

# Vorlesung Algorithmen und Datenstrukturen (AD)

---

**Prof. Dr. Matthias Rarey**



Zentrum für Bioinformatik,  
Universität Hamburg  
Bundesstraße 43, 20146 Hamburg  
[rarey@zbh.uni-hamburg.de](mailto:rarey@zbh.uni-hamburg.de)

**[www.zbh.uni-hamburg.de](http://www.zbh.uni-hamburg.de)**

### ■ T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein

#### engl. Originalausgabe:

Introduction to Algorithms, MIT Press, 2009, (2. Aufl.)

deutsche Übersetzung:

Algorithmen – Eine Einführung, Oldenbourg 2007

### ■ Kopiervorlagen: (in den Übungen)

- Folien / Tafelbilder
- Ggf. kurze Auszüge aus anderen Lehrbüchern



### ■ Themen:

- Entwurf von Algorithmen und Datenstrukturen
- Beschreibung von Algorithmen
- Analyse der Platz- und Zeitkomplexität
- Algorithmen für häufig auftretende Probleme

### ■ Kapitel

1. Algorithmen und deren Komplexität
2. Grundlegende Datenstrukturen
3. Sortieren
4. Suchen
5. Graphen
6. Dynamische Programmierung
7. Lösen schwerer Probleme



## Inhaltsverzeichnis

### ■ Kapitel 1: Algorithmen und deren Komplexität

- 1.1 Beschreibung von Algorithmen
- 1.2 Analyse von Algorithmen
- 1.3 O-Notation
- 1.4 Laufzeitanalysen
- 1.5 Asymptotische Laufzeit rekursiver Funktionen
- 1.6 Ein Beispiel: Das Maximum-Subarray-Problem

### ■ Kapitel 2: Elementare Datenstrukturen

- 2.1 Elementare und Strukturierte Datentypen
- 2.2 Stapel
- 2.3 Lineare Listen
- 2.4 Bäume

### ■ Kapitel 3: Sortieren

- 3.1 Elementare Sortieralgorithmen
- 3.2 Heaps und Heapsort
- 3.3 Quicksort
- 3.4 Eine untere Schranke für das Sortierproblem
- 3.5 Counting-, Radix- und Bucketsort
- 3.6 Algorithmen für Auswahlprobleme



## Inhaltsverzeichnis

### ■ Kapitel 4: Suchen

- 4.1 Suchbäume
- 4.2 Rot-Schwarz-Bäume
- 4.3 AVL-Bäume
- 4.4 Hashing mit Verkettung
- 4.5 Offene Adressierung

### ■ Kapitel 5: Graphalgorithmen

- 5.1 Graphen
- 5.2 Traversieren von Graphen
- 5.3 Topologisches Sortieren
- 5.4 Starke Zusammenhangskomponenten
- 5.5 Minimale Spannbäume
- 5.6 Kürzeste Pfade
- 5.7 Weitere Graphprobleme im Überblick



### ■ Kapitel 6: Dynamische Programmierung

- 6.1 Prinzip der dynamischen Programmierung
- 6.2 Beispiel 1: Ablaufkoordination von Montagebändern
- 6.3 Beispiel 2: Matrix-Kettenmultiplikation

### ■ Kapitel 7: NP-Vollständigkeit

- 7.1 Formalisierung von „Problemen“
- 7.2 Komplexitätsklassen P, NP und NPC
- 7.3 NP-vollständige Probleme
- 7.4 Noch mehr NP-vollständige Probleme

### ■ Kapitel 8: Lösen schwerer Probleme

- 8.1 Approximationsalgorithmen
- 8.2 Exakte Verfahren
- 8.3 Heuristische Verfahren



## Kapitel 1: Algorithmen und deren Komplexität

- 1.1 Beschreibung von Algorithmen 1
- 1.2 Analyse von Algorithmen 5
- 1.3 O-Notation 7
- 1.4 Laufzeitanalysen 9
- 1.5 Aymptotische Laufzeit rekursiver Funktionen 11
- 1.6 Ein Beispiel: Das Maximum-Subarray-Problem 15

## Kapitel 1: Algorithmen und deren Komplexität

Beschreibung von Algorithmen  
Analyse von Algorithmen  
O-Notation  
Das Maxsum-Subarray-Problem

## 1.1 Beschreibung von Algorithmen

### ■ Informatik = Information + Mathematik

- entstand mit der Entwicklung der ersten Rechenanlagen (40'er J.)
- anglo-amerikanisch: Computer Science
- Wissenschaft vom „mechanischen Rechnen“

### ■ Algorithmus (von Al-Chowarizmi, persischer Math., ca. 780) = mechanisch ausführbares Rechenverfahren

BSP [Algorithmus]: GGT (Euklid, 300 v. Chr.)

- ggt(a,b):
1.  $a = b \cdot q + r$  mit  $r < b$  (ganzzahlige Division)
  2. falls  $r=0$ : output b
  3.  $a \leftarrow b; b \leftarrow r;$
  4. gehe zu Schritt 1.



© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al,  
Oldenbourg Verlag, 2004

2

## Der Algorithmenbegriff

### Begriffe

- „**Problem**“: definiert eine Eingabe-Ausgabe-Beziehung
- „**Instanz**“: eine mögliche Eingabe für das Problem
- „**Algorithmus**“: definiert eine Folge elementarer Anweisungen zur Lösung eines Problems
- „**Korrektheit**“: Ein Algorithmus **stoppt** für **jede** mögliche Eingabe mit der korrekten Ausgabe

### ■ Eigenschaften von Algorithmen:

- mechanische Verfahren
- bestehen aus mehreren elementaren Schritten
- Schritte werden ggf. wiederholt durchlaufen [Iteration]
- Schritte werden ggf. bedingt durchlaufen [Selektion]
- Das Verfahren führt sich ggf. selbst mit veränderten Parametern aus [Rekursion]



© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al,  
Oldenbourg Verlag, 2004

3

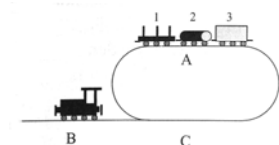
## Beschreibung von Algorithmen

### ■ Spezifikation: **Beschreibung des Problems**

- vollständig: alle Anforderungen und Rahmenbedingungen
- detailliert: welche Hilfsmittel / Basisoperation sind erlaubt
- unzweideutig: klare Kriterien für akzeptable Lösungen

### ■ Bsp: Eine Lokomotive soll die auf Gleis A stehenden Wagen 1,2,3 in Reihenfolge 3,1,2 auf Gleis C abstellen.

- Vollständigkeit:
  - ◆ Wie viele Wagen kann die Lok auf einmal ziehen?
- Detailliertheit:
  - ◆ Welche Aktionen kann die Lok ausführen?
- Unzweideutigkeit:
  - ◆ Darf die Lok am Ende zwischen den Wagen stehen?



### ■ Beschreibung durch Vorbedingung / Nachbedingung



© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al,  
Oldenbourg Verlag, 2004

4

## Beschreibung von Algorithmen

### ■ Beschreibungsformen:

1. Natürliche Sprache
2. Computerprogramme
3. Hardwareentwurf
4. Mischung aus 1. und 2. → Pseudo-Code

Regel: Wie die Spezifikation muss der Algorithmus vollständig, detailliert und unzweideutig beschrieben sein.

### ■ Pseudo-Code:

- Angelehnt an imperative Programmiersprachen (Pascal, C, ...)
- Kontroll- und Datenstrukturen werden aus P-Sprachen übernommen
- Bedingungen, Funktionen werden ggf. natürlich-sprachlich formuliert



## Beschreibung von Algorithmen

### ■ Pseudo-Code Konventionen (aus Cormen et al)

1. Einrücken kennzeichnet die Blockstruktur
2. **if-then-else** Anweisungen für die bedingte Ausführung
3. **while-do, for-to, repeat-until** Anweisungen für die iterative Ausführung
  - **while** <Bedingung ist wahr>
  - **for** <variable>  $\leftarrow$  <Wert/Ausdruck> **to** <Wert/Ausdruck>  
do Anw1  
Anw2 ...
  - **repeat** Anw1  
Anw2 ...  
**until** <Bedingung ist wahr>
  - *Achtung: Variable der for-Schleife ist auch noch nach Schleifenende definiert (C-Konvention)*
4.  $\triangleright$ , // leiten Kommentare ein
5. = steht für den Vergleich von Ausdrücken,  $\leftarrow$  für die Zuweisung



## Beschreibung von Algorithmen

### ■ Pseudo-Code Konventionen

6. Feldelemente werden durch eckige Klammern indiziert, Teilfelder durch Indexbereiche,  $A[i]$ ,  $A[i..j]$
7. Zusammenhängende Daten werden als Objekte mit Attributen dargestellt, das Objekt wird über eckige Klammern spezifiziert,  $länge[A]$  bezeichnet die Anzahl der Elemente von Array A
8. Funktionsparameter werden als Wert übergeben (call-by-value)
9. Boole'sche Operatoren werden träge ausgewertet (lazy evaluation), d.h. von links nach rechts bis der Ausdruck garantiert falsch oder wahr ist.  
Bsp:  $x$  und  $y$ :  $y$  wird nur ausgewertet, wenn  $x$  wahr ist.  
 $x$  oder  $y$ :  $y$  wird nur ausgewertet, wenn  $x$  falsch ist.



## Beispiel: Euklids GGT-Algorithmus

### ■ Algorithmus zur Berechnung des GGT:

BSP [Algorithmus]: GGT (Euklid, 300 v.Chr.)

$ggt(a, b)$ :  
1.  $a = b \cdot q + r$  mit  $r < b$  (ganzzahlige Division)  
2. falls  $r=0$ : output  $b$   
3.  $a \leftarrow b$ ;  $b \leftarrow r$ ;  
4. gehe zu Schritt 1.

Iterative Variante:

Function  $ggt(a, b)$  // Annahme:  $a > b$   
while  $a \bmod b > 0$  do  
     $r \leftarrow a \bmod b$   
     $a \leftarrow b$ ,  $b \leftarrow r$   
output  $b$

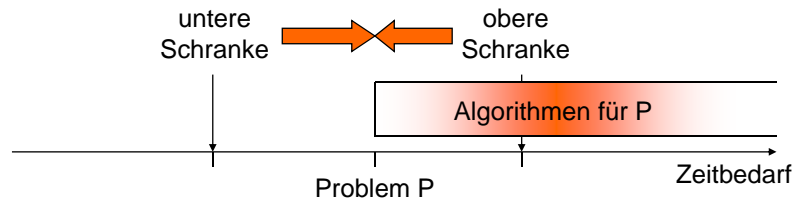
Rekursive Variante:

Function  $ggt(a, b)$  // Annahme:  $a > b$   
     $r \leftarrow a \bmod b$   
    if  $r > 0$  then output  $ggt(b, r)$   
    else output  $b$



## 1.2 Analyse von Algorithmen

- **Ziel:** theoretische (d.h. ohne ein Computerprogramm zu schreiben) Analyse von Problemen und Algorithmen
  - Welche Probleme sind lösbar?
  - Welche Unterschiede gibt es in der Mächtigkeit von Computern?
  - Welche Ressourcen (Zeit, Speicherplatz) werden mindestens benötigt?
  - **Ist der Algorithmus für das Problem korrekt?**
  - **Welche Ressourcen benötigt ein gegebener Algorithmus?**



## Korrektheit von Algorithmen

- **Formale Korrektheit**
  - Angabe von Vor- und Nachbedingung für jede Anweisung
  - Schleifeninvariante: Bedingung, die vor, während (d.h. nach jeder Iteration) und nach Ausführung einer Schleife gültig ist
- **Bsp: Berechnung von  $a^k$  für  $k > 0$** 
  - $b \leftarrow 1, i \leftarrow 0$
  - for**  $i \leftarrow 1$  **to**  $k$
  - do**  $b \leftarrow b * a$
  - return**( $b$ )
  - Invariante:  $\{b = a^i\}$
- **Beweistechniken sind analog zur Mathematik**
  - Insbesondere: vollständige Induktion, Beweis durch Widerspruch



## Asymptotische Laufzeit

- **physikalische Laufzeit**
  - hängt stark vom Computer ab
  - hängt von vielen Details der Eingabedaten ab
- **Modellannahmen Laufzeit**
  - **Uniformes Kostenmaß:** Math. Operationen kosten unabhängig von der Größe der Operanden eine Zeiteinheit
  - **RAM-Modell** (Random-Access-Maschine):
    - ◆ Zugriff auf Daten kostet eine konstante Zeiteinheit
    - ◆ Algorithmen werden sequentiell ausgeführt (nur ein Prozessor)
  - Statt der genauen Laufzeit wird eine Schranke angegeben
  - Konstante Faktoren werden vernachlässigt.
- **Modellannahmen Speicherplatz**
  - **RAM-Modell:** Speicherung eines elementaren Datenobjekts kostet unabhängig vom Wert konstanten Speicherplatz



## Asymptotische Laufzeit

- **Wie können wir die Effizienz eines Algorithmus unabhängig von Details der Eingabe bewerten?**
  - Instanzen (Eingaben) verursachen aufgrund
    - 1. der Größe
    - 2. der individuellen Werteunterschiedliche physikalische Laufzeiten.
  - Bsp: Sortieren einer Zahlenfolge
    - 1. Wie viele Zahlen sollen sortiert werden?
    - 2. In welcher initialen Reihenfolge liegen die Zahlen vor?
- **Asymptotische Laufzeit:**
  - Wie verhält sich der Algorithmus bei immer größeren Instanzen?
    - ◆ im **worst case**: im ungünstigsten Fall
    - ◆ im **average case**: im statistischen Mittel
    - ◆ im **best case**: im besten Fall
    - (bzgl. der Menge aller Instanzen gleicher Länge)
  - Falls nichts weiter angegeben ist, bezieht sich eine Laufzeitanalyse immer auf den Worst Case.



## 1.3 O-Notation

### ■ Größenordnungen von Funktionen

f(N)	Bezeichnung	10	1.000	1.000.000
1	konstant	1	1	1
log(N)	logarithmisch	3	10	20
log <sup>2</sup> (N)	log-quadrat	10	100	400
√N		3	30	1000
N	linear	10	1.000	1.000.000
N log(N)		30	10.000	20.000.000
N <sup>2</sup>	quadratisch	100	1.000.000	10 <sup>12</sup>
N <sup>3</sup>	kubisch	1000	10 <sup>9</sup>	10 <sup>18</sup>
2 <sup>N</sup>	exponentiell	1000	10 <sup>300</sup>	10 <sup>300000</sup>

Hinweis: zur Berechnung der Beispielzahlen wurde log<sub>2</sub>() verwendet.



## O-Kalkül / O-Notation

- O-Kalkül: Eine Funktion  $f$  ist **höchstens von der Ordnung  $g$** , falls Konstanten  $c$  und  $n_0$  existieren mit  $0 \leq f(n) \leq c g(n)$  für alle  $n > n_0$ , d.h.  
 $\exists c > 0 \exists n_0 > 0 \forall n > n_0 : 0 \leq f(n) \leq c g(n)$
- Man schreibt  $f(n) = O(g(n))$ .
  - $O(g(n))$  ist die **Menge** der Funktionen, die nicht stärker wachsen als  $g$  (= hat die Bedeutung von  $\in$ , mit  $O$  ist manchmal eine Menge, manchmal ein Repräsentant gemeint)

$O$	$\exists c \dots$ mit $f(n) \leq c g(n)$	... $f$ ist höchstens von Ordnung $g$
$o$	$\forall c \dots$ mit $f(n) < c g(n) \dots$	... $f$ ist von echt kleinerer Ordnung als $g$
$\Omega$	$\exists c \dots$ mit $f(n) \geq c g(n) \dots$	... $f$ ist mindestens von Ordnung $g \dots$
$\omega$	$\forall c \dots$ mit $f(n) > c g(n) \dots$	... $f$ ist von echt größerer Ordnung als $g$
$\Theta$	$\exists c_1, c_2$ mit $c_1 g(n) \leq f(n) \leq c_2 g(n)$	... $f$ ist von Ordnung $g \dots$



## O-Notation (O-Kalkül)

### ■ Rechenregeln für $O$

1.  $f = O(f)$
2.  $f, g = O(f) \Rightarrow f + g = O(f)$
3.  $f = O(f)$  und  $c$  konstant  $\Rightarrow c * f = O(f)$
4.  $f = O(f)$  und  $g = O(f) \Rightarrow g = O(f)$
5.  $f = O(f)$  und  $g = O(g) \Rightarrow f * g = O(f * g)$
6.  $f = O(f * g) \Rightarrow f = |f| * O(g)$
7.  $f = O(f)$  und  $|f| \leq |g| \Rightarrow f = O(g)$

### ■ Beweis zu 5. (andere Beweise erfolgen analog):

- $f(n) = O(F(n))$ , d.h.  $\exists n_0, c_0 \forall n \geq n_0 : f(n) \leq c_0 F(n)$
- $g(n) = O(G(n))$ , d.h.  $\exists n_1, c_1 \forall n \geq n_1 : g(n) \leq c_1 G(n)$
- Sei  $h(n) = f(n) * g(n)$ . Dann gilt  $\forall n \geq n_2 = \max(n_0, n_1)$ :  
 $f(n) * g(n) \leq c_0 F(n) c_1 G(n) = c_2 F(n) G(n)$ , also  $f * g = O(F * G)$



## O-Notation

### ■ Polynome:

- Ist  $p$  ein Polynom von Grad  $m$  gilt:  $p = O(N^m)$   
 $\diamond n^k = O(N^l)$  für alle  $l \geq k$ , Anwendung von Regel 2

### ■ Weitere Beispiele:

- $f(n) = 3n^2 + 17\sqrt{n} = O(N^2)$   
 $\diamond \sqrt{n} = O(N)$ , Anwendung von Regel 4 und 2
- $f(n) = 10^{300}n + 2n \log(n) = O(N \log N)$
- $f(n) = 2^{2^n} = (2^2)^n = 4^n = O(4^N)$
- $f(n) = \log_{10} n = O(\log N)$   
 $\diamond \log_{10} n = \log_2 n / \log_2(10) = 1/\log_2(10) * \log_2 n = O(\log_2 N) = O(\log N)$   
 $\diamond$  Eine Änderung der Basis führt zu einem konstanten Faktor, die Basis muss im O-Kalkül nicht berücksichtigt werden.





## O-Notation

### ■ O-Notationen in Gleichungen und Ungleichungen

1. Was bedeutet  $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$  ?

- ◆  $\Theta(n)$  bezeichnet eine Menge, gemeint ist:  
 $2n^2 + 3n + 1 = 2n^2 + f(n)$  mit  $f(n) = \Theta(n)$
- ◆  $\Theta(n)$  repräsentiert eine *anonyme Funktion*, d.h. der genaue Funktionsverlauf ist nicht bekannt, lediglich das Wachstumsverhalten.

2. Achtung: O-Notationen in parametrisierten Summen vermeiden!

$$\sum_{i=1}^n O(i) \neq O(1) + O(2) + \dots + O(n)$$

3. Was bedeutet  $2n^2 + \Theta(n) = \Theta(n^2)$  ?

- ◆  $\Theta(n)$  repräsentiert eine anonyme Funktion mit linearem Wachstum  $f(n)$
- ◆ Für jede Funktion  $f(n) = \Theta(n)$  gibt es eine Funktion  $g(n) = \Theta(n^2)$  und Konstanten  $c_1 > 0$ ,  $c_2 > 0$ ,  $n_0 > 0$  mit  
 $c_1 g(n) \leq 2n^2 + f(n) \leq c_2 g(n)$  für alle  $n > n_0$



## 1.4 Laufzeitanalysen

■ Gegeben ist ein Algorithmus A mit Eingabe der Länge N  
Gesucht wird die Laufzeit des Algorithmus:  $T_A(N)$  oder  $T(N)$

### ■ Welche Operation dauert wie lang?

- math. Operationen; Zuweisungen  $O(1)$
- uniformes Kostenmaß:  $O(1)$
- [logarithmisches Kostenmaß:  
x ist der Wert des größten Operanden]  $O(\log x)$
- Klammerung von Anweisungen  $O(1)$
- Bedingte Ausführungen  $O(1)$
- Schleifen  $O(\text{\#Schleifendurchläufe})$
- Funktionsaufrufe  $O(1)$
- Rekursion: Aufruf der eigenen Funktion mit  
Eingabe der Länge  $N'$   $T(N')$



## Laufzeitanalysen

### ■ Bsp: Berechnung von $a^k$ für $k > 0$

exponent(a, k)

b ← 1

for i ← 1 to k do

b ← b \* a

output b

$$T(a, k) = c_0 + (k+1) * c + k * c_1 + c_2 = O(k)$$

$c_0$

c, k+1 Vergl., k Durchläufe

$c_1$

$c_2$

### ■ Bsp: Berechnung von $a^k$ für $k > 0$

1 exponent2(a, k)

2 p ← 1, q ← a, l ← k,

3 while l ≥ 1 do

4 if l MOD 2 = 1 then p ← p \* q

5 l ← l DIV 2

6 q ← q \* q

7 output p

$$T(a, k) = c_0 + (\log_2 k + 1) (c + c_1 + c_2 + c_3) + c_4 = O(\log k)$$

// Invariante:  $a^k = p * q^l$

$c_0$

c,  $(\log_2 k + 1)$  Durchläufe

$c_1$

$c_2$

$c_3$

$c_4$



## Korrektheit von exponent2()

### ■ Schleifeninvariante: $a^k = p * q^l$

### ■ Beweis der Korrektheit:

■ Vor Schleifenbeginn (Zeile 2) gilt:

$$p=1, q=a, l=k \Rightarrow a^k = p * q^l$$

■ Nach jedem Schleifendurchlauf (nach Zeile 6) gilt:

vor Zeile 3 gilt  $a^k = p * q^l$  (Schleifeninvariante)

Fall 1: l ist gerade, d.h.  $l \bmod 2 = 0$

Seien  $l'$  und  $q'$  die Werte von l und q vor Zeile 4. Dann gilt nach

$$\text{Zeile 6: } a^k = p q^{l'} = p (q'^2)^{l'/2} = p q^l$$

Fall 2: l ist ungerade, d.h.  $l \bmod 2 = 1$

Seien  $p'$ ,  $l'$ ,  $q'$  die Werte von p, l, und q vor Zeile 4, Dann gilt nach

$$\text{Zeile 6: } a^k = p' q'^{l'} = p' q' (q'^2)^{(l'-1)/2} = p' q' q^l = p q^l$$

■ Nach Schleifenende (Zeile 7) gilt:

$a^k = p * q^l$  und  $l=0$  (Schleifenterminierung), also  $a^k = p$



## 1.5 Asymptotische Laufzeit rekursiver Funktionen

- **Rekurrenz: Gleichung/Ungleichung, die durch sich selbst mit kleinerem Eingabewert beschrieben wird:**

- **Bsp:**

$$T(n) = \begin{cases} c_0 & : n = 1 \\ T(n-1) + c_1 & : n > 1 \end{cases}$$

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

- **Eigenschaften:**

- $T(n)$  ist typischerweise nur für  $n \in \mathbb{N}$  definiert
- $T(n) = c$  für kleine  $n$
- Auf- und Abrundungen für  $n$  können i.d.R. vernachlässigt werden.

- **Ziel:**

1. bestimme die asymptotische Laufzeit
2. finde eine geschlossene Formel für  $T(n)$



## Substitutionsmethode

- **Schritt 1: Setze  $T(n)$  wiederholt ein:**

$$\begin{aligned} T(n) &= T(n-1) + c_1 \\ &= T(n-2) + c_1 + c_1 \\ &= T(n-(n-1)) + \underbrace{c_1 + c_1 + \dots + c_1}_{n-1 \text{ mal}} \\ &= T(1) + (n-1)c_1 = (n-1)c_1 + c_0 = O(n) \end{aligned}$$

$$\begin{aligned} T(n) &= 2T(n/2) + c \\ &= 2(2T(n/4) + c) + c \\ &= 4T(n/4) + 2c + c \\ &= 2^i T(n/2^i) + 2^{i-1}c + 2^{i-2}c + \dots + c \end{aligned}$$



## Substitutionsmethode

- **Schritt 2: Intuition**

$$T(n) = 2^i T(n/2^i) + 2^{i-1}c + 2^{i-2}c + \dots + c$$

$$= 2^i T(n/2^i) + \sum_{k=0}^{i-1} 2^k c$$

- Wie viele Substitutionsschritte sind notwendig?

$$n/2^i \leq 1 \Leftrightarrow n \leq 2^i \Leftrightarrow \log_2 n \leq i$$

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + c \sum_{k=0}^{i-1} 2^k \\ &\leq 2^{\log_2 n} T(1) + c \sum_{k=0}^{\log_2 n} 2^k \end{aligned}$$

$$= nc + c(2^{\log_2 n+1} - 1) = nc + c(2n - 1) = 3cn - c = O(n)$$

- Beweis erfolgt durch vollständige Induktion

Geometrische / Exponentielle Reihe:

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1} \quad \text{für } x \neq 0, x \in \mathbb{R}$$



## Substitutionsmethode

- **Schritt 3: Beweis der Hypothese** (durch vollständige Induktion)

- Annahme:  $T(n) \leq kn$
- Induktionsanfang:  $T(1) = c \leq k \cdot 1$  für  $k \geq c$
- Induktionsschritt:

$$T(n) = 2T(n/2) + c \leq 2kn/2 + c \leq kn + c$$



- Neue Annahme:  $T(n) \leq kn - c$
- Induktionsanfang:  $T(1) = c \leq k \cdot 1 - c$  für  $k \geq 2c$
- Induktionsschritt:

$$T(n) = 2T(n/2) + c \leq 2(kn/2 - c) + c \leq kn - c$$

- In der Regel werden wir unserer Intuition vertrauen, formal ist der Beweis aber notwendig.



## Hilfsmittel: Variablentransformation

- Transformation der Variablen hilft häufig, einfachere, intuitiv leichter lösbare Gleichungen zu erhalten:
- Bsp:

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

- Ersetze  $m = \log n$

$$T(n) = 2T(\sqrt{2^{\log n}}) + \log n$$

$$T(2^m) = 2T(2^{m/2}) + m$$

- Setze  $S(m) = T(2^m)$

$$S(m) = 2S(m/2) + m = O(m \log m)$$

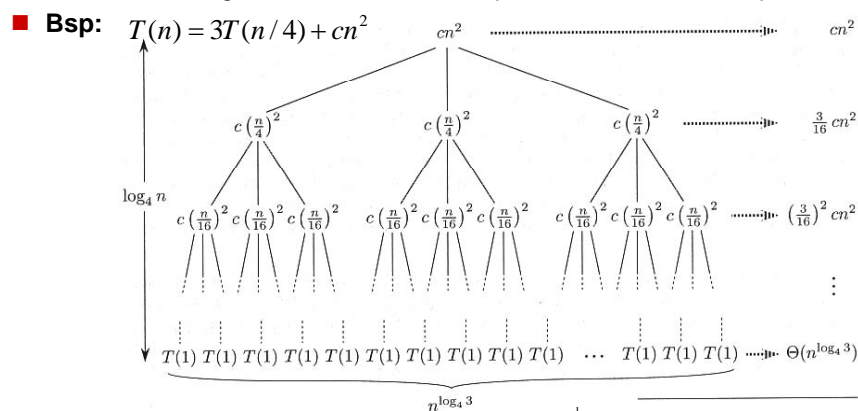
$$T(n) = O(\log n \log \log n)$$



## Hilfsmittel: Rekursionsbaum

- Was tun, wenn die Intuition fehlt?

- Aufbau des Rekursionsbaums
- Abschätzung der Höhe, der Knoten pro Ebene, der Kosten pro Knoten



Hinweis:  $3^{\log_4 n} = 3^{\frac{\log_2 n}{\log_2 4}} = 3^{\frac{\log_2 n}{2}} = \sqrt{3^{\log_2 n}} = n^{\frac{\log_2 3}{2}} = n^{\log_4 3}$



## Hilfsmittel: Rekursionsbaum

- Was tun, wenn die Intuition fehlt?

- Aufbau des Rekursionsbaums
- Abschätzung der Höhe, der Knoten pro Ebene, der Kosten pro Knoten

- Bsp:  $T(n) = 3T(n/4) + cn^2$

$$= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$\leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

Unendl. Geom. / Exp. Reihe:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{für } x \in \mathbb{R}, |x| < 1$$



## Das Master-Theorem

- Auflösung von Rekurrenzen der Form

$$T(n) = \begin{cases} c & : n = 1 \\ aT(n/b) + f(n) & : n > 1 \end{cases}$$

mit  $a \geq 1$  und  $b > 1$ ,  $a$  und  $b$  konstant

- Algorithmische Bedeutung:

- Ein Problem wird in  $a$  Teilprobleme zerlegt
- Jedes Teilproblem hat die Größe  $n/b$  (genauer  $\lceil n/b \rceil$ )
- Zum Aufteilen des Problems und zum Zusammenfügen benötigt man  $f(n)$  Zeit.

Fall 1.  $f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0$  :  $T(n) = \Theta(n^{\log_b a})$

Fall 2.  $f(n) = \Theta(n^{\log_b a})$  :  $T(n) = \Theta(n^{\log_b a} \log_2 n)$

Fall 3.  $f(n) = \Omega(n^{\log_b a + \epsilon}), \epsilon > 0$  :  $T(n) = \Theta(f(n))$   
und  $af(n/b) \leq cf(n), c < 1$



## Das Master-Theorem

### ■ Bsp:

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

$$a = 2, b = 2, f(n) = c$$

### ■ Welcher Fall des Master Theorems?

$$\log_b a = 1 \text{ für } a = b = 2 \Rightarrow f(n) = c = O(n^{1-\epsilon})$$

### ■ Fall 1, also folgt

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$$



## 1.6 Ein Beispiel: Das Maximum-Subarray-Problem

[aus Ottmann, Widmeyer, Algorithmen und Datenstrukturen, Spektrum Verlag, 2002]

### ■ Problem (maximum-subarray):

- geg.: eine Folge X von N ganzen Zahlen
- ges.: Teilfolge, deren Summe maximal ist (max. Teilsumme)

### ■ Bsp.:

- N=10, X: 3, -4, 5, 2, -5, 6, 9, -9, -2, 8 (d.h. X[1]=3, X[2]=-4, ...)
- Teilsumme von X[1..4] : 3-4+5+2 = 6
- max. Teilsumme X[3..7] : 5+2-5+6+9 = 17

### ■ Algorithmus 1: Probiere alle Möglichkeiten

- u: untere Grenze der Teilsumme
- o: obere Grenze der Teilsumme
- tsum: aktuelle Teilsumme
- maxtsum: maximale Teilsumme



## Das Maximum-Subarray-Problem

### ■ Algorithmus 1: Probiere

teilsomme(X,N)	Zeitbedarf
maxtsum ← 0	c
for u ← 1 to N do	N
for o ← u to N do	N-u+1 ≤ N
tsum ← 0	c
for i ← u to o do	o-u+1 ≤ N
tsum ← tsum + X[i]	c
maxtsum ← max( maxtsum, tsum )	c
output maxtsum	c

- Laufzeit: offensichtlich gilt  $T_p(N) = O(N^3)$   
es gilt sogar  $T_o(N) = \Theta(N^3)$  (ohne Beweis)

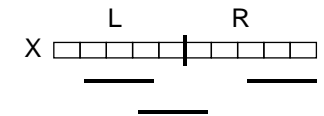


## Das Maximum-Subarray-Problem

### ■ Verbesserung 1: Divide & Conquer Verfahren (Teile und Herrsche)

#### ■ Idee:

- Teile die Folge in linke und rechte Hälfte L, R an der Stelle  $\lfloor N/2 \rfloor$
- Fall 1: max. Teilsumme ist in L
- Fall 2: max. Teilsumme ist in R
- Fall 3: max. Teilsumme enthält  $X[\lfloor N/2 \rfloor]$  und  $X[\lfloor N/2 \rfloor + 1]$



#### ■ Teilproblem: maximale Randteilsumme

```

RTS_links(X,l,r)
// finde max. Randteilsumme am linken Rand in X[l,...,r]
lmax ← 0; sum ← 0;
for i ← l to r do
    sum ← sum + X[i]
    lmax ← max( lmax, sum )
return ( lmax )
    
```

Laufzeit:  $T_{RTS}(N) = O(N)$



## Das Maximum-Subarray-Problem

### ■ Algorithmus 2: Divide&Conquer (Aufruf Teilsumme(X,1,N))

Teilsumme(X,l,r)	Laufzeit $T_{DC}(N)$
if l = r then	
output max(X[l],0)	c
else // DIVIDE: Teile die Daten	
$m \leftarrow \lfloor (l+r)/2 \rfloor$	c
// CONQUER: Löse Teilprobleme	
Fall 1: maxtsum_Links $\leftarrow$ Teilsumme(X,l,m)	$T_{DC}(N/2)$
Fall 2: maxtsum_Rechts $\leftarrow$ Teilsumme(X,m+1,r)	$T_{DC}(N/2)$
Fall 3: maxtsum_LRand $\leftarrow$ RTS_rechts(X,l,m)	c N
maxtsum_RRand $\leftarrow$ RTS_links(X,m+1,r)	c N
maxtsum_Mitte $\leftarrow$ maxtsum_LRand + maxtsum_RRand	c
% MERGE: Füge Ergebnis zusammen	
maxtsum $\leftarrow$ max(maxtsum_Links, maxtsum_Rechts,	c
maxtsum_Mitte)	
output maxtsum	



## Das Maximum-Subarray-Problem

### ■ Rekurrenz für Divide&Conquer:

$$T_{DC}(N) = \begin{cases} c & : N = 1 \\ 2T_{DC}(N/2) + cN & : N > 1 \end{cases}$$

### ■ Master-Theorem: $a = b = 2$ ; $f(n) = cN = O(N)$

$$T_{DC}(N) = \Theta(N \log N)$$

### ■ Wie viel Zeit benötigt man mindestens zur Lösung des Problems?

- Man muss jedes  $X[i]$  mindestens ein mal betrachten
- $T_{opt}(N) = \Omega(N)$

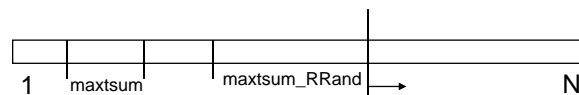


## Das Maximum-Subarray-Problem

### ■ Verbesserung 2: Scan-Line Verfahren

#### ■ Idee:

- durchlaufe X von links nach rechts
- merke dir das bisherige Maximum maxtsum
- merke dir die rechte Randteilsumme maxtsum\_RRand



Scan-Line

- Scan-Line am linken Rand: maxtsum = maxtsum\_RRand = 0
- Scan-Line geht von i nach i+1:
  - ◆ maxtsum\_RRand: addiere  $X[i+1]$ ; setze auf 0 falls negativ
  - ◆ maxtsum: Maximum von maxtsum oder maxtsum\_RRand
- Scan-Line ist rechts angekommen: maxtsum ist das Ergebnis



## Das Maximum-Subarray-Problem

### ■ Algorithmus 3: Scan-Line-Verfahren

Teilsumme(X,N)	Laufzeit $T_{SL}(N)$
maxtsum $\leftarrow$ 0	c
maxtsum_RRand $\leftarrow$ 0	c
for i $\leftarrow$ 1 to N do	N
maxtsum_RRand $\leftarrow$ max(maxtsum_RRand + $X[i]$ , 0)	c
maxtsum $\leftarrow$ max(maxtsum, maxtsum_RRand)	c
output maxtsum	

$$T_{SL}(N) = cN = \Theta(N)$$

### ■ Das Scan-Line Verfahren löst das Maximum-Subarray-Problem in optimaler asymptotischer Laufzeit.



## **Kapitel 2: Elementare Datenstrukturen**

- 2.1 Elementare und Strukturierte Datentypen 21
- 2.2 Stapel 22
- 2.3 Lineare Listen 28
- 2.4 Bäume 36

## Kapitel 2: Elementare Datenstrukturen

Elementare und strukturierte Datentypen  
Stapel und Warteschlangen  
Listen  
Bäume

## 2.1 Elementare und Strukturierte Datentypen

### ■ Abstrakter Datentyp (ADT) / Datenstruktur:

- Ein oder Mehrere Objekt(e) (Beschreibung der Daten) und
- Operationen (Manipulation der Daten)

### ■ Beispiel: Datum

- Objekt D: Tag.Monat.Jahr 1.11.2002
- Objekt T: Tage 17 Tage
- Operationen:
  - ◆ Addition:  $D \times T \rightarrow D$  1.11.2002 + 5 Tage  $\rightarrow$  6.11.2002
  - ◆ Subtraktion 1:  $D \times T \rightarrow D$  1.11.2002 - 5 Tage  $\rightarrow$  27.10.2002
  - ◆ Subtraktion 2:  $D \times D \rightarrow T$  1.11.2002 - 27.10.2002  $\rightarrow$  5 Tage
  - ◆ IstFeiertag:  $D \rightarrow \{\text{wahr, falsch}\}$  IstFeiertag(1.11.2002)  $\rightarrow$  falsch (in HH)

### ■ Entwurf von Datentypen (konstruktive Methode):

- Definition der Objekte (Bestehend aus elementaren Datentypen)
- Definition aller Operationen (Operanden, Ergebnis, Spezifikation)



## Elementare und strukturierte Datentypen

### ■ Elementare Datentypen: ADTs, die typischerweise (in einer Programmiersprache) zur Verfügung stehen:

- INTEGER: ganze Zahlen
- REAL: reelle Zahlen
- BOOLEAN: Wahrheitswerte {TRUE, FALSE}
- CHAR: Zeichen

[Achtung: in Cormen et al wird dieser Begriff anders verwendet!]

### ■ Strukturierter Datentyp: aus elementaren (oder strukturierten!) Datentypen zusammengesetzte Datentypen, wichtige Strukturierungsmethoden:

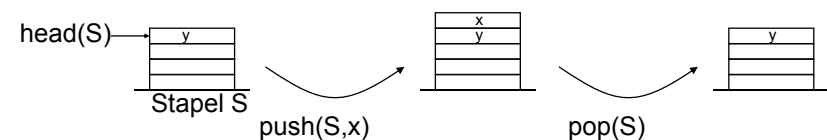
- ARRAY: über natürliche Zahlen indizierte Menge
- RECORD/STRUCT: Gruppierung ggf. verschiedener Datentypen
- ENUM: konstante Wertemenge
- UNION: Vereinigung verschiedener Datentypen

### ■ Referenzen: Verweise auf andere Daten (Zeiger, Adressen)



## 2.2 Stapel

- LIFO-Prinzip (last-in first-out): Speicherung mit Zugriffsmöglichkeit nur auf dem zuletzt gespeicherten Objekt
- Stapel (Stack): linearer Speicher mit folgenden Operationen
  - head(S): Wert des ‚obersten‘ Elements
  - push(S,x): Lege x oben auf den Stapel
  - pop(S): Entferne oberstes Element vom Stapel



### ■ Implementierung: Sequenzielle oder verkettete Speicherung



## Stapel: Sequentielle Speicherung

- Datenstruktur:  
stack S : Array mit Elementen 1,...,MAXE  
top[S] : Index des ‚obersten Elements‘
- EMPTY-STACK(S)  
if top[S] = 0 return TRUE  
else return FALSE
- PUSH(S, x)  
if top[S] = MAXE then error „Überlauf!“  
else top[S] ← top[S]+1  
S[top[S]] ← x
- POP(S)  
if EMPTY-STACK(S) then error „Unterlauf!“  
else top[S] ← top[S]-1  
return S[top[S]+1]
- HEAD(S)  
if EMPTY-STACK(S) then error „Unterlauf!“  
else return S[top[S]]



## Stapel: Anwendung

- Problem: Erkennung wohlgeformter Klammerausdrücke
  - finde zu jeder schließenden Klammer die zugehörige öffnende
  - Eingabe Array brackstr[1,...,nofbrack]
  - Ausgabe Array pairno[1,...,nofbrack]  
(Index der öffnenden/schließenden Klammer)

Beispiel:

```
index:    1  2  3  4  5  6  7  8  9 10 11 12
brackstr: (  (  )  (  )  (  (  )  (  )  )  )
pairno:   12 3  2  5  4 11 8  7 10  9  6  1
```

- Lösung: speichere Index öffnender Klammern in einem Stack



## Stapel: Anwendung

- BRACKETS( brackstr, pairno )  
// brackstr: array[1,...,nofbrackstr] of char (Eingabe)  
// pairno: array[1,...,nofbrackstr] of integer (Ausgabe)  
  
S ← INIT(); // initialisiere Stack S  
for p ← 1 to nofbrackstr do  
if brackstr[p] = '(' then PUSH(S,p)  
else  
if EMPTY-STACK(S) then error „Opening bracket missing!“  
else pairno[p] ← HEAD(S)  
pairno[HEAD(S)] ← p  
POP(S)  
if not EMPTY-STACK(S) ) then error „Closing bracket missing!“  
else return „Format correct.“



## Stapel: Anwendung

Beispiel:

```
index:    1  2  3  4  5  6  7  8  9 10 11 12
brackstr: (  (  )  (  )  (  (  )  (  )  )  )
pairno:   0  0  0  0  0  0  0  0  0  0  0  0
```

↑  
p

```
  2
  1
  --
Stack
p=3
```





## Stapel: Anwendung

Beispiel:

index:	1	2	3	4	5	6	7	8	9	10	11	12
brackstr:	(	(	)	(	)	(	(	)	(	)	)	)
pairno:	0	3	2	0	0	0	0	0	0	0	0	0

↑  
p

2	4
1	1
Stack	Stack
p=3	p=5



## Stapel: Anwendung

Beispiel:

index:	1	2	3	4	5	6	7	8	9	10	11	12
brackstr:	(	(	)	(	)	(	(	)	(	)	)	)
pairno:	0	3	2	5	4	0	8	7	0	9	0	0

↑  
p

2	4	9
1	1	6
Stack	Stack	Stack
p=3	p=5	p=9



## Stapel: Anwendung

Beispiel:

index:	1	2	3	4	5	6	7	8	9	10	11	12
brackstr:	(	(	)	(	)	(	(	)	(	)	)	)
pairno:	12	3	2	5	4	11	8	7	10	9	6	1

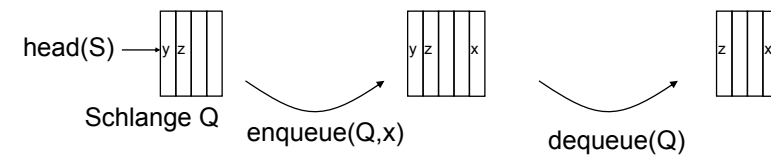
↑  
p

2	4	9	6
1	1	1	1
Stack	Stack	Stack	Stack
p=3	p=5	p=9	p=13



## Schlange

- FIFO-Prinzip (first-in first-out): Speicherung mit Zugriffsmöglichkeit nur auf dem zuerst gespeicherten Objekt
- Schlange (Queue): linearer Speicher mit folgenden Operationen
  - head(Q): Wert des ‚vordersten‘ Elements
  - enqueue(Q,x): Füge x am Ende der Schlange an
  - dequeue(Q): Entferne vorderstes Element der Schlange



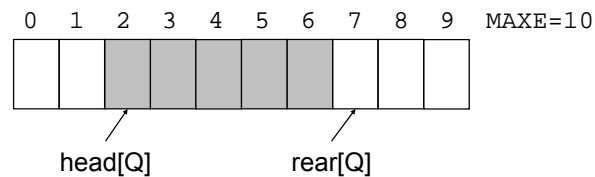
- Implementierung: Sequenzielle oder verkettete Speicherung



## Schlange: Sequentielle Speicherung

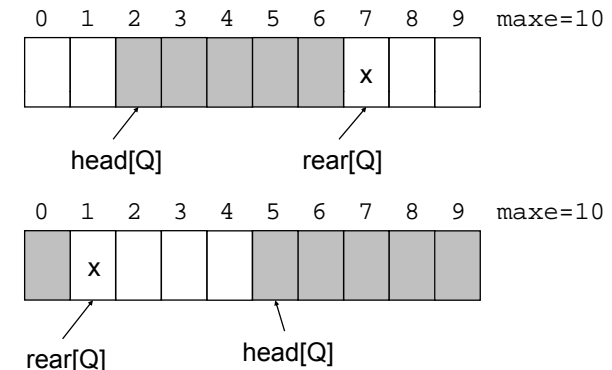
- Datenstruktur:  
 queue Q : Array 0,...,MAXE  
 head[Q] : Position des ersten Elements  
 rear[Q] : Pos. des ersten freien Elements (hinter der Schlange)  
 // ACHTUNG: Implementierung im Cormen mit Array 1..n

- INIT(Q)  
 head[Q]  $\leftarrow$  rear[Q]  $\leftarrow$  1
- EMPTY-QUEUE(Q: queue)  
 if head[Q] = rear[Q] then return TRUE  
 else return FALSE



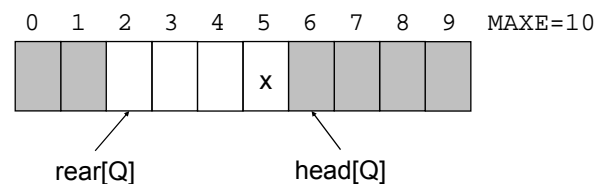
## Schlange: Sequentielle Speicherung

- ENQUEUE ( Q, x )  
 if (rear[Q] + 1) mod MAXE = head[Q] then error „Queue overflow!“  
 else Q[rear[Q]]  $\leftarrow$  x  
 rear[Q]  $\leftarrow$  rear[Q] + 1 mod MAXE



## Schlange: Sequentielle Speicherung

- DEQUEUE( Q )  
 if head[Q] = rear[Q] then error „Queue underflow!“  
 else x  $\leftarrow$  Q[head[Q]]  
 head[Q]  $\leftarrow$  head[Q] + 1 mod MAXE  
 return x



## 2.3 Lineare Listen

- Lineare Liste:
  - Endliche Folge von Elementen eines Grundtyps
  - Elemente haben eine Ordnung:  $a_1, a_2, a_3, \dots, a_n$
  - Grundtyp ist von untergeordneter Bedeutung (hier Integer)
  - Nomenklatur:  $L = \langle a_1, a_2, a_3, \dots, a_n \rangle$ ; leere Liste:  $\langle \rangle$
- Grundoperationen:
  - Einfügen(x,p,L): einfügen von x an Stelle p in L  
 $\langle a_1, \dots, a_p, a_{p+1}, \dots, a_n \rangle \rightarrow \langle a_1, \dots, a_p, x, a_{p+1}, \dots, a_n \rangle$
  - Entfernen(p,L): entfernen des p-ten Elements  
 $\langle a_1, \dots, a_{p-1}, a_p, a_{p+1}, \dots, a_n \rangle \rightarrow \langle a_1, \dots, a_{p-1}, a_{p+1}, \dots, a_n \rangle$
  - Suchen(x,L): Position von Element mit Wert x  
 $\langle a_1, \dots, a_{p-1}, x, a_{p+1}, \dots, a_n \rangle \rightarrow p$
  - Zugriff(p,L): Wert des p-ten Elements  
 $\langle a_1, \dots, a_{p-1}, x, a_{p+1}, \dots, a_n \rangle \rightarrow x$



## Lineare Listen

### ■ weiterführende Operationen:

■ Verketteten( $L_a, L_b$ ): verbindet zwei Listen zu einer

$$\langle a_1, \dots, a_n \rangle \parallel \langle b_1, \dots, b_m \rangle \rightarrow \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle$$

■ Leer( $L$ ): wahr, falls  $L = \langle \rangle$

■ Länge( $L$ ): Anzahl der Elemente in  $L$

$$\langle a_1, \dots, a_n \rangle \rightarrow n$$

■ Anhängen( $L, x$ ): hängt ein neues Element  $x$  an  $L$  an

$$\langle a_1, \dots, a_n \rangle \rightarrow \langle a_1, \dots, a_n, x \rangle$$

■ Kopf( $L$ ), rest( $L$ ): zerlegt eine Liste in erstes Element und Rest

$$\text{kopf: } \langle a_1, \dots, a_n \rangle \rightarrow a_1 \quad \text{rest: } \langle a_1, \dots, a_n \rangle \rightarrow \langle a_2, \dots, a_n \rangle$$

■ Entfernen2( $x, L$ ): Entfernen(Suchen( $x, L$ ),  $L$ )

:

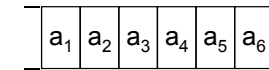
:



## Lineare Listen

■ Wie können lineare Listen am effizientesten realisiert werden?

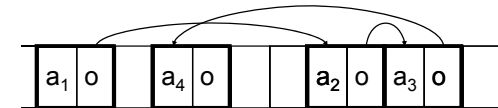
■ Variante 1: Sequentielle Speicherung



Speicher

■ Vorteil: schneller Zugriff    Nachteil: langsames Einfügen

■ Variante 2: verkettete Speicherung



Speicher

■ Vorteil: schnelles Einfügen    Nachteil: langsamer Zugriff, höherer Speicherbedarf



## Lineare Listen: Sequentielle Speicherung

### ■ Datenstruktur:

list  $L$  : Array 0,...,MAXE; Speicherung ab Position 1

size[ $L$ ] : Anzahl Elemente in der Liste

Zugriff	direkt über Index $p$ : $L.\text{element}[p]$	$O(1)$
Suchen	durchlaufe die Liste bis $x$ gefunden wurde	$O(N)$
Einfügen	verschiebe Elemente $p+1, \dots, n$ um eine Position nach hinten	$O(N)$
Entfernen	verschiebe Elemente $p+1, \dots, n$ um eine Position nach vorne	$O(N)$
Verketteten	füge die Elemente von Liste 2 hinter Liste 1 ein	$O(N)$



## Lineare Listen: Sequentielle Speicherung

■ LIST-SEARCH( $x, L$ )

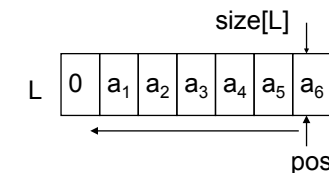
$L[0] \leftarrow x$

$\text{pos} \leftarrow \text{size}[L]$

**while**  $L[\text{pos}] \neq x$  **do**

$\text{pos} \leftarrow \text{pos} - 1$

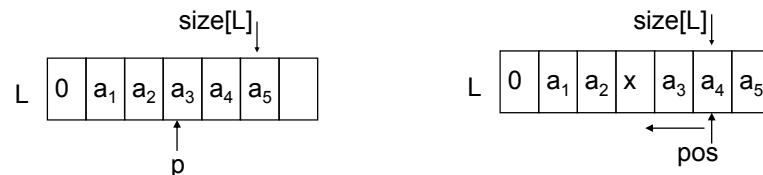
**return**  $\text{pos}$



## Lineare Listen: Sequentielle Speicherung

```

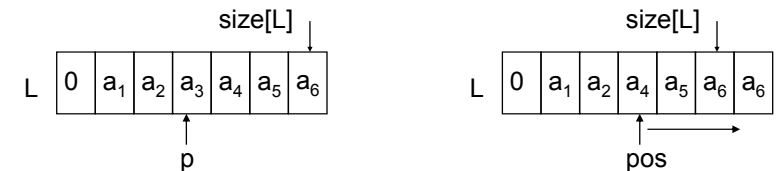
■ LIST-INSERT(x, p, L)
  if size[L] = MAXE then error "List overflow!"
  else
    if (p > size[L] + 1 or p < 1) then error "Invalid position!"
    else
      for pos ← size[L] downto p do
        L[pos+1] ← L[pos]
      L[p] ← x
      size[L] ← size[L] + 1
  
```



## Lineare Listen: Sequentielle Speicherung

```

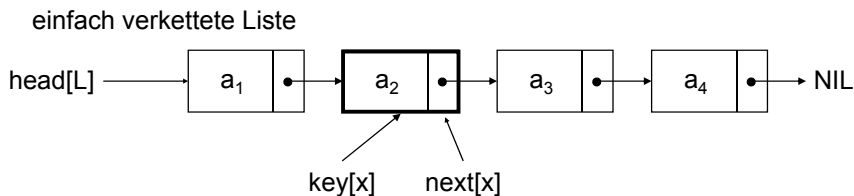
■ LIST-DELETE(p, L)
  if size[L] = 0 then error "Empty list!"
  else
    if (p > size[L] or p < 1) then error "Invalid position!"
    else
      size[L] ← size[L] - 1
      for pos ← p to size[L]
        L[pos] ← L[pos+1]
  
```



## Lineare Listen: Verkettete Speicherung

```

■ Datenstruktur:
List node x : Listenelement
key[x]      : Schlüssel des Elements x
next[x]     : Zeiger auf das Nachfolger-Element von x
head[L]     : Zeiger auf das erste Listenelement
  
```

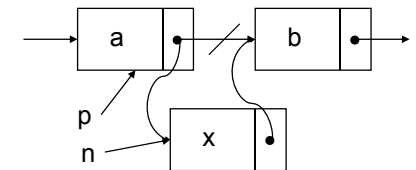


## Lineare Listen: Verkettete Speicherung

■ Durchlaufen der Liste (für Suchen, Einfügen, Entfernen)

```

p ← head[L]
while p ≠ NIL do
  < tue etwas mit key[p] >
  p ← next[p]
  :
  
```



■ Einfügen von x hinter Element p:

```

new(n)
key[n] ← x
next[n] ← next[p]
next[p] ← n
  
```

■ Entfernen hinter Position p:

```

:
d ← next[p]
next[p] ← next[next[p]]
delete(d)
  
```

■ Achtung: besondere Regeln zum Einfügen/Entfernen am Listenanfang

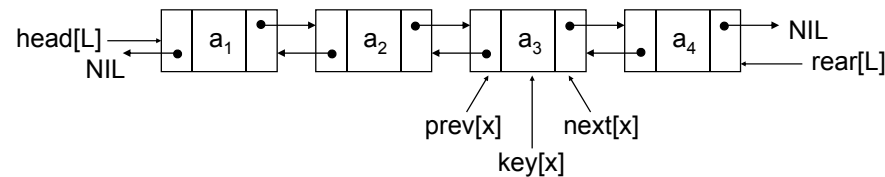


## Lineare Listen: Doppelt verkettete Speicherung

### ■ Datenstruktur:

List node  $x$  : Listenelement  
 key[x] : Schlüssel des Elements  $x$   
 next[x] : Zeiger auf das Nachfolger-Element von  $x$   
 prev[x] : Zeiger auf das Vorgänger-Element von  $x$   
 head[L] : Zeiger auf das erste Listenelement  
 rear[L] : Zeiger auf das letzte Listenelement

### doppelt verkettete Liste

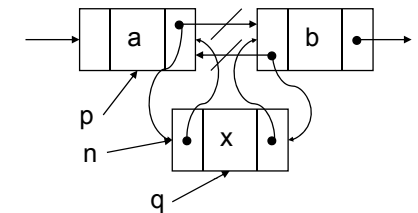


## Doppelt-verkettete Listen

### ■ Durchlaufen der Liste vorwärts und rückwärts möglich

$p \leftarrow \text{head}[L]$   
**while**  $p \neq \text{NIL}$  **do**  
 < tue etwas mit key[p] >  
 $p \leftarrow \text{next}[p]$

rückwärts:  
 $\text{head} \rightarrow \text{rear}$ ,  $\text{next} \rightarrow \text{prev}$



### ■ Einfügen hinter Position p:

$\text{new}(n)$   
 key[n]  $\leftarrow x$   
 $\text{next}[n] \leftarrow \text{next}[p]$   
 $\text{prev}[n] \leftarrow p$   
 $\text{prev}[\text{next}[n]] \leftarrow n$   
 $\text{next}[\text{prev}[n]] \leftarrow n$

### ■ Entfernen an Position q:

:  
 $\text{next}[\text{prev}[q]] \leftarrow \text{next}[q]$   
 $\text{prev}[\text{next}[q]] \leftarrow \text{prev}[q]$   
 $\text{delete}(q)$



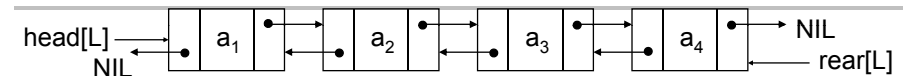
## Lineare Listen: (Doppelt-)verkettete Speicherung

Zugriff	durchlaufe die Liste bis Position p	$O(N)$ $[O(1)]$
Suchen	durchlaufe die Liste bis x gefunden wurde	$O(N)$
Einfügen	erzeuge neues Element, 'verbiege' Zeiger	$O(N)$ $[O(1)]$
Entfernen	'verbiege' Zeiger, lösche das Element	$O(N)$ $[O(1)]$
Verketteten	next-Zeiger des letzten Elements der Liste 1 zeigt auf erstes Element der Liste 2	$O(1)$

[ ] : falls pos-Zeiger bereits an Position p



## Doppelt-verkettete Listen mit Wächter



### ■ Spezieller Code für das Einfügen und Löschen am Listenanfang / -ende notwendig:

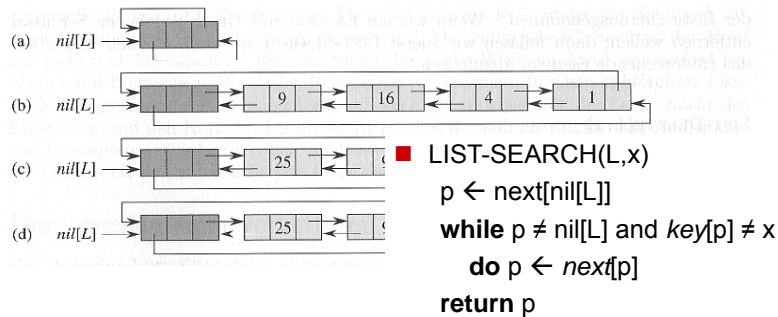
### ■ LIST-INSERT(L, x, p) // fügt Element x hinter p ein

$\text{prev}[x] \leftarrow p$   
**if**  $p = \text{NIL}$  **then** // Listenanfang  
 $\text{next}[x] \leftarrow \text{head}[L]$   
**if**  $\text{head}[L] = \text{NIL}$  **then**  $\text{rear}[L] \leftarrow x$  // Listenanfang und -ende  
**else**  $\text{prev}[\text{head}[L]] \leftarrow x$   
 $\text{head}[L] \leftarrow x$   
**else** // Listenmitte oder -ende  
 $\text{next}[x] \leftarrow \text{next}[p]$   
 $\text{next}[p] \leftarrow x$   
**if**  $\text{next}[x] = \text{NIL}$  **then**  $\text{rear}[L] = x$  // Listenende  
**else**  $\text{prev}[\text{next}[x]] \leftarrow x$   
 $\text{size}[L] \leftarrow \text{size}[L] + 1$



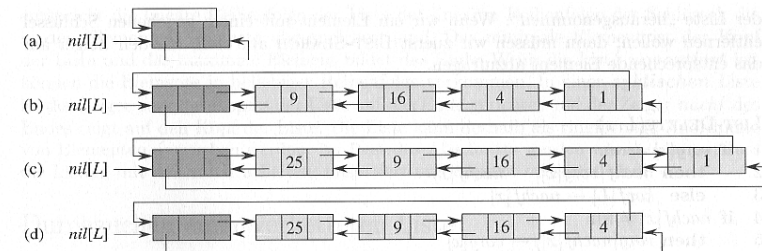
## Doppelt-verkettete Liste mit Wächter

- Wächter  $nil[L]$ : spezielles Listenelement, repräsentiert Listenanfang und -ende
- $next[nil[L]]$ : Listenanfang (=  $head[L]$ )
- $prev[nil[L]]$ : Listenende (=  $rear[L]$ )
- $key[nil[L]]$ : NIL (speichert keinen Wert)



## Doppelt-verkettete Liste mit Wächter

- LIST-INSERT(L, x, p)
  - $next[x] \leftarrow next[p]$
  - $prev[x] \leftarrow p$
  - $next[prev[x]] \leftarrow x$
  - $prev[next[x]] \leftarrow x$
- LIST-DELETE(L, x)
  - $next[prev[x]] \leftarrow next[x]$
  - $prev[next[x]] \leftarrow prev[x]$



## 2.4 Bäume

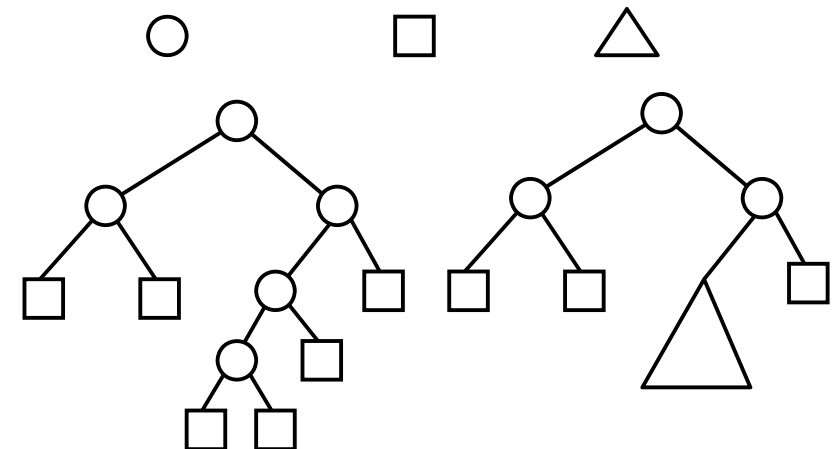
- Datenstruktur (Binär-)Baum:
  - wie Listen, jedoch mit einer endlichen Anzahl (2) Nachfolger
- Nomenklatur:
  - Knoten: Element eines Baums
  - (direkter) Vorgänger: vorheriges Element im Baum (eindeutig!) (Vater, Elter, parent)
  - (direkter) Nachfolger: nachfolgendes Element im Baum (Sohn, Kind, child)
  - Vorfahre: Knoten auf dem Weg zur Wurzel
  - Nachfahre: Knoten auf dem Weg zu einem Blatt
  - geordneter Baum: Nachfolger haben eine feste Reihenfolge (left, right bei Binärbäumen)
  - Wurzel: Knoten ohne Vorgänger
  - innerer Knoten: mit Nachfolger
  - Blatt / externer Knoten: Knoten ohne Nachfolger
  - Ordnung: Anzahl direkter Nachfolger eines Knotens
  - Pfad  $(v_1, \dots, v_k)$ : Folge von Knoten mit  $v_i = \text{parent}(v_{i+1})$



## Bäume

- Nomenklatur:

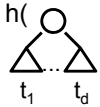
- innerer Knoten
- Blatt
- Teilbaum



## Bäume: Höhen und Tiefen

- Höhe eines Baums: (rekursive Definition)

- $h(\square) = 0$ ;  $h(\bigcirc) = \max\{h(t_1), \dots, h(t_d)\} + 1$



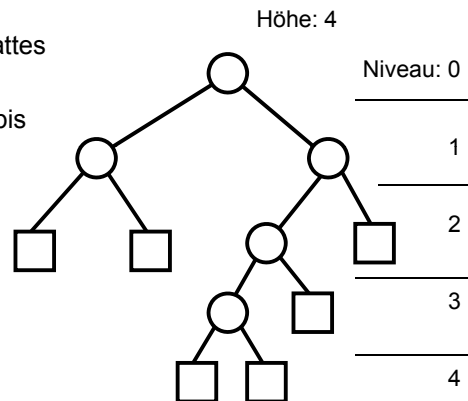
- oder: max. Tiefe eines Blattes

- Tiefe eines Knotens k:

- Anzahl der Kanten von k bis zur Wurzel

- Niveau / Ebene i:

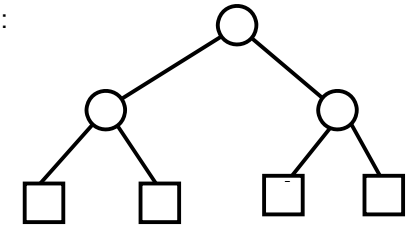
- alle Knoten der Tiefe i



## Bäume: Zahlen

- Vollständiger Binärbaum der Höhe h:

- auf jedem Niveau die maximal mögliche Anzahl Knoten
  - alle Blätter auf Niveau h



- Für einen vollst. Binärbaum der Höhe h gilt:

- Anzahl Knoten auf Ebene i:  $2^i$

- Anzahl der Blätter:  $2^h$

- Anzahl der Knoten:  $|K| = \sum_{i=0}^h 2^i = 2^{h+1} - 1$

- zur Speicherung von  $|K|$  Knoten benötigt man einen Binärbaum der Höhe:

$$h = \log_2 \left( \frac{|K| + 1}{2} \right) = \log_2(|K| + 1) - 1 = \Theta(\log |K|)$$



## Darstellung von binären Bäumen

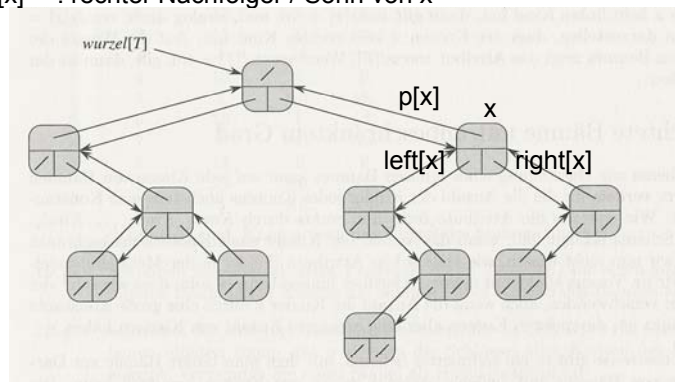
- $\text{root}[T]$  : Wurzel des Baums T

- Sei x ein Knoten des Baumes, dann ist

- $p[x]$  : Vorgänger / Vater von x

- $\text{left}[x]$  : linker Nachfolger / Sohn von x

- $\text{right}[x]$  : rechter Nachfolger / Sohn von x



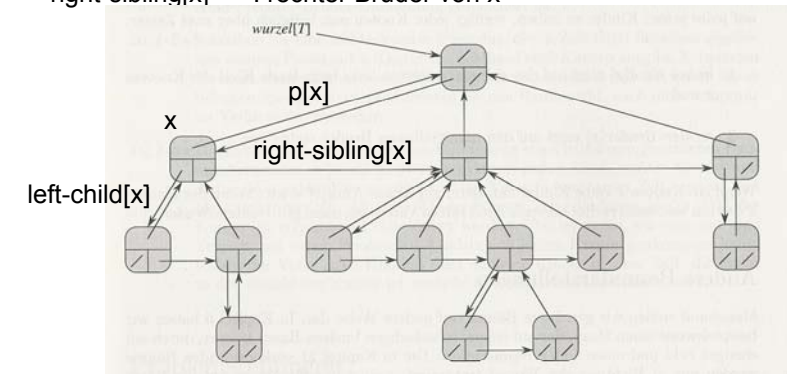
## Darstellung von Bäumen mit unbeschränktem Grad

- Vermeidung von Nachfolger Listen/Arrays durch *left-child/right-sibling* Repräsentation:

- $p[x]$  : Vorgänger / Vater von x

- $\text{left-child}[x]$  : links-stehender Nachfolger von x

- $\text{right-sibling}[x]$  : rechter Bruder von x



## Kapitel 3: Sortieren

- 3.1 Elementare Sortieralgorithmen 41
- 3.2 Heaps und Heapsort 46
- 3.3 Quicksort 53
- 3.4 Eine untere Schranke für das Sortierproblem 59
- 3.5 Counting,- Radix- und Bucketsort 60
- 3.6 Algorithmen für Auswahlprobleme 66



## Kapitel 3: Sortieren

Elementare Sortieralgorithmen  
Heaps und Heapsort  
Quicksort  
Eine untere Schranke für das Sortierproblem  
Counting-, Radix- und Bucketsort  
Algorithmen für Auswahlprobleme

## 3.1 Elementare Sortieralgorithmen

- Problem:
  - Geg. Menge von Datensätzen  $s_i$  mit Schlüsseln  $k_i$ .
  - Geg. totale Ordnungsrelation  $\leq$
  - Ges.: Permutation  $\pi$  mit:  $k_{\pi(1)} \leq k_{\pi(2)} \leq \dots \leq k_{\pi(n)}$
- partielle Ordnung R:
  - reflexiv: es gilt  $a R a$
  - antisymmetrisch:  $a R b$  und  $b R a$  folgt  $a = b$
  - transitiv:  $a R b$  und  $b R c$  folgt  $a R c$
- totale Ordnung R:
  - R ist partielle Ordnung
  - für alle  $a, b$  gilt:  $a R b$  oder  $b R a$
- Bsp: Ordnung auf der Menge der Menschen
  - $R = \text{'ist Nachfahre von'}$  partiell (und nicht reflexiv)
  - $R = \text{'hat größere Ausweisnummer'}$  nicht reflexiv!
  - $R = \text{'hat keine kleinere Ausweisnummer'}$  total



## Elementare Sortieralgorithmen

- Für alle Sortieralgorithmen
  - Datentyp:
    - item = record
    - key : integer
    - info: Grundtyp
  - end
- Eingabe: A: array[1...N] of item
- Ausgabe: A mit vertauschten Einträgen, so dass  $A[i] \leq A[i+1]$  für  $1 \leq i < N$  gilt.
- Laufzeitanalyse:
  - ◆ Zugriffsoperation  $c_z$
  - ◆ Vergleichsoperationen  $c_v$
  - ◆ Datenaustauschoperationen  $c_D$ 
    - ◆ ggf. ist  $c_D \gg c_v$ : Sortieren der Schlüssel, anschließend Datenaustausch



## Sortieren durch Auswahl

- Idee:
  - Iteriere von 1 bis N-1:
    - im i-ten Durchlauf: Schreibe das kleinste Element aus  $A[i...n]$  an Position  $A[i]$

SELECTION-SORT(A)

	index:	1	2	3	4	5	6	7	8
for i ← 1 to length[A]-1 do									
// suche kleinstes Element aus $A[i..N]$		A: 3	8	14	2	9	23	1	4
min ← i			↑ <sub>i</sub>						↑ <sub>min</sub>
for j ← i+1 to length[A] do									
if $A[j] < A[\text{min}]$ then min ← j		A: 1	8	14	2	9	23	3	4
// vertausche $A[i]$ und $A[\text{min}]$			↑ <sub>i</sub>		↑ <sub>min</sub>				
SWAP( $A[\text{min}]$ , $A[i]$ )		A: 1	2	14	8	9	23	3	4
SWAP(a,b)									
t ← a; a ← b; b ← t									

- Laufzeit:  $T(N) = \Theta(N^2)$
- nur  $O(N)$  Datenaustausche



## Sortieren durch Einfügen

### Idee:

- Iteriere von 2 bis N:  
im i-ten Durchlauf: Füge das i-te Element in die sortierte Folge  $A[1 \dots i-1]$  ein

INSERTION-SORT(A)  
**for**  $j \leftarrow 2$  **to**  $\text{length}[A]$  **do**  
 key  $\leftarrow A[j]$   
 // setze  $A[j]$  sortiert in  $A[1..j-1]$  ein  
 $i \leftarrow j-1$   
**while**  $i > 0$   $A[i] > \text{key}$  **do**  
 $A[i+1] \leftarrow A[i]$   
 $i \leftarrow i-1$   
 $A[i+1] \leftarrow \text{key}$

index: 1 2 3 4 5 6 7 8  
 A: 3 8 14 2 9 23 1 4  
 ↑ i                    ↑ j  
 A: 2 3 8 14 9 23 1 4  
 ↑ i                    ↑ j  
 A: 2 3 8 9 14 23 1 4  
 :  
 :  
 :

### Laufzeit (im Worst Case): $T(N) = \Theta(N^2)$



## Bubblesort

### Idee:

- Iteriere solange, bis keine Vertauschung mehr durchgeführt wird:  
im i-ten Durchlauf (for-Schleife): bringe Minimum von  $A[i \dots N]$  an Position  $A[i]$  durch sukzessives Vertauschen benachbarter Elemente

BUBBLESORT(A)  
**for**  $i \leftarrow 1$  **to**  $\text{length}[A]$   
 noswap  $\leftarrow \text{TRUE}$   
**for**  $j \leftarrow \text{length}[A]$  **downto**  $i+1$  **do**  
**if**  $A[j] < A[j-1]$  **then**  
 SWAP(  $A[j], A[j-1]$  )  
 noswap = FALSE  
**if** noswap **then break**;

index: 1 2 3 4 5 6 7 8  
 A: 3 8 14 2 9 23 1 4  
 A: 3 8 14 2 9 1 23 4  
 A: 1 3 8 14 2 9 23 4  
 i=2 A: 1 3 8 14 2 9 23 4  
 A: 3 8 2 9 14 3 4 23  
 :  
 :  
 :

### Laufzeit (im Worst Case): $T(N) = \Theta(N^2)$



## Mergesort

### Idee: Divide & Conquer

- Teile die Folge in zwei Teilfolgen  $L = A[1 \dots \lfloor N/2 \rfloor]$ ,  $R = A[\lfloor N/2 \rfloor + 1 \dots N]$
- Sortiere L und R
- Verschmelze sortierte Folgen L und R zu einer sortierten Folge

MERGE-SORT(A, l, r)  
**if**  $p < r$  **then**  
 $m \leftarrow \text{floor}((l+r) / 2)$   
 MERGE-SORT(A, l, m)  
 MERGE-SORT(A, m+1, r)  
 MERGE(A, l, m, r)

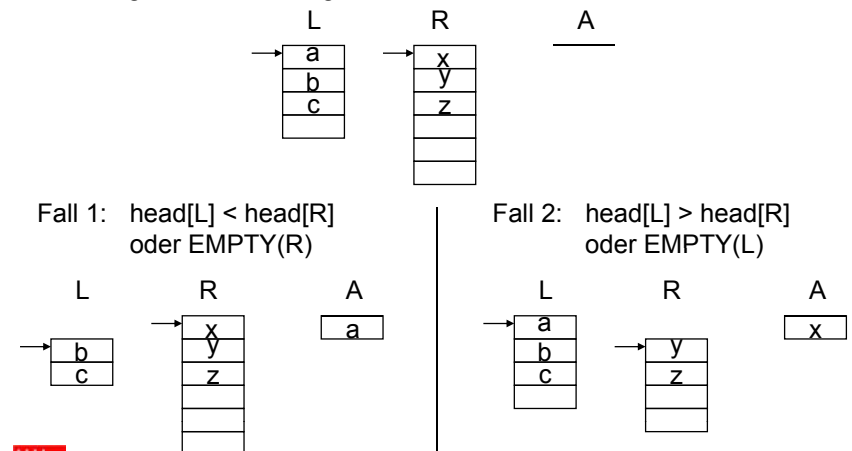
index: 1 2 3 4 | 5 6 7 8  
 Divide:  
 A: 3 8 14 2 | 9 23 1 4  
 Conquer:  
 A: 2 3 8 14 | 1 4 9 23  
 Merge:  
 A: 1 2 3 4 8 9 14 23



## 2-Wege Merge-Operation

### Merge-Operation (mit Listen)

- geg.: zwei aufsteigend sortierte Listen L und R
- ges.: eine aufsteigend sortierte Liste A aus L und R



## 2-Wege Merge-Operation

- Listenoperationen:
  - INIT: initialisiere leere Liste      EMPTY: wahr, falls Liste leer
  - head: erstes Element der Liste      TAIL: Liste ohne erstes Element
  - APPEND: einfügen am Ende der Liste
- MERGE-LISTS(L,R : list)
  - A ← INIT()
  - while not (EMPTY(L) and EMPTY(R)) do
    - if EMPTY(R) or head[L] ≤ head[R] then
 APPEND(A, head[L])
 L ← tail(L)
    - if EMPTY(L) or head[L] ≥ head[R] then
 APPEND(A, head[R])
 R ← TAIL(R)
  - return A

Laufzeit:  $O(|L|+|R|)$



## 2-Wege Merge-Operation mit Arrays

```

procedure merge (var a : sequence; l, m, r : integer);
{verschmilzt die beiden sortierten Teilfolgen a[l]...a[m]
und a[m+1]...a[r] und speichert sie in a[l]...a[r]}
var
  b : sequence; {Hilfsfeld zum Verschmelzen}
  h, i, j, k : integer;
begin
  i := l; {inspiziere noch a[i] bis a[m] der ersten Teilfolge}
  j := m + 1; {inspiziere noch a[j] bis a[r] der zweiten Teilfolge}
  k := l; {das nächste Element der Resultatfolge ist b[k]}
  while (i ≤ m) and (j ≤ r) do
    begin {beide Teilfolgen sind noch nicht erschöpft}
      if a[i].key ≤ a[j].key
      then {übernimm a[i] nach b[k]}
        begin
          b[k] := a[i];
          i := i + 1;
        end
      else {übernimm a[j] nach b[k]}
        begin
          b[k] := a[j];
          j := j + 1;
        end;
      k := k + 1;
    end;
  if i > m
  then {erste Teilfolge ist erschöpft; übernimm zweite}
    for h := j to r do b[k + h - j] := a[h];
  else {zweite Teilfolge ist erschöpft; übernimm erste}
    for h := i to m do b[k + h - i] := a[h];
  {speichere sortierte Folge von b zurück nach a}
  for h := l to r do a[h] := b[h];
end
    
```

- Liste L: a[l...m]    Liste R: a[m+1...r]
- Liste A: zunächst in b, wird abschließend kopiert nach a
- i: index von head[L]
- j: index von head[R]
- k: index von rear[A]
- ... APPEND(...)
- ... TAIL(...)
- andere Schleifenstruktur:
  - ◆ Teil 1: beide Listen nicht leer
  - ◆ Teil 2: Liste L leer
  - ◆ Teil 3: Liste R leer
- Kopieren von b nach a



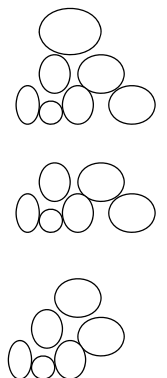
## Laufzeit Mergesort

- Mergesort
  1. Teile die Folge in zwei Teilfolgen  
 $L = A[1..\lceil n/2 \rceil]$ ,  $R = A[\lceil n/2 \rceil + 1..n]$        $O(1)$
  1. Sortiere L und R       $2 * T(N/2)$
  2. Verschmelze sortierte Folgen L und R zu einer sortierten Folge       $O(N)$
- Laufzeit:  $T(N) = 2 * T(N/2) + cN = O(N \log N)$
- Speicherbedarf:
  - Array A:  $c*N$
  - Zwischenspeicherung in zweitem Array:  $c*N$
  - doppelter Speicherbedarf!
- k-Wege Mergesort: Teile und Verschmelze k Teilfolgen  
 Laufzeit:  $T(N) = k * T(N/k) + c k N = O(N \log N)$   
 (k \* N, da in jeder Iteration das Minimum aus k Listenköpfen bestimmt werden muss)



## 3.2 Heaps und Heapsort

- Sortieren durch Auswahl (siehe 3.1)
  - das kleinste Element wird ausgewählt und an die bereits sortierte Liste angehängt
  - Wie gestaltet man den Auswahlsschritt?
- Neue Datenstruktur: Heap
  - 'Objekte liegen auf einer Halde, ein Objekt ist immer mindestens so groß wie die darunter liegenden Objekte' (Heap-Bedingung)
  - Operationen:
    - ◆ BUILD-MAX-HEAP: baue eine Halde aus einer Menge von Elementen
    - ◆ HEAP-EXTRACT-MAX: nehme das größte Element von der Halde
    - ◆ MAX-HEAPIFY: stelle die Heap-Bedingung nach Entnahme wieder her



## Der Heapsort-Algorithmus

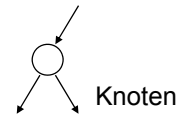
- **HEAPSORT-WITH-HEAP(A)**  
 // Heapsort mit HEAP-Datenstruktur  
 $H \leftarrow \text{INIT}()$   
**for**  $i \leftarrow 1$  **to**  $\text{length}[A]$  **do**  
    $\text{HEAP-INSERT}(H, A[i])$   
**for**  $i \leftarrow \text{length}[A]$  **downto**  $1$  **do**  
    $A[i] \leftarrow \text{HEAP-EXTRACT-MAX}(H)$
- füge alle Elemente in den Heap ein (Zeit  $T_{\text{build}}(N)$ )
- entnehme das größte Element (und stelle die Heap-Eigenschaft wieder her) (Zeit  $T_{\text{xmax}}(N)$ )
- **HEAPSORT(A)**  
 // Heap mit  $i$  Elementen steht im Array  $A[1..i]$ ,  
 //  $A[1]$  enthält das Maximum  
 $\text{BUILD-MAX-HEAP}(A)$   
 $\text{heap\_size} \leftarrow \text{length}[A]$   
**for**  $i \leftarrow \text{length}[A]$  **downto**  $2$  **do**  
    $\text{SWAP}(A[i], A[1])$   
    $\text{heap\_size} \leftarrow \text{heap\_size} - 1$   
    $\text{MAX-HEAPIFY}(A, 1)$
- Laufzeit:  
 ■  $T(N) = O(T_{\text{build}}(N) + N T_{\text{xmax}}(N))$



## Der Heap

- Der Heap basiert auf einer *binären Baumstruktur*:

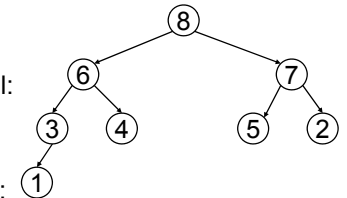
- ◆ jeder Knoten  $i$  hat bis zu zwei Nachfolger  $\text{LEFT}(i)$ ,  $\text{RIGHT}(i)$
- ◆ alle bis auf ein Knoten haben genau einen Vorgänger  $\text{PARENT}(i)$
- ◆ jeder Knoten speichert ein Element  $A[i]$



- **Heap-Eigenschaft:**

Max-Heap:  $\forall$  Knoten  $i$  außer der Wurzel:  
 $A[\text{PARENT}(i)] \geq A[i]$

Min-Heap:  $\forall$  Knoten  $i$  außer der Wurzel:  
 $A[\text{PARENT}(i)] \leq A[i]$



## Der Heap

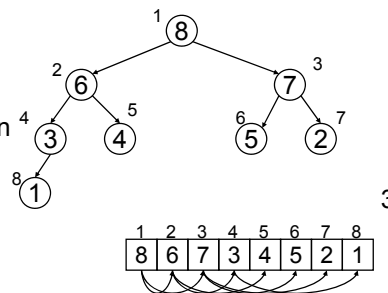
- Ein Heap kann durch ein Array repräsentiert werden:
  - $\text{PARENT}(i) = i \text{ div } 2$
  - $\text{LEFT}(i) = 2 * i$
  - $\text{RIGHT}(i) = 2 * i + 1$

- Begriffe: Sei  $A$  ein Heap

- **Höhe von  $A$ :** max Anzahl Knoten auf dem direktem Weg zu einem Blatt
- $\text{size}[A]$ : Anzahl Elemente in  $A$

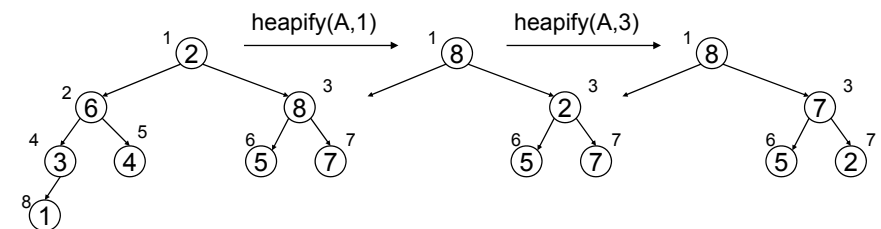
- Weitere Eigenschaften:

- Ein Heap der Größe  $N$  hat max. Höhe:  $\lfloor \log_2 N \rfloor$
- auf Ebene  $i$  liegen  $2^i$  Knoten



## Die MAX-HEAPIFY-Operation

- Heap-Eigenschaft (top-down): Sei  $A$  ein Heap, dann gilt
  - $A[i] \geq A[\text{LEFT}(i)]$  und  $A[i] \geq A[\text{RIGHT}(i)]$
- Heapify (Versickern):
  - Annahme:  $A$  erfüllt Heap-Eigenschaft bis auf in einem Teilbaum unter Position  $i$
  - $\text{MAX-HEAPIFY}(A, i)$ : repariert den Heap (stellt die Heap-Eigenschaft wieder her)



HEAPIFY: - bestimme direkten Nachfolger  $c$  mit max. Wert  $A[c]$   
 - vertausche  $A[i]$  mit  $A[c]$   
 -  $\text{HEAPIFY}(A, c)$



## Die MAX-HEAPIFY-Operation

### ■ MAX-HEAPIFY(A,i)

if LEFT(i) ≤ size[A] and A[LEFT(i)] > A[i] then

c ← LEFT(i)

else c ← i

if RIGHT(i) ≤ size[A] and A[RIGHT(i)] > A[c] then

c ← RIGHT(i)

if c ≠ i then

SWAP(A[i], A[c])

MAX-HEAPIFY(A, c)

c: Index des größten Elements aus der Menge i, left(i), right(i) innerhalb von A

c = i: Heap-Eigenschaft war nicht verletzt

c ≠ i: vertausche Elemente c und i

### ■ Laufzeit:

$$T_{\text{heapify}}(n) = T_{\text{heapify}}(2n/3) + \Theta(1)$$

Master-Theorem, Fall 2:

$$T_{\text{heapify}}(n) = O(\log(N))$$



## Laufzeit der MAX-HEAPIFY-Operation

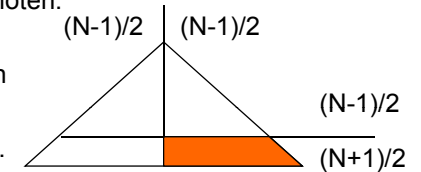
### ■ Rekurrenzgleichung:

Im Worst-Case gilt:  $T_{\text{heapify}}(n) = T_{\text{heapify}}(2n/3) + \Theta(1)$

### ■ Beweis:

■ Betrachte zunächst einen vollständigen binären Baum B der Höhe  $k = \lfloor \log_2 n \rfloor + 1$  und  $N = 2^k - 1$  Knoten:

B hat  $(N-1)/2$  Blätter und je  $(N-1)/2$  Knoten in den Teilbäumen unter der Wurzel.



■ Ein Heap der Höhe k ist ein vollst. binärer Baum mit fehlenden Blättern.

Worst-Case:  $(N+1)/4$  Blätter fehlen:

Größe des linken Teilbaums:  $L = (N-1)/2$  Knoten

mit  $n = (3N-1)/4 \Leftrightarrow N = (4n-1)/3 \Rightarrow L = (4n-4)/6 \leq 2n/3$

Worst-Case: Verzweigung in den linken Teilbaum resultiert in die oben genannte Rekurrenzgleichung.



## Die HEAP-EXTRACT-MAX-Operation

### ■ HEAP-EXTRACT-MAX:

Entnahme des maximalen Element aus dem Heap

### ■ HEAP-EXTRACT-MAX(A)

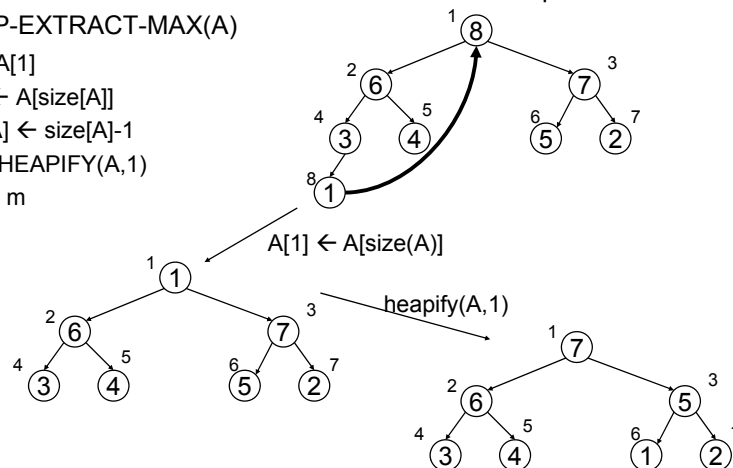
m ← A[1]

A[1] ← A[size[A]]

size[A] ← size[A]-1

MAX-HEAPIFY(A, 1)

return m



## Die BUILD-MAX-HEAP-Operation

### ■ BUILD-MAX-HEAP:

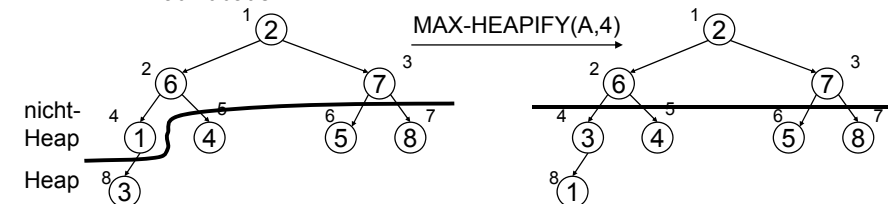
■ Eingabe: unsortiertes Array A

■ Ausgabe: Heap (als Array in A)

■ Idee:

◆ Elemente  $\lfloor \text{size}[A]/2 \rfloor + 1 - \text{size}[A]$  (die Blätter) sind ein-elementige Heaps

◆ Verwende MAX-HEAPIFY um den Heap von unten nach oben aufzubauen



## Die BUILD-MAX-HEAP-Operation

- BUILD-MAX-HEAP(A)
  - size[A]  $\leftarrow$  length[A]
  - for  $i \leftarrow \lfloor \text{size}[A]/2 \rfloor$  downto 1 do
    - MAX-HEAPIFY(A,i) N/2 Durchläufe  
 $O(\log N)$
- Korrektheit von BUILD-MAX-HEAP:
  - Schleifeninvariante:  $\forall k \in \{i+1, \dots, n = \text{length}[A]\}$ : A[k] ist die Wurzel eines Max-Heap.
  - Initialisierung:
    - ◆ Knoten  $A[\lfloor n/2 \rfloor + 1], A[\lfloor n/2 \rfloor + 2], \dots, n$  sind Blätter und erfüllen somit die Max-Heap-Eigenschaft.
  - Fortsetzung: Für Knoten i gilt:
    - ◆ LEFT(i) und RIGHT[i] sind Max-Heaps wg. der Schleifeninvariante
    - ◆ MAX-HEAPIFY(A,i) stellt somit die Max-Heap-Eigenschaft her.
  - Terminierung:
    - ◆ Schleife terminiert mit  $i=0$ , damit in A[1] die Wurzel eines Max-Heap.



## Die BUILD-MAX-HEAP-Operation

- Laufzeit von BUILD-MAX-HEAP(A)
  - Sei  $n = \text{length}[A]$ , offensichtlich gilt  $T_{\text{build}}(n) = O(N \log N)$ .
- Gilt auch  $T_{\text{build}}(n) = \Theta(N \log N)$ ?
  - Laufzeit ist proportional zur akkumulierten Laufzeit der MAX-HEAPIFY-Operationen
  - MAX-HEAPIFY für einen Knoten mit Höhe h:  $O(h)$
  - Höhe variiert von  $h=0$  bis  $\lfloor \log n \rfloor$
  - Anzahl Knoten mit Höhe h:  $\lceil n / 2^{h+1} \rceil$   
(Anzahl Knoten halbiert sich von Ebene zu Ebene)

$$T(N) = c \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h = O\left(N \sum_{h=0}^{\log n} \frac{h}{2^h}\right) = O(N)$$

$$\text{mit } \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1 - 1/2)^2} = 2 \quad \left( \text{Ableitung der geom. Reihe, siehe Cormen A.8} \right)$$



## Prioritätswarteschlangen (Priority Queues)

- Priority Queue:
  - Ziel: Warteschlange mit Prioritäten
  - Operationen:
    - ◆ HEAP-INSERT(A,x): füge ein Element x in die Warteschlange A ein
    - ◆ HEAP-EXTRACT-MAX(A): entnehme das Element mit max. Priorität
    - ◆ HEAP-INCREASE-KEY(A, i, k): erhöhe den Wert von Schlängenelement Nummer i auf k,  $k > A[i]$
    - ◆ HEAP-DECREASE-KEY(A, i, k): verringere den Wert von Schlängenelement Nummer i auf k,  $k < A[i]$
  - Realisierung: durch einen Heap
    - ◆ HEAP-EXTRACT-MAX-Operation: wie zuvor, Laufzeit  $O(\log N)$
    - ◆ HEAP-DECREASE-KEY-Operation: wie MAX-HEAPIFY, Laufzeit  $O(\log N)$

Achtung:

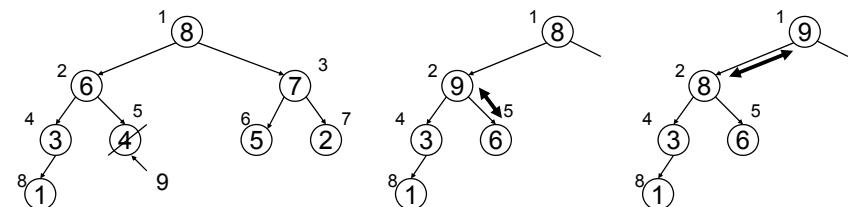
• Typo im Cormen S. 135: INCREASE-KEY(S,i,k), nicht (S,x,k)!

• Im Gegensatz zum Buch ist hier DECREASE-KEY() in einem Max-Heap gemeint.



## Die HEAP-INCREASE-KEY-Operation

- HEAP-INCREASE-KEY(A, i, k )
  - if  $k < A[i]$  then error „Increase-Error“
  - else
    - $A[i] \leftarrow k$
    - while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$  do
      - SWAP( $A[i]$ ,  $A[\text{PARENT}(i)]$ )
      - $i \leftarrow \text{parent}(i)$
- HEAP-INCREASE-KEY funktioniert wie heapify in umgekehrter Richtung:



## Die INSERT-Operation

- **HEAP-INSERT(A,x)**  
size[A]  $\leftarrow$  size[A]+1  
A[size[A]]  $\leftarrow -\infty$   
HEAP-INCREASE-KEY(A, size[A], x)
- Mit einem Heap kann eine Priority-Queue mit Laufzeit  $O(\log N)$  für die Operationen insert, delete, increase, decrease, extract-max realisiert werden.
- **Anmerkung:**  
Zum Auffinden eines Wertes x benötigt man eine zusätzliche Datenstruktur, z.B.
  - **HEAP-INCREASE-KEY2( A, x, k ):**  
Problem: Finde i mit  $A[i] = x$



## 3.3 Quicksort

- **Idee: Divide & Conquer**
  - Teile die Folge in zwei Teilfolgen  $A[1...q-1]$  und  $A[q+1...n]$  mit:  
 $A[i] \leq A[q] \quad \forall i < q$  und  $A[i] \geq A[q] \quad \forall i > q$  (\*)
  - Sortiere die Teilfolgen  $A[1...q-1]$  und  $A[q+1...n]$
  - (Kombination der Teilfolgen nicht nötig aufgrund von (\*))
- **QUICKSORT( A, l, r )**  
if  $l < r$  then  
     $q \leftarrow \text{PARTITION}(A, l, r)$  // Bestimmt q und stellt Bedingung (\*) her  
    QUICKSORT(A, l, q-1)  
    QUICKSORT(A, q+1, r)
- initialer Aufruf: QUICKSORT(A, 1, length[A])



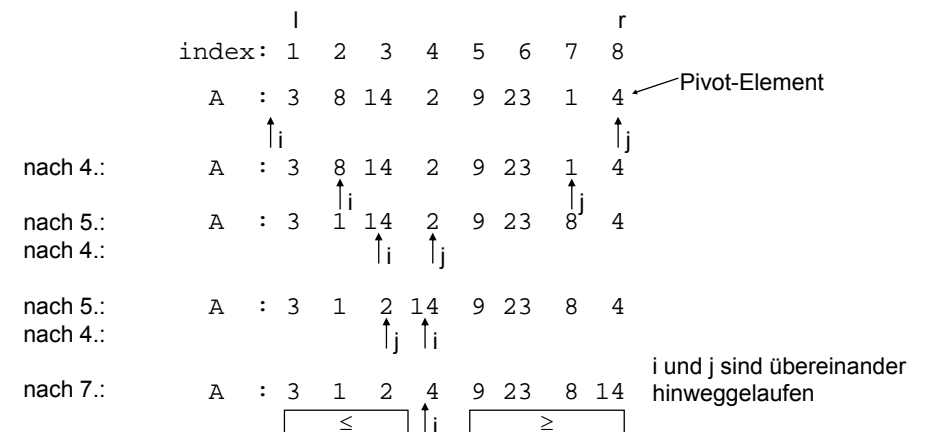
## Partition-Funktion

- **Partition-Funktion (Hoare-Zerlegung)**
    - geg: ein Array  $A[l...r]$
    - ges: q mit  $l \leq q \leq r$ , umsortiertes Array A mit  
 $A[i] \leq A[q] \quad \forall i < q$  und  $A[i] \geq A[q] \quad \forall i > q$  //  $A[q]$ : *Pivot-Element*
- PARTITION\_HOARE(A, l, r)**  
// verwende  $A[r]$  als Pivot-Element  
1  $i \leftarrow l-1; j \leftarrow r;$   
2 **while**  $i < j$  **do**  
3   **repeat**  $i \leftarrow i+1$  **until**  $A[i] \geq A[r]$   
4   **repeat**  $j \leftarrow j-1$  **until**  $A[j] \leq A[r]$   
5   **if**  $i < j$  **then** SWAP( $A[i], A[j]$ )  
6 SWAP( $A[i], A[r]$ ) // schiebt das Pivot-Element in die Mitte  
7 **return** i



## Partition-Funktion

- **Beispiel: Partition-Funktion nach Hoare**





## Partition-Funktion

- Alternative Prozedur mit nur einer Schleife:

### PARTITION(A, l, r)

// verwende A[r] als Pivot-Element

$i \leftarrow l-1$

for  $j \leftarrow l$  to  $r-1$  do

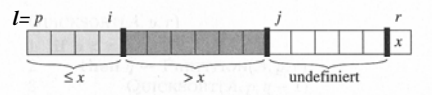
if  $A[j] \leq A[r]$  then

$i \leftarrow i+1$

SWAP( A[i], A[j] )

SWAP( A[i+1], A[r] )

return i+1



- Funktionsweise:

PARTITION zerlegt A in vier (ggf. leere) Bereiche:

- für  $l \leq k \leq i$  gilt:  $A[k] \leq A[r]$
- für  $i < k < j$  gilt:  $A[k] > A[r]$
- für  $j \leq k < r$  gilt:  $A[k] ? A[r]$
- für  $k = r$  gilt:  $A[k] = A[r]$



## Partition-Funktion

### PARTITION(A, l, r)

// verwende A[r] als Pivot-Element

$i \leftarrow l-1$

for  $j \leftarrow l$  to  $r-1$  do

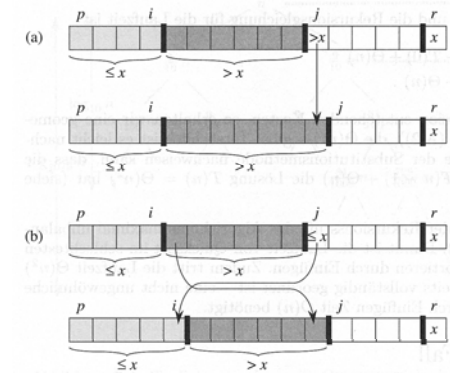
if  $A[j] \leq A[r]$  then

$i \leftarrow i+1$

SWAP( A[i], A[j] )

SWAP( A[i+1], A[r] )

return i+1



- Korrektheit: (informell)

- Fall 1:  $A[j] > A[r]$  : Verlängere Bereich 2 um ein Feld
- Fall 2:  $A[j] \leq A[r]$  : Vertausche A[i] und A[j], vergrößere Bereich 1 um ein Feld, verlängere Bereich 2 um ein Feld



## Analyse von Quicksort

- Rekurrenzgleichung für Quicksort:

- Sei q die Position des Pivot-Elements, dann gilt

$$T(N) = T(q-1) + T(N-q) + cN$$

- Annahme: Pivot-Element ist immer das Maximum der sortierten Teilfolge, d.h.  $q = N$ :

$$T(N) = T(N-1) + T(0) + c \cdot N = c \cdot (N + N-1 + N-2 + \dots + 1) = \Theta(N^2)$$

- Annahme: Pivot-Element liegt in der Mitte der sortierten Teilfolge, d.h.  $q = \lceil N/2 \rceil$

$$T(N) = T(\lceil N/2 \rceil - 1) + T(N - \lceil N/2 \rceil) + cN \approx 2T(N/2) + cN = \Theta(N \log N)$$

- Annahme: Pivot-Element liegt bei  $q = aN$  für ein konstantes  $0,5 < a < 1$

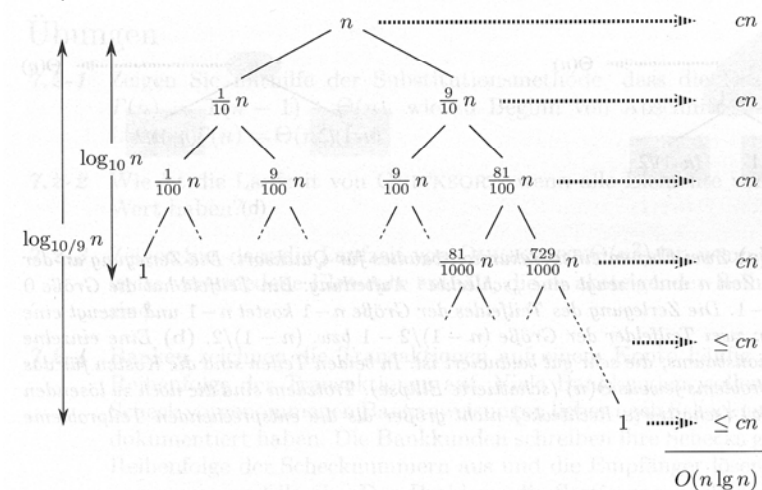
$$T(N) = T(\lceil aN \rceil) + T(\lceil (1-a)N \rceil) + cN$$

- Rekursionsbaum mit maximal  $-\log_a N = \log_{1/a} N$  Ebenen
- Laufzeit pro Ebene des Rekursionsbaums:  $cN$
- $T(N) = O(N \log N)$



## Analyse von Quicksort

- Beispiel für  $a = 0,9$ :





## Der randomisierte Quicksort

- Randomisierter Algorithmus: Algorithmus, der unter Verwendung von Zufallszahlen operiert.
  - ACHTUNG: Randomisierte Algorithmen sind (i.d.R.) deterministisch (d.h. sie liefern immer das gleiche Ergebnis)!
- RAND-PARTITION(A, l, r)
 

```
i ← RANDOM(l, r) // wählt eine ganzzahlige Zufallszahl aus [l,r]
SWAP( A[i], A[r] )
return PARTITION(A, l, r)
```
- RAND-QUICKSORT(A, l, r)
 

```
if l < r then
    q ← RAND-PARTITION(A, l, r)
    RAND-QUICKSORT(A, l, q-1)
    RAND-QUICKSORT(A, q+1, r)
```
- Erwartete Laufzeit von RAND-QUICKSORT ist unabhängig von der Vorsortierung



## Analyse des randomisierten Quicksort

- **Theorem:** Angenommen, alle Werte sind voneinander verschieden, dann ist die erwartete Laufzeit von Quicksort  $O(N \log N)$ .
- **Beweisalternative 1:** (Mittelung über die Rekurrenzgleichung)
 
$$T_{\text{exp}}(N) = \left[ \frac{1}{N} \sum_{q=1}^N T(q-1) + T(N-q) \right] + cN \leq \left[ \frac{2}{N} \sum_{q=1}^{\lceil N/2 \rceil} T(q) \right] + cN \stackrel{?}{=} O(N \log N)$$
- **Beweisalternative 2:** (Erwartete Anzahl Vergleiche)
  - Laufzeit von Quicksort wird durch PARTITION dominiert.
  - Laufzeit von PARTITION ist proportional zur Anzahl Vergleiche.
  - Sei X die Anzahl Vergleiche, dann ist  $T(N, X) = O(N + X)$ .
- Wie hoch ist die erwartete Anzahl Vergleiche  $E[X]$  ?
  - Sei  $X_{ij}$  das Ereignis, dass  $A[i]$  mit  $A[j]$  verglichen wird.
  - Wenn  $X_{ij}=1$ , ist entweder  $A[i]$  oder  $A[j]$  Pivotelement, jeder Paarvergleich findet nur ein mal statt:

$$E[X] = E \left[ \sum_{i=1}^{N-1} \sum_{j=i+1}^N X_{ij} \right] = \sum_{i=1}^{N-1} \sum_{j=i+1}^N E[X_{ij}] = \sum_{i=1}^{N-1} \sum_{j=i+1}^N \Pr\{X_{ij} = 1\}$$



## Analyse des randomisierten Quicksort

- Wie hoch ist die Wahrscheinlichkeit, dass  $A[i]$  mit  $A[j]$  verglichen wird, d.h.  $\Pr\{X_{ij}=1\}$ ?
  - Sei x der Pivotwert, dann gilt für alle  $A[i] < x < A[j]$ :  $\Pr\{X_{ij}=1\} = 0$
  - Für jedes Teil-Array  $A[i...j]$  gilt:  $A[i]$  wird mit  $A[j]$  verglichen, genau dann wenn entweder  $A[i]$  oder  $A[j]$  als Pivotelement gewählt werden.
  - $A[i...j]$  enthält  $j-i+1$  Elemente, die Wahrscheinlichkeit zur Auswahl des Pivotelements ist gleichverteilt (RAND-PARTITION):

$$\Pr\{X_{ij}=1\} = \Pr\{A[i] \text{ oder } A[j] \text{ ist erstes Pivotelement aus } A[i...j]\}$$

$$\Pr\{X_{ij}=1\} = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

Typo im Cormen:  
S. 156:  $k=j-i$

$$E[X] = \sum_{i=1}^{N-1} \sum_{j=i+1}^N \Pr\{X_{ij}=1\} = \sum_{i=1}^{N-1} \sum_{j=i+1}^N \frac{2}{j-i+1} \quad \text{mit } k = j-i$$

$$= \sum_{i=1}^{N-1} \sum_{k=1}^{N-i} \frac{2}{k+1} < \sum_{i=1}^{N-1} \sum_{k=1}^N \frac{2}{k} = \sum_{i=1}^{N-1} 2 \sum_{k=1}^N \frac{1}{k} \leq \sum_{i=1}^{N-1} 2(\log N + 1) \quad \text{mit A.10 (harmonische Reihe)}$$

$$= 2(N-1)(\log N + 1) = O(N \log N)$$



## Abschließende Kommentare zu Merge-, Heap- und Quicksort

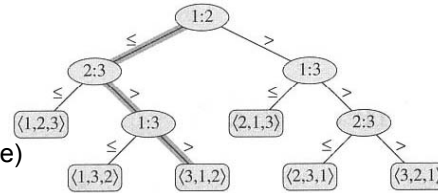
- **Sortieren durch Einfügen** hat eine Worst-Case Laufzeit von  $O(N^2)$ , ist allerdings vorteilhaft, wenn die Eingabe nahezu sortiert ist.
- **Mergesort** hat eine Worst-Case Laufzeit von  $O(N \log N)$ , ist aber kein in-place Sortierverfahren.
- **Mergesort** eignet sich gut für große Datenbestände auf Sekundärspeichern.
- **Heapsort** hat eine Worst-Case Laufzeit von  $O(N \log N)$  und sortiert in-place.
- Die Worst-Case Laufzeit von **Quicksort** ist  $O(N^2)$ , die erwartete Laufzeit des randomisierten Verfahrens  $O(N \log N)$ . Um eine Worst-Case Laufzeit von  $O(N \log N)$  zu erhalten, muß der Median der Eingabefolge in  $O(N)$  Zeit bestimmt werden.
- **Quicksort** zeigt in der Praxis ein sehr gutes Laufzeitverhalten (niedrige Konstanten).
- **Quicksort** kann in der Praxis weiter beschleunigt werden, in dem ein rekursiver Aufruf durch eine Iteration ersetzt wird (Endrekursion).

**Sind  $O(N \log N)$ -Sortierverfahren laufzeitoptimal?**



### 3.4 Eine untere Schranke für das Sortierproblem

- Annahme: alle zu sortierenden Elemente sind paarweise verschieden.
- Vergleichende Sortierverfahren:
  - basieren ausschließlich auf Paarvergleiche zwischen den Elementen der Eingabe.
  - Operationen auf Elemente beschränkt auf:  $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$
  - Operationen können auf eine reduziert werden:  $\leq$
- Das Entscheidungsbaummodell:
  - vollständiger binärer Baum:
  - innere Knoten: Vergleich zwischen zwei Elementen  $A[i]$  und  $A[j]$
  - Blätter: Permutation (Rangfolge) der Elemente



### Eine untere Schranke für das Sortierproblem

- **Theorem:** Jedes vergleichende Sortierverfahren benötigt bei  $n$  zu sortierenden Datenelementen im Worst-Case  $\Omega(n \log n)$  Vergleichsoperationen.
  - Jedes Sortierverfahren führt eine Serie von Vergleichsoperationen aus.
  - diese entspricht einem Pfad von Wurzel zu Blatt in einem entsprechenden Entscheidungsbaum.
  - Jedes Sortierverfahren muss jede mögliche Permutation erzeugen können.
  - Der Entscheidungsbaum hat somit  $l = n!$  Blätter.
  - Die Höhe des Entscheidungsbaums ist eine untere Schranke für die Anzahl der Vergleiche:

$$h \geq \log_2 l = \log_2 n! = \Omega(N \log N)$$

Gleichung 3.18  
Cormen S. 55



### 3.5 Counting-, Radix- und Bucketsort

- Gibt es Sortierverfahren, die nicht ‚vergleichend‘ sind?
  - Im allgemeinen nicht, da wir für die zu sortierenden Objekte lediglich die Ordnungsrelation kennen.
- **Annahme 1:** Die zu sortierenden Schlüssel sind ganzzahlig, paarweise verschieden und aus  $\{1, \dots, N\}$ 
  - TRIVIAL\_SORT( $A$ )
 

```
for i ← 1 to N do B[A[i]] ← A[i]
return B
```
- **Annahme 2:** Die zu sortierenden Schlüssel sind ganzzahlig aus  $\{0, \dots, k\}$ ,  $k = O(N)$ 
  - Idee: Zähle die Anzahl der Elemente mit kleinerem Schlüssel.
  - → Countingsort



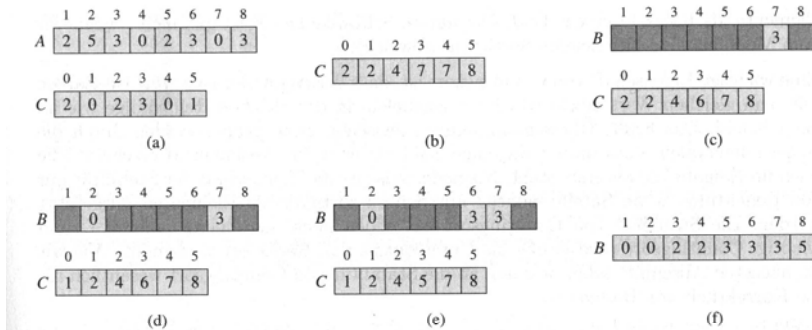
### Countingsort

- Feld A: Eingabe, Feld B: sortierte Ausgabe
- Feld  $C[0 \dots k]$ : Zähler-Array, ändert die Bedeutung während des Alg.
- COUNTING-SORT( $A, B, k$ )
 

```
1 for i ← 0 to k do C[i] ← 0
2 for j ← 1 to length[A] do C[A[j]] ← C[A[j]] + 1
   // C[i] enthält die Anzahl der Elemente, deren Schlüssel gleich i sind
3 for i ← 1 to k do C[i] ← C[i] + C[i-1]
   // C[i] enthält die Anzahl der Elemente, deren Schlüssel ≤ i sind
4 for j ← length[A] downto 1 do
5   B[C[A[j]]] ← A[j]
6   C[A[j]] ← C[A[j]] - 1
7 return B
```
- Laufzeit von Countingsort:  $T(N, k) = O(N + k)$   
Für  $k = O(N)$  sortiert Countingsort in Linearzeit.



## Countingsort: Ein Beispiel



- Beispiel für  $N=8$ ,  $k=5$
- a) nach Zeile 2      b) nach Zeile 3      c) nach 1. Iteration Zeile 4
- d) nach 2. Iteration    e) nach 3. Iteration    f) nach Zeile 7



## Radixsort (Sortieren durch Fächerverteilen)

- **Definition:** Ein Sortierverfahren heißt *stabil*, genau dann wenn für alle  $i < j$ ,  $i, j \in \{1, \dots, N\}$  mit  $A[i] = A[j]$  gilt  $\pi(i) < \pi(j)$
- **Satz:** Countingsort ist ein stabiles Sortierverfahren.

- **Annahme 3:** Sortierung erfolgt nach einem  $m$ -adischen Schlüssel  $A[i] = (k_1, k_2, \dots, k_d)$ ;  $\forall 1 \leq j \leq d$ :  $k_j$  ist ganzzahlig und  $0 \leq k_j \leq m-1$

- Idee:

- Verteilphase: Sortiere die Daten in  $m$  Fächer nach dem ersten (niederwertigsten!) Schlüssel
- Sammelphase: Sammle die Daten wieder zu einer Liste
- wiederhole mit dem nächsten Schlüssel
- wichtig: im  $i$ -ten Durchlauf bleibt die relative Ordnung des  $i-1$ -ten Durchlaufs erhalten, d.h. die Verteilphase setzt ein stabiles Sortierverfahren als Subroutine voraus



## Radixsort (Sortieren durch Fächerverteilen)

- Beispiel:  $N=7$ ,  $m=10$ ,  $d=3$

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- **RADIX-SORT**( $A$ ,  $d$ ,  $m$ )  
 for  $j \leftarrow 1$  to  $d$  do  
   //  $\text{key}[j]$ : Attribut, beschreibt den  $j$ -ten Schlüssel von  $A$   
   COUNTING-SORT(  $\text{key}[j][A]$ ,  $B$ ,  $m-1$  )  
    $A \leftarrow B$



## Radixsort mit Warteschlangen

- **RADIXSORT**( $A$ ,  $d$ ,  $m$ )  
 //  $L$ : array  $[0 \dots m-1]$  von Warteschlangen  
 for  $j \leftarrow 0$  to  $m-1$  do  $L[j] \leftarrow \text{init}()$   
 for  $t \leftarrow 1$  to  $d$  do                   // sortiere nach Schlüssel  $t$   
   // Verteilphase  
   for  $i \leftarrow 1$  to  $N$  do  $\text{ENQUEUE}(L[\text{key}[t][A[i]]], A[i])$   
   // Sammelphase  
    $i \leftarrow 1$   
   for  $j \leftarrow 0$  to  $m-1$  do  
   while not empty( $L[j]$ ) do  
    $A[i] \leftarrow \text{DEQUEUE}(L[j])$   
    $i \leftarrow i+1$



## Radixsort: Laufzeit- und Speicherbedarf

- Parameter
  - m: Anzahl Schlüsselwerte
  - d: Anzahl Schlüsselemente
  - N: Anzahl der Datensätze
- Laufzeit  $T(N,d,m) = O(d(N + m))$ 
  - Fall 1:  $d = O(1)$ ,  $m = O(N)$ :  $O(N)$
  - Fall 2: Alle Schlüssel verschieden:  $d \geq \log_m N$ :  $O(N \log N)$
- Speicherbedarf:
  - Implementierung mit Arrays:  $S(m,N) = O(mN)$
  - Implementierung mit Schlangen:  $S(m,N) = O(m + N)$

Radixsort sortiert somit nicht *in-place*.

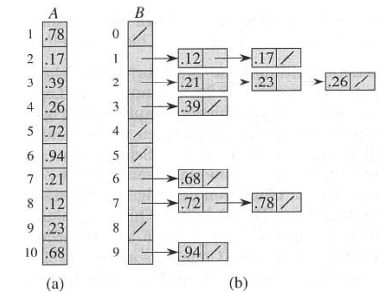


## Bucketsort

- **Annahme 4:** Für alle Element  $A[i]$  gilt:  $A[i] \in \mathbb{R}$ ,  $0 \leq A[i] < 1$ . Die Elemente sind in  $[0,1)$  gleichverteilt.
- Idee:
  - Teile den Wertebereich in  $N$  gleich große Teilintervalle  $T[i] = [i/N, (i+1)/N)$ , für  $0 \leq i < N$
  - Erzeuge  $N$  Fächer  $B[0, \dots, N-1]$  mit  $N$  linearen Listen
  - Sortiere durch Fächerverteilung und verbinde die Listen
- BUCKET-SORT(A)
 

```

1 for i ← 1 to length[A] do
2   LIST-INSERT( B[⌊length[A] A[i]⌋], A[i] )
3 i ← 1
4 for j ← 0 to length[A]-1 do
5   INSERTION-SORT(B[j])
6   while not empty( B[j] ) do
7     A[i] ← LIST-HEAD[B[j]];
8     B[j] ← LIST-TAIL[B[j]]; i ← i+1
            
```



## Bucketsort: Laufzeitanalyse

- Bucketsort ohne Zeile 5 benötigt  $O(N)$  Zeit. Sei  $n_i = \text{length}[B[i]]$ : INSERTION-SORT für Liste  $B[i]$  benötigt  $O(n_i^2)$  Zeit.
- Worst-Case:  $n_0 = N$ ,  $n_i = 0$  für alle  $i > 0$ :  $T_{\text{Bucket}}(N) = O(N^2)$   
Dieses Szenario widerspricht allerdings der Gleichverteilungsannahme.
- Worst-Case erwartete Laufzeit:

$$T_{\text{bucket}}(N) = cN + \sum_{i=0}^{N-1} cn_i^2$$

$$E[T_{\text{bucket}}(N)] = E\left[cN + \sum_{i=0}^{N-1} cn_i^2\right] = cN + \sum_{i=0}^{N-1} cE[n_i^2]$$

- Angenommen, es gilt:  $E[n_i^2] = 2 - 1/N$

$$\text{dann folgt: } E[T_{\text{bucket}}(N)] = cN + \sum_{i=0}^{N-1} c(2 - 1/N) = cN(3 - 1/N) = O(N)$$



## Bucketsort: Laufzeitanalyse

- Bleibt zu zeigen:  $E[n_i^2] = 2 - 1/N$   
Sei  $X_{ij}$  das Ereignis, dass  $A[j]$  in Bucket  $i$  einsortiert wird. Dann gilt:

$$E[n_i^2] = E\left[\left(\sum_{j=1}^N X_{ij}\right)^2\right] = \sum_{j=1}^N E[X_{ij}^2] + \sum_{j=1}^N \sum_{\substack{k=1 \\ k \neq j}}^N E[X_{ij} X_{ik}]$$

(Ausmultiplizieren des Quadrats zur Trennung stochastisch abhängigen von unabhängigen Termen)

Ereignis  $X_{ij}$  tritt mit Wahrscheinlichkeit  $1/N$  auf (Gleichverteilung!):  $E[X_{ij}^2] = 1 \cdot \frac{1}{N} + 0 \cdot \left(1 - \frac{1}{N}\right) = \frac{1}{N}$

Ereignisse  $X_{ij}$  und  $X_{ik}$  sind unabhängig:  $E[X_{ij} X_{ik}] = E[X_{ij}] E[X_{ik}] = \frac{1}{N^2}$

Insgesamt folgt:  $E[n_i^2] = \sum_{j=1}^N \frac{1}{N} + \sum_{j=1}^N \sum_{\substack{k=1 \\ k \neq j}}^N \frac{1}{N^2} = N \cdot \frac{1}{N} + N(N-1) \cdot \frac{1}{N^2} = 2 - \frac{1}{N}$



## Bucketsort: Laufzeitanalyse

- Alternative Herleitung für:  $E[n_i^2] = 2 - 1/N$   
Bestimmung über die Varianz:  $E[n_i^2] = V[n_i] + E^2[n_i]$   
(dadurch entfällt die explizite Berechnung des Funktionsquadrats)  
Im folgenden laufen alle Summen ohne expl. Indizes von  $j=1, \dots, N$ .  
Es gilt:  
$$E[n_i] = E\left[\sum_{j=1}^N X_{ij}\right] = \sum_{j=1}^N E[X_{ij}] = \sum_{j=1}^N \frac{1}{N} = 1$$
$$V[X_{ij}] = \sum_{z=0}^1 \Pr\{X_{ij} = z\} (z - E[X_{ij}])^2$$
$$= \underbrace{\left(1 - \frac{1}{N}\right) \left(0 - \frac{1}{N}\right)^2}_{z=0} + \underbrace{\left(\frac{1}{N}\right) \left(1 - \frac{1}{N}\right)^2}_{z=1} = \left(1 - \frac{1}{N}\right) \left(\frac{1}{N}\right)^2 + \frac{1}{N} - \left(\frac{1}{N}\right)^2 = \frac{1}{N} - \frac{1}{N^2}$$
$$V[n_i] = V\left[\sum_{j=1}^N X_{ij}\right] = \sum_{j=1}^N V[X_{ij}] = \sum_{j=1}^N \left(\frac{1}{N} - \frac{1}{N^2}\right) = 1 - \frac{1}{N}$$
$$E[n_i^2] = V[n_i] + E^2[n_i] = 1 - \frac{1}{N} + 1^2 = 2 - \frac{1}{N}$$

Achtung: Die Umformung setzt stochastische Unabhängigkeit der  $X_{ij}$  voraus.



## Abschließende Kommentare zu Sortieren in Linearzeit

- Wenn wir nur die Vergleichsrelation für die Sortierung von Objekten zur Verfügung haben, gilt die **untere Laufzeitschranke**  $\Omega(N \log N)$ .
- Wenn arithmetische Operationen auf Schlüsseln möglich sind, kann diese Schranke unter Umständen durchbrochen werden:
  - Annahme 1: Die zu sortierenden Schlüssel sind ganzzahlig, paarweise verschieden und aus  $\{1, \dots, N\}$  → **Trivalsort**
  - Annahme 2: Die zu sortierenden Schlüssel sind ganzzahlig aus  $\{0, \dots, k\}$ ,  $k = O(N)$  → **Countingsort**
  - Annahme 3: Sortierung erfolgt nach einem  $m$ -adischen Schlüssel  $A[i] = (k_1, k_2, \dots, k_d)$ ;  $\forall 1 \leq j \leq d$ :  $k_j$  ist ganzzahlig und  $0 \leq k_j \leq m-1$  → **Radixsort**
  - Annahme 4: Für alle Element  $A[i]$  gilt:  $A[i] \in \mathbb{R}$ ,  $0 \leq A[i] < 1$ . Die Elemente sind in  $[0, 1)$  gleichverteilt. → **Bucketsort**
  - **Bucketsort** kann auch verwendet werden, wenn wir die Verteilung der Elemente kennen und eine Funktion entwickeln können, die die Elemente auf  $[0, \dots, N-1]$  gleichverteilt.



## 3.6 Algorithmen für Auswahlprobleme

- **Suchproblem:**
  - Eingabe eine sortierte Folge  $A$  mit  $N$  Schlüsseln  $A[i]$  und einen Suchschlüssel  $q$
  - Ausgabe: Index eines Elements  $k$  mit  $A[k] = q$  oder  $-1$ , falls  $A[k] \neq q$  für alle  $k$
- **Auswahlproblem:**
  - Eingabe: Eine Menge  $A$  aus  $N$  (paarweise verschiedenen) Zahlen und einen Suchindex  $i$  mit  $1 \leq i \leq N$
  - Ausgabe: Das Element  $x \in A$  mit  $A[k] < x$  für  $i-1$  Elemente  $A[k]$



## Suchen in sortierten Folgen

- Algorithmus 1: Lineare Suche
  - LINEAR-SEARCH( $A, q$ )
  - for**  $i \leftarrow 1$  **to**  $\text{length}[A]$  **do** **if**  $A[i] = q$  **then return**  $i$
  - return**  $-1$
  - Laufzeit:  $O(k)$ , falls  $q$  in der Folge vorkommt,  $O(N)$  sonst
- Algorithmus 2: Binäre Suche (Aufruf mit BINARY-SEARCH( $A, 1, \text{length}[A], q$ ))
  - BINARY-SEARCH( $A, l, r, q$ )
  - if**  $l > r$  **then return**  $-1$
  - else**
  - $m \leftarrow \lfloor (l+r)/2 \rfloor$
  - if**  $A[m] > q$  **then** BINARY-SEARCH( $A, l, m-1, q$ )
  - else if**  $A[m] < q$  **then** BINARY-SEARCH( $A, m+1, r, q$ )
  - else return**  $m$
  - Laufzeit:  $T(N) = T(N/2) + c = O(\log N)$



## Suchen in sortierten Folgen

### ■ Algorithmus 3: Exponentielle Suche

- Idee: Verdopple den rechten Rand bis  $A[r] > q$  gilt, führe binäre Suche im Intervall  $[r/2, \dots, r]$  durch

### ■ EXPONENTIAL-SEARCH(A, q)

```

1  $r \leftarrow 1$ 
2 while  $A[r] < q$  and  $r < N$  do  $r \leftarrow \min(2r, N)$ 
3 return BINARY-SEARCH(A,  $r/2+1, r, q$ )

```

### ■ Laufzeit:

- ◆ Fall 1:  $\exists k : A[k] = q$

Schleife 2:  $\log_2 k + 1$  Durchläufe, dann gilt  $r/2 < k \leq r$   $O(\log k)$

Binäre Suche:  $O(\log r/2) = O(\log r - 1) =$   $O(\log k)$

Gesamtzeit:  $O(\log k)$

- ◆ Fall 2:  $\forall k : A[k] \neq q$

$O(\log N)$

Die Laufzeit von EXPONENTIAL-SEARCH ist **output-sensitiv**, d.h. sie hängt vom Resultat der Suche ab.



## Das Auswahlproblem

### ■ Auswahlproblem:

- Eingabe: Eine Menge A aus N (paarweise verschiedenen) Zahlen und einen Suchindex i mit  $1 \leq i \leq N$
- Ausgabe: Das Element  $x \in A$  mit  $A[k] < x$  für i-1 Elemente  $A[k]$

### ■ Bsp:

- i=1: Minimum-Problem  $O(N)$
- i=N: Maximum-Problem  $O(N)$
- i=N/2: Median-Problem ?

### ■ Algorithmus 1: Iteratives Löschen der Minima

TRIVIAL-SELECT(A, i)

$n \leftarrow \text{length}[A]$

**while**  $i > 0$  **do**

$k \leftarrow \text{MIN-INDEX}(A, n)$  // gibt den Index des min. Elements aus  $A[1..n]$

SWAP( $A[k], A[n]$ )

$i \leftarrow i-1; n \leftarrow n-1;$

**return**  $A[n+1]$

Laufzeit:  $T(N, i) = \Theta(i N) = O(N^2)$



## Der QUICK-SELECT Algorithmus

### ■ Algorithmus 2: Sortieren

SORT-SELECT(A, i)

HEAPSORT(A)

**output**  $A[i]$

Laufzeit:  $O(N \log N)$

### ■ Algorithmus 3: Partielle Sortierung mit Quicksort

RAND-SELECT(A, l, r, i)

1 **if**  $l = r$  **then return**  $A[l]$  // Annahme:  $l \leq i \leq r$

2 **else**

3  $q \leftarrow \text{RAND-PARTITION}(A, l, r)$

4 **if**  $i = q$  **then return**  $A[q]$

5 **else if**  $i < q$  **then** RAND-SELECT(A, l, q-1, i)

6 **else** RAND-SELECT(A, q+1, r, i)

Achtung: RAND-SELECT() und RANDOMIZED-SELECT() [Cormen Kap. 9.2] weichen geringfügig voneinander ab, lösen aber das gleiche Problem.



## Der RAND-SELECT Algorithmus

### ■ Beispiel: N=8, i=6

index: 1 2 3 4 5 6 7 8

A: 3 8 14 2 9 23 1 6 ← Pivot-Element

nach 3.: A: 3 1 2 6 9 23 8 14

$\boxed{\leq}$   $\uparrow q$   $\boxed{\geq}$

A: x x x x 9 23 8 14

nach 3.: A: x x x x 9 8 14 23

$\boxed{\leq}$   $\uparrow q$   $\boxed{\geq}$

A: x x x x 9 8 x x

nach 3.: A: x x x x 8 9 x x

$\uparrow q$   $\boxed{\geq}$

A: x x x x x 9 x x

$\uparrow l=r=i$





## Analyse des RAND-SELECT Algorithmus

- Rekurrenzgleichung für Rand-Select:
  - Sei  $q$  die Position des Pivot-Elements, dann gilt  
 $T(N) = T(q-1) + aN$  falls  $i < q$ ,  $T(N) = T(N-q) + aN$  sonst.
  - Worst-Case:  $q=N \Rightarrow T(N) = T(N-1) + aN = \Theta(N^2)$
- Wie hoch ist die erwartete Laufzeit  $E[T(N)]$  für Rand-Select?
  - Alle Elemente  $A[k]$  können gleichwahrscheinlich als Pivot-Element gewählt werden.
  - Im Worst-Case liegt der gesuchte Rang  $i$  immer in der größeren Partition

$$E[T(N)] \leq \sum_{k=1}^N \Pr\{q=k\} E[T(\max(k-1, N-k) + aN)]$$

$$= \sum_{k=1}^N \frac{1}{N} E[T(\max(k-1, N-k) + aN)] = \frac{1}{N} \sum_{k=1}^N E[T(\max(k-1, N-k)] + aN$$

- Für  $k < N/2$  ergibt sich für  $k$  und  $N-k+1$  der gleiche Summand:  
 $\max(k-1, N-k) = N-k = \max(N-k+1-1, N-(N-k+1))$

$$E[T(N)] \leq \frac{2}{N} \sum_{k=\lfloor N/2 \rfloor}^{N-1} E[T(k)] + aN$$



## Analyse des Rand-SELECT Algorithmus

- Welche Laufzeitschranke sollten wir intuitiv erwarten?
  - Im Vergleich zu Quicksort erfolgt nur ein rekursiver Aufruf, also höchstens  $O(N \log N)$
  - Angenommen,  $k=N/2$ , dann folgt  $T(N)=T(N/2)+cN = O(N)$
- Annahme:  $E[T(N)] = O(N)$ , d.h.  $E[T(N)] \leq cN$  für ein konstantes  $c$
- Beweis:

$$E[T(N)] \leq \frac{2}{N} \sum_{k=\lfloor N/2 \rfloor}^{N-1} ck + aN$$

$$= \frac{2c}{N} \sum_{k=\lfloor N/2 \rfloor}^{N-1} (k + \lfloor N/2 \rfloor) + aN$$

$$= \frac{2c(\lfloor N/2 \rfloor - 1)(\lfloor N/2 \rfloor)}{N} + \frac{2c(\lfloor N/2 \rfloor - 1)(\lfloor N/2 \rfloor)}{N} + aN$$

$$\leq \frac{2c(N/2)(N/2)}{N} + \frac{2c(N/2)(N/2+1)}{N} + aN$$

$$= \frac{cN}{2} + \frac{cN}{4} + \frac{c}{2} + aN = \frac{3cN}{4} + \frac{c}{2} + aN$$

$$= cN - \left( \frac{cN}{4} - \frac{c}{2} - aN \right) \leq cN \quad \text{für } N \geq \frac{2c}{c-4a}$$



## Der SELECT-Algorithmus

- Gibt es einen deterministischen Algorithmus für das Auswahlproblem mit asymptotisch linearer Laufzeit im Worst-Case?
- Idee: Verwende Rand-Select und garantiere, dass die Position  $q$  des Pivotelement nahe bei  $N/2$  liegt.

- SELECT( $A, l, r, i$ )
 

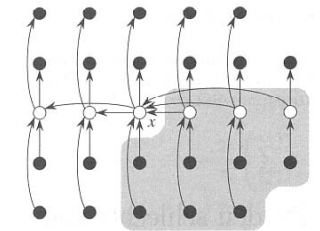
```

1 if l = r then return A[l]           // Annahme: l ≤ i ≤ r
2 else
3   q ← FIND-PIVOT-POSITION(A, l, r)
4   SWAP(A[q], A[r])
5   q ← PARTITION(A, l, r)
6   if i = q then return A[q]
7   else if i < q then SELECT(A, l, q-1, i)
8   else SELECT(A, q+1, r, i)
            
```



## Der SELECT-Algorithmus

- Idee: Teile Menge in Fünfer-Gruppen, berechne die Fünfer-Mediane und den Median der Fünfer-Mediane:



- FIND-PIVOT-POSITION( $A, l, r$ )
 

```

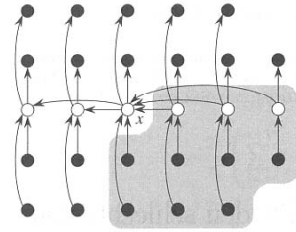
n ← r-l+1
// berechne die Mediane von je 5 benachbarten Elementen
for i ← 1 to ⌈n/5⌉ do
    INSERTION-SORT(A[ (l + 5(i-1)), ..., min(l + 5i, r) ])
    M[i] ← A[ l + 5i - 3 ]
// berechne rekursiv den Median der Mediane-von-5
x ← SELECT(M, 1, ⌈n/5⌉, ⌊⌈n/5⌉/2⌋)
return (l + 5x - 3)
            
```



## Die Analyse des SELECT-Algorithmus

- FIND-PIVOT-POSITION: Wie viele Elemente aus  $A[l, \dots, r]$  sind größer als  $A[x]$ ?

- die Hälfte der Fünfer-Mediane + je zwei weitere Elemente
- ohne die letzte Gruppe und
- ohne die Gruppe, die  $x$  enthält



$$3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3N}{10} - 6 = G(N)$$

- Der rekursive Aufruf von SELECT (Zeile 7 oder 8) erfolgt mit maximal  $7N/10 + 6$  Elementen.



## Die Analyse des SELECT-Algorithmus

- Die Rekurrenz-Gleichung für SELECT:

$$T(N) = \begin{cases} c & \text{falls } n \leq 70 \\ T(\lceil N/5 \rceil) + T(7N/10 + 6) + aN & \text{sonst} \end{cases}$$

$\nwarrow$  SELECT-Aufruf in FIND-PIVOT-POSITION       $\nwarrow$  SELECT-Aufruf in SELECT       $\nwarrow$  - Bestimmung der Fünfer-Mediane - Aufruf von PARTITION

- Annahme:  $T(N) = O(N)$ , d.h.  $T(N) \leq cN$  für ein konstantes  $c$

- Beweis:  $T(N) = c$  für alle  $N \leq 70$ . Sei  $N > 70$ , dann gilt

$$\begin{aligned}
 T(N) &\leq c \left\lceil \frac{N}{5} \right\rceil + c \left( \frac{7N}{10} + 6 \right) + aN \\
 &\leq \frac{cN}{5} + c + \frac{7cN}{10} + 6c + aN = \frac{9cN}{10} + 7c + aN \\
 &= cN - \left( \frac{1}{10}cN - 7c - aN \right) \leq cN \\
 &\Leftrightarrow \frac{1}{10}cN - 7c - aN \geq 0 \Leftrightarrow c \geq 10a \left( \frac{N}{N-70} \right) \text{ für } N > 70
 \end{aligned}$$



## Abschließende Bemerkungen zum Auswahlproblem

- Das Auswahlproblem kann in asymptotisch erwarteter linearer Zeit mit dem **RAND-SELECT-Algorithmus** gelöst werden.
- Es gibt einen deterministischen Algorithmus (**SELECT-Algorithmus**), der das Auswahlproblem im Worst Case in asymptotisch linearer Zeit löst.
- Der **SELECT-Algorithmus** ist aufgrund hoher Konstanten nur von Interesse, falls die Eingaben sehr groß sind und eine Laufzeitgarantie gefordert ist.
- Lehren für die Algorithmenentwicklung:
  - Ein randomisierter, rekursiver Algorithmus, der in asymptotisch linearer Zeit die Eingabe gleichverteilt um einen Faktor  $\alpha \in [0, 1)$  verkleinert, hat asymptotisch eine erwartete lineare Laufzeit.
  - Ein deterministischer, rekursiver Algorithmus, der in asymptotisch linearer Zeit die Eingabe um einen Faktor  $\alpha \in [0, 1)$  verkleinert, hat asymptotisch eine lineare Laufzeit.





## **Kapitel 4: Suchen**

- 4.1 Suchbäume 75
- 4.2 Rot-Schwarz-Bäume 83
- 4.3 AVL-Bäume 98
- 4.4 Hashing mit Verkettung 101
- 4.5 Offene Adressierung 107

## Kapitel 4: Suchen

Suchbäume  
Rot-Schwarz-Bäume  
AVL-Bäume  
Hashing mit Verkettung  
Hashing mit offener Adressierung

## 4.1 Suchbäume

### ■ Datenstrukturen für Wörterbücher:

- dynamische Datenstruktur: Objekte können eingefügt (INSERT-Operation) und gelöscht (DELETE-Operation)
- Objekte besitzen einen Schlüssel  $key[]$ , für den die totale Ordnungsrelation  $\leq$  definiert ist.
- Die Datenstruktur erlaubt den geordneten Zugriff auf Objekte:
  - ◆  $SEARCH(T, x)$ : Objekt mit dem Schlüssel  $x$
  - ◆  $MINIMUM(T)$  /  $MAXIMUM(T)$ : Objekt mit kleinstem/größtem Schlüssel
  - ◆  $PREDECESSOR(T, x)$  /  $SUCCESSOR(T, x)$ : Objekt mit nächst kleinerem/größerem Schlüssel

- **Ziel:** Entwicklung einer Datenstruktur mit asymptotischer Worst-Case Laufzeit  $O(\log N)$  für alle Operationen bei Speicherung von  $N$  Objekten.



## Suchbäume

### ■ Binärer Suchbaum:

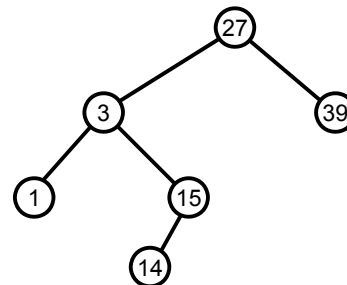
Geordneter Baum  $T$  mit ausgezeichnete Wurzel  $root[T]$ , die Knoten  $x$  haben die Attribute:

- $key[x]$  /  $schlüssel[x]$ : Schlüssel des gespeicherten Objekts
- $left[x]$  /  $links[x]$ : linker Kind-Knoten von  $x$
- $right[x]$  /  $rechts[x]$ : rechter Kind-Knoten von  $x$
- $p[x]$ : Elter-Knoten von  $x$

### ■ Binäre Suchbaumeigenschaft:

Für jeden Knoten  $x$  gilt:

Sei  $y$  ein Knoten im linken Teilbaum von  $x$ , dann gilt  $key[y] < key[x]$ .  
Sei  $y$  ein Knoten im rechten Teilbaum von  $x$ , dann gilt  $key[y] \geq key[x]$ .



## Traversieren von Bäumen

### ■ Traversierung: Systematisches Durchlaufen aller Knoten eines Baumes

#### ■ Mögliche Reihenfolgen:

- **Preorder:** Wurzel, linker Teilbaum, rechter Teilbaum
- **Postorder:** linker Teilbaum, rechter Teilbaum, Wurzel
- **Inorder:** linker Teilbaum, Wurzel, rechter Teilbaum

### ■ INORDER-TREE-WALK( $x$ )

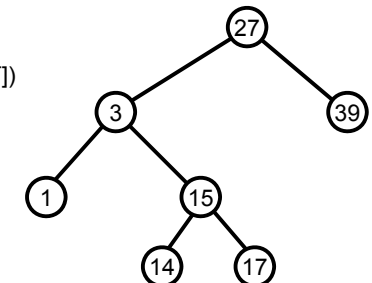
// Aufruf mit  $INORDER-TREE-WALK(root[T])$

**if**  $x \neq NIL$  **then**

$INORDER-TREE-WALK(left[x])$

**print**  $key[x]$

$INORDER-TREE-WALK(right[x])$



Inorder: 1, 3, 14, 15, 17, 27, 39



## Traversierung von Bäumen

- **Theorem 1:** Erfüllt ein Baum T die binäre Suchbaumeigenschaft, dann gibt die Funktion INORDER-TREE-WALK() die Objekte nach aufsteigenden Schlüsselwerten aus.
- Beweis: Für jedes Objektpaar x, y mit  $x \leq y$  gilt: Sei p der kleinste gemeinsame Vorfahr von x und y.
  - Fall 1:  $x=p$  befindet sich im rechten Teilbaum.
  - Fall 2:  $y=p$  befindet sich im linken Teilbaum.
  - Fall 3:  $x \neq p \neq y$  x befindet sich im rechten, y im linken Teilbaum.
- **Theorem 2:** INORDER-TREE-WALK() benötigt auf einem Baum mit N Knoten die Laufzeit  $O(N)$
- Beweis: Sei k die Größe des linken Teilbaums. Es gilt die Rekurrenzgleichung:

$$T(N) = \begin{cases} c & : N \leq 1 \\ T(k) + T(N-k-1) + d & : \text{sonst} \end{cases}$$

Es gilt  $T(N) = (c+d)N + c = O(N)$ . (Beweis durch Induktion)



## Suchen in Suchbäume

- **TREE-SEARCH(x,k)**

```

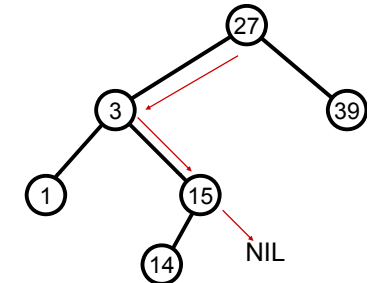
if x = NIL or k = key[x] then return x
else if k < key[x] then return TREE-SEARCH(left[x], k)
else return TREE-SEARCH(right[x], k)
            
```

- **ITERATIVE-TREE-SEARCH(x,k)**

```

while x ≠ NIL and k ≠ key[x] do
  if k < key[x] then x ← left[x]
  else x ← right[x]
            
```

- Bsp: TREE-SEARCH(root[T], 17):



- **Theorem 3:** Sei h die Höhe des Suchbaum T, dann hat TREE-SEARCH(root[T],k) Laufzeit  $O(h)$ .
- Beweis: TREE-SEARCH durchläuft einen Pfad von der Wurzel bis zu einem Blatt mit konstanter Laufzeit pro besuchten Knoten.



## Minimum- und Successor-Funktion

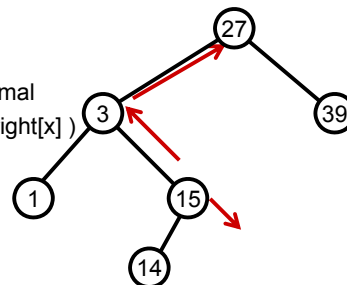
- **TREE-MINIMUM(x)**

```

// Aufruf mit TREE-MINIMUM(root[T])
while left[x] ≠ NIL do x ← left[x]
return x
            
```
- **TREE-SUCCESSOR(x)**

```

// finde Knoten y mit key[y] ≥ key[x], key[y] minimal
if right[x] ≠ NIL then return TREE-MINIMUM(right[x])
else
  y ← p[x]
  while y ≠ NIL and x = right[y] do
    x ← y; y ← p[y]
  return y
            
```



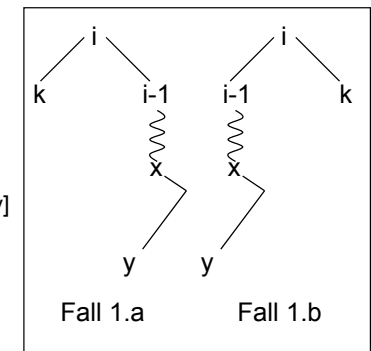
- **Theorem 4:** Sei h die Höhe des Suchbaum T, dann hat TREE-MINIMUM(root[T]) Laufzeit  $O(h)$ ; TREE-SUCCESSOR() für jeden Knoten x des Suchbaums Worst-Case Laufzeit  $O(h)$ .
- Beweis: Für jeden Knoten x durchläuft TREE-SUCCESSOR entweder den Pfad von x zu einem Blatt oder zur Wurzel.



## Korrektheit von TREE-SUCCESSOR()

- **Theorem 5:** TREE-SUCCESSOR(x) bestimmt Knoten y mit den Eigenschaften  $\text{key}[y] \geq \text{key}[x]$  und  $\text{key}[y]$  minimal.
- Beweis: Sei  $T[k]$  der Teilbaum unter Knoten k. Sei  $x = p_0, p_1, \dots, p_l = \text{root}[T]$  der Pfad von x zur Wurzel.
  - Fall 1: x hat einen rechten Nachfolger: Sei  $y \in T[\text{right}[x]]$  mit  $\text{key}[y]$  minimal.  $\forall i \in \{1, \dots, l\}$  gilt:

- ◆ Fall 1.a:  $p_{i-1} = \text{right}[p_i]$ :  
 $\forall k \in T[\text{left}[p_i]]: \text{key}[k] \leq \text{key}[p_i] \leq \text{key}[x]$
- ◆ Fall 1.b:  $p_{i-1} = \text{left}[p_i]$ :  
 $\forall k \in T[\text{right}[p_i]]: \text{key}[k] \geq \text{key}[p_i] \geq \text{key}[y]$



## Korrektheit von TREE-SUCCESSOR()

- Fall 2: x hat keinen rechten Nachfolger. Sei  $y = p_j$  mit  $p_{j-1} = \text{left}[p_j]$ , j minimal.

- ♦ Fall 2.a:  $\forall i \in \{1, \dots, j-1\}$ :

$$\forall k \in T[\text{left}[p_i]]: \text{key}[k] \leq \text{key}[p_i] \leq \text{key}[x]$$

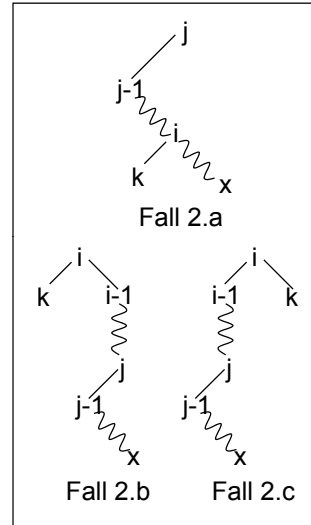
- ♦ Fall 2.b:  $\forall i \in \{j+1, \dots, l\}$  mit  $p_{i-1} = \text{right}[p_i]$ :

$$\forall k \in T[\text{left}[p_i]]: \text{key}[k] \leq \text{key}[p_i] \leq \text{key}[x]$$

- ♦ Fall 2.c:  $\forall i \in \{j+1, \dots, l\}$  mit  $p_{i-1} = \text{left}[p_i]$ :

$$\forall k \in T[\text{right}[p_i]]: \text{key}[k] \geq \text{key}[p_i] \geq \text{key}[y]$$

Da TREE-SUCCESSOR() y gemäß Fall 1 oder 2 berechnet, folgt das Theorem.



## Einfügen in Suchbäume

- Idee: laufe unter Berücksichtigung der Suchbaumeigenschaft von der Wurzel zu einem Blatt, hänge das neue Objekt an.

- TREE-INSERT(T, z)

$x \leftarrow \text{root}[T]$

$y \leftarrow \text{NIL}$  // speichert p[x]

**while**  $x \neq \text{NIL}$  **do**

$y \leftarrow x$

**if**  $\text{key}[z] < \text{key}[x]$  **then**  $x \leftarrow \text{left}[x]$

**else**  $x \leftarrow \text{right}[x]$

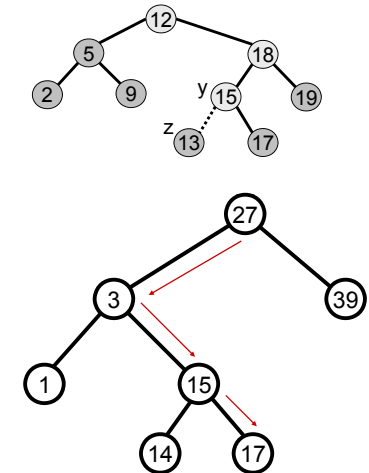
$p[z] \leftarrow y$

**if**  $y = \text{NIL}$  **then**  $\text{root}[T] \leftarrow z$

**else if**  $\text{key}[z] < \text{key}[y]$  **then**  $\text{left}[y] \leftarrow z$

**else**  $\text{right}[y] \leftarrow z$

- Bsp: TREE-INSERT() mit  $\text{key}[z]=17$ :



## Einfügen in Suchbäume

- REC-TREE-INSERT(x, z)

// Aufruf mit REC-TREE-INSERT(root[T], z) für nicht-leere Bäume

**if**  $\text{key}[z] < \text{key}[x]$  **then**

**if**  $\text{left}[x] \neq \text{NIL}$  **then** REC-TREE-INSERT(left[x], z)

**else**  $p[z] \leftarrow x$ ;  $\text{left}[x] \leftarrow z$

**else**

**if**  $\text{right}[x] \neq \text{NIL}$  **then** REC-TREE-INSERT(right[x], z)

**else**  $p[z] \leftarrow x$ ;  $\text{right}[x] \leftarrow z$

- Theorem 6:** Sei h die Höhe des Suchbaum T, dann hat TREE-INSERT(T, z) und REC-TREE-INSERT(root[T], z) die asymptotische Worst-Case Laufzeit  $O(h)$ .



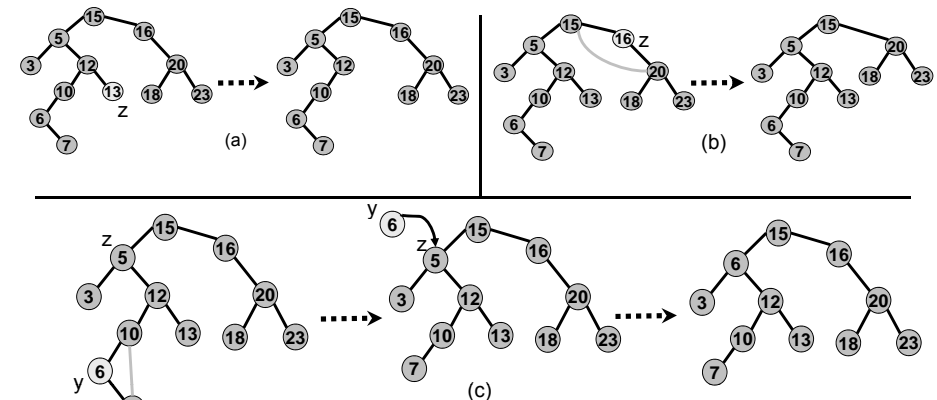
## Löschen aus Suchbäumen

- Löschen eines Knotens z unter Erhalt der Suchbaum-Eigenschaft:

- Fall 1: z ist ein Blatt: Lösche z

- Fall 2: z hat nur einen Nachfolger x: x wird Nachfolger von p[z]

- Fall 3: z hat zwei Nachfolger: Ersetze z durch TREE-SUCCESSOR(z)



## Löschen aus Suchbäumen

```

■ TREE-DELETE-NODE(T, z)      // Cormen: TREE-DELETE()
// y: der (nach Tausch) zu löschende Knoten
1 if left[z] = NIL or right[z] = NIL then y ← z
2 else y ← TREE-MINIMUM( right[z] ) // Cormen: TREE-SUCCESSOR(z)
// x: Kind von y, das nicht NIL ist
3 if left[y] ≠ NIL then x ← left[y] else x ← right[y]
// entferne y aus der Baumstruktur
4 if x ≠ NIL then p[x] ← p[y]
5 if p[y] = NIL then root[T] ← x
6 else if y = left[p[y]] then left[p[y]] ← x
7           else right[p[y]] ← x
// kopiere Daten von y nach z
8 if y ≠ z then key[z] ← key[y]
9 delete y

■ TREE-DELETE(T, k)
TREE-DELETE-NODE( T, TREE-SEARCH(root[T], k) )

```



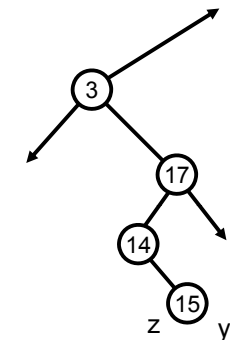
## Löschen aus Suchbäumen

```

■ TREE-DELETE-NODE(T, z)
// y: der (nach Tausch) zu löschende Knoten
1 if left[z] = NIL or right[z] = NIL then y ← z
2 else y ← TREE-MINIMUM( right[z] )
// x: Kind von y, das nicht NIL ist
3 if left[y] ≠ NIL then x ← left[y] else x ← right[y]
// entferne y aus der Baumstruktur
4 if x ≠ NIL then p[x] ← p[y]
5 if p[y] = NIL then root[T] ← x
6 else if y = left[p[y]] then left[p[y]] ← x
7           else right[p[y]] ← x
// kopiere Daten von y nach z
8 if y ≠ z then key[z] ← key[y]
9 delete y

```

Fall 1:



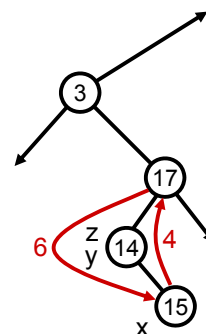
## Löschen aus Suchbäumen

```

■ TREE-DELETE-NODE(T, z)
// y: der (nach Tausch) zu löschende Knoten
1 if left[z] = NIL or right[z] = NIL then y ← z
2 else y ← TREE-MINIMUM( right[z] )
// x: Kind von y, das nicht NIL ist
3 if left[y] ≠ NIL then x ← left[y] else x ← right[y]
// entferne y aus der Baumstruktur
4 if x ≠ NIL then p[x] ← p[y]
5 if p[y] = NIL then root[T] ← x
6 else if y = left[p[y]] then left[p[y]] ← x
7           else right[p[y]] ← x
// kopiere Daten von y nach z
8 if y ≠ z then key[z] ← key[y]
9 delete y

```

Fall 2:



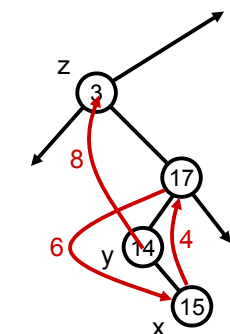
## Löschen aus Suchbäumen

```

■ TREE-DELETE-NODE(T, z)
// y: der (nach Tausch) zu löschende Knoten
1 if left[z] = NIL or right[z] = NIL then y ← z
2 else y ← TREE-MINIMUM( right[z] )
// x: Kind von y, das nicht NIL ist
3 if left[y] ≠ NIL then x ← left[y] else x ← right[y]
// entferne y aus der Baumstruktur
4 if x ≠ NIL then p[x] ← p[y]
5 if p[y] = NIL then root[T] ← x
6 else if y = left[p[y]] then left[p[y]] ← x
7           else right[p[y]] ← x
// kopiere Daten von y nach z
8 if y ≠ z then key[z] ← key[y]
9 delete y

```

Fall 3:



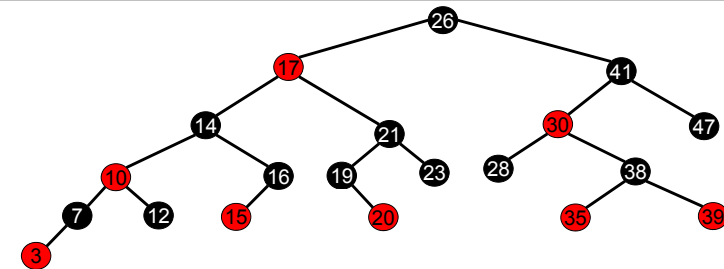
## Suchbäume

- **Theorem 7:** Sei  $h$  die Höhe des Suchbaum  $T$ , dann hat  $\text{TREE-DELETE}(T, k)$  die asymptotische Worst-Case Laufzeit  $O(h)$ .
  - $\text{TREE-SEARCH}$  durchläuft einen Pfad von der Wurzel zum zu löschenden Knoten  $z$ .
  - $\text{TREE-DELETE-NODE}$  durchläuft einen Pfad vom zu löschenden Knoten  $z$  zu einem Blatt.
- Alle Wörterbuch-Operationen können im Suchbaum in Laufzeit  $O(h)$  realisiert werden.
- Definition: Ein Suchbaum  $T$  mit Höhe  $h(T)$  heißt **balanciert**, g.d.w.  $h(T) = O(\log |T|)$  gilt.
- **Welche Höhe hat ein Suchbaum?**

■ Best Case	Zufällig erzeugter Binärbaum	Worst Case
$O(\log N)$	$O(\log N)$	$\Omega(N)$



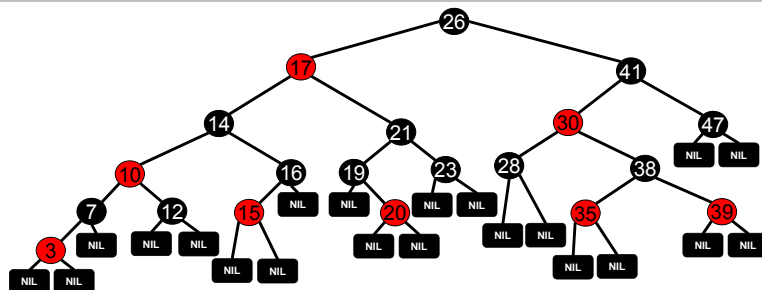
## 4.2 Rot-Schwarz-Bäume



- **Rot-Schwarz-Bäume:** Suchbäume mit den Knoten-Attributen:
  - $p[x]$  : Vorgänger
  - $\text{left}[x], \text{right}[x]$  : linker und rechter Nachfolger
  - $\text{key}[x]$  : Schlüsselement
  - $\text{color}[x]$  : Farbe (rot oder schwarz)
  - $\text{color}[]$  erfüllt die *Rot-Schwarz-Eigenschaften*
- Blätter werden durch den Wächter  $\text{NIL} = \text{nil}[T]$  repräsentiert
- Definition: Sei  $T(x)$  der Teilbaum unter  $x$ ,  $h(T)$  die Höhe eines Baumes



## Rot-Schwarz-Bäume



- **Rot-Schwarz-Eigenschaften:**
  1. Jeder Knoten ist entweder rot oder schwarz.
  2. Die Wurzel ist schwarz.
  3. Jedes Blatt (der Wächter) ist schwarz.
  4. Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.
  5. Für jeden Knoten  $x$  gilt: Alle Pfade von  $x$  zu einem Blatt enthalten die gleiche Anzahl  $bh(x)$  schwarzer Knoten.
- Definition:  $bh(x)$ : Schwarz-Höhe des Knotens  $x$



## Höhe von Rot-Schwarz-Bäumen

- **Lemma:** Ein Rot-Schwarz-Baum  $T$  mit  $N$  inneren Knoten hat höchstens die Höhe  $2\log_2(N+1)$ .
- Beweis:
  - **Teil 1:** Zeige  $\forall x |T(x)| \geq 2^{bh(x)} - 1$  durch Induktion über  $h(T(x))$ 
    - Für  $x$  mit  $h(T(x))=0$  gilt:  $bh(x)=0, 1 = |T(x)| \geq 2^0 - 1 = 0$
    - ◆ Sei  $x$  ein Knoten mit  $h(T(x)) > 0$ :
    - ◆ **Fall 1:**  $\text{color}[x] = \text{rot}$ 
      - $\Rightarrow \text{color}[\text{left}[x]] = \text{color}[\text{right}[x]] = \text{schwarz}$  und somit
      - $bh(\text{left}[x]) = bh(\text{right}[x]) = bh(x)$
      - $|T(x)| = 1 + |T(\text{left}[x])| + |T(\text{right}[x])| \geq 1 + 2^{bh(x)-1} + 2^{bh(x)-1} \geq 2^{bh(x)} - 1$
    - ◆ **Fall 2:**  $\text{color}[x] = \text{schwarz}$ 
      - $bh(\text{left}[x]) = bh(\text{right}[x]) = bh(x) - 1$
      - $|T(x)| = 1 + |T(\text{left}[x])| + |T(\text{right}[x])| \geq 1 + 2^{bh(x)-1-1} + 2^{bh(x)-1-1} = 2^{bh(x)} - 1$
  - **Teil 2:**
    - Es gilt  $h(T) \leq 2 bh(\text{root}[T])$  (nach Rot-Schwarz-Eigenschaft 4)
    - $\Rightarrow N = |T| \geq 2^{bh(\text{root}[T])} - 1 \geq 2^{h(T)/2} - 1 \Leftrightarrow h(T) \leq 2 \log_2(N+1)$

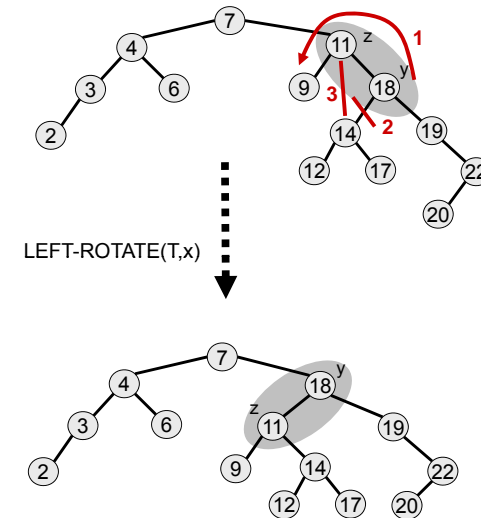


## Such-Komplexität in Rot-Schwarz-Bäumen

- Ein Rot-Schwarz-Baum mit  $n$  Knoten hat eine maximale Höhe von  $O(\log N)$  und ist somit balanciert.
- Sei  $T$  ein Suchbaum, der die Rot-Schwarz-Eigenschaft erfüllt.
  - Die Operationen `SEARCH()`, `MINIMUM()`, `MAXIMUM()`, `SUCCESSOR()` und `PREDECESSOR()` laufen in Zeit  $O(h) = O(\log N)$
  - Die Operationen `TREE-INSERT()` und `TREE-DELETE()` laufen in Zeit  $O(\log N)$
- `TREE-INSERT()` und `TREE-DELETE()` können zu Suchbäumen führen, die die Rot-Schwarz-Eigenschaft verletzen!
  - Korrektur-Mechanismus zur Wiederherstellung der Rot-Schwarz-Eigenschaft (und somit der Balance) wird benötigt.



## Rotationen in Suchbäumen

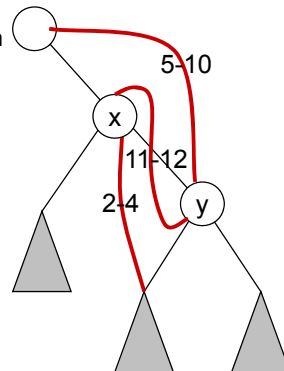


LEFT-ROTATE( $T, x$ )



## Rotationen in Suchbäumen

- Definition: Eine **Rotation** ist eine lokale Operation (Umsetzen einer konstanten Anzahl von Zeigern) in Suchbäumen, die die Suchbaum-Eigenschaft erhält.
- LEFT-ROTATE( $T, x$ )
  - 1  $y \leftarrow \text{right}[x]$
  - //  $y$ 's linker Teilbaum  $\rightarrow x$ 's rechter Teilbaum
  - 2  $\text{right}[x] \leftarrow \text{left}[y]$
  - 3 if  $\text{left}[y] \neq \text{nil}[T]$  then  $p[\text{left}[y]] \leftarrow x$
  - // verbinde Vater von  $x$  mit  $y$
  - 5  $p[y] \leftarrow p[x]$
  - 6 if  $p[x] = \text{nil}[T]$  then  $\text{root}[T] \leftarrow y$
  - 8 else if  $x = \text{left}[p[x]]$  then  $\text{left}[p[x]] \leftarrow y$
  - 10 else  $\text{right}[p[x]] \leftarrow y$
  - // verschiebe  $x$  auf die linke Seite von  $y$
  - 11  $\text{left}[y] \leftarrow x$
  - 12  $p[x] \leftarrow y$



## Einfügen in Rot-Schwarz-Bäumen

- Durchlaufe den Suchbaum von der Wurzel bis einem Blatt, füge den Schlüssel ein, färbe den Knoten rot.
- RB-INSERT( $T, z$ )
 

```

x ← root[T]
y ← nil[T] // speichert p[x]
while x ≠ nil[T] do
    y ← x
    if key[z] < key[x] then x ← left[x]
    else x ← right[x]
p[z] ← y
if y = nil[T] then root[T] ← z
else if key[z] < key[y] then left[y] ← z
    else right[y] ← z
left[z] ← right[z] ← nil[T]
color[z] ← ROT
// korrigiere die Rot-Schwarz-Eigenschaften
RB-INSERT-FIXUP(T, z)
            
```



## Korrektur der Rot-Schwarz-Eigenschaft

### ■ Welche Eigenschaften können verletzt sein?

1. Jeder Knoten ist entweder rot oder schwarz. ✓
2. Die Wurzel ist schwarz.
3. Jedes Blatt (der Wächter) ist schwarz. ✓
4. Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.
5. Für jeden Knoten x gilt: Alle Pfade von x zu einem Blatt enthalten die gleiche Anzahl  $bh(x)$  schwarzer Knoten. ✓

### ■ zu 2.: Ist verletzt, falls z die Wurzel ist

### ■ zu 4.: Ist verletzt, falls $color[p[z]] = ROT$ ist

### ■ Die Rot-Schwarz-Eigenschaften sind höchstens an einem Knoten (Eigenschaft 2: z oder Eigenschaft 4: p[z]) verletzt.

### ■ Idee von RB-INSERT-FIXUP:

Korrigiere die Eigenschaftsverletzung 4 lokal oder verschiebe die Eigenschaftsverletzung 4 sukzessiv zum Vorgänger. Wenn die Wurzel erreicht wird, korrigiere Eigenschaft 2.

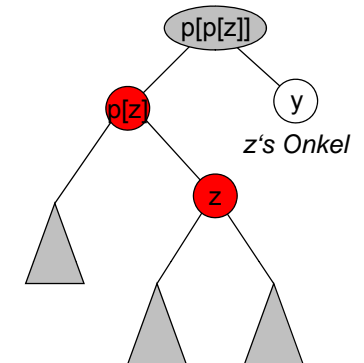


## Die Funktion RB-INSERT-FIXUP()

### ■ RB-INSERT-FIXUP(T, z)

```

1 while color[p[z]] = ROT do
2   if p[z] = left[p[p[z]]] then
3     y ← right[p[p[z]]]
4     if color[y] = ROT then
5       color[p[z]] ← SCHWARZ
6       color[y] ← SCHWARZ
7       color[p[p[z]]] ← ROT
8       z ← p[p[z]]
9     else if z = right[p[z]] then
10      z ← p[z]
11      LEFT-ROTATE(T, z)
12      color[p[z]] ← SCHWARZ
13      color[p[p[z]]] ← ROT
14      RIGHT-ROTATE(T, p[p[z]])
15   else // p[z] = right[p[p[z]]]
        : // wie then-Teil, vertausche right ⇔ left
16 color[root[T]] ← SCHWARZ
    
```

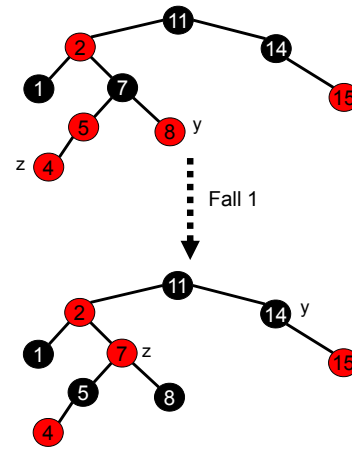


## Die Funktion RB-INSERT-FIXUP() – Fall 1

### ■ RB-INSERT-FIXUP(T, z)

```

1 while color[p[z]] = ROT do
2   if p[z] = left[p[p[z]]] then
3     y ← right[p[p[z]]]
4     if color[y] = ROT then
5       color[p[z]] ← SCHWARZ
6       color[y] ← SCHWARZ
7       color[p[p[z]]] ← ROT
8       z ← p[p[z]]
9     else if z = right[p[z]] then
10      z ← p[z]
11      LEFT-ROTATE(T, z)
12      color[p[z]] ← SCHWARZ
13      color[p[p[z]]] ← ROT
14      RIGHT-ROTATE(T, p[p[z]])
15   else // p[z] = right[p[p[z]]]
        : // wie then-Teil, vertausche right ⇔ left
16 color[root[T]] ← SCHWARZ
    
```



Fall 4-6 (symmetrisch zu 1-3)

Fall 1

Fall 2

Fall 3

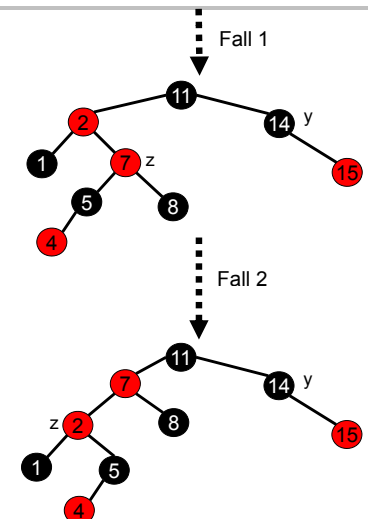


## Die Funktion RB-INSERT-FIXUP() – Fall 2

### ■ RB-INSERT-FIXUP(T, z)

```

1 while color[p[z]] = ROT do
2   if p[z] = left[p[p[z]]] then
3     y ← right[p[p[z]]]
4     if color[y] = ROT then
5       color[p[z]] ← SCHWARZ
6       color[y] ← SCHWARZ
7       color[p[p[z]]] ← ROT
8       z ← p[p[z]]
9     else if z = right[p[z]] then
10      z ← p[z]
11      LEFT-ROTATE(T, z)
12      color[p[z]] ← SCHWARZ
13      color[p[p[z]]] ← ROT
14      RIGHT-ROTATE(T, p[p[z]])
15   else // p[z] = right[p[p[z]]]
        : // wie then-Teil, vertausche right ⇔ left
16 color[root[T]] ← SCHWARZ
    
```



Fall 4-6 (symmetrisch zu 1-3)

Fall 1

Fall 2

Fall 3





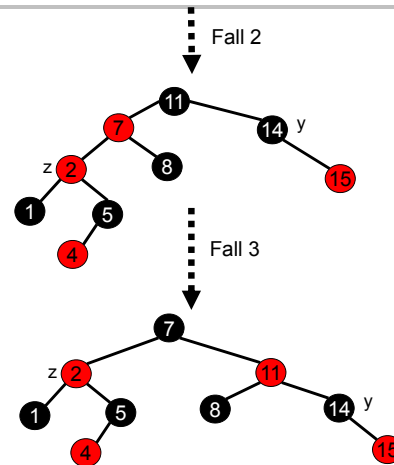
## Die Funktion RB-INSERT-FIXUP() – Fall 3

### ■ RB-INSERT-FIXUP(T, z)

```

1 while color[p[z]] = ROT do
2   if p[z] = left[p[p[z]]] then
3     y ← right[p[p[z]]]
4     if color[y] = ROT then
5       color[p[z]] ← SCHWARZ
6       color[y] ← SCHWARZ
7       color[p[p[z]]] ← ROT
8       z ← p[p[z]]
9     else if z = right[p[z]] then
10      z ← p[z]
11      LEFT-ROTATE(T, z)
12      color[p[z]] ← SCHWARZ
13      color[p[p[z]]] ← ROT
14      RIGHT-ROTATE(T, p[p[z]])
15   else // p[z] = right[p[p[z]]]
        : // wie then-Teil, vertausche right ↔ left
16 color[root[T]] ← SCHWARZ

```



Fall 4-6 (symmetrisch zu 1-3)

Fall 1

Fall 2

Fall 3



## Die Funktion RB-INSERT-FIXUP() - Korrektheit

### ■ Invariante der while-Schleife 1-15:

1.  $\text{color}[z] = \text{ROT}$
2. Falls  $p[z] = \text{root}[T]$ , gilt:  $\text{color}[p[z]] = \text{SCHWARZ}$
3. Es gibt maximal eine Verletzung der Rot-Schwarz-Eigenschaft:
  1. Eigenschaft 2  $\Rightarrow z = \text{root}[T]$  und  $\text{color}[z] = \text{ROT}$
  2. Eigenschaft 4  $\Rightarrow \text{color}[z] = \text{ROT}$  und  $\text{color}[p[z]] = \text{ROT}$

### ■ Gültigkeit der Invariante zum Zeitpunkt der Initialisierung:

- zu 1.: Knoten  $z$  mit  $\text{color}[z] = \text{ROT}$  wurde in einen Rot-Schwarz-Baum ohne Verletzung eingefügt.
- zu 2.: Falls  $p[z] = \text{root}[T]$ :  $\text{color}[p[z]]$  wurde nicht verändert, da  $T$  ein Rot-Schwarz-Baum war, gilt  $\text{color}[p[z]] = \text{SCHWARZ}$ .
- zu 3.: Offensichtlich gelten Rot-Schwarz-Eigenschaften 1, 3 und 5 (siehe Folie 25).



## Die Funktion RB-INSERT-FIXUP() - Korrektheit

### ■ Korrektheit von RB-INSERT-FIXUP() zum Zeitpunkt der Terminierung:

- Schleifenbedingung nicht mehr erfüllt:  $\text{color}[p[z]] = \text{SCHWARZ}$
- [Falls  $z$  die Wurzel ist, ist  $p[z] = \text{nil}[T]$  (Wächter) mit  $\text{color}[\text{nil}[T]] = \text{SCHWARZ}$ ]

Rot-Schwarz-Eigenschaften (zur Erinnerung)

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt (der Wächter) ist schwarz.
4. Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.
5. Für jeden Knoten  $x$  gilt: Alle Pfade von  $x$  zu einem Blatt enthalten die gleiche Anzahl  $bh(x)$  schwarzer Knoten.

- Rot-Schwarz-Eigenschaften 1,3 und 5 sind erfüllt.
- $\text{color}[z] = \text{ROT}$  und  $\text{color}[p[z]] = \text{SCHWARZ} \Rightarrow$  Eigenschaft 4. ist nicht verletzt.
- Eigenschaft 2 kann nach Terminierung der Schleife verletzt sein. Zeile 16 von RB-INSERT-FIXUP() stellt Eigenschaft 2 her.

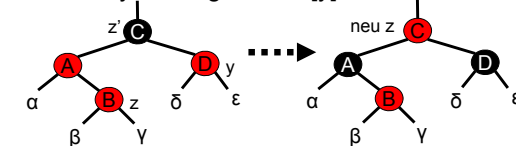


## Die Funktion RB-INSERT-FIXUP() - Korrektheit

### ■ Die Invariante bleibt bei der Fortsetzung der Schleife erhalten:

- Ist  $p[p[z]]$  definiert?  $\text{color}[p[z]] = \text{ROT}$ , mit Teil 2. der Invariante gilt  $p[z] \neq \text{root}[T]$ , damit existiert  $p[p[z]]$ .

### ■ Fall 1: Für Onkel $y$ von $z$ gilt: $\text{color}[y] = \text{ROT}$

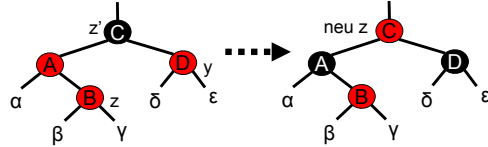


- $\text{color}[p[p[z]]] = \text{SCHWARZ}$ , da  $\text{color}[y] = \text{ROT}$  und  $p[y] = p[p[z]]$  gilt
- Alle Nachfolger  $\alpha-\epsilon$  von  $z$ ,  $p[z]$  (außer  $z$ ) und  $y$  sind schwarz und haben die gleiche Schwarz-Höhe  $bh()$ .
- Sei  $z'$  der Wert der Variablen  $z$  nach Ausführung der Iteration:
  1.  $\text{color}[z] = \text{ROT}$   
 $z' = p[p[z]]$  und  $\text{color}[p[p[z]]] = \text{ROT}$  nach Zeilen 7 und 8.
  2. Falls  $p[z] = \text{root}[T]$ , gilt:  $\text{color}[p[z]] = \text{SCHWARZ}$   
 $p[z'] = p[p[p[z]]]$  ändert seine Farbe nicht.



## Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Fall 1:** Für Onkel  $y$  von  $z$  gilt:  $\text{color}[y] = \text{ROT}$



- Fortsetzung:

3. Es gibt maximal eine Verletzung der Rot-Schwarz-Eigenschaft:

1. Eigenschaft 2  $\Rightarrow z = \text{root}[T]$  und  $\text{color}[z] = \text{ROT}$
2. Eigenschaft 4  $\Rightarrow \text{color}[z] = \text{ROT}$  und  $\text{color}[p[z]] = \text{ROT}$

Eigenschaft 1., 3. und 5. (siehe Zeichnung) sind nicht verletzt.

Annahme:  $z' = \text{root}[T]$

- $\text{color}[z'] = \text{ROT} \Rightarrow$  Eigenschaft 2 ist verletzt

- $\text{color}[p[z']] = \text{SCHWARZ} \Rightarrow$  Eigenschaft 4 ist nicht verletzt

Annahme:  $z' \neq \text{root}[T]$

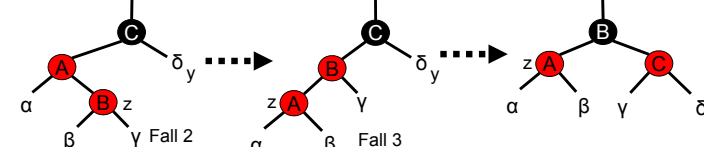
- $\text{color}[\text{root}[T]]$  hat sich nicht geändert  $\Rightarrow$  Eigenschaft 2 ist nicht verletzt.

- Eigenschaft 4 an Knotenpaar  $(z, p[z])$  erfüllt; falls  $\text{color}[p[z']] = \text{ROT}$ , liegt eine Verletzung von Eigenschaft 4 an Knotenpaar  $(z', p[z'])$  vor.



## Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Fall 2:**  $\text{color}[y] = \text{SCHWARZ}$  und  $z = \text{right}[p[z]]$



- Alle Nachfolger  $\alpha-\gamma$  von  $z, p[z]$  (außer  $z$ ) und  $\delta$  sind schwarz und haben die gleiche Schwarz-Höhe  $bh()$ .
- Überführung in Fall 3 durch Links-Rotation (Zeilen 10-11), Eigenschaft 5 bleibt erhalten.

- **Fall 3:**  $\text{color}[y] = \text{SCHWARZ}$  und  $z = \text{left}[p[z]]$

1.  $\text{color}[z] = \text{ROT}$

nach Fall 2 gilt  $z' = p[z]$  und  $\text{color}[p[z]] = \text{ROT}$

in Fall 3 werden weder  $z$  noch  $\text{color}[z]$  verändert.

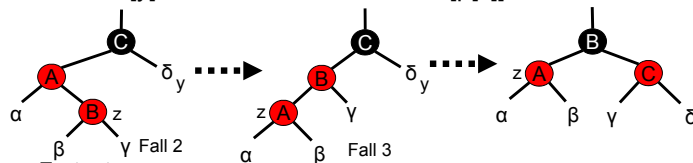
2. Falls  $p[z] = \text{root}[T]$ , gilt:  $\text{color}[p[z]] = \text{SCHWARZ}$

nach Fall 3 (Zeile 12) gilt  $\text{color}[p[z]] = \text{SCHWARZ}$ .



## Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Fall 3:**  $\text{color}[y] = \text{SCHWARZ}$  und  $z = \text{left}[p[z]]$



- Fortsetzung:

3. Es gibt maximal eine Verletzung der Rot-Schwarz-Eigenschaft:

1. Eigenschaft 2  $\Rightarrow z = \text{root}[T]$  und  $\text{color}[z] = \text{ROT}$
2. Eigenschaft 4  $\Rightarrow \text{color}[z] = \text{ROT}$  und  $\text{color}[p[z]] = \text{ROT}$

Eigenschaft 1., 3. und 5. (siehe Zeichnung) sind nicht verletzt.

- ◆ Eigenschaft 2 ist nicht verletzt, da  $\text{color}[p[z']] = \text{SCHWARZ}$

- ◆ Eigenschaft 4 wird für das Knotenpaar  $(z, p[z])$  korrigiert. Da  $\text{color}[p[z]] = \text{SCHWARZ}$ , gibt es keine weitere Verletzung von Eigenschaft 4.

- **Theorem 8:** RB-INSERT-NODE() fügt einen Knoten unter Erhalt der Rot-Schwarz-Eigenschaften in einen Rot-Schwarz-Baum in Zeit  $O(\log N)$  ein.

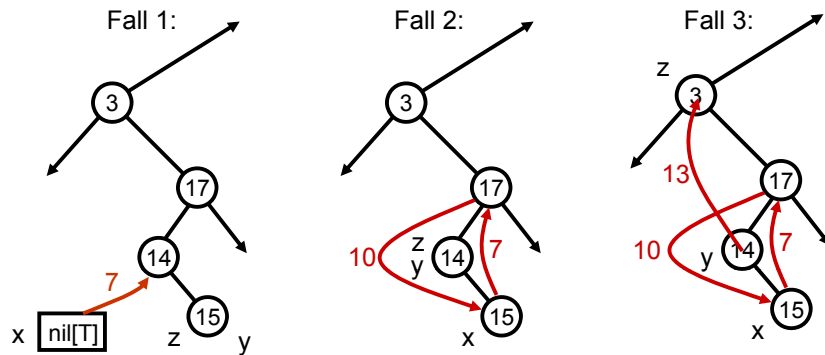


## Löschen aus Rot-Schwarz-Bäumen

- RB-DELETE-NODE( $T, z$ ) // Cormen: RB-DELETE()
- //  $y$ : der (nach Tausch) zu löschende Knoten
- 1 if  $\text{left}[z] = \text{nil}[T]$  or  $\text{right}[z] = \text{nil}[T]$  then  $y \leftarrow z$
- 3 else  $y \leftarrow \text{TREE-MINIMUM}(\text{right}[z])$  // Cormen: TREE-SUCCESSOR( $z$ )
- //  $x$ : Kind von  $y$ , das nicht NIL ist
- 4 if  $\text{left}[y] \neq \text{nil}[T]$  then  $x \leftarrow \text{left}[y]$  else  $x \leftarrow \text{right}[y]$
- // entferne  $y$  aus der Baumstruktur
- 7 if  $x \neq \text{NIL}$  then  $p[x] \leftarrow p[y]$  // if-Abfrage wg. Wächter unnötig
- 8 if  $p[y] = \text{NIL}$  then  $\text{root}[T] \leftarrow x$
- 10 else if  $y = \text{left}[p[y]]$  then  $\text{left}[p[y]] \leftarrow x$
- 12 else  $\text{right}[p[y]] \leftarrow x$
- // kopiere Daten von  $y$  nach  $z$
- 13 if  $y \neq z$  then  $\text{key}[z] \leftarrow \text{key}[y]$
- 16 if  $\text{color}[y] = \text{SCHWARZ}$  then RB-DELETE-FIXUP( $T, x$ )
- 18 delete  $y$



## Löschen aus Rot-Schwarz-Bäumen



### Korrektheit von RB-DELETE-NODE() für den Fall $\text{color}[y] = \text{ROT}$ :

- Die Schwarz-Höhen ändern sich nicht.
- Es entstehen keine benachbarten roten Knoten.
- $y \neq \text{root}[T]$ , da  $\text{color}[\text{root}[T]] = \text{SCHWARZ}$



## Die Funktion RB-DELETE-FIXUP()

### Welche Rot-Schwarz-Eigenschaften können verletzt sein:

- Jeder Knoten ist entweder rot oder schwarz. ✓
- Die Wurzel ist schwarz. -
- Jedes Blatt (der Wächter) ist schwarz. ✓
- Für jeden roten Knoten gilt: beide Nachfolger sind schwarz. -
- Für jeden Knoten  $x$  gilt: Alle Pfade von  $x$  zu einem Blatt enthalten die gleiche Anzahl  $bh(x)$  schwarzer Knoten. -

- zu 2.: verletzt, falls  $y = \text{root}[T]$  und  $\text{color}[x] = \text{ROT}$
- zu 4.: verletzt, falls  $\text{color}[p[y]] = \text{ROT}$  und  $\text{color}[x] = \text{ROT}$
- zu 5.: verletzt für alle Knoten auf den Pfad von  $p[x]$  bis zur Wurzel  
Alternative Sicht:  $\text{color}[x] = \text{'doppelt-schwarz'}$  oder  $\text{'rot-schwarz'}$   
Dann ist 5. erfüllt und dafür 1. (nur an Knoten  $x$ ) verletzt.



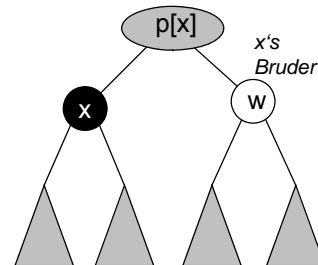
## Die Funktion RB-DELETE-FIXUP()

### RB-DELETE-FIXUP( $T, x$ )

```

1 while  $x \neq \text{root}[T]$  and  $\text{color}[x] = \text{SCHWARZ}$  do
2   if  $x = \text{left}[p[x]]$  then
3      $w \leftarrow \text{right}[p[x]]$  // w zeigt auf den Bruder von x
4     if  $\text{color}[w] = \text{ROT}$  then
5        $\text{color}[w] \leftarrow \text{SCHWARZ}$ 
6        $\text{color}[p[x]] \leftarrow \text{ROT}$ 
7       LEFT-ROTATE( $T, p[x]$ )
8        $w \leftarrow \text{right}[p[x]]$ 
9     if  $\text{color}[\text{left}[w]] = \text{SCHWARZ}$  and
        $\text{color}[\text{right}[w]] = \text{SCHWARZ}$  then
10       $\text{color}[w] \leftarrow \text{ROT}$ 
11       $x \leftarrow p[x]$ 
12    else if  $\text{color}[\text{right}[w]] = \text{SCHWARZ}$  then
13       $\text{color}[\text{left}[w]] \leftarrow \text{SCHWARZ}$ 
14       $\text{color}[w] \leftarrow \text{ROT}$ 
15      RIGHT-ROTATE( $T, w$ )
16       $w \leftarrow \text{right}[p[x]]$ 
17       $\text{color}[w] \leftarrow \text{color}[p[x]]$ 
18       $\text{color}[p[x]] \leftarrow \text{SCHWARZ}$ 
19       $\text{color}[\text{right}[w]] \leftarrow \text{SCHWARZ}$ 
20      LEFT-ROTATE( $T, p[x]$ )
21       $x \leftarrow \text{root}[T]$ 
22    else //  $x = \text{right}[p[x]]$  // wie then-Teil, vertausche right  $\leftrightarrow$  left
23       $\text{color}[x] \leftarrow \text{SCHWARZ}$ 

```



Fall 5-8 (symmetrisch zu 1-4)

Fall 1

F.2

Fall 3

Fall 4



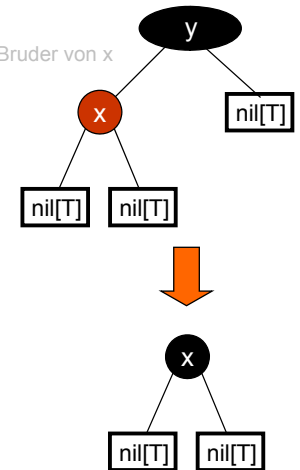
## RB-DELETE-FIXUP() – Korrektur von Eigenschaft 2

### RB-DELETE-FIXUP( $T, x$ )

```

1 while  $x \neq \text{root}[T]$  and  $\text{color}[x] = \text{SCHWARZ}$  do
2   if  $x = \text{left}[p[x]]$  then
3      $w \leftarrow \text{right}[p[x]]$  // w zeigt auf den Bruder von x
4     if  $\text{color}[w] = \text{ROT}$  then
5        $\text{color}[w] \leftarrow \text{SCHWARZ}$ 
6        $\text{color}[p[x]] \leftarrow \text{ROT}$ 
7       LEFT-ROTATE( $T, p[x]$ )
8        $w \leftarrow \text{right}[p[x]]$ 
9     if  $\text{color}[\text{left}[w]] = \text{SCHWARZ}$  and
        $\text{color}[\text{right}[w]] = \text{SCHWARZ}$  then
10       $\text{color}[w] \leftarrow \text{ROT}$ 
11       $x \leftarrow p[x]$ 
12    else if  $\text{color}[\text{right}[w]] = \text{SCHWARZ}$  then
13       $\text{color}[\text{left}[w]] \leftarrow \text{SCHWARZ}$ 
14       $\text{color}[w] \leftarrow \text{ROT}$ 
15      RIGHT-ROTATE( $T, w$ )
16       $w \leftarrow \text{right}[p[x]]$ 
17       $\text{color}[w] \leftarrow \text{color}[p[x]]$ 
18       $\text{color}[p[x]] \leftarrow \text{SCHWARZ}$ 
19       $\text{color}[\text{right}[w]] \leftarrow \text{SCHWARZ}$ 
20      LEFT-ROTATE( $T, p[x]$ )
21       $x \leftarrow \text{root}[T]$ 
22    else //  $x = \text{right}[p[x]]$  // wie then-Teil, vertausche right  $\leftrightarrow$  left
23       $\text{color}[x] \leftarrow \text{SCHWARZ}$ 

```



Fall 5-8 (symmetrisch zu 1-4)

Fall 1

F.2

Fall 3

Fall 4



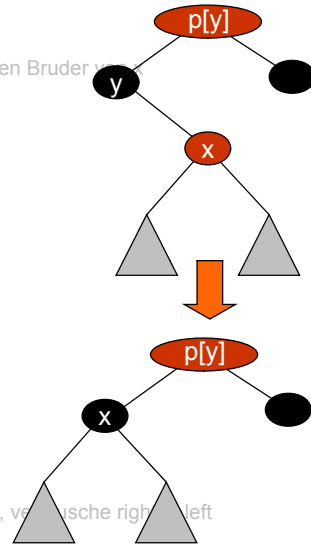
## RB-DELETE-FIXUP() – Korrektur von Eigenschaft 4

### RB-DELETE-FIXUP(T, x)

```

1 while x ≠ root[T] and color[x] = SCHWARZ do
2   if x = left[p[x]] then
3     w ← right[p[x]]           // w zeigt auf den Bruder von x
4     if color[w] = ROT then
5       color[w] ← SCHWARZ
6       color[p[x]] ← ROT
7       LEFT-ROTATE(T, p[x])
8       w ← right[p[x]]
9     if color[left[w]] = SCHWARZ and
       color[right[w]] = SCHWARZ then
10      color[w] ← ROT
11      x ← p[x]
12    else if color[right[w]] = SCHWARZ then
13      color[left[w]] ← SCHWARZ
14      color[w] ← ROT
15      RIGHT-ROTATE(T, w)
16      w ← right[p[x]]
17      color[w] ← color[p[x]]
18      color[p[x]] ← SCHWARZ
19      color[right[w]] ← SCHWARZ
20      LEFT-ROTATE(T, p[x])
21      x ← root[T]
22    else // x = right[p[x]]    // wie then-Teil, vertausche right ↔ left
23      color[x] ← SCHWARZ

```



Fall 5-8 (symmetrisch zu 1-4)

Fall 1  
F.2  
Fall 3  
Fall 4



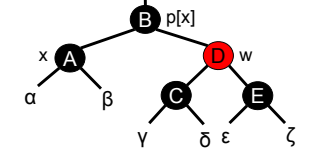
## RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 1

### RB-DELETE-FIXUP(T, x)

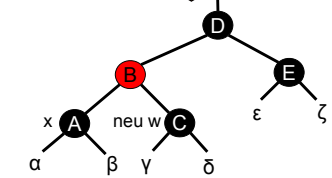
```

1 while x ≠ root[T] and color[x] = SCHWARZ do
2   if x = left[p[x]] then
3     w ← right[p[x]]           // w zeigt auf den Bruder von x
4     if color[w] = ROT then
5       color[w] ← SCHWARZ
6       color[p[x]] ← ROT
7       LEFT-ROTATE(T, p[x])
8       w ← right[p[x]]
9     if color[left[w]] = SCHWARZ and
       color[right[w]] = SCHWARZ then
10      color[w] ← ROT
11      x ← p[x]
12    else if color[right[w]] = SCHWARZ then
13      color[left[w]] ← SCHWARZ
14      color[w] ← ROT
15      RIGHT-ROTATE(T, w)
16      w ← right[p[x]]
17      color[w] ← color[p[x]]
18      color[p[x]] ← SCHWARZ
19      color[right[w]] ← SCHWARZ
20      LEFT-ROTATE(T, p[x])
21      x ← root[T]
22    else // x = right[p[x]]    // wie then-Teil, vertausche right ↔ left
23      color[x] ← SCHWARZ

```



Überführung in Fall 2-4



Fall 5-8 (symmetrisch zu 1-4)

Fall 1  
F.2  
Fall 3  
Fall 4



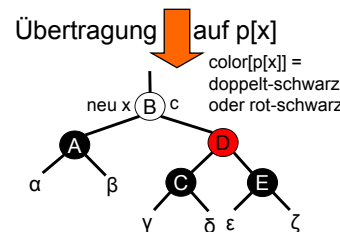
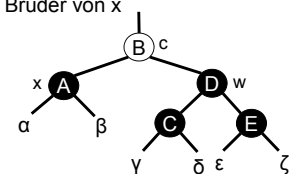
## RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 2

### RB-DELETE-FIXUP(T, x)

```

1 while x ≠ root[T] and color[x] = SCHWARZ do
2   if x = left[p[x]] then
3     w ← right[p[x]]           // w zeigt auf den Bruder von x
4     if color[w] = ROT then
5       color[w] ← SCHWARZ
6       color[p[x]] ← ROT
7       LEFT-ROTATE(T, p[x])
8       w ← right[p[x]]
9     if color[left[w]] = SCHWARZ and
       color[right[w]] = SCHWARZ then
10      color[w] ← ROT
11      x ← p[x]
12    else if color[right[w]] = SCHWARZ then
13      color[left[w]] ← SCHWARZ
14      color[w] ← ROT
15      RIGHT-ROTATE(T, w)
16      w ← right[p[x]]
17      color[w] ← color[p[x]]
18      color[p[x]] ← SCHWARZ
19      color[right[w]] ← SCHWARZ
20      LEFT-ROTATE(T, p[x])
21      x ← root[T]
22    else // x = right[p[x]]    // wie then-Teil, vertausche right ↔ left
23      color[x] ← SCHWARZ

```



Fall 5-8 (symmetrisch zu 1-4)

Fall 1  
F.2  
Fall 3  
Fall 4



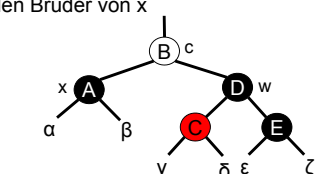
## RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 3

### RB-DELETE-FIXUP(T, x)

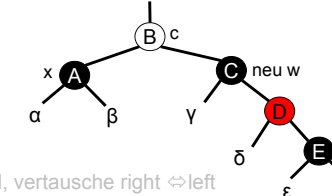
```

1 while x ≠ root[T] and color[x] = SCHWARZ do
2   if x = left[p[x]] then
3     w ← right[p[x]]           // w zeigt auf den Bruder von x
4     if color[w] = ROT then
5       color[w] ← SCHWARZ
6       color[p[x]] ← ROT
7       LEFT-ROTATE(T, p[x])
8       w ← right[p[x]]
9     if color[left[w]] = SCHWARZ and
       color[right[w]] = SCHWARZ then
10      color[w] ← ROT
11      x ← p[x]
12    else if color[right[w]] = SCHWARZ then
13      color[left[w]] ← SCHWARZ
14      color[w] ← ROT
15      RIGHT-ROTATE(T, w)
16      w ← right[p[x]]
17      color[w] ← color[p[x]]
18      color[p[x]] ← SCHWARZ
19      color[right[w]] ← SCHWARZ
20      LEFT-ROTATE(T, p[x])
21      x ← root[T]
22    else // x = right[p[x]]    // wie then-Teil, vertausche right ↔ left
23      color[x] ← SCHWARZ

```



Überführung durch Rotation in Fall 4



Fall 5-8 (symmetrisch zu 1-4)

Fall 1  
F.2  
Fall 3  
Fall 4



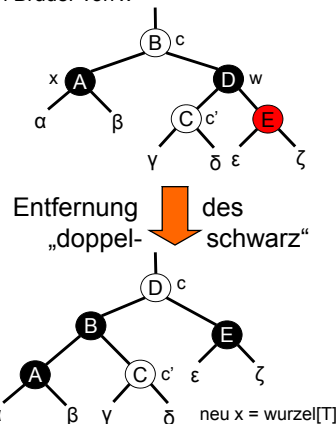
## RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 4

### ■ RB-DELETE-FIXUP(T, x)

```

1 while x ≠ root[T] and color[x] = SCHWARZ do
2   if x = left[p[x]] then
3     w ← right[p[x]]           // w zeigt auf den Bruder von x
4     if color[w] = ROT then
5       color[w] ← SCHWARZ
6       color[p[x]] ← ROT
7       LEFT-ROTATE(T, p[x])
8       w ← right[p[x]]
9     if color[left[w]] = SCHWARZ and
       color[right[w]] = SCHWARZ then
10      color[w] ← ROT
11      x ← p[x]
12     else if color[right[w]] = SCHWARZ then
13       color[left[w]] ← SCHWARZ
14       color[w] ← ROT
15       RIGHT-ROTATE(T, w)
16       w ← right[p[x]]
17     color[w] ← color[p[x]]
18     color[p[x]] ← SCHWARZ
19     color[right[w]] ← SCHWARZ
20     LEFT-ROTATE(T, p[x])
21     x ← root[T]
22   else // x = right[p[x]]
23     color[x] ← SCHWARZ

```



Fall 5-8 (symmetrisch zu 1-4)



## RB-DELETE-NODE() - Komplexität

■ **Theorem 8:** RB-DELETE-NODE() löscht einen Knoten unter Erhalt der Rot-Schwarz-Eigenschaften in einen Rot-Schwarz-Baum in Zeit  $O(\log N)$ .

- Die Korrektheit von RB-DELETE-NODE() folgt aus der Diskussion der Fälle.
- Die Bearbeitung aller Fälle erfolgt in konstanter Zeit.
- Nach Fall 1, 3 und 4 terminiert die Schleife
- In Fall 2 wird die Schleife auf dem Elter-Knoten ausgeführt.

■ Insgesamt folgt: Ein Rot-Schwarz-Baum realisiert ein Wörterbuch mit asymptotischer Worst-Case Laufzeit  $O(\log N)$  für die Operationen INSERT, DELETE und SEARCH.



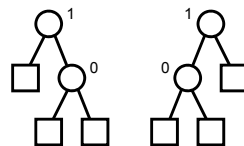
## 4.3 AVL-Bäume

[aus Ottmann/Widmeyer, Spektrum Akad. Verlag, 2002]

- AVL-Baum: (Adel'son, Velskii und Landis, 1962)
- Für jeden Knoten p gilt:
  - Sei  $bal(p) = \text{Höhe}(\text{rechten Teilbaum}) - \text{Höhe}(\text{linker Teilbaum})$
  - Es gilt  $bal(p) \in \{-1, 0, 1\}$
- Ist ein AVL-Baum balanciert?
  - ein AVL-Baum der Höhe h hat mindestens  $F_{h+2}$  Blätter ( $F_n$ : h-te Fibonacci-Zahl;  $F_0 = 0, F_1 = 1, F_{i+2} = F_i + F_{i+1}$ )



Höhe 1

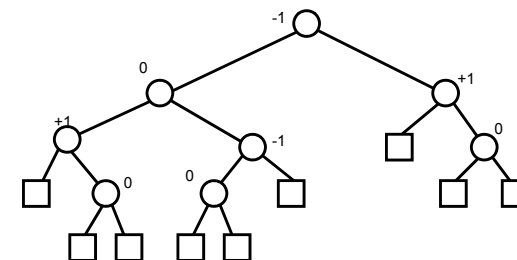


Höhe 2, min. Anzahl Blätter

- Es gilt:  $F_h \approx 0.72 \cdot 1.62^h$
- $N = 2 \cdot \text{Anzahl der Blätter} - 1 \Rightarrow N_h \geq 2 \cdot F_{h+2} - 1$
- $h = O(\log N)$



## Einfügen in AVL-Bäume 1

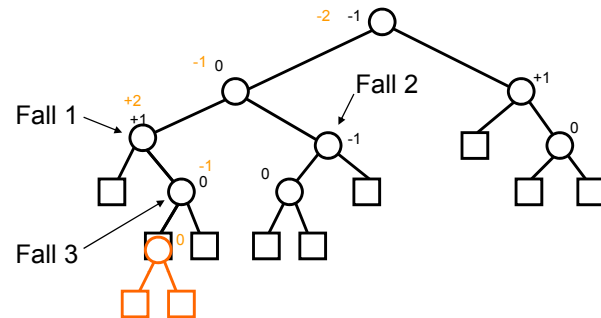


■ Einfügen in AVL-Bäume:

- Teil 1: Einfügen wie in natürlichen Bäumen (Einfügestelle p)  
Balancewerte auf dem Pfad von p zur Wurzel ändern sich
- Teil 2: Wandere von p zur Wurzel und rebalanciere durch Rotationen



## Einfügen in AVL-Bäume 2



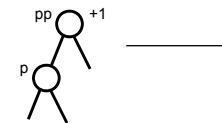
Einfügen unter Knoten p:

- Fall 1:  $\text{bal}(p) = +1$ : füge x als linken Sohn ein,  $\text{bal}(p) = 0$
  - Fall 2:  $\text{bal}(p) = -1$ : füge x als rechten Sohn ein,  $\text{bal}(p) = 0$
  - Fall 3.1:  $\text{bal}(p) = 0$  und  $x > p.\text{key}$ : füge x als rechten Sohn ein,  $\text{bal}(p) = +1$
  - Fall 3.2:  $\text{bal}(p) = 0$  und  $x < p.\text{key}$ : füge x als linken Sohn ein,  $\text{bal}(p) = -1$
- Im Fall 3 verändern sich die Balancewerte von p zur Wurzel!



## Balancieren nach dem Einfügen: upin-Prozedur

- $\text{upin}(p : \text{Zeiger auf AVL\_node})$ 
  - Vorbedingung:  $\text{bal}(p) \in \{-1, 1\}$ ,
  - Höhe des Teilbaums unter p ist um 1 gewachsen
  - $\text{upin}(p)$  balanciert die Teilbäume unter  $\text{parent}(p)$  und geht dann rekursiv zum Vater
  - Es gibt 6 verschiedene Fälle (sei  $pp = \text{parent}(p)$ ):
    - ◆  $pp.\text{left} = p$  (p ist linker Sohn) und  $\text{bal}(pp) = +1, 0$  oder  $-1$  Fall 1.1-1.3
    - ◆  $pp.\text{right} = p$  (p ist rechter Sohn) und  $\text{bal}(pp) = +1, 0, -1$  Fall 2.1-2.3
  - Fall 1.1:  $pp.\text{left} = p$  und  $\text{bal}(pp) = +1$

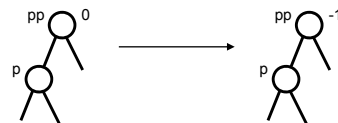


- $\text{bal}(pp) \leftarrow 0$
- Höhe von pp bleibt gleich



## Balancieren nach dem Einfügen: upin-Prozedur

- Fall 1.2:  $pp.\text{left} = p$  und  $\text{bal}(pp) = 0$

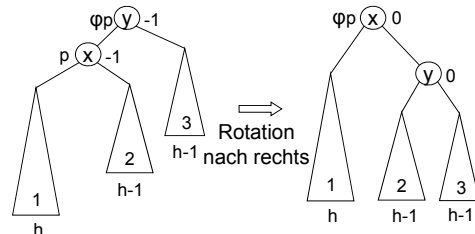


- $\text{bal}(pp) \leftarrow -1$
- Höhe von pp wächst um 1
- $\rightarrow$  führe  $\text{upin}(pp)$  aus

- Fall 1.3:  $pp.\text{left} = p$  und  $\text{bal}(pp) = -1$

- $\text{bal}(pp) = -2$ , Anpassung der Baumstruktur notwendig

- Fall 1.3.1:  $\text{bal}(p) = -1$

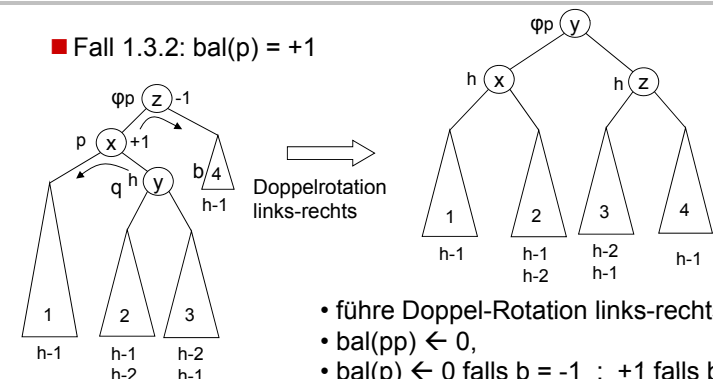


- führe Rechts-Rotation durch
- $\text{bal}(pp) \leftarrow 0$ ,  $\text{bal}(p) \leftarrow 0$
- Höhe von pp bleibt gleich



## Balancieren nach dem Einfügen: upin-Prozedur

- Fall 1.3.2:  $\text{bal}(p) = +1$



- führe Doppel-Rotation links-rechts aus
- $\text{bal}(pp) \leftarrow 0$ ,
- $\text{bal}(p) \leftarrow 0$  falls  $b = -1$ ;  $+1$  falls  $b = +1$
- $\text{bal}(q) \leftarrow -1$  falls  $b = -1$ ;  $0$  falls  $b = +1$
- Höhe von pp bleibt gleich

- Fälle 2.1 – 2.3 sind spiegelsymmetrisch analog

- Löschen in AVL-Bäumen: ähnliche Überlegung, Funktion  $\text{upout}$  zur Höhenbalancierung





## Abschließende Kommentare zu Suchbäumen

- Suchbäume eignen sich als **dynamische Datenstruktur für Wörterbücher** mit Operationen INSERT, DELETE und SEARCH.
- Die Laufzeit aller Operationen sind abhängig von der Höhe des Suchbaums. Im Average Case verhält sich die Höhe logarithmisch zur Anzahl gespeicherter Objekte.
- Ein Suchbaum hat im Worst Case die Höhe  $\Theta(N)$ , im Best Case und Average Case die Höhe  $O(\log N)$ . Ein Suchbaum mit Höhe  $O(\log N)$  heißt **balanciert**.
- **Rot-Schwarz-Bäume** stellen einen Mechanismus zum Balancieren der Suchbäume nach INSERT- und DELETE-Operationen zur Verfügung. Beide Operationen haben im Worst Case Laufzeit  $O(\log N)$ .
- **AVL-Bäume** verwenden ein alternatives Balance-Kriterium. INSERT- und DELETE-Operationen erzeugen ebenfalls in  $O(\log N)$  Zeit balancierte Suchbäume.



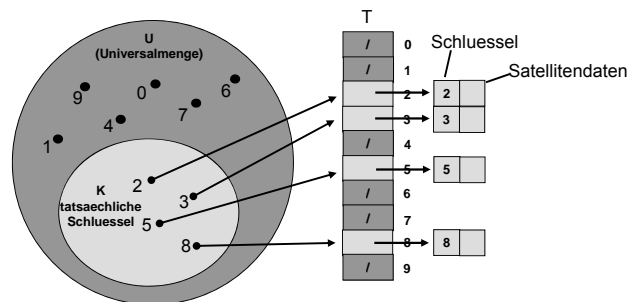
## 4.4 Hashing mit Verkettung

- Bisherige Voraussetzung:
  - Schlüssel mit Ordnungsrelation  $\leq$  : Suchbäume mit  $O(\log N)$ -Laufzeiten für Einfügen, Löschen und Suchen
- Neue Annahme:
  - Schlüssel lassen sich auf natürliche Zahlen abbilden.
- Idee (Hashing):
  - Berechne aus dem Schlüssel einen Index  $h(\text{key})$ , speichere die Daten in einem Array an Position  $h(\text{key})$ . Im RAM-Modell erfolgt der Array-Zugriff in konstanter Zeit.
- Probleme:
  - Mehrere Schlüssel werden auf den gleichen Index abgebildet → Kollision
  - Der Index-Raum ist viel größer als die zu speichernden Daten



## Direkte Adressierung

- Annahme:
  - Schlüssel sind aus der Menge  $U = \{0, \dots, m-1\}$  (*Universalmenge*).
  - Alle Schlüssel sind voneinander verschieden.
  - $m = O(|K|)$ , wobei  $K$  die Menge der verwendeten Schlüssel ist.
- Idee: Adresstabelle mit direktem Zugriff:



- Operationen INSERT, DELETE, SEARCH sind in  $O(1)$  realisierbar.



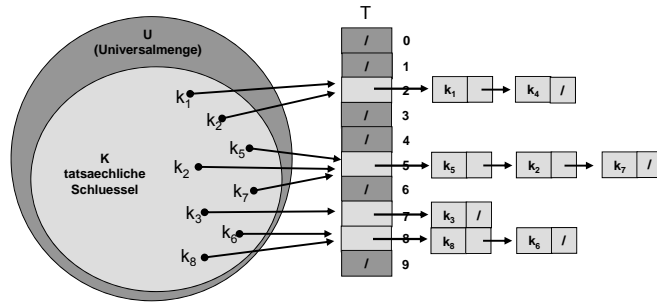
## Hashing

- In der Praxis:
  - $|U| \gg |K|$ , d.h. direkte Adressierung ist speicher-ineffizient.
- Ziel: Speicherung mit Platzbedarf  $\Theta(|K|)$  unter Erhalt der Laufzeit
- **Hashfunktion**:
  - Funktion zur Abbildung der Universalmenge auf die Menge verfügbarer Speicheradressen:  $h: U \rightarrow \{0, \dots, m-1\}$ .  $h(k)$  ist der **Hashwert** des Schlüssels  $k$ . Die Daten werden in einer **Hashtabelle**  $T[0, \dots, m-1]$  gespeichert.
- **Kollisionen**:
  - Gilt für zwei Schlüssel  $k \neq k' : h(k) = h(k')$ , spricht man von einer **Kollision**.
  - Da  $|U| > m$  gilt, sind Kollisionen nicht vermeidbar.
- Entwicklungsziele:
  - Handhabung von Kollisionen
  - Minimierung der auftretenden Kollisionen durch Wahl einer geeigneten Hashfunktion



## Kollisionsauflösung durch Verkettung

- Idee:
  - Speicherung aller Elemente mit gleichem Hashwert in einer linearen Liste.
- Operationen:
  - INSERT(T,x): füge x an den Kopf der Liste T[h(key[x])] ein  $O(1)$   
(ohne Test auf Vorhandensein von x)
  - DELETE(T,x): entferne x aus der Liste T[h(key[x])]  $O(1)$   
(dies setzt eine doppelte Verkettung voraus)
  - SEARCH(T,k): suche in der Liste T[h(k)]  $O(|T[h(key[x])|])$



## Analyse von Hashing mit Verkettung

- **Belegungsfaktor:**  $\alpha = n / m$ 
  - $n = |K|$ : Anzahl der in der Hashtabelle gespeicherter Objekte
  - $m = |T|$ : Die Größe der Hashtabelle
  - beschreibt die mittlere Anzahl Objekte pro Eintrag in der Hashtabelle
- Annahme: **einfaches, gleichmäßiges Hashing**  
Sei  $X_{ij}$  das Ereignis, dass Objekt i den Hashwert j erhält. Dann sei  $\Pr\{X_{ij}=1\} = 1/m$  und  $X_{ij}$  und  $X_{ik}$  sind stochastisch unabhängig.
- Lemma: Sei  $n_j = \text{length}[T[j]]$ , dann gilt  $E[n_j] = \alpha = n / m$
- **Theorem 9:** In einer Hashtabelle mit Verkettung unter Annahme des einfachen, gleichmäßigen Hashings benötigt die *erfolgreiche* Suche  $O(1 + \alpha)$  erwartete Laufzeit.
  - Ein nicht enthaltener Schlüssel k wird im Wahrscheinlichkeit  $1/m$  in Liste  $h(k)$  abgelegt. Es gilt  $E[n_{h(k)}] = \alpha$ .
  - Bei erfolgloser Suche wird die Liste  $T[h(k)]$  in erwarteter Zeit  $O(\alpha)$  vollständig durchlaufen, die Berechnung von  $h(k)$  erfolgt in  $O(1)$ .



## Analyse von Hashing mit Verkettung

- **Theorem 10:** In einer Hashtabelle mit Verkettung unter Annahme des einfachen, gleichmäßigen Hashings benötigt die *erfolgreiche* Suche  $O(1 + \alpha)$  erwartete Laufzeit.
  - Sei  $X = x_1, \dots, x_n$  die Folge, in der die Objekte mit Schlüssel  $k_1, \dots, k_n$  in die Hashtabelle eingefügt wurden. Sei  $X_{ij}$  das Kollisions-Ereignis  $h(k_i) = h(k_j)$ .
  - SEARCH(T,  $x_i$ ) durchläuft alle Objekte  $x_j$ , die nach  $x_i$  eingefügt wurden und in der gleichen Liste stehen, also mit  $X_{ij}=1$ . Dann folgt:

$$\begin{aligned}
 E[T_{\text{search}}(n)] &= E\left[\frac{c}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij}\right)\right] \\
 &= \frac{c}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}]\right) && \text{wg. Linearität des Erwartungswertes} \\
 &= \frac{c}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) && \text{wg. einfachem gleichmäßigen Hashing} \\
 &= c + \frac{c}{nm} \sum_{i=1}^n \left(\sum_{j=i+1}^n 1\right) && \text{wg. Gauß'scher Summenformel} \\
 &= c + \frac{c}{nm} \sum_{i=1}^n (n-i) = c + \frac{c}{nm} \left(n^2 - \frac{n(n+1)}{2}\right) = c \left(1 + \frac{n-1}{2m}\right) = c \left(1 + \frac{\alpha}{2} - \frac{\alpha}{2n}\right) = O(1 + \alpha)
 \end{aligned}$$



## Hashfunktionen

- Anforderungen an Hashfunktionen:
  - leicht und schnell berechenbar
  - gleichmäßige Verteilung der Schlüsselmenge auf die Plätze  $0, \dots, m-1$
  - Achtung: Häufung hängt von der Anwendung ab, d.h.
    - ◆ Vermeidung anwendungsverursachter Häufungen
- Überprüfung der Qualität von Hashfunktionen:
  - Es gibt viele mögliche Hashfunktionen, zur sinnvollen Auswahl muss man etwas über die Verteilung von K wissen.
- Heuristische Wahl:
  - Hashwerte sollten keine Häufungen für erwartete Muster zeigen.
- Annahme: Im folgenden gelte:
  - Schlüsselwerte k sind nicht-negative, ganze Zahlen, also  $k \in \mathbb{N}_0$
  - die meisten Schlüssel lassen sich sinnvoll in Zahlen aus  $\mathbb{N}_0$  konvertieren
    - ◆ Universalität des binären Codes
    - ◆ Auch bei dieser Konvertierung kann bereits Häufung eine Rolle spielen.





## Hashfunktionen: Divisions- und Multiplikationsmethode

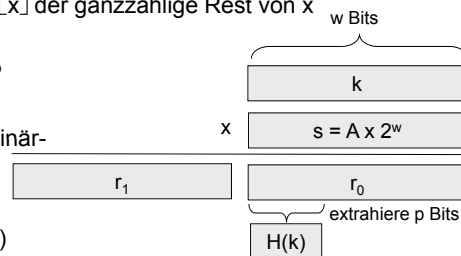
### ■ Divisionsmethode $h: U \rightarrow \{0, \dots, m-1\} : h(k) = k \bmod m$

- ◆ m gerade: k gerade  $\Rightarrow h(k)$  ist gerade
  - ◆  $m = 2^p$ : nur die p-letzten Bits werden zur Generierung von  $h(k)$  herangezogen
  - ◆ Im allgemeinen sollte m möglichst wenig Teiler haben (Primzahl!)
- ABER: Wenn die Schlüsselmenge K in U gleichverteilt ist, dann ergibt auch  $m = 2^p$  keine Häufung. Operation „mod  $2^p$ “ ist wesentlich effizienter als „mod  $m$ “ mit einer Primzahl m!

### ■ Multiplikationsmethode $h: U \rightarrow \{0, \dots, m-1\} : h(k) = \lfloor m(kA \bmod 1) \rfloor$

- Definition: Es sei  $x \bmod 1 = x - \lfloor x \rfloor$  der ganzzahlige Rest von x

- ◆ m kann unabhängig von A gewählt werden, z.B.  $m = 2^p$
- ◆ mit  $A = s/2^w$  (w = Wortlänge des Computers) kann die Binärarithmetik des Prozessors optimal ausgenutzt werden
- ◆  $A \approx (\sqrt{5} - 1)/2$  (D. Knuth)



## Universelles Hashing

- Für jede Hashfunktion gibt es eine Menge von n Schlüsseln, die auf den gleichen Platz abgebildet werden (vorausgesetzt  $U > nm$ ).
- Wie können wir im Mittel eine gute Performanz erhalten?
- **Universelles Hashing:** Zufällige Auswahl einer Hashfunktion unabhängig von der Schlüsselmenge
- Definition: Eine Menge  $H = \{h: U \rightarrow \{0, \dots, m-1\}\}$  von Hashfunktionen heißt **universell**, g.d.w.  $\forall k, l \in U : |\{h \in H \mid h(k) = h(l)\}| \leq |H|/m$
- Bei zufälliger Wahl einer Hashfunktion und zweier Zahlen  $k, l \in U$  ist die Wahrscheinlichkeit der Kollision  $h(k) = h(l) \leq 1/m$ .



## Universelles Hashing

- **Theorem 11:** Sei  $h \in H$  aus einer Menge universeller Hashfunktionen mit  $h: U \rightarrow \{0, \dots, m-1\}$ . Nach Speicherung von n Schlüsseln in Tabelle T (Hashing mit Verkettung) gilt für einen Schlüssel k die erwartete Listenlänge:
 
$$E[n_{h(k)}] \leq \begin{cases} \alpha & : k \in T \\ 1 + \alpha & : \text{sonst} \end{cases}$$

### ■ Beweis:

- Für ein Paar  $k, l \in U$  sei  $X_{kl}$  das Kollisionsereignis  $h(k) = h(l)$ .
- Da H universell ist, folgt  $\Pr\{h(k) = h(l)\} \leq 1/m$  und somit

$$E[X_{kl}] = \sum_{z=0}^1 z \Pr\{X_{kl} = z\} \leq 1/m$$

- Fall 1:  $k \notin T$

$$E[n_{h(k)}] = E\left[\sum_{l \in T} X_{kl}\right] = \sum_{l \in T} E[X_{kl}] \leq \sum_{l \in T} \frac{1}{m} = \frac{n}{m} = \alpha$$

- Fall 2:  $k \in T$

$$E[n_{h(k)}] = E\left[\sum_{l \in T} X_{kl}\right] = 1 + \sum_{l \in T \setminus \{k\}} E[X_{kl}] \leq 1 + \sum_{l \in T \setminus \{k\}} \frac{1}{m} = 1 + \frac{n-1}{m} < 1 + \alpha$$



## Universelles Hashing

- **Korollar:** Bei universellem Hashing mit Verkettung mit m Plätzen ist die erwartete Laufzeit von p INSERT-, SEARCH- und DELETE-Operationen mit  $O(m)$  INSERT-Operationen  $O(p)$ .
- Beweis:
  - Nach  $O(m)$  INSERT-Operationen gilt  $\alpha \leq \frac{cm}{m} = O(1)$
  - $T_{\text{INSERT}} = O(1)$ ,  $T_{\text{DELETE}} = O(1)$
  - SEARCH-Operation mit Schlüssel k:
    - ◆ Für die erwartete Listenlänge gilt  $E[n_{h(k)}] \leq \alpha$ , bzw.  $E[n_{h(k)}] \leq 1 + \alpha$
    - ◆ Somit folgt  $E[T_{\text{SEARCH}}] = O(1 + \alpha) = O(1)$
  - Für p Operationen ergibt sich somit eine erwartete Laufzeit von  $O(p)$
- Klasse von universellen Hashfunktionen (ohne Beweis):
 

Sei p eine Primzahl mit  $p > m$  und  $p > k \forall k \in U$ , sei  $Z_x = \{0, \dots, x-1\}$ . Dann ist  $H_{p,m}$  mit

$$h_{a,b}: U \rightarrow Z_m : h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

$$H_{p,m} = \{h_{a,b} : a \in Z_p \setminus \{0\} \text{ und } b \in Z_p\}$$

eine Klasse universeller Hashfunktionen mit  $|H_{p,m}| = p(p-1)$



## 4.5 Offene Adressierung

- Nachteile von Hashing mit Verkettung:
  - zusätzlicher Speicherbedarf für Zeiger
  - komplexe Speicherverwaltung (Belegen und Befreien von Listenelementen)
  - ineffiziente Speichernutzung bei Häufung
- Offene Adressierung
  - einmaliges Anlegen von Speicher für die Hashtabelle
  - für Überläufer wird ein anderer Platz in der Hashtabelle gesucht (eine offene Stelle)
  - dieser andere Platz wird berechnet, so dass Zeiger zur Verkettung nicht notwendig sind.
  - **Sondieren**: Sukzessives Überprüfen der Hashtabelle nach freien Plätzen
  - **Sondierungssequenz**: Folge von Hashadressen, die für einen Schlüssel  $k$  nach offenen Stellen durchsucht werden
    - ◆ Hashfunktion  $h: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$
    - ◆ **Sondierungszahl** (2. Parameter): gibt den Sondierungsversuch an



## Offene Adressierung: Einfügen

- Eigenschaften von Sondierungsfolgen
  - $\forall k \in U: h(k, j)$  ist eine bijektive Funktion
  - Im Laufe der Sondierung werden alle Speicherplätze getestet.
  - Sondierungssequenz hängt vom Schlüssel  $k$  ab.
- Eigenschaften der Hashtabelle
  - $T[0, \dots, m-1], T[j] \in \{NIL, DELETED\} \cup U$
  - keine Satellitendaten (diese können jedoch einfach integriert werden)
  - Initialisierung:  $T[i] = NIL \forall i \in \{0, \dots, m-1\}$
- HASH-INSERT-PRE( $T, k$ )
  - $i \leftarrow 0$
  - repeat**
    - $j \leftarrow h(k, i)$
    - if**  $T[j] = NIL$  **then**  $T[j] \leftarrow k$ ; **return**  $j$
    - else**  $i \leftarrow i+1$
  - until**  $i = m$
  - error** „Hashtable overflow!“



## Offene Adressierung: Suchen

- Suchalgorithmus:
  - Folge der Sondierungssequenz für  $k$  und vergleiche den Schlüssel
- HASH-SEARCH( $T, k$ )
  - $i \leftarrow 0$
  - repeat**
    - $j \leftarrow h(k, i)$
    - if**  $T[j] = k$  **then** **return**  $j$
    - else**  $i \leftarrow i+1$
  - until**  $T[j] = NIL$  **or**  $i = m$
  - return**  $NIL$
- Was passiert, wenn ein Schlüssel  $k$  gelöscht wird?
  - Schlüssel, die nach  $k$  gespeichert werden, können nicht wiedergefunden werden, da die Repeat-Schleife bei  $T[j]=NIL$  terminiert!



## Offene Adressierung: Einfügen und Löschen

- Löschverfahren:
  - Suche den Platz  $T[i]$  mit  $T[i] = k$  und setze  $T[i] = DELETED$
  - Suchen: behandle  $DELETED$  wie einen belegten Platz
  - Einfügen: behandle  $DELETED$  wie einen freien Platz
- HASH-INSERT( $T, k$ )
  - $i \leftarrow 0$
  - repeat**
    - $j \leftarrow h(k, i)$
    - if**  $T[j] \in \{NIL, DELETED\}$  **then**
      - $T[j] \leftarrow k$ ; **return**  $j$
    - else**  $i \leftarrow i+1$
  - until**  $i = m$
  - error** „Hashtable overflow!“
- HASH-DELETE( $T, k$ )
  - $i \leftarrow 0$
  - repeat**
    - $j \leftarrow h(k, i)$
    - if**  $T[j] = k$  **then**
      - $T[j] = DELETED$ ; **return**  $j$
    - else**  $i \leftarrow i+1$
  - until**  $T[j] = NIL$  **or**  $i = m$
  - return**  $NIL$
- ACHTUNG: Laufzeit von HASH-SEARCH() hängt nicht mehr vom Belegungsfaktor  $\alpha$  ab, sondern von  $|\{i \in \{0, \dots, m-1\} \mid T[i] \neq NIL\}|$   
=> In Anwendungen mit vielen DELETE-Operationen sollte Hashing mit Verkettung zur Kollisionsauflösung verwendet werden.



## Offene Adressierung: Ein Beispiel

### ■ Beispiel:

- Markierungen: N: NIL  
D: DELETED  
U: USED, d.h. ein Schlüssel ist dort gespeichert
- Hashfunktion:  $h(k,i) = (k + i) \bmod 7$

leere Hashtabelle

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

insert(15):

0	1	2	3	4	5	6
	15					
N	U	N	N	N	N	N

insert(36):

0	1	2	3	4	5	6
	15	36				
N	U	U	N	N	N	N

search(71):  $\rightarrow -1$

0	1	2	3	4	5	6
	15	36				
N	U	U	N	N	N	N



## Offene Adressierung: Ein Beispiel

insert(17):

0	1	2	3	4	5	6
	15	36	17			
N	U	U	U	N	N	N

insert(29):

0	1	2	3	4	5	6
	15	36	17	29		
N	U	U	U	U	N	N

search(36):  $\rightarrow 2$

0	1	2	3	4	5	6
	15	36	17	29		
N	U	U	U	U	N	N

delete(36):

0	1	2	3	4	5	6
	15	36	17	29		
N	U	D	U	U	N	N

search(29):  $\rightarrow 4$

0	1	2	3	4	5	6
	15	36	17	29		
N	U	D	U	U	N	N

insert(71):

0	1	2	3	4	5	6
	15	71	17	29		
N	U	U	U	U	N	N



## Offene Adressierung: Sondierungsfunktionen

- Hilfshashfunktion  $h': U \rightarrow \{0, \dots, m-1\}$
- Sondierungsfunktionen:
  - **Lineares Sondieren** (linear probing):  $h(k,i) = (h'(k) + i) \bmod m$ 
    - ◆ Von  $h'(k)$  aus wird jeweils der nächst größere Platz getestet.
    - ◆ Problem: Es entstehen lange Ketten besetzter Plätze:  
 $\Pr\{\text{Kette der Länge } i \rightarrow \text{Kette der Länge } i+1\} = (i+1)/m$
    - ◆ Bei insert ist die Wahrscheinlichkeit groß, lange Ketten zu verlängern  
 $\rightarrow$  „**primäres Clustern**“
  - **Quadratisches Sondieren** (quadratic probing):
    - ◆  $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$  mit  $c_2 \neq 0$
    - ◆  $m$ ,  $c_1$  und  $c_2$  müssen so gewählt werden, dass  $h(k,i)$  bzgl.  $i$  bijektiv ist.
    - ◆ Problem: Bei Kollisionen  $h(k) = h(k')$  haben beide Schlüssel die gleiche Sondierungsfolge  $\rightarrow$  „**sekundäres Clustern**“



## Offene Adressierung: Doppeltes Hashing

- Hilfshashfunktionen  $h_1: U \rightarrow \{0, \dots, m-1\}$ ,  $h_2: U \rightarrow \{1, \dots, m'\}$  mit  $m' < m$ 
  - Verwende  $h_1$ , um den Startpunkt der Sondierungsfolge zu bestimmen
  - Verwende  $h_2$ , um den Offset der Sondierungsfolge zu bestimmen
- **Doppeltes Hashing** (double hashing):
  - $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$ 
    - ◆  $h(k,i)$  ist bzgl.  $i$  nur bijektiv, falls  $h_2(k) \neq 0$  und  $h_2(k)$  teilerfremd zu  $m$  ist
    - ◆ Ideal:  $h_1$  und  $h_2$  sind bzgl. der Kollisionswahrscheinlichkeit stochastisch **unabhängig**:  
 $P(h_1(k)=h_1(k') \text{ und } h_2(k)=h_2(k')) = P(h_1(k)=h_1(k')) \cdot P(h_2(k)=h_2(k'))$
    - ◆ Es werden  $\Theta(m^2)$  verschiedene Sondierungssequenzen verwendet.
- Beispiele:
  1.  $m$  ist eine Primzahl,  $h_1(k) = k \bmod m$ ;  $h_2(k) = 1 + k \bmod (m-2)$
  2.  $m = 2^l$ ;  $h_1(k) = k \bmod m$ ;  $h_2(k) = 1 + 2k \bmod (m/2)$



## Offene Adressierung: Analyse

- Annahmen:
  - Komplexität in Abhängigkeit vom Belegungsfaktor  $\alpha$ ,  $\alpha < 1$
  - Gleichmäßiges Hashing: Alle Sondierungssequenzen  $h(k,i)$ ,  $0 \leq i < m$  sind gleich wahrscheinlich (Achtung: durch keine Sondierungsfunktion erfüllt!)
- Wie viele Sondierungsschritte sind im Falle einer erfolglosen / erfolgreichen Suche zu erwarten?
- **Theorem 12:** Für eine Hashtabelle mit offener Adressierung und Belegungsfaktor  $\alpha < 1$  ist die Anzahl der Sondierungen bei erfolgloser Suche unter der Annahme des gleichmäßigen Hashings höchstens  $1 / (1 - \alpha)$ .
- Beweis:
  - Jede Sondierung außer die letzte greift auf einen belegten Platz zu.
  - $X$ : Anzahl der durchgeführten Sondierungen
  - $A_i$ : Ereignis, dass es eine  $i$ -te Sondierung gibt, die auf einen belegten Platz zugreift.



## Offene Adressierung: Analyse

- Beweis Theorem 12 (Fortsetzung)

1. Hilfsgleichung:

$$\begin{aligned} E[X] &= \sum_{i=1}^m i \Pr\{X = i\} = \sum_{i=1}^m i (\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^m i \Pr\{X \geq i\} - \sum_{i=1}^m i \Pr\{X \geq i+1\} \\ &= \sum_{i=1}^m i \Pr\{X \geq i\} - \sum_{i=2}^{m+1} (i-1) \Pr\{X \geq i\} \\ &= \sum_{i=1}^m i \Pr\{X \geq i\} - \sum_{i=1}^m (i-1) \Pr\{X \geq i\} \\ &= \sum_{i=1}^m \Pr\{X \geq i\} \end{aligned}$$

siehe Cormen C.24

2. Hilfsgleichung:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad \text{für } |x| < 1$$

unendlich fallende geometrische Reihe, siehe Cormen A.6



## Offene Adressierung: Analyse

- Beweis Theorem 12 (Fortsetzung)

$$\begin{aligned} \Pr\{X \geq i\} &= \Pr\left\{\bigcap_{k=1}^{i-1} A_k\right\} \\ &= \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdots \Pr\left\{A_{i-1} | \bigcap_{k=1}^{i-2} A_k\right\} \\ &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1} \\ E[X] &= \sum_{i=1}^m i \Pr\{X = i\} = \sum_{i=1}^m \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} \end{aligned}$$



## Offene Adressierung: Analyse

- Interpretation:

- $E[X] \leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots$
- Eine Sondierung findet immer statt.
- Die Wahrscheinlichkeit für eine zweite Sondierung ist  $\alpha$
- Die Wahrscheinlichkeit für eine dritte Sondierung ist  $\alpha^2 \dots$

- Beispiele (erfolglose Suche):

- $\alpha = 0,5$ , d.h. die Hashtabelle ist halb voll:  $E[X] \leq 1/(1-0,5) = 2$
- $\alpha = 0,9$ , d.h. die Hashtabelle ist 90% voll:  $E[X] \leq 1/(1-0,9) = 10$
- $\alpha = 0,99$ , d.h. die Hashtabelle ist 99% voll:  $E[X] \leq 1/(1-0,99) = 100$

- **Korollar:** Unter der Annahme des gleichmäßigen Hashings benötigt HASH-INSERT() im Mittel höchstens  $1/(1-\alpha)$  Sondierungen.
- HASH-INSERT() benötigt konstante Zeit pro Sondierung + konstante Zeit für das Einfügen des Elements, also asymptotisch im Mittel  $O(1/(1-\alpha))$ .



## Offene Adressierung: Analyse

- **Theorem 13:** Für eine Hashtabelle mit offener Adressierung und Belegungsfaktor  $\alpha < 1$  ist die Anzahl der Sondierungen bei erfolgreicher Suche unter den Annahmen (1) gleichmäßiges Hashing und (2) gleiche Wahrscheinlichkeit für den Suchschlüssel höchstens  $(1/\alpha) \ln(1/(1-\alpha))$ .

- **Beweis:**

- Angenommen, der gesuchte Schlüssel  $k$  war der  $(i+1)$ -te Schlüssel bzgl. der Einfüge-Reihenfolge. Dann war zum Zeitpunkt der Einfügung

$$\alpha = i/m \text{ und nach Theorem 12 gilt: } E[X(\text{HASH-SEARCH}(T, k))] \leq \frac{1}{1-i/m}$$

- Mittelung über alle möglichen Suchschlüssel ergibt:

$$\begin{aligned} E[X] &= \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1-i/m} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \left( \sum_{i=0}^{n-1} \frac{1}{m-i} \right) \\ &= \frac{1}{\alpha} \left( \frac{1}{m} + \frac{1}{m-1} + \dots + \frac{1}{m-n+1} \right) = \frac{1}{\alpha} \left( \sum_{i=m-n+1}^m \frac{1}{i} \right) \\ &= \frac{1}{\alpha} \left( \sum_{i=1}^m \frac{1}{i} - \sum_{i=1}^{m-n} \frac{1}{i} \right) = \frac{1}{\alpha} (H_m - H_{m-n}) \end{aligned}$$

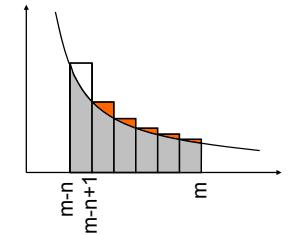
$H_m$ : m-te harmonische Zahl  
siehe Cormen A.7



## Offene Adressierung: Analyse

- **Beweis Theorem 13 (Fortsetzung)**

$$\begin{aligned} E[X] &= \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1-i/m} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \left( \sum_{i=0}^{n-1} \frac{1}{m-i} \right) \\ &= \frac{1}{\alpha} \left( \frac{1}{m} + \frac{1}{m-1} + \dots + \frac{1}{m-n+1} \right) = \frac{1}{\alpha} \left( \sum_{i=m-n+1}^m \frac{1}{i} \right) \\ &\leq \frac{1}{\alpha} \int_{m-n}^m \frac{1}{x} dx \\ &= \frac{1}{\alpha} (\ln(m) - \ln(m-n)) \\ &= \frac{1}{\alpha} \ln \left( \frac{m}{m-n} \right) = \frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right) \end{aligned}$$



Approximation von Summe durch Integral, siehe Cormen A.2

- **Beispiele:**

- $\alpha = 0,5$ , d.h. die Hashtabelle ist halbvoll:  $E[X] \leq 2 \ln(2) \approx 1,39$
- $\alpha = 0,9$ , d.h. die Hashtabelle ist 90% voll:  $E[X] \leq 10/9 \ln(10) \approx 2,56$
- $\alpha = 0,99$ , d.h. die Hashtabelle ist 99% voll:  $E[X] \leq 100/99 \ln(100) \approx 4,65$



## Einfügen mit Vertauschen: Brents INSERT-Operation

[siehe Ottmann, Widmayer, Spektrum Akad. Verlag 2002, Kap. 4.3]

- **Doppeltes Hashing mit Brents INSERT-Algorithmus:**

- Betrachte zwei alternative Plätze für  $k$  ( $j_{\text{next}}$ ) und  $T[k]$  ( $j_{\text{alt}}$ )

- **BRENT-INSERT( $T, k$ )**

```
t ← 0; j ← h1(k)
while T[j] ∈ {NIL, DELETED} and t < m do
    t ← t+1
    jnext ← (j + h2(k)) mod m
    jalt ← (j + h2(T[j])) mod m
    if T[jnext] ∈ {NIL, DELETED} or T[jalt] ∈ {NIL, DELETED} then j ← jnext
    else SWAP(k, T[j]); j ← jalt
if t = m then error „Hashtable overflow!“
else T[j] ← k
```

**Der Algorithmus ist korrekt, allerdings ist es nicht Brents INSERT-Operation.**

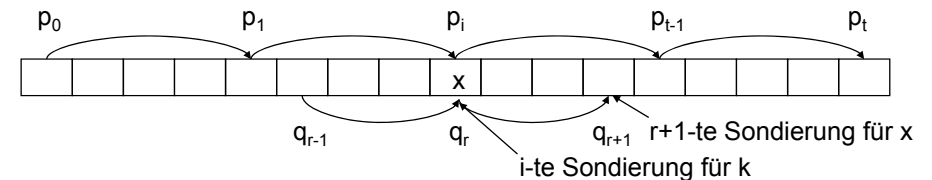


## Einfügen mit Vertauschen: Brents INSERT-Operation

[siehe Knuth, Art of Computer Programming, Vol 3, Kap. 6.4]

- **Doppeltes Hashing mit Brents INSERT-Algorithmus:**

- Betrachte für eine Sondierungsfolge  $p_0, p_1, \dots, p_t$  nach jedem erfolglosem Sondierungsschritt  $i$  die Möglichkeit, die Schlüssel  $T[p_0], \dots, T[p_{i-1}]$  durch  $(t-i-1)$ -faches Sondieren zu platzieren.
- Worst-Case Laufzeit für BRENT-INSERT() ist quadratisch in der Anzahl der Sondierungen
- durchschn. Suchzeit  $< 2.5$  (ohne Löschen) ist unabhängig von  $\alpha$ !



Falls  $T[q_{r+(t-i-1)}] \in \{\text{NIL}, \text{DELETED}\}$ :

- $T[q_{r+(t-i-1)}] \leftarrow x$  Sondierungsfolge  $q$  für  $x$  verlängert sich um  $t-i-1$
- $T[p_i] \leftarrow k$  Sondierungsfolge  $p$  für  $k$  verkürzt sich um  $t-i$

**Bilanz:** Sondierungsfolgen sind um 1 kürzer als bei doppeltes Hashing



## Einfügen mit Vertauschen: Brents INSERT-Operation

```
■ BRENT-INSERT(T, k)
  t ← 0          // zählt die Anzahl der Sondierungsschritte
  j ← h1(k)       // aktuell sondierte Position für Schlüssel k
  while T[j] ∉ {NIL, DELETED} and t < m do
    t ← t+1
    j ← (j + h2(k)) mod m
    if t ≥ 2 and T[j] ∉ {NIL, DELETED} then      // Brents Modifikation
      p_i ← h1(k)
      for i ← 0 to t-2 do
        // teste, ob T[p_i] mit einem weiteren Sondierungsschritt eingefügt
        // werden kann
        x ← T[p_i]
        q_next ← (p_i + (t-i-1)h2(x)) mod m
        if T[q_next] ∈ {NIL, DELETED} then
          T[q_next] ← x
          T[p_i] ← DELETED; j ← p_i
          break;
      p_i ← (p_i + h2(k)) mod m
  if t = m then error „Hashtable overflow!“
  else T[j] ← k; return j
```



## Abschließende Kommentare zu Hashing

- Durch Umrechnung von Schlüssel in Indizes (**Hashfunktion**) lässt sich ein Wörterbuch-Datenstruktur mit nahezu konstanten durchschnittlichen Laufzeiten für INSERT, DELETE und SEARCH realisieren.
- Laufzeiten im Average Case hängen vom **Belegungsfaktor** ab.
- **Kollisionen sind nicht vermeidbar**, im Worst-Case haben die Wörterbuch-Operationen Laufzeit  $\Theta(N)$ .
- Werden häufig Daten gelöscht, sollten **Kollisionen durch Verkettung** aufgelöst werden.
- **Offene Adressierung** ist speichereffizienter als Verkettung, allerdings wirkt die Operation DELETE laufzeitschädlich auf SEARCH. Offene Adressierung sollte nur angewandt werden, wenn wenige DELETE-Operationen notwendig sind.
- Bei der offenen Adressierung ist **Doppeltes Hashing** die zu bevorzugende Sondierungsmethode.
- **Brents INSERT-Operation** mit Doppeltem Hashing ermöglicht das Einfügen mit durchschnittlich weniger als 2.5 Sondierungsschritten.
- Für statische Datenmengen gibt es **perfekte Hashverfahren** mit Worst-Case Laufzeit  $O(1)$  für INSERT und SEARCH (nicht Teil dieser Vorlesung).



## **Kapitel 5: Graphalgorithmen**

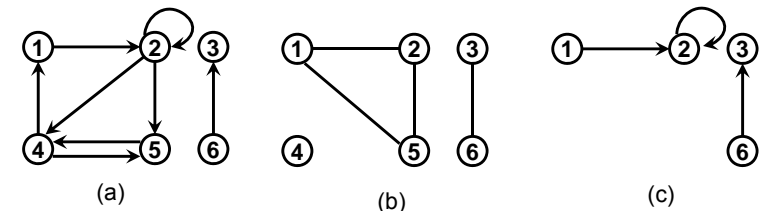
- 5.1 Graphen 117
- 5.2 Traversieren von Graphen 120
- 5.3 Topologisches Sortieren 131
- 5.4 Starke Zusammenhangskomponenten 134
- 5.5 Minimale Spannbäume 138
- 5.6 Kürzeste Pfade 147
- 5.7 Weitere Graphprobleme im Überblick 158

## Kap. 5: Graphalgorithmen

Graphen  
Traversieren von Graphen  
Topologisches Sortieren  
Starke Zusammenhangskomponenten  
Spannbäume  
Kürzeste Pfade  
Weiterführende Graphalgorithmen im Überblick

## 5.1 Graphen

- Graph:  $G = (V, E)$ 
  - $V$  = Menge der **Knoten**,  $E$  = Menge der **Kanten**,  $|V| = N$ ,  $|E| = M$
  - **ungerichtet**:  $E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\}$  keine Schleifen
  - **gerichtet**:  $E \subseteq \{(x, y) \mid x, y \in V\}$  Schleifen
  - **Multigraph**: ungerichtet mit multiplen Kanten + Schleifen
  - **Hypergraph**:  $E \subseteq 2^V$
  - Knoten  $v, w$  sind **adjazent**:  $\{v, w\} \in E$
  - Kante  $e$  ist **inzident** mit Knoten  $v$ :  $v \in e$ 
    - ◆ gerichteter Graph:  $e = (v, w)$ ,  $e$  ist **inzident von**  $v$  und **inzident nach**  $w$



© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al, Oldenbourg Verlag, 2004

2

## Graphen

- **Grad** eines Knotens:  $\deg(v) = |\{e \in E \mid v \in e\}|$ 
  - gerichtet:  $\text{indeg}(v) = |\{e \in E \mid e = (x, v)\}|$ ,  $\text{outdeg}(v) = |\{e \in E \mid e = (v, x)\}|$ ,  $\deg(v) = \text{indeg}(v) + \text{outdeg}(v)$
- **Pfad** der Länge  $k$ :  $(v_0, v_1, \dots, v_k)$ ,  $v_i \in V$ ,  $(v_{i-1}, v_i) \in E$  für alle  $1 \leq i \leq k$ 
  - einfacher Pfad:  $v_i \neq v_j$  für alle  $i \neq j$ ,  $1 \leq i, j \leq k$
  - Teilpfad:  $(v_i, \dots, v_j)$  mit  $1 \leq i < j \leq k$
  - Kreis / Zyklus:  $v_0 = v_k$
  - Graph ohne Kreise: azyklisch
- **Erreichbarkeit**:
  - $v$  ist **erreichbar** von  $w$ , g.d.w. es existiert ein Pfad von  $w$  nach  $v$
  - ungerichteter Graph  $G=(V, E)$  ist **verbunden** (connected), g.d.w.  $\forall v, w \in V \exists$  Pfad von  $v$  nach  $w$
  - gerichteter Graph  $G=(V, E)$  ist **stark verbunden** (strongly connected), g.d.w.  $\forall v, w \in V \exists$  Pfad von  $v$  nach  $w$  und von  $w$  nach  $v$



© Matthias Rarey, ZBH, Universität Hamburg

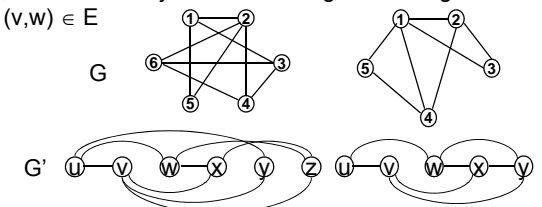
gescannte Abbildungen: © Cormen et al, Oldenbourg Verlag, 2004

3

## Graphen

- **Subgraph**:
  - $G'=(V', E')$  ist ein **Subgraph** von  $G=(V, E)$ , falls  $V' \subseteq V$ ,  $E' \subseteq E$
  - $G'=(V', E')$  ist der durch  $V'$  induzierte Subgraph, falls  $E' = \{(u, v) \in E \mid u, v \in V'\}$
- **Zusammenhang**:
  - **Zusammenhangskomponente** (connected component): maximal zusammenhängenden Subgraphen eines ungerichteten Graphen
  - **starke Zusammenhangskomponente** (strongly connected component): analog für gerichtete Graphen
- **Isomorphie**:
  - $G$  und  $G'$  sind **isomorph**, falls es eine bijektive Abbildung  $f: V \rightarrow V'$  gibt mit  $(f(v), f(w)) \in E'$  g.d.w.  $(v, w) \in E$

- a) isomorph  
( $1=u, 2=v, \dots, 6=z$ )
- b) nicht isomorph



© Matthias Rarey, ZBH, Universität Hamburg

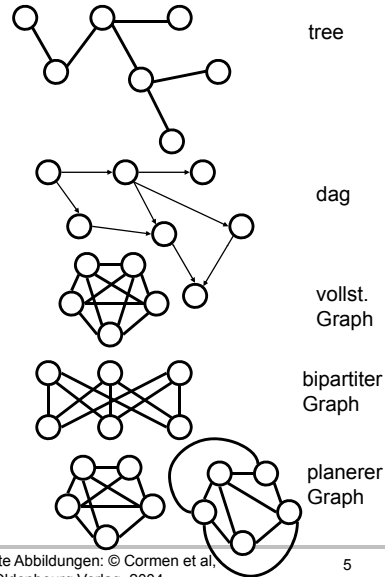
gescannte Abbildungen: © Cormen et al, Oldenbourg Verlag, 2004

4



## Spezielle Graphen

- **Baum (tree):**
  - ungerichteter, zusammenhängender Graph ohne Kreise
  - gewurzelt (rooted): ausgezeichnete Wurzelknoten
- **Wald (forest):**
  - ungerichteter Graph ohne Kreise
- **dag (directed acyclic graph):**
  - gerichteter Graph ohne Kreise
- **vollständiger Graph (complete graph):**
  - ungerichteter Graph, in dem alle Paare von Knoten adjazent sind
- **bipartiter Graph:**
  - Knotenmenge lässt sich in zwei disjunkte Teilmengen  $U, W$  aufteilen,  $\forall (v, w) \in E : v \in U \text{ und } w \in W \text{ oder } v \in W \text{ und } w \in U$
- **planarer Graph:**
  - Graph lässt sich ohne Kantenüberschneidungen zeichnen

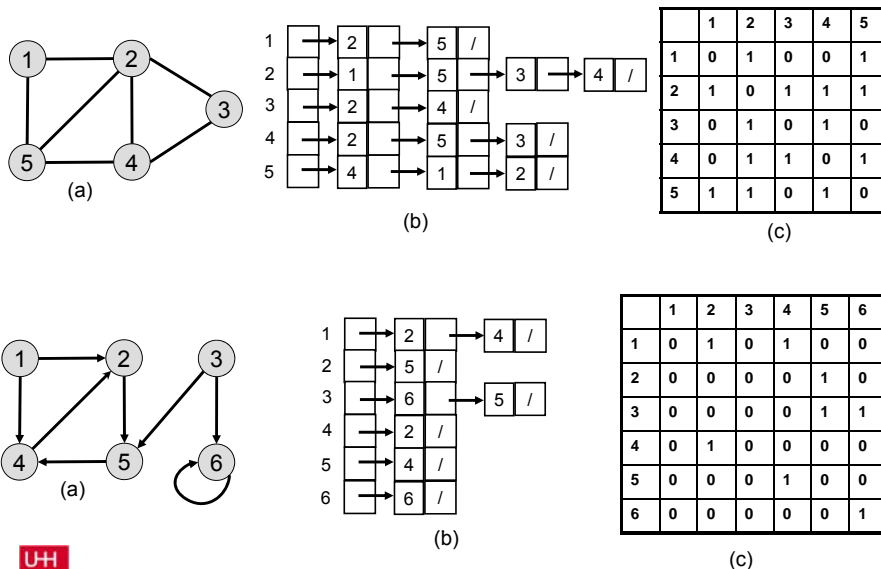


## Repräsentation von Graphen

- **Knoten:** Array oder Liste, nummeriert von 1 bis  $N$
- **Kanten:**
  - **Adjazenzmatrix:**  $N \times N$  Matrix
    - ◆  $A[i, j] = 1$  falls  $(i, j) \in E$ , bzw.  $\{i, j\} \in E$ ;  $A[i, j] = 0$  sonst
    - ◆ Speicherbedarf  $\Theta(N^2)$
    - ◆ ungerichtete Graphen:  $A[i, j] = A[j, i]$  (Matrix ist symmetrisch)
    - ◆ gerichtete Graphen:  $A^T = ([a_{ij}])^T = [a_{ji}]$  ist die Adjazenzmatrix des transponierten Graphen (Umkehr aller Kanten)
    - ◆ Effizienter Test auf das Vorhandensein einer Kante
  - **Adjazenzlisten:**
    - ◆ Pro Knoten eine Liste mit Nummern der adjazenten Knoten
    - ◆ Speicherbedarf  $\Theta(N+M)$ 
      - ◆ genauer:  $2M$  falls  $G$  ungerichtet,  $M$  falls  $G$  gerichtet ist
    - ◆ Effizientes Durchlaufen aller zu einem Knoten  $v$  adjazenten Knoten  $w$
- **Kanten und Knoten tragen anwendungsspezifische Information**
  - **gewichtete Graphen:**  $w: E \rightarrow \mathbb{R}$  (jede Kante hat ein Gewicht)
  - **knotengewichtete Graphen:**  $w: V \rightarrow \mathbb{R}$



## Repräsentation von Graphen



## 5.2 Traversieren von Graphen

- **Ziel:**
  - besuche alle Knoten eines Graphen in einer systematischer Reihenfolge
  - Grundgerüst für spätere Graphalgorithmen
- **Breitensuche (breadth first search):**
  - Alle Knoten werden zunächst als weiß markiert (noch nicht besucht)
  - Zum Zeitpunkt der ersten Entdeckung werden diese grau gefärbt.
  - Ein Knoten wird schwarz, wenn alle adjazenten Knoten bereits entdeckt wurden (d.h. nicht mehr weiß sind).
  - Start an einem beliebigen Knoten  $s$
  - Besuche iterativ die noch nicht besuchten Nachbarn von  $s$ , dann die Nachbarn der Nachbarn von  $s$ , usw.
  - Realisierung durch Warteschlange
  - Vorgänger / Vater von  $u$ : wird  $u$  erstmals über die Kante  $(v, u)$  besucht, wird  $v$  als der Vorgänger von  $u$  bezeichnet
  - Die Eigenschaft ‚Vorgänger von‘ beschreibt einen Baum (BFS-Baum) mit Wurzel  $s$
  - Distanz von  $u$ : Pfadlänge von der Wurzel  $s$  bis  $u$  im BFS-Baum



## Breitensuche (breadth-first-search)

### BFS(G,s)

```

1 for each  $u \in V[G] \setminus \{s\}$  do
2   color[u]  $\leftarrow$  WEISS
3   d[u]  $\leftarrow \infty$ ;  $\pi[u] \leftarrow \text{NIL}$ 
4 color[s]  $\leftarrow$  GRAU
5 d[s]  $\leftarrow$  0;  $\pi[s] \leftarrow \text{NIL}$ 
6 Q  $\leftarrow \emptyset$ ; ENQUEUE(Q,s)
7 while Q  $\neq \emptyset$  do
8   u  $\leftarrow$  DEQUEUE(Q)
9   for each  $v \in \text{Adj}[u]$  do
10    if color[v] = WEISS then
11      color[v]  $\leftarrow$  GRAU
12      d[v]  $\leftarrow$  d[u]+1;  $\pi[v] \leftarrow u$ 
13      ENQUEUE(Q, v)
14 color[u]  $\leftarrow$  SCHWARZ

```

### color:

- weiß: noch nicht besucht
- grau: besucht, hat noch nicht besuchte Nachbarn
- schwarz: besucht, alle Nachbarn besucht

### d[u]:

- Pfadlänge von u zu s

### $\pi[u]$ :

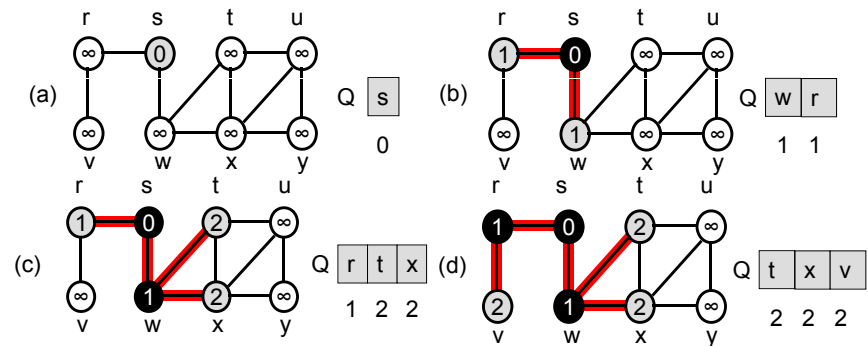
- Vorgänger von u

### Warteschlange Q:

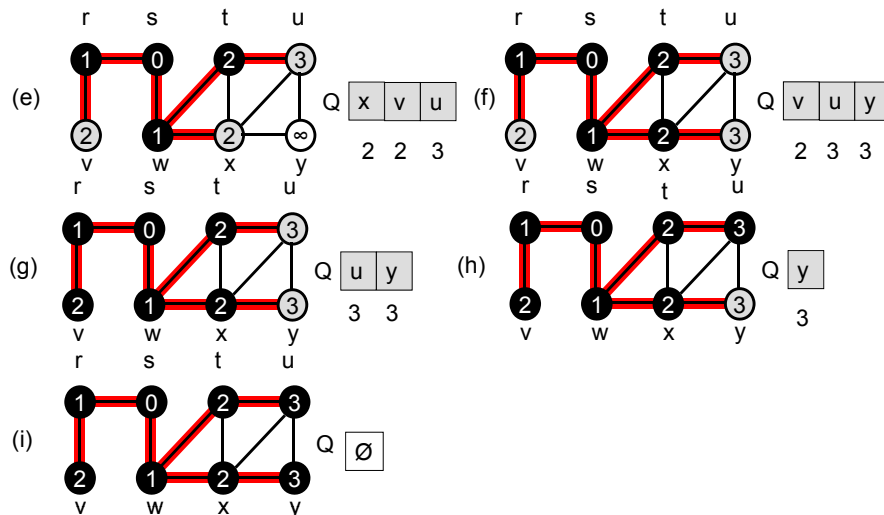
- alle grauen Knoten, Suchfront



## Breitensuche: Beispiel



## Breitensuche: Beispiel



## Breitensuche: Analyse

### BFS-Baum: (Vorgängerteilgraph)

- Kanten  $E_\pi = \{(\pi(u), u) \mid u \in V, \pi(u) \neq \text{NIL}\}$  bezeichnen die Kanten, über die die Knoten erstmalig besucht werden.
- Sei  $V_\pi = \{u \in V \mid \pi(u) \neq \text{NIL}\} \cup \{s\}$  die während des BFS besuchten Knoten
- Graph  $G_\pi = (V_\pi, E_\pi)$  beschreibt einen Baum mit Wurzel s:
  - ◆  $G_\pi$  ist zusammenhängend: Für alle Knoten v existiert ein Pfad von v zur Wurzel über die  $(u, \pi(u))$ -Kanten
  - ◆  $G_\pi$  enthält keine Kreise: jeder Knoten wird nur einmalig grau gefärbt, zu diesem Zeitpunkt wird eine Kante zu einem Vorgänger eingefügt.

### Die asymptotische Laufzeit von BFS() ist $O(N+M)$ .

- Initialisierung:  $O(|V|)$
- Jeder Knoten wird einmalig in Q eingefügt und wieder entnommen:  $O(|V|)$
- In der while-Schleife, wird jede Adjazenzliste einmal durchlaufen:

$$T_{BFS}(V, E) = c|V| + \sum_{v \in V} \sum_{(v,u) \in E} c = c|V| + 2c|E| = O(|V| + |E|)$$



## Breitensuche: Bestimmung kürzester Pfade

- Sei  $\delta(s,u)$  die minimale Anzahl Kanten eines Pfades von  $s$  nach  $u$ . Existiert kein solcher Pfad sei  $\delta(s,u)=\infty$ .  $\delta(s,u)$  wird als der **kürzeste Abstand**, der Pfad von  $s$  nach  $u$  als **kürzester Pfad** bezeichnet.
- **Lemma 22.1:** Struktur kürzester Pfade  
Sei  $G = (V,E)$  ein Graph,  $s \in V$ . Für alle Kanten  $e = (u,v) \in E$  gilt:  
 $\delta(s,v) \leq \delta(s,u) + 1$ .
  - Beweis:  
Fall 1:  $\delta(s,u) = \infty$   
Dann ist  $u$  nicht erreichbar von  $s$  und somit auch  $v$  nicht, d.h.  $\delta(s,v) = \infty$   
Fall 2:  $\delta(s,u) < \infty$   
Es gibt einen Pfad von  $s$  zu  $u$  und dann über Kante  $e$  zu  $v$ . Da  $\delta(s,v)$  den kürzesten Abstand beschreibt, gilt  $\delta(s,v) \leq \delta(s,u) + 1$
- **Lemma 22.2:** BFS beschränkt  $\delta(s,u)$  von oben  
Sei  $G=(V,E)$  ein Graph,  $s \in V$ ,  $d[]$  durch BFS berechnet, dann gilt  $d[v] \geq \delta(s,v)$ .



## Breitensuche: Bestimmung kürzester Pfade

- Beweis: (durch Induktion über Einfüge-Reihenfolge in  $Q$ )  
Annahme:  $d[v] \geq \delta(s,v) \forall v \in V$   
Induktionsanfang:  $d[s] = 0 = \delta(s,s)$  und  $d[v] = \infty \geq \delta(s,v)$ .  
Induktionsschritt: Betrachte den Zeitpunkt, in den  $v$  über Kante  $(u,v)$  in  $Q$  eingefügt wird:  
$$\begin{aligned} d[v] &= d[u] + 1 && \text{(Zeile 15 des BFS)} \\ &\geq \delta(s,u) + 1 && \text{(Induktionsannahme)} \\ &\geq \delta(s,v) && \text{(Lemma 22.1)} \end{aligned}$$
- **Lemma 22.3:** Struktur von  $Q$   
Sei  $G(V,E)$  ein Graph,  $s \in V$ ,  $d[]$  durch BFS berechnet,  $Q = (v_1, \dots, v_r)$  die Warteschlange im BFS. Zu jedem Zeitpunkt gilt  $d[v_1] \leq d[v_2] \leq \dots \leq d[v_i] \leq d[v_{i+1}] \leq \dots \leq d[v_r] \leq d[v_1] + 1$ .
  - Beweis: (durch Induktion über die Operationen auf  $Q$ )  
Induktionsanfang:  $Q = (s)$  mit  $d[s] = 0$ .



## Breitensuche: Bestimmung kürzester Pfade

- Beweis: (Fortsetzung)  
Induktionsschritt:  
Fall 1: ausgeführte Operation war DEQUEUE()
  - ◆  $v_2$  wird neuer Kopf der Liste. Da  $d[v_2] \geq d[v_1]$ , folgt mit der Induktionsannahme das Lemma.
- Fall 2: ausgeführte Operation war ENQUEUE()
  - ◆ Sei  $v$  der neu eingefügte Knoten  $v_{r+1}$ ,  $u$  der zuvor aus  $Q$  entnommene Knoten. Dann gilt  $d[v] = d[u] + 1$  (Zeile 15).
  - ◆ Nach Induktionsannahme gilt  $d[u] \leq d[v_1]$ . Es folgt  $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$ . Zudem gilt  $d[v_r] \leq d[u] + 1 = d[v_{r+1}]$ .
- Struktur von  $Q$ :
  - Die  $d[]$ -Werte der in  $Q$  gespeicherten Knoten sind monoton steigend.
  - Da die  $d[]$ -Werte diskret sind, gibt es unter den gespeicherten Knoten maximal zwei mögliche  $d[]$ -Werte:  $d[\text{HEAD}(Q)]$  und  $d[\text{HEAD}(Q)] + 1$ .



## Breitensuche: Bestimmung kürzester Pfade

- **Theorem 22.5:** Sei  $G=(V,E)$  ein Graph,  $s \in V$ . Nach Durchführung von BFS() gilt für alle Knoten  $v$ , die von  $s$  aus erreichbar sind:
  1.  $\text{color}[v] = \text{SCHWARZ}$
  2.  $d[v] = \delta(s,v)$  (d.h.  $d[v]$  entspricht dem kürzesten Abstand)
  3. Es gibt einen kürzesten Pfad von  $s$  nach  $v$ , der mit der Kante  $(\pi[v],v)$  endet.
- Beweis (Eigenschaft 2 durch Widerspruch)  
Aus Lemma 22.1 folgt bereits  $d[v] \geq \delta(s,v)$ .  
Annahme: Es gibt einen Knoten  $v$  mit  $d[v] > \delta(s,v)$ .  
Offensichtlich ist  $v \neq s$ , da  $d[s] = 0 = \delta(s,s)$  und  $v$  ist von  $s$  aus erreichbar, da sonst  $d[v] = \infty = \delta(s,v)$  gilt. Sei  $v$  gewählt mit  $d[v] > \delta(s,v)$  und  $\delta(s,v)$  minimal.  
Sei  $u$  der Vorgänger von  $v$  auf dem kürzesten Pfad, d.h.  $\delta(s,v) = \delta(s,u) + 1$ .  
Aufgrund der Wahl von  $v$  (Minimalität) gilt für  $u$ :  $d[u] = \delta(s,u)$ . Es folgt:  
 $d[v] > \delta(s,v) = \delta(s,u) + 1 = d[u] + 1$



## Breitensuche: Bestimmung kürzester Pfade

### ■ Beweis von Theorem 22.5 (Fortsetzung)

Betrachte den Zeitpunkt der Entnahme von  $u$  aus  $Q$ :

Fall 1:  $\text{color}[v] = \text{WEISS}$

$d[v] = d[u] + 1$  (lt. Zeile 15 des  $\text{BFS}()$ ). ⚡

Fall 2:  $\text{color}[v] = \text{SCHWARZ}$

$v$  wurde bereits aus  $Q$  entfernt, nach Lemma 22.2 folgt  $d[v] \leq d[u]$  ⚡

Fall 3:  $\text{color}[v] = \text{GRAU}$

Sei  $w$  der Knoten, bei dessen Entnahme  $v$  grau gefärbt wurde. Es gilt

$d[w] \leq d[u]$  (wg. Lemma 22.2;  $w$  wurde vor  $u$  entfernt) und zudem

$d[v] = d[w] + 1 \leq d[u] + 1$ . ⚡

■ Eigenschaft 1: folgt direkt aus Eigenschaft 2, da  $d[v] = \infty$ , falls  $v$  nicht bearbeitet wurde

■ Eigenschaft 3: folgt direkt aus  $d[v] = d[\pi(v)] + 1$



## Tiefensuche (depth-first-search)

### ■ Strategie:

■ Suche in die Tiefe, d.h. für jeden Knoten, arbeite zuerst den ersten Nachfolger vollständig ab, dann den zweiten, u.s.w

■ Statt einer Schlange wird ein Stapel verwendet (durch Rekursion)

■ Farben wie bei BFS:

◆ weiß: noch nicht besucht

◆ grau: besucht, Adjazenzliste noch nicht abgearbeitet

◆ schwarz: vollständig abgearbeitet

■ Diskrete Zeitstempel:

◆  $d[u]$ : Zeitpunkt des erstmaligen Eintrags (weiß  $\rightarrow$  grau, discovery time)

◆  $f[u]$ : Zeitpunkt der vollst. Abarbeitung (grau  $\rightarrow$  schwarz, finishing time)

◆ Hinweise:

◆ Offensichtlich gilt  $d[u] < f[u]$

◆  $d[u]$  beschreibt nun nicht mehr den Abstand zu  $s$ !

■ DFS-Wald (Tiefensuchwald):

◆ Kanten die bei erstmaligen Besuch durchlaufen werden, beschreiben eine Menge von Bäumen

◆  $\pi[u]$ : Vorgänger im DFS-Wald



## Tiefensuche

### ■ DFS( $G$ )

Laufzeit:  $O(N+M)$

1 **for** each  $u \in V[G]$  **do**

2    $\text{color}[u] \leftarrow \text{WEISS}; \pi[u] \leftarrow \text{NIL}$

4  $\text{time} \leftarrow 0$

5 **for** each  $u \in V[G]$  **do**

6   **if**  $\text{color}[u] = \text{WEISS}$  **then**  $\text{DFS-VISIT}(u)$

### ■ DFS-VISIT( $u$ )

1  $\text{color}[u] \leftarrow \text{GRAU}$

// Entdeckung von  $u$

2  $\text{time} \leftarrow \text{time} + 1; d[u] \leftarrow \text{time}$

4 **for** each  $v \in \text{Adj}[u]$  **do**

// Sondierung der Adjazenzliste

5   **if**  $\text{color}[v] = \text{WEISS}$  **then**

6      $\pi[v] \leftarrow u$

7      $\text{DFS-VISIT}(v)$

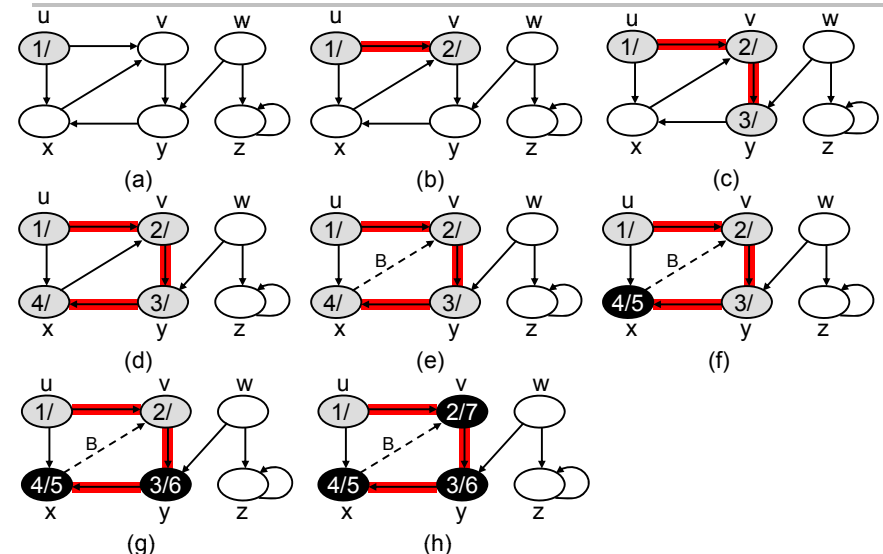
8  $\text{color}[u] \leftarrow \text{SCHWARZ}$

9  $\text{time} \leftarrow \text{time} + 1; f[u] \leftarrow \text{time}$

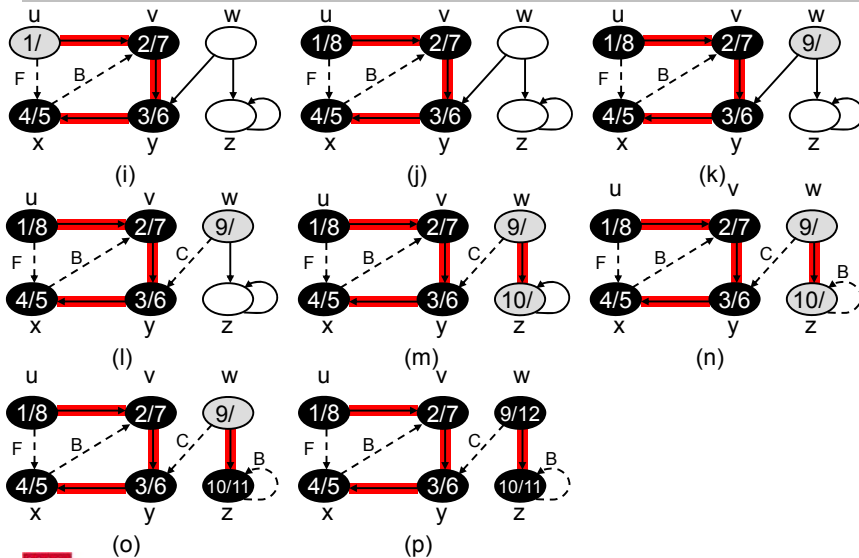
// Vollständige Abarbeitung von  $u$



## Tiefensuche: Ein Beispiel



## Tiefensuche: Ein Beispiel



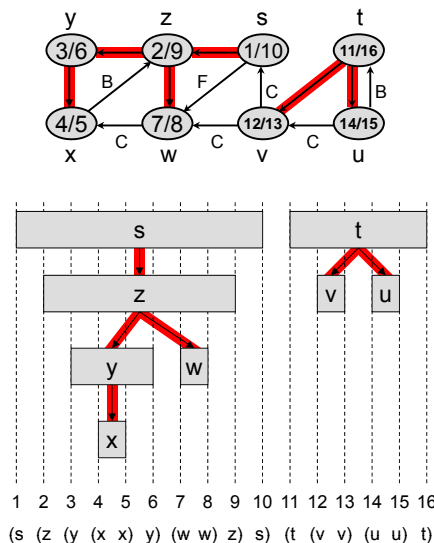
## Tiefensuche: Struktureigenschaften

- **Theorem 22.7** (Klammerungstheorem): Nach Durchführung von DFS gelten für die Entdeckungs- und Endzeiten  $d[]$  und  $f[]$  für je zwei Knoten  $u$  und  $v$ :
  1.  $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$  und  $u$  ist im DFS-Wald weder Vorfahre noch Nachfahre von  $v$
  2.  $[d[u], f[u]] \subset [d[v], f[v]]$  und  $u$  ist im DFS-Wald Nachfahre von  $v$
  3.  $[d[u], f[u]] \supset [d[v], f[v]]$  und  $u$  ist im DFS-Wald Vorfahre von  $v$
- Beweis: O.B.d.A. nehmen wir  $d[u] < d[v]$  an.
  - Fall 1:  $d[v] < f[u]$ :** Zum Zeitpunkt  $d[v]$  galt  $\text{color}[u] = \text{GRAU}$ 
    1.  $\Rightarrow u$  ist ein Vorfahre von  $v$ , da  $u$  noch nicht abgearbeitet ist.
    2.  $\Rightarrow$  alle Kanten von  $v$  werden sondiert, bevor  $u$  abgearbeitet wird, d.h.  $f[v] < f[u]$ $\Rightarrow$  Fall 3 des Theorems.
  - Fall 2:  $d[v] > f[u]$ :** Zum Zeitpunkt  $d[v]$  galt  $\text{color}[u] = \text{SCHWARZ}$ 
    1.  $\Rightarrow u$  ist bereits abgearbeitet, somit ist  $u$  kein Vorfahre von  $v$
    2.  $\Rightarrow v$  wird nach  $u$  entdeckt, somit ist  $u$  kein Nachfahre von  $v$
    3.  $\Rightarrow$  da  $d[v] < f[v]$  und  $d[u] < f[u] < d[v] < f[v]$ $\Rightarrow$  Fall 1 des Theorems.



## Tiefensuche: Bedeutung des Klammerungstheorems

- Beispiel:
  - Knoten enthalten  $d[v]/f[v]$
  - Baumkanten sind grau unterlegt
  - Startknoten ist  $s$  (dann  $t$ )
- Knoten können durch Intervalle auf der Zeitachse dargestellt werden.
- Intervalle sind entweder disjunkt oder vollständig enthalten.
- DFS-Wald lässt sich durch Intervallstruktur ableiten.
- ACHTUNG: der DFS-Wald ist nicht eindeutig:
  - Wahl des Startknotens
  - Reihenfolge der Kanten in den Adjazenzlisten



## Tiefensuche: Struktureigenschaften

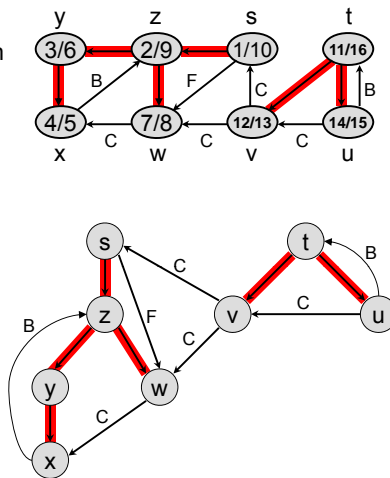
- **Theorem 22.9** (Weiße Pfade): In einem DFS-Wald ist  $v$  ein Nachfahre von  $u$  genau dann wenn zum Zeitpunkt  $d[u]$  ein Pfad  $(u, w_1, \dots, w_n=v)$  von  $u$  von  $v$  existiert mit  $\text{color}[w_i] = \text{WEISS}$  für  $i=1, \dots, n$ .
- Beweis: Teil 1 ( $\Rightarrow$ ):  $v$  ist Nachfahre von  $u$ 
  - Für alle Nachfahren  $w$  von  $u$  gilt:  $d[w] > d[u]$  (Theorem 22.7) und somit  $\text{color}[w] = \text{WEISS}$ . Damit existiert ein weißer Pfad (im DFS-Wald).
- Teil 2 ( $\Leftarrow$ ): Es existiert ein weißer Pfad  $(u, w_1, \dots, w_{n-1}=w, w_n=v)$ .
  - Annahme:  $v$  ist kein Nachfahre von  $u$
  - Offensichtlich gilt  $d[v] > d[u]$  ( $v$  ist weiß zum Zeitpunkt  $d[u]$ ).
  - Sei o.B.d.A.  $v$  entlang des Pfades der erste Knoten, der kein Nachfahre von  $u$  ist, d.h.  $w$  ist Nachfahre von  $u$ .
    - ◆  $f[w] \leq f[u]$  ( $w$  ist Nachfahre von  $u$  und nach Theorem 22.7 Fall 2)
    - ◆  $d[v] < f[w]$  (wg. Kante  $(w,v)$  wird  $v$  entdeckt bevor  $w$  abgearbeitet ist) $\Rightarrow d[u] < d[v] < f[w] \leq f[u] \Rightarrow [d[u], f[u]] \supset [d[v], f[v]]$ 
 $\Rightarrow u$  ist Vorfahre von  $v$ , d.h.  $v$  ist Nachfahre von  $u$ . ⚡



## Tiefensuche: Kantenklassifikation

### Klassifikation von Kanten $e = (u,v)$ :

- Baumkante:** wird durchlaufen beim ersten Besuch von  $v$   
[dicke graue Kanten]
  - Rückkante:** zeigt auf einen Vorgänger im DFS-Baum  
[Typ B]
  - Vorwärtskante:** zeigt auf einen Nachfolger im DFS-Baum  
[Typ F]
  - Querkante:** nicht 1-3  
[Typ C]
- Die Zuordnung der Kantentypen in einem DFS-Wald ist eindeutig, die Kantentypen sind jedoch NICHT eindeutig für den Graph  $G$ .



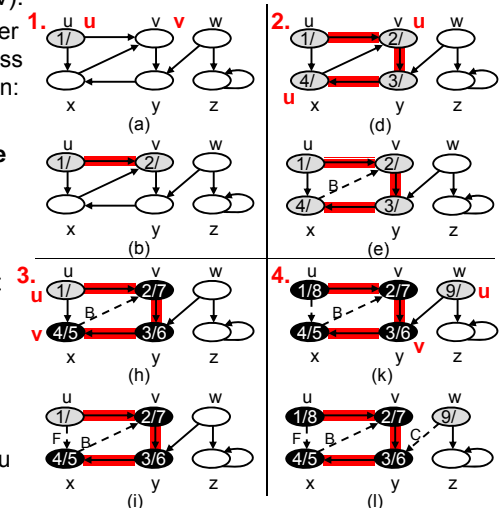
## Tiefensuche: Kantenklassifikation

### Klassifikation von Kanten $e = (u,v)$ :

- Zum Zeitpunkt der Sondierung der Kante  $e = (u,v)$  gilt zum einen, dass  $\text{color}[u] = \text{GRAU}$  ist, zum anderen:

- $\text{color}[v] = \text{WEISS} \Rightarrow$  **Baumkante**
- $\text{color}[v] = \text{GRAU} \Rightarrow$  **Rückkante**
- $\text{color}[v] = \text{SCHWARZ}$ ,  $d[u] < d[v] \Rightarrow$  **Vorwärtskante**
- $\text{color}[v] = \text{SCHWARZ}$ ,  $d[u] > d[v] \Rightarrow$  **Querkante**

[da:  $d[u] < d[v] < f[v] < f[u]$  gilt, ist nach Theorem 22.7 Fall 2  $u$  ein Vorfahre von  $v$ ]  
[da  $d[v] < d[u]$  und  $f[v] < f[u]$  gilt nach Theorem 22.7 Fall 1, dass  $u$  weder Vorfahre noch Nachfahre von  $v$  ist]



## Tiefensuche mit Kantenklassifikation

### DFS( $G$ )

```

1 for each  $u \in V[G]$  do
2    $\text{color}[u] \leftarrow \text{WEISS}$ ;  $\pi[u] \leftarrow \text{NIL}$ 
3    $\text{time} \leftarrow 0$ 
4   for each  $u \in V[G]$  do
5     if  $\text{color}[u] = \text{WEISS}$  then DFS-VISIT( $u$ )

```

Laufzeit:  $O(N+M)$

### DFS-VISIT( $u$ )

```

1  $\text{color}[u] \leftarrow \text{GRAU}$  // Entdeckung von  $u$ 
2  $\text{time} \leftarrow \text{time} + 1$ ;  $d[u] \leftarrow \text{time}$ 
3 for each  $v \in \text{Adj}[u]$  do // Sondierung der Adjazenzliste
4   if  $\text{color}[v] = \text{WEISS}$  then
5      $\text{type}[(u,v)] \leftarrow \text{TREE}$ ;  $\pi[v] \leftarrow u$ 
6     DFS-VISIT( $v$ ) // ACHTUNG:  $\text{color}[v]$  ändert sich!
7   else if  $\text{color}[v] = \text{GRAU}$  then  $\text{type}[(u,v)] \leftarrow \text{BACK}$ 
8   else if  $d[u] < d[v]$  then  $\text{type}[(u,v)] \leftarrow \text{FORWARD}$ 
9   else  $\text{type}[(u,v)] \leftarrow \text{CROSS}$ 
10  $\text{color}[u] \leftarrow \text{SCHWARZ}$ 
11  $\text{time} \leftarrow \text{time} + 1$ ;  $f[u] \leftarrow \text{time}$  // Vollständige Abarbeitung von  $u$ 

```



## Tiefensuche: Kantenklassifikation in ungerichteten Graphen

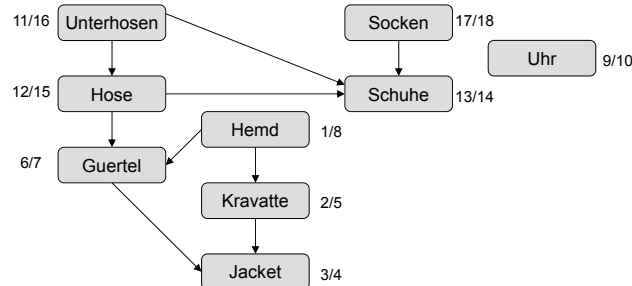
- Im Falle von ungerichteten Graphen wird jede Kante zwei mal betrachtet. Die Klassifikation wird bei der **ersten Sondierung** vorgenommen.
- Angenommen ein ungerichteter Graph  $G=(V,E_u)$  wird durch einen gerichteten Graphen  $(V, E)$  mit  $E = \{(u,v) \in V \times V \mid \{u,v\} \in E_u\}$  repräsentiert:
- Angenommen,  $(u,v)$  wird vor  $(v,u)$  sondiert, dann gilt:
  - $\text{type}[(u,v)] = \text{TREE} \Rightarrow \text{type}[(v,u)] = \text{BACK}$
  - $\text{type}[(u,v)] = \text{BACK} \Rightarrow \text{type}[(v,u)] = \text{FORWARD}$
  - $\text{type}[(u,v)] = \text{FORWARD}$  oder  $\text{type}[(u,v)] = \text{CROSS} \Rightarrow \text{color}[v] = \text{SCHWARZ}$   
 $\Rightarrow$  alle Kanten von  $v$  wurden bereits sondiert  
 $\Rightarrow (v,u)$  wurde vor  $(u,v)$  sondiert. ⚡
- Theorem 22.10:** Ist  $G$  ein ungerichteter Graph, so ist jede Kante in  $G$  entweder eine Baum- oder Rückkante.
  - Beweis: s.o.





## 5.3 Topologisches Sortieren

- DAG (directed acyclic graph): gerichteter Graph ohne Kreise
- **Topologische Sortierung** der Knotenmenge  $V$ :
  - Sei  $G=(V,E)$  ein gerichteter Graph. Eine bijektive Funktion  $\pi: V \rightarrow \{1, \dots, |V|\}$  heißt topologische Sortierung, genau dann wenn für alle  $e=(v,w) \in E$  gilt:  $\pi(v) < \pi(w)$
- Anwendung der topologischen Sortierung:
  - Sortierung bei partieller Ordnung
  - Priorisierungs- und Schedulingprobleme:



## Topologisches Sortieren: Existenz in DAGS

- **Lemma 22.x:** Sei  $G=(V,E)$  ein gerichteter Graph. Für  $G$  existiert eine topologische Sortierung, genau dann wenn  $G$  keine Kreise enthält (d.h. ein DAG ist).
  - Beweis: Teil 1 ( $\Rightarrow$ ): Sei  $\pi$  eine topologische Sortierung für  $G$ . Angenommen  $(v_1, v_2, \dots, v_r=v_1)$  sei ein Kreis in  $G$ . Es gilt  $\pi(v_1) < \pi(v_2) < \dots < \pi(v_r) = \pi(v_1)$ .
  - Teil 2 ( $\Leftarrow$ ): Sei  $G$  ein DAG. Wir zeigen zunächst, dass ein Knoten  $u \in V$  mit  $\text{indeg}(u) = 0$  existiert: Sie  $G^T = (V, E^T)$  der zu  $G$  transponierte Graph (d.h. alle Kanten werden umgekehrt). Betrachte den folgenden Algorithmus:  
 SEARCH-SOURCE( $V$ )  
 $v \leftarrow$  select from  $V$  arbitrarily  
 while  $\text{Adj}^T[v] \neq \emptyset$  do  
 $e=(v,w) \leftarrow \text{HEAD}(\text{Adj}^T[v])$   
 $v \leftarrow w$   
 SEARCH-SOURCE() terminiert, da jeder Knoten aus  $V$  maximal einmal besucht wird (beim zweiten Besuch hätten wir einen Kreis detektiert).



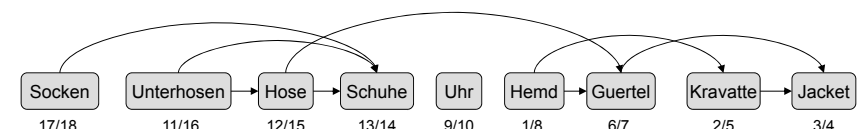
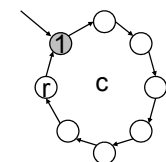
## Topologisches Sortieren: Existenz in DAGs

- Beweis: Teil 2 ( $\Leftarrow$ ): (Fortsetzung)  
 Beweis durch Induktion über die Anzahl der Knoten  $|V|$ :  
 Induktionsanfang: Falls  $|V| = 1$ , ist  $\pi(v) = 1$  eine gültige topologische Sortierung.  
 Induktionsschritt: Sei  $u \in V$  mit  $\text{indeg}(u) = 0$ ,  $V' = V \setminus \{u\}$  und  $G' = (V', E')$  der durch  $V'$  induzierte Subgraph.  $G'$  ist offensichtlich ein DAG mit  $|V|-1$  Knoten. Nach Induktionsvoraussetzung existiert eine top. Sortierung  $\pi'$ .  
 Sei  $\pi: V \rightarrow \{1, \dots, |V|\}$  definiert als:
 
$$\pi(v) = \begin{cases} 1 & : v = u \\ 1 + \pi'(v) & : v \in V' \end{cases}$$
  - ◆ Für  $e = (u, v) \in E$  gilt:  $\pi(u) = 1 < 1 + \pi'(v) = \pi(v)$ .
  - ◆ Für  $e = (v, w) \in E$  mit  $v \neq u$  gilt:  $\pi(v) = 1 + \pi'(v) < 1 + \pi'(w) = \pi(w)$ , da  $(v, w) \in E'$  und  $\pi'$  eine top. Sortierung von  $G'$  ist.
  - ◆ Kanten  $e = (v, u)$  existieren nicht, da  $\text{indeg}(u) = 0$ .
 Folglich ist  $\pi$  eine top. Sortierung für  $G$ .
- In einem DAG werden Knoten mit  $\text{indeg}()=0$  **Quellen** und mit  $\text{outdeg}()=0$  **Senken** genannt.



## Topologisches Sortieren: Erkennung von DAGs

- **Lemma 22.11:** Ein ungerichteter Graph  $G=(V,E)$  ist ein DAG, genau dann wenn ein DFS-Durchlauf keine Rückkante detektiert.
  - Beweis: Teil 1 ( $\Rightarrow$ ): Sei  $G$  ein DAG. Angenommen es gibt eine Kante  $e = (u, v)$  mit  $\text{type}[(u, v)] = \text{BACK}$ .  $v$  ist ein Vorgänger von  $u$ , daher existiert ein Pfad von  $v$  nach  $u$  und mit  $e$  ein Kreis.
  - Teil 2 ( $\Leftarrow$ ): Sei  $c = (v_1, \dots, v_r, v_{r+1})$  ein Kreis und  $v_1$  der Knoten, der zuerst in einem DFS-Durchlauf entdeckt wird. Zum Zeitpunkt  $d[v_1]$  gilt  $\text{color}[v_i] = \text{WEISS}$  für  $i=2, \dots, r$ . Nach dem Theorem der weißen Pfade sind  $v_2, \dots, v_r$  Nachfolger von  $v_1$ . Wenn  $v_r$  abgearbeitet wird, gilt  $\text{color}[v_1] = \text{GRAU}$ , somit ist  $(v_r, v_1)$  eine Rückkante.



■ **TOPOLOGICAL-SORT(G)**

```

1 for each  $u \in V[G]$  do
2    $\text{color}[u] \leftarrow \text{WEISS}; \pi[u] \leftarrow \text{NIL}$ 
4  $\text{time} \leftarrow 0; L \leftarrow \text{LIST-INIT}()$ 
5 for each  $u \in V[G]$  do
6   if  $\text{color}[u] = \text{WEISS}$  then DFS-VISIT( $u, L$ )
7 return  $L$ 
    
```

■ **DFS-VISIT( $u, L$ )**

```

1  $\text{color}[u] \leftarrow \text{GRAU}$  // Entdeckung von  $u$ 
2  $\text{time} \leftarrow \text{time} + 1; d[u] \leftarrow \text{time}$ 
4 for each  $v \in \text{Adj}[u]$  do // Sondierung der Adjazenzliste
5   if  $\text{color}[v] = \text{WEISS}$  then
6      $\pi[v] \leftarrow u$ 
7     DFS-VISIT( $v, L$ )
8   else if  $\text{color}[v] = \text{GRAU}$  then ERROR „G is not a DAG!“
9  $\text{color}[u] \leftarrow \text{SCHWARZ}; \text{LIST-INSERT}(L, u)$ 
10  $\text{time} \leftarrow \text{time} + 1; f[u] \leftarrow \text{time}$  // Vollständige Abarbeitung von  $u$ 
    
```



■ **Theorem 22.12:** Korrektheit von TOPOLOGICAL-SORT()

Sei  $G=(V,E)$  ein DAG. TOPOLOGICAL-SORT( $G$ ) berechnet in  $L$  eine topologische Sortierung der Knoten in  $V$ .

- Beweis:  $L$  enthält alle Knoten  $u \in V$  in der Reihenfolge fallender Endzeiten  $f[u]$  (Knoten werden nach Bearbeitung jeweils vorne in  $L$  eingefügt).

Sei  $e = (u,v) \in E$  eine beliebige Kante. Wir zeigen  $f[u] > f[v]$  mit dem Klammerungstheorem (Theorem 22.7):

- ◆ Fall 1:  $\text{type}[(u,v)] = \text{TREE}$

$d[u] < d[v] < f[u] \Rightarrow f[u] > f[v]$ .

- ◆ Fall 2:  $\text{type}[(u,v)] = \text{FORWARD}$  oder  $\text{type}[(u,v)] = \text{CROSS}$

Zum Zeitpunkt der Sondierung von  $e$  ist  $\text{color}[v] = \text{SCHWARZ}$  und  $f[u]$  noch nicht gesetzt  $\Rightarrow f[u] > f[v]$

- ◆ Fall 3:  $\text{type}[(u,v)] = \text{BACK}$

Dieser Fall tritt nach auf, da  $G$  ein DAG ist (Lemma 22.11).

Bzgl. der Ordnung  $\pi$  in  $L$  gilt somit für jede Kante  $(u,v) \in E$ :  $\pi(u) < \pi(v)$ .



## 5.4 Starke Zusammenhangskomponenten

■ **Definition:**

$u \rightarrow v$  beschreibt die Relation „Es existiert ein Pfad von  $u$  nach  $v$ “.

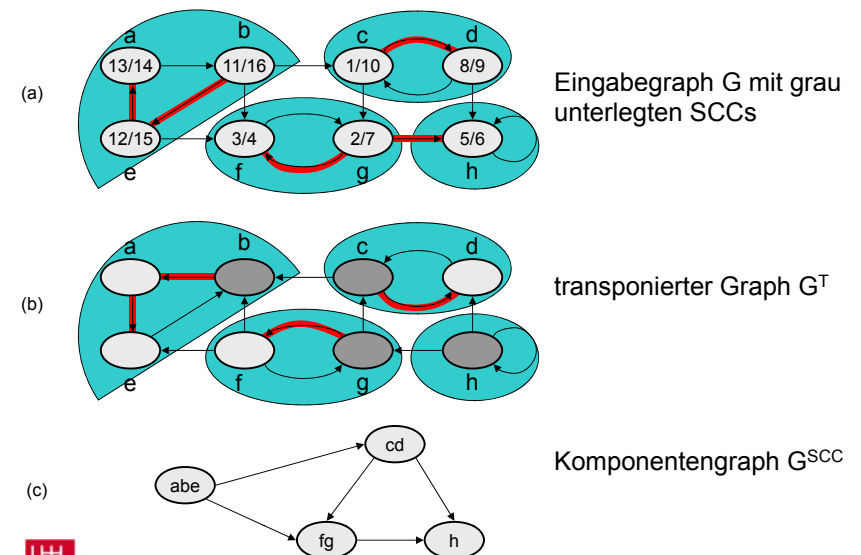
Sei  $G=(V,E)$  ein gerichteter Graph. Eine maximale Menge  $C \subseteq V$ , in der  $\forall u,v \in C: u \rightarrow v$  und  $v \rightarrow u$  gilt, heißt **starke Zusammenhangskomponente** (strongly connected component, SCC).

■ **Lemma 22.x/22.13:** Struktur von SCCs

- Die Knotenmenge  $V$  zerfällt in eine Menge disjunkter starker Zusammenhangskomponenten.
  - ◆ Beweis: Seien  $C_1, C_2$  SCCs mit  $C_1 \cap C_2 \neq \emptyset, v \in C_1 \cap C_2$ . Sei  $u \in C_1$  und  $w \in C_2$  beliebig gewählt. Dann gilt  $u \rightarrow v \rightarrow w$  und  $w \rightarrow v \rightarrow u$  und somit  $C_1 = C_2$ .
- Die Graphen  $G=(V,E)$  und  $G^T=(V,E^T)$  mit  $E^T = \{(u,v) : (v,u) \in E\}$  haben die gleichen starken Zusammenhangskomponenten.
- Der **Komponentengraph**  $G^{SCC}=(V^{SCC}, E^{SCC})$  ist definiert als  $V^{SCC} =$  Menge der SCCs,  $E^{SCC} = \{(C,D) \mid \exists u \in C \exists v \in D: (u,v) \in E\}$ .  $G^{SCC}$  ist ein DAG.
  - ◆ Beweis: analog zu 1.



## Starke Zusammenhangskomponenten: Ein Beispiel





## Starke Zusammenhangskomponenten: Berechnung

```

STRONGLY-CONNECTED-COMPONENTS(G)
  1 for each  $u \in V[G]$  do  $\text{color}[u] \leftarrow \text{WEISS}$ 
  2  $\text{time} \leftarrow 0$ 
  3 for each  $u \in V[G]$  do
  4   if  $\text{color}[u] = \text{WEISS}$  then  $\text{DFS-VISIT}(u, \text{fv}, \text{NIL}, 0)$ 

  5  $\text{INVERT-EDGES}(G)$  // berechnet  $E^T$  aus  $E$ 

  6 for each  $u \in V[G]$  do  $\text{color}[u] \leftarrow \text{WEISS}$ 
  7  $\text{scc\_no} \leftarrow 0$ 
  8 for  $i \leftarrow |V[G]|$  down to 1 do
  9   if  $\text{color}[\text{fv}[i]] = \text{WEISS}$  then
  10     $\text{scc\_no} \leftarrow \text{scc\_no} + 1$ 
  11     $\text{DFS-VISIT}(\text{fv}[i], \text{NIL}, \text{scc}, \text{scc\_no})$ 
  12 return(  $\text{scc}$  )
  
```

**DFS-Lauf 1:** Ordne Knoten nach Endzeit  $f[u] \rightarrow \text{fv}[i]$

**DFS-Lauf 2** (in Reihenfolge  $f[u]$ ): Bestimmung der SCCs



## Starke Zusammenhangskomponenten: Berechnung

```

DFS-VISIT( $u, \text{fv}, \text{scc}, \text{scc\_no}$ )
  1  $\text{color}[u] \leftarrow \text{GRAU}$ 
  2 for each  $v \in \text{Adj}[u]$  do
  3   if  $\text{color}[v] = \text{WEISS}$  then  $\text{DFS-VISIT}(v)$ 
  4  $\text{color}[u] \leftarrow \text{SCHWARZ}$ 
  5 if  $\text{fv} \neq \text{NIL}$  then  $\text{time} \leftarrow \text{time} + 1; \text{fv}[\text{time}] \leftarrow u$ 
  6 else  $\text{scc}[u] \leftarrow \text{scc\_no}$ 

INVERT-EDGES( $G$ )
  1 for each  $v \in V[G]$  do
  2   for each  $u \in \text{Adj}[v]$  do  $\text{LIST-INSERT}(\text{AdjT}[u], v)$ 
  3 for each  $v \in V$  do  $\text{Adj}[v] \leftarrow \text{AdjT}[v]$ 
  
```

**DFS-Lauf 1:** Ordne Knoten nach Endzeit  $f[u]$

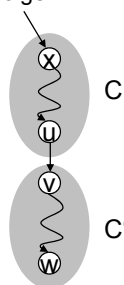
**DFS-Lauf 2** (in Reihenfolge  $f[u]$ ): Bestimmung der SCCs

**STRONGLY-CONNECTED-COMPONENTS(G)** benötigt asymptotisch  $O(N+M)$  Zeit.



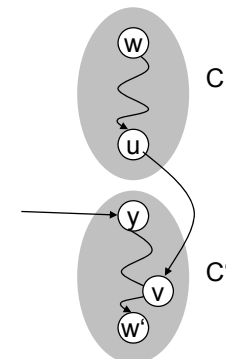
## Starke Zusammenhangskomponenten: Korrektheit

- Im folgenden:
  - beziehen sich  $d[]$  und  $f[]$  auf die Entdeckungs- und Endzeiten des ersten DFS-Laufs.
  - sei für  $U \subseteq V$  definiert:  $d(U) = \min_{u \in U} d[u]$  und  $f(U) = \max_{u \in U} f[u]$
- Lemma 22.14:** (SCCs und Endzeiten  $f[]$ )  
 Seien  $C, C'$  zwei SCCs des Graphen  $G=(V,E)$  mit  $C \neq C'$ . Für alle Kanten  $(u,v) \in E$  mit  $u \in C$  und  $v \in C'$  gilt:  $f(C) > f(C')$ .
  - Beweis:** Fallunterscheidung nach Entdeckungsreihenfolge
    - Fall 1:  $d(C) < d(C')$ :  
 Sei  $x \in C$  mit  $d[x] = d(C)$ . Zum Zeitpunkt  $d[x]$  gilt: Für jeden Knoten  $w \in C'$  existiert ein weißer Pfad  $x \rightarrow u \rightarrow v \rightarrow w$ .  
 $\Rightarrow w$  ist Nachfahre von  $x$  (Theorem der weißen Pfade)  
 $\Rightarrow f[x] = f(C) > f(C')$  (Klammerungstheorem)



## Starke Zusammenhangskomponenten: Korrektheit

- Beweis von Lemma 22.14 (Fortsetzung)**
  - Fall 2:  $d(C) > d(C')$   
 Sei  $y \in C'$  mit  $d[y] = d(C')$ . Zum Zeitpunkt  $d[y]$  gilt: Für jeden Knoten  $w' \in C'$  existiert ein weißer Pfad  $y \rightarrow w'$ .  
 $\Rightarrow w'$  ist Nachfahre von  $y$  (Theorem der weißen Pfade)  
 und  $f(C') = f[y]$  (Klammerungstheorem)  
 Da  $d(C) > d(C')$ , gilt zum Zeitpunkt  $d[y]$ :  
 $\forall w \in C: \text{color}[w] = \text{WEISS}$ .  
 $G^{\text{SCC}}$  ist ein DAG und  $(C, C') \in E^{\text{SCC}}$   
 $\Rightarrow (C', C) \notin E^{\text{SCC}}$ , d.h.  $\forall r \in C', s \in C: r \rightarrow s$  ist falsch.  
 $\Rightarrow$  Zum Zeitpunkt  $f[y]$  gilt:  
 $\forall w \in C: \text{color}[w] = \text{WEISS}$   
 $\Rightarrow f(C) > d(C) > f[y] = f(C')$  (Klammerungstheorem)
- Korollar 22.15:** Für Kanten  $(u,v) \in E^T$  mit  $u \in C, v \in C'$  gilt  $f(C) < f(C')$



■ **Theorem 22.16:** Korrektheit des SCC-Algorithmus

Sei  $G=(V,E)$  ein gerichteter Graph, dann bestimmt der Algorithmus STRONGLY-CONNECTED-COMPONENTS() die starken Zusammenhangskomponenten von  $G$ .

- Beweis: (durch vollständige Induktion über Schleife Zeile 8-11)

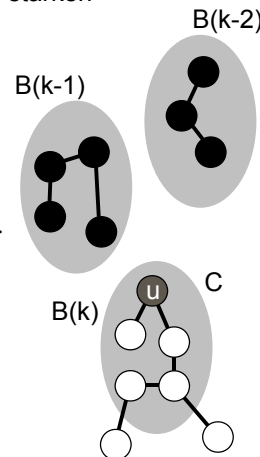
Annahme: der im  $k$ -ten Aufruf von DFS-VISIT() erzeugte DFS-Baum  $B(k)$  ist eine SCC in  $G$ .

Induktionsschritt: Sei  $u$  die Wurzel des  $k$ -ten DFS-Baums  $B(k)$ , sei  $C$  eine SCC in  $G$  mit  $u \in C$  und sei  $X$  der Zeitpunkt, in dem in der Schleife <Zeile 8-11>  $u$  gewählt wird.

**Teil 1:  $C \subseteq B(k)$ :** Zum Zeitpunkt  $X$  existieren keine grauen Knoten. Alle schwarzen Knoten gehören nach Induktionsannahme zu anderen SCCs.

$\Rightarrow \forall w \in C$ : color[w] = WEISS und es existiert  $u \rightarrow w$  in  $C$ , da  $C$  eine SCC ist.

$\Rightarrow w \in B(k)$  (Theorem der weißen Pfade)



- Beweis Theorem 22.16: (Fortsetzung)

**Teil 2:  $C \supseteq B(k)$**

Sei  $w \in B(k)$ . Angenommen, es sei  $w \in C'$ ,  $C' \neq C$ . Sei  $w$  so gewählt, dass  $\delta(u,w)$  minimal ist.

♦  $\forall i \in \{1, \dots, k-1\}$ : Zum Zeitpunkt  $X$  sind alle Knoten aus  $B(i)$  bereits schwarz,  $w$  jedoch weiß.

$\Rightarrow C' \neq B(i) \forall i \in \{1, \dots, k-1\}$

♦  $w$  ist Nachfahre von  $u$ , da  $w \in B(k)$  ist.

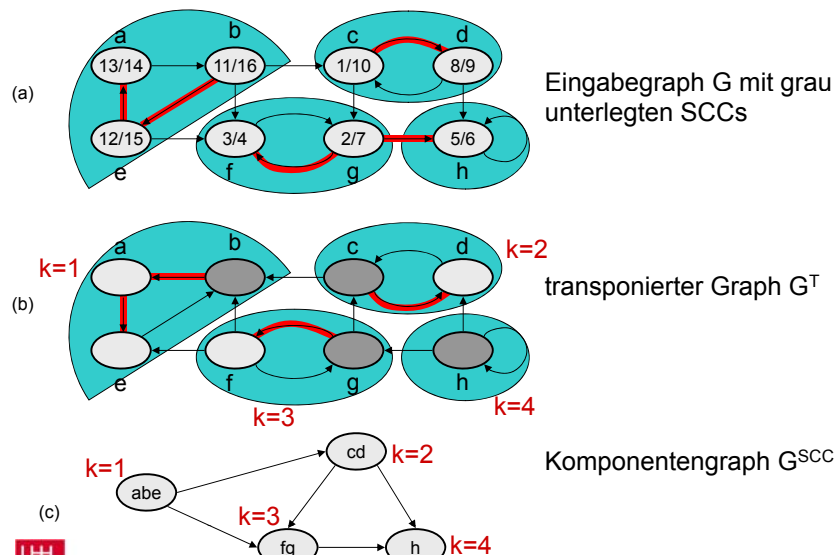
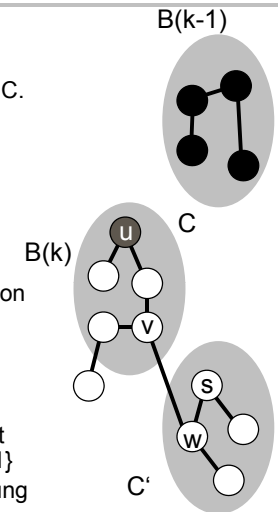
$\Rightarrow$  Zum Zeitpunkt  $X$  existiert ein weißer Pfad von  $u$  nach  $w$ . Sei  $e=(v,w)$  die letzte Kante des Pfades. Aufgrund der Wahl von  $w$  ist  $v \in C$ .  $(v,w) \in E^T \Rightarrow f(C) < f(C')$  (Korollar 22.15)

$\Rightarrow$  Sei  $s \in C'$  mit  $f[s] = f(C')$ .

Dann folgt  $f[u] \leq f(C) < f(C') = f[s]$ . Zudem gilt color[s] = WEISS, da  $C' \neq B(i) \forall i \in \{1, \dots, k-1\}$

$\Rightarrow$  Zum Zeitpunkt  $X$  wird aufgrund der Sortierung  $s$  statt  $u$  gewählt.

$\Rightarrow w$  existiert nicht, d.h.  $C \supseteq B(k)$

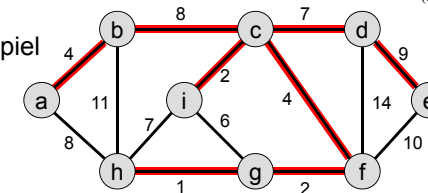


- MST-Problem (minimal spanning tree):

■ geg: (ungerichteter) Graph  $G=(V,E)$  mit Kantengewichten  $w:E \rightarrow \mathbb{R}$

■ ges: Baum  $(V,T)$  mit  $T \subseteq E$  und  $w(T) = \sum_{(u,v) \in T} w((u,v))$  minimal

- Beispiel



- $(V,T)$  wird als **(minimaler) Spannbaum** bezeichnet.

- Das MST-Problem ist ein **Optimierungsproblem**: Aus einer Menge gültiger Lösungen wird die mit optimalen Kosten gesucht.

- **Greedy-Algorithmen**:

■ Lösung wird durch Einzelentscheidungen aufgebaut, die – einmal getroffen – nie wieder zurückgenommen werden.

■ häufig keine Optimalitätsgarantie

## Greedy-Algorithmen für das MST-Problem

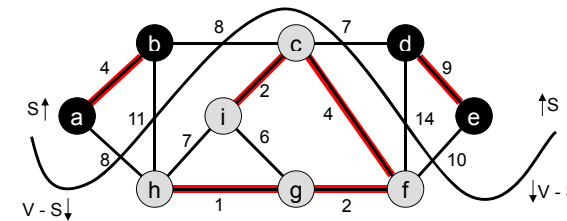
- (Greedy-)Strategie:
  - Aufbau des MST durch sukzessives Hinzunehmen von Kanten
- Definition (sichere Kante):
  - Sei  $A$  eine Teilmenge eines MST. Die Kante  $(u,v)$  ist für  $A$  **sicher**, g.d.w.  $A \cup \{(u,v)\}$  ist eine Teilmenge eines MST
- Anzahl Kanten:
  - Ein Spannbaum über  $G=(V,E)$  hat  $|V|-1$  Kanten.
- GENERIC-MST( $G, w$ )
 

```

A ← ∅
while |A| < |V|-1 do
    e ← FIND-SAVE-EDGE(G, A)
    A ← A ∪ {e}
return A
            
```
- Korrektheit von GENERIC-MST:
  - folgt direkt aus der Definition der sicheren Kanten.



## Theorem der sicheren Kanten



- Definition (Schnitt, etc.):
  - Ein **Schnitt**  $(S, V - S)$  ist eine Partition der Knotenmenge,  $S \subseteq V$ .
  - Eine Kante  $e=(u,v)$  **kreuzt** den Schnitt, falls  $u \in S, v \in V - S$  gilt,
  - sonst **respektiert** die Kante  $e=(u,v)$  den Schnitt.
  - Eine  $S$  kreuzende Kante  $e$  heißt **leichte Kante**, g.d.w.  $w(e)$  unter allen  $S$  kreuzenden Kanten minimal ist.



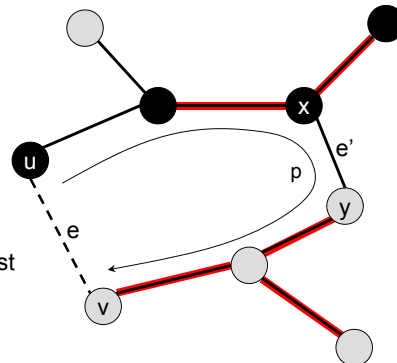
## Theorem der sicheren Kanten

- **Theorem 23.1** (sichere Kanten):
 

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter Graph,  $w: E \rightarrow \mathbb{R}$  eine Gewichtsfunktion, sei  $A$  Teilmenge eines MST. Sei  $(S, V-S)$  ein Schnitt, der  $A$  respektiert und  $e$  eine leichte Kante die  $(S, V-S)$  kreuzt. Dann ist  $e$  eine sichere Kante.

### ■ Beweis:

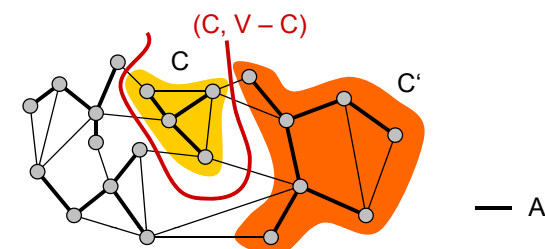
- Sei  $T$  ein MST, mit  $A \subseteq T$  und  $(S, V-S)$  und  $e=(u,v)$  wie im Theorem gefordert. Angenommen,  $e \notin T$ :
- Es gibt einen eindeutigen Kreis, der außer  $e$  nur Kanten aus  $T$  enthält.
- Es gibt eine Kante  $e'=(x,y) \in T$ , die  $(S, V-S)$  kreuzt;  $e' \notin A$
- Sei  $T' = T \setminus \{e'\} \cup \{e\}$ . Offensichtlich ist  $T'$  ein Spannbaum
- $w(T') = w(T) - w(e') + w(e) \leq w(T)$  da  $w(e) \leq w(e')$  ( $e$  ist leichte Kante)
- $A \subseteq T'$  und  $e \in T' \Rightarrow e$  ist für  $A$  sicher.



## Theorem der sicheren Kanten

- Aus dem Theorem der sicheren Kanten folgt die Korrektheit der Greedy-Strategie:
- Korollar 23.2:
 

Sei  $G=(V,E)$  ein ungerichteter, zusammenhängender Graph,  $w: E \rightarrow \mathbb{R}$  eine Gewichtsfunktion, sei  $A$  Teilmenge eines MST.  $A$  definiert den Wald  $G_A = (V,A)$ . Sei  $C$  eine Zusammenhangskomponente in  $G_A$  und  $e$  eine leichte Kante, die  $C$  mit einer anderen Komponente  $C'$  verbindet. Dann ist  $e$  für  $A$  sicher.
- Beweis: Der Schnitt  $(C, V - C)$  respektiert  $A$  und  $e$  ist eine leichte Kante für  $(C, V - C)$ . Somit ist  $e$  für  $A$  sicher.



## Kruskals Algorithmus

- Idee:
  - A beschreibt einen Wald, mit jeder Kante werden zwei Bäume zu einem verschmolzen.
  - Zu Beginn haben wir  $|V|$  Bäume mit je einem Knoten.
  - Kanten werden mit aufsteigendem Gewicht eingefügt.
  - Schnitt trennt jeweils einen Baum vom Rest des Graphen.
- Kruskals Algorithmus benötigt eine Datenstruktur, die für je zwei Knoten effizient entscheidet, ob diese zum gleichen Baum gehören oder nicht:
- **Disjoint-Set** Datenstruktur:
  - MAKE-SET: erstellt eine Menge
  - UNION: vereinigt zwei disjunkte Mengen
  - FIND-SET: liefert ein repräsentatives Element einer Menge



## Kruskals Algorithmus

### MST-KRUSKAL( $G, w$ )

```

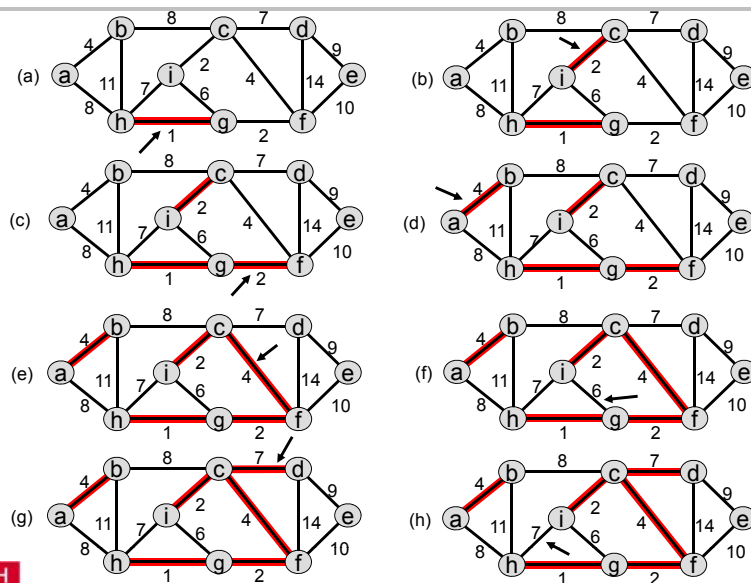
A ← ∅
for each v ∈ V[G] do
    MAKE-SET(v)
SORT-EDGES-BY-INCREASING-WEIGHT(E[G])
for each (u,v) ∈ E[G] (ordered by increasing weight) do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A ← A ∪ {(u,v)}
        UNION(u,v)
return A
    
```

- Korrektheit: Aufgrund der aufsteigenden Sortierung ist  $(u,v)$  eine leichte Kante. Die Korrektheit folgt somit direkt aus Korollar 23.2.
- Laufzeit: Seien  $T_{\text{make-set}}, T_{\text{find-set}}, T_{\text{union}}$  die Laufzeiten der Operationen der Disjoint-Set-Datenstruktur, dann gilt:  

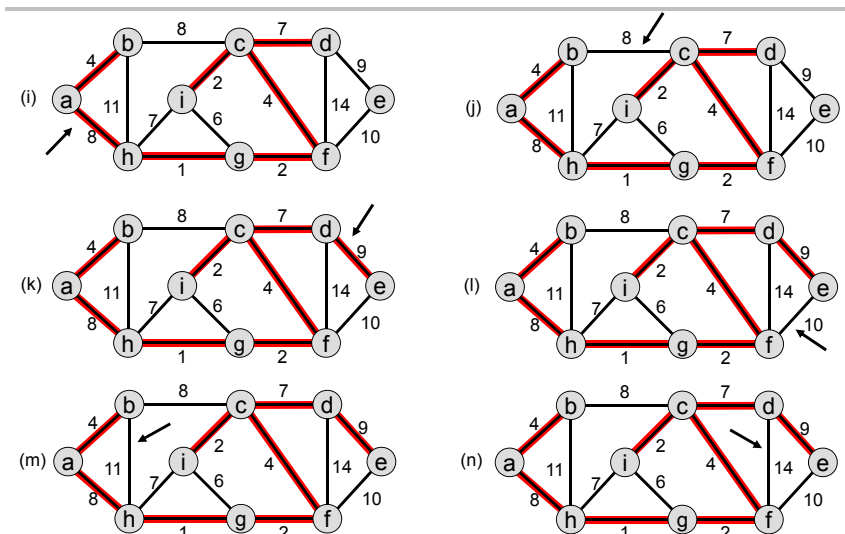
$$T_{\text{Kruskal}}(N, M) = N * T_{\text{make-set}} + O(M \log N) + 2M * T_{\text{find-set}} + (N-1) * T_{\text{union}}$$



### Beispiel: Kruskals Algorithmus



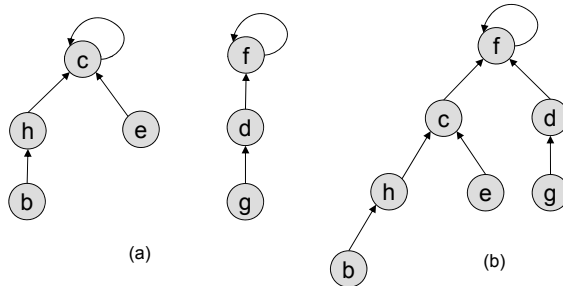
### Beispiel: Kruskals Algorithmus



## Eine Datenstruktur für disjunkte Mengen (Union-Find Datenstruktur)

### Idee:

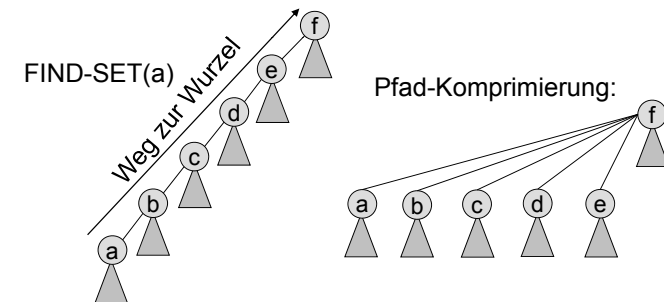
- Menge wird durch einen gewurzelten Baum repräsentiert
- Repräsentant ist die Wurzel des Baums
- FIND-SET: gibt die Wurzel des Baums aus
- UNION(c,f): hängt einen Baum c unter einen anderen Baum f



## Die Union-Find Datenstruktur

### Heuristiken

- union-by-rank:** (UNION-Operation)
  - hängen den flacheren Baum unter den tieferen
- path-compression:** (FIND-SET-Operation)
  - hängen alle Teilbäume auf dem Weg zur Wurzel unter die Wurzel



## Die Union-Find Datenstruktur

### Datenstruktur:

- $p[x]$ : Vorgänger / Elter-Knoten
- $rank[x]$ : Obere Schranke für die Höhe des Teilbaums unter x

### MAKE-SET(x) // Initialisierung

$p[x] \leftarrow x$   
 $rank[x] \leftarrow 0$

### LINK(x, y) // hängt flacheren Teilbaum unter den anderen

if  $rank[x] > rank[y]$  then  $p[y] \leftarrow x$  else  $p[x] \leftarrow y$   
 if  $rank[x] = rank[y]$  then  $rank[y] \leftarrow rank[y] + 1$

### UNION(x, y) // Vereinigung zweier Mengen, die durch je

// einen Repräsentanten gegeben sind

LINK( FIND-SET(x), FIND-SET(y) )



## Die Union-Find Datenstruktur

### Rekursives FIND-SET:

FIND-SET(x)  
 if  $x \neq p[x]$  then  
      $p[x] \leftarrow \text{FIND-SET}(p[x])$   
 return  $p[x]$

### Iteratives FIND-SET:

FIND-SET(x)  
 $r \leftarrow x$  // 1. Suche die Wurzel  
 while  $r \neq p[r]$  do  $r \leftarrow p[r]$   
 while  $x \neq r$  do // 2. Pfad-Kompression  
      $px \leftarrow p[x]; p[x] \leftarrow r; x \leftarrow px$   
 return r

### FIND-SET mit Pfad-Kompression korrigiert $rank[x]$ nicht!

### Amortisierte Laufzeit für

$\left. \begin{array}{l} \text{N MAKE-SET-Operationen} \\ \text{N-1 UNION-Operationen} \\ \text{M FIND-SET-Operationen} \end{array} \right\} T_{\text{UNION-FIND}}(N, M) = O(M \alpha(N))$   
 $\alpha()$  ist die Inverse der Ackermann-Funktion,  $\alpha(x) \leq 4 \quad \forall x \leq 10^{80}$

### Laufzeit Kruskals MST-Algorithmus:

$$T_{\text{Kruskal}}(N, M) = N * T_{\text{make-set}} + O(M \log N) + 2M * T_{\text{find-set}} + (N-1) * T_{\text{union}} \\ = O(M \log N) + O(M \alpha(N)) = O(M \log N)$$



## Prims Algorithmus

### Idee:

- A ist zunächst leer
- Der Algorithmus beginnt an einem beliebig wählbaren Startknoten r
- In jeder Iteration wird ein zusammenhängender minimaler (Teil-) Spannbaum um eine Kante erweitert.
- Eine Prioritätswarteschlange Q speichert alle Knoten, die noch nicht durch A verbunden sind mit Priorität nach min. Kantengewicht, um sie zu verbinden
- $\pi$  speichert eine Vorgänger-Relation (wie bei der Breitensuche)
- Die Menge A ist implizit definiert:  $A = \{ (v, \pi(v)) \mid v \in V - \{r\} - Q \}$



## Prims Algorithmus

### MST-PRIM(G, w, r)

```

for each  $u \in V[G]$  do
     $\text{key}[u] \leftarrow \infty$ 
     $\pi[u] \leftarrow \text{NIL}$ 
 $\text{key}[r] \leftarrow 0$ 
 $Q \leftarrow V[G]$ 
while  $Q \neq \emptyset$  do
     $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
    for each  $v \in \text{Adj}[u]$  do
        if  $v \in Q$  and  $w(u,v) < \text{key}[v]$  then
             $\pi[v] \leftarrow u$ 
             $\text{key}[v] \leftarrow w(u,v)$ 
            DECREASE-KEY( $Q, v, w(u,v)$ )
    
```

### key[u]:

- Entfernung von u zum Baum A, falls u eine Kante von A entfernt ist,  $\infty$  sonst

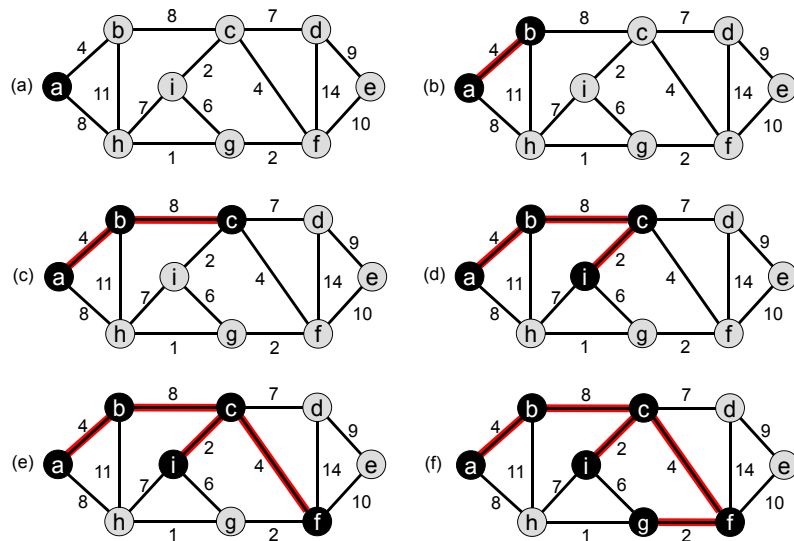
### $\pi[u]$ :

- Vater von u im MST
- $A = \{ (u, \pi[u]) \mid u \in V - \{r\} \}$

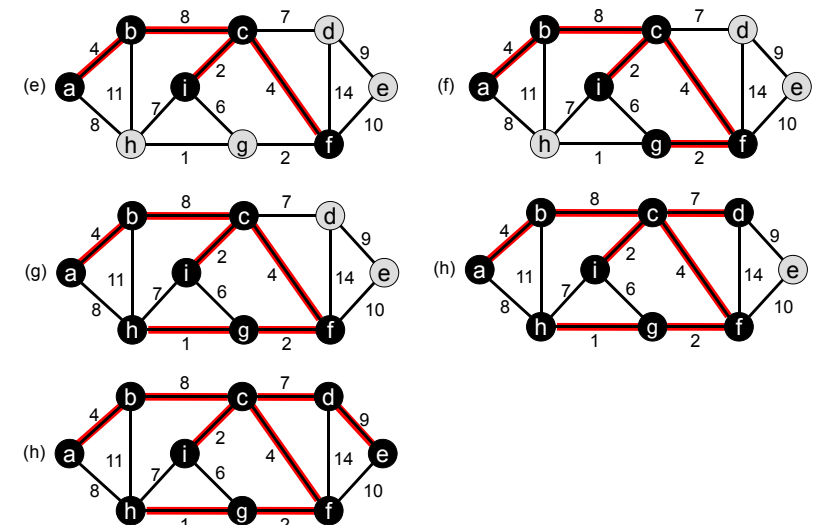
ACHTUNG: In Cormen et al wird ein implizites DECREASE angenommen.



## Beispiel: Prims Algorithmus



## Beispiel: Prims Algorithmus



- MST-PRIM(G, w, r)
  - 1 for each  $u \in V[G]$  do
  - 2     $key[u] \leftarrow \infty$
  - 3     $\pi[u] \leftarrow NIL$
  - 4     $key[r] \leftarrow 0$
  - 5     $Q \leftarrow V[G]$
  - 6 while  $Q \neq \emptyset$  do
  - 7     $u \leftarrow EXTRACT-MIN(Q)$
  - 8    for each  $v \in Adj[u]$  do
  - 9     if  $v \in Q$  and  $w(u,v) < key[v]$  then
  - 10       $\pi[v] \leftarrow u$
  - 11       $key[v] \leftarrow w(u,v)$
  - 12    DECREASE-KEY(Q, v,  $w(u,v)$ )
- Laufzeitanalyse ( $N = |V|$ ,  $M = |E|$ )
  - Initialisierung (1-5):  $O(N)$
  - Schleife 6-12:  $N$  Durchläufe
  - Schleife 8-12: insgesamt  $2M$  Durchläufe
  - Priority-Queue: binärer MIN-Heap mit zusätzlichem Zeigerfeld zum Auffinden von  $v$  in  $Q$ :  
EXTRACT-MIN:  $O(\log N)$   
DECREASE-KEY:  $O(\log N)$

$$T_{PRIM}(N,M) = O(M \log N)$$

## Korrektheit:

Wir betrachten den Schnitt  $(Q, V - Q)$ . Wenn  $u$  aus  $Q$  entfernt wird, ist  $(u, \pi[u])$  eine leichte Kante. Die Korrektheit folgt dann aus Korollar 23.2.



## Problem:

- geg: Graph  $G=(V,E)$ , Kantengewichte  $w:E \rightarrow \mathbb{R}$
- ges: kürzeste Pfade (bzw. Wege) zwischen Knoten aus  $V$ ,
  - ◆ Länge oder Gewicht eines Pfades  $(v_0, v_1, \dots, v_k): w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$
  - ◆ Gewicht des kürzesten Pfades:
 
$$\delta(u, v) = \begin{cases} \min \{w(p) : u \xrightarrow{p} v\} & \text{falls ein Pfad von } u \text{ nach } v \text{ existiert} \\ \infty & \text{sonst} \end{cases}$$

## Kantengewichte:

- Distanzen, Zeit, Kosten, Erfolgswahrscheinlichkeiten, Verlust

## Problemvarianten:

- single-source: kürzeste Pfade von einem Knoten zu allen anderen
- single-pair: kürzester Pfad zw. zwei ausgewählten Knoten
- all-pairs: kürzeste Pfade zwischen allen Knotenpaaren

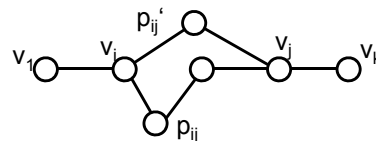


# Eigenschaften kürzester Pfade

## Lemma 24.1 (Optimale Teilstruktur):

Sei  $G = (V,E)$  ein gewichteter, gerichteter Graph,  $w: E \rightarrow \mathbb{R}$  eine Gewichtsfunktion. Sei  $p = (v_1, v_2, \dots, v_k)$  ein kürzester Pfad von  $v_1$  zu  $v_k$ . Für alle  $1 \leq i \leq j \leq k$  gilt:  $p_{ij} = (p_i, p_{i+1}, \dots, p_j)$  ist ein kürzester Pfad von  $v_i$  nach  $v_j$ .

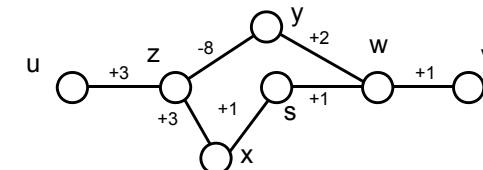
- Beweis: Angenommen es existiert ein Pfad  $p'_{ij}$  von  $p_i$  nach  $p_j$  mit  $p'_{ij} \neq p_{ij}$  und  $w(p'_{ij}) < w(p_{ij})$ . Dann gilt für  $p' = (v_1, \dots, v_{i-1}, p'_{ij}, v_{j+1}, \dots, v_k)$ :  $w(p') < w(p)$ . Somit ist  $p$  kein kürzester Pfad. ⚡



# Eigenschaften kürzester Pfade

## Was geschieht bei Kanten mit negativen Kantengewichten?

- $\delta(u,v)$  ist im Prinzip auch für negative Kantengewichte definiert
- Angenommen, es gibt auf dem Pfad von  $u$  nach  $v$  einen Zyklus mit negativer Gesamtlänge:



$$w((u,z,x,s,w,v)) = 9 \quad w((u,z,y,w,v)) = -2$$

$$w((u,z,y,w,s,x,z,y,w,v)) = -3$$

Mit jedem Durchlauf des negativen Zyklus verkürzt sich der Pfad!

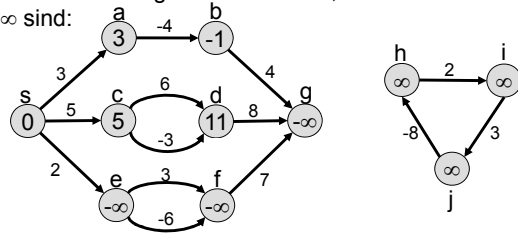
- per Definition:  $\delta(u,v) = -\infty$ , falls es auf einen Pfad von  $u$  nach  $v$  einen Zyklus negativer Gesamtlänge gibt.





## Eigenschaften kürzester Pfade

- Ein Zyklus negativer Gesamtlänge bedeutet nicht, dass alle Distanzen in einem Graphen  $-\infty$  sind:



- Falls  $|\delta(u,v)| < \infty$ , existiert ein kürzester Pfad mit höchstens  $|V|-1$  Kanten.  
oder: Falls  $|\delta(u,v)| < \infty$ , existiert ein kürzester Pfad ohne Zyklen.
  - Annahme: ein kürzester Pfad  $p$  von  $u$  nach  $v$  enthält  $|V|$  Kanten
    - $\Rightarrow p$  enthält  $|V|+1$  Knoten
    - $\Rightarrow$  in  $p$  kommt mindestens ein Knoten doppelt vor
    - $\Rightarrow p$  enthält einen Zyklus  $c$ :
      - $w(c) < 0$ :  $\delta(u,v) = -\infty$
      - $w(c) = 0$ :  $c$  kann aus  $p$  gelöscht werden, ohne dass sich  $w(p)$  verändert
      - $w(c) > 0$ : durch Löschen von  $c$  entsteht ein Pfad  $p'$  mit  $w(p') < w(p)$



## Repräsentation kürzester Pfade

### Ziel:

- Speicherung eines kürzesten Pfades von einem Startknoten  $s$  zu allen anderen Knoten
- Speicherung aller kürzesten Pfadlängen

### Methode:

- bzgl. eines Zielknotens  $s$ , haben alle Knoten  $v$  einen Vorgänger  $\pi[v]$  auf dem kürzesten Pfad zu  $s$  (Lemma 24.1):  
Vorgängerteilgraph  $G_\pi = (V_\pi, E_\pi)$  mit:
 
$$V_\pi = \{v \in V \mid \pi[v] \neq NIL\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi - \{s\}\}$$

$\Rightarrow$  Jeder Knoten hat bzgl. eines kürzesten Pfades maximal einen Vorgänger, d.h. kürzesten Pfade können als Baumstruktur gespeichert werden (Shortest-Paths Tree)

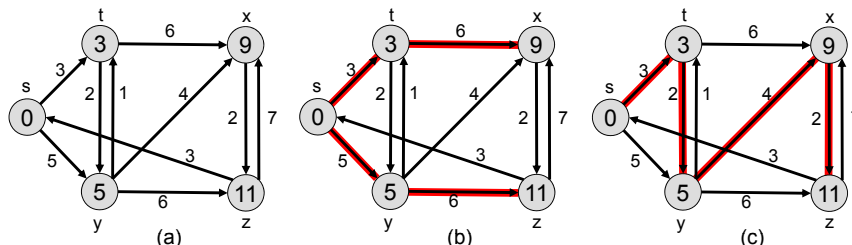
- $d[v]$ : Obere Schranke für  $\delta(s,v)$
- $\delta(s,v)$ : Länge des kürzesten Pfades von  $v$  zu  $s$
- $\pi[v]$ : Vorgänger von  $v$  auf dem kürzesten Pfad zu  $s$



## Repräsentation kürzester Pfade

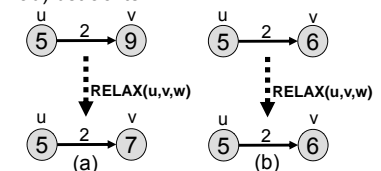
Der Baum kürzester Pfade  $G' = (V', E')$  ist ein gerichteter Teilgraph des Eingabegraphen  $G=(V,E)$  mit

- $V'$  ist die Menge der in  $G$  von  $s$  aus erreichbaren Knoten
- $G'$  bildet einen gerichteten Baum mit der Wurzel  $s$
- Für alle  $v \in V'$  ist der eindeutige, einfache Pfad von  $s$  nach  $v$  in  $G'$  ein kürzester Pfad von  $s$  nach  $v$  in  $G$
- Achtung: Weder der Baum kürzester Pfade noch kürzeste Pfade selbst sind eindeutig.



## Relaxation

- Berechnung kürzester Pfade mittels Relaxation:
  - Attribut  $d[v]$  ist eine obere Schranke für die Länge eines kürzesten Pfades von  $s$  nach  $v$ :  $\delta(s,v)$
  - $d[v]$  wird zunächst mit  $\infty$  initialisiert und anschließend von oben an  $\delta(s,v)$  angenähert
  - Annäherung erfolgt durch sukzessives **Relaxieren** von Kanten
- INITIALIZE-SINGLE-SOURCE( $G,s$ )
  - for each  $v \in V[G]$  do
    - $d[v] \leftarrow \infty$
    - $\pi[v] \leftarrow NIL$
    - $d[s] \leftarrow 0$
- Relaxation einer Kante  $e = (u,v)$ :
  - Test, ob der bisher gefundene kürzeste Pfad zu  $v$  verbessern können, in dem wir  $u$  als Vorgänger von  $v$  (auf dem kürzesten Pfad) betrachten
  - Falls ja, werden  $d[v]$  und  $\pi[v]$  angepasst:
- RELAX( $u, v, w$ )
  - if  $d[v] > d[u] + w(u,v)$  then
    - $d[v] \leftarrow d[u] + w(u,v)$
    - $\pi[v] \leftarrow u$





## ■ Dreiecksungleichung:

$$\forall e=(u,v) \in E: \delta(s,v) \leq \delta(s,u) + w(u,v)$$

## ■ Pfad-Relaxations-Eigenschaft:

Sei  $p=(s=v_0, v_1, v_2, \dots, v_k=v)$  ein kürzester Weg von  $s$  zu  $v$ . Werden alle Kanten  $(v_i, v_{i+1})$  in der Reihenfolge von  $s$  zu  $v$  relaxiert, so gilt  $d[v] = \delta(s,v)$ .

## ■ Allgemeine Eigenschaften der Kürzeste-Pfade Algorithmen:

## ■ Eigenschaft der oberen Schranke (Upper-bound property):

$$d[v] \geq \delta(s,v) \text{ und fällt monoton}$$

## ■ Kein-Pfad-Eigenschaft (No-path property):

Gibt es keinen Pfad von  $s$  zu  $v$ , so gilt  $d[v] = \infty$

## ■ Konvergenzeigenschaft (Convergence property):

Sei  $u$  Vorgänger von  $v$  auf einem kürzesten Pfad von  $s$  zu  $v$ . Gilt  $d[u] = \delta(s,u)$  zu irgend einem Zeitpunkt vor der Relaxation der Kante  $(u,v)$ , so gilt zu jedem Zeitpunkt nach der Relaxation von  $(u,v)$   $d[v] = \delta(s,v)$

## ■ Vorgängerteilgraph-Eigenschaft (Shortest-path-tree property):

Wenn  $d[v] = \delta(s,v)$  gilt, beschreibt der Vorgängerteilgraph einen Baum kürzester Pfade mit Wurzel  $s$

 Achtung: Fehler  
im Cormen S. 590


## ■ Beweis der Dreiecksungleichung:

■ Fall 1: Es gibt einen kürzesten Pfad von  $s$  nach  $v$ 

Der Pfad  $s \rightarrow u \rightarrow v$  existiert und ist mindestens so lang wie der kürzeste Pfad von  $s$  nach  $v$ .

■ Fall 2: Es gibt keinen kürzesten Pfad von  $s$  nach  $v$  und  $\delta(s,v) = \infty$ 

Angenommen,  $\delta(s,u) < \infty$ , dann existiert ein Pfad von  $s$  über  $u$  nach  $v$ .

■ Fall 3: Es gibt keinen kürzesten Pfad von  $s$  nach  $v$  und  $\delta(s,v) = -\infty$ 

Trivial.

## ■ Beweis der Pfad-Relaxations-Eigenschaft:

Induktionsannahme: Nach Relaxation der  $i$ -ten Kante des Pfades  $p$  gilt  $d[v_i] = \delta(s,v_i)$ .

Induktionsanfang ( $i=0$ ): Nach Initialisierung gilt  $d[s] = 0 = \delta(s,s)$

Induktionsschritt: Angenommen, es gilt  $d[v_{i-1}] = \delta(s,v_{i-1})$ . Nach Relaxation der  $i$ -ten Kante  $(v_{i-1}, v_i)$  gilt:  $d[v_i] \leq \delta(s,v_{i-1}) + w(v_{i-1}, v_i) = \delta(s,v_i)$ , da ein kürzester Pfad von  $s$  nach  $v_i$  über  $v_{i-1}$  führt. Da  $d[v_i]$  eine obere Schranke ist, folgt  $d[v_i] = \delta(s,v_i)$



## Bellman-Ford-Algorithmus (allgemeiner Fall)

## ■ Idee:

- Anfang: für alle Knoten gilt:  $d[v] = \infty$
- Die Kanten werden in allen möglichen Reihenfolgen relaxiert:

■ BELLMAN-FORD( $G, w, s$ )

```

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2 for  $i \leftarrow 1$  to  $|V[G]|-1$  do
3   for each edge  $(u,v) \in E[G]$  do
4     RELAX( $u, v, w$ )
5 for each edge  $(u,v) \in E[G]$  do
6   if  $d[v] > d[u] + w(u,v)$  then return FALSE
7 return TRUE

```

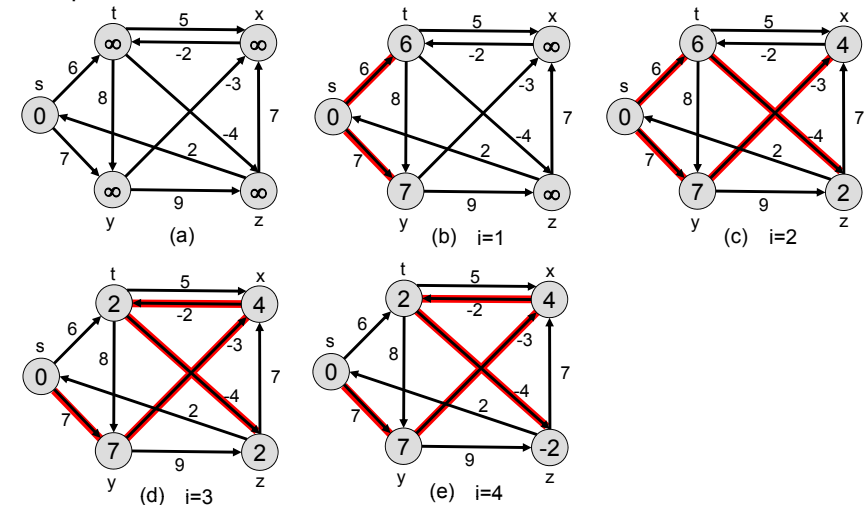
■ Laufzeit:  $O(|V| |E|)$ 

Der Bellman-Ford-Algorithmus erlaubt negative Kantengewichte. Negative Zyklen werden erkannt, das Resultat ist nur korrekt, falls keine negativen Zyklen auftreten.



## Bellman-Ford Algorithmus

## ■ Bsp:



## Korrektheit des Bellman-Ford-Algorithmus

- Sei  $G = (V, E)$  ein gerichteter Graph mit Gewichtsfunktion  $w: E \rightarrow \mathbb{R}$ ,  $s \in V$  ein beliebiger Startknoten.

- **Lemma 24.2:** (Korrekte Berechnung von  $\delta(s, v)$  für  $0 \leq \delta(s, v) < \infty$ )  
Angenommen,  $G$  enthält keine von  $s$  aus erreichbaren negativen Zyklen.  
Für einen von  $s$  aus erreichbaren Knoten  $v$  gilt  $d[v] = \delta(s, v)$ .

**Beweis:** Sei  $p = (s = v_0, v_1, v_2, \dots, v_k = v)$  ein kürzester Pfad von  $s$  nach  $v$ .  
Nach  $i$  Iterationen der Schleife 2-4 wurde die Kantenmenge  $E$   $i$  mal relaxiert.

$\Rightarrow$  Die ersten  $i$  Kanten von  $p$  wurden in der Reihenfolge, wie sie in  $p$  vorkommen, relaxiert.

$\Rightarrow d[v_i] = \delta(s, v_i)$  (Pfad-Relaxations-Eigenschaft)

$p$  hat höchstens  $|V|-1$  Kanten (Zyklenfreiheit kürzester Pfade)

$\Rightarrow d[v] = \delta(s, v)$

- **Korollar 24.3:** (Korrekte Berechnung von  $\delta(s, v)$  für  $\delta(s, v) = \infty$ )

Für jeden Knoten  $v$  gilt:  $s \rightsquigarrow v \Leftrightarrow d[v] < \infty$

**Beweis:**  $s \rightsquigarrow v \Rightarrow d[v] < \infty$  : folgt aus Lemma 24.2

$s \rightsquigarrow v \Rightarrow d[v] = \infty$  : entspricht der Kein-Pfad-Eigenschaft



## Korrektheit des Bellman-Ford-Algorithmus

- **Theorem 24.4:** (Korrektheit des Rückgabewerts)

1.  $G$  enthält keine von  $s$  aus erreichbaren negativen Zyklen:

1. Für alle Knoten  $v$  gilt  $d[v] = \delta(s, v)$
2.  $G_\pi$  beschreibt einen Baum kürzester Pfade
3. Der Rückgabewert ist TRUE

2.  $G$  enthält von  $s$  aus erreichbare negative Zyklen:

1. Der Rückgabewert ist FALSE

**Beweis:**

1.1: folgt aus Lemma 24.2 / Korollar 24.3

1.2: folgt aus der Vorgängerteilgraph-Eigenschaft

1.3: folgt aus 1.1 und der Dreiecksungleichung:

Schleife 5-7 testet die Dreiecksungleichung für alle Kanten:

Für alle Kanten  $e = (u, v)$  gilt:

$$d[v] \leq \delta(s, v)$$

$$\leq \delta(s, u) + w(u, v) = d[u] + w(u, v)$$



## Korrektheit des Bellman-Ford-Algorithmus

- **Beweis (cont.):**

2.1: (Beweis durch Widerspruch) Sei  $c = (v_0, v_1, v_2, \dots, v_k = v_0)$  ein von  $s$  erreichbarer Zyklus mit  $w(c) = \sum_{i=1}^k w(v_{i-1}, v_i) < 0$

Annahme: Der Rückgabewert ist TRUE

$\Rightarrow d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$  für  $i=1, \dots, k$

$$\begin{aligned} \Rightarrow \sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i) \\ &= \sum_{i=1}^k d[v_i] + d[v_0] - d[v_k] + w(c) \\ &= \sum_{i=1}^k d[v_i] + w(c) \quad \text{da } v_0 = v_k \\ &\Leftrightarrow 0 \leq w(c) \quad \text{⚡} \end{aligned}$$



## Kürzeste Pfade von einem Startknoten in DAGs

- **Idee:**

- Knoten können nur in der Reihenfolge der topologischen Sortierung in einem kürzesten Weg vorkommen
- betrachte Knoten in topologisch sortierter Reihenfolge  $\Rightarrow$   
für alle Wege  $p = (s = v_0, v_1, \dots, v_k)$  werden die Kanten in Reihenfolge des Weges relaxiert.

- **DAG-SHORTEST-PATHS( $G, w, s$ )**

- 1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 2  $VL \leftarrow \text{TOPOLOGICAL-SORT}(G)$
- 3 **while**  $VL \neq \emptyset$  **do**
- 4  $u \leftarrow \text{head}[VL]$ ; LIST-DELETE(1,  $VL$ )
- 5 **for each**  $v \in \text{Adj}[u]$  **do**
- 6 RELAX( $u, v, w$ )

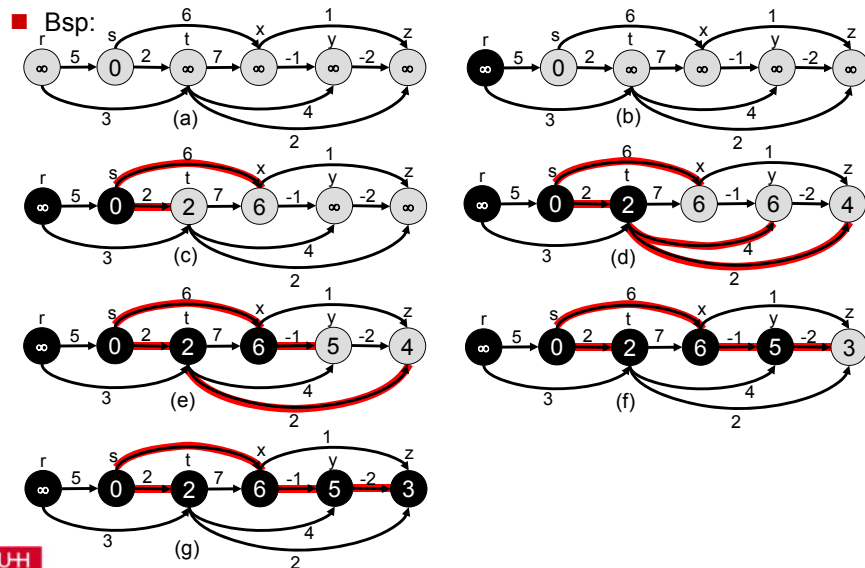
- **Laufzeit:**

- Topologische Suche:  $O(N + M)$
- Initialisierung:  $O(N)$
- Schleife 3-6:  $O(M)$

Insgesamt  $T_{\text{DAG-SHORTEST-PATHS}}(N, M) = O(N + M)$



## Kürzeste Pfade von einem Startknoten in DAGs



## Korrektheit von DAG-SHORTEST-PATHS

### ■ Theorem 24.5 (Korrektheit von DAG-SHORTEST-PATHS):

Sei  $G=(V,E)$  ein DAG mit Startknoten  $s$ . Nach Terminierung von DAG-SHORTEST-PATHS gilt  $d[v] = \delta(s,v)$  und  $G_\pi$  beschreibt einen Baum kürzester Pfade.

#### Beweis:

Fall 1:  $s \not\rightarrow v \Rightarrow d[v] = \delta(s,v) = \infty$  : folgt aus der Kein-Pfad-Eigenschaft.

Fall 2:  $s \rightarrow v \Rightarrow d[v] = \delta(s,v)$ . Sei  $p=(s=v_0, v_1, v_2, \dots, v_k=v)$  ein kürzester Pfad von  $s$  nach  $v$ . Bzgl. der topologischen Sortierung  $t$  gilt  $t(v_{i-1}) < t(v_i)$  für  $i = \{1, \dots, k\}$ .

$\Rightarrow$  DAG-SHORTEST-PATHS bearbeitet die Kanten von  $p$  in der Reihenfolge von  $p$

$\Rightarrow d[v] = \delta(s,v)$  (Pfad-Relaxations-Eigenschaft)

### ■ DAG-SHORTEST-PATHS

- arbeitet auch mit negativen Kantengewichten korrekt.
- eignet sich auch zur Berechnung längster Pfade.



## Dijkstras Algorithmus (nicht-negative Kantengewichte)

### ■ Einschränkung: alle Kantengewichte sind nicht-negativ

#### ■ Idee:

- speichere in  $S$  alle Knoten, deren kürzeste Pfade bereits berechnet worden sind (Anfang  $S = \{s\}$ )
- gehe von  $S$  aus zum Knoten  $u \notin S$ , der am dichtesten an einem Knoten aus  $S$  liegt und berechne  $d[u]$
- speichere alle Knoten  $v \notin S$  in einer Priority-Queue, Priorität: obere Schranke  $d[v]$

### ■ DIJKSTRA( $G, w, s$ )

INITIALIZE-SINGLE-SOURCE( $G, s$ )

$S \leftarrow \emptyset$

$Q \leftarrow V[G]$

**while**  $Q \neq \emptyset$  **do**

$u \leftarrow \text{EXTRACT-MIN}(Q)$

$S \leftarrow S \cup \{u\}$

**for each**  $v \in \text{Adj}[u]$  **do**

        RELAX-DECREASE( $u, v, w, Q$ )

RELAX-DECREASE( $u, v, w, Q$ )

**if**  $d[v] > d[u] + w(u,v)$  **then**

$d[v] \leftarrow d[u] + w(u,v);$

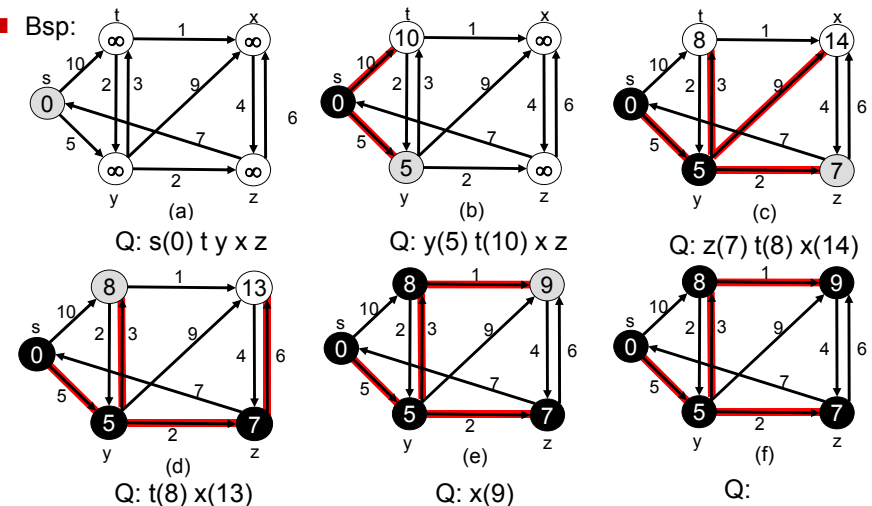
$\pi[v] \leftarrow u$

        DECREASE-KEY( $Q, v, d[v]$ )



## Dijkstras Algorithmus

### ■ Bsp:



Beschreibung Priority Queue  $Q$ :  $v(d[v])$   $u(d[u])$  ...

### Theorem 24.6 (Korrektheit des Dijkstra-Algorithmus):

Sei  $G=(V,E)$  ein gerichteter Graph mit Gewichtsfunktion  $w: E \rightarrow \mathbb{R}_0^+$ ,  $s \in V$  ein Startknoten. Nach Terminierung des Dijkstra-Algorithmus gilt für alle  $u \in V$ :  $d[u] = \delta(s,u)$ .

**Beweis:** Sei  $\delta^S(s,u)$  die Länge eines kürzesten Weges, der nur Knoten aus  $S$  enthält. Sei  $N = \{v \in V \mid d[v] < \infty\} \setminus S$

Invariante der while-Schleife:

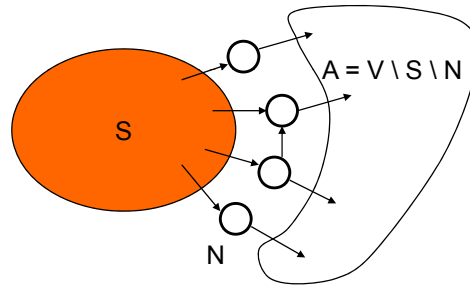
1.  $\forall v \in S: d[v] = \delta(s,v)$

2.  $\forall v \in N: d[v] = \delta^S(s,v)$

■ Es ist ausreichend, den Zeitpunkt der Einfügung von  $v$  in  $S$  zu betrachten (Konvergenzeigenschaft)

■ Initialisierung:  $S = \emptyset$ ,  $N = \{s\}$   
die Invariante ist somit erfüllt

■ Terminierung:  $S = V$ , mit Teil 1 der Invariante ist das Theorem bewiesen.



### Beweis von Theorem 24.6 (cont.)

■ Fortsetzung: Sei  $u \in V - S$  mit  $d[u]$  minimal wie im Algorithmus gewählt

◆ Teil 1: Zeige  $d[u] = \delta(s,u)$

Fall 1:  $d[u] = \infty$ , d.h.  $u \notin N$

=> es gibt keine Kante  $e=(v,u)$  mit  $v \in S$  und  $\delta(s,v) < \infty$ , da  $e$  relaxiert wäre. Dies gilt auch für alle  $v \in V - S$ , da  $d[u]$  minimal ist.

=> es gibt keinen Pfad  $s \rightarrow u \Rightarrow \delta(s,u) = \infty$

Fall 2:  $d[u] < \infty$ , d.h.  $u \in N$ . Sei  $p = (s, \dots, x, y, \dots, u)$  ein kürzester Pfad von  $s$  zu  $u$ .  $(x,y)$  sei so gewählt, dass  $y$  der erste Knoten mit  $y \notin S$  gilt.

=>  $d[u] \leq d[y]$  (wg. Wahl von  $u$ )

$= \delta^S(s,y)$  (wg. Schleifeninvariante)

$= \delta(s,y)$  (wg. Lemma 24.1, optimale Teilpfade)

$\leq \delta(s,u)$  (wg. Lemma 24.1, optimale Teilpfade)

◆ Teil 2: Gültigkeit der Invariante nach Durchlaufen der Schleife

1.  $\delta(s,u) \leq d[u] \leq \delta(s,u)$  (wg. Eigenschaft der oberen Schranke und Teil 1)

2. folgt direkt aus Schleifeninvariante und Relaxation aller zu  $u$  inzidenten Kanten  $(u,v)$ .

### Laufzeit:

■ while-Schleife:  $N = |V|$  Iterationen

◆  $N$  EXTRACT-MIN Operationen auf einer  $N$ -elementigen Warteschlange

◆ for-Schleife: insgesamt  $M = |E|$  Iterationen (jede Kante ein mal)

◆  $M$  RELAX- und damit DECREASE-KEY Operationen auf einer  $N$ -elementigen Warteschlange

■ Implementierung der Warteschlange:

◆ durch binären Heap mit zusätzlichem  $N$ -elementigen Array zum Auffinden der Knoten im Heap (für DECREASE-KEY)

◆ EXTRACT-MIN:  $O(\log N)$  DECREASE-KEY:  $O(\log N)$

■  $T_{\text{Dijkstra}}(N,M) = O(N \log N) + O(M \log N) = O(M \log N)$

■ Fibonacci-Heaps:

◆ EXTRACT-MIN:  $O(\log N)$  DECREASE-KEY:  $O(1)$  (amortisiert)

◆  $T_{\text{Dijkstra}}(N,M) = O(N \log N + M)$

### Das All-Pairs-Shortest-Paths Problem

■ berechne kürzeste Wege zwischen allen Knotenpaaren

Ausgabe:  $N \times N$  Matrix  $D = (d_{ij})$  mit allen paarweisen Distanzen

$N \times N$  Matrix  $\Pi = (\pi_{ij})$  mit allen Vorgängern

■ Annahme: nicht-negative Kantengewichte

■ Algorithmus:  $N \cdot$  Dijkstras Algorithmus

■ Laufzeit:  $O(N(N+M) \log N)$

■ sonst:

■ Algorithmus:  $N \cdot$  Bellman-Ford Algorithmus

■ Laufzeit  $O(N^2 M)$

■ Repräsentation von Graphen für all-pairs:

■ gewichtete Adjazenzmatrix:

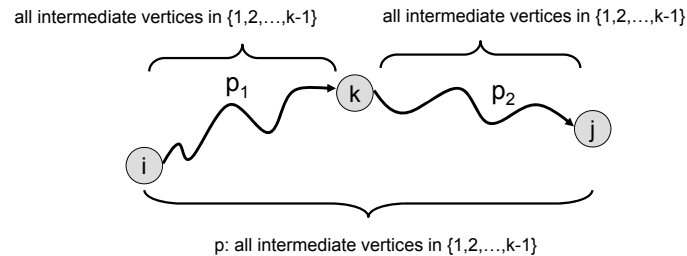
$$w_{ij} = \begin{cases} 0 & : i = j \\ w(i, j) & : i \neq j \wedge (i, j) \in E \\ \infty & : i \neq j \wedge (i, j) \notin E \end{cases}$$

## Das All-Pairs-Shortest-Paths Problem

### ■ Floyd-Warshall-Algorithmus:

- $d_{ij}^{(k)}$  : Länge des kürzesten Weges von  $i$  nach  $j$ , auf dessen Pfad neben  $i$  und  $j$  nur Knoten aus  $\{1, \dots, k\}$  besucht werden

- Rekursive Definition: 
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & : k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & : k > 0 \end{cases}$$



## Floyd-Warshall Algorithmus

### ■ FLOYD-WARSHALL(W)

```

1  $n \leftarrow \text{rows}[W]$ 
2  $D^{(0)} \leftarrow W$ 
3 for  $k \leftarrow 1$  to  $n$ 
4   do for  $i \leftarrow 1$  to  $n$ 
5     do for  $j \leftarrow 1$  to  $n$ 
6       do  $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
7 return  $D^{(n)}$ 
    
```

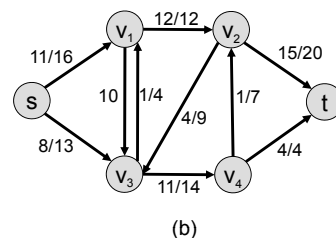
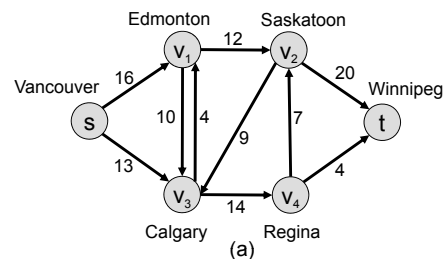
■ Laufzeit:  $T(N) = \Theta(N^3)$       Platz:  $S(N) = \Theta(N^2)$



## Maximaler Fluss

### ■ Maximaler Fluss (Maximum Flow):

- Gerichteter Graph  $G=(V,E)$  mit Kantenkapazitäten  $c:E \rightarrow \mathbb{R}^+$
- zwei ausgezeichnete Knoten: Quelle  $s$  und die Senke  $t$
- Was ist der maximale Fluss  $f(s,t)$  unter den Randbedingungen:
  - ◆ Capacity constraint:  $f(u,v) \leq c(u,v)$
  - ◆ Skew symmetry:  $f(u,v) = -f(v,u)$
  - ◆ Flow conservation:  $\sum_v f(u,v) = 0 \quad \forall u \in V \setminus \{s,t\}$



## Maximaler Fluss

### ■ Augmentierender Pfad:

- Pfad  $(s=v_0, v_1, v_2, \dots, v_k=t)$  mit  $f(v_i, v_{i-1}) < c(v_i, v_{i-1})$  für alle Kanten  $i = 1, \dots, k$

### ■ Ford-Fulkerson-Methode:

### ■ FORD-FULKERSON-METHOD( $G, s, t$ )

```

1 initialize flow  $f$  to 0
2 while there exists an augmenting path  $p$ 
3   do augment flow  $f$  along  $p$ 
4 return  $f$ 
    
```

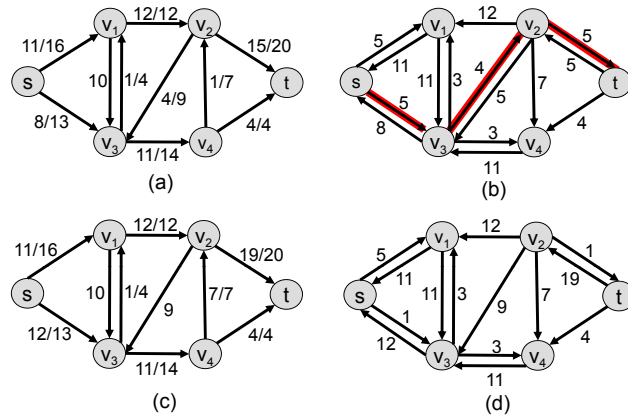
### ■ Residuale Netzwerke (residual networks):

- setze Kapazität auf residuale Kapazität:  $c_r(u,v) = c(u,v) - f(u,v)$
- füge Kanten in Gegenrichtung mit Kapazität  $c_r(v,u) = f(u,v)$  ein



## Maximaler Fluss

### ■ Bsp:

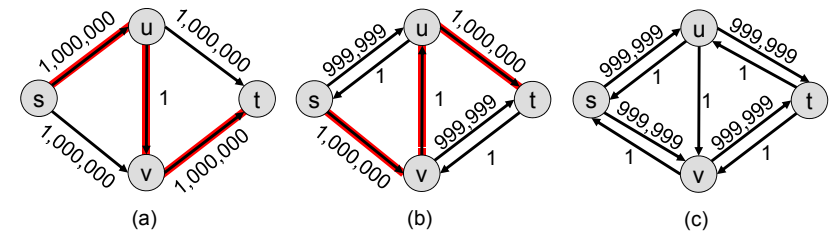


- a) Flussnetzwerk: Kante: Fluss / Kapazität
- b) Residuales Netzwerk, Augmentierender Pfad
- c) Flussnetzwerk nach Augmentierung
- d) Residuales Netzwerk



## Maximaler Fluss

### ■ Laufzeit Ford-Fulkerson: $O(E |f^*|)$



### ■ Edmonds-Karp Algorithmus:

- Wähle als augmentierenden Pfad den kürzesten Weg (uniformes Kantengewicht von 1) von s nach t
- Man kann zeigen, dass die Distanz von s zu t im Residualen Netzwerk monoton steigt  $\Rightarrow$  # Augmentierungen =  $O(|V||E|)$
- Finden eines kürzesten Weges bei uniformen Kantengewichten: BFS-Algorithmus,  $O(|E|)$  Zeit
- Gesamtzeit:  $O(|V| |E|^2)$



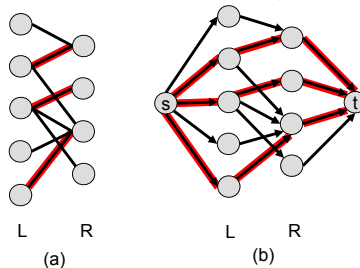
## Maximales bipartites Matching

### ■ Matching

- Teilmenge  $M \subseteq E$ , so dass  $\forall v \in V: |\{e \in M \mid v \in e\}| \leq 1$

### ■ Maximales bipartites Matching:

- Bipartiter Graph  $G = (V \cup U, E)$
- Finde maximales Matching  $M$ , d.h.  $|M|$  maximal



- Beschreibung als Flussproblem, Laufzeit:  $O(|V||E|)$

### ■ maximales, gewichtetes bipartites Matching:

- Laufzeit:  $O(|V||E|\log |V|)$



## Schwere Graphprobleme

- Für viele Graphprobleme sind keine Algorithmen mit Laufzeit  $O(|V|^k)$  für konstantes k bekannt (polynomielle Laufzeit).
- Theoretische Betrachtungen zeigen, dass wenn eines dieser Probleme in polynomieller Zeit lösbar ist, dann gilt dies auch für sehr viele andere (siehe Kap. 7).

### ■ Beispiele:

- Hamiltonian Path: gibt es einen einfachen Kreis der Länge  $|V|$ ?
- Subgraph: Ist Graph G ein Teilgraph von  $G'$ ?
- Graphisomorphie: Sind Graphen G und  $G'$  isomorph?
- Clique: Gibt es einen vollständig verbundenen Teilgraphen mit k Knoten?
- Dreifärbbarkeit: Gibt es eine Funktion  $f: V \rightarrow \{0, 1, 2\}$  mit  $f(v) \neq f(u) \forall \{u, v\} \in E$ ?



## **Kapitel 6: Dynamische Programmierung**

- 6.1 Prinzip der dynamischen Programmierung 165
- 6.2 Beispiel 1: Ablaufkoordination von Montagebändern 166
- 6.3 Beispiel 2: Matrix-Kettenmultiplikation 172



## Kapitel 6: Dynamische Programmierung

Prinzip der Dynamischen Programmierung

Beispiel 1: Montagebänder

Beispiel 2: Matrix-Kettenmultiplikation

## 6.1 Prinzip der dynamischen Programmierung

- Häufig verwendete Lösungsstrategie für komplexe Probleme:
  - Zerlege die Eingabe des Problems in ein/mehrere Teilproblem(e)
  - Wende dieses Prinzip rekursiv an
  - Unterschreitet die Eingabegröße einen Grenzwert, kann die Lösung einfach berechnet werden
  - Konstruiere die Lösung des Problems aus der Lösung des Teilproblems / den Lösungen der Teilprobleme
- → Rekursive Algorithmen
- Beispiel: Divide&Conquer-Prinzip (z.B. Mergesort, Quicksort)
  - Teilung in zwei voneinander unabhängige zu lösende Teilprobleme
  - Rekursive Lösung der Teilprobleme
  - Zusammenfügen der Teillösungen zur Gesamtlösung
- Komplexe rekursive Schema können dazu führen, dass Teilprobleme mehrfach im Rekursionsbaum auftreten.
  - Rekursive Implementierung führt zu hoher Laufzeit
  - Berechne die Lösung von Teilproblemen nur einmal und speichere sie in einer Tabelle:

### Dynamische Programmierung



© Matthias Rarey, ZBH, Universität Hamburg

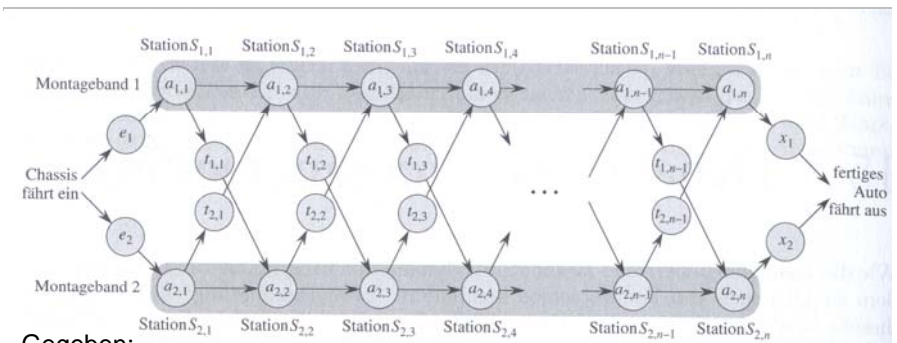
gescannte Abbildungen: © Cormen et al,  
Oldenbourg Verlag, 2004

2

## Prinzip der Dynamischen Programmierung

- ‚Programmierung‘ in Dynamischer Programmierung hat historische Gründe und steht für das ‚systematische Füllen von Tabellen‘, nicht für das Schreiben von Computerprogrammen.
- Entwicklungsschritte in der Dynamischen Programmierung:
  1. **Charakterisiere** die Struktur einer optimalen Lösung
  2. **Definiere** den Wert einer optimalen Lösung rekursiv (Die Umsetzung in einen rekursiven Algorithmus würde zu einem top-down-Ansatz führen)
  3. **Berechne** den Wert einer optimalen Lösung mit einem bottom-up-Ansatz (Speichere dabei die bereits berechneten Teillösungen)
  4. **Konstruiere** eine zugehörige optimale Lösung
- Dynamische Programmierung
  - wird häufig zur Lösung von Optimierungsproblemen eingesetzt.
  - ist ein sehr mächtiges Paradigma im Algorithmenentwurf.
  - sollte bzgl. seiner Anwendbarkeit für ein neues Optimierungsproblem (mittels Durchführung von Schritt 1) getestet werden.

## 6.2 Beispiel 1: Ablaufkoordination von Montagebändern



Gegeben:

- Zwei Montagebänder mit n Stationen  $S_{1,1}, \dots, S_{1,n}$  und  $S_{2,1}, \dots, S_{2,n}$
  - Montagezeiten für alle Stationen:  $a_{1,1}, \dots, a_{1,n}$  und  $a_{2,1}, \dots, a_{2,n}$
  - Ein- und Ausfahrzeiten  $e_1, x_1$  und  $e_2, x_2$
  - Transferzeiten bei Montagebandwechsel  $t_{1,1}, \dots, t_{1,n-1}$  und  $t_{2,1}, \dots, t_{2,n-1}$
- Gesucht: Schnellst mögliche Montage (unter Verwendung beider Bänder)



© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al,  
Oldenbourg Verlag, 2004

3



© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al,  
Oldenbourg Verlag, 2004

4



## Schritt 1: Charakterisierung der Struktur der schnellsten Montagefahrt

### ■ Größe des Lösungsraums:

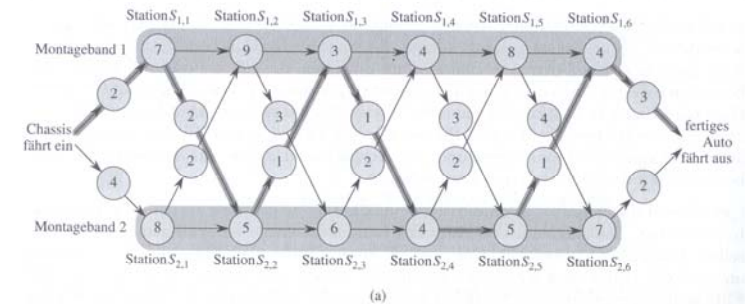
- Für jeden der  $n$  Montageschritte können wir uns für eine der beiden Stationen entscheiden  
=> es gibt  $\Omega(2^N)$  verschiedene Montagefahrten

### ■ Struktur einer optimalen Lösung:

- Lässt sich eine optimale Lösung aus einer optimalen Lösung eines Teilproblems konstruieren/ableiten?
- Betrachte eine schnellste Montagefahrt bis zur Station  $S_{1,j}$ :
  - ◆  $j=1$ : Chassis fährt über zu Band 1, Zeit:  $e_1$ , keine Alternative
  - ◆  $j>1$ : Möglichkeit 1: Chassis fährt von  $S_{1,j-1}$  direkt zu  $S_{1,j}$   
Möglichkeit 2: Chassis wechselt das Band und kommt von  $S_{2,j-1}$  zu  $S_{1,j}$  und nimmt die Transferzeit  $t_{2,j-1}$  in Kauf
- Schnellste Montagefahrt setzt sich aus optimalen Teilfahrten zusammen:
  - ◆ Führt eine schnellste Montagefahrt von  $S_{1,j-1}$  zu  $S_{1,j}$  (Möglichkeit 1), so ist die Teilfahrt zu  $S_{1,j-1}$  ebenfalls eine schnellste Montagefahrt (sonst könnte die schnellste Montagefahrt verkürzt werden).
  - ◆ Analog folgt, dass auch die Teilfahrt zu  $S_{2,j-1}$  optimal sein muss.



## Schritt 1: Charakterisierung der Struktur der schnellsten Montagefahrt



### ■ Eigenschaft der optimalen Teilstruktur:

- Eine optimale Lösung des Problems beinhaltet optimale Lösungen von Teilproblemen.

konkret:

- Eine schnellste Montagefahrt bis zur Station  $S_{i,j}$  besteht aus einer der schnellsten Montagefahrten bis zu den Stationen  $S_{i,j-1}$ .

=> Voraussetzung für die Anwendbarkeit der Dynamischen Programmierung.



## Schritt 2: Rekursive Lösung des Problems

- Eigenschaft der optimalen Teilstruktur erlaubt eine rekursive Lösung des Problems:

- $f^*$ : Zeit einer optimalen Montagefahrt
- $f_i[j]$ : optimale Zeit für eine Montagefahrt zur Station  $S_{i,j}$  (inkl. Montagezeit  $a_{i,j}$ )  
 $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$   
 $f_1[1] = e_1 + a_{1,1}$   
 $f_2[1] = e_2 + a_{2,1}$

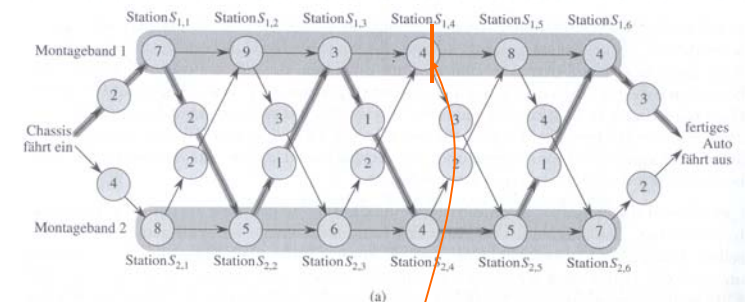
- für  $j>1$  gibt es die Alternativen, über Station  $S_{1,j-1}$  oder  $S_{2,j-1}$  zu laufen:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{falls } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{falls } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{falls } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{falls } j \geq 2 \end{cases}$$



## Schritt 2: Rekursive Lösung des Problems



$j$	1	2	3	4	5	6	
$f_1[j]$	9	18	20	24	32	35	$f^* = 38$
$f_2[j]$	12	16	22	25	30	37	

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{falls } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{falls } j \geq 2 \end{cases}$$



## Schritt 2: Rekursive Lösung des Problems

- **CALL-FASTEST-WAY(a, t, e, x, n)**  
**return** min( RECURSIVE-FASTEST-WAY(a, t, e, 1, n) + x<sub>1</sub> ,  
RECURSIVE-FASTEST-WAY(a, t, e, 2, n) + x<sub>2</sub>)
- **RECURSIVE-FASTEST-WAY(a, t, e, i, j)**  
// berechnet Zeit f<sub>i,j</sub> einer optimalen Montagefahrt zur Station S<sub>i,j</sub>  
**if** j = 1 **then return** (e<sub>i</sub> + a<sub>i,1</sub>)  
**else**  
// Möglichkeit 1: kein Bandwechsel  
f<sub>i</sub> ← RECURSIVE-FASTEST-WAY(a, t, e, i, j-1) + a<sub>i,j</sub>  
// Möglichkeit 2: Bandwechsel von Band (3-i) auf Band i  
f<sub>3-i</sub> ← RECURSIVE-FASTEST-WAY(a, t, e, 3-i, j-1) + t<sub>3-i,j-1</sub> + a<sub>i,j</sub>  
**return** min( f<sub>i</sub>, f<sub>3-i</sub> )  

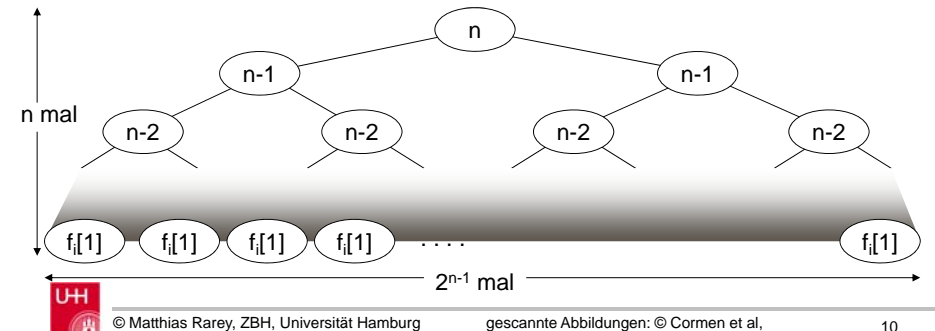
$$T_{RFW}(n) = \begin{cases} c & : n = 1 \\ 2T_{RFW}(n-1) + c & : n > 1 \end{cases}$$



## Schritt 2: Rekursive Lösung des Problems

$$T_{RFW}(n) = \begin{cases} c & : n = 1 \\ 2T_{RFW}(n-1) + c & : n > 1 \end{cases}$$

$$T_{RFW}(n) = \underbrace{2(2(\dots 2(2T_{RFW}(1) + c) + \dots c) + c)}_{n-1 \text{ mal}} = 2^{n-1}c + \sum_{i=0}^{n-1} 2^i c = (2^{n-1} + 2^n - 1)c = \left(\frac{3}{2}2^n - 1\right)c = \Theta(2^n)$$



## Schritt 3: Berechne den Wert der optimalen Lösung

- **Lösung:**
  - Berechnung der Funktionswerte f<sub>i,j</sub> bottom-up (Reihenfolge j=1,2, ..., n)
  - Speicherung der Resultate in den Arrays f<sub>1</sub>[] und f<sub>2</sub>[]
- **TIME-OF-FASTEST-WAY(a,t,e,x,n)**

```

1 f1[1] ← e1 + a1,1
2 f2[1] ← e2 + a2,1
3 for j ← 2 to n do
4   f1[j] ← min( f1[j-1] + a1,j, f2[j-1] + t2,j-1 + a1,j )
5   f2[j] ← min( f2[j-1] + a2,j, f1[j-1] + t1,j-1 + a2,j )
6 return min( f1[n] + x1, f2[n] + x2 )

```
- **Laufzeit:** T<sub>TIME-OF-FASTEST-WAY</sub>(n) = Θ(n)
- **Speicherbedarf:** S<sub>TIME-OF-FASTEST-WAY</sub>(n) = Θ(n)



## Schritt 3: Berechne den Wert der optimalen Lösung

- **FASTEST-WAY(a,t,e,x,n)**

```

1 f1[1] ← e1 + a1,1; f2[1] ← e2 + a2,1
3 for j ← 2 to n do
4   if f1[j-1] + a1,j ≤ f2[j-1] + t2,j-1 + a1,j then
5     f1[j] ← f1[j-1] + a1,j
6     l1[j] ← 1
7   else f1[j] ← f2[j-1] + t2,j-1 + a1,j
8     l1[j] ← 2
9   if f2[j-1] + a2,j ≤ f1[j-1] + t1,j-1 + a2,j then
10    f2[j] ← f2[j-1] + a2,j
11    l2[j] ← 2
12   else f2[j] ← f1[j-1] + t1,j-1 + a2,j
13    l2[j] ← 1
14 if f1[n] + x1 ≤ f2[n] + x2 then
15   f* ← f1[n] + x1
16   l* ← 1
17 else f* ← f2[n] + x2
18   l* ← 2
19 return ( f*, l* )

```

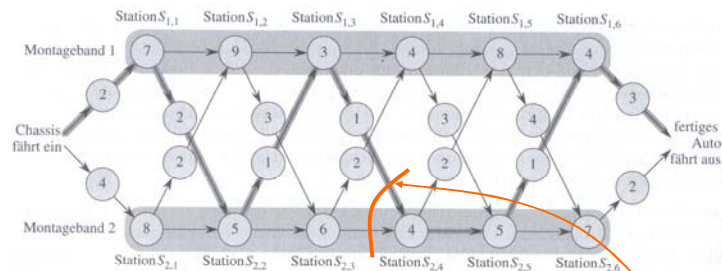
Berechnung  
von f<sub>1</sub>[] und l<sub>1</sub>[]

Berechnung  
von f<sub>2</sub>[] und l<sub>2</sub>[]

Berechnung  
von f\* und l\*



#### Schritt 4: Konstruktion einer schnellsten Montagefahrt



- Funktion TIME-OF-FASTEST-WAY() liefert bereits die Montagezeit, allerdings nicht die zugehörige Fahrt (d.h. Auswahl der Stationen).  
→ Speicherung der Montageband-Nummer (1 oder 2), die zur kürzesten Fahrt geführt hat:  $l_1[j]$  und  $l_2[j]$ :  
 $l_1[j]$ : Nummer des Bandes, dessen Station (j-1) auf der Fahrt mit Zeit  $f_1[j]$  zu Station  $S_{1,j}$  verwendet wurde,  $j=2, \dots, n$   
 $l_2[j]$ : Nummer des Bandes, dessen Station n verwendet wurde

	j	2	3	4	5	6
$l_1[j]$		1	2	1	1	2
$l_2[j]$		1	2	1	2	2

13

#### Schritt 3: Berechne den Wert der optimalen Lösung

- FASTEST-WAY(a,t,e,x,n)
  - $f_1[1] \leftarrow e_1 + a_{1,1}; f_2[1] \leftarrow e_2 + a_{2,1}$
  - for** j  $\leftarrow 2$  **to** n **do**
  - if**  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + a_{2,j}$  **then**
  - $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
  - $l_1[j] \leftarrow 1$
  - else**  $f_1[j] \leftarrow f_2[j-1] + a_{2,j} + a_{1,j}$
  - $l_1[j] \leftarrow 2$
  - if**  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + a_{1,j} + a_{2,j}$  **then**
  - $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
  - $l_2[j] \leftarrow 2$
  - else**  $f_2[j] \leftarrow f_1[j-1] + a_{1,j} + a_{2,j}$
  - $l_2[j] \leftarrow 1$
  - if**  $f_1[n] + x_1 \leq f_2[n] + x_2$  **then**
  - $f^* \leftarrow f_1[n] + x_1$
  - $l^* \leftarrow 1$
  - else**  $f^* \leftarrow f_2[n] + x_2$
  - $l^* \leftarrow 2$
  - return**(  $f^*, l^*$  )

Berechnung von  $f_1[j]$  und  $l_1[j]$

Berechnung von  $f_2[j]$  und  $l_2[j]$

Berechnung von  $f^*$  und  $l^*$

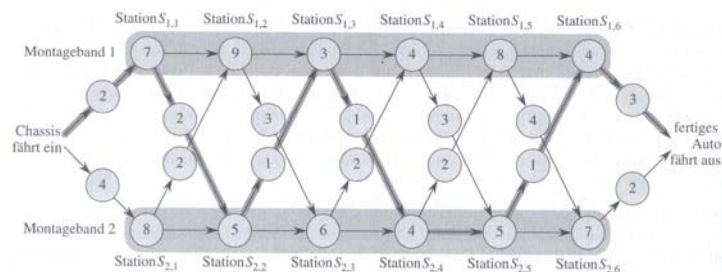
© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al, Oldenbourg Verlag, 2004

14

#### Schritt 4: Konstruktion einer schnellsten Montagefahrt

- Beginnend mit  $l^*$  lässt sich die schnellste Montagefahrt rekonstruieren:



- PRINT-STATIONS( $l, n$ )
  - $i \leftarrow l^*$
  - write "Band" i, "Station" n
  - for** j  $\leftarrow n$  **downto** 2
  - do**  $i \leftarrow l_i[j]$
  - write "Band" i, "Station" j-1

	j	2	3	4	5	6
$l_1[j]$		1	2	1	1	2
$l_2[j]$		1	2	1	2	2

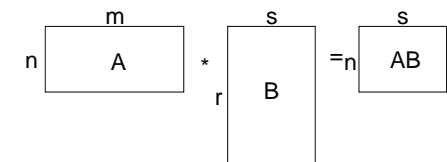
15

#### 6.3 Beispiel 2: Matrix-Kettenmultiplikation

$$A = [a_{ij}]_{1 \leq i \leq n, 1 \leq j \leq m}$$

$$B = [b_{ij}]_{1 \leq i \leq r, 1 \leq j \leq s}$$

$$AB = \begin{cases} \sum_{k=1}^m a_{ik} b_{kj} & : \text{falls } m = r \\ \text{n.d.} & : \text{sonst} \end{cases}$$



- Assoziativität der Matrixmultiplikation:  $A(BC) = (AB)C$
- MATRIX-MULTIPLY(A,B)
  - if** spalten[A]  $\neq$  zeilen[B] **then**
  - error** "inkompatible Dimensionen"
  - else for** i  $\leftarrow 1$  **to** zeilen[A] **do**
  - for** j  $\leftarrow 1$  **to** spalten[B] **do**
  - $C[i,j] \leftarrow 0$
  - for** k  $\leftarrow 1$  **to** spalten[A] **do**
  - $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$
  - return** C

Laufzeit hängt von den Dimensionen der Matrizen ab:  
 $T_{\text{MATRIX-MULTIPLY}}(A,B) = O(z[A] s[A] s[B])$   
 $= O(z[A] z[B] s[B])$

© Matthias Rarey, ZBH, Universität Hamburg

gescannte Abbildungen: © Cormen et al, Oldenbourg Verlag, 2004

16

## Beispiel 2: Matrix-Kettenmultiplikation

### ■ Problem der **Matrix-Kettenmultiplikation**:

geg: Sequenz von Matrizen  $\langle A_1, A_2, \dots, A_n \rangle$

ges: Reihenfolge der Matrixmultiplikation (vollständige Klammerung), die die Anzahl skalarer Multiplikationen minimiert. (Die Berechnung des Produkts  $A_1 A_2 \dots A_n$  wird nicht als Teil des Problems betrachtet.)

### ■ Beispiel:

- $A_1$ : 10 x 100 Matrix     $A_2$ : 100 x 5 Matrix     $A_3$ : 5 x 50 Matrix
  - $A_1 A_2$  ist eine 10 x 5 Matrix  
 $A_2 A_3$  ist eine 100 x 50 Matrix
  - $((A_1 A_2) A_3)$  : # Multiplikationen  $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7.500$   
 $(A_1 (A_2 A_3))$  : # Multiplikationen  $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75.000$
- $((A_1 A_2) A_3)$  kann 10\* schneller berechnet werden als  $(A_1 (A_2 A_3))$



## Schritt 1: Struktur der optimalen Klammerung

### ■ Größe des Lösungsraums $P(n)$

Wie viele verschiedene Klammerungen  $P(n)$  gibt es bei der Multiplikation von  $n$  Matrizen?

$$n=1 : P(n) = 1$$

$$n=2 : P(n) = 1$$

$$n=3 : P(n) = 2$$

$$n=4 : P(n) = 5$$

$(A (B (C D)))$   
 $(A ((B C) D))$   
 $((A B) (C D))$   
 $((A (B C)) D)$   
 $(( (A B) C) D)$

$n > 1$  : Es gibt  $n-1$  Möglichkeiten für die letzte auszuführende Multiplikation  
 Liegt diese zwischen  $k$  und  $k+1$ , so gibt es  $P(k)$  Möglichkeiten für die Klammerung des ersten,  $P(n-k)$  Möglichkeiten für den zweiten Faktor.

$$P(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{falls } n \geq 2 \end{cases} \quad P(n) = \Omega\left(\frac{4^n}{n^{2/3}}\right)$$

*Catalan-Zahlen*, wachsen exponentiell mit  $n$



## Schritt 1: Struktur der optimalen Klammerung

### ■ Wie kann eine optimale Lösung aus optimalen Teillösungen konstruiert werden?

- Sei  $A_{i..j} = A_i A_{i+1} \dots A_j$  das Produkt der Matrizen  $A_i$  bis  $A_j$
  - Die Matrix  $A_i$  sei eine  $p_{i-1} \times p_i$  - Matrix
  - Für  $i < j$  gibt es eine Position  $k$  der zuletzt ausgeführten Matrix-Multiplikation:  
 $A_{i..j} = (A_i \dots A_k) (A_{k+1} \dots A_j)$
  - Für die zuletzt ausgeführte Matrix-Multiplikation (an Position  $k$ ) werden  $p_{i-1} p_k p_j$  skalare Multiplikationen benötigt. Diese Zahl ist unabhängig davon, wie  $A_{i..k}$  und  $A_{k+1..j}$  berechnet werden.
  - Die optimale Anzahl skalarer Multiplikationen ist die Summe über die jeweils optimale Anzahl zur Berechnung von  $A_{i..k}$  und  $A_{k+1..j}$  und  $p_{i-1} p_k p_j$
- Die Lösung des Matrix-Kettenmultiplikationsproblems erfüllt die Eigenschaft der optimalen Teilstruktur.



## Schritt 2: Rekursive Lösung des Matrix-Kettenmultiplikationsproblems

### ■ Rekursive Beschreibung:

- Sei  $m[i,j]$  die minimale Anzahl skalarer Multiplikationen zur Berechnung von  $A_{i..j}$ . Liegt die letzte auszuführende Multiplikation zwischen  $k$  und  $k+1$ , gilt:  $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

- $m[i,j]$  lässt sich durch Minimierung über alle möglichen Werte  $k$  bestimmen:

$$m[i, j] = \begin{cases} 0 & \text{falls } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{falls } i < j \end{cases}$$

### ■ RECURSIVE-MATRIX-CHAIN( $p, i, j$ ) // Erster Aufruf mit $i=1, j=n$

if  $i = j$  then return 0

else

$m \leftarrow \infty$

for  $k \leftarrow i$  to  $j-1$  do

$m \leftarrow \min(m, \text{RECURSIVE-MATRIX-CHAIN}(p, i, k) + \text{RECURSIVE-MATRIX-CHAIN}(p, k+1, j) + p[i-1]p[k]p[j])$

return  $m$



### Schritt 3: Berechnung der minimalen Anzahl Multiplikationen

- Die Laufzeit von RECURSIVE-MATRIX-CHAIN ist exponentiell:

$$T_{RMCO}(n) = \begin{cases} c & : n=1 \\ \sum_{k=1}^{n-1} (T(k) + T(n-k) + c) + c & : n>1 \end{cases} \quad T_{RMCO}(n) = \Omega(2^n)$$

(Beweis: Zeige  $T_{RMCO}(n) \geq 2^{n-1}$ )

- Überlappende Teilprobleme:

- Es gilt  $1 \leq i \leq j \leq n$ , somit gibt es  $n(n+1)/2$  verschiedene Teilprobleme  
 $\Rightarrow$  in der Berechnung von RECURSIVE-MATRIX-CHAIN gibt es überlappende Teilprobleme
- Speichere das Resultat für die Eingabe  $(p, i, j)$  in einer  $n \times n$ -Matrix  $m$  an Position  $m[i, j]$

- Bottom-up Berechnung

- Zur Berechnung von  $m[i, j]$  werden nur Matrixwerte  $m[k, l]$  verwendet mit kleinerer Kettenlänge, also  $k-l+1 < i-j+1$
- Berechne die Matrix mit steigenden  $(i-j+1)$ -Werten



### Schritt 3: Berechnung der minimalen Anzahl Multiplikationen

- MATRIX-CHAIN-ORDER(p)

```

1 n ← length[p]-1
2 for i ← 1 to n do m[i,i] ← 0 // Initialisierung trivialer m[]-Werte
3 for l ← 2 to n do // Berechnung erfolgt in der Reihenfolge
                        // wachsender Kettenlängen l:
4   for i ← 1 to n-l+1 do // Iteriere durch alle Paare (i,j) mit Länge l
5     j ← i+l-1
6     m[i,j] ← ∞
7     for k ← i to j-1 do // Minimiere über alle möglichen k-Werte
8       q ← m[i,k] + m[k+1,j] + p[i-1] p[k] p[j]
9       if q < m[i,j] then
10        m[i,j] ← q
11      s[i,j] ← k
12
13 return m, s // der gesuchte Wert steht in m[1,n]
```

- Laufzeit:  $T_{\text{MATRIX-CHAIN-ORDER}}(n) = O(n^3)$

- Speicherbedarf:  $S_{\text{MATRIX-CHAIN-ORDER}}(n) = \Theta(n^2)$

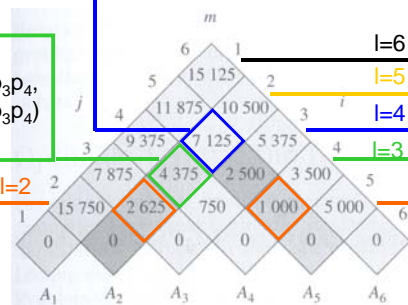


### Ein Beispiel für $n=6$ (Teil 1)

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} = 7125$$

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 p_3 p_4 \\ m[2,3] + m[4,4] + p_2 p_3 p_4 \end{cases} = 4375$$

$$m[2,3] = p_1 \cdot p_2 \cdot p_3 = 35 \cdot 15 \cdot 5 = 2625$$



$$l=6 \quad m[1,6] = 15125$$

$$m[4,5] = p_3 \cdot p_4 \cdot p_5 = 5 \cdot 10 \cdot 20 = 1000$$



### Schritt 4: Konstruktion einer optimalen vollständigen Klammerung

- Sei  $s[i, j]$  der  $k$ -Wert, der die optimale Lösung für das Teilprodukt  $A_{i..j}$  liefert, d.h.  $A_{i..j}$  soll durch  $(A_{i..k})(A_{k+1..j})$  berechnet werden

- MATRIX-CHAIN-ORDER(p)

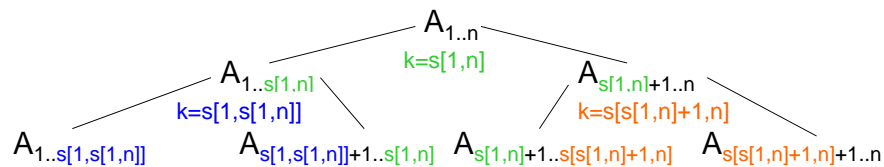
```

1 n ← length[p]-1
2 for i ← 1 to n do m[i,i] ← 0 // Initialisierung trivialer m[]-Werte
3 for l ← 2 to n do // Berechnung erfolgt in der Reihenfolge
                        // wachsender Kettenlängen l:
4   for i ← 1 to n-l+1 do // Iteriere durch alle Paare (i,j) mit Länge l
5     j ← i+l-1
6     m[i,j] ← ∞
7     for k ← i to j-1 do // Minimiere über alle möglichen k-Werte
8       q ← m[i,k] + m[k+1,j] + p[i-1] p[k] p[j]
9       if q < m[i,j] then
10        m[i,j] ← q
11      s[i,j] ← k
12
13 return m, s // der gesuchte Wert steht in m[1,n]
```



#### Schritt 4: Konstruktion einer optimalen vollständigen Klammerung

- Bestimmung einer optimalen vollständigen Klammerung mittels  $s[i,j]$ :
  - Berechnung von  $A_{1..n} = (A_{1..s[1,n]}) (A_{s[1,n]+1..n})$



- Inorder-Traversal des durch  $s[i,j]$  definierten Baumes:

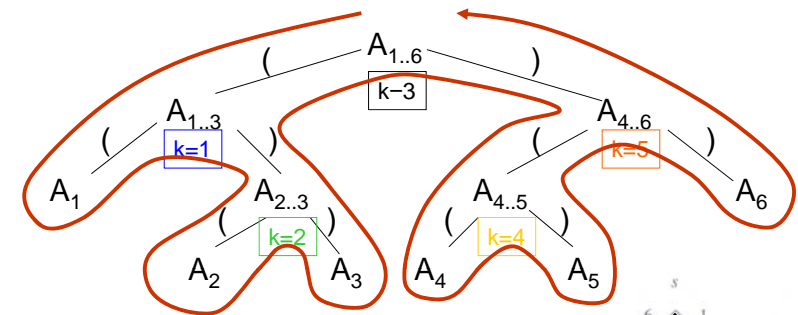
- PRINT-OPTIMAL-PARENS( $s,i,j$ )

```

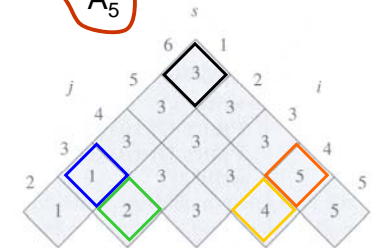
1 if i = j
2   then write "A"
3   else write "("
4       PRINT-OPTIMAL-PARENS(s,i,s[i,j])
5       PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
6   write ")"
    
```



#### Ein Beispiel für $n=6$ (Teil 2)



Optimale vollständige Klammerung:  
 $((A_1(A_2A_3))((A_4A_5)A_6))$



#### Abschließende Bemerkungen zur Dynamischen Programmierung

- Dynamische Programmierung erlaubt die effiziente Lösung von Optimierungsproblemen mit zwei Eigenschaften:
  1. **Optimale Teilstruktur:** Optimale Lösung kann aus optimalen Teillösungen konstruiert werden. Die optimalen Teillösungen können **unabhängig** voneinander bestimmt werden.
  2. **Überlappende Teilprobleme:** Es gibt eine (polynomielle) Anzahl von Teilproblemen, deren Lösungen immer wieder zur Lösung größerer Teilprobleme herangezogen werden.
- Entwicklung von Algorithmen nach dem Prinzip der Dynamischen Programmierung:
  1. **Struktur der optimalen Lösung** bestimmen (optimale Teilstruktur nachweisen)
  2. Eine **rekursive Lösung** entwickeln (Top-Down-Berechnung)
  3. Berechnung der optimalen Kosten durch Umkehr der Berechnungsreihenfolge (**Bottom-Up-Berechnung** + Memoisation)
  4. Über die optimalen Kosten die **optimale Lösung rekonstruieren**



## **Kapitel 7: NP-Vollständigkeit**

- 7.1 Formalisierung von Problemen 182
- 7.2 Komplexitätsklassen P, NP und NPC 184
- 7.3 NP-vollständige Probleme 186
- 7.4 Noch mehr NP-vollständige Probleme 193



## Kap. 7: NP-Vollständigkeit

**Formalisierung und Codierung von Problemen**  
**Komplexitätsklassen P, NP und NPC**  
**NP-vollständige Probleme, Reduktionsbeweis**  
**Noch mehr NP-vollständige Probleme**

## NP-Vollständigkeit

- bisher:
  - Probleme analysiert und effiziente Algorithmen / Datenstrukturen entwickelt
- Vorgehen, wenn man keinen effizienten Algorithmus findet:
  1. Vermutung: Es gibt einen effizienten Algorithmus
    - Weitersuchen!
  2. Vermutung: Es gibt keinen effizienten Algorithmus
    - Nachweis durch eine untere Laufzeitschranke
      - extrem schwierig, gelingt nur sehr selten
    - Nachweis, dass das Problem zu einer Klasse von scheinbar schwer zu lösenden Problemen gehört
      - viel einfacher, gelingt häufig!
- Ziel: Formalismus, mit dem man die Schwierigkeit eines Problems verdeutlichen kann



## Schwere und einfache Probleme

- |   |  |
|---|--|
| 1. Finde den längsten, einfachen Weg in einem Graphen   | <b>schwer!</b>   |
| 2. Finde den kürzesten, einfachen Weg in einem Graphen  | <b><math>O( E )</math></b>                                     |
| 3. Hamilton-Kreis: einfacher Kreis, der alle Knoten enthält   | <b>schwer!</b>   |
| 4. Euler-Tour: Kreis, der alle Kanten enthält   | <b><math>O( E )</math></b>                                     |
| 5. Clique: vollständiger Teilgraph mit $k$ Knoten   | <b>schwer!</b>   |
| 6. Independent Set: Teilgraph mit $k$ Knoten ohne Kanten  | <b>schwer!</b>   |
| 7. Färbbarkeit: adjazente Knoten haben unterschiedliche Farben. <ul style="list-style-type: none"><li>■ Färbbar mit zwei Farben?</li><li>■ Färbbar mit drei Farben?</li></ul> | <b><math>O( E )</math></b><br><b>schwer!</b><br><b>schwer!</b> |
| 8. Subset-Sum: geg. eine Menge ganzer Zahlen, lässt sich die Menge in zwei Teilmengen gleicher Summe zerlegen?  | <b>schwer!</b>   |



## 7.1 Formalisierung von ‚Problemen‘

- Problemtypen:
  - Optimierungsproblem: finde eine gültige Lösung mit einem minimalen Wert bzgl. einer Bewertungsfunktion
  - Entscheidungsproblem: prüfe, ob eine gültige Lösung existiert.
- Sei  $P$  ein Optimierungsproblem:
  - $E_P = \text{‚Gibt es eine Lösung für } P \text{ mit Wert } \leq k? \text{‘}$  ist das zugehörige Entscheidungsproblem ( **$k$ -Threshold-Problem**)
  - Eingabe ist die Eingabe von  $P$  und der Wert  $k$
  - Sei  $A$  ein Algorithmus für ein  $k$ -Threshold-Problem  $E_P$ ,  
 $P$  kann durch die wiederholte Anwendung von  $A$  mit binärer Suche gelöst werden
- Im folgenden betrachten wir nur Entscheidungsprobleme.
  - Komplexität von Optimierungsproblemen lassen sich über das zugehörige  $k$ -Threshold-Problem bewerten.





## Codierung von Problemen

- Codierung:
  - Abbildung von Objekten in die Menge der Binärstrings  $\{0,1\}^*$ 
    - ◆ Abstraktes Problem: Problem definiert über komplexe Objekte
    - ◆ Konkretes Problem: Funktion  $f: \{0,1\}^* \rightarrow \{0,1\}$
  - Codierung bildet ein abstraktes Problem auf ein konkretes Problem ab
  - für nicht belegte Eingabecodes  $x$  gilt  $f(x) = 0$
- Formale Sprachen:
  - konkretes Entscheidungsproblem  $f$  kann als Menge betrachtet werden:  
 $L_P = \{x \in \{0,1\}^* \mid f(x) = 1\}$
- Länge der Codierung
  - Ein abstraktes Problem mit Eingabe  $n$  lässt sich als ein konkretes Problem mit Eingabelänge  $l(n) = O(n^k)$ ,  $k$  konstant codieren



## Codierung von Problemen

- Bsp (Länge der Codierung):  
Graph mit  $n$  Knoten und  $m$  Kanten
- Verwende
  - 00: binär 0,                      01: binär 1,
  - 10: neues Listenelement        11: neue Liste
- Adj.-Liste zu Knoten  $i$ :  $11\text{bin}(i)10\text{bin}(\text{Nachbar } 1)10\text{bin}(\text{Nachbar } 2)\dots$
- Länge  $l(m,n)$  der binärcodierten Eingabe
  - ◆  $2\log_2 n$  pro Knotennummer, bei  $n$  Knoten und  $m$  Kanten:  
 $l(n,m) = n \cdot 2 \log_2 n + 2 \cdot m \cdot (2 + 2\log_2 n) = O(m \log n)$



## 7.2 Komplexitätsklassen P, NP und NPC

- $f: \{0,1\}^* \rightarrow \{0,1\}^*$  heißt polynomzeit-berechenbar  
g.d.w. ein Algorithmus  $A$  zur Berechnung von  $f$  existiert mit Laufzeit  $T_A(n) = O(n^k)$  für eine Konstante  $k$
- Komplexitätsklasse P:  
Menge aller konkreten Entscheidungsprobleme  $f: \{0,1\}^* \rightarrow \{0,1\}$ , die polynomzeit-berechenbar sind.
  - umgangssprachlich:
    - ◆ abstrakte Entscheidungsprobleme  $f$  sind in P, g.d.w. es eine Codierung polynomieller Länge gibt und das zugehörige konkrete Entscheidungsproblem in P
    - ◆ Ein Optimierungsproblem  $f$  ist in P, g.d.w. zugehörige  $k$ -Threshold-Problem  $E_f$  in P ist.
  - Polynome sind gegen Verkettung abgeschlossen, d.h. sind  $f$  und  $g$  Polynome, so ist  $h: x \rightarrow f(g(x))$  ebenfalls ein Polynom
  - Solange die Codierung polynomiell ist, ändert sie nichts an der Tatsache, ob ein abstraktes Problem in P ist oder nicht.



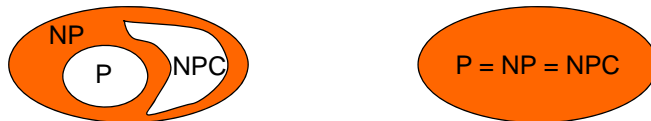
## Die Klasse NP

- $f: \{0,1\}^* \rightarrow \{0,1\}$  heißt polynomzeit-verifizierbar
  - zu jeder Eingabe  $x \in \{0,1\}^*$  es existiert ein Zertifikat  $y \in \{0,1\}^*$  mit  $|y| = O(|x|^k)$
  - es existiert ein Algorithmus  $A$  mit  $A(x,y) = 1$  g.d.w.  $f(x) = 1$  und  $T_A(n) = O(n^k)$  für ein konstantes  $k$
- umgangssprachlich:
  - Zertifikat stellt die Lösung dar, der Algorithmus  $A$  überprüft in polynomieller Zeit, ob die Lösung korrekt ist.
  - Bsp. für Zertifikate:
    - ◆  $k$ -Clique: Menge der Knoten, die eine  $k$ -Clique bilden
    - ◆ 3-Färbbarkeit: Funktion, die jedem Graphknoten eine Farbe zuweist
- Komplexitätsklasse NP:
  - Menge aller konkreten Entscheidungsprobleme  $f: \{0,1\}^* \rightarrow \{0,1\}$ , die polynomzeit-verifizierbar sind
  - Offensichtlich gilt:  $P \subseteq NP$ . Gilt  $P = NP$ ? (vermutlich nicht!)



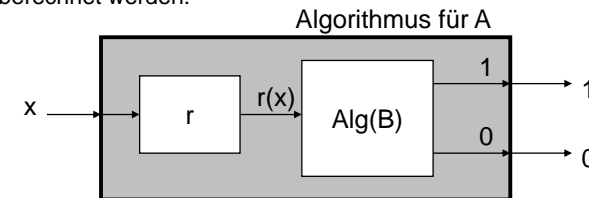
## NP-Vollständigkeit

- Da  $P \subseteq NP$  gilt, können wir schwierige Probleme nicht identifizieren
- Was sind die schwierigsten Probleme in NP?
- NP-Vollständigkeit:
  - ein konkretes Entscheidungsproblem  $A \in NP$  heißt NP-vollständig, g.d.w. aus  $A \in P \Rightarrow P = NP$ , bzw.  $P \neq NP \Rightarrow A \notin P$
- Komplexitätsklasse NPC
  - NPC ist die Menge der NP-vollständigen Entscheidungsprobleme
  - Ein Optimierungsproblem  $f$  heißt NP-schwer, wenn das zugehörige k-Threshold-Problem  $E_f$  NP-vollständig ist
- Zwei Szenarien sind denkbar:



## Polynomzeit-Reduktion

- Seien A, B Entscheidungsprobleme:
- Wie können wir zeigen, dass B mindestens so komplex ist wie A?
  - Suche eine Reduktionsfunktion  $r: \{0,1\}^* \rightarrow \{0,1\}^*$  mit den Eigenschaften:
    - ◆  $x \in A \Leftrightarrow r(x) \in B$
    - ◆  $r$  kann in polynomieller Zeit berechnet werden
  - Gibt es ein entsprechendes  $r$ , so heißt A polynomzeit-reduzierbar auf B, oder  $A \leq_P B$
  - Mit der Funktion  $r$  und einen Algorithmus  $Alg(B)$  für B kann A wie folgt berechnet werden:



- Angenommen  $B \in P \Rightarrow A \in P$



## Nachweis von NP-Vollständigkeit

- Satz: Ein Problem A ist NP-vollständig ( $A \in NPC$ ), g.d.w.
    1.  $A \in NP$
    2. für alle  $B \in NP$  gilt:  $B \leq_P A$
  - Angenommen  $A \in P$ , dann können wir Problem B mit folgenden Algorithmus  $Alg(B)$  lösen:
    - ◆ Berechne Reduktionsfunktion  $r(x)$
    - ◆ Verwende Algorithmus für A
  - $Alg(B)$  hat polynomielle Laufzeit, folglich gilt  $B \in P$
  - Damit gilt für alle  $B \in NP$ :  $B \in P$ , also  $P = NP$
  - Wie weist man nun NP-Vollständigkeit eines Problems A nach?
    - zeige, dass  $A \in NP$  gilt
    - wähle ein beliebiges Problem  $Q \in NPC$
    - zeige, dass  $Q \leq_P A$  gilt (Polynomzeit-Reduktion)
- $Q \in NPC$ , also gilt für alle  $B \in NP$ :  $B \leq_P Q$ . mit  $Q \leq_P A$  folgt für alle  $B \in NP$ :  $B \leq_P A$



## 7.3 NP-vollständige Probleme

- Satisfiability-Problem (SAT)
  - Eingabe: logischer Ausdruck A in CNF (konjunktiver Normalform)
    - ◆ boole'schen Variablen  $x_i$  (mögliche Belegung: 0 oder 1)
    - ◆ NOT, AND und OR Operatoren
    - ◆ CNF: AND-Verknüpfung von *Klauseln*; eine Klausel ist eine OR-Verknüpfung von *Literalen* ((potentiell negierten) Variablen)
  - Ausgabe: 1 g.d.w. es eine Belegung gibt, so dass A wahr(1) ist.
- Beispiel:
 
$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Ausgabe: 1      Belegung:  $x_1=1, x_2=1, x_3=0, x_4=1$
- Cook's Theorem:  $SAT \in NPC$



## Das erste Problem in NPC

### ■ Cook's Beweis (grobe Skizze)

#### 1. Zeige $\text{SAT} \in \text{NP}$

- Zertifikat: gültige Belegung für die Variablen
  - Verifikation durch den folgenden Algorithmus A:
    - Sei  $x$  eine Eingabe des SAT-Problems (logischer Ausdruck in CNF);  $y$ , das zugehörige Zertifikat (Belegung der Variablen)
    - Setze die Variablenbelegung  $y$  in den Ausdruck  $x$  ein
    - für jede Klausel von  $x$ : prüfe, ob die Klausel wahr ist
  - Es gilt  $|y| = O(|x|)$  (ein Bit für jede Variable, die in  $x$  vorkommt)
  - Algorithmus A hat polynomielle Laufzeit  $T_A(n) = O(n)$
- Es folgt, SAT ist polynomzeit-verifizierbar und somit  $\text{SAT} \in \text{NP}$



## Das erste Problem in NPC

### ■ Cook's Beweis (grobe Skizze)

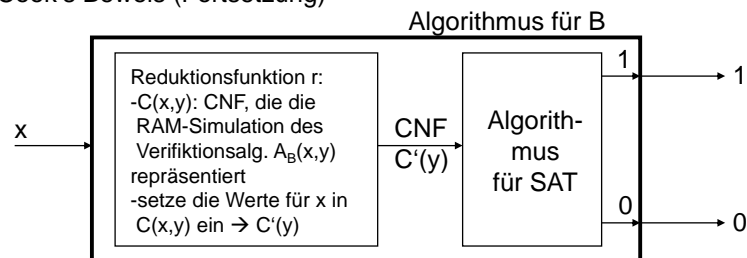
#### 2. Zeige für ein beliebiges $B \in \text{NP}$ : $B \leq_p \text{SAT}$

- Sei  $A_B(x,y)$  der Algorithmus zur Polynomzeit-Verifikation von  $B$   
 $A_B(x,y)$  benötige  $T(n)$  Schritte auf einer RAM  
 Die Funktionsweise einer RAM kann auf der Basis logischer Schaltkreise beschrieben werden  
 Die Funktionsweise logischer Schaltkreise kann durch boole'sche Formeln beschrieben werden  
 Der Zustand einer RAM kann vollständig durch eine boole'sche Formel in CNF beschrieben werden (modelliere Register, Akkumulator, Rechenwerk, Steuerwerk, etc.)  
 Aus dem Zustand der RAM zum Zeitpunkt  $i$  kann der Zustand zum Zeitpunkt  $i+1$  durch boole'sche Formeln beschrieben werden.  
 Eine vollständige Rechnung einer RAM mit  $t$  Schritten kann durch eine CNF polynomieller Länge beschrieben werden



## Das erste Problem in NPC

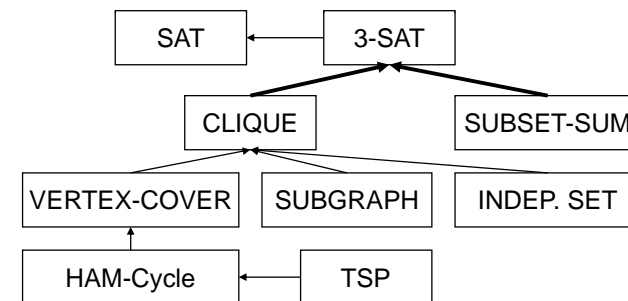
### ■ Cook's Beweis (Fortsetzung)



- Ist  $x \in B$ :
  - so existiert ein  $y$  mit  $A_B(x,y)=1$ , für die Belegung  $y$  ist somit  $C'(y)=1$   
 $\Rightarrow C'(y)$  ist erfüllbar  $\Rightarrow r(x)=C'(y) \in \text{SAT}$
- Ist  $x \notin B$ :
  - so gilt für alle  $y$ :  $A_B(x,y)=0$ , für alle Belegungen ist somit  $C'(y)=0 \Rightarrow C'(y)$  ist nicht erfüllbar  $\Rightarrow r(x)=C'(y) \notin \text{SAT}$
- $|r(x)| = O(n^k)$  und  $r(x)$  kann aus  $x$  in  $O(n^k)$  berechnet werden.



## NP-vollständige Probleme



3-SAT: Jede Klausel hat maximal 3 Literale

VERTEX-COVER: Knotenmenge  $V'$ ,  $|V'| \leq k$ , jede Kante ist zu einem Knoten in  $V'$  inzident

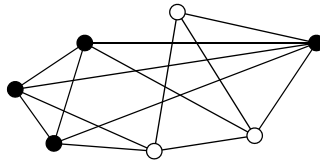
HAM-Cycle: Gibt es einen einfachen Kreis, der alle Knoten enthält

TSP: Traveling Salesperson Problem (kürzeste Rundtour in einem vollständigen, kantengewichteten Graphen)



## Das CLIQUE-Problem

- $\text{CLIQUE} = \{ \langle G, k \rangle : G \text{ ist ein Graph mit einer } k\text{-Clique} \}$
- Bsp.:



- CLIQUE ist NP-vollständig
  - Beweis Teil 1: Zeige  $\text{CLIQUE} \in \text{NP}$ 
    - ◆ Zertifikat:  $V' \subseteq V$ :  $V'$  ist eine  $k$ -CLIQUE, es gilt  $|V'| = O(|V|)$
    - ◆ Verifikationsalgorithmus  $A(\langle G=(V,E), k \rangle, V')$ 

```

if  $|V'| \neq k$  then return FALSE
for each pair  $v, w \in V'$  do
    if  $\{v, w\} \notin E$  then return FALSE
return TRUE
                    
```



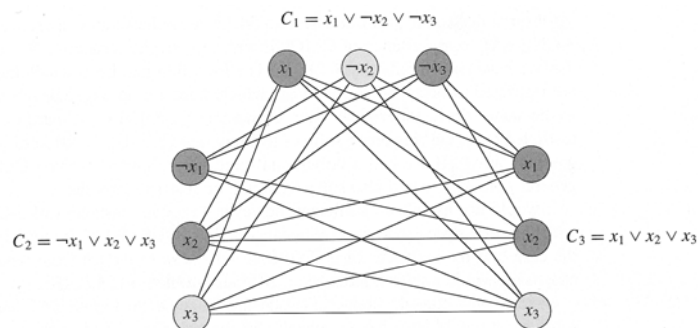
## Das CLIQUE-Problem

- Beweis Teil 2: Zeige  $3\text{-SAT} \leq_p \text{CLIQUE}$ 
  - ◆ Ziel: Wandle Eingabe  $x$  des 3-SAT Problems in Eingabe  $r(x)$  des CLIQUE-Problems mit  $x \in 3\text{-SAT} \Leftrightarrow r(x) \in \text{CLIQUE}$
  - ◆ Eingabe des 3-SAT Problems: 3-CNF:  $C_1 \wedge C_2 \wedge C_3 \dots \wedge C_n$   
jede Klausel  $C_i$ :  $l_i^1 \vee l_i^2 \vee l_i^3$  ( $l_j$ :  $j$ -te Literal der  $i$ -ten Klausel)
  - ◆ Funktion  $r(x)$ : baut aus 3-CNF einen Graph  $G=(V,E), k$ 
    - ◆ Knoten  $V$ :  $v_i^j$  je ein Knoten pro Literal
    - ◆ Kanten  $E$ :  $v_i^r$  und  $v_j^s$  sind adjazent  $\Leftrightarrow$ 
      1. zugehörige Literale gehören zu unterschiedlichen Klauseln, d.h.  $i \neq j$
      2. und zugehörige Literale können gleichzeitig erfüllt werden, d.h.  $l_i^r \neq \neg l_j^s$
    - ◆ Clique-Größe: Anzahl der Klauseln, d.h.  $k=n$
  - ◆  $r(x)$  kann in  $O(N^2)$  berechnet werden



## Das CLIQUE-Problem

- Bsp.: Reduktionsfunktion  $r$ :
  - 3-CNF:  $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
  - Anzahl Klauseln: 3
  - Eingabe für CLIQUE:  $\langle G=(V,E), k \rangle$ ,  $k=3$



## Das CLIQUE-Problem

- Beweis Teil 2 (Fortsetzung):
  - Zeige  $x \in 3\text{-SAT} \Rightarrow r(x) \in \text{CLIQUE}$ 
    - ◆  $x$  ist erfüllbar, d.h. in jeder Klausel gibt es mindestens ein erfülltes Literal, d.h.  $\forall 1 \leq i \leq n: \exists 1 \leq j \leq 3 : l_i^j = 1$
    - ◆  $V'' = \{ v_i^j \in V \mid l_i^j = 1 \}$ ,  $V' \subseteq V''$ : wähle pro Klausel genau ein Literal
    - ◆  $|V'| = k$ ,  $V'$  eine Clique, denn  $\forall \{v_i^r, v_j^s\} \subseteq V'$ :
      1.  $i \neq j$  (pro Klausel wurde nur ein Literal gewählt)
      2.  $l_i^r = 1$ ;  $l_j^s = 1$  gemäß  $V''$ , also gilt  $l_i^r \neq \neg l_j^s$
  - ◆ Zeige  $r(x) \in \text{CLIQUE} \Rightarrow x \in 3\text{-SAT}$ 
    - ◆ Sei  $V'$  eine  $k$ -CLIQUE in  $r(x)$ ,  $L'$  die zugehörigen Literale
    - ◆  $L'$  enthält aus jeder Klausel ein Literal, da  $|V'|=k$  und Knoten zu Literalen innerhalb einer Klausel nicht adjazent sind.
    - ◆ Alle Literale aus  $L'$  können erfüllt werden, da Knoten zu inkonsistenten Literalen nicht adjazent sind.
    - ◆ Setze Variable  $x_i = 1$  falls  $x_i \in L'$ ,  $x_i = 0$  falls  $\neg x_i \in L'$ , ansonsten beliebig. Die Belegung erfüllt die 3-CNF  $x$ .
  - ◆ Es gilt:  $r$  ist polynomzeit-berechenbar und  $x \in 3\text{-SAT} \Leftrightarrow r(x) \in \text{CLIQUE}$ , d.h.  $3\text{-SAT} \leq_p \text{CLIQUE}$



## Das SUBSET-SUM Problem

### ■ Problem (SUBSET-SUM)

■ geg.: Menge von S von Zahlen, Zielwert t

■ Gibt es ein  $S' \subseteq S$  mit  $\sum_{s \in S'} s = t$

■ Bsp.:  $t=300$   $S = \{ 3, 17, 39, 48, 103, 111, 113, 132, 254 \}$

■  $S' = \{ 17, 48, 103, 132 \}$

■ arithmetisches Problem, Größe der Zahlen muss bei der Komplexitätsanalyse berücksichtigt werden

### ■ Theorem 34.15:

SUBSET-SUM ist NP-vollständig

### ■ Beweis Teil 1: SUBSET-SUM $\in$ NP

◆ Zertifikat y: Indizes der Elemente in S, die zu  $S'$  gehören

◆  $A(\langle S, t \rangle, y)$ : addiere die Elemente in  $S'$  und vergleiche mit t

◆ Laufzeit:  $O(N)$



## Das SUBSET-SUM Problem

### ■ Beweis Teil 2: $3\text{-SAT} \leq_p \text{SUBSET-SUM}$

◆ Ziel: Wandle Eingabe x des 3-SAT Problems in Eingabe r(x) des SUBSET-SUM Problems, so dass x erfüllbar ist g.d.w. r(x) eine Teilsumme mit Wert t hat.

◆ Sei F ein logischer Ausdruck in 3-CNF

◆ Entferne alle Klauseln die Variable und ihr Komplement enthalten (sind sowieso immer erfüllt)

◆ Entferne alle Variablen, die in keiner Klausel vorkommen (spielen bzgl. der Erfüllbarkeit keine Rolle)

◆ F bestehe aus den Variablen  $x_1, \dots, x_n$  und den Klauseln  $C_1, \dots, C_k$

◆ Reduktionsfunktion r:

◆ verwende Zahlen im Zehnersystem mit  $n+k$  Stellen

◆ die ersten n Stellen repräsentieren Variablen, die folgenden k Stellen repräsentieren Klauseln

◆ konstruiere je zwei Zahlen  $v_i, v_i'$  und  $s_j, s_j'$  für Variablen und Klauseln:



## Das SUBSET-SUM Problem

### ■ Beweis Teil 2: (Fortsetzung)

◆  $v_i$ : 1 an Stelle  $x_i$ , 1 an Stellen aller Klauseln, die  $x_i$  enthalten

◆  $v_i'$ : 1 an Stelle  $x_i$ , 1 an Stellen aller Klauseln, die  $\neg x_i$  enthalten

◆  $s_j$ : 1 an Stelle  $C_j$   $s_j'$ : 2 an Stelle  $C_j$  t:  $\underbrace{11\dots1}_{n \text{ mal}} \underbrace{444\dots4}_{k \text{ mal}}$

### ■ Beispiel: 3-CNF:

$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge$   
 $C_1$

$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge$   
 $C_2$

$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge$   
 $C_3$

$(x_1 \vee x_2 \vee x_3)$   
 $C_4$

VAR:	$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$v_1$	1	0	0	1	0	0	1
$v_1'$	1	0	0	0	1	1	0
$v_2$	0	1	0	0	0	0	1
$v_2'$	0	1	0	1	1	1	0
$v_3$	0	0	1	0	0	1	1
$v_3'$	0	0	1	1	1	0	0
$s_1$	0	0	0	1	0	0	0
$s_1'$	0	0	0	2	0	0	0
$s_2$	0	0	0	0	1	0	0
$s_2'$	0	0	0	0	2	0	0
$s_3$	0	0	0	0	0	1	0
$s_3'$	0	0	0	0	0	2	0
$s_4$	0	0	0	0	0	0	1
$s_4'$	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4



## Das SUBSET-SUM Problem

### ■ Beweis Teil 2: (Fortsetzung)

◆ r(x) hat Länge  $O((n+k)^2)$  und kann in  $O((n+k)^2)$  Zeit berechnet werden

◆ Zeige  $F \in 3\text{-SAT} \Rightarrow r(F) \in \text{SUBSET-SUM}$

F ist erfüllbar; sei B eine Belegung, die F erfüllt.

Wähle  $S' = \{ v_i \mid x_i = 1 \text{ in } B \} \cup \{ v_i' \mid x_i = 0 \text{ in } B \}$ , sei  $t' = \sum_{s \in S'} s$

$t'$  hat an den ersten n Stellen eine 1, da entweder

$v_i \in S'$  oder  $v_i' \in S'$

$t'$  hat an den hinteren k Stellen eine 1, 2 oder 3, da jede Klausel erfüllt ist (daher  $> 0$ ) und jede Klausel max. 3 Literale hat (daher  $\leq 3$ )

Erweitere  $S'$  um Variablen  $s_j$  und/oder  $s_j'$ , so dass an den hinteren k Stellen je eine 4 steht.

Offensichtlich gilt  $\sum_{s \in S'} s = t$ .



## Das SUBSET-SUM Problem

### ■ Beweis Teil 2: (Fortsetzung)

◆ Zeige:  $r(F) \in \text{SUBSET-SUM} \Rightarrow F \in 3\text{-SAT}$

Sei  $S'$  die Teilmenge der Zahlen mit  $\sum_{s \in S'} s = t$

Um in der Summe an der  $i$ -ten Stelle eine 1 zu erhalten, muss gelten: entweder  $v_i \in S'$  oder  $v_i' \in S'$ .

Wähle die Belegung  $x_i=1$  falls  $v_i \in S'$ ,  $x_i=0$  falls  $v_i' \in S'$ .

Für jede Klausel  $C_j$  gilt: die  $n+j$ -te Stelle ist 4, da  $s_j + s_j' = 3$  gibt es ein  $v_i$  oder  $v_i'$  mit einer 1 an der  $n+j$ -ten Stelle.

Angenommen, es ist ein  $v_i$ :

$x_i = 1$  und kommt in  $C_j$  vor, damit ist  $C_j$  erfüllt.

Angenommen, es ist ein  $v_i'$ :

$x_i = 0$  und  $\neg x_i$  kommt in  $C_j$  vor, damit ist  $C_j$  erfüllt.

◆ Es gilt:  $r$  ist polynomzeit-berechenbar und  $F \in 3\text{-SAT} \Leftrightarrow r(F) \in \text{SUBSET-SUM}$ , d.h.  $3\text{-SAT} \leq_p \text{SUBSET-SUM}$ .



## 7.4 Noch mehr NP-vollständige Probleme

### ■ Beispiele aus

■ Garey/Johnson, Computers and Intractability (1979)

### ■ Graphen:

■ Aufteilung in kantendisjunkte Dreiecke

■ Aufteilung in weniger als  $k$  kantendisjunkte Bäume

■ Gibt es einen bipartiten Subgraph mit mehr als  $K$  Kanten?

■ Gibt es einen planaren Subgraph mit mehr als  $K$  Kanten?

■ Enthält ein Graph  $G$  einen gegebenen Subgraph  $H$ ?

■ Gibt es einen Spannbaum, in dem jeder Knotengrad  $< K$  ist?

■ Gibt es einen Spannbaum, in dem die Summe über allen paarweisen Distanzen zwischen Knoten  $< K$  ist?



## Noch mehr NP-vollständige Probleme

### ■ Netzwerk-Design

#### ■ (K-th SHORTEST PATH)

Geg. Graph mit positiven Kantengewichten, Start- und Zielknoten, zwei Zahlen  $K$  und  $B$

Gibt es  $K$  kantendisjunkte Wege von  $s$  nach  $t$ , jeder mit Länge  $< B$ ?

Das Problem ist polynomzeit-reduzierbar, es ist unbekannt, ob es in NP ist.

#### ■ QUADRATIC ASSIGNMENT

Geg.  $n$  Objekte mit paarweisen Abstandskosten  $c_{ij}$ ,  $m$  Slots mit paarweisen Abständen  $d_{ij}$

Gibt es eine Zuordnung  $f: \{1, \dots, n\} \rightarrow \{1, \dots, m\}$  der Objekte zu den

Slots mit 
$$\sum_{\{i,j\} \subseteq \{1, \dots, n\}} c_{ij} d_{f(i)f(j)} \leq B$$



## Noch mehr NP-vollständige Probleme

### ■ Mengen und Partitionen

#### ■ MINIMUM COVER

Geg. eine Sammlung  $C$  von Teilmengen über eine endl. Menge  $S$ , eine positive Zahl  $K$

Gibt es ein  $C' \subseteq C$  mit  $|C'| \leq K$ , die  $S$  überdeckt, d.h.  $\bigcup_{c \in C'} c = S$

#### ■ BIN PACKING

Geg. Menge  $U$  von Objekten mit positiver Größe, eine positive Behältergröße  $B$ , eine positive Zahl  $K$

Gibt es eine Aufteilung von  $U$  in  $\leq K$  Teilmengen, so dass für jede Teilmenge  $U_i$  gilt:  $\sum_{u \in U_i} u \leq B$



## Noch mehr NP-vollständige Probleme

### ■ Zeichenketten

#### ■ SHORTEST COMMON SUPERSEQUENCE

Geg. eine Menge  $R$  von Strings, eine positive Zahl  $K$

Gibt es einen String  $s$  mit  $|s| \leq K$ , der alle  $r \in R$  als Teilsequenz (mit Unterbrechungen) enthält, d.h.  $s = w_0 r_0 w_1 r_1 \dots w_k r_k w_{k+1}$  mit

$$r_0 r_1 \dots r_k = r?$$

#### ■ SHORTEST COMMON SUPERSTRING

Geg. eine Menge  $R$  von Strings, eine positive Zahl  $K$

Gibt es einen String  $s$  mit  $|s| \leq K$ , der alle  $r \in R$  als Teilstring (ohne Unterbrechungen) enthält, d.h.  $s = w_0 r w_1$ ?

#### ■ LONGEST COMMON SUBSEQUENCE

Geg. eine Menge  $R$  von Strings, eine positive Zahl  $K$

Gibt es einen String  $s$  mit  $|s| \geq K$ , der eine Teilsequenz aller Strings in  $R$  ist?



## Noch mehr NP-vollständige Probleme

### ■ Scheduling

#### ■ MULTIPROCESSOR SCHEDULING

Geg. Anzahl  $m$  von Prozessoren, Menge  $T$  von Aufgaben, eine positive Bearbeitungsdauer  $z(t)$  für jede Aufgabe und eine positive Zahl  $D$  (Deadline)

Können die Aufgaben aus  $T$  auf die  $m$  Prozessoren verteilt werden, so dass alle Aufgaben bis zum Zeitpunkt  $D$  bearbeitet sind?

#### ■ TIMETABLE DESIGN (informell)

Gibt es einen gültigen Stundenplan für  $H$  Arbeitsperioden,  $C$  Handwerker und  $T$  Aufgaben

### ■ Packungsprobleme

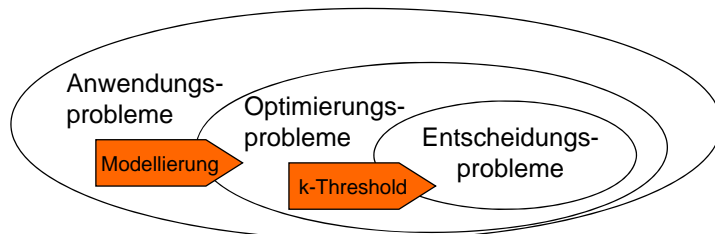
#### ■ KNAPSACK

Geg. eine Menge  $U$  von Objekten mit einer Größe  $s(u) > 0$  und einem Wert  $v(u) > 0$ , zwei positive Zahlen  $B$  und  $K$

Gibt es eine Teilmenge  $U'$  von  $U$  mit  $\sum_{u \in U'} s(u) \leq B$  und  $\sum_{u \in U'} v(u) \geq K$



## Komplexität von Anwendungsprobleme



### ■ Komplexitätsaussagen erfolgen über Entscheidungsprobleme

#### ■ Optimierungsprobleme:

◆  $k$ -Threshold-Problem (+ binäre Suche) definiert einen klaren Bezug zu Entscheidungsproblem

◆ Komplexitätsaussage hat uneingeschränkte Gültigkeit

#### ■ Anwendungsproblemen (z.B. Bioinformatik)

◆ Modellierung (Was sind die Freiheitsgrade, Was ist eine Energiefunktion, etc.) definiert ein Optimierungsproblem

◆ Komplexitätsaussage hat nur eingeschränkte Gültigkeit

„Problem  $X$  ist NP-schwer unter einer gegebenen Modellierung“



## **Kapitel 8: Lösen schwerer Probleme**

- 8.1 Approximationsalgorithmen 200
- 8.2 Exakte Verfahren 209
- 8.3 Heuristische Verfahren 230



## Kap. 8: Lösen schwerer Probleme

### Approximationsalgorithmen Exakte Verfahren Heuristische Verfahren

## Lösen schwerer Probleme

- Wie kann man ein Problem P angehen, von dem man weiß/vermutet, dass es NP-schwer ist?
- 1. Exakte Verfahren
  - Algorithmus mit exponentieller Laufzeit, der P löst
  - wird angewendet, falls
    - ◆ exakte Lösung wichtig ist,
    - ◆ notwendige Rechenkapazität zur Verfügung steht
    - ◆ Eingabegrößen hinreichend klein sind
- 2. Approximationsalgorithmen
  - Algorithmus mit polynomieller Laufzeit, der P mit einem garantierten max. Fehler löst
  - wird angewendet, falls
    - ◆ 1. nicht möglich ist
    - ◆ Fehlerabschätzung bei der Lösung verlangt wird (z.B. Risikoabschätzung, stat. Analyse der Ergebnisse, etc.)
- 3. Heuristische Verfahren
  - Algorithmus mit polynomieller Laufzeit, der eine *wahrscheinlich* gute Lösung für P (ohne Gütegarantie!) berechnet

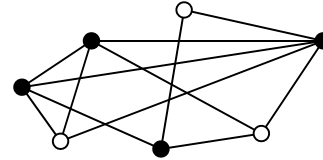


## 8.1 Approximationsalgorithmen

- Approximationsfaktor:
  - Annahme: jede Lösung hat positive Kosten
  - Ein Algorithmus A hat einen Approximationsfaktor  $\rho(n)$ , falls für jede Input-Größe n gilt:  $\max(C/C^*, C^*/C) \leq \rho(n)$   
C: von A gefundene Lösung; C\*: optimale Lösung
  - A heißt  $\rho(n)$ -Approximations-Algorithmus.
  - Falls  $\rho(n)$  konstant ist, auch  $\rho$ -Approximations-Algorithmus
  - Ein 1-Approximations-Algorithmus ist ein exakter Algorithmus.
- Approximationsschema:
  - Algorithmus A für ein Problem P, der neben der Eingabe für P eine Zahl  $\varepsilon$  als Parameter hat.
  - A heißt Approximationsschema, falls für jedes  $\varepsilon > 0$  A ein  $(1+\varepsilon)$ -Approximations-Algorithmus ist.
  - A heißt Polynomzeit-Approximationsschema, falls A für jedes konstante  $\varepsilon > 0$  einen max. polynomiellen Laufzeitbedarf hat.



## Ein Approximations-Algorithmus für Vertex-Cover

- Problem (Opt-Vertex-Cover):
  - gegeben: ungerichteter Graph  $G=(V,E)$
  - gesucht: minimale Teilmenge  $C \subseteq V$  mit:  
 $\forall (v,w) \in E : v \in C \text{ oder } w \in C$
  - Teilmengen mit dieser Eigenschaft werden als **Knotenüberdeckung** bezeichnet.
- Bsp:

schwarze Knoten  
sind ein Vertex-Cover
- Opt-Vertex-Cover ist ein NP-schweres Optimierungs-problem.



## Vertex-Cover ist NP-schwer

- VERTEX-COVER =  
 $\{ \langle G, k \rangle \mid \text{Der Graph } G=(V,E) \text{ besitzt eine Knotenüberdeckung der Größe } k \}$

### ■ Theorem 34.12:

Das k-Threshold-Problem VERTEX-COVER ist NP-vollständig.

Beweis:

Teil 1: VERTEX-COVER  $\in$  NP

Zertifikat  $y$ : Knotenmenge  $V'$  (die Knotenüberdeckung)

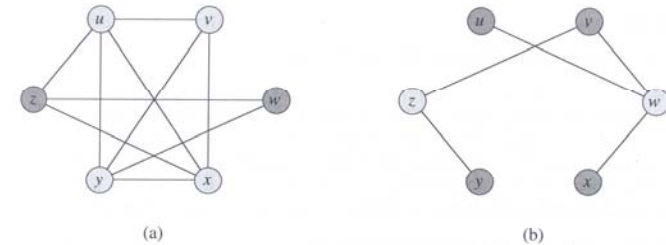
- ◆  $|y| = O(|V|)$
- ◆ Es kann in  $O(|V|+|E|)$  Zeit geprüft werden, ob  $|V'| = k$  und, eine Knotenüberdeckung ist.

Teil 2: CLIQUE  $\leq_p$  VERTEX-COVER

Sei  $\hat{G} = (V, \hat{E})$  mit  $\hat{E} = \{ (u,v) \in V \times V \mid (u,v) \notin E \}$ .  $\hat{G}$  ist der Komplement-Graph zu  $G$ .



## VERTEX-COVER ist NP-schwer



- Sei  $\Gamma$  die Menge der ungerichteten Graphen
- Reduktionsfunktion  $r(x): \Gamma \times \mathbb{N} \rightarrow \Gamma \times \mathbb{N}$   
 Sei  $\langle G, k \rangle$  eine Eingabe des CLIQUE-Problems, dann definieren wir  $r(\langle G, k \rangle) = \langle \hat{G}, |V| - k \rangle$ .
- Teil 2.1:  $r(x)$  ist in Zeit  $O(|V|^2)$  berechenbar:
  - konvertiere die Adjazenzlisten in eine Adjazenzmatrix und
  - invertiere alle nicht-diagonalen Matrixelemente



## VERTEX-COVER ist NP-schwer

- Teil 2.2:  $\langle G, k \rangle \in \text{CLIQUE} \Rightarrow \langle \hat{G}, |V| - k \rangle \in \text{VERTEX-COVER}$   
 Sei  $V'$  eine k-Clique. Wir betrachten die Knotenmenge  $V'' = V - V'$  in  $\hat{G}$ :

- ◆  $|V''| = |V| - k$
- ◆ Sei  $e = (v, w) \in \hat{E}$   
 $\Rightarrow (v, w) \notin E$   
 $\Rightarrow$  entweder  $v \notin V'$  oder  $w \notin V'$ , da  $V'$  eine Clique ist  
 $\Rightarrow e$  wird durch  $V''$  in  $\hat{G}$  überdeckt.

- Teil 2.3:  $\langle \hat{G}, |V| - k \rangle \in \text{VERTEX-COVER} \Rightarrow \langle G, k \rangle \in \text{CLIQUE}$

Sei  $V'$  eine Knotenüberdeckung der Größe  $|V| - k$  in  $\hat{G}$ . Wir betrachten  $V'' = V - V'$  in  $G$ :

- ◆  $|V''| = k$
- ◆ Sei  $e = (v, w) \in V'' \times V''$   
 $\Rightarrow e \notin \hat{E}$ , da sonst  $V'$  keine gültige Knotenüberdeckung wäre  
 $\Rightarrow e \in E \Rightarrow V''$  ist eine Clique.



## Algorithmen für Vertex-Cover

- APPROX-VERTEX-COVER(  $G$  )

```

C ← ∅
E' ← E[G]
while E' ≠ ∅ do
    (u,v) ← arbitrary e ∈ E'
    C ← C ∪ {u, v}
    E' ← E' \ ( { (u,w) | w ∈ Adj[u] } ∪ { (w,v) | w ∈ Adj[v] } )
return C
    
```

- Greedy-Strategie: nehme solange Knoten hinzu bis alle Kanten überdeckt sind.
- Offensichtlich ist  $C$  am Ende von solve\_vertex\_cover ein Vertex-Cover.
- Laufzeit:  $O(|E|)$  bei geschickter Implementierung der Mengenoperationen



## Algorithmen für Vertex-Cover

### ■ APPROX-VERTEX-COVER-2( G )

$C \leftarrow \emptyset$

$V' \leftarrow V[G]$

$E' \leftarrow E[G]$

**while**  $E' \neq \emptyset$  **do**

$v \leftarrow \text{select } v \in V' \text{ with maximal node degree in } G=(V', E')$

$C \leftarrow C \cup \{v\}$

$E' \leftarrow E' \setminus \{ (v, w) \mid w \in \text{Adj}[v] \}$

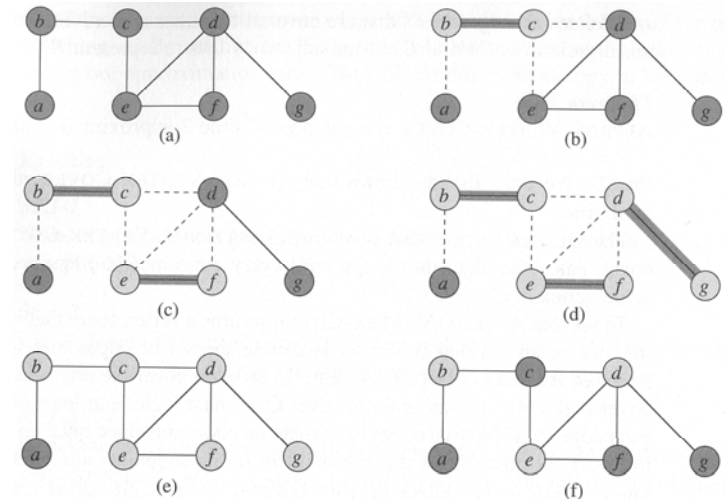
**return** C

- Greedy-Strategie: nehme solange Knoten hinzu bis alle Kanten überdeckt sind.
- Offensichtlich ist C am Ende von solve\_vertex\_cover ein Vertex-Cover.
- Laufzeit:  $O(|E|)$  bei geschickter Implementierung der Mengenoperationen



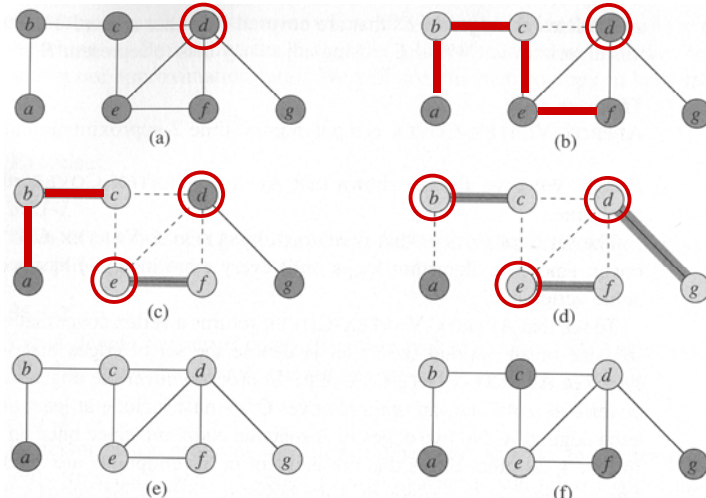
## Vertex-Cover (berechnet mit APPROX-VERTEX-COVER())

### ■ Bsp:



## Vertex-Cover (berechnet mit APPROX-VERTEX-COVER2())

### ■ Bsp:



## Vertex-Cover

### ■ Theorem 35.1:

APPROX-VERTEX-COVER ist eine 2-Approximationsalgorithmus mit polynomieller Laufzeit.

- Beweis Teil 1: APPROX-VERTEX-COVER hat polynomielle Laufzeit

◆ siehe vorherige Seite.

- Beweis Teil 2:  $|C| / |C^*| \leq 2$

$C^*$ : minimaler Vertex-Cover,

C: von APPROX-VERTEX-COVER berechnete Knotenüberdeckung

Sei A die Menge der Kanten, die in Zeile (\*\*) ausgewählt werden.

◆ Kanten in A haben keinen gemeinsamen Knoten

=> zur Überdeckung der Kanten in A benötigt man  $|A|$  Knoten

da  $A \subseteq E$ , folgt  $|C^*| \geq |A|$

◆ in jedem Schleifendurchlauf wird A um eine Kante, C um zwei Knoten erweitert, es folgt  $|C| = 2 |A|$

Insgesamt gilt  $|C| = 2 |A| \leq 2 |C^*|$ .



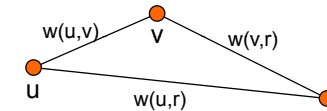
## Vertex-Cover

- Kommentare:
  - Wie funktionieren die Beweise?
    - ◆ optimale Lösung ist unbekannt, finde zunächst eine untere Schranke (falls minimiert wird)
    - ◆ setze die untere Schranke in Beziehung zu der errechneten Lösung.
  - Gibt es Verbesserungsmöglichkeiten von APPROX-VERTEX-COVER?
    - ◆ APPROX-VERTEX-COVER hat einen nicht aufgelösten Freiheitsgrad:
      - Wahl der Kante, die zu A hinzugenommen wird
    - ◆ mögliche Heuristiken zur Verbesserung
      - ◆ wähle Kante, dessen inzidente Knoten einen hohen Grad haben
  - Nicht jeder Algorithmus ist ein guter Approximationsalgorithmus
    - ◆ APPROX-VERTEX-COVER-2 ist **kein** 2-Approximationsalgorithmus!



## Ein Approximations-Algorithmus für Triangle-TSP

- Problem (Triangle-TSP):
  - geg. vollständiger Graph  $G=(V,E)$ , mit Kantengewichtsfunktion  $w$ ,  $w$  erfüllt die Dreiecksungleichung
  - ges. kürzeste Rundtour durch alle Knoten  $v \in V$
- Dreiecksungleichung:
  - für alle Knoten  $u, v, r$  gilt:  $w(u,v) + w(v,r) \geq w(u,r)$



- Triangle-TSP ist wie TSP NP-schwer.



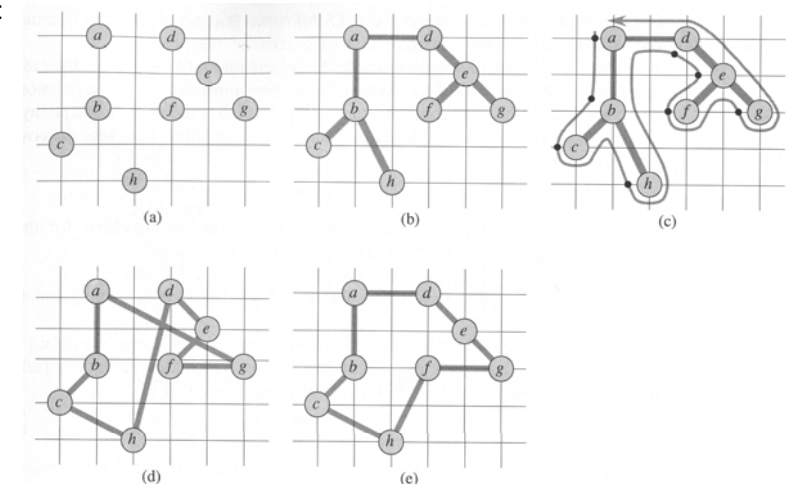
## Ein Algorithmus für Triangle-TSP

- APPROX-TSP-TOUR(  $G, c$  )
  - $T \leftarrow \text{MINIMUM-SPANNING-TREE}((V,E), c)$
  - $r \leftarrow \text{arbitrary } v \in V$
  - $H \leftarrow \text{PREORDER-TRAVERSAL}(T, r, \text{NIL})$
  - return**  $H$
- Schritt 1: berechne minimalen Spannbaum  $T$
- Schritt 2: durchlaufe den Baum und gib alle Knoten beim ersten Besuch aus (sei  $\text{TAdj}[v]$  die Adjazenzliste des MST):
  - $\text{PREORDER-TRAVERSAL}(T, v, \text{parent})$
  - output**  $v$
  - for each**  $w \in \text{TAdj}[v]$  **do**
  - if**  $w \neq \text{parent}$  **then**  $\text{PREORDER-TRAVERSAL}(T, w, v)$



## Triangle-TSP

- Bsp:



## Triangle-TSP

### Theorem 35.2

APPROX-TSP-TOUR ist eine Polynomzeit 2-Approximationsalgorithmus für Triangle-TSP.

#### Beweis Teil 1: APPROX-TSP-TOUR hat polynomielle Laufzeit

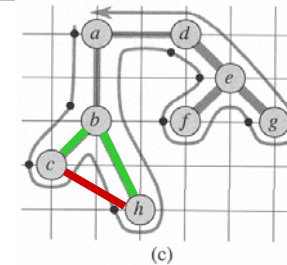
- ◆ Minimaler Spannbaum berechnen:  $O(N^2 \log N)$
- ◆ preorder-Durchlauf durch T:  $O(N)$
- ◆ Gesamtlaufzeit:  $O(N^2 \log N)$

#### Beweis Teil 2: $w(H)/w(H^*) \leq 2$

- ◆ H: Lösung von APPROX-TSP-TOUR,  $H^*$ : optimale Lösung
- ◆ Das Gewicht eines minimalen Spannbaums  $T^*$  stellt eine untere Schranke für die Länge einer TSP-Tour  $H^*$  dar:
  - ◆ Durch Löschen einer Kante  $e$  aus einer TSP-Tour  $H$  entsteht ein Spannbaum  $T$  mit  $w(H) = w(T) + w(e)$
  - ◆ Für die optimale TSP-Tour  $H^*$  gilt somit:  $w(H^*) \geq w(T^*)$



## Triangle-TSP



$$\begin{aligned} C: & a-b-c-b-h-b-a-d-e-f-e-g-e-d-a \\ C': & a-b-c-h-... \\ w(C') &= w(C) - [w(c,b) + w(b,h)] + w(c,h) \\ &\leq w(C) \end{aligned}$$

#### Beweis Teil 2 (Fortsetzung.):

- ◆ Sei  $C$  der Zyklus, der sich durch einmaliges Umlaufen von  $T$  ergibt.
- ◆  $C$  enthält jede Kante  $e$  aus  $T$  zwei mal, d.h.  $w(C) = 2 w(T)$
- ◆ Löschen eines Knotens  $w$  aus Zyklus  $C$  verkürzt die Länge des neuen Zyklus  $C'$  aufgrund der Dreiecksungleichung:
- ◆  $H$  entsteht aus  $C$  durch Löschen aller doppelt auftretenden Knoten, also gilt  $w(H) \leq w(C) = 2 w(T) \leq 2 w(H^*)$



## Das allgemeine TSP-Problem

### Theorem 35.3 (Approximierbarkeit von TSP):

- ◆ Falls  $P \neq NP$  gilt für jedes  $\rho > 1$ , dass für TSP kein Polynomzeit Approximations-Algorithmus mit Approximationsfaktor  $\rho$  existiert.

#### Beweis:

- ◆ Idee: nutze  $\rho$ -Approx.-Alg. A zur Lösung eines NP-vollständigen Problems (HAM-CYCLE). O.B.d.A. gilt:  $\rho$  ist ganzzahlig.

#### Polynomzeitreduktion:

Sei  $G=(V,E)$  Eingabe zu HAM-CYCLE. Konstruiere Eingabe für TSP:

$$G'=(V,E'), E'=\{(u,v) : u,v \in V\},$$

$$c(u,v) = \begin{cases} 1 & (u,v) \in E \\ \rho |V| + 1 & \text{sonst} \end{cases}$$

- ◆  $G \in \text{HAM-CYCLE} \Rightarrow G'$  hat ein Tour der Länge  $|V|$
- ◆  $G \notin \text{HAM-CYCLE} \Rightarrow$  alle Touren in  $G'$  haben Länge  $> \rho |V|$   
 $\Rightarrow A$  löst das HAM-CYCLE Problem in polynomieller Zeit.



## Kommentare zu Approximationsalgorithmen

- ◆ Approximationsalgorithmen ermöglichen die Lösung NP-schwerer Optimierungsprobleme mit einer **Gütegarantie** in polynomieller Zeit.
- ◆ Zum Nachweis des **Approximationsfaktors** ist es notwendig, eine Schranke für die optimale Lösung zu finden und diese mit der berechneten Lösung in Beziehung zu setzen.
- ◆ Es gibt Probleme, die nachweislich nicht mit einem konstanten Faktor in polynomieller Zeit approximierbar sind (unter der Annahme, dass  $P \neq NP$ )
- ◆ **Nicht-Approximierbarkeit** kann ebenfalls durch Reduktion nachgewiesen werden (NP-Vollständigkeit des  $k$ -Threshold-Problems mit Approximationsfaktor)
- ◆ Zusätzliche **Randbedingungen** (wie z.B. die Dreiecksungleichung) können das Problem deutlich vereinfachen, so dass sie in Polynomzeit approximierbar oder sogar lösbar werden.



## 8.2 Exakte Verfahren

- Vorgehen zur exakten Lösung
  - Bei vielen Problemen setzt sich die Lösung aus einer Menge von kleinen Teilentscheidungen zusammen.
  - Bsp:
    - ◆ CLIQUE: Gehört ein Knoten zu einer k-CLIQUE oder nicht?
    - ◆ TSP: Welchen Knoten besucht man nach Knoten v?
    - ◆ SUBSET-SUM: Gehört eine Zahl in die ausgewählte Teilmenge oder nicht?
  - Bei kombinatorischen Problemen ist es in der Regel einfach, die Menge der möglichen Lösungen aufzuzählen.
    - ◆ Selektionsprobleme: Bestimmung einer Teilmenge von Objekten
    - ◆ Zuordnungs-/Reihenfolgeprobleme: Bestimmung einer Permutation von Objekten
  - Lösungsansatz
    - ◆ Einfach: Enumeriere alle möglichen Lösungen und bestimme das Optimum.
    - ◆ Besser: Enumeriere alle möglichen Lösungen, die noch besser sein können, als das bisher gefundene Optimum.



## Enumeratoren

- Enumeration von Selektionen
  - Auswahl einer Teilmenge T über eine endliche Grundmenge G kann durch einen Binärstring der Länge |G| beschrieben werden
    - ◆  $b_i = 0$  : i-tes Element der Grundmenge ist nicht in T enthalten
    - ◆  $b_i = 1$  : i-tes Element der Grundmenge ist in T enthalten
  - Mögliche Implementierung (Cormen Kap. 17.1)  
Sei  $B[0..length[B]-1]$  ein Bit-Array mit  $length[B]$  Elementen  
BINARY-INCREMENT( B )  
 $i \leftarrow 0$   
**while**  $i < length[B]$  **and**  $B[i] = 1$  **do**  
     $B[i] \leftarrow 0; i \leftarrow i+1$   
**if**  $i < length[B]$  **then**  $B[i+1] \leftarrow 1$
  - BINARY-INCREMENT berechnet die nächste Auswahl in einer n- elementigen Menge (aufsteigender Binärwert)



## Enumeratoren

- Enumeration von Permutationen
    - Zuordnungen und Reihenfolgen können über Permutationen beschrieben werden
    - Beispiel:
      - ◆ TSP: Lösungsraum ist Permutation der Knoten des Graphen
      - ◆ QUADR. ASSIGNMENT: Lösungsraum ist die Permutation p der Objekte ( $p[i] = j$  bedeutet: Objekt i wird in Slot j platziert)
- ALL-PERMUTATIONS( P, i )  
// Sei  $P[1..length[P]]$  ein Array mit Zahlen  $1, \dots, length[P]$   
**if**  $i = 1$  **then** DO-SOMETHING(P)  
**else**  
    ALL-PERMUTATIONS(P, i-1)  
    **for**  $j \leftarrow 1$  **to**  $i-1$  **do**  
        SWAP(  $P[j], P[i]$  )  
        ALL-PERMUTATIONS(P, i-1)  
        SWAP(  $P[j], P[i]$  )



## Enumeratoren

- Iterative Implementierung
    - Berechne jeweils die nächst größere Permutation in lexikographischer Reihenfolge  
Sei  $P[1..length[P]]$  ein Array mit Zahlen  $1, \dots, length[P]$
- PERM-INCREMENT( P )  
 $t \leftarrow 1$   
**while**  $t < length[P]$  **and**  $P[t+1] > P[t]$  **do**  $t \leftarrow t+1$   
**if**  $t < length[P]$  **then**  
     $s \leftarrow t$   
    **while**  $s > 1$  **and**  $P[t+1] < P[s-1]$  **do**  $s \leftarrow s-1$   
    SWAP(  $P[t+1], P[s]$  )  
    **for**  $i \leftarrow 1$  **to**  $\lfloor t/2 \rfloor$  **do** SWAP(  $P[i], P[t-i+1]$  )



## Enumeratoren

### ■ Bsp: BINARY-INCREMENT

i: 9 8 7 6 5 4 3 2 1 0  
B: 0 1 1 0 0 1 1 1 1 1

BINARY-INCREMENT:

B: 0 1 1 0 1 0 0 0 0 0

BINARY-INCREMENT:

B: 0 1 1 0 1 0 0 0 0 1

BINARY-INCREMENT:

B: 0 1 1 0 1 0 0 0 1 0

suche die erste Stelle mit einer 0,  
kehre alle Werte bis zu dieser  
Stelle um.

### ■ Bsp: PERM-INCREMENT

i: 9 8 7 6 5 4 3 2 1  
P: 6 3 2 7 9 8 5 4 1

↑ t ↑ s  
PERM-INCREMENT:

P: 6 3 2 8 1 4 5 7 9

PERM-INCREMENT:

P: 6 3 2 8 1 4 5 9 7

PERM-INCREMENT:

P: 6 3 2 8 1 4 7 5 9

t: letzte Position der aufstei-genden  
Folge

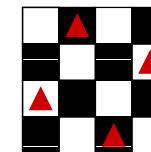
s: kleinster Wert der aufsteigen-den  
Folge, der größer P[t+1] ist.



## Das n-Damen-Problem

### ■ n-Damen-Problem:

Platziere n Damen auf einem n x n Schachbrett,  
so dass sie sich gegenseitig nicht schlagen  
können (d.h. unterschiedliche Spalten, Zeilen  
und Diagonalen)



Lösung durch Enumeration:

### ■ Variante 1: Enumeration über Schachbrettpositionen

N-QUEENS-1( n )

for i ← 0 to n<sup>2</sup>-1 do B[i] ← 0

do

if CHECK-N-QUEENS(B) then output B

BINARY-INCREMENT(B)

while B ≠ [0, 0, ..., 0]

n=8:  $2^{64} \approx 18.446.744.000.000.000.000$  Positionen



## Das n-Damen-Problem

### ■ Variante 2: Enumeration über Damenpositionen

Nutze die Eigenschaft, dass genau 8 Damen platziert werden  
müssen.

N-QUEENS-2( P, n, d )

// platziert die d-te Dame auf das Schachbrett Position 0, ..., n<sup>2</sup>-1

// P[i] speichert die Position der i-ten Dame, i = 1, ..., n

for i ← 1 to n<sup>2</sup>-1 do

P[d] ← i

if d < n then N-QUEENS-2(P, n, d+1)

else if CHECK-N-QUEENS(P) then output P

n=8:  $64^8 = 2^{48} \approx 281.474.497.000.000$  Positionen



## Das n-Damen-Problem

### ■ Variante 3: Ordne die Damen,

so dass P[i] < P[j] für alle i < j gilt

N-QUEENS-3( P, n, d )

// platziert die d-te Dame auf das Schachbrett Position 0, ..., n<sup>2</sup>-1

// P[i] speichert die Position der i-ten Dame, i = 1, ..., n; P[0] = -1

for i ← P[d-1]+1 to n<sup>2</sup>-1 do

P[d] ← i

if d < n then N-QUEENS-3(P, n, d+1)

else if CHECK-N-QUEENS(P) then output P

n=8:  $\binom{64}{8} = 64 \cdot 63 \cdot \dots \cdot 57 / (8!) = 4.426.165.368$





## Das n-Damen-Problem

- Variante 4: Nutze die Eigenschaft, dass zwei Damen nicht in der gleichen Zeile stehen können

Sei  $P[i]$  nun die Spalte, in der die Dame der  $i$ -ten Zeile steht

$N\text{-QUEENS-4}(P, n, d)$

// platziert die  $d$ -te Dame auf das Schachbrett in Zeile  $d$ , Spalte  $P[d]$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$P[d] \leftarrow i$

**if**  $d < n$  **then**  $N\text{-QUEENS-4}(P, n, d+1)$

**else if**  $CHECK\text{-}N\text{-QUEENS}(P)$  **then output**  $P$

$n=8$ :  $8^8 = 2^{24} = 16.777.216$



## Das n-Damen-Problem

- Variante 5: Nutze die Eigenschaft, dass zwei Damen weder in der gleichen Zeile, noch in der gleichen Spalte stehen können.

Da in jeder Spalte genau eine Dame stehen muss, können wir die Platzierung als Permutation der Spalten  $1, \dots, 8$  interpretieren.

Sei  $P[i]$  nun die Spalte, in der die Dame der  $i$ -ten Zeile steht

$N\text{-QUEENS-5}(P, n, d)$

// platziert die  $d$ -te Dame auf das Schachbrett in Zeile  $d$ , Spalte  $P[d]$

**for**  $i \leftarrow 1$  **to**  $n$  **do**  $P[i] \leftarrow i$

**do**

**if**  $CHECK\text{-}N\text{-QUEENS}(P)$  **then output**  $P$

$PERM\text{-}INCREMENT(P)$

**while**  $P \neq [1, 2, \dots, n]$

$n=8$ :  $8! = 40.320$



## Backtracking

- Schwäche aller bisherigen Lösungsversuche:  
 $CHECK\text{-}N\text{-QUEENS}()$  wird erst aufgerufen, nachdem ALLE Damen platziert wurden.
- Testen und Wiederverwenden von Teillösungen:
  - Zur Auswertung von Lösung werden Rechenschritte über gemeinsame Teillösungen wiederholt ausgeführt.
    - ◆ Bsp.: Auswertung beim  $n$ -Damen-Problem  
 $CHECK\text{-}N\text{-QUEENS}()$  überprüft für alle Paare von Damen, ob sie sich schlagen können.  
Seien  $P$  und  $P'$  zwei Platzierungen von Damen, mit  $P[i] = P'[i]$  für alle  $i < d$ :  
Falls für  $i < j < d$  gilt: Dame  $P[i]$  schlägt Dame  $P[j]$  nicht, so gilt dies für  $P$  und für  $P'$ . Der Test muss nur ein mal ausgeführt werden.  
Falls für  $i < j < d$  gilt: Dame  $P[i]$  schlägt Dame  $P[j]$ : so kann weder  $P$  noch  $P'$  zu einer gültigen Lösung erweitert werden.
- Wiederholte Auswertung von Teillösungen kann vermieden werden, wenn die Lösungen sukzessive aufgebaut werden.



## Backtracking

- Betrachte Lösungsraum als ein Baum (Suchbaum):
  - Wurzel: leere Teillösung
  - innere Knoten: Teillösungen
  - Blätter: Lösungen
  - Auf jeder Ebene wird eine Teilentscheidung gefällt und damit die Teillösung des Parent-Knotens erweitert.
  - Die Child-Knoten repräsentieren die alternativen Möglichkeiten respektive der Teilentscheidung
- Backtracking-Algorithmus:
  - systematische Tiefensuche in einer (den Lösungsraum repräsentierenden) Baumstruktur nach der optimalen Lösung
  - Teillösungen, die sich nicht zu vollständigen Lösungen erweitern lassen, müssen nicht weiter betrachtet werden.
  - Die Baumstruktur existiert nur implizit.





## Backtracking-Algorithmus für OPT-CLIQUE

```

CLIQUE-BACKTRACK(G, I, d, M)
  if d = |V| then % Clique gefunden
    if ||I| > |M| then M ← I
  else
    vd ← select d-th node from V[G]
    // Fall 1: Knoten vd ∈ I
    is_clique ← TRUE % prüfe, ob I ∪ {vd} eine Clique ist
    for each u ∈ I do (**)
      if u ∉ Adj[vd] then is_clique ← FALSE
    if is_clique then CLIQUE-BACKTRACK(G, I ∪ {vd}, d+1, M)
    // Fall 2: Knoten vd ∉ I
    CLIQUE-BACKTRACK(G, I, d+1, M)

```

Aufruf: G: Eingabe-Graph,  
 I=∅ (bereits in der Teilmenge enthaltene Elemente),  
 d=1 (Tiefe im Baum)  
 M=∅ (maximale Clique)



## Backtracking-Algorithmus für CLIQUE

- Laufzeit:
  - Sei  $N = |V|$ , dann hat der Suchbaum  $2^{N+1}$  Knoten
  - Ein rekursiver Aufruf von CLIQUE-BACKTRACK kann in  $O(n)$  Zeit bearbeitet werden  
 (\*\*) wird durch Vergleich sortierter Listen realisiert
  - Worst-Case Laufzeit:  $O(N \cdot 2^N)$   
 (z.B. falls G ein vollständiger Graph ist)
  - Tatsächliche Laufzeit hängt stark von der Kantendichte des Graphs ab
- Kommentare:
  - Der Algorithmus kann - leicht modifiziert - zum Aufzählen aller Cliques, bzw. aller maximalen Cliques verwendet werden
  - Der Algorithmus ist – je nach Fragestellung - bereits sehr effizient in der Praxis
  - besser: Bron-Kerbosch-Algorithmus (Chemieinformatik)



## Kommentare zu Enumeratoren und Backtracking

- Je genauer die Randbedingungen an die Lösung für die Enumeration modelliert werden, umso weniger Varianten müssen enumeriert werden: N-QUEENS (für  $N=8$ , 0,1 µsec/CHECK-N-QUEENS-Aufruf)

Algorithmus	# Varianten	Zeit
1. Schachbrett	>18.446.744.000.000.000.000	58,5 Mio y
2. Damen-Positionen	>281.474.497.000.000	892 y
3. Reihenfolge-Unabhängigkeit	4.426.165.368	123 h
4. Zeilen-Eigenschaft	16.777.216	28 min
5. Spalten-Eigenschaft	40.320	4,0 sec
6. Teillösungen (Backtracking)	2.057	0,2 sec

- Randbedingungen, die in der Enumeration modelliert sind, müssen nicht explizit geprüft werden.
- Durch den hierarchischen Aufbau von Lösungen können Teillösungen bereits auf Gültigkeit geprüft werden → Backtracking
- Enumeration eignet sich prinzipiell nur für diskrete (d.h. ganzzahlig modellierbare) Probleme



## Branch & Bound

- Wie lässt sich die Laufzeit weiter verkürzen?
  - Reduktion der Anzahl der besuchten Knoten im Suchbaum
  - Pruning: Abschneiden von Teilbäumen, die nicht zu einer optimalen Lösung führen können
    - ◆ alle Lösungen in diesem Teilbaum sind ungültig
    - ◆ alle Lösungen in diesem Teilbaum sind garantiert schlechter als die optimale Lösung
- zu 1.: prüfe Gültigkeit der Lösungen
  - CLIQUE-BACKTRACK betrachtet keine Teilmengen, die keinen vollständigen Teilgraph beschreiben.
- zu 2.: berechne Schranken für die Güte der Lösungen
  - Annahme:
    - ◆ wir maximieren die Zielfunktion f, sei C\* die optimale Lösung
  - untere Schranke L für C\*:
    - ◆ eine gültige Lösung C stellt eine untere Schranke für C\* dar:  $f(C^*) \geq f(C) = L$



## Branch & Bound

- obere Schranke  $U(v)$ :
  - ◆ Sei  $v$  ein Knoten im Suchbaum
  - ◆ Alle Lösungen im Teilbaum unter  $v$  haben eine Teillösung  $I$  gemeinsam
  - ◆ Berechne, wie gut eine Lösung, die  $I$  enthält maximal werden kann
- Vorgehen Branch & Bound:
  - Suche eine gültige Lösung  $C$ , setze  $L \leftarrow f(C)$
  - Führe ein Backtracking durch, für jeden besuchten Knoten  $v$ :
    - ◆ Falls  $v$  eine gültige Lösung  $C'$  mit  $f(C') > L$  repräsentiert, setze  $L \leftarrow f(C')$
    - ◆ Berechne obere Schranke  $U(v)$
    - ◆ Falls  $U(v) \leq L$  (d.h. alle Lösungen im Teilbaum unter  $v$  sind garantiert schlechter als die bisher gefundene beste Lösung)
      - ◆ Rückkehr zum Parent-Knoten
    - ◆ Falls  $U(v) > L$  (d.h. es könnten Lösungen im Teilbaum unter  $v$  sein, die besser sind als die bisher gefundene beste Lösung)
      - ◆ Erweitere Teillösung, rekursiver Aufruf

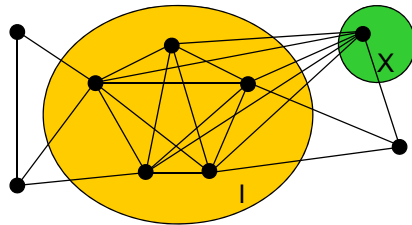


## Branch & Bound-Algorithmus für CLIQUE

- untere Schranke  $L$ :
  - max. Anzahl der Knoten in allen bisher gefundenen Cliques
  - Teilmengen sind bereits Cliques und können zur Verbesserung von  $L$  herangezogen werden
- obere Schranke an Knoten  $v$  auf Ebene  $d$  mit Teillösung  $I$   $U(d, I)$ :
  - alle Cliques unter  $v$  enthalten die Knoten aus  $I$
  - die Knotenmenge aus  $I$  kann maximal um  $(n-d)$  Knoten erweitert werden.
    - $\Rightarrow U(d, I) = |I| + n-d$
  - Verbesserung: Bestimme Knotenmenge  $X$  („extension set“) von Knoten, die nicht in  $I$  sind, aber zu allen Knoten aus  $I$  adjazent sind.
    - $\Rightarrow U(I, X) = |I| + |X|$



## Branch & Bound-Algorithmus für CLIQUE



CLIQUE-BB( $G, I, X, d, M$ )

$G$ : Eingabegraph

$I$ : Knotenmenge; aktuell betrachtete Teillösung

$X$ : Knotenmenge; Menge der Knoten, um die die aktuelle Teillösung erweitert werden kann (Extensionsmenge)

$d$ : Anzahl bereits betrachteter Knoten (Rekursionstiefe)

$M$ : Knotenmenge; maximale Clique, die gefunden wurde

Erster Aufruf: CLIQUE-BB( $G, \emptyset, V, 0, M$ )



## Branch & Bound-Algorithmus für CLIQUE

CLIQUE-BB( $G, I, X, d, M$ )

if  $d = |V[G]|$  then % Clique gefunden

if  $|I| > |M|$  then  $M \leftarrow I$

else

$v_d \leftarrow$  select  $d$ -th node from  $V[G]$

// obere Schranke  $U = |I| + |X|$ , untere Schranke ist  $|M|$

if  $|I| + |X| < |M|$  then return // BOUND!

// Fall 1: Knoten  $v_d \in I$  BRANCH!

//  $I \cup \{v_d\}$  ist eine Clique, g.d.w.  $v_d \in X$

if  $v_d \in X$  then CLIQUE-BB( $G, I \cup \{v_d\}, X \cap \text{Adj}[v_d], d+1, M$ )

// Fall 2: Knoten  $v_d \notin I$

CLIQUE-BB( $G, I, X \setminus \{v_d\}, d+1, M$ )



## Branch & Bound für das Assignment-Problem

### ■ Assignment-Problem:

■ geg.: Anzahl  $n$  von Agenten und Jobs, Kostenmatrix  $[c_{ij}]_{1 \leq i, j \leq n}$  mit  $c_{ij}$  = Kosten der Ausführung von Job  $j$  durch Agent  $i$ ,  $c_{ij} > 0$

■ ges.: Zuordnung  $p: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  der Agenten zu den Jobs, so dass:

- ♦ jeder Job durch einen Agenten bearbeitet wird
- ♦ minimalen Gesamtkosten  $C(p)$  entstehen

$$C(p) = \sum_{i=1}^n c_{ip(i)}$$

Beispiel:

el:		Jobs			
	$[c_{ij}]$	1	2	3	4
Agenten	a	11	12	18	40
	b	14	15	13	22
	c	11	17	19	23
	d	17	14	20	28

$$C(\text{yellow circle}) = 69$$



## Branch & Bound für das Assignment-Problem

### ■ Beschreibung des Lösungsraums:

- ♦ Abbildung der Agenten auf die Jobs  $f: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$   
 $f(i) = j$ : Agent  $i$  erfüllt Job  $j$
- ♦ Randbedingung: Abbildung  $f$  muss bijektiv sein

### ■ Obere Schranke für die optimale Lösung $C^*$ :

■ Wähle  $f(i) = i$  oder  $f(i) = n+1-i$

■ Untere Schranke für die optimale Lösung:  $C(f) = \sum_{i=1}^n c_{if(i)}$

■ Jeder Job muss bearbeitet werden:  $l_{\text{jobs}} = \sum_{j=1}^n \min_{1 \leq i \leq n} \{c_{ij}\}$

■ Jeder Agent muss einen Job bearbeiten:  $l_{\text{agents}} = \sum_{i=1}^n \min_{1 \leq j \leq n} \{c_{ij}\}$



## Branch & Bound für das Assignment-Problem

		Jobs				
		$[c_{ij}]$	1	2	3	4
Agenten	a		11	12	18	40
	b		14	15	13	22
	c		11	17	19	23
	d		17	14	20	28
		$[c_{ij}]$	1	2	3	4
Agenten	a		11	12	18	40
	b		14	15	13	22
	c		11	17	19	23
	d		17	14	20	28
		$[c_{ij}]$	1	2	3	4
Agenten	a		11	12	18	40
	b		14	15	13	22
	c		11	17	19	23
	d		17	14	20	28

$$C(f) = \sum_{i=1}^n c_{if(i)} = 73$$

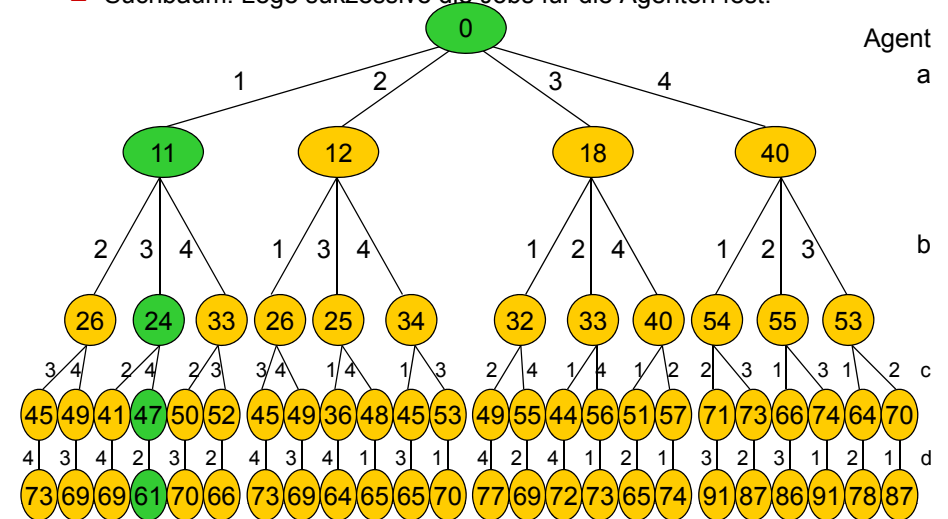
$$l_{\text{jobs}} = \sum_{j=1}^n \min_{1 \leq i \leq n} \{c_{ij}\} = 58$$

$$l_{\text{agents}} = \sum_{i=1}^n \min_{1 \leq j \leq n} \{c_{ij}\} = 49$$



## Branch & Bound für das Assignment-Problem

### ■ Suchbaum: Lege sukzessive die Jobs für die Agenten fest:



## Branch & Bound für das Assignment-Problem

- Berechnung der oberen / unteren Schranken im Algorithmus:
  - für einen Knoten v auf Ebene k gilt: Sei f die Zuordnungsfunktion

♦ die Zuordnung f der ersten k Agenten zu Jobs ist erfolgt  
 => Kosten für die ersten k Agenten können bereits berechnet werden, für die Agenten k+1, ..., n werden die Schranken verwendet:

=> obere Schranke: wähle eine beliebige Zuordnung für  $i > k$   
 Sei  $u: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  eine bijektive Funktion mit  $u(i) = f(i)$   
 für  $i \leq k$ , dann ist  $U = \sum_{i=1}^n c_{iu(i)}$  eine obere Schranke für  $C^*$

=> untere Schranke: wähle minimale Kosten für Agenten  $i > k$ :

$$\begin{aligned}
 L &= \sum_{i=1}^k c_{if(i)} + \sum_{i=k+1}^n \min_{j \in \{1, \dots, n\} \setminus \{f(h) \mid h \leq k\}} c_{ij} \\
 &= \sum_{i=1}^k u_{if(i)} + \sum_{i=k+1}^n \min_{j \in \{u(k+1), \dots, u(n)\}} c_{ij}
 \end{aligned}$$



## Branch & Bound für das Assignment-Problem

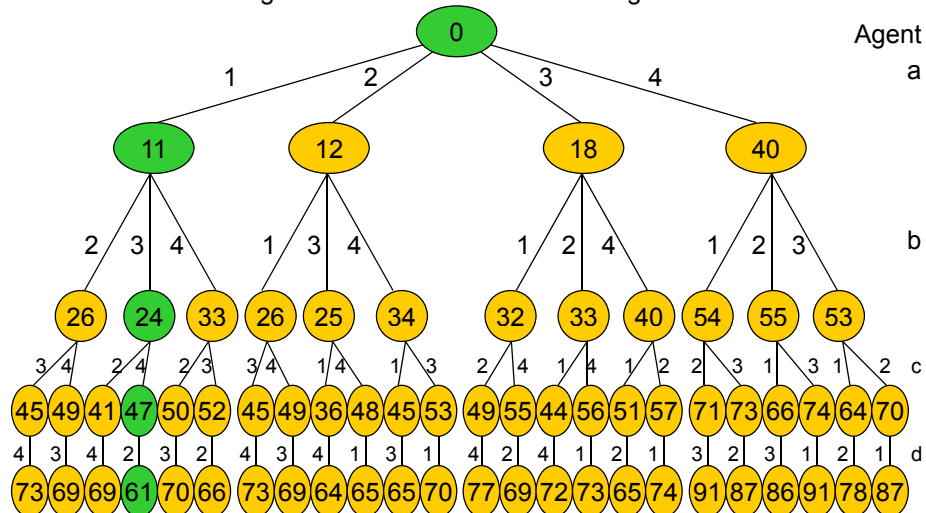
- ASSIGNMENT-UPPER-BOUND(c, f, n)
  - $u \leftarrow 0$
  - for**  $i \leftarrow 1$  **to**  $n$  **do**  $u \leftarrow u + c[i, f[i]]$
  - return**  $u$
- ASSIGNMENT-LOWER-BOUND(c, f, k, n)
  - $l \leftarrow 0$
  - for**  $i \leftarrow 1$  **to**  $n$  **do**
    - if**  $i \leq k$  **then**  $l \leftarrow l + c[i, f[i]]$
    - else**
      - $m \leftarrow c[i, f[k+1]]$
      - for**  $h \leftarrow k+2$  **to**  $n$  **do**
        - if**  $c[i, f[h]] < m$  **then**  $m \leftarrow c[i, f[h]]$
      - $l \leftarrow l + m$
  - return**  $l$

$c$ : Kostenmatrix  
 $n$ : Anzahl Agenten  
 $f$ : aktuelle Zuordnung  
 $k$ : Anzahl bereits fest zugeordneter Agenten



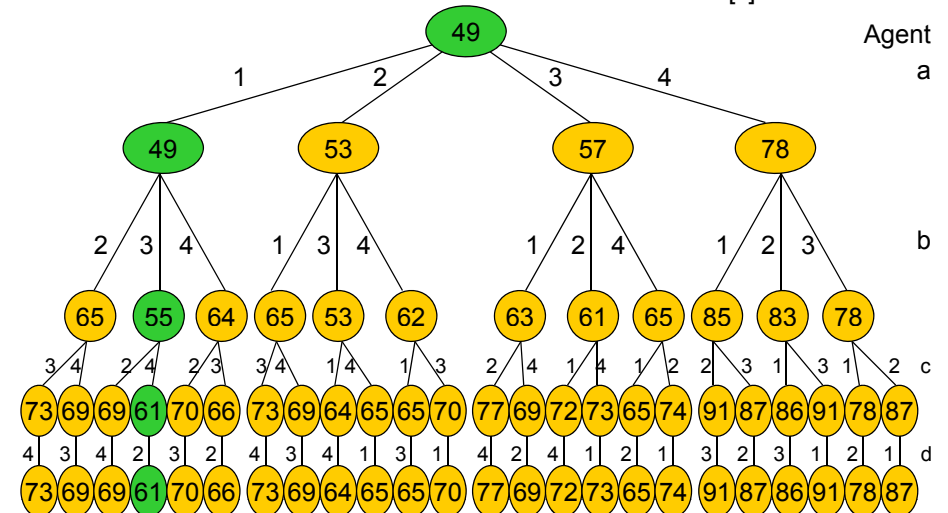
## Branch & Bound für das Assignment-Problem

- Suchbaum: Lege sukzessive die Jobs für die Agenten fest:



## Branch & Bound für das Assignment-Problem

- Suchbaum: Knoten v enthalten nun untere Schranke lb[v]:





## Branch & Bound für das Assignment-Problem

- Allgemeines Schema: Sei  $U$  die globale obere Schranke für  $C^*$ 
  - wähle einen Knoten  $v$  des Suchbaums:
    - $\text{level}[v]$  : Ebene des Suchbaums
    - $f[v]$  : Zuordnungsfunktion des Knotens
    - $\text{lb}[v]$  : Untere Schranke der Lösungen unter Knoten  $v$
  - falls  $\text{lb}[v] > U$  verwirfe die Lösung  $f[v]$
  - sonst teste alle möglichen Zuordnungen für Agent  $\text{level}[v]+1$ :
    - ◆ berechne und aktualisiere neue obere Schranke  $U$
    - ◆ erzeuge neue Knoten für die erweiterte Lösung
- Suchstrategien:
  - Depth-First (Tiefensuche): gehe rekursiv von  $\text{level}[v]$  zu allen Nachfolgern auf Ebene  $\text{level}[v]+1$
  - Breadth-First (Breitensuche): gehe Ebene für Ebene vor
  - **Best-First** (Kombinierte Tiefen- und Breitensuche): wähle den Knoten mit  $\text{lb}[v]$  minimal



## Branch & Bound für das Assignment-Problem

- Umsetzung der Best-First-Strategie:
  - Verwende Prioritätswarteschlange  $Q$  zur Speicherung der zur Auswertung bereitstehenden Knoten (Suchfront)
    - ◆ Prioritätskriterium: Untere Schranke  $\text{lb}[v]$
- ASSIGNMENT( $c, n$ )
  - $Q \leftarrow \text{INIT-MIN-HEAP}()$
  - // berechne Wurzel
  - for**  $i \leftarrow 1$  **to**  $n$  **do**  $f[v][i] \leftarrow i$
  - $\text{level}[v] \leftarrow 0$
  - $\text{lb}[v] \leftarrow \text{ASSIGNMENT-LOWER-BOUND}(c, f[v], \text{level}[v], n)$
  - // berechne initiale obere Schranke  $U$
  - $U \leftarrow \text{ASSIGNMENT-UPPER-BOUND}(c, f[v], n)$
  - INSERT-MIN-HEAP( $Q, v$ )
  - // Fortsetzung ...



## Branch & Bound für das Assignment-Problem

- Fortsetzung von ASSIGNMENT()

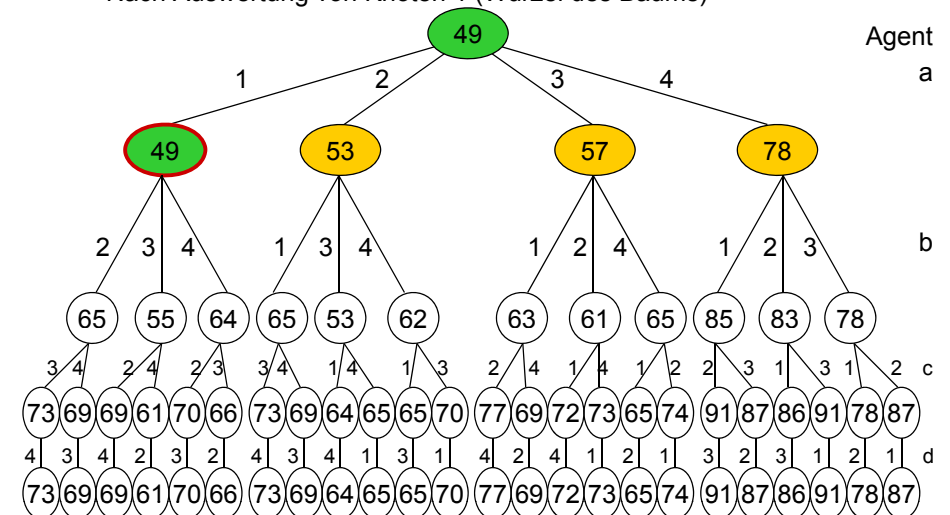
```

while  $Q \neq \emptyset$  do
   $v \leftarrow \text{EXTRACT-MIN-HEAP}(Q)$ 
  if  $\text{level}[v] < n-1$  and  $\text{lb}[v] < U$  then
    for  $i \leftarrow \text{level}[v]+1$  to  $n$  do
      // erzeuge neue Lösung w: vertausche in  $f$  Positionen  $\text{level}[v]$  und  $i$ 
       $w \leftarrow v$ 
       $\text{level}[w] \leftarrow \text{level}[v]+1$ 
      SWAP(  $f[w][\text{level}[w]], f[w][i]$  )
       $\text{lb}[w] \leftarrow \text{ASSIGNMENT-LOWER-BOUND}(c, f[w], \text{level}[w], n)$ 
      // berechne neue obere Schranke für  $C^*$ 
       $U \leftarrow \min( U, \text{ASSIGNMENT-UPPER-BOUND}(c, f[w], n) )$ 
      if  $\text{lb}[w] \leq U$  then INSERT-MIN-HEAP( $Q, w$ )
  return  $U$ 
  
```



## Branch & Bound für das Assignment-Problem

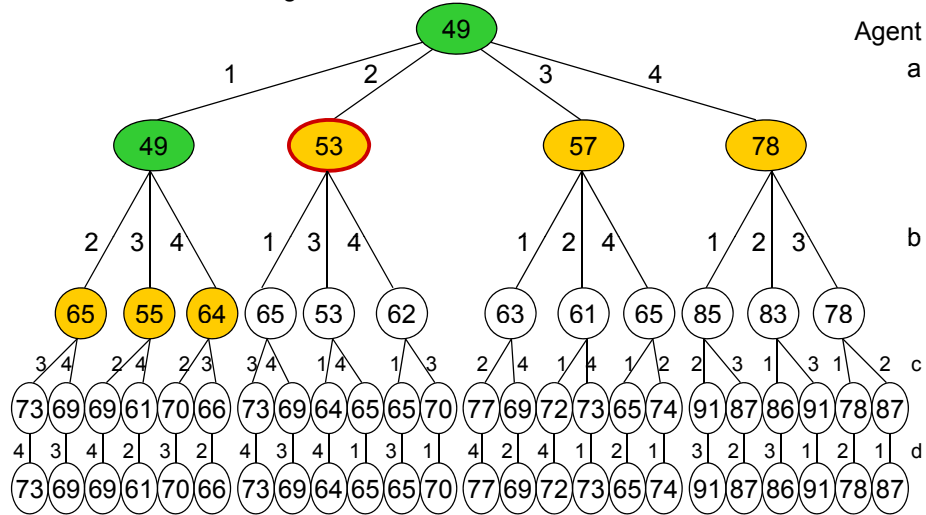
- Nach Auswertung von Knoten 1 (Wurzel des Baums)





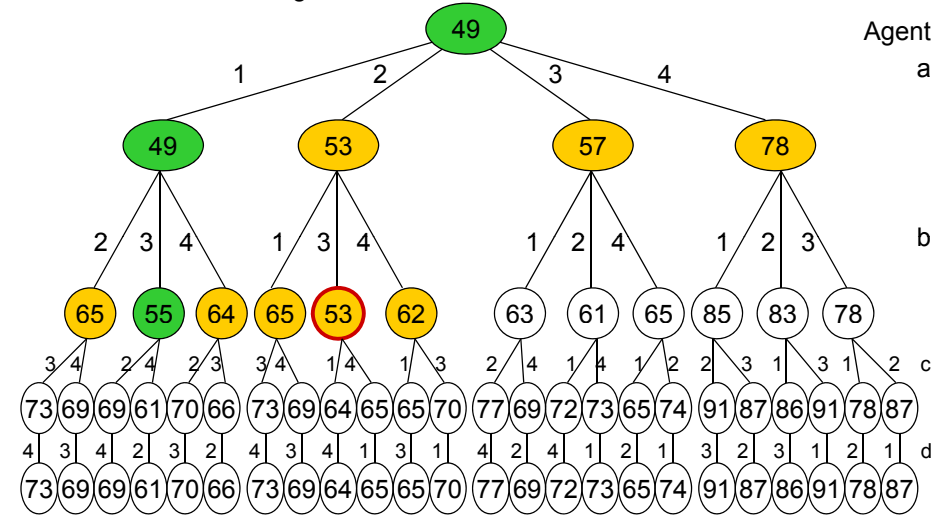
## Branch & Bound für das Assignment-Problem

### ■ Nach Auswertung von Knoten 2



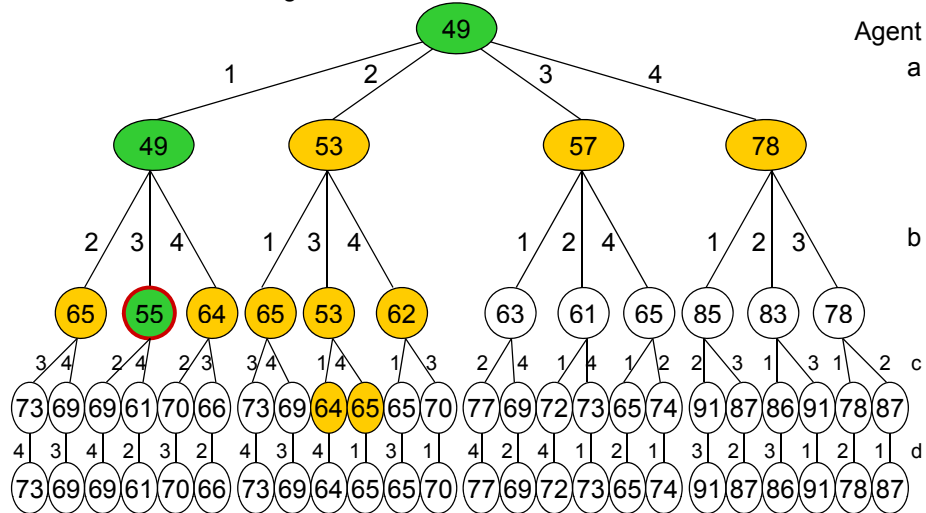
## Branch & Bound für das Assignment-Problem

### ■ Nach Auswertung von Knoten 3



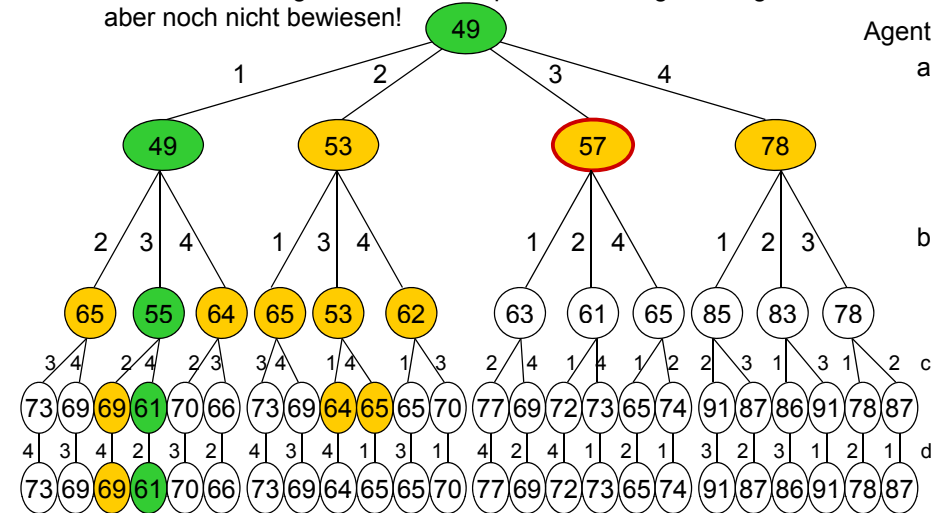
## Branch & Bound für das Assignment-Problem

### ■ Nach Auswertung von Knoten 4



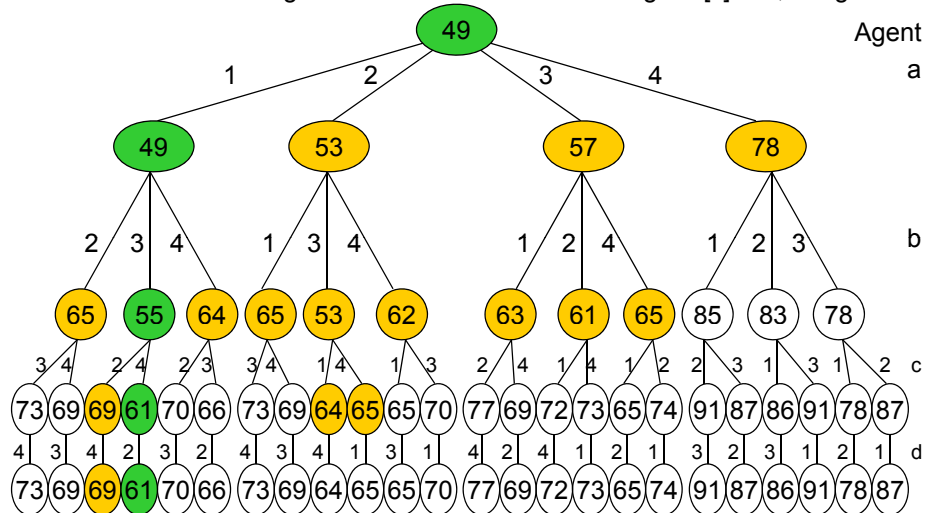
## Branch & Bound für das Assignment-Problem

### ■ Nach Auswertung von Knoten 5: optimale Lösung wurde gefunden, aber noch nicht bewiesen!



## Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 6: für alle Knoten gilt  $\text{lb}[v] \geq U$ , fertig!



## Kommentare zu Branch & Bound

- **Branch & Bound** gehört zu den besten bekannten Strategien für NP-schwere, kombinatorische Optimierungsprobleme.
- Die Qualität der Schranken hat einen großen Einfluss auf die Laufzeit des Algorithmus.
- Das **Finden der oberen Schranke** (bei einem Maximierungsproblem) ist der schwierigste Schritt bei der Entwicklung von Branch&Bound Algorithmen.
- Die **Reihenfolge**, in der Knoten besucht werden, hat einen Einfluss auf die Laufzeit des Algorithmus.
- Gängige **Suchstrategien** sind Depth-First und Best-First:
  - **Depth-First**: durchmustere den Suchbaum rekursiv in die Tiefe
    - ♦ Vorteil: speichereffizient    Nachteil: potentiell längere Laufzeit
  - **Best-First**: wähle vielversprechendsten Knoten aus der aktuellen Suchfront
    - ♦ Vorteil: potentiell kürzere Laufzeit    Nachteil: hoher Speicherbedarf
- Depth-First und Best-First sind zu einer Strategie, die sich je nach Speicher- und Laufzeitverbrauch adaptiert, kombinierbar.

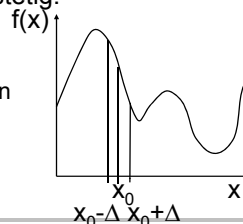
## 8.3 Heuristische Verfahren

### ■ Heuristische Verfahren:

- Algorithmen zur Lösung von Optimierungsproblemen ohne Gütegarantie
- Alternative Vorgehensweisen:
  - ♦ Generische Optimierungsstrategien, die sich auf viele verschiedene Probleme anwenden lassen
  - ♦ Implementierung einer intuitiven Vorgehensweise zur Optimierung des Problems

### ■ Lokale Suche

- Exploration des Suchraums durch schrittweises Verändern einer Lösung (Transition: Übergang zu einer benachbarten Lösung)
- Annahme: Optimierungsfunktion ist annähernd stetig.
- Bestandteile:
  - ♦ Funktion zur Generierung einer gültigen Lösung
  - ♦ Definition von ‚Nachbarschaft‘ zwischen Lösungen
    - ♦ geringfügige Änderung der Lösung
    - ♦ möglichst nur geringfügige Änderung des Optimierungsfunktionswertes



## Lokale Suche

### ■ Idee:

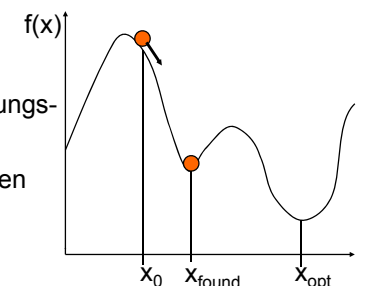
- bestimme eine gültige Lösung
- wiederhole bis keine Verbesserung mehr erzielt wird:
  - ♦ explore die Nachbarschaft der Lösung
  - ♦ wähle besten Nachbarn

### ■ Eigenschaften der Lokalen Suche:

- führt von einer geg. Lösung zum nächsten lokalen Optimum
- kann ein lokales Optimum nicht wieder verlassen

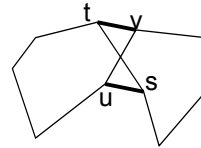
### ■ Bedeutung der ‚Anzahl der Nachbarn‘

- gering: kurze Laufzeit eines Optimierungsschritts
- hoch: geringe Chance, in einem lokalen Optimum stecken zu bleiben



## Lokale Suche für das TSP-Problem

- Nachbarschaft: *two-interchange move*
  - wähle zwei Kanten  $(u,v), (s,t)$  der Tour
  - ersetze sie durch Kanten  $(u,s), (v,t)$
  - Anzahl der Nachbarn:  $n(n-3)/2$



- Algorithmus

LOCAL-SEARCH-TSP(  $G, c$  )

//  $T$  stellt eine Tour dar, Generierung z.B. mit Approximation über

// minimalen Spannbaum

$opt \leftarrow \text{CREATE-TSP-TOUR}(G, c)$

**do**

$T \leftarrow opt$

**for each**  $(u,v), (s,t) \in T$  **do**

**if**  $c(opt) > c(T) - c(u,v) - c(s,t) + c(u,s) + c(v,t)$  **then**

$opt \leftarrow \text{TWO-INTERCHANGE-MOVE}(T, (u,v), (s,t))$

**while**  $T \neq opt$

**return**  $T$



## Simulated Annealing

- Wie kann man das „Stecken-bleiben“ in lokalen Minima vermeiden?
  - Akzeptanz auch von Verschlechterungen während der lokalen Suche
  - Verschlechterungen werden nur mit einer gewissen Wahrscheinlichkeit akzeptiert, die vom Ausmaß der Verschlechterung abhängt.
  - Die Wahrscheinlichkeit zur Annahme von Verschlechterungen sinkt über die Optimierungszeit.
- **Simulated Annealing**
  - Optimierungsschema in Anlehnung an den physikalischen Prozess der langsamen Abkühlung zur Erzeugung niederenergetischer Festkörper (Kristalle)
  - hohe Temperatur: Verschlechterungen werden mit hoher Wahrscheinlichkeit akzeptiert.
  - niedrige Temperatur: Verschlechterungen werden mit geringer Wahrscheinlichkeit akzeptiert.
  - Während der Optimierung verringert sich die Temperatur nach einem festen Abkühlungsschema (*Cooling Schedule*)



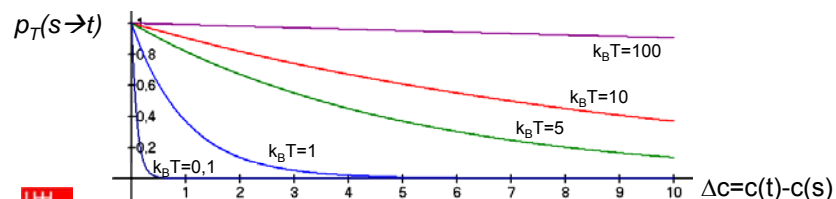
## Metropolis-Kriterium

- Wahrscheinlichkeitsverteilung der Akzeptanzfunktion  
 $p_T(s \rightarrow t)$  (angenommen, wir minimieren)
  - $p$  wird kleiner, je größer die Differenz von  $c(s)$  und  $c(t)$  ist
  - $p$  wird größer, je höher die Temperatur ist

$$p_T(s \rightarrow t) = e^{-\frac{c(t) - c(s)}{k_B T}}$$

$k_B$ : Boltzmann Konstante  
 $T$ : Temperatur  
 $c$ : Optimierungsfunktion

- Funktion ist gewählt in Anlehnung an die Thermodynamik
- in der Optimierung, dient  $k_B$  als Skalierungsfaktor, da die Optimierungsfunktion typischerweise keine Energie ist



## Simulated-Annealing Algorithmus

- Schritte des SA-Algorithmus (auch Metropolis-Algorithmus)
  - // berechne eine initiale, gültige Lösung  $A$
  - // wähle Anfangstemperatur  $T$  und Abkühlungsschema  $f(T, l)$
  - // Variablen: ( $T$ : aktuelle Temperatur,  $l$ : Zeitäquivalent)
  - $l \leftarrow 0$
  - while**  $T > \varepsilon$  **do**
    - $B \leftarrow \text{arbitrary select } B \in \text{NEIGHBORHOOD}(A)$
    - if**  $c(B) \leq c(A)$  **then**  $A \leftarrow B$
    - else**
      - $r \leftarrow \text{random number from } (0,1)$
      - if**  $r < p_T(A \rightarrow B)$  **then**  $A \leftarrow B$
  - $l \leftarrow l+1; T \leftarrow f(T, l);$
- return**  $A$



## Simulated Annealing

- Wahl der Systemparameter:
  - **Initiale Temperatur:** wird so gewählt, das möglichst alle Transitionen akzeptiert werden können
  - **Abkühlungsschema:** typisch d Zeitschritte konstant, danach Reduktion um einen Faktor  $0.8 \leq r \leq 0.99$ , d.h.  $f(T, I) = r^{\lfloor I/d \rfloor} T$
- Kommentare
  - SA konvergiert – bei richtiger Wahl von Nachbarschaft, Anfangstemperatur, Abbruchkriterium und Abkühlungsschema – statistisch gegen das globale Optimum.
  - Konvergenz ist nur logarithmisch, SA kann nicht dazu eingesetzt werden, eine optimale Lösung mit Gütegarantie zu erhalten
  - SA ist typischerweise sehr rechenintensiv
  - zur Wahl der Systemparameter sind viele Tests notwendig
  - typischerweise werden deutlich bessere Ergebnisse erzielt als mit einer einfachen lokalen Suche



## Genetische Algorithmen (GA): Motivation

- Heuristisches Optimierungsverfahren, inspiriert durch Optimierungsprozesse der Natur:
  - Darstellung von Lösungen (Individuen) durch Vektoren über einen endlichem Alphabet, z.B. Bitstrings (Gene)
  - Erzeugung von neuen Lösungen durch Reproduktion und Rekombination (cross-over) bekannter Lösungen
  - Bewertung und Auswahl von Lösungen durch Fitness-Funktion
  - Zufällige Modifizierung der generierten Lösungen (Mutation)
  - Verwerfen von Lösungen in Abhängigkeit von ihrem Fitnesswert (Selektion)
- Unterscheidende Eigenschaften von GA's im Vergleich zu Simulated Annealing:
  - Erzeugung einer Menge von Lösungen (Population)
  - Keine lokal beschränkte Suche (Rekombination)



## Beispiel: Genetischer Algorithmus für TSP

- Gesucht: Kürzester Weg über alle Städte 1, 2, ..., n
- 1. Ausgangspopulation  $t = 0$ :  $P_0 = \{\alpha_1, \alpha_2, \dots, \alpha_k\}$ ,  $k = 30$  Individuen mit zufälligen Permutationen der Zahlen 1 bis n
- 2. Berechnung der Fitness  $fitness(\alpha_i)$  für alle  $i = 1, 2, \dots, k$
- 3. Zufällige Auswahl von  $k/2$  Paaren  $(\beta^1_j, \beta^2_j)$ ,  $j = 1, 2, \dots, k/2$   
→ Wahrscheinlichkeitsverteilung  $P(\alpha_i)$  korreliert mit  $fitness(\alpha_i)$  !
  - 1. Erzeugung von zwei Nachkommen durch cross over-Rekombination jedes Paares  $(\beta^1_j, \beta^2_j)$ 
    - Wähle zufällig cross over-Punkt  $1 \leq c \leq n$
    - Übernehme Reihenfolge der ersten c Städte aus  $\beta^1_j$  und ordne die restlichen Städte in der Reihenfolge an, wie sie in  $\beta^2_j$  vorkommen (und umgekehrt)
  - 2. Einfügen der Nachkommen in Population  $P_{t+1}$
- 4. Zufällige Mutation jedes Individuums in  $P_{t+1}$ , z.B. durch Vertauschung benachbarter Zahlen
- 5. Wenn maximale Populationszahl  $t_{max}$  erreicht oder Fitness-Zielwert erreicht, dann stop, sonst, gehen zu Schritt 2



## Genetische Algorithmen: Schema-Theorem

- Schema: Gemeinsame Eigenschaften mehrerer Individuen, dargestellt durch ein gemeinsames, lokal begrenztes Bitmuster:

\*\*\*\*\*0110\*\*01001\*\*\*\*\*  
                    └──────────┘  
                    Schema der Länge 11

- Erklärung der Funktionsweise von GA's durch das Schema-Theorem:

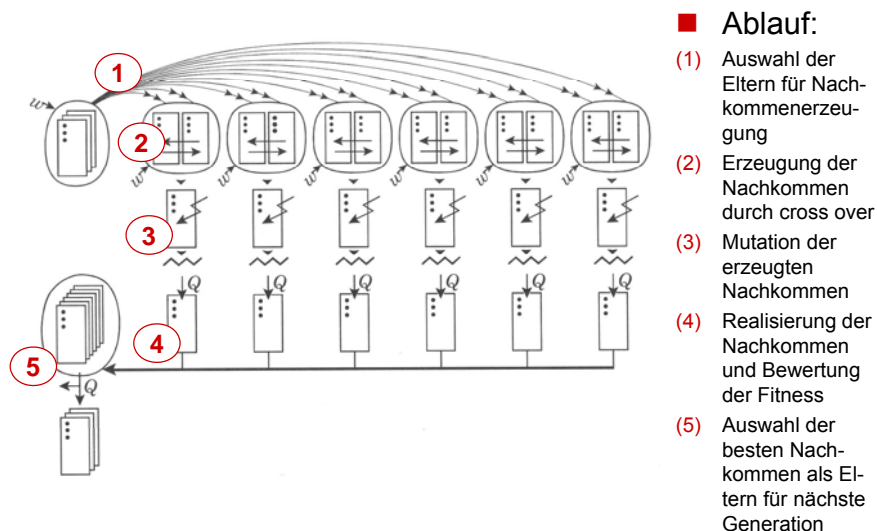
Kurze Schemata mit hohem Fitnesswert werden ihr Vorkommen in einer Population im Verlauf der Evolution steigern (building block-Hypothese)  
→ „Resistenz“ gegen cross-over Operation



- Vielseitige, heuristische Verfahren zur Lösung von Optimierungsproblemen
- Angelehnt an das Evolutionsprinzip der Natur; ähnliche Konzepte wie bei Genetischen Algorithmen:
  - Darstellung von Lösungen durch Individuen und Populationen
  - Variation der Lösungen z.B. durch Rekombination oder Mutation
  - Auswahl und Überleben der Lösungen nach ihrem Fitnesswert
- Unterschiede:
  - ES basieren auf metrisch skalierbaren Variablenwerten und einem stetigen Lösungsraum (Prinzip der starken Kausalität)
  - Philosophien:
    - ES: Phänotypischer Algorithmus (Imitation der Wirkung von genetischen Operationen)
    - GA: Genotypischer Algorithmus (Imitation der Funktion von genetischen Operationen)

- Einheitliche Beschreibung von Evolutionsstrategien durch  $(\mu/\rho \pm \lambda)$ -Notation:
  - $\mu$  Eltern erzeugen  $\lambda$  Nachkommen
  - Zur Erzeugung eines Nachkommen werden jeweils  $\rho$  der  $\mu$  Eltern ausgewählt und ihre Variablenausprägungen gemischt
  - Die Eltern der neuen Generation werden aus den Nachkommen ( $(\mu, \lambda)$ -ES) bzw. aus Eltern und Nachkommen ( $(\mu + \lambda)$ -ES) selektiert
- Erweiterung: ES mit Mutationsschrittweitenregelung (MSR)
  - Jedes Individuum  $i$  speichert neben der Ausprägung  $x_i^g$  einer Variable  $g$  auch die Schrittweite  $\delta_i^g$ , mit der diese Variable verändert werden kann

## Schema einer (3/2, 6)-gliedrigen Evolutionsstrategie



## Eigenschaften von ES und GA

- Die Qualität der gefundenen Lösungen wird u.U. stark durch die Initialisierung der Anfangspopulation und den verwendeten Zufallsgenerator beeinflusst (bedingte Reproduzierbarkeit)
- Richtige Wahl von Parametern erfordert in der Regel viele Tests
  - Darstellung/Codierung der Individuen
  - Populationsgröße
  - Nachkommenzahl
  - Mutationswahrscheinlichkeit und/oder -schrittweite
- GA erfordern eine diskrete Codierung der Lösungen, möglichst nach dem Prinzip der starken Kausalität
  - Problembeispiel: Inversion nur eines führenden Bits bei binärcodierten Dezimalzahlen bedeutet große Veränderung des Zahlenwertes (schwache Kausalität)

## Kommentare zu heuristischen Verfahren

- Heuristische Verfahren ermöglichen die Lösung einer Vielzahl verschiedener Optimierungsprobleme mit einem einheitlichen algorithmischen Schema.
- Heuristische Verfahren liefern keine Gütegarantie.
- Algorithmische Schemata gibt es viele:
  - Simulated Annealing
  - Genetische Algorithmen und Evolutionsstrategien
  - Tabu Search, Flooding, Particle Swarm Optimization, Ant Colony Optimization, etc.
- Heuristische Verfahren stellen das letzte Mittel dar: Sie sollten angewendet werden, wenn
  - das Problem schwer zu lösen ist
  - eine Approximation nicht bekannt ist
  - eine exakte Lösung zu zeitintensiv ist
  - eine kombinatorische Modellierung nicht möglich ist.



## Rückblick auf die Vorlesung

### 1. Einführung

- Beschreibung und
- Analyse von Algorithmen

### 2. Elementare DS

- Beschreibung und
- Analyse von DS

### 3. Sortieren

- Divide&Conquer
- Heaps
- stochastische Analyse
- Selektion / Median

### 8. Schwere Probleme

- Approximationsalgorithmen
- Exakte Verfahren (B & B)
- Heuristische Verfahren

## Algorithmen und Datenstrukturen

### 4. Suchen

- balancierte  
Suchbäume
- Hashing

### 7. NP-Vollständigkeit

- Komplexität von Problemen
- Zugehörigkeit zu NPC
- Reduktionsbeweise

### 6. Dyn. Programmierung

- Charakterisierung
- 4-Phasen Entwicklung
- Matrix-Kettenmultiplikation

### 5. Graphen

- Systematische Suche
- Spannbäume
- Kürzeste Wege



## Rückblick auf die Vorlesung

- **Lernziele** (aus dem Modulhandbuch)  
Vermittlung von Problemlösungskompetenz (Konzept und Realisierung) zur Lösung formalisierbarer, schwieriger Probleme
  - Selbstständiges, kreatives **Entwickeln** von Alg. und DS
  - **Korrektheitsbeweise** und **Effizienzanalyse**
  - Selbstständiges **Aneignen** neuer Alg. und DS
  - **Übertragung** bekannter Alg. auf neue Probleme
  - **Modifikation** bekannter Alg. auf veränderte Anforderungen
  - **Beurteilung** der Qualität von Alg.
  - Erkennen grundlegender **Beschränkungen** von Alg.
  - Einschätzung von Problemen in Hinblick auf ihre **Komplexität**



## Weiterführendes Modul **Algorithmik** (9LP, ab WiSe 07/08)

- **Analysemethoden:** Amortisierte Analyse
- **Fortgeschrittene Datenstrukturen:** Binomial / Fibonacci-Heaps
- **Weiterführende Graphalgorithmen:** All-Pairs und algebraische kürzeste Wege, Netzwerkfluss-Algorithmen, Matching, ...
- **Algorithmische Geometrie:** Schnittpunkte, A&D zu Raumanfragen, Konvexe Hüllen, Voronoi-Diagramme und Delauney-Triangulierung, Umschließende Kreise, ...
- **Lineare Programmierung**
- **NPC:** Reduktionsbeweise, Approximationsalgorithmen und polynomielle Approximationsschemata, Branch&Bound über Relaxation
- ...

