

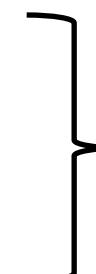
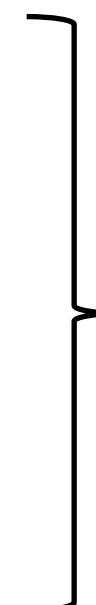
Algorithmen und Datenstrukturen

SS 2018

Robert Elsässer



Inhaltsangabe

- Einleitung, Motivation
 - Pseudocode, Invarianten, Laufzeitanalyse
 - Gro β -O-Notation
- 
- Grundlagen**
-
- Inkrementelle Algorithmen, Insertion-Sort
 - Divide & Conquer Algorithmen, Merge-Sort
 - Quick-Sort Analyse und Varianten
 - Rekursionsgleichungen, Master-Theorem
 - Algorithmen mit Datenstrukturen, Heaps, Heap-Sort
 - Untere Schranke für Vergleichssortierer
 - Counting-Sort
- 
- Sortieralgorithmen**

- ADTs und Datenstrukturen, Stacks, Queues, Listen, Bäume
 - Hashtabellen
 - Binäre Suchbäume
- 
- Elementare Graphalgorithmen, Breitensuche
 - Tiefensuche
 - Zusammenhangskomponenten
 - Minimale Spannbäume, Algorithmus von Prim
 - Algorithmus von Kruskal, disjunkte Vereinigungsmengen
- 
- Gierige Algorithmen, Scheduling-Probleme
 - Dynamische Programmierung, Längste gemeinsame Teilfolge
 - Optimale Suchbäume
 - Rucksack-Problem
- 

1. Einführung

- Was ist ein Algorithmus (eine Datenstruktur)?
- Welche Probleme kann man damit lösen?
- Warum betrachten wir (effiziente) Algorithmen?
- Wie beschreiben wir Algorithmen?
- Nach welchen Kriterien beurteilen wir Algorithmen?
- Welche Algorithmen betrachten wir?
- Wie passt Algorithmen und Datenstrukturen ins Informatikstudium?

Was ist ein Algorithmus?

Definition 1.1

Ein *Algorithmus* ist eine eindeutige Beschreibung eines Verfahrens zur Lösung einer bestimmten Klasse von Problemen.

Genauer:

Ein Algorithmus ist eine Menge von Regeln für ein Verfahren, um aus gewissen Eingabegrößen bestimmte Ausgabegrößen herzuleiten.

Dabei muss:

1. das Verfahren in einem endlichen Text beschreibbar sein.
2. jeder Schritt des Verfahrens auch tatsächlich ausführbar sein.
3. der Ablauf des Verfahrens zu jedem Zeitpunkt eindeutig definiert sein.

Beispiel Sortieren

Eingabe bei Sortieren:

Folge von n Zahlen (a_1, a_2, \dots, a_n) .

Ausgabe bei Sortieren:

Umordnung (b_1, b_2, \dots, b_n) der Eingabefolge, sodass $b_1 \leq b_2 \leq \dots \leq b_n$.

Sortieralgorithmus:

Verfahren, das zu jeder Folge (a_1, a_2, \dots, a_n) die sortierte Umordnung (b_1, b_2, \dots, b_n) berechnet.

Eingabe: (31,41,59,26,51,48)

Ausgabe: (26,31,41,48,51,59)

Was ist ein Algorithmus?

Definition 1.2

Eine Datenstruktur ist eine bestimmte Art, Daten im Speicher eines Computers anzugeordnen, so dass Operationen wie z.B. Suchen, Einfügen, Löschen einfach zu realisieren sind.

Einfache Beispiele:

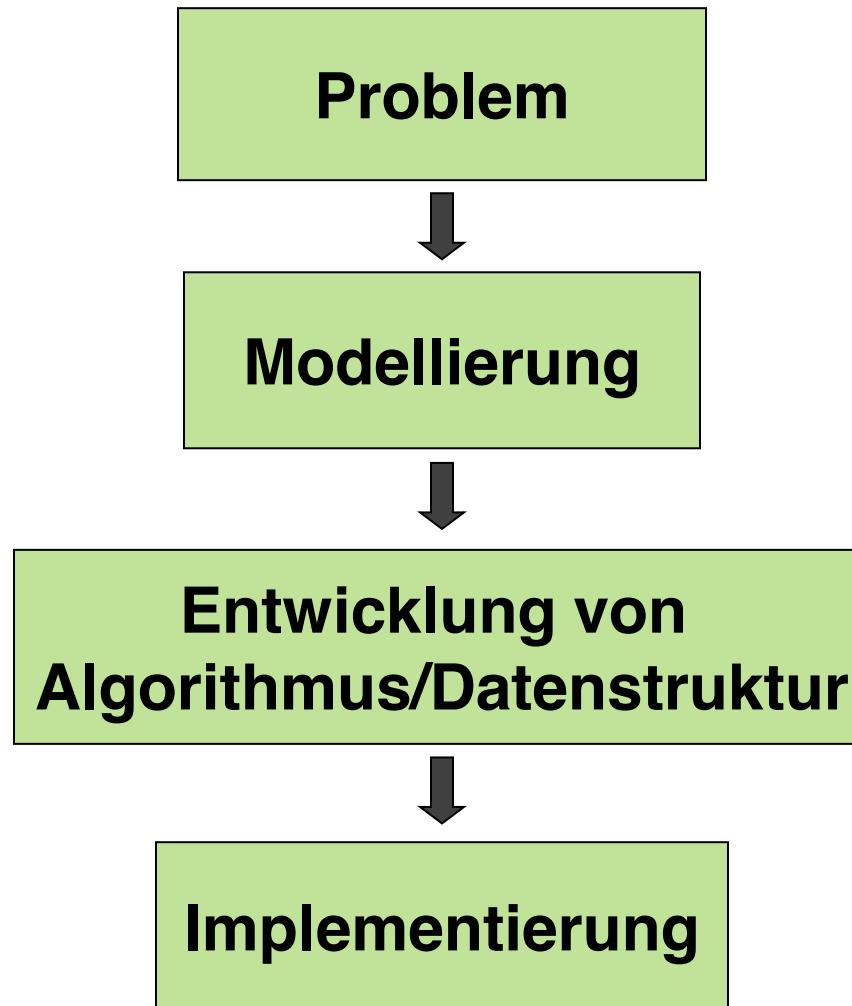
- Listen
- Arrays

Algorithmen und Datenstrukturen sind eng verbunden:
Gute Datenstrukturen häufig unerlässlich für gute Algorithmen.

Beispielprobleme

- Wie findet ein Navigationssystem gute Verbindungen zwischen zwei Orten?
- Wie werden im Internet Informationen geroutet?
- Wie berechnet ein Unternehmen eine möglichst gute Aufteilung seiner Ressourcen, um seinen Gewinn zu maximieren?
- Wie werden etwa in Google Informationen schnell gefunden?
- Wie werden Gleichungssysteme der Form $Ax = b$ gelöst?

Softwareentwicklung



Kriterien für Algorithmen

- Algorithmen müssen korrekt sein.
 - Benötigen *Korrektheitsbeweise*.
- Algorithmen sollen *zeit- und speichereffizient* sein.
 - Benötigen Analysemethoden für Zeit-/Speicherbedarf.
- Analyse basiert *nicht auf* empirischen Untersuchungen, sondern auf mathematischen Analysen.
- Nutzen hierfür **Pseudocode** und **Basisoperationen**.

Algorithmenentwurf

Zum Entwurf eines Algorithmus gehören:

1. Beschreibung des Algorithmus
2. Korrektheitsbeweis
3. Zeit- bzw. Speicherbedarfsanalyse

Beschreibung von Algorithmen

- Zunächst informell (mathematisches) Verfahren zur Lösung.
- Dann Präzisierung durch Pseudocode.
- Meist *keine* Beschreibung durch Programmcode in Java oder C, C++ in der Vorlesung.

„Architekten“ in der Informatik



Tom Leighton

- CEO Akamai Technologies
- Professor am MIT

Autor des Buches

„Parallel Algorithms and Architectures“

Quelle: mitx.org

„Architekten“ in der Informatik



Leslie Lamport

- Researcher, Microsoft Research
- Turing Award 2013:

„He devised important algorithms and developed formal modeling and verification protocols that improve the quality of distributed systems.“

Quelle: microsoft.com

„Architekten“ in der Informatik



Monika Henzinger

- Director of Research,
Google, 2001-2005
- Seit 2009 Professorin an
der Uni Wien

Quelle: Uni Wien

Wozu gute Algorithmen?

Computer werden zwar immer schneller und Speicher immer größer und billiger, **aber**:

- Geschwindigkeit und Speicher werden **immer** begrenzt sein.
- Daher muss mit den Ressourcen **Zeit** und **Speicher** geschickt umgegangen werden.
- Außerdem werden in Anwendungen immer größere Datenmengen verarbeitet.
- Diese wachsen schneller als Geschwindigkeit oder Speicher von Computern.

Effekt guter Algorithmen - Sortieren

Insertion-Sort:

Sortiert n Zahlen mit $c_1 n^2$ Vergleichen.

Computer A:

10^9 Vergleiche/Sek.

$$c_1 = 2, n = 10^6.$$

Dann benötigt A:

$$\frac{2 \cdot (10^6)^2}{10^9} = 2000 \text{ Sek.}$$

Merge-Sort:

Sortiert n Zahlen mit $c_2 n \log(n)$ Vergleichen.

Computer B:

10^7 Vergleiche/Sek.

$$c_2 = 50, n = 10^6.$$

Dann benötigt B:

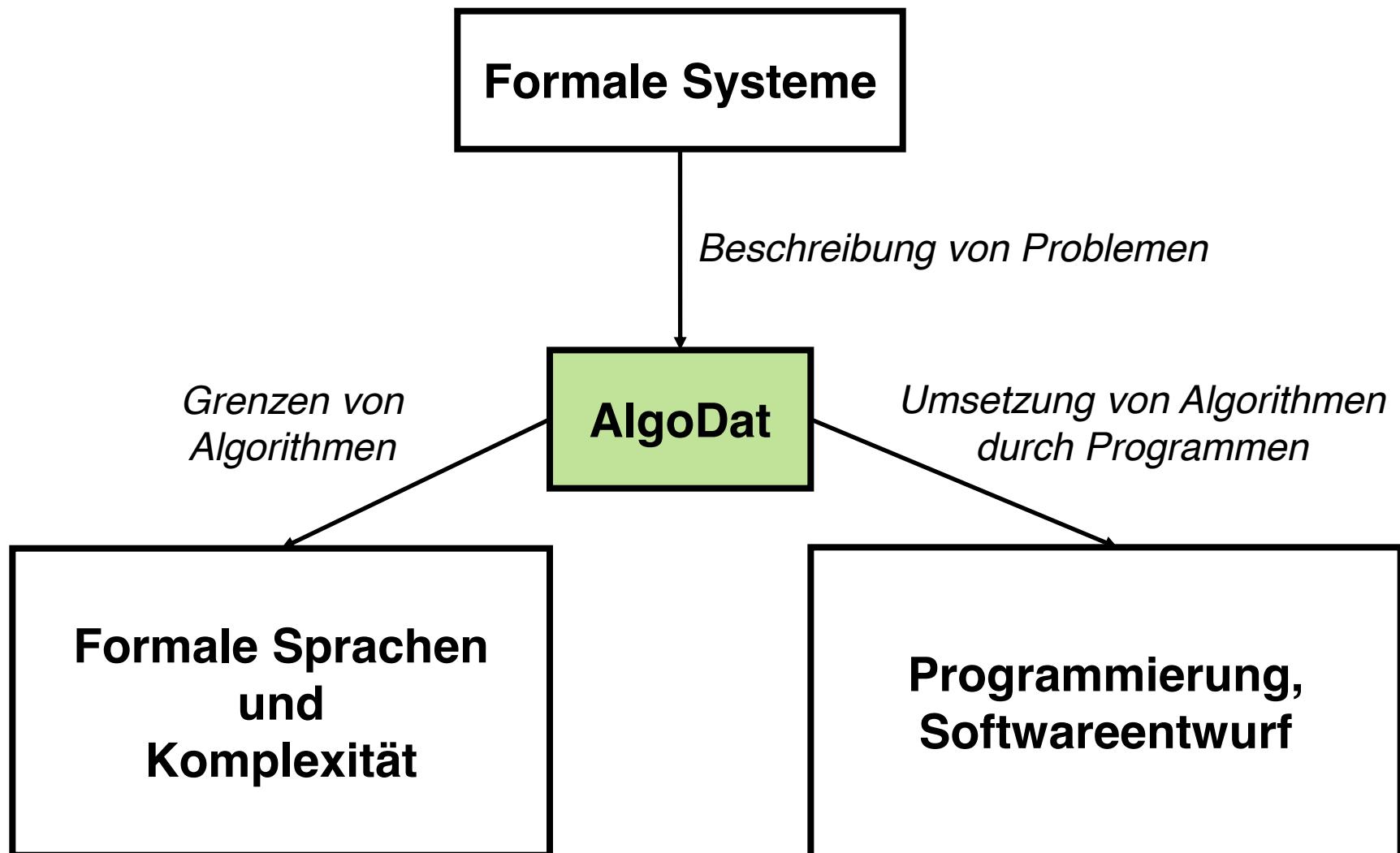
$$\frac{50 \cdot (10^6) \log(10^6)}{10^7} \approx 100 \text{ Sek.}$$

Probleme, Algorithmen, Ziele

Ziel der Vorlesung ist es

1. Wichtige Probleme, Algorithmen und Datenstrukturen kennen zu lernen.
2. Wichtige Algorithmentechniken und Entwurfsmethoden kennen und *anwenden* zu lernen.
3. Wichtige Analysemethoden kennen und *anwenden* zu lernen.
4. Das Zusammenspiel von Algorithmen und Datenstrukturen zu verstehen.

AlgoDat + Informatik-Studium



2. Grundlagen

- Beschreibung von Algorithmen durch **Pseudocode**.
- Korrektheit von Algorithmen durch **Invarianten**.
- Laufzeitverhalten beschreiben durch **O-Notation**.

Beispiel Minimum-Suche

Eingabe bei Minimum-Suche:

Folge von n Zahlen (a_1, a_2, \dots, a_n) .

Ausgabe bei Minimum-Suche:

Index i , so dass $a_i \leq a_j$ für alle Indizes $1 \leq j \leq n$.

Minimumsalgorithmus:

Verfahren, das zu jeder Folge (a_1, a_2, \dots, a_n) den Index eines kleinsten Elements berechnet.

Eingabe: (31,41,59,26,51,48)

Ausgabe: 4

Min-Search in Pseudocode

Min-Search(A)

```
1   $min = 1$ 
2  for  $j = 2$  to  $A.length$ 
3      if  $A[j] < A[min]$ 
4           $min = j$ 
5  return  $min$ 
```

Pseudocode

- **Schleifen (for, while, repeat – until)**
- **Bedingtes Verzweigen (if – else)**
- **(Unter-)Programmaufruf/Übergabe (return)**
- **Zuweisung** durch: =
- **Kommentar** durch: //
- **Daten** als Objekte mit einzelnen Feldern oder Eigenschaften
(z.B. $A.length :=$ Länge des Arrays A)
- **Blockstruktur** durch **Einrückung**

Min-Search

Min-Search(A)

```
1   $min = 1$ 
2  for  $j = 2$  to  $A.length$ 
3      if  $A[j] < A[min]$ 
4           $min = j$ 
5  return  $min$ 
```

Eingabe: (31,41,59,**26**,51,48)

$min = 4$

Pseudocodes

Algorithm 1: Modified value iteration algorithm

Input: A weighted graph (G, w) , a sorted list A of admissible values for the minimal energies

Output: The minimal energy of (G, w)

```
1  $e(u) \leftarrow \min A$  for every  $u \in V$                                 // Initialization
   // Repeat as long as some node  $u$  violates the first two conditions
   // of Lemma 7
2 while there is a node  $u \in V$  such that  $u \in V_A$  and
   ( $u \in V_A$  and  $\forall(u, v) \in E : e(u) + w(u, v) < e(v)$ ) or
   ( $u \in V_B$  and  $\exists(u, v) \in E : e(u) + w(u, v) < e(v)$ ) do
   | // Update  $e(u)$ 
3   if  $u \in V_A$  then
4     |  $e(u) \leftarrow \min_{(u,v) \in E} (e(v) - w(u, v))$ 
5   else if  $u \in V_B$  then
6     |  $e(u) \leftarrow \max_{(u,v) \in E} (e(v) - w(u, v))$ 
7   | // Increase  $e(u)$  to next admissible value
8   |  $e(u) \leftarrow \min\{r \in A \mid r \geq e(u)\}$ 
9 return  $e$ 
```

Quelle: Algorithmica, Springer Verlag

Invarianten

Definition 2.1

Eine (Schleifen-) **Invariante** ist eine Eigenschaft eines Algorithmus, die vor und nach jedem Durchlaufen einer Schleife erhalten bleibt.

- Invarianten dienen dazu, die Korrektheit von Algorithmen zu beweisen.
- Sie werden in der Vorlesung immer wieder auftauchen und spielen eine große Rolle.

Invarianten und Korrektheit

Invariante und Korrektheit von Algorithmen wird bewiesen, indem **gezeigt** wird, dass

- Die Invarianten vor dem ersten Schleifendurchlauf erfüllt ist
 → **Initialisierung**
- Die Eigenschaft bei jedem Schleifendurchlauf erhalten bleibt
 → **Erhaltung**
- Die Invariante nach Beendigung der Schleife etwas über die Ausgabe des Algorithmus aussagt, Algorithmus korrekt ist
 → **Terminierung**

Invariante bei Min-Search

Invariante:

Vor Schleifendurchlauf mit Index i ist $A[\min]$ kleinstes Element in $A[1 \dots i - 1]$.

Initialisierung:

Der kleinste Index für die Schleife ist $i = 2$.
Davor ist $A[\min] = A[1]$.

Erhaltung:

`if`-Abfrage ersetzt korrekt Minimum, wenn zusätzlich $A[i]$ betrachtet wird.

Terminierung:

Vor Durchlauf mit $i = n + 1$ ist $A[\min]$ das Minimum der Zahlen in $A[1 \dots n]$.

Laufzeitanalyse und Rechenmodell

Für eine **präzise mathematische Laufzeitanalyse** benötigen wir ein **Rechenmodell**, das definiert:

- welche Operationen zulässig sind,
- welche Datentypen es gibt,
- wie Daten gespeichert werden und
- wie viel Zeit Operationen auf bestimmten Daten benötigen.

Formal ist ein solches Rechenmodell gegeben durch die **Random Access Maschine (RAM)**.

RAMs sind Idealisierung von 1 - Prozessorrechner mit einfachem aber unbegrenzt großem Speicher.

Basisoperationen – Kosten

Definition 2.2

Als Basisoperationen bezeichnen wir

- **Arithmetische Operationen:**
Addition, Multiplikation, Division, Ab-, Aufrunden
- **Datenverwaltung:**
Laden, Speichern, Kopieren
- **Kontrolloperationen:**
Verzweigungen, Programmaufrufe, Wertübergaben

Kosten:

Zur Vereinfachung nehmen wir an, dass jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt.

In weiterführenden Veranstaltungen werden Sie andere (und häufig realistischere) Kostenmodelle kennen lernen.

Eingabegröße – Laufzeit

Definition 2.3

Die Laufzeit $T(I)$ eines Algorithmus A bei Eingabe I ist definiert als die Anzahl von Basisoperationen, die Algorithmus A zur Berechnung der Lösung bei Eingabe I benötigt.

Definition 2.4

Die (worst-case) Laufzeit eines Algorithmus A ist eine Funktion $T: N \rightarrow R^+$, wobei

$$T(n) := \max\{T(I): I \text{ hat Eingabegröße } \leq n\}$$

Eingabegröße – Laufzeit

- **Laufzeit** angegeben als **Funktion der Größe der Eingabe**.
- Eingabegröße **abhängig vom Problem** definiert.
- Eingabegröße Minimumsuche = Größe des Arrays.
- **Laufzeit bei Minimumsuche:**
 A Array, für das Minimum bestimmt werden soll.

$T(A) :=$ Anzahl der Operationen, die zur Bestimmung
des Minimums in A benötigt werden.

Satz 2.5

Algorithmus Min-Search hat Laufzeit $T(n) \leq an + b$ für Konstanten a, b .

Minimum-Suche

Min-Search(A)	cost	times
1 $min = 1$	c_1	1
2 for $j = 2$ to $A.length$	c_2	n
3 if $A[j] < A[min]$	c_3	$n - 1$
4 $min = j$	c_4	t
5 return min	c_5	1

Hierbei ist t die **Anzahl der Minimumwechsel**.

Es gilt: $t \leq n - 1$

O -Notation

Definition 2.6

Sei $g: N \rightarrow R^+$ eine Funktion.

Dann bezeichnen wir mit $O(g(n))$ die folgende Menge von Funktionen:

$$O(g(n)) := \left\{ f(n) : \begin{array}{l} \text{Es existieren Konstanten } c > 0, n_0, \text{ sodass} \\ \text{für alle } n \geq n_0 \text{ gilt } 0 \leq f(n) \leq c g(n) \end{array} \right\}$$

- $O(g(n))$ formalisiert:
Die Funktion $f(n)$ wächst asymptotisch nicht schneller als $g(n)$.
- Statt $f(n)$ in $O(g(n))$ in der Regel $f(n) = O(g(n))$.

Ω -Notation

Definition 2.7

Sei $g: N \rightarrow R^+$ eine Funktion.

Dann bezeichnen wir mit $\Omega(g(n))$ die folgende Menge von Funktionen:

$$\Omega(g(n)) := \left\{ f(n) : \begin{array}{l} \text{Es existieren Konstanten } c > 0, n_0, \text{ sodass} \\ \text{für alle } n \geq n_0 \text{ gilt } 0 \leq c g(n) \leq f(n) \end{array} \right\}$$

- $\Omega(g(n))$ formalisiert:
Die Funktion $f(n)$ wächst asymptotisch mindestens so schnell wie $g(n)$.
- Statt $f(n)$ in $\Omega(g(n))$ in der Regel $f(n) = \Omega(g(n))$.

Θ -Notation

Definition 2.8

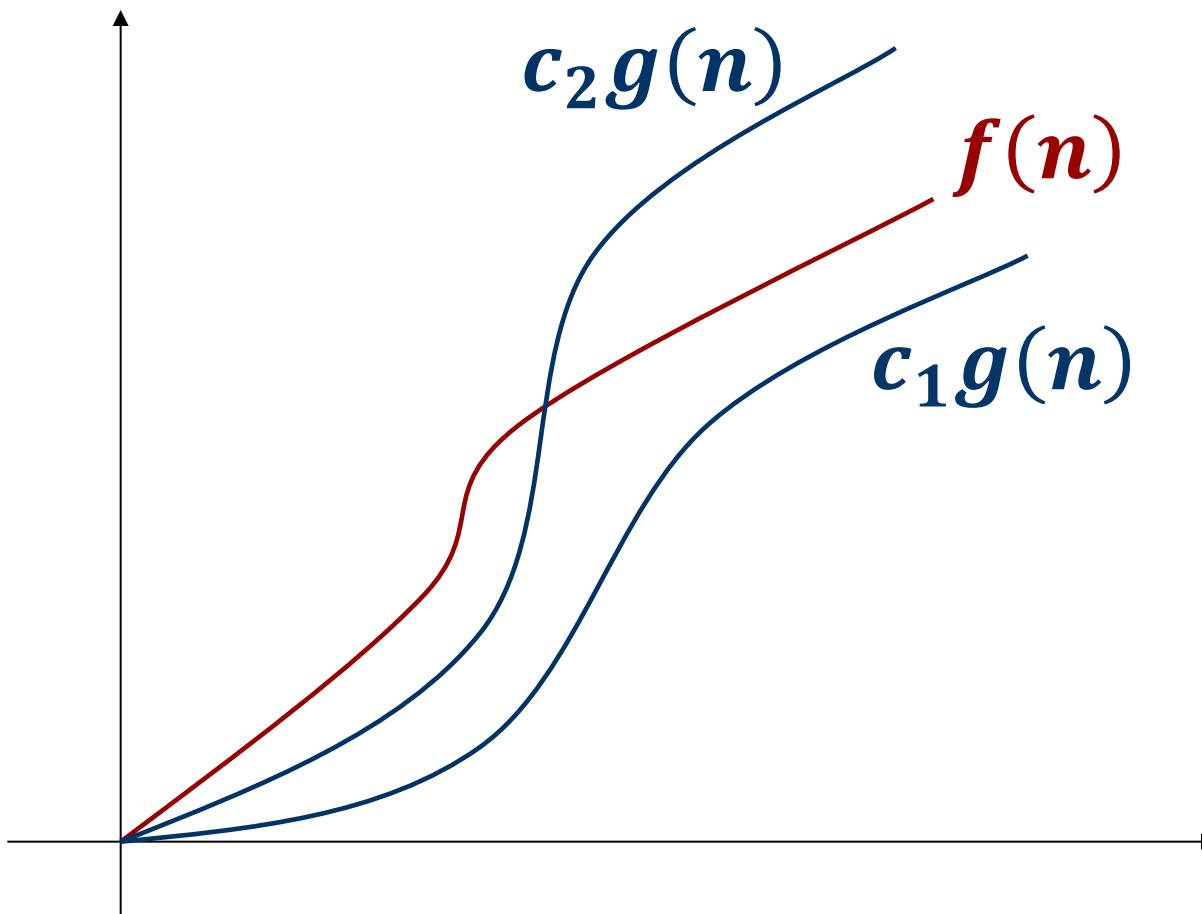
Sei $g: N \rightarrow R^+$ eine Funktion.

Dann bezeichnen wir mit $\Theta(g(n))$ die folgende Menge von Funktionen:

$$\Theta(g(n)) := \left\{ f(n) : \begin{array}{l} \text{Es existieren Konstanten } c_1 > 0, c_2, n_0, \text{ sodass f\"ur} \\ \text{alle } n \geq n_0 \text{ gilt } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \end{array} \right\}$$

- $\Theta(g(n))$ formalisiert:
Die Funktion $f(n)$ w\"achst asymptotisch genau so schnell wie $g(n)$.
- Statt $f(n)$ in $\Theta(g(n))$ in der Regel $f(n) = \Theta(g(n))$.

Illustration von $\Theta(g(n))$



Regeln für Kalküle - Transitivität

O -, Ω - und Θ -Kalkül sind **transitiv**, d.h.:

- Aus $f(n) = O(g(n))$ und $g(n) = O(h(n))$
folgt $f(n) = O(h(n))$.
- Aus $f(n) = \Omega(g(n))$ und $g(n) = \Omega(h(n))$
folgt $f(n) = \Omega(h(n))$.
- Aus $f(n) = \Theta(g(n))$ und $g(n) = \Theta(h(n))$
folgt $f(n) = \Theta(h(n))$.

Regeln für Kalküle - Reflexivität

- 0-, Ω - und Θ -Kalkül sind **reflexiv**, d.h.:

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

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

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

- Θ -Kalkül ist **symmetrisch**, d.h.:

$$f(n) = \Theta(g(n)) \text{ genau dann, wenn } g(n) = \Theta(f(n)).$$

Regeln für Kalküle

Satz 2.10

Sei $f: N \rightarrow R^+$ mit $f(n) \geq 1$ für alle n .

Weiter sei $k, l \geq 0$ mit $k \geq l$. Dann gilt:

1. $f(n)^l = O(f(n)^k)$
2. $f(n)^k = \Omega(f(n)^l)$

Satz 2.11

Seien $\varepsilon, k > 0$ beliebig. Dann gilt:

1. $\log(n)^k = O(n^\varepsilon)$
2. $n^\varepsilon = \Omega(\log(n)^k)$

Anwendung auf Laufzeiten – Min-Search

Satz 2.11

Minimum-Search besitzt Laufzeit $\Theta(n)$.

Zum Beweis ist zu zeigen:

1. Es gibt ein c_2 , so dass die Laufzeit von Min-Search bei allen Eingaben der Größe n immer höchstens $c_2 n$ ist.
2. Es gibt ein c_1 , so dass für alle n eine Eingabe I_n der Größe n existiert bei der Min-Search mindestens Laufzeit $c_1 n$ besitzt.

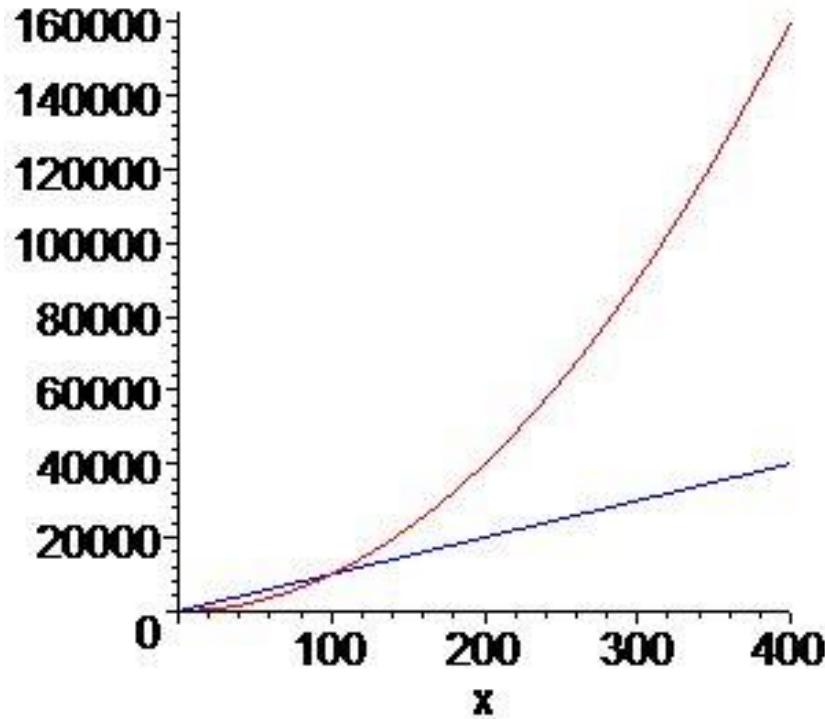
Anwendung auf Laufzeiten

- **O -Notation** erlaubt uns, **Konstanten zu ignorieren**.
- Wollen uns auf **asymptotische Laufzeit** konzentrieren.
- Werden in Zukunft **Laufzeiten immer** mit Hilfe von **O -, Ω - und Θ -Notation** angeben.

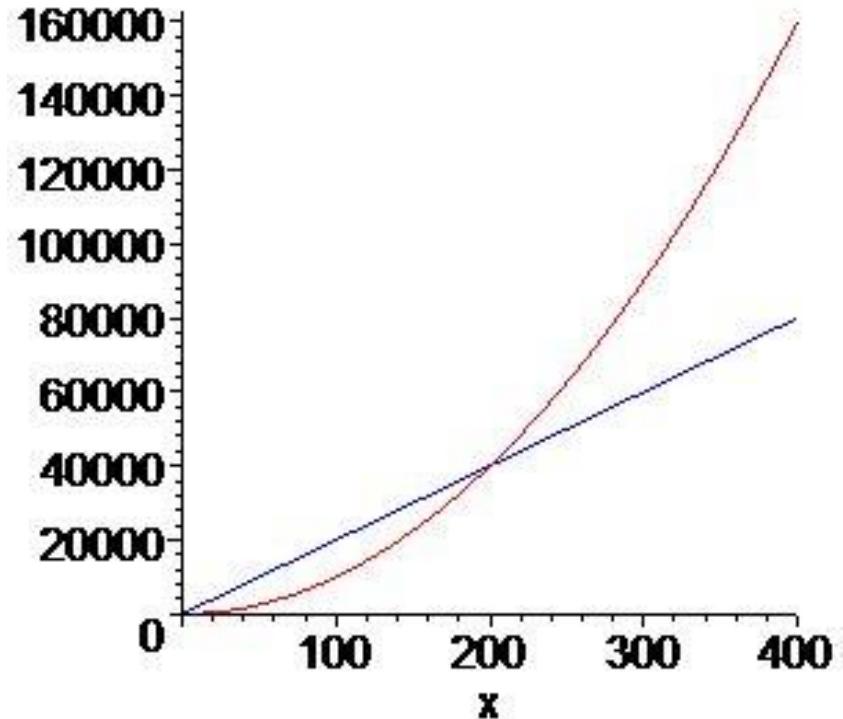
Wiederholung

- Beschreibung von Algorithmen durch **Pseudocode**.
- Korrektheit von Algorithmen durch **Invarianten**.
- Laufzeitverhalten beschreiben durch **O -Notation**.
 - Wollen uns auf asymptotische Laufzeit konzentrieren.
 - Werden in Zukunft Laufzeiten immer mit Hilfe von O -, Ω -, Θ -Notation angeben.

Illustration von $O(g(n))$



$$g(x) = x^2$$
$$f(x) = 100x$$



$$g(x) = x^2$$
$$f(x) = 200x$$

Regeln für Kalküle - Transitivität

O -, Ω - und Θ -Kalkül sind **transitiv**, d.h.:

- Aus $f(n) = O(g(n))$ und $g(n) = O(h(n))$
folgt $f(n) = O(h(n))$.
- Aus $f(n) = \Omega(g(n))$ und $g(n) = \Omega(h(n))$
folgt $f(n) = \Omega(h(n))$.
- Aus $f(n) = \Theta(g(n))$ und $g(n) = \Theta(h(n))$
folgt $f(n) = \Theta(h(n))$.

3. Inkrementelle Algorithmen

Definition 3.1

Bei einem inkrementellen Algorithmus wird sukzessive die Teillösung für die ersten i Objekte aus der bereits bekannten Teillösung für die ersten $i - 1$ Objekte berechnet, $i = 1, \dots, n$.

Beispiel Min-Search:

- Objekte sind Einträge des Eingabearrays.
- Teilproblem bestehend aus ersten i Objekten bedeutet Minimum im Teilarray $A[1 \dots i]$ bestimmen.

Sortieren und inkrementelle Algorithmen

Eingabe bei Sortieren:

Folge von n Zahlen (a_1, a_2, \dots, a_n) .

Ausgabe bei Sortieren:

Umordnung (b_1, b_2, \dots, b_n) der Eingabefolge, sodass $b_1 \leq b_2 \leq \dots \leq b_n$.

Sortieralgorithmus:

Verfahren, das zu jeder Folge (a_1, a_2, \dots, a_n) die sortierte Umordnung (b_1, b_2, \dots, b_n) berechnet.

Eingabe: (31,41,59,26,51,48)

Ausgabe: (26,31,41,48,51,59)

Insertion-Sort

Idee:

Sukzessive wird eine Sortierung der Teilarrays $A[1 \dots i]$,
 $1 \leq i \leq A.length$ berechnet.

Insertion-Sort(A)

```
1 for  $j = 2$  to  $A.length$ 
2    $key = A[j]$ 
3   // Füge  $A[j]$  in sortierte Folge  $A[1 \dots j - 1]$  ein
4    $i = j - 1$ 
5   while  $i > 0$  and  $A[i] > key$ 
6      $A[i + 1] = A[i]$ 
7      $i = i - 1$ 
8    $A[i + 1] = key$ 
```

Insertion-Sort

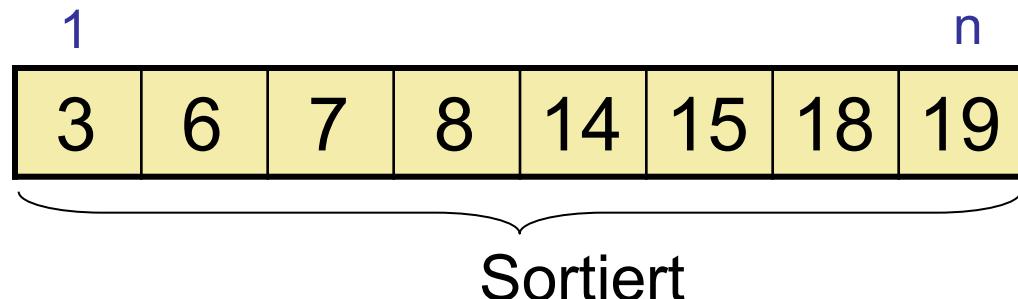
Insertion-Sort(A) // Eingabegröße n
// $A.length = n$

```
1 for  $j = 2$  to  $A.length$ 
2    $key = A[j]$ 
3    $i = j - 1$ 
4   while  $i > 0$  and  $A[i] > key$ 
5      $A[i + 1] = A[i]$ 
6      $i = i - 1$ 
7    $A[i + 1] = key$ 
```

1								n
8	15	3	14	7	6	18	19	

Insertion-Sort

```
Insertion-Sort(A)                                // Eingabegröße n
                                                    // A.length = n
1 for j = 2 to A.length
2   key = A[j]
3   i = j - 1                                // verschiebe alle
4   while i > 0 and A[i] > key          // A[i ... j - 1], die größer
5     A[i + 1] = A[i]                      // als key sind eine
6     i = i - 1                            // Stelle nach rechts
7   A[i + 1] = key                        // speichere key in
                                                // „Lücke“
```



Invariante bei Insertion-Sort

Invariante:

Vor Durchlauf der `for`-Schleife (Zeilen 1-8) für Index j gilt, dass $A[1 \dots j - 1]$ die $j - 1$ Eingabezahlen in sortierter Reihenfolge enthält.

Initialisierung:

$j = 2$ und $A[1]$ ist sortiert, da es nur eine Zahl enthält.

Erhaltung:

`while`-Schleife (Zeilen 5-7) zusammen mit Zeile 8 sortiert $A[j]$ korrekt ein.

Terminierung:

Vor Durchlauf mit $j = A.length + 1$ ist $A[1 \dots A.length]$ sortiert.

Invariante bei Insertion-Sort

Satz 3.2

Insertion-Sort sortiert eine Folge von n Zahlen aufsteigend.

Beweis:

- Wir zeigen, dass die Schleifeninvariante erfüllt ist
- Da die Schleife mit $j = n + 1$ terminiert, folgt aus der Invariante die Korrektheit des Algorithmus

Induktionsanfang ($j = 2$):

- $A[1]$ ist sortiert ✓

Induktionsannahme (I.A.):

- Invariante gilt für $j = N$

Invariante bei Insertion-Sort

Induktionsschritt ($N \rightarrow N + 1$):

- Betrachte Durchlauf mit $j = N$
- Insertionsort merkt sich $A[N]$ in Variable key
- Sei $1 \leq k \leq N - 1$ der kleinste Index mit $A[k] > key$ oder $k = N$, falls ein solcher nicht existiert
- Der Algorithmus verschiebt $A[k, \dots, N - 1]$ nach $A[k + 1, \dots, N]$
- Dann wird $A[k]$ auf den Wert key gesetzt
- Danach gilt:
 - (1) $A[1] \leq A[2] \leq \dots \leq A[k - 1]$ nach I.A.
 - (2) $A[k - 1] \leq A[k] \leq A[k + 1]$ nach Ablauf der Schleife
 - (3) $A[k + 1] \leq A[k + 2] \leq \dots \leq A[N]$ nach I.A.
- Aus (1)-(3) folgt die Behauptung für $N + 1$

Insertion-Sort – Analyse

Insertion-Sort(A)	cost	times
1 for $j = 2$ to $A.length$	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Einfügen von $A[j]$		
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

Hierbei ist t_j die Anzahl der Durchläufe der Abfrage in Zeile 5.

Invariante bei Insertion-Sort

Satz 3.3

Insertion-Sort besitzt Laufzeit $\Theta(n^2)$.

Beweis:

1. Es gibt ein c_2 , so dass die Laufzeit von Insertion-Sort bei allen Eingaben der Größe n immer höchstens $c_2 n^2$ ist.
2. Es gibt ein c_1 , so dass für alle n eine Eingabe I_n der Größe n existiert bei der Insertion-Sort mindestens Laufzeit $c_1 n^2$ besitzt.

4. Divide & Conquer – Merge-Sort

Definition 4.1

Divide & Conquer (Teile & Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Ein Divide & Conquer-Algorithmus löst ein Problem in 3 Schritten:

1. **Teile** ein Problem in mehrere Unterprobleme.
2. **Erobere** jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
3. **Kombiniere** die Lösungen der Teilprobleme zu einer Gesamtlösung.

Divide&Conquer und Sortieren

- Teile ein Problem in Unterprobleme.
 - Erobere jedes einzelne Teilproblem, durch rekursive Lösung.
 - Kombiniere die Lösungen zu einer Gesamtlösung.
-
- Teile eine n -elementige Teilfolge auf in zwei Teilfolgen mit jeweils etwa $\frac{n}{2}$ Elementen.
 - Sortiere die beiden Teilfolgen rekursiv.
 - Mische die sortierten Teilfolgen zu einer sortierten Gesamtfolge.

Merge-Sort

Merge-Sort ist eine mögliche Umsetzung des Divide&Conquer-Prinzips auf das Sortierproblem.

Merge-Sort(A, p, r)

```
1  if   $p < r$ 
2       $q = \left\lfloor \frac{p+r}{2} \right\rfloor$ 
3      Merge-Sort( $A, p, q$ )
4      Merge-Sort( $A, q + 1, r$ )
5      Merge( $A, p, q, r$ )
```

- Merge ist Algorithmus zum Mischen zweier sortierter Teilfolgen.
- Aufruf zu Beginn mit $\text{Merge-Sort}(A, 1, A.length)$

Illustration von Merge-Sort (1)

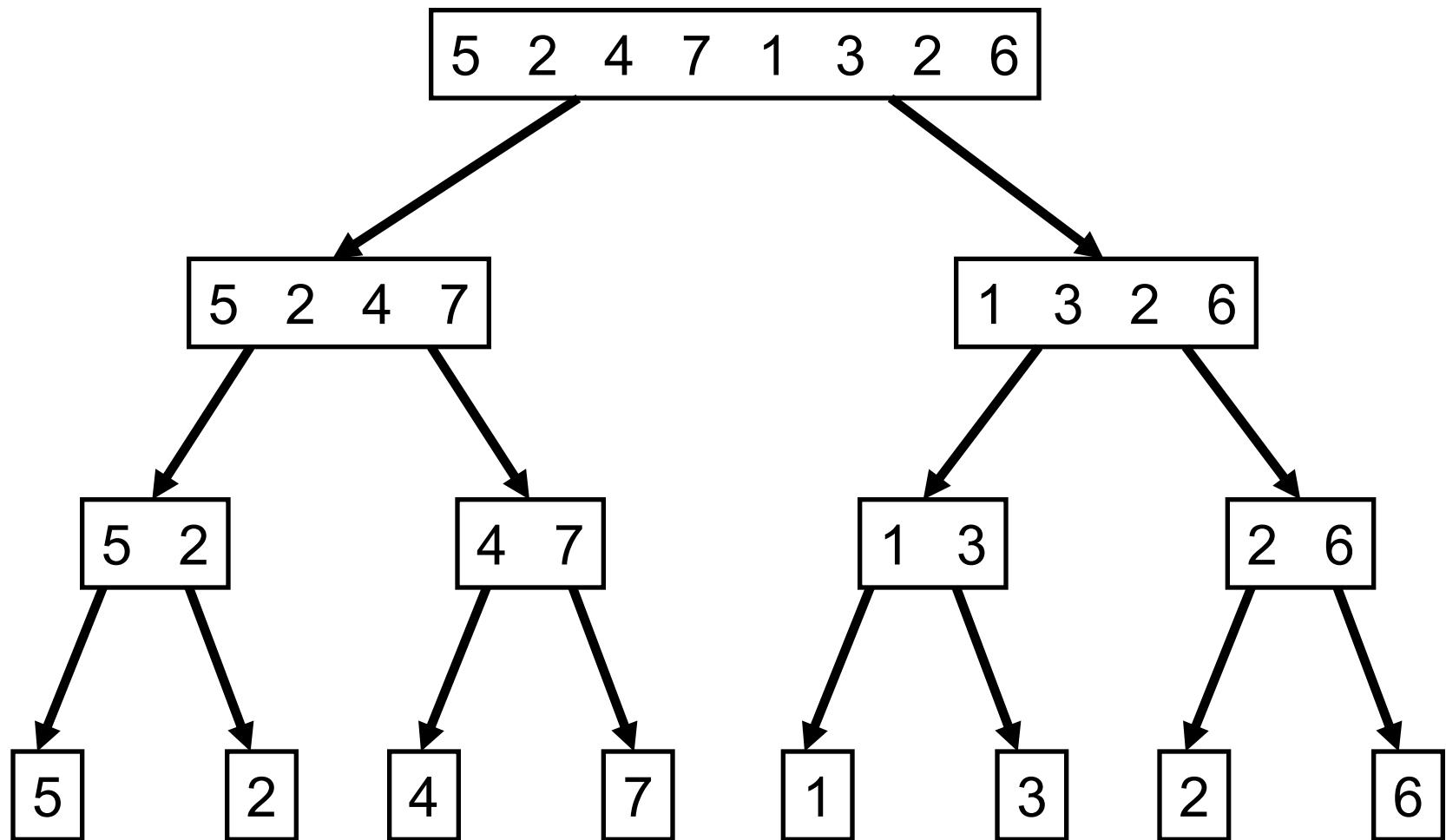


Illustration von Merge-Sort (2)

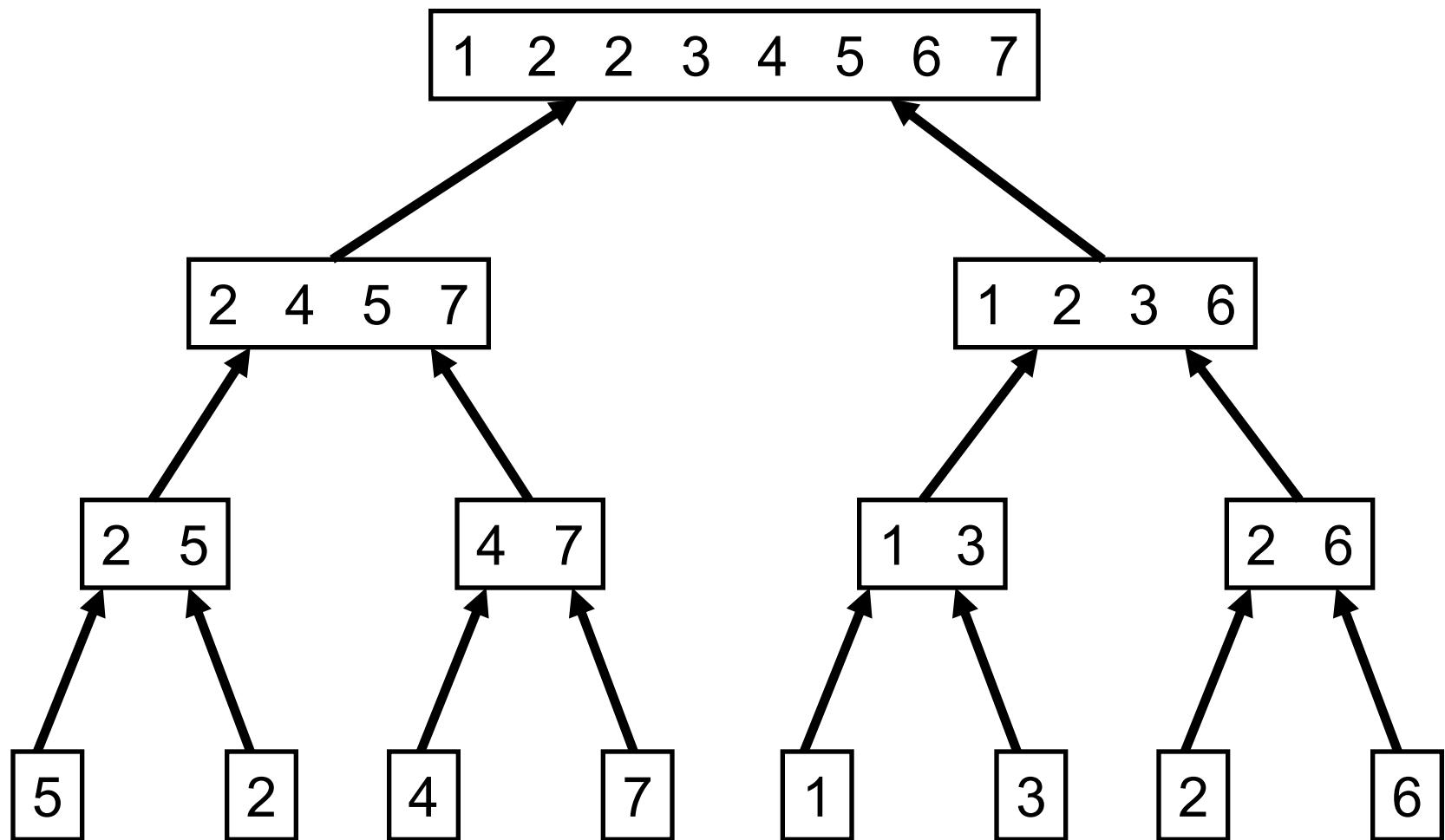


Illustration von Merge

Merge(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  //Erzeuge Arrays  $L[1..n_1 + 1], R[1..n_2 + 1]$ 
4  for  $i = 1$  to  $n_1$ 
5     $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7     $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] = L[i]$ 
15      $i = i + 1$ 
16   else  $A[k] = R[j]$ 
17      $j = j + 1$ 
```

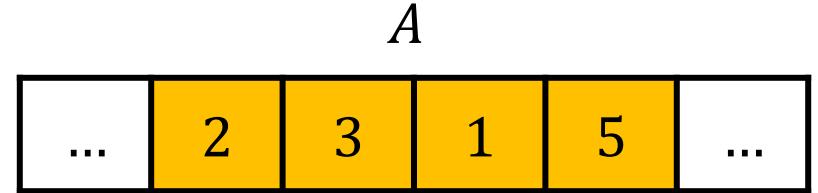
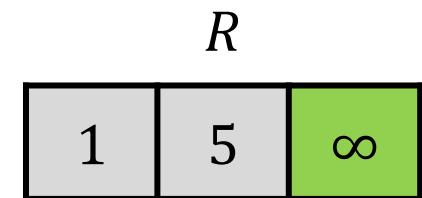
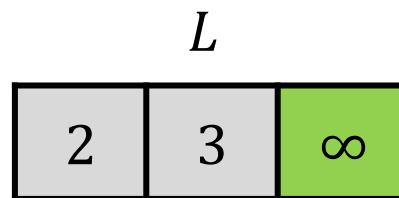
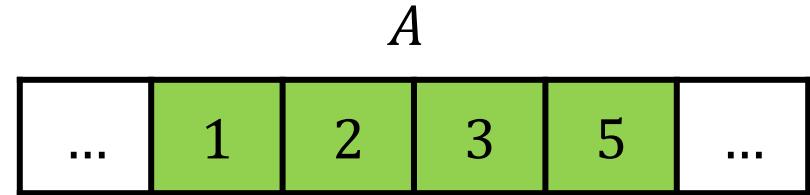


Illustration von Merge

Merge(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  //Erzeuge Arrays  $L[1..n_1 + 1], R[1..n_2 + 1]$ 
4  for  $i = 1$  to  $n_1$ 
5     $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7     $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] = L[i]$ 
15      $i = i + 1$ 
16   else  $A[k] = R[j]$ 
17      $j = j + 1$ 
```



Korrektheit von Merge-Sort

Zeigen Korrektheit von Merge-Sort mit **vollständiger Induktion** über Größe des Eingabearrays A . Zeigen hierzu:

1. Ist $A.length = 1$, dann ist Merge-Sort korrekt.
(Induktionsverankerung)
2. Ist Merge-Sort für alle Arrays A mit $A.length < n$ korrekt, dann ist Merge-Sort auch für Arrays A mit $A.length = n$ korrekt.
(Induktionsschluss)

zu 1:

Ist $A.length = 1$, so gilt in Merge $p = r$. Array $A[p]$ wird nicht verändert und Merge-Sort ist korrekt.

zu 2:

Folgt, wenn Merge aus sortierten Teilarrays $A[p, \dots, q], A[q + 1, \dots, r]$ ein sortiertes Array $A[p, \dots, r]$ berechnet.

Korrektheit von Merge - Invariante

Lemma 4.2

Erhält Algorithmus Merge als Eingabe ein Teilarray $A[p, \dots, r]$, so dass die beiden Teilarrays $A[p, \dots, q]$ und $A[q + 1, \dots, r]$ sortiert sind, so ist nach Durchlauf von Merge das Teilarray $A[p, \dots, r]$ ebenfalls sortiert.

Invariante:

Vor Durchlauf der Schleife in Zeilen 12-17 mit Index k enthält das Array $A[p \dots k - 1]$ die $k - p$ kleinsten Zahlen aus den Arrays L und R in sortierter Reihenfolge.

Außerdem sind $L[i]$ und $R[j]$ jeweils die kleinsten noch nicht einsortierten Elemente in den Arrays L bzw. R .

Korrektheit von Merge – 3 Schritte

Initialisierung:

Vor Durchlauf mit $k = p$ ist das Array $A[p \dots k - 1]$ leer.

Daher ist die Invariante erfüllt.

Erhaltung:

Es gelte $L[i] \leq R[j]$.

Dann ist $L[i]$ kleinstes noch nicht einsortiertes Element.

Damit enthält $A[p \dots k]$ die $k - p + 1$ kleinsten Elemente.

Zusammen mit Erhöhung der Zähler i, k garantiert dies die Erhaltung der Invariante. Analoge Argumentation im Fall $L[i] > R[j]$.

Terminierung:

Nach Ende der Schleife enthält $A[p \dots r]$ die $r - p + 1$ kleinsten Elemente sortiert. Also sind dann alle Elemente sortiert.

Laufzeit von Merge

Lemma 4.3

Ist die Eingabe von Merge ein Teilarray der Größe n ,
so ist die Laufzeit von Merge $\Theta(n)$.

Laufzeit von D&C-Algorithmen

$T(n)$:= Gesamtlaufzeit bei Eingabegröße n

a := Anzahl der Teilprobleme durch Teilung

$\frac{n}{b}$:= Größe der Teilprobleme

$D(n)$:= Zeit für Teilungsschritt

$C(n)$:= Zeit für Kombinationsschritt

Für $n \leq u$ wird Algorithmus mit Laufzeit $\leq c$ benutzt.

Dann gilt:

$$T(n) \leq \begin{cases} c & \text{falls } n \leq u \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{sonst} \end{cases}$$

Laufzeit von Merge-Sort (1)

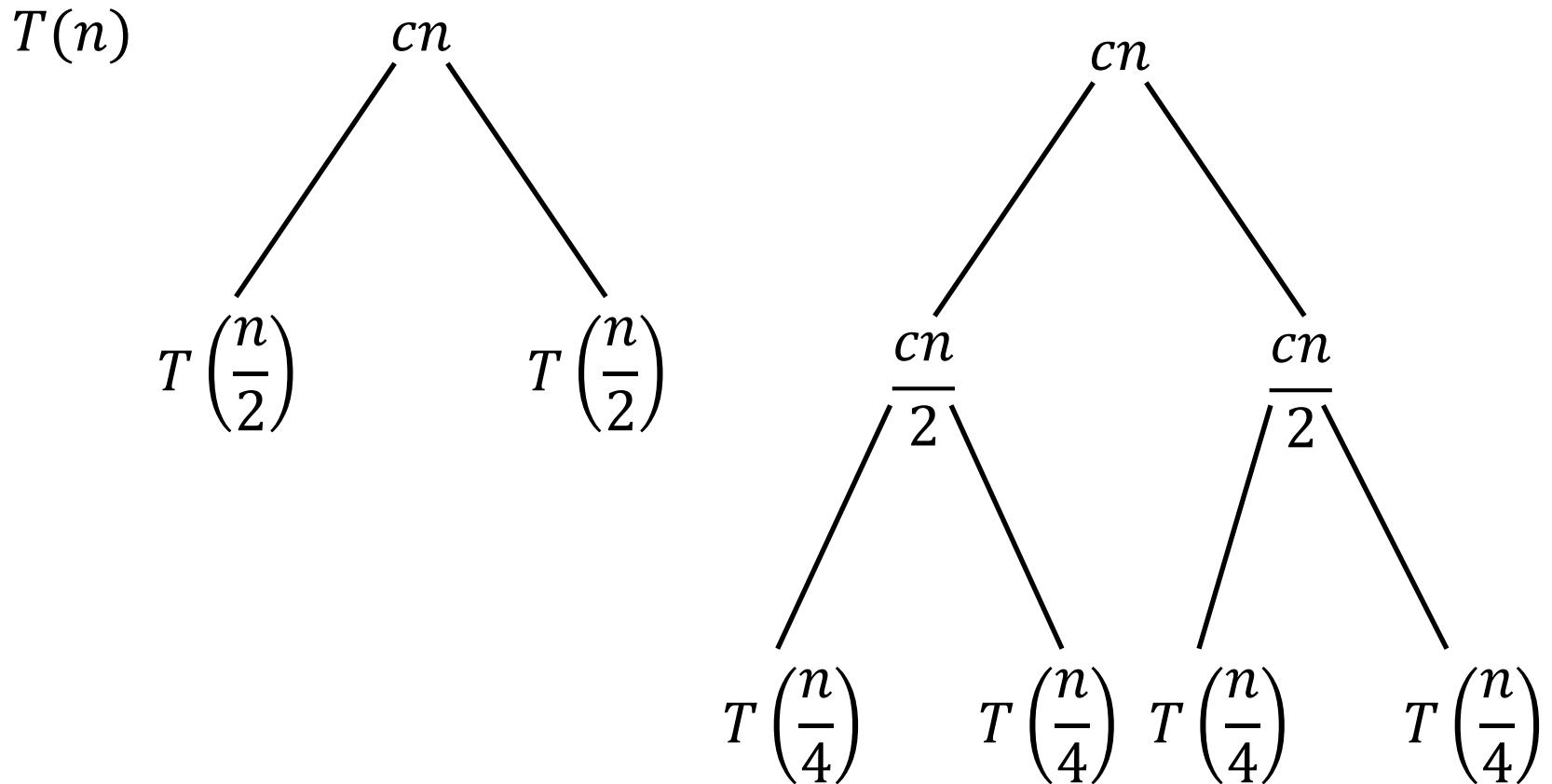
- $u = 1, a = 2, b \approx 2$
- $D(n) = \Theta(1), C(n) = \Theta(n)$ (Lemma 4.3)
- Sei c so gewählt, dass eine Zahl in Zeit c sortiert werden kann und $D(n) + C(n) \leq cn$ gilt.

Lemma 4.4

Für die Laufzeit $T(n)$ von Merge-Sort gilt:

$$T(n) \leq \begin{cases} c & \text{falls } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{sonst} \end{cases}$$

Laufzeit von Merge-Sort (2)



Laufzeit von Merge-Sort (3)

Satz 4.5

Merge-Sort besitzt Laufzeit $\Theta(n \log(n))$.

Zum Beweis wird gezeigt:

1. Es gibt ein c_2 , so dass die Laufzeit von Merge-Sort bei allen Eingaben der Größe n immer höchstens $c_2 n \log(n)$ ist.
2. Es gibt ein c_1 , so dass für alle n eine Eingabe I_n der Größe n existiert bei der Merge-Sort mindestens Laufzeit $c_1 n \log(n)$ besitzt.

Laufzeit von Merge-Sort (5)

Eingabegröße n

Laufzeit	10	100	1.000	10.000	100.000
n^2	100	10.000	1.000.000	100.000.000	10.000.000.000
$n \log n$	33	664	9.965	132.877	1.660.964

Beobachtung:

- n^2 wächst viel stärker als $n \log n$
- Selbst bei großen Konstanten wird Merge-Sort schnell besser
- Konstanten spielen kaum eine Rolle
→ Θ-Notation ist entscheidend für große n

Elementare Wahrscheinlichkeitstheorie

Average-Case Laufzeit:

- Betrachten alle Permutationen der n Eingabezahlen.
- Berechnen für jede Permutation Laufzeit des Algorithmus bei dieser Permutation.
- Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.

Definition 4.6

Eine Permutation ist eine bijektive Abbildung einer endlichen Menge auf sich selbst.

Alternativ: Eine Permutation ist eine Anordnung der Elemente einer endlichen Menge in einer geordneten Folge.

Elementare Wahrscheinlichkeitstheorie

Lemma 4.7

Zu einer n -elementigen Menge gibt es genau
 $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1) \cdot n$ Permutationen.

Beweis: Induktion über n .

(I.A.) $n = 1$ klar.

(I.V.) Der Satz gilt für n .

(I.S.) $n + 1$:

An letzter Stelle steht die i -te Zahl. Es gibt $n!$ unterschiedliche Anordnungen der restlichen n Zahlen.

Da i jeden Wert zwischen 1 und $n + 1$ annehmen kann, gibt es $(n + 1) n! = (n + 1)!$ Anordnungen der $n + 1$ Zahlen.

Beispiel: Menge $\{2,3,6\}$

Permutationen: $(2,3,6), (2,6,3), (3,2,6), (3,6,2), (6,2,3), (6,3,2)$

Elementare Wahrscheinlichkeitstheorie

Definition (Wahrscheinlichkeitsraum)

Ein Wahrscheinlichkeitsraum S ist eine Menge von Elementarereignissen.

Ein Elementarereignis kann als der Ausgang eines (Zufalls)Experiments betrachtet werden.

Beispiel:

- Münzwurf mit zwei unterscheidbaren Münzen
- Ergebnis eines Münzwurfs können wir als Zeichenkette der Länge 2 über $\{K, Z\}$ (Kopf, Zahl) darstellen
- Wahrscheinlichkeitsraum ist $S = \{KK, KZ, ZK, ZZ\}$
- Elementarereignisse sind also die möglichen Ausgänge des Münzwurfs

Elementare Wahrscheinlichkeitstheorie

Definition (Ereignis) —

Ein Ereignis ist eine Untermenge eines Wahrscheinlichkeitsraums.

(Diese Definition ist etwas vereinfacht, aber für unsere Zwecke ausreichend)

Beispiel:

$\{KK, KZ, ZK\}$ ist das Ereignis, dass bei unserem Münzwurf mindestens eine Münze Kopf zeigt.

Elementare Wahrscheinlichkeitstheorie

Definition (Wahrscheinlichkeitsverteilung)

Eine Wahrscheinlichkeitsverteilung $\Pr[]$ auf einem Wahrscheinlichkeitsraum S ist eine Abbildung der Ereignisse von S in die reellen Zahlen, die folgende Axiome erfüllt:

1. $\Pr[A] \geq 0$ für jedes Ereignis A
2. $\Pr[S] = 1$
3. $\Pr[A \cup B] = \Pr[A] + \Pr[B]$ für alle Ereignisse A, B mit $A \cap B = \emptyset$

$\Pr[A]$ bezeichnet die Wahrscheinlichkeit von Ereignis A .

Elementare Wahrscheinlichkeitstheorie

Beispiel:

- Bei einem fairen Münzwurf haben wir $\Pr[A] = \frac{1}{4}$ für jedes Elementarereignis $A \in [KK, KZ, ZK, ZZ]$
- Die Wahrscheinlichkeit für das Ereignis $[KK, KZ, ZK]$ („mindestens eine Münze zeigt Kopf“) ist
$$\Pr[\{KK, KZ, ZK\}] = \Pr[KK] + \Pr[KZ] + \Pr[ZK] = \frac{3}{4}$$

Bemerkung:

Eine Verteilung, bei der jedes Elementarereignis aus S dieselbe Wahrscheinlichkeit hat, nennen wir auch Gleichverteilung über S .

Elementare Wahrscheinlichkeitstheorie

Abhängigkeiten:

Was passiert, wenn man schon etwas über den Ausgang eines Zufallsexperiments weiß?

Frage:

Jemand hat beobachtet, dass der Ausgang des Münzwurfs mit zwei Münzen mindestens einmal Kopf zeigt.

Wie groß ist die Wahrscheinlichkeit für zweimal Kopf?

Elementare Wahrscheinlichkeitstheorie

Definition (bedingte Wahrscheinlichkeit)

Die bedingte Wahrscheinlichkeit eines Ereignisses A unter der Voraussetzung, dass Ereignis B auftritt ist

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

wenn $\Pr[B] \neq 0$ ist.

Elementare Wahrscheinlichkeitstheorie

Beispiel:

- Jemand beobachtet den Ausgang unseres Münzwurfexperiments und sagt uns, dass es mindestens einmal Zahl gibt.
- Was ist die Wahrscheinlichkeit für zweimal Zahl (Ereignis A) unter dieser Beobachtung (Ereignis B)?

$$\Pr[A|B] = \frac{\frac{1}{4}}{\frac{3}{4}} = \frac{1}{3}$$

Elementare Wahrscheinlichkeitstheorie

Definition (Zufallsvariable)

Eine Zufallsvariable X ist eine Funktion von einem Wahrscheinlichkeitsraum in die reellen Zahlen.

Bemerkung:

Eine Zufallsvariable liefert uns zu jedem Ausgang eines Zufallsexperiments einen Wert.

Elementare Wahrscheinlichkeitstheorie

Beispiel:

- Wurf zweier Münzen
- Sei X Zufallsvariable für die Anzahl Münzen, die Zahl zeigen

$$X(KK) = 0,$$

$$X(KZ) = 1,$$

$$X(ZK) = 1,$$

$$X(ZZ) = 2$$

Elementare Wahrscheinlichkeitstheorie

Für Zufallsvariable X und reelle Zahl x können wir das Ereignis $X = x$ definieren als $\{s \in S : X(s) = x\}$. Damit gilt:

$$\Pr[X = x] = \Pr[\{s \in S : X(s) = x\}]$$

Beispiel:

Was ist die Wahrscheinlichkeit, dass wir bei zwei Münzwürfen genau einmal Kopf erhalten?

$$\Pr[X = 1] = \Pr[\{KZ \cup ZK\}] = \Pr[KZ] + \Pr[ZK] = \frac{1}{2}$$

(bei der selben Definition von X wie auf der letzten Folie)

Elementare Wahrscheinlichkeitstheorie

Definition (Erwartungswert)

Der Erwartungswert einer Zufallsvariable ist definiert als

$$E[X] = \sum x \cdot \Pr[X = x]$$

Interpretation:

Der Erwartungswert gibt den „durchschnittlichen“ Wert der Zufallsvariable an, wobei die Wahrscheinlichkeiten der Ereignisse berücksichtigt werden.

Elementare Wahrscheinlichkeitstheorie

Beispiel:

Erwartete Anzahl „Kopf“ bei 2 Münzwürfen

$$\begin{aligned}E[X] &= 0 \cdot \Pr[X = 0] + 1 \cdot \Pr[X = 1] + 2 \cdot \Pr[X = 2] \\&= 0 + \frac{1}{2} + 2 \cdot \frac{1}{4} \\&= 1\end{aligned}$$

Elementare Wahrscheinlichkeitstheorie

Linearität des Erwartungswerts:

$$E[X + Y] = E[X] + E[Y]$$

Bemerkung:

Eine der wichtigsten Formeln im Bereich randomisierte Algorithmen.

Anwendung:

Man kann komplizierte Zufallsvariablen als Summe einfacher Zufallsvariablen schreiben und dann den Erwartungswert der einfachen Zufallsvariablen bestimmen.

Elementare Wahrscheinlichkeitstheorie

Average-Case Laufzeit:

- Betrachten alle Permutation der n Eingabezahlen.
- Berechnen für jede Permutation Laufzeit des Algorithmus bei dieser Permutation.
- Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Average-case Laufzeit ist die erwartete Laufzeit einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der n Eingabezahlen.

Elementare Wahrscheinlichkeitstheorie

Satz 4.8

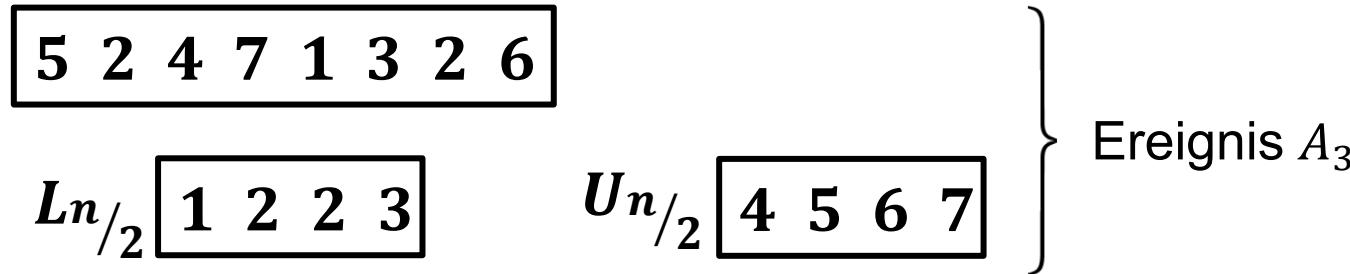
Insertion-Sort besitzt Average-case Laufzeit $\Theta(n^2)$.

Beweis:

Wir zeigen: mit Wahrscheinlichkeit $1/2$ hat man mindestens $n^2/16$ Vergleiche
(Annahme: n ist gerade).

- Sei $L_{n/2}$ die Menge der $n/2$ kleinsten Zahlen.
- Sei $U_{n/2}$ die Menge der $n/2$ größten Zahlen.
- Sei A_i das Ereignis, dass in einer zufälligen Permutation der n Zahlen genau i Elemente aus $U_{n/2}$ in der ersten Hälfte der Permutation platziert werden.
- Sei B_i das Ereignis, dass in einer zufälligen Permutation der n Zahlen genau i Elemente aus $L_{n/2}$ in der ersten Hälfte der Permutation platziert werden.

Elementare Wahrscheinlichkeitstheorie



- $\Pr\left[\bigcup_{n/4 \leq i \leq n/2} A_i\right] = \sum_{n/4 \leq i \leq n/2} \Pr[A_i]$
- $\Pr[A_i] = \Pr[B_i] \rightarrow \Pr[A_i] = \Pr[A_{n/2-i}]$
- $\sum_{0 \leq i \leq n/2} \Pr[A_i] = 1$
- $\sum_{n/4 \leq i \leq n/2} \Pr[A_i] = 1 - \sum_{0 \leq i < n/4} \Pr[A_i] = \Pr[A_{n/4}] + 1 - \sum_{n/4 \leq i \leq n/2} \Pr[A_i]$
- $2 \sum_{n/4 \leq i \leq n/2} \Pr[A_i] = 1 + \Pr[A_{n/4}] > 1 \rightarrow \sum_{n/4 \leq i \leq n/2} \Pr[A_i] > 1/2$
- Mit Wahrscheinlichkeit $1/2$ befindet sich mindestens die Hälfte von $U_{n/2}$ in der ersten Hälfte und mindestens die Hälfte von $L_{n/2}$ in der zweiten Hälfte einer zufälligen Permutation.

Elementare Wahrscheinlichkeitstheorie

Insertion-Sort(A)

```
1 for  $j = 2$  to  $A.length$ 
2    $key = A[j]$ 
3    $i = j - 1$ 
4   while  $i > 0$  and  $A[i] > key$ 
5      $A[i + 1] = A[i]$ 
6      $i = i - 1$ 
7    $A[i + 1] = key$ 
```

Mit Wahrscheinlichkeit $1/2$ gibt es mindestens $n/4$ Elemente $A[j]$ aus $L_{n/2}$ mit $j \geq n/2$.

Sei $j \geq n/2$ und $A[j]$ in $L_{n/2}$. Dann wird die **while**-Schleife mindestens $n/4$ mal durchlaufen.

$n^2/16$ Vergleiche

Im Durchschnitt produziert Insertion-Sort mehr als $n^2/32$ Vergleiche.

Quicksort

- Quicksort ist wie Merge-Sort ein auf dem **Divide&Conquer-Prinzip** beruhender Sortieralgorithmus.
- Von Quicksort existieren unterschiedliche Varianten, von denen einige in der Praxis besonders effizient sind.
- Die **worst-case Laufzeit** von Quicksort ist $\Theta(n^2)$.
- Die **durchschnittliche Laufzeit** ist jedoch $\Theta(n \log(n))$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta(n \log(n))$.

Quicksort

Eingabe:

Ein zu sortierendes Teilarray $A[p \dots r]$.

Teilungsschritt:

Berechne einen Index $q, p \leq q \leq r$ und vertausche die Reihenfolge der Elemente in $A[p \dots r]$, so dass die Elemente in $A[p \dots q - 1]$ nicht größer und die Elemente in $A[q + 1 \dots r]$ nicht kleiner sind als $A[q]$.

Eroberungsschritt:

Sortiere rekursiv die beiden Teilarrays $A[p \dots q - 1]$ und $A[q + 1 \dots r]$.

Kombinationsschritt:

Entfällt, da nach Eroberungsschritt das Array $A[p \dots r]$ bereits sortiert ist.

Quicksort

Quicksort(A, p, r)

```
1 if  $p < r$ 
2    $q = \text{Partition}(A, p, r)$ 
3   Quicksort( $A, p, q - 1$ )
4   Quicksort( $A, q + 1, r$ )
```

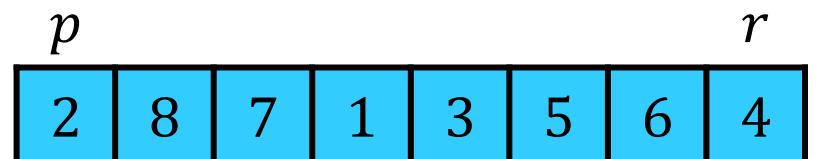
Aufruf, um Array A zu sortieren:

Quicksort($A, 1, A.length$)

Quicksort

Partition(A, p, r)

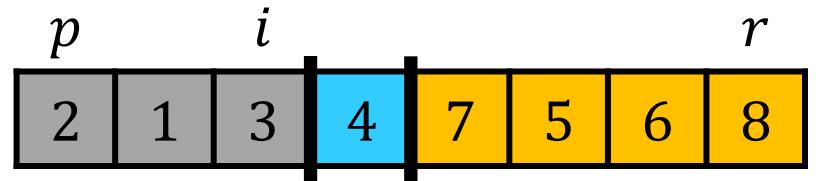
```
1  $x = A[r]$ 
2  $i = p - 1$ 
3 for  $j = p$  to  $r - 1$ 
4   if  $A[j] \leq x$ 
5      $i = i + 1$ 
6     exchange  $A[i]$  with  $A[j]$ 
7 exchange  $A[i + 1]$  with  $A[r]$ 
8 return  $i + 1$ 
```



Quicksort

Partition(A, p, r)

```
1  $x = A[r]$ 
2  $i = p - 1$ 
3 for  $j = p$  to  $r - 1$ 
4   if  $A[j] \leq x$ 
5      $i = i + 1$ 
6     exchange  $A[i]$  with  $A[j]$ 
7 exchange  $A[i + 1]$  with  $A[r]$ 
8 return  $i + 1$ 
```



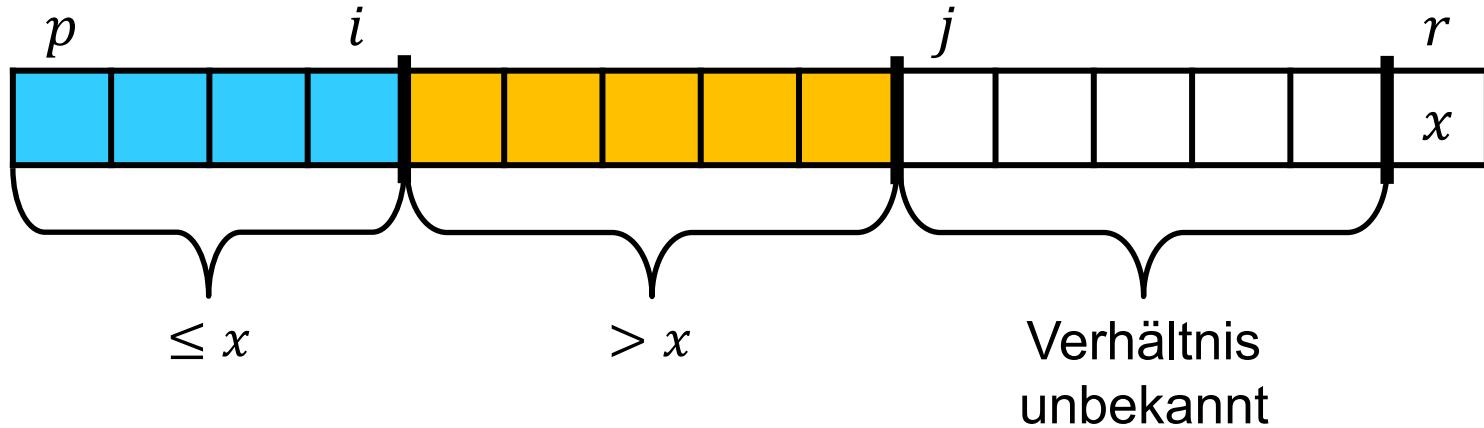
Quicksort

Invariante:

Vor Durchlauf der Schleife in Zeilen 3 – 6 mit Index j gilt für jeden Index k :

1. Falls $p \leq k \leq i$, dann ist $A[k] \leq x$.
2. Falls $i + 1 \leq k \leq j - 1$, dann ist $A[k] > x$.
3. Falls $k = r$, dann ist $A[k] = x$.

Quicksort



Quicksort

Initialisierung:

Vor dem ersten Schleifendurchlauf gilt $i = p - 1$ und $j = p$.

Daher gibt es in diesem Fall keine Indizes zwischen p und i (*1. Bedingung*) bzw. zwischen $i + 1$ und $j - 1$ (*2. Bedingung*).

Die erste Zeile sorgt dafür, dass die dritte Bedingung ebenfalls erfüllt ist.

Erhaltung:

Unterscheiden zwei Fälle:

1. $A[j] > x$
2. $A[j] \leq x$

Quicksort

1. Fall ($A[j] > x$):

Nur j wird erhöht.

Damit ist dann die zweite Bedingung auch für $k = j$ erfüllt.

(siehe Folie 10)

2. Fall ($A[j] \leq x$):

Element $A[j]$ kommt an Position i .

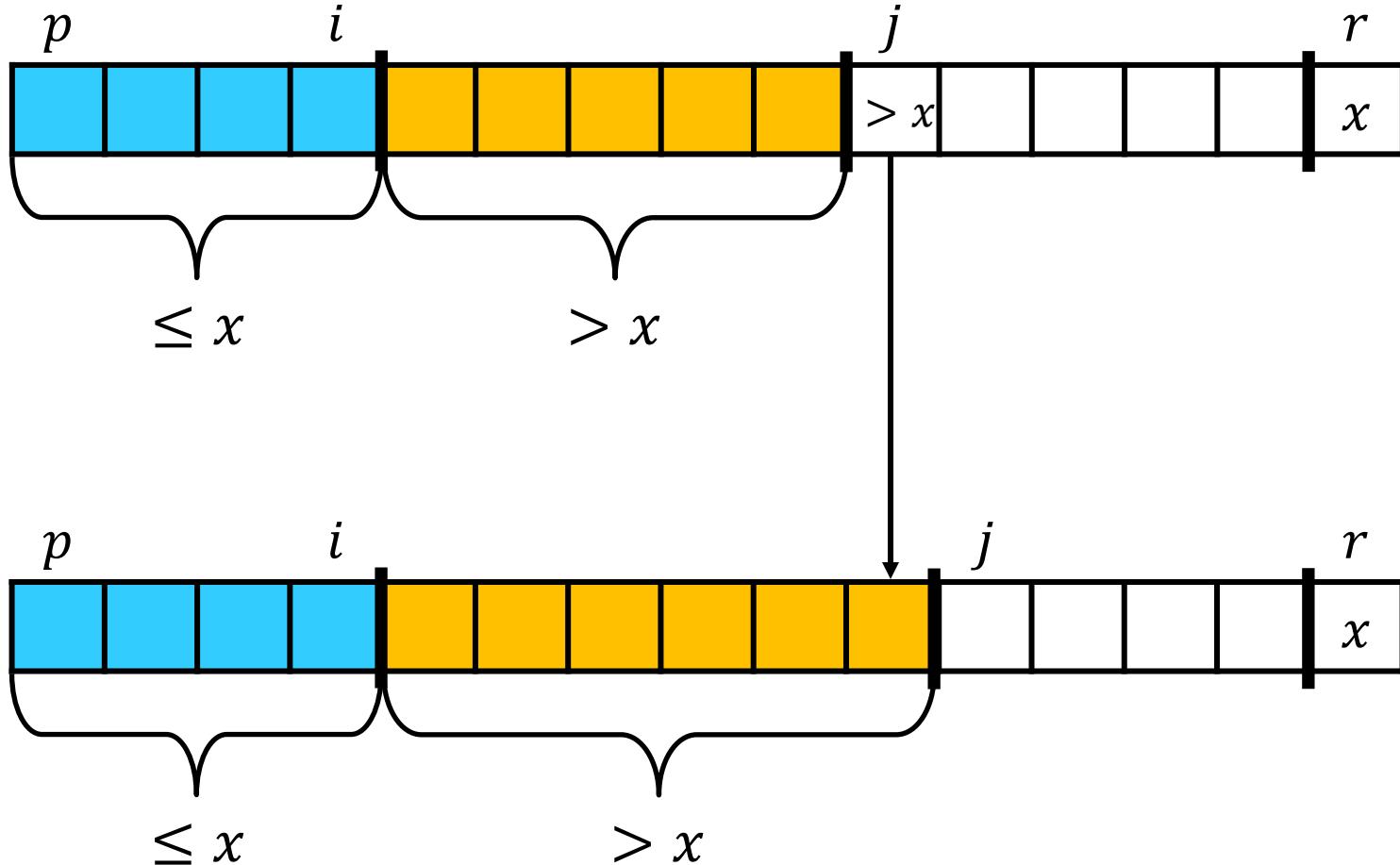
Da $A[j] \leq x$ ist die erste Bedingung weiter erfüllt.

Für neues $A[j - 1]$ gilt nach Voraussetzung $A[j - 1] > x$.

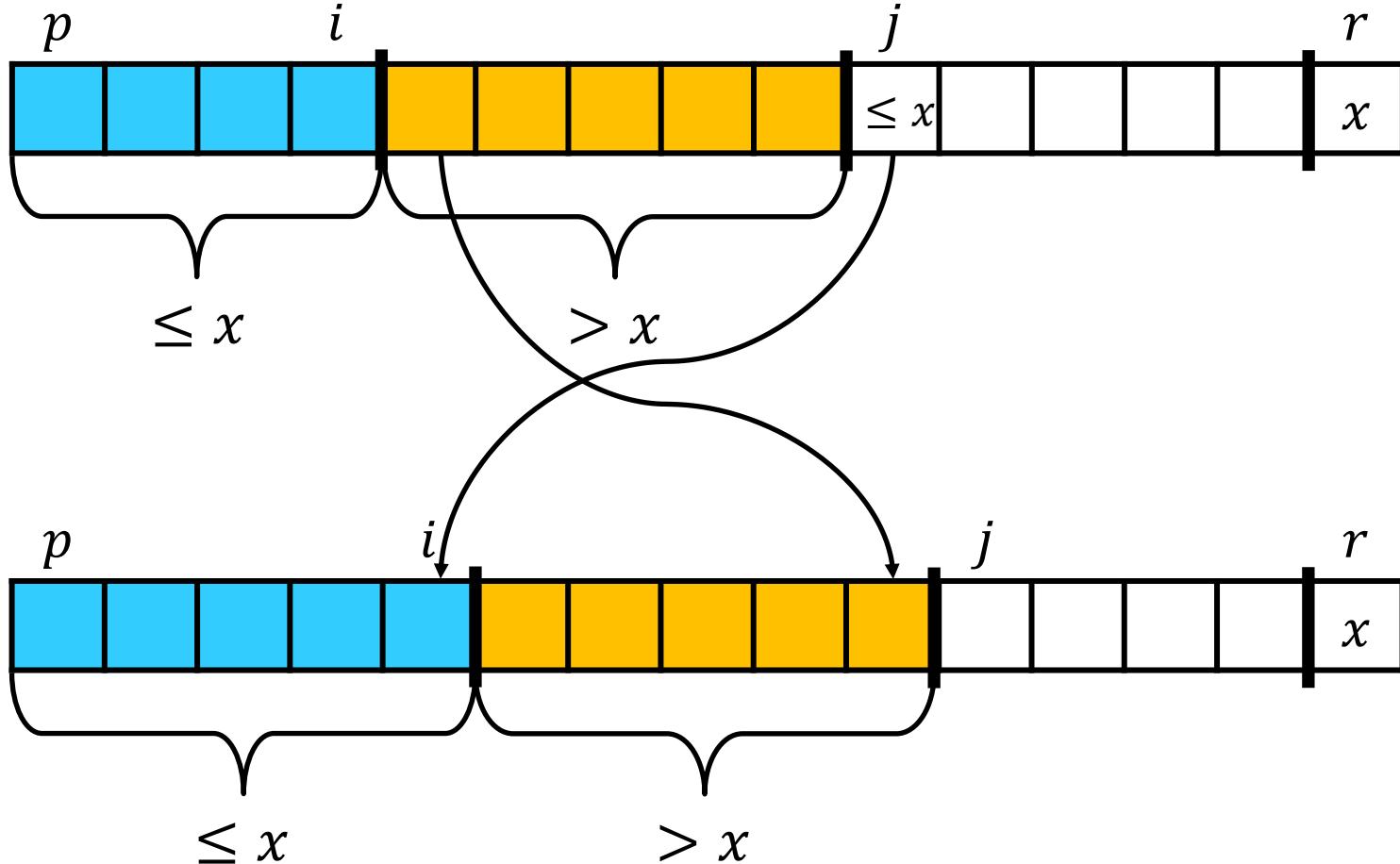
Damit ist die zweite Bedingung auch erfüllt.

(siehe Folie 11)

Quicksort



Quicksort



Quicksort

Terminierung:

Es gilt $j = r$ und alle Elemente des Arrays wurden mit x verglichen.

Zeile 9 stellt nun sicher, dass x zwischen die Elemente $< x$ und die Elemente $> x$ platziert wird.

Damit genügt die von Partition berechnete Aufteilung immer den Anforderungen von Quicksort.

Quicksort

Partition(A, p, r)

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4    if  $A[j] \leq x$ 
5       $i = i + 1$ 
6      exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

- Pro Zeile konstante Zeit.
- Schleife in Zeilen 3-6 wird $n = r - p$ -mal durchlaufen.

Satz 4.9

Partition hat Laufzeit $\Theta(n)$ bei Eingabe eines Teilarrays mit n Elementen.

Quicksort

Satz 4.10

Es gibt ein $c > 0$, so dass für alle n und alle Eingaben der Größe n Quicksort mindestens Laufzeit $c n \log(n)$ besitzt.

Satz 4.11

Quicksort besitzt worst-case Laufzeit $\Theta(n^2)$.

Satz 4.12

Quicksort besitzt average-case Laufzeit $\Theta(n \log(n))$.

Quicksort

Average-case Laufzeit:

- Betrachten alle Permutation der n Eingabezahlen.
- Berechnen für jede Permutation Laufzeit von Quicksort bei dieser Permutation.
- Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.

Quicksort

- Schlechte Eingaben für Quicksort können vermieden werden durch **Randomisierung**, d. h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta(n^2)$.
- Es gibt **keine schlechten Eingaben**. Dies sind Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta(n^2)$ besitzt.
- Laufzeit ist in diesem Modell erwartete Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Erwartete Laufzeit ist $\Theta(n \log(n))$.

Quicksort

Randomized-Partition(A, p, r)

- 1 $i = \text{Random}(p, r)$
- 2 exchange $A[r]$ with $A[i]$
- 3 **return** Partition(A, p, r)

Hierbei ist Random eine Funktion, die zufällig einen Wert aus $[p \dots r]$ wählt. Dabei gilt für alle $i \in [p \dots r]$:

$$\Pr(\text{Random}(p, r) = i) = \frac{1}{r-p+1}$$

Quicksort

Randomized-Quicksort(A, p, r)

- 1 **if** $p < r$
- 2 $q = \text{Randomized-Partition}(A, p, r)$
- 3 Randomized-Quicksort($A, p, q - 1$)
- 4 Randomized-Quicksort($A, q + 1, r$)

Satz 4.13

Die erwartete Laufzeit von Randomized-Quicksort ist $\Theta(n \log(n))$.
Dabei ist der Erwartungswert über die Zufallsexperimente in Randomized-Partition genommen.

Quicksort

- Verbesserung der Güte von Aufteilungen, indem nicht ein festes Element zur Aufteilung benutzt wird, sondern z. B. das mittlere von drei Elementen zur Aufteilung benutzt wird.
- Können etwa drei zufällige Elemente wählen oder $A[p], A[q], A[r]$ mit $q := \left\lfloor \frac{p+r}{2} \right\rfloor$.
- Beide Varianten in der Praxis erfolgreich.
- Aber nur zufällige Variante kann gut analysiert werden:
 - $\Theta(n \log(n))$ erwartete Laufzeit.

Quicksort

Median(A, i, j, k)

```
1 if ( $A[i] \leq A[j]$ ) and ( $A[k] \leq A[i]$ )
2   return  $i$ 
3 else if ( $A[i] \leq A[j]$ ) and ( $A[k] \geq A[j]$ )
4   return  $j$ 
5 else
6   return  $k$ 
```

Quicksort

Median-Partition(A, p, r)

- 1 $i = \text{Median}\left(p, \left\lfloor \frac{p+r}{2} \right\rfloor, r\right)$
- 2 exchange $A[r]$ with $A[i]$
- 3 **return** Partition(A, p, r)

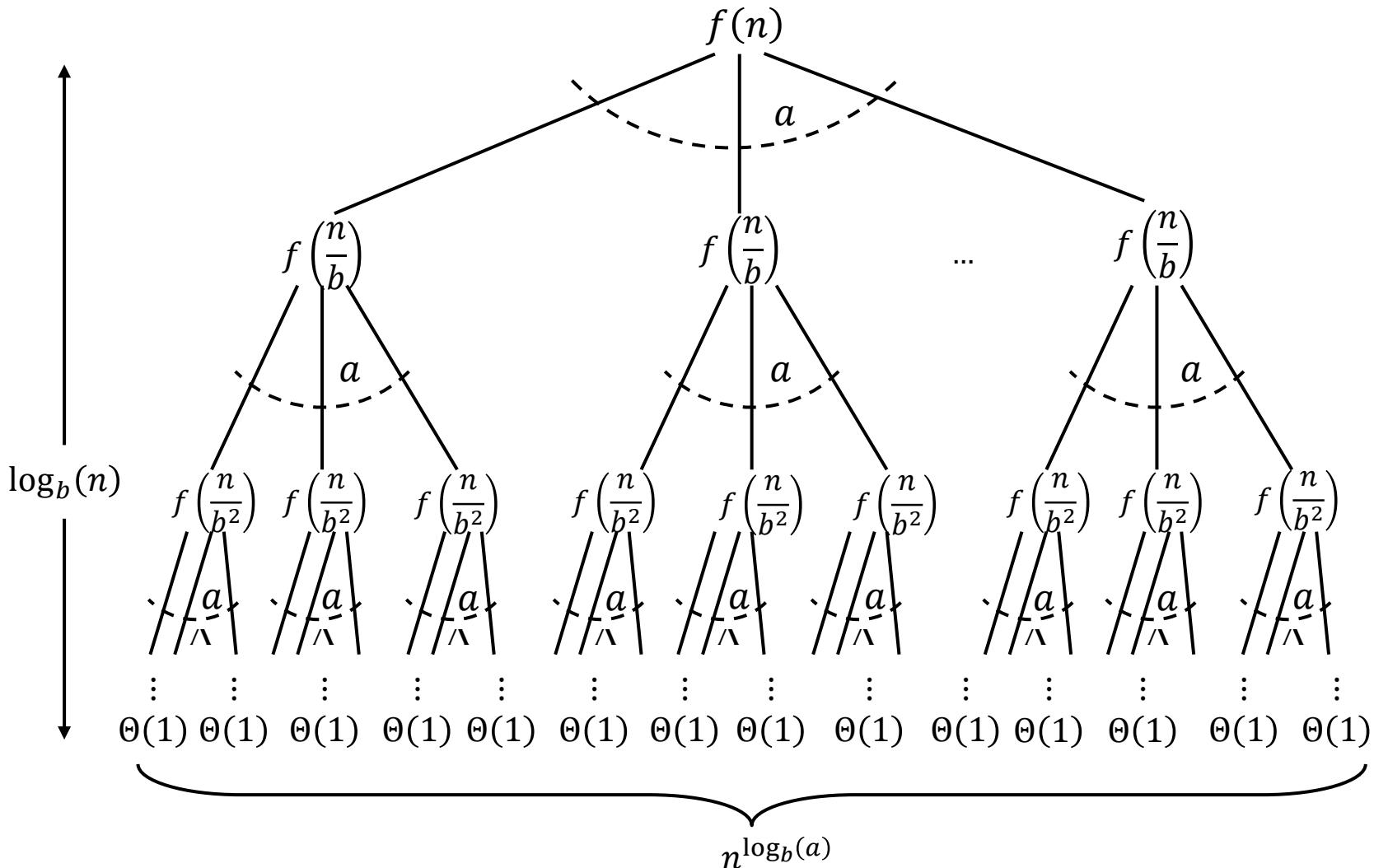
Median-Quicksort(A, p, r)

- 1 **if** $p < r$
- 2 $q = \text{Median-Partition}(A, p, r)$
- 3 Median-Quicksort($A, p, q - 1$)
- 4 Median-Quicksort($A, q + 1, r$)

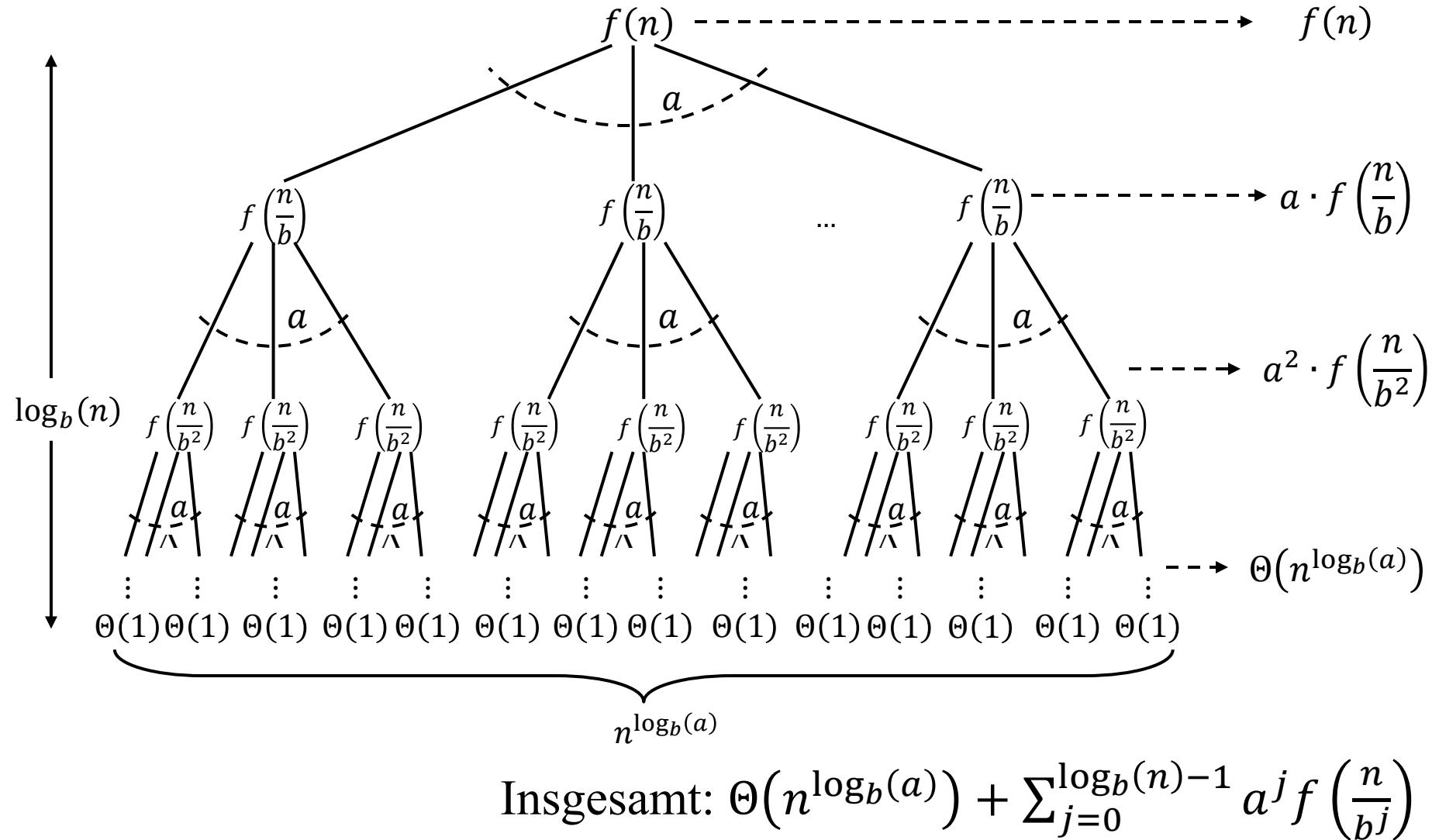
5. Rekursionen

- Laufzeiten, insbesondere von **Divide&Conquer** Algorithmen, werden häufig durch Rekursionsgleichungen beschrieben.
- werden eine Methode kennenlernen, um solche Gleichungen zu lösen: **Rekursionsbaum-Methode**
- Diese Methode kann verfeinert werden, um das Master-Theorem für Rekursionsgleichungen zu beweisen.
- Formulieren das Master-Theorem nur.
- Anwendung des Master-Theorems mit großer Vorsicht!

Rekursionsbaum für $T(n) = aT\left(\frac{n}{b}\right) + f(n)$



Rekursionsbaum und Summation



Das Master-Theorem

Satz 5.1 (Master-Theorem für Rekursionsgleichungen)

Seien $a, b > 1$ Konstanten, sei $f(n)$ eine Funktion und sei $T(n)$ definiert durch die Rekursionsgleichung $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

Hierbei kann $\frac{n}{b}$ auch durch $\left\lfloor \frac{n}{b} \right\rfloor$ oder $\left\lceil \frac{n}{b} \right\rceil$ ersetzt werden.

Dann kann $T(n)$ folgendermaßen abgeschätzt werden:

1. Ist $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right)$ für $\varepsilon > 0$, dann gilt $T(n) = O\left(n^{\log_b(a)}\right)$.
2. Ist $f(n) = \Theta\left(n^{\log_b(a)}\right)$, dann gilt $T(n) = O\left(n^{\log_b(a)} \cdot \log(n)\right)$.
3. Ist $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ für $\varepsilon > 0$, und ist $af\left(\frac{n}{b}\right) \leq cf(n)$ für eine Konstante $c < 1$ und $n \rightarrow \infty$, dann gilt $T(n) = \Theta(f(n))$.

6. Heapsort

- Werden sehen, wie wir durch geschicktes Organisieren von Daten effiziente Algorithmen entwerfen können.
- Genauer werden wir immer wieder benötigte Operationen durch **Datenstrukturen** unterstützen.
- Werden im Laufe des Semesters viel mehr über Datenstrukturen und ihren Zusammenhang mit effizienten Algorithmen lernen.
- Beispiel Sortieren: Warum nicht Array sortieren, indem wir immer wieder Minimum berechnen?

Heaps

Definition 6.1

Ein Heap über ein Array A ist das Array A zusammen mit einem Parameter $A.\text{heap-size}$ und drei Funktionen

$$\text{PARENT}, \text{LEFT}, \text{RIGHT}: \{1, \dots, A.\text{heap-size}\} \rightarrow \{1, \dots, A.\text{length}\}.$$

Dabei gilt:

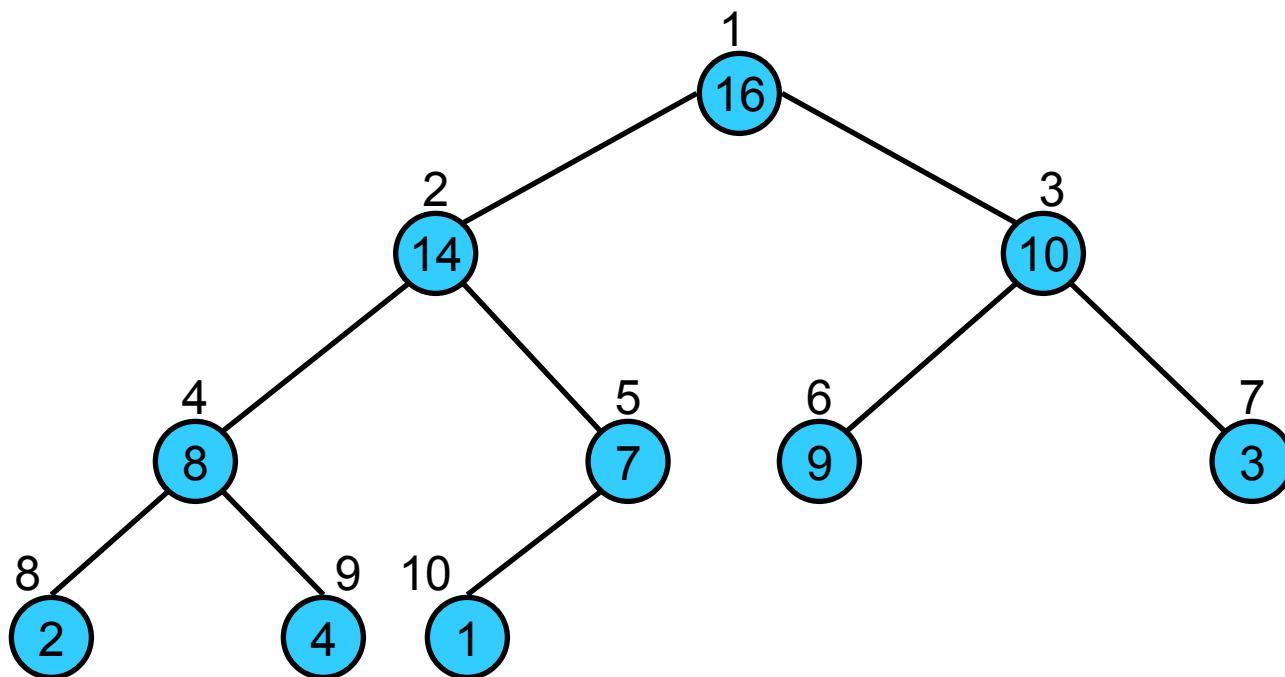
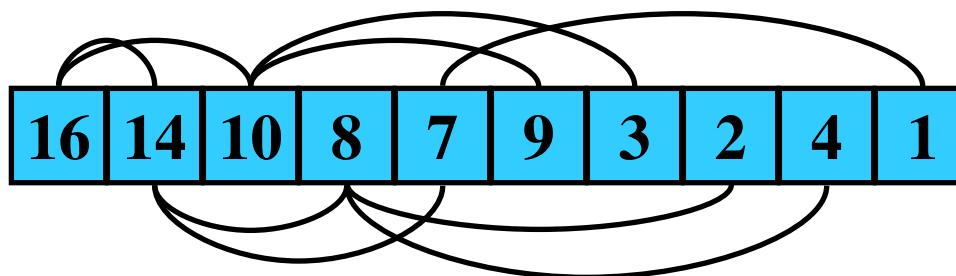
1. $1 \leq A.\text{heap-size} \leq A.\text{length}$
2. $\text{PARENT}(i) = \left\lfloor \frac{i}{2} \right\rfloor$ für alle $1 \leq i \leq A.\text{heap-size}$
3. $\text{LEFT}(i) = 2i$ für alle $1 \leq i \leq A.\text{heap-size}$
4. $\text{RIGHT}(i) = 2i + 1$ für alle $1 \leq i \leq A.\text{heap-size}$

Arrayelemente $A[1], \dots, A[A.\text{heap-size}]$ heißen Heapelemente.

Heaps (2)

- Die Funktionen ***PARENT***, ***LEFT***, ***RIGHT*** versehen ein Array mit einer binären Baumstruktur.
- Nehmen dabei an, dass der Baum fast vollständig ist, d. h. der Baum ist bis auf die letzte Ebene auf jeder Ebene vollständig besetzt. Die letzte Ebene ist von links nach rechts besetzt.
- ***PARENT*** liefert Elternknoten, ***LEFT*** und ***RIGHT*** liefern linkes und rechtes Kind eines Knotens.
- Liegt dabei ein Funktionswert außerhalb des Intervalls $[1, \dots, A.\text{heap-size}]$, so bedeutet dies, dass der Knoten kein Elternknoten oder kein linkes bzw. rechtes Kind besitzt.
- Es gibt nur einen Knoten ohne Eltern, dies ist die Wurzel $A[1]$.

Arrays und Bäume



Heaps

Definition 6.2

Ein Heap heißt

1. *max – Heap*, wenn für alle $1 \leq i \leq A.\text{heap-size}$ gilt:

$$A[\text{PARENT}(i)] \geq A[i].$$

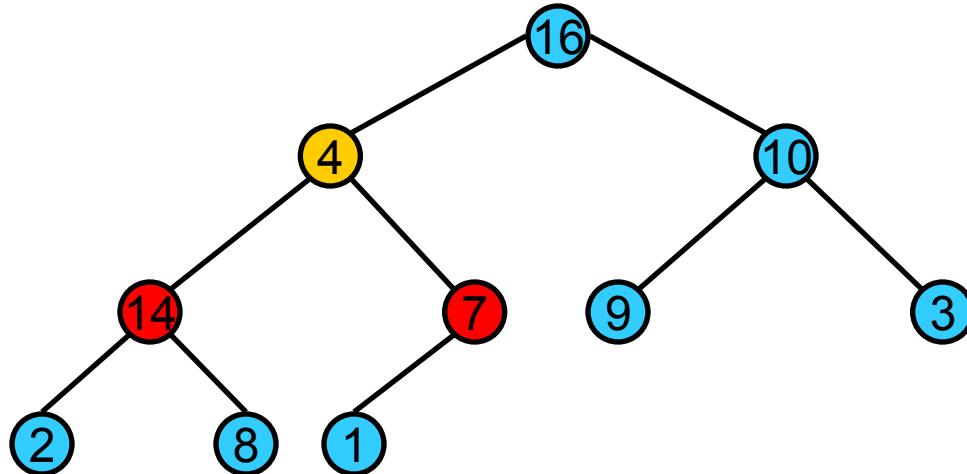
2. *min – Heap*, wenn für alle $1 \leq i \leq A.\text{heap-size}$ gilt:

$$A[\text{PARENT}(i)] \leq A[i].$$

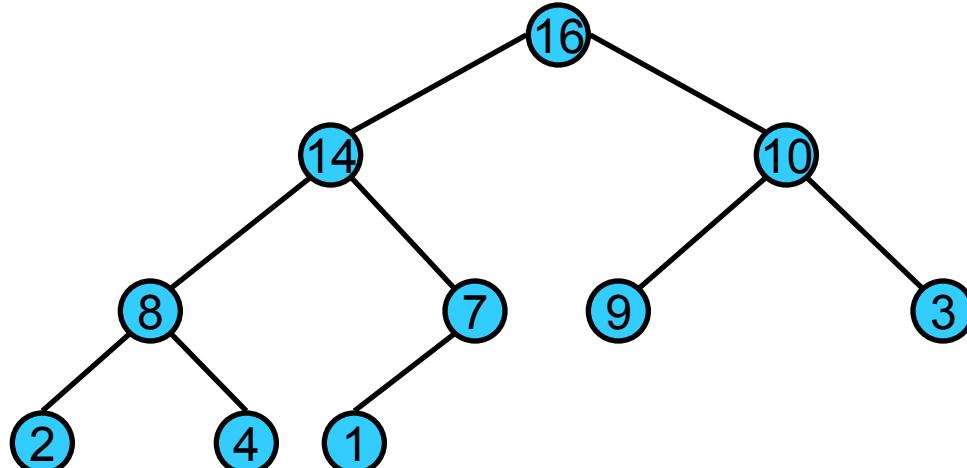
Die Eigenschaften in 1. und 2. werden *max – Heap-* bzw. *min – Heap-* Eigenschaft genannt.

Max-Heaps

Kein *max-Heap!*



max-Heap!



Höhe von Knoten und Heaps

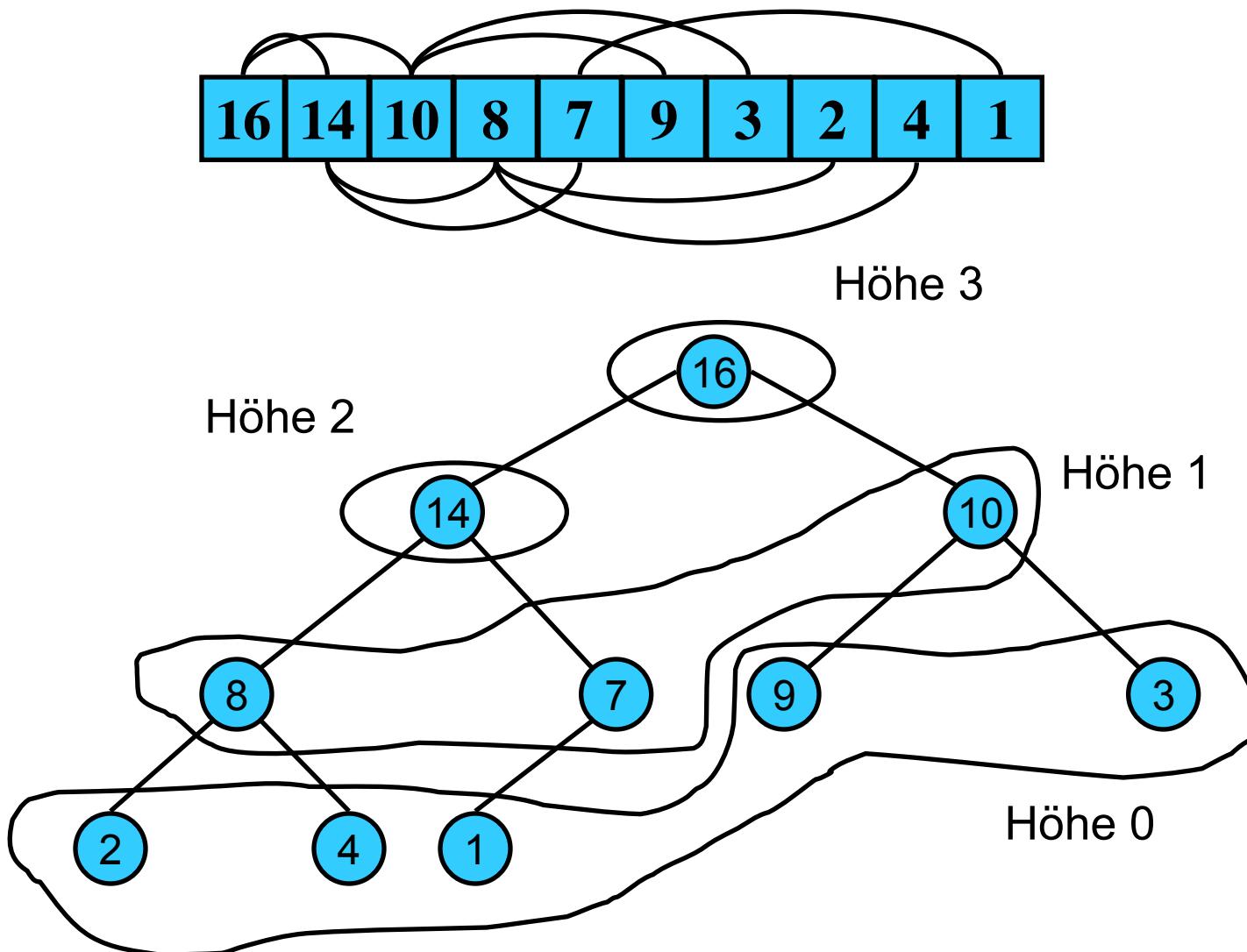
Definition 6.3

1. In einem binären Baum ist die Höhe h eines Knotens v des Baums die Länge des längsten abwärts gerichteten Pfades von v zu einem Blatt des Baums.
Dabei ist ein Pfad abwärtsgerichtet, wenn jede Kante von einem Knoten w zu einem Kind von w führt.
2. Die Höhe eines binären Baums ist die maximale Höhe eines Knotens des Baums.
3. Die Höhe eines Heaps ist die Höhe des durch den Heap gegebenen Baums.

Satz 6.4

Ein Heap mit n Elementen hat Höhe $\lfloor \log(n) \rfloor$.

Höhe eines Heaps



Algorithmen auf Heaps

- **MAX-HEAPIFY** wird benutzt, um die max-Heap-Eigenschaft aufrecht zu erhalten.
- **BUILD-MAX-HEAP** konstruiert aus einem unstrukturierten Array einen max-Heap.
- **HEAPSORT** sortiert mit Hilfe von Heaps.

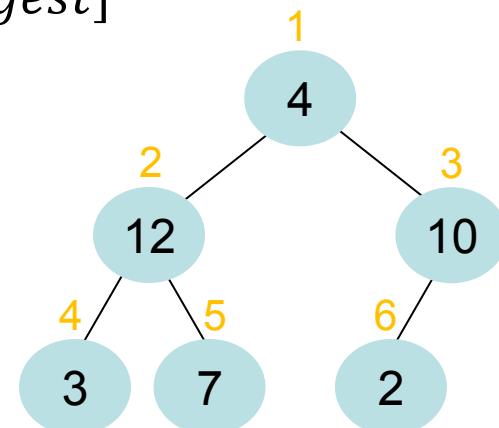
Mit Heaps können **Prioritätswarteschlangen** realisiert werden.

Max-Heapify

MAX-HEAPIFY(A, i)

```
1  $l = \text{LEFT}(i)$ 
2  $r = \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4    $largest = l$ 
5 else  $largest = i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7    $largest = r$ 
8 if  $largest \neq i$ 
9   exchange  $A[i]$  with  $A[largest]$ 
10  MAX-HEAPIFY( $A, largest$ )
```

1	2	3	4	5	6
4	12	10	3	7	2



Max-Heapify

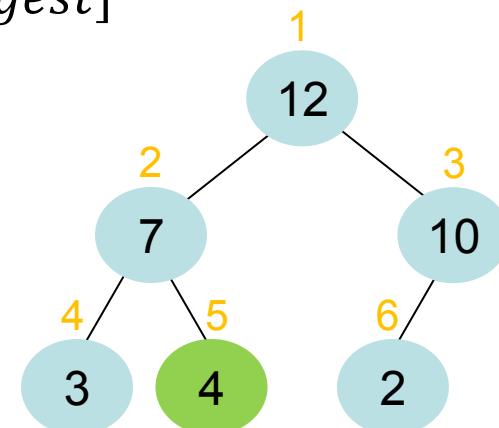
MAX-HEAPIFY(A, i)

```
1  $l = \text{LEFT}(i)$ 
2  $r = \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4    $largest = l$ 
5 else  $largest = i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7    $largest = r$ 
8 if  $largest \neq i$ 
9   exchange  $A[i]$  with  $A[largest]$ 
10  MAX-HEAPIFY( $A, largest$ )
```

1	2	3	4	5	6
12	7	10	3	4	2

$i=10$

$r=11$



Max-Heapify – Korrektheit

Lemma 6.5

Sei A ein Heap und $1 \leq i \leq A.length$. Die Teilbäume mit Wurzel $\text{LEFT}(i)$ und $\text{RIGHT}(i)$ erfüllen die max-Heap-Eigenschaft.

Dann erfüllt nach Durchlauf von $\text{MAX-HEAPIFY}(A, i)$ der Teilbaum mit Wurzel i die max-Heap-Eigenschaft.

Invariante: o.B.d.A. gilt $i = 1$.

- Vor jedem Aufruf von $\text{MAX-HEAPIFY}(A, \text{largest})$ (Zeile 10) wird die max-Heap-Eigenschaft, wenn überhaupt, nur von $\text{LEFT}(\text{largest})$ und/oder $\text{RIGHT}(\text{largest})$ verletzt.
- Des Weiteren gilt: $\text{PARENT}(\text{largest}) \geq \text{LEFT}(\text{largest})$ und $\text{PARENT}(\text{largest}) \geq \text{RIGHT}(\text{largest})$.

Max-Heapify – Analyse

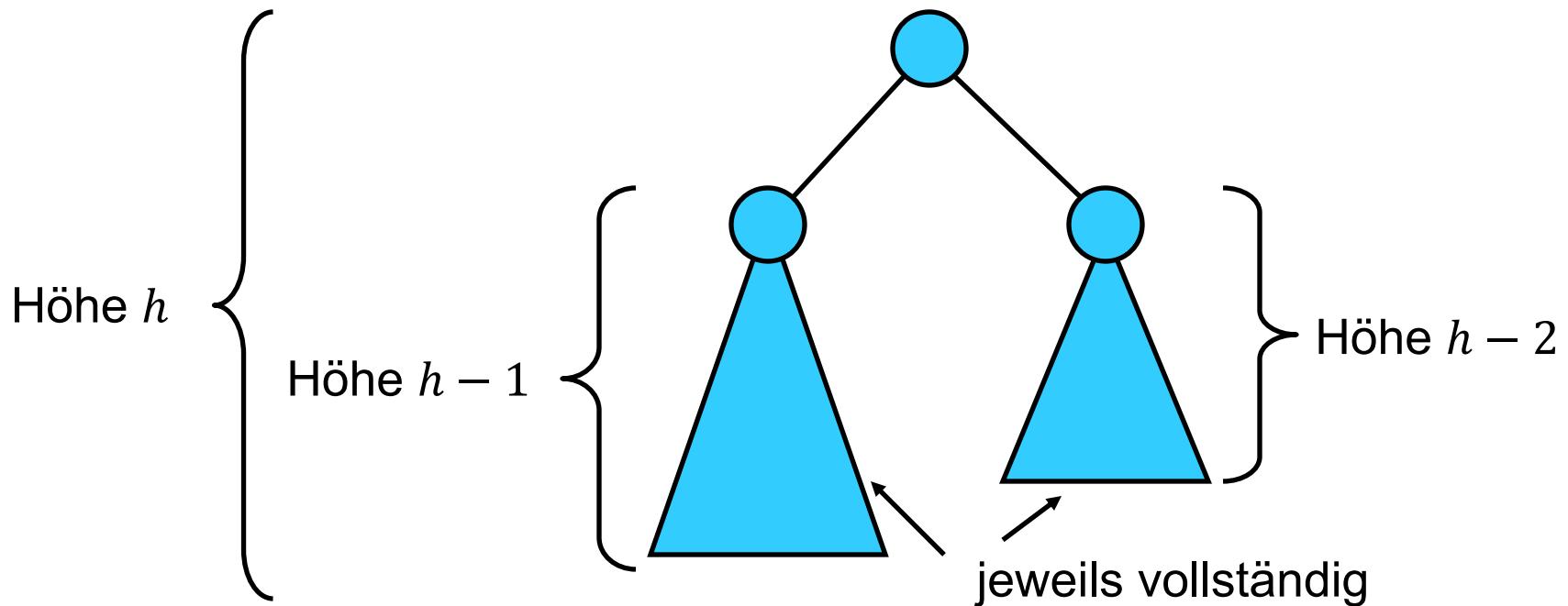
Lemma 6.6

1. Ist h die Höhe des Teilbaums mit Wurzel i im Heap A ,
so hat MAX-HEAPIFY(A, i) Laufzeit $\Theta(h)$.
2. Ist n die Größe des Teilbaums mit Wurzel i im Heap A ,
so hat MAX-HEAPIFY(A, i) Laufzeit $\Theta(\log(n))$.

Lemma 6.7

In einem Heap der Größe n haben sowohl der linke als auch der rechte
Teilbaum der Wurzel höchstens Größe $\frac{2}{3}n$.

Maximal unbalancierte Teilbäume



Gesamtgröße des Baums:

$$2^h - 1 + 2^{h-1} - 1 + 1 = 3 \cdot 2^{h-1} - 1$$

Größe rechter Teilbaum:

$$2^{h-1} - 1$$

$$\frac{\text{Größe rechter/linker Teilbaum}}{\text{Größe Gesamtbaum}} \leq \frac{2}{3}$$

Aufbau eines Heaps

Eingabe: (unstrukturiertes) Array A

Ausgabe: max-Heap über A

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \left\lfloor \frac{A.\text{length}}{2} \right\rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

Aufbau eines Heaps

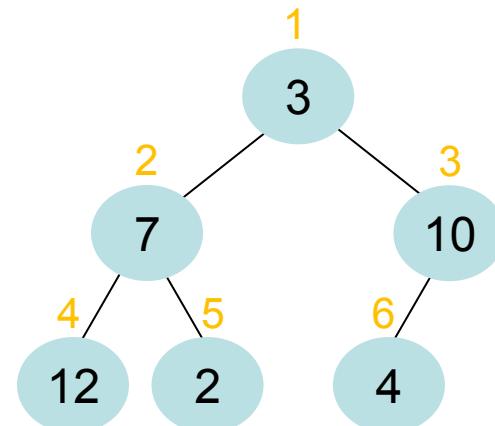
Aufbau eines Heaps:

- jedes Blatt ist ein Heap
- baue Heap „von unten nach oben“ mit MAX-HEAPIFY auf

1	2	3	4	5	6
3	7	10	12	2	4

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \left\lfloor \frac{A.\text{length}}{2} \right\rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)



Aufbau eines Heaps

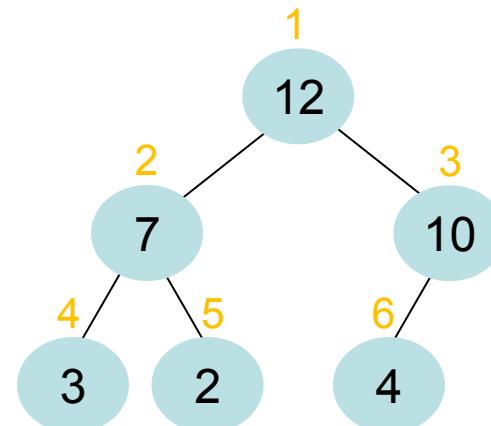
Aufbau eines Heaps:

- jedes Blatt ist ein Heap
- baue Heap „von unten nach oben“ mit MAX-HEAPIFY auf

1	2	3	4	5	6
12	7	10	3	2	4

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \left\lfloor \frac{A.\text{length}}{2} \right\rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)



Invariante für Build-Max-Heap

Invariante: Vor Durchlauf der for-Schleife in den Zeilen 2 und 3 für Index i sind die Knoten $i + 1, \dots, n$ Wurzeln von max-Heaps.

Initialisierung: Zu Beginn gilt $i = \left\lfloor \frac{n}{2} \right\rfloor$. Knoten $i > \left\lfloor \frac{n}{2} \right\rfloor$ sind Blätter und damit Wurzeln von max-Heaps.

Erhaltung: MAX-HEAPIFY(A, i) stellt max-Heap-Eigenschaft bei Knoten i her und erhält max-Heap-Eigenschaft bei Knoten $j > i$.

Terminierung: Knoten 1 ist Wurzel eines max-Heaps. Damit ist der Algorithmus korrekt.

Analyse für Build-Max-Heap (1)

1. Ein n -elementiger Heap hat Höhe $\lfloor \log(n) \rfloor$ und besitzt höchstens $\left\lceil \frac{n}{2^h} \right\rceil$ Knoten der Höhe h .
2. Es existiert eine Konstante c , so dass MAX-HEAPIFY bei Knoten der Höhe h Laufzeit höchstens ch besitzt.
3. Laufzeit von BUILD-MAX-HEAP ist dann höchstens

$$\sum_{h=0}^{\lfloor \log(n) \rfloor} \left\lceil \frac{n}{2^h} \right\rceil ch \leq 4cn \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}$$

Analyse für Build-Max-Heap (2)

4. Es gilt

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

5. Damit ist

$$\sum_{h=0}^{\lfloor \log(n) \rfloor} \left\lceil \frac{n}{2^h} \right\rceil ch \leq 2cn \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h} \leq 2cn \sum_{h=0}^{\infty} \frac{h}{2^h} = 4cn$$

Satz 6.8

Die Laufzeit von BUILD-MAX-HEAP ist $O(n)$.

Heapsort

Eingabe: Array A

Ausgabe: Zahlen in A in aufsteigender Reihenfolge sortiert.

HEAPSORT(A)

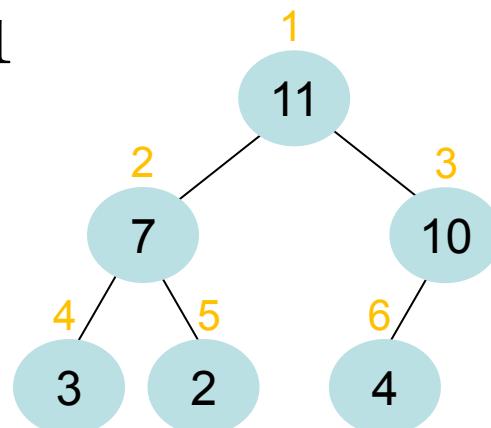
- 1 BUILD-MAX-HEAP(A)
- 2 **for** $i = A.length$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap-size = A.heap-size - 1$
- 5 MAX-HEAPIFY($A, 1$)

Heapsort

HEAPSORT(A)

```
1 BUILD-MAX-HEAP( $A$ )
2 for  $i = A.length$  downto 2
3   exchange  $A[1]$  with  $A[i]$ 
4    $A.heap-size = A.heap-size - 1$ 
5   MAX-HEAPIFY( $A, 1$ )
```

1	2	3	4	5	6
11	7	10	3	2	4



Heapsort

HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 **for** $i = A.length$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap-size = A.heap-size - 1$
- 5 MAX-HEAPIFY($A, 1$)

1	2	3	4	5	6
2	3	4	7	10	11

Laufzeit von Heapsort

Satz 6.9

Heapsort besitzt Laufzeit $O(n \log n)$.

Beweisskizze:

1. Aufruf von BUILD-MAX-HEAP: $O(n)$.
2. die `for`-Schleife in Zeilen 2-5 $(n - 1)$ -mal durchlaufen.
3. Pro Durchlauf Laufzeit $O(\log(n))$ (MAX-HEAPIFY).

Extract Max

Maximum-Suche

HEAP-EXTRACT-MAX (A)

- 1 if $A.\text{heap-size} < 1$
- 2 **error** „heap underflow“
- 3 $\max = A[1]$
- 4 $A[1] = A[A.\text{heap-size}]$
- 5 $A.\text{heap-size} = A.\text{heap-size} - 1$
- 6 MAX-HEAPIFY($A, 1$)
- 7 **return** \max

Insert Key

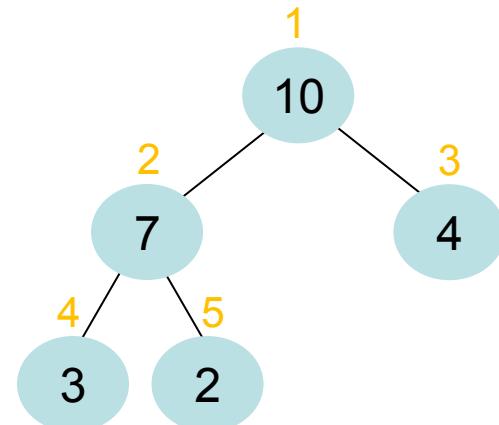
1	2	3	4	5
10	7	4	3	2

HEAP-INCREASE-KEY(A, i, key)

- 1 **if** $key < A[i]$
- 2 **error** „new key is smaller than current key“
- 3 $A[i] = key$
- 4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
- 5 exchange $A[i]$ with $A[\text{PARENT}(i)]$
- 6 $i = \text{PARENT}(i)$

MAX-HEAP-INSERT (A, key)

- 1 $A.\text{heap-size} = A.\text{heap-size} + 1$
- 2 $A[A.\text{heap-size}] = -\infty$
- 3 HEAP-INCREASE-KEY($A, A.\text{heap-size}, key$)



MAX-HEAP-INSERT ($A, 11$)

Insert Key

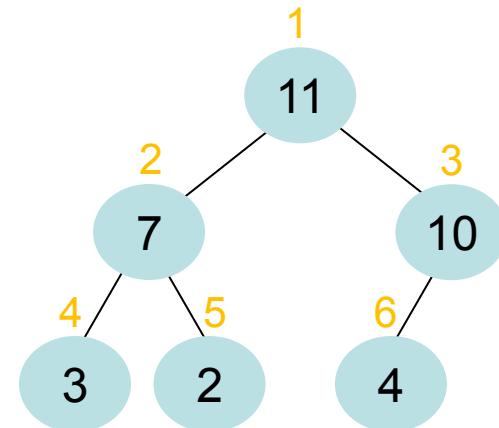
1	2	3	4	5	6
11	7	10	3	2	4

HEAP-INCREASE-KEY(A, i, key)

- 1 **if** $key < A[i]$
- 2 **error** „new key is smaller than current key“
- 3 $A[i] = key$
- 4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
- 5 exchange $A[i]$ with $A[\text{PARENT}(i)]$
- 6 $i = \text{PARENT}(i)$

MAX-HEAP-INSERT (A, key)

- 1 $A.\text{heap-size} = A.\text{heap-size} + 1$
- 2 $A[A.\text{heap-size}] = -\infty$
- 3 HEAP-INCREASE-KEY($A, A.\text{heap-size}, key$)



Laufzeit: $O(\log n)$

Heap-Insert – Korrektheit

Invariante:

Vor dem Schleifendurchlauf mit Index i gilt:

- Die *max-Heap*-Eigenschaft ist erfüllt im gesamten Heap außer vielleicht an der Stelle $A[i]$ (d.h. $A[i] > A[PARENT(i)]$)
- alle Elemente von A sind im Heap enthalten
- das neu einzufügende Element ist in $A[i]$

Initialisierung:

Vor dem ersten Schleifendurchlauf gilt:

$i = A.heap_size$, $A[i]$ enthält das neue Element; *max-Heap*-Eigenschaft ist erfüllt außer vielleicht an der Stelle $A[i]$

Heap-Insert – Korrektheit

Erhaltung:

I.V.: Die Invariante gilt vor dem Schleifendurchlauf mit Index i .

I.S. (i wird zu $\text{PARENT}(i)$):

Vor dem Schleifendurchlauf mit Index i gilt:

- falls $A[\text{PARENT}(i)] < key$ und $i > 1$
 - $A[i] \leftrightarrow A[\text{PARENT}(i)]$
 - $i = \text{PARENT}(i)$
- falls $A[\text{PARENT}(i)] \geq key$ und $i > 1$
 - Schleife wird beendet, $A[i] = key$
- $i = 1$
 - Schleife wird beendet, $A[i] = key$

Heap-Insert – Korrektheit

Terminierung:

Die Schleifeninvariante terminiert, wenn $A[\text{PARENT}(i)] \geq key$ oder $i = 1$.

- falls $i = A.\text{heap-size}$, dann ist die max-Heap-Eigenschaft trivialerweise erfüllt
- falls $i < A.\text{heap-size}$, dann ist $A[\text{PARENT}(i)] \geq key$ oder $i = 1$; da die max-Heap-Eigenschaft nach dem letzten Schleifendurchlauf erfüllt war und $A[i] = key$ ist, ist die max-Heap-Eigenschaft im gesamten Heap erhalten

Laufzeit von Sortieralgorithmen

	Laufzeit	
	worst-case	average-case
Algorithmus		
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge-Sort	$\Theta(n \log(n))$	$\Theta(n \log(n))$
Quick-Sort	$\Theta(n^2)$	$\Theta(n \log(n))$
Heap-Sort	$\Theta(n \log(n))$	-

7. Untere Schranken für Sortieren

- Alle bislang betrachteten Sortieralgorithmen hatten Laufzeit $\Omega(n \log(n))$.
- Werden nun gemeinsame Eigenschaften dieser Algorithmen untersuchen.
- Fassen gemeinsame Eigenschaften in Modell des Vergleichssortierers zusammen.
- Zeigen dann, dass jeder Vergleichssortierer Laufzeit $\Omega(n \log(n))$ besitzt.

Vergleichssortierer

Definition 7.1

Ein Vergleichssortierer ist ein Algorithmus, der zu jeder beliebigen Eingabefolge (a_1, a_2, \dots, a_n) von Zahlen eine Permutation π berechnet, so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$.

Dabei benutzt ein Vergleichssortierer außer den durch den Pseudocode definierten Kontrolloperationen nur die Vergleichsoperationen $=, \neq, \leq, \geq, <, >$.

Bemerkungen:

1. Nehmen an, dass Eingabezahlen immer paarweise verschieden sind. Benötigen daher $=$ nicht.
2. Können uns auf den Vergleich \leq einschränken.
Andere Vergleiche sind hierzu äquivalent.

Vergleichssortierer

Definition 7.2

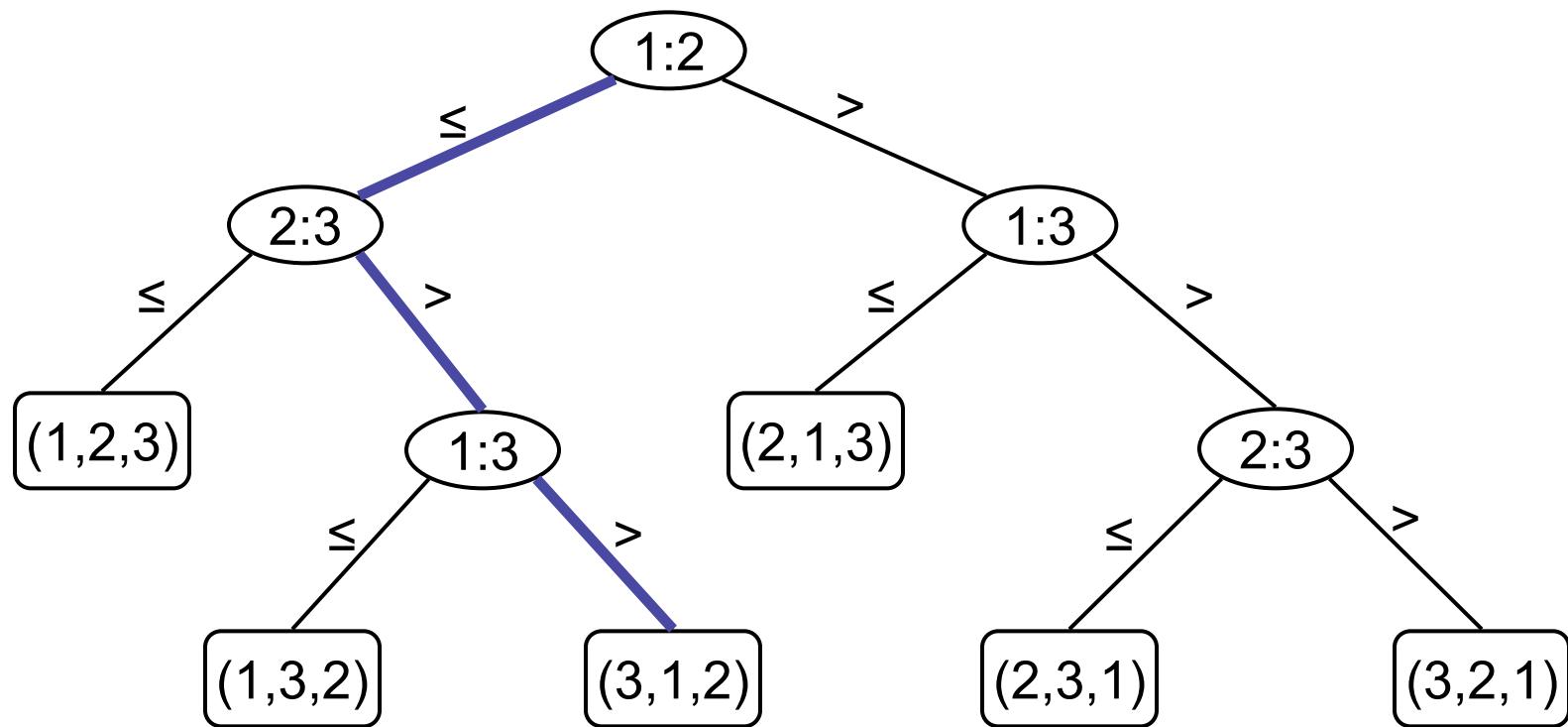
Ein Entscheidungsbaum über n Zahlen ist ein binärer Baum, bei dem

1. Jeder innere Knoten mit $i:j, 1 \leq i, j \leq n$ gelabelt ist.
2. Jedes Blatt mit einer Permutation π auf $\{1, \dots, n\}$ gelabelt ist.

Entscheidungsbäume und Sortieren

- Mit Entscheidungsbäumen können Vergleichssortierer modelliert werden. Hierzu:
 - wird bei Eingabe (a_1, \dots, a_n) ein Pfad von der Wurzel des Baums zu einem Blatt des Baums durchlaufen.
 - wird an einem inneren Knoten gelabelt mit $i: j$ die Kante zum linken Kind genommen, falls $a_i \leq a_j$, sonst wird die Kante zum rechten Kind genommen.
 - wird die Permutation π des Blattes am Ende des Pfades ausgegeben
- Zu einem Vergleichssortierer gibt es für jede Eingabegröße n einen Entscheidungsbau.

Entscheidungsbaum für Insertion-Sort



Eingabe: $a_1 = 6, a_2 = 8, a_3 = 5$

Untere Schranke für Vergleichssortierer

Lemma 7.3

Für Eingaben der Größe n hat ein Entscheidungsbaum für einen Vergleichssortierer mindestens $n!$ Blätter.

Lemma 7.4

$$\log(n!) = \Theta(n \log(n))$$

Satz 7.5

Die von einem Vergleichssortierer bei Eingabegröße n benötigte Anzahl von Vergleichen ist $\Theta(n \log(n))$.

Korollar 7.6

Die von Merge-Sort und Heap-Sort benötigte Laufzeit von $\Theta(n \log(n))$ ist asymptotisch optimal.

8. Sortieren in linearer Zeit

- Es gibt Algorithmen, die die $\Omega(n \log(n))$ untere Schranke für Vergleichssortierer schlagen. Diese Algorithmen haben u.U. Laufzeit $O(n)$.
- Diese Algorithmen benutzen neben Vergleichen und Kontrolloperationen noch andere Operationen, z.B. arithmetische Operationen auf den zu sortierenden Zahlen.
- Außerdem werden noch Annahmen über die zu sortierenden Zahlen getroffen, um die Laufzeit von $O(n)$ zu zeigen und die Korrektheit der Algorithmen zu beweisen.
- Zu diesen Algorithmen gehören ***Counting-Sort***, ***Bucket-Sort*** und ***Radix-Sort***.

Sortieren durch Abzählen (1)

Annahme:

Es gibt einen, dem Algorithmus Counting-Sort bekannten Parameter k , so dass für die Eingabefolge (a_1, \dots, a_n) gilt:

$$0 \leq a_i \leq k \text{ mit } a_i \text{ natürliche Zahlen für alle } 1 \leq i \leq n$$

Algorithmusidee:

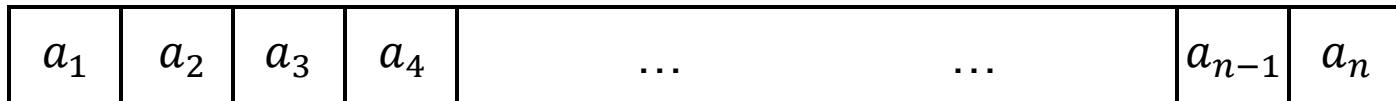
1. Für alle i , $0 \leq i \leq k$ bestimme Anzahl C_i der a_j mit $a_j \leq i$.
2. Kopiere a_j mit $a_j = i$ in Felder $B[C_{i-1} + 1], \dots, B[C_i]$ eines Arrays B mit $B.length = n$. Dabei gilt $C_{-1} = 0$.

Es gilt:

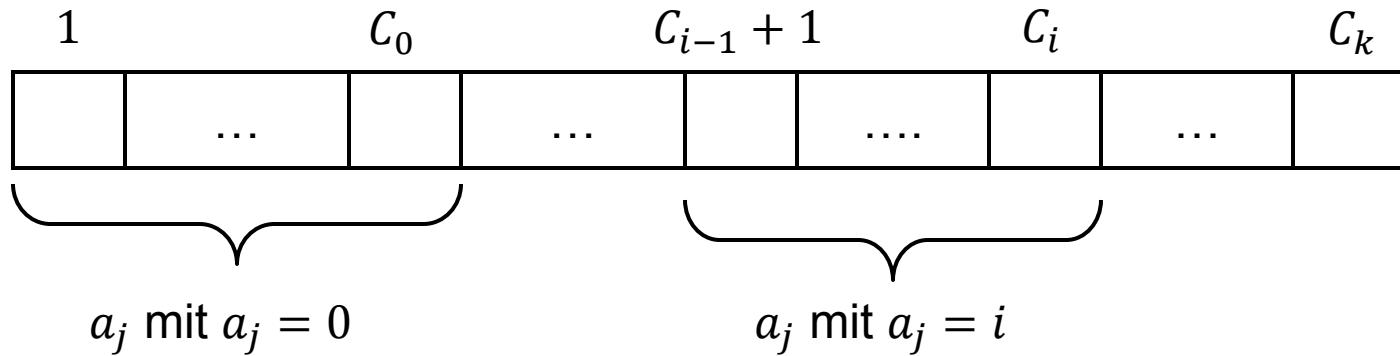
$C_i - C_{i-1}$ ist die Anzahl der a_j mit $a_j = i$.

Sortieren durch Abzählen (2)

A



B



Counting-Sort

COUNTING-SORT(A, B, k)

```
1 let  $C[0 \dots k]$  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 number of elements equal to  $i$ 
7 for  $i = 1$  to  $k$ 
8    $C[i] = C[i] + C[i - 1]$ 
9 //  $C[i]$  now contains number of elements less than or equal to  $i$ 
10 for  $j = A.length$  downto 1
11    $B[C[A[j]]] = A[j]$ 
12    $C[A[j]] = C[A[j]] - 1$ 
```

Illustration für Counting-Sort

A

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

C
(nach Zeilen 4-5)

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

C
(nach Zeilen 7-8)

Illustration für Counting-Sort

A

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

C
(nach Zeilen 4-5)

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

C
(nach Zeilen 7-8)

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

Illustration für Counting-Sort

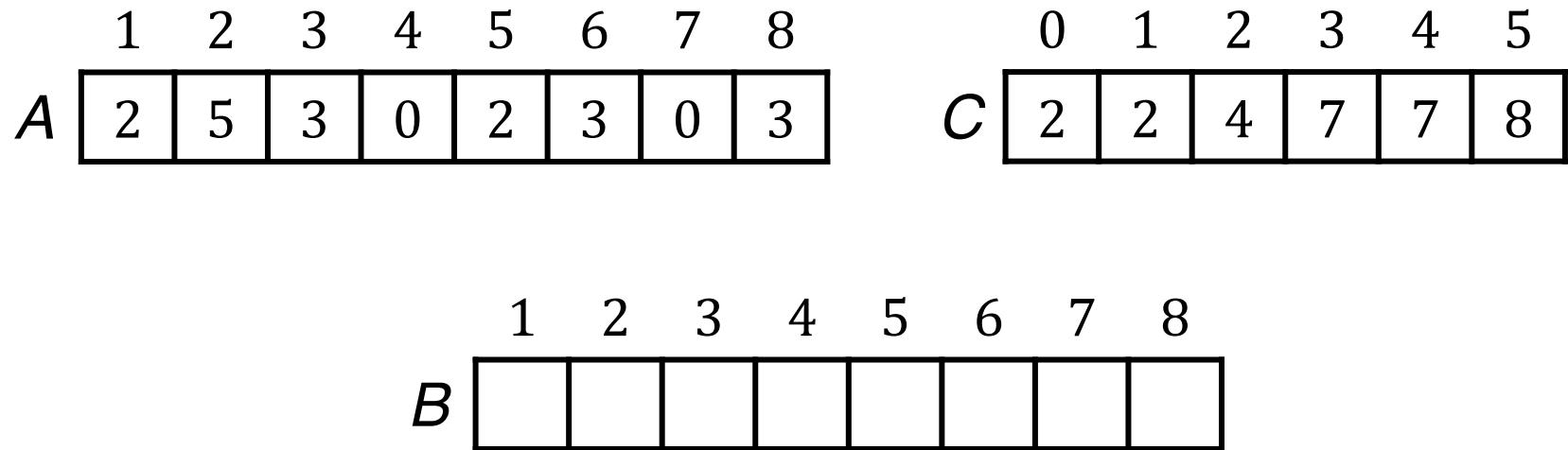


Illustration für Counting-Sort

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

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

	0	1	2	3	4	5
C	0	2	2	4	7	7

Analyse von Counting-Sort (1)

COUNTING-SORT(A, B, k)

```
1 let  $C[0 \dots k]$  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 number of elements equal to  $i$ 
7 for  $i = 1$  to  $k$ 
8    $C[i] = C[i] + C[i - 1]$ 
9 //  $C[i]$  now contains number of elements less than or equal to  $i$ 
10 for  $j = A.length$  downto 1
11    $B[C[A[j]]] = A[j]$ 
12    $C[A[j]] = C[A[j]] - 1$ 
```

- Zeilen 2-3, 7-8: jeweils $O(k)$
- Zeilen 4-5, 10-12: jeweils $O(n)$

Analyse von Counting-Sort (2)

Satz 8.1

Counting-Sort besitzt Laufzeit $O(n + k)$.

Korollar 8.2

Gilt $k = O(n)$, so besitzt Counting-Sort Laufzeit $O(n)$.

9. Elementare Datenstrukturen

Definition 9.1

Eine dynamische Menge ist gegeben durch eine oder mehrere Mengen von Objekten sowie Operationen auf diesen Mengen und den Objekten der Mengen. Dynamische Mengen werden auch abstrakte Datentypen (ADT) genannt.

Definition 9.2

1. Werden auf einer Menge von Objekten die Operationen Einfügen, Entfernen und Suchen betrachtet, so spricht man von einem Wörterbuch.
2. Werden auf einer Menge von Objekten die Operationen Einfügen, Entfernen und Suchen des Maximums betrachtet, so spricht man von einer Warteschlange.

Datenstrukturen

Ein grundlegendes Datenbank-Problem

- Speicherung von Datensätzen

Beispiel:

- Kundendaten
 - Name, Adresse, Wohnort,
 - Kundennummer, offene Rechnungen, offene Bestellungen, ...

Anforderungen:

- Schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

Objekte in dynamischen Mengen

- Objekte bestehen aus verschiedenen Feldern.
- Ein Feld speichert den **Schlüssel** des Objekts.
- Identifikation eines Objekts erfolgt über Schlüssel.
- Schlüssel von Objekten sind nicht notwendig paarweise verschieden.
- Falls Schlüssel paarweise verschieden sind, ist Menge von Objekten identisch zur Menge von Schlüsseln.
- weitere Felder relevant für Datenstruktur
- z.B. Felder für Referenzen auf andere Objekte
- Andere Felder nur relevant für Anwendung, nicht für Datenstruktur
– die Daten dieser Felder nennt man Satellitendaten.

Datenstrukturen

Zugriff auf Daten:

- Jedes Objekt hat einen Schlüssel
- Eingabe des Schlüssels liefert Datensatz
- Schlüssel sind vergleichbar
 - es gibt totale Ordnung der Schlüssel

Beispiel:

- Kundendaten
 - Name, Adresse, Kundennummer
- Schlüssel: Name
- Totale Ordnung: Lexikographische Ordnung

Datenstrukturen

Zugriff auf Daten:

- Jedes Objekt hat einen Schlüssel
- Eingabe des Schlüssels liefert Datensatz
- Schlüssel sind vergleichbar
 - es gibt totale Ordnung der Schlüssel

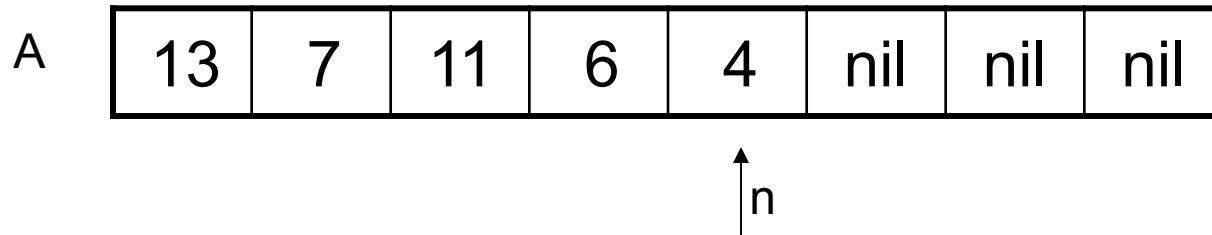
Beispiel:

- Kundendaten
 - Name, Adresse, Kundennummer
- Schlüssel: Kundennummer
- Totale Ordnung: \leq

Datenstrukturen

Unsere erste Datenstruktur:

- Feld $A[1, \dots, max]$
- Integer $n, 1 \leq n \leq max$
- n bezeichnet Anzahl Elemente in Datenstruktur

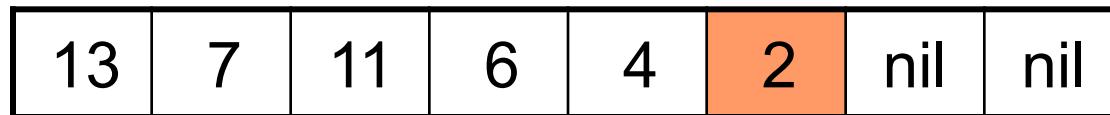


Datenstrukturen

EINFÜGEN(s)

```
1 if  $n = max$ 
2   return „Fehler: Kein Platz in Datenstruktur“
3 else
4    $n = n + 1$ 
5    $A[n] = s$ 
```

A



↑
 n

Einfügen(2)

Datenstrukturen

EINFÜGEN(s)

```
1 if  $n = max$ 
2   return „Fehler: Kein Platz in Datenstruktur“
3 else
4    $n = n + 1$ 
5    $A[n] = s$ 
```

A

13	7	11	6	4	2	nil	nil
----	---	----	---	---	---	-----	-----



Laufzeit: $\Theta(1)$

Datenstrukturen

SUCHE(x)

```
1 for  $i = 1$  to  $n$ 
2   if  $A[i] = x$ 
3     return  $i$ 
4 return nil
```

$i=3$

A

13	7	11	6	4	2	nil	nil
----	---	----	---	---	---	-----	-----

$\uparrow n$

Suche(11)

Datenstrukturen

SUCHE(x)

```
1 for  $i = 1$  to  $n$ 
2   if  $A[i] = x$ 
3     return  $i$ 
4 return nil
```

A

13	7	11	6	4	2	nil	nil
----	---	----	---	---	---	-----	-----

↑
n

Laufzeit: $\Theta(n)$

Datenstrukturen

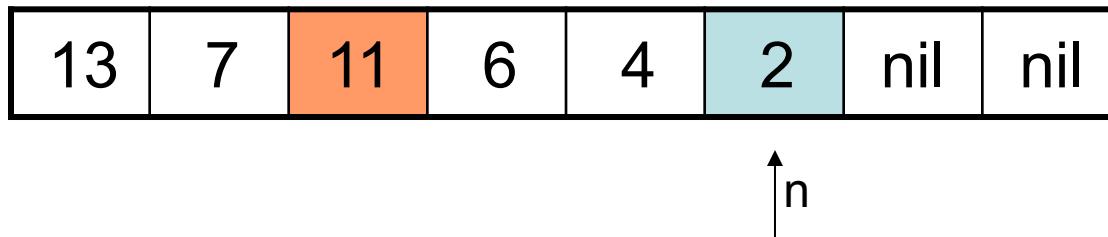
LÖSCHEN(i)

1 $A[i] = A[n]$

2 $A[n] = \text{nil}$

3 $n = n - 1$

A



Löschen(3)

Datenstrukturen

LÖSCHEN(i)

- 1 $A[i] = A[n]$
- 2 $A[n] = \text{nil}$
- 3 $n = n - 1$

A

13	7	2	6	4	nil	nil	nil
----	---	---	---	---	-----	-----	-----

↑
n

Laufzeit: $\Theta(1)$

Datenstrukturen

Datenstruktur Feld:

- Platzbedarf $\Theta(\max)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile:

- Schnelles Einfügen und Löschen

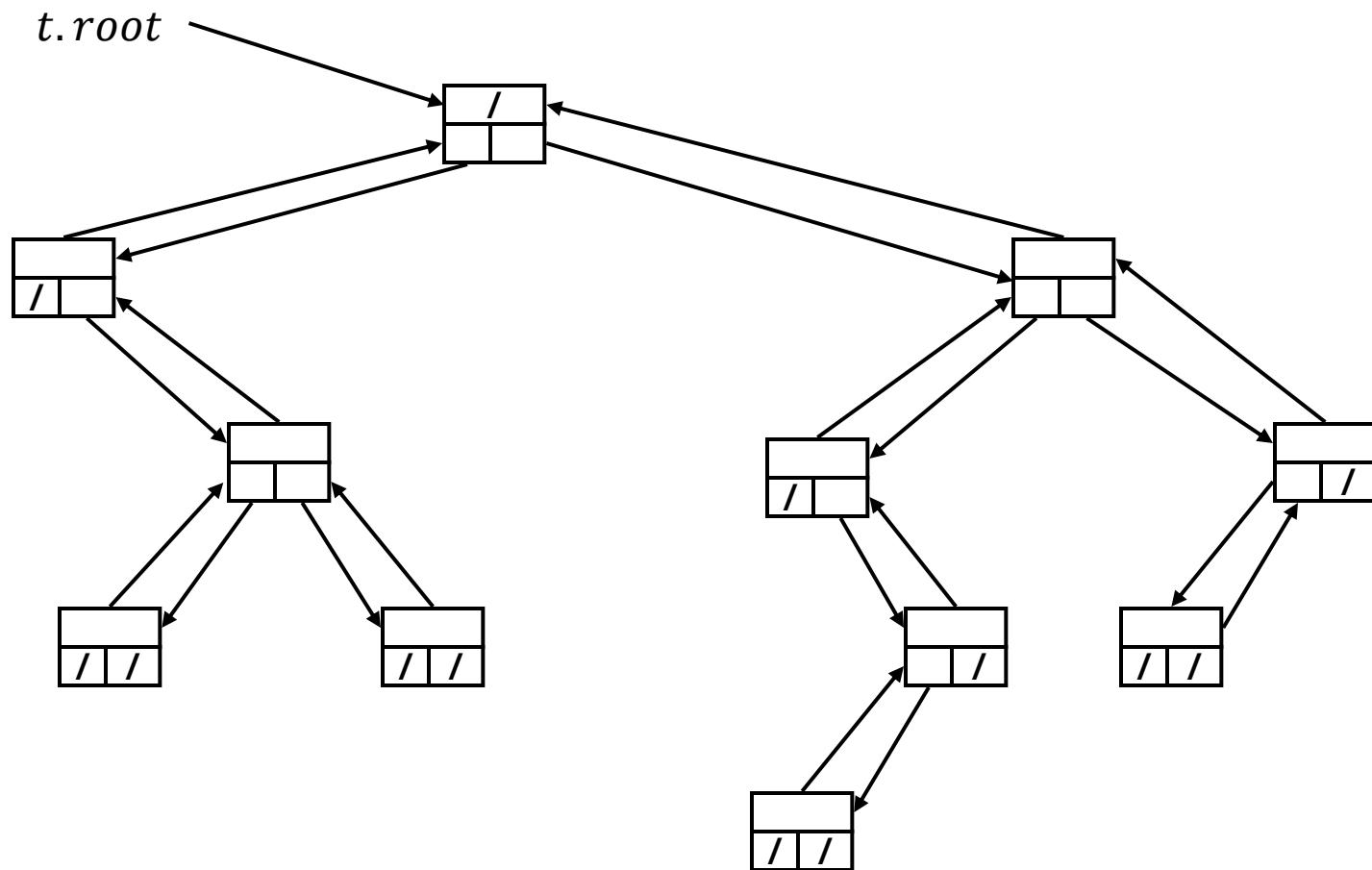
Nachteile:

- Speicherbedarf abhängig von \max (nicht vorhersagbar)
- Hohe Laufzeit für Suche

Binäre Bäume

- Neben Felder für Schlüssel und Satellitendaten Felder `parent`, `left`, `right`.
- x Knoten:
 - $x.parent$ (Verweis auf Elternknoten),
 - $x.left$ (Verweis auf linkes Kind),
 - $x.right$ (Verweis auf rechtes Kind)
- Falls $x.parent == NIL$, dann ist x Wurzel des Baums.
- $x.left/x.right == NIL$: kein linkes/rechtes Kind.
- Zugriff auf Baum t durch Verweis $t.root$ auf Wurzelknoten.

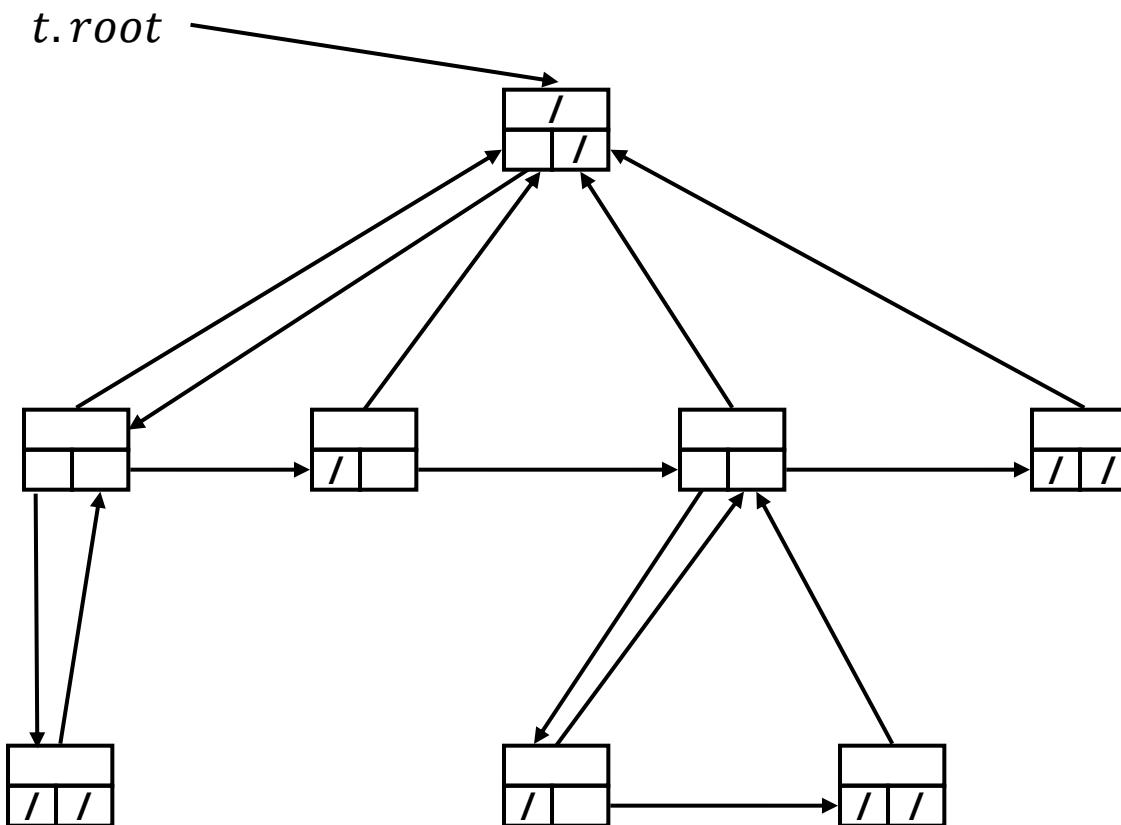
Binäre Bäume - Illustration



Allgemeine Bäume

- Darstellung für binäre Bäume auch möglich für k -näre Bäume, k fest.
- Ersetze $x.left/x.right$ durch $x.child1, \dots, x.childk$.
- Bei Bäumen mit unbeschränktem Grad nicht möglich,
- oder ineffizient, da Speicher für viele Felder reserviert werden muss.
- Nutzen linkes-Kind/rechtes-Geschwister- Darstellung.
- Feld $parent$ für Eltern bleibt. Dann Feld $left$ für linkes Kind und Feld $right - sibling$ für rechtes Geschwister.

Allgemeine Bäume



Binäre Bäume

Binäre Suchbäume

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“

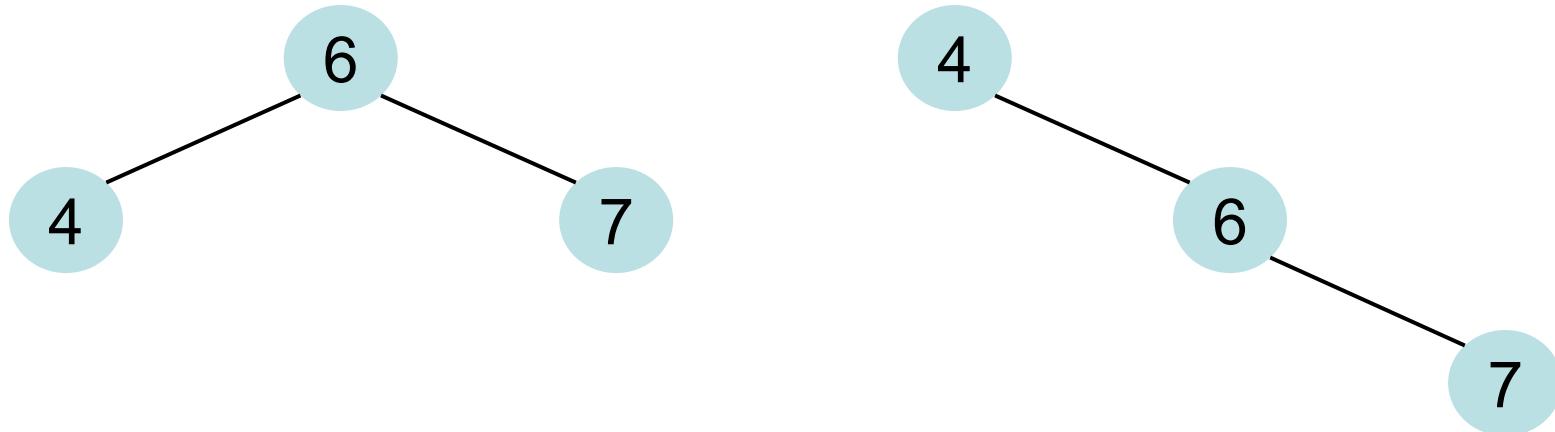
Binäre Suchbaumeigenschaft:

- Sei x Knoten im binären Suchbaum
- Ist y Knoten im **linken Unterbaum** von x , dann gilt
 $y.key \leq x.key$
- Ist y Knoten im **rechten Unterbaum** von x , dann gilt
 $y.key \geq x.key$

Binäre Suchbäume

Unterschiedliche Suchbäume

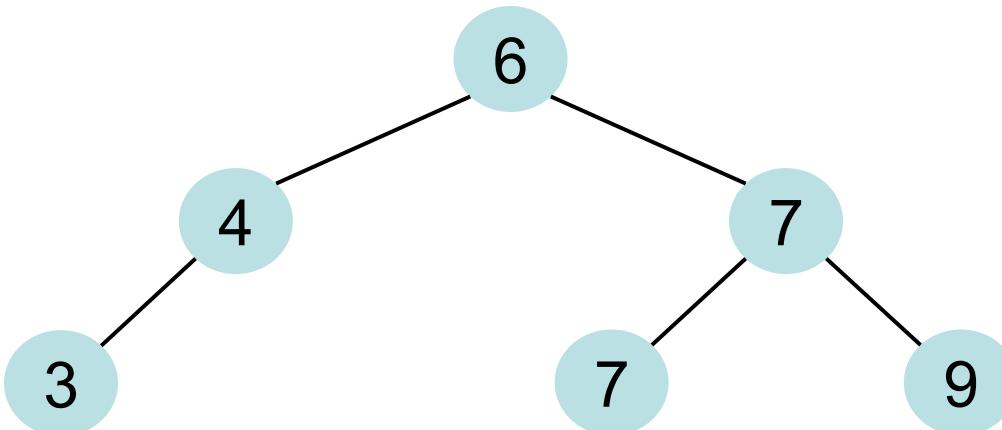
- Schlüsselmenge 4, 6, 7
- Wir erlauben mehrfaches Vorkommen desselben Schlüssels



Binäre Suchbäume

Ausgabe aller Schlüssel

- Gegeben binärer Suchbaum
- Wie kann man alle Schlüssel aufsteigend sortiert in $\Theta(n)$ Zeit ausgeben?



Binäre Suchbäume

Inorder–Tree–Walk(x)

```
1  if  $x \neq NIL$ 
2      Inorder–Tree–Walk( $x.left$ )
3      print  $x.key$ 
4      Inorder–Tree–Walk( $x.right$ )
```

Laufzeit $\Theta(n)$

Binäre Suchbäume

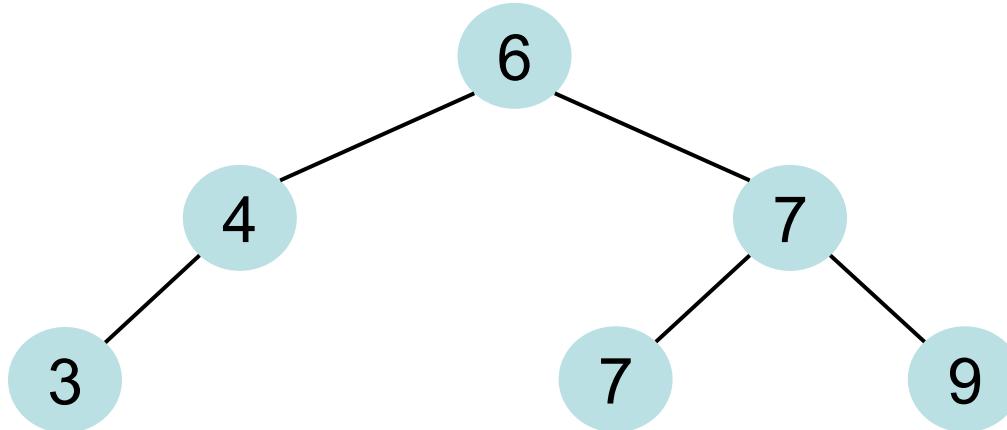
Suchen in Binärbäumen

- Gegeben ist Schlüssel k
- Gesucht ist ein Knoten mit Schlüssel k

Binäre Suchbäume

Baumsuche(x, k)

```
1 if  $x == NIL$  or  $k == x.key$ 
2   return  $x$ 
3 if  $k < x.key$ 
4   return Baumsuche( $x.left, k$ )
5 else return Baumsuche( $x.right, k$ )
```



Binäre Suchbäume

Weitere Operationen in Binärbäumen

- Minimum/Maximumsuche
- Nachfolger/Vorgänger

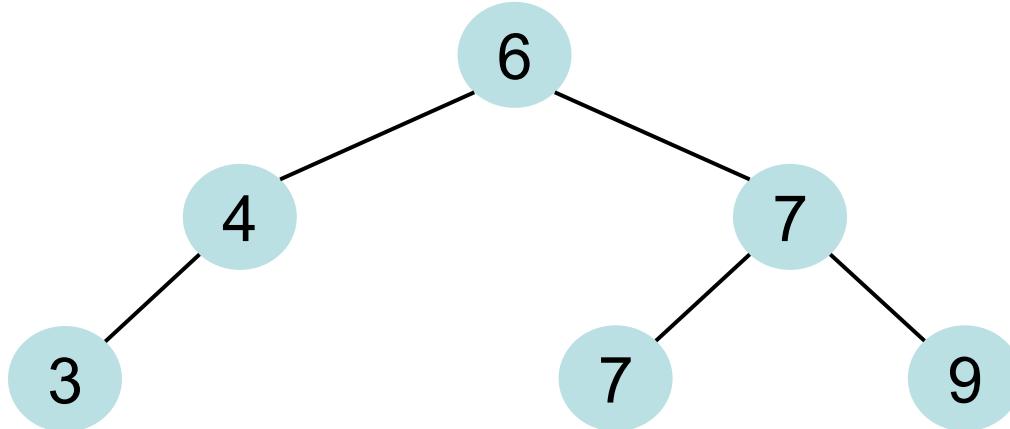
Minimum- und Maximumsuche:

- Suchbaumeigenschaft:
- Alle Knoten im rechten Unterbaum eines Knotens x sind größer gleich $x.key$
- Alle Knoten im linken Unterbaum von x sind $\leq x.key$

Binäre Suchbäume

Tree-Minimum(x)

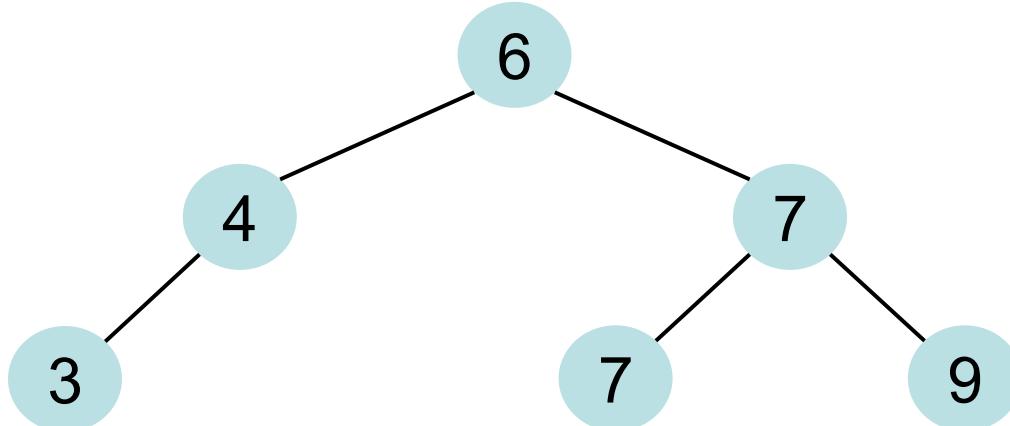
```
1 while  $x.left \neq NIL$ 
2    $x = x.left$ 
3 return  $x$ 
```



Binäre Suchbäume

Tree-Maximum(x)

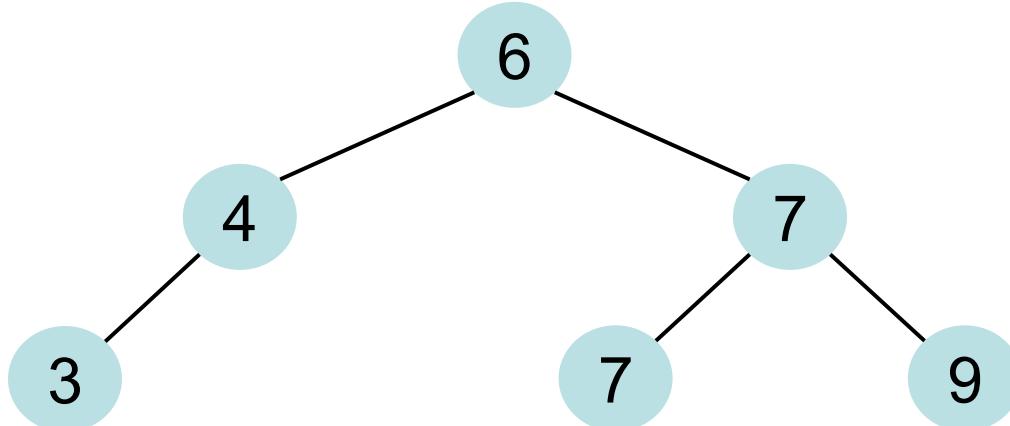
```
1 while  $x.right \neq NIL$ 
2    $x = x.right$ 
3 return  $x$ 
```



Binäre Suchbäume

Nachfolgersuche:

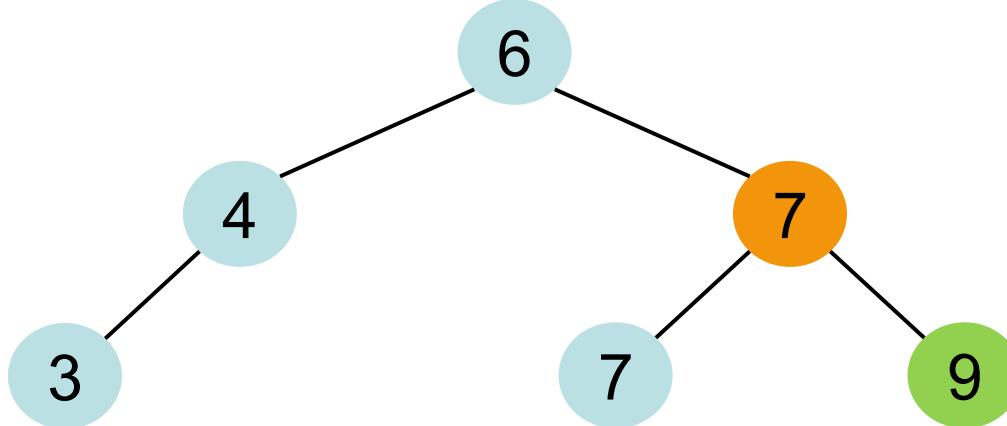
- Nachfolger bzgl. Inorder–Tree–Walk
- Wenn alle Schlüssel unterschiedlich, dann ist das der nächstgrößere Schlüssel



Binäre Suchbäume

Nachfolgersuche:

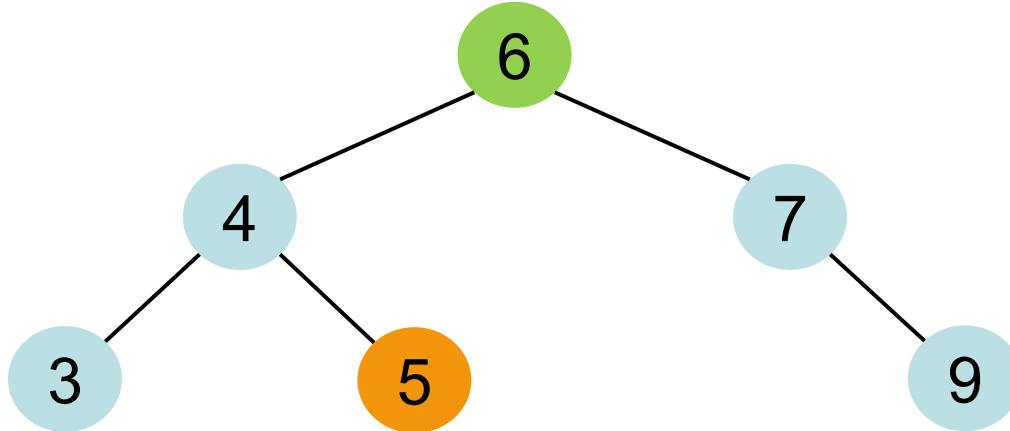
- Fall 1 (rechter Unterbaum von x nicht leer):
Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Binäre Suchbäume

Nachfolgersuche:

- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y): Dann ist y der niedrigste Nachfolger von x , dessen linkes Kind Vorgänger von x ist

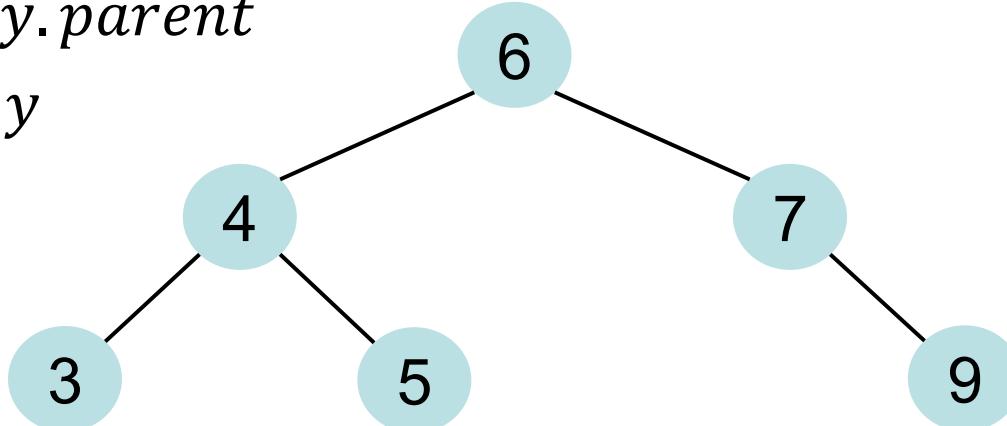


Binäre Suchbäume

Tree-Successor(x)

```
1 if  $x.right \neq NIL$ 
2   return Tree-Minimum( $x.right$ )
3  $y = x.parent$ 
4 while  $y \neq NIL$  and  $x == y.right$ 
5    $x = y$ 
6    $y = y.parent$ 
7 return  $y$ 
```

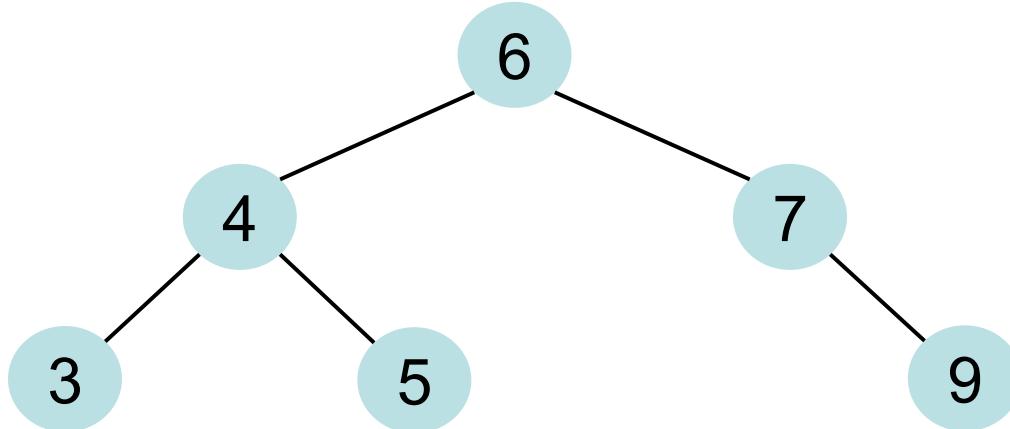
Laufzeit $O(h)$



Binäre Suchbäume

Vorgängersuche:

- Symmetrisch zu Nachfolgersuche
- Daher ebenfalls $O(h)$ Laufzeit



Binäre Suchbäume

Binäre Suchbäume:

- Aufzählen der Elemente mit Inorder–Tree–Walk in $O(n)$ Zeit
- Suche in $O(h)$ Zeit
- Minimum/Maximum in $O(h)$ Zeit

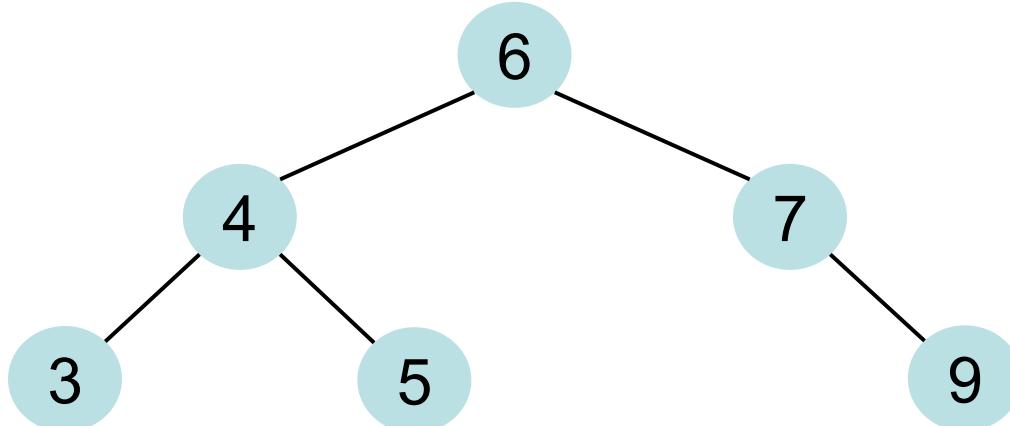
Dynamische Operationen?

- Einfügen und Löschen
- Müssen Suchbaumeigenschaft aufrecht erhalten
- Auswirkung auf Höhe des Baums?

Binäre Suchbäume

Einfügen:

- Ähnlich wie Baumsuche: Finde Blatt, an das neuer Knoten angehängt wird
- Danach wird der NIL-Zeiger durch neues Element ersetzt



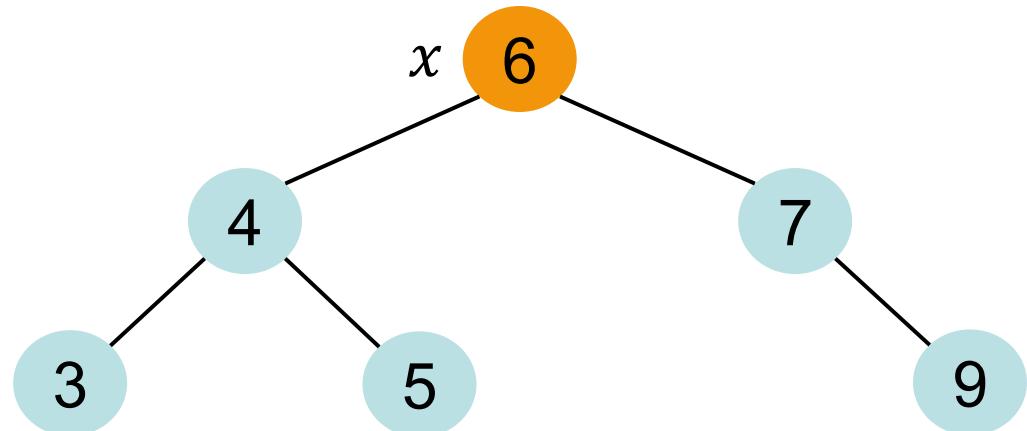
Binäre Suchbäume

Tree–Insert(t, z)

```
1   $y = NIL$ 
2   $x = t.root$ 
3  while  $x \neq NIL$ 
4     $y = x$ 
5    if  $z.key < x.key$ 
6       $x = x.left$ 
7    else  $x = x.right$ 
8   $z.parent = y$ 
9  if  $y == NIL$ 
10    $t.root = z$ 
11  elseif  $z.key < y.key$ 
12    $y.left = z$ 
13  else  $y.right = z$ 
```

y ist Vater des einzufügenden Elements

Einfügen(8)

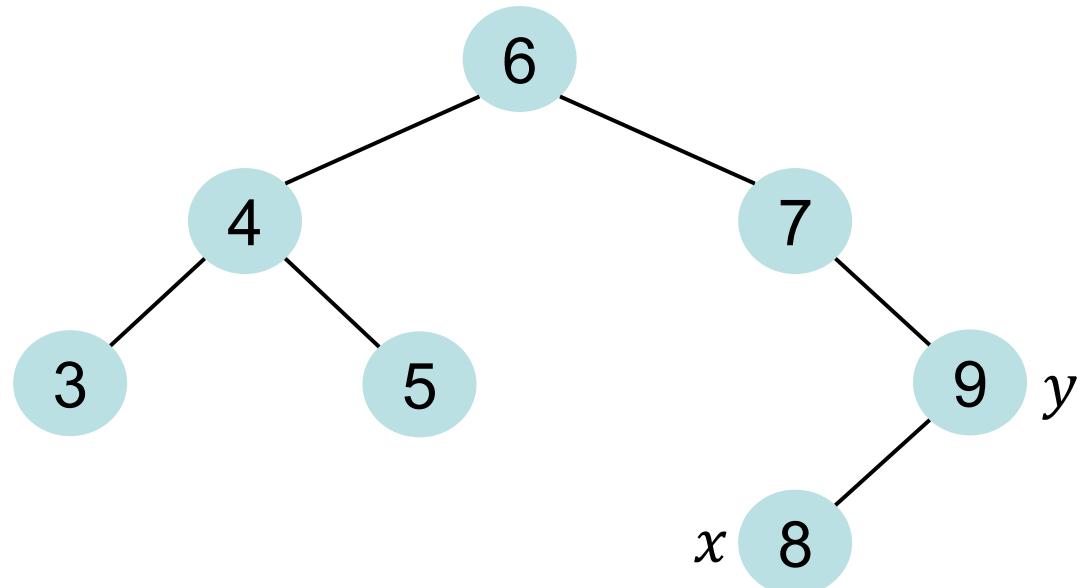


Binäre Suchbäume

Tree-Insert(t, z)

```
1  $y = NIL$ 
2  $x = t.root$ 
3 while  $x \neq NIL$ 
4    $y = x$ 
5   if  $z.key < x.key$ 
6      $x = x.left$ 
7   else  $x = x.right$ 
8    $z.parent = y$ 
9   if  $y == NIL$ 
10     $t.root = z$ 
11   elseif  $z.key < y.key$ 
12     $y.left = z$ 
13   else  $y.right = z$ 
```

Einfügen(8)



Binäre Suchbäume

Tree–Insert(t, z)

```
1  $y = NIL$ 
2  $x = t.root$ 
3 while  $x \neq NIL$ 
4      $y = x$ 
5     if  $z.key < x.key$ 
6          $x = x.left$ 
7     else  $x = x.right$ 
8      $z.parent = y$ 
9     if  $y == NIL$ 
10         $t.root = z$ 
11    elseif  $z.key < y.key$ 
12         $y.left = z$ 
13    else  $y.right = z$ 
```

Laufzeit: $O(h)$

Binäre Suchbäume

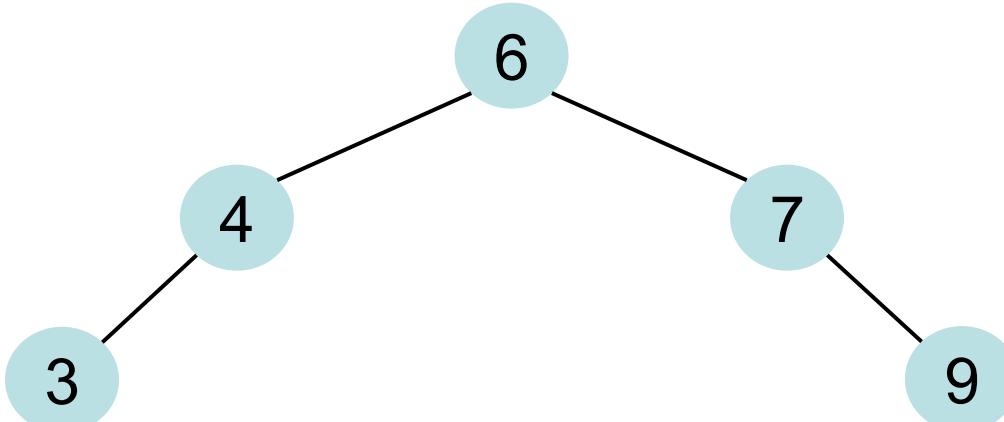
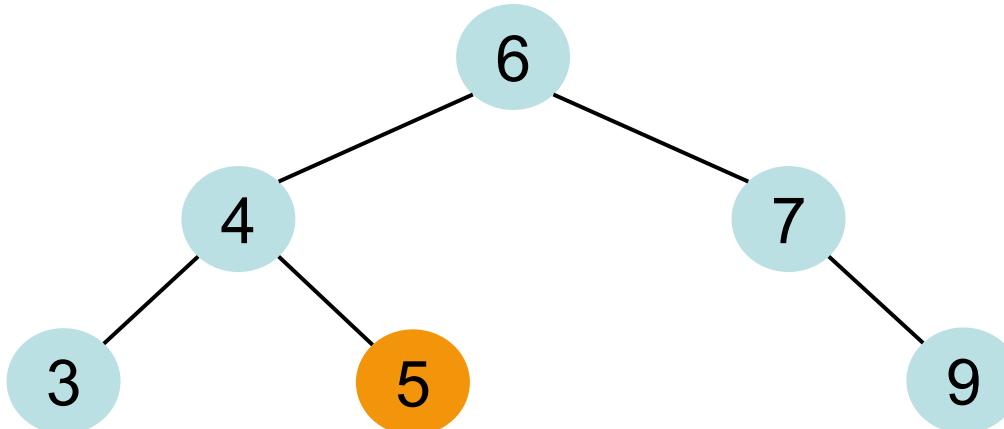
Löschen:

3 unterschiedliche Fälle

- a) zu löschesndes Element z hat keine Kinder
- b) zu löschesndes Element z hat ein Kind
- c) zu löschesndes Element z hat zwei Kinder

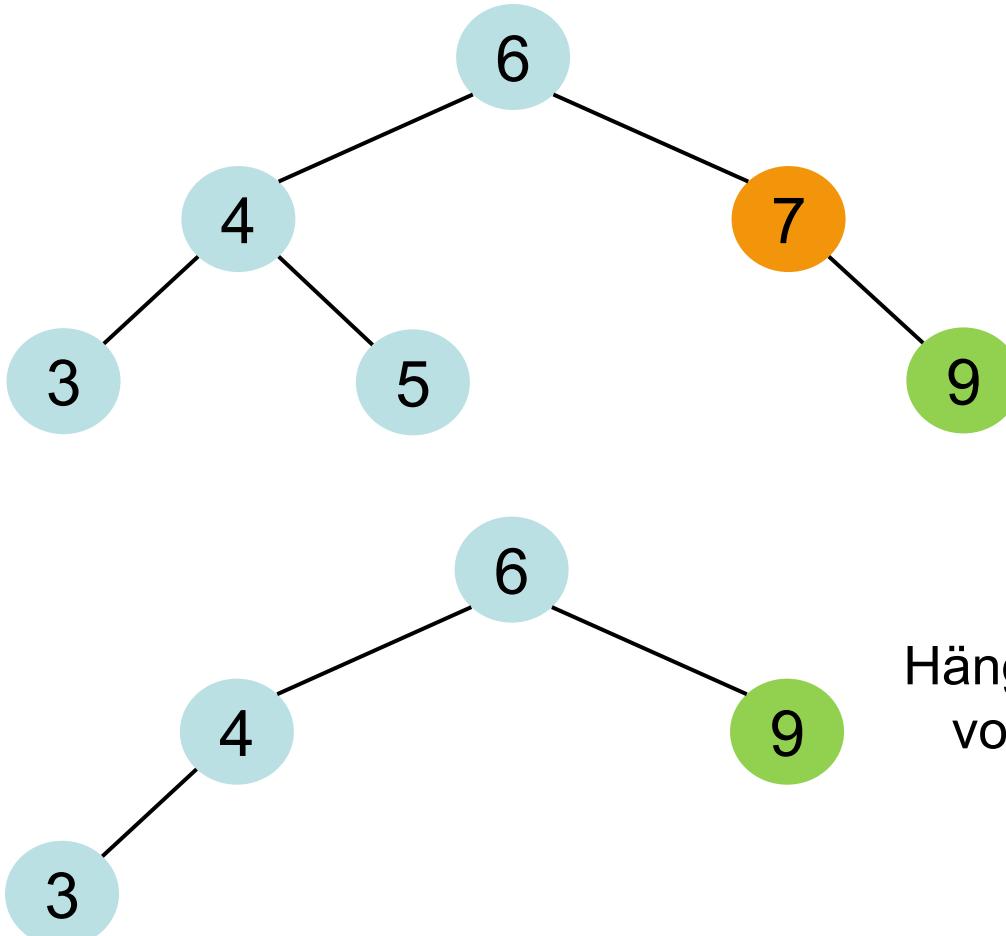
Binäre Suchbäume

Fall a): zu lösches Element z hat keine Kinder



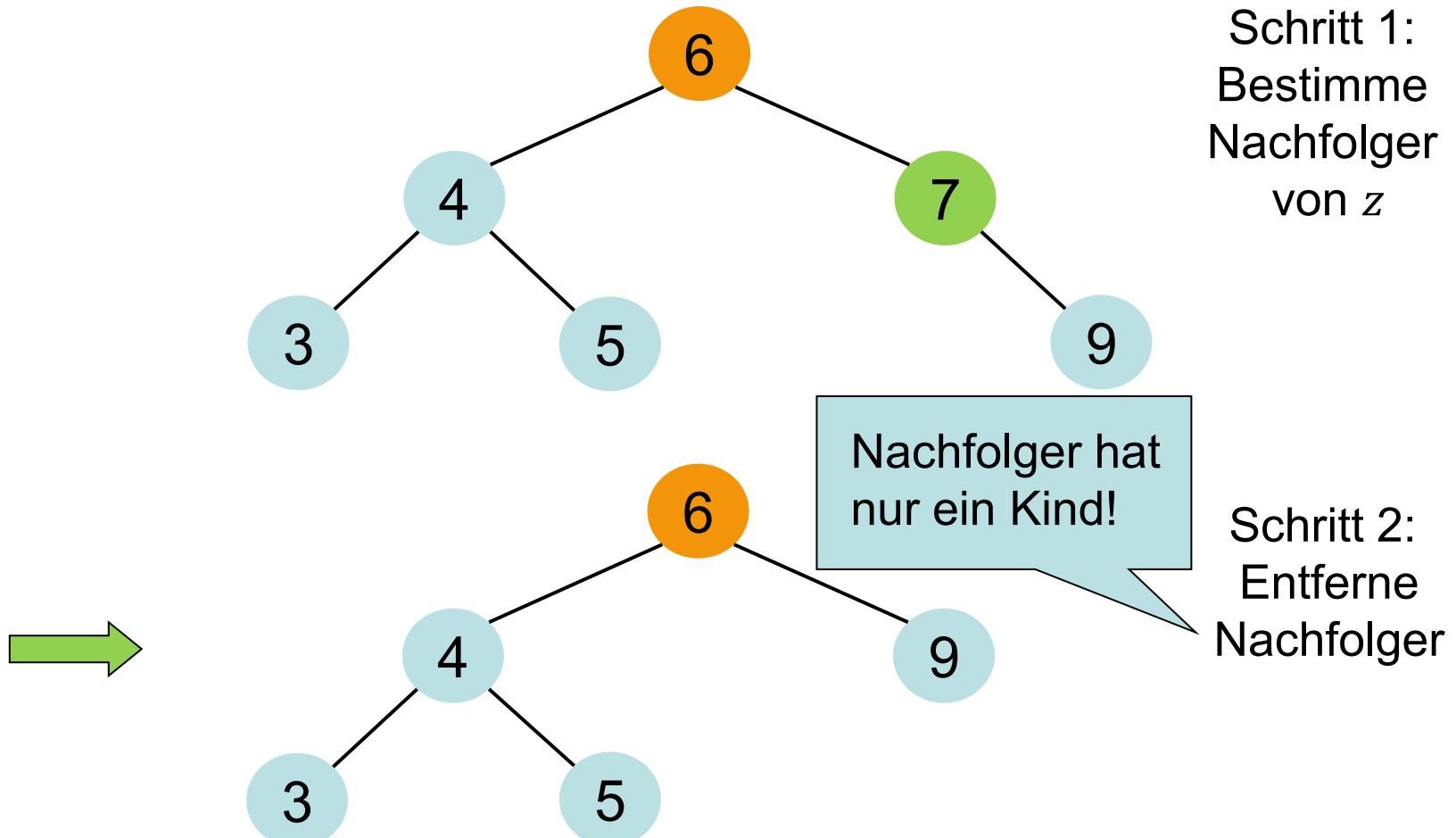
Binäre Suchbäume

Fall b): zu löschendes Element z hat ein Kind



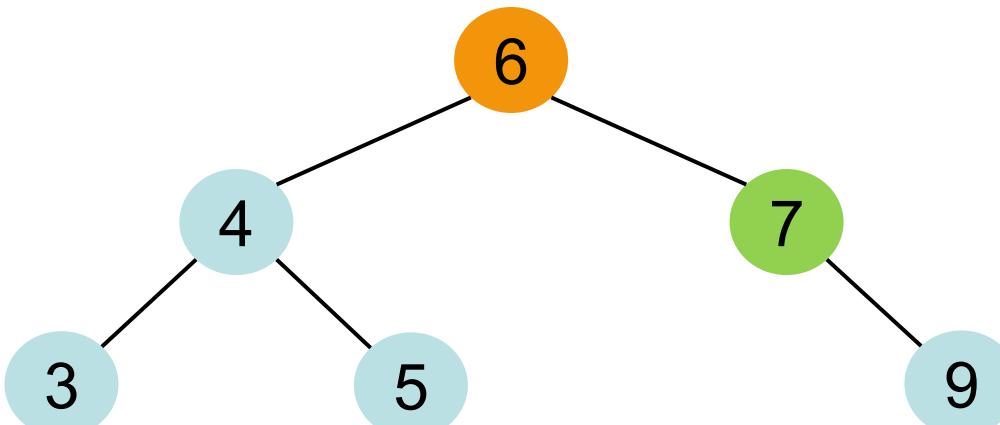
Binäre Suchbäume

Fall c): zu löschesndes Element z hat zwei Kinder

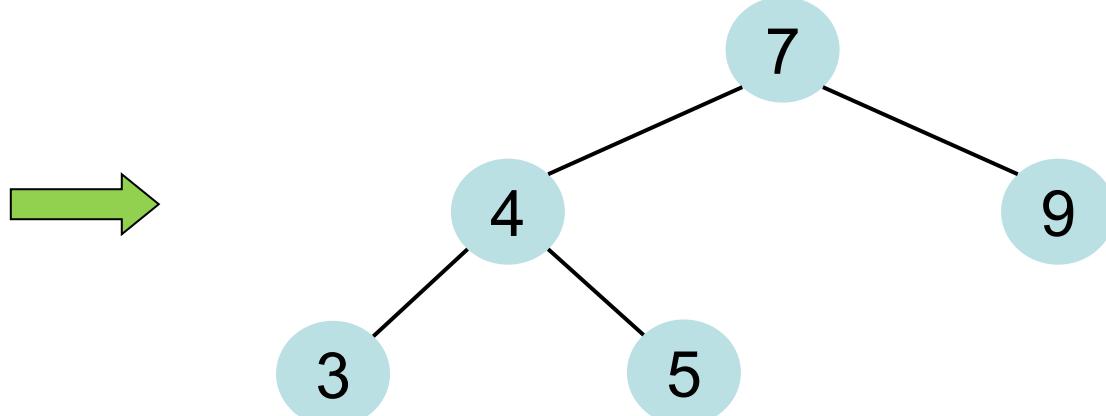


Binäre Suchbäume

Fall c): zu löschesndes Element z hat zwei Kinder



Schritt 1:
Bestimme
Nachfolger
von z



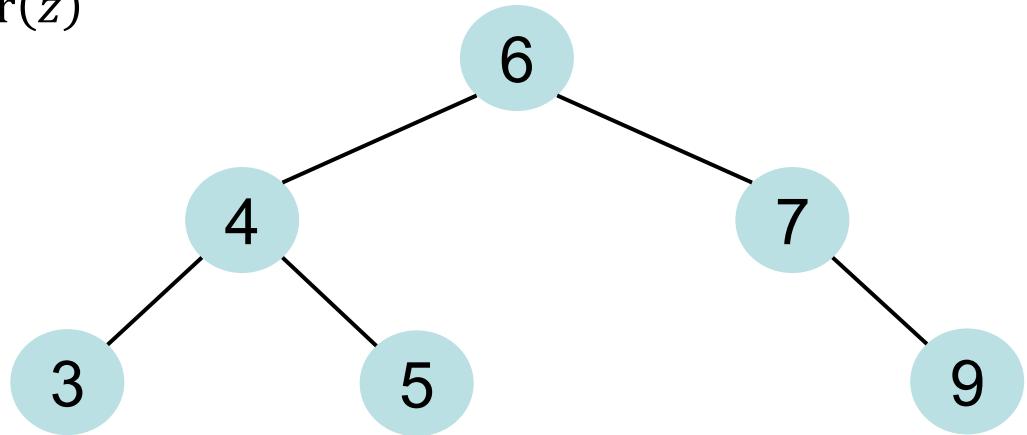
Schritt 2:
Entferne
Nachfolger
Ersetze durch
Nachfolger

Binäre Suchbäume

Tree–Delete(t, z)

```
1 if  $z.left == NIL$  or  $z.right == NIL$ 
2    $y = z$ 
3 else  $y = \text{Tree-Successor}(z)$ 
4 if  $y.left \neq NIL$ 
5    $x = y.left$ 
6 else  $x = y.right$ 
7 if  $x \neq NIL$ 
8    $x.parent = y.parent$ 
9 if  $y.parent == NIL$ 
10    $t.root = x$ 
11 elseif  $y == y.parent.left$ 
12    $y.parent.left = x$ 
13 else  $y.parent.right = x$ 
14 if  $y \neq z$ 
15    $z.key = y.key$ 
```

Löschen(6)

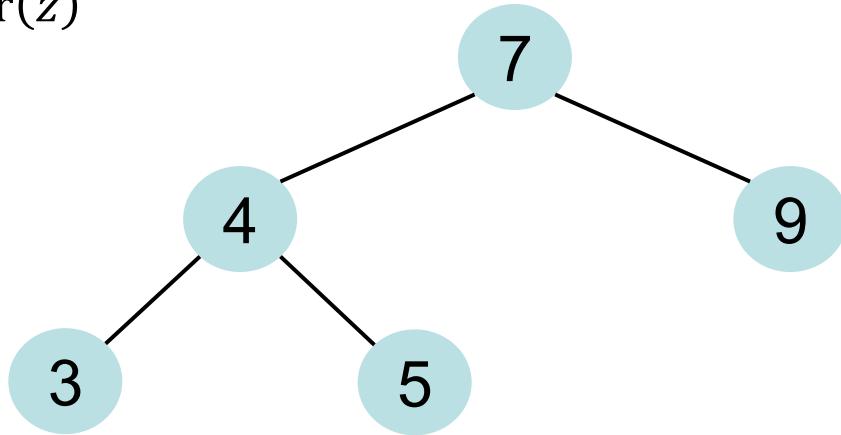


Binäre Suchbäume

Tree–Delete(t, z)

```
1 if  $z.left == NIL$  or  $z.right == NIL$ 
2    $y = z$ 
3 else  $y = \text{Tree-Successor}(z)$ 
4 if  $y.left \neq NIL$ 
5    $x = y.left$ 
6 else  $x = y.right$ 
7 if  $x \neq NIL$ 
8    $x.parent = y.parent$ 
9 if  $y.parent == NIL$ 
10    $t.root = x$ 
11 elseif  $y == y.parent.left$ 
12    $y.parent.left = x$ 
13 else  $y.parent.right = x$ 
14 if  $y \neq z$ 
15    $z.key = y.key$ 
```

Löschen(6)



Laufzeit: $O(h)$

Binäre Suchbäume

Binäre Suchbäume:

- Ausgabe aller Elemente in $O(n)$
- Suche, Minimum, Maximum, Nachfolger in $O(h)$
- Einfügen, Löschen in $O(h)$

Frage:

- Wie kann man eine „kleine“ Höhe unter Einfügen und Löschen garantieren?

10. Balancierte binäre Suchbäume

Problem:

- Gegeben sind n Objekte O_1, \dots, O_n mit zugehörigen Schlüsseln $O_i.key$

Operationen:

- Suche(k): Ausgabe O mit Schlüssel $O.key = k$; NIL , falls kein Objekt mit Schlüssel k in Datenbank
- Einfügen(O): Einfügen von Objekt O in Datenbank
- Löschen(O): Löschen von Objekt O aus der Datenbank

Balancierte binäre Suchbäume

Binäre Suchbäume:

- Ausgabe aller Elemente in $O(n)$
- Suche, Minimum, Maximum, Nachfolger in $O(h)$
- Einfügen, Löschen in $O(h)$

Frage:

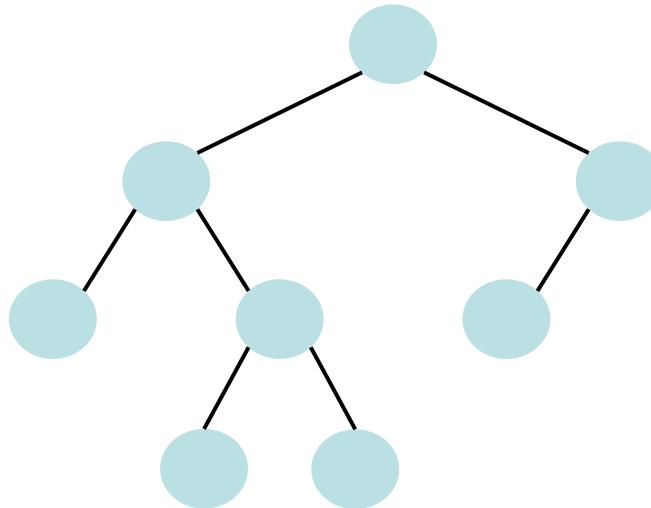
- Wie kann man eine „kleine“ Höhe unter Einfügen und Löschen garantieren?

Balancierte binäre Suchbäume

AVL-Bäume

Ein Baum heißt AVL-Baum, wenn für jeden Knoten gilt:

Die Höhe seines linken und rechten Teilbaums unterscheidet sich höchstens um 1.



Balancierte binäre Suchbäume

Satz 10.1 —

Für jeden AVL-Baum der Höhe $h - 1$ mit n Knoten gilt:

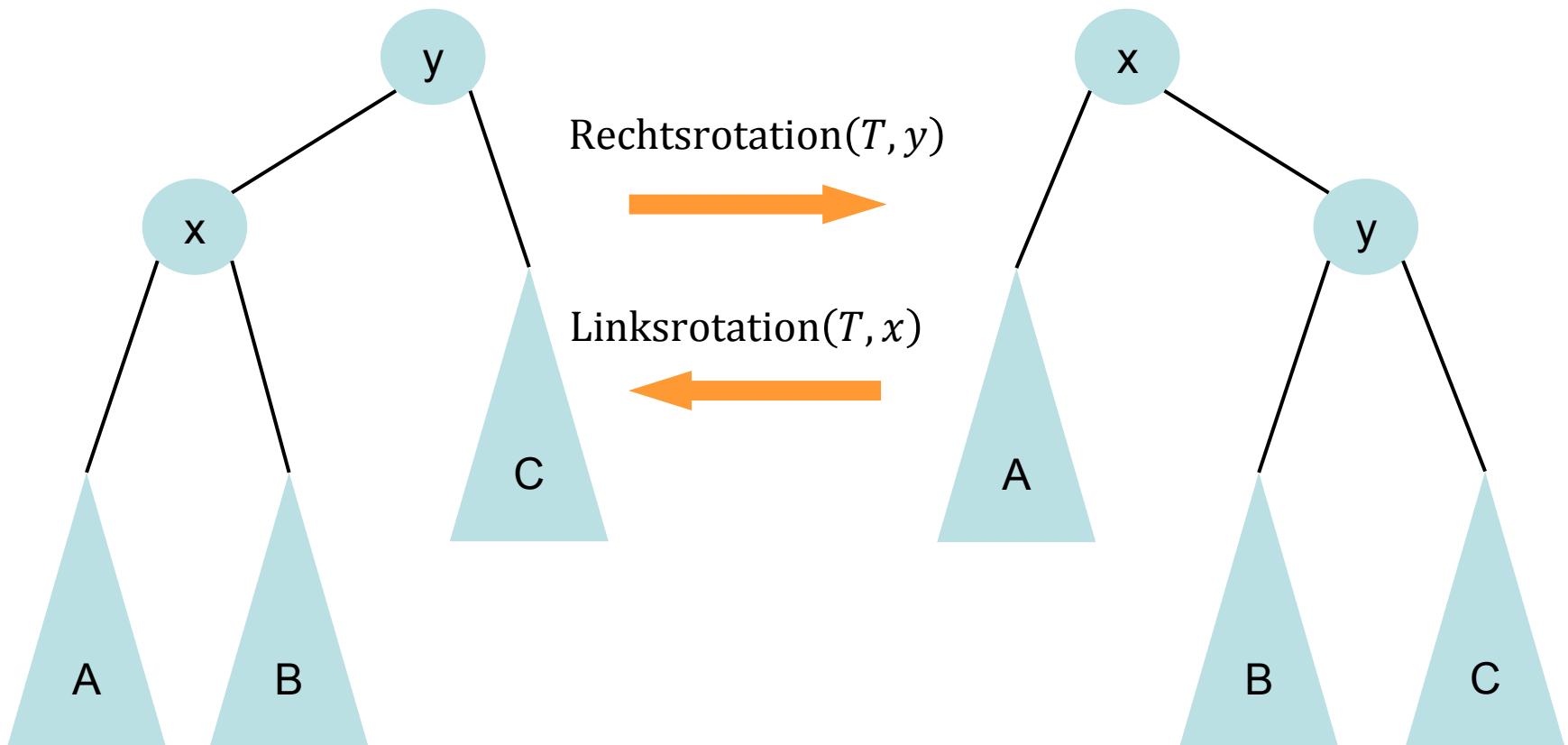
$$\left(\frac{3}{2}\right)^{h-1} \leq n \leq 2^h - 1$$

Korollar 10.2 —

Ein AVL-Baum mit n Knoten hat Höhe $\Theta(\log n)$.

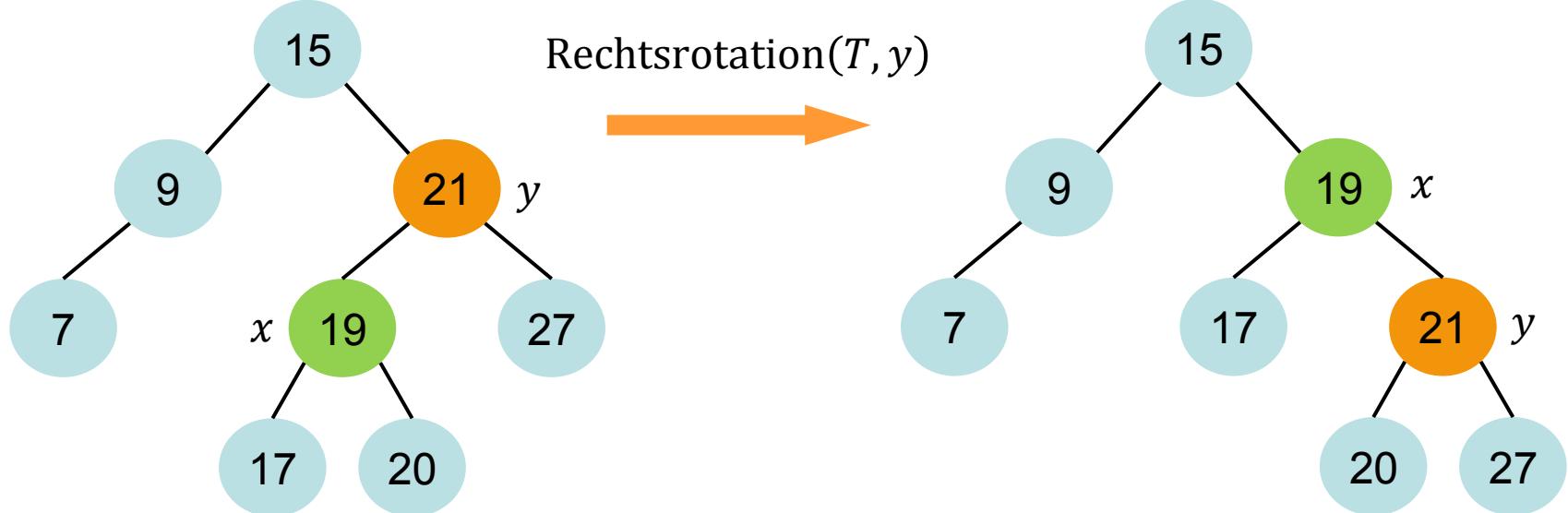
Balancierte binäre Suchbäume

Rotationen:



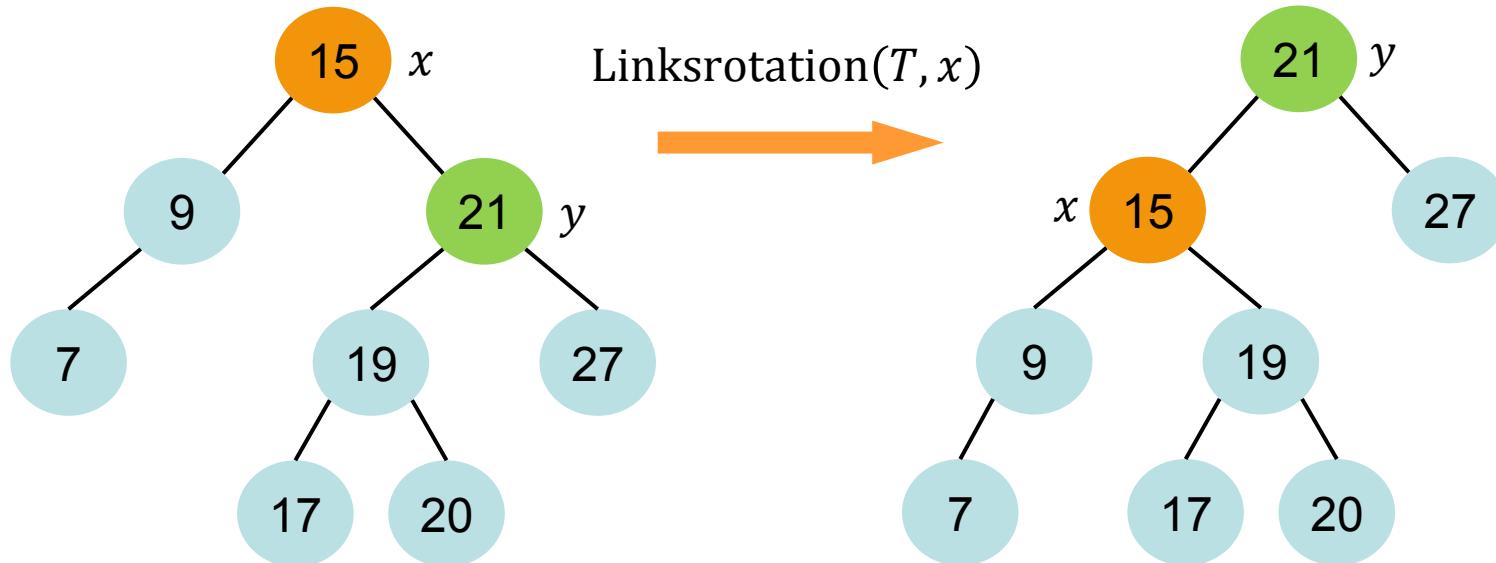
Balancierte binäre Suchbäume

Beispiel 1:



Balancierte binäre Suchbäume

Beispiel 2:



Balancierte binäre Suchbäume

Left–Rotate(t, x)

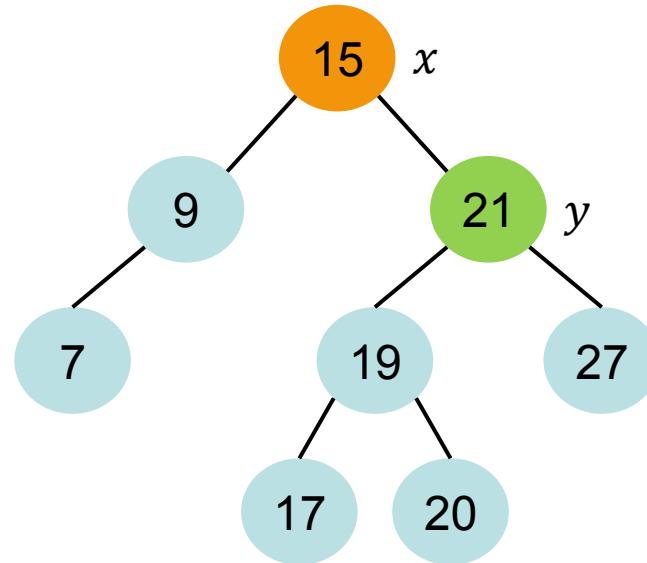
```
1  $y = x.right$ 
2  $x.right = y.left$ 
3 if  $y.left \neq NIL$ 
4    $y.left.parent = x$ 
5  $y.parent = x.parent$ 
6 if  $x.parent == NIL$ 
7    $t.root = y$ 
8 elseif  $x == x.parent.left$ 
9    $x.parent.left = y$ 
10 else  $x.parent.right = y$ 
11  $y.left = x$ 
12  $x.parent = y$ 
13  $x.height = 1 + \max(x.left.height, x.right.height)$ 
14  $y.height = 1 + \max(y.left.height, y.right.height)$ 
```

Zusätzlich müssen die Höhen der (Teil)bäume mit aktualisiert werden.
(die Zeilen 13 und 14 werden im Folgenden nicht betrachtet)

Balancierte binäre Suchbäume

Left–Rotate(t, x)

```
1  $y = x.right$ 
2  $x.right = y.left$ 
3 if  $y.left \neq NIL$ 
4    $y.left.parent = x$ 
5  $y.parent = x.parent$ 
6 if  $x.parent == NIL$ 
7    $t.root = y$ 
8 elseif  $x == x.parent.left$ 
9    $x.parent.left = y$ 
10 else  $x.parent.right = y$ 
11  $y.left = x$ 
12  $x.parent = y$ 
```

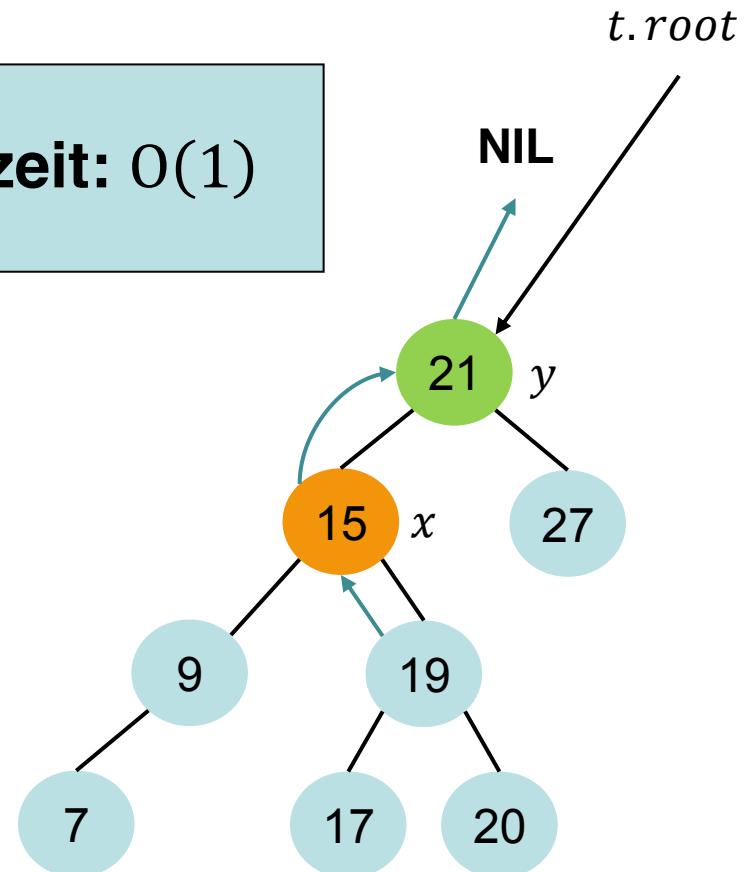


Balancierte binäre Suchbäume

Left-Rotate(t, x)

```
1  $y = x.right$ 
2  $x.right = y.left$ 
3 if  $y.left \neq NIL$ 
4    $y.left.parent = x$ 
5  $y.parent = x.parent$ 
6 if  $x.parent == NIL$ 
7    $t.root = y$ 
8 elseif  $x == x.parent.left$ 
9    $x.parent.left = y$ 
10 else  $x.parent.right = y$ 
11  $y.left = x$ 
12  $x.parent = y$ 
```

Laufzeit: $O(1)$



Balancierte binäre Suchbäume

Dynamische AVL-Bäume

- Operationen Suche, Einfügen, Löschen, Min/Max, Vorgänger/Nachfolger,... wie in der letzten Vorlesung
- Laufzeit $O(h)$ für diese Operationen
- Nur Einfügen/Löschen verändern Struktur des Baums

Nach Korollar 10.2 gilt
 $h = \Theta(\log n)$.

Idee:

- Wir brauchen Prozedur, um AVL-Eigenschaft nach Einfügen/Löschen wiederherzustellen.

Balancierte binäre Suchbäume

Definition 10.3

Ein Baum heißt beinahe-AVL-Baum, wenn die AVL-Eigenschaft in jedem Knoten außer der Wurzel erfüllt ist und die Höhen der Unterbäume der Wurzel sich um höchstens 2 unterscheiden.

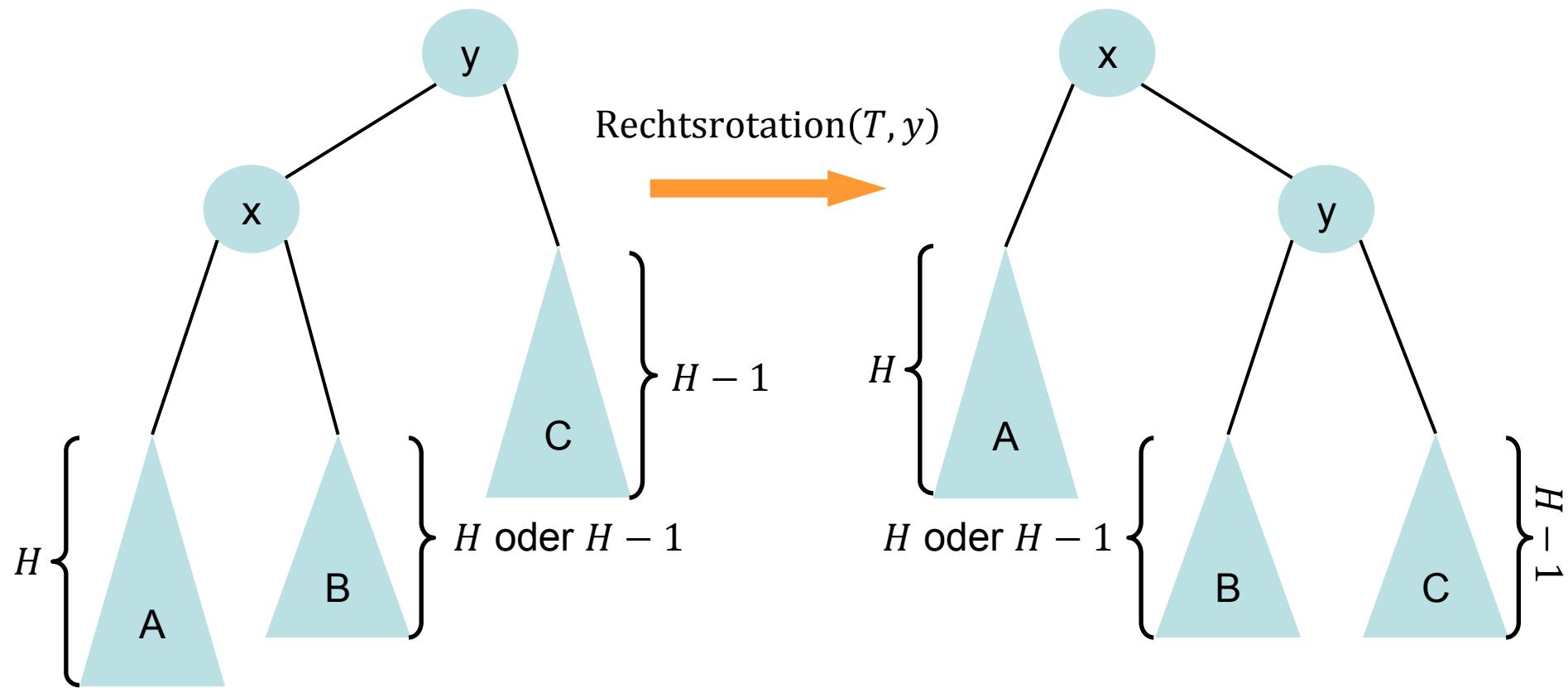
Balancierte binäre Suchbäume

Problem:

- Umformen eines beinahe-AVL-Baums in einen AVL-Baum mit Hilfe von Rotationen
- O.b.d.A.: Linker Teilbaum der Wurzel höher als der rechte

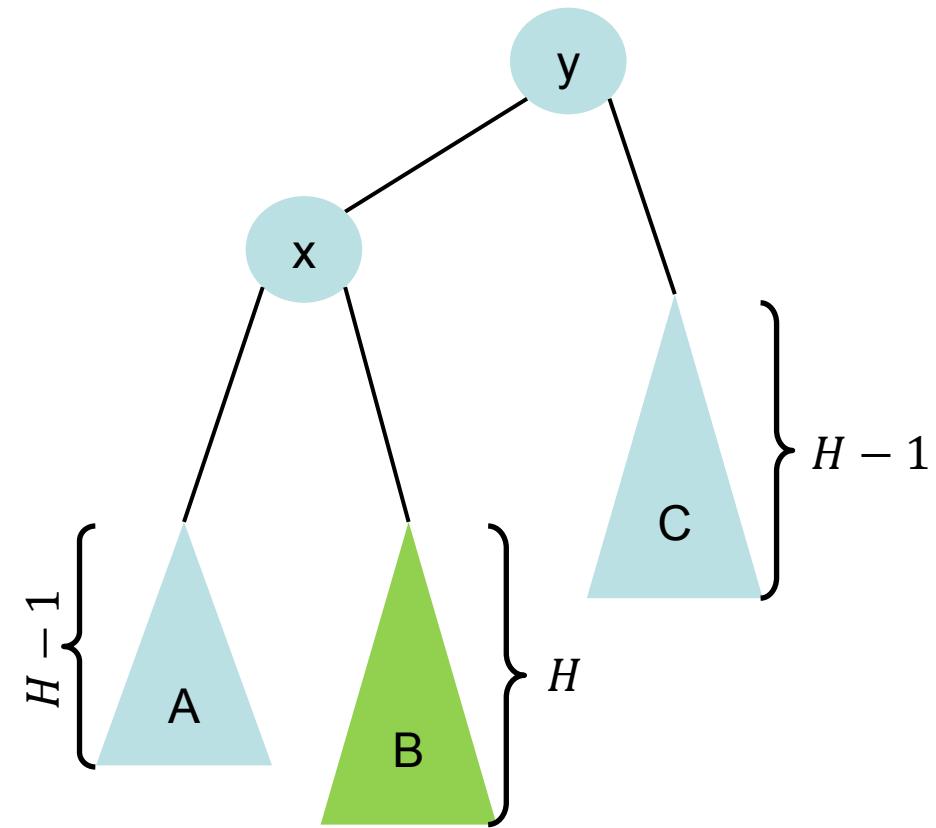
Balancierte binäre Suchbäume

Fall 1:



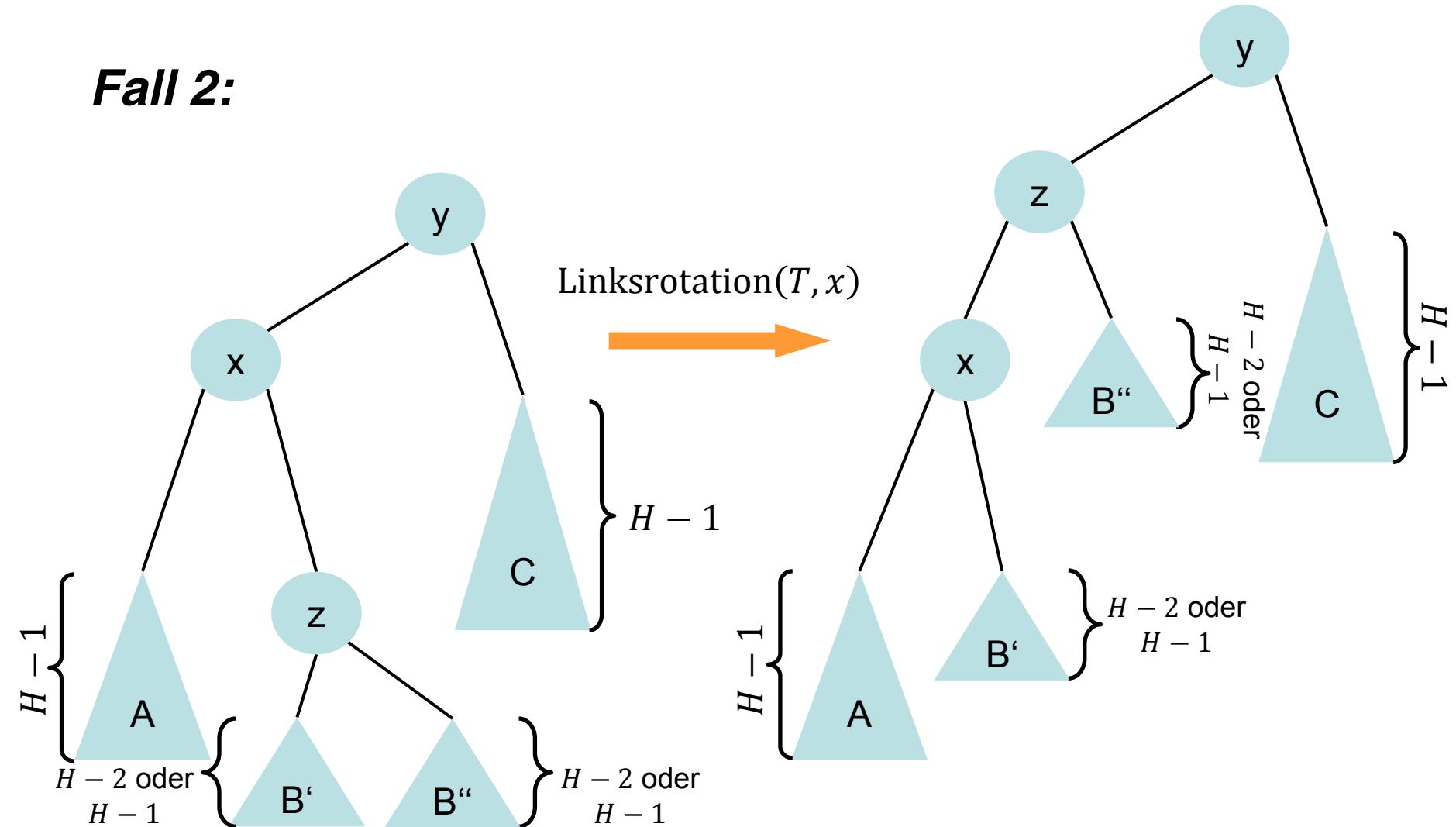
Balancierte binäre Suchbäume

Fall 2:



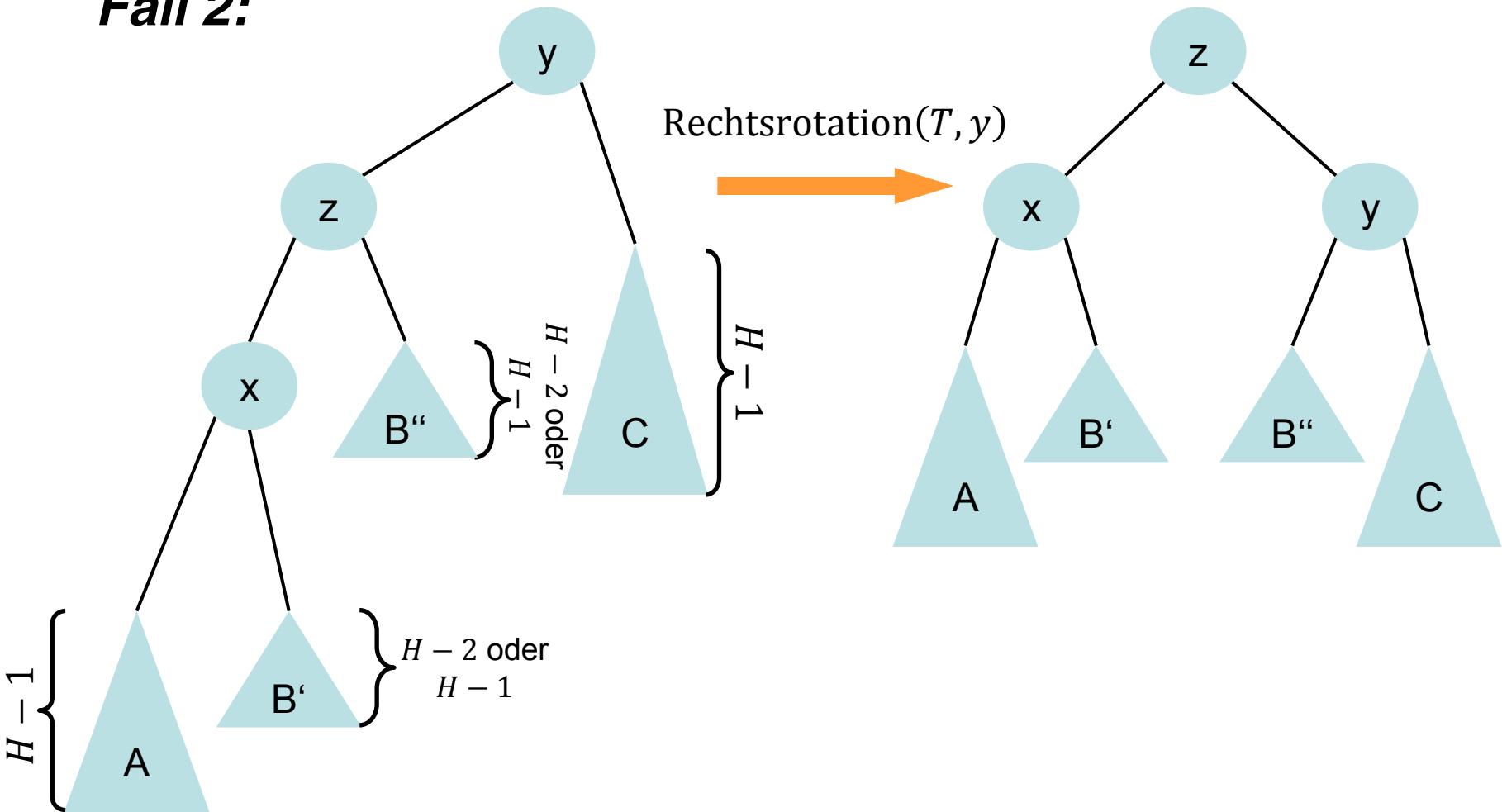
Balancierte binäre Suchbäume

Fall 2:



Balancierte binäre Suchbäume

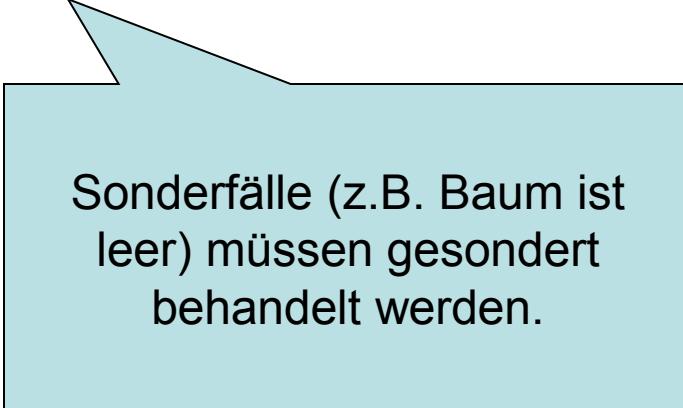
Fall 2:



Balancierte binäre Suchbäume

Balance(t)

```
1 if  $t.left.height > t.right.height + 1$ 
2   if  $t.left.left.height < t.left.right.height$ 
3     Left-Rotate( $t.left$ )
4     Right-Rotate( $t$ )
5   elseif  $t.right.height > t.left.height + 1$ 
6     if  $t.right.right.height < t.right.left.height$ 
7       Right-Rotate( $t.right$ )
8       Left-Rotate( $t$ )
```

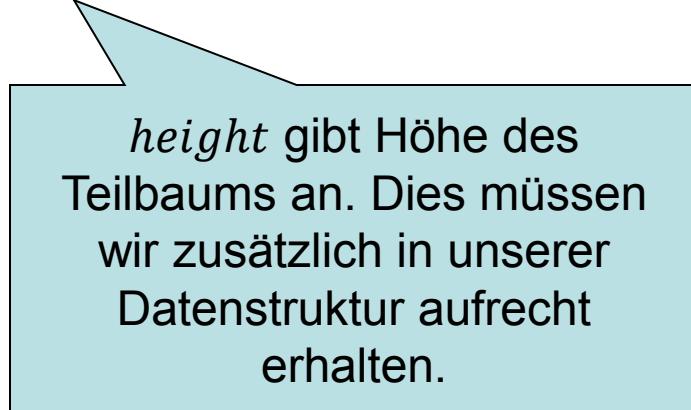


Sonderfälle (z.B. Baum ist leer) müssen gesondert behandelt werden.

Balancierte binäre Suchbäume

Balance(t)

- 1 **if** $t.left.height > t.right.height + 1$
- 2 **if** $t.left.left.height < t.left.right.height$
- 3 Left–Rotate($t.left$)
- 4 Right–Rotate(t)
- 5 **elseif** $t.right.height > t.left.height + 1$
- 6 **if** $t.right.right.height < t.right.left.height$
- 7 Right–Rotate($t.right$)
- 8 Left–Rotate(t)



height gibt Höhe des Teilbaums an. Dies müssen wir zusätzlich in unserer Datenstruktur aufrecht erhalten.

Balancierte binäre Suchbäume

Balance(t)

```
1 if  $t.left.height > t.right.height + 1$ 
2   if  $t.left.left.height < t.left.right.height$ 
3     Left–Rotate( $t.left$ )
4     Right–Rotate( $t$ )
5   elseif  $t.right.height > t.left.height + 1$ 
6     if  $t.right.right.height < t.right.left.height$ 
7       Right–Rotate( $t.right$ )
8       Left–Rotate( $t$ )
```

Laufzeit $O(1)$

Balancierte binäre Suchbäume

Kurze Zusammenfassung:

- Wir können aus einem beinahe-AVL-Baum mit Hilfe von maximal 2 Rotationen einen AVL-Baum machen
- Dabei erhöht sich die Höhe des Baums nicht

Einfügen:

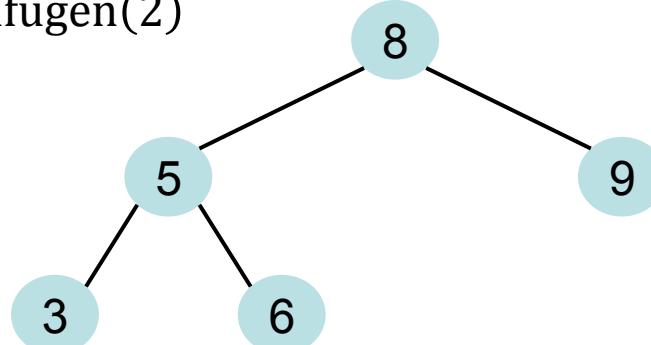
- Wir fügen ein wie früher
- Dann laufen wir den Pfad zur Wurzel zurück
- An jedem Knoten balancieren wir, falls der Unterbaum ein beinahe-AVL-Baum ist

Balancierte binäre Suchbäume

AVL-Insert(t, x)

```
1 if  $t == NIL$ 
2    $t = new node(x)$ 
3   return
4 elseif  $x.key < t.key$ 
5   AVL-Insert( $t.left, x$ )
6 elseif  $x.key > t.key$ 
7   AVL-Insert( $t.right, x$ )
8 else return           // Schlüssel schon vorhanden
9    $t.height = 1 + \max(t.left.height, t.right.height)$ 
10  Balance( $t$ )
```

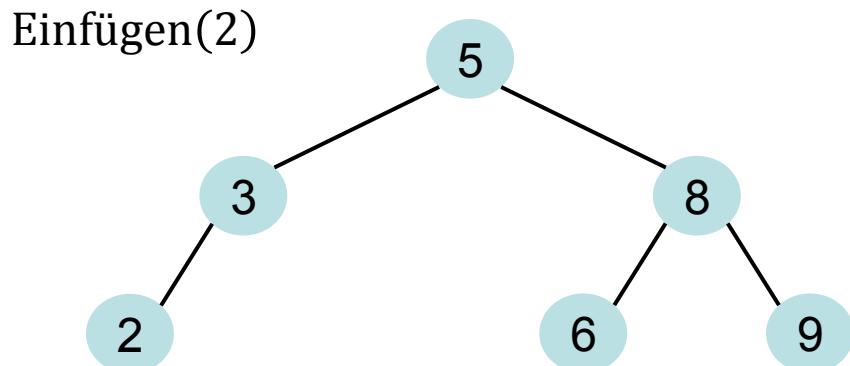
Einfügen(2)



Balancierte binäre Suchbäume

AVL-Insert(t, x)

```
1 if  $t == NIL$ 
2    $t = new node(x)$ 
3   return
4 elseif  $x.key < t.key$ 
5   AVL-Insert( $t.left, x$ )
6 elseif  $x.key > t.key$ 
7   AVL-Insert( $t.right, x$ )
8 else return           // Schlüssel schon vorhanden
9    $t.height = 1 + \max(t.left.height, t.right.height)$ 
10  Balance( $t$ )
```



Laufzeit
 $O(h) = O(\log n)$

Balancierte binäre Suchbäume

Korrektheit (Skizze):

- Induktion über die Länge des Suchpfads
(startend vom eingefügten Blatt):
- Einfügen in einen leeren Baum funktioniert
- ist t nicht leer, dann wird x in einen der beiden Teilbäume eingefügt
- Nach I.A. sind diese Teilbäume bereits AVL-Bäume und unterscheiden sich nur um maximal 2 in ihrer Höhe
- Falls die AVL-Eigenschaft nicht erhalten bleibt, dann wird Balance ausgeführt.

Balancierte binäre Suchbäume

AVL–Delete(t, k)

```
1 if  $k < t.key$ 
2   AVL–Delete( $t.left, k$ )
3 elseif  $k > t.key$ 
4   AVL–Delete( $t.right, k$ )
5 elseif  $t == NIL$ 
6   return //  $k$  nicht im Baum
7 elseif  $t.left == NIL$ 
8    $t = t.right$ 
9 elseif  $t.right == NIL$ 
10   $t = t.left$ 
11 else
12    $u = \text{MaximumSuche}(t.left)$ 
13    $t.key = u.key$ 
14   AVL–Delete( $t.left, u.key$ )
15   Balance( $t$ )
```



heights müssen wir zusätzlich in unserer Datenstruktur aufrecht erhalten

Balancierte binäre Suchbäume

Korrektheit:

ähnlich wie beim Einfügen

Satz 10.4

Mit Hilfe von AVL-Bäumen kann man Suche, Einfügen, Löschen, Minimum und Maximum in einer Menge von n Zahlen in $\Theta(\log n)$ Laufzeit durchführen.

Balancierte binäre Suchbäume

Zusammenfassung und Ausblick:

- Effiziente Datenstruktur für das Datenbank Problem mit Hilfe von Suchbäumen
- Kann man eine bessere Datenstruktur finden?
- Was muss man ggf. anders machen?
(untere Schranke für vergleichsbasierte Strukturen)

11. Hashing

AVL-Bäume:

- Ausgabe aller Elemente in $O(n)$
- Suche, Minimum, Maximum, Nachfolger in $O(\log n)$
- Einfügen, Löschen in $O(\log n)$

Frage:

- Kann man Einfügen, Löschen und Suchen in $O(1)$ Zeit?

11. Hashing

- **Hashing:**
 - einfache Methode, um Wörterbücher zu implementieren
 - d.h. Hashing unterstützt die Operationen Search, Insert, Delete.
- Worst-case Zeit für Search: $\Theta(n)$.
- In der Praxis jedoch sehr gut.
- Unter gewissen Annahmen erwartete Suchzeit $O(1)$.
- **Hashing:**
 - Verallgemeinerung von direkter Adressierung durch Arrays

Direkte Adressierung mit Arrays

- Schlüssel für Objekte der dynamischen Menge aus $U := \{0, \dots, m - 1\}$. Menge U wird **Universum** genannt.
- Nehmen an, dass alle Objekte unterschiedliche Schlüssel haben.
- Legen Array $T[0, \dots, m - 1]$ an. Position k in T reserviert für Objekt mit Schlüssel k .
- Dabei verweist $T[k]$ auf Objekt mit Schlüssel k . Falls kein Objekt mit Schlüssel k in Struktur, so gilt $T[k] = NIL$.

Operationen bei direkter Adressierung (1)

DIRECT-ADDRESS-SEARCH(T, k)

1 return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

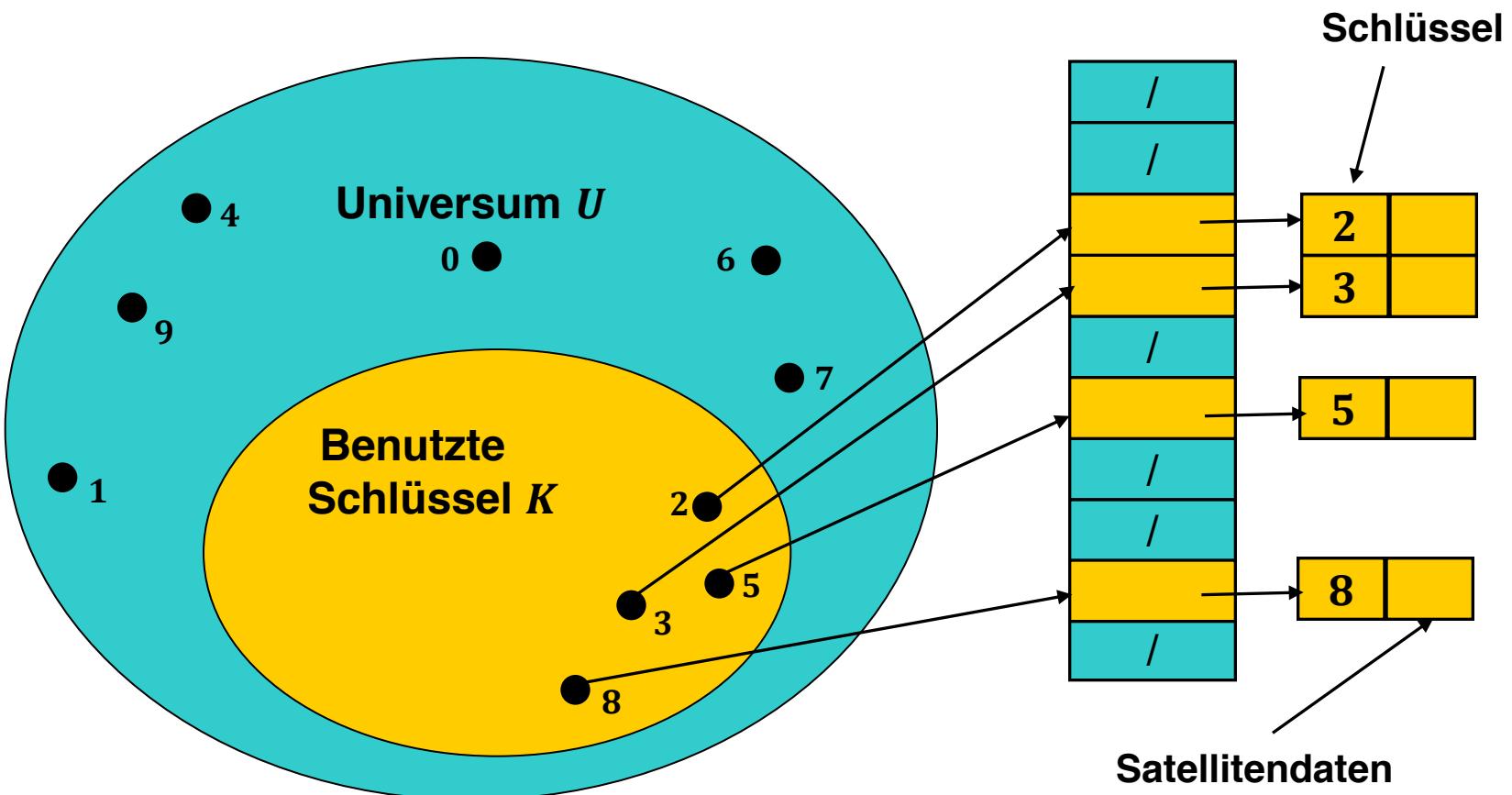
DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = NIL$

Operationen bei direkter Adressierung (2)

- Laufzeit jeweils $O(1)$.
- Direkte Adressierung nicht möglich, wenn Universum U sehr groß ist.
- Speicherineffizient ($\Theta(|U|)$), wenn Menge der aktuell zu speichernden Schlüssel deutlich kleiner als U ist.

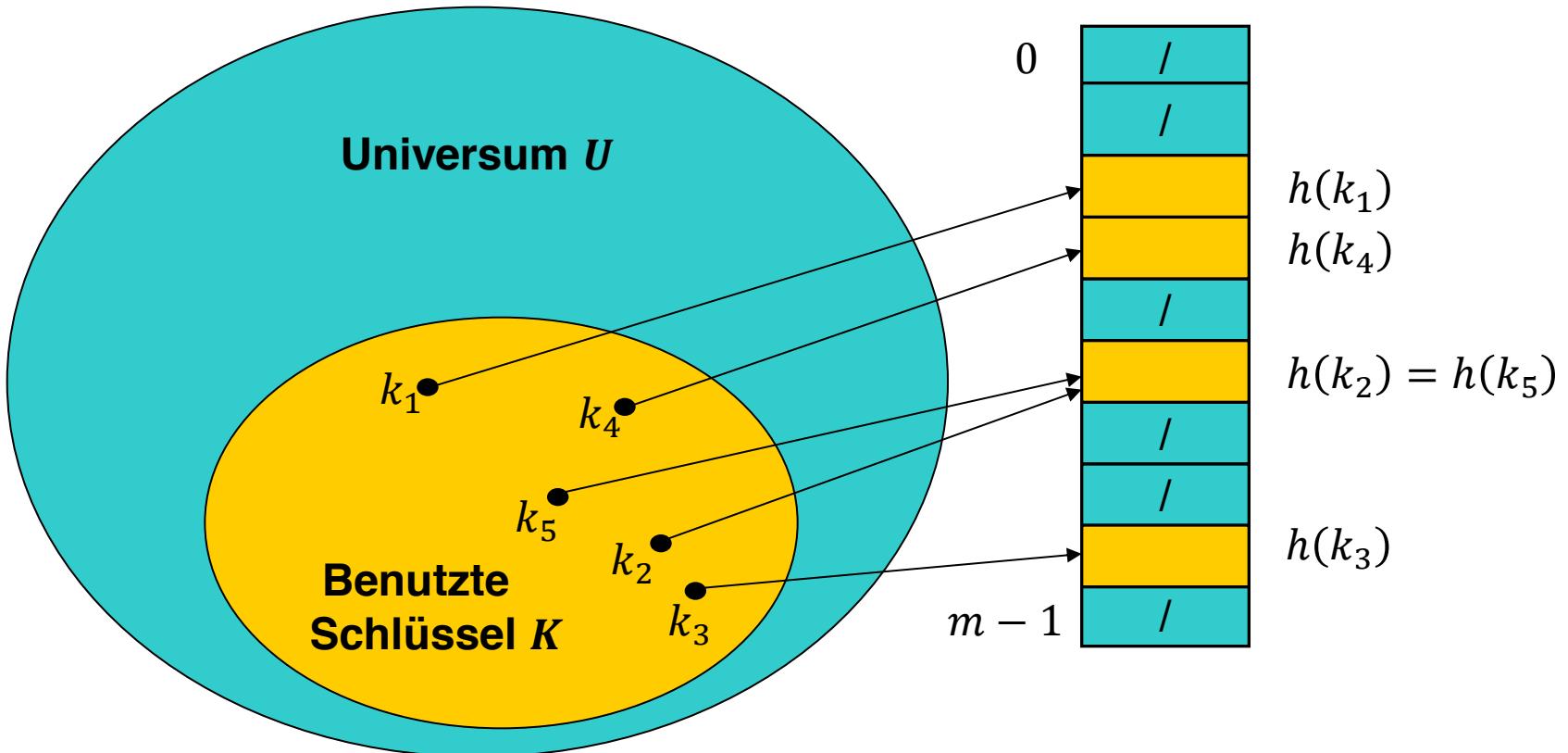
Direkte Adressierung - Illustration



Hashing – Idee (1)

- Nehmen an, dass die Menge K der zu speichernden Schlüssel immer kleiner als das Universum U ist.
- Hashing hat dann Speicherbedarf $\Theta(|K|)$. Zeit für Insert wie bei direkter Adressierung $O(1)$.
- Zeit für Search und Delete ebenfalls $O(1)$, aber nur im Durchschnitt bei geeigneten Annahmen.
- Legen Array $T[0, \dots, m - 1]$ an, wobei $m < |U|$. Nennen T **Hashtabelle**.
- Benutzen **Hashfunktion** $h: U \rightarrow \{0, \dots, m - 1\}$.
- Verweis auf Objekt mit Schlüssel k in $T[h(k)]$.

Hashing – Illustration



Hashing – Idee (2)

- Da $m < |U|$, gibt es Objekte, deren Schlüssel auf den selben Wert gehasht werden, d.h., es gibt Schlüssel k_1, k_2 mit $k_1 \neq k_2$ und $h(k_1) = h(k_2)$.
Dieses wird **Kollision** genannt.
- Verwaltung von Kollision erfolgt durch **Verkettung**.
- Speichern Objekte, deren Schlüssel auf den Hashwert h abgebildet werden, in einer doppelt verketteten Liste L_h . Dann verweist $T[h]$ auf den Beginn der Liste, speichert also $head[L_h]$.
- Insert, Delete, Search jetzt mit Listenoperationen.

Operationen bei Kollisionsverwaltung

CHAINED–HASH–INSERT(T, x)

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

CHAINED–HASH–SEARCH(T, k)

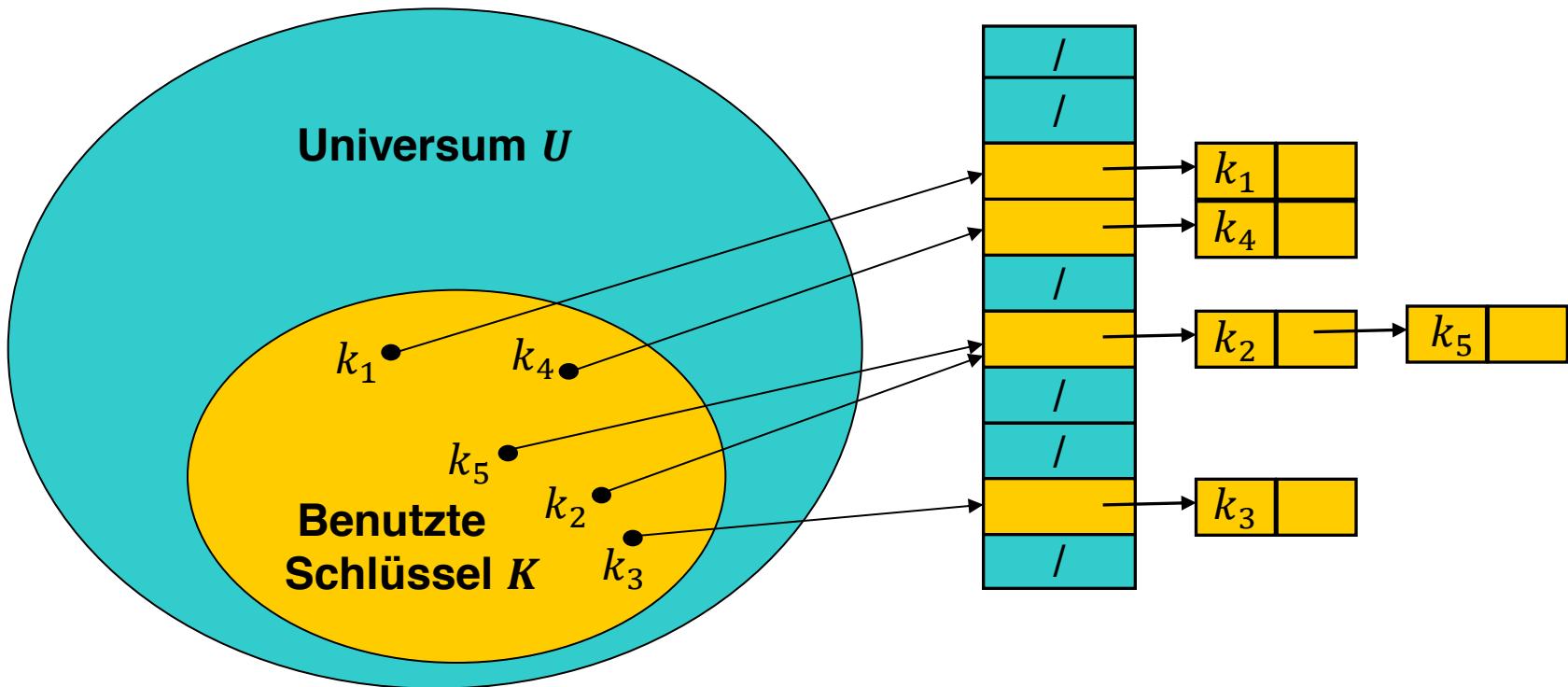
1 search for an element with key k in list $T[h(k)]$

CHAINED–HASH–DELETE(T, x)

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

- Laufzeit für Insert $O(1)$.
- Laufzeit für Delete, Search proportional zur Länge von $T[h(k)]$.

Hashing mit Listen - Illustration



Analyse von Hashing mit Listen (1)

- Sei m die Größe der Hashtabelle T , n Anzahl der gespeicherten Objekte.
Dann heißt $\alpha := \frac{n}{m}$ der Lastfaktor von T .
- Genauer: α ist die durchschnittliche Anzahl von Elementen in einer verketteten Liste.
- Werden alle Objekte auf denselben Wert gehasht, so benötigt Suche bei n Elementen Zeit $\Theta(n)$. Das selbe Verhalten wie verkettete Listen.
- Im Durchschnitt aber ist Suche deutlich besser, falls eine gute Hashfunktion h benutzt wird.
- Gute Hashfunktion streut Werte wie zufällige Funktion.

Einfaches uniformes Hashing

Definition 11.1

Wenn wir annehmen, dass bei jedem neuen Objekt mit Schlüssel k die Hashfunktion h den Schlüssel k gleichverteilt und unabhängig von anderen bereits festgelegten Hashwerten auf die möglichen Hashwerte abbildet, so sprechen wir von einfachem uniformem Hashing.

Analyse von einfachem uniformem Hashing

- Für $j = 0, 1, \dots, m - 1$ sei n_j die Größe der Liste $T[j]$.
- Dann gilt: $n = n_0 + n_1 + \dots + n_{m-1}$.
- Erwartungswert $E[n_j]$ ist $\alpha = \frac{n}{m}$.
- Nehmen zusätzlich an, dass die Hashfunktion h in Zeit $O(1)$ ausgewertet werden kann.

Analyse von einfachem uniformem Hashing

Satz 11.2

Bei einfachem uniformem Hashing benötigt eine **nicht** erfolgreiche Summe im Erwartungswert Zeit $\Theta(1 + \alpha)$.

Satz 11.3

Bei einfachem uniformem Hashing benötigt eine erfolgreiche Summe im Erwartungswert und bei zufälligem Suchobjekt Zeit $\Theta(1 + \alpha)$.

In beiden Fällen:

Ist $n = O(m)$, so ist erwartete Laufzeit $O(1)$.

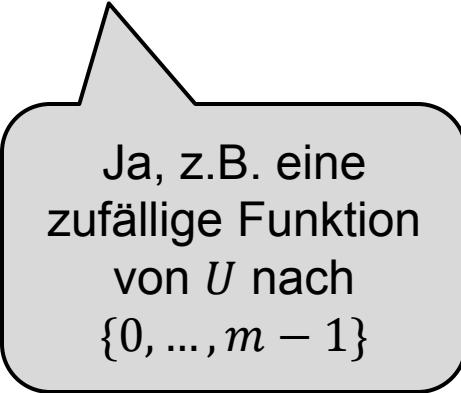
Einfaches uniformes Hashing – Realisierung

Interpretation:

Ist die Größe der Hashtabelle proportional zur Anzahl gespeicherter Elemente, dann ist die durchschnittliche Laufzeit $O(1)$.

Fragen:

- Gibt es Hashfunktionen, die die Annahme einfaches uniformes Hashing erfüllen?



Ja, z.B. eine
zufällige Funktion
von U nach
 $\{0, \dots, m - 1\}$

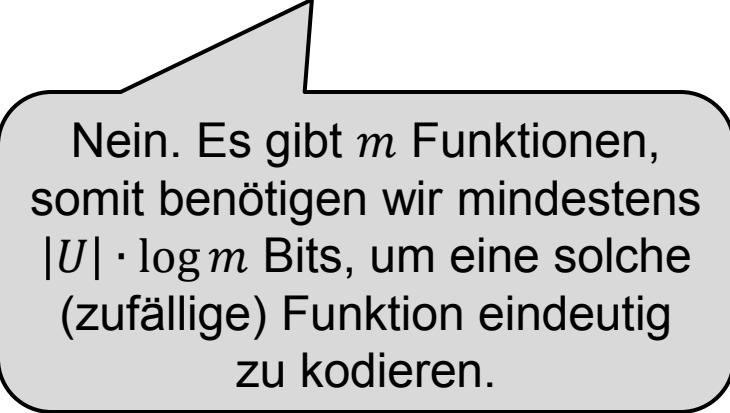
Einfaches uniformes Hashing – Realisierung

Interpretation:

Ist die Größe der Hashtabelle proportional zur Anzahl gespeicherter Elemente, dann ist die durchschnittliche Laufzeit $O(1)$.

Fragen:

- Gibt es Hashfunktionen, die die Annahme einfaches uniformes Hashing erfüllen?
- Kann man diese effizient konstruieren und abspeichern?



Nein. Es gibt m Funktionen, somit benötigen wir mindestens $|U| \cdot \log m$ Bits, um eine solche (zufällige) Funktion eindeutig zu kodieren.

Einfaches uniformes Hashing – Realisierung

Interpretation:

Ist die Größe der Hashtabelle proportional zur Anzahl gespeicherter Elemente, dann ist die durchschnittliche Laufzeit $O(1)$.

Fragen:

- Gibt es Hashfunktionen, die die Annahme einfaches uniformes Hashing erfüllen?
- Kann man diese effizient konstruieren und abspeichern?
- Wie konstruiert man gute Hash Funktionen?



Man nimmt sich eine kleinere Klasse von Funktionen, mit der Eigenschaft, dass eine zufällige Funktion aus dieser Klasse sich ähnlich verhält wie eine zufällige Funktion von U nach $\{0, \dots, m - 1\}$.

Einfaches uniformes Hashing – Realisierung

Einfaches uniformes Hashing kann realisiert werden, indem:

1. jedem hinzugefügten Objekt beim Einfügen ein Schlüssel zufällig gleichverteilt zugewiesen wird.
2. für die Hashfunktion h und für jeden Hashwert $w \in \{0, 1, \dots, m - 1\}$ gilt:

Die Anzahl der Schlüssel k ,
die auf w gehasht werden, ist genau $\frac{|U|}{m}$.

Hashfunktion mit 2. Eigenschaft ist z.B. $h(k) = k \bmod m$, falls $|U|$ Vielfaches von m ist.

Einfaches uniformes Hashing – Realisierung

- 1. Eigenschaft ist sehr unrealistisch.
- Stattdessen Hashfunktionen, die Schlüssel regelmäßig streuen und Regelmäßigkeiten in den Daten umgehen.

Beispiel:

Schlüssel sind Eigennamen. Alphabetisch nahe Eigennamen sollten weit auseinander liegende Hashwerte erhalten.

Beispiele für Hashfunktionen

1. $h(k) = k \bmod m$
(Divisionsmethode)
 2. $h(k) = \lfloor m(kA \bmod 1) \rfloor$, mit $kA \bmod 1 = kA - \lfloor kA \rfloor$
(Multiplikationsmethode)
- Hashing mit Divisionsmethode schnell
 - pro Hashwert eine Division
 - Wert m sollte keine Zweierpotenz sein. Andernfalls besteht Hashwert nur aus unteren Bits des Schlüssels.
 - Gute Wahl ist Primzahl m , die nicht sehr nah an Zweierpotenzen liegt.

Multiplikationsmethode (1)

Methode: $h(k) = \lfloor m(k \cdot A \bmod 1) \rfloor$.

Parameterwahl:

1. Wahl von m irrelevant, häufig $m = 2^p$.
2. Wahl von A wichtig, häufig gewählt als
gute Approximation zum **goldenen Schnitt** $\frac{\sqrt{5}-1}{2}$.

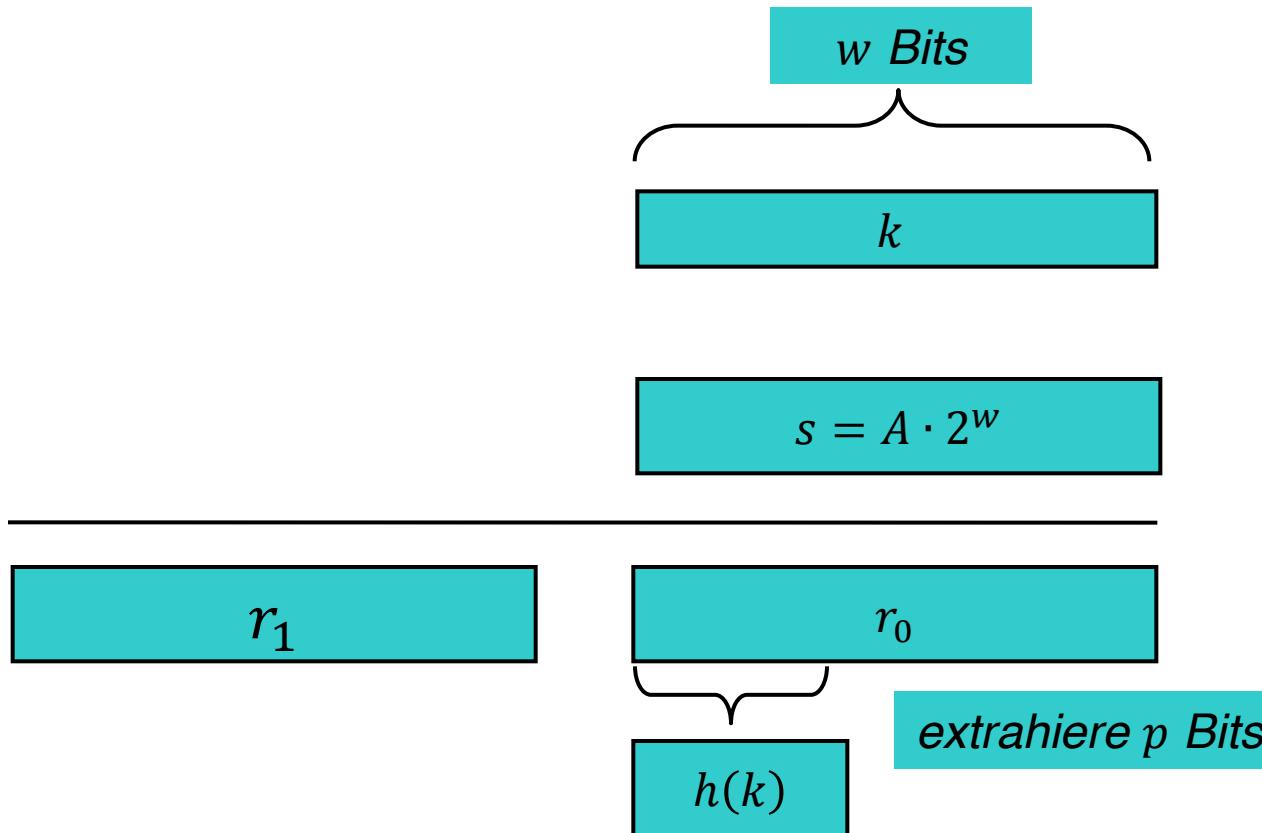
Multiplikationsmethode (2)

Berechnung $h(k)$:

(bei $m = 2^p$, $A = \frac{s}{2^w}$, $p \leq w$, $k \leq 2^w$ für alle Schlüssel k)

1. Berechne $r = k \cdot A \cdot 2^w$.
2. Schreibe r als $r_1 2^w + r_0$, $0 \leq r_0 \leq 2^w - 1$.
3. Binärdarstellung von $h(k)$ gegeben durch die oberen p Bits von r_0 .

Multiplikationsmethode – Illustration



Doppelt verkettete Listen

- **Verkettete Listen** bestehen aus einer Menge linear angeordneter Objekte.
- Anordnung realisiert durch **Verweise**.
- Unterstützen alle dynamischen Mengen mit den Operationen INSERT, DELETE, SEARCH, SEARCH-MINIMUM, usw.
Unterstützung ist nicht unbedingt effizient.
- Objekte in doppelt verketteter Liste L besitzen mindestens drei Felder: $key, next, prev$.
Außerdem Felder für Satellitendaten möglich.
- Zugriff auf Liste L durch Verweis/Zeiger $L.head$.

Doppelt verkettete Listen

- $L.\text{head}$ verweist auf erstes Element der Liste L .
- x ist Objekt in Liste L :
 - $x.\text{next}$ verweist auf nächstes Element in Liste,
 - $x.\text{prev}$ verweist auf voriges Element in Liste
- $x.\text{prev} = \text{NIL}$:
 - x besitzt keinen Vorgänger
 - dann ist x erstes Element der Liste und $L.\text{head}$ verweist auf x
- $x.\text{next} = \text{NIL}$:
 - x besitzt keinen Nachfolger
 - dann ist x letztes Element der Liste
- $L.\text{head} = \text{NIL}$:
 - Liste L ist leer

Varianten verketteter Listen

Einfach verkettete Listen:

Feld $prev$ nicht vorhanden.

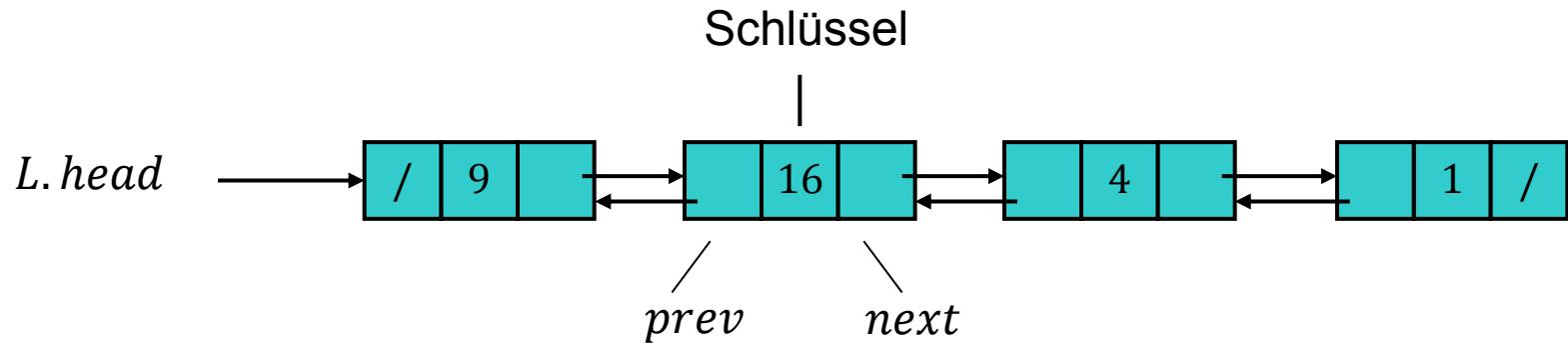
Sortierte verkettete Liste:

Schlüssel können sortiert werden. Reihenfolge in Liste entspricht sortierter Reihenfolge der Schlüssel.

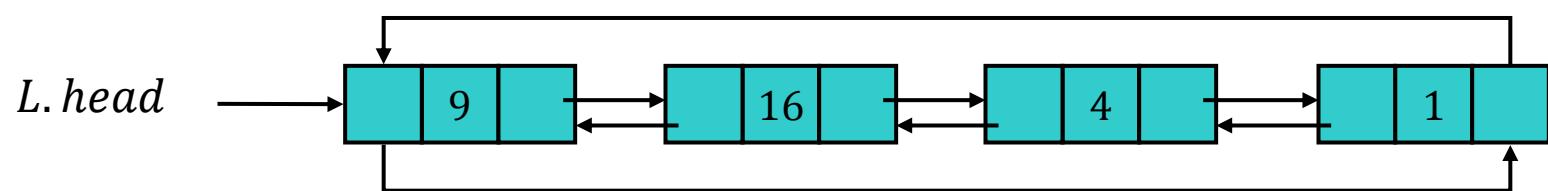
zyklisch/kreisförmig verkettete Listen:

- $next$ des letzten Objekts zeigt auf erstes Objekt,
- $prev$ des ersten Objekts zeigt auf letztes Objekt

Doppelt verkettete Listen – Beispiel



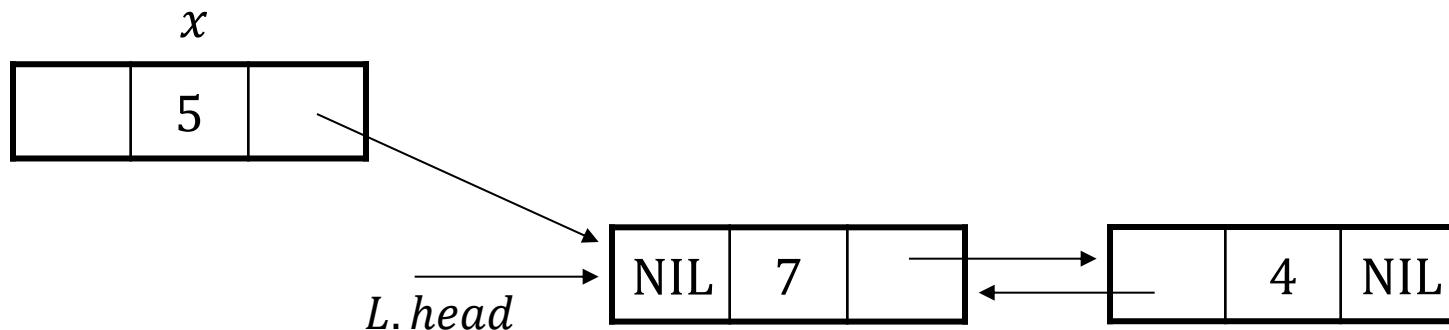
zyklisch doppelt verkettete Liste:



Doppelt verkettete Listen – Beispiel

LIST-INSERT(L, x)

- 1 $x.next = L.head$
- 2 **if** $L.head \neq \text{NIL}$
- 3 $L.head.prev = x$
- 4 $L.head = x$
- 5 $x.prev = \text{NIL}$

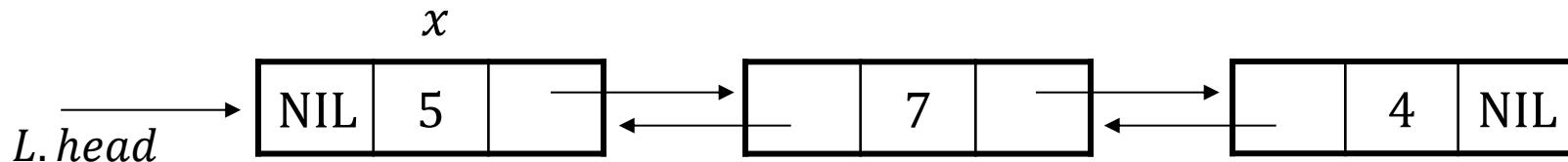


Doppelt verkettete Listen – Beispiel

LIST-INSERT(L, x)

- 1 $x.next = L.head$
- 2 **if** $L.head \neq \text{NIL}$
- 3 $L.head.prev = x$
- 4 $L.head = x$
- 5 $x.prev = \text{NIL}$

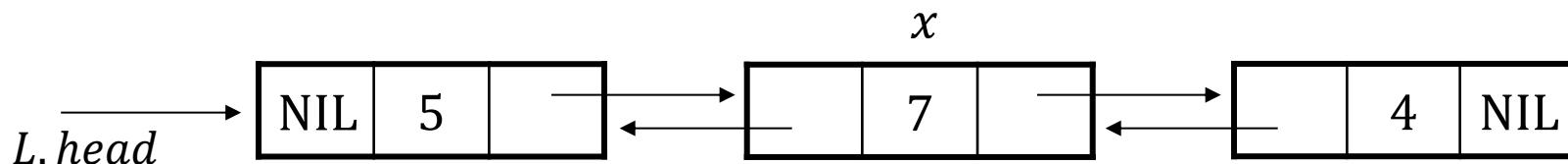
Laufzeit:
 $O(1)$



Doppelt verkettete Listen – Beispiel

LIST-DELETE(L, x)

- 1 **if** $x.prev \neq \text{NIL}$
- 2 $x.prev.next = x.next$
- 3 **else** $L.head = x.next$
- 4 **if** $x.next \neq \text{NIL}$
- 5 $x.next.prev = x.prev$

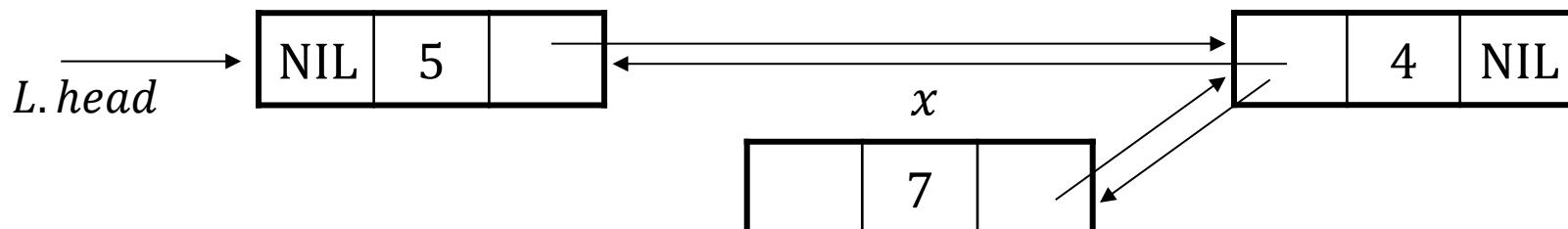


Doppelt verkettete Listen – Beispiel

LIST-DELETE(L, x)

- 1 **if** $x.prev \neq \text{NIL}$
- 2 $x.prev.next = x.next$
- 3 **else** $L.head = x.next$
- 4 **if** $x.next \neq \text{NIL}$
- 5 $x.next.prev = x.prev$

Laufzeit:
 $O(1)$

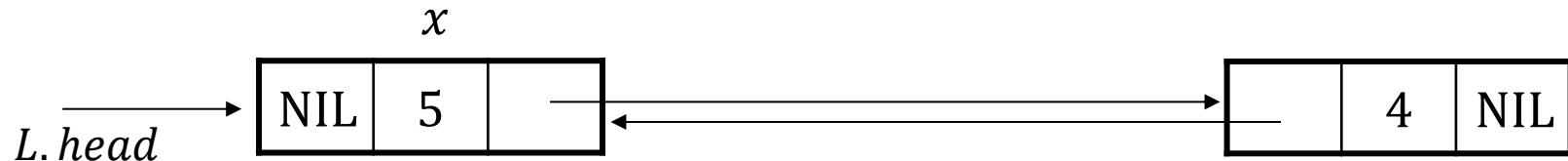


Doppelt verkettete Listen – Beispiel

LIST–SEARCH(L, x)

- 1 $x = L.nil.next$
- 2 **while** $x \neq L.nil$ and $x.key \neq k$
- 3 $x = x.next$
- 4 **return** x

LIST–SEARCH($L, 4$)

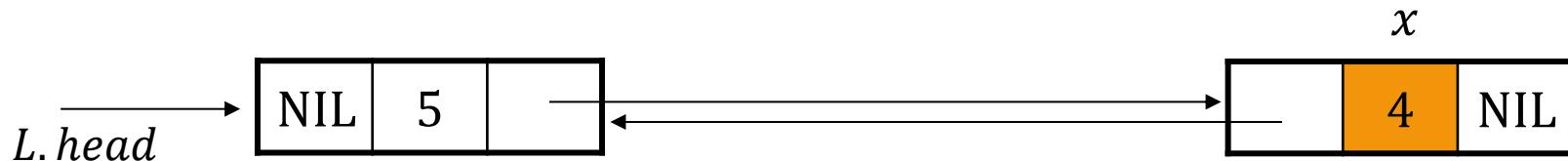


Doppelt verkettete Listen – Beispiel

LIST-SEARCH(L, x)

- 1 $x = L.nil.next$
- 2 **while** $x \neq L.nil$ and $x.key \neq k$
- 3 $x = x.next$
- 4 **return** x

Laufzeit:
 $O(n)$



Doppelt verkettete Listen

Datenstruktur Liste:

- Platzbedarf: $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile:

- Schnelles Einfügen/Löschen
- Speicherbedarf: $O(n)$

Nachteile:

- Hohe Laufzeit für Suche

Offene Adressierung (1)

- Hashing mit Kollisionsvermeidung weist Objekt mit gegebenem Schlüssel **feste Position** in Hashtabelle zu.
- Bei Hashing durch offene Adressierung wird Objekt mit Schlüssel **keine feste Position** zugewiesen.
- Position abhängig von Schlüssel und bereits belegten Positionen in Hashtabelle.
- Für neues Objekt wird erste freie Position gesucht. Dazu wird Hashtabelle nach freier Position durchsucht.
- Reihenfolge der Suche hängt vom Schlüssel des einzufügenden Objekts ab.

Offene Adressierung (2)

- Keine Listen zur Kollisionsvermeidung.
Wenn Anzahl eingefügter Objekte gleich m , dann sind keine weiteren Einfügungen mehr möglich.
- Listen zur Kollisionsvermeidung möglich, aber Ziel von offener Adressierung ist es, Verfolgen von Verweisen zu vermeiden.
- Da keine Listen benötigt werden, kann die Hashtabelle vergrößert werden.
- Suchen von Objekten in der Regel schneller, da keine Listen linear durchsucht werden müssen.

Offene Adressierung (3)

- Laufzeit für Einfügen im Durchschnitt nur noch $\Theta(1)$.
- Entfernen von Objekten schwierig, deshalb Anwendung von offener Adressierung oft nur, wenn Entfernen nicht benötigt wird.

Hashfunktionen bei offener Adressierung

- Hashfunktion legt für jeden Schlüssel fest, in welcher Reihenfolge für Objekte mit diesem Schlüssel nach freier Position in Hashtabelle gesucht wird.
- Hashfunktion h von der Form

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

m := Größe der Hashtabelle

- Verlangen, dass für alle Schlüssel k die Folge $(h(k, 0), h(k, 1), \dots, h(k, m - 1))$ eine Permutation der Folge $(0, 1, \dots, m - 1)$ ist.
- Folge $(h(k, 0), h(k, 1), \dots, h(k, m - 1))$ heißt **Testfolge** bei Schlüssel k .

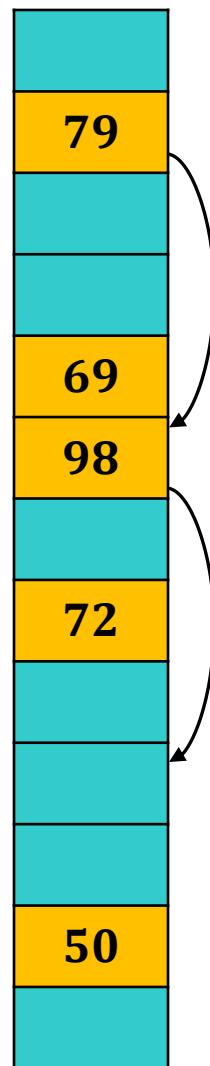
Einfügen bei offener Adressierung

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“
```

Dabei zur Vereinfachung angenommen, dass keine Satellitendaten vorhanden, d.h., Objekt ist Schlüssel.

Offene Adressierung – Illustration



Einfügen bei offener Adressierung

HASH-SEARCH(T, k)

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

Probleme bei Entfernen

- Können Felder i mit gelöschten Schlüsseln nicht wieder mit NIL belegen, denn dann wird Suche nach Schlüsseln, bei deren Einfügung Position i getestet wird, fehlerhaft sein.
- Mögliche Lösung ist, Felder gelöschter Schlüssel mit **DELETED** zu markieren.
- Aber dann werden Laufzeiten für HASH-INSERT und HASH-DELETE nicht mehr nur vom Lastfaktor $\alpha = \frac{n}{m}$ abhängen.
- Daher Anwendung von offener Adressierung nur, wenn keine Objekte entfernt werden müssen.

Mögliche Hashfunktionen

1. $h': U \rightarrow \{0, 1, \dots, m - 1\}$ Funktion.

Lineares Hashen:

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

2. $h': U \rightarrow \{0, 1, \dots, m - 1\}$ Funktion, $c_1, c_2 \neq 0$.

Quadratisches Hashen:

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

Vergleich Hashfunktionen

- Im linearen und quadratischen Hashen bestimmt erste getestete Position gesamte Testfolge.
- Damit jeweils nur m mögliche Testfolgen.
- Bei linearen Testfolgen zusätzlich lange zusammenhängende Folgen von besetzten Positionen.

Analyse von offener Adressierung (1)

Definition 11.4

Nehmen wir an, dass für jeden Schlüssel jede der $m!$ Permutationen der Folge $(0, 1, \dots, m - 1)$ mit gleicher Wahrscheinlichkeit die Testfolge ist, so nennen wir dieses uniformes offenes Hashen.

Satz 11.5

Ist der Lastfaktor einer Hashtabelle bei uniformem offenem Hashing α , so ist die erwartete Anzahl von Tests bei einer nicht erfolgreichen Suche $\frac{1}{1-\alpha}$.

Korollar 11.6

Ist der Lastfaktor einer Hashtabelle bei uniformem offenem Hashing α , so ist die erwartete Anzahl von Tests bei einer Einfügung $\frac{1}{1-\alpha}$.

Analyse von offener Adressierung (2)

Satz 11.7

Ist der Lastfaktor einer Hashtabelle bei uniformem offenem Hashing α , so ist bei einem zufälligen Suchobjekt die erwartete Anzahl von Tests bei einer erfolgreichen Suche höchstens

$$\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

Hashing – Zusammenfassung (1)

- Hashing ist eine einfache Methode, Wörterbücher zu implementieren.
- Hashing unterstützt also Einfügen, Entfernen und Suchen von Objekten.
- Einfügen und Entfernen benötigen im worst-case konstante Laufzeit.
- Suchen benötigt bei geeigneten Annahmen im Erwartungswert ebenfalls konstante Zeit.
- In der Praxis ist Hashing sehr erfolgreich.

Hashing – Zusammenfassung (2)

- Haben zwei Varianten kennen gelernt:
 1. Hashing mit Kollisionsverwaltung durch Listen.
 2. Offene Adressierung zur Kollisionsvermeidung.

Universelles Hashen – Idee

- Bei jeder Hashfunktion gibt es schlechte Eingaben, d.h., eine Folge von Einfüge-Operationen, sodass die Suche Zeit linear in der Anzahl der gespeicherten Objekte benötigt. Es müssen Objekte und Schlüssel so gewählt werden, dass alle Hashwerte identisch sind.
- Lösung besteht darin, nicht eine Hashfunktion zu wählen, sondern eine ganze Menge von Hashfunktionen.
- Unabhängig von Schlüsseln wird dann eine der Hashfunktionen ausgewählt.
Führt zu **universellem Hashing**.

Universelles Hashen – Definition

Definition 11.8

Sei H eine endliche Menge von Funktionen von U nach $\{0, 1, \dots, m - 1\}$. Die Menge H heißt universell, wenn für je zwei Schlüssel k, l mit $k \neq l$ gilt:

Die Anzahl der Funktionen $h \in H$ mit $h(k) = h(l)$ ist höchstens $\frac{|H|}{m}$.

Ausblick – Perfektes Hashen

- Soll nur Einfügen und Suchen unterstützt werden, und werden alle Objekte zu Beginn eingefügt, so kann Hashing verbessert werden.
- zu Beginn werden n Objekte in Zeit $\Theta(n)$ eingefügt
- Jede Suche benötigt danach im worst-case Zeit $\Theta(1)$.
- Idee wie bei Hashing mit Kollisionsverwaltung.
Allerdings werden nicht Listen zur Kollisionsverwaltung benutzt, sondern wieder Hashtabellen.
- Verfahren wird **perfektes Hashing** genannt.

12. Elementare Graphalgorithmen

- Graphen sind eine der wichtigsten **Modellierungskonzepte** der Informatik.
- Graphalgorithmen bilden die **Grundlage** vieler Algorithmen in der Praxis.
- Zunächst kurze Wiederholung von Graphen.
- Dann Darstellungen von Graphen.
- Schließlich einfache Graphalgorithmen:
 - Breiten- und Tiefensuche,
 - Zusammenhangskomponenten,
 - Minimalspannende Bäume

Wiederholung Graphen – Gerichtete Graphen

- Ein **gerichteter Graph** G ist ein Paar (V, E) , wobei V eine endliche Menge ist und $E \subseteq V \times V$.
- Elemente aus V heißen **Knoten**.
- Elemente aus E heißen **Kanten**.
- Entsprechend heißt V **Knotenmenge** und E heißt **Kantenmenge** von G .
- Kanten sind geordnete Paare von Knoten. Kanten der Form $(u, u), u \in V$ sind zugelassen und heißen **Schleifen**.
- Ist $(u, v) \in E$, so sagen wir, dass die Kante von u nach v führt. Wir sagen auch, dass u und v **adjazent** sind.
Müssen dann aber noch die **Richtung** berücksichtigen.

Wiederholung Graphen – Ungerichtete Graphen

- Ein **ungerichteter Graph** G ist ein Paar (V, E) , wobei V eine endliche Menge ist und E eine Menge von 2-elementigen Teilmengen von V ist.
- Elemente aus V heißen **Knoten**.
- Elemente aus E heißen **Kanten**.
- Entsprechend heißt V **Knotenmenge** und E heißt **Kantenmenge** von G .
- Formal haben Kanten die Form $\{u, v\}$ mit $u, v \in V$. Wir schreiben aber wie bei gerichteten Graphen Kanten als Paare (u, v) , unterscheiden dabei aber nicht zwischen (u, v) und (v, u) . Kanten der Form (u, u) sind nicht zugelassen.
- Ist $(u, v) \in E$, so sagen wir, dass u und v **adjazent** sind.

Darstellung von Graphen – Adjazenzlisten

- Eine **Adjazenzlisten-Darstellung** eines Graphen $G = (V, E)$ besteht aus einem **Array** Adj von $|V|$ Listen. Pro Knoten u enthält Adj damit genau eine Liste $Adj[u]$.
- Für alle $u \in V$ enthält $Adj[u]$ alle Knoten, die zu u adjazent sind. Äquivalent: $Adj[u]$ enthält alle Knoten v , sodass $(u, v) \in E$.
- **Knoten** in $Adj[u]$ sind in **beliebiger Reihenfolge** gespeichert.

Darstellung von Graphen – Adjazenzmatrix

- Für die **Adjazenzmatrix-Darstellung** eines Graphen nehmen wir an, dass die Knoten in V mit den Zahlen von $1, \dots, |V|$ nummeriert sind.
- Die Adjazenzmatrix von G ist dann eine $|V| \times |V|$ Matrix $A = (a_{ij})$ mit

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

- Ist G ein **ungerichteter** Graph, so ist die Matrix A **symmetrisch**, d.h. für alle (i, j) gilt $a_{ij} = a_{ji}$.
- Adjazenzmatrix benötigt immer Speicher $\Theta(|V|^2)$.

Breitensuche – Überblick

- Die **Breitensuche** ist ein Algorithmus, der die Grundlage vieler Graphenalgorithmen bildet.
- **Ziel** der Breitensuche ist es, bei einem Graphen $G = (V, E)$ und einer **Quelle** $s \in V$ alle Knoten $v \in V$ zu finden, die von s aus erreichbar sind. Dabei ist ein Knoten v von s aus erreichbar, wenn es in G einen Pfad von s nach v gibt.
- Die Breitensuche berechnet auch für alle Knoten v den **Abstand** $\delta(s, v)$ von s zu v . Dabei ist der Abstand von s zu v die minimale Anzahl von Kanten auf einem Pfad von s nach v .

Breitensuche – Überblick

- Wird v entdeckt, während $Adj[u]$ nach neuen Knoten durchsucht wird, so heißt u **Vorgänger** von v .
- Knoten sind entweder **weiß**, **grau** oder **schwarz**:
 - Weiß sind alle *noch nicht entdeckten* Knoten.
 - Grau sind alle *entdeckten* Knoten, deren Adjazenzliste *noch nicht vollständig* nach neuen Knoten *durchsucht* wurde.
 - Schwarz sind alle anderen Knoten, d.h. schwarze Knoten wurden bereits entdeckt und ihre Adjazenzliste wurde *vollständig* durchsucht.

Breitensuche

BFS (G, s)

```
1 for each vertex  $u \in G.V - \{s\}$ 
2    $u.color = \text{WHITE}$ 
3    $u.d = \infty$ 
4    $u.\pi = \text{NIL}$ 
5    $s.color = \text{GRAY}$ 
6    $s.d = 0$ 
7    $s.\pi = \text{NIL}$ 
8    $Q = \emptyset$ 
9   ENQUEUE( $Q, s$ )
10  while  $Q \neq \emptyset$ 
11     $u = \text{DEQUEUE}(Q)$ 
12    for each  $v \in G.Adj[u]$ 
13      if  $v.color == \text{WHITE}$ 
14         $v.color = \text{GRAY}$ 
15         $v.d = u.d + 1$ 
16         $v.\pi = u$ 
17        ENQUEUE( $Q, v$ )
18     $u.color = \text{BLACK}$ 
```

- BFS benutzt Queue für graue Knoten
- $u.color :=$ Feld für Farbe von v .
Initial WHITE.
- $u.d :=$ Feld für bislang berechneten Abstand zu s .
Initial ∞ .
- $u.\pi :=$ Feld für Vorgänger.
Initial NIL.

Breitensuche – Laufzeitanalyse

- Gesamtzeit für Durchläufe der Schleife in den Zeilen 11-18 insgesamt $O(|E|)$.
- Denn Gesamtgröße aller Adjazenzlisten $\Theta(|E|)$.

Satz 12.1

Bei Eingabe von Graph $G = (V, E)$ und Quelle s besitzt Algorithmus BFS Laufzeit $O(|V| + |E|)$.

Breitensuche – Erreichbarkeit, Kürzeste Pfade

Lemma 12.2

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

Sei $s \in V$ beliebig. Für jede Kante $(u, v) \in E$ gilt

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

Lemma 12.3

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

Sei $s \in V$ beliebig und s, G Eingabe für BFS.

Nach Beendigung von BFS gilt für jeden Knoten $v \in V$

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

Breitensuche – Erreichbarkeit, Kürzeste Pfade

Lemma 12.4

Die Quelle Q enthalte zu einem beliebigen Zeitpunkt des Ablaufs von BFS die Knoten (v_1, \dots, v_r) , wobei v_1 der Kopf ($Q.\text{head}$) und v_r das Ende ($Q.\text{tail}$) sei. Dann gilt

$$v_r.d \leq v_1.d + 1 \text{ und}$$

$$v_i.d \leq v_{i+1}.d, \quad i = 1, \dots, r - 1.$$

Lemma 12.5

Knoten v_i werde während des Ablaufs von BFS vor Knoten v_j in die Queue Q eingefügt.

Zum Zeitpunkt, an dem v_j in die Queue Q eingefügt wird, gilt

$$v_i.d \leq v_j.d.$$

Breitensuche – Erreichbarkeit, Kürzeste Pfade

Satz 12.6

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

Sei $s \in V$ beliebig und s, G Eingabe für BFS.

Nach Beendigung von BFS gilt für jeden Knoten $v \in V$

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

Insbesondere sind die von s aus erreichbaren Knoten v die Knoten mit $v.d < \infty$.

Weiter ist für jeden von s aus erreichbaren Knoten $v \neq s$ ein kürzester Pfad von s zu v gegeben durch einen kürzesten Pfad von s zum Vorgänger $v.\pi$ von v erweitert um die Kante $(v.\pi, v)$.

Breitensuchbäume

Betrachten nach BFS mit Eingabe G, s den Graphen $G_\pi = (V_\pi, E_\pi)$ mit

$$V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\} \text{ und}$$

$$E_\pi := \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$$

Satz 12.7

$G_\pi = (V_\pi, E_\pi)$ ist ein Baum. V_π enthält genau die von s aus erreichbaren Knoten in G .

Für jeden Knoten $v \in V_\pi$ ist der eindeutige Pfad von s zu v in G_π ein kürzester Pfad von s zu v in G .

Tiefensuche

- Suche **zunächst** „tiefer“ im Graph.
- **Neue Knoten** werden immer vom **zuletzt gefundenen** Knoten entdeckt.
- Sind alle adjazenten Knoten des zuletzt gefundenen Knotens ν bereits entdeckt, springe zurück zum Knoten, von dem aus ν entdeckt wurde.
- Wenn irgendwelche unentdeckten Knoten **übrigbleiben**, starte Tiefensuche von einem **dieser Knoten**.

Tiefensuche

Invariante Tiefensuche:

- zu Beginn: alle Knoten weiß
- entdeckte Knoten werden grau
- abgearbeitete Knoten werden schwarz
- zwei Zeitstempel: $v.d$ und $v.f$ (liegen zwischen 1 und $2|V|$)
 - $v.d$: v ist entdeckt
 - $v.f$: v ist abgearbeitet

Tiefensuche

DFS (G)

```
1 for each vertex  $u \in G.V$ 
2    $u.\text{color} = \text{WHITE}$ 
3    $u.\pi = \text{NIL}$ 
4    $\text{time} = 0$ 
5   for each vertex  $u \in G.V$ 
6     if  $u.\text{color} == \text{WHITE}$ 
7       DFS-VISIT( $G, u$ )
```

DFS-VISIT (G, u)

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

Tiefensuche

DFS (G)

```
1 for each vertex  $u \in G.V$ 
2    $u.color = \text{WHITE}$ 
3    $u.\pi = \text{NIL}$ 
4    $time = 0$ 
5   for each vertex  $u \in G.V$ 
6     if  $u.color == \text{WHITE}$ 
7       DFS-VISIT( $G, u$ )
```

DFS-VISIT (G, u)

```
1  $time = time + 1$ 
2  $u.d = time$ 
3  $u.color = \text{GRAY}$ 
4 for each  $v \in G.Adj[u]$ 
5   if  $v.color == \text{WHITE}$ 
6      $v.\pi = u$ 
7     DFS-VISIT ( $G, v$ )
8    $u.color = \text{BLACK}$ 
9    $time = time + 1$ 
10   $u.f = time$ 
```

Laufzeit:
 $O(|V| + |E|)$

Tiefensuche

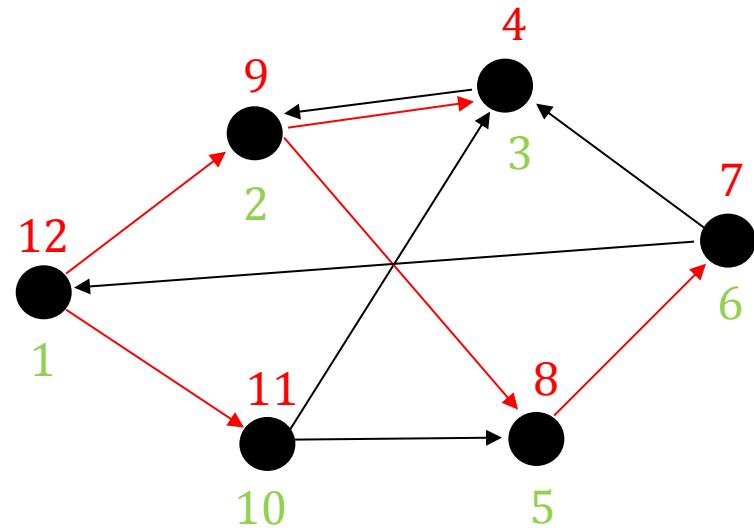
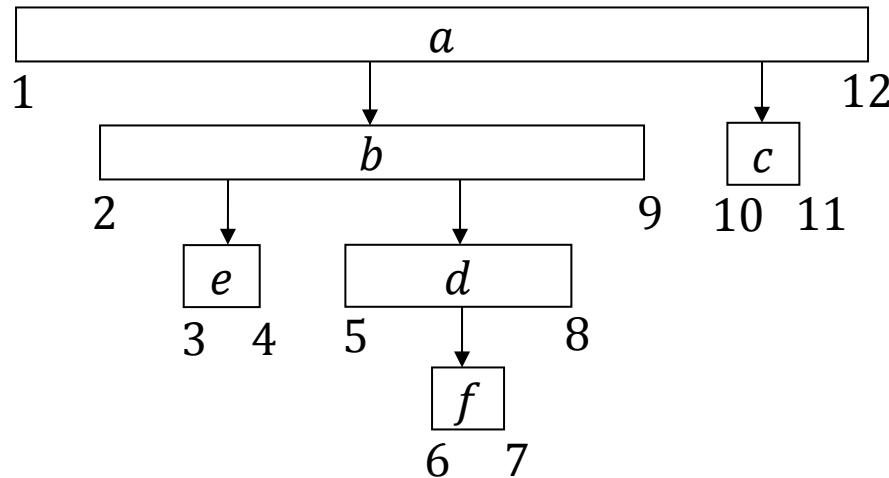
Satz 12.8 (Klammersatz zur Tiefensuche)

In jeder Tiefensuche eines gerichteten oder ungerichteten Graphen gilt für jeden Knoten u und v genau eine der folgenden drei Bedingungen:

1. Die Intervalle $[u.d, u.f]$ und $[v.d, v.f]$ sind vollständig disjunkt.
2. Intervall $[u.d, u.f]$ ist vollständig im Intervall $[v.d, v.f]$ enthalten und u ist Nachfolger von v im DFS-Baum.
3. Intervall $[v.d, v.f]$ ist vollständig im Intervall $[u.d, u.f]$ enthalten und v ist Nachfolger von u im DFS-Baum.

Tiefensuche

Beispiel:



Korollar 12.9

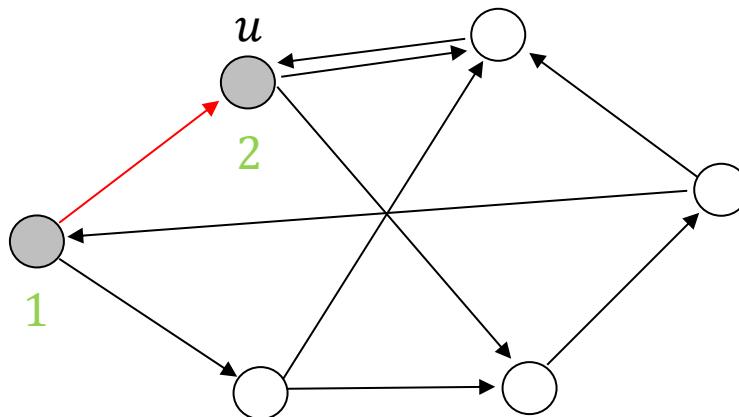
Knoten v ist echter Nachfolger von Knoten u im DFS-Baum von G , genau dann, wenn:

$$u.d < v.d < v.f < u.f$$

Tiefensuche

Satz 12.10 (Satz vom weißen Weg)

In einem DFS-Wald eines gerichteten oder ungerichteten Graphen G ist Knoten v ein Nachfolger von Knoten u , gdw. zum Zeitpunkt $u.d \leq v$ über einen Pfad weißer Knoten erreicht werden kann.



Tiefensuche – Laufzeitanalyse

Satz 12.11

Bei Eingabe von Graph $G = (V, E)$ besitzt Algorithmus DFS Laufzeit $O(|V| + |E|)$.

Analyse:

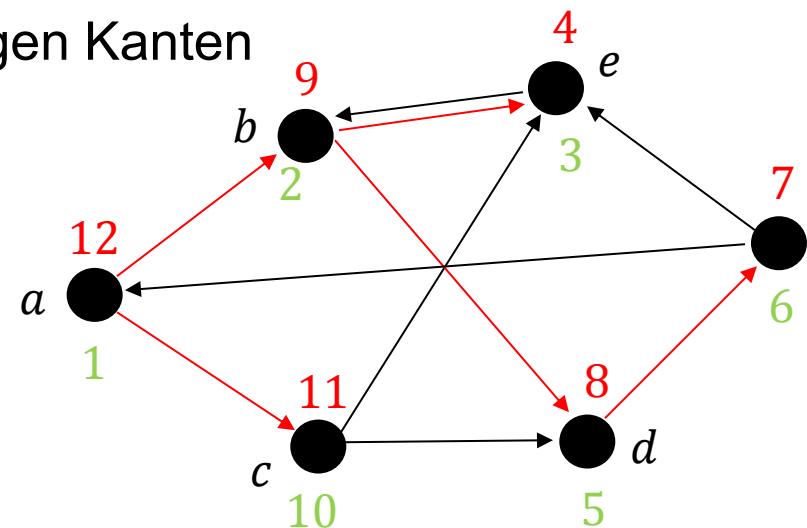
Wie bei Breitensuche.

Wir nutzen aus, dass die **Gesamtgröße** aller **Adjazenzlisten** $O(|E|)$.

Tiefensuche

Klassifikation von Kanten:

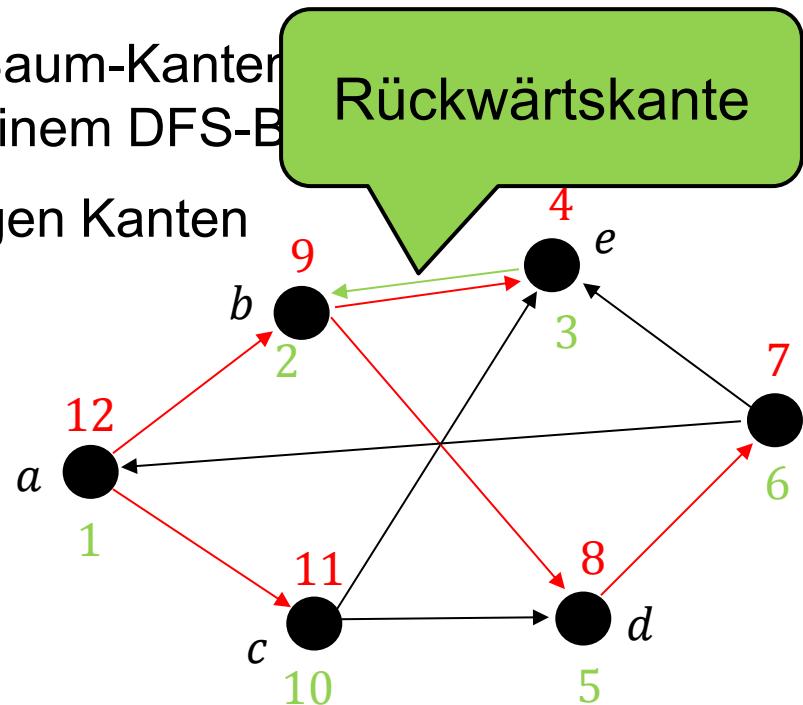
- **Baumkanten** sind Kanten des DFS-Walds G
- **Rückwärtskanten** sind Kanten (u, v) , die Knoten u mit Ahnen von u im DFS-Baum verbinden
- **Vorwärtskanten** sind die Nicht-Baum-Kanten (u, v) , die u mit einem Nachfolger v in einem DFS-Baum verbinden
- **Kreuzungskanten** sind alle übrigen Kanten



Tiefensuche

Klassifikation von Kanten:

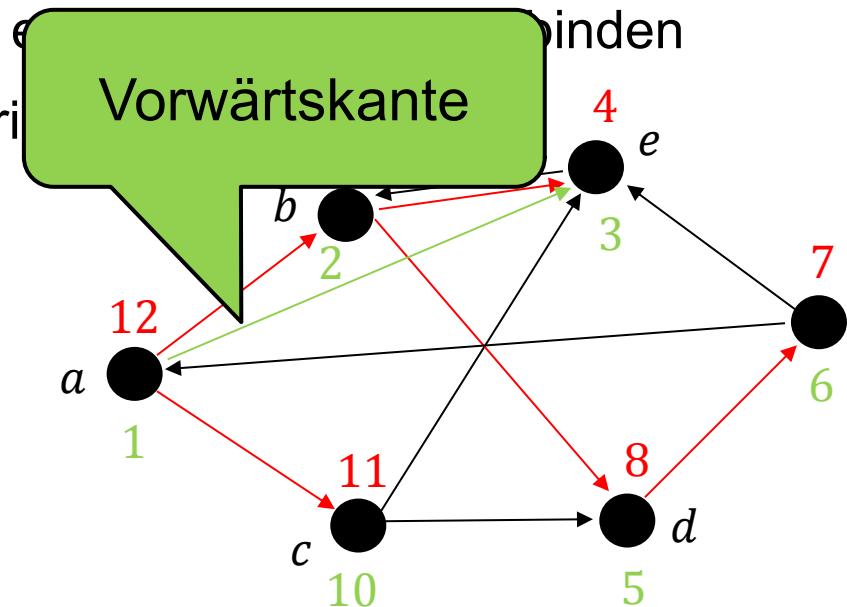
- **Baumkanten** sind Kanten des DFS-Walds G
- **Rückwärtskanten** sind Kanten (u, v) , die Knoten u mit Ahnen von u im DFS-Baum verbinden
- **Vorwärtskanten** sind die Nicht-Baum-Kanten, die u mit einem Nachfolger v in einem DFS-Baum verbinden
- **Kreuzungskanten** sind alle übrigen Kanten



Tiefensuche

Klassifikation von Kanten:

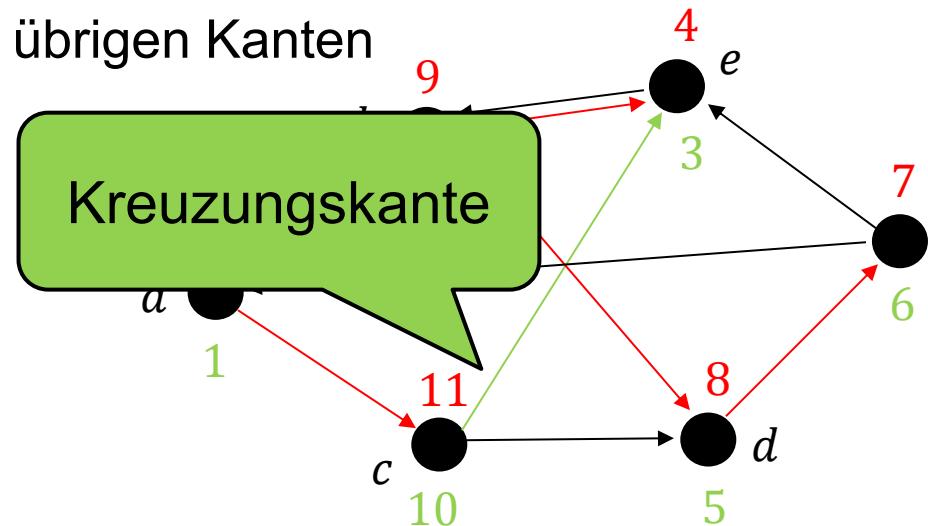
- **Baumkanten** sind Kanten des DFS-Walds G
- **Rückwärtskanten** sind Kanten (u, v) , die Knoten u mit Ahnen von u im DFS-Baum verbinden
- **Vorwärtskanten** sind die Nicht-Baum-Kanten (u, v) , die u mit einem Nachfolger v in einer Tiefensuche verbinden
- **Kreuzungskanten** sind alle übrigen Kanten



Tiefensuche

Klassifikation von Kanten:

- **Baumkanten** sind Kanten des DFS-Walds G
- **Rückwärtskanten** sind Kanten (u, v) , die Knoten u mit Ahnen von u im DFS-Baum verbinden
- **Vorwärtskanten** sind die Nicht-Baum-Kanten (u, v) , die u mit einem Nachfolger v in einem DFS-Baum verbinden
- **Kreuzungskanten** sind alle übrigen Kanten



Gewichtete Graphen

- Ein **gewichteter Graph** G ist ein Paar (V, E) zusammen mit einer Gewichtsfunktion w , wobei $E \subseteq V \times V$ und $w: E \rightarrow R$.
- Elemente aus V heißen **Knoten**, Elemente aus E heißen **Kanten**. Entsprechend heißt V **Knotenmenge** und E heißt **Kantenmenge** von G .
Für e aus E heißt $w(e)$ das Gewicht von e .
- Ein **kürzester Weg** von Knoten u zum Knoten v ist der Weg mit dem kleinsten Gewicht von u nach v . Dabei ist das Gewicht eines Weges die Summe der Kantengewichte auf diesem Weg.

Berechnung kürzester Wege

- kürzeste Wege von einem Startknoten s
(Single Source Shortest Path)
- kürzeste Wege zwischen allen Knotenpaaren
(All Pairs Shortest Path)
- Single Source Shortest Path (SSSP):
 - Gegeben: Gewichteter Graph G und Startknoten s
 - Gesucht: Für alle Knoten v die Distanz $\delta(s, v)$ sowie ein kürzester Weg.

Kürzeste Wege

Annahme: Alle Kantengewichte sind positiv

Algorithmus von Dijkstra:

- 1 Sei S die Menge der entdeckten Knoten
- 2 Invariante: Merke optimale Lösung für S :
Für alle $v \in S$ sei $v.d = \delta(s, v)$ die Länge
des kürzesten Weges von s nach v
- 3 Zu Beginn: $S = \{s\}$ und $s.d = 0$
- 4 **while** $V \neq S$
- 5 wähle Knoten $v \in V \setminus S$ mit mindestens einer Kante
zu Knoten aus S und für den $v.d' = \min_{(u,v) \in E} u.d + w(u, v)$
so klein wie möglich ist (u aus S)
- 6 füge v zu S hinzu und setze $v.d = v.d'$

Kürzeste Wege

Wie kann man Pfade berechnen?

- Wie bei BFS/DFS über Feld π
- Wenn (u, v) die Kante ist, für die das Minimum in Zeile 5 erreicht wird, dann setze $v.\pi = u$
- Kürzester s - u -Weg $P(u)$ ist implizit gespeichert:
 - Für $u = s$ haben wir den leeren Weg als kürzesten Weg von s nach s
 - Für $u \neq s$ gilt:
 $P(u)$ besteht aus Weg $P(u.\pi)$ gefolgt von Kante $(u.\pi, u)$

Kürzeste Wege

Satz 12.14 (Korrektheit)

Invariante: Für jedes $u \in S$ ist zu jedem Zeitpunkt der Ausführung des Algorithmus der Weg $P(u)$ ein kürzester $s-u$ -Weg.

Kürzeste Wege

Wie kann man Dijkstras Algorithmus effizient implementieren?

- Naiver Ansatz:
Überprüfe für jeden Knoten aus $V - S$ alle Kanten
- Laufzeit $O(|V| \cdot |E|)$

Besser:

- Halte $\nu.d'$ Werte für alle $\nu \in V - S$ aufrecht
- Speichere alle Knoten aus $V - S$ in Prioritätenschlange ab mit Schlüssel $\nu.d'$
- Problem: Was ist, wenn sich Schlüssel verändern

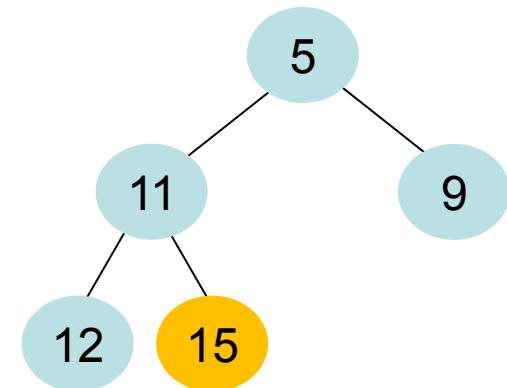
Kürzeste Wege

Heaps mit Decrease-Key:

- Datenstruktur MIN-HEAP A
- neue Funktion DECREASE-KEY($A, i, newkey$):
wir bekommen Index i des Elements des Heaps,
dessen Wert (Schlüssel) auf den Wert $newkey$
- $newkey$ ist kleiner als der alte Schlüssel

DECREASE-KEY($A, i, newkey$)

- 1 $A[i] = newkey$
- 2 **while** $i > 1$ and $A[i.parent] > A[i]$
- 3 exchange $A[i.parent]$ with $A[i]$
- 4 $i = i.parent$



DECREASE-KEY($A, 5, 3$)

Kürzeste Wege

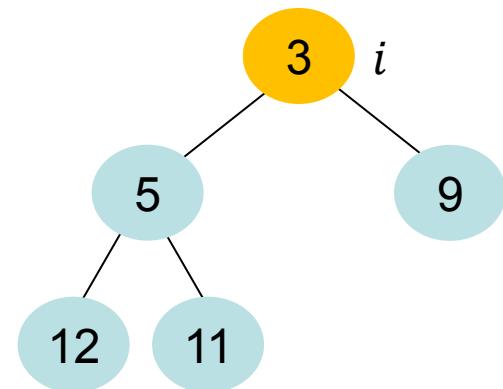
Heaps mit Decrease-Key:

- Datenstruktur MIN-HEAP A
- neue Funktion DECREASE-KEY($A, i, newkey$):
wir bekommen Index i des Elements des Heaps,
dessen Wert (Schlüssel) auf den Wert $newkey$
- $newkey$ ist kleiner als der alte Schlüssel

Laufzeit:
 $O(\log n)$

DECREASE-KEY($A, i, newkey$)

- 1 $A[i] = newkey$
- 2 **while** $i > 1$ and $A[i.parent] > A[i]$
- 3 exchange $A[i.parent]$ with $A[i]$
- 4 $i = i.parent$



DECREASE-KEY($A, 5, 3$)

Kürzeste Wege

Wie kann man Dijkstras Algorithmus effizient implementieren?

- Halte $v.d'$ Werte für alle $v \in V - S$ aufrecht
- Speichere alle Knoten aus $V - S$ in Prioritätenschlange ab mit Schlüssel $v.d'$
- Für jeden Knoten v speichere Zeiger auf sein Vorkommen im Heap
- alle $v.d'$ Werte vergrößern sich nie
- benutze DECREASE-KEY, wenn sich der Wert $v.d'$ verringert

Kürzeste Wege

DIJKSTRA(G, w, s)

- 1 INITIALIZE–SINGLE–SOURCE(G, s)
- 2 $S = \emptyset$
- 3 $Q = G.V$
- 4 **while** $Q \neq \emptyset$
 - 5 $u = \text{EXTRACT–MIN}(Q)$
 - 6 $S = S \cup \{u\}$
 - 7 **for** each vertex $v \in G.\text{Adj}[u]$
 - 8 RELAX(u, v, w)

Beobachtung:

Wir können Felder d und d' durch ein Feld d ersetzen, weil d nur für Knoten aus S benutzt wird und d' nur für Knoten aus $V - S$.

INITIALIZE–SINGLE–SOURCE(G, s)

- 1 **for** each vertex $v \in G.V$
 - 2 $v.d = \infty; v.\pi = \text{NIL}$
 - 3 $s.d = 0$

RELAX(u, v, w)

- 1 **if** $v.d > u.d + w(u, v)$
 - 2 $v.d = u.d + w(u, v)$
 - 3 $v.\pi = u$

Kürzeste Wege

DIJKSTRA(G, w, s)

- 1 INITIALIZE–SINGLE–SOURCE(G, s)
- 2 $S = \emptyset$
- 3 $Q = G.V$
- 4 **while** $Q \neq \emptyset$
 - 5 $u = \text{EXTRACT–MIN}(Q)$
 - 6 $S = S \cup \{u\}$
 - 7 **for** each vertex $v \in G.\text{Adj}[u]$
 - 8 RELAX(u, v, w)

Hier muss zusätzlich noch eine DECREASE–KEY Operation durchgeführt werden, da Q ein MIN–HEAP mit den d -Werten der Knoten als Schlüssel ist.

INITIALIZE–SINGLE–SOURCE(G, s)

- 1 **for** each vertex $v \in G.V$
 - 2 $v.d = \infty; v.\pi = \text{NIL}$
 - 3 $s.d = 0$

RELAX(u, v, w)

- 1 **if** $v.d > u.d + w(u, v)$
 - 2 $v.d = u.d + w(u, v)$
 - 3 $v.\pi = u$

Kürzeste Wege

DIJKSTRA(G, w, s)

- 1 INITIALIZE–SINGLE–SOURCE(G, s)
- 2 $S = \emptyset$
- 3 $Q = G.V$
- 4 **while** $Q \neq \emptyset$
- 5 $u = \text{EXTRACT–MIN}(Q)$
- 6 $S = S \cup \{u\}$
- 7 **for** each vertex $v \in G.Adj[u]$
- 8 RELAX(u, v, w)

Laufzeit:

$$O((|V| + |E|) \log |V|)$$

INITIALIZE–SINGLE–SOURCE(G, s)

- 1 **for** each vertex $v \in G.V$
- 2 $v.d = \infty; v.\pi = \text{NIL}$
- 3 $s.d = 0$

RELAX(u, v, w)

- 1 **if** $v.d > u.d + w(u, v)$
- 2 $v.d = u.d + w(u, v)$
- 3 $v.\pi = u$

Dijkstra

Dijkstras Algorithmus

- Kürzeste Wege von einem Knoten und mit positiven Kantengewichten
- Laufzeit $O((|V| + |E|) \log|V|)$
- Bessere Laufzeit von $O((|V| \log|V| + |E|))$ möglich mit besseren Prioritätenschlangen

Fragen:

- Was passiert bei negativen Kantengewichten?
- Wie kann man das Kürzeste-Wege-Problem für alle Paare von Knoten effizient lösen?

Wiederholung – Kürzeste Wege

Dijkstras Algorithmus

- Kürzeste Wege von einem Knoten und mit positiven Kantengewichten
- Laufzeit $O((|V| + |E|) \log|V|)$
- Bessere Laufzeit von $O(|V| \log|V| + |E|)$ möglich mit besseren Prioritätenschlangen

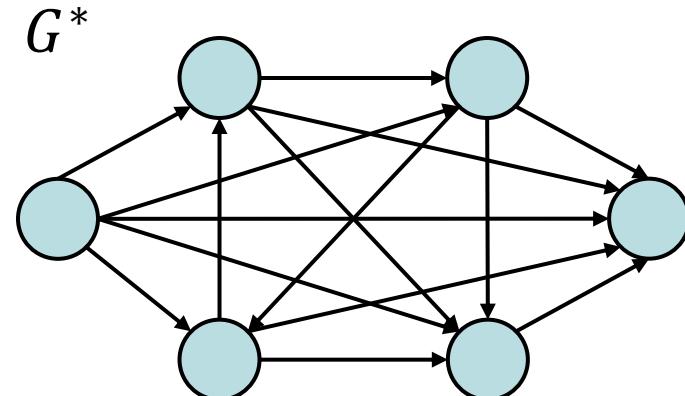
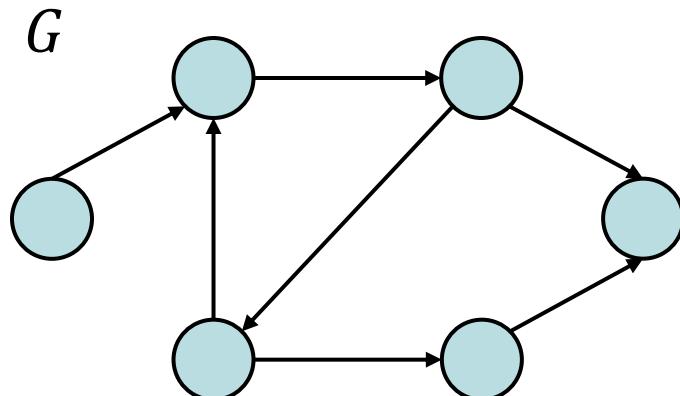
Fragen

- Was passiert bei negativen Kantengewichten?
- Wie kann man das Kürzeste-Wege-Problem für alle Paare von Knoten effizient lösen?

Kürzeste Wege

Das „transitive Hülle“-Problem:

- Gegeben sei ein gerichteter, ungewichteter Graph $G = (V, E)$
- Gesucht: Die transitive Hülle $G^* = (V, E^*)$ von G , wobei $E^* = \{(u, v) : \text{es gibt einen Weg von } u \text{ nach } v \text{ in } G\}$



Kürzeste Wege

Das „transitive Hülle“-Problem:

- Gegeben sei ein gerichteter, ungewichteter Graph $G = (V, E)$
- Gesucht: Die transitive Hülle $G^* = (V, E^*)$ von G , wobei $E^* = \{(u, v) : \text{es gibt einen Weg von } u \text{ nach } v \text{ in } G\}$

Transitive Hülle:

- In $O(|V|^2 + |V| |E|)$ Zeit mit Breiten- oder Tiefensuche von jedem Knoten.
- Geht das auch schneller?

Kürzeste Wege

Graphen und Matrixmultiplikation:

- Sei A die $n \times n$ -Adjazenzmatrix von Graph G mit Knotenmenge $\{1, \dots, n\}$
- Was ist $A \cdot A$?

Behauptung 12.15

Sei $Z = A \cdot A$. Dann gilt $z_{ij} > 0$, gdw. es in G einen Pfad der Länge 2 von Knoten i zu Knoten j gibt.

Kürzeste Wege

Behauptung 12.16

Sei $Z' = A \cdot A + A$. Dann gilt, dass $z'_{ij} > 0$, gdw. es einen Weg der Länge 1 oder 2 von Knoten i zu Knoten j gibt.

Konstruiere Matrix B mit:

$$b_{ij} = 1 \Leftrightarrow z'_{ij} > 0$$

Behauptung 12.17

Im Graphen der Matrix B ist eine Kante von i nach j , gdw. es in G einen Weg der Länge höchstens 2 von i nach j gibt.

Kürzeste Wege

Behauptung 12.18 —

Sei P ein Weg der Länge $k > 1$ von i nach j in G .

Dann existiert ein Weg der Länge höchstens $\frac{2}{3}k$ von i nach j in dem von der Matrix B beschriebenen Graphen G' .

Konsequenz aus 12.17 und 12.18:

- Wenn wir die Berechnung von $B \log_{3/2} n$ mal iterieren, haben wir die transitive Hülle berechnet.

Kürzeste Wege

Satz 12.19 —

Der Algorithmus TRANSITIVE–CLOSURE berechnet die transitive Hülle eines Graphen G in $O(M(n) \log n)$ Zeit, wobei $M(n)$ die Laufzeit zur Matrixmultiplikation bezeichnet.

13. Minimale Spannbäume

Definition 13.1 —

1. Ein gewichteter ungerichteter Graph (G, w) ist ein ungerichteter Graph $G = (V, E)$, zusammen mit einer Gewichtsfunktion $w: E \rightarrow R$.
2. Ist $H = (U, F)$, $U \subseteq V, F \subseteq E$ ein Teilgraph von G , so ist das Gewicht $w(H)$ von H definiert als

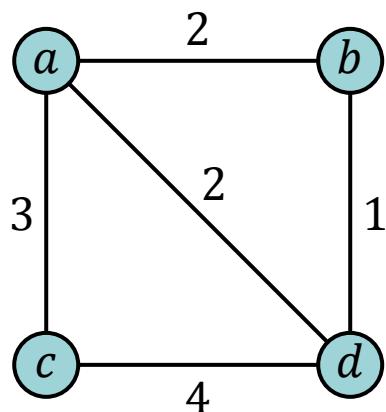
$$w(H) = \sum_{e \in F} w(e)$$

Minimale Spannbäume – Definition

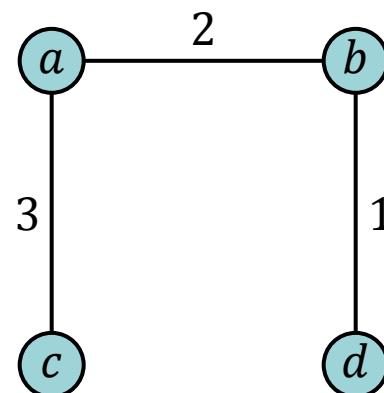
Definition 13.2 —

1. Ein Teilgraph H eines ungerichteten Graphen G heißt Spannbaum von G , wenn H ein Baum auf den Knoten von G ist (H muss alle Knoten enthalten).
2. Ein Spannbaum S eines gewichteten ungerichteten Graphen G heißt minimaler Spannbaum von G , wenn S minimales Gewicht unter allen Spannbäumen von G besitzt.

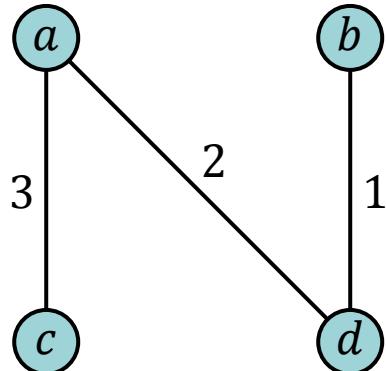
Illustration von minimalen Spannbäumen



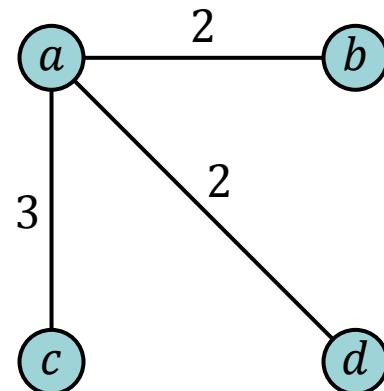
Graph $G = (V, E)$



Spannbaum für G (minimal)

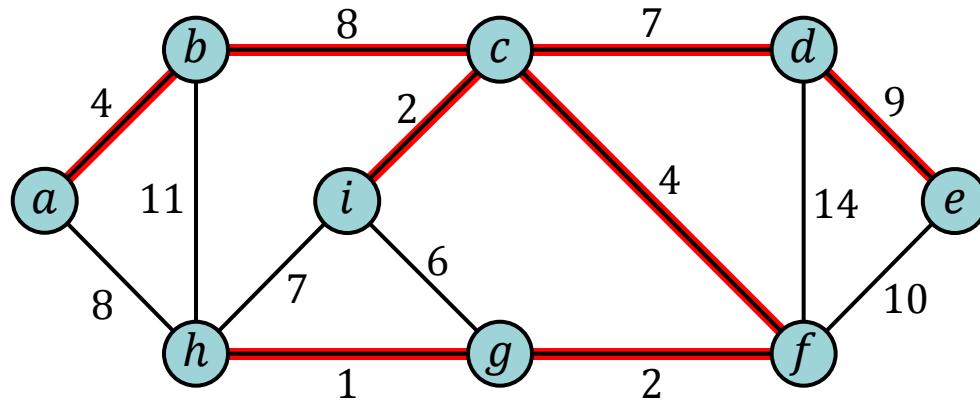


Spannbaum für G (minimal)



Spannbaum für G (nicht minimal)

Illustration von minimalen Spannbäumen



Berechnung minimaler Spannbäume

Ziel:

Gegeben sei ein gewichteter ungerichteter Graph (G, w) , $G = (V, E)$.

Wollen effizient einen minimalen Spannbaum von (G, w) finden.

Vorgehensweise:

Erweitern sukzessive eine Kantenmenge $A \subseteq E$ zu einem minimalen Spannbaum:

1. Zu Beginn $A = \{ \}$.
2. Ersetzen in jedem Schritt A durch $A \cup \{(u, v)\}$,
wobei (u, v) eine A -sichere Kante ist.
3. Solange bis $|A| = |V| - 1$

Definition 13.3

(u, v) heißt A -sicher, wenn mit A auch $A \cup \{(u, v)\}$ zu einem minimalen Spannbaum erweitert werden kann.

Generischer MST-Algorithmus

GENERIC-MST(G, w)

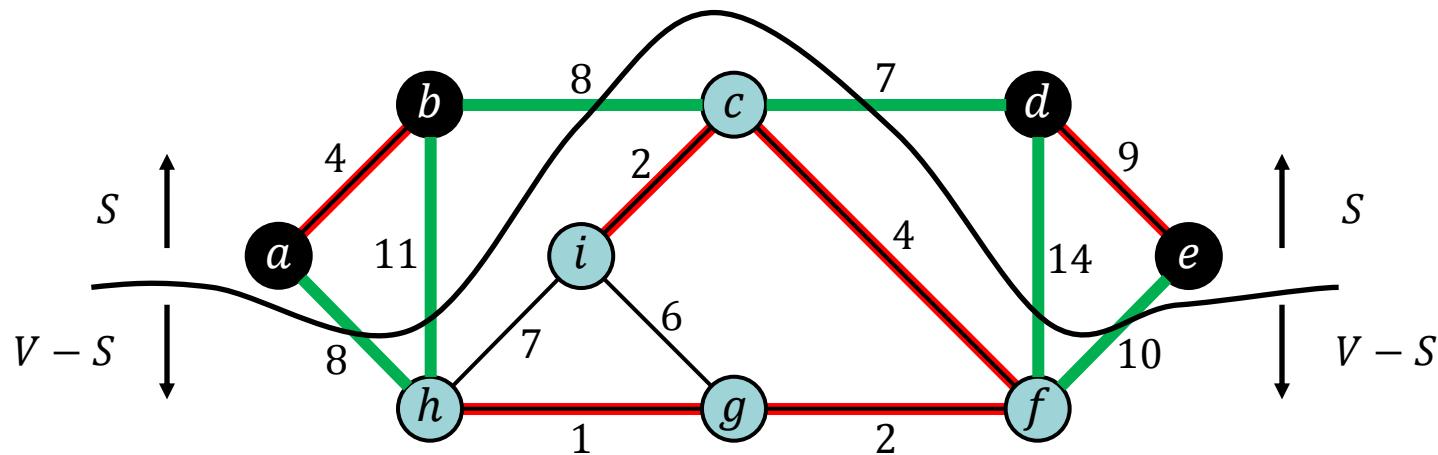
- 1 $A = \emptyset$
- 2 **while** A does not form a spanning tree
- 3 find an edge (u, v) that is safe for A
- 4 $A = A \cup \{(u, v)\}$
- 5 **return** A

Schnitte in Graphen

Definition 13.4 —

1. Ein **Schnitt** $(C, V - C)$ in einem Graphen $G = (V, E)$ ist eine Partition der Knotenmenge V des Graphen.
2. Eine Kante von G **kreuzt** einen Schnitt $(C, V - C)$, wenn ein Knoten der Kante in C und der andere Knoten in $V - C$ liegt.
3. Ein Schnitt $(C, V - C)$ ist mit einer Teilmenge $A \subseteq E$ **verträglich**, wenn kein Element von A den Schnitt kreuzt.
4. Eine $(C, V - C)$ kreuzende Kante heißt **leicht**, wenn sie eine Kante minimalen Gewichts unter den $(C, V - C)$ kreuzenden Kanten ist.

Schnitt in einem Graphen



— Schnittkanten

Charakterisierung sicherer Kanten

Satz 13.5

Sei (G, w) , $G = (V, E)$ ein gewichteter ungerichteter Graph.

Die Kantenmenge $A \subseteq E$ sei in einem minimalen Spannbaum von (G, w) enthalten. Weiter sei $(C, V - C)$ ein mit A verträglicher Schnitt und (u, v) sei eine leichte $(C, V - C)$ kreuzende Kante.

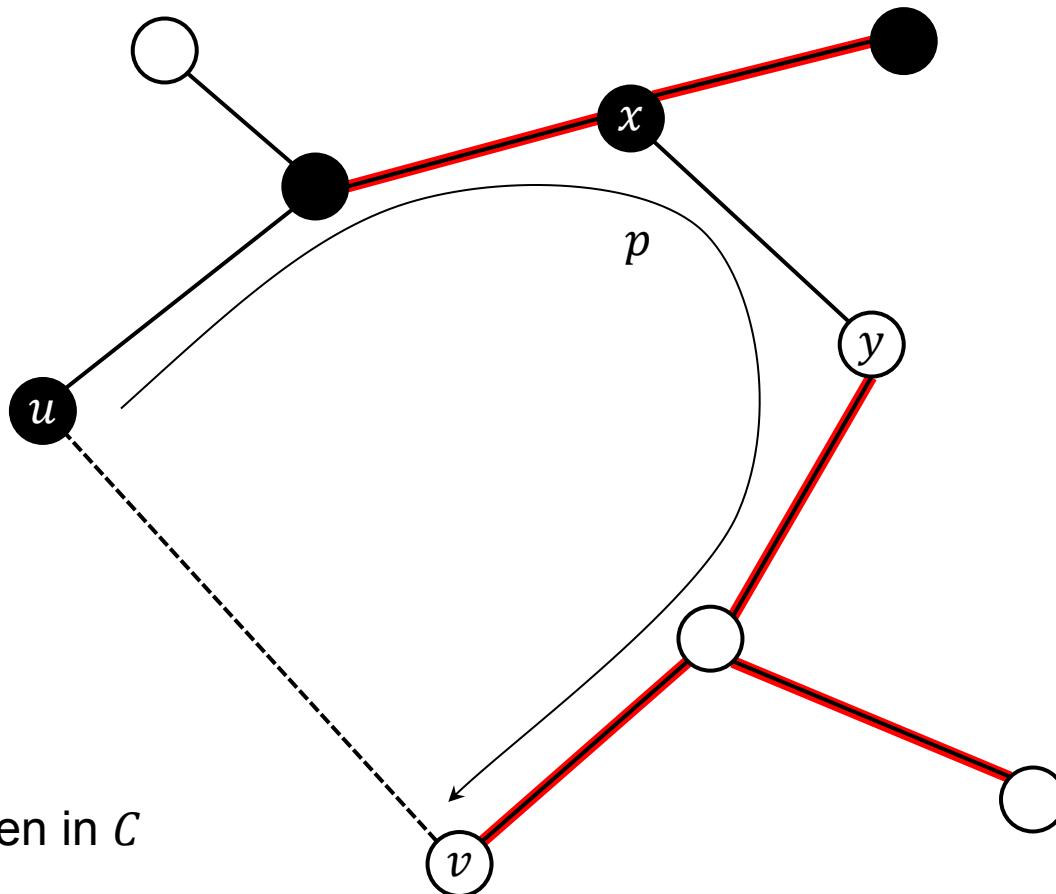
Dann ist (u, v) eine A -sichere Kante.

Korollar 13.6

Sei (G, w) , $G = (V, E)$ ein gewichteter ungerichteter Graph.

Die Kantenmenge $A \subseteq E$ sei in einem minimalen Spannbaum von (G, w) enthalten. Ist (u, v) eine Kante minimalen Gewichts, die eine Zusammenhangskomponente C von $G_A = (V, A)$ mit dem Rest des Graphen G_A verbindet, dann ist (u, v) eine A -sichere Kante.

Beweis des Satzes – Illustration



Algorithmus von Prim

- Zu jedem Zeitpunkt des Algorithmus besteht der Graph $G_A = (V, A)$ aus einem Baum T_A und einer Menge von isolierten Knoten I_A .
- Eine Kante minimalen Gewichts, die einen Knoten in I_A mit T_A verbindet, wird zu A hinzugefügt.
- Die Knoten in I_A sind in einem MIN–HEAP organisiert. Dabei ist der Schlüssel $\nu.\text{key}$ eines Knotens $\nu \in I_A$ gegeben durch das minimale Gewicht einer Kante, die ν mit T_A verbindet.

Algorithmus von Prim – Pseudocode

MST-PRIM(G, w, r)

```
1 for each  $u \in G.V$ 
2      $u.key = \infty$ 
3      $u.\pi = \text{NIL}$ 
4  $r.key = 0$ 
5  $Q = G.V$ 
6 while  $Q \neq \emptyset$ 
7      $u = \text{EXTRACT-MIN}(Q)$ 
8     for each  $v \in G.Adj[u]$ 
9         if  $v \in Q$  and  $w(u, v) < v.key$ 
10             $v.\pi = u$ 
11             $v.key = w(u, v)$ 
```

Heap-Decrease-Key

- Die Funktion $\text{DECREASE-KEY}(H, v, v.\text{key})$ ersetzt den aktuellen Schlüssel von v durch den neuen Wert $v.\text{key}$. Dabei muss der neue Schlüssel kleiner als der alte Schlüssel sein.
- Kann bei einem Heap mit n Elementen in Zeit $O(\log(n))$ ausgeführt werden.

Algorithmus von Prim – Laufzeit

- Zeile 7 pro Durchlauf $O(\log(|V|))$ wird $|V|$ mal durchlaufen.
- Zeilen 8-11 $|E|$ mal durchlaufen, da Größe aller Adjazenzlisten zusammen genau $2|E|$.

Satz 13.7

Der Algorithmus von Prim berechnet in Zeit $O(|V| \log(|V|) + |E| \log(|V|))$ einen minimalen Spannbaum eines gewichteten ungerichteten Graphen $(G, w), G = (V, E)$.

Mit **Fibonacci-Heaps** Verbesserung auf $O(|V| \log(|V|) + |E|)$ möglich.

Algorithmus von Kruskal – Idee

- Kruskals Algorithmus berechnet minimale Spannbäume, allerdings mit anderer Strategie als Prims Algorithmus.
- Kruskals Algorithmus erweitert sukzessive Kantenmenge A zu einem Spannbaum. Zu jedem Zeitpunkt besteht $G_A = (V, A)$ aus einer Menge von Bäumen.
- Eine Kante minimalen Gewichts, die in G_A keinen Kreis erzeugt, wird zu A in jedem Schritt hinzugefügt.
- Solch eine Kante muss zwei Bäume in G_A verbinden.
- Korrektheit folgt dann aus Satz 13.5 und Korollar 13.6.

Algorithmus von Kruskal – Datenstruktur

- Kruskals Algorithmus benötigt Datenstruktur, mit deren Hilfe
 1. Für jede Kante $(u, v) \in E$ effizient entschieden werden kann, ob u und v in derselben Zusammenhangskomponente von G_A liegen.
 2. Zusammenhangskomponenten effizient verschmolzen werden können.
- Solch eine Datenstruktur ist **Union-Find-Datenstruktur für disjunkte dynamische Mengen**.

Disjunkte dynamische Mengen

- Gegeben ist eine Menge U von Objekten.
- Verwaltet wird immer eine Familie $\{S_1, S_2, \dots, S_k\}$ von disjunkten Teilmengen von U .
- Teilmengen S_i werden identifiziert mit Hilfe eines Elements aus S_i , einem sogenannten Repräsentanten.

Disjunkte dynamische Mengen

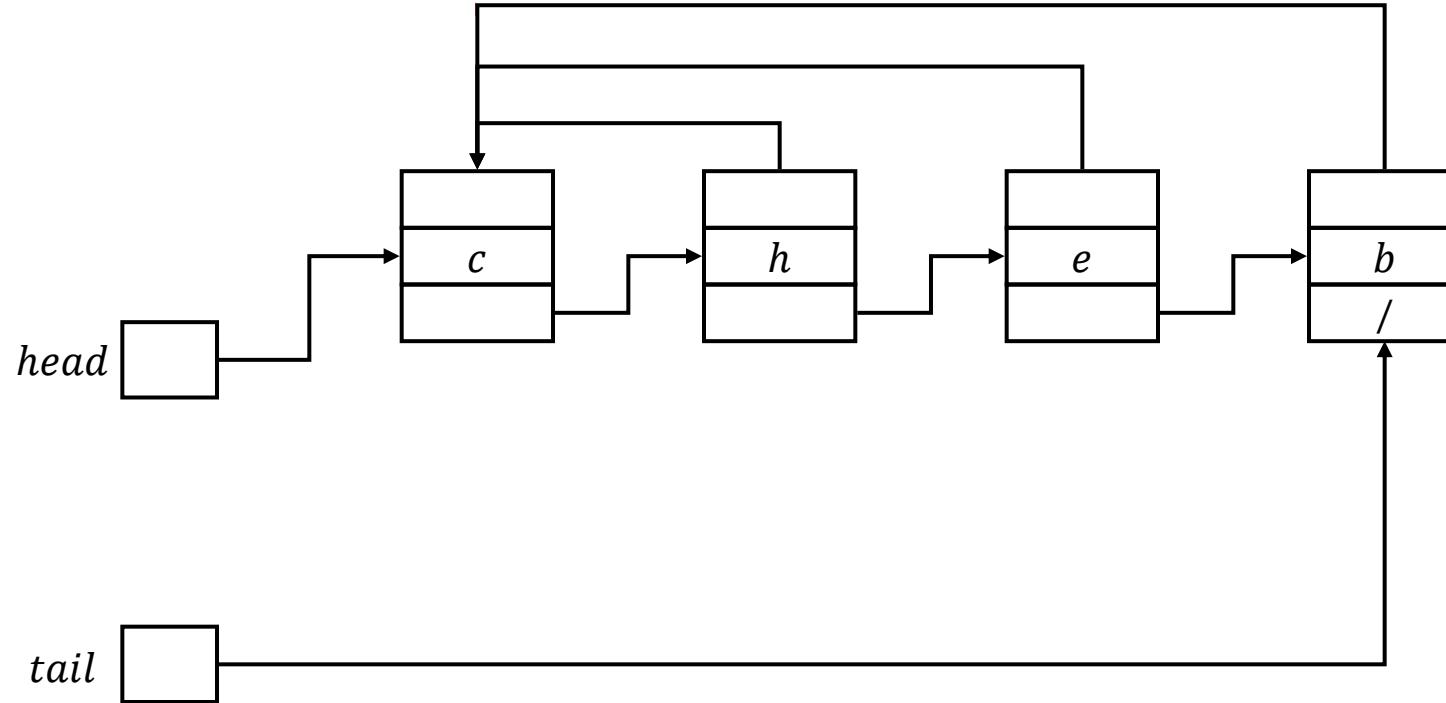
- Die folgenden Operationen sollen unterstützt werden:
 1. MAKE–SET(x) erzeugt Teilmenge $\{x\}$ mit Repräsentant x .
 2. UNION(x, y) vereinigt die beiden Teilmengen, die x bzw. y enthalten.
 3. FIND–SET(x) liefert den Repräsentanten derjenigen Menge, die x enthält.

Union-Find mit verketteten Listen

Realisierung einer Datenstruktur für disjunkte dynamische Menge kann mit verketteten Listen erfolgen:

1. Für jede Teilmenge S gibt es eine verkettete Liste S der Objekte in S .
2. Repräsentant ist dabei das erste Objekt der Liste.
3. Zusätzlich $S.\text{head}$ und $S.\text{tail}$ – Verweise auf erstes bzw. letztes Element der Liste. Feld $S.\text{size}$ speichert Größe der Liste.
4. Für jedes Element x der Liste Verweis auf Repräsentanten der Liste.

Union-Find mit verketteten Listen – Illustration



Analyse vieler Operationen

Satz 13.9

Werden verkettete Listen als Union-Find-Datenstruktur benutzt und werden insgesamt m Operationen MAKE-SET, FIND-SET und UNION ausgeführt, von denen n Operationen MAKE-SET-Operationen sind, so können alle Operationen zusammen in Zeit $O(m + n \log(n))$ ausgeführt werden.

Kruskals Algorithmus und Union-Find

- Benutzen UNION–FIND-Datenstruktur, um Zusammenhangskomponenten von $G_A = (V, A)$ zu verwalten.
- Test, ob (u, v) zwei verschiedene Zusammenhangskomponenten verbindet durch Test, ob $\text{FIND–SET}(u) = \text{FIND–SET}(v)$.
- Verschmelzen von Zusammenhangskomponenten durch UNION.

Algorithmus von Kruskal – Pseudocode

MST-KRUSKAL(G, w)

- 1 $A = \emptyset$
- 2 **for** each vertex $v \in G.V$
- 3 MAKE-SET(v)
- 4 sort the edges of $G.E$ into nondecreasing order by weight w
- 5 **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
- 6 **if** FIND-SET(u) \neq FIND-SET(v)
- 7 $A = A \cup \{(u, v)\}$
- 8 UNION(u, v)
- 9 **return** A

Laufzeitanalyse von Kruskals Algorithmus

Satz 13.10 —

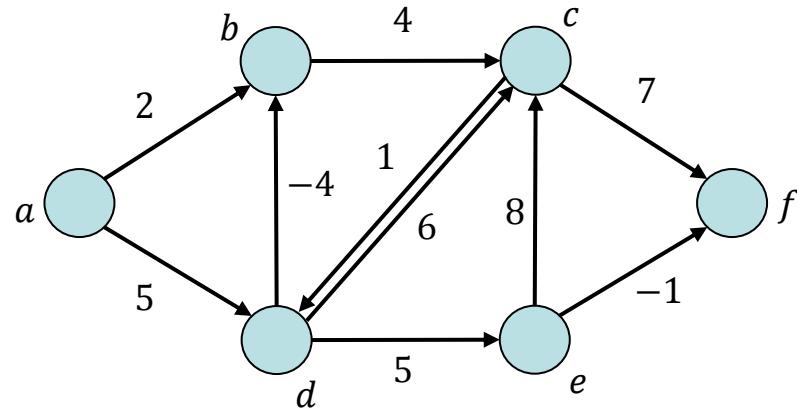
Werden verkettete Listen als Union-Find-Datenstruktur benutzt, so ist die Laufzeit von Kruskals Algorithmus $O(|V| \log(|V|) + |E| \log(|E|))$.

14. All Pairs Shortest Path (ASPS)

Eingabe: Gewichteter Graph $G = (V, E)$

Ausgabe: Für jedes Paar von Knoten $u, v \in V$ die Distanz von u nach v , sowie ein kürzester Weg

	a	b	c	d	e	f
a	0	1	5	5	10	9
b	∞	0	4	5	10	9
c	∞	-3	0	1	6	5
d	∞	-4	0	0	5	4
e	∞	5	8	9	0	-1
f	∞	∞	∞	∞	∞	0

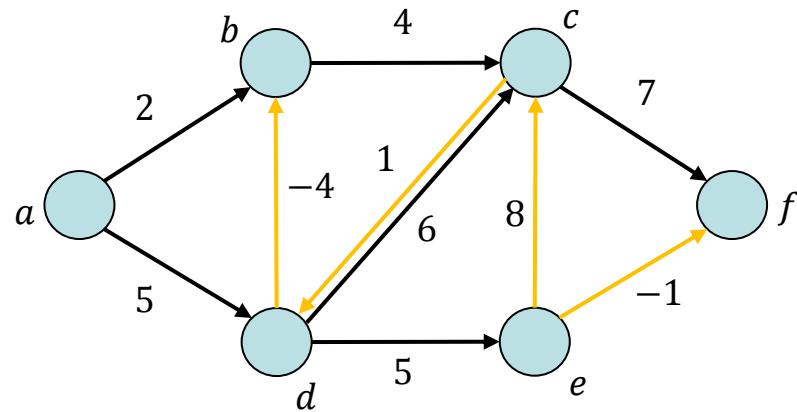


All Pairs Shortest Path (ASPS)

Eingabe: Gewichteter Graph $G = (V, E)$

Ausgabe: Für jedes Paar von Knoten $u, v \in V$ die Distanz von u nach v (und evtl. einen kürzesten Weg)

	a	b	c	d	e	f
a	0	1	5	5	10	9
b	∞	0	4	5	10	9
c	∞	-3	0	1	6	5
d	∞	-4	0	0	5	4
e	∞	5	8	9	0	-1
f	∞	∞	∞	∞	∞	0

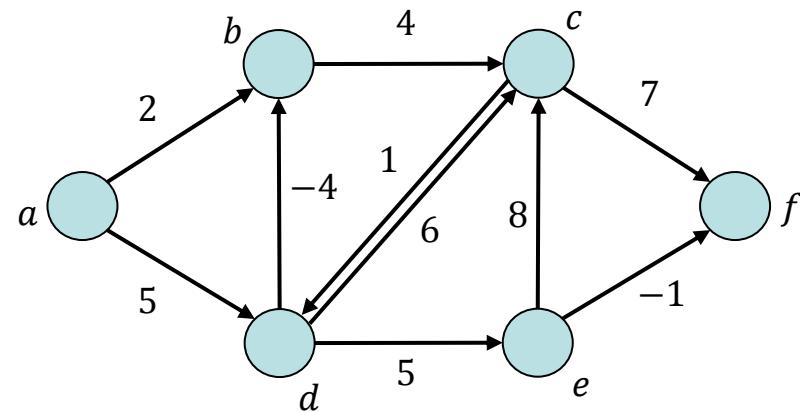


All Pairs Shortest Path (ASPS)

Matrix $W = (w_{ij})$, die Graphen repräsentiert

$$w_{ij} = \begin{cases} 0 & \text{wenn } i = j \\ \text{Gewicht der ger. Kante } (i, j) & \text{wenn } i \neq j \text{ und } (i, j) \in E \\ \infty & \text{wenn } i \neq j \text{ und } (i, j) \notin E \end{cases}$$

	a	b	c	d	e	f
a	0	2	∞	5	∞	∞
b	∞	0	4	∞	∞	∞
c	∞	∞	0	1	∞	7
d	∞	-4	6	0	5	∞
e	∞	∞	8	∞	0	-1
f	∞	∞	∞	∞	∞	0



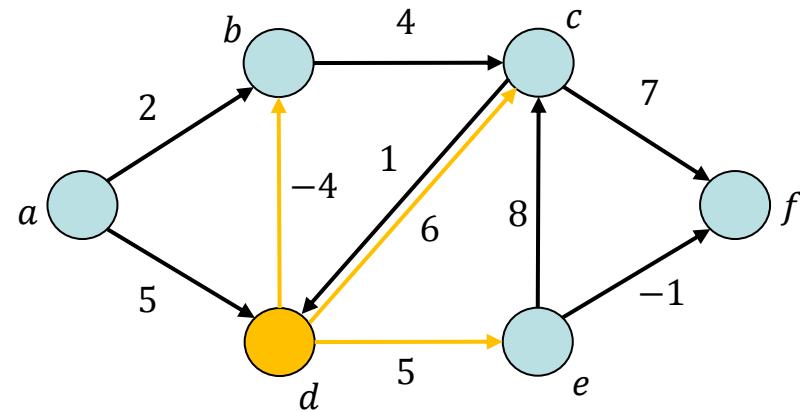
All Pairs Shortest Path (ASPS)

Matrix $W = (w_{ij})$, die Graphen repräsentiert

$$w_{ij} = \begin{cases} 0 & \text{wenn } i = j \\ \text{Gewicht der ger. Kante } (i, j) & \text{wenn } i \neq j \text{ und } (i, j) \in E \\ \infty & \text{wenn } i \neq j \text{ und } (i, j) \notin E \end{cases}$$

Annahme:
Keine negativen Zyklen!

	a	b	c	d	e	f
a	0	2	∞	5	∞	∞
b	∞	0	4	∞	∞	∞
c	∞	∞	0	1	∞	7
d	∞	-4	6	0	5	∞
e	∞	∞	8	∞	0	-1
f	∞	∞	∞	∞	∞	0

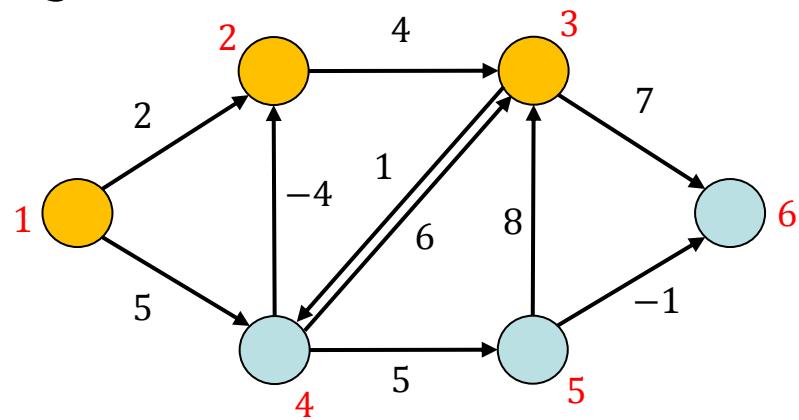


All Pairs Shortest Path (ASPS)

- Nummeriere Knoten von 1 bis $n = |V|$
- Betrachte kürzeste $i-j$ -Wege, die nur über Knoten 1 bis k laufen

	1	2	3	4	5	6
1	0	2	6	5	∞	13
2	∞	0	4	5	∞	11
3	∞	∞	0	1	∞	7
4	∞	-4	0	0	5	7
5	∞	∞	8	14	0	-1
6	∞	∞	∞	∞	∞	0

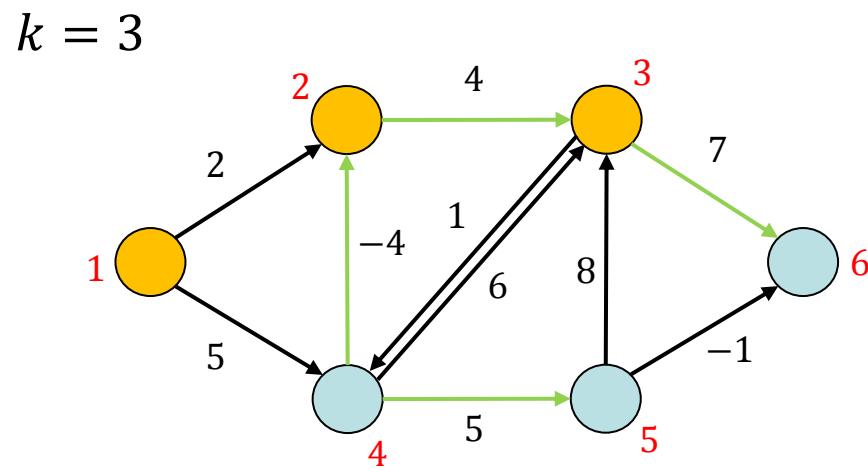
$$k = 3$$



All Pairs Shortest Path (ASPS)

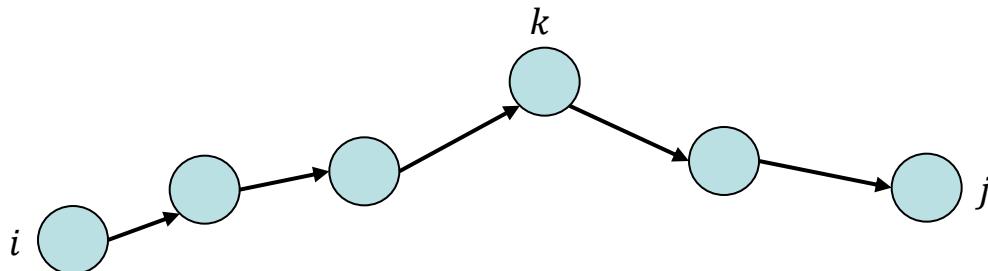
- Nummeriere Knoten von 1 bis $n = |V|$
- Betrachte kürzeste $i-j$ -Wege, die nur über Knoten 1 bis k laufen

	1	2	3	4	5	6
1	0	2	6	5	∞	13
2	∞	0	4	5	∞	11
3	∞	∞	0	1	∞	7
4	∞	-4	0	0	5	7
5	∞	∞	8	14	0	-1
6	∞	∞	∞	∞	∞	0



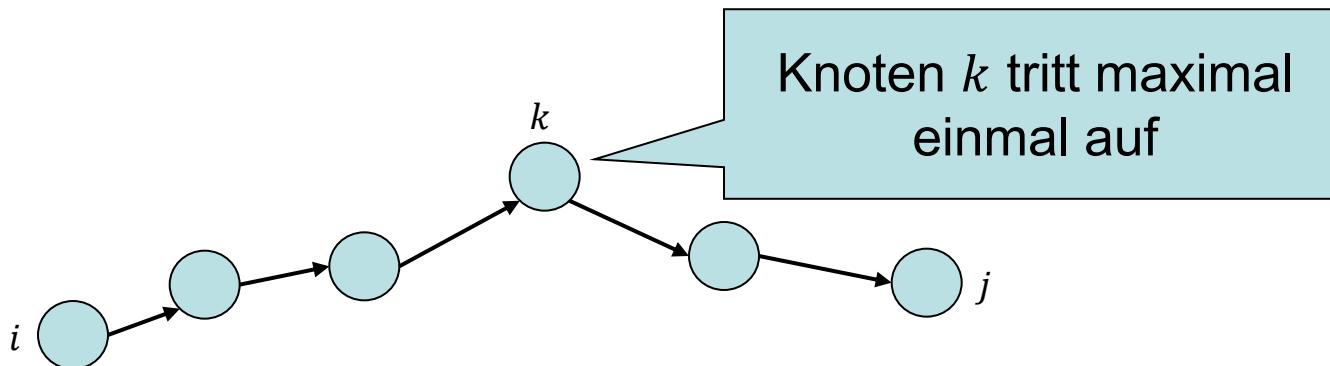
All Pairs Shortest Path (ASPS)

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten i - j -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte i - j -Weg, der nur über Knoten aus $\{1, \dots, k\}$ läuft:



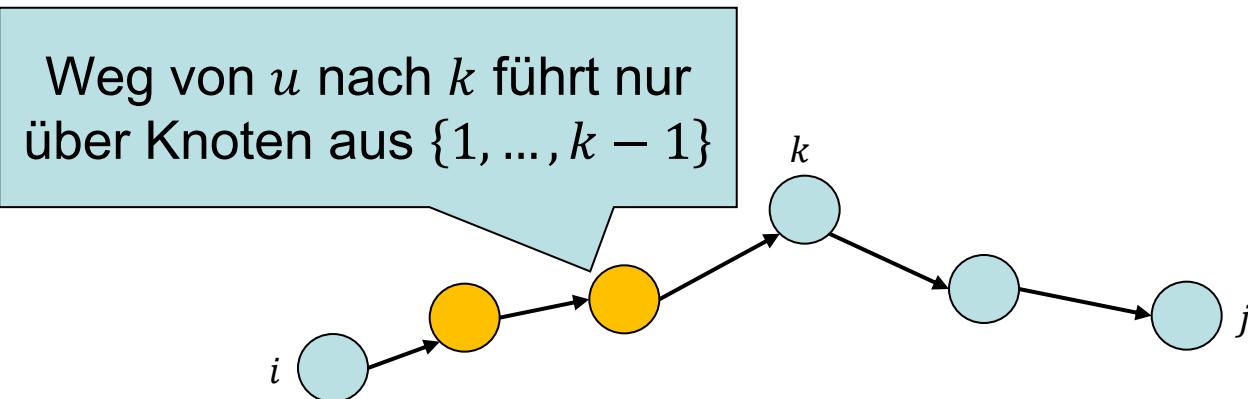
All Pairs Shortest Path (ASPS)

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten $i-j$ -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte $i-j$ -Weg, der nur über Knoten aus $\{1, \dots, k\}$ läuft:



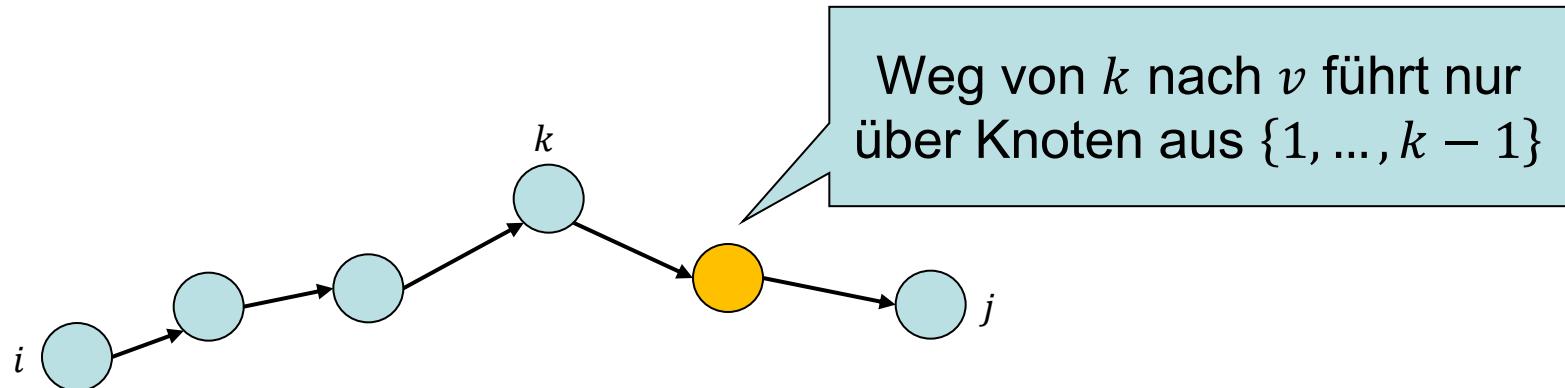
All Pairs Shortest Path (ASPS)

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten $i-j$ -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte $i-j$ -Weg, der nur über Knoten aus $\{1, \dots, k\}$ läuft:



All Pairs Shortest Path (ASPS)

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten i - j -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte i - j -Weg, der nur über Knoten aus $\{1, \dots, k\}$ läuft:

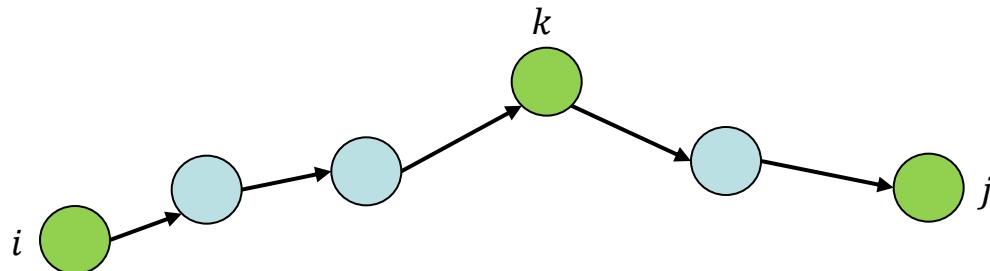


All Pairs Shortest Path (ASPS)

Kürzester i - j -Weg über Knoten aus $\{1, \dots, k\}$ ist:

- a) kürzester i - j -Weg über Knoten aus $\{1, \dots, k - 1\}$ oder
- b) kürzester i - j -Weg über Knoten aus $\{1, \dots, k - 1\}$ gefolgt von kürzestem k - j -Weg über Knoten aus $\{1, \dots, k - 1\}$

Fall b):



All Pairs Shortest Path (ASPS)

Die Rekursion:

- Sei $d_{ij}^{(k)}$ die Länge eines kürzesten i - j -Wegs mit Knoten aus $\{1, \dots, k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{falls } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{falls } k \geq 1 \end{cases}$$

- Matrix $D^{(n)} = d_{ij}^{(n)}$ enthält die gesuchte Lösung

All Pairs Shortest Path (ASPS)

FLOYD-WARSHALL(W)

```
1  $n = W.\text{rows}$ 
2  $D^{(0)} = W$ 
3 for  $k = 1$  to  $n$ 
4     let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5     for  $i = 1$  to  $n$ 
6         for  $j = 1$  to  $n$ 
7              $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8 return  $D^{(n)}$ 
```

All Pairs Shortest Path

Satz 14.1

Sei $G = (V, E)$ ein Graph mit nicht-negativen Zyklen.

Dann berechnet der Algorithmus von Floyd-Warshall die Entfernung zwischen jedem Knotenpaar in $O(|V|^3)$ Zeit.

All Pairs Shortest Path

Aufrechterhalten der kürzesten Wege:

- Konstruiere Vorgängermatrix Π
- Dazu konstruiere Sequenz $\Pi^{(1)}, \dots, \Pi^{(n)}$ mit $\Pi = \Pi^{(n)}$
- weiters ist $\Pi^{(k)}$ Vorgängermatrix zu $D^{(k)}$
- außerdem ist $\pi_{ij}^{(k)}$ Vorgänger von Knoten j auf dem kürzesten Weg von Knoten i über Knoten aus $\{1, \dots, k\}$
- Die Startmatrix:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{falls } i = j \text{ oder } w_{ij} = \infty \\ i & \text{falls } i \neq j \text{ und } w_{ij} < \infty \end{cases}$$

All Pairs Shortest Path

Aufrechterhalten der kürzesten Wege:

- Konstruiere Vorgängermatrix Π
- Dazu konstruiere Sequenz $\Pi^{(1)}, \dots, \Pi^{(n)}$ mit $\Pi = \Pi^{(n)}$
- weiters ist $\Pi^{(k)}$ Vorgängermatrix zu $D^{(k)}$
- außerdem ist $\pi_{ij}^{(k)}$ Vorgänger von Knoten j auf dem kürzesten Weg von Knoten i über Knoten aus $\{1, \dots, k\}$
- Das Aufrechterhalten:

$$\pi_{ij}^{(0)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{falls } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{falls } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Divide & Conquer

Teile und herrsche:

- Problem in Teilprobleme aufteilen
- Teilprobleme rekursiv lösen
- Lösung aus Teillösungen zusammensetzen

Probleme:

- Wie setzt man zusammen?
 - erfordert algorithmisches Geschick und Übung
- Laufzeitanalyse (Auflösen der Rekursion)
 - ist normalerweise nach Standardschema; erfordert ebenfalls Übung

Divide & Conquer

Teile und herrsche:

- Problem in Teilprobleme aufteilen
- Teilprobleme rekursiv lösen
- Lösung aus Teillösungen zusammensetzen

Probleme:

- Merge-Sort
- Quick-Sort

Matrix-Multiplikation

$$\begin{pmatrix} 3 & 7 & 5 & 4 \\ 0 & 3 & 2 & 4 \\ 10 & 2 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 0 & 1 & 3 \\ 3 & 1 & 1 & 0 \\ 2 & 3 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 29 & 20 & 23 & 29 \\ 14 & 14 & 13 & 17 \\ 23 & 11 & 13 & 6 \\ 5 & 2 & 2 & 0 \end{pmatrix}$$

Matrix-Multiplikation

Teile und herrsche:

- Problem: Berechne das Produkt zweier $n \times n$ Matrizen
- Eingabe: Matrizen X, Y
- Ausgabe: Matrix $Z = X \cdot Y$

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \end{pmatrix}, Y = \begin{pmatrix} y_{1,1} & y_{1,2} & y_{1,3} & y_{1,4} \\ y_{2,1} & y_{2,2} & y_{2,3} & y_{2,4} \\ y_{3,1} & y_{3,2} & y_{3,3} & y_{3,4} \\ y_{4,1} & y_{4,2} & y_{4,3} & y_{4,4} \end{pmatrix}$$

Matrix-Multiplikation

SQUARE-MATRIX-MULTIPLY(A, B)

1	$n = A.\text{rows}$	$\Theta(n^2)$
2	let C be a new $n \times n$ matrix	
3	for $i = 1$ to n	$\Theta(n)$
4	for $j = 1$ to n	$\Theta(n^2)$
5	$c_{ij} = 0$	$\Theta(n^2)$
6	for $k = 1$ to n	$\Theta(n^3)$
7	$c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$	$\Theta(n^3)$
8	return C	$\Theta(1)$
		<hr/> $\Theta(n^3)$

Laufzeit:

Matrix-Multiplikation

Teile und herrsche:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Aufwand:

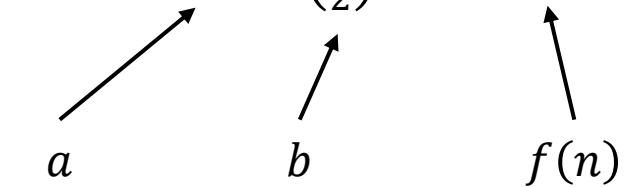
- 8 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$ Matrizen
- 4 Additionen von $\frac{n}{2} \times \frac{n}{2}$ Matrizen

Laufzeit:

- $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2)$

Matrix-Multiplikation

Laufzeit:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + k \cdot n^2$$


Master Theorem:

$$f(n) = k \cdot n^2$$

Das Master-Theorem

Satz (Master-Theorem für Rekursionsgleichungen)

Seien $a, b > 1$ Konstanten, sei $f(n)$ eine Funktion und sei $T(n)$ definiert durch die Rekursionsgleichung

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

Hierbei kann $\frac{n}{b}$ auch durch $\left\lfloor \frac{n}{b} \right\rfloor$ oder $\left\lceil \frac{n}{b} \right\rceil$ ersetzt werden.

Dann kann $T(n)$ folgendermaßen abgeschätzt werden:

1. Ist $f(n) = O(n^{\log_b(a)-\varepsilon})$ für $\varepsilon > 0$, dann gilt $T(n) = O(n^{\log_b(a)})$.
2. Ist $f(n) = \Theta(n^{\log_b(a)})$, dann gilt $T(n) = O(n^{\log_b(a)} \cdot \log(n))$.
3. Ist $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ für $\varepsilon > 0$, und ist $af\left(\frac{n}{b}\right) \leq cf(n)$ für eine Konstante $c < 1$ und $n \rightarrow \infty$, dann gilt $T(n) = \Theta(f(n))$.

Matrix-Multiplikation

Laufzeit:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + k \cdot n^2$$

The diagram illustrates the components of the recurrence relation. It shows three parts: 'a' pointing to the recursive call $T\left(\frac{n}{2}\right)$, 'b' pointing to the quadratic term n^2 , and $f(n)$ pointing to the full formula $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + k \cdot n^2$.

Master Theorem:

- $f(n) = k \cdot n^2$
- Fall 1: Laufzeit $\Theta(n^{\log_b a}) = \Theta(n^3)$
- Formaler Beweis durch Induktion!!
- **Nicht besser als einfacher Algorithmus!**

Matrix-Multiplikation

Teile und herrsche (Algorithmus von Strassen):

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Trick:

$$P_1 = A \cdot (F - H)$$

$$P_5 = (A + D) \cdot (E + H)$$

$$P_2 = (A + B) \cdot H$$

$$P_6 = (B - D) \cdot (G + H)$$

$$P_3 = (C + D) \cdot E$$

$$P_7 = (A - C) \cdot (E + F)$$

$$P_4 = D \cdot (G - E)$$

$$AE + BG = P_5 + P_4 - P_2 + P_6$$

$$AF + BH = P_1 + P_2$$

$$CE + DG = P_3 + P_4$$

$$CF + DH = P_5 + P_1 - P_3 - P_7$$

Matrix-Multiplikation

Teile und herrsche:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Trick:

$$P_1 = A \cdot (F - H)$$

$$P_5 = (A + D) \cdot (E + H)$$

$$P_2 = (A + B) \cdot H$$

$$P_6 = (B - D) \cdot (G + H)$$

$$P_3 = (C + D) \cdot E$$

$$P_7 = (A - C) \cdot (E + F)$$

$$P_4 = D \cdot (G - E)$$

$$AE + BG = P_5 + P_4 - P_2 + P_6$$

$$AF + BH = P_1 + P_2$$

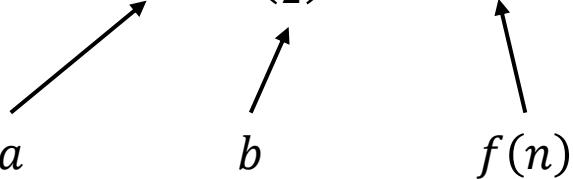
$$CE + DG = P_3 + P_4$$

$$CF + DH = P_5 + P_1 - P_3 - P_7$$

7 Multiplikationen!!!

Matrix-Multiplikation

Laufzeit:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + k \cdot n^2$$


Master Theorem:

- $f(n) = k \cdot n^2$
- Fall 1: Laufzeit $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 7}) = O(n^{2,81})$
- Verbesserter Algorithmus!
(erheblicher Unterschied für große Eingaben)

Matrix-Multiplikation

Satz 15.1

Das Produkt zweier $n \times n$ Matrizen kann in $O(n^{2,81})$ Laufzeit berechnet werden.

16. Gierige Algorithmen

- Gierige Algorithmen sind eine Algorithmenmethode, um sogenannte **Optimierungsprobleme** zu lösen.
- Bei einem Optimierungsproblem gibt es zu jeder *Probleminstanz* viele mögliche oder **zulässige Lösungen**.
Lösungen haben **Werte** gegeben durch eine **Zielfunktion**.
Gesucht ist dann eine möglichst gute zulässige Lösung.
- Also eine Lösung mit möglichst kleinem oder möglichst großem Wert (**Minimierungs-** bzw. **Maximierungsproblem**).

Gierige Algorithmen – Minimale Spannbäume

- Beispiel: Minimale Spannbäume

1. Probleminstanz

gewichteter, ungerichteteter, zusammenhängender Graph

2. Zulässige Lösungen

Spannbäume

3. Zielfunktion

Gewicht eines Spannbaums

4. Gesucht

Spannbaum minimalen Gewichts

Gierige Algorithmen – Idee und Prim

1. Gierige Algorithmen bestimmen Lösung durch sukzessives Erweitern von bereits gefundenen Teillösungen.
2. Erweitern geschieht durch lokal optimale Wahlen.
3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führen.

1. Algorithmus von Prim bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.
2. Prims Algorithmus wählt möglichst leichte Kante, die isolierten Knoten mit Teilbaum verbindet.
3. Analyse mit Hilfe von Schnitten zeigte, dass Teilbaum immer in einem minimalen Spannbaum enthalten ist.

Gierige Algorithmen – Idee und Kruskal

1. Gierige Algorithmen bestimmen Lösung durch sukzessives Erweitern von bereits gefundenen Teillösungen.
2. Erweitern geschieht durch lokal optimale Wahlen.
3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führen.

1. Algorithmus von Kruskal bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.
2. Kruskals Algorithmus wählt möglichst leichte Kante, die Zusammenhangskomponenten verbindet.
3. Analyse mit Hilfe von Schnitten zeigte, dass Teilbaum immer in einem minimalen Spannbaum enthalten ist.

Gieriges 1-Prozessor-Scheduling

- Gegeben sind n Jobs j_1, \dots, j_n mit **Dauer** t_1, \dots, t_n .
- Jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden. Der Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten.
Ein einmal begonnener Job darf nicht abgebrochen werden.
- Gesucht ist eine Reihenfolge, in der die Jobs abgearbeitet werden, sodass die **durchschnittliche Bearbeitungszeit** der Jobs möglichst **gering** ist.

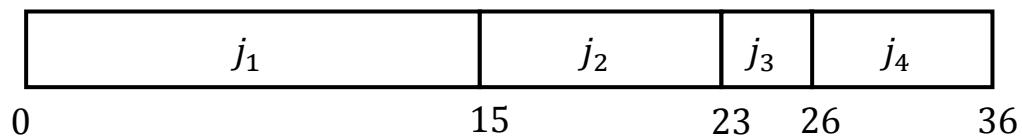
Gieriges 1-Prozessor-Scheduling

- **Bearbeitungszeit** eines Jobs ist der **Zeitpunkt**, an dem der Job **vollständig bearbeitet** wurde.
- Werden die Jobs in Reihenfolge $j_{\pi(1)}, \dots, j_{\pi(n)}$ ausgeführt, wobei π eine Permutation ist, so ist die Bearbeitungszeit von Job $j_{\pi(1)}$ genau $t_{\pi(1)}$, die Bearbeitungszeit von Job $j_{\pi(2)}$ ist $t_{\pi(1)} + t_{\pi(2)}$, usw.

Gieriges 1-Prozessor-Scheduling – Beispiel

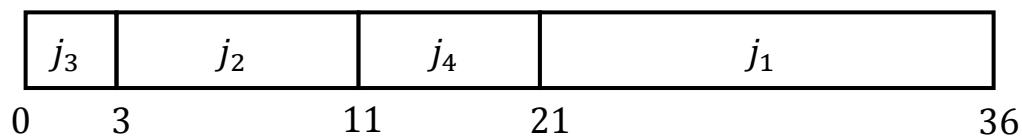
Job	Laufzeit
j_1	15
j_2	8
j_3	3
j_4	10

Schedule Nr. 1:



Durchschnittliche Beendigung: 25

Schedule Nr. 2:



Durchschnittliche Beendigung: 17,75

Gieriges 1-Prozessor-Scheduling

Lemma 16.1

Bei Permutation π ist die durchschnittliche Bearbeitungszeit gegeben durch

$$\frac{1}{n} \sum_{i=1}^n (n - i + 1) t_{\pi(i)}$$

Lemma 16.2

Eine Permutation π führt genau dann zu einer minimalen durchschnittlichen Bearbeitungszeit, wenn die Folge $(t_{\pi(1)}, \dots, t_{\pi(n)})$ aufsteigend sortiert ist.

Gieriges Mehr-Prozessor-Scheduling

- Gegeben sind n Jobs j_1, \dots, j_n mit **Dauer** t_1, \dots, t_n und m identische Prozessoren.
- Jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden. Jeder Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten. Ein einmal begonnener Job darf nicht abgebrochen werden.
- Gesucht ist eine Aufteilung der Jobs auf die Prozessoren und für jeden Prozessor eine Reihenfolge der ihm zugewiesenen Jobs, sodass die durchschnittliche Bearbeitungszeit der Jobs möglichst gering ist.
- Bearbeitungszeit eines Jobs wie vorher definiert.

Gieriges Mehr-Prozessor-Scheduling

Lemma 16.3

Die Permutation π sei so gewählt, dass die Folge $(t_{\pi(1)}, \dots, t_{\pi(n)})$ aufsteigend sortiert ist.

Weiter werde dann Job $j_{\pi(i)}$ auf dem Prozessor mit Nummer $i \bmod m$ ausgeführt.

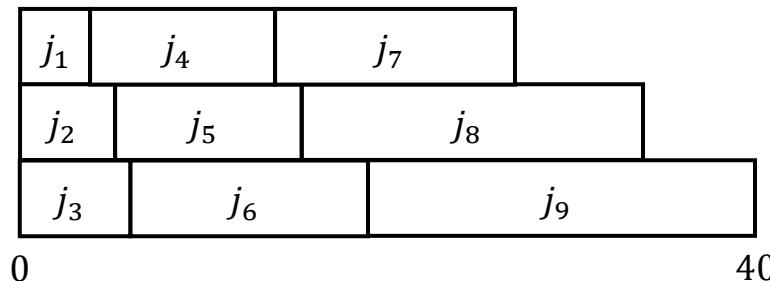
Das so konstruierte Scheduling minimiert dann die durchschnittliche Bearbeitungszeit.

Hierbei identifizieren wir 0 und m , d.h. ist $i \bmod m = 0$, so wird Job $j_{\pi(i)}$ auf dem Prozessor mit Nummer m gestartet.

Gieriges Mehr-Prozessor-Scheduling – Beispiel

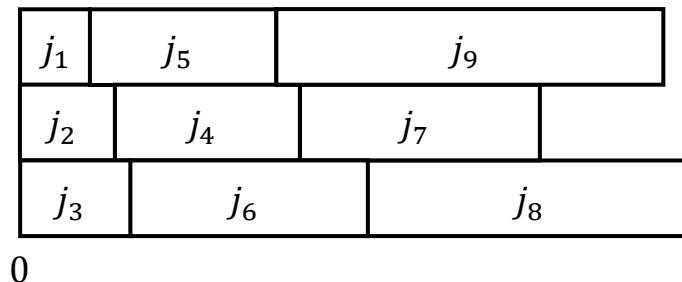
Job	Laufzeit
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

Schedule Nr. 1:



Durchschnittliche Beendigung: 18,33

Schedule Nr. 2:



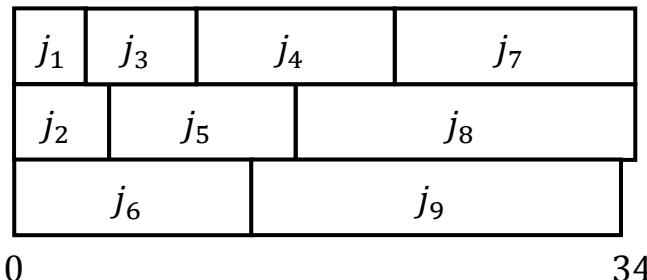
Durchschnittliche Beendigung: 18,33

Gierig ist nicht immer optimal

Job	Laufzeit
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

- Betrachten dasselbe Szenario wie vorher mit Jobs und Prozessoren.
- Wollen aber jetzt den Zeitpunkt minimieren, an dem alle Jobs beendet sind.

Schedule Nr. 3:



Durchschnittliche Beendigung: 18,55

Gierige (Greedy) Algorithmen

Gierige Strategie für Optimierungsprobleme:

- Aufbau einer Lösung in „kleinen“ Schritten
- In jedem dieser Schritte wird entsprechend eines definierten Optimierungskriteriums eine irreversible Entscheidung getroffen

Frage 1:

Wann kann eine solche Strategie zu einer optimalen Lösung führen?

Frage 2:

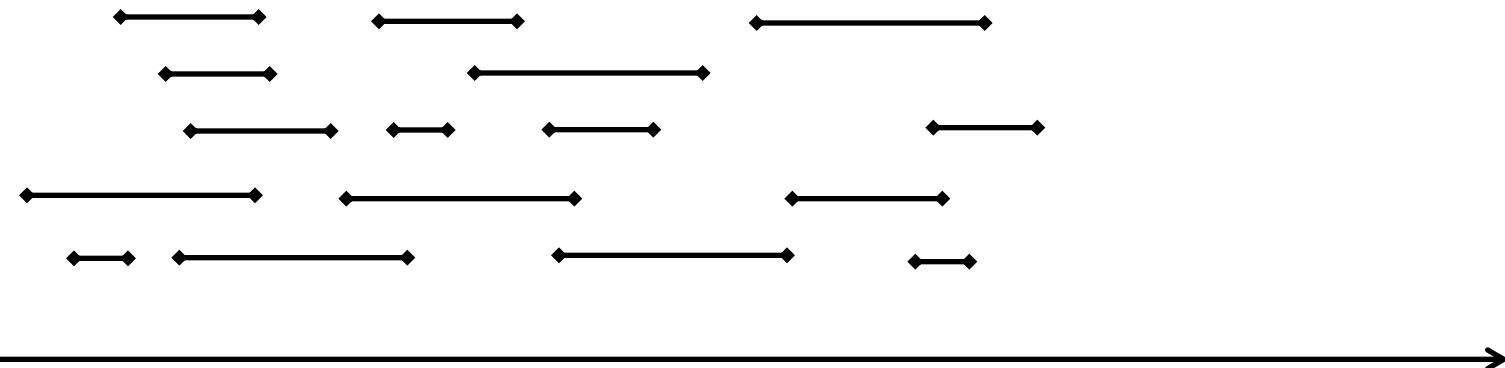
Wie beweist man, dass ein gieriger Algorithmus eine optimale Lösung liefert?

Anwendungsbeispiel: Intervall Scheduling

Gierige Algorithmen – Intervall Scheduling

Intervall Scheduling:

- eine Ressource (Hörsaal, Parallelrechner, ...)
- Menge von Anfragen der Art:
Kann ich die Ressource für den Zeitraum $[t_1, t_2]$ nutzen?

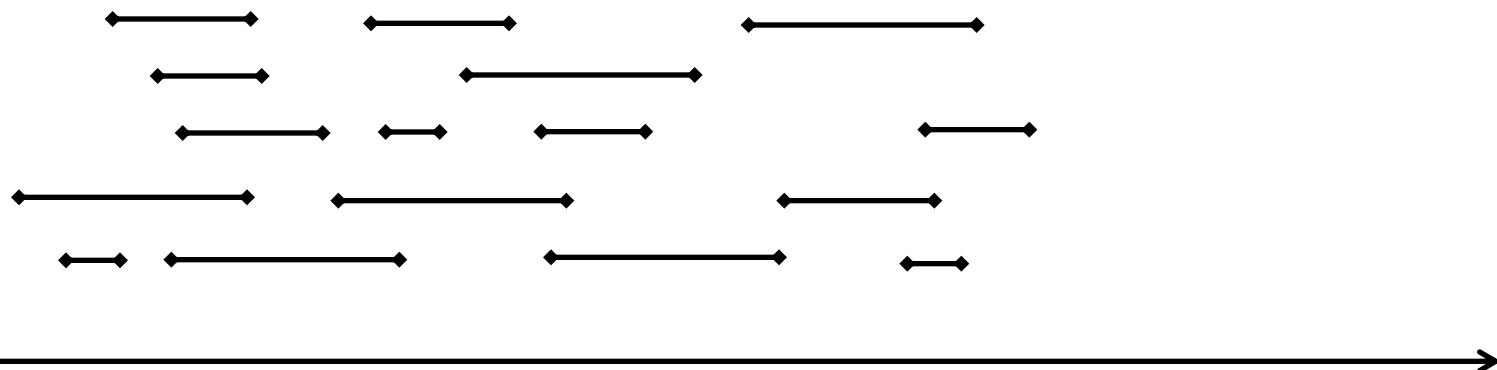


- Optimierungs-Ziel: möglichst viele Anfragen erfüllen

Gierige Algorithmen – Intervall Scheduling

Definition

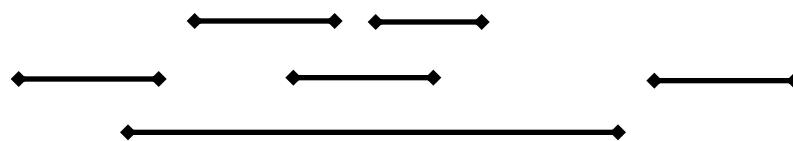
Zwei Anfragen heißen **kompatibel**, wenn sich die Intervalle nicht überschneiden.



Gierige Algorithmen – Intervall Scheduling

Generelle Überlegung:

- Wähle erste Anfrage i_1 „geschickt“
- Ist i_1 akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage i_2 und weise alle Anfragen zurück, die nicht mit i_2 kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



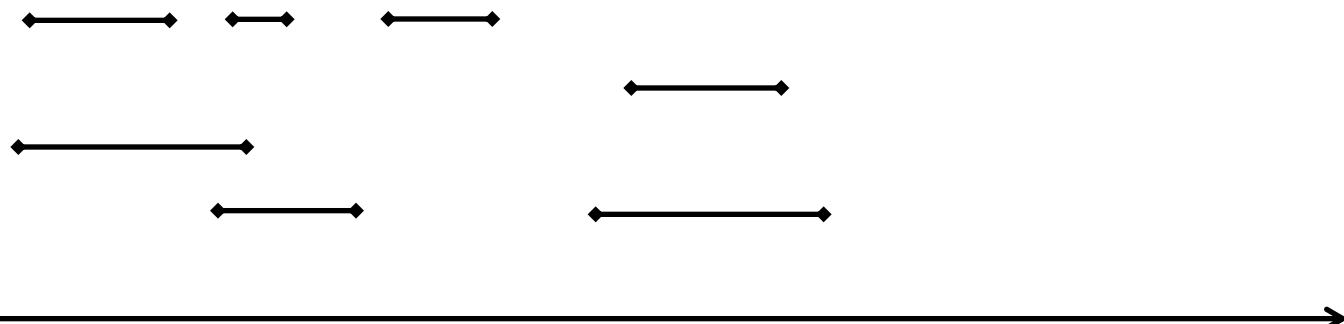
Gierige Algorithmen – Intervall Scheduling

Was ist nun aber eine „geschickte“ Auswahlstrategie?

Gierige Algorithmen – Intervall Scheduling

Strategie 1:

- Wähle immer die Anfrage, die am frühesten beginnt

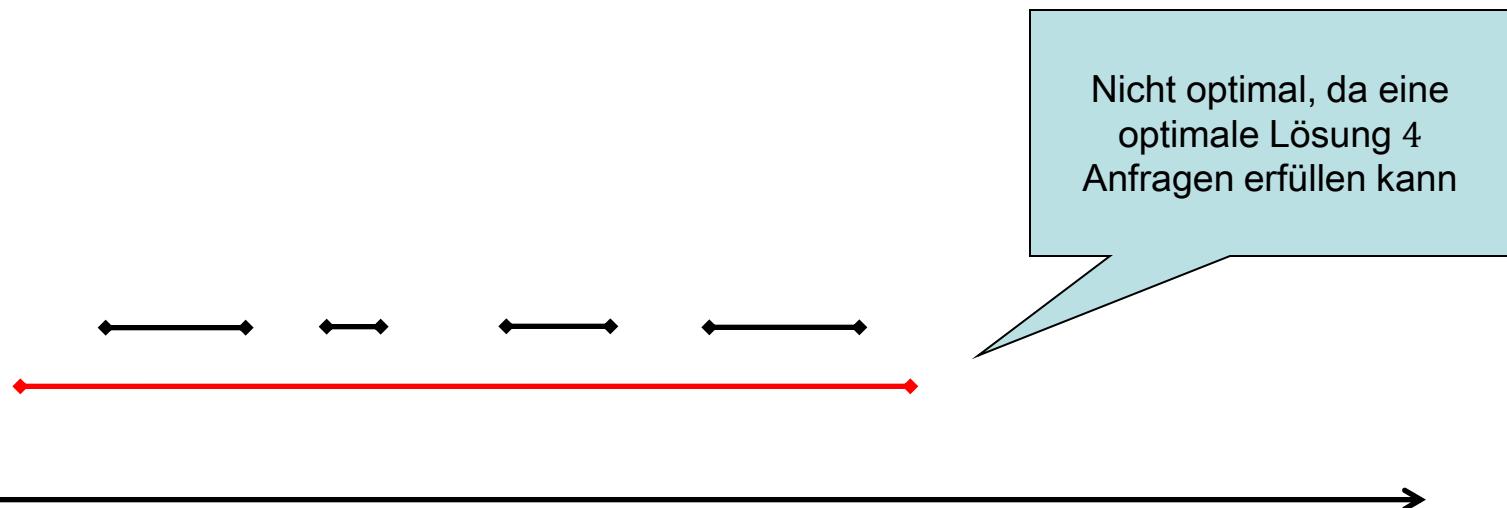


Gierige Algorithmen – Intervall Scheduling

Strategie 1:

- Wähle immer die Anfrage, die am frühesten beginnt

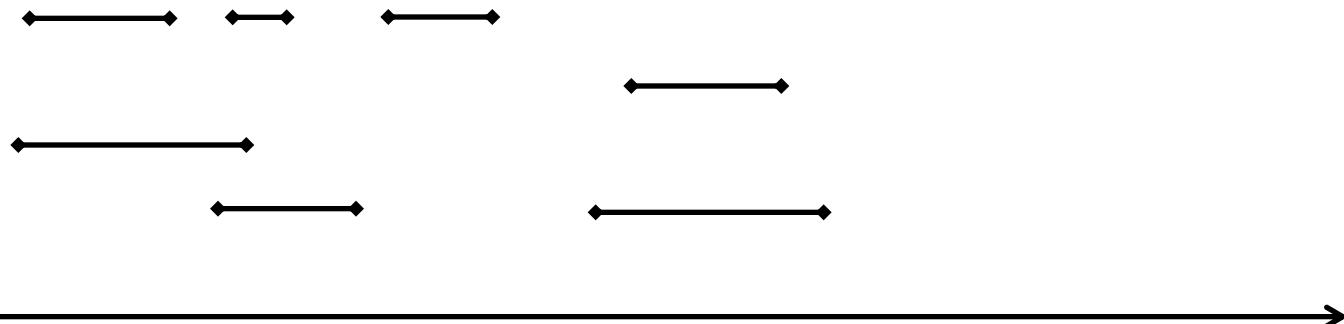
Optimalität?



Gierige Algorithmen – Intervall Scheduling

Strategie 2:

- Wähle immer das kürzeste Intervall

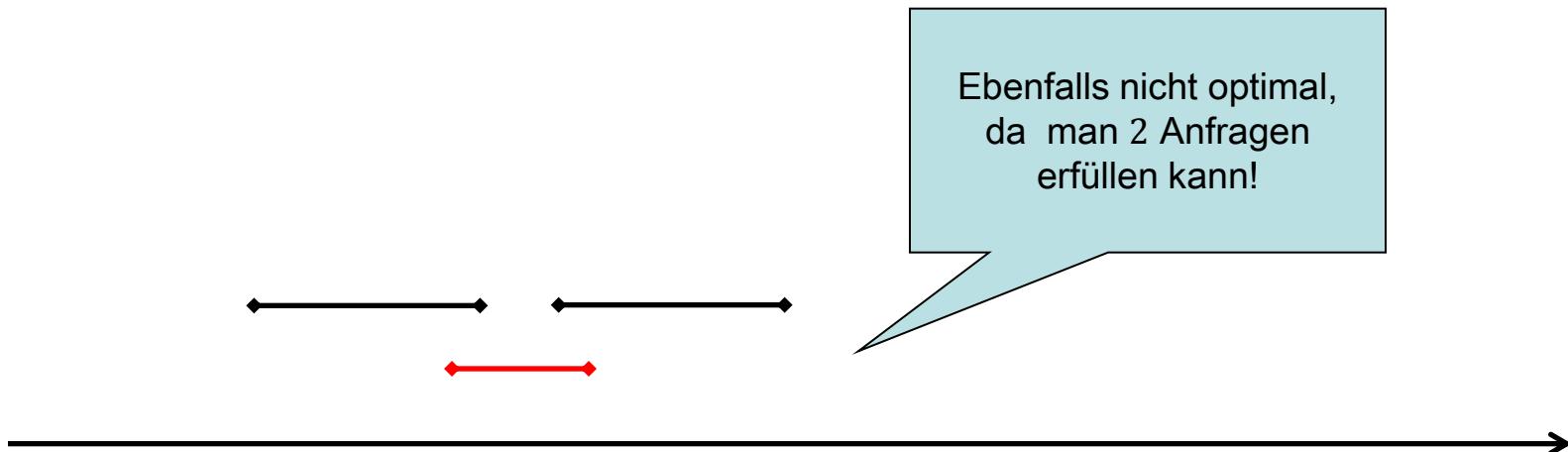


Gierige Algorithmen – Intervall Scheduling

Strategie 2:

- Wähle immer das kürzeste Intervall

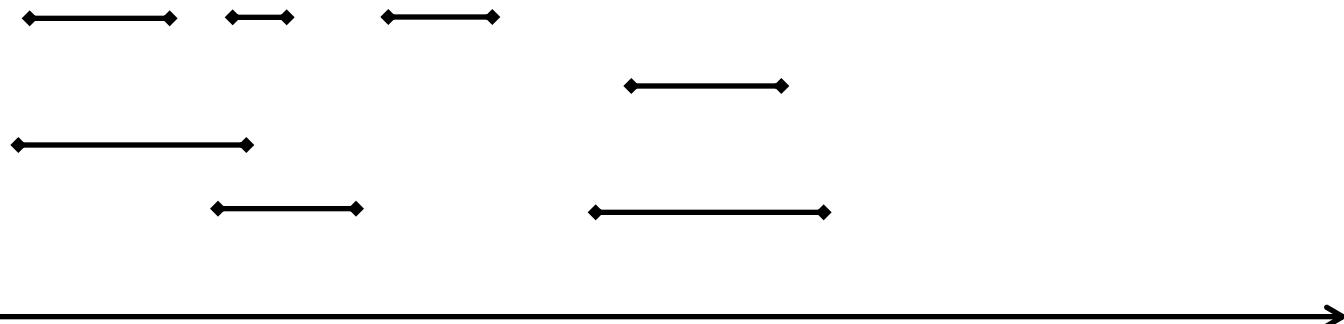
Optimalität?



Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

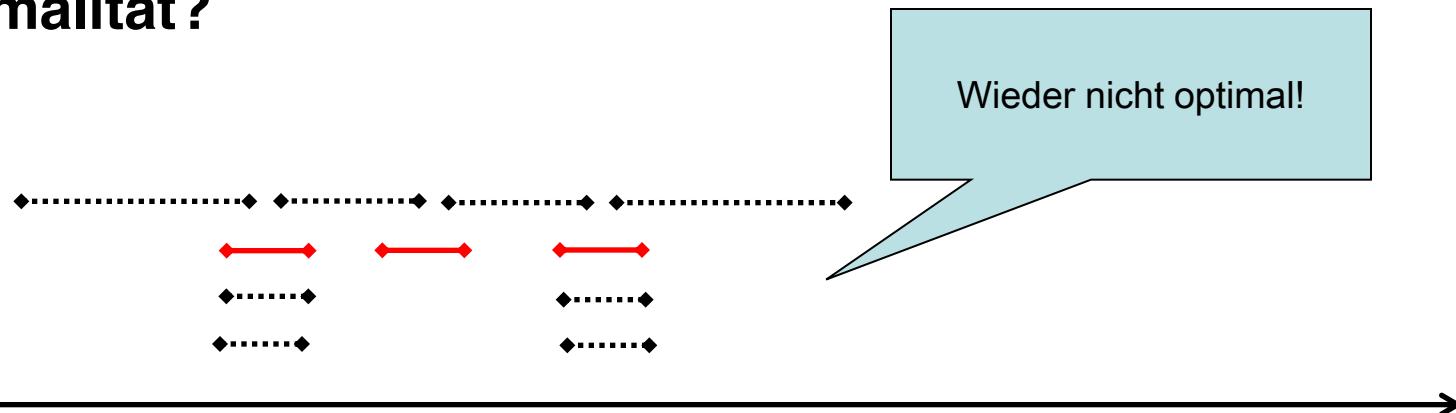


Gierige Algorithmen – Intervall Scheduling

Strategie 3:

- Wähle immer das Intervall mit den wenigsten nicht kompatiblen Intervallen
- bei Gleichheit wähle das kürzeste Intervall

Optimalität?



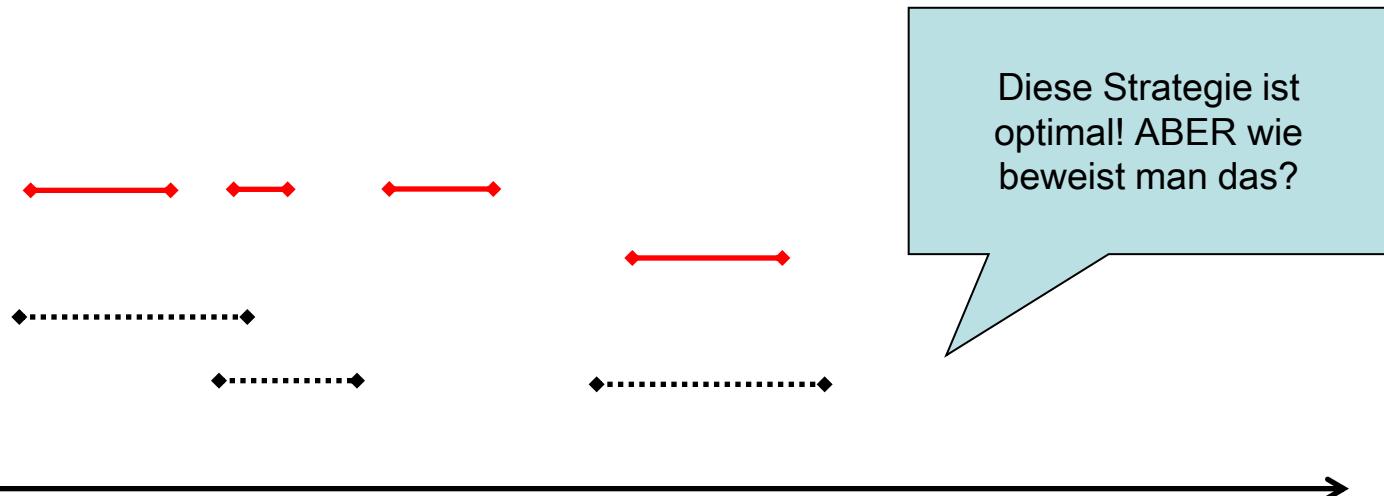
Gierige Algorithmen – Intervall Scheduling

Worauf muss man achten?

- Ressource muss möglichst früh wieder frei werden!

Neue Strategie:

- Nimm die Anfrage, die am frühesten fertig ist!



Gierige Algorithmen – Intervall Scheduling

Formale Problemformulierung:

- **Problem:** Intervall Scheduling
- **Eingabe:** Felder s und f , die die Intervalle $[s[i], f[i]]$ beschreiben
- **Ausgabe:** Indizes der ausgewählten Intervalle

o.B.d.A. nehmen wir an:

- Eingabe ist nach Intervallendpunkten sortiert, d.h.:

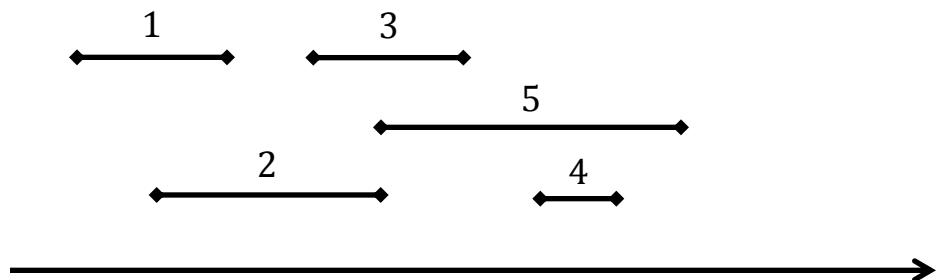
$$f[1] \leq f[2] \leq \dots \leq f[n]$$

Gierige Algorithmen – Intervall Scheduling

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  $n = s.length$ 
2  $A = \{a_1\}$ 
3  $k = 1$ 
4 for  $m = 2$  to  $n$ 
5   if  $s[m] \geq f[k]$ 
6      $A = A \cup \{a_m\}$ 
7      $k = m$ 
8 return  $A$ 
```

s	1	2	4	7	5
f	3	5	6	8	9

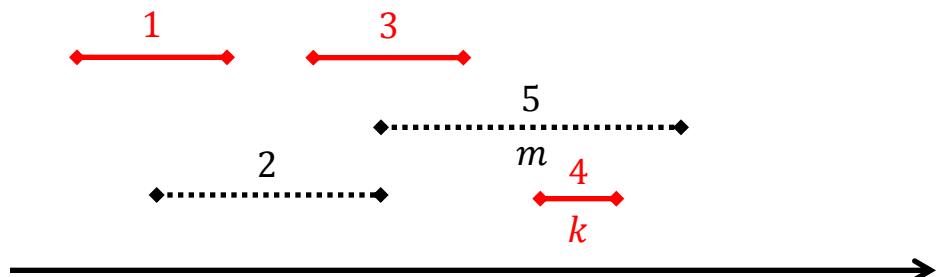


Gierige Algorithmen – Intervall Scheduling

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  $n = s.length$ 
2  $A = \{a_1\}$ 
3  $k = 1$ 
4 for  $m = 2$  to  $n$ 
5   if  $s[m] \geq f[k]$ 
6      $A = A \cup \{a_m\}$ 
7      $k = m$ 
8 return  $A$ 
```

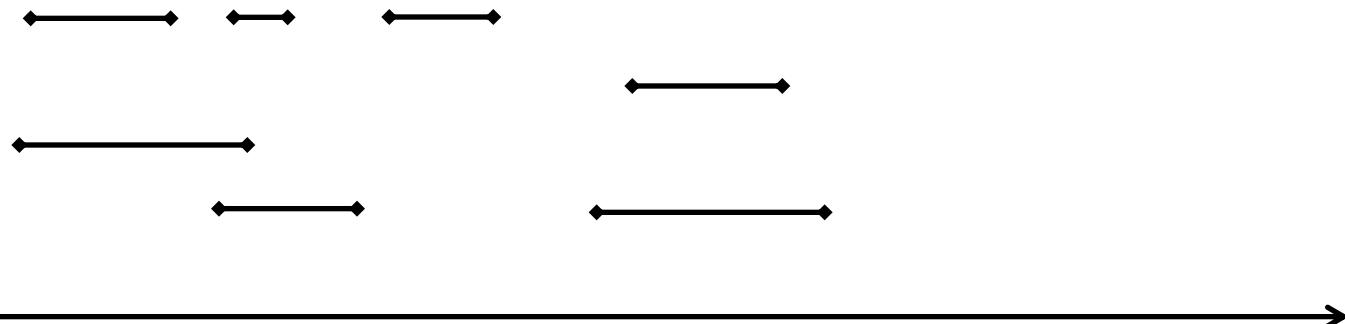
s	1	2	4	7	5
f	3	5	6	8	9



Gierige Algorithmen – Intervall Scheduling

Wie können wir Optimalität zeigen?

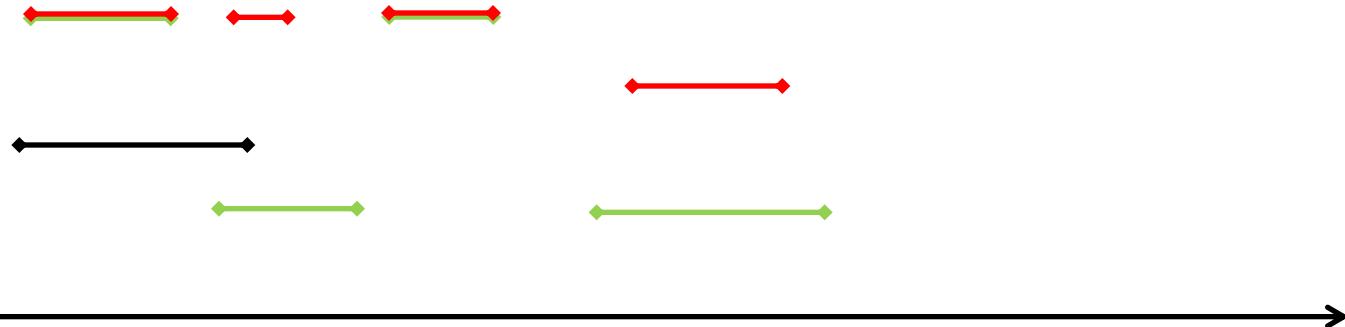
- Sei O optimale Menge von Intervallen
- Können wir $A = O$ zeigen?



Gierige Algorithmen – Intervall Scheduling

Wie können wir Optimalität zeigen?

- Sei O optimale Menge von Intervallen
- u. U. viele optimale Lösungen
⇒ **wir zeigen:** $|A| = |O|$



Gierige Algorithmen – Intervall Scheduling

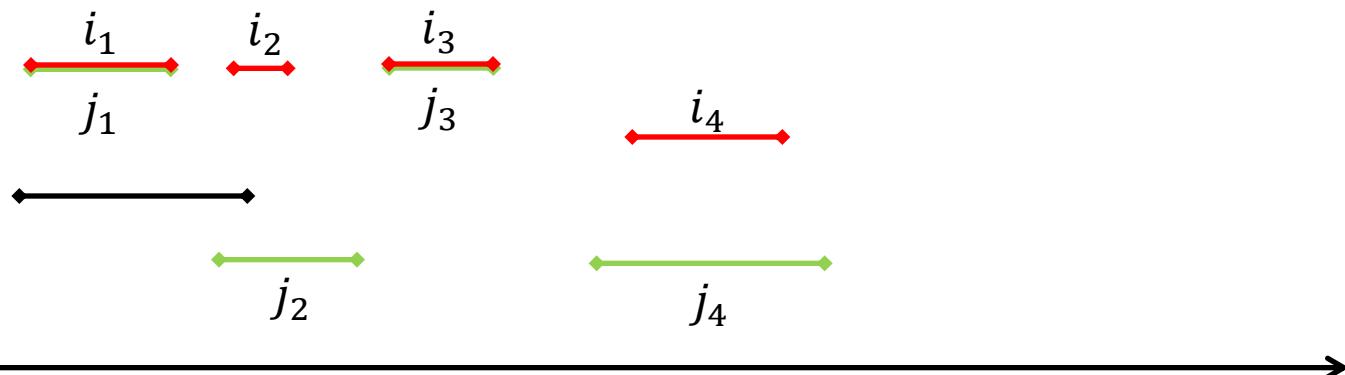
Beobachtung:

- A ist eine Menge von kompatiblen Anfragen, d.h. die Menge A bildet eine zulässige resp. gültige Lösung

Gierige Algorithmen – Intervall Scheduling

Notation:

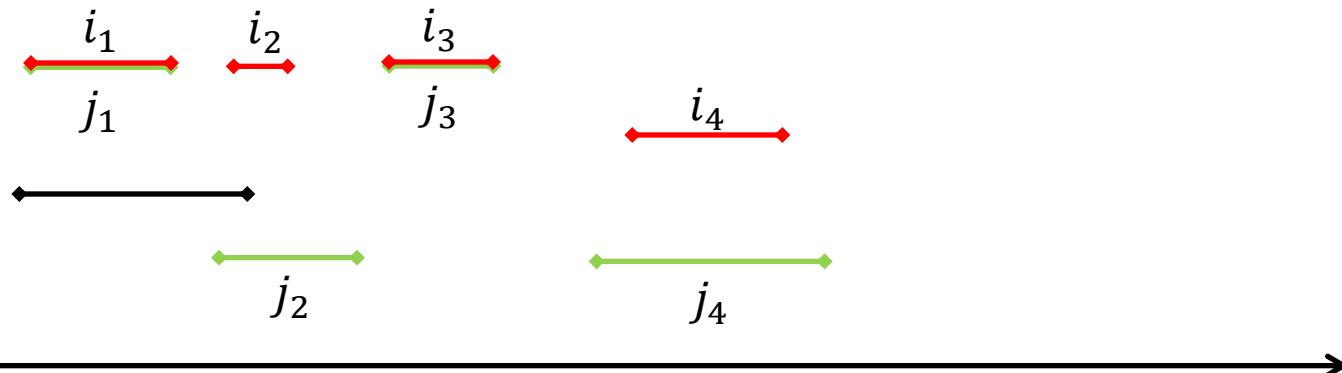
- i_1, \dots, i_k Intervalle von A in Ordnung des Hinzufügens
- j_1, \dots, j_m Intervalle von O sortiert nach Endpunkt



Gierige Algorithmen – Intervall Scheduling

Wir zeigen: Der gierige Algorithmus „liegt vorn“:

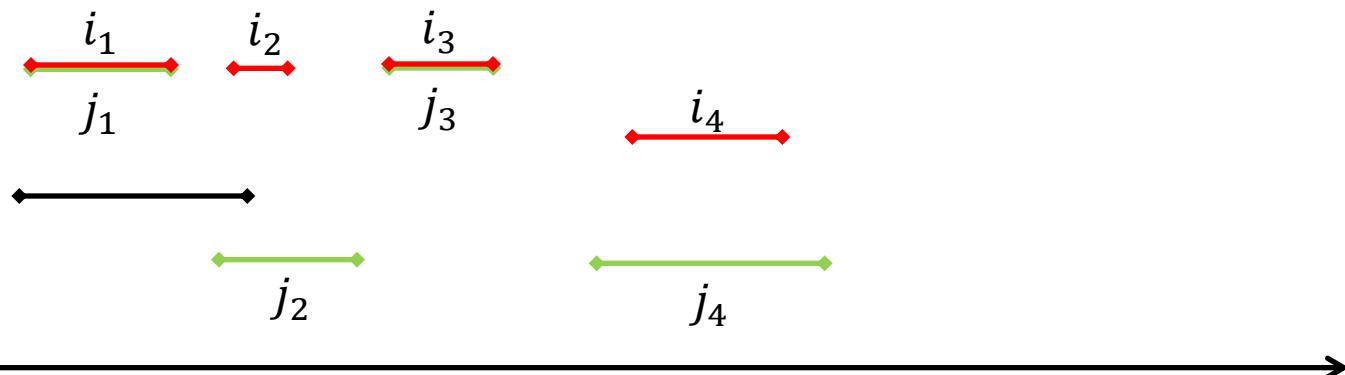
- d.h. jedes Intervall in A gibt die Ressource mindestens so früh wieder frei, wie das korrespondierende Intervall in O
- Dies ist wahr für das erste Intervallpaar: $f[i_1] \leq f[j_1]$
- Zu zeigen: dies gilt für alle anderen Intervallpaare!



Gierige Algorithmen – Intervall Scheduling

Lemma 16.4

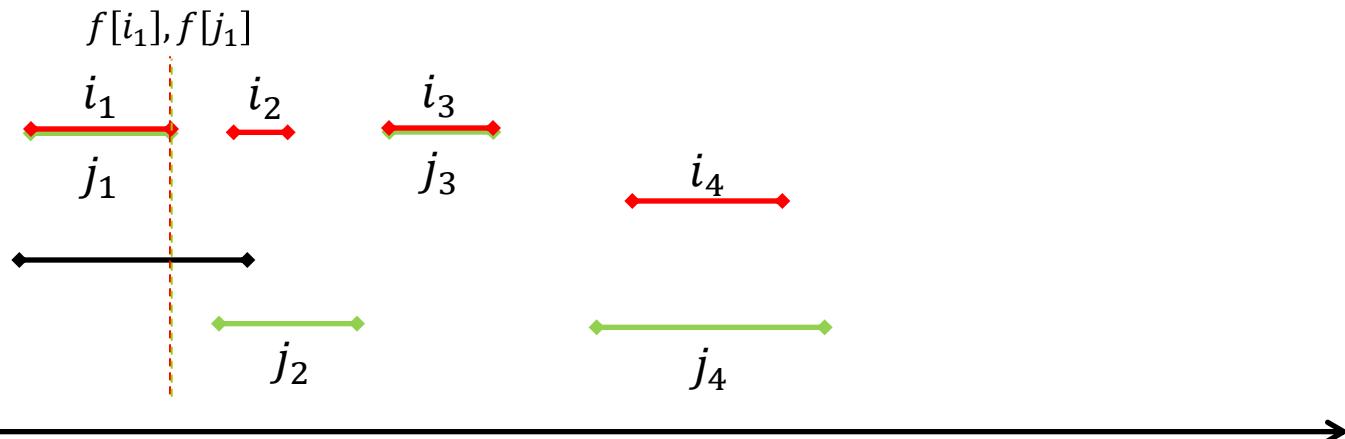
Für alle $r \leq k$ gilt $f[i_r] \leq f[j_r]$.



Gierige Algorithmen – Intervall Scheduling

Lemma 16.4

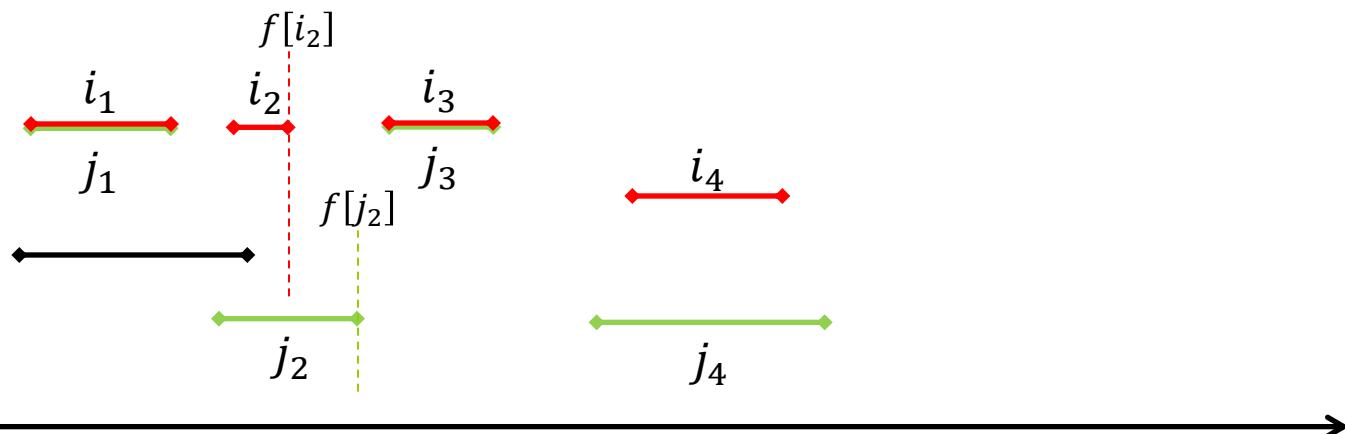
Für alle $r \leq k$ gilt $f[i_r] \leq f[j_r]$.



Gierige Algorithmen – Intervall Scheduling

Lemma 16.4

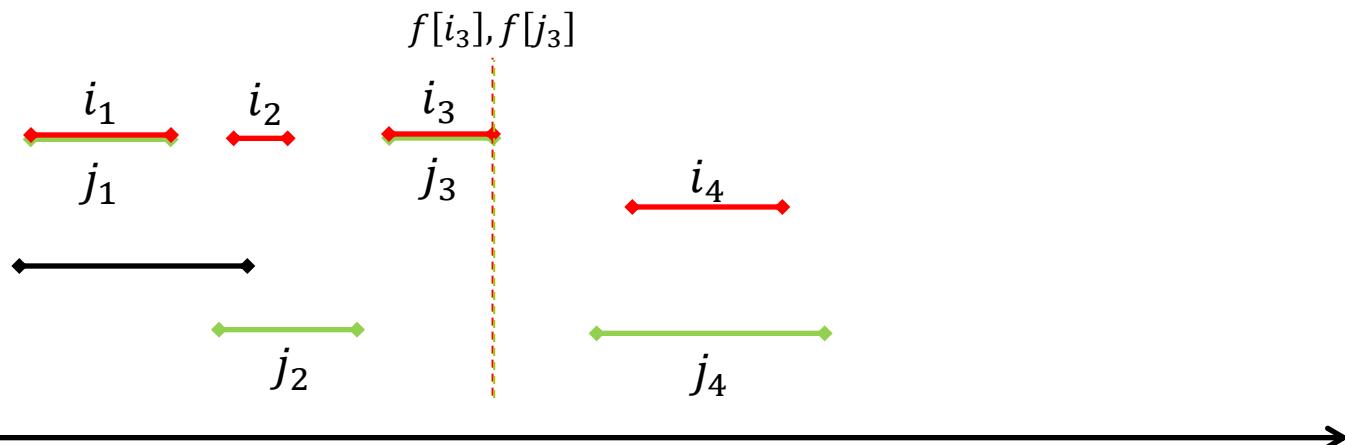
Für alle $r \leq k$ gilt $f[i_r] \leq f[j_r]$.



Gierige Algorithmen – Intervall Scheduling

Lemma 16.4

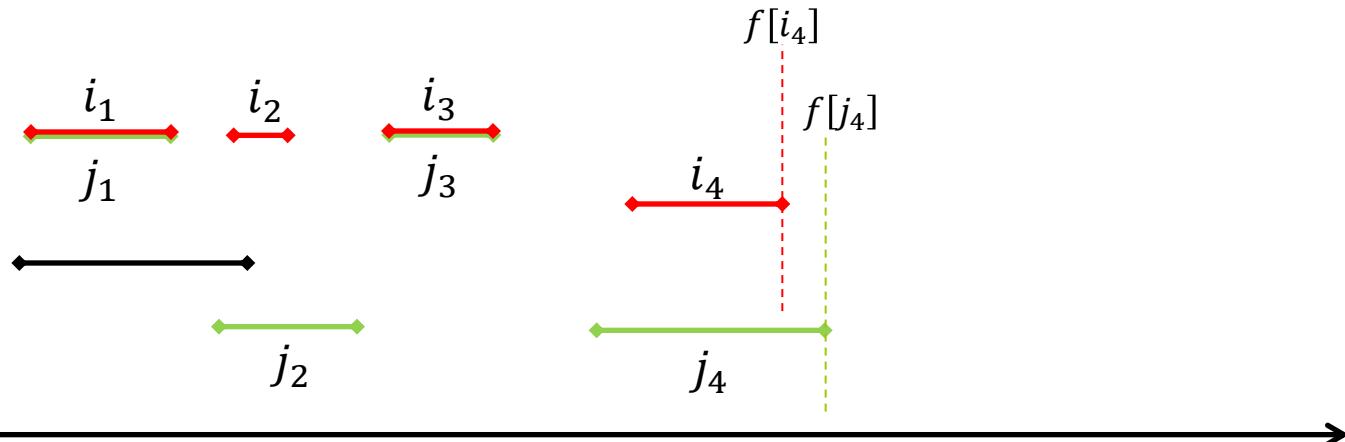
Für alle $r \leq k$ gilt $f[i_r] \leq f[j_r]$.



Gierige Algorithmen – Intervall Scheduling

Lemma 16.4

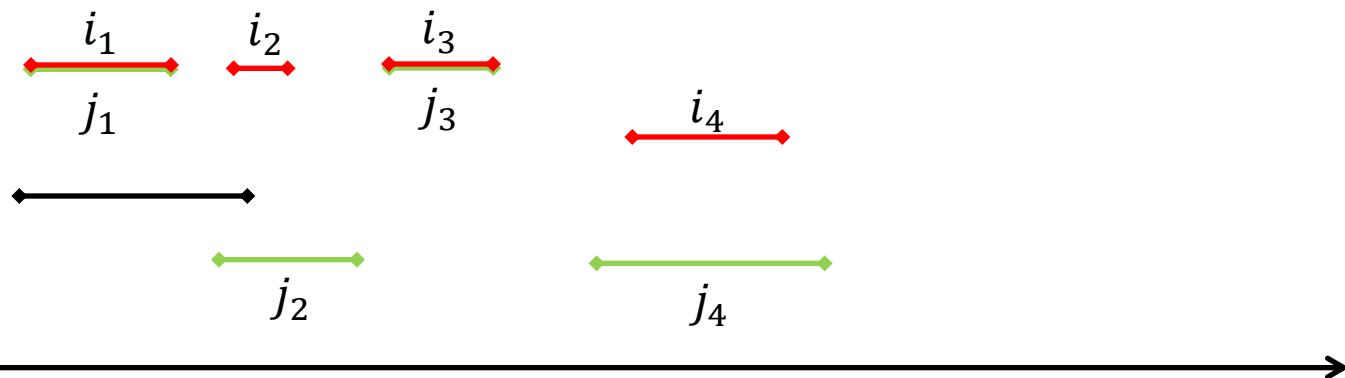
Für alle $r \leq k$ gilt $f[i_r] \leq f[j_r]$.



Gierige Algorithmen – Intervall Scheduling

Lemma 16.5

Die von Algorithmus GREEDY–ACTIVITY–SELECTOR berechnete Lösung A ist optimal.



Gierige Algorithmen – Intervall Scheduling

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  $n = s.length$ 
2  $A = \{a_1\}$ 
3  $k = 1$ 
4 for  $m = 2$  to  $n$ 
5   if  $s[m] \geq f[k]$ 
6      $A = A \cup \{a_m\}$ 
7      $k = m$ 
8 return  $A$ 
```

$$\left. \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \right\} \Theta(1)$$
$$\left. \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \Theta(n)$$
$$\left. \begin{array}{l} 8 \end{array} \right\} \Theta(1)$$

 $\Theta(n)$

Gierige Algorithmen – Intervall Scheduling

Satz 16.6

Der Algorithmus GREEDY–ACTIVITY–SELECTOR berechnet in $\Theta(n)$ Zeit eine optimale Lösung, wenn die Eingabe nach Endzeit der Intervalle (rechter Endpunkt) sortiert ist.

Die Sortierung kann in $\Theta(n \log n)$ Zeit berechnet werden.

Anwendungsbeispiel: Datenkompression

Gierige Algorithmen – Datenkompression

Datenkompression:

- Reduziert Größen von Files
- Viele Verfahren für unterschiedliche Anwendungen:
 - MP3
 - MPEG
 - JPEG
 - ...

Zwei Typen von Kompression:

- Verlustbehaftete Kompression (Bilder, Musik, Filme, ...)
- Verlustfreie Kompression (Programme, Texte, ...)

Gierige Algorithmen – Datenkompression

Kodierung:

- Computer arbeiten auf Bits (Symbole 0 und 1), nutzen also das Alphabet {0,1}
- Menschen nutzen umfangreichere Alphabete
 - z.B. Alphabete von Sprachen
- Darstellung auf Rechner erfordert Umwandlung in Bitfolgen

Gierige Algorithmen – Datenkompression

Beispiel:

- Alphabet $\Sigma = \{a, b, c, d, \dots, x, y, z, ., :, !, ?, \&\}$ (32 Zeichen)
- es sind 5 Bits pro Symbol: $2^5 = 32$ Möglichkeiten

a	b	...	z	.	:	!	?	&	
00000	00001		11001	11010	11011	11100	11101	11110	11111

Fragen?

- Sind 4 Bits pro Symbol nicht genug?
- Müssen wir im Durchschnitt 5 Bits für jedes Vorkommen eines Symbols in langen Texten verwenden?

Gierige Algorithmen – Datenkompression

Beobachtung:

- Nicht jeder Buchstabe kommt gleich häufig vor
- z.B. kommen x, y und z in der Deutschen Sprache viel seltener vor als e, n oder r

Idee:

- Benutze kurze Bitstrings für Symbole, die häufig vorkommen

Effekt:

- Gesamtlänge der Kodierung einer Symbolfolge (eines Textes) wird reduziert

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- **Eingabe:** Text über dem Alphabet Σ
- **Gesucht:** Eine binäre Kodierung von Σ , sodass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- Alphabet $\Sigma = \{0,1,2,\dots,9\}$
- Text = 00125590004356789 (17 Zeichen)

Länge der Kodierung:
 $4 \cdot 17 = 68$ Bits

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Gierige Algorithmen – Datenkompression

Grundlegendes Problem:

- **Eingabe:** Text über dem Alphabet Σ
- **Gesucht:** Eine binäre Kodierung von Σ , sodass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- Alphabet $\Sigma = \{0,1,2,\dots,9\}$
- Text = 00125590004356789 (17 Zeichen)

Länge der Kodierung:
 $5 \cdot 1 + 3 \cdot 2 + 9 \cdot 5 = 56$ Bits

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Gierige Algorithmen – Datenkompression

Morse-Code:

- Elektrische Pulse über Kabel
- Punkte (kurze Pulse)
- Striche (lange Pulse)

Beispiele aus dem Morse-Code:

- Buchstabe *e* ist 0 (ein einzelner Punkt)
- Buchstabe *t* ist 1 (ein einzelner Strich)
- Buchstabe *a* ist 01 (Strich-Punkt)

Problem:

- Ist 0101 *eta, aa, etet* oder *aet*?

Gierige Algorithmen – Datenkompression

Problem Mehrdeutigkeit:

- Ist die Kodierung eines Buchstabens ein Präfix der Kodierung eines anderen Buchstabens, dann ist die Kodierung nicht eindeutig

Beispiel:

- Buchstabe $e = 0$
- Buchstabe $a = 01$
- ABER: **0** ist Präfix von **01**

Gierige Algorithmen – Datenkompression

Präfix-Kodierung:

Eine Präfix-Kodierung für ein Alphabet Σ ist eine Funktion γ , die jeden Buchstaben $x \in \Sigma$ auf eine endliche Sequenz von 0 und 1 abbildet, sodass für $x, y \in \Sigma$, $x \neq y$ die Sequenz $\gamma(x)$ **kein** Präfix der Sequenz $\gamma(y)$ ist.

Beispiel (Präfix-Kodierung):

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	0100	0110	0111	1001	1010	1011	1101	1110	1111

Gierige Algorithmen – Datenkompression

Definition (Frequenz)

Die **Frequenz** $f[x]$ eines Buchstabens $x \in \Sigma$ bezeichnet den Bruchteil der Buchstaben im Text, die x sind.

Beispiel:

- $\Sigma = \{0,1,2\}$
- Text = „0010022001“ (10 Zeichen)
- $f[0] = 3/5$
- $f[1] = 1/5$
- $f[2] = 1/5$

Gierige Algorithmen – Datenkompression

Definition (Kodierungslänge)

Die **Kodierungslänge** eines Textes mit n Zeichen bzgl. einer Kodierung γ ist definiert als

$$\text{Kodierungslänge} = \sum_{x \in \Sigma} n \cdot f[x] \cdot |\gamma(x)|$$

Anzahl der Vorkommen
von x im Text

Beispiel:

- $\Sigma = \{a, b, c, d\}$
- $\gamma(a) = 0; \gamma(b) = 101; \gamma(c) = 110; \gamma(d) = 111$
- Text = „aacdaabb“
- Kodierungslänge = 16

Gierige Algorithmen – Datenkompression

Definition (Kodierungslänge)

Die **Kodierungslänge** eines Textes mit n Zeichen bzgl. einer Kodierung γ ist definiert als

$$\text{Kodierungslänge} = \sum_{x \in \Sigma} n \cdot f[x] \cdot |\gamma(x)|$$

Länge der Kodierung
von x

Beispiel:

- $\Sigma = \{a, b, c, d\}$
- $\gamma(a) = 0; \gamma(b) = 101; \gamma(c) = 110; \gamma(d) = 111$
- Text = „aacdaabb“
- Kodierungslänge = 16

Gierige Algorithmen – Datenkompression

Definition (durchschn. Kodierungslänge) —

Die **durchschnittliche Kodierungslänge** eines Buchstabens in einem Text mit n Zeichen und bzgl. einer Kodierung γ ist definiert als

$$\text{ABL}(\gamma) = \sum_{x \in \Sigma} f[x] \cdot |\gamma(x)|$$

Beispiel:

- $\Sigma = \{a, b, c, d\}$
- $\gamma(a) = 0; \gamma(b) = 101; \gamma(c) = 110; \gamma(d) = 111$
- Text = „aacdaabb“
- durchschnittliche Kodierungslänge = $^{16}/_8 = 2$

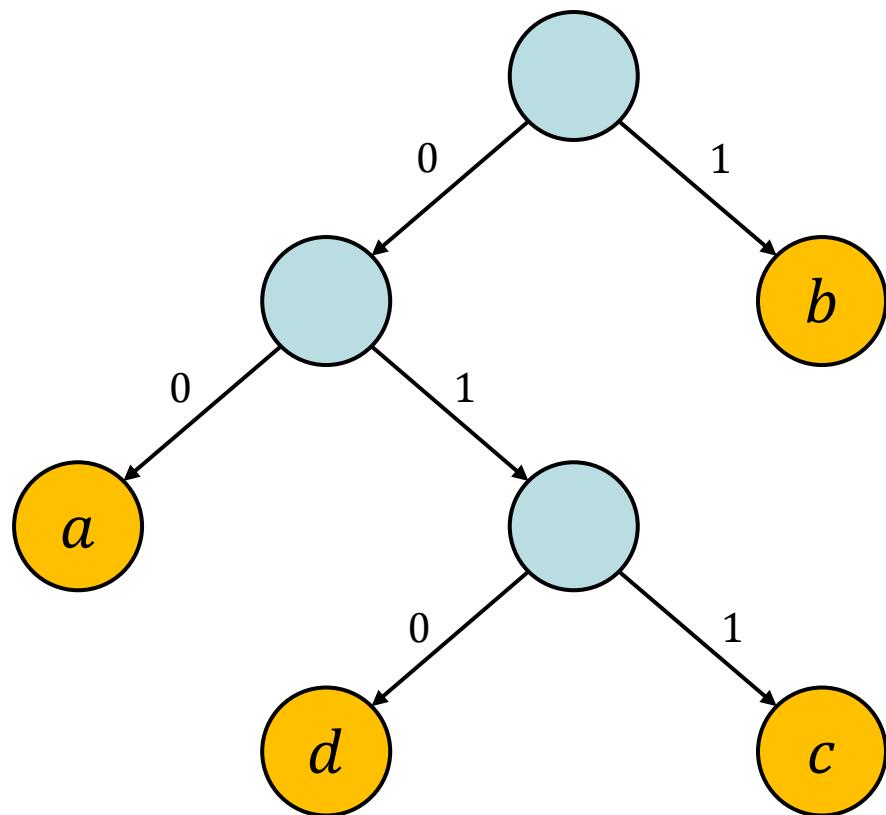
Gierige Algorithmen – Datenkompression

Problem einer optimalen Präfix-Kodierung:

- **Eingabe:** Alphabet Σ und für jedes $x \in \Sigma$ seine Frequenz $f[x]$
- **Ausgabe:** Eine Kodierung γ , die $ABL(\gamma)$ minimiert

Gierige Algorithmen – Datenkompression

Binärbäume und Präfix-Kodierungen:

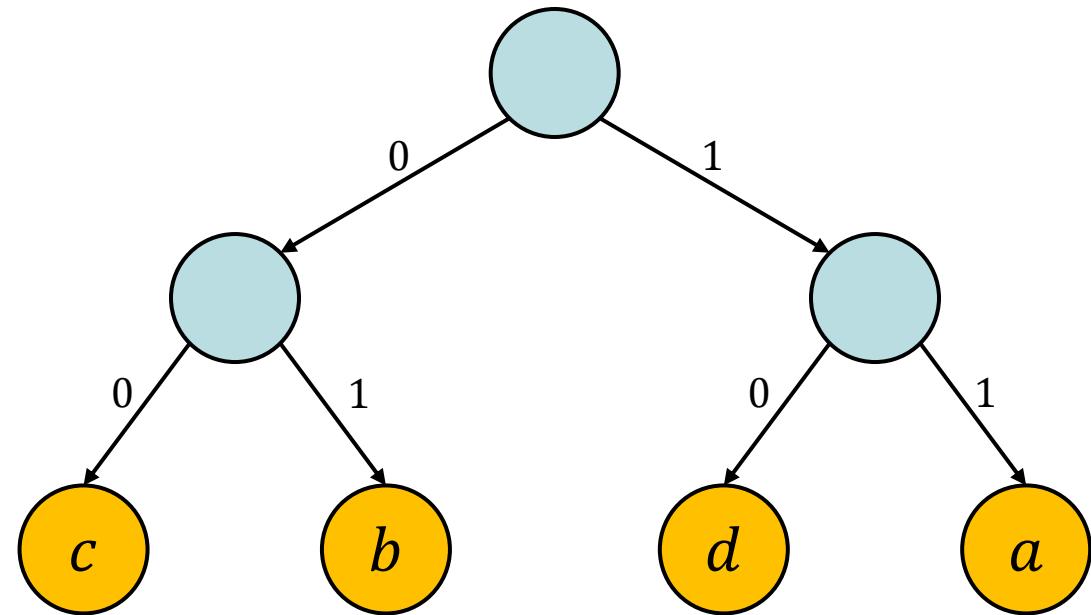


$x \in \Sigma$	$\gamma(x)$
a	00
b	1
c	011
d	010

Gierige Algorithmen – Datenkompression

Präfix-Kodierungen und Binärbäume:

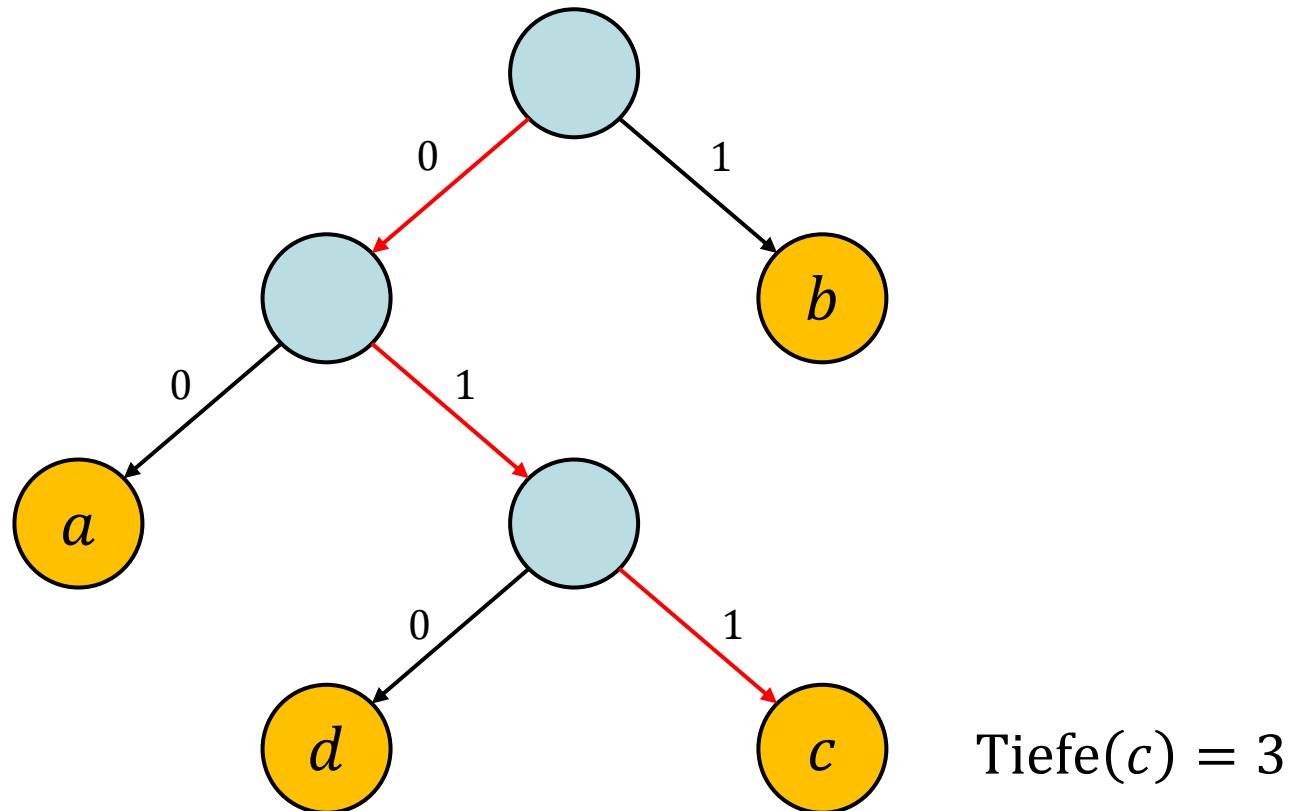
$x \in \Sigma$	$\gamma(x)$
a	11
b	01
c	00
d	10



Gierige Algorithmen – Datenkompression

Definition

Die **Tiefe** eines Baumknotens ist die Länge seines Pfades zur Wurzel.



Gierige Algorithmen – Datenkompression

Neue Problemformulierung

Suche Binärbaum T , dessen Blätter die Symbole aus Σ sind und der

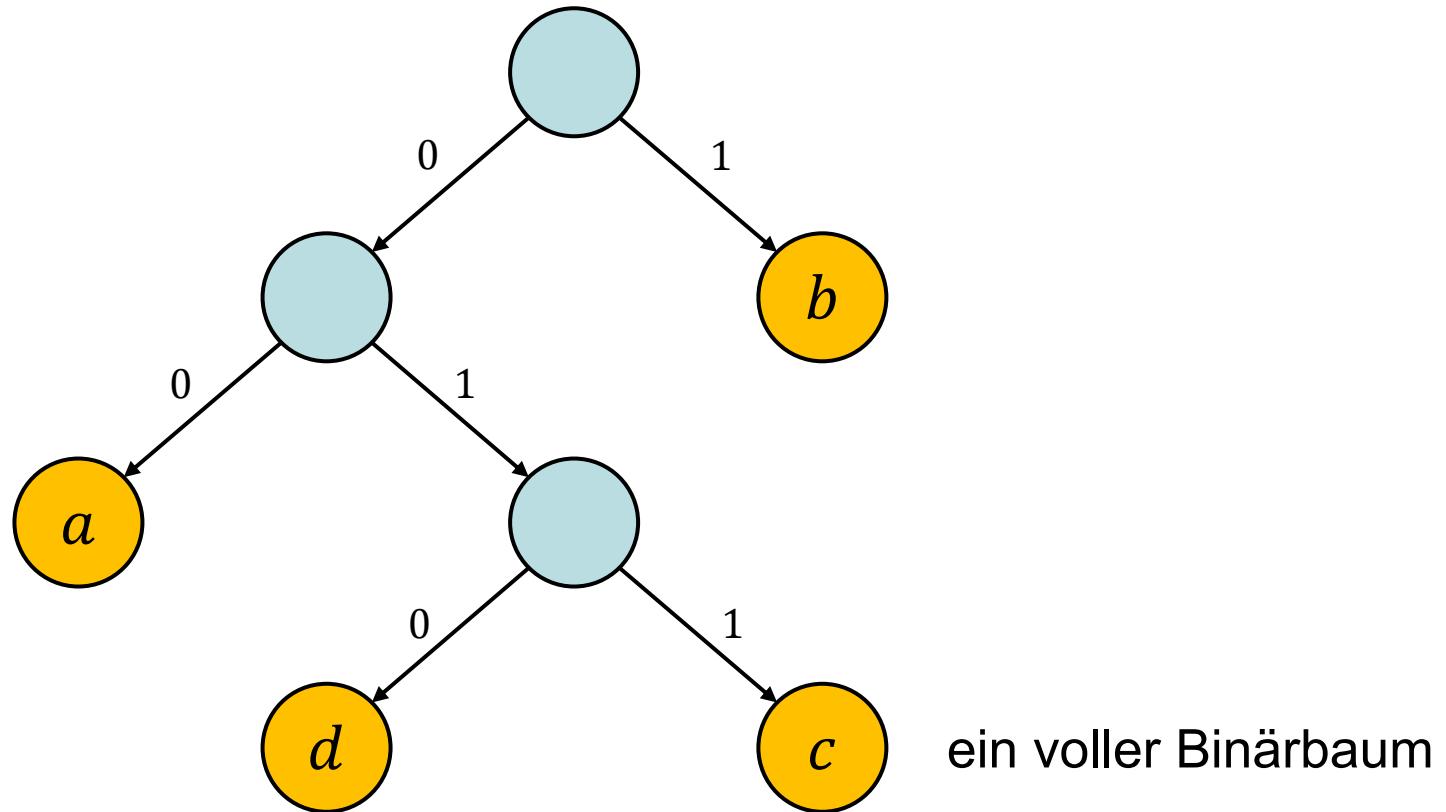
$$\text{ABL}(T) = \sum_{x \in \Sigma} f[x] \cdot \text{TIEFE}_T(x)$$

minimiert.

Gierige Algorithmen – Datenkompression

Definition

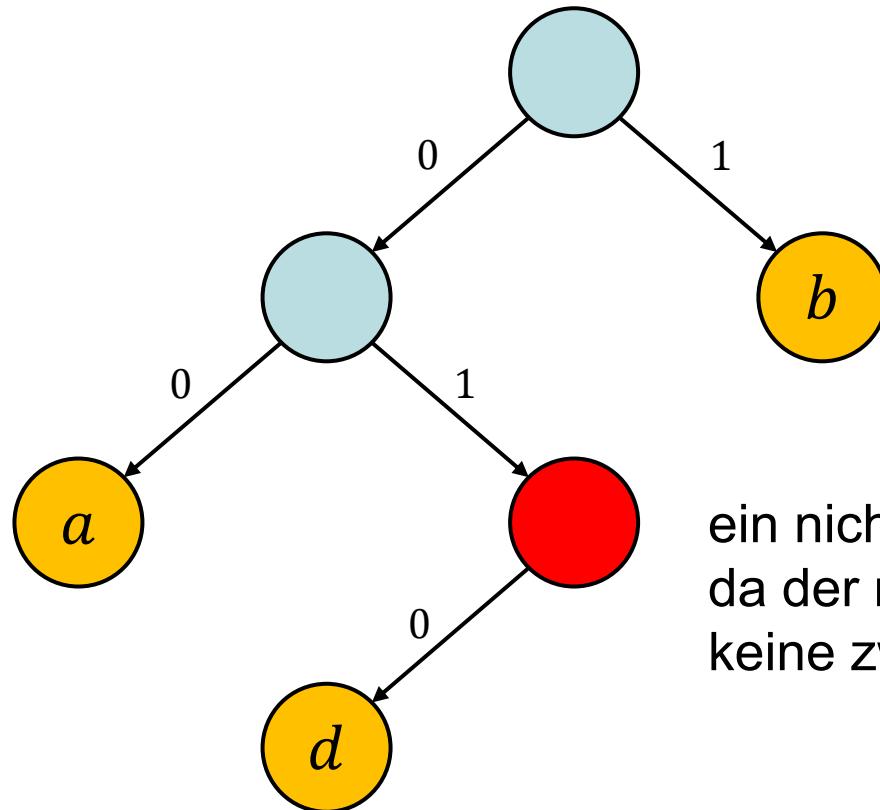
Ein Binärbaum heißt **voll**, wenn jeder innere Knoten genau zwei Kinder hat.



Gierige Algorithmen – Datenkompression

Definition

Ein Binärbaum heißt **voll**, wenn jeder innere Knoten genau zwei Kinder hat.

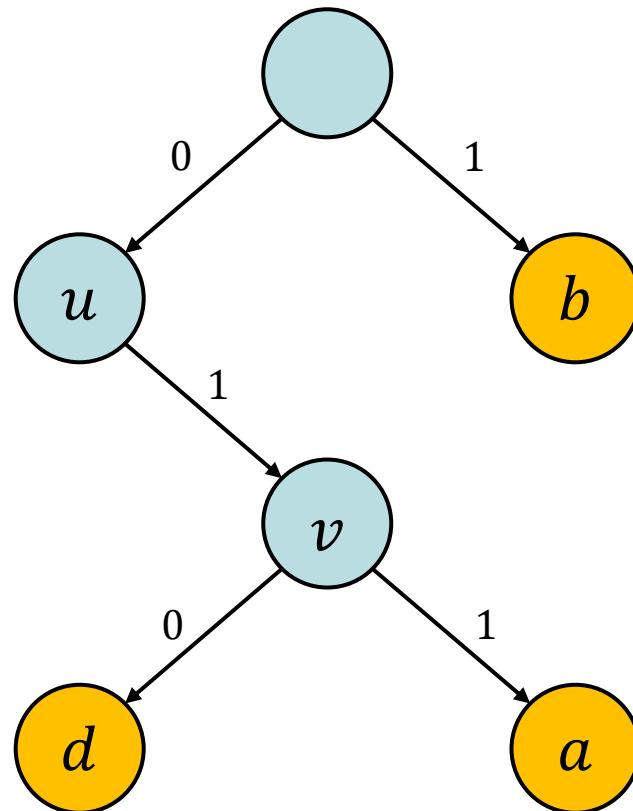


ein nicht voller Binärbaum,
da der rote innere Knoten
keine zwei Kinder hat

Gierige Algorithmen – Datenkompression

Lemma 16.7

Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.



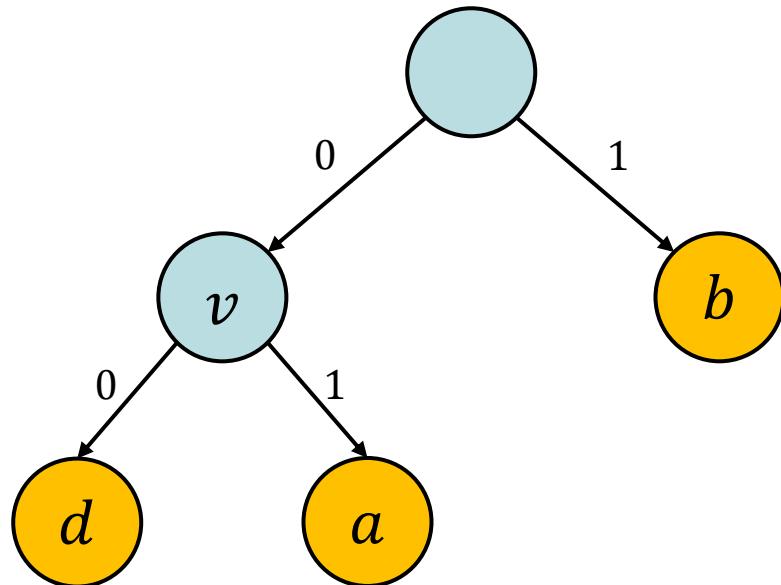
Beweis:

- Annahme: T ist optimal und hat inneren Knoten u mit einem Kind v
- Ersetze u durch v
- Dies verkürzt die Tiefe einiger Knoten, erhöht aber keine Tiefe
- Damit verbessert man die Kodierung

Gierige Algorithmen – Datenkompression

Lemma 16.7

Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.



Beweis:

- Annahme: T ist optimal und hat inneren Knoten u mit einem Kind v
- Ersetze u durch v
- Dies verkürzt die Tiefe einiger Knoten, erhöht aber keine Tiefe
- Damit verbessert man die Kodierung

Gierige Algorithmen – Datenkompression

Ein Gedankenexperiment:

- Angenommen, jemand gibt uns den optimalen Baum T^* , aber nicht die Bezeichnung der Blätter
- Wie schwierig ist es, die Bezeichnungen zu finden?

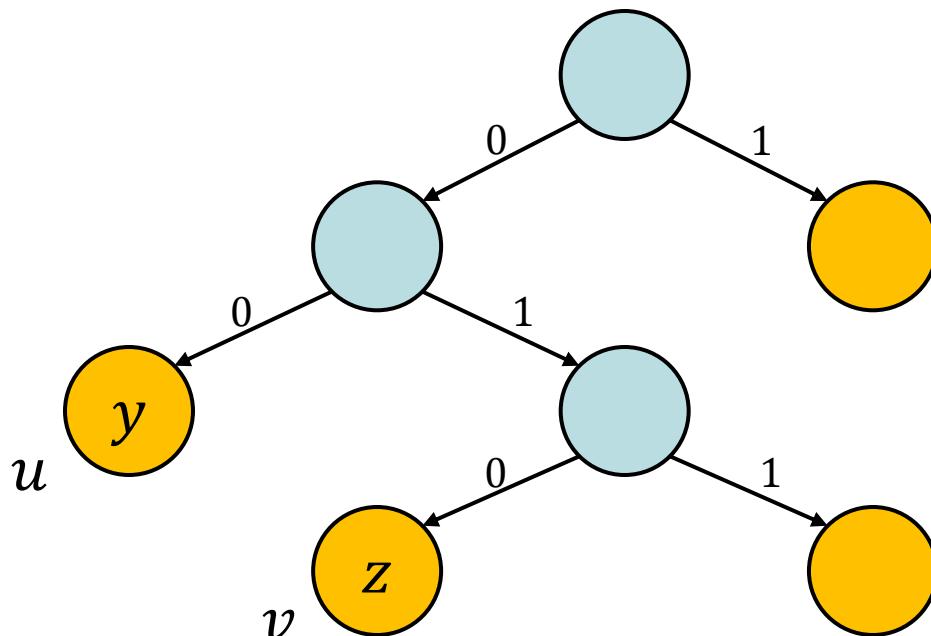
Gierige Algorithmen – Datenkompression

Lemma 16.8

Seien u und v Blätter von T^* mit $\text{TIEFE}(u) < \text{TIEFE}(v)$.

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet.

Dann gilt $f[y] \geq f[z]$.



$x \in \Sigma$	$f[x]$
a	10%
b	14%
c	16%
d	60%

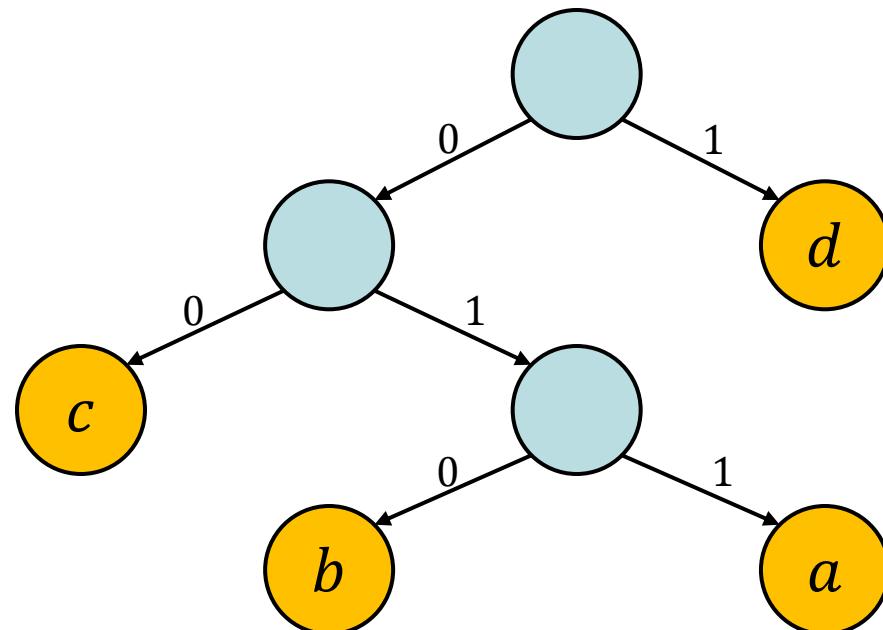
Gierige Algorithmen – Datenkompression

Lemma 16.8

Seien u und v Blätter von T^* mit $\text{TIEFE}(u) < \text{TIEFE}(v)$.

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet.

Dann gilt $f[y] \geq f[z]$.



$x \in \Sigma$	$f[x]$
a	10%
b	14%
c	16%
d	60%

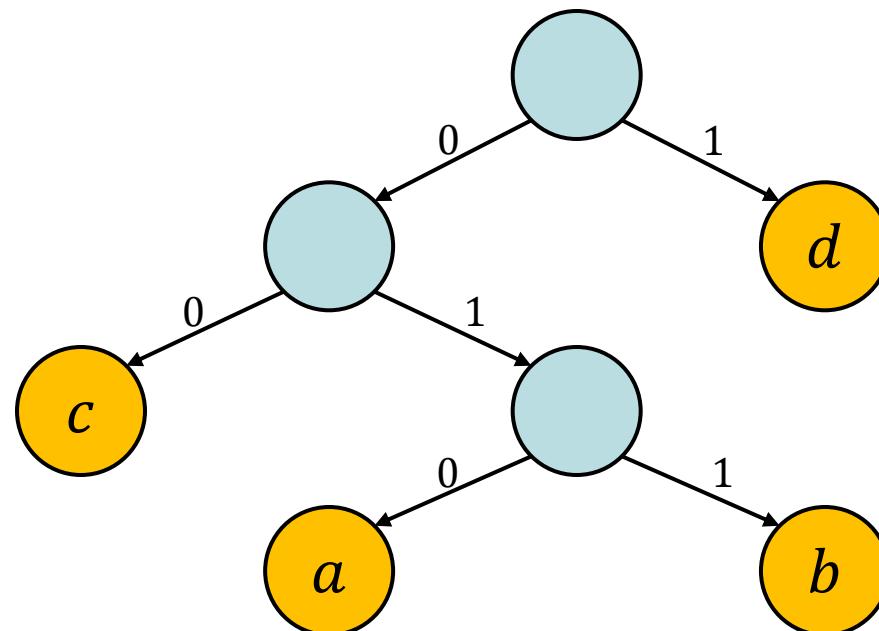
Gierige Algorithmen – Datenkompression

Lemma 16.8

Seien u und v Blätter von T^* mit $\text{TIEFE}(u) < \text{TIEFE}(v)$.

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet.

Dann gilt $f[y] \geq f[z]$.



Anordnung innerhalb einer Tiefenstufe ist egal!

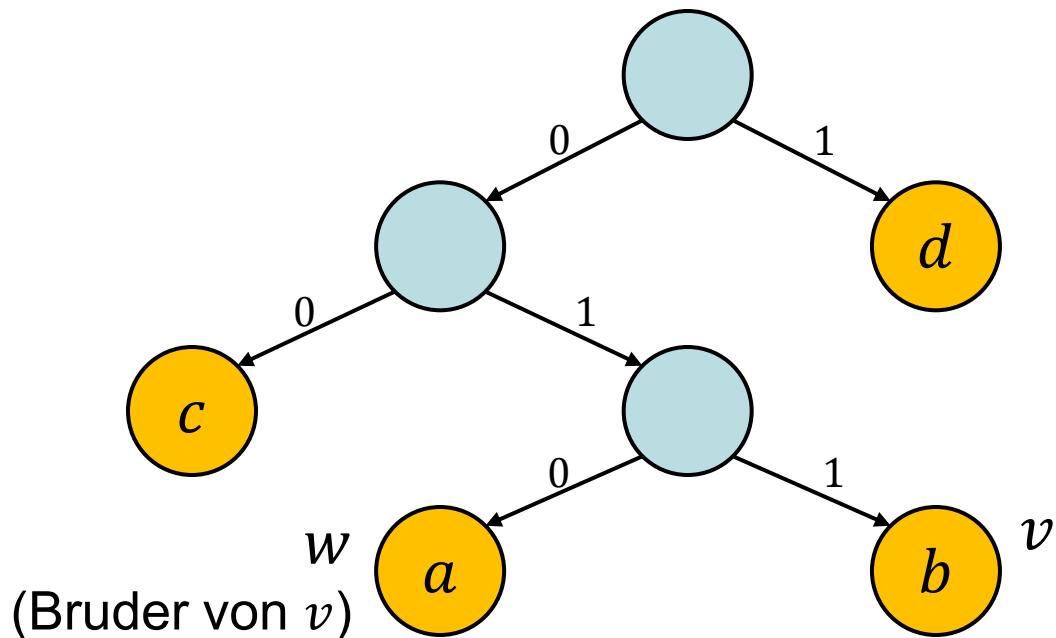
$x \in \Sigma$	$f[x]$
a	10%
b	14%
d	60%

Gierige Algorithmen – Datenkompression

Beobachtung:

Sei ν der tiefste Blattknoten in T^* .

Dann hat ν einen Bruder, dieser ist ebenfalls ein Blattknoten.



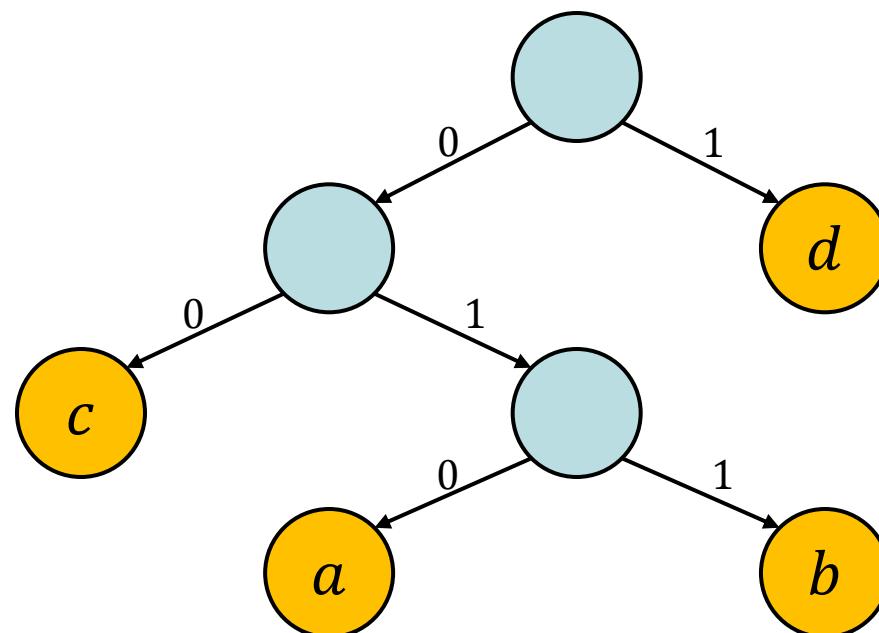
Beweis:

- Da ein optimaler Baum voll ist, hat ν einen Bruder w
- Wäre w kein Blatt, dann hätte ein Nachfolger von w größere Tiefe als ν

Gierige Algorithmen – Datenkompression

Zusammenfassende Behauptung:

Es gibt eine optimale Präfix-Kodierung mit zugehörigem Baum T^* , sodass die beiden Blattknoten, denen die Symbole mit den kleinsten Frequenzen zugewiesen wurden, Bruderknoten in T^* sind.



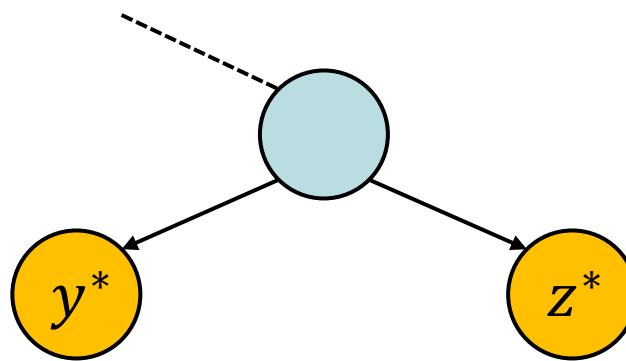
$x \in \Sigma$	$f[x]$
a	10%
c	16%
d	60%

a und b erfüllen die Bedingungen der Behauptung!

Gierige Algorithmen – Datenkompression

Idee des Algorithmus:

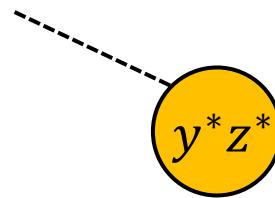
- Die beiden Symbole y^* und z^* mit den niedrigsten Frequenzen sind Bruderknoten
- Fasse y^* und z^* zu einem neuen Symbol zusammen
- Löse das Problem für die übrigen $n - 1$ Symbole
 - z.B. rekursiv



Gierige Algorithmen – Datenkompression

Idee des Algorithmus:

- Die beiden Symbole y^* und z^* mit den niedrigsten Frequenzen sind Bruderknoten
- Fasse y^* und z^* zu einem neuen Symbol zusammen
- Löse das Problem für die übrigen $n - 1$ Symbole
 - z.B. rekursiv



Gierige Algorithmen – Datenkompression

HUFFMAN(C)

```
1  $n = |C|$ 
2  $Q = C$ 
3 for  $i = 1$  to  $n - 1$ 
4     allocate a new node  $z$ 
5      $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6      $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7      $z.freq = x.freq + y.freq$ 
8      $\text{INSERT}(Q, z)$ 
9 return EXTRACT-MIN( $Q$ )
// return the root of the tree
```

$x \in C$	$f[x]$
a	23%
b	12%
c	55%
d	10%



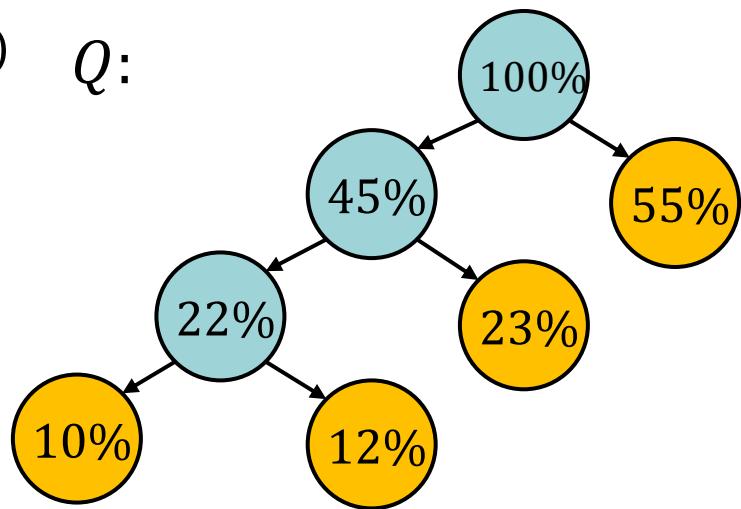
Gierige Algorithmen – Datenkompression

HUFFMAN(C)

```
1  $n = |C|$ 
2  $Q = C$ 
3 for  $i = 1$  to  $n - 1$ 
4     allocate a new node  $z$ 
5      $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6      $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7      $z.freq = x.freq + y.freq$ 
8      $\text{INSERT}(Q, z)$ 
9 return EXTRACT-MIN( $Q$ )
// return the root of the tree
```

$i = 3$

$Q:$



Gierige Algorithmen – Datenkompression

Satz 16.9

Algorithmus HUFFMAN(C) berechnet eine optimale Präfix-Kodierung.

Gierige Algorithmen

Gierige Algorithmen:

- Berechne Lösung schrittweise
- In jedem Schritt mache lokal optimale Wahl

Anwendbar:

- Wenn optimale Lösung eines Problems optimale Lösung von Teilproblemen enthält

Algorithmen:

- Scheduling-Probleme
- Optimale Präfix-Kodierung (Huffman Codes)

Dynamische Programmierung

Rekursiver Ansatz:

- Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt

Phänomen:

- Mehrfachberechnungen von Lösungen

Methode:

- Lösungen zu Teilproblemen werden iterativ beginnend mit den Lösungen der kleinsten Teilprobleme berechnet (*bottom-up*).
- Speichern einmal berechneter Lösungen in einer Tabelle.

Dynamische Programmierung

- typische Anwendung für dynamisches Programmieren:
Optimierungsprobleme
- eine optimale Lösung für das Ausgangsproblem setzt sich aus
optimalen Lösungen für kleinere Probleme zusammen

Dynamische Programmierung

Mit Greedy-Algorithmen:

- Algorithmenmethode, um Optimierungsprobleme zu lösen

Mit Divide-&-Conquer

- Lösung eines Problems aus Lösungen zu Teilproblemen
- Aber: Lösungen zu Teilproblemen werden *nicht* rekursiv gelöst

Dynamische Programmierung – Längste gemeinsame Teilfolge

Definition

Seien $X = (x_1, \dots, x_m)$ und $Y = (y_1, \dots, y_n)$ zwei Teilfolgen, wobei $x_i, y_i \in A$ für ein endliches Alphabet A

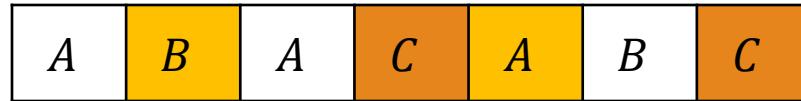
Dann heißt Y Teilfolge von X , wenn es aufsteigend sortierte Indizes i_1, \dots, i_n gibt mit $x_{i_j} = y_j$ für $j = 1, \dots, n$.

Beispiel:

Folge Y



Folge X



Y ist Teilfolge von X , wähle $(i_1, i_2, i_3, i_4) = (2, 4, 5, 7)$

Dynamische Programmierung – Längste gemeinsame Teilfolge

Definition

Seien X, Y Folgen über A .

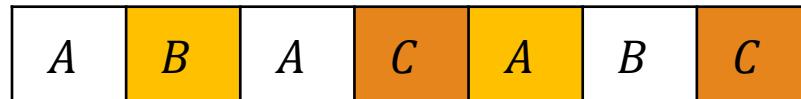
Dann heißt Z **gemeinsame Teilfolge** von X und Y , wenn Z Teilfolge sowohl von X , als auch von Y ist.

Beispiel:

Folge Z



Folge X



Folge Y



Z ist gemeinsame Teilfolge von X und Y

Dynamische Programmierung – Längste gemeinsame Teilfolge

Definition

Seien X, Y Folgen über A .

Dann heißt Z **längste gemeinsame Teilfolge** von X und Y , wenn Z gemeinsame Teilfolge von X und von Y ist und es keine andere gemeinsame Teilfolge von X und Y gibt, die größere Länge als Z besitzt.

Beispiel:

Folge X

A	B	A	C	A	B	C
-----	-----	-----	-----	-----	-----	-----

Folge Y

B	A	C	C	A	B	B	C
-----	-----	-----	-----	-----	-----	-----	-----

Dynamische Programmierung – Längste gemeinsame Teilfolge

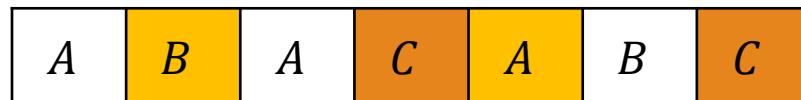
Definition

Seien X, Y Folgen über A .

Dann heißt Z **längste gemeinsame Teilfolge** von X und Y , wenn Z gemeinsame Teilfolge von X und von Y ist und es keine andere gemeinsame Teilfolge von X und Y gibt, die größere Länge als Z besitzt.

Beispiel:

Folge X



Folge Y



Folge Z_1



Dynamische Programmierung – Längste gemeinsame Teilfolge

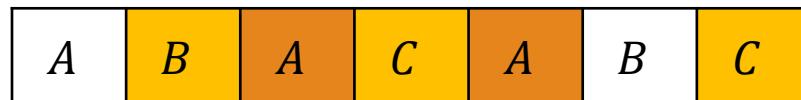
Definition

Seien X, Y Folgen über A .

Dann heißt Z **längste gemeinsame Teilfolge** von X und Y , wenn Z gemeinsame Teilfolge von X und von Y ist und es keine andere gemeinsame Teilfolge von X und Y gibt, die größere Länge als Z besitzt.

Beispiel:

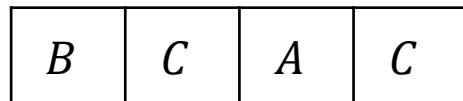
Folge X



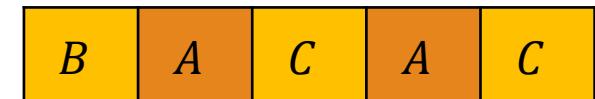
Folge Y



Folge Z_1



Folge Z_2



Dynamische Programmierung – Längste gemeinsame Teilfolge

Eingabe:

- Folge $X = (x_1, \dots, x_m)$
- Folge $Y = (y_1, \dots, y_n)$

Ausgabe:

- Längste gemeinsame Teilfolge Z (**L**ongest **C**ommon **S**ubsequence)

Beispiel:

Folge X

A	B	C	B	D	A	B
---	---	---	---	---	---	---

Folge Y

B	D	C	A	B	A
---	---	---	---	---	---

Dynamische Programmierung – Längste gemeinsame Teilfolge

Algorithmus:

- Erzeuge alle möglichen Teilfolgen von X
- Teste für jede Teilfolge von X , ob auch Teilfolge von Y
- Merke zu jedem Zeitpunkt bisher längste gemeinsame Teilfolge

Laufzeit:

- es gibt 2^m mögliche Teilfolgen
- Exponentielle Laufzeit!

Satz 17.1

Seien $X = (x_1, \dots, x_m)$ und $Y = (y_1, \dots, y_n)$ beliebige Folgen und sei $Z = (z_1, \dots, z_k)$ eine längste gemeinsame Teilfolge von X und Y .

Dann gilt:

1. Ist $x_m = y_n$, dann ist $z_k = x_m = y_n$ und (z_1, \dots, z_{k-1}) ist eine längste gemeinsame Teilfolge von (x_1, \dots, x_{m-1}) und (y_1, \dots, y_{n-1}) .
2. Ist $x_m \neq y_n$ und $z_k \neq x_m$, dann ist Z eine längste gemeinsame Teilfolge von (x_1, \dots, x_{m-1}) und Y .
3. Ist $x_m \neq y_n$ und $z_k \neq y_n$, dann ist Z eine längste gemeinsame Teilfolge von X und (y_1, \dots, y_{n-1}) .

Dynamische Programmierung – Längste gemeinsame Teilfolge

Lemma 17.2

Sei $c[i, j]$ die Länge einer längsten gemeinsamen Teilfolge von (x_1, \dots, x_i) und (y_1, \dots, y_j) . Dann gilt:

$$c[i, j] = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0 \\ c[i - 1, j - 1] + 1 & \text{falls } i, j > 0 \text{ und } x_i = y_j \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Beobachtung:

Rekursive Berechnung der $c[i, j]$ würde zu Berechnung immer wieder derselben Werte führen. Dies ist ineffizient. Wir berechnen daher die Werte $c[i, j]$ iterativ, nämlich zeilenweise.

Dynamische Programmierung – Längste gemeinsame Teilfolge

LCS-LENGTH(X, Y)

```
1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i - 1, j - 1] + 1$ 
12              $b[i, j] = "↖"$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = "↑"$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = "←"$ 
18 return  $c$  and  $b$ 
```

Dynamische Programmierung – Längste gemeinsame Teilfolge

j	0	1	2	3	4	5	6
i	y_1	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$
2	B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$
3	C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$
4	B	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$
5	D	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$
6	A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$
7	B	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$

Dynamische Programmierung – Längste gemeinsame Teilfolge

j	0	1	2	3	4	5	6
i	y_1	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$
2	B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$
3	C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$
4	B	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$
5	D	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$
6	A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$
7	B	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$

Dynamische Programmierung – Längste gemeinsame Teilfolge

Lemma 17.3

Der Algorithmus LCS–LENGTH hat Laufzeit $O(nm)$, wenn die Folgen X, Y Länge n und m haben.

Lemma 17.4

Die Ausgabe der längsten gemeinsamen Teilfolge anhand der Tabelle hat Laufzeit $O(n + m)$, wenn die Folgen X, Y Länge n und m haben.

Dynamische Programmierung – Längste gemeinsame Teilfolge

1. Bestimme rekursive Struktur einer optimalen Lösung.
2. Entwerfe rekursive Methode zur Bestimmung des Wertes einer optimalen Lösung.
3. Transformiere rekursive Methode in eine iterative (bottom-up) Methode zur Bestimmung des Wertes einer optimalen Lösung.
4. Bestimme aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechnete Zusatzinformationen eine optimale Lösung.

Dynamische Programmierung – Längste gemeinsame Teilfolge

Algorithmenentwurfstechnik:

- Oft bei Optimierungsproblemen angewandt

Einsatz:

- Bei rekursiven Problemlösungen, wenn Teillösungen mehrfach benötigt werden

Lösungsansatz:

- Tabellieren von Teilergebnissen

Vorteil:

- Laufzeitverbesserungen, oft polynomiell statt exponentiell

Dynamische Programmierung

Stand der Dinge:

- Dynamische Programmierung vermeidet Mehrfachberechnung von Zwischenergebnissen
- Bei Rekursion einsetzbar
- Häufig einfache *bottom-up*-Implementierung möglich

Das LCS-Problem:

- Algorithmus für schwieriges Problem
- Laufzeit hängt von Eingabewert ab

Dynamische Programmierung – Rucksack

Das Rucksackproblem:

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Dynamische Programmierung – Rucksack

Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen und haben Gesamtwert 13
- Optimal?
- Objekt 2, 3 und 4 passen und haben Gesamtwert 15!

Dynamische Programmierung – Rucksack

Das Rucksackproblem (Optimierungsversion):

- **Eingabe:** n Objekte $\{1, \dots, n\}$;
Objekt i hat ganzzahlige positive
Größe $g[i]$ und Wert $v[i]$;
Rucksackkapazität W
- **Ausgabe:** Menge $S \subseteq \{1, \dots, n\}$ mit
$$\sum_{i \in S} g[i] \leq W$$
und maximalem Wert
$$\sum_{i \in S} v[i]$$

Dynamische Programmierung – Rucksack

Herleiten einer Rekursion:

- Sei O optimale Lösung
- Bezeichne $Opt(i, w)$ den Wert einer optimalen Lösung aus Objekten 1 bis i bei Rucksackgröße w

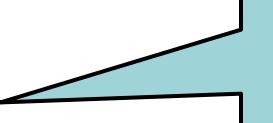
Unterscheide, ob Objekt n in O ist:

- **Fall 1** (n nicht in O): $Opt(n, W) = Opt(n - 1, W)$
- **Fall 2** (n in O): $Opt(n, W) = v[n] + Opt(n - 1, W - g[n])$

Dynamische Programmierung – Rucksack

Rekursion:

- $Opt(i, 0) = 0$ für $0 \leq i \leq n$
- $Opt(0, i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$, dann $Opt(i, w) = Opt(i - 1, w)$
- Sonst: $Opt(i, w) = \max\{Opt(i - 1, w), v[i] + Opt(i - 1, w - g[i])\}$



Kein Objekt
passt in den
Rucksack.

Dynamische Programmierung – Rucksack

Rekursion:

- $Opt(i, 0) = 0$ für $0 \leq i \leq n$
- $Opt(0, i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$, dann $Opt(i, w) = Opt(i - 1, w)$
- Sonst: $Opt(i, w) = \max\{Opt(i - 1, w), v[i] + Opt(i - 1, w - g[i])\}$

Kein Objekt
steht zur
Auswahl.

Dynamische Programmierung – Rucksack

Rekursion:

- $Opt(i, 0) = 0$ für $0 \leq i \leq n$
- $Opt(0, i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$, dann $Opt(i, w) = Opt(i - 1, w)$
- Sonst: $Opt(i, w) = \max\{Opt(i - 1, w), v[i] + Opt(i - 1, w - g[i])\}$

Passt aktuelles
Objekt in
den Rucksack?

Dynamische Programmierung – Rucksack

Rekursion:

- $Opt(i, 0) = 0$ für $0 \leq i \leq n$
- $Opt(0, i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$, dann $Opt(i, w) = Opt(i - 1, w)$
- Sonst: $Opt(i, w) = \max\{Opt(i - 1, w), v[i] + Opt(i - 1, w - g[i])\}$

Sonst verwende
Rekursion.

Dynamische Programmierung – Rucksack

RUCKSACK(n, W, G, V)

```
1 let  $A[0..n, 0..W]$  be a new table
2 for  $i = 0$  to  $n$ 
3    $A[i, 0] = 0$ 
4 for  $j = 0$  to  $W$ 
5    $A[0, j] = 0$ 
6 for  $i = 1$  to  $n$ 
7   for  $j = 1$  to  $W$ 
8     if  $G[i] \leq j$ 
9       if  $V[i] + A[i - 1, j - G[i]] > A[i - 1, j]$ 
10       $A[i, j] = V[i] + A[i - 1, j - G[i]]$ 
11    else  $A[i, j] = A[i - 1, j]$ 
12  else  $A[i, j] = A[i - 1, j]$ 
13 return  $A[n, W]$ 
```

Die Arrays G und V enthalten die Größen bzw. die Werte der Objekte.

Dynamische Programmierung – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0								
0	0	0	0	0	0	0	0	0	W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung – Rucksack

Beispiel:

n	0	2	3	5	7	9	10	12	13
0	0	2	3	5	7	9	10	12	13
1	0	2	3	5	6	7	9	10	10
2	0	2	3	5	6	7	9	10	10
3	0	1	3	4	5	7	8	8	8
4	0	1	1	4	5	5	5	5	6
5	0	0	0	4	4	4	4	4	6
6	0	0	0	0	0	2	2	2	2
7	0	0	0	0	0	0	0	0	0

W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung – Rucksack

Satz 17.6

Algorithmus RUCKSACK berechnet in $\Theta(nW)$ Zeit den Wert einer optimalen Lösung, wobei n die Anzahl der Objekte ist und W die Größe des Rucksacks.

Dynamische Programmierung – Rucksack

Teile und herrsche:

- Aufteilen der Eingabe in mehrere Unterprobleme
- Rekursives Lösen der Unterprobleme
- Zusammenfügen

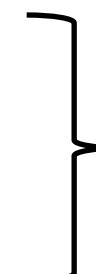
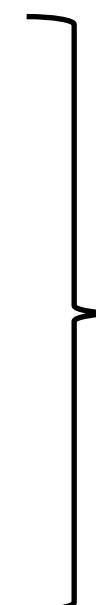
Gierige Algorithmen:

- Konstruiere Lösung Schritt für Schritt
- In jedem Schritt optimiere einfaches, lokales Kriterium

Teile und herrsche:

- Formuliere Problem rekursiv
- Vermeide mehrfache Berechnung von Teilergebnissen
- Verwende „bottom-up“-Implementierung

Inhaltsangabe

- Einleitung, Motivation
 - Pseudocode, Invarianten, Laufzeitanalyse
 - Gro β -O-Notation
- 
- Grundlagen**
-
- Inkrementelle Algorithmen, Insertion-Sort
 - Divide & Conquer Algorithmen, Merge-Sort
 - Quick-Sort Analyse und Varianten
 - Rekursionsgleichungen, Master-Theorem
 - Algorithmen mit Datenstrukturen, Heaps, Heap-Sort
 - Untere Schranke für Vergleichssortierer
 - Counting-Sort
- 
- Sortieralgorithmen**

- ADTs und Datenstrukturen, Stacks, Queues, Listen, Bäume
 - Hashtabellen
 - Binäre Suchbäume
- 
- Elementare Graphalgorithmen, Breitensuche
 - Tiefensuche
 - Zusammenhangskomponenten
 - Minimale Spannbäume, Algorithmus von Prim
 - Algorithmus von Kruskal, disjunkte Vereinigungsmengen
- 
- Gierige Algorithmen, Scheduling-Probleme
 - Dynamische Programmierung,
Längste gemeinsame Teilfolge
 - Optimale Suchbäume
 - Rucksack-Problem
- 