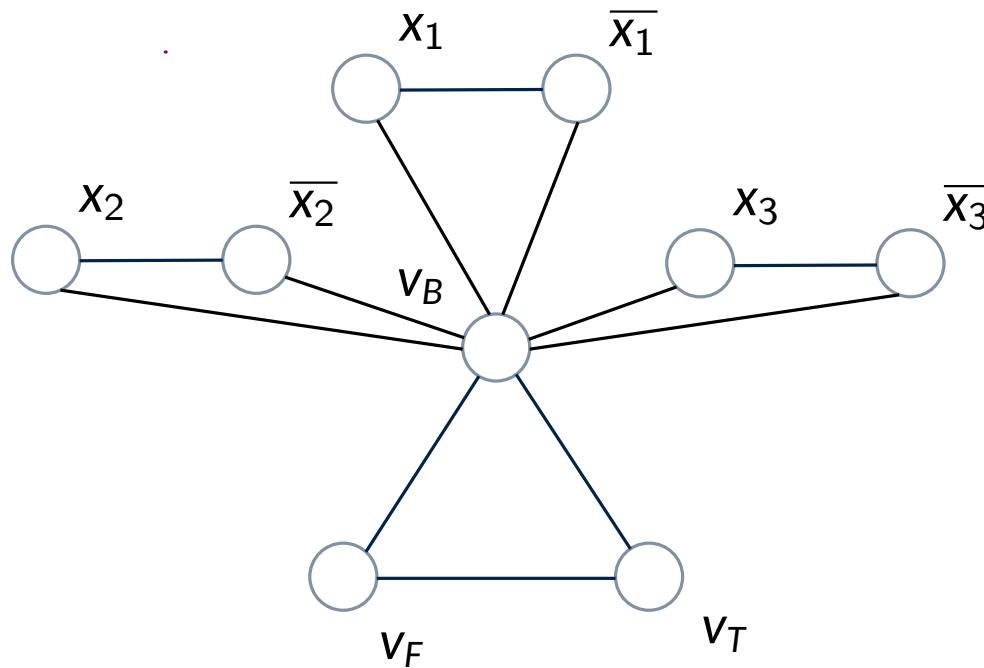
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



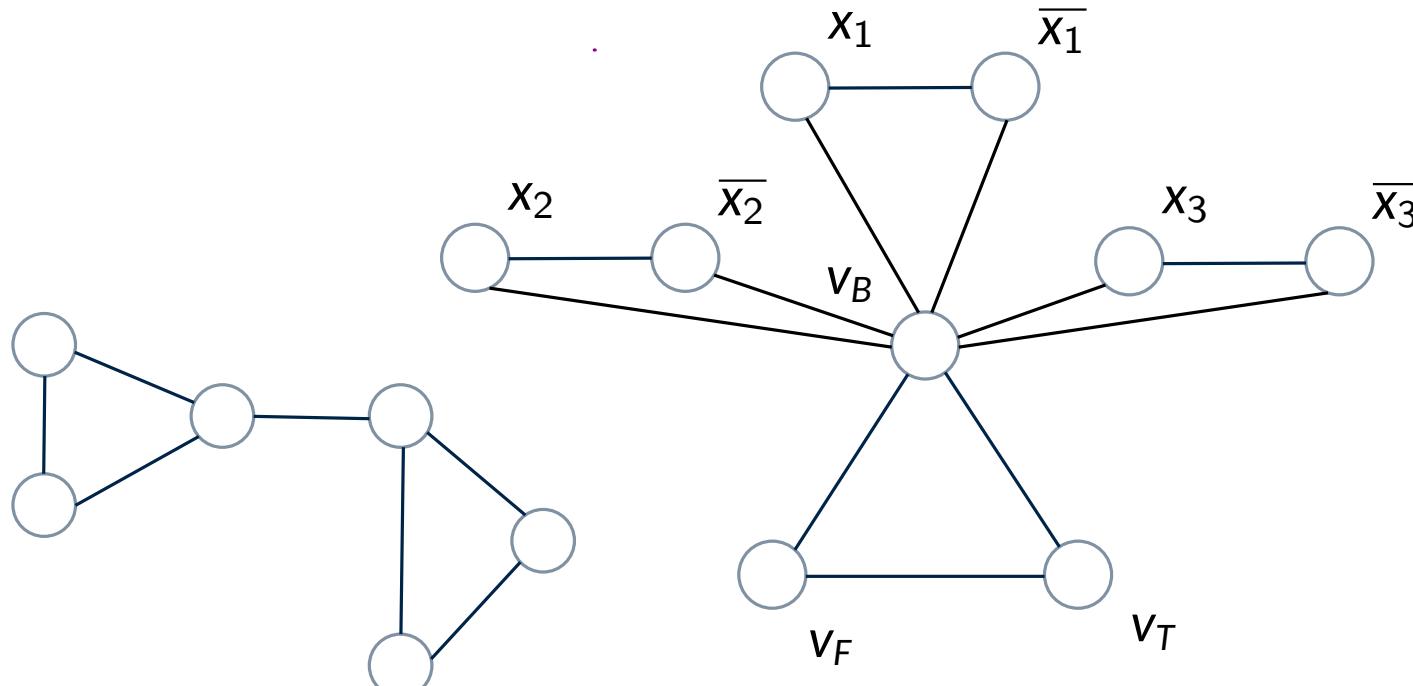
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



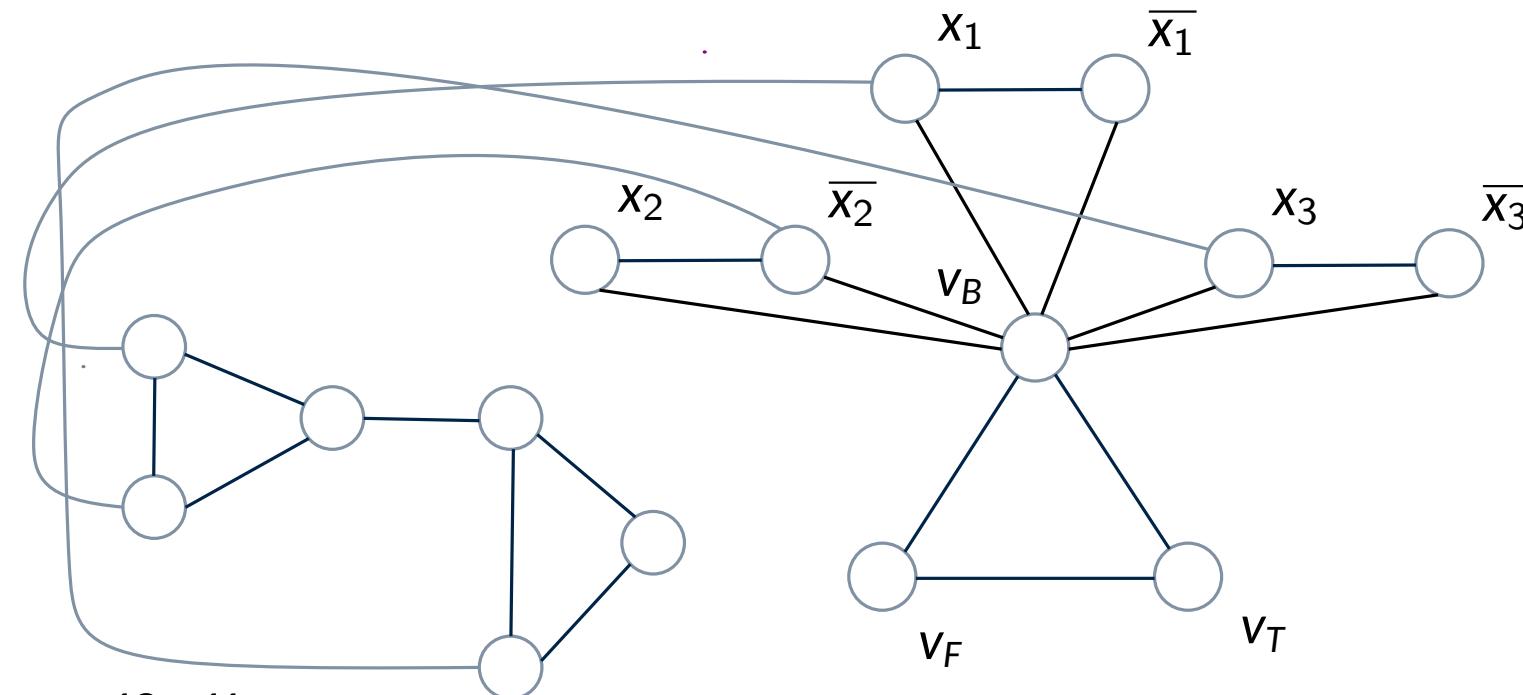
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein sowie entsprechende Knoten zu  $v_{\ell_1}, v_{\ell_2}, v_{\ell_3}$  sowie  $v_F$  und  $v_B$

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



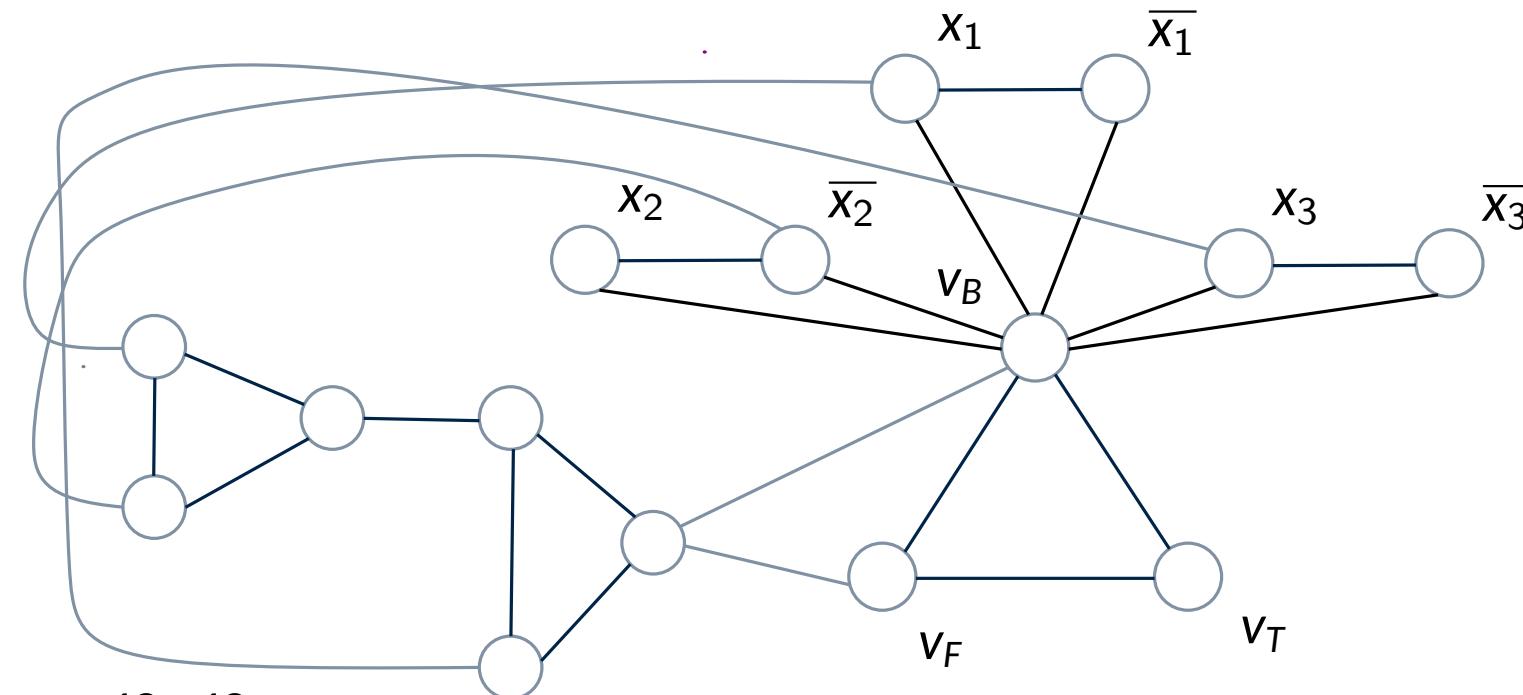
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein sowie entsprechende Knoten zu  $v_{\ell_1}, v_{\ell_2}, v_{\ell_3}$  sowie  $v_F$  und  $v_B$

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



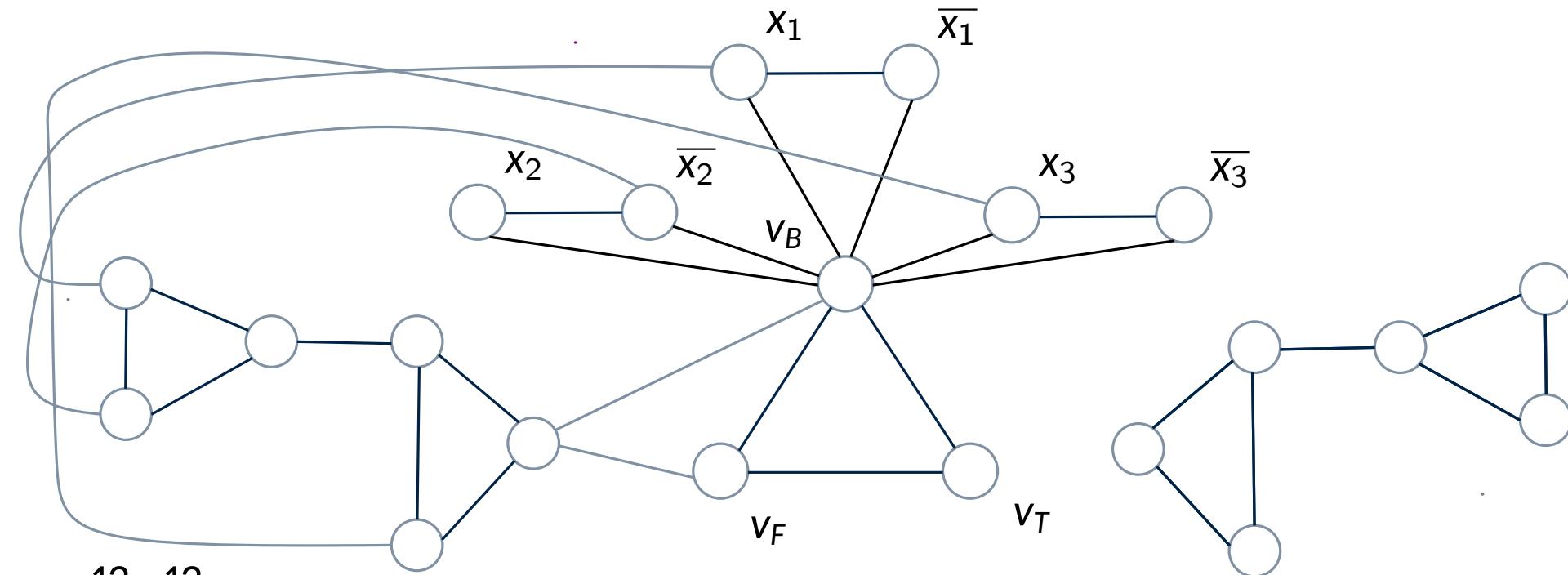
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein sowie entsprechende Knoten zu  $v_{\ell_1}, v_{\ell_2}, v_{\ell_3}$  sowie  $v_F$  und  $v_B$

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



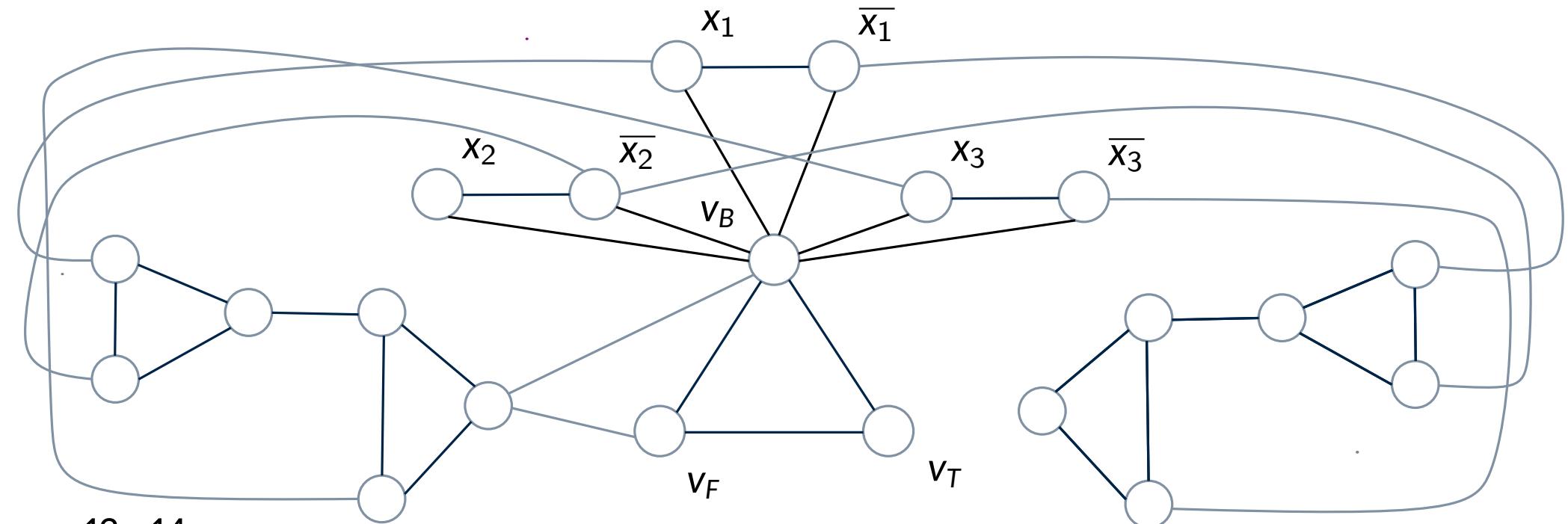
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein sowie entsprechende Knoten zu  $v_{\ell_1}, v_{\ell_2}, v_{\ell_3}$  sowie  $v_F$  und  $v_B$

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



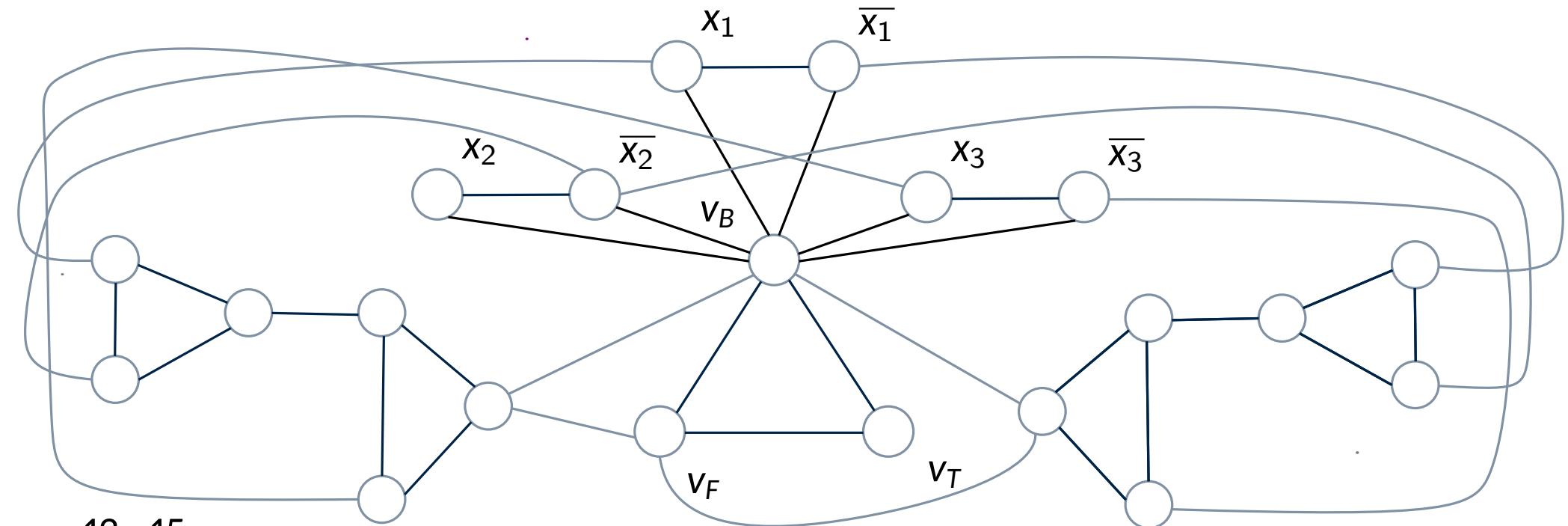
# Gesamte Konstruktion

## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein sowie entsprechende Knoten zu  $v_{\ell_1}, v_{\ell_2}, v_{\ell_3}$  sowie  $v_F$  und  $v_B$

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$



# Gesamte Konstruktion

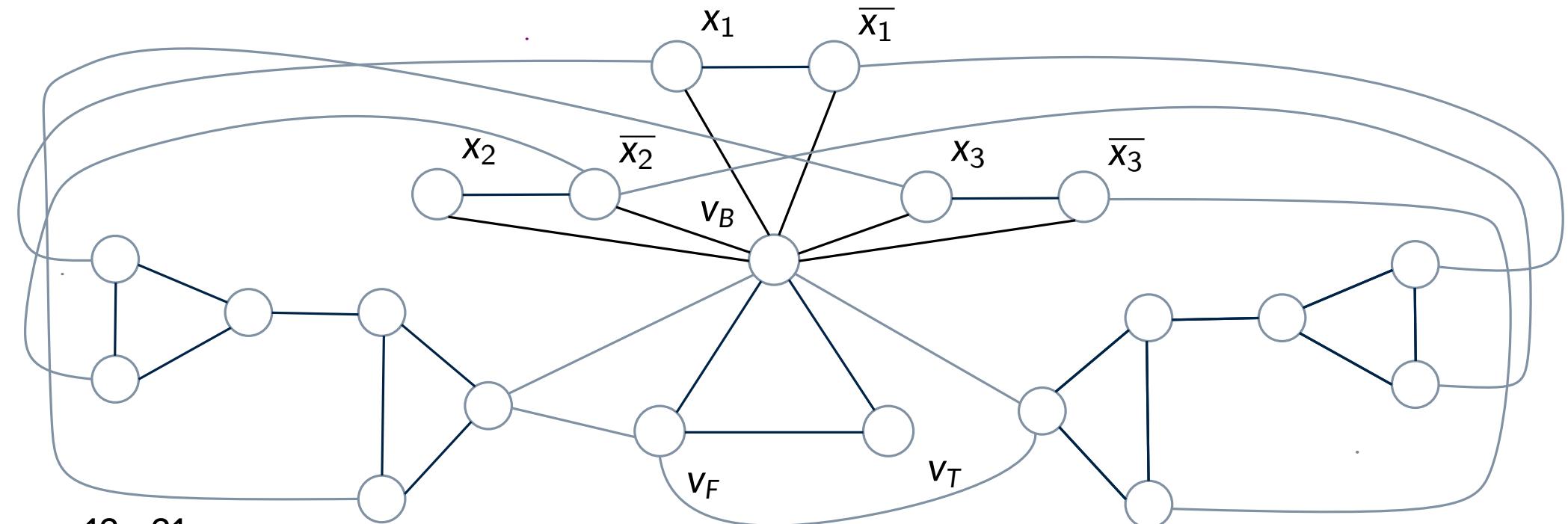
## Reduktion:

Gegeben eine 3-KNF-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ , erzeugen den Graphen wie beschrieben:

- wir führen das Dreieck  $v_B, v_F, v_T$  ein
- wir führen die Literalknoten  $v_{x_1}, v_{\bar{x}_1}, \dots, v_{x_n}, v_{\bar{x}_n}$  mit ihren entsprechenden Kanten ein
- für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  führen wir neue Klauselknoten ein sowie entsprechende Knoten zu  $v_{\ell_1}, v_{\ell_2}, v_{\ell_3}$  sowie  $v_F$  und  $v_B$

**Laufzeit:** Anzahl Knoten des Graphen ist:  $3 + 2n + 6m = O(n + m)$   
Anzahl Kanten also  $O((n + m)^2)$   
G kann in Polynomialzeit konstruiert werden.

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

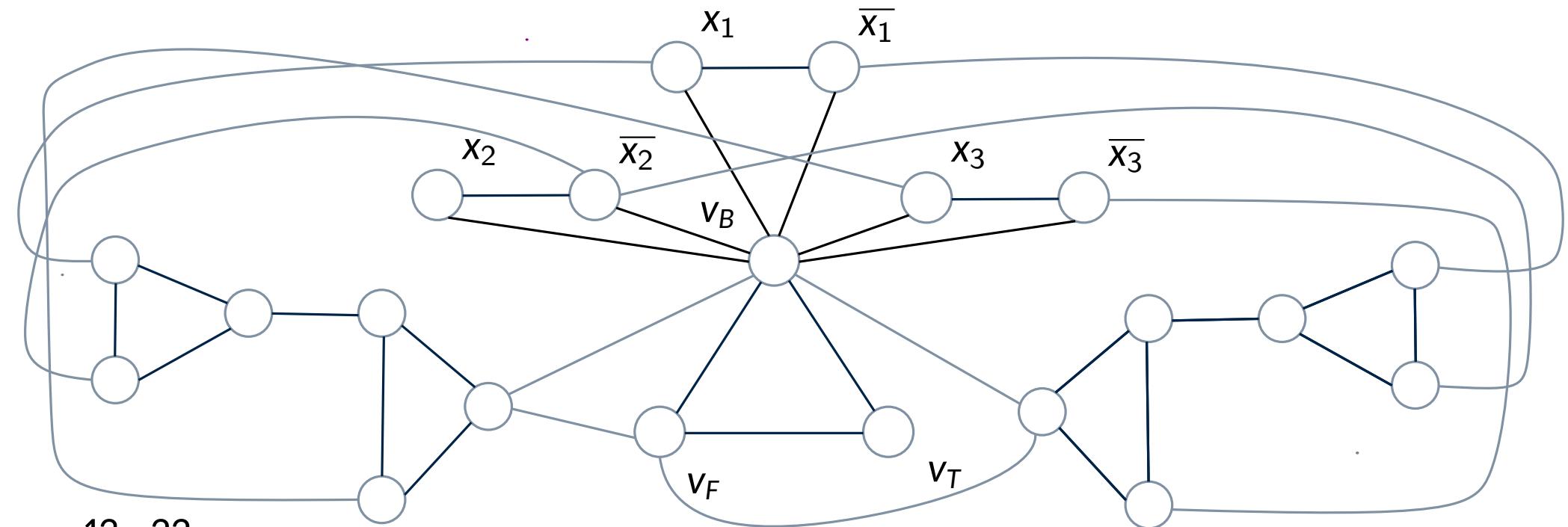


# Gesamte Konstruktion

Korrektheit:

$\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

Beispiel:  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$

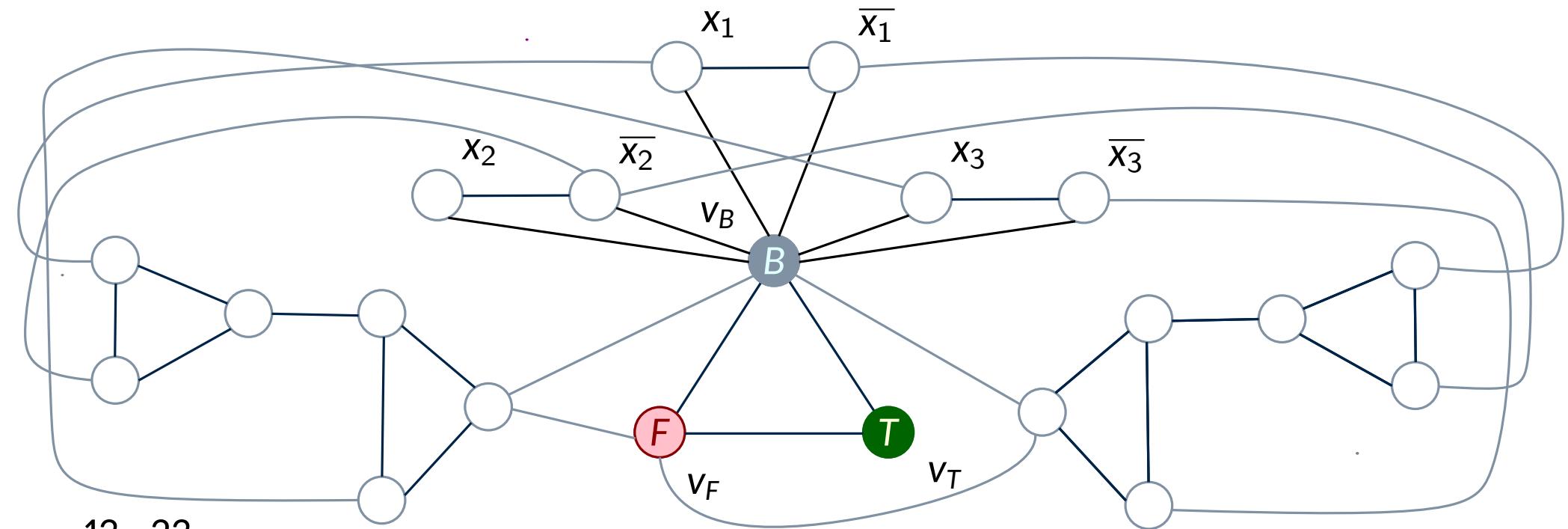


# Gesamte Konstruktion

Korrektheit:

$\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

Beispiel:  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$



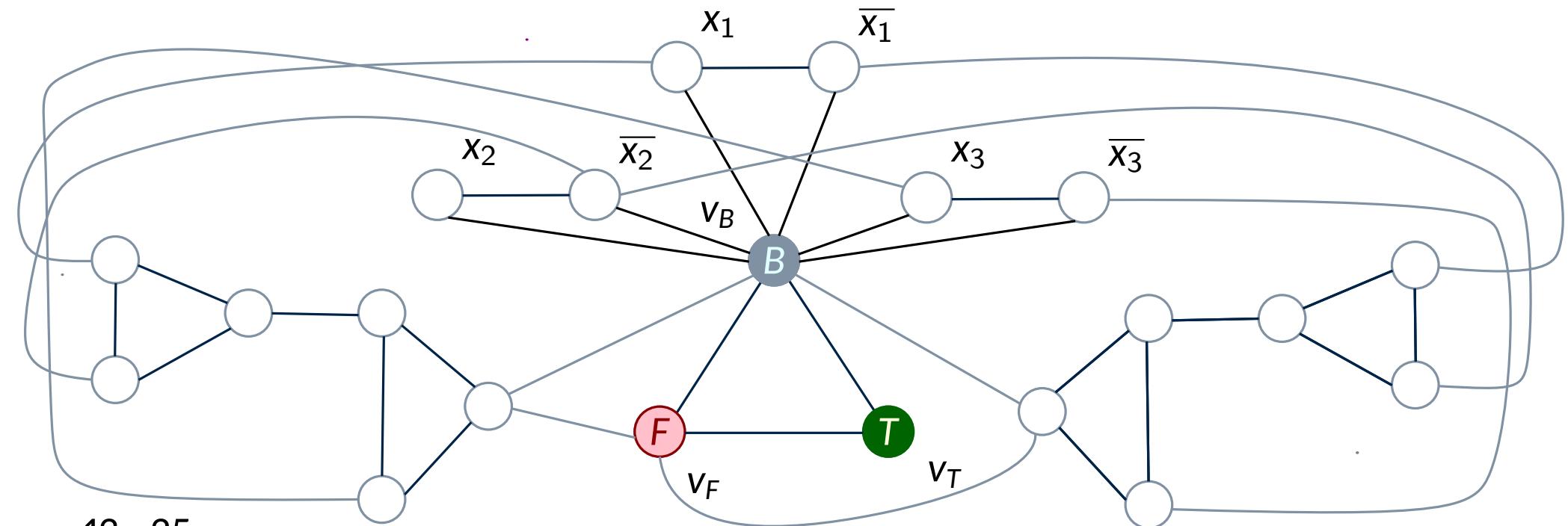
# Gesamte Konstruktion

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Rightarrow$ " Sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

**Beispiel:**  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$

$$b = (1, 0, 1)$$



# Gesamte Konstruktion

Korrektheit:

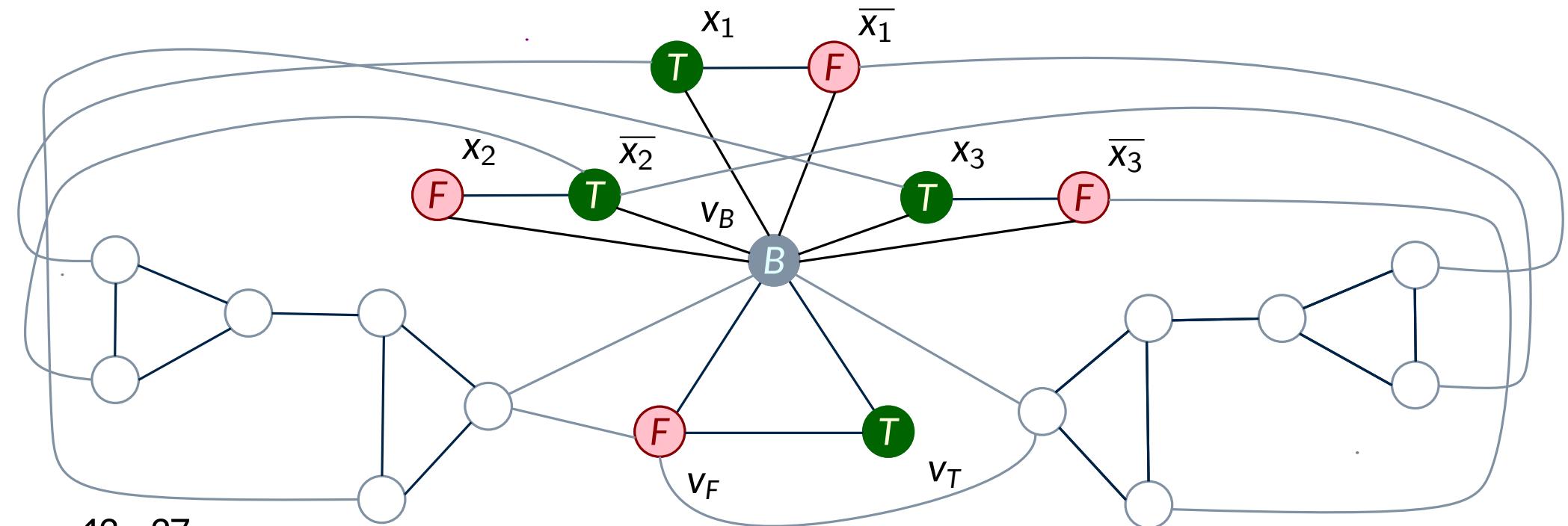
$\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Rightarrow$ " Sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

Wenn  $b_i = 1$ , färben wir  $v_{x_i}$  mit **T** und  $v_{\bar{x}_i}$  mit **F**. Wenn  $b_i = 0$ , färben wir  $v_{x_i}$  und  $v_{\bar{x}_i}$  umgekehrt.

Beispiel:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$b = (1, 0, 1)$



# Gesamte Konstruktion

**Korrektheit:**

$\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Rightarrow$ " Sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

Wenn  $b_i = 1$ , färben wir  $v_{x_i}$  mit **T** und  $v_{\bar{x}_i}$  mit **F**. Wenn  $b_i = 0$ , färben wir  $v_{x_i}$  und  $v_{\bar{x}_i}$  umgekehrt.

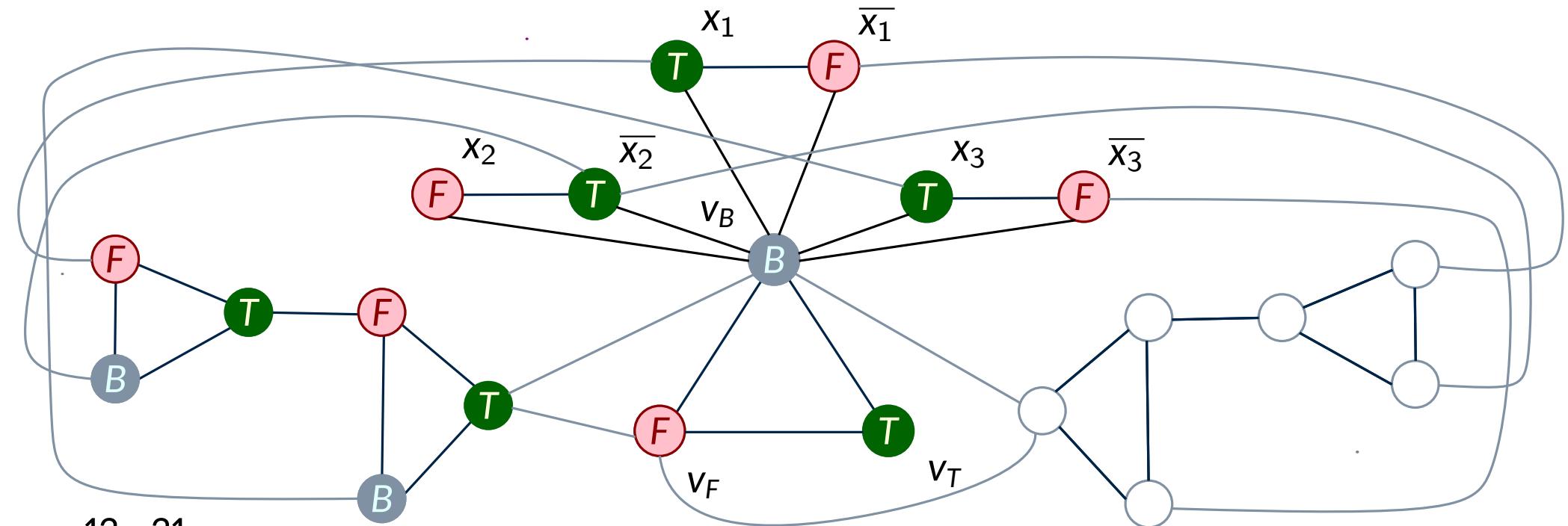
Da  $b$  erfüllend sind, gilt für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$ :

- mindestens ein "Input"-Literalknoten ist mit **T** gefärbt

$\Rightarrow$  wir können eine gültige 3-Färbung der Knoten des Klauselgadgets für  $C_i$  finden

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$b = (1, 0, 1)$



# Gesamte Konstruktion

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Rightarrow$ " Sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

Wenn  $b_i = 1$ , färben wir  $v_{x_i}$  mit **T** und  $v_{\bar{x}_i}$  mit **F**. Wenn  $b_i = 0$ , färben wir  $v_{x_i}$  und  $v_{\bar{x}_i}$  umgekehrt.

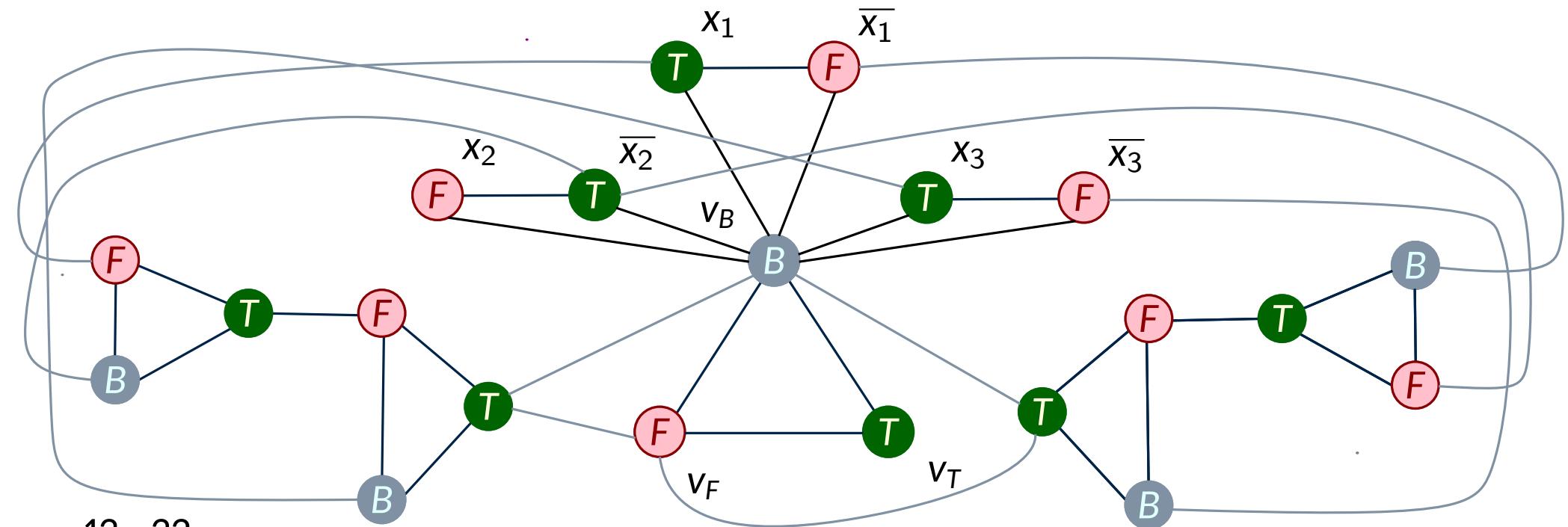
Da  $b$  erfüllend sind, gilt für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$ :

- mindestens ein "Input"-Literalknoten ist mit **T** gefärbt

$\Rightarrow$  wir können eine gültige 3-Färbung der Knoten des Klauselgadgets für  $C_i$  finden

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$b = (1, 0, 1)$



# Gesamte Konstruktion

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Rightarrow$ " Sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

Wenn  $b_i = 1$ , färben wir  $v_{x_i}$  mit **T** und  $v_{\bar{x}_i}$  mit **F**. Wenn  $b_i = 0$ , färben wir  $v_{x_i}$  und  $v_{\bar{x}_i}$  umgekehrt.

Da  $b$  erfüllend sind, gilt für jede Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$ :

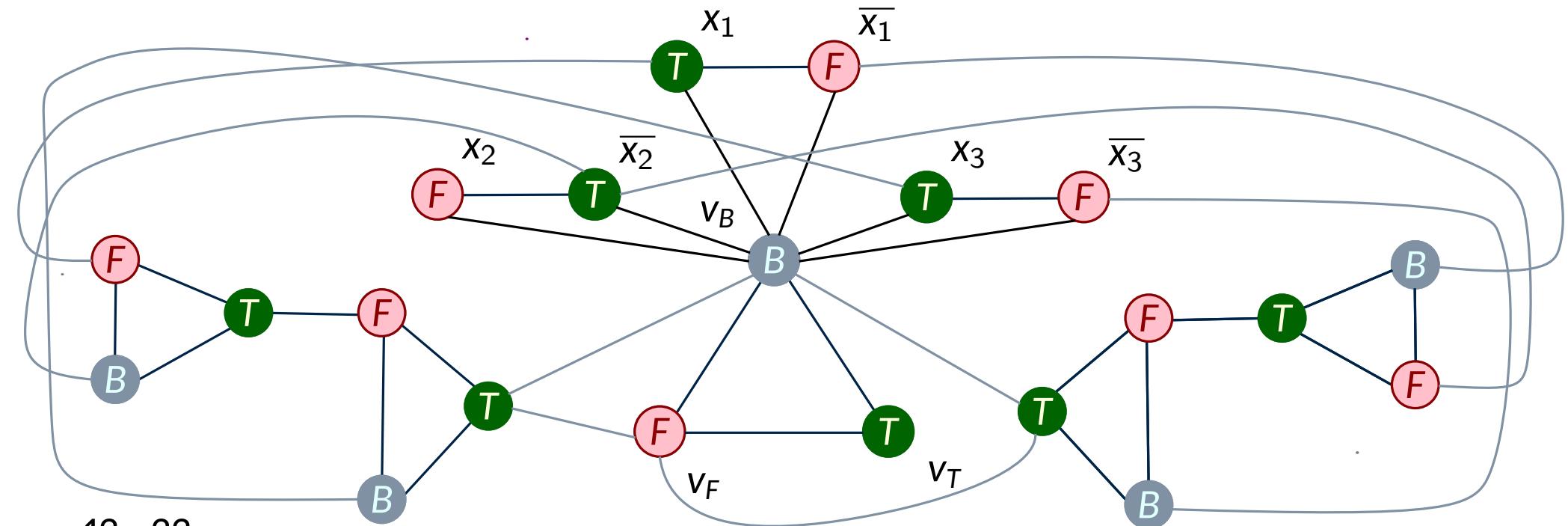
- mindestens ein "Input"-Literalknoten ist mit **T** gefärbt

$\Rightarrow$  wir können eine gültige 3-Färbung der Knoten des Klauselgadgets für  $C_i$  finden

$\Rightarrow$  alle Knoten können gültig eingefärbt werden

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$b = (1, 0, 1)$

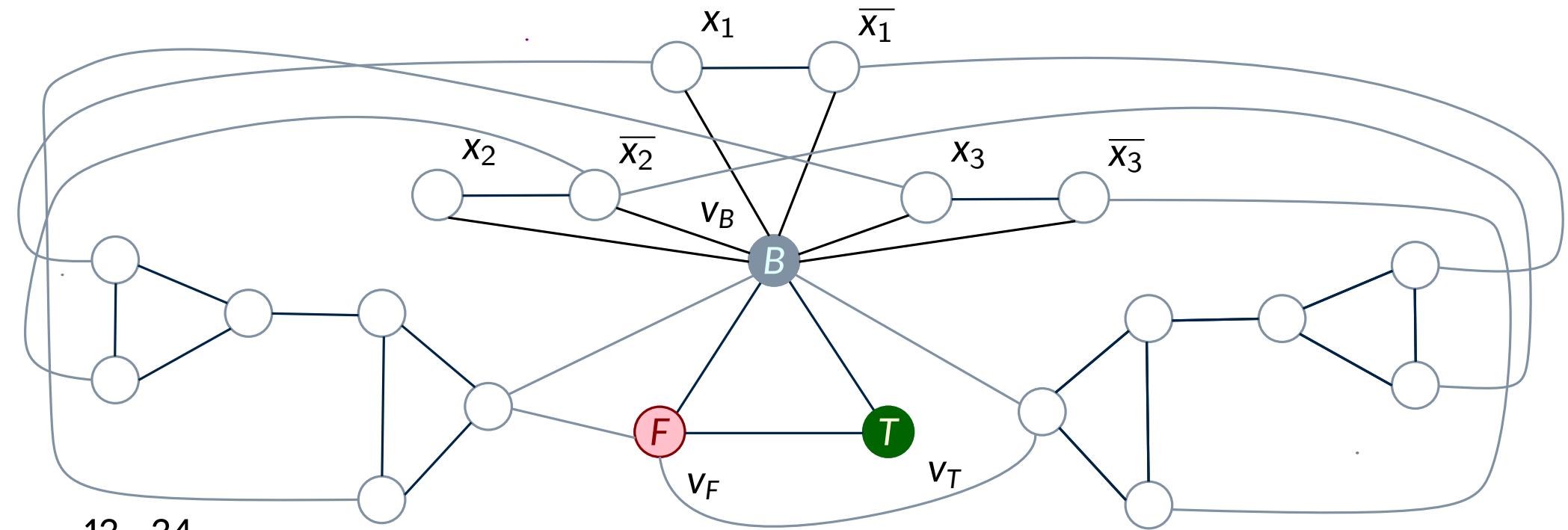


# Gesamte Konstruktion

Korrektheit:

$\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

Beispiel:  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$

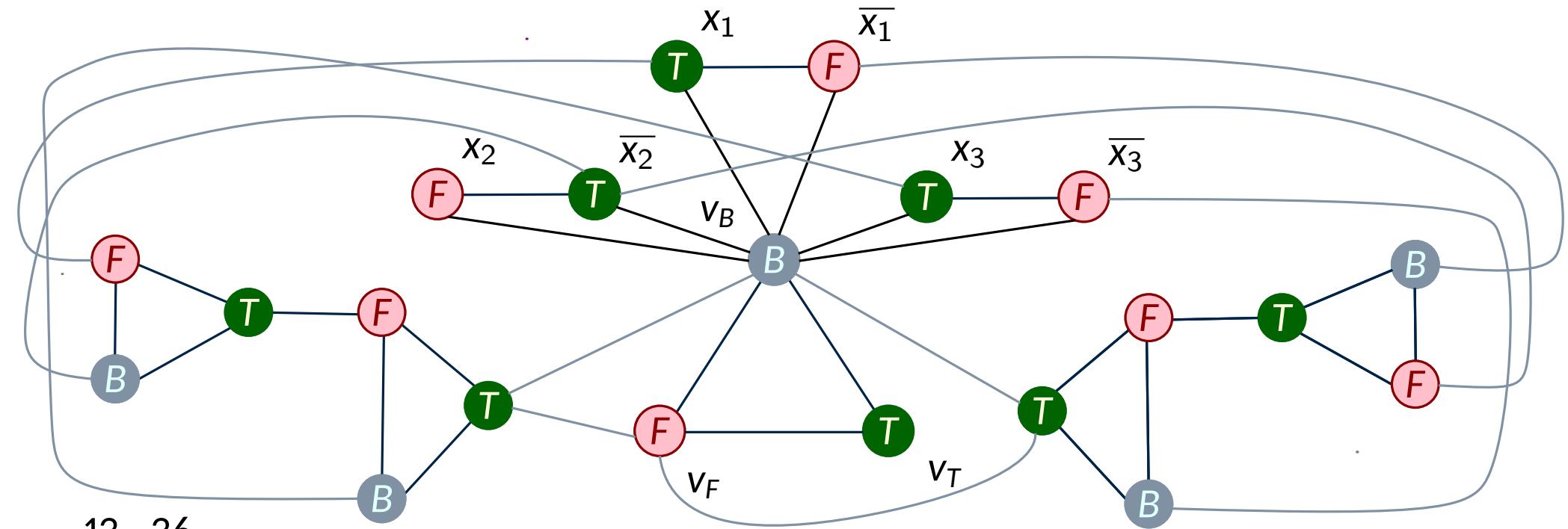


# Gesamte Konstruktion

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Leftarrow$ " Sei  $c$  eine gültige 3-Färbung von  $G$ .

**Beispiel:**  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$



# Gesamte Konstruktion

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

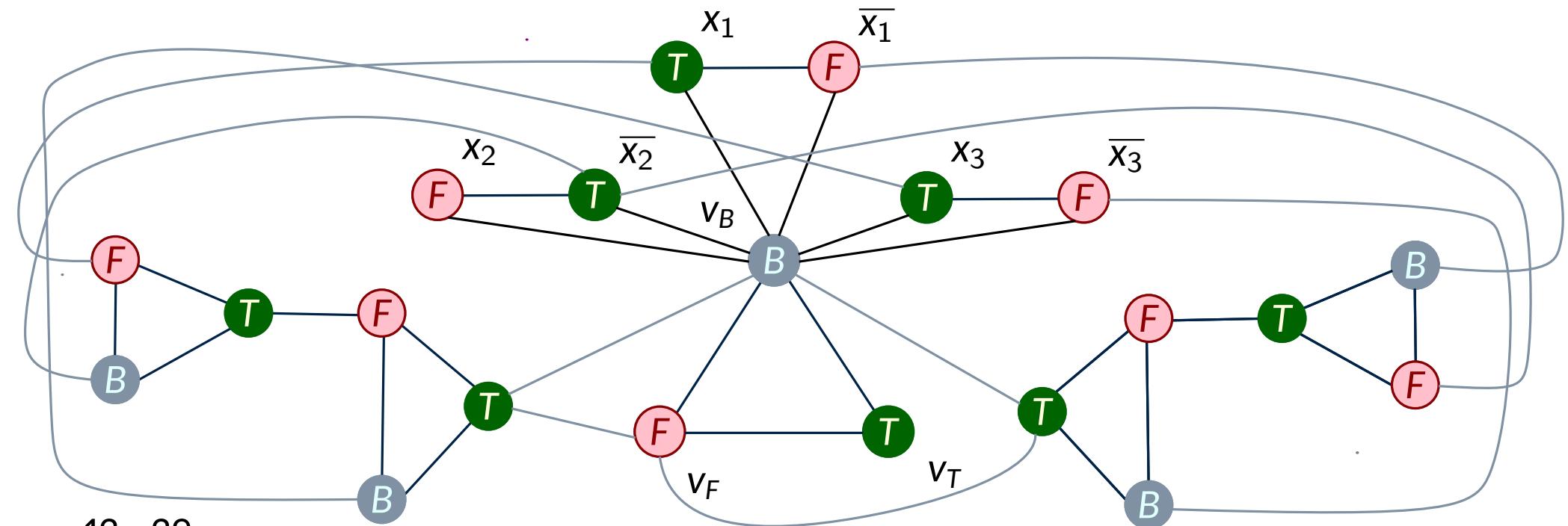
" $\Leftarrow$ " Sei  $c$  eine gültige 3-Färbung von  $G$ .

Für jedes  $x_i$  muss genau ein Literalknoten  $v_{x_i}$  oder  $v_{\bar{x}_i}$  mit **T** gefärbt sein.

Wir definieren eine Belegung  $b \in \{0, 1\}^n$  durch  $b_i = 1 \Leftrightarrow v_{x_i}$  ist **T** gefärbt.

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$$b = (1, 0, 1)$$



# Gesamte Konstruktion

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

" $\Leftarrow$ " Sei  $c$  eine gültige 3-Färbung von  $G$ .

Für jedes  $x_i$  muss genau ein Literalknoten  $v_{x_i}$  oder  $v_{\bar{x}_i}$  mit **T** gefärbt sein.

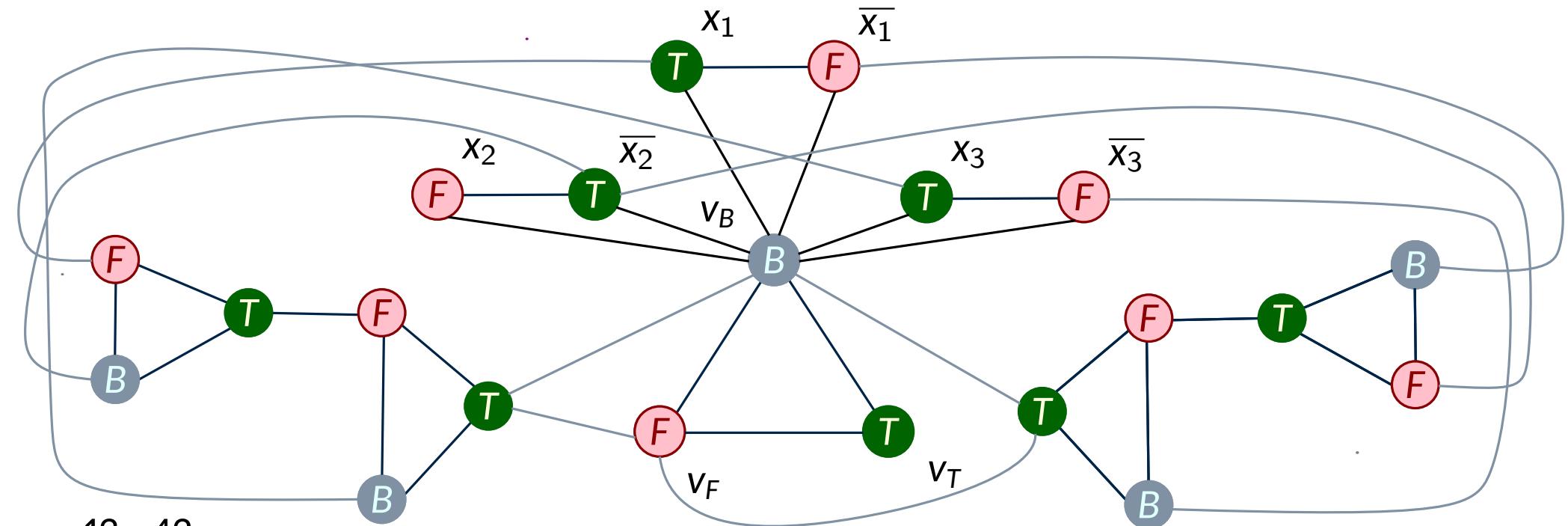
Wir definieren eine Belegung  $b \in \{0, 1\}^n$  durch  $b_i = 1 \Leftrightarrow v_{x_i}$  ist **T** gefärbt.

$b$  ist erfüllend, denn:

es kann keine Klausel  $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$  geben, sodass  $\ell_1, \ell_2, \ell_3$  alle **F** gefärbt sind  
 $\rightarrow$  für das entsprechende Klauselgadget gäbe es keine gültige 3-Färbung

**Beispiel:**  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$$b = (1, 0, 1)$$



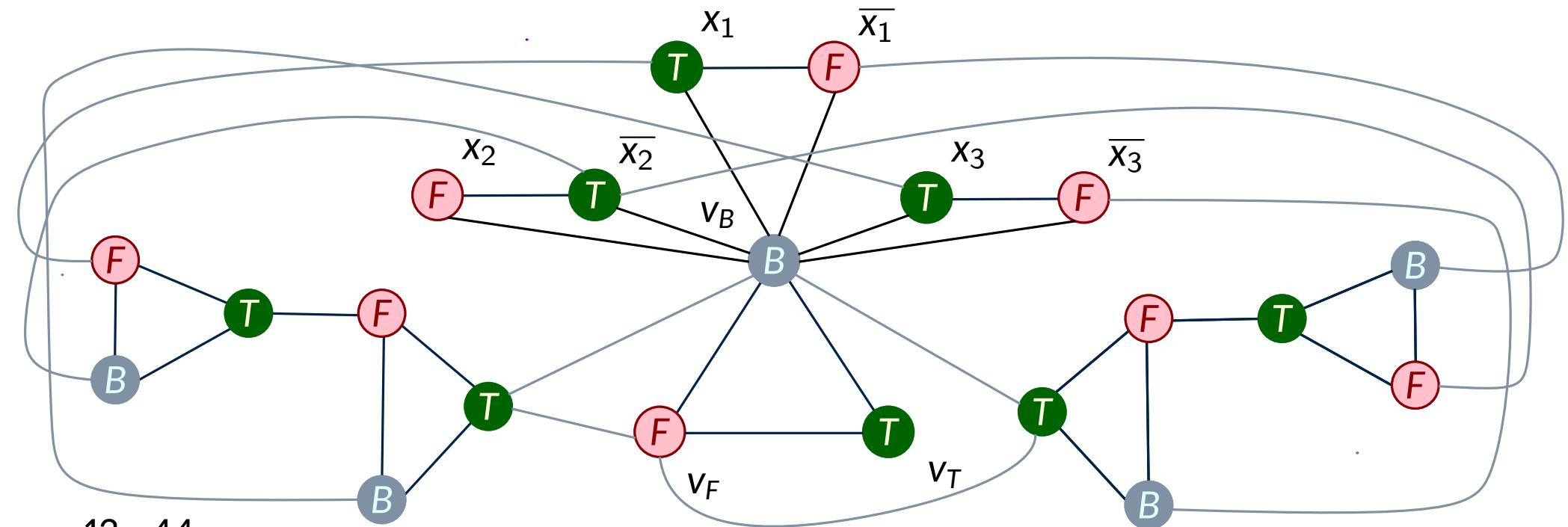
# Gesamte Konstruktion

Bemerkung:

Wir sind im Beweis implizit davon ausgegangen, dass alle Klauseln **genau** drei Literale haben.

Frage: Was machen wir mit Klauseln, die nur ein oder zwei Literale haben?

Beispiel:  $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$



# **Traveling Salesperson Problem**

# Traveling Salesperson Problem: Definition

## Traveling Salesperson Problem (TSP).

vollständiger Graph: enthält alle möglichen Kanten, d.h.  $E = \binom{V}{2}$

**Gegeben:** Ein vollständiger ungerichteter, gewichteter Graph  $G = (V, E, w)$

**Gesucht:** Eine Rundreise mit den geringsten Kosten

Eine **Rundreise** (Tour, Hamiltonkreis) ist ein geschlossener Pfad, der alle Knoten besucht.

Kosten einer Rundreise  $r = (r_0, \dots, r_n)$ :  $\sum_{i=0}^{n-1} w(r_i, r_{i+1})$

Hinweis:  $r_0 = r_n$

↗ Präsentation

<http://kiosk.mpi-inf.mpg.de/misc/tsp/index.html>

## Entscheidungsvariante des Traveling Salesperson Problems (TSP-E).

**Gegeben:** Ein vollständiger ungerichteter, gewichteter Graph  $G = (V, E, w)$  und  $W \in \mathbb{N}$

**Gesucht:** Gibt es eine Rundreise mit Kosten  $\leq W$ ?

Wir werden zeigen, dass TSP-E ein NP-vollständiges Problem ist.

Dazu betrachten wir zunächst ein verwandtes Problem: Hamiltonpfad bzw. -kreis

# Directed Hamiltonian Path

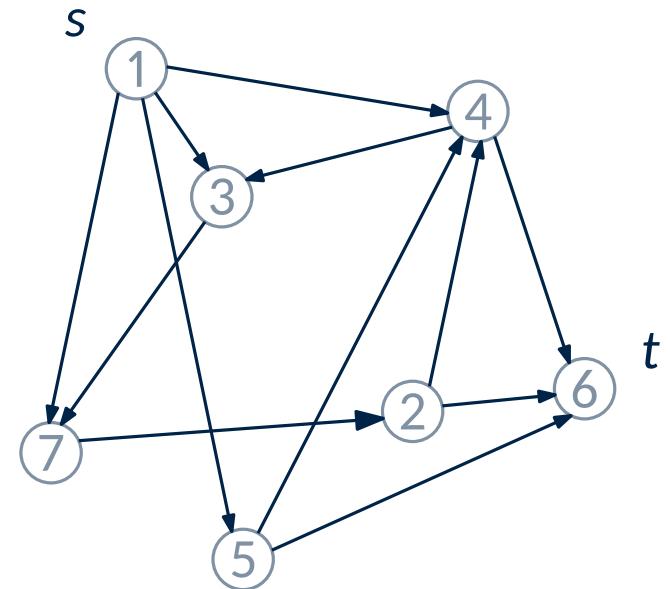
**Directed Hamiltonian Path (dHamPath).**

**Gegeben:** Ein gerichteter Graph  $G = (V, E)$ , sowie Startknoten  $s \in V$  und Zielknoten  $t \in V$ .

**Gesucht:** Gibt es einen **Hamiltonpfad** von  $s$  nach  $t$  in  $G$ ?

Ein Pfad  $p = (p_0, \dots, p_{n-1})$  von  $p_0 = s$  nach  $p_{n-1} = t$  heißt **Hamiltonpfad**, wenn:  
jeder Knoten in  $G$  wird genau einmal von  $p$  besucht

**Frage:** Hat  $G$  einen Hamiltonpfad von  $s$  nach  $t$ ?



# Directed Hamiltonian Path

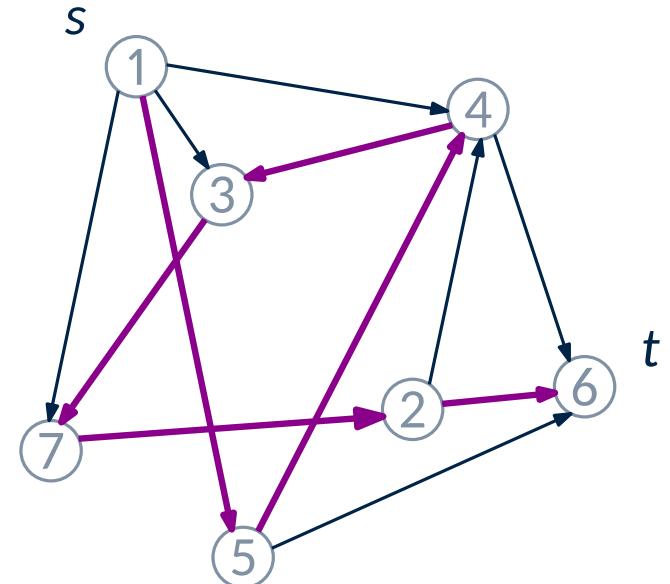
**Directed Hamiltonian Path (dHamPath).**

**Gegeben:** Ein gerichteter Graph  $G = (V, E)$ , sowie Startknoten  $s \in V$  und Zielknoten  $t \in V$ .

**Gesucht:** Gibt es einen **Hamiltonpfad** von  $s$  nach  $t$  in  $G$ ?

Ein Pfad  $p = (p_0, \dots, p_{n-1})$  von  $p_0 = s$  nach  $p_{n-1} = t$  heißt **Hamiltonpfad**, wenn:  
jeder Knoten in  $G$  wird genau einmal von  $p$  besucht

Frage: Hat  $G$  einen Hamiltonpfad von  $s$  nach  $t$ ? Ja.



# Directed Hamiltonian Path

**Directed Hamiltonian Path (dHamPath).**

**Gegeben:** Ein gerichteter Graph  $G = (V, E)$ , sowie Startknoten  $s \in V$  und Zielknoten  $t \in V$ .

**Gesucht:** Gibt es einen **Hamiltonpfad** von  $s$  nach  $t$  in  $G$ ?

Ein Pfad  $p = (p_0, \dots, p_{n-1})$  von  $p_0 = s$  nach  $p_{n-1} = t$  heißt **Hamiltonpfad**, wenn:  
jeder Knoten in  $G$  wird genau einmal von  $p$  besucht

Frage: Hat  $G$  einen Hamiltonpfad von  $s$  nach  $t$ ? Ja.

**Theorem.**

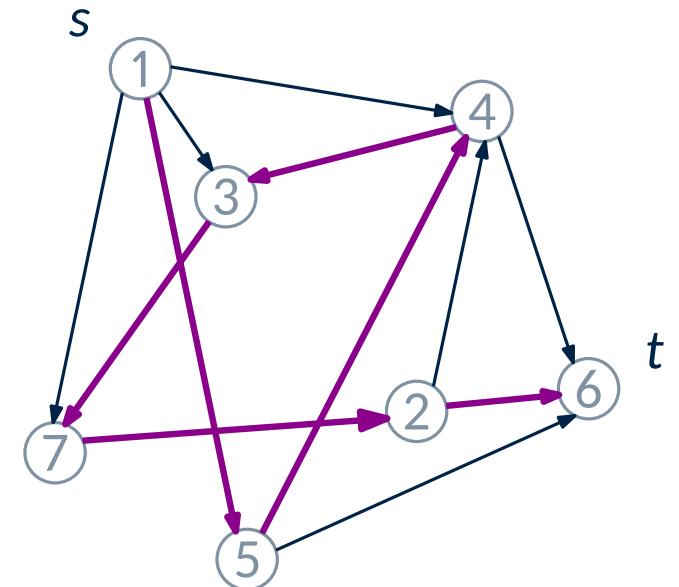
Directed Hamiltonian Path ist **NP-vollständig**.

**Beweis:** 1. Directed Hamiltonian Path ist in NP.

Frage: Warum? Beweis hier ausgelassen...

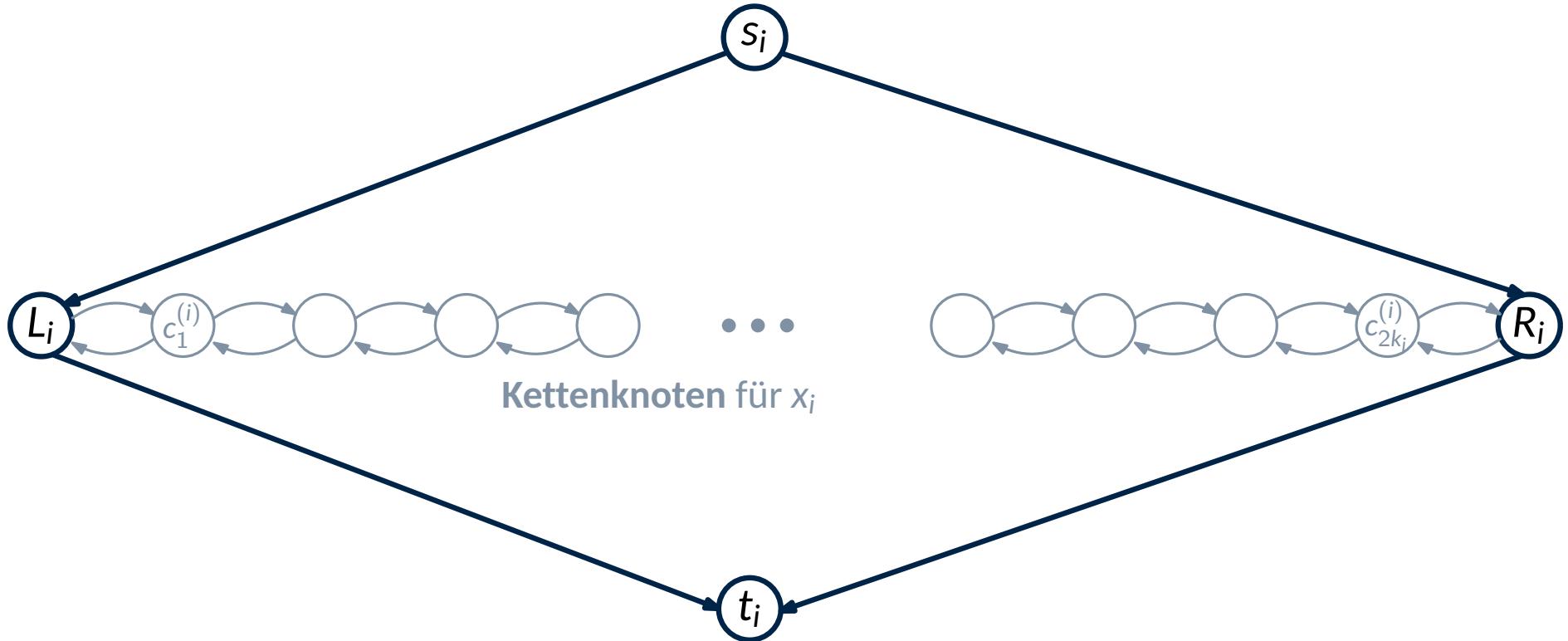
2. Wir reduzieren von SAT. Wollen also zeigen:  $SAT \leq_P dHamPath$

3-5. Nächste Folien: Beschreibung einer Graphkonstruktion  
für eine gegebene SAT-Formel  $\phi$



# Diamantengadget

Für jede Variable  $x_i$  definieren wir folgendes **Diamantengadget**:



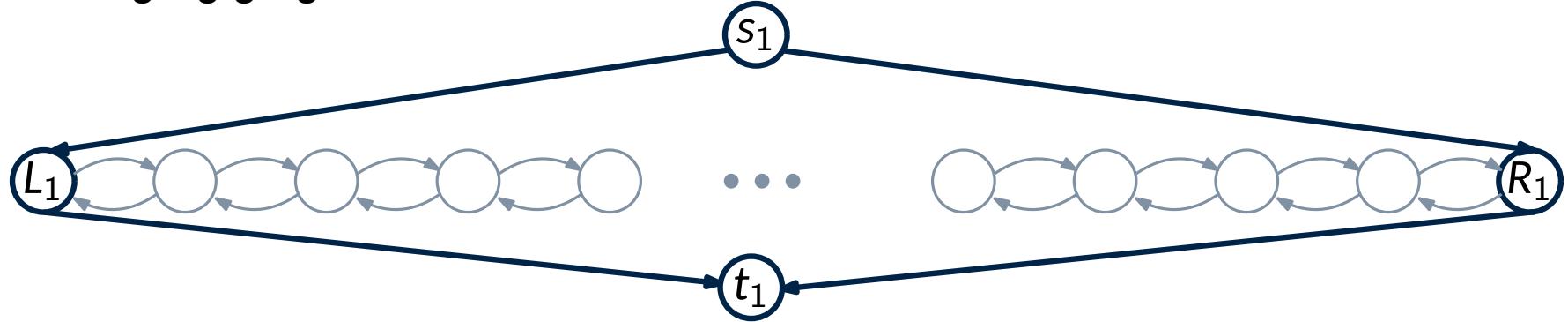
Frage: Wie sehen Hamiltonpfade von  $s_i$  nach  $t_i$  aus?

Festlegung: Für  $x_i$  führen wir  $2k_i$  Kettenknoten  $c_1^{(i)}, \dots, c_{2k_i}^{(i)}$  ein,  
wobei  $k_i$  die Anzahl der Literale  $x_i$  und  $\bar{x}_i$  in  $\phi$  bezeichnet.

# Belegungsgadget

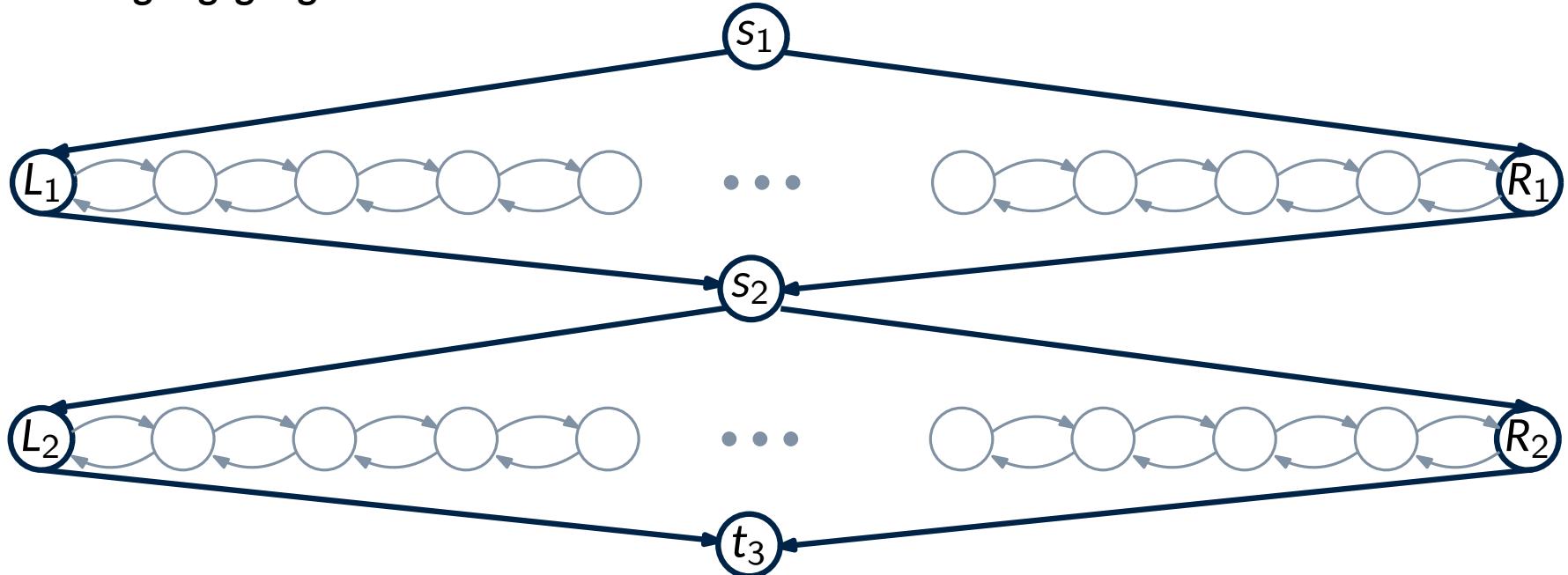
---

Wir schalten die Diamantengadgets für  $x_1, \dots, x_n$  in Reihe (sodass  $s_{i+1} = t_i$ ), um ein **Belegungsgadget** zu erhalten:



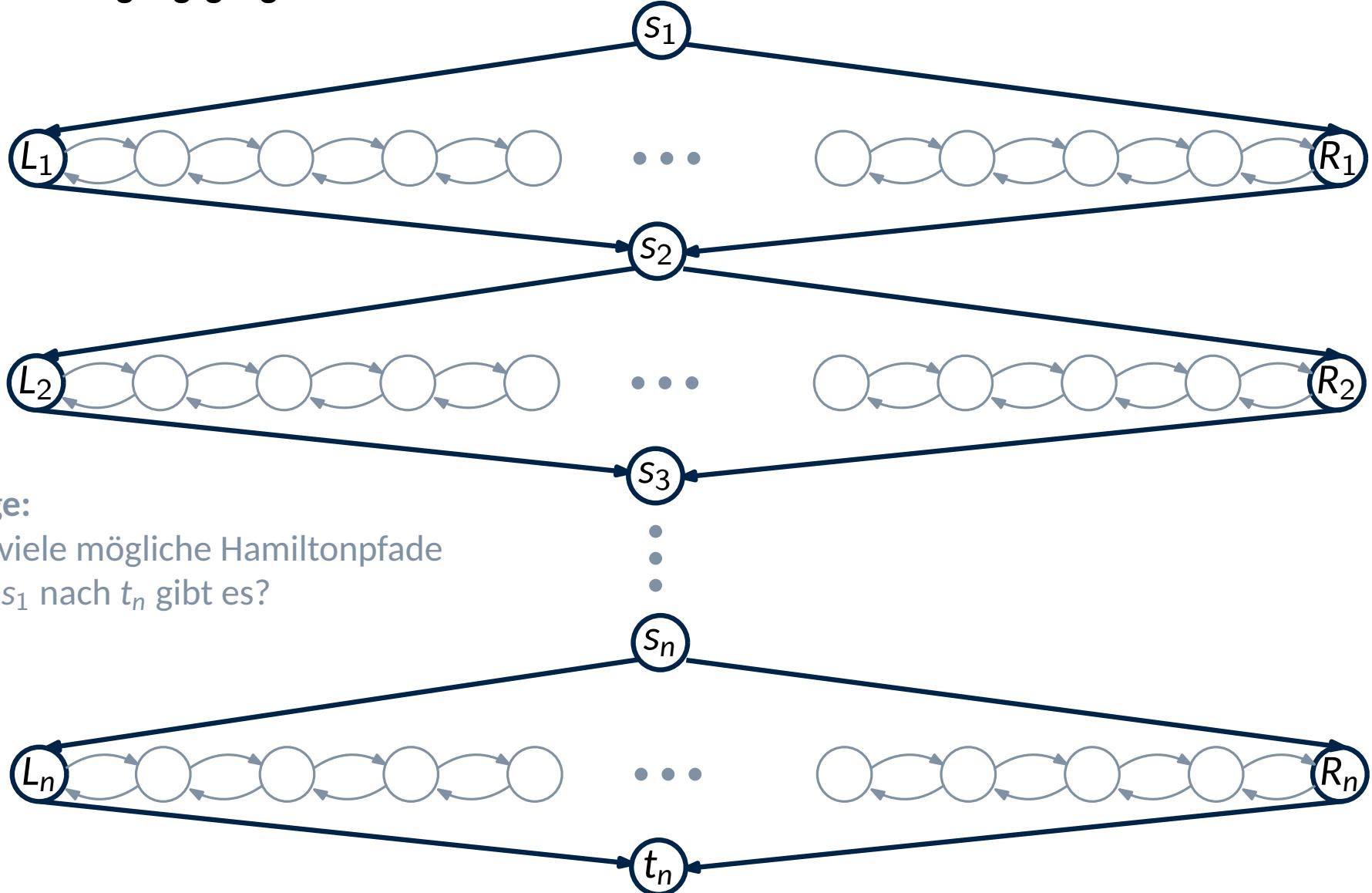
# Belegungsgadget

Wir schalten die Diamantengadgets für  $x_1, \dots, x_n$  in Reihe (sodass  $s_{i+1} = t_i$ ), um ein **Belegungsgadget** zu erhalten:



# Belegungsgadget

Wir schalten die Diamantengadgets für  $x_1, \dots, x_n$  in Reihe (sodass  $s_{i+1} = t_i$ ), um ein **Belegungsgadget** zu erhalten:

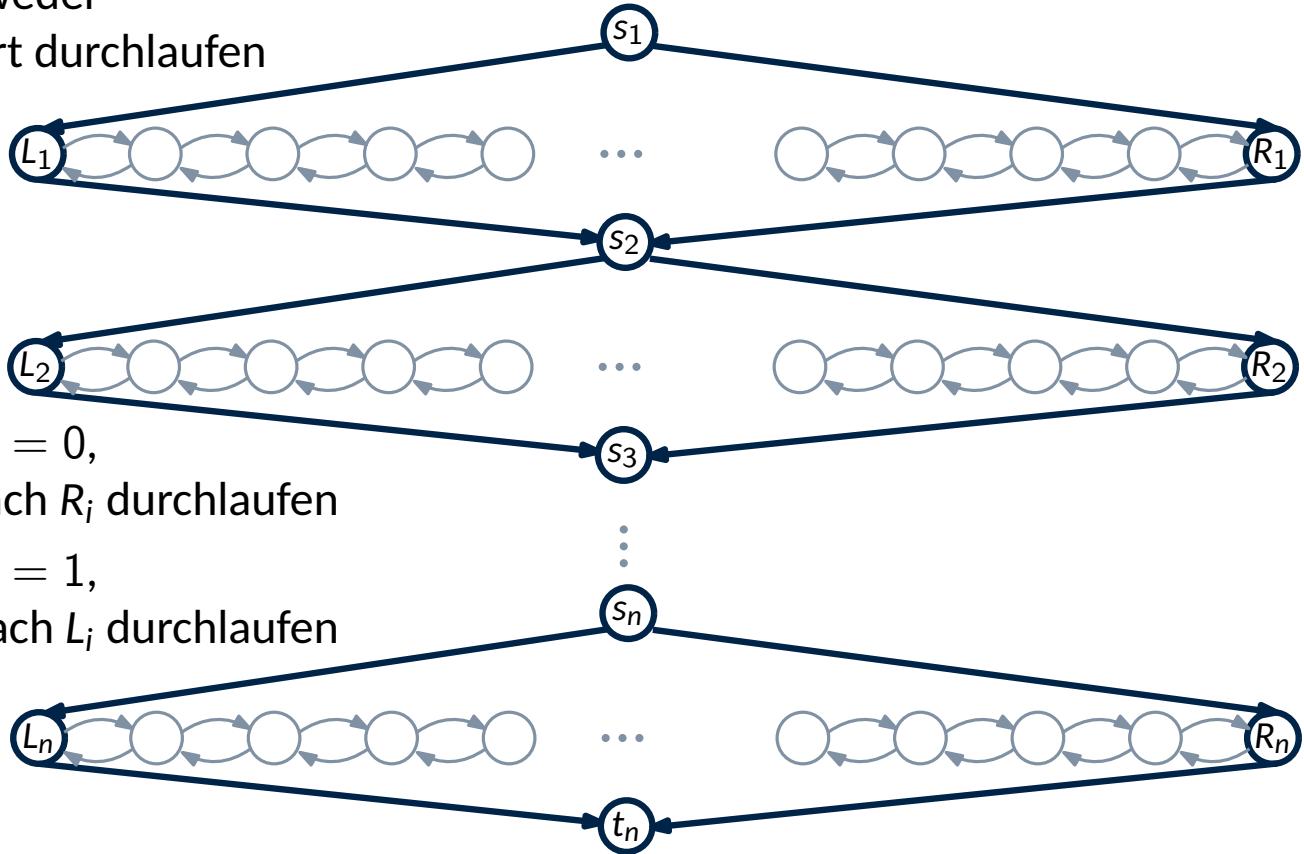


Frage:

Wieviele mögliche Hamiltonpfade  
von  $s_1$  nach  $t_n$  gibt es?

# Belegungen und Pfade

Die Kettenknoten für  $x_i$  werden entweder von links nach rechts oder umgekehrt durchlaufen

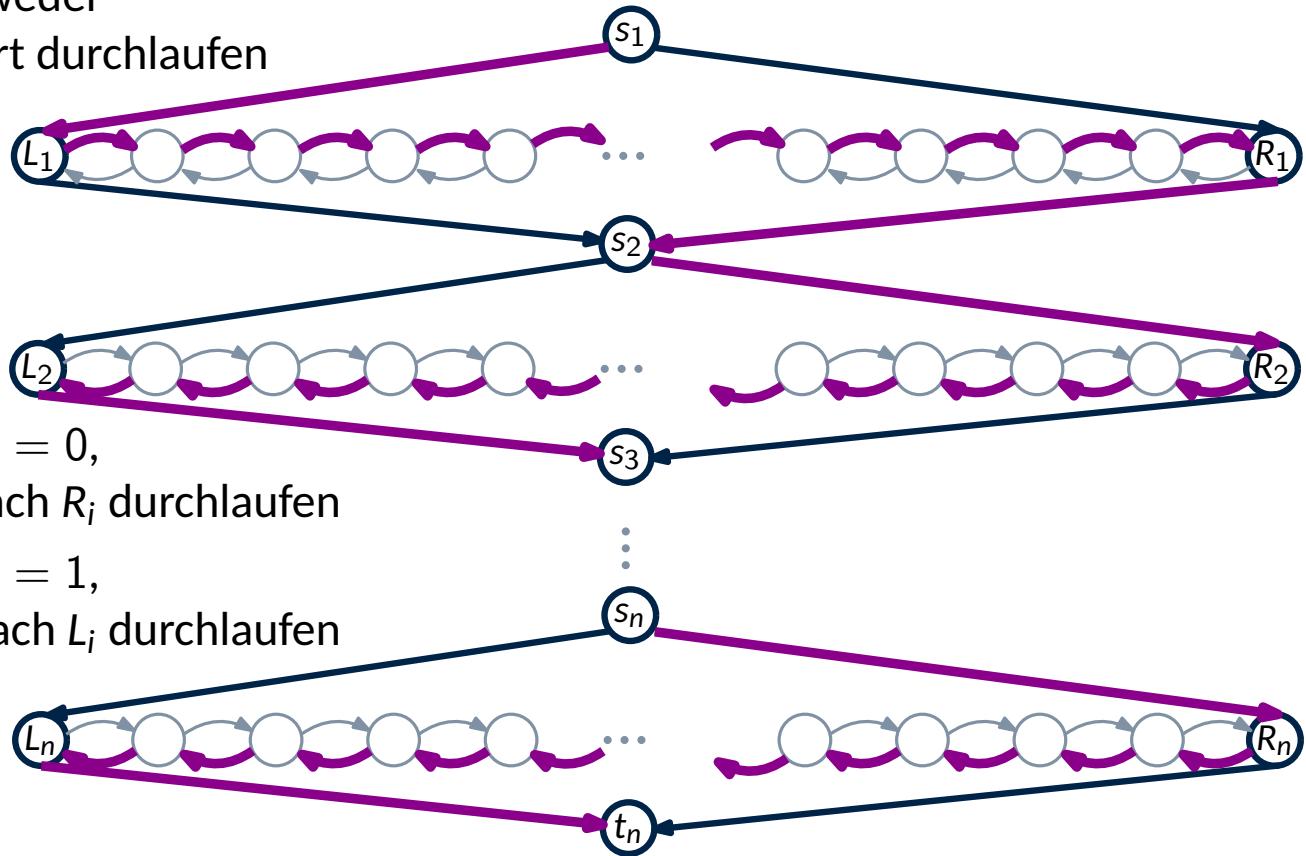


Wir interpretieren es als Belegung  $x_i = 0$ ,  
wenn wir  $x_i$ 's Kettenknoten von  $L_i$  nach  $R_i$  durchlaufen

Wir interpretieren es als Belegung  $x_i = 1$ ,  
wenn wir  $x_i$ 's Kettenknoten von  $R_i$  nach  $L_i$  durchlaufen

# Belegungen und Pfade

Die Kettenknoten für  $x_i$  werden entweder von links nach rechts oder umgekehrt durchlaufen



Wir interpretieren es als Belegung  $x_i = 0$ ,  
wenn wir  $x_i$ 's Kettenknoten von  $L_i$  nach  $R_i$  durchlaufen

Wir interpretieren es als Belegung  $x_i = 1$ ,  
wenn wir  $x_i$ 's Kettenknoten von  $R_i$  nach  $L_i$  durchlaufen

**Beispiel:** Der angezeigte Hamiltonpfad von  $s_1$  nach  $t_1$  entspricht  $b = (0, 1, \dots, 1)$

⇒ **Bijektion** zwischen Belegungen und Hamiltonpfaden von  $s_1$  nach  $t_n$

# Klauselgadget

Für jede Klausel  $C_j = (\ell_1 \vee \dots \vee \ell_k)$  wollen wir überprüfen, ob die Klausel erfüllt ist

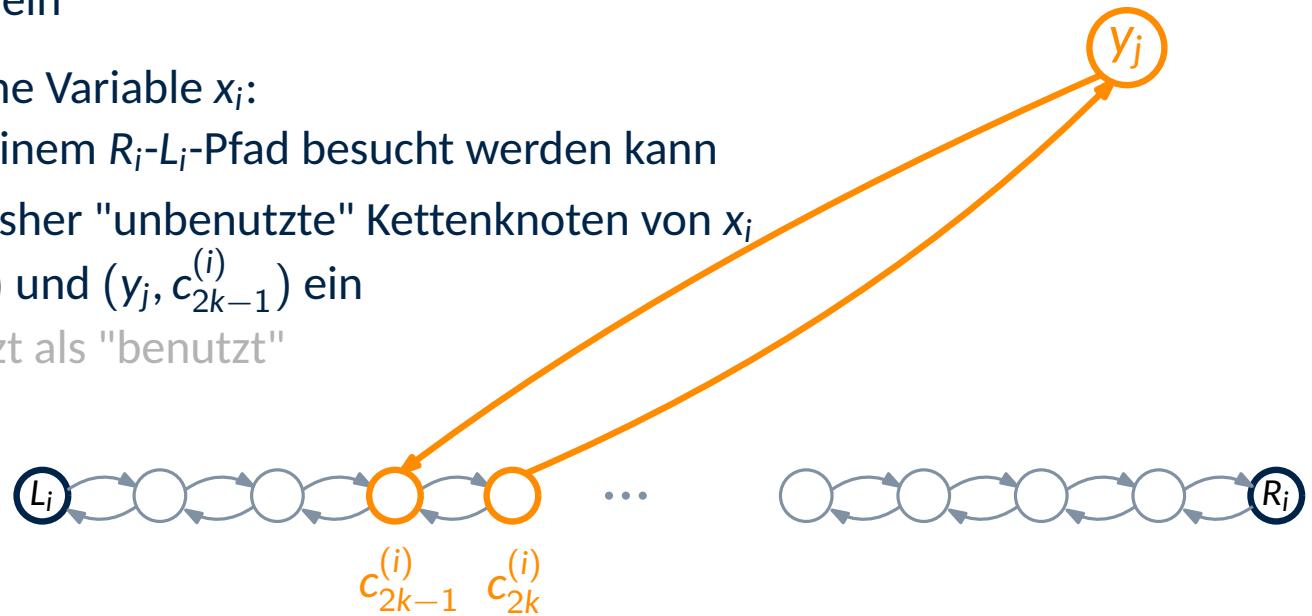
Dazu definieren wir folgendes **Klauselgadget** für Klausel  $C_j$ :

Wir führen einen neuen Knoten  $y_j$  ein

Für jedes Literal der Form  $x_i$  für eine Variable  $x_i$ :

→ füge Kanten ein, sodass  $y_j$  auf einem  $R_j$ - $L_i$ -Pfad besucht werden kann

- **genauer:**
- es seien  $c_{2k-1}^{(i)}, c_{2k}^{(i)}$  bisher "unbenutzte" Kettenknoten von  $x_i$
  - füge Kanten  $(c_{2k}^{(i)}, y_j)$  und  $(y_j, c_{2k-1}^{(i)})$  ein
  - $c_{2k-1}^{(i)}, c_{2k}^{(i)}$  gelten jetzt als "benutzt"



# Klauselgadget

Für jede Klausel  $C_j = (\ell_1 \vee \dots \vee \ell_k)$  wollen wir überprüfen, ob die Klausel erfüllt ist

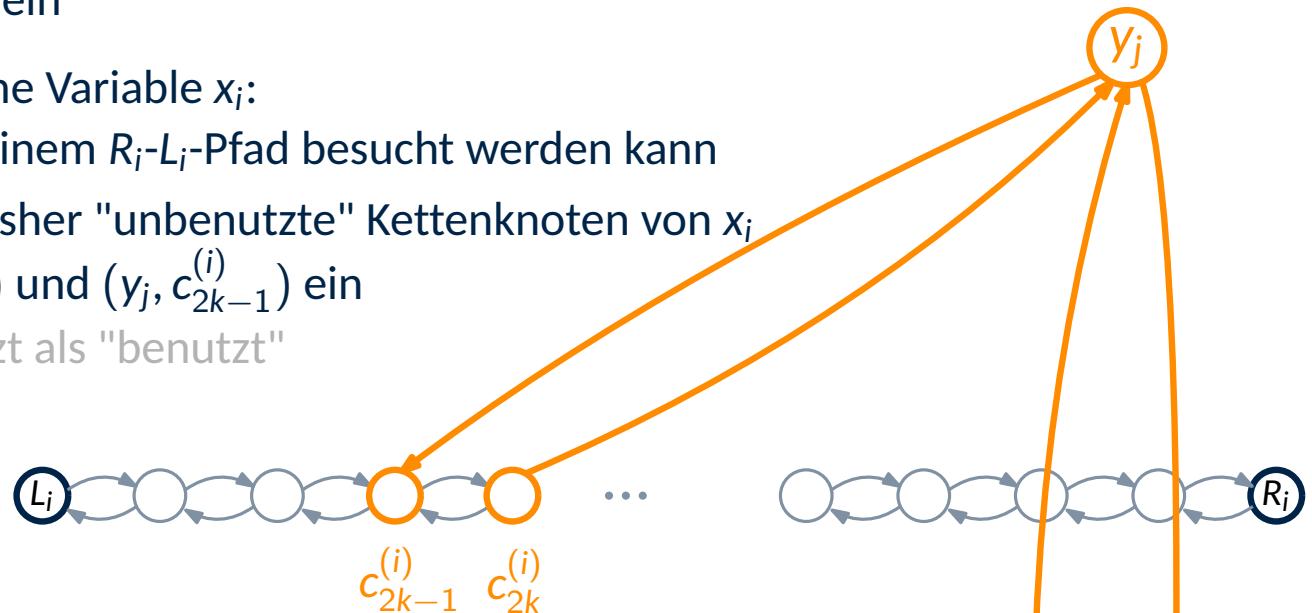
Dazu definieren wir folgendes **Klauselgadget** für Klausel  $C_j$ :

Wir führen einen neuen Knoten  $y_j$  ein

Für jedes Literal der Form  $x_i$  für eine Variable  $x_i$ :

→ füge Kanten ein, sodass  $y_j$  auf einem  $R_i$ - $L_i$ -Pfad besucht werden kann

- **genauer:**
- es seien  $c_{2k-1}^{(i)}, c_{2k}^{(i)}$  bisher "unbenutzte" Kettenknoten von  $x_i$
  - füge Kanten  $(c_{2k}^{(i)}, y_j)$  und  $(y_j, c_{2k-1}^{(i)})$  ein
  - $c_{2k-1}^{(i)}, c_{2k}^{(i)}$  gelten jetzt als "benutzt"



Für jedes Literal der Form  $\bar{x}_{i'}$  für eine Variable  $x_{i'}$ :

→ füge Kanten ein, sodass  $y_j$  auf einem  $L_{i'}-R_{i'}$ -Pfad besucht werden kann

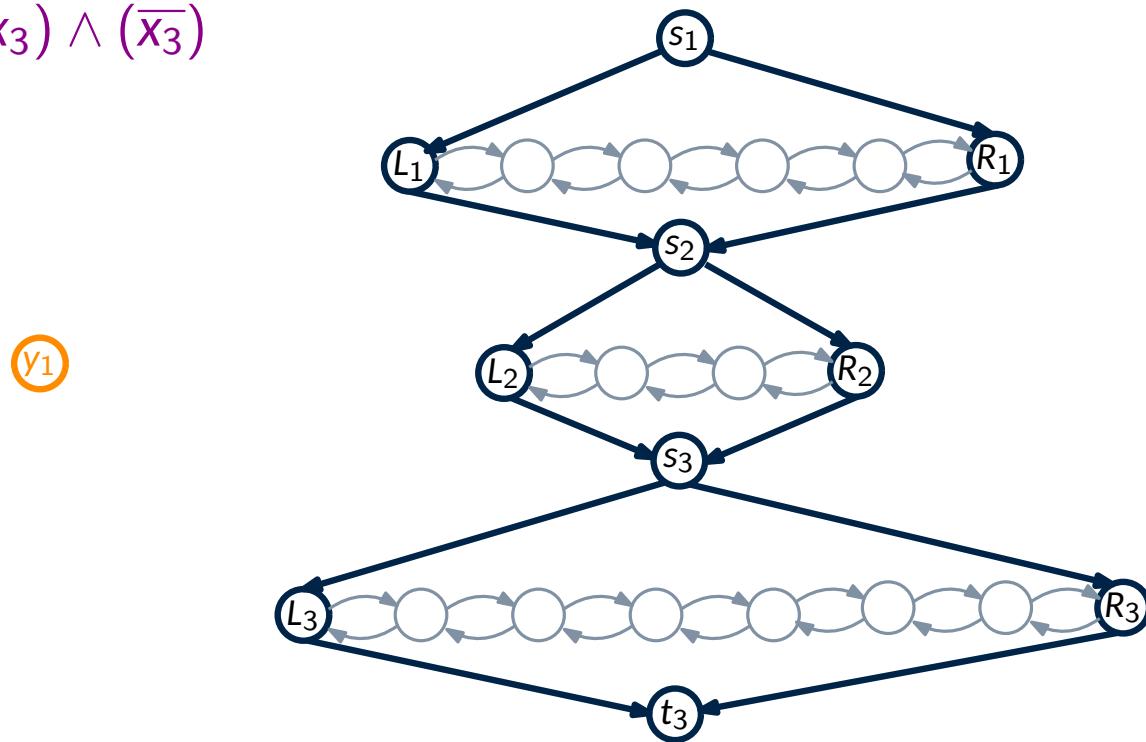
- **genauer:**
- es seien  $c_{2k-1}^{(i')}, c_{2k}^{(i')}$  bisher "unbenutzte" Kettenknoten von  $x_{i'}$
  - füge Kanten  $(c_{2k-1}^{(i')}, y_j)$  und  $(y_j, c_{2k}^{(i')})$  ein
  - $c_{2k-1}^{(i')}, c_{2k}^{(i')}$  gelten jetzt als "benutzt"



# Beispiel für die gesamte Konstruktion

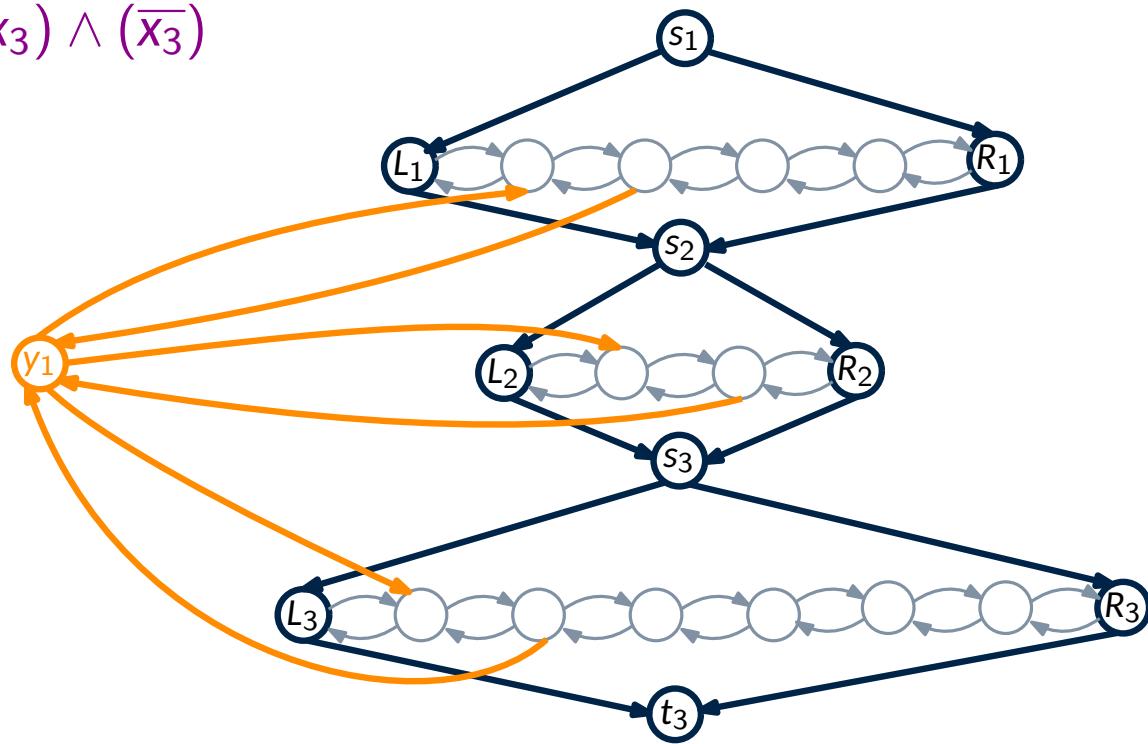
---

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$



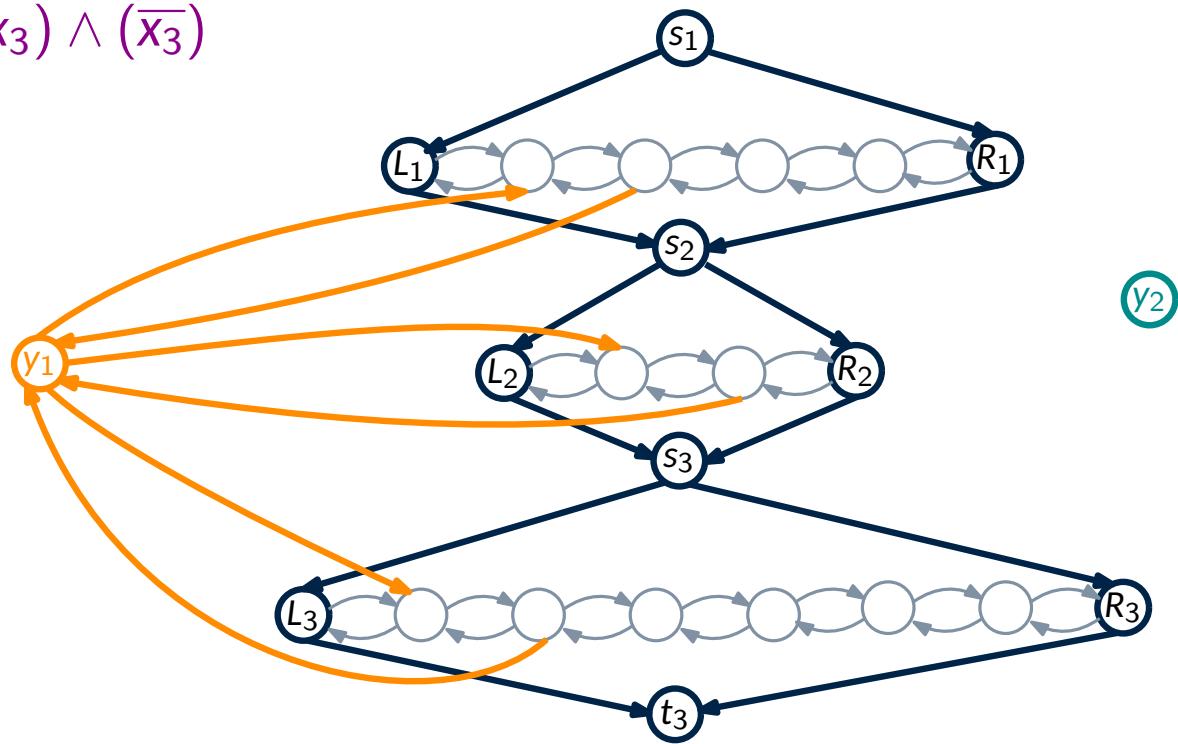
# Beispiel für die gesamte Konstruktion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$



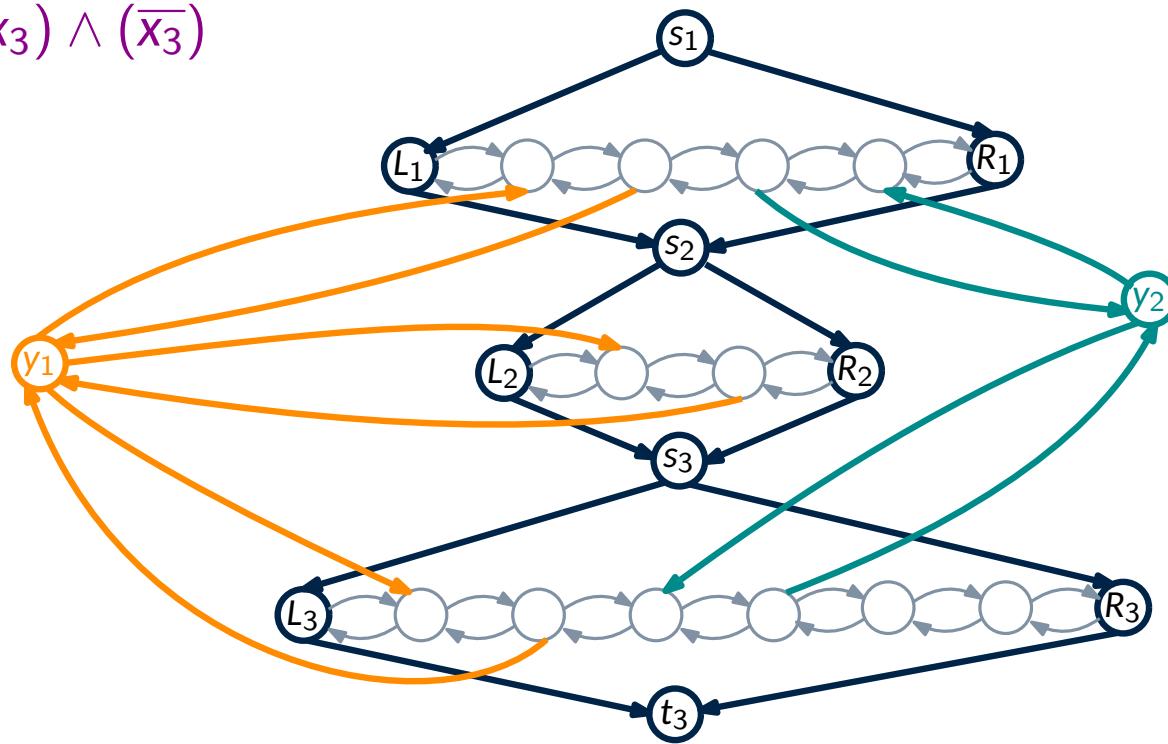
# Beispiel für die gesamte Konstruktion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$



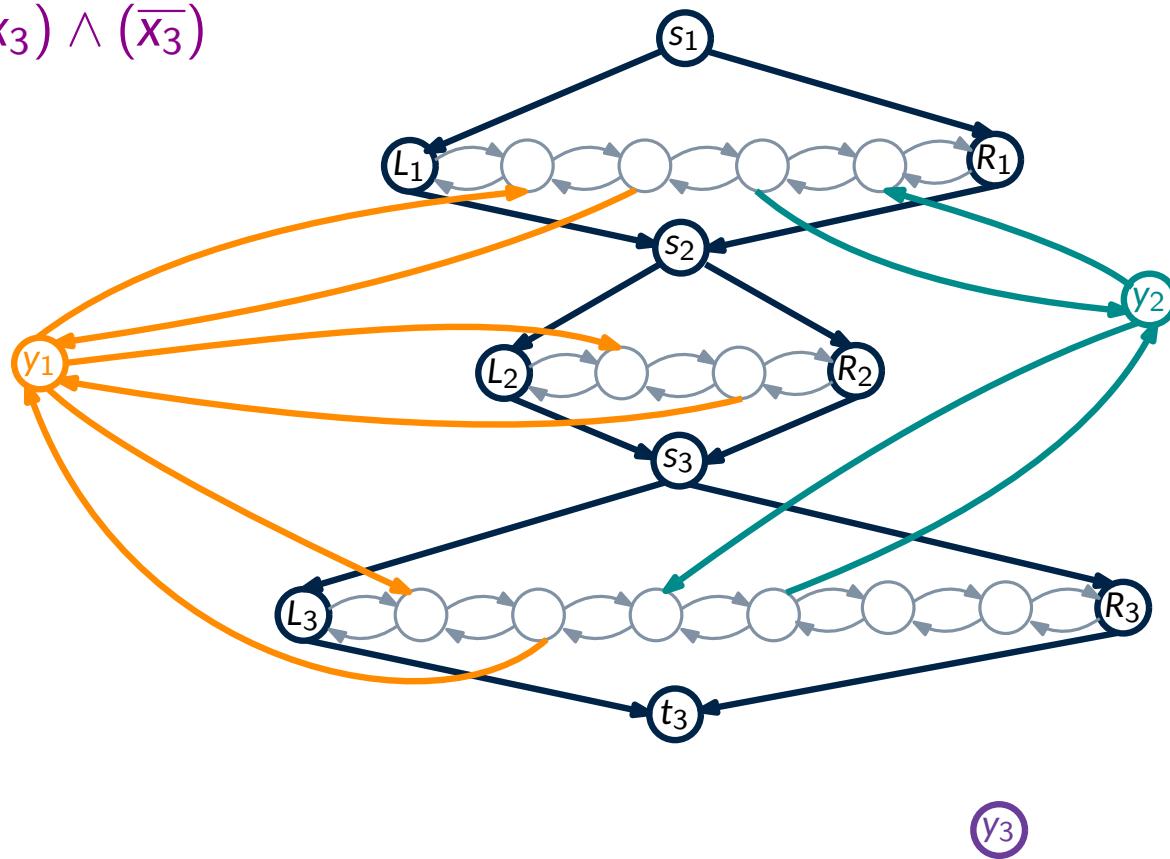
# Beispiel für die gesamte Konstruktion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$



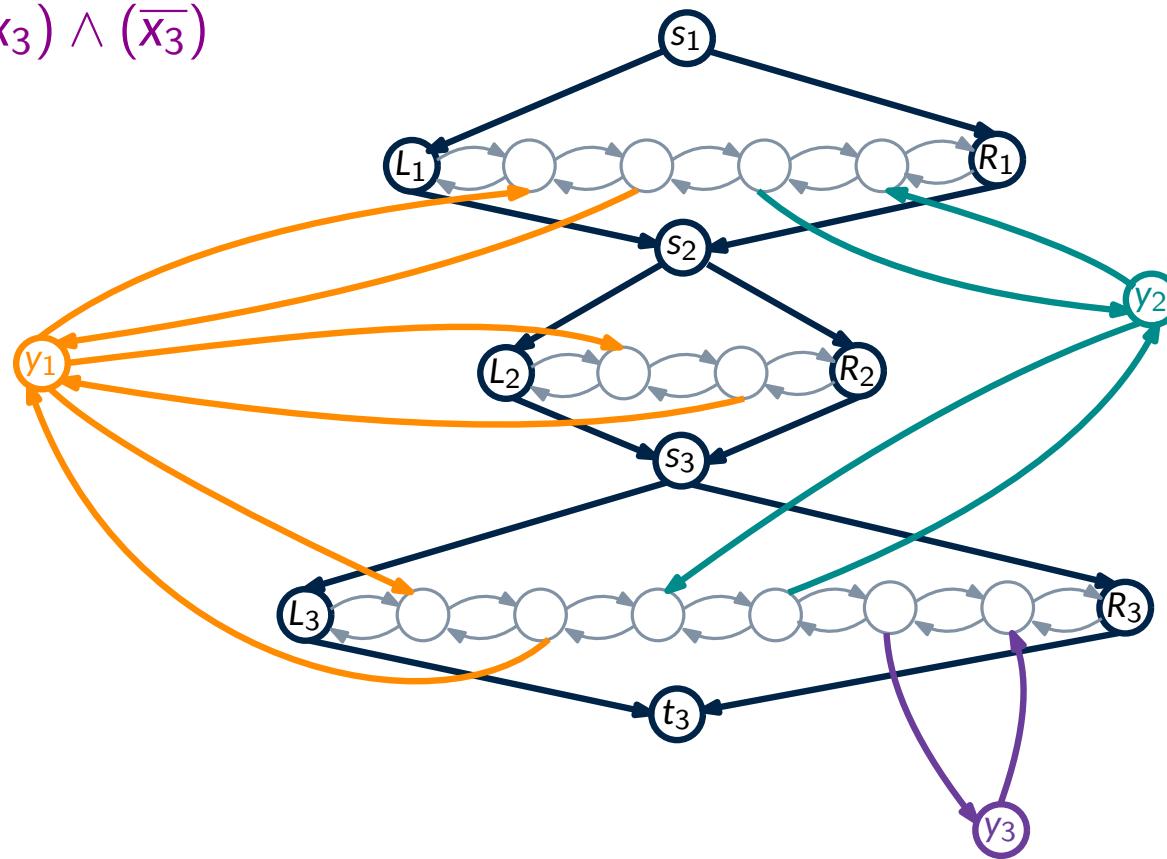
# Beispiel für die gesamte Konstruktion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$



# Beispiel für die gesamte Konstruktion

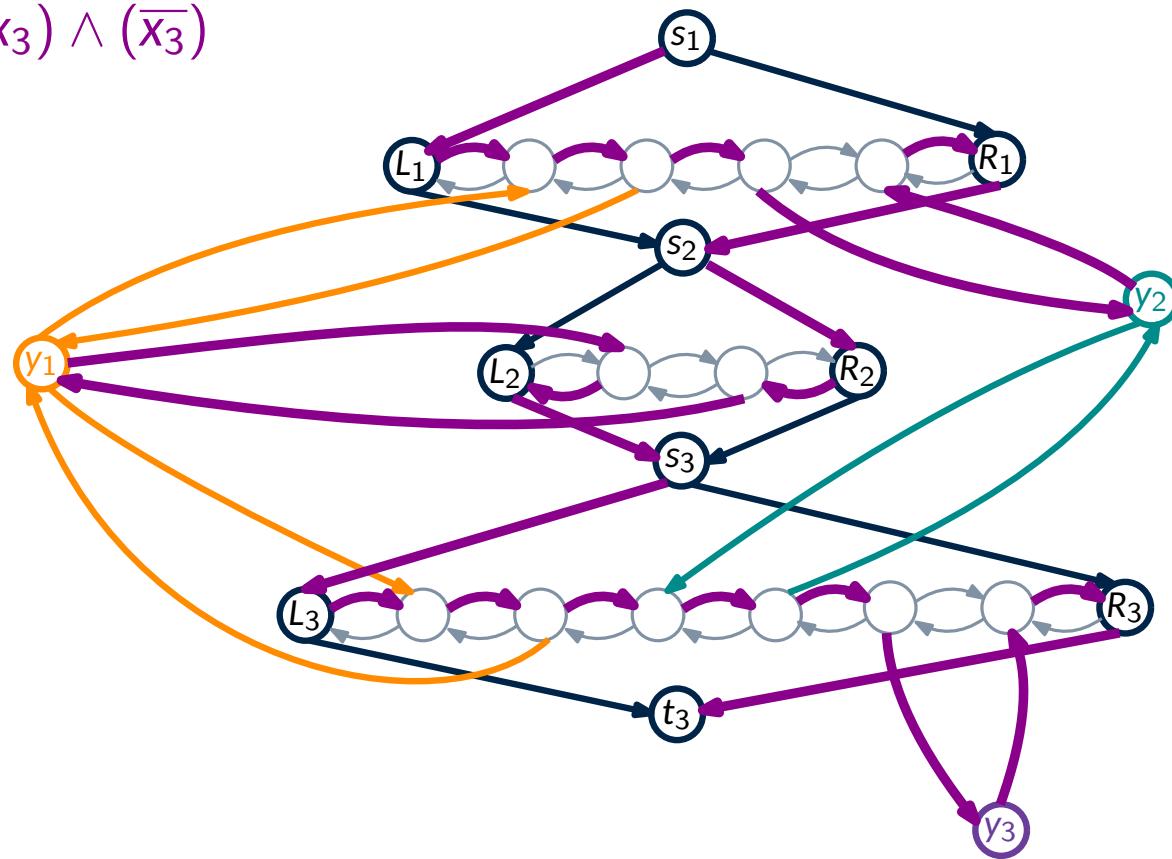
$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$



# Beispiel für die gesamte Konstruktion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$

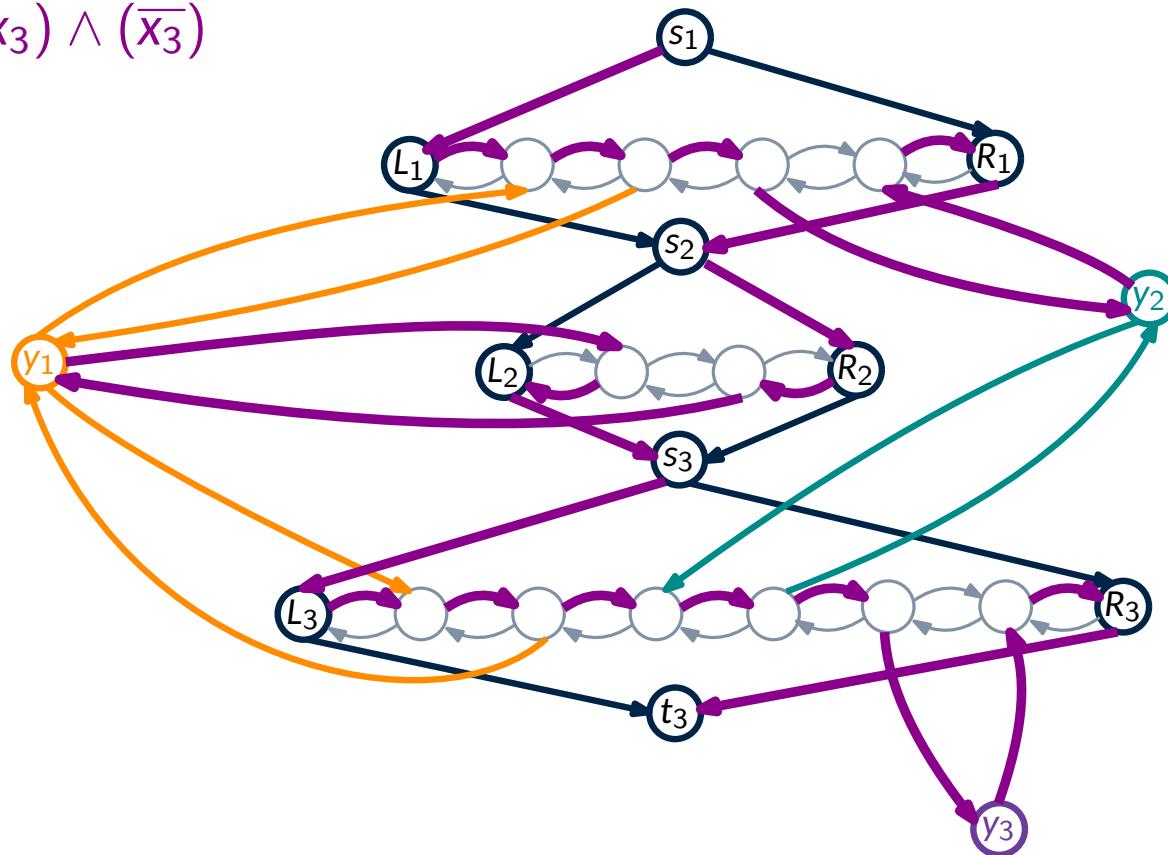
$b = (0, 1, 0)$  ist erfüllend



# Beispiel für die gesamte Konstruktion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_3})$$

$$b = (0, 1, 0) \text{ ist erfüllend}$$



**Behauptung:** 1. Es gibt einen Hamiltonpfad von  $s_1$  nach  $t_n$

$\Leftrightarrow$

$\phi$  ist erfüllbar

2. Der Graph kann in polynomieller Zeit berechnet werden

# Ungerichteter Fall

---

## Undirected Hamiltonian Path (HamPath).

**Gegeben:** Ein **ungerichteter Graph**  $G = (V, E)$ , sowie Startknoten  $s \in V$  und Zielknoten  $t \in V$ .

**Gesucht:** Gibt es einen **Hamiltonpfad** von  $s$  nach  $t$  in  $G$ ?

Ein Pfad  $p = (p_0, \dots, p_{n-1})$  von  $p_0 = s$  nach  $p_{n-1} = t$  heißt **Hamiltonpfad**, wenn:  
jeder Knoten in  $G$  wird genau einmal von  $p$  besucht

## Theorem.

Undirected Hamiltonian Path ist **NP-vollständig**.

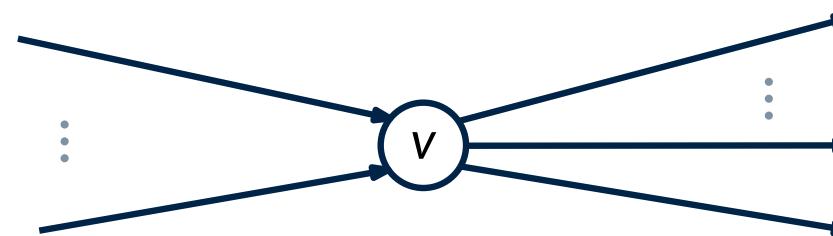
## Beweis::

1. HamPath ist in NP: Analog zu Directed Hamiltonian Path
2. Wir reduzieren von Directed Hamiltonian Path
- 3.-5.: Wir geben auf der nächsten Folie die Beweisidee, aber führen nicht alle Details aus

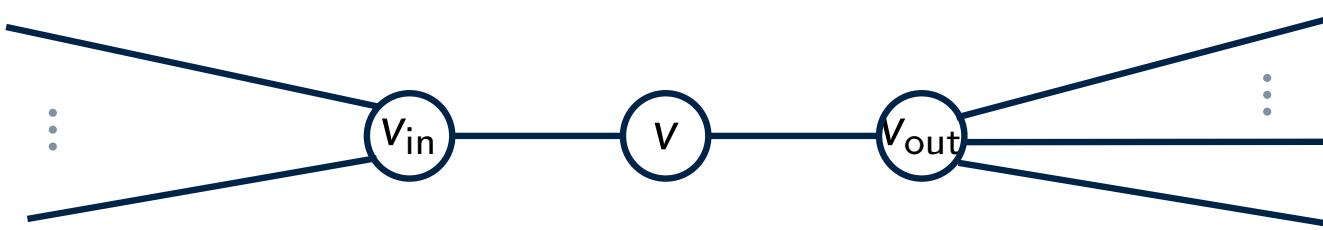
# $d\text{HamPath} \leq_P \text{HamPath}$

Idee:

Jeder Knoten  $v \in V$



wird ersetzt durch:



→ eine Kante  $(u, v)$  im ursprünglichen Graphen  $G$  wird zu einer Kante  $\{u_{out}, v_{in}\}$

→ erhalten einen ungerichteten Graphen  $G'$  mit  $3n$  Knoten und  $m + 2n$  Kanten

Eigenschaft: Für  $s \neq t$  gilt:

$G$  hat Hamiltonpfad von  $s$  nach  $t$

$\Leftrightarrow$

$G'$  hat Hamiltonpfad von  $s_{in}$  nach  $t_{out}$

Beweisidee:

In  $G'$  gilt: Wenn ich einen Knoten  $v_{in}$  bzw.  $v_{out}$  betrete,

muss ich direkt  $v$ ,  $v_{out}$  bzw.  $v$ ,  $v_{in}$  betreten, um einen Hamiltonpfad zu erhalten.

# NP-Vollständigkeit von TSP-E

---

**Entscheidungsvariante des Traveling Salesperson Problems (TSP-E).**

**Gegeben:** Ein vollständiger ungerichteter, gewichteter Graph  $G = (V, E, w)$  und  $W \in \mathbb{N}$

**Gesucht:** Gibt es eine Rundreise mit Kosten  $\leq W$ ?

**Theorem.**

TSP-E ist **NP-vollständig**.

**Beweis:** 1. TSP-E ist in NP:

**Beweisskizze:**

Das Zertifikat sei eine Kodierung eines Rundweges  $r = (r_0, \dots, r_n)$ .

Wir überprüfen, ob  $r$  ein Rundweg ist (geschlossener Pfad der Länge  $n$ )

Wir berechnen die Kosten von  $r$  und akzeptieren, wenn diese  $\leq W$  sind

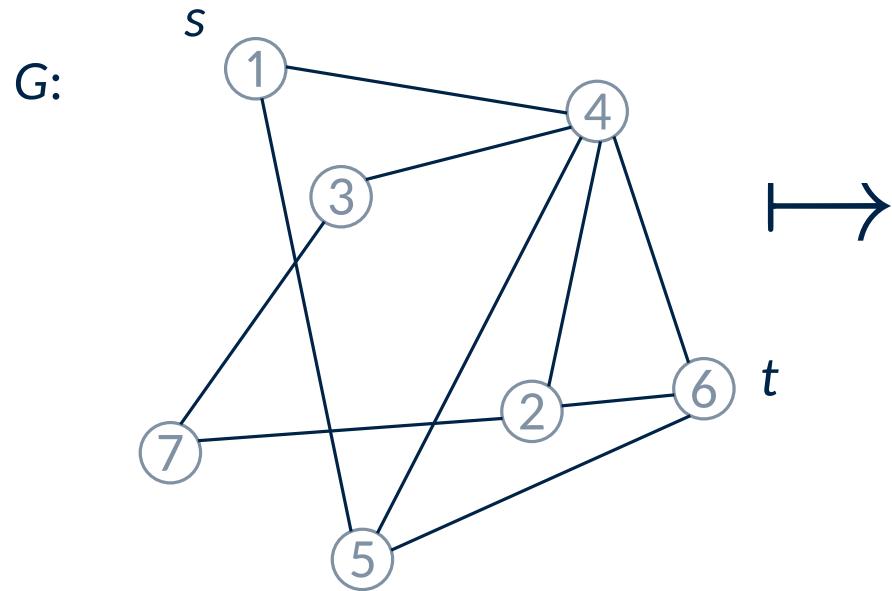
2. Wir reduzieren von Undirected Hamiltonian Path (HamPath)

**Beweisidee:** ↗ nächste Folie

# HamPath $\leq_P$ TSP-E

---

Eingabe für HamPath:



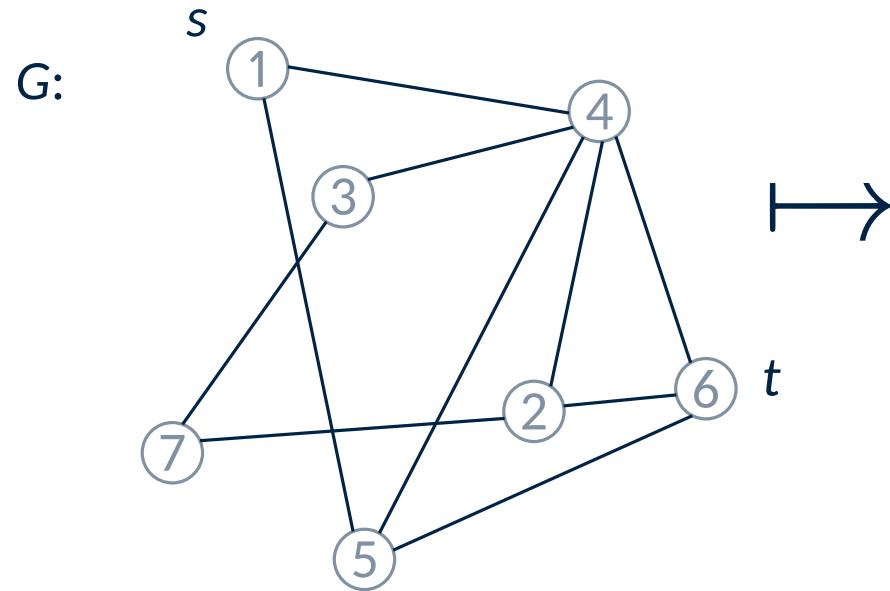
Eingabe für TSP-E:

**Frage:** Wie können wir den Graphen ändern, sodass Rundweg existiert gdw. G hat Hamiltonpfad von s nach t?

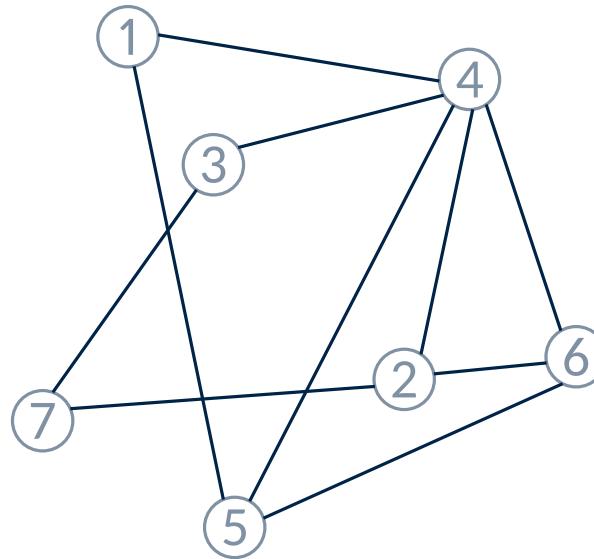
# HamPath $\leq_P$ TSP-E

---

Eingabe für HamPath:



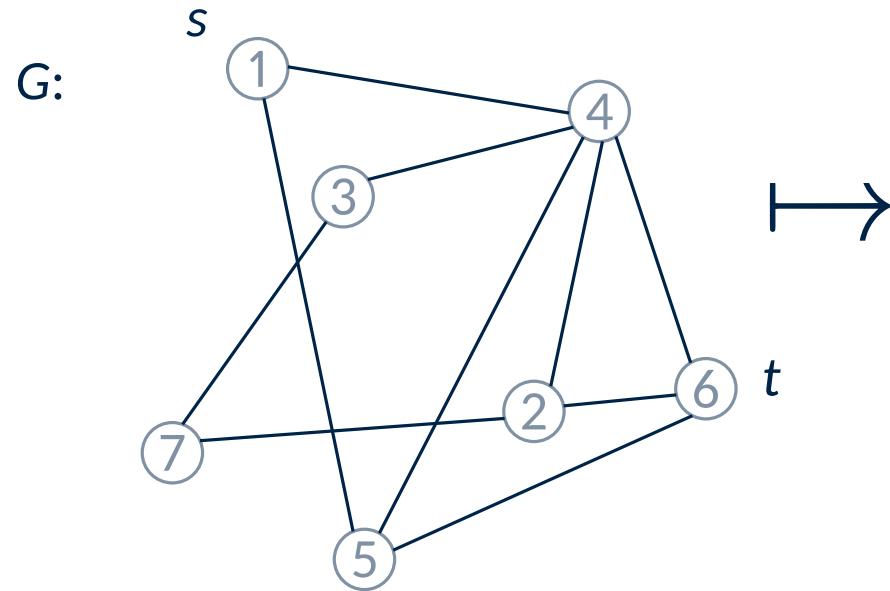
Eingabe für TSP-E:



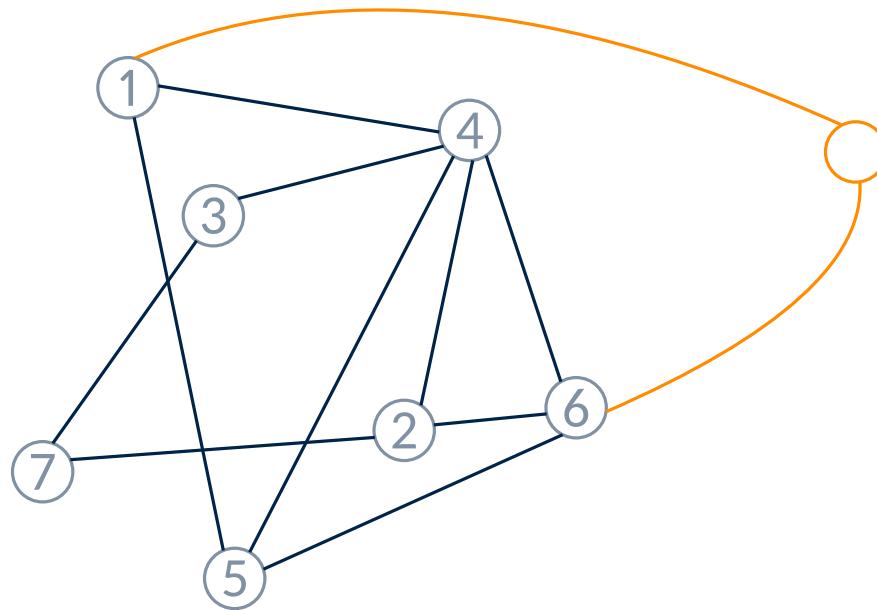
# HamPath $\leq_P$ TSP-E

---

Eingabe für HamPath:



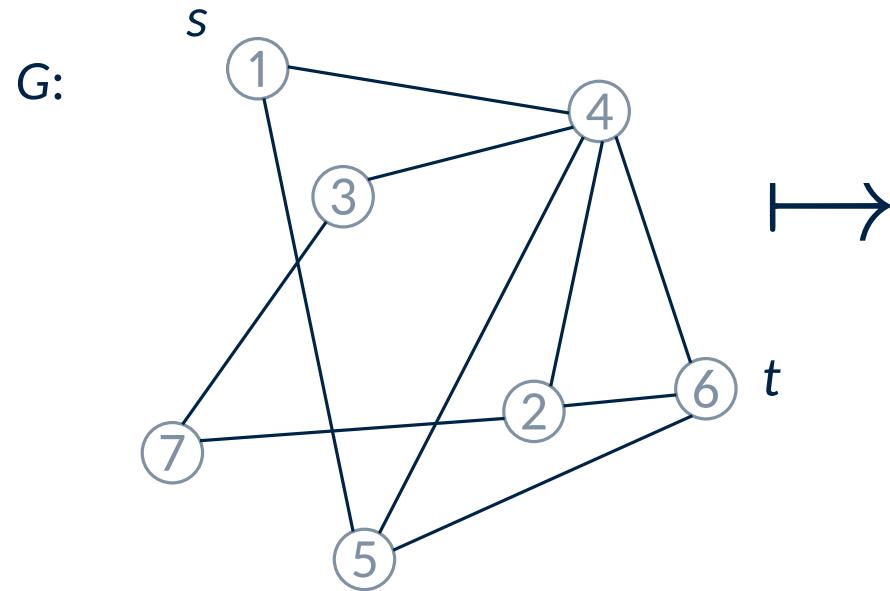
Eingabe für TSP-E:



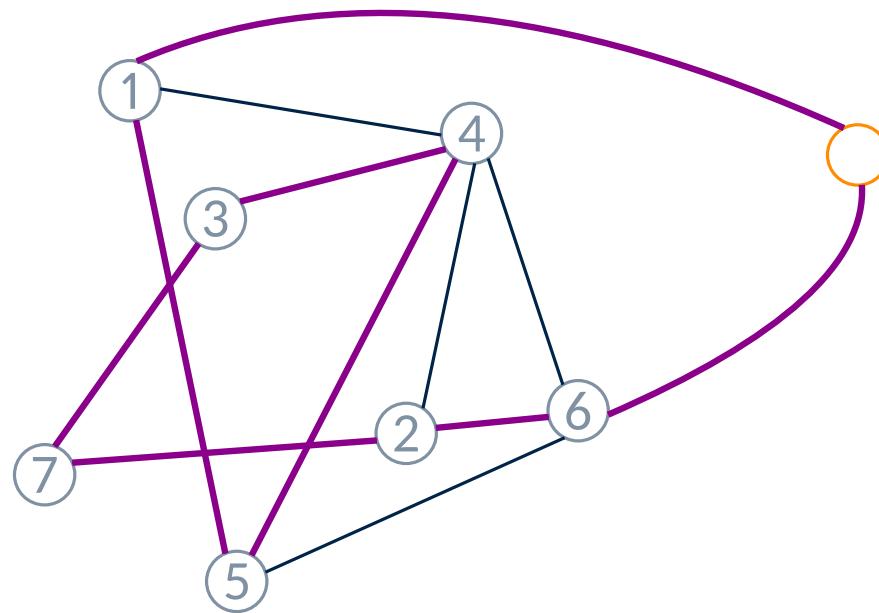
# HamPath $\leq_P$ TSP-E

---

Eingabe für HamPath:



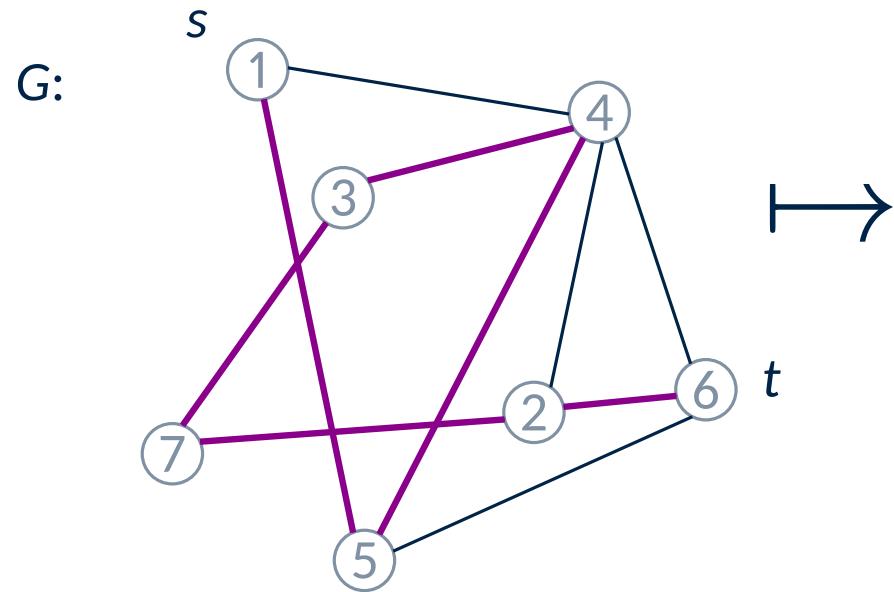
Eingabe für TSP-E:



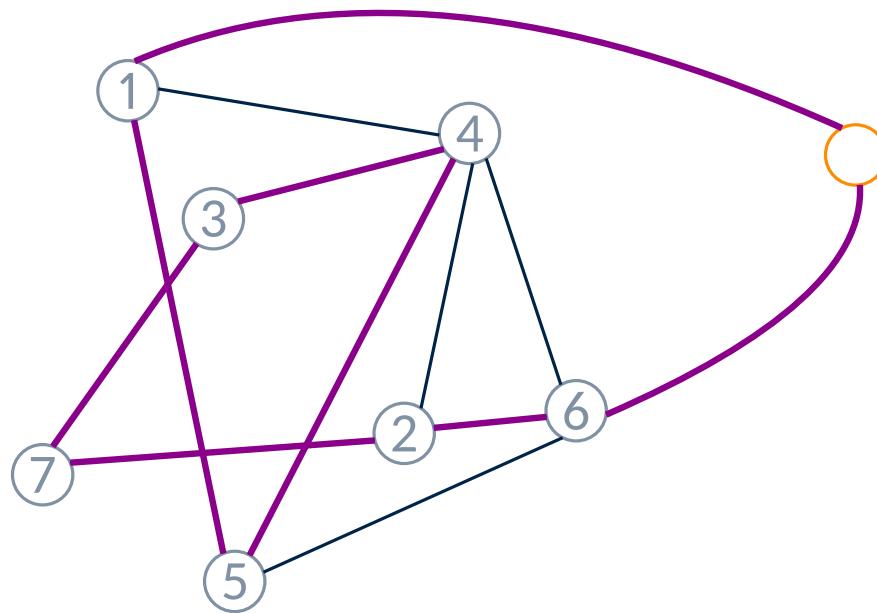
# HamPath $\leq_P$ TSP-E

---

Eingabe für HamPath:

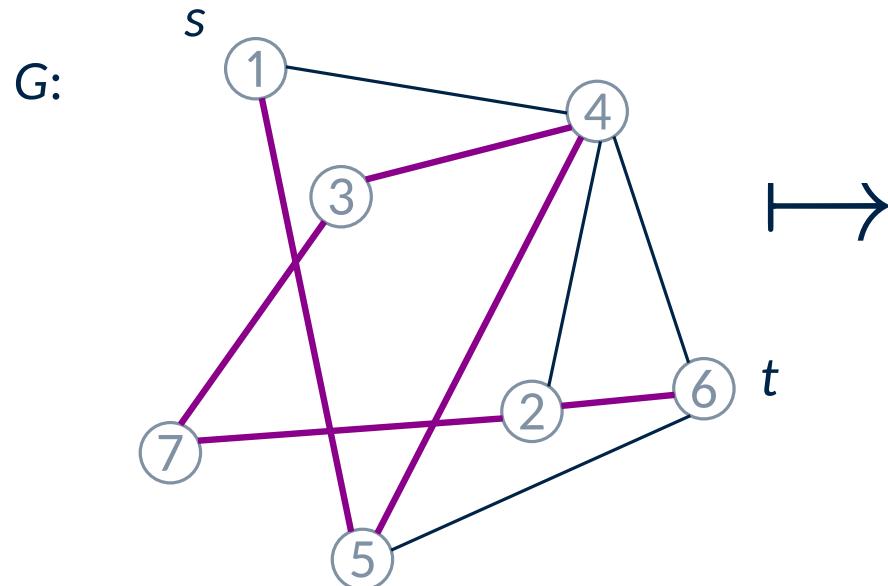


Eingabe für TSP-E:

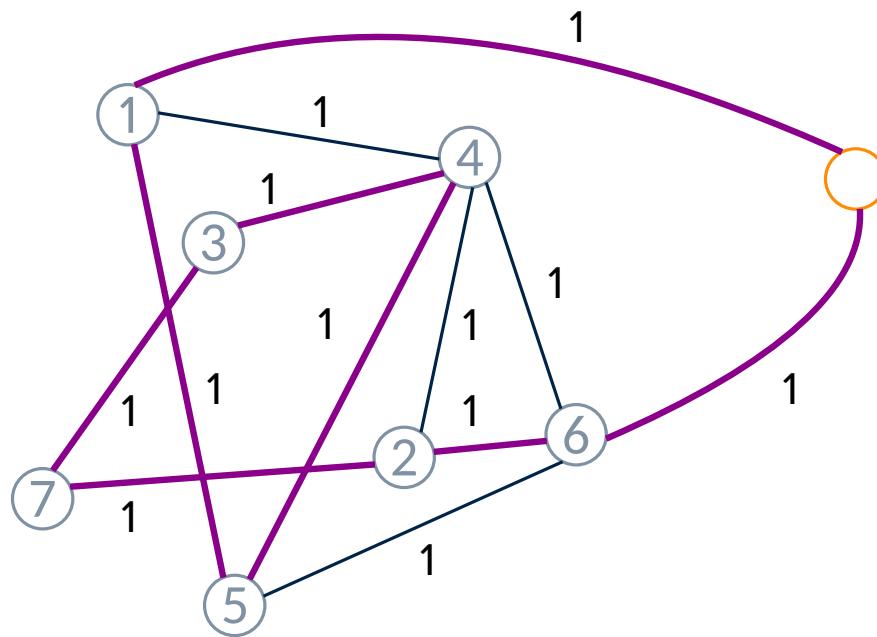


# HamPath $\leq_P$ TSP-E

Eingabe für HamPath:



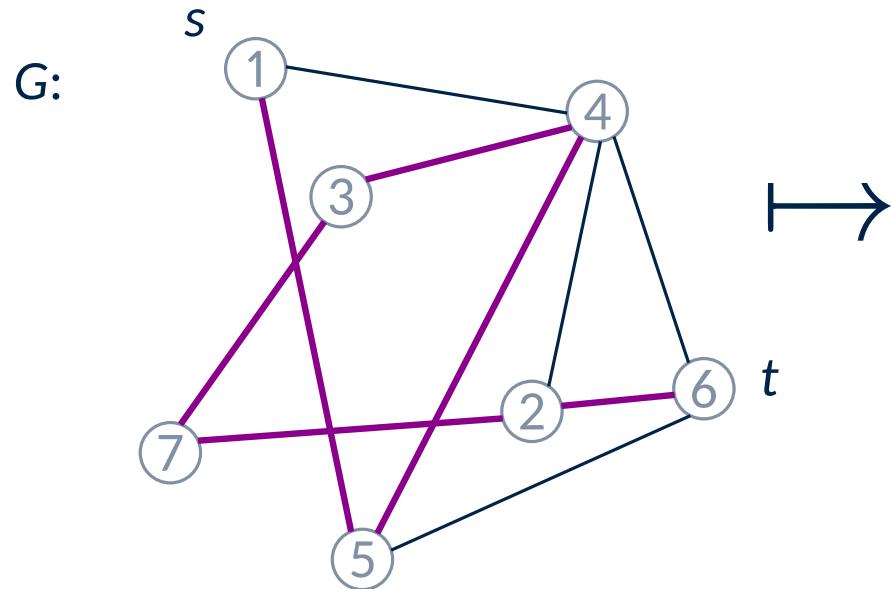
Eingabe für TSP-E:



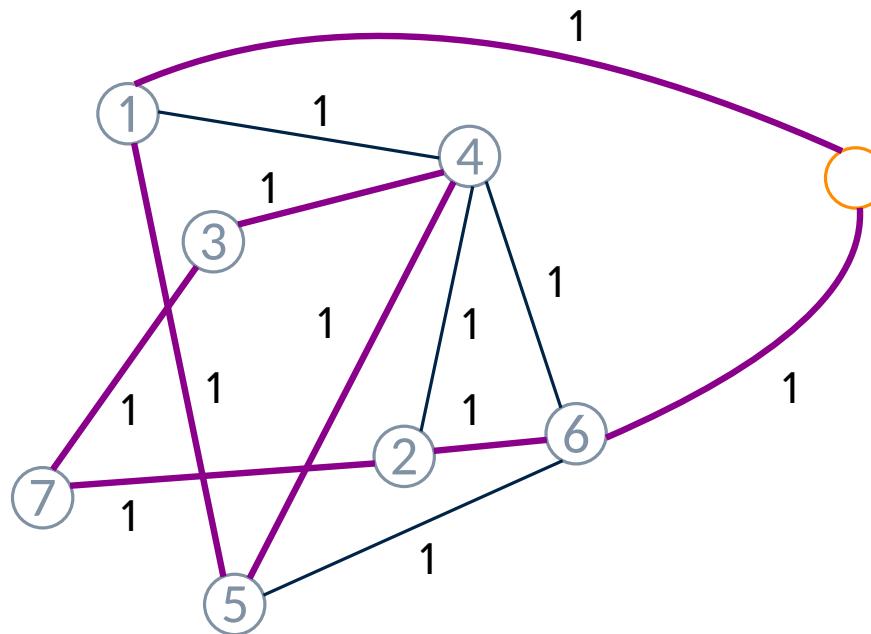
· alle gezeigten Kanten bekommen Kantengewicht 1

# HamPath $\leq_P$ TSP-E

Eingabe für HamPath:

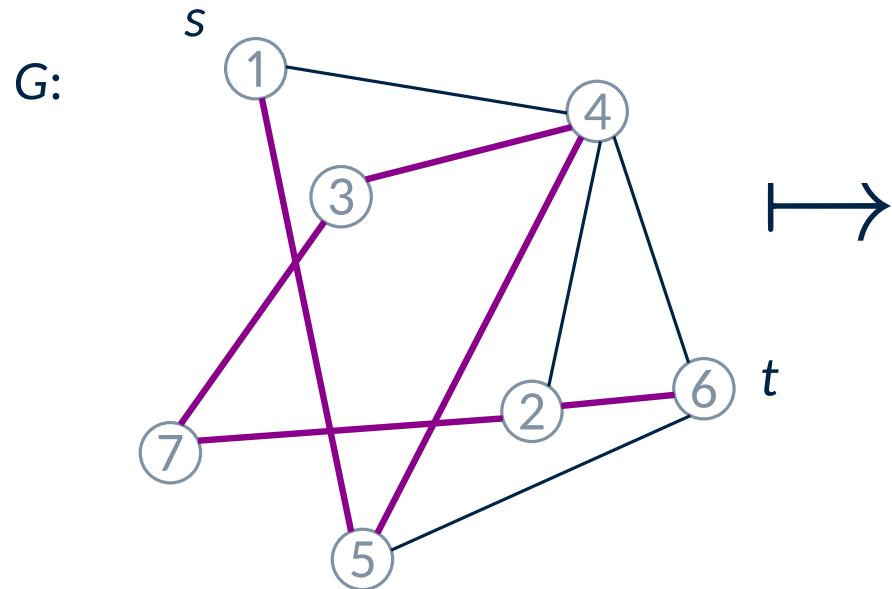


Eingabe für TSP-E:

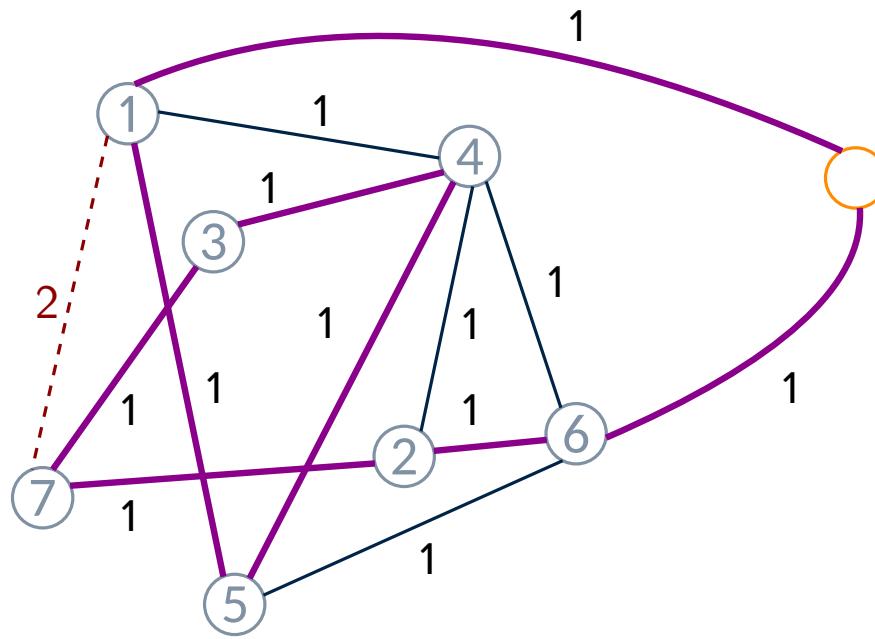


# HamPath $\leq_P$ TSP-E

Eingabe für HamPath:



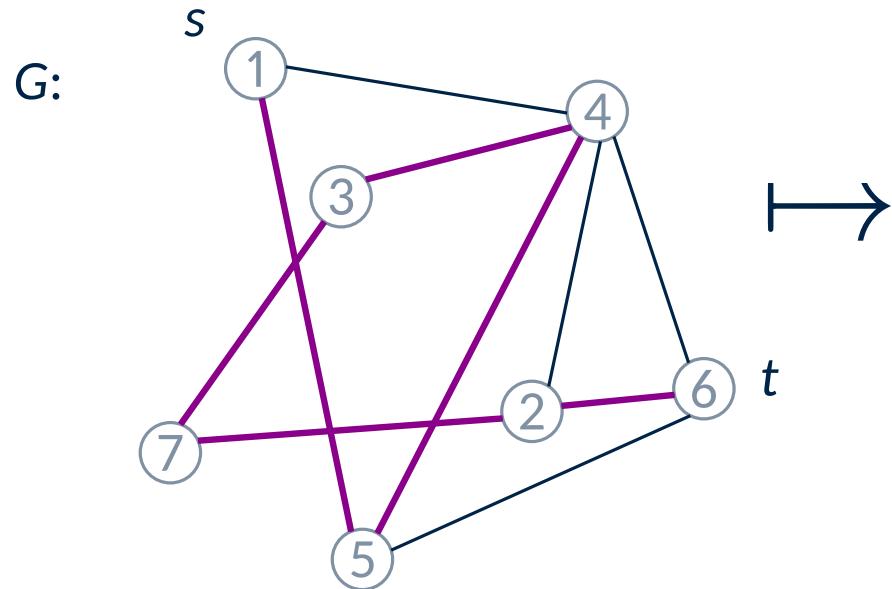
Eingabe für TSP-E:



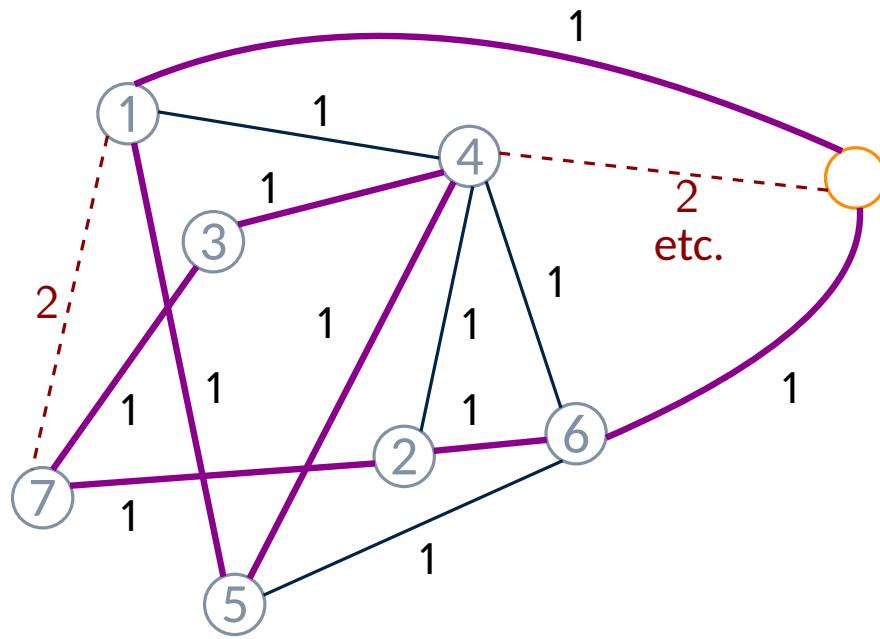
- alle gezeigten Kanten bekommen Kantengewicht 1
- alle nicht gezeigten Kanten bekommen Kantengewicht 2

# HamPath $\leq_P$ TSP-E

Eingabe für HamPath:

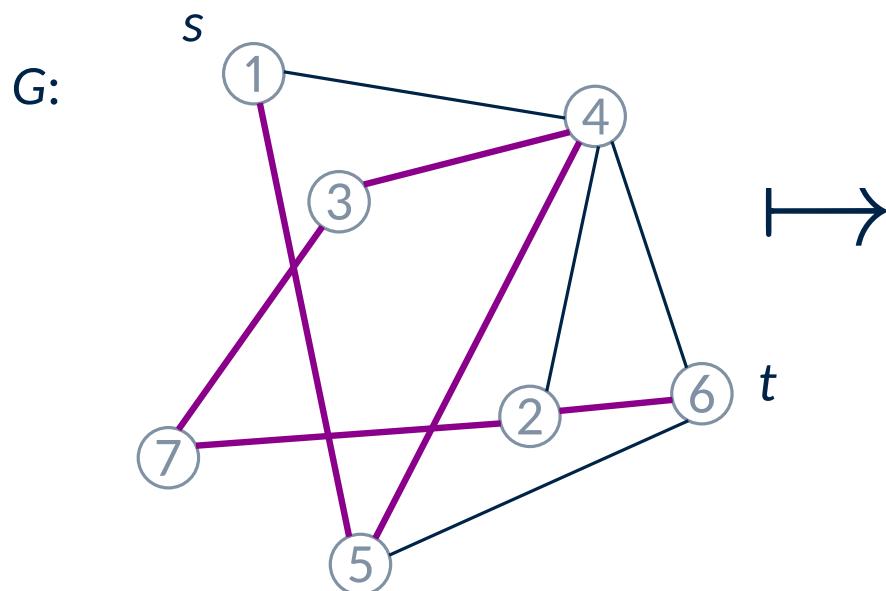


Eingabe für TSP-E:

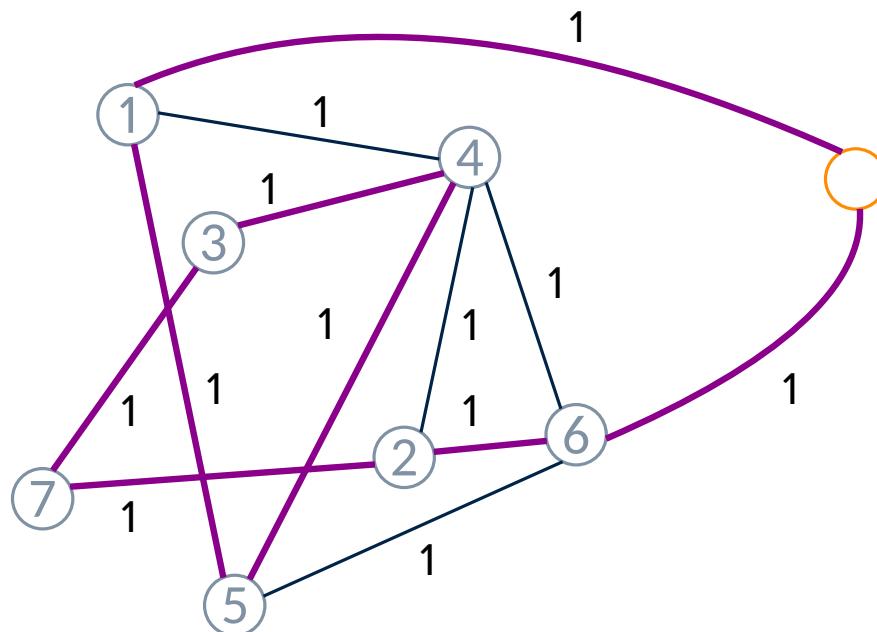


# HamPath $\leq_P$ TSP-E

Eingabe für HamPath:



Eingabe für TSP-E:



- alle gezeigten Kanten bekommen Kantengewicht 1
- alle nicht gezeigten Kanten bekommen Kantengewicht 2

Im neuen Graphen gibt es einen Rundweg mit Kosten  $n + 1$

$\Leftrightarrow$

Im ursprünglichen Graphen gibt es einen Hamiltonpfad von  $s$  nach  $t$

# **Subset Sum & Verwandschaft**

# Subset Sum

---

**Subset Sum (SUBSET-SUM).**

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten** sowie ein **Zielwert**  $t \in \mathbb{N}$

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau  $t$  ist?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$w(S) := \sum_{i \in S} w_i = t?$$

8

10

13

15

21

26

27

$t = 46$

# Subset Sum

---

**Subset Sum (SUBSET-SUM).**

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten** sowie ein **Zielwert**  $t \in \mathbb{N}$

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau  $t$  ist?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$w(S) := \sum_{i \in S} w_i = t?$$

Frage:

Ja-Instanz?      8 kg    10 kg    13 kg    15 kg    21 kg    26 kg    27 kg         $t = 46$  kg

# Subset Sum

---

**Subset Sum (SUBSET-SUM).**

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten** sowie ein **Zielwert**  $t \in \mathbb{N}$

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau  $t$  ist?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$w(S) := \sum_{i \in S} w_i = t?$$

Frage:

8 kg    10 kg    13 kg    15 kg    21 kg    26 kg    27 kg     $t = 46$  kg

Ja-Instanz?    1 €    2 €    5 €    5 €    20 €    50 €     $t = 14$  €

# Subset Sum

---

**Subset Sum (SUBSET-SUM).**

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten** sowie ein **Zielwert**  $t \in \mathbb{N}$

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau  $t$  ist?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$w(S) := \sum_{i \in S} w_i = t?$$

Frage:

8 kg	10 kg	13 kg	15 kg	21 kg	26 kg	27 kg	$t = 46 \text{ kg}$
------	-------	-------	-------	-------	-------	-------	---------------------

Ja-Instanz?	1 €	2 €	5 €	5 €	20 €	50 €	$t = 14 \text{ €}$
-------------	-----	-----	-----	-----	------	------	--------------------

**Theorem.**

Subset Sum ist **NP-vollständig**.

**Beweis:** 1. SubsetSum ist in NP.

Frage: Warum?

# Subset Sum

---

**Subset Sum (SUBSET-SUM).**

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten** sowie ein **Zielwert**  $t \in \mathbb{N}$

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau  $t$  ist?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$w(S) := \sum_{i \in S} w_i = t?$$

Frage:

8 kg    10 kg    13 kg    15 kg    21 kg    26 kg    27 kg     $t = 46$  kg

Ja-Instanz?    1 €    2 €    5 €    5 €    20 €    50 €     $t = 14$  €

**Theorem.**

Subset Sum ist **NP-vollständig**.

**Beweis:** 1. SubsetSum ist in NP.

**Beweisidee:** Zertifikat  $z \in \{0, 1\}^N$  zeigt die gesuchte Auswahl  $S$  der Gewichtswerte an  
Verifiziererin berechnet  $w(S)$  und akzeptiert gdw.  $w(S) = t$

# Subset Sum

**Subset Sum (SUBSET-SUM).**

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten** sowie ein **Zielwert**  $t \in \mathbb{N}$

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau  $t$  ist?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$w(S) := \sum_{i \in S} w_i = t?$$

Frage:

8 kg	10 kg	13 kg	15 kg	21 kg	26 kg	27 kg	$t = 46 \text{ kg}$
------	-------	-------	-------	-------	-------	-------	---------------------

Ja-Instanz?	1 €	2 €	5 €	5 €	20 €	50 €	$t = 14 \text{ €}$
-------------	-----	-----	-----	-----	------	------	--------------------

**Theorem.**

Subset Sum ist **NP-vollständig**.

**Beweis:** 1. SubsetSum ist in NP.

**Beweisidee:** Zertifikat  $z \in \{0, 1\}^N$  zeigt die gesuchte Auswahl  $S$  der Gewichtswerte an  
Verifiziererin berechnet  $w(S)$  und akzeptiert gdw.  $w(S) = t$

2. Wir reduzieren von 3SAT. Wollen also zeigen:  $\text{3SAT} \leq_P \text{SubsetSum}$
3. Nächste Folien: Beschreibung einer Konstruktion von Gewichtswerten und Zielwert für eine gegebene 3SAT-Formel  $\phi$

# Große Struktur

Gegeben eine 3SAT-Formel  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$  mit  $n$  Variablen und  $m$  Klauseln  
werden wir Gewichtswerte konstruieren, die wir als  $(n + m)$ -stellige Dezimalzahlen repräsentieren.

→ jeder Gewichtswert ist der Form

$$w = w(1)w(2) \dots w(n) \boxed{w(n+1) \dots w(n+m)}_{10}$$

↓ Variablenziffern      ↓ Klauselziffern      ↑  $w(i)$  - i. Ziffer von  $w$

**Literalwerte:**

Für jede Variable  $x_i$  erzeugen wir:

- ein **Literalwert**  $w_i$  für das Literal  $x_i$ , sowie
- ein **Literalwert**  $\bar{w}_i$  für das Literal  $\bar{x}_i$ ,

Für die Variablenziffern  $j = 1, \dots, n$  definieren wir:

$$w_i(j) = \bar{w}_i(j) = \begin{cases} 1 & \text{wenn } j = i, \\ 0 & \text{ansonsten} \end{cases}$$

Für die Klauselziffern  $n + j$  mit  $j = 1, \dots, m$  definieren wir:

$$w_i(n+j) = \begin{cases} 1 & \text{wenn } x_i \text{ ein Literal von } C_j \text{ ist,} \\ 0 & \text{ansonsten} \end{cases} \quad \mid \quad \bar{w}_i(n+j) = \begin{cases} 1 & \text{wenn } \bar{x}_i \text{ ein Literal von } C_j \text{ ist,} \\ 0 & \text{ansonsten} \end{cases}$$

# Beispiel der Literalwerte

---

Beispiel:

$$\phi = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

# Auffüllwerte und Zielwert

Für jede Klausel  $C_j$  führen wir zwei zusätzliche **identische** Gewichtswerte  $h_j$  und  $h'_j$  ein:

$$h_j(k) = h'_j(k) = \begin{cases} 1 & \text{wenn } k = n + j, \\ 0 & \text{ansonsten} \end{cases}$$

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

Der **Zielwert**  $t$  ergibt sich aus 1-Ziffern für jede Variable und 3-Ziffern für jede Klausel:

$$t(k) = \begin{cases} 1 & \text{wenn } k \leq n, \\ 3 & \text{ansonsten} \end{cases}$$

$$t =$$

1	1	1	3	3	3	3
---	---	---	---	---	---	---

# Gesamte Reduktion

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$   
" $\Rightarrow$ " Es sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$   $b = (0, 1, 0)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion

---

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " Es sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

Für jedes  $i \in \{1, \dots, n\}$  wählen wir  $w_i$  wenn  $b_i = 1$  und  $\bar{w}_i$  wenn  $b_i = 0$

$\rightarrow$  dies liefert eine Teilauswahl mit Summe  $w$

**Bemerkung:** ·  $w(i) = 1$  für alle  $i \in \{1, \dots, n\}$

·  $w(n+j) \in \{1, 2, 3\}$  für alle  $j \in \{1, \dots, m\}$  denn  $b$  erfüllt zwischen 1 und 3 Literale von  $C_j$

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$   $b = (0, 1, 0)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1
$\rightarrow w =$	1	1	1	2	1	3	2

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion

---

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " Es sei  $b \in \{0, 1\}^n$  eine erfüllende Belegung für  $\phi$ .

Für jedes  $i \in \{1, \dots, n\}$  wählen wir  $w_i$  wenn  $b_i = 1$  und  $\bar{w}_i$  wenn  $b_i = 0$

$\rightarrow$  dies liefert eine Teilauswahl mit Summe  $w$

**Bemerkung:** ·  $w(i) = 1$  für alle  $i \in \{1, \dots, n\}$

·  $w(n+j) \in \{1, 2, 3\}$  für alle  $j \in \{1, \dots, m\}$  denn  $b$  erfüllt zwischen 1 und 3 Literale von  $C_j$

Für jede Klausel  $C_j$  wählen wir zusätzlich  $h_j$  aus wenn  $w(n+j) \leq 2$  sowie  $h'_j$  wenn  $w(n+j) = 1$

$\rightarrow$  dies liefert unsere Gesamtauswahl mit Summe  $t$  ✓

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$   $b = (0, 1, 0)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1
$\rightarrow w =$	1	1	1	2	1	3	2

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion: Korrektheit II

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " ✓

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

32 - 1  $t =$

1	1	1	3	3	3	3
---	---	---	---	---	---	---

# Gesamte Reduktion: Korrektheit II

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " ✓

" $\Leftarrow$ " Es sei  $S$  eine beliebige Auswahl der Gewichtswerte mit Summe  $t$

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion: Korrektheit II

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " ✓

" $\Leftarrow$ " Es sei  $S$  eine beliebige Auswahl der Gewichtswerte mit Summe  $t$

**Wichtige Bemerkung:** Für jede Ziffer  $k \in \{1, \dots, n+m\}$  muss gelten:  $t(k) = \sum_{w \in S} w(k)$

→ nicht offensichtlich, da beim Aufaddieren Überträge entstehen könnten!

Für uns allerdings der Fall, denn: für jede Ziffer  $k$  gibt es  $\leq 5$  Gewichte  $w$  mit  $w(k) = 1$ .  $\Rightarrow$  keine Überträge

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

$$32 - 5 \quad t =$$

1	1	1	3	3	3	3
---	---	---	---	---	---	---

# Gesamte Reduktion: Korrektheit II

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " ✓

" $\Leftarrow$ " Es sei  $S$  eine beliebige Auswahl der Gewichtswerte mit Summe  $t$

**Wichtige Bemerkung:** Für jede Ziffer  $k \in \{1, \dots, n+m\}$  muss gelten:  $t(k) = \sum_{w \in S} w(k)$

Wir können nun eine erfüllende Belegung ablesen:

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

$$32 - 6 \quad t =$$

$$\begin{array}{cccc} 1 & 1 & 1 & \\ 3 & 3 & 3 & 3 \end{array}$$

# Gesamte Reduktion: Korrektheit II

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " ✓

" $\Leftarrow$ " Es sei  $S$  eine beliebige Auswahl der Gewichtswerte mit Summe  $t$

**Wichtige Bemerkung:** Für jede Ziffer  $k \in \{1, \dots, n+m\}$  muss gelten:  $t(k) = \sum_{w \in S} w(k)$

Wir können nun eine erfüllende Belegung ablesen:

- $S$  wählt genau ein Literalwert für jedes  $x_i$  (denn  $\sum_{w \in S} w(i) = 1$ )  $\rightsquigarrow$  Belegung  $b$

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$   $b = (1, 0, 0)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion: Korrektheit II

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Korrektheit:**  $\phi$  ist erfüllbar  $\Leftrightarrow$  es existiert Auswahl der Gewichtswerte mit Summe  $t$

" $\Rightarrow$ " ✓

" $\Leftarrow$ " Es sei  $S$  eine beliebige Auswahl der Gewichtswerte mit Summe  $t$

**Wichtige Bemerkung:** Für jede Ziffer  $k \in \{1, \dots, n+m\}$  muss gelten:  $t(k) = \sum_{w \in S} w(k)$

Wir können nun eine erfüllende Belegung ablesen:

- $S$  wählt genau ein Literalwert für jedes  $x_i$  (denn  $\sum_{w \in S} w(i) = 1$ )  $\rightsquigarrow$  Belegung  $b$
- diese Literalwerte müssen in Ziffer  $n+j$  auf  $\geq 1$  aufsummieren, ansonsten ist  $\sum_{w \in S} w(n+j) < 3$   
 $\rightarrow b$  erfüllt jede Klausel  $C_j$

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$   $b = (1, 0, 0)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# Gesamte Reduktion: Laufzeit

---

Gegeben ein 3-KNF-Formel  $\phi$  mit  $n$  Variablen und  $m$  Klauseln

berechnen wir die Gewichtswerte  $w_1, \bar{w}_1, \dots, w_n, \bar{w}_n, h_1, h'_1, \dots, h_m, h'_m$  sowie den Zielwert  $t$ .

**Laufzeit:** · Anzahl an Gewichten:  $2n + 2m$

- jedes Gewicht und der Zielwert  $t$  haben jeweils  $n + m$  Ziffern als Dezimalzahl  
→  $O(n + m)$  Bits in Binärdarstellung

Die Ausgabe der Reduktion kann in Zeit  $O((n + m)^2)$  berechnet werden

→ polynomiell beschränkte Laufzeit

**Beispiel:**  $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$        $b = (1, 0, 0)$

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$w_1 =$	1	0	0	1	1	0	0
$\bar{w}_1 =$	1	0	0	0	0	1	1
$w_2 =$	0	1	0	1	0	1	0
$\bar{w}_2 =$	0	1	0	0	1	0	1
$w_3 =$	0	0	1	0	0	0	0
$\bar{w}_3 =$	0	0	1	1	1	1	1

	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$h_1 =$	0	0	0	1	0	0	0
$h'_1 =$	0	0	0	1	0	0	0
$h_2 =$	0	0	0	0	1	0	0
$h'_2 =$	0	0	0	0	1	0	0
$h_3 =$	0	0	0	0	0	1	0
$h'_3 =$	0	0	0	0	0	1	0
$h_4 =$	0	0	0	0	0	0	1
$h'_4 =$	0	0	0	0	0	0	1

# DP-Algorithmus für Subset Sum

Wir entwerfen einen Algorithmus für das Subset-Sum-Problem via **Dynamischer Programmierung**:

**Ansatz:** Wir erstellen eine Tabelle  $T[i, v]$  indiziert durch  $i \in \{0, \dots, n\}$  und  $v \in \{0, \dots, t\}$ :

$$T[i, v] = \begin{cases} 1 & \text{wenn es } S \subseteq \{1, \dots, i\} \text{ gibt mit } \sum_{i \in S} w_i = v \\ 0 & \text{ansonsten} \end{cases}$$

Auswahl der ersten  $i$  Gewichtswerte mit Summe  $v$

**Bemerkung:** Für  $i = 0$  gilt:  $T[0, 0] = 1$  und  $T[0, v] = 0$  für alle  $v > 0$ .

Für  $i > 0$  gilt:

$$T[i, v] = \max\{T[i - 1, v], T[i - 1, v - w_i]\}$$

Hierbei interpretieren wir  $T[i - 1, v']$  mit  $v' < 0$  als 0

$w_i$	3	1	1
$i$	0	1	2
$v$	0	1	2
	3		
	4		
	5		

Wenn wir alle Werte  $T[i - 1, v]$  mit  $v \in \{0, \dots, t\}$  kennen, können wir alle Werte  $T[i, v]$  mit  $v \in \{0, \dots, t\}$  in Zeit  $O(t)$  berechnen

# DP-Algorithmus für Subset Sum

Wir entwerfen einen Algorithmus für das Subset-Sum-Problem via **Dynamischer Programmierung**:  
↗ nächste VL

**Ansatz:** Wir erstellen eine Tabelle  $T[i, v]$  indiziert durch  $i \in \{0, \dots, n\}$  und  $v \in \{0, \dots, t\}$ :

$$T[i, v] = \begin{cases} 1 & \text{wenn es } S \subseteq \{1, \dots, i\} \text{ gibt mit } \sum_{i \in S} w_i = v \\ 0 & \text{ansonsten} \end{cases}$$

Auswahl der ersten  $i$  Gewichtswerte mit Summe  $v$

**Bemerkung:** Für  $i = 0$  gilt:  $T[0, 0] = 1$  und  $T[0, v] = 0$  für alle  $v > 0$ .

Für  $i > 0$  gilt:

$$T[i, v] = \max\{T[i - 1, v], T[i - 1, v - w_i]\}$$

Hierbei interpretieren wir  $T[i - 1, v']$  mit  $v' < 0$  als 0

$w_i$	3	1	1
$i$	0	1	2
$v$	0	1	0
	1	0	
	2	0	
	3	0	
	4	0	
	5	0	

Wenn wir alle Werte  $T[i - 1, v]$  mit  $v \in \{0, \dots, t\}$  kennen,  
können wir alle Werte  $T[i, v]$  mit  $v \in \{0, \dots, t\}$  in Zeit  $O(t)$  berechnen

# DP-Algorithmus für Subset Sum

Wir entwerfen einen Algorithmus für das Subset-Sum-Problem via **Dynamischer Programmierung**:  
↗ nächste VL

**Ansatz:** Wir erstellen eine Tabelle  $T[i, v]$  indiziert durch  $i \in \{0, \dots, n\}$  und  $v \in \{0, \dots, t\}$ :

$$T[i, v] = \begin{cases} 1 & \text{wenn es } S \subseteq \{1, \dots, i\} \text{ gibt mit } \sum_{i \in S} w_i = v \\ 0 & \text{ansonsten} \end{cases}$$

Auswahl der ersten  $i$  Gewichtswerte mit Summe  $v$

**Bemerkung:** Für  $i = 0$  gilt:  $T[0, 0] = 1$  und  $T[0, v] = 0$  für alle  $v > 0$ .

Für  $i > 0$  gilt:

$$T[i, v] = \max\{T[i - 1, v], T[i - 1, v - w_i]\}$$

Hierbei interpretieren wir  $T[i - 1, v']$  mit  $v' < 0$  als 0

	$w_i$	3	1	1
	$i$	0	1	2
$v$	0	1	1	
	1	0	0	
	2	0	0	
	3	0	1	
	4	0	0	
	5	0	0	

Wenn wir alle Werte  $T[i - 1, v]$  mit  $v \in \{0, \dots, t\}$  kennen,  
können wir alle Werte  $T[i, v]$  mit  $v \in \{0, \dots, t\}$  in Zeit  $O(t)$  berechnen

# DP-Algorithmus für Subset Sum

Wir entwerfen einen Algorithmus für das Subset-Sum-Problem via **Dynamischer Programmierung**:

↗ nächste VL

**Ansatz:** Wir erstellen eine Tabelle  $T[i, v]$  indiziert durch  $i \in \{0, \dots, n\}$  und  $v \in \{0, \dots, t\}$ :

$$T[i, v] = \begin{cases} 1 & \text{wenn es } S \subseteq \{1, \dots, i\} \text{ gibt mit } \sum_{i \in S} w_i = v \\ 0 & \text{ansonsten} \end{cases}$$

Auswahl der ersten  $i$  Gewichtswerte mit Summe  $v$

**Bemerkung:** Für  $i = 0$  gilt:  $T[0, 0] = 1$  und  $T[0, v] = 0$  für alle  $v > 0$ .

Für  $i > 0$  gilt:

$$T[i, v] = \max\{T[i - 1, v], T[i - 1, v - w_i]\}$$

Hierbei interpretieren wir  $T[i - 1, v']$  mit  $v' < 0$  als 0

$w_i$	3	1	1
$i$	0	1	2
$v$	1	0	1
0	1	1	1
1	0	0	1
2	0	0	0
3	0	1	1
4	0	0	1
5	0	0	0

Wenn wir alle Werte  $T[i - 1, v]$  mit  $v \in \{0, \dots, t\}$  kennen,  
können wir alle Werte  $T[i, v]$  mit  $v \in \{0, \dots, t\}$  in Zeit  $O(t)$  berechnen

# DP-Algorithmus für Subset Sum

Wir entwerfen einen Algorithmus für das Subset-Sum-Problem via **Dynamischer Programmierung**:

↗ nächste VL

**Ansatz:** Wir erstellen eine Tabelle  $T[i, v]$  indiziert durch  $i \in \{0, \dots, n\}$  und  $v \in \{0, \dots, t\}$ :

$$T[i, v] = \begin{cases} 1 & \text{wenn es } S \subseteq \{1, \dots, i\} \text{ gibt mit } \sum_{i \in S} w_i = v \\ 0 & \text{ansonsten} \end{cases}$$

Auswahl der ersten  $i$  Gewichtswerte mit Summe  $v$

**Bemerkung:** Für  $i = 0$  gilt:  $T[0, 0] = 1$  und  $T[0, v] = 0$  für alle  $v > 0$ .

Für  $i > 0$  gilt:

$$T[i, v] = \max\{T[i - 1, v], T[i - 1, v - w_i]\}$$

Hierbei interpretieren wir  $T[i - 1, v']$  mit  $v' < 0$  als 0

$w_i$	3	1	1	
$i$	0	1	2	3
$v$	1	0	0	1
0	1	1	1	1
1	0	0	1	1
2	0	0	0	1
3	0	1	1	1
4	0	0	1	1
5	0	0	0	1

Wenn wir alle Werte  $T[i - 1, v]$  mit  $v \in \{0, \dots, t\}$  kennen,  
können wir alle Werte  $T[i, v]$  mit  $v \in \{0, \dots, t\}$  in Zeit  $O(t)$  berechnen

# DP-Algorithmus für Subset Sum

Wir entwerfen einen Algorithmus für das Subset-Sum-Problem via **Dynamischer Programmierung**:

**Ansatz:** Wir erstellen eine Tabelle  $T[i, v]$  indiziert durch  $i \in \{0, \dots, n\}$  und  $v \in \{0, \dots, t\}$ :

$$T[i, v] = \begin{cases} 1 & \text{wenn es } S \subseteq \{1, \dots, i\} \text{ gibt mit } \sum_{i \in S} w_i = v \\ 0 & \text{ansonsten} \end{cases}$$

Auswahl der ersten  $i$  Gewichtswerte mit Summe  $v$

**Bemerkung:** Für  $i = 0$  gilt:  $T[0, 0] = 1$  und  $T[0, v] = 0$  für alle  $v > 0$ .

Für  $i > 0$  gilt:

$$T[i, v] = \max\{T[i - 1, v], T[i - 1, v - w_i]\}$$

Hierbei interpretieren wir  $T[i - 1, v']$  mit  $v' < 0$  als 0

$w_i$	3	1	1	
$i$	0	1	2	3
$v$	1	0	0	1
0	1	1	1	1
1	0	0	1	1
2	0	0	0	1
3	0	1	1	1
4	0	0	1	1
5	0	0	0	1

Wenn wir alle Werte  $T[i - 1, v]$  mit  $v \in \{0, \dots, t\}$  kennen,  
können wir alle Werte  $T[i, v]$  mit  $v \in \{0, \dots, t\}$  in Zeit  $O(t)$  berechnen

⇒ Wir können  $T[n, t]$  in Zeit  $O(nt)$  berechnen und damit die Subset-Sum-Instanz entscheiden.

# Pseudopolynomielle Algorithmen

---

Wir können eine Subset-Sum-Instanz  $w_1, \dots, w_n$  mit Zielwert  $t$  in Zeit  $O(nt)$  lösen.

**Frage:** Warum zeigt das nicht, dass  $P = NP$ ?  
SubsetSum ist doch NP-vollständig?

# Pseudopolynomielle Algorithmen

---

Wir können eine Subset-Sum-Instanz  $w_1, \dots, w_n$  mit Zielwert  $t$  in Zeit  $O(nt)$  lösen.

**Frage:** Warum zeigt das nicht, dass  $P = NP$ ?  
SubsetSum ist doch NP-vollständig?

Generell gehen wir davon aus, dass Eingabezahlen in Binärdarstellung gegeben sind.  
Wenn also  $t$  als  $m$ -Bit-Zahl gegeben ist, dann kann  $t$  Werte bis  $2^m$  annehmen.  
→  $t$  kann exponentiell in der Eingabegröße sein

Wir sprechen hier von einem **pseudo-polynomiellen Algorithmus**  
Ein solcher Algorithmus läuft in Polynomialzeit, wenn die Eingabezahlen in unärer Darstellung gegeben wären

# PARTITION

---

## PARTITION.

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten**

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau der Summe der nicht ausgewählten Elemente entspricht?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$\sum_{i \in S} w_i = \sum_{i \notin S} w_i? \quad \text{also: } w(S) = w(\{1, \dots, N\} \setminus S)$$



# PARTITION

## PARTITION.

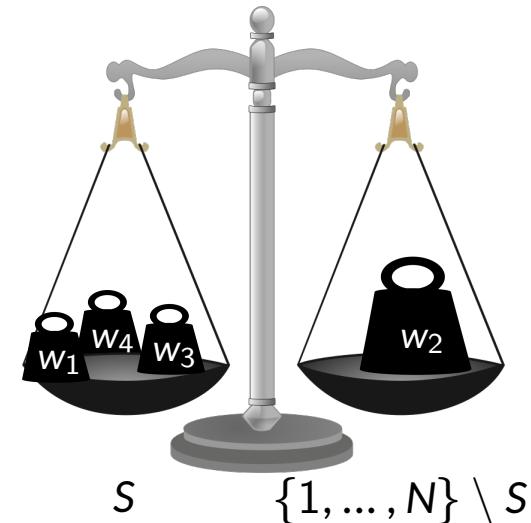
**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten**

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau der Summe der nicht ausgewählten Elemente entspricht?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$\sum_{i \in S} w_i = \sum_{i \notin S} w_i? \quad \text{also: } w(S) = w(\{1, \dots, N\} \setminus S)$$

**Frage:** Gilt  $\text{PARTITION} \leq_P \text{SUBSETSUM}$ ?



# PARTITION

## PARTITION.

**Gegeben:** Eine Sequenz  $w_1, \dots, w_N \in \mathbb{N}$  von **Gewichtswerten**

**Gesucht:** Gibt es eine Auswahl von **Gewichtswerten**, deren Summe genau der Summe der nicht ausgewählten Elemente entspricht?

Das heißt, gibt es eine Menge  $S \subseteq \{1, \dots, N\}$  sodass

$$\sum_{i \in S} w_i = \sum_{i \notin S} w_i? \quad \text{also: } w(S) = w(\{1, \dots, N\} \setminus S)$$

Frage: Gilt  $\text{PARTITION} \leq_P \text{SUBSETSUM}$ ?

## Theorem.

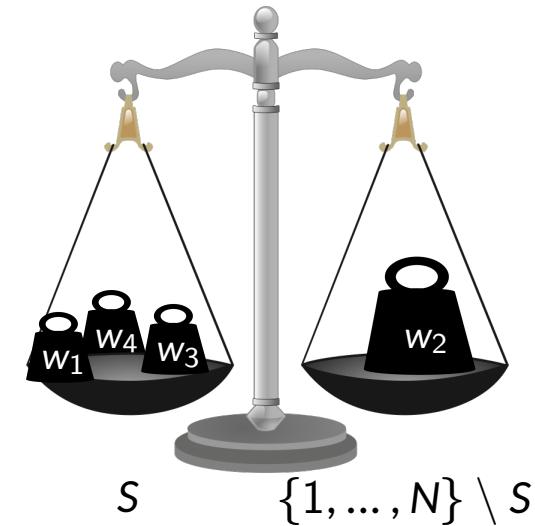
PARTITION ist **NP-vollständig**.

**Beweis:** 1. PARTITION ist in NP.

Beweis analog zu Subset Sum und hier ausgelassen

2. Wir werden zeigen, dass  $\text{SUBSETSUM} \leq_P \text{PARTITION}$

3.-5. nächste Folie



# SUBSETSUM $\leq_P$ PARTITION

---

Gegeben Gewichtswerte  $w_1, \dots, w_N$  und Zielwert  $t$  für Subset Sum,  
konstruieren wir eine PARTITION-Instanz wie folgt:

Es sei  $W := w_1 + w_2 + \dots + w_N$

Wir definieren die zusätzlichen Gewichtswerte  $w_{N+1} = W + t$  und  $w_{N+2} = 2W - t$

**Behauptung:**

$w_1, \dots, w_N$  und  $t$  ist JA-Instanz für Subset Sum  $\Leftrightarrow w_1, \dots, w_{N+2}$  ist JA-Instanz für PARTITION

**Beweis:** " $\Rightarrow$ " Es sei  $S \subseteq \{1, \dots, N\}$  sodass  $\sum_{i \in S} w_i = t$ .

Dann bildet  $S \cup \{N+2\}$  eine gültige Auswahl für PARTITION, denn:

$$\begin{aligned} (\sum_{i \in S} w_i) + w_{N+2} &= t + 2W - t = 2W \\ (\sum_{i \in \{1, \dots, N\} \setminus S} w_i) + w_{N+1} &= (W - t) + W + t = 2W \end{aligned}$$

" $\Leftarrow$ " Es sei  $S \subseteq \{1, \dots, N+2\}$ , sodass  $\sum_{i \in S} w_i = \sum_{i \notin S} w_i = 2W$

Es gilt entweder  $N+1 \notin S$  und  $N+2 \in S$  oder umgekehrt, da  $w_{N+1} + w_{N+2} = 3W$ .

Betrachte den Fall, dass  $N+2 \in S$  (ansonsten betrachte  $\{1, \dots, N+2\} \setminus S$  als  $S$ ):

$$\text{Es gilt: } (\sum_{i \in S \setminus \{N+2\}} w_i) + w_{N+2} = 2W$$

$$\Rightarrow \sum_{i \in S \setminus \{N+2\}} w_i = 2W - w_{N+2} = t$$

$\Rightarrow S \setminus \{N+2\}$  bildet eine gültige Auswahl für die Subset-Sum-Instanz. □

Die Reduktion kann offensichtlich in Polynomialzeit berechnet werden.

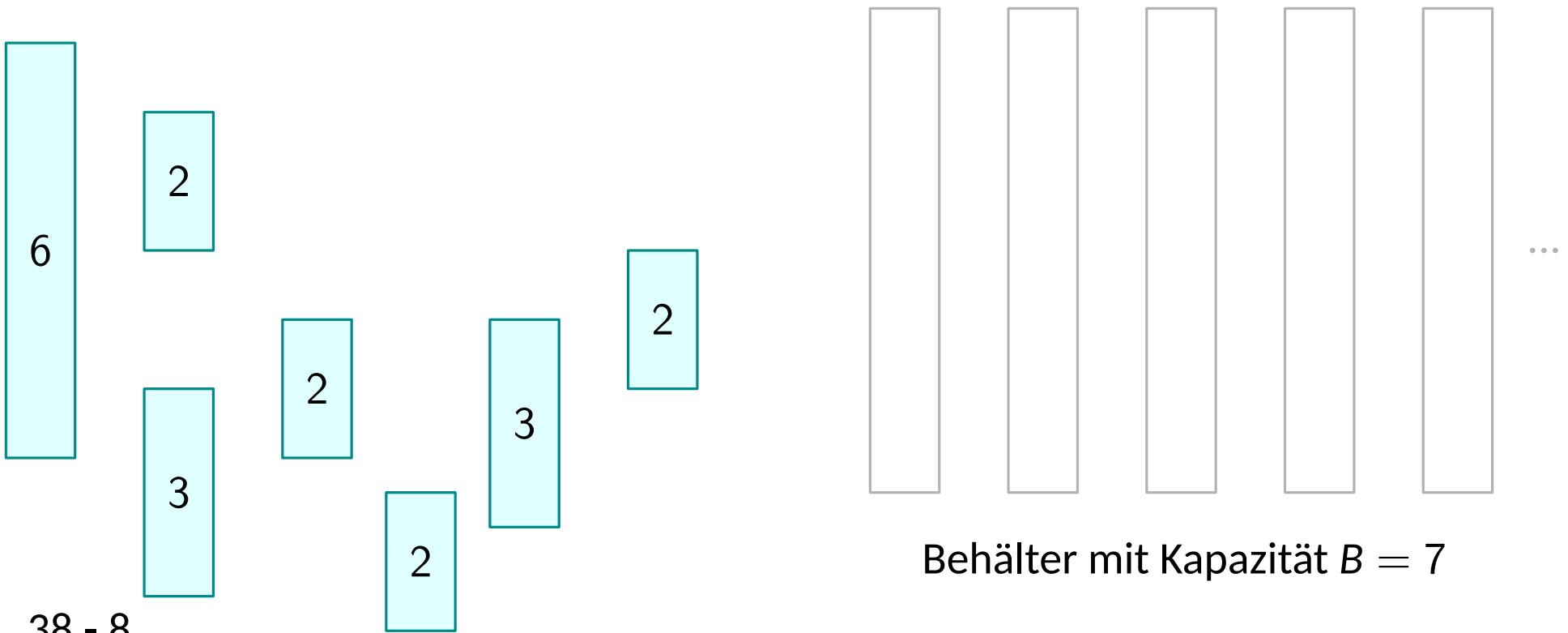
# Bin Packing

## Bin Packing (BPP)

**Gegeben:** Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

**Gesucht:** Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



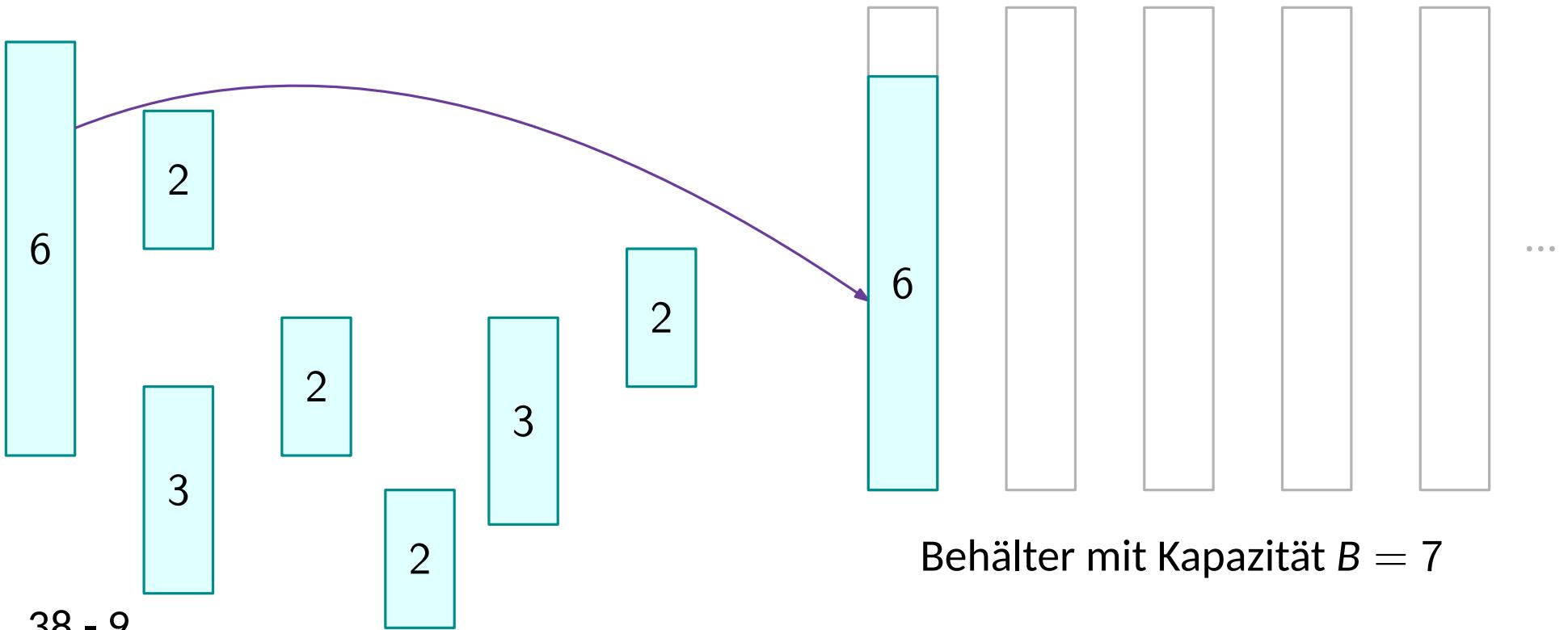
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



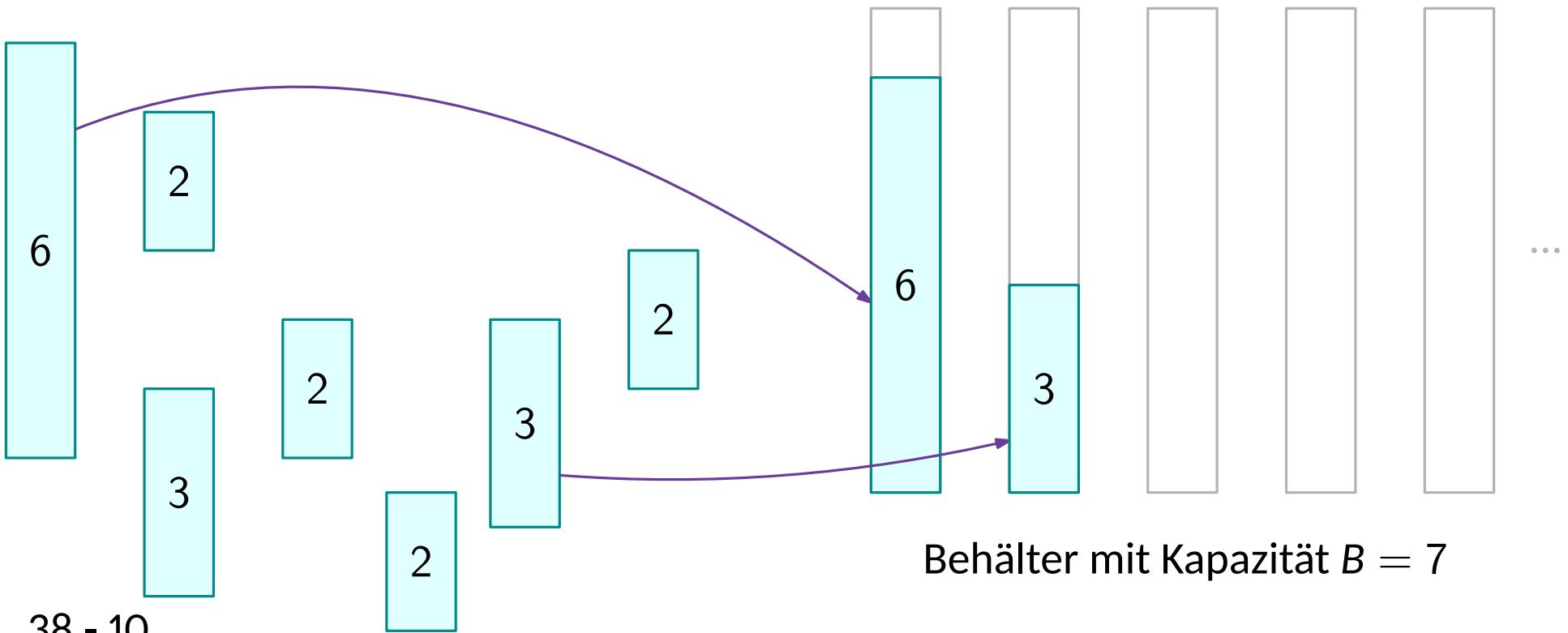
# Bin Packing

## Bin Packing (BPP)

**Gegeben:** Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

**Gesucht:** Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



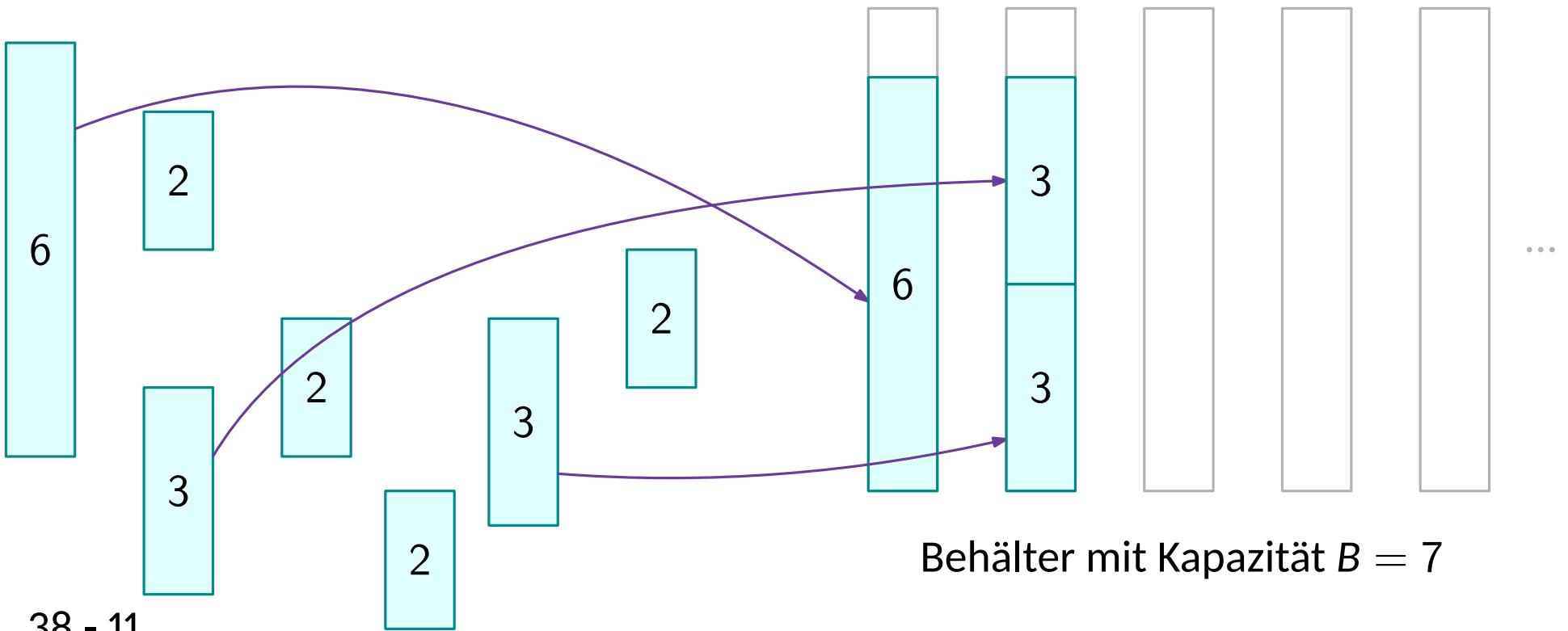
# Bin Packing

## Bin Packing (BPP)

**Gegeben:** Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

**Gesucht:** Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



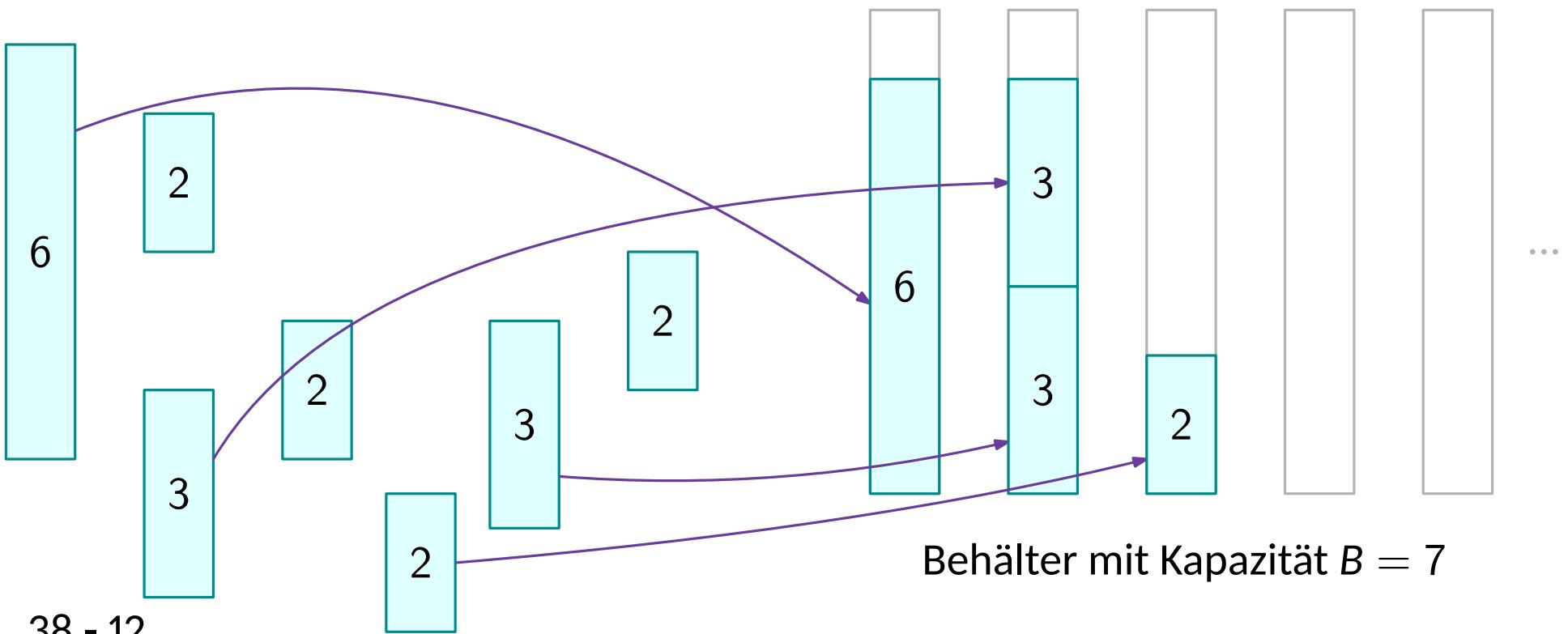
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



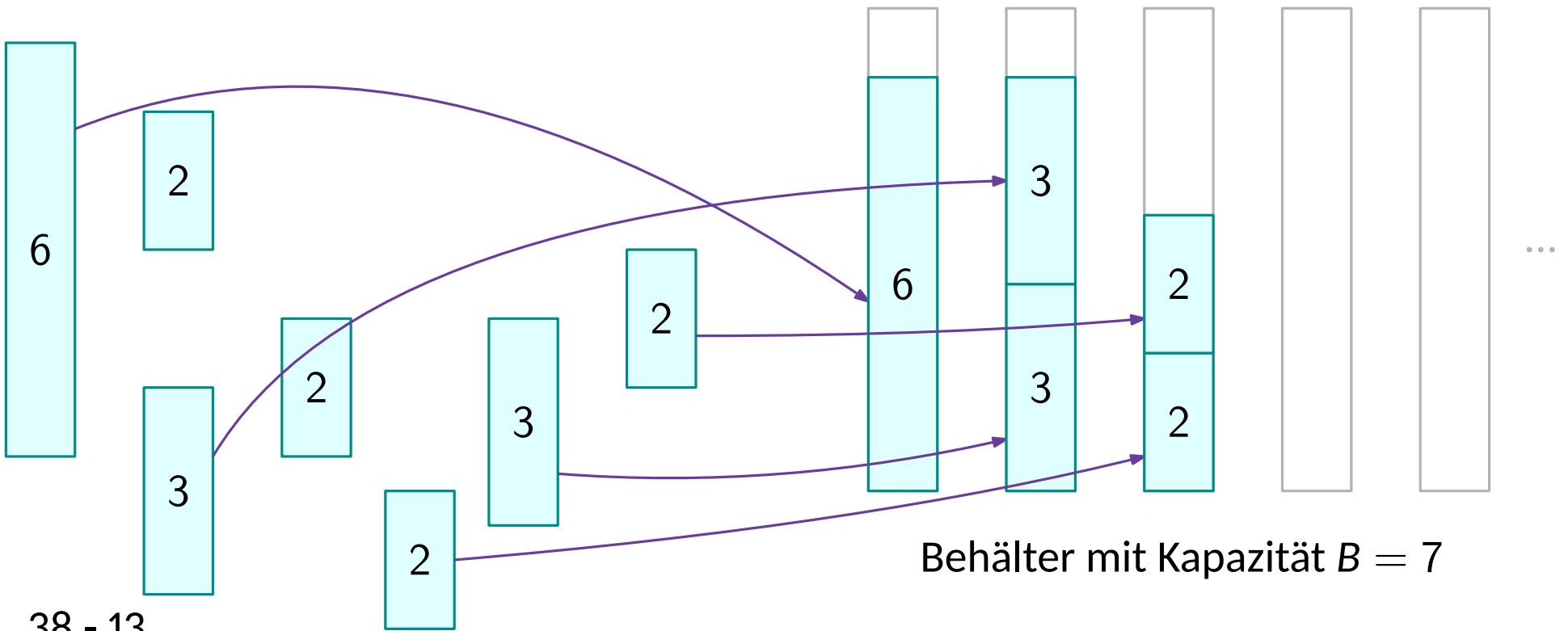
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



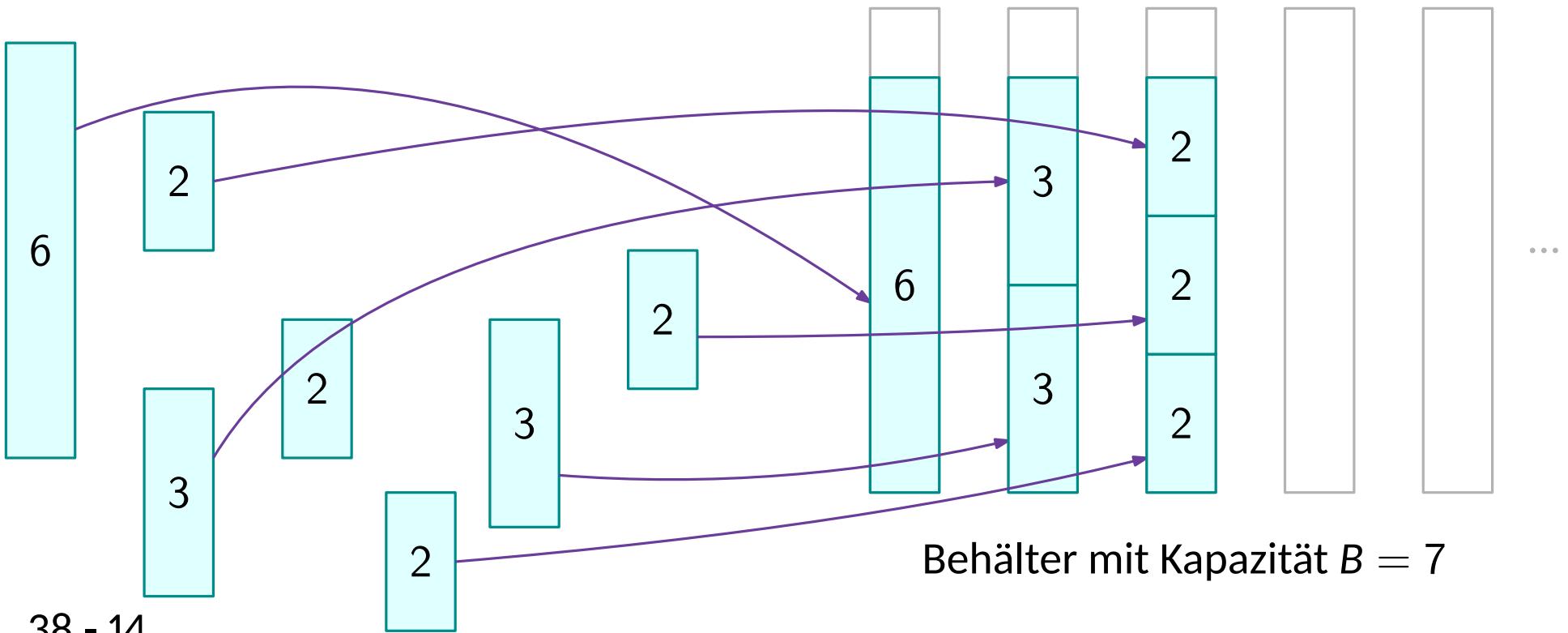
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



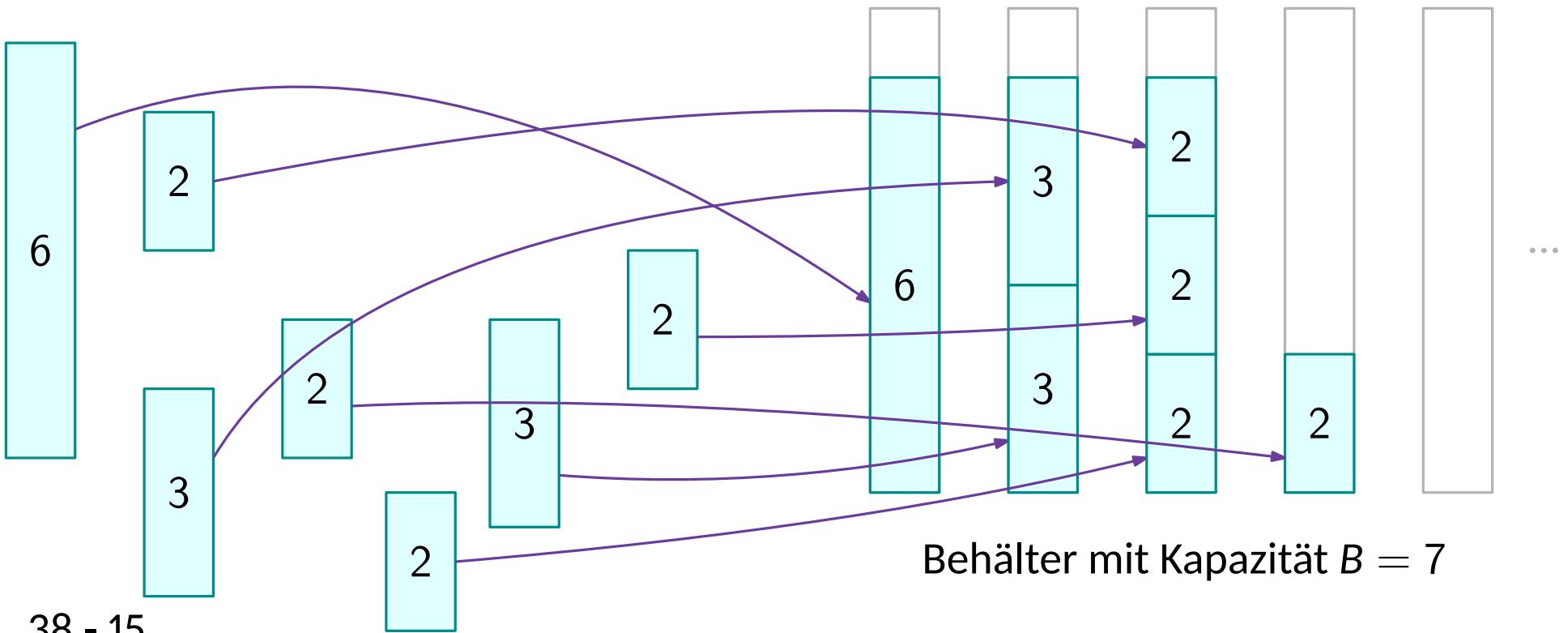
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



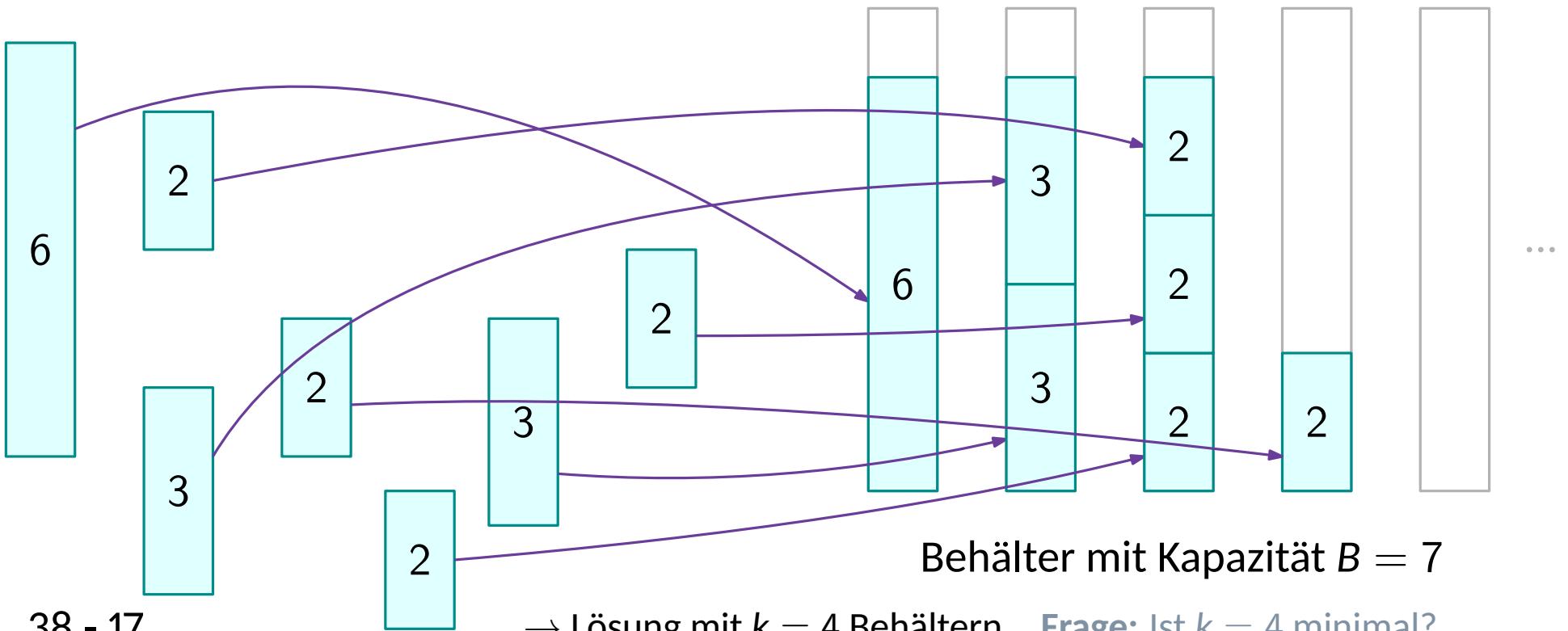
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



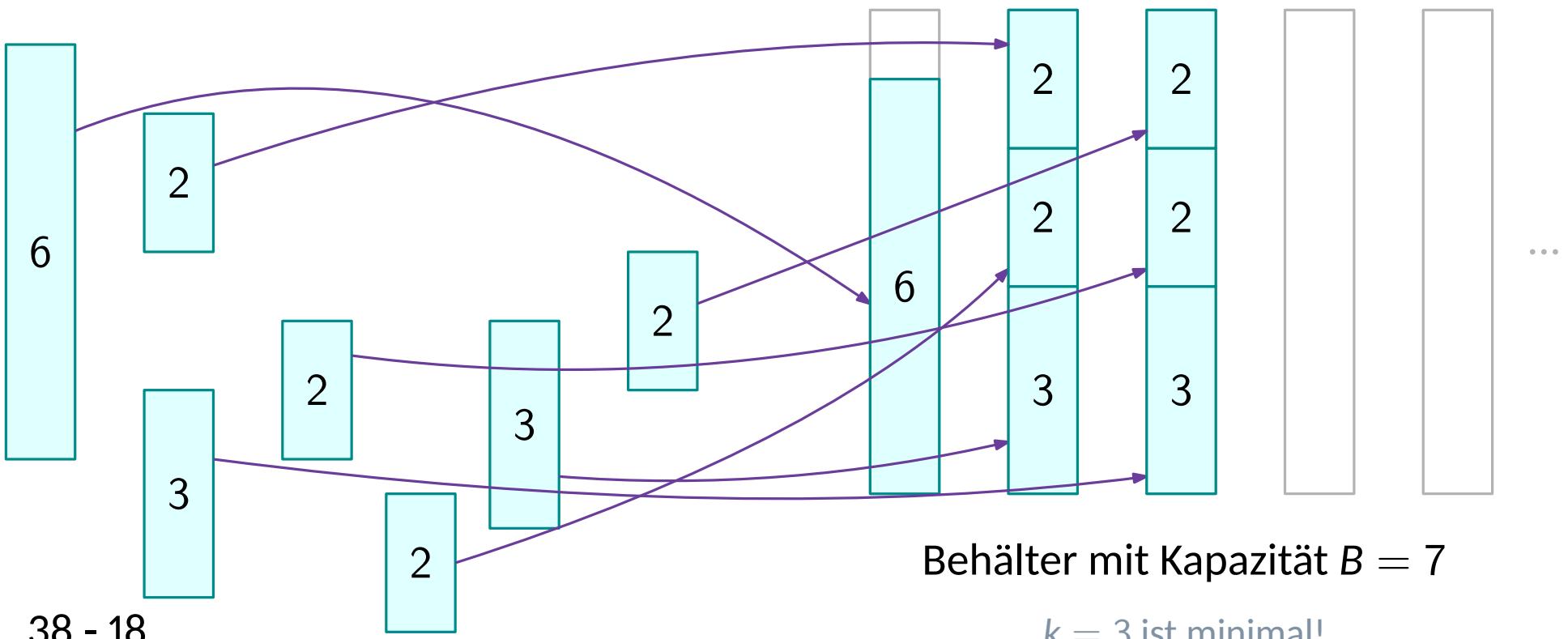
# Bin Packing

## Bin Packing (BPP)

Gegeben: Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten

Gesucht: Finde eine Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter, sodass gilt:

- jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$
- die Anzahl Behälter  $k$  ist minimal



# Bin Packing - Entscheidungsvariante

## Bin Packing - Entscheidungsvariante (BPP-E)

**Gegeben:** · Gewichtskapazität  $B \in \mathbb{N}$  und eine Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten  
·  $k \in \mathbb{N}$

**Gesucht:** Gibt es eine valide Zuweisung  $f : \{1, \dots, N\} \rightarrow \{1, \dots, k\}$  der Gewichte auf  $k$  Behälter?

D.h.:

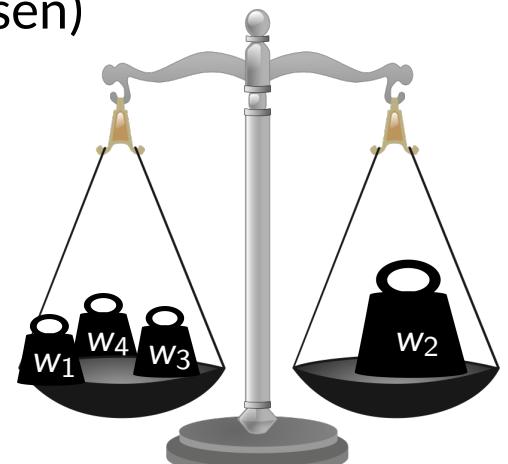
· jeder Behälter trägt Gesamtgewicht  $\leq B$ : für jedes  $j \in \{1, \dots, k\}$  ist  $\sum_{i:f(i)=j} w_i \leq B$

## Theorem.

BP-E ist NP-vollständig.

1. BP-E ist in NP. Warum? (Beweis ausgelassen)
2. Wir reduzieren von PARTITION
3. – 5. Frage: Wie?

Hinweis: das ist einfacher als man vielleicht denkt...



# Rucksackproblem

---

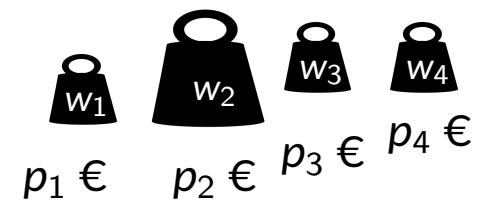
## Rucksackproblem (KNAPSACK)

**Gegeben:** · **Gewichtskapazität**  $B \in \mathbb{N}$

- Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von **Gewichtswerten**
- Sequenz  $p_1, \dots, p_N \in \mathbb{N}$  von zugehörigen **Profiten**

**Gesucht:** Finde eine Auswahl  $S \subseteq \{1, \dots, N\}$ , sodass gilt:

- das Gesamtgewicht ist höchstens  $B$ :  $\sum_{i \in S} w_i \leq B$
- der Profit  $\sum_{i \in S} p_i$  ist maximal.



# Rucksackproblem

## Rucksackproblem - Entscheidungsvariante (KNAPSACK-E)

**Gegeben:** · Gewichtskapazität  $B \in \mathbb{N}$ , Profitwert  $P \in \mathbb{N}$

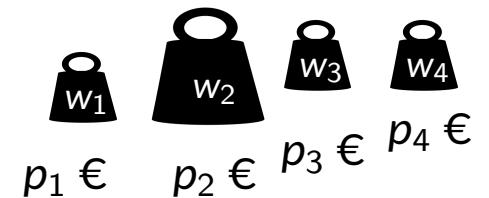
- Sequenz  $w_1, \dots, w_N \in \{1, \dots, B\}$  von Gewichtswerten
- Sequenz  $p_1, \dots, p_N \in \mathbb{N}$  von zugehörigen Profiten

**Gesucht:** Finde eine Auswahl  $S \subseteq \{1, \dots, N\}$ , sodass gilt:

- das Gesamtgewicht ist höchstens  $B$ :  $\sum_{i \in S} w_i \leq B$
- der Profit ist mindestens  $P$ :  $\sum_{i \in S} p_i \geq P$

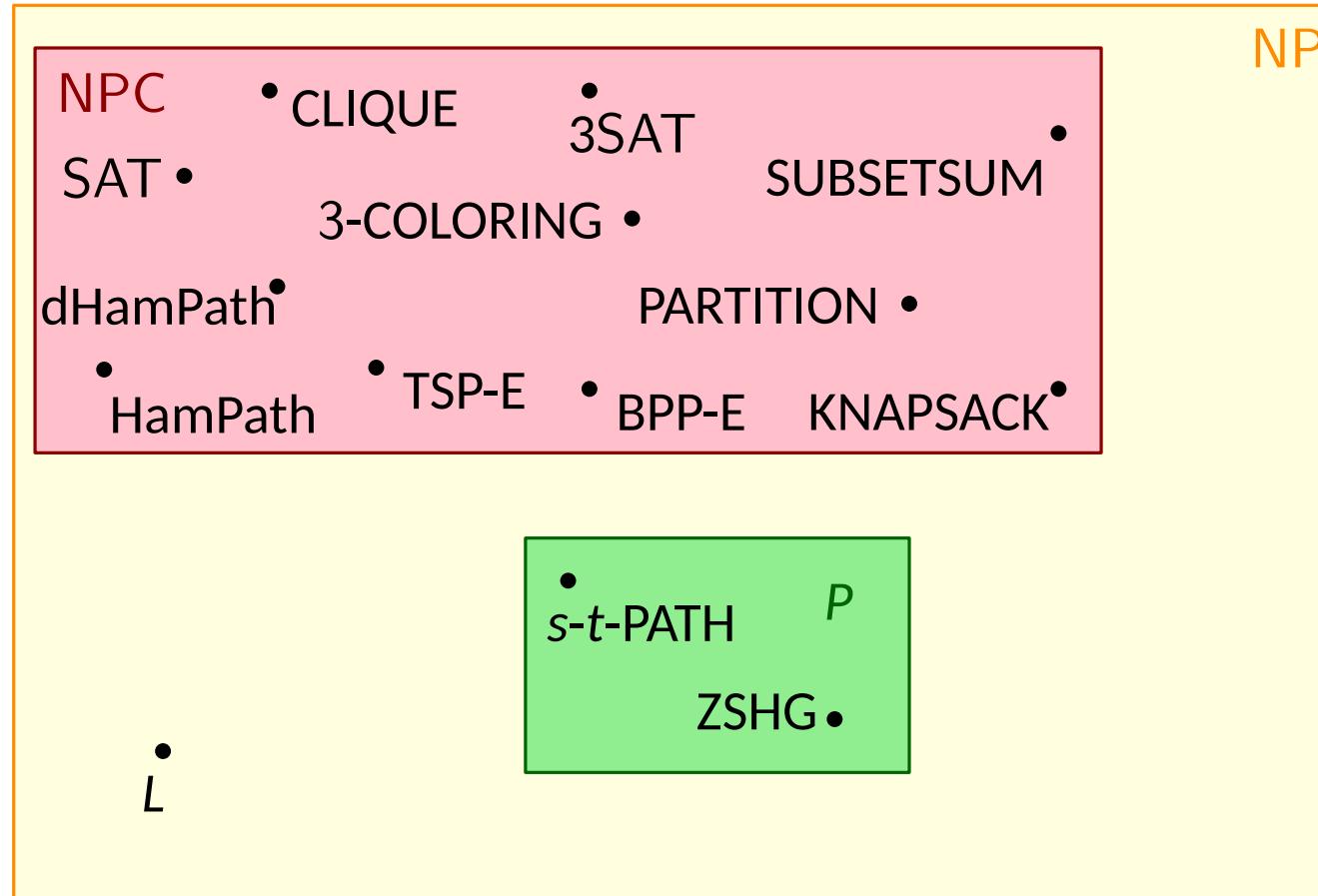
**Theorem.**

KNAPSACK-E ist NP-vollständig.



# Zusammenfassung

Für dieses Bild nehmen wir an, dass  $P \neq NP$ :



**Ausblick:** Gibt es Probleme, die weder in P noch in NPC liegen (angenommen  $P \neq NP$ )?

**Satz von Ladner:** Wenn  $P \neq NP$ , dann gibt es  $L \in NP$ , sodass  $L \notin P$  und  $L \notin NPC$   
→ solche Probleme heißen **NP-intermediate**

Algorithmen und Datenstrukturen SS'23

# Kapitel 17: Weiteres zur Komplexitätstheorie

Marvin Künemann

AG Algorithmen & Komplexität

# VLU-Feedback: Kleine Umfrage

---

**Vielen Dank** für die hilfreichen und detaillierten Kommentare!

Aufgrund einzelner Kommentare eine kleine Umfrage:

1. Stimmt die Mischung zwischen "Tafel"präsentation und Folien?

- a) lieber mehr Tafelpräsentationen
- b) derzeitige Mischung stimmt
- c) lieber mehr Folienpräsentation

2. Stimmt die Informationsdichte der Folien?

- a) lieber mehr Inhalt pro Folie
- b) derzeitige Mischung stimmt
- c) lieber weniger Inhalt pro Folie

# Kapitelüberblick

---

Letzte Kapitel: Einige NP-Vollständige Probleme

Dieses Kapitel: Weitere Bemerkungen zur Komplexitätstheorie

Insbesondere besprechen wir in diesem Kapitel:

- Turingmaschinen
- Nichtdeterministische Turingmaschinen

# **Ein weiteres Berechnungsmodell: Die Turingmaschine**

# Motivation

---

Die genaue Wahl des Berechnungsmodells ist nicht entscheidend

→ es gibt verschiedene Möglichkeiten, die in einem gewissen Sinne **äquivalent** zueinander sind.

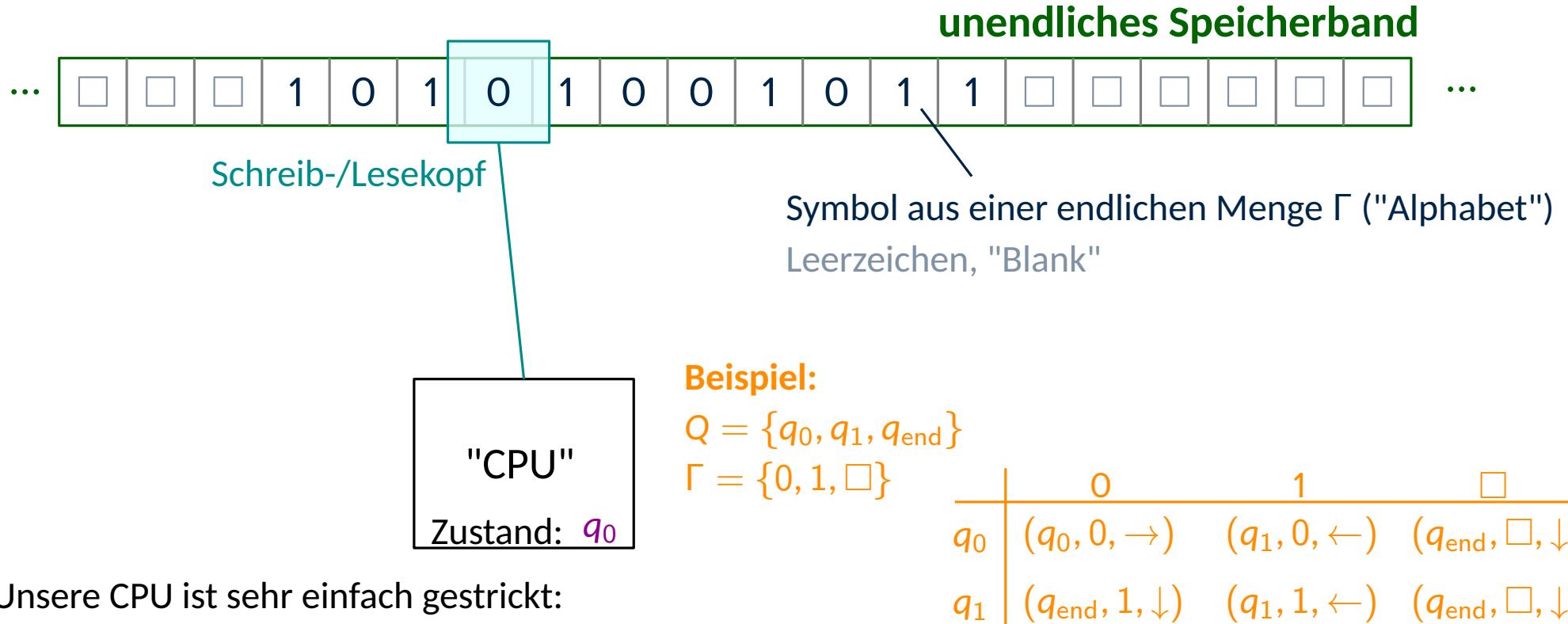
Wir führen jetzt die **Turingmaschine** ein

→ diese ist besonders geeignet für die Komplexitätstheorie

**Ziel:** Berechnungsmodell, das

- einfach zu beschreiben ist, aber
- trotzdem so mächtig wie unsere RAM

# Turingmaschine: Intuition



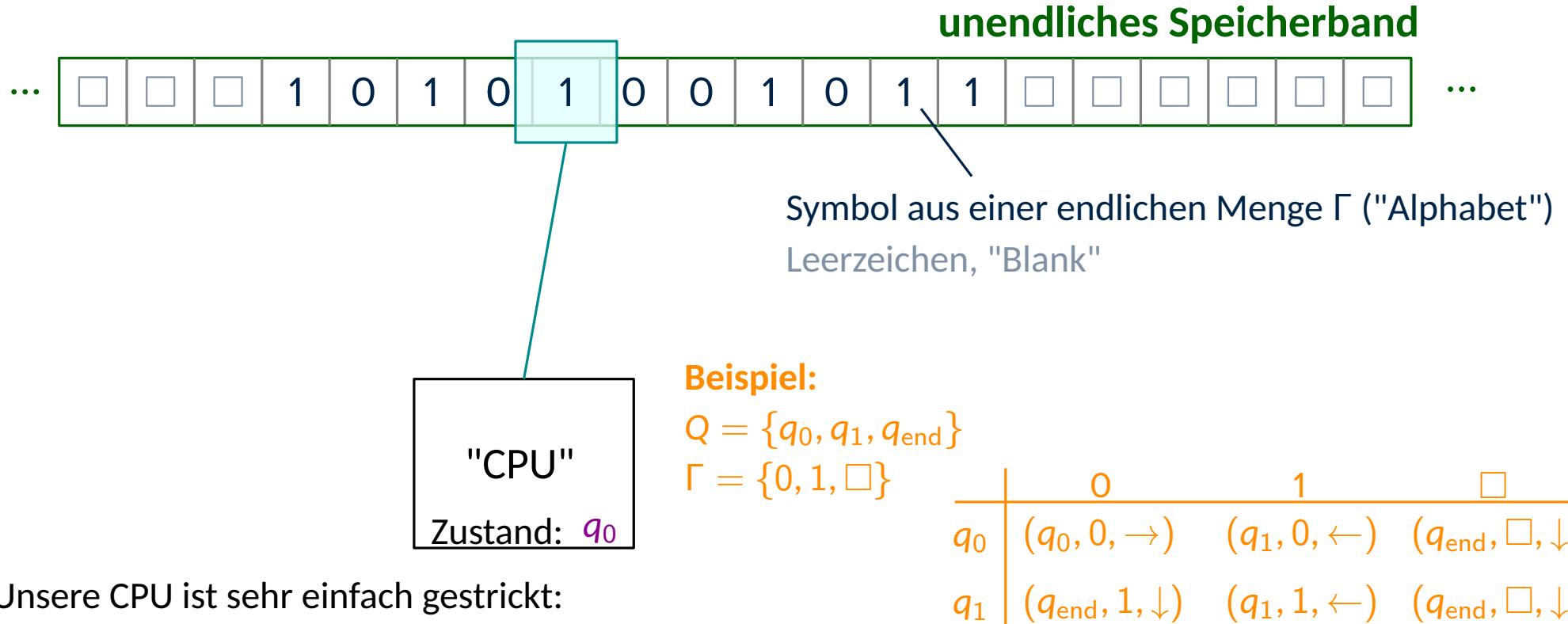
Unsere CPU ist sehr einfach gestrickt:

- CPU hat einen **Zustand** aus einer endlichen **Zustandsmenge**  $Q$
- erlaubte Instruktionen sind folgender Art:

Wenn CPU in Zustand  $q$  ist und das Schreib-/Lesekopf Symbol  $\sigma$  liest, dann

- gehe in den Zustand  $q'$  über
- schreibe das neue Symbol  $\sigma'$  an die Position des Schreib-/Lesekopfes
- bewege den Schreib-/Lesekopf *nach links/nach rechts/nicht*

# Turingmaschine: Intuition



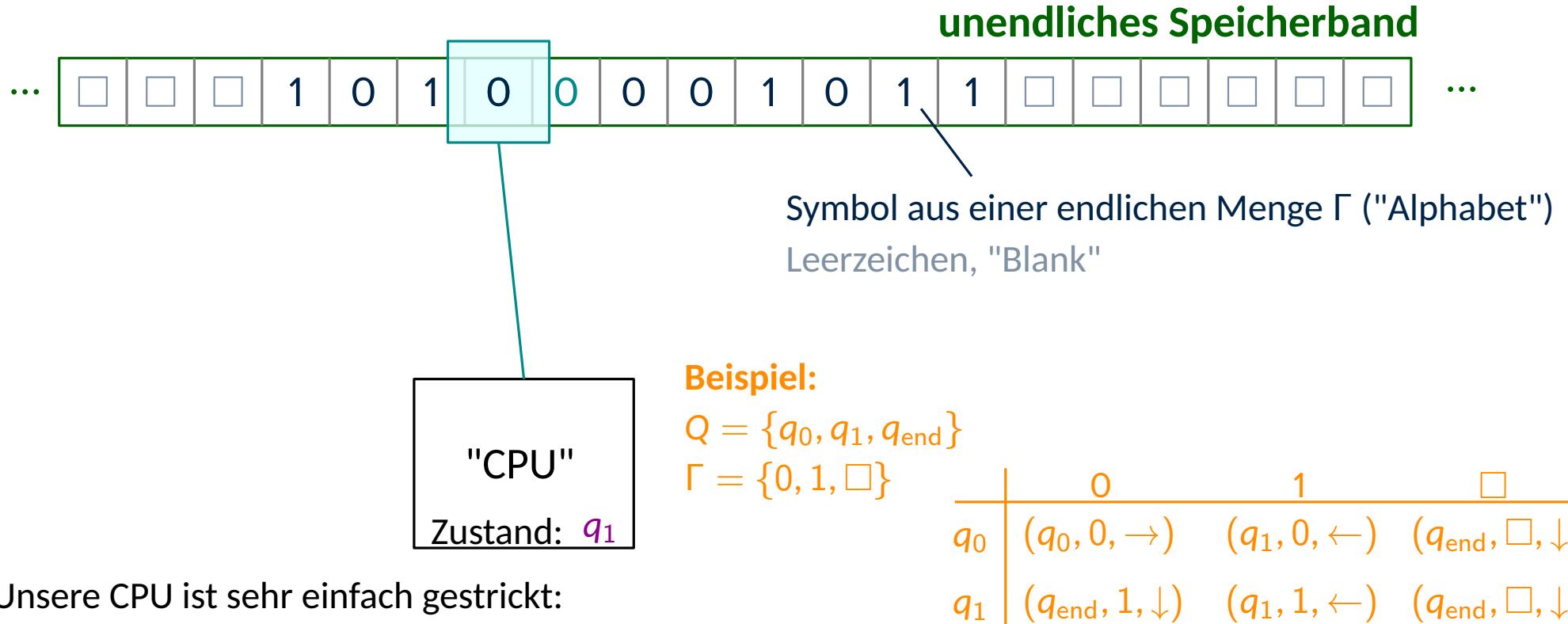
Unsere CPU ist sehr einfach gestrickt:

- CPU hat einen **Zustand** aus einer endlichen **Zustandsmenge**  $Q$
- erlaubte Instruktionen sind folgender Art:

Wenn CPU in Zustand  $q$  ist und das Schreib-/Lesekopf Symbol  $\sigma$  liest, dann

- gehe in den Zustand  $q'$  über
- schreibe das neue Symbol  $\sigma'$  an die Position des Schreib-/Lesekopfes
- bewege den Schreib-/Lesekopf *nach links/nach rechts/nicht*

# Turingmaschine: Intuition



Unsere CPU ist sehr einfach gestrickt:

- CPU hat einen **Zustand** aus einer endlichen **Zustandsmenge**  $Q$
- erlaubte Instruktionen sind folgender Art:

Wenn CPU in Zustand  $q$  ist und das Schreib-/Lesekopf Symbol  $\sigma$  liest, dann

- gehe in den Zustand  $q'$  über
- schreibe das neue Symbol  $\sigma'$  an die Position des Schreib-/Lesekopfes
- bewege den Schreib-/Lesekopf *nach links/nach rechts/nicht*

# Formale Definition einer Turingmaschine

---

**Definition (Turingmaschine).**

Es seien

$Q$  eine endliche **Zustandsmenge**,

$\Sigma$  ein endliches **Eingabealphabet**,

$\Gamma$  ein endliches **Bandalphabet** mit  $\Sigma \subseteq \Gamma$ ,

$\square \in \Gamma \setminus \Sigma$  das Leerzeichensymbol ("Blank"),

$q_0 \in Q$  der **Startzustand**,

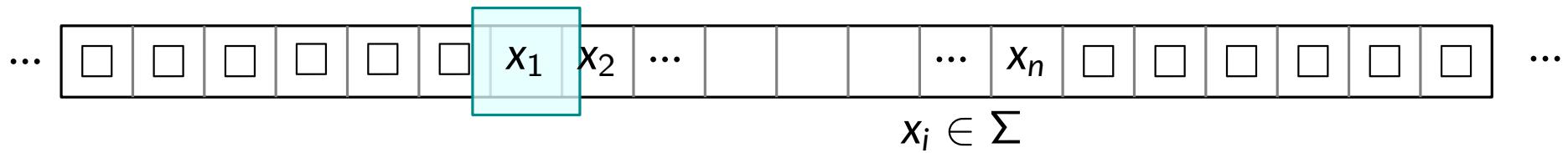
$q_{\text{end}} \in Q$  der **Endzustand**,

$\delta : (Q \setminus \{q_{\text{end}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}$  die **Überführungsfunktion**.

Dann ist  $T = (Q, \Sigma, \Gamma, \square, q_0, q_{\text{end}}, \delta)$  eine **Turingmaschine (TM)**.

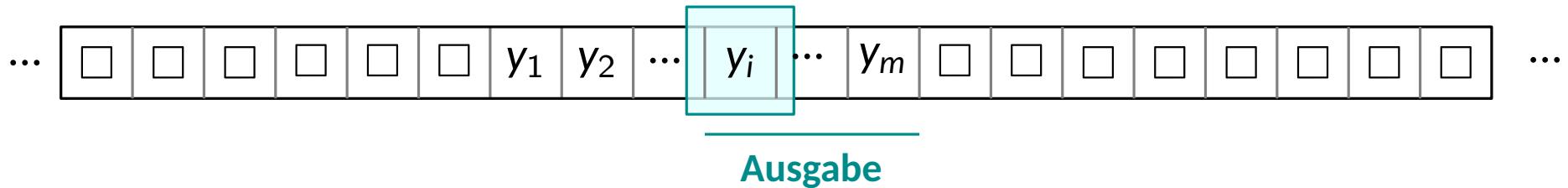
# Algorithmische Probleme auf der Turingmaschine

- Ausgangszustand:**
- Die **Eingabe** ist umrahmt von Blanks □
  - Der Schreib-/Lesekopf befindet sich an der ersten Position der Eingabe
  - Die Maschine befindet sich im Startzustand  $q_0$



Wir wenden die Überführungsfunktion solange an, bis wir den Endzustand  $q_{\text{end}}$  erreichen.

**Endzustand:** Wir definieren als **Ausgabe** die Positionen zwischen Schreib-/Lesekopf & erstem □



Die von der Turingmaschine berechnete Funktion ist definiert durch:

$$f(x_1, \dots, x_n) = \begin{cases} (y_i, \dots, y_m) & \text{wenn die Maschine hält} \\ \perp & \text{ansonsten} \end{cases}$$

**Laufzeit:** Anzahl ausgeführter Überführungen, bis wir den Endzustand erreicht haben.

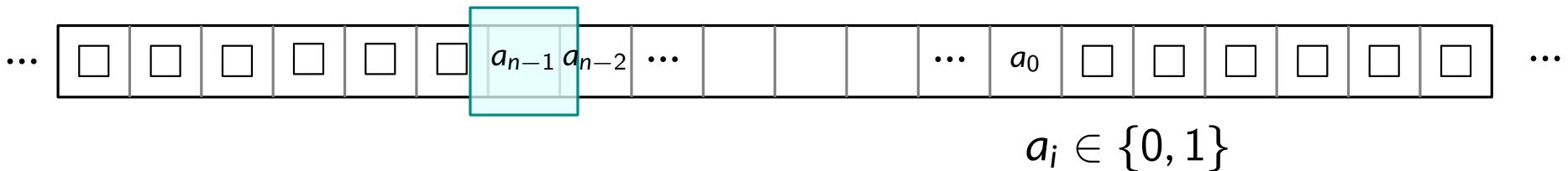
**Speicher:** Anzahl beschriebener Positionen, bis wir den Endzustand erreicht haben.

# Beispiel: Zahl inkrementieren

Problem (Zahl inkrementieren).

gegeben:  $n$ -stellige Binärzahl  $a = a_{n-1}, \dots, a_0$

gesucht:  $a + 1$  als Binärzahl



High-Level-Beschreibung einer Turingmaschine für dieses Problem:

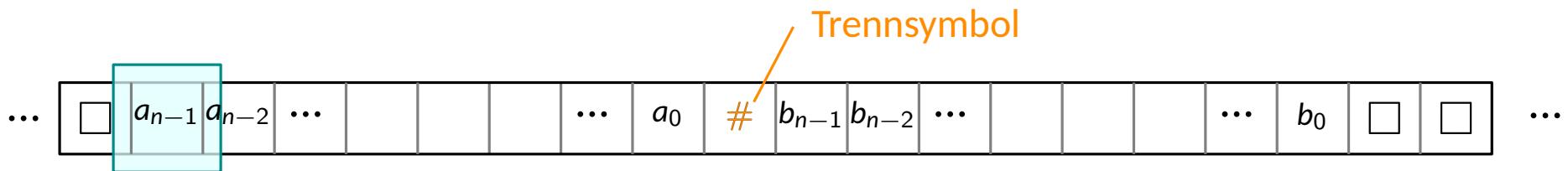
- Zustand  $q_0$ : · laufe nach rechts (ohne Zelleninhalte zu verändern), bis wir das erste  $\square$  erreichen  
→ beim ersten  $\square$ : Wechsel auf  $q_1$
- Zustand  $q_1$ : · laufe nach links, bis wir die erste 0 erreichen  
· setze dabei jede gelesene 1 auf 0  
· bei der ersten 0: überschreibe sie mit 1 und gehe in den Zustand  $q_2$
- Zustand  $q_2$ : · laufe nach links, bis wir das erste  $\square$  erreichen  
→ gehe dann nach rechts und in den Endzustand  $q_{\text{end}}$  über.

# Beispiel: Zahlen addieren

Problem (Zahlen addieren).

gegeben:  $n$ -stellige Binärzahlen  $a = a_{n-1}, \dots, a_0$  und  $b = b_{n-1}, \dots, b_0$

gesucht:  $z = a + b$  als Binärzahl



Eine (recht ineffiziente) Turingmaschine:

Solange  $b > 0$ :

- benutze vorigen Algorithmus (angepasst), um  $a$  zu inkrementieren
- benutze ähnlichen Algorithmus (angepasst), um  $b$  zu dekrementieren

Lösche die  $\#0 \dots 0$  Zeichen von  $b$  und bewege den Schreib-/Lesekopf an den Anfang

Effizienterer Ansatz:

- Implementiere die Schulmethode für die Addition  
(Benutze dabei einen festen Bereich von Zellen als "Arbeitsspeicher")

# **RAM vs. Turingmaschine**

## **Ist ein Modell mächtiger als das andere?**

# RAM-Programme auf der TM

## Behauptung

Es gibt eine Konstante  $c$  sodass folgende Aussage wahr ist:

Kleines Detail:  
Wir nehmen an, dass  $t(n) \geq n$ .

Sei  $R$  ein RAM-Programm, das eine Funktion  $f$  in  $t(n)$  Schritten berechnet.  
Dann gibt es eine Turingmaschine  $T$ , die  $f$  in Zeit  $t(n)^c$  berechnet.

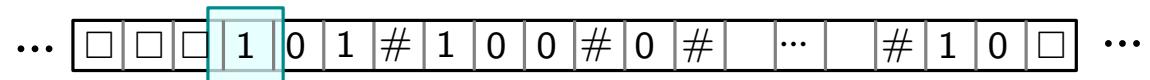
Wir werden diese Aussage nicht beweisen, sondern etwas Intuition geben:

Addieren von  $x_1$  und  $x_n$  auf der RAM:

⋮	
$x_n$	2
⋮	
$x_3$	0
$x_2$	4
$x_1$	5

- lade  $x_n$  in den Akkumulator
- ADD 1

Addieren von  $x_1$  und  $x_n$  auf der TM:



# RAM-Programme auf der TM

## Behauptung

Es gibt eine Konstante  $c$  sodass folgende Aussage wahr ist:

Kleines Detail:  
Wir nehmen an, dass  $t(n) \geq n$ .

Sei  $R$  ein RAM-Programm, das eine Funktion  $f$  in  $t(n)$  Schritten berechnet.  
Dann gibt es eine Turingmaschine  $T$ , die  $f$  in Zeit  $t(n)^c$  berechnet.

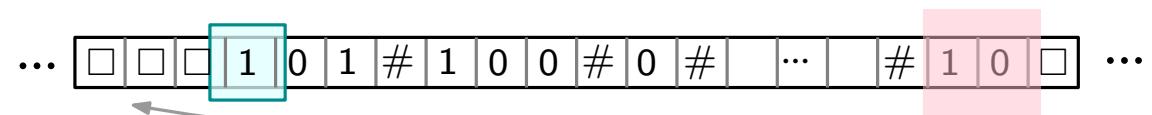
Wir werden diese Aussage nicht beweisen, sondern etwas Intuition geben:

Addieren von  $x_1$  und  $x_n$  auf der RAM:

⋮	
$x_n$	2
⋮	
$x_3$	0
$x_2$	4
$x_1$	5

- lade  $x_n$  in den Akkumulator
- ADD 1

Addieren von  $x_1$  und  $x_n$  auf der TM:



- Kopiere  $x_n$  an die Positionen links von  $x_1$

# RAM-Programme auf der TM

## Behauptung

Es gibt eine Konstante  $c$  sodass folgende Aussage wahr ist:

Kleines Detail:

Wir nehmen an, dass  $t(n) \geq n$ .

Sei  $R$  ein RAM-Programm, das eine Funktion  $f$  in  $t(n)$  Schritten berechnet.  
Dann gibt es eine Turingmaschine  $T$ , die  $f$  in Zeit  $t(n)^c$  berechnet.

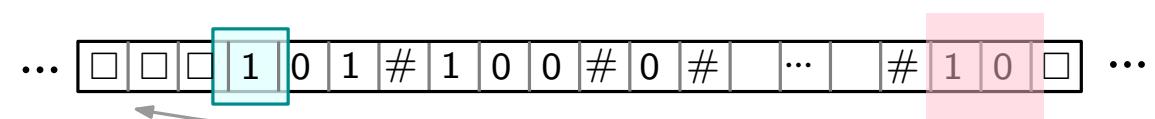
Wir werden diese Aussage nicht beweisen, sondern etwas Intuition geben:

Addieren von  $x_1$  und  $x_n$  auf der RAM:

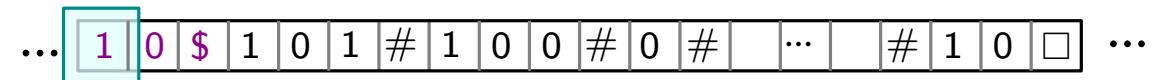
⋮	
$x_n$	2
⋮	
$x_3$	0
$x_2$	4
$x_1$	5

- lade  $x_n$  in den Akkumulator
- ADD 1

Addieren von  $x_1$  und  $x_n$  auf der TM:



- Kopiere  $x_n$  an die Positionen links von  $x_1$



# RAM-Programme auf der TM

## Behauptung

Es gibt eine Konstante  $c$  sodass folgende Aussage wahr ist:

Kleines Detail:

Wir nehmen an, dass  $t(n) \geq n$ .

Sei  $R$  ein RAM-Programm, das eine Funktion  $f$  in  $t(n)$  Schritten berechnet.  
Dann gibt es eine Turingmaschine  $T$ , die  $f$  in Zeit  $t(n)^c$  berechnet.

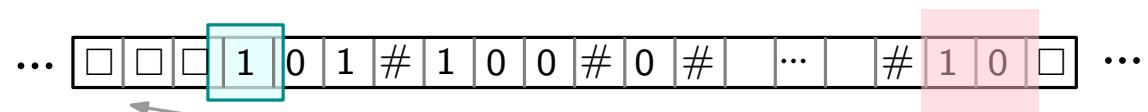
Wir werden diese Aussage nicht beweisen, sondern etwas Intuition geben:

Addieren von  $x_1$  und  $x_n$  auf der RAM:

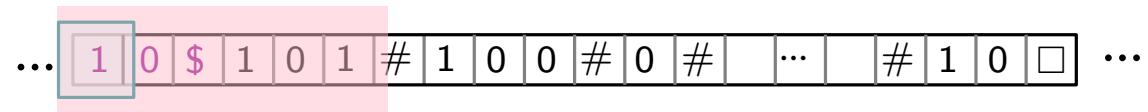
⋮	
$x_n$	2
⋮	
$x_3$	0
$x_2$	4
$x_1$	5

- lade  $x_n$  in den Akkumulator
- ADD 1

Addieren von  $x_1$  und  $x_n$  auf der TM:



- Kopiere  $x_n$  an die Positionen links von  $x_1$



- Addiere zwei Zahlen



# RAM-Programme auf der TM

## Behauptung

Es gibt eine Konstante  $c$  sodass folgende Aussage wahr ist:

Kleines Detail:

Wir nehmen an, dass  $t(n) \geq n$ .

Sei  $R$  ein RAM-Programm, das eine Funktion  $f$  in  $t(n)$  Schritten berechnet.  
Dann gibt es eine Turingmaschine  $T$ , die  $f$  in Zeit  $t(n)^c$  berechnet.

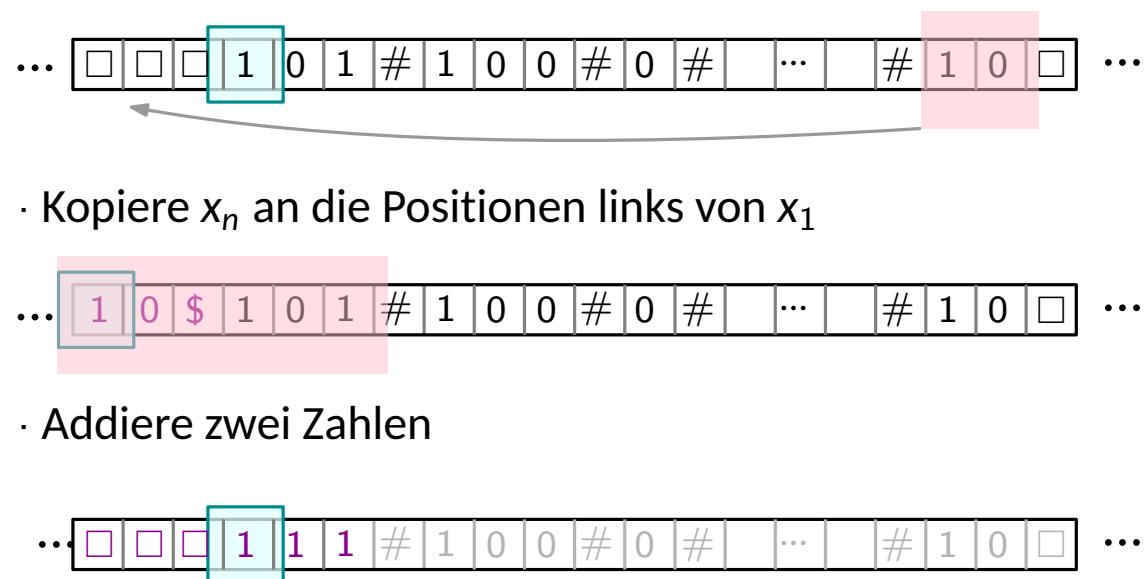
Wir werden diese Aussage nicht beweisen, sondern etwas Intuition geben:

Addieren von  $x_1$  und  $x_n$  auf der RAM:

⋮	
$x_n$	2
⋮	
$x_3$	0
$x_2$	4
$x_1$	5

- lade  $x_n$  in den Akkumulator
- ADD 1

Addieren von  $x_1$  und  $x_n$  auf der TM:



⇒ alles, was auf der RAM berechnet werden kann, kann ohne riesigen Zeitverlust auf einer Turingmaschine berechnet werden

# Turingmaschinen auf der RAM

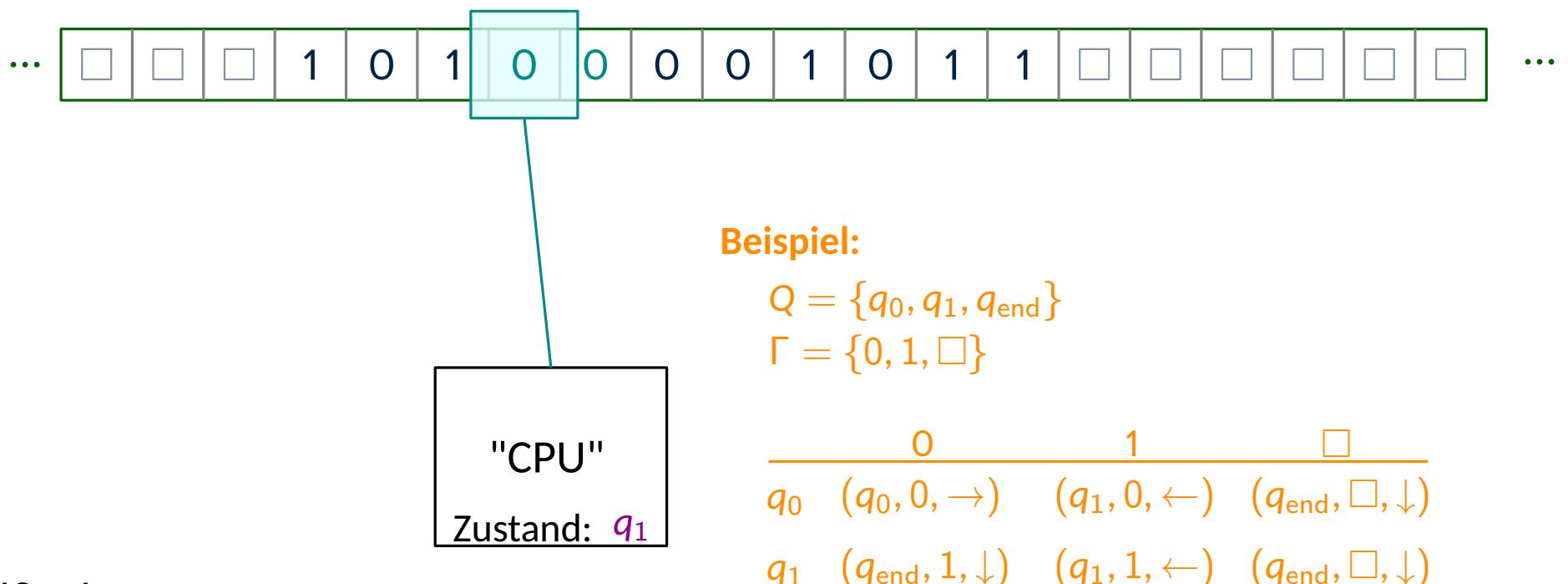
## Behauptung

Es gibt eine Konstante  $c$  sodass folgende Aussage wahr ist:

Kleines Detail:  
Wir nehmen an, dass  $t(n) \geq n$ .

Sei  $T$  eine Turingmaschine, die eine Funktion  $f$  in  $t(n)$  Schritten berechnet.  
Dann gibt es ein RAM-Programm  $R$ , das  $f$  in Zeit  $t(n)^c$  berechnet.

Frage: Wie kann man diese Aussage zeigen?



# Church-Turing-These

---

Wir sagen, dass eine Funktion  $f$  **TM-berechenbar** ist, wenn es eine Turingmaschine  $T$  gibt, die  $f$  berechnet.

Man geht davon aus, dass es kein "mächtigeres" Berechnungsmodell gibt. Dieser Gedanke ist ausgedrückt in der folgenden (nicht wohldefinierten) These:

## Church-Turing-These

Die Funktionen, die "intuitiv berechenbar" sind, sind genau die TM-berechenbaren Funktionen.

Allerdings:

Der Zeitbedarf könnte sich möglicherweise unterscheiden (zB. Quantencomputing vs Turingmaschinen)

# Turingmaschinen und Sprachen

---

Für Entscheidungsprobleme interpretieren wir die Ausgabe einer Turingmaschine  $T$  wie folgt:

- wenn  $T$  als letztes Zeichen eine 1 schreibt, so **akzeptiert**  $T$  die Eingabe
- ansonsten **verwirft**  $T$  die Eingabe

Die von  $T$  akzeptierte Sprache  $L_T$  ist gegeben durch:

$$x \in L_T \Leftrightarrow T \text{ akzeptiert die Eingabe } x$$

# **Nichtdeterministische Turingmaschinen**

## **- alternative Definition von NP**

# Nichtdeterministische Turingmaschine (NTM)

---

## Definition (Nichtdeterministische Turingmaschine).

Eine **nichtdeterministische Turingmaschine (NTM)** ist definiert wie eine Turingmaschine, mit dem Unterschied, dass die **Überführungsfunction**  $\delta$  zu einer **Überführungsrelation**  $\delta$  wird:

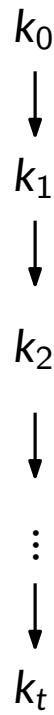
$$\delta \subseteq ((Q \setminus \{q_{\text{end}}\}) \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}).$$

D. h. die Berechnungsschritte einer NTM sind **nicht eindeutig** festgelegt.  
→ sie trifft **nichtdeterministische** Entscheidungen in jedem Berechnungsschritt

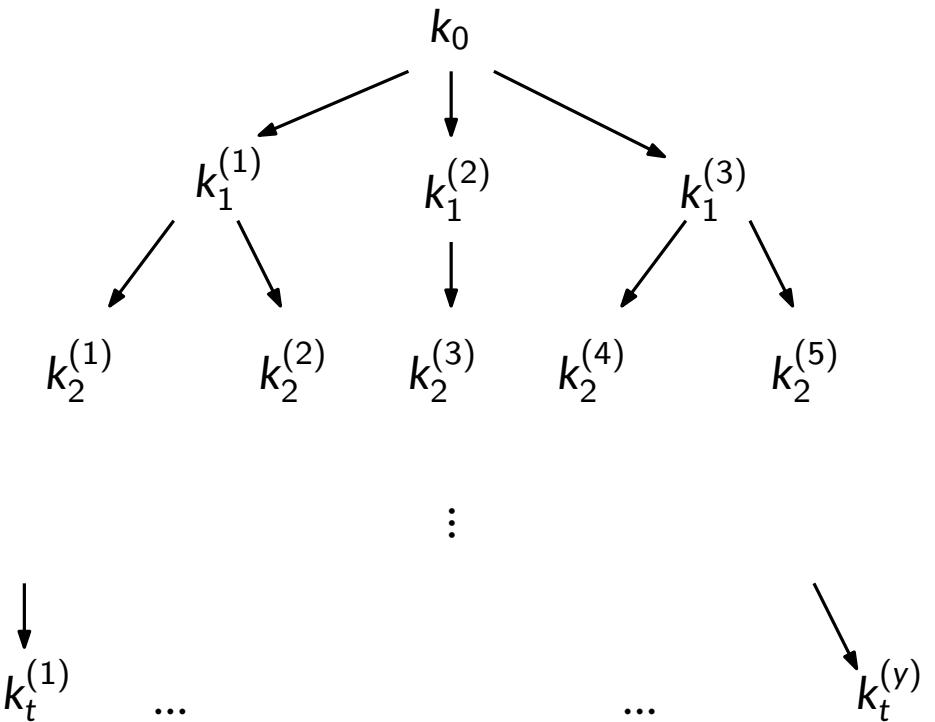
# Berechnungsbaum statt -pfad

Konfiguration  $k$ : aktueller Zustand  $q$ , Bandinhalt  $c$ , sowie Kopfposition  $h$

(Deterministische) Turingmaschine



Nichtdeterministische Turingmaschine



akzeptiert, wenn  $k_t$  akzeptierend ist

akzeptiert, wenn **mindestens eine** erreichbare Konfiguration akzeptierend ist

# NTM: erkannte Sprache und Laufzeit

---

Wir definieren die von einer NTM  $M$  erkannte Sprache  $L_M$  durch:

$x \in L_M \Leftrightarrow$  es existieren nichtdeterministische Entscheidungen von  $M$ , sodass  $M$  die Eingabe  $x$  akzeptiert

$M$  hat Laufzeit  $t(n)$ , wenn  $M$  auf allen Eingaben der Länge  $n$  und unter allen nichtdeterministischen Entscheidungen nach  $\leq t(n)$  Schritten hält.

# Alternative Definition von NP

---

NP ist die Menge aller Sprachen  $L$ , sodass:

- $L = L_M$  für eine NTM  $M$  mit polynomiell beschränkter Laufzeit  $t(n)$ .

Diese Definition ist äquivalent zu unserer bisherigen Definition:

$L$  wird von einer NTM erkannt  $\Leftrightarrow$   $L$  hat eine polynomiell beschränkte Verifiziererin

Algorithmen und Datenstrukturen SS'23

# Kapitel 18: Algorithmische Entwurfstechniken

Marvin Künemann

AG Algorithmen & Komplexität

# Ankündigungen I: Hörsaalübung, Musterlösungen

---

Die letzte Vorlesung (20. 7.) ist eine **Hörsaalübung**

**Thema:** Greedy-Algorithmen + Dynamische Programmierung

Es wird **keine Aufnahme und keinen Livestream** geben.

Es wird ein **unbepunktetes Übungsblatt** herausgegeben und direkt bearbeitet.

Einige Musterlösungen zu regulären Übungsblättern werden zeitnah bereitgestellt.

# Ankündigungen II: Zusatzangebote

---

## Freiwillige Zusatzangebote zur Klausurvorbereitung:

### 1. Vorbereitungsblatt mit früheren Klausuraufgaben

→ Feedback-Angebot:

Jede(r) Teilnehmer(in) **kann** die Bearbeitung **einer** Aufgabe einreichen, um Feedback zu erhalten

→ dazu werden im OLAT Gruppen gebildet

→ Gruppe der Größe  $g$  darf  $g$  bearbeitete Aufgaben einreichen

→ Abgabe voraussichtlich bis **Mitte/Ende August**

### 2. Wiederholung ausgewählter Themen

→ die Tutoren bieten Termine zu ausgewählten Themen an

→ kurze Präsentation zum Thema

→ danach Fragen & Antworten

→ Termine gegen **Ende August/Anfang September**

Elementare Datenstrukturen & amortisierte Analyse  
Entwurfstechniken & Sortieren  
Suchbäume & Wörterbücher  
Graphen  
Komplexitätstheorie

Diese Angebote sind **freiwillig** als Angebot zur Verständnisvertiefung

# Kapitelüberblick

---

Bisher: viele verschiedene Algorithmen und Datenstrukturen

Jetzt: allgemeine Herangehensweisen zum Algorithmenentwurf

Insbesondere besprechen wir in diesem Kapitel:

- erschöpfende Suche
- Greedy-Algorithmen
- Dynamische Programmierung (DP)

# Thematische Schwerpunkte

---

Eingabe →  → Ausgabe

## Grundbegriffe und Handwerkszeug

- Was ist ein Algorithmus und wie analysiere ich ihn? ✓
- Korrektheit und asymptotische Laufzeitschranken ✓
- formale Methoden und Beweistechniken ✓

## Algorithmen und Datenstrukturen

- Elementare Datenstrukturen ✓
- Suchen, Sortieren, Wörterbücher ✓
- Spezielle Strukturen: Listen, Graphen ✓

## Diese Woche

### Entwurfsmethoden

- Komplexe Algorithmen aus einfachen Algorithmen aufbauen
- Standardtechniken für bestimmte Problemklassen

### Theorie ✓ (vorgezogen)

- Komplexitätstheorie: Grenzen der effizienten Berechenbarkeit

# Ein kleiner Rückblick

---

Wir haben bereits viele verschiedene Algorithmen gesehen:

- Selection Sort
- Bubble Sort
- Heap Sort
- Quicksort
- Counting Sort
- DFS
- BFS
- Dijkstra
- viele algorithmische Aufgaben in den Übungen
- ...

Gibt es allgemeine Techniken zum Entwurf schneller Algorithmen?  
Herangehensweisen

# Ausgewählte algorithmische Techniken

---

Es gibt verschiedene allgemeine Entwurfsprinzipien:

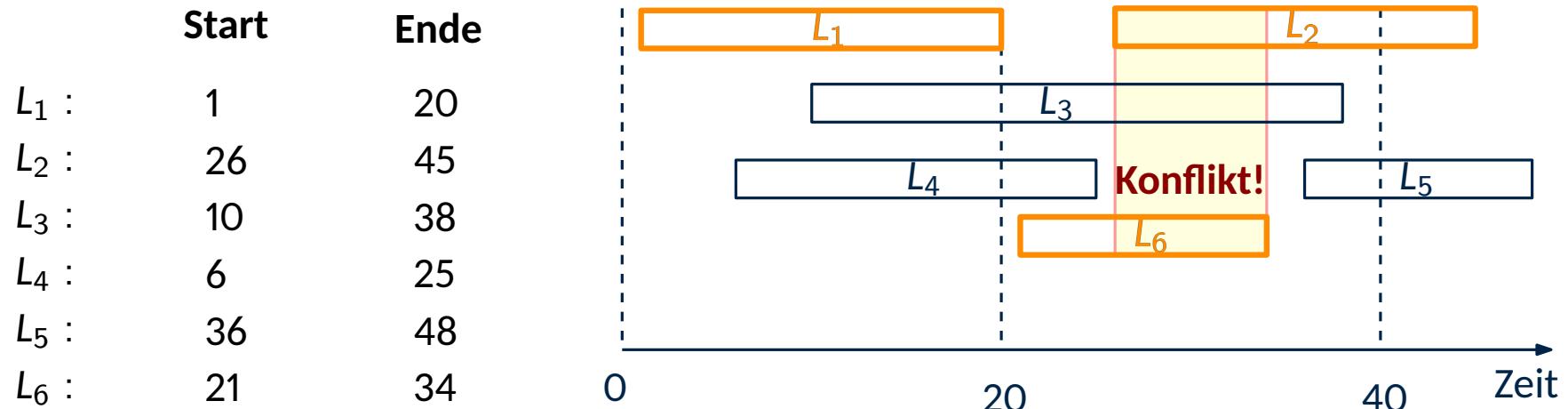
- Erschöpfende Suche (*Exhaustive Search*)
  - systematisches Erkunden aller möglichen Lösungen
- Teile-und-Herrsche (*Divide & Conquer*)
  - Aufteilen, rekursives Lösen & Zusammenfügen
- Greedy-Algorithmen
  - beste Entscheidung zum aktuellen Zeitpunkt
- Dynamische Programmierung
  - tabellarisches Bestimmen optimaler Lösungen für Teilprobleme
- randomisiertes Sampling
  - Zufallsmethoden zur Bestimmung guter Lösungen
- viele weitere: Lokale Suche, Lineare Programmierung, Gradient Descent, ...

Hinweis: Diese Techniken sind **nicht immer** sinnvoll anwendbar

# Ein Beispielproblem: Ambitioniertes Studieren

**Szenario:** Es gibt zu viele interessante Vorlesungen → wie kann ich größtmögliche Anzahl besuchen?

**Beispiel:** Die RPTU bietet Vorlesungen an:



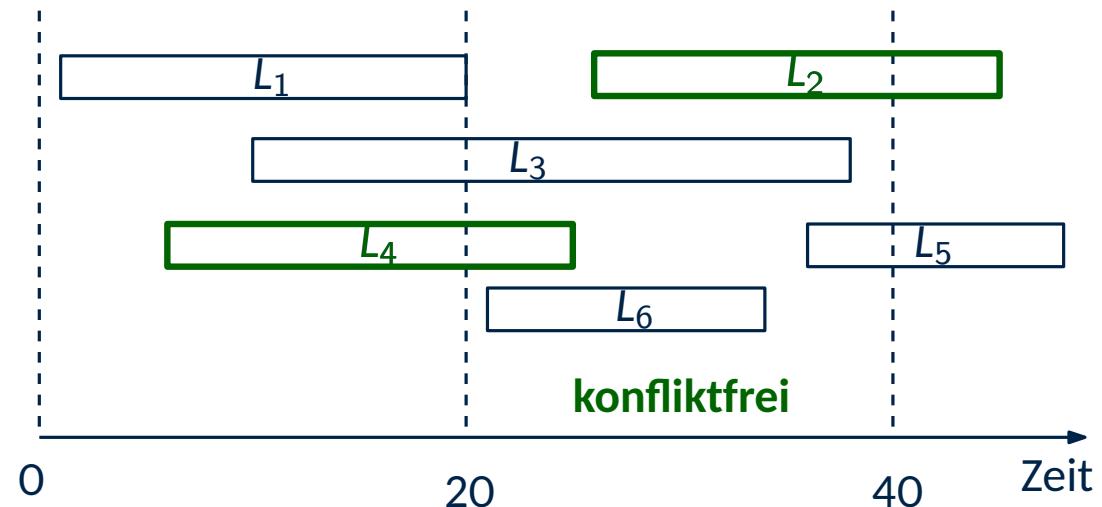
Ich kann keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")  $L_2$  und  $L_6$  überschneiden sich

# Ein Beispielproblem: Ambitioniertes Studieren

**Szenario:** Es gibt zu viele interessante Vorlesungen → wie kann ich größtmögliche Anzahl besuchen?

**Beispiel:** Die RPTU bietet Vorlesungen an:

	Start	Ende
$L_1$ :	1	20
$L_2$ :	26	45
$L_3$ :	10	38
$L_4$ :	6	25
$L_5$ :	36	48
$L_6$ :	21	34

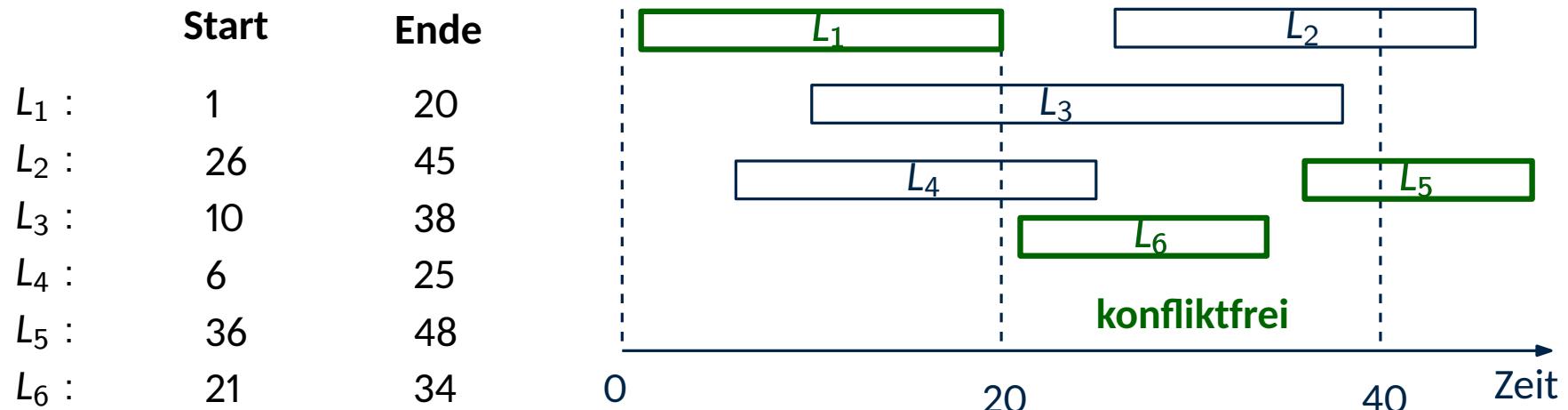


Ich kann keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")

# Ein Beispielproblem: Ambitioniertes Studieren

**Szenario:** Es gibt zu viele interessante Vorlesungen → wie kann ich größtmögliche Anzahl besuchen?

**Beispiel:** Die RPTU bietet Vorlesungen an:

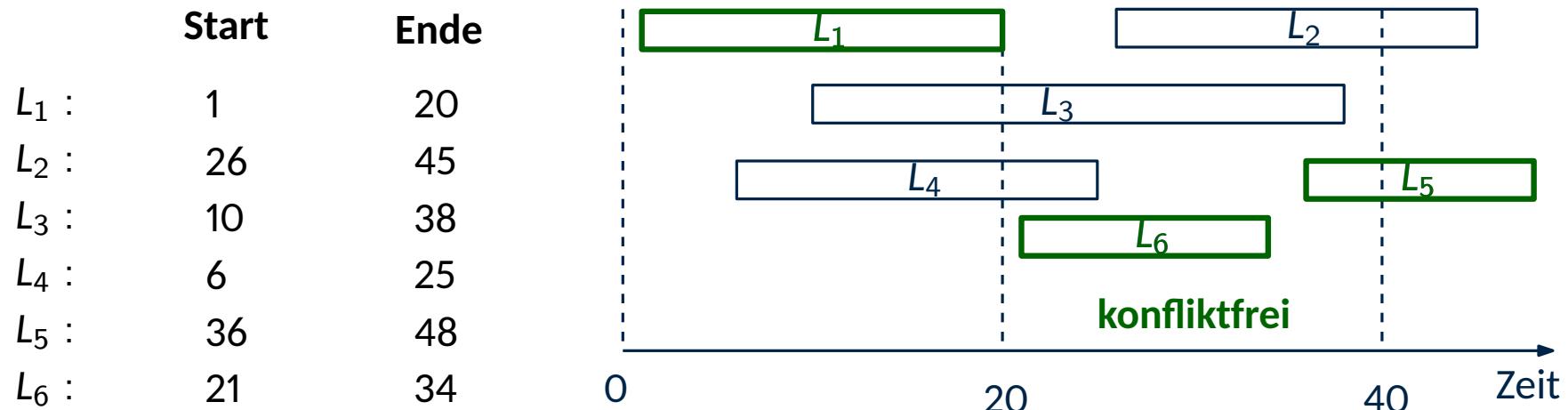


Ich kann keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")  
→ größtmögliche Auswahl konfliktfreier Vorlesungen: 3 (via  $L_1, L_6, L_5$ )

# Ein Beispielproblem: Ambitioniertes Studieren

**Szenario:** Es gibt zu viele interessante Vorlesungen → wie kann ich größtmögliche Anzahl besuchen?

**Beispiel:** Die RPTU bietet Vorlesungen an:



Ich kann keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")  
→ größtmögliche Auswahl konfliktfreier Vorlesungen: 3 (via  $L_1, L_6, L_5$ )

## Problem des ambitionierten Studierens (PAS)

**Gegeben:** Vorlesungen  $L_1 = [s_1, f_1), \dots, L_n = [s_n, f_n)$

**Gesucht:** Das größte  $k$ , sodass es eine **konfliktfreie** Auswahl von  $k$  Vorlesungen  $L_{i_1}, \dots, L_{i_k}$  gibt:

- $i_1, \dots, i_k \in \{1, \dots, n\}$  sind paarweise verschieden
- **konfliktfrei:**  $L_j \cap L_{j'} = \emptyset$  für alle  $j, j' \in \{i_1, \dots, i_k\}$

**Hinweis:** Die Intervalle sind **halboffen** →  $[0, 1), [1, 3), [3, 9)$  sind konfliktfrei

# **Erschöpfende Suche**

# Erschöpfende Suche: Grundprinzip

---

Die vielleicht einfachste Herangehensweise:

Probiere systematisch alle möglichen Lösungen durch

**Typischer Anwendungsfall:** Wir suchen ein Objekt mit bestimmten Eigenschaften

<b>Beispiele:</b>	<b>Sortieren eines Arrays <math>A[1 \dots n]</math></b>	<b>Ambitioniertes Studieren</b>
<b>Objekt:</b>	Permutation $i_1, \dots, i_n$ von $\{1, \dots, n\}$ ,	Vorlesungs-Auswahl $i_1, \dots, i_k \in \{1, \dots, n\}$
<b>Eigenschaft:</b>	$A[i_1] \leq \dots \leq A[i_n]$ ,	$L_{i_1}, \dots, L_{i_k}$ sind konfliktfrei, $k$ maximal

**Vorgehen:** teste für jedes Objekt, ob die Eigenschaft erfüllt ist  
→ leite daraus das Ergebnis ab

**"Bruteforce":** Mit "roher Gewalt" werden alle Möglichkeiten durchprobiert.  
*Exhaustive Search*

# Erschöpfende Suche: Sortieren

---

Sortieren eines Arrays  $A[1 \dots n]$

**Objekt:** Permutation  $i_1, \dots, i_n$  von  $\{1, \dots, n\}$ ,

**Eigenschaft:**  $A[i_1] \leq \dots \leq A[i_n]$ ,

Erschöpfende Suche für das Sortieren:

Achtung: sehr vager Pseudocode

```
for all Permutationen  $i_1, \dots, i_n$  von  $1, \dots, n$  do
    if ( $A[i_1] \leq A[i_2] \leq \dots \leq A[i_n]$ ) then
        | return  $A[i_1], \dots, A[i_n]$ 
```

**Laufzeit:**

- $n!$  Permutationen
- $O(n)$  Zeit für Überprüfen der Sortiertheit

⇒ Gesamlaufzeit  $O(\textcolor{red}{(n!)}) \cdot n$

Alle Sortieralgorithmen, die wir bisher gesehen haben, sind **wesentlich schneller!**

# Erschöpfende Suche: Ambitioniertes Studieren

---

## Ambitioniertes Studieren

**Objekt:** Vorlesungs-Auswahl  $i_1, \dots, i_k \in \{1, \dots, n\}$

**Eigenschaft:**  $L_{i_1}, \dots, L_{i_k}$  sind konfliktfrei,  $k$  maximal

Erschöpfende Suche für PAS:

Achtung: sehr vager Pseudocode

```
for k from n downto 1 do
    for all Auswahlen  $i_1, \dots, i_k \in \{1, \dots, n\}$  do
        if ( $L_{i_1}, \dots, L_{i_k}$  sind konfliktfrei) then
            return k
```

**Laufzeit:** Frage: Wieviele Auswahlen von  $k$  Vorlesungen  $L_{i_1}, \dots, L_{i_k}$  gibt es?

Antwort:  $\binom{n}{k}$

Im Worst-Case betrachten wir alle möglichen Auswahlen für alle  $k = 1, \dots, n$

→ höchstens  $\sum_{k=1}^n \binom{n}{k} \leq \sum_{k=0}^n \binom{n}{k} = 2^n$  Möglichkeiten

Frage: Wie schnell können wir überprüfen, ob  $L_{i_1}, \dots, L_{i_k}$  konfliktfrei sind?

Antwort:  $O(k^2)$  durch paarweises Testen,  $O(k \log k)$  durch Sortieren nach Startzeiten.

→ Gesamlaufzeit höchstens  $O(2^n \cdot n \log n)$

keine besonders gute Laufzeitgarantie...

# Erschöpfende Suche: Fazit

---

Die vielleicht einfachste Herangehensweise:

Probiere systematisch alle möglichen Lösungen durch

- naheliegender Ansatz
- typischerweise leicht zu implementieren
- hilfreich, einen ersten Baseline-Algorithmus zu bekommen

**aber:** häufig nicht sehr effizient

**Vielleicht überraschend:**

Es gibt wichtige algorithmische Probleme, für die erschöpfende Suche zu den schnellsten bekannten Algorithmen gehört

↗ z.B. für einige NP-vollständige Probleme

# **Greedy-Algorithmen**

# Greedy-Ansatz: Grundprinzip

---

Wir bauen die gesuchte Lösung **Schritt für Schritt** auf.

In jedem Schritt treffen wir die Entscheidung, die **zum aktuellen Zeitpunkt am besten wirkt**.

**Achtung:**

In sehr vielen Fällen ist dieser Ansatz nicht optimal!

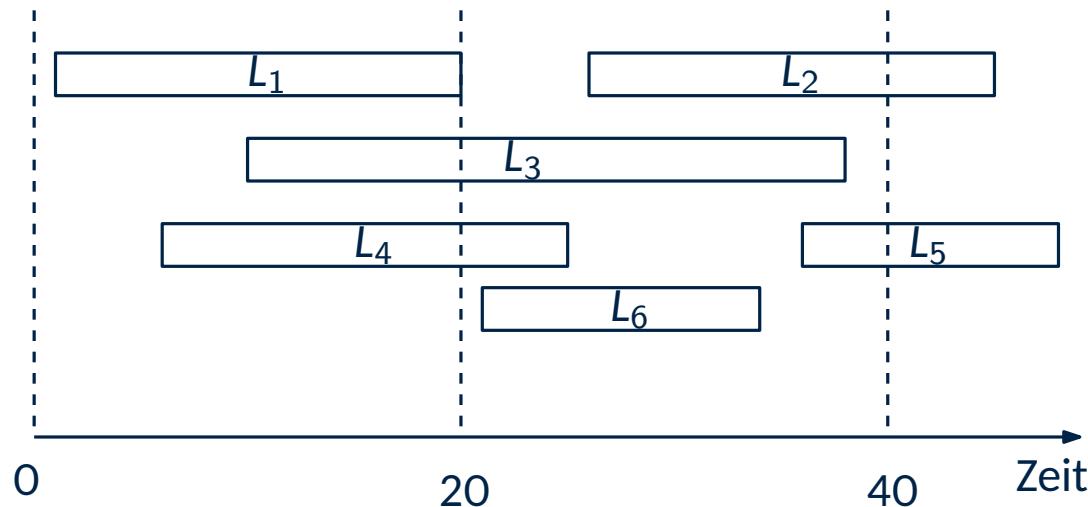
→ es kann passieren, dass wir die **falsche Entscheidung** treffen

# 1. Greedy-Ansatz für das Ambitionierte Studieren

---

## 1. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.

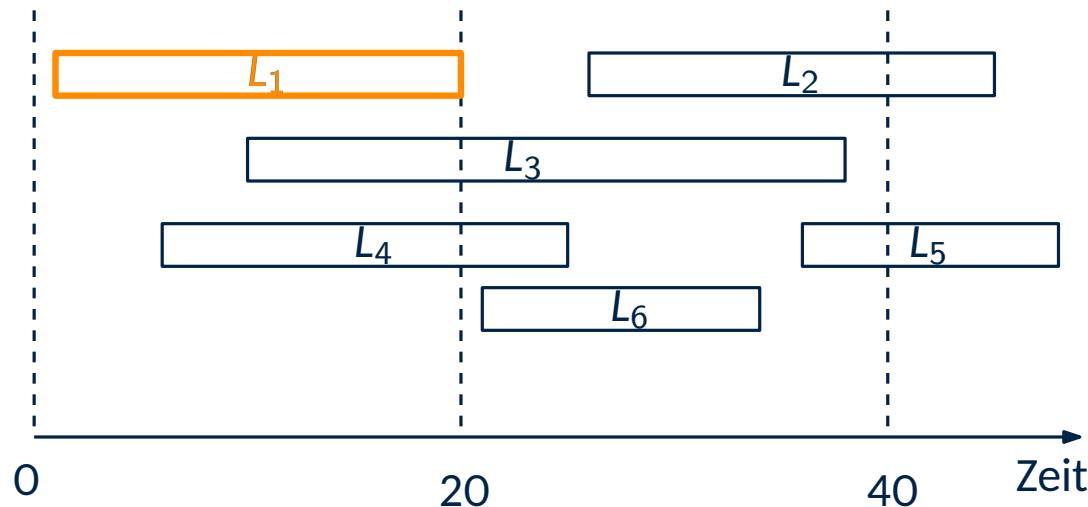


# 1. Greedy-Ansatz für das Ambitionierte Studieren

---

## 1. Idee:

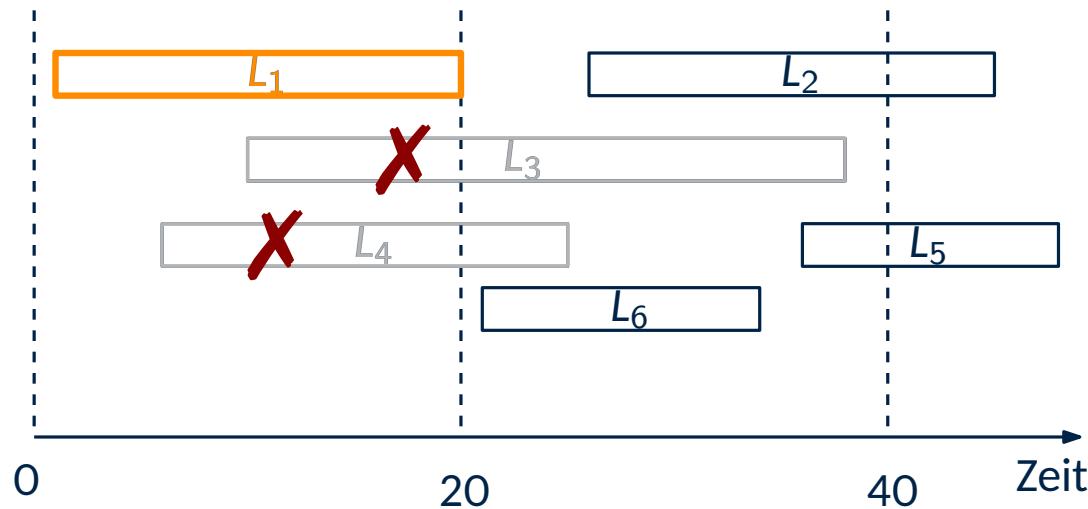
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

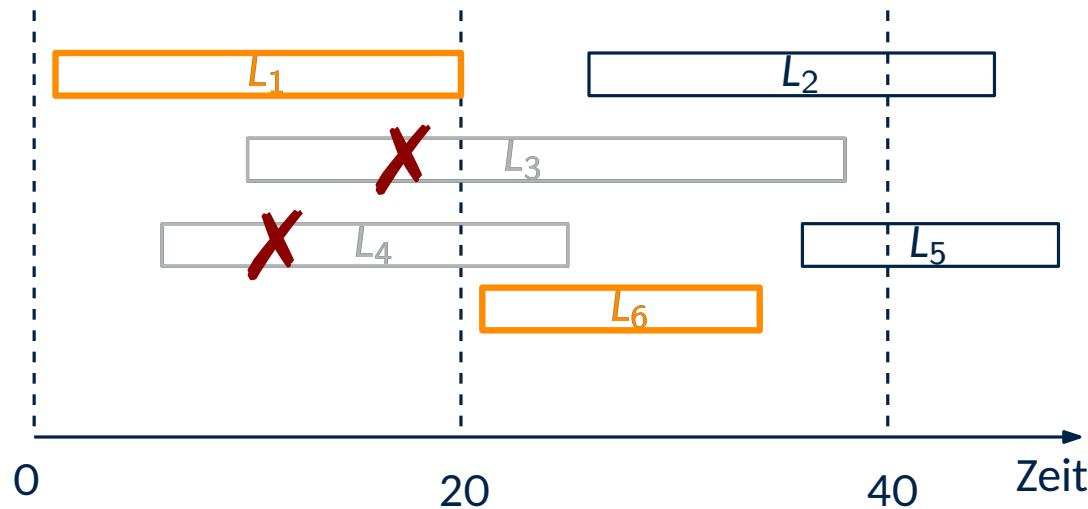
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

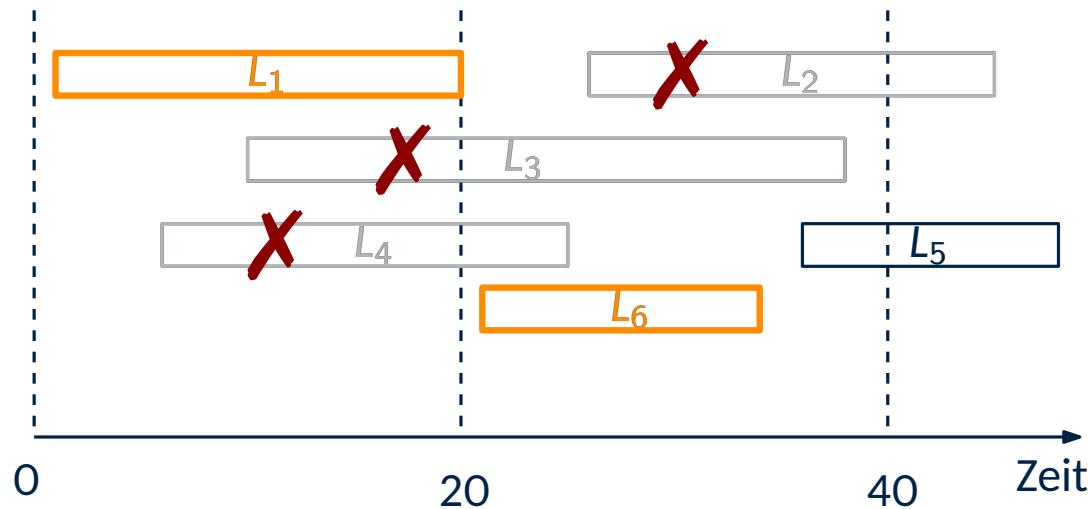
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

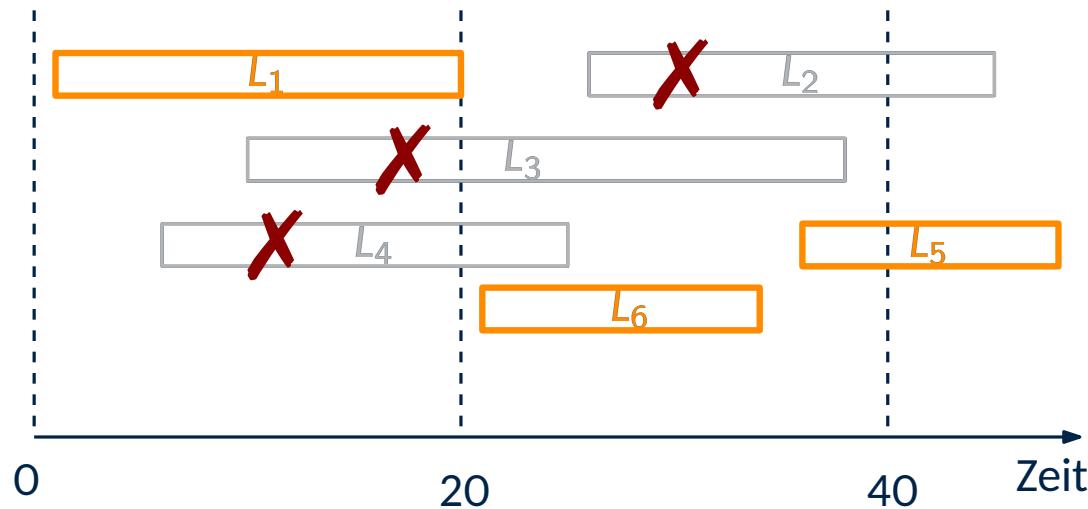
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

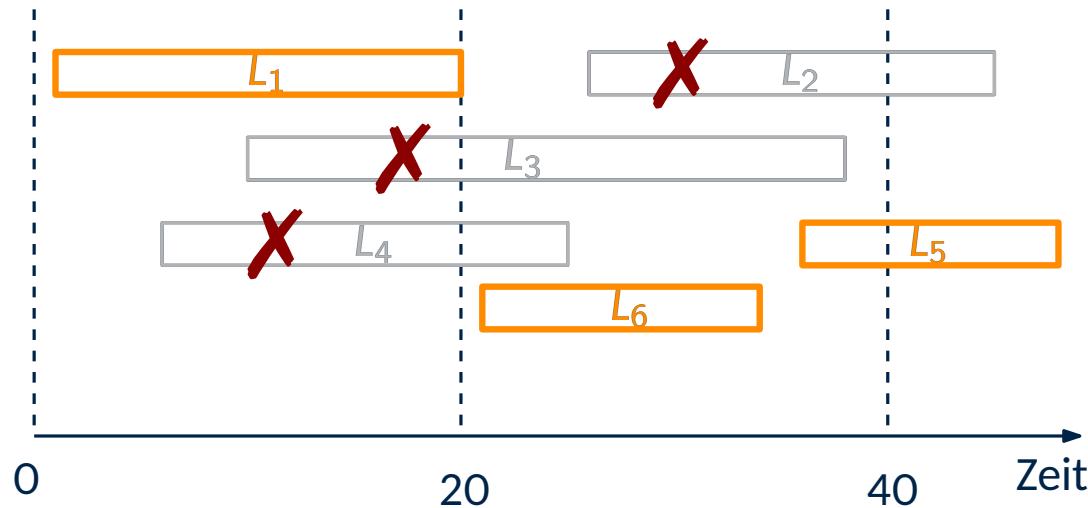
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



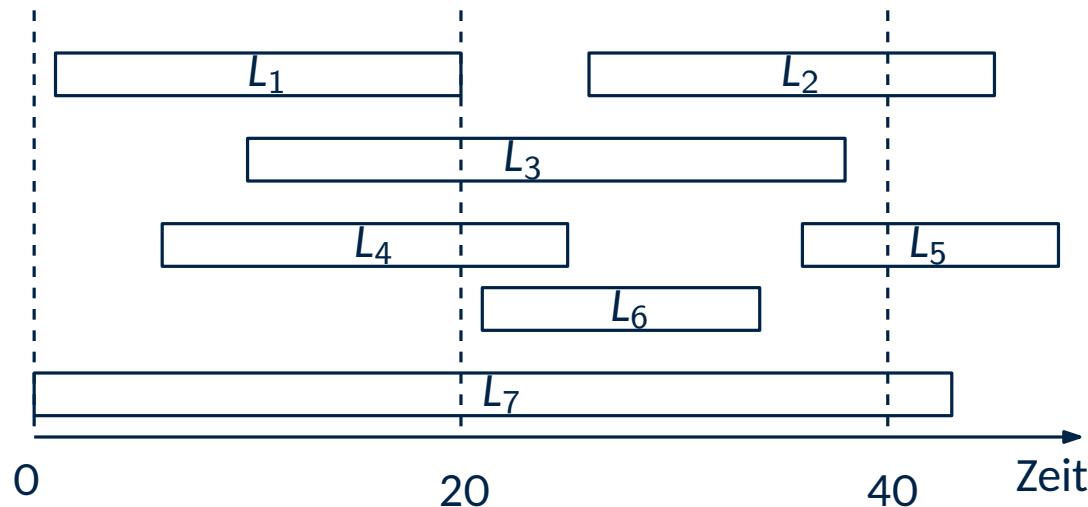
**Frage:** Liefert dieser Ansatz immer die größtmögliche Auswahl?

**Antwort:** Nein!

# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



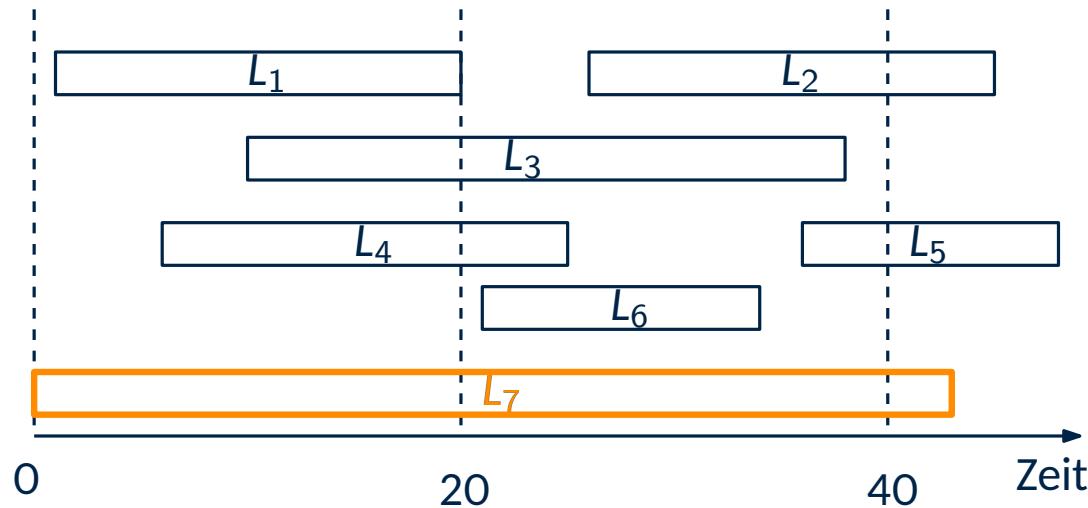
**Frage:** Liefert dieser Ansatz immer die größtmögliche Auswahl?

**Antwort:** Nein!

# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



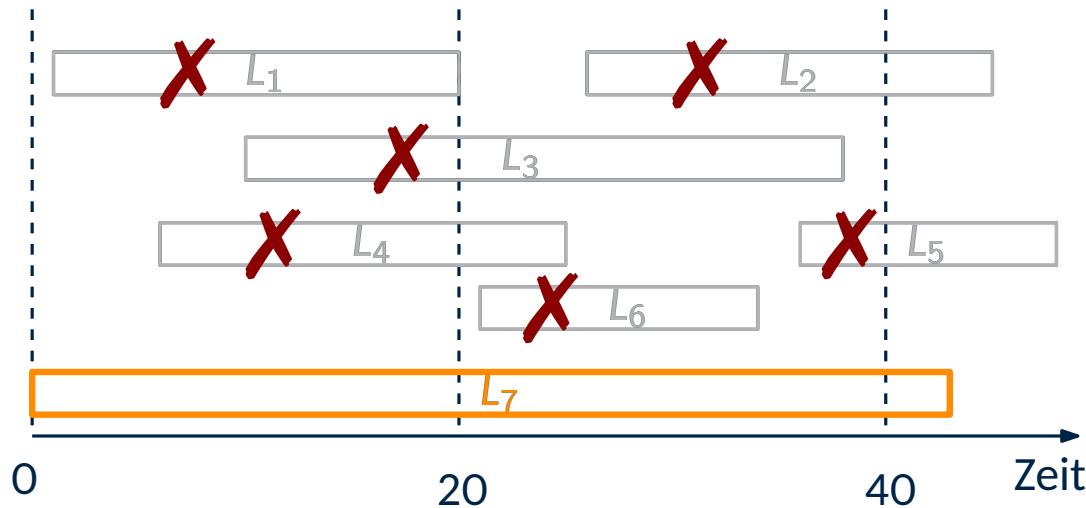
**Frage:** Liefert dieser Ansatz immer die größtmögliche Auswahl?

**Antwort:** Nein!

# 1. Greedy-Ansatz für das Ambitionierte Studieren

## 1. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich beginnt**.



**Frage:** Liefert dieser Ansatz immer die größtmögliche Auswahl?

**Antwort:** Nein!

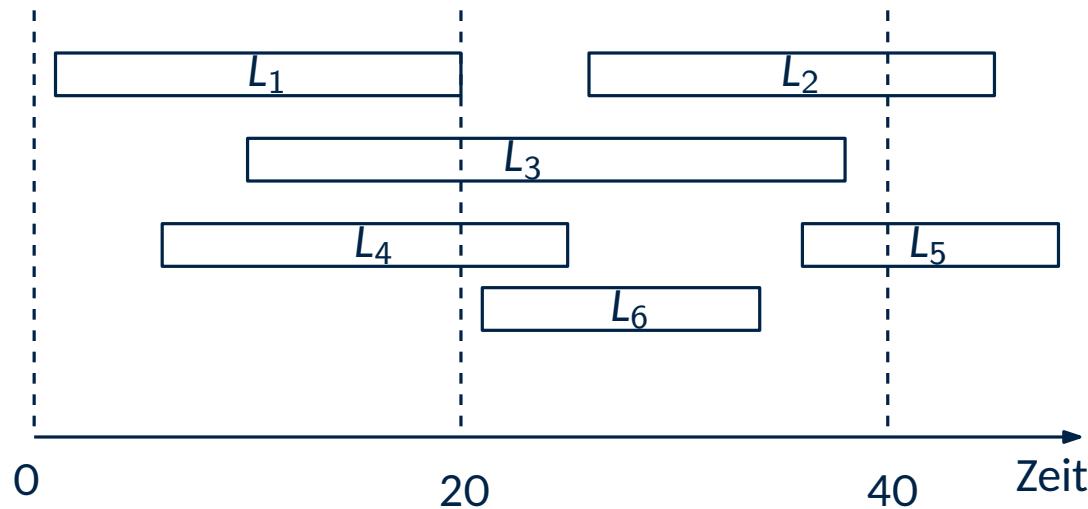
Es kann z.B. dazu führen, dass nur **eine einzelne**, aber sehr lange Vorlesung, ausgewählt wird

# 2. Greedy-Ansatz für das Ambitionierte Studieren

---

## 2. Idee:

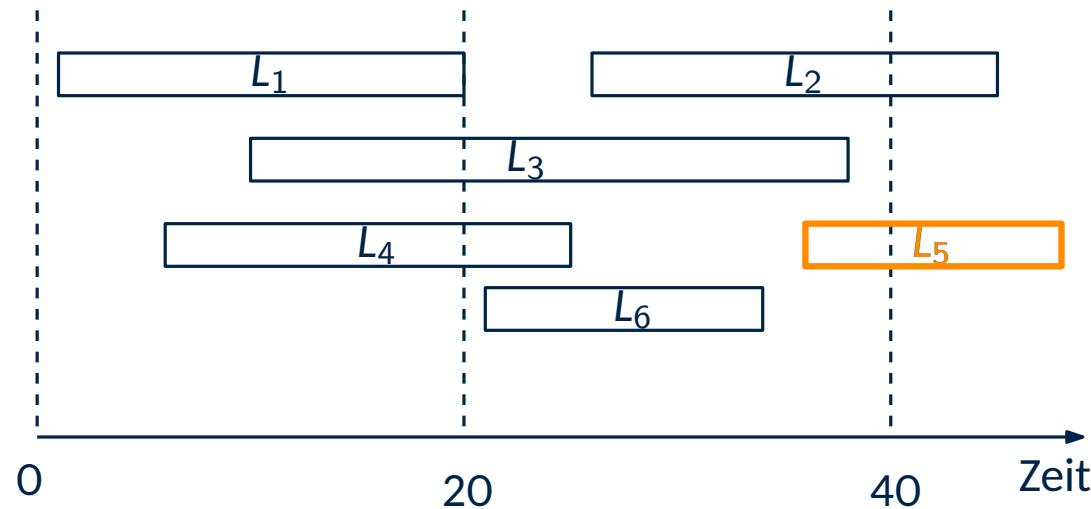
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

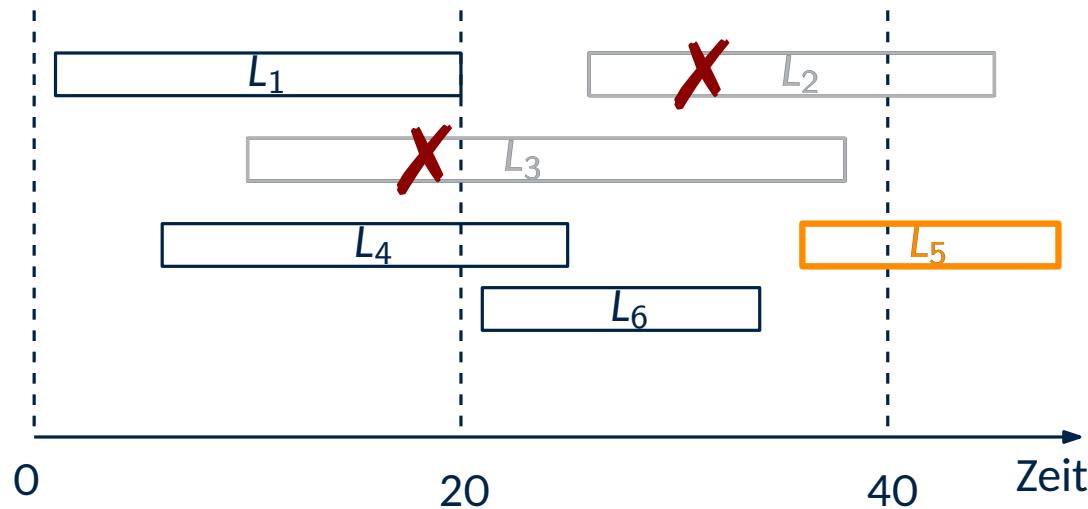
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

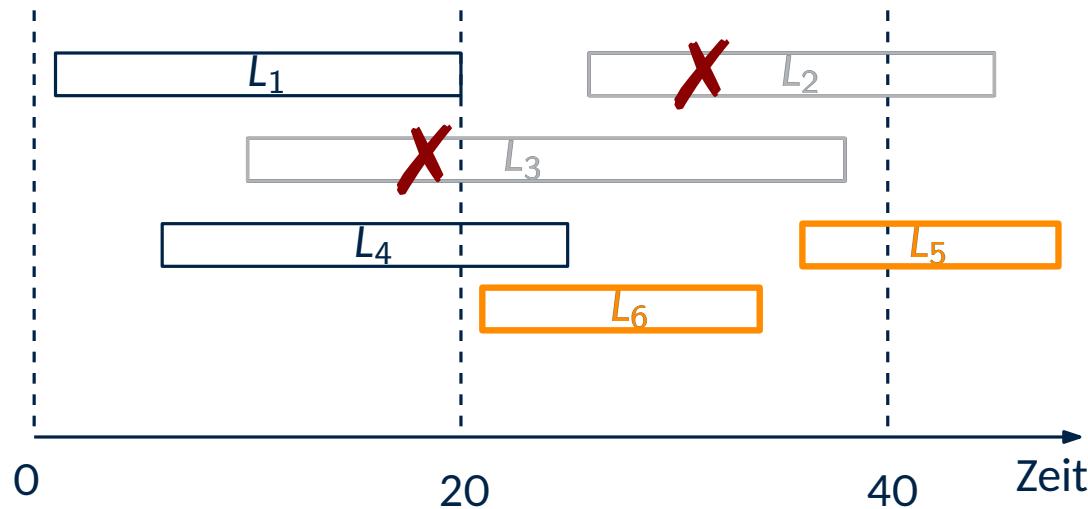
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

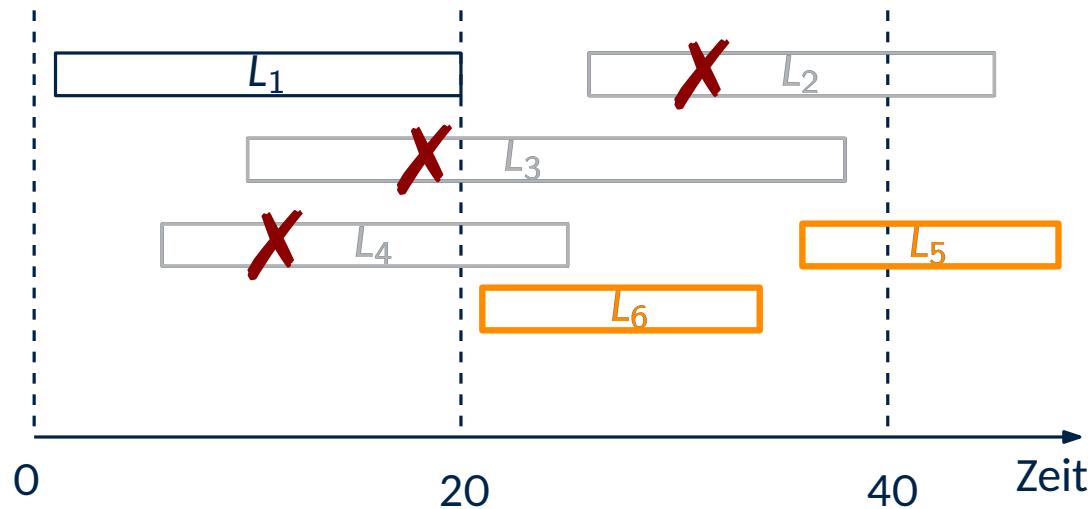
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

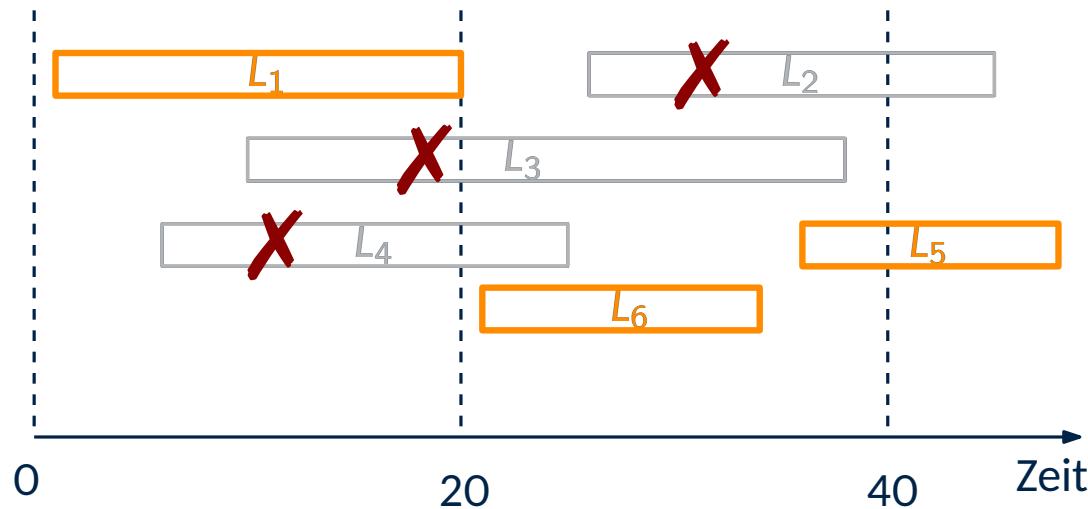
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

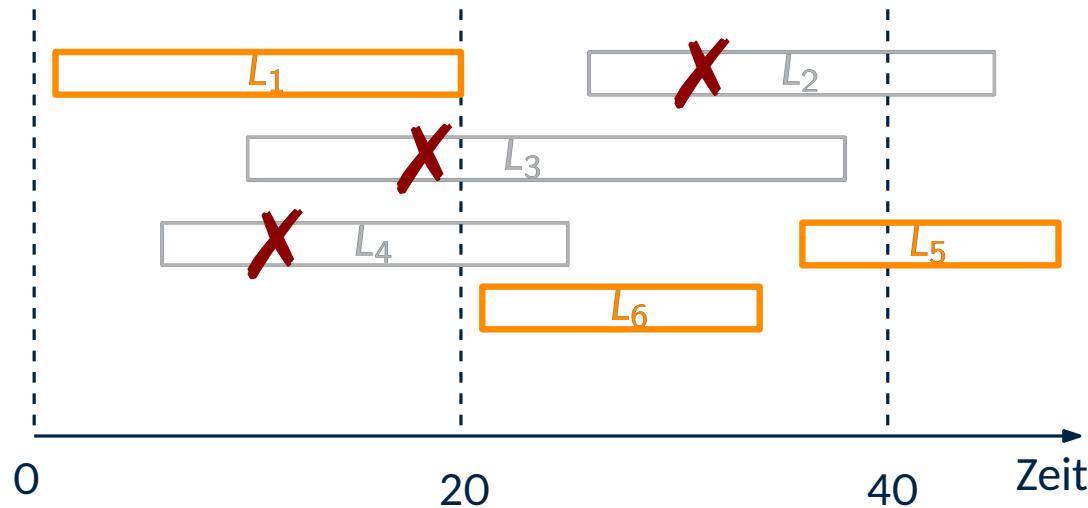
In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



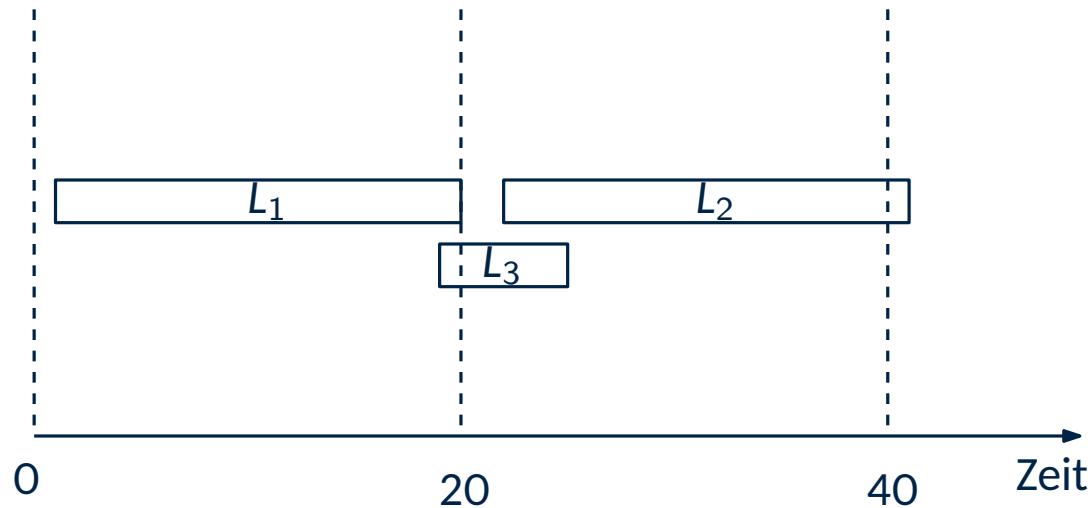
Frage: Liefert dieser Ansatz immer die größtmögliche Auswahl?

# 2. Greedy-Ansatz für das Ambitionierte Studieren

---

## 2. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



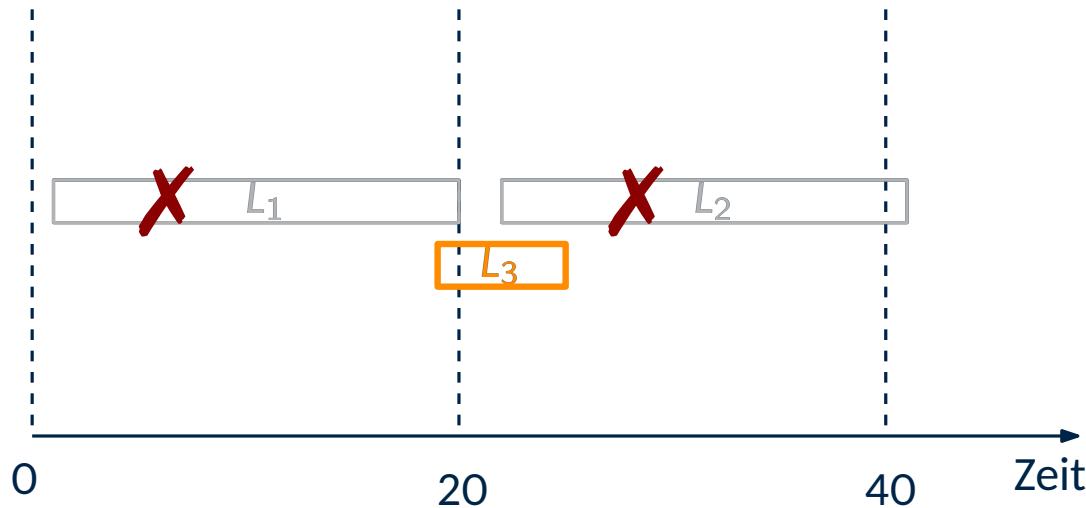
**Frage:** Liefert dieser Ansatz immer die größtmögliche Auswahl?

**Antwort:** Nein!

# 2. Greedy-Ansatz für das Ambitionierte Studieren

## 2. Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **kurz wie möglich** ist.



**Frage:** Liefert dieser Ansatz immer die größtmögliche Auswahl?

**Antwort:** Nein!

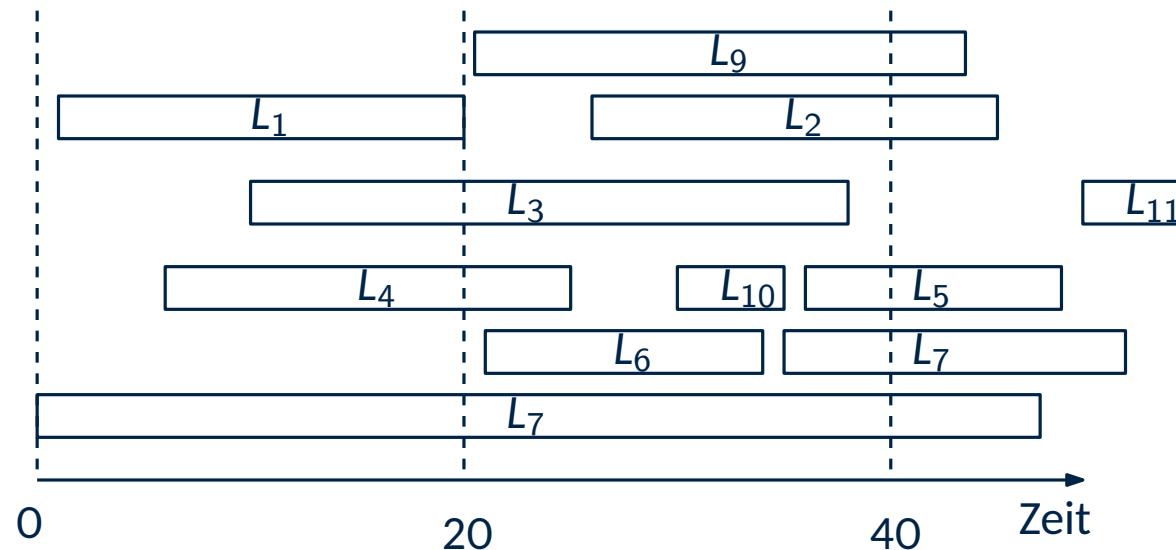
Es kann z.B. dazu führen, dass wie **eine** kurze, statt **zwei** längere Vorlesungen auswählen

# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

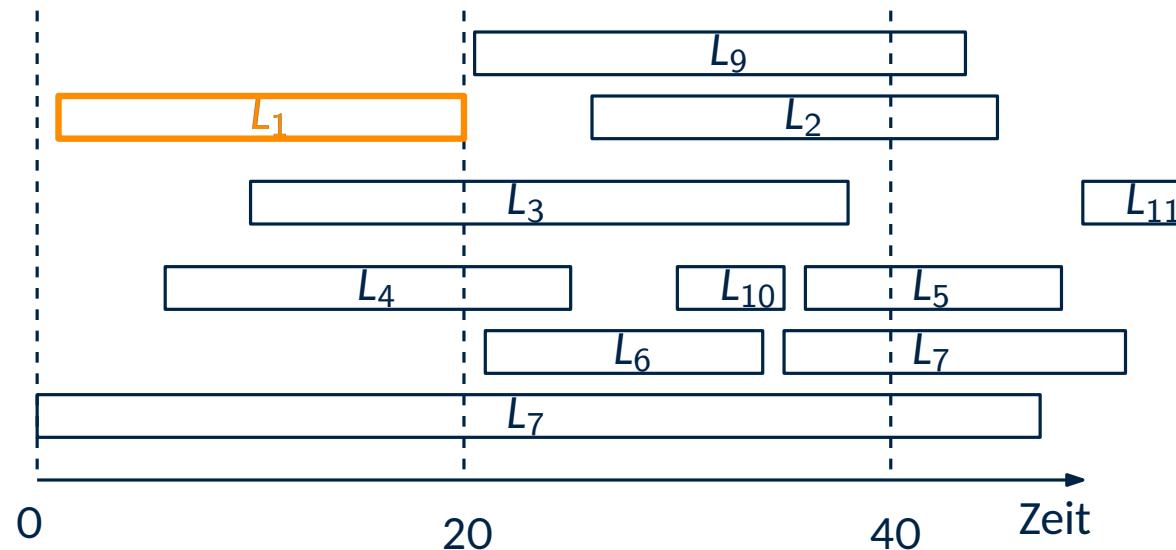


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

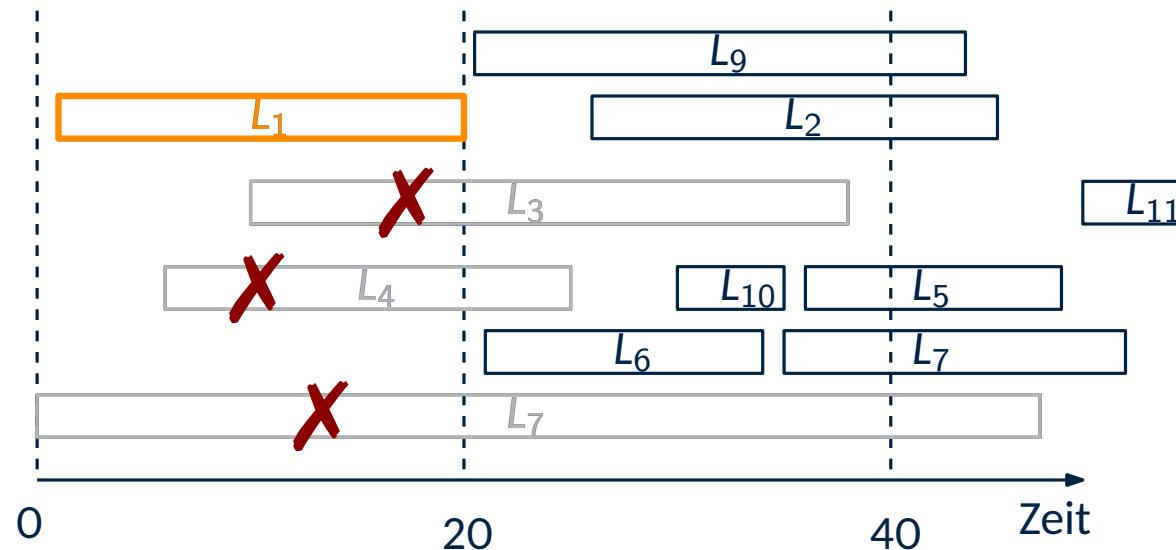


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

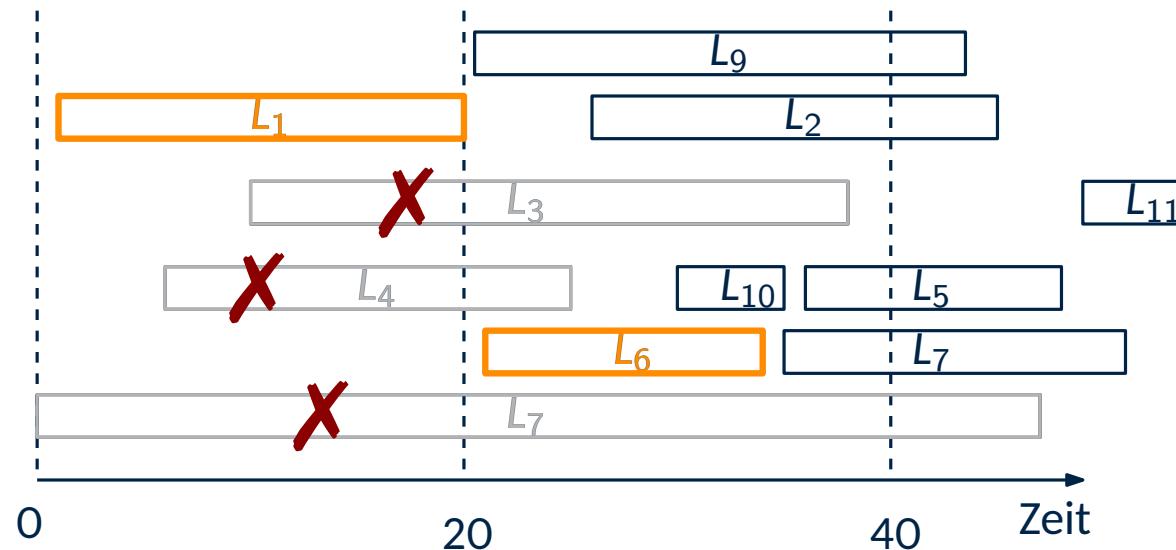


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

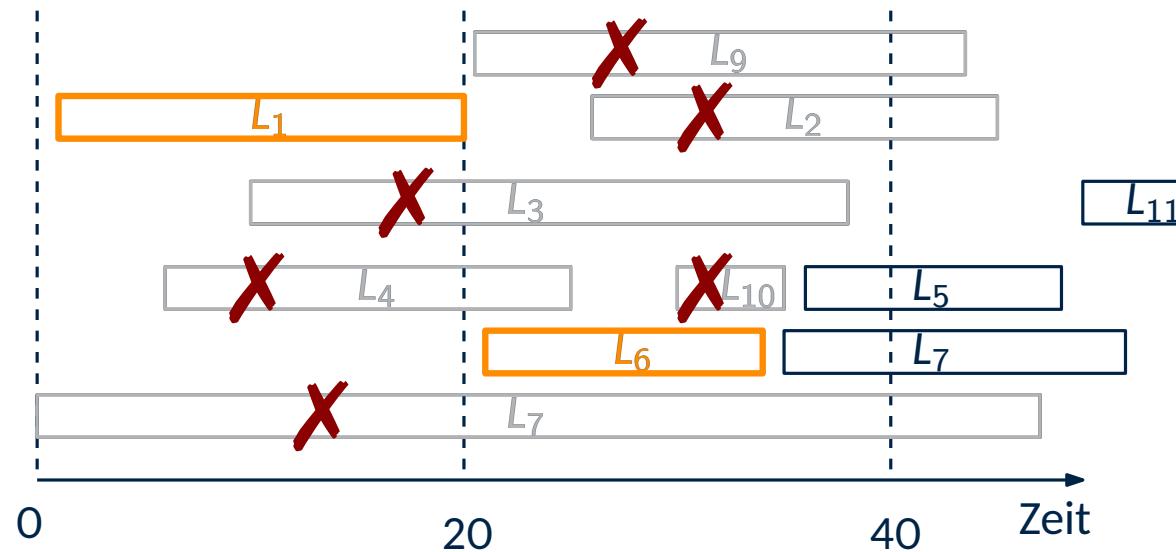


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

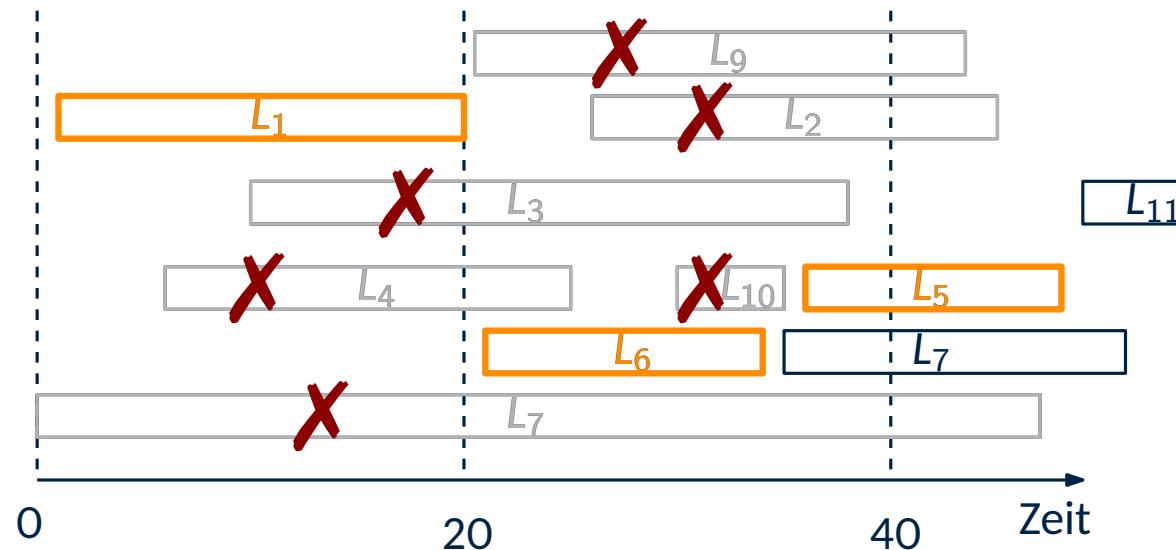


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

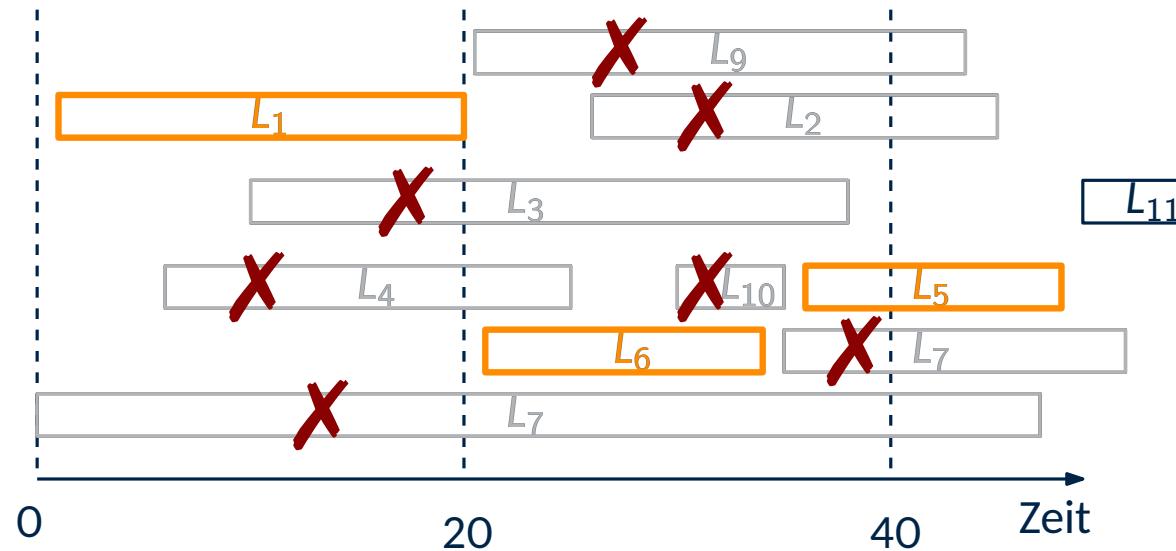


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

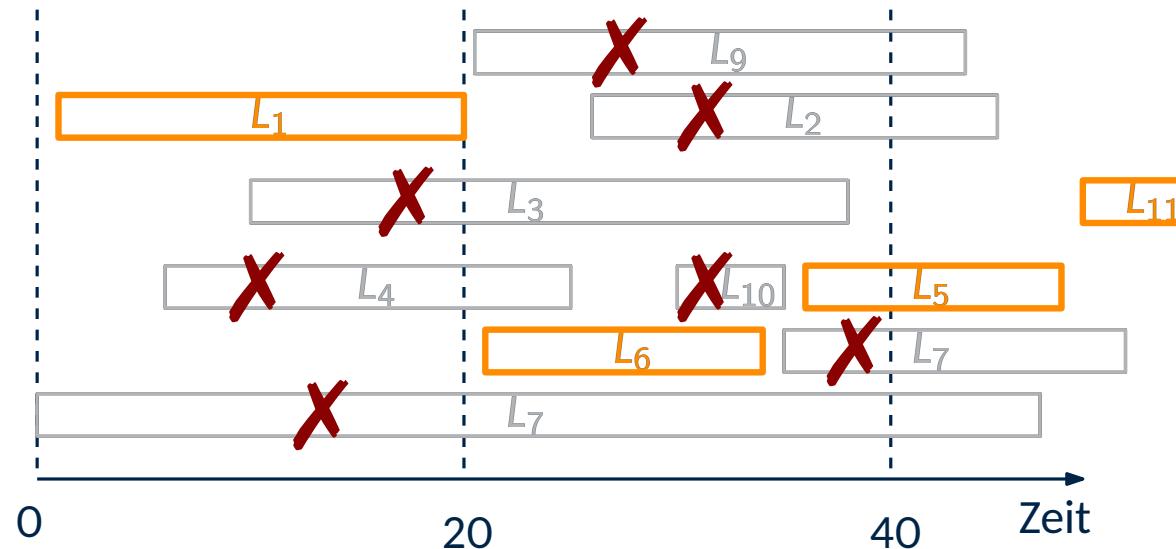


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.

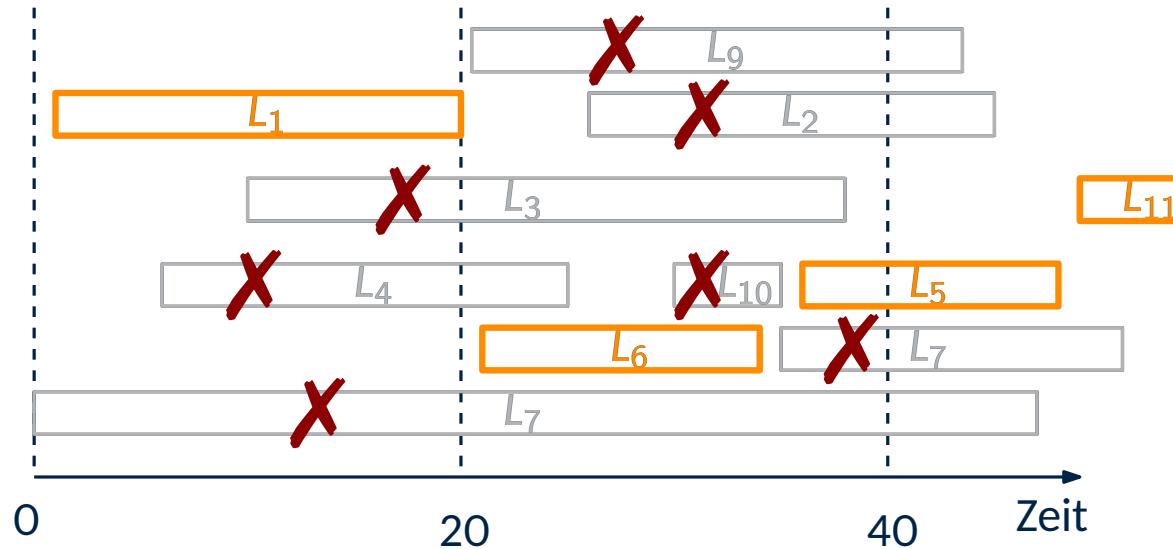


# 3. Greedy-Ansatz für das Ambitionierte Studieren

Gibt es überhaupt einen funktionierenden Greedy-Ansatz für PAS? Ja!

Funktionierende Idee:

In jedem Schritt nehmen wir die Vorlesung in unsere Auswahl auf, die so **früh wie möglich endet**.



Wir nennen diesen Ansatz **Earliest-Finish-Greedy**

**Behauptung:**

Earliest-Finish-Greedy liefert immer die größtmögliche Auswahl konfliktfreier Vorlesungen!

# Korrektheit von Earliest-Finish-Greedy

## Lemma

Earliest-Finish-Greedy (EFG) liefert immer die größtmögliche Auswahl konfliktfreier Vorlesungen.

## Beweis

Sei  $S^* = \{L_{i_1}, \dots, L_{i_k}\}$  eine optimale Lösung, nummeriert sodass  $f_{i_1} \leq f_{i_2} \leq \dots \leq f_{i_k}$

Sei  $L_{j_1}, \dots, L_{j_{k'}}$  die Auswahl von EFG, nummeriert sodass  $f_{j_1} \leq f_{j_2} \leq \dots \leq f_{j_{k'}}$

**Behauptung.** Es gilt folgende **Invariante**:

Im Schritt  $t \in \{1, \dots, k\}$  wählt EFG eine Vorlesung  $L_{j_t}$  mit  $f_{j_t} \leq f_{i_t}$

→ Invariante impliziert, dass  $k' = k$

**Induktiver Beweis der Invariante:**

1. Für  $t = 1$  gilt:  $f_{j_1} = \min_\ell f_\ell \leq f_{i_1}$

2. Die Invariante gelte für Schritt  $t$ . Wir zeigen, dass sie für Schritt  $t + 1$  gilt:

· Sei  $S_t$  die Menge an Vorlesungen, die nach  $L_{j_t}$  anfangen:

$$S_t = \{L_\ell \mid s_\ell \geq f_{j_t}\}$$

· Vorlesung  $L_{i_{t+1}}$  gehört dazu, d.h.,  $L_{i_{t+1}} \in S_t$ , denn:

$$s_{i_{t+1}} \geq f_{i_t} \geq f_{j_t}$$

↑      ↑  
 $S^*$  ist konfliktfrei    Invariante für Schritt  $t$

⇒ Behauptung folgt, da EFG die frühest endende Vorlesung aus  $S_t$  auswählt:

$$f_{j_{t+1}} = \min\{f_\ell \mid L_\ell \in S_t\} \leq f_{i_{t+1}}$$



# Earliest-Finish-Greedy: Pseudocode

```
Earliest-Finish-Greedy( $L_1, \dots, L_n$ )
```

```
sortiere die Vorlesungen sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ 
```

```
Initialisiere eine leere Liste  $S$  // speichert ausgewählte Vorlesungen
```

```
 $i = 1$ 
```

```
while ( $i \leq n$ ) do
```

```
     $S.push\_back(i)$ 
```

```
     $j = i + 1$ 
```

```
    while ( $s_j < f_i$ ) do
```

```
        |  $j = j + 1$ 
```

```
        |  $i = j$ 
```

```
return  $S$ 
```

$O(n \log n)$

$O(n)$

→ Earliest-Finish-Greedy liefert größtmögliche konfliktfreie Auswahl  $L_i, i \in S$

- Laufzeit:**
- Sortieren möglich in Zeit  $O(n \log n)$
  - danach wird jede Vorlesung höchstens einmal als  $i$  oder  $j$  betrachtet
- Earliest-Finish-Greedy läuft in Zeit  $O(n \log n)$

# Greedy-Ansatz: Fazit

---

**Wenn (!) ein Greedy-Ansatz korrekt ist, führt er häufig zu einfachen & effizienten Implementierungen**

Suche nach einem funktionierenden Greedy-Ansatz muss nicht unbedingt einfach sein.

**Wichtige Aufgabe:** Nachweis, dass er das optimale Ergebnis liefert.

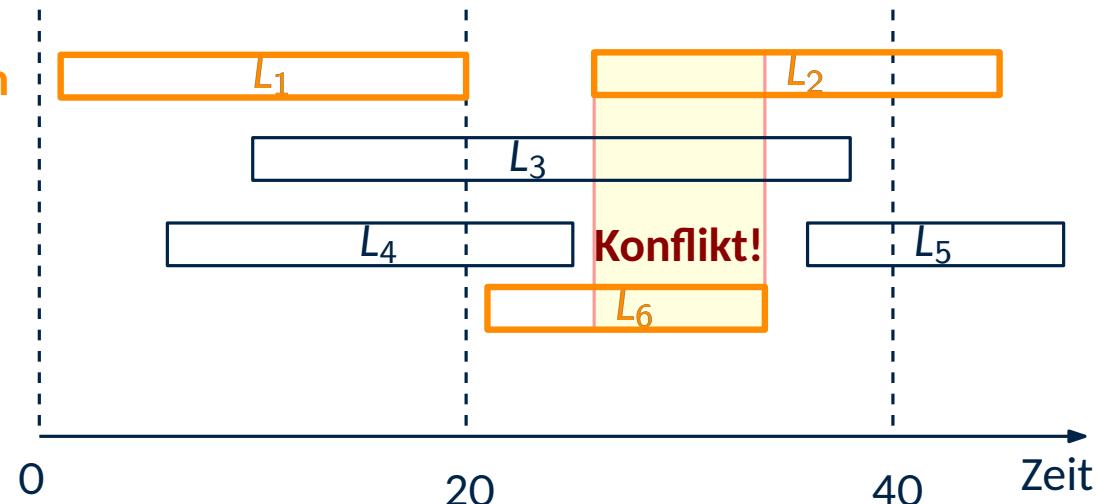
# **Dynamische Programmierung**

# Erweiterung des ambitionierten Studieren

verändertes Szenario: Qualität geht über Quantität! → wie kann ich Erkenntnisgewinn maximieren?

Beispiel: Die RPTU bietet Vorlesungen an:

	Start	Ende	Erkenntnisgewinn
$L_1$ :	1	20	1
$L_2$ :	26	45	5
$L_3$ :	10	38	100
$L_4$ :	6	25	9
$L_5$ :	36	48	1
$L_6$ :	21	34	1



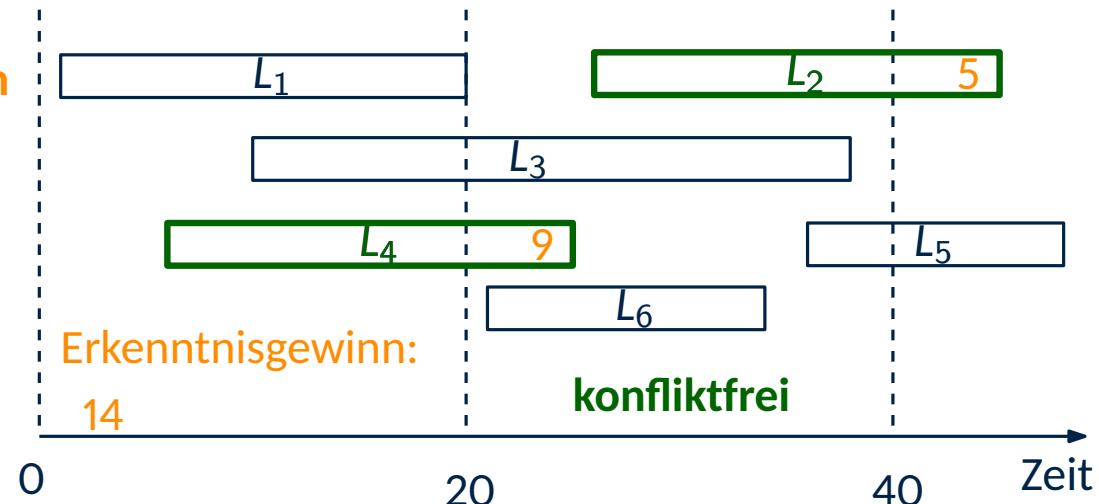
Ich kann immer noch keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")  $L_2$  und  $L_6$  überschneiden sich

# Erweiterung des ambitionierten Studieren

verändertes Szenario: Qualität geht über Quantität! → wie kann ich Erkenntnisgewinn maximieren?

Beispiel: Die RPTU bietet Vorlesungen an:

	Start	Ende	Erkenntnisgewinn
$L_1$ :	1	20	1
$L_2$ :	26	45	5
$L_3$ :	10	38	100
$L_4$ :	6	25	9
$L_5$ :	36	48	1
$L_6$ :	21	34	1



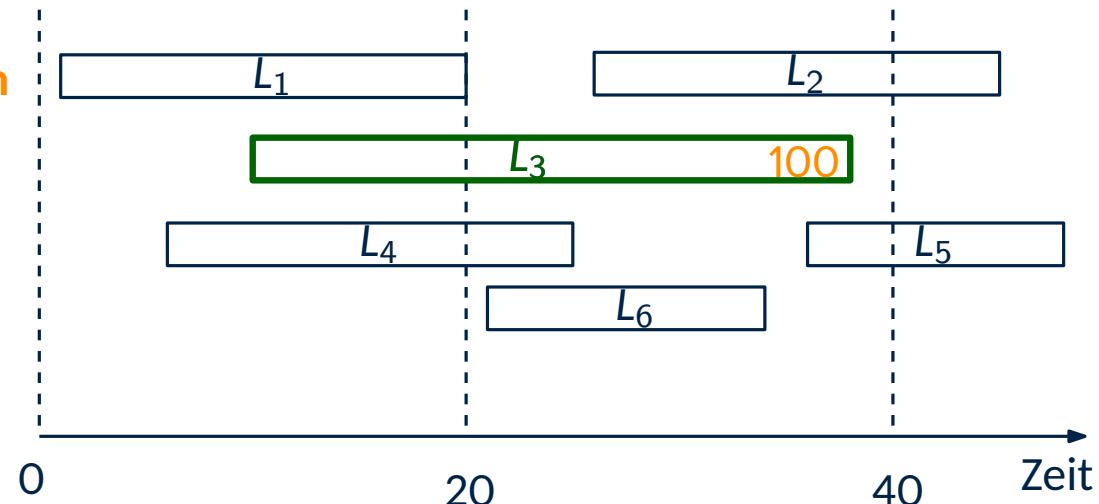
Ich kann immer noch keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")

# Erweiterung des ambitionierten Studieren

verändertes Szenario: Qualität geht über Quantität! → wie kann ich Erkenntnisgewinn maximieren?

Beispiel: Die RPTU bietet Vorlesungen an:

	Start	Ende	Erkenntnisgewinn
$L_1$ :	1	20	1
$L_2$ :	26	45	5
$L_3$ :	10	38	100
$L_4$ :	6	25	9
$L_5$ :	36	48	1
$L_6$ :	21	34	1



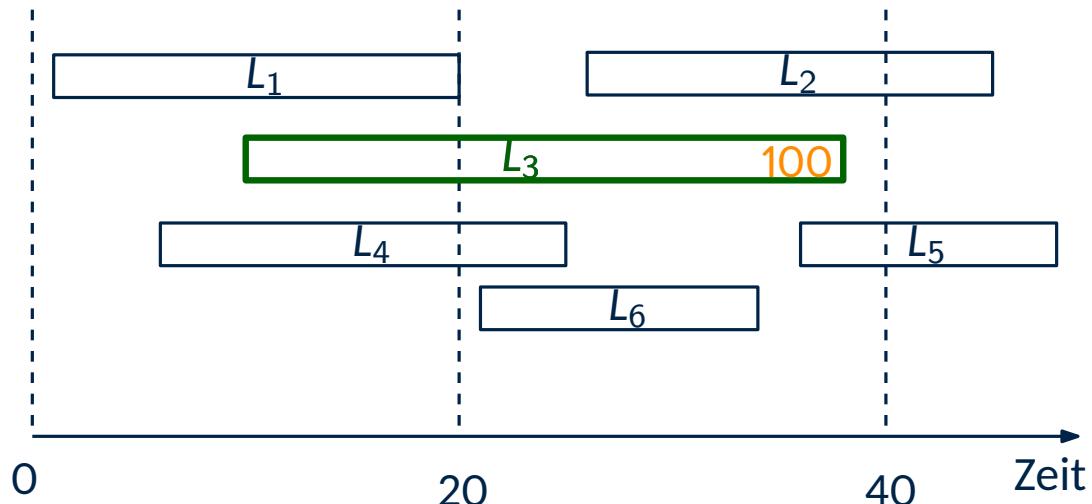
Ich kann immer noch keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")  
→ größtmöglicher Erkenntnisgewinn bei Auswahl konfliktfreier Vorlesungen: 100 (via  $L_3$ )

# Erweiterung des ambitionierten Studieren

verändertes Szenario: Qualität geht über Quantität! → wie kann ich Erkenntnisgewinn maximieren?

Beispiel: Die RPTU bietet Vorlesungen an:

	Start	Ende	Erkenntnisgewinn
$L_1$ :	1	20	1
$L_2$ :	26	45	5
$L_3$ :	10	38	100
$L_4$ :	6	25	9
$L_5$ :	36	48	1
$L_6$ :	21	34	1



Ich kann immer noch keine zwei Vorlesungen gleichzeitig besuchen ("Konflikt")  
→ größtmöglicher Erkenntnisgewinn bei Auswahl konfliktfreier Vorlesungen: 100 (via  $L_3$ )

**Problem des gewichteten ambitionierten Studierens (wPAS)**

**Gegeben:** Vorlesungen  $L_1 = [s_1, f_1], \dots, L_n = [s_n, f_n]$  mit **Profiten**  $p_1, \dots, p_n$

**Gesucht:** Eine **konfliktfreie** Auswahl von  $k$  Vorlesungen  $L_{i_1}, \dots, L_{i_k}$ , d.h.

- $i_1, \dots, i_k \in \{1, \dots, n\}$  sind paarweise verschieden,
- sie ist **konfliktfrei**:  $L_j \cap L_{j'} = \emptyset$  für alle  $j, j' \in \{i_1, \dots, i_k\}$ ,

die den **Gesamtprofit**  $\sum_{\ell=1}^k p_{i_\ell}$  maximiert.

# Dynamische Programmierung (DP): Grundprinzip

---

**Ansatz:** Wir benutzen optimale Lösungen für (überlappende) **Teilprobleme**, um eine optimale Lösung für das **Gesamtproblem** zu erhalten.

# Teilprobleme für wPAS

$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das ***i.* Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i.* Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

1.  $v_0^* = 0$  und  $v_1^* = p_1$

# Teilprobleme für wPAS

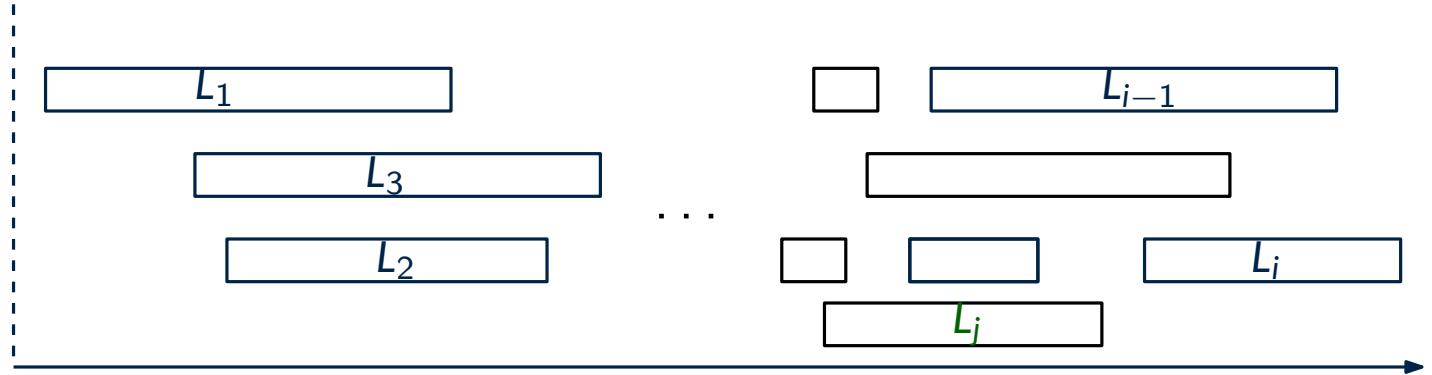
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des  $i$ . Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

# Teilprobleme für wPAS

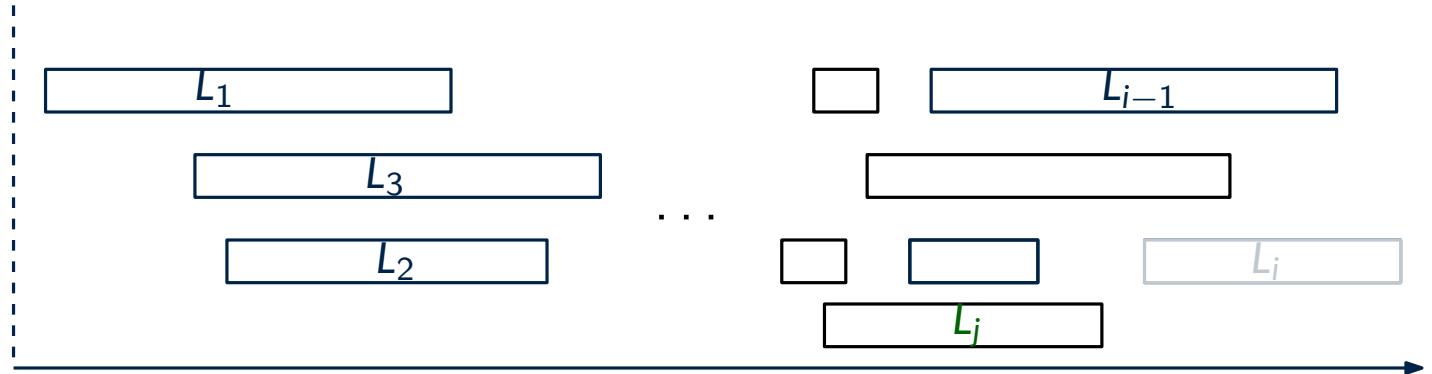
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i.* Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

# Teilprobleme für wPAS

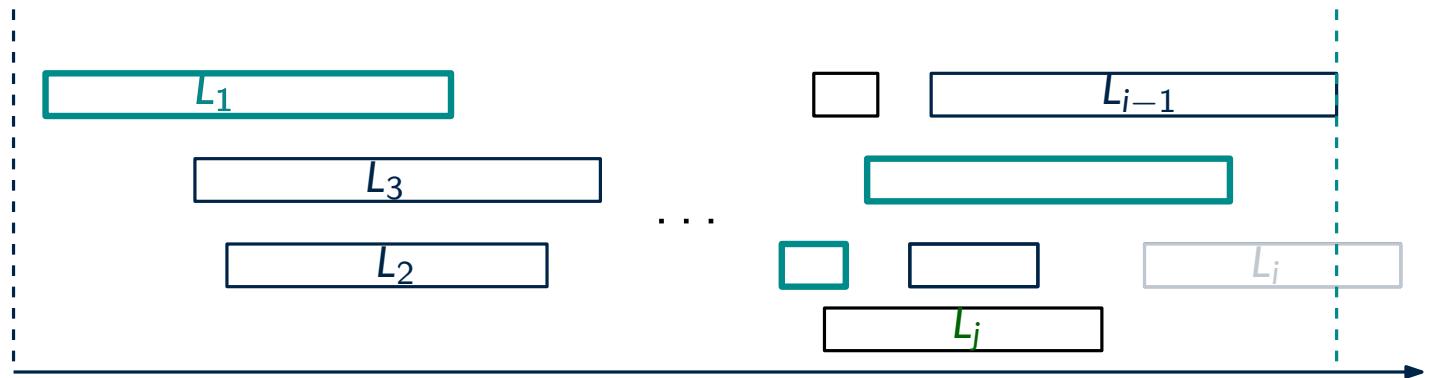
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i.* Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

maximaler Profit von Auswahl aus  $L_1, \dots, L_{i-1}$

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

# Teilprobleme für wPAS

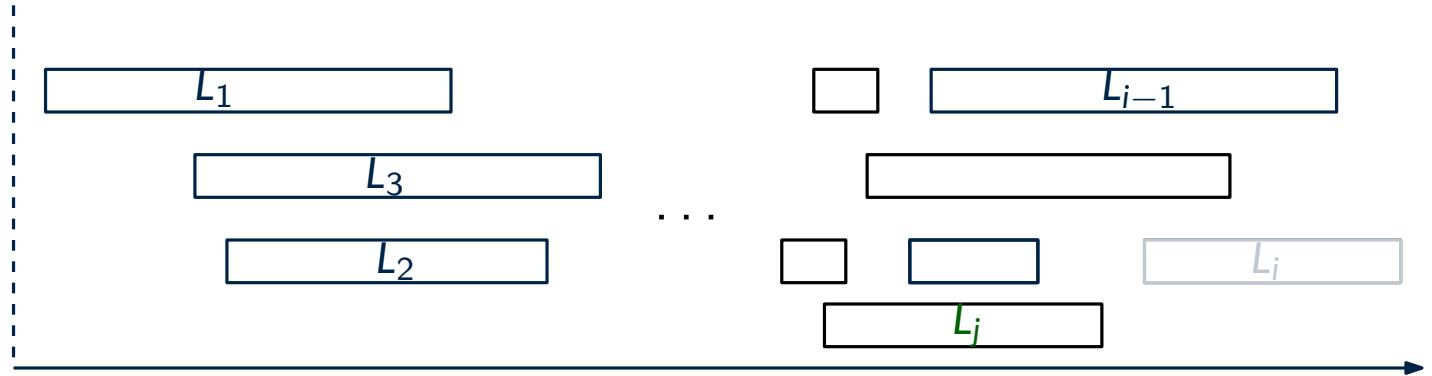
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i.* Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

# Teilprobleme für wPAS

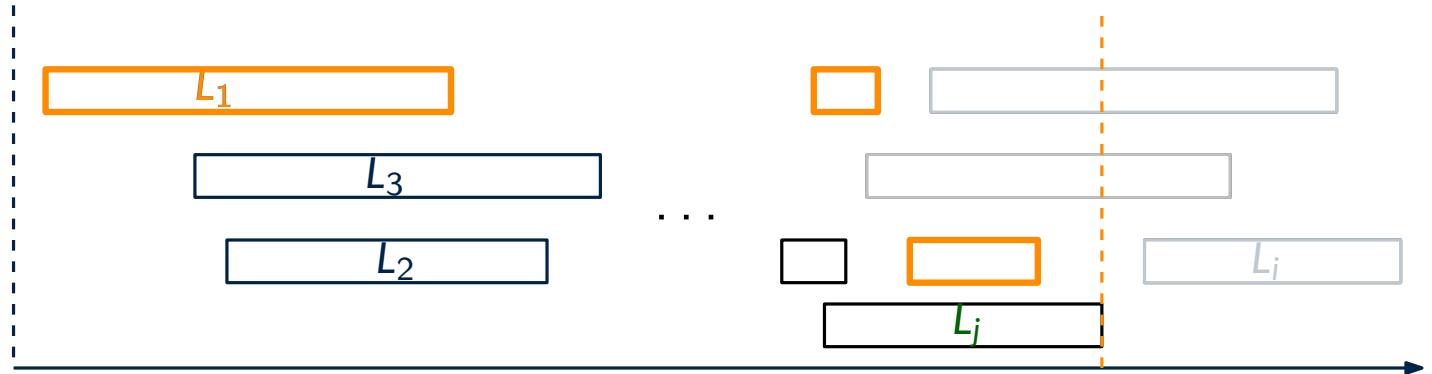
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i*. Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

maximaler Profit von Auswahl aus  $L_1, \dots, L_j$ ,

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

# Teilprobleme für wPAS

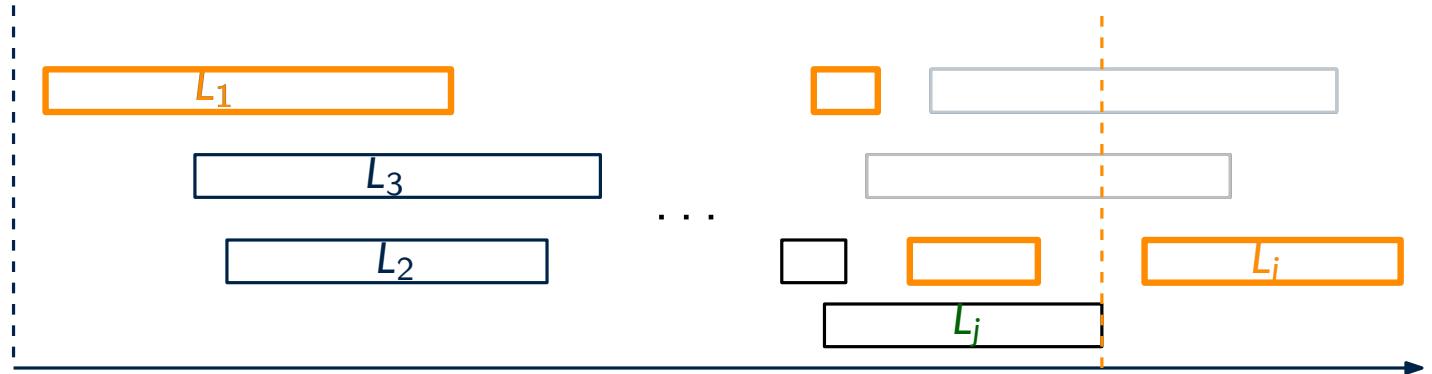
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i*. Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

maximaler Profit von Auswahl aus  $L_1, \dots, L_j$ ,  
erweitert um  $L_i$

# Teilprobleme für wPAS

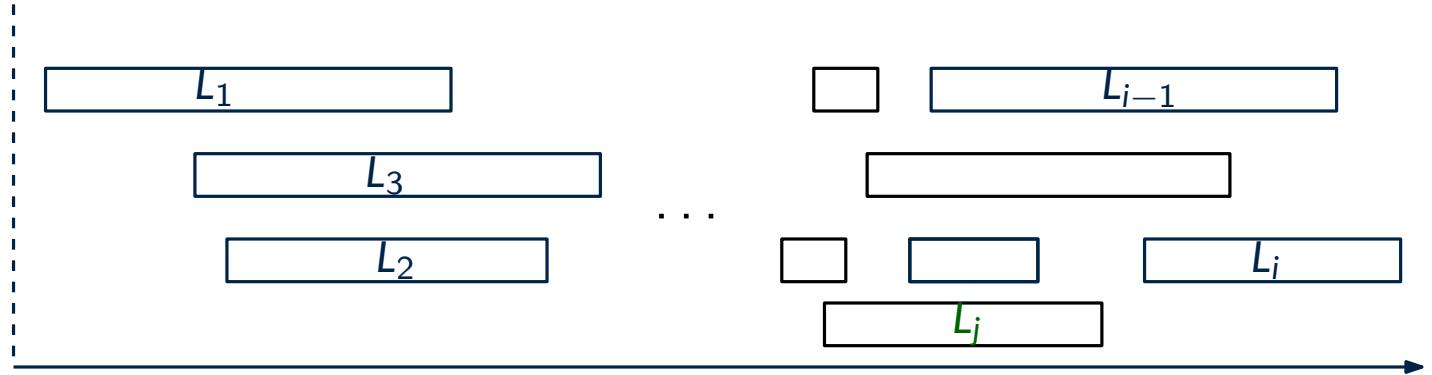
$L_1, \dots, L_n$  seien sortiert nach Endzeitpunkten, d.h.  $f_1 \leq f_2 \leq \dots \leq f_n$

**Definition (Teilprobleme für wPAS):**

Sei  $0 \leq i \leq n$ . Das **i. Teilproblem** besteht aus den Vorlesungen  $L_1, \dots, L_i$ .

Mit  $v_i^*$  bezeichnen wir den **optimale Profit** des *i*. Teilproblems.

Also:  $v_i^*$  ist der maximale Gesamtprofit einer konfliktfreien Auswahl  $S_i^*$  der Vorlesungen  $L_1, \dots, L_i$



**Lemma.**

Es gilt:

$$1. v_0^* = 0 \text{ und } v_1^* = p_1$$

$$2. \text{ Für } i \geq 2 \text{ gilt: } v_i^* = \max\{ v_{i-1}^*, p_i + v_j^* \}$$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  → letzte Vorlesung, die endet, bevor  $i$  beginnt.  
Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ , setzen wir  $j = 0$

**Eigenschaft der optimalen Teilstruktur:**

# Beweis des Lemmas

---

1.  $v_0^* = 0$  und  $v_1^* = p_1$  sind trivial. ✓
  2. Für  $i \geq 2$  gilt:  $v_i^* = \max\{v_{i-1}^*, p_i + v_j^*\}$  wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$ :
    - $S_\ell^*$  bezeichne die optimale Auswahl des  $\ell$ . Teilproblems
    - **Bemerkung:**  $S_{i-1}$  und  $\{L_i\} \cup S_i^*$  sind konfliktfreie Auswahlen des  $i$ . Teilproblems  
→  $v_i^* \geq v_{i-1}^*$  und  $v_i^* \geq p_i + v_j^*$
    - Es verbleibt zu zeigen:  $v_i^* \leq \max\{v_{i-1}^*, p_i + v_j^*\}$ :  
**Fall 1:**  $L_i \notin S_i^*$   
Dann ist  $S_i^*$  eine konfliktfreie Auswahl aus  $L_1, \dots, L_{i-1}$ .  
⇒ Profit von  $S_i^*$  ist höchstens  $v_{i-1}^*$ . ✓
    - **Fall 2:**  $L_i \in S_i^*$   
Dann darf  $S_i^* \setminus \{L_i\}$  keinen Konflikt mit  $L_i$  enthalten (alle Endzeiten  $\leq s_i$ )  
→ muss also eine konfliktfreie Auswahl aus  $L_1, \dots, L_j$  sein  
⇒ Profit von  $S_i^*$  ist höchstens  $p_i + v_j^*$ . ✓
- $L_i$        $S_i^* \setminus \{L_i\}$



# DP-Algorithmus für wPAS

**Lemma.** Es gilt:

1.  $v_0^* = 0$  und  $v_1^* = p_1$
2. Für  $i \geq 2$  gilt:  $v_i^* = \max\{ v_{i-1}^* , p_i + v_j^* \}$   
wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ ,  
setzen wir  $j = 0$

Das Lemma legt nahe, wie man  $v_n^*$  berechnen kann.

## 1. Ansatz: Rekursives Vorgehen

Um den optimalen Profit für  $L_1, \dots, L_n$  zu bestimmen:

- Wenn  $n = 1$ , gib  $p_1$  zurück. Ansonsten:
- Bestimme  $j = \max\{\ell \mid f_j \leq s_n\}$
- Berechne **rekursiv** den optimalen Profit  $v_{n-1}^*$  für  $L_1, \dots, L_{n-1}$
- Berechne **rekursiv** den optimalen Profit  $v_j^*$  für  $L_1, \dots, L_j$
- Gib  $\max\{v_{n-1}^*, p_i + v_j^*\}$  zurück

**Problem:** Wir erhalten eine exponentielle Laufzeit!

**Einsicht:** möglicherweise mehrfache Berechnung der optimalen Lösung des gleichen Teilproblems

→ **besser Ansatz: Memoization**

wir berechnen & speichern die optimale Lösung für alle Subprobleme (von  $i = 1$  bis  $i = n$ )

# DP-Algorithmus für wPAS

**Lemma.** Es gilt: 1.  $v_0^* = 0$  und  $v_1^* = p_1$

2. Für  $i \geq 2$  gilt:  $v_i^* = \max\{ v_{i-1}^* , p_i + v_j^* \}$

wobei  $j = \max\{\ell \mid f_\ell \leq s_i\}$  Wenn es kein  $\ell$  gibt mit  $f_\ell \leq s_i$ ,  
setzen wir  $j = 0$

**Tabellarischer Ansatz für wPAS:** Wir berechnen eine Tabelle  $T[0 \dots n]$ , wobei  
 $T[i] = v_i^*$  (maximaler Profit für das  $i$ -te Teilproblem)

maxProfit( $L_1, \dots, L_n$ ):

sortiere  $L_1, \dots, L_n$  sodass  $f_1 \leq f_2 \leq \dots \leq f_n$

$O(n \log n)$

Berechne  $J[1 \dots n]$  sodass  $J[i] = \max\{\ell \mid f_\ell \leq s_i\}$

**Behauptung:** möglich in Zeit  $O(n \log n)$   
Frage: Wie?

$T[0] = 0$

$O(n)$

**for**  $i = 1, \dots, n$  **do**

|  $T[i] = \max(T[i - 1], p_i + T[J[i]])$

**return**  $T[n]$

**Korrektheit:** folgt aus dem Lemma

**Laufzeit:**  $O(n \log n)$

# Weiteres Beispiel für DP: Profitsequenz

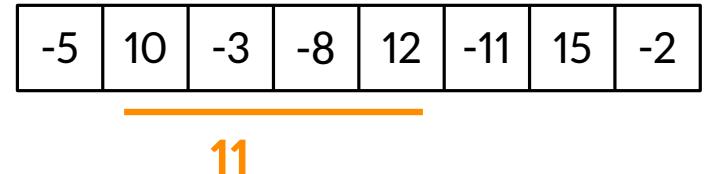
---

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$



# Weiteres Beispiel für DP: Profitsequenz

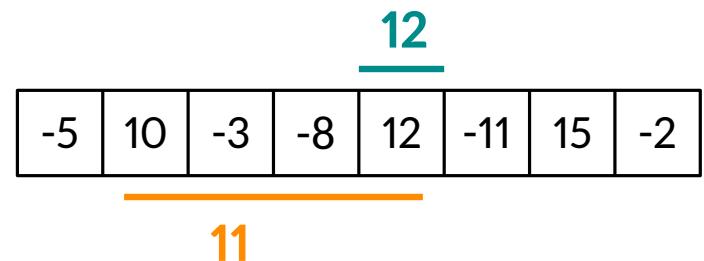
---

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$



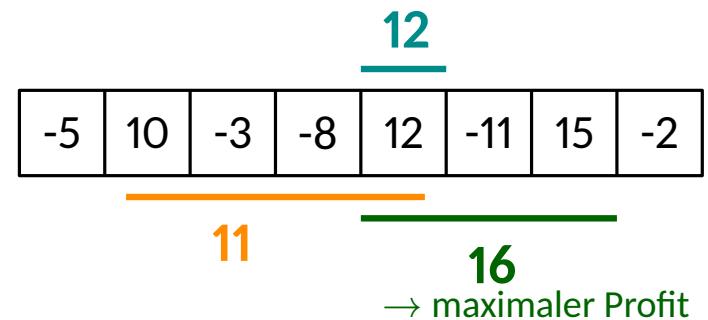
# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

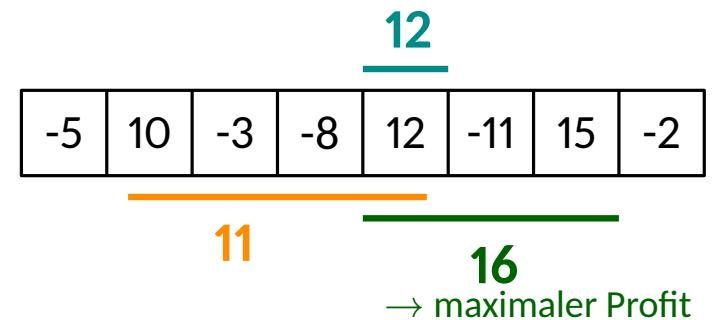
**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

**Lemma** (optimale Teilstruktur)

$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

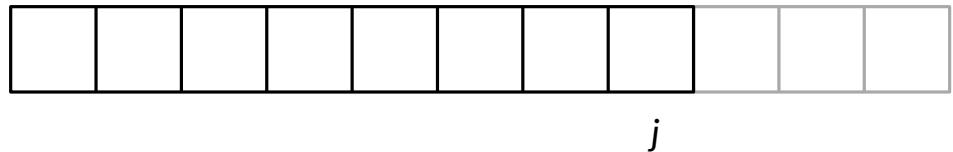
$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

**Lemma** (optimale Teilstruktur)

$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

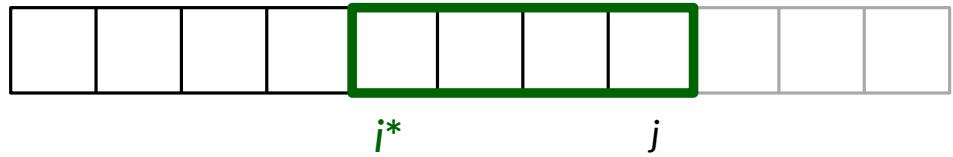
$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

**Lemma** (optimale Teilstruktur)

$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

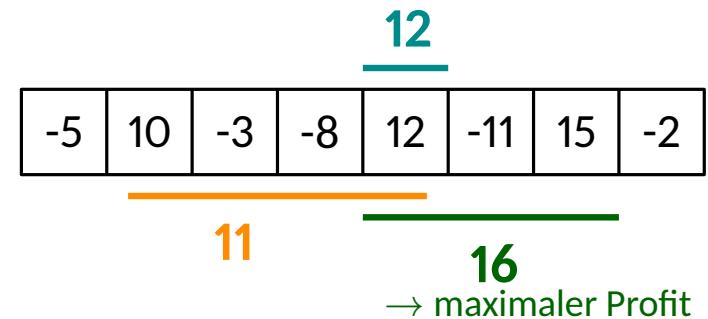
**Lemma** (optimale Teilstruktur)

$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit

**Fall 1:**  $i^* = j$



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

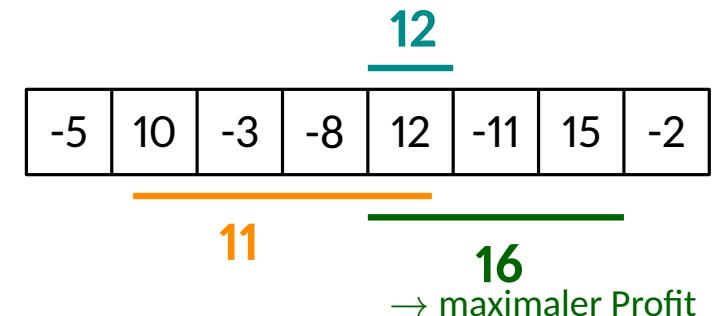
**Lemma** (optimale Teilstruktur)

$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit

$$\text{Fall 1: } i^* = j \rightarrow T[j] = P[j]$$



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

**Lemma** (optimale Teilstruktur)

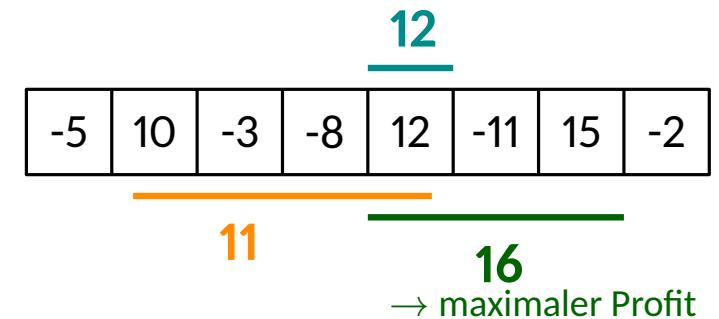
$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit

**Fall 1:**  $i^* = j \rightarrow T[j] = P[j]$

**Fall 2:**  $i^* < j$



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

**Lemma** (optimale Teilstruktur)

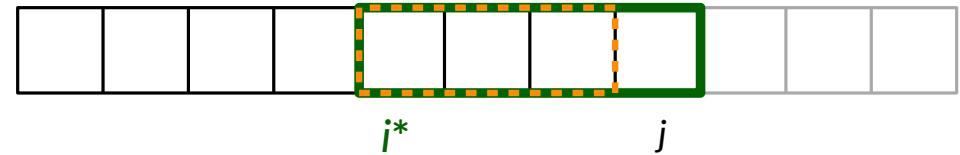
$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit

**Fall 1:**  $i^* = j \rightarrow T[j] = P[j]$

**Fall 2:**  $i^* < j$



# Weiteres Beispiel für DP: Profitsequenz

**Problem** der maximalen Profitsequenz

**Gegeben:** Array  $P[1, \dots, n]$  von Profiten in  $\mathbb{Z}$

**Gesucht:** **Interval**  $I$  mit dem maximalen Profit, d.h.

$$I = \{i, \dots, j\} \text{ sodass } \sum_{\ell=i}^j P[\ell] \text{ maximal ist}$$

Simple erschöpfende Suche: für alle  $1 \leq i \leq j \leq n$ :

berechne  $\sum_{\ell=i}^j P[\ell]$  und merke den größten Wert  
→  $O(n^3)$ -Zeit Algorithmus

Wir versuchen den DP-Ansatz:

**Idee:** Wir definieren  $T[j]$  als maximalen Profit eines Intervalls, das in  $j$  endet.

$$T[j] = \max_{1 \leq i \leq j} \sum_{\ell=i}^j P[\ell]$$

**Lemma** (optimale Teilstruktur)

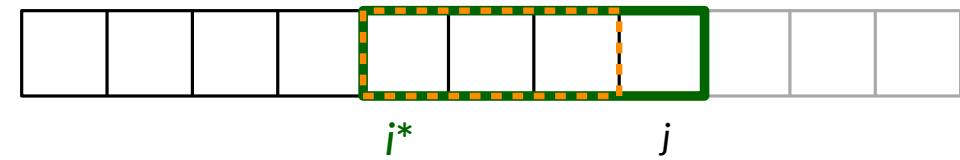
$$1. T[1] = P[1]$$

$$2. \text{ für } j \geq 2: T[j] = \max\{P[j], T[j-1] + P[j]\} = \max\{0, T[j-1]\} + P[j]$$

**Beweis:** Sei  $j \geq 2$  und  $I^* = \{i^*, \dots, j\}$  das Interval mit dem maximalen Profit

**Fall 1:**  $i^* = j \rightarrow T[j] = P[j]$

**Fall 2:**  $i^* < j$



$$\{i^*, \dots, j-1\} \text{ ist optimales Interval, das auf } j-1 \text{ endet. } \rightarrow T[j] = T[j-1] + P[j]$$

# Weiteres Beispiel für DP: Profitsequenz (2)

**Lemma** (optimale Teilstruktur)

1.  $T[1] = P[1]$
2. für  $j \geq 2$ :  $T[j] = \max\{P[j], T[j - 1] + P[j]\} = \max\{0, T[j - 1]\} + P[j]$

```
maxProfit( $P[1 \dots n]$ )
   $T[1] = P[1]$ 
  for  $j = 2, \dots, n$  do
    if ( $T[j - 1] \geq 0$ ) then
      |  $T[j] = T[j - 1] + P[j]$ 
    else
      |  $T[j] = P[j]$ 
  largest = 0
  for  $j = 1, \dots, n$  do
    if ( $T[j] > \text{largest}$ ) then
      |  $\text{largest} = T[j]$ 
  return largest
```

→ maxProfit bestimmt den **Profit** der optimalen Lösung  
kann aber leicht angepasst werden, um ein optimales **Interval** zurückzugeben

# DP: Fazit

---

**Dynamische Programmierung** ist eine sehr hilfreiche Herangehensweise  
→ wenn anwendbar, führt es häufig zu sehr effizienten Lösungen

**wichtige Aufgabe:** Identifizieren der **Teilprobleme**  
→ wir besprechen ein paar häufig auftretende Muster

# Muster I

---

Das algorithmische Problem hat als **Eingabe** eine **Sequenz**  $x_1, \dots, x_n$

Mögliche Definition von Teilproblemen:

Das  $i$ . Teilproblem ist:  $x_1, \dots, x_i, x_{i+1}, \dots, x_n$

→  $n$  Teilprobleme

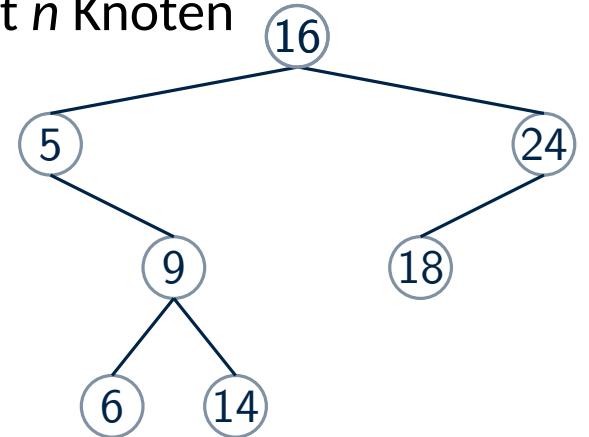
**Zentrale Überlegung:**

Wie können wir optimale Lösungen der Teilprobleme  $1, \dots, i - 1$  benutzen, um die optimale Lösung des  $i$ . Teilproblems zu bestimmen?

# Muster II

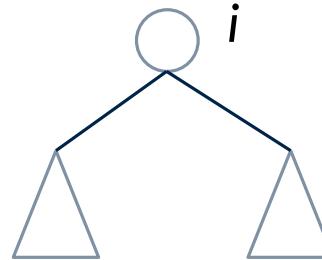
---

Das algorithmische Problem hat als **Eingabe** einen **Baum  $T$**  mit  $n$  Knoten



Mögliche Definition von Teilproblemen:

Das  $i$ . Teilproblem ist: der Teilbaum an Knoten  $i$   
→  $n$  Teilprobleme



**Zentrale Überlegung:**

Wie können wir optimale Lösungen der Teilbäume der Kinder benutzen,  
um die optimale Lösung des Baumes zu bestimmen?

# Muster III

---

Das algorithmische Problem hat als **Eingabe** eine **Sequenz**  $x_1, \dots, x_n$

Mögliche Definition von Teilproblemen:

Das Teilproblem  $(i, j)$  ist:  $x_1, \dots, x_{i-1}, x_i, \dots, x_j, x_{j+1}, \dots, x_n$

$\rightarrow \Theta(n^2)$  Teilprobleme

# Muster IV

---

Das algorithmische Problem hat als **Eingabe zwei Sequenzen**  $x_1, \dots, x_n$  und  $y_1, \dots, y_m$

Mögliche Definition von Teilproblemen:

Das Teilproblem  $(i, j)$  ist:

$x_1, \dots, x_i, x_{i+1}, \dots, x_n$

$\rightarrow \Theta(n \cdot m)$  Teilprobleme

$y_1, \dots, y_j, y_{j+1}, \dots, y_m$

# Zusammenfassung

---

Es gibt verschiedene allgemeine Entwurfsprinzipien:

- Erschöpfende Suche (*Exhaustive Search*) ✓
  - systematisches Erkunden aller möglichen Lösungen
- Teile-und-Herrsche (*Divide & Conquer*) ✓
  - Aufteilen, rekursives Lösen & Zusammenfügen
- Greedy-Algorithmen ✓
  - beste Entscheidung zum aktuellen Zeitpunkt
- Dynamische Programmierung ✓
  - tabellarisches Bestimmen optimaler Lösungen für Teilprobleme
- randomisiertes Sampling
  - Zufallsmethoden zur Bestimmung guter Lösungen
- viele weitere: Lokale Suche, Lineare Programmierung, Gradient Descent, ...

Hinweis: Diese Techniken sind **nicht immer** sinnvoll anwendbar

Algorithmen und Datenstrukturen SS'23

# Kapitel 19: Zusammenfassung & Ausblick

Marvin Künnemann

AG Algorithmen & Komplexität

# Thematische Schwerpunkte

---

Eingabe →  → Ausgabe

## Grundbegriffe und Handwerkszeug

- Was ist ein Algorithmus und wie analysiere ich ihn? ✓
- Korrektheit und asymptotische Laufzeitschranken ✓
- formale Methoden und Beweistechniken ✓

## Algorithmen und Datenstrukturen

- Elementare Datenstrukturen ✓
- Suchen, Sortieren, Wörterbücher ✓
- Spezielle Strukturen: Listen, Graphen ✓

## Entwurfsmethoden

- Komplexe Algorithmen aus einfachen Algorithmen aufbauen ✓
- Standardtechniken für bestimmte Problemklassen ✓

## Theorie

- Komplexitätstheorie: Grenzen der effizienten Berechenbarkeit ✓

# Persönliche Bemerkungen

---

- wir hoffen, Sie konnten einiges lernen!
- investierte Zeit lohnt sich → algorithmisches Denken
- Ihre konstruktiven Kommentare (VLU etc.) sind wichtig und gern gesehen
  - können zwar nicht immer sofort umgesetzt werden
  - sorgen aber für Verbesserungen für diese & kommende Iterationen

**Vielen Dank für das Feedback!**

# Weitere Themen der Algorithmik

---

- **Fortgeschrittene Algorithmen:**
  - Graphalgorithmen: maximale Flüsse, matchings, APSP, ...
  - Stringalgorithmen: String-Matching, Sequenz-Analyse, ...
  - geometrische Algorithmen: Konvexe Hülle, Orthogonal Range Search, ...
  - Optimierung: Linear Programming, SDP, gradient descent, ...
  - algebraische Methoden, ...
- **Approximationsalgorithmen**
- **Exponentialzeitalgorithmen**
- **randomisierte Algorithmen**
- **Algorithm Engineering**
- ...

# Weitere Themen der Komplexitätstheorie

---

- **Fortgeschrittene Themen zur Komplexitätstheorie:**

- Klassen über NP, Polynomialzeithierarchie
- Diagonalisierung und Zeithierarchietheoreme
- Platzkomplexität
- Schaltkreiskomplexität
- Power of randomness
- interaktive Beweise
- Kryptografie
- Hardness of approximation
- ...

- **Fine-grained Complexity Theory:**

Ist  $\approx n^3$  Zeit bestmöglich für APSP?

Ist  $\approx n^2$  Zeit bestmöglich für LCS?

→ neue Methoden, um genaue Laufzeitschranken selbst im Polynomialzeitbereich zu bestimmen!