Algorithmen und Datenstrukturen

01 – Grundlagen

Prof. Dr. Rainer Gemulla

Universität Mannheim

HWS 2014

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

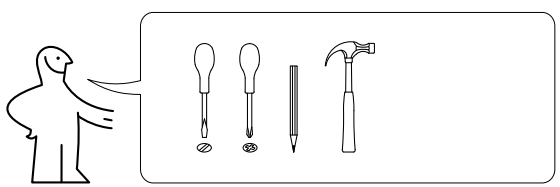
4. Wachstumsordnung

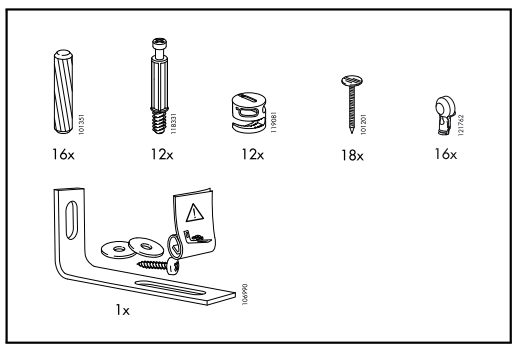
5. Rekurrenzen

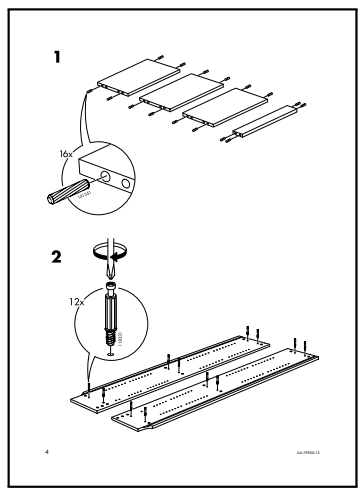
6. Wahrscheinlichkeit und Zufall

Algorithmus (allgemein)

- Bezieht sich auf einen **Akteur**, der bestimmte **Elementaraktionen** auf einer Menge möglicher **Objekte** ausführen kann
- Ist eine **Handlungsvorschrift** für diesen Akteur, für gegebene Eingabeobjekte eine eindeutig bestimmte Folge von Elementaraktionen auszuführen
- Beispiel
  - IKEA-Bauanleitung
  - Kochrezept







Algorithmus (hier)

- Akteur = Rechner, Objekte = Daten
  - Arbeitet taktweise
  - Liest **Eingabedaten** und speichert sie ab
  - Führt auf abgespeicherten Daten Operationen aus einer vorgegebenen Menge von **Elementaroperationen** aus (inklusive einer STOPP-Operation)
  - Kann abgespeicherte Daten **ausgeben**
- Algorithmus
  - Algorithmus A ist **Berechnungsvorschrift** an den Rechner
  - Kann verschiedenen dargestellt werden: Text, Pseudo-Code, Programm, ...
  - Für gegebene Eingabedaten x wird eine eindeutig bestimmte Folge von **Rechenschritten** (jeweils eine Elementaroperation) ausgeführt
  - Folge heißt **Berechnung** von A auf x
  - Berechnungen können *endlich* oder *unendlich* sein
- Ziel des Entwurfs von Algorithmen ist das Lösen von Berechnungsproblemen

Datenstruktur

Eine **Datenstruktur** ist eine Art und Weise, Daten abzulegen und zu organisieren so dass *Zugriff* und *Modifikationen* ermöglicht werden.

- Darstellung der Eingaben, Hilfsdaten, Ergebnisse
- Darstellung beinflusst Effizienz erheblich
  - Telefonbuch nach Namen sortiert
  - Telefonbuch nach Telefonnummern sortiert
- Keine Datenstruktur ist gut für alle Zwecke

Berechnungsprobleme

Ein **Berechnungsproblem** $\Pi \subseteq X \times Y$ ist eine Relation zwischen einer Menge X gültiger Eingabedaten und einer Menge Y gültiger Ausgabedaten. $(x, y) \in \Pi$ bedeutet: y is Lösung zur Eingabe x bzgl. Π.

Beispiel: Sortierproblem Π_S

- **Eingabe:** Folge von n Elementen $\langle a_1, a_2, \ldots, a_n \rangle$ aus einer geordneten Menge M
- **Ausgabe:** Permutation (Umordnung) $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ der Elemente so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$, $\pi \in \mathcal{S}_n$
- Beispieleingabe $\langle 31, 41, 59, 26, 41, 58 \rangle$; genannt **Instanz** des Sortierproblems
- Beispielausgabe $\langle 26, 31, 41, 41, 58, 59 \rangle$
- $(\langle 31, 41, 59, 26, 41, 58 \rangle, \langle 26, 31, 41, 41, 58, 59 \rangle) \in \Pi_S$

Lösen von Berechnungsproblemen durch Algorithmen

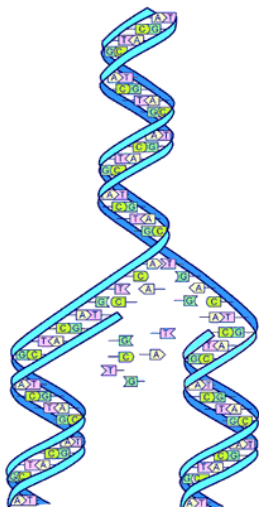
Ein Algorithmus A **löst** ein gegebenes Berechnungsproblem $\Pi \subseteq X \times Y$, falls

- A bezieht sich auf eine Vorschrift, wie Eingaben $x \in X$ im Rechner abgespeichert werden
- A bezieht sich auf eine Vorschrift, wie Ausgaben $y \in Y$ durch den Rechner ausgegeben werden
- Auf *jeder* Eingabe $x \in X$ stoppt der Algorithmus nach *endlich vielen* Takten und produziert dabei eine Ausgabe $y = A(x) \in Y$, für die gilt $(x, y) \in \Pi$

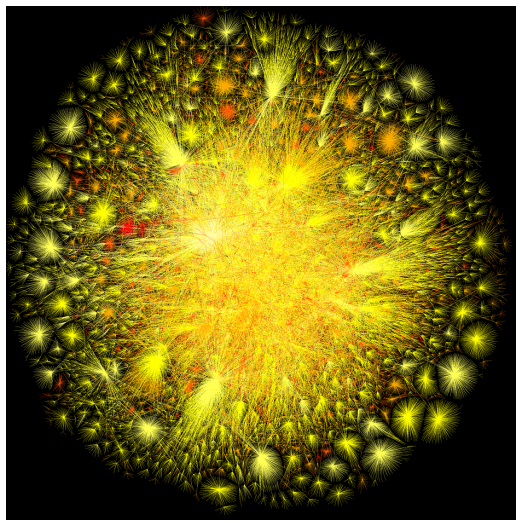
Wir sagen dann, dass A **korrekt** ist.

Algorithmen überall

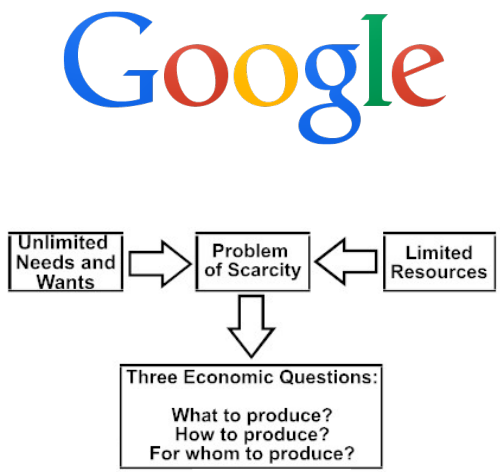
- Menschliche DNA sequenzieren und analysieren
- Internet-Pakete vom Absender zum Empfänger senden
- Das World Wide Web durchsuchen
- Knappe Ressourcen gut einsetzen
- Datensicherheit garantieren
- ...

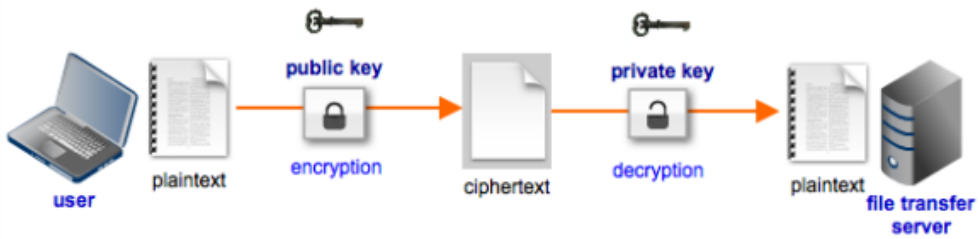


Human Genome Project



The Opte Project





Charakteristiken algorithmischer Probleme

1. Viele mögliche Kandidaten für Algorithmen
   - Aber diese lösen oftmals das Problem nicht
   - Korrekte Algorithmen manchmal schwer zu finden
   - Korrekte Lösungen sind unterschiedlich gut
   - Welcher Algorithmus ist der "beste"?
2. Praktische Anwendungen
   - Berechnungsprobleme sind oft Abstraktionen oder Verallgemeinerungen von praktischen Problemen

Beispiel: Kürzeste-Wege-Problem

- Eingabe: Graph G, Quellknoten s, Zielknoten t
- Ausgabe: Kürzester Pfad von s nach t
- Navigationsystem: Knoten = Orte, Kanten = Straßen
- Routing im Internet: Knoten = Rechner, Kanten = physische Kommunikationskanäle

Kosten von Berechnungen

Gegeben ein Algorithmus A, der sich auf Eingaben $x \in X$ bezieht.

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

Wann ist ein Algorithmus effizient?

- Kosten sind eingabeabhängig...
- Idee: Analyse des Verhaltens von Algorithmen in Abhängigkeit der Größe der Eingaben

Eingaben und Eingabelängen

Eingaben $x \in X$ wird in der Regel eine Eingabelänge $|x| \in \mathbb{N}$ zugeordnet. Man erhält Partition

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

wobei $X_n = \{ x \in X \mid |x| = n \}$.

Beispiele

- Eingaben $x \in \mathbb{N}$
  - $|x| = \lfloor \log_2(x) \rfloor + 1 =$ Bitlänge
- Eingabe $m \times n$ Matrizen M über $\{0, 1\}$
  - $|M| = m \cdot n =$ Anzahl der Elemente
- Eingabe ungerichtete Graphen $G = (V, E)$
  - $|G| = |V| =$ Anzahl Knoten,
  - $|G| = |E| =$ Anzahl Kanten oder
  - $|G| = |V| + |E|$

  je nach Anwendungszusammenhang

Kostenverhalten von Algorithmen

Die Kosten eines Algorithmus hängen oft nicht nur von der Größe der Eingabe ab, sondern auch von Ihrem Wert. Wie können wir das Laufzeitverhalten eines Algorithmus A beschreiben?

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

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

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

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

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

$$time_A(n) = \mathbb{E}_{x \in \mathbb{P}_n X_n} [time_A(x)]$$

für eine Wahrscheinlichkeitsverteilung $\mathbb{P}_n$ über Eingaben X_n (z.B. Gleichverteilung)

Algorithmen als Technologie

Zwei Algorithmen für das Sortierproblem

- Eingabegröße sei Anzahl n der zu sortierenden Elemente
- Insertion Sort, $time(n) \approx c_1 n^2$ Instruktionen
- Merge Sort, $time(n) \approx c_2 n \lg n$ Instruktionen
- Schneller Computer A, 10 Mrd. Instruktionen/s, Insertion Sort von sehr gutem Programmierer ($c_1 = 2$)
- Langsamer Computer B, 10 M Instruktionen/s, Merge Sort von durchschnittlichem Programmierer ($c_2 = 50$)
- Welcher Rechner sortiert 10M Elemente schneller?
  - A: $\frac{2(10^7)^2 \text{ Instruktionen}}{10^{10} \text{ Instruktionen/s}} = 20\,000\,\text{s} > 5.5\,\text{h}$
  - B: $\frac{50 \cdot 10^7 \lg 10^7 \text{ Instruktionen}}{10^7 \text{ Instruktionen/s}} \approx 1\,163\,\text{s} < 20\,\text{min}$

Algorithmen können die Kosten des Problemlösens dramatisch beeinflussen. Wir sehen sie – ebenso wie auch Hardware – als eine wichtige Technologie an.

Entwurf und Analyse von Algorithmen

Algorithmen entwerfen und analysieren heißt

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

Inhalt dieser Vorlesung

- Algorithmen für wichtige Probleme (z.B. Sortieren)
- Grundlegende Datenstrukturen (z.B. Heaps)
- Techniken zum Problemlösen und Algorithmenentwurf (z.B. Teile-und-Herrsche)
- Analyse der Eigenschaften von Algorithmen (v.a. Laufzeitverhalten)
- Korrektheitsbeweise

Ziel: Probleme selbst lösen, Lösungen beurteilen.

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

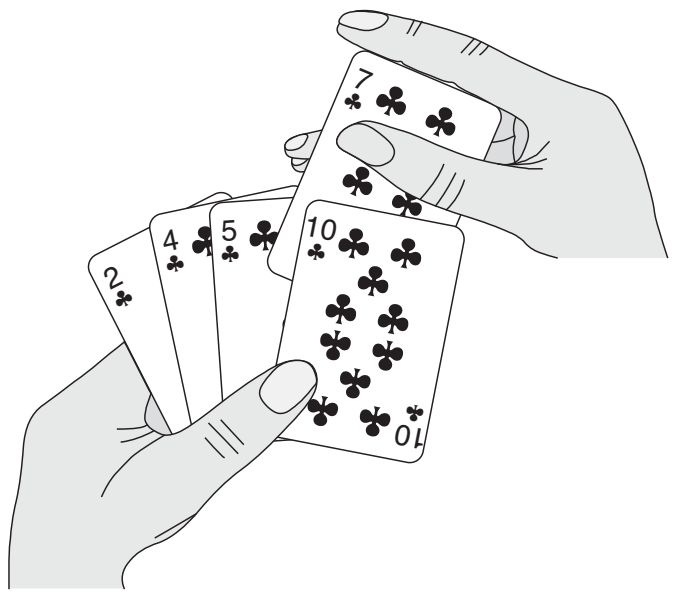
6. Wahrscheinlichkeit und Zufall

Sortieren durch Einfügen (Idee)

Wie beim Kartenspielen

1. Nimm die erste Karte auf
2. Füge jede weitere Karte von *rechts* so in die bereits aufgenommenen Karten ein, dass die Kartenfolge auf der Hand immer sortiert ist

Merke: Karten müssen im zweiten Schritt verschoben werden, damit sie sich nicht verdecken.



Pseudo-Code

Mit **Pseudo-Code** können wir Algorithmen kompakt angeben.

- // kodiert einen Kommentar
- $\leftarrow$ kodiert eine Zuweisung
- $[\cdot]$ ist ein Array-Zugriff
- Datenstrukturen haben Eigenschaften/Attribute; über „Punkt“ abgerufen (z.B. $A.length$)
- Wir verwenden Kontrollstrukturen wie **if-else**, **while**, **for**, ...
- Einrückungen entsprechen Blöcken/Rümpfen
- Zähler aus **for**-Schleifen behalten ihren Wert nach Beenden der Schleife verwendbar (erster Wert außerhalb der Schleifengrenzen)

Beispiel

```
1: for i ← 1 to 10                    // Wird 11x ausgeführt
2:     A[i] ← 0                       // Wird 10x ausgeführt
3: print(i)                           // Ausgabe: 11
```

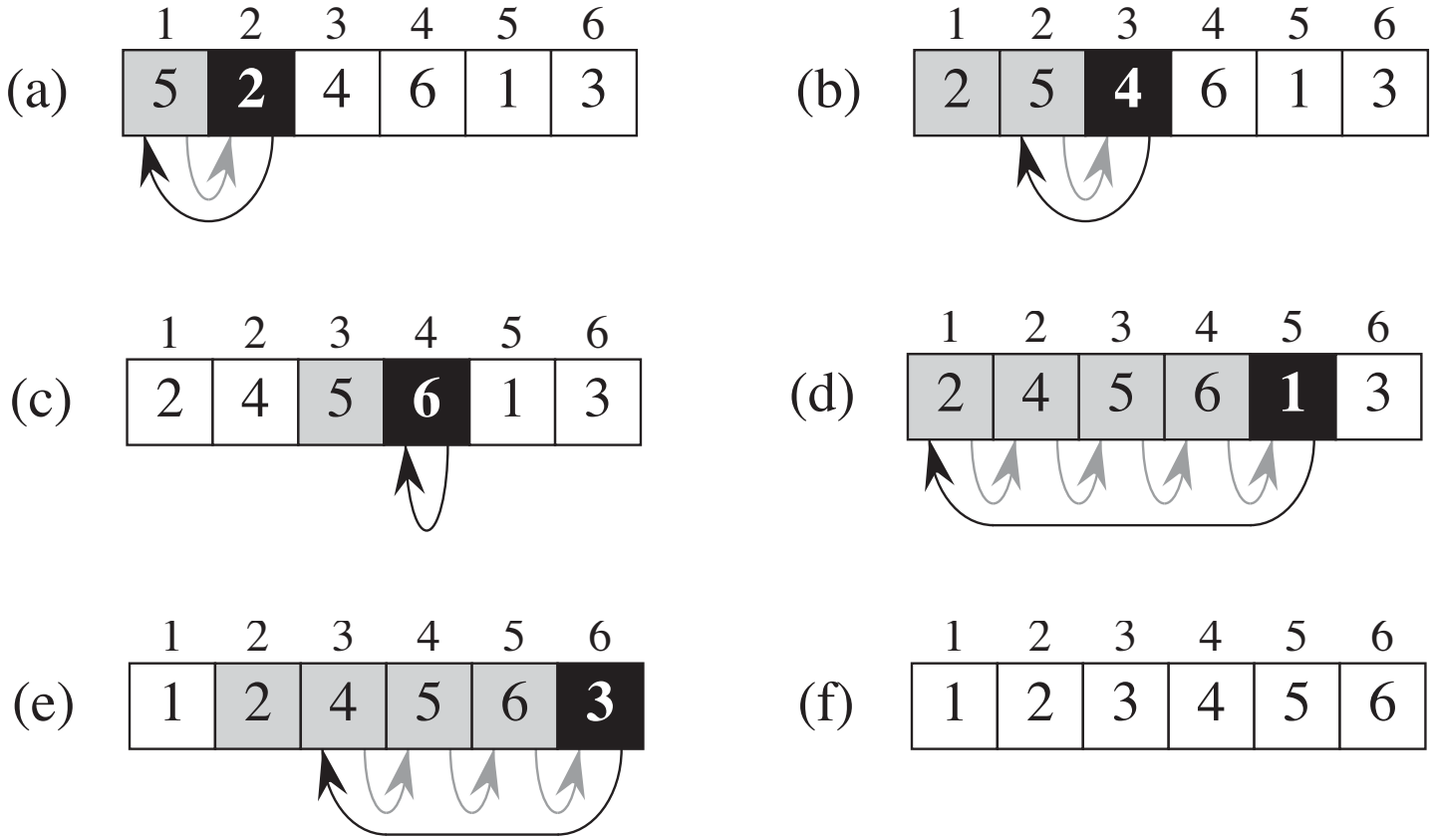
Insertion Sort (Pseudo-Code)

INSERTION-SORT(A)

```
1: for j ← 2 to A.length
2:     key ← A[j]
3:     // Insert A[j] into the sorted sequence A[1..j − 1]
4:     i ← j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] ← A[i]
7:         i ← i − 1
8:     A[i + 1] ← key
```

- Löst das Sortierproblem Π_S auf $a_1, \cdots, a_n \in (M, \leq)$
- Eingabe: $A.length = n$, $A[i] = a_i$, $i = 1, \ldots, n$
- Ausgabe $A[i] = a_{\pi(i)}$, $i = 1, \ldots, n$, mit $a_{\pi(1)} \leq \cdots \leq a_{\pi(n)}$.

Beispiel: $\langle 5, 2, 4, 6, 1, 3 \rangle$



Korrektheitsbeweis (1)

Wie kann man zeigen, dass INSERTION-SORT das Sortierproblem löst?

Beobachtung

Am Beginn jeder Iteration der äußeren Schleife (Z. 1–8) gilt: Das Teilfeld $A[1\,..\,j-1]$ *enthält die Elemente, die ursprünglich in* $A[1\,..\,j-1]$ *standen, aber in sortierter Reihenfolge.*

So eine Aussage nennt man **Schleifeninvariante**. Zu zeigen:

1. **Induktionsanfang**: Gilt vor der ersten Iteration der Schleife
2. **Induktionsschritt**: Wenn Schleifeninvariante vor einer Iteration gilt, dann auch vor der nächsten Iteration
3. **Terminierung**: Nützlich nach Ablauf der Schleife

Korrektheitsbeweis (2)

Am Beginn jeder Iteration der äußeren Schleife (Z. 1–8) gilt: (i) Das Teilfeld $A[1\,..\,j-1]$ *enthält die Elemente, die ursprünglich in* $A[1\,..\,j-1]$ *standen, aber (ii) in sortierter Reihenfolge.*

Induktionsanfang

- Zu Beginn der Schleife ist $j = 2$ und $A[1\,..\,j-1] = A[1]$, welches (i) noch den ursprünglichen Wert trägt und (ii) trivialerweise sortiert ist

Korrektheitsbeweis (3)

Am Beginn jeder Iteration der äußeren Schleife (Z. 1–8) gilt: (i) Das Teilfeld $A[1\,..\,j-1]$ *enthält die Elemente, die ursprünglich in* $A[1\,..\,j-1]$ *standen, aber (ii) in sortierter Reihenfolge.*

Induktionsschritt (informal)

- Induktionsannahme: Schleifeninvariante gilt für $2 \leq j \leq n$
- Zu zeigen: Teilfeld $A[1\,..\,j]$ erfüllt Schleifeninvariante nach Ausführung des Schleifenkörpers (aber vor Erhöhung von j)
  - (i) $A[j]$ trägt seinen ursprünglichen Wert. Z. 4–7 verschieben Elemente $A[j-1]$, $A[j-2]$, usw. eine Position nach rechts, bis die richtige Position für $A[j]$ gefunden ist; dort wird $A[j]$ dann eingefügt. $A[1\,..\,j]$ wird umgeordnet, behält aber seine Elemente
  - (ii) Sei s_j die Einfügeposition von $A[j]$. Elemente $A[1\,..\,s_j-1]$ werden nicht verändert; sie bleiben sortiert und sind kleiner als $A[j]$. Elemente $A[s_j\,..\,j-1]$ werden nach rechts verschoben; sie behalten ihre sortierte Reihenfolge und sind allesamt größer als $A[j]$. Somit ist $A[1\,..\,j]$ sortiert

Korrektheitsbeweis (4)

Am Beginn jeder Iteration der äußeren Schleife (Z. 1–8) gilt: (i) Das Teilfeld $A[1\,..\,j-1]$ enthält die Elemente, die ursprünglich in $A[1\,..\,j-1]$ standen, aber (ii) in sortierter Reihenfolge.

Terminierung

- Schleife terminiert mit $j = n + 1$
- Somit enthält $A[1\,..\,j-1] = A[1\,..\,n]$ die Elemente, die ursprünglich in $A[1\,..\,n]$ standen, aber in sortierter Reihenfolge
- $A[1\,..\,n]$ ist das gesamte Array

Somit haben wir bewiesen:

Theorem

INSERTION-SORT *löst das Sortierproblem.*

Kostenanalyse (1)

Einfaches Modell: Wir weisen jedem Schritt Kosten zu und zählen die Ausführungshäufigkeit. Sei t_j die Anzahl der Ausführung des Schleifentests in Z. 5 für ein gegebenes j.

INSERTION-SORT(A)	Kosten	Anzahl
1: **for** $j \leftarrow 2$ **to** $A.length$	c_1	n
2: $key \leftarrow A[j]$	c_2	$n-1$
3: // Insert $A[j]$ into the sorted // sequence $A[1..j-1]$	0	$n-1$
4: $i \leftarrow j-1$	c_4	$n-1$
5: **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6: $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7: $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8: $A[i+1] \leftarrow key$	c_8	$n-1$

Kostenanalyse (2)

Die Gesamtlaufzeit (Kosten · Anzahl) beträgt somit

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) \\ &\quad + c_5 \sum_{j=2}^{n} t_j \\ &\quad + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_8(n-1). \end{aligned}$$

- Was sagt uns diese Formel?
- Was machen wir mit t_j?
  - Ist eingabeabhängig
  - Entspricht 1 + Anzahl der nach rechts verschobenen Elemente
  - Wir bestimmen: Best Case-Laufzeit und Worst Case-Laufzeit

Kostenanalyse: Best Case

Den Best Case erhalten wir, wenn die Eingabefolge bereits sortiert ist. Dann ist $t_j = 1$ für $2 \leq j \leq n$. Aus

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \end{aligned}$$

wird dann

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Wir erhalten eine Funktion der Form $T(n) = an + b$.
$\Rightarrow$ $T(n)$ ist **linear** in n.

Kostenanalyse: Worst Case (1)

Den Worst Case erhalten wir, wenn die Eingabefolge umgekehrt sortiert ist. Dann ist $t_j = j$ für $2 \leq j \leq n$. Mit $\sum_{j=1}^{n} j = n(n+1)/2$:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\
&\quad + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \\
&= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\
&\quad + (c_6 + c_7) \frac{n(n-1)}{2} + c_8(n-1) \\
&= \frac{c_5 + c_6 + c_7}{2} n^2 + (c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8) n \\
&\quad - (c_2 + c_4 + c_5 + c_8).
\end{aligned}
$$

Wir erhalten eine Funktion der Form $T(n) = an^2 + bn + c$.
$\Rightarrow$ $T(n)$ ist **quadratisch** in n.

Kostenanalyse: Worst Case (2)

Wir können jetzt verschiedene Kosten für die einzelnen Operationen ansetzen.

$$T(n) = \frac{c_5 + c_6 + c_7}{2} n^2 + (c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

- Anzahl der Rechenschritte ($c_1 = c_2 = \cdots = c_8 = 1$)

$$T(n) = \frac{3}{2} n^2 + \frac{7}{2} n - 4$$

- Anzahl der Elementvergleiche ($c_5 = 1$, Rest 0)

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

Worst Case-Laufzeitanalyse

Wir konzentrieren uns oft auf die Analyse der Worst Case-Laufzeit.

Gründe

- Garantierte obere Laufzeitschranke für jede Eingabe
- Tritt für einige Algorithmen oft auf (z.B. oft: bei Suche nach Element, welches nicht vorhanden ist)
- Meist einfacher zu bestimmen als Average Case-Laufzeit
- Average Case ist häufig ähnlich „schlecht“ wie Worst Case; z.B.
  - INSERTION-SORT mit zufällig geordneter Eingabe
  - Im Schnitt ist $A[j]$ kleiner als die Hälfte der Elemente in $A[1\,..\,j-1]$ (und größer als die andere Hälfte)
  - D.h. $t_j \approx j/2$
  - $T(n)$ ist immer noch quadratisch

Wachstumsordnung

- Analyse und Bewertung wird durch genaue Berechnung der Anzahl der Schritte erschwert
- Da wir nicht an den Details einer Funktion $an^2 + bn + c$ für Konstanten a, b, c interessiert sind, wollen wir die Konstanten ignorieren
- Da des weiteren n^2 schneller wächst als n oder jede Konstante, vernachlässigen wir auch die kleineren Terme
- Wir sind also nur an der **Wachstumsordnung** (oder Wachstumsrate) n^2 interessiert. Hierfür führen wir die Notation $\Theta(n^2)$ (zunächst informell) ein
- Wir sagen also beispielsweise für INSERTION-SORT, daß er eine Worst Case-Laufzeitkomplexität von $\Theta(n^2)$ hat (sprich: „Theta n Quadrat“)
- In dieser Notation sehen wir schnell, daß ein Algorithmus mit Laufzeitkomplexität $\Theta(n^3)$ „schlechter“ ist als einer mit $\Theta(n^2)$

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Teile und Herrsche

- Algorithmen zu entwerfen ist eine Kunst
- Es gibt nur wenige Verfahren
- Ein Verfahren ist **Teile und Herrsche** (engl. *divide and conquer*, lat. *divide et impera*)
- Probleme werden solange rekursive in kleiner/einfachere Teilprobleme zerlegt, bis sie klein genug sind, um sie direkt zu lösen („beherrschen“)
  1. **Teile**: Zerlege ursprüngliches Problem in Teilprobleme
  2. **Herrsche**: Löse die Teilprobleme rekursiv
  3. **Kombiniere**: Füge die Teillösungen zusammen
- Warum?
  - Schwierige Probleme lösen
  - Kann helfen, effiziente Algorithmen zu finden
  - Teilprobleme können parallel abgearbeitet werden (von uns nicht betrachtet)
  - Bessere Ausnutzung der Speicherhierarchie (Lokalitätsprinzip)

Sortieren durch Mischen

Das Teile-und-Herrsche-Prinzip lässt sich auf das Sortierproblem anwenden. Eine Möglichkeit ist das **Sortieren durch Mischen** (*merge sort*):

1. **Teile**: Zerlege Folge der Länge n in zwei Teilfolgen der Länge $n/2$
2. **Herrsche**: Sortiere Teilfolgen rekursiv
3. **Kombiniere**: Füge sortierte Teilfolgen durch **Mischen** (*merge*) zu einer sortierten Folge zusammen

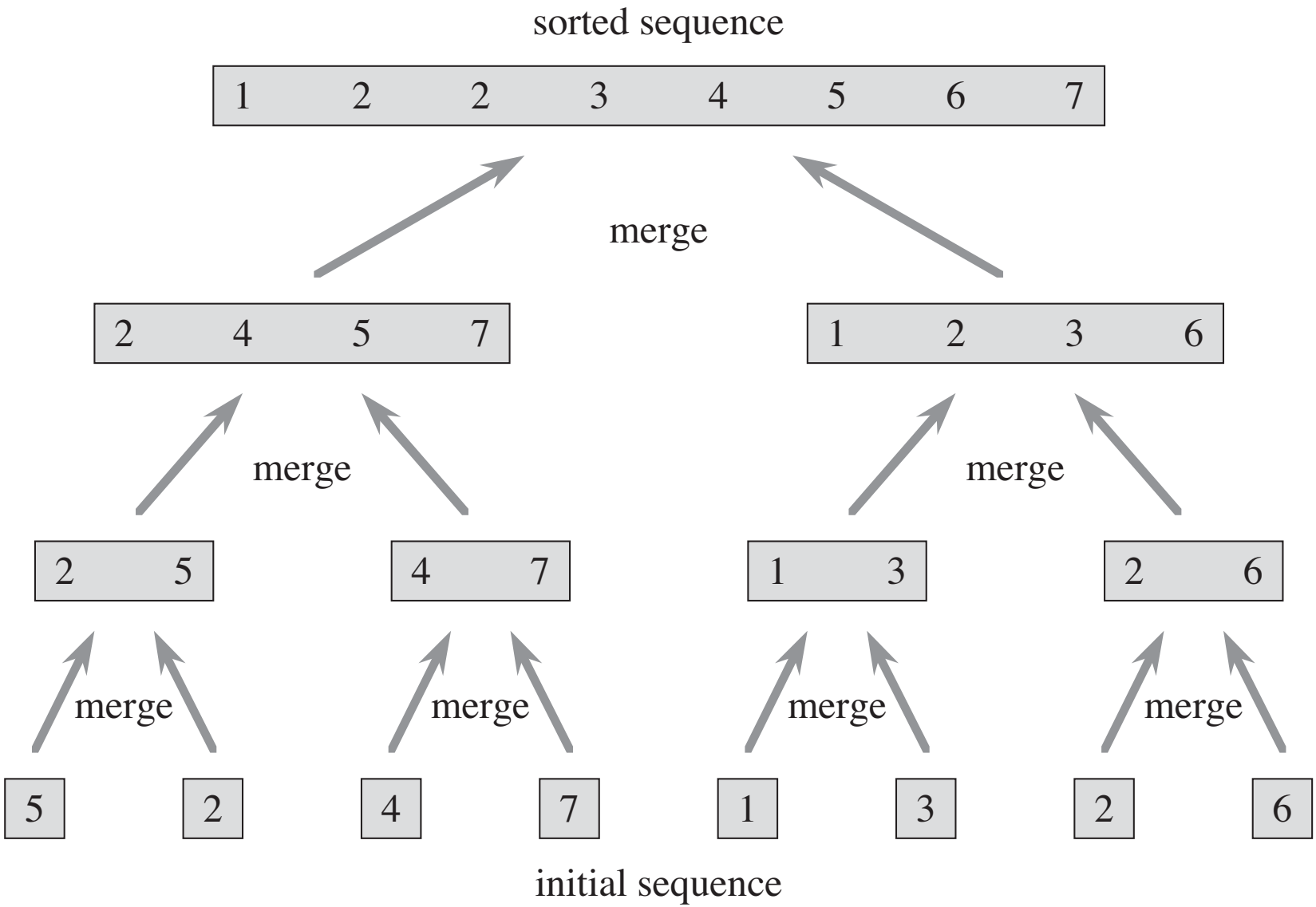
Beachte: Teilfolgen der Länge 0 oder 1 sind schon sortiert.

Sortieren durch Mischen (Pseudo-Code)

```
MERGE-SORT(A, p, r)                                  // Sortiert A[p .. r]
1: if p < r
2:     q ← ⌊(p + r)/2⌋
3:     MERGE-SORT(A, p, q)
4:     MERGE-SORT(A, q + 1, r)
5:     MERGE(A, p, q, r)
```

- Aufruf $\textsc{Merge-Sort}(A, 1, n)$
- $\textsc{Merge}(A, p, q, r)$ mischt die Teilfolgen $A[p \,..\, q]$ und $A[q + 1 \,..\, r]$
  - Aufwand $\Theta(n)$ mit $n = r - p + 1$
  - Kernidee: Wähle in jedem Schritt das jeweils kleinere Element der Teilfolgen als nächstes Element der zu konstruierenden Gesamtfolge

Beispiel: $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$



Analyse von Teile-und-Herrsche-Algorithmen

- Sind oft rekursiv
- Laufzeit daher am besten durch „rekursive“ Gleichungen beschrieben; genannt **rekurrente Gleichungen** oder **Rekurrenzen** beschreiben

Aufstellen der Rekurrenz

- Gesamtaufwand $T(n)$ für Problemgröße n
- Für kleine Probleme ($n < c$ für eine Konstante c) ist Aufwand konstant; notiert $\Theta(1)$
- Sonst Aufteilung des Problems in a Teilprobleme der Größe n/b
  - Aufwand $D(n)$ für das Aufteilen
  - Aufwand $aT(n/b)$ für das Lösen der Teilprobleme
  - Aufwand $C(n)$ für das Kombinieren
- Somit erhalten wir

$$T(n) = \begin{cases} \Theta(1) & \text{wenn } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{wenn } n > c \end{cases}$$

Analyse von Merge Sort (1)

- Annahme: n ist Zweierpotenz, d.h. $n = 2^k$ für ein geeignetes k
- Teilfolgen haben dann immer genau Größe n/2
- Wachstumsordnung durch Annahme nicht beeinflusst

Aufstellen der Rekurrenz

$$T(n) = \begin{cases} \Theta(1) & \text{wenn } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{wenn } n > c \end{cases}$$

- **Teile**: Mitte der Folge wird in konstanter Zeit ermittelt $\rightarrow D(n) = \Theta(1)$
- **Herrsche**: Wir teilen in zwei Teile der Größe $n/2 \rightarrow 2T(n/2)$
- **Kombiniere**: MERGE hat linearen Aufwand $\rightarrow C(n) = \Theta(n)$

Analyse von Merge Sort (2)

Da $\Theta(1) + \Theta(n)$ immer noch linearem Aufwand entspricht, können wir diese Summe durch $\Theta(n)$ ersetzen. Wir erhalten:

$$T(n) = \begin{cases} \Theta(1) & \text{wenn } n \leq c \\ 2T(n/2) + \Theta(n) & \text{wenn } n > c. \end{cases}$$

Die Lösung dieser Rekurrenz ist $\Theta(n \lg n)$.

- MERGE-SORT mit Laufzeit $\Theta(n \lg n)$ ist für große n besser als
- INSERTION-SORT mit Worst Case-Laufzeit $\Theta(n^2)$.

n	$16 = 2^4$	$256 = 2^8$	$1\,048\,576 = 2^{20}$
$n \lg n$	64	10 240	20 971 520
n^2	256	1 048 576	1 099 511 627 776

(Letzte Spalte: Telefonbuch von Köln sortieren.)

Zusammenfassung

- Zwei Lösungen für das Sortierproblem
- Merge-Sort schneller als Insertion-Sort
- Laufzeit quantifizierbar in $\Theta(\cdot)$
- Teile und Herrsche als Methode des Algorithmenentwurfs

Nachfolgend

- Was bedeutet $\Theta(\cdot)$ genau?
- Wie löst man Rekurrenzen?

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung
4.1 Die Θ-Notation
4.2 Die O- und Ω-Notation
4.3 Asymptotische Notation in Gleichungen
4.4 Die o- und ω-Notation
4.5 Vergleichen von Funktionen

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Wachstumsordnung (1)

- Zählen der Schritte
  - Wie schnell ist ein Algorithmus?
  - Welcher Algorithmus ist besser?
- Oft sehr aufwendig $\rightarrow$ Wachstumsordnung
- Nur der führende Term der Laufzeitformel $T(n)$ wird betrachtet
  - Alle anderen Terme werden ignoriert
  - Konstanter Koeffizient des führenden Terms wird auch ignoriert
- Beispiel: $T(n) = an^2 + bn + c$
  - Führender Term: an^2
  - Ohne Konstante: n^2
  - $T(n)$ wächst wie n^2 (aber nicht: $T(n)$ ist gleich n^2)
  - Notiert: Laufzeit ist $\Theta(n^2)$

Wachstumsordnung (2)

- Warum Wachstumsordnung?
  - Zählen der Schritte wird einfacher

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < c \\ 2T(n/2) + \Theta(n) & \text{sonst.} \end{cases}$$

  - Vergleichen der Laufzeiten wird einfacher
  - Treffen von maschinenunabhängigen Aussagen wird möglich
- Erreicht durch Abstraktion/Vereinfachung der Analyse und Bewertung der Laufzeit
  - Abschätzung der Anzahl der Schritte
  - Beschreibung der Anzahl der Schritte in Abhängigkeit der Problemgröße n
  - Weglassen von Details
  - Beschreibung des asymptotischen Verhaltens ($n \to \infty$) des Wachstums einer Funktion

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung
4.1 Die Θ-Notation
4.2 Die O- und Ω-Notation
4.3 Asymptotische Notation in Gleichungen
4.4 Die o- und ω-Notation
4.5 Vergleichen von Funktionen

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Die Θ-Notation (1)

Wir betrachten Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$.

Definition

Sei $g(n)$ eine Funktion. Wir definieren $\Theta(g(n))$ als die **Menge von Funktionen**

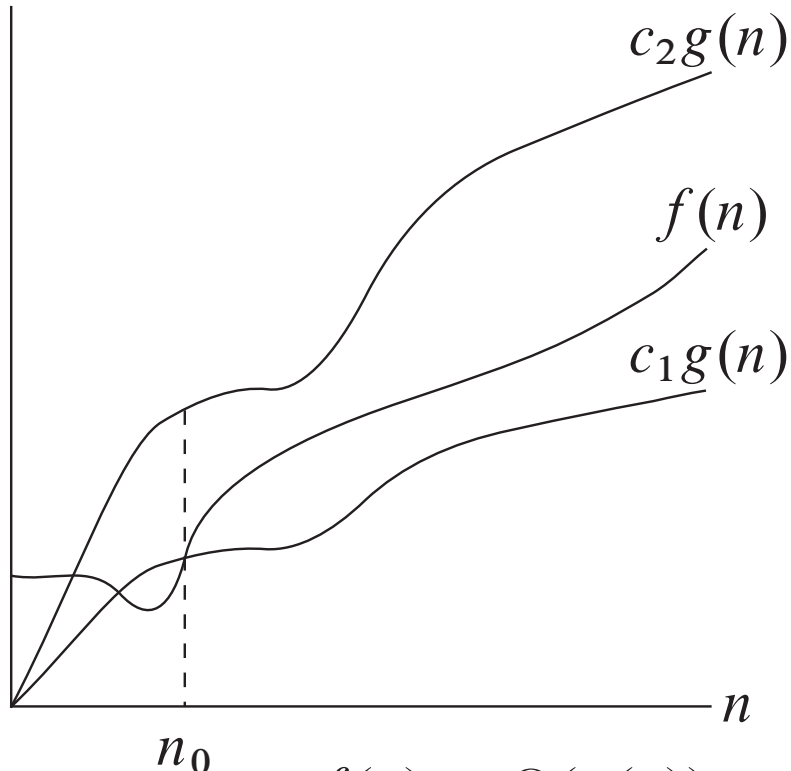
$$\Theta(g(n)) := \{ f(n) \mid \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0 : \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}.$$

- Wir schreiben $f(n) \in \Theta(g(n))$ oder $f(n) = \Theta(g(n))$
- Falls $f(n) \in \Theta(n)$, sagen wir: $f(n)$ **wächst asymptotisch genauso wie** $g(n)$

Die Θ-Notation (2)

Es gilt $f(n) \in \Theta(g(n))$, falls $f(n)$ zwischen $c_1 g(n)$ und $c_2 g(n)$ eingeklemmt werden kann.

- $0 < c_1 \leq c_2$ beliebig (aber: unabhängig von n)
- Nur für große $n \geq n_0$ (beliebig, unabhängig von n)



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

Beispiel (1)

Motiviert wurde Θ durch Weglassen von Konstanten und Termen niedriger Ordnung. Dass dies durchaus richtig ist, demonstrieren wir an einem Beispiel. Wir zeigen:

$$1/2n^2 - 3n = \Theta(n^2)$$

Dazu benötigen wir Konstanten c_1, c_2, n_0 mit

$$c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2$$

für alle $n \geq n_0$. Dividieren durch n^2 ergibt

$$c_1 \leq 1/2 - 3/n \leq c_2.$$

- Die rechte Gleichung gilt (bspw.) für alle $n \geq 1$, falls $c_2 \geq 1/2$.
- Die linke Gleichung gilt (bspw.) für alle $n \geq 7$, falls $c_1 \leq 1/14$.
- Mit $c_1 = 1/14$, $c_2 = 1/2$ und $n_0 = 7$ haben wir bewiesen, dass $1/2n^2 - 3n = \Theta(n^2)$.

Beispiel (2)

Wir zeigen jetzt:

$$6n^3 \neq \Theta(n^2).$$

Annahme: Es existieren $c_2, n_0 > 0$, so dass

$$6n^3 \leq c_2 n^2$$

für alle $n \geq n_0$. Daraus folgt aber

$$n \leq c_2/6$$

Das kann aber nicht sein, da n beliebig groß werden kann und c_2 eine Konstante sein muß. Widerspruch.

Beispiel (3)

Wir zeigen jetzt:

$$2n \neq \Theta(n^2).$$

Annahme: Es existieren $c_1, n_0 > 0$, so dass

$$c_1 n^2 \leq 2n$$

für alle $n \geq n_0$. Daraus folgt aber

$$n \leq 2/c_1$$

Das kann aber nicht sein, da n beliebig groß werden kann und c_1 eine Konstante sein muß. Widerspruch.

Allgemeineres Beispiel

Für

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

mit $a > 0$ gilt

$$f(n) = \Theta(n^2).$$

Dazu kann man bspw. die Konstanten

$$\begin{aligned} c_1 &= a/4 \\ c_2 &= 7a/4 \\ n_0 &= 2\max(|b|/a, \sqrt{|c|/a}) \end{aligned}$$

benutzen. (Übung)

Anmerkungen

- Falls $f(n) = \Theta(g(n))$, so ist $f(n)$ ab einem bestimmten $n > n_0$ nie weiter als einen konstanten Faktor von $g(n)$ entfernt
- Falls dies gilt, so sagt man auch, daß $g(n)$ eine **asymptotisch enge Grenze** für $f(n)$ ist
- Wir benutzen auch reelle Funktion $f, g : \mathbb{R} \to \mathbb{R}$ und erweitern die Θ-Notation entsprechend. Dann:
  - Für $f(n) = \Theta(g(n))$ muß $f(n)$ asymptotisch nicht negativ sein
  - Für jedes $g(n)$, das nicht asymptotisch nicht negativ ist, ist $\Theta(g(n)) = \emptyset$
  - Daher setzen wir voraus, daß alle $g(n)$, die innerhalb von $\Theta(\cdot)$ auftauchen, asymptotisch nicht negativ sind. Das gleiche gilt auch für andere asymptotische Notationen, die wir noch einführen werden

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung
4.1 Die Θ-Notation
4.2 Die O- und Ω-Notation
4.3 Asymptotische Notation in Gleichungen
4.4 Die o- und ω-Notation
4.5 Vergleichen von Funktionen

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Obere und untere Schranken

- $f(n) = \Theta(g(n))$ beschränkt
  - Funktion $f(n)$ von unten durch $g(n)$ bis auf eine Konstante
  - Funktion $f(n)$ von oben durch $g(n)$ bis auf eine Konstante
- Dies ist nicht für alle Funktionen möglich; z.B.
  - $f(n) = \begin{cases} n & \text{wenn } n \text{ gerade} \\ n^2 & \text{sonst} \end{cases}$
  - $f(n) \notin \Theta(n)$, da $n^2 \notin \Theta(n)$
  - $f(n) \notin \Theta(n^2)$, da $n \notin \Theta(n^2)$
- Daher führen wir Notation für obere und untere Schranken ein
  - $f(n) = \Omega(n)$ für die untere Schranke
  - $f(n) = O(n^2)$ für die obere Schranke

Die O-Notation (1)

Definition

Sei $g(n)$ eine Funktion. Wir definieren $O(g(n))$ als die **Menge von Funktionen**

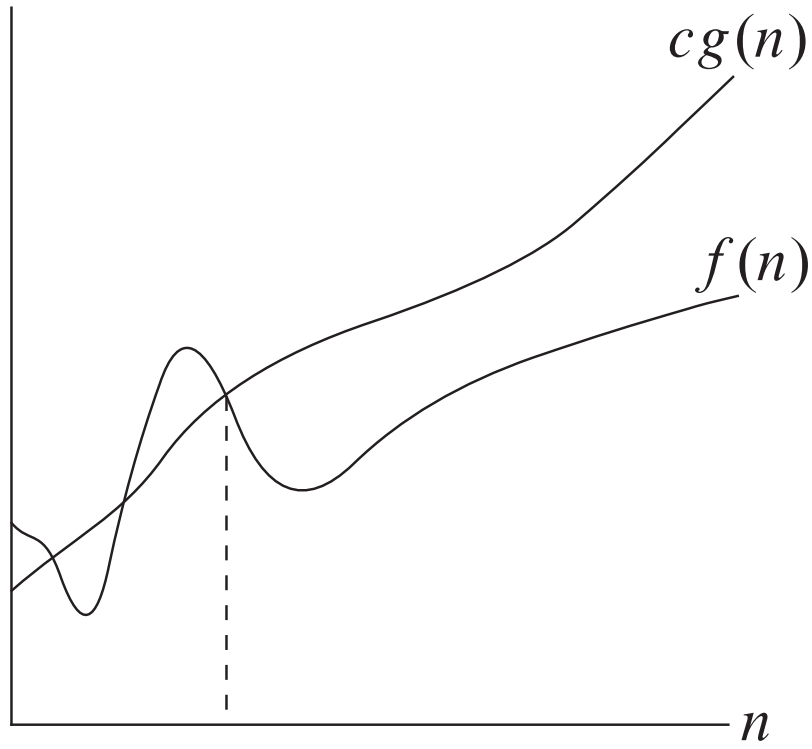
$$O(g(n)) := \{ f(n) \mid \exists c, n_0 > 0 : \forall n \geq n_0 : \\ 0 \leq f(n) \leq cg(n) \}.$$

- Sprich: "groß O" (*big-O*), oft vereinfachend auch nur "O"
- Verwendung für asymptotische obere Schranken
- Nützlich für einfache Worst Case-Abschätzungen
- Wir schreiben auch wieder $f(n) = O(g(n))$ und sagen: $f(n)$ **wächst asymptotisch nicht schneller als** $g(n)$
- Es gilt: $\Theta(g(n)) \subseteq O(g(n))$
  - Somit z.B. $an^2 + bn + c = O(n^2)$

Die O-Notation (2)

Es gilt $f(n) \in O(g(n))$, falls $f(n)$ nicht größer als $cg(n)$ wird.

- $0 < c$ beliebig (aber: unabhängig von n)
- Nur für große $n \geq n_0$ (beliebig, unabhängig von n)



Beispiel

- Wir wollen zeigen:

$$\textit{Für } f(n) = an + b \textit{ gilt } f(n) = O(n^2) \ .$$

- Zu finden sind c, n_0 mit $0 \leq an + b \leq cn^2$ für alle $n \geq n_0$.
- Setze $c = a + |b|$ und $n_0 = 1$. Dann gilt für alle $n \geq n_0$:

$$\begin{aligned} an + b &\leq an + |b| \\ &\leq an + |b|n \\ &= (a + |b|)n \\ &\leq (a + |b|)n^2 \\ &= cn^2 \end{aligned}$$

Beispiel: Obere Schranke für Insertion-Sort

Insertion-Sort(A)

```
1: for j ← 2 to A.length
2:     key ← A[j]
3:     // Insert A[j] into the sorted sequence A[1..j − 1]
4:     i ← j − 1
5:     while i > 0 and A[i] > key
6:         A[i + 1] ← A[i]
7:         i ← i − 1
8:     A[i + 1] ← key
```

Schnelle obere Schranke

- Die beiden Schleifen werden jeweils höchstens n-mal durchlaufen (eigentlich weniger oft)
- Die beiden Schleifen sind geschachtelt
- Das impliziert sofort für die Worst Case-Laufzeit: $T(n) = O(n^2)$

Die Ω-Notation (1)

Definition

Sei $g(n)$ eine Funktion. Wir definieren $\Omega(g(n))$ als die **Menge von Funktionen**

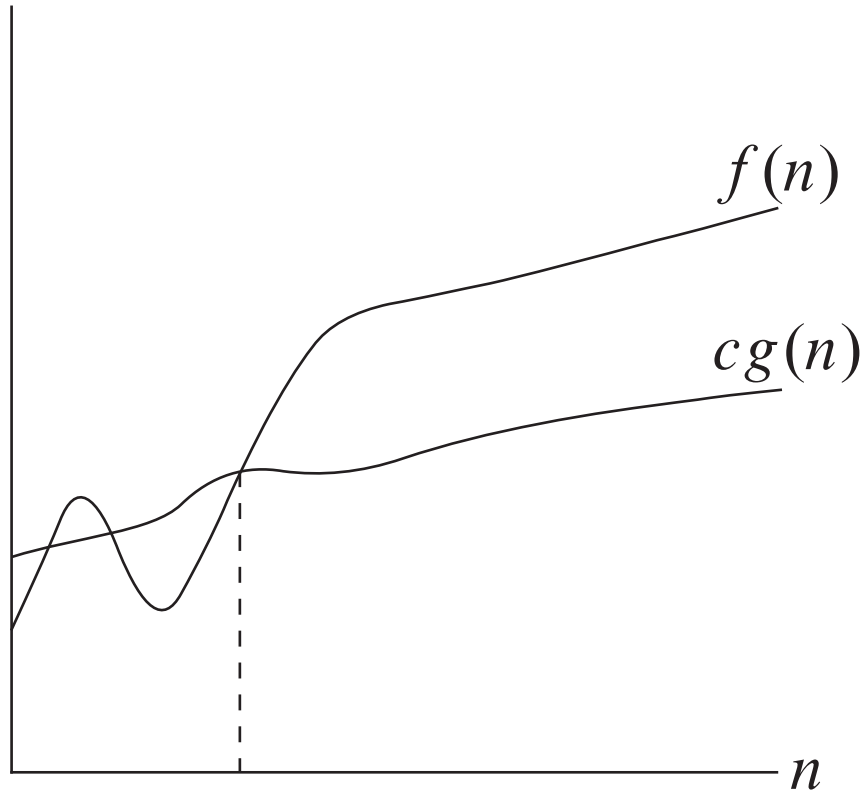
$$\Omega(g(n)) := \{\, f(n) \mid \exists c, n_0 > 0 : \forall n \geq n_0 : \\ 0 \leq cg(n) \leq f(n) \,\}.$$

- Sprich: "groß Omega" (*big-Omega*), oft vereinfachend nur Ω
- Verwendung für asymptotische untere Schranken
- Ermöglicht einfache Best Case-Abschätzungen
- Wir schreiben auch wieder $f(n) = \Omega(g(n))$ und sagen: $f(n)$ **wächst asymptotisch nicht langsamer als** $g(n)$
- Es gilt: $\Theta(g(n)) \subseteq \Omega(g(n))$
  - Somit z.B. $an^2 + bn + c = \Omega(n^2)$

Die Ω-Notation (2)

Es gilt $f(n) \in \Omega(g(n))$, falls $f(n)$ nicht kleiner als $cg(n)$ wird.

- $0 < c$ beliebig (aber: unabhängig von n)
- Nur für große $n \geq n_0$ (beliebig, unabhängig von n)



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

Konvention zur Verwendung von O und Ω

- Es ist technisch gesehen ein Fehler zu sagen, daß die Laufzeit von Insertion-Sort $O(n^2)$ ist
  - Die Laufzeit hängt noch von der konkreten Eingabefolge abhängt
  - Deutlicher: Die Laufzeit von INSERTION-SORT ist keine Funktion in n (sondern in A)
- Wir sehen über dieses technisches Detail hinweg
- Die Worst/Best/Average Case-Laufzeiten sind Funktionen in n
- Somit die Konvention: Wenn wir sagen, die Laufzeit eines Algorithmus ist
  - $O(f(n))$, meinen wir den Worst Case,
  - $\Omega(f(n))$, meinen wir den Best Case.
- Aber: Die folgenden Aussagen sind verschieden
  - Algorithmus A hat eine Laufzeit von $O(n^2)$
  - Die Worst Case-Laufzeit von A is $\Theta(n^2)$

Asymptotisch enge Schranken

Theorem

Für je zwei Funktionen $f(n)$ und $g(n)$ gilt:

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)).$$

(Beweis: Übung)

- Theorem erlaubt asymptotisch enge Schranken aus unteren und oberen Schranken
- Nicht immer möglich; Beispiel: Insertion-Sort
  - Best Case: $T(n) = \Omega(n)$
  - Worst Case: $T(n) = O(n^2)$

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung
4.1 Die Θ-Notation
4.2 Die O- und Ω-Notation
4.3 Asymptotische Notation in Gleichungen
4.4 Die o- und ω-Notation
4.5 Vergleichen von Funktionen

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Asymptotische Notation rechts

In der Einleitung haben wir bereits asymptotische Notationen in Gleichungen verwendet, z.B.

- $T(n) = \Theta(1) + \Theta(n)$
- $T(n) = 2T(n/2) + \Theta(n)$
- $T(n) = O(n^2)$

Was ist die Semantik?

- Schon bekannt: Falls auf der linken Seite der Gleichung eine Funktion und auf der rechten Seite asymptotische Notation steht, so ist dies als Mengenzugehörigkeit zu interpretieren.
- $T(n) = O(n^2)$ bedeutet also $T(n) \in O(n^2)$

Ausdruck mit asymptotischer Notation rechts (1)

Falls auf der linken Seite ein Funktion und auf der rechten Seite ein Ausdruck mit aymptotischer Notation steht, dann interpretieren wir die asymptotische Notation als eine anonyme Funktion f der entsprechenden Klasse.

Beispiel: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

- Interpretieren wir als

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

  für $f(n) \in \Theta(n)$
- Dabei ist $f(n)$ implizit existenzquantifiziert, d.h.

$$\exists f(n) \in \Theta(n) \colon 2n^2 + 3n + 1 = 2n^2 + f(n)$$

- Hier: $f(n) = 3n + 1$

Ausdruck mit asymptotischer Notation rechts (2)

Wichtig: Jedes Vorkommen einer asymptotischen Notation entspricht **genau einer** anonymen Funktion.

- Anzahl asymptotischer Notation = Anzahl anonyme Funktionen
- Auch: zwei gleiche asymptotische Ausdrücke $\rightarrow$ zwei Funktionen
- Auch: asymptotischer Ausdruck in Summe $\rightarrow$ eine Funktion

Extremes Beispiel

In

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

bezeichnet $O(i)$ **eine** Funktion. Damit ist dieser Ausdruck ungleich von folgendem Ausdruck

$$O(1) + O(2) + O(3) + \ldots + O(n).$$

(Dieser Ausdruck hat keine saubere Interpretation.)

Asymptotische Notationen auf beiden Seiten

Falls auf der linken und rechten Seite jeweils ein Ausdruck mit aymptotischer Notation steht, dann interpretieren wir: Für alle möglichen anonymen Funktionen links, existieren passende anonyme Funktionen rechts, so dass die Gleichung gilt.

Beispiel

- $2n^2 + \Theta(n) = \Theta(n^2)$
- Interpretiert als: Für alle $f(n) \in \Theta(n)$ existiert ein $g(n) \in \Theta(n^2)$ so dass

$$2n^2 + f(n) = g(n)$$

- Äquivalent

$$\forall f(n) \in \Theta(n) \colon \exists g(n) \in \Theta(n^2) \colon 2n^2 + f(n) = g(n)$$

Gleichungsketten

- Die beiden Beispielgleichungen zusammen:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

- Die erste Gleichung besagt, daß es eine Funktion $f(n) \in \Theta(n)$ gibt mit

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

- Die zweite Gleichung besagt, daß es für jedes $f(n) \in \Theta(n)$ ein $g(n) \in \Theta(n^2)$ gibt mit

$$2n^2 + f(n) = g(n)$$

- Diese beiden Aussagen implizieren:

$$\textit{Es gibt ein } g(n) \in \Theta(n^2) \textit{ mit } 2n^2 + 3n + 1 = g(n).$$

Dies ist genau das, was wir intuitiv erwarten würden.

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung
4.1 Die Θ-Notation
4.2 Die O- und Ω-Notation
4.3 Asymptotische Notation in Gleichungen
4.4 Die o- und ω-Notation
4.5 Vergleichen von Funktionen

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Asymptotisch nicht-enge Schranken

Betrachte:

$$\begin{aligned} n &= O(n^2) \\ 2n^2 &= O(n^2) \end{aligned}$$

Die zweite Gleichung ist eng, die erste ist es nicht.

- o („klein o“) erfasst nun nur die nicht engen asymptotischen oberen Schranken
- Analog: ω („klein omega“) erfasst nun nur die nicht engen asymptotischen unteren Schranken

Die o-Notation

Definition

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

- Vgl. O: $\exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$
- $f(n) \in o(g(n))$: $f(n)$ **wächst asymptotisch echt langsamer als** $g(n)$
- Wenn $f(n) \in o(g(n))$, dann wird $f(n)$ insignifikant relativ zu $g(n)$ wenn $n \to \infty$, d.h.

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

Die ω-Notation

Definition

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

- $f(n) \in \omega(g(n))$: $f(n)$ **wächst asymptotisch echt schneller als** $g(n)$
- Impliziert für $f(n) = \omega(g(n))$: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Beispiele

- $n^2 = \omega(n)$
- $1000n^2 \neq \omega(n^2)$
- $n^2 \lg n = \omega(n^2)$
- $n^{2.0001} = \omega(n^2)$
- $n^2 = o(n^3)$
- $n^2/1000 \neq o(n^2)$
- $n^2/\lg n = o(n^2)$
- $n^{1.9999} = o(n^2)$

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung
4.1 Die Θ-Notation
4.2 Die O- und Ω-Notation
4.3 Asymptotische Notation in Gleichungen
4.4 Die o- und ω-Notation
4.5 Vergleichen von Funktionen

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Eigenschaften (1)

Viele Eigenschaften, die beispielsweise für die reellen Zahlen gelten, gelten auch für die asymptotischen Notationen.

Transitivität

$$
\begin{aligned}
f(n) = \Theta(g(n)) \text{ und } g(n) = \Theta(h(n)) &\implies f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) \text{ und } g(n) = O(h(n)) &\implies f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) \text{ und } g(n) = \Omega(h(n)) &\implies f(n) = \Omega(h(n)) \\
f(n) = o(g(n)) \text{ und } g(n) = o(h(n)) &\implies f(n) = o(h(n)) \\
f(n) = \omega(g(n)) \text{ und } g(n) = \omega(h(n)) &\implies f(n) = \omega(h(n))
\end{aligned}
$$

Reflexivität

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

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

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

Eigenschaften (2)

Symmetrie

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

Antisymmetrie

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$
$$f(n) = o(g(n)) \iff g(n) = \omega(f(n))$$

Analogien zu den reellen Zahlen

Wegen dieser Eigenschaften, kann man folgende (hinkende, siehe unten) Analogie aufstellen:

$$\begin{array}{lcl} f(n) = O(g(n)) & \approx & a \leq b \\ f(n) = \Omega(g(n)) & \approx & a \geq b \\ f(n) = \Theta(g(n)) & \approx & a = b \\ f(n) = o(g(n)) & \approx & a < b \\ f(n) = \omega(g(n)) & \approx & a > b \end{array}$$

Folgende Eigenschaft gilt jedoch nicht:

> *Für je zwei reelle Zahlen a und b gilt entweder $a = b$, $a < b$, oder $a > b$.*

Es ist aber möglich, dass für zwei verschiedene Funktionen $f(n)$ und $g(n)$ weder $f(n) = o(g(n))$, noch $f(n) = \Theta(g(n))$, noch $f(n) = \omega(g(n))$ gilt.

- Beispiel: n und $n^{1+\sin(n)}$

Asymptotisches Wachstum von Polynomen

Lemma

Für f, g existiere der Grenzwert $\gamma = \lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R} \cup \{\infty\}$. Dann gilt

- *Falls $0 < \gamma < \infty$ so $f(n) = \Theta(g(n))$,*
- *Falls $\gamma = 0$ so $f(n) = o(g(n))$,*
- *Falls $\gamma = \infty$ so $f(n) = \omega(g(n))$.*

Anwendung auf Polynome

$p(n) = p_k n^k + p_{k-1} n^{k-1} + \cdots + p_1 n + p_0$ und
$q(n) = q_r n^r + q_{r-1} n^{r-1} + \cdots + q_1 n + q_0$.

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

Häufige Wachstumsordnungen

Für $0 < \delta < 1$:

$\Theta(1)$	*Durch Konstanten beschränktes Wachstum* $(= \Theta(n^0))$
$\Theta(\log n)$	*Logarithmisches Wachstum* Basis irrelevant, da $\log_a x = \log_a b \log_b x$ und somit $\log_a n = \Theta(\log_b n)$ für alle $a, b \geq 1$
$\log^{O(1)} n$	*Polylogarithmische Beschränktheit*
$\Theta(n^\delta)$	*Sublineares Wachstum*
$\Theta(n)$	*Lineares Wachstum*
$\Theta(n \log n)$	*"$n \log n$" (auch: linearithmisch, quasilinear)*
$\Theta(n^2)$	*Quadratisches Wachstum*
$\Theta(n^3)$	*Kubisches Wachstum*
$n^{O(1)}$	*Polynomielle Beschränktheit* $f(n) = n^{O(1)}$ wenn es ein $k > 0$ gibt, so dass $f(n) = O(n^k)$
$e^{\Omega(n^\delta)}$	*Schwach exponentielles Wachstum*
$e^{\Omega(n)}$	*Exponentielles Wachstum*

Exponentielles versus polynomielles Wachstum

Theorem

Exponentielle Funktionen wachsen echt schneller als polynomiell beschränkte Funktionen, d.h. für alle reellen Konstanten k und $a > 1$ gilt $n^k = o(a^n)$.

Beweis. Wir nehmen oBdA an, dass $k \in \mathbb{N}^+$. Es gilt $e^n = \sum_{i=0}^{\infty} \frac{n^i}{i!}$ und somit

$$a^n = e^{n \ln a} = \sum_{i=0}^{\infty} \frac{(\ln a)^i n^i}{i!}$$

Setze

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

Es gilt $a^n > p(n) = \Theta(n^{k+1}) = \omega(n^k)$. Die Behauptung folgt durch Antisymmetrie. □

- Bemerke: Es gilt somit auch $n^k = o(a^{c_1 n + c_2})$ für $a > 1$, $c_1 > 0$

Polylogarithmisches versus polynomielles Wachstum

Theorem

Polylogarithmisch beschränkte Funktionen wachsen echt langsamer als sublineare oder polynomielle Funktionen, d.h. für alle reellen Konstanten b und $a, k > 0$ gilt $\lg^b n = o(n^k)$.

Beweis. Wir haben gerade gezeigt, dass

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

für alle $a > 1$. Wähle $a = 2^k > 1$ und ersetze n durch $\lg n$:

$$0 = \lim_{n\to\infty} \frac{\lg^b n}{(2^k)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^k}. \qquad \square$$

- Bemerke: Es gilt somit auch $\log_a^b n = o(n^k)$ für $a > 1$, $k > 0$

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Erinnerung: Sortieren durch Mischen

```
MERGE-SORT(A, p, r)                                  // Sortiert A[p..r]
1: if p < r
2:     q ← ⌊(p + r)/2⌋
3:     MERGE-SORT(A, p, q)
4:     MERGE-SORT(A, q + 1, r)
5:     MERGE(A, p, q, r)
```

- Die Laufzeit von $\textsc{Merge-Sort}(A, 1, n)$ ist

$$T(n) = \begin{cases} \Theta(1) & \text{wenn } n \leq c \\ 2T(n/2) + \Theta(n) & \text{wenn } n > c. \end{cases}$$

- Der Werte von c spielt für die Wachstumsordnung keine Rolle, somit schreiben wir kurz: $T(n) = 2T(n/2) + \Theta(n)$
- Wir haben behauptet: $T(n) = \Theta(n \log n)$
- Warum?

Lösen von Rekurrenzen

Rekurrenzen entstehen oft bei der Analyse von Teile-und-Herrsche-Algorithmen. Es gibt verschiedene Arten, diese zu lösen:

1. Substitutionsmethode
   - Lösung schätzen
   - Mittels Induktion Konstanten finden und zeigen, dass Lösung korrekt ist
2. Rekursionsbäume/Iterationsmethode
   - Schätzungen finden
3. Master Theorem
   - Lösungen ablesen

Für die Rekurrenz von MERGE-SORT verwenden wir das Master Theorem. Dieses ist am einfachsten, wenn anwendbar.

Beispiel: Substitutionsmethode

$$T(n) = \begin{cases} c & \text{wenn } n = 1 \\ 2T(n/2) + cn & \text{wenn } n > 1 \end{cases}$$

- Vereinfachende Annahme: n ist Zweierpotenz
- Vermutung: $T(n) = O(n \log n)$ und insbesondere $T(n) \leq dn \lg n$ für $d > 0$ und $n > 1$
- Induktionsanfang ($n = 2$)

$$T(n) = 4c \leq dn \lg n \qquad \text{für alle } d \geq 2c$$

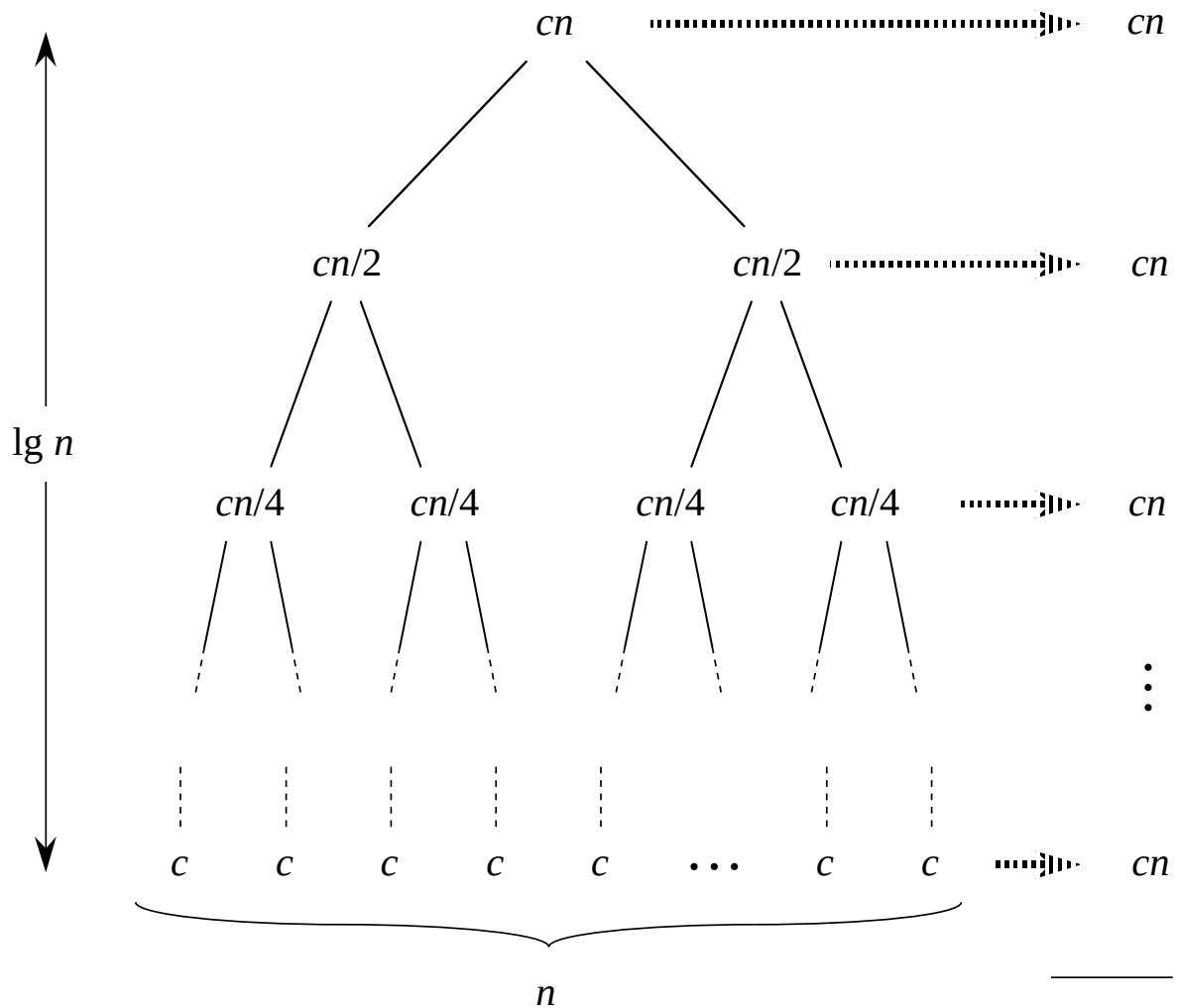
- Induktionsschritt

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &\leq 2d(n/2)\lg(n/2) + cn \\ &= dn \lg(n/2) + cn = dn \lg n - dn + cn \\ &\leq dn \lg n \qquad \text{wenn } d \geq c \end{aligned}$$

- Somit $T(n) \leq dn \lg n$ für $d \geq 2c$ und somit $T(n) = O(n \log n)$
- Ähnlich untere Schranke $\Omega(n \log n)$ und somit $T(n) = \Theta(n \log n)$

Beispiel: Rekursionsbaum

$$T(n) = \begin{cases} c & \text{wenn } n = 1 \\ 2T(n/2) + cn & \text{wenn } n > 1 \end{cases}$$



Total: $cn \lg n + cn$

Beispiel: Iterationsmethode (1)

$$T(n) = \begin{cases} c & \text{wenn } n = 1 \\ 2T(n/2) + cn & \text{wenn } n > 1 \end{cases}$$

Vereinfachende Annahme: n ist Zweierpotenz, d.h. $n = 2^N$.

$$\begin{aligned} T(n) &= cn + 2T(n/2) \\ &= cn + 2(cn/2 + 2T(n/4)) \\ &= cn + 2cn/2 + 4(cn/4 + 2T(n/8)) \\ &= cn + 2cn/2 + 4cn/4 + 8T(n/8) \\ &= \sum_{i=0}^{N} 2^i \cdot \frac{cn}{2^i} \\ &= (N+1)cn \\ &= cn(\lg n + 1) \\ &= \Theta(n \log n) \end{aligned}$$

Beispiel: Iterationsmethode (2)

$$T(n) = \begin{cases} 1 & \text{wenn } n = 1 \\ 2T(n/2) + n^2 & \text{wenn } n > 1 \end{cases}$$

Vereinfachende Annahme: n ist Zweierpotenz, d.h. $n = 2^N$.

$$\begin{aligned} T(n) &= n^2 + 2T(n/2) \\ &= n^2 + 2(n^2/4 + 2T(n/4)) \\ &= n^2 + 2n^2/4 + 4(n^2/16 + 2T(n/8)) \\ &= n^2 + 2n^2/4 + 4n^2/16 + 8T(n/8) \\ &= \sum_{i=0}^{N} 2^i \cdot \left(\frac{n}{2^i}\right)^2 = n^2 \sum_{i=0}^{N} 2^{-i} \\ &= n^2 \frac{1 - 2^{-(N+1)}}{1 - 2^{-1}} = 2n^2 - n = \Theta(n^2) \end{aligned}$$

Beispiel: Iterationsmethode (3)

$$T(n) = \begin{cases} 1 & \text{wenn } n = 1 \\ 16\,T(n/2) + n & \text{wenn } n > 1 \end{cases}$$

Vereinfachende Annahme: n ist Zweierpotenz, d.h. $n = 2^N$.

$$\begin{aligned} T(n) &= n + 2^4\,T(n/2) \\ &= n + 2^4(n/2 + 2^4\,T(n/4)) \\ &= n^2 + 2^4 n/2 + 2^8(n/4 + 2^4\,T(n/8)) \\ &= n^2 + 2^4 n/2 + 2^8 n/4 + 2^{12}\,T(n/8) \\ &= \sum_{i=0}^{N} 2^{4i}\frac{n}{2^i} = n\sum_{i=0}^{N} 8^i \\ &= n\frac{8^{(N+1)} - 1}{8 - 1} = 8n^4/7 - n/7 = \Theta(n^4) \end{aligned}$$

Master Theorem

Theorem

Es gelte $T(n) = aT(n/b) + f(n)$, *wobei* n/b *als* $\lfloor n/b \rfloor$ *oder* $\lceil n/b \rceil$ *interpretiert werden kann.*

1. *Falls* $f(n) = O(n^{\log_b a - \epsilon})$ *für ein* $\epsilon > 0$, *dann* $T(n) = \Theta(n^{\log_b a})$.
2. *Falls* $f(n) = \Theta(n^{\log_b a})$, *dann* $T(n) = \Theta(n^{\log_b a} \log(n))$.
3. *Falls* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *für ein* $\epsilon > 0$ *und* $af(n/b) \leq cf(n)$ *für ein* $c < 1$ *und alle genügend großen* n, *dann* $T(n) = \Theta(f(n))$.

Beispiele

- $a = b = 2$, $\log_b a = 1$, $f(n) = cn = \Theta(n^1)$
  $\rightarrow$ Fall 2: $T(n) = \Theta(n \log(n))$
- $a = b = 2$, $\log_b a = 1$, $f(n) = n^2 = \Omega(n^{1+1})$,
  $2f(n/2) = n^2/2 \leq \frac{1}{2} \cdot f(n) \rightarrow$ Fall 3: $T(n) = \Theta(n^2)$
- $a = 16$, $b = 2$, $\log_b a = 4$, $f(n) = n = O(n^{4-3})$
  $\rightarrow$ Fall 1: $T(n) = \Theta(n^4)$

Beweisidee: Master Theorem (1)

Vereinfachende Annahme $n = b^N$ und $T(1) = 1$. Auflösung der Rekursion liefert

$$T(n) = \sum_{i=0}^{N} a^i f(n/b^i).$$

Setze $c = \log_b a$.

Fall 1: $f(n) = n^{c-\epsilon}$

$$\begin{aligned} T(n) &= \sum_{i=0}^{N} a^i \frac{n^{c-\epsilon}}{b^{i(c-\epsilon)}} \\ &= n^{c-\epsilon} \sum_{i=0}^{N} a^i \frac{b^{\epsilon i}}{a^i} \\ &= n^{c-\epsilon} \frac{b^{(N+1)\epsilon} - 1}{b^\epsilon - 1} = \Theta(n^c), \end{aligned}$$

da $b^c = a$ und $b^N = n$.

Beweisidee: Master Theorem (2)

Fall 2: $f(n) = n^c$

$$T(n) = \sum_{i=0}^{N} a^i \frac{n^c}{b^{ic}} = n^c \sum_{i=0}^{N} a^i \frac{1}{a^i} = \Theta(n^c \log(n)),$$

da $N = \log_b n$.

Fall 3: $f(n) = n^{c+\epsilon}$

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

$$= n^{c+\epsilon} \cdot \frac{1 - (b^{-\epsilon})^{N+1}}{1 - b^{-\epsilon}} = \Theta(n^{c+\epsilon}),$$

da $b^{-\epsilon} < 1$, also $\frac{1-(b^{-\epsilon})^{N+1}}{1-b^{-\epsilon}} = \Theta(1)$. □

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall
6.1 Diskrete Wahrscheinlichkeitstheorie
6.2 Probabilistische Analyse
6.3 Randomisierte Algorithmen

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall
6.1 Diskrete Wahrscheinlichkeitstheorie
6.2 Probabilistische Analyse
6.3 Randomisierte Algorithmen

Stichprobenraum

Definition

Der **Stichprobenraum** Ω eines **Zufallsexperiments** ist die Menge aller möglichen Ergebnisse.

Wir nehmen an, dass Ω endlich ist.

Beispiel

- Münzwurf: $\Omega = \{ \text{Kopf}, \text{Zahl} \}$
- Würfeln: $\Omega = \{ 1, 2, 3, 4, 5, 6 \}$

Im Allgemeinen können wir das Ergebnis eines Zufallsexperiments nicht mit Sicherheit vorhersagen.

Ereignis (1)

Definition

Ein **Ereignis** $A \subseteq \Omega$ ist eine Teilmenge des Stichprobenraums.
$\emptyset$ nennt man das **unmögliche Ereignis**, Ω das **sichere Ereignis**.
Für jedes $\omega \in \Omega$ nennt man $\{\omega\}$ **Elementarereignis**.

Beispiel (Münze)

- Ergebnis ist Kopf: $A = \{\text{Kopf}\}$
- Ergebnis ist Kopf oder Zahl: $A = \{\text{Kopf}, \text{Zahl}\}$
- Ergebnis ist Kopf und Zahl: $A = \emptyset$
- Ergebnis ist nicht Kopf: $A = \{\text{Zahl}\}$

Beispiel (Würfel)

- Ergebnis ist gerade: $A = \{2, 4, 6\}$
- Ergebnis ist gerade und ≤ 3: $A = \{2\}$

Ereignis (2)

Für Ereignisse $A, B \subseteq \Omega$ repräsentieren $A \cup B$, $A \cap B$ und A^{c}, jeweils "A oder B", "A und B", und "nicht A". Zwei Ereignisse A und B sind **disjunkt** wenn $A \cap B = \emptyset$.

Beispiel (Münze)

- Ergebnis ist Kopf: $A = \{\,\mathsf{Kopf}\,\}$
- Ergebnis ist Kopf oder Zahl: $A = \{\,\mathsf{Kopf}, \mathsf{Zahl}\,\} = \{\,\mathsf{Kopf}\,\} \cup \{\,\mathsf{Zahl}\,\}$
- Ergebnis ist Kopf und Zahl: $A = \emptyset = \{\,\mathsf{Kopf}\,\} \cap \{\,\mathsf{Zahl}\,\}$
- Ergebnis ist nicht Kopf: $A = \{\,\mathsf{Zahl}\,\} = \{\,\mathsf{Kopf}\,\}^{\mathrm{c}}$

Beispiel (Würfel)

- Ergebnis ist gerade: $A = \{\,2, 4, 6\,\} = \{\,2\,\} \cup \{\,4\,\} \cup \{\,6\,\}$
- Ergebnis ist gerade und ≤ 3: $A = \{\,2\,\} = \{\,2, 4, 6\,\} \cap \{\,1, 2, 3\,\}$

Wahrscheinlichkeitsraum

Definition

Ein **Wahrscheinlichkeitsmaß** $(2^\Omega, \mathbb{P})$ ist eine Funktion $\mathbb{P} : 2^\Omega \to [0,1]$ so dass

1. $\mathbb{P}(\emptyset) = 0$ und $\mathbb{P}(\Omega) = 1$,
2. $A_1, \ldots, A_n$ paarweise disjunkt $\implies \mathbb{P}(\bigcup_{i=1}^n A_n) = \sum_{i=1}^n \mathbb{P}(A_n)$.

Das Tripel $(\Omega, 2^\Omega, \mathbb{P})$ nennt man **Wahrscheinlichkeitsraum**.

Wir schreiben für $\mathbb{P}(\{\omega\})$ oft kurz $\mathbb{P}(\omega)$.

Beispiel

- Münze: $2^\Omega = \{\emptyset, \{\text{Kopf}\}, \{\text{Zahl}\}, \{\text{Kopf}, \text{Zahl}\}\}$
- Faire Münze: $\mathbb{P}(\text{Kopf}) = \mathbb{P}(\text{Zahl}) = \frac{1}{2}$
  Impliziert $\mathbb{P}(\emptyset) = 0$, $\mathbb{P}(\{\text{Kopf}, \text{Zahl}\}) = 1$
- Fairer Würfel: $\mathbb{P}(1) = \cdots = \mathbb{P}(6) = \frac{1}{6}$ (Rest impliziert)
- Ergebnis ist gerade: $\mathbb{P}(\{2,4,6\}) = \mathbb{P}(2) + \mathbb{P}(4) + \mathbb{P}(6) = \frac{1}{2}$
- Ergebnis ist ≤ 3: $\mathbb{P}(\{1,2,3\}) = \mathbb{P}(1) + \mathbb{P}(2) + \mathbb{P}(3) = \frac{1}{2}$

Bedingte Wahrscheinlichkeit

Definition

Wenn $\mathbb{P}(B) > 0$, dann ist die **bedingte Wahrscheinlichkeit** von A gegeben B definiert als:

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

Beispiel

Zwei Würfel; W. dass Summe > 6 wenn erster Würfel $= 3$?

- $\Omega = \{1, \ldots, 6\}^2$
- Summe > 6: $A = \{(a, b) \in \Omega : a + b > 6\}$
- Erster Würfel $= 3$: $B = \{(3, b) : 1 \leq b \leq 6\}$
- $A \cap B = \{(3, 4), (3, 5), (3, 6)\}$
- $\mathbb{P}(A \mid B) = \mathbb{P}(A \cap B) / \mathbb{P}(B) = \frac{3}{36} / \frac{6}{36} = \frac{1}{2}$

Unabhängigkeit

Definition

Zwei Ereignisse A und B sind **unabhängig** wenn $\mathbb{P}(A \cap B) = \mathbb{P}(A)\,\mathbb{P}(B)$.

Wenn $\mathbb{P}(B) > 0$, impliziert dass $\mathbb{P}(A \mid B) = \mathbb{P}(A)$.

Beispiel

Zwei unabhängige Ereignisse

- Würfel gerade: $A = \{2, 4, 6\}$
- Würfel ≤ 4: $B = \{1, 2, 3, 4\}$:
- $\mathbb{P}(A \cap B) = \mathbb{P}(\{2, 4\}) = \frac{1}{3} = \frac{1}{2} \cdot \frac{2}{3} = \mathbb{P}(A)\,\mathbb{P}(B)$

Nicht unabhängig

- Würfel ungerade: $C = \{1, 3, 5\}$
- $\mathbb{P}(A \cap C) = \mathbb{P}(\emptyset) = 0 \neq \frac{1}{2} \cdot \frac{1}{2} = \mathbb{P}(A)\,\mathbb{P}(C)$

Disjunkt $\neq$ unabhängig.

Bedingte Unabhängigkeit ⋆

Definition

Seien A, B, C Ereignisse mit $\mathbb{P}(C) > 0$. A und B sind **bedingt unabhängig gegeben** C wenn

$$\mathbb{P}(A \cap B \mid C) = \mathbb{P}(A \mid C)\,\mathbb{P}(B \mid C).$$

Beispiel

- Würfel gerade: $A = \{2, 4, 6\}$
- Würfel ≤ 3: $B = \{1, 2, 3\}$
- $\mathbb{P}(A \cap B) = \frac{1}{6} \neq \frac{1}{2} \cdot \frac{1}{2} = \mathbb{P}(A)\,\mathbb{P}(B)$
  $\rightarrow$ A und B sind nicht unabhängig
- Würfel kein Vielfaches von 3: $C = \{1, 2, 4, 5\}$
- $\mathbb{P}(A \cap B \mid C) = \frac{1}{4} = \frac{1}{2} \cdot \frac{1}{2} = \mathbb{P}(A \mid C)\,\mathbb{P}(B \mid C)$
  $\rightarrow$ A und B sind bedingt unabhängig gegeben C

Produktraum ⋆

Definition

Seien $(\Omega_1, 2^{\Omega_1}, \mathbb{P}_1)$ und $(\Omega_2, 2^{\Omega_2}, \mathbb{P}_2)$ zwei Wahrscheinlichkeitsräume. Ihr **Produktraum** ist definiert als $(\Omega_{12}, 2^{\Omega_{12}}, \mathbb{P}_{12})$ mit $\Omega_{12} = \Omega_1 \times \Omega_2$ und

$$\mathbb{P}_{12}(A_1 \times A_2) = \mathbb{P}_1(A_1)\,\mathbb{P}_2(A_2).$$

Beispiel: Zwei faire Würfel

- $\Omega_1 = \Omega_2 = \{1, 2, 3, 4, 5, 6\}$
- $\Omega_{12} = \{(1,1), (1,2), \ldots, (6,6)\}$
- Erster Würfel: $A_1 = \{1, 2, 3\} \subseteq \Omega_1$
- Zweiter Würfel: $A_2 = \{2, 3, 4\} \subseteq \Omega_2$
- $\mathbb{P}_{12}(A_1 \times A_2) = \mathbb{P}_1(A_1)\,\mathbb{P}_2(A_2) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$

Produkträume kombinieren die Ergebnisse verschiedener *unabhängiger* Experimente.

Zufallsvariable

Definition

Eine reelle **Zufallsvariable** ist eine Funktion $X : \Omega \to \mathbb{R}$. Wir schreiben $\{ X = x \}$ bzw. $\{ X \leq x \}$ für Ereignisse $\{ \omega : X(\omega) = x \}$ bzw. $\{ \omega : X(\omega) \leq x \}$. Die **Wahrscheinlichkeitsfunktion** von X ist die Funktion $f_X : \mathbb{R} \to [0, 1]$ gegeben durch $f_X(x) = \mathbb{P}(X = x)$; die **Verteilungsfunktion** ist gegeben durch $F_X(x) = \mathbb{P}(X \leq x)$.

Beispiel: Zwei faire Würfel

- Summe der Ergebnisse: $X((a, b)) = a + b$
- $f_X(6) = \mathbb{P}(X = 6) = \mathbb{P}(\{ (1,5), (2,4), (3,3), (4,2), (5,1) \}) = \frac{5}{36}$
- $F_X(3) = \mathbb{P}(X \leq 3) = \mathbb{P}(\{ (1,1), (1,2), (2,1) \}) = \frac{1}{12}$

Bedingte Wahrscheinlichkeit, Unabhängigkeit und bedingte Unabhängigkeit lassen sich auf Zufallsvariablen übertragen.

Erwartungswert

Definition

Der **Erwartungswert** einer Zufallsvariable X ist gegeben durch

$$\mathbb{E}[X] = \sum_x x\, f_X(x).$$

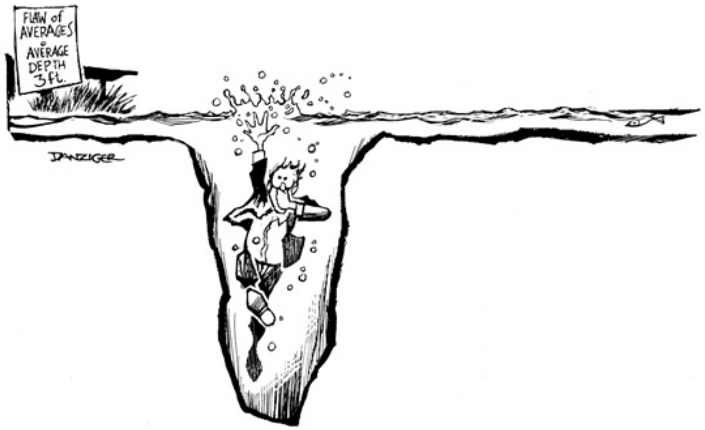
Wenn $g : \mathbb{R} \to \mathbb{R}$, dann

$$\mathbb{E}[g(X)] = \sum_x g(x) f_X(x).$$

Beispiel: Fairer Würfel (mit X Identitätsfunktion)

- $\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \cdots + 6 \cdot \frac{1}{6} = 3.5$
- Betrachte $g(x) = \lfloor x/2 \rfloor$
- $\mathbb{E}[g(x)] = 0 \cdot \frac{1}{6} + 1 \cdot \frac{1}{6} + \cdots + 3 \cdot \frac{1}{6} = 1.5$
- Aber: $g(\mathbb{E}[X]) = 1$

“Flaw of averages”



Mittel korrekt, Varianz ignoriert. $\mathbb{E}[g(X)] \neq g(\mathbb{E}[X])$

Vorsicht mit Erwartungswerten!

Savage, 2009.

Bedingter Erwartungswert $\star$

Definition

Seien X und Y Zufallsvariablen. Der *bedingte Erwartungswert* von Y gegeben X ist die Zufallsvariable $\psi(X)$ mit

$$\psi(x) = \mathbb{E}[\, Y \mid X = x \,] = \sum_y y \, f_{Y|X}(y \mid x),$$

wobei $f_{Y|X}(y \mid x) = \mathbb{P}(\, Y = y \mid X = x \,)$.

Beispiel

- *Indikatorvariable*: $I_A(\omega) = \begin{cases} 1 & \text{wenn } \omega \in A \\ 0 & \text{sonst} \end{cases}$
- Fairer Würfel; setze $X = I_{\text{even}} = I_{\{\,2,4,6\,\}}$; Y ist Identität
- $\mathbb{E}[\, Y \mid X = 1 \,] = 1 \cdot 0 + 2 \cdot \frac{1}{3} + 3 \cdot 0 + 4 \cdot \frac{1}{3} + 5 \cdot 0 + 6 \cdot \frac{1}{3} = 4$
- $\mathbb{E}[\, Y \mid X = 0 \,] = 1 \cdot \frac{1}{3} + 2 \cdot 0 + 3 \cdot \frac{1}{3} + 4 \cdot 0 + 5 \cdot \frac{1}{3} + 6 \cdot 0 = 3$
- $\mathbb{E}[\, Y \mid X \,](\omega) = \begin{cases} 4 & \text{if } X(\omega) = 1 \\ 3 & \text{if } X(\omega) = 0 \end{cases}$

Wichtige Eigenschaften

Wir schreiben $\mathbb{P}(X)$ für $\mathbb{P}(X = x)$.

Theorem

$$\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B)$$

$$\mathbb{P}(A^c) = 1 - \mathbb{P}(A)$$

Wenn $B \supseteq A$, $\quad \mathbb{P}(B) = \mathbb{P}(A) + \mathbb{P}(B \setminus A) \geq \mathbb{P}(A)$

$$\mathbb{P}(X) = \sum_y \mathbb{P}(X, Y = y) \qquad \textit{(Summenregel)}$$

$$\mathbb{P}(X, Y) = \mathbb{P}(Y \mid X)\,\mathbb{P}(X) \qquad \textit{(Produktregel)}$$

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(B \mid A)\,\mathbb{P}(A)}{\mathbb{P}(B)} \qquad \textit{(Bayes' Theorem)}$$

$$\mathbb{E}[aX + b] = a\,\mathbb{E}[X] + b \qquad \textit{(Linearität Erwartungswert)}$$

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall
6.1 Diskrete Wahrscheinlichkeitstheorie
6.2 Probabilistische Analyse
6.3 Randomisierte Algorithmen

Das Sekretärinnenproblem

- Wir wollen eine Sekretärin einstellen
- Aber nicht irgendeine, sondern um jeden Preis die beste
- Jeden Tag interviewen wir eine Sekretärin und entscheiden uns sofort
- Wenn sie besser geeignet ist,
  - Feuern wir die aktuelle Sekretärin
  - Und stellen die neue Sekretärin ein
- Dabei entstehen Kosten
  - Kosten für das Interview: niedrig
  - Kosten für das Einstellen: hoch
- Wieviel kostet es, immer die beste Sekretärin zu haben?
  - Modelliert häufiges Berechnungsparadigma: Folge elementweise untersuchen um Minimum/Maximum zu bestimmen; dabei Warten des momentanen Gewinners
  - Problem modelliert Anzahl der Änderungen der Gewinner
  - Beispiel: die k kleinsten Elemente einer Folge ermitteln

Das Sekretärinnenproblem (Pseudo-Code)

HIRE-ASSISTANT(n)

```
1: best ← 0   // candidate 0 is a least-qualified dummy candidate
2: for i ← 1 to n
3:     interview candidate i                          // Kosten: c_i
4:     if candidate i is better than candidate best
5:         best ← i
6:         hire candidate i                           // Kosten: c_h
```

- Wir nehmen an $c_i \leq c_h$
- Analyse der Kosten ähnlich Analyse der Laufzeit
  $\rightarrow$ Zählen, wie oft bestimmte Operationen ausgeführt werden
- Wenn wir n Kandidaten interviewen und m davon einstellen, ist die Laufzeit $O(c_i n + c_h m)$
- Aber was ist m?
  - Worst Case: $m = n$, Kosten $O(c_h n)$
  - Was passiert in einem "typischen" Fall?

Probabilistische Analyse

- **Probabilistische Analyse** bedeutet, Wahrscheinlichkeitstheorie zur Analyse eines Algorithmus zu verwenden (häufig: Laufzeit)
- Ansatz
  - Wir treffen Annahmen über die Wahrscheinlichkeitsverteilung der möglichen Eingaben
  - Wir analysieren dann die erwartete Laufzeit
- **Average Case-Laufzeit** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \mathbb{E}_{x \in \mathbb{P}_n X_n}\left[\, time_A(x) \,\right]$$

  für eine Wahrscheinlichkeitsverteilung $\mathbb{P}_n$ über Eingaben X_n
- Wahl der Verteilung $\mathbb{P}_n$ muss sorgfältig erfolgen
  - Probabilistische Analyse nur anwendbar, wenn sinnvolle Verteilung bestimmt werden kann

Zufallspermutation

- Für das Sekretärinnenproblem können wir annehmen, dass die Reihenfolge der n Sekretärinnen zufällig ist
- Vereinfachende Annahme: Für jedes Kandidatenpaar können wir bestimmen, welcher Kandidat besser geeignet ist
  $\rightarrow$ Totale Ordnung
- **Rang** eines Kandidaten = Position in der sortierten Kandidatenliste
  - Am wenigsten geeignete Kandidate hat Rang 1
  - Am besten geeigneter Kandidat hat Rang n
- Ränge der Eingaben $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ sind Permutation von $\langle 1, 2, \ldots, n \rangle$
- Zufällige Reihenfolge = Zufallspermutation
  - Jede der $n!$ Permutationen mit gleicher Wahrscheinlichkeit
  - Für jede Permutation $s_n \in \mathcal{S}_n$ ist $\mathbb{P}(s_n) = 1/n!$

Indikatorvariablen (1)

Definition

Sei A ein Ereignis. Die **Indikatorvariable** I_A für A ist gegeben durch

$$I_A = \begin{cases} 1 & \text{wenn } A \text{ eintritt} \\ 0 & \text{wenn } A \text{ nicht eintritt.} \end{cases}$$

Beispiel (Münzwurf)

- Stichprobenraum: $\Omega = \{\,\text{Kopf}, \text{Zahl}\,\}$
- $\mathbb{P}(\,\text{Kopf}\,) = \mathbb{P}(\,\text{Zahl}\,) = 1/2$
- $I_{\text{Kopf}} = \begin{cases} 1 & \text{wenn Kopf auftritt} \\ 0 & \text{wenn Zahl auftritt} \end{cases}$
- Erwartete Anzahl "Köpfe" pro Wurf

$$\mathbb{E}\,[\,I_{\text{Kopf}}\,] = 1 \cdot \mathbb{P}(\,\text{Kopf}\,) + 0 \cdot \mathbb{P}(\,\text{Zahl}\,) = \mathbb{P}(\,\text{Kopf}\,)$$

Indikatorvariablen (2)

Lemma

Es gilt $\mathbb{E}[I_A] = \mathbb{P}(A)$.

- Indikatorvariablen sind nützlich, um Erwartungswerte zu berechnen (insbesondere wenn Abhängigkeiten bestehen können)
- Beispiel: n Münzwürfe
  - Sei X die Anzahl der Köpfe; Erwartungswert ist

$$\mathbb{E}[X] = \sum_{i=0}^{n} i\,\mathbb{P}(X = i) = \sum_{i=0}^{n} i \binom{n}{i} \left(\frac{1}{2}\right)^{i} \left(\frac{1}{2}\right)^{n-i}$$

  - Das kann man mit Indikatorvariablen einfacher berechnen
  - Sei $X_i = I_{i\text{-ter Wurf ist Kopf}}$
  - Beobachte: $X = \sum_{i=1}^{n} X_i$
  - Somit

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$$

Analyse des Sekretärinnenproblems (1)

Wie oft stellen wir eine neue Sekretärin ein?

- Annahme: zufällige Reihenfolge
- $X =$ Anzahl der Einstellungen
- $X_i = \begin{cases} 1 & \text{Kandidat } i \text{ wird eingestellt} \\ 0 & \text{Kandidat } i \text{ wird nicht eingestellt} \end{cases}$
- $X = \sum_{i=1}^{n} X_i$
- Wir wissen das $\mathbb{E}[X_i] = \mathbb{P}(\text{Kandidat } i \text{ wird eingestellt})$
  - Betrachte die ersten i Kandidaten
  - Einer dieser Kandidaten ist der Beste bisher
  - Da die Reihenfolge zufällig ist, ist das Kandidat i mit Wahrscheinlichkeit $1/i$
  - Also $\mathbb{E}[X_i] = 1/i$

Analyse des Sekretärinnenproblems (2)

- Somit

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$$

- Wir interviewen also n Kandidaten, stellen aber im Schnitt nur $\approx \ln n$ ein
- Das ist deutlich weniger als der Worst Case

Lemma

Unter der Annahme zufälliger Kandidatenreihenfolge hat HIRE-ASSISTANT *Average Case-Einstellungskosten von* $O(c_h \ln n)$.

Analyse des Sekretärinnenproblems (3)

- Algorithmus ist deterministisch
- Einstellungskosten für gleiche Eingabe immer gleich
- Insbesondere: Hängen nur von $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ ab
- Einige Rangpermutationen haben hohe Kosten, z.B. $\langle 1, 2, \ldots, n \rangle$
- Einige Rangpermutationen haben niedrige Kosten, z.B. $\langle n, ... \rangle$
- Andere sind dazwischen

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall
6.1 Diskrete Wahrscheinlichkeitstheorie
6.2 Probabilistische Analyse
6.3 Randomisierte Algorithmen

Randomisierte Algorithmen

- Für die probabilistische Analyse müssen wir die Eingabeverteilung angeben $\rightarrow$ oft schwierig
- Wahrscheinlichkeit und Zufall sind aber auch im Algorithmenentwurf nützlich
- Algorithmen, deren Abarbeitung "vom Zufall abhängt", nennt man **randomisierte Algorithmen**
  - Verhalten bestimmt durch Eingabe...
  - und von Werten, die von einem **Zufallszahlengenerator** erzeugt werden
- Hier: $\textsc{Random}(a, b)$ liefert ganzzahlige Zufallszahl in $[a, b]$
  - Jede Zahl mit gleicher Wahrscheinlichkeit $(1/(b - a + 1))$
  - Wiederholte Aufrufe unabhängig
  - In Praxis oft durch **Pseudo-Zufallszahlengenerator** implementiert (deterministisch, aber Werte sehen "zufällig" aus)
- **Erwartete Laufzeit** ist Erwartungswert der Laufzeit über die Zufallszahlenverteilung
  - Vgl. Average Case: Erwartungswert über Eingabenverteilung

Randomisierung des Sekretärinnenproblems (1)

Idee: Anstatt Eingabeverteilung anzunehmen, erzwingen wir sie und permutieren die Kandidaten zufällig vor dem ersten Interview.

RANDOMIZED-HIRE-ASSISTANT(n)

```
1: randomly permute list of candidates
2: best ← 0   // candidate 0 is a least-qualified dummy candidate
3: for i ← 1 to n
4:     interview candidate i                          // Kosten: c_i
5:     if candidate i is better than candidate best
6:         best ← i
7:         hire candidate i                           // Kosten: c_h
```

Randomisierung des Sekretärinnenproblems (2)

- Wir erwarten immer noch $O(\ln n)$ Einstellungen
- Aber jetzt: wir erwarten dies für *jede* Eingabe anstatt für Eingaben mit einer bestimmten Verteilung
  - Selbst bei gleichen Eingaben kann Anzahl der Einstellungen in jeder Ausführung des Algorithmus variieren
  - Hier und oft: keine bestimmte Eingabe erzwingt das Worst Case-Verhalten (d.h. es gibt keine schlechten Eingaben)
  - Algorithmus nur dann schlecht, wenn zufällige Permutation ungünstig ausgefallen

Lemma

RANDOMIZED-HIRE-ASSISTANT *hat erwartete Einstellungskosten von* $O(c_h \ln n)$.

Erzeugen von Zufallspermutation

RANDOMIZE-IN-PLACE(A, n)

1: **for** $i \leftarrow 1$ **to** n
2: swap $A[i]$ with $A[\text{RANDOM(i,n)}]$

- In Iteration i wählen wir ein zufälliges, noch nicht ausgewähltes Element für $A[i]$ aus
- Laufzeit: $O(1)$ per Iteration, $O(n)$ gesamt
- Auch *Fischer-Yates shuffle* oder *Knuth shuffle* genannt

Outline

1. Einführung

2. Beispiel: Insertion Sort

3. Beispiel: Merge Sort

4. Wachstumsordnung

5. Rekurrenzen

6. Wahrscheinlichkeit und Zufall

Zum Nachlesen

- CLRS, Ch. 1, *The Role of Algorithms in Computing*
- CLRS, Ch. 2, *Getting Started*
- CLRS, Ch. 3, *Growth of Functions*
- CLRS, Ch. 4, *Divide-and-Conquer*
- CLRS, Ch. 5, *Probabilistic Analysis*
- CLRS, App. A, *Summations*
- CLRS, App. B, *Sets, Etc.*
- CLRS, App. C, *Counting and Probability*

Algorithmen und Datenstrukturen

02 – Sortieralgorithmen

Prof. Dr. Rainer Gemulla

Universität Mannheim

HWS 2014

Das Sortierproblem

Eingabe

Folge von n Elementen

$$\langle a_1, a_2, \ldots, a_n \rangle$$

aus einer geordneten Menge M

Ausgabe

Permutation (Umordnung)

$$\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$$

der Elemente so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$, $\pi \in \mathcal{S}_n$

Die Struktur der Daten

- In der Praxis sortiert man typischerweise **Datensätze** (Tupel, *records*) nach einem **Schlüssel** (*key*)
  - z.B. Personendatensätze nach Namen
  - z.B. E-Mails nach Zeit
  - z.B. Hotels nach Preis
- Sortieralgorithmen permutieren Datensätze oder Zeiger auf Datensätze
  - Letzteres spart Kopierkosten für große Datensätze
- Wir formulieren Sortieralgorithmen aber mit Zahlen
  - Umformulierung/Umkodierung für konkretes Problem notwendig
  - Algorithmus ist Abstraktion

Warum Sortieralgorithmen?

- Manche Anwendungen müssen inherent Daten sortieren
- Viele Algorithmen verwenden Sortieren als Unteroperation
- Wichtige Techniken des Algorithmenentwurfs sowie wichtige Datenstrukturen finden sich in den vielen existierenden Sortieralgorithmen wieder
- Untere Schranke für Laufzeitkomplexität beweisbar
  $\rightarrow$ untere Schranke für bestimmte andere Probleme
- Implementation von Sortieralgorithmen system- und datenabhängig; davon abstrahieren wir

Übersicht

Algorithmus	Worst Case-Laufzeit	Average Case-/erwartete Laufzeit
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$\Theta(n \lg n)$	
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$
Counting Sort	$O(n + k)$	$O(n + k)$
Radix Sort	$O(d(n + k))$	$O(d(n + k))$
Bucket Sort	$O(n^2)$	$O(n)$

- Heapsort und Quicksort sortieren **in place** (Merge-Sort nicht)
  - Quicksort einer der schnellsten Algorithmen in der Praxis
- Untere Schranke $\Omega(n \lg n)$ für vergleichsbasiertes Sortieren
- Geschlagen durch
  - Counting-Sort für n Zahlen aus $\{1, \ldots, k\}$
  - Radix-Sort für n Zahlen bestehend aus d Ziffern mit je k möglichen Werten
  - Bucket-Sort für reelle Zahlen mit bekannter Verteilung

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

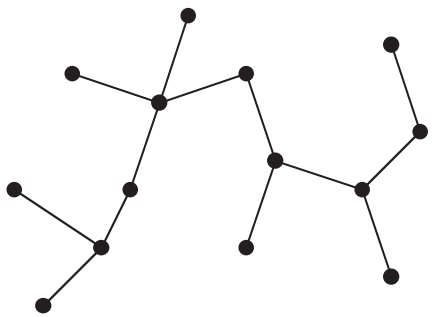
Freier Baum, Wald

Definition

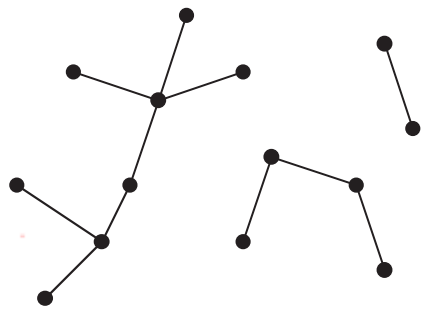
Ein **freier Baum** (*free tree*) ist ein verbundener, azyklischer, ungerichteter Graph. Wenn ein ungerichteter Graph azyklisch aber möglicherweise nicht verbunden ust, heißt er **Wald** (*forrest*).

Wenn wir sagen, dass ein Graph ein **Baum** ist, meinen wir einen freien Baum.

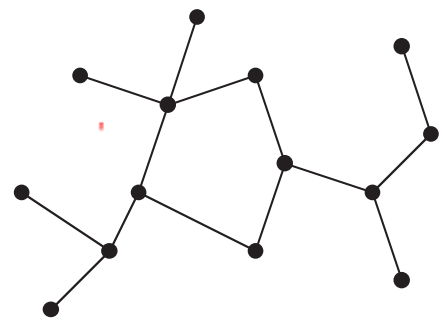
Beispiel



Freier Baum, Wald



Wald



Weder freier Baum noch Wald

Charakterisierung

Theorem

Sei $G = (V, E)$ ein ungerichteter Graph. Die folgenden Aussagen sind äquivalent:

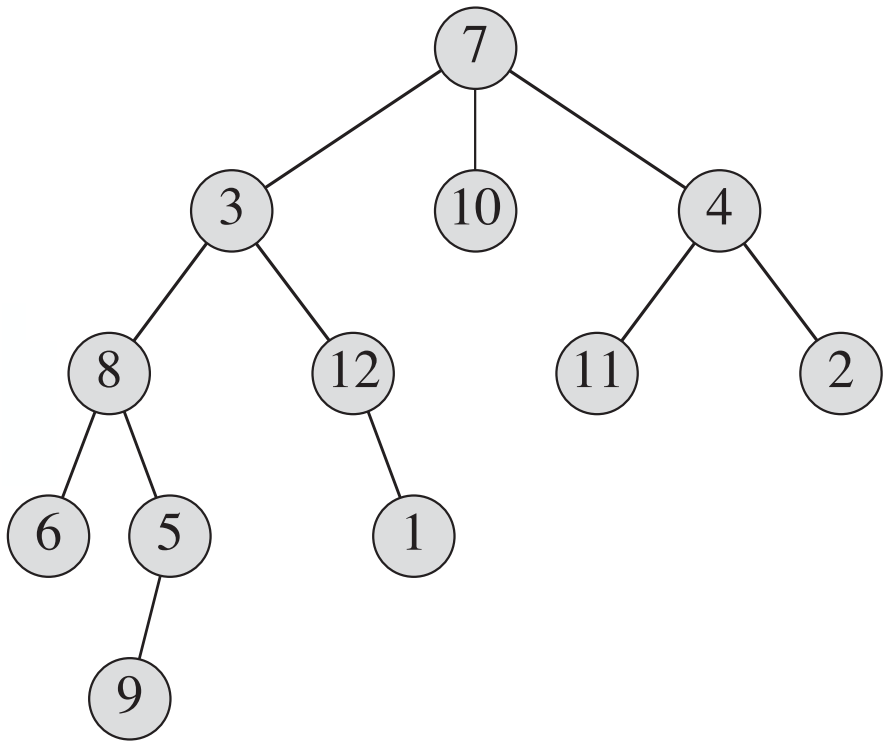
1. *G ist ein freier Baum.*
2. *Jedes Knotenpaar in G ist durch genau einen einfachen Pfad verbunden.*
3. *G ist verbunden, aber wenn eine Kante entfernt wird, ist der resultierende Graph nicht verbunden.*
4. *G ist verbunden und $|E| = |V| - 1$.*
5. *G ist azyklisch und $|E| = |V| - 1$.*
6. *G ist azyklisch, aber wenn eine Kante hinzugefügt wird, ist der resultierende Graph zyklisch.*

Wurzelbaum

Definition

Ein **Wurzelbaum** (*rooted tree*) ist ein freier Baum, in dem ein Knoten als **Wurzelknoten** (*root*) ausgezeichnet ist.

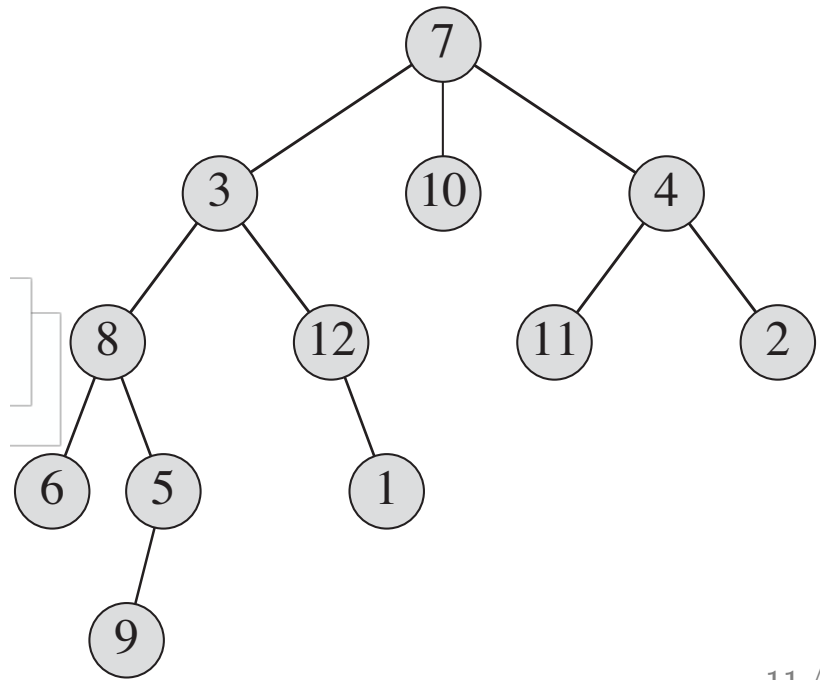
Wir zeichnen den Wurzelknoten oben.



Begriffe (1)

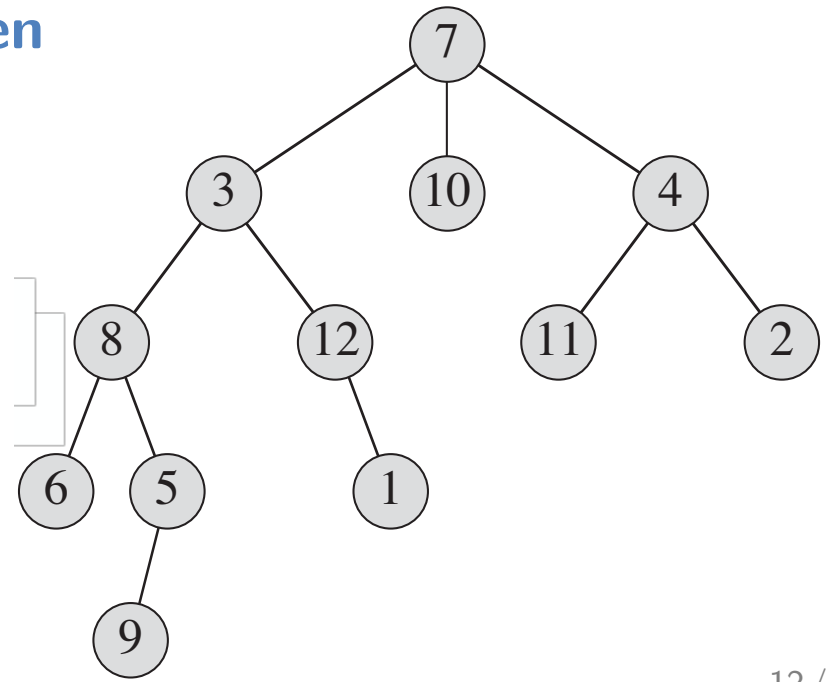
Betrachte einen Knoten v in einem Baum T mit Wurzel w.

- Jeder Knoten t auf einem Pfad von w zu v heisst **Vorgänger** (*ancestor*) von v
- Falls t ein Vorgänger von v ist, so heisst v **Nachfolger** (*descendant*) von t
- Jeder Knoten ist Vorgänger und Nachfolger von sich selbst
- Ist $v \neq t$, so sprechen wir von **echten Vorgängern und Nachfolgern**
- Der **Teilbaum** des Knotens v ist der von den Nachfolgern von v induzierte Teilbaum von T (d.h. alle Nachfolger und die sie verbindenden Kanten)



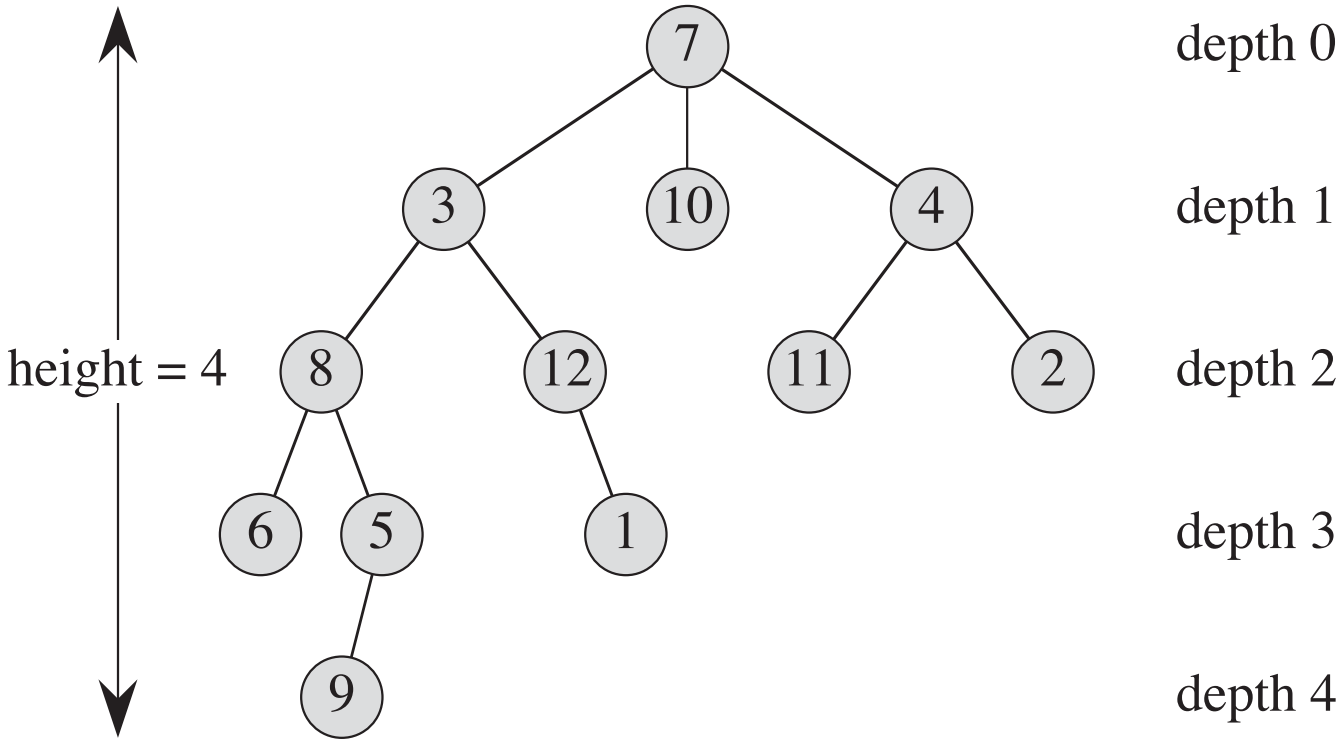
Begriffe (2)

- Falls (p, v) die letzte Kante im Pfad von w nach v ist, so heisst p **Vaterknoten/Elternknoten** (*parent*) von v und v heisst **Sohnknoten/Kindknoten** (child) von p
- Falls v und t denselben Vaterknoten haben, so heissen sie **Geschwisterknoten** (*sibling*)
- Die Wurzel ist der einzige Knoten ohne Vaterknoten
- Knoten ohne Kindknoten heissen **äussere Knoten** oder **Blattknoten**
- Alle anderen Knoten heissen **innere Knoten**
- Die Anzahl der Sohnknoten eines Knotens v heisst **Grad** von v



Begriffe (3)

- Die Länge des Pfades (Anzahl Kanten) von w nach v heisst **Tiefe** (Ebene, Level, *depth*) von v
- Die maximale Tiefe eines Knoten heisst **Höhe** (*height*) von T
- Die **Höhe** eines Knoten v ist längster einfacher Pfad von v zu einem Nachfolgerblatt (Beachte: *Alle* Blätter haben Höhe 0)



Geordneter Baum

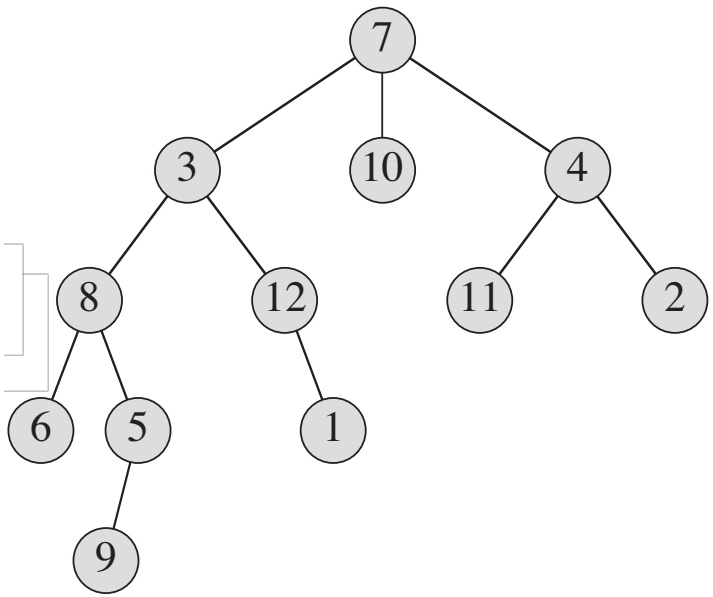
Definition

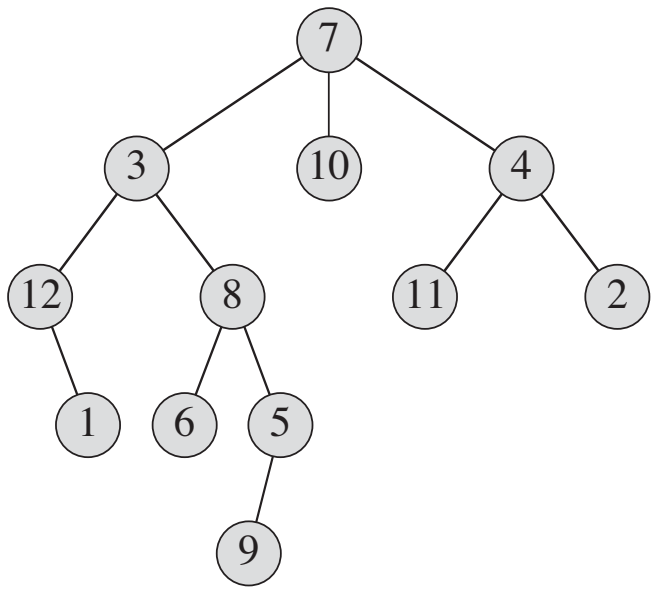
Ein **geordneter Baum** ist ein Wurzelbaum, bei dem die Kindknoten geordnet sind.

Wir sprechen vom vom 1., 2., bis zum k-ten Sohn.

Beispiel

- Als Wurzelbäume sind beide Bäume identisch
- Als geordnete Bäume sind sie aber verschieden





Binärbaum (1)

Definition

Ein **Binärbaum** ist eine Struktur auf einer endlichen Knotenmenge, die entweder

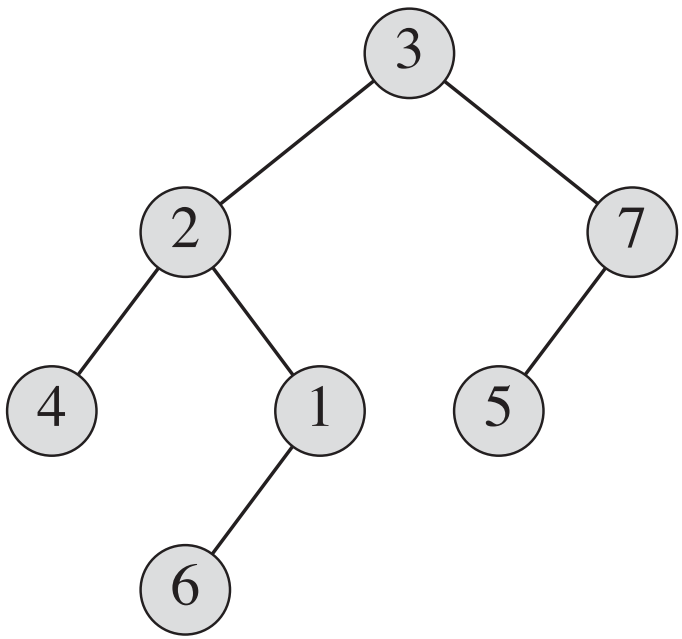
- keine Knoten enthält (**leerer Baum**) oder
- aus drei disjunkten Teilmengen zusammengesetzt ist: einem Wurzelknoten w und zwei Binärbäumen genannt **linker Teilbaum** und **rechter Teilbaum**.

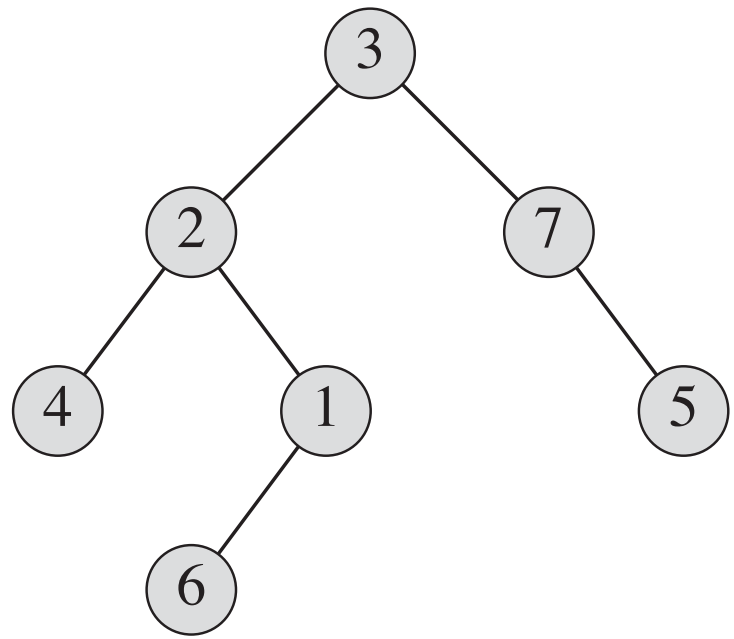
- Für einen Knoten v sprechen wir von **rechtem und linkem Sohn** für die Wurzeln nicht-leerer linker und rechter Teilbäume
- Selbst wenn ein Knoten nur einen Sohn hat, ist dieser entweder linker oder rechter Sohn (somit ist ein Binärbaum mehr als ein geordneter Baum)
- Verallgemeinerung: **k-äre Bäume** mit k positionierten Teilbäumen

Binärbaum (2)

Beispiel

- Als Wurzelbäume/geordnete Bäume sind beide Bäume gleich
- Als Binärbäume sind sie aber verschieden



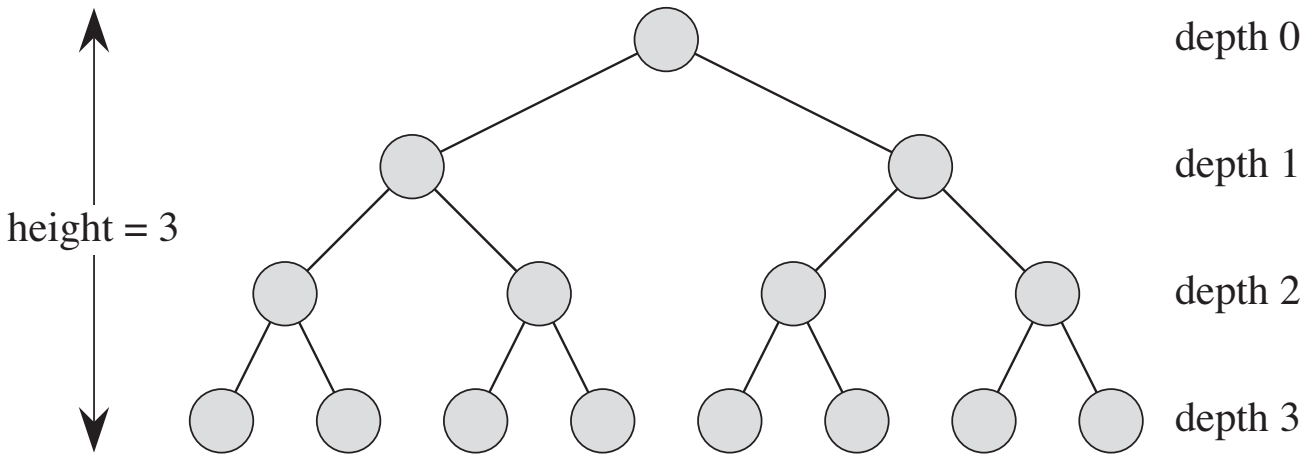


Vollständige k-äre Bäume

Definition

Ein k-ärer Baum der Höhe h heißt **vollständig** (*complete*), wenn jedes Blatt Höhe h besitzt und jeder innere Knoten Grad k besitzt.

- Besteht aus $(k^{h+1} - 1)/(k - 1)$ Knoten
  - k^h Blattknoten
  - $(k^h - 1)/(k - 1)$ innere Knoten
- Ein vollständiger k-ärer Baum mit n Blättern hat Höhe $\log_k n$
- Ein k-ärer Baum der Höhe h hat höchstens k^h Blätter und höchstens $(k^{h+1} - 1)/(k - 1)$ Knoten



Outline

1. Bäume

2. Heapsort
2.1 Heap
2.2 Heapsort
2.3 Prioritätswarteschlangen

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Heapsort

Die wichtigsten Eigenschaften von Heapsort in Kürze:

- Worst Case: $O(n \lg n)$
- In-place
- Benutzte Datenstruktur: Heap

Die besten Eigenschaften von Merge Sort ($O(n \lg n)$) und Insertion Sort (in place) werden somit kombiniert.

Ein Heap wird in vielen Algorithmen benötigt, z.B.:

- Prioritätswarteschlangen (*priority queues*)
- Selektion der k kleinsten Elemente (Übung)

Outline

1. Bäume

2. Heapsort
2.1 Heap
2.2 Heapsort
2.3 Prioritätswarteschlangen

3. Quicksort

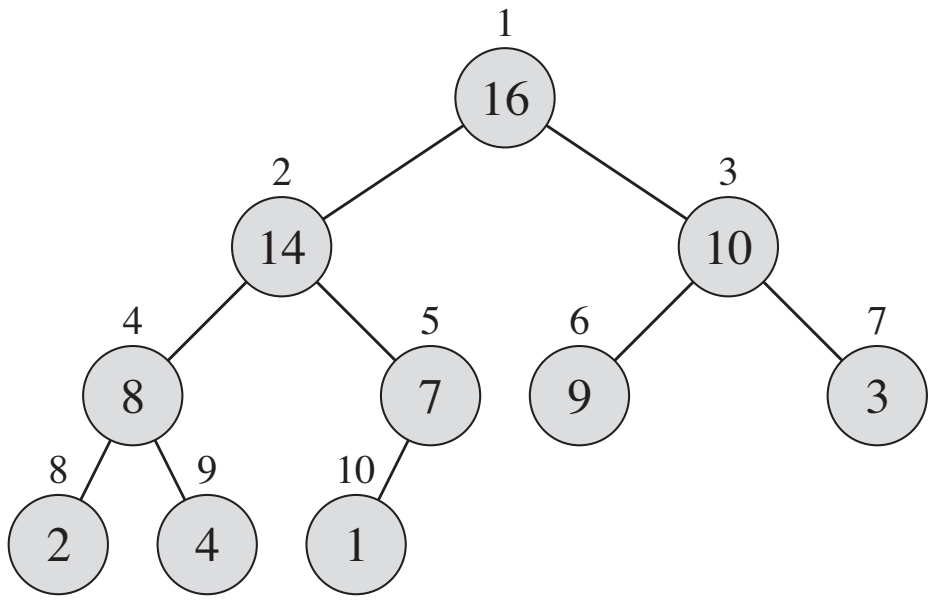
4. Untere Schranke

5. Sortieren in Linearzeit

Heap

Ein binärer Heap ist ein "fast vollständiger" binärer Baum:

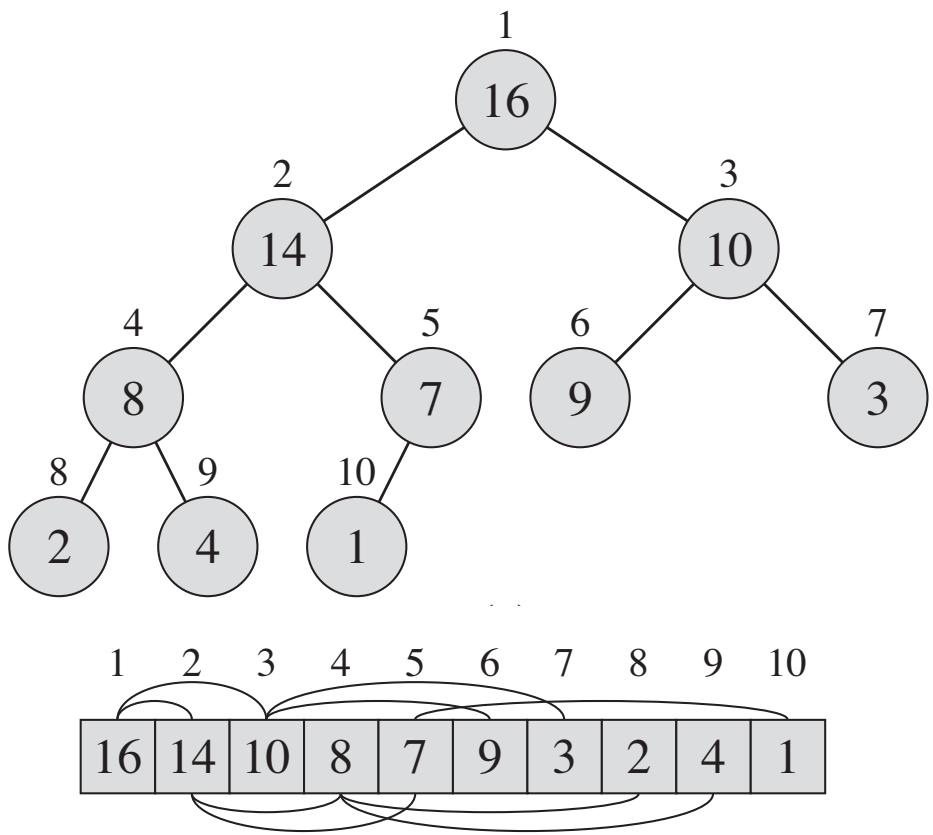
- Alle Ebenen ausser der möglicherweise der unteresten sind vollständig gefüllt
- Die unterste Ebene ist von links nach rechts gefüllt
- Ein Heap mit n Elementen hat Höhe $\lfloor \lg(n) \rfloor = \Theta(\lg n)$



Heap als Array (1)

Ein Heap kann als ein Array repräsentiert werden.

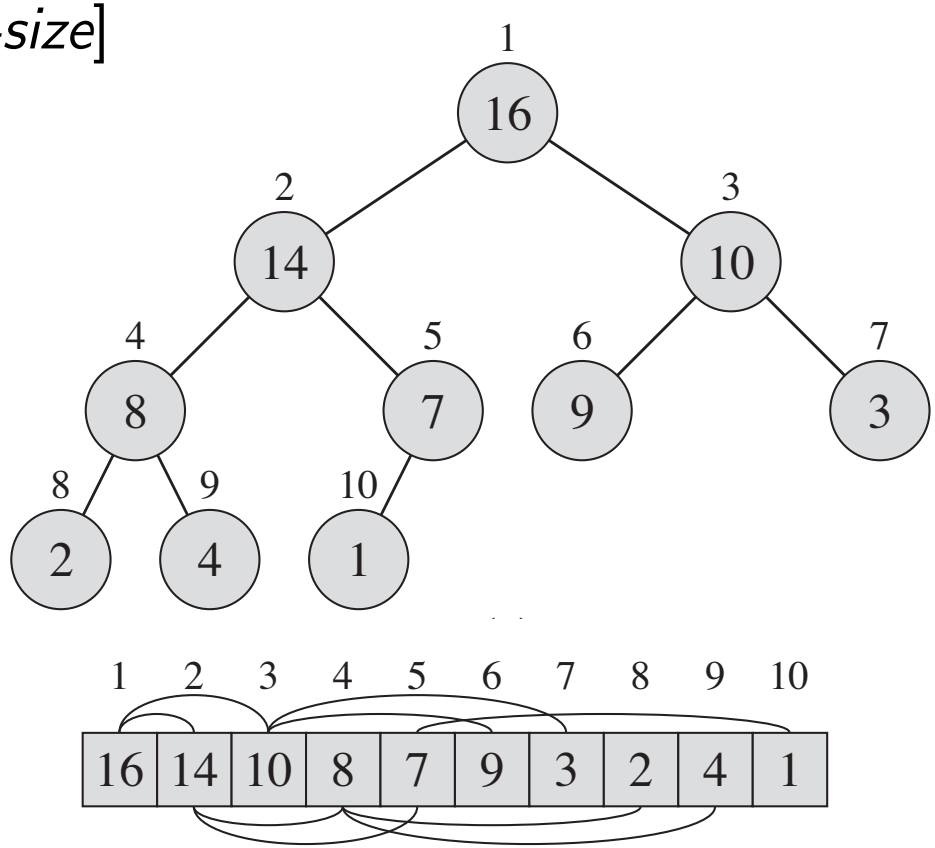
- Knoten entsprechen Arrayelementen
- Ebenenweise abgelegt
- Wurzel $\rightarrow$ Position 1
- Kinder der Wurzel $\rightarrow$ Positionen 2 und 3
- Kinder der Kinder der Wurzel
  $\rightarrow$ Positionen 4, 5, 6 und 7
- ...



Heap als Array (2)

Sei A ein Array, dass einen Heap repräsentiert.

- Zwei Attribute
  - $A.\mathit{length}$: Größe des Arrays (Anzahl Speicherplätze)
  - $A.\mathit{heap\text{-}size}$: Größe des Heaps (Anzahl Knoten)
- $A.\mathit{heap\text{-}size} \leq A.\mathit{length}$
- Nur das Teilarray $A[1\,..\,A.\mathit{heap\text{-}size}]$ gehört zum Heap
- LEFT(i): **return** $2i$
- RIGHT(i): **return** $2i + 1$
- PARENT(i): **return** $\lfloor i/2 \rfloor$



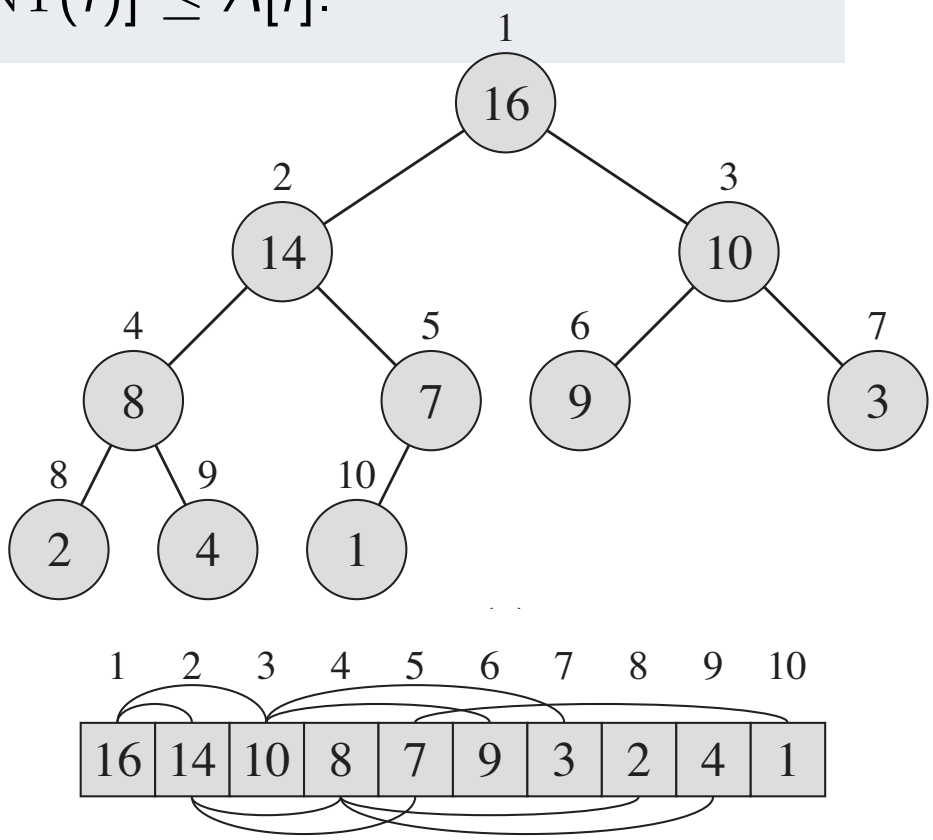
Heap-Eigenschaft

Definition

Für Heaps gilt eine **Heap-Eigenschaft**. Für jeden Knoten i ausser der Wurzel gilt:

- Bei einem **Max-Heap**: $A[\textsc{Parent}(i)] \geq A[i]$,
- Bei einem **Min-Heap**: $A[\textsc{Parent}(i)] \leq A[i]$.

- In einem Max-Heap steht das größte Element in der Wurzel
- In einem Min-Heap das kleinste Element



Heap-Operationen

Wir betrachten einen Max-Heap der Größe n.

Für das Sortieren

- MAX-HEAPIFY, $O(\lg n)$
  Hält Heap-Eigenschaft aufrecht
- BUILD-MAX-HEAP, $O(n)$
  Erstellt einen Heap aus einem Array
- HEAPSORT, $O(n \lg n)$
  Sortiert ein Array in-place

Für Prioritätswarteschlangen

- EXTRACT-MAX, $O(\lg n)$
  Liefert und entfernt Maximum der im Heap gespeicherten Werte
- INSERT, $O(\lg n)$
  Fügt Element in Heap ein

Outline

1. Bäume

2. Heapsort
2.1 Heap
2.2 Heapsort
2.3 Prioritätswarteschlangen

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Max-Heapify

Ziel

- Herstellen/Erhalten der Max-Heap-Eigenschaft
- Für alle i ausser der Wurzel: $A[\textsc{Parent}(i)] \geq A[i]$

Funktion

- Gegeben: ein i
- Vorher: $A[i]$ kann kleiner als Sohn sein
- Annahme: Linke und rechte Teilbäume von i sind Max-Heaps
- Max-Heapify lässt $A[i]$ "nach unten sinken"
- Nachher: Teilbaum mit Wurzel i ist Heap

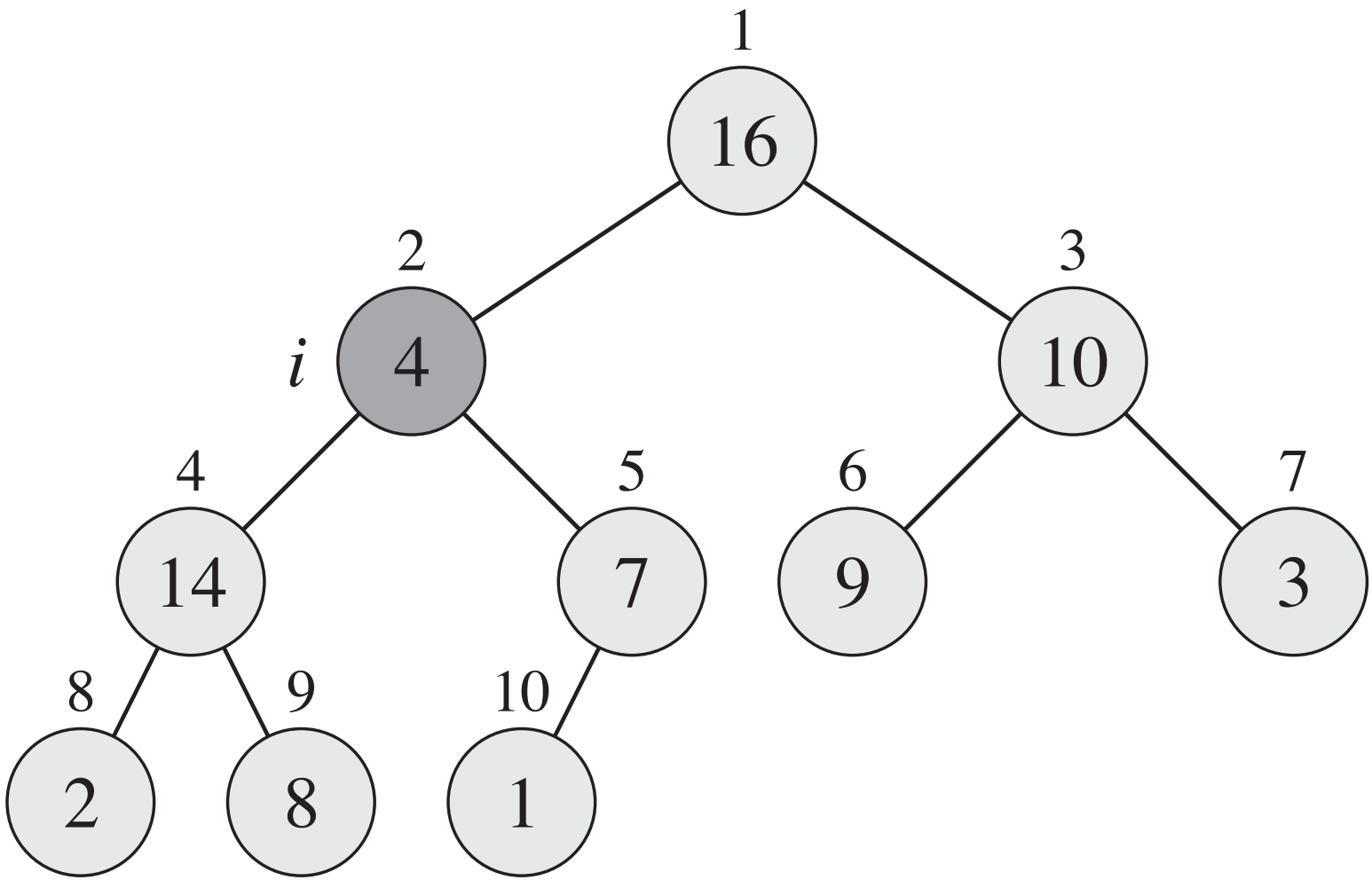
Max-Heapify (Pseudo-Code)

MAX-HEAPIFY(A, i)

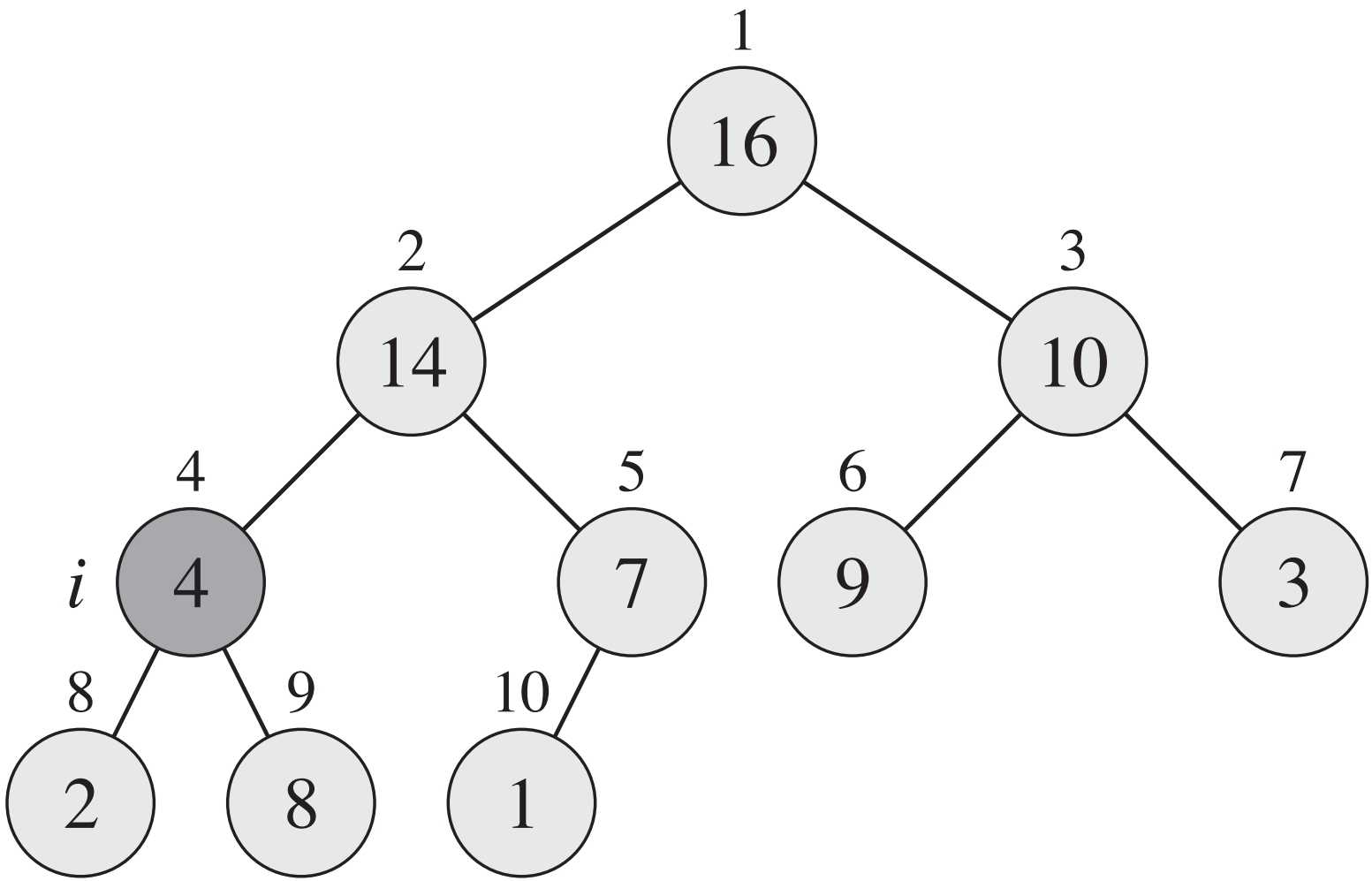
```
1:  l ← LEFT(i)
2:  r ← RIGHT(i)
3:  if l ≤ A.heap-size and A[l] > A[i]
4:      largest ← l
5:  else
6:      largest ← i
7:  if r ≤ A.heap-size and A[r] > A[largest]
8:      largest ← r
9:  if largest ≠ i
10:     exchange A[i] with A[largest]
11:     MAX-HEAPIFY(A, largest)
```

- Z. 1–8: Vergleiche $A[i]$, $A[\textsc{Left}(i)]$ und $A[\textsc{Right}(i)]$
- Z. 9–11: Wenn notwendig, vertausche $A[i]$ mit dem größeren seiner Kinder und wiederhole am entsprechenden Teilbaum
- Wiederholung endet wenn Teilbaum ein Max-Heap

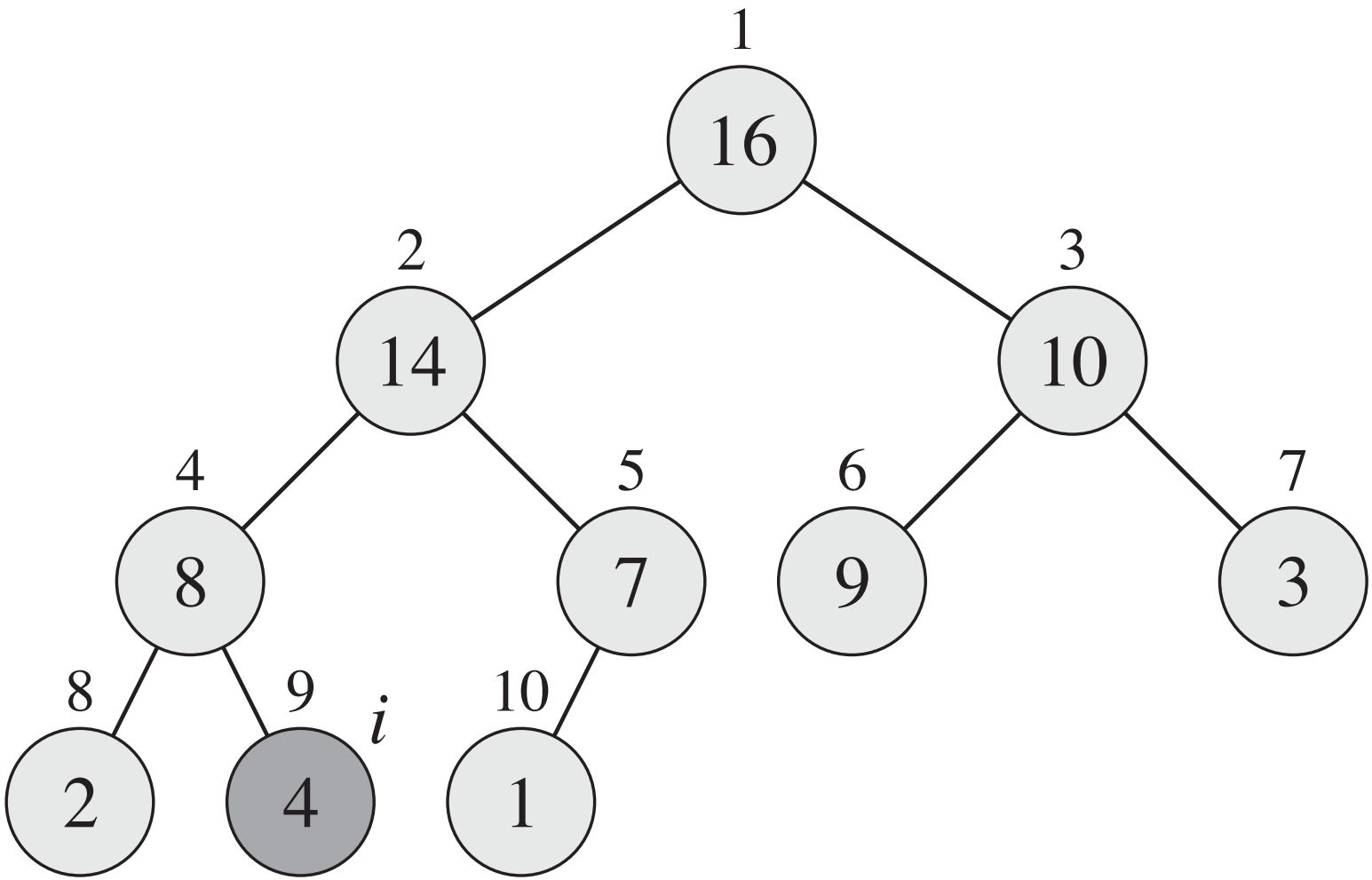
Beispiel (1)



Beispiel (2)



Beispiel (3)



Max-Heapify: Analyse

Laufzeit von MAX-HEAPIFY für Teilbaum der Größe n:

- Maximum suchen + tauschen (Z. 1–10): $\Theta(1)$
- Rekursiver Aufruf (Z. 11), Größe des Teilbaums?
  - Maximale Größe ist $2n/3$
  - Unterste Ebene ganz voll: $n/2$
  - Unterste Ebene halb voll: $2n/3$

Es ergibt sich also für die *Worst Case-Laufzeit* folgende Rekurrenz:

$$T(n) \leq T(2n/3) + \Theta(1)$$

- Das Master Theorem kann angewendet werden
- $a = 1$, $b = 3/2$, $\log_b a = 0$, $f(n) = \Theta(1) = \Theta(n^0)$
  $\rightarrow$ Fall 2: $T(n) = \Theta(\lg n)$
- Somit Laufzeit von MAX-HEAPIFY: $O(\lg n)$
- Oder in Abhängigkeit der Höhe: $O(h)$

Build-Max-Heap

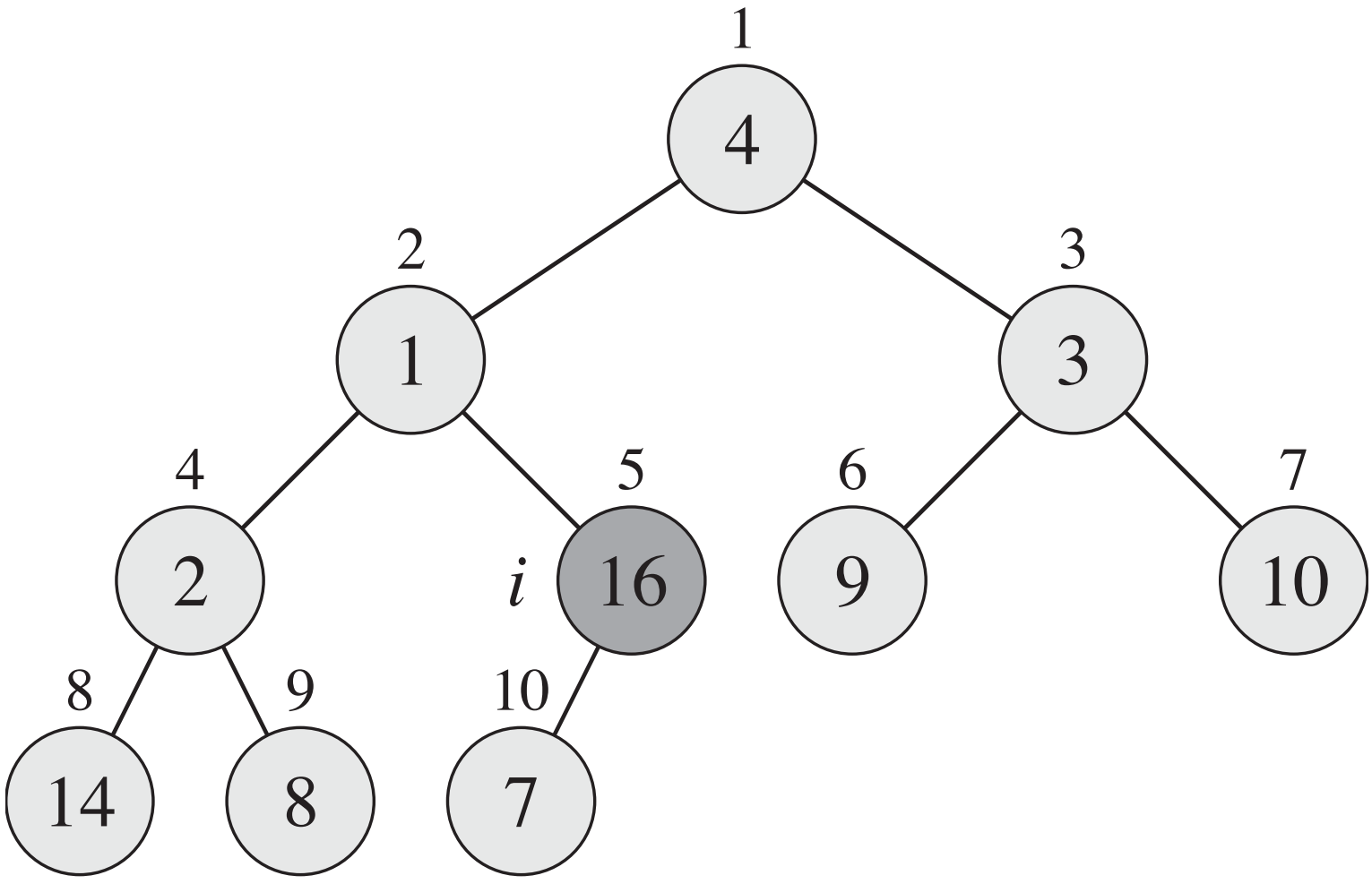
Die folgende Prozedur erzeugt einen Heap aus einem gegebenen Array A.

BUILD-MAX-HEAP(A)

1: $A.heap\text{-}size \leftarrow A.length$
2: **for** $i \leftarrow \lfloor A.length/2 \rfloor$ **downto** 1
3: MAX-HEAPIFY(A, i)

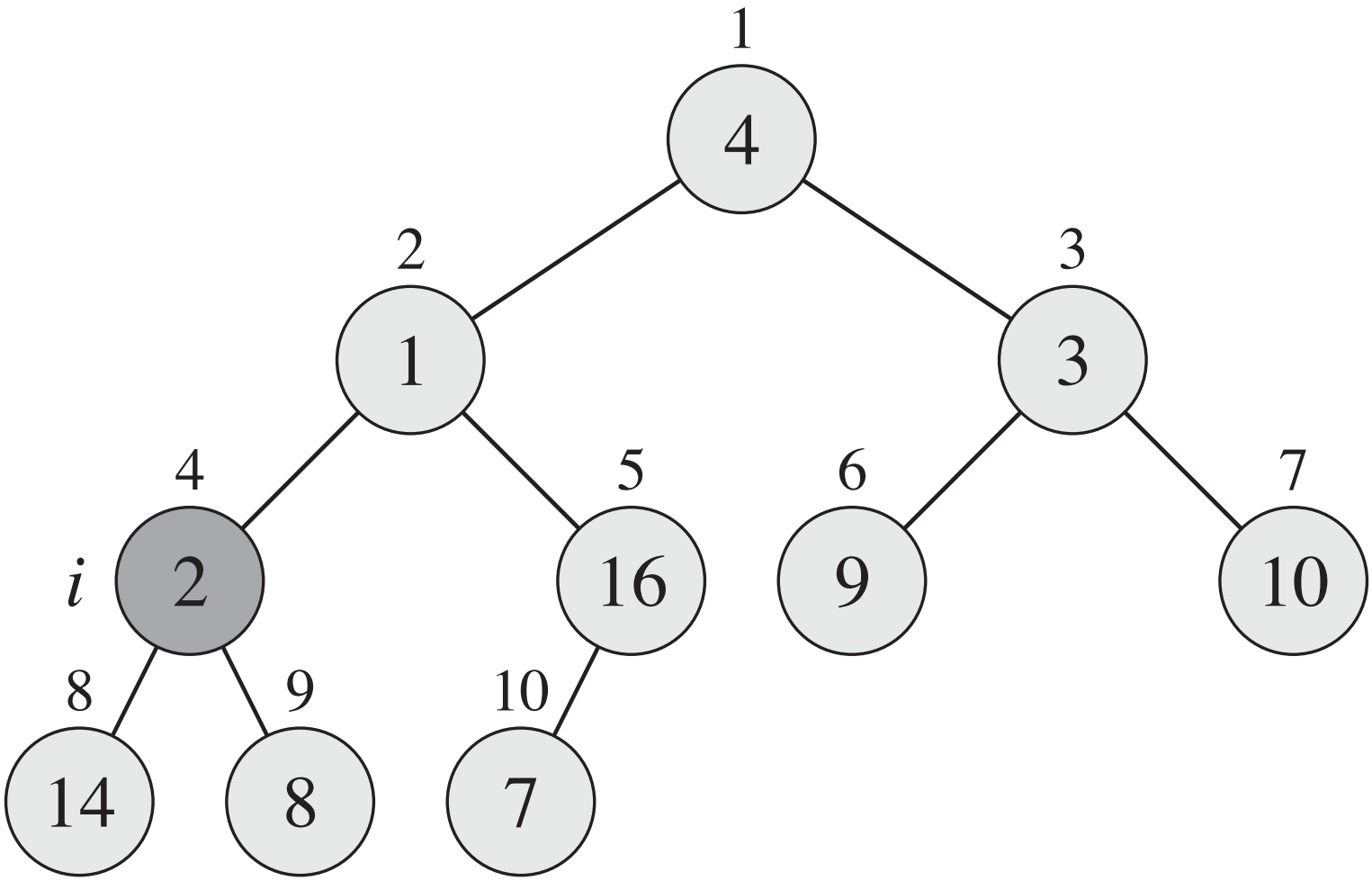
- MAX-HEAPIFY von Blättern aufwärts zur Wurzel aufgerufen
- Schleifeninvariante: Zu Beginn jeder Iteration der For-Schleife, sind $i+1, \ldots, n$ Wurzeln eines Max-Heaps
- Induktionsanfang ($i = \lfloor n/2 \rfloor$)
  - $A[(\lfloor n/2 \rfloor + 1) \,..\, n]$ sind allesamt Blätter $\rightarrow$ triviale Heaps
- Induktionsschritt
  - Die Kindknoten von $A[i]$ sind haben Indizes $> i$ und sind somit Max-Heaps nach Induktionsannahme
  - Unter dieser Bedingung stellt MAX-HEAPIFY die Heap-Eigenschaft für $A[i]$ her

Beispiel (1)

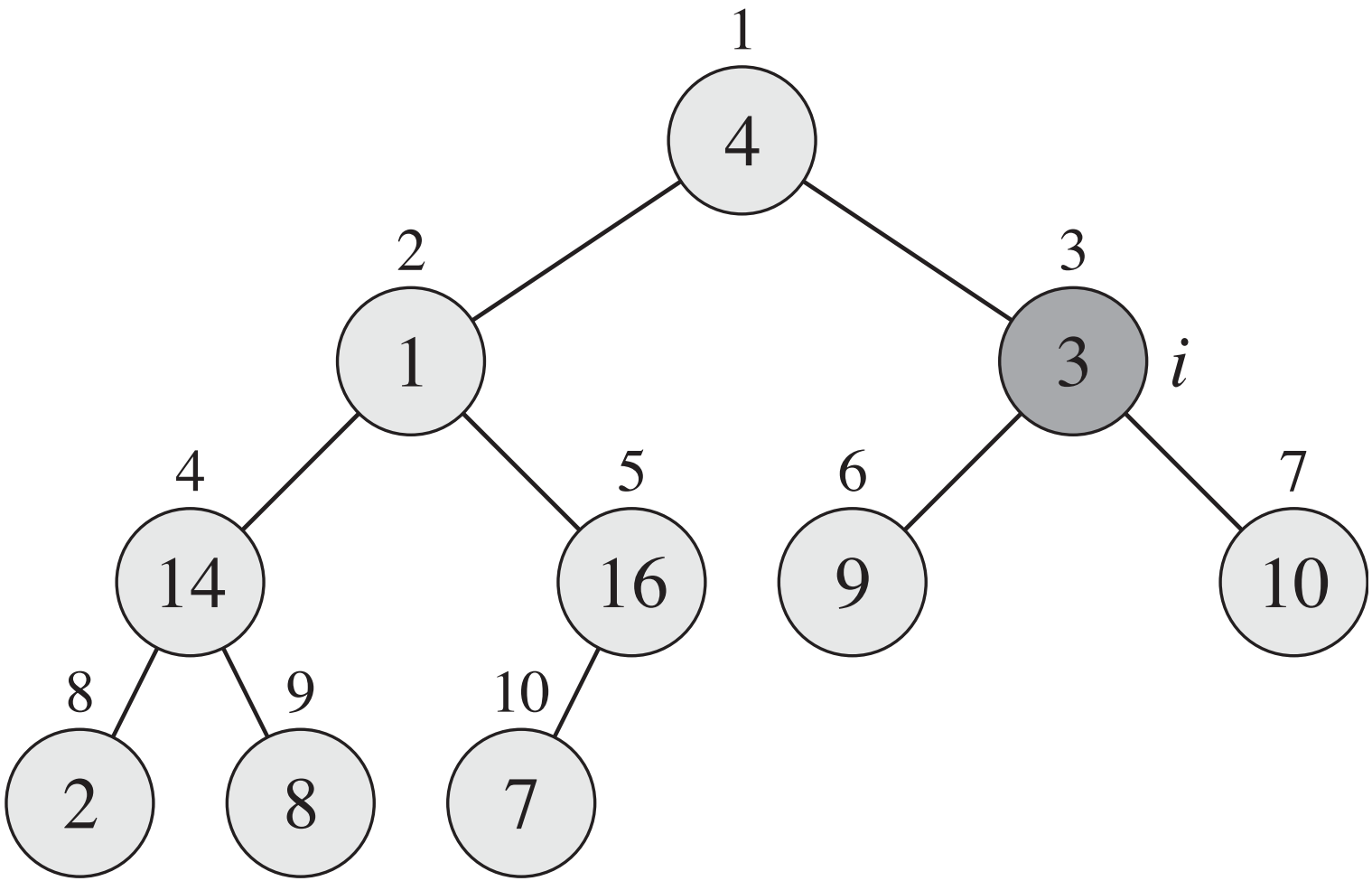


A	4	1	3	2	16	9	10	14	8	7

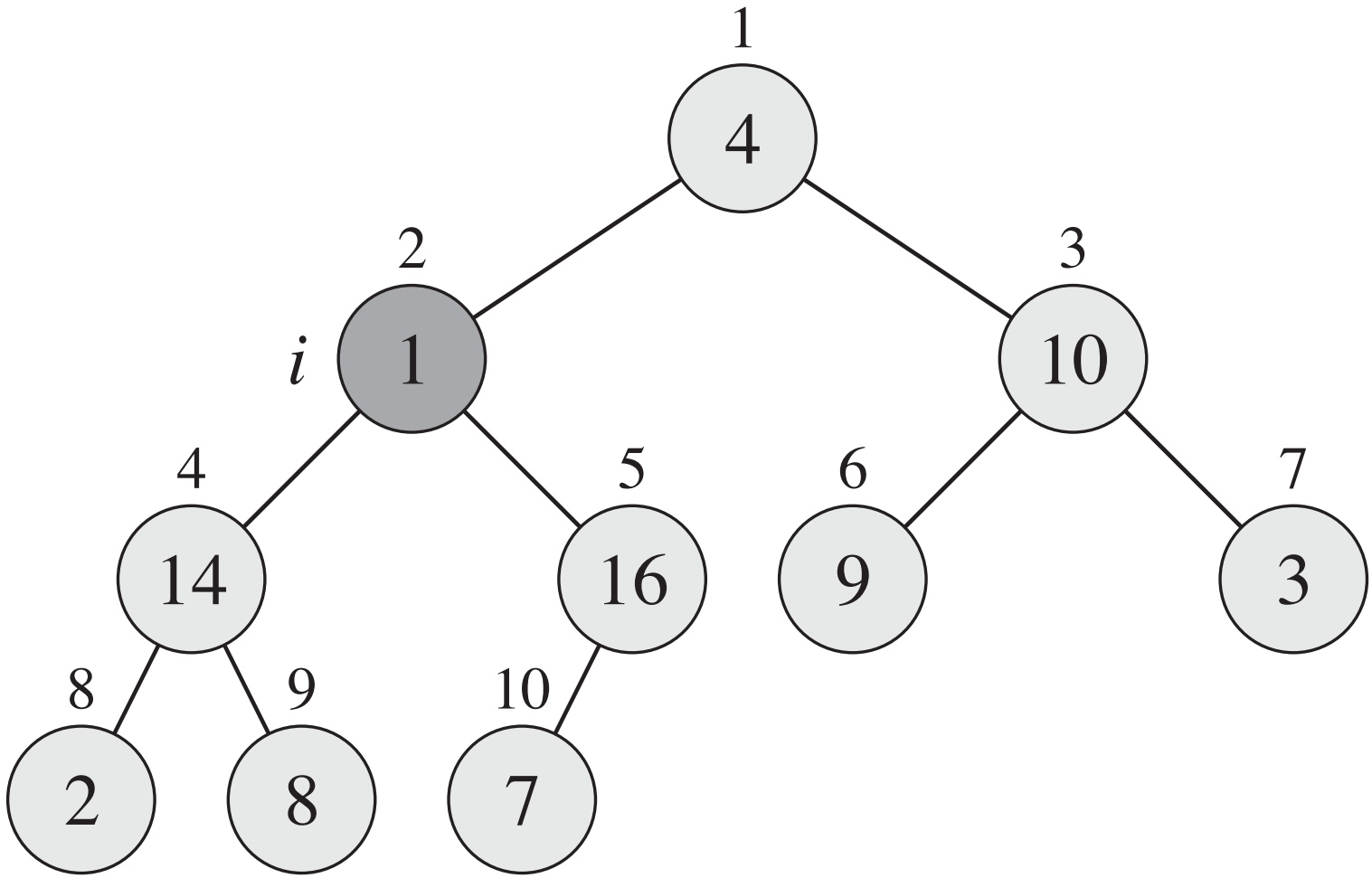
Beispiel (2)



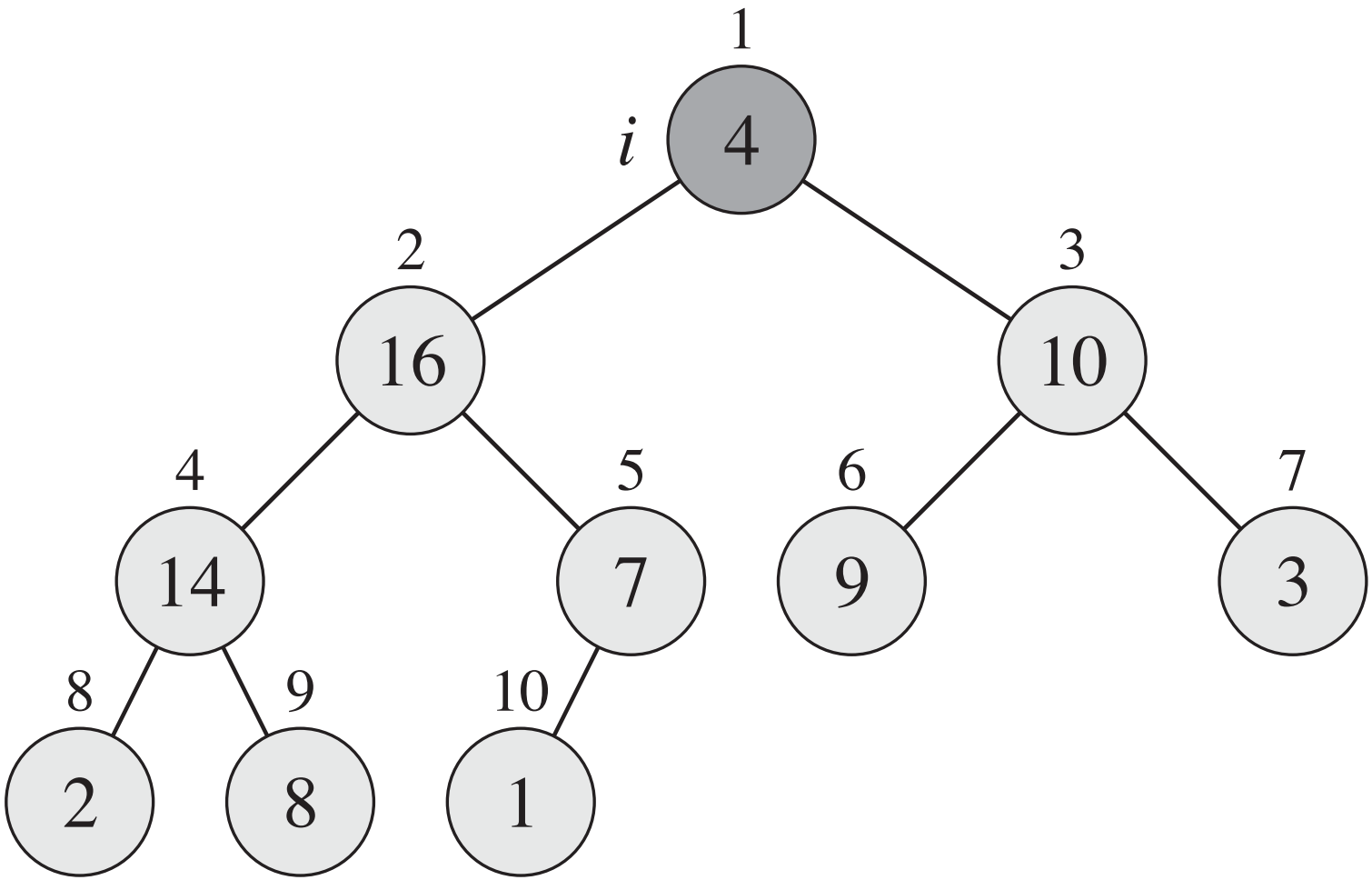
Beispiel (3)



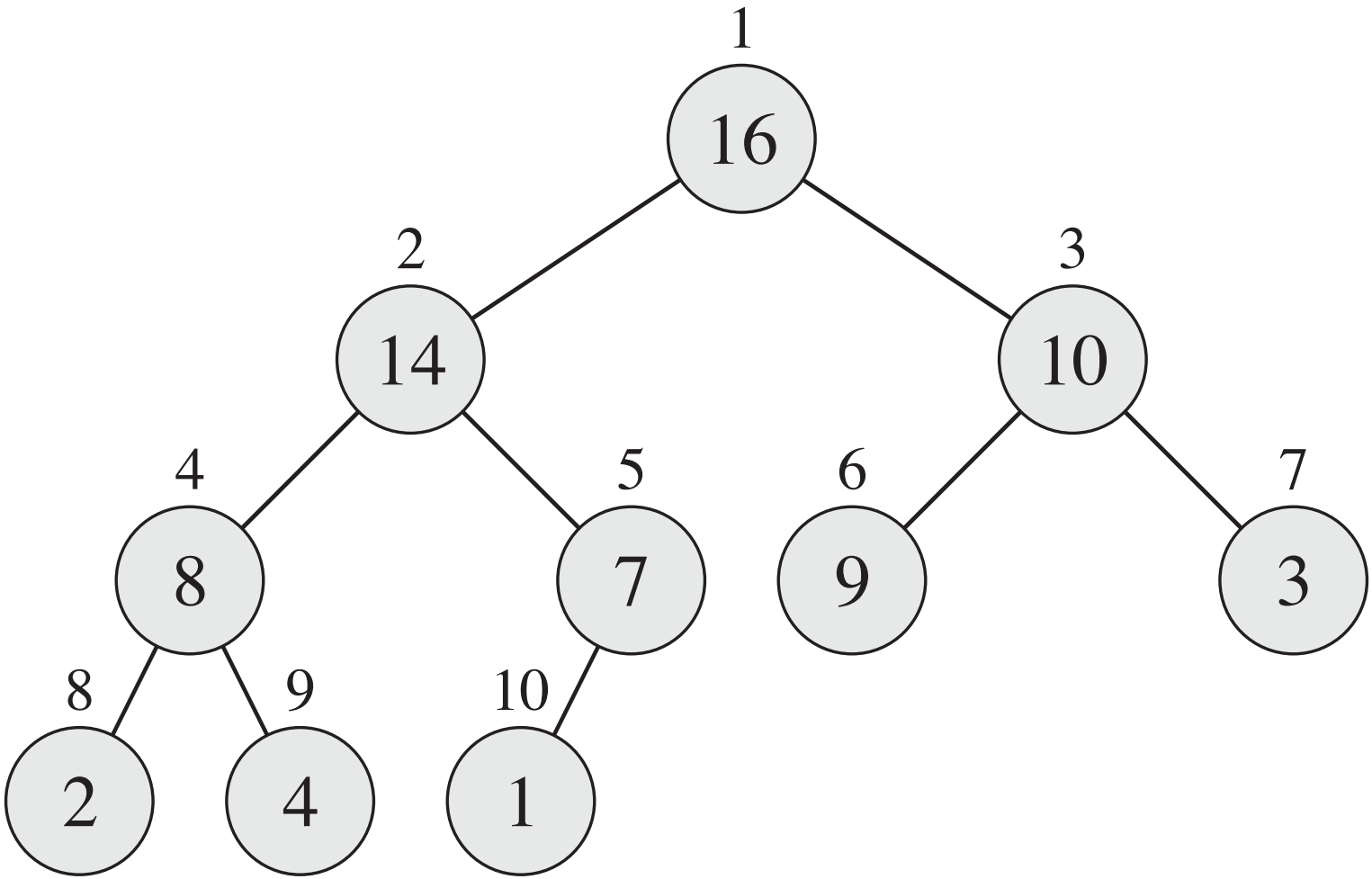
Beispiel (4)



Beispiel (5)



Beispiel (6)



Build-Max-Heap: Analyse (1)

Eine einfache obere Schranke

- Jeder Aufruf von MAX-HEAPIFY kostet $O(\lg n)$
- Es gibt $O(n)$ solche Aufrufe
- Laufzeit: $O(n \lg n)$

Geht es besser?

- Diese Schranke ist korrekt, jedoch nicht eng
- Beobachtung: Laufzeit von MAX-HEAPIFY variiert mit der Höhe
- Viele Knoten haben kleine Höhe
- Wieviele genau?

Build-Max-Heap: Analyse (2)

Lemma

In einem n-elementigen Heap gibt es höchstens $\lceil n/2^{h+1} \rceil$ Knoten der Höhe h.

- Im vollständigem Heap der Höhe H
  - 2^d Knoten der Tiefe d
  - Da vollständig: $h = H - d$ und somit 2^{H-h} Knoten der Höhe h
  - $n = 2^{H+1} - 1$ und somit $\lceil n/2^{h+1} \rceil = \lceil (2^{H+1} - 1)/(2^{h+1}) \rceil = 2^{H-h}$
- Allgemein: Übung (per Induktion über h)

Build-Max-Heap: Analyse (3)

Die Worst Case-Laufzeit ist somit:

$$
\begin{aligned}
T(n) &= \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\
&= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \\
&\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\
&= O(n \cdot 2) \\
&= O(n).
\end{aligned}
$$

Die vorletzte Gleichung gilt da für $|r| < 1$ (hier: $r = 1/2$):

$$\sum_{k=0}^{\infty} k r^k = r/(1-r)^2.$$

Somit hat Build-Max-Heap eine Laufzeit von $O(n)$.

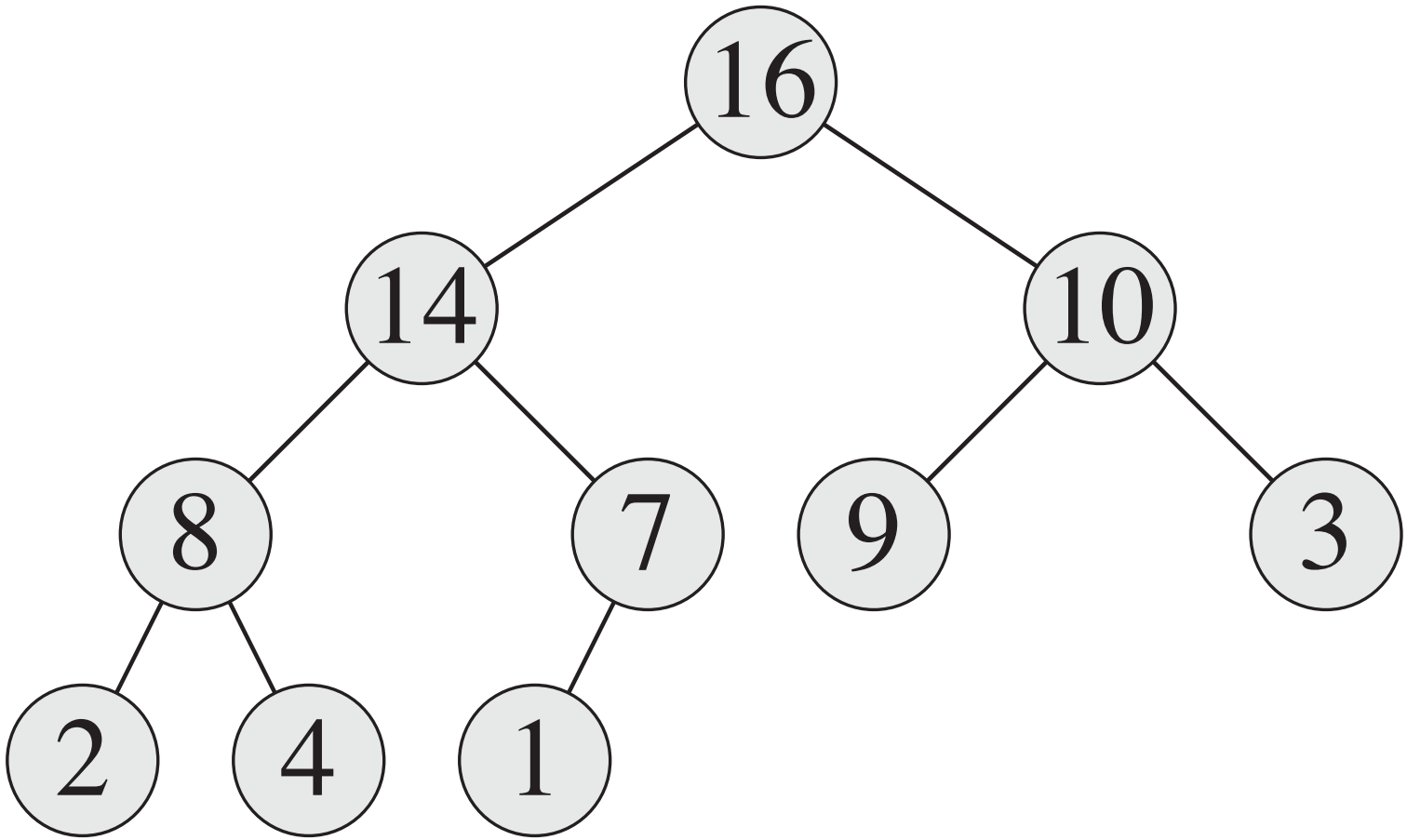
Heapsort

HEAPSORT(A)

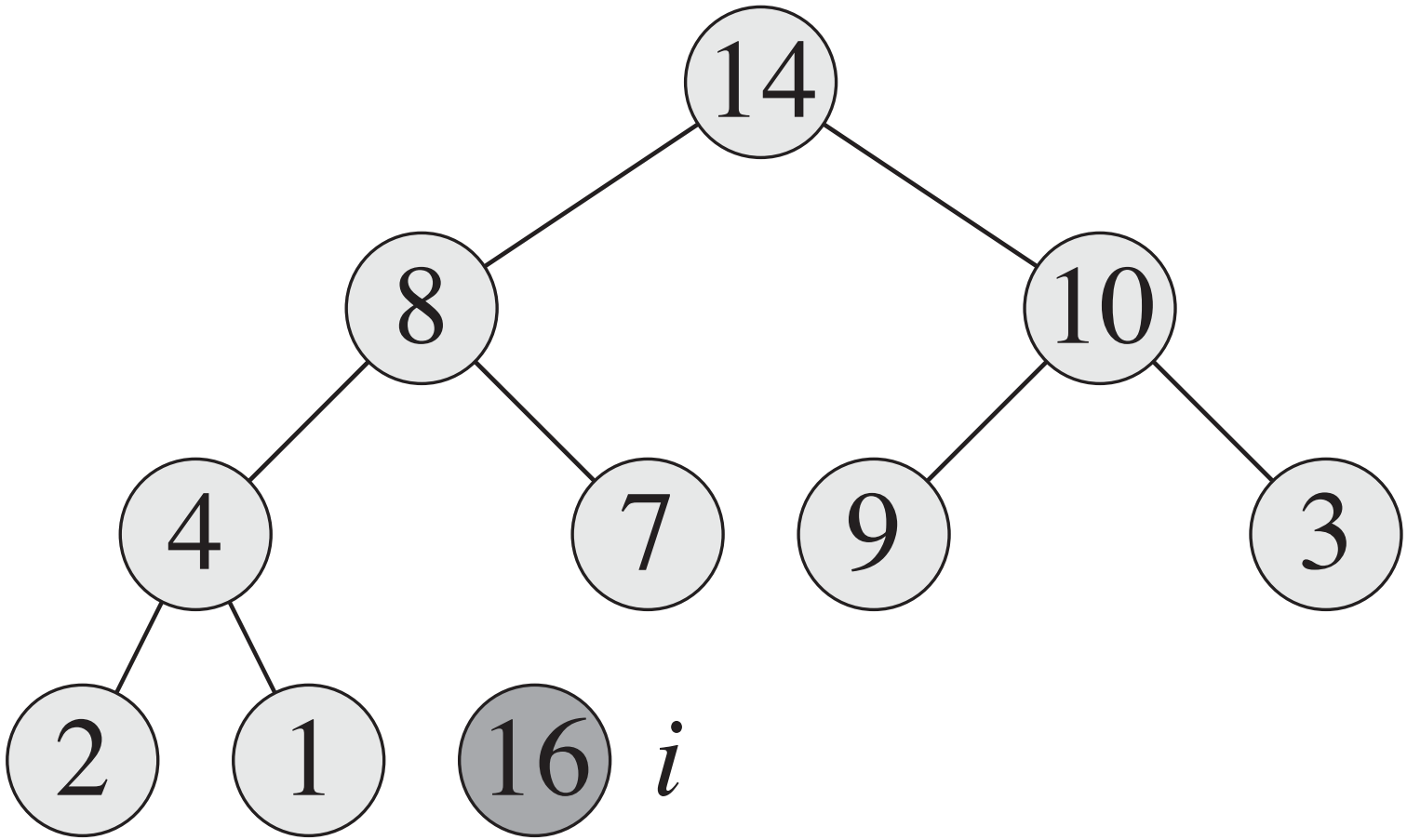
1: BUILD-MAX-HEAP(A)
2: **for** $i \leftarrow A.\mathit{length}$ **downto** 2
3: exchange $A[1]$ with $A[i]$
4: $A.\mathit{heap\text{-}size} \leftarrow A.\mathit{heap\text{-}size} - 1$
5: MAX-HEAPIFY($A, 1$)

- Heapsort erstellt zunächst einen Heap
- Damit steht das größte Element an der Wurzel
- Tausche Wurzel ($= A[1]$) mit letztem Element im Heap ($= A[i] = A[A.\mathit{heap\text{-}size}]$)
  $\rightarrow$ letztes Element an richtiger Position
- Verkleinere den Heap $\rightarrow$ letztes Element "entfernt"
- Wiederherstellen der Heap-Eigenschaft
- ...

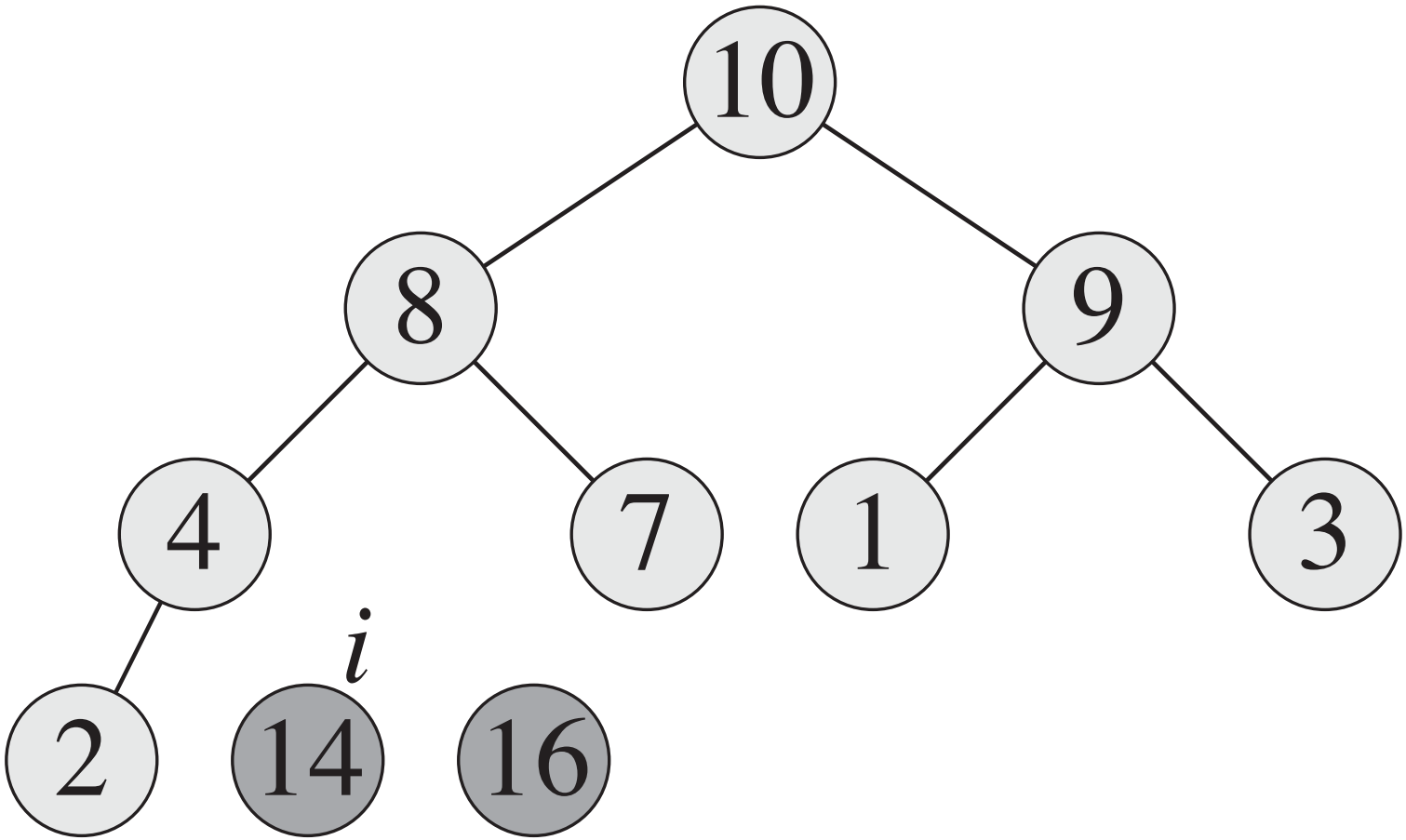
Beispiel (1)



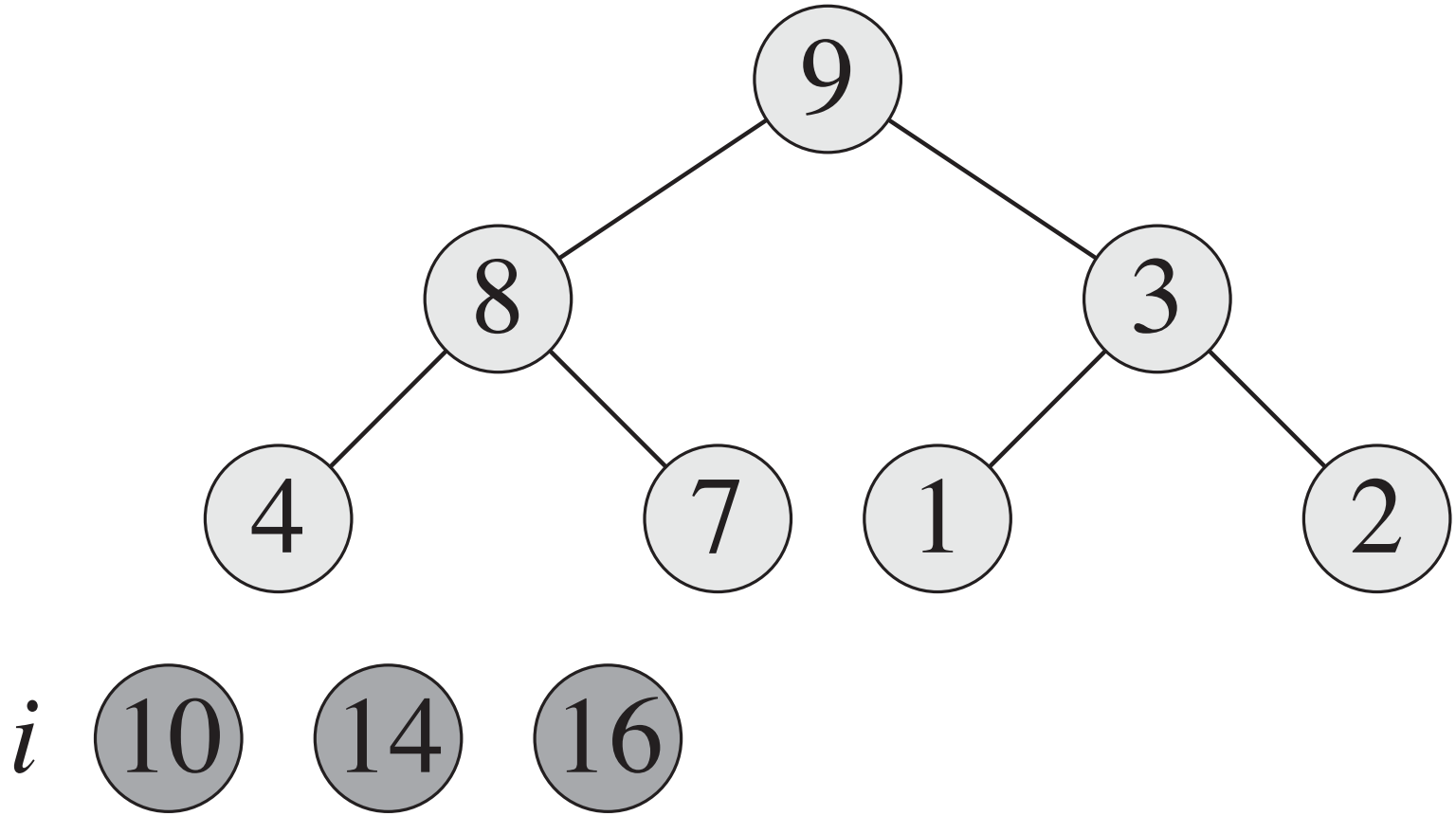
Beispiel (2)



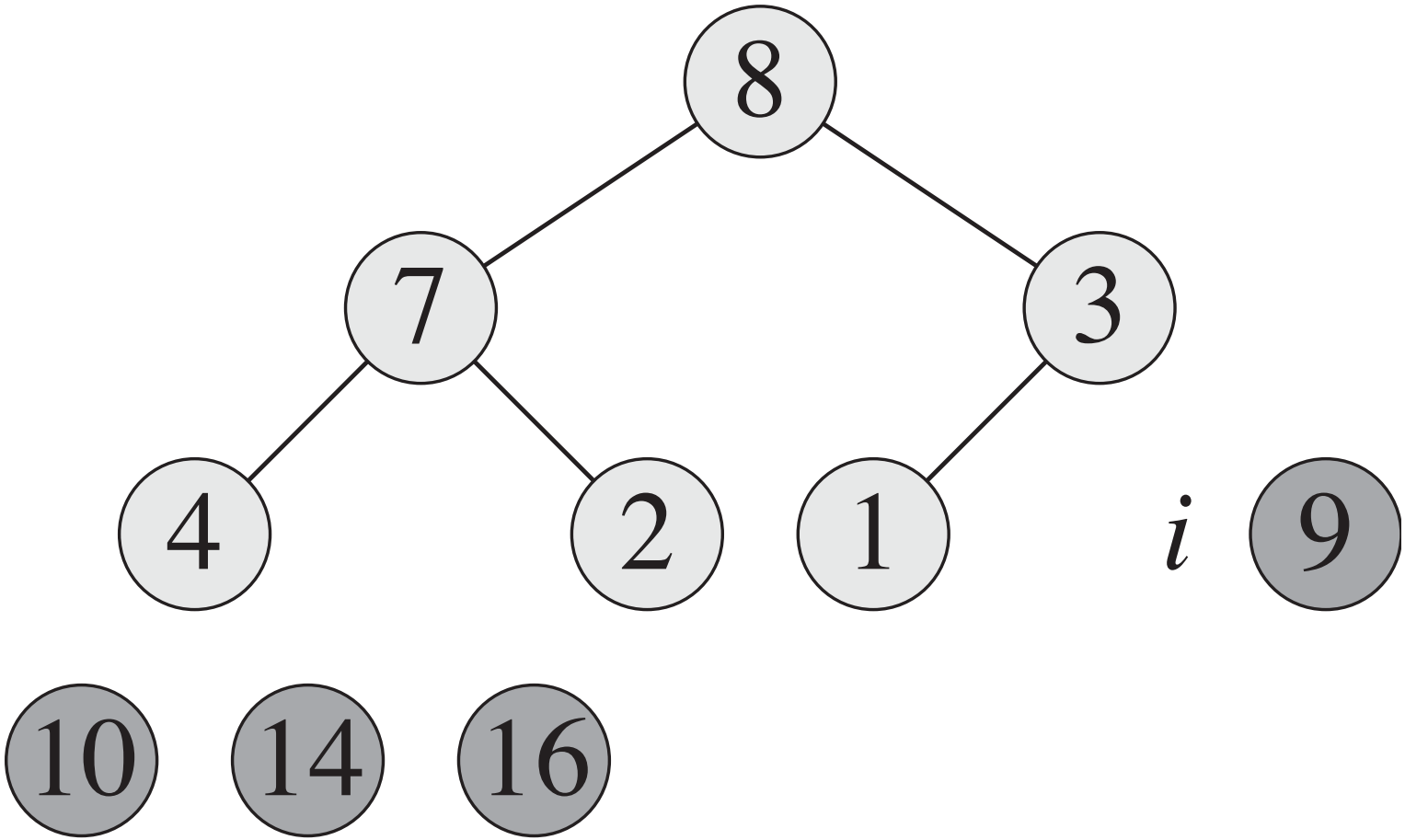
Beispiel (3)



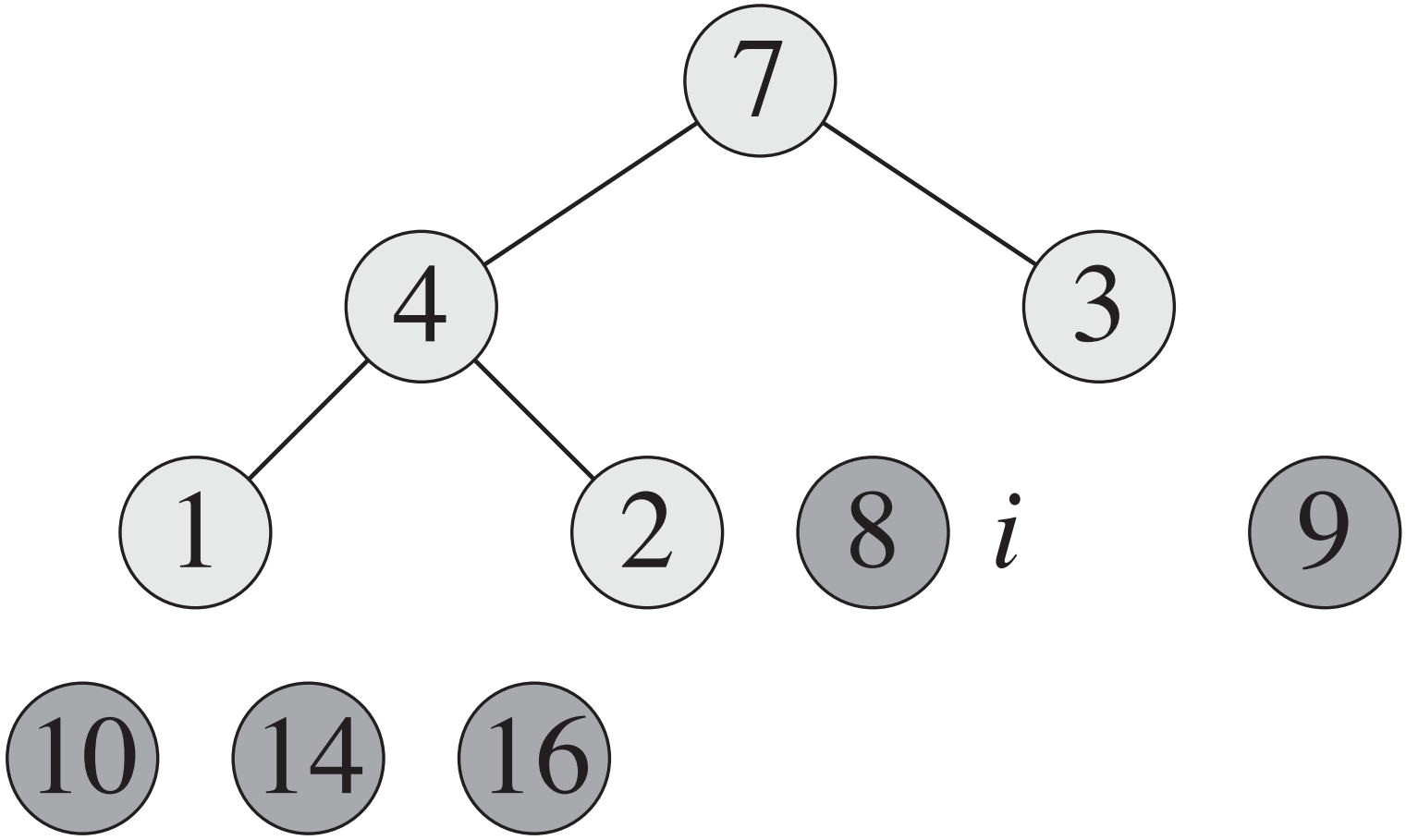
Beispiel (4)



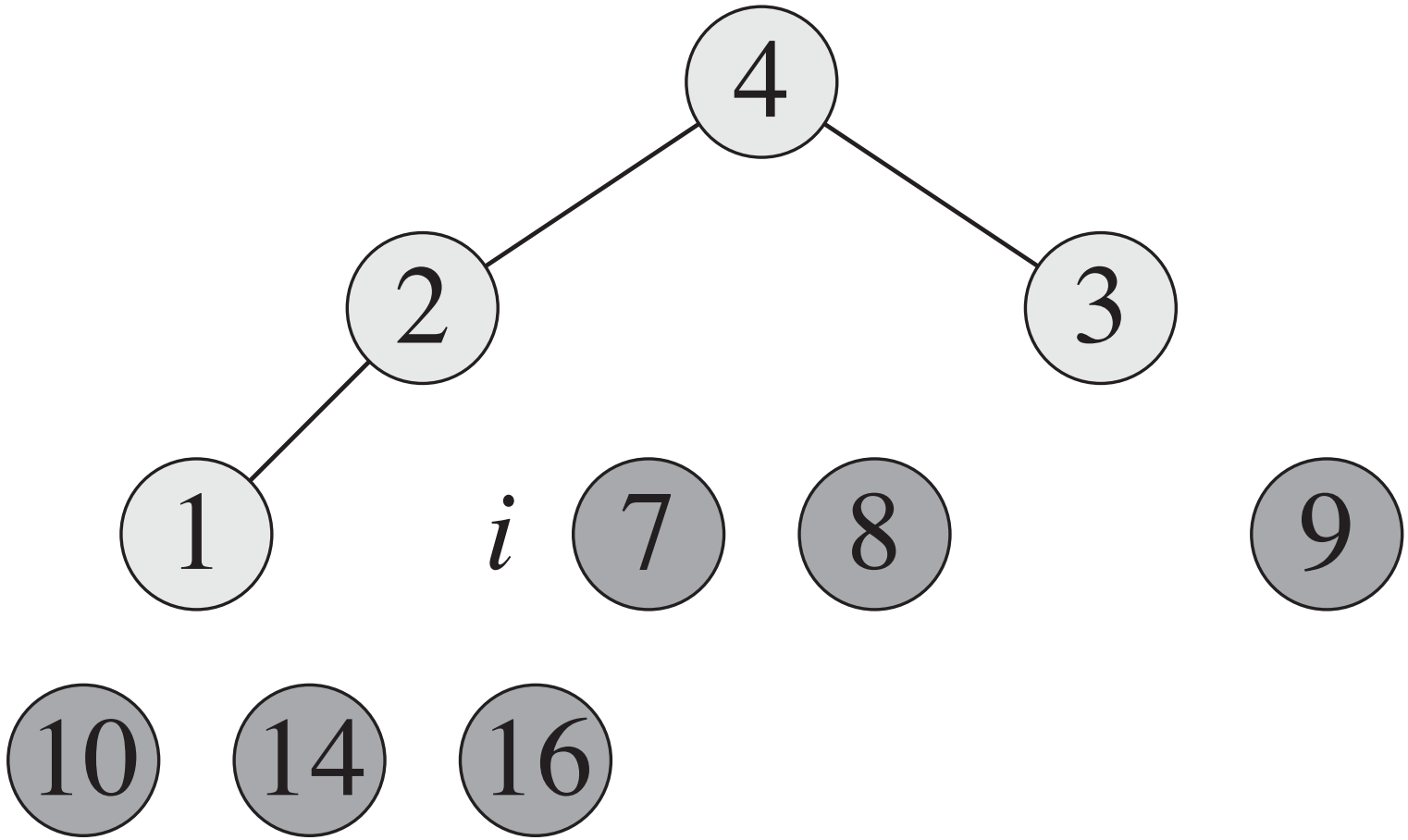
Beispiel (5)



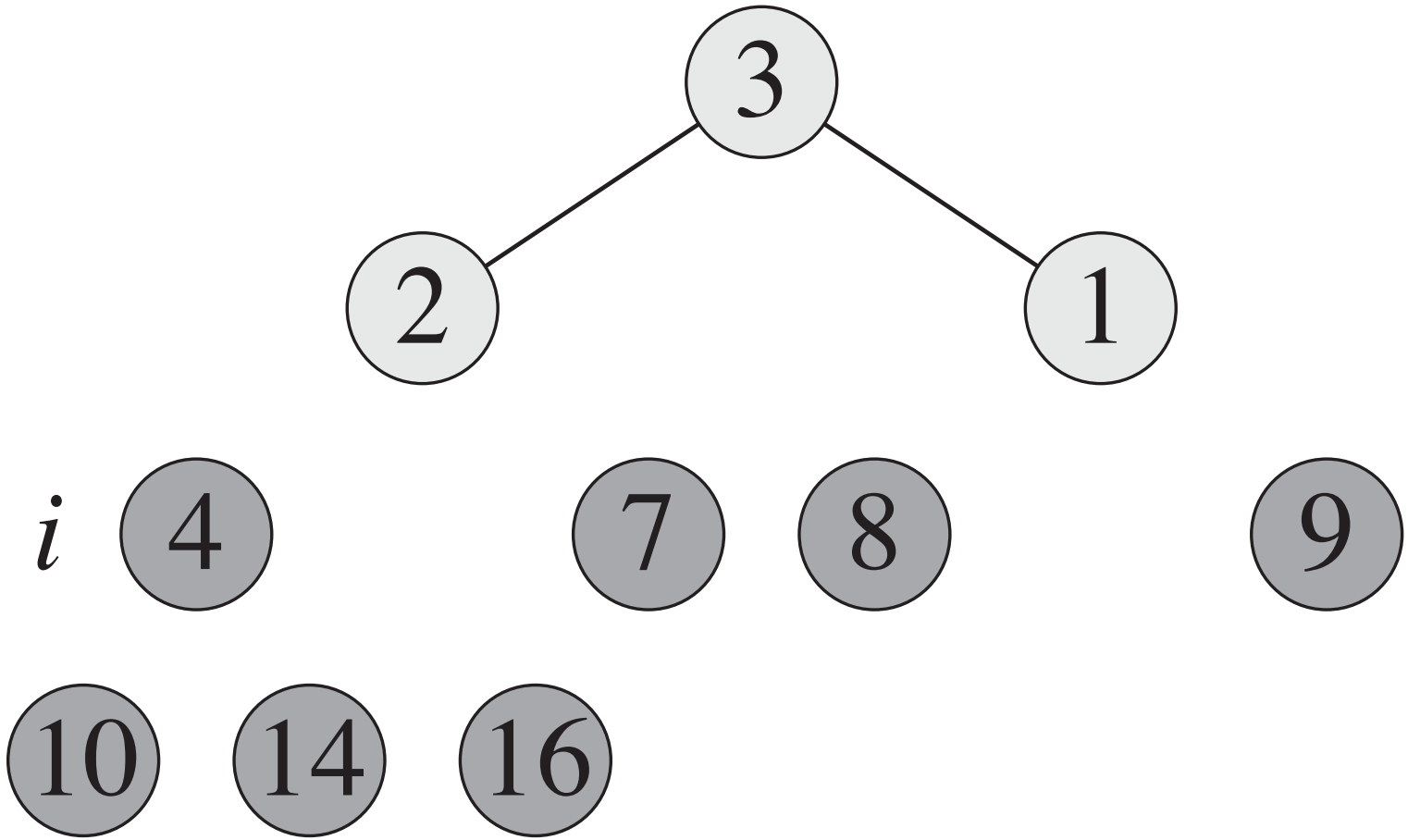
Beispiel (6)



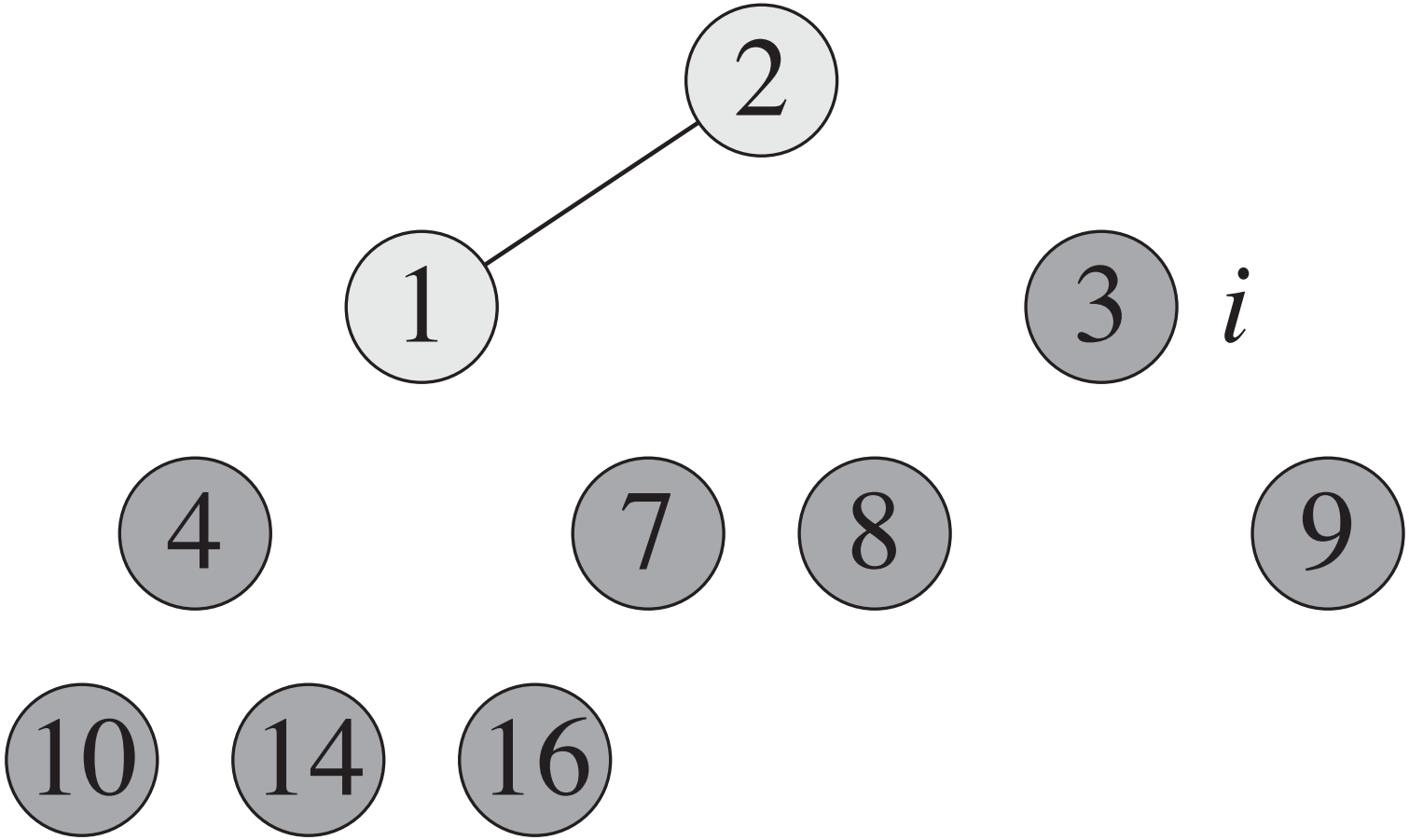
Beispiel (7)



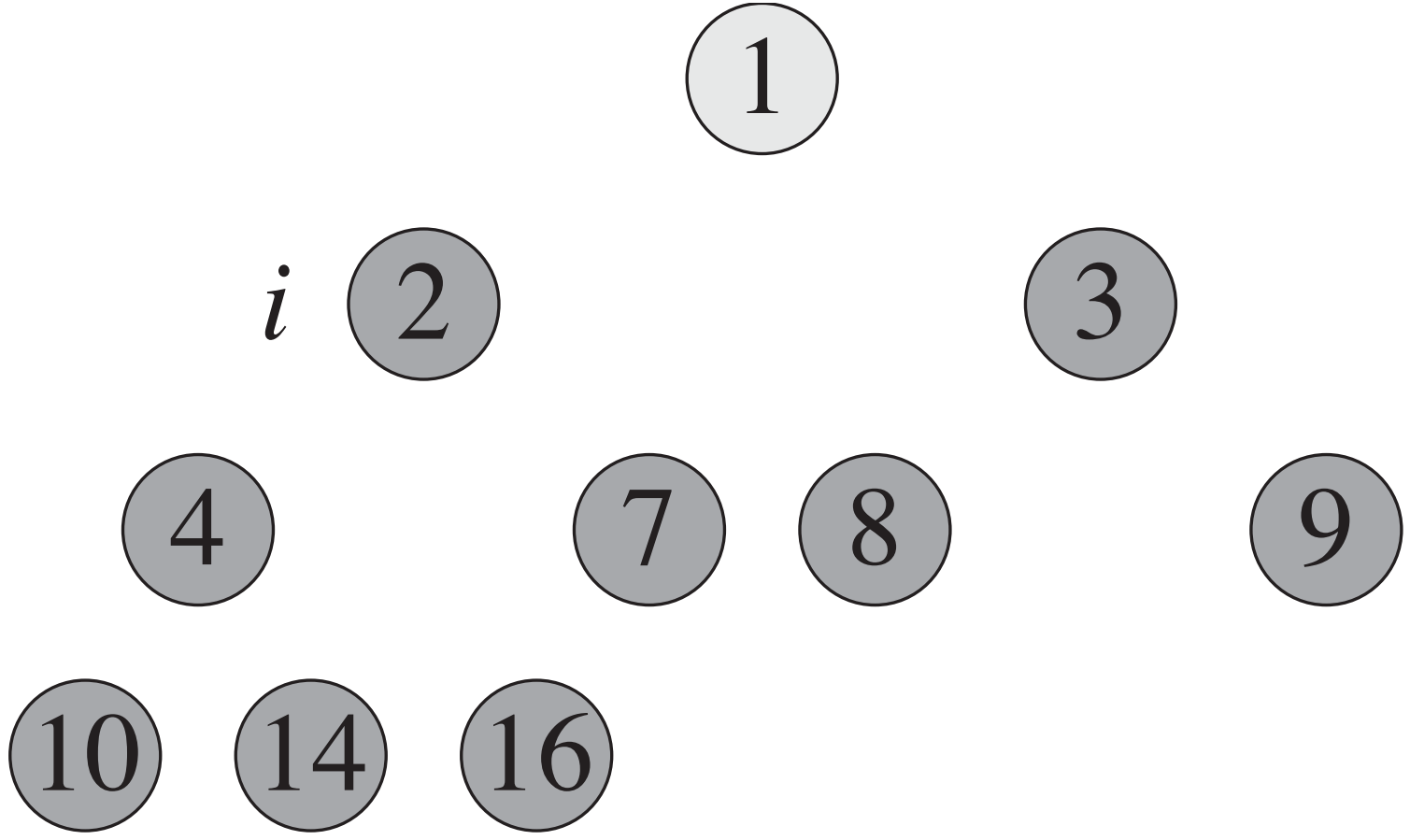
Beispiel (8)



Beispiel (9)



Beispiel (10)



A	1	2	3	4	7	8	9	10	14	16

Heapsort: Analyse

HEAPSORT benötigt

$$O(n \lg n)$$

Zeit, da

- Aufruf von BUILD-MAX-HEAP ist $O(n)$,
- Jeder der $n - 1$ Aufrufe von MAX-HEAPIFY ist $O(\lg n)$.

QUICKSORT wird schlechteres Worst Case-Verhalten haben, ist aber in der Praxis meistens schneller als Heapsort.

Outline

1. Bäume

2. Heapsort
2.1 Heap
2.2 Heapsort
2.3 Prioritätswarteschlangen

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Prioritätswarteschlange

Eine Prioritätswarteschlange ist eine Datenstruktur. Sie verwaltet eine dynamische Menge von Elementen, von denen jedes mit einem Schlüssel ausgestattet ist.

Operationen

- $\textsc{Insert}(S, x)$
  Fügt ein Element x in S ein; äquivalent $S \leftarrow S \cup \{ x \}$
- $\textsc{Maximum}(S)$
  Liefert das Element von S mit dem größten Schlüssel
- $\textsc{Extract-Max}(S)$
  Liefert das Element von S mit dem größten Schlüssel und entfernt es aus S

Anmerkungen

- Unsere Definition entspricht einer Max-Prioritätswarteschlange
  - Anwendung z.B. zum Prioritätsschedulung
- Dual: Min-Prioritätswarteschlangen
  - Anwendung z.B. zur Implementierung der MERGE-Operation zum Mischen von (mehreren) sortierten Listen

Umsetzung

- Mit Heaps können wir Prioritätswarteschlangen effizient umsetzen
- Guter Kompromiss zwischen schnellem Einfügen aber langsamem Löschen und umgekehrt
  - Beide Operationen $O(\lg n)$
  - Was sind Alternativen? (Übung)
- Der Heap ist eine Datenstruktur, Prioritätswarteschlangen ebenso $\rightarrow$ Wir verwenden Datenstrukturen, um andere Datenstrukturen zu implementieren

Heap-Maximum und Heap-Extract-Max

Umsetzung ähnlich der Sortierphase von Heapsort.

HEAP-MAXIMUM(A)

```
1: return A[1]
```

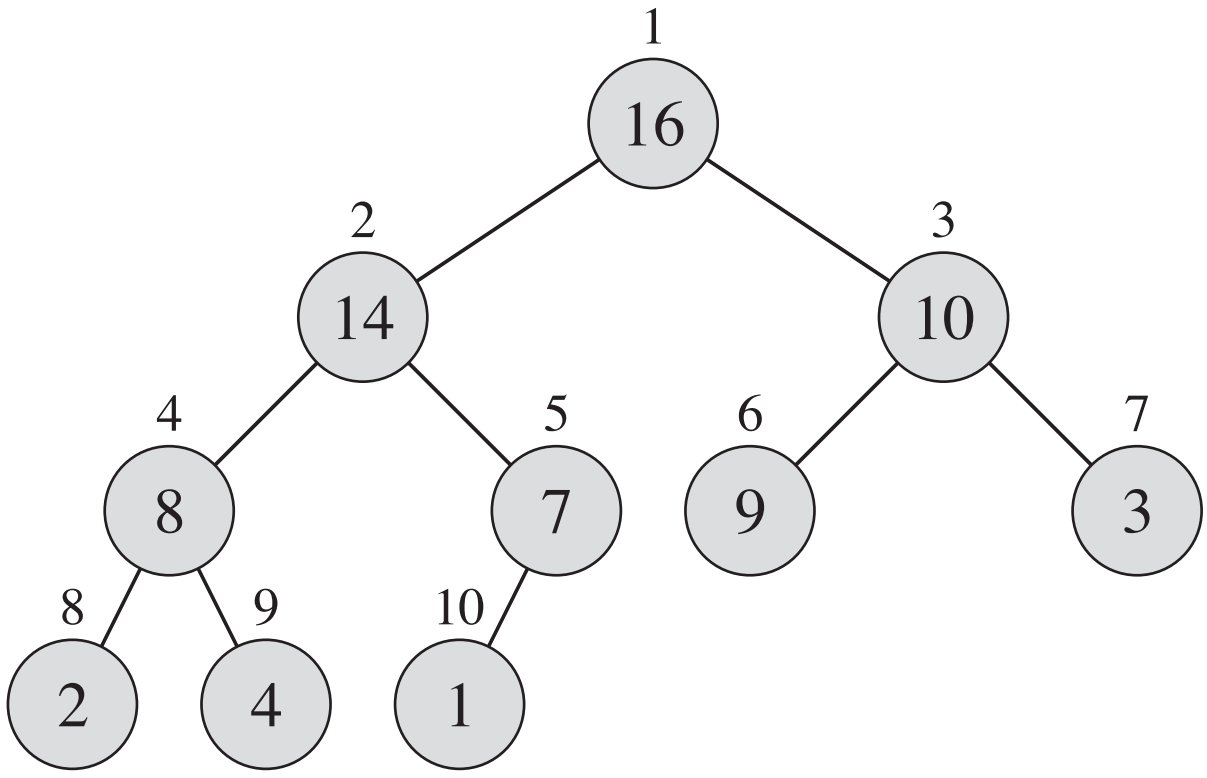
Laufzeit: $\Theta(1)$

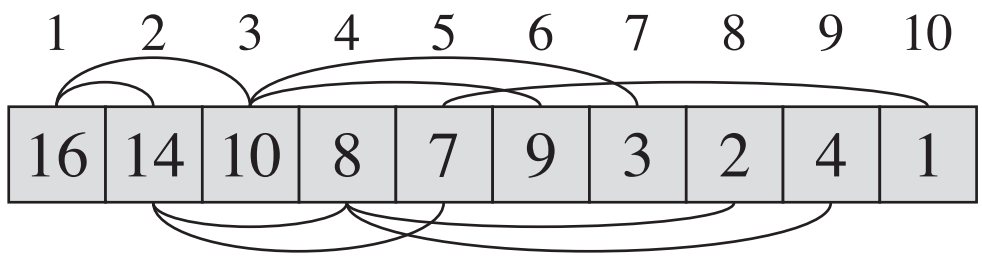
HEAP-EXTRACT-MAX(A)

```
1: if A.heap-size < 1
2:     error "heap underflow"
3: max ← A[1]
4: A[1] ← A[A.heap-size]
5: A.heap-size ← A.heap-size − 1
6: MAX-HEAPIFY(A, 1)
7: return max
```

Laufzeit: $O(\lg n)$

Beispiel (Heap-Extract-Max)





Heap-Insert

Idee

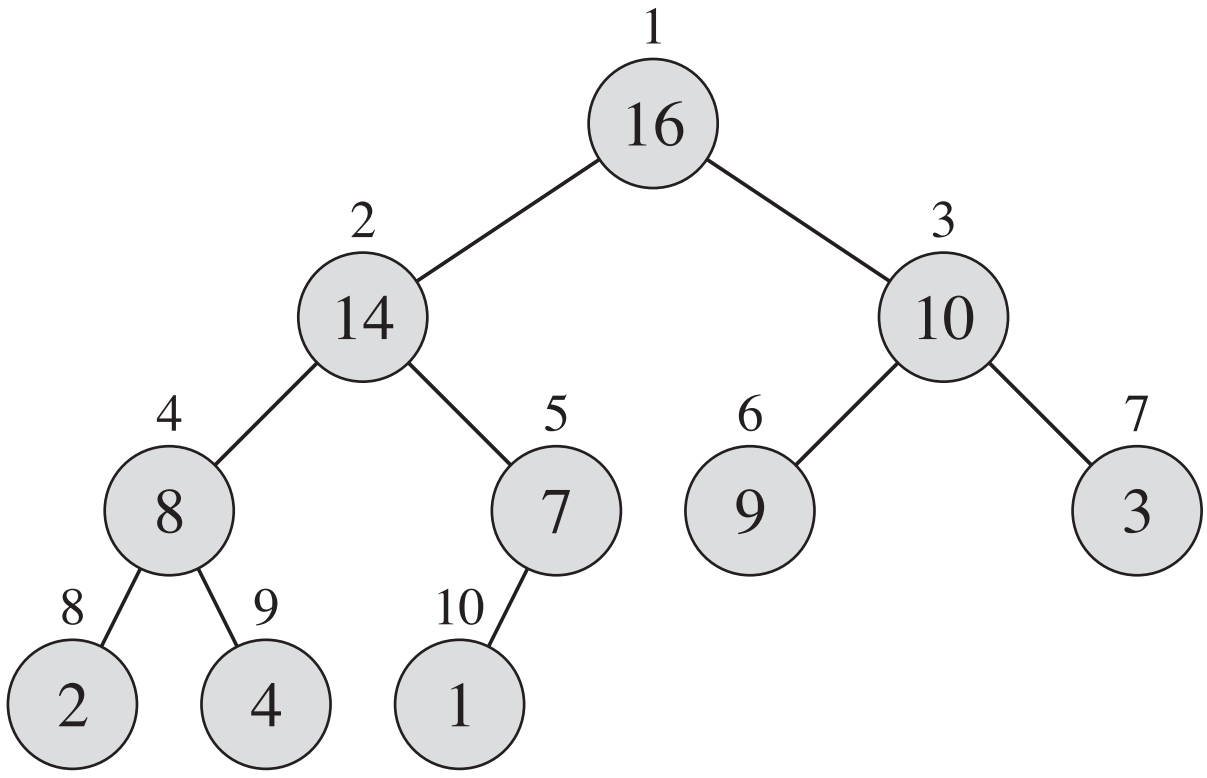
- Einfügen des neuen Elementes ans Ende des Arrays
- Dann wie Insertion-Sort auf dem Pfad bis zur Wurzel richtigen Platz suchen

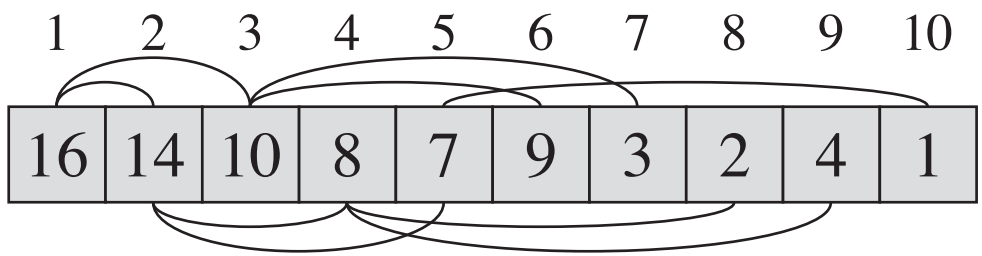
HEAP-INSERT(A, *key*)

1: $A.\textit{heap-size} \leftarrow A.\textit{heap-size} + 1$
2: $i \leftarrow A.\textit{heap-size}$
3: **while** $i > 1$ and $A[\text{PARENT}(i)] < \textit{key}$
4: $\quad A[i] \leftarrow A[\text{PARENT}(i)]$
5: $\quad i \leftarrow \text{PARENT}(i)$
6: $A[i] \leftarrow \textit{key}$

Laufzeit: $O(\lg n)$

Beispiel (Heap-Insert, 15)





Outline

1. Bäume

2. Heapsort

3. Quicksort
3.1 Algorithmus
3.2 Analyse
3.3 Randomisiertes Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Outline

1. Bäume

2. Heapsort

3. Quicksort
3.1 Algorithmus
3.2 Analyse
3.3 Randomisiertes Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Überblick

Eigenschaften

- Teile-und-Herrsche-Algorithmus
- Worst Case: $\Theta(n^2)$
- Average Case: $\Theta(n \lg n)$
- Versteckte Konstanten in $\Theta(n \lg n)$ sehr klein
- Sortiert in-place

Idee

- Teile nach Elementgrößen

Vergleiche: Merge Sort teilt nach Lage (Mitte).

Teile-und-Herrsche-Schritte

Teile-und-Herrsche-Schritte um $A[p\,..\,r]$ zu sortieren:

1. **Teile**: $A[p\,..\,r]$ wird umgeordnet und aufgeteilt in (möglicherweise leere) Teilfolgen
   - $A[p\,..\,q-1]$,
   - $A[q+1\,..\,r]$,

   so dass die Elemente von $A[p\,..\,q-1]$ kleiner sind als $A[q]$, welches wiederum kleiner ist als die $A[q+1\,..\,r]$. Dazu wird eine Partitionierungsfunktion verwendet, welche auch Index q berechnet.
2. **Herrsche**: Sortiere Teilfolgen $A[p\,..\,q-1]$ und $A[q+1\,..\,r]$ rekursiv.
3. **Kombiniere**: Da in-place sortiert wird, ist in diesem Schritt keine Arbeit notwendig.

Quicksort (Pseudo-Code)

QUICKSORT(A, p, r)

1: **if** $p < r$
2: $\quad q \leftarrow$ PARTITION(A, p, r)
3: $\quad$ QUICKSORT$(A, p, q-1)$
4: $\quad$ QUICKSORT$(A, q+1, r)$

Ein Aufruf QUICKSORT$(A, 1, A.length)$ sortiert das ganze Array A.

Partition (Pseudo-Code)

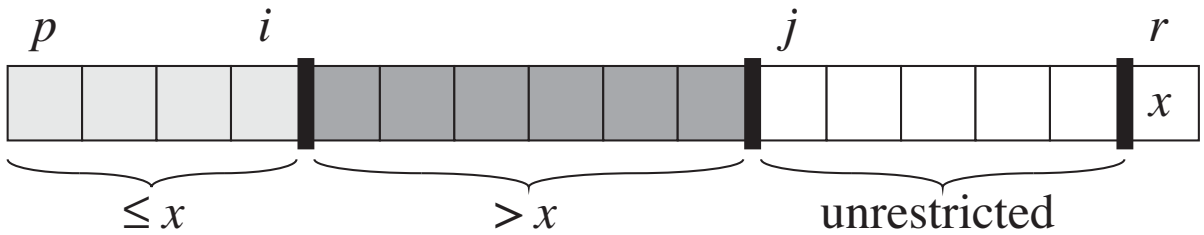
PARTITION(A, p, r)

1: $x \leftarrow A[r]$
2: $i \leftarrow p - 1$
3: **for** $j \leftarrow p$ **to** $r - 1$
4: **if** $A[j] \leq x$
5: $i \leftarrow i + 1$
6: Exchange $A[i]$ with $A[j]$
7: Exchange $A[i + 1]$ with $A[r]$
8: **return** $i + 1$

- Das letzte Element $A[r]$ in Teilarray $A[p \,..\, r]$ wird als **Pivotelement** gewählt

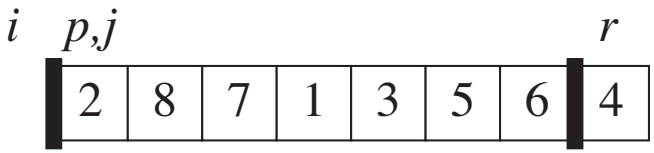
Schleifeninvariante

- Das letzte Element $A[r]$ wird als Pivot verwendet
- Schleifeninvariante
  1. Alle Werte in $A[p\,..\,i]$ sind $\leq$ Pivot
  2. Alle Werte in $A[i+1\,..\,j-1]$ sind $>$ Pivot
  3. $A[r] =$ Pivot

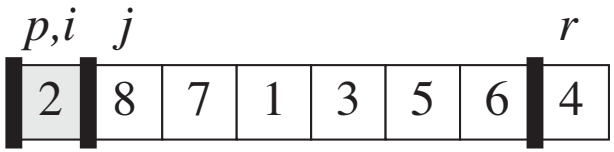


Beispiel

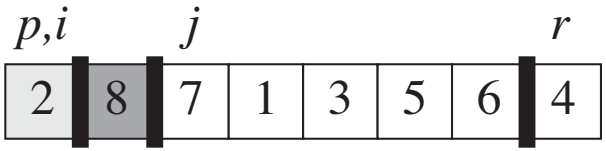
a)



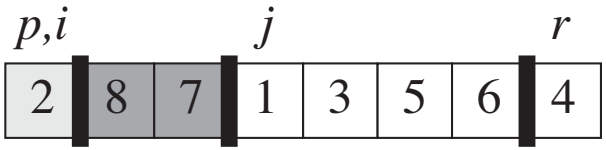
b)



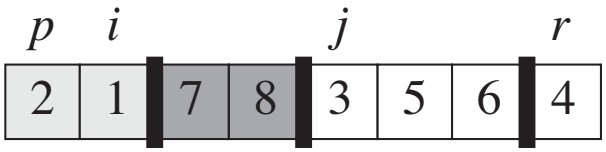
c)



d)



e)

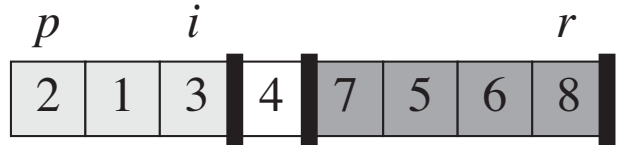


f)

g)

h)

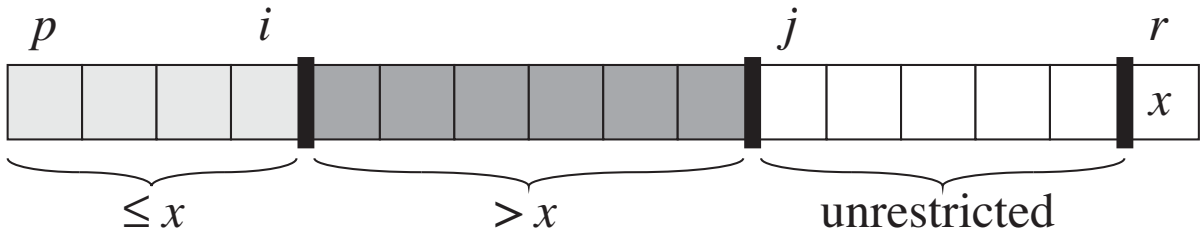
i)



Korrektheit der Schleifeninvariante (1)

Schleifeninvariante

1. Alle Werte in $A[p\,..\,i]$ sind $\leq$ Pivot
2. Alle Werte in $A[i+1\,..\,j-1]$ sind $>$ Pivot
3. $A[r] = x =$ Pivot



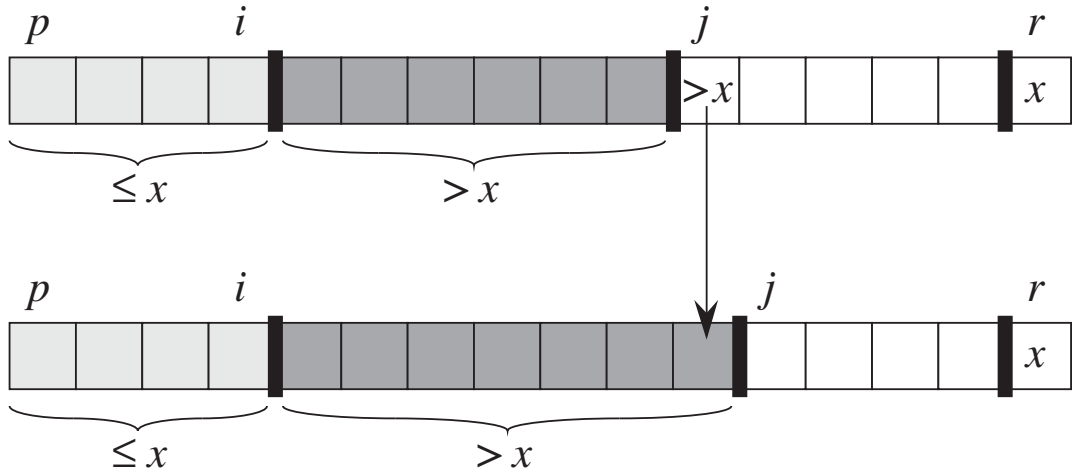
Induktionsanfang

- (1) hält, da $i = p - 1$ und somit $A[p\,..\,i]$ leer
- (2) hält, da $j = p$ und somit $A[i+1\,..\,j-1]$ leer
- (3) hält, da $x \leftarrow A[r]$ in Z. 1 und A unverändert

Korrektheit der Schleifeninvariante (2)

Induktionsschritt (1)

Fall 1: $A[j] \leq x$ in Z. 4 *nicht* erfüllt.



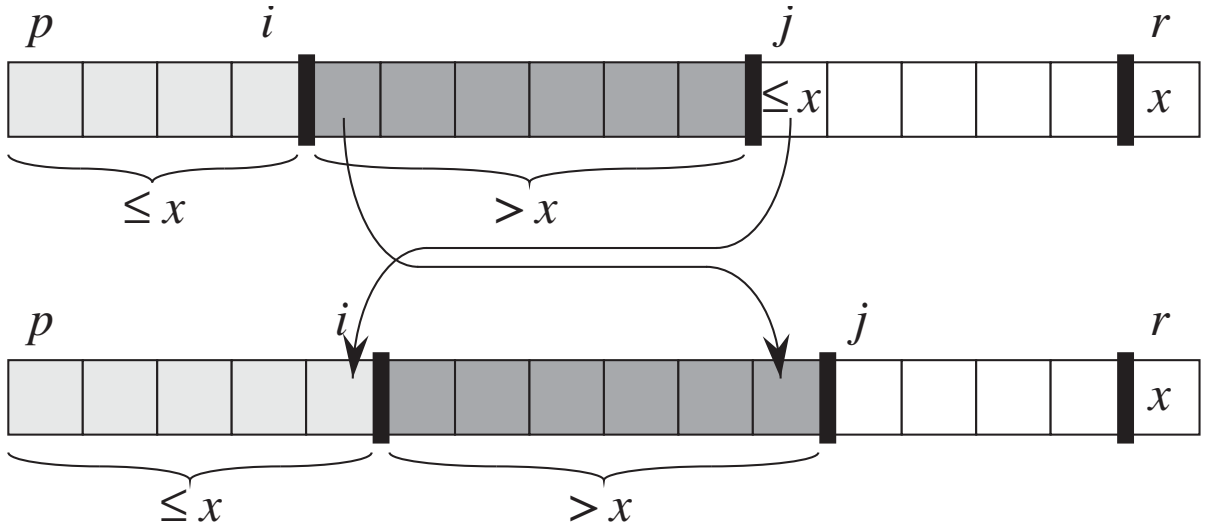
- Einzige Aktion: j inkrementieren
- (1) und (3) halten, da unverändert
- (2) hält, da vor Erhöhung von j
  - Alle Werte in $A[i+1\,..\,j-1]$ sind $\leq x$ nach Induktionsannahme
  - $A[j] > x$
  - Alle Werte in $A[i+1\,..\,j]$ sind $\leq$ Pivot

  und somit nach Erhöhung von j
  - Alle Werte in $A[i+1\,..\,j-1] \leq x$

Korrektheit der Schleifeninvariante (3)

Induktionsschritt (2)

Fall 2: $A[j] \leq x$ in Z. 4 erfüllt.



- i erhöhen: Jetzt ist $A[i] > x \rightarrow$ (1) hält nicht mehr
- $A[i]$ und $A[j]$ vertauschen
  - Jetzt ist $A[i] \leq x \rightarrow$ (1) hält wieder
  - Außerdem ist jetzt $A[j] > x$
- j erhöhen (Ende der Schleife): (3) hält

Korrektheit der Schleifeninvariante (4)

Schleifeninvariante

1. Alle Werte in $A[p\,..\,i]$ sind $\leq$ Pivot
2. Alle Werte in $A[i+1\,..\,j-1]$ sind $>$ Pivot
3. $A[r] = x =$ Pivot

Terminierung

Es gilt $j = r$ somit ist das gesamte Teilarray aufgeteilt in Bereiche

- $A[p\,..\,i]$ mit Werten $\leq x$,
- $A[i+1\,..\,r-1]$ mit Werten $> x$,
- $A[r]$ mit Wert $= x$.

Jetzt tauschen wir noch $A[i+1]$ und $A[r]$ und setzen somit das Pivotelement an seine korrekte Position.

Outline

1. Bäume

2. Heapsort

3. Quicksort
3.1 Algorithmus
3.2 Analyse
3.3 Randomisiertes Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Analyse von Partition

PARTITION(A, p, r)

1: $x \leftarrow A[r]$
2: $i \leftarrow p - 1$
3: **for** $j \leftarrow p$ **to** $r - 1$
4: **if** $A[j] \leq x$
5: $i \leftarrow i + 1$
6: Exchange $A[i]$ with $A[j]$
7: Exchange $A[i + 1]$ with $A[r]$
8: **return** $i + 1$

- Jede Zeile $\Theta(1)$
- Für Teilarraygröße $n = r - p + 1$, Gesamtlaufzeit $\Theta(n)$

Analyse von Quicksort

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

- Laufzeit abhängig von Balancierung, d.h. dem Verhältnis der Größen beider Partitionen
- Wenn balanciert: asymptotische Laufzeit $O(n \lg n)$
- Wenn unbalanciert: asymptotische Laufzeit $O(n^2)$

Quicksort: Worst Case

- Tritt auf wenn $n-1$ Werte in einem der Teilarrays, 0 in dem anderen
- Rekurrenz ist dann

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

(Via Substitutionsmethode)

- Somit ist der Worst Case von QUICKSORT nicht besser als der von INSERTION-SORT
  - Tritt auf falls A schon sortiert
  - Dann braucht INSERTION-SORT aber nur $\Theta(n)$

Quicksort: Best Case

- Tritt auf wenn $\lfloor n/2 \rfloor$ Werte in einem der Teilarrays, $\lceil n/2 \rceil - 1$ in dem anderen (beide $\leq n/2$)
- Rekurrenz ist dann

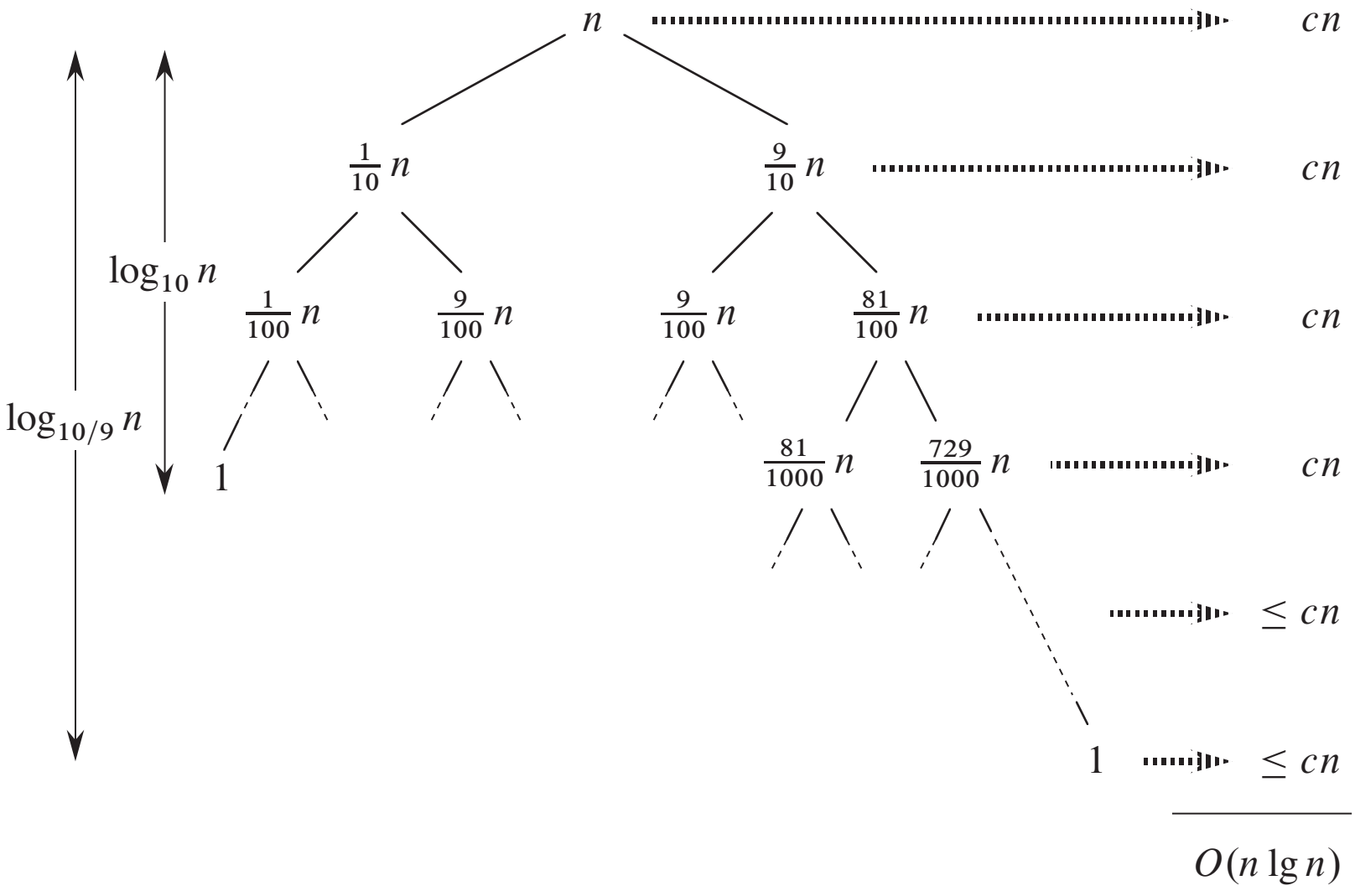
$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

  (Via Master Theorem)
- Jetzt ist QUICKSORT viel schneller
- Eine gute Partitionierung ist somit wichtig
- Wir werden sehen: Average Case ist nah am Best Case, nicht am Worst Case

Einfluss der Partitionierung (1)

Annahme: Partition partitioniert im Verhältnis 9:1.

$$T(n) = T(9n/10) + T(n/10) + n = O(n \lg n)$$

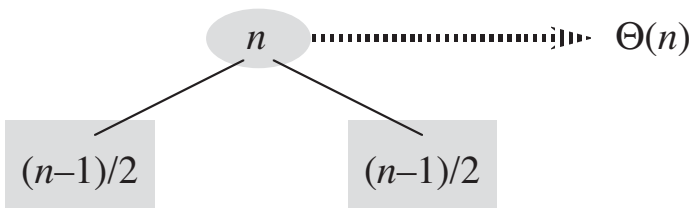


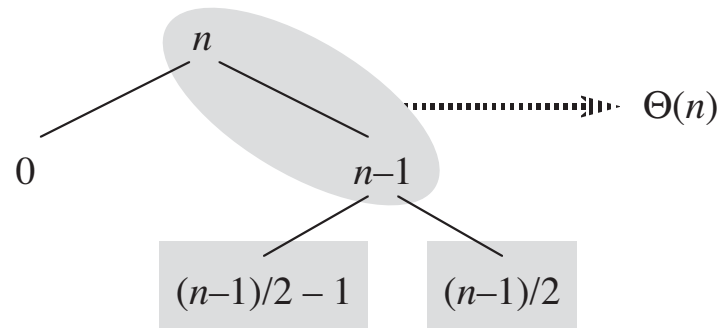
Einfluss der Partitionierung (2)

- $\log_{10} n$ volle Ebenen
- $\log_{10/9} n = \Theta(\lg n)$ Ebenen insgesamt
- Somit Laufzeit immer noch $\Theta(n \lg n)$
- Ähnlich
  - 99:1
  - 999:1
- Allgemein
  - Falls das Verhältnis der Partitionsgrößen durch eine Konstante < 1 beschränkt ist, so ist die Laufzeit von Quicksort $\Theta(n \lg n)$

Intuition Average Case (1)

- In einem typischen Lauf sind die Partitionsgrößen nicht immer durch Konstante beschränkt
- Es gibt "gute" und "schlechte" Partitionierungen
- Beispiel: jede zweite Partitionierung schlecht





→ Kosten der Extraebene durch Θ-Notation versteckt
→ Auch $\Theta(n \lg n)$ Gesamtlaufzeit

Intuition Average Case (2)

Wieviele schlechte Partitionierungen gibt es?

- Annahme: Eingabe (und somit Teilarray für PARTITION) ist Zufallspermutation
- Wir haben ein 9:1 Verhältnis mit Wahrscheinlichkeit $\approx 80\%$
  $\rightarrow$ Das ist viel besser als 50%!
- Somit benötigt Quicksort intuitiv $\Theta(n \log n)$ im Average Case
- Formales Argument folgt für randomisiertes Quicksort

Outline

1. Bäume

2. Heapsort

3. Quicksort
3.1 Algorithmus
3.2 Analyse
3.3 Randomisiertes Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Randomisiertes Quicksort

Oft als bester Algorithmus für große n angesehen.

RANDOMIZED-PARTITION(A, p, r)

1: $i \leftarrow$ RANDOM(p, r)
2: Exchange $A[r]$ with $A[i]$
3: **return** PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)

1: **if** $p < r$
2: $\quad q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)
3: $\quad$ RANDOMIZED-QUICKSORT$(A, p, q-1)$
4: $\quad$ RANDOMIZED-QUICKSORT$(A, q+1, r)$

- Pivotelement wird jetzt zufällig ausgewählt
- Worst Case und Best Case bleiben erhalten
  (d.h. aber nicht mehr durch bestimmte Eingaben ausgelöst)

Worst Case-Laufzeit (1)

- Wir haben schon argumentiert, dass der schlechteste Fall dann eintritt, wenn in jedem Teile-Schritte eine der Partitionen $n-1$ Element enthält
- Das analysieren wir jetzt genauer
- Sei $T(n)$ die Worst Case-Laufzeit und q die Größe der ersten Partition:

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

- Wir verwenden die Substitutionsmethode und raten

$$T(n) \leq cn^2$$

Worst Case-Laufzeit (2)

Damit:

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \\ &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

- 2. Ableitung von $q^2 + (n-q)^2$ nach q ist positiv (Übung)
- Somit liegt Maximum bei 0 oder $n-1$
- Hier: beide Werte gleich
- Somit

$$\begin{aligned} \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) &= 0^2 + (n-1)^2 \\ &= n^2 - 2n + 1 \end{aligned}$$

Worst Case-Laufzeit (3)

Also

$$\begin{aligned} T(n) &\leq cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

falls c so groß, dass $\Theta(n)$ durch $c(2n-1)$ dominiert wird.

RANDOMIZED-QUICKSORT hat somit eine Worst Case-Laufzeit von $O(n^2)$. Man kann auch zeigen: Rekurrenz ist $\Omega(n^2)$.

Theorem

RANDOMIZED-QUICKSORT *hat eine Worst Case-Laufzeit von* $\Theta(n^2)$.

Erwartete Laufzeit: Vorbetrachtung

- Hauptkosten enstehen durch Partitionierung
- PARTITION entfernt das Pivotelement aus der weiteren Betrachtung $\rightarrow$ Höchstens n-mal aufgerufen
- Arbeit in PARTITION ist eine Konstante plus die Anzahl der Vergleiche in Z. 4
- Sei X die Gesamtanzahl der Vergleiche in allen Aufrufen von PARTITION
- Gesamtlaufzeit ist dann $O(n + X)$
- Wir nehmen an, dass alle Elemente verschieden sind
  - Wie verhält sich Quicksort sonst? (Übung)

Unser Ziel ist es, die erwartete Anzahl der Vergleiche zu beschränken.

Erwartete Laufzeit (1)

- Um die Analyse einfacher zu machen, nennen wir die Elemente in A in $z_1, z_2, \ldots, z_n$ um, sodass $z_1 < z_2 < \cdots < z_n$
- Sei $Z_{ij} = \{ z_i, z_{i+1}, \ldots, z_j \}$
- Beobachtung: Jedes Paar von Elementen wird höchstens einmal verglichen
  - Da nur mit dem Pivotelement verglichen wird
  - Und dieses nach dem Aufruf von Partition nicht mehr betrachtet wird
- Sei $X_{ij} = I_{z_i \text{ wird mit } z_j \text{ verglichen}}$; dann

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

- Und somit

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}(z_i \text{ wird mit } z_j \text{ verglichen})$$

Erwartete Laufzeit (2)

- Elemente in verschiedenen Partitionen werden nicht miteinander verglichen
  - $\langle 8, 1, 6, 4, 0, 3, 9, 5 \rangle$
  - Pivot 5
  - Elemente aus $\{ 0, 1, 3, 4 \}$ werden nicht mit Elementen aus $\{ 6, 8, 9 \}$ verglichen
- Wenn ein Pivot x gewählt wird, sodass $z_i < x < z_j$, dann werden z_i und z_j danach nicht mehr miteinander verglichen
- Wenn z_i als allererstes Element aus Z_{ij} als Pivot ausgewählt wird, dann wird es mit allen Elementen in Z_{ij} vergleichen
- Somit: Wahrscheinlichkeit, dass z_i mit z_j verglichen wird, entspricht der Wahrscheinlichkeit dass entweder z_i oder z_j zuerst als Pivot (aus Z_{ij}) gewählt wird
- Z_{ij} enthält $j - i + 1$ Elemente, jedes davon wird als erster Pivot gewählt mit Wahrscheinlichkeit $1/(j - i + 1)$

Erwartete Laufzeit (3)

Wir erhalten:

$$
\begin{aligned}
&\mathbb{P}\left(z_i \text{ wird mit } z_j \text{ verglichen} \right) \\
&\quad = \mathbb{P}\left(z_i \text{ oder } z_j \text{ ist erster Pivot aus } Z_{ij} \right) \\
&\quad = \mathbb{P}\left(z_i \text{ ist erster Pivot aus } Z_{ij} \right) + \mathbb{P}\left(z_j \text{ ist erster Pivot aus } Z_{ij} \right) \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&\quad = \frac{2}{j-i+1}.
\end{aligned}
$$

Somit

$$
\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}
$$

Einschub: Approximation mit Integralen (1)

Theorem

Es sei $m, n \in \mathbb{N}$, $m < n$ und $f : [m, n] \to \mathbb{R}^+$. Wenn f monoton wachsend ist, dann

$$\int_{x=m-1}^{n} f(x)\, dx \leq \sum_{i=m}^{n} f(i) \leq \int_{x=m}^{n+1} f(x)\, dx.$$

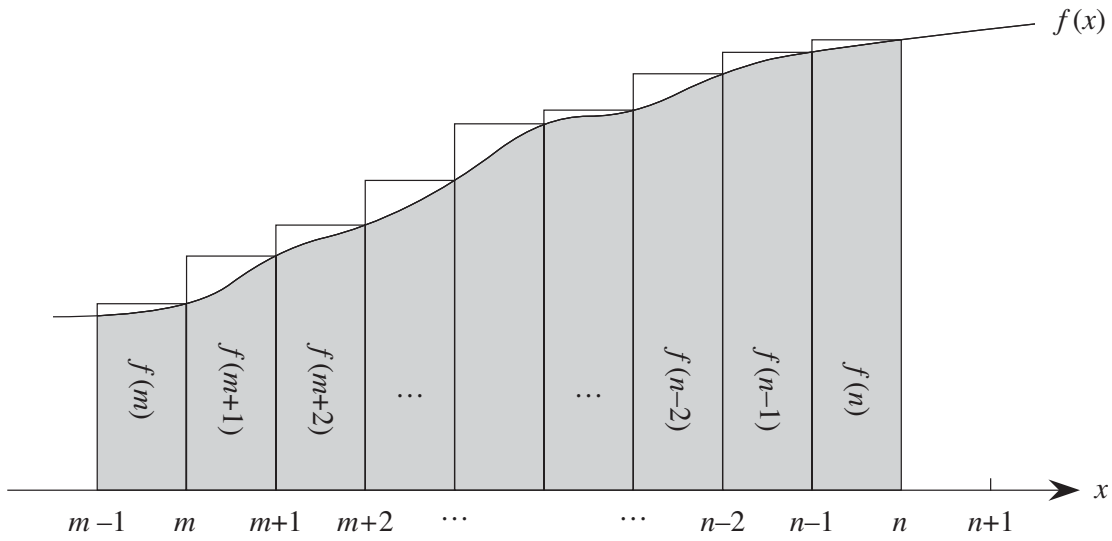
Wenn f monoton fallend ist, dann

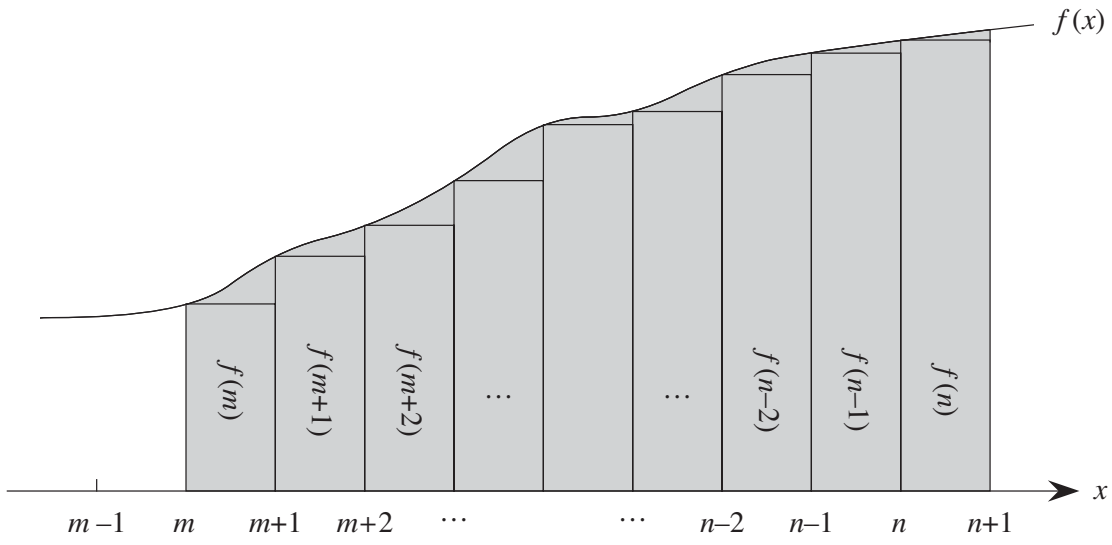
$$\int_{x=m}^{n+1} f(x)\, dx \leq \sum_{i=m}^{n} f(i) \leq \int_{x=m-1}^{n} f(x)\, dx.$$

Damit können wir Summen von unten und oben beschränken.

Einschub: Approximation mit Integralen (2)

Beweis (monoton wachsend).





Einschub: Approximation mit Integralen (3)

Beispiel: Harmonische Zahlen

- $H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i}$
- $f(i) = 1/i$, monoton fallend
- Stammfunktion $F(i) = \ln i$
- Somit gilt

$$F(x)\Big|_1^{n+1} \leq H_n \leq 1 + F(x)\Big|_1^n$$
$$\ln(n+1) \leq H_n \leq \ln n + 1$$

- Wir erhalten: $H_n = \Theta(\lg n)$

Einschub: Approximation mit Integralen (4)

Beispiel: Summe von Logarithmen

- $L_n = \sum_{i=1}^{n} \ln i = \sum_{i=2}^{n} \ln i$
- $f(i) = \ln i$, monoton steigend
- Stammfunktion $F(x) = x(\ln x - 1)$
- Somit gilt

$$F(x)\Big|_1^n \leq L_n \leq F(x)\Big|_2^{n+1}$$
$$n(\ln n - 1) + 1 \leq L_n \leq (n+1)(\ln(n+1) - 1) - 2(\ln 2 - 1)$$

- Wir erhalten: $L_n = \Theta(n \lg n)$

Erwartete Laufzeit (4)

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&= \sum_{k=1}^{n-1} \Theta(\lg i) \\
&= \Theta(n \lg n)
\end{aligned}
$$

Theorem

RANDOMIZED-QUICKSORT *hat eine erwartete Laufzeit von* $\Theta(n \lg n)$.

Zusammenfassung

Bis jetzt

- Mehrere Algorithmen in $\Theta(n \lg n)$
- Alle Algorithmen basierten auf Vergleichen

Jetzt

- Beweis einer unteren Schranke für vergleichbasierte Algorithmen
- Schnellere Algorithmen, die nicht auf Vergleichen basieren

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Untere Schranken für Sortieren

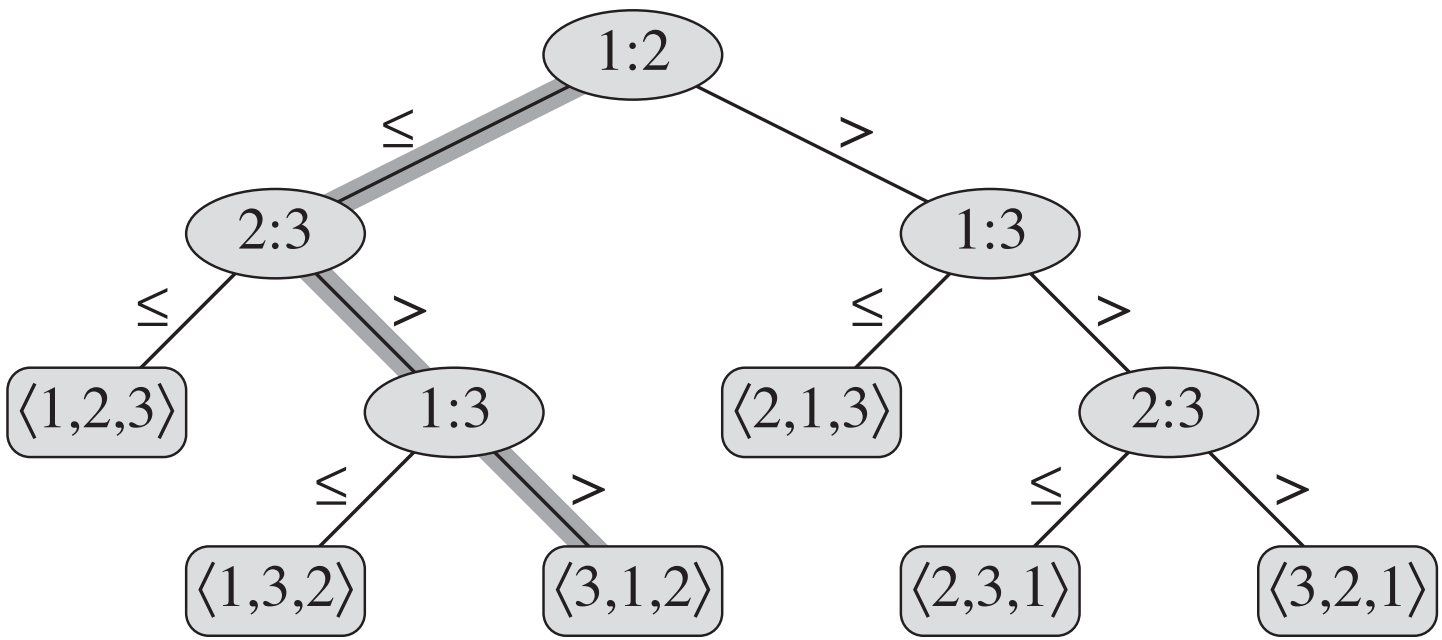
- Einfache untere Schranke: $\Omega(n)$
  - Denn: Eingabe lesen ist $\Omega(n)$
- Alle bisher betrachteten Algorithmen benötigen $\Omega(n \lg n)$
- Geht es besser? Gibt es einen besseren Algorithmus?
  - Dafür müssen wir die "Spielregeln" beschreiben
  - Welche Annahmen darf ein Algorithmus über die Eingabe treffen?
  - Welche Operationen dürfen auf den Elementen ausgeführt werden?
- Ein **vergleichsbasierter Algorithmus** darf nur Vergleiche verwenden ($<, \leq, =, \geq, >$), um Informationen über die Ordnung zu erhalten
- Wie nehmen oBdA. an, dass alle Elemente unterschiedlich sind
  - Dann $=$ nicht benötigt
  - Und $<, \leq, \geq, >$ liefern äquivalente Information

Entscheidungsbaum

- Ein **Entscheidungsbaum** ist eine Abstraktion der Arbeitweise eines vergleichsbasierten Algorithmus
  - Repräsentiert die Vergleiche, die ein gegebener Algorithmus für Eingaben einer gegebenen Größe ausführt
  - Alles andere wird vernachlässigt: Kontrollfluss, Datenbewegung
  - Wir zählen die Anzahl der Vergleiche
- Es existieren $n!$ Permutationen der Eingabefolge
- Nur eine ergibt sortierte Folge
- Diese wird durch den Sortieralgorithmus (implizit) ermittelt
- Aufbau des Entscheidungsbaum
  - Binärbaum
  - Innere Knoten repräsentieren durch Algorithmus durchgeführte Vergleiche
  - Sohnknoten repräsentieren Ergebnis des Vergleichs $\leq, >$
  - Blätter sind mit einer Permutation markiert

Entscheidungsbaum, Beispiel

- Für InsertionSort, $n = 3$
- $i : j$ bedeutet Vergleich der Elemente a_i und a_j (i und j beziehen sich auf die ursprüngliche Position der Elemente)
- Markiert: Ablauf für $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$



Untere Schranke für Worst Case (1)

Beobachtungen

- Jede Eingabe $\langle a_1, a_2, \ldots, a_n \rangle$ entspricht einem Pfad von Wurzel zu Blatt mit Markierung π so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$
- Für jede Permutation $\pi \in \mathcal{S}_n$ existiert eine Eingabe, die durch π sortiert wird
- Somit muss jede Permutation $\pi \in \mathcal{S}_n$ als ein Blatt vorkommen
- Wie hoch ist der Entscheidungsbaum dann mindestens?

Untere Schranke für Worst Case (2)

Theorem

Ein Entscheidungsbaum, der n Elemente sortiert, hat Höhe $\Omega(n \lg n)$.

Beweis. Wir wissen: Ein binärer Baum der Höhe h hat $\leq 2^h$ Blätter. Es gibt $n!$ Permutationen, jede muss mindestens einmal als Blatt auftreten. Daher gilt:

$$2^h \geq n!$$

und somit

$$h \geq \ln(n!) = \sum_{i=1}^{n} \ln i = \Omega(n \lg n)$$

nach Folie 95 oder **Stirlings Formel**

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n))\,.$$

□

Optimalität von Merge Sort und Heap Sort

Korollar

MERGE-SORT *und* HEAP-SORT *sind asymptotisch optimale Sortieralgorithmen.*

Beweis. Die $O(n \lg n)$-Schranke der Worst Case-Laufzeit beider Algorithmen entspricht der unteren Schranke für die Worst Case-Laufzeit von vergleichsbasierten Sortieralgorithmen. □

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit
5.1 Counting Sort
5.2 Radix Sort
5.3 Bucket Sort

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit
5.1 Counting Sort
5.2 Radix Sort
5.3 Bucket Sort

Counting Sort

Annahme

Eingabeelemente sind natürliche Zahlen im Bereich $[0, k]$

Grundidee

- Für jedes $i \in [0, k]$ bestimme die Anzahl der Elemente $\leq i$
- Damit kann man die Position von i in der sortierten Folge bestimmen
- Falls 17 Elemente $\leq i$, dann muß i an die 18. Stelle (falls keine Duplikate)

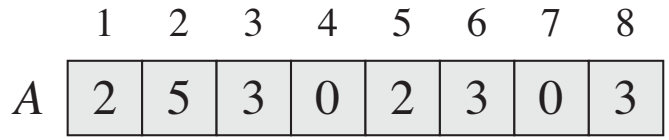
Counting Sort (Pseudo-Code)

COUNTING-SORT(A, B, k)

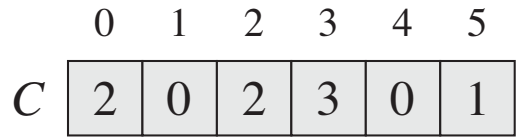
```
1:  Let C[0..k] be a new array
2:  for i ← 0 to k
3:      C[i] ← 0
4:  for j ← 1 to A.length
5:      C[A[j]] ← C[A[j]] + 1
6:  // C[i] now contains the number of elements = i
7:  for i ← 1 to k
8:      C[i] ← C[i] + C[i − 1]
9:  // C[i] now contains the number of elements ≤ i
10: for j ← A.length downto 1
11:     B[C[A[j]]] ← A[j]
12:     C[A[j]] ← C[A[j]] − 1
```

- A ist zu sortierende Eingabe
- $B[1..n]$ wird mit sortierter Ausgabe beschrieben
- $C[0..k]$ für Zwischenergebnisse

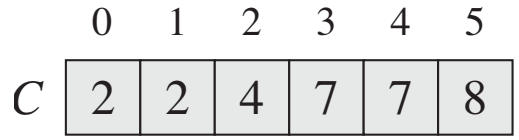
Beispiel



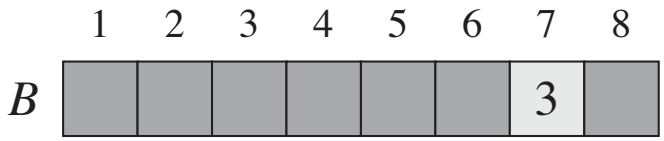
Z. 6:

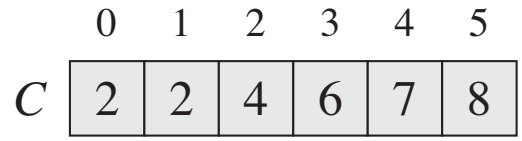


Z. 9:

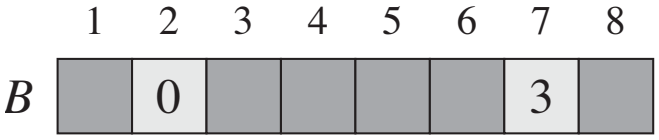


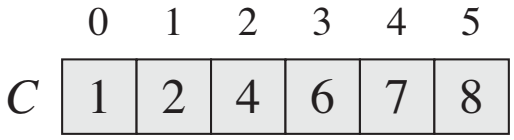
1x Z. 10-12:



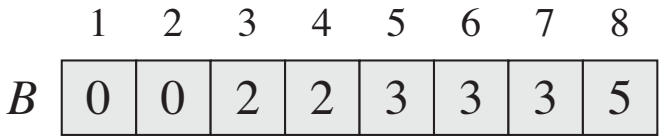


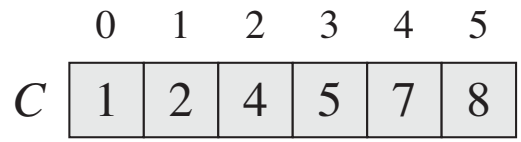
2x Z. 10-12:



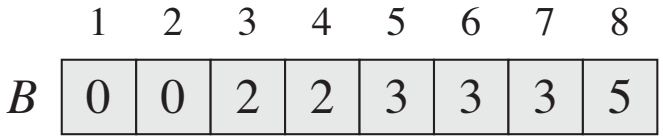


3x Z. 10-12:





Endergebnis:



Analyse

Laufzeit

- Z. 1–3: $\Theta(k)$
- Z. 4–6; $\Theta(n)$
- Z. 7–9: $\Theta(k)$
- Z. 10–12: $\Theta(n)$
- Also gesamt: $\Theta(n + k)$

Speicherbedarf

- $\Theta(k)$ für C
- Nicht in-place (aber hier: B ist Eingabe)

Bemerkungen

- Wenn $k = O(n)$, dann Gesamtlaufzeit $O(n)$
- Wird verwendet wenn k klein; in Praxis
  - Gut geeignet für 4-Bit-Zahlen ($k = 15$), wenn n nicht sehr klein
  - Nicht geeignet für 32-Bit-Zahlen
  - Dazwischen: je nachdem
- Counting-Sort ist nicht vergleichsbasiert (es gibt keinen einzigen Vergleich)
- Die $\Omega(n \lg n)$-Schranke gilt somit nicht
- Für $k = O(n)$ ist Counting-Sort asymptotisch besser als jeder vergleichsbasierte Algorithmus
- Counting-Sort ist **stabil** (*stable*)
  - Datensätze mit gleichem Schlüssel haben im Ergebnis die gleiche Reihenfolge wie in der Eingabe
  - Counting-Sort und die Stabilitätseigenschaft werden in Radix Sort verwendet

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit
5.1 Counting Sort
5.2 Radix Sort
5.3 Bucket Sort

Radix Sort

Ursprung

- Lochkarten sortieren
- Automatische Sortierung nur nach einer Stelle unterstützt
- Wie mehrstellige Zahlen sortieren?

Grundidee

- Sortiere stabil nach am wenigsten signifikanter Stelle
- Dann nächst signifikantere Stelle
- ...

Radix Sort (Pseudo-Code)

Sortiere Folge von d-stelligen Zahlen $(z_{d-1} z_{d-2} \cdots z_0)$.

RADIX-SORT(A, d)

1: **for** $i \leftarrow 0$ **to** $d - 1$
2: Use a stable sort to sort A on digit i

Beispiel

Eingabe		Stelle 0		Stelle 1		Stelle 2
326		690		704		326
453		751		608		435
608		453		326		453
835	$\rightarrow$	704	$\rightarrow$	835	$\rightarrow$	608
751		835		435		690
435		435		751		704
704		326		453		751
690		608		690		835

Korrektheit

- Per Induktion über d
- **Induktionsanfang**: $d = 1$
  - Nur eine Stelle; korrekt da nach dieser Stelle sortiert wird
- **Induktionsvoraussetzung**
  - Stellen $d - 2, \ldots, 0$ korrekt sortiert
- **Induktionsschritt**
  - Wir wenden ein stabiles Sortierverfahren auf Ziffer $d - 1$ an
  - Zwei Zahlen mit verschiedener Ziffer an Stelle $d - 1$ sind danach korrekt sortiert
  - Zwei Zahlen mit gleicher Ziffer an Stelle $d - 1$ sind nach Induktionsvoraussetzung bereits in richtiger Reihenfolge; sie bleiben es auch, da stabil sortiert wird
  - Somit Stellen $d - 1, \ldots, 0$ korrekt sortiert

Analyse

- Laufzeit abhängig vom benutzen Sortierverfahren
- Falls Stellen in $[0, k]$ und k nicht zu groß, dann
  - Counting-Sort gute Wahl
  - Laufzeit $\Theta(d(n + k))$
  - Wenn d konstant und $k = O(n)$, dann hat Radix-Sort linearen Aufwand
- Wie schnell kann man damit b-Bit-Worte sortieren?
  - Wir teilen die Worte in r-Bit-Ziffern auf
  - Dann $d = \lceil b/r \rceil$, $k = 2^r - 1$ und somit Laufzeit $\Theta(\frac{b}{r}(n + 2^r))$
  - Wenn $b < \lfloor \lg n \rfloor$, setzen wir $r = b$ und erhalten $\Theta(n)$
  - Wenn $b \geq \lfloor \lg n \rfloor$, setzen wir $r = \lfloor \lg n \rfloor$ und erhalten $\Theta(bn/\lg n)$

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit
5.1 Counting Sort
5.2 Radix Sort
5.3 Bucket Sort

Bucket Sort

Annahme

- Eingabe sind unabhängige, gleichverteilte Zufallszahlen in $[0, 1)$

Idee

- Aufteilen der Zahlen auf n gleichgroße **Buckets** (Körbe)
- z.B. für $n = 10$: $[0, 0.1)$, $[0.1, 0.2)$, $\ldots$, $[0.9, 1)$
- Dann Buckets sortieren

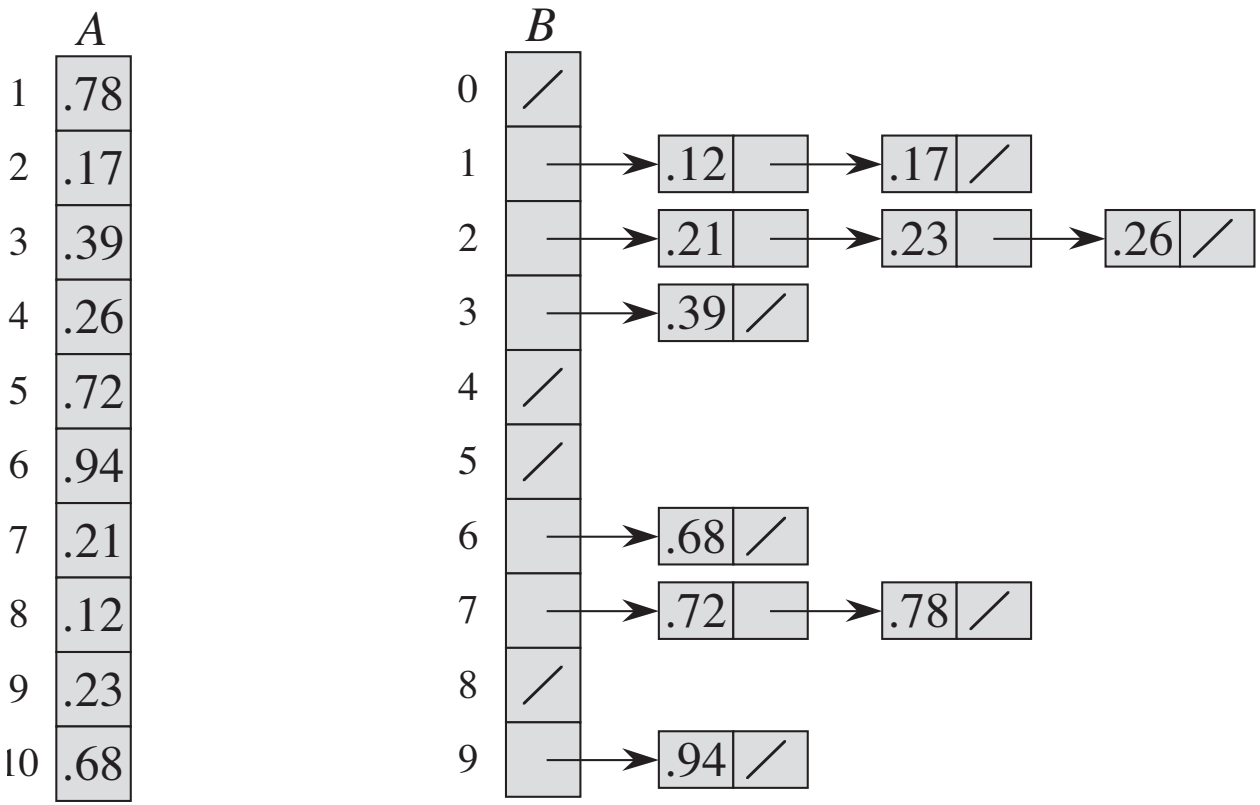
Bucket Sort (Pseudo-Code)

BUCKET-SORT(A)

1: $n \leftarrow A.length$
2: Let $B[0\,..\,n-1]$ be a new array
3: **for** $i \leftarrow 0$ **to** $n-1$
4: Make $B[i]$ an empty list
5: **for** $i \leftarrow 1$ **to** n
6: Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7: **for** $i \leftarrow 0$ **to** $n-1$
8: Sort list $B[i]$ with insertion sort
9: Concatenate the lists $B[0]$, $B[1]$, ..., $B[n-1]$ together in order

- Bucket i erhält alle Zahlen x mit $\lfloor nx \rfloor = i$
- D.h. im Bereich $\left[\frac{i}{n}, \frac{i+1}{n}\right]$

Beispiel


A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94

Analyse

- Jede Zeile außer Z. 8 insgesamt $O(n)$ im Worst Case
- Zeile 8: $O(n_i^2)$ falls Länge von Bucket i gleich n_i
  - Wir erwarten 1 Zahl in jedem Bucket, d.h. $\mathbb{E}[n_i] = 1$
  - Aber das reicht nicht, da $\mathbb{E}\left[n_i^2\right] \neq \mathbb{E}[n_i]$
  - Man kann zeigen: $\mathbb{E}\left[n_i^2\right] = 2 - 1/n = O(1)$
  - Erwartete Laufzeit für alle Ausführungen von Z. 8 ist somit $O(n)$
- Erwartete Laufzeit unter unseren Annahmen ist somit $O(n)$
- Wiederum: Keine Vergleiche während Aufteilung in Buckets
- Bemerkung: Ähnlicher Trick oft anwendbar wenn Verteilung der zu sortierenden Schlüssel bekannt ist

Outline

1. Bäume

2. Heapsort

3. Quicksort

4. Untere Schranke

5. Sortieren in Linearzeit

Zum Nachlesen

- CLRS, Ch. 6, *Heapsort*
- CLRS, Ch. 7, *Quicksort*
- CLRS, Ch. 8, *Sorting in Linear Time*
- Timo Bingmann, The Sound of Sorting

Algorithmen und Datenstrukturen

03 – Datenstrukturen

Prof. Dr. Rainer Gemulla

Universität Mannheim

HWS 2014

Dynamische Mengen

Wir müssen in Algorithmen oft **dynamische Mengen** verwalten, d.h. Mengen, die sich mit der Zeit ändern.

Operationen

- Einfügen
- Löschen
- Mitgliedschaft testen
- Weitere je nach Anwendung

Parameter

- Mengenzugehörigkeit (oft: Schlüsselattribut)
- Ordnung der Elemente
- Mehrfachmengen

Operationen (1)

- Wir arbeiten mit Zeigern auf die Elemente einer Menge S
- Jedes Element besitzt
  - Ein Schlüsselattribut names *key*
  - Beliebige Attribute für Satellitendaten
  - Ggf. weitere Attribute für die Implementierung der Mengendatenstruktur
- Allokation von Speicherplatz für die Elemente nicht betrachtet

Dictionary

Eine Datenstruktur, die die folgenden Operationen implementiert, nennt man **Dictionary**.

- $\textsc{Search}(S, k)$ gibt Zeiger x auf ein Element in S zurück so dass $x.key = k$; falls nicht vorhanden, $\textsc{nil}$
- $\textsc{Insert}(S, x)$ fügt Element, auf das x zeigt, in S ein
- $\textsc{Delete}(S, x)$ entfernt Element, auf das x zeigt, aus S

Operationen (2)

Operationen für total geordnete Schlüsselmengen

- $\textsc{Minimum}(S)$ gibt Zeiger auf minimales Element zurück
- $\textsc{Maximum}(S)$ analog
- $\textsc{Successor}(S, x)$ gibt Zeiger auf Nachfolgeelement von x zurück (oder $\textsc{nil}$) falls letztes
- $\textsc{Predecessor}(S, x)$ analog

Bei Mehrfachmengen: $\textsc{Minimum}$ plus $(|S| - 1)$-mal $\textsc{Successor}$ enumeriert alle Elemente (d.h. Elemente mit gleichem Schlüssel werden zuerst durchlaufen).

Kosten

- Gemessen in $|S|$

Outline

1. Elementare Datenstrukturen
2. Hash-Tabellen
3. Binäre Suchbäume
4. Rot-Schwarz-Bäume
5. Anreicherung von Datenstrukturen
6. Disjunkte Mengen

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

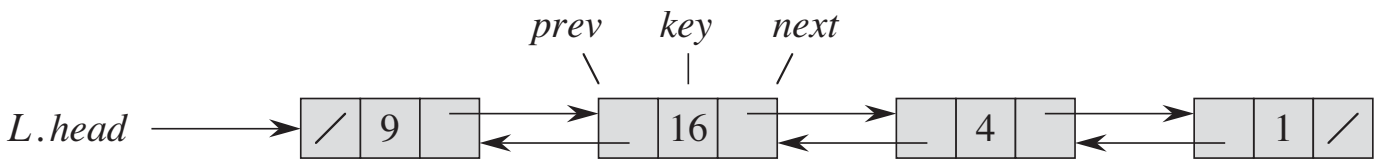
4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

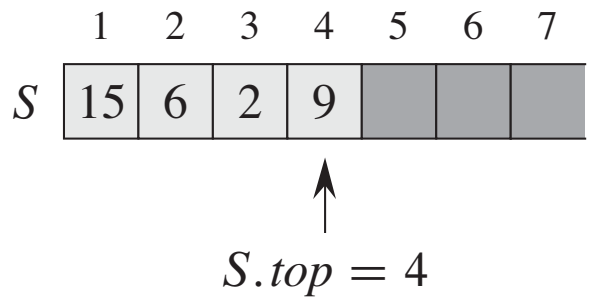
Verkettete Liste

- Reihenfolge durch zeigerbasierte Verkettung
- Einfach verkettet (Elementattribut: *next*)
- Oder mehrfach verkettet (Elementattribute: *prev*, *next*)
- Operationen und Kosten
  - LIST-SEARCH(L, k) in Worst Case-Laufzeit $\Theta(n)$
  - LIST-INSERT(L, x) in $O(1)$
  - LIST-DELETE(L, x) in $O(1)$ für doppelt verkettete Listen



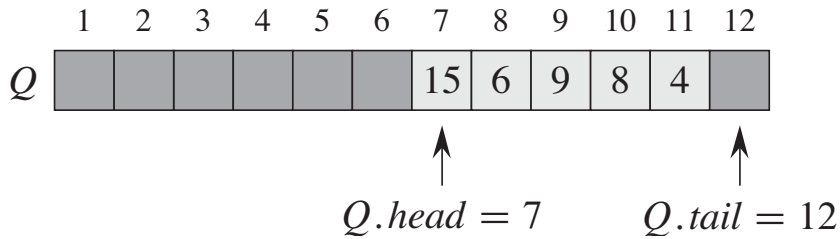
Stapel (stack)

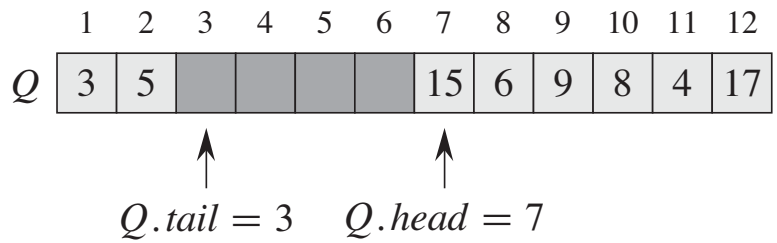
- Last-in-first-out (LIFO) queue
- Operationen und typische Kosten
  - $\textsc{Push}(S, x)$ in $O(1)$
  - $\textsc{Pop}(S)$ in $O(1)$
  - $\textsc{Stack-Empty}(S)$ in $O(1)$
- Beispielumsetzungen
  - Array (Nachteil: begrenzte Größe)
  - Verkettete Liste (Nachteil: Speicherplatzaufwand für Zeiger)



Schlange (queue)

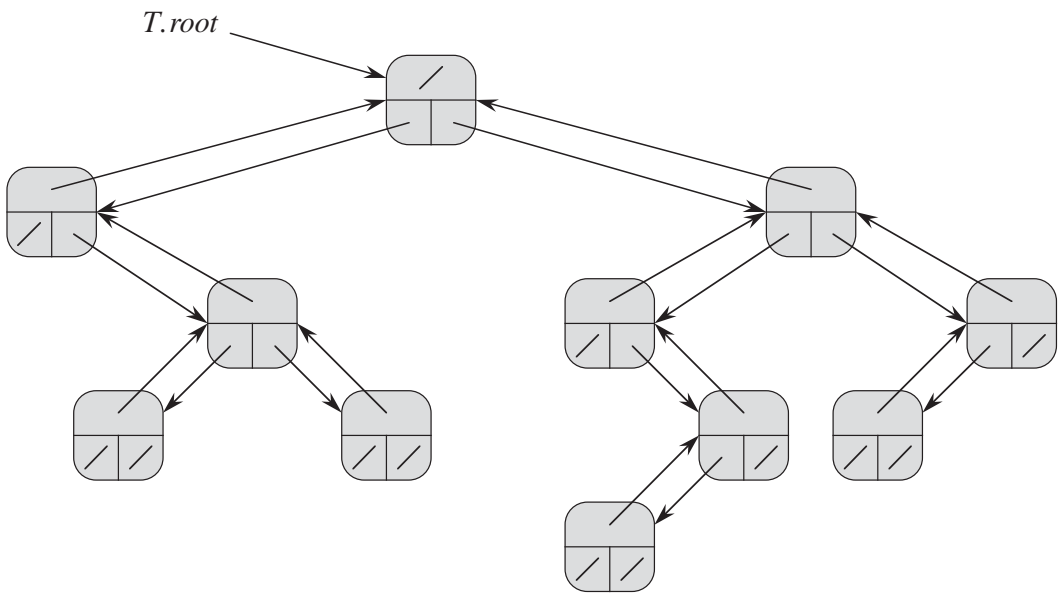
- First-in-first-out (FIFO) queue
- Operationen und typische Kosten
  - $\textsc{Enqueue}(Q, x)$ in $O(1)$
  - $\textsc{Dequeue}(Q)$ in $O(1)$
- Beispielumsetzungen
  - Als **Ringpuffer** organisiertes Array
  - Als verkettete Liste (mit zusätzlichem Zeiger *L.tail*)





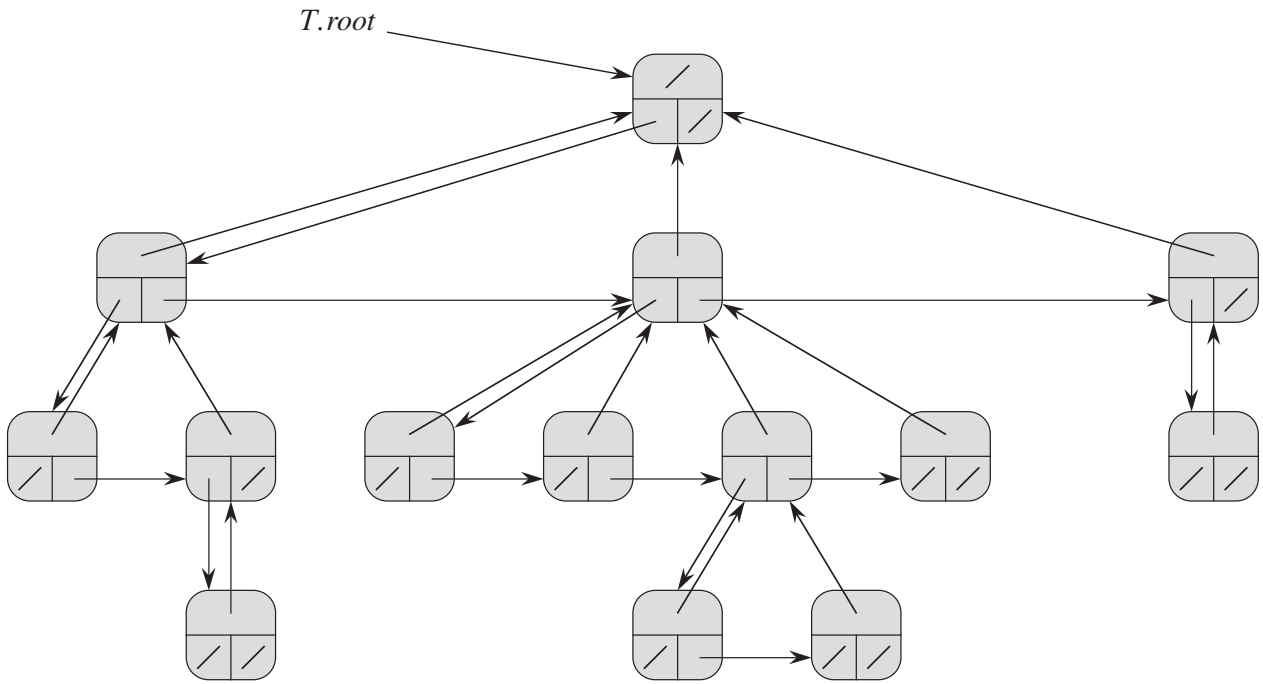
Binärbäume

- Elementattribute
  - p für Vaterknoten
  - *left* für linken Sohn
  - *right* für rechten Sohn
- $T.root = \text{NIL} \Leftrightarrow$ Baum ist leer
- $T.root = x \Leftrightarrow x$ ist Wurzel $\Leftrightarrow x.p = \text{NIL}$
- Dieses Schema anwendbar, solange Knotengrad beschränkt ist



Bäume mit unbegrenztem Grad

- **Left-child, right-sibling Repräsentation**
- Attribute
  - p für Vaterknoten
  - $x.\mathit{left\text{-}child}$ für das erste Kind
  - $x.\mathit{right\text{-}sibling}$ für den nächsten Geschwisterknoten
- $x.\mathit{left\text{-}child} = \text{NIL} \Leftrightarrow$ Blatt
- $x.\mathit{right\text{-}sibling} = \text{NIL} \Leftrightarrow x$ ist letzter Sohn (am weitesten rechts)



Andere Repräsentationen für Bäume

- Heap
- Einige Zeiger können wegfallen
- Viele Möglichkeiten, durch Anwendung bestimmt

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen
2.1 Direkte Adressierung
2.2 Chaining
2.3 Hash-Funktionen
2.4 Offene Adressierung

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Dictionaries und Hash-Tabellen

- Viele Anwendungen benötigen dynamische Mengen, die nur die Dictionary-Operationen unterstützen
  - $\textsc{Search}(S, k)$ gibt Zeiger x auf ein Element in S zurück so dass $x.key = k$; falls nicht vorhanden, $\textsc{nil}$
  - $\textsc{Insert}(S, x)$ fügt Element, auf das x zeigt, in S ein
  - $\textsc{Delete}(S, x)$ entfernt Element, auf das x zeigt, aus S
- Beispiel: Symboltabelle in einem Compiler
- Mit Dictionaries kann man auch Abbildung Schlüssel $\rightarrow$ Wert realisieren (key-Attribut = Schlüssel, Satellitendaten = Wert)
- Hash-Tabellen erlauben eine effektive Umsetzungen eines Dictionaries
  - Einfügen und Löschen in $O(1)$
  - Suche in $O(1)$ Average Case / erwartete Zeit unter bestimmten Annahmen, aber $\Theta(n)$ im Worst Case

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen
2.1 Direkte Adressierung
2.2 Chaining
2.3 Hash-Funktionen
2.4 Offene Adressierung

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Direkte Adressierung

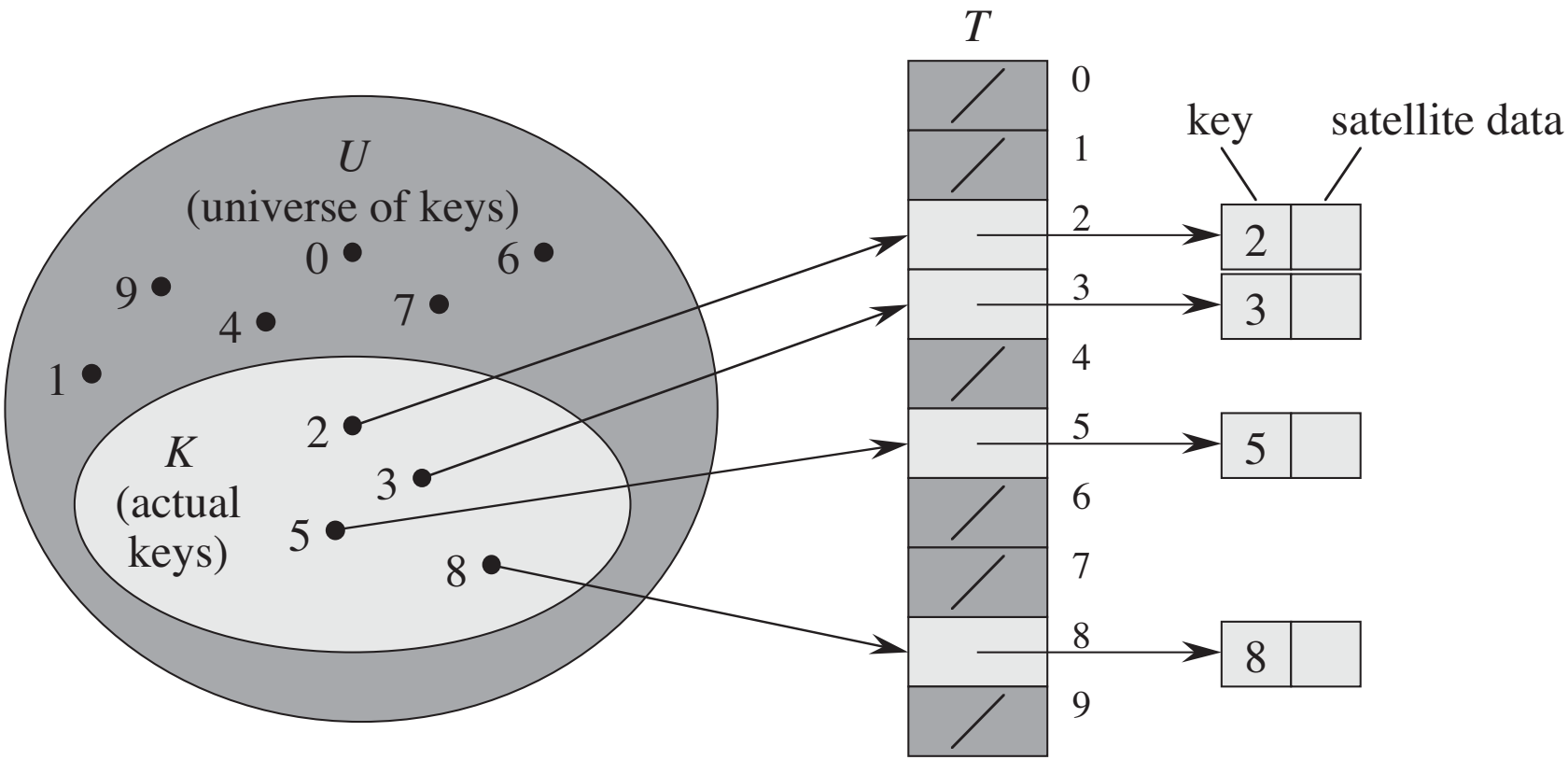
Annahme

- Alle Schlüssel aus $U = \{0, 1, \ldots, m-1\}$ für kleines m
- Alle Schlüssel sind verschieden

Eine Tabelle mit **direkter Adressierung** (*direct-address table*) ist ein Array $T[0..m-1]$.

- Idee: Schlüssel als Index
- Jeder Eintrag, genannt **Slot**, entspricht einem Schlüssel
- Wenn ein Element x mit Schlüssel k existiert, dann enthält $T[k]$ Zeiger auf x
- Ansonsten $T[k] = \text{NIL}$

Beispiel für direkte Adressierung



Direkte Adressierung (Pseudo-Code)

DIRECT-ADDRESS-SEARCH(T, k)

1: **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1: **return** $T[x.key] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)

1: **return** $T[x.key] \leftarrow$ NIL

- Alle Operationen $O(1)$

Hash-Tabellen

Probleme mit direkter Adressierung

- Großes U, aber nicht dicht besetzt
- Speicheraufwand ist $\Theta(|U|)$ selbst wenn nur Schlüssel aus $K \subseteq U$ mit $|K| \ll |U|$ tatsächlich gespeichert werden

Hash-Tabellen

- O(1) Zeit, jetzt allerdings Average Case / erwartete Laufzeit
- $\Theta(|K|)$ Speicher, also gut wenn $|K| \ll |U|$

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen
2.1 Direkte Adressierung
2.2 Chaining
2.3 Hash-Funktionen
2.4 Offene Adressierung

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Hash-Funktion

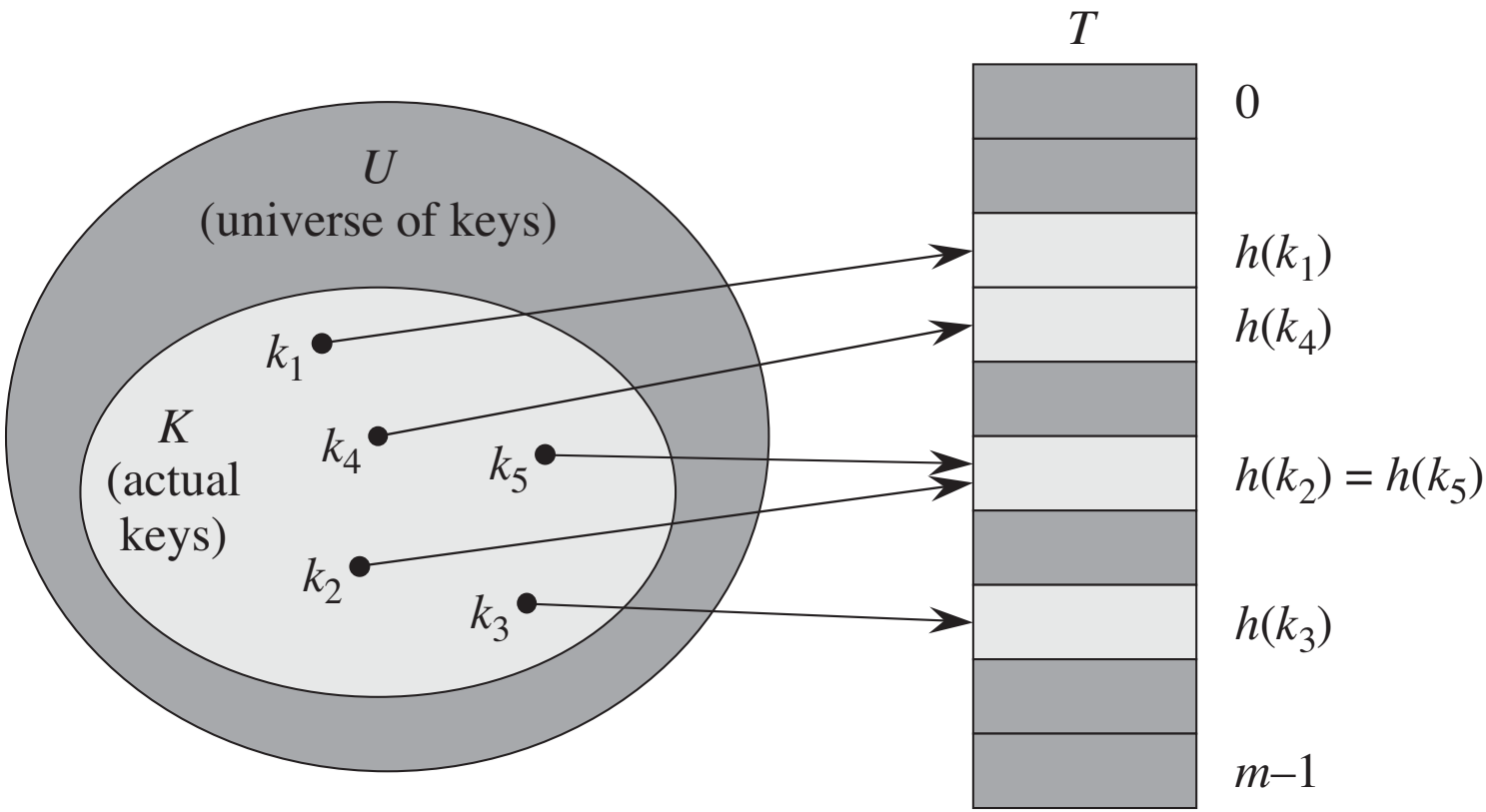
Bei direkter Adressierung wurde ein Element k an Adresse k gespeichert. Hash-Tabellen verallgemeinern dies:

- Speichere k an Adresse $h(k)$
- h heisst **Hash-Funktion**
- $h : U \to \{0, \ldots, m-1\}$
- $h(k)$ heisst **Hash-Wert** (*hash value*) von k

Kollision

- $h(k_1) = h(k_2)$ heißt **Kollision**
- h sollte so gestaltet sein, dass Kollisionen vermieden werden
  - Oft: Möglichst zufällig $\to$ Kollisionen unwahrscheinlich
- Kollisionen treten trotzdem auf, da normalerweise $|U| > m$
- Daher wichtig: **Kollisionsbehandlung**

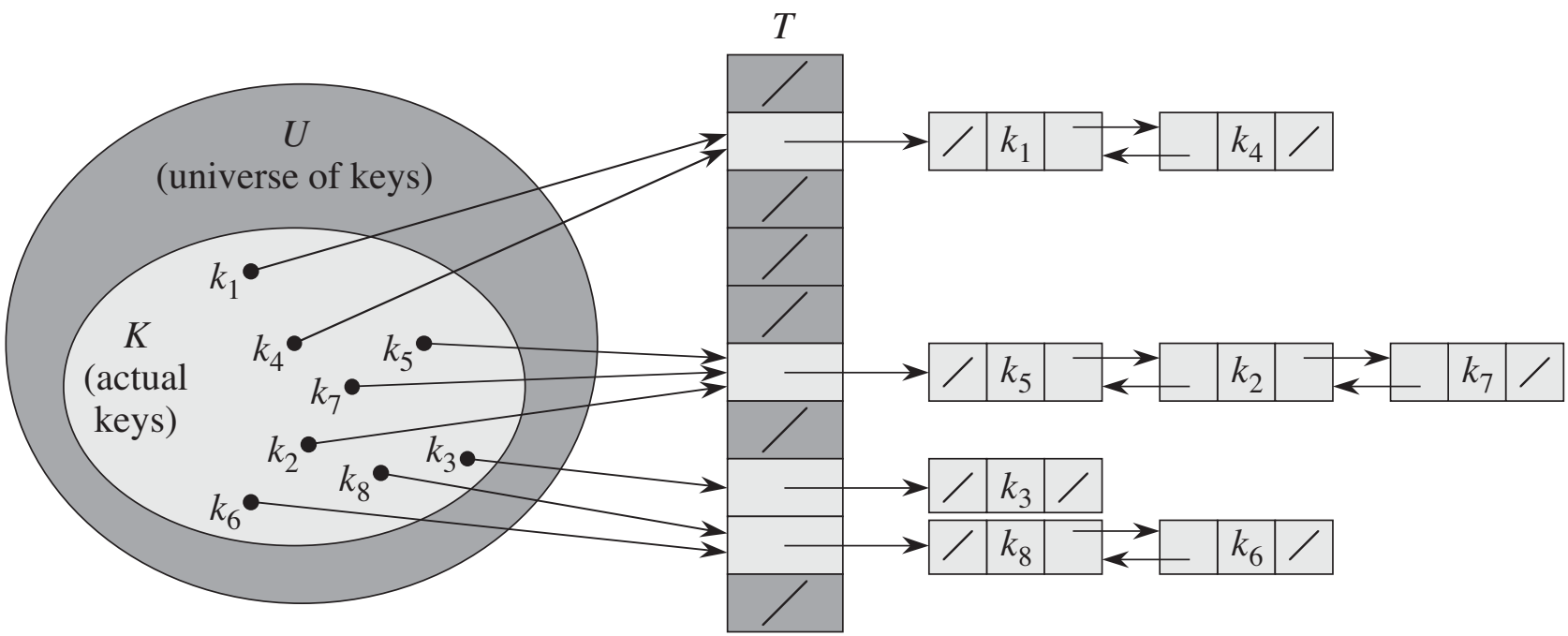
Beispiel mit Hash-Funktion



Kollisionsbehandlung durch Listen

Idee

- Jeder Slot ist eine verkettete Liste, die alle kollidierenden Elemente enthält
- Das nennt man **Chaining**



Chaining (Pseudo-Code)

CHAINED-ADDRESS-INSERT(T, x)

1: Insert x at the head of list $T[h(x.key)]$

CHAINED-ADDRESS-SEARCH(T, k)

1: Search for an element with key k in $T[h(k)]$

CHAINED-ADDRESS-DELETE(T, x)

1: Delete x from the list $T[h(x.key)]$

Analyse

- CHAINED-ADDRESS-INSERT ist $O(1)$
- CHAINED-ADDRESS-DELETE ist $O(1)$
  - Da Eingabe nicht Schlüssel, sondern Zeiger auf x
  - Wenn nur Schlüssel gegeben, dann muss man erst suchen
- Laufzeit CHAINED-ADDRESS-SEARCH ist proportional zur Länge der Liste; diese schauen wir uns jetzt an

Analyse

- Gesucht: Durchschnittliche Anzahl der betrachteten Elemente
- Analyse basiert auf **Füllgrad** α (*load factor*)
  - $n =$ Anzahl der Elemente in der Hash-Tabelle
  - $m =$ Anzahl der Slots $=$ Anzahl der verketteten Listen
  - $\alpha = n/m$ ist durchschnittliche Anzahl der Elemente pro Liste
  - Möglich: $\alpha < 1$, $\alpha = 1$, $\alpha > 1$
- Worst Case: Alle Elemente in einer Liste der Länge n
  $\rightarrow \Theta(n)$ Aufwand für Suche
- Was ist der Average Case? Annahmen:
  - h verteilt Schlüssel gleichmäßig auf alle Slots
  - Genauer: **simple uniform hashing**, d.h. jedes Element wird mit gleicher Wahrscheinlichkeit und unabhängig von anderen Elementen auf jeden Slot abgebildet
  - h berechenbar in $\Theta(1)$
- Kosten werden verursacht durch Ablaufen der Kollisionslisten
  - Fall 1: Suche nicht erfolgreich
  - Fall 2: Suche erfolgreich

Erfolglose Suche

Theorem

Eine erfolglose Suche hat Average Case-Laufzeit $\Theta(1+\alpha)$.

Beweis.

- Unter der Gleichverteilungsannahme: Jedes k wird auf einen der m Slots mit gleicher Wahrscheinlichkeit gehasht
- Average Case-Laufzeit entspricht dem durchschnittlichen Aufwand zum vollständigen Durchlaufen einer Kollisionsliste
- Erwartete Länge der durchsuchten Liste $= \alpha = n/m$
- Somit n/m Elemente durchschnittlich durchlaufen
- Average Case (mit Berechnung von $h(k)$) ist $\Theta(1+\alpha)$ □

Erfolgreiche Suche (1)

Weitere Annahme: Jedes Element wird mit gleicher Wahrscheinlichkeit gesucht.

Theorem

Eine erfolgreiche Suche hat Average Case-Laufzeit $\Theta(1 + \alpha)$.

Beweis.

- Anzahl betrachteter Elemente ist 1 + Anzahl aller Elemente vor dem gesuchten Element x
- Das sind alle die Elemente, die nach x eingefügt wurden und auf den gleichen Slot hashen
- Wie viele sind das für ein zufällig gewähltes x?

Erfolgreiche Suche (2)

- $x_i = i$-tes eingefügtes Element
- $k_i = x_i.key$
- Indikatorvariable $X_{ij} = I_{h(k_i)=h(k_j)}$
- Unabhängig gleichverteilt: $\mathbb{P}\left(h(k_i) = h(k_j) \right) = 1/m$, somit $\mathbb{E}\left[X_{ij} \right] = 1/m$
- Anzahl Elemente

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

$$= 1 + \frac{1}{nm} \sum_{i=1}^{n} (n - i) = 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right)$$

$$= 1 + \frac{\alpha}{2} - \frac{1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha) \quad \square$$

Bewertung der Laufzeiten

Unter unseren Annahmen

- Wenn $n = O(m)$, dann $\alpha = n/m = O(m)/m = O(1)$
- Dann benötigen alle Operationen $O(1)$ Zeit im Average Case

Jetzt: Welche Hash-Funktion verwenden wir?

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen
2.1 Direkte Adressierung
2.2 Chaining
2.3 Hash-Funktionen
2.4 Offene Adressierung

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Hash-Funktionen

Was ist eine gute Hash-Funktion?

- Erfüllt idealerweise die Annahme des Simple Uniform Hashing
- Das ist in Praxis nicht möglich, denn wir kennen die Schlüsselverteilung nicht
- Daher werden oft Heuristiken verwendet
- Außerdem: effizient berechenbar

Beispiel

- Divisionsmethode
- Multiplikationsmethode
- Universelles Hashing

Schlüssel als natürliche Zahlen

Wir interpretieren Schlüssel gewöhnlich als natürliche Zahlen.

Beispiel

- Vorzeichenbehaftete Ganzzahlen: Vorzeichen weglassen oder als vorzeichenlos interpretieren
- Zeichenketten
  - Interpretation als natürliche Zahl zu passender Basis
  - ASCII-Werte: A=65, &=38, D=68
  - ASCII hat 128 verschiedene Zeichen $\rightarrow$ Basis 128
  - Interpretation: $65 \cdot 128^2 + 38 \cdot 128 + 68 = 1\,069\,892$

Divisionsmethode

Definition

$$h(k) = k \bmod m$$

Diskussion

- Beispiel: $m = 20,\ k = 91 \rightarrow h(k) = 11$
- Vorteil: Schnell, nur eine Division
- Nachteil: Nicht für alle Werte von m geeignet
  - Keine Zweierpotenz, da sonst nur letzten lg m Bits berücksichtigt
  - Keine Zehnerpotenzen, falls wir im Dezimalsystem sind
  - Für Zeichenketten mit Basis 2^p ist $m = 2^p - 1$ ungünstig: Vertauschen von Zeichen ändern den Hashwert nicht
- Gute Wahl: Primzahl die nicht in der Nähe einer Zweierpotenz ist

Multiplikationsmethode

Definition

1. Wähle $0 < A < 1$
2. Multipliziere Schlüssel mit A
3. Nachkommastellen von kA extrahieren
4. Mit m multipizieren
5. Danach abrunden

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Diskussion

- Vorteil: Güte nicht abhängig von m (kann 2er-Potenz sein)
- Nachteil: Langsamer als Divisionsmethode
  (Aber kann effizient implementiert werden, z.B. wenn m 2er-Potenz)
- Arbeitet mit jeder gültigen Wahl von A
  - Aber nicht mit jeder Wahl gleich gut
  - Donald Knuth empfiehlt: $A = (\sqrt{5} - 1)/2 = 0.6180339887\ldots$

Universelles Hashing

- Worst Case-Verhalten kann bei Kenntnis der Hash-Funktion durch bösartige Wahl der Elemente erzwungen werden
- Ein Lösung: Randomisierung der Hash-Funktion
  - Aber nicht beliebig, sondern zufällige Wahl aus einer Menge von guten Funktionen
- Eine Klasse $\mathscr{H}$ von Hash-Funktionen von $U \to \{0, \ldots, m-1\}$ ist **universell**, wenn für jedes Schlüsselpaar $k \neq l$ die Anzahl der Hashfunktionen $h \in \mathscr{H}$ für die $h(k) = h(l)$ nicht größer als $|\mathscr{H}|/m$
- Beispiel: Für Primzahl p mit $p > |U|$ und $a, b \in \mathbb{N}$: $\{(ax + b \bmod p) \bmod m \mid 0 < a < p, 0 \leq b < p\}$
- Es gilt: Zwei Schlüssel kollidieren bei zufällig gewählter Hash-Funktion mit Wahrscheinlichkeit $\leq 1/m$
- Man kann zeigen: Mit Chaining und universellem Hashing haben alle Dictionary-Operationen erwartete Laufzeit $O(1 + \alpha)$

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen
2.1 Direkte Adressierung
2.2 Chaining
2.3 Hash-Funktionen
2.4 Offene Adressierung

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Offene Adressierung

- Alternative zur Kollisionsbehandlung
- Chaining: Elemente in Liste abgelegt
- **Offene Adressierung**: Schlüssel direkt in Hash-Tabelle ablegen
- Jeder Slot enthält also entweder einen Schlüssel oder NIL
- Somit $\alpha \leq 1$
- Idee: Speichere k in
  - Slot $h(x.key)$ falls frei
  - Sonst in einem anderen freien Feld

Suche

- $h(k)$ ausrechnen und Slot $h(k)$ untersuchen; genannt **Probe**
- Falls Slot $h(k)$
  - Den Schlüssel k enthält, ist die Suche erfolgreich
  - NIL enthält, ist die Suche nicht erfolgreich
- Ansonsten enthält Slot $h(k)$ einen Schlüssel $\neq k$
  - Dann berechnen wir den Index eines anderen Slots, um diesen zu probieren
  - Berechnung erfolgt abhängig von k mit Hilfe einer erweiterten Hash-Funktion

Probensequenz

Wir suchen freies Feld mit Hilfe einer erweiterten Hashfunktion

$$h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Probennummer}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Slotnummer}},$$

wobei für jeden Schlüssel k die **Probensequenz**

$$\langle h(k, 0),\ h(k, 1),\ \ldots,\ h(k, m-1) \rangle$$

eine Permutation von $\{0, 1, \ldots, m-1\}$ sein muss.

- Probensequenz wird genutzt um
  1. Speicherplatz für ein Element zu finden
  2. Ein Element zu suchen
- Doppeltes Probieren eines Slots ist nutzlos
  $\to$ Permutationsanforderung

Einfügen (Pseudo-Code)

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 "hash table overflow"
```

Suche (Pseudo-Code)

HASH-SEARCH(T, k)

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

Und Löschen?

- Schwierig
- Chaining besser geeignet, wenn Löschoperationen auftreten

Hash-Funktionen für offene Adressierung

- Ideal wäre **uniform hashing**, d.h. alle $m!$ Permutation unabhängig und gleichwahrscheinlich für jeden Schlüssel
- Schwer umzusetzen$\rightarrow$ Heuristiken
  - Erzeugen garantiert Permutation als Probensequenz
  - Aber nicht alle Permutationen
- Wir erweitern dazu gegebene Hash-Funktionen zu erweiterten Hash-Funktionen
  - Lineares Hashing
  - Quadratisches Hashing
  - Doppeltes Hashing

Lineares Hashing

Definition

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

Bemerkungen

- Sequentiell nächsten Slot probieren
- Initiale Probe bestimmt gesamte Probensequenz
  $\rightarrow$ Nur m mögliche Probensequenzen
- Nachteil: **Primäres Clustering**
  - Lange besetzte Folgen in der Hash-Tabelle werden immer länger
  - Denn: Leerer Slot, vor dem i volle Slots stehen, wird besetzt mit Wahrscheinlichkeit $(i + 1)/m$
  - Dadurch höhere durchschnittliche Einfüge- und Suchkosten

Quadratisches Hashing

Definition

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

mit $c_1, c_2 \neq 0$ und sorgfältig gewählt

Bemerkungen

- Startet auch mit Slot $h'(k)$, springt danach aber auf Basis quadratischer Funktion
- Das funktioniert besser als lineares Hashing, aber
  - Nicht alle Kombinationen von m, c_1, c_2 liefern Permutation
  - Immer noch nur m mögliche Probensequenzen
- Nachteil: **Sekundäres Clustering** (milder)
  - Zwei Schlüssel mit gleichem Hash-Wert haben gleiche Probensequenz

Doppeltes Hashing

Definition

$$h(k, i) := (h_1(k) + ih_2(k)) \bmod m$$

mit $h_2(k)$ relativ prim zu m

Bemerkungen

- Jetzt zwei Hash-Funktionen: eine für initiale Probe, eine für alle weiteren
- Funktioniert besser als lineares und quadratisches Hashing
  - $\Theta(m^2)$ Probesequenzen möglich
- Relativ prim heißt keine gemeinsamen Primfaktoren außer 1
  - Garantiert Permutation
  - z.B. m ist 2er-Potenz und h_2 erzeugt ungerade Zahl
  - z.B. m Primzahl und $1 < h_2(k) < m$

Analyse

Annahmen

- Analyse abhängig vom Ladefaktor
- Hash-Tabelle niemals voll, d.h. $0 \leq \alpha < 1$
- Uniform Hashing
- Kein Löschen
- In erfolgreicher Suche wird jeder Schlüssel mit gleicher Wahrscheinlichkeit gesucht

Vorgehen

- Wir berechnen die erwartete Anzahl von Proben
- Wieder zuerst erfolglose, dann erfolgreiche Suche

Erfolglose Suche (1)

Theorem

Die erwartete Anzahl von Proben für eine erfolglose Suche ist höchstens $1/(1-\alpha)$.

Beweis.

- Erfolglos heißt
  - Letzte Probe auf freien Slot
  - Jede vorherige auf besetzten Slot
- Ereignis $A_i = i$-te Probe wird durchgeführt und Slot besetzt
  - Wenn durchgeführt: $i-1$ Slots schon probiert, alle besetzt
  - $m-i+1$ Slots noch nicht probiert
    $\rightarrow$ Unter unseren Annahmen wird jeder Slot mit gleicher Wahrscheinlichkeit als nächster ausgewählt
  - $n-i+1$ noch nicht probierte, besetzte Slots

$$\mathbb{P}\left(A_i \mid A_1, A_2, \ldots, A_{i-1}\right) = \frac{n-i+1}{m-i+1}$$

Erfolglose Suche (2)

- Somit

$$\begin{aligned}\mathbb{P}(A_i) &= \mathbb{P}(A_1) \cdot \mathbb{P}(A_2 \mid A_1) \cdots \mathbb{P}(A_i \mid A_1, A_2, \ldots, A_{i-1}) \\ &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} = \prod_{j=0}^{i-1} \frac{n-j}{m-j} \\ &\leq \left(\frac{n}{m}\right)^i = \alpha^i\end{aligned}$$

- Zufallsvariable $X = \#\text{Proben}$
  - $\mathbb{P}(X \geq i) = \mathbb{P}(A_{i-1})$
- Es gilt für X natürlichzahlig:

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=0}^{\infty} i\,\mathbb{P}(X = i) = \sum_{i=0}^{\infty} i(\mathbb{P}(X \geq i) - \mathbb{P}(X \geq i+1)) \\ &= \sum_{i=1}^{\infty} \mathbb{P}(X \geq i)\end{aligned}$$

Erfolglose Suche (3)

- Alles zusammen

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} \mathbb{P}(X \geq i) = \sum_{i=1}^{\infty} \mathbb{P}(A_{i-1}) = \sum_{i=0}^{\infty} \mathbb{P}(A_i)$$
$$\leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

□

Interpretation

- Wenn α konstant, dann $O(1)$
- Wenn $\alpha = 0.5$, dann ≤ 2 Proben in Erwartung
- Wenn $\alpha = 0.9$, dann ≤ 10 Proben in Erwartung

Korollar

Die erwartete Anzahl von Proben für das Einfügen ist $\leq 1/(1-\alpha)$.

Beweis. Gleiche Folge wie erfolglose Suche. □

Erfolgreiche Suche (1)

Theorem

Die erwartete Anzahl von Proben für eine erfolgreiche Suche ist höchstens

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}.$$

Beweis.

- Suche nach $k \rightarrow$ gleiche Probensequenz wie Einfügen des Schlüssels k
- Wenn k der $(i+1)$-te eingefügte Schlüssel war, dann zu dieser Zeit
  - $\alpha = i/m$
  - Erwartete Länge der Probensequenz $\leq 1/(1-i/m) = m/(m-i)$
- Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit abgefragt

Erfolgreiche Suche (2)

- $X = \#$ Proben

$$\begin{aligned}\mathbb{E}[X] &\leq \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\ &\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\,\mathrm{d}x \\ &= \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\end{aligned}$$

□

Interpretation

- Es gilt $1/\alpha \ln(1/(1-\alpha)) \leq 1/1-\alpha$ (wie erwartet)
- Wenn α konstant, dann $O(1)$
- Wenn $\alpha = 0.5$, dann ≤ 1.4 Proben in Erwartung
- Wenn $\alpha = 0.9$, dann ≤ 2.6 Proben in Erwartung

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Binäre Suchbäume

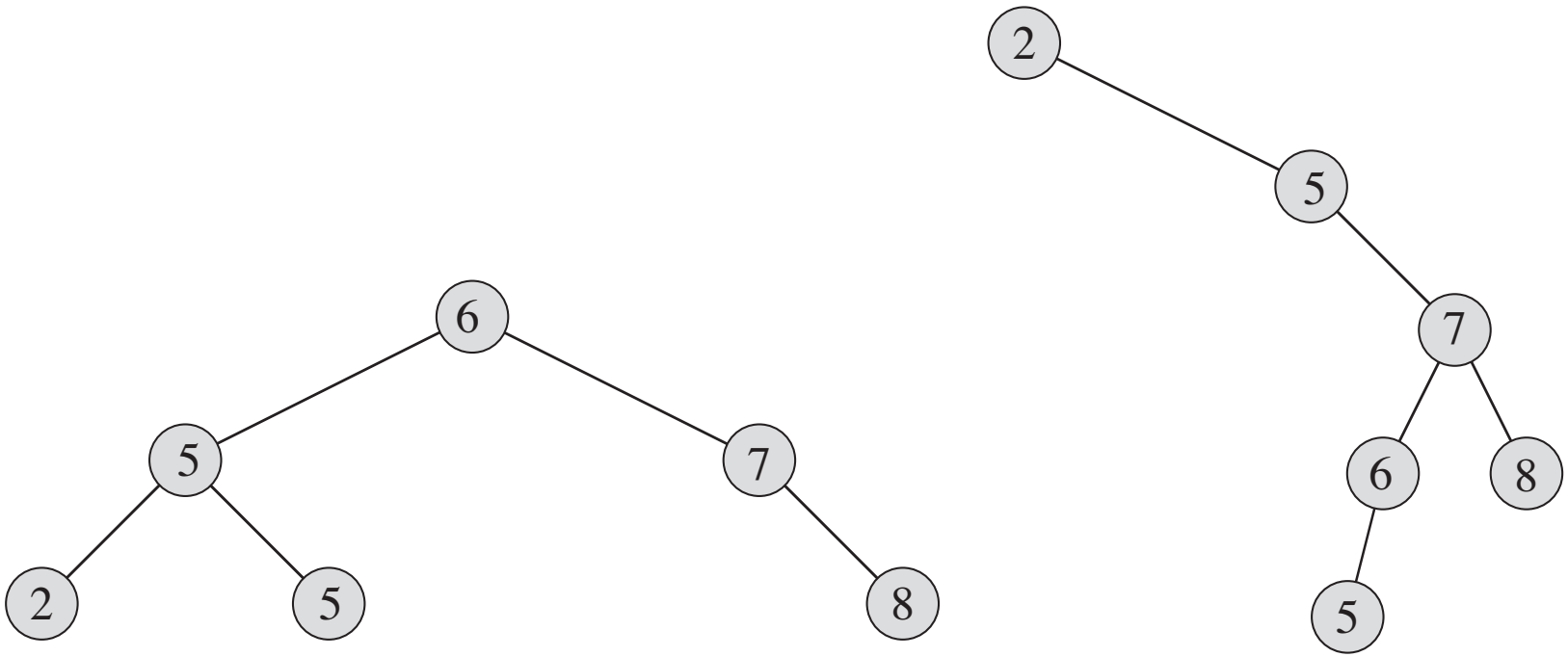
- Wichtige Datenstruktur für geordnete dynamische Mengen
- Unterstützt viele Operationen, z.B.
  - Suche
  - Minimum und Maximum
  - Vorgänger und Nachfolger
  - Einfügen und Löschen
- Operationen in $O(h)$ mit h Höhe des Baumes
- Höhe variiert
  - Vollständiger Binärbaum: $\Theta(\lg n)$
  - Liste: $\Theta(n)$

Binäre Suchbäume (BSB)

Definition

Ein **binärer Suchbaum** ist ein

- binärer Baum mit Attributen *left*, *right*, *p* und *key*, der
- die **binäre Suchbaumeigenschaft** erfüllt
  - Wenn y im linken Teilbaum von x, dann $y.key \leq x.key$
  - Wenn y im rechten Teilbaum von x, dann $y.key \geq x.key$



Traversierung des Suchbaums

Elemente in einem binären Suchbaum können leicht in aufsteigender Reihenfolge durchlaufen werden.

INORDER-TREE-WALK(x)

```
1: if x ≠ NIL
2:     INORDER-TREE-WALK(x.left)
3:     Output x.key
4:     INORDER-TREE-WALK(x.right)
```

- Kosten $\Theta(n)$
- Analog: PREORDER-TREE-WALK und POSTORDER-TREE-WALK

Suchen

TREE-SEARCH(x, k)

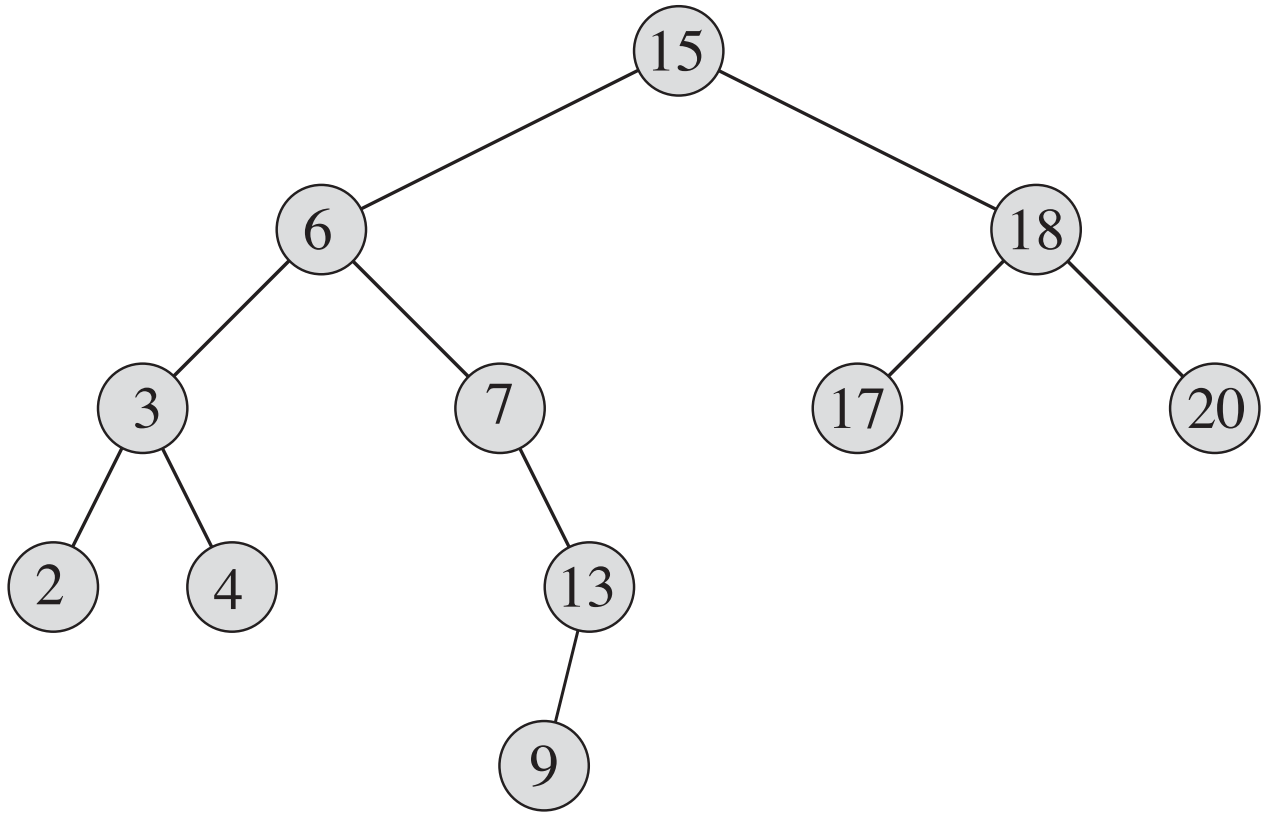
```
1: if x = NIL or k = x.key
2:     return x
3: if k < x.key
4:     return TREE-SEARCH(x.left, k)
5: else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH-ITERATIVE(x, k)

```
1: while x ≠ NIL and k ≠ x.key
2:     if k < x.key
3:         x ← x.left
4:     else x ← x.right
5: return x
```

- Aufruf: TREE-SEARCH$(T.root, x)$
- Laufzeit: $O(h)$

Beispiel



Minimum und Maximum

Die binäre Suchbaumeigenschaft garantiert, dass sich

- der kleinste Schlüssel im Knoten ganz links und
- der größte Schlüssel im Knoten ganz rechts

befindet.

TREE-MINIMUM(x)

```
1: while x.left ≠ NIL
2:     x ← x.left
3: return x
```

TREE-MAXIMUM(x)

```
1: while x.right ≠ NIL
2:     x ← x.right
3: return x
```

- Laufzeit: $O(h)$
- Bemerke: Keine Vergleiche notwendig

Nachfolger

- Wenn alle Schlüssel verschieden, dann ist Nachfolger von x der kleinste Schlüssel y so dass $y.key > x.key$
- Nachfolger des größten Schlüssels ist NIL
- Baumstruktur ausreichend, keine Vergleiche notwendig

TREE-SUCCESSOR(x)

```
1: if x.right ≠ NIL
2:     return TREE-MINIMUM(x.right)
3: y ← x.p
4: while y ≠ NIL and x = y.right
5:     x ← y
6:     y ← x.p
7: return y
```

- Laufzeit: $O(h)$
- TREE-PREDECESSOR analog

Änderungsoperationen

Änderungsoperationen sind

- Einfügen
- Löschen

Diese müssen die binäre Suchbaumeigenschaft bewahren.
Beide werden Laufzeit $O(h)$ haben.

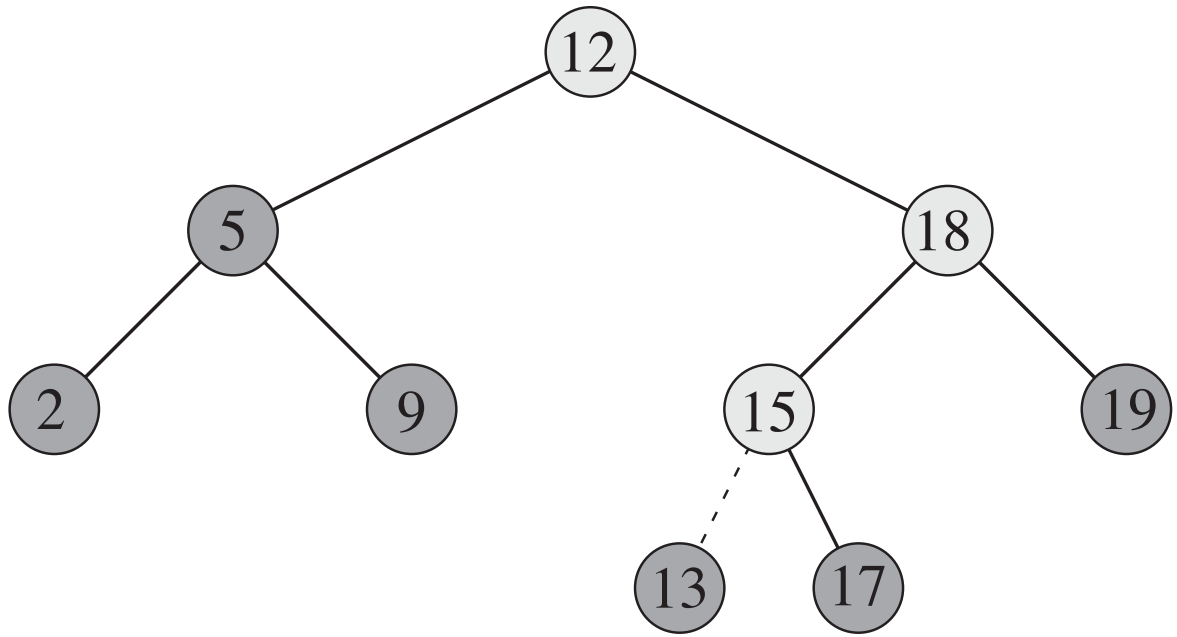
Einfügen

TREE-INSERT(T, z)

```
 1: y ← NIL
 2: x ← T.root
 3: while x ≠ NIL
 4:     y ← x
 5:     if z.key < x.key
 6:         x ← x.left
 7:     else x ← x.right
 8: p[z] ← y
 9: if y = NIL
10:     T.root ← z
11: else if z.key < y.key
12:     y.left ← z
13: else y.right ← z
```

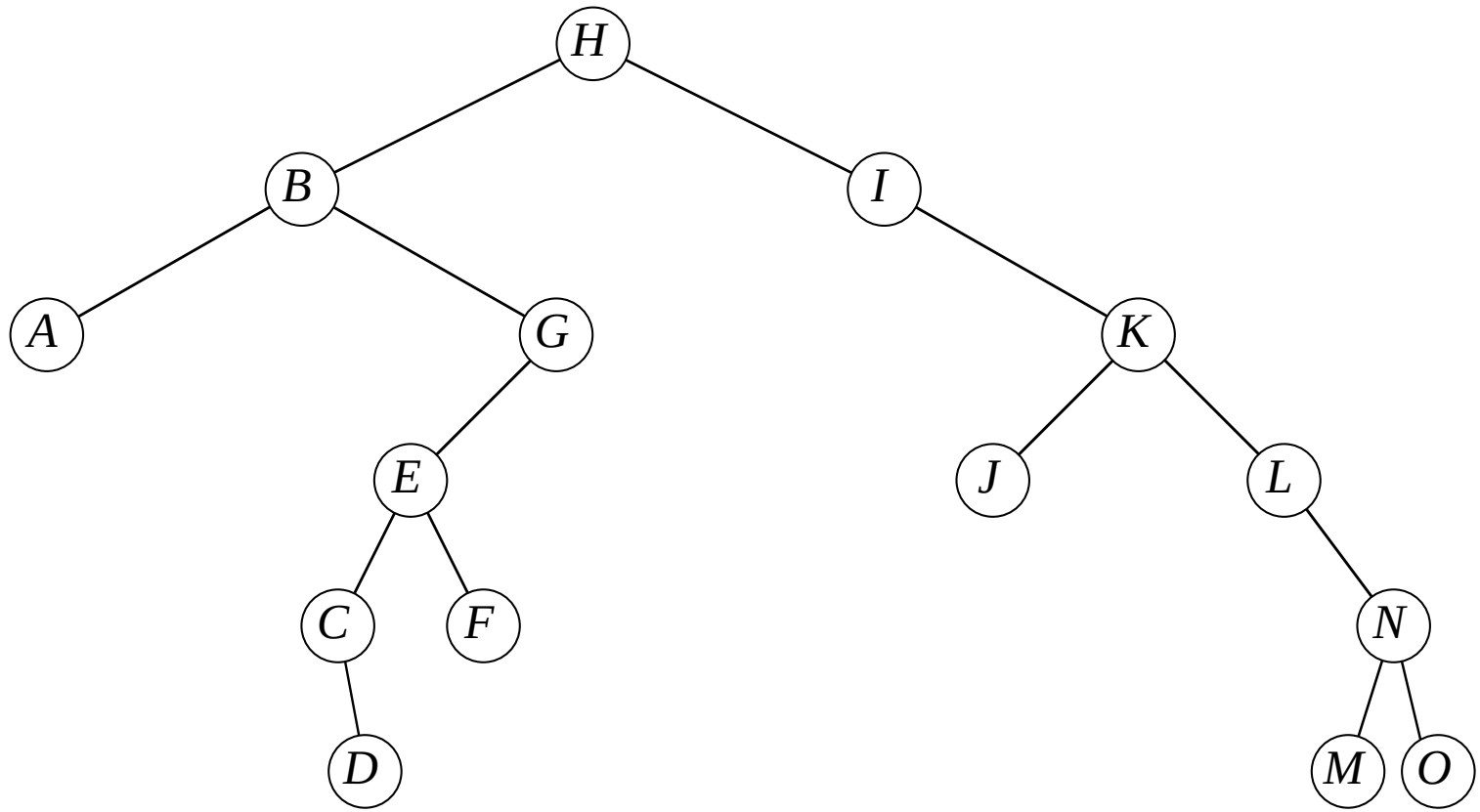
- Laufzeit: $O(h)$
- Kann zusammen mit INORDER-TREE-WALK zum Sortieren verwendet werden

Beispiel: Einfügen des Schlüssels 13



Löschen (1)

1. z ist Blatt $\rightarrow$ Ersetzen durch NIL
2. z hat einen Sohn $\rightarrow$ Ersetzen durch Sohn
3. z hat zwei Söhne $\rightarrow$ Ersetzen durch Nachfolger (komplexer)



Löschen (2)

Wir benötigen eine Methode, die in T den Teilbaum mit Wurzel u durch den Teilbaum mit Wurzel v ersetzt.

TRANSPLANT(T, u, v)

```
1: if u.p = NIL
2:     T.root ← v
3: else if u = u.p.left
4:     u.p.left ← v
5: else u.p.right ← v
6: if v ≠ NIL
7:     v.p ← u.p
```

- Laufzeit: $\Theta(1)$

Löschen (3)

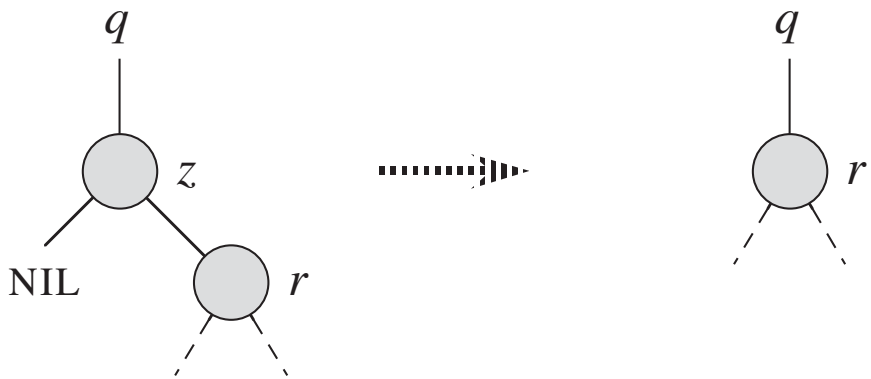
TREE-DELETE(T, z)

```
 1: if z.left = NIL
 2:     TRANSPLANT(T, z, z.right)                    // z has no left child
 3: else if z.right = NIL
 4:     TRANSPLANT(T, z, z.left)                     // z has just a left child
 5: else                                             // z has two children
 6:     y ← TREE-MINIMUM(z.right)                    // y is z's successor
 7:     if y.p ≠ z                                   // y is not right child of z
 8:         TRANSPLANT(T, y, y.right)                // delete y
 9:         y.right ← z.right
10:         y.right.p ← y
11:     // Replace z by y
12:     TRANSPLANT(T, z, y)
13:     y.left ← z.left
14:     y.left.p ← y
```

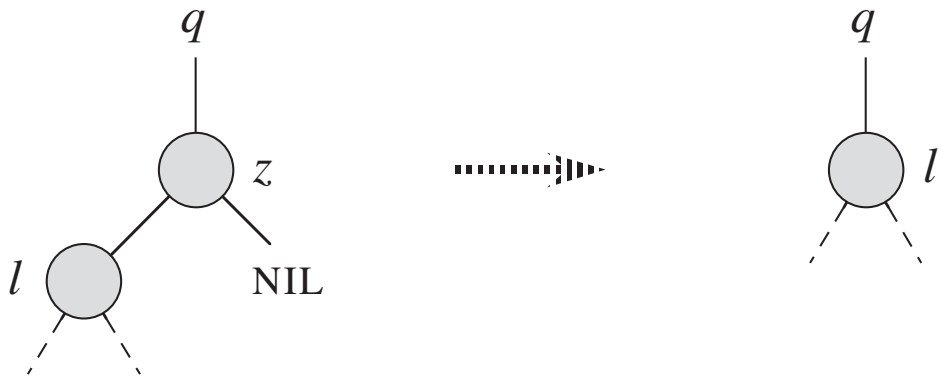
- Laufzeit: $O(h)$

Fälle für Tree-Delete (1)

Kein linker Sohn

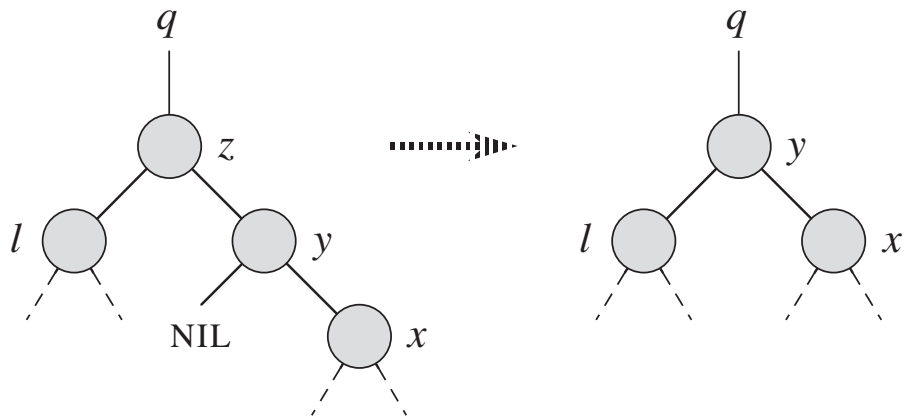


Nur linker Sohn

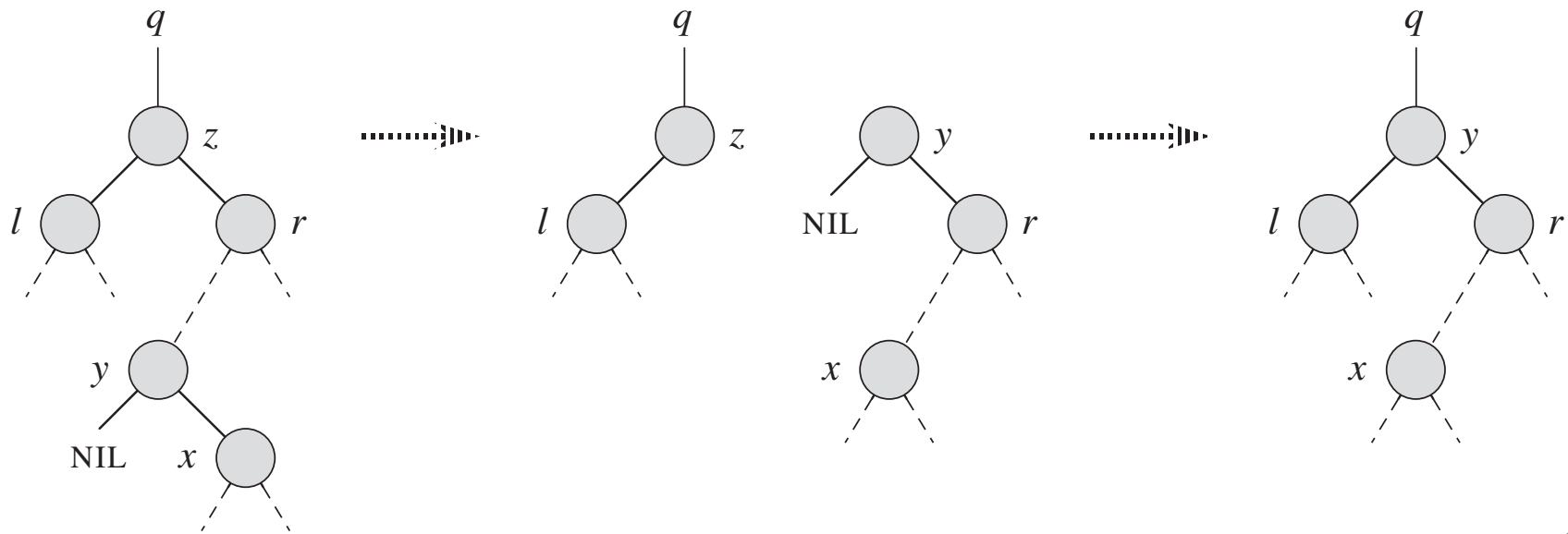


Fälle für Tree-Delete (2)

Zwei Söhne, rechter Sohn ist Nachfolger



Zwei Söhne, rechter Sohn ist nicht Nachfolger



Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume
4.1 Definition und Eigenschaften
4.2 Einfügen
4.3 Löschen

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Motivation

- Einfacher binärer Suchbaum kann unbalanciert sein
  - D.h. Pfade von Wurzel zu Blatt (NIL) haben sehr unterschiedliche Länge
  - Im ungünstigsten Fall: $h = n - 1$ (Liste)
  - Im besten Fall $h = \lceil \lg(n+1) \rceil - 1$ (vollständig)
- Mehrere Möglichkeiten Balanciertheit zu erreichen
  - Ziel ist $h = O(\lg n)$
  - Viele Operationen $O(h)$, wenn balanciert $O(\lg n)$
  - Änderungsoperationen werden entsprechend angepasst
- Hier: **Rot-Schwarz-Baum** (*red-black tree*)
  - Jeder Knoten ist entweder rot oder schwarz
  - Nur bestimmte Farbmuster erlaubt
  - Garantiert approximative Balanciertheit
  - Insbesondere: Höhe $O(\lg n)$

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume
4.1 Definition und Eigenschaften
4.2 Einfügen
4.3 Löschen

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Rot-Schwarz-Baum

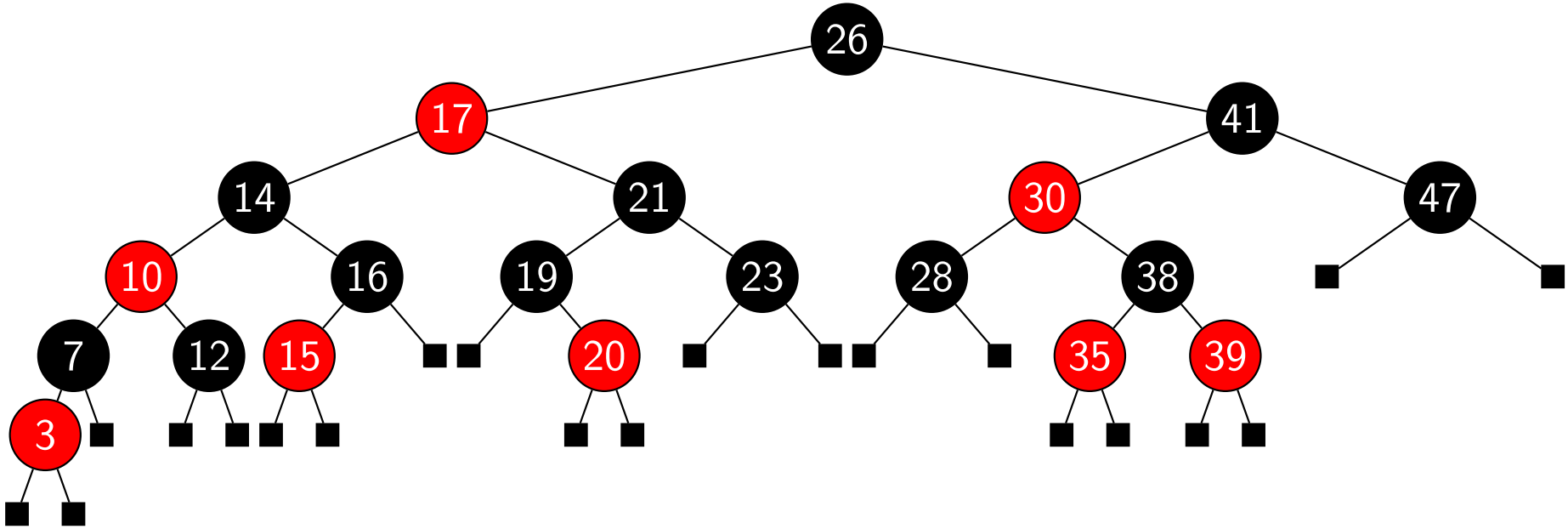
Definition

Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit den Attributen *key*, *left*, *right*, *p* und *color*, der folgende **Rot-Schwarz-Eigenschaften** erfüllt:

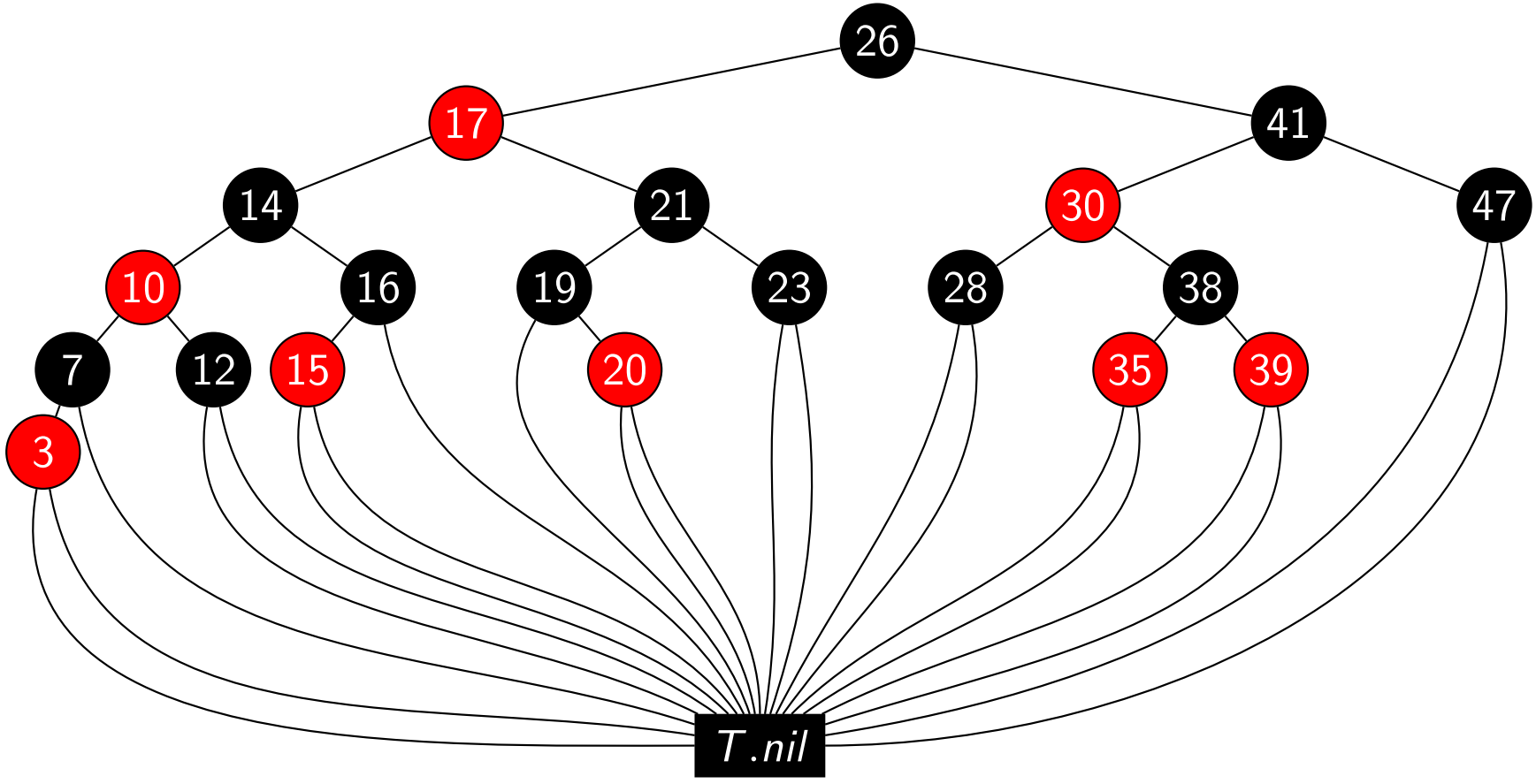
1. Jeder Knoten ist entweder schwarz oder rot
2. Die Wurzel ist schwarz
3. Jedes Blatt (NIL) ist schwarz
4. Falls ein Knoten rot ist, so sind beide Söhne schwarz
5. Jeder einfache Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten

Wir betrachten NIL-Werte (Blätter) als äußere Knoten.

Beispiel (mit äußeren Knoten)

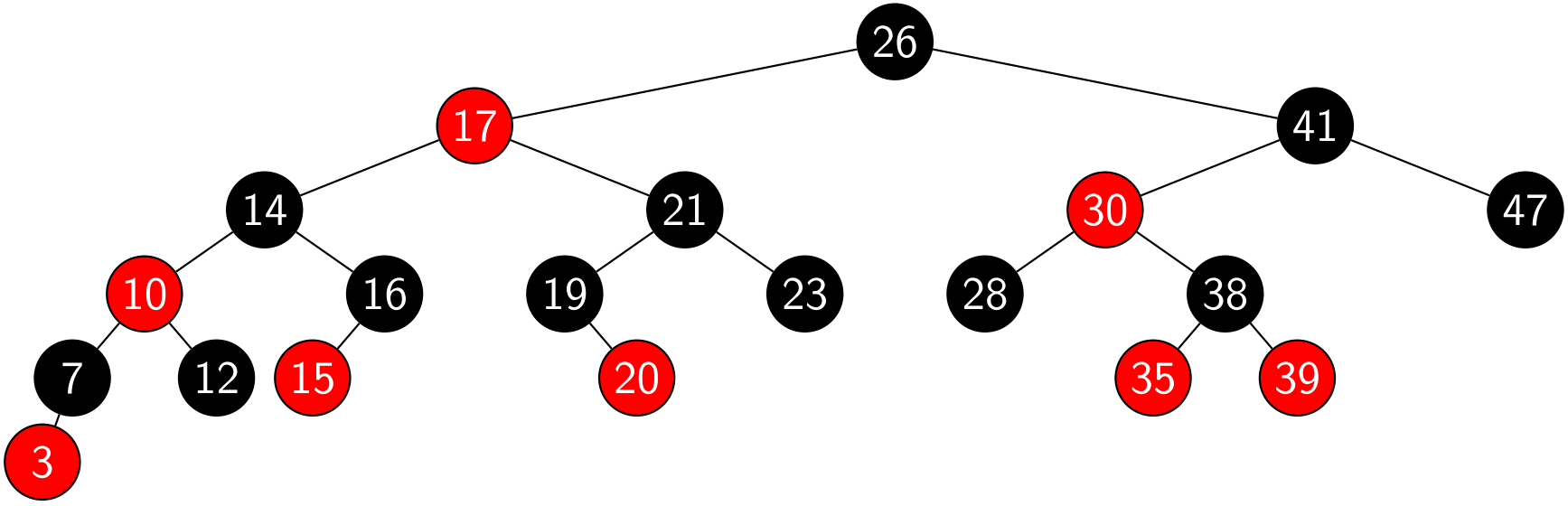


Beispiel (mit Sentinel)



- Explizite Speicherung der äußeren Konten ist aufwendig und verkompliziert Algorithmen
- Wir ersetzen sie daher durch einen einzigen Blattknoten $T.nil$
  - Genannt **Sentinel**
  - Immer schwarz, Attribute *key*, *left*, *right*, *p* beliebig

Beispiel (Kurzdarstellung)



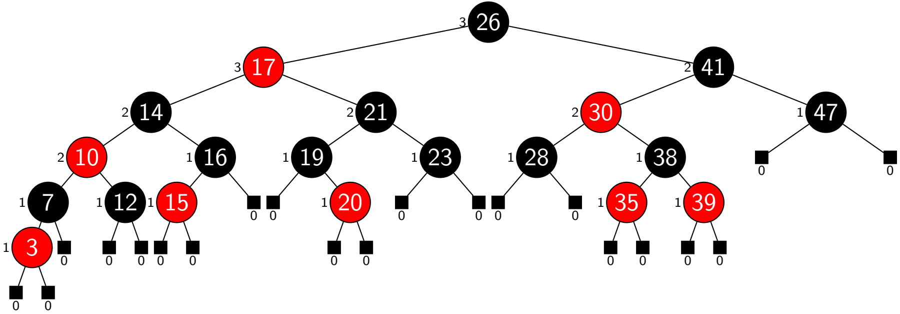
- In Zeichnungen lassen wir den Sentinel zur Übersichtlichkeit weg

Schwarzhöhe

Definition

Die Anzahl von schwarzen Knoten auf einem einfachen Pfad von x (ohne x mitzuzählen) zu einem Blatt heisst **Schwarzhöhe** (*black-height*) von x, notiert $\mathrm{bh}(x)$.

- Nach Eigenschaft 5 wohldefiniert



Balanciertheit (1)

Lemma

Ein Rot-Schwarz-Baum mit n inneren Knoten hat eine Höhe von maximal $2\lg(n+1)$.

Beweis. Zunächst zeigen wir per Induktion: Ein Teilbaum mit Wurzel x hat mindesten $2^{\mathrm{bh}(x)} - 1$ innere Knoten.

- *Induktionsanfang*, $\mathrm{bh}(x) = 0$
  - x ist Blatt, Teilbaum hat $2^0 - 1 = 0$ innere Knoten
- *Indukionsschritt*, $\mathrm{bh}(x) > 0$
  - Jeder Sohn von x hat entweder Schwarzhöhe $\mathrm{bh}(x)$ oder $\mathrm{bh}(x) - 1$, abhängig von seiner Farbe
  - Der Teilbaum hat somit zumindest

$$1 + 2(2^{\mathrm{bh}(x)-1} - 1) = 2^{\mathrm{bh}(x)} - 1$$

Knoten

Balanciertheit (2)

- Sei h die Höhe des RS-Baumes
- Eigenschaft 4 impliziert, dass mindestens die Hälfte der Knoten auf einem einfachen Pfad von der Wurzel zu einem Blatt schwarz sind
- Also gilt $\mathrm{bh}(T.root) \geq h/2$ und somit

$$n \geq 2^{\mathrm{bh}(T.root)} - 1 \geq 2^{h/2} - 1$$

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

□

Bewertung

- Operationen Suche, Minimum, Maximum, Vorgänger, Nachfolger in $O(\lg n)$ Zeit
- Selbiges gilt auch für Einfügen und Löschen
- Allerdings: Wie erhält man die Rot-Schwarz-Eigenschaft?
- Welche Farbe hat ein eingefügter Knoten?
  - Rot? $\rightarrow$ Eigenschaft 4 kann verletzt werden
  - Schwarz? $\rightarrow$ Eigenschaft 5 kann verletzt werden
- Was passiert beim Löschen eines Knotens?
  - Rot $\rightarrow$ OK
  - Schwarz $\rightarrow$ Eigenschaft 4 oder 5 können verletzt werden
- Wir werden sehen, dass korrektes Einfügen und Löschen auch in $O(\lg n)$ Zeit möglich ist

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume
4.1 Definition und Eigenschaften
4.2 Einfügen
4.3 Löschen

5. Anreicherung von Datenstrukturen

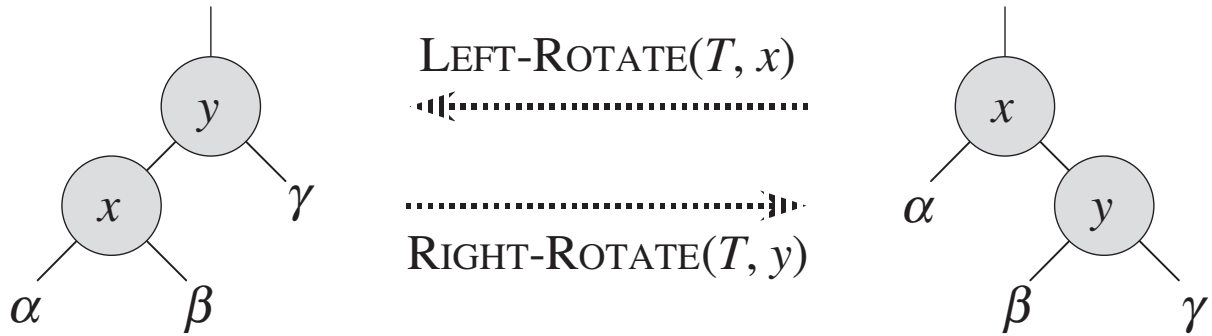
6. Disjunkte Mengen

Grundidee

- Wir fügen das neue Element mit der bereits bekannten Einfügemethode ein und färben es rot
- Wenn die Rot-Schwarz-Eigenschaften nicht zerstört werden, tun wir nichts
- Ansonsten stellen wir die Eigenschaften durch geeignete Umformungen und Umfärbungen wieder her
- Dazu verwenden wir **Rotationen** nach links und rechts
  - LEFT-ROTATE(T, x)
  - RIGHT-ROTATE(T, y)

Links- und Rechtsrotation

Wir verwenden **Rotation**, um die Baumstruktur zu verändern.

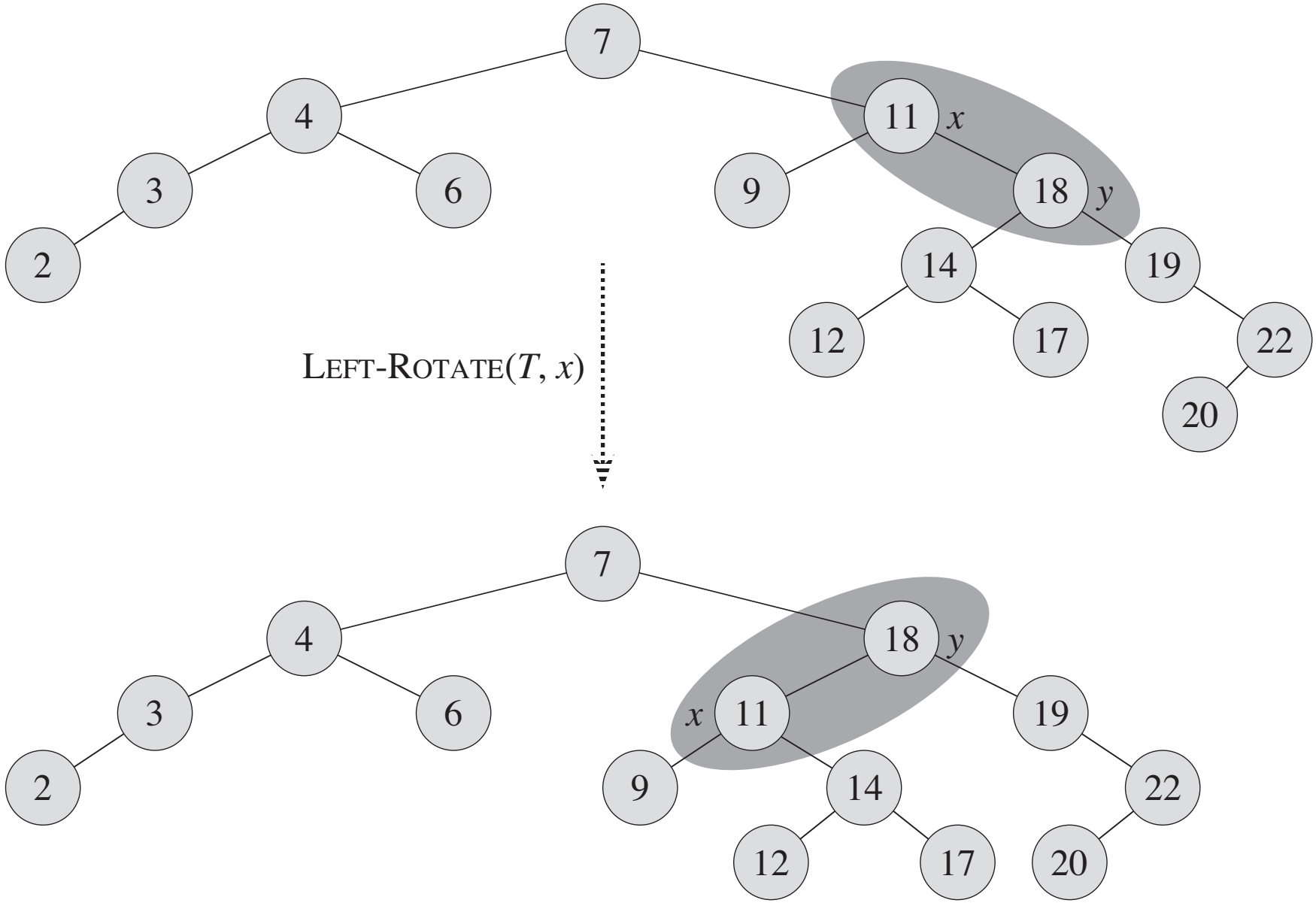


- α, β und γ repräsentieren beliebige Teilbäume
- Suchbaumeigenschaft bleibt erhalten

$$\alpha \leq x \leq \beta \leq y \leq \gamma$$

- Rot-Schwarz-Eigenschaft kann zerstört (oder eingeführt) werden
- Lokale Operation: nur Zeiger der Elemente um x und y werden verändert

Links- und Rechtsrotation: Beispiel



Links- und Rechtsrotation (Pseudo-Code)

LEFT-ROTATE(T, x)

```
 1: y ← x.right                                   // set y
 2: x.right ← y.left          // y's left becomes x's right subtree
 3: if y.left ≠ T.nil
 4:     y.left.p ← x
 5: y.p ← x.p                                     // link x's parent to y
 6: if x.p = T.nil
 7:     T.root ← y
 8: else if x = x.p.left
 9:     x.p.left ← y
10: else x.p.right ← y
11: y.left ← x                                    // Put x on y's left
12: x.p ← y
```

- Annahme: $x.right \neq T.nil$ (sonst Linksrotation nicht möglich)
- Annahme: Vaterknoten der Wurzel ist $T.nil$
- Laufzeit $O(1)$
- Rechtsrotation dual (tausche überall *left* und *right*)

RB-INSERT(T, z)

```
 1: y ← T.nil
 2: x ← T.root
 3: while x ≠ T.nil
 4:     y ← x
 5:     if z.key < x.key
 6:         x ← x.left
 7:     else x ← x.right
 8: z.p ← y
 9: if y = T.nil
10:     T.root ← z
11: else if z.key < y.key
12:     y.left ← z
13: else y.right ← z
14: z.left ← T.nil
15: z.right ← T.nil
16: z.color ← RED
17: RB-INSERT-FIXUP(T, z)
```

Verletzung der Rot-Schwarz-Eigenschaften

Sei z der eingefügte Knoten (rot).

1. Jeder Knoten ist entweder schwarz oder rot
   - OK
2. Die Wurzel ist schwarz
   - Verletzt wenn z Wurzel, sonst OK
3. Jedes Blatt (NIL) ist schwarz
   - OK
4. Falls ein Knoten rot ist, so sind beide Söhne schwarz
   - Verletzt wenn $z.p$ rot, sonst OK
5. Jeder einfache Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten
   - OK

Bemerke: Entweder keine Eigenschaft oder nur (2) oder nur (4) sind verletzt.

RB-Insert-Fixup(T, z)

```
 1: while z.p.color = RED
 2:     if z.p = z.p.p.left
 3:         y ← z.p.p.right
 4:         if y.color = RED
 5:             z.p.color ← BLACK                          // case 1
 6:             y.color ← BLACK                            // case 1
 7:             z.p.p.color ← RED                          // case 1
 8:             z ← z.p.p                                  // case 1
 9:         else
10:             if z = z.p.right
11:                 z ← z.p                                // case 2
12:                 LEFT-ROTATE(T, z)                      // case 2
13:             z.p.color ← BLACK                          // case 3
14:             z.p.p.color ← RED                          // case 3
15:             RIGHT-ROTATE(T, z.p.p)                     // case 3
16:     else ...        // as above, with "right" and "left" exchanged
17: T.root.color ← BLACK
```

Schleifeninvariante

Zu Beginn jeder Iteration der **while**-Schleife gilt:

a) z ist rot

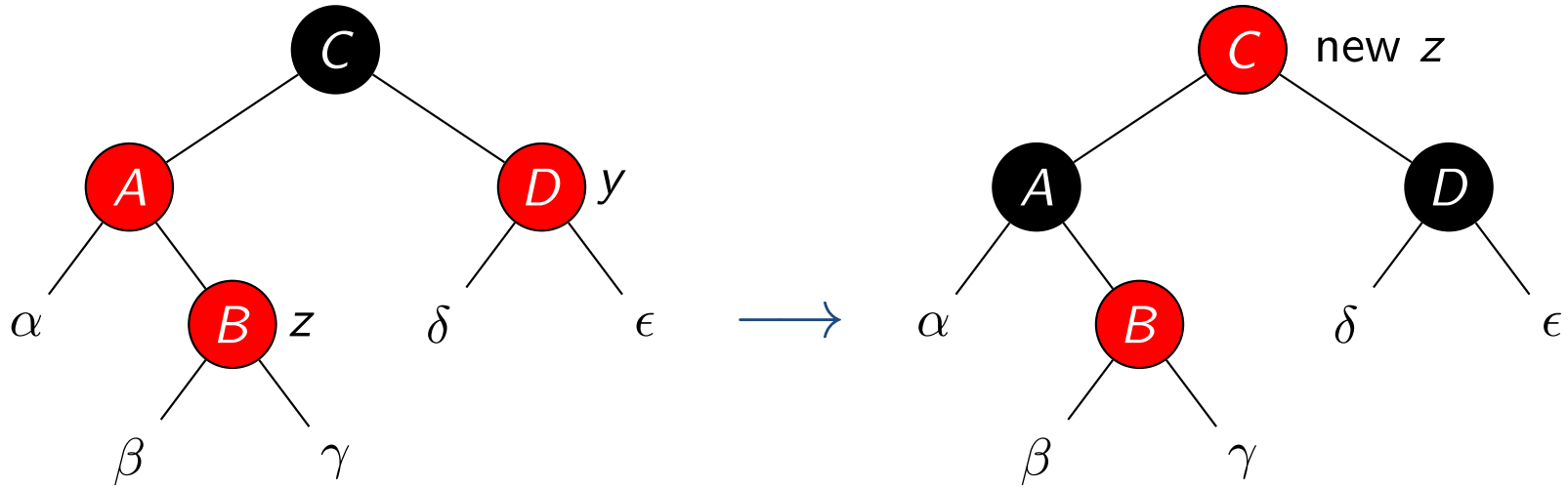
b) Höchstens eine Verletzung der Rot-Schwarz-Eigenschaften
   - Eigenschaft 2 (E2): z ist rote Wurzel
   - Eigenschaft 4 (E4): z und $z.p$ sind beide rot

Beweis.

- *Induktionsanfang*: Schon besprochen
- *Terminierung*
  - Terminiert wenn $z.p$ schwarz
  - Eigenschaft 4 ist OK
  - Eigenschaft 2 verletzt, wenn z Wurzel
    $\rightarrow$ Wird in Z. 17 wiederhergestellt
- *Induktionsschritt*: 6 Fälle, drei davon symmetrisch
  1. Onkel ist rot
  2. Onkel ist schwarz, z ist rechter Sohn
  3. Onkel ist schwarz, z ist linker Sohn

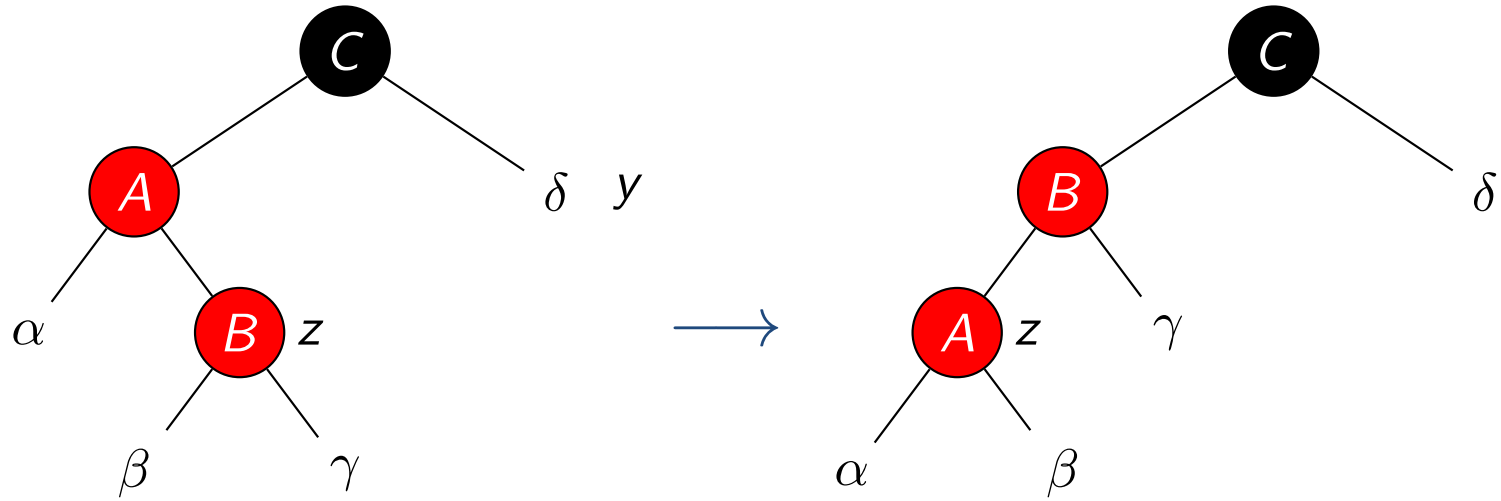
Fall 1: Onkel ist rot

Sei y der Onkel von z (Z. 3). Hier: z rechter Sohn (linker S. dual).



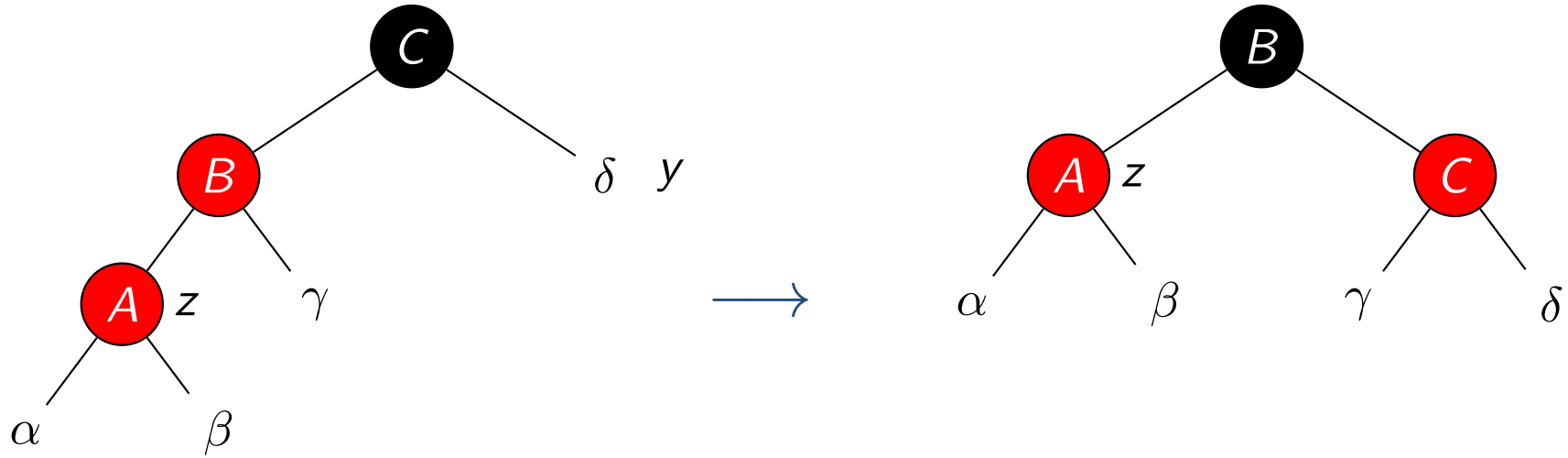
- Vater $z.p$ ist rot (Z. 1), Großvater schwarz (Invariante)
- Onkel $y = z.p.p.right$ ist rot (Z. 2–4)
- Impliziert dass Großvater $z.p.p$ schwarz (Eigenschaft 4)
- Schwarzfärben von $z.p$ und y (Z. 5–6) $\rightarrow$ E2 OK, E5 potentiell verletzt
- Rotfärben von $z.p.p$ (Z. 7)
  $\rightarrow$ E5 OK, aber $z.p.p$ verletzt potentiell E2 oder E4
- Deswegen: $z.p.p$ wird neues z (Z. 8)

Fall 2: Onkel ist schwarz, z ist rechter Sohn



- z nach oben schieben und links rotieren (Z. 11–12)
- z bleibt auf gleicher Tiefe, sofort Eintritt in Fall 3 (Z. 13–15)

Fall 3: Onkel ist schwarz, z ist linker Sohn



- Färbe $z.p$ schwarz und $z.p.p$ rot (Z. 13–14)
- Rechtsrotation um $z.p.p$ (Z. 15)
- Keine zwei roten Knoten nebeneinander $\rightarrow$ E4 erfüllt
- z ist nicht rote Wurzel $\rightarrow$ E2 erfüllt
- z ist schwarz $\rightarrow$ Schleife wird nicht noch einmal betreten $\square$

Analyse

- Iteration der **while**-Schleife: $O(1)$
- Jede Iteration ist entweder die letzte (Fall 2/3) oder setzt z zwei Ebenen nach oben (Fall 1)
- $O(\lg n)$ Ebenen $\rightarrow$ $O(\lg n)$ Iterationen
- Somit auch $O(\lg n)$ Laufzeit
- Bemerke: Maximal 2 Rotationen benötigt

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume
4.1 Definition und Eigenschaften
4.2 Einfügen
4.3 Löschen

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Löschen (Pseudo-Code, 1)

Wir benötigen eine Methode, die in T den Teilbaum mit Wurzel u durch den Teilbaum mit Wurzel v ersetzt.

RB-TRANSPLANT(T, u, v)

```
1: if u.p = T.nil
2:     T.root ← v
3: else if u = u.p.left
4:     u.p.left ← v
5: else u.p.right ← v
6: v.p ← u.p                                  // now unconditional
```

- $v.p$ zeigt jetzt *immer* auf den Vater von u
- Auch wenn $v = T.nil$ (diese Eigenschaft nutzen wir später)

Löschen (Pseudo-Code, 2)

RB-DELETE(T, z)

```
 1: y ← z
 2: y-original-color ← y.color
 3: if z.left = T.nil
 4:     x ← z.right
 5:     RB-TRANSPLANT(T, z, z.right)          // z has no left child
 6: else if z.right = T.nil
 7:     x ← z.left
 8:     RB-TRANSPLANT(T, z, z.left)       // z has just a left child
 9: else                                      // z has two children
10:     y ← TREE-MINIMUM(y.right)             // y is z's successor
11:     y-original-color ← y.color
12:     x ← y.right
```

Löschen (Pseudo-Code, 3)

```
13:     if y.p ≠ z                                  // y is not right child of z
14:         RB-Transplant(T, y, y.right)                    // delete y
15:         y.right ← z.right
16:         y.right.p ← y
17:     else                                        // y is right child of z
18:         x.p ← y
19:     // Replace z by y
20:     RB-Transplant(T, z, y)
21:     y.left ← z.left
22:     y.left.p ← y
23:     y.color ← z.color
24: if y-original-color = BLACK
25:     RB-Delete-Fixup(T, x)
```

Unterschiede zu Tree-Delete

1. y ist Knoten der entfernt wird (falls z weniger als zwei Söhne) bzw. der verschoben wird (falls z zwei Söhne)
   - Sonst bleibt der Baum unverändert
   - Wenn verschoben, dann eingefärbt wie z (Z. 23)
2. y's ursprüngliche Farbe wird ermittelt, denn wenn schwarz, kann Rot-Schwarz-Eigenschaften verletzt werden
3. x ist der Knoten, der an y's ursprüngliche Position verschoben wird
   - Entweder y's ursprünglicher einziger Sohn
   - Oder $T.nil$ (wenn y Blatt war)
4. $x.p$ wird auf Position von y's ursprünglichen Vater gesetzt
   - Entweder in Z. 18 (wenn z Vater von y)
   - Oder in RB-Transplant in Z. 14 (sonst)
5. Wenn y schwarz, dann können Verletzungen der Rot-Schwarz-Eigenschaften auftreten $\rightarrow$ Aufruf von RB-Delete-Fixup

Verletzung der Rot-Schwarz-Eigenschaft (y rot)

Wenn entfernter/verschobener Knoten y rot war, dann

1. Jeder Knoten ist entweder schwarz oder rot $\rightarrow$ OK
2. Die Wurzel ist schwarz
   - Wenn y entfernt, dann konnte es nicht Wurzel sein $\rightarrow$ OK
   - Wenn y verschoben, dann übernimmt es Farbe von z (Z. 23) $\rightarrow$ OK
3. Jedes Blatt (NIL) ist schwarz $\rightarrow$ OK
4. Falls ein Knoten rot ist, so sind beide Söhne schwarz
   - Fall 1: z hat keinen oder einen Sohn, dann $y = z$ und somit roter Knoten entfernt $\rightarrow$ OK
   - Fall 2: z hat zwei Söhne, dann wurde y verschoben indem es zuerst entfernt wurde (Z. 14–16, OK da rot) und dann an z's Stelle gesetzt und wie z eingefärbt wird (Z. 20–23) $\rightarrow$ OK
5. Jeder einfache Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten
   - OK, Fälle wie oben

Verletzung der Rot-Schwarz-Eigenschaft (y schwarz)

Wenn entfernter/verschobener Knoten y (mit ursprünglichem Sohn x) schwarz war, dann

1. Jeder Knoten ist entweder schwarz oder rot $\rightarrow$ OK
2. Die Wurzel ist schwarz
   - Verletzt wenn z Wurzel mit nur einem Sohn und dieser rot war
   - Sonst OK
3. Jedes Blatt (NIL) ist schwarz $\rightarrow$ OK
4. Falls ein Knoten rot ist, so sind beide Söhne schwarz
   - x tritt an Position von y
   - Verletzt wenn x rot und y's ursprünglicher Vater ($= x.p$) auch rot
   - Sonst OK
5. Jeder einfache Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten
   - Jeder einfache Pfad der ursprünglich y enthielt hat jetzt einen schwarzen Knoten weniger
   - $\rightarrow$ Verletzt für jeden ursprünglichen Vorfahren von y
   - Wäre OK, wenn wir x (an ursprünglicher Position von y) ein **Extraschwarz** geben würden

Extraschwarz

- Wir geben Knoten x ein Extraschwarz, wenn wir die Anzahl der schwarzen Knoten von einem inneren Knoten zu einem Blatt zählen
- Eigenschaft 5 ist dann OK, aber Eigenschaft 1 verletzt
  - x is entweder **doppelt-schwarz** (wenn $x.color = \text{BLACK}$)
  - oder **rot-schwarz** (wenn $x.color = \text{RED}$)
- Bemerke: Das Extraschwarz ist gedacht, d.h. $x.color$ ist immer noch rot oder schwarz
- Stattdessen: Der Knoten, auf den x zeigt, besitzt das Extraschwarz

Grundidee

Wir verschieben das Extraschwarz nach oben bis

- Extraschwarz auf rotem Knoten liegt $\rightarrow$ Knoten schwarz färben
- Extraschwarz auf Wurzel liegt $\rightarrow$ einfach weglassen
- Extraschwarz durch bestimmte Rotationen und Umfärbungen entfernt werden kann

Algorithmus RB-Delete-Fixup

- Behebt alle Verletzungen (wir zeigen E5, E2+E4 Übung)
- Innerhalb der **while**-Schleife
  - x zeigt auf doppelt schwarzen Nicht-Wurzel-Knoten
  - w ist x's Geschwisterknoten
  - w ist nicht $T.nil$ (sonst E5 an $w.p = x.p$ verletzt)
- 8 Fälle, 4 symmetrisch
  1. w ist rot
  2. w ist schwarz und beide Söhne ebenso
  3. w ist schwarz, linker Sohn rot, rechter Sohn schwarz
  4. w ist schwarz, rechter Sohn rot

Löschen (Pseudo-Code, 1)

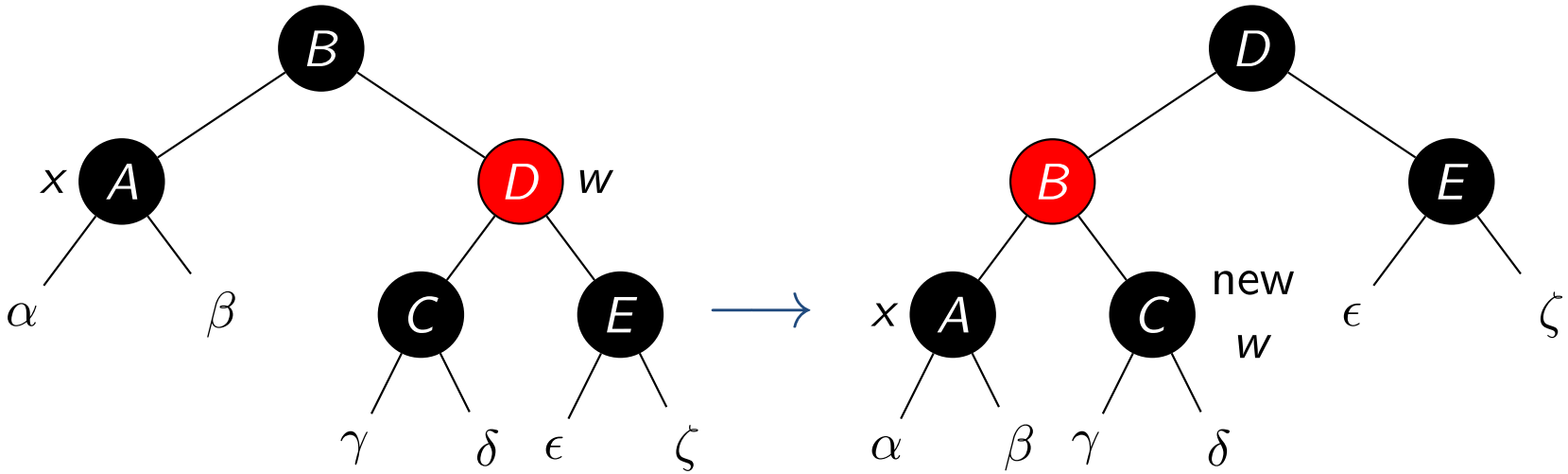
RB-DELETE-FIXUP(T, x)

```
1: while x ≠ T.root and x.color = BLACK
2:     if x = x.p.left
3:         w ← x.p.right
4:         if w.color = RED
5:             w.color ← BLACK                          // case 1
6:             x.p.color ← RED                          // case 1
7:             LEFT-ROTATE(T, x.p)                      // case 1
8:             w ← x.p.right                            // case 1
9:         // continue to case 2, 3, or 4
```

Löschen (Pseudo-Code, 2)

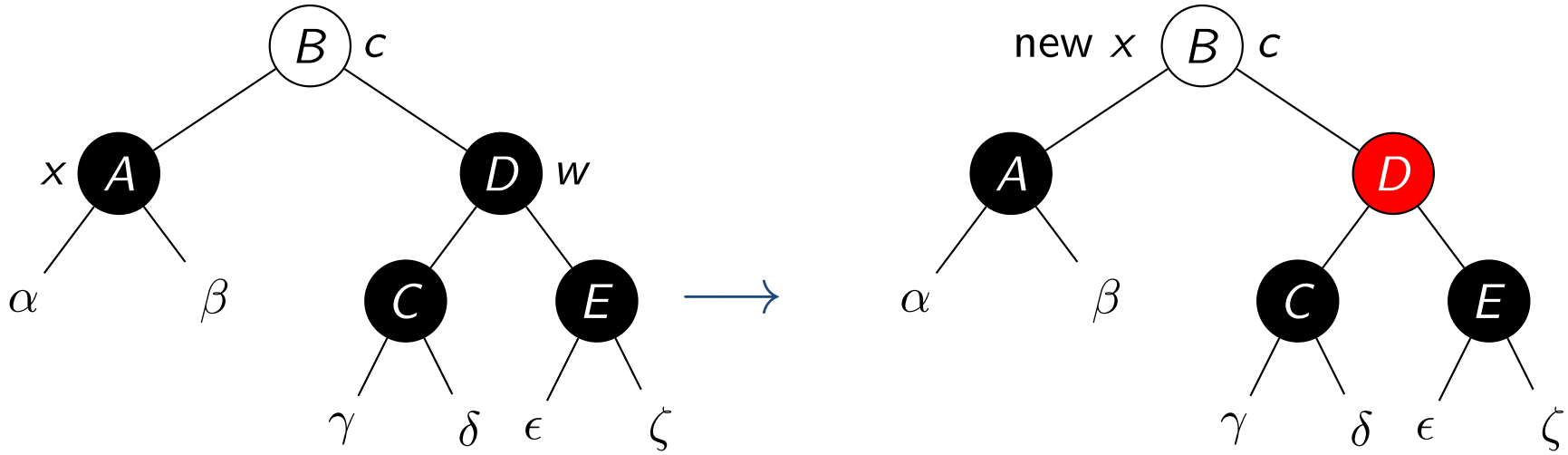
```
10:         if w.left.color = BLACK and w.right.color = BLACK
11:             w.color ← RED                                   // case 2
12:             x ← x.p                                         // case 2
13:         else
14:             if w.right.color = BLACK
15:                 w.left.color ← BLACK                        // case 3
16:                 w.color ← RED                               // case 3
17:                 RIGHT-ROTATE(T, w)                          // case 3
18:                 w ← x.p.right                               // case 3
19:             w.color ← x.p.color                             // case 4
20:             x.p.color ← BLACK                               // case 4
21:             w.right.color ← BLACK                           // case 4
22:             LEFT-ROTATE(T, x.p)                             // case 4
23:             x ← T.root                                      // case 4
24:     else ...      // as above, with "right" and "left" exchanged
25: x.color ← BLACK
```

Fall 1: w ist rot



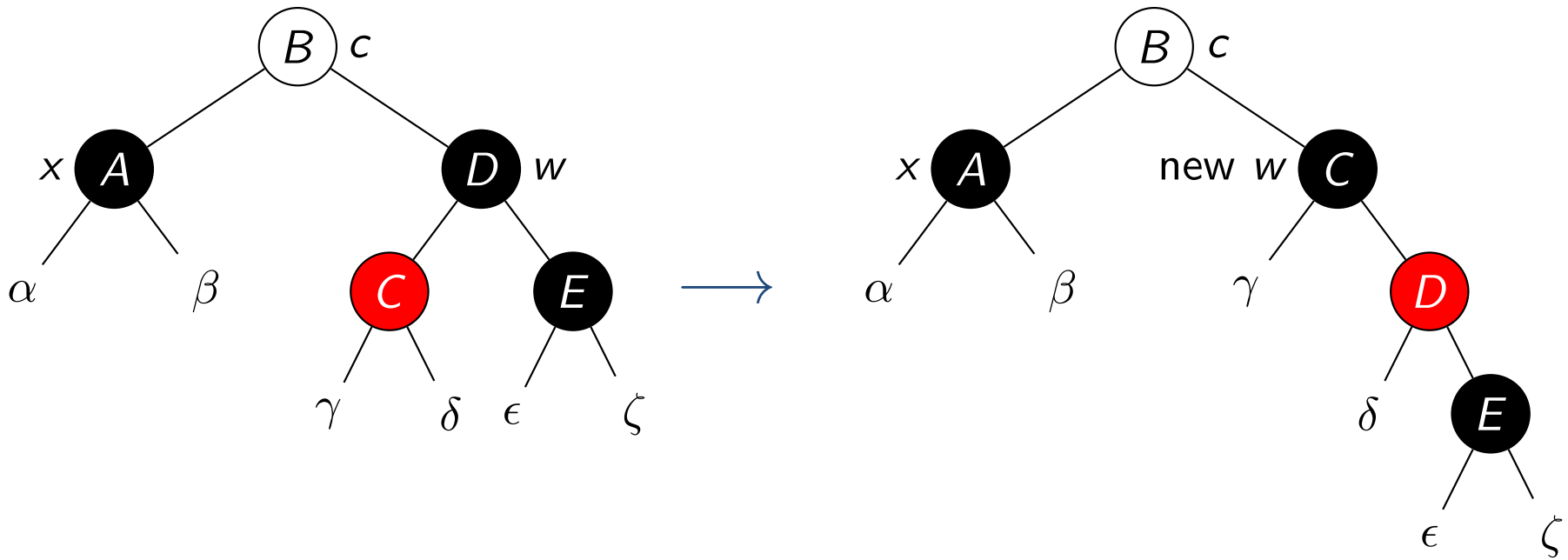
- w muss schwarze Kinder haben
- Wir färben w schwarz und dessen Vater rot
- Dann rotieren wir links um $x.p$
- Neuer Geschwisterknoten von x muss schwarz sein (ursprünglich Kind von w)
- Wir gehen sofort zu Fall 2, 3 oder 4

Fall 2: w ist schwarz und beide Söhne ebenso



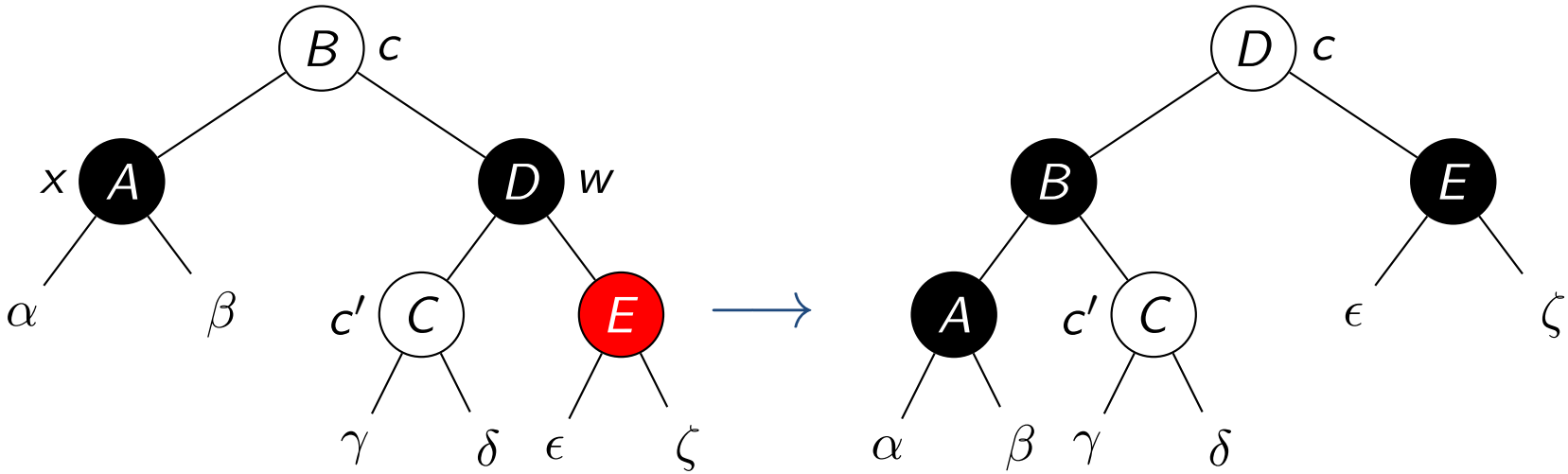
- $x.p$ hat beliebige Farbe (c)
- Wir ziehen das Extraschwarz nach oben
- und gleichen aus, indem wir D rot färben
- Wenn wir von Fall 1 kommen, dann ist das neue x rot
  $\rightarrow$ Schleife wird verlassen und x schwarzgefärbt

Fall 3: w ist schwarz, linker Sohn rot, rechter Sohn schwarz



- Färbe w rot und linken Sohn schwarz
- Rechtsrotiere um w
- Neuer Geschwisterknoten ist schwarz mit rotem rechten Sohn $\rightarrow$ Fall 4

Fall 4: w ist schwarz, rechter Sohn rot



- w und Vater $w.p$ tauschen Farben, w's rechter Sohn wird schwarz
- Dann Linksrotation auf Vater $w.p$
- Entfernen des Extraschwarz (E5 wird nicht verletzt)
- Wir sind fertig; $x \leftarrow T.root$ um zu terminieren

Analyse

- $O(\lg n)$ Zeit für Löschen ohne RB-Delete-Fixup
- Fall 1, 3, 4 werden höchstens einmal ausgeführt
  $\rightarrow \leq 3$ Rotationen, $O(1)$
- Fall 2 kann mehrmals ausgeführt werden, aber dann steigt x nach oben $\rightarrow O(\lg n)$
- Somit: $O(\lg n)$ gesamt

Bemerke: Sowohl Einfügen als auch Löschen führen $O(1)$ Rotationen aus.

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen
5.1 Dynamische Ordnungsstatistiken
5.2 Anreicherung von Rot-Schwarz-Bäumen
5.3 Intervallbäume

6. Disjunkte Mengen

Datenstrukturen im Algorithmenentwurf

- Standarddatenstrukturen sind hilfreich für den Algorithmenentwurf
- Manchmal erfüllen sie allerdings nicht alle Anforderungen
  - Z.B.: Eine benötigte Operation kann nicht durchgeführt werden oder ist zu aufwändig
- Es ist jedoch ungewöhnlich, dass komplett neue Datenstrukturen benötigt werden
- Häufig: Bekannte Datenstrukur verwenden und **anreichern** (*augmentation*)
  - Mehr Informationen verwalten, um neue Operationen zu unterstützen
  - Aber: neue Information muss gewartet werden ohne dass Effizienz der bereits existierenden Operationen verlorengeht
- Hier: Anreicherung von Rot-Schwarz-Bäumen

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen
5.1 Dynamische Ordnungsstatistiken
5.2 Anreicherung von Rot-Schwarz-Bäumen
5.3 Intervallbäume

6. Disjunkte Mengen

Ordnungstatistiken und Selektion

Die **i-te geordnete Statistik** einer n-elementigen Menge ist das i-t kleinste Element.

- Minimum ist erste geordnete Statistik ($i = 1$)
- Maximum ist n-te geordnete Stasistik ($i = n$)
- Median ist "auf halben Weg" ($i = (n + 1)/2$ wenn n ungerade)

Selektion

- Das **Selektionsproblem** fragt nach der i-ten Ordnungsstatistik
- Einfache Lösung: Sortieren und i-tes Element der sortierten Folge zurückliefern ($O(n \lg n)$)
- Kann aber auch in $O(n)$ gelöst werden (wie in Übung)

Dynamische Ordnungstatistiken

Wir wollen Rot-Schwarz-Bäume so erweitern, dass sie zusätzlich die folgenden beiden Operationen effizient unterstützen:

- OS-SELECT(x, i) liefert Zeiger auf i-t kleinstes Element im Teilbaum mit Wurzel x
- OS-RANK(T, x) liefert Rang von x
- Bei Schlüsselduplikaten: Verwende Ordnung der In-Order-Traversierung

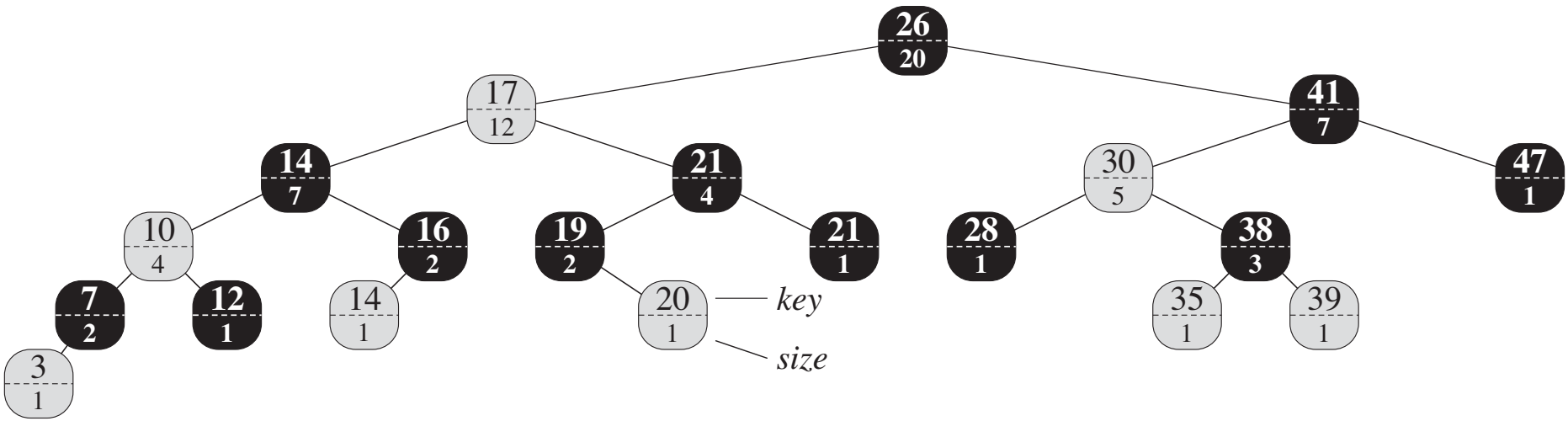
Umsetzung

- Mit Rot-Schwarz-Bäumen kann man beide Operationen in $O(i + \lg n)$ Zeit umsetzen
  - Gut für kleine i / kleinen Rang
  - Median aber z.B. aufwändig ($\Omega(n)$)
- Unser Ziel: $O(\lg n)$ durch Anreicherung

Ordnungsstatistikbaum

Ein **Ordnungsstatistikbaum** ist ein Rot-Schwarz-Baum, in dem jeder Knoten x ein zusätzliches Attribute $x.size$ besitzt.

- $x.size$ = **Teilbaumgröße** = Anzahl der inneren Knoten im Teilbaum mit Wurzel x (inklusive x)
- $x.size = 0$ für Blätter (und somit $T.nil.size = 0$)
- $x.size = x.left.size + x.right.size + 1$ für innere Knoten



Selektion

OS-SELECT(x, i)

```
1: r ← x.left.size + 1
2: if i = r
3:     return x
4: else if i < r
5:     OS-SELECT(x.left, i)
6: else OS-SELECT(x.right, i − r)
```

- Aufruf mit OS-SELECT$(T.root, i)$
- $O(\lg n)$

Rang

OS-Rank(T, x)

```
1: r ← x.left.size + 1
2: y ← x
3: while y ≠ T.root
4:     if y = y.p.right
5:         r ← r + y.p.left.size + 1
6:     y ← y.p
7: return r
```

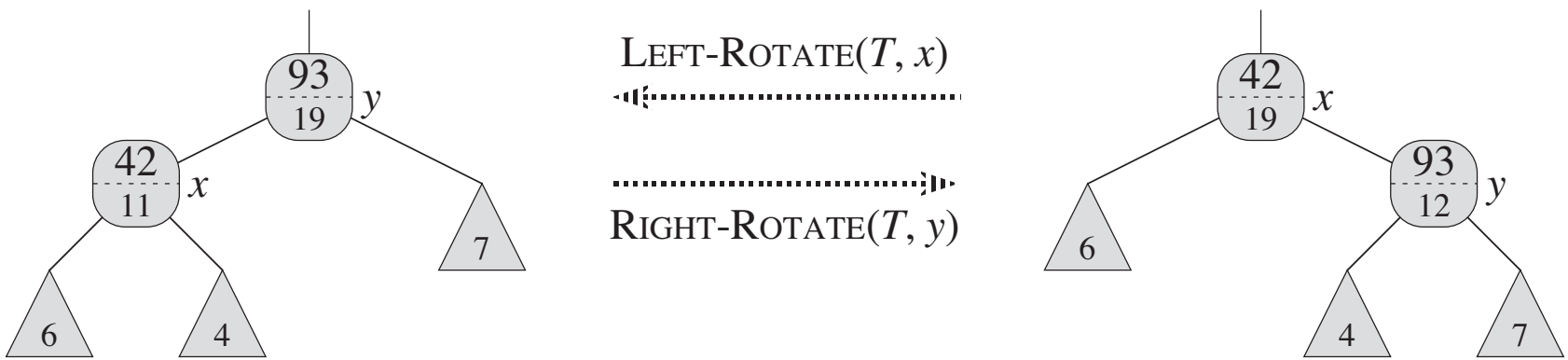
- Schleifeninvariante: r ist Rang von x in Teilbaum mit Wurzel y
- $O(\lg n)$

Wartung der Teilbaumgrößen

- Teilbaumgrößen müssen effizient gewartet werden
- Sonst muss man sie ggf. alle neu berechnen $\Theta(n)$
- Das ist in $O(\lg n)$ möglich $\rightarrow$ Ordnungsstatistikbäume bewahren die Effizienz von Rot-Schwarz-Bäumen

Einfügen

- Während Abwärtssuche nach Einfügeposition erhöhen wir den Zähler auf jedem betrachteten Knoten um 1
- Danach werden die Rot-Schwarz-Eigenschaften wiederhergestellt
  - Aufwärts, d.h. vom eingefügten Knoten zur Wurzel
  - $O(\lg n)$ Farbänderungen
  - ≤ 2 Rotationen
- Farbänderungen verändern Teilbaumgrößen nicht
- Rotationen dagegen schon



- Teilbaumgrößen von rotierten Knoten können in $O(1)$ neu berechnet werden werden

Löschen

- Zuerst wird ein Knoten y aus dem Baum entfernt oder verschoben
  - Direkt vorher: Wir folgen dem Pfad von y zur Wurzel und erniedrigen den Zähler jedes Knotens um 1
  - Falls verschoben: Wir berechnen $y.size$ aus Söhnen neu
  - $O(\lg n)$
- Danach werden die Rot-Schwarz-Eigenschaften wiederhergestellt
  - Nur Farbveränderungen und ≤ 3 Rotationen
  - Somit wieder: $O(1)$ Kosten für Wartung der Teilbaumgrößen

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen
5.1 Dynamische Ordnungsstatistiken
5.2 Anreicherung von Rot-Schwarz-Bäumen
5.3 Intervallbäume

6. Disjunkte Mengen

Schritte für Anreicherung

1. Wähle eine zugrundeliegende Datenstruktur
2. Bestimme zusätzlich benötigte Information
3. Prüfe dass zusätzliche Information für existierende Operationen gewartet werden kann
4. Entwickle die neuen Operationen

Anreicherung von Rot-Schwarz-Bäumen

Rot-Schwarz-Bäume eignen sich zur Anreicherung.

Theorem

Sei f ein neues Attribut in einem angereicherten Rot-Schwarz-Baum, sodass für jeden Knoten x der Wert von $x.f$ aus den Werten von x, $x.left$ und $x.right$ in $O(1)$ berechnet werden kann. Dann können die Werte von f in allen Knoten während des Einfügens und Löschens gewartet werden, ohne die asymptotische Laufzeit von $O(\lg n)$ zu verändern.

Beweisidee.

- Änderungen eines Knotens wirken sich nur auf dessen Vorfahren aus $\rightarrow$ können in $O(\lg n)$ nach oben propagiert werden
- Es werden während des Einfügens und Löschens konstant viele relevante Änderungen vorgenommen
  - Wir propagieren jede Änderung nach oben $\rightarrow$ jeweils $O(\lg n)$
  - Insgesamt immer noch $O(\lg n)$ □

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen
5.1 Dynamische Ordnungsstatistiken
5.2 Anreicherung von Rot-Schwarz-Bäumen
5.3 Intervallbäume

6. Disjunkte Mengen

Intervallbäume

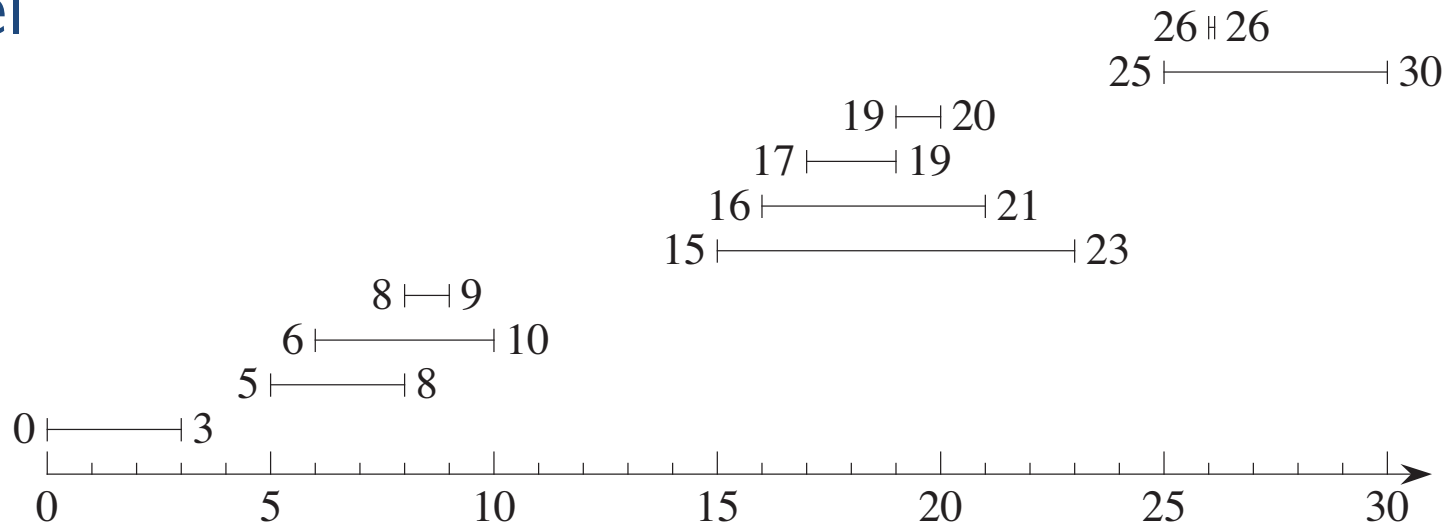
Ziel: Wartung einer dynamischen Menge von Intervallen.

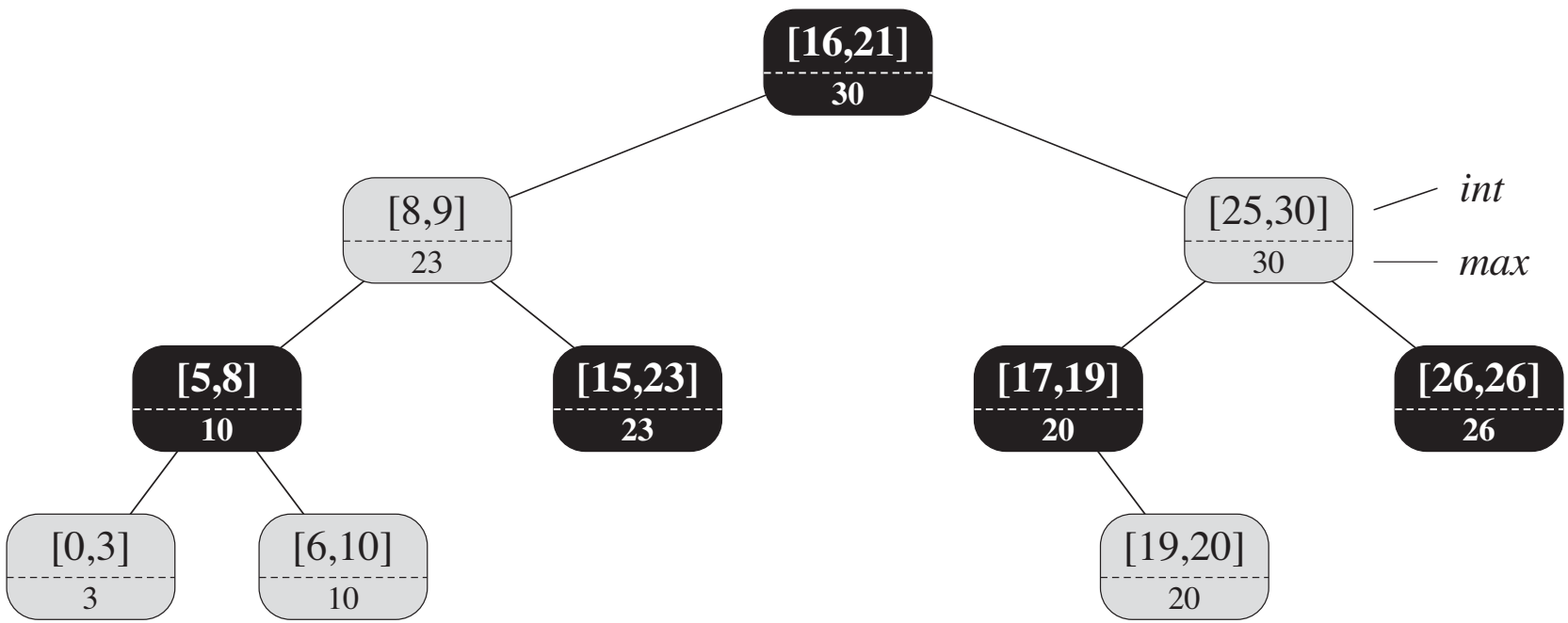
- z.B. Zeitintervalle
- Hier: geschlossene Intervalle $i = [t_1, t_2]$ der reellen Zahlen
- $i.low = t_1$, $i.high = t_2$
- Verallgemeinerung auf (halb-)offene Intervalle einfach

Operationen

- INTERVAL-INSERT(T, x)
- INTERVAL-DELETE(T, x)
- INTERVAL-SEARCH(T, i) liefert Zeiger auf ein Intervall, das mit i überlappt, oder NIL wenn kein solches Intervall existiert

Beispiel


26 26
25 30
19 20
17 19
16 21
15 23
8 9
6 10
5 8
0 3
0 5 10 15 20 25 30


[16,21]
30
[8,9]
23
[25,30]
30
int
max
[5,8]
10
[15,23]
23
[17,19]
20
[26,26]
26
[0,3]
3
[6,10]
10
[19,20]
20

Anreicherung

1. Rot-Schwarz-Baum.
   - Jeder Knoten x trägt Attribute $x.int$ für Intervall
   - Schlüssel ist $x.int.low$
2. Wir fügen hinzu:
   - $x.max$ = maximaler Wert eine Intervallendpunkts im Teilbaum mit Wurzel x
3. Wartung in $O(\lg n)$ ist möglich, da
   $x.max = \max(x.int.high, x.left.max, x.right.max)$
   $\rightarrow$ Theorem anwenden
4. Implementation
   - Interval-Insert und Interval-Delete (Übung)
   - Interval-Search wie nachfolgend

Intervallsuche (Pseudo-Code)

INTERVAL-SEARCH(T, i)

```
1: x ← T.root
2: while x ≠ NIL and i does not overlap x.int
3:     if x.left ≠ T.nil and x.left.max ≥ i.low
4:         x ← x.left
5:     else x ← x.right
6: return x
```

- $O(\lg n)$
- Warum ist dieser Algorithmus korrekt?

Korrektheit (1)

Interval-Search betrachtet höchstens ein Kind. Wir zeigen, dass wir keine "falsche" Wahl treffen.

Theorem

Wenn Suche nach rechts geht (Z. 5), dann

- *entweder Überlappung mit Intervall im rechten Teilbaum*
- *oder gar keine Überlappung in beiden Teilbäumen.*

Wenn Suche nach links geht (Z. 4), dann entweder

- *entweder Überlappung mit Intervall im linken Teilbaum*
- *oder gar keine Überlappung in beiden Teilbäumen.*

Einschub: Intervalltrichotomie

Je zwei Intervalle i und i' erfüllen genau eine der folgenden Eigenschaften:

a) i und i' überschneiden sich

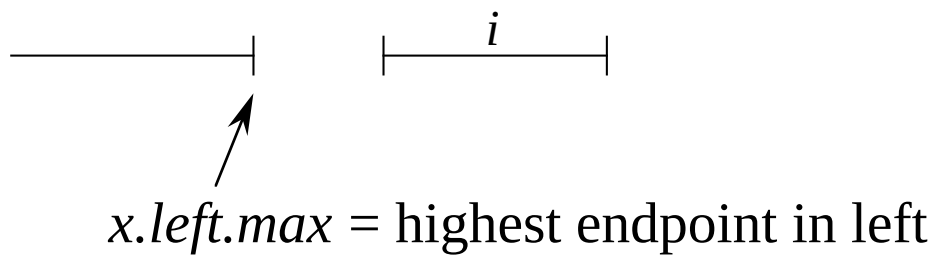
b) i liegt links von i' (d.h. $i.high < i'.low$)

c) i liegt rechts von i' (d.h. $i.low > i'.high$)

Korrektheit (2)

Fall 1: Suche geht nach rechts

- Überlappendes Intervall im rechten Teilbaum $\rightarrow$ OK
- z.Z.: Sonst gibt es auch keine Überlappung im linken Teilbaum
- Wir sind nach rechts gegangen weil
  - $x.left = T.nil \rightarrow$ OK
  - $x.left.max < i.low \rightarrow$ OK (Intervalltrichotomie)



Korrektheit (3)

Fall 2: Suche geht nach links

- Überlappendes Intervall im linken Teilbaum $\rightarrow$ OK
- z.Z.: Sonst gibt es auch keine Überlappung im rechten Teilbaum
- Wir sind nach links gegangen weil

$$i.low \leq x.left.max$$
$$= j.high \text{ für ein } j \text{ im linken Teilbaum}$$

- Da es links keine Überlappung gibt, überlappen i und j nicht

$$i.low > j.high \quad \text{oder} \quad i.high < j.low$$

- Da $i.low \leq j.high$, muss $i.high < j.low$ gelten
- Betrachte beliebiges Intervall k im rechten Teilbaum. Es gilt

$$j.low \leq k.low$$

- i und k überlappen nicht, denn

$$i.high < j.low \leq k.low$$

□

Outline

1. Elementare Datenstrukturen

2. Hash-Tabellen

3. Binäre Suchbäume

4. Rot-Schwarz-Bäume

5. Anreicherung von Datenstrukturen

6. Disjunkte Mengen

Datenstrukturen für disjunkte Mengen

- Anwendungen müssen manchmal Objekte in k disjunkte Mengen einteilen
  - z.B.: Graph partitioniert in Zusammenhangskomponenten
- Sind diese Mengen dynamisch, eignen sich **Datenstrukturen für disjunkte Mengen** (auch: *union-find*)
- Verwaltet einer Menge $\mathcal{S} = \{ S_1, \ldots, S_k \}$ von *dynamischen* disjunkten Mengen
- Identifiziert jede disjunkte Menge durch einen **Repräsentanten** (ein Element der Menge)

Operationen

- $\textsc{Make-Set}(x)$ erzeugt eine neue Menge mit x als einzigem Element und Repräsentanten
  - Da Mengen disjunkt, darf x noch nicht in einer der anderen Mengen sein
  - $S_i \leftarrow \{ x \}$; $\mathcal{S} \leftarrow \mathcal{S} \cup \{ S_i \}$
- $\textsc{Union}(x, y)$ vereinigt die durch x and y repräsentierten Mengen
  - x und y kommen aus verschiedenen Mengen
  - Wenn $x \in S_x$ und $y \in S_y$, dann werden S_x und S_y "zerstört" und durch eine neue Menge $S_{xy} = S_x \cup S_y$ ersetzt
  - $\mathcal{S} = \mathcal{S} \setminus \{ S_x \} \setminus \{ S_y \} \cup \{ S_{xy} \}$
  - Repräsentant wird beliebiges Element aus S_x oder S_y (oft: x oder y)
- $\textsc{Find-Set}(x)$ liefert Repräsentanten der Menge, die x enthält
  - Anforderung: Repräsentant darf sich nicht ändern, wenn keine Änderungsoperationen aufgerufen werden

Analyse

Wir analysieren die Laufzeit der Datenstrukturen (und der Algorithmen, die sie verwenden) in Abhängigkeit von 2 Parametern

- n: Anzahl der MAKE-SET-Operationen
- m Gesamtanzahl der Operationen
  - D.h. MAKE-SET, UNION und FIND-SET
- Bemerkungen
  - $m \geq n$
  - Höchstens $n - 1$ UNION-Operationen
  - Für Analyse nehmen wir oft an, dass die ersten n Operationen MAKE-SET-Operationen sind (hilfreich, aber nicht notwendig)

Zusammenhangskomponenten

Berechnung der Zusammenhangskomponenten für ungerichtete Graphen

CONNECTED-COMPONENTS(G)

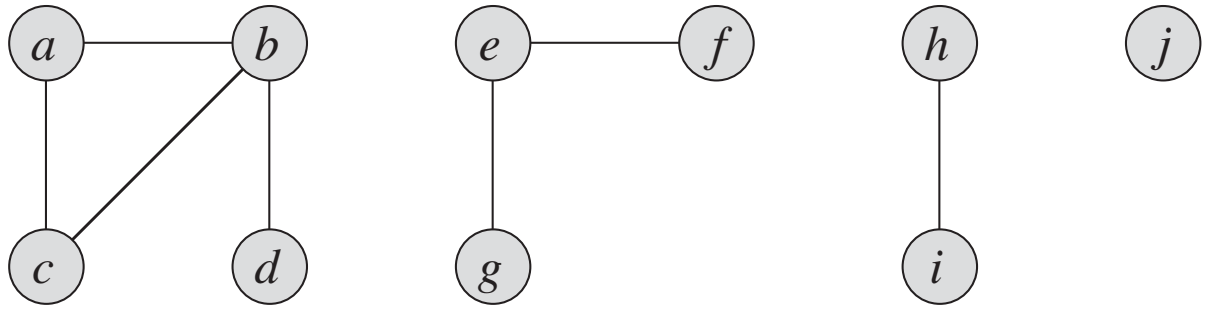
```
1: for each vertex v ∈ G.V
2:     MAKE-SET(v)
3: for each edge (u, v) ∈ G.E
4:     if FIND-SET(u) ≠ FIND-SET(v)
5:         UNION(u, v)
6:
```

SAME-COMPONENT(u, v)

```
1: if FIND-SET(u) = FIND-SET(v)
2:     return TRUE
3: else return FALSE
```

(Für statische Graphen schneller mit Tiefensuche)

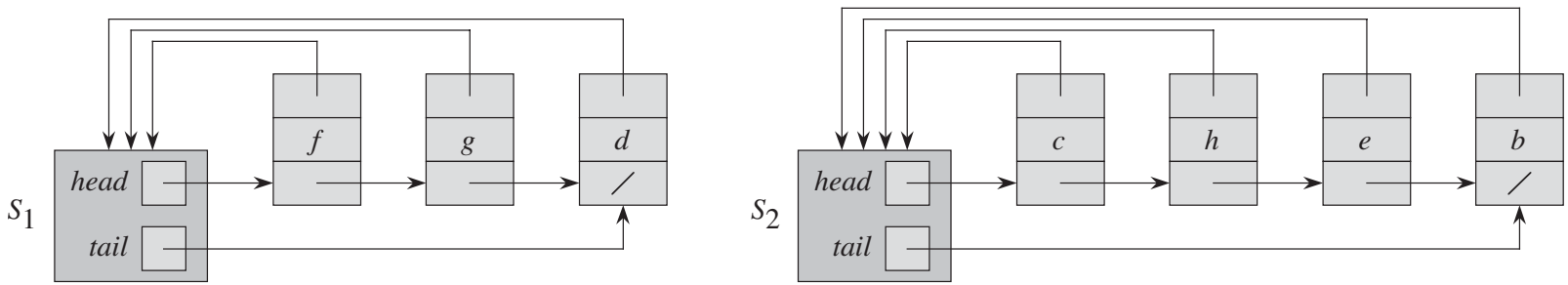
Beispiel



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Umsetzung mit verketteten Listen (1)

- Jede Menge S_i wird durch doppelt-verkettete Liste repräsentiert
- Kopf der Liste (*head*) ist Repräsentant
- Jedes Element hat Zeiger auf seine Liste



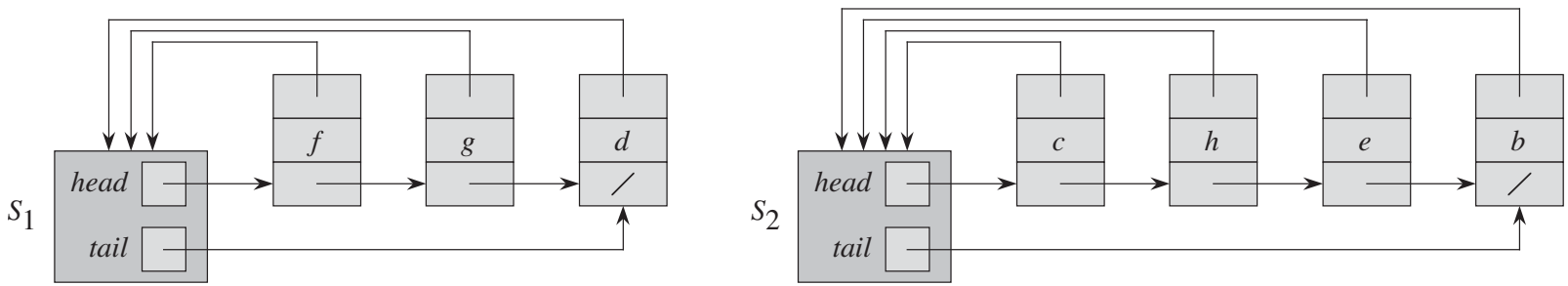
Operationen

- MAKE-SET: Liste anlegen
- FIND-SET: Zeiger zum Listenobjekt folgen, Kopf zurückliefern
- UNION: Mehrere Möglichkeiten

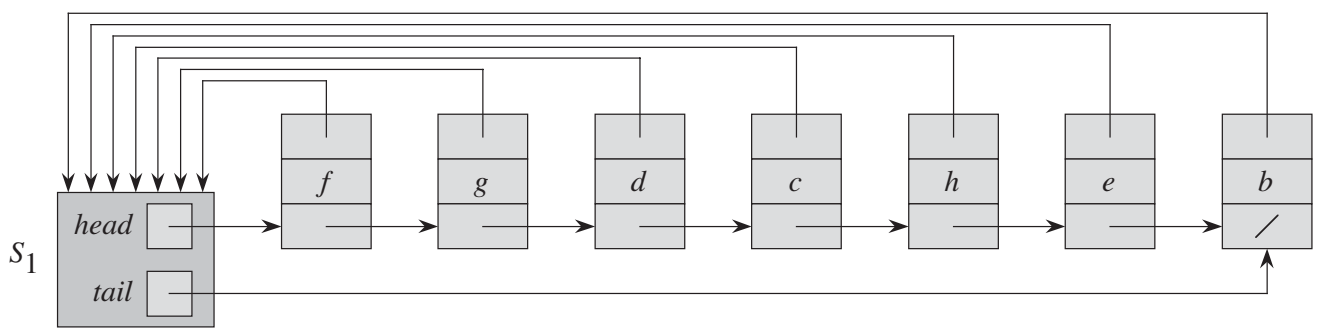
Umsetzung mit verketteten Listen (2)

Einfache Variante

- Liste von y an Ende der Liste von x anhängen
- Zeiger auf Listenobjekt für Elemente der (ursprünglichen) Liste von y anpassen



Union(f, e)



Analyse

- Im ungünstigsten Fall ist Liste von y lang und Liste von x kurz
- Dann: Anpassen der Zeiger teuer

Operation	Anzahl aktualisierter Objekte
$\textsc{Make-Set}(x_1)$	1
$\textsc{Make-Set}(x_2)$	1
$\ldots$	1
$\textsc{Make-Set}(x_n)$	1
$\textsc{Union}(x_2, x_1)$	1
$\textsc{Union}(x_3, x_2)$	2
$\textsc{Union}(x_4, x_3)$	3
$\ldots$	$\ldots$
$\textsc{Union}(x_n, x_{n-1})$	$n-1$
	$\Theta(n^2)$

- Das sind $2n-1$ Operationen
- Somit: $\Theta(n)$ pro Operation (= **amortisierte Laufzeit**)

Die Weighted-Union-Heuristik (1)

- **Weighted-Union-Heuristik**: Immer kürzere Liste an längere Liste anhängen
- Dadurch müssen weniger Zeiger angepasst werden
- Listen müssen dazu ihre Länge verwalten

Theorem

Mit der **Weighted-Union-Heuristik** *benötigt eine Sequenz von m Operationen auf n Elementen eine $O(m + n \lg n)$ Zeit.*

Die Weighted-Union-Heuristik (2)

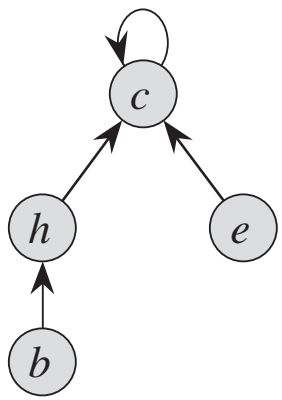
Beweis.

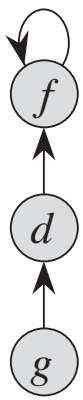
- MAKE-SET und FIND-SET jeweils $O(1)$, insgesamt $O(m)$
- Es gibt höchstens $n - 1$ UNION-Operationen
- Wie oft wird der Listenzeiger eines Elements x höchstens aktualisiert?
  - x ist immer in der kleineren Liste $\rightarrow$ Wie groß ist diese?
  - Bei erster Aktualisierung: ≥ 1 Element
  - Bei zweiter Aktualisierung: ≥ 2 Elemente
  - Bei dritter Aktualisierung: ≥ 4 Elemente
  - ...
- Somit gilt für $k \leq n$: Ist x in einer Menge mit k Elementen, wurde der Zeiger von x höchstens $\lceil \lg k \rceil$ mal geändert
- Da die größte Menge höchstens n Elemente hat, ergibt sich $O(n \lg n)$ für alle UNION-Operationen
- Die Gesamtlaufzeit ist also $O(m + n \lg n)$ □

Es geht aber noch besser!

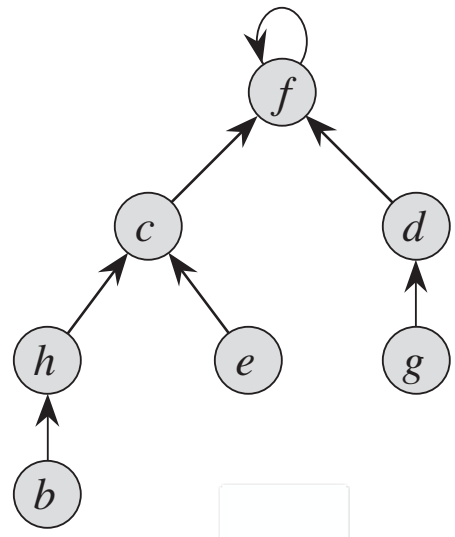
Umsetzung mit einem Wald

- Schneller: Wald disjunkter Bäume
  - Wurzel ist Repräsentant
  - Jeder Knoten besitzt nur einen Zeiger auf seinen Vaterknoten
  - Wurzelknoten zeigt auch sich selbst





Union



- MAKE-SET: Baum mit nur einem Knoten
- FIND-SET: Pfad zur Wurzel folgen
- UNION: Vater einer Wurzel auf andere ändern
- Effizient durch zwei gute Heuristiken

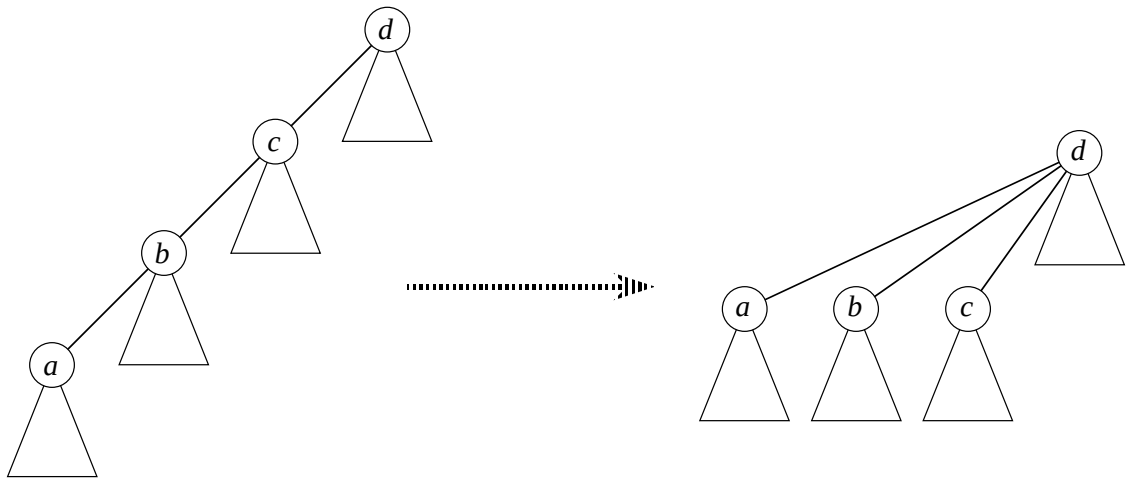
Heuristiken

Union by Rank

- Idee: Kleinerer Baum wird angehängt
- Statt Baumgröße verwenden wir den Rang des Wurzelknotens
  - Rang ist obere Schranke für Höhe (und Kosten von FIND-SET)
  - Rang wird inkrementell gewartet

Path Compression

- Während FIND-SET werden alle Zeiger auf dem Pfad zur Wurzel auf die Wurzel selbst gesetzt



Pseudo-Code

MAKE-SET(x)

```
1: x.p ← x
2: x.rank ← 0
```

FIND-SET(x)

```
1: if x ≠ x.p
2:     x.p ← FIND-SET(x.p)
3: return x.p
```

UNION(x, y)

```
1: LINK(FIND-SET(x),
        FIND-SET(y))
```

LINK(x, y)

```
1: if x.rank > y.rank
2:     y.p ← x
3: else
4:     x.p ← y
5:     if x.rank = y.rank
6:         y.rank = y.rank + 1
```

Analyse

- Ohne Heuristiken: $\Theta(m^2)$
- Nur mit Union by Rank: $O(m \lg n)$
- Nur mit Path Compression für f FIND-SET-Operationen: $O(n + f(1 + \log_{(2+f/n)} n))$
- Beide Heuristiken: $O(m\alpha(n))$
  - α ist die **inverse Ackermannfunktion** und wächst sehr langsam
  - In jeder vorstellbaren Anwendung ist $\alpha(n) \leq 5$

Ackermannfunktion

Die **Ackermannfunktion** ist gegeben durch

$$A(k,i) = \begin{cases} i+1 & \text{wenn } k = 0 \\ A(k-1,1) & \text{wenn } k > 0 \text{ und } i = 0 \\ A(k-1, A(k,i-1)) & \text{wenn } k > 0 \text{ und } i > 0. \end{cases}$$

- Sehr schnell wachsend (monoton steigend in k und i)

$A(k,i)$	$i=0$	$i=1$	$i=2$	$i=3$
$k=0$	1	2	3	4
$k=1$	2	3	4	5
$k=2$	3	5	7	9
$k=3$	5	13	29	61
$k=4$	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$
$k=5$	65533	$\boldsymbol{A(4,65533)}$	$A(4,A(4,65533))$	$A(4,A(5,2))$

- $\alpha(n) = \min\{\, k \mid A(k,1) \geq n \,\}$
- Für $n < A(4,65533)$ ist $\alpha(n) \leq 5$

Outline

1. Elementare Datenstrukturen
2. Hash-Tabellen
3. Binäre Suchbäume
4. Rot-Schwarz-Bäume
5. Anreicherung von Datenstrukturen
6. Disjunkte Mengen

Zum Nachlesen

- CLRS, Ch. 10, *Elementary Data Structures*
- CLRS, Ch. 11, *Hash Tables*
- CLRS, Ch. 12, *Binary Search Trees*
- CLRS, Ch. 13, *Red-Black Trees*
- CLRS, Ch. 14, *Augmenting Data Structures*
- CLRS, Ch. 21, *Data Structures for Disjoint Sets*

Algorithmen und Datenstrukturen

04 – Graphalgorithmen

Prof. Dr. Rainer Gemulla

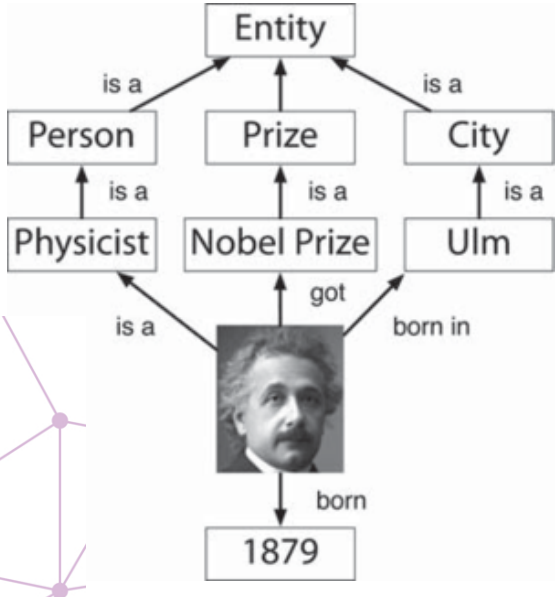
Universität Mannheim

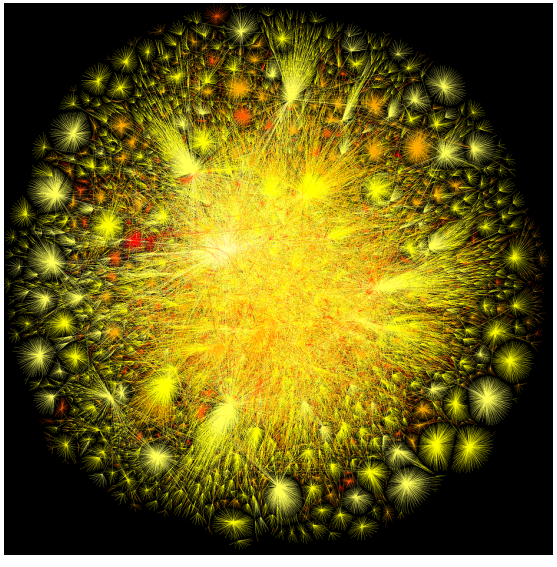
HWS 2014

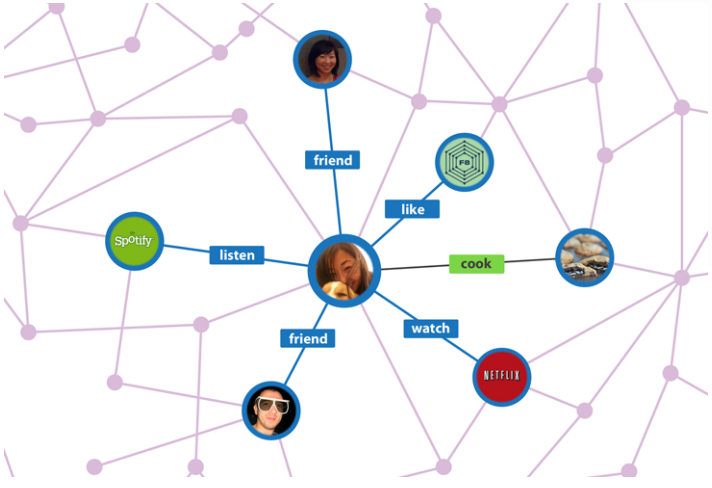
Graphen überall

Graphen tauchen überall in der Informatik auf.

- Straßennetze
- Kommunikationsnetzwerke
- Linkstruktur des World Wide Web
- Soziale Netzwerke
- Wissensbasen
- Linguistik
- Berechnungen
- ...







Outline

1. Elementare Graphalgorithmen

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Outline

1. Elementare Graphalgorithmen
1.1 Repräsentation
1.2 Breitensuche
1.3 Tiefensuche
1.4 Topologisches Sortieren
1.5 Starke Zusammenhangskomponenten

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Outline

1. Elementare Graphalgorithmen
1.1 Repräsentation
1.2 Breitensuche
1.3 Tiefensuche
1.4 Topologisches Sortieren
1.5 Starke Zusammenhangskomponenten

2. Minimale Spannbäume

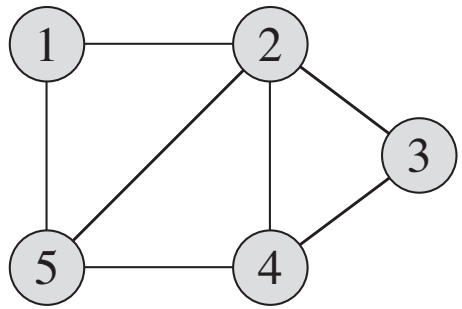
3. Kürzeste Wege

4. Alle kürzesten Wege

Vereinbarungen

Sei $G = (V, E)$ ein Graph mit Knotenmenge V und Kantenmenge E.

- In Algorithmen betrachten wir V und E als Attribute des Graphen, d.h. wir schreiben $G.V$ und $G.E$
- Komplexität von Algorithmen hängt sowohl von $|V|$ als auch $|E|$ ab $\rightarrow$ Zwei Parameter
- Innerhalb asymptotischer Notationen (und nur dort) kürzen wir $|V|$ und $|E|$ als V und E ab
  - Also ist $O(VE)$ die Abkürzung von $O(|V||E|)$



Repräsentationen von Graphen

- Graph kann entweder gerichtet oder ungerichtet sein
- Zwei gebräuchliche Möglichkeiten
  - Adjazenzlisten
  - Adjazenzmatrizen
- Repräsentation beeinflusst Speicherbedarf und Operationen
- Adjazenzlisten oft bevorzugt, insbesondere bei **dünn besetzten** (*sparse*) Graphen, d.h. solche, bei denen $|E|$ viel kleiner als $|V|^2$ ist
- Für **dichte** Graphen auch öfter Adjazenzmatrizen

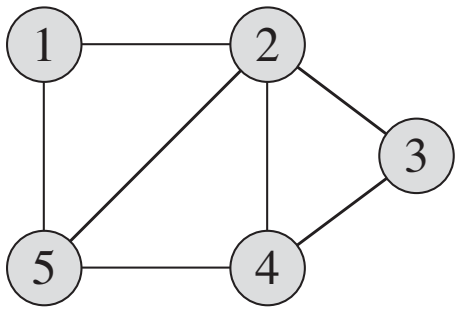
Adjazenzlisten

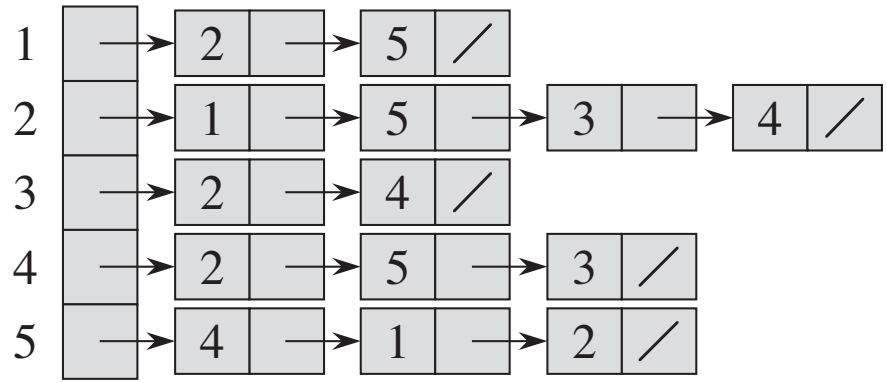
Die **Adjazenzlistenrepräsentation** eines Graphen $G = (V, E)$ besteht aus einem Array Adj von $|V|$ Listen.

- Eine Liste pro Knoten
- Liste von $u \in V$ enthält diejenigen Knoten $v \in V$ für die $(u, v) \in E$
- Im Pseudo-Code schreiben wir $G.Adj$ bzw. $G.Adj[u]$
- Unterstützt gerichtete und ungerichtete Graphen
  - Für gerichtete Graphen ist die Summe aller Listenlängen gleich $|E|$
  - Für ungerichtete Graphen gleich $2|E|$
- Unterstützt gewichtete Graphen
  - Typischerweise durch **Gewichtsfunktion** $w : E \to \mathbb{R}$ gegeben
  - Gewicht $w(u, v)$ wird mit v in der Adjazenzliste von u gespeichert
  - Ähnlich können andere Kantenattribute gespeichert werden
- Speicherplatzbedarf: $O(max(V, E)) = O(V + E)$
- Test ob $(u, v) \in E$: $O(\text{degree}(u))$
- Alle Nachbarn von u bestimmen: $\Theta(\text{degree}(u))$

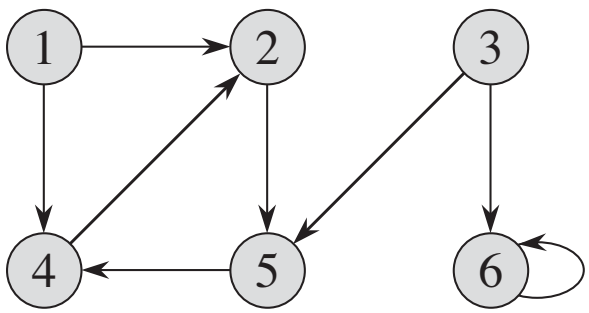
Beispiel

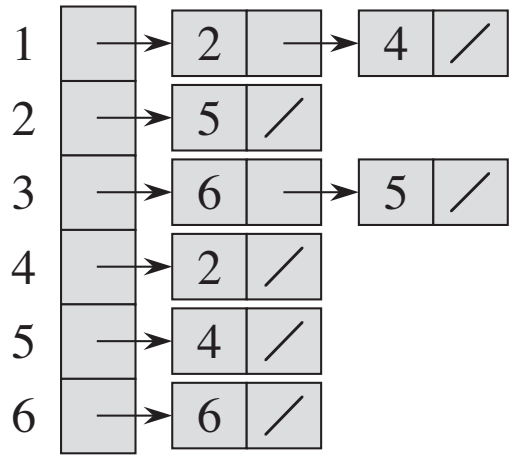
Ungerichteter Graph





Gerichteter Graph





Adjazenzmatrizen

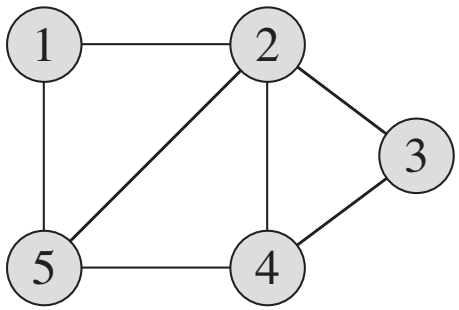
Sei $G = (V, E)$ ein Graph mit $V = \{1, 2, \ldots, |V|\}$. Die **Adjazenzmatrix** von G ist eine $|V| \times |V|$-Matrix $\boldsymbol{A}$ mit

$$a_{ij} = \begin{cases} 1 & \text{für } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

- Ist $V \neq \{1, 2, \ldots, |V|\}$, nummerieren wir die Knoten beliebig
- Unterstützt gerichtete und ungerichtete Graphen
  - Für ungerichtete Graphen ist $\boldsymbol{A}$ symmetrisch ($a_{ij} = a_{ji}$, $\boldsymbol{A} = \boldsymbol{A}^T$)
    $\rightarrow$ Speicherplatz kann fast halbiert werden, indem man nur Diagonale und Einträge überhalb speichert
- Unterstützt gewichtete Graphen
  - Gewicht $w(i,j)$ wird in a_{ij} abgelegt
  - Spezieller Marker für nicht existierende Kanten (oft: 0, ∞ oder NIL)
- Speicherplatzbedarf: $\Theta(V^2)$
- Test ob $(u, v) \in E$: $\Theta(1)$
- Alle Nachbarn von u bestimmen: $\Theta(V)$

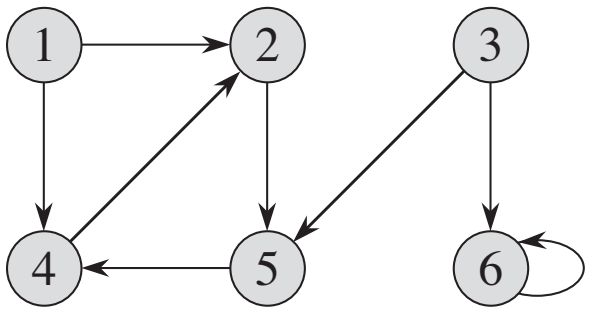
Beispiel

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

Gerichteter Graph



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

Attribute

Es ist oft notwendig, Attribute zu Knoten oder Kanten zu speichern.

- Wir schreiben $v.d$ für Attribut d des Knotens v
- Und $(u, v).f$ für Attribute f von Kante (u, v)
- Kann verschiedenen implementiert werden, z.B.
  - In objektorientierter Sprache: Kanten und Knoten als Klassen
  - Parallele Arrays ($d[u]$ enthält Wert $u.d$)

Outline

1. Elementare Graphalgorithmen
1.1 Repräsentation
1.2 Breitensuche
1.3 Tiefensuche
1.4 Topologisches Sortieren
1.5 Starke Zusammenhangskomponenten

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Breitensuche

Sei $G = (V, E)$ ein Graph und $s \in V$ ein ausgewählter **Startknoten** (*source vertex*). Die **Breitensuche** ist ein fundamentaler Algorithmus zur Suche in Graphen.

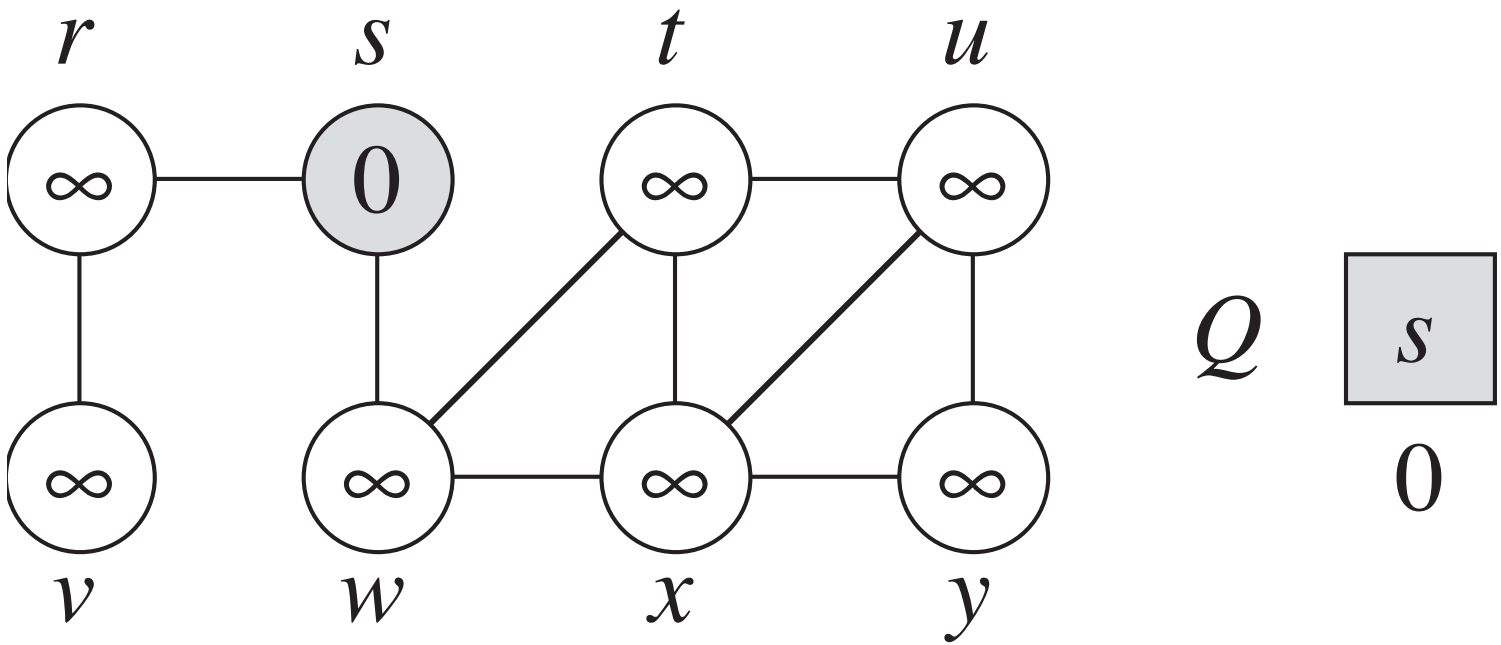
- Besucht systematisch alle von s aus erreichbaren Knoten
- Berechnet die *Distanz* zu jedem erreichbaren Knoten (d.h. kleinste Anzahl Kanten)
- Berechnet einen **Breitensuchebaum**
  - Wurzel s
  - Enthält alle erreichbaren Knoten
  - Für jeden Knoten v, einfacher Pfad von s nach v in Breitensuchebaum entspricht **kürzestem Weg** von s nach v in G

Idee

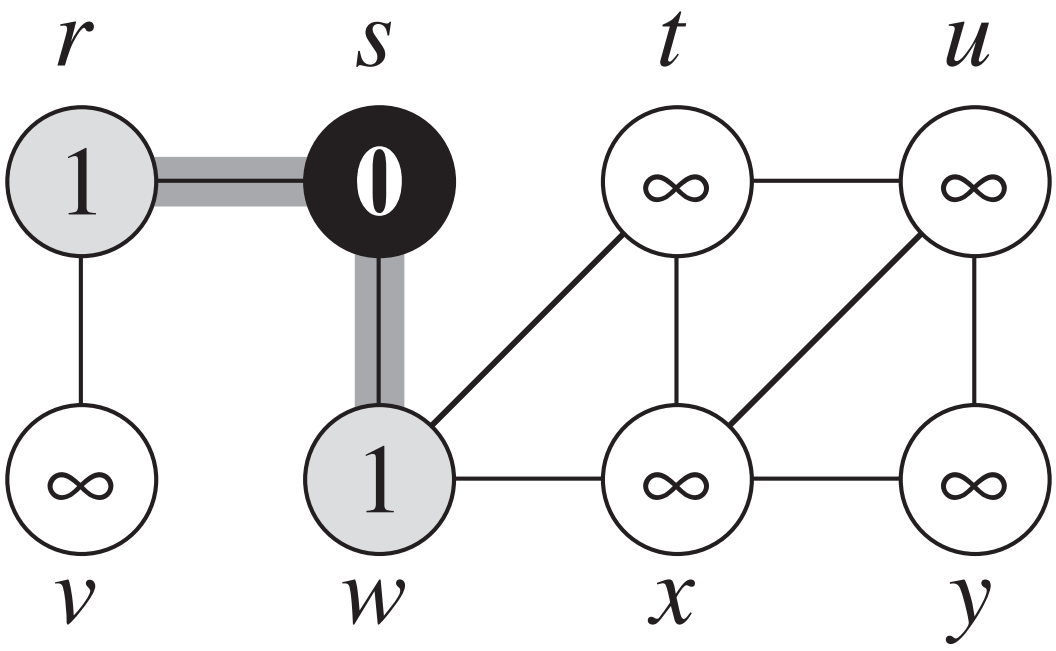
- Graph wird ausgehend von s traversiert
- Maxime: Besuche erst alle Söhne, bevor du einen Enkel besuchst
- Dazu: Knoten werden gefärbt
  - Weiß = unbesucht
  - Grau = besucht, aber Söhne noch nicht
  - Schwarz = vollständig abgearbeitet
- Knotenattribute
  - *color*
  - d (Distanz von s)
  - π (Vorgänger / Vater in Breitensuchebaum)
- Graue Knoten werden außerdem in einer Warteschlange Q abgelegt
  - FIFO (*first in first out*)

```
BFS(G, s)
 1: for each vertex u ∈ G.V − {s}
 2:     u.color ← WHITE
 3:     u.d ← ∞
 4:     u.π ← NIL
 5: s.color ← GRAY
 6: s.d ← 0
 7: s.π ← NIL
 8: Q ← ∅
 9: ENQUEUE(Q, s)
10: while Q ≠ ∅
11:     u ← DEQUEUE(Q)
12:     for each vertex v ∈ G.Adj[u]
13:         if v.color = WHITE
14:             v.color ← GRAY
15:             v.d = u.d + 1
16:             v.π = u
17:             ENQUEUE(Q, v)
18:     u.color = BLACK
```

Beispiel (1)



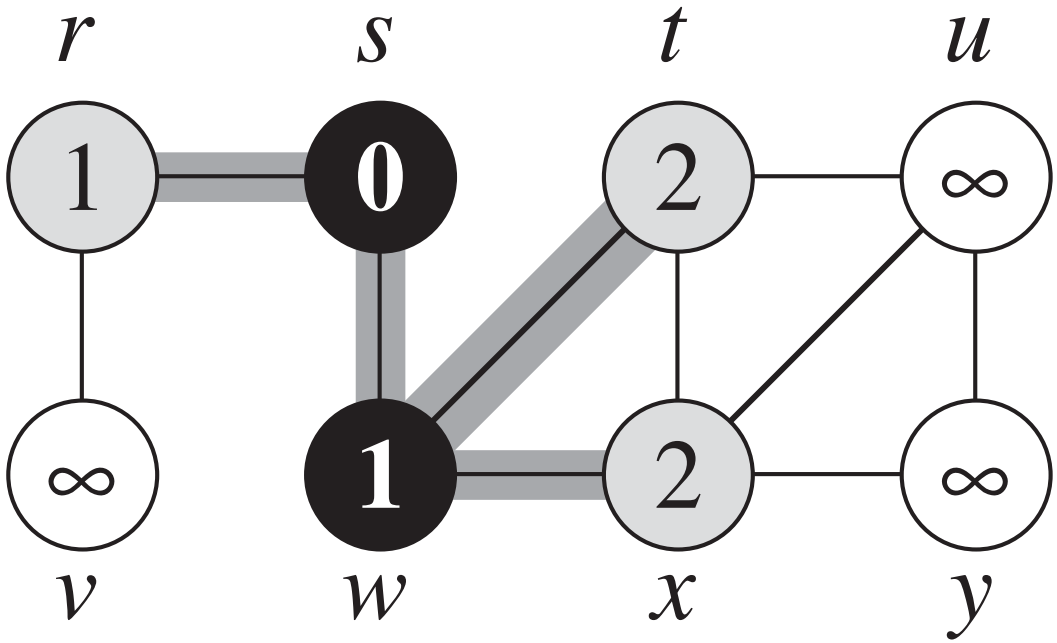
Beispiel (2)



Q

w	r
1	1

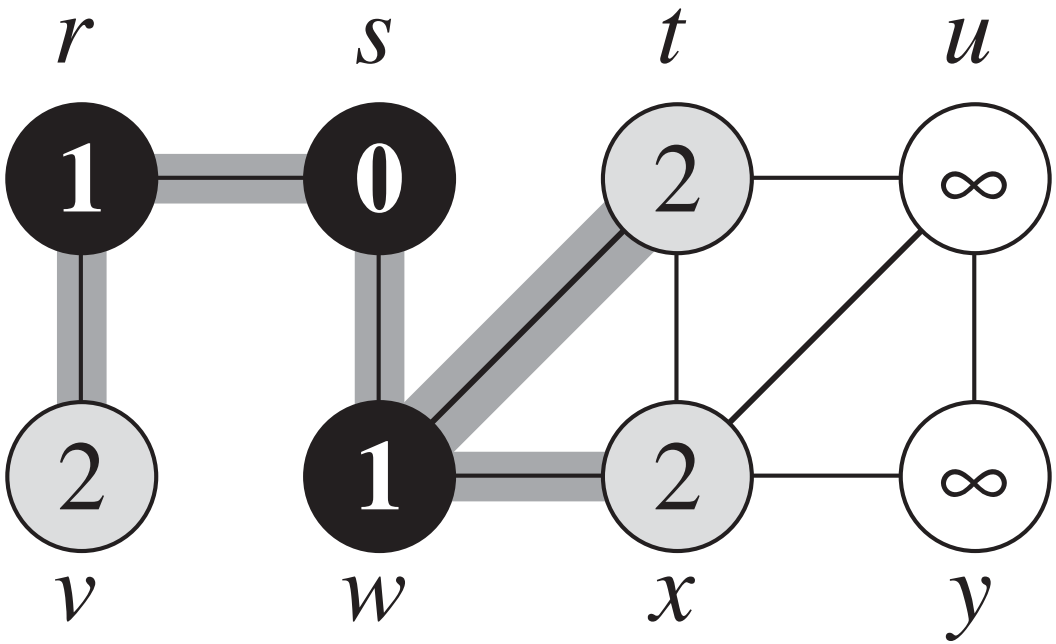
Beispiel (3)



Q

r	t	x
1	2	2

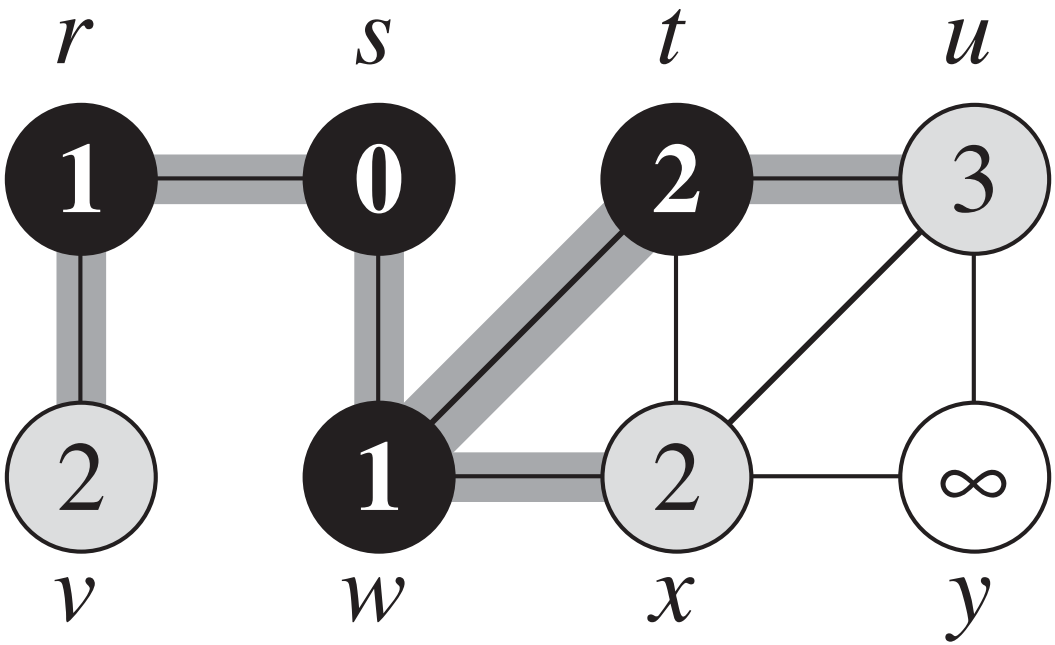
Beispiel (4)



Q

t	x	v
2	2	2

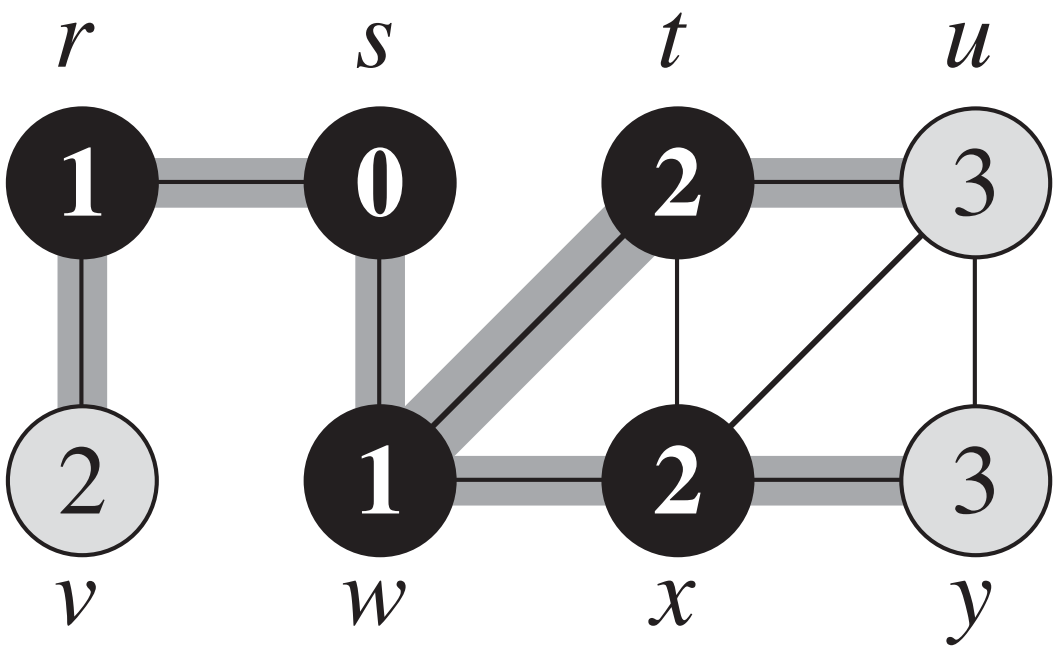
Beispiel (5)



Q

x	v	u
2	2	3

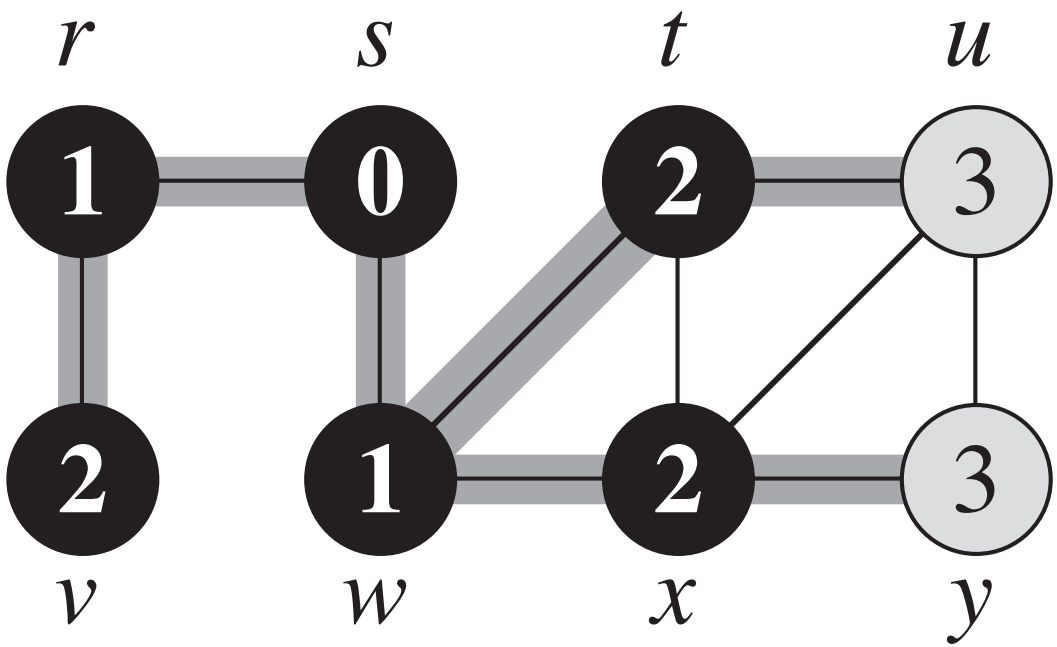
Beispiel (6)



Q

v	u	y
2	3	3

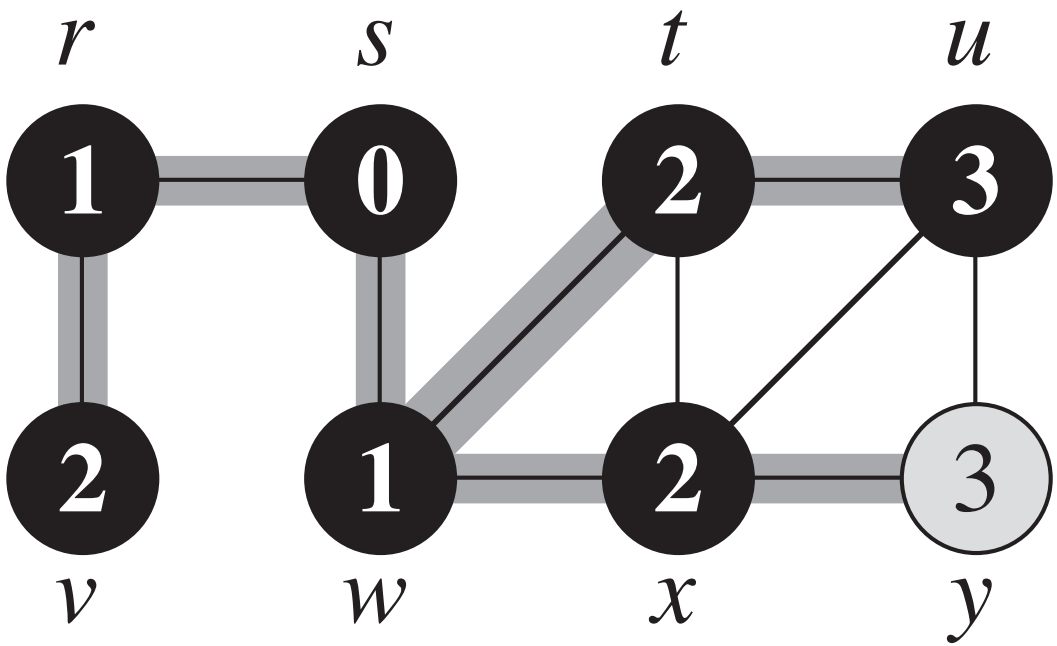
Beispiel (7)

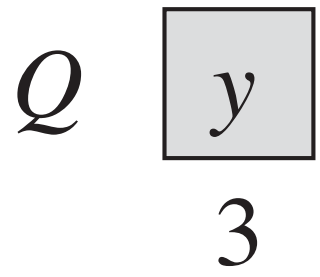


Q

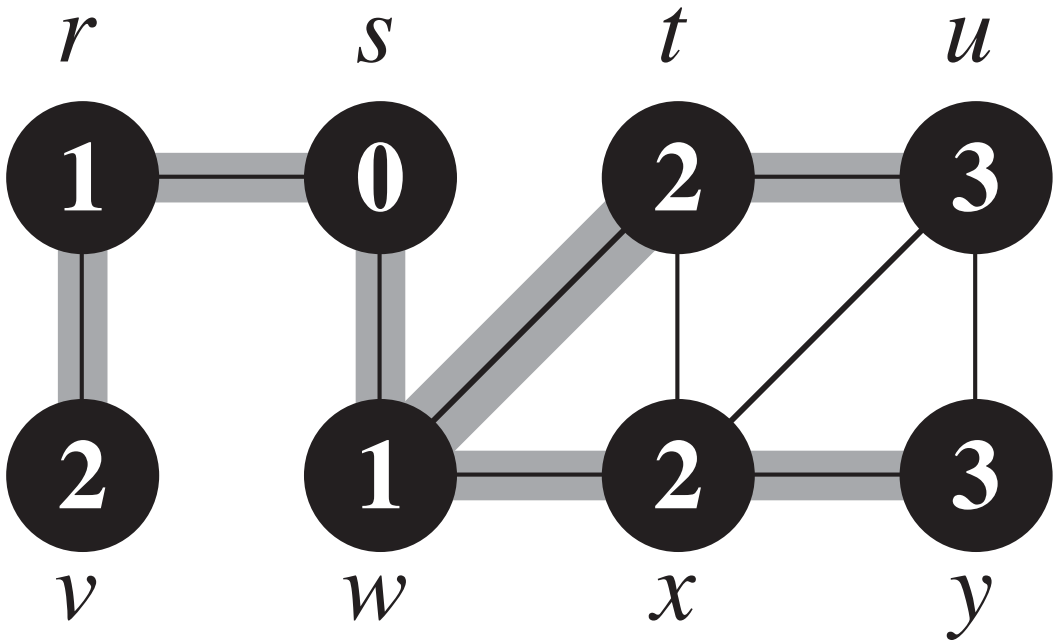
u	y
3	3

Beispiel (8)





Beispiel (9)



$Q \quad \emptyset$

Analyse der Laufzeit

- Weiß färben (Z. 1–4): $O(V)$
- Warteschlangenoperationen: $O(V)$
  - Nur weiße Knoten werden eingefügt (Z. 13)
  - Und dabei grau gefärbt (Z. 14)
  - Somit: Jeder Knoten nur einmal eingefügt (und entfernt)
- Adjazenzlisten ablaufen (Z. 12): $O(E)$
  - Eine Adjazenzliste wird nur dann abgelaufen, wenn u aus der Warteschlange genommen wird (DEQUEUE)
- Gesamt: $O(V + E)$

Distanz und kürzeste Pfade

Definition

Die **Distanz** (*shortest-path distance*) $\delta(s, v)$ zwischen zwei Knoten s und v ist die minimale Anzahl Kanten in einem Pfad von s nach v (∞ falls kein Pfad existiert). Pfade von s nach v mit Länge $\delta(s, v)$ nennen wir **kürzeste Pfade** (*shortest paths*).

Wir wollen zeigen: $v.d = \delta(s, v)$.

Eigenschaften (1)

Lemma (F28)

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und $s \in V$. Dann gilt für alle $(u, v) \in E$:

$$\delta(s, v) \leq \delta(s, u) + 1$$

Beweis.

- Wenn u nicht von s erreichbar, $\delta(s, u) = \infty \rightarrow$ OK
- Sonst: Kürzester Pfad von s nach v kann nicht länger sein als kürzester Pfad von s nach u, gefolgt von Kante (u, v) □

Eigenschaften (2)

Lemma (F29)

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Nach der Anwendung von Bfs *auf G für ein $s \in V$ gilt für alle $v \in V$:*

$$v.d \geq \delta(s, v).$$

Somit ist $v.d$ zumindest nicht kleiner als Distanz.

Beweis. Per Induktion über die Anzahl der Enqueue-Aufrufe

- Induktionsanfang (nach Z. 9)
  - Gilt für s da $s.d = 0 \geq \delta(s, s)$
  - Gilt für alle $v \in V - \{ s \}$, da $v.d = \infty \geq \delta(s, v)$

Eigenschaften (3)

- Induktionsschritt
  - Betrachte weißen Knoten v in $Adj[u]$
  - Nach Induktionsvorraussetzung: $u.d \geq \delta(s, u)$
  - Wir setzen

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) \qquad \text{(nach Lemma F28)} \end{aligned}$$

  - Somit gilt Induktionshypothese für v wenn eingefügt (Z. 17)
  - $v.d$ wird nie wieder verändert (da v nicht mehr weiß) □

Eigenschaften (4)

Um zu zeigen, dass $v.d = \delta(s, v)$ gilt, müssen wir uns genauer anschauen, wie BFS mit Q umgeht. Das nächste Lemma zeigt, dass zu jedem Zeitpunkt höchstens 2 verschiedene Werte für d in Q sind.

Lemma (F31)

Q enthalte während der Ausführung von BFS auf $G = (V, E)$ die Knoten $\langle v_1, \ldots, v_r \rangle$. Dann gilt:

$$\begin{aligned} v_r.d &\leq v_1.d + 1, \\ v_i.d &\leq v_{i+1}.d \qquad \text{für alle } 1 \leq i \leq r-1. \end{aligned}$$

Eigenschaften (5)

Beweis. Per Induktion über die Anzahl der Warteschlangenoperationen

- Induktionsanfang (nach Z. 9)
  - $Q = \langle s \rangle \rightarrow$ OK
- Induktionsschritt: Aufruf von DEQUEUE (Z. 11)
  - $u = v_1$ wird entfernt, v_2 wird neuer Kopf
  1. Es gilt $v_r.d \leq v_1.d + 1 \leq v_2.d + 1 \rightarrow$ OK
  2. Unverändert
- Anschließende Aufrufe von ENQUEUE (Z. 17)
  - $v = v_{r+1}$ wird angehangen, Kopf ist v_2
  1. Es gilt $v_{r+1}.d = v.d = u.d + 1 = v_1.d + 1 \leq v_2.d + 1$
  2. $v_r.d \leq v_1.d + 1 = u.d + 1 = v.d = v_{r+1}.d$ □

Korrektheit von BFS (1)

Theorem

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Dann besucht $\textsc{Bfs}(G, s)$, $s \in V$, jeden von s aus erreichbaren Knoten v. Nach Terminierung gilt:

- *$v.d = \delta(s, v)$ für alle $v \in V$,*
- *Für jeden Knoten $v \neq s$, der von s aus erreichbar ist, gilt: ein kürzester Pfad von s nach v ist ein kürzester Pfad von s zu $v.\pi$ gefolgt von der Kante $(v.\pi, v)$.*

Korrektheit von BFS (2)

Beweis.

Fall 1: v nicht von s erreichbar

- $v.d \geq \delta(s, v) = \infty$ nach Lemma F29
- Somit kann $v.d$ in Z. 15 nicht auf einen endlichen Wert gesetzt worden sein
- Somit wurde v nicht entdeckt

Fall 2: v von s erreichbar

- Annahme für Widerspruch: Ein Knoten hat einen falschen d-Wert
- Wähle einen kleinsten solchen Knoten, d.h. $v.d \neq \delta(s, v)$ und $\delta(s, v)$ minimal
- Es gilt: $v \neq s$
- $v.d \geq \delta(s, v)$ nach Lemma F29
- Da $v.d$ inkorrekt, $v.d > \delta(s, v)$

Korrektheit von BFS (3)

- Sei u ein Knoten direkt vor v auf einem kürzesten Pfad von s nach v, somit $\delta(s,u) < \delta(s,v)$
- Durch unserere Wahl von v: $u.d = \delta(s,u)$
- Zusammen

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

- Irgendwann wird u in Z. 11 von Q entfernt
  - Wenn v dann weiß, setzt Z. 15 $v.d = u.d + 1 \rightarrow$ Widerspruch
  - Wenn v dann schwarz, dann schon von Q entfernt und $v.d \leq u.d$ $\rightarrow$ Widerspruch
  - Wenn v dann grau, dann wurde es bei Abarbeitung eines Knoten w so gefärbt. Es gilt $w.d \leq u.d$ und $v.d = w.d + 1 \leq u.d + 1$ $\rightarrow$ Widerspruch
- Somit $v.d = \delta(s,v)$ für alle v
- Wenn $v.\pi = u$, dann $\delta(s,v) = v.d = u.d + 1 = \delta(s,u) + 1$ $\square$

Vorgängerteilgraph und Breitensuchebaum

Definition

Für einen Graphen $G = (V, E)$ mit Wurzel s definieren wir den **Vorgängerteilgraphen** (*predecessor subtree*) von G als $G_\pi = (V_\pi, E_\pi)$ mit

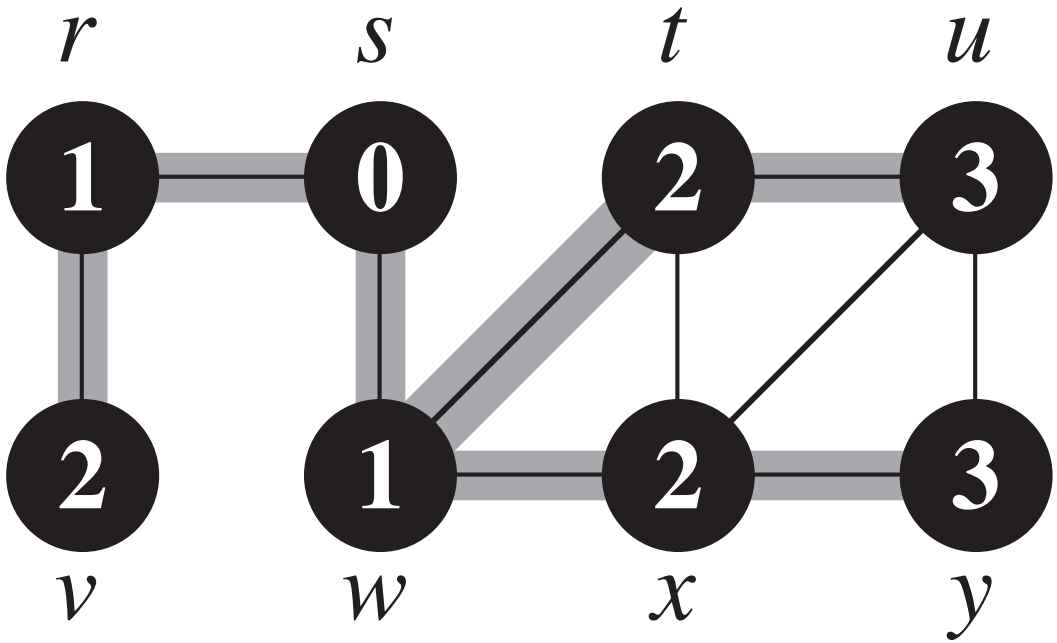
$$V_\pi = \{ v \in V \mid v.\pi \neq \text{NIL} \} \cup \{ s \},$$
$$E_\pi = \{ (v.\pi, v) \in E \mid v \in V_\pi \setminus \{ s \} \}.$$

Definition

G_π ist ein **Breitensuchebaum**, falls

1. V_π besteht aus allen von s aus erreichbaren Knoten
2. Für alle $v \in V_\pi$ existiert ein eindeutiger einfacher Pfad von s nach v in G_π; dieser ist kürzester Pfad von s nach v in G

Beispiel



$Q \quad \emptyset$

Breitensuchebaum und BFS

Lemma

Es werde BFS auf einen gerichteten oder ungerichteten Graphen $G = (V, E)$ angewendet. Nach Terminierung hat BFS die π-Attribute so belegt, dass G_π ein Breitensuchebaum ist.

Beweis.

1. Durch Korrektheit BFS
2. G_π ist ein Baum, da verbunden und $|E_\pi| = |V_\pi| - 1$
   - Somit eindeutiger einfacher Pfad
   - Korrektheit BFS ergibt Minimalität dieses Pfads

Ausgeben eines kürzesten Pfades

PRINT-PATH(G, s, v)

```
1: if v = s
2:     Print s
3: else if v.π = NIL
4:     Print "no path from s to v exisits"
5: else
6:     PRINT-PATH(G, s, v.π)
7:     Print v
```

- Laufzeit: $O(\delta(s, v))$

Outline

1. Elementare Graphalgorithmen
1.1 Repräsentation
1.2 Breitensuche
1.3 Tiefensuche
1.4 Topologisches Sortieren
1.5 Starke Zusammenhangskomponenten

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Tiefensuche

- **Tiefensuche** ist eine Suchstrategie, in der die Kanten eines entdeckten Knotens v sofort exploriert werden
  $\rightarrow$ Suche in die Tiefe
- Ist dies abgeschlossen, springt die Suche zurück (*backtracking*) zu dem Knoten von dem aus v gefunden wurde
- Im Gegensatz zur Breitensuche, kein expliziter Startknoten
  - Jede Kante wird exploriert
  - Suche beginnt bei einem der Knoten
  - Falls notwendig, werden weitere Startknoten gewählt
- Berechnet Zeitstempel und Vorgängergraph

Knotenattribute

- Färbung
  - Weiß = unbesucht
  - Grau = entdeckt, aber noch nicht vollständig abgearbeitet
  - Schwarz = abgearbeitet (alle erreichbaren Knoten gefunden)
- Knotenattribute
  - $color$, π
  - d = **Zeit der Entdeckung** (*discovery time*)
  - f = **Zeit der Abarbeitung** (*finishing time*)
- Zeitstempel
  - Nützlich für Graphalgorithmen sowie Analyse der Tiefensuche
  - Jeder Knoten wird einmal entdeckt und einmal abgearbeitet
    $\rightarrow$ Zeitstempel eindeutig zwischen 1 und $2|V|$
  - Es gilt $v.d < v.f$ für alle $v \in V$

Pseudo-Code (1)

DFS(G)

```
1: for each vertex u ∈ G.V
2:     u.color ← WHITE
3:     u.π ← NIL
4: time ← 0
5: for each vertex u ∈ G.V
6:     if u.color = WHITE
7:         DFS-VISIT(G, u)
```

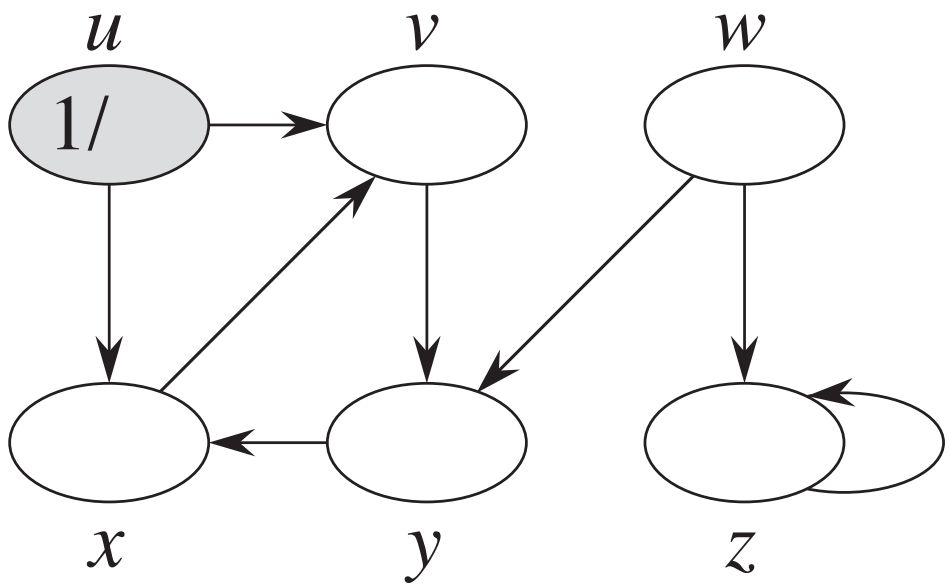
- *time* ist ein globaler Zeitstempel

Pseudo-Code (2)

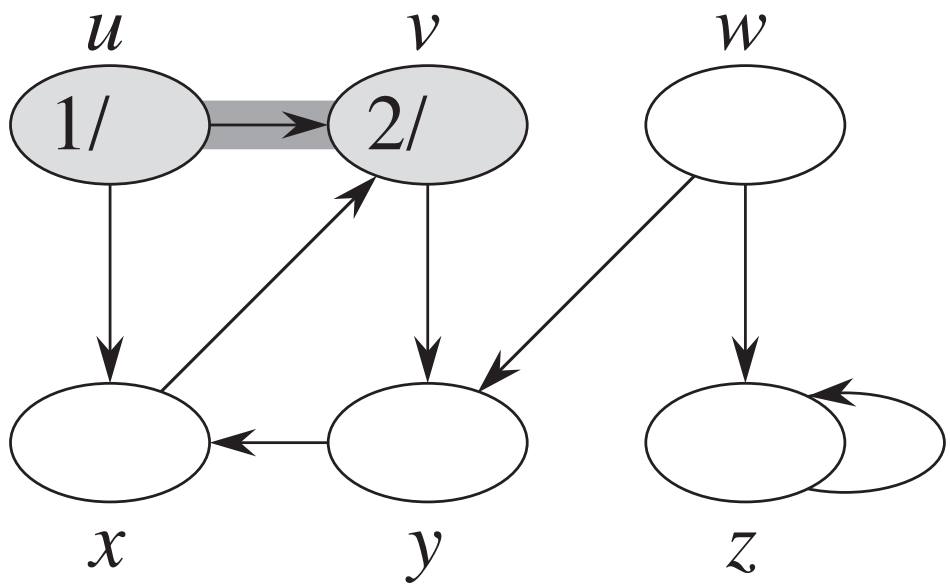
DFS-VISIT(G, u)

```
1:  time ← time + 1      // white vertex u has just been discovered
2:  u.d ← time
3:  u.color ← GRAY
4:  for each v ∈ G.Adj[u]                    // explore edge (u, v)
5:      if v.color = WHITE
6:          v.π = u
7:          DFS-VISIT(G, v)
8:  u.color ← BLACK                          // blacken u; it is finished
9:  time ← time + 1
10: u.f ← time
```

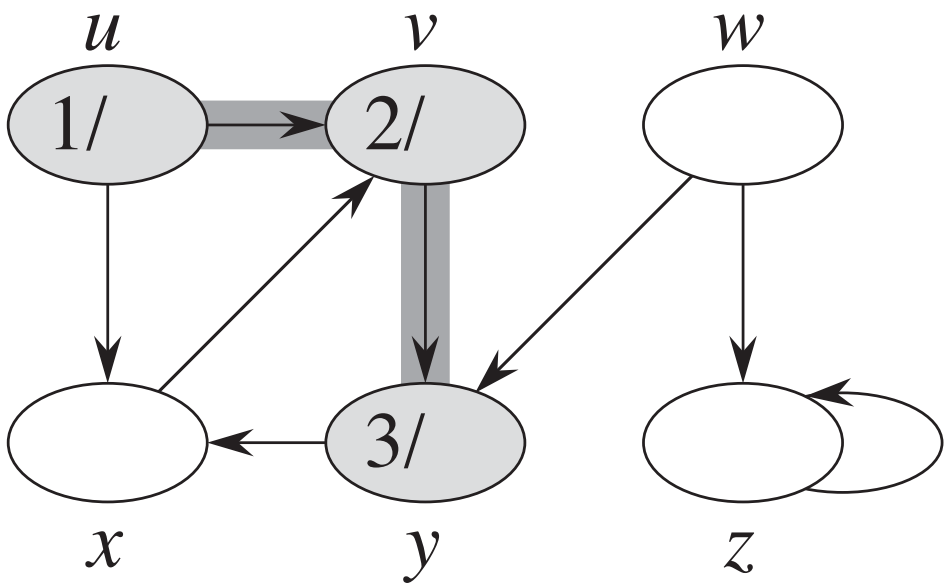
Beispiel (1)



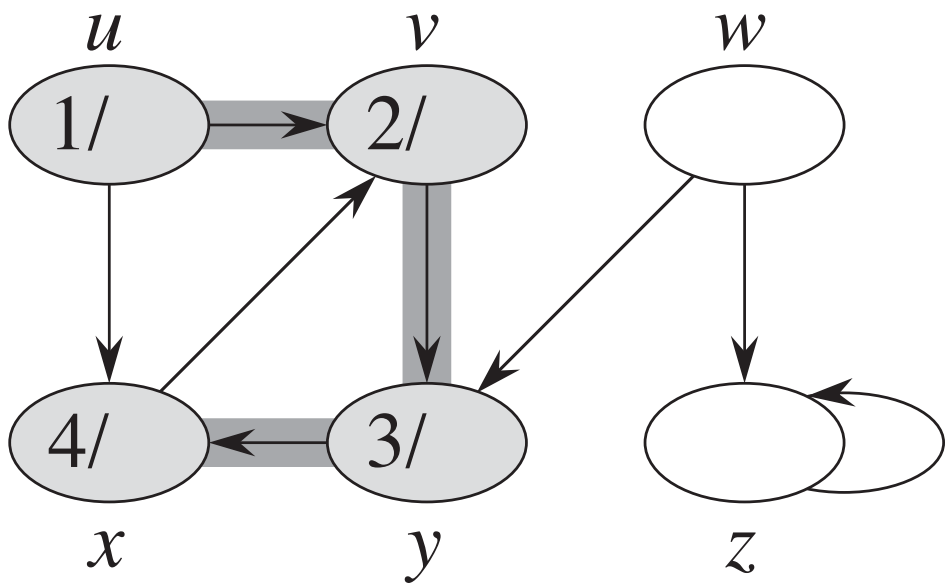
Beispiel (2)



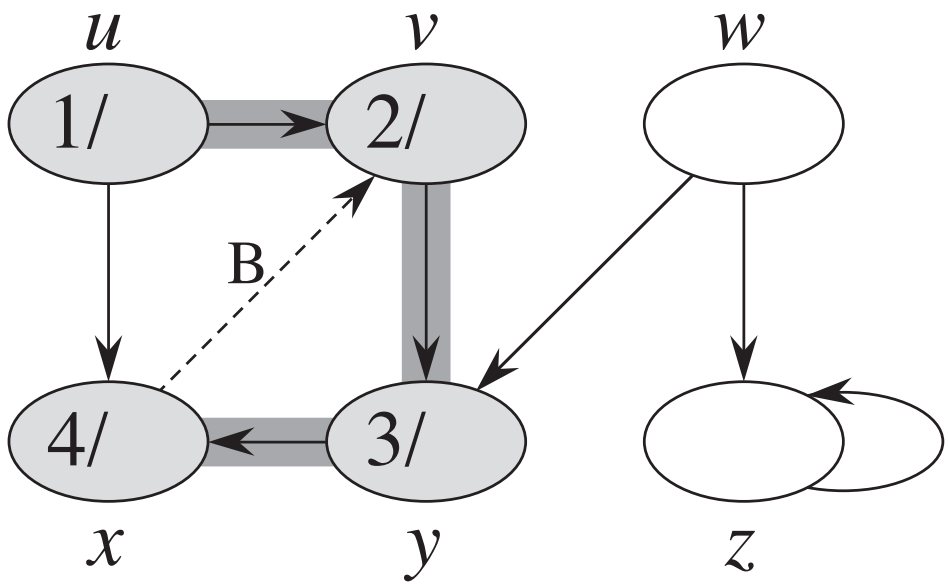
Beispiel (3)



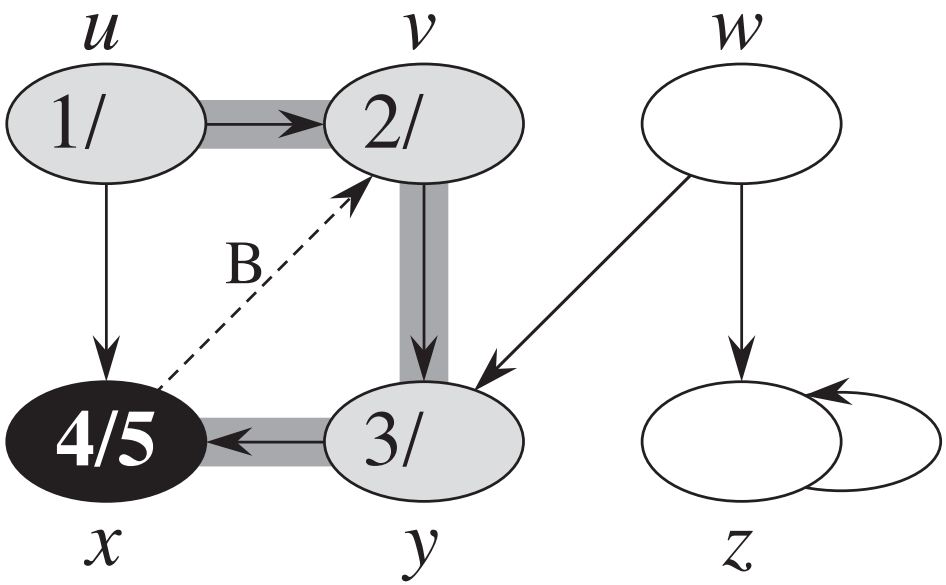
Beispiel (4)



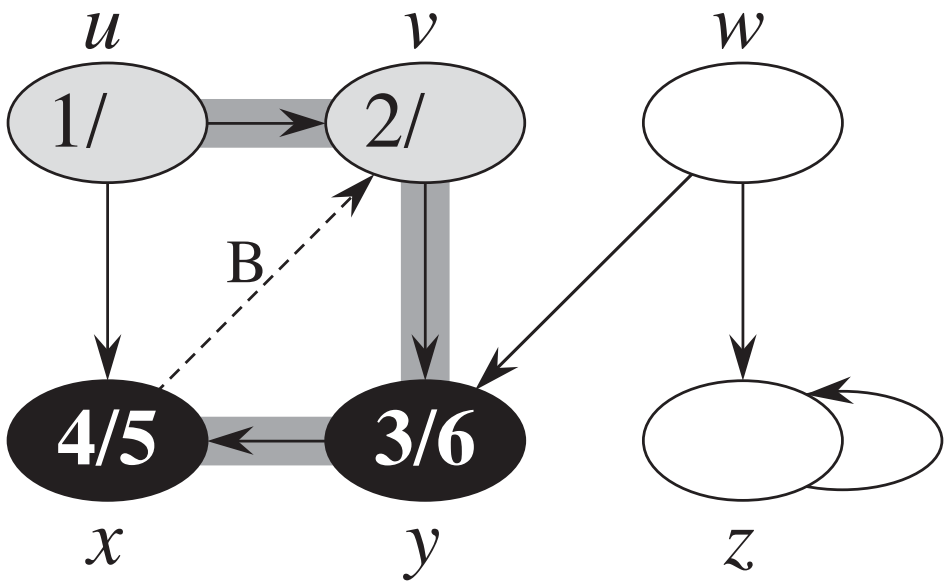
Beispiel (5)



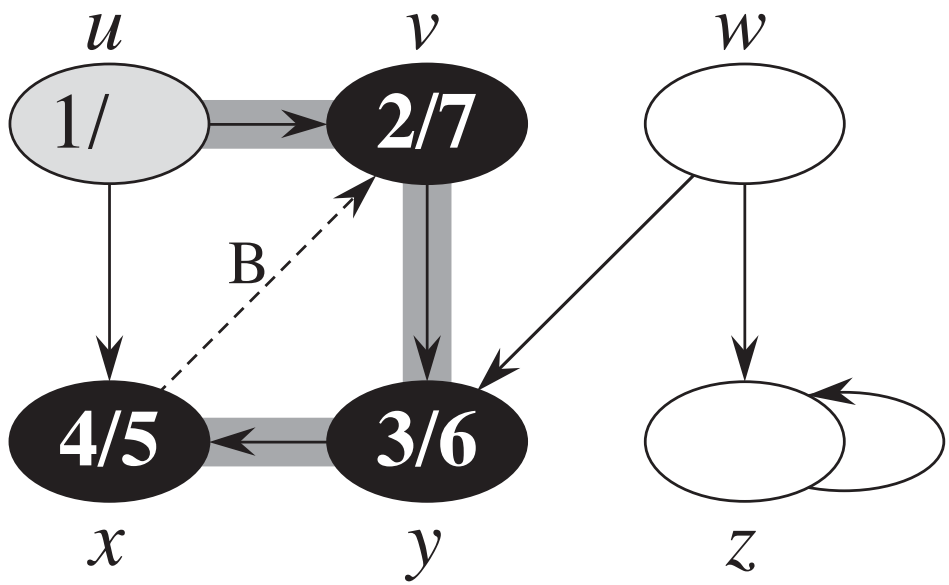
Beispiel (6)



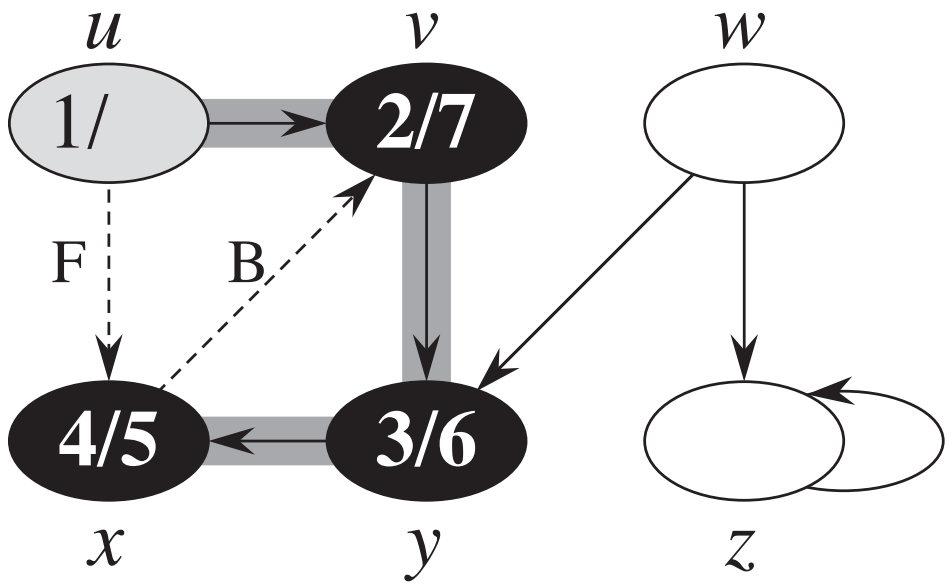
Beispiel (7)



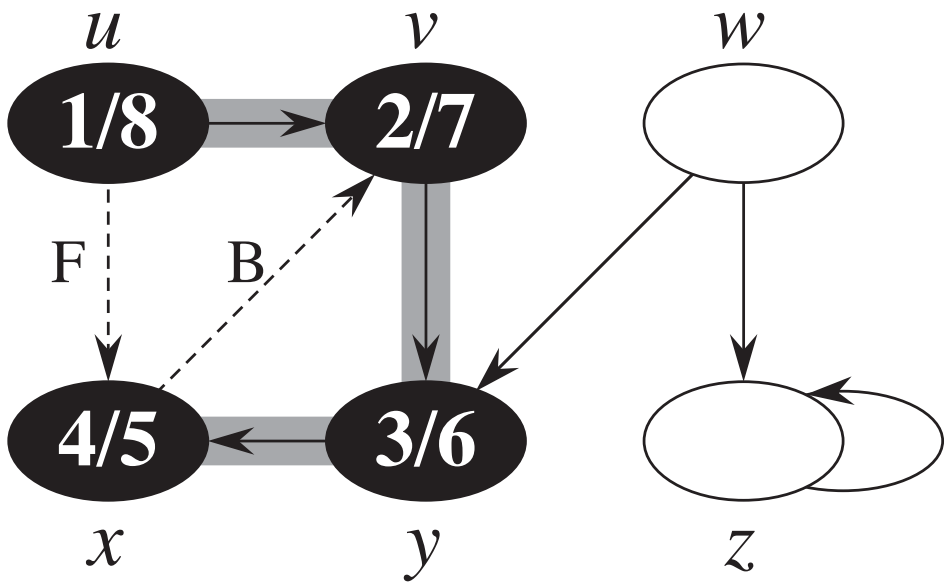
Beispiel (8)



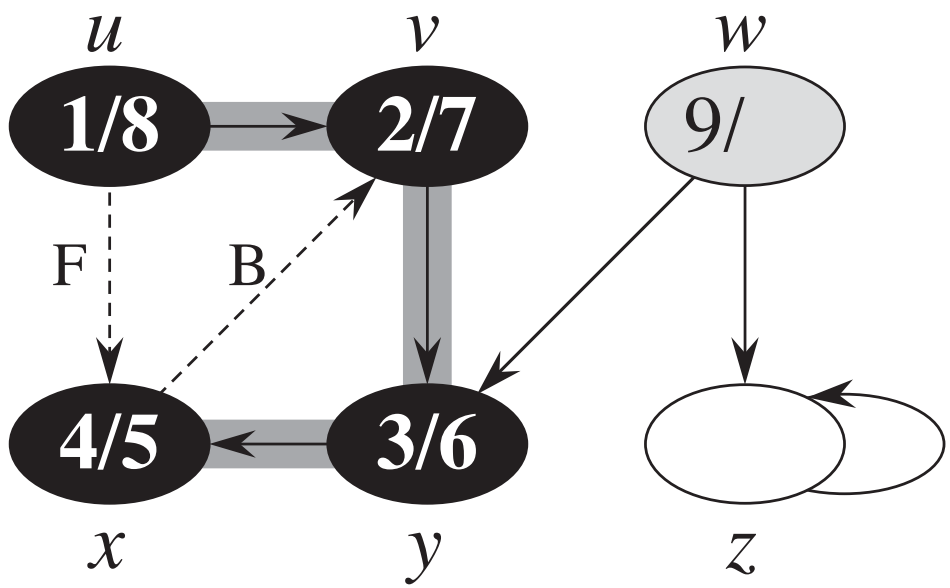
Beispiel (9)



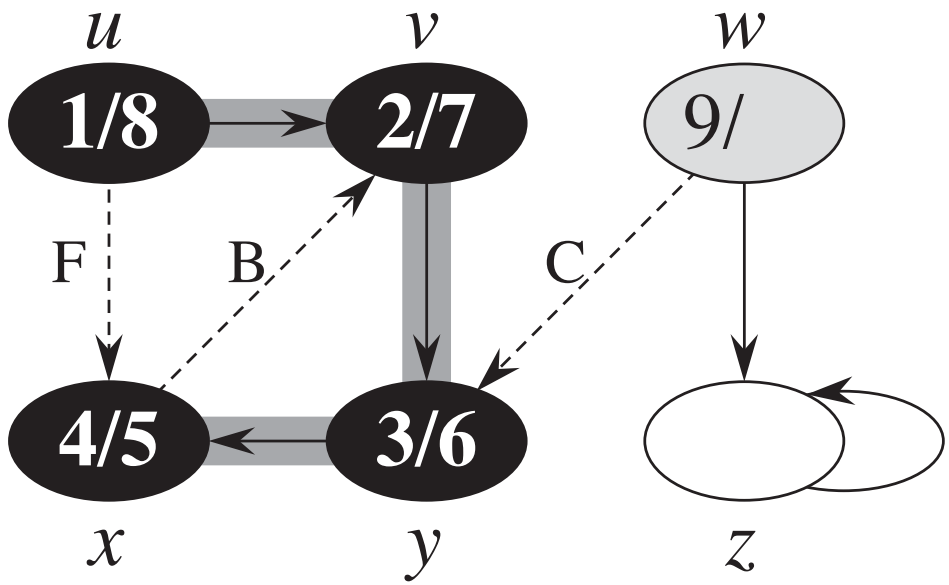
Beispiel (10)



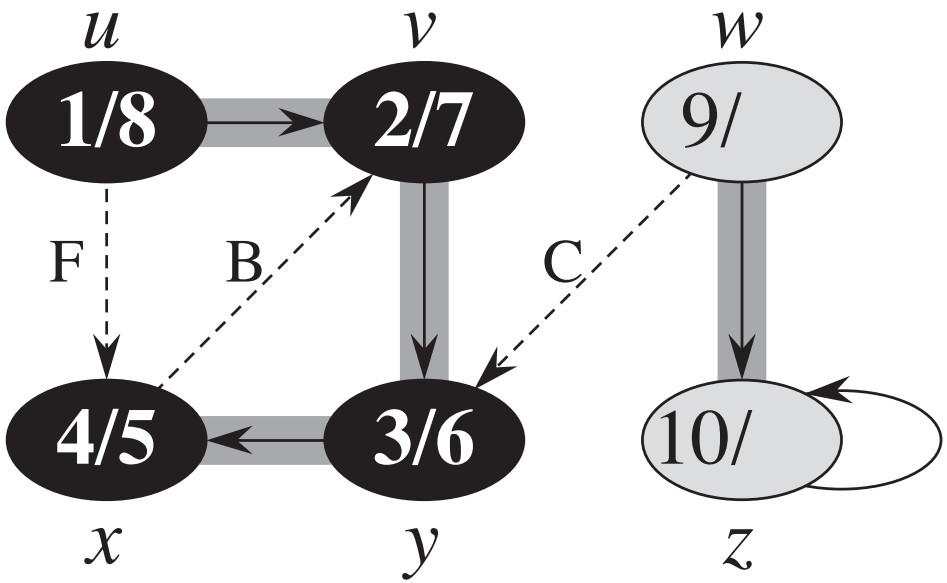
Beispiel (11)



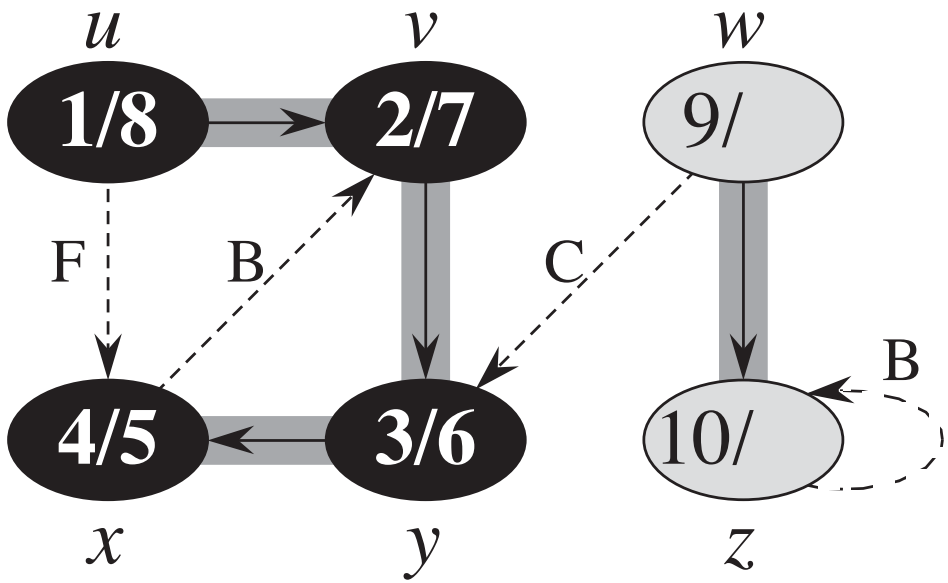
Beispiel (12)



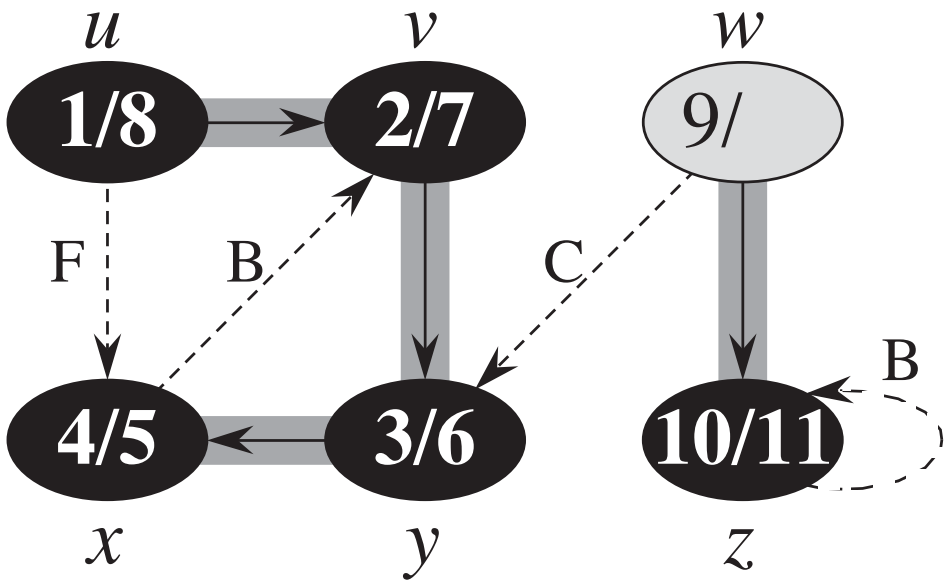
Beispiel (13)



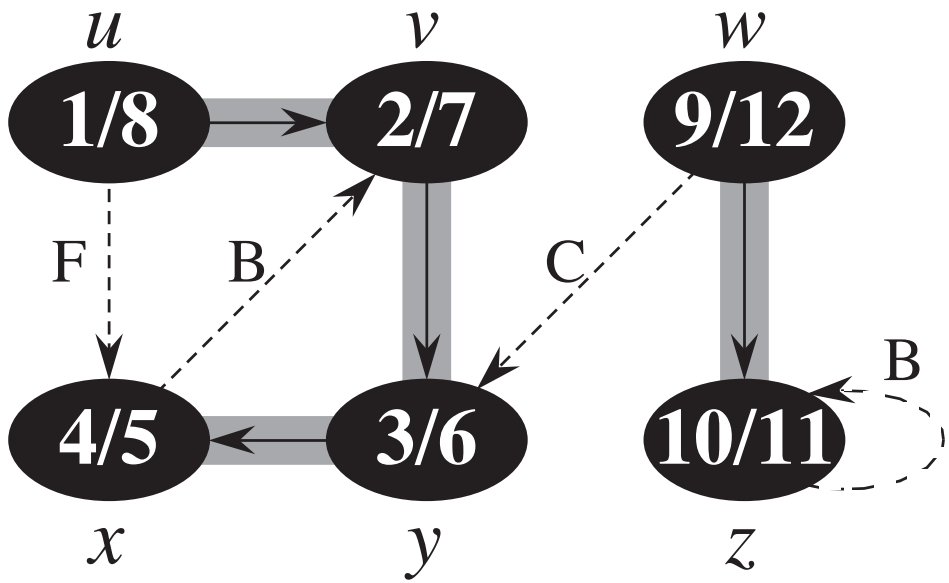
Beispiel (14)



Beispiel (15)



Beispiel (16)



Laufzeit

- Analyse ähnlich BFS
- Knoten weiß färben und als Startknoten probieren (ohne DFS-VISIT): $\Theta(V)$
- DFS-VISIT wird genau einmal pro Knoten ausgeführt
  - Adjazenzlisten werden in Z. 4 durchlaufen $\rightarrow$ jeder Eintrag einmal
  - Kosten für DFS-VISIT insgesamt: $\Theta(E)$
- Gesamtlaufzeit $\Theta(V + E)$

Vorgängerteilgraph und Tiefensuchewald

Leicht veränderte Definition, da kein expliziter Startknoten.

Definition

Der **Vorgängerteilgraph einer Tiefensuche** ist der Graph $G_\pi = (V, E_\pi)$ mit

$$E_\pi = \{ (v.\pi, v) \mid v \in V, v.\pi \neq \text{NIL} \}.$$

- G_π ist ein **Tiefensuchewald**, bestehend aus mehreren **Tiefensuchebäumen**
- Kanten in E_π nennt man **Baumkanten**
- Wenn $u = v.\pi$, dann wurde DFS-VISIT(G, v) während der Suche in u's Adjazenzliste aufgerufen
  $\rightarrow$ Tiefensuchewald beschreibt Struktur der rekursiven Aufrufe
- v ist Nachfolger von $u \Leftrightarrow v$ entdeckt während u grau

Klammertheorem (1)

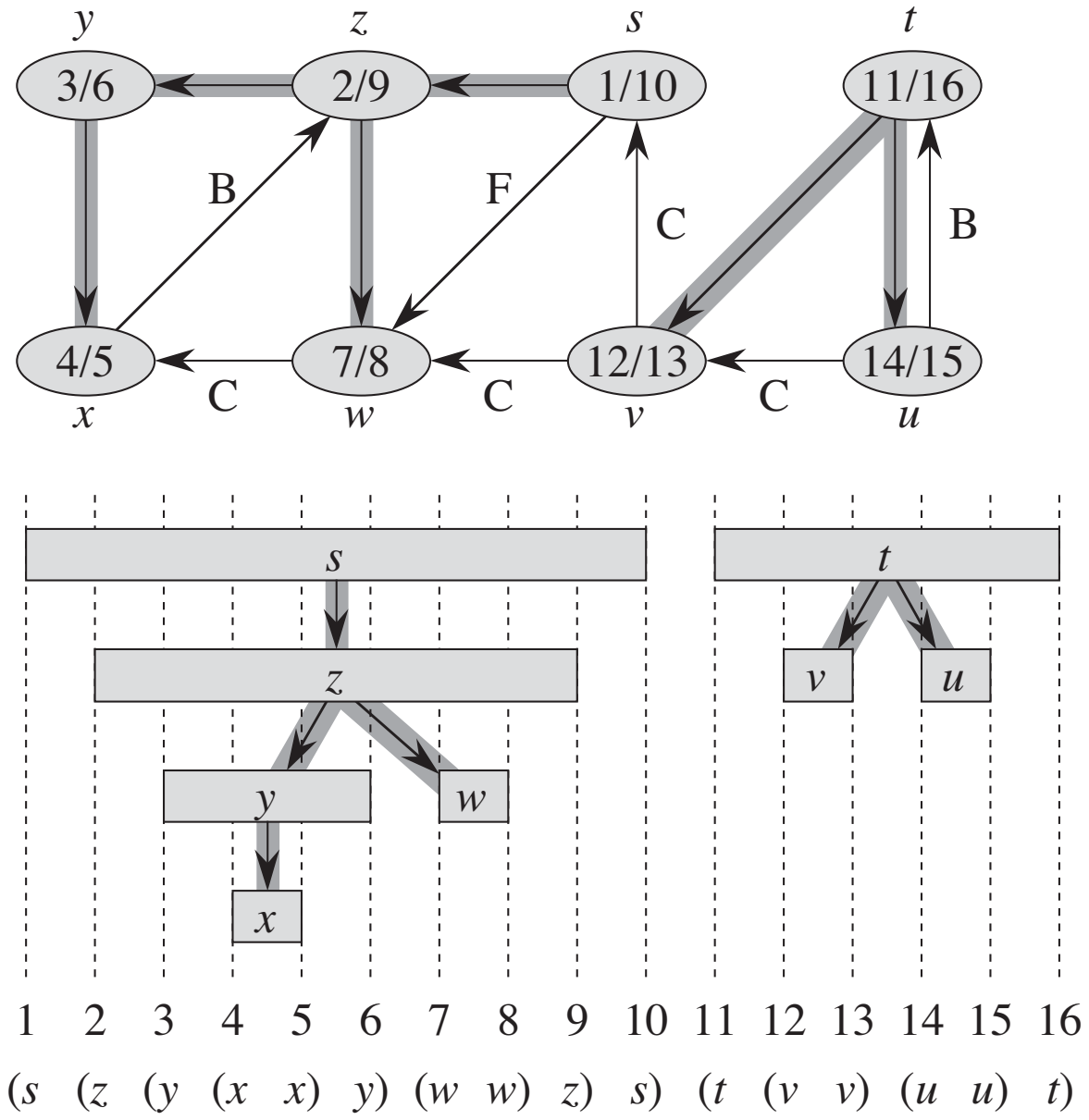
Theorem

Nach der Tiefensuche eines gerichteten oder ungerichteten Graphen $G = (V, E)$ gilt für jedes Knotenpaar $u, v \in V$ genau eine der drei folgenden Bedingungen:

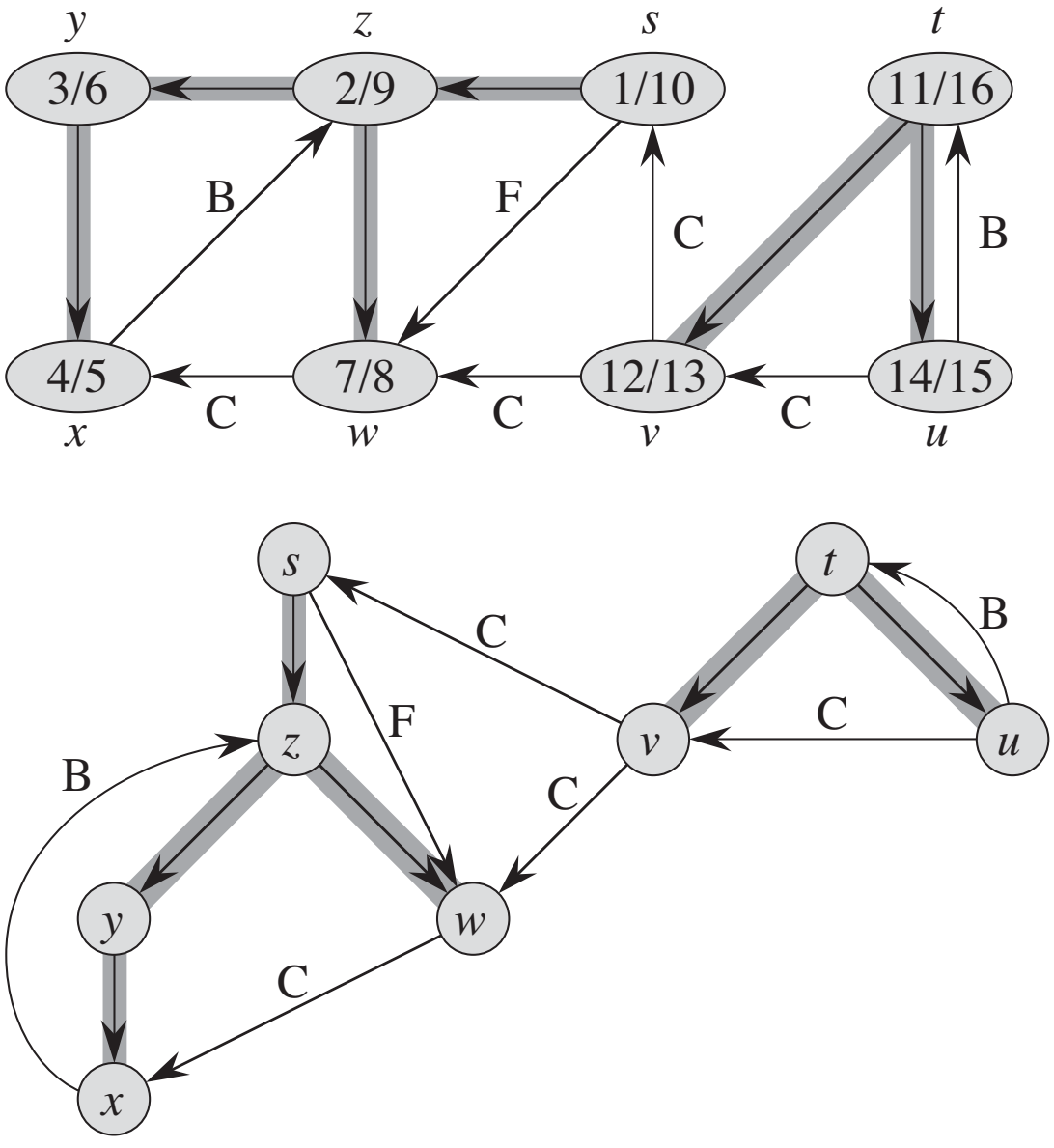
1. *die Intervalle $[u.d, u.f]$ und $[v.d, v.f]$ sind disjunkt und weder u noch v ist Nachfolger des anderen Knotens im Tiefensuchewald,*
2. *$[u.d, u.f]$ enthält $[v.d, v.f]$ und v ist ein Nachfolger von u in einem Tiefensuchebaum,*
3. *$[v.d, v.f]$ enthält $[u.d, u.f]$ und u ist ein Nachfolger von v in einem Tiefensuchebaum.*

- Namensgebung: siehe nächste Folie
- Korollar: v ist Nachfolger von $u \Leftrightarrow u.d < v.d < v.f < u.f$

Beispiel (1)


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u
s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

Beispiel (2)


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u
s
t
C
B
F
C
z
v
u
B
C
y
w
C
x

Klammertheorem (2)

Beweis.

- Fall 1: $u.d < v.d$
- Fall 1a: $u.d < v.d$ und $v.d < u.f$
  - v entdeckt als u grau $\rightarrow$ v Nachfolger von u
  - Da v nach u entdeckt, wird v abgearbeitet bevor Suche zu u zurückkehrt
  - Somit $v.f < u.f$ und $[u.d, u.f]$ enthält $[v.d, v.f]$
- Fall 1b: $u.d < v.d$ und $u.f < v.d$
  - Da $v.d < v.f$, gilt dass $[u.d, u.f]$ und $[v.d, v.f]$ disjunkt
  - Somit keiner der Knoten entdeckt, während der andere grau war $\rightarrow$ keiner der Knoten ist Nachfolger des anderen
- Fall 2: $u.d > v.d$ analog $\square$

Theorem des weißen Pfades (1)

Eine andere Charakterisierung des Tiefensuchewaldes:

Theorem

Im Tiefensuchewald eines gerichteten oder ungerichteten Graphen $G = (V, E)$ ist ein Knoten v ein Nachfolger eines Knotens u genau dann wenn zum Zeitpunkt $u.d$ ein Pfad von u nach v existiert, der vollständig aus weißen Knoten existiert.

Beweis.

- Nachfolgerschaft $\Rightarrow$ weißer Pfad
  - u ist weiß zum Zeitpunkt der Zuweisung von $u.d$ (Z. 2)
  - Da v Nachfolger von u, gilt $v.d > u.d$ nach Klammertheorem sodass auch v weiß ist
  - Dies gilt für jeden Nachfolger und somit auch für jeden Knoten auf dem einfachen Pfad von u nach v im Tiefensuchewald

Theorem des weißen Pfades (2)

- Weißer Pfad $\Rightarrow$ Nachfolgerschaft
  - Nehmen wir an, dass zum Zeitpunkt $u.d$ ein weißer Pfad von u nach v existiert
  - Annahme: v kein Nachfolger von u
  - oBdA: Alle anderen Knoten auf dem weißen Pfad sind Nachfolger von u (sonst: Wähle als v ersten Knoten, der kein Nachfolger)
  - Sei w Vorgänger von v (somit Nachfolger von u bzw. $=$u)
  - Es gilt $w.f \leq u.f$ nach dem Klammertheorem
  - v wird nach u entdeckt
  - v wird vor Abarbeitung von w entdeckt (da Kante (w, v) existiert)
  - Somit $u.d < v.d < w.f \leq u.f$
  - Klammertheorem impliziert dass $[v.d, v.f]$ in $[u.d, u.f]$ enthalten
  - Somit v Nachfolger von $u \rightarrow$ Widerspruch $\square$

Kantenklassifikation (1)

Die Kanten des Graphen lassen sich wie folgt klassifizieren:

- **Baumkanten** sind alle Kanten in E_π
  - (u, v) wenn v bei Ablaufen der Kante (u, v) zuerst entdeckt
- **Rückkanten** (B) verbinden Knoten mit Vorgängern
  - Wir betrachten Schleifen (u, u) in gerichteten Graphen als Rückkanten
- **Vorwärstkanten** (F) verbinden Knoten mit Nachfolgern, sind aber keine Baumkanten
- **Kreuzkanten** (C) sind alle anderen Kanten

Kantenklassifikation (2)

Modifikation von DFS-VISIT um Kanten (u, v) anhand der Farbe von v zu klassifizieren:

1. v ist weiß $\rightarrow$ Baumkante
2. v ist grau: Rückkante
3. v ist schwarz: entweder Vorwärts- oder Kreuzkante
   - Vorwärtskante gdw. $u.d < v.d$ (Übung)

Kantenklassifikation (3)

Im ungerichteten Graph kann es Mehrdeutigkeiten bezüglich der Klassifikation geben, da (u, v) und (v, u) die gleiche Kante darstellen. In so einem Fall gilt die **erste** gefundene Klassifikation.

Theorem

Die Kanten in einem ungerichteten Graphen $G = (V, E)$ werden durch die Tiefensuche entweder als Baumkanten oder als Rückkanten klassifiziert.

Beweis.

- Sei (u, v) eine beliebige Kante
- oBdA: $u.d < v.d$
- $\Rightarrow u.d < v.d < v.f < u.f$ da $v \in G.Adj[u]$
- Falls (u, v) zuerst in Richtung $u \rightarrow v$ durchlaufen, dann wird (u, v) eine Baumkante
- Falls (u, v) zuerst in Richtung $v \rightarrow u$ durchlaufen, so wird (u, v) eine Rückkante, da u noch grau ist $\square$

Outline

1. Elementare Graphalgorithmen
1.1 Repräsentation
1.2 Breitensuche
1.3 Tiefensuche
1.4 Topologisches Sortieren
1.5 Starke Zusammenhangskomponenten

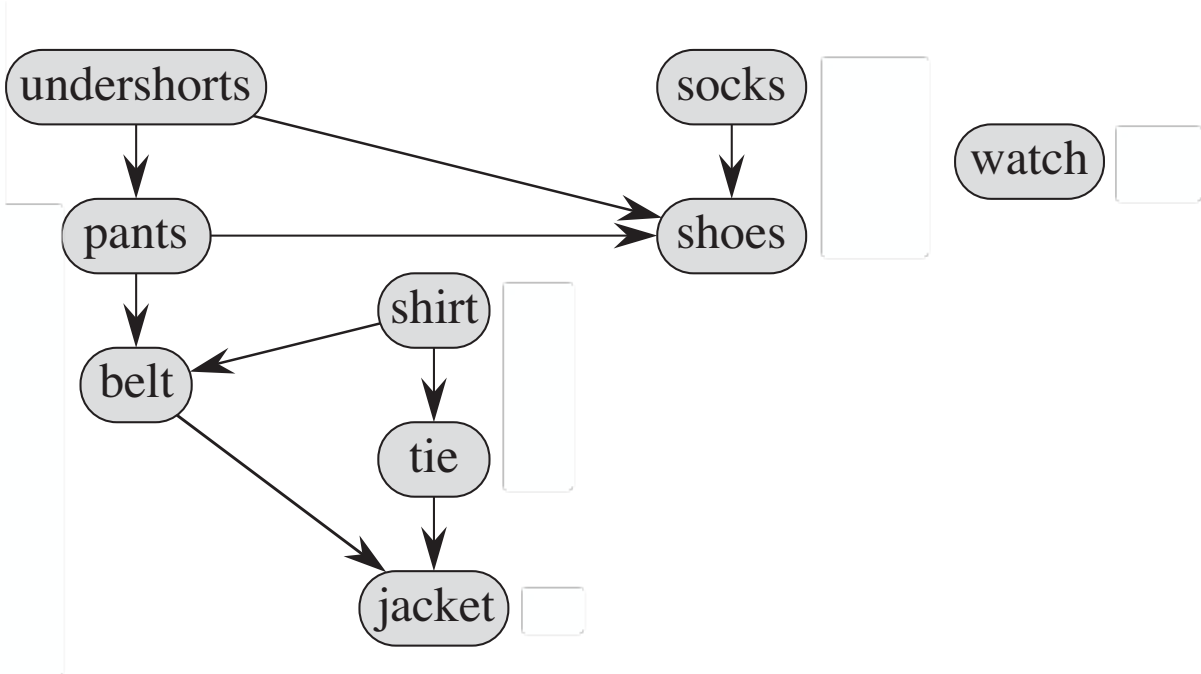
2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Gerichtete azyklische Graphen (DAG)

- Beispiel: Prozessmodellierung mit DAGs



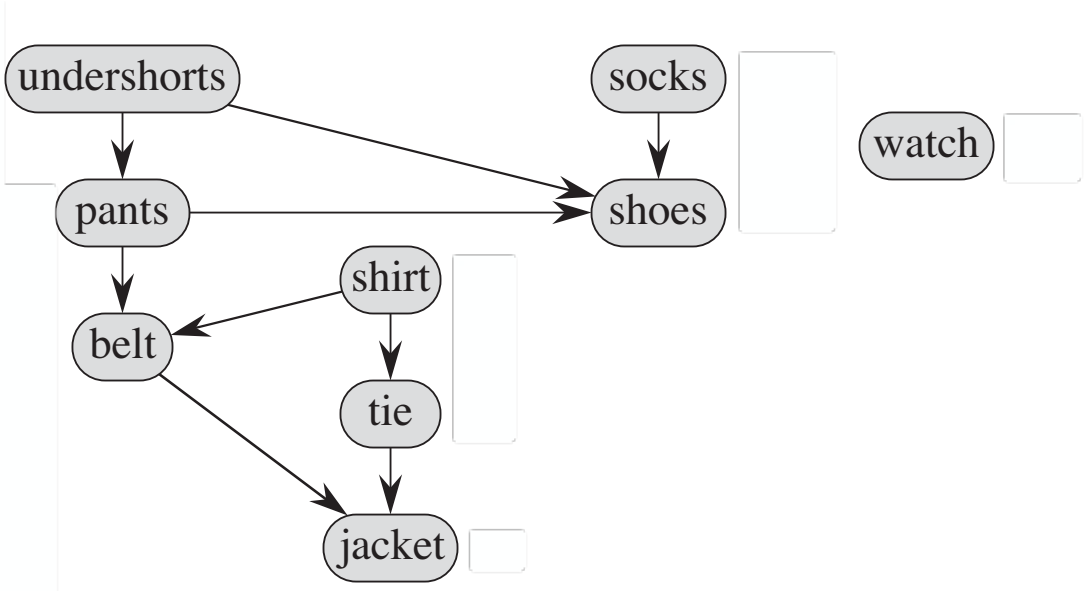
- Jeder DAG definiert eine Halbordnung
  - Es gilt $a < b$ und $b < c$ impliziert $a < c$
  - Aber: es kann Paare $a \neq b$ geben, sodass weder $a < b$ noch $a > b$

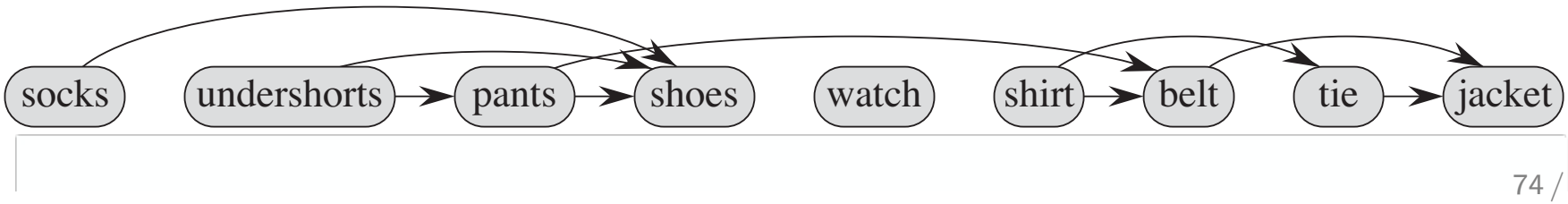
Topologische Sortierung

Definition

Eine **topologische Sortierung** $<$ eines DAG $G = (V, E)$ ist eine totale Ordnung aller Knoten in G sodass $u < v$ für $(u, v) \in E$.

Ordnet man die Knoten nach $<$ auf einer Linie an, gehen alle gerichteten Kanten von links nach rechts.





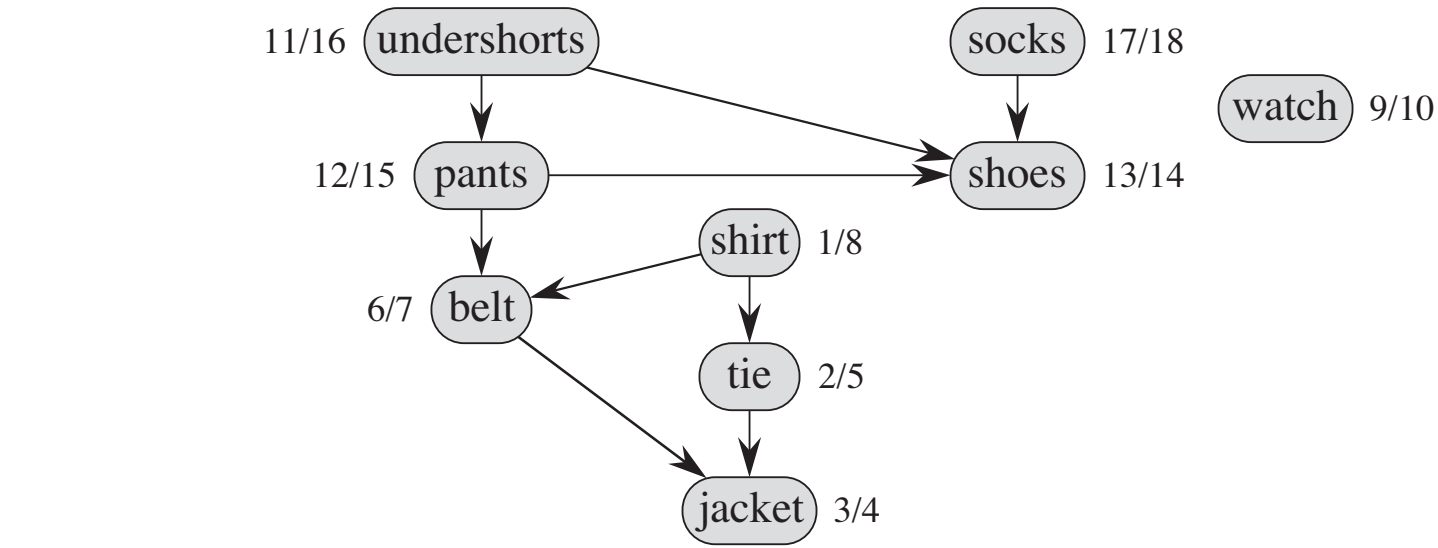
Pseudo-Code

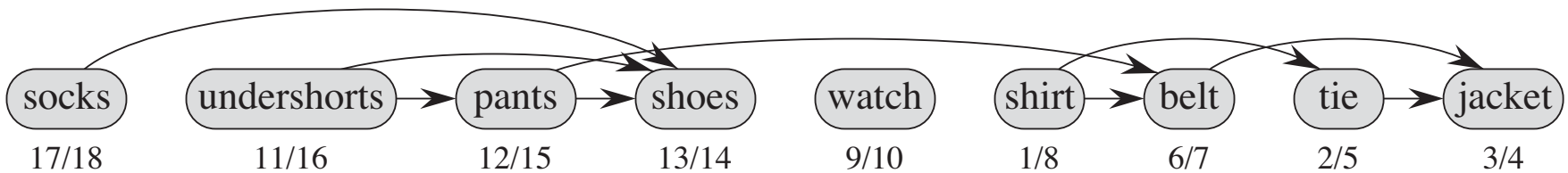
TOPOLOGICAL-SORT(G)

1: Call DFS(G) to compute finishing times $v.f$ for each vertex v
2: As each vertex is finished, insert it to the front of a linked list
3: **return** this linked list of vertices

- Ausgabe erfolgt also in umgekehrter Reihenfolge der Abarbeitungszeiten
- Kosten $O(V + E)$

Beispiel


11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4


socks
17/18
undershorts
11/16
pants
12/15
shoes
13/14
watch
9/10
shirt
1/8
belt
6/7
tie
2/5
jacket
3/4

Korrektheit (1)

Lemma

Ein gerichteter Graph G ist azyklisch genau dann, wenn DFS *keine Rückkanten ergibt.*

Somit kann man Kreisfreiheit mittels DFS erkennen.

Beweis.

- Rückkante $\Rightarrow$ Zyklus
  - Annahme: es existiert eine Rückkante (u, v)
  - Dann ist v ein Vorgänger von u im Tiefensuchewald
  - Also gibt es einen Pfad von v nach u in G
  - Und mit der Rückkante (u, v) ergibt sich ein Zyklus
- Zyklus $\Rightarrow$ Rückkante
  - Annahme: G enthält einen Zyklus C
  - Sei v der erste entdeckte Knoten in C und (u, v) die Vorgängerkante in C
  - Zum Zeitpunkt $v.d$ gibt es einen weißen Pfad von v nach u $\rightarrow$ u wird Nachfolger von v
  - Somit ist (u, v) Rückkante

□

Korrektheit (2)

Theorem

TOPOLOGICAL-SORT(G) *erzeugt eine topologische Sortierung eines DAG* $G = (V, E)$.

Beweis.

- z.Z.: Für je zwei verschiedene Knoten $u, v \in V$ gilt:

$$(u, v) \in E \Rightarrow v.f < u.f$$

- Wenn DFS Kante (u, v) abarbeitet, kann v nicht grau sein (sonst wäre (u, v) eine Rückkante)
- Falls v weiß, so wird v Nachfolger von u und somit $v.f < u.f$
- Falls v schwarz gilt unmittelbar $v.f < u.f$ □

Outline

1. Elementare Graphalgorithmen
1.1 Repräsentation
1.2 Breitensuche
1.3 Tiefensuche
1.4 Topologisches Sortieren
1.5 Starke Zusammenhangskomponenten

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

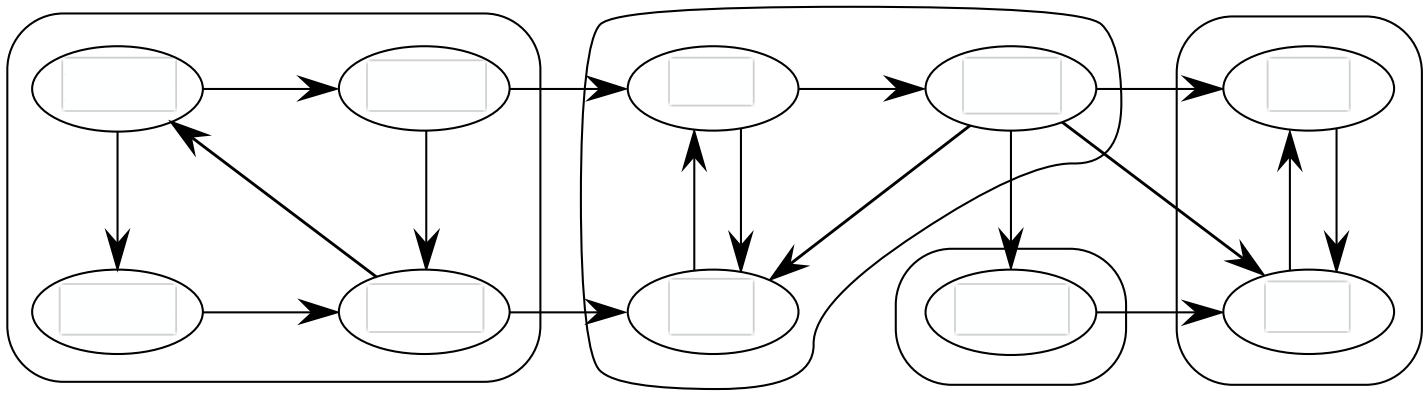
Starke Zusammenhangskomponenten

Definition

Eine **starke Zusammenhangskomponente** (*strongly connected component*, *SCC*) eines gerichteten Graphen $G = (V, E)$ ist eine maximale Knotenmenge $C \subseteq V$ sodass für jedes Paar $u, v \in C$ gilt: $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

- $u \rightsquigarrow v$ heißt: von u kann v erreicht werden
- Zerlegung in SCCs wird oft als erster Schritt in Algorithmen benötigt, um ein Problem in Teilproblem zu zerlegen

Beispiel



Transponierter Graph

- Zerlegung kann mit Hilfe von DFS durchgeführt werden
- Dazu verwenden wir den **transponierten Graphen** $G^T = (V, E^T)$ mit

$$E^T = \{ (v, u) \mid (u, v) \in E \}$$

- Transponierter Graph kann bei Adjazenzlistendarstellung in $O(V + E)$ erzeugt werden
- Bemerke: G und G^T haben die gleichen starken Zusammenhangskomponenten

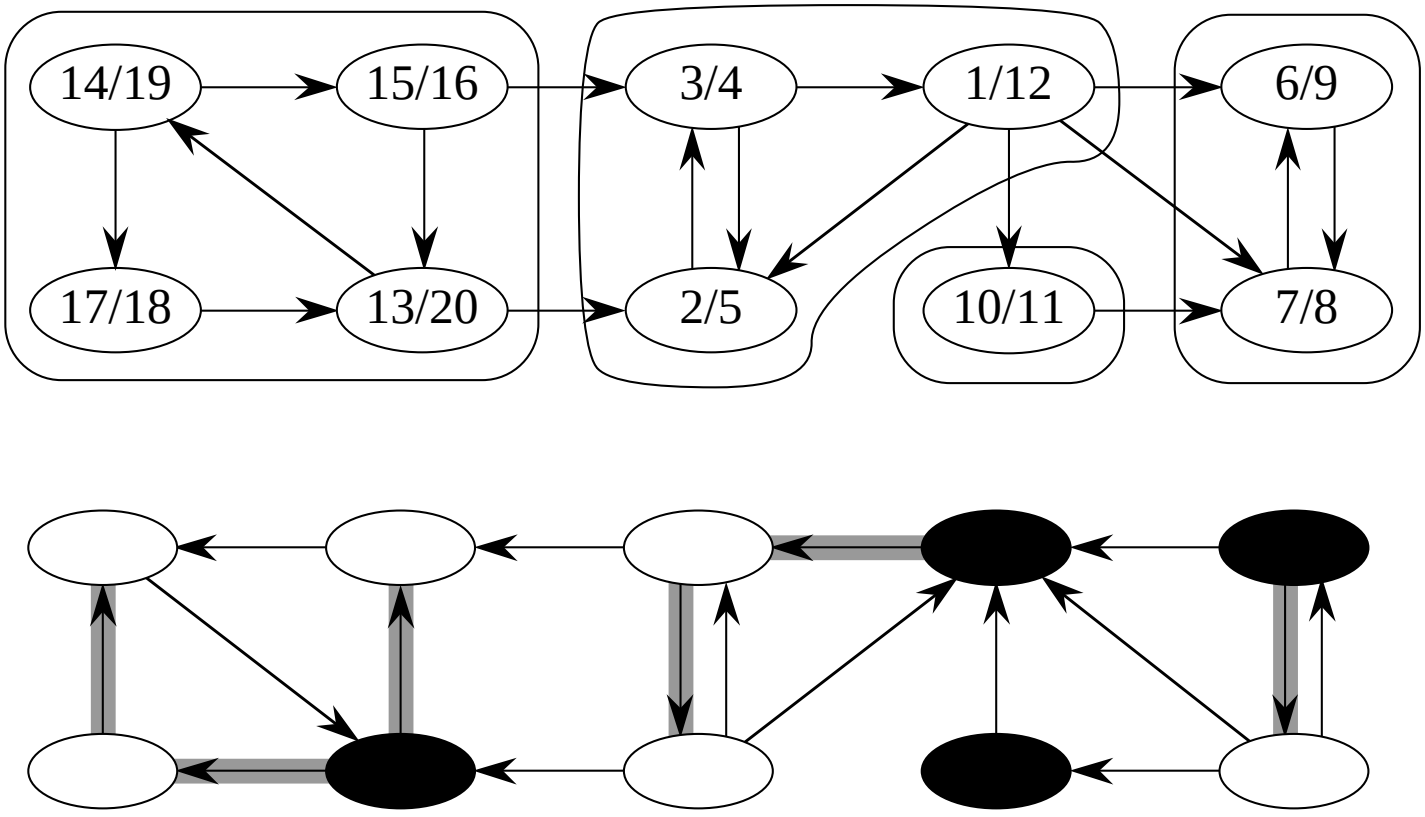
Pseudo-Code

STRONGLY-CONNECTED-COMPONENTS(G)

1: Call DFS(G) to compute finishing times $u.f$ for each vertex u
2: Compute G^T
3: Call DFS(G^T), but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS)
4: Output the vertices of each tree of the depth-first forest resulting from line 3 as a separate SCC

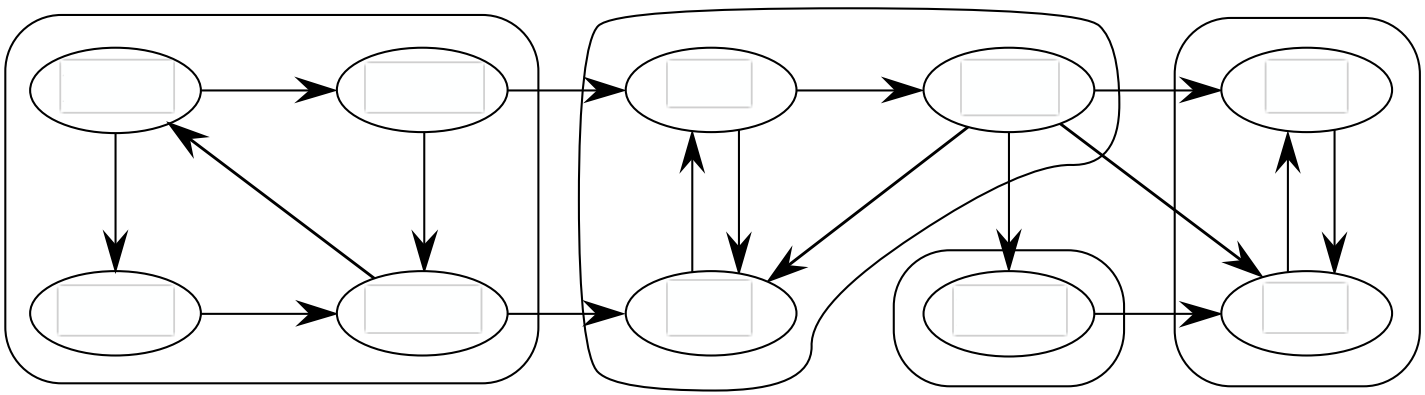
- Laufzeit: $O(V + E)$

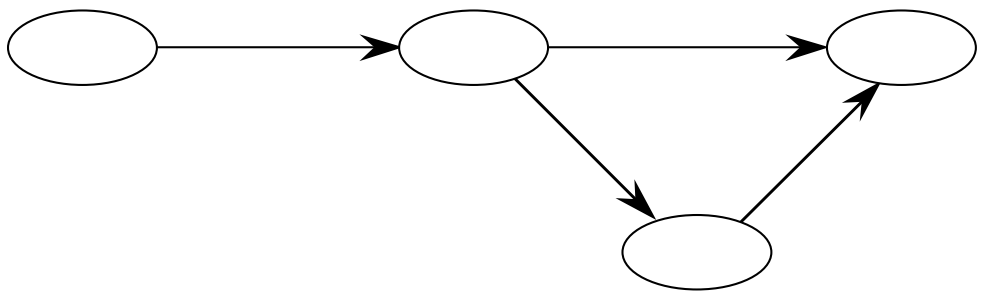
Beispiel



Komponentengraph

- **Komponentengraph** $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ nützlich zum Verständnis des Algorithmus
- Ein Knoten pro SCC in G
- Kante zwischen zwei Knoten, wenn entsprechende SCCs in G durch Kante verbunden
- Alternativ: Knoten innerhalb einer SCC werden verschmolzen





Komponentengraph ist DAG

Lemma

G^{SCC} ist ein DAG. Genauer: Wenn C und C' zwei verschiedene SCCs mit $u, v \in C$, $u', v' \in C'$ und ein Pfad $u \rightsquigarrow u'$ in G existiert, dann gibt es keinen Pfad $v' \rightsquigarrow v$ in G.

Somit: Falls zwei Knoten in der selben SCC, so verlässt kein Pfad zwischen diesen Knoten die SCC.

Beweis.

- Es gibt einen Pfad $u \rightsquigarrow u' \rightsquigarrow v'$
- Nehmen wir an es gäbe außerdem Pfad $v' \rightsquigarrow v$
- Dann gibt es auch einen Pfad $v' \rightsquigarrow v \rightsquigarrow u$
- Aber dann sind u und v' in der gleichen SCC$\rightarrow$ Widerspruch $\square$

Korrektheit (1)

Kernidee. Da Knoten in zweiter Tiefensuche in umgekehrter Reihenfolge der Abarbeitungszeiten geordnet, werden die Knoten des Komponentengraphs in topologischer Ordnung besucht.

- Somit: Wenn eine Kante im transponierten Graph zu anderen Komponente führt, dann wurde diese andere Komponente schon abgearbeitet
- Das schauen wir uns jetzt genauer an
- Ab jetzt: $u.d$ und $u.f$ beziehen sich auf erste Tiefensuche
- Außerdem erweitern wir die Definitionen auf Knotenmengen

$$d(C) = \min_{u \in C} u.d \qquad \text{(früheste Entdeckung)}$$

$$f(C) = \max_{u \in C} u.f \qquad \text{(letzte Abarbeitung)}$$

Korrektheit (2)

Lemma (F87)

Seien C und C' verschiedene SCCs in $G = (V, E)$. Wenn es eine Kante $(u, v) \in E$ gibt, sodass $u \in C$ und $v \in C'$, dann

$$f(C) > f(C').$$

Beweis.

- Fall 1: $d(C) < d(C')$
  - Sei x erster entdeckter Knoten in C
  - Zum Zeitpunkt $x.d$ sind alle Knoten in C und C' weiß
  - Somit existiert ein weißer Pfad von x zu jedem Knoten in C oder C'
  - Alle Knoten in C und C' werden Nachfolger von x in Tiefensuchebaum
  - Nach Klammertheorem: $x.f = f(C) > f(C')$

Korrektheit (3)

- Fall 2: $d(C') < d(C)$
  - Sei y erster entdeckter Knoten in C'
  - Zum Zeitpunkt $y.d$ sind alle Knoten in C' weiß und von y erreichbar
    $\rightarrow$ Werden Nachfolger von y sodass $y.f = f(C')$
  - Zum Zeitpunkt $y.d$ sind alle Knoten in C auch weiß
  - Da Komponentengraph ein DAG und (u, v) existiert, gibt es keinen Pfad von C' nach C
    $\rightarrow$ Kein Knoten in C ist von y aus erreichbar
  - Somit sind zum Zeitpunkt $y.f$ alle Knoten in C immer noch weiß
  - Impliziert für alle $w \in C$, dass $w.f > y.f$
  - Somit $f(C) > f(C')$ □

Korrektheit (4)

Korollar

Seien C und C' verschiedene SCCs in $G = (V, E)$. Wenn es eine Kante $(u, v) \in E^T$ gibt, sodass $u \in C$ und $v \in C'$, dann

$$f(C) < f(C').$$

Das folgende Korollar fasst unsere Erkenntnisse zusammen.

Korollar (F89)

Seien C und C' verschiedene SCCs in $G = (V, E)$ und sei $f(C) > f(C')$. Dann gibt es keine Kante von C nach C' in G^T.

Korrektheit (5)

Theorem

STRONGLY-CONNECTED-COMPONENTS *ermittelt die starken Zusammenhangskomponenten des gerichteten Graphen G.*

Beweisidee.

- In zweiter Tiefensuche starten wir mit Knoten x_1 sodass $x_1.f$ maximal
  - Alle Knoten in x_1's SCC C_1 werden entdeckt
  - Aber: Kein Knoten aus einer anderen SCC in G^T erreichbar (nach Korollar F89)
  - Somit nur Knoten in C_1 entdeckt
    $\rightarrow$ Tiefensuchebaum mit Wurzel x_1 entspricht C_1

Korrektheit (6)

- Wir besuchen als nächstes den unbesuchten Knoten x_2 mit $x_2.f$ maximal
  - Alle Knoten in x_2's SCC C_2 werden entdeckt
  - Kanten, die C_2 verlassen, müssen zu Knoten in C_1 gehen
  - C_1 ist aber schon abgearbeitet $\rightarrow$ Keine Baumkanten
  - Tiefensuchebaum mit Wurzel x_2 entspricht C_2
- Das kann man weiter fortsetzen (Induktion)
- Von jedem Wurzelknoten erreicht man
  - Alle Knoten innerhalb der entsprechenden SCC (Baumkanten zu diesen Knoten)
  - Sonst nur Knoten, die schon entdeckt worden sind (keine Baumkanten) □

Outline

1. Elementare Graphalgorithmen

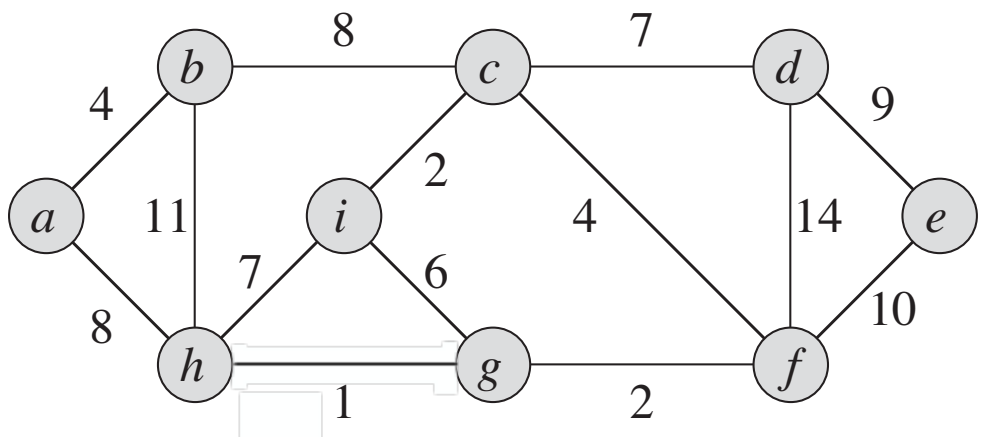
2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Motivation

- Beispiel: Verkabelung
- V sind Orte, E sind mögliche Kabelstrecken, $(u, v) \in E$ hat assoziierte Kosten $w(u, v)$
- Welche Kabelstrecken verbinden alle Orte und haben minimale Gesamtkosten?
- Wir suchen also eine Teilmenge $T \subseteq E$ der Kanten
  - (V, T) ist verbunden
  - T ist zyklenfrei



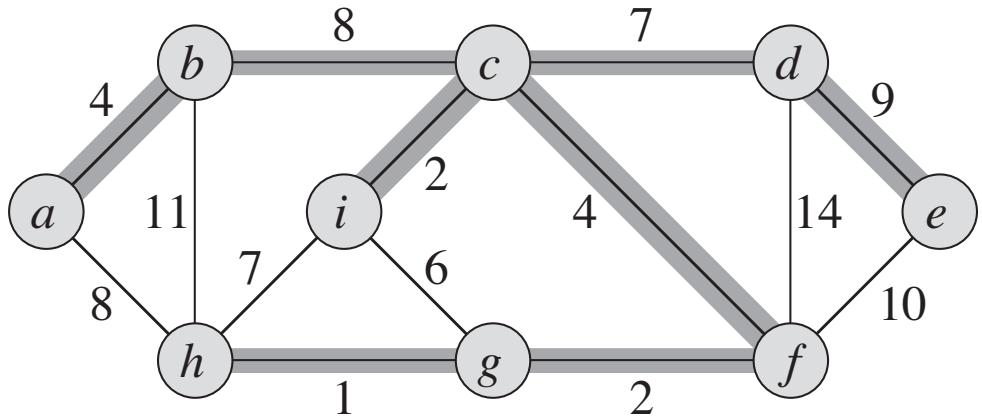
Minimaler Spannbaum

Definition

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$.

- Eine Teilmenge $T \subset E$ der Kanten heißt **Spannbaum** (*spanning tree*) von G, wenn (V, T) ein Baum ist.
- Ein Spannbaum T ist ein **minimaler Spannbaum** (*minimum spanning tree*, *MST*), falls $w(T) \leq w(T')$ für alle Spannbäume T' von G.

Hierbei sei $w(T) = \sum_{(u,v)\in T} w(u, v)$ das **Gewicht** des Spannbaums T.



Berechnung eines minimalen Spannbaums

- Zuerst: Generischer Algorithmus
- Dann: Zwei Varianten
  - Algorithmus von Prim
  - Algorithmus von Kruskal
- Idee: Wir berechnen die Kantenmenge A des MST
  - Initial $A = \emptyset$
  - Kanten werden Schritt für Schritt hinzugefügt
- **Invariante**: A ist immer eine Teilmenge eines minimalen Spannbaums
  - Eine Kante (u, v) heißt **sicher**, falls auch $A \cup (u, v)$ eine Teilmenge eines minimalen Spannbaums ist
  - Wir fügen nur sichere Kanten zu A hinzu

Generischer Algorithmus (Pseudo-Code)

GENERIC-MST(G,w)

1: $A \leftarrow \emptyset$
2: **while** A does not form a spanning tree
3: Find an edge (u, v) that is safe for A
4: $A \leftarrow A \cup \{ (u, v) \}$
5: **return** A

Korrektheit

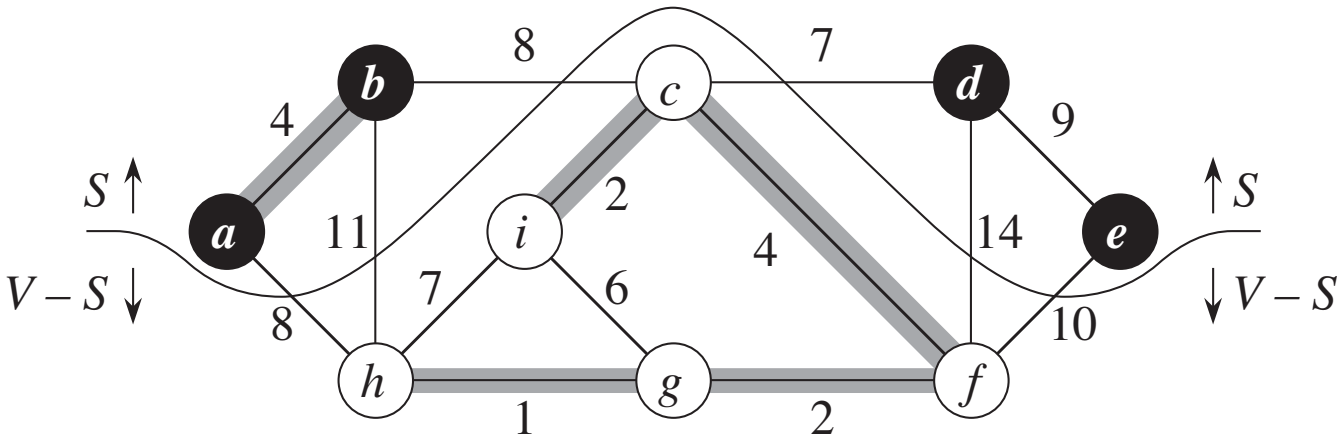
- **Induktionsanfang**: $A = \emptyset$ erfüllt Invariante
- **Induktionsschritt**: Nur sichere Kanten hinzugefügt $\rightarrow$ A bleibt Teilmenge eines MST
- **Terminierung**: A ist Spannbaum und alle Kanten in A sind auch in einem MST $\rightarrow$ A ist MST

Aber: Welche Kanten sind sicher?

Schnitte (Cuts)

Definition

- Ein **Schnitt** (*cut*) $(S, V \setminus S)$ ist eine Partitionierung von V.
- Eine Kante $(u, v) \in E$ **kreuzt** (*crosses*) Schnitt $(S, V \setminus S)$ falls $u \in S$ und $v \in V \setminus S$ (oder umgekehrt).
- Eine Kantenmenge $A \subseteq E$ **respektiert** einen Schnitt, falls keine Kante in A den Schnitt kreuzt.
- Eine kreuzende Kante heißt **leicht**, falls ihr Gewicht minimal unter allen kreuzenden Kanten ist.



Bestimmung von sicheren Kanten (1)

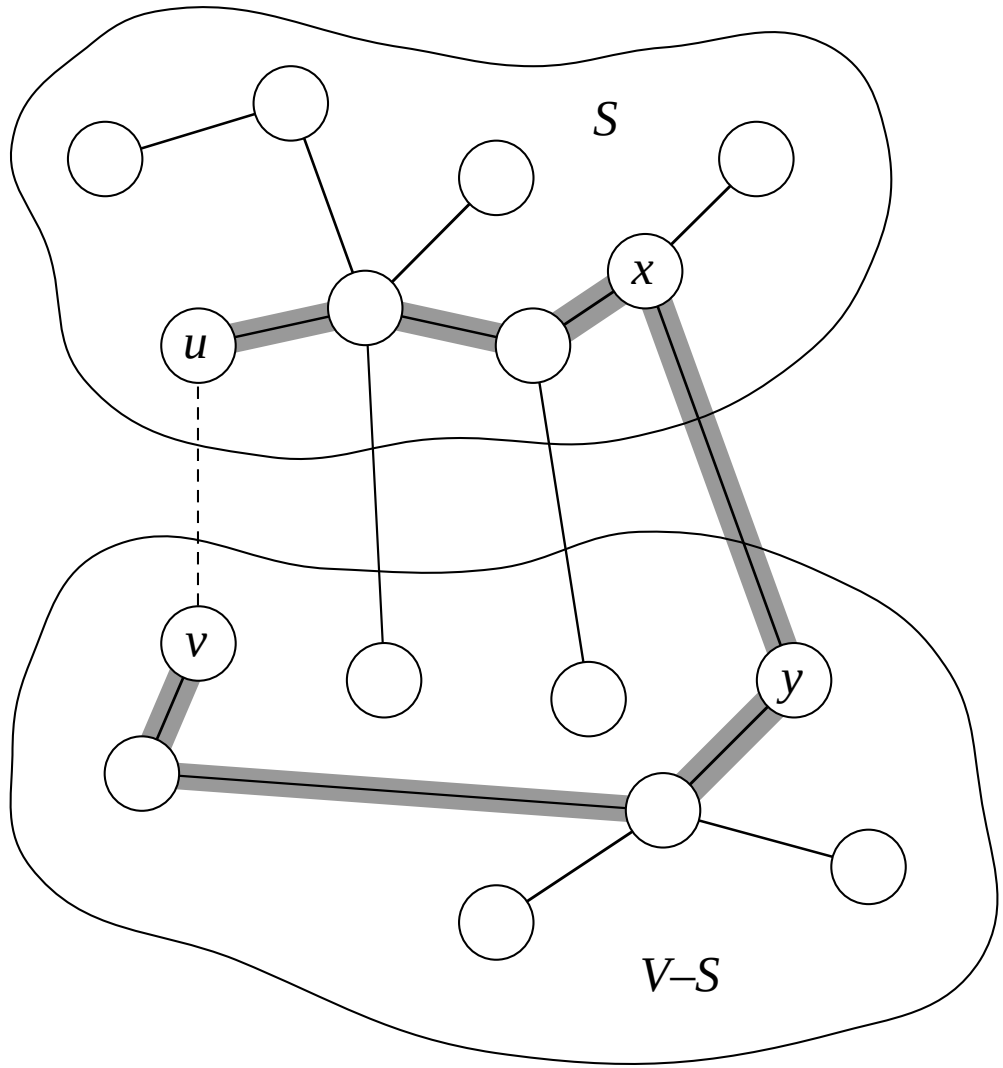
Theorem

Sei $A \subseteq E$ in einem minimalen Spannbaum für G enthalten. Sei $(S, V \setminus S)$ ein Schnitt von G, der A respektiert und (u, v) eine leichte, kreuzende Kante. Dann ist (u, v) sicher für A.

Beweis.

- Sei T ein MST der A enthält
- Sei $(u, v) \notin T$ (sonst: Behauptung gilt)
- Hinzufügen von (u, v) zu T erzeugt einen Zyklus
- Da u und v auf gegenüberliegenden Seiten liegen, gibt es mindestens eines Kante $(x, y) \in T$ auf dem Pfad von u nach v, die den Schnitt ebenso kreuzt

Bestimmung von sicheren Kanten (2)



Bestimmung von sicheren Kanten (3)

- Das Entfernen von (x, y) zerlegt T in zwei Zusammenhangskomponenten
- Hinzufügen von (u, v) verbindet diese Komponenten
  $\rightarrow$ Spannbaum $T' = T \setminus \{ (x, y) \} \cup \{ (u, v) \}$
- T' ist (auch) ein minimaler Spannbaum, denn
  - (u, v) ist leicht für Schnitt $(S, V - S)$
  - (x, y) kreuzt Schnitt $(S, V - S)$
  - Somit $w(u, v) \leq w(x, y)$
  - Somit $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- (u, v) ist sicher da $(u, v) \in T'$ und $A \cup \{ (u, v) \} \subseteq T'$ □

Algorithmische Fragen

- Welchen Schnitt wählen wir?
- Wie finden wir schnell eine leichte Kante?

Algorithmus von Kruskal

Korollar

Sei $A \subseteq E$ eine Teilmenge eines minimalen Spannbaums. Sei $C = (V_C, E_C)$ eine Zusammenhangskomponente im Wald $G_A = (V, A)$ und (u, v) eine leichte Kante, die C mit einer anderen Zusammenhangskomponente verbindet. Dann ist (u, v) sicher für A.

Beweis.

- (u, v) kreuzt den Schnitt $(V_C, V \setminus V_C)$
- Korrektheit folgt aus Theorem mit $S = V_C$ □

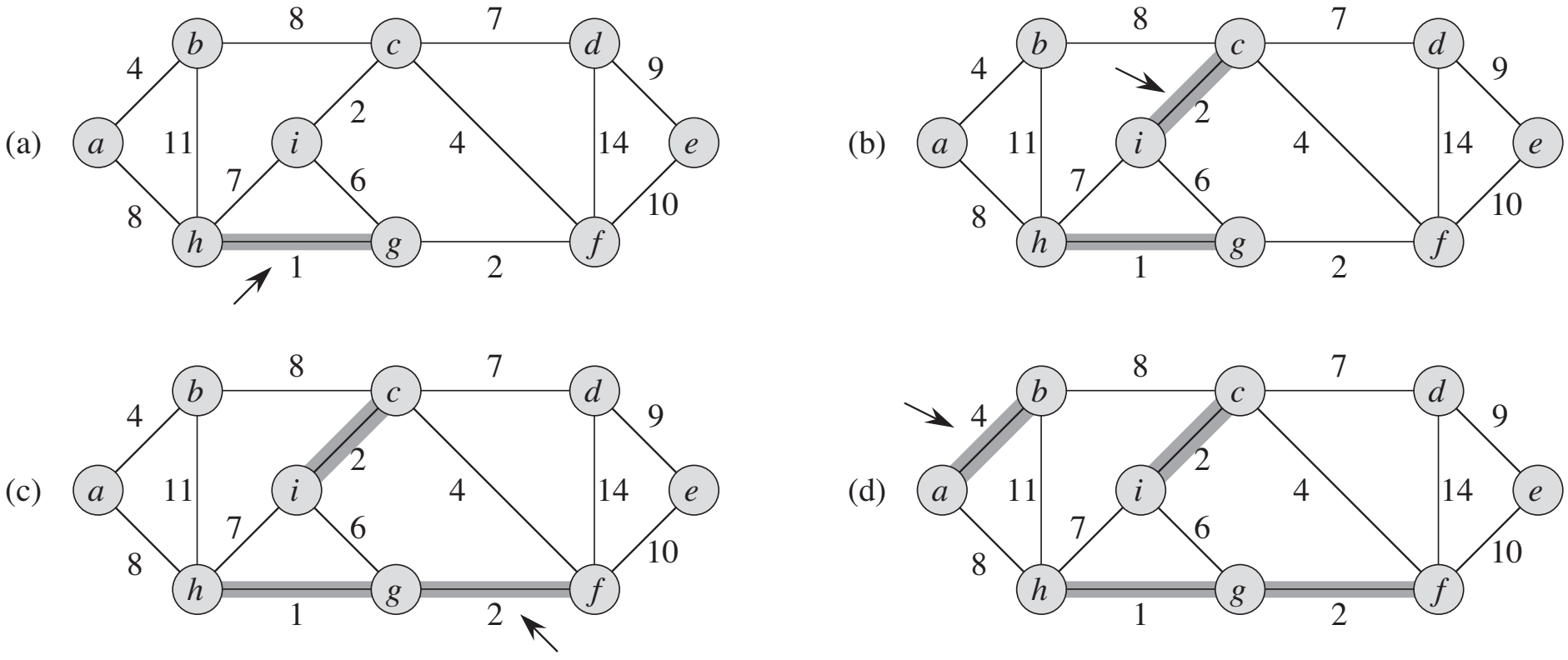
Dieses Korollar führt zum Algorithmus von Kruskal.

Algorithmus von Kruskal (Pseudo-Code)

MST-KRUSKAL(G, w)

1: $A \leftarrow \emptyset$
2: **for** each vertex $v \in G.V$
3: MAKE-SET(v)
4: Sort the edges of E by nondecreasing weight w
5: **for** each edge $(u, v) \in G.E$ in non-decreasing order by weight
6: **if** FIND-SET$(u) \neq$ FIND-SET(v)
7: $A \leftarrow A \cup \{ (u, v) \}$
8: UNION(u, v)
9: **return** A

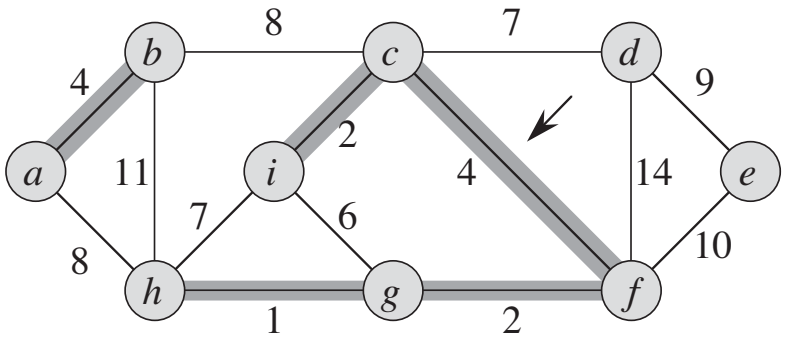
Beispiel (1)



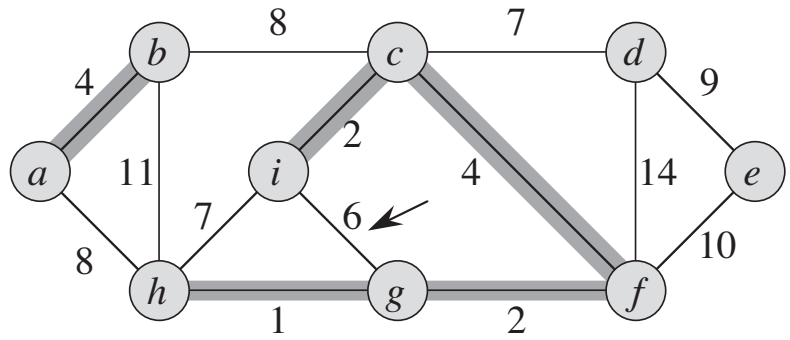
- 1, (g, h): sicher
- 2, (c, i): sicher
- 2, (f, g): sicher
- 4, (a, b): sicher
- 4, (c, f)
- 6, (i, g)
- 7, (c, d)
- 7, (h, i)
- 8, (a, h)
- 8, (b, c)
- 9, (d, e)
- 10, (e, f)
- 11, (b, h)
- 14, (d, f)

Beispiel (2)

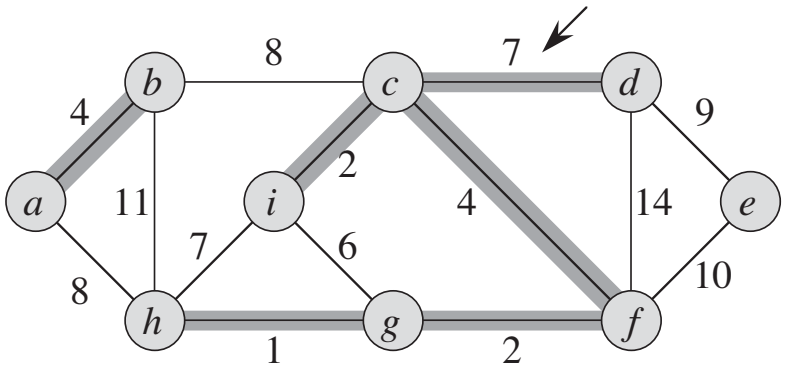
(e)



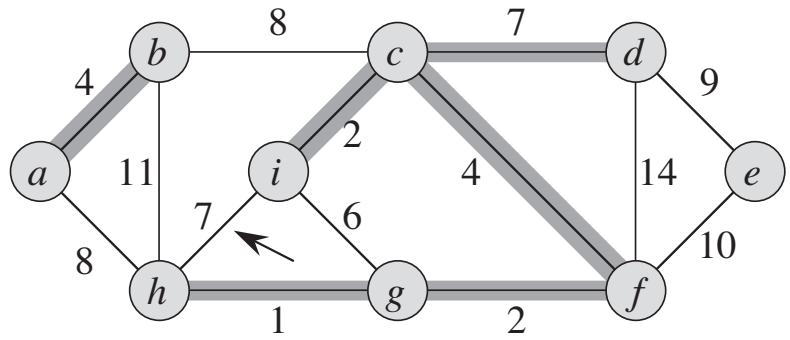
(f)



(g)



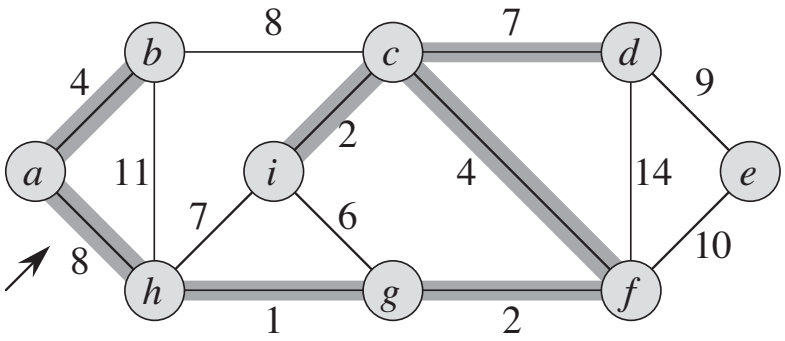
(h)



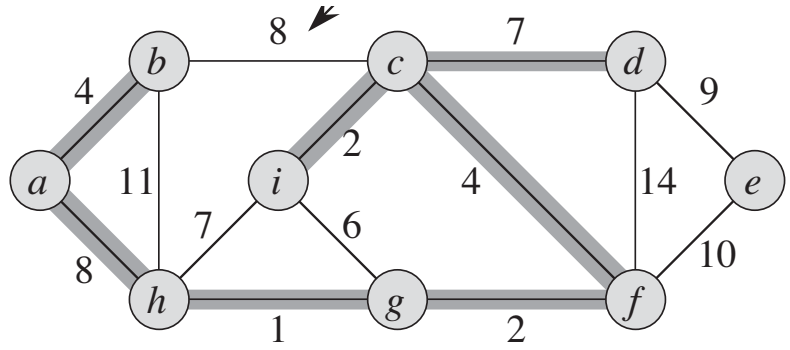
- $1, (g, h)$: sicher
- $2, (c, i)$: sicher
- $2, (f, g)$: sicher
- $4, (a, b)$: sicher
- $4, (c, f)$: sicher
- $6, (i, g)$: ablehnen
- $7, (c, d)$: sicher
- $7, (h, i)$: ablehnen
- $8, (a, h)$
- $8, (b, c)$
- $9, (d, e)$
- $10, (e, f)$
- $11, (b, h)$
- $14, (d, f)$

Beispiel (3)

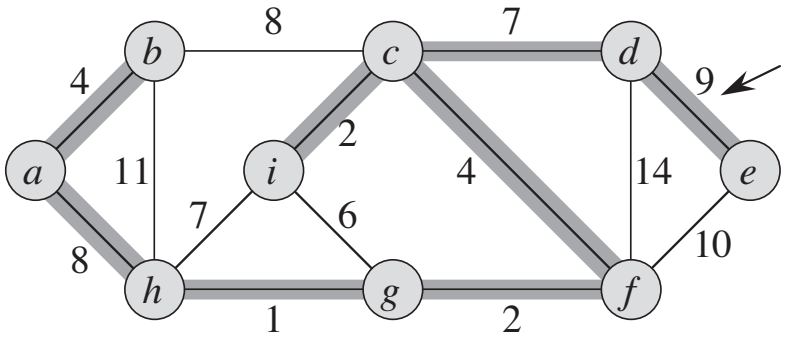
(i)



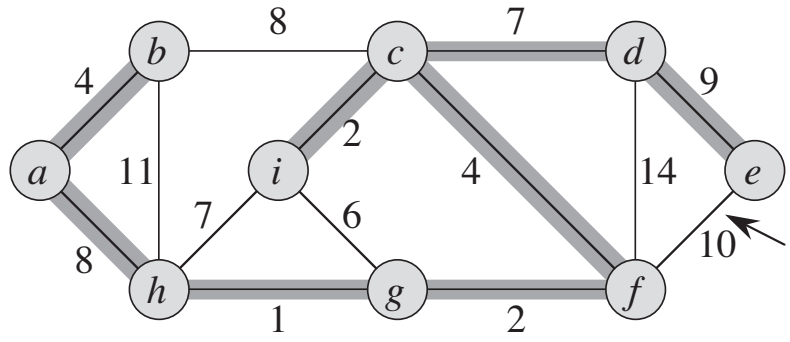
(j)



(k)



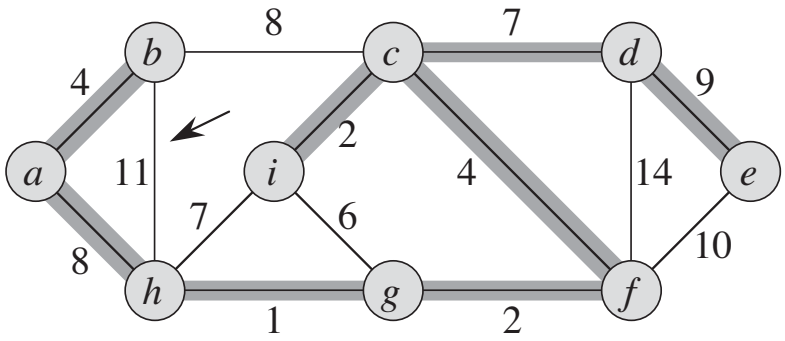
(l)



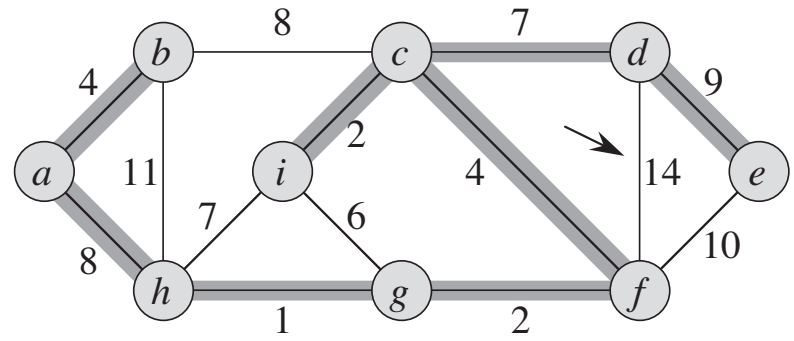
- $1, (g, h)$: sicher
- $2, (c, i)$: sicher
- $2, (f, g)$: sicher
- $4, (a, b)$: sicher
- $4, (c, f)$: sicher
- $6, (i, g)$: ablehnen
- $7, (c, d)$: sicher
- $7, (h, i)$: ablehnen
- $8, (a, h)$: sicher
- $8, (b, c)$: ablehnen
- $9, (d, e)$: sicher
- $10, (e, f)$: ablehnen
- $11, (b, h)$
- $14, (d, f)$

Beispiel (4)

(m)



(n)



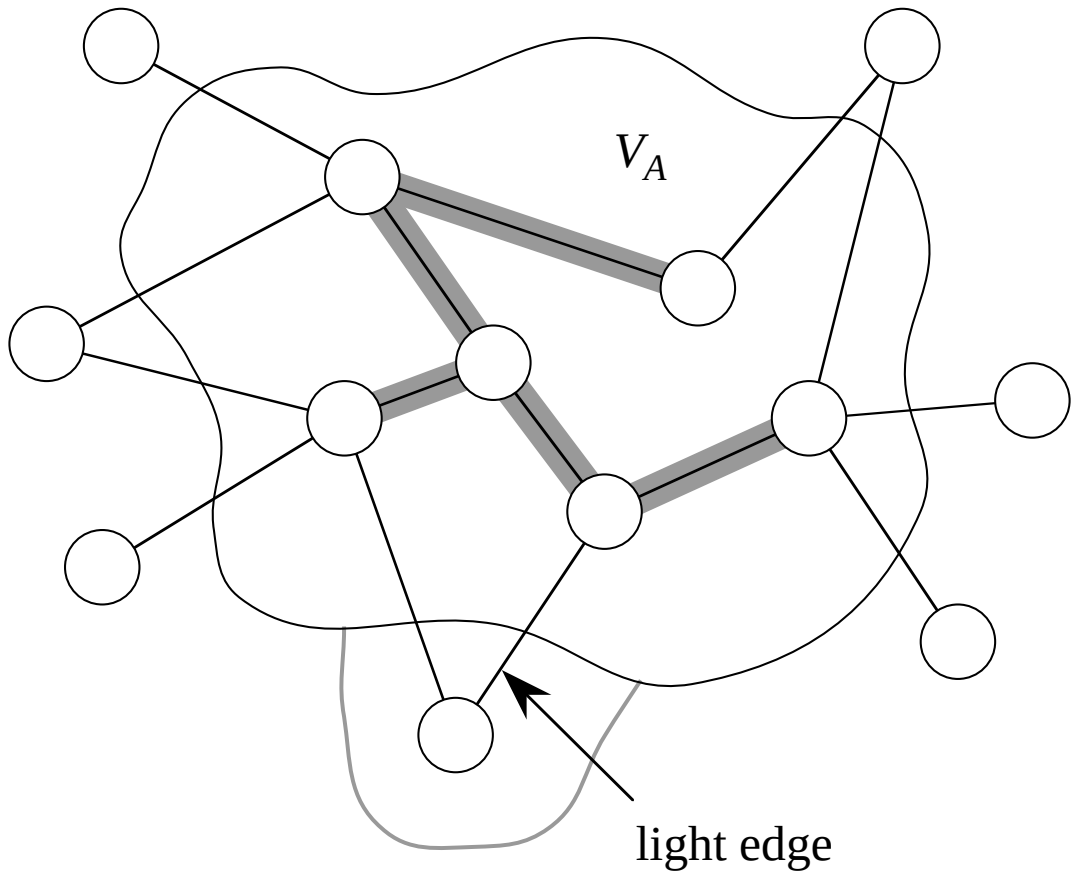
- $1, (g, h)$: sicher
- $2, (c, i)$: sicher
- $2, (f, g)$: sicher
- $4, (a, b)$: sicher
- $4, (c, f)$: sicher
- $6, (i, g)$: ablehnen
- $7, (c, d)$: sicher
- $7, (h, i)$: ablehnen
- $8, (a, h)$: sicher
- $8, (b, c)$: ablehnen
- $9, (d, e)$: sicher
- $10, (e, f)$: ablehnen
- $11, (b, h)$: ablehnen
- $14, (d, f)$: ablehnen

Laufzeit des Algorithmus von Kruskal

- Initialisierung (Z. 1): $O(1)$
- Erste Schleife (Z. 2–3):
- Sortieren (Z. 4): $O(E \log E)$
- Union-Find (Z. 2–4, Z. 5-8)
  - $O(V)$ MAKE-SET-Operationen
  - $O(E)$ FIND-SET-Operationen
  - $O(V)$ UNION-Operationen
  - Da G verbunden gilt $|E| \geq |V| - 1$ und somit $V = O(E)$
  - Also $O(E)$ Operationen gesamt
  - Bestmögliche Laufzeit $O(E\alpha(V))$, erreicht durch Wald mit Path Compression und Union by Rank
- Es gilt $\alpha(|V|) = O(\log V) = O(\log E)$
- Zusammen: $O(E \log E)$
- Alternativ $O(E \log V)$, denn $|E| \leq |V|^2$ und somit $\log|E| = O(\log V)$
- Bemerke: Wenn Kanten schon sortiert, dann $O(E\alpha(V))$, d.h. fast lineare Laufzeit

Algorithmus von Prim (1)

- Startet bei einer beliebigen Knoten und erstellt Baum A mit Wurzel r
- Fügt in jedem Schritt ein Kante hinzu, so dass ein bisher noch nicht erreichbarer Knoten dann von r erreicht wird



Algorithmus von Prim (2)

Wie können wir die leichte Kante schnell finden?

- Sei V_A die Menge der schon erreichten Knoten
- Alle anderen Knoten $V \setminus V_A$ in einer Prioritätswarteschlange Q verwaltet
- Für $v \in Q$ ist Schlüssel $v.key$ das minimales Gewicht einer Kante (u, v) mit $u \in V_A$
  - $v.key = \infty$ falls keine solche Kante existiert
  - Wir wählen immer die leichteste Kante $\rightarrow$ Sicher
- A wird implizit über Vorgängerzeiger $v.\pi$ gehalten

$$A = \{ (v, v.\pi) \mid v \in V \setminus \{ r \} \setminus Q \}$$

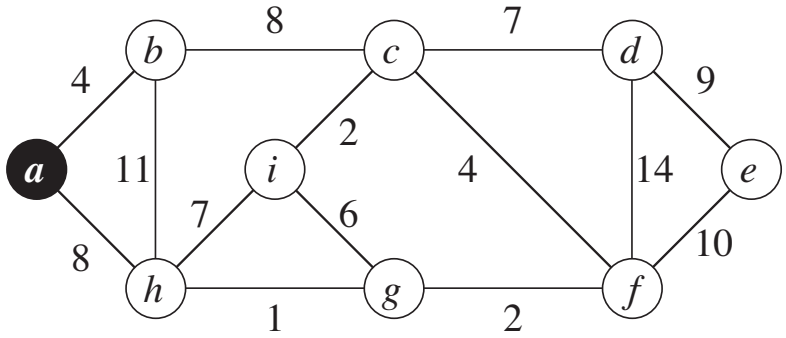
Algorithmus von Prim (Pseudo-Code)

MST-PRIM(G, w, r)

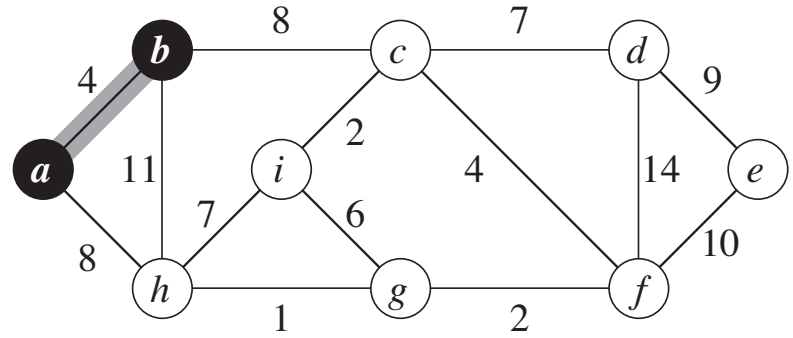
1: $Q \leftarrow \emptyset$
2: **for** each vertex $u \in G.V$
3: $\quad u.key \leftarrow \infty$
4: $\quad u.\pi = \text{NIL}$
5: $\quad$ INSERT(Q, u)
6: DECREASE-KEY$(Q, r, 0)$ // $r.key \leftarrow 0$
7: **while** $Q \neq \emptyset$
8: $\quad u \leftarrow$ EXTRACT-MIN(Q)
9: $\quad$ **for** each vertex $v \in G.Adj[u]$
10: $\quad\quad$ **if** $v \in Q$ and $w(u, v) < v.key$
11: $\quad\quad\quad v.\pi = u$
12: $\quad\quad\quad$ DECREASE-KEY$(Q, v, w(u, v))$ // $v.key \leftarrow w(u, v)$

Beispiel (1)

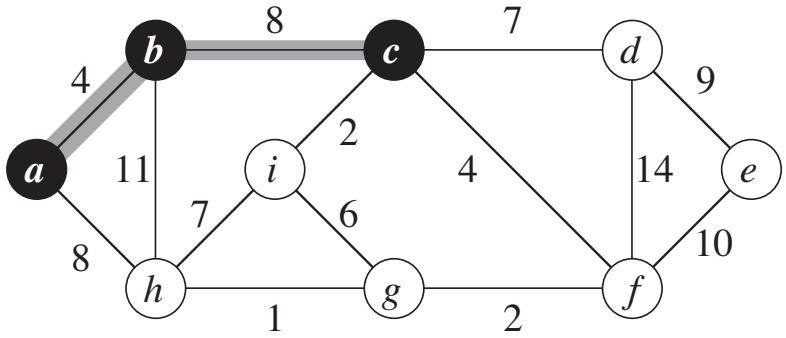
(a)



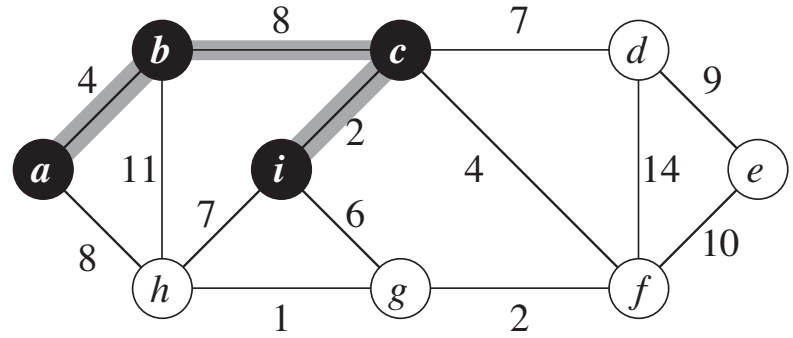
(b)



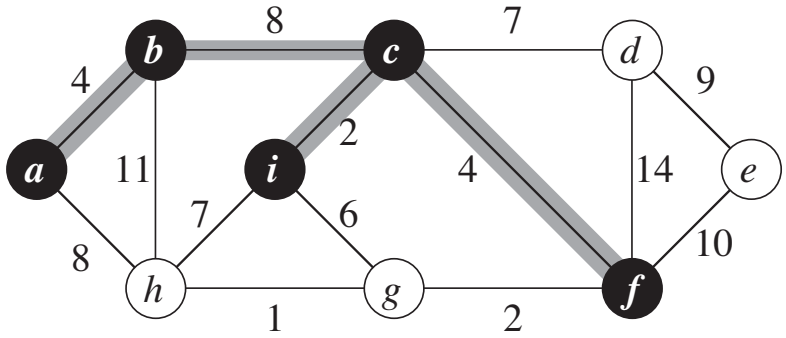
(c)



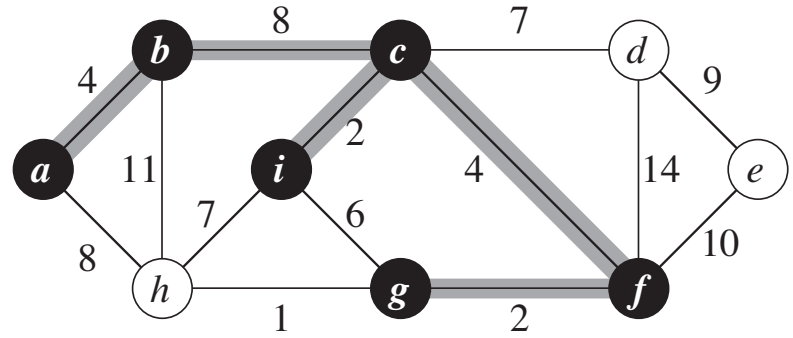
(d)



(e)

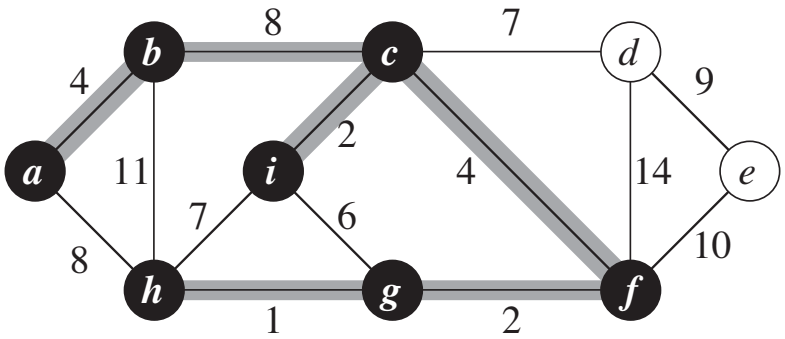


(f)

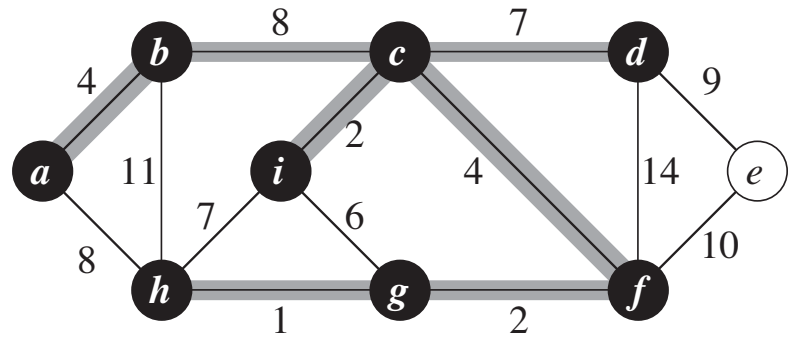


Beispiel (2)

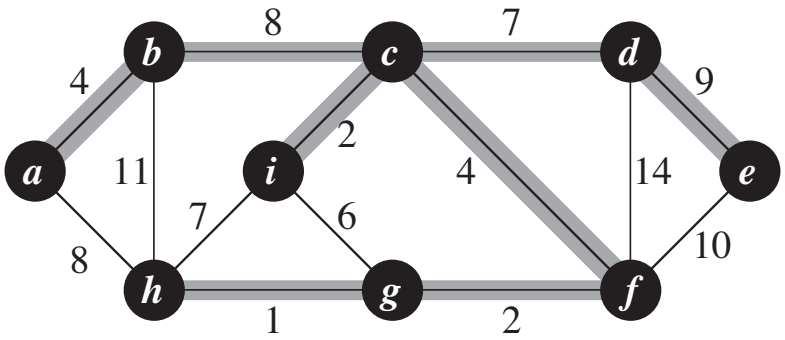
(g)



(h)



(i)



Laufzeit des Algorithmus von Prim

- Laufzeit hängt von Implementierung der Prioritätswarteschlange ab
- Mit binärem Heap
  - DECREASE-KEY kann in $O(\log V)$ umgesetzt werden (Übung)
  - Initialisierung (Z. 1–5): $O(V \log V)$
  - DECREASE-KEY von r: $O(\log V)$
  - $|V|$ EXTRACT-MIN-Operationen in **while**-Schleife: $O(V \log V)$
  - $\leq |E|$ DECREASE-KEY-Operationen in **for**-Schleife: $O(E \log V)$
  - Gesamt: $O(E \log V)$
- Mit **Fibonacci-Heaps** (hier nicht behandelt)
  - DECREASE-KEY in $O(1)$ amortisierter Zeit
  - $\leq |E|$ DECREASE-KEY-Operationen in **while**-Schleife: $O(E)$
  - Gesamt: $O(V \log V + E)$

Outline

1. Elementare Graphalgorithmen

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

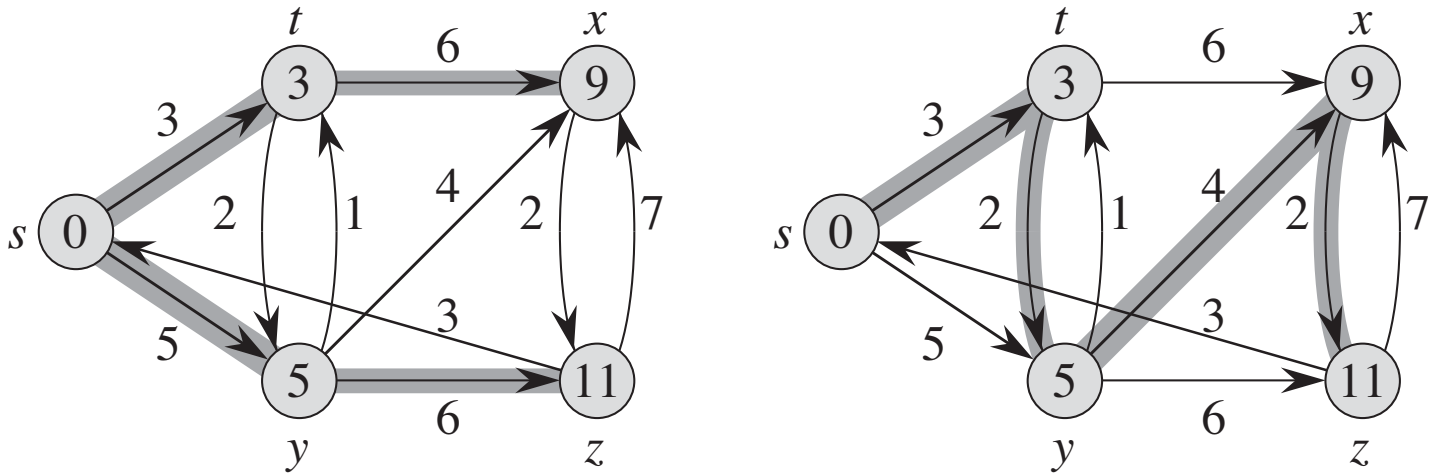
Kürzeste Pfade

- Kürzeste Pfade oft benötigt; z.B.
  - Kürzeste (oder schnellste) Reiseroute von A nach B
  - Route von Laptop zu Webserver im Internet mit minimaler Latenz
  - Minimale Anzahl von Zügen zur Lösung eines Zauberwürfels
- Gegeben: Gerichteter Graph $G = (V, E)$ und $w : E \to \mathbb{R}$
- **Gewicht** (auch: *Länge*) des Pfades $p = \langle v_0, \ldots, v_k \rangle$: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
  - Gewicht akkumuliert sich linear entlang eines Pfades
  - Soll minimiert werden
  - Zeit, Kosten, Entfernung, Verluste, Züge, ...
- **Gewicht des kürzesten Pfades** von u nach v

$$\delta(u, v) = \begin{cases} \min\{ w(p) \mid u \overset{p}{\rightsquigarrow} v \} & \text{falls } u, v \text{ verbunden} \\ \infty & \text{sonst} \end{cases}$$

- **Kürzester Pfad** ist Pfad mit diesem Gewicht (d.h., $w(p) = \delta(u, v)$)

Beispiel



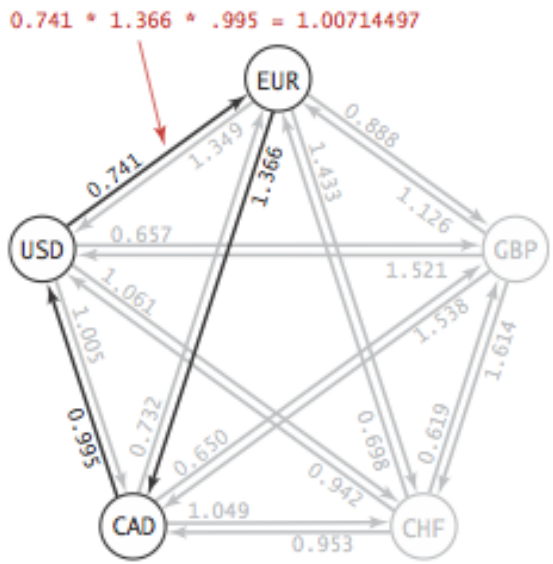
- Kürzeste Pfade sind nicht eindeutig
- **Kürzeste-Pfade-Baum** (auch: *Kürzeste-Wege-Baum*) G_π ähnlich Breitensuchebaum
  - Aber: Gewichte anstatt Anzahl Kanten
  - Ablage *eines* kürzesten Pfades von s zu jedem Knoten in $O(V)$ Speicherplatz
  - Repräsentiert mit Hilfe von Vorgängerattribut π für jeden Knoten
  - Einfacher Pfad von Wurzel s nach v in G_π entspricht kürzestem Pfad von s nach v in G

Varianten

- **Single-source** (*SSSP*)
  - Berechne kürzesten Pfad von **Startknoten** $s \in V$ zu jedem Knoten $v \in V$
  - Ausgabe ist Kürzeste-Pfade-Baum
- **Single-destination** (*SDSP*)
  - Berechne kürzesten Pfad von jedem Knoten $v \in V$ zu einem **Zielknoten** $t \in V$
  - Äquivalent SSSP auf G^T mit $s = t$
- **Single-pair** (*SPSP*)
  - Berechne kürzesten Pfad von $s \in V$ nach $t \in V$
  - Im Allgemeinen kein Algorithmus im Worst Case besser als SSSP
- **All-pairs** (*APSP*)
  - Berechne kürzesten Pfad von u nach v für alle $u, v \in V$
  - Naiv: $|V|$ SSSP-Probleme; kann aber besser gelöst werden

Negative Kantengewichte (1)

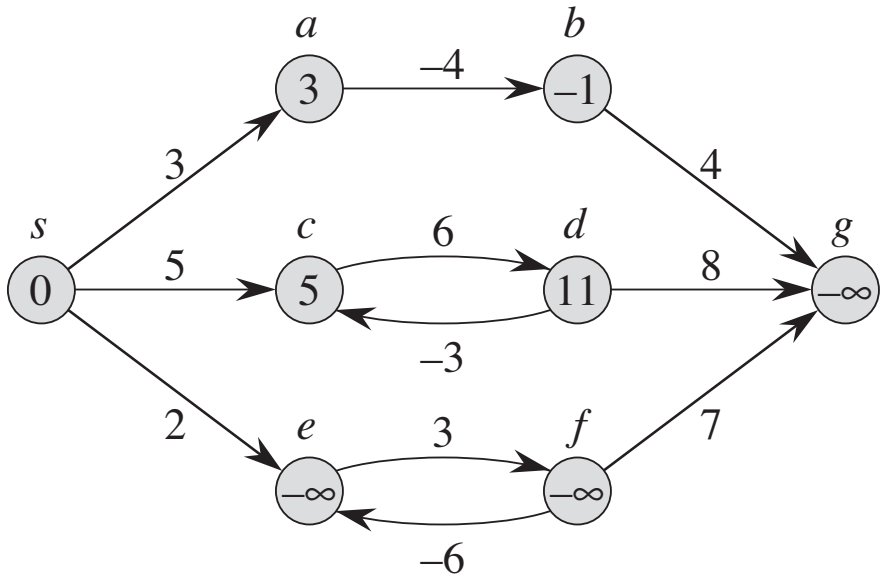
- Negative Kantengewichte treten in einigen Anwendungen auf
- Beispiel: Währungen umtauschen
  - Knoten = Währungen, Kanten = Wechselkurs
  - Pfad $\langle v_0, v_1, \ldots, v_k \rangle$ entspricht Sequenz von Währungstauschen
  - Resultierender Wechselkurs ist Produkt der Kantengewichte
  - Ziel: Maximierung des Wechselkurs bzw. Arbitrage
  - Äquivalent: Summe der negativen Logarithmen der Kantengewichte minimieren
    $\rightarrow$ Das ist ein Kürzeste-Pfade-Problem



An arbitrage opportunity

Negative Kantengewichte (2)

- Durch negative Kantengewichten können kürzere Pfade durch "Umwege" entstehen
- Negative Gewichte kein Problem solange von s kein Zyklus mit negativen Gewicht erreicht wird
  - Sonst: Zyklus kann wiederholt durchlaufen werden um beliebig kurze Pfade zu erzeugen
  - Dann setzen wir $\delta(s, v) = -\infty$; es existiert *kein* kürzester Pfad
- Nicht von allen Algorithmen unterstützt (z.B. Dijkstra: nur nicht-negative Kantengewichte)



Optimalitätsprinzip

Lemma (Optimalitätsprinzip)

Falls $p = \langle v_1, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k ist, so ist jeder Pfad $p_{ij} = \langle v_i, \ldots, v_j \rangle$ mit $1 \leq i \leq j \leq k$ ein kürzester Pfad von v_i nach v_j.

Beweis.

- Sei $q = \langle v_i, \ldots, v_j \rangle$ kürzer als p_{ij}
- Dann ist $p' = \langle v_1, \ldots, v_{i-1}, q, v_{i+1}, \ldots, v_k \rangle$ kürzer als p
  $\rightarrow$ Widerspruch

Kreisfreiheit

Lemma (Kreisfreiheit)

Sei von s kein Kreis mit negativen Gewicht erreichbar. Dann existiert für alle $v \in V$ ein einfacher (d.h. kreisfreier) kürzester Pfad von s nach v.

Beweis.

- Sei p ein kürzester Pfad von s nach v
- Wenn p kreisfrei, folgt das Lemma
- Sonst hat Kreis nicht-negatives Gewicht nach Annahme
  $\rightarrow$ Kann gestrichen werden
- Wiederholen bis p kreisfrei □

Bemerke: Ein einfacher kürzester Pfad hat höchstens $|V| - 1$ Kanten

Knotenattribute

INITIALIZE-SINGLE-SOURCE(G, s)

1: **for** each vertex $v \in G.V$
2: $\quad v.d \leftarrow \infty$
3: $\quad v.\pi \leftarrow \text{NIL}$
4: $s.d \leftarrow 0$

Nach Abarbeitung eines SSSP-Algorithmus mit Startknoten s gilt für jeden Knoten $v \in V$

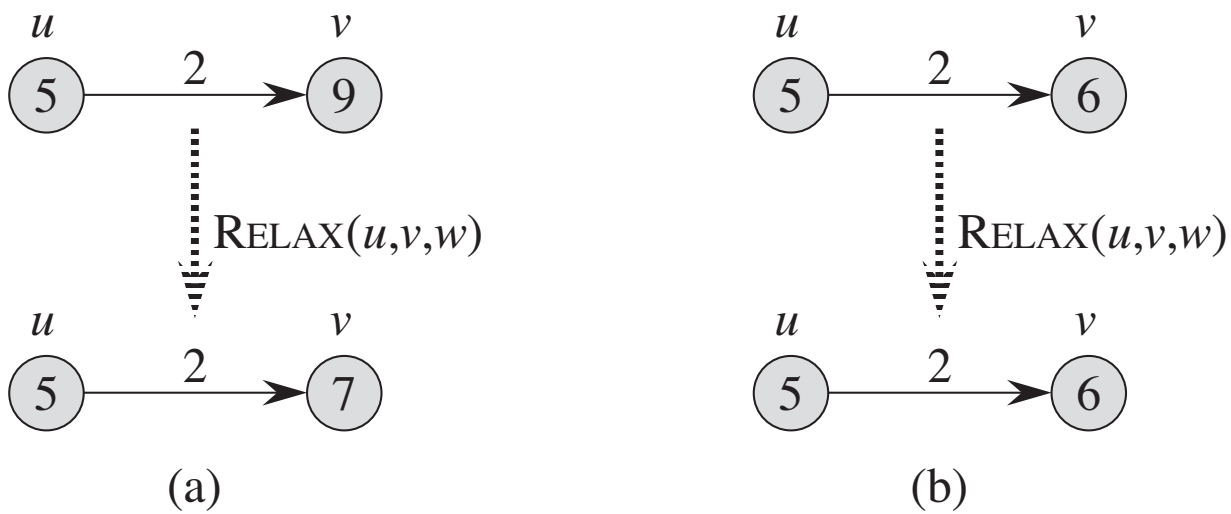
- $v.d = \delta(s, v)$
  - Initial $v.d = \infty$
  - Wird ggf. während Ausführung erniedrigt, aber immer: $v.d \geq \delta(s, v)$
  - Während Ausführung ist $v.d$ ist **Schätzung** der Pfadlänge
- $v.\pi$ ist Vorgänger von v auf kürzesten Weg von s nach v
  - Falls kein Vorgänger: $v.\pi = \text{NIL}$
  - Vorgängergraph von π ist Kürzeste-Pfade-Baum
  - Das ist möglich durch Optimalitätsprinzip und Kreisfreiheit

Relaxation

RELAX(u, v, w) // $(u, v) \in E$

1: **if** $v.d > u.d + w(u, v)$
2: $\quad v.d \leftarrow u.d + w(u, v)$
3: $\quad v.\pi \leftarrow u$

- Überprüft, ob momentane Schätzung für v verbessert werden, indem wir von s nach u gehen und dann Kante (u, v) folgen
- Falls ja $\rightarrow$ Bessere Schätzung (und Vorgänger) merken
- $O(1)$ Laufzeit
- Wir betrachten Algorithmen, die wiederholt relaxieren



Eigenschaften der Relaxation (1)

Wir rufen INITIALIZE-SINGLE-SOURCE einmal auf, gefolgt von beliebigen Relaxationen. Welche Eigenschaften gelten dann für $v.d$ und $v.\pi$?

Lemma (Dreiecksungleichung)

Sei $s \in V$ ein Startknoten. Für alle Kanten $(u, v) \in E$ gilt

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Beweis.

- Wenn u nicht von s erreichbar, $\delta(s, u) = \infty \rightarrow$ OK
- Sonst: Kürzester Pfad von s nach v kann nicht länger sein als kürzester Pfad von s nach u, gefolgt von Kante (u, v) □

Eigenschaften der Relaxation (2)

Lemma (F125, Obere Schranke)

Es gilt immer $v.d \geq \delta(s, v)$. Sobald $v.d = \delta(s, v)$, ändert sich es nicht mehr.

Beweis.

- Gilt anfangs
- Sei $v \in V$ erster Knoten für den $v.d < \delta(s, v)$ gesetzt wurde
- Sei u der Knoten durch den dies veranlasst wurde (durch $\textsc{Relax}(u, v, w)$)
- Dann gilt $v.d = u.d + w(u, v)$ und außerdem

$$\begin{aligned} v.d &< \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) && \text{(Dreiecksungleichung)} \\ &\leq u.d + w(u, v) && (v \text{ ist erste Verletzung}) \end{aligned}$$

- Somit $v.d < u.d + w(u, v) \rightarrow$ Widerspruch
- Ändert sich nicht mehr, da $v.d \geq \delta(s, v)$ und Relaxationen $v.d$ nicht erhöhen

Eigenschaften der Relaxation (3)

Lemma (Konvergenz)

Wenn $s \rightsquigarrow u \to v$ ein kürzester Pfad ist, $u.d = \delta(s,u)$ und wir $\textsc{Relax}(u,v,w)$ *aufrufen, dann gilt anschließend $v.d = \delta(s,v)$.*

Beweis.

- Nach jeder Relaxation gilt $v.d \leq u.d + w(u,v)$
- Somit $v.d \leq u.d + w(u,v) = \delta(s,u) + w(u,v) = \delta(s,v)$
- Da $v.d \geq \delta(s,v)$ folgt $v.d = \delta(s,v)$ □

Eigenschaften der Relaxation (4)

Lemma (Pfadrelaxation)

Sei $p = \langle v_0, v_1, \ldots, v_k \rangle$ ein kürzester Pfad von $s = v_0$ zu v_k. Wenn die Kanten (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) in dieser Reihenfolge relaxiert werden, dann gilt anschließend $v_k.d = \delta(s, v)$. Dies gilt auch wenn zwischendurch beliebige andere Relaxationen vorgenommen werden.

Beweis.

- Per Induktion: $v_i.d = \delta(s, v_i)$ nach Relaxation der Kanten bis (v_{i-1}, v_i)
- Induktionsanfang: $v_0.d = s.d = 0 = \delta(s, v_0)$
- Induktionsschritt
  - Sei $v_{i-1}.d = \delta(s, v_{i-1})$
  - Relaxiere Kante (v_{i-1}, v_i)
  - Dann nach Konvergenz $v_i.d = \delta(s, v_i)$ □

Eigenschaften der Relaxation (5)

Enthalte G keine negativen Zyklen, die von s erreichbar sind.

Lemma (Baumeigenschaft)

Direkt nach INITIALIZE-SINGLE-SOURCE(G, s) *ist* G_π *ein Baum mit Wurzel* s*. Diese Eigenschaft ist invariant gegenüber* RELAX*-Aufrufen.*

Lemma (F128, Kürzeste-Wege-Baum)

Gilt nach eine Folge von RELAX*-Aufrufen* $v.d = \delta(s, v)$ *für alle* $v \in V$*, dann ist* G_π *ein Kürzester-Pfade-Baum mit Wurzel* s*.*

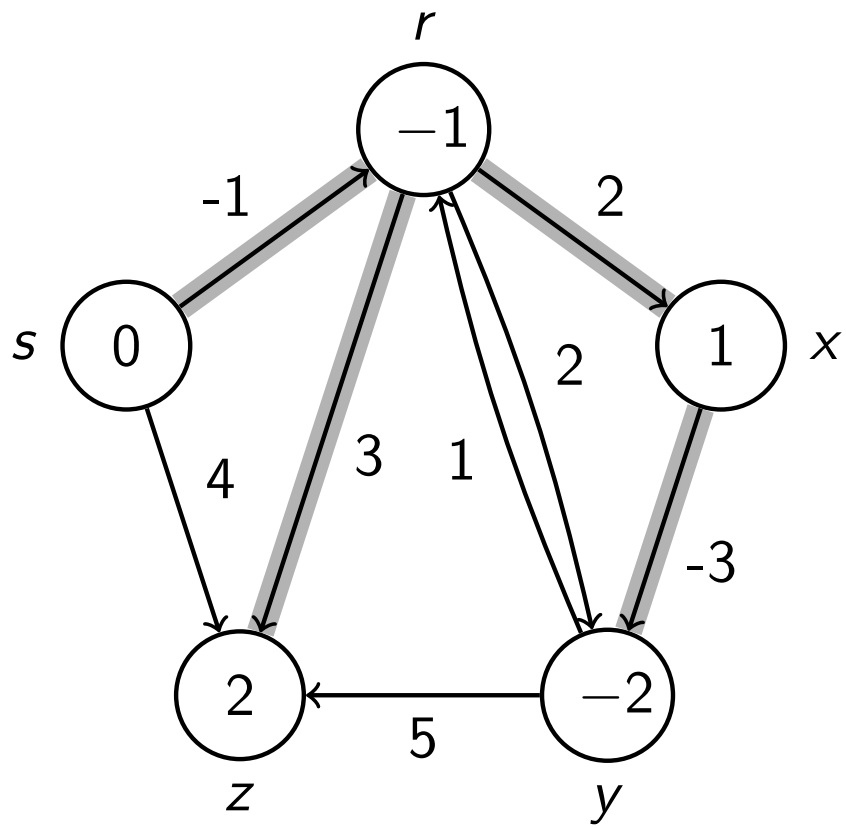
Bellman-Ford-Algorithmus

BELLMAN-FORD(G, w, s)

```
1: INITIALIZE-SINGLE-SOURCE(G, s)
2: for i ← 1 to |G.V| − 1
3:     for each edge (u, v) ∈ G.E
4:         RELAX(u, v, w)
5: for each edge (u, v) ∈ G.E
6:     if v.d > u.d + w(u, v)
7:         return FALSE
8: return TRUE
```

- Negative Kantengewichte erlaubt
- Liefert TRUE, falls kein Kreis mit negativen Gewicht von s erreichbar; sonst FALSE
- Falls TRUE, berechnet Kürzeste-Wege-Baum
- Laufzeit: $O(VE)$

Beispiel



- Konvergenzgeschwindigkeit hängt von Kantenreihenfolge ab
- Ohne Negativzyklen: keine Änderungen mehr nach $|V| - 1$ Iterationen

Korrektheit (1)

Lemma (F131)

Falls G keinen von s aus erreichbaren Zyklus mit negativem Gewicht enthält, so gilt nach Terminierung von Bellman-Ford *$v.d = \delta(s, v)$ für $v \in V$.*

Beweis.

- v von s nicht erreichbar, d.h. $\delta(s, v) = \infty$
  $\rightarrow$ Da $v.d \geq \delta(s, v)$ obere Schranke (Lemma F125), gilt $v.d = \infty$
- Sonst sei $p = \langle v_0, \ldots, v_k \rangle$ einfacher kürzester Pfad von $s = v_0$ zu $v = v_k$
- Da p kreisfrei, gilt $k \leq |V| - 1$
- Jede Kante in p wird in jeder Iteration relaxiert; insbesondere
  - Kante (v_0, v_1) in 1. Iteration
  - Kante (v_1, v_2) in 2. Iteration
  - Kante (v_{k-1}, v_k) in k-ter Iteration
- Kanten von p werden also ihrer Reihenfolge nach relaxiert
  $\rightarrow$ Pfadrelaxationseigenschaft garantiert $v_k.d = \delta(s, v)$

Korrektheit (2)

Theorem

Falls G keinen von s aus erreichbaren Zyklus mit negativem Gewicht enthält, so gibt BELLMAN-FORD TRUE *zurück, andernfalls* FALSE. *Im ersten Fall gilt zusätzlich $v.d = \delta(s, v)$ für alle $v \in V$ und G_π ist ein Kürzeste-Pfade-Baum mit Wurzel s.*

Beweis.

Fall 1: G enthält keinen Zyklus mit negativen Gewicht

- $v.d = \delta(s, v)$ nach Lemma F131
- G_π ist Kürzeste-Pfade-Baum nach Lemma F128
- Algorithmus liefert TRUE da zum Terminierungszeitpunkt

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \qquad \text{(Dreiecksungleichung)} \\ &= u.d + w(u, v), \end{aligned}$$

sodass keine der Bedingungen in Z. 6 erfüllt ist

Korrektheit (3)

Fall 2: G enthält einen von s aus erreichbaren Zyklus $c = \langle v_0, \ldots, v_k \rangle$ mit $v_0 = v_k$ mit negativem Gewicht

- Dann $w(c) = \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$
- Annahme für Widerspruch: BELLMAN-FORD gibt TRUE zurück
- Dann gilt $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ für $i = 1, \ldots, k$
- Aufsummieren dieser Ungleichungen ergibt

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Da $v_k = v_0$, gilt $\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$ und somit

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) = w(c)$$

- Widerspruch zu $w(c) < 0$ □

Algorithmus für DAGs

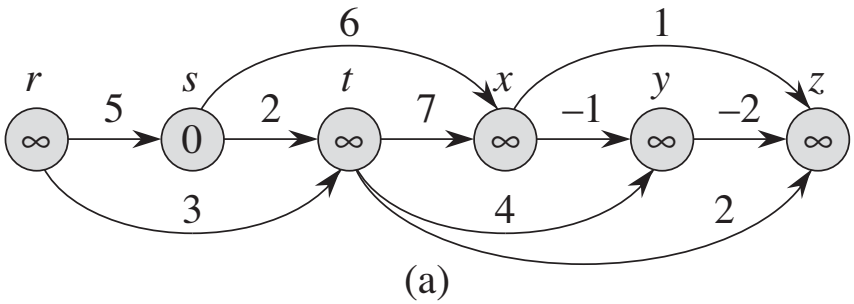
Für DAGs können wir einen effizienteren Algorithmus finden.

- Negative Zyklen können nicht auftreten
  $\rightarrow$ Kürzeste Pfade wohldefiniert
- Anwendungsbeispiel: Kritische Pfade in PERT-Diagrammen
- Laufzeit $\Theta(V + E)$

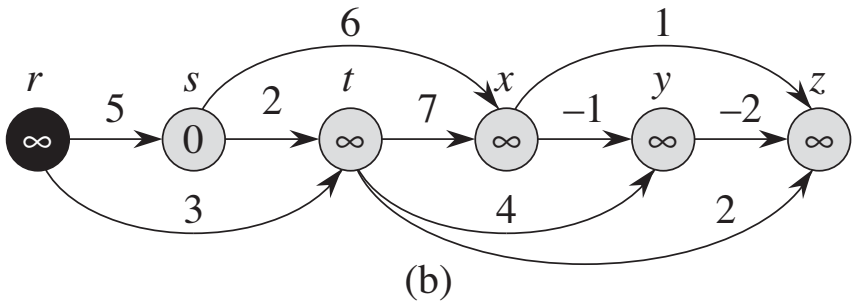
DAG-Shortest-Paths(G, w, s)

```
1: Initialize-Single-Source(G, s)
2: for each vertex u taken in topologically sorted order
3:     for each vertex v ∈ G.Adj[u]
4:         Relax(u, v, w)
```

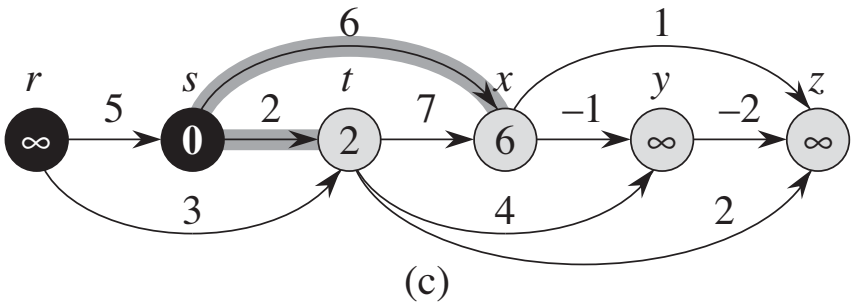
Beispiel (1)



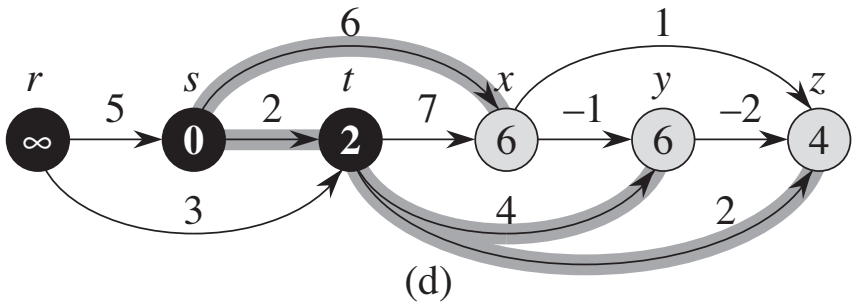
(a)



(b)

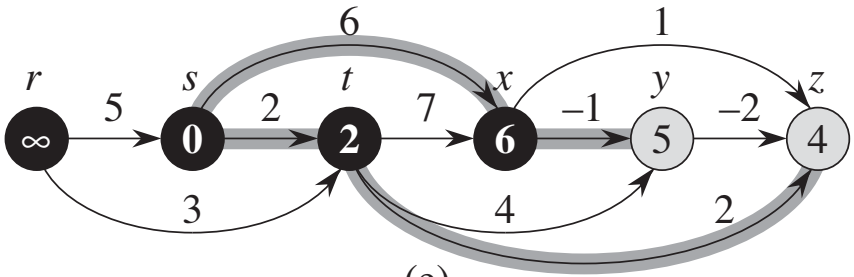


(c)

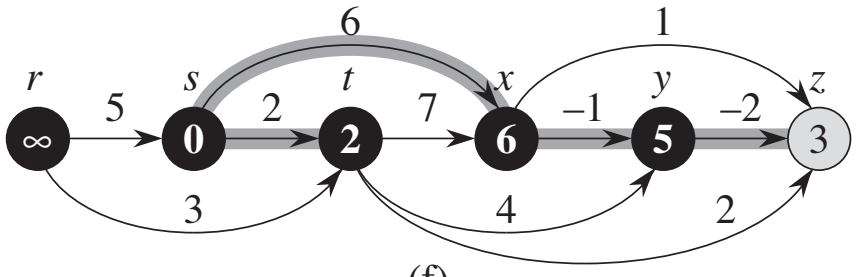


(d)

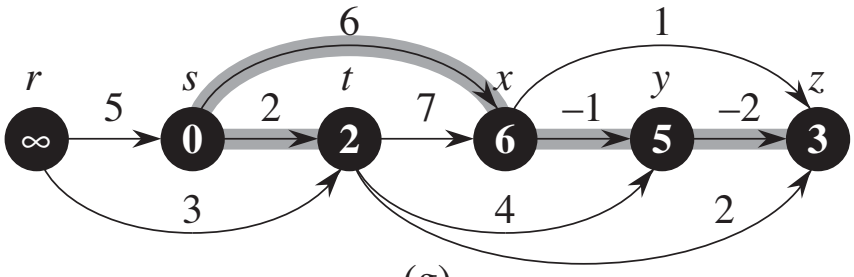
Beispiel (2)



(e)



(f)



(g)

Korrektheit

Theorem

Sei G ein DAG. Dann gilt nach Beendigung von DAG-Shortest-Paths, *dass $v.d = \delta(s, v)$ für alle $v \in V$ und G_π ist ein Kürzeste-Pfade-Baum mit Wurzel s.*

Beweis.

- Wie vorher: v von s nicht erreichbar $\rightarrow$ $v.d = \delta(s, v) = \infty$
- Sonst sei $p = \langle v_0, \ldots, v_k \rangle$ ein kürzester Pfad mit $v_0 = s$ und $v_k = v$
- Da Knoten topologisch sortiert, werden diese Kanten in der Reihenfolge

  $$(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$$

  relaxiert
- $v.d = \delta(s, v)$ nach Pfadrelaxionseigenschaft (und da $v.d \geq \delta(s, v)$)
- G_π ist Kürzeste-Pfade-Baum nach Lemma F128 □

Dijkstras Algorithmus

- Keine negativen Kantengewichte erlaubt
- Entspricht gewichteter Version der Breitensuche
  - Verwendet Prioritätswarteschlange anstelle einer FIFO-Warteschlange
  - Schlüssel sind Abstände $v.d$
- Verwaltet zwei Schlüsselmengen
  - S: Knoten u, für die $\delta(s, u)$ schon ermittelt
  - Q: Prioritätswarteschlange mit Elementen $V \setminus S$
- Somit: In jedem Schritt wird Knoten aus Q gewählt, der "am nächsten" zu s liegt

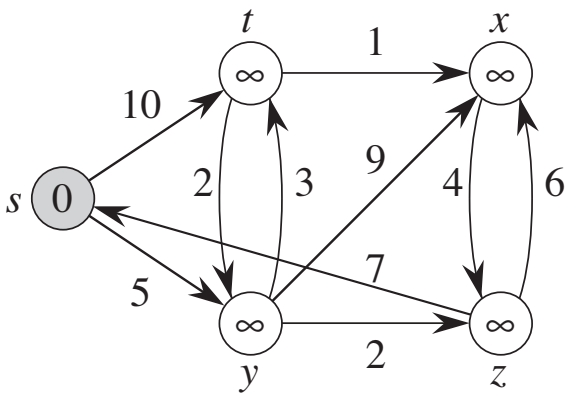
Dijkstras Algorithmus (Pseudo-Code)

DIJKSTRA(G, w, s)

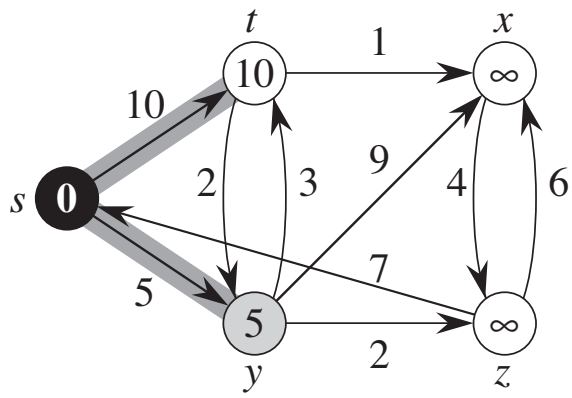
1: INITIALIZE-SINGLE-SOURCE(G, s)
2: $S \leftarrow \emptyset$
3: $Q \leftarrow G.V$
4: **while** $Q \neq \emptyset$
5: $\quad u \leftarrow$EXTRACT-MIN(Q)
6: $\quad S \leftarrow S \cup \{ u \}$
7: $\quad$ **for** each vertex $v \in G.Adj[u]$
8: $\quad\quad$ RELAX(u, v, w)

- Wird Gewicht $v.d$ während Relaxation reduziert, muss Prioritätswarteschlange aktualisiert werden (mittels DECREASE-KEY)

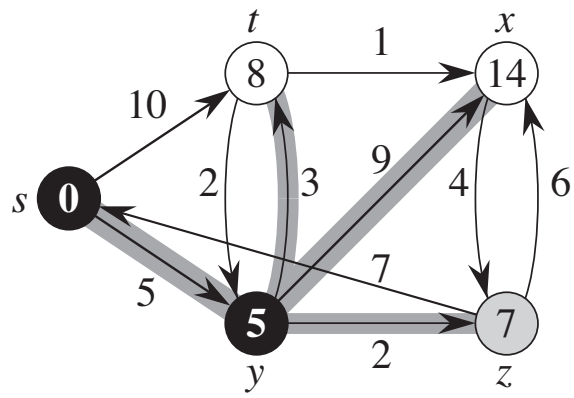
Beispiel



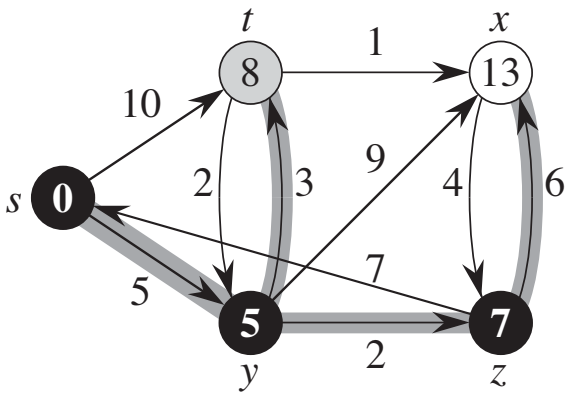
(a)



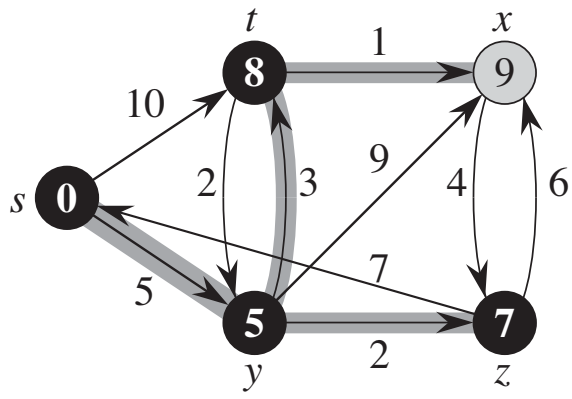
(b)



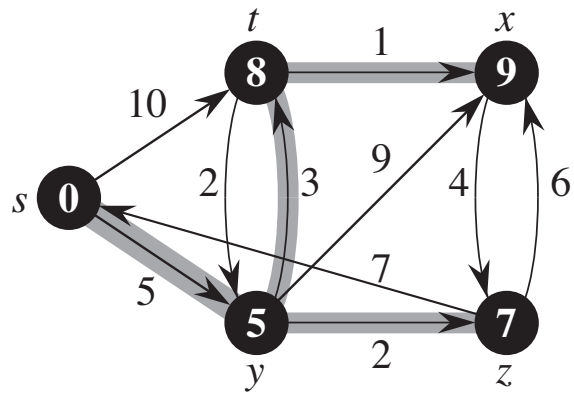
(c)



(d)



(e)



(f)

Laufzeit

- Ohne Warten von Q: $O(V + E)$
- Warten von Q
  - Initialisierung (Z. 3)
  - $|V|$ EXTRACT-MIN-Operationen (Z. 5)
  - $\leq E$ DECREASE-KEY-Operationen (Z. 8)
  - Laufzeit hängt von Implementierung der Warteschlange ab
- Q als Array: $O(V^2 + E)$
- Q als Heap: $O((V + E) \log V)$
- Q als Fibonacci Heap: $O(V \log V + E)$

Korrektheit (1)

Theorem

Enthalte G keinen von s aus erreichbaren Zyklus mit negativem Gewicht. Dann gilt nach Beendigung von DIJKSTRA, *dass $v.d = \delta(s, v)$ für alle $v \in V$ und G_π ist ein Kürzeste-Pfade-Baum mit Wurzel s.*

Beweis.

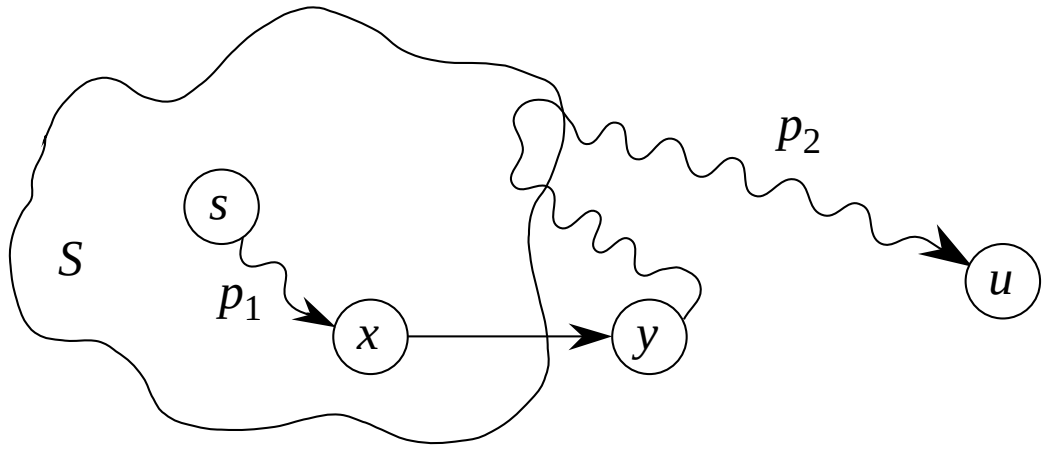
- **Invariante:** Zu Beginn jeder Iteration der **while**-Schleife gilt $v.d = \delta(s, v)$ für alle $v \in S$.
- Dadurch bei **Terminierung**
  - $Q = \emptyset \Rightarrow S = V \Rightarrow v.d = \delta(s, v)$ für alle $v \in V$
  - Somit G_π Kürzeste-Pfade-Baum nach Lemma F128
- Zu Zeigen: Invariante gilt

Korrektheit (2)

- **Invariante:** Zu Beginn jeder Iteration der **while**-Schleife gilt $v.d = \delta(s, v)$ für alle $v \in S$.
- **Induktionsanfang**
  - $S = \emptyset$ initial, Invariante gilt trivial
- **Induktionsschritt**
  - Für Widerspruch: Sei u erster Knoten für den $u.d \neq \delta(s, u)$ zum Zeitpunkt des Hinzufügens von u zu S
  - $u \neq s$, da $s.d = 0 = \delta(s, s) \Rightarrow S \neq \emptyset$
  - Es gibt einen Pfad $s \rightsquigarrow u$
    - Sonst wäre $\delta(s, u) = \infty$
    - Und da $u.d \geq \delta(s, u)$ somit auch $u.d = \infty = \delta(s, u)$
  - Aber dann gibt es auch einen kürzesten Pfad $s \overset{p}{\rightsquigarrow} u$

Korrektheit (3)

- Sei $y \in p$ erster Knoten mit $y \notin S$ und x dessen Vorgänger in p



- $p = s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$
- Dann: $y.d = \delta(s, y)$ wenn u zu S hinzugefügt wird
  - $x.d = \delta(s, x)$ als x zu S hinzugefügt wurde (da u erste Verletzung)
  - Dabei (x, y) relaxiert, somit $y.d = \delta(s, y)$ durch Konvergenz
- Es gilt $\delta(s, y) \leq \delta(s, u)$ da Kantengewichte nicht-negativ
- Zusammen: $y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$ nach Lemma F125
- Außerdem $u.d \leq y.d$ da u gewählt und $u, y \notin S$
- Aber dann $y.d = \delta(s, y) = \delta(s, u) = u.d$
- Widerspruch zu $u.d \neq \delta(s, u)$ □

Outline

1. Elementare Graphalgorithmen

2. Minimale Spannbäume

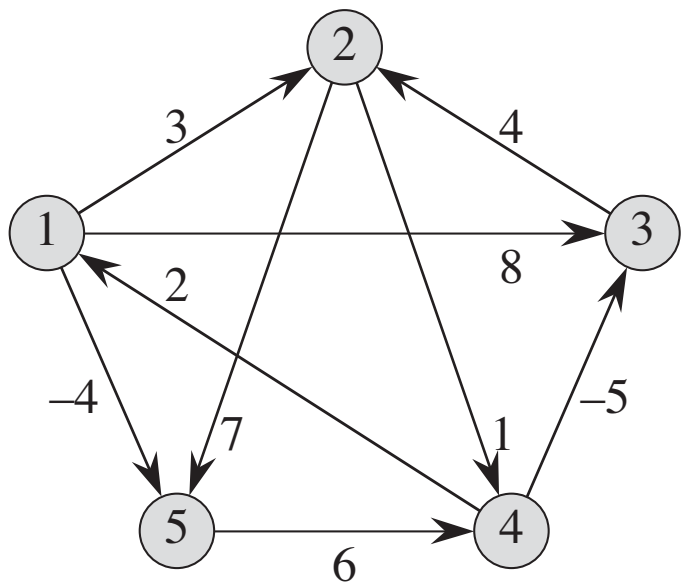
3. Kürzeste Wege

4. Alle kürzesten Wege

Alle kürzesten Pfade

Wir behandeln jetzt Algorithmen, die für alle Paare von Knoten den kürzesten Pfad zwischen diesen Knoten findet (APSP).

- Gegeben: Gerichteter Graph $G = (V, E)$ und $w : E \to \mathbb{R}$
  - Knoten nummeriert von $1, 2, \ldots, n$ mit $n = |V|$
- Gesucht
  - $n \times n$ **Entfernungsmatrix** D sodass $d_{ij} = \delta(i, j)$



$$D = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Lösung mit SSSP-Algorithmen

Wir können einen Single Source Shortest Path-Algorithmus auf jeden Knoten einmal anwenden, um APSP zu lösen.

- Bellman-Ford
  - $O(V^2E)$, für dichte Graphen $O(V^4)$
- Falls keine negativen Kantengewichte, Dijkstra
  - Array-Implementierung: $O(V^3 + VE) = O(V^3)$
  - Binärer Heap-Implementierung: $O(EV \log V)$, für dichte Graphen $O(V^3 \log V)$
  - Fibonacci Heap-Implementierung: $O(V^2 \log V + EV)$, für dichte Graphen $O(V^3)$

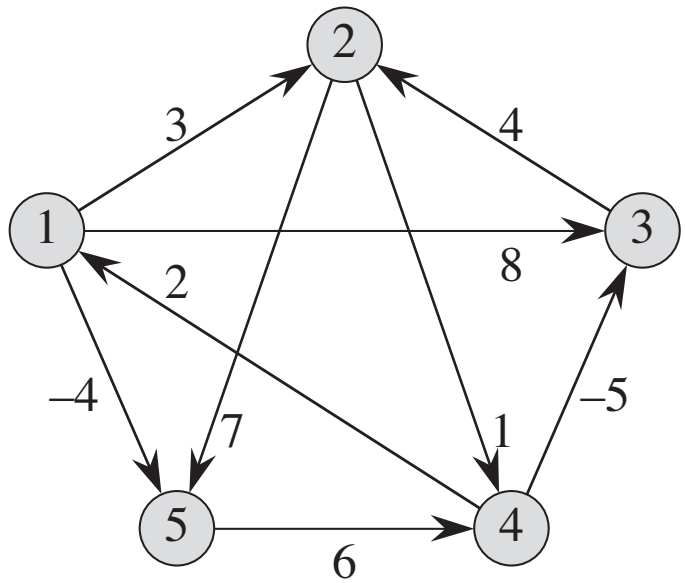
Wir betrachten im folgenden Algorithmen mit $O(V^3)$ Laufzeit in allen Fällen.

- Annahme: Keine negativen Zyklen

Vorgängermatrix (1)

Zur Repräsentation der kürzesten Pfade verwenden wir eine **Vorgängermatrix Π**:

- $\pi_{ij} = \text{NIL}$ falls $i = j$ oder kein Pfad von i nach j existiert
- Sonst π_{ij} ist Vorgänger von j auf kürzestem Pfad von i nach j
  - Wohldefiniert nach Optimalitätsprinzip
- Somit: Kürzeste-Wege-Baum mit Wurzel i in i-ter Zeile
  - Vorgängerteilgraph $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$
  - $V_{\pi,i} = \{ j \in V \mid \pi_{ij} \neq \text{NIL} \} \cup \{ i \}$
  - $E_{\pi,i} = \{ (\pi_{ij}, j) \mid j \in V_{\pi,i}, \pi_{ij} \neq \text{NIL} \}$



$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Vorgängermatrix (2)

Mit folgender Prozedur kann der kürzeste Pfad von i nach j ausgeben werden.

PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)

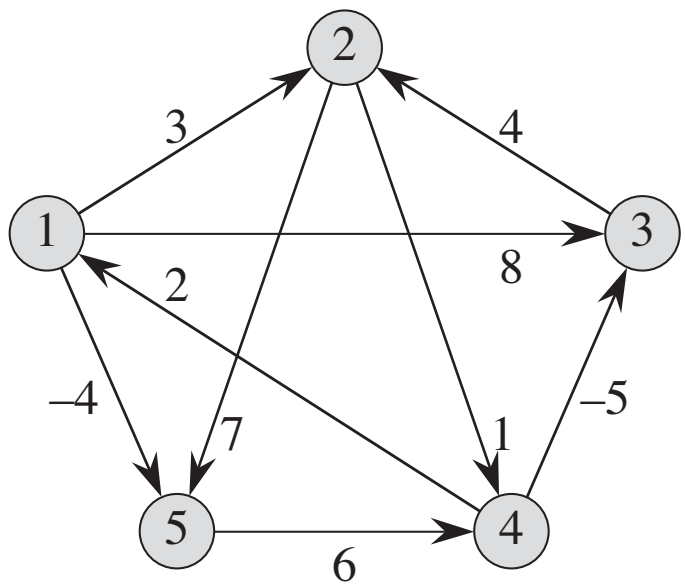
if $i = j$
    Print i
else if $\pi_{ij} = \text{NIL}$
    Print "no path from" i "to" j "exists"
else
    PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_{ij})
    Print j

- Wir betrachten im Folgenden nur die Berechnung von D
- Gegeben D kann Π in $O(V^3)$ berechnet werden (Übung)
- Ebenso kann die Berechnung von Π direkt in APSP-Algorithmen integriert werden (Übung)

Adjazenzmatrix

Viele APSP-Algorithmen repräsentieren Graphen $G = (V, E)$ mittels einer Adjazenzmatrix W anstatt einer Adjazenzliste

$$w_{ij} = \begin{cases} 0 & \text{wenn } i = j \\ w(i,j) & \text{wenn } i \neq j \text{ und } (i,j) \in E \\ \infty & \text{wenn } i \neq j \text{ und } (i,j) \notin E \end{cases}$$



$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Kürzeste Pfade und Matrizenmultiplikation

Der erste Ansatz zur Berechnung aller kürzesten Pfade beruht auf **dynamischem Programmieren**.

Vorgehensweise

1. Charakterisierung der Struktur einer optimalen Lösung
2. Rekursive Definition des Wertes der optimalen Lösung
3. Berechnung des optimalen Wertes (typischerweise Bottom-Up)
4. Berechnung der optimalen Lösung aus so berechneten Informationen

Problemstruktur (1)

- Optimalitätsprinzip: Teilpfade eines kürzesten Pfades sind wieder kürzeste Pfade
- Somit ist optimale Lösung aus optimalen Teillösungen zusammengesetzt
- Sei p ein kürzester Pfad mit k Knoten; p enthält
  - $k-1$ optimale Teilpfade mit einer Kante
  - $k-2$ optimale Teilpfade mit zwei Kanten
  - ...
  - 1 optimaler Teilpfad mit k Kanten
- Wir werden Teilprobleme (wenige Kanten) zuerst lösen und dann weiterverwenden (mehr Kanten)
- Falls es keine Zyklen mit negativem Gewicht gibt, so besteht jeder kürzeste Pfad aus $\leq n-1$ Kanten

Problemstruktur (2)

- Sei $l_{ij}^{(m)}$ das kleinste Gewicht eines Pfades von $i \rightsquigarrow j$ mit höchstens m Kanten
  - Es ist möglich, dass ein kürzerer Pfad mit mehr Kanten existiert
- Sei $p_{ij}^{(m)}$ ein solcher Pfad
- Beispiel: $m = 0$
  - Kürzester Pfad mit ≤ 0 Kanten existiert nur, wenn $i = j$

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{wenn } i = j \\ \infty & \text{wenn } i \neq j \end{cases}$$

- Beispiel: $m = 1$
  - Kürzester Pfad mit ≤ 1 Kanten $\rightarrow$ $i = j$ oder $(i, j) \in E$

$$l_{ij}^{(1)} = \begin{cases} 0 & \text{wenn } i = j \\ w_{ij} & \text{wenn } (i, j) \in E \\ \infty & \text{sonst} \end{cases}$$

  - Alternativ: $l_{ij}^{(1)} = \min\{ l_{ij}^{(0)}, w_{ij} \} = w_{ij}$

Problemstruktur (3)

- Nehmen wir an, wir kennen die Gewichte $l_{ij}^{(m-1)}$
- Können wir dann $l_{ij}^{(m)}$ berechnen?
- Betrachte kürzesten Pfad $p_{ij}^{(m)}$
  - Möglichkeit 1: $p_{ij}^{(m)}$ hat $< m$ Kanten, dann $l_{ij}^{(m)} = l_{ij}^{(m-1)}$
  - Möglichkeit 2: Hat m Kanten, dann existiert ein Vorgängerknoten k sodass

$$p_{ij}^{(m)} = i \overset{p_{ik}^{(m-1)}}{\rightsquigarrow} k \to j$$

  und somit $l_{ij}^{(m)} = l_{ik}^{(m-1)} + w_{kj}$
- Zusammen

$$\begin{aligned} l_{ij}^{(m)} &= \min\left[l_{ij}^{(m-1)}, \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\}\right] \\ &= \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\} \end{aligned}$$

Rekursive Lösung (1)

Wir schreiben die Gewichte $l_{ij}^{(m)}$ in eine Matrix L. Die folgende Prozedur implementiert dann eine Iteration zur Berechnung von $L^{(m)}$ aus $L^{(m-1)}$ und W.

EXTEND(L, W, n)

1: Let L' be a new $n \times n$ matrix
2: **for** $i \leftarrow 1$ **to** n
3: **for** $j \leftarrow 1$ **to** n
4: $l'_{ij} \leftarrow \infty$
5: **for** $k \leftarrow 1$ **to** n
6: $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$
7: **return** L'

- Laufzeit $O(V^3)$

Rekursive Lösung (2)

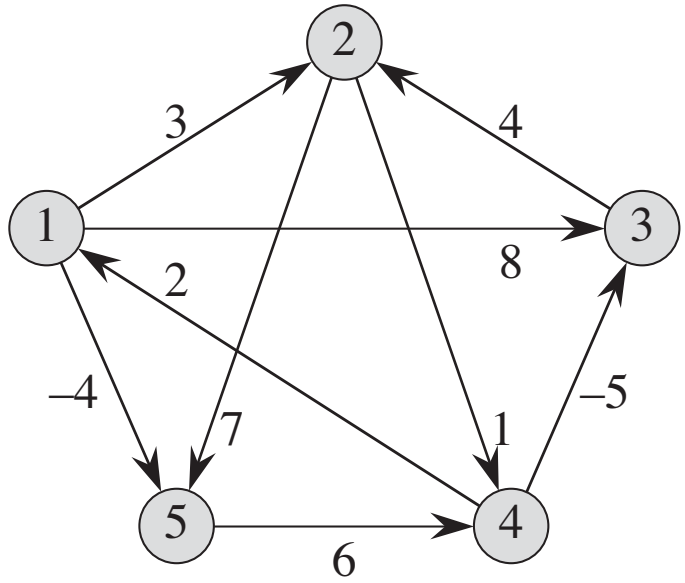
Wir beginnen mit $L^{(1)} = W$ und berechnen $L^{(2)}, L^{(3)}, \ldots, L^{(n-1)}$ schrittweise.

SLOW-APSP(W, n)

1: $L^{(1)} \leftarrow W$
2: **for** $m \leftarrow 2$ **to** $n - 1$
3: $\quad L^{(m)} \leftarrow \text{EXTEND}(L^{(m-1)}, W, n)$
return $L^{(n-1)}$

- Laufzeitkomplexität: $O(V^4)$
- Bemerke: $D = L^{(n-1)}$ falls G keine Zyklen mit negativem Gewicht enthält

Beispiel



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Ähnlichkeit mit Matrixmultiplikation (1)

Mit den Substitutionen

$$\begin{aligned} L^{(m-1)} &\to A \\ W &\to B \\ L^{(m)} &\to C \\ \min &\to + \\ + &\to * \\ \infty &\to 0 \end{aligned}$$

wird EXTEND zu einem Algorithmus zur Multiplikation zweier $n \times n$ Matrizen A und B mit Ergebnis C.

Ähnlichkeit mit Matrixmultiplikation (2)

SQUARE-MATRIX-MULTIPLY(A, B, n)

1: Let C be a new $n \times n$ matrix
2: **for** $i \leftarrow 1$ **to** n
3: **for** $j \leftarrow 1$ **to** n
4: $c_{ij} \leftarrow 0$
5: **for** $k \leftarrow 1$ **to** n
6: $c_{ij} \leftarrow l_{ij} + l_{ik} \cdot w_{kj}$
7: **return** L'

- Laufzeitkomplexität: $O(V^3)$

Beschleunigung (1)

- Wir benötigen nur $L^{(n-1)}$, nicht aber $L^{(1)}, \ldots L^{(n-2)}$
- Sei $A \cdot B$ das "Matrixprodukt", das EXTEND berechnet
- SLOW-APSP berechnet somit

$$
\begin{array}{lcll}
L^{(1)} & = & L^{(0)} \cdot W & = & W \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^2 \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^3 \\
\ldots & & & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1}
\end{array}
$$

Beschleunigung (2)

- Matrixmultiplikation ist assoziativ, unser EXTEND-Matrixprodukt ebenso (Übung)
- Ausnutzen durch **fortgesetztes Quadrieren**

$$
\begin{array}{lllll}
L^{(1)} & = & W & & \\
L^{(2)} & = & W^2 & = & W \cdot W \\
L^{(4)} & = & W^4 & = & W^2 \cdot W^2 \\
L^{(8)} & = & W^8 & = & W^4 \cdot W^4 \\
\ldots & & & & \\
L^{(2^r)} & = & W^{(2^r)} & = & L^{(2^{r-1})} \cdot L^{(2^{r-1})}
\end{array}
$$

- 2^r ist kleinste Zweierpotenz $\geq n-1$
  - $r = \lceil \lg(n-1) \rceil$
- Falls keine negativen Zyklen: $L^{(m)} = L^{(n-1)}$ für $m \geq n-1$
  - Somit $L^{(2^r)} = D$

Beschleunigung (3)

FASTER-APSP(W, n)
  $L^{(1)} \leftarrow W$
  $m \leftarrow 1$
  while $m < n - 1$
    $L^{(2m)} \leftarrow \text{EXTEND}(L^{(m)}, L^{(m)})$
    $m \leftarrow 2m$
  return $L^{(m)}$

- Laufzeitkomplexität: $O(V^3 \log V)$
- Möglich: überall L verwenden (ohne Hochstellungen)

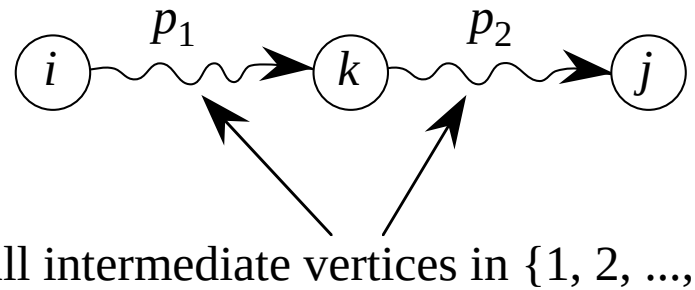
Floyd-Warshall-Algorithmus

Der **Floyd-Warshall-Algorithmus** ist ein anderer Ansatz basierend auf dynamischer Programmierung.

- Betrachtet eine andere Substruktur
- Basiert auf **Zwischenknoten** eines Pfades
  - Pfad $p = \langle v_1, \ldots, v_n \rangle$
  - Zwischenknoten sind $p = v_2, \ldots, v_{n-1}$
- Idee: Schrittweise mehr und mehr Zwischenknoten betrachten

Problemstruktur

- Sei $d_{ij}^{(k)}$ das kleinste Gewicht eines Pfades $i \overset{p}{\rightsquigarrow} j$, der nur Zwischenknoten aus $\{1, \ldots, k\}$ verwendet
- Wie ist p aus optimalen Pfaden mit Zwischenknoten aus $\{1, \ldots, k-1\}$ zusammengesetzt?
- Fall 1: k ist kein Zwischenknoten in p
  - Alle Zwischenknoten in p sind somit in $\{1, \ldots, k-1\}$
  - Kürzester Pfad mit Zwischenknoten in $\{1, \ldots, k-1\}$ ist somit auch kürzester Pfad mit Zwischenknoten in $\{1, \ldots, k\}$
- Fall 2: k ist Zwischenknoten in p
  - Dann $p = i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$
  - Da p kreisfrei: Pfade in p_1 und p_2 haben alle Zwischenknoten in $\{1, \ldots, k-1\}$



Rekursive Lösung (1)

Sei $d_{ij}^{(k)}$ das kleinste Gewicht eines Pfades von i nach j, der nur Zwischenknoten aus $\{1, \ldots, k\}$ verwendet.

- Dann gilt

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{wenn } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{wenn } k > 0 \end{cases}$$

- Außerdem $d_{ij}^{(n)} = \delta(i, j)$

Rekursive Lösung (2)

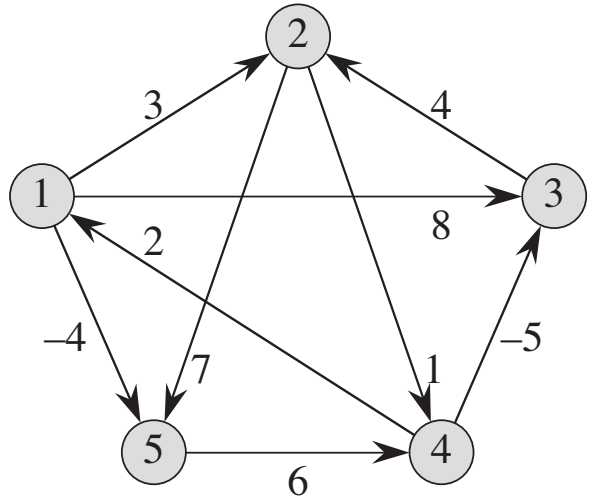
- Wir legen $d_{ij}^{(k)}$ wieder in einer $n \times n$ Matrix $D^{(k)}$ ab
- Und berechnen $W = D^{(0)}, D^{(1)}, \ldots, D^{(n)}$

FLOYD-WARSHALL(W, n)

$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
  for $i \leftarrow 1$ **to** n
    for $j \leftarrow 1$ **to** n
      $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
return $D^{(n)}$

- Laufzeitkomplexität: $O(V^3)$
- Möglich: D bzw. d direkt verwenden (ohne Hochstellungen, Übung)

Beispiel



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

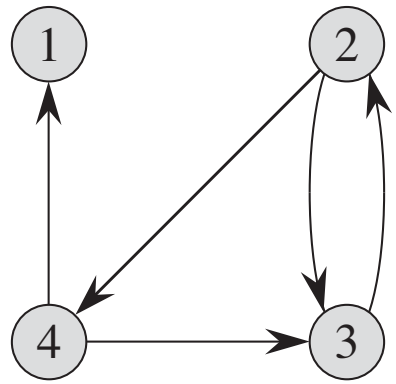
$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Transitive Hülle (1)

- Die **transitive Hülle** ist definiert als der Graph $G^* = (V, E^*)$

$$E^* = \{ (i,j) \mid \text{es existiert ein Pfad } i \rightsquigarrow j \text{ in } G \}$$



- Berechnung mittels Floyd-Warshall
  - Ordne jeder Kante das Gewicht 1 zu
  - Berechne Entfernungsmatrix D
  - Pfad von $i \rightsquigarrow j$ existiert wenn $\delta(i,j) = d_{ij} < \infty$
  - $O(n^3)$
- Schneller/platzsparender in Praxis: Arbeiten mit Boolschen Operationen (aber auch $O(n^3)$)

Transitive Hülle (2)

- Verwende ungewichtete Adjazenzmatrix
- Ersetze min durch $\vee$ (ODER)
- Ersetze $+$ durch $\wedge$ (UND)
- Definiere

$$t_{ij}^{(k)} = \begin{cases} 1 & \text{wenn es einen Pfad } i \rightsquigarrow j \text{ mit Zwischenknoten} \\ & \text{in } \{1, \ldots, k\} \text{ gibt} \\ 0 & \textit{sonst} \end{cases}$$

- Es gilt dann

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{wenn } i \neq j \text{ und } (i,j) \notin E \\ 1 & \text{wenn } i = j \text{ oder } (i,j) \in E \end{cases}$$

- und

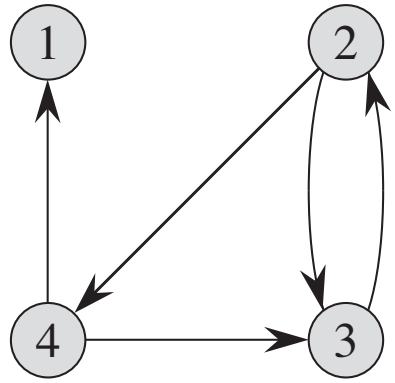
$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

Transitive Hülle (3)

TRANSITIVE-CLOSURE(G)

1: $n \leftarrow |V|$
2: Let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix
3: **for** $i \leftarrow 1$ **to** n
4: **for** $j \leftarrow 1$ **to** n
5: **if** $i = j$ or $(i, j) \in G.E$
6: $t_{ij}^{(0)} = 1$
7: **else** $t_{ij}^{(0)} = 0$
8: **for** $k \leftarrow 1$ **to** n
9: **for** $i \leftarrow 1$ **to** n
10: **for** $j \leftarrow 1$ **to** n
11: $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
12: **return** $T^{(n)}$

Beispiel



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Johnsons Algorithmus

- Fortgesetztes Quadrieren: $O(V^3 \lg v)$
- Floyd-Warshall: $O(V^3)$
- Dijkstra (mit Fibonacci-Heap) für jeden Knoten: $O(V^2 \log V + VE)$
  - Für dichte Graphen: $E = \Theta(V^2)$, Laufzeit $O(V^3)$
  - Für dünn besetzte Graphen: $E = o(V^2)$, Laufzeit $o(V^3)$
    $\rightarrow$ Asymptotisch schneller als Floyd-Warshall
- Aber: Dijkstras Algorithmus unterstützt keine negativen Kantengewichte
- **Johnsons Algorithmus** zeigt, wie wir Dijkstra trotzdem verwenden können
- Verwendet Dijkstra, Bellman-Ford und die Technik der **Neugewichtung**

Neugewichtung

Johnsons Algorithmus berechnet zuerst eine **neue Gewichtungsfunktion** $\hat{w}$ so dass gilt:

1. Für alle Knoten $u, v \in V$ ist die Menge der kürzesten Pfade zwischen u und v unter Gewichtung w die gleiche wie unter Gewichtung $\hat{w}$.
2. Für alle Kanten $(u, v) \in E$ ist $\hat{w}(u, v)$ nicht negativ.

Diskussion

- Dann kann Dijkstra mit Gewichtsfunktion $\hat{w}$ für jeden Knoten angewendet werden; man erhält die kürzesten Pfade für w
- Neugewichtung hat Laufzeit $O(VE)$
- Dijkstra mit mit Fibonacci-Heap: gesamt $O(V^2 \log V + VE)$
  - Mit zusätzlicher Neugewichtung ebenso
- Dijkstra mit binärem Heap: gesamt $O(V^2 \log V + VE \log V)$
  - Asymptotisch schneller als Floyd-Warshall wenn Graph hinreichend dünn besetzt, d.h. $E = o(V^2 / \log V)$

Die 1. Eigenschaft (1)

Lemma

Sei $G = (V, E)$ ein gewichteter gerichteter Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$. und $h : V \to \mathbb{R}$ eine beliebige Funktion. Definiere für jede Kante $(u, v) \in E$:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v).$$

Dann gilt:

- *Sei $p = \langle v_0, \ldots, v_k \rangle$ ein Pfad von v_0 nach v_k. Es gilt $w(p) = \delta(v_0, v_k)$ gdw. $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.*
- *G hat einen Zyklus mit negativen Gewicht mit w gdw. G hat einen Zyklus mit negativen Gewicht mit $\hat{w}$.*

Kurz:

- Kürzeste Pfade gleich
- Existenz negativer Zyklus gleich

Die 1. Eigenschaft (2)

Beweis.

- Wir zeigen zuerst

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k)$$

- Es gilt

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}$$

Die 1. Eigenschaft (3)

- Wir zeigen jetzt, dass $w(p) = \delta(v_0, v_k)$ gdw. $\hat{w}(p) = \hat{\delta}(v_0, v_k)$
- Sei $w(p) = \delta(v_0, v_k)$. Annahme für Widerspruch: Es existiert mit $\hat{w}$ ein Pfad p' von v_0 nach v_k mit

$$\hat{w}(p') < \hat{w}(p)$$

- Dann gilt

$$\begin{aligned} w(p') + h(v_0) - h(v_k) &= \hat{w}(p') \\ &< \hat{w}(p) = w(p) + h(v_0) - h(v_k) \end{aligned}$$

  und somit $w(p') < w(p) \rightarrow$ Widerspruch
- Andere Richtung analog

Die 1. Eigenschaft (4)

- Noch zu zeigen: G hat einen Zyklus mit negativen Gewicht mit w gdw. G hat einen Zyklus mit negativen Gewicht mit $\hat{w}$
- Sei $c = \langle v_0, \ldots, v_k \rangle$ mit $v_0 = v_k$ ein beliebiger Zyklus
- Dann

$$\begin{aligned} \hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) \end{aligned}$$

- Somit haben Zyklen unter w und $\hat{w}$ gleiches Gewicht □

Die 2. Eigenschaft (1)

Wir wollen $\hat{w}$ jetzt so definieren, dass $\hat{w}(u, v)$ nicht negativ für alle Kanten $(u, v) \in E$.

- Dazu erstellen wir einen neuen Graphen $G' = (V', E')$ mit
  - $V' = V \cup \{ s \}$ für einen neuen Knoten $s \notin V$
  - $E' = E \cup \{ (s, v) \mid v \in V \}$
  - $w(s, v) = 0$ für alle $v \in V$
- s hat keine einfallende Kante; somit
  - Kein kürzester Pfad mit Startknoten $\neq s$ enthält s
  - G' hat Zyklus mit negativem Gewicht gdw. G hat einen solchen Zyklus

Die 2. Eigenschaft (2)

- Wir setzen

$$h(v) = \delta(s, v)$$
$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

- $h(v) \leq 0$ nach Konstruktion
- Weiterhin gilt

$$h(v) \leq h(u) + w(u, v) \qquad \text{(Dreiecksungleichung)}$$
$$0 \leq h(u) + w(u, v) - h(v)$$

- Somit

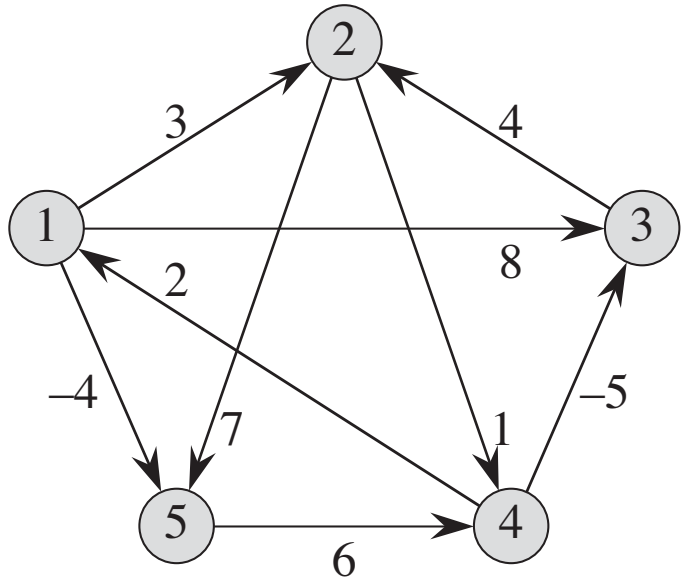
$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$$

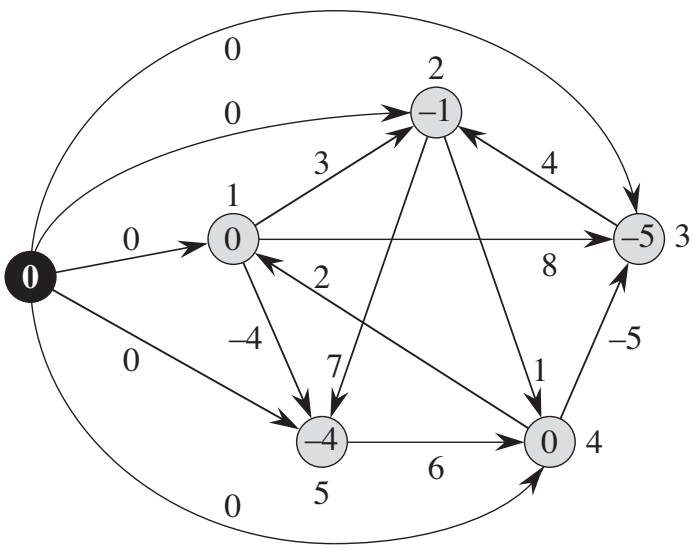
Johnsons Algorithmus (Pseudo-Code)

JOHNSON(G)

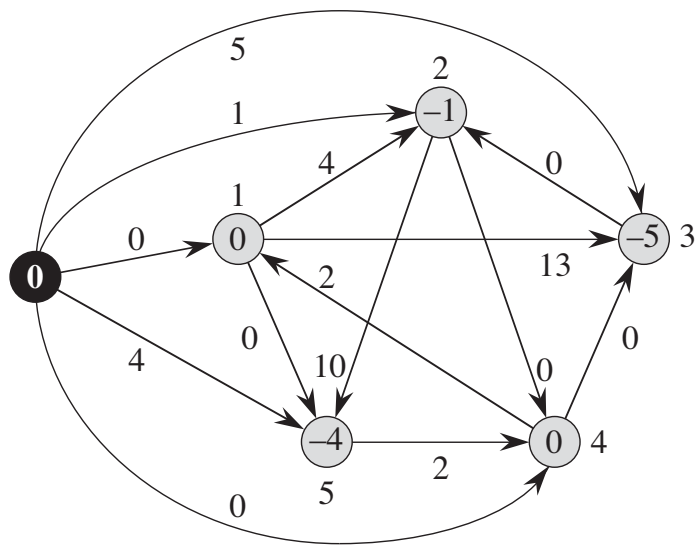
1: Compute graph G', where $V' = V \cup \{ s \}$,
   $E' = E \cup \{ (s, v) \mid v \in V \}$, $w(s, v) = 0$ for all $v \in V$
2: **if** BELLMAN-FORD$(G', w, s) =$ FALSE
3: Print "graph contains negative-weight cycle"
4: **else**
5: **for** each vertex $v \in G'.V$
6: $h(v) \leftarrow$ value of $\delta(s, v)$ computed by BELLMAN-FORD
7: **for** each edge $(u, v) \in G'.E$
8: $\hat{w}(u, v) \leftarrow w(u, v) + h(u) - h(v)$
9: Let $D = (d_{uv})$ be a new $n \times n$ matrix
10: **for** each vertex $u \in G.V$
11: Run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in V$
12: **for** each vertex $v \in G.V$
13: $d_{uv} = \hat{\delta}(u, v) - h(u) + h(v)$
14: **return** D

Beispiel (1)



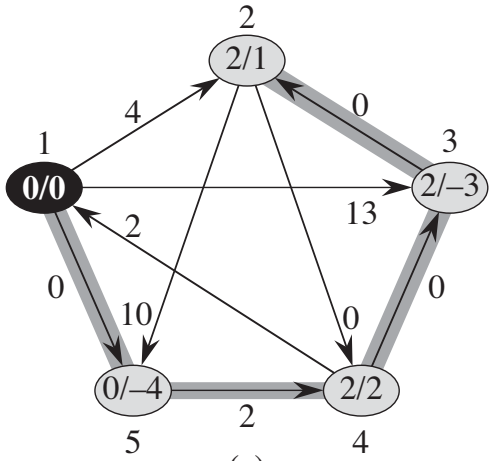


(a)

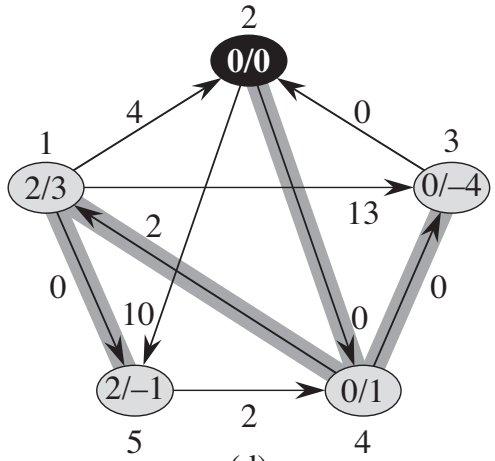


(b)

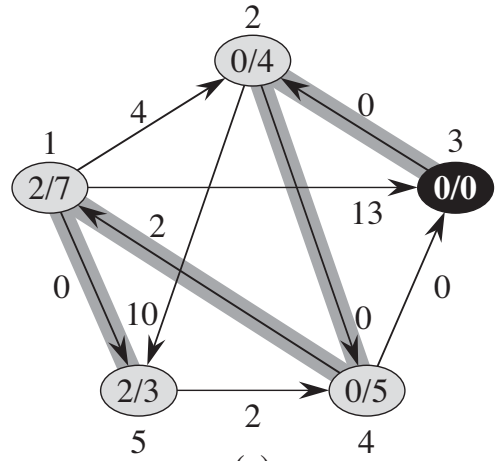
Beispiel (2)



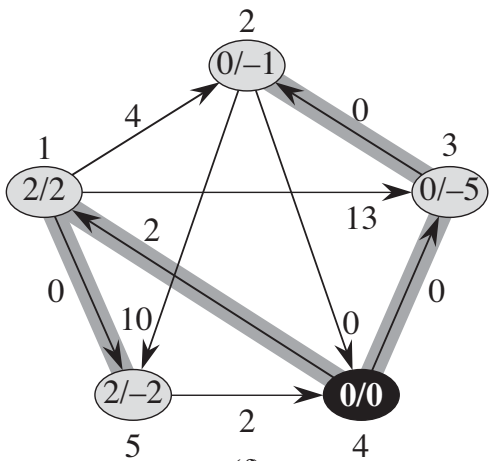
(c)



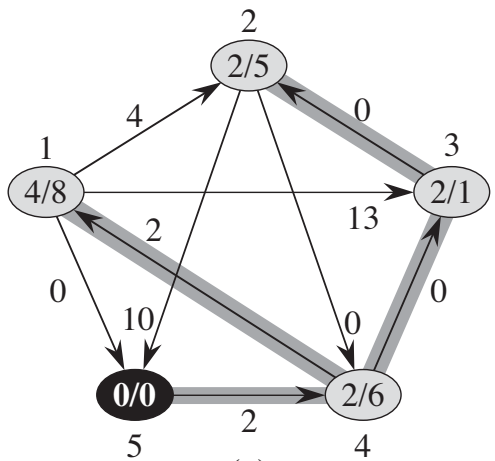
(d)



(e)



(f)



(g)

Outline

1. Elementare Graphalgorithmen

2. Minimale Spannbäume

3. Kürzeste Wege

4. Alle kürzesten Wege

Zum Nachlesen

- CLRS, App. B.4, *Graphs*
- CLRS, App. B.5, *Trees*
- CLRS, Ch. 22, *Elementary Graph Algorithms*
- CLRS, Ch. 23, *Minimum Spanning Trees*
- CLRS, Ch. 24, *Single-Source Shortest Paths*
- CLRS, Ch. 25, *All-Pairs-Shortest Paths*
- http://www-m9.ma.tum.de/Allgemeines/Routenplanung

Algorithmen und Datenstrukturen

05 – Zeichenkettenabgleich

Prof. Dr. Rainer Gemulla

Universität Mannheim

HWS 2014

Zeichenkettenabgleich

Wir betrachten das Problem des **Zeichenkettenabgleichs** (*string matching*).

- Gegeben: zwei **Zeichenketten** (*strings*)
  - Textdokument $T[1\,..\,n]$
  - **Muster** (*pattern*) $P[1\,..\,m]$
  - Beide über Alphabet Σ
- Gesucht
  - Alle Vorkommen von P in T
- Beispiel
  - $T = \texttt{xabxyabxyabxz}$
  - $P = \texttt{abxyabxz}$
- Beispielanwendungen
  - Suche in Textdokumenten
  - Mustersuche in einer DNA-Sequenz

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

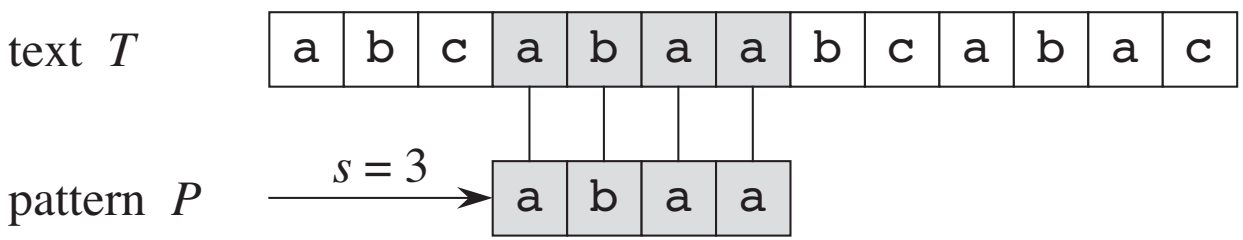
3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Formalisierung

- P kommt mit **Verschiebung** (*shift*) s im Text T vor, falls
  - $0 \leq s \leq n - m$
  - $T[s+1\,..\,s+m] = P[1\,..\,m]$
    d.h. $T[s+j] = P[j]$ für $1 \leq j \leq m$
  - Äquivalent: P fängt an Position $s+1$ in T an
- Falls P mit Verschiebung s in T vorkommt, nennen wir s eine **gültige Verschiebung**, sonst heißt s eine **ungültige Verschiebung**
- Das **Zeichenkettenabgleichsproblem** besteht in der Aufgabe alle gültigen Verschiebungen für T und P zu finden



Übersicht

- Viele Algorithmen verwenden **Vorverarbeitung** basierend auf dem Muster und führen danach erst den Zeichenkettenabgleich durch
- Wir schauen uns an:

Algorithmus	Vorverarbeitung	Abgleich
Naiv	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Endlicher Automat	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Definitionen (1)

- Σ^* bezeichne alle die Menge aller endlichen Wörter über Σ
- ϵ bezeichne das leere Wort
- $|x|$ bezeichne Länge des Wortes x
- xy bezeichne Konkatenation der Wörter x und y

- w ist ein **Präfix** von x, wenn $x = wy$ für ein $y \in \Sigma^*$
  - Geschrieben $w \sqsubseteq x$
  - z.B. $\texttt{ab} \sqsubseteq \texttt{abcca}$
- w ist ein **Suffix** von x, wenn $x = yw$ für ein $y \in \Sigma^*$
  - Geschrieben $w \sqsupseteq x$
  - z.B. $\texttt{cca} \sqsupseteq \texttt{abcca}$
- Bemerke
  - ϵ ist Präfix und Suffix jeder Zeichenkette
  - Wenn $w \sqsubseteq x$, dann $|w| \leq |x|$ (ebenso wenn $w \sqsupseteq x$)
  - $\sqsubseteq$ und $\sqsupseteq$ sind reflexiv und transitiv

Überlappende Suffixe (1)

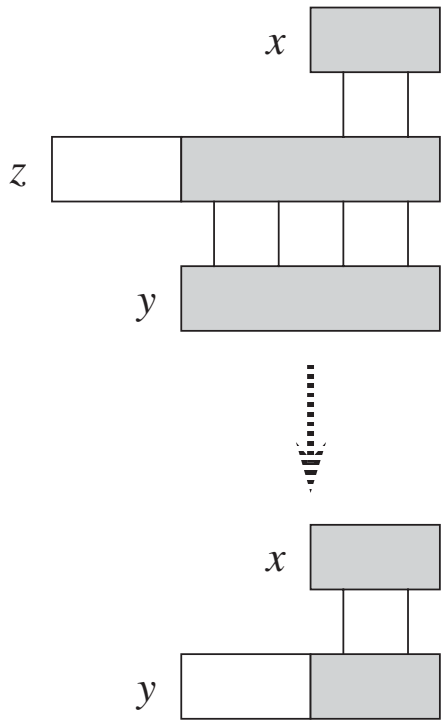
Lemma (F8, Lemma der überlappenden Suffixe)

Seien $x, y, z \in \Sigma^*$ mit $x \sqsupseteq z$ und $y \sqsupseteq z$. Dann gilt

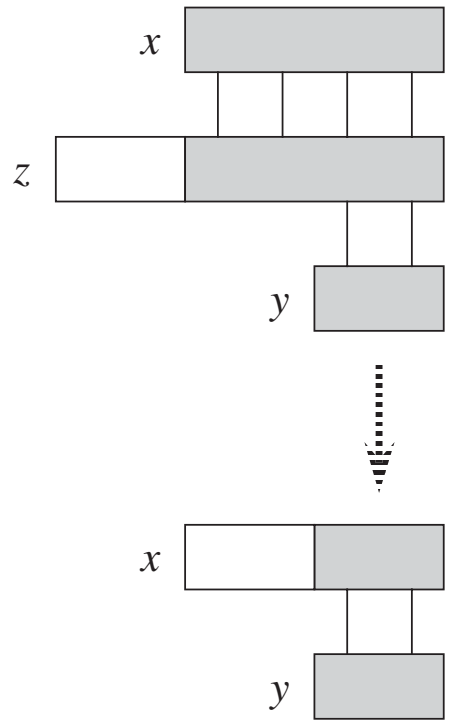
$$
\begin{aligned}
|x| \leq |y| &\implies x \sqsupseteq y \\
|x| \geq |y| &\implies y \sqsupseteq x \\
|x| = |y| &\implies y = x
\end{aligned}
$$

Überlappende Suffixe (2)

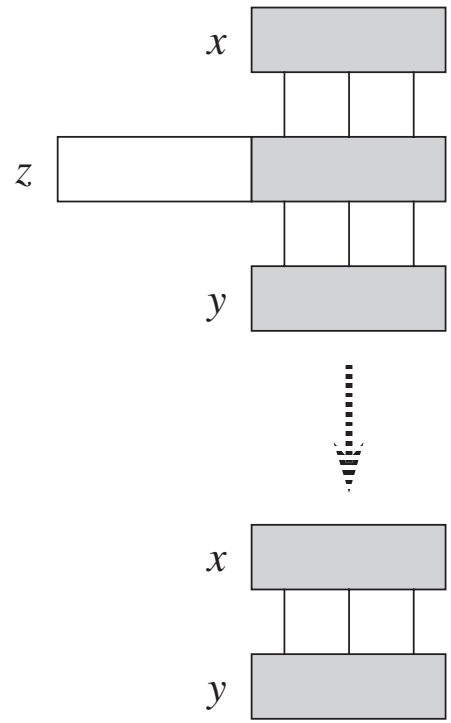
Beweis.



$|x| \leq |y|$



$|x| \geq |y|$



$|x| = |y|$

□

Definitionen (2)

- Sei $T_k = T[1\,..\,k]$ und $P_k = P[1\,..\,k]$
- Wir können das Zeichenkettenabgleichsproblem mit Präfixen und Suffixen ausdrücken
  - Finde alle diejenigen Verschiebungen $0 \leq s \leq n - m$, sodass $P \sqsupseteq T_{s+m}$
- Dazu sind ggf. **Zeichenkettenvergleiche** notwendig
  - Wir schreiben $x = y$
  - Und vergleichen zeichenweise von links nach rechts
  - Sei t Länge des längsten gemeinsamen Präfixes z, d.h. $z \sqsubseteq x$ und $z \sqsubseteq y$
  - Dann Laufzeit $\Theta(t + 1)$

Naiver Algorithmus

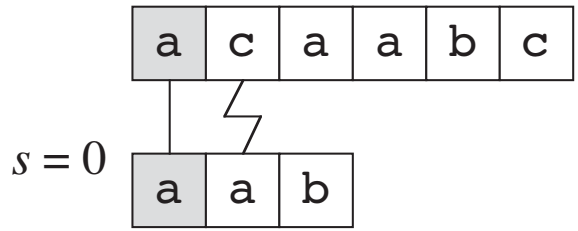
NAIVE-STRING-MATCHER(T, P)

1: $n \leftarrow T.length$
2: $m \leftarrow P.length$
3: **for** $s \leftarrow 0$ **to** $n - m$
4: **if** $P[1..m] = T[s + 1..s + m]$
5: Print "Pattern occurs with shift s"

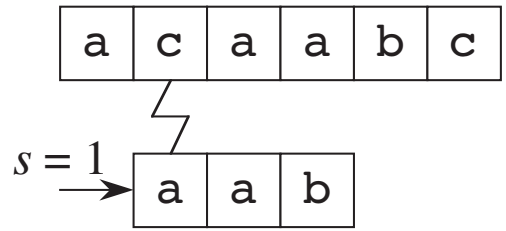
Laufzeit

- Laufzeit Z. 4 ist $O(m)$
- Laufzeit Algorithmus: $O((n - m + 1)m)$
- Worst Case-Laufzeit ist $\Theta((n - m + 1)m)$
  - Tritt z.B. auf wenn $T = \mathrm{a}^n$ und $P = \mathrm{a}^m$ für $\mathrm{a} \in \Sigma$
  - Wenn $m = \lfloor n/2 \rfloor$: $\Theta(n^2)$

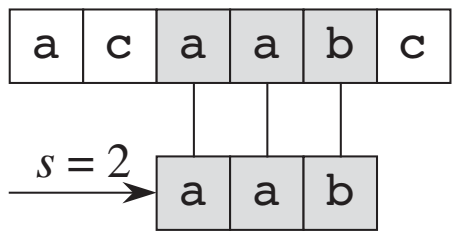
Beispiel



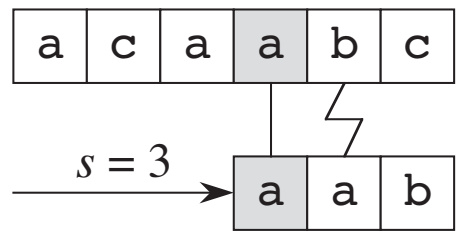
(a)



(b)



(c)



(d)

Vorverarbeitung

- Naiver Algorithmus ineffizient, da er Informationen, die während des Zeichenkettenvergleichs gewonnen werden, ignoriert
  - z.B. $P = \texttt{aaab}$
  - Wenn $s = 0$ gültig, dann folgt $s = 1, 2, 3$ ungültig
- Ziel: Vergleiche reduzieren, gewonnene Informationen nutzen
- Dazu notwendig: Vorverarbeitung des Musters P
  - Vorverarbeitung fließt in Laufzeit mit ein
  - Sollte effizient sein

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Idee

- Betrachte Wort als Zahl in d-ärer Darstellung mit $d = |\Sigma|$
- Sei vorerst $\Sigma = \{0, \ldots, 9\}$
  - Beispiel: Wort 123 entspricht Zahl 123
- Sei p die entsprechende Zahl für $P[1..m]$
- Sei t_s die Zahl des Teilwortes $T[s+1, \ldots, s+m]$
- Bemerke: s ist gültig gdw. $t_s = p$
- Wenn wir
  - p in $O(m)$ berechnen können,
  - Alle t_s zusammen in $O(n)$ berechnen können und
  - Ein Zahlenvergleich in $O(1)$ durchführen können,

  erhalten wir einen Algorithmus mit $O(m+n)$ Laufzeit
- Das ist fast der Rabin-Karp-Algorithmus

Anwenden des Horner-Schemas

- Wir berechnen p in $O(m)$ mittels **Horner-Schema**:

$$\begin{aligned} p &= P[m] + dP[m-1] + d^2P[m-2] + \cdots \\ &\quad + d^{m-2}P[2] + d^{m-1}P[1] \\ &= P[m] + d(P[m-1] + d(P[m-2] + \ldots \\ &\quad + d(P[2] + dP[1])\cdots)) \end{aligned}$$

- Berechnung von t_0 aus $T[1\,..\,m]$ in gleicher Weise
- Berechnung von $t_1, \ldots, t_{n-m}$ in $O(n-m)$

$$t_{s+1} = d(t_s - d^{m-1}T[s+1]) + T[s+m+1]$$

Beispiel

- $P[1..3] = 456$

$$p = 6 + 10(5 + 10 \cdot 4) = 6 + 10(45) = 456$$

- $T[1..5] = 34567$

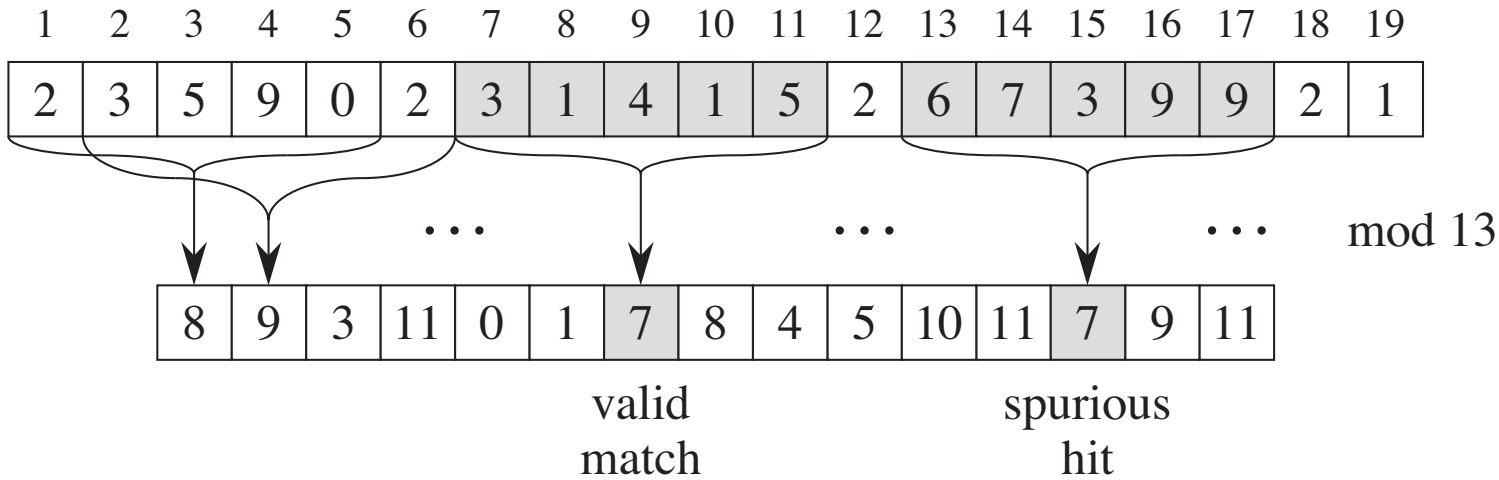
$$t_0 = 5 + 10(4 + 10 \cdot 3) = 5 + 10(34) = 345$$
$$t_1 = 10(345 - 100 \cdot 3) + 6 = 450 + 6 = 456$$
$$t_2 = 10(456 - 100 \cdot 4) + 7 = 560 + 7 = 567$$

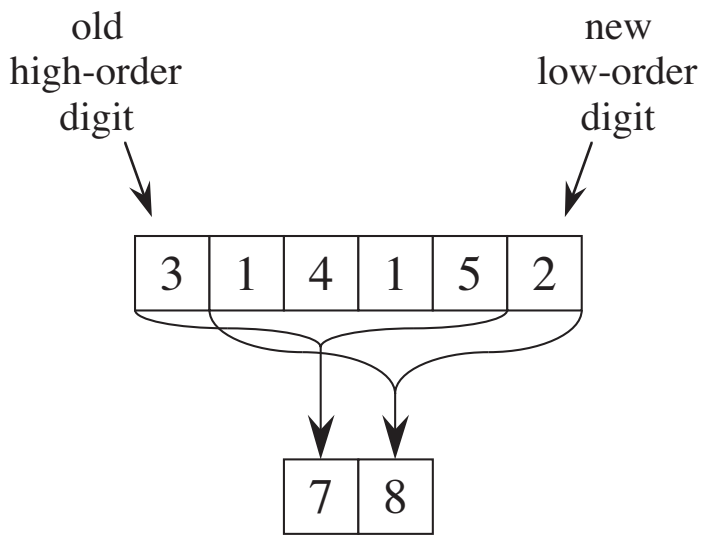
Problem

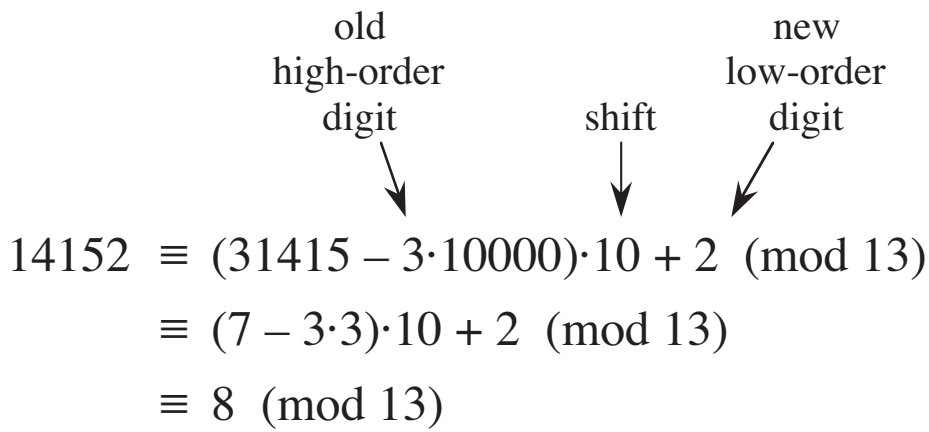
- Mit der bisher besprochenen Methode können die Zahlen groß werden
  - 4 ASCII-Zeichen $\rightarrow$ 28-Bit-Zahlen
  - 15 ASCII-Zeichen $\rightarrow$ 105-Bit-Zahlen
  - 30 DNA-Zeichen (TACG) $\rightarrow$ 60-Bit-Zahlen
  - 500 DNA-Zeichen (TACG) $\rightarrow$ 1000-Bit-Zahlen
- Abhilfe
  - Berechnungen werden modulo q durchgeführt (für ein geeignetes q)
  - Denke: t_s ist Hashwert von $T[s+1\,..\,s+m]$ (Abbildung wird **rollende Hashfunktion** genannt)
  - Vorteil: Alle Zahlen klein, d.h. in $0, 1, \ldots, q-1$
  - Nachteil: ungültige Treffer möglich (Kollisionen: unterschiedliche Zeichenketten, gleiche Zahlen) = **false positive**
- Entdecken ungültiger Treffer durch expliziten Zeichenkettenvergleich

Beispiel

$P = 31415,\ q = 13,\ 31415 \equiv 7 \pmod{13}$







Berechnung modulo q

- Die Rekurrenz wird

$$t_{s+1} = (d(t_s - hT[s+1]) + T[s+m+1]) \bmod q$$

  mit $h \equiv d^{m-1} \pmod{q}$

- h entspricht dem Wert der Ziffer 1 an der höchsten Stelle einer Zahl mit m Ziffern
- Wahl von q
  - So dass dq gerade noch in ein Rechenwort passt
    $\rightarrow$ Arithmetische Operationen in $O(1)$
  - q eine Primzahl ist

Rabin-Karp-Algorithmus (Pseudo-Code)

RABIN-KARP-MATCHER(T, P, d, q)

1: $n \leftarrow T.length$
2: $m \leftarrow P.length$
3: $h \leftarrow d^{m-1} \bmod q$
4: $p \leftarrow 0$
5: $t_0 \leftarrow 0$
6: **for** $i \leftarrow 1$ **to** m
7: $\quad p \leftarrow (dp + P[i]) \bmod q$
8: $\quad t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9: **for** $s \leftarrow 0$ **to** $n - m$
10: $\quad$ **if** $p = t_s$
11: $\quad\quad$ **if** $P[1..m] = T[s+1..s+m]$
12: $\quad\quad\quad$ Print "Pattern occurs with shift s"
13: $\quad$ **if** $s < n - m$
14: $\quad\quad t_{s+1} \leftarrow (d(t_s - hT[s+1]) + T[s+m+1]) \bmod q$

Laufzeit Rabin-Karp-Algorithmus

- Worst Case wie naiver Algorithmus
  - $P = \mathtt{a}^m$, $T = \mathtt{a}^n$, $\Theta((n-m+1)m)$
- Annahme: Abbildung $\Sigma^* \to \mathbb{Z}_q$ so gewählt, dass jeder Wert in $\mathbb{Z}_q$ mit gleicher Wahrscheinlichkeit vorkommt
  - Sei v die Anzahl gültiger Verschiebungen
  - Dann erwartete Laufzeit $O(n + m(v + n/q))$
  - Für $O(1)$ gültige Verschiebungen und $q \geq m$ ist erwartete Laufzeit $O(n)$

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Erinnerung: Endliche Automaten

Der nächste Algorithmus benutzt endliche Automaten.

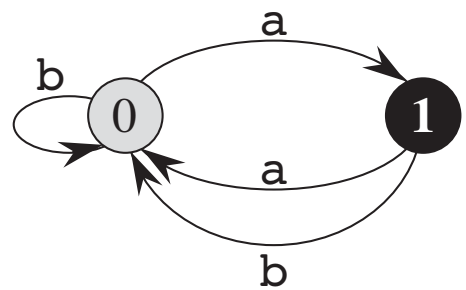
Definition

Ein **endlicher deterministischer Automat** (*DEA*, *DFA*) ist ein Tupel $M = (Q, q_0, A, \Sigma, \delta)$ mit

- Q ist eine endliche Menge von Zuständen
- $q_0 \in Q$ ist der Startzustand
- $A \subseteq Q$ ist eine Menge von Endzuständen
- Σ ist ein endliches Alphabet
- δ ist Zustandsübergangsfunktion $Q \times \Sigma \to Q$

Beispiel

state	input a	input b
0	1	0
1	0	0



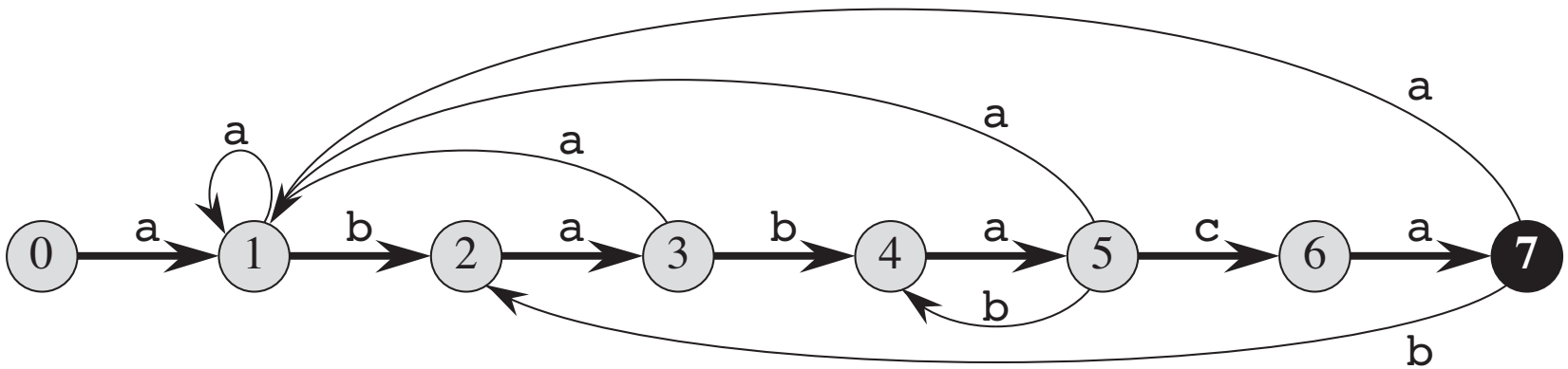
- Letztes Semester
  - Von regulären Ausdrücken zu Automaten
  - Vorgehensweise: NFA konstruieren, in DFA umwandeln
- Jetzt
  - Nicht reguläre Ausdrücke, sondern Zeichenketten
  - Nicht Zeichenkette als Ganzes akzeptieren, sondern Vorkommen innerhalb der Zeichenkette suchen
  - Direkte Konstruktion des DFAs
- Anschließend: Vermeidung der expliziten DFA-Konstruktion

Zeichenkettenabgleichsautomat

Jedes Muster kann auf einen Automaten abgebildet werden, der dann für einen gegebenen Text erkennt, ob das Muster in diesem Text vorkommt. Genauer:

- Automat akzeptiert alle Wörter $w \in \Sigma^*$, die mit Muster P enden (d.h. $L(M) = \{ w \in \Sigma^* \mid P \sqsupseteq w \}$)
- Wenn Endzustand nach Lesen des i-ten Zeichens i, dann Vorkommen von P mit Verschiebung $i - m$

Beispiel: $P =$ ababaca



Konvention: Rückkanten zu Zustand 0 werden nicht gezeichnet.

Zustandsfunktion

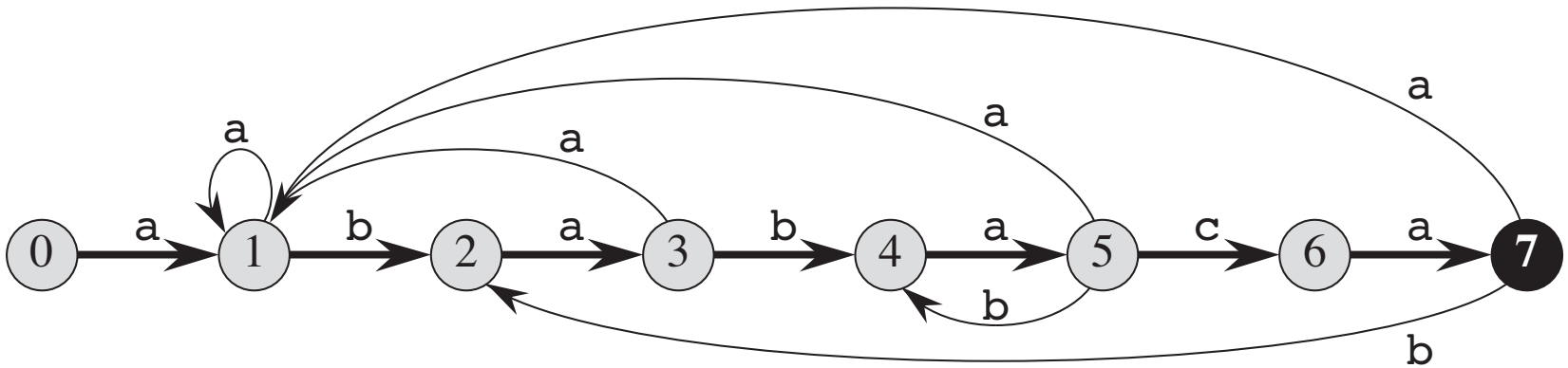
Definition

Die Zustandsübergangsfunktion δ kann kanonisch auf Wörter fortgesetzt werden. Wir erhalten die **Zustandsfunktion** $\phi : \Sigma^* \to Q$ mit

$$\begin{aligned} \phi(\epsilon) &= q_0 \\ \phi(wa) &= \delta(\phi(w), a). \end{aligned}$$

- $\phi(w)$ ist der Zustand des DFA nach Abarbeitung von w
- Automat akzeptiert ein Wort w gdw. $\phi(w) \in A$

Arbeitsweise des Automaten (Beispiel)



state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

Suffixfunktion

Fixiere ein P.

Definition

Die **Suffixfunktion** $\sigma : \Sigma^* \to \{1, \ldots, m\}$ ist gegeben durch

$$\sigma(x) = \max\{k \mid P_k \sqsupseteq x\}.$$

- $\sigma(x) =$ Länge des *längsten* Präfixes von P, der ein Suffix von x ist
- $=$ Länge des *längsten* Suffixes von x, der ein Präfix von P ist
- x endet mit den ersten $\sigma(w)$ Zeichen von P
- Beispiel mit $P = \mathtt{ab}$

$$\sigma(\epsilon) = 0$$
$$\sigma(\mathtt{ccaca}) = 1$$
$$\sigma(\mathtt{ccab}) = 2$$

- Bemerke: $\sigma(x) = m$ gdw. x endet mit P

Zeichenkettenabgleichsautomat

Sei $P[1\,..\,m]$ ein Muster. Dann definieren wir den **Zeichenkettenabgleichsautomat** $M = (Q, q_0, A, \Sigma, \delta)$ wie folgt:

$$
\begin{aligned}
Q &= \{\,0, 1, \ldots, m\,\} \\
q_0 &= 0 \\
A &= \{m\} \\
\Sigma &= \text{gegeben} \\
\delta(q, \mathrm{a}) &= \sigma(P_q \mathrm{a}) \qquad \text{für alle } q \in Q \text{ und } \mathrm{a} \in \Sigma
\end{aligned}
$$

- Invariante

$$\phi(T_i) = \sigma(T_i)$$

- D.h.: Automat "berechnet" die Suffixfunktion

Intuition

Nach Verabeitung von T_i:

- Aktueller Zustand q = Anzahl der abgeglichenen Zeichen
- Automat liest nächstes Zeichen $T[i+1] = \mathrm{a}$
- Automat soll dann zum längsten Präfix von P wechseln, der ein Suffix von $T_{i+1} = T_i\mathrm{a}$ ist $\rightarrow \sigma(T_i\mathrm{a})$
- Es gilt $\sigma(T_i\mathrm{a}) = \sigma(P_q\mathrm{a})$ (Lemma F34)
- Fall 1: $\mathrm{a} = P[q+1]$, d.h. passt zum Muster
  - Automat bewegt sich zu Zustand $q' = q+1$
- Fall 2: $\mathrm{a} \neq P[q+1]$
  - Automat bewegt sich in einen Zustand $q' \leq q$ sodass $P_{q'} \sqsupseteq T_{i+1}$ und q maximal
  - D.h. in Zustand $q' = \sigma(P_q\mathrm{a})$

Simulation (Pseudo-Code)

FINITE-AUTOMATON-MATCHER(T, δ, m)

```
1: n ← T.length
2: q ← 0
3: for i ← 1 to n
4:     q ← δ(q, T[i])
5:     if q = m
6:         s ← i − m
7:         Print "Pattern occurs with shift s"
```

- Laufzeit $\Theta(n)$
- Vorab notwendig: Berechnung von δ

Korrektheit (1)

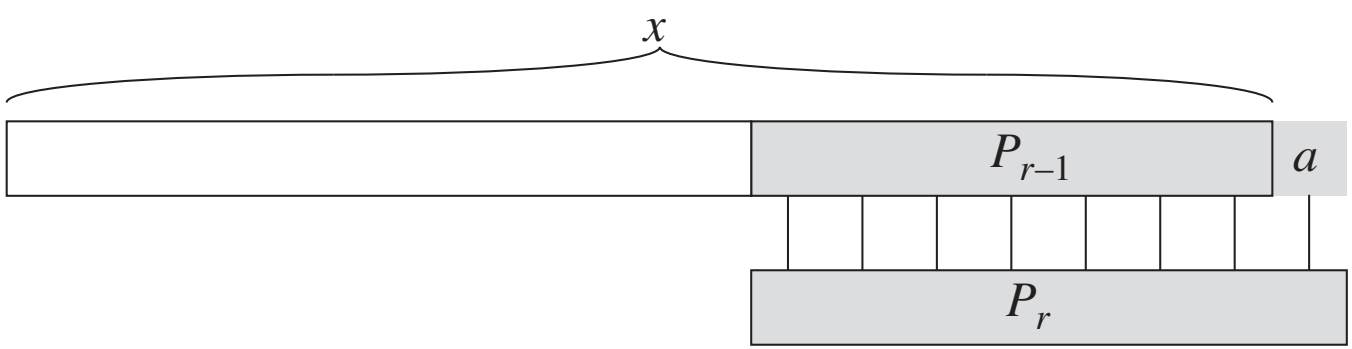
Lemma (Suffixfunktionsungleichung)

Sei $x \in \Sigma^*$ ein Wort und $a \in \Sigma$ ein Buchstabe. Dann gilt

$$\sigma(xa) \leq \sigma(x) + 1.$$

Beweis.

- Sei $r = \sigma(x\mathrm{a})$
- Wenn $r = 0$, folgt Behauptung da σ nicht-negativ
- Sei $r > 0$, dann $P_r \sqsupseteq x\mathrm{a}$ nach Definition von σ
- Somit $P_{r-1} \sqsupseteq x$ und demzufolge $\sigma(x) \geq r - 1$
- Zusammen $\sigma(x\mathrm{a}) = r \leq \sigma(x) + 1$ □



Korrektheit (2)

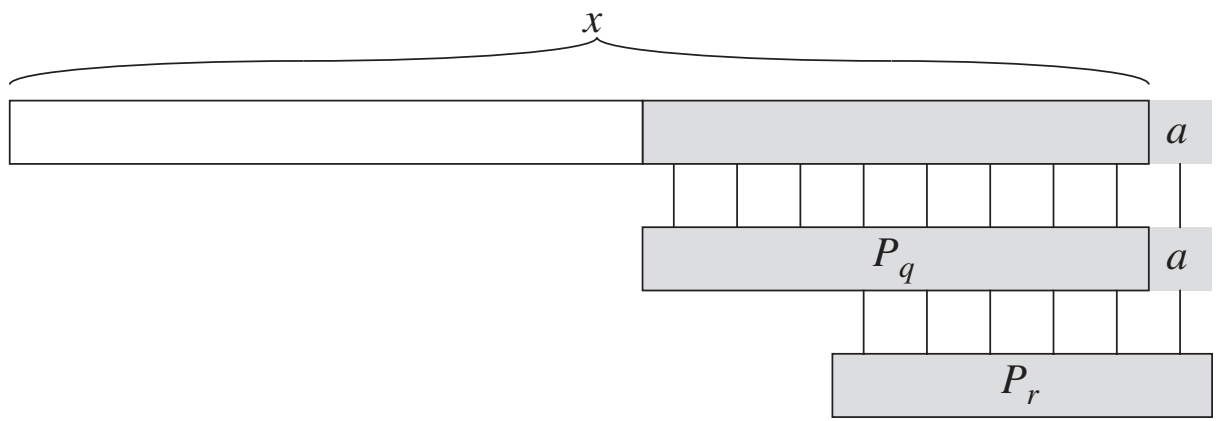
Lemma (F34, Suffixfunktionsrekursion)

Sei $x \in \Sigma^*$ ein Wort und $a \in \Sigma$ ein Buchstabe. Dann gilt

$$q = \sigma(x) \implies \sigma(xa) = \sigma(P_q a)$$

Beweis.

- Sei $r = \sigma(x\mathsf{a})$, dann $P_r \sqsupseteq x\mathsf{a}$ und $r \leq q + 1$ nach Ungleichung
- Da zusätzlich $P_q\mathsf{a} \sqsupseteq x\mathsf{a}$ gilt
  - $\sigma(P_q\mathsf{a}) \leq \sigma(x\mathsf{a})$
  - $P_r \sqsupseteq P_q\mathsf{a}$ (Lemma F8) und somit $\sigma(x\mathsf{a}) = r \leq \sigma(P_q\mathsf{a})$
- Somit $\sigma(P_q\mathsf{a}) = \sigma(x\mathsf{a})$ □



Korrektheit (3)

Theorem

Sei ϕ die Zustandsfunktion eines Zeichenkettenabgleichsautomaten für ein Muster P. Sei $T[1\,..\,n]$ ein Text. Dann gilt für alle $0 \leq i \leq n$

$$\phi(T_i) = \sigma(T_i).$$

Beweis.

- Per Induktion über i
- **Induktionsanfang** ($i = 0$): Trivial, da $T_0 = \epsilon$ und somit $\phi(T_0) = 0 = \sigma(T_0)$

Korrektheit (4)

- **Induktionsschritt**: Es gelte IV $\phi(T_i) = \sigma(T_i)$
  - Sei $q = \phi(T_i)$ und $T[i+1] = \mathtt{a}$
  - Es gilt

$$
\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i\mathtt{a}) \\
&= \delta(\phi(T_i), \mathtt{a}) && \text{(nach Definition } \phi\text{)} \\
&= \delta(q, \mathtt{a}) && \text{(nach Definition } q\text{)} \\
&= \sigma(P_q\mathtt{a}) && \text{(nach Definition } \delta\text{)} \\
&= \sigma(T_i\mathtt{a}) && \text{(nach Lemma F34 und IV)} \\
&= \sigma(T_{i+1}).
\end{aligned}
$$

□

Berechnen der Zustandsübergangsfunktion

COMPUTE-TRANSITION-FUNCTION(P, Σ)

1: $m \leftarrow P.length$
2: **for** $q \leftarrow 0$ **to** m
3: **for** each character a $\in \Sigma$
4: $k \leftarrow \min(m+1, q+2)$
5: **repeat**
6: $k \leftarrow k-1$
7: **until** $P_k \sqsupseteq P_q$a
8: $\delta(q, \text{a}) \leftarrow k$
9: **return** δ

- Laufzeit: $O(m^3|\Sigma|)$
- Kann in $O(m|\Sigma|)$ umgesetzt werden

Beispiel

- $\Sigma = \{\, \mathtt{a}, \mathtt{b}, \mathtt{c} \,\}$
- $P = \mathtt{ababaca}$
- $m = 7$

q	x	k	P_k	$P_q x$	$\delta(q, x)$
0	a	1	a	$\underline{\mathtt{a}}$	1
	b	1	a	$\underline{\mathtt{b}}$	
		0	ϵ	b	0
	c	1	a	$\underline{\mathtt{c}}$	
		0	ϵ	c	0
1	a	2	ab	$\underline{\mathtt{aa}}$	
		1	a	$\mathtt{a}\underline{\mathtt{a}}$	1
	b	2	ab	$\underline{\mathtt{ab}}$	2
	c	2	ab	$\underline{\mathtt{ac}}$	
		1	a	$\mathtt{a}\underline{\mathtt{c}}$	
		0	ϵ	ac	0
2	a	3	aba	$\underline{\mathtt{aba}}$	3
	b	3	aba	$\underline{\mathtt{abb}}$	
		2	ab	$\mathtt{a}\underline{\mathtt{bb}}$	
		1	a	$\mathtt{abb}\underline{\mathtt{b}}$	
		0	ϵ	abb	0
			$\vdots$		

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Knuth-Morris-Pratt-Algorithmus

- Endlicher Automat benötigt
  - $O(|\Sigma|m)$ zur Berechnung der Zustandsübergangsfunktion δ
  - $O(n)$ für die eigentliche Suche
- **Knuth-Morris-Pratt-Algorithmus** (KMP) vermeidet Berechnung von δ
  - Verwendet Array $\pi[1\,..\,m]$, welches für jedes a erlaubt, $\delta(q, a)$ effizient zu berechnen
  - π kann in $O(m)$ berechnet werden $\rightarrow$ Faktor $|\Sigma|$ eingespart
  - Insgesamt $O(m + n)$

Präfixfunktion (Vorbetrachtung)

Für π verwenden wir eine **Präfixfunktion**.

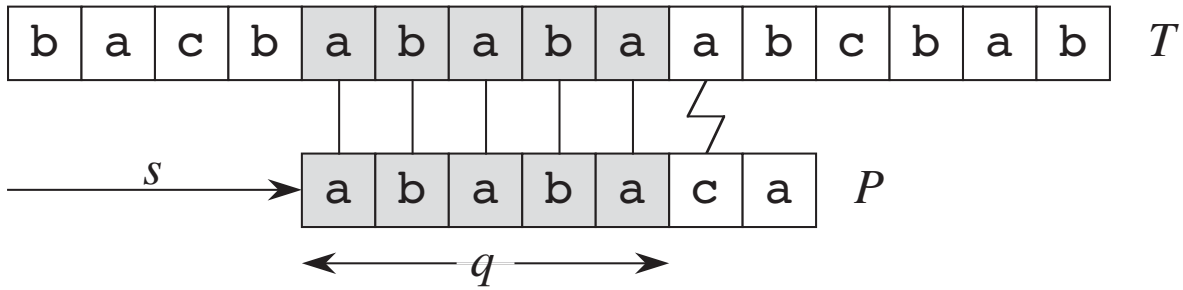
- Enthält Informationen darüber, wie ein Muster sich zu sich selbst verhält, falls es gegenüber sich selbst verschoben wird
- Wird verwendet, um "unnötige" Verschiebungen im naiven Algorithmus nicht zu testen
- Kernidee
  - Sei $P[1\,..\,q] = T[s+1\,..\,s+q]$ für Verschiebung s
  - Bemerke: für $q < m$ kann s gültig oder ungültig sein
  - Nehmen wir an, wir wüssten, dass s ungültig (oder $q = m$)
  - Frage: Was ist dann die nächste Verschiebung $s' > s$, die gültig sein könnte?
  - Antwort: Kleinstes $s' > s$, sodass

    $$P[1\,..\,k] = T[s'+1\,..\,s+q]$$

    mit $k = q - (s' - s)$?
  - Verschiebungen in (s, s') brauchen nicht getestet zu werden

Illustration

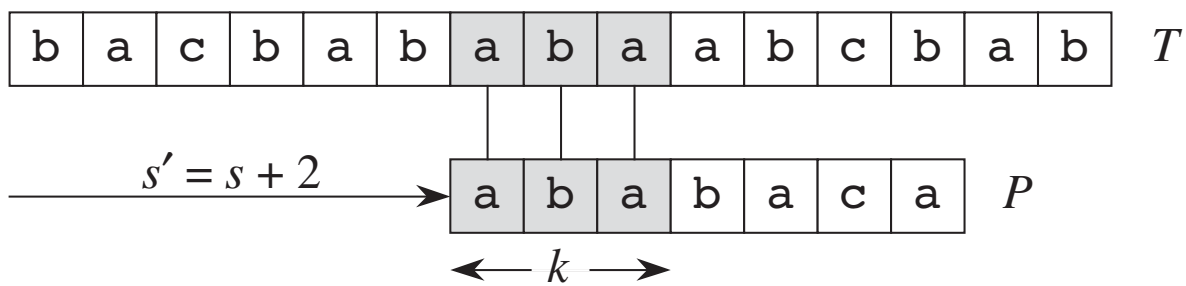
- $P =$ `ababaca`



- Verschiebung $s + 1$ muss ungültig sein, da

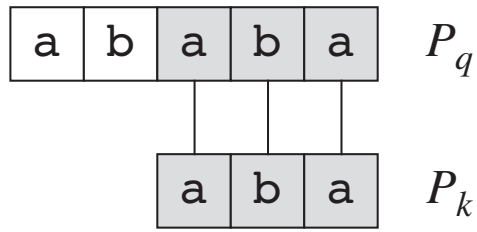
$$P[1] = \texttt{a} \neq \texttt{b} = T[(s+1)+1]$$

- Aber: Verschiebung $s' = s + 2$ konsistent mit P bis $s + q$
  $\rightarrow$ Somit potentiell gültig



Präfixfunktion

- Die nächste potentiell gültige Verschiebung können wir vorberechnen, indem wir das Muster mit sich selbst vergleichen



- Warum?
  - $T[s+1\,..\,s+q] = P[1\,..\,q] \rightarrow T[s'+1\,..\,s+q] \sqsupseteq P_q$
  - Bemerke: P_3 ist längster Präfix von P, der *echter* Suffix von P_5 ist
- Im Allgemeinen: **Präfixfunktion** ist gegeben durch

$$\pi[q] = \max\{\, k \mid k < q \;\wedge\; P_k \sqsupseteq P_q \,\}$$

- Nächste potentiell gültige Verschiebung liegt dann bei

$$s' = s + (q - \pi[q])$$

Knuth-Morris-Pratt-Algorithmus (Pseudo-Code)

KMP-Matcher(T, P)

```
 1: n ← T.length
 2: m ← P.length
 3: π ← Compute-Prefix-Function(P)
 4: q ← 0
 5: for i ← 1 to n
 6:     while q > 0 and P[q + 1] ≠ T[i]
 7:         q ← π[q]
 8:     if P[q + 1] = T[i]
 9:         q ← q + 1
10:     if q = m
11:         Print "Pattern occurs with shift i − m"
12:         q ← π[q]
```

- Amortisierte Analyse: $\Theta(n)$ Laufzeit ohne Berechnung π (Z. 3)
  - q wird nur in Z. 9 erhöht, in Z. 7 aber erniedrigt
  - Da $q > 0$ in Z. 7, wird Z. 7 nicht öfter ausgeführt als Z.9
    $\rightarrow O(n)$ insgesamt

Berechnung der Präfixfunktion (Pseudo-Code)

COMPUTE-PREFIX-FUNCTION(P)

```
1:  m ← P.length
2:  π[1] ← 0
3:  k ← 0
4:  for q ← 2 to m
5:      while k > 0 and P[k + 1] ≠ P[q]
6:          k ← π[k]
7:      if P[k + 1] = P[q]
8:          k ← k + 1
9:      π[q] ← k
10: return π
```

- Amortisierte Analyse: $\Theta(m)$ Laufzeit

Beispiel ($P = \texttt{ababaca}$, $m = 7$)

q	P_q	k	P_{k+1}	Bemerkung
1	a	0	a	$\pi[1] \leftarrow 0$
2	ab	0	a	Verlassen Z. 5 ($\neq$)
		0		$\pi[2] \leftarrow 0$
3	aba	0	a	Verlassen Z. 5 ($=$)
		1		$\pi[3] \leftarrow 1$
4	abab	1	ab	Verlassen Z. 5 ($=$)
		2		$\pi[4] \leftarrow 2$
5	ababa	2	aba	Verlassen Z. 5 ($=$)
		3		$\pi[5] \leftarrow 3$
6	ababac	3	abab	
		1	ab	
		0	a	Verlassen Z. 5 ($\neq$)
		0		$\pi[6] \leftarrow 0$
7	ababaca	0	a	Verlassen Z. 5 ($=$)
		1		$\pi[7] \leftarrow 1$

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Boyer-Moore-Algorithmus

- Oft in der Praxis verwendet (z.B. `grep`)
- Insbesondere gut geeignet wenn
  - Alphabet groß
  - Muster lang
- Wie vorab
  - Muster und Text werden ausgerichtet und verglichen
  - Muster wird nach rechts verschoben
- Drei neue Ideen
  - Muster wird von rechts nach links mit Eingabe verglichen
  - Ungültiges-Zeichen-Verschiebungsregel
  - Gültiger-Suffix-Verschiebungsregel
- Damit
  - Oft weniger als m+n Zeichen betrachtet (sublineare Zeit)
  - Linear im Worst Case (mit bestimmten Erweiterungen, wie KMP)

Von rechts nach links

	1								2								
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7 $\cdots$
T:	x	p	b	c	t	b	x	a	b	p	q	x	c	t	b	p	q $\cdots$
P ($s = 12$):			t	p	a	b	x	a	b								

- Für jede Verschiebung wird P mit T von rechts nach links verglichen
- Im Beispiel: Nichtübereinstimmung von Position $T[15]$ mit $P[3]$ $\rightarrow$ s ist ungültig
- Wiederum: Was ist die kleinste potentiell gültige Verschiebung $s' > s$?

Ungültiges-Zeichen-Verschiebungsregel (Beispiel)

	1									2							
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7 $\cdots$
T:	x	p	b	c	t	b	x	a	b	p	q	x	c	t	b	p	q $\cdots$
P ($s = 12$):			t	p	a	b	x	a	b								
P ($s' = 14$):					t	p	a	b	x	a	b						
P ($s'' = 21$):												t	p	a	b	x	a b

- Idee: Das **ungültige Zeichen** $T[15] = \texttt{t}$ korrigieren
  - Dazu muss P mindestens 2 Zeichen nach rechts verschoben werden
  - Somit $s' = 14$ und Zeichen $T[15] = T[s' + 1] = P[1]$ gültig
- Jetzt: Das **ungültige Zeichen** $T[21] = \texttt{q}$ korrigieren
  - q kommt nicht in P vor
  - Gültige Verschiebung kann $T[21]$ nicht enthalten
    $\rightarrow$ P nach $T[21]$ verschieben
  - Somit $s'' = 21$

Ungültiges-Zeichen-Verschiebungsregel

- Sei $R(x)$ für $x \in \Sigma$ die **Position des letzten Vorkommens** von x in P wenn $x \in P$ bzw. 0 wenn $x \notin P$
- Es gelte für eine Verschiebung s, dass
  - Die $m - i$ rechten Zeichen des Musters mit dem Text übereinstimmen ($P[i+1\,..\,m] = T[s+i+1\,..\,s+m]$)
  - Das nächste Zeichen ungültig ist, d.h. $P[i] \neq T[s+i]$
- Dann besagt die **Ungültiges-Zeichen-Verschiebungsregel**, dass P um

$$\max[1, i - R(T[s+i])]$$

Positionen nach rechts verschoben wird

Diskussion

- Nach Anwendung der Ungültiges-Zeichen-Verschiebungsregel gilt:
  - Falls $R(T[s+i]) < i$: das ungültige Zeichen stimmt mit dem entsprechenden Zeichen des verschobenen Muster überein
  - Falls $R(T[s+i]) = 0$: das Muster wird hinter das ungültige Zeichen verschoben
  - Falls $R(T[s+i]) > i$: Das Muster wird nur um ein Zeichen verschoben
- Sehr effektiv, wenn Fehler am Ende des Musters auftritt
- Aber: Ineffektiv wenn ungültiges Zeichen rechts von $P[i]$ vorkommt

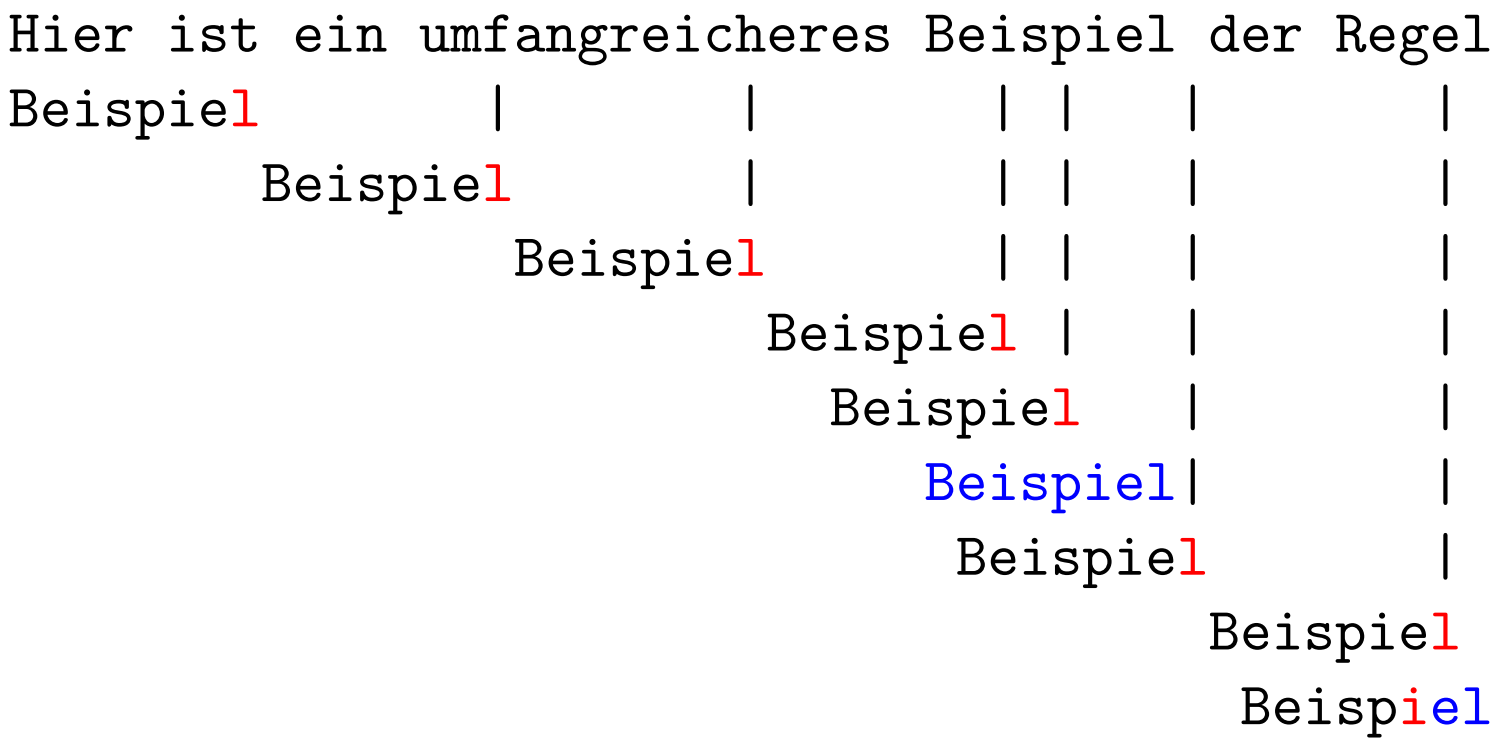
		1								2							
		1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7 ⋯
T':		x	p	b	c	t	b	x	b	b	p	q	x	c	t	b	p	q ⋯
P $(s = 2)$:				t	p	a	b	x	a	b								
P $(s' = 3)$:					t	p	a	b	x	a	b							

Erweiterte Ungültiges-Zeichen-Verschiebungsregel

- **Erweiterte Ungültiges-Zeichen-Verschiebungsregel**:
  Betrachte nicht alle Zeichen in P, sondern nur die Zeichen links der Nichtübereinstimmung
- D.h. verwende anstelle von $R(x)$ eine Funktion $R_i(x)$, welche die Position des letzten Vorkommens von x in P_{i-1} liefert bzw. 0 wenn $x \notin P_{i-1}$
  - Kann durch absteigend sortierte Listen $L[x] \subseteq \{1, \ldots, m\}$ in $O(m)$ Speicherplatz repräsentiert werden
  - Vorverarbeitung $O(m)$ Laufzeit
  - Berechnung $R_i(x)$ mittels Durchlauf durch Liste in $O(m - i)$
    $\rightarrow$ Finden der Nichtübereinstimmung ist ebenso $O(m - i)$

		1								2							
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7 ⋯
T':	x	p	b	c	t	b	x	b	b	p	q	x	c	t	b	p	q ⋯
P $(s = 2)$:			t	p	a	b	x	a	b								
P $(s' = 4)$:					t	p	a	b	x	a	b						

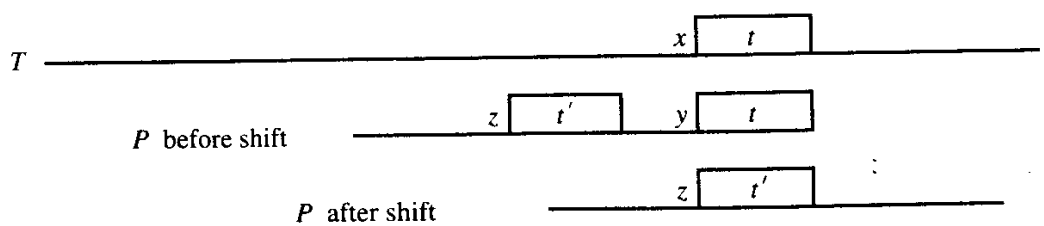
Erweiterte Ungültiges-Zeichen-Verschiebungsregel (Beispiel)



Gültiger-Suffix-Verschiebungsregel (Idee)

	1									2								
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8 ···
T:	p	r	s	t	a	b	s	t	u	b	a	b	v	q	x	r	s	t···
P $(s = 2)$:			q	c	a	b	d	a	b	d	a	b						
P $(s' = 8)$:										q	c	a	b	d	a	b	d	a b

- Ungültiges-Zeichen-Verschiebungsregel oft effektiv, aber
  - Weniger effektiv für kleine Alphabete
  - Keine lineare Worst Case-Laufzeit
- Deswegen: **Gültiger-Suffix-Verschiebungsregel**
  - Ausnutzen des schon abgeglichenen Suffixes
    (d.h. dass $P[i \,..\, m] = T[s + i \,..\, s + m]$)
  - Idee: Muster wird nach rechts verschoben sodass
    - Abgeglichener Suffix immer noch übereinstimmt und
    - Zeichen an Position der Nichtübereinstimmung $\neq P[i - 1]$



Vorverarbeitung

- Sei $L'(i)$ die größte Position kleiner m sodass

  $$P[i\,..\,m] \sqsupseteq P[1\,..\,L'(i)]$$

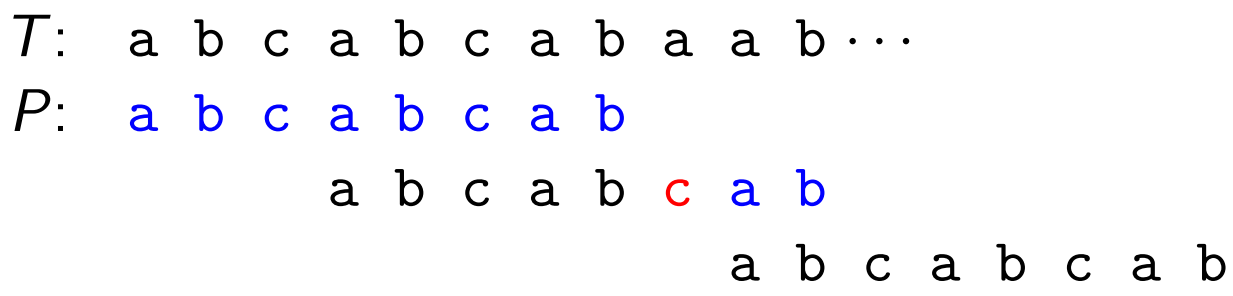
  und

  $$P[i-1\,..\,m] \not\sqsupseteq P[1\,..\,L'(i)],$$

  $\rightarrow$ Vorletztes Auftreten des abgeglichenen Suffixes mit verschiedenem Zeichen davor
- $L'(i) = 0$ wenn keine Position Bedingungen erfüllt
- Beispiel: $P = \texttt{cabdabdab}$
  - $L'(9) = 0$, $L'(8) = 3$, $L'(7) = L'(6) = 0$, $L'(5) = 6$
- Alle $L(i)$ sowie $\sigma(P[i\,..\,m])$ können für alle i in $O(m)$ Laufzeit berechnet werden

Gültiger-Suffix-Verschiebungsregel

- Es gelte für eine Verschiebung s, dass
  - Die $m - i + 1$ rechten Zeichen des Musters mit dem Text übereinstimmen ($P[i\,..\,m] = T[s + i\,..\,s + m]$)
  - Das nächste Zeichen ungültig ist, d.h. $P[i - 1] \neq T[s + i - 1]$
- Nach der **Gültiger-Suffix-Verschiebungsregel** wird P wie folgt nach rechts verschoben:
  1. Wenn $L'(i) > 0$: um $m - L'(i)$ Zeichen
  2. Wenn $L'(i) = 0$: um $m - \sigma(P[i\,..\,m])$ Zeichen
  3. Wenn P gefunden: um $m - \sigma(P[2\,..\,m])$ Zeichen
- Beispiel
  - $m = 8$
  - Fall 3: $\sigma(P[2\,..\,m]) = 5$ (`abcab`) $\rightarrow$ Um 3 Positionen
  - Fall 2: $\sigma(P[7\,..\,8]) = 2$ (`ab`) $\rightarrow$ Um 6 Positionen



Diskussion

- Beide Verschiebungsregeln sind effektiv
- Werden in der Praxis kombiniert
  - Mehrere Varianten
  - z.B.: Ungültiges-Zeichen-Regel wenn erster Zeichenvergleich fehlschlägt, sonst Gültiger-Suffix-Regel
  - z.B.: Immer das Maximum beider Regeln
- $O(n + m)$ Laufzeit wenn Muster nicht im Text vorkommt
- Sonst $\Theta(nm)$ Worst Case-Laufzeit ($T = \mathrm{a}^n$, $P = \mathrm{a}^m$)
- Aber mit Modifikation (**Regel von Galil**): $O(m + n)$ in allen Fällen

Outline

1. Vorbetrachtung

2. Rabin-Karp-Algorithmus

3. Endliche Automaten

4. Knuth-Morris-Pratt-Algorithmus

5. Boyer-Moore-Algorithmus

Zum Nachlesen

- CLRS, Ch. 32, *String Matching*
- Dan Gusfield
  Ch. 2: *Exact Matching: Classical Comparison-Based Methods*
  in *Algorithms on Strings, Trees, and Sequences*
  Cambridge University Press, 1997

Algorithmen und Datenstrukturen

06 – Suchalgorithmen

Prof. Dr. Rainer Gemulla

Universität Mannheim

HWS 2014

Problemlösen durch Suchen

- Wir suchen eine optimale Lösung für ein Problem
- Manchmal ist das Überprüfen aller Möglichkeiten (*exhaustive search*) die einzige Möglichkeit
- Aber
  - Möglichkeiten sollen systematisch durchsucht werden
  - Kein wiederholtes Überprüfen einer Möglichkeit
  - Möglichlichst schnelles Finden einer Lösung
- Dazu werden Suchalgorithmen verwendet, oft:
  - Backtracking
  - Branch-and-Bound

Outline

1. Suchbäume

2. Backtracking

3. Memoization

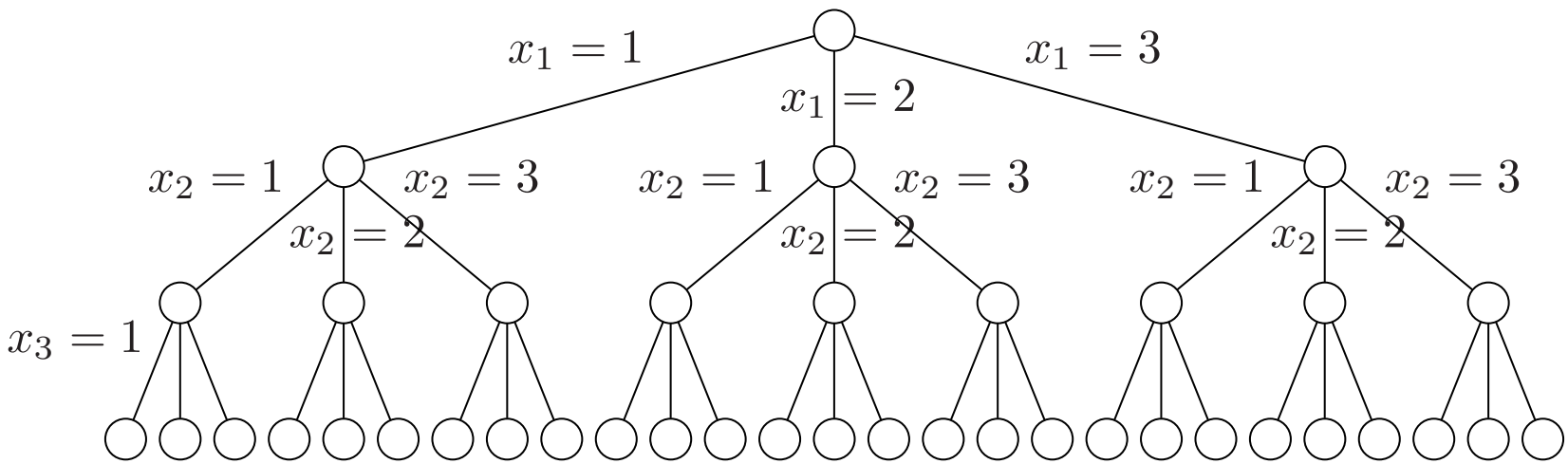
4. Branch-and-Bound

Ausgangssituation

- System mit endlich vielen Zuständen (Möglichkeiten, Konfigurationen)
- Gesucht sind Zustände, die ein bestimmtes Kriterium erfüllen = **Lösungszustände**
- Zustände haben "Struktur"
  - Jeder Zustand lässt sich durch eine Folge von Werten darstellen
  - Genauer: Zustand durch ein n-Tupel $(x_1, \ldots, x_n)$,
  - $x_i \in S_i$ und $|S_i| < \infty$, $i = 1 \ldots n$
  - n kann für verschiedene Zustände verschieden sein
- Beispiel
  - $S_i = S$, $i = 1 \ldots n$, $|S| = d$
  - Zustandsraum $Z = S^n$ der Größe d^n
- Zustandsraum kann dadurch baumartig organisiert werden
  $\rightarrow$ Erleichtert systematische Überprüfung
- Zwei Arten
  1. Zustände entsprechen den Blättern
  2. Zustände entsprechen den Knoten

Variante 1: Zustände entsprechen Blättern

- $S = S_i,\ i = 1 \ldots 3$
- $S = \{1, 2, 3\}$
- $Z = S^3 = \{(1,1,1), (1,1,2), \ldots, (3,3,3)\}$
- $x \in Z$ beschrieben durch $x = (x_1, x_2, x_3)$



Beispiel: Teilsummenproblem

- Gegeben: $W = \{ w_1, \ldots, w_n \} \in \mathbb{R}^n$, alle $w_i > 0$, $M \in \mathbb{R}$
- Gesucht: Alle Teilmengen $U \subseteq W$ mit $\sum_{w \in U} = M$
- Zustandsraum $Z = \{ 0, 1 \}^n$ repräsentiert alle Teilmengen von W
- Zustand $U \subseteq W$ repräsentiert durch

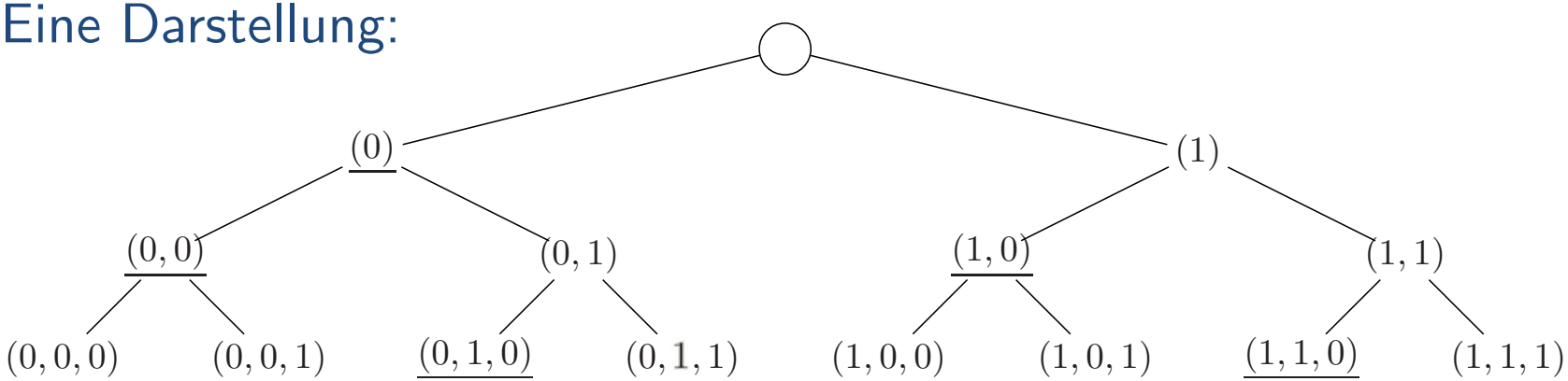
$$(x_1, \ldots, x_n) \qquad \text{mit} \qquad x_i = 1 \Leftrightarrow w_i \in U$$

- Lösungsraum: $\{ (x_1, \ldots, x_n) \mid \sum_{x_i = 1} w_i = M \}$
- Beispiel
  - $W = (11, 13, 24, 7)$, $M = 31$
  - 1. Lösung: $(1, 1, 0, 1)$ entspricht $\{ 11, 13, 7 \}$
  - 2. Lösung: $(0, 0, 1, 1)$ entspricht $\{ 24, 7 \}$

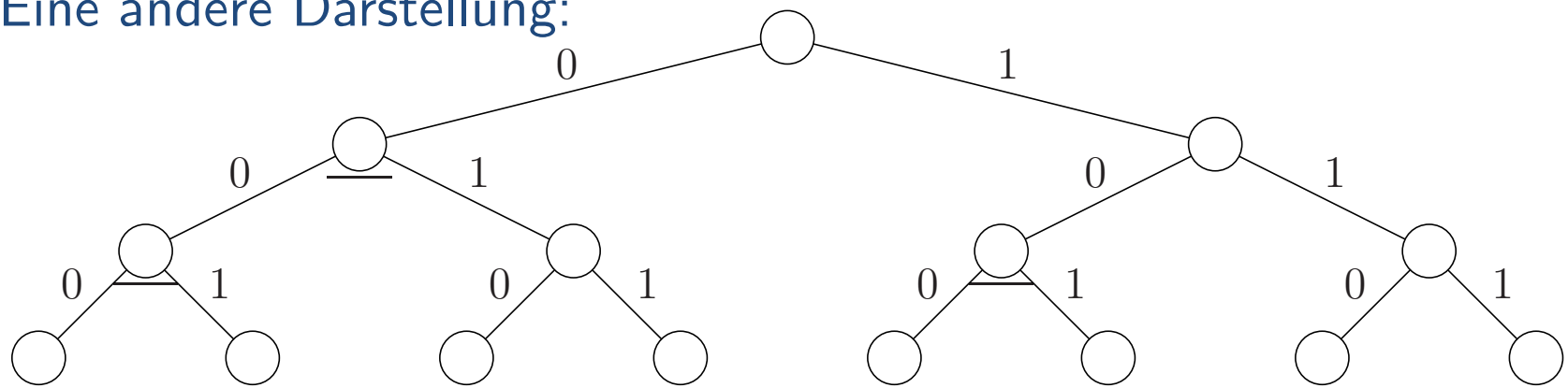
Variante 2: Zustände entsprechen Knoten

- $Z = \{0,1\} \cup \{0,1\}^2 \cup \{0,1\}^3$
- $Z' = \{(0), (0,0), (1,0), (1,1,0), (0,1,0)\} \subseteq Z$ markiert

Eine Darstellung:



Eine andere Darstellung:



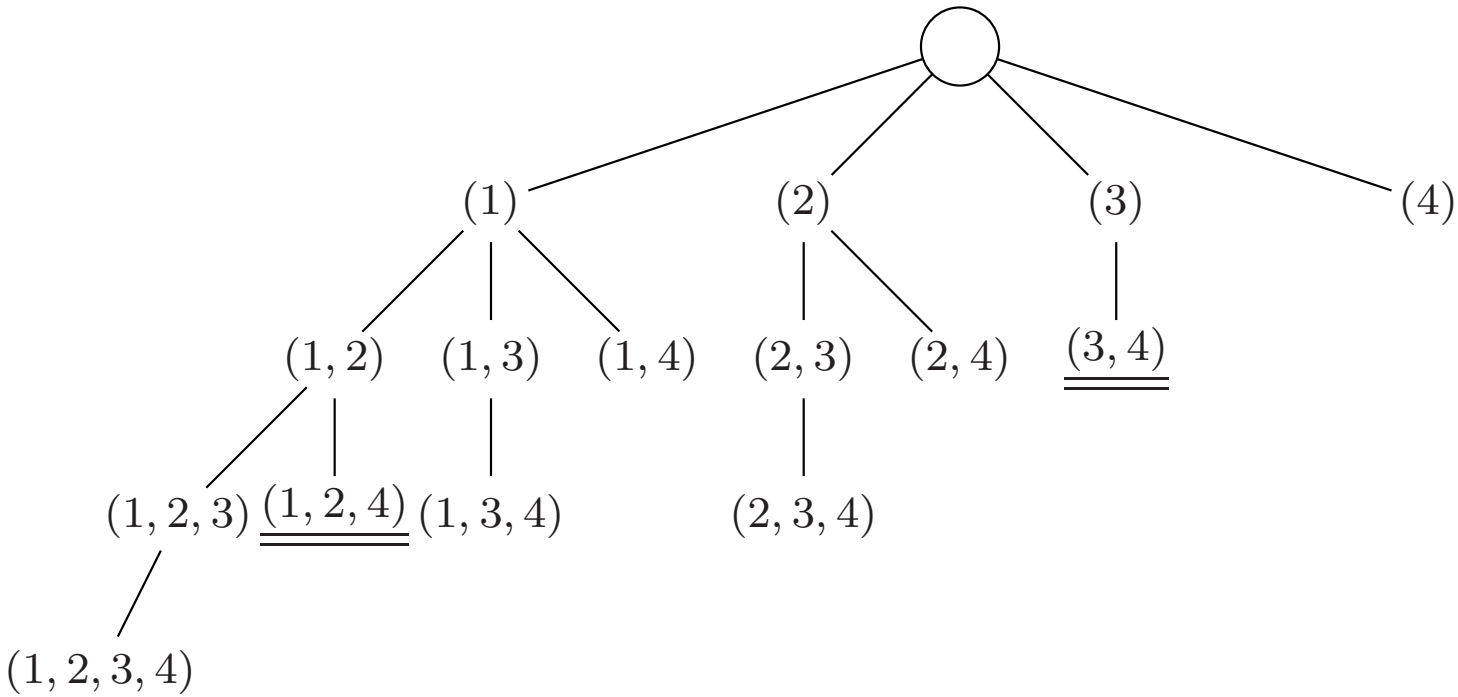
Beispiel: Teilsummenproblem (1)

- Gegeben: $W = \{ w_1, \ldots, w_n \} \in \mathbb{R}^n$, alle $w_i > 0$, $M \in \mathbb{R}$
- Gesucht: Alle Teilmengen $U \subseteq W$ mit $\sum_{w \in U} = M$
- Zustandsraum:

$$Z = \{ (x_1, \ldots, x_k) \mid x_j \in \{ 1, \ldots, n \}, 1 \leq j \leq k \leq n, \\ x_i \neq x_j \text{ für } i \neq j \}$$

  repräsentiert wiederrum alle nicht-leeren Teilmengen von W
- Lösungsraum $\{ (x_1, \ldots, x_k) \mid \sum_{j=1}^{k} w_{x_j} = M \}$
- Beispiel
  - $W = (11, 13, 24, 7)$, $M = 31$
  - 1. Lösung: $(1, 2, 4)$ entspricht $\{ 11, 13, 7 \}$
  - 2. Lösung: $(3, 4)$ entspricht $\{ 24, 7 \}$
- Beobachtung: $(1, 2, 4)$ und $(2, 4, 1)$ beschreiben gleiche Lösung
  $\rightarrow$ Schränke Zustandsraum ein: zusätzlich $x_i < x_j$ für $i < j$

Beispiel: Teilsummenproblem (2)



- Lösungen unterstrichen ($W = (11, 13, 24, 7)$, $M = 31$)
- Beobachtung
  1. Ist ein Knoten eine Lösung, so sind alle Nachfahren keine Lösung
  2. Ist in einem Knoten der Wert M bereits überschritten, so sind alle Nachfahren keine Lösung
- Dadurch Optimierung möglich

Baumerzeugung

- Bäume werden in der Regel dynamisch erzeugt
- Erst die Wurzel, dann weitere Knoten
- Wie explorieren? $\rightarrow$ viele Varianten
- Tiefensuche
  - Niedriger Speicheraufwand (nur aktueller Pfad)
  - Lösungen weit oben im Baum werden ggf. spät gefunden
- Breitensuche
  - Höherer Speicheraufwand (nicht abgearbeitete Zustände)
  - Lösungen weit oben im Baum werden schneller gefunden
- Kostenbasierte Suche
  - Knoten werden "bewertet"
  - Der Knoten mit der besten Bewertung wird expandiert
  - Bewertung mit Hilfe von
    - Heuristiken
    - Kostenfunktion für **Optimierungsprobleme**
- ...

Outline

1. Suchbäume

2. Backtracking

3. Memoization

4. Branch-and-Bound

Backtracking

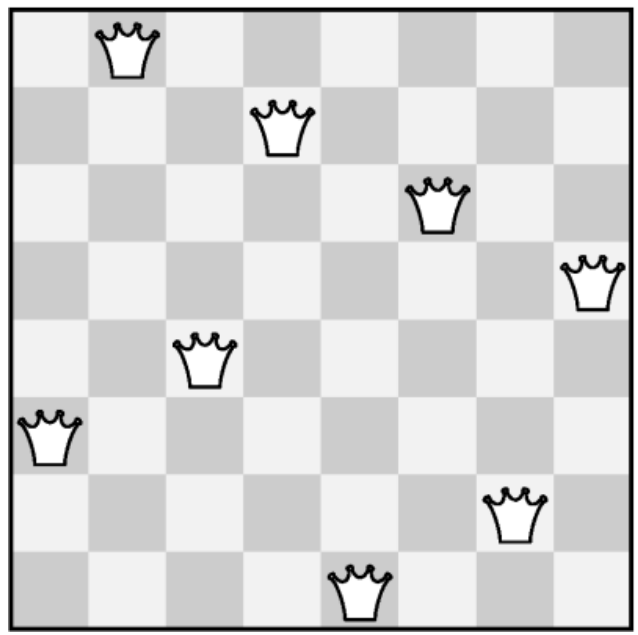
$\textsc{Try}(x_1, \ldots, x_k)$

1: **if** $(x_1, \ldots, x_k)$ is a solution
2: Print "$(x_1, \ldots, x_k)$", optionally halt
3: **for** each $x_{k+1} \in T(x_1, \ldots, x_k)$
4: **if** $B(x_1, \ldots, x_k, x_{k+1})$
5: $\textsc{Try}(x_1, \ldots, x_k, x_{k+1})$

- $T(x_1, \ldots, x_k)$ liefert alle **möglichen Werte** für x_{k+1} unter der Voraussetzung, das $x_1, \ldots, x_k$ schon gewählt sind
- $B(x_1, \ldots, x_k, x_{k+1})$ ist eine **Beschränkungsfunktion**
  - Gibt an, ob Weg $x_1, \ldots, x_k, x_{k+1}$ zu einem Lösungszustand fortgeführt werden kann
  - Wenn $\textsc{False}$, dann keine Lösung mit Präfix $x_1, \ldots, x_k, x_{k+1}$
  - In Praxis oft entscheidend für Laufzeit
- Backtracking entspricht einer Tiefensuche
- Kann verwendet werden, um eine Lösung oder alle Lösungen zu finden

n-Damen-Problem

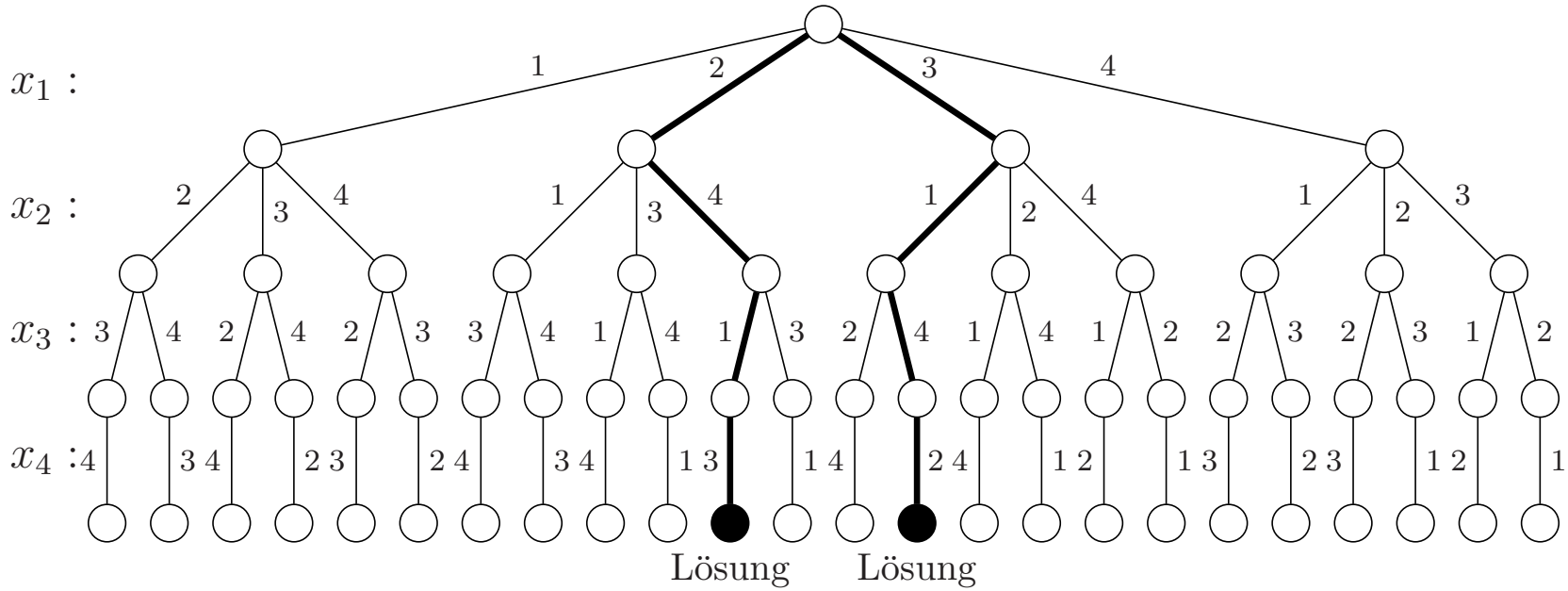
- n Damen auf $n \times n$ so auf Schachfeld platzieren, dass sie sich nicht gegenseitig bedrohen
- Also: Pro Zeile/Spalte/Diagonale nur eine Dame
- Beispiellösung für $n = 8$



Eine einfache Lösung mit Backtracking

- Beobachtung: In jeder Zeile und Spalte muss genau eine Dame stehen
- Zustandsraum somit Menge der Permutationen von $\{1, \ldots, n\}$
  - In Zeile i steht Dame auf Position x_i
  - $x_i \neq x_j$ für $i \neq j$
- Somit ist $T(x_1, \ldots, x_k) = \{1, \ldots, n\} \setminus \{x_1, \ldots, x_k\}$ die Menge der noch nicht verwendeten Spalten
- Einfache Variante: alle Permutationen probieren, d.h. $B(\ldots) = \text{TRUE}$

Beispiel ($n = 4$)



- $n = 4 \rightarrow 24$ mögliche Zustände
- $n = 8 \rightarrow 40\,320$ mögliche Zustände
- $n = 12 \rightarrow 479\,001\,600$ mögliche Zustände
- $n = 20 \rightarrow > 2.4 \cdot 10^{18}$ mögliche Zustände

Eine verbesserte Lösung (1)

- Beschränkungsfunktion wichtig für Backtracking-basierte Verfahren
- Für das n-Damen-Problem
  - Überprüfen, dass neu gesetzte Dame keinen Konflikt hervorruft
  - Zeilen und Spalten konstruktionsbedingt konfliktfrei
  - Es fehlt: Überprüfen, dass Dame $(k+1, x_{k+1})$ auf keiner schon besetzten Diagonale liegt
- Für $n = 8 \rightarrow 2\,057$ Zustände (davon 92 Lösungen, Animation)

Eine verbesserte Lösung (2)

- Dazu /-Diagonalen und \-Diagonalen nummerieren
  - Dame (i, j) liegt auf /-Diagonale $i + j - 1$
  - Dame (i, j) liegt auf \-Diagonale $i - j + n$
  - $B(x_1, \ldots, x_{k+1}) = \text{TRUE}$ gdw. $|k - j + 1| \neq |x_{k+1} - x_j|$ für $1 \leq i \leq k$

(i,j)	1	2	3	4
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6
4	4	5	6	7

$i + j - 1$

(i,j)	1	2	3	4
1	4	3	2	1
2	5	4	3	2
3	6	5	4	3
4	7	6	5	4

$i - j + n$

- Schnellere Implementierung (DEMO)
  - Verwendete Spalten und Diagonalen inkrementell warten
  - 3 Boolesche Felder: *used-col*, *used-diag*, *used-bdiag*
  - Bei jeder Expandierung testen/setzen
  - Bei jedem Backtracking-Schritt löschen

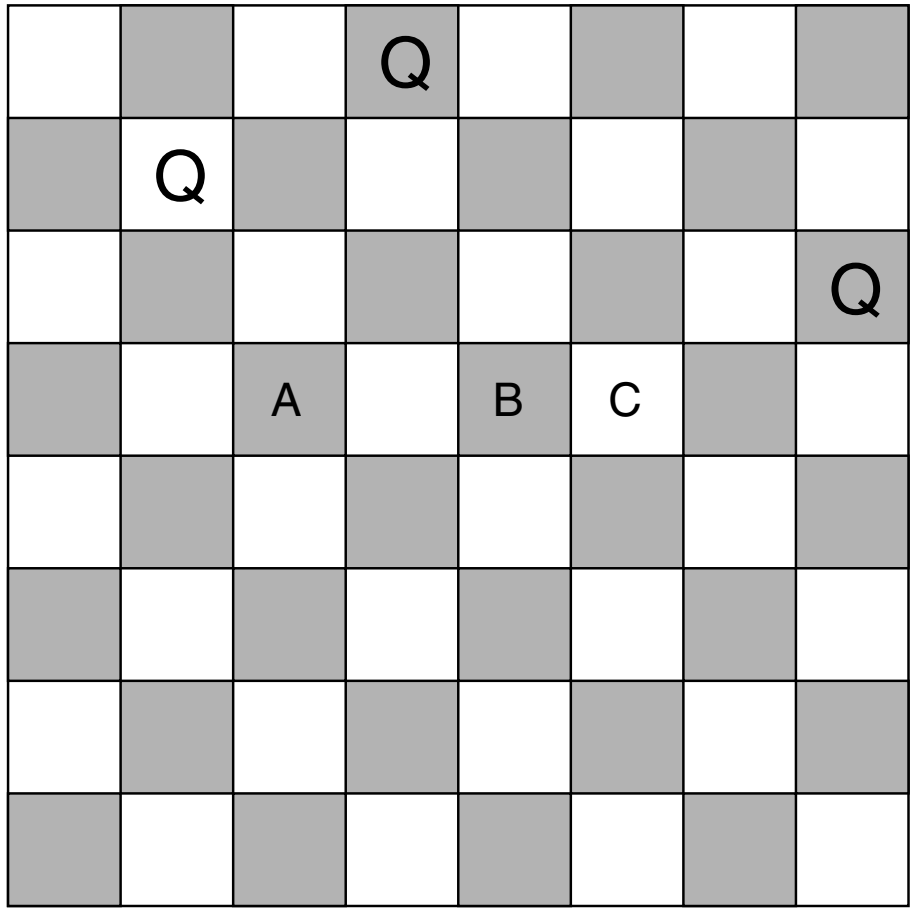
Verwendung von Heuristiken

- Wenn wir nur eine Lösung suchen, ist es hilfreich, vielversprechende Zustände zuerst auszuprobieren
- **Heuristiken** sind Kriterien, Methoden, oder Prinzipien die aus verschiedenen möglichen Aktionen die vielversprechendste heraussuchen
- Repräsentieren Kompromiss zwischen
  - Notwendigkeit von einfachen Kriterien
  - Korrekte Unterscheidung zwischen guten und schlechten Aktionen
- Heuristiken können z.B. Daumenregeln sein
- Wir bewerten also jeden nicht-expandierten Zustand mit Hilfe der Heuristik und wählen dann jeweils den besten aus

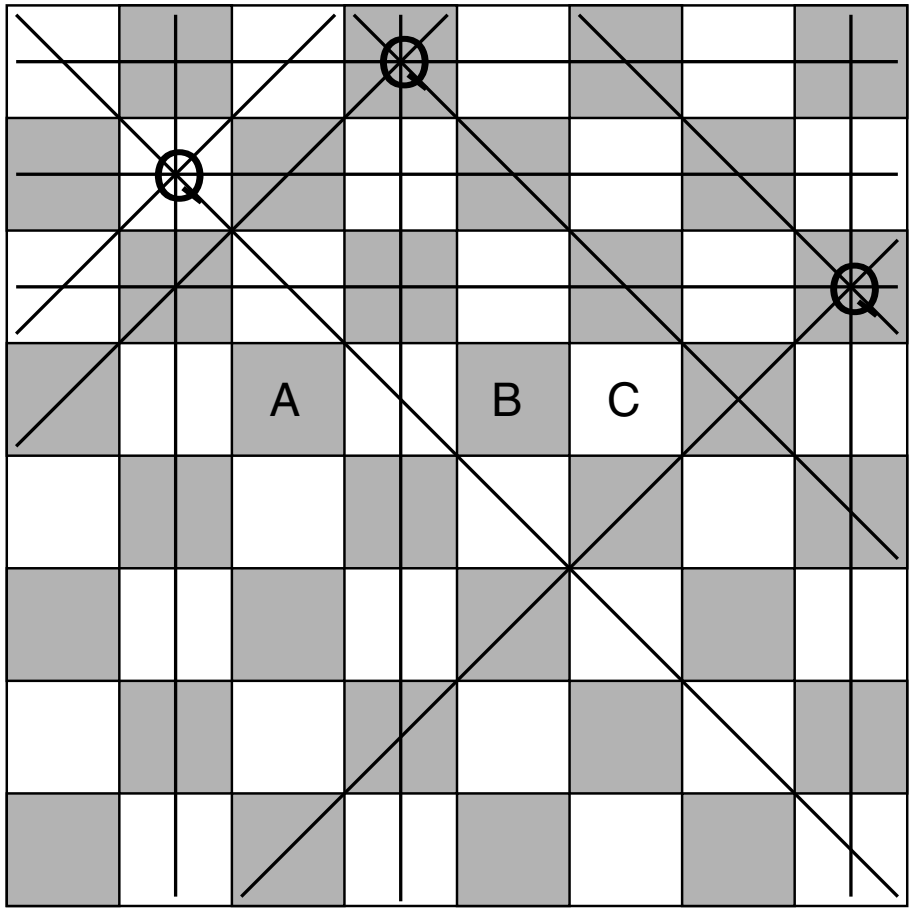
Heuristiken für n-Damen-Problem

- Eine Möglichkeit
  - Wähle in jedem Schritt einen Zustand, der mindestens einen Lösungszustand als Nachfolger hat
  - Garantierte Lösung nach n Schritten
  - Aber: Wie berechnen ohne das Problem selbst zu lösen?
    $\rightarrow$ Kein guter Kompromiss
- Mögliche Ansätze
  - Minimale Anzahl der nicht-bedrohten Felder
  - Minimale Anzahl der nicht bedrohten Felder in einer Zeile
  - Zeilen mit den wenigsten nicht-bedrohten Feldern zuerst abarbeiten

Beispiel (1)

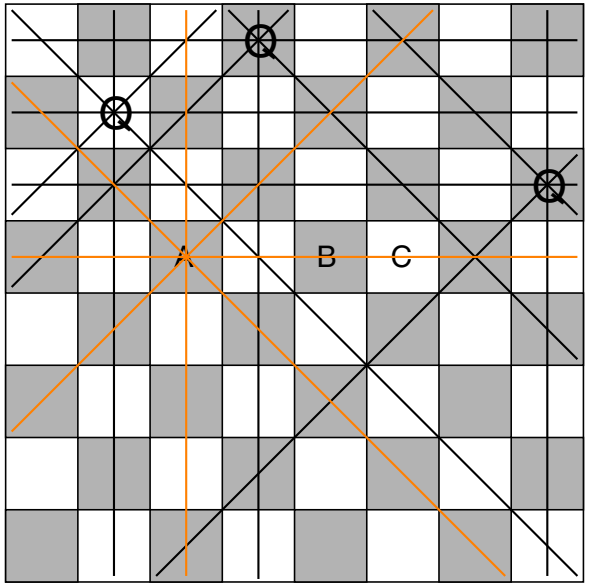


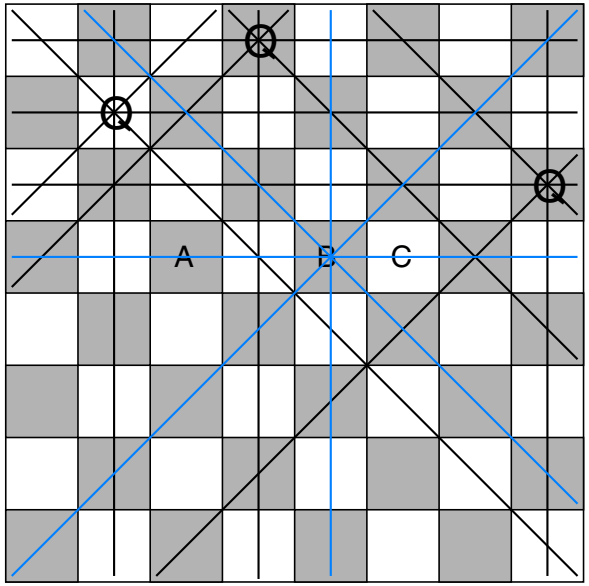
Beispiel (2)

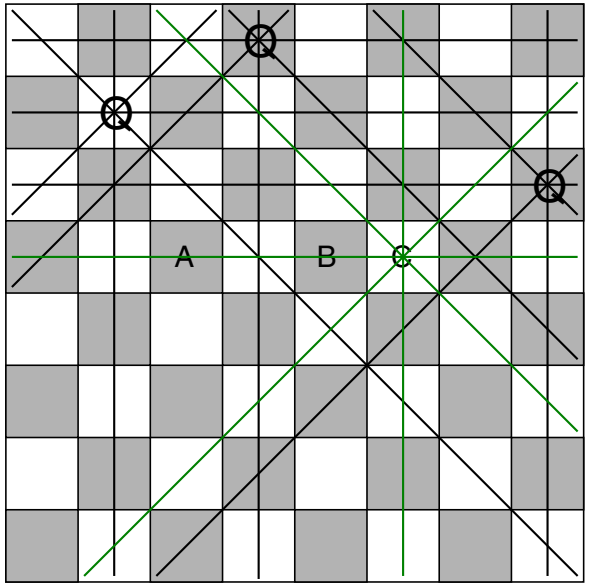


- Momentan: 14 nicht-bedrohte Felder, minimal 3 in einer Zeile

Beispiel (3)



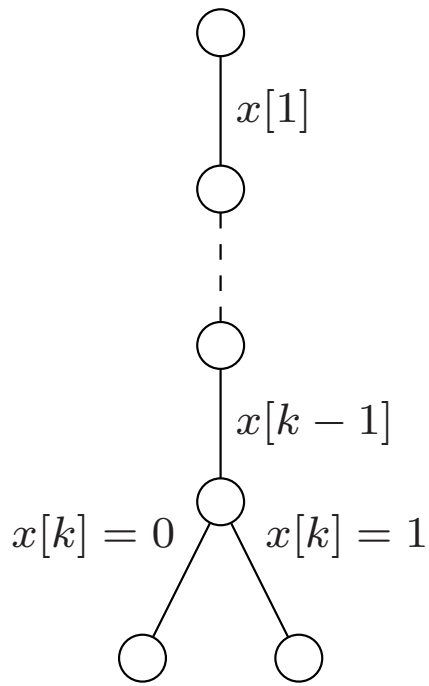




- Wahl von A: 8 nicht-bedrohte Felder, minimal 1 in einer Zeile
- Wahl von B: 9 nicht-bedrohte Felder, minimal 1 in einer Zeile
- Wahl von C: 10 nicht-bedrohte Felder, minimal 2 in einer Zeile

Lösung des Teilsummenproblems mit Backtracking

- Gegeben: $W = \{ w_1, \ldots, w_n \} \in \mathbb{R}^n$, alle $w_i > 0$, $M \in \mathbb{R}$
- Gesucht: Alle Teilmengen $U \subseteq W$ mit $\sum_{w \in U} = M$
- Zustandsraum $Z = \{ 0, 1 \}^n$ repräsentiert alle Teilmengen von W
- Was ist eine gute Beschränkungsfunktion?
  - Sei $x_1, \ldots, x_{k-1}$ die Folge der Kantenmarkierungen
  - Die Entscheidung für x_k steht an



Beschränkungsfunktion (1)

- Ansatz

$$B(x_1, \ldots, x_k) = \text{TRUE} \quad \Leftrightarrow \quad \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M$$

- Beschränkt, wenn alle verbleibenden Gewichte nicht ausreichen, um Gewicht M zu erreichen

Beschränkungsfunktion (2)

- Das lässt sich verbessern
- Seien oBda die Gewichte absteigend sortiert, $w_1 \leq \cdots \leq w_n$
- Dann kann $x_1, \ldots, x_k$ zu keiner Lösung führen, wenn

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > M,$$

  denn dann überschreitet man selbst mit dem kleinsten noch verfügbaren Gewicht die Schranke M
- Somit

$$B(x_1, \ldots, x_k) = \text{TRUE} \quad \Leftrightarrow \quad \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M$$
$$\wedge \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq M$$

Pseudo-Code

- Seien $w_1, \ldots, w_n$ und $x_1, \ldots, x_n$ globale Variablen
- oBdA $w_1 \leq \ldots \leq w_n$ und $w_1 \leq M$
- Sei $s = \sum_{i=1}^{k-1} w_i x_i$ und $r = \sum_{i=k}^{n} x_i$
- Aufruf mit $\textsc{Subset-Sum}(0, 1, \sum_{i=1}^{n} w_i)$

$\textsc{Subset-Sum}(s, k, r)$

1: $x_k \leftarrow 1$
2: **if** $s + w_k = M$
3: Print $\{ w_i \mid 1 \leq i \leq k, x_i = 1 \}$
4: **else if** $s + w_k + w_{k+1} \leq M$ // $s + r \geq M$ known true
5: $\textsc{Subset-Sum}(\underbrace{s + w_k}_{<M}, k + 1, \underbrace{r - w_k}_{>0})$
6: **if** $s + r - w_k \geq M$ and $s + w_{k+1} \leq M$
7: $x_k \leftarrow 0$
8: $\textsc{Subset-Sum}(\underbrace{s}_{<M}, k + 1, \underbrace{r - w_k}_{>0})$

Diskussion

- Abbruch der Rekursion ist gesichert, da in jedem Aufruf $r \neq 0$ und somit $r = \sum_{i=k}^{n} w_i \neq 0$ und somit $k \leq n$
- Beispiel
  - $w = \{ 5, 10, 12, 13, 15, 18 \}$
  - $M = 30$

s	k	r	x	x_k	
0	1	73		1	
5	2	68	1	1	
15	3	58	1,1	1	
15	3	58	1,1	0	
15	4	46	1,1,0	1	
15	4	46	1,1,0	0	
15	5	33	1,1,0,0	1	$\rightarrow$ Ausgabe $\{ 5, 10, 15 \}$
15	5	33	1,1,0,0	0	
5	2	68	1	0	
5	3	58	1,0	1	
17	4	46	1,0,1	1	$\rightarrow$ Ausgabe $\{ 5, 12, 13 \}$
...					

Outline

1. Suchbäume

2. Backtracking

3. Memoization

4. Branch-and-Bound

Memoization

- Manchmal kann man einen Zustand durch verschiedene Tupel beschreiben
- z.B. Teilsummenproblem (naiv)
  - $w = \{1, 3, 13, 7\}$, $M = 20$
  - Tupel = Folge von ausgewählten Gewichten
  - Dann $(1, 7) \equiv (7, 1)$
- Das ist problematisch für die Suche
  - Zustände werden ggf. wiederholt abgearbeitet $\rightarrow$ Effizienz sinkt
  - Manchmal: beliebig lange Tupel möglich $\rightarrow$ Backtracking terminiert ggf. nicht
- Was kann man tun?
  1. Suchraum (und somit Tupel) anpassen
     $\rightarrow$ Gut (siehe Teilsummenproblem), aber nicht immer möglich
  2. **Memoization**
     $\rightarrow$ Bereits abgearbeitete Zustände merken und Lösung bei Wiederauffinden nachschlagen
- Beispiel: Counterfeit Coin Problem

Counterfeit Coin Problem

- Gegeben: 12 Münzen, 11 mit gleichem Gewicht, eine verschieden
- Verfügbar: Eine Balkenwaage
- Gesucht: Die Münze mit dem verschiedenen Gewicht
  - Einfache Lösung: $\leq n$ Messungen (wie?)
- Gesucht: Optimale Strategie, d.h. minimale Anzahl von Messungen

Zustandsraum (1)

- Wie können wir unser Wissen aus bisherigen Messungen repräsentieren? D.h. wie können wir die Menge der möglichen Lösungen beschreiben?
- Wir wiegen M_1 und M_2, Rest ist M_3
- Wenn $w(M_1) < w(M_2)$
  - M_1 enthält gefälschte Münze, wenn diese *zu leicht* ist
  - M_2 enthält gefälschte Münze, wenn diese *zu schwer* ist
  - M_3 enthält nur echte Münzen
- Wenn $w(M_1) = w(M_2)$
  - M_1 und M_2 enthalten nur echte Münzen
  - M_3 enthält gefälschte Münze
- Wenn $w(M_1) > w(M_2)$
  - M_1 enthält gefälschte Münze, wenn diese *zu schwer* ist
  - M_2 enthält gefälschte Münze, wenn diese *zu leicht* ist
  - M_3 enthält nur echte Münzen

Zustandsraum (2)

- Die Münzen lassen sich somit partitionieren
  - L: Menge der Münzen, die möglicherweise zu leicht sind
    $\rightarrow$ Nur in unbalancierten Messungen und immer auf leichter Seite
  - H: Menge der Münzen, die möglicherweise zu schwer sind
    $\rightarrow$ Nur in unbalancierten Messungen und immer auf schwerer Seite
  - R: Menge der echten Münzen
    $\rightarrow$ In einer balancierten Messung oder nicht in einer unbalancierten Messung verwendet oder sowohl auf leichterer und schwererer Seite aufgefunden
  - C: Menge, die gefälschte Münze enthält
    $\rightarrow$ Alle anderen Münzen
- Nach jeder beliebigen Folge von Messungen sind diese Mengen eindeutig bestimmt
- Lösung gefunden wenn $|L \cup H \cup C| = 1$

Zustandsraum (3)

- Uns soll vorerst nur die minimale Anzahl der Messungen interessieren
- Dann merken wir uns nur die Größen der Mengen: $[l, h, c, r]$
- In jedem Schritt teilen wir die Münzen auf
  - Linke Waagschale: $m_1 = l_1 + h_1 + c_1 + r_1$
  - Rechte Waagschale: $m_2 = l_2 + h_2 + c_2 + r_2$
  - $m_1 = m_2 > 0$
  - $\min(r_1, r_2) = 0$
  - $(l_1, h_1, c_1, r_1) + (l_2, h_2, c_2, r_2) \leq (l, h, c, r)$
- Nach Ausgang
  - Wenn $< \rightarrow [l_1 + c_1, h_2 + c_2, 0, r + l_2 + h_1 + c - c_1 - c_2]$
  - Wenn $= \rightarrow [l - l_1 - l_2, h - h_1 - h_2, c - c_1 - c_2, r + m_1 + m_2]$
  - Wenn $> \rightarrow [l_2 + c_2, h_1 + c_1, 0, r + l_1 + h_2 + c - c_1 - c_2]$

Zustandsbaum (1)

- Mit diesem Wissen können wir jetzt einen Zustandsbaum aufbauen
  - Blätter entsprechen Lösungen
  - Innere Knoten entsprechend entweder einem Zustand oder einer Messung (alternierend)
- Beispiel ($n = 4$)
  - Kinder der Wurzel $[0, 0, 4, 0]$
    1. Jeweils eine Münze: Vergleiche $(0, 0, 1, 0)$ mit $(0, 0, 1, 0)$
    2. Jeweils zwei Münzen: Vergleiche $(0, 0, 2, 0)$ mit $(0, 0, 2, 0)$
  - Kinder von (1)
    - $<$: Zustand $[1, 1, 0, 2]$
    - $=$: Zustand $[0, 0, 2, 2]$
    - $>$: Zustand $[1, 1, 0, 2] \rightarrow$ Wiederholter Zustand
  - Kinder von (2)
  - $<$: Zustand $[2, 2, 0, 0]$
  - $=$: nicht möglich
  - $>$: Zustand $[2, 2, 0, 0] \rightarrow$ Wiederholter Zustand

Zustandsbaum (2)

- Wir haben gesehen: Mehrere Folgen von Messungen können zu dem gleichen Ergebnis führen
- Außerdem: Endlose Vergleiche möglich
  - Betrachte Zustand $(2, 2, 0, 0)$
  - Und die Messung $(2, 0, 0, 0)$ und $(0, 2, 0, 0)$
    $\rightarrow$ Wir legen die potentiell leichten Münzen in die eine, die potentiell schweren Münzen in die andere Waagschale
  - Dadurch lernen wir nichts neues: Folgezustand ist $[2, 2, 0, 0]$
- Memoization hilft hier
  - Wir merken uns alle schon entdeckten Zustände und ihre Position im Baum
  - Wenn wir einen Zustand wiederholt erreichen, bearbeiten wir ihn nicht erneut sondern markieren ihn als "wiederholt"
  - Teilbaum des wiederholten Zustands wird durch gemerkten Repräsentanten beschrieben
  - Somit: Jedes Teilproblem nur einmal gelöst

Vom Zustandsbaum zur Lösung (1)

- Was nützt uns der vollständig berechnete Suchbaum?
- Idee: Wir markieren jeden Zustand mit der Anzahl der Messungen, die im ungünstigsten Fall noch zur Lösung notwendig sind = **Kosten**
- Das können wir in Bottom-Up tun
  - Blätter sind gelöst (0)
  - Vergleichsknoten: Alle Teilprobleme müssen gelöst werden
    - Im Worst Case ist das Messergebnis der Sohn mit den höchsten Kosten
    - Also: Markieren mit dem Maximum seiner Söhne...
    - ...plus eins für den Vergleichsknoten selbst
  - Zustandsknoten
    - Wir können einen geeigneten Sohn (=Vergleich) wählen
    - Wir wählen den Vergleich mit den minimalen Kosten
    - Also: Wir markieren den Knoten mit dem Minimum seiner Söhne

Vom Zustandsbaum zur Lösung (2)

- Nach Kostenberechnung ist die Wurzel mit der maximal notwendigen Anzahl Messungen markiert
- Welche Messungen führt man für eine Instanz durch?
  1. Suchbaum zum **Entscheidungsbaum** umwandeln
     $\rightarrow$ Sohn jedes Zustandsknotens ist dann nur der Vergleich mit den minimalen Kosten
  2. Für gegebene Instanz die Mengen L, H, C, und R explizit warten und Aufbau der nächsten Messung jeweils im Entscheidungsbaum nachschlagen (basierend auf $|L|$, $|H|$, $|C|$, $|R|$)
- Siehe DEMO

Outline

1. Suchbäume

2. Backtracking

3. Memoization

4. Branch-and-Bound

Outline

1. Suchbäume
2. Backtracking
3. Memoization
4. Branch-and-Bound