Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 1 – Einführung

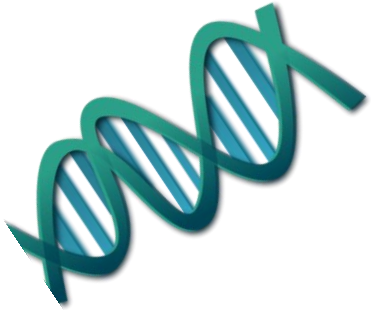
Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

1.0 Motivation zum Einstieg

- Algorithmen „allgegenwärtig“
  - Viel mehr als nur Sortieren!

- Beispiele
  - Human Genome Projekt
    - Identifizierung von Genen der menschlichen DNA
    - 3 Milliarden Elemente
    - Erfordert hochoptimierte Algorithmen
  - E-Commerce
    - Schutz privater Informationen
    - → Verschlüsseln & Authentifizieren
  - Internet
    - Datenspeicherung, Datentransport, Routing, Suchmaschinen
    - → Komprimieren, Fehlerkorrektur, Sortieren, Indizieren, …
    - Hauptarbeitsfeld der Telematiker: Kommunikation in Netzen



1.1 Organisatorisches

- **Überblick über das Institut für Telematik**
  - **Professoren**
    - Prof. Dr. Martina Zitterbart (seit 2001) - Sprecherin
    - Prof. Dr. Sebastian Abeck (seit 1996)
    - Prof. Dr. Michael Beigl (seit 2010)
    - Prof. Dr. Wilfried Juling (seit 1998)
      - Jetzt CIO des KIT
    - Prof. Dr. Hannes Hartenstein (seit 2003)
      - Geschäftsführender Direktor Steinbuch Centre for Computing
    - Prof. Dr. Bernhard Neumair (seit 2010)
      - Direktor Steinbuch Centre for Computing
    - Prof. Dr. Achim Streit (seit 2010)
      - Direktor Steinbuch Centre for Computing
    - Prof. Dr. Gerhard Krüger (seit 1971)
      - Seit 2001 emeritiert
  - **Mitarbeiter**
    - Ca. 45 wissenschaftliche Mitarbeiter
    - Technische Mitarbeiter / Verwaltungsangestellte
  - **Studierende**
    - Ca. 35 Hiwis
    - Über 2000 mündliche Prüfungen
    - Über 40 Diplomarbeiten pro Jahr

Forschungsgruppe Prof. Dr. Zitterbart

Mitarbeiter

5 Technik, Sekretariat

21 Doktoranden

4 Post-Doktoranden

Über 75% Drittmittel

Studierende

102 mündliche Prüfungen in Telematik im Jahr 2010

Forschungsgruppe Prof. Dr. Zitterbart

- Future Internet: Algorithmen, Protokolle, Architekturen

Signalling, Virtualisation, Management

Overlay Networks

Network Security

Wireless Sensor Networks

- Methoden & Werkzeuge: Evaluation, Design-Prozess

Analysis & Simulations

Testbed Experiments

Systematic Design Process

Das Vorlesungsteam

Selber aktiv werden?

- Falls Sie über die Lehrveranstaltungen hinaus Interesse haben, sich mit dem Fachgebiet vertraut zu machen, wie wäre es denn als
  - Hiwi
  - Bachelor-/Studienarbeiter
  - Master-/Diplomarbeiter
  - ... oder als aktiver Teilnehmer an einer/mehreren der Arbeitsgemeinschaften?

- Sowohl die Mitarbeiter als auch ich selbst stehen Ihnen hierzu gerne als Ansprechpartner zur Verfügung
- Schauen Sie doch einfach mal am Institut vorbei!
  - Informatikgebäude am Schloss (Geb. 20.20), 3. Stock

Termine und Organisation

- Vorlesungstermine
  - Montags, 15:45-17:15 Uhr
  - Mittwochs, 14:00-15:30 Uhr
- Übungsmaterial
  - 1 Übungsblatt alle zwei Wochen
- Große Übung
  - Mittwochs, alle zwei Wochen im Wechsel mit der Vorlesung
  - Nachbesprechung der Übungsblätter

- Tutorien
  - 30 Tutorien, über die Woche verteilt
  - Einteilung über Webinscribe
  - Vertiefung des Vorlesungsstoffes
  - Vorbereitung auf die Übungsblätter

Ansprechpartner

- Tutoren
  - Erste Anlaufstelle
  - Direkte Ansprechpartner bei Fragen und Problemen

- Übungsleiter
  - Gemeinsame Sprechstunde
    - Donnerstags, 10:00 - 11:30 Uhr
    - 1. OG rechts, Gebäude 20.50
  - Gemeinsame Emailadresse: algorithmen1@tm.uka.de

- Sprechstunde von Prof. Martina Zitterbart
  - Mittwochs, 13:00 – 14:00 Uhr
    - Genaue Termine auf http://telematics.tm.kit.edu
  - Raum 360, Gebäude 20.20

Modulaufbau

- Das Modul Algorithmen (IW2INF2, IN1INALG1) besteht im Bachelor für Informatik und Informationswirtschaft ausschließlich aus dieser Vorlesung
- Modulnote ist die Note der Algorithmen I-Klausur

- Insbesondere entfällt die bisherige Pflicht einen unbenoteten Übungsschein durch Abgabe der Übungsblätter und erreichen einer Mindestpunktzahl
- Die Abgabe der Übungsblätter ist somit freiwillig
  - Das Bearbeiten der Übungsblätter zur regelmäßigen Nachbereitung der Vorlesung ist jedoch dringend empfohlen!

- Ansprechpartner: Christian Haas <christian.haas@kit.edu>

1.2 Aufbau der Vorlesung

- Algorithmen systematisch betrachten
  1. Prinzip
     - Grundprinzip darstellen, Ablauf verstehen
  2. Beispiel
     - Illustriert das Prinzip, stellt exemplarischen Ablauf dar
  3. Pseudocode
     - Der Algorithmus im Detail
     - Vergleichsweise implementierungsspezifische Darstellung (Varianten möglich)
  4. Komplexitätsanalysen
     - Wie aufwändig ist der Algorithmus in Abhängigkeit bestimmter Eingaben
  5. Details und Wissenswertes
     - Besonderheiten, Abwandlungen und Optimierungen
     - Einsatzbeispiele

Aufbau der Vorlesung

I. Einführung

1. Einführung

- 1.0 Motivation
- 1.1 Organisatorischen
- 1.2 Aufbau der Vorlesung
- 1.3 Pseudocode
- 1.4 Komplexitätsanalyse

II. Suchen und Sortieren

2. Sortieren

III. Datenstrukturen

3. Folgen als Felder und Listen
4. Hashing
5. Heaps
6. Sortierte Listen / Bäume

IV. Graphenalgorithmen

7. Graphrepräsentation
8. Graphtraversierung
9. Kürzeste Wege
10. Minimale Spannbäume

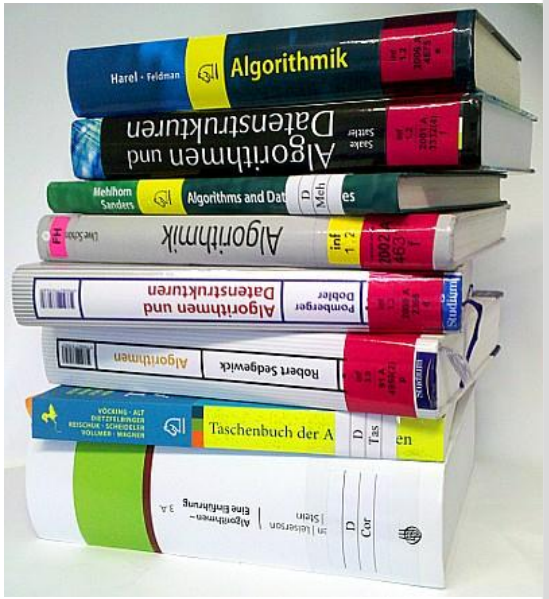
V. Ausblick

11. Generische Optimierungsansätze
12. Zusammenfassung und Ausblick

Literatur

- „Pflicht“-literatur
  - Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung“, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9

    [Corm10]



- Unsere Folien
  - http://telematics.tm.kit.edu/teachingss2011_Algorithmen.php

- Weitere Literaturhinweise
  - Dediziert auf einzelnen Folien
  - Literaturverzeichnis am Ende jedes Kapitels

1.3 Pseudocode

- Abstraktion von spezifischen Programmiersprachen
  - Enthält natürlichsprachliche Komponenten
  - Kein einheitlicher Standard
  - Keine softwaretechnischen Aspekte (Fehlerbehandlung, …)

- Im Rahmen dieser Vorlesung verwendete Syntax
  - Zuweisungen
    - $x = y^2 + \sqrt{z}$ // x wird das Ergebnis der rechten Seite zugewiesen
    - $x = y = z^2$ // x und y wird der Wert von z^2 zugewiesen
  - Felder
    - $A[1]$ // Zugriff auf das *erste* Element im Feld A
    - $A[z]$ // Zugriff auf das z-te Element im Feld A
    - $A[2..z]$ // Teilfeld von A (zweites bis z-tes Element)
    - $A.länge$ // Feldgröße

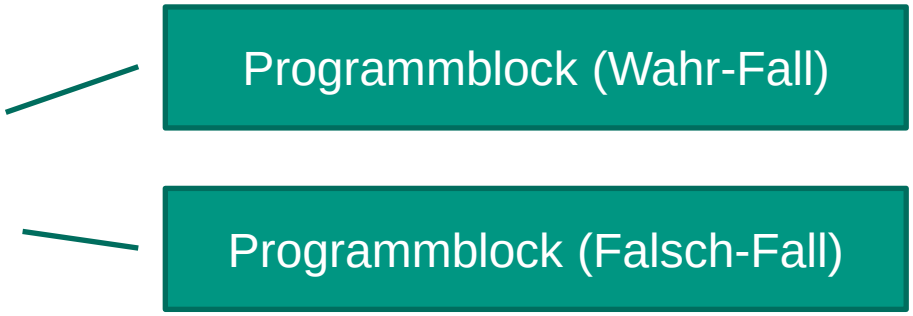
Kommentare beginnen mit //

Pseudocode – Bedingungen

- Einrückungen definieren Programmblöcke

- If-then-else Bedingung

```
1  if z > 3 und A.länge == 10
2        // falls Bedingung wahr
3  else
4        // falls Bedingung falsch
5  // Ende
```



Pseudocode – Schleifen

- Einrückungen definieren Programmblöcke
- For, while und repeat Schleifen
  - 1 **for** $i = 1$ **to** 10
    2 // irgendetwas 10x
    // durchführen
    3 // Ende, es gilt $i == 11$
  - 1 **for** $i = 10$ **downto** 1
    2 // irgendetwas 10x
    // durchführen
    3 // Ende, es gilt $i == 0$
  - 1 **while** $x < 10$
    2 // etwas ausführen
    // solange x < 10 ist
  - 1 **repeat**
    2 // etwas ausführen
    // bis x < 10
    3 **until** $x < 10$

Pseudocode – Prozeduren

- Übergabe von Parametern erfolgt call-by-value
  - Änderungen an den Parametern innerhalb der Prozedur hat keine Auswirkung auf den Aufrufer

- $PROZEDUR(x)$
  1 $x = x + 1$
  2 **return** x

- $ALGORITHMUS()$
  1 $x = 1$
  2 y = $PROZEDUR(x)$
  3 // es gilt $x == 1, y == 2$

1.4 Komplexitätsanalyse

- Aufwand
  - Die Größe der Menge der Eingabedaten für einen Algorithmus kann stark variieren.
  - Das Verhalten eines Algorithmus sollte durch die Größe der Eingabedaten nur gering beeinflusst werden
    Zum Beispiel soll ein Sortieralgorithmus 10, 100, 1000, 10000, ... Datenelemente sortieren können, ohne bei mehr Elementen bedeutend mehr Ressourcen (Zeit, Speicher) zu benötigen.

- Beim Entwurf eines Algorithmus sollte man also danach streben, dass sich der Zeit- und Speicheraufwand unterproportional zur Größe der Eingabedaten verhält.

Vergleich von Algorithmen

- Algorithmen
  - Verbrauchen Rechenzeit
    - Ausführungszeit des Programms selbst
    - In der Praxis auch: Zeit für Ein-/Ausgabe, Zeit für System,...
  - Verbrauchen Speicher für Programm und Datenstrukturen
    - Platz für das Programm selbst
    - Platz für statischen Datenstrukturen
    - Platz für dynamischen Datenstrukturen

- Fragestellungen
  - Ist Algorithmus A schneller/sparsamer als Algorithmus B?
  - Kann ein Algorithmus signifikant verbessert werden?

Vergleich von Algorithmen

- Umfang und Aufwand
  - Umfang n: Anzahl der Eingabewerte, z.B. Länge einer Liste
  - Aufwand $T(n)$: Anzahl der Zeit- bzw. Speichereinheiten, die der Algorithmus für ein Problem mit Umfang n benötigt

- Hängt der Aufwand nicht nur vom Umfang ab, sondern auch von den tatsächlichen Eingabewerten, dann interessiert ferner
  - Aufwand im schlechtesten Fall (worst-case)
  - Mittlerer Aufwand (average-case)
  - Aufwand im besten Fall (best-case)

Maschine mit wahlfreiem Zugriff (RAM)

- Zur Analyse von Algorithmen ist ein Modell der Technologie wichtig, auf der die Algorithmen implementiert werden sollen

- Einfaches Modell: „Maschine mit wahlfreiem Zugriff“
  - Ein Prozessor
  - Streng sequentiell, keine Parallelität
  - Realistischer Befehlssatz
    - Grundrechenarten
    - Daten laden, speichern, kopieren
    - Schleifen, Verzweigungen, …

- Jeder Befehl benötigt bestimmtes Maß an Zeit
  - Obige Befehle alle zeitkonstant, d.h. $O(1)$

Beispiel

- Suchen einer Zahl x in einem Feld $daten[]$ der Größe n

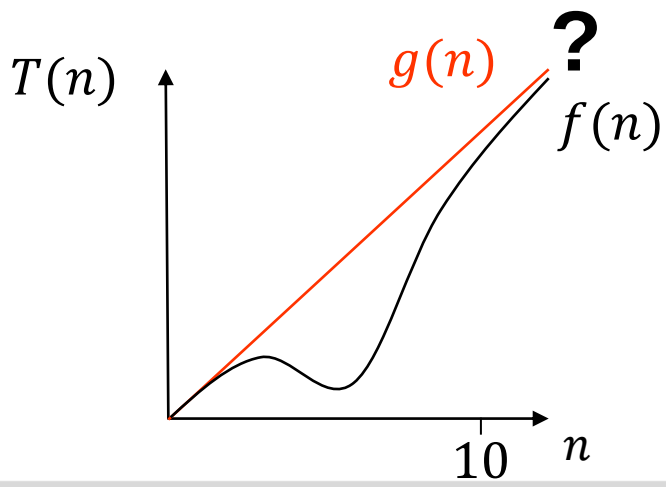
- $ZAHL\_VORHANDEN(x, daten)$

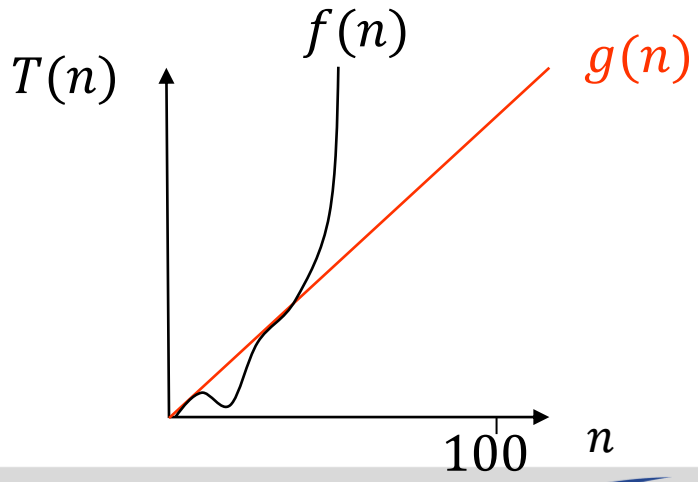
		Schritte	Schritte Maximal
1	$n = daten.länge$	1	1
2	**for** $i = 1$ **to** n	n mal:	$n + 1$
3	**if** $x == daten[i]$	1	n
4	**return** $true$	ggf. 1	1
5	**return** $false$	ggf. 1	1

- Ergebnis: $1 + (n + 1) + n + 1 = 2n + 3$

Asymptotischer Aufwand

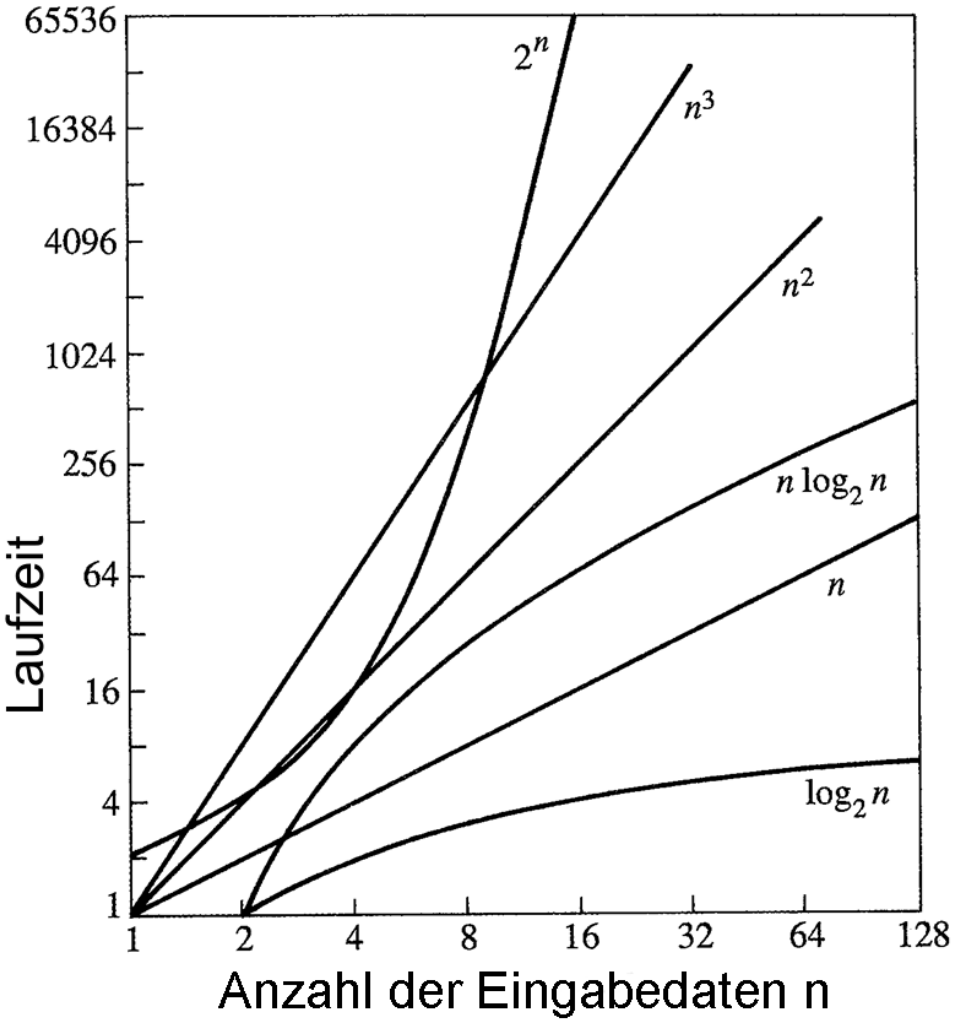
- Der exakte Aufwand eines Algorithmus ist kaum berechenbar
  - Anzahl der Rechenschritte bis zur Terminierung?
  - Verbrauchte Zeit?
- Lineare Faktoren sind für die Theorie uninteressant
  - Verschiedene Rechner sind „um Faktor“ unterschiedlich schnell
  - Wichtiger ist ungefähre Größenordnung, mit der der Aufwand $f(n)$ in Abhängigkeit der Größe der Eingabe n wächst
- Es interessiert vor allem der Aufwand für sehr große Umfänge n





1.4.1 O-Kalkül

- Um die Größenordnung des Aufwandswachstums festzulegen, wird das asymptotische Verhalten der Aufwandsfunktion zumeist mit einem Repräsentanten einer bestimmten Funktionsklasse verglichen
- Das O-Kalkül erlaubt die Beschreibung solcher Vergleiche zwischen Funktionen



Grundgedanke – Obere Schranken

- Gegeben eine irgendwie berechnete Aufwandsfunktion $f(n)$ für einen Algorithmus
- Wir suchen einen Repräsentanten $g(n)$ so, dass $f(n)$ in einer Menge $O(g(n))$ von Funktionen (Ordnung von $g(n)$) liegt derart dass für $c, n, n_0 \in \mathbb{N}$ gilt

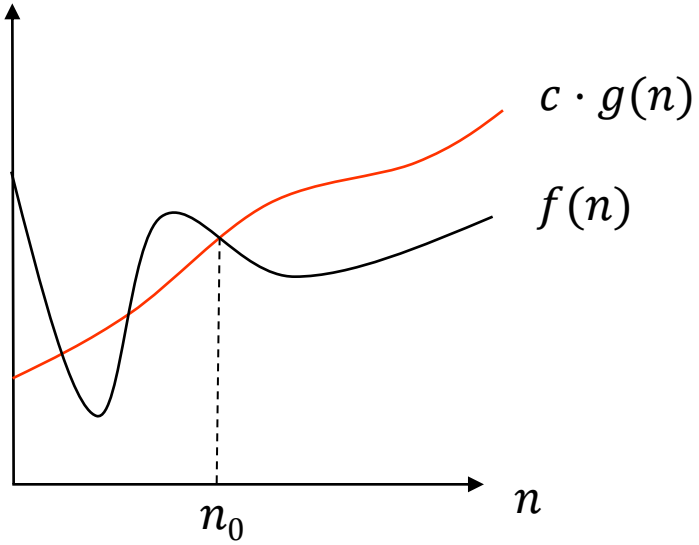
$$O(g(n)) = \{ f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \}$$

Umstiegspunkt n_0 gesucht, bei dem $f(n)$ kleiner als $g(n)$ multipliziert mit einer festzulegenden Konstante

→ Ab Umstiegspunkt n_0 wächst $f(n)$ höchstens so schnell wie $g(n)$

O-Notation

- Asymptotisch obere Schranke
  - $f(n)$ wächst höchstens so schnell wie $g(n)$
- $O\big(g(n)\big) = \{\, f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \; 0 \leq \; f(n) \; \leq \; c \cdot g(n) \,\}$



$f(n) = O(g(n))$

Alternative Schreibweise
$f(n) \in O(g(n))$

- Beispiel: $n^2 + n \; = \; O(n^2)$

Beispiel (fortgesetzt)

- $ZAHL\_VORHANDEN(x, daten)$

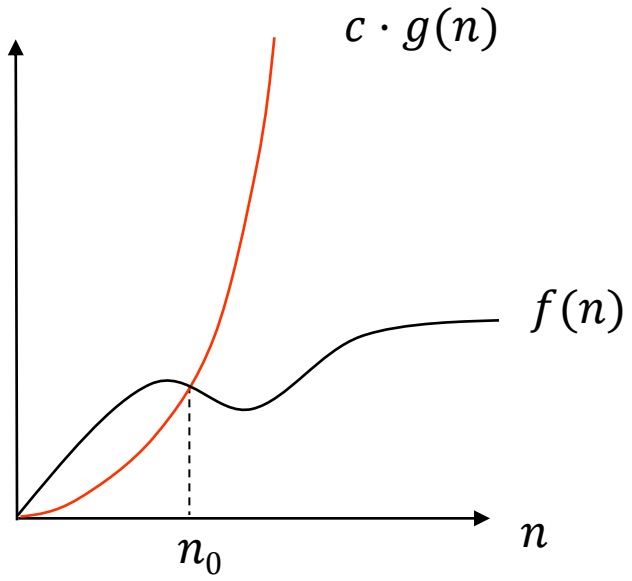
```
1   n = daten.länge
2   for i = 1 to n
3       if x == daten[i]
4           return true
5   return false
```

- Schritte, die der Algorithmus benötigt (maximal)
- Ergebnis: $f(n) = 1 + (n + 1) + n + 1 = 2n + 3$

- In O-Notation ausgedrückt
  - $f(n) = O(n)$
- Oder auch (beispielsweise)
  - $f(n) = O(n^2)$

o-Notation

- Obere Schranke
  - $f(n)$ wächst deutlich langsamer als $g(n)$
- $o\big(g(n)\big) = \{\, f(n) \mid \forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : \; 0 \leq \; f(n) < \; c \cdot g(n) \,\}$



Jetzt: Egal für welches c

$f(n) = o(g(n))$

Alternative Schreibweise
$f(n) \in o(g(n))$

- Es gilt: $lim_{n \to \infty} f(n) \,/\, g(n) \; = \; 0$
- Beispiele: $2n \; = \; o(n^2)$ $\quad 2n^2 \; \neq \; o(n^2)$ $\quad 2n^2 \; = \; o(n^3)$

Beispiel (fortgesetzt)

- $ZAHL\_VORHANDEN(x, daten)$

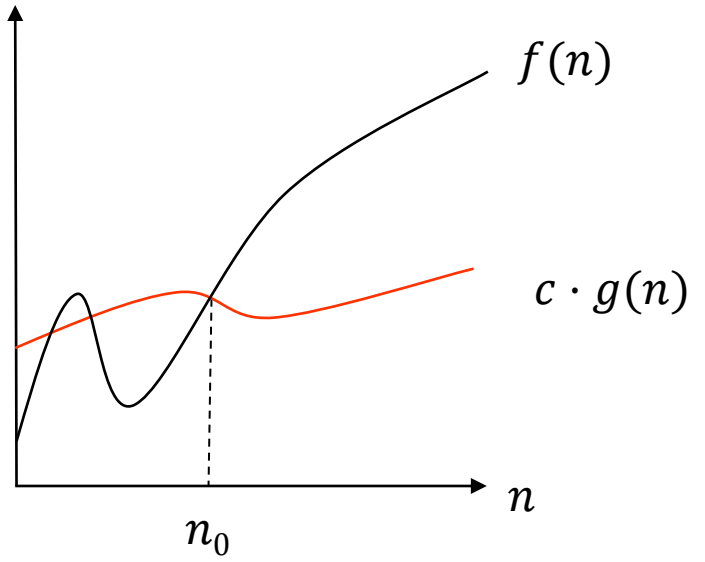
```
1   n = daten.länge
2   for i = 1 to n
3       if x == daten[i]
4           return true
5   return false
```

- Schritte, die der Algorithmus benötigt (maximal)
- Ergebnis: $f(n) = 1 + (n+1) + n + 1 = 2n + 3$

- In o-Notation ausgedrückt
  - $f(n) = o(n^2)$
- Jedoch **nicht**
  - $f(n) = o(n)$

Ω-Notation

- Asymptotisch untere Schranke
  - $f(n)$ wächst mindestens so schnell wie $g(n)$
- $\Omega(g(n)) = \{ f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq c \cdot g(n) \leq f(n) \}$



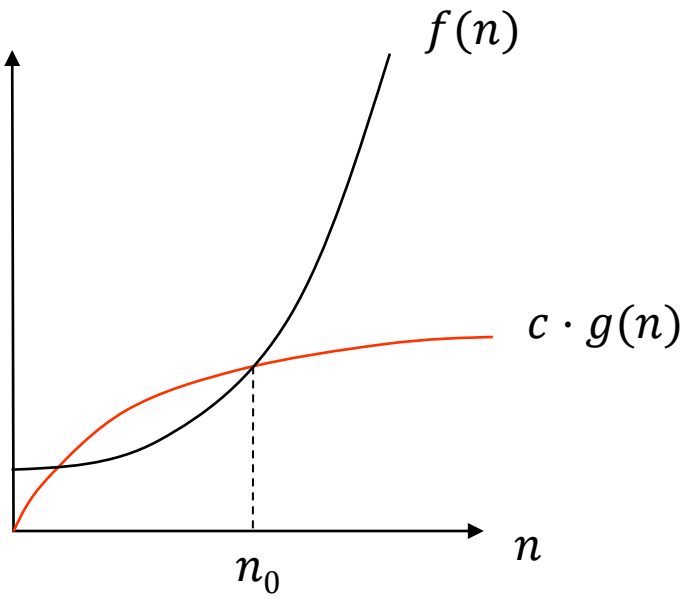
$f(n) = \Omega(g(n))$

Alternative Schreibweise
$f(n) \in \Omega(g(n))$

- Beispiel: $5 \cdot n^2 + 42 \cdot n + 2 = \Omega(n^2)$ $2^n + 5 \cdot n = \Omega(2^n)$

ω-Notation

- Untere Schranke
  - $f(n)$ wächst deutlich schneller als $g(n)$
- $\omega(g(n)) = \{ f(n) \mid \forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n) \}$



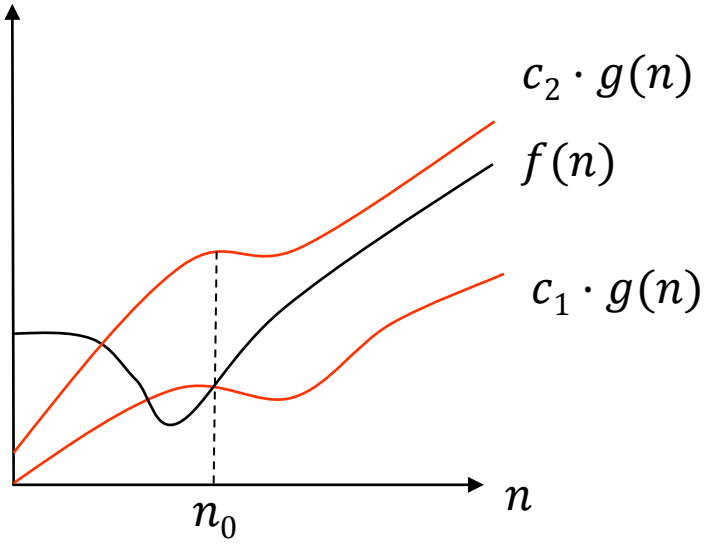
$f(n) = \omega(g(n))$

Alternative Schreibweise
$f(n) \in \omega(g(n))$

- Es gilt: $lim_{n \to \infty} f(n)/g(n) = \infty$
- Beispiele: $n^2/2 = \omega(n)$ $n^2/2 \neq \omega(n^2)$

Θ-Notation

- Asymptotisch gebunden
  - $f(n)$ wächst ebenso schnell wie $g(n)$
- $\Theta(g(n)) = \{ f(n) \mid \exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$



$f(n) = \Theta(g(n))$

Alternative Schreibweise
$f(n) \in \Theta(g(n))$

- Beispiel: $a \cdot n^2 + b \cdot n = \Theta(n^2)$ $\quad a \cdot n^2 \neq \Theta(n)$

Beispiel (fortgesetzt)

- $\mathrm{f(n)} = 1 + (n+1) + n + 1 = 2n + 3$

- Es gilt in Ω -Notation
  - $\mathrm{f(n)} = \Omega(\mathrm{n})$ oder auch $\mathrm{f(n)} = \Omega(1)$
- Es gilt in ω–Notation
  - $\mathrm{f(n)} = \omega(1)$ aber nicht $\mathrm{f(n)} = \omega(\mathrm{n})$
- Und in Θ-Notation
  - $f(n) = \Theta(n)$

Gebräuchliche Komplexitätsklassen

O-Klasse	Eigenschaft
$O(1)$	Höchstens konstanter Aufwand
$O(\log n)$	Höchstens logarithmischer Aufwand
$O(n)$	Höchstens linearer Aufwand
$O(n \log n)$	
$O(n^2)$	Höchstens quadratischer Aufwand
$O(n^k)$	Höchstens polynomialer Aufwand
$O(2^n)$	Höchstens exponentieller Aufwand

Illustration am Beispiel

- Aufgabe: Finde in einem Zahlenvektor der Größe n den Abschnitt, dessen Elemente addiert, die größte Summe ergeben
- Hierfür gibt es einen Algorithmus mit $O(n^3)$ und einen mit $O(n)$.

n	**Alpha 21164A (533 MHz), C-Programm, $O(n^3)$-Algorithmus**	**Radio Shack TRS-80 (2,03 MHz), BASIC-Programm, $O(n)$-Algorithmus**
10^1	0,58µs	195ms
10^2	0,58ms	1,95s
10^3	0,58s	19,5s
10^4	9min 40s	3min 15s
10^5	6d 16h 6min 40s	32min 30s
10^6	18y 142d 23h 6min 40s	5h 25min

Rechenregeln

- Es folgen u.a. diese Zusammenhänge
  - Sei $f(n) = O(r(n))$ und $g(n) = O(s(n))$.
    Dann gilt

$$f(n) + g(n) = O(r(n) + s(n))$$
$$f(n) \cdot g(n) = O(r(n) \cdot s(n))$$

$f(n) = \Theta(g(n))$ gdw. $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$
$f(n) = O(g(n))$ gdw. $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ gdw. $g(n) = \omega(f(n))$

- Reflexivität
  - $f(n) = \Theta(f(n)),\quad f(n) = O(f(n)),\quad f(n) = \Omega(f(n))$
- Symmetrie
  - $f(n) = \Theta(g(n))$ gdw. $g(n) = \Theta(f(n))$

Abhängigkeit von der Eingabe

- Komplexitätsbetrachtung ist immer abhängig von der Eingabe
  - Nicht nur Umfang (Zahl der Elemente n) entscheidend
  - Sondern auch Inhalt

- Beispiele Sortieren
  - Einige Sortieralgorithmen sind sehr schnell, wenn die Eingabe bereits (fast) sortiert ist
  - Nur wenige Operationen notwendig
  - Im Idealfall (= Best-case) z.B. Aufwand in $O(n)$
  - Im ungünstigsten Fall (= Worst-case) dagegen deutlich höher

- Üblicherweise wird mit dem Worst-case gerechnet
  - Praxisorientierter kann dagegen der Durchschnitt sein (= Average-case)

Beispiel (fortgesetzt)

- $ZAHL\_VORHANDEN(x, daten)$

```
1   n = daten.länge
2   for i = 1 to n
3       if x == daten[i]
4           return true
5   return false
```

- Schritte, die der Algorithmus benötigt
  - Worst-Case: $f(n) = 1 + (n+1) + n + 1 = 2n + 3 = \Theta(n)$
    - Z.B. x nicht in $daten$ enthalten
  - Best-Case: $f(n) = 1 + 1 + 1 + 1 = 4 = \Theta(1)$
    - Z.B. x an erster Stelle in $daten$
  - Average-Case
    - Schwer zu bestimmen ohne weitere Informationen: Erwartungswert dass x in $daten$, Verteilung der Werte innerhalb $daten$, …

Aufwand für ein Problem

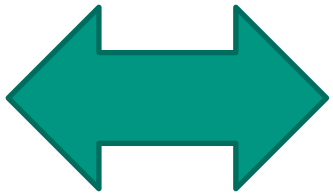
- Aufwand für ein Problem $f: A \rightarrow B$
  - Verschiedene Algorithmen zur Berechnung von f können bei gleicher Eingabe unterschiedlichen Aufwand haben
  - Bedeutsam ist daher auch, welchen Aufwand der „beste“ Algorithmus aufweist
- Daher zwei Aufgaben
  - Bestimme kleinsten möglichen Aufwand für das Problem (Komplexität des Problems)
  - Bestimme Algorithmus mit kleinstmöglichem Aufwand
    - Dauerbeschäftigung des Algorithmikers
    - Auch für uns in weiten Teilen der Vorlesung!
- Auf den tatsächlichen Aufwand haben auch Einfluss
  - Programmiersprache
  - Rechnerorganisation
  - Prozessorgeschwindigkeit

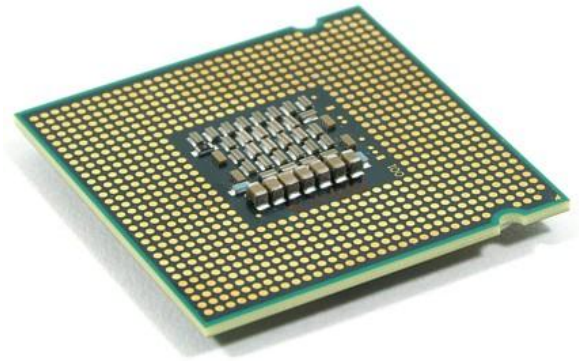
„Algorithm Engineering“

- Design von Algorithmen
  - Theoretische Analyse
  - Experimentelle Evaluierung
  - Einfluss realer Rechnerarchitekturen

- Motiviert aus klassischer Algorithmik
  - Beschränkt sich auf die Theorie
  - Jedoch: Große Lücke zwischen Theorie und Praxis
  - Gebiet „Algorithm Engineering“ soll die Lücke schließen

$O(n^3)$





Zusammenfassung

- Algorithmik beschränkt sich nicht auf Sortieren!
  - Aber: Sortieren ist anschauliches Beispiel
  - Daher: Kapitel 2 – Sortieren

- Die Vorlesung deckt die grundlegenden Gebiete ab
  - Algorithmen werden in einheitlicher Weise betrachtet und analysiert

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 2 – Sortieren

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

I. Einführung

1. Einführung

II. Suchen und Sortieren

2. Sortieren
   - 2.1 Motivation
   - 2.2 Bubblesort
   - 2.3 Insertionsort
   - 2.4 Selectionsort
   - 2.5 Untere Schranke
   - 2.6 Mergesort
   - 2.7 Quicksort
   - 2.8 Radixsort
   - 2.9 Zusammenfassung

III. Datenstrukturen

3. Folgen als Felder und Listen
4. Hashing
5. Heaps
6. Sortierte Listen / Bäume

IV. Graphenalgorithmen

7. Graphrepräsentation
8. Graphtraversierung
9. Kürzeste Wege
10. Minimale Spannbäume

V. Ausblick

11. Generische Optimierungsansätze
12. Zusammenfassung und Ausblick

2.1 Motivation

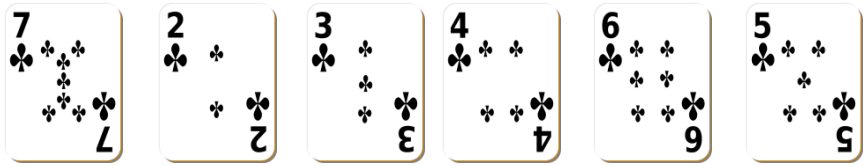
- Warum will man überhaupt Sortieren ?
  - In sortierten Mengen kann man besser suchen

- Beispiele aus der realen Welt
  - Bücherregale die alphabetisch sortiert sind
  - Spielkarten beim Skat spielen nach Wertigkeit sortiert
  - Kleiderschrank nach Farben sortiert
  - Telefonbuch nach Namen sortiert
- Beispiele aus der Informatik
  - Datenbankinhalte anhand von Sortierschlüsseln

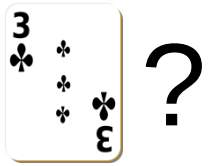
- Wie würden Sie intuitiv ein Bücherregal sortieren ?
  - Erste Entscheidung: Sortieren wir nach Titel oder nach Autorenname ?
  - Zweite Frage: Wie gehen wir vor ?
    - Intuitiv: Alle Autoren mit Anfangsbuchstaben „A“ suchen, Bücher einsortieren und anhand der darauffolgenden Buchstaben weitersortieren ?
    - Alternativen: Immer nebeneinanderstehende Bücher vergleichen und gegebenenfalls vertauschen ? Jeweils ein neues Buch aufnehmen und an die „richtige“ Stelle im Regal einsortieren ?
  - Wie würden Sie dieses Problem auf einem Computer lösen ?

Motivation

- Warum ist Sortieren für den Computer komplex ?
  - Kein Gesamtüberblick
    - Menschen können Gesamtsituation erfassen



  - Keine Intuition
    - Kein Wissen, ob ein zu sortierender Wert eher groß oder eher klein ist

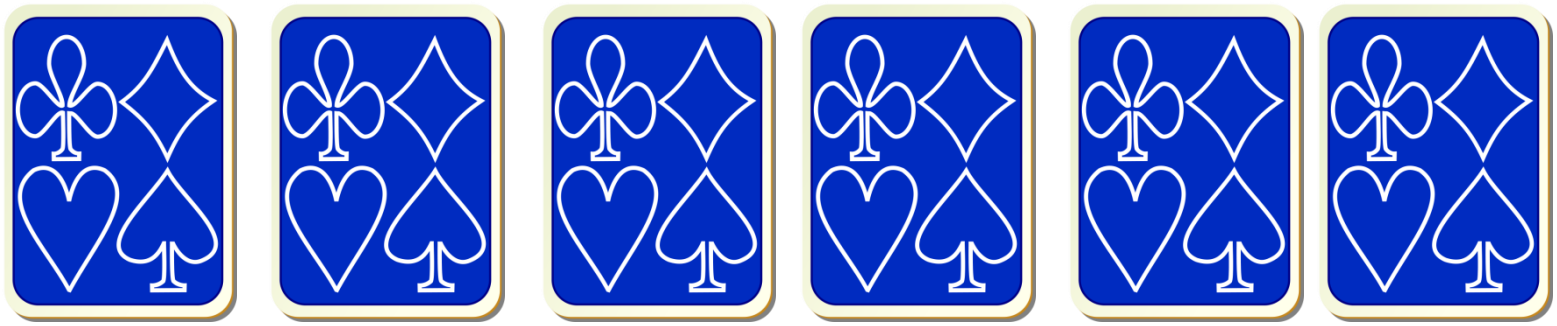

?

  - Kein „kostenloses“ Gedächtnis
    - Das Anlegen von Notizen kostet den Computer Zeit und Speicherplatz

Motivation

- Sortierprozess ist vergleichbar mit
  - Sortieren einer Reihe von Spielkarten
  - Die Spielkarten sind umgedreht
  - Es dürfen maximal zwei gleichzeitig aufgedeckt werden
  - Nach dem Aufdecken vergisst man, was man gesehen hat

- Ideen ?



2.1.2 Das Sortierproblem

- Allgemeine Definition des Sortierproblems
  - Eingabe: Folge von *n* Zahlen $< a_1, a_2, \ldots, a_n >$
  - Ausgabe: Permutation (Umordnung) $< a'_1, a'_2, \ldots, a'_n >$, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ gilt
  - Zu sortierende Zahlen sind sogenannte Schlüssel
- Wir behandeln verschiedene Algorithmen zur Lösung des Sortierproblems
  - Vergleichsbasierte Algorithmen
    - Bubblesort
    - Insertionsort
    - Selectionsort
    - Mergesort
    - Quicksort
  - Andere
    - Radixsort

Das Sortierproblem

- Wir betrachten als wichtigsten Parameter den *Zeitaufwand* der Algorithmen, also
  - *Aufwand* $T(n)$: Anzahl der *Zeiteinheiten*, die der Algorithmus für ein Problem mit Umfang n benötig
- Wir betrachten für die Algorithmen jeweils
  - Aufwand im schlechtesten Fall (Worst-case)
  - Mittlerer Aufwand (Average-case)
  - Aufwand im besten Fall (Best-case)
- Allgemeine Vorgehensweise
  - Betrachte die notwendige Anzahl an *Vergleichen* und *Vertauschungen* des Algorithmus, um auf den Aufwand schließen zu können

Das Sortierproblem

- Weitere Eigenschaften von Sortieralgorithmen
  - *„in-place“* Sortieren
    - Der Algorithmus benötigt zum Sortieren keinen zusätzlichen Speicherplatz
    - Beispiel: *Mergesort* sortiert nicht in-place, benötigt also je nach Implementierung noch $O(n)$ zusätzlichen Speicher
  - *„stabiles“* Sortieren
    - Der Algorithmus behält die Reihenfolge der Sortierschlüssel bei, wenn diese gleich sind
    - Beispiel: *Insertionsort*

- Wir betrachten nur den Aufwand an Rechenzeit, bei sehr großen Datenmengen ist aber auch der Speicheraufwand wichtig
  - Mehr zum Thema *Sortieren und Speicheraufwand* im Kapitel 5, Heaps

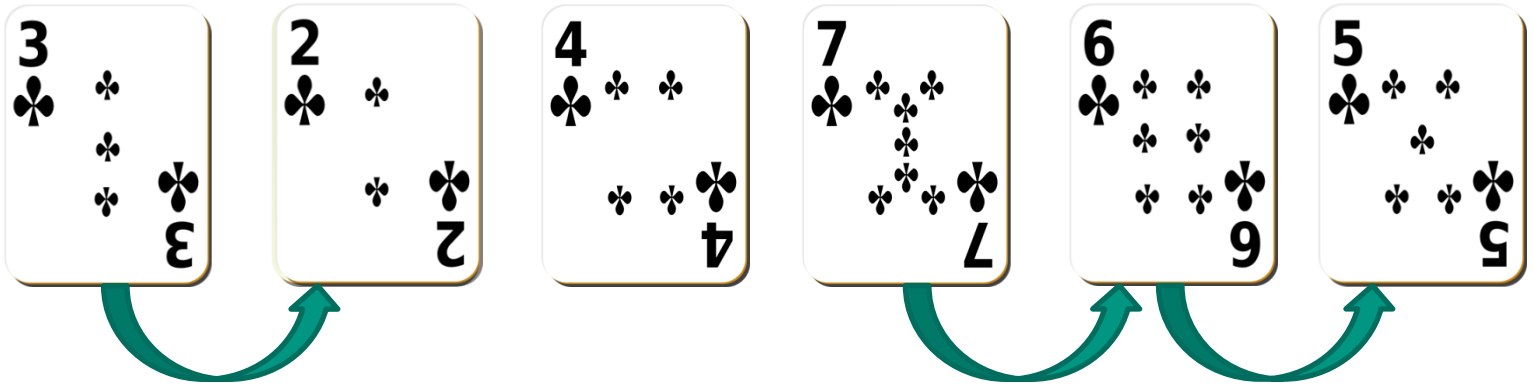
2.2 Bubblesort

- „Sortieren durch Aufsteigen“

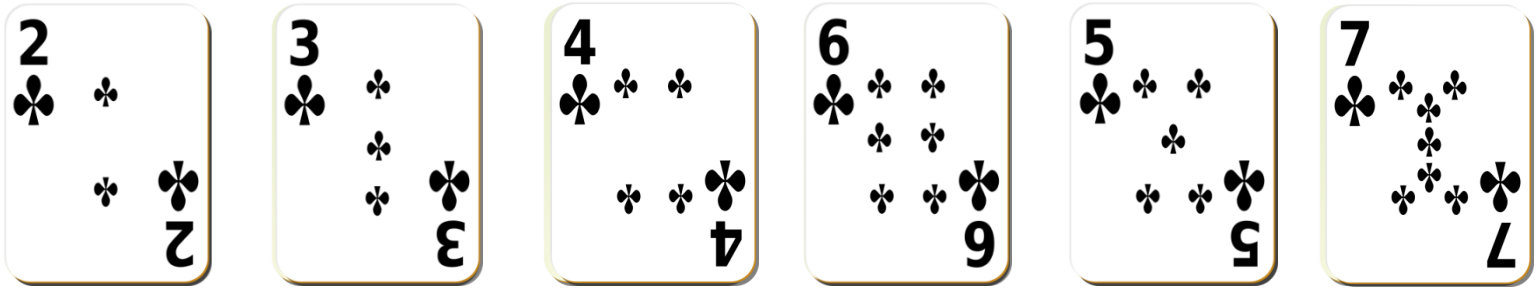
- Grundlegender Ablauf
  - Vergleiche immer zwei nebeneinander liegende Karten
    - Ist die rechte kleiner als die linke, vertausche sie
    - Ansonsten fahre mit dem nächsten Paar fort
  - Nach dem letzten Paar beginne von vorne
  - Sind bei einem kompletten Durchlauf keine Vertauschungen mehr aufgetreten, ist die Reihe korrekt sortiert
  - Sortierverfahren vergleichbar mit aufsteigenden Luftblasen in Wasser
    - Große Elemente „steigen“ schnell auf
    - Daher der Name *Bubblesort*

2.2.1 Bubblesort – Beispiel

- 1. Durchlauf

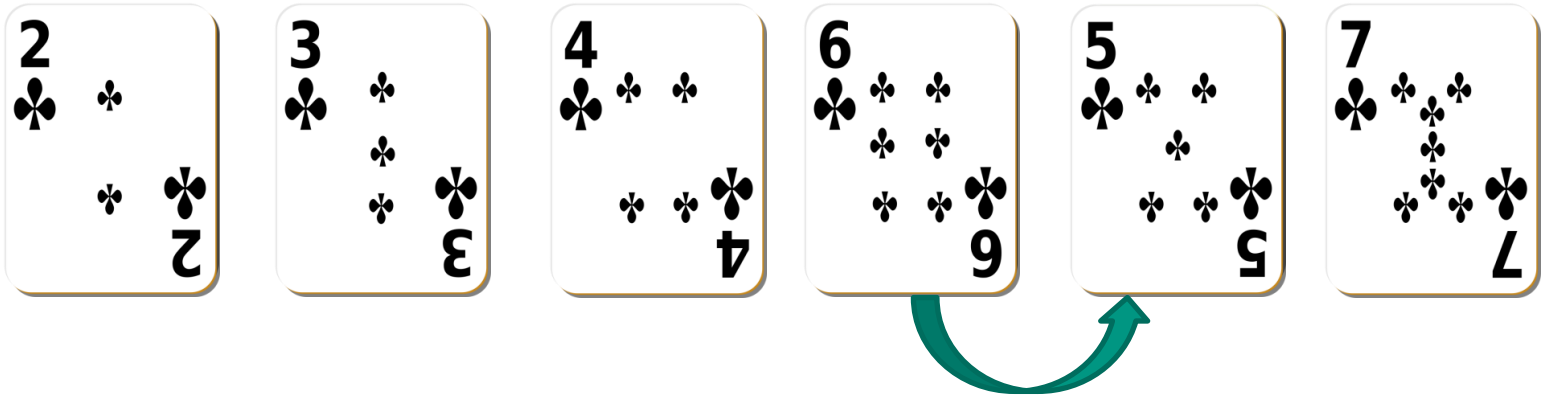


- Nach dem 1. Durchlauf

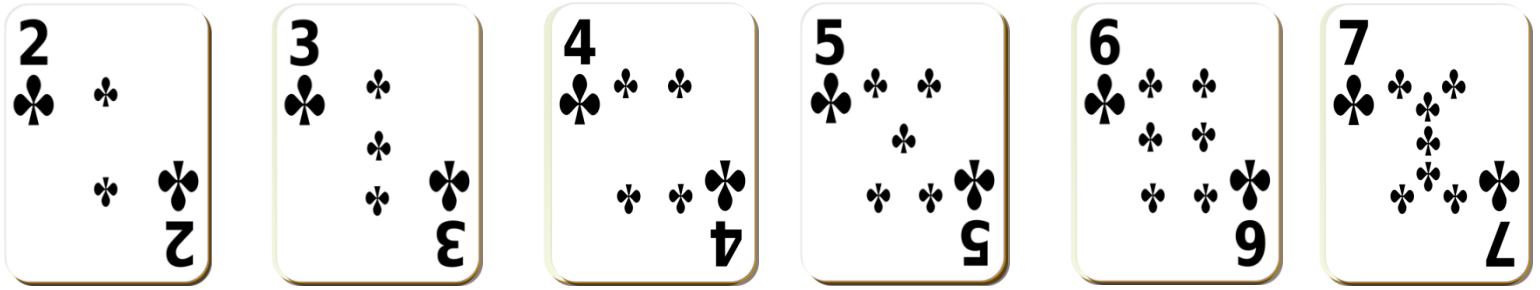


Bubblesort – Beispiel

- 2. Durchlauf

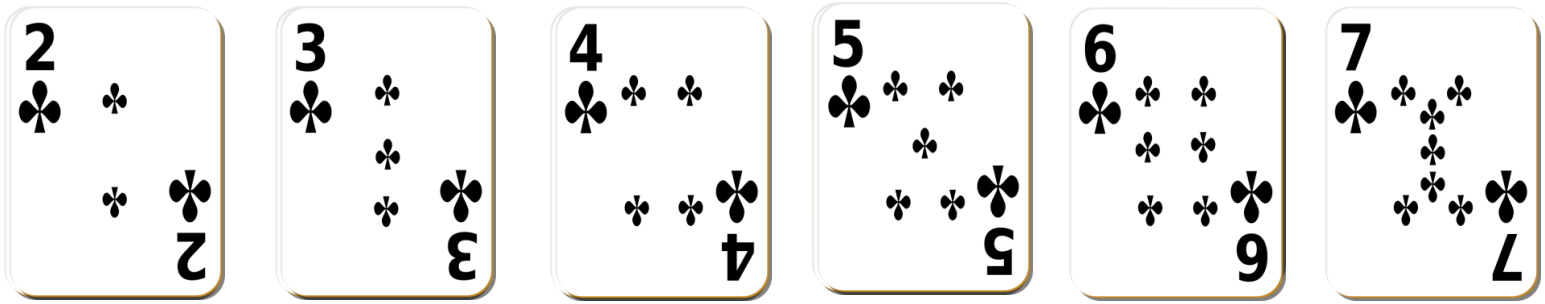


- Nach dem 2. Durchlauf



Bubblesort – Beispiel

- 3. Durchlauf



- *Bubblesort* endet nach 3 Durchläufen

2.2.2 Bubblesort - Pseudocode

- $BUBBLESORT(A)$

```
1 m = A.länge − 1
2 repeat
3         sorted = TRUE
4         for i = 1 to m
5                 if A[i] > A[i + 1]
6                         sorted = FALSE
7                         vertausche A[i], A[i + 1]
8 until sorted
```

Abbruchbedingung

Vergleiche benachbarte Elemente

Vertausche wenn notwendig

Maximal m Durchläufe um alle Elemente zu sortieren

2.2.3 Bubblesort – Komplexitätsanalyse

- Eingabe: Eine Liste A der Länge n

- Best Case – Eine bereits sortierte Liste der Länge n
  - *Bubblesort* braucht genau einen Durchlauf
    - Es werden n-1 Vergleiche durchgeführt
    - Es treten keine Vertauschungen auf
  - Komplexität: $O(n)$

Komplexitätsanalyse

- Eingabe: Eine Liste A der Länge n

- Worst Case – Die Liste ist umgekehrt sortiert
  - Ablauf von *Bubblesort ?*
  - Beobachtung
    - Im ersten Durchlauf wird das größte Element an das Ende der Liste sortiert
    - Im zweiten Durchlauf wird das zweitgrößte Element an die zweitletzte Position der Liste sortiert
    - …
  - Allgemein: In Durchlauf k wird das erste Element an die $n - k + 1$-te Stelle mittels $n - k$ Vertauschungen und Vergleichen sortiert
    - Insbesondere muss nicht in jedem Schritt die ganze Liste durchlaufen werden

Bubblesort optimiert - Pseudocode

- $BUBBLESORTOPT(A)$

```
1 m = A.länge − 1
2 repeat
3         swapped = FALSE
4         for i = 1 to m
5                 if A[i] > A[i + 1]
6                         swapped = TRUE
7                         vertausche A[i], A[i + 1]
8         m = m − 1
9 until (swapped == FALSE oder m < 1)
```

Abbruchbedingung

Vergleiche benachbarte Elemente

Vertausche wenn notwendig

Maximal m Durchläufe um alle Elemente zu sortieren

Bubblesort – Komplexitätsanalyse II

- Eingabe: Eine Liste A der Länge n

- Worst Case – Die Liste ist umgekehrt sortiert
  - Es werden *n-1* Durchläufe benötigt
    - In jedem Durchlauf k $n-k$ Vertauschungen
    - In jedem Durchlauf k $n-k$ Vergleiche

  - Anzahl der Vergleiche oder Vertauschungen lässt sich jeweils berechnen als

$$\sum_{k=1}^{n-1} n-k = n * (n-1) - \sum_{k=1}^{n-1} k = (n^2 - n) - \frac{n^2-n}{2} = \frac{1}{2}(\boldsymbol{n^2 - n})$$

  - Komplexität im Worst-Case für *Bubblesort* daher $O(n^2)$

Bubblesort – Komplexitätsanalyse III

- Average Case – Eine zufällig sortierte Liste
  - Ablauf von *Bubblesort ?*
    - Wie viele Vertauschungen und wie viele Vergleiche werden durchgeführt ?
  - Für die Anzahl der Operationen gilt
    - Betrachtet man ein Element a an der Stelle k, so gilt
      - a_k gehört an die Stelle i in der Liste
      - a_k muss alle kleineren Elemente die weiter rechts stehen überholen
      - a_k muss alle größeren Elemente, die weiter links stehen überholen lassen
    - Die Anzahl der Vertauschungen und Vergleiche die notwendig sind, um a_k an die Stelle *i* zu sortieren ist somit

$$\frac{1}{n}\sum_{k=1}^{n} Anzahl\ Elemente\ links + Anzahl\ Elemente\ rechts =$$

$$\frac{1}{n}\sum_{k=1}^{n} \frac{n-i}{n-1} * (k-1) + \frac{i-1}{n-1} * (n-k) = \frac{n-1}{2}$$

Bubblesort – Komplexitätsanalyse IV

- Average Case – Eine zufällig sortierte Liste
  - Bei *n* Elementen ergibt sich also als Gesamtanzahl der Vertauschungen als

$$\frac{1}{2} * \frac{n * (n-1)}{2} = O(n^2)$$

  - Komplexität im Average-Case für *Bubblesort* daher $O(n^2)$

2.2.4 Bubblesort – Beobachtungen

- Große Elemente wandern schnell nach rechts
  - Im Jargon auch Rabbits genannt
- Kleine Elemente wandern langsam nach links
  - Im Jargon: Turtles

- Für größtenteils sortierte Mengen kann der Algorithmus schnell sein
- Eine bereits sortierte Menge ist in einem Durchlauf, $O(n)$, sortiert
- Jedoch: Im Allgemeinen ist *Bubblesort* sehr langsam, $O(n^2)$

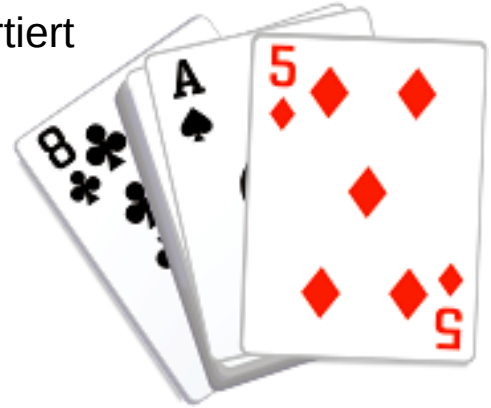
„In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact, that it leads to some interesting theoretical problems.“

- The Art of Computer Programming Vol. 3, Donald Knuth

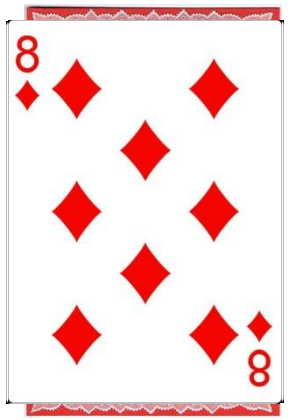
2.3 Insertionsort

- „Sortieren durch Einfügen“

- Grundlegender Ablauf
  - Analog zum Einsortieren von Spielkarten in einer Hand
    - Anfangen mit leerer Hand
    - Schrittweises Aufnehmen und Einfügen der nächsten Karte
    - Korrekte Einfügeposition wird durch Vergleichen der bereits aufgenommenen Karten von rechts nach links ermittelt
    - Karten auf der Hand sind zu jedem Zeitpunkt sortiert



2.3.1 Beispiel: Sortieren von Spielkarten



2.3.2 Insertionsort – Pseudocode

- $INSERTIONSORT(A)$

```
1 for j = 2 to A.länge
2     schlüssel = A[j]
3     // Füge A[j] in die sortierte Sequenz A[1..j − 1] ein.
4     i = j – 1
5     while i > 0 und A[i] > schlüssel
6           A[i + 1] = A[i]
7           i = i – 1
8     A[i + 1] = schlüssel
```

Aktuell zu sortierender Schlüssel

Suche korrekte Einfügeposition

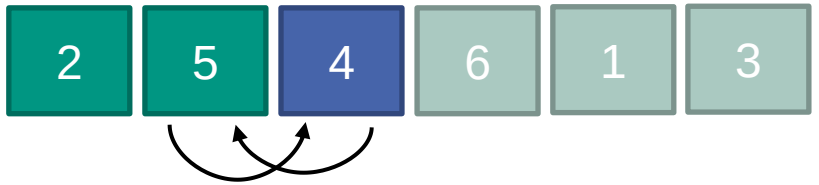
Verschiebe größere Schlüssel nach hinten

Füge Schlüssel an korrekter Position ein

2.3.3 Insertionsort – Beispiel

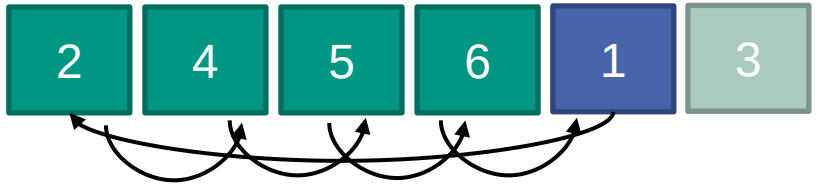
j=2

j=3

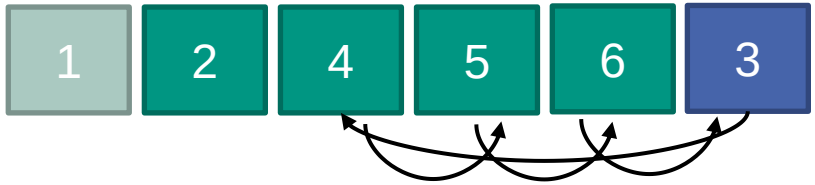


j=4

j=5



j=6



Ausgabe

2.3.4 Insertionsort – Komplexitätsanalyse

- $INSERTIONSORT(A)$

		Anzahl der Ausführungen
1	**for** $j = 2$ **to** $A.länge$	n
2	$schlüssel = A[j]$	$n-1$
3	// Füge $A[j]$ in die sortierte Sequenz $A[1..j-1]$ ein.	
4	$i = j - 1$	$n-1$
5	**while** $i > 0$ und $A[i] > schlüssel$	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] = schlüssel$	$n-1$

Insertionsort – Komplexitätsanalyse II

- Summe der einzelnen Ausführungen

$$T(n) = n + 3 \cdot (n-1) + \sum_{j=2}^{n} t_j + 2 \cdot \sum_{j=2}^{n} (t_j - 1)$$

- Im Worst-Case liegt das Feld zunächst in absteigend sortierter Reihenfolge vor
  - Dann gilt $t_j = j$ für $j = 2, 3, \ldots, n$
  - Wegen $\sum_{j=2}^{n} j = \frac{n(n-1)}{2} - 1$ und $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ ergibt sich

$$T(n) = n + 3 \cdot (n-1) + \frac{n(n-1)}{2} - 1 + 2 \cdot \frac{n(n-1)}{2}$$

Vereinfachung

$$T(n) = an^2 + bn + c$$

- Für den Aufwand im Worst-Case ergibt sich also

$$T(n) = \theta(n^2)$$

Insertionsort – Komplexitätsanalyse (III)

- Im Best-Case liegt das Feld zunächst in sortierter Reihenfolge vor:
  - Dann gilt $t_j = 1$ für $j = 2, 3, \ldots, n$
  - Wegen $\sum_{j=2}^{n} 1 = n - 1$ und $\sum_{j=2}^{n} 0 = 0$ ergibt sich

$$T(n) = n + 3 \cdot (n - 1) + n - 1 + 2 \cdot 0$$

$$T(n) = 5n - 4$$

- Für den Aufwand im Best-Case ergibt sich also

$$T(n) = \theta(n)$$

Insertionsort – Komplexitätsanalyse IV

- Im Average-Case liegt das Feld zunächst in zufälliger Reihenfolge vor
  - Es gilt dann, dass $t_j \approx \frac{j}{2}$ ist
    - Für ein Element $A[j]$ ist im Mittel die Hälfte der Elemente im bereits sortierten Feld $A[1..j-1]$ kleiner, die andere Hälfte größer als $A[j]$
    - Daraus folgt, es werden im Mittel die Hälfte des Teilfeldes überprüft und dann einsortiert
  - Für T(n) gilt daher wie im schlechtesten Fall

$$T(n) = an^2 + bn + c$$

- Für den Aufwand im Average-Case ergibt sich also

$$T(n) = \theta(n^2)$$

2.4 Selectionsort

- “Sortieren durch Auswählen”

- Grundlegender Ablauf
  - Suche das Minimum in Folge $<a_1, a_2, \ldots, a_n>$, also das Element mit dem kleinsten Sortierschlüssel
  - Vertausche dieses Minimum mit dem ersten, unsortierten Element der Folge
  - Man erhält im linken Teil der Folge eine sortierte Teilfolge der Länge 1 und rechts eine unsortierte Teilfolge der Länge $n - 1$.
  - Wiederhole Algorithmus für die unsortierte Teilfolge bis die sortierte Teilfolge die Länge n hat

2.4.1 Selectionsort – Beispiel

Eingabe

Suche Minimum

Vertausche

Suche Minimum

Suche Minimum

Vertausche

Suche Minimum

Vertausche

Ausgabe

2.4.2 Selectionsort – Pseudocode

```
SELECTIONSORT(A)
1     n = A.länge
2     links = 1
3     do
4           min = links
5           for i = links + 1 to n
6                 if A[i] < A[min]
7                       min = i
8           temp = A[min]
9           A[min] = A[links]
10          A[links] = temp
11          links = links + 1
10    while links<n
```

Initialisierung des Algorithmus

Starte Minimussuche

Bestimme neues Minimum durch Vergleich mit aktuellem Minimum

Vertausche Minimum mit erstem Element

Wiederhole dies für alle Elemente

2.4.3 Selectionsort – Komplexitätsanalyse

- *Selectionsort* zerlegt sich in folgende Teilschritte
  - Minimum bestimmen und zuweisen
  - Elemente vergleichen
  - Elemente vertauschen
  - Es gilt für die Anzahl der Vertauschungen und Vergleiche
    - Im ersten Durchgang gibt es eine Vertauschung und $n-1$ Vergleiche, im Zweiten eine Vertauschung und $n-2$ Vergleiche, ….
    - Anzahl setzt sich daher zusammen aus $n-1$ Vertauschungen und

$$(n-1)+(n-2)+\cdots+3+2+1 = \frac{n*(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$ Vergleichen

- Für $T(n)$ gilt also $T(n) = (n-1) + \frac{n^2}{2} - \frac{n}{2}$
  - Unabhängig von den Eingabedaten

- Daraus folgt $T(n) = O(n^2)$ für Best-Case, Average-Case und Worst-Case

2.5 Einschub: Untere Schranke für das Sortierproblem

- Bisher betrachteten Sortieralgorithmen waren relativ schlecht
  - $O(n^2)$ für das Sortieren von großen Datenmengen ungenügend
- Geht es schneller ?
  - … ja!
- Wie schnell geht es ?
  - … ?
- Gibt es eine untere Schranke für das Sortierproblem ?

Einschub: Untere Schranke für das Sortierproblem

- Für jeden vergleichenden Sortieralgorithmus sind im schlechtesten Fall $\Omega(n * \lg n)$ Vergleichsoperationen erforderlich

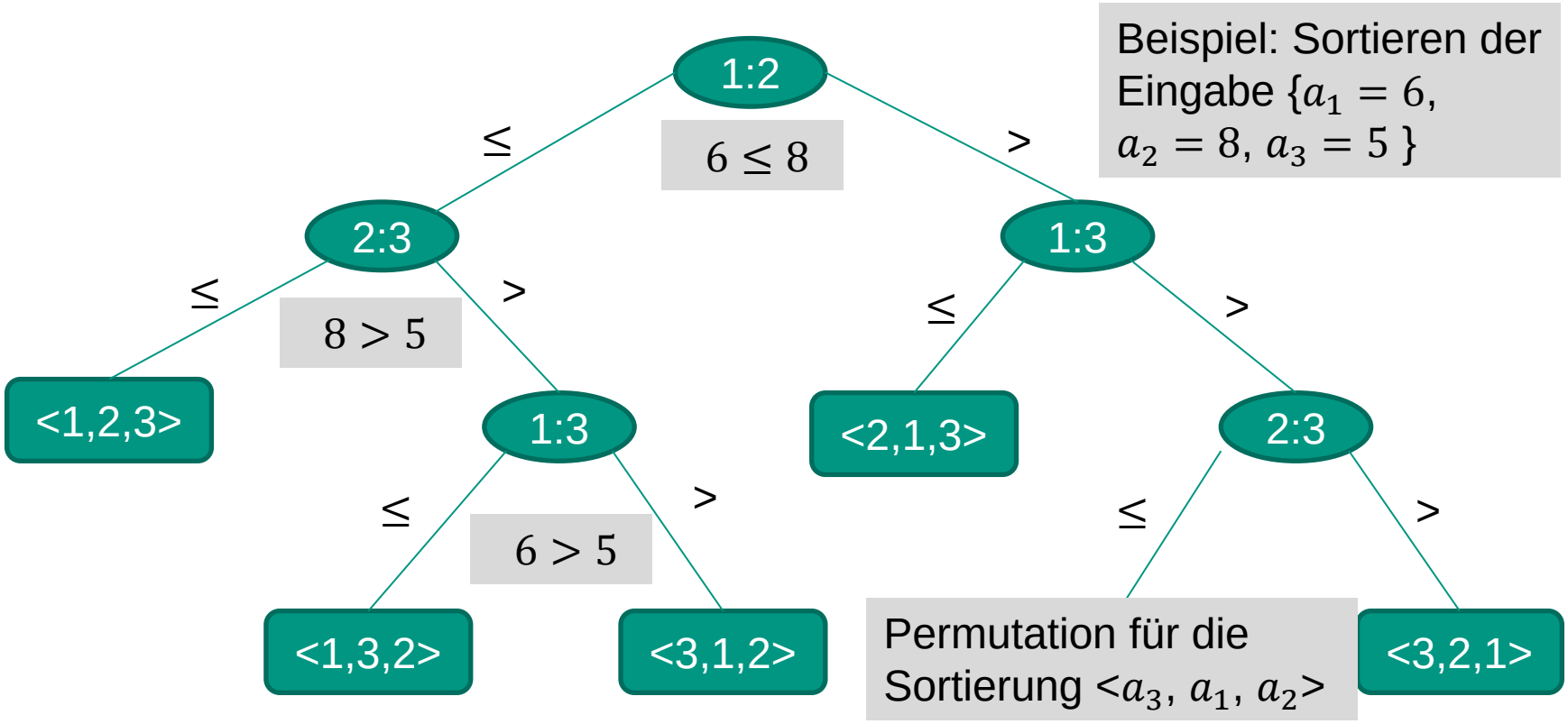
- Beweisidee
  - Modelliere Sortierverfahren als Entscheidungsbaum
    - Entscheidungsbäume sind binäre Bäume, die die Vergleiche in einem Sortierverfahren modellieren
    - Eingabe ist eine zu sortierende Menge mit n Elementen

Einschub: Untere Schranke für das Sortierproblem

- In einem Entscheidungsbaum gilt
  - Jeder innere Knoten wird mit $i:j$ für je ein *i* und *j* aus dem Bereich $1 \leq i, j \leq n$ annotiert
  - Jeder Blattknoten wird mit einer Permutation $< \pi(1), \pi(2), \ldots, \pi(n) >$ annotiert
- Die Ausführung eines Sortieralgorithmus entspricht der Verfolgung eines Pfades von der Wurzel zu einem Blatt des Baumes
  - Ein mit *i:j* annotierter Knoten bedeutet einen Vergleich zwischen a_i und a_j
    - Der linke Teilbaum beschreibt die nachfolgenden Vergleiche falls $a_i \leq a_j$, der Rechte gilt für die nachfolgenden Vergleiche falls $a_i > a_j$
  - Erreichen wir ein Blatt, hat der Sortieralgorithmus die Reichenfolge $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ für eine sortierte Ausgabe festgelegt

Einschub: Untere Schranke für das Sortierproblem

- Beispiel eines Entscheidungsbaums für das Sortieren durch Einfügen für eine Menge mit drei Elementen



Einschub: Untere Schranke für das Sortierproblem

- Jeder korrekte Sortieralgorithmus muss jede der *n!* möglichen Permutationen der *n* Eingabewerte akzeptieren und korrekt sortieren
  - Also müssen auch *n!* Blattknoten im Entscheidungsbaum existieren
  - Alle diese Blätter müssen außerdem erreichbar sein
  - Die Höhe des Entscheidungsbaums ist die Anzahl der notwendigen Vergleiche

- Für einen Entscheidungsbaum der Höhe *h* mit *l* erreichbaren Blättern gilt
  - Ein binärer Baum der Höhe *h* kann nicht mehr als 2^h Blätter haben
  - Jede der $n!$ Permutationen muss erreichbar sein

$$\Rightarrow n! \leq l \leq 2^h$$

  - Durch Anwendung des Logarithmus erhält man

$$h \geq \lg(n!) = \Omega(n * \lg n)$$

2.6 Mergesort

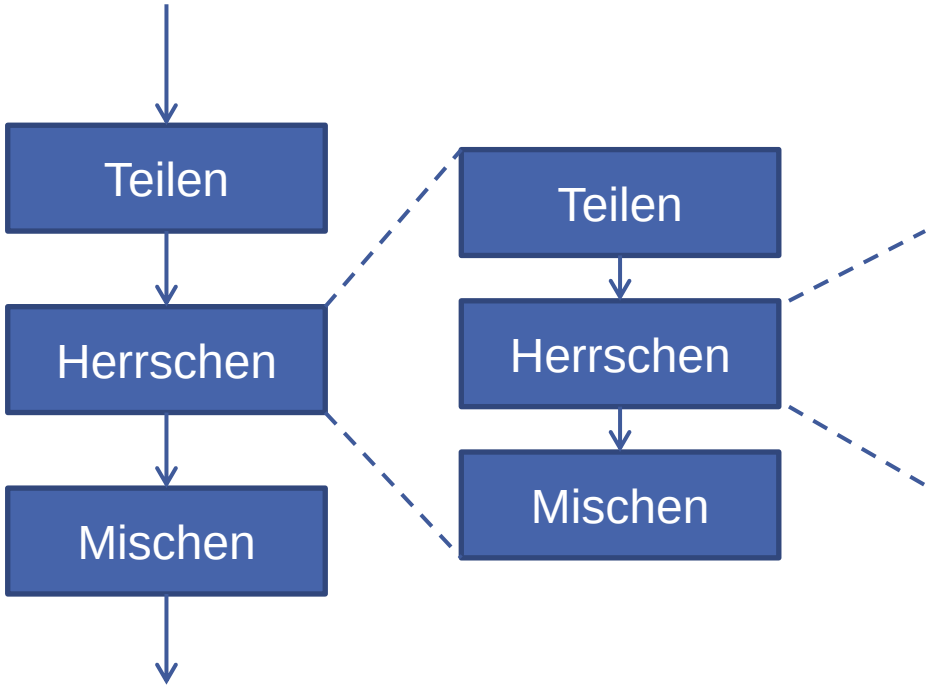
- „Sortieren durch Mischen“
  - Analog zum mischen eines Stapels von Karten
    - Besteht der Stapel nur aus einer Karte, sind wir fertig
    - Ansonsten teile den Stapel in zwei Hälften und sortiere diese rekursiv
    - Sobald die Teile sortiert sind, mische die beiden Stapel gleichzeitig von oben nach unten nach dem Reißverschlussprinzip

- Folgt dem „Teile-und-Herrsche“ Paradigma
  - Problem wird in Teilprobleme zerlegt
  - Ggf. rekursiv weiter teilen
  - Wenn das Teilproblem klein genug ist, auf direktem Weg lösen
  - Lösungen der Teilprobleme zur Gesamtlösung vereinigen

- Vorteile
  - Verarbeitung lässt sich parallelisieren

2.6.1 Mergesort – Prinzip

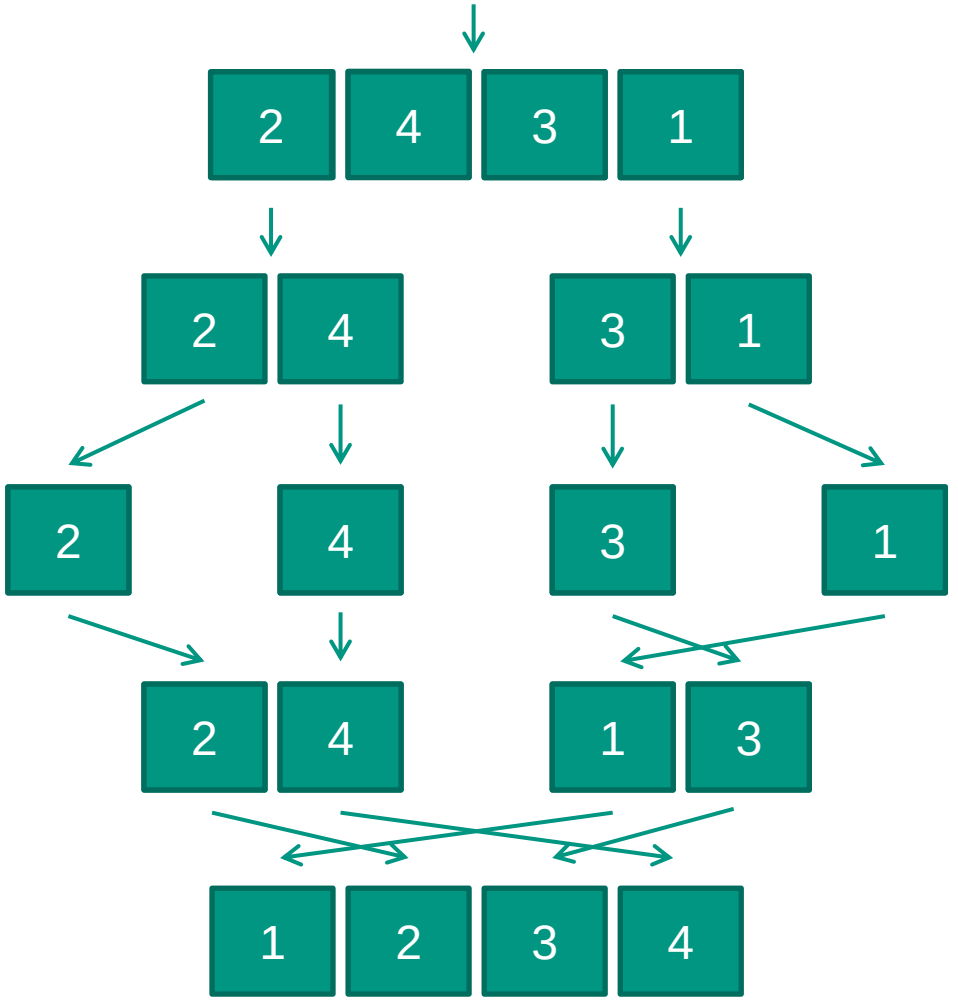
- Vorgehen
  - Teile Folge A in zwei Teilfolgen A_1, A_2 mit je $n/2$ Elementen auf
  - Sortiere die zwei Teilfolgen A_1, A_2 rekursiv mithilfe von *Mergesort*
  - Mische die zwei sortierten Teilfolgen A'_1, A'_2 um die sortierte Lösung zu erhalten



- Rekursion bricht bei Teilfolgelänge 1 ab
  - Folge der Länge 1 ist bereits sortiert!
- Eigentliche Arbeit wird im dritten Teilschritt (Mischen) durchgeführt

2.6.2 Mergesort – Beispiel

1. Eingabe
2. Teilen
3. Rekursiv weiter teilen
4. Mischen
5. Mischen



2.6.3 Mergesort – Pseudeocode

- Sortieren des Feldes A der Größe l
  - Algorithmus in Methoden $MERGESORT$ und $MERGE$ aufgeteilt
  - $MERGESORT(A, p, r)$ sortiert alle Elemente im Feld $A[p..r]$
    - Erster Aufruf mit $MERGESORT(A, 1, l)$
  - $MERGE(A, p, q, r)$ mischt die beiden sortierten Teilfelder $A[p..q]$ und $A[q+1..r]$

- $MERGESORT(A, p = 1, r = l)$

```
1 if p < r
2     q = ⌊(p + r)/2⌋
3     MERGESORT(A, p, q)
4     MERGESORT(A, q + 1, r)
5     MERGE(A, p, q, r)
```

Mitte des Feldes q bestimmen

$MERGESORT$ rekursiv für beide Teilfelder aufrufen

$MERGE$ führt Teilfelder zusammen

Mergesort – Pseudeocode II

- $MERGE(A, p, q, r)$

```
1     n_1 = q − p + 1
2     n_2 = r − q
3     // seien L[1..n_1 + 1], R[1..n_2 + 1] zwei neue Felder
4     for i = 1 to n_1
5           L[i] = A[p + i − 1]
6     for j = 1 to n2
7           R[j] = A[q + j]
8     L[n_1 + 1] = R[n_2 + 1] = ∞
9     i = j = 1
10    for k = p to r
11          if L[i] <= R[j]
12                A[k] = L[i]
13                i = i + 1
14          else
15                A[k] = R[j]
16                j = j + 1
```

n_1, n_2 Größe der Teilfelder

Teilfelder L, R mit Daten aus Feld A füllen

Der Reihe nach die Elemente der Teilfelder L, R miteinander vergleichen

Jeweils das kleinere in das Feld A einfügen

2.6.4 Mergesort – Komplexitätsanalyse

- Der Aufwand von Teile-und-Herrsche-Algorithmen lässt sich als Rekursionsgleichung darstellen

- Es gilt für *Mergesort* im Worst-Case
  - Sortieren eines Elementes liegt in $\theta(1)$
  - Sortieren mehrerer Elemente zerlegt sich in die Teilschritte
    - Teilen – Berechnen der Mitte des Feldes, $\theta(1)$
    - Herrschen – Rekursiver Aufruf der Algorithmus mit 2 Problemen der Größe $\frac{n}{2}$
    - Mischen – Aufwand ist linear $\theta(n)$

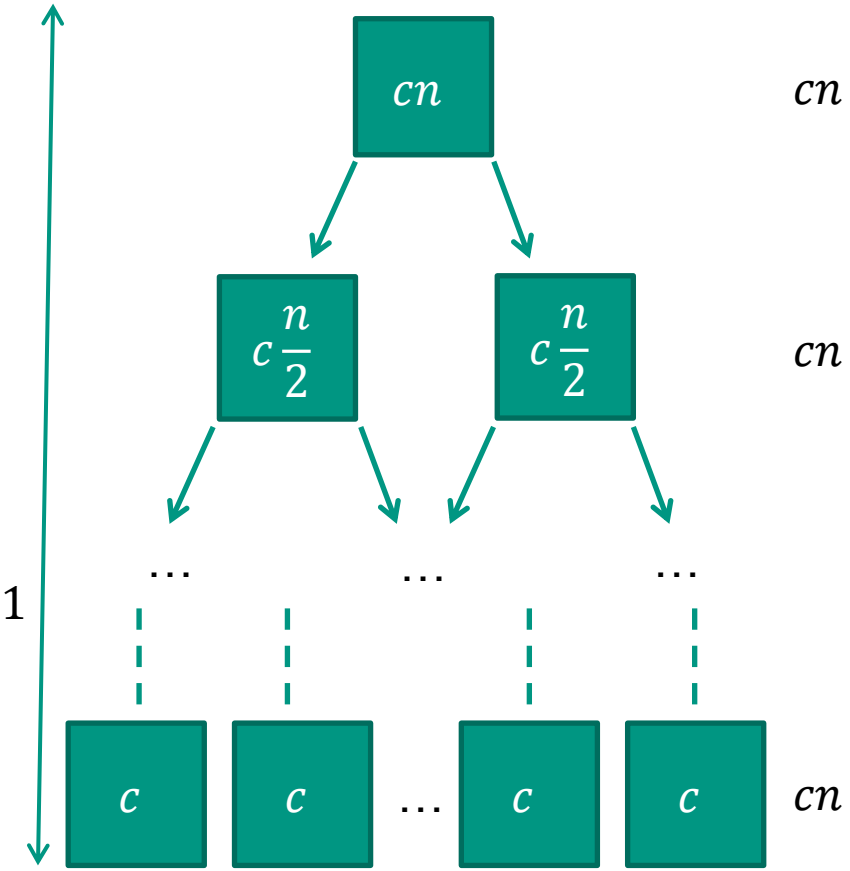
$$T(n) = \begin{cases} \theta(1) & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & n > 1 \end{cases}$$

  - OBdA gilt Annahme $n = 2^x, x \in \mathbb{N}$

Mergesort – Komplexitätsanalyse II

- Lösung der Rekursionsgleichung mit dem Mastertheorem ergibt
  - $T(n) = \theta(n \, lg \, n)$

- Lässt sich auch intuitiv herleiten
  - Mit c als Zeitkonstante für Trivialfall gilt
  - $T(n) = \begin{cases} c \\ 2T\left(\frac{n}{2}\right) + cn \end{cases}$ falls $\begin{matrix} n = 1 \\ n > 1 \end{matrix}$
  - Kosten im Rekursionsbaum auf Ebene i betragen $2^i c\left(\frac{n}{2^i}\right) = cn$
  - Gesamtzahl der Ebenen somit $lg \, n + 1$
  - Gesamtkosten daher $\mathrm{cn}(lg \, n + 1)$ $= cn \, lg \, n + cn \in \theta(n * lg \, n)$



Exkurs: Rekursionsgleichungen

- Eine Rekursionsgleichung ist eine Gleichung, die ihre Funktion durch ihre eigenen Funktionswerte für kleinere Eingaben beschreibt
  - Beispiel *Mergesort*

$$T(n) = \begin{cases} \theta(1) & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & n > 1 \end{cases}$$

  - Es existieren verschiedene Methoden um Rekursionsgleichungen zu lösen, um asymptotische Lösungen zu erhalten
    - **Subsitutionsmethode**: Erraten einer Lösung, Beweis über vollständige Induktion
    - **Rekursionsbaum-Methode:** Umwandlung der Rekursion in einen Rekursionsbaum, dessen Knoten die Kosten pro Rekursionsschritt angeben. Lösen der Rekursionsgleichung mit Hilfe von Techniken zur Beschränkung von Summenformeln
    - **Mastermethode:** Liefert Lösungen für Rekursionsgleichungen der Form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, für $a \geq 1$, $b > 1$ und $f(n)$ ist gegebene Funktion

Exkurs: Rekursionsgleichungen

- **Mastertheorem:** Seien $a \geq 1,\ b > 1$ Konstanten, $f(n)$ eine Funktion und sei *T(n)* durch die Rekursionsgleichung

$$T(n) = \mathrm{a}T\left(\frac{n}{b}\right) + f(n)$$

ausgedrückt, so besitzt *T(n)* die asymptotischen Schranken

1. $T(n) = \theta\left(n^{log_b a}\right)$, falls $\mathrm{f(n)} = \mathrm{O}(n^{log_b a - \epsilon})$
2. $T(n) = \theta\left(n^{log_b a} * \lg n\right)$, falls $\mathrm{f(n)} = \theta(n^{log_b a})$
3. $T(n) = \theta(f(n))$, falls $\mathrm{f(n)} = \Omega(n^{log_b a + \epsilon})$ für $\epsilon > 0$ und $a * f\left(\frac{n}{b}\right) \leq c * f(n)$ mit $c < 1$ und hinreichend großem n

Mergesort – Komplexitätsanalyse II

- Lösung der Rekursionsgleichung mit dem Mastertheorem
  - $T(n) = \begin{cases} \theta(1) & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & n > 1 \end{cases}$ gilt für *Mergesort*

- Anwendung des Mastertheorems
  - Rekursionsgleichung ist in der Form $T(n) = \mathrm{a}T\left(\frac{n}{b}\right) + f(n)$
  - $a = 2\,, b = 2$ und $f(n) = \theta(n)$
  - $n^{log_b a} = n^{log_2 2} = \mathrm{n}$
  - Es lässt sich also Fall 2 von voriger Folie anwenden und es gilt

$$T(n) = \theta(n * lg\, n)$$

Mergesort – Komplexitätsanalyse III

- Aufwand im Average-Case oder Best-Case
  - Bleibt bei $\theta(n * lg\, n)$!
- Begründung Best-Case
  - Bester Fall = Folge bereits sortiert
  - Aufteilen und Mischen der Folgen trotzdem mit gleichem Aufwand verbunden
  - Keine Veränderung der Komplexitätsklasse
  - Jedoch: Vorherige Prüfung ob Folge sortiert ist möglich (in $O(n)$)
    - Erweiterung des Algorithmus!
- Begründung Average-Case
  - Best- und Worst-Case identisch

2.7 Quicksort

- „Sortieren durch Zerlegen“

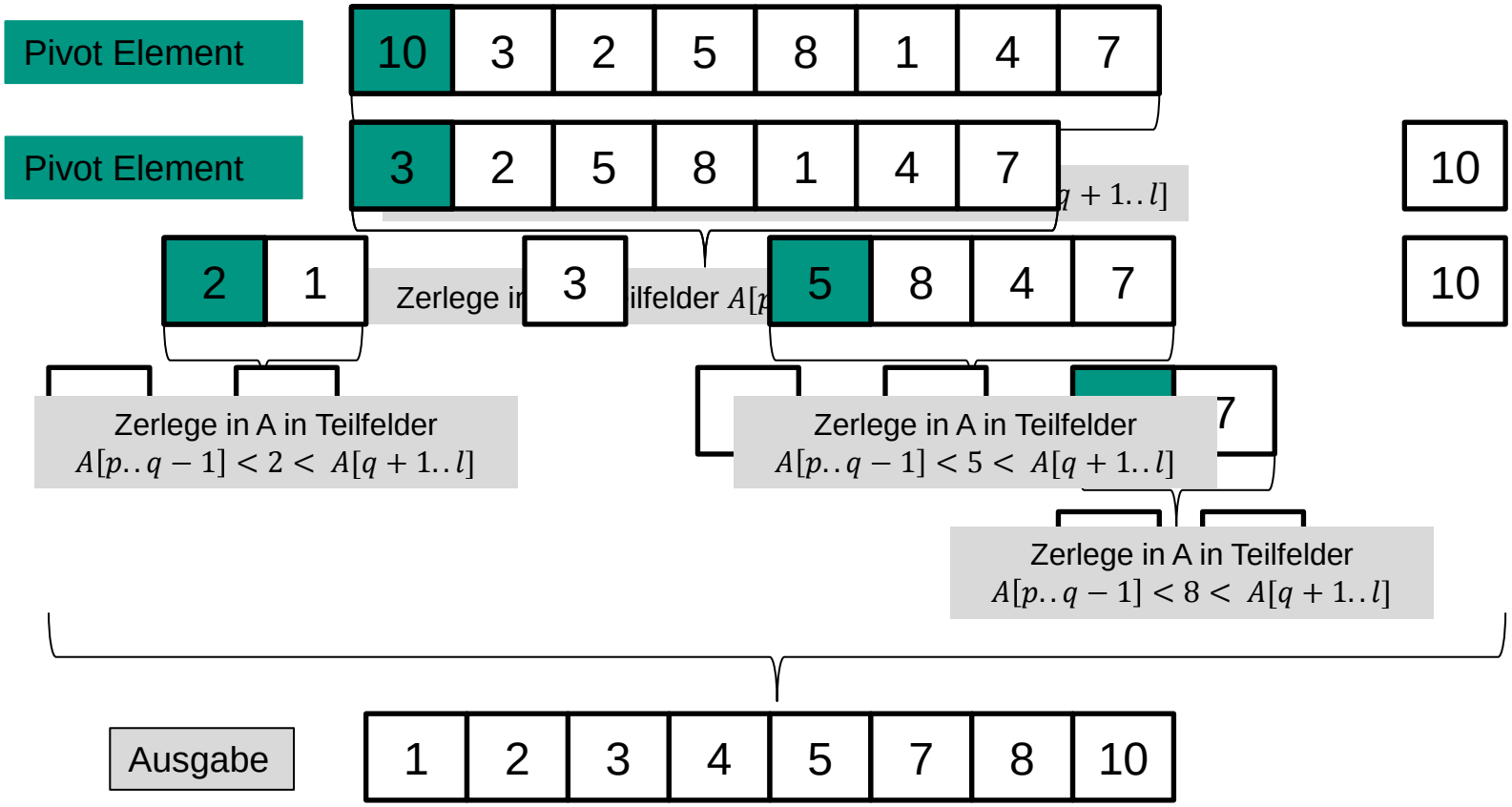
- Folgt dem „Teile-und-Herrsche“ Paradigma
  - Problem wird in Teilprobleme zerlegt
  - Ggf. rekursiv weiter teilen
  - Wenn das Teilproblem klein genug ist, auf direktem Weg lösen
  - Lösungen der Teilprobleme zur Gesamtlösung vereinigen

- Prinzip
  - Zerlege Gesamtaufgabe in zwei Sortierteilaufgaben, indem ein oder mehrere Elemente durch Platztauschen an die korrekte Stelle verbracht werden und die Teilfolgen links und rechts dann sortiert werden
  - 1962 von Hoare vorgeschlagen

2.7.1 Quicksort – Prinzip

- Sortieren des Feldes A der Größe l
  - Teilen
    - Zerlege A in $A[p..q-1]$, $A[q]$ und $A[q+1..l]$ wobei gilt:
    - $A[q-1] < A[q] < A[q+1]$
    - $A[q]$ nennt man *„Pivot Element“*
  - Herrschen
    - Sortiere die beiden Felder $A[p..q-1]$ und $A[q+1..l]$ durch Rekursion
  - Vereinigen
    - Füge die (sortierten) Felder zusammen

2.7.2 Quicksort – Beispiel



2.7.3 Quicksort – Pseudeocode

- Sortieren des Feldes A der Größe l
  - $QUICKSORT(A, p, r)$ sortiert alle Elemente im Feld $A[p..r]$
    - Erster Aufruf mit $QUICKSORT(A, 1, l)$
  - $PARTITION(A, p, r)$ zerlegt das Feld A in zwei Teilfelder $A[p..q-1]$ und $A[q+1..r\,]$ wobei q das Pivot-Element ist

- $QUICKSORT(A, p = 1, r = l)$

```
1 if p < r
2     q = PARTITION(A, p, r)
3     QUICKSORT(A, p, q − 1)
4     QUICKSORT(A, q + 1, r)
```

Zerlege Feld A in Teilfelder

$QUICKSORT$ rekursiv für beide Teilfelder aufrufen

Pseudocode II

- $PARTITION(A, p, r)$

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          Vertausche A[i] mit A[j]
7  Vertausche A[i + 1] mit A[r]
8  return i + 1
```

Wähle Pivot-Element

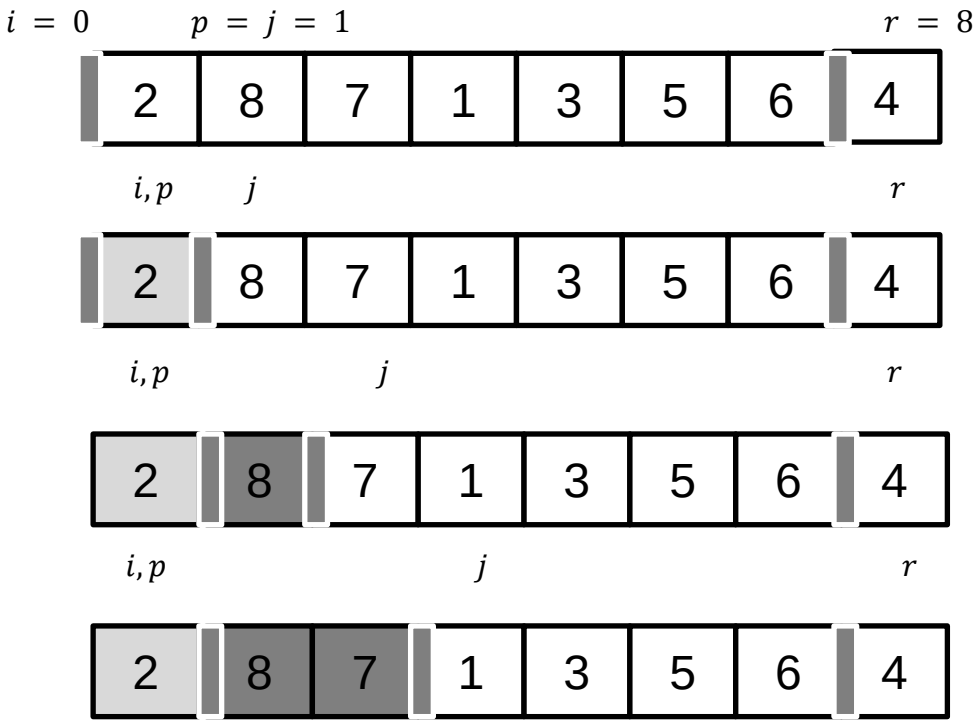
Zerlege A in zwei Teilfelder

Wenn betrachtetes Element kleiner als Pivot-Element sortiere Element in linkes Teilfeld

Sortiere Pivot-Element ein

Gib Pivot Stelle des Pivot Elements aus

2.7.4 Beispiel für *PARTITION*



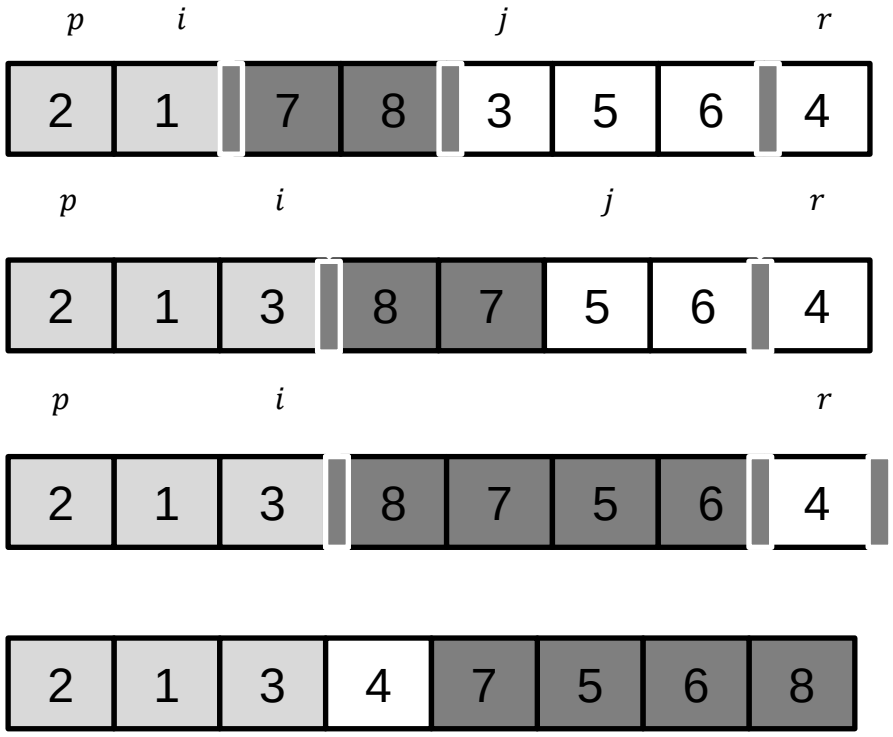
Pivot Element $x = A[r] = 4$

Sortiere 2 in Bereich < x ein

Sortiere 8 in Bereich > x ein

Sortiere 7 in Bereich > x ein

Beispiel für *PARTITION*



Sortiere 1 in Bereich < x ein

Sortiere 3 in Bereich < x ein

Sortiere 5,6 in Bereich > x ein

Vertausche $A[i+1]$ und $A[r]$

2.7.5 Quicksort – Komplexitätsanalyse

- Der Aufwand von Teile-und-Herrsche-Algorithmen lässt sich als Rekursionsgleichung darstellen

- Es gilt für *Quicksort* im Worst-Case
  - Sortieren eines Elementes liegt in $\theta(1)$
  - Sortieren mehrerer Elemente zerlegt sich in die Teilschritte
    - Teilen – Berechnen des Pivot Elements, $\theta(1)$
    - Mischen – Aufwand ist linear $\theta(n)$
    - Herrschen – ?
  - Worst-Case in *Quicksort* tritt ein, wenn $PARTITION$ in jedem rekursiven Aufruf das Feld in Teilfelder mit n-1 Elementen und 0 Elementen zerlegt
    - Im ersten Aufruf sind somit n-1 Vergleiche notwendig, im zweiten Aufruf n-2 Vergleiche,…
    - Allgemein: Im k-ten Aufruf gibt es n-k Vergleiche
  - Für $T(n)$ gilt dann:

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

➡ *Quicksort* hat im Worst-Case eine Laufzeit von $\theta(n^2)$

Komplexitätsanalyse II

- Es gilt für *Quicksort* im Best-Case
  - Sortieren eines Elementes liegt in $\theta(1)$
  - Sortieren mehrerer Elemente zerlegt sich in die Teilschritte
    - Teilen – Berechnen des Pivot Elements, $\theta(1)$
    - Mischen – Aufwand ist linear $\theta(n)$
    - Herrschen -- ?
  - Best-Case in *Quicksort* tritt ein, wenn $PARTITION$ in jedem rekursiven Aufruf das Feld in Teilfelder mit $n/2$ Elementen zerlegt
  - Für $T(n)$ gilt dann
    - $T(n) \leq 2T\left(\frac{n}{2}\right) + \theta(n)$
    - Mit Hilfe der Master-Theorems kann man zeigen, dass $T(n) = \theta(n * \lg n)$

➡ *Quicksort* hat im Best-Case eine Laufzeit von $\theta(n * \lg n)$

Komplexitätsanalyse III

- Frage: Was gilt für *Quicksort* im Average-Case?
  - *Quicksort* auch im Average-Case $T(n) = \theta(n * \lg n)$
  - Die Anzahl der notwendigen Operationen lässt sich ausdrücken mit

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

  - *T(n)* setzt sich zusammen aus
    - *n+1* Vergleichen für alle Elemente des Feldes
    - Durchschnittliche Kosten für eine zufällige Zerlegung
      - Jedes Element k kann mit Wahrscheinlichkeit $\frac{1}{k}$ Pivot-Element werden
      - Damit entstehen zufällige Teilfelder der Größe $k-1$ und $\mathrm{n} - k$
      - Für alle Elemente lassen sich somit die erwarteten zufälligen Kosten aufsummieren
  - Approximation der Gleichung zeigt, dass $T(n) = \theta(n * \lg n)$

2.8 Radixsort

- „Fachverteilen“
- Allgemein wird *Radixsort* eingesetzt, um n d-stellige Zahlen zu sortieren
  - Beispielhafte Vorgehensweise:
    - Zahlen nach der höchstwertigen Stelle sortieren
      - In *d* „Fächer“ ablegen
    - „Fächer“ rekursiv sortieren
    - „Fächer“ in geeigneter Reihenfolge zusammenlegen
  - Problem ?
    - Es werden viele temporäre „Fächer“ benötigt

- *Radixsort* löst dieses Sortierproblem auf effiziente Weise
  - Wurde in Lochkartensystemen eingesetzt

2.8.1 Radixsort – Prinzip

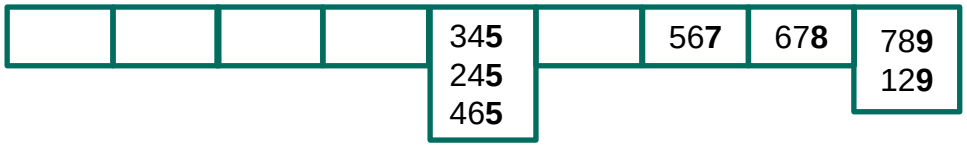
- Grundlegender Ablauf
  - Um *Radixsort* auf eine Folge anzuwenden muss gelten
    - Sortierschlüssel müssen Zeichen eines endlichen Alphabets sein
      - z.B. {0,1,2,3,4,5,6,7,8,9}
    - Es muss eine Totalordnung zwischen den Zeichen bestehen
    - Länge der Sortierschlüssel ist begrenzt
      - z.B. d-stellige Zahlen
  - *Radixsort* funktioniert in zwei Phasen
    - Partitionierungsphase
      - Die zu sortierenden Elemente werden auf Fächer verteilt
    - Sammelphase
      - Die verteilten Elemente werden wieder eingesammelt
    - Diese Phasen werden für jede Stelle der zu sortieren Schlüssel wiederholt

2.8.2 Radixsort – Beispiel

- Alphabet = {1,2,3,4,5,6,7,8,9}
- Folge = {345, 789, 245, 567,129, 465, 678}

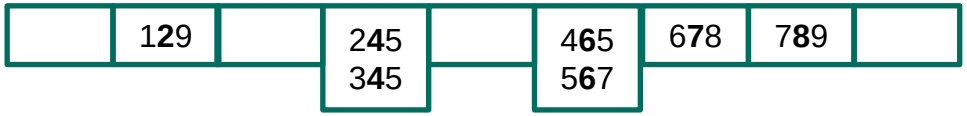
Fächer:

1. Partitionierung



1. Sammelphase 345, 245, 465, 567, 678, 789, 129

2. Partitionierung



2. Sammelphase 129, 245, 345, 465, 567, 678, 789

3. Partitionierung

3. Sammelphase 129, 245, 345, 465, 567, 678, 789

2.8.3 Radixsort – Pseudocode und Komplexitätsanalyse

- Sortieren des Feldes A der Größe l
  - Jedes Element des Feldes hat d Stellen
  - Jede Stelle kann k Werte annehmen
  - Es existiert ein stabiles Sortierverfahren S, das in linearer Zeit sortiert
    - Genauer: S sortiert pro Zwischenschritt in $\theta(n+k)$

- $RADIXSORT(A, d)$
  1 **for** i $= 1$ **to** d
  2 $\quad Sortiere\ A\ nach\ Stelle\ i\ mit\ Hilfe\ von\ S$
  3 $\quad Sammle\ Fächer\ ein$

Iteriere über alle Stellen

Sortiere jeweils i-te Stelle in Fächer und sammle ein

- Komplexität von *Radixsort*
  - Unter Annahme, dass *S* in $\theta(n+k)$ pro Zwischenschritt sortiert, sortiert Radixsort in $\theta(d\ (n+k))$
    - Jeder Durchlauf über n d-stellige Zahlen benötigt $\theta(n+k)$
    - Es gibt d Durchläufe, bis alle Zahlen sortiert sind
- Für ein konstantes d und $k = O(n)$ läuft Radixsort in linearer Zeit

2.9 „Wrap-Up“

- Zusammenfassung der Laufzeiten der hier betrachteten Sortieralgorithmen

Algorithmus	Best-Case	Worst-Case	Average-Case
Bubblesort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertionsort	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Mergesort	$\theta(n * \lg n)$	$\theta(n * \lg n)$	$\theta(n * \lg n)$
Quicksort	$\theta(n * \lg n)$	$\theta(n^2)$	$\theta(n * \lg n)$
Radixsort	$\theta(d(n+k))$	$\theta(d(n+k))$	$\theta(d(n+k))$

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3
- [Sed02] Robert Sedgewick, „Algorithmen“, Pearson Studium, 2. Auflage, 2002, 742 Seiten, ISBN 3827370329
- [Vöc08] Berthold Vöcking, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler, Heribert Vollmer, Dorothea Wagner, „Taschenbuch der Algorithmen“, Springer, 1. Auflage, 2008, 448 Seiten, ISBN 16145216

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 3 – Folgen als Felder und Listen

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

3. Folgen als Felder und Listen

3.0 Dynamische Mengen

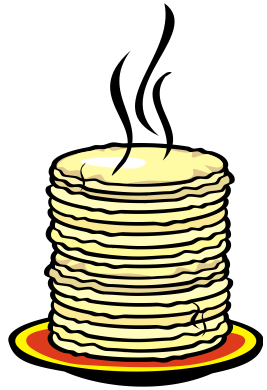
- Algorithmen operieren oft auf dynamischen Mengen
- Typische Operationen auf einer Menge S für ein Element mit Schlüssel k
  - $SEARCH(S, k)$
  - $INSERT(S, k)$
  - $DELETE(S, k)$
  - $MINIMUM(S)$
  - $MAXIMUM(S)$
- Verschiedene Datenstrukturen zur Verwaltung dynamischer Mengen
  - Stapel und Warteschlangen
  - Verkettete Listen
  - Hashtabellen
  - Bäume
  - …

  → Unterscheiden sich u.a. hinsichtlich des Aufwands für obige Operationen

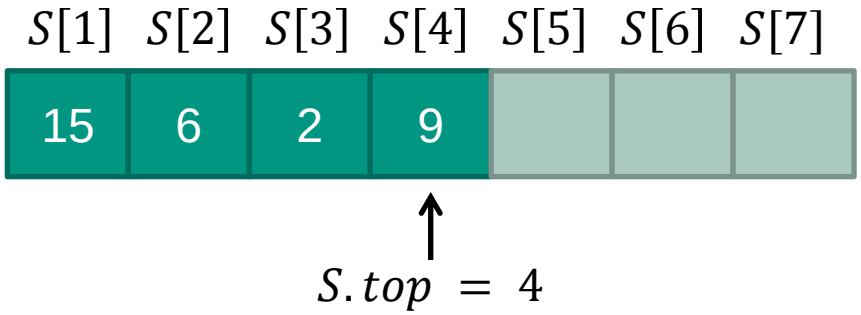
3.1 Stapel und Warteschlangen

- Stapel (engl. *Stacks*) und Warteschlangen (engl. *Queues*) sind einfache Datenstrukturen für dynamische Mengen
- Gemeinsame Eigenschaft: Das zu löschende Elemente durch die $DELETE$-Operation ist vorherbestimmt
  - Stapel: Das zuletzt eingefügte Element wird entfernt (Bedienstrategie: last-in/first out, kurz LIFO)
  - Warteschlangen: Das zuerst eingefügte Element wird entfernt (Bedienstrategie: first-in/first out, kurz FIFO)
- Beiden Datenstrukturen können durch ein einfaches Feld realisiert werden

3.1.1 Stapel



- Anlehnung an reale Stapel, wie z.B. Tellerstapel
  - Teller werden in umgekehrter Reihenfolgen vom Stapel genommen, wie sie auf den Stapel gestellt wurden
  - Nur der oberste Teller ist zugänglich
- $INSERT$ wird bei Stapeln auch als $PUSH$ bezeichnet
- $DELETE$ wird als POP bezeichnet (ohne Angabe eines Arguments)
- Für Stapel mit höchstens n Elementen
  - Einfache Implementierung durch Feld $S[1..n]$
    - Unterstes Element: $S[1]$
    - Oberstes Element: $S[S.top]$
  - Zwei Fehlerfälle
    - Stapelunterlauf (*stack underflow*)
    - Stapelüberlauf (*stack overflow*)



- Anwendungsbeispiel in der Praxis
  - Z.B. Übergabe von Parametern an eine Methode

Stapeloperationen

- Prüfen, ob Stapel leer ist
  - $STACK - EMPTY(S)$
    1 **if** $S.top == 0$
    2 **return** $WAHR$
    3 **else return** $FALSCH$
- Neues Element x hinzufügen
  - $PUSH(S, x)$
    1 $S.top = S.top + 1$
    2 $S[S.top] = x$
- Oberstes Element vom Stapel nehmen
  - $POP(S)$
    1 **if** $STACK - EMPTY(S)$
    2 **error** „Stapelunterlauf“
    3 **else** $S.top = S.top - 1$
    4 **return** $S[S.top + 1]$
- Jede dieser Stapeloperationen hat eine Zeitaufwand von $O(1)$

Index des obersten Elements prüfen

Index des obersten Elements anpassen

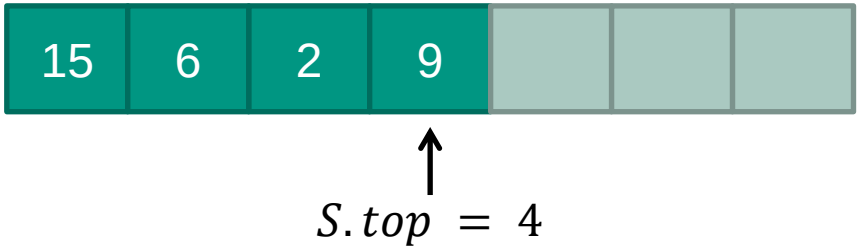
Element hinzufügen

Index des obersten Elements anpassen

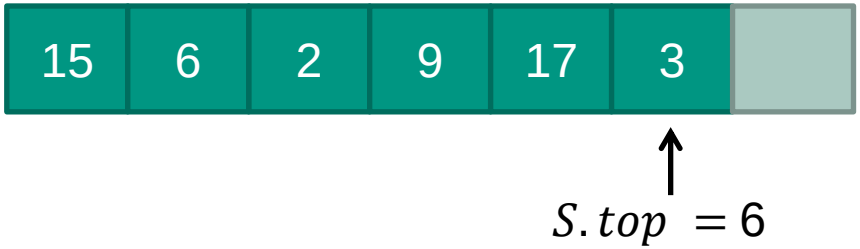
Element zurückgeben

Beispiel für Stapeloperationen

- Ausgangszustand



- Stapel nach $PUSH(S, 17)$ und $PUSH\ (S, 3)$



- Stapel nach $POP(S)$ – es wurde das Element 3 zurückgeliefert

3.1.2 Warteschlangen

- Anlehnung an reale Warteschlangen, wie z.B. Bankkunden am Schalter
  - Aus einer Warteschlange wird immer das Element am Kopf der Schlange zuerst entfernt (längster wartende Kunde)
- $INSERT$ wird bei Warteschlangen als $ENQUEUE$ bezeichnet
- $DELETE$ wird als $DEQUEUE$ bezeichnet
- Warteschlange mit höchstens $n - 1$ Elementen
  - Einfach Implementierung durch Feld $Q[1..n]$
    - Attribut $Q.kopf$ zeigt auf den Kopf der Warteschlange (also auf das älteste Element)
    - Attribut $Q.ende$ zeigt auf die Stelle, an der neues Element eingefügt wird
  - Feld wird als Ring aufgefasst:
    - Nach der Stelle n folgt wieder die Stelle 1

Operationen auf Warteschlangen

- Neues Element x anhängen
  - $ENQUEUE(Q, x)$

```
1  Q[Q.ende] = x
2  if Q.ende == Q.länge
3      Q.ende = 1
4  else Q.ende = Q.ende + 1
```

Element am Ende anhängen

Falls Feldende erreicht, nächstes Element vorne einfügen

- Vorderstes Element entfernen
  - $DEQUEUE(Q)$

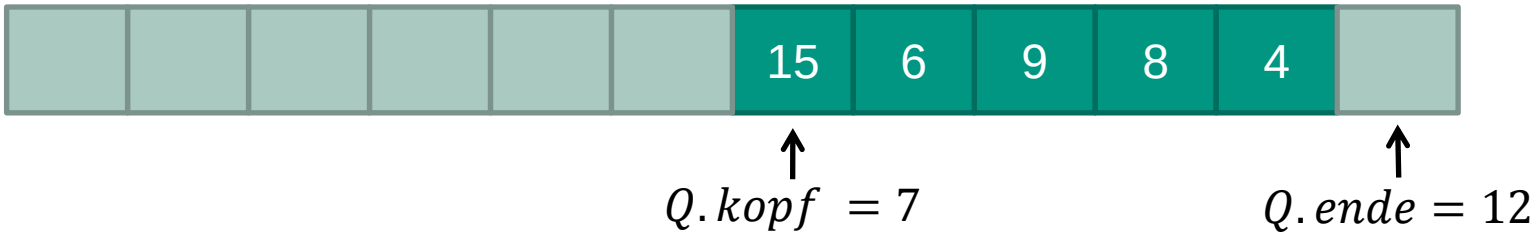
```
1  x = Q[Q.kopf]
2  if Q.kopf == Q.länge
3      Q.kopf = 1
4  else Q.kopf = Q.kopf + 1
5  return x
```

Letztes Element im Feld entfernt → Kopfzeiger auf Feldanfang

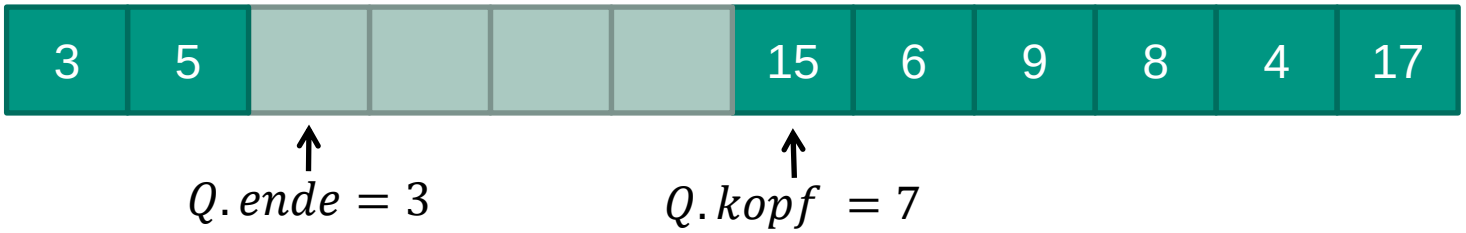
- Beide Operationen haben einen Zeitaufwand von $O(1)$

Beispiel für Operationen auf Warteschlangen

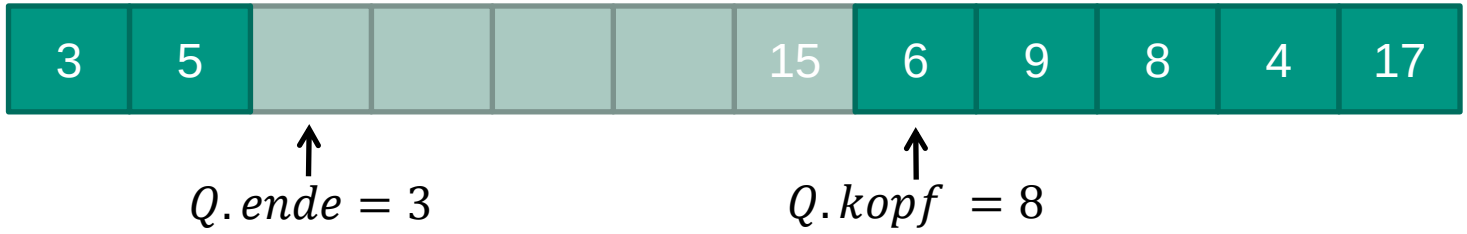
- Ausgangszustand



- Nach ENQUEUE(Q,17), ENQUEUE(Q,3) und ENQUEUE (Q,5)

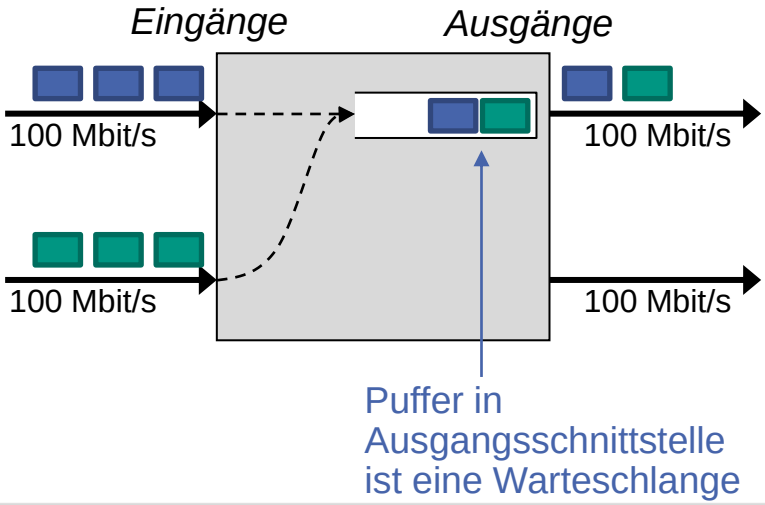


- Nach DEQUEUE(Q) – es wurde das Element 15 zurückgegeben



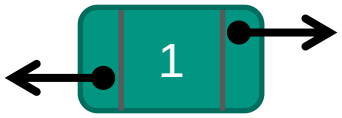
Anwendungsbeispiel aus der Telematik

- Im Internet werden Router eingesetzt um Dateneinheiten weiterzuleiten
  - Mehrere Ein- und Ausgänge, die über Datenleitungen die Verbindung zu anderen Routern und Endsystemen herstellen
  - Router entscheidet anhand der Zieladresse, welcher Ausgang gewählt wird
- Zwei Dateneinheiten mit gleichem Zielausgang: entweder verwerfen oder in einem Puffer zwischenspeichern
- FCFS (First-Come-First-Serve) Strategie
  - Eine Warteschlange pro Ausgang als Puffer (FIFO-Queue)
  - Bearbeitung der Dateneinheiten in Ankunftsreihenfolge
  - Ist kein Pufferplatz mehr frei, so werden neu ankommende Dateneinheiten verworfen („Tail-Drop“)

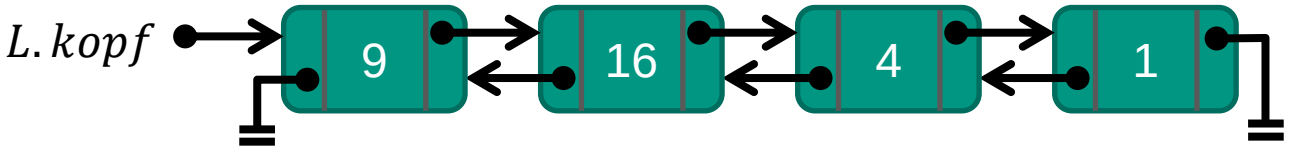


3.2 Verkettete Listen

- Eine verkettete Liste ist eine einfache Datenstruktur, die
  - sehr flexibel ist und alle zuvor genannten Operationen für dynamische Mengen unterstützt
  - alle Objekte in einer linearen Reihenfolge anordnet
- Im Unterschied zum Feld wird Reihenfolge mittels Zeigern (bzw. Referenzen) anstelle von Feldindizes realisiert
- Ein Listenobjekt enthält
  - den Schlüssel (also die eigentlichen „Daten“)
  - Zeiger auf das Nachfolgerobjekt
  - Zeiger auf das Vorgängerobjekt



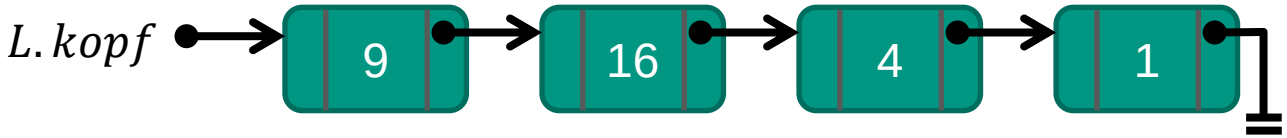
- Zusätzlich hat jede Liste L einen Zeiger $L.kopf$ auf das erste Element



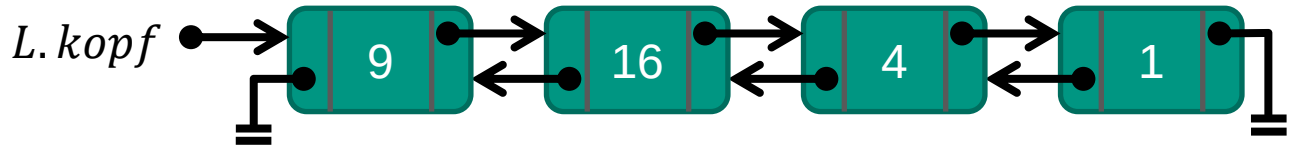
- Gibt es kein Nachfolger- oder Vorgängerelement, ist der Zeiger NIL

3.2.1 Varianten verketteter Listen (I)

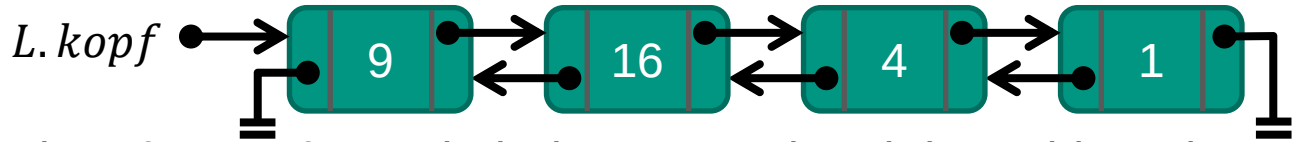
- Verkettungsart
  - Einfach verkettete Listen haben nur einen Nachfolgerzeiger



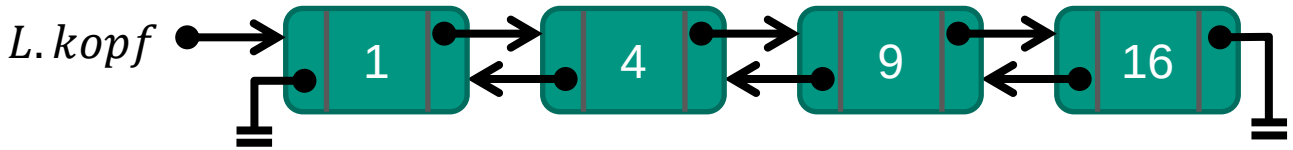
  - Doppelt verkettete Listen haben Vorgänger- und Nachfolgerzeiger



- Sortierung
  - Bei unsortierten Listen spielt die Reihenfolge der Elemente keine Rolle

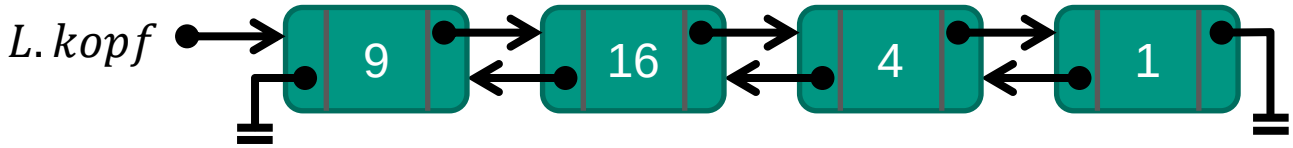


  - Bei sortierten Listen sind Elemente anhand der Schlüssel geordnet

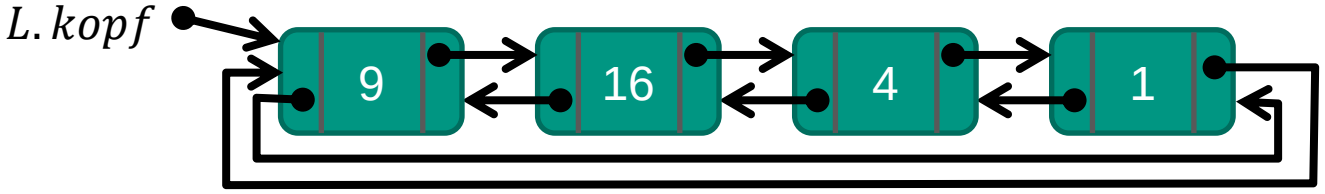


Varianten verketteter Listen (II)

- Verbindung zwischen Kopf und Ende der Liste
  - In nichtzyklische Listen hat das Kopfelement keinen Vorgänger bzw. das letzte Element keinen Nachfolger



  - In zyklischen Listen sind das Kopfelement und das letzte Element miteinander verbunden



- Im Folgenden werden zunächst doppelt verkettete, unsortierte und nichtzyklische Listen betrachtet

3.2.2 Doppelt verkettete Listen: Durchsuchen

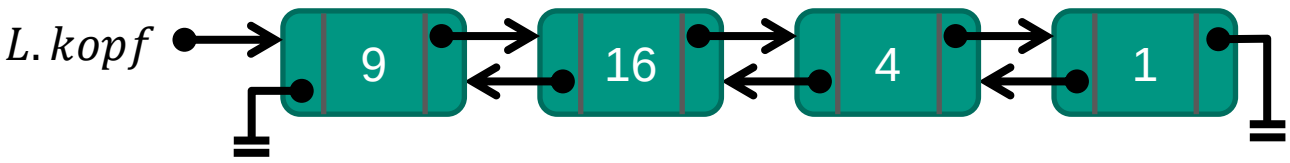
- $LIST - SEARCH(L, k)$

```
1  x = L.kopf
2  while x ≠ NIL and x.schlüssel ≠ k
3      x = x.nachf
4  return x
```

Ende der Liste erreicht?

Schlüssel gefunden?

Nachfolgerelement betrachten



- Aufwand im Worst-Case
  - Für Liste mit n Elementen: $\Theta(n)$

Doppelt verkettete Listen: Einfügen

- Hier: Einfügen am Anfang der Liste
  - Einfügen an beliebiger Stelle als Übungsaufgabe

- $LIST - INSERT(L, x)$

```
1  x.nachf = L.kopf
2  if L.kopf ≠ NIL
3      L.kopf.vorg = x
4  L.kopf = x
5  x.vorg = NIL
```

Liste nicht leer?

Element x ist neuer Vorgänger für bisheriges Kopfelement

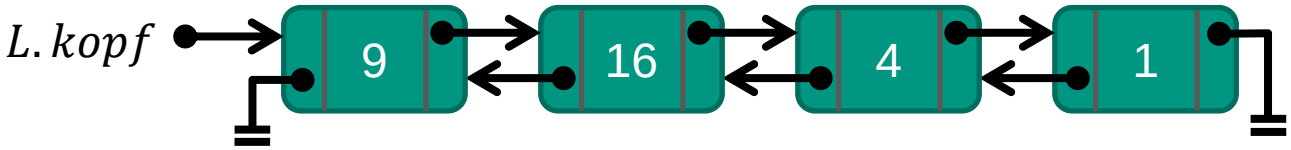
Element x ist neues Kopfelement

Element x hat keinen Vorgänger

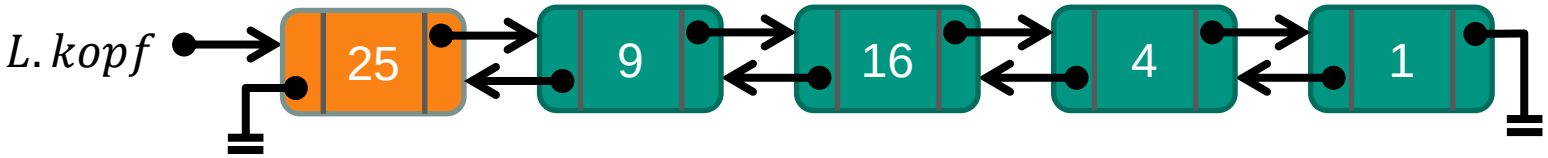
- Aufwand für Einfügen am Anfang der Liste: $O(1)$

Beispiel: Einfügen in doppelt verketteter Liste

- Anfangszustand



- Nach der Ausführung von $LIST - INSERT(L, x)$ mit $x.schlüssel = 25$



Doppelt verkettete Listen: Entfernen

- $LIST - DELETE(L, x)$

```
1   if x.vorg ≠ NIL
2       x.vorg.nachf = x.nachf
3   else L.kopf = x.nachf
4   if x.nachf ≠ NIL
5       x.nachf.vorg = x.vorg
```

Element nicht Kopfelement?

Nachfolgerzeiger anpassen

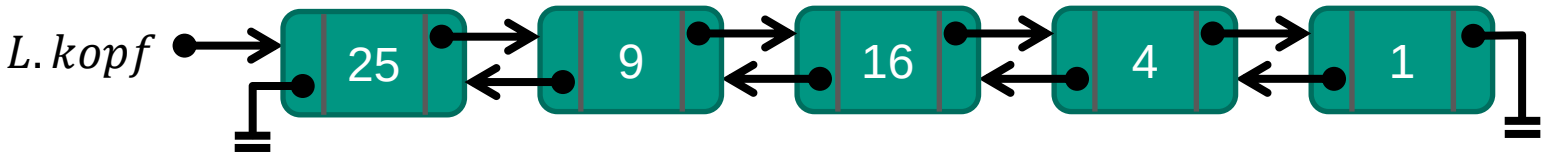
Sonst: Neues Kopfelement

Vorgängerzeiger das Nachfolgers anpassen

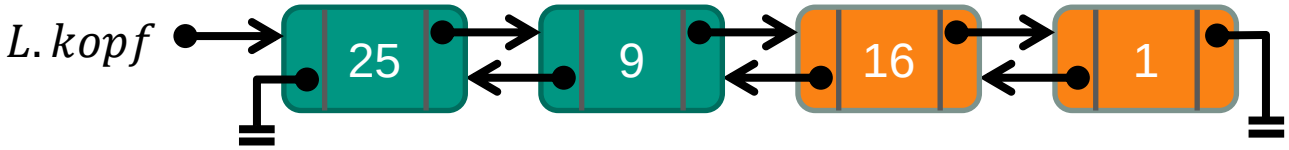
- Aufwand für Entfernen mit $LIST - DELETE$ falls Element bekannt: $O(1)$
- Falls Element zunächst mit $LIST - SEARCH$ gesucht wird: $\Theta(n)$

Beispiel: Entfernen aus doppelt verketteter Liste

- Anfangszustand



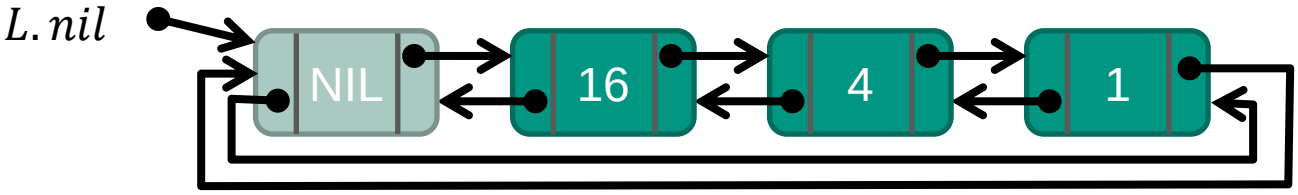
- Nach der Ausführung von $LIST - DELETE(L, x)$ mit $x.schlüssel = 4$



3.2.3 Wächterelement

- Listenoperationen können durch Wächterelement vereinfacht werden
  - Erspart Sonderbehandlung für erstes und letztes Listenelement
- Wächterelement ist ein Dummy-Objekt $L.nil$ mit Wert NIL

- Zyklische, doppelt verkettete Liste mit Wächterelement zwischen Kopf und Ende



- $L.kopf$ nicht mehr benötigt → Wird durch $L.nil.nachf$ ersetzt

Entfernen mit Wächterelement

- Mit Wächterelement ist keine besondere Behandlung am Kopf und am Ende der Liste erforderlich
- $LIST-DELETE(L,x)$

```
1  if x.vorg ≠ NIL
2      x.vorg.nachf = x.nachf
3  else L.kopf = x.nachf
4  if x.nachf ≠ NILL
5      x.nachf.vorg = x.vorg
```

Ohne Wächterelement

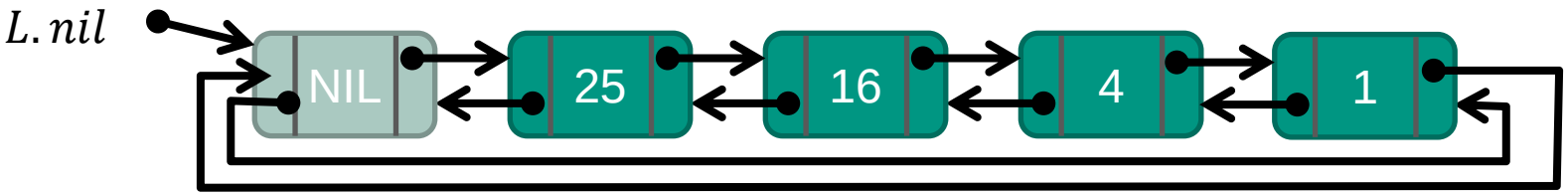
- $LIST-DELETE'(L,x)$

```
1  x.vorg.nachf = x.nachf
2  x.nachf.vorg = x.vorg
```

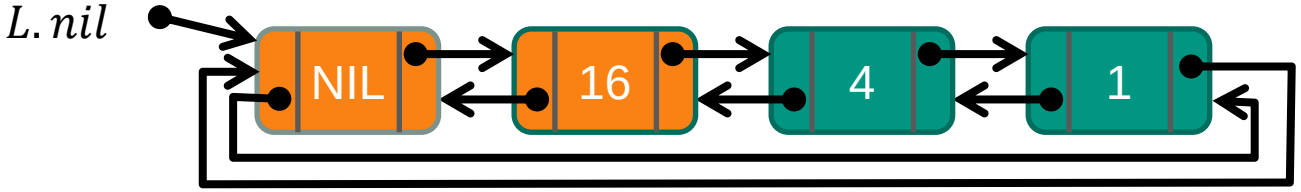
Mit Wächterelement

Beispiel: Entfernen mit Wächterelement

- Anfangszustand



- Nach der Ausführung von $LIST - DELETE(L, x)$ mit $x.schlüssel = 25$



Durchsuchen mit Wächterelement

- $LIST - SEARCH(L, k)$

```
1  x = L.kopf
2  while x ≠ NIL and x.schlüssel ≠ k
3      x = x.nachf
4  return x
```

$L.Kopf$ durch $L.nil.nachf$ ersetzen

NIL durch $L.nil$ ersetzen

Ohne Wächterelement

- $LIST - SEARCH'(L, k)$

```
1  x = L.nil.nachf
2  while x ≠ L.nil and x.schlüssel ≠ k
3      x = x.nachf
4  return x
```

Mit Wächterelement

Einfügen mit Wächterelement

- $LIST - INSERT(L, x)$

```
1  x.nachf = L.kopf
2  if L.kopf ≠ NIL
3      L.kopf.vorg = x
4  L.kopf = x
5  x.vorg = NIL
```

$L.Kopf$ durch $L.nil.nachf$ ersetzen

Ohne Wächterelement

NIL durch $L.nil$ ersetzen

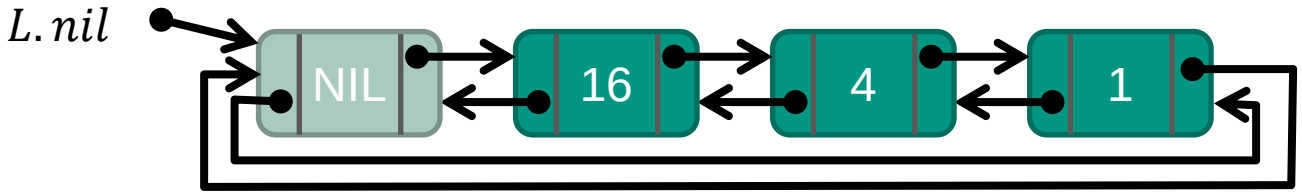
- $LIST - INSERT'(L, x)$

```
1  x.nachf = L.nil.nachf
2  L.nil.nachf.vorg = x
3  L.nil.nachf = x
4  x.vorg = L.nil
```

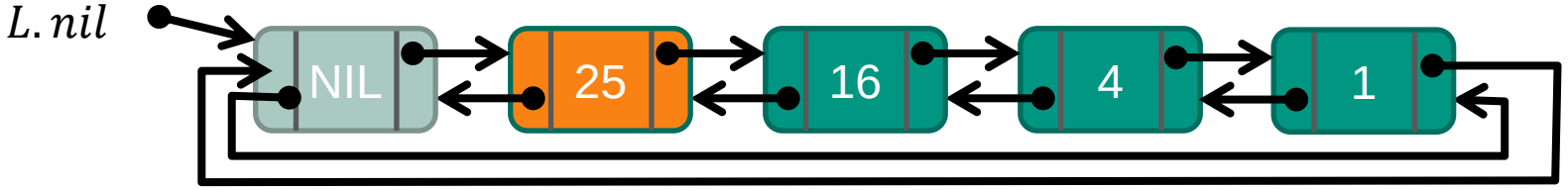
Mit Wächterelement

Beispiel: Einfügen mit Wächterelement

- Anfangszustand



- Nach der Ausführung von $LIST - INSERT(L, x)$ mit $x.schlüssel = 25$

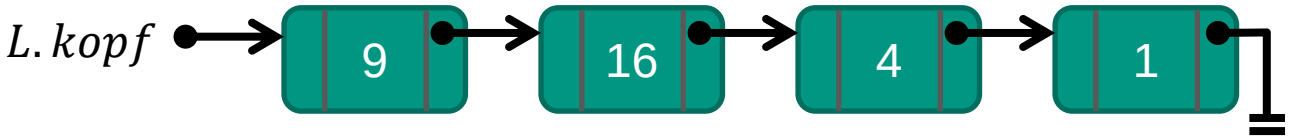


Bewertung von Wächterelementen

- Vorteile
  - Code wird vereinfacht und dadurch leichter lesbar
  - In einigen Fällen kann die Laufzeit geringfügig beschleunigt werden
    - Asymptotische Zeitschranken ändern sich jedoch nicht
    - Lediglich kleinere Koeffizienten
- Nachteile
  - Zusätzlicher Speicherverbrauch
    - Insbesondere nachteilig bei vielen kleinen Listen

3.2.4 Einfach verkettete Listen

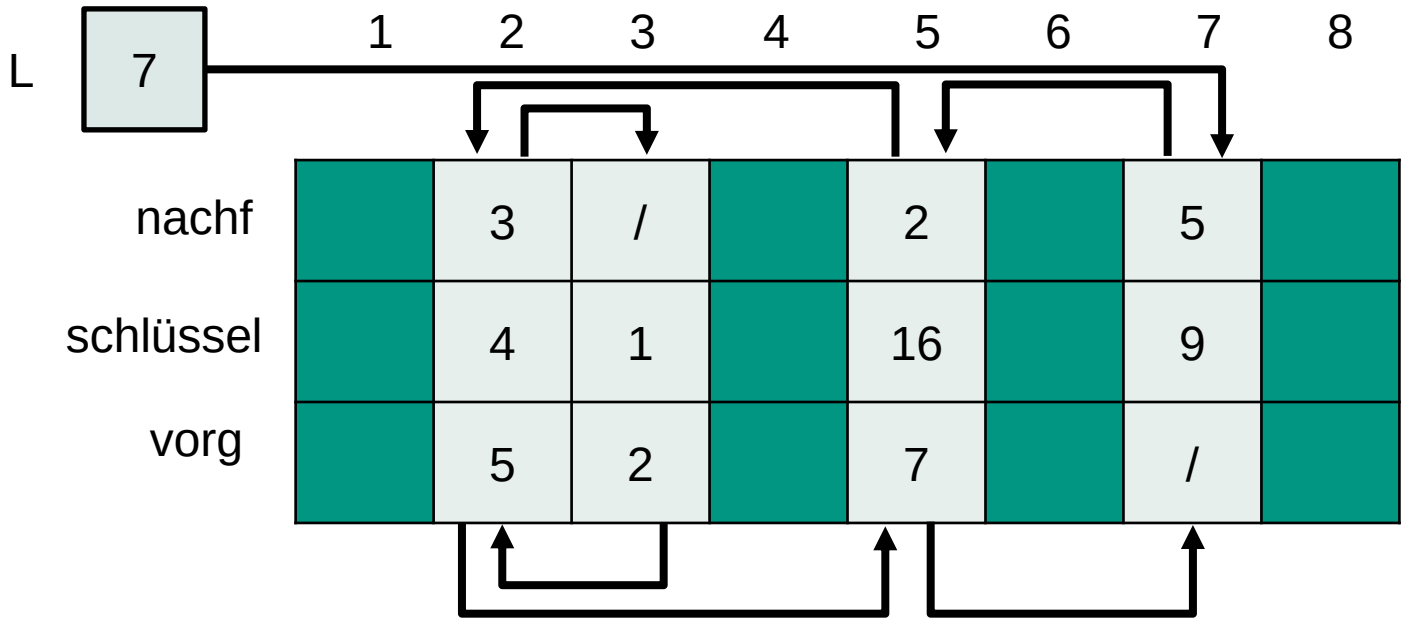
- Nur Zeiger auf Nachfolgerelement



- Vorteile
  - Geringerer Speicherbedarf
  - Setzen der Vorgängerzeiger entfällt
    - Einfacherer Code für Durchsuchen und Einfügen
    - Etwas geringerer Zeitaufwand beim Einfügen
- Nachteile
  - Löschen von Elementen aufwändig
    - Liste von vorne durchgehen und jeweils Vorgängerelement merken, bis zu löschendes Element gefunden wurde → $O(n)$
  - Vorgängerelement kann nicht einfach ermittelt werden (wird in sortierten Listen manchmal benötigt)

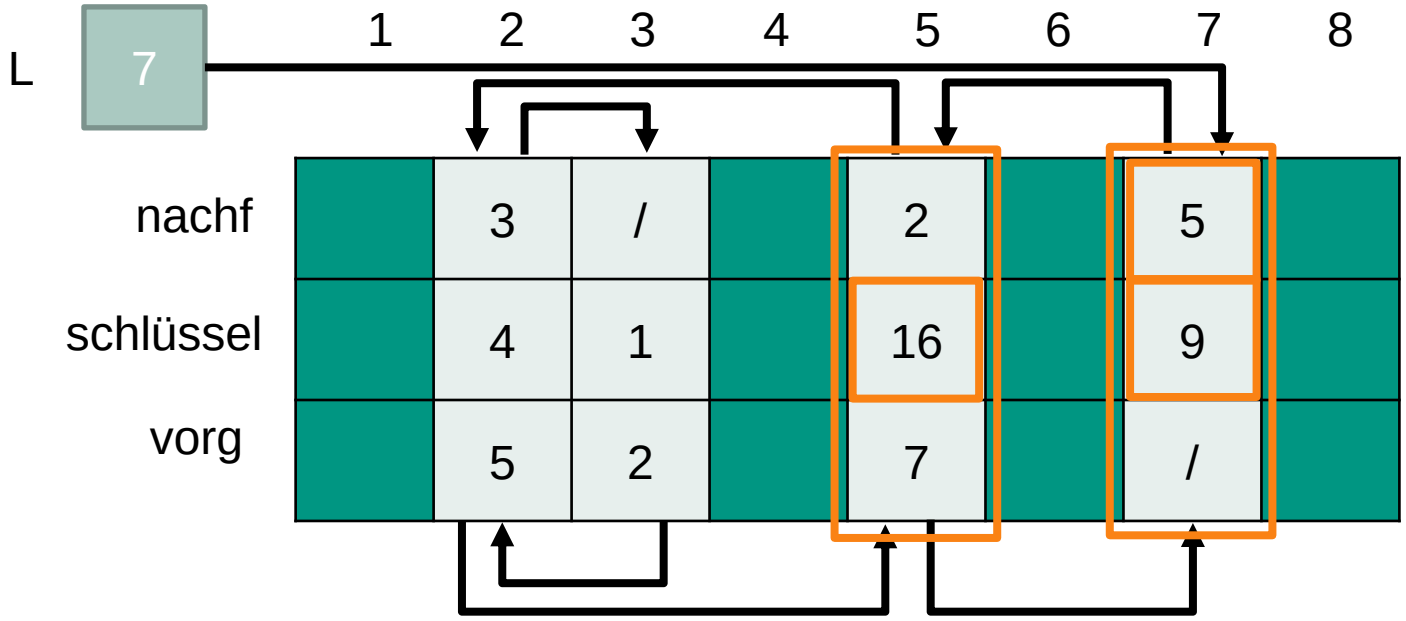
3.3 Implementierung von Zeigern und Objekten (I)

- Wie können Zeiger und Objekte implementiert werden, falls diese von einer Programmiersprache nicht bereitgestellt werden?
- Einfache Lösung: Verwendung mehrerer Felder
  - schlüssel[] enthält Schlüssel, die momentan in der Liste sind
  - vorg[] und nachf[] enthalten Zeiger auf Vorgänger und Nachfolger
  - „Zeiger“ sind hier einfach Feldindizes



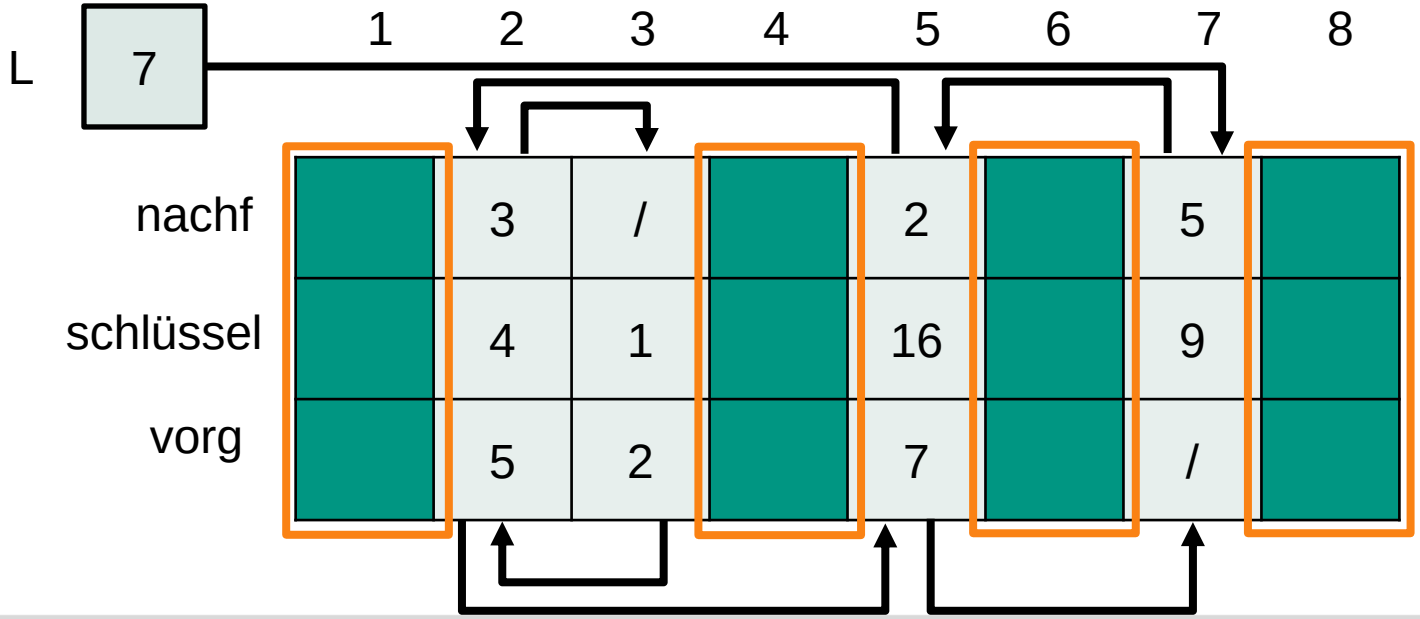
Implementierung von Zeigern und Objekten (II)

- Beispiel
  - Kopfelement hat Index 7
  - Schlüssel des ersten Elements ist $schlüssel[7] = 9$
  - Index des nächsten Elements ist $nachf[7] = 5$
  - Schlüssel des nächsten Elements ist $schlüssel[5] = 16$



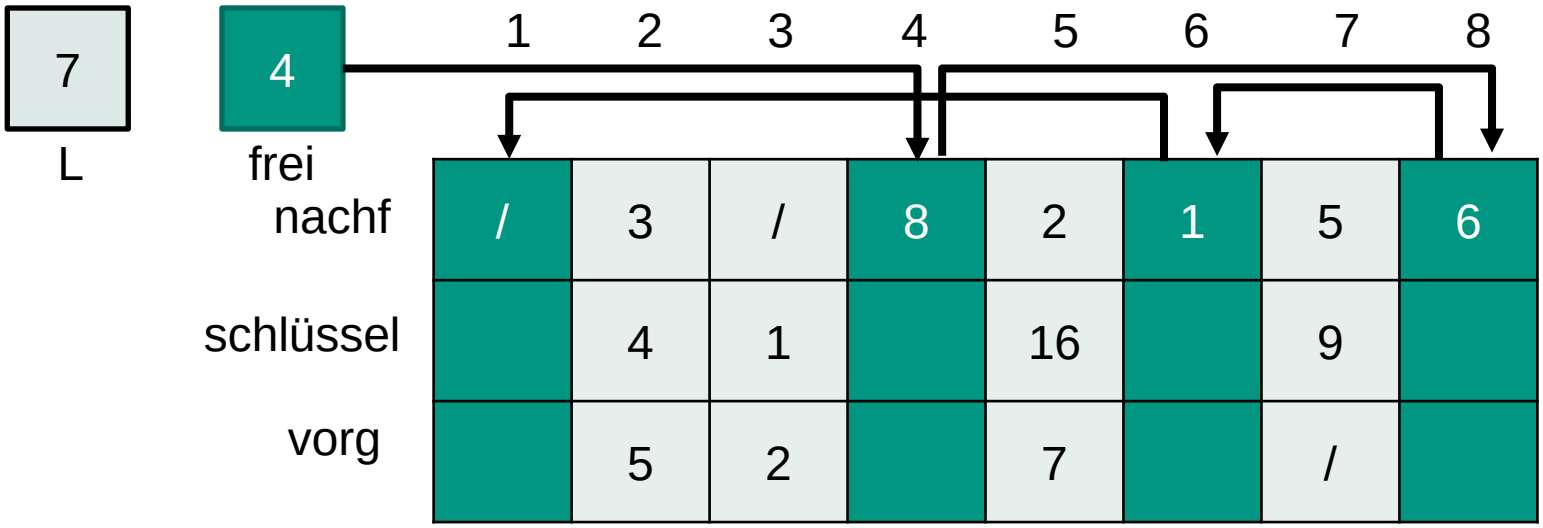
Speicherverwaltung (I)

- Speicherverwaltung, wenn Objekte eingefügt oder gelöscht werden?
  - Bei Java gibt der Garbage Collector automatisch Speicher für nicht mehr genutzte Objekte frei
  - In vielen Fällen muss Speicherverwaltung jedoch selbst von der Anwendung realisiert werden
- Für verkettete Listen in der Mehrfelddarstellung gilt
  - In Liste mit n Schlüsseln und Feldern der Länge m sind $m - n$ Objekte frei



Speicherverwaltung (II)

- Die freien Objekte können für zukünftige einzufügende Elemente verwendet werden
- Freie Objekte werden in einfach verketteter Freiliste gespeichert
  - Freiliste $frei$ kann mit der Liste L verflochten gespeichert werden
  - Kopf der Liste wird in Variable frei gespeichert
  - vorg[] wird in der Freiliste nicht verwendet (da nur einfach verkettet)



Speicherverwaltung (III)

- Freiliste arbeitet wie ein Stapel
  - Das als nächstes allokierte Objekt ist das zuletzt freigegebene

- $ALLOCATE-OBJECT()$

```
1  if frei == NIL
2      error „Speicherüberlauf“
3  else
4      x = frei
5      frei = x.nachf
6      return x
```

- $FREE-OBJECT(x)$

```
1  x.nachf = frei
2  frei = x
```

Keine freien Objekte vorhanden

Erstes Element der Freiliste verwendet

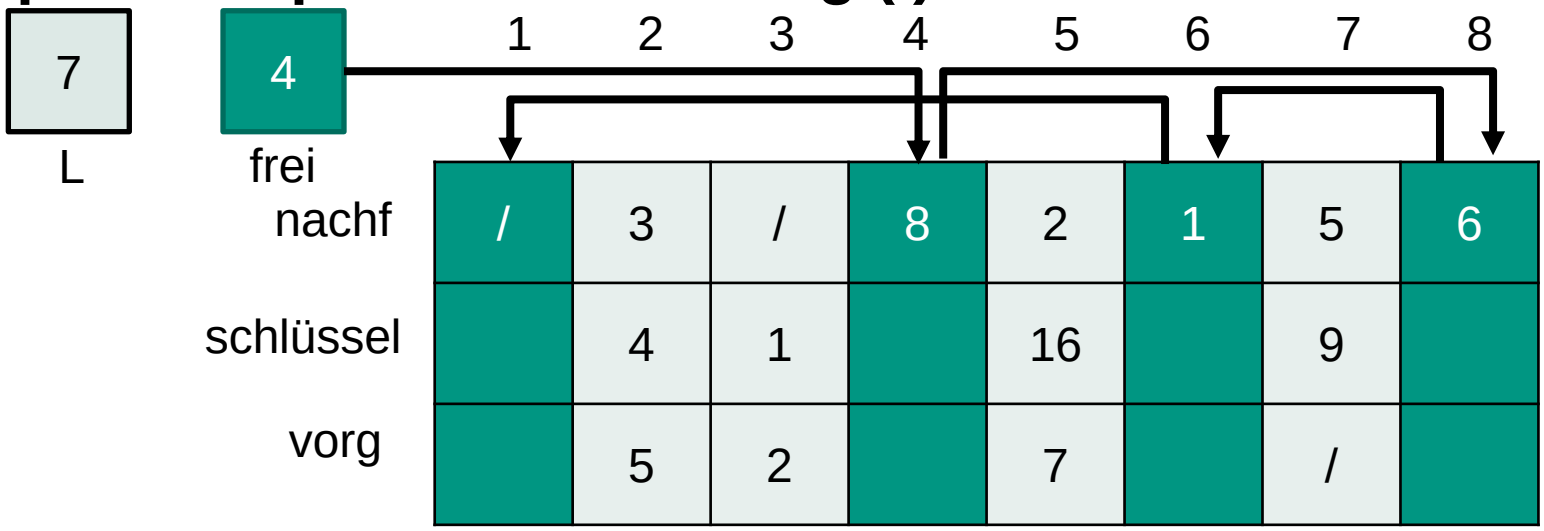
Kopfindex der Freiliste aktualisieren

Freigegebenes Element x vorne in Freiliste einhängen

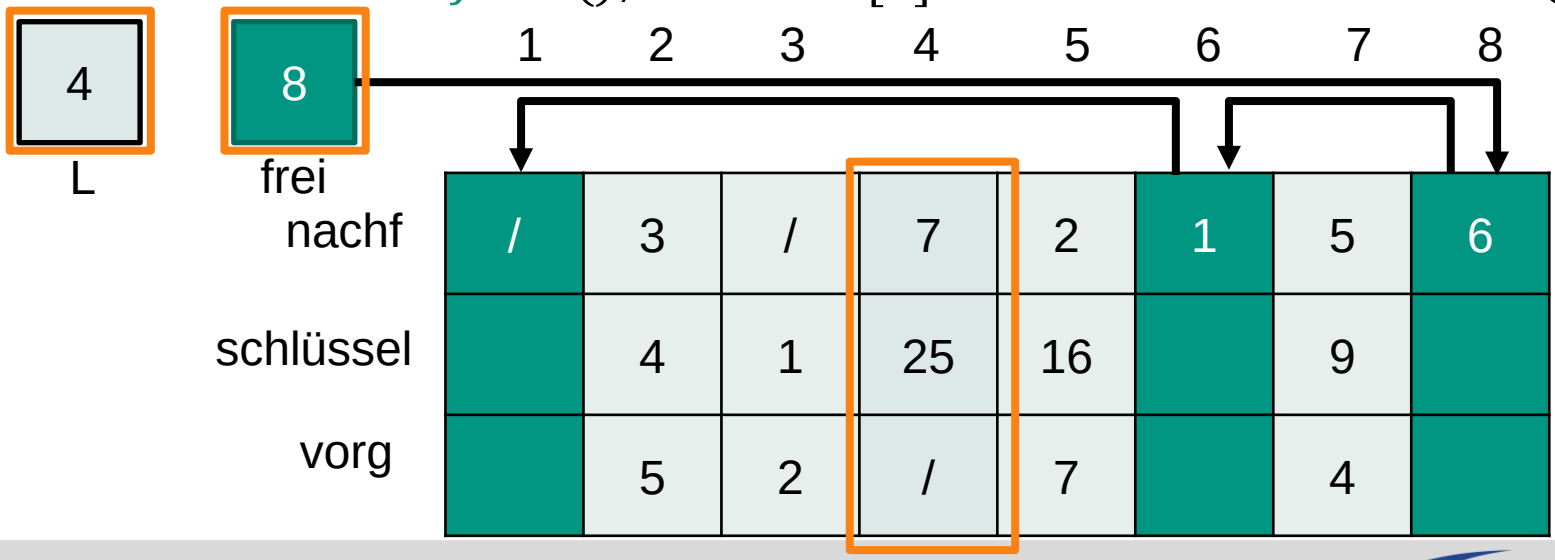
Kopfindex der Freiliste aktualisieren

- Beide Operationen haben einen Zeitaufwand von $O(1)$

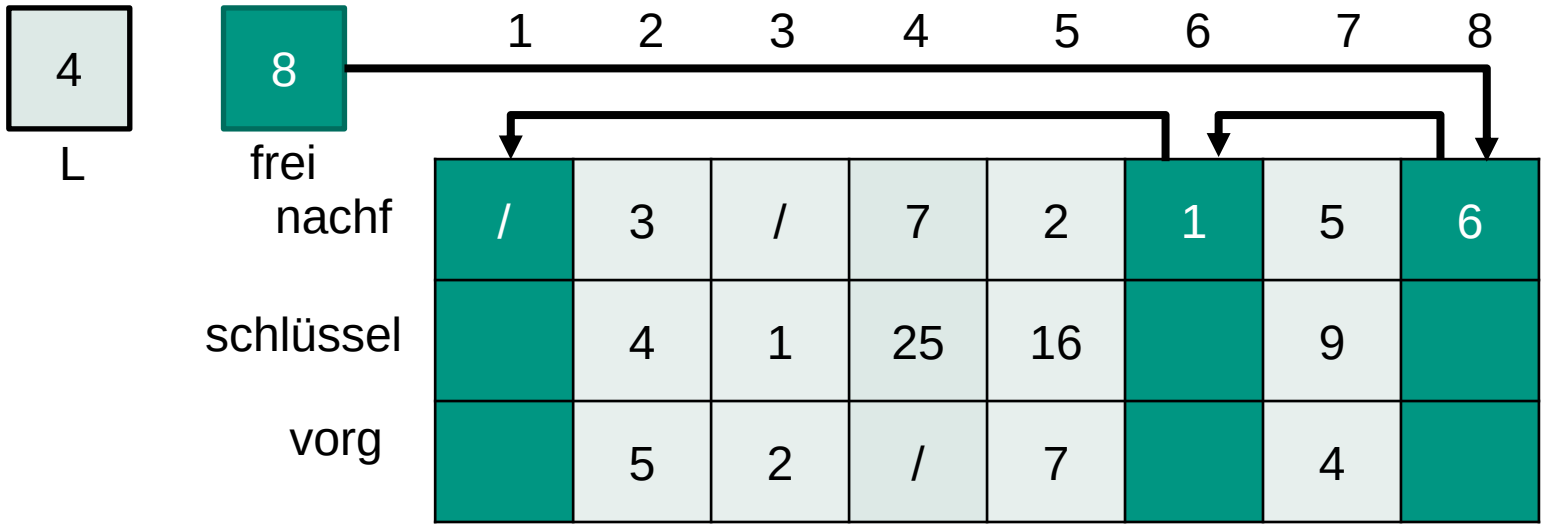
Beispiel für Speicherverwaltung (I)



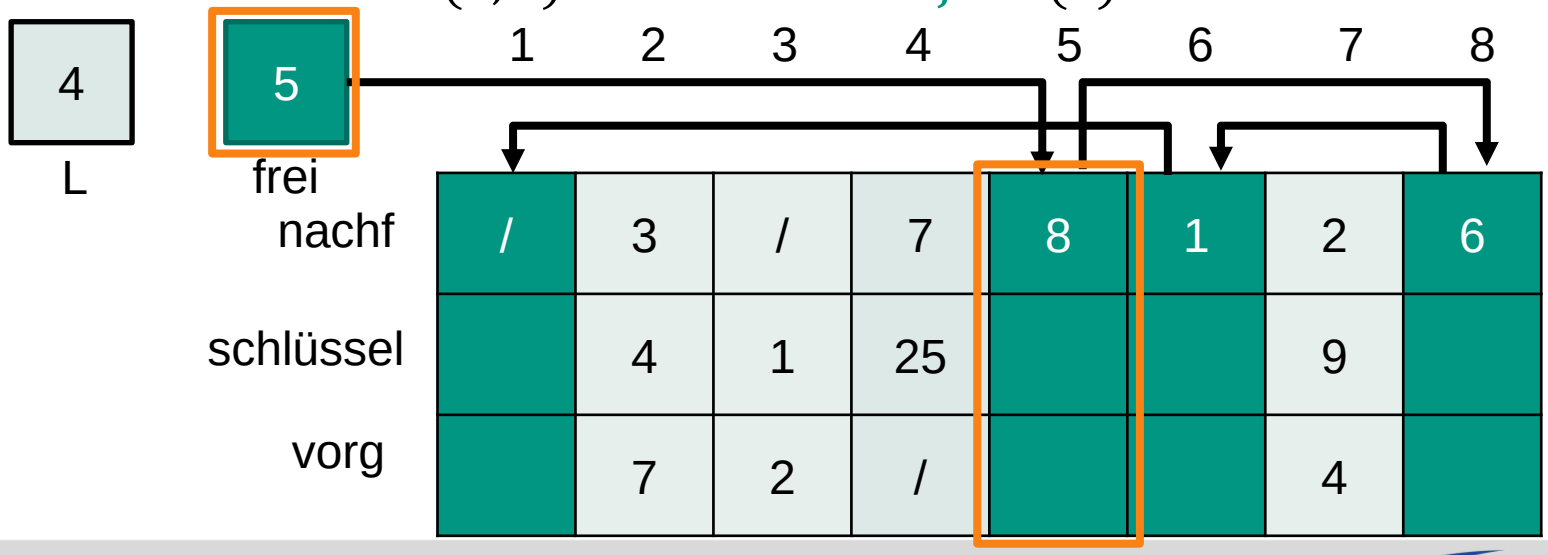
- Nach $ALLOCATE-OBJECT()$, $schlüssel[4] = 25$ und $LIST-INSERT(L, 4)$



Beispiel für Speicherverwaltung (II)



- Nach $LIST - DELETE(L, 5)$ und $\text{FREE} - \text{OBJECT}(5)$



3.4 Unbeschränkte Felder

- Oft ist unklar, wie groß ein Feld zur Laufzeit eines Programmes maximal werden wird
  - Beispiel: Zeilenweises Einlesen einer Datei
- Bei unbeschränkten Feldern wird die Feldgröße dynamisch angepasst
  - Element anhängen, aber kein Platz mehr vorhanden
    - Größeres (beschränktes) Feld allokieren und Elemente umkopieren
  - Element löschen und unnötig viel freier Platz im Feld
    - Kleineres (beschränktes) Feld allokieren und Elemente umkopieren
- Geringerer Speicherverbrauch, aber höherer Zeitaufwand
  - Kopieren eines Feldes mit n Elementen dauert $O(n)$
  - Idee: Besser etwas allokierten Speicher ungenutzt lassen und dafür seltener das Feld umkopieren!
- Im Folgenden Beispiel für die Stapeloperationen $PUSH$ und POP

Pseudocode für unbeschränkte Felder

- $REALLOCATE(n)$ allokiert neues Feld der Größe n, kopiert Element vom alten Feld und löscht dieses
- Neues Element x hinzufügen
  - $PUSH(S, x)$

```
1  if S.top == S.size
2      REALLOCATE(2 · S.top)
3  S.top = S.top + 1
4  S[S.top] = x
```

Falls Feld voll belegt ist, wird die Feldgröße verdoppelt

- Letztes Element entnehmen
  - $POP(S)$

```
1  if STACK − EMPTY(S)
2      error „Stapelunterlauf“
3  else
4      x = S[S.top]
5      S.top = S.top − 1
6      if 4 · S.top ≤ S.size
7          REALLOCATE(2 · S.top)
8      return x
```

Falls Feld nur ¼ belegt ist, wird die Feldgröße halbiert

Amortisierte Laufzeitanalyse (I)

- Der Aufruf $REALLOCATE(n)$ benötigt einen Zeitaufwand $O(n)$
  - Wird aber nicht bei jeder POP- bzw. $PUSH$-Operation durchgeführt
  - Zeitaufwand einer Operation ist entweder $O(1)$ oder $O(n)$
- Die amortisierte Laufzeitanalyse betrachtet den gemittelten Aufwand für eine Folge von Operationen im Worst-Case
  - Im Unterscheid zur Betrachtung des Average-Case werden bei der amortisierten Laufzeitanalyse keine Wahrscheinlichkeiten einbezogen
- Behauptung: Die Operationen POP und $PUSH$ für unbeschränkte Felder haben eine amortisierte Laufzeit von lediglich $O(1)$
- Beweis mittels Account-Methode
  - Jeder Operation werden Zeiteinheiten in Form von Token zugewiesen
  - Jede günstige Operation zahlt nicht verbrauchte Token auf ein Guthabenkonto ein
  - Teure Operationen können mit Token von diesem Guthabenkonto bezahlt werden

Amortisierte Laufzeitanalyse (II)

- Wir setzen folgende Kosten an
  - $PUSH$: 3 Token
    - 1 Token wird sofort für das Einfügen eines Elements verbraucht
    - 2 Token werden auf das Guthabenkonto eingezahlt
  - POP: 2 Token
    - 1 Token wird sofort für das Entfernen des Elements verbraucht
    - 1 Token wird auf das Guthabenkonto eingezahlt
- Falls das Feld n Elemente enthält und durch Aufruf von $REALLOCATE(2n)$ verkleinert oder vergrößert wird, müssen n Elemente kopiert werden
  - Für diesen Fall müssen n Token auf dem Guthabenkonto vorhanden sein

Amortisierte Laufzeitanalyse (III)

- Anfangszustand nach $REALLOCATE$
  - n Elemente, Feldgröße $m = 2n$, kein Guthaben
- Betrachtung einer Sequenz von $PUSH$-Operationen
  - Neuer $REALLOCATE$ nach n $PUSH$-Operationen
    - Guthaben durch n $PUSH$-Operationen: $2n$ Token
    - Kosten für $REALLOCATE$ (Kopieren von $2n$ Elementen): $2n$ Token
    - → Ausreichend Guthaben vorhanden
- Betrachtung einer Sequenz von POP-Operationen
  - Neuer $REALLOCATE$ nach $\mathrm{n}/2$ POP-Operationen
    - Guthaben nach $n/2$ POP-Operationen: $n/2$ Token
    - Kosten für $REALLOCATE$ (Kopieren von $n/2$ Elementen): $n/2$ Token
    - → Ausreichend Guthaben vorhanden
- Der amortisierte Zeitaufwand ist somit $O(1)$

3.5 Vergleich der Datenstrukturen

Operation	Liste (doppelt)	Liste (einfach)	Stapel	Warte-schlange
SEARCH	$O(n)$	$O(n)$	–	–
INSERT (am Anfang)	$O(1)$	$O(1)$	–	–
DELETE	$O(1)$	$O(n)$	–	–
PUSH/POP	–	–	$O(1)$	–
ENQUEUE/ DEQUEUE	–	–	–	$O(1)$
EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$

- Nicht jede Datenstrukturen bietet alle Operationen
- Unterschiede im Zeitaufwand je nach Datenstruktur
  → Abhängig vom Anwendungsfall passende Struktur wählen!

3.6 Zusammenfassung

- Elementare Datenstrukturen für dynamische Mengen
  - Stapel
    - Letztes hinzugefügtes Element kann entfernt werden
  - Warteschlange
    - Zuerst hinzugefügtes Element kann entfernt werden
  - Verkettete Listen
    - Flexible Datenstruktur - bietet alle wesentlichen Operationen
    - Doppelt oder einfach verkettet
    - Wächterelement ermöglicht Vereinfachung
- Verkettete Liste kann ohne Zeiger mit einfachem Feld realisiert werden
  - Speicherverwaltung durch Freiliste
- Datenstrukturen unterscheiden sich im Aufwand für Operationen

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3

Karlsruher Institut für Technologie

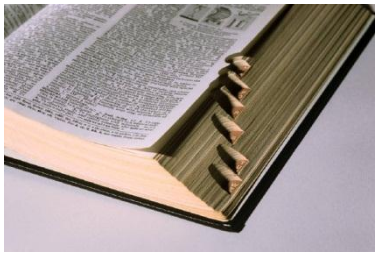
Vorlesung Algorithmen I
Kapitel 4 – Hashing

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

4. Hashing

4.0 Motivation

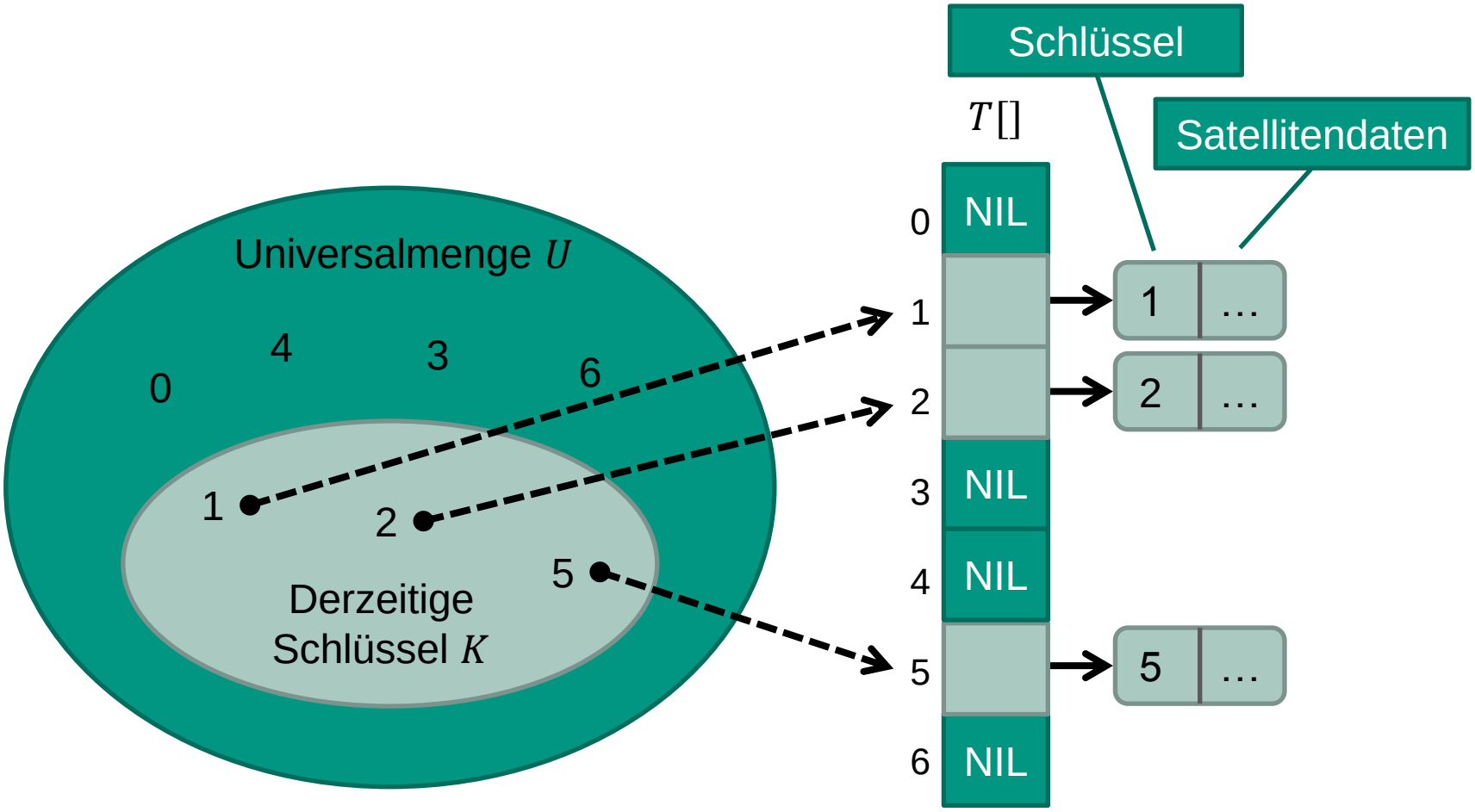


- Anwendungen benötigen oft nur folgende Wörterbuchoperationen auf dynamischen Mengen
  - $INSERT$, $SEARCH$ und $DELETE$
- Beispielanwendung: Compiler für Programmiersprache
  - Verwaltet eine Symboltabelle (= Wörterbuch), in der Schlüsselworte der Programmiersprache als Schlüssel hinterlegt sind
- Hashtabellen bieten sehr effiziente Datenstruktur für diese Operationen
  - Unter guten Bedingungen nur Laufzeit von $O(1)$ im Average-Case
  - Im Worst-Case für $SEARCH$ jedoch $\Theta(n)$ wie bei verketteter Liste
- Können als Erweiterung einfacher Felder betrachtet werden
  - Einfache Felder erlauben die direkte Adressierung einer beliebigen Position in $O(1)$

4.1 Adresstabellen mit direktem Zugriff (I)

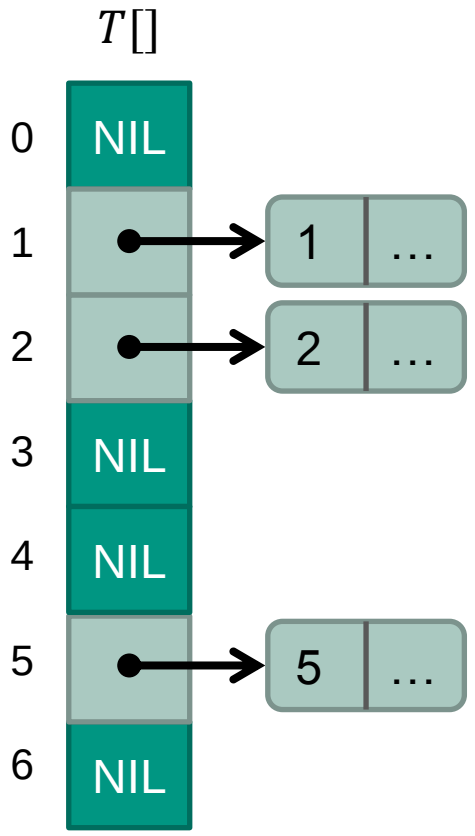
- Ein Element x einer dynamischen Menge besteht aus
  - Schlüssel ($x.schlüssel$)
  - Optionalen Satellitendaten
- Direkte Adressierung geeignet, falls Universalmenge U aller Schlüssel klein ist
  - $U = \{0, 1, \ldots, m - 1\}$, mit m hinreichend klein
  - Annahme: Keine zwei Elemente mit gleichem Schlüssel
- Darstellung der dynamischen Menge als Feld $T[0..m-1]$
  - Feld wird als Adresstabelle mit direktem Zugriff bezeichnet
  - Jede Position (*Slot*) entspricht einem Schlüssel der Universalmenge U
  - $T[k]$ zeigt auf Element mit Schlüssel k
    - Falls Element nicht existiert, ist $T[k] = NIL$

Adresstabellen mit direktem Zugriff (II)



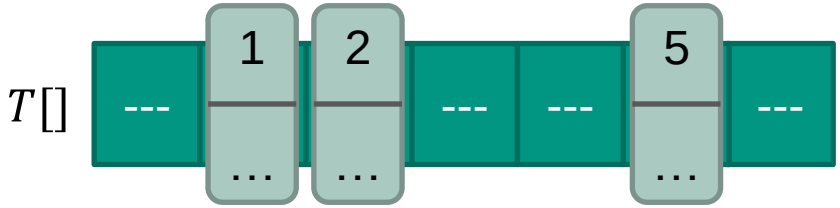
Adresstabellen mit direktem Zugriff (III)

- Suchen eines Elements anhand seines Schlüssels k
  - $DIRECT-ADDRESS-SEARCH(T,k)$
    1 **return** $T[k]$
- Einfügen eines Elements x
  - $DIRECT-ADDRESS-INSERT(T,x)$
    1 $T[x.schlüssel] = x$
- Löschen eines Elements x
  - $DIRECT-ADDRESS-DELETE(T,x)$
    1 $T[x.schlüssel] = NIL$
- Jede Operation benötigt nur Zeit $O(1)$

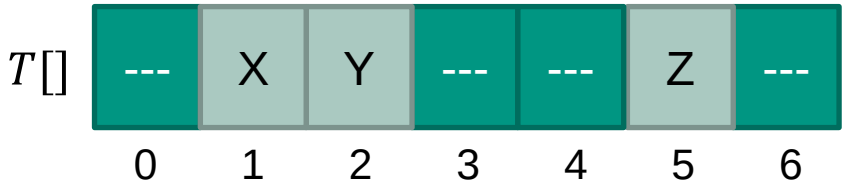


Direkte Adresstabellen ohne Zeiger

- Elemente können auch direkt im Feld gespeichert werden
  - Bedingung: Gleiche Länge für alle Elemente



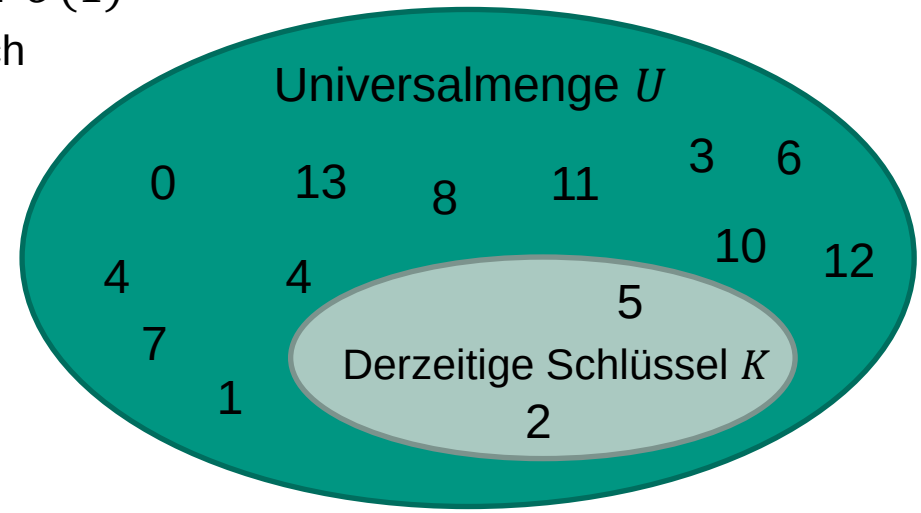
- Zudem kann auf das Speichern des Schlüssels verzichtet werden
  → Herleitung des Schlüssel aus Slot-Index möglich



  - Falls Slot nicht belegt: Markierung durch spezielles Satellitendatum

4.2 Hashtabellen

- Direkte Adressierung oft unpraktikabel
  - Falls Universalmenge U groß ist, steht oft nicht genug Speicher für ein Feld der Länge $|U|$ zur Verfügung
  - Falls Menge der tatsächlich gespeicherten Schlüssel K deutlich kleiner als U, wird zudem Speicher verschwendet
- Hashtabellen bieten in diesen Fällen eine effiziente Datenstruktur
  - Speicheraufwand nur $\Theta(|K|)$ statt $\Theta(|U|)$
  - Zeitaufwand im Average Case nur $O(1)$
    - Direkte Adressierung bietet jedoch O(1) auch im Worst-Case



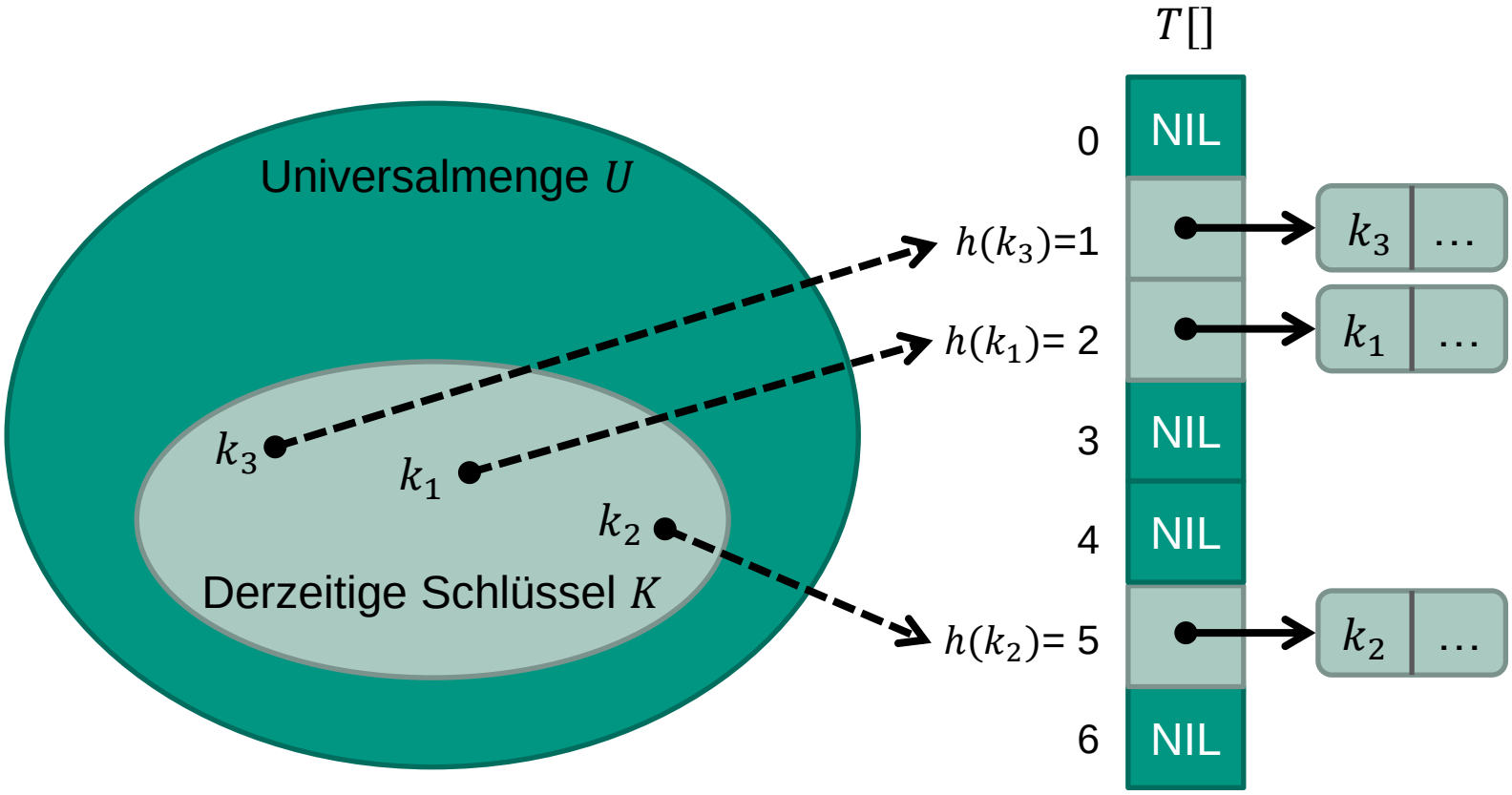
4.2.1 Grundprinzip von Hashtabellen

- Ähnlich der direkten Adresstabelle, aber Slot wird durch Funktion $h(k)$ anstelle von k bestimmt
- h ist eine sogenannte Hashfunktion
  - Bildet die Universalmenge U in die Menge der Slots einer Hashtabelle $T[0..m-1]$ ab:

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

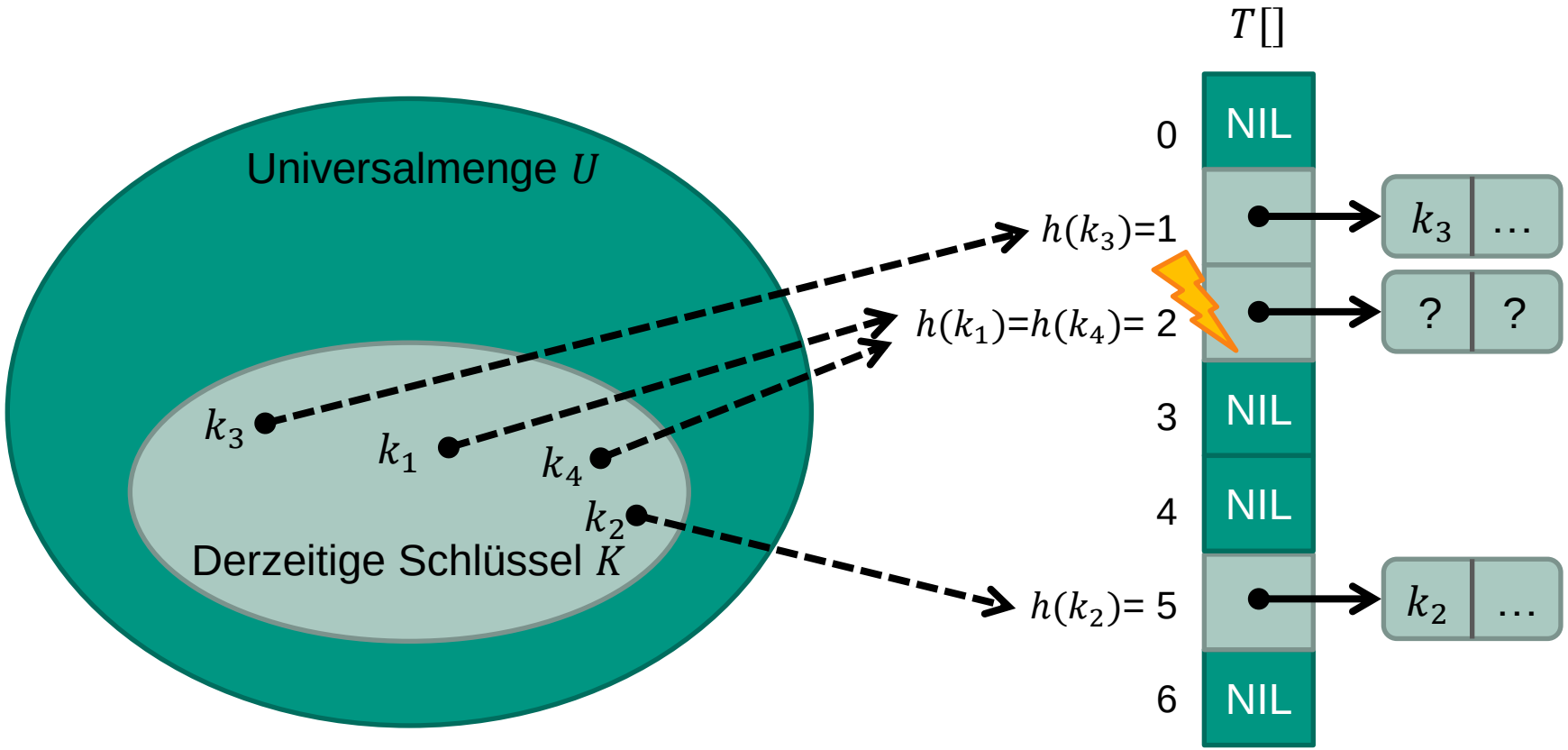
  - Dabei gilt üblicherweise $m \ll |U|$
  - $h(k)$ ist der sogenannte Hashwert von k

Grundprinzip von Hashtabellen (II)



Problem: Kollisionen

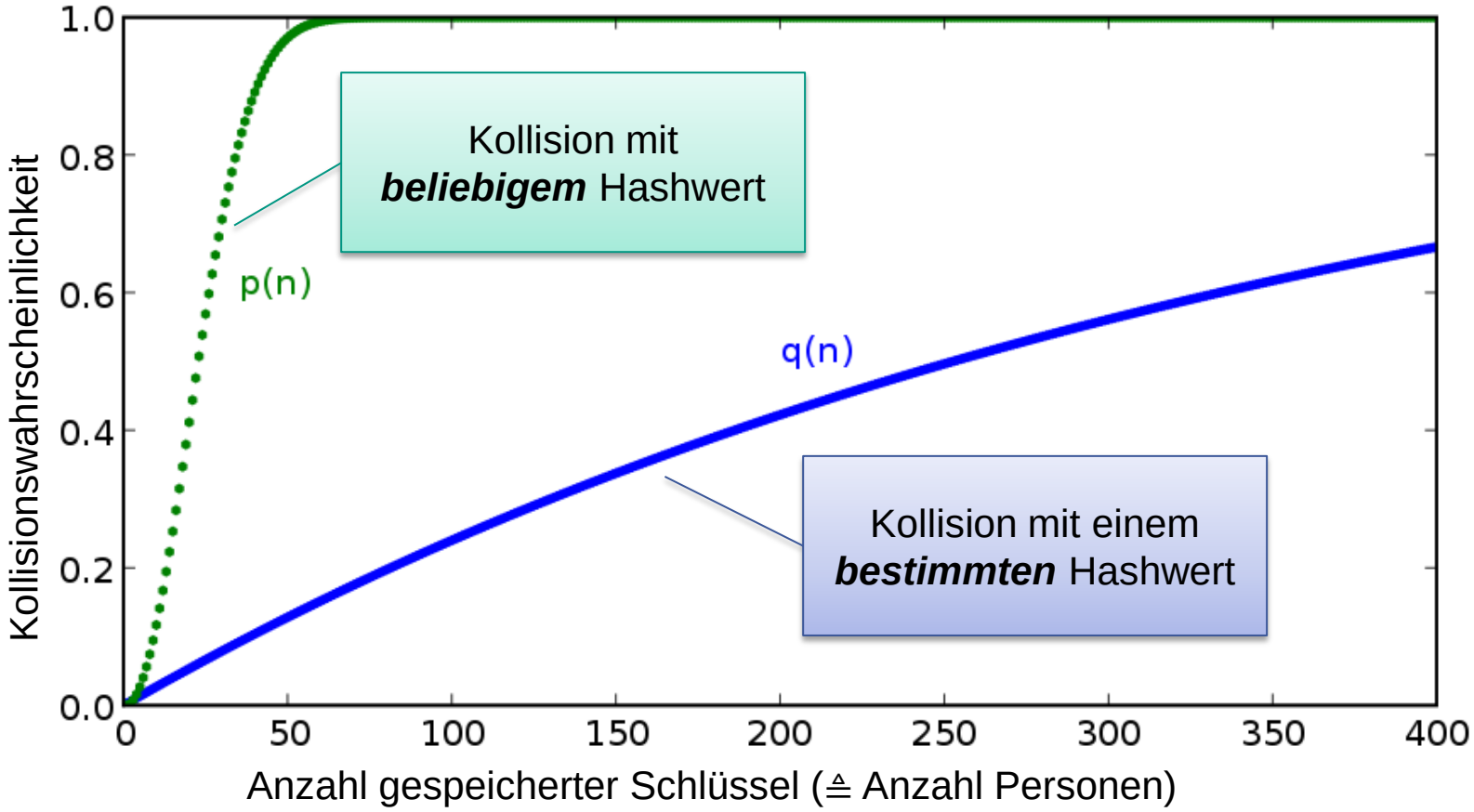
- Zwei Schlüssel können auf den gleichen Slot abgebildet werden
  - Beispiel für eine Kollision: $h(k_1) = h(k_4) = 2$



Wie wahrscheinlich ist eine Kollision?

- Beispiel Geburtstagsparadoxon: Wie hoch ist die Wahrscheinlichkeit, dass von 23 Personen mindestens 2 am gleichen Tag Geburtstag haben?
  - Über 50%!
- Mathematische Herleitung
  - p(1. Person anderer Geburtstag als 2. Person) = $\frac{364}{365}$
  - p(3. Person anderer Geburtstag als die beiden Ersten) = $\frac{363}{365}$
  - …
  - Wegen $\frac{364}{365} \cdot \frac{363}{365} \cdots \frac{343}{365} < 0{,}5$ ist es ab 23 Personen wahrscheinlicher, dass zwei am selben Tag Geburtstag haben, als dass alle an verschiedenen Tagen Geburtstag haben
- Bezogen auf Hashing: Für $m = 365$ Slots und nur $|K| = 23$ gespeicherte Schlüssel über 50% Wahrscheinlichkeit für Kollision!

Geburtstagsparadoxon: Grafische Darstellung



- Wahrscheinlichkeit für eine Kollision in Hashtabelle mit $m = 365$ Slots

Wie wahrscheinlich ist eine Kollision? (III)

- Erwartungswert für Anzahl Kollisionen für Hashtabelle mit n Elementen und m Slots
  - Zufallsvariable $X_{ij} = \begin{cases} 1, & \text{falls } h(i) = h(j) \\ 0, & \text{sonst} \end{cases}$
  - Für $i \neq j$ gilt $E(X_{ij}) = \frac{1}{m}$
  - Sei X die Gesamtanzahl an Kollisionen. Dann gilt

$$E(X) = E\left(\sum_{i<j} X_{ij}\right) = \sum_{i<j} E(X_{ij}) = \sum_{i<j} \frac{1}{m} = \binom{n}{2} \cdot \frac{1}{m}$$

  - Beispiel: Ab 28 Personen überschreitet die zu erwartende Anzahl gleicher Geburtstage den Wert 1 (d.h. $n = 28, m = 365 \Rightarrow E(X) \geq 1$)

Herausforderungen Hashing

- Wahl einer „guten“ Hashfunktion $h(k)$
  - Determinismus
    - Eingabe k liefert immer gleiche Ausgabe $h(k)$
  - Anzahl an entstehenden Kollisionen durch geeignete Streuung ist möglichst gering
  - Beispiele für guten Hashfunktionen kommen später in Kapitel 4.3
- Auflösung von Kollisionen
  - Wegen $|U| > m$ gibt es auch bei einer guten Hashfunktion mindestens zwei Schlüssel mit gleichem Hashwert

    Kollisionen können also immer auftreten und müssen aufgelöst werden!

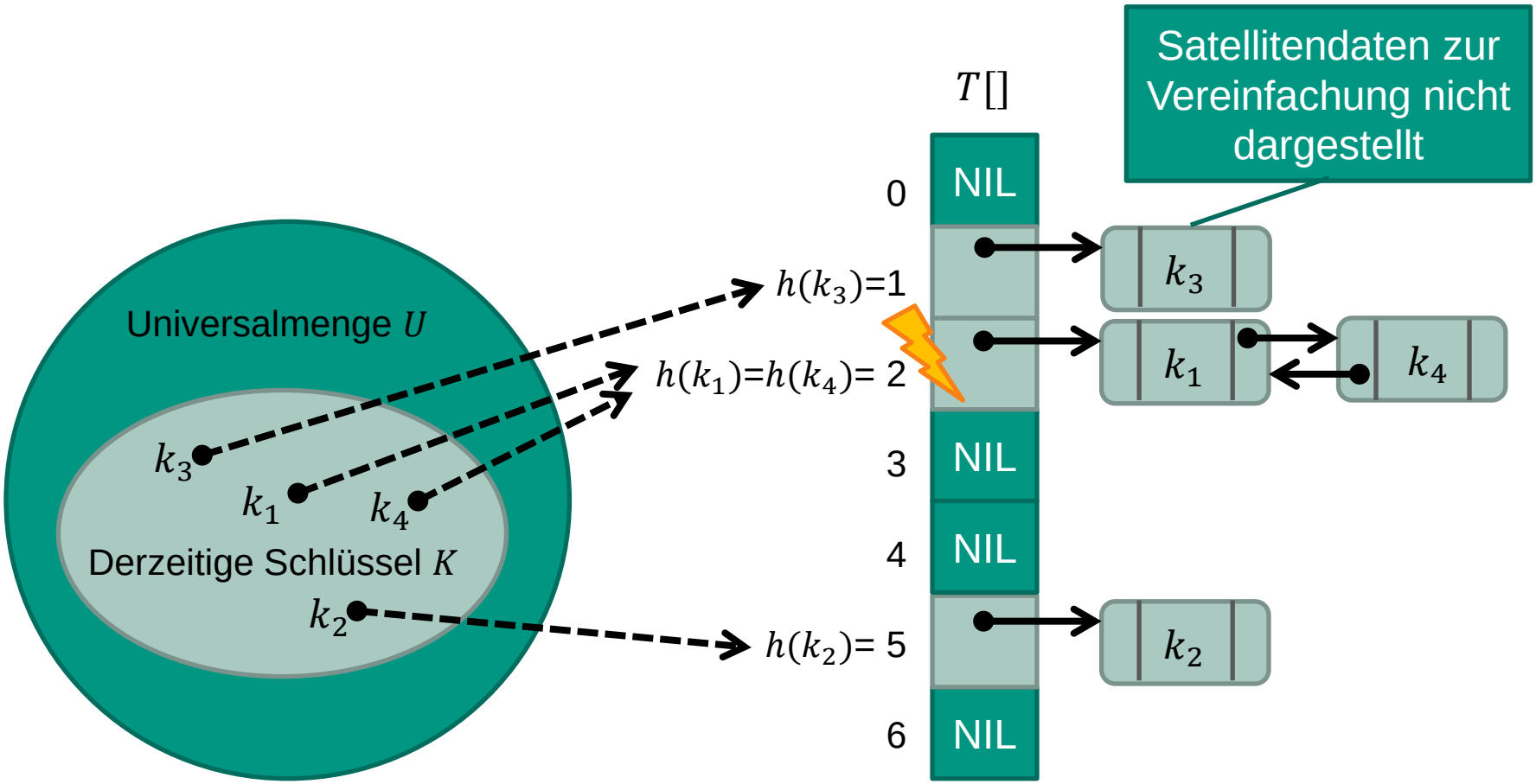
  - Verschiedene Varianten für Kollisionsauflösung
    - Kollisionsauflösung durch Verkettung (kommt im Anschluss)
    - Offene Adressierung (später in Kapitel 4.4)

4.2.2 Kollisionsauflösung durch Verkettung (I)

- Einfache Kollisionsauflösung durch verkettete Liste möglich
- Slot j enthält Zeiger auf Kopf einer verketteten Liste L_j
- Liste L_j enthält alle Elemente, deren Schlüssel den gleichen Hashwert j haben
- Falls es kein Element mit Hashwert j gibt, enthält der Slot j den Wert NIL
- Liste kann einfach oder doppelt verkettet sein
  - Mit doppelt verketteten Listen können Objekte schneller gelöscht werden

Kollisionsauflösung durch Verkettung (II)

- Beispiel für Kollisionsauflösung mit doppelt verketteter Liste
  - Kollision für die Schlüssel k_1 und k_4 mit $h(k_1) = h(k_4) = 2$



Operationen auf Hashtabellen mit Verkettung

- $CHAINED-HASH-INSERT(T, x)$
  1 füge x am Anfang der Liste $T[h(x.schlüssel)]$ ein

- $CHAINED-HASH-SEARCH(T, k)$
  1 suche in der Liste $T[h(k)]$ nach einem Element mit Schlüssel k

- $CHAINED-HASH-DELETE(T, x)$
  1 entferne x aus der Liste $T[h(x.schlüssel)]$

- Einfügen und Entfernen von Elementen ist in $O(1)$ möglich
  - Beim Einfügen wird vorausgesetzt, dass das Element nicht bereits in der Liste vorhanden ist
  - Zum Entfernen in $O(1)$ muss die Liste doppelt verkettet sein und bereits ein Zeiger auf das Element x bekannt sein
    → Anderenfalls zusätzlicher Aufwand fürs Suchen

4.2.3 Komplexitätsanalyse für das Suchen in Hashtabellen mit Verkettung

- Gegeben: Hashtabelle mit m Slots, die n Elemente speichert
- Annahme: Berechnung des Hashwerts $h(k)$ ist in $O(1)$ möglich
- Im Worst-Case werden alle n Schlüssel auf den gleichen Slot abgebildet
  - Ergibt eine verkettete Liste der Länge n
  - Daraus folgt im Worst-Case einen Zeitaufwand von $\Theta(n)$
- Im Best-Case wird in jeden Slot maximal ein Schlüssel abgelegt
  - Da immer auf das erste Element der verketteten Liste zugegriffen wird, beträgt im Best-Case der Zeitaufwand $0(1)$

Aufwand für das Suchen im Average-Case (I)

- Der Average-Case ist aufwändiger zu bestimmen
  - Hängt vom Belegungsfaktor $\alpha = n/m$ ab
    - m ist die Anzahl von Slots
    - n ist die Anzahl gespeicherter Element
    - α ist somit die mittlere Anzahl an Elementen in einer verketten Liste der Hashtabelle
  - Hängt davon ab, wie gut Hashfunktion h die Schlüssel auf die Slots verteilt
  - Annahme
    - Jedes Element wird mit gleicher Wahrscheinlichkeit auf einen der Slots verteilt
    - Die Verteilung eines Elements ist unabhängig von den anderen Elementen

→ Einfaches gleichmäßiges Hashing

Aufwand für das Suchen im Average-Case (II)

- Länge der verketteten Liste in Slot j wird mit n_j bezeichnet
  - Es gilt: $n = n_0 + n_1 + \cdots + n_{m-1}$
  - Der Mittelwert von n_j ist also $E[n_j] = \alpha = n/m$
- Zeit für das Suchen eines Elements mit Schlüssel k hängt linear von der Länge $n_{h(k)}$ der Liste in Slot $h(k)$ ab
- Fallunterscheidung
  - Element ist nicht in der Liste enthalten
  - Element ist in der Liste enthalten

Aufwand für das Suchen im Average-Case (IV)

- Element ist nicht in der Liste enthalten
  - Zeit für die erfolglose Suche nach einem Schlüssel k ist gleich der Zeit für die Suche bis ans Ende der Liste in Slot $h(k)$
  - Zahl der erwarteten Elemente in der Liste ist $E[n_{h(k)}] = \alpha$
  - Überprüfen von α Elementen benötigt $\Theta(\alpha)$
  - Gesamtaufwand inklusive Hashwert berechnen ist somit $\Theta(1 + \alpha)$

Aufwand für das Suchen im Average-Case (III)

- Element ist in der Liste enthalten
  - Sei x_i das i-te Element, das in die Hashtabelle eingefügt wurde
  - Zufallsvariable $X_{ij} = \begin{cases} 1, & \text{falls } h(i) = h(j) \\ 0, & \text{sonst} \end{cases}$
  - Für $i \neq j$ gilt $E(X_{ij}) = \frac{1}{m}$

Einfaches gleichmäßiges Hashing

  - Mittlere Anzahl zu prüfender Elemente, um x_i zu finden ist
    - Mittlere Anzahl an Elementen, die in Slot $h(x_i.schlüssel))$ eingefügt wurden, nachdem x_i eingefügt wurde
    - +1 (für das Element x_i selbst)
  - Erwartungswert für die Anzahl zu prüfender Elemente ist daher

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

Aufwand für das Suchen im Average-Case (IV)

- Element ist in der Liste enthalten
  - Erwartungswert für die Anzahl zur prüfender Elemente ist daher

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

Linearität des Erwartungswertes

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

Einfaches gleichmäßiges Hashing

$$=\cdots=1+\frac{n-1}{2m}=1+\frac{\alpha}{2}+\frac{\alpha}{2m}$$

  - Gesamtaufwand ist also ebenfalls $\Theta(1+\alpha)$ und hängt somit nur vom Belegungsfaktor ab

…und was bedeutet das nun?

- Falls Anzahl der Slots m mindestens proportional zur Anzahl der gespeicherten Elemente n ist, dann ist $n = O(m)$
- Daher gilt

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

- Also Suchen im Average-Case auch in $O(1)$
- Fazit: Die Wörterbuchoperationen $INSERT$, $DELETE$ und $SEARCH$ können im Average-Case in $O(1)$ durchgeführt werden, falls
  - die Hashfunktion einfaches gleichmäßiges Hashing bietet
  - eine doppelt verkettete Liste zur Kollisionsauflösung verwendet wird
  - die Anzahl der Slots m proportional zur Anzahl der gespeicherten Elemente n ist

4.3 Hashfunktionen

- Was macht eine gute Hashfunktion aus?
  - Möglichst einfach und schnell berechenbar
  - Bietet (näherungsweise) einfaches gleichmäßiges Hashing
    - Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit auf einen der m Slots abgebildet, unabhängig davon wie ein anderer Schlüssel abgebildet wurde
  - Falls Schlüssel unabhängige und gleichverteilte reelle Zahlen aus dem Intervall $0 \leq k < 1$ sind, ist folgendes eines gute Hashfunktion

$$h(k) = \lfloor k \cdot m \rfloor$$

  - Problem: Oft ist die Verteilung der Schlüssel unbekannt
  - In der Praxis: Heuristische Verfahren zur Erzeugung der Hashfunktion, die Informationen über die Schlüsselverteilung berücksichtigen
  - Manche Anwendungen fordern mehr als einfaches gleichmäßiges Hashing
    - Bei linearem Sondieren (siehe Kapitel 4.4) sollen nahe beieinanderliegende Schlüssel möglichst weit auseinanderliegende Hashwerte haben

Schlüssel als natürliche Zahlen interpretieren

- Die meisten Hashfunktionen setzen als Universalmenge die Menge der natürlichen Zahlen $\mathbb{N} = \{0,1,2, \ldots\}$ voraus
- Falls Schlüssel keine natürlichen Zahlen sind, müssen diese zunächst umgewandelt werden

- Umwandlung von Zeichenketten in natürliche Zahlen
  - Zeichenweise Umwandlung anhand des ASCII-Codes der Zeichen
  - Jedes Zeichen wird als Zahl zur Basis 128 interpretiert
  - Beispiel für Zeichenkette *„pt“*
    - Im ASCII-Code ist $p = 112$ und $t = 116$
    - „pt“ kann also als das Paar (112, 116) interpretiert werden
    - Umwandlung von „pt“ in natürliche Zahl: $(112 \cdot 128) + 116 = 14452$

- Im Folgenden werden nur natürliche Zahlen als Schlüssel betrachtet

Überblick geeigneter Hashfunktionen

- Kapitel 4.3.1: Divisionsmethode
  - Einfache Variante für geeignete Hashfunktion
- Kapitel 4.3.2: Multiplikationsmethode
  - Verbesserte Variante, bei der die Wahl geeigneter Parameter weniger kritisch ist
- Kapitel 4.3.3: Universelles Hashing
  - Verfahren um zu verhindern, dass ein Angreifer gezielt solche Schlüssel auswählen kann, die alle auf den selben Slot abgebildet werden

4.3.1 Divisionsmethode (I)

- Bei der Divisionsmethode wird ein Schlüssel k auf einen von m möglichen Hashwerten abgebildet, indem k durch m geteilt wird
- Der Rest der Division ergibt den Hashwert
- Die Hashfunktion lautet also $h(k) = k \, mod \, m$
- Beispiel
  - Hashtabelle mit $m = 12$ Slots
  - Schlüssel $k = 100$
  - Dann ist der Hashwert $h(k) = 100 \, mod \, 12 = 4$

Divisionsmethode (II)

- Der Parameter m der Hashfunktion $h(k) = k \bmod m$ muss geeignet gewählt werden
- Zum Beispiel ist $m = 2^p$ eine schlechte Wahl, da $h(k)$ dann nur von den p niederwertigsten Bits von k abhängt
- Gute Wahl für m: Primzahl, die nicht nahe an einer Potenz von 2 liegt

- Beispiel
  - Sollen z.B. $n = 2000$ Zeichenketten abgelegt werden und im Mittel $\alpha = 3$ Elemente pro Liste gespeichert werden, ist $m = 701$ ein geeigneter Wert
    - m ist Primzahl
    - m ist nicht nahe an einer Potenz von 2
    - $\mathrm{n} \approx \alpha \cdot m = 3 \cdot 701 = 2103$

4.3.2 Multiplikationsmethode

- Die Multiplikationsmethode zur Erzeugung von Hashfunktionen arbeitet in zwei Schritten
  1. Multiplikation des Schlüssels k mit einer Konstanten A mit $0 < A < 1$
  2. Multiplikation des gebrochenen Rests der vorherigen Multiplikation mit m und Abrunden des Resultats ergibt den Hashwert
- Die Hashfunktion lautet also $h(k) = \lfloor m(k \cdot A \bmod 1) \rfloor$
- Im Gegensatz zur Divisionsmethode ist die Wahl von m nicht kritisch
- Wird $m = 2^p$ gewählt, kann der Hashwert besonders effizient berechnet werden
- Die Konstante A kann im Prinzip frei gewählt werden, bestimmte Werte führen aber zu besseren Ergebnissen
  - Knuth schlägt zum Beispiel $A \approx \frac{(\sqrt{5}-1)}{2} \approx 0{,}6180339887$ vor

[Knut98]

4.3.3 Universelles Hashing (I)

- Bei einer festen Hashfunktion könnte ein Angreifer gezielt n Schlüssel auswählen, die alle auf den selben Slot abgebildet werden
  - Ergibt schlechte Laufzeit von $\Theta(n)$ im Average-Case
- Abhilfe: Hashfunktion wird unabhängig von den Schlüsseln zufällig ausgewählt (Universelles Hashing)
  - Durch zufällige Auswahl wird garantiert, dass nicht eine spezielle Eingabe jedes Mal das schlechteste Verhalten hervorruft
  - Algorithmus kann sich durch zufällige Auswahl bei jeder Durchführung unterschiedlich verhalten
  - Die Menge von möglichen Hashfunktionen $\mathcal{H}$ heisst universell, wenn für jedes Schlüsselpaar $k, l \in U, k \neq l \, mit \, |U| = m$ die Anzahl der Hashfunktionen mit $h(k) = h(l)$ höchstens gleich $|\mathcal{H}|/m$ ist
    - Anders gesagt: Für eine zufällig gewählte Hashfunktion ist die Wahrscheinlichkeit für eine Kollision zwischen k und l nicht größer als $1/m$

Universelles Hashing (II)

- Theorem (Beweis siehe  [Corm10])
  - Annahme: Die zufällig aus der universellen Menge $\mathcal{H}$ ausgewählte Hashfunktion h wird verwendet, um n Schlüssel in eine Tabelle T der Größe m abzubilden (mit Kollisionsauflösung durch Verkettung)
  - Wenn Schlüssel k nicht in T enthalten ist, dann ist die erwartete Länge $E[n_{h(k)}]$ der Liste, auf die k abgebildet wird, höchstens gleich dem Belegungsfaktor $\alpha = n/m$
  - Wenn Schlüssel k in T enthalten ist, dann ist die erwartete Länge $E[n_{h(k)}]$ höchstens $1 + \alpha$

Universelles Hashing (III)

- Daraus folgt: Erwartete Laufzeit für jede Folge von n $INSERT$-, $SEARCH$- und $DELETE$-Operationen in eine leere Tabelle mit m Slots ist $\Theta(n)$, sofern davon nur $O(m)$ $INSERT$-Operationen sind
- Beweis
  - $O(m)$ $INSERT$-Operationen → $\alpha = O(1)$
  - $INSERT$ und $DELETE$ benötigen $O(1)$, $SEARCH$ nach Theorem ebenfalls $O(1)$
  - Damit erwartete Zeit für gesamte Folge $\Theta(n)$

4.4 Offene Adressierung

- Alternative zu Kollisionsauflösung mit verketteten Listen
  - Bei offener Adressierung werden alle Element direkt in der Hashtabelle gespeichert
- Bei der Suche nach einem Element werden systematisch mehrere Tabellenslots überprüft, bis das Element gefunden wurde
- Da alle Elemente direkt in der Tabelle gespeichert werden, können bei m Slots maximal m Element gespeichert werden (d.h. $\alpha \leq 1$)
- Gegenüber der Kollisionsauflösung mit verketteten Listen wird der Speicherplatz für die Zeigerverwaltung eingespart
- Zum Einfügen muss die Hashtabelle sukzessive sondiert werden, um einen leeren Slot zu finden

Sondieren bei offener Adressierung

- Slots werden nicht in fester Reihenfolge $0,1,\dots,m-1$ sondiert, da sonst die Suchzeit $\Theta(n)$ betragen würde
- Stattdessen wird die Reihenfolge der Slots abhängig vom einzufügenden Schlüssel bestimmt
- An die Hashfunktion h wird zusätzlich eine Sondierungszahl übergeben

$$h : U \times \{0,1,\dots,m-1\} \rightarrow \{0,1,\dots,m-1\}$$

- Die Sondierungssequenz $< h(k,0), h(k,1), \dots, h(k,m-1) >$ muss dabei für jeden Schlüssel k eine Permutation von $< 0,1,\dots,m-1 >$ sein (d.h. jeder Slot wird potentiell für die Ablage des Schlüssels berücksichtigt)

Einfügen mit offener Adressierung

- Die Methode $HASH - INSERT(T, k)$ fügt den Schlüssel k in die Hashtabelle T ein und liefert die Nummer des entsprechenden Slots zurück

- $HASH - INSERT(T, k)$

```
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error „Überlauf“
```

Sondierungszahl (mit 0 anfangen)

Falls Slot noch frei, Schlüssel einfügen

Sonst Sondierungszahl inkrementieren und erneut versuchen

Falls kein Slot mehr frei, Fehler ausgeben

Suchen mit offener Adressierung

- Die Methode $HASH - SEARCH(T, k)$ sucht einen Schlüssel k und liefert die Nummer des Slots zurück oder NIL, falls der Schlüssel nicht gefunden wurde

- $HASH - SEARCH(T, k)$

```
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Sondierungszahl (mit 0 anfangen)

Falls Schlüssel gefunden, Nummer des Slots zurückgeben

Sonst Sondierungszahl inkrementieren und erneut versuchen

Falls auf leeren Slot gestoßen wird oder alles Slots überprüft wurden, wird Suche abgebrochen

Entfernen mit offener Adressierung

- Bei offener Adressierung können wir ein Element nicht einfach löschen, indem wir den Slot mit NIL markieren
  - Sonst würden Schlüssel ggf. nicht gefunden, da die Suche vorzeitig abgebrochen würde
- Lösungsvariante
  - Slots mit gelöschten Elementen werden mit dem speziellen Wert $ENTFERNT$ markiert
  - $HASH-INSERT$ wird so modifiziert, dass diese Slots wie leere Slots behandelt werden und neue Schlüssel abgelegt werden können
  - Nachteil: Die Suchzeit hängt nicht länger vom Belegungsfaktor α ab, sondern wird durch gelöschte Elemente vergrößert!

  → Falls Elemente gelöscht werden sollen, wird daher eher die Kollisionsauflösung durch verkettete Listen verwendet

Gleichmäßiges Hashing

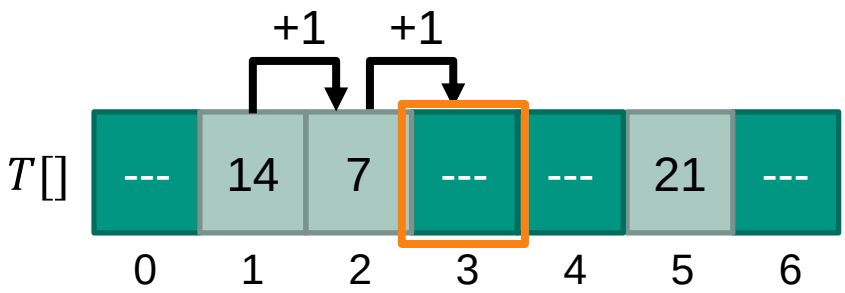
- Unsere Aufwandsabschätzungen setzten gleichmäßiges Hashing voraus
  - Jede der $m!$ Permutation von $< 0,1, \ldots, m-1 >$ ist mit gleicher Wahrscheinlichkeit die Sondierungssequenz eines Schlüssels
- Verallgemeinerung des *einfachen gleichmäßigen Hashings*
  - Hashfunktion erzeugt nicht nur einzelne Zahl, sondern eine ganze Sondierungssequenz
- Gleichmäßiges Hashing ist schwer zu implementieren
- Im Folgenden werden drei Varianten vorgestellt, die für das Sondieren zumindest eingeschränkt geeignet sind
  - Lineares Sondieren
  - Quadratisches Sondieren
  - Doppeltes Hashing
- Achtung: Alle Varianten liefern jedoch maximal m^2 anstelle der geforderten $m!$ Sondierungssequenzen für gleichmäßiges Hashing!

4.4.1 Lineares Sondieren (I)

- Zusätzliche Hilfshashfunktion $h' : U \to \{0,1,\ldots,m-1\}$
- Hashfunktion für lineares Sondieren verwendet die Hilfshashfunktion h'

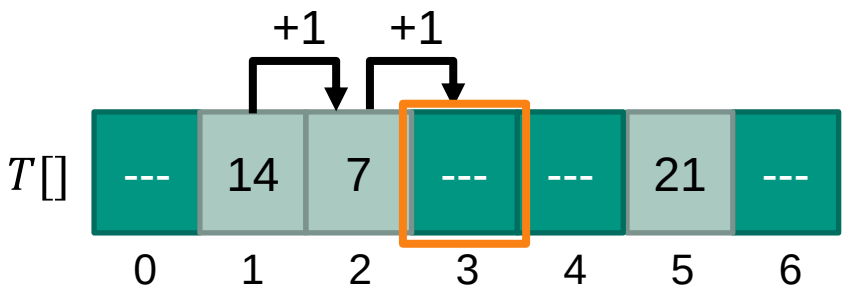
$$h(k,i) = (h'(k) + i) \; mod \; m$$

- Zu einem Schlüssel k wird also zunächst der Slot $h'(k)$ sondiert, danach $h'(k)+1$ usw. bis Slot $m-1$
- Falls der letzte Slot $m-1$ erreicht wurde, wird bei Slot 0 fortgesetzt, bis schließlich $h'(k)-1$ erreicht wird



Lineares Sondieren (II)

- Insgesamt gibt es nur m verschiedene Sondierungssequenzen
- Problem: Primäres Clustern
  - Es bilden sich lange Folgen belegter Slots, wodurch sich die mittlere Suchzeit erhöht
  - Cluster entstehen, weil ein leerer Slot, der auf i besetzte Slots folgt, mit Wahrscheinlichkeit $(i + 1)/m$ als nächstes belegt wird

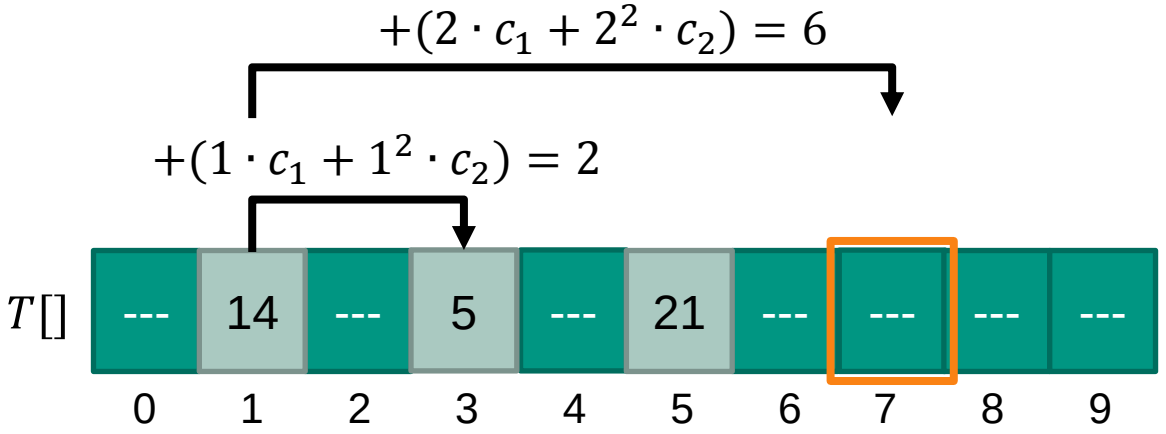


4.4.2 Quadratisches Sondieren (I)

- Quadratisches Sondieren verwendet die folgende Hashfunktion

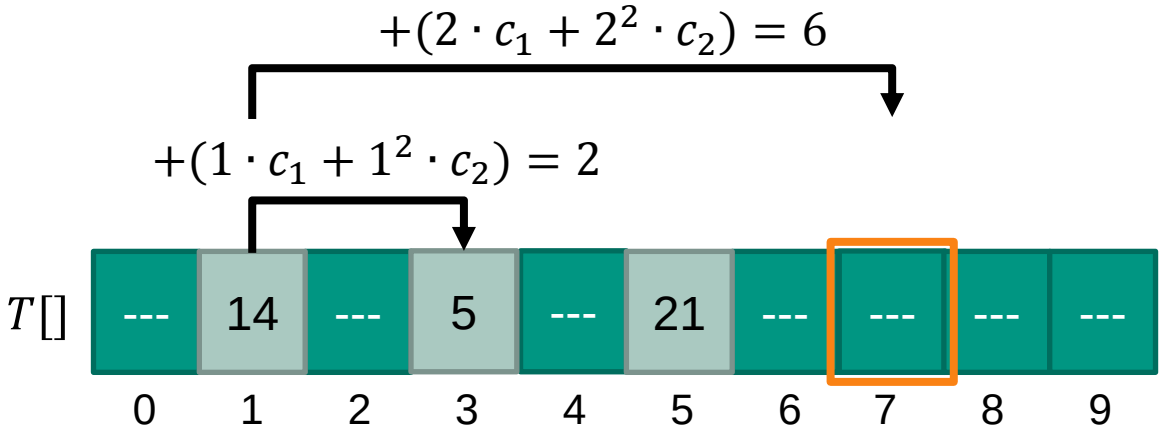
$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \; mod \; m$$

- Wie beim lineares Sondieren wird die Hilfshashfunktion h' verwendet
- Zusätzlich werden die positiven Hilfskonstanten c_1 und c_2 benötigt
- Der erste Sondierungsslot ist wieder $h'(k)$, die weiteren Slots werden jedoch quadratisch abhängig von i verschoben
- Beispiel: $c_1 = 1, c_2 = 1$



Quadratisches Sondieren (II)

- Die Konstanten c_1,c_2 und m müssen für optimales Ergebnis bestimmte Nebenbedingungen erfüllen
- Sind die Anfangsslots gleich, ergeben sich die gleichen Sondierungssequenzen → sekundäres Clustern
- Wie beim linearen Sondieren wird durch $h'(k)$ die gesamte Sondierungssequenz festgelegt und somit existieren nur m verschiedene Sondierungssequenzen



4.4.3 Doppeltes Hashing

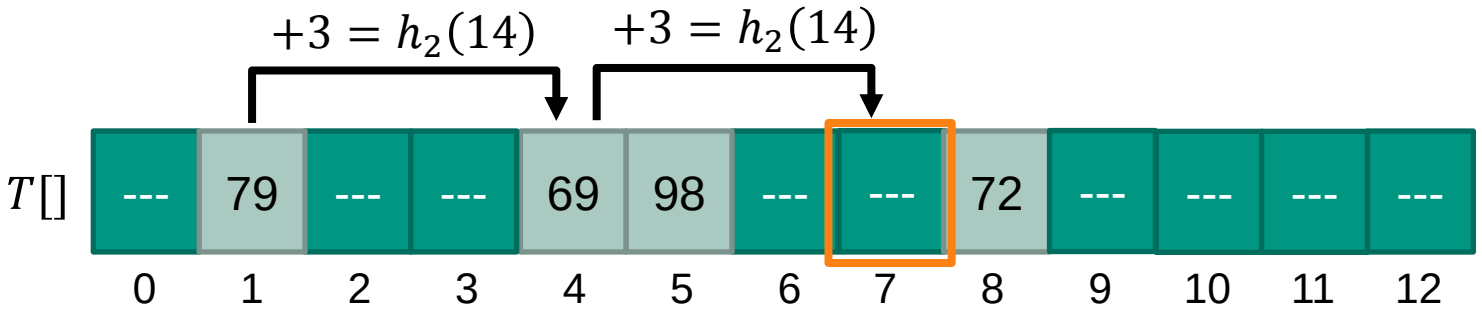
- Doppeltes Hashing ist eines der besten Verfahren für offene Adressierung
- Es wird die folgende Hashfunktion verwendet, wobei zwei Hilfshashfunktionen eingesetzt werden:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \; mod \; m$$

- Im Unterscheid zu linearem oder quadratischem Sondieren ist die Verschiebung vom Schlüssel k abhängig
  - In jedem Sondierungsschritt wird um $h_2(k)$ verschoben
- Der Wert $h_2(k)$ muss teilerfremd zu m sein, damit die gesamte Hashtabelle durchsucht wird
  - Wird für m eine Primzahl gewählt, erfüllen die folgenden Hilfshashfunktionen diese Bedingung
    - $h_1(k) = k \; mod \; m$
    - $h_2(k) = 1 + (k \; mod \; m')$ mit $m' = m - 1$

Beispiel: Doppeltes Hashing

- Hilfshashfunktion $h_1(k) = k \; mod \; 13$
- Hilfshashfunktion $h_2(k) = 1 + (k \; mod \; 12)$
- Einfügen von Schlüssel 14
  - $h_1(14) = 14 \; mod \; 13 \equiv 1$, aber Slot 1 ist bereits belegt
  - Weitere Slots werden im Abstand $h_2(14) = 1 + (14 \; mod \; 12) \equiv 3$ sondiert
  - $i = 1$: Slot 4 ist ebenfalls belegt
  - $i = 2$: Slot 7 ist frei und wird zum Einfügen verwendet



4.4.4 Aufwand für Hashing mit offener Adressierung

- Wie bei Hashing mit Verkettung hängt Aufwand für Hashing mit offener Adressierung vom Belegungsfaktor $\alpha = n/m$ ab
  - Bei offener Adressierung maximal 1 Element pro Slot
    - Somit $n \leq m$ und damit $\alpha \leq 1$
- Für die erfolglose Suche werden mit gleichmäßigem Hashing höchstens $1/(1-\alpha)$ Sondierungen benötigt
- Das Einfügen mit gleichmäßigem Hashing erfordert im Mittel ebenfalls höchstens $1/(1-\alpha)$ Sondierungen
- Die erfolgreiche Suche mit gleichmäßigem Hashing erfordert höchstens $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ Sondierungen (sofern nach jedem Schlüssel mit gleicher Wahrscheinlichkeit gesucht wird)
- Zahlenbeispiele
  - Tabelle zu 50% gefüllt: $< 1{,}387$ Sondierungen
  - Tabelle zu 90% gefüllt: $< 2{,}559$ Sondierungen

4.5 Zusammenfassung Hashing-Grundlagen

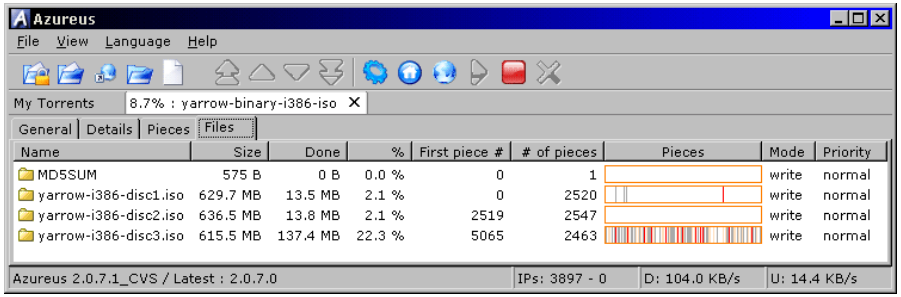
- Hashtabellen unterstützen die Wörterbuchoperationen $INSERT$, $SEARCH$ und $DELETE$ für dynamische Menge sehr effizient
- Unter günstigen Bedingungen können im Average-Case alle Operationen in $O(1)$ durchgeführt werden
- Im Worst-Case benötigt $SEARCH$ jedoch wie bei verketteten Listen $\Theta(n)$
- Mehrere Ansätze für gute Hashfunktionen
  - Divisionsmethode
  - Multiplikationsmethode
- Da zwei Schlüssel auf den gleichen Hashwert abgebildet werden können, werden Verfahren zur Kollisionsauflösung benötigt
  - Kollisionsauflösung durch Verkettung
  - Offene Adressierung

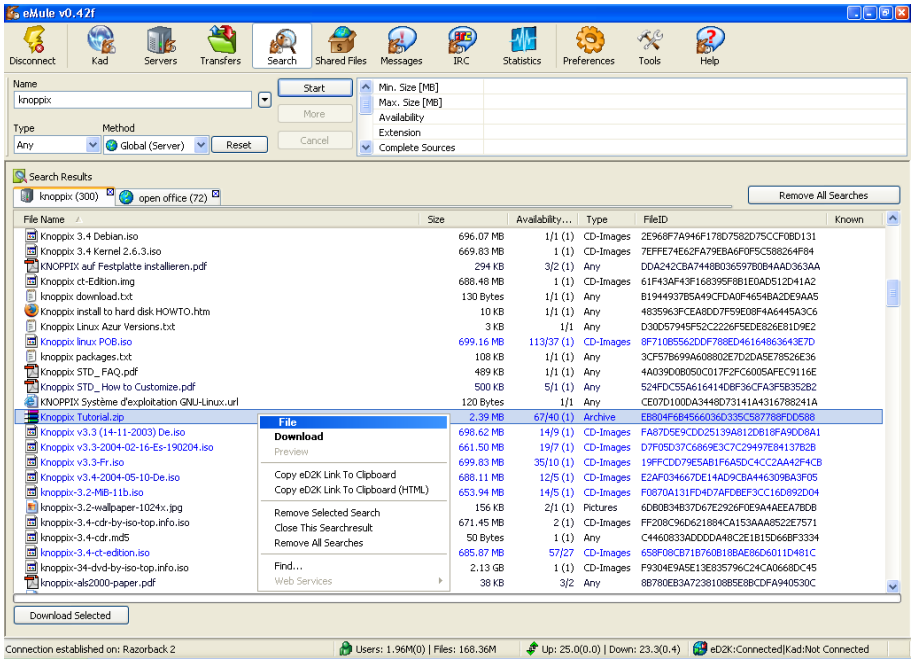
4.6 Exkurs: Verteilte Hashtabellen

- Anwendungsbeispiel aus der Telematik
- Verteilte Hashtabellen werden zur dezentralen Datenablage in Peer-to-Peer-Netzwerken verwendet
- Grundidee von Peer-to-Peer (P2P): Keine festen Client-/Server-Rollen
  - Clients können auch Server sein und Dienste anbieten
  - Prominentes Beispiel in junger Zeit: Filesharing („Tauschbörsen“)
    - Eigentlicher Zweck: Entlastung zentraler Server und deren Leitungen
- Fand in kurzer Zeit große Verbreitung
  - P2P verursacht ungefähr 18% des globalen Inter-Domänen-Verkehrs  [Labo10]
  - “BitTorrent, Inc. Grows to Over 100 Million Active Monthly Users”  [Bit11]
  - P2P Filesharing
    - 3,5 Exabytes pro Monat weltweit
    - Zwischen 2000 und 2010 größter „Verkehrstyp“ im Internet  [Cisco10]

Beispiel Filesharing

- Einfache Nutzung
- Häufig problemloser Betrieb





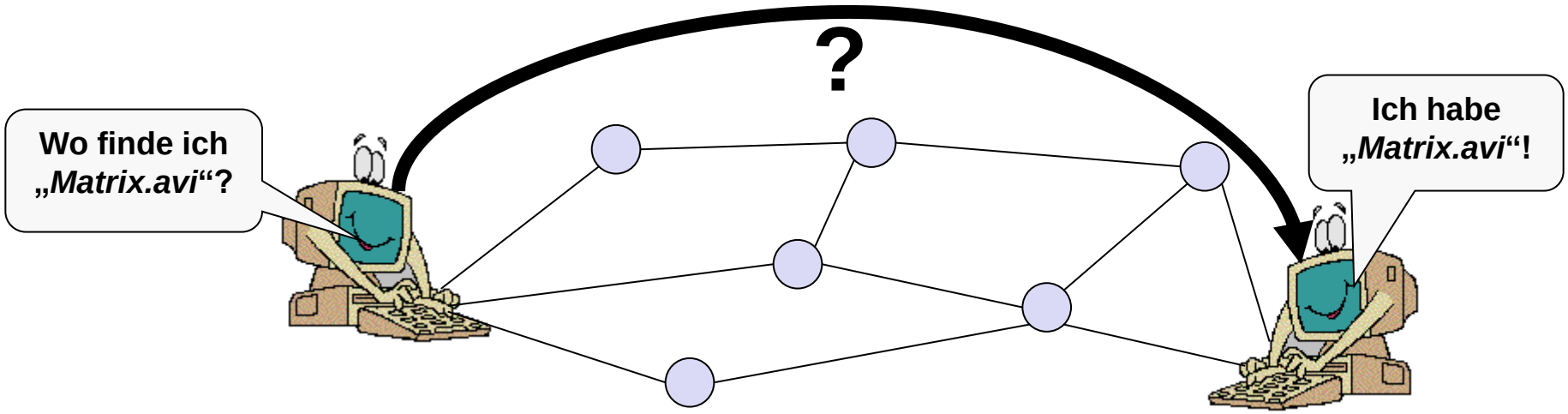
Client installieren,

ins P2P-Netzwerk einloggen,

Inhalte suchen,

von verfügbaren Quellen herunterladen

Informationssuche in Peer-to-Peer-Systemen



Wesentliches Problem in allen Peer-to-Peer-Systemen?

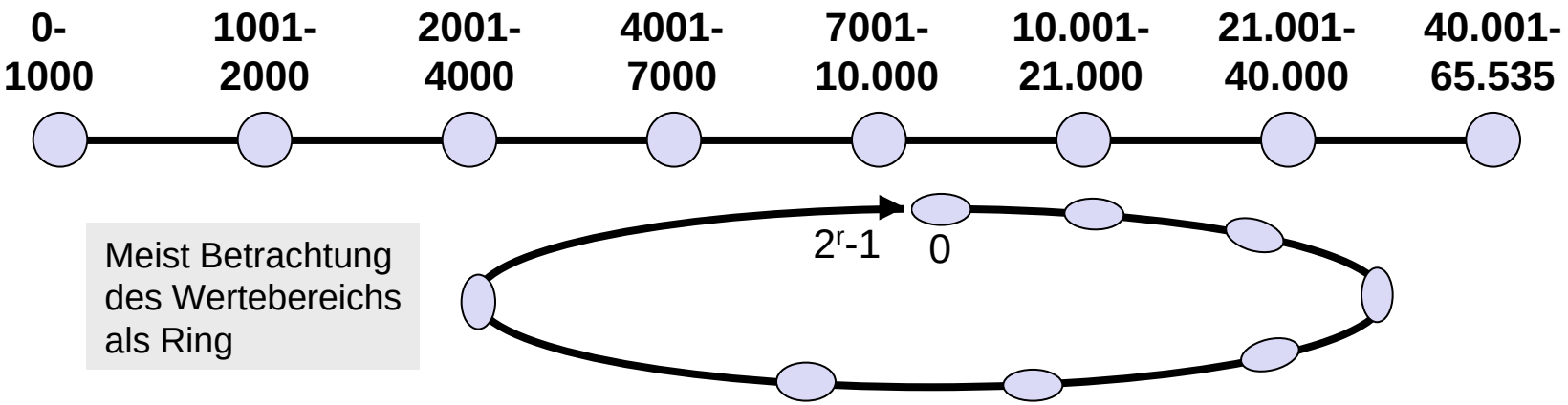
- Auffinden eines Datums im verteilten System
  - Wo soll Datum gespeichert werden?
    - `Publish(“Inhalt”, …)`
  - Wie findet Anfrage den Speicherort?
    - `Lookup(“Inhalt”)`
- Geringer Aufwand: Kommunikation, Speicher
- Robust gegen Ausfälle und häufige Änderungen

Prinzip verteilter Hashtabellen (I)

- Idee verteilter Hashtabellen
  - Verteile Daten (Inhalte) über alle Knoten
    - Alternativ: Information über tatsächliche Lokation eines Inhalts (Indirektion)
  - Bei Anforderung eines Datums frage Knoten, der Inhalt speichert
- Herausforderungen
  - Gleichmäßige Verteilung der Inhalte auf Knotenmenge
    - Effizientes Auffinden von Inhalten
  - Ständige Anpassung durch Ausfall, Beitritt und Austritt von Knoten
    → Selbstorganisation
    - Vergabe von Zuständigkeiten an neue Knoten
    - Übernahme und Neuverteilung von Zuständigkeiten bei Ausfall, Austritt

Prinzip verteilter Hashtabellen (II)

- Schritt 1:
  - Abbildung des Inhalts auf linearen Wertebereich
    - Inhalt wird auf Identifikator aus diesem Wertebereich abgebildet:
      ID $\in [0, \ldots, 2^r - 1]$ >> N_{max}
      (N_{max}: Anzahl der maximal zu speichernden Objekte)
    - Abbildung des Inhalts in Wertebereich durch Hashfunktion $h(k)$,
      z.B. $h($„`/movie/Matrix/divx/en`“$)$ = 2313
  - Verteilung des Wertebereichs über alle Knoten

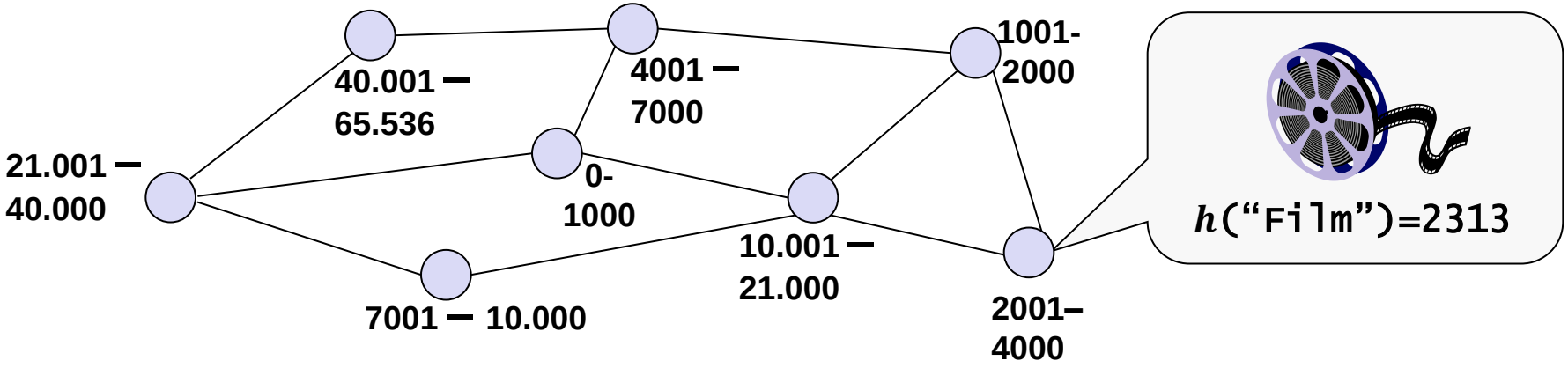


Speicherung der Inhalte: Direkt

Wie werden „Inhalte“ in Knoten gespeichert?

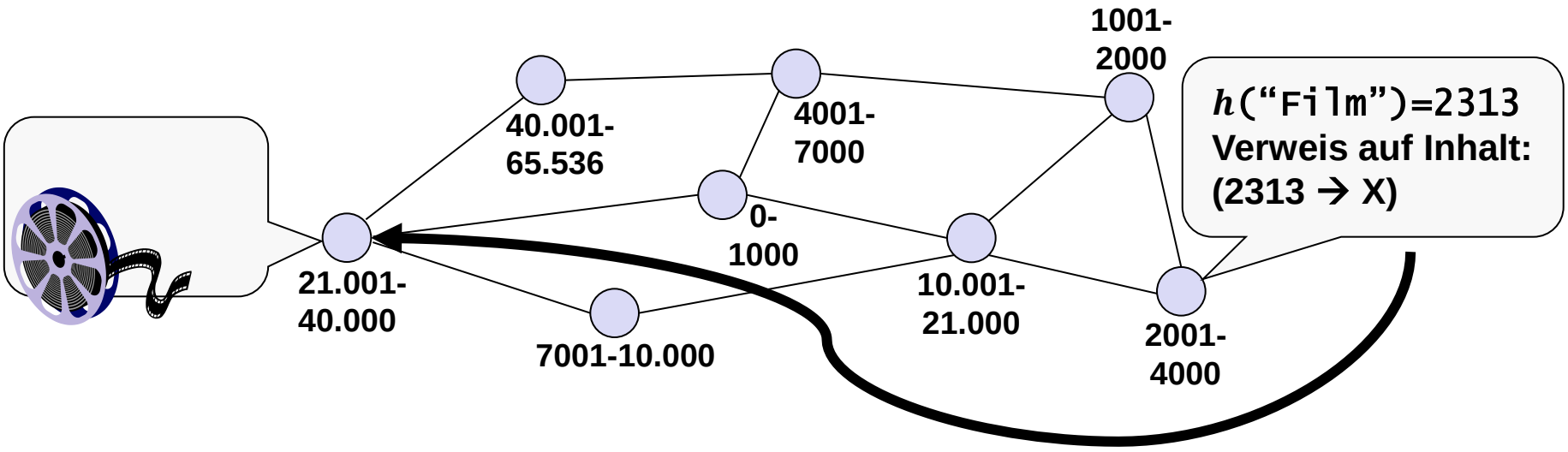
- Annahme: h(“`Inhalt`”) ist Abbildung in Wertebereich
- Abbildung Wertebereich → Knoten notwendig
  (Knoten haben dazu eine zufällige gewählte *NodeID*)

- Direkt
  - Inhalt wird in Knoten h(“`Inhalt`”) gespeichert
    → unflexibel für große Inhalte



Speicherung der Inhalte: Indirekt

- Indirekt
  - Knoten in verteilter Hashtabelle speichern Tupel `(Schlüssel,Wert)`
    - Schlüssel = h(„`/movie/Matrix/divx/en`“) → 2313
    - Wert: meist reale Adresse, an der Inhalt gespeichert wird: (IP, Port) = (129.13.15.3, 4711)
  - Flexibler, aber ein Schritt mehr zum Inhalt

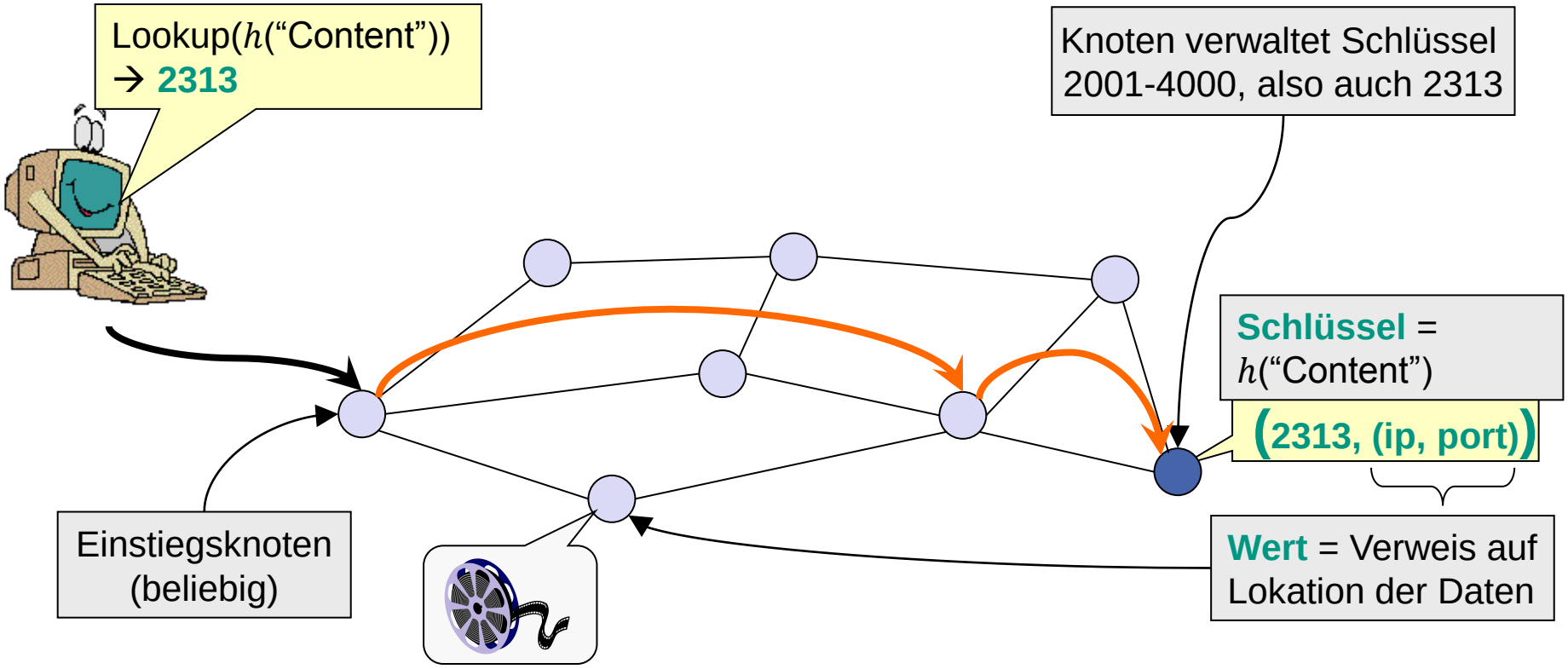


Auffinden der Inhalte (I)

- Schritt 2: Auffinden des Inhalts
  (Inhaltsbasierte Suche – Content based Routing)
  - Achtung: keine Suche wie z.B. Finde alle Daten, die bestimmtes Stichwort enthalten
- Ziel: geringer, skalierbarer Aufwand
  - $O(1)$ bei zentraler Hashtabelle
    - Aber: Verwaltung zentraler Hashtabelle zu umfangreich (→Server!)
  - Angestrebter Aufwand bei verteilter Hashtabelle
    - $O(\log N)$: Hops zum Auffinden eines Objekts
    - $O(\log N)$: Anzahl der Schlüssel und Routing-Infos pro Knoten

Auffinden der Inhalte (II)

- Suche nach Schlüssel
  - Einstieg bei beliebigem Knoten
  - Routing zu gesuchtem Inhalt (Schlüssel)



Weitere Herausforderungen

- Einloggen in ein P2P-Netz
  - Wo einloggen, wenn es keine zentralen Server gibt?
  - Verbundene P2P-Knoten können Teilnehmerinformationen austauschen
  - Aber: Wo Anschluss finden, wenn noch nicht verbunden?
  - Sogenanntes Bootstrapping-Problem
  - Die meisten „Lösungen“ heutzutage „schummeln“
    - P2P-Software bringt Listen von gut verfügbaren Teilnehmer-Systemen mit
    - Zentrale Ablage von Teilnehmer-Systemen (z.B. auf Webseiten)
    - Widersprüche zum puristischen P2P-Konzept
- Replikation von Daten
  - Was passiert, wenn ein zuständiger Knoten offline geht?
  - Durchschnittliche Verweildauer eines Knotens: 1,5 h
  - Verfügbarkeit von Daten erfordert andere Strategien als bei einem Server
- Sicherheit
  - Teilnehmer könnte gefälschte Inhalte einspeisen…
  - …oder Suchanfragen nicht beantworten und nicht weiterleiten

Resümee: Verteilte Hashtabellen

- Datensätze werden gleichmäßig über alle Teilnehmer verteilt (oft mit Redundanz)
  - Erlaubt stetiges Wachstum der Anzahl gespeicherter Datensätze
  - Toleriert Ausfälle von Teilnehmern
  - Überlebt gezielte Attacken
- Selbstorganisierend
  - Einfache und günstige Umsetzung
  - Breites Feld von Anwendungen
    - Schlüssel hat keine semantische Bedeutung
    - Wert ist anwendungsabhängig
- Aber: Eine ganze Reihe ungelöster Probleme wie
  - Sicherheit
  - Bootstrapping
  - …

Literatur

Hashing-Grundlagen:

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3
- [Knut98] Donald E. Knuth, „*Sorting and Searching*, volume 3 of *The Art of Computer Programming*“, Addison-Wesley, 2. Auflage, 1998

Verteile Hashtabellen:

- [Skype06] S. A. Baset und H. Schulzrinne, An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol, April 2006, IEEE INFOCOM 2006, Barcelona, Spanien
- [Labo10] C. Labovitz, S. Lekel-Johnson, D. McPherson, J. Oberheide, F. Jahanian, Internet Inter-Domain Traffic, August 2010, ACM SIGCOMM 2010, Neu-Deli, Indien
- [Cisco10] Cisco Inc., Cisco Visual Networking Index: Forecast and Methodology, 2009-2014, Juni 2010 (online)
- [Bit11] Bittorrent Inc. Pressemitteilung, 1. Januar 2011 (online)

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 5 – Heaps

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

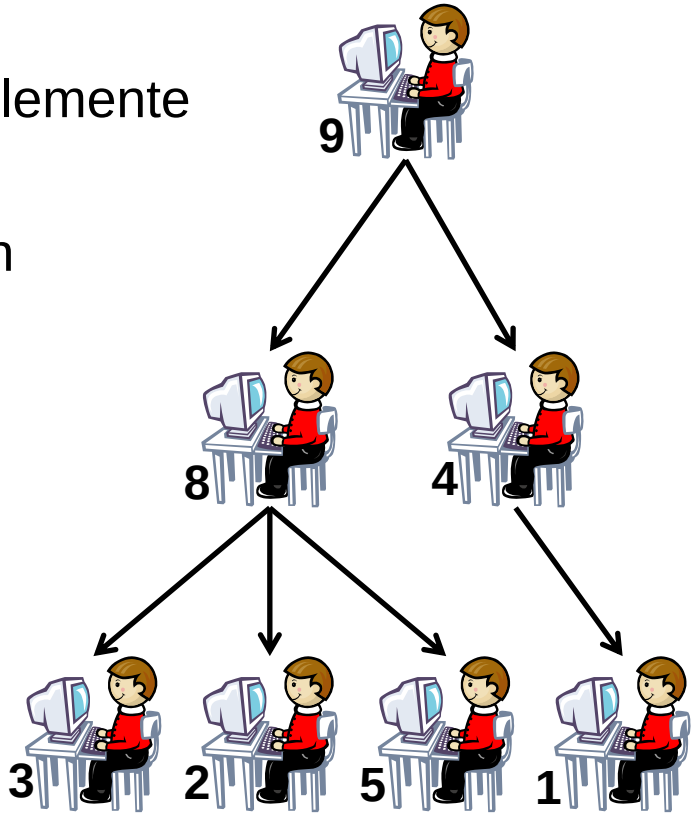
5.1 Definition und Motivation

- Heap – „Haufen“ oder „Halde“

- Datenstruktur zur Speicherung von Datenelementen
  - Elemente besitzen Schlüssel zur Sortierung
- Totale Ordnung bestimmt Reihenfolge der Elemente

- Beispiel: Organisationsplan im Unternehmen
  - Stellung eines Mitarbeiters wird durch eine Zahl ausgedrückt (= Schlüssel)
  - Zahl ist Größer als die der Unterstellten
  - Zahl ist kleiner als die der Vorgesetzten

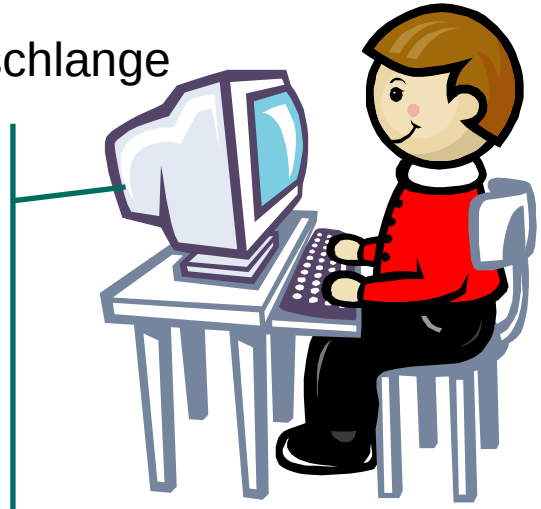
- Ergebnis: Partiell geordneter Baum
  - Siehe Beispiel rechts



Definition und Motivation

- Weiteres Beispiel: Elemente gemäß ihrer Priorität (zwischen)speichern
  - Tasks in einem Betriebssystem
  - Verwendungsbeispiel: Einsatz als Vorrangwarteschlange

Task	Prio.
Task A	(1)
Task F	(3)
Task C	(4)
Task H	(6)
…	



- Anforderungen
  - Einfacher Zugriff auf das höchst-priore Element
  - Einfügen und Löschen von Elementen
  - Verschmelzen zweier Listen
  - Hohe Effizienz aller Aktionen

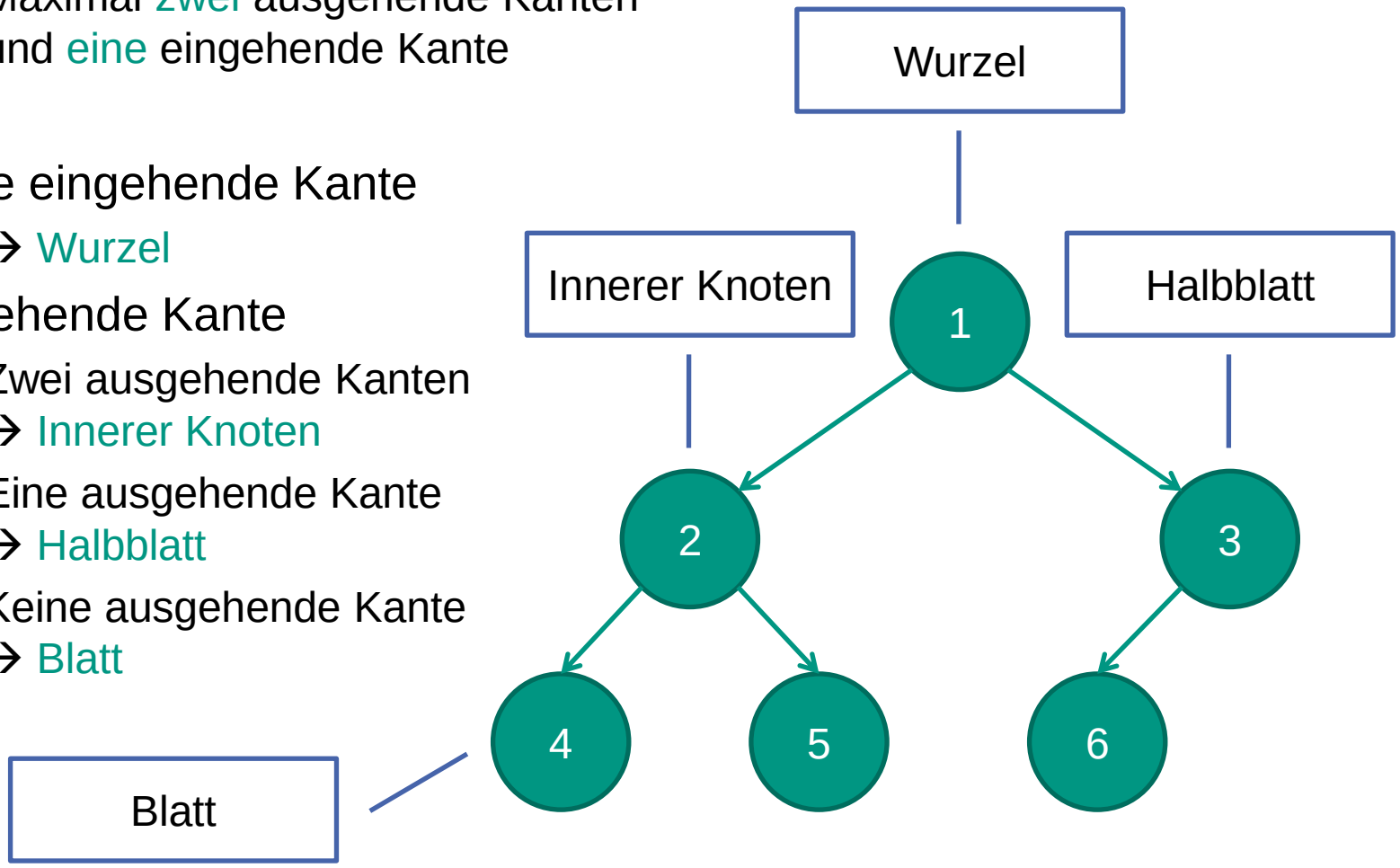
5.2 Binäre Heaps

- Als grundlegende Datenstruktur um z.B. eine Prioriätsliste zu realisieren kommen in der Regel binäre Heaps zum Einsatz
  - Darstellung als Binär-Baum

- Im Heap können Elemente abgelegt und wieder entnommen werden
  - Jedem Element ist ein Schlüssel zugeordnet
- Totale Ordnung über der Menge der Schlüssel bestimmt die Reihenfolge der Elemente
  - Beispiele
    Kleiner-Relation „<“ (Min-Heap),
    Größer-Relation „>“ (Max-Heap)

5.2.1 Einschub: Binär-Bäume – Knotentypen

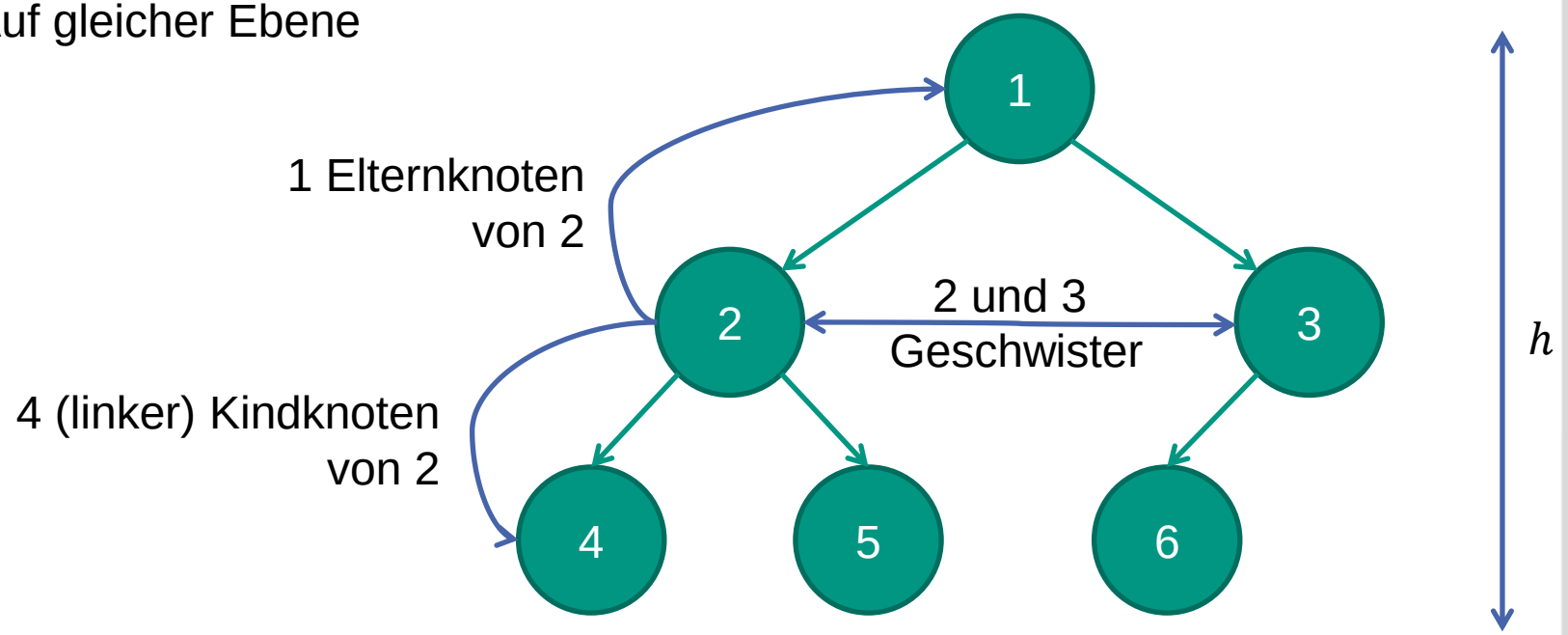
- Gerichteter Graph
  - Maximal zwei ausgehende Kanten und eine eingehende Kante
- Keine eingehende Kante
  - → Wurzel
- Eingehende Kante
  - Zwei ausgehende Kanten → Innerer Knoten
  - Eine ausgehende Kante → Halbblatt
  - Keine ausgehende Kante → Blatt



Einschub: Binär-Bäume – Knotenbeziehungen

- Elternknoten
  - Vorgängerknoten
- Kindknoten
  - Nachfolgerknoten
- Geschwister
  - Auf gleicher Ebene

- Zahl der Elemente n
- Höhe des Baumes h
  - $h = \lfloor \lg n \rfloor$
  - Hinweis: $\lg n = \log_2 n$



5.2.2 Darstellung und Operationen

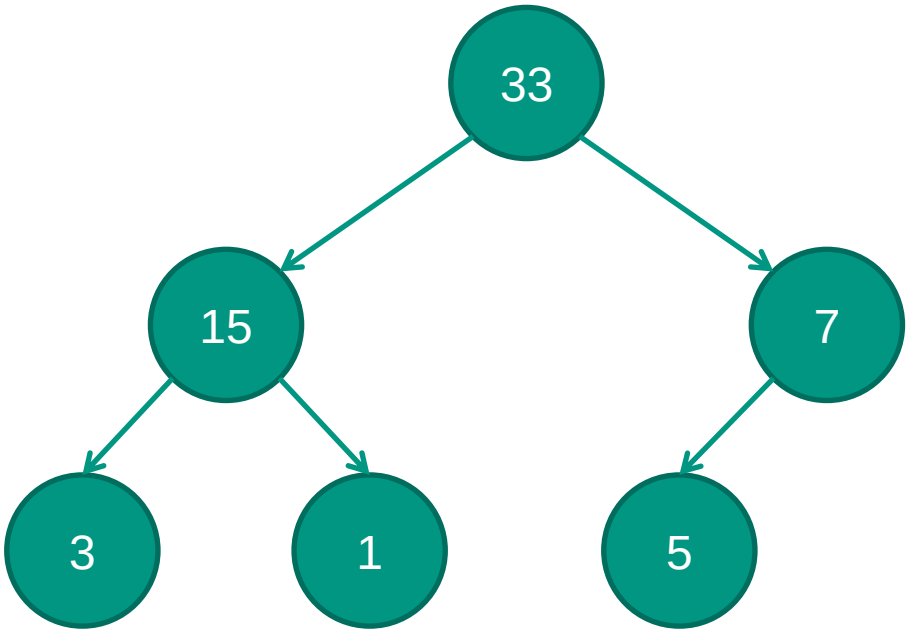
- Heaps lassen sich als Graph (genauer: Binär-Baum) aber auch als Array darstellen
  - Graph = Anschaulich für Untersuchungen und Beispiele
  - Array = Effizient für Implementierung in Programmen

- Operationen im Max-Heap
  - Insert, Einfügen eines Elementes in den Heap
  - Maximum, Rückgabe des Elementes mit maximalem Schlüssel
  - ExtractMax, Rückgabe und Entfernen des Elementes mit maximalem Schlüssel
- Operationen im Min-Heap
  - Insert, Einfügen eines Elementes in den Heap
  - Minimum, Rückgabe des Elementes mit minimalem Schlüssel
  - ExtractMin, Rückgabe und Entfernen des Elementes mit minimalem Schlüssel

Darstellungsformen (I)

- Darstellung als Graph
  - Generell gilt $M.key \times N.key$ für alle Knoten M unterhalb von N, wobei $\times$ die totale Ordnungsrelation darstellt
  - Keine Ordnung innerhalb einer Ebene

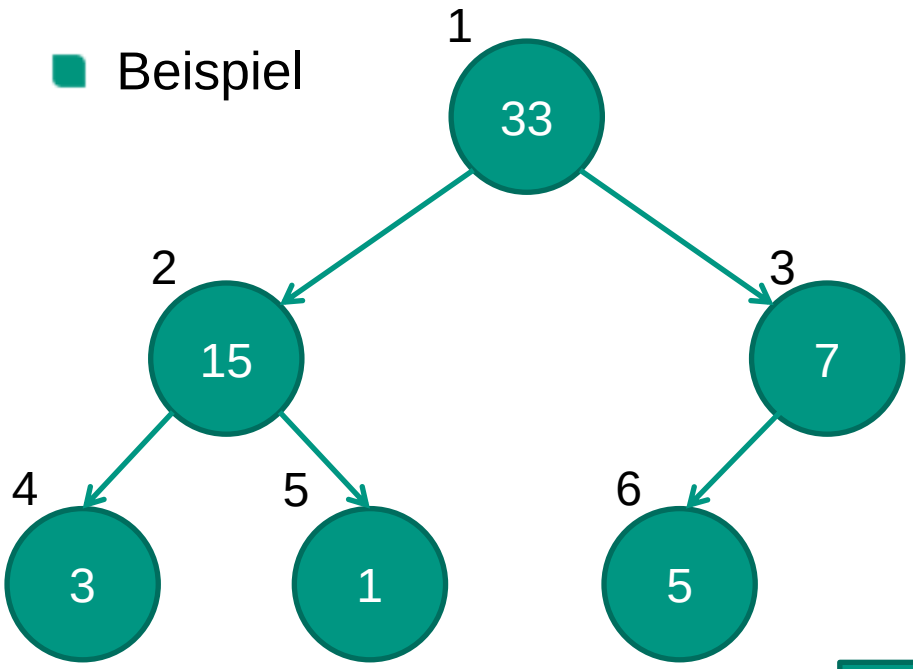
- Beispiel
  - Max-Heap (Relation „>“)
  - $M.key > N.key$



Darstellungsformen (II)

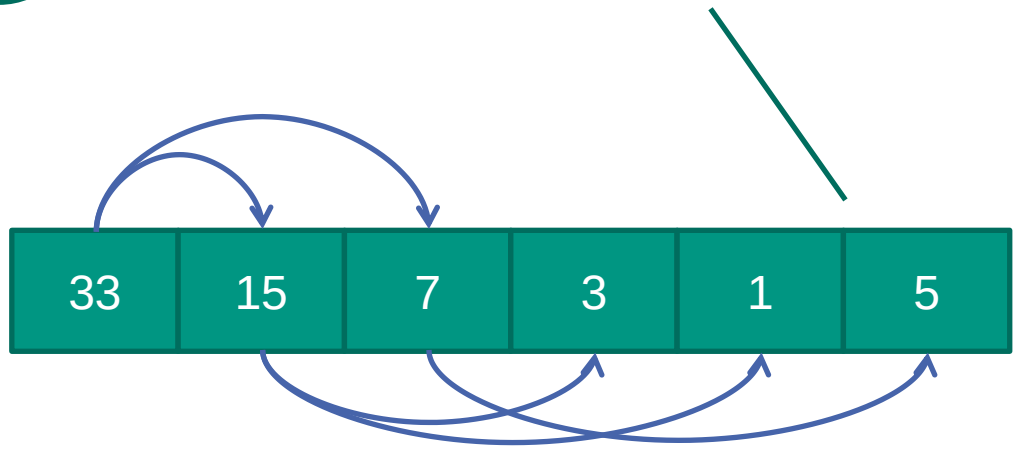
- Darstellung als Array
  - Kinder des Elements $A[i]$ sind $A[2i]$ und $A[2i+1]$

- Beispiel



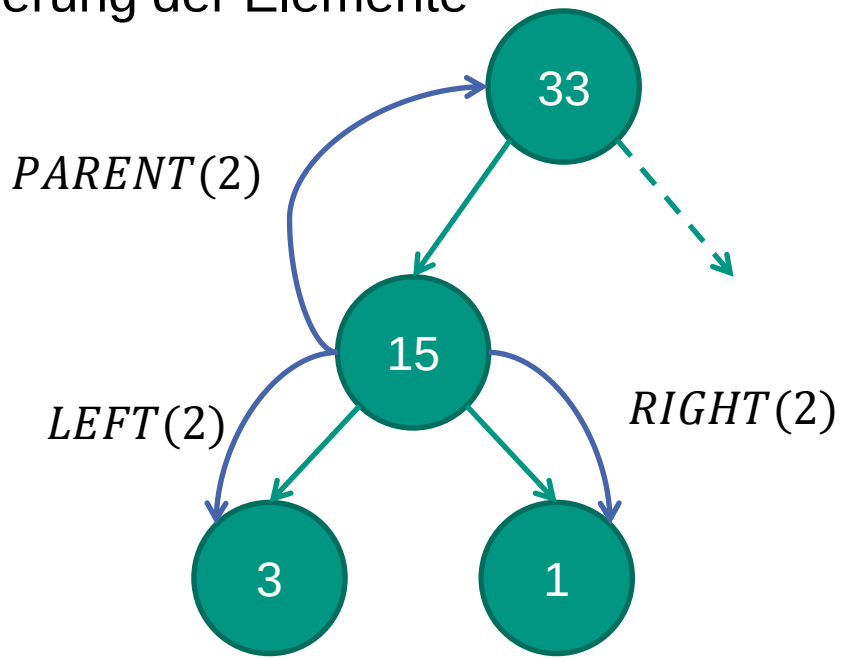
Max-Heap
mit $A.heapgröße = 6$

Keine Lücken im Array!
(bedingt durch
Baumstruktur und -konstruktion)



Orientierung im Graphen

- Pseudocode für die relative Adressierung der Elemente
  - $PARENT(i)$
    1 **return** $\lfloor i/2 \rfloor$
  - $LEFT(i)$
    1 **return** 2i
  - $RIGHT(i)$
    1 **return** 2i + 1



- Damit lässt sich die Heap-Eigenschaft formal definieren
  - Für Min-Heaps gilt: $A[PARENT(i)] < A[i]$
  - Für Max-Heaps gilt: $A[PARENT(i)] > A[i]$

5.2.3 Operationen – Maximum

- Zugriff auf das Maximum trivial
  - Wurzel im Baum bzw.
  - Erster Eintrag des Arrays

- $HEAP - MAXIMUM(A)$
  1 **return** $A[1]$

- Komplexität
  - Konstanter Aufwand = $\theta(1)$

- „Haken“ an der Sache
  - Üblicherweise soll das Element anschließend auch entfernt werden
    → ExtractMax
  - Entfernen des Elements komplexer

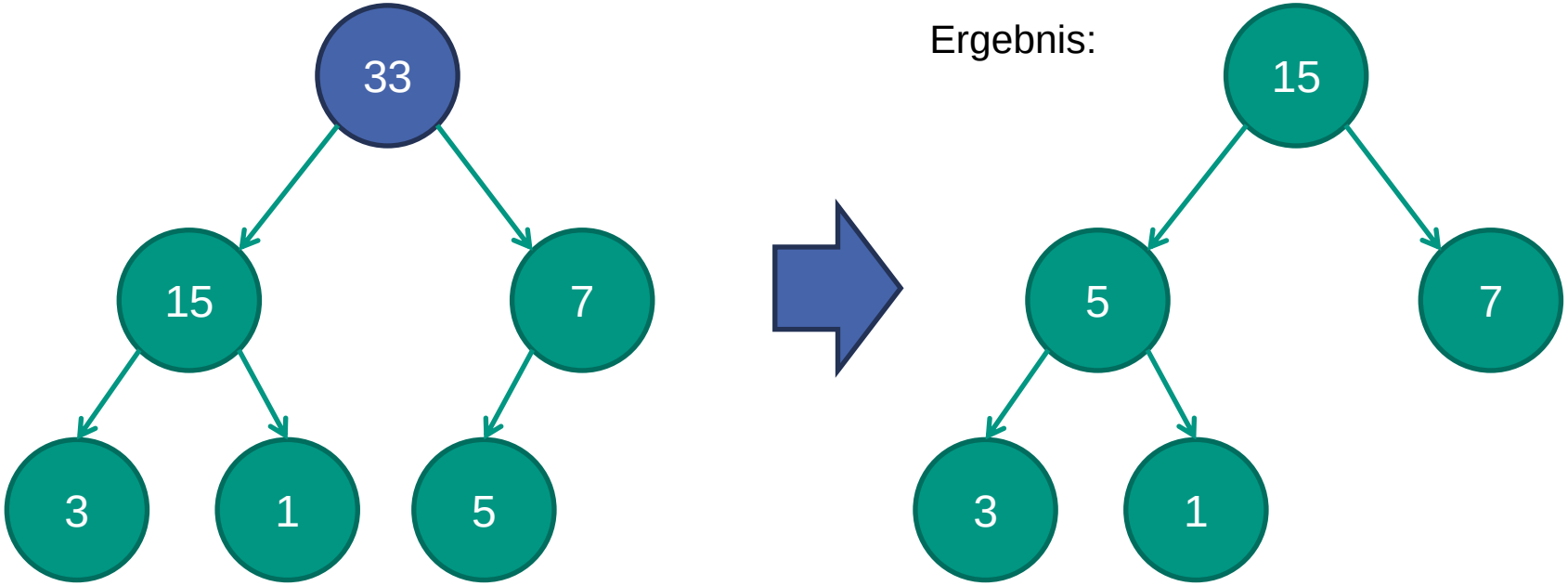
Operationen – ExtractMax

- Prinzip
  - Maximum speichern, aus dem Heap entfernen
  - Das letzte Element ($A[A.heapgröße]$) an die Stelle des Maximums ($A[1]$) platzieren
  - Heap-Eigenschaft wiederherstellen
    → Eigener kleiner Algorithmus

- Heap-Eigenschaft wiederherstellen
  - Annahme
    - Heap-Eigenschaft wird nur durch ein einzelnes Element i verletzt
    - Z.B. durch Einfügen oder Ersetzen eines vorhandenen Elementes (s.o.)
  - Lösung
    - Das Element i muss im Baum entsprechend nach unten verschoben werden bis die Heap-Eigenschaft wiederhergestellt ist

Beispiel

- Entfernen des Maximums (33) und anschließendes Wiederherstellen der Heap-Eigenschaft



Pseudocode – ExtractMax

- $HEAP - EXTRACT - MAX(A)$

```
1  if A.heapgröße < 1
2        error „Heap-Unterlauf“
3  max = HEAP − MAXIMUM(A)
4  A[1] = A[A.heapgröße]
5  A.heapgröße = A.heapgröße – 1
6  MAX − HEAPIFY(A, 1)
7  return max
```

Maximum speichern

Letztes Element verschieben

Heap-Eigenschaft herstellen

Pseudocode – ExtractMax / Heap-Eigenschaft

- $MAX - HEAPIFY(A, i)$

```
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heapgröße und A[l] > A[i]
4       maximum = l
5   else
6       maximum = i
7   if r ≤ A.heapgröße und A[r] > A[maximum]
8       maximum = r
9   if maximum ≠ i
10      vertausche A[i] mit A[maximum]
11      MAX − HEAPIFY(A, maximum)
```

Größtes Element aus $A[i]$, $A[LEFT(i)]$ und $A[RIGHT(i)]$ bestimmen

Falls Maximum nicht $A[i]$ Tausch notwendig

Ggf. rekursiv weiter

Komplexitätsbetrachtung Heap-Eigenschaft

- $MAX - HEAPIFY(A, i)$

```
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heapgröße und A[l] > A[i]
4       maximum = l
5   else
6       maximum = i
7   if r ≤ A.heapgröße und A[r] > A[maximum]
8       maximum = r
9   if maximum ≠ i
10      vertausche A[i] mit A[maximum]
11      MAX − HEAPIFY(A, maximum)
```

$O(1)$

?

- $MAX - HEAPIFY$ wird auf Teilbaum der von i ausgeht, erneut aufgerufen
- Maximale Größe des Teilbaums ist $\frac{2n}{3}$ Knoten ($n = A.heapgröße$)
  - Der Fall, wenn unterste Ebene des Baumes exakt halb gefüllt

Komplexitätsbetrachtung Heap-Eigenschaft

- ...

  9 **if** $maximum \neq i$

  10 vertausche $A[i]$ mit $A[maximum]$

  11 $MAX - HEAPIFY(A, maximum)$

  } $T(\frac{2n}{3})$

- Rekursionsgleichung für Worst-Case: $T(n) \leq T\left(\frac{2n}{3}\right) + O(1)$
  - Lösbar mit Master-Theorem
    - $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ mit $a = 1, b = \frac{3}{2}, f(n) = O(1)$
    - $f(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_b 1}\right) = \Theta(1)$ → Fall 2
  - Ergebnis: $T(n) = O(n^0 \lg n) = O(\lg n)$

- Laufzeit von $MAX - HEAPIFY$ damit $O(\lg n)$
  - Alternativ, hinsichtlich der Baumhöhe in $O(h)$

Komplexitätsbetrachtung ExtractMax

- $HEAP - EXTRACT - MAX(A)$

```
1  if A.heapgröße < 1
2       error „Heap-Unterlauf“
3  max = HEAP − MAXIMUM(A)
4  A[1] = A[A.heapgröße]
5  A.heapgröße = A.heapgröße− 1
6  MAX − HEAPIFY(A, 1)
7  return max
```

$O(1)$

$O(\lg n)$

- $O(\lg n) + O(1) = O(\lg n)$

- Das Entnehmen und Entfernen des Maximums weist damit im schlechtesten Fall insgesamt logarithmische Komplexität auf

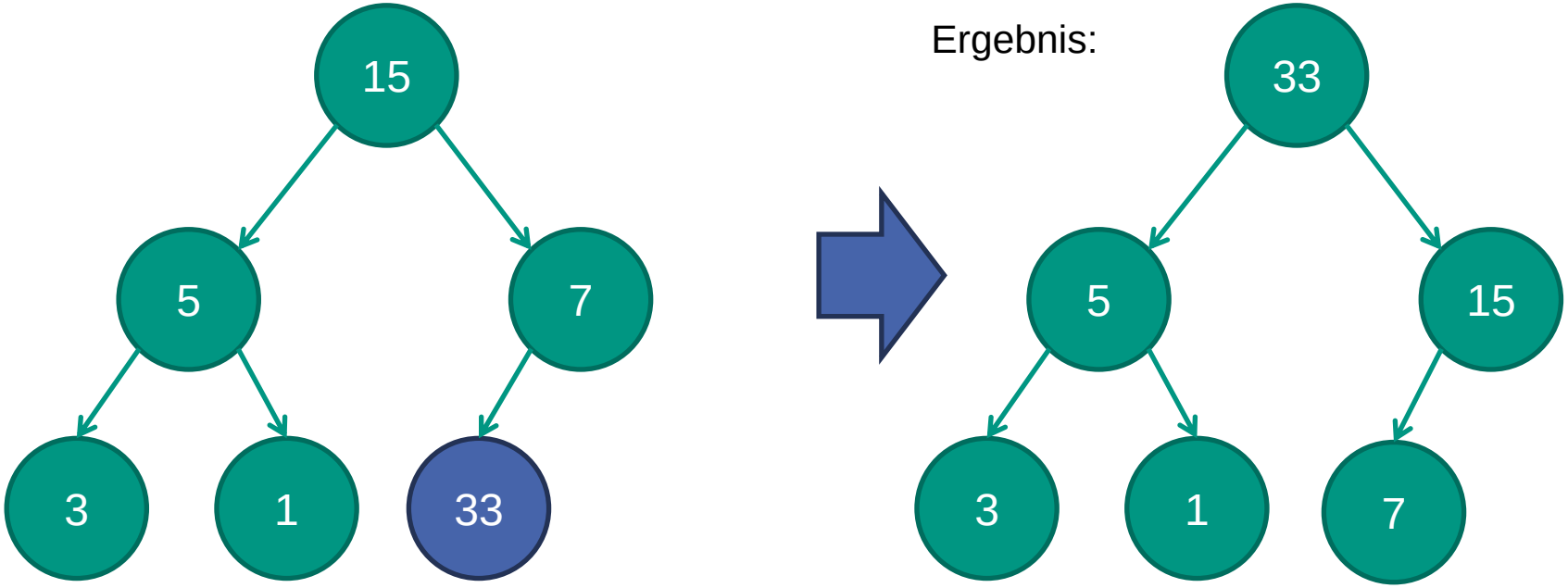
Operationen – Insert

- Prinzip
  - Neues Element an den Heap anhängen
  - Heap-Eigenschaft wiederherstellen

- Heap-Eigenschaft wiederherstellen
  - Andere Voraussetzungen als beim Entnehmen des Maximums
  - $MAX - HEAPIFY()$ kann deshalb nicht verwendet werden
  - Statt dessen muss das Element so lange im Baum nach oben verschoben werden, bis die Heap-Eigenschaft wiederhergestellt ist

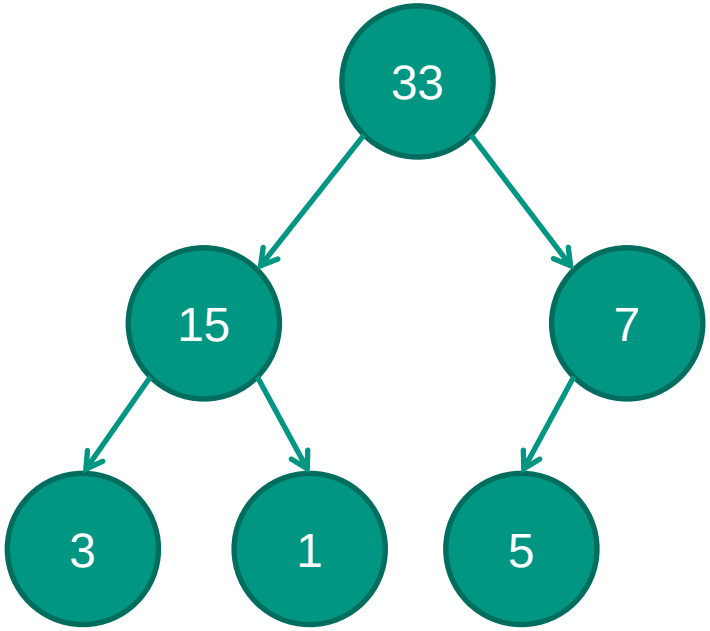
Beispiel

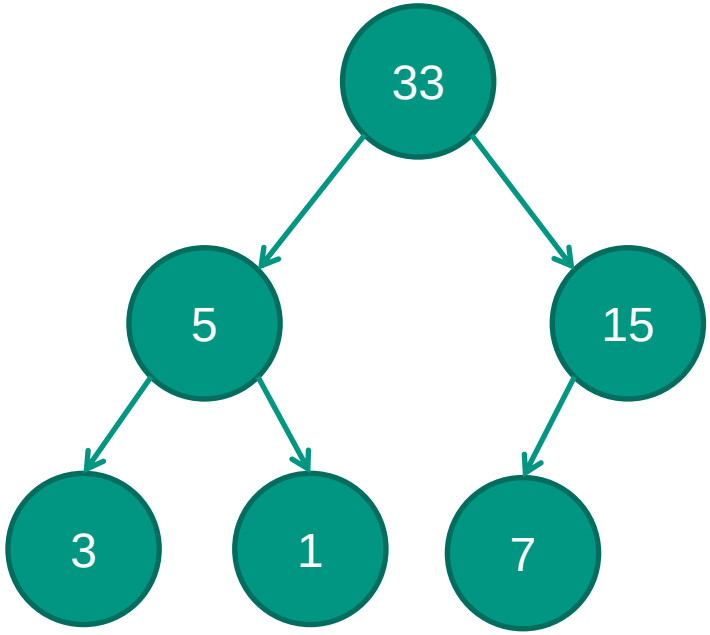
- Einfügen eines Elements (33) und anschließendes Wiederherstellen der Heap-Eigenschaft



Vergleich mit Baum aus Beispiel zu Insert

- Entfernen und Einfügen von „33“ ergibt abweichenden Heap





- → Es kann mehrere Wege geben, eine Elementenmenge als Heap darzustellen

Pseudocode – Insert

- $MAX - HEAP - INSERT(A, schlüssel)$
  1 $A.heapgröße = A.heapgröße + 1$
  2 $A[A.heapgröße] = -\infty$
  3 $INCREASE - KEY(A, A.heapgröße, schlüssel)$

Element anhängen

Schlüssel setzen und Heap-Eigenschaft herstellen

Pseudocode – Insert / Heap-Eigenschaft

- $INCREASE - KEY(A, i, schlüssel)$

```
1  if schlüssel < A[i]
2      error „neuer Schlüssel kleiner als aktueller Schlüssel“
3  A[i] = schlüssel
4  while i > 1 und A[PARENT(i)] < A[i]
5      vertausche A[i] mit A[PARENT(i)]
6      i = PARENT(i)
```

Solange Heap-Eigenschaft verletzt Tausch notwendig

Komplexitätsbetrachtung Heap-Eigenschaft

- $INCREASE - KEY(A, i, schlüssel)$

```
1  if schlüssel < A[i]
2      error „neuer Schlüssel kleiner
       als aktueller Schlüssel“
3  A[i] = schlüssel
4  while i > 1 und A[PARENT(i)] < A[i]
5      vertausche A[i] mit A[PARENT(i)]
6      i = PARENT(i)
```

$O(1)$

$O(\lg n)$

- Worst-Case Betrachtung
- Die **while**-Schleife wird höchstens $\lg n$ mal durchlaufen, da die Pfadlänge vom eingefügten Blatt bis zur Wurzel maximal $h = \lg n$ beträgt

Komplexitätsbetrachtung Insert

- $MAX - HEAP - INSERT(A, schlüssel)$
  1 $A.heapgröße = A.heapgröße + 1$
  2 $A[A.heapgröße] = -\infty$
  3 $INCREASE - KEY(A, A.heapgröße, schlüssel)$

$O(1)$

$O(\lg n)$

- $O(\lg n) + O(1) = O(\lg n)$

- Das Einfügen eines Elementes weist damit im schlechtesten Fall ebenfalls logarithmische Komplexität auf

5.2.4 Heaps aufbauen

- Bei den vorherigen Operationen wurde immer von einem existierenden Heap ausgegangen
  - Wie kann aus unsortierter Menge an Elementen ein Heap erstellt werden?

- Zwei verschiedene Ansätze

- Prinzip A: Erzeugen durch Einfügen
  - Der Reihe nach einzelne Elemente der unsortierten Menge in einen Heap einfügen
  - Der Heap ist anfangs leer
- Prinzip B: Alternativer Ansatz
  - Unsortierte Menge als Heap betrachten, dessen Heap-Eigenschaft verletzt ist
  - Heap-Eigenschaft schrittweise wiederherstellen

Erzeugen durch Einfügen

- Existierendes Feld A mit $A.länge = n$ Elemente

- $BUILD - MAX - HEAP - A(A)$
  1 $A.heapgröße = 1$
  2 **for** $i = 2$ **to** $A.länge$
  3 $MAX - HEAP - INSERT(A, A[i])$

  $O(1)$

  $(n-1)$ mal: $O(\lg n)$

- Komplexität
  - $MAX - HEAP - INSERT$ weist im schlechtesten Fall logarithmischen Aufwand auf
  - Die Länge des Feldes A entspricht der Zahl n der Elemente
  - Damit ergibt sich $O(n) * O(\lg n) = O(n \lg n)$
    - Genauer liegt der Aufwand sogar in $\Theta(n \lg n)$ – wichtig im Folgenden

Alternativer Ansatz

- $BUILD - MAX - HEAP - B(A)$

```
1  A.heapgröße = A.länge
2  for i = ⌊A.länge/2⌋ downto 1
3      MAX − HEAPIFY(A, i)
```

$O(1)$

$\left(\frac{n}{2}\right)$ mal: $O(\lg n)$

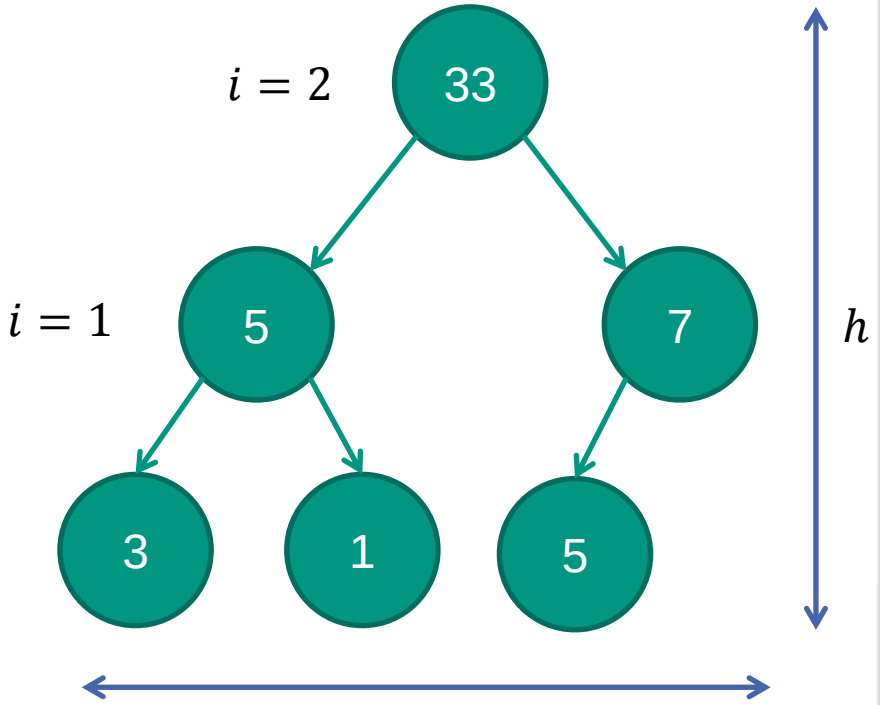
- Komplexität
  - $MAX - HEAPIFY$ liegt in $O(\lg n)$
  - $BUILD - MAX - HEAP - B$ ruft $MAX - HEAPIFY$ $O(n)$ mal auf
  - Damit ergibt sich als grobe Abschätzung $O(n \lg n)$

  - Es kann jedoch gezeigt werden, dass für diesen Ansatz sogar $O(n)$ gilt, der damit besser als $BUILD - MAX - HEAP - A$ ist
    - Beweis im Folgenden!

Komplexitätsbetrachtung

- Motivation: Abschätzung von $MAX - HEAPIFY()$ mit $O(\lg n)$ nur grob
  - Laufzeit variiert abhängig von der Position des Knotens i im Baum

- Beobachtungen
  - Höhe des Baumes der den Heap A als Graph darstellt ist $h = \lfloor \lg n \rfloor$
  - Höchstens $\left\lceil \frac{n}{2^{i+1}} \right\rceil$ Knoten der Höhe i im Baum



- Damit gilt
  - $MAX\_HEAPIFY$ in $O(i)$
  - $\sum_{i=0}^{h} \left\lceil \frac{n}{2^{i+1}} \right\rceil$ Aufrufe von $MAX\_HEAPIFY$ in $BUILD - MAX - HEAP - B$

$\leq \left\lceil \frac{n}{2^{i+1}} \right\rceil$

Komplexitätsbetrachtung

- Die neue Abschätzung ergibt sich
  - Aus der besseren Abschätzung von $MAX\_HEAPIFY$ mit $O(i)$
  - Und der Abschätzung der Zahl der Aufrufe $\sum_{i=0}^{h} \left\lceil \frac{n}{2^{i+1}} \right\rceil$

- Als Gesamtaufwand ergibt sich

$$\sum_{i=0}^{h} \left\lceil \frac{n}{2^{i+1}} \right\rceil O(i) = n \sum_{i=0}^{h} \left\lceil \frac{1}{2^{i+1}} \right\rceil O(i)$$

- Nach oben abgeschätzt

$$= O\left(n \sum_{i=0}^{h} \frac{i}{2^i} \right)$$

- Jetzt noch vereinfachen!

Komplexitätsbetrachtung

- Gesamtaufwand so noch nicht klar ersichtlich

$$O\left(n\sum_{i=0}^{h}\frac{i}{2^i}\right)$$

- Dank folgender geometrischen Reihe kann die Abschätzung weiter vereinfacht werden

$$\sum_{k=0}^{\infty}kx^k=\frac{x}{(1-x)^2}$$

- Dadurch gilt mit $x=\frac{1}{2}$ und $k=i$

$$\sum_{i=0}^{\infty}\frac{i}{2^i}=\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}=2\ \Rightarrow\ O\left(n\sum_{i=0}^{h}\frac{i}{2^i}\right)=O(2n)=O(n)$$

q.e.d.

5.3 Sortieren mit Heaps

- Heaps können zum effizienten Sortieren genutzt werden
  - Effizient hinsichtlich Zeitaufwand → $O(n \lg n)$
  - Effizient hinsichtlich Speicheraufwand → „in-place“

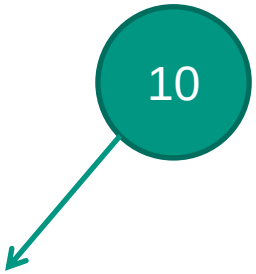
- Gegeben: Eingabemenge im Feld A mit $A.länge = n$ Elementen

- Prinzip
  1. Aus der Menge der zu sortierenden Elemente A einen Max-Heap bauen
  2. Das maximale Element liegt danach bei $A[1]$ → vertauschen mit $A[n]$
  3. Jetzt die Elemente $A[1..n-1]$ als Heap betrachten, $A[1]$ verletzt durch Vertauschung potentiell die Heap-Eigenschaft → Wiederherstellen
  4. Anschließend liegt bei $A[1]$ das Maximum des verkleinerten Heaps
  5. Schritte 2-4 werden wiederholt und der Heap weiter verkleinert
- Ergebnis: A ist aufsteigend sortiert
  - Absteigende Sortierung analog!

Beispiel

- Unsortiertes Feld A $(n = 6)$

- Heap erstellen mit $BUILD - MAX - HEAP - A$



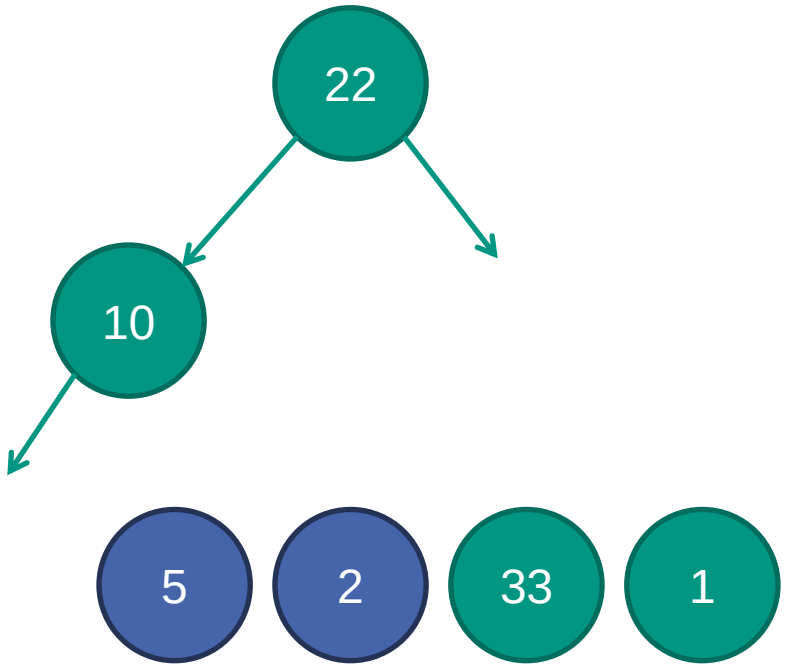
  - Einfügen von 10
    - Heap wird erstellt
  - Einfügen von 22
    - Tausch mit 10 notwendig

Beispiel

- Unsortiertes Feld A $(n = 6)$

10	22	5	2	33	1

- Heap erstellen mit BUILD-MAX-HEAP-A



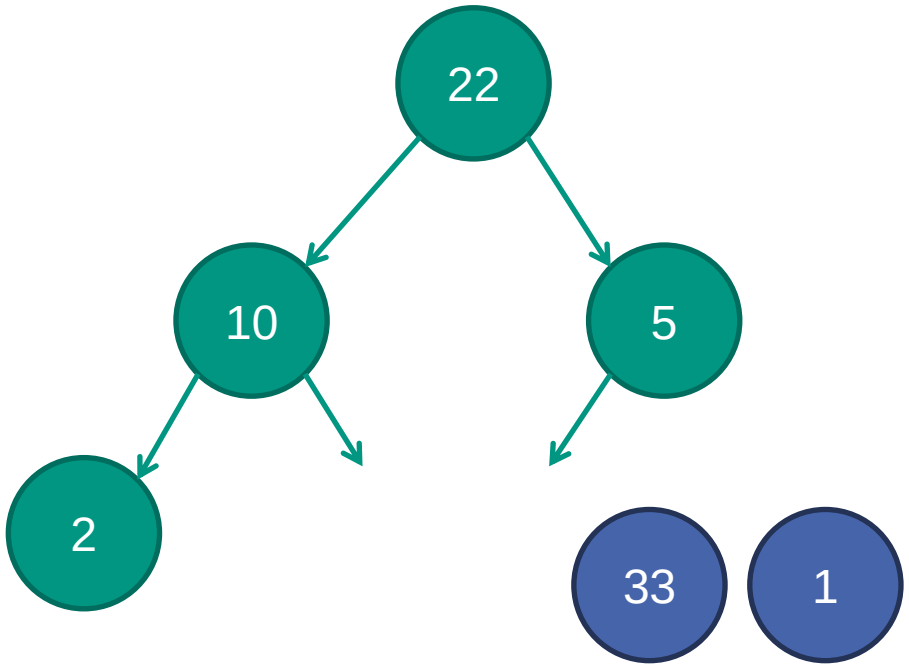
  - Einfügen von 5
    - Kein Tausch notwendig
  - Einfügen von 2
    - Kein Tausch notwendig

Beispiel

- Unsortiertes Feld A $(n = 6)$

10	22	5	2	33	1

- Heap erstellen mit BUILD-MAX-HEAP-A



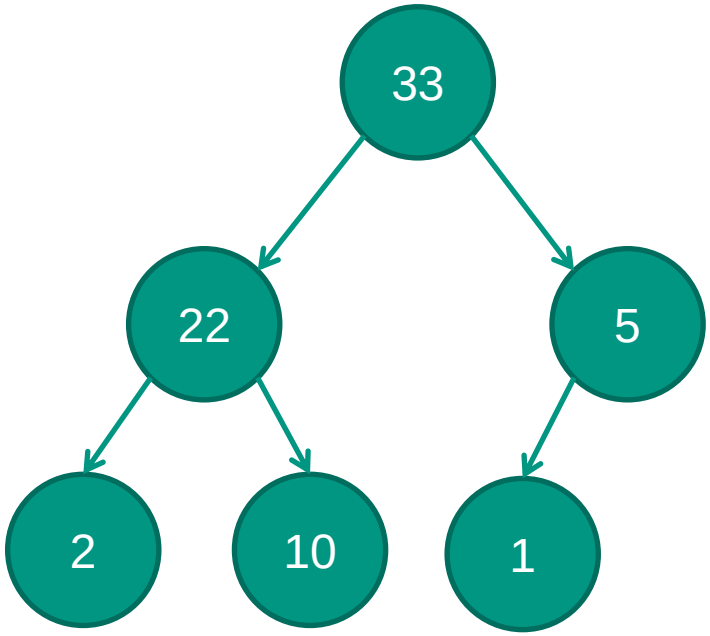
- Einfügen von 33
  - Tausch bis zur Wurzel
- Einfügen von 1
  - Kein Tausch notwendig

Beispiel

- Unsortiertes Feld A $(n = 6)$

10	22	5	2	33	1

- Heap erstellen mit BUILD-MAX-HEAP-A



Beispiel

- Unsortiertes Feld A $(n = 6)$

- Heap erstellt $(A.heapgröße = 6)$

- Maximum ans Ende, Heap verkleinern $(A.heapgröße = 5)$

- Der neue Heap aus den Elementen $A[1..5]$ erfüllt die Heap-Eigenschaften noch nicht → $MAX - HEAPIFY()$

Beispiel

- Heap-Eigenschaft wiederherstellen

- Heap-Eigenschaft wiederhergestellt

- Maximum ans Ende, Heap verkleinern ($A.heapgr\ddot{o}\ss e = 4$)

- Der neue Heap aus den Elementen $A[1..4]$ erfüllt die Heap-Eigenschaften noch nicht → $MAX - HEAPIFY()$

Beispiel

- Heap-Eigenschaft wiederherstellen

- Heap-Eigenschaft wiederhergestellt

- Maximum ans Ende, Heap verkleinern ($A.heapgr\ddot{o}\ss e = 3$)

- Der neue Heap aus den Elementen $A[1..3]$ erfüllt die Heap-Eigenschaften noch nicht → $MAX - HEAPIFY()$

Beispiel

- Heap-Eigenschaft wiederherstellen

- Heap-Eigenschaft wiederhergestellt

- Maximum ans Ende, Heap verkleinern ($A.heapgr\ddot{o}\beta e = 2$)

- Der neue Heap aus den Elementen $A[1..2]$ erfüllt die Heap-Eigenschaften noch nicht → $MAX - HEAPIFY()$

Beispiel

- Heap-Eigenschaft wiederherstellen

- Heap-Eigenschaft wiederhergestellt

- Maximum ans Ende, Heap verkleinern ($A.heapgr\ddot{o}\beta e = 1$)

- Der neue Heap besteht nur aus einem Element, Heap-Eigenschaft erfüllt, Feld sortiert, Fertig!

Pseudocode

- Am Beispiel aufsteigende Sortierung mit Max-Heaps

- $HEAPSORT(A)$
  1 $BUILD-MAX-HEAP-B(A)$
  2 **for** $i = A.länge$ **downto** 2
  3 vertausche $A[1]$ mit $A[i]$
  4 $A.heapgröße = A.heapgröße - 1$
  5 $MAX-HEAPIFY(A, 1)$

Maximum ans Ende des Feldes

Heap verkleinern

Heap-Eigenschaft herstellen

Komplexität

- $HEAPSORT(A)$

```
1  BUILD − MAX − HEAP − B(A)
2  for i = A.länge downto 2
3      vertausche A[1] mit A[i]
4      A.heapgröße = A.heapgröße – 1
5      MAX − HEAPIFY(A, 1)
```

$O(n)$

$O((\mathrm{n}-1)\ lg\ n)$

- Als Gesamtkomplexität ergibt sich $O((\mathrm{n}-1)\lg n) + O(n) = O(n \lg n)$

5.4 Zusammenfassung

- Heaps sind effiziente Datenstrukturen zur Speicherung von Elementen
  - Zugriff auf Maximum bzw. Minimum in konstanter Zeit
  - Entfernen und Einfügen in logarithmischer Zeit
  - Aufbauen in linearer Zeit
- Hauptverwendung als Prioritätswarteschlange
- Außerdem als Möglichkeit zum Sortieren interessant
  - Sortieren in $O(n \lg n)$
  - In der Praxis ist Quick-Sort jedoch meistens schneller

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3
- [Dewd94] Alexander K. Dewdney, „Der Turing Omnibus. Eine Reise durch die Informatik mit 66 Stationen“, Springer, 496 Seiten, Oktober 1994, ISBN 978-3-540-57780-5.

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 6 – Bäume

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

I. Einführung

1. Einführung

II. Suchen und Sortieren

2. Sortieren

III. Datenstrukturen

3. Folgen als Felder und Listen
4. Hashing
5. Heaps

6. Bäume

- 6.1 Bäume
- 6.2 Suchbäume
- 6.3 Rot/Schwarz Bäume
- 6.4 B-Bäume
- 6.5 Digitale Bäume

IV. Graphenalgorithmen

7. Graphrepräsentation
8. Graphtraversierung
9. Kürzeste Wege
10. Minimale Spannbäume

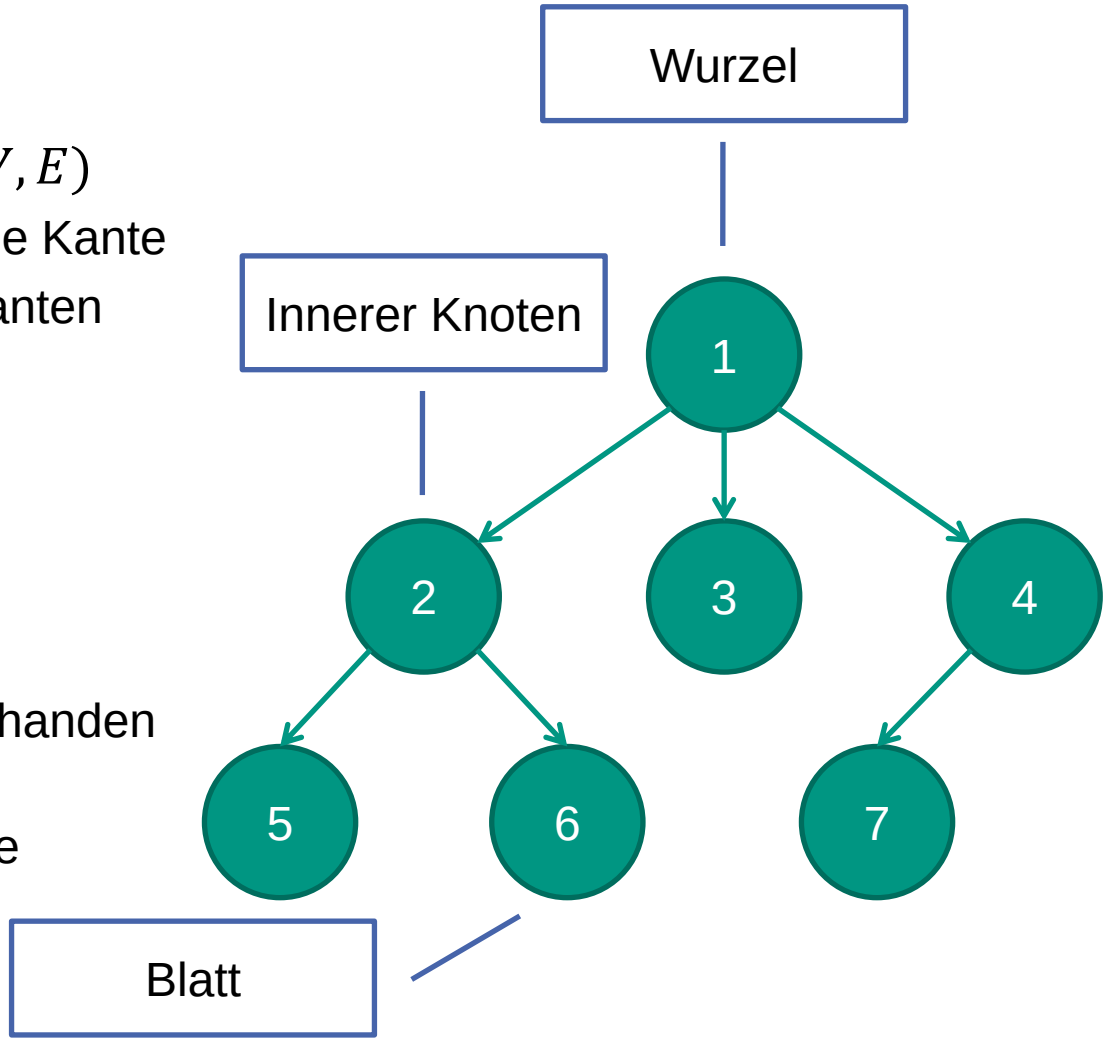
V. Ausblick

11. Generische Optimierungsansätze
12. Zusammenfassung und Ausblick

6.1 Bäume

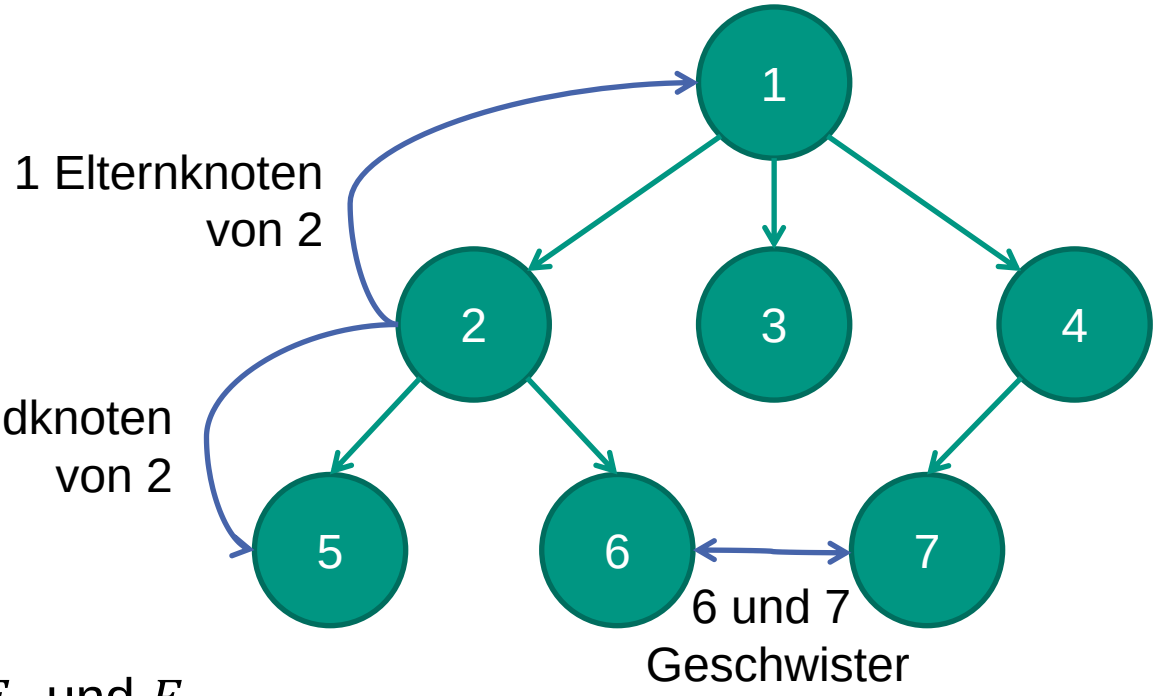
- Gerichteter Graph G mit Knoten und Kanten $G = (V, E)$
  - Maximal eine eingehende Kante
  - Mehrere ausgehende Kanten möglich
- Keine eingehende Kante → Wurzel
- Eingehende Kante
  - Ausgehende Kanten vorhanden → Innerer Knoten
  - Keine ausgehende Kante → Blatt
- Beispiel rechts

$$G = (\{1,2,3,4,5,6,7\}, \{(1,2); (1,3); (1,4); (2,5); (2,6); (4,7)\})$$



Knotenbeziehungen und Eigenschaften

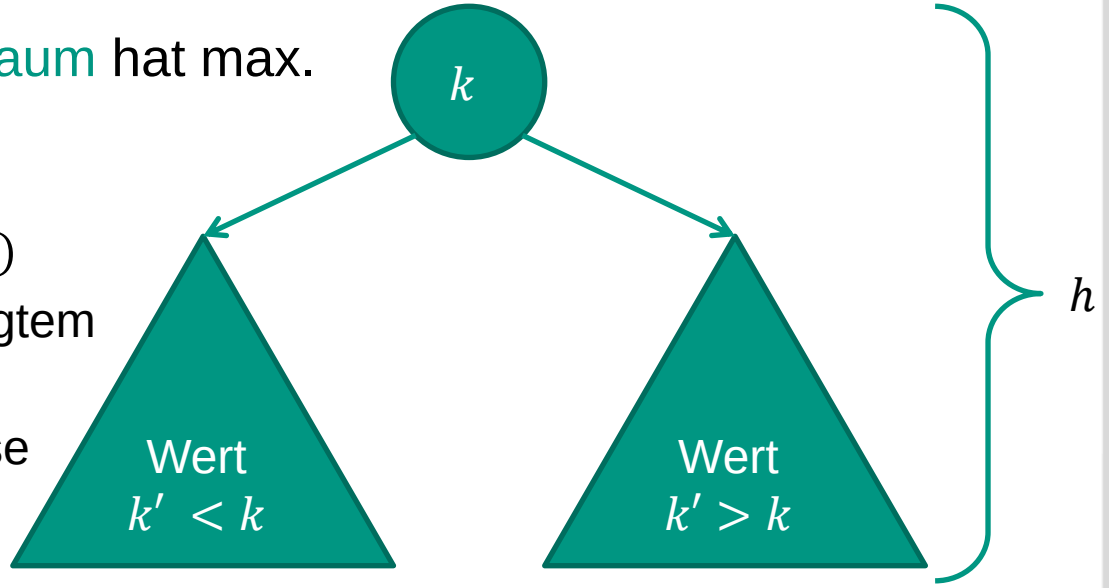
- Elternknoten
  - Vorgängerknoten
- Kindknoten
  - Nachfolgerknoten
- Geschwister
  - Auf gleicher Ebene



- G ist azyklisch
- Zwischen zwei Knoten E_1 und E_2 gibt es genau einen Weg

6.2 Suchbäume

- Effizientes Suchen
  - Grundgedanke: Kombiniere Offenheit der verketteten Liste mit Aufwand $O(\lg n)$ des Teile-und-Herrsche-Prinzips
  - Bäume sind strukturelles Abbild des Teile-und-Herrsche-Prinzips
    - Beim Zugriff jeweils pro Knoten nur einen Sohn weiterverfolgen
    - → Suchbaum
- Beispiel: Ein binärer Suchbaum hat max. zwei Kinder je Knoten
  - Suchen und ähnliche Operationen daher in $O(h)$
  - $h = \lg n$ bei zufällig erzeugtem Baum
  - Aber: $h = n$ im Worst-Case



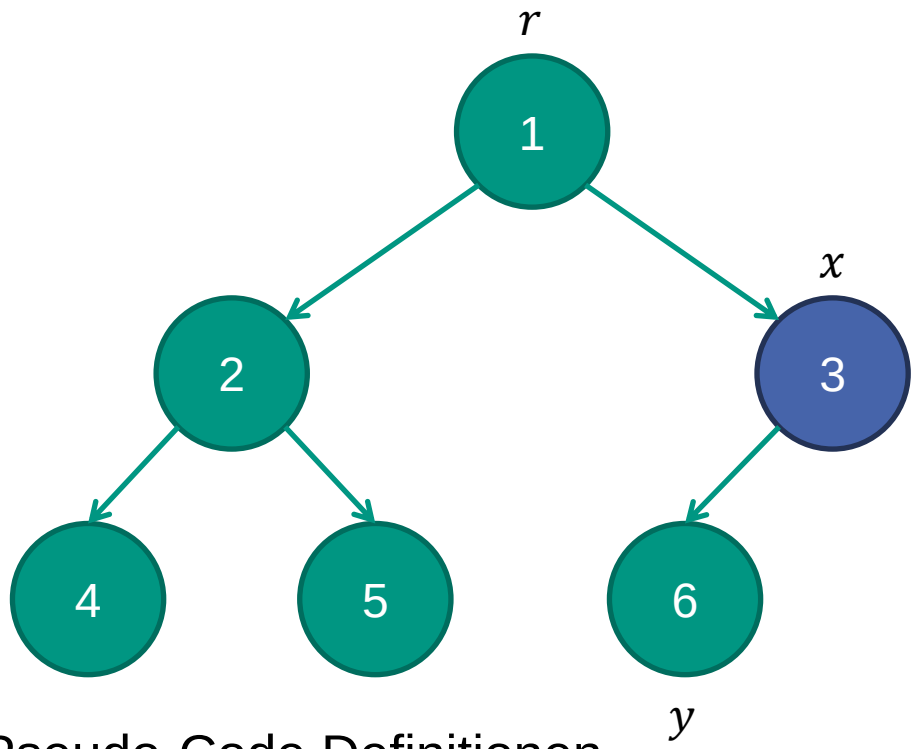
Klassifikation von Suchbäumen

- Anzahl der Kindknoten
  - Binärbaum: Anzahl $0 \leq m \leq 2$
    - Ideal bei Implementierung von Mengen im Hauptspeicher
  - Vielwegbaum: Anzahl beliebig
- Speicherung der Nutzdaten
  - Blattbaum: Nur in den Blattknoten
    - Geeignet für Implementierung von Mengen auf Hintergrundspeicher
  - Natürlicher Baum: Bei jedem Knoten
- Balance
  - Balancierter Baum: Für jeden Knoten unterscheiden sich die Höhen der Unterbäume um maximal 1
    - Geeignet für Implementierung von Mengen auf Hintergrundspeicher
  - Unbalancierter Baum: Es gibt keine derartige Einschränkung

Im Folgenden Beschränkung auf Binäre Suchbäume

Attribute

- Folgende Attribute finden im Pseudo-Code Verwendung
  - Beispiele jeweils auf blauen Knoten rechts bezogen



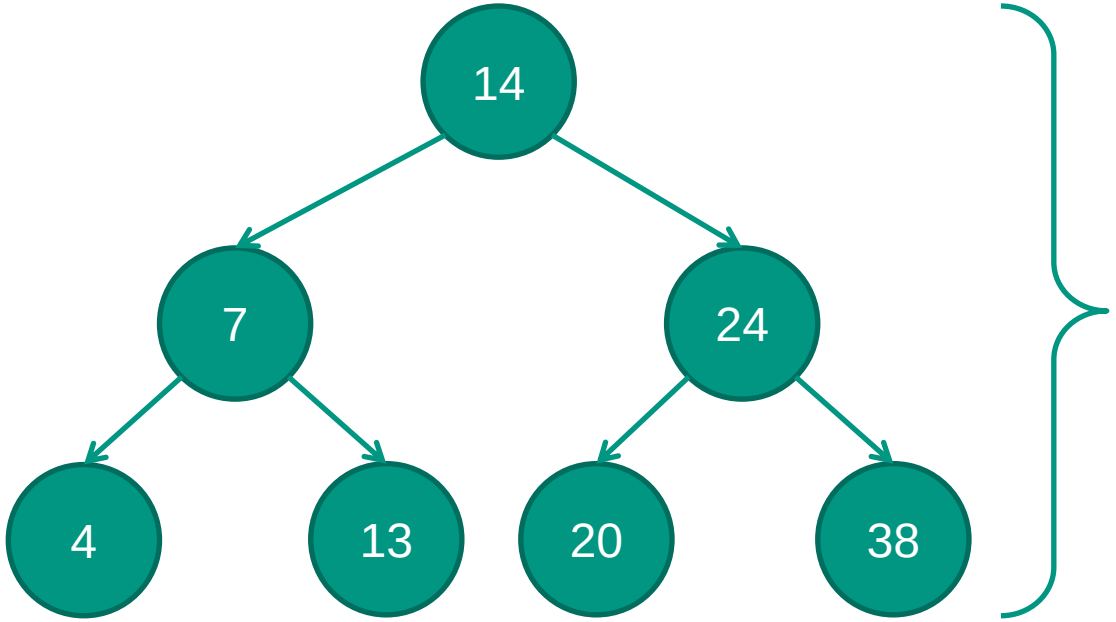
- Schlüssel
  - Im Beispiel $x.schlüssel = 3$
- Vater
  - Im Beispiel $x.vater = r$
- Links
  - Im Beispiel $x.links = y$
- Rechts
  - Im Beispiel $x.rechts = NIL$

- Suchbaumeigenschaft formal mit Pseudo-Code Definitionen
  - $\forall\, y \in linker\ Teilbaum\ von\ x\colon\ y.schlüssel \leq x.schlüssel$
  - $\forall\, y \in rechter\ Teilbaum\ von\ x\colon\ y.schlüssel \geq x.schlüssel$

Beispiel

Binäre Suche

Binärer Suchbaum



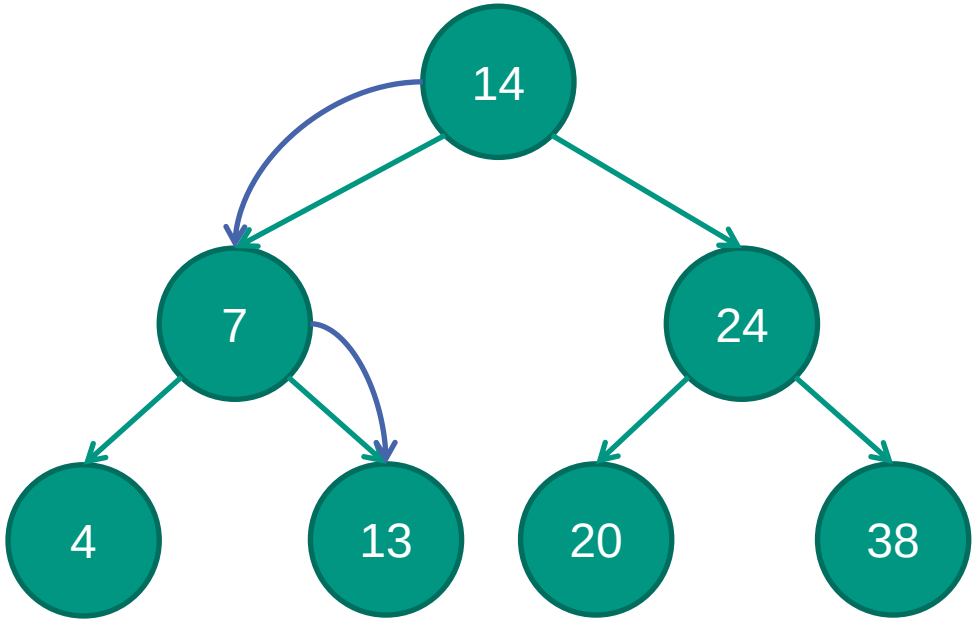
- Höhe h eines Knotens entspricht der Anzahl der Kanten des längsten Pfades zu einem von diesem Knoten aus erreichbaren Blatt
- Höhe h eines Baums ist die Höhe der Wurzel

6.2.1 Suchen eines Elements

- Prinzip
  - Beginn bei der Baum-Wurzel
  - Jeweils in den Teilbaum absteigen, der gemäß Suchbaumeigenschaft das gesuchte Element enthalten muss
  - Entweder wird das Element so gefunden
  - Oder es ist nicht im Baum enthalten

Suchen eines Elements

- Suche „13“



Pseudeocode

- Suche Schlüssel k in Baum mit Wurzel x

- $TREE - SEARCH(x, k)$

```
1    if x == NIL or k == x.schlüssel
2        return x
3    if k < x.schlüssel
4        return TREE − SEARCH(x.links, k)
5    else
6        return TREE − SEARCH(x.rechts, k)
```

Schlüssel gefunden oder x leer

Suche geht im linken oder rechten Teilbaum weiter

- $TREE - MINIMUM(x)$

```
1    while x.links ≠ NIL
2        x = x.links
3    return x
```

Komplexitätsbetrachtung

- $TREE - SEARCH(x, k)$

```
1    if x == NIL or k == x.schlüssel
2            return x
3    if k < x.schlüssel
4            return TREE − SEARCH(x.links, k)
5    else
6            return TREE − SEARCH(x.rechts, k)
```

$O(1)$

$T(\frac{n}{2})$

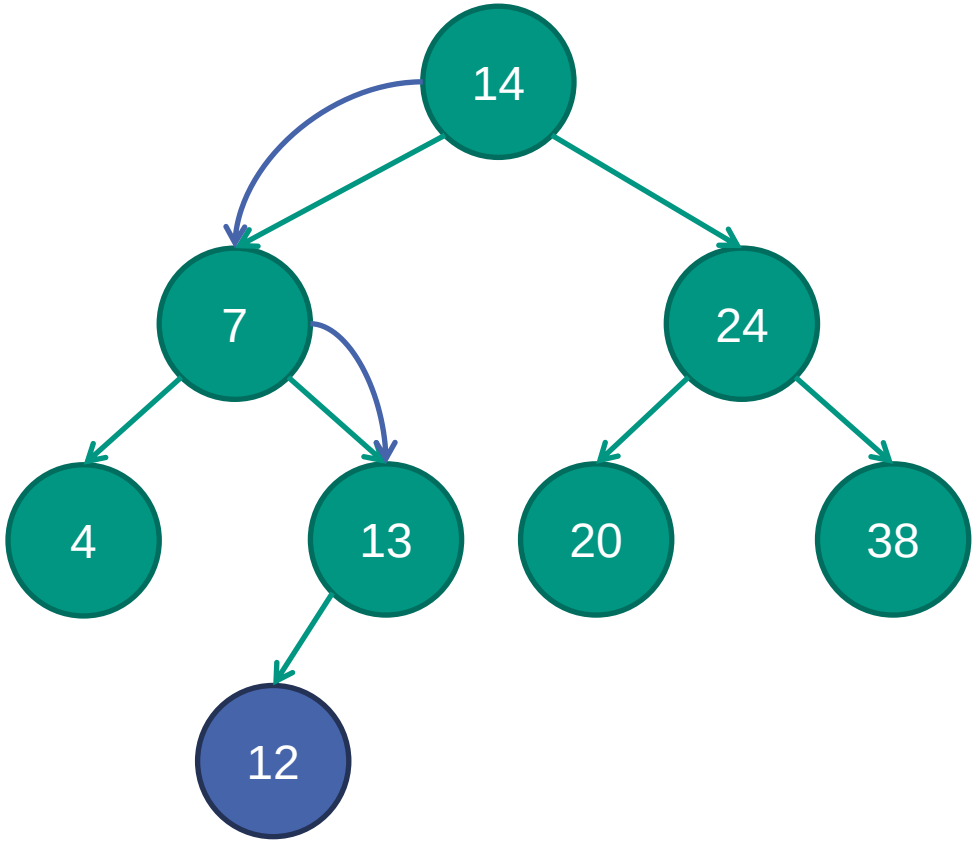
- Im Fall eines balancierten Baums
  - $T(n) = T\left(\frac{n}{2}\right) + O(1)$
  - Nach Master Theorem (Kapitel 2) gilt $T(n) = O(\lg n)$
- Im Worst-Case kann jedoch der rekursive Aufruf von $TREE - SEARCH$ $T(n-1)$ Operationen kosten
  - Insgesamt dann $O(n)$

6.2.2 Einfügen eines Elements

- Prinzip
  - Suche nach der Einfügestelle
  - Suche endet an einem Blattknoten der auf NIL zeigt
  - Hier das einzufügende Element anhängen

Einfügen eines Elements

- Grundidee am Beispiel des Einfügens von „12“



Pseudeocode

- Einfügen eines Knotens z mit Wert $z.schlüssel = v$
- $TREE - INSERT(T, z)$

```
1    y = NIL
2    x = T.wurzel
3    while x ≠ NIL
4        y = x
5        if z.schlüssel < x.schlüssel
6            x = x.links
7        else
8            x = x.rechts
9    z.vater = y
10   if y == NIL
11       T.wurzel = z
12   else if z.schlüssel < y.schlüssel
13       y.links = z
14   else
         y.rechts = z
```

Suche passende Stelle für einzufügendes Element

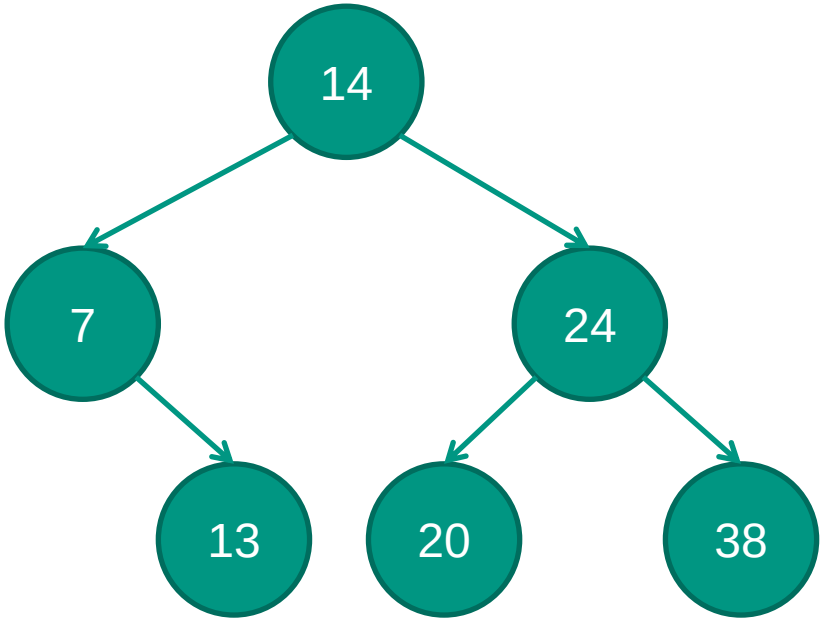
Baum war leer

Element an Blatt links oder rechts anhängen

Pseudeocode am Beispiel

- Einfügen eines Knotens z mit Wert $z.schlüssel = v$
- $TREE - INSERT(T, z)$

```
1    y = NIL
2    x = T.wurzel
3    while x ≠ NIL
4        y = x
5        if z.schlüssel < x.schlüssel
6            x = x.links
7        else
8            x = x.rechts
9    z.vater = y
10   if y == NIL
11       T.wurzel = z
12   else if z.schlüssel < y.schlüssel
13       y.links = z
14   else
         y.rechts = z
```



Komplexitätsbetrachtung

- $TREE - INSERT(T, z)$

```
1    y = NIL                                          O(1)
2    x = T.wurzel                                     O(1)
3    while x ≠ NIL                                    O(h)
4            y = x                                    O(h)
5            if z.schlüssel < x.schlüssel             O(h)
6                    x = x.links                      O(h)
7            else                                     O(h)
8                    x = x.rechts                     O(h)
9    z.vater = y                                      O(1)
10   if y == NIL                                      O(1)
11           T.wurzel = z                             O(1)
12   else if z.schlüssel < y.schlüssel                O(1)
13           y.links = z                              O(1)
14   else                                             O(1)
             y.rechts = z                             O(1)
```

6.2.3 Löschen eines Elements

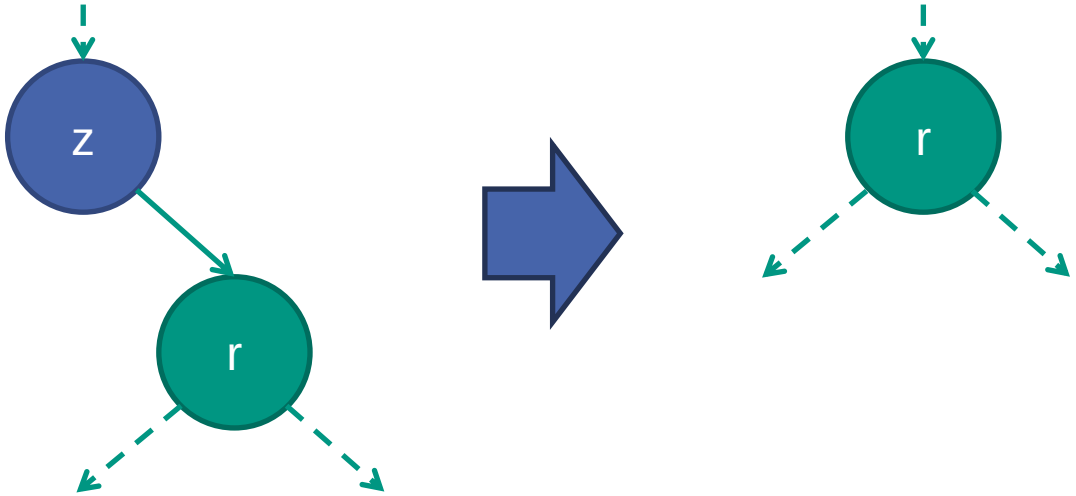
- Drei Basisfälle beim Löschen eines Elementes z

1. Knoten z ist Blattknoten
   - Der Knoten kann einfach entfernt werden
2. Knoten z besitzt genau ein Kind y
   - Der Kindknoten y kann an die Stelle von z gehängt werden
3. Knoten z besitzt genau zwei Kinder x, y
   - Hier kann z nicht einfach durch x oder y ersetzt werden!

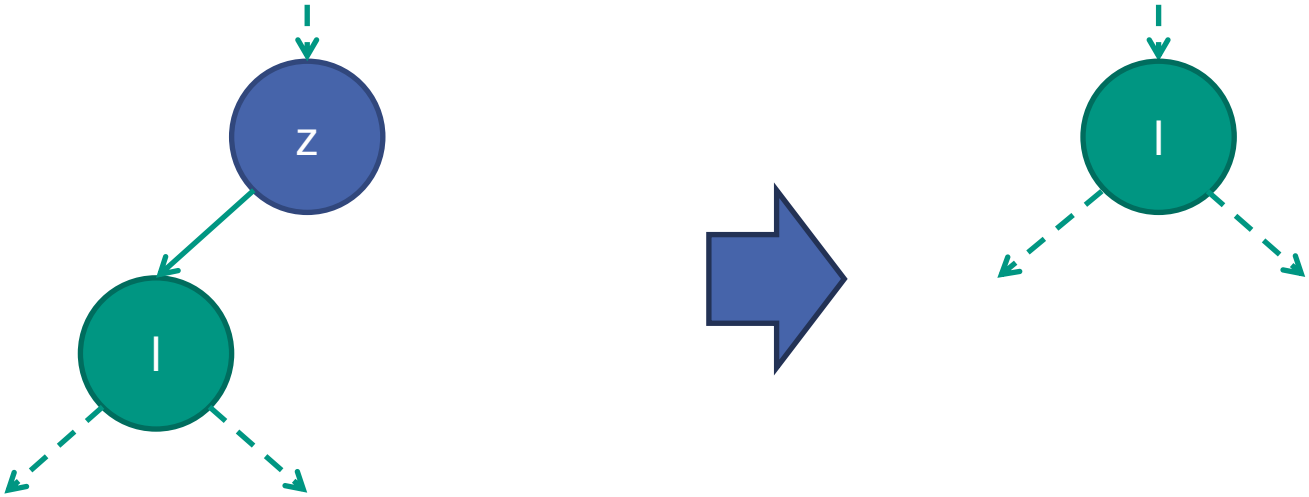
- Fall 2 und 3 genauer

Löschen eines Elements – Fall 2

- Fall 2(a)

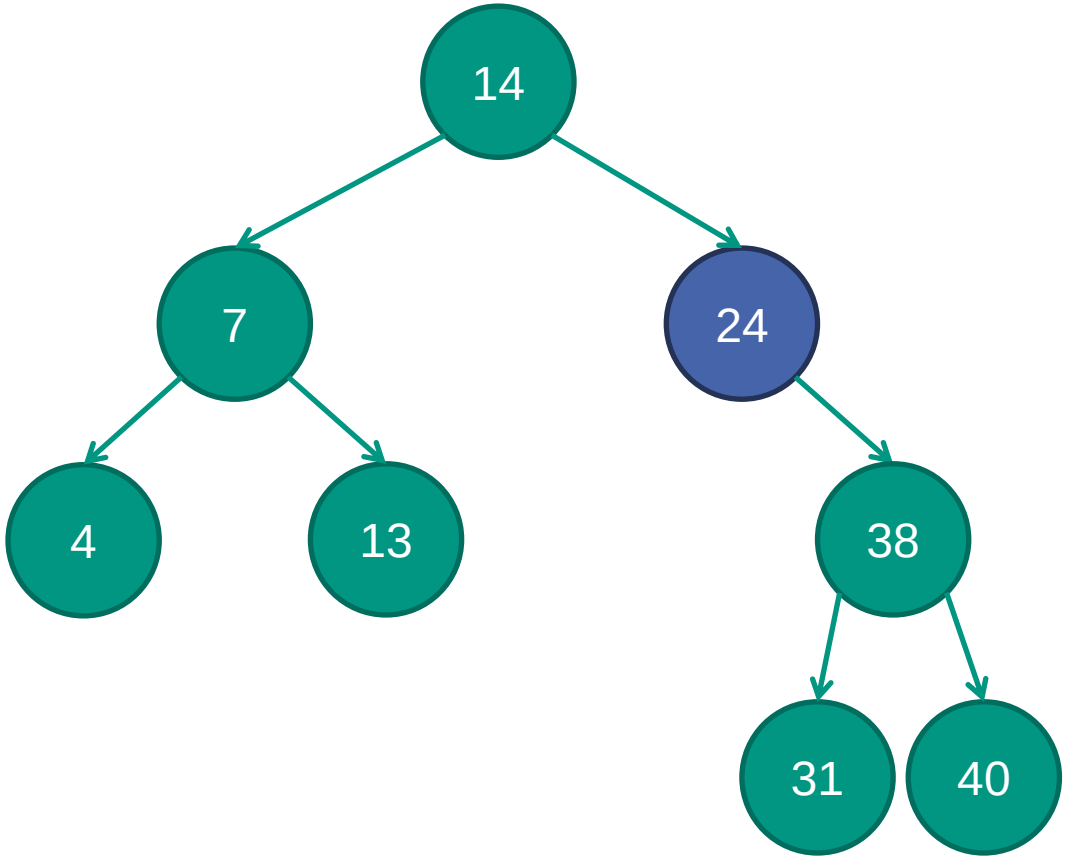


- Fall 2(b)

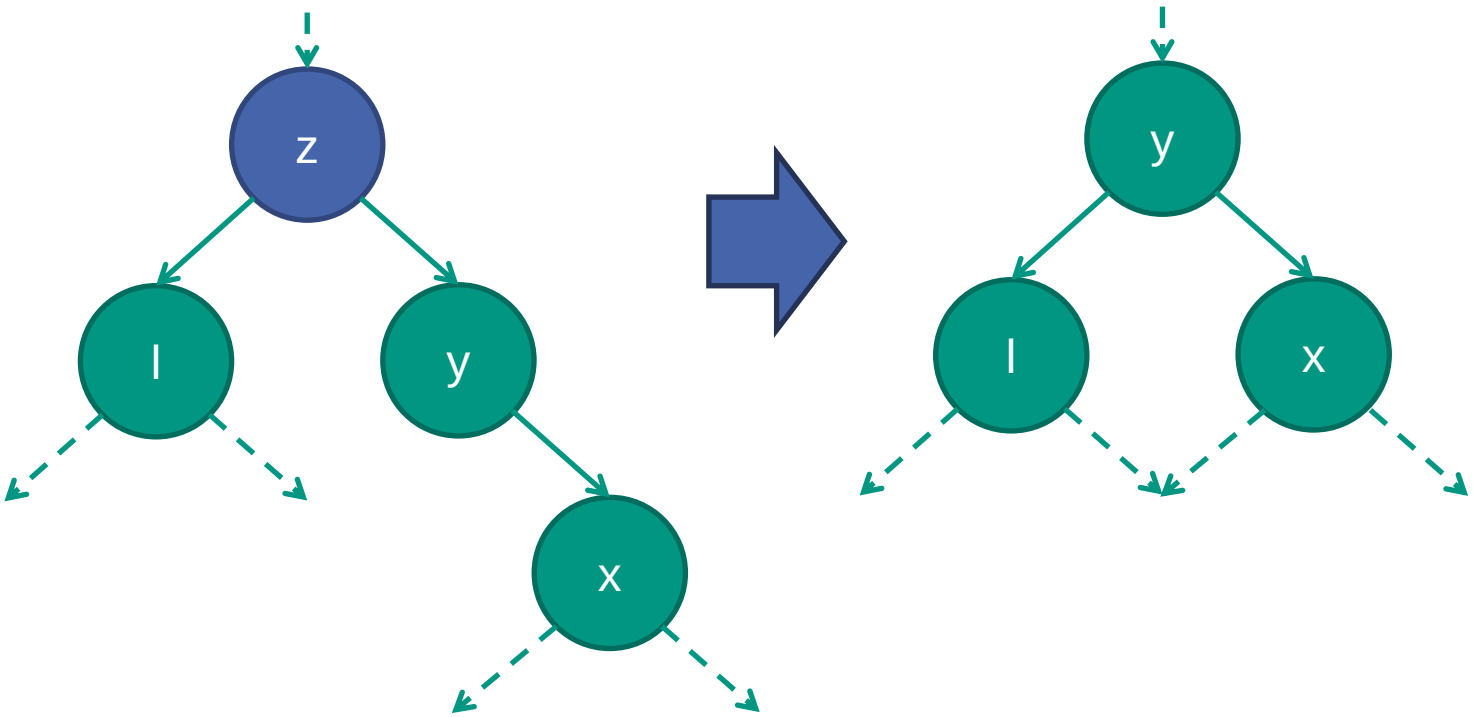


Löschen eines Elements – Beispiel Fall 2

- Löschen von „24“
  - Fall 2(a)



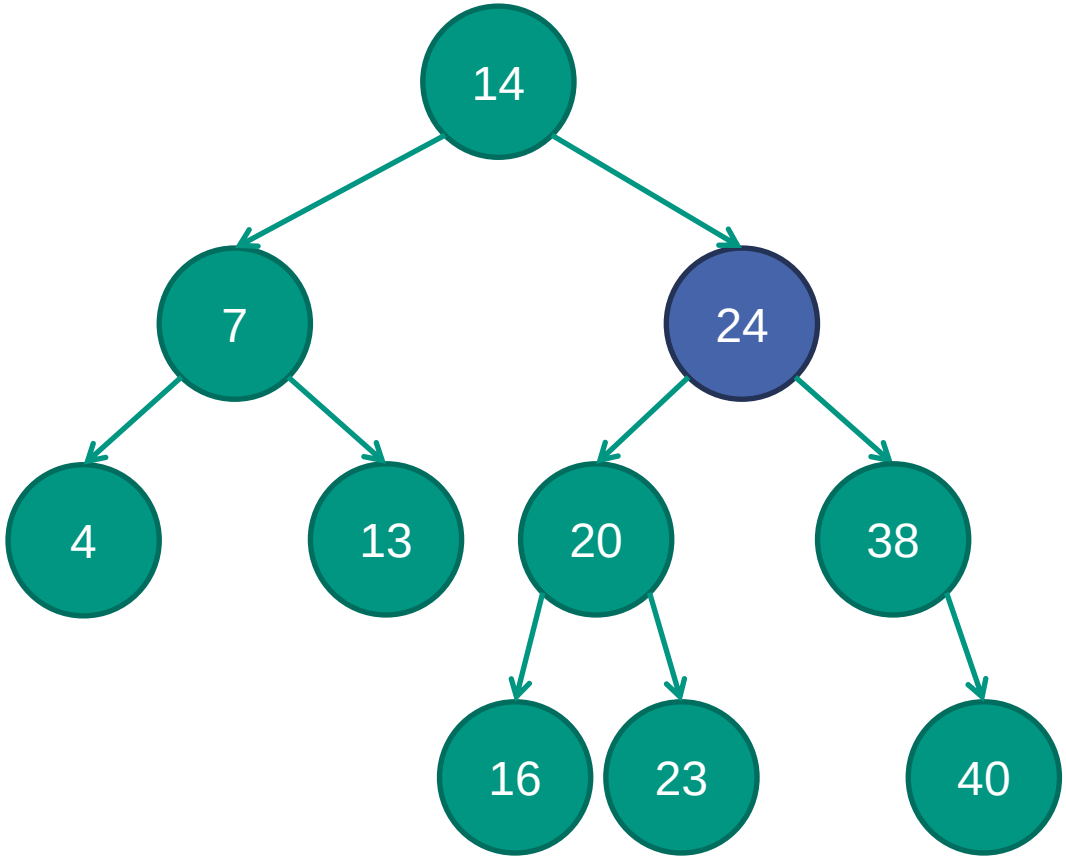
Löschen eines Elements – Fall 3(a)



- Zu Beachten: $y.links == NIL$ sonst lässt sich dieser Fall nicht anwenden
  - Sonst: Fall 3(b)

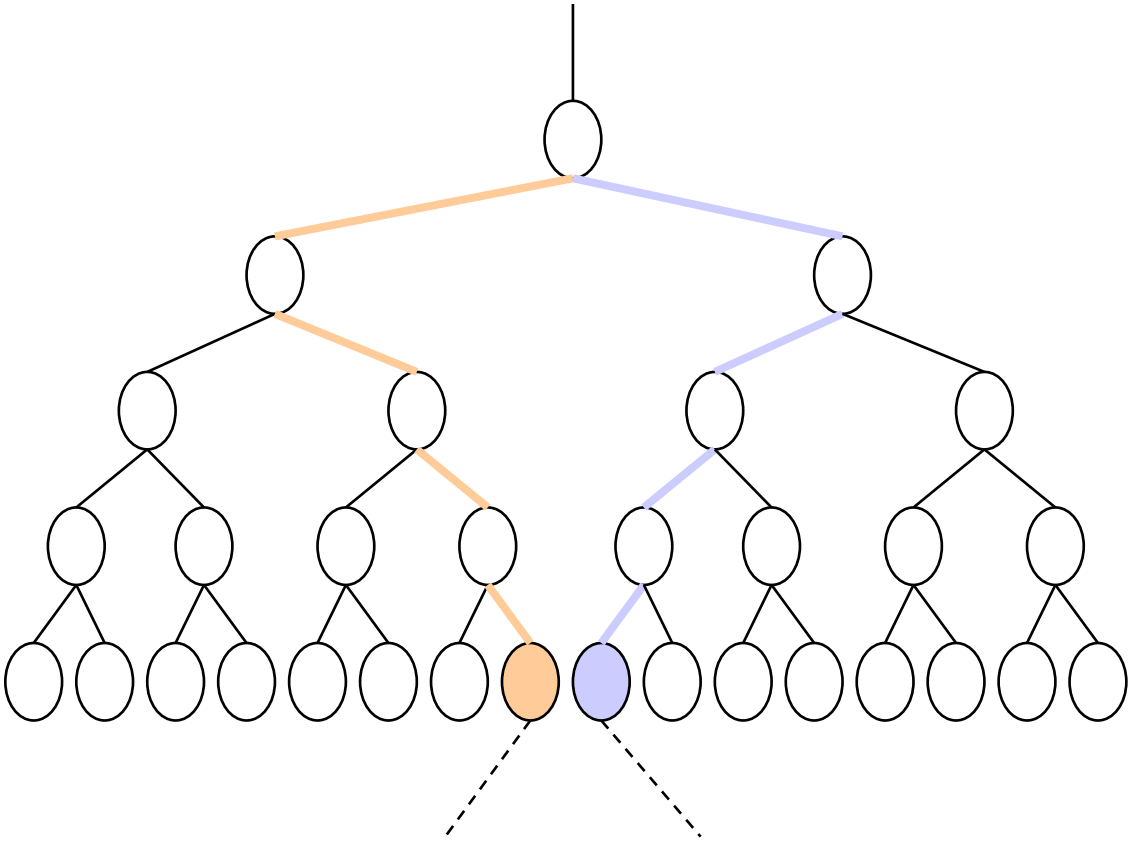
Löschen eines Elements – Beispiel Fall 3(a)

- Löschen von „24“
  - Fall 3(a)

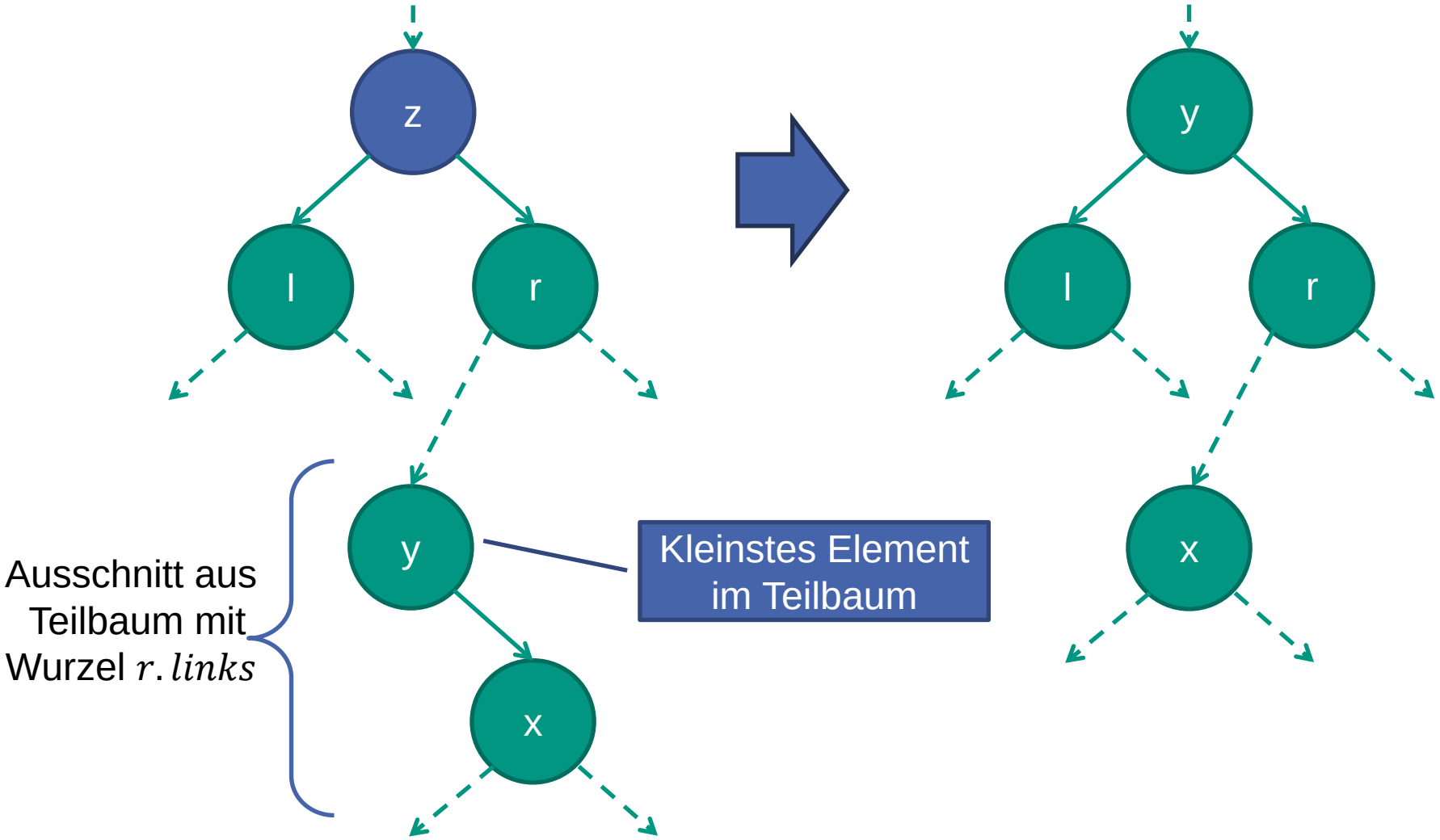


Löschen eines Elements – Fall 3(b)

- In allen anderen Fällen
  - Ersetze zu löschenden Knoten durch größten Knoten im linken Teilbaum oder kleinsten Knoten im rechten Teilbaum (der hat max. 1 Nachfolger)

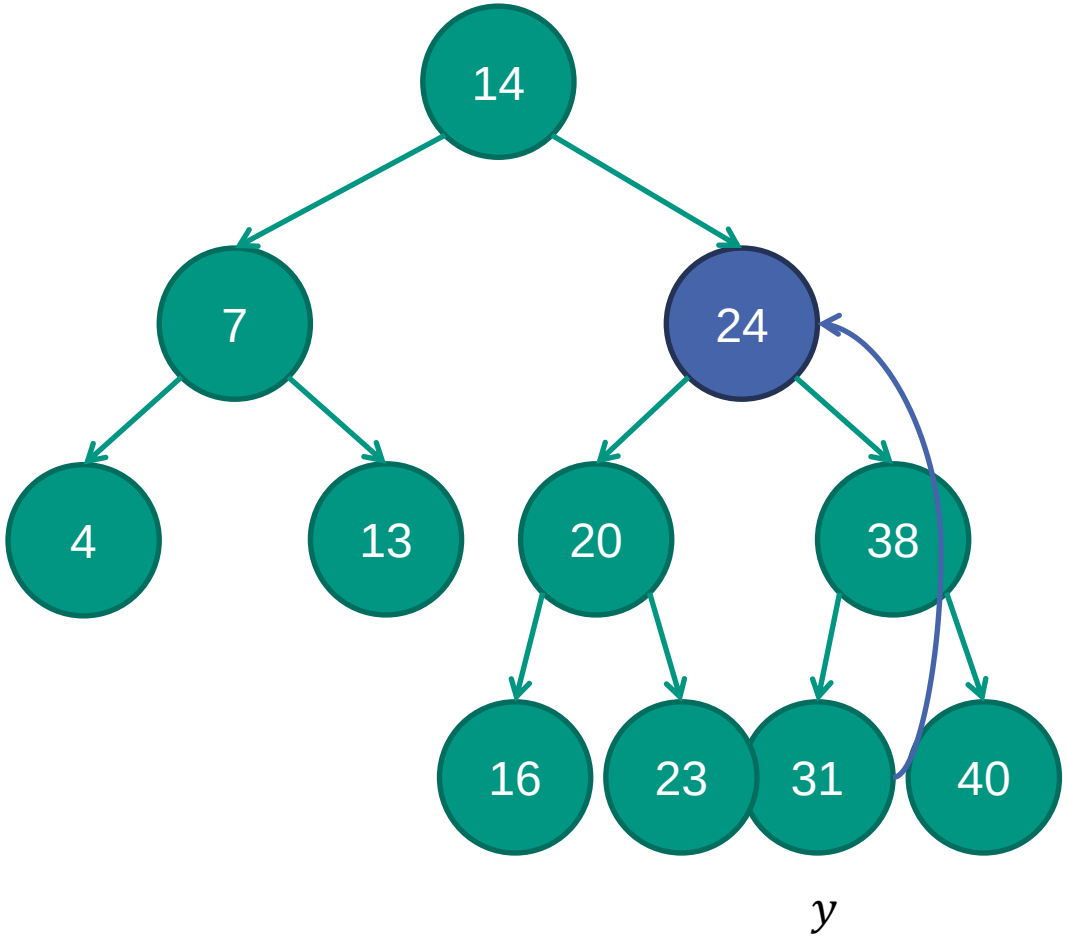


Löschen eines Elements – Fall 3(b)



Löschen eines Elements – Beispiel Fall 3(b)

- Löschen von „24“
  - Fall 3(b)



Löschen eines Elements – Pseudeocode

- $TREE - DELETE(T, z)$

```
1   if z.links == NIL
2       TRANSPLANT(T, z, z.rechts)
3   elseif z.rechts == NIL
4       TRANSPLANT(T, z, z.links)
5   else
6       y = TREE − MINIMUM(z.rechts)
7       if y.vater ≠ z
8           TRANSPLANT(Z, y, y.rechts)
9           y.rechts = z.rechts
10          y.rechts.vater = y
11      TRANSPLANT(T, z, y)
12      y.links = z.links
13      y.links.vater = y
```

Fall 1 und 2(a)

Fall 2(b)

Fall 3

- $TRANSPLANT(T, u, v)$ hängt den Teilbaum mit Wurzel v an Stelle von u

Löschen eines Elements – Pseudeocode

- $TRANSPLANT(T, u, v)$

```
1  if u.vater == NIL
2      T.wurzel = v
3  elseif u == u.vater.links
4      u.vater.links = v
5  else
6      u.vater.rechts = v
7  if v ≠ NIL
8      v.vater = u.vater
```

u ist Wurzel von T

u ist linkes Kind

u ist rechtes Kind

Komplexitätsbetrachtung

- $TREE - DELETE(T, z)$

```
1   if z.links == NIL
2       TRANSPLANT(T, z, z.rechts)
3   elseif z.rechts == NIL
4       TRANSPLANT(T, z, z.links)
5   else
6       y = TREE − MINIMUM(z.rechts)
7       if y.vater ≠ z
8           TRANSPLANT(Z, y, y.rechts)
9           y.rechts = z.rechts
10          y.rechts.vater = y
11      TRANSPLANT(T, z, y)
12      y.links = z.links
13      y.links.vater = y
```

$O(1)$

$O(h)$

$O(1)$

- Gesamtkomplexität in $O(h)$
  - h Höhe des Baumes

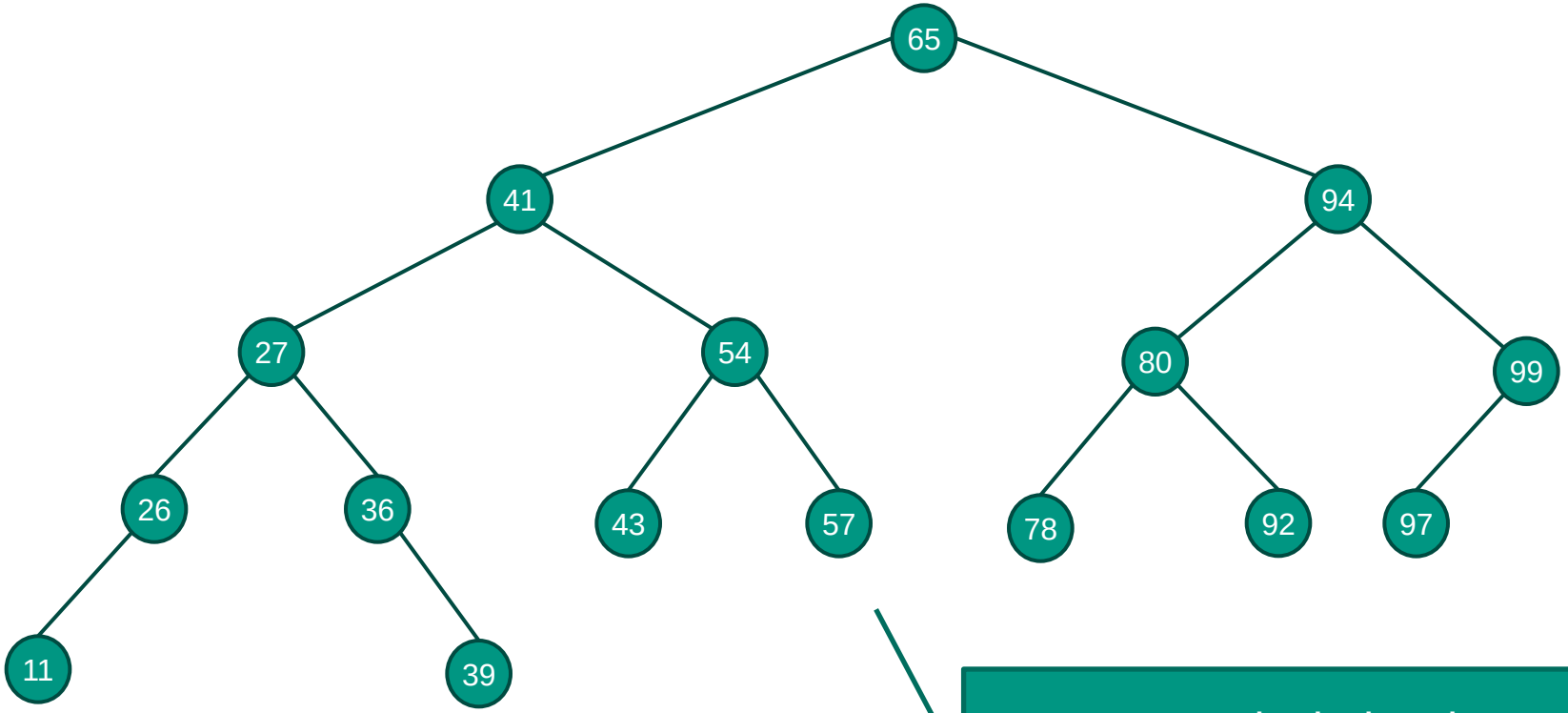
6.2.4 Zwischenfazit

- Suchen
  - Balancierter Suchbaum spiegelt die Binärsuche strukturell wider
  - Suchen in $O(h) = O(\lg n)$ bei balancierten Bäumen
- Einfügen
  - Einfügen in $O(h) = O(\lg n)$ bei balancierten Bäumen
- Löschen
  - Löschen in $O(h) = O(\lg n)$ bei balancierten Bäumen

- → Balance entscheidendes Kriterium für Effizienz

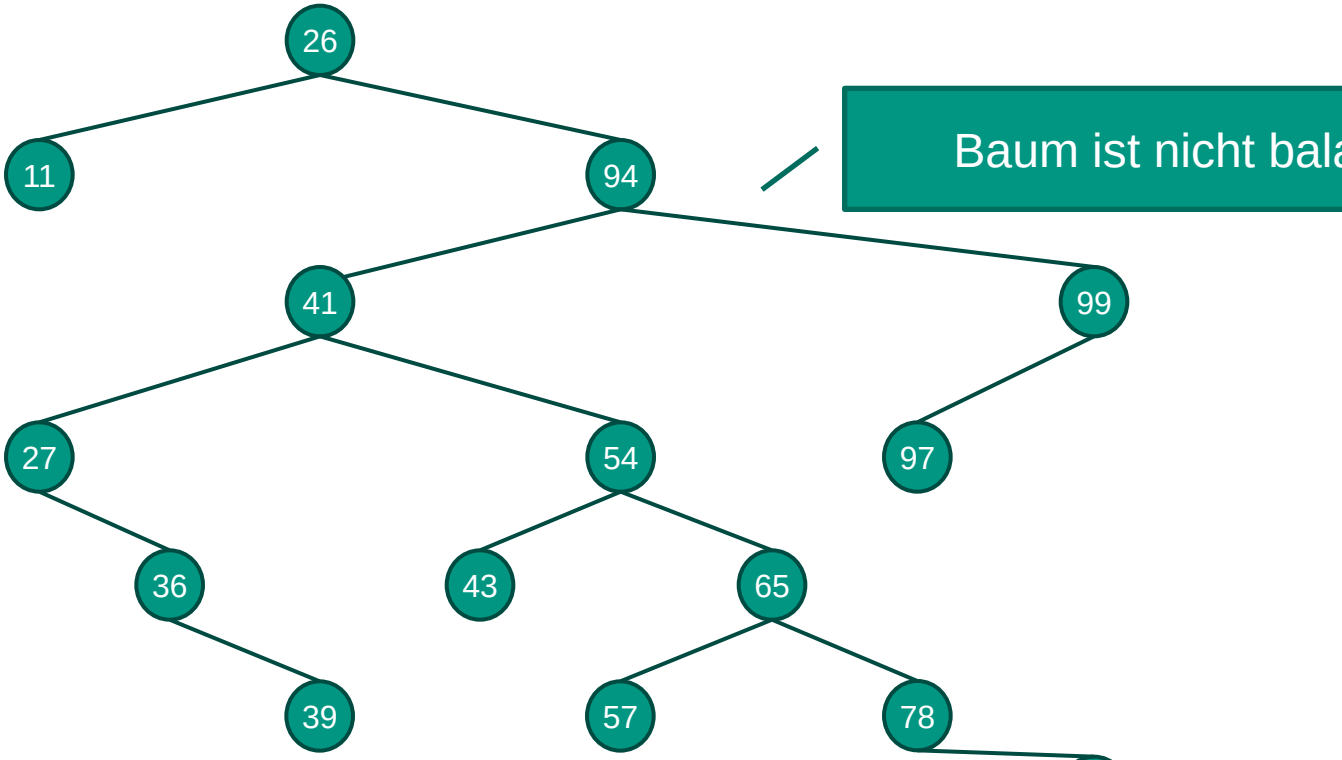
6.2.5 Balance

- Ist ein Baum immer ausbalanciert?
- Beispiel: Einfüge-Reihenfolge
  - 65, 41, 54, 94, 57, 99, 43, 27, 97, 26, 80, 92, 11, 36, 39, 78



Balance

- Andere Reihenfolge der Eingabe
  - 26, 94, 41, 54, 99, 65, 57, 78, 80, 27, 97, 36, 11, 43, 39, 92

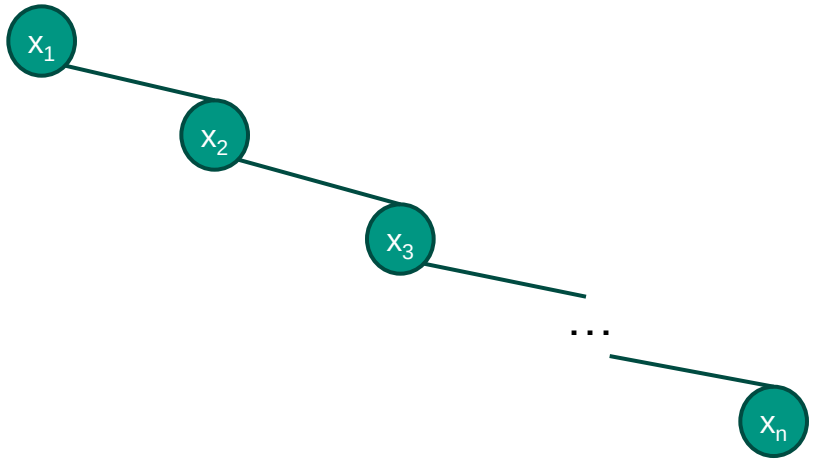


Baum ist nicht balanciert

Balance hängt von Eingabedaten und Reihenfolge ab

Balance

- Binäre Suchbäume sind nicht balanciert
  - Z.B. Einfügen einer aufsteigend sortierten Folge x_1, x_2 , x_3, ... x_n liefert



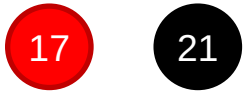
- Aufwand steigt also von $O(\lg n)$ auf $O(n)$ im schlechtesten Fall!

6.2.6 Zwischenfazit

- Suchbäume einfache Datenstruktur 
  - Effizienz abhängig von Balance
- Suchbäume bieten keine Möglichkeit Balance zu kontrollieren
  - → Erweiterungen notwendig
  - Rot-Schwarz-Bäume Abschnitt 6.3
- Suchbäume eigenen sich vor allem für schnellen Arbeitsspeicher
  - Optimierungen für Hintergrundspeicher (Festplatte etc.) notwendig
  - B-Bäume Abschnitt 6.4

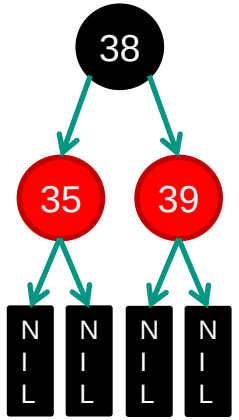
6.3 Rot-Schwarz Bäume

- Binärer Suchbaum
- Zusätzliches Knoten-Attribut $farbe$
  - 0 = Schwarz
  - 1 = Rot



- Berücksichtigung der Balance
  - Jeder Pfad von der Wurzel zu einem Blatt ist maximal doppelt so lang wie jeder andere
  - Also annährend balanciert

- Nicht vorhandener Kind-Knoten
  - Speziellen NIL-Knoten als Kind einfügen, dessen Wert das NIL-Objekt repräsentiert
    - Kann wie normaler Knoten behandelt werden
  - Bislang durch Setzen der Attribute rechts / link auf NIL repräsentiert

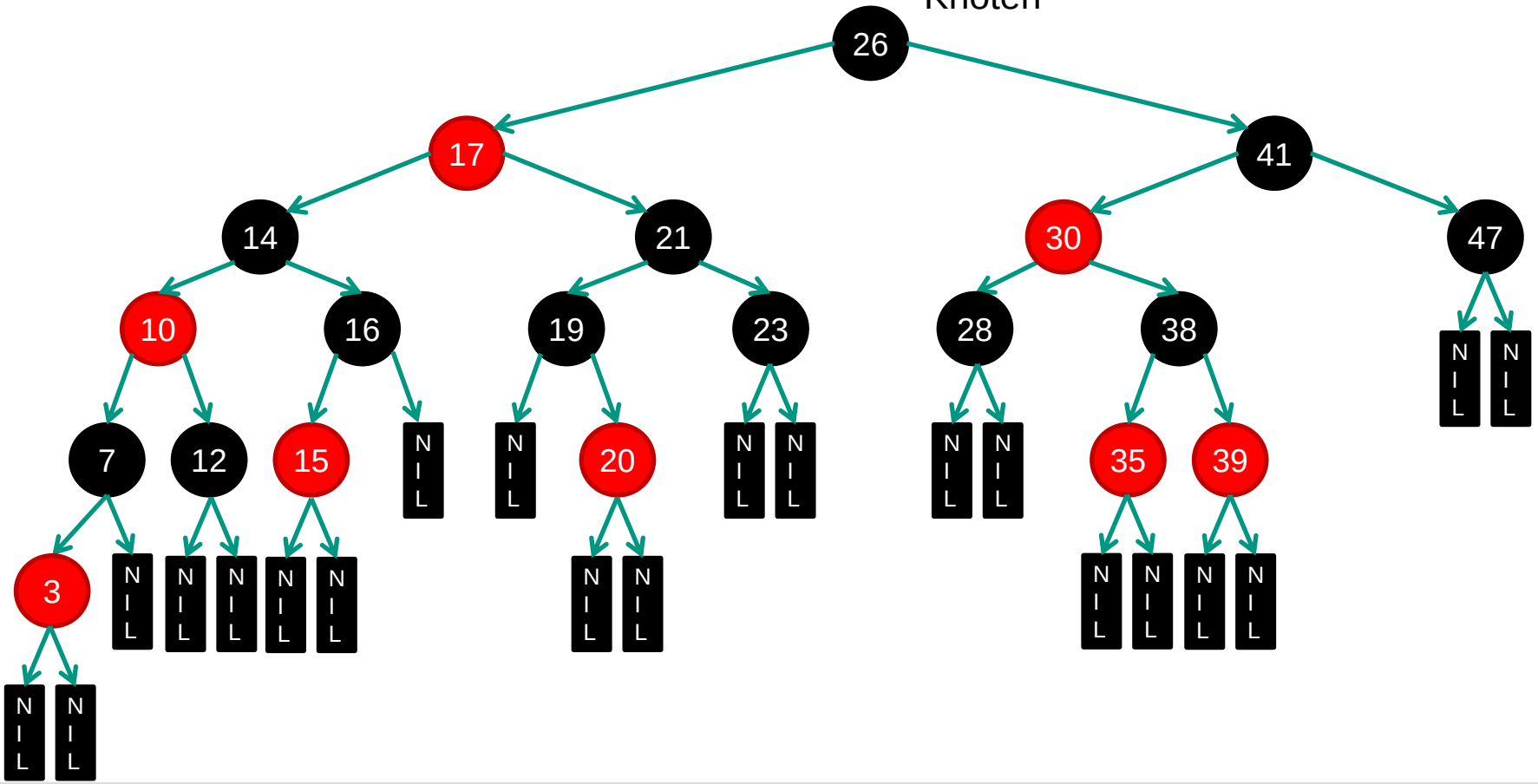


Eigenschaften Rot-Schwarz Baum

1. Jeder Knoten ist entweder rot oder schwarz
2. Die Wurzel ist schwarz
3. Jedes Blatt NIL ist schwarz
4. Wenn ein Knoten rot ist, so sind beide Kinder schwarz
5. Für jeden Knoten gilt, dass alle Pfade vom Knoten zu einem Blatt die selbe Anzahl schwarzer Knoten beinhalten

Eigenschaften Rot-Schwarz Baum – Beispiel

1. Rot oder schwarz
2. Die Wurzel ist schwarz
3. Blätter schwarz
4. Knoten rot → beide Kinder schwarz
5. Für alle Knoten: alle Pfade vom Knoten zu einem Blatt, selbe Anzahl schwarzer Knoten



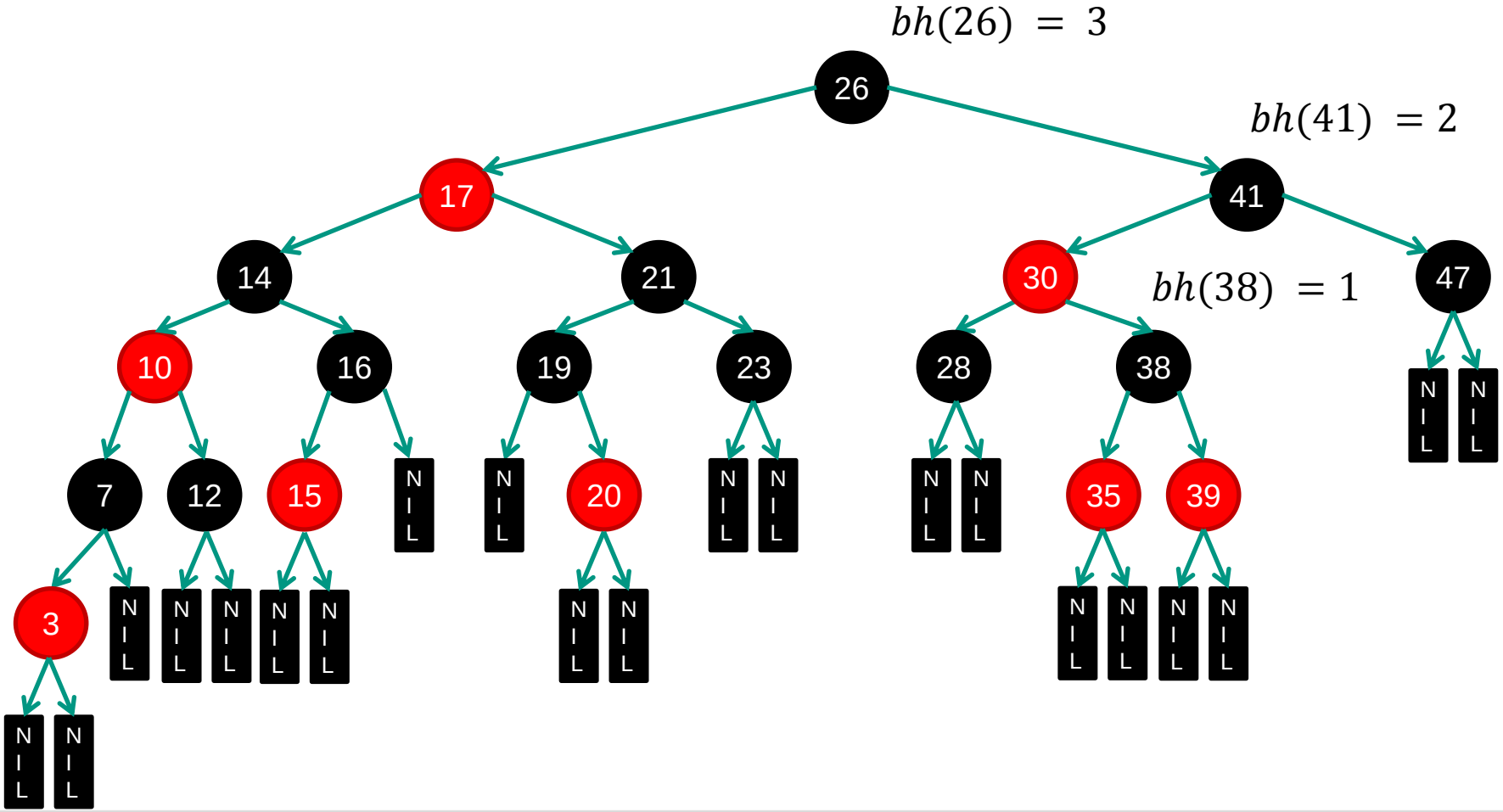
6.3.1 Schwarz-Höhe

- Definition
  - Schwarz-Höhe (Black-Height) eines Knotens k: $bh(k)$
    - Anzahl der schwarzen Knoten auf beliebigen Pfaden von k zu einem Blatt
      Hierbei wird k selbst nicht mitgezählt
  - Schwarz-Höhe eines Baums
    - Schwarz-Höhe der Wurzel w: $bh(w)$

- Anmerkung
  - Die Schwarz-Höhe-Eigenschaft ist aufgrund der Eigenschaft (5) wohl definiert

Schwarz-Höhe – Beispiel

- Schwarz-Höhe der Knoten 38, 41, 26



Höhe eines Rot/Schwarz-Baums

- Satz
  - Die Höhe eines Rot/Schwarz-Baums mit n inneren Knoten beträgt maximal $2\lg(n+1)$

$$h \leq 2\lg(n+1)$$

- Beweis
  - Jeder durch einen Knoten x aufgespannte Teilbaum hat wenigstens $2^{bh(x)} - 1$ innere Knoten
- Vollständige Induktion über die Höhe h von x
  - Induktionsanfang
    $h = 0 \Rightarrow x$ ist ein Blatt (=NIL-Knoten) und der durch x aufgespannte Teilbaum enthält $2^{bh(x)} - 1 = 2^0 - 1 = 0$ innere Knoten
  - Induktionsannahme
    Ein durch x aufgespannter Teilbaum der Höhe h hat wenigstens $2^{bh(x)} - 1$ innere Knoten

Höhe eines Rot/Schwarz-Baums

- Induktionsschritt
  Ein durch x aufgespannter Teilbaum der Höhe $h + 1$ hat wenigstens $2^{bh(x)} - 1$ innere Knoten

- Knoten x sei innerer Knoten und habe die Höhe $h + 1$ und zwei Kinder

- Jedes Kind hat die Schwarz-Höhe $bh(x)$ oder $bh(x) - 1$ abhängig davon, ob seine Farbe Rot oder Schwarz ist

- Unter Verwendung der Induktionsannahme
  Jedes Kind von x hat mindestens $2^{bh(x)-1} - 1$ innere Knoten

- Der durch x aufgespannte Teilbaum hat mindestens $\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$ innere Knoten

Höhe eines Rot/Schwarz-Baums

- Bisher bewiesen
  - Ein durch x aufgespannter Teilbaum der Höhe $h + 1$ hat wenigstens $2^{bh(x)} - 1$ innere Knoten

- Sei nun h die Höhe des Baums entsprechend Eigenschaft (4) müssen mindestens die Hälfte aller Knoten auf allen Pfaden von der Wurzel zu einem Blatt (ausschließlich der Wurzel) schwarz gefärbt sein
  - Die Black-Height der Wurzel muss mindestens $\frac{h}{2}$ betragen

$$n \geq 2^{\frac{h}{2}} - 1 \leftrightarrow n + 1 \geq 2^{\frac{h}{2}} \leftrightarrow \lg(n+1) \geq \frac{h}{2} \leftrightarrow h \leq 2\lg(n+1)$$

q.e.d.

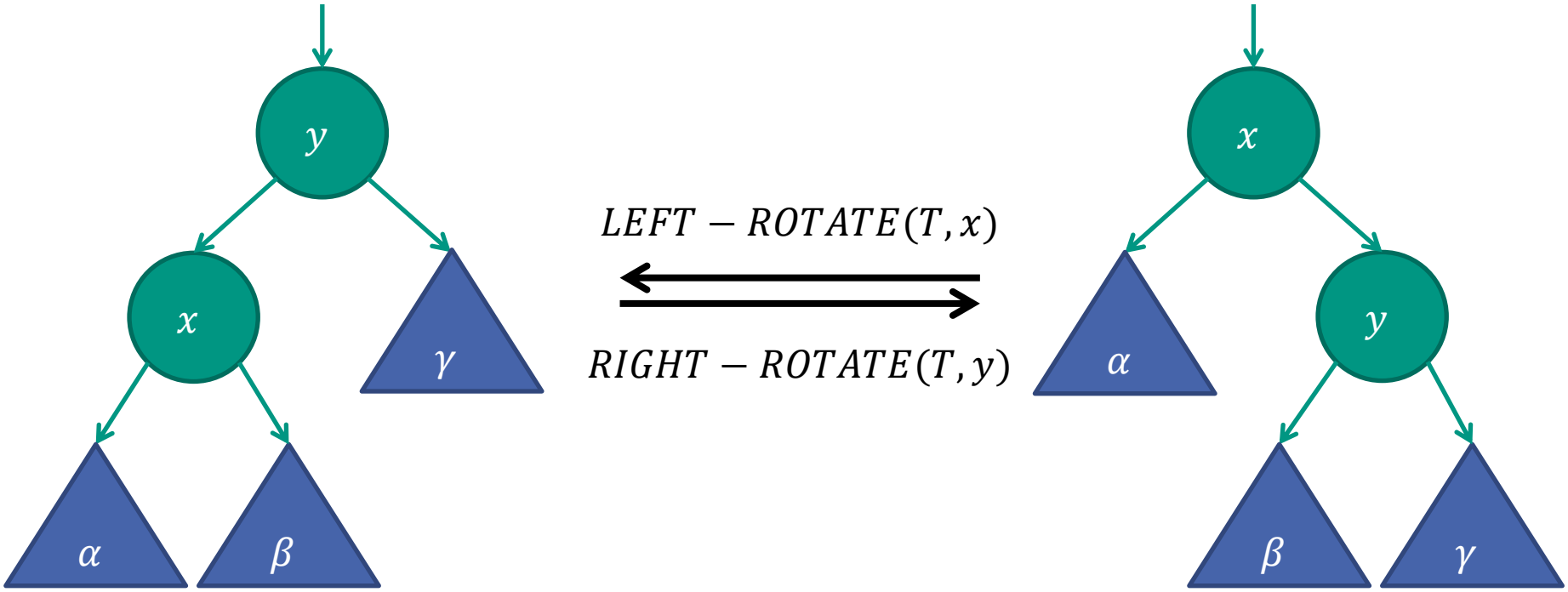
6.3.2 Rotationen

- Einfüge- und Löschoperationen auf Rot/Schwarz-Bäumen können Eigenschaften eines Rot-Schwarz Baums verletzen

- Wiederherstellen der 5 Baum-Eigenschaften durch
  - Umfärben bestimmter Knoten
  - Höhe des Baumes verändern → Rotationen

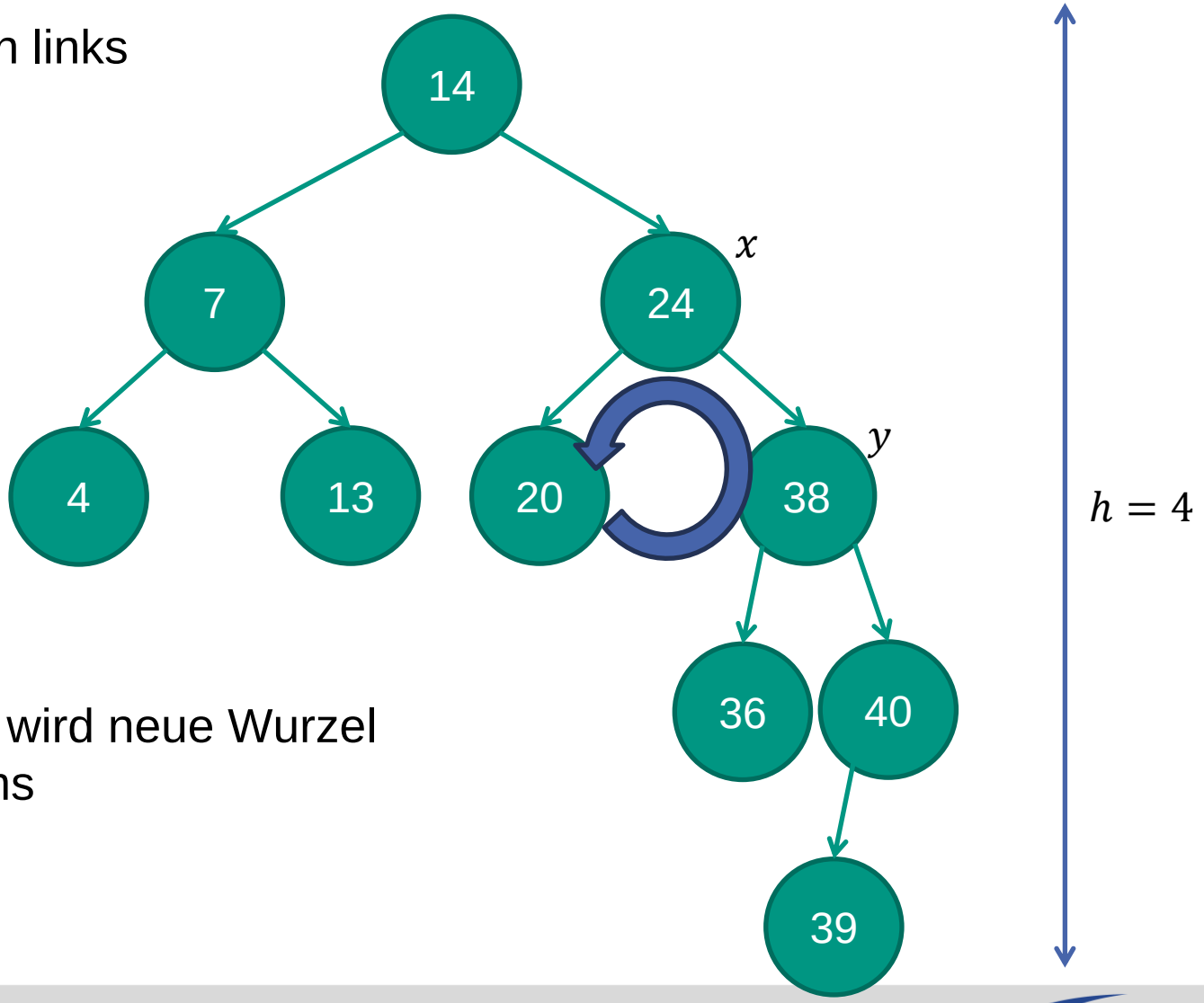
- Rotation
  - Lokale Operation auf Teil des Baumes
  - Rechts- und Linksrotation möglich
  - Erhält die Suchbaum-Eigenschaft

Rotationen – Prinzip



Beispiel – Linksrotation

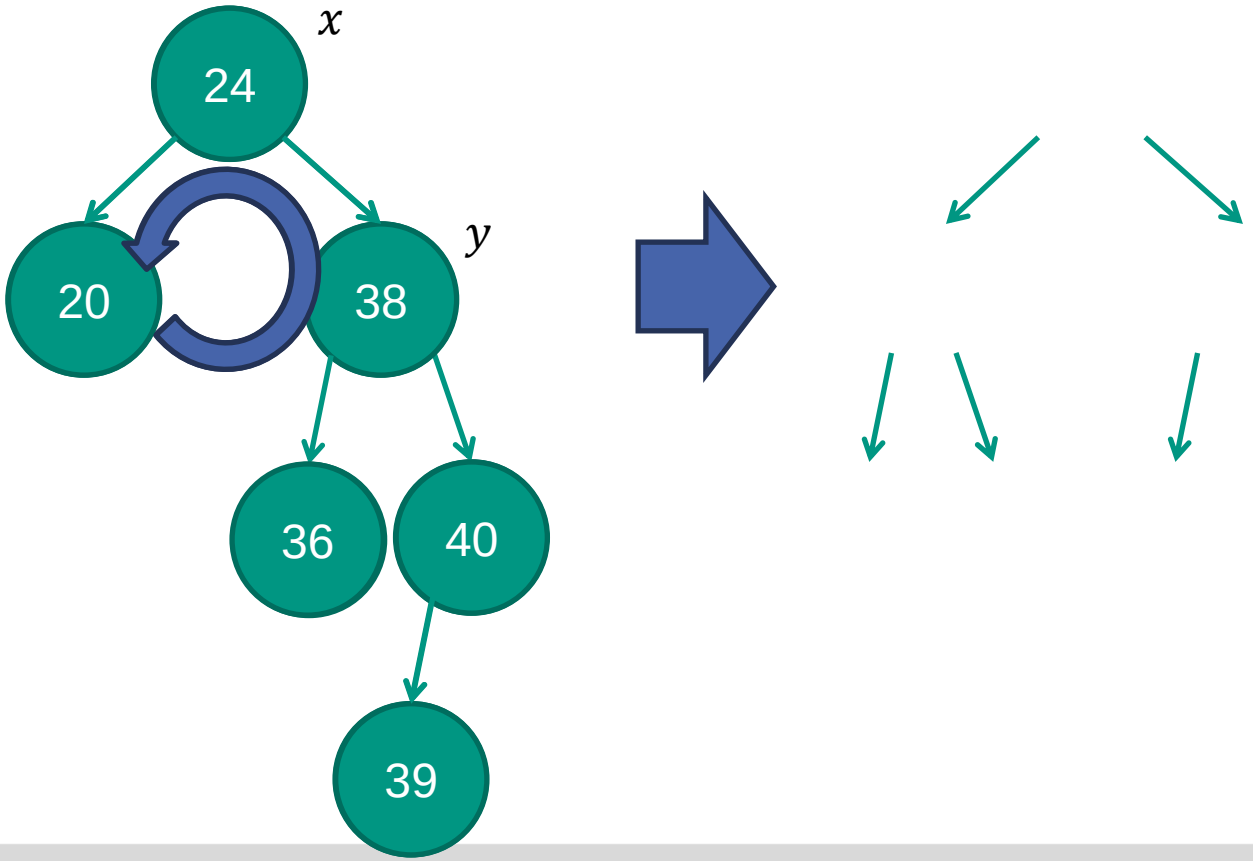
- Rotation nach links am Knoten x



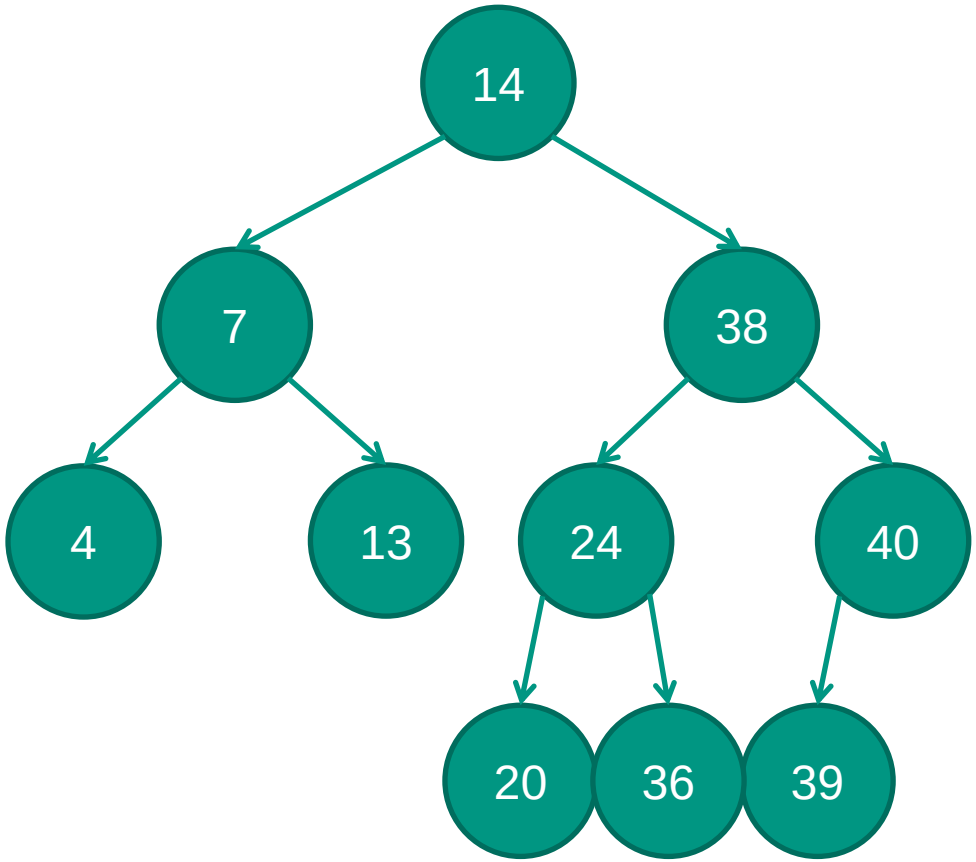
- $y = x.rechts$ wird neue Wurzel des Teilbaums

Beispiel – Linksrotation

- Linksrotation Schritt für Schritt



Beispiel – Linksrotation



$h = 3$

- Ergebnis
  - Die Höhe des Baumes ist um 1 reduziert worden

Pseudocode – Linksrotation

```
LEFT − ROTATE(T, x)
1    y = x.rechts
2    x.rechts = y.links
3    if y.links ≠ NIL
4        y.links.vater = x
5    y.vater = x.vater
6    if x.vater == NIL
7        T.wurzel = y
8    elseif x == x.vater.links
9        x.vater.links = y
10   else
11       x.vater.rechts = y
12   y.links = x
13   x.vater = y
```

Bestimme y

y‘s linker Teilbaum → x‘s rechter

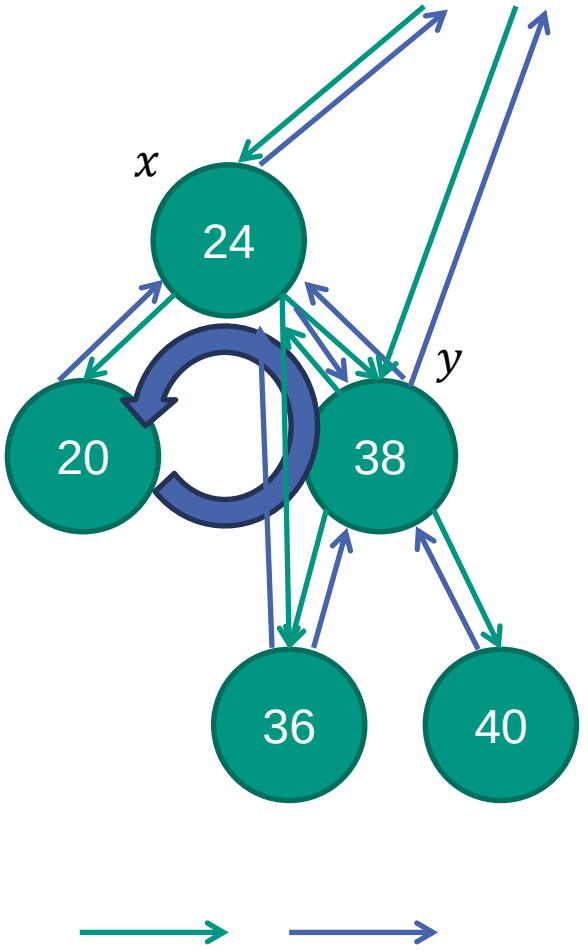
y‘s Vater → x‘s Vater

x wird y‘s linkes Kind

Pseudocode – Linksrotation

- $LEFT-ROTATE(T,x)$

```
1   y = x.rechts
2   x.rechts = y.links
3   if y.links ≠ NIL
4       y.links.vater = x
5   y.vater = x.vater
6   if x.vater == NIL
7       T.wurzel = y
8   elseif x == x.vater.links
9       x.vater.links = y
10  else
11      x.vater.rechts = y
12  y.links = x
13  x.vater = y
```



Komplexitätsbetrachtung – Linksrotation

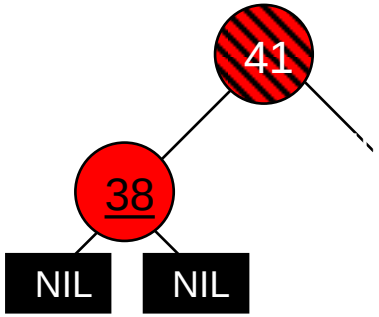
- $LEFT - ROTATE(T, x)$

```
1   y = x.rechts
2   x.rechts = y.links
3   if y.links ≠ NIL
4       y.links.vater = x
5   y.vater = x.vater
6   if x.vater == NIL
7       T.wurzel = y
8   elseif x == x.vater.links
9       x.vater.links = y
10  else
11      x.vater.rechts = y
12  y.links = x
13  x.vater = y
```

$O(1)$

- Alle Operationen laufen in konstanter Zeit

6.3.3 Einfügen eines Elements



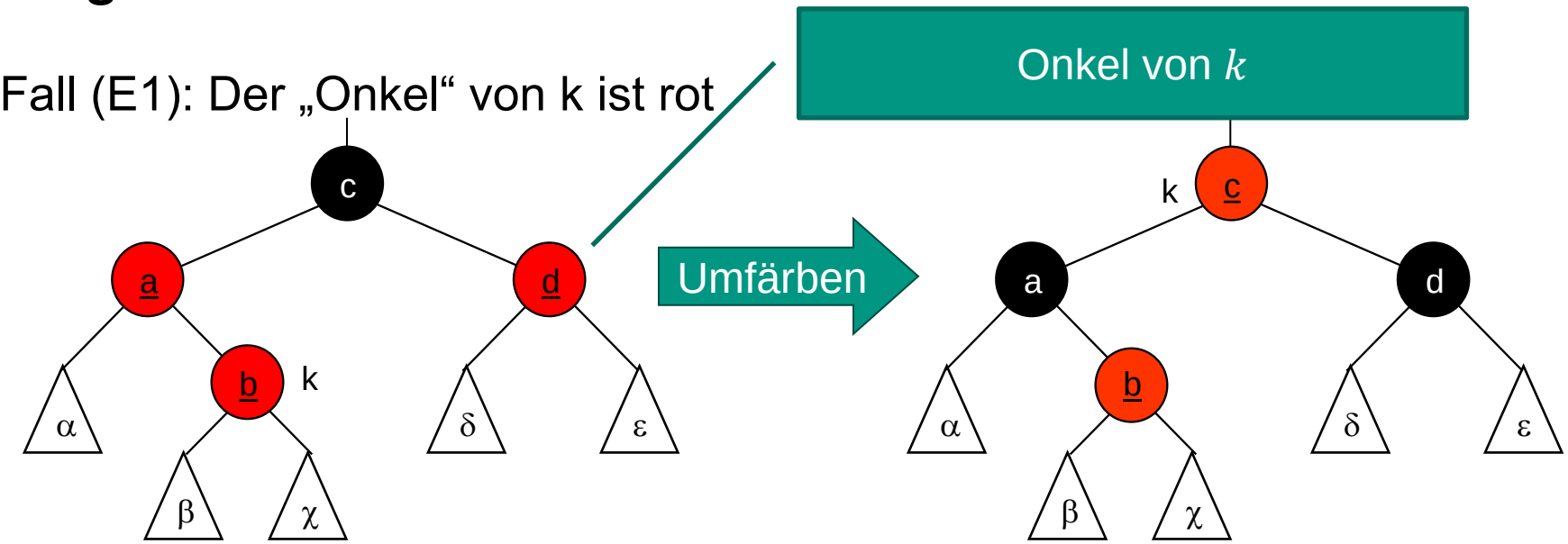
- Vorgehensweise
  - Rot-färben des eingefügten Elements k
  - Ergänzung zweier NIL-Knoten als Kinder
- Überprüfen der Eigenschaften
  - (1) Jeder Knoten ist entweder rot oder schwarz. → Nicht verletzt
  - (2) Die Wurzel ist schwarz
    - Verletzt nur, wenn k in leeren Baum eingefügt wird → k Schwarz färben
  - (3) Jedes Blatt (NIL) ist schwarz. → Nicht verletzt
  - (4) Wenn ein Knoten rot ist, so sind beide Kinder schwarz
    - Nicht durch k verletzt, da beide Kinder schwarze NIL-Knoten
    - Verletzt, falls k als Kind eines roten Vater-Knotens eingefügt wird
  - (5) Für jeden Knoten gilt, dass alle Pfade vom Knoten zu einem Blatt die selbe Anzahl schwarzer Knoten beinhalten
    - Da nur ein roter Knoten hinzukommt, ist diese Eigenschaft ebenfalls nicht verletzt

Einfügen eines Elements

- Maßnahmen, um Eigenschaft (4) wieder herzustellen
  - Geeignete Links/Rechts-Rotation zwecks Höhenausgleich
    - Einfärbung der Knoten gibt Aufschluss über die notwendigen Rotationen
  - Korrektur der Einfärbung der falsch eingefärbten Knoten
- Einfärbung wird in Richtung der Wurzel korrigiert
  - Funktion zum Zugriff auf den Vaterknoten eines Knotens k nötig: $k.vater$
  - In folgenden Algorithmen muss $k.vater$ entsprechend berücksichtigt und ggf. aktualisiert werden!
- Sechs Fälle sind zu unterscheiden
  - $k.vater$ ist linkes Kind von $k.vater.vater$
    - (E1) Der „Onkel“ von k ist rot
    - (E2) Der „Onkel“ von k ist schwarz und k ist rechtes Kind
    - (E3) Der „Onkel“ von k ist schwarz und k ist linkes Kind
  - $k.vater$ ist rechtes Kind von $k.vater.vater$
    - (E4-E6) 3 analoge Fälle

Einfügen eines Elements

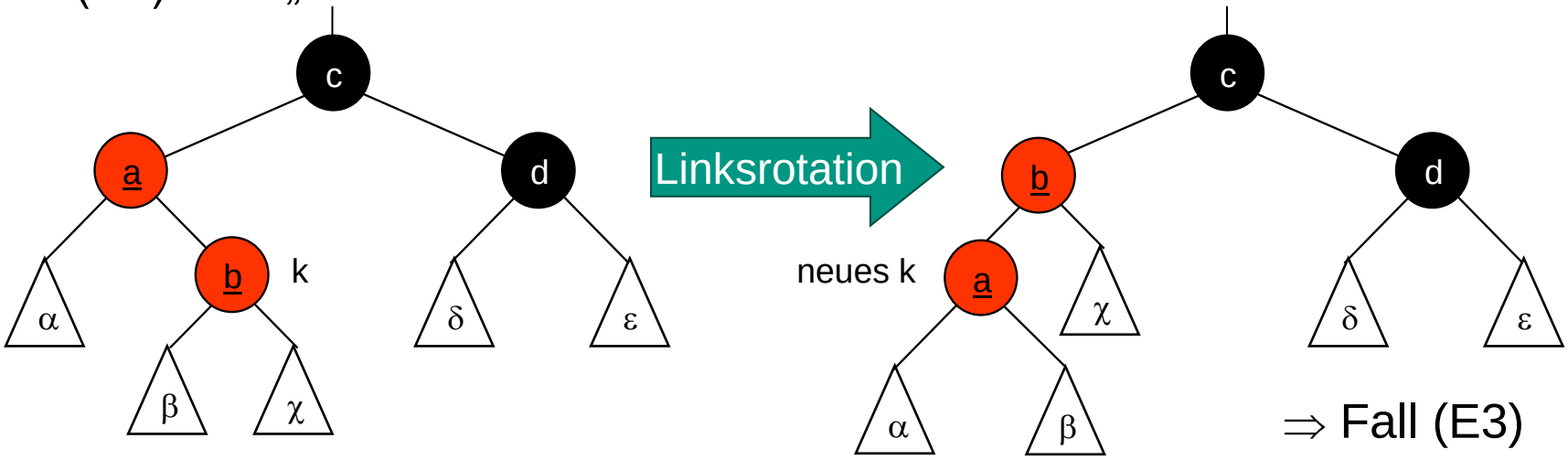
- Fall (E1): Der „Onkel“ von k ist rot



- $K.vater$ und Onkel von k schwarz färben
  - Eigenschaft (4) wieder erfüllt
- $k.vater.vater$ rot färben
  - Eigenschaft (5) wieder erfüllt
  - $k.vater.vater$ kann ebenfalls wieder Kind eines roten Knoten sein (erneute Verletzung von Eigenschaft (4))
  - Rekursive Wiederherstellung der Eigenschaft (4)
    - Aufsteigen um 2 Ebenen $k = k.vater.vater$
    - Falls Wurzel, schwarz färben, fertig

Einfügen eines Elements

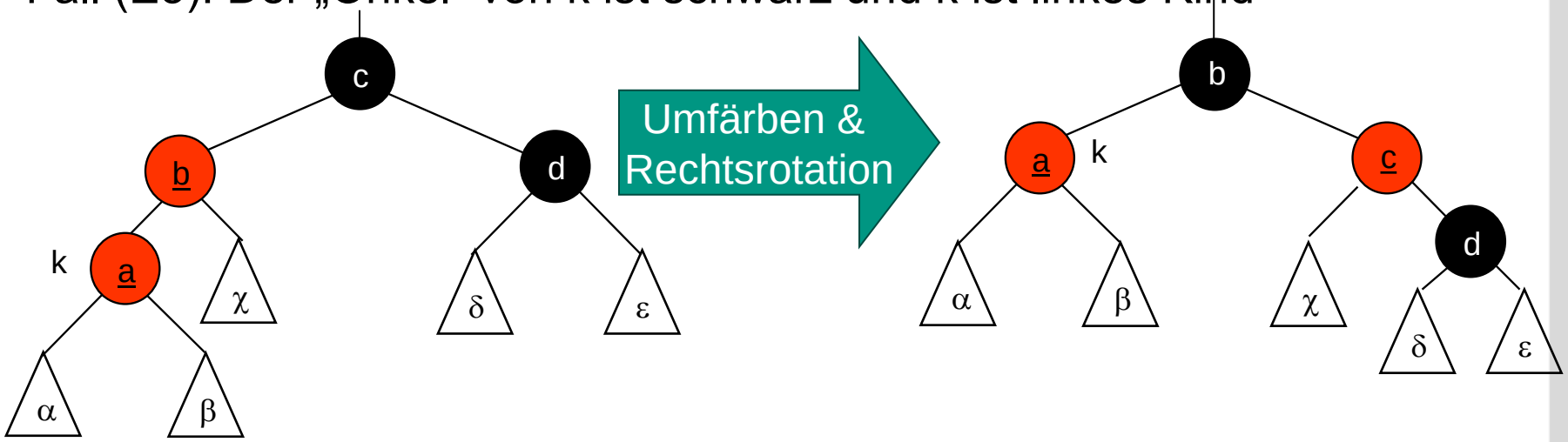
- Fall (E2): Der „Onkel“ von k ist schwarz und k ist rechtes Kind



- Durch Linksrotation der Knoten k und $k.vater$ entsteht Fall (E3) in Bezug auf den ehemaligen Vater-Knoten von k
  - Der „Onkel“ von k ist schwarz und k ist linkes Kind
- Keine Veränderungen hinsichtlich der Eigenschaften und der bh
- Weiter wie bei Fall (E3) beschrieben

Einfügen eines Elements

- Fall (E3): Der „Onkel“ von k ist schwarz und k ist linkes Kind



  - Vorgehen
    - Rechtsrotation und Rot-färbung des Knoten c
    - Schwarz-färbung von Knoten b
  - Knoten c hatte links und rechts gleiche Anzahl schwarzer Knoten
    - Da b vormals rot war, wird die Anzahl nach Übernahme des rechten Teilbaums von b als linken Teilbaum dies nicht ändern
  - Knoten b bekommt rechts schwarzen Knoten d hinzu
    - Da Anzahl der schwarzen Knoten der von b über a oder c laufenden Pfade aber gleich der von d (inklusive) ausgehenden Pfade war, bleibt Eigenschaft (5) gültig

Pseuocode – Einfügen

- $RB - INSERT(T, z)$

```
1   y = NIL
2   x = T.wurzel
3   while x ≠ NIL
4       y = x
5       if z.schlüssel < x.schlüssel
6           x = x.links
7       else
8           x = x.rechts
9   z.vater = y
10  if y == NIL
11      T.wurzel = z
12  elseif z.schlüssel < y.schlüssel
13      y.links = z
14  else
15      y.rechts = z
16  z.links = z.rechts = NIL
17  z.farbe = ROT
18  RB − INSERT − FIXUP(T, z)
```

Bis hier identisch mit $TREE - INSERT$

Färbung und NIL-Blätter

Rot/Schwarz Baum-Eigenschaften wiederherstellen

Pseuocode – Einfügen (2)

- $RB - INSERT - FIXUP(T, z)$

```
1   while z.vater.farbe == ROT
2       if z.vater == z.vater.vater.links
3           y = z.vater.vater.rechts
4           if y.farbe == ROT
5               z.vater.farbe = SCHWARZ
6               y.farbe = SCHWARZ
7               z.vater.vater.farbe = ROT
8               z = z.vater.vater
9           else if z == z.vater.rechts
10                  z = z.vater
11                  LEFT - ROTATE(T, z)
12              z.vater.farbe = SCHWARZ
13              z.vater.vater.farbe = ROT
14              RIGHT - ROTATE(T, z.vater.vater)
15      else (analog zum then-Fall)
16  T.wurzel.farbe = SCHWARZ
```

Fall (E1)

Fall (E2)

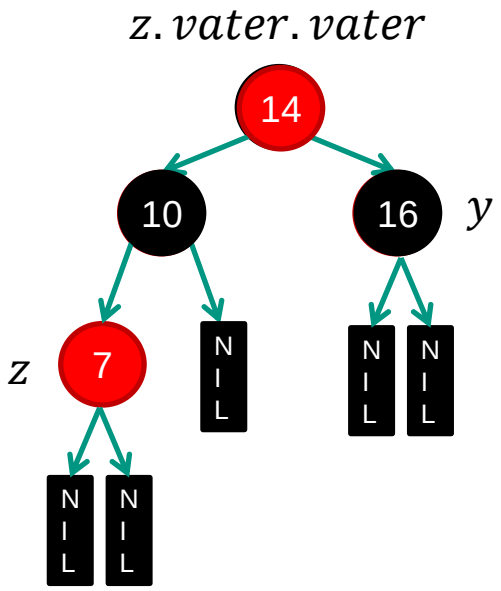
Fall (E3)

Pseuocode – Einfügen am Beispiel

- $RB - INSERT - FIXUP(T, z)$

```
1   while z.vater.farbe == ROT
2       if z.vater == z.vater.vater.links
3           y = z.vater.vater.rechts
4           if y.farbe == ROT
5               z.vater.farbe = SCHWARZ
6               y.farbe = SCHWARZ
7               z.vater.vater.farbe = ROT
8               z = z.vater.vater
9           else if z == z.vater.rechts
10              z = z.vater
11              LEFT − ROTATE(T, z)
12          z.vater.farbe = SCHWARZ
13          z.vater.vater.farbe = ROT
14          RIGHT − ROTATE(T, z.vater.vater)
15      else (analog zum then-Fall)
16  T.wurzel.farbe = SCHWARZ
```

In T wird die „7“ eingefügt:

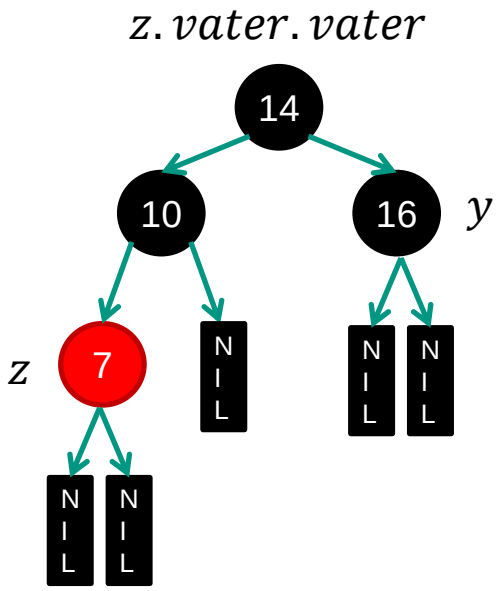


Pseuocode – Einfügen am Beispiel

- $RB - INSERT - FIXUP(T, z)$

```
1   while z.vater.farbe == ROT
2       if z.vater == z.vater.vater.links
3           y = z.vater.vater.rechts
4           if y.farbe == ROT
5               z.vater.farbe = SCHWARZ
6               y.farbe = SCHWARZ
7               z.vater.vater.farbe = ROT
8               z = z.vater.vater
9           else if z == z.vater.rechts
10              z = z.vater
11              LEFT − ROTATE(T, z)
12          z.vater.farbe = SCHWARZ
13          z.vater.vater.farbe = ROT
14          RIGHT − ROTATE(T, z.vater.vater)
15      else (analog zum then-Fall)
16  T.wurzel.farbe = SCHWARZ
```

In T wird die „7“ eingefügt:



Komplexitätsbetrachtung

- $RB - INSERT(T, z)$

```
1    y = NIL
2    x = T.wurzel
3    while x ≠ NIL
4            y = x
5            if z.schlüssel < x.schlüssel
6                    x = x.links
7            else
8                    x = x.rechts
9    z.vater = y
10   if y == NIL
11           T.wurzel = z
12   elseif z.schlüssel < y.schlüssel
13           y.links = z
14   else
15           y.rechts = z
16   z.links = z.rechts = NIL
17   z.farbe = ROT
18   RB - INSERT - FIXUP(T, z)
```

$O(\lg n)$

$O(\lg n)$

6.3.4 Löschen eines Elements

- Analog zum Einfügen!
  - Löschen eines Elements k mit üblichem Algorithmus für Löschen im binären Suchbaum
  - Element mit zwei Kindern durch kleinstes Element des rechten Teilbaums m ersetzen
    - Zuerst $k.schlüssel = m.schlüssel$
    - Dann m an Stelle von k entfernen
    - Knoten m hat maximal 1 Kind

6.3.4 Löschen eines Elements

- Analyse der Folgen für die Eigenschaften des Baums
  - Entfernter Knoten hat die Farbe Rot
    - Keine Verletzung der Eigenschaften
  - Entfernter Knoten k hat die Farbe Schwarz
    - Verletzung der Eigenschaft (5) (es sei denn Baum bestand nur aus k)
  - Entfernen der Wurzel des Baums und Nachrücken eines roten Knotens
    - Verletzung der Eigenschaft (2)
  - Vaterknoten von k hat die Farbe Rot und sein neuer Stellvertreter (Kind oder Vorgänger/Nachfolger aus dem linken/rechten Teilbaum) ebenfalls
    - Verletzung der Eigenschaft (4)

Löschen eines Elements

- Eigenschaften des Baums sind beginnend mit dem Kind x des entfernten Knotens k wieder herzustellen durch
  - Rotieren
  - Umfärben der Knoten

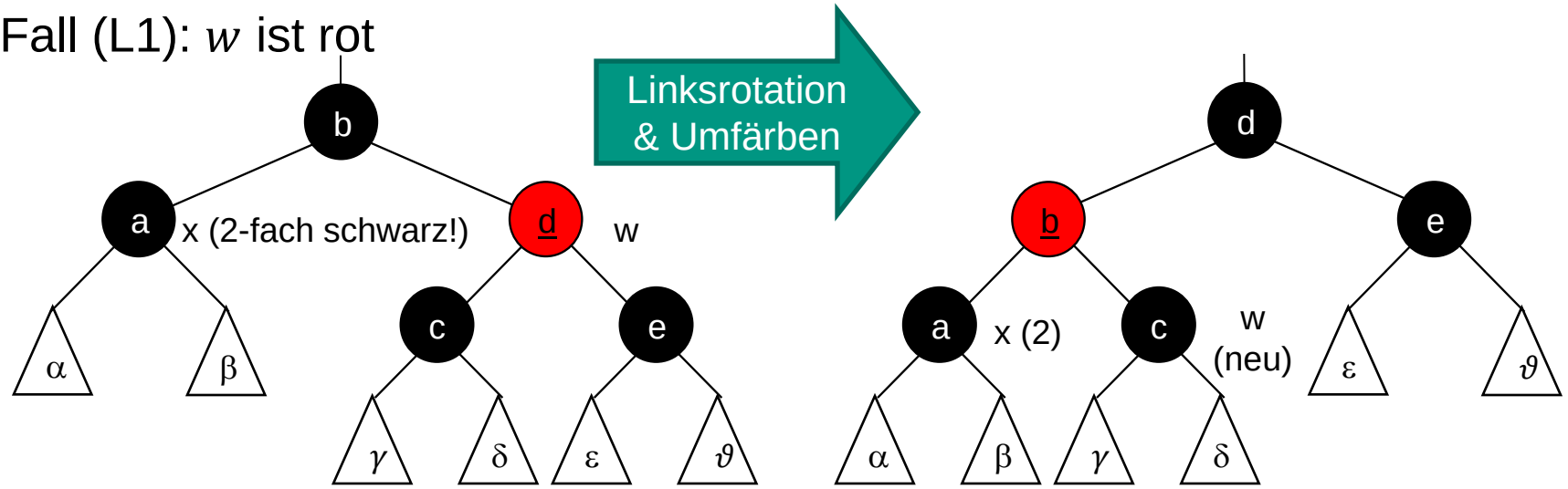
- Eigenschaften des Rot/Schwarz-Baums bis zur Wurzel wieder herstellen
  - Sollte die neue Wurzel die Farbe rot haben, diese Schwarz färben
    - Eigenschaft (2) ist wieder hergestellt
  - Entfernen des schwarzen Knotens führt zu Unterzahl schwarzer Knoten auf dem Pfad von der Wurzel über x
    - Beginnend von x wird ein erster auf dem Pfad hin zur Wurzel liegender roter Knoten schwarz gefärbt → Eigenschaft (4)
    - Ggf. werden Rotationen/Umfärbungen hierbei nötig

Löschen eines Elements

- Ausgangspunkt
  - Knoten x trage virtuell eine „doppelte“ Schwarz-Färbung, die nun in Richtung Wurzel auszugleichen ist
  - Verletzt Eigenschaft (1)
- Zwei Fälle
  - x ist linkes Kind von $x.vater$
  - x ist rechtes Kind von $x.vater$
  - Fälle in ihrer Behandlung symmetrisch, im Folgenden linkes Kind
- Sei w rechter Bruder von x
  - Da x die ehemalige Überzahl von schwarz gefärbten Knoten trägt, kann w nicht ein NIL-Knoten sein
- Vier Fälle
  - (L1) w ist rot
  - (L2) w und beide Kinder von w sind schwarz
  - (L3) w und dessen rechtes Kind sind schwarz, linkes Kind ist rot
  - (L4) w ist schwarz und rechtes Kind ist rot

Löschen eines Elements

- Fall (L1): w ist rot

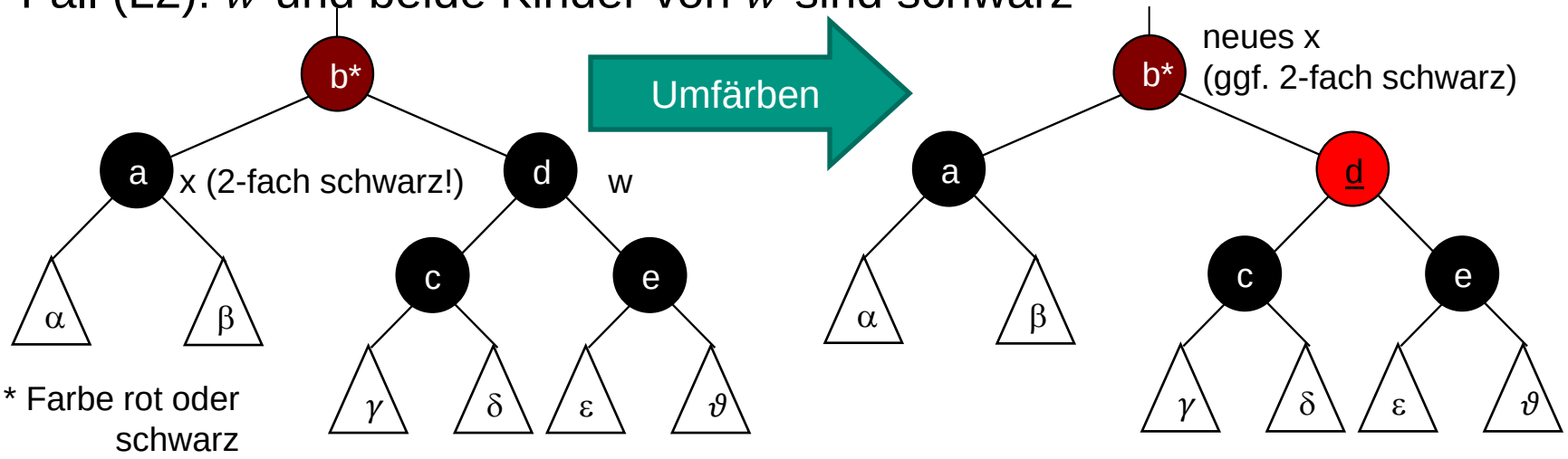


  - Vorgehen
    - Umfärben w und $x.vater$
    - Linksrotation um $x.vater$
    - Durch Linksrotation und Umfärben wird einer der Fälle (L2), (L3) oder (L4) generiert.
    - Neuer Bruder von x ist schwarz
  - Unverändert bleibt die Anzahl schwarzer Knoten auf Pfaden von der Wurzel zu den Teilbäumen α, β (Anzahl = 3 inkl. dem doppelt gezählten Knoten x), γ, δ, ε oder ϑ (Anzahl = 2)

→ Fall (L2),(L3) oder (L4)

Löschen eines Elements

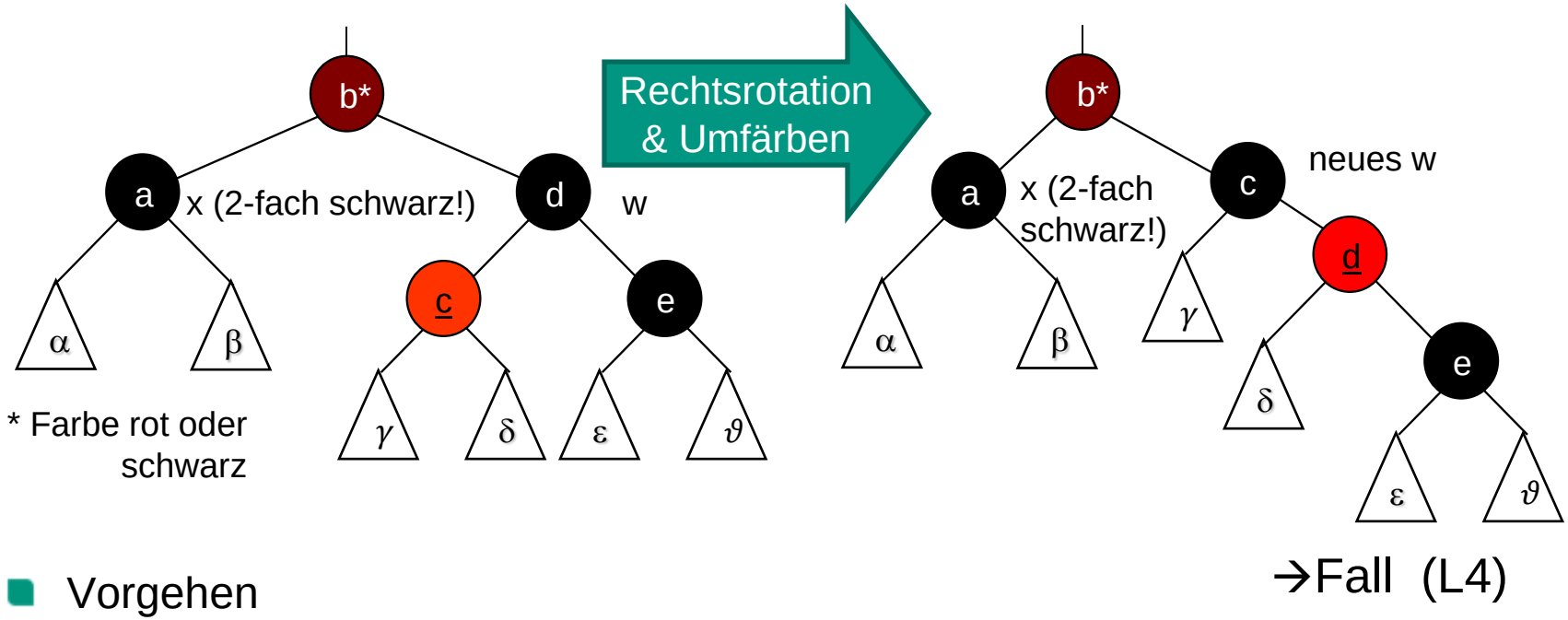
- Fall (L2): w und beide Kinder von w sind schwarz



  - Vorgehen
    - Umfärben von w und Reduzierung der Schwarz-Färbungen des Knotens x auf 1
    - $x.vater$ um eine virtuelle Schwarz-Färbung ergänzen
    - Wiederholen mit $x = x.vater$
  - Eigenschaft (5) für den Teilbaum wieder hergestellt
    - Keine Änderung bei den anderen Eigenschaften

Löschen eines Elements

- Fall (L3): w und dessen rechtes Kind sind schwarz, linkes Kind ist rot

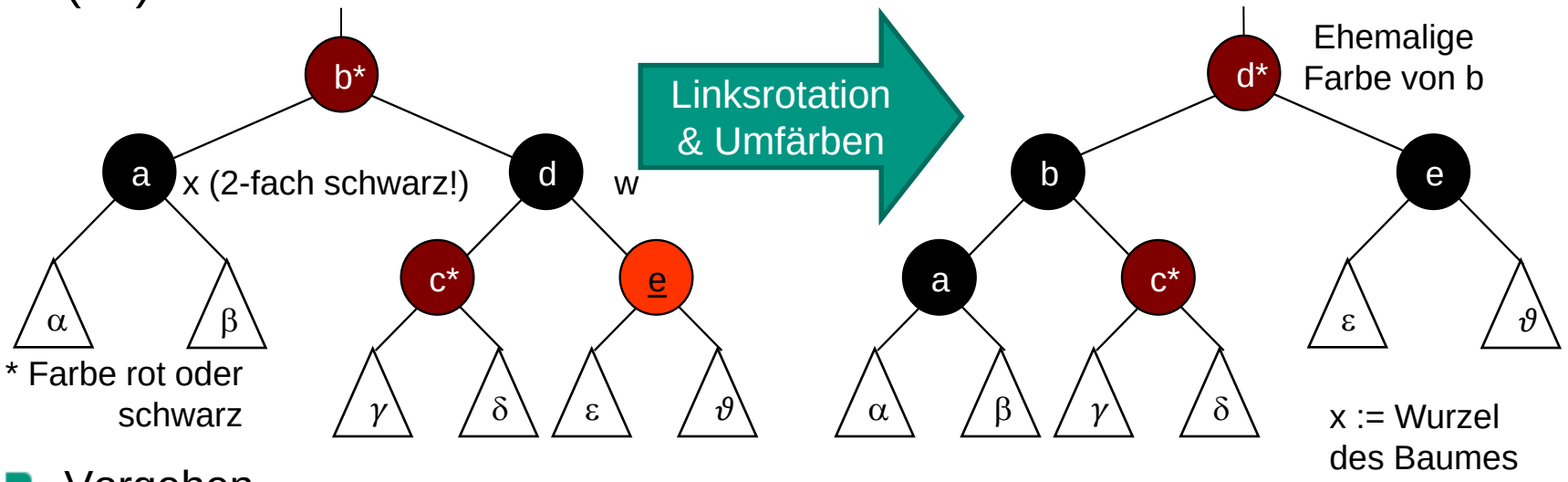


→Fall (L4)

- Vorgehen
  - Umfärben von w und dessen linkem Kind und nachfolgende Rechtsrotation
  - Transformation zu Fall (L4)
  - Eigenschaften bleiben unverändert

Löschen eines Elements

- Fall (L4): w ist schwarz und rechtes Kind ist rot



  - Vorgehen
    - Färbungen
      - Knoten w in Farbe von $x.vater$
      - $x.vater$ und $w.rechts$ in schwarz
    - Linksrotation um $x.vater$
    - Schwarz-Färbung des Knotens x auf 1 reduziert
  - Wurzel des Baums wird als neuer Knoten x betrachtet
    - Evtl. Schwarz-färben der Wurzel, Ende

Pseudocode – Hilfsfunktion Transplant

- Vorgehen analog zu $TREE - DELETE$
  - Dazu folgende Anpassung der Funktion $TRANSPLANT$

- $RB - TRANSPLANT(T, u, v)$

```
1  if u.vater == NIL
2      T.wurzel = v
3  elseif u == u.vater.links
4      u.vater.links = v
5  else
6      u.vater.rechts = v
7  v.vater = u.vater
```

Unterschied zu $TRANSPLANT$: Zeiger auf Vater aktualisieren

Pseudocode

$RB - DELETE(T, z)$

```
1   y = z
2   y.farbe‘ = y.farbe
3   if z.links == NIL
4       x = z.rechts
5       RB − TRANSPLANT(T, z, z.rechts)
6   elseif z.rechts == NIL
7       x = z.links
8       RB − TRANSPLANT(T, z, z.links)
9   else
10      ...

23 if y.farbe‘ == SCHWARZ
24      RB − DELETE − FIXUP(T, x)
```

Fall: z hat nur ein Kind rechts

Fall: z hat nur ein Kind links

Fall: sonst (→ nächste Folie)

Rot/Schwarz Baum-Eigenschaften wiederherstellen

Pseudocode

$RB - DELETE(T, z)$

```
...
10      y = TREE − MINIMUM(z.rechts)
11      y.farbe‘ = y.farbe
12      x = y.rechts
13      if y.vater == z
14          x.vater = y
15      else
16          RB − TRANSPLANT(T, y, y.rechts)
17          y.rechts = z.rechts
18          y.rechts.vater = y
19      RB − TRANSPLANT(T, z, y)
20      y.links = z.links
21      y.links.vater = y
22      y.farbe = z.farbe
...
```

Neuer Knoten an Stelle von z

Pseudocode

```
RB − DELETE − FIXUP(T, x)
1   while x ≠ T.wurzel und x.farbe == SCHWARZ
2       if x == x.vater.links
3           w = x.vater.rechts
4           if w.farbe == ROT
5               w.farbe = SCHWARZ; x.vater.farbe = ROT
6               LEFT − ROTATE(T, x.vater)
7               w = x.vater.rechts
8           if w.links.farbe == SCHWARZ und w.rechts.farbe == SCHWARZ
9               w.farbe = ROT
10              x = x.vater
11          else if w.rechts.farbe == SCHWARZ
12              w.links.farbe = SCHWARZ; w.farbe = ROT
13              RIGHT − ROTATE(T, w)
14              w = x.vater.rechts
15          w.farbe = x.vater.farbe
16          x.vater.farbe = w.rechts.farbe = SCHWARZ
17          LEFT − ROTATE(T, x.vater)
18          x = T.wurzel
19      else (analog zum then Fall)
20  x.farbe = SCHWARZ
```

Fall (L1)

Fall (L2)

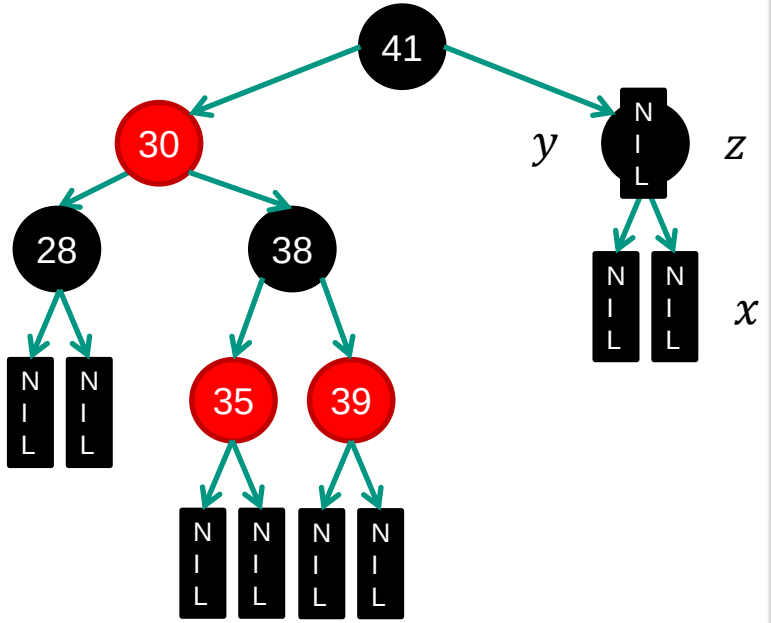
Fall (L3)

Fall (L4)

Pseudocode – Beispiel

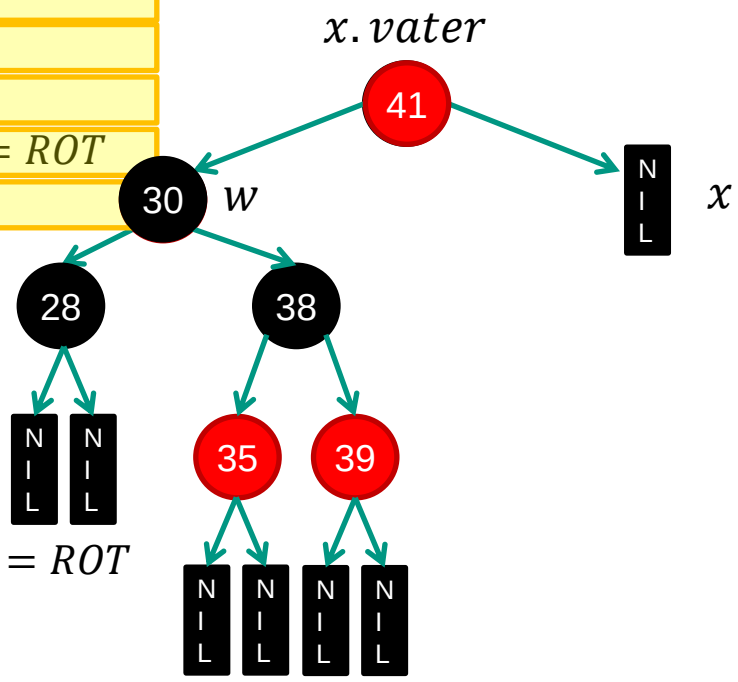
$RB - DELETE(T, z)$

```
1   y = z
2   y.farbe‘ = y.farbe
3   if z.links == NIL
4         x = z.rechts
5         RB − TRANSPLANT(T, z, z.rechts)
6   elseif z.rechts == NIL
7         x = z.links
8         RB − TRANSPLANT(T, z, z.links)
9   else
...
23  if y.farbe‘ == SCHWARZ
24        RB − DELETE − FIXUP(T, x)
```



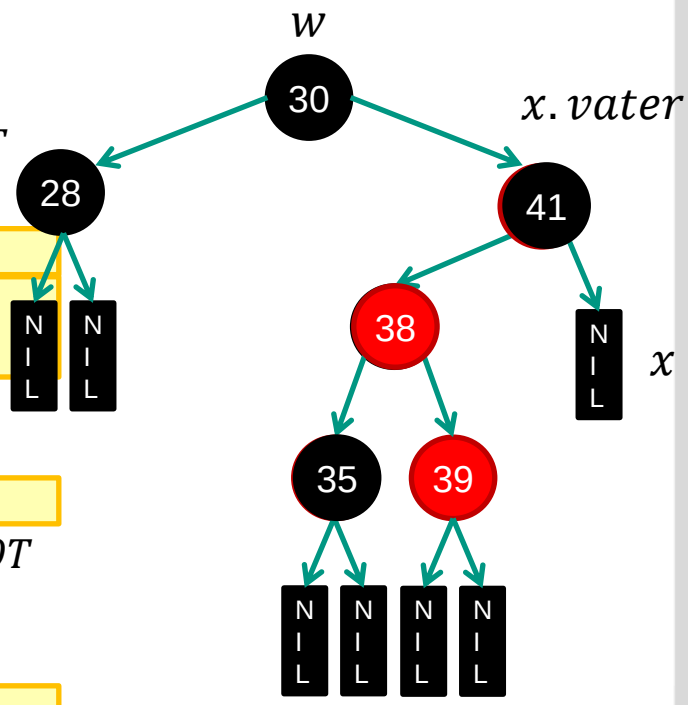
Pseudocode – Beispiel

```
RB − DELETE − FIXUP(T, x)
1   while x ≠ T.wurzel und x.farbe == SCHWARZ
2       if x == x.vater.rechts
3           w = x.vater.links
4           if w.farbe == ROT
5               w.farbe = SCHWARZ; x.vater.farbe = ROT
6               RIGHT − ROTATE(T, x.vater)
7               w = x.vater.links
8           if w.rechts.farbe == SCHWARZ
            und w.links.farbe == SCHWARZ
9               w.farbe = ROT
10              x = x.vater
11          else if w.links.farbe == SCHWARZ
12              w.rechts.farbe = SCHWARZ; w.farbe = ROT
13              LEFT − ROTATE(T, w)
14              w = x.vater.links
15          w.farbe = x.vater.farbe
16          x.vater.farbe = w.links.farbe = SCHWARZ
17          RIGHT − ROTATE(T, x.vater)
18          x = T.wurzel
20  x.farbe = SCHWARZ
```



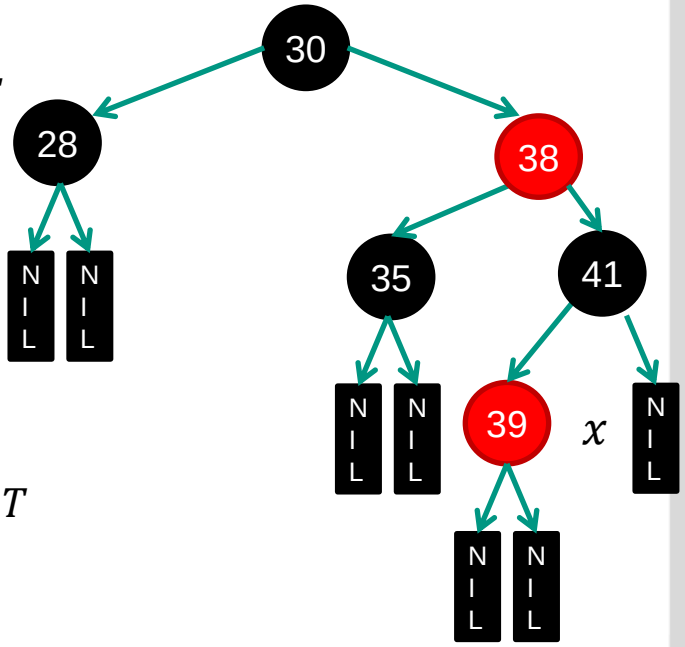
Pseudocode – Beispiel

```
RB − DELETE − FIXUP(T, x)
1   while x ≠ T.wurzel und x.farbe == SCHWARZ
2       if x == x.vater.rechts
3           w = x.vater.links
4           if w.farbe == ROT
5               w.farbe = SCHWARZ; x.vater.farbe = ROT
6               RIGHT − ROTATE(T, x.vater)
7               w = x.vater.links
8           if w.rechts.farbe == SCHWARZ
            und w.links.farbe == SCHWARZ
9               w.farbe = ROT
10              x = x.vater
11          else if w.links.farbe == SCHWARZ
12              w.rechts.farbe = SCHWARZ; w.farbe = ROT
13              LEFT − ROTATE(T, w)
14              w = x.vater.links
15          w.farbe = x.vater.farbe
16          x.vater.farbe = w.links.farbe = SCHWARZ
17          RIGHT − ROTATE(T, x.vater)
18          x = T.wurzel
20  x.farbe = SCHWARZ
```



Pseudocode – Beispiel

```
RB − DELETE − FIXUP(T, x)
1   while x ≠ T.wurzel und x.farbe == SCHWARZ
2       if x == x.vater.rechts
3           w = x.vater.links
4           if w.farbe == ROT
5               w.farbe = SCHWARZ; x.vater.farbe = ROT
6               RIGHT − ROTATE(T, x.vater)
7               w = x.vater.links
8           if w.rechts.farbe == SCHWARZ
            und w.links.farbe == SCHWARZ
9               w.farbe = ROT
10              x = x.vater
11          else if w.links.farbe == SCHWARZ
12              w.rechts.farbe = SCHWARZ; w.farbe = ROT
13              LEFT − ROTATE(T, w)
14              w = x.vater.links
15          w.farbe = x.vater.farbe
16          x.vater.farbe = w.links.farbe = SCHWARZ
17          RIGHT − ROTATE(T, x.vater)
18          x = T.wurzel
20  x.farbe = SCHWARZ
```



Komplexitätsbetrachtung

- $RB - DELETE$ ohne $RB - DELETE - FIXUP$ liegt in $O(\lg n)$
  - Grund: Analog zu Suchbaum-Funktion $TREE - DELETE$

- $RB - DELETE - FIXUP$
  - Jeder der Fälle L1, L3, L4 lässt Funktion nach konstanter Zeit in $O(1)$ terminieren, maximal drei Rotationen
  - Fall L2 besitzt Schleife: Maximal $O(\lg n)$ Schritte durch den Baum
  - Gesamtlaufzeit damit ebenfalls in $O(\lg n)$

Zwischenfazit

- Suchbäume einfache Datenstruktur 
  - Effizienz abhängig von Balance
  - Worst-Case Einfügen, Suchen, Löschen linear
- Rot-Schwarz-Bäume erweiterten das Konzept 
  - Balance wird in bestimmten Grenzen eingehalten
  - Worst-Case logarithmisch
- Suchbäume eigenen sich vor allem für schnellen Arbeitsspeicher
  - Optimierungen für Hintergrundspeicher (Festplatte etc.) notwendig
  - B-Bäume Abschnitt 6.4

6.4 B-Bäume

- Bisher (implizit) angenommen
  - Bäume passen vollständig in den Hauptspeicher
- Aber … teilweise sehr große Datenmengen
  - Passen nicht mehr in den Hauptspeicher
  - Müssen auf Hintergrundspeicher ausgelagert werden (z.B. Festplatte)
    - Im Folgenden immer Annahme, dass direkter Zugriff auf Hintergrundspeicher möglich (also keine Magnetbänder – sequentieller Zugriff)
  - Zugriff auf den Hintergrundspeicher teuer!
    - … typischerweise um mehrere Größenordnungen höher als Kosten für Zugriff auf den Hauptspeicher

B-Bäume

- Grundlegende Idee
  - Anzahl von Zugriffen auf Hintergrundspeicher gering halten
    - Vielwegbaum nutzen (viele Verzweigungen pro Knoten auch tausende) aber geringe Tiefe des Baums
    - Anzahl der Zugriffe abhängig von Höhe des Baums
  - Knoten im Baum werden als „Index“ gesehen
    - Index ist also hierarchisch organisiert
  - … Index muss eventuell vom Hintergrundspeicher geladen werden

- Beobachtung
  - Hintergrundspeicher unterstützt meist das blockweise Lesen/Schreiben „größerer" Datenmengen bei einem Zugriff

- Also …
  - „Zusammengehörende“ Daten möglichst in einem Block speichern

6.4.1 Eigenschaften von B-Bäumen

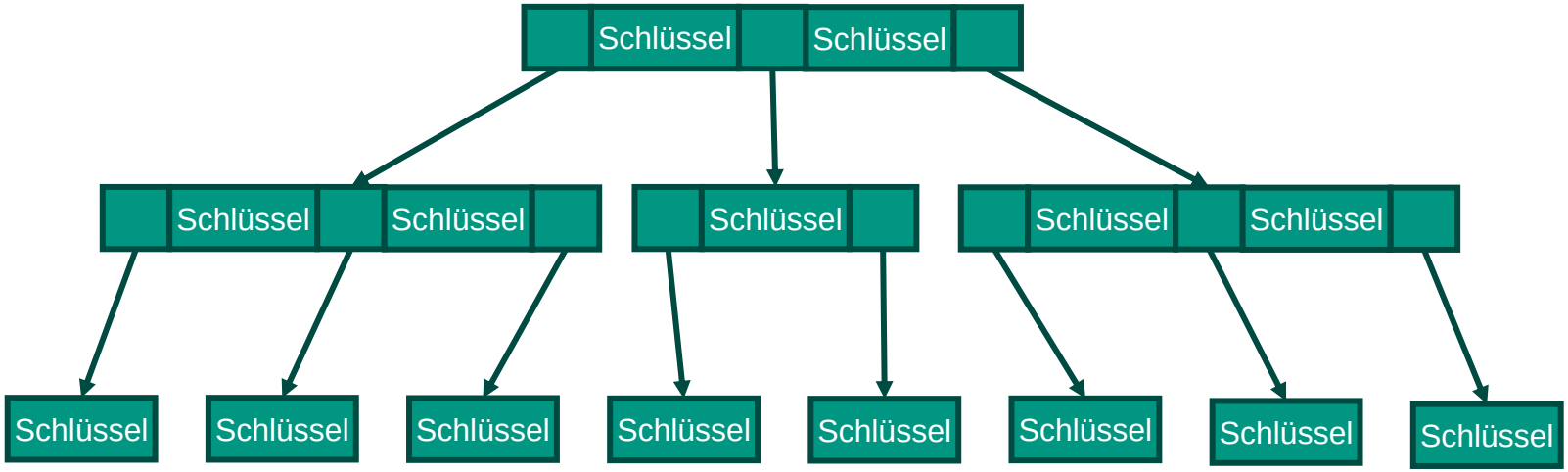
- Knoten in B-Bäumen
  - Strukturierung innerhalb eines Knotens
    - Schlüsselwert
    - Nutzdaten
    - Zeiger

- Sei p innerer Knoten eines B-Baums
  - p hat l Schlüssel s_x
  - p hat $(l + 1)$ Kinder
  - Zeiger p_y zeigen jeweils auf Kinder

- Aufbau in einem inneren Knoten

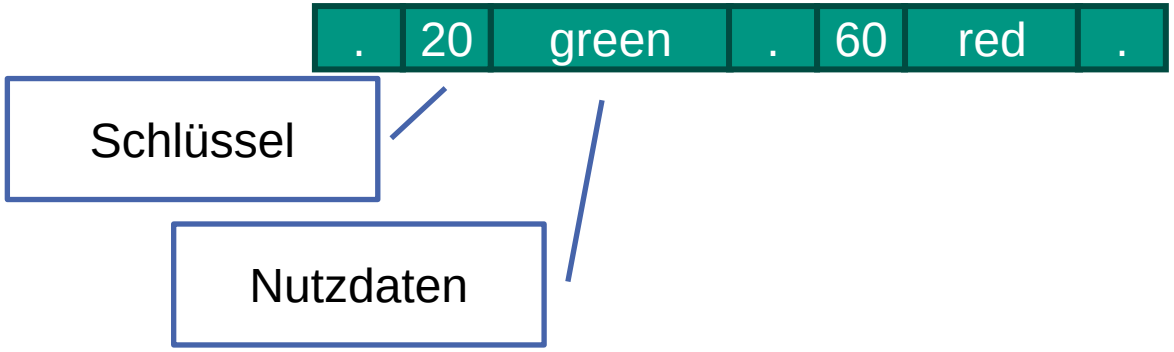
Strukturelle Eigenschaften von B-Bäumen

- B-Baum der Ordnung m
  - (1) Alle Blätter haben die gleiche Tiefe
  - (2) Jeder Knoten hat mindestens $\left\lceil \frac{m}{2} \right\rceil$ Kinder
  - (3) Die Wurzel hat mindestens 2 Kinder
  - (4) Jeder Knoten hat höchstens m Kinder
  - (5) Jeder Knoten mit i Kindern hat $i - 1$ Schlüssel
- Beispiel
  - B-Baum der Ordnung $m = 3$



Knoten in B-Bäumen – Beispiel

- Zu jedem Schlüssel gehören üblicherweise auch (größere) Nutzdaten auf die über den Schlüssel zugegriffen wird
- Beispiel



- Seien $s_1,\ldots,s_l$ die Schlüssel eines Knotens p mit $l+1$ Kindern. Seien $p_0, p_1, \ldots, p_l$ die Zeiger auf diese Kinder
  - p_0 weist auf Teilbaum mit Schlüsseln kleiner s_1
  - p_i $(i = 1, \ldots, l-1)$ weist auf Teilbaum, dessen Schlüssel echt zwischen s_i und s_{i+1} liegen
  - p_l weist auf Teilbaum mit Schlüsseln größer s_l
  - In den Blattknoten sind die Zeiger nicht definiert

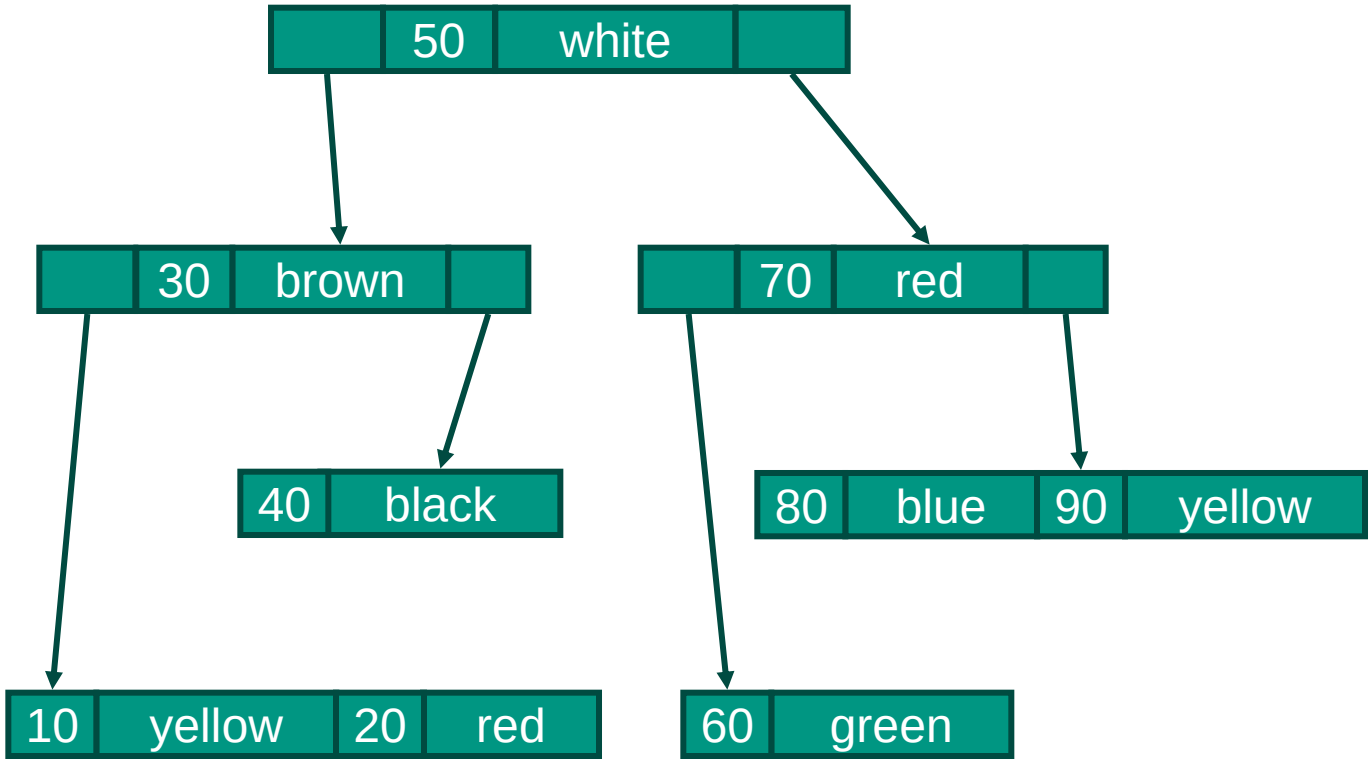
B-Baum – Beispiel

- Betrachte Objektmenge

Annahme: $m = 3$

Id	Farbe
10	yellow
20	red
30	brown
40	black
50	white
60	green
70	red
80	blue
90	yellow

Schlüssel



6.4.2 Höhe eines B-Baumes

- Ein B-Baum T wächst logarithmisch zur Anzahl Schlüssel n
  - Für die Höhe gilt konkret $h \leq \log_{\lceil \frac{m}{2} \rceil} \frac{n+1}{2}, n \geq 1$
  - Basis des Logarithmus abhängig von m, nicht wie bei Binärbäumen $\log_2$

- Beweis
  - Wurzel enthält mindestens 1 Schlüssel, andere Knoten mindestens $\lceil \frac{m}{2} \rceil - 1$ Schlüssel
  - T enthält
    - In der Ebene 0 mindestens 1 Knoten
    - In der Ebene 1 mindestens 2 Knoten
    - In der Ebene 2 mindestens $2(\lceil \frac{m}{2} \rceil)$ Knoten
    - …
    - In der Ebene h mindestens $2\left(\lceil \frac{m}{2} \rceil\right)^{h-1}$ Knoten

Höhe eines B-Baumes

- Beweis (Fortsetzung)
  - T enthält in der Ebene h mindestens $2\left(\left\lceil\frac{m}{2}\right\rceil\right)^{h-1}$ Knoten
  - Für Zahl der Schlüssel n gilt daher

$$n \geq 1 + \left(\left\lceil\frac{m}{2}\right\rceil - 1\right)\sum_{i=1}^{h} 2\left(\left\lceil\frac{m}{2}\right\rceil\right)^{i-1} = 1 + 2\left(\left\lceil\frac{m}{2}\right\rceil - 1\right)\left(\frac{\left(\left\lceil\frac{m}{2}\right\rceil\right)^{h} - 1}{\left(\left\lceil\frac{m}{2}\right\rceil\right) - 1}\right)$$

$$= 1 + 2\left(\left(\left\lceil\frac{m}{2}\right\rceil\right)^{h} - 1\right) = 1 + 2\left(\left\lceil\frac{m}{2}\right\rceil\right)^{h} - 2 = 2\left(\left\lceil\frac{m}{2}\right\rceil\right)^{h} - 1$$

  - Umformen ergibt $\left(\left\lceil\frac{m}{2}\right\rceil\right)^{h} \leq \frac{n+1}{2}$
  - Logarithmus $\log_{\left\lceil\frac{m}{2}\right\rceil}$ auf beiden Seiten → $h \leq \log_{\left\lceil\frac{m}{2}\right\rceil}\frac{n+1}{2}$

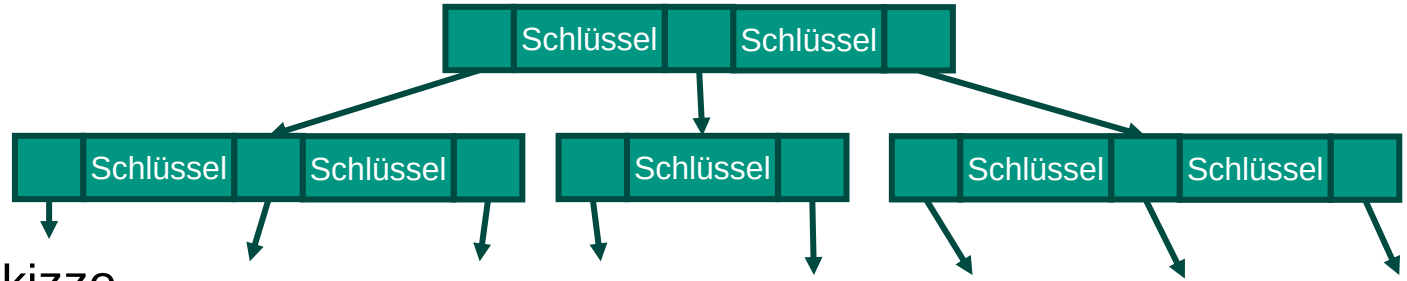
q.e.d.

Vergleich zu Rot/Schwarz-Bäumen

- B-Bäume wachsen auch „nur“ logarithmisch
- Die flexible Wahl der Basis des Logarithmus $\left\lceil \frac{m}{2} \right\rceil$ ermöglicht aber Verbesserungen die sich in der Praxis auswirken
- Beispiel $m = 199$
  - B-Bäume mit bis zu 1999999 Schlüsseln haben höchstens Höhe 4
- → Anzahl der Zugriffe auf Hintergrundspeicher proportional zur Höhe!

6.4.3 Suchen eines Objekts

- Vergleichbar mit Suchen in binärem Suchbaum
  - Statt binärer Entscheidung Mehrwege-Entscheidung



- Skizze
  - Start bei Wurzel. Schlüssel im Knoten? Wenn nein, dann geeignete Verzweigungsstelle bestimmen
  - Schlüssel gefunden oder erfolgloses Ende am Blatt
- Aufwand
  - Anzahl Zugriffe auf Hintergrundspeicher maximal: $O(\log_{\lceil \frac{m}{2} \rceil} n)$
  - Innerhalb eines Knotens: $O(\lceil \frac{m}{2} \rceil)$
  - Insgesamt also: $O(\lceil \frac{m}{2} \rceil \log_{\lceil \frac{m}{2} \rceil} n)$

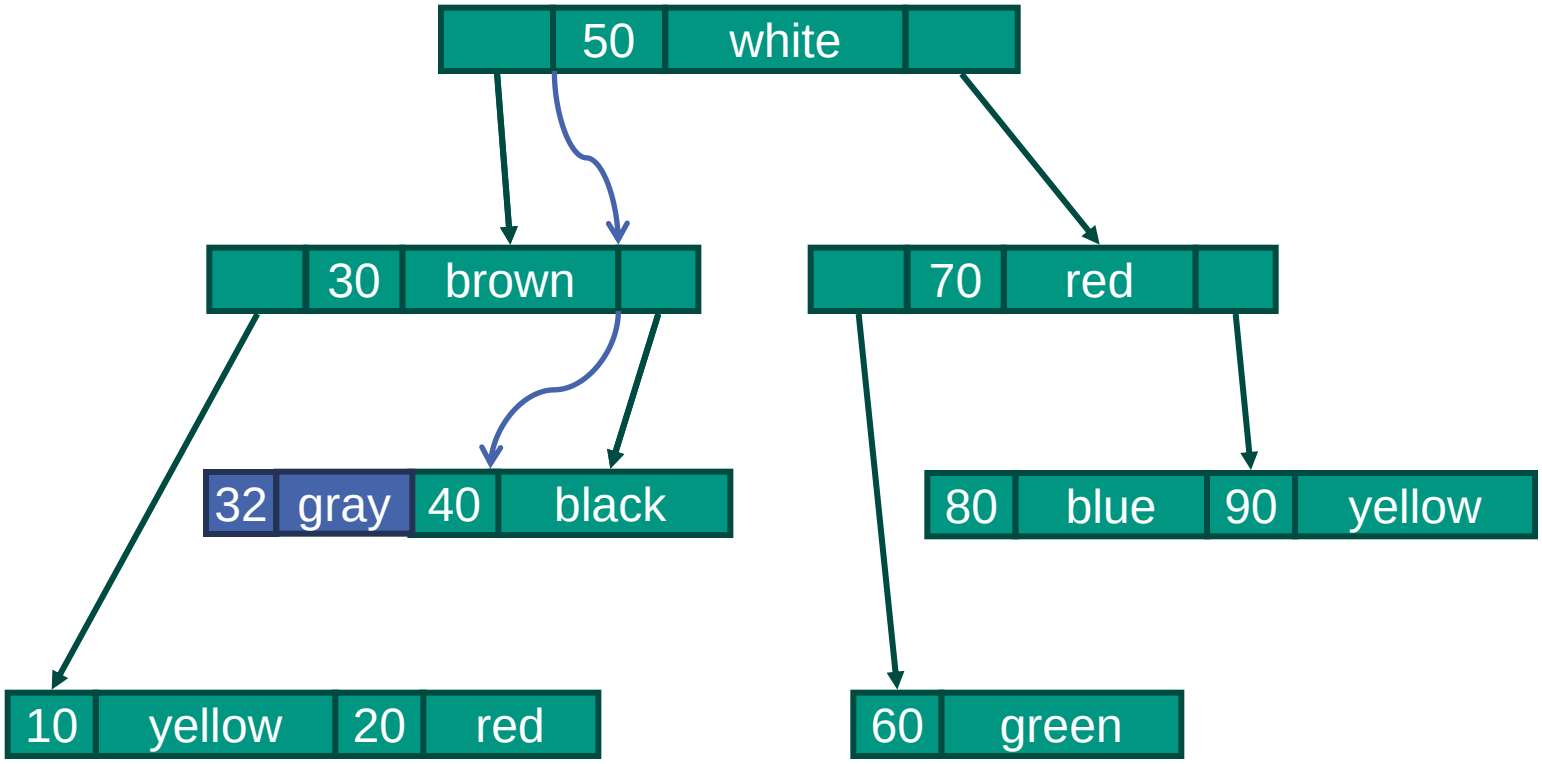
6.4.4 Einfügen eines Objekts

- Zu beachten
  - Es gilt immer: jeder Knoten enthält n Schlüssel mit
    - $\left\lceil \frac{m}{2} \right\rceil - 1 \leq n \leq m - 1$

- Vorgehen
  - Suche nach dem Schlüssel des neuen Objekts
    - Da noch nicht enthalten, Suche bis zum Blatt
  - Fall 1
    - Blattknoten hat noch Platz für das Objekt
    - Einfügen des neuen Objekts an die entsprechende Stelle im Knoten
  - Fall 2
    - Überlauf im Blattknoten, da bereits $m - 1$ Schlüssel gespeichert

Beispiel Einfügen

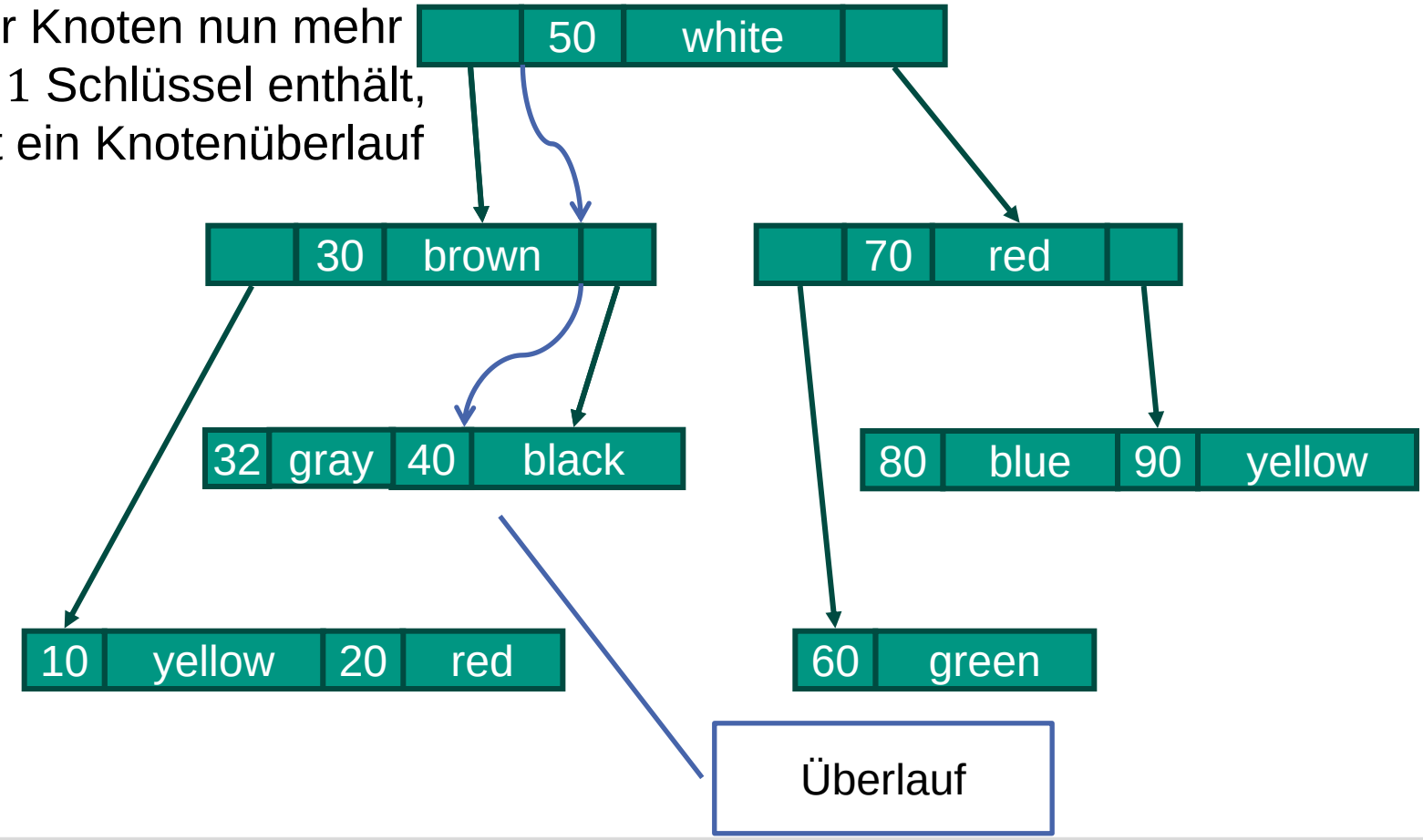
- Einfügen von [32, gray]



Beispiel Einfügen

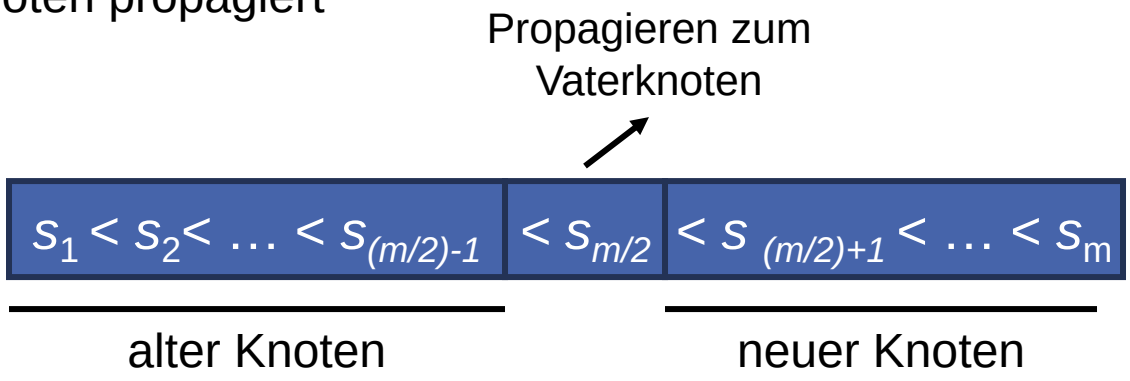
- Einfügen von [42, rose]

- Falls der Knoten nun mehr als $m - 1$ Schlüssel enthält, entsteht ein Knotenüberlauf



Beispiel Einfügen

- Bei Überlauf eines Knotens → Split
  - Einträge des Knotens werden auf zwei Knoten verteilt
  - Das Objekt, das in der Mitte des übergelaufenen Knotens liegt, wird zum Vaterknoten propagiert

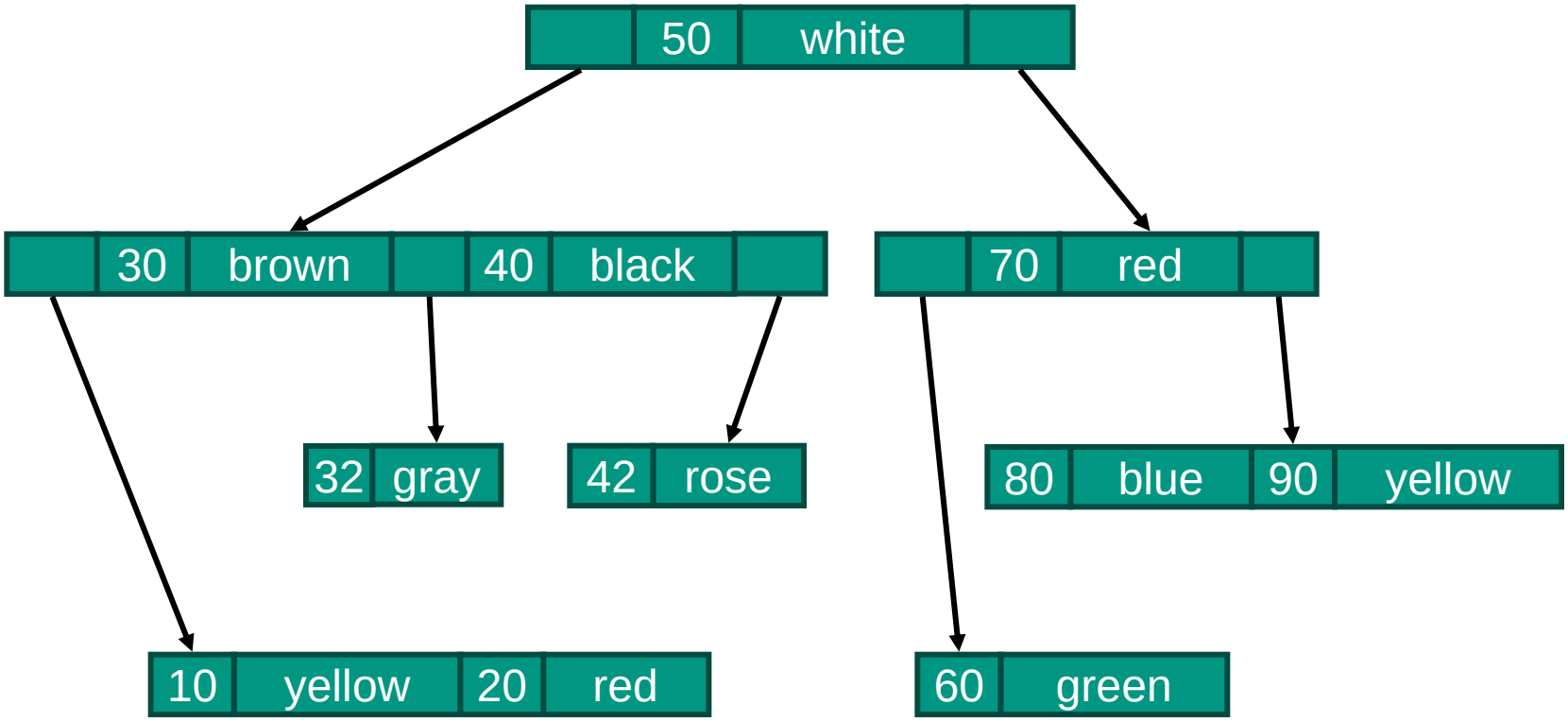


Hier:

  - Schema
    - Die ersten $m/2$ Werte verbleiben im alten Knoten
    - Die letzten $m/2$ Werte werden verschoben in neuen Knoten
    - Das mittlere Element wandert in den Vaterknoten

Beispiel Einfügen

- Einfügen von [42, rose]



Einfügen eines Objekts

- Spalten von Knoten kann sich im Vaterknoten rekursiv fortsetzen

- Schlimmster Fall
  - Alle Knoten bis zur Wurzel laufen über und werden gespalten
  - Wenn sogar die Wurzel überläuft, wird auch sie gespalten
  - Folge: Der Baum wächst in der Höhe um 1

- Aufwand
  - Insgesamt also: $O\left(\left\lceil\frac{m}{2}\right\rceil \log_{\left\lceil\frac{m}{2}\right\rceil} n\right)$

6.4.5 Entfernen eines Objekts

- Suche nach dem zu entfernenden Objekt anhand seines Schlüssels

- Zwei Fälle
  - Fall 1: Objekt befindet sich in einem Blatt
    - Blatt kann einfach entfernt werden
    - Aber: Knotenunterlauf möglich
  - Fall 2: Objekt befindet sich in einem inneren Knoten oder in der Wurzel
    - Vorgehen komplexer

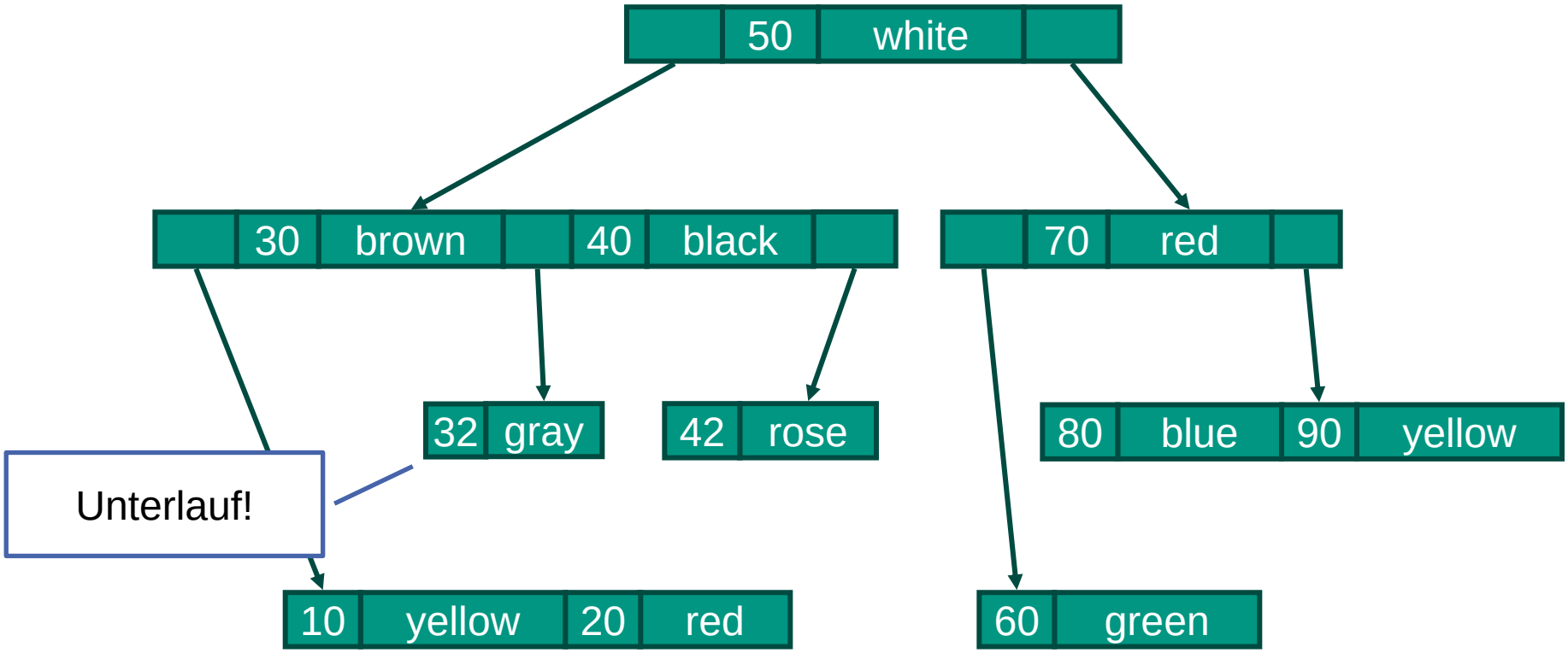
Fall 1 – Blattknoten entfernen

- Knotenunterlauf möglich
  - Versuche Objekte aus den Nachbarknoten in den Knoten mit Unterlauf zu verschieben
  - Indirektion durch Bewegen über Vaterknoten!

- Problem: Alle Nachbarknoten minimal belegt
  - Es kann kein Ausgleich stattfinden
    - Entfernen von Objekten aus den Nachbarknoten führt dort zu Unterlauf
  - → Unterlaufknoten mit einem seiner Nachbarknoten verschmelzen
    - Auch machbar, wenn 1 Nachbarknoten minimal belegt
  - Das Zusammenlegen zweier Knoten führt zu dem Verschieben des trennenden (Separator-) Objektes aus dem Vaterknoten in den neuen, durch das Verschmelzen entstehenden Knoten
  - Verschmelzen von Knoten kann sich bis zur Wurzel fortsetzen
  - Der Baum schrumpft in der Höhe um 1

Entfernen – Beispiel 1

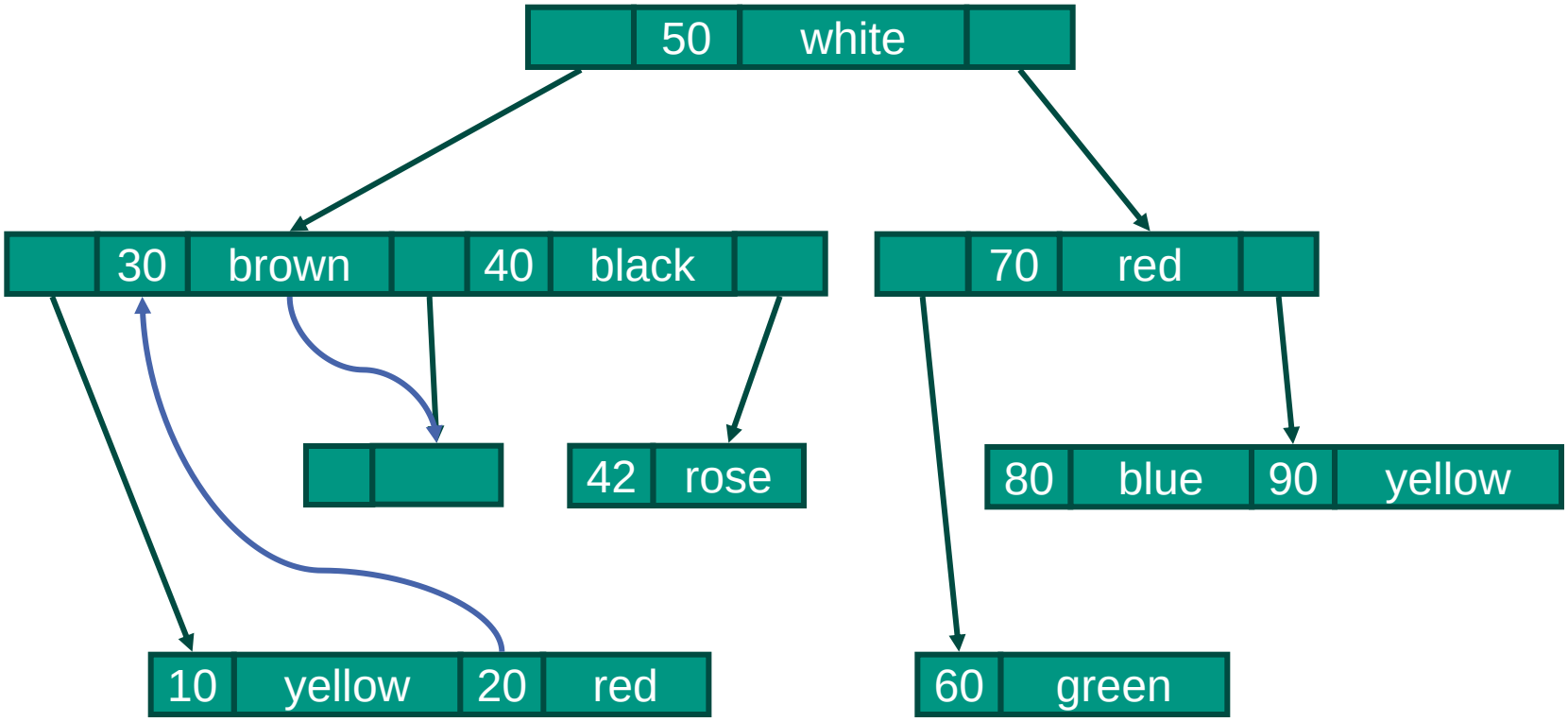
- Entfernen des Objekts [32, gray] (Blattknoten)



Falls der Blattknoten nunmehr weniger als $\left\lceil \frac{m}{2} \right\rceil - 1$ Objekte enthält, entsteht ein Knotenunterlauf!

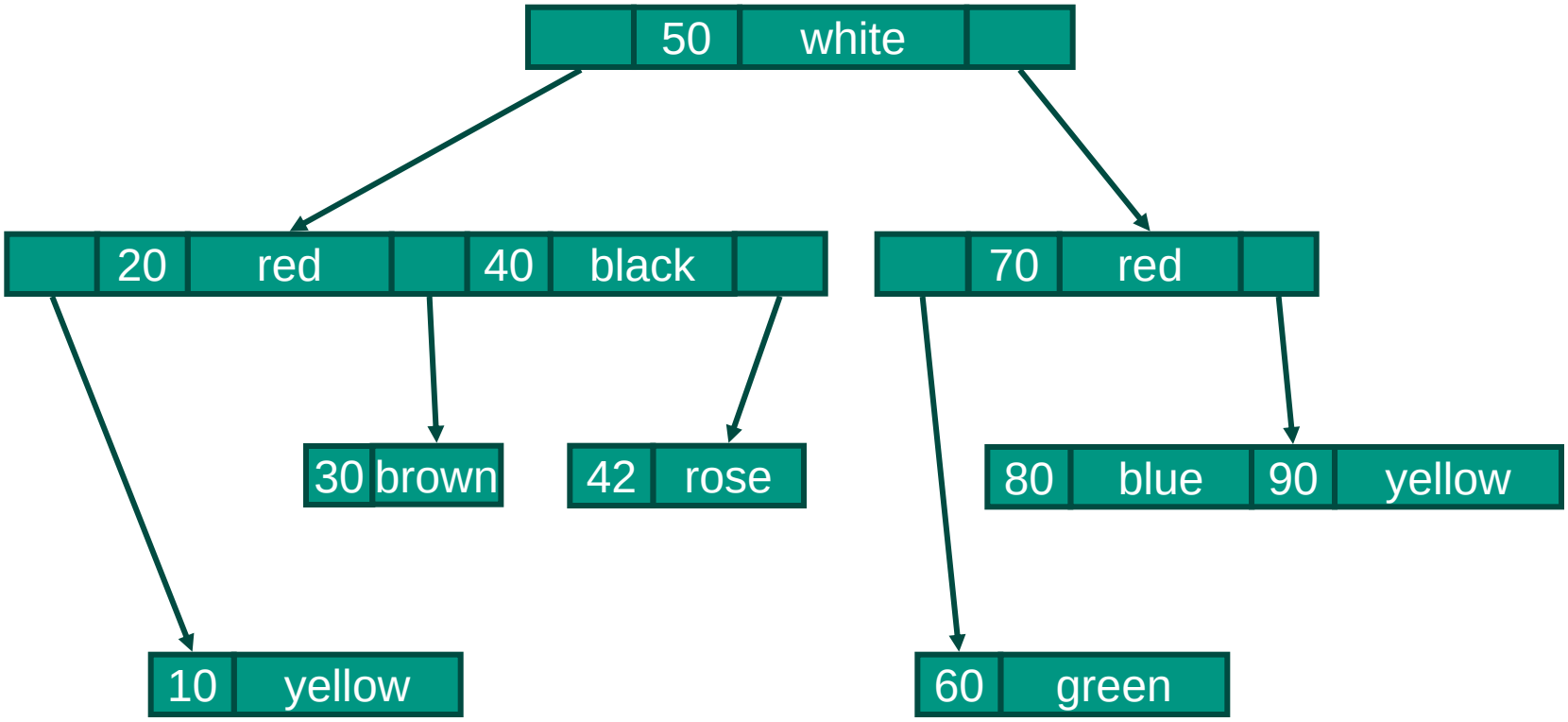
Entfernen – Beispiel 1

- Entfernen des Objekts [32, gray] (Blattknoten)



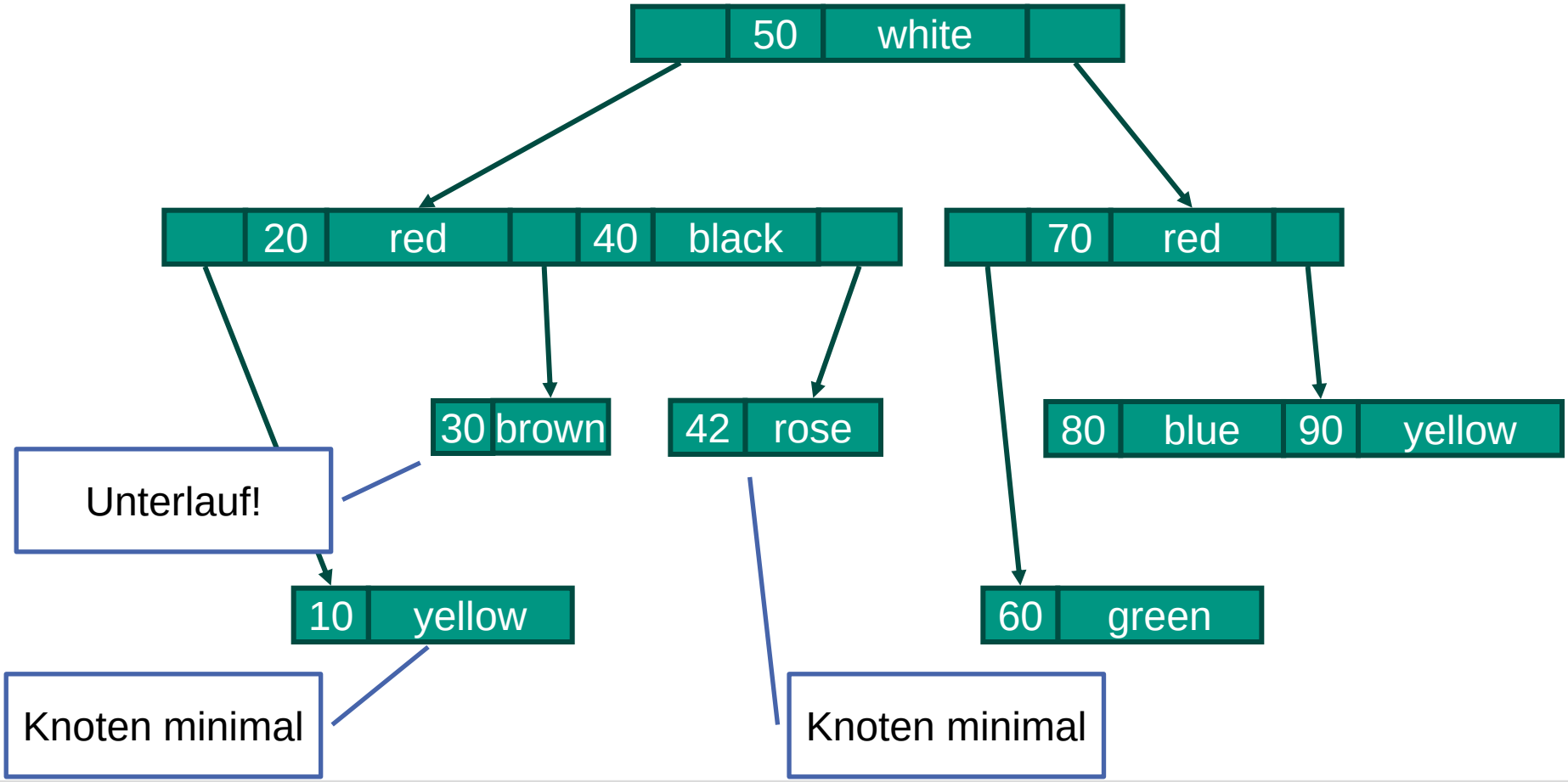
Entfernen – Beispiel 1

- Entfernen des Objekts [32, gray] (Blattknoten)



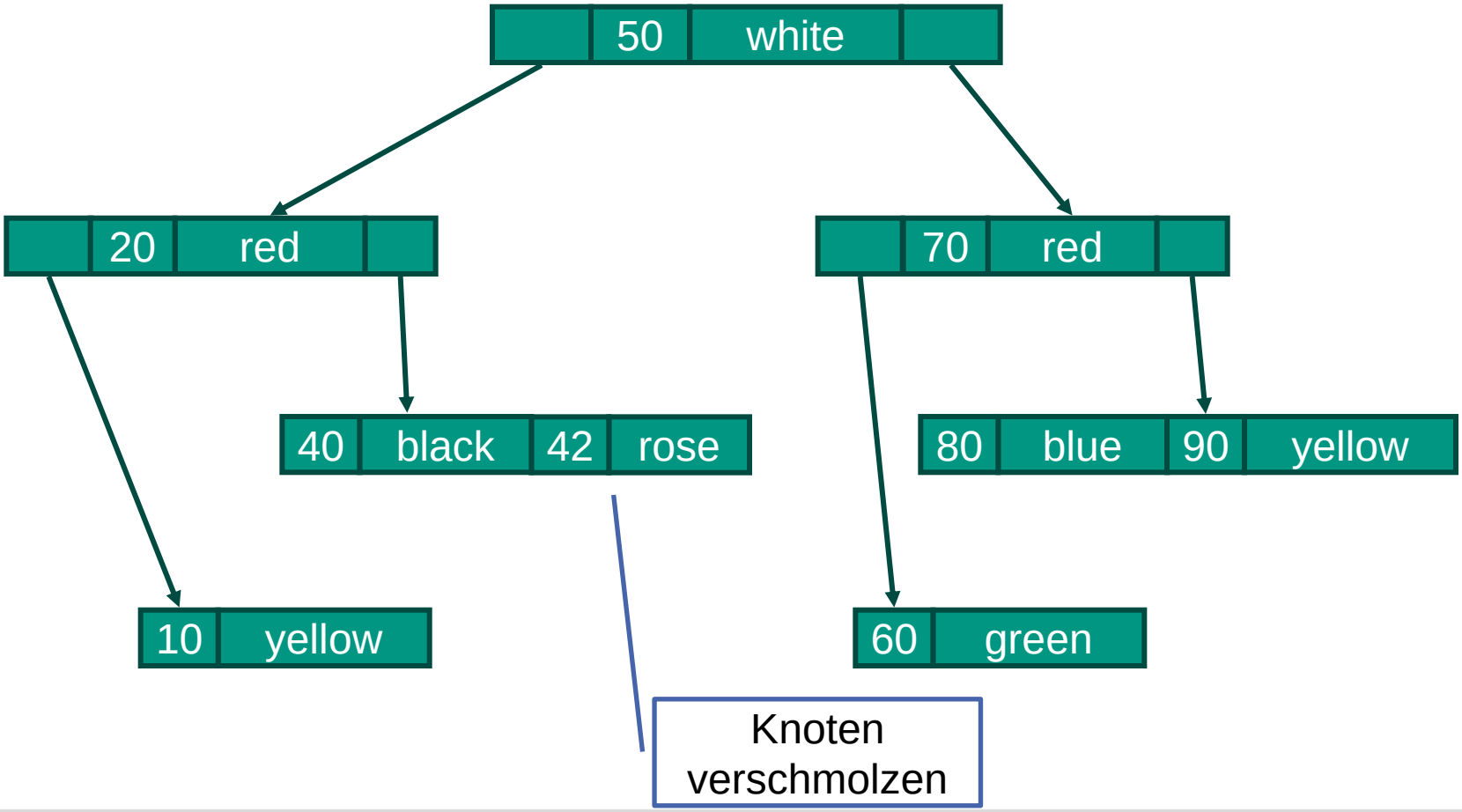
Entfernen – Beispiel 2

- Entfernen des Objekts [30, brown] (Blattknoten)



Entfernen – Beispiel 2

- Entfernen des Objekts [30, brown] (Blattknoten)



Fall 2 – Inneren Knoten entfernen

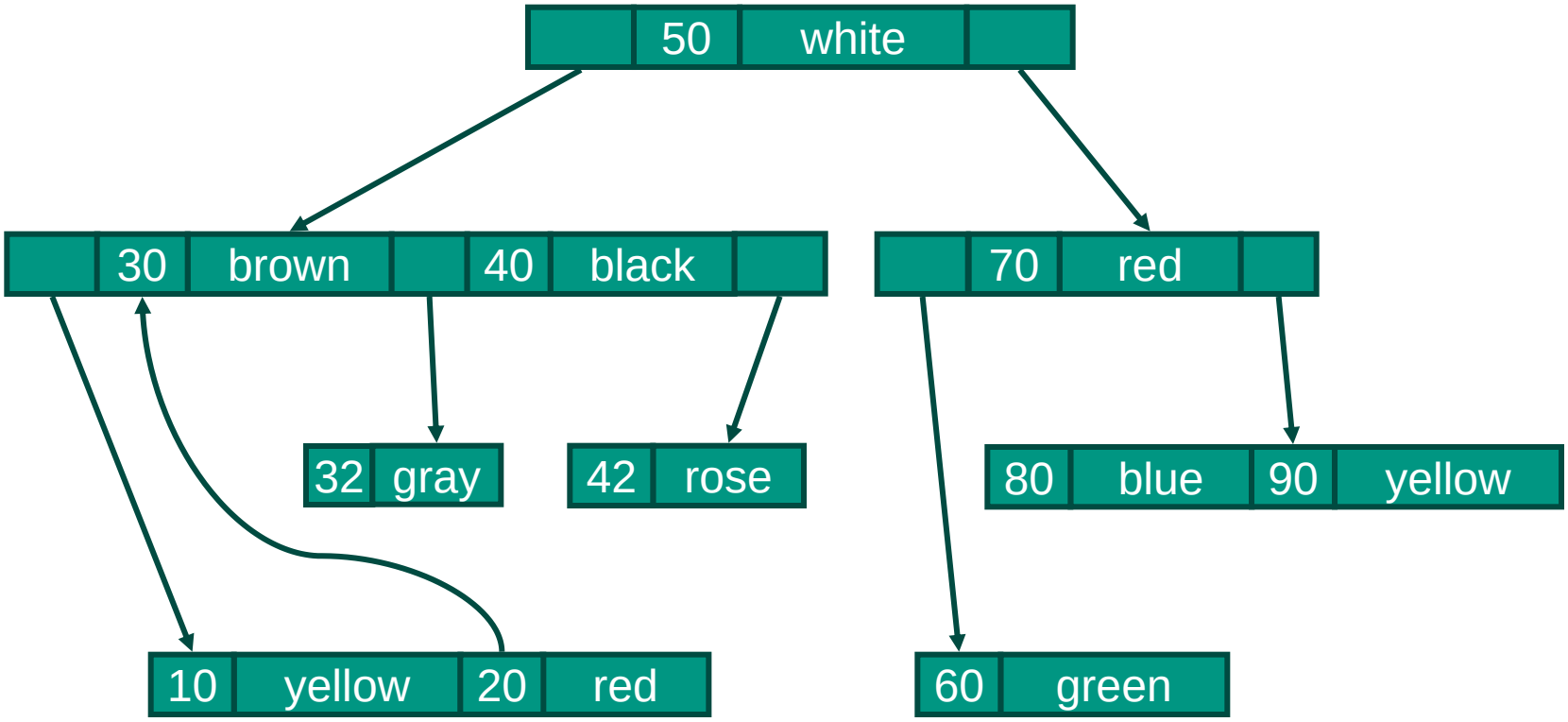
- Finde ein neues Separator-Objekt als Ersatz für das zu entfernende Objekt

- Zwei Alternativen
  - Wähle das Objekt mit dem kleinsten Schlüssel, der größer ist als der Schlüssel des zu entfernenden Objekts
  - Wähle das Objekt mit dem größten Schlüssel, der kleiner ist als der Schlüssel des zu entfernenden Objekts

- Vorgehen
  - Separator-Objekt aus seinem Knoten entfernen
    - Rekursiver Aufruf der Operation zum Entfernen von Objekten
  - Separator-Objekt ersetzt das zu entfernende Objekt in dessen Knoten

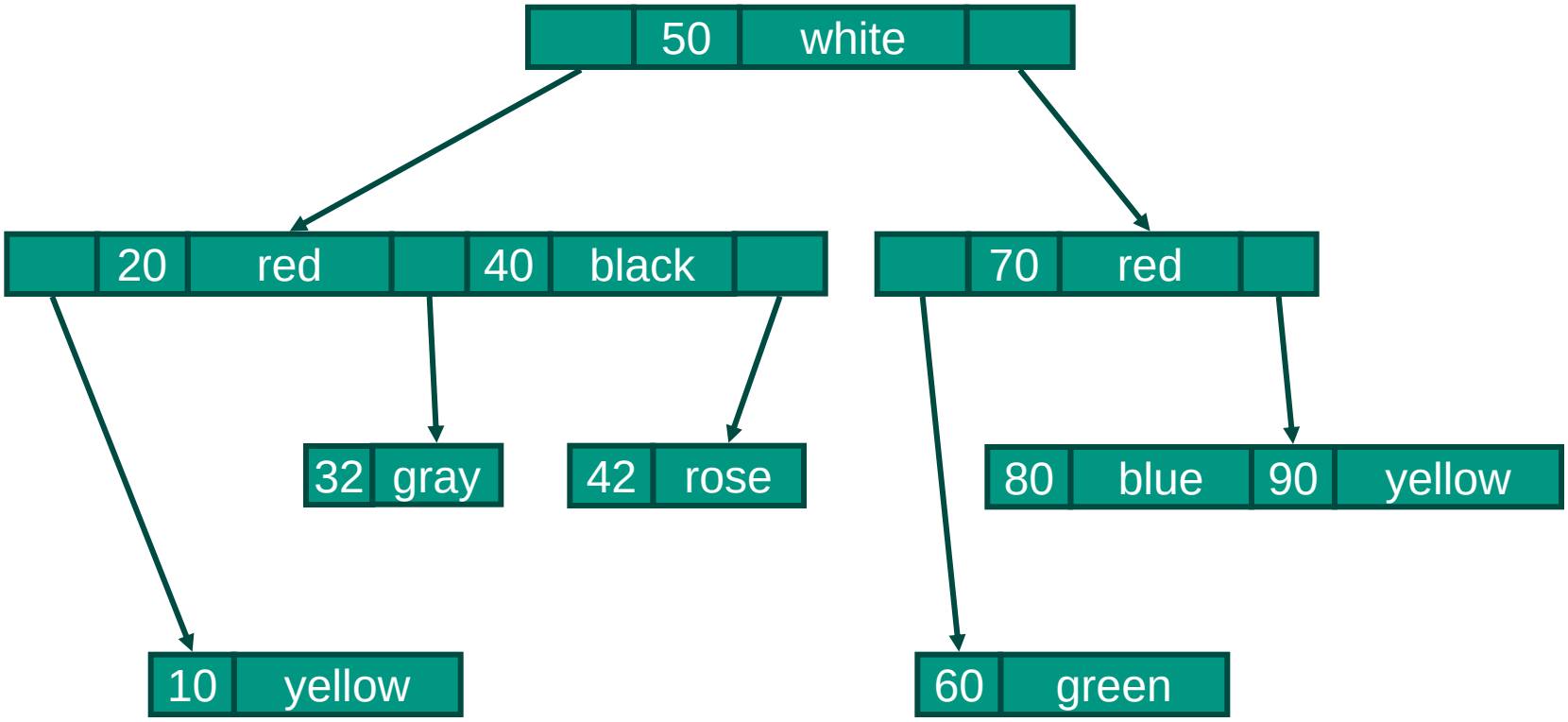
Entfernen – Beispiel 3

- Entfernen des Objekts [30, brown] (Innerer Knoten)



Entfernen – Beispiel 3

- Entfernen des Objekts [30, brown] (Innerer Knoten)



Entfernen

- Im B-Baum befinden sich „die meisten“ Knoten in Blättern
  - Hohe Wahrscheinlichkeit, dass Fall 1 zutrifft

- Aufwand
  - Vergleichsweise komplexes Vorgehen
  - Aufwand trotzdem vergleichbar zu Suchen und Einfügen
  - Insgesamt im Worst-Case $O(\left\lceil \frac{m}{2} \right\rceil \log_{\left\lceil \frac{m}{2} \right\rceil} n)$

Zwischenfazit

- Suchbäume einfache Datenstruktur
  - Effizienz abhängig von Balance
  - Worst-Case Einfügen, Suchen, Löschen linear
- Rot-Schwarz-Bäume erweiterten das Konzept
  - Balance wird in bestimmten Grenzen eingehalten
  - Worst-Case logarithmisch
- B-Bäume speichern mehrere Objekte in einem Knoten
  - Optimierungen für Hintergrundspeicher (Festplatte etc.)
  - Flexibel durch Ordnungsfaktor m
- Es existieren viele weitere Varianten von Bäumen
  - Eine Idee: Objekt über mehrere Knoten verteilt speichern
  - Digitale Bäume Abschnitt 6.5

6.5 Digitale Bäume

- Konzept bisher vorgestellter Bäume
  - Jeder Knoten nimmt einen Wert bzw. Schlüssel vollständig auf
- Probleme, wenn Worte als Werte von Knoten eines Suchbaumes gespeichert werden
  - Worte müssen aufwändig Buchstabe für Buchstabe verglichen werden
  - Beispiel: „Organisationslehre“ < „Organisationsmuster“
  - Kommen Wortanfänge mehrfach vor
    - Geht Speicherplatz verloren
    - Müssen Vergleiche mehrfach bis zum ersten unterschiedlichen Buchstaben durchgeführt werden
- Idee: Schlüssel besteht aus Zeichen aus Alphabet S
  - Zeichen des Schlüssels werden über mehrere Knoten verteilt gespeichert

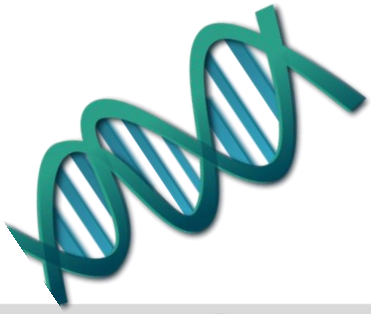

[Sedg10]

Anwendungen

- Suchmaschinen
  - Suchen nach Worten über einem Alphabet S z.B. in Texten oder HTML-Seiten durch WWW-Suchmaschinen, Content-Management-Systeme etc.
  - Alphabet $S_{Suche} = \{A, B, C, \ldots, Z\}$

- Internet-Router
  - Bei gegebener Zieladresse den Nachbarknoten finden, an den Paket zu senden ist (Routing-Tabelle)
  - Alphabet $S_{Routing} = \{0,1\}$

- Bio-Informatik
  - Suche von Gen-Sequenzen
  - Alphabet von Genen überschaubar $S_{Gene} = \{A, C, G, T\}$



Idee

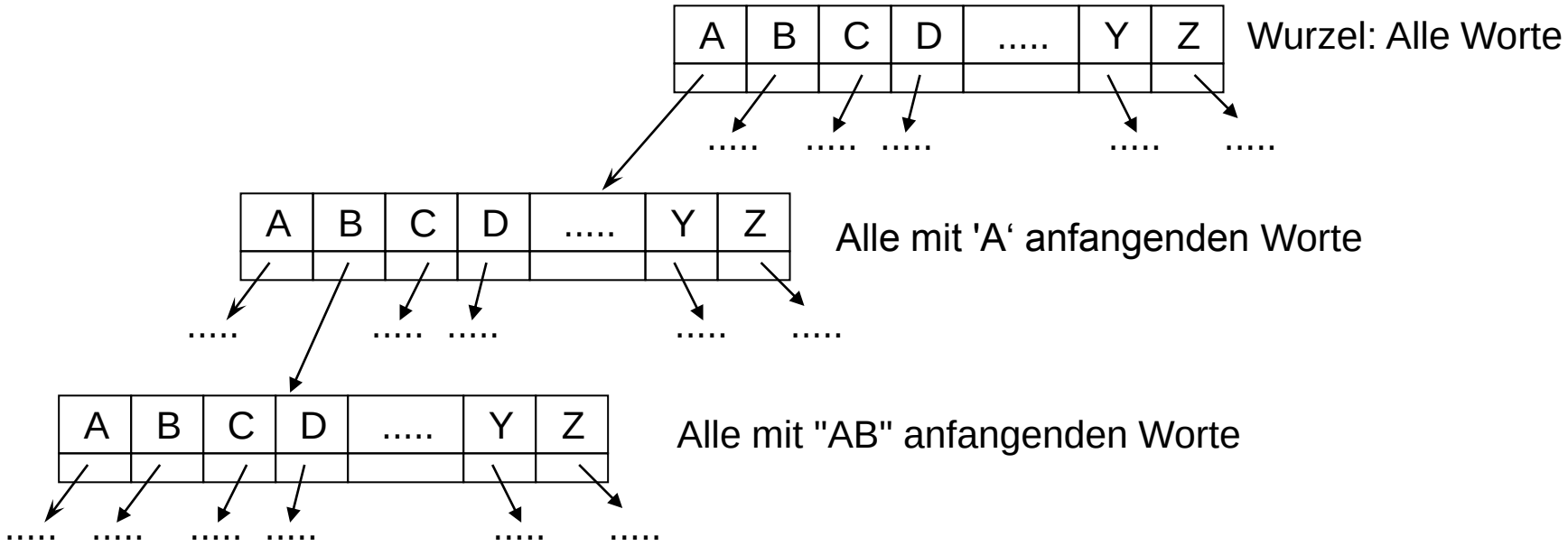
- Suchbaum mit mehrwertigen Knoten, deren Werte
  - Den Buchstaben des Alphabets entsprechen
  - Jeweils auf einen Kindknoten verweisen

- Das Verfolgen von Verweisen bis zu einem Blatt ergibt dann die gespeicherten Worte
  - Prinzipiell unabhängig von den gespeicherten Schlüsselwerten
  - Geeignet für Datentypen, bei denen eine derartige feste Verzweigung sinnvoll ist
    - Zeichenketten über einem festen Alphabet mit Verzweigung nach dem jeweils ersten bzw. nächsten Buchstaben
  - Bezeichnung digital entstammt der Interpretation von Zahlen eines Zahlensystems als Worte über dem Alphabet der Ziffern

- Tries setzen Konzept digitaler Bäume um

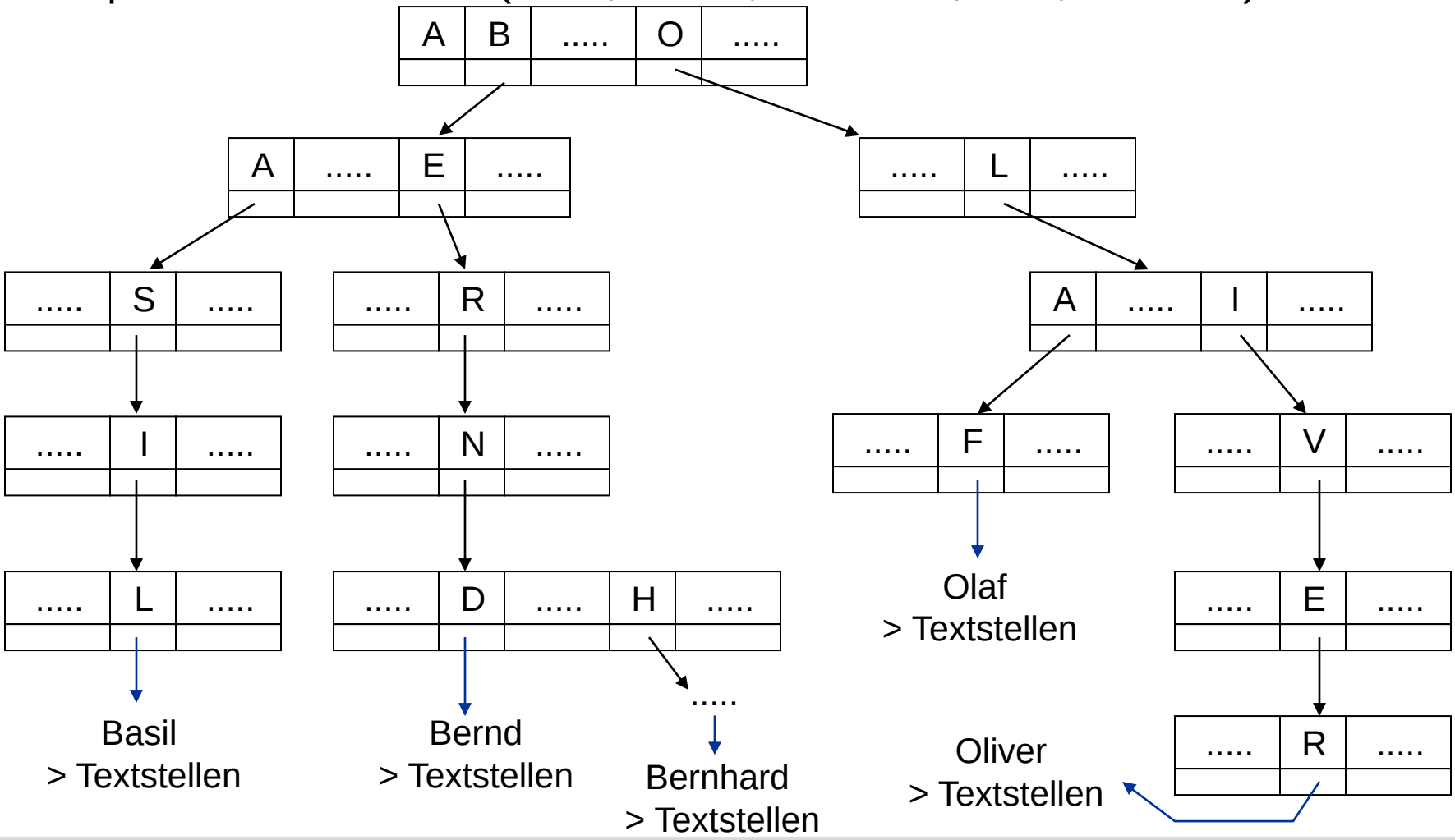
Tries

- Das Wort Trie (gesprochen „try“) entstammt der Hauptanwendung der Struktur, dem Text-Retrieval.
  - Tries erlauben große Dokumentenbestände nach Zeichenketten zu durchsuchen
  - Der Trie bildet einen sogenannten Index zu diesen Dokumenten



Trie-Beispiel

- Beispiel: Namens-Index (Basil, Bernd, Bernhard, Olaf, Oliver ...)

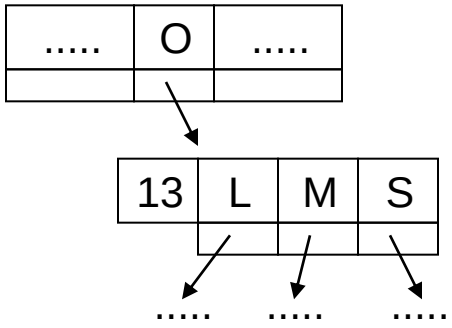


Probleme von Tries

- Ungleichmäßig verteilte Daten führen zu unbalancierten (teilweise sogar stark entarteten) Suchbäumen
  - Nicht vorkommende Buchstabenkombinationen ("QX" "QXY" etc.) führen zum Ende des Suchpfades bzw. zu Werten in Knoten ohne Verweise
  - Längere Worte ohne verwandte Worte führen zu Knoten, die nur einen Nachfolger haben und entarten im weiteren Verlauf zu Listen
    - Erzeugt ein ungünstiges Verhältnis von inneren Knoten zu Blättern
- Beispiel: „Xylophonspielerinnenvereinigung“
  - Ab 'y' folgt eine Liste von 29 Knoten, von denen jeder nur ein Kind hat (sofern keine ähnlichen Worte enthalten sind)
  - Unnötig lange Folgen von Knoten müssen verfolgt werden
- Mögliche Lösung: Patricia-Tries

Patricia-Trie

- Patricia
  - Pracitical Algorithm to Retrieve Information Coded in Alphanumeric
  - Ursprüngliche Form bezieht sich auf Binärzahlen
- Idee
  - Teile von Zeichenketten, die im weiteren Vergleich nicht zu Verzweigungen führen, werden übersprungen
  - Jeder Knoten enthält Anzahl der zu überspringenden Zeichen sowie nur die Buchstaben, die danach zu Vergleichen lohnt
- Beispiel

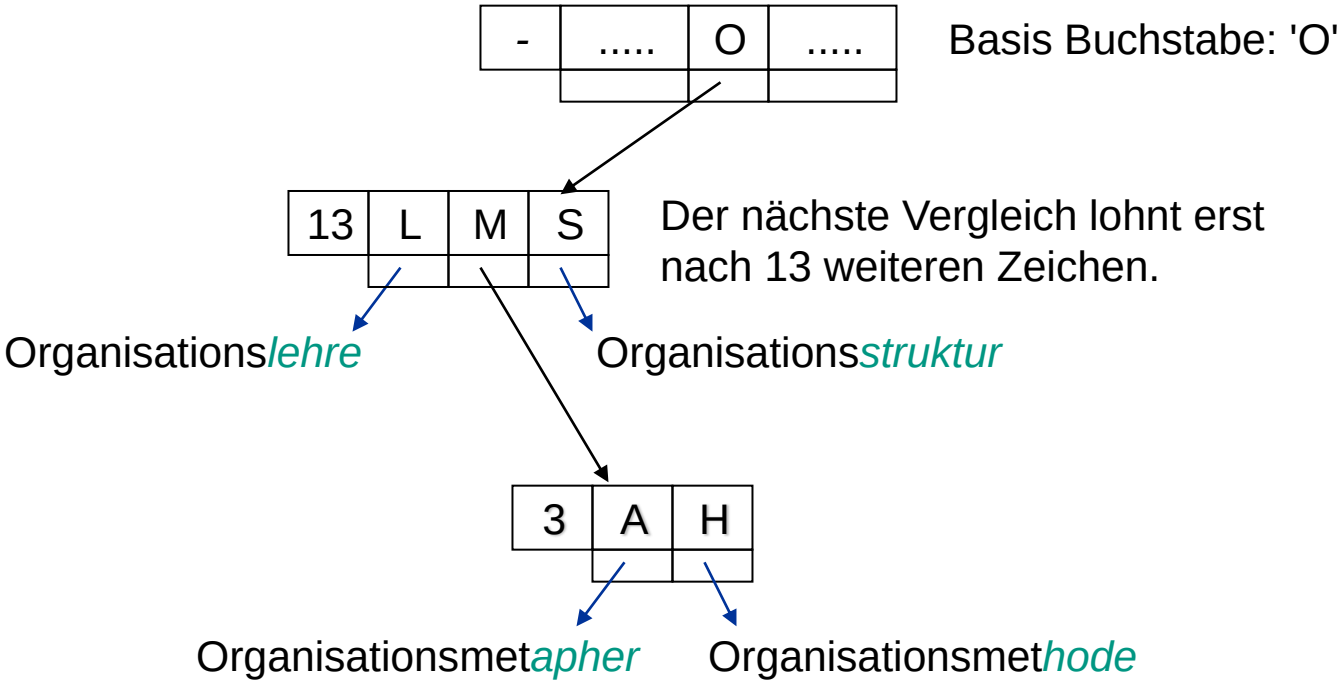


Basis ist Buchstabe 'O'

Der nächste Vergleich lohnt erst nach 13 weiteren Zeichen und nur mit den Zeichen 'L', 'M', und 'S', ..

Patricia-Trie-Beispiel

- Speicherung der Worte Organisationslehre, Organisationsstruktur, Organisationsmetapher und Organisationsmethode



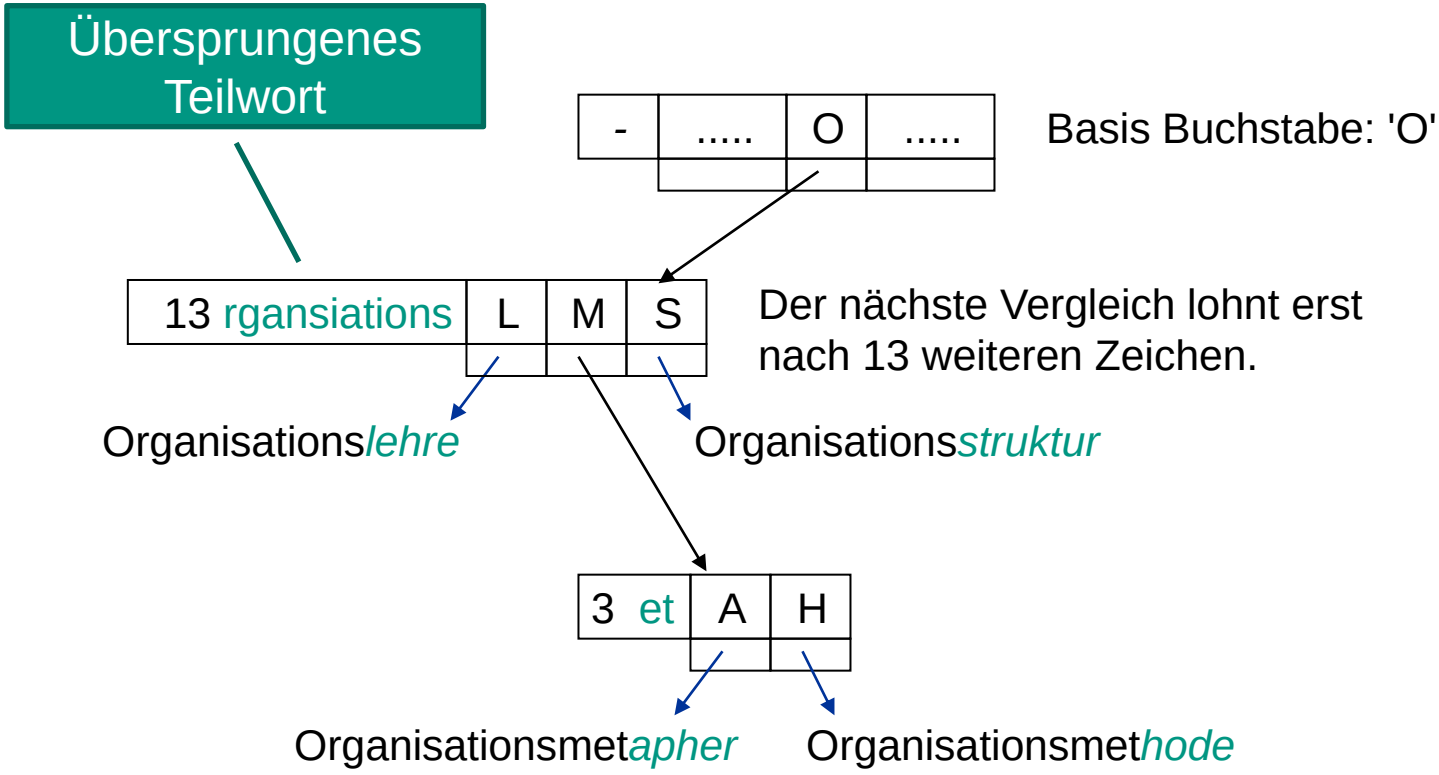
Patricia-Trie und Präfix-Bäume

- Vorteile
  - Eine gegenüber Tries komprimierte Darstellung
  - Reduzierung der bei der Suche zu „durchwandernden“ Knoten insbesondere bei sehr langen „einsamen“ Worten

- Variante des Patricia-Baumes: der Präfix-Baum
  - Zusätzlich zum Index des zu testenden Zeichens wird auch der Wert des übersprungenen Teilwortes im Knoten abgespeichert.

Präfix-Baum-Beispiel

- Speicherung der Worte Organisationslehre, Organisationsstruktur, Organisationsmetapher und Organisationsmethode



Komplexitätsanalyse

- Länge des längsten Pfades ist Länge der größten Übereinstimmung zweier Schlüssel
  - In vielen Anwendungen recht klein
- Suchen, Einfügen und Löschen von der Methodik ähnlich zu (binären) Suchbäumen
  - Entscheidend jetzt jedoch Länge der Schlüssel
  - Im Worst-Case $O(b)$, mit Anzahl Zeichen des größten Schlüssels = b
  - Im Average-Case $O(\log n)$, bei n gespeicherten Schlüsseln
    - Basis abhängig von der Größe des Alphabets S
- Patrica Tries verbessern Average-Case
  - Abhängig von den konkret gespeicherten Schlüsseln, dem Alphabet und der Verteilung
  - Worst-Case bleibt jedoch gleich

Zwischenfazit

- Suchbäume einfache Datenstruktur
  - Effizienz abhängig von Balance
  - Worst-Case Einfügen, Suchen, Löschen linear
- Rot-Schwarz-Bäume erweiterten das Konzept
  - Balance wird in bestimmten Grenzen eingehalten
  - Worst-Case logarithmisch
- B-Bäume speichern mehrere Objekte in einem Knoten
  - Optimierungen für Hintergrundspeicher (Festplatte etc.)
  - Flexibel durch Ordnungsfaktor m
- Es existieren viele weitere Varianten von Bäumen
  - Eine Idee: Objekt über mehrere Knoten verteilt speichern

Fazit

- Bäume als effiziente Datenstruktur zur Speicherung verschiedenster Informationen
- Komplexität der Grundoperationen oft in $O(\lg n)$
  - Trotzdem Unterschiede in der Praxis
- Verschiedene Einsatzzwecke beachten!
  - Einfache Suchbäume für kleine Datenmengen
  - Rot/Schwarz-Bäume für effiziente Organisation größerer Datenmengen
  - B-Bäume zur effizienten Verwaltung von Daten auf Hintergrundspeichern
  - Digital Bäume zur Indizierung und effizienten Speicherung geeigneter Datenmengen mit festgelegtem Alphabet

Literatur 

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3
- [Sedg10] Robert Sedgewick, „Algorithmen“, Addison-Wesley, 2. Auflage, 2002, 742 Seiten, ISBN 3-8273-7032-9

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 7 – Graphrepräsentation

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

I. Einführung

1. Einführung

II. Suchen und Sortieren

2. Sortieren

III. Datenstrukturen

3. Folgen als Felder und Listen
4. Hashing
5. Heaps
6. Sortierte Listen / Bäume

IV. Graphenalgorithmen

7. Graphrepräsentation

- 7.1 Motivation
- 7.2 Notation und Konventionen
- 7.3 Adjazenzlisten
- 7.4 Adjazenzmatrizen
- 7.5 Adjazenzfelder

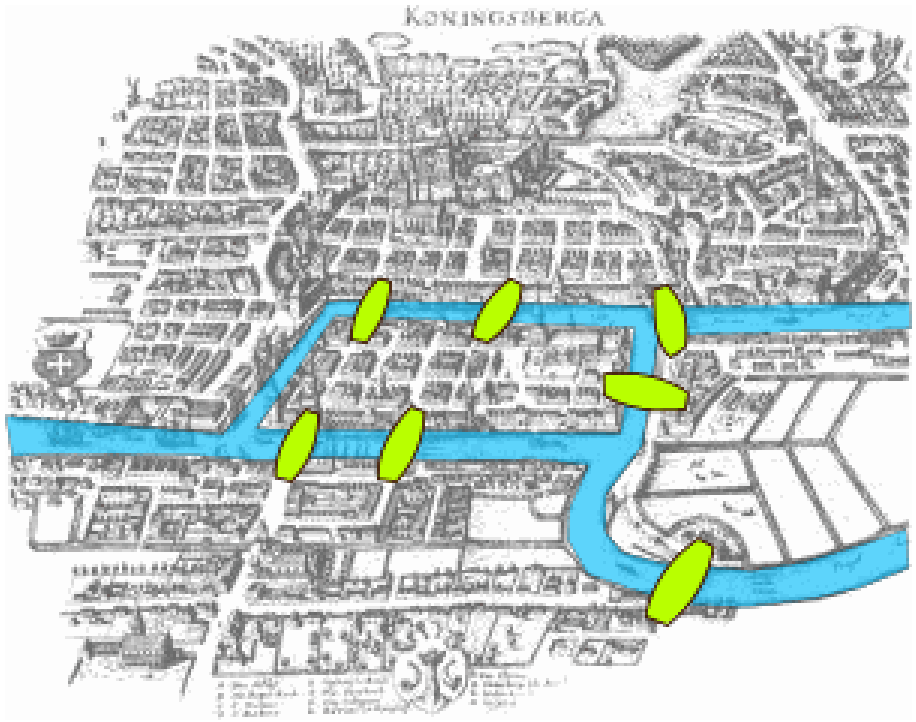
8. Graphtraversierung
9. Kürzeste Wege
10. Minimale Spannbäume

V. Ausblick

11. Generische Optimierungsansätze
12. Zusammenfassung und Ausblick

7.1 Motivation

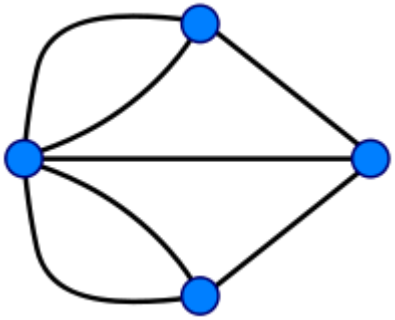
- *„Gibt es einen Rundgang durch die Stadt Königsberg, der jede der sieben Brücken genau einmal benutzt?“*



Motivation

- Nichtexistenz wurde 1736 von Leonhard Euler bewiesen
  - Methoden werden heute der Graphentheorie zugeordnet

- Vereinfachte Darstellung des Problems
  - Ortsteile werden zu Knoten
  - Brücken werden zu Kanten

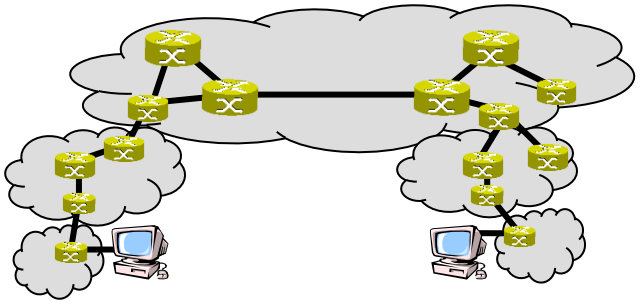


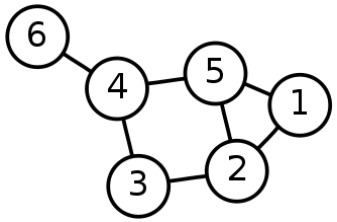
- Neuformulierung
  - *„Existiert ein Zyklus, der alle Kanten genau einmal benutzt“*
  - So ein Zyklus wird heute Eulerkreis genannt

- Eulerkreis existiert genau dann, wenn der Graph zusammenhängend ist und alle Knoten geraden Grad haben

Motivation

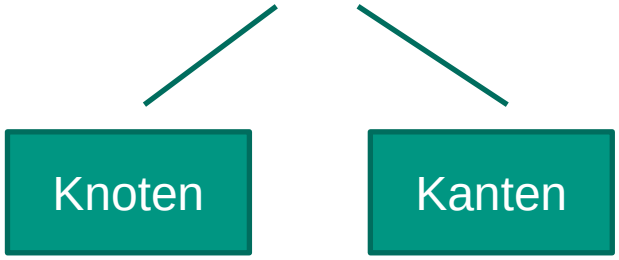
- Was ist ein Graph?
  - Das Haltestellen- und Schienennetz eines Verkehrsverbundes
  - Die Freundschaftsbeziehungen innerhalb eines sozialen Netzwerks
  - Internetrouter und die Kommunikations-verbindungen zwischen ihnen
  - Abstrakt: Eine Menge von Knoten und eine Menge von Kanten zwischen den Knoten





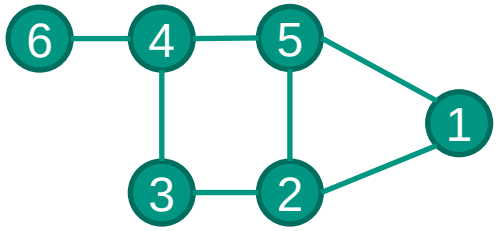
7.2 Notation und Konvention

- Ein Graph G ist definiert durch (V, E)



- Mit $E \subseteq V \times V$
  - Jede Kante $e \in E$ ist durch ein Knotenpaar dargestellt

- Gebräuchliche Notation
  - $n = |V|$
  - $m = |E|$



$\equiv$

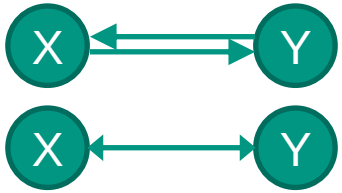
$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{(1,2), (1,5), (2,3), (2,5)$$
$$(3,4), (4,5), (4,6)\}$$
$$n = 6,$$
$$m = 7$$

Ungerichtete und Gerichtete Graphen

- Gerichteter Graph
  - Eine Kante ist eine unidirektionale Beziehung
  - Dargestellt durch Pfeile
  - Bei gerichteten Graphen ist in der Mengen-schreibweise die Reihenfolge ausschlaggebend
    - Von X nach Y: $\{(X,Y)\} \neq \{(Y,X)\}$: von Y nach X

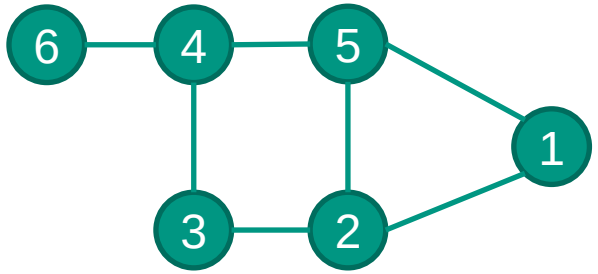
- Ungerichteter Graph
  - Jede Kante ist eine bidirektionale Beziehung
  - Dargestellt durch einfache Linie

- Andere Darstellungsmöglichkeiten bidirektionaler Beziehungen
  - Werden auch in gerichteten Graphen verwendet
  - Ein Pfeil je Richtung
  - Linien mit zwei Pfeilenden

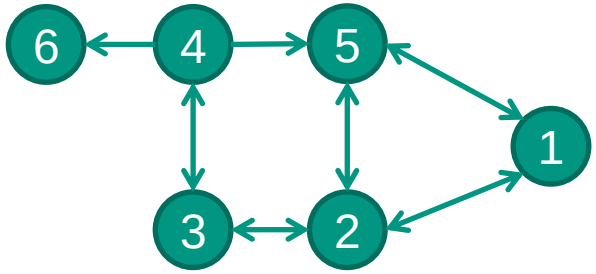


Definitionen

- In ungerichteten Graphen
  - Grad $d(n)$ eines Knotens n ist die Anzahl der Kanten, die den Knoten mit anderen Knoten verbinden
  - Beispiel: $d(5) = 3$

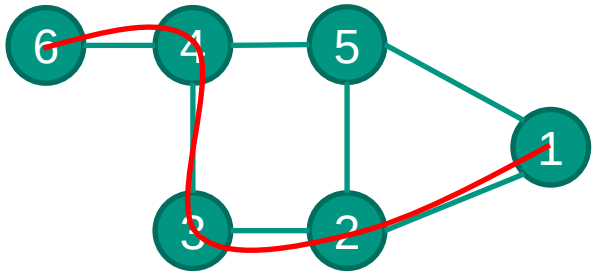


- In gerichteten Graphen
  - Ausgangsgrad $d^+(n)$
    - Anzahl der ausgehenden Kanten
    - Beispiel: $d^+(5) = 2$
  - Eingangsgrad $d^-(n)$
    - Anzahl der eingehenden Kanten
    - Beispiel: $d^-(5) = 3$



Definitionen

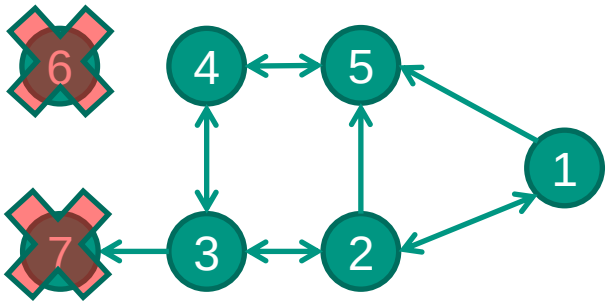
- Pfad (auch Weg, Kantenzug)
  - Endliche Folge von Knoten $v_1 \ldots v_n$ so dass $\forall\, 1 \leq k < n$ gilt $(v_k, v_{k+1}) \in E$



- Zusammenhängender Graph
  - Für alle $v, w \in V$ gilt: Es existiert ein Pfad, der v und w verbindet

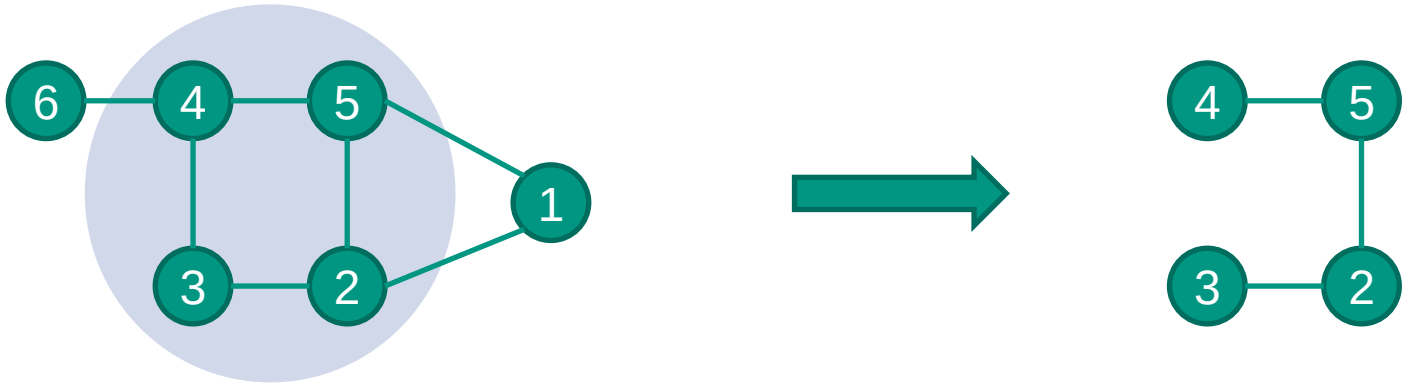
Es existiert kein Pfad, der Knoten 6 mit einem anderen Knoten verbindet

Es existiert kein Pfad, der Knoten 7 bspw. mit Knoten 3 verbindet



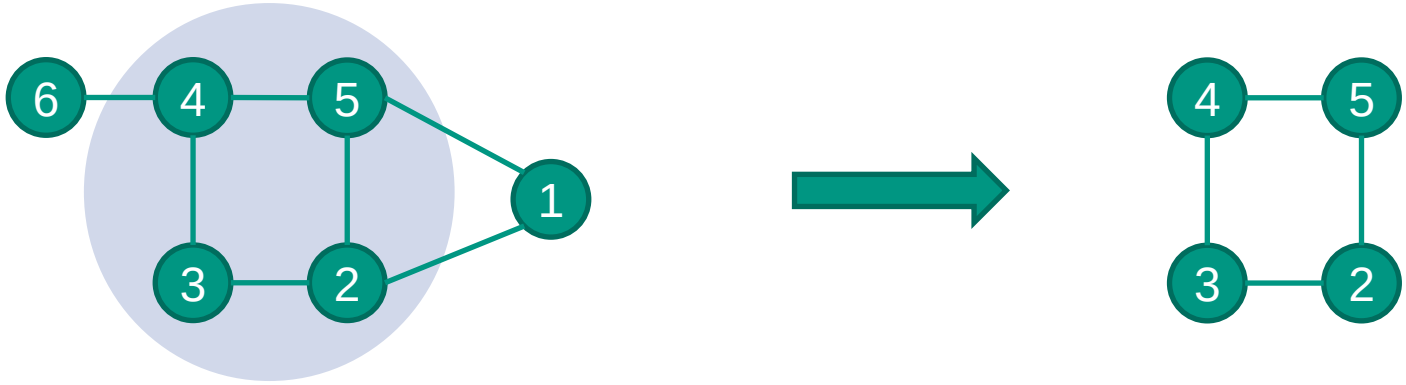
Definitionen

- Ein Graph $G' = (V', E')$ ist ein Teilgraph eines Graphen $G = (V, E)$ genau dann wenn $V' \subseteq V$ und $E' \subseteq E$
  - Knotenauswahl und Kantenauswahl ist unabhängig voneinander



Definitionen

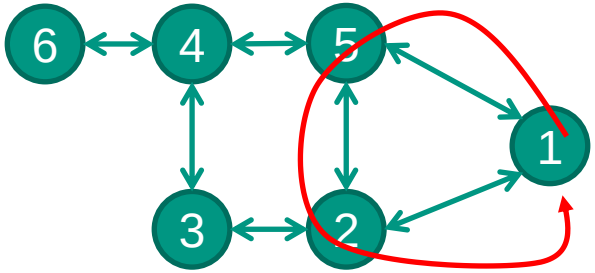
- Ein Graph $G' = (V', E')$ ist ein induzierter Teilgraph eines Graphen $G = (V, E)$ genau dann wenn $V' \subseteq V$ und $E' = \{(u, v) \in E \mid u, v \in V'\}$
  - Induzierte Teilgraphen sind durch die Knotenmenge eindeutig festgelegt



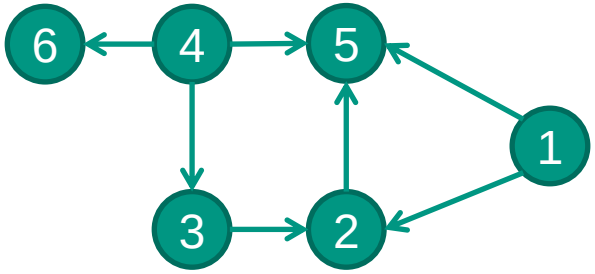
Definitionen

- Schleife
  - Kante, deren Ausgangs- und Endknoten derselbe ist: $(v, v) \in E$

- Zyklus
  - Pfad $(v_1, \ldots, v_n)$ mit $v_n = v_1, \quad n \geq 1$

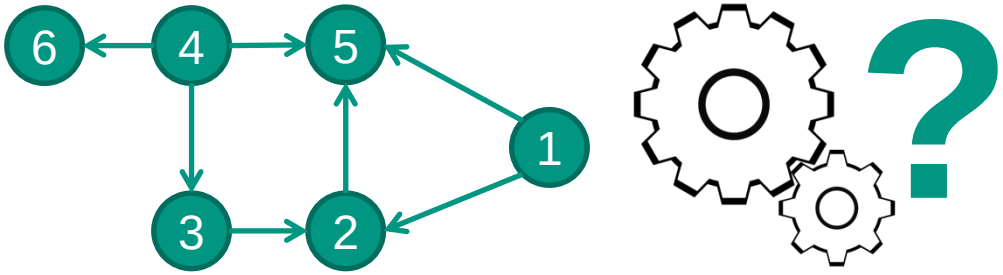


- Gerichteter azyklischer Graph (DAG, Directed Acyclic Graph)
  - Gerichteter Graph ohne Zyklen
  - Knoten v eines DAG heißt Wurzel, falls $d^-(v) = 0$
  - DAG mit nur einer Wurzel heißt Wurzelgraph

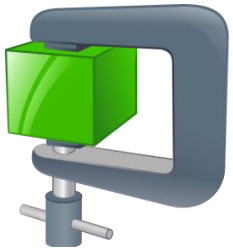


Darstellung von Graphen

- Graphische Darstellung für Menschen intuitiv lesbar
  - Für maschinelle Bearbeitung ungeeignet



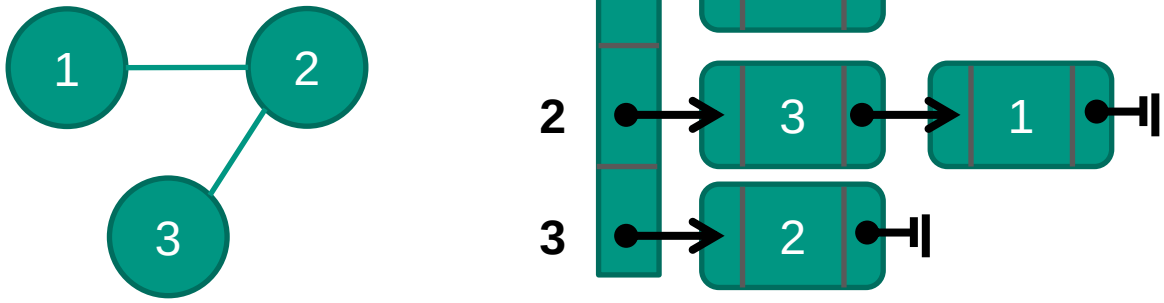
- Verschiedene Verfahren zur Repräsentation mit unterschiedlicher Zielsetzung
  - Geschwindigkeit
  - Geringer Speicherverbrauch



- Nachfolgend werden drei mögliche Repräsentationen betrachtet

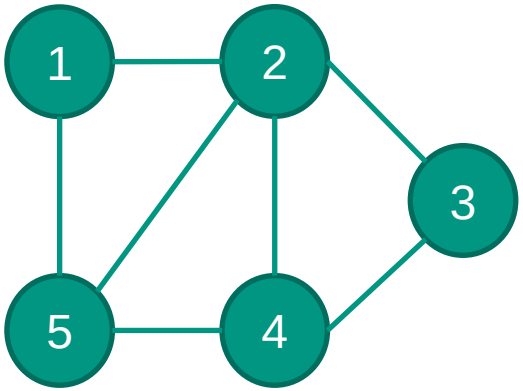
7.3 Adjazenzlisten

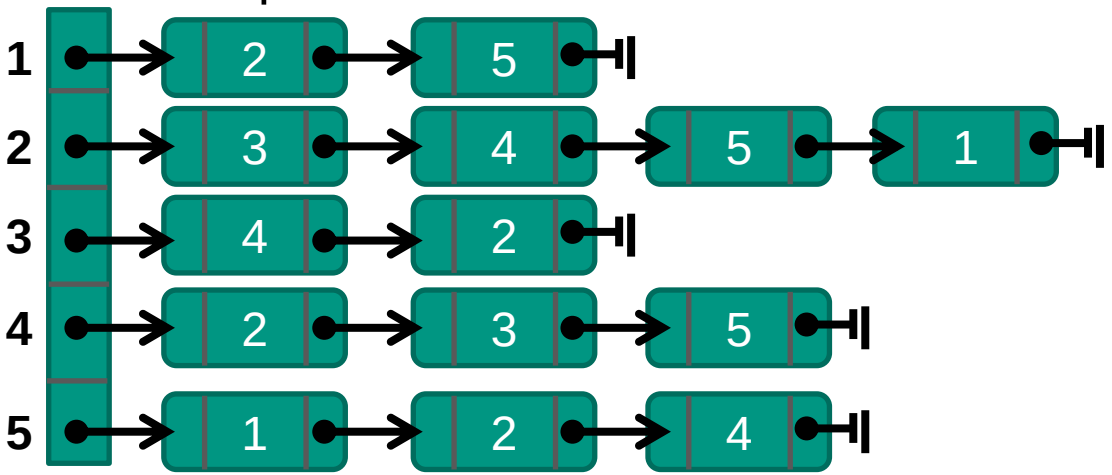
- Adjazenzlistenrepräsentation eines Graphen $G = (V, E)$
  - Feld Adj der Länge $|V|$
    - Jedem Knoten aus V ist ein Eintrag in Adj zugeordnet
  - Für jeden Knoten $u \in V$ enthält die Liste $Adj[u]$ alle Knoten v für die eine Kante $(u, v) \in E$ existiert
    - $\Rightarrow Adj[u]$ enthält alle Knoten in G, die durch eine Kante an u angrenzen



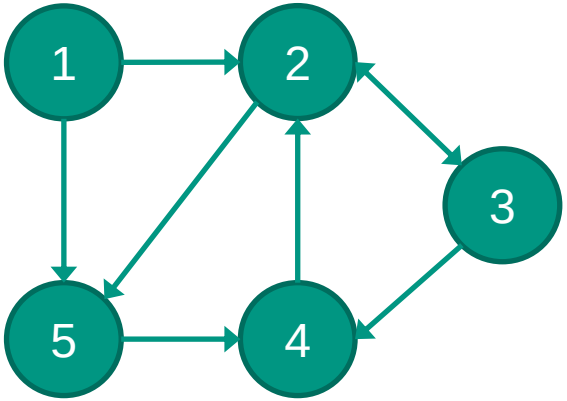
Adjazenzlisten

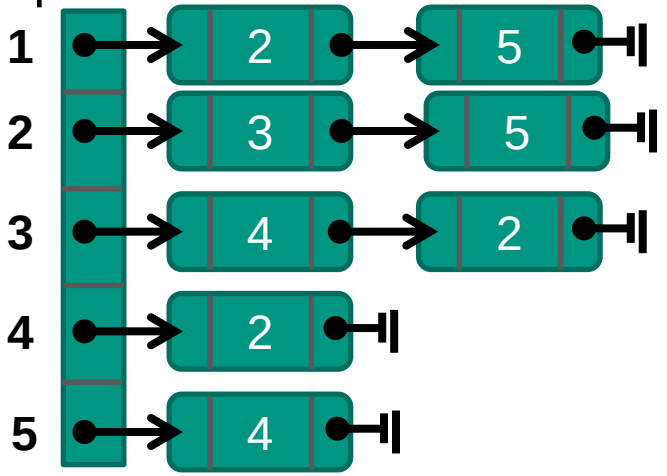
- Adjazenzlisten von ungerichteten Graphen





- Adjazenzlisten von gerichteten Graphen





Adjanzenzlisten – Speicherverbrauch

- Maximaler Speicherverbrauch
  - Tritt ein bei Vollvermaschung → $\forall\, u, v \in V: (u, v) \in E$
    - Auch vollständiger Graph genannt
  - $Adj[u]$ enthält alle Knoten außer u selbst
  - → $n^2 - n$ Einträge in verketteten Listen, n Einträge in Adj

- Minimaler Speicherverbrauch
  - Tritt ein bei Graphen ohne Kanten
  - Adj enthält keine verketteten Listen
  - → n (leere) Einträge in Adj

- Unscharfe Einteilung von Graphen in
  - Dünne Graphen → wenige Kanten: $|E| \ll |V|^2$
  - Dichte Graphen → viele Kanten: $|E| \sim |V|^2$

Adjazenzlisten – Vor- und Nachteile

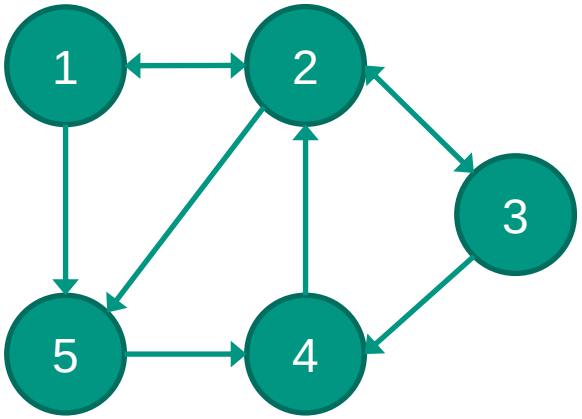
- Vorteile
  - Kompakte Darstellung
    - Ein Zeiger pro Knoten
    - Zwei Zeiger pro Kante
    - Speicherverbrauch in $\Theta(V + E)$
    - Dünne Graphen besonders kompakt
  - Einfache Kantenoperationen
    - Einfügen und Löschen in verketteter Liste

- Nachteile
  - Kostspielige Suche nach Kanten
    - Existiert Kante von 2 nach 4?
    - → Lineare Suche in Kantenliste von Knoten 2

7.4 Adjazenzmatrizen

- Adjazenzmatrizenrepräsentation eines Graphen $G = (V, E)$
  - Adjazenzmatrix ist eine $|V| \times |V|$ Matrix $A = (a_{ij})$ so, dass
  - $a_{ij} = \begin{cases} 1, & \text{wenn } (i,j) \in E \\ 0, & \text{sonst} \end{cases}$

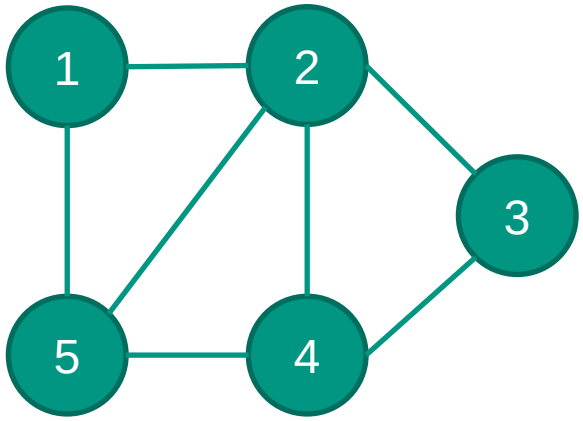
- Beispiel: Gerichteter Graph



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

Adjazenzmatrizen

- Beispiel: Ungerichteter Graph



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

Hauptdiagonale

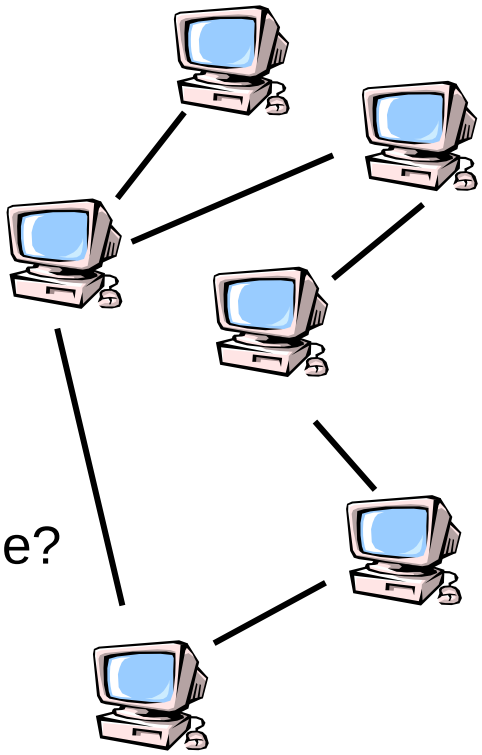
- Bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch
  - Spiegelung an der Hauptdiagonalen, also $a_{ij} = a_{ji}$

Adjazenzmatrizen – Speicherverbrauch

- Konstanter Speicherverbrauch
  - Abhängig von Anzahl Knoten in G
- Bei gerichteten Graphen: $|V| \times |V|$ - Matrix
  - → $\Theta(n^2)$ Speicherverbrauch mit $n = |V|$
- Bei ungerichteten Graphen
  - Optimierungsmöglichkeit durch Symmetrieeigenschaft der Matrix
  - → Untere Hälfte der Matrix muss nicht gespeichert werden
    - Bei Anfrage nach a_{ij} mit $i > j$ wird a_{ji} zurückgegeben
  - → $\sum_{k=1}^{n} k$ Speicherverbrauch
    - 1. Zeile n Einträge
    - 2. Zeile $n - 1$ Einträge
    - …
    - n-te Zeile 1 Eintrag

Anwendungsbeispiel Adjazenzmatrizen

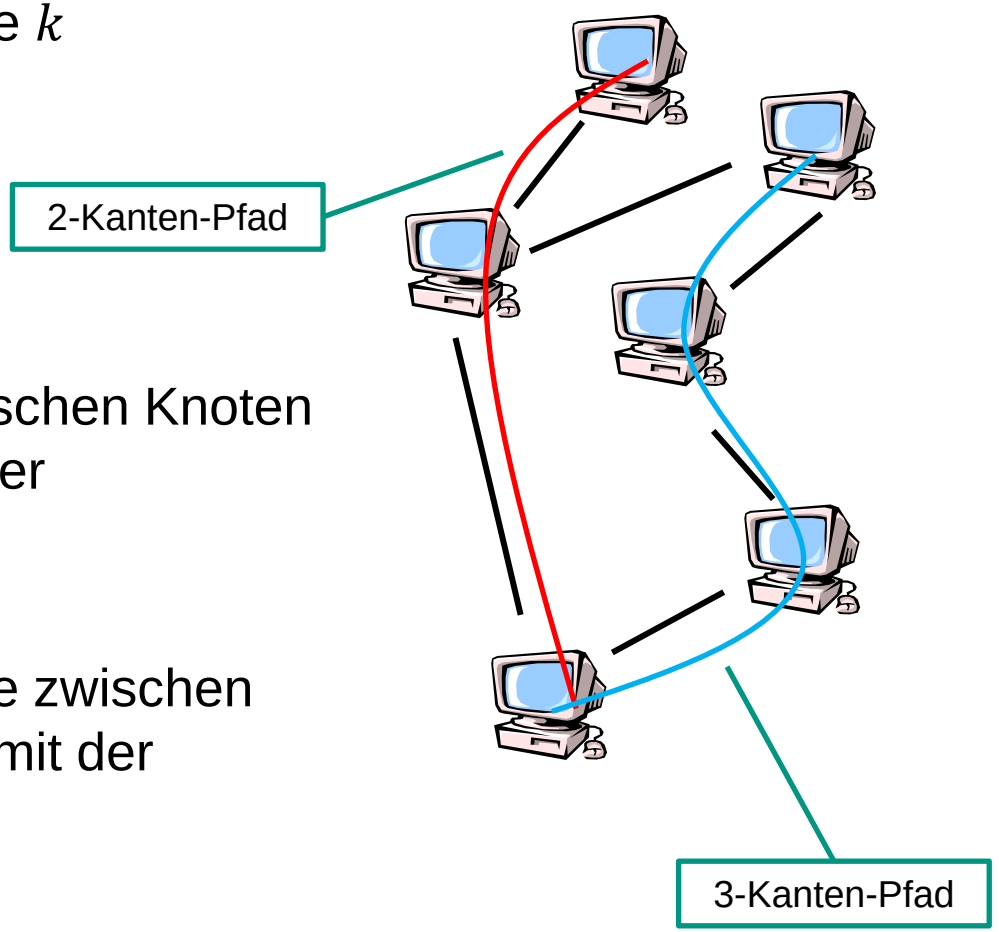
- Gegeben sei ein Computernetz
- Vorgabe: Jedes System soll mit jedem anderen System über maximal ein anderes System kommunizieren können?
  - Anders: Zwischen zwei Knoten soll immer mindestens ein Pfad der Länge 1 oder 2 existieren
- Frage: Erfüllt das nebenstehende Netz diese Vorgabe?
- Adjazenzmatrix-Darstellung erlaubt einfache Lösung dieser Frage



- Gesucht: Knotenpaare die weder direkt verbunden (Pfad der Länge 1), noch durch einen Pfad der Länge 2 verbunden sind

k-Kanten-Pfade

- k-Kanten-Pfad = Pfad der Länge k

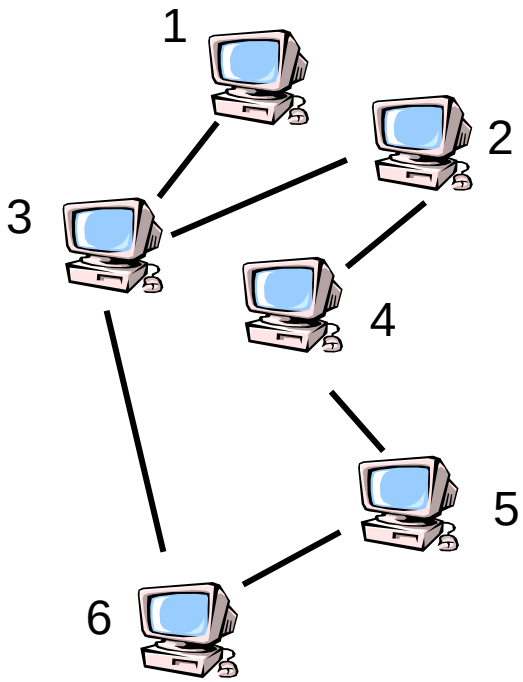


- Anzahl der k-Kanten-Pfade zwischen Knoten lässt sich an der k-ten Potenz der Adjazenzmatrix ablesen

- Beispiel: Anzahl 3-Kanten-Pfade zwischen Knoten 1 und 4 eines Graphen mit der Adjazenzmatrix A → $A^3{}_{1,4}$

Anwendungsbeispiel Adjazenzmatrizen

- Adjazenzmatrix des Netzes



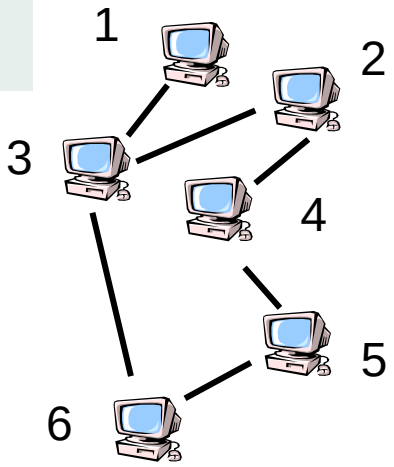
	1	2	3	4	5	6
1	0	0	1	0	0	0
2	0	0	1	1	0	0
3	1	1	0	0	0	1
4	0	1	0	0	1	0
5	0	0	0	1	0	1
6	0	0	1	0	1	0

Anwendungsbeispiel Adjazenzmatrizen

- Matrixpotenzierung der Adjazenzmatrix

$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 0 & 1 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 1 & 0 & 0 \\
3 & 1 & 1 & 0 & 0 & 0 & 1 \\
4 & 0 & 1 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 1 \\
6 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}^2 =$$

	1	2	3	4	5	6
1	1	1	0	0	0	1
2	1	2	0	0	1	1
3	0	0	3	1	1	0
4	0	0	1	2	0	1
5	0	1	1	0	2	0
6	1	1	0	1	0	2



- A^2 enthält für jedes Knotenpaar die Anzahl der 2-Kanten-Pfade die sie verbinden
  - → Es fehlen noch die 1-Kanten-Pfade

Anwendungsbeispiel Adjazenzmatrizen

- $A^2 + A$ enthält für jedes Knotenpaar die Anzahl der 2-Kanten-Pfade + die Anzahl der 1-Kanten-Pfade, die sie verbinden

A^2

	1	2	3	4	5	6
1	1	1	0	0	0	1
2	1	2	0	0	1	1
3	0	0	3	1	1	0
4	0	0	1	2	0	1
5	0	1	1	0	2	0
6	1	1	0	1	0	2

$+$

A

	1	2	3	4	5	6
1	0	0	1	0	0	0
2	0	0	1	1	0	0
3	1	1	0	0	0	1
4	0	1	0	0	1	0
5	0	0	0	1	0	1
6	0	0	1	0	1	0

$=$

$A^2 + A$

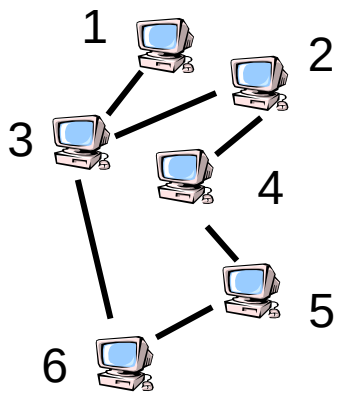
	1	2	3	4	5	6
1	1	1	1	0	0	1
2	1	2	1	1	1	1
3	1	1	3	1	1	1
4	0	1	1	2	1	1
5	0	1	1	1	2	1
6	1	1	1	1	1	2

Anwendungsbeispiel Adjazenzmatrizen

$$A^2 + A$$

	1	2	3	4	5	6
1	1	1	1	0	0	1
2	1	2	1	1	1	1
3	1	1	3	1	1	1
4	0	1	1	2	1	1
5	0	1	1	1	2	1
6	1	1	1	1	1	2

- 0 in $A^2 + A$ markiert Knotenpaare, für die kein Pfad der Länge 1 oder 2 zwischen ihnen existiert
- Von Knoten 1 nach jeweils Knoten 4 und 5 existiert kein solcher Pfad
  - Durch Symmetrie auch umgekehrt



- → Netz erfüllt die Anforderung nicht

Adjazenzmatrizen – Vor- und Nachteile

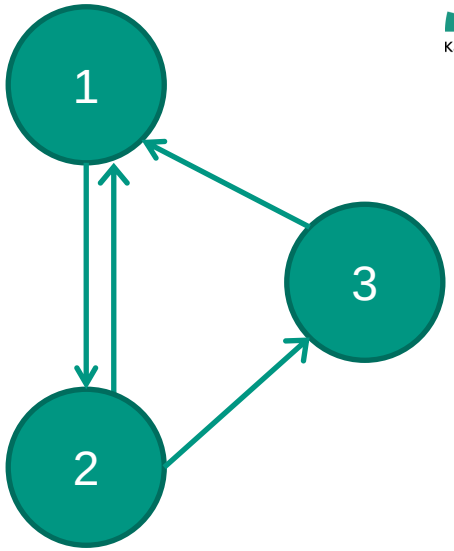
- Vorteile
  - Schneller Zugriff auf Kanten
    - Existiert Kante von 2 nach 4?
    - → Ein Zugriff auf a_{24}
  - Einfaches Ändern von Kanten
  - Speicherverbrauch unabhängig von Anzahl Kanten im Graph
  - Manche Graphenoperationen durch Matrizenoperationen durchführbar

- Nachteile
  - Speicherverbrauch von $\Theta(|V|^2)$
    - Bei ungerichteten Graphen Optimierungen mit konstantem Faktor möglich
  - Kein geringerer Speicherverbrauch bei dünnen Graphen

7.5 Adjazenzfelder

- Adjazenzfeldrepräsentation eines Graphen $G = (V, E)$
  - Zwei Felder
    - Knotenfeld V
    - Kantenfeld E
    - Häufig aus Effizienzgründen direkt hintereinander angeordnet
  - $V[n]$ speichert den Index in E, ab dem die ausgehenden Kanten von Knoten n in E aufgelistet sind
  - In E werden die Zielknoten der Kanten nacheinander abgespeichert
  - Ende der Auflistung der Kanten für Knoten n ist implizit durch $V[n+1]$, dem Index der Kantenliste für den nächsten Knoten, gegeben
  - → Alle Kanten in E von Index $V[n]$ bis exklusive Index $V[n+1]$ sind Ausgangskanten von n
    - Für einfachere Implementierung wird häufig Dummy-Element an das Ende von V angehängt, das hinter die letzte Kante in E zeigt

Adjazenzfelder – Aufbau



- Gegeben Graph G
- $|V| = 3$ → Knotenfeld der Länge 3
- $|E| = 4$ → Kantenfeld der Länge 4
- $V[1] =$ erster freier Eintrag in E → 1

- Für jede ausgehende Kante eines Knotens wird der Zielknoten in E eingetragen
  - $E[1] = 2$
- $V[2] =$ nächster freier Eintrag in E → 2
  - $E[2] = 1$
  - $E[3] = 3$
- $V[3] = 4$
  - $E[4] = 1$

V

Index	1	2	3
Wert	1	2	4

E

Index	1	2	3	4
Wert	2	1	3	1

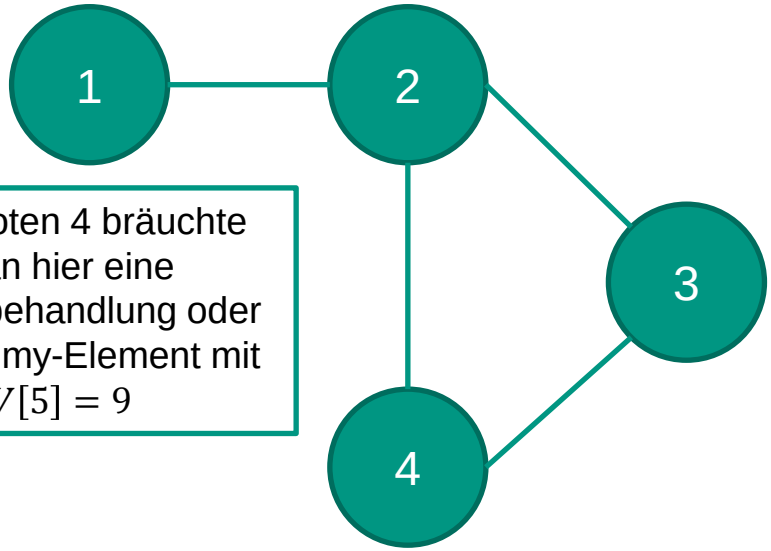
Adjazenzfelder – Interpretation

Index	1	2	3	4	1	2	3	4	5	6	7	8
Wert	1	2	5	7	2	1	3	4	2	4	2	3

Knotenfeld V **Kantenfeld E**

- Beispiel an Knoten 3
  - Index der ersten Kante
    - $V[3] = 5$
  - Index der letzten Kante
    - $V[3+1] - 1 = 6$
  - Zielknoten der Kanten
    - $E[5] = 2$
    - $E[6] = 4$

Für Knoten 4 bräuchte man hier eine Sonderbehandlung oder ein Dummy-Element mit $V[5] = 9$



Adjazenzfelder – Vor und Nachteile

- Vorteile
  - Sehr platzsparende Repräsentation
    - Speicherverbrauch in $\Theta(V + E)$
    - Um einen konstanten Faktor weniger als Adjazenzlisten
- Nachteile
  - Änderungen sehr aufwändig
  - Benötigt Dummy-Eintrag um Ende explizit zu markieren

Zusammenfassung

- Ein Graph durch viele Repräsentationen darstellbar
- Hier vorgestellt
  - Adjazenzlisten
  - Adjazenzmatrizen
  - Adjazenzfelder

- „Beste“ Repräsentation ist abhängig von
  - Beschränkungen
    - Rechenzeit
    - Speicherplatz
  - Anwendung
    - Änderungen am Graphen oder statischer Graph
    - Zugriff auf Knoten oder Kantenbasis
    - Erwarteter Graph (beispielsweise dünn oder dicht)

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, *„Algorithmen – Eine Einführung “*, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, *„Algorithms and Data structures“*, Springer, 300 Seiten, ISBN 978-3-540-77977-3

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 8 – Graphtraversierung

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

I. Einführung

1. Einführung

II. Suchen und Sortieren

2. Sortieren

III. Datenstrukturen

3. Folgen als Felder und Listen
4. Hashing
5. Heaps
6. Sortierte Listen / Bäume

IV. Graphenalgorithmen

7. Graphrepräsentation

8. Graphtraversierung

- 8.1 Motivation
- 8.2 Tiefensuche
- 8.3 Topologisches Sortieren
- 8.4 Starke Zusammenhangskomponenten
- 8.5 Breitensuche
- 8.6 Bipartite Graphen

9. Kürzeste Pfade
10. Minimale Spannbäume

V. Ausblick

11. Generische Optimierungsansätze
12. Zusammenfassung und Ausblick

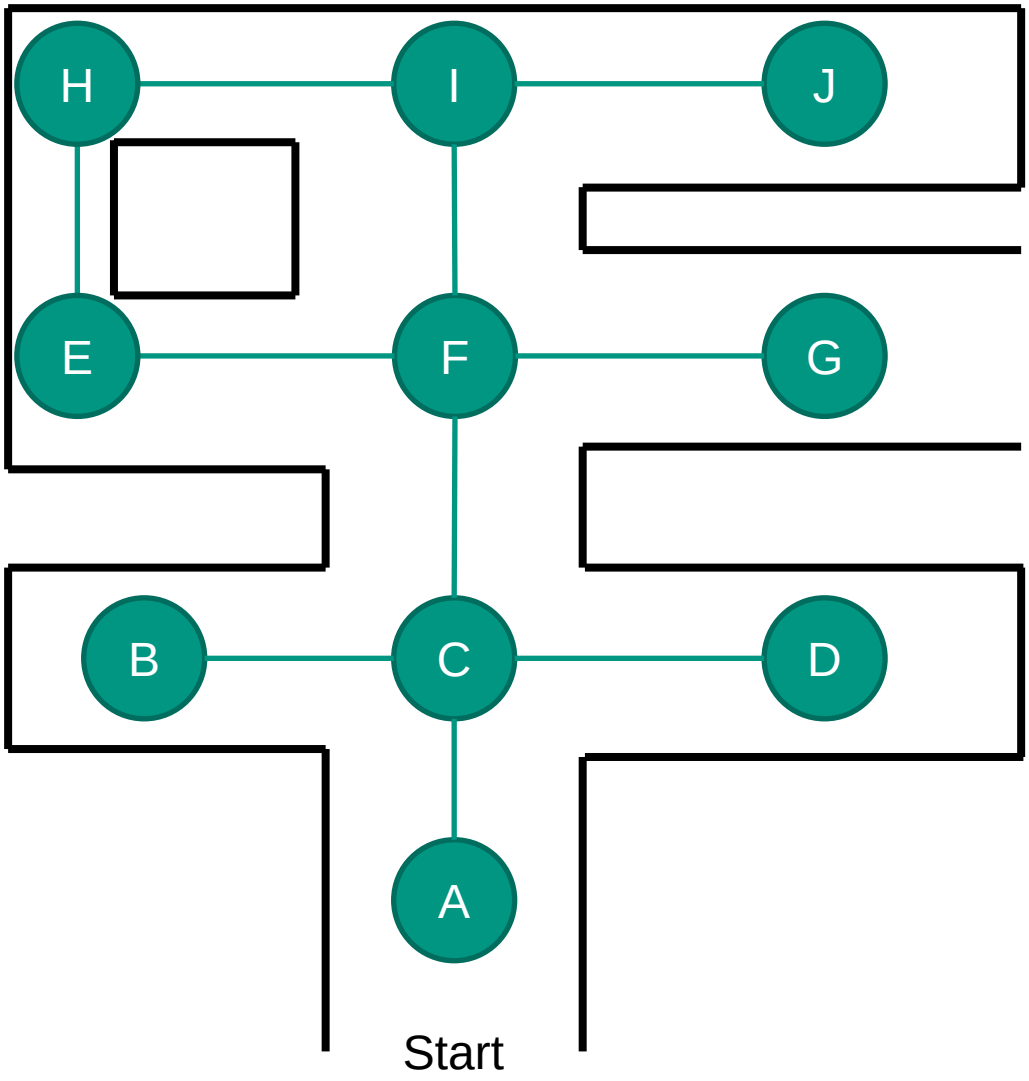
8.1 Motivation

- In der griechischen Mythologie soll Theseus den Minotaurus erschlagen, der in einem Labyrinth gefangen gehalten wird

- Zur Unterstützung wird ihm von Ariadne eine Rolle Faden mitgegeben

- Diesen Faden rollte er ab, während er das Labyrinth durchsuchte und konnte dadurch
  - Vermeiden gleiche Teile des Labyrinths mehrfach zu durchsuchen
  - Den Ausgang wiederfinden

Motivation



- Kreuzungen, Ecken und Sackgassen als Knoten interpretieren
- Gänge als Kanten interpretieren

Motivation

- Graph ist häufig Eingabe für verschiedenste Algorithmen
  - Meist sind strukturelle Informationen über den Graphen von Interesse
    - Anzahl Knoten
    - Anzahl Kanten
    - Kürzeste Wege (Distanz) zwischen Knoten
    - Ansammlungen von untereinander verbundenen Knoten
- → Methodisches Abarbeiten (Graphtraversierung) nötig
- Im Folgenden vorgestellte Algorithmen zur Graphtraversierung
  - Tiefensuche
  - Breitensuche

8.2 Tiefensuche

- Tiefensuche (Depth-First-Search, DFS) erkundet Graphen entlang der Kanten des als letzten entdeckten Knotens v, der noch nicht erkundete ausgehende Kanten besitzt

- Wenn alle Kanten von v erkundet sind, fährt die Suche mit dem Knoten fort, von dem aus v entdeckt wurde (Backtracking)

- Algorithmus stoppt, wenn alle Knoten, die vom ursprünglichen Startknoten erreichbar sind, gefunden wurden

- Falls weitere, bisher nicht gefundene, Knoten existieren
  - Wähle einen davon als neuen Startknoten
  - Wiederhole den Algorithmus

Farbkodierung

- Tiefensuche färbt Knoten ein um ihren Zustand zu kodieren
- Weiß = Unbesuchter Knoten
  - Alle Knoten werden mit Weiß initialisiert
- Grau = Entdeckter Knoten
  - Knoten wird vom Algorithmus betrachtet
  - Es existieren noch ausgehende Kanten, die noch nicht betrachtet wurden
- Schwarz = Abgeschlossener Knoten
  - Betrachtung des Knoten ist abgeschlossen
  - Alle ausgehenden Kanten wurden besucht

Zeitstempel

- Zeitstempel
  - Jeder Knoten hat zwei Zeitstempel
  - Bei entdecken des Knotens (= grau färben) wird der $discovered$ Zeitstempel gesetzt
  - Bei Abschließen der Betrachtung eines Knotens (= schwarz färben) wird der $finalized$ Zeitstempel gesetzt
  - Zeitstempel werden als fortlaufende Ganzzahlen dargestellt
  - Bei Entdeckung oder Abschließen werden sie inkrementiert
    - → Zeitstempel sind zwischen 1 und $2|V|$

- Zeitstempel tragen nicht zur Funktion des Algorithmus bei
  - Bieten Informationen über die Struktur des Graphen
  - Für Anwendungszwecke häufig interessant

Depth-First-Forest

- Nebenprodukt der Tiefensuche ist der Depth-First Forest
  - Beim Entdecken eines Knotens wird der Knoten von dem er aus entdeckt wird als dessen Vorgänger gespeichert
  - Vorgängergraph $G_{pred} = (V, E_{pred})$
    - mit $E_{pred} = \{ (v.pred, v) : v \in V$ und $v.pred \neq$ NIL$\}$

- Innerhalb eines Durchlaufs entsteht ein Baum
  - Durchlauf = Von einem Startknoten bis Algorithmus stoppt und gegebenenfalls ein neuer Startknoten gewählt wird

- Durch mehrere Startknoten entsteht ein Wald
  - Wald = disjunkte Vereinigung von Bäumen
  - Farbkodierung im Algorithmus garantiert disjunkte Bäume

Tiefensuche – Pseudocode I

- $DFS(G)$

```
1     foreach u ∈ G.V
2        u.color = WHITE
3        u.predecessor = NIL
4     time = 0
5     foreach u ∈ G.V
6        if u.color == WHITE
7           DFS_VISIT(G, u)
```

Initialisierung

Wenn ein Startknoten abgesucht ist, neuen wählen

Tiefensuche – Pseudocode II

- $DFS\_VISIT(G, u)$

```
1     time = time + 1
2     u.discovered = time
3     u.color = GRAY
4     foreach v ∈ G.Adj[u]
5         if v.color == WHITE
6             v.predecessor = u
7             DFS_VISIT(G, v)
8     u.color = BLACK
9     time = time + 1
10    u.finalized = time
```

u wurde gerade entdeckt

Erkunde Kanten (u, v)

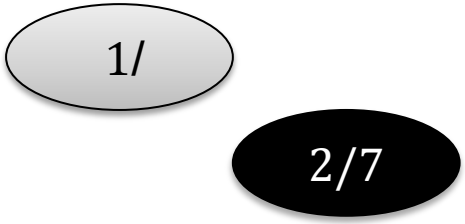
Alle Kanten von u wurden besucht

Kantenklassifikation

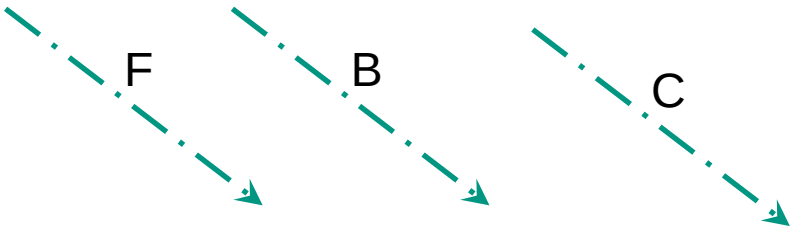
- Baumkante
  - Kante, die in G_{pred} ist
  - (u, v) ist eine Baumkante, wenn v über die Kante (u, v) entdeckt wurde
- Rückwärtskante
  - Kante (u, v), die u mit einem Vorgänger (in G_{pred}) v verbindet
  - Schleifen gelten als Rückwärtskanten
- Vorwärtskante
  - Kante (u, v) die nicht Baumkante ist und u mit einem Nachfolger (in G_{pred}) v verbindet
- Querkante
  - Alle verbleibenden

Legende für Beispiel

- Knoten
  - Erster Wert: $u.discovered$
  - Zweiter Wert: $u.finalized$

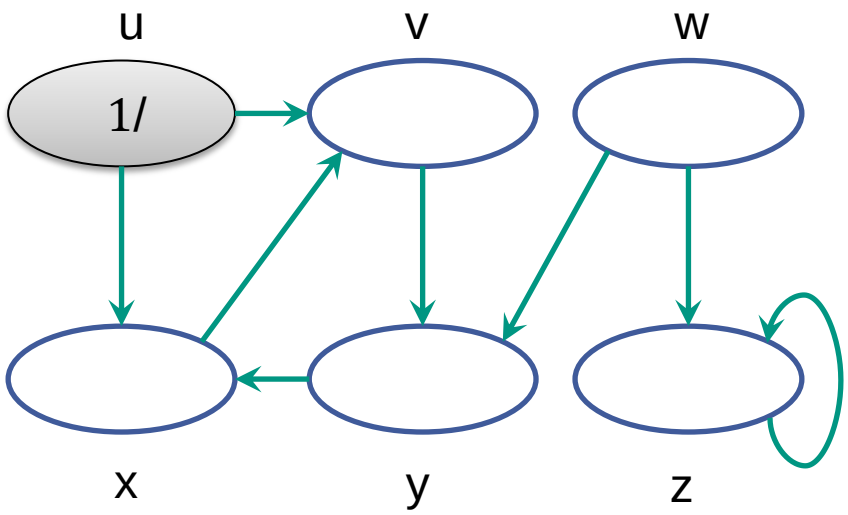


- Kanten
  - Baumkanten werden dargestellt durch blauen Pfeil
  - Vorwärtskanten (F)
  - Rückwärtskanten (B)
  - Querkanten (C)

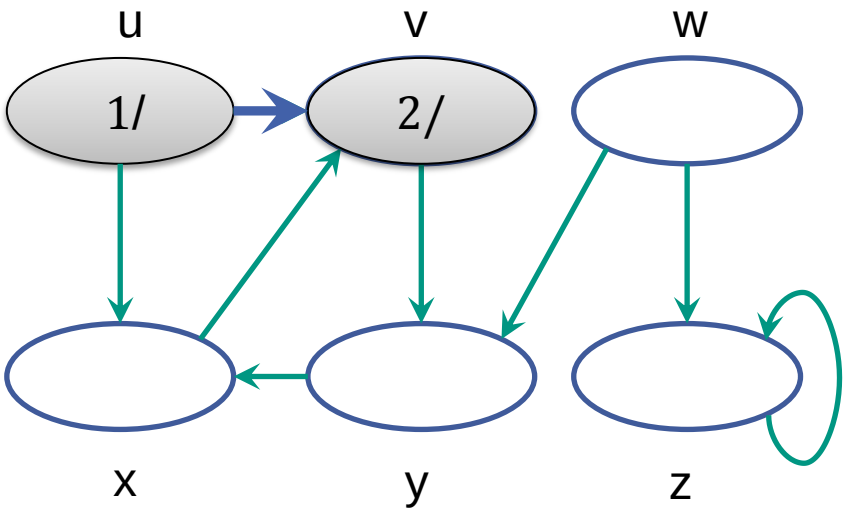


Tiefensuche – Beispiel

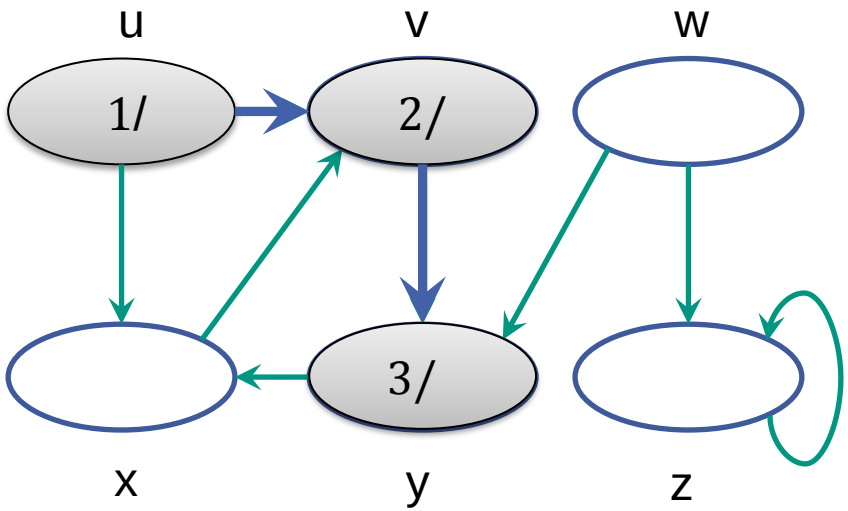
1



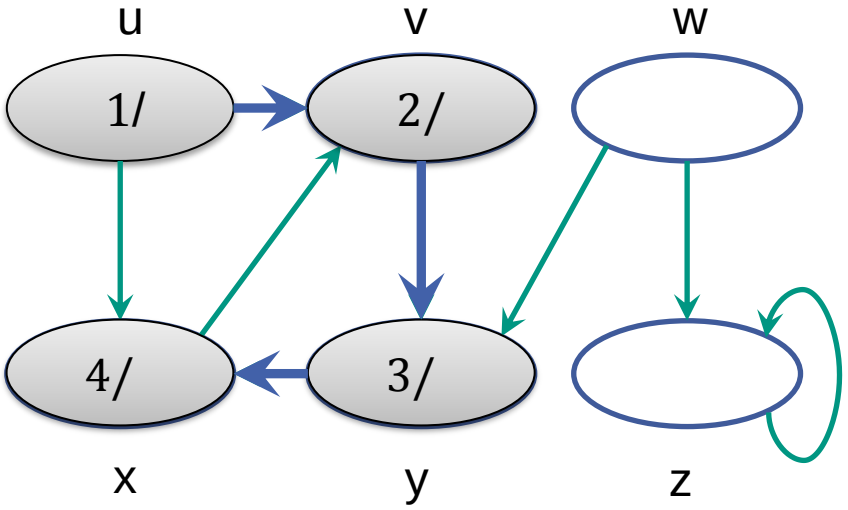
2



3

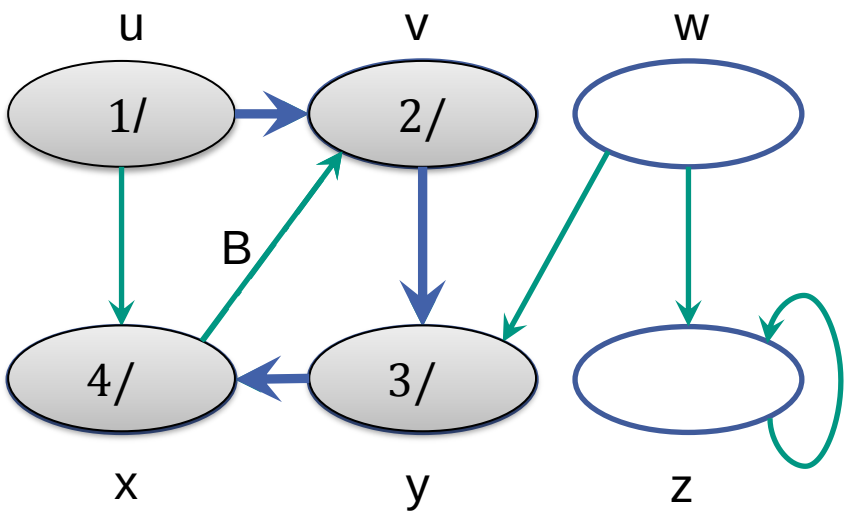


4

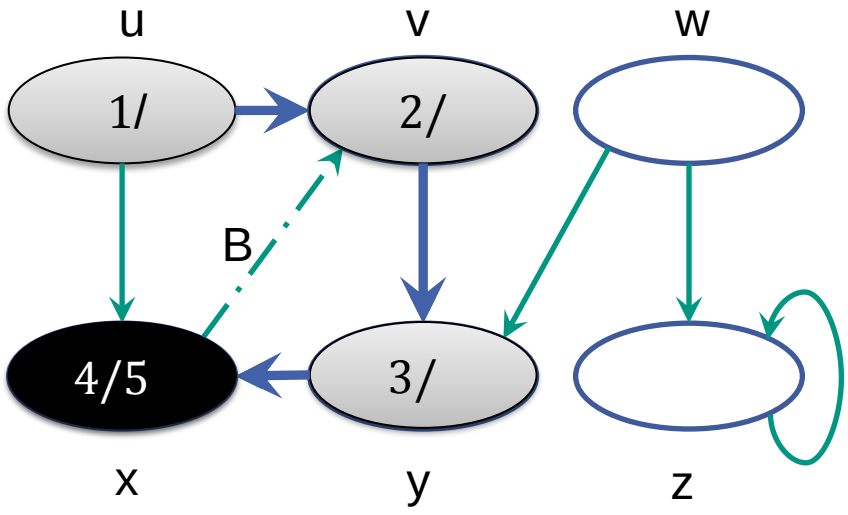


Tiefensuche – Beispiel

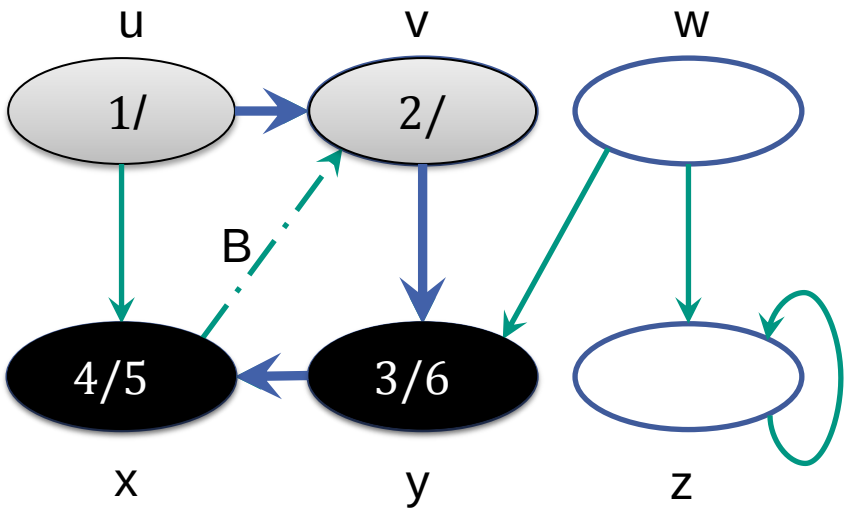
5



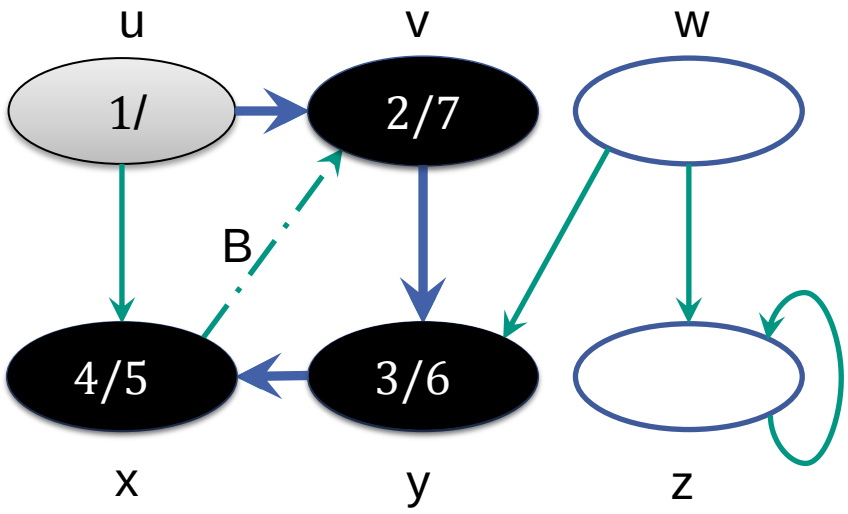
6



7

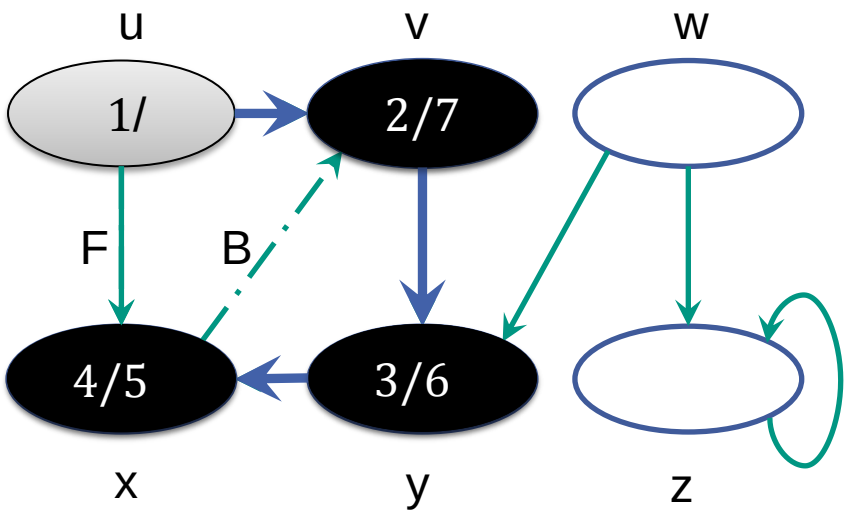


8

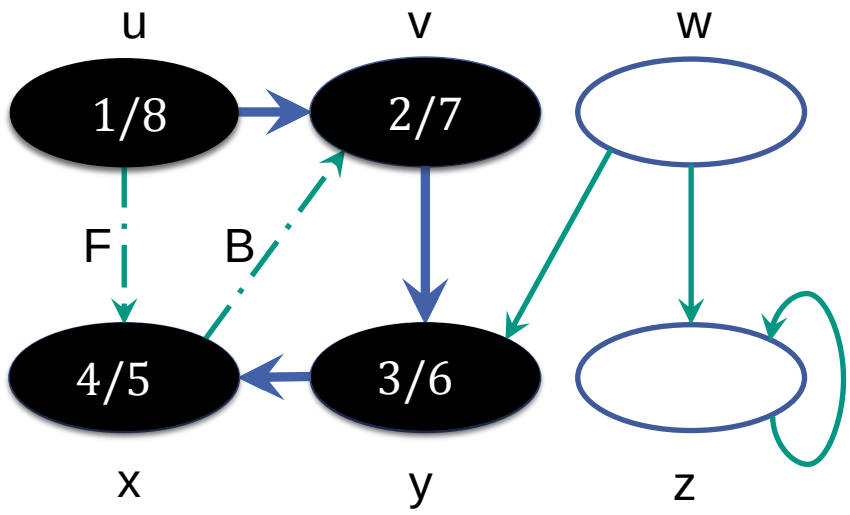


Tiefensuche – Beispiel

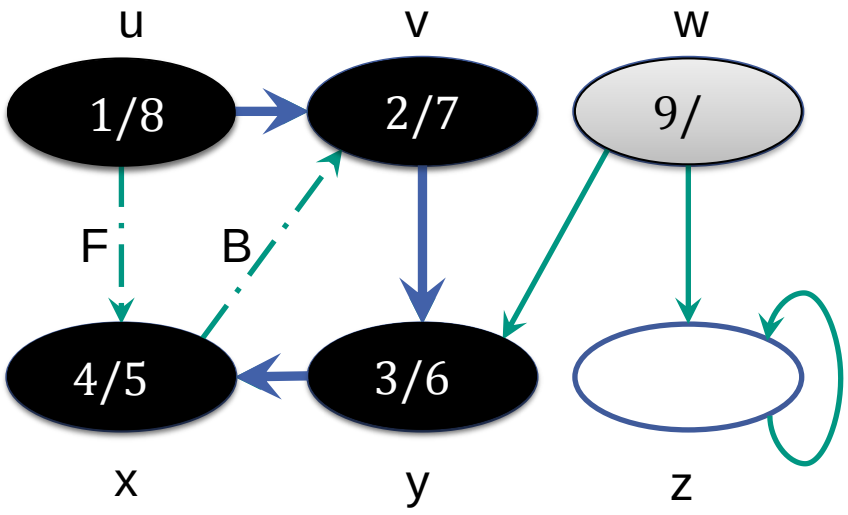
9



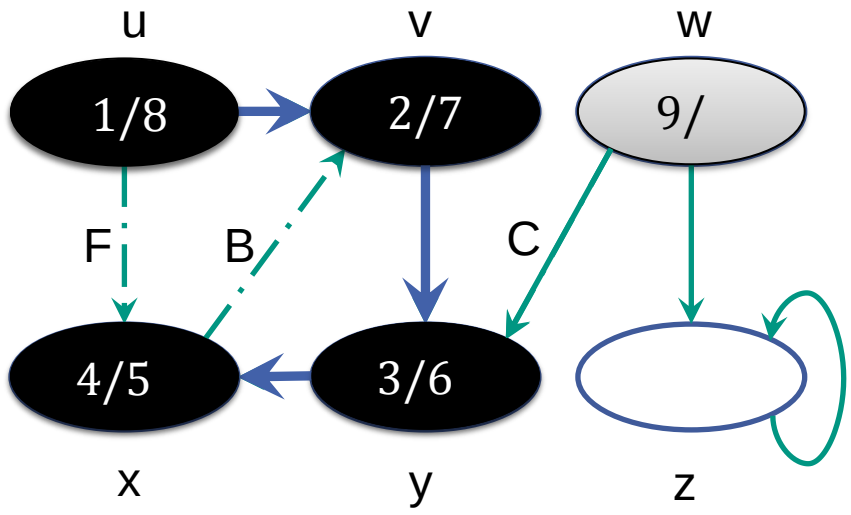
10



11

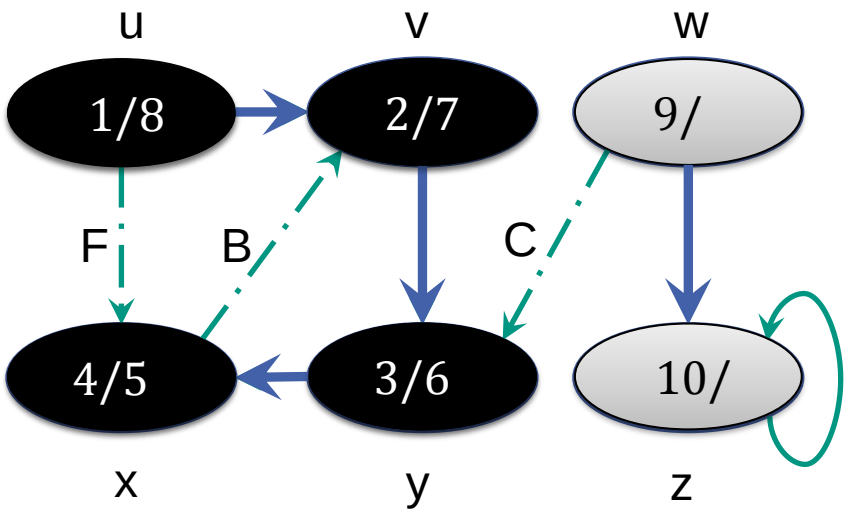


12

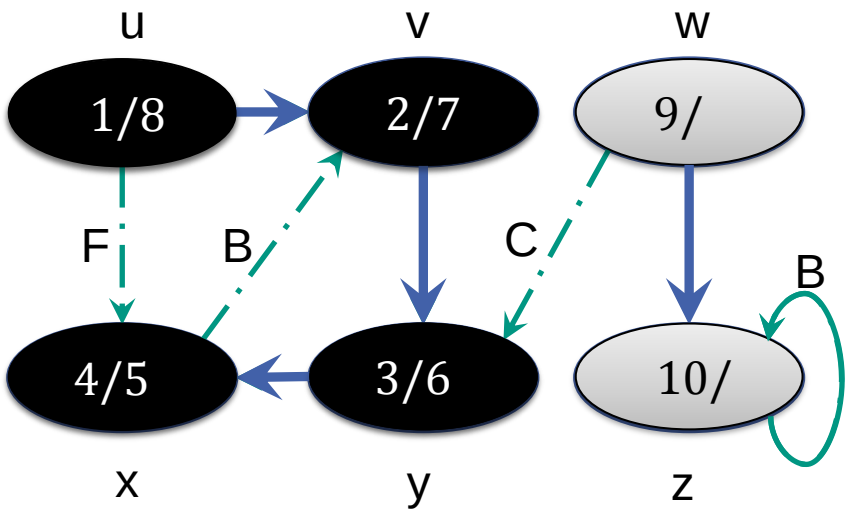


Tiefensuche – Beispiel

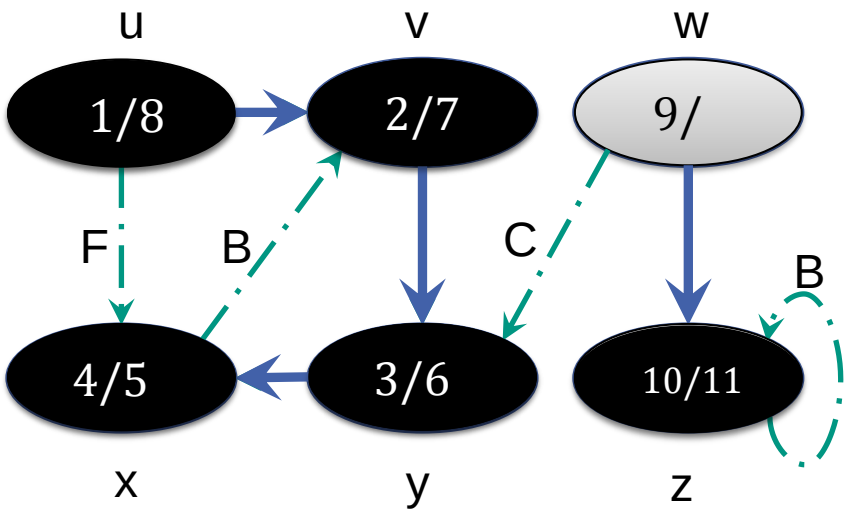
13



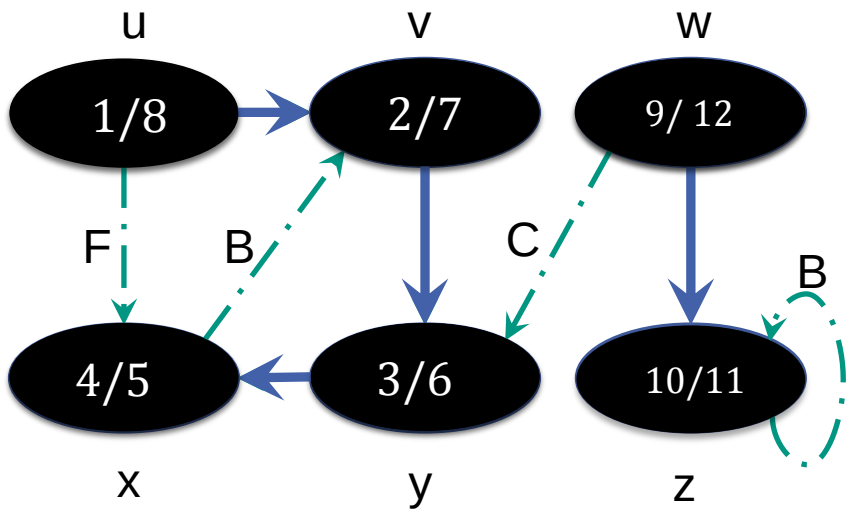
14



15



16



Tiefensuche – Analyse I

- $DFS(G)$

```
1    foreach u ∈ G.V
2        u.color = WHITE
3        u.predecessor = NIL
4    time = 0
5    foreach u ∈ G.V
6        if u.color == WHITE
7            DFS_VISIT(G, u)
```

Initialisierung in $\Theta(|V|)$

$\Theta(|V|)$

DFS_VISIT wird nur für weiße Knoten aufgerufen

Tiefensuche – Analyse II

- $DFS\_VISIT(G, u)$

```
1     time = time + 1
2     u.discovered = time
3     u.color = GRAY
4     foreach v ∈ G.Adj[u]
5         if v.color == WHITE
6             v.predecessor = u
7             DFS_VISIT(G, v)
8     u.color = BLACK
9     time = time + 1
10    u.finalized = time
```

Knoten wird sofort grau markiert → DFS_VISIT wird genau einmal pro Knoten aufgerufen

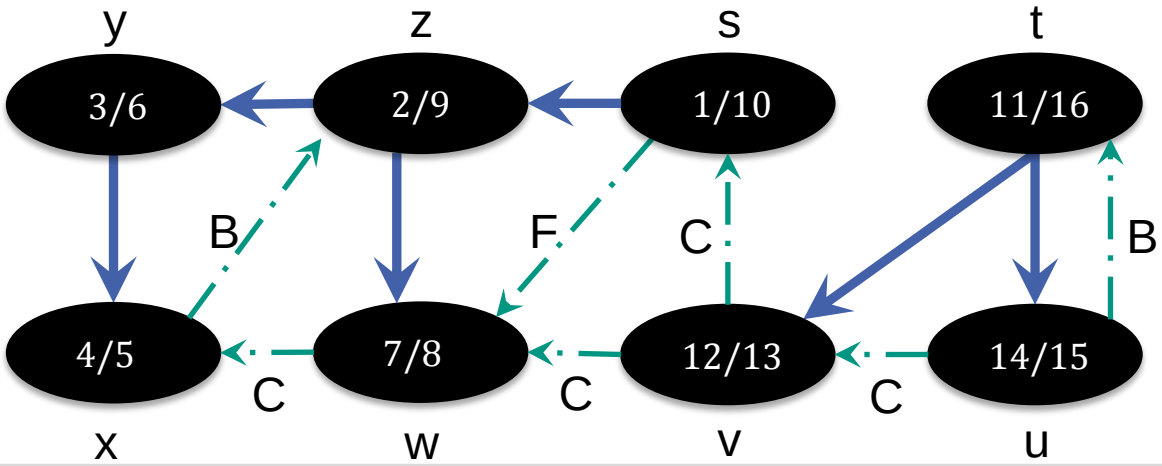
Läuft genau |Adj[u]|-mal

Tiefensuche – Analyse III

- Initialisierung in $\Theta(|V|)$
- Mit Aufwand $\Theta(|V|)$ wird DFS_VISIT genau einmal pro Knoten aufgerufen
- Schleife in DFS_VISIT läuft genau $|\mathrm{Adj[u]}|$-mal für jeden Knoten u
  - Summe der Länge der Adjazenzlisten $\sum_{u \in V} |Adj[u]| = \Theta(|E|)$
  - → Aufwand liegt in $\Theta(|E|)$
- Gesamtaufwand in $\Theta(|V| + |\mathrm{E}|)$

Klammern-Theorem

- Entdecken- und Abschließen-Zeitstempel haben Klammerstruktur
  - Entdecken eines Knoten u wird mit „$(u$“ repräsentiert
  - Abschließen eines Knoten u wird mit „$u)$“ repräsentiert
- Dabei entstehender Ausdruck hat korrekte Klammerschachtelung
- Beispiel am Ergebnis einer abgeschlossenen Tiefensuche
  - Startknoten ist Knoten s

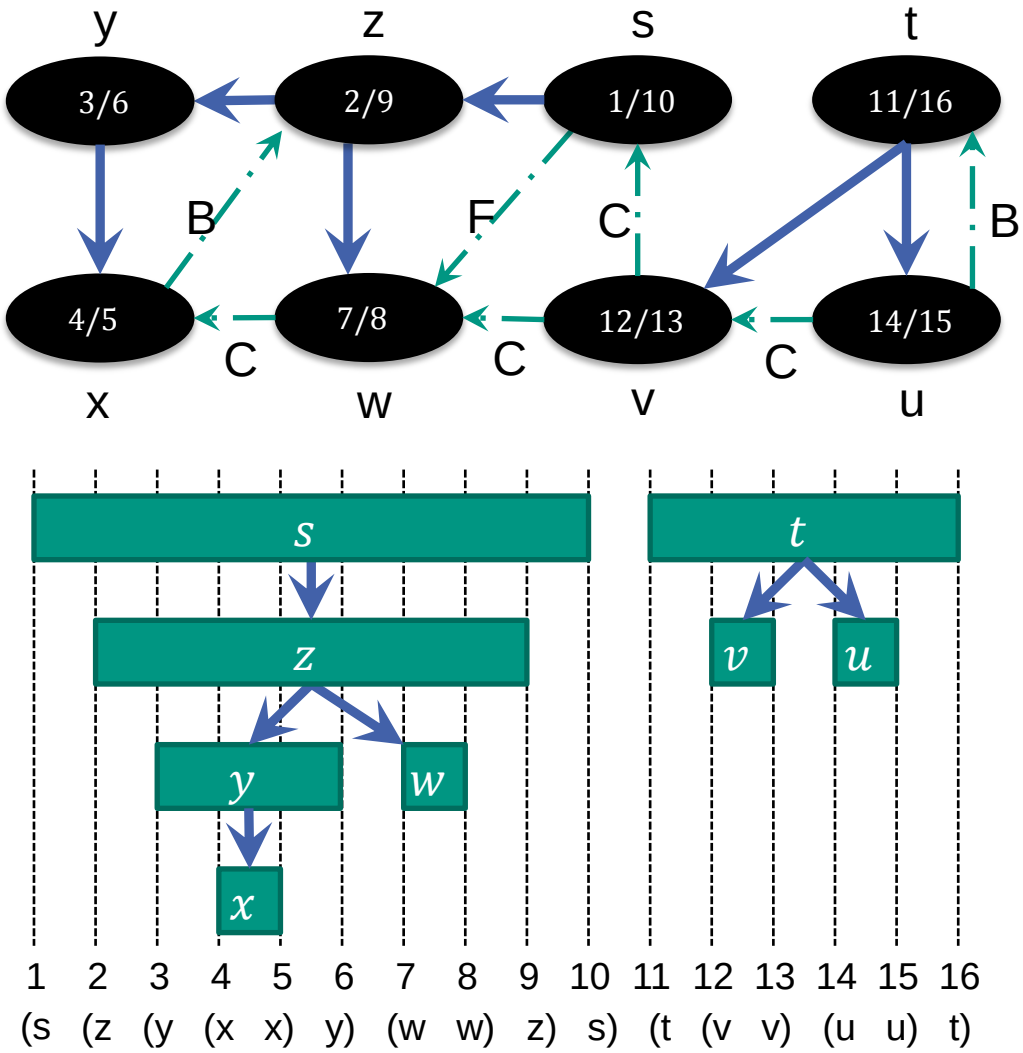


Klammern-Theorem

- Startknoten s wurde von Zeitstempel 1 bis 10 besucht
- Von s aus wurde z entdeckt und von Zeitstempel 2 bis 9 besucht
- Von z aus wurde y entdeckt und von 3 bis 6 besucht
- Von y aus wurde x entdeckt und von 4 bis 5 besucht
- Von z aus wurde auch w entdeckt und von 7 bis 8 besucht

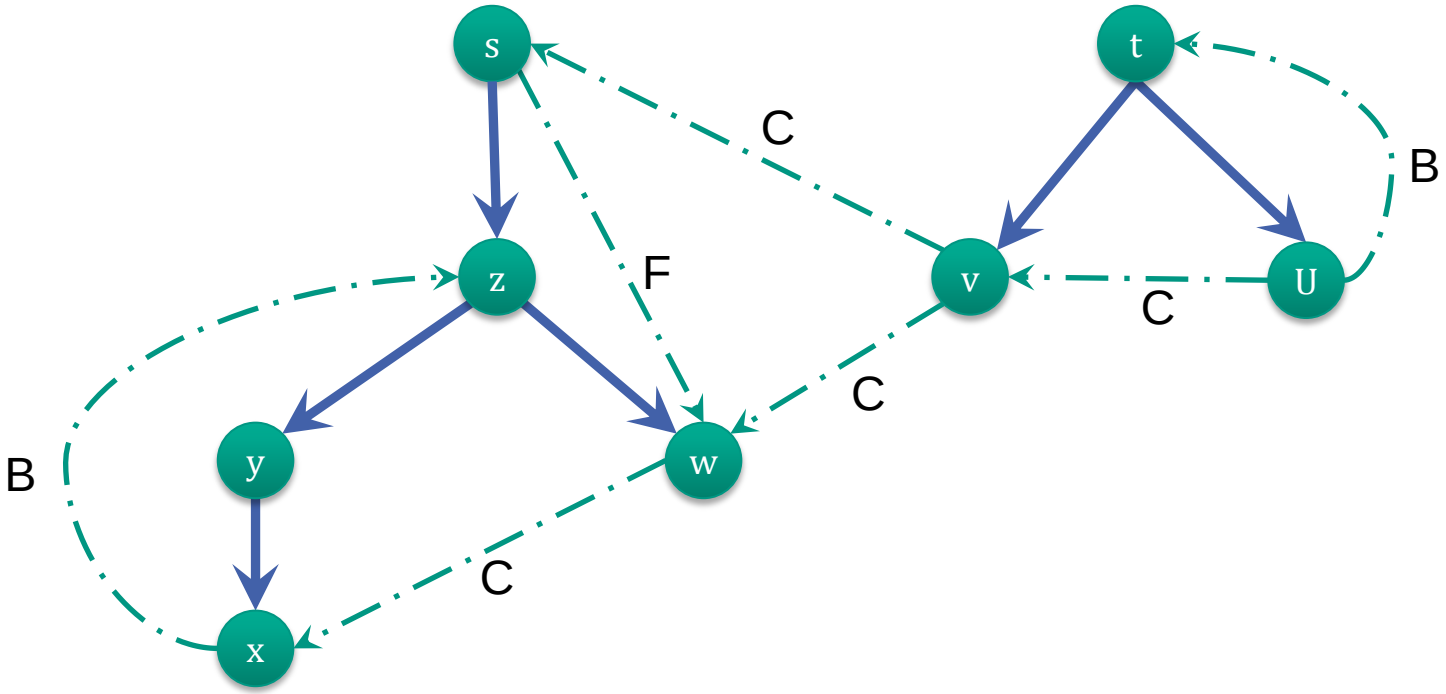
- Neuer Startknoten t, besucht von 11 bis 16
- Von t aus wurde v entdeckt, besucht von 12 bis 13
- Von t aus wurde auch u entdeckt, besucht von 14 bis 15

- Klammerstruktur ist korrekt geschachtelt



Klammern-Theorem

- Zur Anschauung andere Darstellung
  - Alle Baum- und Vorwärtskanten zeigen nach unten
  - Alle Rückwärtskanten nach oben auf einen Vorgänger



Klammern-Theorem

- Für jede Tiefensuche eines Graphen $G = (V, E)$ gilt für jedes Knotenpaar u, v genau eine der drei folgenden Bedingungen
  - Die Intervalle $[u.discovered, u.finalized]$ und $[v.discovered, v.finalized]$ sind disjunkt und weder u noch v ist Nachfolger des anderen im Depth-First-Forest
  - Das Intervall $[u.discovered, u.finalized]$ ist komplett eingeschlossen in $[v.discovered, v.finalized]$ und u ist Nachfolger von v in einem Depth-First-Baum
  - Das Intervall $[v.discovered, v.finalized]$ ist komplett eingeschlossen in $[u.discovered, u.finalized]$ und v ist Nachfolger von u in einem Depth-First-Baum

Klammern-Theorem Beweis

- Sei $u.discovered < v.discovered$
  - Falls $v.discovered < u.finalized$ gilt
    - v wurde entdeckt während u noch grau markiert war → v ist Nachfolger von u
    - Da v nach u entdeckt wurde, werden alle ausgehenden Kanten von v abgearbeitet und v als abgeschlossen markiert, bevor u abgeschlossen werden kann
    - → Das Intervall $[v.discovered, v.finalized]$ ist komplett eingeschlossen in $[u.discovered, u.finalized]$
  - Falls $u.finalized < v.discovered$ gilt
    - $u.discovered < u.finalized < v.discovered < v.finalized$, da $\forall\, n \in V: n.discovered < n.finalized$
    - → Die Intervalle $[u.discovered, u.finalized]$ und $[v.discovered, v.finalized]$ sind disjunkt
    - → Keiner der Knoten wurde entdeckt während der andere grau markiert war
    - → Keiner der Knoten ist Nachfolger des anderen
- Der Fall $v.discovered < u.discovered$ ist analog mit vertauschtem u, v

q.e.d

Tiefensuche in ungerichteten Graphen

- Annahme: Bei einer Tiefensuche auf einem ungerichteten Graphen G ist jede Kante von G entweder eine Baum- oder eine Rückwärtskante

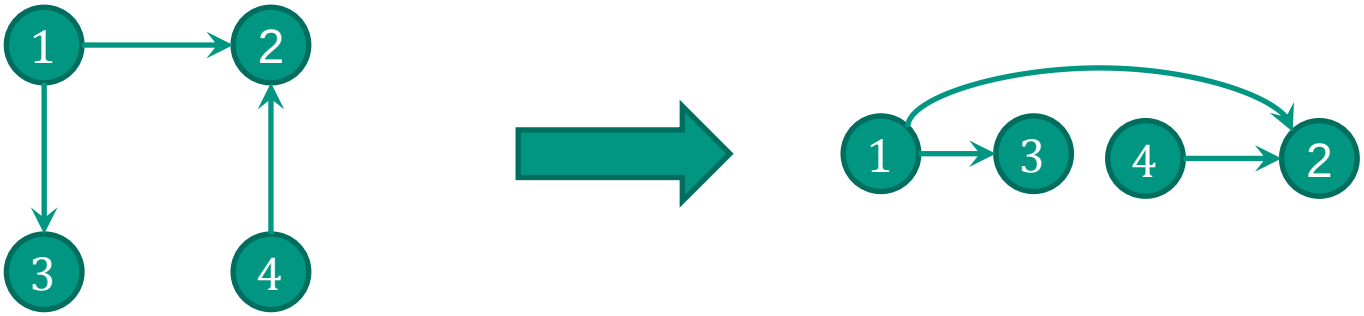
- Beweis: Sei (u, v) eine beliebige Kante in G und ohne Beschränkung der Allgemeinheit $u.discovered < v.discovered$
  - Da v in der Adjazenzliste von u ist, muss die Suche v entdecken und abschließen, bevor u abgeschlossen werden kann
  - Falls (u, v) zum ersten mal von u aus erkundet wird
    - Ist v bis dahin noch unerkundet, also weiß markiert
    - → (u, v) ist eine Baumkante
  - Falls (u, v) zum ersten mal von v aus erkundet wird
    - Ist u grau markiert da $u.discovered < v.discovered$
    - → (u, v) ist eine Rückwärtskante

q.e.d

8.3 Topologisches Sortieren

- Topologische Sortierung eines Graphen $G = (V, E)$
  - Ordnung aller Knoten in G, so dass
  - $u < v$ falls $(u, v) \in E$
  - Falls G Zyklen enthält ist keine Sortierung möglich

- Anschaulich werden Knoten auf einer Linie angeordnet
  - Alle Kanten gehen von links nach rechts



Topologisches Sortieren

- DAGs (gerichtete, zyklenfreie Graphen) werden häufig zur Präzedenzmodellierung benutzt
  - Abhängigkeiten zwischen Vorgängen werden als gerichtete Kanten dargestellt
  - (u, v) bedeutet, dass v von u abhängt
  - u muss abgearbeitet sein bevor v begonnen wird

Tiefensuche in DAGs

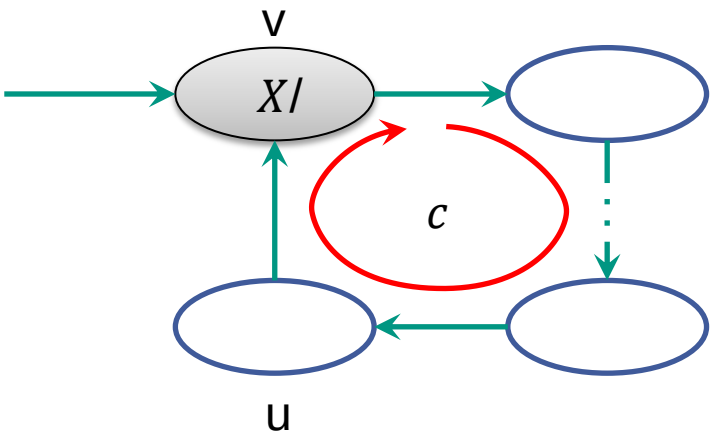
- Annahme: Ein gerichteter Graph ist zyklenfrei, genau dann wenn eine Tiefensuche auf G keine Rückwärtskanten liefert

- Beweis
  - ⇒:
    - Sei (u, v) eine von der Tiefensuche entdeckte Rückwärtskante
    - → v ist Vorgänger von u im Depth-First-Forest
    - → G enthält einen Pfad von v nach u und die Rückwärtskante (u, v) schließt einen Zyklus

Tiefensuche in DAGs

- ⇐:
  - Sei c ein Zyklus in G und v der Knoten in c, der als erster entdeckt wird
  - Weiterhin sei (u, v) die Kante innerhalb von c, die den Zyklus zu v schließt
  - Zum Zeitpunkt $v.discovered$ bilden die Knoten von c einen Pfad von weißen Knoten von v nach u
  - Dadurch wird u Nachfolger von v im Depth-First-Forest
    Beweis siehe *Theorem 22.9 [Corm10]*
  - → $(u. v)$ ist eine Rückwärtskante

q.e.d



Topologisches Sortieren

- Einfacher Algorithmus zur Implementierung einer topologischen Sortierung mittels Tiefensuche

- $TOPO\_SORT(G)$
  1 $DFS(G)$ um $v.finalized$ für alle $v \in V$ zu berechnen
  2 Immer wenn $finalized$ gesetzt wird
  3 Füge Knoten an den Anfang einer Liste an
  4 Ausgabe: Dabei entstandene Knotenliste

- Aufwandsabschätzung
  - Tiefensuche benötigt $\Theta(|V| + |E|)$
  - Einfügen jedes Knotens an den Anfang einer verketteten Liste in je $O(1)$
  - → Gesamtaufwand in $\Theta(|V| + |E|)$

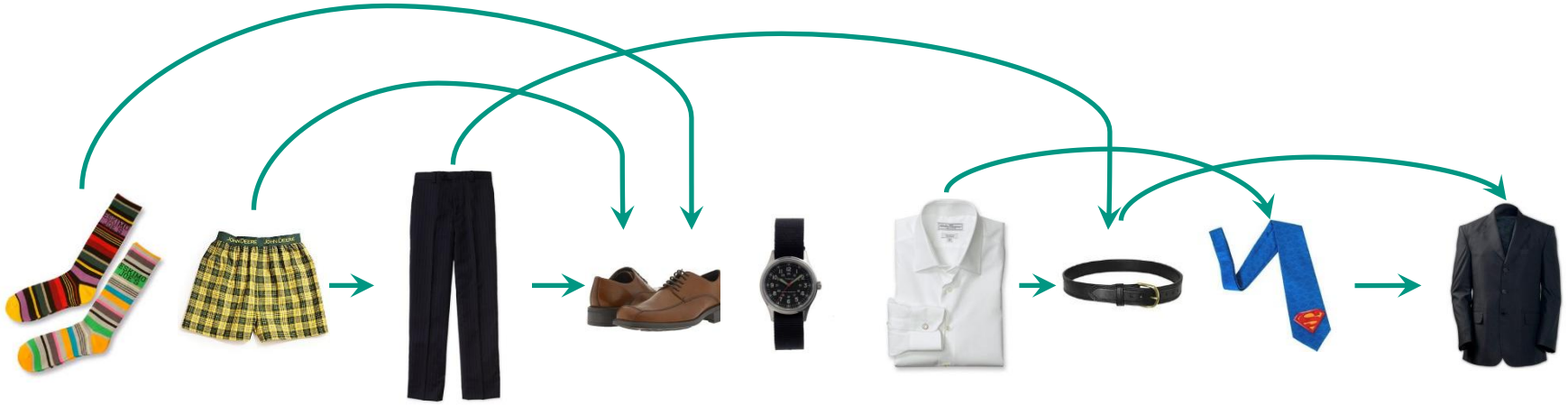
Beispiel

- Abhängigkeiten beim Ankleidevorgang

Entstehende Liste:

Beispiel

- Abhängigkeiten beim Ankleidevorgang



- Von links nach rechts mit absteigendem $finalized$ Wert

Beweis der Korrektheit von TOPO_SORT

- Annahme: $TOPO\_SORT$ sortiert den übergebenen gerichteten, zyklenfreien Graph topologisch
- Beweis
  - Tiefensuche sei auf den gerichteten, zyklenfreien Graph $G = (V, E)$ angewendet worden um die Abschlusszeiten der Knoten zu ermitteln
  - Es genügt zu zeigen, dass $\forall\, u, v \in V$: falls $(u, v) \in E$ dann muss $v.finalized < u.finalized$ sein
  - Sei (u, v) eine beliebige Kante in G, die von der Tiefensuche erkundet wird
    - → v kann nicht grau markiert sein, da v sonst Vorgänger von u wäre und (u, v) damit eine Rückwärtskante (Widerspruch zu „keine Rückwärtskanten in Tiefensuchen von DAGs“)
    - → v muss entweder weiß oder schwarz markiert sein
      - Weiß: v wird Nachfolger von u und damit gilt $v.finalized < u.finalized$
      - Schwarz: $v.finalized$ wurde bereits gesetzt und da wir von u kommen, wird $u.finalized$ später größer als $v.finalized$ gesetzt
  - → Für jede Kante (u, v) im Graphen gilt $v.finalized < u.finalized$

q.e.d

8.4 Starke Zusammenhangskomponenten

- Die Relation $\overset{*}{\leftrightarrow}$ auf Knoten eines gerichteten Graphen $G = (V, E)$ sei definiert durch
  - $u \overset{*}{\leftrightarrow} v$ falls ein Pfad von u nach v und ein Pfad von v nach u existiert
  - $\overset{*}{\leftrightarrow}$ stellt eine Äquivalenzrealtion dar
  - Die Äquivalenzklasse von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponente

- Starke Zusammenhangskomponenten werden häufig für große, gerichtete Graphen bei der Vorverarbeitung genutzt
- Vorgehen
  - Starke Zusammenhangskomponenten identifizieren
  - Algorithmen separat auf den Komponenten ausführen
  - Lösungen passend zu den Verbindungen zwischen den Komponenten zusammenführen

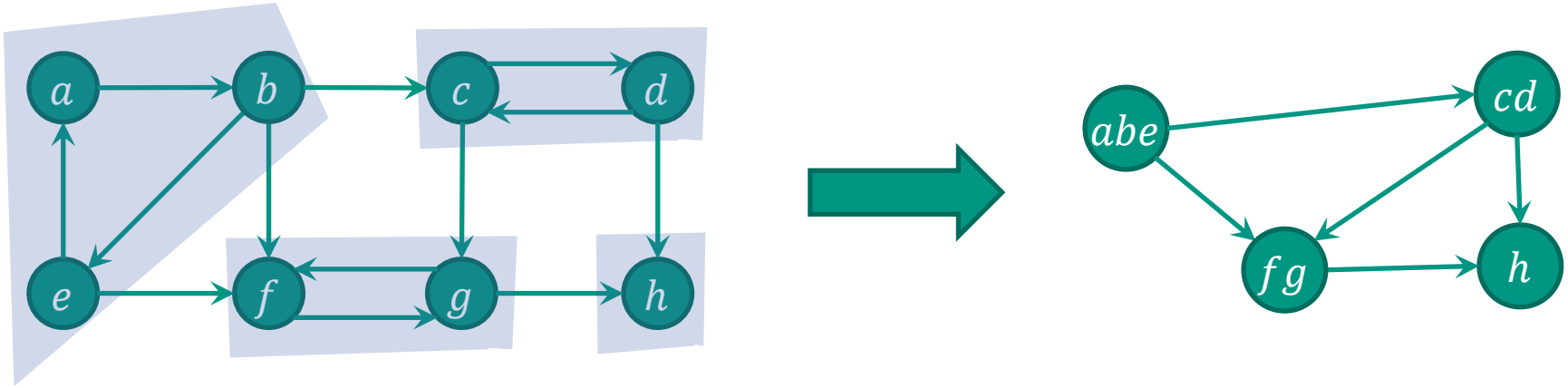
Komponentengraph

- Komponentengraph $G^{SCC} = (V^{SCC}, E^{SCC})$
  - Sei $G = (V, E)$ ein Graph mit stark zusammenhängenden Komponenten $C_1, C_2, \ldots, C_k$.
  - $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$ enthält einen Knoten v_i für jede stark zusammenhängende Komponente C_i
  - Es existiert eine Kante $(v_i, v_j) \in E^{SCC}$ falls G eine Kante (x, y) mit $x \in C_i$ und $y \in C_j$ enthält

- Anschaulich
  - Jede starke Zusammenhangskomponente wird durch einen Knoten repräsentiert
  - Jede Kante, die zwischen starken Zusammenhangskomponenten war, ist nun zwischen den jeweiligen Repräsentanten

- Mittels Tiefensuche kann dies in Linearzeit durchgeführt werden → Algorithmen II

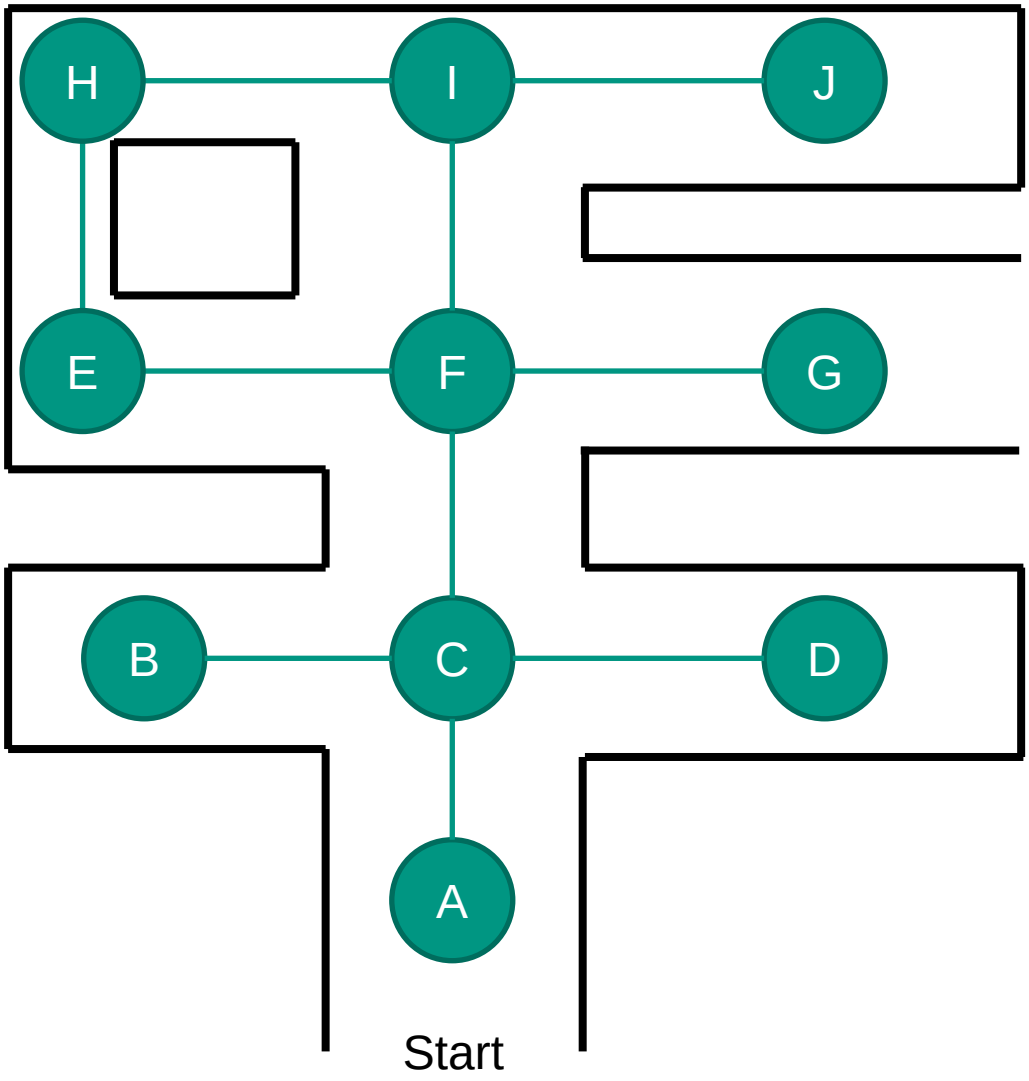
Beispiel



Rückblick

- Tiefensuche steigt rekursiv in einen Graphen ab
- Gesamtaufwand liegt in in $\Theta(|V| + |\mathrm{E}|)$
- Entdecken- und Abschließen-Zeitstempel der Tiefensuche haben Klammerstruktur mit korrekter Schachtelung
- Beispielhafte Anwendungen der Tiefensuche
  - Topologisches Sortieren
  - Starke Zusammenhangskomponenten

8.5 Breitensuche – Motivation



- Tiefensuche kann zur Erforschung des ganzen Labyrinths
- obwohl das
- e am
- ten ist

Breitensuche

- Breitensuche (BFS, **B**readth **F**irst **S**earch)
  - Einer der einfachsten Algorithmen zur Suche in Graphen
  - Grundlegend für viele wichtige Graphenalgorithmen
    - Dijkstra-Algorithmus für kürzeste Wege
      - Siehe Kapitel 9
    - Algorithmus von Prim zur Berechnung von minimalen Spannbäumen
      - Siehe Kapitel 10

- Mit $G = (V, E)$ ein Graph und $s \in V$ ein Knoten (im Folgenden auch Startknoten genannt)
  - BFS durchsucht systematisch die Kanten von G um jeden Knoten in G zu entdecken, der von s aus erreichbar ist
  - BFS berechnet die Distanz (kleinste Anzahl Kanten) von s zu jedem erreichbaren anderen Knoten in G

Breitensuche

- Unterschiedliche Betrachtung von Breiten- und Tiefensuche
  - Tiefensuche ohne feste Startknoten betrachtet
  - Breitensuche nun mit festgelegtem Startknoten

- Grund: Übliche Anwendung der Algorithmen
  - Tiefensuche meist Bestandteil eines anderen Algorithmus
    → Allgemeine Informationen über den gesamten Graph von Interesse
  - Breitensuche häufig für kürzeste Pfade Probleme
    → Startknoten meist vorgegeben

Breitensuche - Vorgängerbaum

- Nebenprodukt der Breitensuche ist der Breadth-First-Tree

- BFS speichert für jeden Knoten von welchem Knoten aus er entdeckt wurde, also seinen Vorgänger

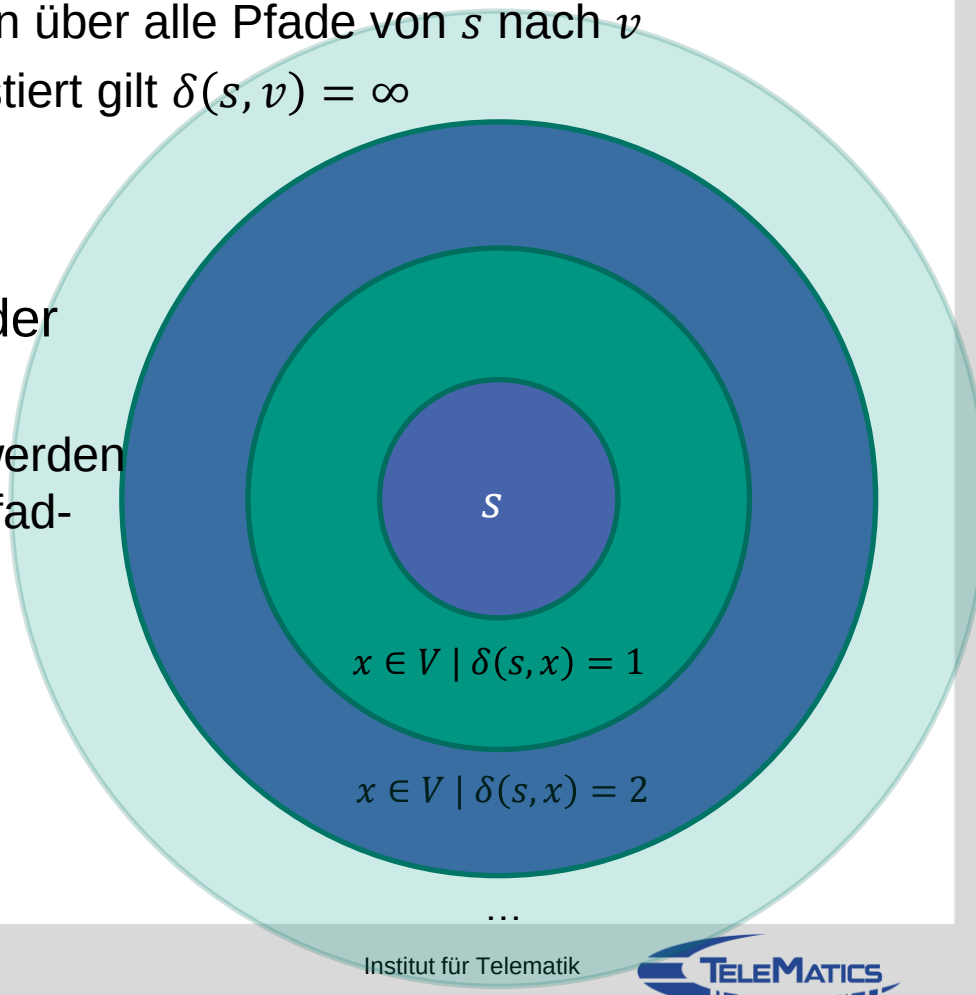
- Vorgängergraph $G_{pred} = (V_{pred}, E_{pred})$
  - Mit $V_{pred} = \{v \in V : v.pred \neq NIL\} \cup \{s\}$
  - und $E_{pred} = \{ (v.pred, v) : v \in V_{pred} - \{s\} \}$

- Vorgängergraph einer Breitensuche ist ein Baum mit Wurzel s, der alle Knoten in G enthält, die vom Startknoten aus erreichbar sind
- Für jeden von s aus erreichbaren Knoten v entspricht der Pfad von s nach v im Breadth-First-Tree dem kürzesten Pfad in G
  - Beweis folgt

Kürzeste Pfade

- $\delta(s, v)$ ist die kürzeste Pfad-Distanz von s nach v
  - Pfad-Distanz = Distanz, für die mindestens ein Pfad dieser Länge existiert
  - $\delta(s, v) =$ Minimale Anzahl Kanten über alle Pfade von s nach v
  - Falls kein Pfad von s nach v existiert gilt $\delta(s, v) = \infty$

- BFS entdeckt Knoten mit steigender Pfad-Distanz vom Startknoten s
  - Alle Knoten mit Pfad-Distanz k werden entdeckt bevor Knoten mit der Pfad-Distanz $k + 1$ entdeckt werden



Breitensuche – Pseudocode

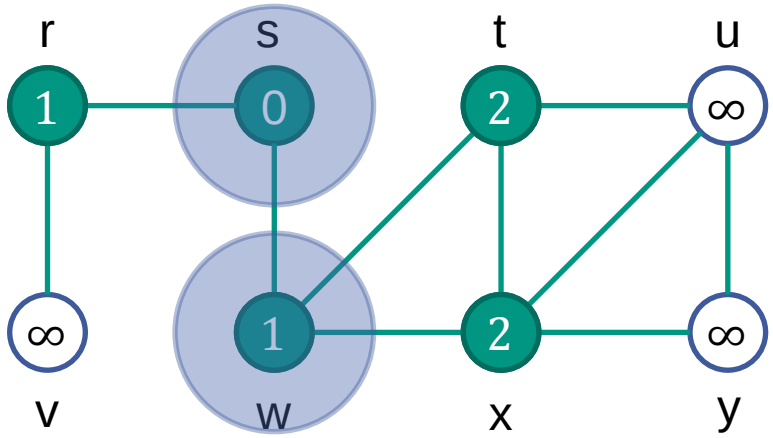
- BFS(G, s)

```
1     foreach u ∈ G.V
2             u.visited = false
3             u.distance = ∞
4             u.predecessor = NIL
5     s.distance = 0
6     Q = ∅
7     ENQUEUE(Q, s)
8     while Q ≠ ∅
9             u = DEQUEUE(Q)
10            foreach v ∈ G.Adj[u]
11                    if v.visited == false
12                            v.visited = true
13                            v.distance = u.distance + 1
14                            v.predecessor = u
15                            ENQUEUE(Q, v)
```

Warteschlangenoperationen

Durchsuchen der Adjazenzliste von u

Breitensuche – Beispiel



Q s

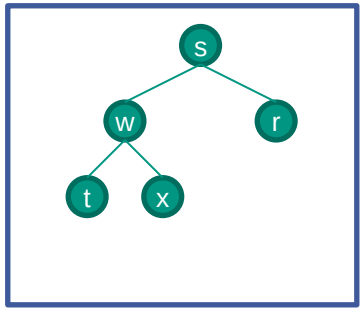
s ← Q w r

w ← Q r t x

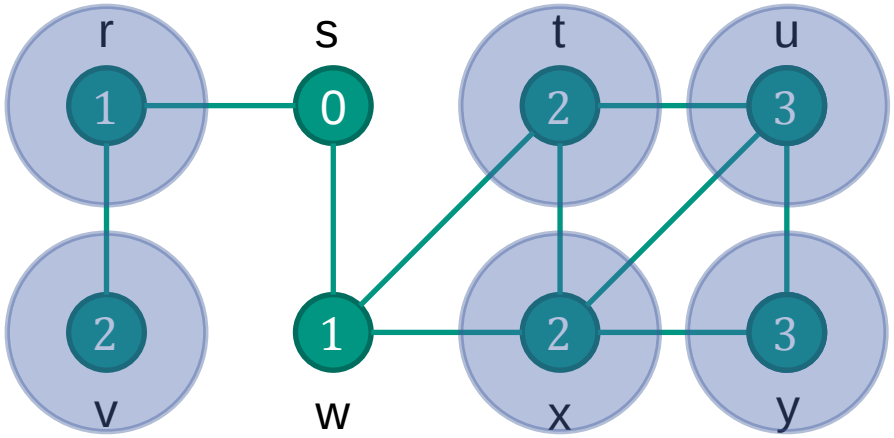
$visited = \text{false},$
$distance = \infty$ ∞

$visited = \text{true},$
$distance = 0$ 0

Breadth-First-Tree



Breitensuche – Beispiel

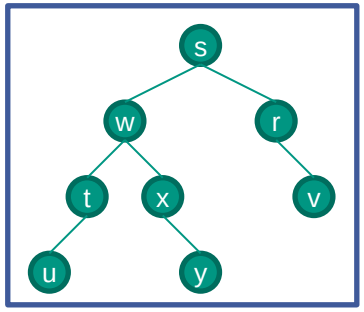


	Q	s				
s ←	Q	w	r			
w ←	Q	r	t	x		
r ←	Q	t	x	v		
t ←	Q	x	v	u		
x ←	Q	v	u	y		
v ←	Q	u	y			
u ←	Q	y				
y ←	Q					

$visited = \text{false},$
$distance = \infty$ ∞

$visited = \text{true},$
$distance = 0$ 0

Breadth-First-Tree



Breitensuche – Analyse I

- $BFS(G, s)$

```
1  foreach u ∈ G.V
2     u.visited = false
3     u.distance = ∞
4     u.predecessor = NIL
5  s.distance = 0
6  Q = ∅
7  ENQUEUE(Q, s)
8  while Q ≠ ∅
9     u = DEQUEUE(Q)
10    foreach v ∈ G.Adj[u]
11       if v.visited == false
12          v.visited = true
13          v.distance = u.distance + 1
14          v.predecessor = u
15          ENQUEUE(Q, v)
```

Einziger Aufruf von visited = false

Initialisierungsaufwand in $O(|V|)$

Jeweils einmal pro Knoten $ENQUEUE$ und $DEQUEUE$ jeweils in $O(1)$
→ Gesamte Warteschlangenoperationen in $O(|V|)$

Pro Knoten genau einmal

Breitensuche – Analyse II

- $BFS(G, s)$

```
1  foreach u ∈ G.V
2     u.visited = false
3     u.distance = ∞
4     u.predecessor = NIL
5  s.distance = 0
6  Q = ∅
7  ENQUEUE(Q, s)
8  while Q ≠ ∅
9     u = DEQUEUE(Q)
10    foreach v ∈ G.Adj[u]
11       if v.visited == false
12          v.visited = true
13          v.distance = u.distance + 1
14          v.predecessor = u
15          ENQUEUE(Q, v)
```

Jede Adjazenzliste wird maximal einmal durchsucht

Summe der Länge aller Adjazenzlisten insgesamt in $\Theta(|E|)$
→ Gesamtaufwand für Durchsuchen von Adjazenzlisten in $O(|E|)$

Breitensuche – Zusammenfassung Analyse

- Gesamtaufwand Breitensuche
  - Initialisierung in $O(|V|)$
  - Warteschlangenoperationen in $O(|V|)$
  - Durchsuchen der Adjazenzlisten in $O(|E|)$

- → Gesamtaufwand in $O(|V| + |E|)$

Beweis I

- Mittels der folgenden Beweise wird gezeigt, dass Breitensuche die kürzesten Pfade zu allen Knoten vom Startknoten aus liefert

- Annahme: Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und $s \in V$ ein beliebiger Knoten.
  Dann gilt $\forall\ (u, v) \in E: \delta(s, v) \leq \delta(s, u) + 1$

- Beweis I
  - Falls u von s aus erreichbar ist, ist auch v von s aus erreichbar
    - → Der kürzeste Pfad von s nach v kann nicht länger sein, als der kürzeste Pfad von s nach u gefolgt von der Kante (u, v)
  - Falls u nicht von s aus erreichbar ist, ist $\delta(s, u) = \infty$
    - → Die Ungleichung gilt

q.e.d

Beweis II

- Annahme: Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph auf dem eine Breitensuche mit Startknoten $s \in V$ durchgeführt wurde
  - Nach Abschluss der Breitensuche gilt $\forall v \in V: v.distance \geq \delta(s, v)$

- Beweis II: durch Induktion über die Einreihungen in die Warteschlange
  - Induktionsanfang direkt nach Einreihung von s in Zeile 7 von BFS
    - Annahme gilt, da $s.distance = 0 = \delta(s, s)$ und
    - $\forall v \in V - \{s\}$: $v.distance = \infty \geq \delta(s, v)$
  - Induktionsschritt
    - Sei v ein Knoten mit $v.visited = false$ der von Knoten u aus entdeckt wird
    - Die Annahme fordert, dass $u.distance \geq \delta(s, u)$
    - Aus Zeile 13 und dem vorherigen Beweis I folgt
      - $v.distance = u.distance + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$
  - Da v nur einmal in die Warteschlange eingereiht wird und sich $v.distance$ danach nicht mehr ändert gilt die Annahme am Ende der Breitensuche

q.e.d

Beweis III

- Um $v.distance = \delta(s, v)$ zu zeigen, muss die Warteschlange Q näher betrachtet werden. Im Folgenden wird bewiesen, dass Q maximal zwei verschiedene $distance$-Werte enthält

- Annahme: Sei $Q = \langle v_1, v_2, \ldots, v_r \rangle$ die Warteschlange während einer Breitensuche auf dem Graphen $G = (V, E)$, wobei v_1 das vorderste und v_r das letzte Element darstellt.
  - Dann gilt $v_r.distance \leq v_1.distance + 1$ und
  - $v_i.distance \leq v_{i+1}.distance$ für $i = 1,2, \ldots, r-1$

- Beweis III: durch Induktion über die Einreihungen in die Warteschlange
  - Bei Einreihung von s gilt die Annahme
  - Für Induktionsschritt muss bewiesen werden, dass die Annahme sowohl bei Einreihung, als auch bei Entfernen eines Knotens gilt.

Beweis III

- Entfernen von v_1 aus Q
  - Falls Q nun leer ist, gilt die Annahme
  - Sonst wird v_2 neues vorderstes Element
  - Induktionsannahme liefert
    - $v_1.distance \leq v_2.distance$ und
    - $v_r.distance \leq v_1.distance + 1 \leq v_2.distance + 1$
  - Die übrigen Ungleichungen sind von der Operation nicht betroffen

  - → Annahme gilt mit v_2 als neues vorderstes Element

$$Q = \langle v_1, v_2, \ldots, v_r \rangle$$

Beweis III

$$Q = \langle v_1, v_2, \ldots, v_r \rangle$$

- Einfügen eines neuen Knotens v_{r+1}
  - Zu diesem Zeitpunkt wurde ein Knoten u bereits aus Q entfernt
    - Dessen Adjazenzliste wird gerade durchsucht
  - Es gilt für das neue vorderste Element $v_1.distance \geq u.distance$
  - → $v_{r+1}.distance = u.distance + 1 \leq v_1.distance + 1$
  - Durch Induktionsannahme gilt $v_r.distance \leq u.distance + 1 = v_{r+1}.distance$
  - Die übrigen Ungleichungen sind von der Operation nicht betroffen
  - → Annahme gilt nach Einreihung eines neuen Knotens

q.e.d

Beweis IV

- Annahme: v_i und v_j werden während einer Breitensuche in die Warteschlange eingereiht und v_i wird vor v_j eingereiht.
  - Dann gilt $v_i.distance \leq v_j.distance$ zum Zeitpunkt zu dem v_j eingereiht wird

- Beweis IV
  - Folgt direkt aus Beweis III und weil jedem Knoten innerhalb einer Breitensuche höchstens einmal ein endlicher $distance$-Wert zugewiesen wird.

    q.e.d

- Nun kann gezeigt werden, dass Breitensuche die kürzesten Pfade zu allen Knoten vom Startknoten aus liefert

Korrektheit der Breitensuche

- Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und Breitensuche wird auf G mit einem gegebenen Startknoten s ausgeführt
  - Dann entdeckt Breitensuche jeden Knoten $v \in V$ der von s aus erreichbar ist und nach Abschluss gilt $v.distance = \delta(s, v)$ für alle $v \in V$
  - Zusätzlich gilt für jeden Knoten $v \neq s$, der von s aus erreichbar ist, dass einer der kürzesten Pfade von s nach v der kürzeste Pfad von s nach $v.predecessor$ gefolgt von der Kante $(v.predecessor, v)$ ist
- Beweis durch Widerspruch
  - Aus der Annahme, dass ein Knoten einen zu großen $distance$-Wert zugewiesen bekommt wird eine Ungleichung gefolgert, die im Widerspruch zu den vorausgehenden Beweisen steht

Korrektheit der Breitensuche

- Beweis
  - Sei v ein Knoten dessen $distance$-Wert $\neq \delta(s,v)$ ist
  - → $v \neq s$
  - Da $v.distance \geq \delta(s,v)$ gelten muss (siehe Beweis II) folgt $v.distance > \delta(s,v)$
  - → v muss von s aus erreichbar sein, da sonst $\delta(s,v) = \infty$
  - Sei u der direkte Vorgänger von v auf einem kürzesten Pfad von s nach v, so dass $\delta(s,v) = \delta(s,u) + 1$
  - Da $\delta(s,u) < \delta(s,v)$ und durch die Wahl von u $u.distance = \delta(s,u)$ gilt, folgt die Ungleichung

  - $v.distance > \delta(s,v) = \delta(s,u) + 1 = u.distance + 1$

Korrektheit der Breitensuche

- Betrachte den Zeitpunkt, zu dem die Breitensuche u aus Q entfernt
- v kann zu diesem Zeitpunkt in folgenden Stadien sein
  - v ist noch unentdeckt, also $v.visited = false$
    - → $v.distance$ wird auf $u.distance + 1$ gesetzt
  - $v.visited = true$ und v ist nicht mehr in Q
    - Nach Beweis IV folgt $v.distance \leq u.distance$
  - $v.visited = true$ und v ist noch in Q
    - v wurde zu Q früher, beim Entfernen eines Knotens w, hinzugefügt
    - → w wurde früher als u aus Q entfernt
    - → $w.distance \leq u.distance$ und damit
      $v.distance = w.distance + 1 \leq u.distance + 1$

$v.distance > u.distance + 1$

$v.distance > u.distance + 1$

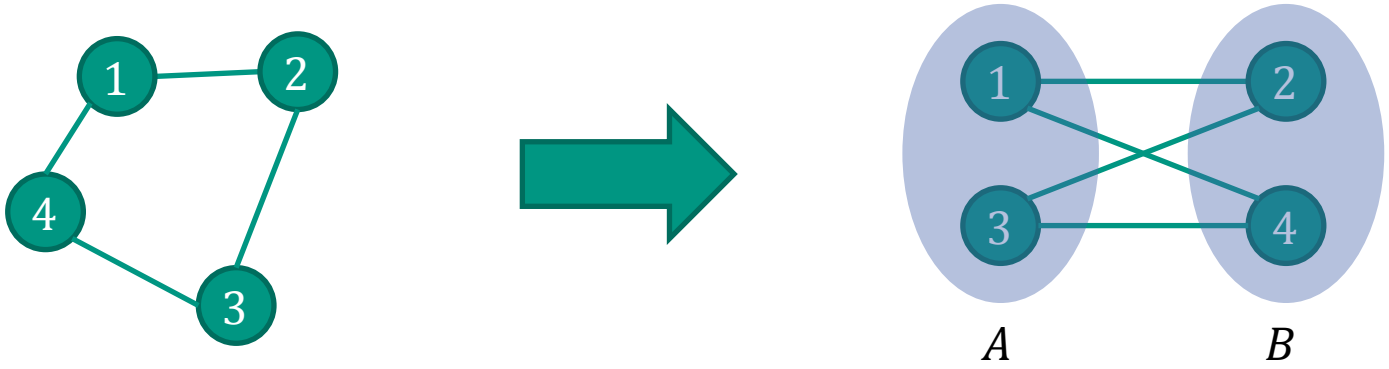
$v.distance > u.distance + 1$

- Alle von s aus erreichbaren Knoten müssen entdeckt werden, da sonst $\infty = v.distance > \delta(s, v)$
- Falls $v.predecessor = u$ → $v.distance = u.distance + 1$
  - → Kürzester Pfad von s nach v ist der kürzeste Pfad von s nach $v.predecessor$ gefolgt von der Kante $(v.predecessor, v)$

q.e.d

8.6 Bipartite Graphen

- Ein Graph ist bipartit, wenn sich seine Knoten in zwei Mengen A und B aufteilen lassen, so dass innerhalb der Teilmengen keine Kanten verlaufen
- → $\forall (u,v) \in E: (u \in A, v \in B)\ oder\ (u \in B, v \in A)$



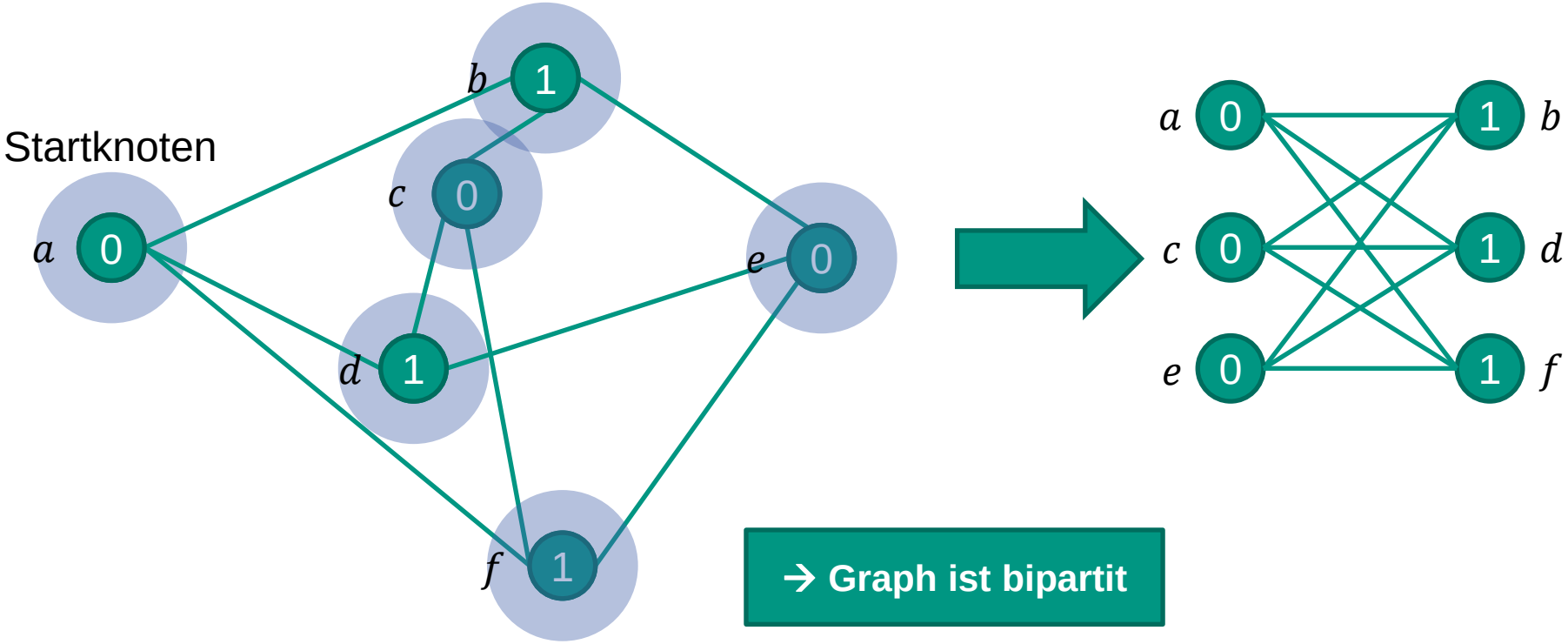
- Für zusammenhängenden Graphen kann mittels Breitensuche entschieden werden, ob er bipartit ist

Bipartit-Test

- Vorgehensweise bipartit-Test
  - Breitensuche mit beliebigem Knoten starten
  - Startknoten mit Parität 0 markieren
  - Jeder neu entdeckte Knoten bekommt die invertierte Parität seines Vorgängers
    - Bei Vorgängerparität 0 folgt neue Parität 1
    - Bei Vorgängerparität 1 folgt neue Parität 0
  - Jeder wieder entdeckte Knoten wird auf Parität überprüft
    - Ignorieren bei unterschiedlicher Parität zum Vorgängerknoten
    - Abbruch bei gleicher Parität zum Vorgängerknoten

- Ergebnis
  - Bei Abbruch ist der Graph nicht bipartit
  - Andernfalls ist der Graph bipartit
    - Parität der Knoten ordnet sie in zwei Mengen ein

Beispiel


Startknoten
a
0
b
1
c
0
d
1
e
0
f
1
a
0
c
0
e
0
1
b
1
d
1
f
→ Graph ist bipartit

Rückblick

- Breitensuche betrachtet Knoten mit aufsteigender Distanz vom Startknoten
- Gesamtaufwand liegt in in $\Theta(|V| + |\mathrm{E}|)$
- Breitensuche liefert kürzeste Pfade vom Startknoten zu allen erreichbaren Knoten
- Beispielhafte Anwendungen der Breitensuche
  - Prüfen, ob ein Graph bipartit ist

Zusammenfassung

- Breitensuche und Tiefensuche haben denselben Aufwand
  - Auswahl hauptsächlich von der Anwendung abhängig
  - Anwendung ist häufig Variation des hier vorgestellten Algorithmus

- Vorteil Breitensuche
  - Keine Rekursion nötig

- Vorteil Tiefensuche
  - Keine explizite TODO-Datenstruktur nötig

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 9 – Kürzeste Pfade

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

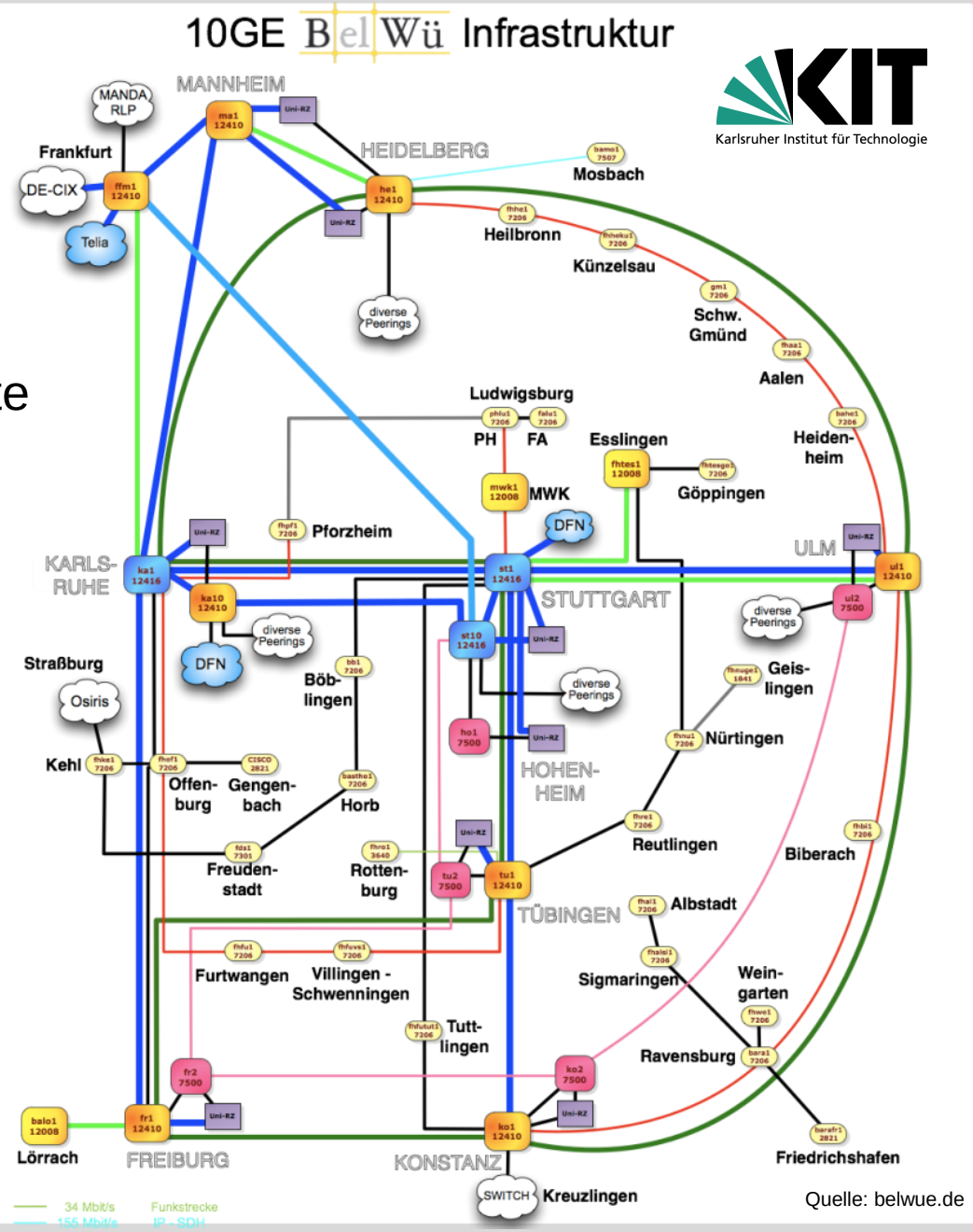
Aufbau der Vorlesung

9.1 Motivation

- Reise von Karlsruhe nach Berlin
  - Welche Route ist am kürzesten?
- Straßenkarte liefert
  - Straßenkreuzungen
  - Abstände dazwischen
- Naive Lösung
  - Alle möglichen Routen aufzählen
  - Distanzen ausrechnen
  - Minimum auswählen
- Problem
  - Unnötig viele Routen betrachtet

Motivation

- Router im Internet
- Über welches Interface Pakete weiterleiten?
- Welcher Link ist besser von Karlsruhe nach Frankfurt?
  - 10Gbit/s über Mannheim
  - 2,4Gbit/s direkt
- Kürzester Weg nicht immer bester Weg
  - Bewertungsfunktion nötig



Quelle: belwue.de

9.2 Kürzeste Pfade

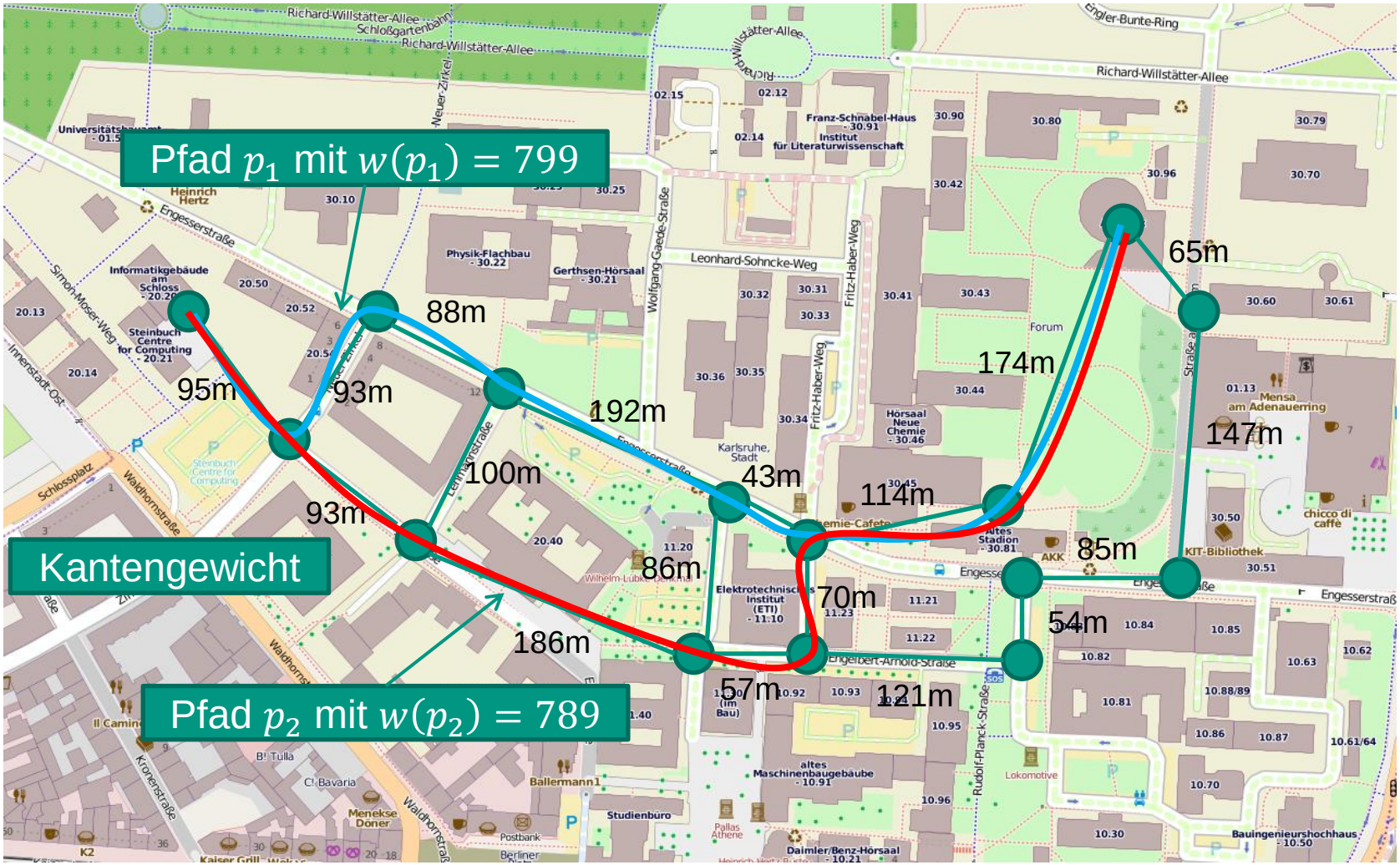
- Eingabeparameter für kürzeste Pfade
  - Gerichteter, gewichteter Graph $G = (V, E)$
  - Abbildung $w : E \rightarrow \mathbb{R}$, die jeder Kante eine Gewicht zuweist

- Gewicht eines Pfades $w(p)$ mit $p = \langle v_0, v_1, \ldots, v_k \rangle$
  - Summe der Gewichte der benutzten Kanten
  - $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

- Gewicht eines kürzesten Pfades $\delta(u, v)$ von u nach v
  - $\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{wenn Pfad von u nach v existiert} \\ \infty, & \text{sonst} \end{cases}$

- Kürzester Pfad von u nach v ist jeder Pfad p mit
  - $w(p) = \delta(u, v)$

Kürzeste Pfade – Beispiel

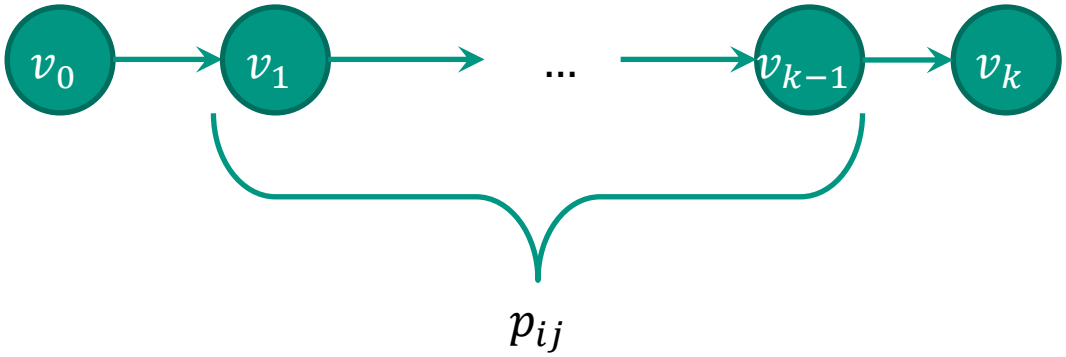


Kürzeste Pfade – Problemvarianten

- Single-source
  - Berechne kürzeste Pfade zu allen Knoten von einem Startknoten aus
- Single-destination
  - Berechne kürzeste Pfade von allen Knoten zu einem Zielknoten
  - Durch Invertieren aller Kanten überführbar in ein Single-source Problem
- Single-pair
  - Berechne kürzesten Pfad von u nach v für gegebene Knoten
  - Single-source mit u als Startknoten löst dieses Problem
- All-pairs
  - Berechne kürzeste Pfade für alle möglichen Paare u, v
  - $|V|$ -mal Single-source

Teilpfade kürzester Pfade

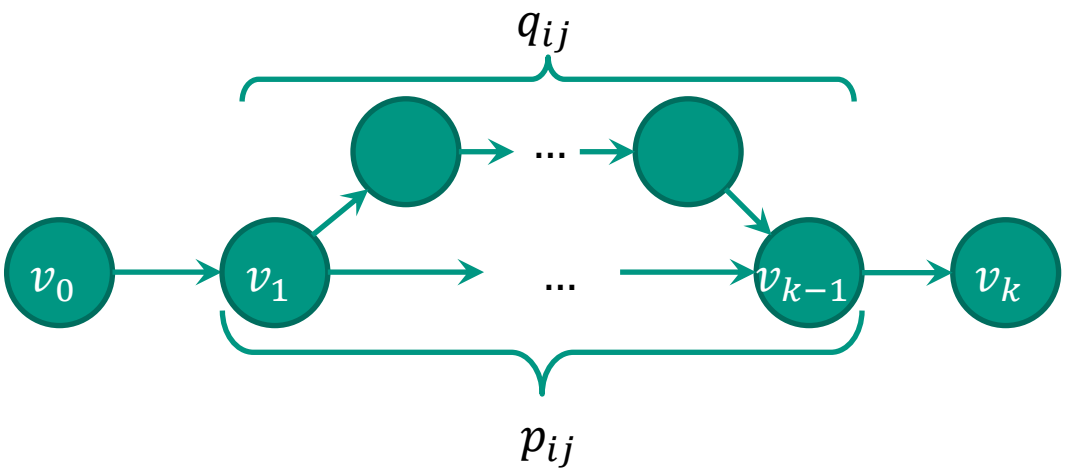
- Teilpfade kürzester Pfade sind selbst kürzeste Pfade
  - Wenn $p = \langle v_0, v_1, \ldots, v_k \rangle$ ein kürzester Pfad von v_0 nach v_k ist
  - Sei $p_{ij} = \langle v_i, \ldots, v_j \rangle$ für i, j mit $0 \leq i \leq j \leq k$ ein Teilpfad von p
  - → p_{ij} ist ein kürzester Pfad von v_i nach v_j



Teilpfade kürzester Pfade

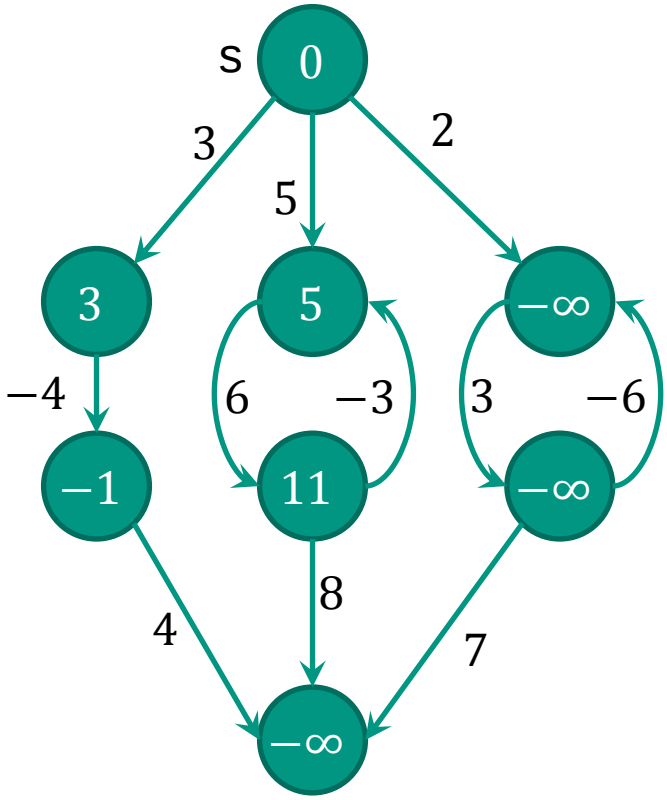
- Beweis
  - Zerlege p in $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} \mathrm{v_k}$
  - → $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$
  - Angenommen q_{ij} wäre ein Pfad von v_i nach v_j mit $w(q_{ij}) < w(p_{ij})$
  - Dann wäre $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{q_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} \mathrm{v_k}$ ein Pfad von v_0 nach v_k mit Gewicht $w(p_{0i}) + w(q_{ij}) + w(p_{jk}) < w(p)$
  - Widerspruch zu p ist ein kürzester Pfad

q.e.d



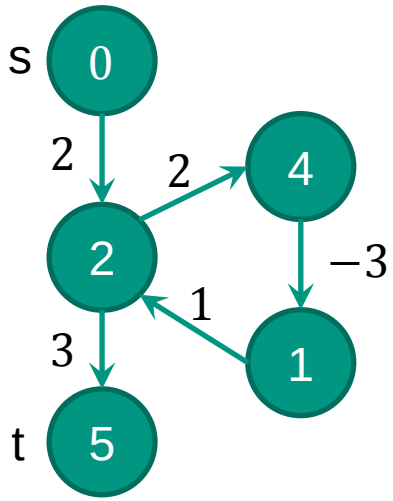
Negative Kantengewichte

- Negative Kantengewichte
  - Prinzipiell kein Problem
  - $\delta(u, v)$ kann auch negativ sein
  - Bei erreichbaren Zyklen mit negativem Kantengewicht problematisch
    - Kein Pfad kann kürzester sein, da ein weiterer Durchlauf durch den negativen Zyklus einen noch kürzeren Pfad ergibt
  - $\delta(u, v) = -\infty$ falls ein solcher Zyklus auf einem Pfad von u nach v liegt



Zyklen in kürzesten Pfaden

- Kann ein kürzester Pfad einen Zyklus enthalten?
  - Keine Zyklen mit negativem Kantengewicht
    - Siehe vorherige Folie
  - Keine Zyklen mit positivem Kantengewicht
    - Entfernen des Zyklus aus dem Pfad
    - Neuer Pfad mit gleichen Start- und Zielknoten
    - Gewicht des Pfades geringer → kürzerer Pfad
  - Keine Zyklen mit neutralem Kantengewicht
    - Zyklus mit Kantengewicht 0 kann entfernt werden
    - Neuer Pfad mit gleichem Gewicht, aber weniger Kanten
    - Zyklen können sukzessive entfernt werden bis der Pfad zyklenfrei ist
    - Im Weiteren wird O.B.d.A. von zyklenfreien kürzesten Pfaden ausgegangen



Repräsentation kürzester Pfade

- Nicht nur Gewicht des kürzesten Pfades, sondern auch die Knoten auf dem Pfad sind von Interesse
- Lösung ähnlich zum Breadth-First-Tree aus Kapitel 8
  - Für jeden Knoten v wird sein Vorgänger in $v.predecessor$ gespeichert
    - $v.predecessor$ ist entweder ein anderer Knoten oder NIL
  - Durch Folgen der $predecessor$-Werte beginnend bei Knoten v traversiert man den kürzesten Pfad vom Startknoten s nach v in umgekehrter Reihenfolge
  - → Vorgängergraph $G_{pred} = (V_{pred}, E_{pred})$ mit
    - $V_{pred} = \{v \in V : v.predecessor \neq NIL\} \cup \{s\}$
    - $E_{pred} = \{(v.predecessor, v) \in E : v \in V_{pred} - \{s\}$

„Gliederung“

- Initialisierung
  - $INITIALIZE\_SINGLE\_SOURCE(G, s)$

```
1    foreach v ∈ G.V
2        v.distance = ∞
3        v.predecessor = NIL
4    s.distance = 0
```

Laufzeit in $\Theta(|V|)$

- Relaxation
  - „Verbesserung“ des aktuell kürzesten Pfads von s nach v
- Algorithmus für kürzeste Pfade
  - Bellman-Ford
  - Dijkstra

Relaxation

- Relaxation
  - Methode in den im Folgenden vorgestellten Algorithmen
  - Für jeden Knoten $v \in V$ wird Attribut $v.distance$ verwaltet
  - $v.distance$ ist obere Schranke für das Gewicht des kürzesten Pfades von Startknoten s nach v
  - $v.distance =$ Abschätzung des kürzesten Pfades

- Relaxieren einer Kante (u, v)
  - Prüfen, ob bisher gefundener kürzester Pfad nach v durch Pfad über u verkürzbar
  - Falls ja: Attribute $v.distance$ und $v.predecessor$ entsprechend anpassen

Relaxation

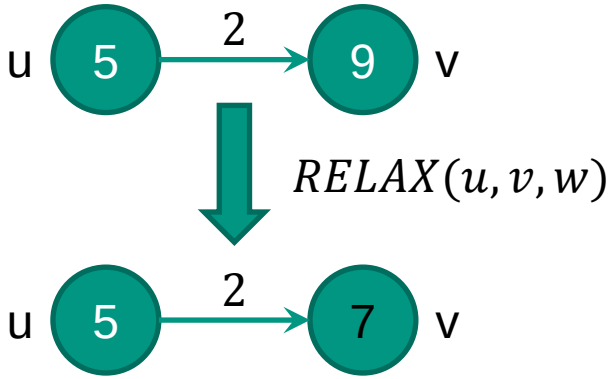
- $RELAX(u, v, w)$

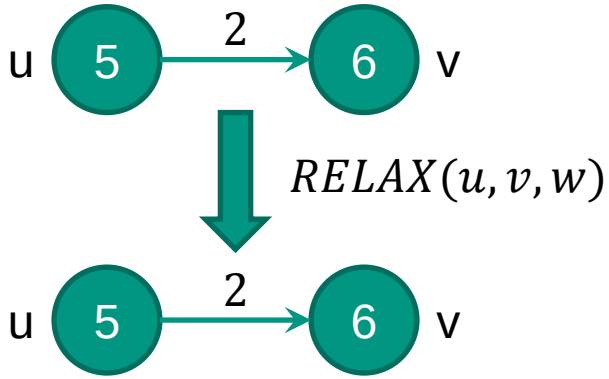
```
1    if v.distance > u.distance + w(u,v)
2        v.distance = u.distance + w(u,v)
3        v.predecessor = u
```

Gewichtungsfunktion

Laufzeit in $O(1)$

- Beispiele





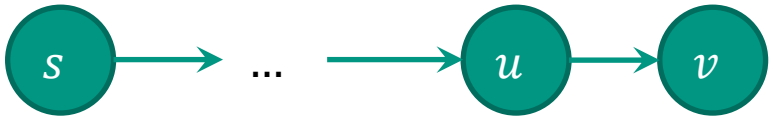
Wichtige Eigenschaften

- Wichtige Eigenschaften kürzester Pfade und der Relaxation
  - Im Folgenden für Korrektheitsbeweise verwendet
  - Annahme: Schätzungen der kürzesten Pfade und Vorgänger ändern sich nur durch Relaxationsschritte

- Für die folgenden Beweise gilt
  - Sei $G = (V, E)$ ein gewichteter, gerichteter Graph mit Gewichtungsfunktion $w: E \to \mathbb{R}$
  - Sei $s \in V$ der Startknoten
  - Der Graph wurde mit $INITIALIZE\_SINGLE\_SOURCE(G, s)$ initialisiert

Dreiecksungleichung

- Dreiecksungleichung
  - Für alle Kanten $(u, v) \in E$ gilt $\delta(s, v) \leq \delta(s, u) + w(u, v)$



- Beweis
  - Sei p ein kürzester Pfad von s nach v.
  - → Das Gewicht von p ist kleiner oder gleich dem Gewicht aller anderen Pfade von s nach v
  - Im Speziellen ist es auch kleiner als das des Pfades, der einen kürzesten Weg von s nach u und abschließend die Kante (u, v) nimmt

q.e.d

Obere Schranke

- Obere Schranke
  - Für alle Knoten $v \in V$ gilt $v.distance \geq \delta(s,v)$ und sobald $v.distance = \delta(s,v)$ ändert sich $v.distance$ nicht mehr

- Beweis durch Induktion über Relaxationsschritte
  - Nach Initialisierung gilt $v.distance \geq \delta(s,v)$ da $v.distance = \infty \; \forall v \in V - \{s\}$ und $s.distance = 0$
  - Sei (u,v) eine Kante, die relaxiert wird
    - Vor der Relaxation gilt $\forall x \in V: x.distance \geq \delta(s,x)$
    - Nur der d-Wert von v kann sich während der Relaxation ändern
    - $v.distance = u.distance + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,v)$ (Induktionsannahme; Dreiecksungleichung)
    - → Auch nach der Relaxation gilt $\forall x \in V: x.distance \geq \delta(s,x)$
  - Da $v.distance$ durch Relaxation nur kleiner werden kann und $\forall v \in V: v.distance \geq \delta(s,v)$ gilt, ändert sich $v.distance$ nicht mehr, wenn es den Wert $\delta(s,v)$ einmal angenommen hat

q.e.d

„Kein-Pfad“ Eigenschaft

- „Kein-Pfad“ Eigenschaft
  - Falls kein Pfad von s nach v existiert gilt $v.distance = \delta(s, v) = \infty$

- Beweis
  - Durch obere Schranke gilt $\forall v \in V\colon\ v.distance \geq \delta(s, v)$
  - Da $\delta(s, v) = \infty$ und $\delta(s, v) \leq v.distance$ gilt
  - auch $v.distance = \infty = \delta(s, v)$

q.e.d

Konvergenzeigenschaft

- Konvergenzeigenschaft
  - Wenn $s \rightsquigarrow u \rightarrow v$ für ein Knotenpaar $u, v \in V$ ein kürzester Pfad von s nach v in G ist und $u.distance = \delta(s,u)$ vor der Relaxation der Kante (u,v) gilt, dann gilt $v.distance = \delta(s,v)$ nach der Relaxation.

- Beweis
  - Wenn $u.distance = \delta(s,u)$ zu einem Zeitpunkt vor der Relaxierung von (u,v) galt, dann gilt es auch danach (obere Schranke)
  - Im Speziellen gilt nach Relaxierung von (u,v)
    - $v.distance \leq u.distance + w(u,v) = \delta(s,u) + w(u,v) = \delta(s,v)$

    Teilpfade von kürzesten Pfade sind kürzeste Pfade
  - Durch die obere Schranke muss auch $v.distance \geq \delta(s,v)$ und daher gilt
  - $v.distance = \delta(s,v)$ nach der Relaxation.

q.e.d

Pfadrelaxation

- Pfadrelaxation
  - Wenn $p = \langle v_0, v_1, \ldots, v_k \rangle$ ein kürzester Pfad von $s = v_0$ nach v_k ist und die Kanten von p in Reihenfolge $(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$ relaxiert werden, dann gilt $v_k.distance = \delta(s, v_k)$
  - Dies gilt ungeachtet aller anderen Relaxationsschritte

- Beweis durch Induktion über die Relaxationsschritte
  - Zu zeigen: Nach dem Relaxieren der i-ten Kante in p gilt $v_i.distance = \delta(s, v_i)$
  - Für $i = 0$ gilt durch die Initialisierung $v_0.distance = s.distance = 0 = \delta(s, s)$
    - Durch die obere Schranke ändert sich dieser Wert nicht mehr
  - Induktionsschritt: Sei $v_{i-1}.distance = \delta(s, v_{i-1})$
  - Relaxiere die Kante (v_{i-1}, v_i)
    - Durch Konvergenzeigenschaft gilt nach der Relaxation $v_i.distance = \delta(s, v_i)$
    - Dieser Wert ändert sich nicht mehr

q.e.d

Rückblick

- Grundlegendes Vokabular
  - Pfad, Kantengewicht, kürzeste Pfade, …
- Problemklassifikation
  - Hier Konzentration auf Single-Source
- „Hilfsmittel“ für nachfolgende Algorithmen
  - Initialisierung
  - Relaxieren von Kanten
- Grundlegende Eigenschaften als Handwerkszeug für nachfolgende Beweise

9.3 Bellman-Ford-Algorithmus

- Von den amerikanischen Mathematikern Richard Bellman und Lester Ford Jr. Entwickelt
- Veröffentlicht 1958 unter dem Titel „On a routing problem“

Richard Bellman

- Löst single-source Probleme im allgemeinen Fall
  - Gewichteter, gerichteter Graph $G = (V, E)$ mit Startknoten s und Gewichtsfunktion $w : E \to \mathbb{R}$
  - Auch für negative Kantengewichte
- Erreichbare Zyklen mit negativen Kantengewichten werden entdeckt

Lester Ford Jr.

- Vorgehensweise
  - Gibt TRUE zurück falls Lösung existiert, sonst FALSE
  - Algorithmus relaxiert Kanten
  - Reduziert dadurch $v.distance$ für alle $v \in V$ bis $\delta(s, v)$ erreicht ist

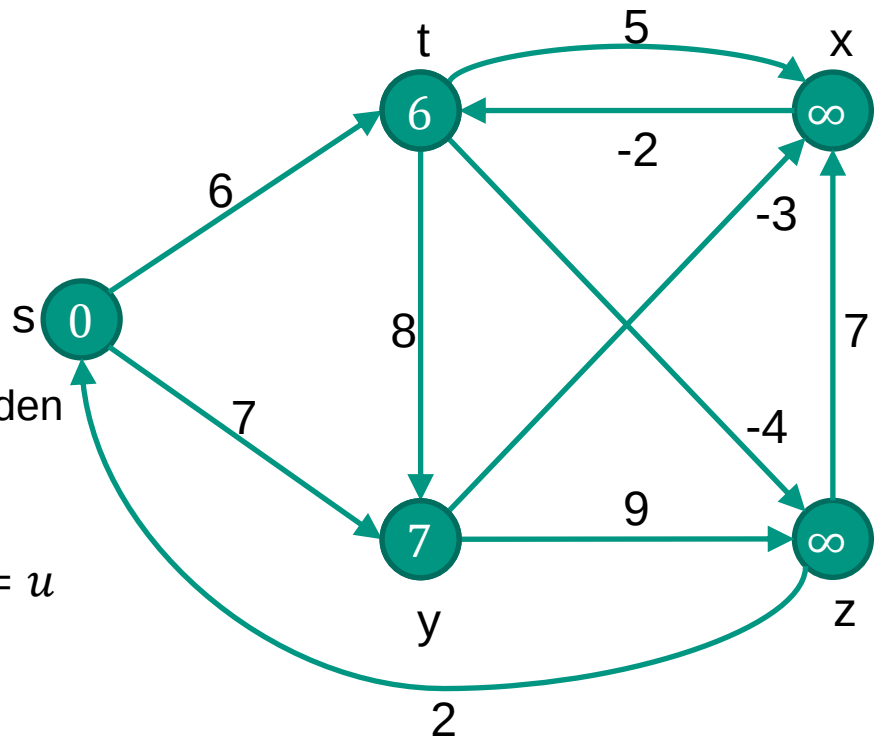
Bellman-Ford-Algorithmus

- $BELLMAN\_FORD(G, w, s)$

```
1   INITIALIZE_SINGLE_SOURCE(G, s)
2   for i = 1 to |G.V| − 1
3       foreach (u, v) ∈ G.E
4           RELAX(u, v, w)
5   foreach (u, v) ∈ G.E
6       if v.distance > u.distance + w(u, v)
7           return false
8   return true
```

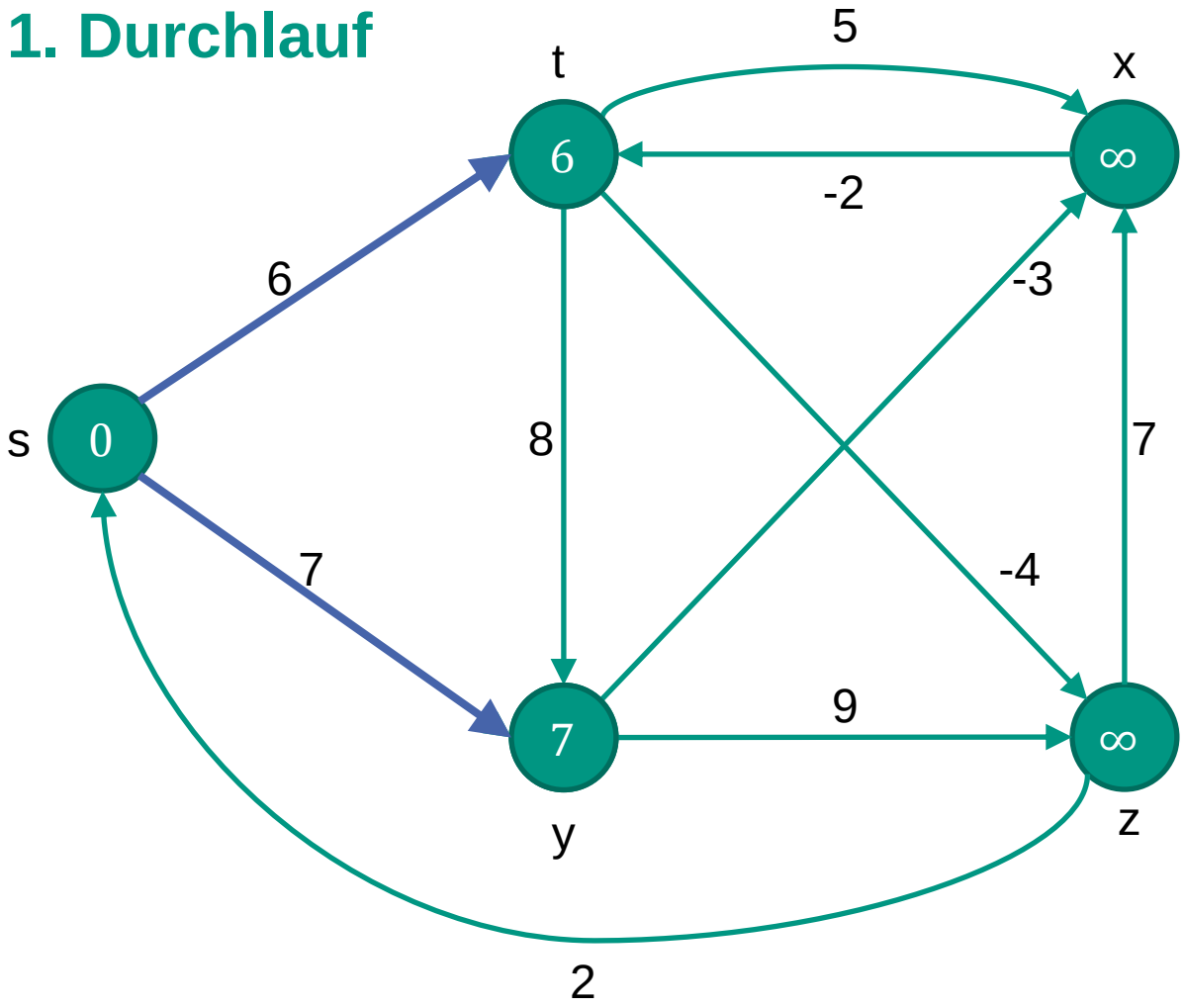
Bellman-Ford-Algorithmus – Beispiel

- Beispielgraph
  - Enthält negative Kantengewichte
  - $distance$-Werte im Knoten dargestellt
    - Im Folgenden wird aus Platzgründen $v.distance$ mit $v.d$ abgekürzt
  - Kante (u, v) ist dicker und blau gezeichnet wenn $v.predecessor = u$
  - Relaxationsreihenfolge hier
    - $(t,x), (t,y), (t,z)$
    - (x,t)
    - $(y,x), (y,z)$
    - $(z,x), (z,s)$
    - $(s,t), (s,y)$



Bellman-Ford-Algorithmus – Beispiel

1. Durchlauf



⋮

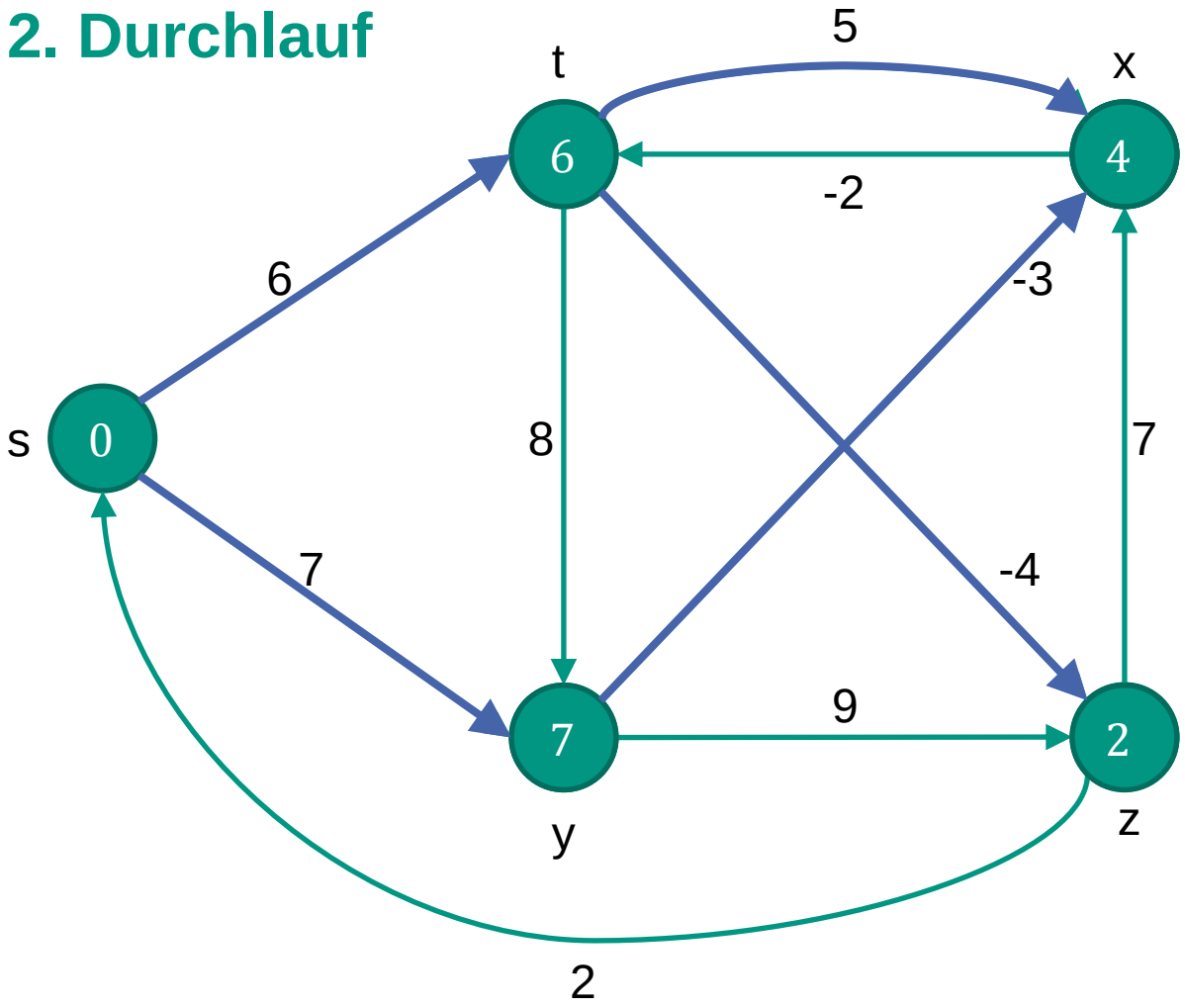
Keine Änderungen bis Knoten s

$RELAX(s, t, 6) \rightarrow t.d = \mathbf{6}$

$RELAX(s, y, 7) \rightarrow y.d = \mathbf{7}$

Bellman-Ford-Algorithmus – Beispiel

2. Durchlauf



$RELAX(t, x, 5) \rightarrow x.d = \mathbf{11}$

$RELAX(t, y, 8) \rightarrow y.d = 7$

$RELAX(t, z, -4) \rightarrow z.d = \mathbf{2}$

$RELAX(x, t, -2) \rightarrow t.d = 6$

$RELAX(y, x, -3) \rightarrow x.d = \mathbf{4}$

$RELAX(y, z, 9) \rightarrow z.d = 2$

$RELAX(z, x, 7) \rightarrow x.d = 4$

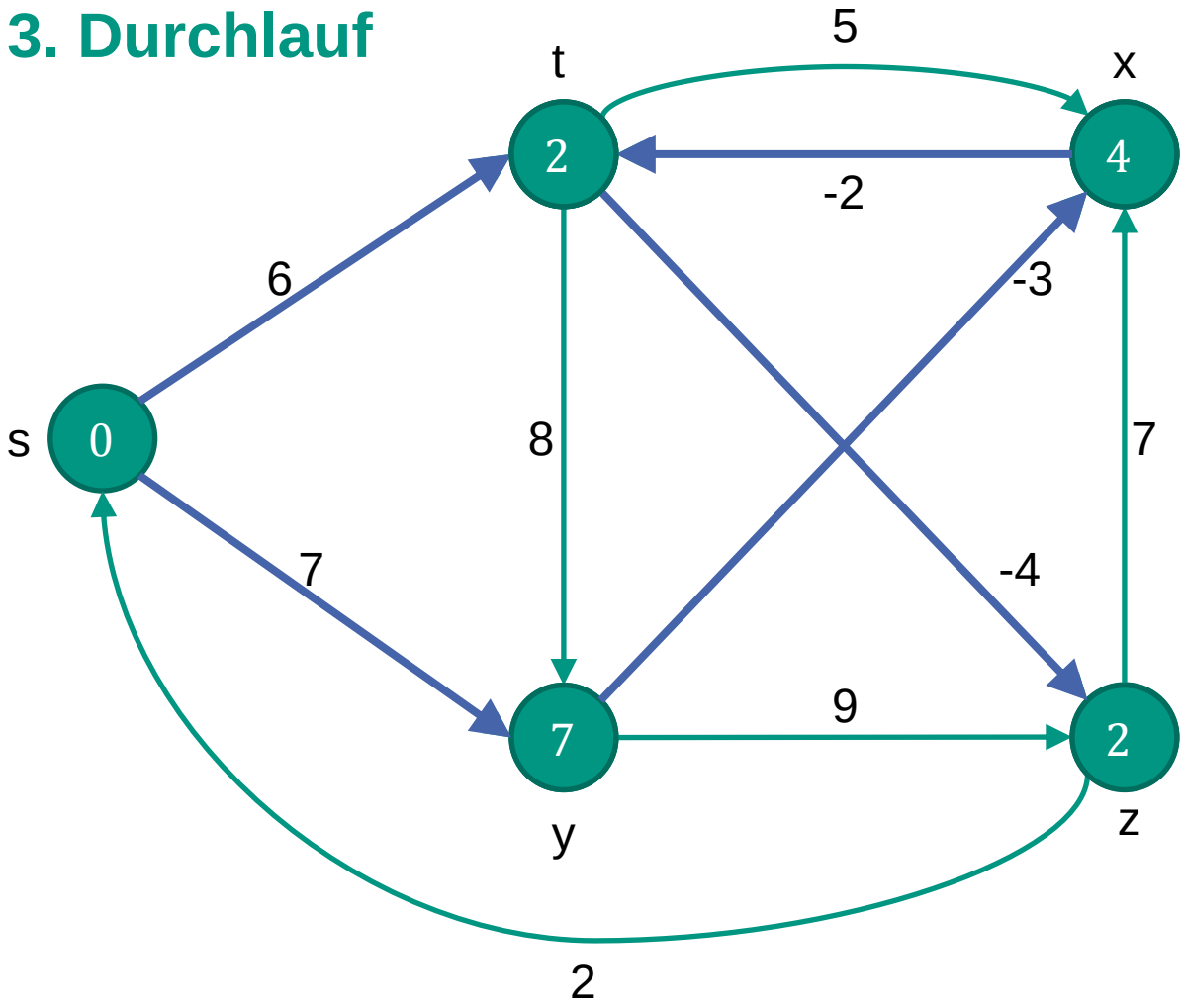
$RELAX(z, s, 2) \rightarrow s.d = 0$

$RELAX(s, t, 6) \rightarrow t.d = 6$

$RELAX(s, y, 7) \rightarrow t.d = 7$

Bellman-Ford-Algorithmus – Beispiel

3. Durchlauf



$RELAX(t, x, 5) \rightarrow x.d = 4$

$RELAX(t, y, 8) \rightarrow y.d = 7$

$RELAX(t, z, -4) \rightarrow z.d = 2$

$RELAX(x, t, -2) \rightarrow t.d = \mathbf{2}$

$RELAX(y, x, -3) \rightarrow x.d = 4$

$RELAX(y, z, 9) \rightarrow z.d = 2$

$RELAX(z, x, 7) \rightarrow x.d = 4$

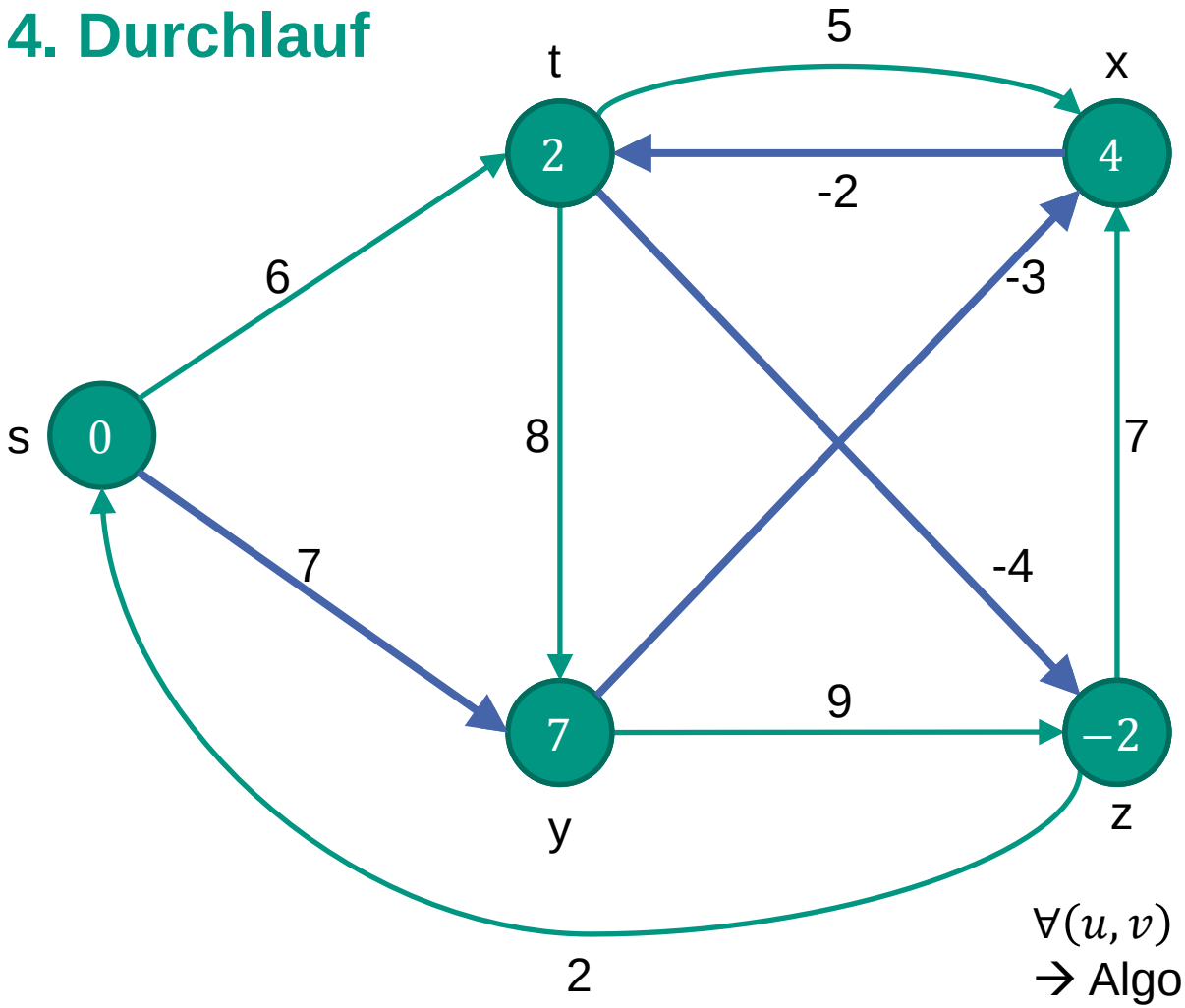
$RELAX(z, s, 2) \rightarrow s.d = 0$

$RELAX(s, t, 6) \rightarrow t.d = 2$

$RELAX(s, y, 7) \rightarrow t.d = 7$

Bellman-Ford-Algorithmus – Beispiel

4. Durchlauf



$RELAX(t, x, 5) \rightarrow x.d = 4$

$RELAX(t, y, 8) \rightarrow y.d = 7$

$RELAX(t, z, -4) \rightarrow z.d = \mathbf{-2}$

$RELAX(x, t, -2) \rightarrow t.d = 2$

$RELAX(y, x, -3) \rightarrow x.d = 4$

$RELAX(y, z, 9) \rightarrow z.d = 2$

$RELAX(z, x, 7) \rightarrow x.d = 4$

$RELAX(z, s, 2) \rightarrow s.d = 0$

$RELAX(s, t, 6) \rightarrow t.d = 2$

$RELAX(s, y, 7) \rightarrow t.d = 7$

$\forall (u, v) \in E: v.d \leq u.d + w(u, v)$

→ Algorithmus liefert true zurück

Bellman-Ford-Algorithmus – Analyse

```
BELLMAN_FORD(G, w, s)
1    INITIALIZE_SINGLE_SOURCE(G, s)
2    for i = 1 to |G.V| − 1
3        foreach (u, v) ∈ G.E
4            RELAX(u, v, w)
5    foreach (u, v) ∈ G.E
6        if v.distance > u.distance + w(u, v)
7            return false
8    return true
```

Laufzeit in $\Theta(|V|)$

In jedem der $|V| - 1$ Durchläufe $\Theta(|E|)$, da für jede Kante ein Relaxierungsschritt

Laufzeit in $O(|E|)$

- Gesamtlaufzeit in $O(|V| * |E|)$

Bellman-Ford-Algorithmus – Korrektheit

- Sei $G = (V, E)$ ein gewichteter, gerichteter Graph mit Startknoten s und Gewichtsfunktion $w: E \rightarrow \mathbb{R}$, der keine von s aus erreichbaren Zyklen mit negativem Gewicht enthält

Für allgemeine Korrektheit siehe [Corm10] Kapitel 24.1

- Dann gilt nach $|V| - 1$ Iterationen der for-Schleife der Zeilen 2-4 die Gleichung $v.distance = \delta(s, v)$ für alle von s aus erreichbaren Knoten v
- Beweis
  - Sei v ein von s aus erreichbarer Knoten und $p = \langle v_0, v_1, \ldots, v_k \rangle$ ein beliebiger kürzester Pfad mit $v_0 = s$ und $v_k = v$
  - Da kürzeste Pfade o.B.d.A. zyklenfrei sind, hat p maximal $|V| - 1$ Kanten
  - → $k \leq |V| - 1$
  - Jede der $|V| - 1$ Iterationen relaxiert alle $|E|$ Kanten
  - Für $i = 1,2, \ldots, k$ ist die Kante (v_{i-1}, v_i) unter den, in der i-ten Iteration relaxierten Kanten
  - Aufgrund der Pfadrelaxation gilt $v.distance = v_k.distance = \delta(s, v_k) = \delta(s, v)$

q.e.d

9.4 Kürzeste Pfade in DAGs

- Kürzeste Pfade in gewichteten, gerichteten, zyklenfreien Graphen
  - Auch bei negativen Kantengewichten kein negativer Zyklus da generell zyklenfrei

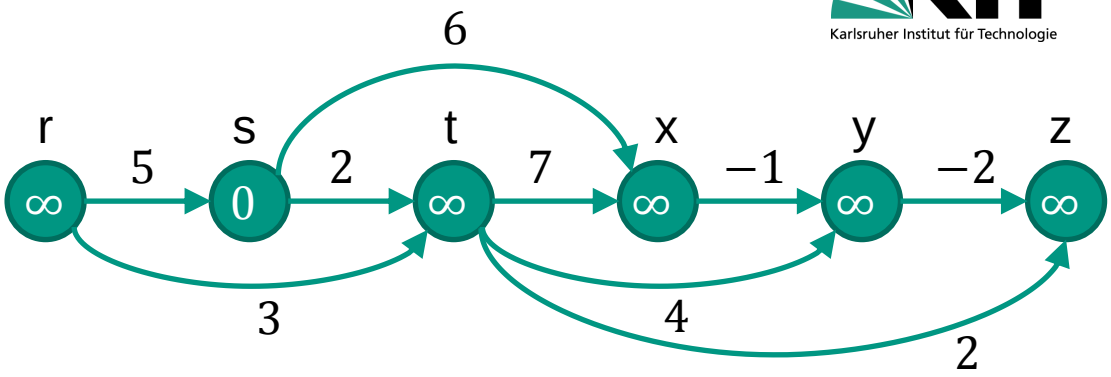
- Single-Source Probleme lassen sich mit Topologischem Sortieren einfach lösen
  - Sortiere den Graphen topologisch
  - Arbeite Knoten des Graphen in topologischer Reihenfolge ab
    - Relaxiere jede ausgehende Kante des gerade betrachteten Knoten

Pseudocode

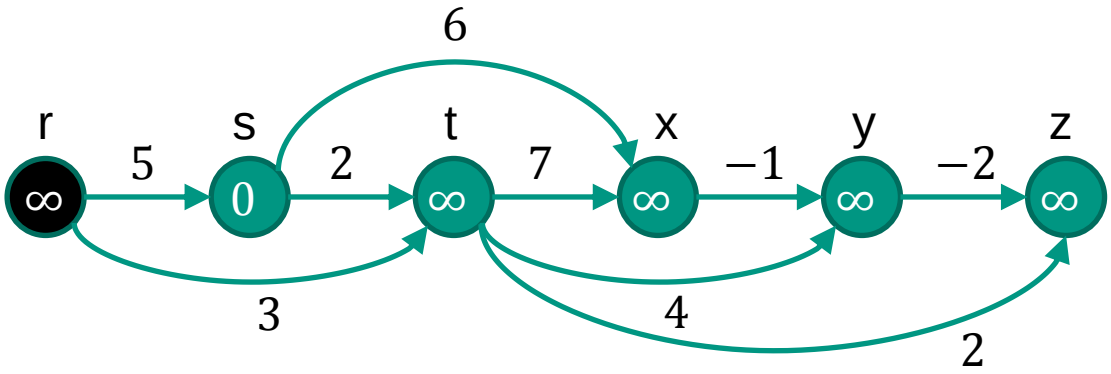
- $DAG\_SHORTEST\_PATHS(G, w, s)$
  1 Sortiere Knoten von G topologisch
  2 $INITIALIZE\_SINGLE\_SOURCE(G, s)$
  3 **foreach** $u \in V$ *in topologischer Reihenfolge*
  7 foreach $v \in G.Adj[u]$
  8 $RELAX(u, v, w)$

Beispiel

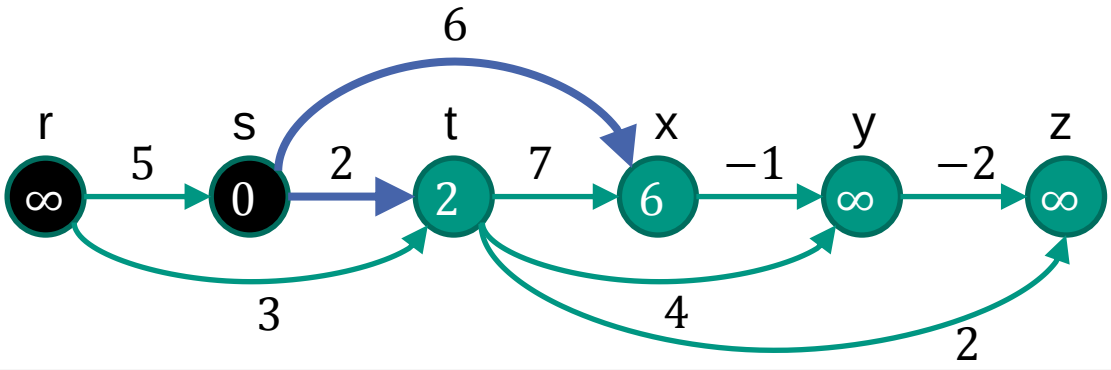
- Zustand nach der Vorverarbeitung



- $RELAX(r, s, 5) \rightarrow s.d = 0$
- $RELAX(r, t, 3) \rightarrow t.d = \infty$



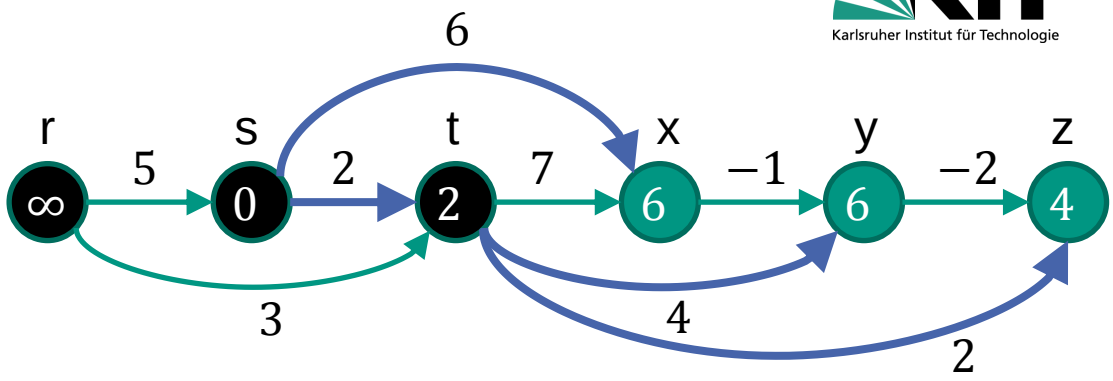
- $RELAX(s, t, 2) \rightarrow \mathbf{t.d = 2}$
- $RELAX(s, x, 6) \rightarrow \boldsymbol{x.d = 6}$



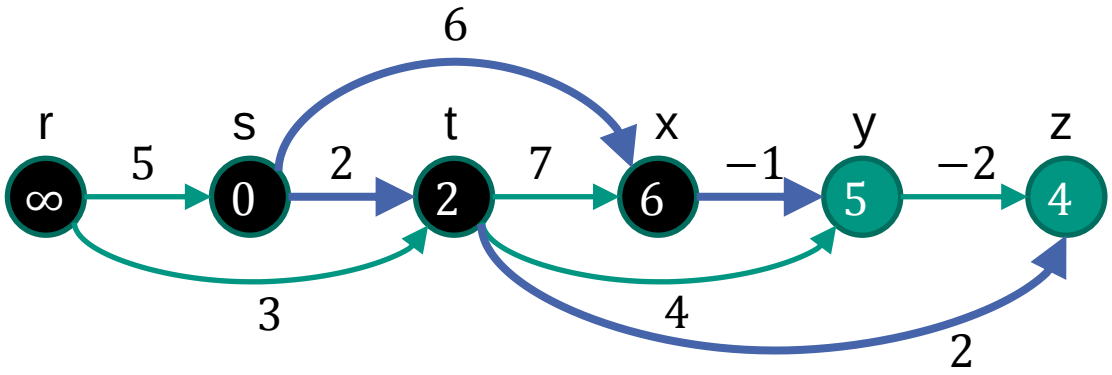
- $RELAX(t, x, 7) \rightarrow x.d = 6$
- $RELAX(t, y, 4) \rightarrow \boldsymbol{y.d = 6}$
- $RELAX(t, z, 2) \rightarrow \boldsymbol{z.d = 4}$

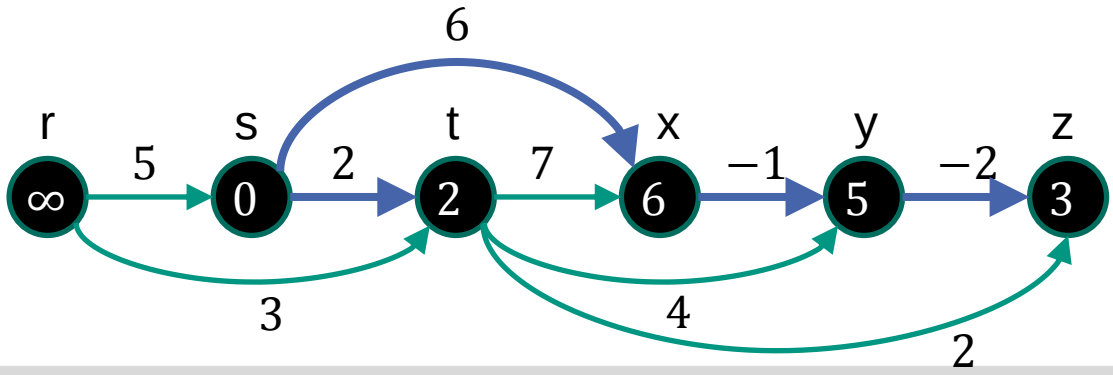
Beispiel

- $RELAX(x, y, -1)$
  $\rightarrow \boldsymbol{y.d = 5}$



- $RELAX(y, z, -2)$
  $\rightarrow \mathbf{z.d = 3}$
- Da z keine ausgehenden Kanten hat → fertig





Analyse

- $DAG\_SHORTEST\_PATHS(G, w, s)$
  1 Sortiere Knoten von G topologisch
  2 $INITIALIZE\_SINGLE\_SOURCE(G, s)$
  3 **foreach** $u \in V$ *in topologischer Reihenfolge*
  7 foreach $v \in G.Adj[u]$
  8 $RELAX(u, v, w)$

$\Theta(|V| + |E|)$

$\Theta(|V|)$

Eine Iteration pro Knoten

Eine Iteration pro Kante des Knoten

Gesamtaufwand in $\Theta(|V| + |E|)$

Korrektheit

- Annahme: Sei $G = (V, E)$ ein gewichteter, gerichteter Graph ohne Zyklen und s ein Startknoten. Dann gilt:
  - Nach Abschluss von $DAG\_SHORTEST\_PATHS$ gilt für alle $v \in V$ $v.distance = \delta(s, v)$

- Beweis
  - Falls v nicht von s aus erreichbar ist, gilt $v.distance = \delta(s, v) = \infty$ durch die „Kein Pfad“-Eigenschaft
  - Falls v von s aus erreichbar ist, existiert ein kürzester Pfad $\mathrm{p} = < v_0, v_1, \ldots, v_k >$ mit $v_0 = s$ und $v_k = v$
  - Da Knoten in topologischer Reihenfolge verarbeitet werden, wurden die Kanten in p in der Reihenfolge $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ verarbeitet
  - Durch die Pfad-Relaxation-Eigenschaft gilt $v_i.distance = \delta(s, v)$ für $i = 0,1, \ldots, k$

q.e.d

9.5 Dijkstra-Algorithmus

- Veröffentlicht vom dänischen Informatiker Edsger Dijkstra 1959 unter dem Titel „A note on two problems in connexion with graphs“

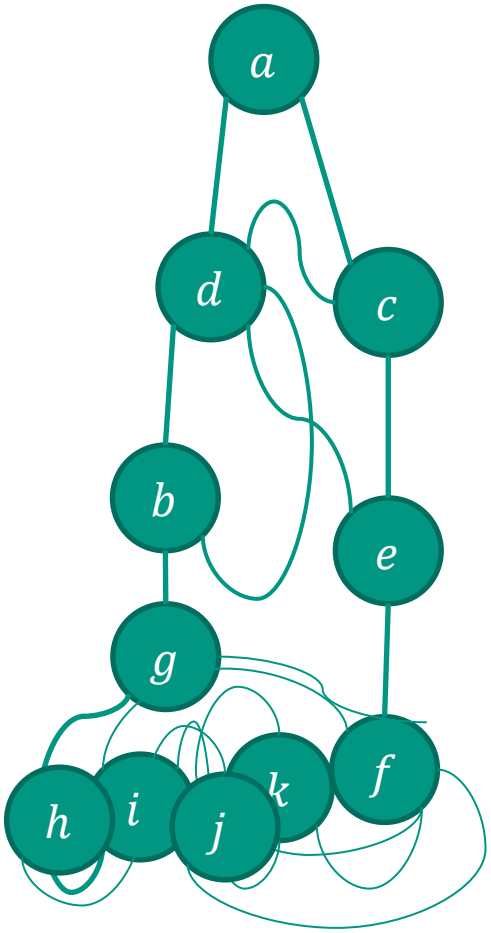
Edsger Dijkstra

- Löst single-source Problem für gewichtete, gerichtete Graphen mit nicht-negativen Kantengewichten
  - Daher gilt $\forall (u,v) \in E\colon\ w(u,v) \geq 0$

- Vorgehensweise
  - Verwaltet Liste mit Knoten, zu denen kürzeste Pfade bereits bestimmt sind
  - Schrittweise aus den übrigen Knoten den auswählen, dessen $distance$-Wert minimal ist und dessen ausgehende Kanten relaxieren

Dijkstra-Algorithmus

- Anschauliche Lösung ohne Rechner
  - Kanten werden zu Fäden
  - Kantengewicht ≙ Fadenlänge
  - Knoten werden zu Knoten
- Knäuel am Startknoten anheben
- Straff gespannte Fäden gehören zu kürzesten Pfaden vom Startknoten aus
- Falls ein kürzerer Pfad als der über einen straff gespannten Faden existieren würde, so wäre einer seiner Fäden zerrissen



Dijkstra-Algorithmus

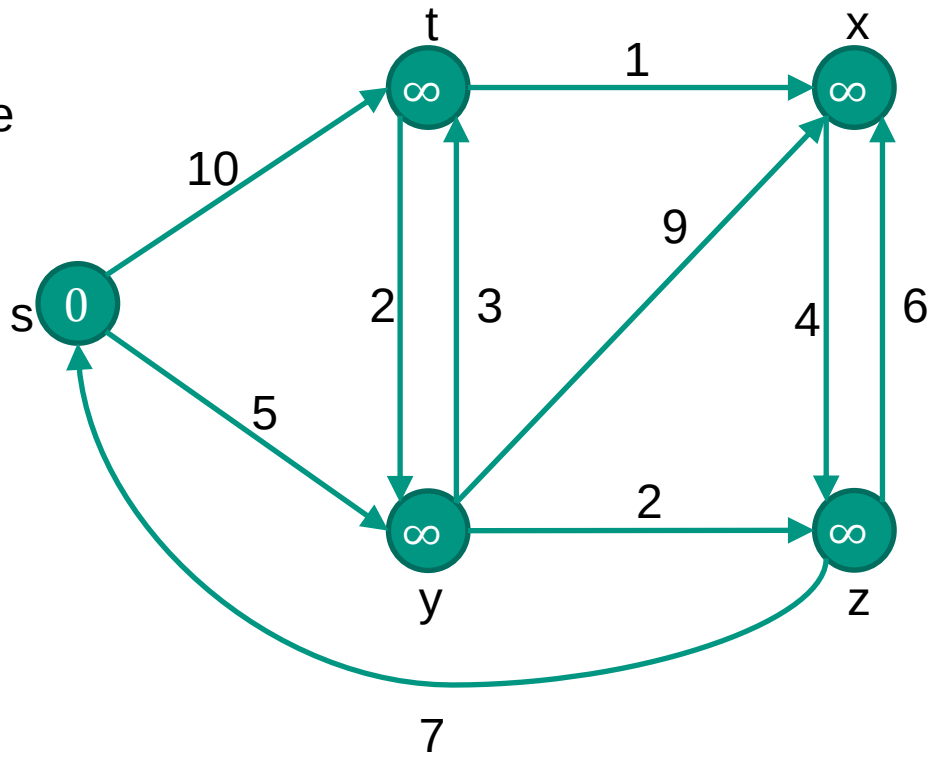
- $DIJKSTRA(G, w, s)$

```
1    INITIALIZE_SINGLE_SOURCE(G, s)
2    S = ∅
3    Q = G.V
4    while Q ≠ ∅
5        u = EXTRACT_MIN(Q)
6        S = S ∪ {u}
7        foreach v ∈ G.Adj[u]
8            RELAX(u, v, w)
```

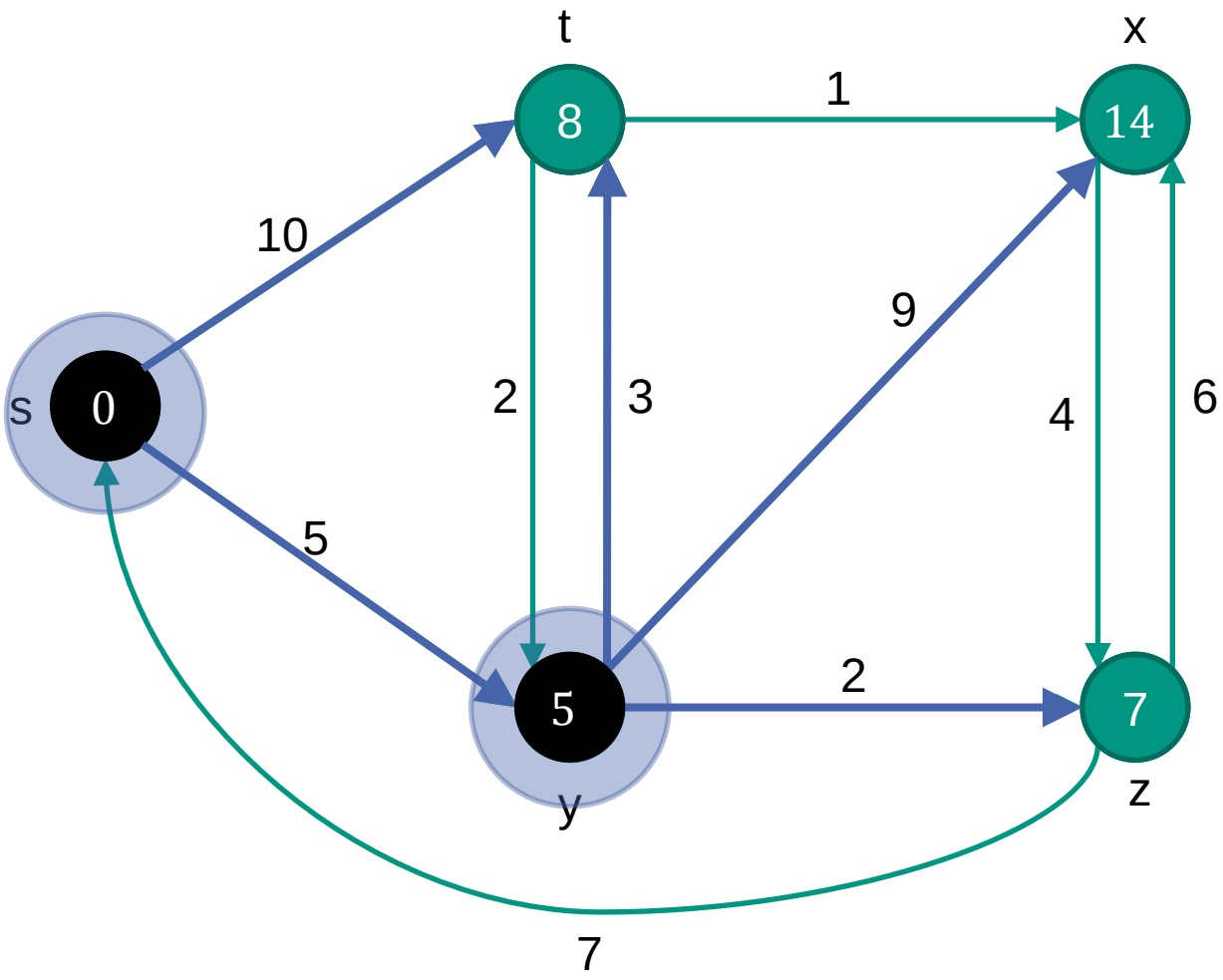
Operationen auf Prioritätswarteschlange

Dijkstra-Algorithmus – Beispiel

- Beispielgraph
  - Keine negativen Kantengewichte
  - $distance$-Werte in Knoten dargestellt
    - Abkürzung $v.d$ für $v.distance$
  - Kante (u, v) ist dicker und blau gezeichnet
    - → $v.predecessor = u$
  - Aktuell betrachteter Knoten wird blau markiert
  - Knoten, die aus Q entfernt werden, werden schwarz gefärbt



Dijkstra-Algorithmus – Beispiel



1. Iteration

$RELAX(s, t, 10) \rightarrow t.d = \mathbf{10}$

$RELAX(s, y, 5) \rightarrow y.d = \mathbf{5}$

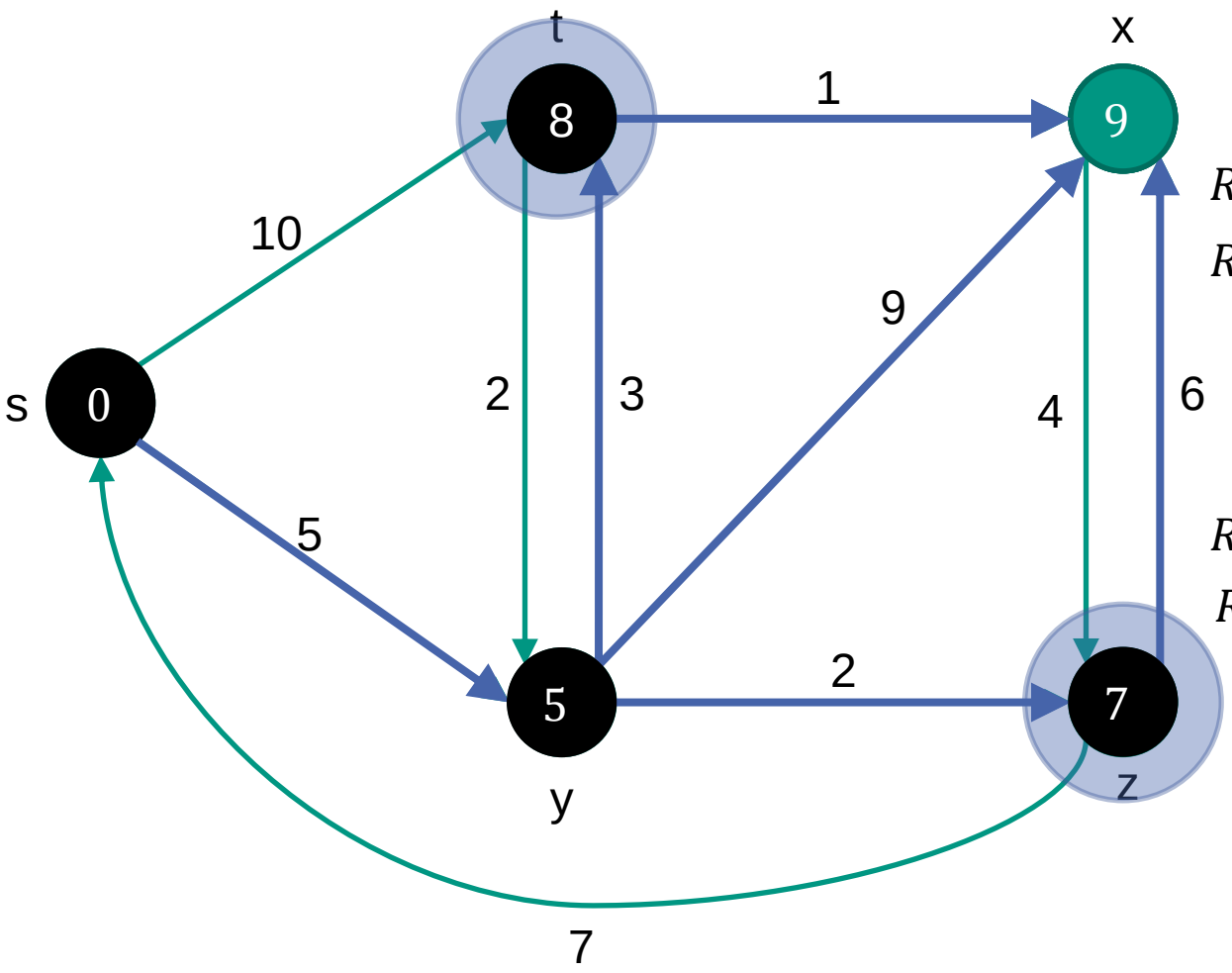
2. Iteration

$RELAX(y, t, 3) \rightarrow t.d = \mathbf{8}$

$RELAX(y, x, 9) \rightarrow x.d = \mathbf{14}$

$RELAX(y, z, 2) \rightarrow z.d = \mathbf{7}$

Dijkstra-Algorithmus – Beispiel



3. Iteration

$RELAX(z, s, 7) \rightarrow s.d = 0$

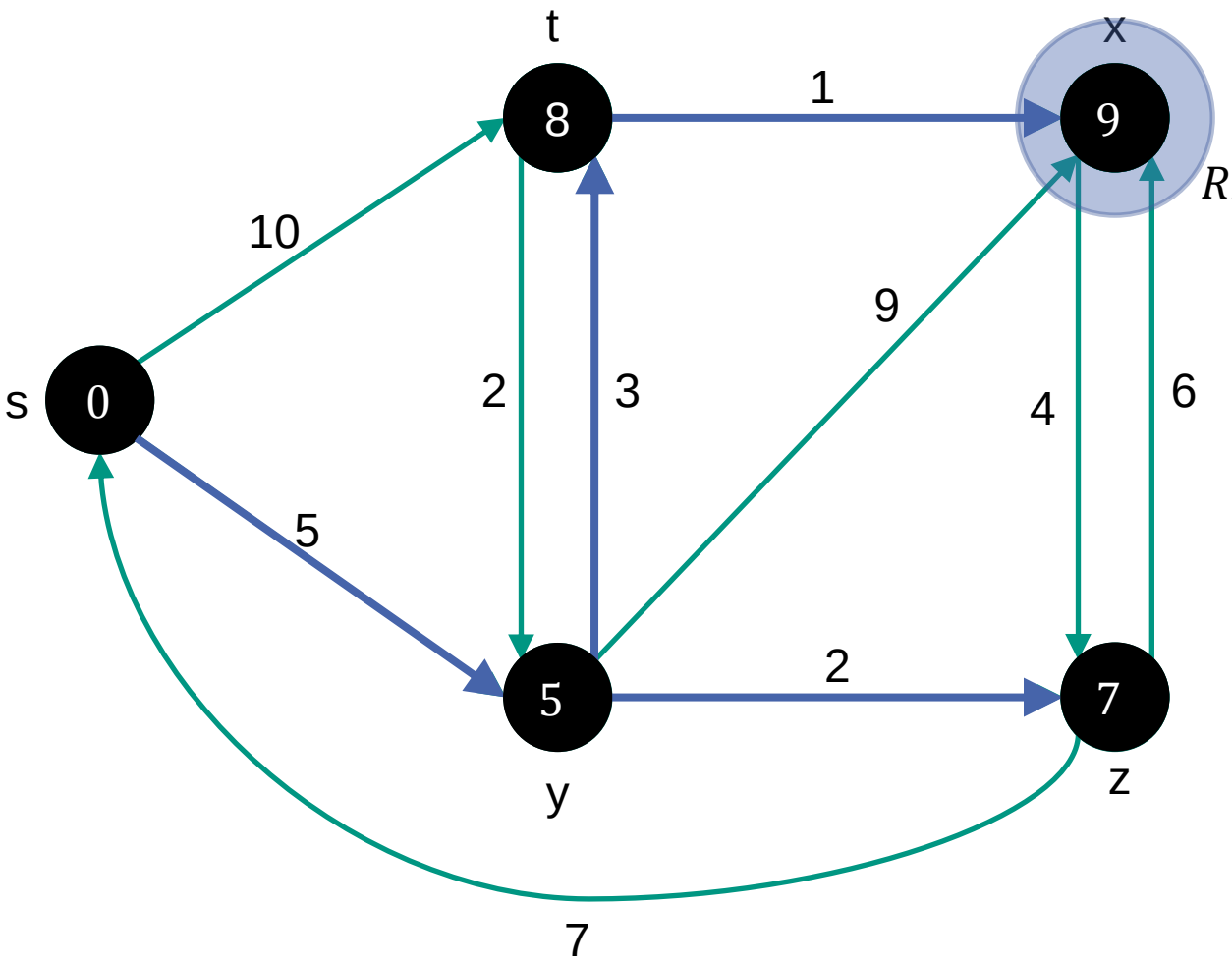
$RELAX(z, x, 6) \rightarrow x.d = \mathbf{13}$

4. Iteration

$RELAX(t, y, 2) \rightarrow y.d = 5$

$RELAX(t, x, 1) \rightarrow x.d = \mathbf{9}$

Dijkstra-Algorithmus – Beispiel



5. Iteration

$RELAX(x, z, 4) \rightarrow z.d = 7$

Dijkstra-Algorithmus – Analyse

- $DIJKSTRA(G, w, s)$

```
1    INITIALIZE_SINGLE_SOURCE(G, s)
2    S = ∅
3    Q = G.V
4    while Q ≠ ∅
5        u = EXTRACT_MIN(Q)
6        S = S ∪ {u}
7        foreach v ∈ G.Adj[u]
8            RELAX(u, v, w)
```

Einmal Einfügen in Prioritätswarteschlange pro Knoten

Einmal Entfernen aus Prioritätswarteschlange pro Knoten

Schleife läuft $|E|$-mal

Aktualisiert die Prioritätswarteschlange

Dijkstra-Algorithmus – Analyse

- Laufzeit des Dijkstra-Algorithmus hängt hauptsächlich von Implementierung der Prioritätswarteschlange ab

- Mittels eines binären Min-Heaps ergibt sich
  - Aufbau des Heaps in $O(|V|)$
  - $|V|$ $EXTRACT\_MIN$ Operationen mit Aufwand $O(\lg |V|)$
  - Maximal $|E|$ Aktualisierungen mit Aufwand $O(\lg |V|)$
  - Gesamtaufwand von $O((|V| + |E|) \lg |V|)$

- Mittels eines Fibonacci-Heaps ist eine Laufzeit von $O(|V| \lg|V| + |E|)$ möglich
  - Die Entwicklung von Fibonacci-Heaps war motiviert durch den Dijkstra-Algorithmus

Dijkstra-Algorithmus – Korrektheit

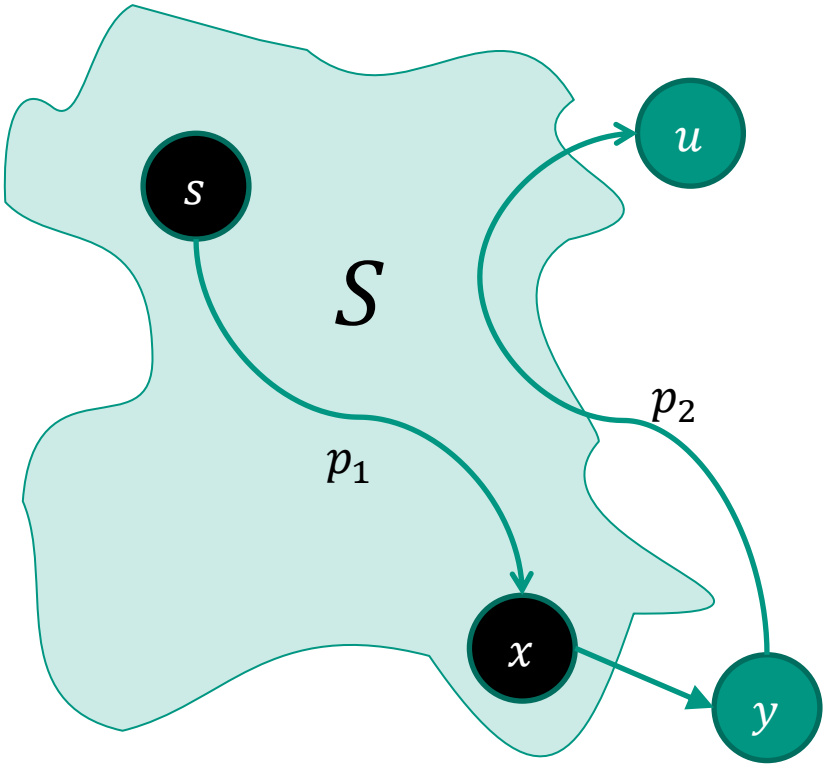
- Annahme: Eine Anwendung des Dijkstra-Algorithmus auf einen gewichteten, gerichteten Graphen $G = (V, E)$ mit nicht-negativen Gewichtungsfunktion w und Startknoten s terminiert mit $u.distance = \delta(s, u)$ für alle Knoten $u \in V$

- Beweis
  - Schleifeninvariante
    - Zu Beginn jeder Iteration der while-Schleife gilt $v.distance = \delta(s, v)$ für alle $v \in S$
  - Es genügt zu zeigen, dass für alle Knoten $u \in V$ $u.distance = \delta(s, u)$ zum Zeitpunkt des Einfügens in die Menge S gilt
    - $u.distance$ ändert sich danach nicht mehr (obere Schranke)
  - Initialisierung
    - Nach der Initialisierung ist $S = \emptyset$ und damit gilt die Invariante

Dijkstra-Algorithmus – Korrektheit

- Zu zeigen: In jeder Iteration gilt $u.distance = \delta(s,u)$ für den in S eingefügten Knoten
- Beweis durch Widerspruch
  - Annahme: Sei u der erste Knoten, für den zum Zeitpunkt des Einfügens in S $u.distance \neq \delta(s,u)$ gilt
  - → $u \neq s$, da durch Initialisierung $s.distance = \delta(s,s) = 0$ und s damit als erster Knoten korrekt in S eingefügt wird
  - → $S \neq \emptyset$ wenn u eingefügt wird, da mindestens s bereits in S ist
  - Aus $u.distance \neq \delta(s,u)$ folgt, dass ein Pfad von s nach u existiert, da sonst durch das „kein Pfad“ Kriterium $u.distance = \delta(s,u) = \infty$
  - → Es existiert ein kürzester Pfad p von s nach u

Dijkstra-Algorithmus – Korrektheit



- Bevor u zu S hinzugefügt wird, verbindet p einen Knoten in S (nämlich s) und einen Knoten in $V - S$ (nämlich u)
- Angenommen y ist der erste Knoten entlang p, der nicht in S ist und x der Vorgänger von y in S
- → $p = s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$
- Beachte: p_1 und p_2 könnten auch keine Kanten haben, also $s = x$ und $u = y$ gelten

Dijkstra-Algorithmus – Korrektheit

- Behauptung: $y.distance = \delta(s, y)$ zum Zeitpunkt, zu dem u in S eingefügt wird

- Beweis
  - Da u als erster Knoten $u.distance \neq \delta(s, u)$ beim Einfügen in S gewählt wurde
  - → $x.distance = \delta(s, x)$ als x zu S hinzugefügt wurde
  - Kante (x, y) wurde zu diesem Zeitpunkt relaxiert
  - Durch Konvergenzeigenschaft folgt die Behauptung

Dijkstra-Algorithmus – Korrektheit

- Da y vor u auf dem kürzesten Pfad von s nach u liegt und alle Kantengewichte nicht negativ sind gilt
  - $\delta(s,y) \leq \delta(s,u)$ und damit
  - $y.distance = \delta(s,y) \leq \delta(s,u) \leq u.distance$

- Da sowohl u als auch y noch nicht in S waren, als u ausgewählt wurde, muss $u.distance \leq y.distance$ gelten
  - → Obige Ungleichung wird zur Gleichung
  - $y.distance = \delta(s,y) = \delta(s,u) = u.distance$
  - Im Speziellen $u.distance = \delta(s,u)$
  - → Widerspruch zu „Sei u der erste Knoten, für den zum Zeitpunkt des Einfügens in S $u.distance \neq \delta(s,u)$ gilt“

- → $u.distance = \delta(s,u)$ zum Zeitpunkt, zu dem u zu S hinzugefügt wird und danach

Dijkstra-Algorithmus – Korrektheit

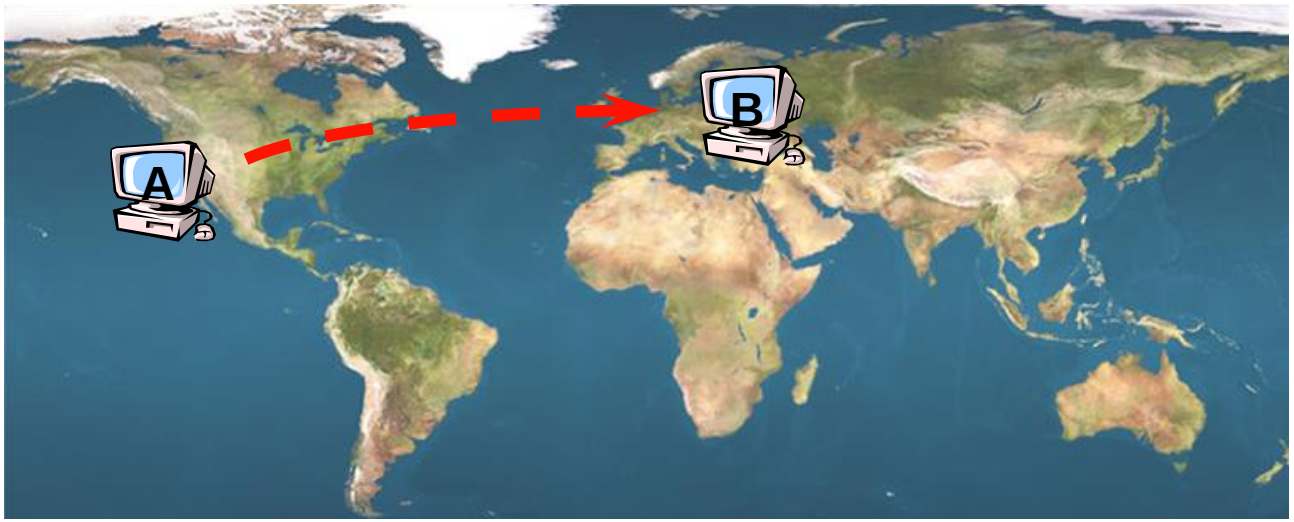
- Bleibt zu zeigen: Bei Abschluss des Algorithmus gilt $u.distance = \delta(s,u)$ für alle Knoten $u \in V$

- Beweis
  - Bei Abschluss ist $Q = \emptyset$
  - Da $Q = V - S$ gilt $S = V$
  - → $u.distance = \delta(s,u)$ für alle Knoten $u \in V$

q.e.d

9.6 Wegewahl im Internet

- Kürzeste Pfade im Internet
  - Endsystem A möchte Daten an Endsystem B senden



- Grundlegende Fragestellungen
  - Wie findet das Internet einen Weg zum Zielsystem?
    - Falls möglich einen „guten“ Weg / eine „gute“ Route
    - → Wegewahl / Routing

Route und Router

- Router
  - Zwischensystem, das Wegewahl-Funktionen für eingehende Dateneinheiten ausübt
  - Üblicherweise mit Schnittstellen zu mehreren anderen Systemen ausgestattet
  - Tauscht über Routingprotokolle Informationen mit anderen Routern aus

- Route
  - Hier: Weg einer Dateneinheit zum Ziel
    - Daher auch „Wege“wahl für die Auswahl der besten Route
  - In der Literatur unter wechselnden Bezeichnungen und Bedeutungen
    - Pfad, Weg, …

Routing

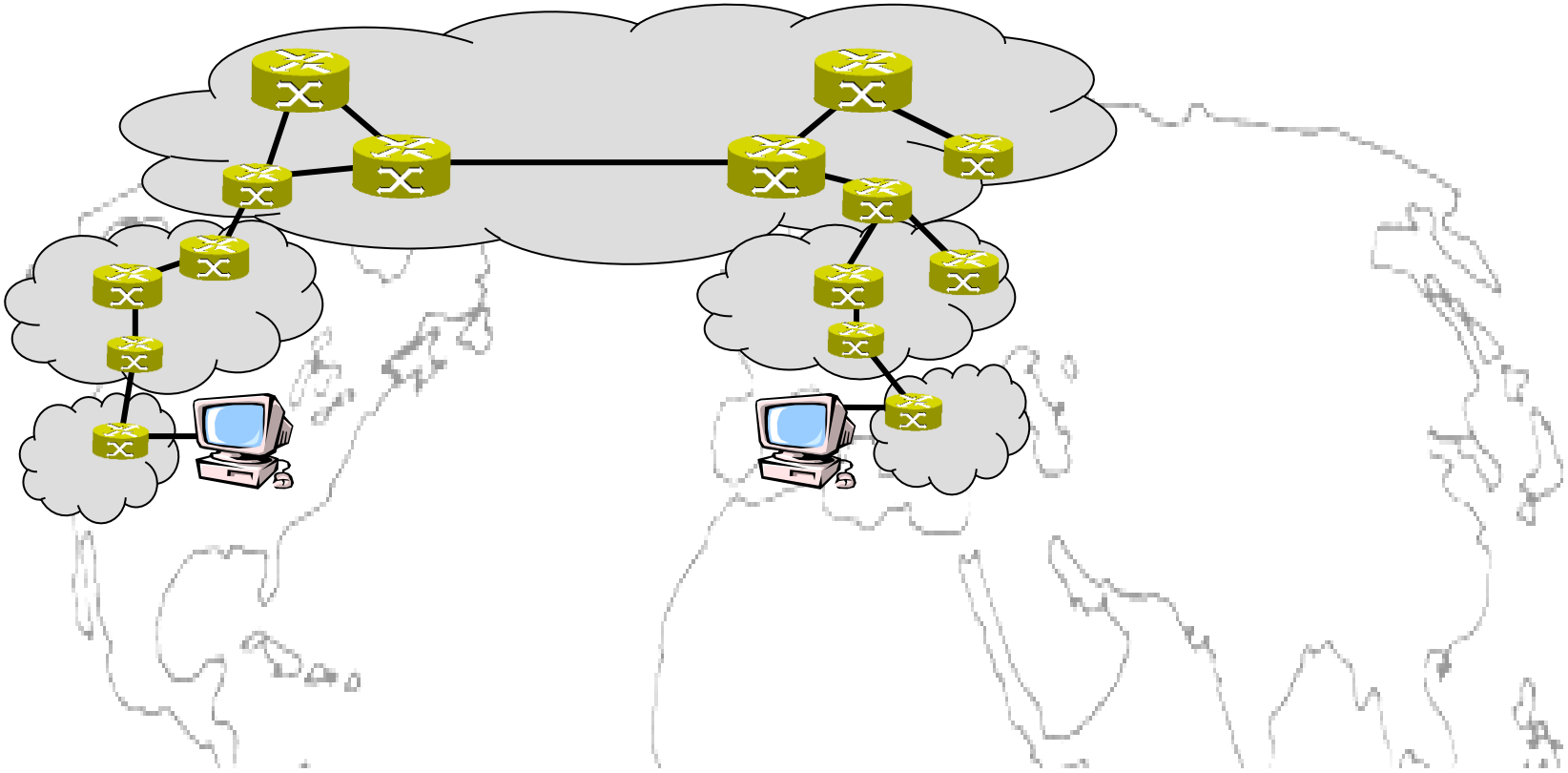
- Routing-Metrik
  - Bewertungskriterium einzelner Übertragungsabschnitte
    - Beeinflusst deren Bevorzugung/Vermeidung
    - Üblicherweise Ganzzahl, die unidirektional den Übertragungsabschnitten zu weiteren Zwischensystemen zugeordnet wird
      - Auch als Kosten oder Gewicht bezeichnet

- Modellierung
  - Netz wird als Graph modelliert
    - Router und Endsysteme sind die Knoten, die Verbindungen die Kanten
  - Den Kanten werden jeweils Kosten zugewiesen

- Aber
  - Graph ändert sich ständig

Autonome Systeme

- Internet gegliedert in sogenannte Autonome Systeme (AS)
  - Netzgraphen innerhalb eines AS
  - Netzgraphen zwischen ASen



Routingprotokolle

- OSPF (Open Shortest Path First)
  - Routingprotokoll innerhalb von ASen
    - Austausch von Information zwischen Routern
    - Jeder Router erhält so Information zum Netzgraphen seines AS
  - Jeder Router berechnet mittels Dijkstras Algorithmus den kürzesten Pfad zu allen anderen Routern in seinem AS
  - Eingehende Dateneinheiten werden dementsprechend weitergeleitet

- Offene Fragen
  - Wie bekommt ein Router den Netzgraph?
  - Was wenn sich der Netzgraph ändert?
  - Wie findet man Pfade über AS-Grenzen hinweg?

- Antworten hierzu in den Vorlesungen Einführung in Rechnernetze und Telematik

Zusammenfassung

- Wahl des Algorithmus hängt u.a. von Eingangsgraph ab

- Bellman-Ford Algorithmus
  - Auch bei negativen Kantengewichten
  - Laufzeit in $O(|V| * |E|)$

- DAG-Shortest-Paths
  - Für zyklenfreie Graphen
  - Laufzeit in $O(|V| + |E|)$

- Dijkstra Algorithmus
  - Keine negativen Kantengewichte
  - Laufzeit hängt von Implementierung der Prioritätswarteschlange ab
    - Bei guter Implementierung schneller als Bellman-Ford-Algorithmus
    - Mit Fibonacci-Heap $O(|V| \lg |V| + |E|)$ möglich

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 10 – Minimale Spannbäume

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

I. Einführung

1. Einführung

II. Suchen und Sortieren

2. Sortieren

III. Datenstrukturen

3. Folgen als Felder und Listen
4. Hashing
5. Heaps
6. Sortierte Listen / Bäume

IV. Graphenalgorithmen

7. Graphrepräsentation
8. Graphtraversierung
9. Kürzeste Wege
10. Minimale Spannbäume

- 10.1 Motivation
- 10.2 Aufbau eines Spannbaumes
- 10.3 Algorithmus von Kruskal
- 10.4 Algorithmus von Prim

V. Ausblick

11. Generische Optimierungsansätze
12. Zusammenfassung und Ausblick

10.1 Definition und Motivation

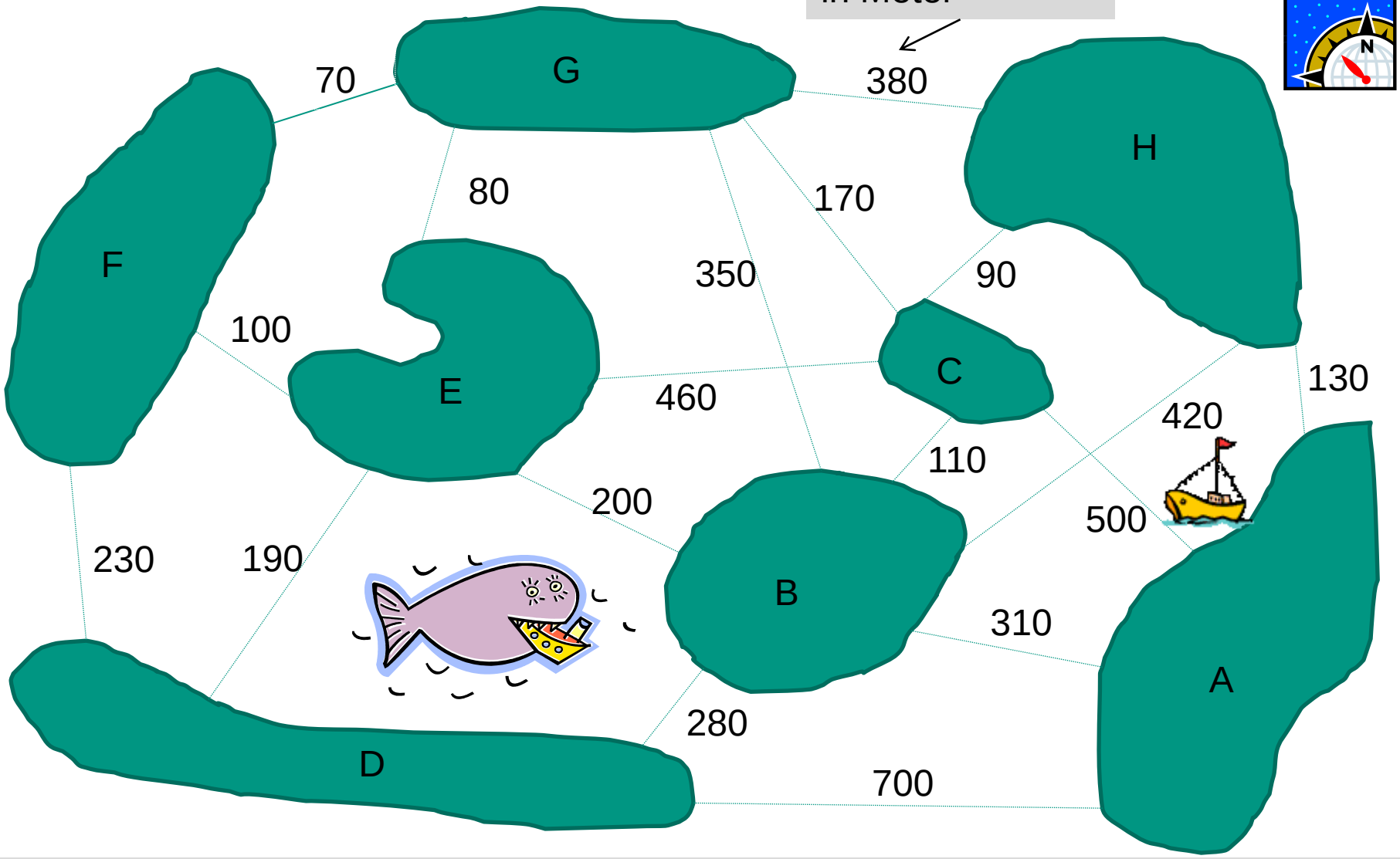
- Eingabeparameter für einen minimalen Spannbaum
  - Zusammenhängender, ungerichteter Graph $G = (V, E)$
  - Abbildung $\omega : E \rightarrow \mathbb{R}$, die jeder Kante eine Gewicht zuweist

- Gesucht: *Minimaler Spannbaum (MST)*
  - Ein *MST* ist eine azyklische Teilmenge $T \subseteq E$, die alle Knoten V verbindet und $\omega(T) = \sum_{(u,v)\epsilon T} \omega\,(u, v)$ minimiert

Beispiele für MSTs

- Beispiele für einen *MST*
  - Ein elektronischer Schaltkreis soll verschiedene Komponenten mit Kabeln verbinden, um sie auf das gleiche Potential zu legen
    - Hierfür verwendet man bei n Komponenten $n-1$ Anschlüsse
    - Gewünscht ist eine Konfiguration, die die verwendete Länge an Kabel aus Kostengründen minimiert
  - Computer sollen verkabelt werden
    - Leitungslänge soll so minimal wie möglich sein
  - Näherungslösungen für schwere Probleme
    - Handlungsreisendenproblem, Steinerbaumproblem

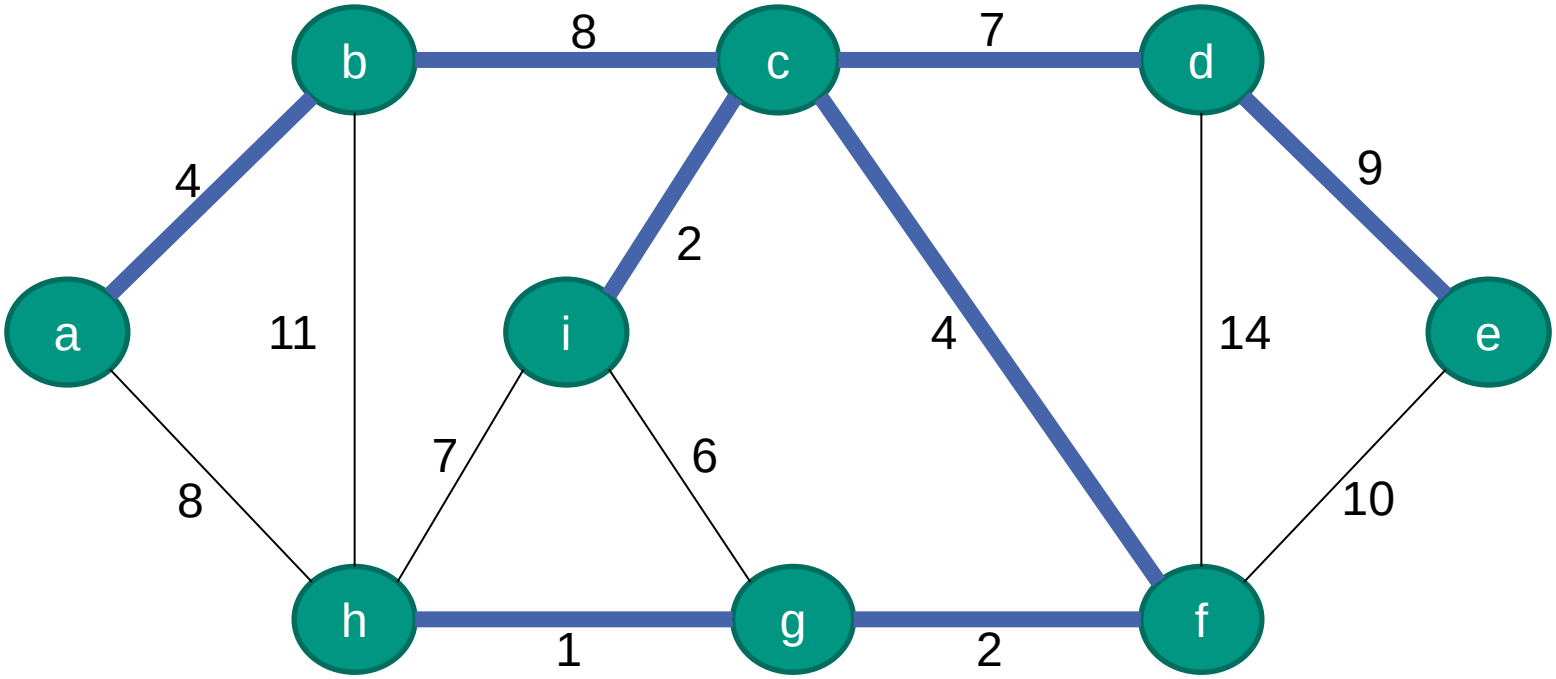
Beispiel Brücken bauen



10.1.2 Beispiel

Gegeben: Knoten *V*

Gegeben: Kanten *E*



Gegeben: Gewichte *w(u,v)*

Gesucht: *MST*

Gewicht des MST ist 37

10.2 Aufbau eines minimalen Spannbaums

- Problem: Wie bestimmt man nun *algorithmisch* einen MST ?
  - In diesem Kapitel werden zwei Algorithmen vorgestellt
  - Beide Algorithmen basieren auf Greedy-Ansatz
    - Greedy Algorithmen treffen jeweils diejenige Entscheidung, die im Moment am optimalsten erscheint
      - Lokal optimale Entscheidung
      - Greedy Algorithmen liefern deshalb häufig nicht optimale Ergebnisse
- Zunächst: *Generischer Algorithmus* zur Bestimmung eines MST
  - Darauf aufbauend zwei konkrete Algorithmen

10.2.1 Generischer Ansatz

- Gegeben
  - Zusammenhängender, ungerichteter Graph $G = (V, E)$
  - Gewichtungsfunktion $\omega : E \longrightarrow \boldsymbol{R}$

- Gesucht
  - Minimaler Spannbaum von Graph G
    - Genauer: Kantenmenge A des minimalen Spannbaums, also eine azyklische Teilmenge $T \subseteq E$, die alle Knoten V verbindet und $\omega(T) = \sum_{(u,v)\epsilon T} \omega(u, v)$ minimiert

- Vorgehen
  - Für Kantenmenge A gilt folgende Schleifeninvariante
    - Vor jeder Iteration ist A eine Teilmenge des minimalen Spannbaums
  - In jedem Schritt wird eine Kante (u, v) bestimmt, die wir zu A hinzufügen können ohne die Invariante zu verletzen
    - Es gilt: $(u, v) \cup A$ ist ebenfalls Teilmenge des *MST*
    - Kante (u, v) wird als **sichere Kante** bezeichnet

10.2.2 Pseudocode

- $GENERIC - MST(G, \omega)$

```
1 A = ∅
2 while A ist kein Spannbaum
3       finde sichere Kante (u, v) für A
4       A = A ∪ {(u, v)}
5 return A
```

Starte mit leerer Kantenmenge

Solange noch nicht alle Knoten verbunden sind

Füge eine sichere Kante zu A hinzu

A bildet einen MST

10.2.3 Beweis der Schleifeninvariante

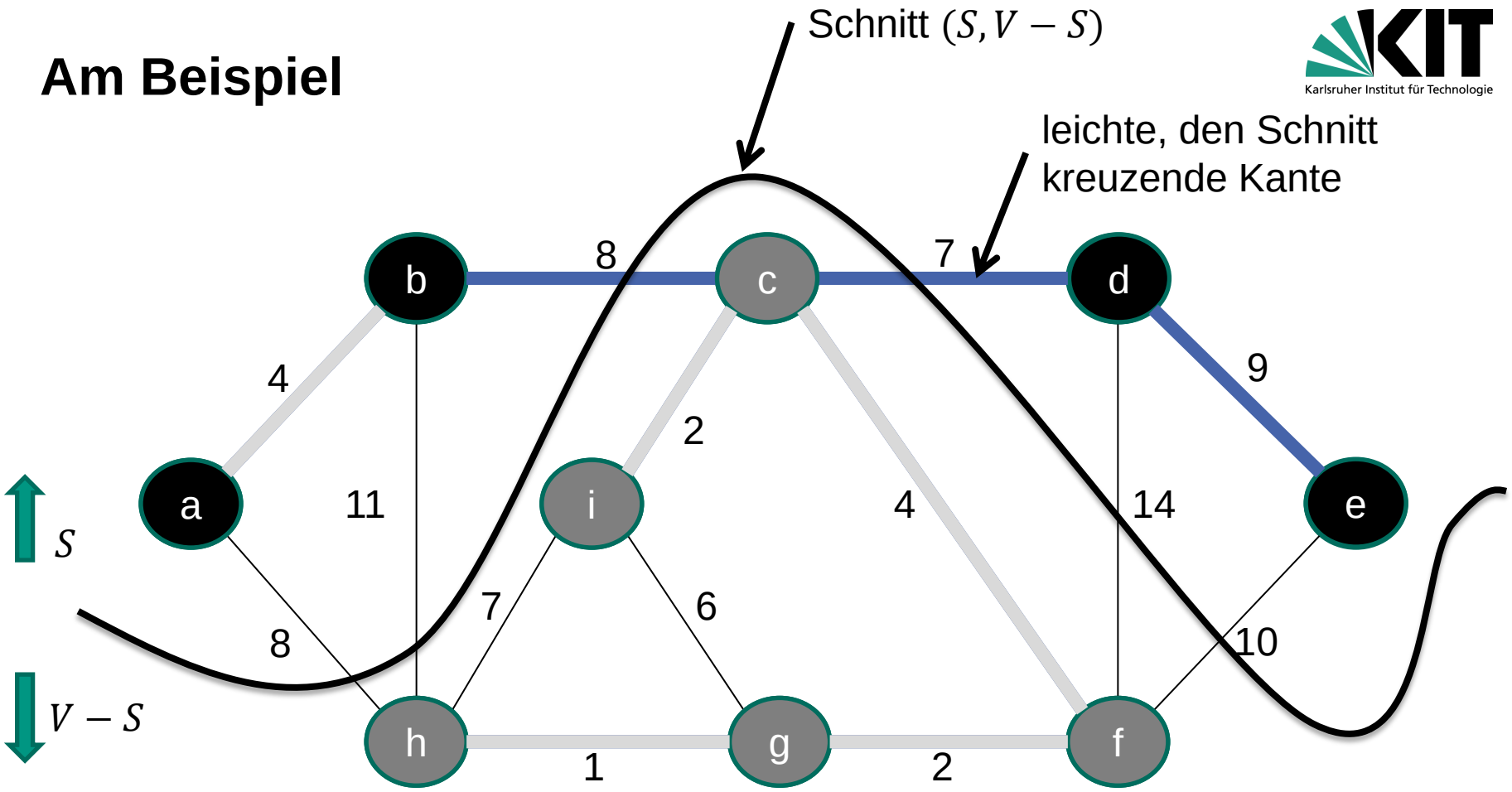
- Beweis der Schleifeninvariante in $GENERIC - MST$
  - Initialisierung
    - Nach der ersten Zeile ist die Schleifeninvariante erfüllt
  - Fortsetzung
    - Zeile 2-4 erhalten die Schleifeninvariante, da nur sichere Kanten hinzugefügt werden
  - Terminierung
    - Alle Kanten, die zu A hinzugefügt wurden, befinden sich in einem MST
    - Daher wird in Zeile 5 ein MST zurückgegeben

- Problem gelöst?
  - Wie bestimmt man algorithmisch eine sichere Kante ?
  - Die Algorithmen von *Kruskal* und *Prim* lösen dieses Problem

10.2.4 Hilfsdefinitionen

- Ein **Schnitt** $(S, V - S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Partitionierung von V
- Eine Kante $(u, v) \epsilon E$ **kreuzt** den Schnitt $(S, V - S)$, falls einer ihrer **Endpunkte** in S und der Andere in $V - S$ liegt
- Ein Schnitt $(S, V - S)$ **respektiert** eine Kantenmenge A, falls keine Kante von A den Schnitt kreuzt
- Eine Kante $(u, v) \epsilon E$ ist eine **leichte**, den Schnitt **kreuzende** Kante, falls sie das kleinste Gewicht aller Kanten hat, die den Schnitt kreuzen

Am Beispiel



Knoten in S enthalten

Knoten in $V-S$ enthalten

Schnitt $(S, V-S)$ respektiert A

10.2.5 Theorem zur Bestimmung sicherer Kanten

- Sei $G = (V, E)$ zusammenhängender, ungerichteter Graph mit Gewichtsfunktion $\omega : E \longrightarrow \boldsymbol{R}$
- Seien
  - A eine Teilmenge von E, die in einem minimalen Spannbaum von G enthalten ist.
  - $(S, V - S)$ ein beliebiger Schnitt von G, der A respektiert
  - (u, v) eine leichte Kante, die $(S, V - S)$ kreuzt

    → Dann ist die Kante (u, v) sicher für A

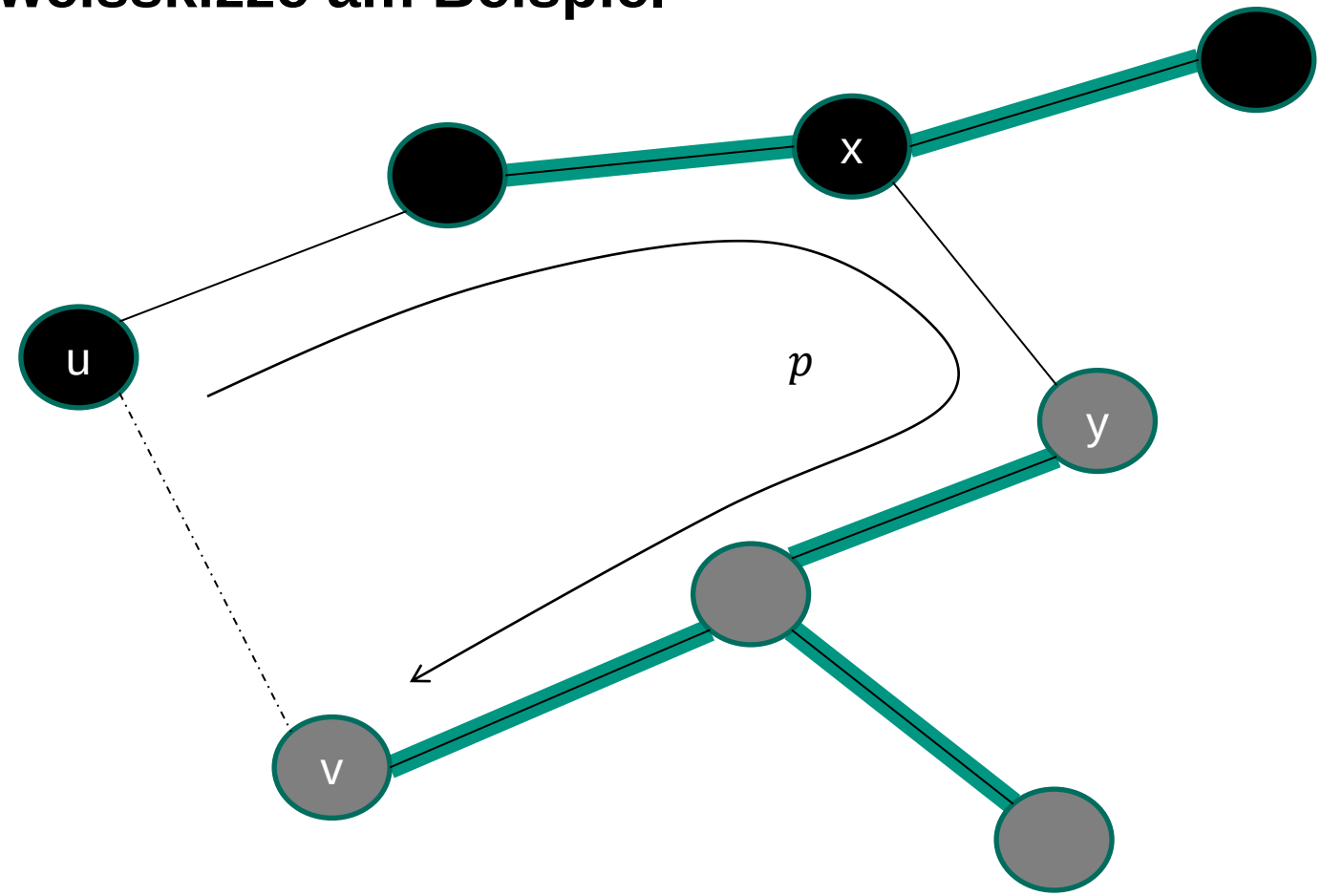
- Beweisskizze
  - Annahmen
    - Sei T ein minimaler Spannbaum, der die Kantenmenge A enthält
    - T enthält die leichte Kante (u, v) nicht
  - Beweisziel
    - Konstruiere einen anderen minimalen Spannbaum T', der $A \cup \{(u, v)\}$ enthält

      → Damit ist (u, v) eine sichere Kante

Beweisskizze

- Konstruiere Spannbaum T', der $A \cup \{(u,v)\}$ enthält
- Sei p ein Pfad in T von u nach v
  - Dann bildet p mit (u,v) einen Zyklus
- Da (u,v) eine Kante ist, die den Schnitt $(S, V-S)$ kreuzt, existiert eine Kante in T auf dem eindeutigen Pfad p, die auch den Schnitt kreuzt
  - Dies sei (x,y)
  - (x,y) gehört nicht zu A, da der Schnitt A respektiert
  - Entfernt man (x,y) aus T, zerfällt T in zwei Komponenten
  - Fügt man (u,v) zu den beiden Komponenten hinzu, erhält man einen neuen Spannbaum $T' = T - \{(x,y)\} \cup \{(u,v)\}$

Beweisskizze am Beispiel



Beweisskizze

- Was fehlt noch ?
  - T' muss minimaler Spannbaum sein
  - (u, v) muss sichere Kante sein

- T' ist ein minimaler Spannbaum
  - (u, v) ist leichte, kreuzende Kante, (x, y) kreuzt den Schnitt ebenfalls
    - $\omega(u, v) \leq \omega(x, y)$
    - $\omega(T') = \omega(T) - \omega(x, y) + \omega(u, v) \leq \omega(T)$
    - Da $\omega(T) \leq \omega(T')$ gelten muss, ist T' auch minimaler Spannbaum

- (u, v) ist sichere Kante für A
  - Da $A \subseteq T$ und $(x, y) \notin A$ gilt $A \subseteq T'$
  - Also auch $A \cup \{(u, v)\} \subseteq T'$
  - Da T' minimaler Spannbaum, folgt (u, v) ist sichere Kante

q.e.d

10.2.5 Korollar zu Theorem

- Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, für den Gewichtsfunktion $\omega: E \longrightarrow \boldsymbol{R}$ auf E definiert ist

- Seien
  - A eine Teilmenge von E, die in einem minimalen Spannbaum von G enthalten ist
  - $C = (V_c, E_c)$ ein Baum aus dem Wald $G_A = (V, A)$

- Falls (u, v) eine leichte Kante ist, die C mit einer anderen Komponente von G_A verbindet, dann ist (u, v) für A sicher

- Beweis
  - Der Schnitt $(V_c, V - V_c)$ respektiert A und (u, v) ist eine leichte Kante für den Schnitt

    → Daraus folgt, dass (u, v) sicher ist für A

q.e.d

10.3 Algorithmus von Kruskal

- Eingabeparameter für einen minimalen Spannbaum
  - Zusammenhängender, ungerichteter Graph $G = (V, E)$
  - Abbildung $w : E \rightarrow \mathbb{R}$, die jeder Kante eine Gewicht zuweist

- Gesucht: *Minimaler Spannbaum (MST)*

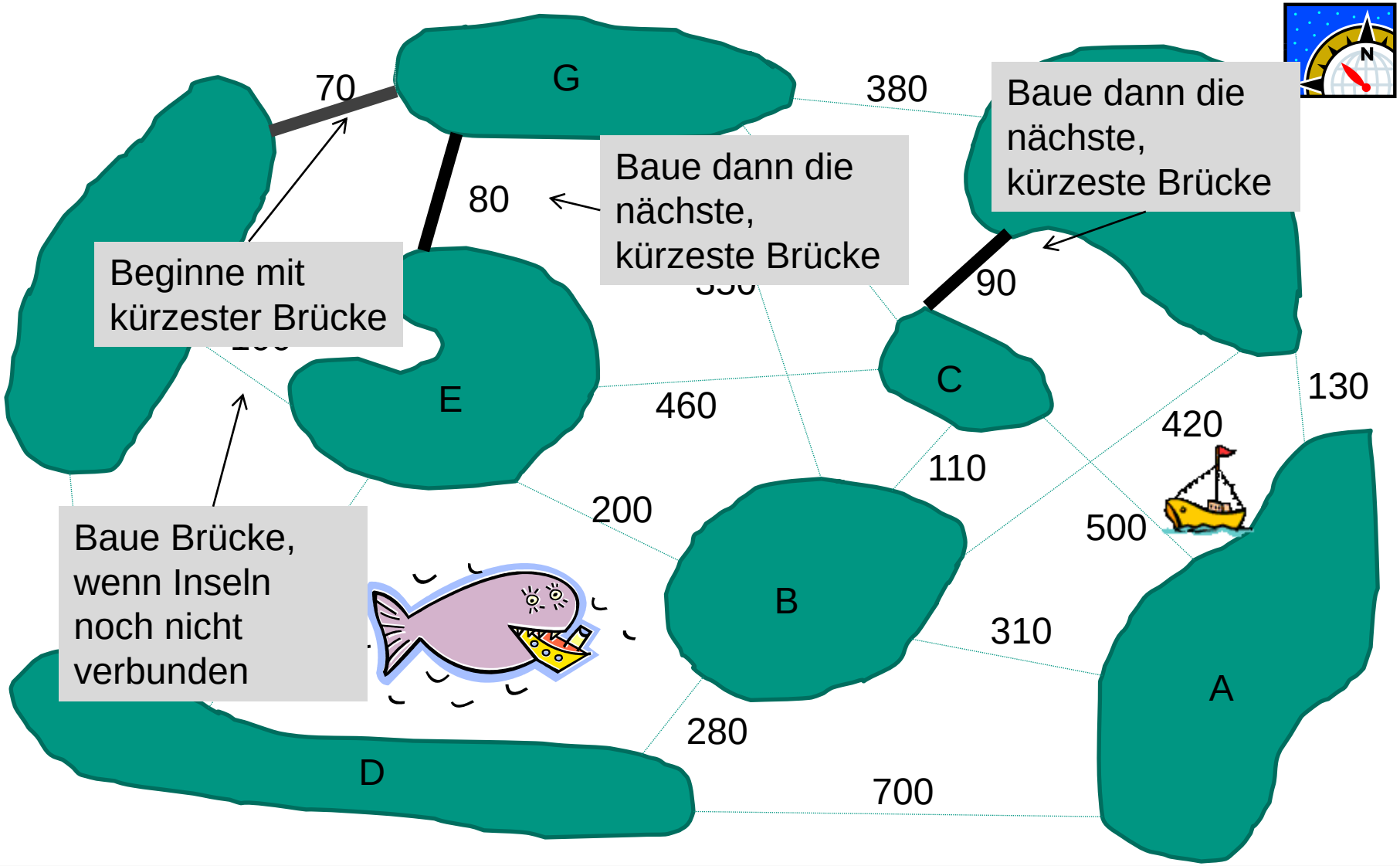
- Kantenmenge A ist ein Wald
  - Die zu A hinzugefügte Kante
    - ist eine Kante mit minimalem Gewicht
    - verbindet zwei Komponenten

Algorithmus von Kruskal

- Ablauf
  - Führe folgende Schritte solange aus, bis alle Knoten mit Kanten verbunden sind
    - Wähle eine kürzeste Kante die zwei Knoten verbindet, die bisher nicht durch Kanten verbunden waren
    - Füge diese Kante zum MST hinzu

- Beweis, dass die ausgewählte Kante eine sichere Kante ist folgt aus Korollar
  - Seien C_1 und C_2 die zwei Bäume des Waldes, die durch (u, v) verbunden sind
  - Da (u, v) eine leichte Kante sein muss, die C_1 mit einem anderen Baum verbindet, ist (u, v) eine sichere Kante für C_1

Brücken bauen mit Kruskal



10.3.2 Pseudocode

- MST-Kruskal(G, ω)

```
1 A = ∅
2 for jeden Knoten v ∈ G.V
3       MAKE − SET(v)
4 sortiere Kanten aus G.E in nichtfallender Reihenfolge nach ω
5 for jede Kante (u, v) ∈ G.E, in nichtfallender Reihenfolge nach Gewichten
6       if FIND − SET(u) ≠ FIND − SET(v)
7             A = A ∪ {(u, v)}
8             UNION(u, v)
9 return A
```

$MAKE - SET$, $FIND - SET$ und $UNION$ seien Operationen auf disjunkten Mengen um disjunkte Mengen zu Erzeugen ($MAKE - SET$), zu Vereinigen ($UNION$) und Zeiger auf Repräsentanten der Menge zurückzugeben ($FIND - SET$)

Exkurs: Datenstrukturen disjunkter Mengen

- Viele Algorithmen operieren mit disjunkten Mengen
- Speziell sind zwei Operationen häufig auszuführen
  - Ein ausgesuchtes Elementen einer Menge bestimmen
    - Beispielsweise das kleinste Element auswählen
  - Zwei Mengen vereinigen
    - Beispielsweise zwei Knotenmengen vereinigen

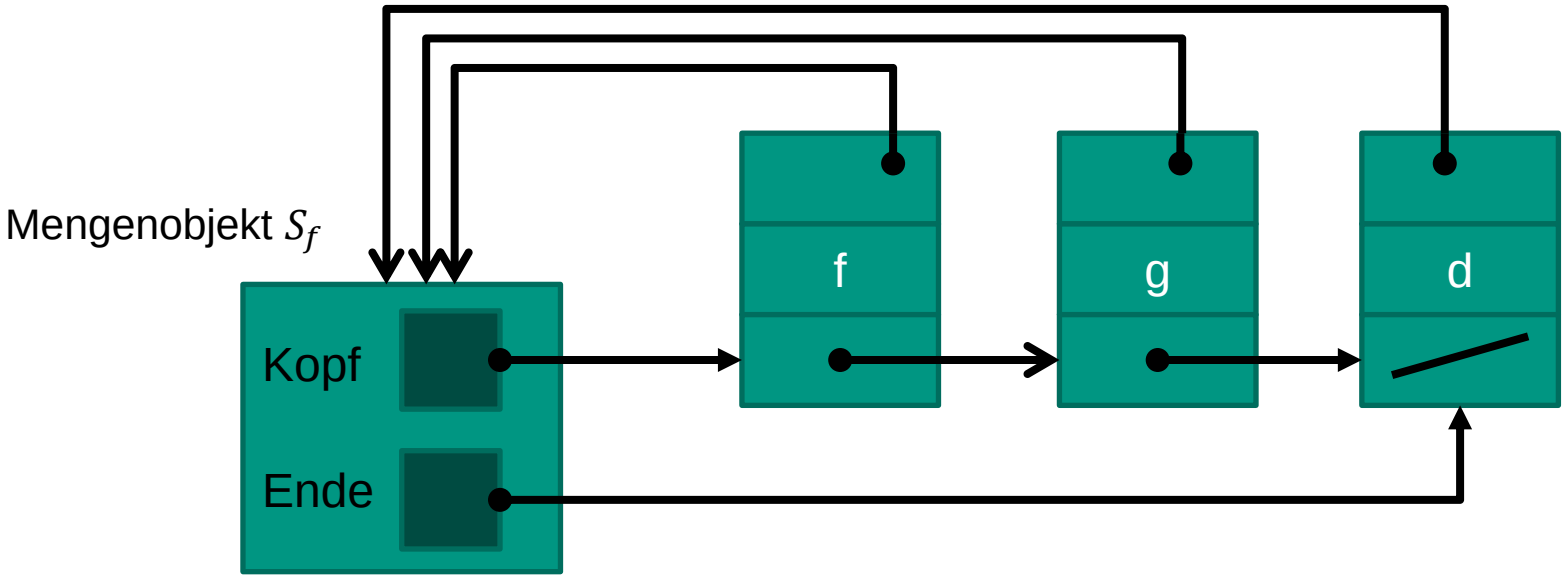
Operationen auf disjunkten Mengen

- Datenstruktur disjunkter Mengen verwaltet Ensemble $S = \{S_1, S_2, S_3 \dots S_k\}$ von disjunkten dynamischen Mengen
- Jede Menge durch einen Repräsentanten gekennzeichnet
  - Repräsentant ist beliebiges Element der Menge
  - Bei mehrmaliger Abfrage einer Menge: gleicher Repräsentant
- Jedes Element der Menge wird als Objekt dargestellt

- $MAKE - SET(x)$
  - Erzeugt Menge, deren einziges Element *x* ist
  - Da die Mengen disjunkt sind, darf *x* in keiner anderen Menge vorhanden sein
- $UNION(x, y)$
  - Vereinigt zwei disjunkte Mengen S_x und S_y , die die Objekte *x* und *y* enthalten zu neuer Menge $S_x \cup S_y$
  - Repräsentant von $S_x \cup S_y$ ist beliebiges Element der Menge
  - S_x und S_y werden nach dem Vereinigen „zerstört“
- $FIND - SET(x)$
  - Gibt Zeiger auf Repräsentanten der Menge zurück, die *x* enthält

Darstellung mithilfe verketteter Listen

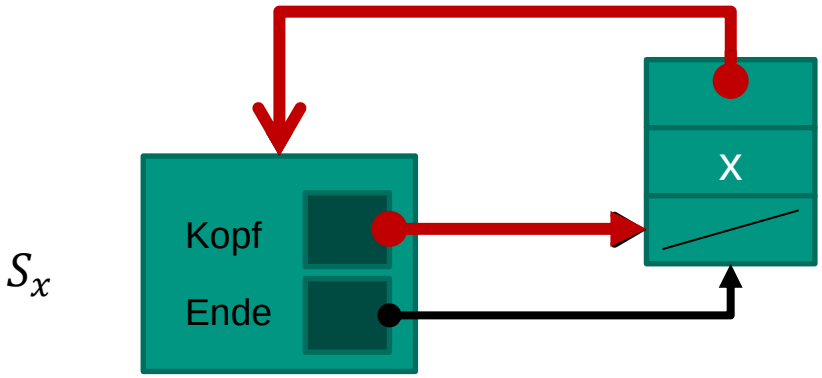
- Beispiel



- Jede Menge wird ausgehend von einem Mengenobjekt als verkette Liste dargestellt (hier Menge S_f)
- Jedes Objekt einer Liste enthält Element der Menge, Zeiger auf das nächste Objekt und Zeiger auf das Mengenobjekt

Umsetzung der Operationen

- $MAKE - SET(x)$
  - Neue, verkette Liste mit dem Element x erzeugt
  - Aufwand hierfür $O(1)$

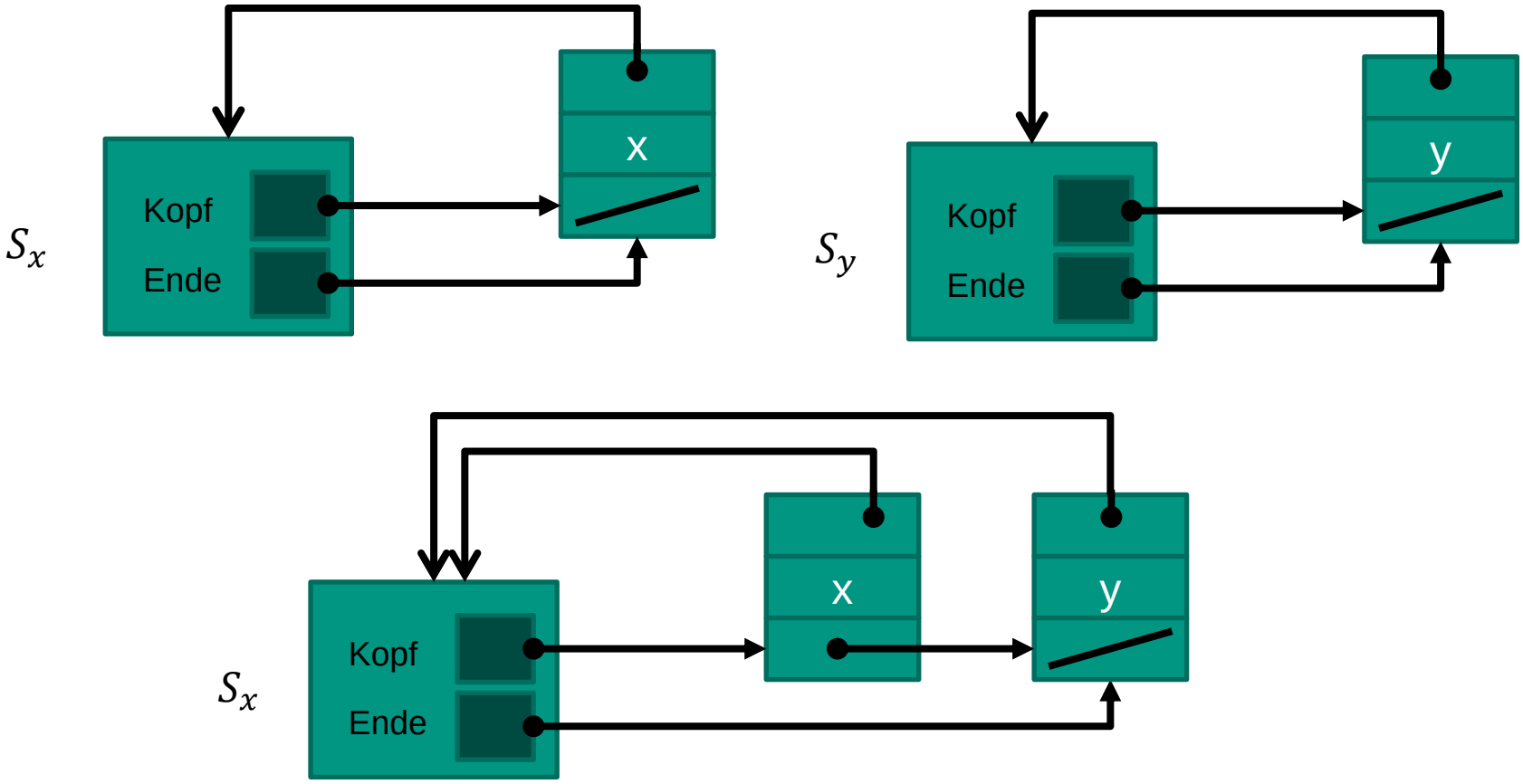


- $FIND - SET(x)$
  - Liefert Repräsentant der Menge die x enthält
  - Repräsentant finden
    - Zeiger von *x* auf das Mengenobjekt folgen
    - Element ausgeben, auf das Zeiger *Kopf* zeigt
    - Aufwand also $O(1)$

Umsetzung der Operationen

- UNION(x,y)
  - Liste *y* wird an das Ende der Liste *x* angehängt
  - Repräsentant der Liste x wird Repräsentant der resultierenden Liste
  - Der Zeiger *Ende* wird verwendet, um das Ende der Liste *x* zu finden
  - Alle Zeiger in der angehängten Liste *y* aktualisieren
    - Also wird eine zur Länge der Liste y lineare Zeit benötigt
  - Zeiger *Ende* der Liste x aktualisieren
  - Danach kann die ursprüngliche Liste y gelöscht werden
  - Aufwand ?
    - Folgt mit Hilfe amortisierter Analyse

Beispiel UNION(x,y)



Amortisierte Analyse der Darstellung mit verketteten Listen

- Wir konstruieren eine Folge von *m* Operationen auf *n* Objekten, die die Zeit $\theta(n^2)$ Zeit benötigt
  - Auf den Objekten $x_1, x_2, \ldots x_n$ führen wir n $MAKE-SET$ Operationen, gefolgt von $n-1$ $UNION$ Operationen aus

Operation	Anzahl der aktualisierten Objekte
$MAKE-SET(x_1)$	1
$MAKE-SET(x_2)$	1
….	….
$MAKE-SET(x_n)$	1
$UNION(x_2, x_1)$	1
$UNION(x_3, x_2)$	2
$UNION(x_4, x_3)$	3
…	….
$UNION(x_n, x_{n-1})$	$n-1$

Amortisierte Analyse der Darstellung mit verketteten Listen

- Wir konstruieren eine Folge von *m* Operationen auf *n* Objekten, die die Zeit $\theta(n^2)$ Zeit benötigt
  - Es werden also insgesamt $m = 2n - 1$ Operationen ausgeführt
  - Für die n $MAKE - SET$ Operationen benötigen wir $\theta(n)$ Zeit
  - Die $i - te$ $UNION$ Operation aktualisiert i Objekte
  - Für $n - 1$ $UNION$ Operation ergibt sich als Anzahl der aktualisierten Objekte

$$\sum_{i=1}^{n-1} i = \theta(n^2)$$

  - Da insgesamt $2n - 1$ Operationen ausgeführt werden, ergibt sich als amortisierte Laufzeit einer Operation

$$\theta(n)$$

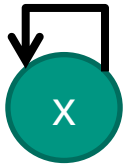
Darstellung mithilfe von Bäumen

- $MAKE - SET(x)$ erzeugt Baum mit einem Knoten
- $FIND - SET(x)$ folgt den Vaterzeigern bis zur Wurzel des Baumes
- $UNION(x, y)$ lässt die Wurzel des einen Baumes auf die Wurzel des anderen zeigen

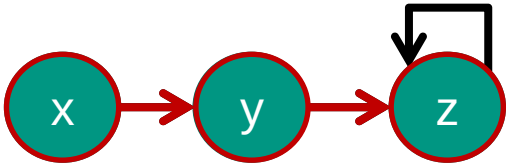
- Mit Hilfe von Heuristiken zur Laufzeitverbesserungen kann erreicht werden, dass die Laufzeit im schlechtesten Fall für *m* $MAKE - SET$, $FIND - SET$ und $UNION$ Operationen, von denen *n* Operationen $MAKE - SET$ Operationen sind, $O(m * \propto (n))$ ist
  - $\propto (n)$ sei hierbei eine sehr langsam wachsende Funktion für die $\propto (n) < 4$ gilt

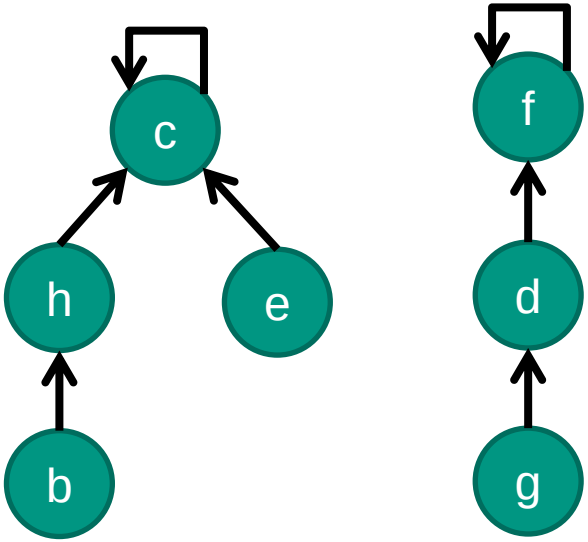
Darstellung mithilfe von Bäumen

$MAKE - SET(x)$

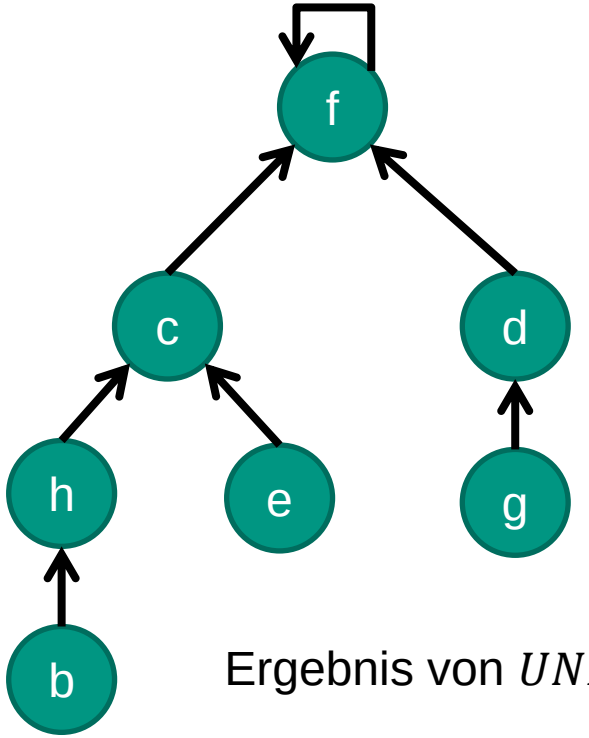


$FIND - SET(x)$ gibt einen Zeiger auf Element z zurück





Mengen S_c und S_f vor der Vereinigung



Ergebnis von $UNION(f, c)$

10.3.2 Pseudocode

- MST-Kruskal(G, ω)

```
1 A = ∅
2 for jeden Knoten v ∈ G.V
3       MAKE − SET(v)
4 sortiere Kanten aus G.E in nichtfallender Reih
5 for jede Kante (u, v) ∈ G.E, in nichtfallender Reihenfolge nach Gewichten
6       if FIND − SET(u) ≠ FIND − SET(v)
7             A = A ∪ {(u, v)}
8             UNION(u, v)
9 return A
```

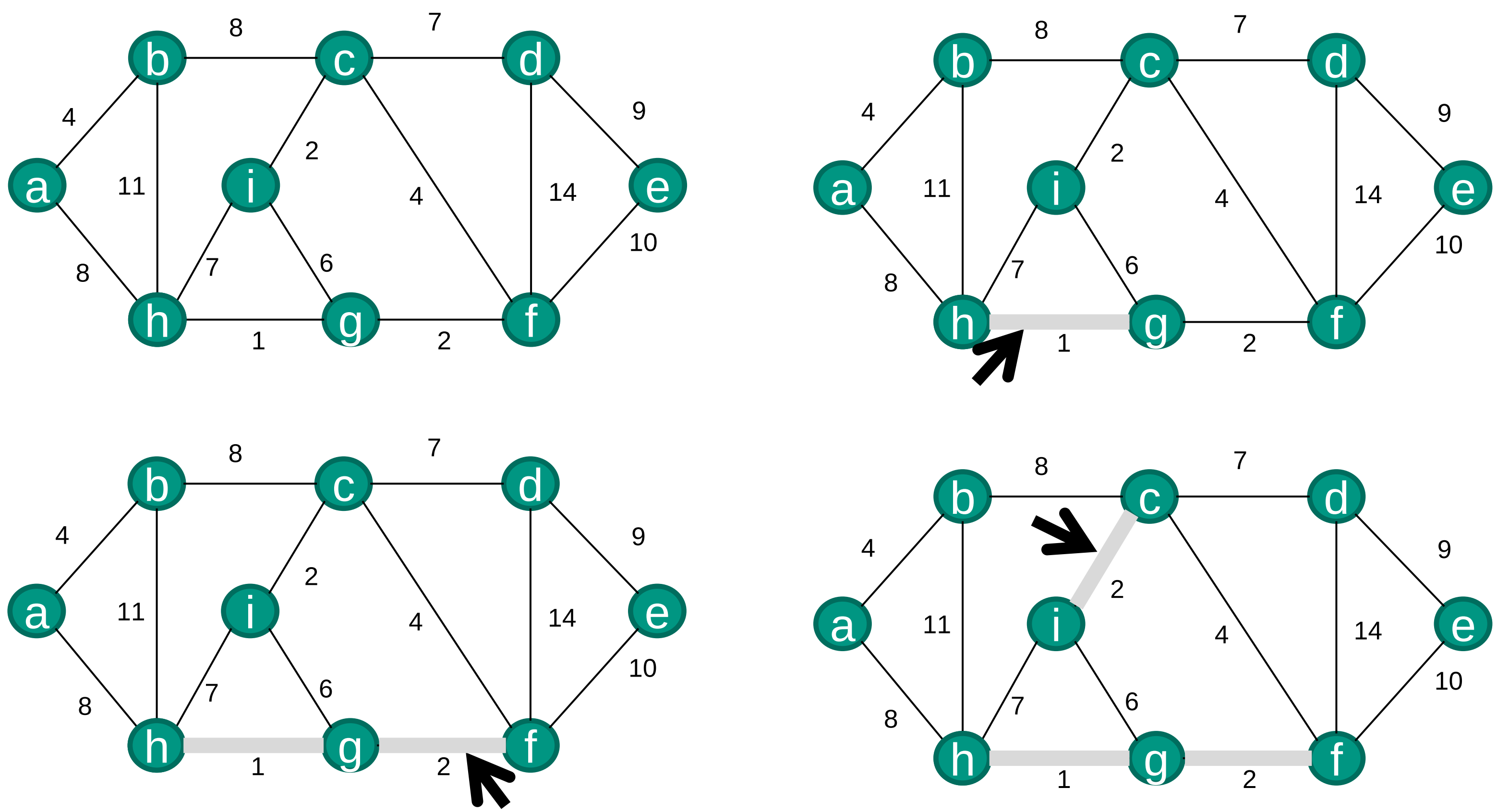
Starte mit leerer Kantenmenge

Erzeuge |V| Bäume die je einen Knoten enthalten

Durchlaufe Kanten in nichtfallender Reihenfolge nach ihren Gewichten. Gehören die Kanten zu unterschiedlichen Bäumen, füge Kante hinzu und verschmelze Bäume

A bildet einen MST

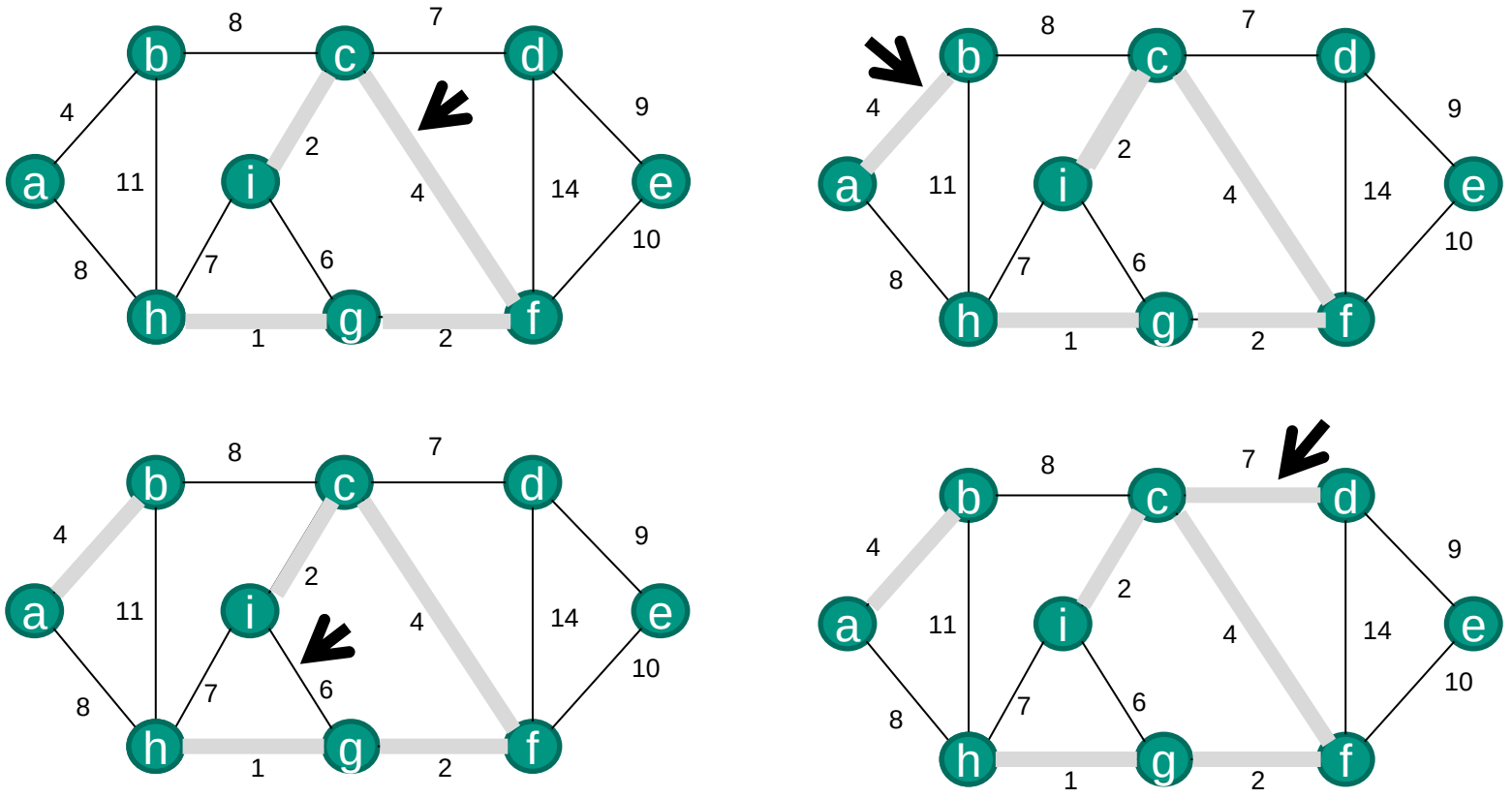
10.3.3 Beispiel



Teil des Spannbaums

Im nächsten Schritt betrachtete Kante

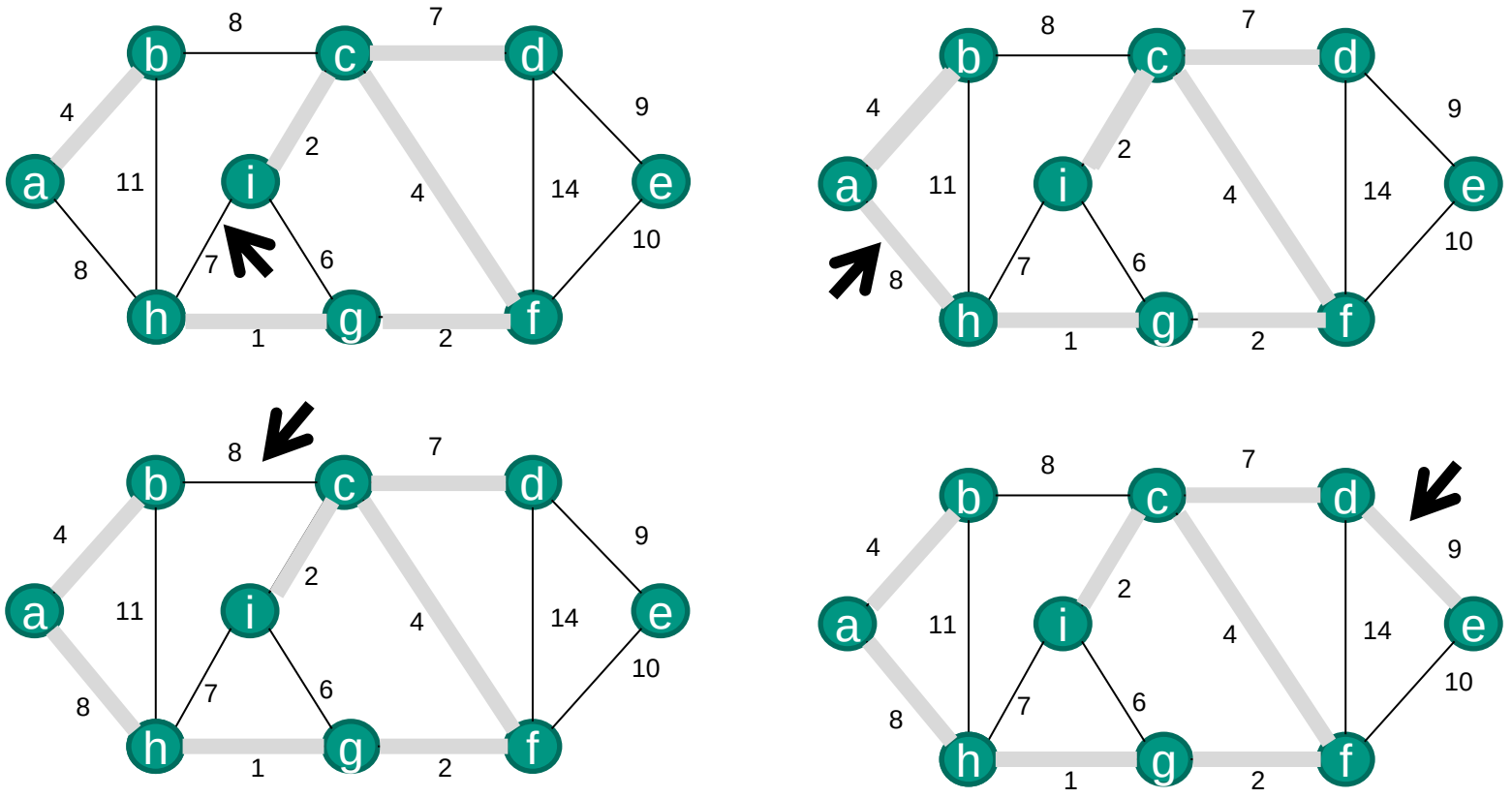
10.3.3 Beispiel



Teil des Spannbaums

Im nächsten Schritt betrachtete Kante

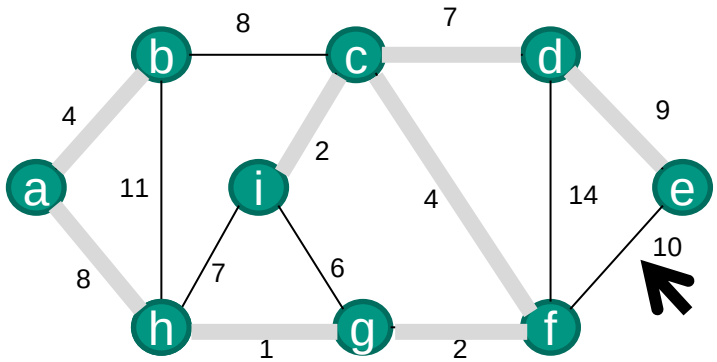
10.3.3 Beispiel

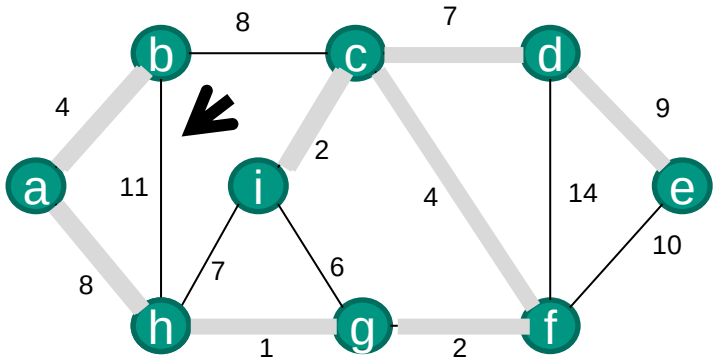


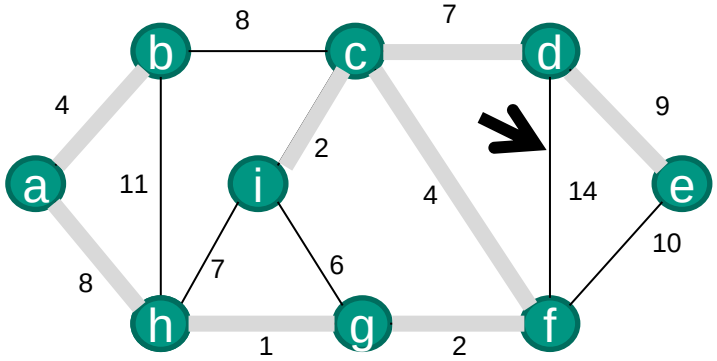
Teil des Spannbaums

Im nächsten Schritt betrachtete Kante

10.3.3 Beispiel







Teil des Spannbaums

Im nächsten Schritt betrachtete Kante

10.3.4 Komplexität

- MST-Kruskal(G, ω)

1 $A = \emptyset$
2 **for** jeden Knoten $v \in G.V$
3 $\quad MAKE - SET(v)$
4 sortiere Kanten aus $G.E$ in nichtfallender Reihenfolge nach ω
5 **for** jede Kante $(u, v) \in G.E$, in nichtfallender Reihenfolge nach Gewichten
6 $\quad$ **if** $FIND - SET(u) \neq FIND - SET(v)$
7 $\quad\quad A = A \cup \{(u, v)\}$
8 $\quad\quad UNION(u, v)$
9 **return** A

$\theta(1)$

$|V|$ mal $MAKE - SET$

$\theta(E \lg E)$

$O(E)$ mal $FIND - SET$ und $UNION$

Wir nehmen an, wir verwenden die schnellsten derzeitig bekannten Implementierungen für Operation auf disjunkten Mengen

Komplexität

- Komplexität der Operationen auf disjunkten Mengen insgesamt

$$O((V+E) \propto (V))$$

  - $\propto$ sei hierbei eine sehr langsam wachsende Funktion
    - Zur genauen Definition von $\propto$ siehe Cormen, Abschnitt 21.4
  - Da *G* zusammenhängend ist, gilt $|E| \geq |V| - 1$
    - Insgesamt benötigen wir also $O(E \propto (V))$ als Laufzeit

- Gesamtlaufzeit von Kruskal
  - Es gilt $\propto (|V|) = O(\lg V) = O(\lg E)$
  - Daraus folgt Laufzeit liegt in $O(E\ lg\ E)$
    - Da $|E| < |V^2|$ gilt

⟹ Gesamtlaufzeit $O(E \lg V)$

10.4 Algorithmus von Prim

- Eingabeparameter für einen minimalen Spannbaum
  - Zusammenhängender, ungerichteter Graph $G = (V, E)$
  - Abbildung $w : E \rightarrow \mathbb{R}$, die jeder Kante eine Gewicht zuweist

- Gesucht: *Minimaler Spannbaum (MST)*

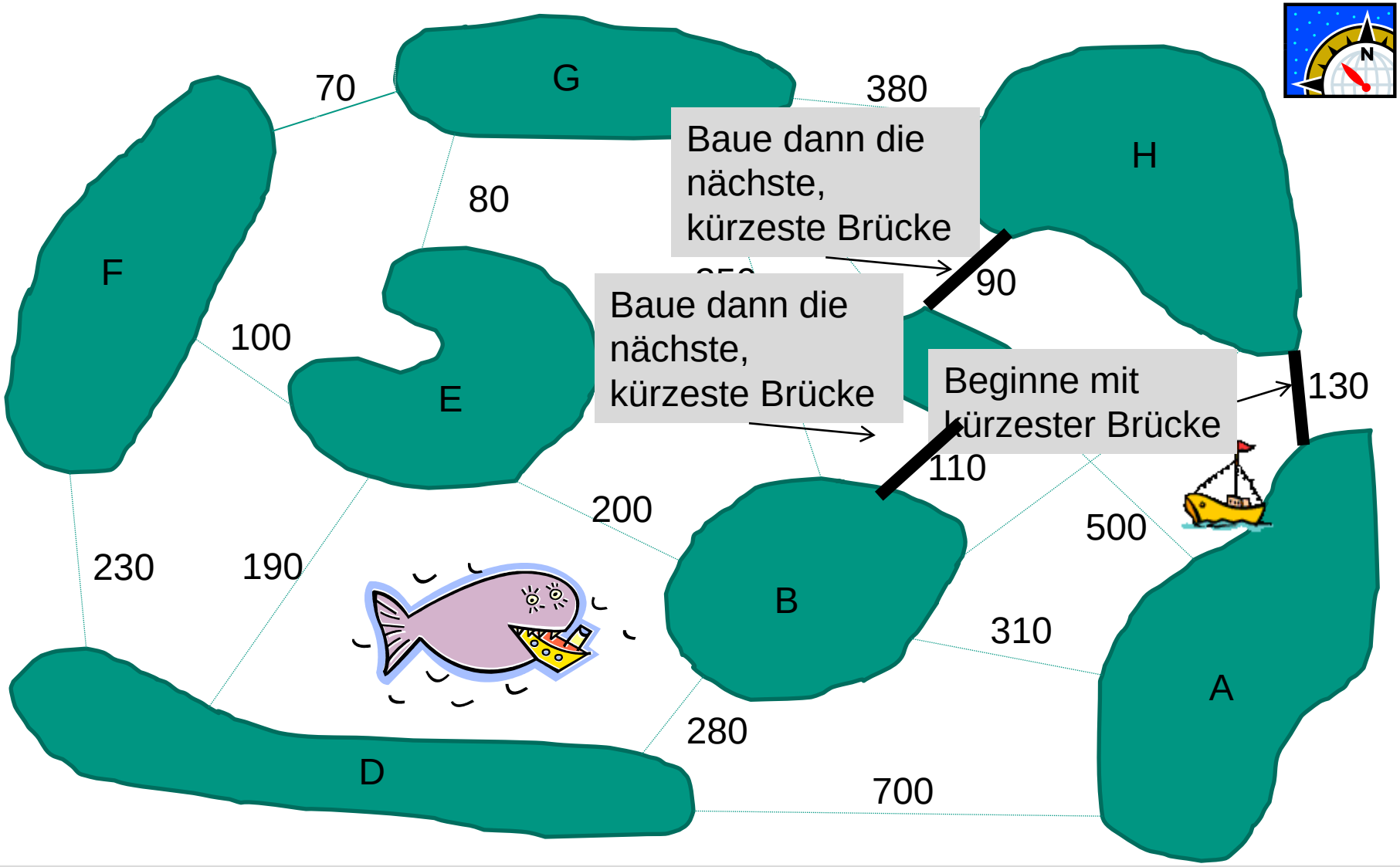
- Kantenmenge *A* bildet jeweils einen Baum
- Die zum Baum *A* hinzugefügte Kante
  - ist eine leichte Kante
  - verbindet einen isolierten Knoten aus $G_A = (V, A)$ mit A
    - also einen Knoten, zu dem bisher keine Kante führt

Algorithmus von Prim

- Ablauf
  - Wähle einen Knoten aus und nenne ihn „erreichbar“
    - Alle anderen Knoten sind nicht erreichbar
  - Führe den folgenden Schritt so oft aus, bis alle Knoten erreichbar sind
    - Wähle eine kürzeste Kante zwischen zwei Knoten, von denen einer erreichbar und der andere nicht erreichbar ist
    - Nenne den bislang nicht erreichbaren Knoten erreichbar

- Beweis, dass die ausgewählte Kante eine sichere Kante ist folgt aus Korollar

Brücken bauen mit Prim



10.4.1 Pseudocode

- MST-Prim(G, ω, r)

```
1 for alle u ϵ G.V
2       u.schlüssel = ∞
3       u.π = NIL
4 r.schlüssel = 0
5 Q = G.V
6 while Q ≠ ∅
7       u = EXTRACT − MIN(Q)
8       for alle v ϵ G.Adj[u]
9             if v ϵ Q und ω(u, v) < v.schlüs
10                  v.π = u
11                  v.schlüssel = ω(u, v)
```

r ist die Wurzel des Baumes

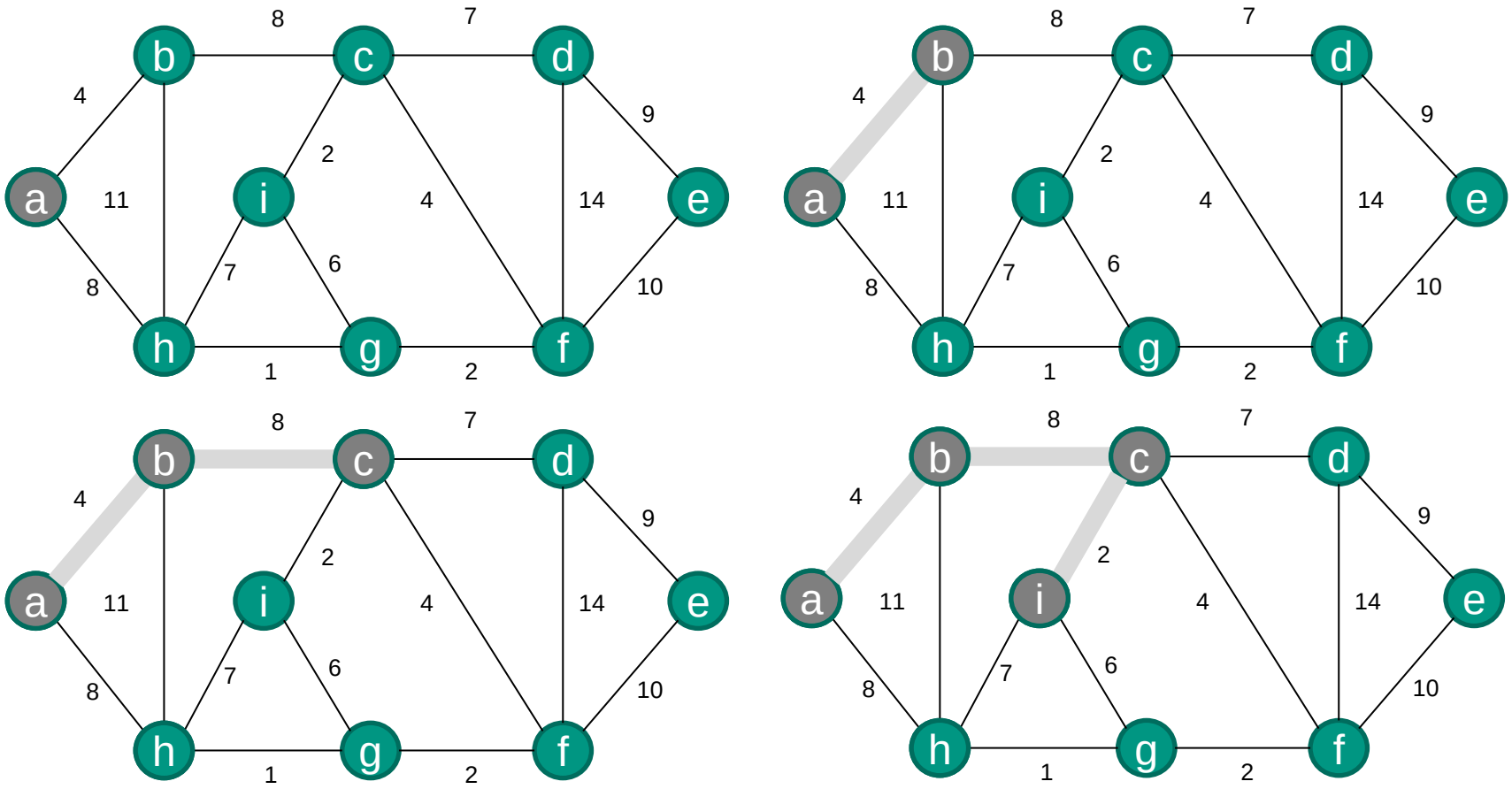
π ist der Vater des Knotens

$schlüssel$ bezeichnet das kleinste Gewicht aller Kanten, die den Knoten mit dem Baum verbinden

Q ist eine Min-Prioritätswarteschlange, die auf $schlüssel$ basiert

Betrachte alle Kanten in der Adjazenzliste

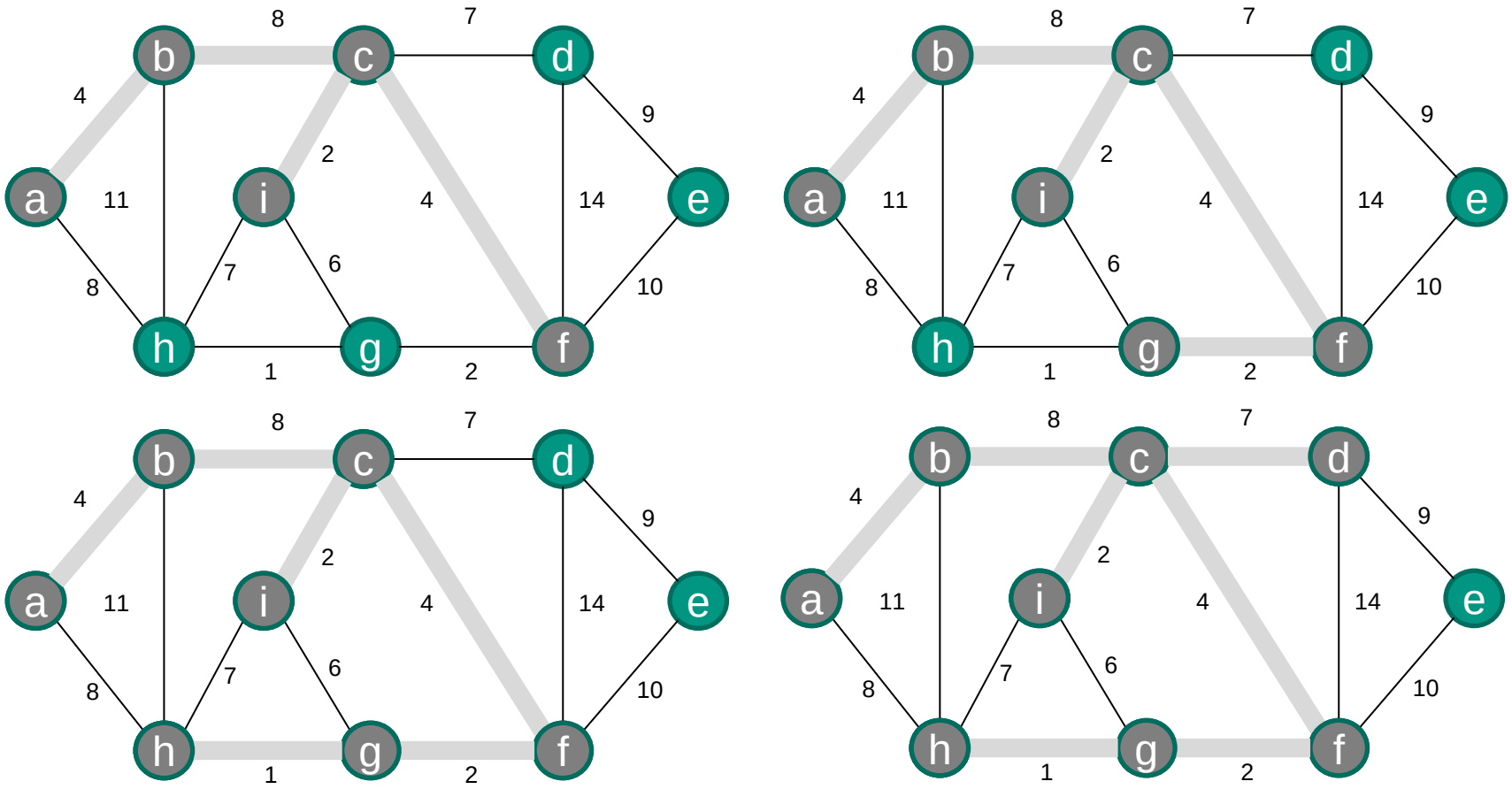
10.4.2 Beispiel



Teil des Spannbaums

Im nächsten Schritt betrachteter Knoten

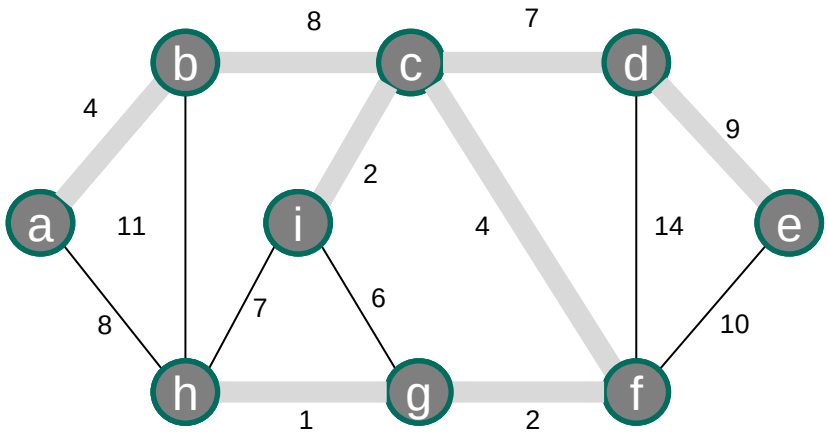
10.4.2 Beispiel



Teil des Spannbaums

a Im nächsten Schritt betrachteter Knoten

10.4.2 Beispiel



Teil des Spannbaums

a Im nächsten Schritt betrachteter Knoten

10.4.3 Komplexität

- MST-Prim(G, ω, r)

```
1  for alle u ϵ G.V
2        u.schlüssel = ∞
3        u.π = NIL
4  r.schlüssel = 0
5  Q = G.V
6  while Q ≠ ∅
7        u = EXTRACT − MIN(Q)
8        for alle v ϵ G.Adj[u]
9              if v ϵ Q und ω(u,v) < v.schlüssel
10                   v.π = u
11                   v.schlüssel = ω(u,v)
```

(Wenn Q als binärer MIN-HEAP implementiert ist)

BUILD-MIN-HEAP
$O(V)$

|V|-mal ausgeführt

$O(\lg V)$

$O(E)$-mal ausgeführt

Implizites
$DECREASE - KEY$
$O(lgV)$

Gesamtlaufzeit $\boldsymbol{O(VlgV + ElgV) = O(ElgV)}$

10.5 Vergleich der beiden Algorithmen

- Pro Prim
  - Asymptotisch gut für alle $|E|, |V|$
  - Sehr schnell für $|E| >> |V|$
- Pro Kruskal
  - Gut für $|E| = O(|V|)$
  - Es wird nur eine Kantenliste benötigt
  - Profitiert von schnellem Suchen
  - Verfeinerungen auch gut für große $\frac{|E|}{|V|}$

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3

Karlsruher Institut für Technologie

Vorlesung Algorithmen I
Kapitel 11 – Generische Optimierungsansätze

Prof. Dr. Martina Zitterbart, Dr. Ingmar Baumgart, Sören Finster, Christian Haas
[zit, baumgart, finster, haas]@tm.uka.de

Institut für Telematik, Prof. Zitterbart

Aufbau der Vorlesung

11.1 Motivation

- Beim Entwurf von Algorithmen kommt es häufig zu einem Optimierungsproblem
  - Ein Problem hat viele mögliche Lösungen
  - Jede Lösung besitzt einen bestimmten Wert
  - Gesucht ist möglichst optimale Lösung, beispielsweise minimaler Wert

- Für solche Probleme und deren Lösung werden zwei generische Optimierungsstrategien vorgestellt
  - Dynamische Programmierung
  - Greedy-Algorithmen

11.2 Dynamische Programmierung

- Dynamische Programmierung typischerweise eingesetzt für die Lösung von Optimierungsproblemen
  - Also möglichst optimale Lösung im Lösungsraum suchen

- Grundgedanke der dynamischen Programmierung
  - Dynamische Programmierung löst Probleme, indem Lösungen von Teilproblemen kombiniert werden
  - Also dynamische Programmierung = Teile-und-Herrsche ?
    - Wdh.: Teile-und-Herrsche zerlegt ein Problem in disjunkte Teilprobleme und löst diese rekursiv
  - Nein!
    - Dynamische Programmierung wird angewendet, wenn sich Teilprobleme überlappen
    - Teilprobleme lösen ihrerseits wieder die gleichen Teilprobleme
    - Beim dynamischen Programmieren wird jedes Teilprobleme genau einmal gelöst und Lösung zur Wiederverwendung gespeichert

Elemente der dynamische Programmierung

- Bei der Lösung eines Problems mit einem dynamischen Algorithmus werden folgende Schritte durchgeführt
  - Charakterisieren der Struktur einer optimalen Lösung
  - Rekursive Definition des Werts einer optimalen Lösung
  - Wert einer optimalen Lösung mit Hilfe eines bottom-up-Ansatz berechnen
  - Konstruktion einer zugehörigen optimalen Lösung aus berechnetem optimalen Wert
    - Wenn nur Wert einer optimalen Lösung von Interesse ist, kann der letzte Schritt entfallen

11.2.1 Dynamische Programmierung am Beispiel

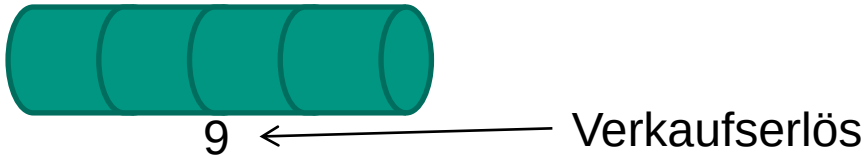
- Beispiel: Stabzerlegungsproblem
  - Gegeben:
    - Stange der Länge n cm
    - Preisliste p_i für $i = 1,2,\dots,n$
      - Gibt Preis eines Stabes der Länge *i* an
  - Gesucht: Maximaler Erlös r_n
    - Hierfür Zerlegung in kleine Stäbe möglich

- Beispielhafte Preisliste für $n = 10$

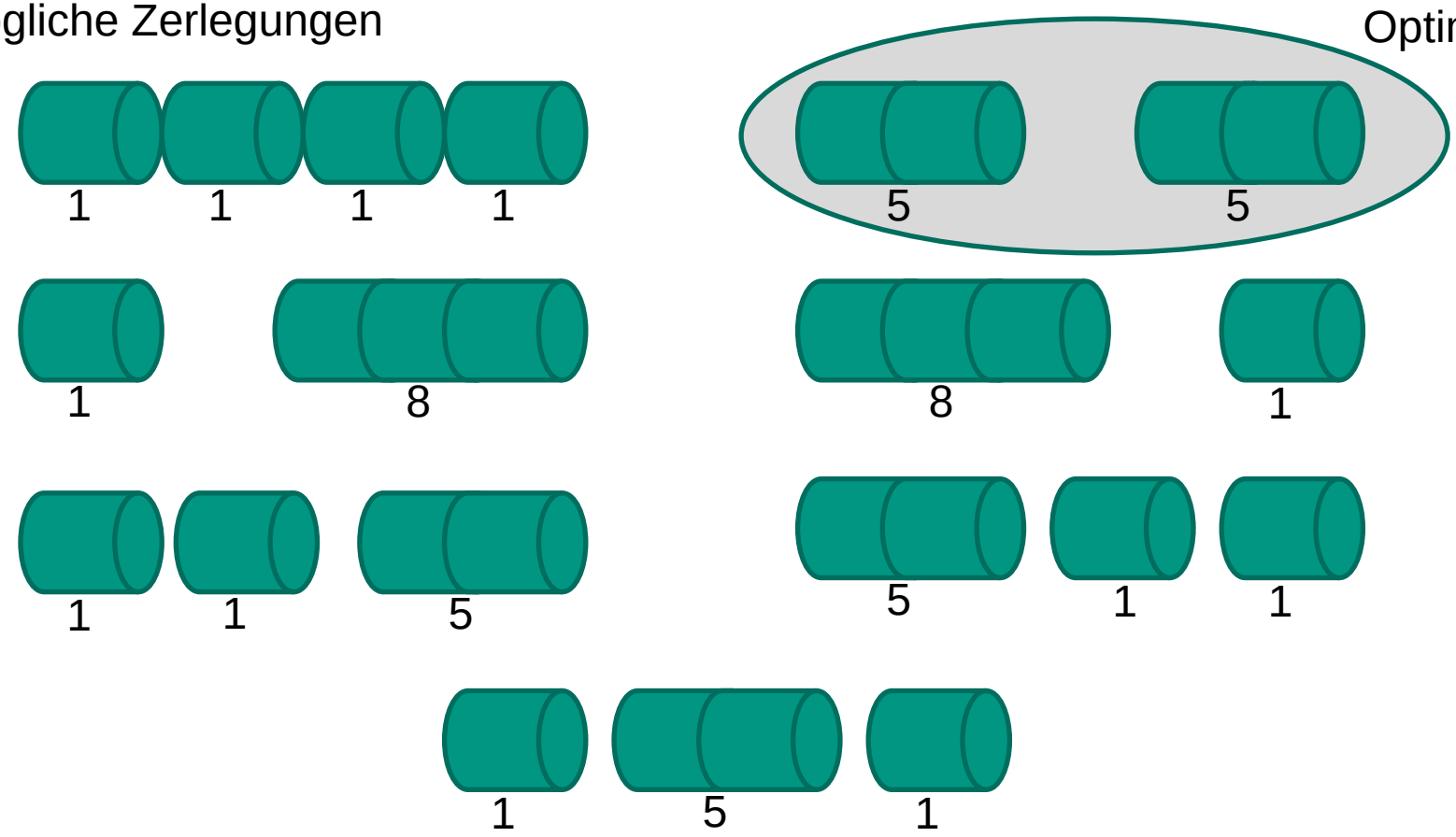
$Länge\ i$	1	2	3	4	5	6	7	8	9	10
$Preis\ p_i$	1	5	8	9	10	17	17	20	24	30

Zerlegung am Beispiel

Gegeben $n = 4$



Mögliche Zerlegungen



Struktur einer optimalen Lösung

- Eisenstange der Länge n kann in 2^{n-1} verschiedenen Weisen zerlegt werden
  - Für jedes $i = 1, 2, \ldots, n-1$ an der Position i freie Wahl, zu teilen oder nicht zu teilen
  - Darstellung einer Zerlegung in der Form $7 = 2 + 2 + 3$
    - Entspricht einer Zerlegung in 2 Stücke der Länge 2 und 1 Stück der Länge 3

- Falls eine optimale Lösung den Eisenstab in k Teilstäbe für ein $1 \leq k \leq n$ schneidet, gilt
  - Optimale Zerlegung $n = i_1 + i_2 + \cdots + i_k$ ergibt optimalen Erlös $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$

Struktur einer optimalen Lösung am Beispiel

- Optimalen Beträge r_i für $i = 1, 2, \ldots, 10$ ergeben sich schrittweise aus optimalen Zerlegungen

$Länge\ i$	1	2	3	4	5	6	7	8	9	10
$Preis\ p_i$	1	5	8	9	10	17	17	20	24	30

r_i	ergibt sich aus	Schnitte ?
$r_1 = 1$	1=1	Nein
$r_2 = 5$	2=2	Nein
$r_3 = 8$	3=3	Nein
$r_4 = 10$	4=2+2	Ja
$r_5 = 13$	5=2+3	Ja
$r_6 = 17$	6=6	Nein
$r_7 = 18$	7=1+8 oder 7=2+2+3	Ja
$r_8 = 22$	8=2+6	Ja
$r_9 = 25$	9=3+6	Ja
$r_{10} = 30$	10=10	Nein

Struktur einer optimalen Lösung am Beispiel

- Generell kann r_n für $n \geq 1$ aus optimalen Erlösen kürzerer Stäbe zusammengesetzt werden

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

- Untersuche alle möglichen Zerlegungen und wählen diejenige mit dem höchsten Erlös
  - Lösen von kleineren Probleme des gleichen Typs, um das ursprüngliche Problem der Größe n zu lösen
- Stabproblem hat sogenannte optimale-Teilstruktur-Eigenschaft
  - Optimale Lösungen eines Problems setzen sich zusammen aus optimalen Lösungen zugehöriger Teilprobleme, die unabhängig voneinander gelöst werden können

Rekursive Lösung des Stabzerlegungsproblem

- Rekursiv ausgedrückt, lässt sich der optimale Erlös darstellen als

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

- Zerlegung in Teilstäbe der Länge i und der Länge $n - i$
  - Beginne am linken Ende mit der Zerlegung in Teilstab der Länge i
  - Danach Teilstab der Länge $n - i$ weiter zerlegen

- Obige Formel lässt sich einfach rekursiv implementieren

Eine rekursive top-down Implementierung

- $CUT - ROD(p, n)$

```
1 if n == 0
2         return 0
3 q = −∞
4 for i = 1 to n
5         q = max(q, p[i] + CUT − ROD(p, n − i))
6 return q
```

Eingabe sind die Länge des zu teilenden Stabes und die Preisliste p

Abbruchbedingung der Rekursion

Rekursiver Aufruf von $CUT - ROD$

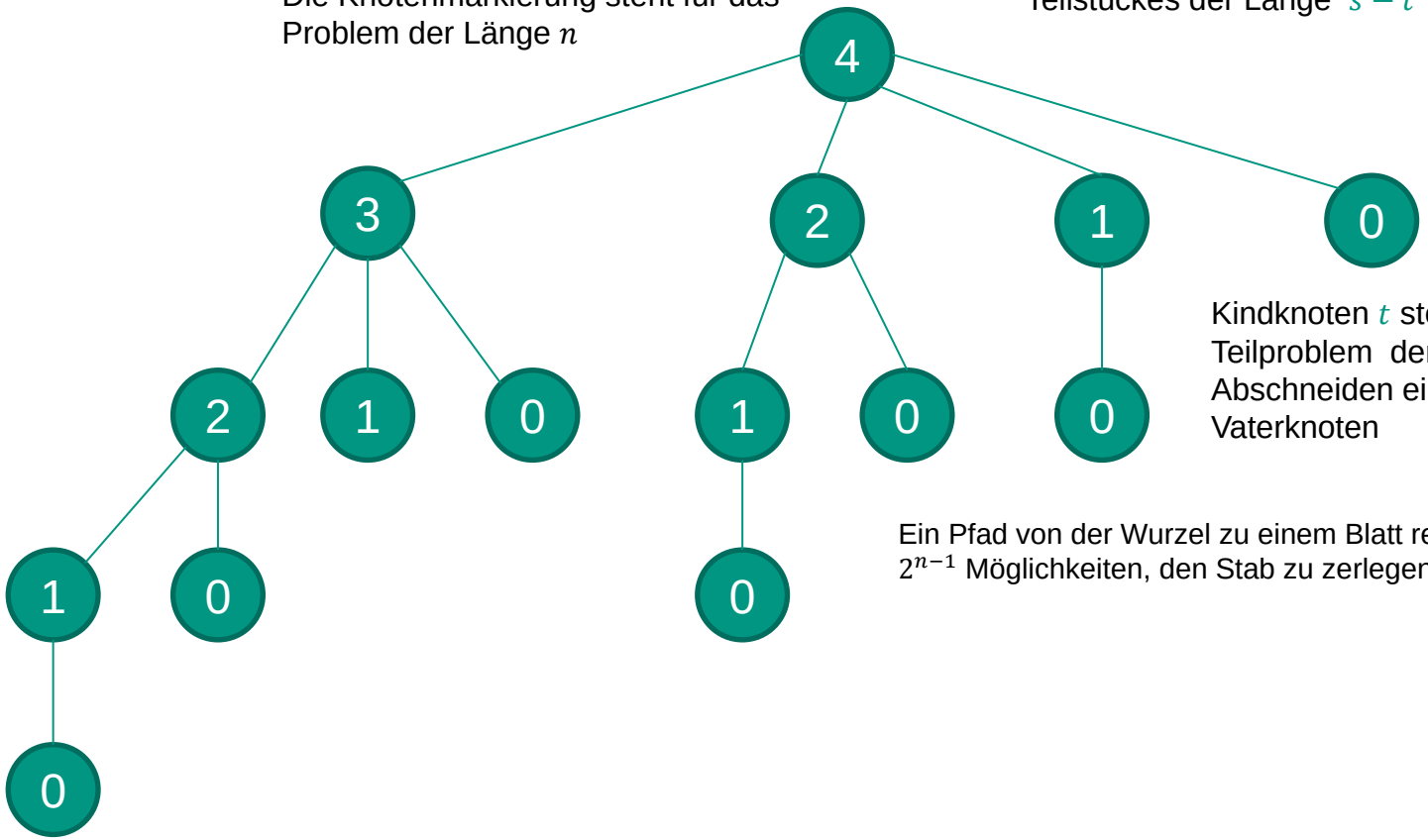
q enthält am Ende den maximal erreichbaren Erlös

Komplexität der rekursiven top-down Implementierung

- Rekursionsbaum beim Aufruf für $n = 4$

Die Kante von einem Vaterknoten s zu einem Kindknoten t steht für das Abschneiden eines Teilstückes der Länge $s - t$

Die Knotenmarkierung steht für das Problem der Länge n



Kindknoten t steht für das Teilproblem der Länge t nach Abschneiden eines Teilstückes vom Vaterknoten

Ein Pfad von der Wurzel zu einem Blatt repräsentiert eine der 2^{n-1} Möglichkeiten, den Stab zu zerlegen

Komplexität der rekursiven top-down Implementierung

- Zählen der Aufrufe die von $CUT - ROD(p, n)$ verursacht werden
  - Gesamtanzahl an Aufrufen sei $T(n)$
    - Entspricht Anzahl der Knoten im Rekursionsbaum inklusive Wurzel

- Es gilt
  - $T(0) = 1$ und $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$
    - Erklärung: $CUT - ROD(p, n)$ ruft $CUT - ROD(p, j)$ für $alle\ j = 0, 2, \dots, n-1$ auf
  - Da

$$T(n) = \ 1 + \sum_{j=0}^{n-1} T(j) =$$
$$1 + (2^n - 1) = \ 2^n$$

⟹ Gesamtlaufzeit von $CUT - ROD$ $\quad O(2^n)$

Stabzerlegungsproblem mit dynamischer Programmierung

- Der rekursiver Ansatz ist ineffizient
  - Jedes Teilproblem wird mehrmals gelöst
- Lösung mit Hilfe von dynamischer Programmierung
  - Jedes Teilproblem wird genau einmal gelöst
    - Speichern einer einmal berechneten Lösung
  - Muss das gleiche Teilproblem nochmals gelöst werden
    - Schlage Lösung nach
  - Also ein Laufzeit-Speicher-Tradeoff
- Auf dynamischer Programmierung basierender Ansatz läuft in polynomieller Zeit, wenn
  - die Anzahl der Teilprobleme polynomiell in der Größe der Eingabe ist
  - jedes Teilproblem in polynomieller Zeit zu lösen ist

Stabzerlegungsproblem mit dynamischer Programmierung

- Im allgemeinen zwei äquivalente Möglichkeiten, Ansatz basierend auf dynamischer Programmierung zu implementieren
  - Top-down Memoisation
    - Programm weiterhin rekursiv, speichern der Ergebnisse der Teilprobleme in einem Feld oder Hashtabelle
    - Vor jedem rekursiven Aufruf überprüfen, ob Teilproblem bereits gelöst
  - Bottom-up-Methode
    - Sortieren der Teilprobleme nach ihrer Größe; Lösung vom kleinsten Teilproblem ausgehend aufwärts
    - Für ein spezielles Teilproblem gilt: Alle kleineren Teilprobleme gelöst und Ergebnisse gespeichert

Pseudocode für die top-down Memoisation

- $MEMOIZED - CUT - ROD(p, n)$

```
1 // sei r[0..n] ein neues Feld
2 for i = 0 to n
3         r[i] = −∞
4 return MEMOIZED − CUT − ROD − AUX(p, n, r)
```

- $MEMOIZED - CUT - ROD - AUX(p, n, r)$

```
1 if r[n] ≥ 0
2         return r[n]
3 if n == 0
4         q = 0
5 else q = −∞
6         for i = 1 to n
7                 q = max(q, p[i] + MEMOIZED − CUT − ROD − AUX(p, n − i, r))
8 r[n] = q
9 return q
```

Eingabe sind die Länge des zu teilenden Stabes und die Preisliste p

Initialisierung des Feldes

Aufruf der Hilfsprozedur

Wenn der Wert schon bekannt ist, gebe ihn zurück

Rekursive Lösung

Speichern der Teillösung

Pseudocode für die bottom-up-Methode

- $BOTTOM - UP - CUT - ROD(p, n)$

```
1 // sei r[0..n] ein neues Feld
2 r[0] = 0
3 for j = 1 to n
4         q = −∞
5         for i = 1 to j
6                 q = max(q, p[i] + r[j − i])
7         r[j] = q
8 return r[n]
```

Eingabe sind die Länge des zu teilenden Stabes und die Preisliste p

Lösen der Teilprobleme in aufsteigender Reihenfolge

Rückgabe des optimalen Wertes

Komplexität

- Bottom-up und top-down Ansatz haben gleiche Komplexität

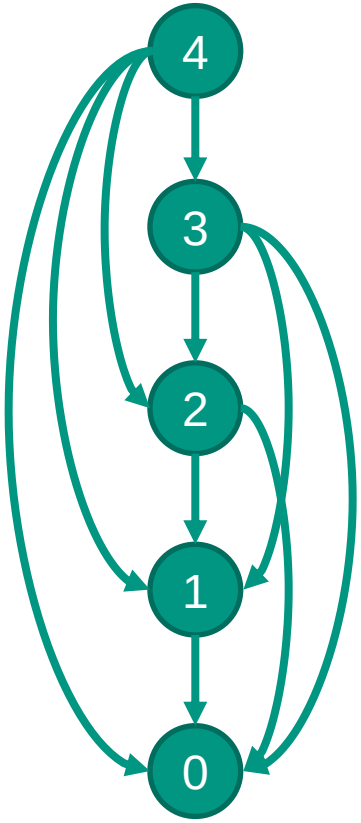
- $BOTTOM - UP - CUT - ROD$
  - Für Aufwand verantwortlich sind folgende Zeilen im Pseudocode
    - 5 **for** i $= 1$ **to** j
      6 $q = \max(q, p[i] + r[j - i])$
  - Anzahl der Iterationen ergibt die arithmetische Reihe
    - Aufwand daher $\theta(n^2)$

- $MEMOIZED - CUT - ROD$
  - Algorithmus löst Teilprobleme der Größe $0, 1, 2, \dots, n$
  - Für Aufwand verantwortlich sind folgende Zeilen im Pseudocode
    - 6 **for** $i = 1$ **to** n
      7 q = $\max(q, p[i] + MEMOIZED - CUT - ROD - AUX(p, n - i, r))$
  - Anzahl der Iterationen ergibt auch hier die arithmetische Reihe
    - Aufwand daher $\theta(n^2)$

Teilproblemgraph

- Enthält Menge der zu lösenden Teilprobleme und deren Abhängigkeiten
  - Ähnlich dem Rekursionsbaum des rekursiven Ansatzes

Teilproblemgraph zum Stabzerlegungsproblem mit $n = 4$



Knotenmarkierung gibt die Größe des Teilproblems an

Gerichtete Kante (x, y): Zur Lösung des Teilproblems x wird Lösung des Teilproblems y benötigt

Teilproblemgraph

- Ist „reduzierte“ Version des Rekursionsbaumes
  - Alle Knoten, die gleiches Teilproblem lösen werden in einem Knoten zusammengefasst

- $BOTTOM - UP - CUT - ROD$
  - Betrachtet Knoten des Teilproblemgraphen in einer Reihenfolge, die umgekehrt topologisch sortiert ist
  - Kein Teilproblem wird betrachtet, bis nicht alle zur Lösung benötigten Teilprobleme gelöst sind

- $MEMOIZED - CUT - ROD$
  - Tiefensuche im Teilproblemgraph

Rekonstruktion einer Lösung

- Bisher: Mit Hilfe der dynamischen Programmierung einen optimalen Wert einer Lösung ermittelt
  - Allerdings meist auch die tatsächliche Lösung notwendig
  - Im Beispiel die Zerlegung des Stabes

- Nun: Erweiterung, um eine optimale Lösung auszugeben
  - Zusätzliches Feld zur Speicherung der Lösung
    - Genauer: Speicherung der optimalen Größe der Teilstäbe

Rekonstruktion einer Lösung - Pseudocode

- $EXTENDED - BOTTOM - UP - CUT - ROD(p, n)$

```
1  // seien r[0..n] und s[0..n] neue Felder
2  r[0] = 0
3  for j = 1 to n
4        q = -∞
5        for i = 1 to j
6              if q < p[i] + r[j - i]
7                    q = p[i] + r[j - i]
8                    s[j] = i
9        r[j] = q
10 return r und s
```

Initialisierung der Felder

Lösen der Teilprobleme in aufsteigender Reihenfolge

Speichern der optimalen Größe i des ersten Teilstabes beim Lösen des Teilproblems der Größe j

Ausgabe einer Lösung - Pseudocode

- $PRINT - CUT - ROD(p, n)$

```
1 (r, s) = EXTENDED − BOTTOM − UP − CUT − ROD(p, n)
2 while n > 0
3       print s[n]
4       n = n − s[n]
```

- Beim Aufruf von $EXTENDED - BOTTOM - UP - CUT - ROD(p, 10)$ gilt

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- Ausgabe beim Aufruf $PRINT - CUT - ROD(p, 7)$?

11.2.2 Matrizen Kettenmultiplikation

- Gegeben
  - Folge von n Matrizen $< A_1, A_2, \dots, A_n >$
- Gesucht
  - Das Produkt $A_1 * A_2 * \dots * A_n$
- Eigenschaften
  - Klammerung des Ausdrucks notwendig
    - Produkt von Matrizen heißt vollständig geklammert, wenn
      - Einzelne Matrix
      - Produkt zweier vollständig geklammerter Matrizenprodukte
  - Matrizen müssen kompatibel zueinander sein
    - Anzahl der Spalten von A gleich Anzahl der Zeilen von B, für $A * B$

Standardalgorithmus zur Matrizenmultiplikation

- $MATRIX - MULTIPLY(A, B)$

```
1 if A.spalten ≠ B.zeilen
2       error „inkompatible Dimensionen“
3 else sei C eine neue A.zeilen x B.spalten Matrix
4       for i = 1 to A.zeilen
5             for j = 1 to B.spalten
6                   c_ij = 0
7                   for k = 1 to A.spalten
8                         c_ij = c_ij + a_ik * b_kj
9 return C
```

Überprüfung ob Matrizen kompatibel sind

Multiplikation $A.Zeile\ x\ B.Spalte$

Komplexität dominiert durch Multiplikationen

Ausgabe der Ergebnis Matrix

Einfluß der Klammerung am Beispiel

- Beispiel: 6 Matrizen

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} \begin{pmatrix} d_{11} & d_{12} \end{pmatrix} \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

Matrix A	B	C	D	E	F
(4x2)	(2x3)	(3x1)	(1x2)	(2x2)	(2x3)

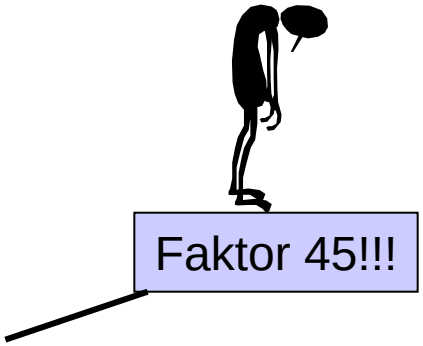
- Frage: Wie hoch ist die Gesamtzahl der benötigten Skalar-Multiplikationen?
  - Erste Idee:
    Multipliziere zunächst Matrix A mit B
    Multipliziere das Ergebnis mit Matrix C
    Multipliziere das Ergebnis mit Matrix D
    …

24 Multiplikationen (4x2x3)
+12 Multiplikationen (4x3x1)
+8 Multiplikationen (4x1x2)

Beispiel Fortsetzung

- Insgesamt 84 Skalar-Multiplikationen notwendig, bei dieser Art des Vorgehens: „von links nach rechts Multiplizieren“
  - Klammerung: (((((AB)C)D)E)F)

- Beobachtung
  - Wenn wir stattdessen von „rechts nach links Multiplizieren“
    - Klammerung: (A(B(C(D(EF)))))
      - ⇒ insgesamt nur 69 Skalar-Multiplikationen!
  - In diesem Beispiel ist der Unterschied relativ gering (15 Skalar-Multiplikationen)
  - Komplexeres Beispiel
    - Betrachte B, C, F nicht mit Dimension 3 sondern 300
      - rechts-links ⇨ 6024 Skalar-Multiplikationen
      - links-rechts ⇨ 274.200 Skalar-Multiplikationen



Matrizenkettenmultiplikation

- Problem der Matrizenkettenmultiplikation
  - Gegeben: Kette $< A_1,\ A_2, \dots ,\ A_n >$ von n Matrizen
    - Für $i = 1,2, \dots, n$ besitzt A_i die Dimension $p_{i-1} x\ p_i$
  - Gesucht
    - Klammerung, die die Anzahl der skalaren Multiplikationen minimiert

- Anwendung von dynamischer Programmierung
  - Charakterisieren der Struktur einer optimalen Lösung
  - Definition des Werts einer optimalen Lösung rekursiv
  - Wert einer optimalen Lösung mit Hilfe eines bottom-up-Ansatz berechnen
  - Konstruktion einer zugehörigen optimale Lösung aus schon berechneten Daten

Struktur einer optimalen Lösung

- Gesucht
  - Klammerung, die Anzahl der skalaren Multiplikationen minimiert

- Beobachtung
  - Um $A_i A_{i+1}$ ... A_j zu klammern, muss Produkt aufgespaltet werden
    - Aufspaltung zwischen A_k und A_{k+1} für $i \leq k < j$ notwendig
  - Also: Lösung von $A_{i..k}$ und $A_{k+1..j}$ notwendig, um $A_{i..j}$ zu erhalten
    - Sei $A_{i..j}$ eine Bezeichnung für das Produkt $A_i A_{i+1}$ ... A_j
  - Kosten für die Berechnung $A_i A_{i+1}$ ... A_j setzen sich zusammen aus
    - Kosten für Berechnung $A_{i..k}$
    - Kosten für Berechnung $A_{k+1..j}$
    - Kosten für Multiplikation $A_{i..k}$ und $A_{k+1..j}$

Struktur einer optimalen Lösung

- Annahme zur optimalen Klammerung
  - Spaltung des Produkts $A_i A_{i+1}$ ... A_j zwischen A_k und A_{k+1}
  - Es existiert optimale Klammerung von $A_i A_{i+1}$ ... A_j

- Es muss gelten
  - Klammerung des Teilproblems $A_{i..k}$ innerhalb von optimaler Klammerung von $A_i A_{i+1}$ ... A_j muss auch optimal sein
  - Warum?
    - Wenn günstigere Klammerung von $A_{i..k}$ existieren würde, wäre durch Einsetzen in $A_{i..j}$ eine günstigere Klammerung von $A_{i..j}$ möglich → Widerspruch!
    - Gleiches gilt für $A_{i..j}$ und $A_{k+1..j}$

- Also: Zerlegung in Teilprobleme $A_{i..k}$ und $A_{k+1..j}$ mit anschließender Lösung der Teilprobleme führt zu optimaler Lösung

Rekursive Lösung

- Gesucht
  - Klammerung, die Anzahl der skalaren Multiplikationen minimiert
  - Genauer: Minimale Kosten, um $A_i A_{i+1} .. A_j$ zu klammern
    - Es gilt $1 \leq i \leq j \leq n$

- Sei $m[i, j]$ das Minimum für die Anzahl der für Berechnung von $A_{i..j}$ benötigten skalaren Multiplikationen
  - Für Gesamtproblem also $m[1, n]$ gesucht

Rekursive Lösung

- Rekursive Definition von $m[i,j]$
  - Für $i = j$ Problem trivial
    - $A_{i..i} = A_i$, keine Multiplikationen notwendig
    - Also $m[i,i] = 0$ für alle $i = 1,2,\dots,n$
  - Für $i < j$ nutzen wir Struktur der optimalen Lösung
    - Aufspaltung zwischen A_k und A_{k+1}um Produkt optimal zu klammern
    - $m[i,j]$ setzt sich zusammen aus
      - minimalen Kosten um $A_{i..k}$ und $A_{k+1..j}$ zu berechnen
      - Kosten der Multiplikation dieser Matrizen

    $\Rightarrow \quad m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$

  - Problem gelöst?

Rekursive Lösung

- Wir kennen den optimalen Wert für k nicht
  - Es existieren allerdings nur $j - i$ Möglichkeiten für k
    - $(k = i, i+1 .. j-1)$
  - Also alle ausprobieren, um besten Wert von k zu finden

- Rekursive Definition für minimalen Kosten also
  - $m[i,j] = 0$ falls $i = j$
  - $m[i,j] = \min_{i \le k \le j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}$ falls $i < j$

- Was fehlt noch?
  - Mit Hilfe von $m[i,j]$ kennen wir die minimalen Kosten der Klammerung
  - Wert von k ?
    - Hilfsdefinition: Sei $s[i,j]$ ein Wert von k, für den $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ gilt

Pseudocode für rekursiven Ansatz

- $RECURSIVE - MATRIX - CHAIN(p, i, j)$

```
1  if i = j
2        return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5        q = RECURSIVE − MATRIX − CHAIN(p, i, k) +
6            RECURSIVE − MATRIX − CHAIN(p, k + 1, j) +
7            p_{i−1} * p_k * p_j
8        if q < m[i, j]
9              m[i, j] = q
10 return m[i, j]
```

$p =< p_0, p_1 .. p_n >$ seien die Dimensionen der Matrizen, wobei jede Matrix die Dimension $p_{i-1} x\ p_i$ für $i = 1,2, \ldots, n$ hat

Rekursive Lösung analog zur rekursiven Definition

Speicherung und Ausgabe der optimalen Lösung

→ Komplexität von $RECURSIVE - MATRIX - CHAIN(p, 1, n)$ in $O(2^n)$

Berechnung der optimalen Kosten

- Tabellarischer bottom-up-Ansatz
- Voraussetzungen
  - Dimension der Matrix $p_{i-1} x\ p_i$ für $i = 1, 2, \ldots, n$
  - Eingabe $p = < p_0, p_1 .. p_n >$ mit $p.länge = n + 1$
  - Für $k = i, i + 1, \ldots, j - 1$ ist
    - $A_{i..k}$ ein Produkt von $k - i + 1$ Matrizen
    - $A_{k+1..j}$ ein Produkt von $j - k$ Matrizen
    - Also Teilprobleme jeweils kleiner als Kettenprodukt der $i + j - 1$Matrizen der Eingabe

→ Algorithmus soll Tabelle in Reihenfolge füllen, die der Berechnung der optimalen Klammerung von Matrizenketten in wachsender Reihenfolge entspricht

Pseudocode für bottom-up Ansatz

- $MATRIX - CHAIN - ORDER(p)$

```
1  n = p.länge − 1
2  // seien m[1..n, 1..n] und s[1..n − 1, 2..n] neue Tabellen
3  for i = 1 to n
4          m[i, i] = 0
5  for l = 2 to n
6          for i = 1 to n − l + 1
7                  j = i + l − 1
8                  m[i, j] = ∞
9                  for k = 1 to j − 1
10                         q = m[i, k] + m[k + i, j] + p_{i−1} p_k p_j
11                         if q < m[i, j]
12                                 m[i, j] = q
13                                 s[i, j] = k
14 return m und s
```

l ist Länge der Matrizenkette

Rekursive Lösung in aufsteigender Reihenfolge

Kosten hängen nur von bereits gelösten Teilproblemen ab

Beispiel für bottom-up Ansatz

- Eingabe:

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	$30x35$	$35x15$	$15x5$	$5x10$	$10x20$	$20x25$

m

j \ i	1	2	3	4	5	6
6	15125	10500	5375	3500	5000	0
5	11875	7125	2500	1000	0	
4	9375	4375	750	0		
3	7875	2625	0			
2	15750	0				
1	0					

Also $p =< 30,35,15,5,10,20,25 >$

Tabelle m wie sie beim Aufruf von $MATRIX - CHAIN - ORDER(p)$ erstellt wird

Beispiel: $m[1,6] = 15125$
$m[2,5] = 7125$

Allgemein gilt $m[i,j] = \min_{i \le k \le j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}$

Komplexität des bottom-up Ansatz

- $MATRIX - CHAIN - ORDER(p)$

```
1  n = p.länge − 1
2  // seien m[1..n, 1..n] und s[1..n − 1, 2..n] neue Tabellen
3  for i = 1 to n
4          m[i, i] = 0
5  for l = 2 to n
6          for i = 1 to n − l + 1
7                  j = i + l − 1
8                  m[i, j] = ∞
9                  for k = 1 to j − 1
10                 q = m[i, k] + m[k + q, j] + p_{i−1} p_k p_j
11                 if q < m[i, j]
12                         m[i, j] = q
13                         s[i, j] = k
14 return m und s
```

2 Felder der Größe n^2

n mal

$n - 1$ mal

$n - 1$ mal

$n - 1$ mal

Laufzeitkomplexität $O(n^3)$

Speicherkomplexität $\theta(n^2)$

Konstruktion einer optimalen Lösung

- $MATRIX - CHAIN - ORDER(p)$ bestimmt Anzahl skalarer Multiplikationen
  - Es fehlt noch Ausgabe der konkreten Klammerung
  - Dies kann mit Hilfe von $s[1..n-1,2..n]$ ausgeben werden
    - Jeder Eintrag $s[i,j]$ speichert die Stelle k der optimalen Klammerung
  - Also für $A_{1..n}$ ist die letzte Multiplikation $A_{1..s[1,n]} * A_{s[1,n]+1..n}$
    - $s[1,s[1,n]]$ gibt analog die letzte Multiplikation für $A_{1..s[1,n]}$ an
    - $s[s[1,n]+1,n]$ gibt die letzte Multiplikation für $A_{s[1,n]+1..n}$ an
- Berechnung der Klammerung also über Rekursion möglich

Pseudocode für Konstruktion einer optimalen Lösung

- $PRINT - OPTIMAL - PARENS(s, i, j)$

```
1 if i == j
2       print "A"_i
3 else print "("
4       PRINT − OPTIMAL − PARENS(s, i, s[i, j])
5       PRINT − OPTIMAL − PARENS(s, s[i, j] + 1, j)
6       print ")"
```

Letzter Rekursionschritt

Rekursiver Aufruf um Teilprobleme zu lösen

$PRINT - OPTIMAL - PARENS(s, 1,6)$ liefert Ausgabe $((A_1(A_2A_3))((A_4A_5)A_6))$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	$30x35$	$35x15$	$15x5$	$5x10$	$10x20$	$20x25$

11.3 Greedy-Algorithmen

- Bei vielen Optimierungsproblemen ist es übertrieben, dynamische Programmierung zu verwenden
  - Häufig gibt es effizientere und einfachere Algorithmen
  - Insbesondere Zwischenspeicherung der Ergebnisse aufwendig

- Greedy-Algorithmen treffen jeweils diejenige Entscheidung, die im Moment am optimalsten erscheint
  - Lokal optimale Entscheidung
  - Greedy-Algorithmen liefern häufig nicht optimale Ergebnisse
  - Beispiele für Greedy-Algorithmen aus der Vorlesung
    - Minimale Spannbäume, Dijkstra-Algorithmus

11.3.1 Das Aktivitäten-Auswahl-Problem

- Gegeben
  - Menge $S = \{a_1, a_2, .., a_n\}$ von n anstehenden Aktivitäten
  - Jede der a_i Aktivitäten will eine Ressource nutzen
    - Immer nur eine Aktivität kann gleichzeitig Ressource nutzen
  - Jede Aktivität a_i besitzt Startzeit s_i und Endzeit f_i
    - Es gilt $0 \leq s_i < f_i < \infty$
    - Ausgewählte Aktivität findet im Zeitintervall $[s_i, f_i)$ statt
  - Aktivitäten a_i und a_j sind kompatibel, wenn sich $[s_i, f_i)$ und $[s_j, f_j)$ nicht überlappen
    - Also wenn $s_i \geq f_j$ oder $s_j \geq f_i$
  - Aktivitäten seien in monoton aufsteigender Reihenfolge nach Endzeiten sortiert
    - $f_1 \leq f_2 \leq \cdots f_n$
- Gesucht
  - Eine maximale Teilmenge paarweise kompatibler Aktivitäten

➡ Aktivitäten-Auswahl-Problem

Beispiel

Gegeben: Menge S an Aktivitäten

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Teilmenge $\{a_3, a_9, a_{11}\}$ ist kompatibel. Menge auch maximale Teilmenge ?

Teilmenge $\{a_1, a_4, a_9, a_{11}\}$ ist kompatibel und maximale Teilmenge

Entwurf der optimalen Teilstruktur einer Lösung

- Annahmen
  - Sei S_{ij} die Menge der Aktivitäten, die
    - starten, nachdem a_i endet
    - enden, bevor a_j beginnt
  - Sei A_{ij} eine maximale Menge paarweise kompatibler Aktivitäten aus S_{ij}
    - Weiterhin enthalte A_{ij} die Aktivität a_k

- Wenn a_k in optimaler Lösung enthalten, dann gilt
  - Aktivitäten der Menge S_{ik} sind paarweise kompatibel
  - Aktivitäten der Menge S_{kj} sind paarweise kompatibel
  - Also $A_{ik} = A_{ij} \cap S_{ik}$ und $A_{kj} = A_{ij} \cap S_{kj}$
  - ⇒ $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - ⇒ Maximale Menge kompatibler Aktivitäten $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$

Entwurf der optimalen Teilstruktur einer Lösung

- Es gilt: Optimale Lösung A_{ij} setzt sich zusammen aus
  - Optimaler Lösung von S_{ik}
  - Optimaler Lösung von S_{kj}

- Beweis
  - Sei $A_{kj}{}'$ aus S_{kj} eine paarweise zueinander kompatible Menge
  - Sei $|A'_{kj}| > A_{kj}$
  - Dann einsetzen in maximale Menge kompatibler Aktivitäten

$$|A_{ik}|+1 + |A'_{kj}| > |A_{ik}|+1 + |A_{kj}| = |A_{ij}|$$

  - Widerspruch zu maximaler Menge!
    - Argumentation gilt auch für S_{ik}

Entwurf einer rekursiven Lösung

- Sei $c[i,j]$ Größe einer optimalen Lösung für die Menge S_{ij}
- Es gilt
  - $c[i,j] = c[i,k] + c[k,j] + 1$
- Da k unbekannt ist gilt
  - $c[i,j] = 0$ falls $S_{ij} = \emptyset$
  - $c[i,j] = \max_{a_k \in S_{ij}} \{ c[i,k] + c[k,j] + 1 \}$ falls $S_{ij} \neq \emptyset$
- Also nun Lösung mit dynamischer Programmierung?
  - Führt zu optimaler Lösung
  - Aber: Es geht einfacher

Entwurf einer rekursiven Lösung

- Anstatt alle Lösungen zu berechnen Greedy-Wahl treffen
  - Intuitive Wahl: Wähle Aktivität, die zum frühesten Zeitpunkt endet
    - Also wähle a_1, da Aktivitäten monoton aufsteigend sortiert
  - Was folgt danach?
    - Suche Aktivität, die startet nachdem a_1 endet
    - Dies gilt da $s_1 < f_1$ und f_1 frühester Endzeitpunkt
    - Alle zu a_1 kompatiblen Aktivitäten müssen also nach f_1 starten
  - Allgemein
    - Sei $S_k = \{a_i \in S: s_i \geq f_k\}$ die Menge an Aktivitäten, die starten nachdem a_k endet
    - Wählen a_1 in der Greedy-Entscheidung, dann ist nur noch S_1 zu lösen
  - Ist die Greedy-Wahl korrekt ?
    - Finden wir so eine optimale Lösung ?

Entwurf einer rekursiven Lösung

- Es gilt
  - Sei S_k eine nicht leere Teilmenge
  - Sei a_m eine Aktivität aus S_k mit frühester Endzeit
  - Dann a_m in einer maximal großen Teilmenge von paarweise zueinander kompatiblen Aktivitäten aus S_k enthalten
- Beweis
  - Sei A_k maximale große Teilmenge kompatibler Aktivitäten und a_j eine Aktivität aus A_k mit frühester Endzeit
    - Wenn $a_j = a_m$, dann ist a_j in Teilmenge enthalten
    - Wenn $a_j \neq a_m$, dann
      - $A'_k = A_k - \{a_j\} \cup \{a_m\}$
      - Alle Aktivitäten in A'_k sind disjunkt da
        - Alle Aktivitäten in A_k disjunkt
        - a_j die erste Aktivität aus A_k ist die endet
        - $f_m \leq f_j$
    - Wegen $|A'_k| = |A_k|$ folgt, A'_k ist maximal große Menge und enthält a_m q.e.d

Pseudocode einer rekursiven Lösung

- Eingabe
  - Felder f und s mit End- bzw. Startzeitpunkten
  - Index k, der das zu lösende Teilproblem S_k definiert
    - Aktivitäten in monoton steigender Reihenfolge nach Endzeitpunkten sortiert
    - Wenn nicht, sortiere in $O(n \lg n)$
- Initialer Aufruf mit $\text{RECURSIVE} - ACTIVITY - SELECTOR(s, f, 0, n)$
  - Vorher einfügen einer fiktiven Aktivität a_0 mit Endzeitpunkt $f_0 = 0$

Pseudocode einer rekursiven Lösung

- $\text{RECURSIVE} - \textit{ACTIVITY} - \textit{SELECTOR}(s, f, k, n)$

```
1 m = k + 1
2 while m ≤ n und s[m] < f[k]
3         m = m + 1
4 if m ≤ n
5         return {a_m} ∪ RECURSIVE − ACTIVITY − SELECTOR(s, f, m, n)
6 else return ∅
```

Eingabe sind Startzeitpunkte in s und die Endzeitpunkte in f, Index k des zu lösenden Teilproblems und die Größe n

Suche erste kompatible Aktivität in S_k

Rekursive Lösung der Teilprobleme
Terminiert wenn $m > n$

Komplexität $\theta(n)$, da jede Aktivität nach allen rekursiven Aufrufen genau 1 mal in der **while**-Schleife überprüft wurde

Beispiel

Gegeben: Menge S an Aktivitäten

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Betrachten der Werte für k, $s_k$und f_k und die Auswahl der Aktivitäten

1. Schritt

k	s_k	f_k
0	-	0

Füge Aktivität a_0 ein, starte $\text{RECURSIVE} - ACTIVITY - SELECTOR(s, f, 0, 11)$

2. Schritt

k	s_k	f_k
1	1	4

Wähle Aktivität a_1 aus, starte $\text{RECURSIVE} - ACTIVITY - SELECTOR(s, f, 1, 11)$

3. Schritt

k	s_k	f_k
4	5	7

Wähle Aktivität a_4 aus, da $s_4 > f_1$. Starte $\text{RECURSIVE} - ACTIVITY - SELECTOR(s, f, 4, 11)$

Beispiel

Gegeben: Menge S an Aktivitäten

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

4. Schritt

k	s_k	f_k
8	8	11

Wähle Aktivität a_8 aus, da $s_8 > f_4$, starte $\text{RECURSIVE} - ACTIVITY - SELECTOR(s, f, 8, 11)$

5. Schritt

k	s_k	f_k
11	12	16

Wähle Aktivität a_{11} aus, da $s_{11} > f_8$, starte $\text{RECURSIVE} - ACTIVITY - SELECTOR(s, f, 11, 11)$

6. Schritt

Aktivitäten a_1, a_4, a_8 und a_{11} sind eine optimale Lösung

Elemente der Greedy-Strategy

- Allgemein: Entwurf eines Greedy-Algorithmus mit folgenden Schritten
  - Optimierungsproblem in folgender Form darstellen
    - Greedy-Wahl treffen
    - Danach existiert nur noch ein zu lösendes Teilproblem
  - Beweise, dass immer optimale Lösung existiert, die Greedy-Wahl enthält
    - Somit ist es nie falsch, die Greedy-Wahl zu treffen
  - Zeigen, dass die optimale-Teilstruktur-Eigenschaft gilt
    - Wenn eine optimale Lösung des entstandenen Teilproblems mit der Greedy-Wahl kombiniert, führt dies zu optimaler Lösung des ursprünglichen Problems

11.3.2 Greedy-Strategy vs. Dynamische Programmierung

- Sowohl dynamische Programmierung als auch Greedy-Strategy nutzen optimale-Teilstruktur-Eigenschaft
  - Also immer Greedy-Strategy anwenden , wenn Lösung mit dynamischer Programmierung existiert ?

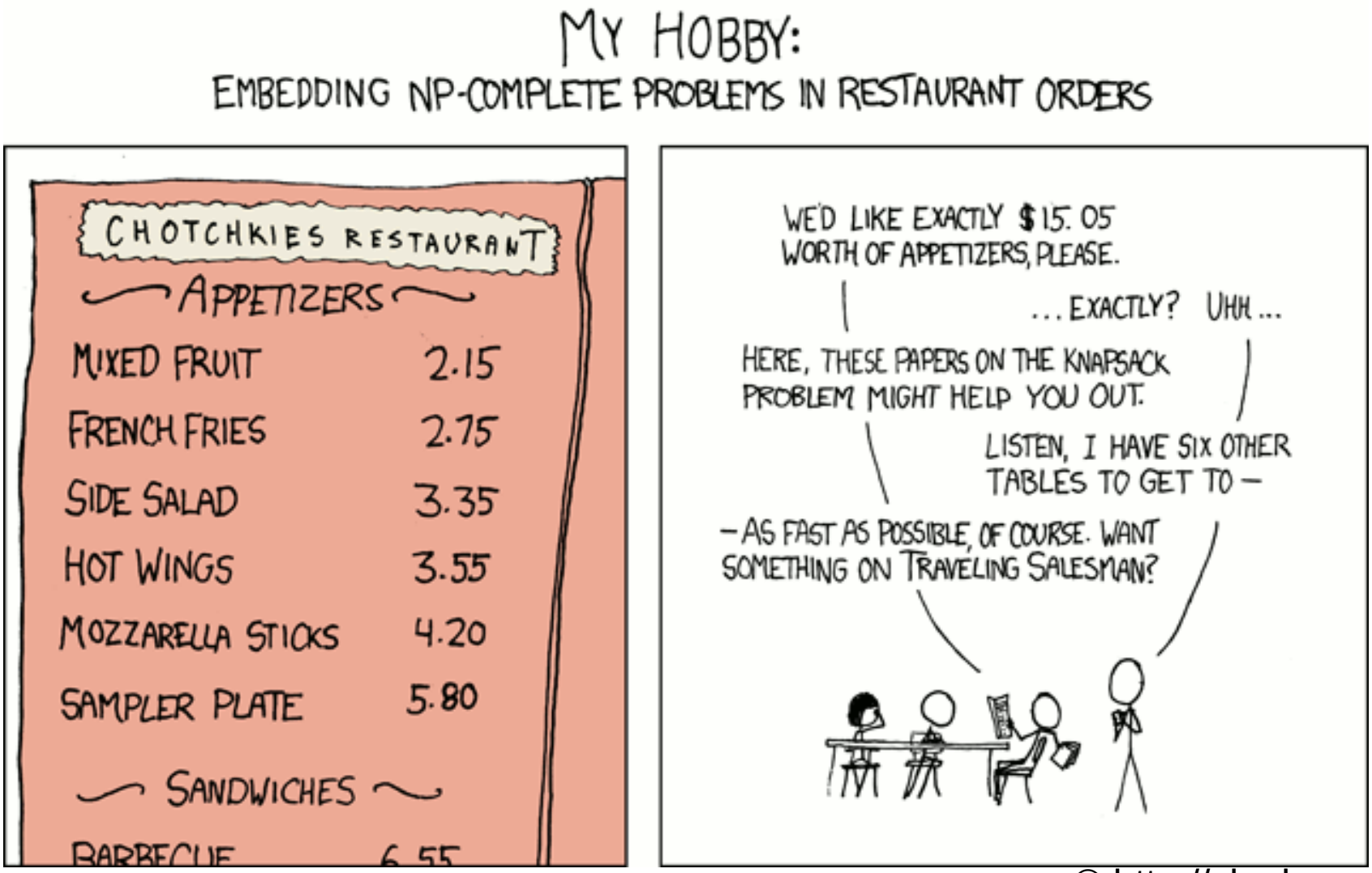
- Wir betrachten *0-1-Rucksackproblem* und das *gebrochene Rucksackproblem*

Das *0-1*-Rucksackproblem

- Gegeben sei das *0-1*-Rucksackproblem
  - Es existieren n Gegenstände
  - Der $i-te$ Gegenstand sei v_i Euro wert und wiege w_i Kilo
  - Es können insgesamt W Kilo in einem Rucksack transportiert werden
    - W, v_i und w_i sind ganze Zahlen

- Gesucht
  - Welche Gegenstände müssen im Rucksack transportiert werden, damit der Wert der transportierten Gegenstände maximal ist?

Beispiel



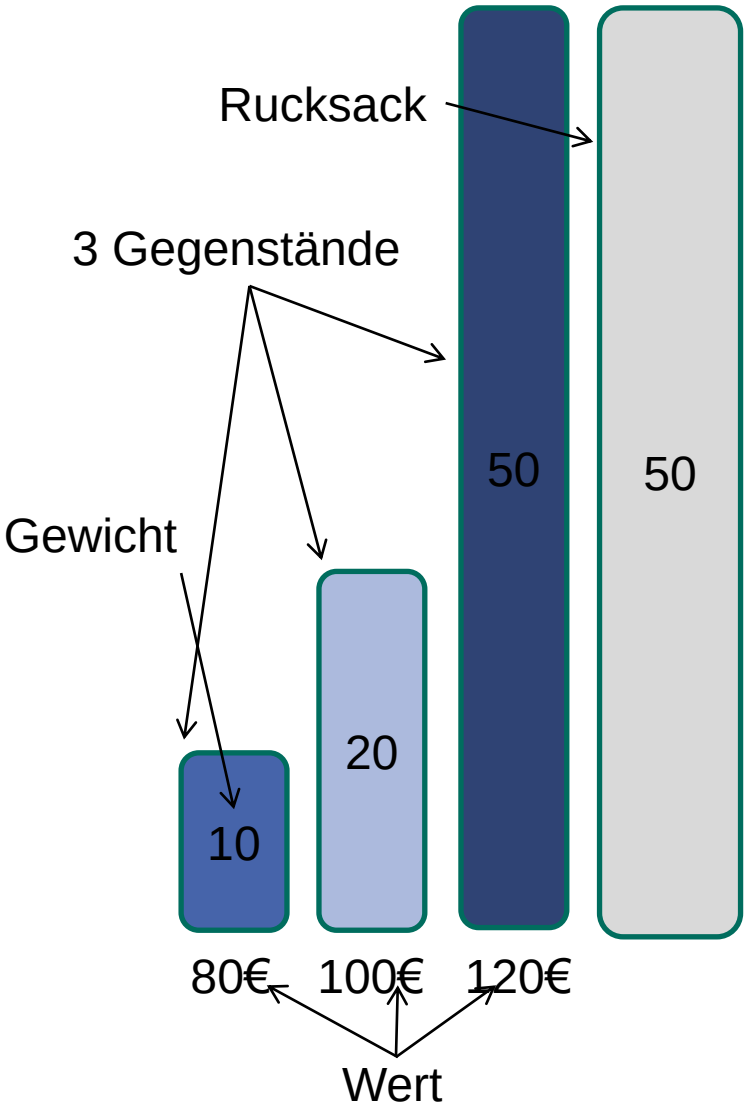
© http://xkcd.com/

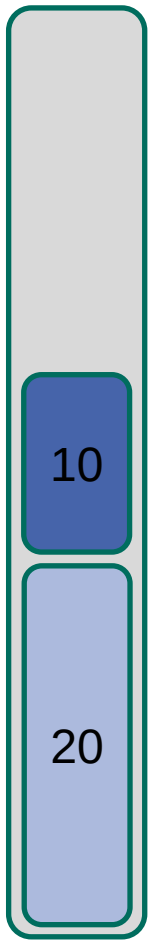
Greedy-Strategy *0-1*-Rucksackproblem

- Optimale-Teilstruktur Eigenschaft
  - Betrachte wertvollste Rucksackladung, die höchstens W Kilo schwer ist
  - Entferne Gegenstand j aus Rucksackladung
    - Verbleibende Rucksackladung wiegt maximal $W - w_j$
    - Verbleibende Rucksackladung immer noch wertvollste Rucksackladung, die aus $n - 1$ Gegenständen ohne j gebildet werden kann

- Greedy-Wahl
  - Wähle Gegenstand mit größtem Wert ?
  - Wähle Gegenstand mit kleinstem Gewicht ?
  - Wähle Gegenstand mit größtem Gewicht ?
  - Wähle Gegenstand mit größtem Wert pro Kilo?

Gegenbeispiel größter Wert





≠

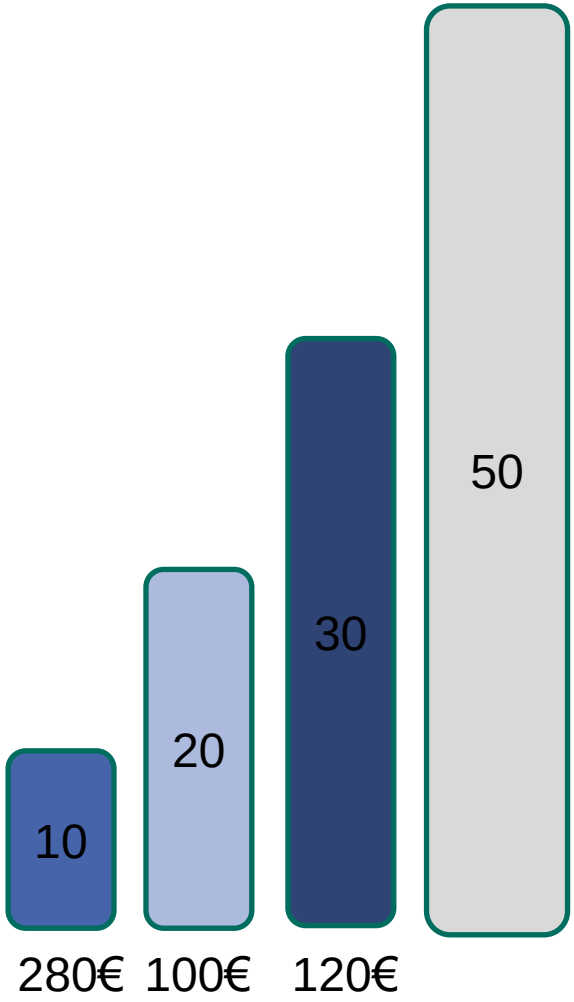
Optimale Lösung?

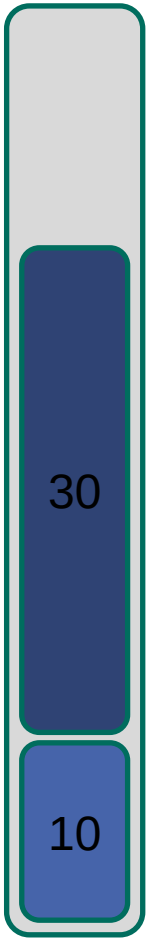
Wert = 180 €

Wähle Gegenstand mit größtem Wert

Wert = 120 €

Gegenbeispiel größtes Gewicht

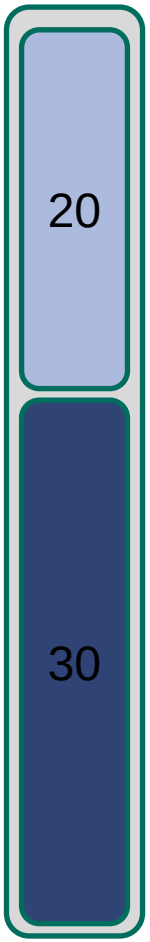




Optimale Lösung?

Wert = 400 €

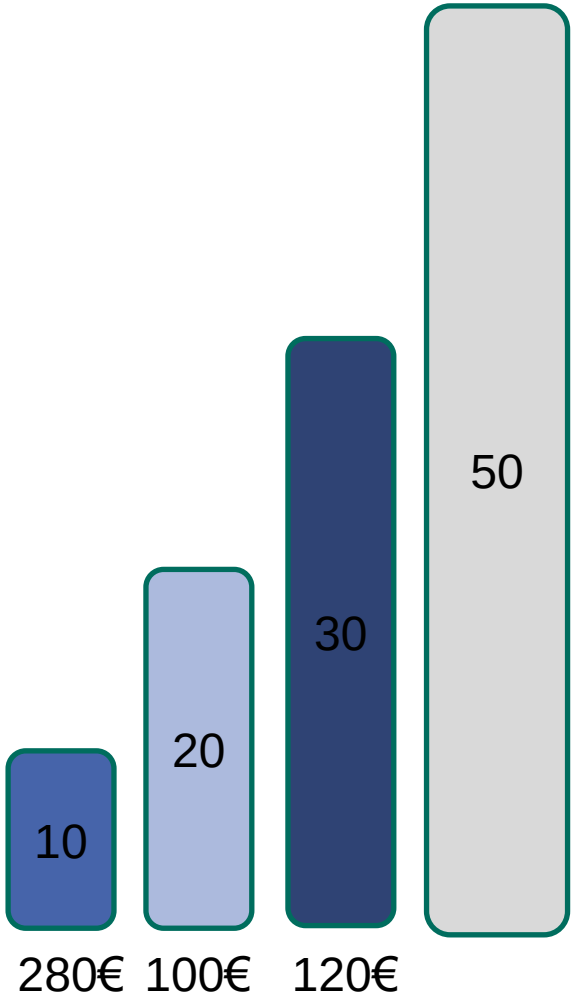
≠



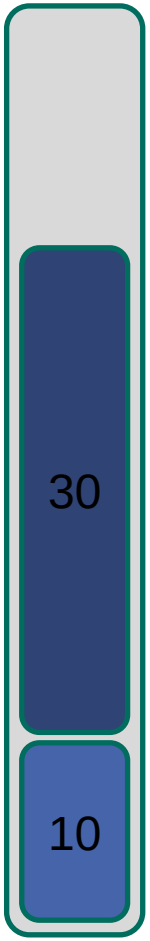
Wähle Gegenstand mit größtem Gewicht

Wert = 220 €

Gegenbeispiel kleinstes Gewicht



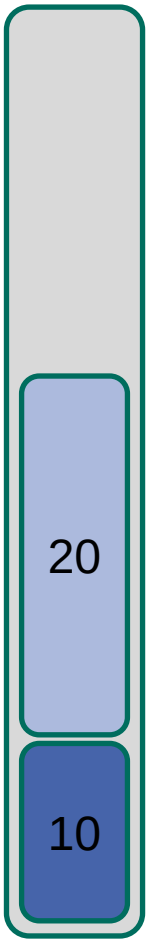
280€ 100€ 120€



Optimale Lösung?

Wert = 400 €

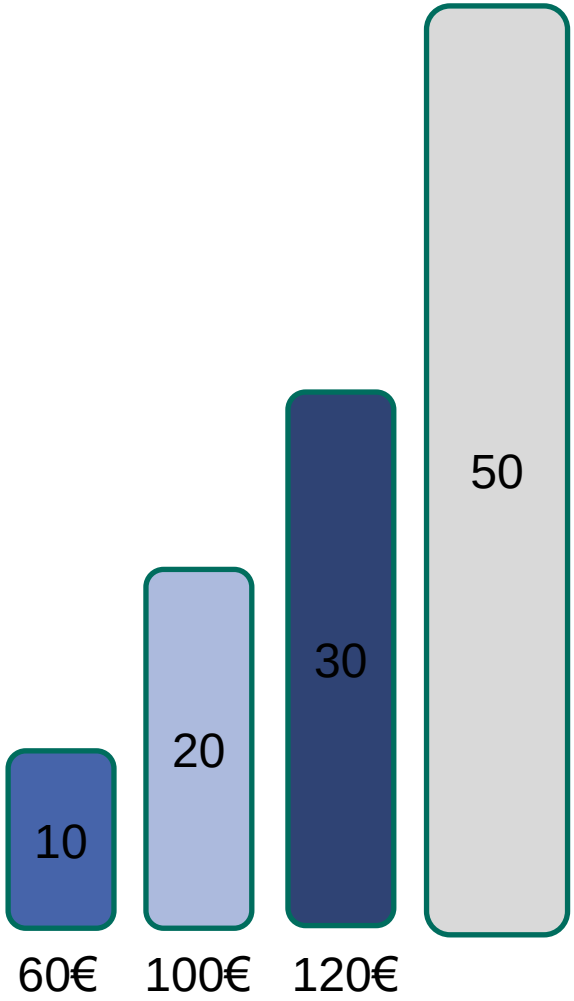
≠

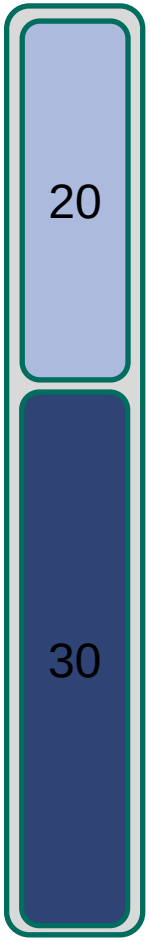


Wähle Gegenstand mit kleinstem Gewicht

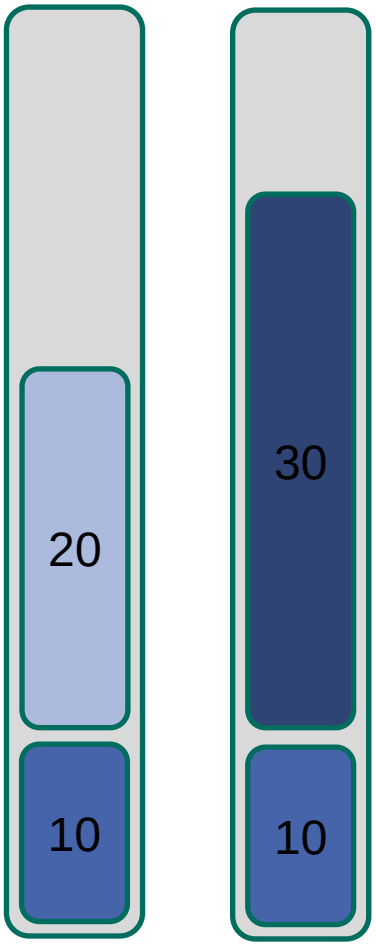
Wert = 380 €

Gegenbeispiel Wert pro Kilo





≠



Optimale Lösung?

Wert = 220 €

Greedy-Wahl ?
Wähle Gegenstand mit größtem Wert pro Kilo?

0-1-Rucksackproblem mit dynamischer Programmierung

- Gegeben
  - Sei S eine optimale Lösung für Rucksack mit W Pfund
  - Sei i der am höchsten nummerierte Gegenstand aus S

- Dann gilt
  - $S' = S - \{i\}$ ist optimale Lösung für Problem der Größe $W - w_i$ mit den Gegenständen $1 \ldots i - 1$
  - Wert dieser Lösung setzt sich zusammen aus
    - Wert des Gegenstand i v_i
    - Wert der Teilproblems S'

0-1-Rucksackproblem mit dynamischer Programmierung

- Sei $c[i, w]$ der Wert einer Lösung des Problems für Gegenstände $1,2, \ldots i$ mit dem maximalen Gewicht w

- Es gilt
  - $c[i, w] = 0$ falls $i = 0$ oder $w = 0$
  - $c[i, w] = c[i-1, w]$ falls $w_i > w$
  - $c[i, w] = \max(v_i + c[i-1, w-w_i], c[i-1, w])$ falls $i > 0$ und $w > w_i$

- Algorithmus $DYNAMIC - 0 - 1 - KNAPSACK(v, w, n, W)$ setzt diese Beziehung um
  - Eingabe maximales Gewicht W, Anzahl der Gegenstände n und Mengen $v = \{v_1, v_2, \ldots, v_n\}$ und $w = \{w_1, w_2, \ldots, w_n\}$
  - Ergebnisse werden in Tabelle $c[0..n, 0..W]$ gespeichert
    - $c[n, W]$ enthält maximalen Wert der in Rucksack passt

Pseudocode dynamisches *0-1*-Rucksackproblem

- $DYNAMIC - 0 - 1 - KNAPSACK(v, w, n, W)$

```
1  // sei c[0..n, 0..W] eine neue Tabelle
2  for w = 0 to W
3        c[0, w] = 0
4  for i = 1 to n
5        c[i, 0] = 0
6        for w = 1 to W
7              if w_i ≤ w
8                    if v_i + c[i − 1, w − w_i] > c[i − 1, w]
9                          c[i, w] = v_i + c[i − 1, w − w_i]
10                   else c[i, w] = c[i − 1, w]
11             else c[i, w] = c[i − 1, w]
```

Initialisierung des Feldes

Reihenweise Füllung der Tabelle

Komplexität *0-1*-Rucksackproblem mit dynamischer Programmierung

- Tabelle c wird reihenweise gefüllt
  - $(n+1)(W+1)$ Einträge
  - Jeweils $\theta(1)$ Zeit zum Füllen
    - ⟹ Komplexität $\theta(nW)$

- Rekonstruktion einer Lösung
  - Einträge von $c[n, W]$ aus zurückverfolgen
    - Wenn $c[i, w] = c[i-1, w]$, dann i nicht Teil der Lösung
      - Weiter mit $c[i-1, w]$
    - Sonst i Teil der Lösung
      - Weiter mit $c[i-1, w-w_i]$
  - Da n Reihen
    - ⟹ Komplexität $O(n)$

Das gebrochene Rucksackproblem

- Gegeben sei das gebrochene Rucksackproblem
  - Es existieren n Gegenstände, der $i-te$ Gegenstand sei v_i Euro wert und wiege w_i Kilo
  - Es können insgesamt W Kilo in einem Rucksack transportiert werden
    - W, v_i und w_i sind ganze Zahlen
  - Allerdings nun auch Teile eines Gegenstandes transportierbar
    - Also keine binäre $0-1$ –Entscheidung

- Gesucht
  - Welche Gegenstände oder Teile eines Gegenstandes müssen im Rucksack transportiert werden, damit der Wert der transportierten Gegenstände maximal ist?

Greedy-Strategy gebrochenes Rucksackproblem

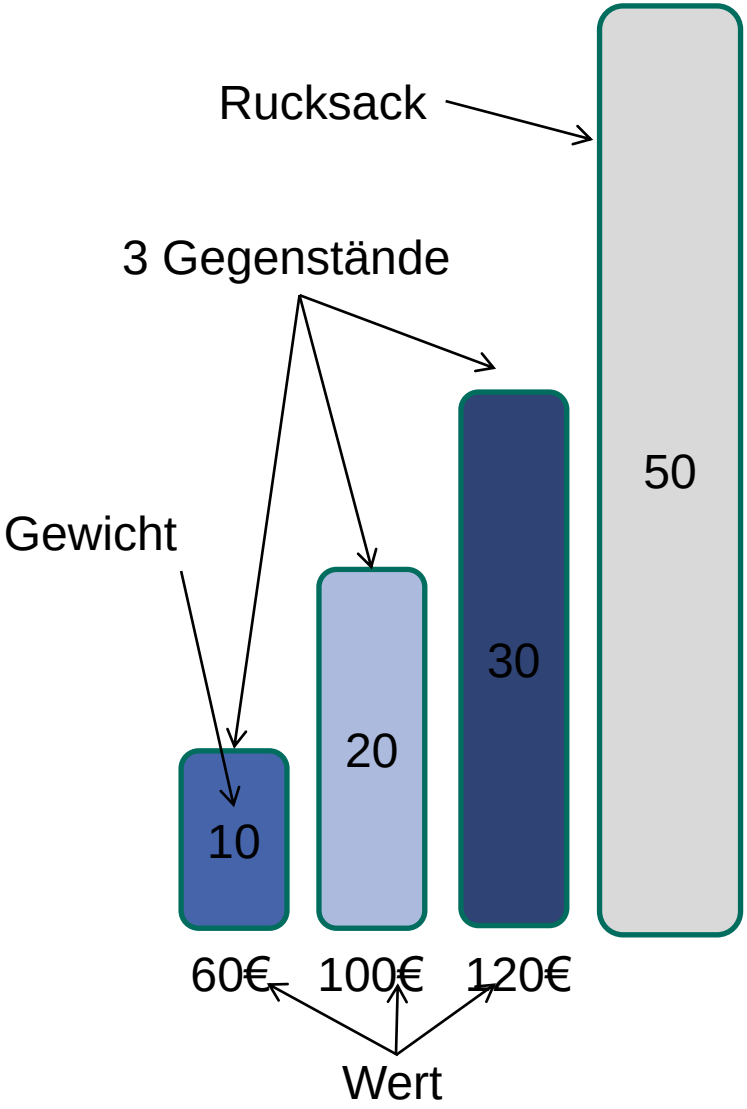
- Lösung mit Hilfe der Greedy-Strategy möglich!
  - Wähle Gegenstand mit größtem Wert pro Kilo
  - Wenn noch Restkapazität im Rucksack vorhanden
    - Wähle Gegenstand mit zweitgrößtem Wert pro Kilo
    - …..

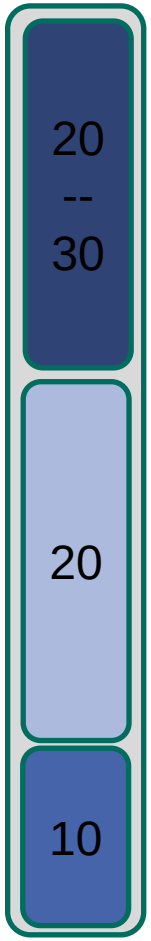
- Implementierung
  - Sortiere Gegenstände nach ihrem Wert pro Pfund
  - Fülle Rucksack in absteigender Reihenfolge mit Gegenstände nach ihrem Wert pro Pfund

  - Komplexität
    - Sortieren in $O(nlgn)$
    - Füllen des Rucksacks in $O(n)$

      ⟹ Komplexität $O(n\ lgn)$

Gebrochenes Rucksackproblem mit Greedy-Strategy





Optimale Lösung!

Zusammenfassung

- Für Optimierungsprobleme und deren Lösung wurden zwei generische Optimierungsstrategien vorgestellt
  - Dynamische Programmierung
    - Berechne alle möglichen Ergebnisse, speichere Teillösungen zur Wiederverwendung
    - Im allgemeinen zwei äquivalente Möglichkeiten, Ansatz basierend auf dynamischer Programmierung zu implementieren
      - Top-down Memoisation
      - Bottom-up-Methode
  - Greedy-Algorithmen
    - Anstatt alle Lösungen zu berechnen können wir Greedy-Wahl treffen
    - Kann zu nicht optimalen Ergebnissen führen

Literatur

- [Corm10] Thomas H. Cormen, Ch. Leiserson, R. Rivest, C. Stein, „Algorithmen – Eine Einführung “, Oldenburg, 3. Auflage, 2010, 1320 Seiten, ISBN 978-3-486-59002-9
- [MeSa10] Kurt Mehlhorn, Peter Sanders, „Algorithms and Data structures“, Springer, 300 Seiten, ISBN 978-3-540-77977-3