

# Algorithmen

Michael Kaufmann (Leitung)  
Henry Förster, Axel Kuckuk (Übungsorganisation)  
viele Tutoren ...

03/11/2020

# Organisatorisches

**Vorlesungstermine:** Di 15.30-16.30 und Mi 14.00 -15.00

**Besprechung der Übungsaufgaben:** Mo 16.30-18.00

**Help Desk:** Online vor der Dienstag-Vorlesung 12.30 - 14:30

**Klausur:** Mi 03.03.2021, 15-17 Uhr

**Nachklausur:** letzte Woche vor SoSe Beginn

Teilnahme nur an Nachklausur möglich (dann keine 2. Chance!)

**Übungsblätter:** 50 Prozent der Punkte gefordert, Boni für Klausur

Anmeldung für Tutorien (am besten heute !!!)

**Organisation:** <https://moodle.zdv.uni-tuebingen.de/>

vorbereitende Präsenzübungen im Tutorium

Start der Übungen: 2. Woche. Blatt 1 Donnerstag !

# Gliederung in Themen

## **I. Einführung**

- Motivation und Notationen
- Laufzeitanalyse, Rekursionen

## **II. Grundlegende Datenstrukturen**

## **III. Graphenalgorithmen**

## **IV. Sortieren**

## **V. Suchen**

## **VI. Generische algorithmische Methoden**

## **VII. Algorithmen auf Zeichenketten**

# Algorithmen kommen überall vor !

- Zugverkehr:
  - Rechne einen Zugfahrplan aus
  - Weise jeder Zuglinie eine physische Lok + Waggon zu
  - Berücksichtige Zugausfälle/Verspätungen, um neue Empfehlungen zu geben
- Börse
  - Löse schwierige kombinatorische Probleme
  - Zeitkritisch ! Gute Lösungen in Millisekunden !
- Biologie
  - Sequenziere ein Genom !
  - Analysiere und Vergleiche mit anderen !

# Algorithmen kommen überall vor !

- Tutorienverwaltung:
  - 300 Studis, werden verteilt auf 15 Gruppen
  - Jede hat zeitliche Präferenzen
  - Berechne Zuweisung, so dass jede zufrieden ist.
- Internet:
  - Netzwerkprotokolle, möglichst robust gegen Fehler
  - Routing Standards → Vorteile/Nachteile verschiedener kürzester Wege !
- Firmen wie Google:
  - Indiziere alle Webseiten !
  - Wie kann man Speicherkapazität sparen, ohne Redundanz aufzugeben ?
  - Beantworte Suchanfragen hoch qualitativ und schnell

# Informatik ist viel mehr als Programmieren ...

- Viele Leute können programmieren (oder denken es). Braucht kein Informatiker zu sein, um Java Code zu schreiben.
- Um praktische Probleme zu lösen, braucht es mehr als die Syntax einer PS. Man kann nicht einfach hinsitzen und Lösungen implementieren.
- Was ein Informatiker können sollte:
  - Gegen ein praktisches Problem, 'abstrahiere' von allen schmutzigen, anwendungsabhängigen Details.
  - Werkzeugkasten !
  - Analyse: Wie schwierig ist das Problem? Wie gut ist mein Ansatz? Geht es auch besser, und mit welchem Aufwand ?
  - Beschreibe deine Lösung genau genug, so dass ein Programmierer sie umsetzen kann.

# Warum ist das wichtig für Sie?

- Viele Informatiker laufen unweigerlich in algorithmische Probleme im späteren Beruf
- Sie sollten es wenigstens merken, wenn es passiert.
- Sie sollten wenigstens ein Grundverständnis haben, wie solche Probleme angegangen werden sollten
- Dieses Grundverständnis ist notwendig, um vernünftige Lösungen zu finden (sogar wenn Sie in der Literatur nachlesen)

# Literatur

- S. Dasgupta, Ch. Papadimitriou, U. Vazirani. Algorithms, McGraw-Hill, 2008.
- T.Cormen, Ch.Leiserson, R.Rivest, C.Stein: Introduction to algorithms and data structures. 3rd edition. MIT Press, 2009.
- J. Kleinberg, E. Tardos: Algorithm Design. 2005.
- K. Mehlhorn, P. Sanders, M. Dietzfelbinger: Algorithmen und Datenstrukturen. Springer, 2014.
- Außerdem: Vöcking et al.: Taschenbuch der Algorithmen. Springer, 2008. (Engl. Algorithms unplugged)
- Sowie: J. MacCormick: 9 Algorithms that changed the future, Princeton University Press, 2012.



## **Was sind Algorithmen und wie geht Algorithmenanalyse ?**

*Wikipedia:* Ein A. ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.

A's bestehen aus endlich vielen, wohldefinierten Einzelschritten. Damit können sie zur Ausführung in ein Computerprogramm implementiert, aber auch in menschlicher Sprache formuliert werden. Bei der Problemlösung wird eine bestimmte Eingabe in eine bestimmte Ausgabe überführt.

**Name:** – Buch von Al Khwarizmi (9. Jhd.) über Dezimalsystem.

- Genaue eindeutige Beschreibungen numerischer Operationen wie Addition, Dividieren, lineare Gleichungen
- latein. Übersetzung (12. Jhd): Algoritmi de numero Indorum

## Erstes Beispiel aus Dasgupta: Fibonacci Zahlen

$$F(n) = \left\{ \begin{array}{ll} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F(n-1) + F(n-2), & \text{falls } n > 2 \end{array} \right\}$$

Ergibt die Folge 0,1,1,2,3,5,8,13,21, ....

Zahlen wachsen sehr schnell. Für  $n \geq 6$  haben wir  
 $2^{n/2} \leq F(n) \leq 2^n$ .

Wie sehen potentielle Algorithmen aus ?

→ Schauen wir uns mal 2 an. Es sollte einfach sein, oder ? ...

rekursiv oder nicht-rekursiv ! Probieren Sie beides !

# Algorithmus 1: nicht-rekursive einfache Schleife

**Input:**  $n \in \mathbb{N}$

**Output:** Die  $n$ -te Fibonacci-Zahl  $F_n$

```
1 Algorithm: FibLoop( $n$ )
2 Allokieren  $F$  als ein Array der Länge  $n + 1$ ;
3  $F[0] \leftarrow 0$ ;
4  $F[1] \leftarrow 1$ ;
5 for  $i \in \{2, \dots, n\}$  do
6   |  $F[i] \leftarrow F[i - 1] + F[i - 2]$ ;
7 end
8 return  $F[n]$ 
```

## Algorithmus 2: Rekursion

**Input:**  $n \in \mathbb{N}$

**Output:** Die  $n$ -te Fibonacci-Zahl  $F_n$

```
1 Algorithm: FibRekursiv( $n$ )  
2 if  $n = 0$  then  
3   | return  $0$   
4 end  
5 if  $n = 1$  then  
6   | return  $1$   
7 end  
8 return  $\text{FibRekursiv}(n - 1) + \text{FibRekursiv}(n - 2)$ 
```

WELCHER ALGORITHMUS IST BESSER ?

WAS IST DAS RICHTIGE ARGUMENT ?

## Laufzeiten der Fibonacci Algorithmen: Zählen die 'Elementaroperationen'

FibLoop(n)

- Alloziere ein Array der Länge  $n$ , entweder 1 OP (Pointer auf den Anfang des Arrays, oder  $\leq n$  OPs (Initialisiere mit Nullen)
- 2 Elementaroperationen für 2 Zuweisungen.
- In jedem Schleifendurchlauf addieren wir 2 Zahlen.  
Da  $2^{n/2} \leq F(n) \leq 2^n$  für  $n \geq 6$ , ist  $F(n)$  etwa  $n$  Bits lang.
- Schleife wird  $(n - 1)$ - mal durchlaufen.

Anzahl der Elementaroperationen:  $1 + 2 + (n - 1) \cdot n \approx n^2$

## Laufzeiten der Fibonacci Algorithmen: Zählen die 'Elementaroperationen'

FibRekursive(n)

- Anfangs jeweils 1 Elementaroperation
- Dann eine Addition und die Zeit für die rekursiven Aufrufe.
- Also  $T(n) = 4 + T(\text{addition}) + T(n-1) + T(n-2) \geq 4 + T(n-1) + T(n-2)$
- Vergleich mit Definition von  $F_n$  zeigt, dass  $T(n) \geq F_n$ .
- Wegen  $F_n \geq 2^{n/2}$  für  $n \geq 6$ , wir erhalten  $T(n) \geq 2^{n/2}$ .

## Laufzeiten der Fibonacci Algorithmen: Vergleich

- Laufzeit von FibRekursive ist exponentiell in  $n$ . Also ist die Laufzeit sogar für mittelgroße  $n$  zu groß für alle denkbare Computer.
- Laufzeit von FibLoop ist quadratisch in  $n$ . Okay für relativ große  $n$ , aber nicht toll. Immerhin 'polynomiell'.
- Geht es besser als mit FibLoop ? Übungsblatt 1.

## Was lernen wir daraus ?

- Sogar für ganz einfache Probleme, bisschen unterschiedliche Ansätze können zu riesigen Unterschieden führen.
- Schnelle Implementierungen sind nicht immer offensichtlich.

## Zweck der Vorlesung:

- Sie lernen ein paar schnelle elegante Algorithmen für Grundprobleme kennen
- Sie erhalten Werkzeuge, um Algorithmen zu analysieren und rauszufinden, ob sie 'gut' sind
- Sie lernen 'algorithmisches Denken', was Sie zum Entwickeln neuer Algorithmen brauchen



# Algorithmen

Michael Kaufmann

04/11/2020 – 2. Vorlesung  
Pseudocode - O-Notation - Laufzeitanalyse

## Erstes Beispiel aus Dasgupta: Fibonacci Zahlen

$$F(n) = \left\{ \begin{array}{ll} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F(n-1) + F(n-2), & \text{falls } n > 2 \end{array} \right\}$$

Ergibt die Folge 0,1,1,2,3,5,8,13,21, ....

## Algorithmus 1: nicht-rekursive einfache Schleife

```
1 Algorithm: FibLoop( $n$ )
2 Allokieren  $F$  als ein Array der Länge  $n + 1$ ;
3  $F[0] \leftarrow 0$ ;
4  $F[1] \leftarrow 1$ ;
5 for  $i \in \{2, \dots, n\}$  do
6   |  $F[i] \leftarrow F[i - 1] + F[i - 2]$ ;
7 end
8 return  $F[n]$ 
```

## Algorithmus 2: Rekursion

```
1 Algorithm: FibRekursiv( $n$ )  
2 if  $n = 0$  then  
3   | return  $0$   
4 end  
5 if  $n = 1$  then  
6   | return  $1$   
7 end  
8 return  $\text{FibRekursiv}(n - 1) + \text{FibRekursiv}(n - 2)$ 
```

## Laufzeiten der Fibonacci Algorithmen: Zählen die 'Elementaroperationen'

FibLoop(n)

- Alloziere ein Array der Länge  $n$ , entweder 1 OP (Pointer auf den Anfang des Arrays, oder  $\leq n$  OPs (Initialisiere mit Nullen)
- 2 Elementaroperationen für 2 Zuweisungen.
- In jedem Schleifendurchlauf addieren wir 2 Zahlen.  
Da  $2^{n/2} \leq F(n) \leq 2^n$  für  $n \geq 6$ , ist  $F(n)$  etwa  $n$  Bits lang.
- Schleife wird  $(n - 1)$ - mal durchlaufen.

Anzahl der Elementaroperationen:  $1 + 2 + (n - 1) \cdot n \approx n^2$

## Laufzeiten der Fibonacci Algorithmen: Zählen die 'Elementaroperationen'

FibRecursive(n)

- Anfangs jeweils 1 Elementaroperation
- Dann eine Addition und die Zeit für die rekursiven Aufrufe.
- Also  $T(n) = 4 + T(\text{addition}) + T(n-1) + T(n-2) \geq 4 + T(n-1) + T(n-2)$
- Vergleich mit Definition von  $F_n$  zeigt, dass  $T(n) \geq F_n$ .
- Wegen  $F_n \geq 2^{n/2}$  für  $n \geq 6$ , wir erhalten  $T(n) \geq 2^{n/2}$ .

## Laufzeiten der Fibonacci Algorithmen: Vergleich

- Laufzeit von FibRecursive ist exponentiell in  $n$ . Also ist die Laufzeit sogar für mittelgroße  $n$  zu groß für alle denkbare Computer.
- Laufzeit von FibLoop ist quadratisch in  $n$ . Okay für relativ große  $n$ , aber nicht toll. Immerhin 'polynomiell'.
- Geht es besser als mit FibLoop ? Überlege !

Pseudocode



## Formulieren der Algorithmen in Pseudocode

- Brauchen kompakte, informelle high-level Beschreibung der Funktionsweise
- Ähnlich wie Programmiersprache, aber verständlich für Menschen, weniger für Maschinen

### Konventionen:

- Benutzen Standardkonstrukte wie IF, FOR, WHILE
- keine allzu feste Syntax
- oftmals high-level Anweisungen: Alloziere / Sortiere ein Array
- Oft unterschiedliche Syntax, nicht so wichtig hier. Hauptsache lesbar.
- Schreibe einfachen, verständlichen Code, keine Tricks
- Benutze Einrückungen
- latex mit 'usepackagealgorithm2e'

## Beispiel: Suchen in einem Array $A$ nach einem Element $a$

```
1 Algorithm: NaiveSuche( $A, a$ )
2  $it \leftarrow 0$ ; foundit  $\leftarrow$  false;
3 while (NOT foundit) AND ( $it < \text{length}(A)$ ) do
4   |    $it \leftarrow it + 1$ ;
5   |   if  $A(it) = a$  then
6   |   |   foundit = true; position = it
7   |   end
8 end
9 if foundit then
10  |   return position
11 end
12 else
13  |   return nicht gefunden
14 end
```

Wie man algorithmische Qualität misst:  
O-Notation

# O-Notation

- Betrachten Laufzeiten, abhängig von der 'Größe' des Problems'

- Problemgröße ist Länge der Eingabe

- Interessant sind große Probleminstanzen, kleine gehen eh schnell

⇒ betrachten Funktionenwachstum (i.e. Laufzeitverhalten)

- Genauer: Interessieren uns nur für Größenordnung:

- Also statt Laufzeit  $3 \cdot n^3 + 50 \cdot n^2 + 10000$  betrachte  $n^3$  (keine Konstanten, nur größter Term)

⇒ O-Notation

# O-Notation

$f \in O(g)$  bedeutet 'f ist von Ordnung höchstens g':

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) \leq cg(n)$$

$$\Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$f \in o(g)$  bedeutet 'f ist von Ordnung echt kleiner als g':

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) < cg(n)$$

$$\Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

# O-Notation

$f \in \Omega(g)$  bedeutet 'f ist von Ordnung mindestens g':

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) \geq cg(n) > 0$$

$$\Leftrightarrow 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$$

$$\Leftrightarrow g \in O(f)$$

$f \in \omega(g)$  bedeutet 'f ist von Ordnung echt größer als g':

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) > cg(n) > 0$$

$$\Leftrightarrow \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$\Leftrightarrow g \in o(f)$$

$f \in \Theta(g)$  bedeutet 'f ist von selber Ordnung wie g'

$$\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\Leftrightarrow 0 < \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$\Leftrightarrow f \in O(g) \text{ und } f \in \Omega(g)$$

## Beispiele:

- Betrachte  $f(n) = n^2 + 3n + 7$ .  
 $f \in O(n^8); f \in o(n^8); f \in \omega(\log n); f \in \Theta(n^2)$
- Betrachte  $f(n) = n^2 + 10n + 5$ .  
 $f \in O(n^2);$  aber  $f \notin o(n^2); f \in \Omega(n^2);$  aber  $f \notin \omega(n^2)$

## Spezialfälle:

- Ist  $f$  Polynom mit Grad  $d$ , so ist  $f \in \Theta(n^d)$ . Außerdem  $f \in o(\exp(n))$  sowie  $f \in \omega(\log n)$ .
- Bei Logarithmen ist Basis egal:  $\log_a \in \Theta(\log_b(n))$ .
- Funktionen, die durch eine Konstante  $b > 0$ , wobei  $b$  unabhängig von  $n$  ist, ist  $O(1)$ .

## Rechenregeln:

- $f \in O(g_1 + g_2)$  und  $g_1 \in O(g_2) \Rightarrow f \in O(g_2)$ .
- $f_1 \in O(g_1)$  und  $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$
- $f \in g_1 \cdot O(g_2) \Rightarrow f \in O(g_1 g_2)$
- Für Konstante  $c > 0$  gilt:  $O(cg)$  ist gleich  $O(g)$ .
- $f \in O(g_1)$ ,  $g_1 \in O(g_2) \Rightarrow f \in O(g_2)$
- Oft schreibt man  $f = O(g)$ , wenn man  $f \in O(g)$  meint.



# Laufzeitanalyse

## **Keine Laufzeitmessung !**

Da abhängig von benutztem Rechner und jeweiliger Eingabe

Benutzen 'RAM' Modell und zählen 'Elementaroperationen'

**Random Access Machine (RAM):** sequentielles Abarbeiten,  
uniformer Speicherzugriff

## **Elementaroperationen:**

- Laden und Schreiben eines Bits vom und auf den Speicher
- Elementare Arithmetik in konstanter Zeit ( $O(1)$ )
- Zahlen in 64-Bit Arithmetik ebenfalls in  $O(1)$
- Bei langen Zahlen Arithmetik nicht mehr elementar !!

## **Beispiele:** Mitüberlegen ! In O-Notation !

- Was kostet Addition zweier standard Integer? Zweier Zahlen der Länge  $n$  ?
- Was kostet Multiplikation zweier standard Integer? Zweier Zahlen der Längen  $n$  und  $m$  ?
- Lies  $n$  Strings der Länge 10 ein
- Was kostet Operation  $a + b \cdot c$ , falls  $a$  aus  $n^2$  Bits besteht,  $b$  aus  $\log n$  Bits,  $c$  aus  $n$  Bits

**Wie ist nun die Laufzeit eines Algorithmus ?**

**Wie schwierig ist ein 'Problem' ?**

# Probleme und Instanzen

**Problem:** Abstrakte Frage, die wir lösen wollen. Menge von Eingaben plus die gewünschten Ausgaben

Verschiedene Typen von Problemen: Entscheidungsproblemen, Optimierungsprobleme, Suchprobleme, etc

Formales gibt es in 'Theoretischer Informatik' (4. Semester)

**Instanz:** Konkreter Fall eines Problems, z.b. explizites Array, das zu sortieren ist.

Wie ist die Laufzeit eines Algorithmus?

→ Instanzen haben verschiedene Laufzeiten

→ Worst Case / Average Case ?

# Worst Case Laufzeit

Was ist die längste Laufzeit für den Algorithmus, maximiert über alle möglichen Instanzen ?

- Sei  $\mathcal{I}_n$  der Raum aller Instanzen der Länge  $n$
- Sei  $T(I_n)$  die Laufzeit des Algorithmus an Instanz  $I_n$ .
- Dann ist die Worst Case Laufzeit für Eingaben der Länge  $n$  definiert als:

$$T_{wc}(n) := \max\{T(I_n) \mid I_n \in \mathcal{I}_n\}$$

Beispiel: NaiveSuche( $A, a$ ).

Schaue nach:

Beachte, im Worst Case müssen wir  $n$  while-Schleifen durchlaufen. Sollte sowas wie  $2 + n \cdot 7 + 2 \in O(n)$  rauskommen

## Untere Schranken / obere Schranken im Worst Case

- Für eine untere Schranke  $T_{lower}$ , finde eine Instanz  $I_n$  mit Laufzeit mindestens  $T_{lower}$ . Dann gilt:  $T_{wc}(n) \geq T_{lower}$
- Für die obere Schranke  $T_{upper}$ , muss gelten, dass die Laufzeit für alle Instanzen höchstens  $T_{upper}$  ist. Dann gilt:  $T_{wc}(n) \leq T_{upper}$

### Diskussion:

- Worst Case Szenarios sind wichtig in sicherheitskritischen Systemen.
  - Oft ist das Laufzeitverhalten viel besser, und das Worst Case Verhalten tritt nur in ganz seltenen Fällen auf.
- Mittlere Laufzeit (Average Case)

# Average Case Laufzeit

Was ist die mittlere Laufzeit für den Algorithmus, gemittelt über alle möglichen Instanzen ?

- Sei  $\mathcal{I}_n$  der Raum aller Instanzen der Länge  $n$
- Sei  $T(I_n)$  die Laufzeit des Algorithmus an Instanz  $I_n$ .
- Dann ist die Average Case Laufzeit für Eingaben der Länge  $n$  definiert als:

$$T_{ac}(n) := \frac{1}{|\mathcal{I}_n|} \cdot \sum_{I_n \in \mathcal{I}_n} T(I_n)$$

## Bemerkung:

- Oft ist es kritisch, anzunehmen, dass alle Instanzen gleichwahrscheinlich sind. Manche kommen öfters vor als andere.
- Hier meist Worst Case Szenarien.

**Nützliches zum Rechnen:  
Logarithmen, Fakultätsfunktion,...**



# Logarithmen-Regeln

$$\log(a \cdot b) = \log a + \log b$$

$$\log(a^b) = b \log a$$

$$\log_a b = \frac{\log b}{\log a}$$

$$b^{\log_b a} = a$$

Überlege: Was ist  $2^{\log_4 n}$  ?

$n!$

$$n! = n \cdot (n - 1) \cdot (n - 2) \dots 3 \cdot 2 \cdot 1$$

$$n! = \sqrt{2\pi n} \cdot (n/e)^n \cdot (1 + o(1)) \quad \text{Stirling-Approx.}$$

$$\text{einfacher: } (n/2)^{n/2} \leq n! \leq n^n$$

# Summen

$$\sum_{i=1}^n i = n(n+1)/2$$

$$\sum_{i=0}^n x^i = \frac{x^{n+1}-1}{x-1} \quad \text{geometr. Reihe}$$

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

$$\sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2} \quad \text{für } |x| < 1 \quad \text{warum ? Ableiten !!}$$

$$\sum_{i=1}^n 1/i = 1 + 1/2 + 1/3 + \dots = \ln n + O(1) \quad n\text{-te harmon. Zahl}$$

# Algorithmen

Michael Kaufmann

10/11/2020 – 3. Vorlesung  
Rekursionen

# Teaser aus Dasgupta, Mehlhorn et al.

## Multiplikation zweier Integer

Für Integer der Länge jeweils  $n$  braucht Schulmethode  $O(n^2)$  Elementaroperationen

Formell: Wir nehmen Binärzahlen der Länge  $n$  an, und sei  $n$  eine Zweierpotenz (kann man immer schön halbieren).

Machen einen anderen Ansatz, nämlich Divide & Conquer:

- Spalte  $x$  und  $y$  in linke und rechte Hälften, so dass

$$x = 2^{n/2}x_l + x_r \text{ und } y = 2^{n/2}y_l + y_r$$

- Multipliziere kleinere Teilprobleme und Addiere:

$$x \cdot y = 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r$$

→ haben jetzt 4 Teilprobleme,

i.e. 4 Multiplikationen zweier Zahlen der Länge  $n/2$

# Multiplikation zweier Zahlen

Nun verwenden wir Rekursion auf die Multiplikation zweier Zahlen der Länge  $n/2$ .

Das ergibt

$$\begin{aligned} T(n) &= 4 \cdot T(n/2) + O(n) \text{ wobei} \\ T(1) &= 1 \end{aligned}$$

Dabei ist  $T(n)$  die Laufzeit für die Multiplikation zweier Zahlen der Länge  $n$ ,  $n > 1$ .

Kurzes Nachdenken führt zu  $O(n^2)$ .

# Multiplikation - Verbesserung

Versuche eine der 4 Multiplikationen zu sparen !

Das Ziel ist dann :  $T(n) = 3T(n/2) + O(n)$  für  $n > 1$ .

Und zwar so:

$$\begin{aligned}x \cdot y &= 2^n x_l y_l + 2^{n/2} (x_l y_r + x_r y_l) + x_r y_r \\&= 2^n x_l y_l + 2^{n/2} ((x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r) + x_r y_r \\&= (2^n - 2^{n/2}) x_l y_l + (1 - 2^{n/2}) x_r y_r + 2^{n/2} ((x_l + x_r)(y_l + y_r))\end{aligned}$$

Das ergibt 3 Multiplikationen rekursiv und ein paar Summationen mehr ( $O(n)$ ).

Lösung obiger Rekursionsgleichung ergibt  $O(n^{\log 3}) = O(n^{1.59})$

Wie kommt man da drauf ? Wie löst man Rekursionen ?

# Rekursionen

**Merge-Sort:**  $T(n) = 2T(n/2) + n$

**Binäre Suche:**  $T(n) = T(n/2) + 1$

**Beachte:**  $n$  wird als Zweierpotenz angenommen !

**Beachte:**  $T(1)$  muss spezifiziert werden, meist  $T(1) = 1$ .

## 1. Raten und Beweisen:

a. **Raten**  $T(n) = O(\log n)$  für binäre Suche

b. **Beweisen**  $T(n) \leq c \log n$  für geeignete Konstante  $c$

per Induktion:  $T(2)$  ist ok,  $T(n)$  auch durch Einsetzen mit  $c = 2$ .



# Rekursion: 2. Ausrechnen

$T(n) = T(n/2) + 1$ ,  $T(1) = 1$  und  $n$  ist Zweierpotenz.

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= T(n/4) + 1 + 1 \\ &= T(n/8) + 1 + 1 + 1 \\ &\quad \vdots \\ &= T(n/2^i) + i \\ &= T(n/2^{i+1}) + 1 + i \end{aligned}$$

make here Induktion über  $i$  !

für  $i + 1 = \log n$ :

$$\begin{aligned} &= T(n/2^{\log n}) + \log n \\ &= T(1) + \log n \\ &= 1 + \log n = O(\log n) \end{aligned}$$

# Rekursion: 3. Mastertheorem

**Satz:** Sei  $a, b \geq 1$  konstant.  $f(n), T(n) \geq 0$  mit

$$T(n) = aT(n/b) + f(n)$$

1. Für  $f(n) = O(n^{\log_b a - \epsilon})$  mit  $\epsilon > 0$  gilt  $T(n) = \Theta(n^{\log_b a})$
2. Für  $f(n) = \Theta(n^{\log_b a})$  gilt  $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$
3. Für  $f(n) = \Omega(n^{\log_b a + \epsilon})$  mit  $\epsilon > 0$  und  
und  $a \cdot f(n/b) \leq c \cdot f(n)$  für  $c < 1$  und  $n$  genügend groß, gilt  
 $T(n) = \Theta(f(n))$

# Rekursion: 3. Mastertheorem

**Annahme:**  $n = b^i$ , also Teilproblemgrößen  $1, b, b^2, \dots$

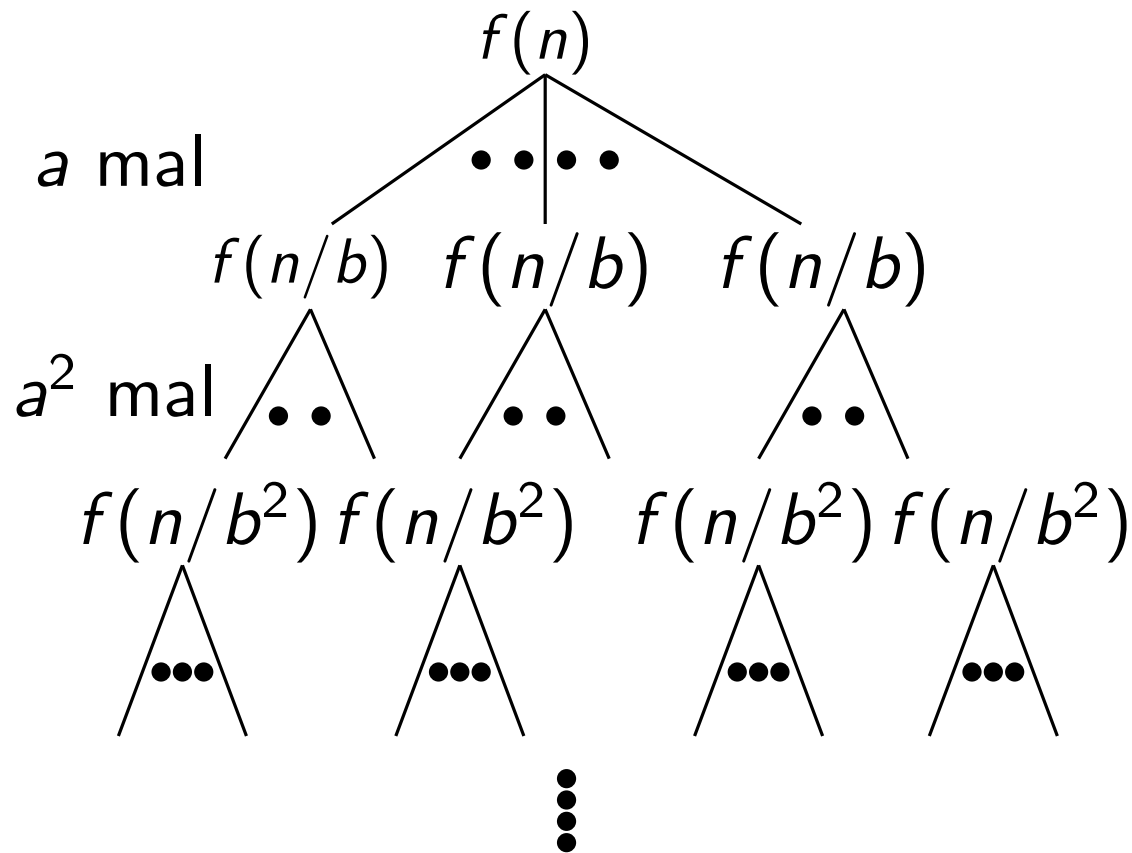
**Lemma:** Sei  $a, b \geq 1$ ,  $f(n) \geq 0$ ,  $n = b^i$  für  $i \in \mathbb{N}$ . Mit  $T(1) = \Theta(1)$  gilt

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j \cdot f(n/b^j)$$

**Beweis:** 
$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + a^2 T(n/b^2) \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + a^3 T(n/b^3) \\ &\quad \vdots \\ &= f(n) + \dots + a^{\log_b n - 1} \cdot f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\ &= \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j \cdot f(n/b^j) \end{aligned}$$

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j \cdot f(n/b^j)$$

Rekursionsbaum:



⇒ 3 Fälle

$f(n)$

$a f(n/b)$

$a^2 f(n/b^2)$

$a^3 f(n/b^3)$

⋮

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

---

obige Summe

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j \cdot f(n/b^j)$$

**Lemma:** Sei  $a, b \geq 1$ ,  $f(n)$  definiert auf Potenzen von  $b$ .

$$\text{Sei } g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j).$$

1. Ist  $f(n) = O(n^{\log_b a - \epsilon})$  mit  $\epsilon > 0$ , so ist  $g(n) = O(n^{\log_b a})$
2. Ist  $f(n) = \Theta(n^{\log_b a})$ , so ist  $g(n) = \Theta(n^{\log_b a} \cdot \log n)$ .
3. Ist  $af(n/b) \leq cf(n)$ , mit  $c < 1$ ,  $n > b$ , so ist  $g(n) = \Theta(f(n))$ .

$\Rightarrow$  **Mastertheorem:**

1.  $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$ .
2.  $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^{\log_b a} \cdot \log n)$ .
3. Ist  $f(n) = \Omega(n^{\log_b a + \epsilon})$  mit  $\epsilon > 0$ , und  $af(n/b) \leq cf(n)$ , so ist  $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$ .

# Mastertheorem - Anwendung

## Beispiele:

Einfacher Ansatz für Multiplikation:

$$T(n) = 4 \cdot T(n/2) + O(n) \text{ mit } T(1) = 1$$

Setze ein:  $a = 4, b = 2, f(n) = cn$ .

Damit gilt:  $\log_b a = \log_2 4 = 2$ . Und  $n^{2-\epsilon} > cn$  für  $\epsilon = 1/2$

Wende Fall 1 des Mastertheorems an und somit

$$T(n) = O(n^{\log_b a}) = O(n^2)$$

Besserer Ansatz für Multiplikation:

$$T(n) = 3 \cdot T(n/2) + O(n) \text{ mit } T(1) = 1$$

Setze ein:  $a = 3, b = 2, f(n) = cn$ .

Also:  $\log_b a = \log_2 3$  Und  $n^{\log_2 3 - \epsilon} > cn$  für  $\epsilon = 1/2$

Wende Fall 1 des Mastertheorems an und somit

$$T(n) = O(n^{\log_b a}) = O(n^{\log_2 3}) = O(n^{1.59})$$

## Weitere Beispiele:

Binäre Suche:

$$T(n) = T(n/2) + O(1) \text{ mit } T(1) = 1$$

Setze ein:  $a = 1, b = 2, f(n) = 1 = n^0$

Damit gilt:  $\log_b a = \log_2 1 = 0$ . Und  $n^0 = f(n)$ .

Wende also Fall 2 des Mastertheorems an und somit

$$T(n) = O(n^0 \log_2 n) = O(\log_2 n)$$

GIBT ES FÄLLE, WO MASTERTHEOREM NICHT  
ANWENDBAR ? ÜBERLEGE ! UND DANN ??

# Algorithmen

Michael Kaufmann

11/11/2020 – 4. Vorlesung  
Grundlegende Datenstrukturen



## **II. Grundlegende Datenstrukturen**

### **a. Arrays und Listen**

## II.a Arrays und Listen

**Arrays:** Die allereinfachste Struktur, sollte jeder kennen

- Array der Länge  $n$  ist eine Anordnung von  $n$  Objekten des selben Typs in fortlaufenden Adressen im Speicher.
- Standardoperationen: Lese/Schreibe ein Element. Geht in  $O(1)$ , da Adresse im Speicher bekannt.
- Nachteil: Müssen Speicher für Array im Voraus allozieren. Größe ist fest.

**Listen:** Können ihre Größe (= Länge) ändern.

- Elemente besteht aus Schlüssel und Zeiger *next* auf das nächste Element Zusätzlich gibt es noch einen Zeiger auf das Startelement der Liste.
- In doppelt verketteten Listen gibt es jeweils noch Zeiger *prev* auf das Vorgängerelement

# Operationen für Listen (doppelt verkettet)

- Einfügen ein Element  $x$  nach Element  $e$  in der Liste:  $O(1)$  (falls man Element  $e$  hat)
- Löschen eines Elements aus der Liste:  $O(1)$
- Suche Element mit Schlüsselwert  $k$ :  $O(n)$
- Löschen einer Teilliste, wenn erstes/letztes Element bekannt ist):  $O(1)$
- Füge zweite Liste nach einem speziellen Element in der ersten Liste ein:  $O(1)$

Implementierung mit speziell markierten Elementen *head* und *tail* for Anfang und Ende  
oder ringförmig mit einem 'Wächter'-Element ?

# Operationen für Listen (einfach verkettet)

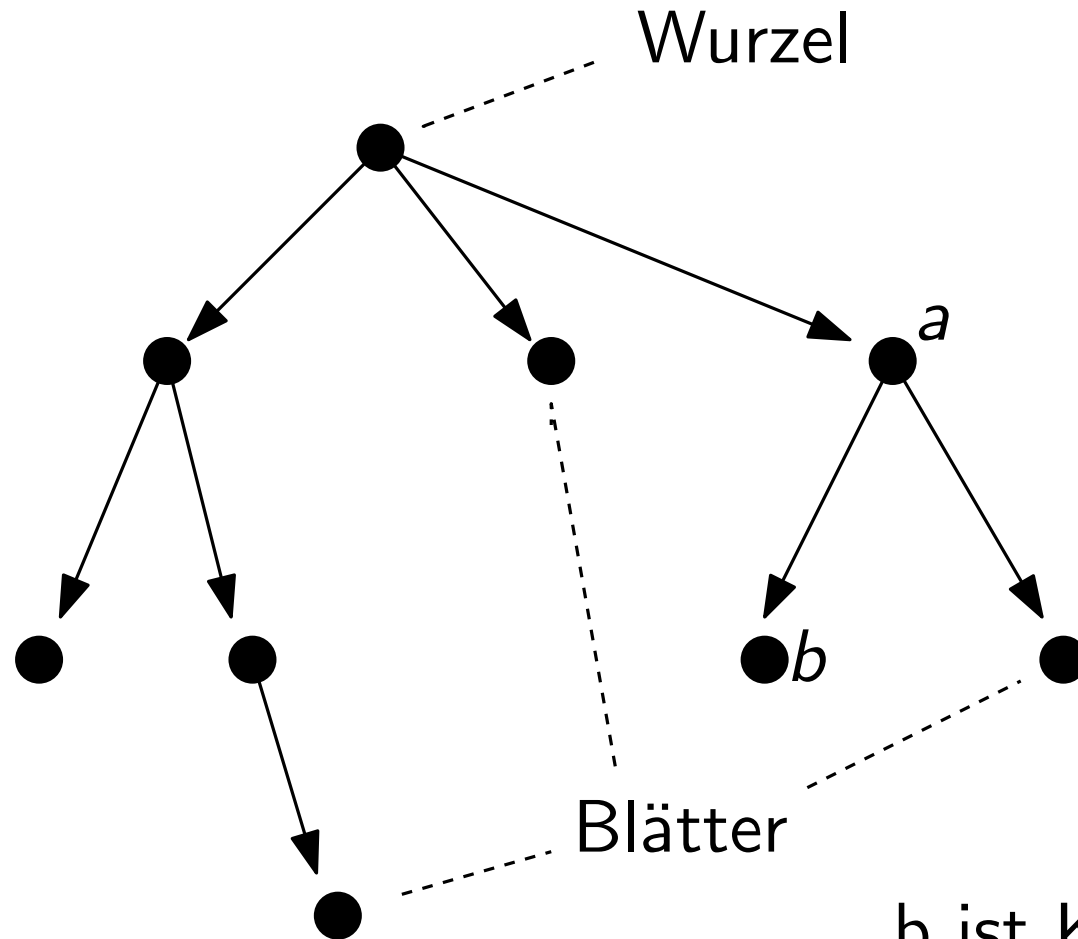
- Da nur ein Zeiger pro Element, weniger Speicherplatz
- Löschen von Elementen geht nicht so gut, da wir keine prev - Zeiger haben

# Vergleich Listen und Arrays

**Listen:** dynamisch, wir brauchen nicht Platz zu allozieren, aber Zugriff auf bestimmtes Element kann  $O(n)$  dauern

**Arrays:** Schneller Zugriff über Index, Nachteil: Allozieren. Müssen Größe von vorneherein festlegen

## II b. Bäume



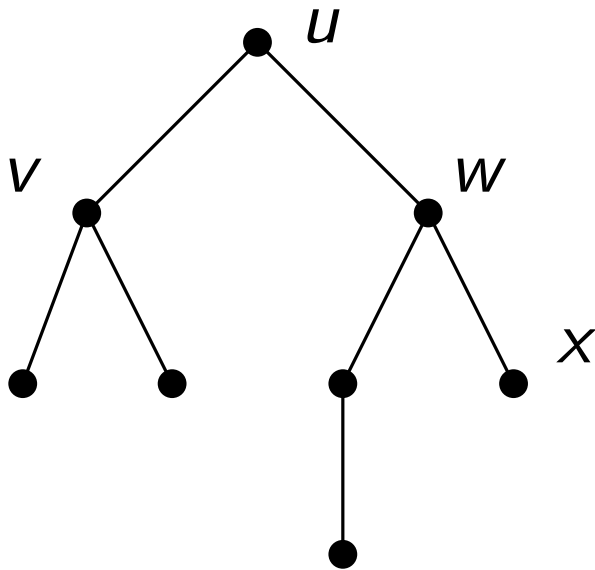
b ist Kind von a.  
a ist Vater/parent von b.

# Bäume

Sei  $v$  Knoten in einem Baum  $T$ .

$\text{Tiefe}(v) = 0$  falls  $v$  Wurzel,  
 $= \text{Tiefe}(\text{Vater}(v)) + 1$  falls  $v$  nicht Wurzel,

$\text{Höhe}(v) = 0$ , falls  $v$  Blatt,  
 $= 1 + \max\{\text{Höhe}(w), w \text{ Kind von } v\}$ , sonst

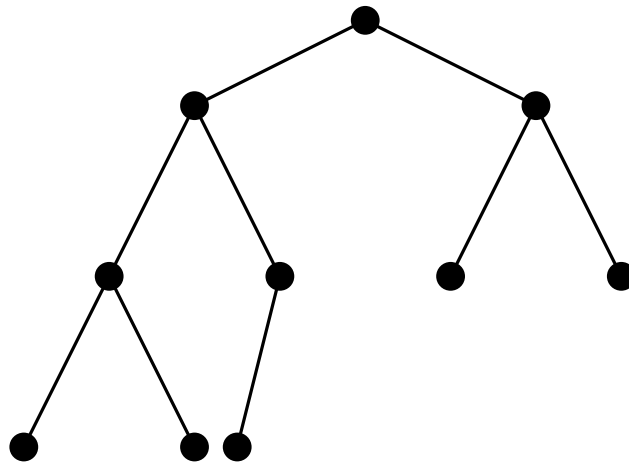


- $\text{Tiefe}(u) = 0$
- $\text{Tiefe}(v) = \text{Tiefe}(w) = 1$
- $\text{Tiefe}(x) = 2$
- $\text{Höhe}(x) = 0$
- $\text{Höhe}(v) = 1$
- $\text{Höhe}(u) = 3$

**Allgemein:**  $\text{Höhe}(T) = \max\{\text{Höhe}(v), v \in T\}$   
 $\text{Tiefe}(T) = \max\{\text{Tiefe}(v), v \in T\}$

# Binärbäume

- Jeder Knoten hat höchstens 2 Kinder, ein linkes und ein rechtes (lson/rson)
- In einem *vollständigen/complete* Binärbaum sind alle Level außer dem untersten gefüllt. Dort stehen die Knoten möglichst weit links.
- In einem *vollen/full* Binärbaum ist auch das unterste Level komplett voll.



Vollständig !



# Verhältnis Höhe zur Knotenanzahl

**Lemma:** (a) Ein voller Binärbaum mit  $n$  Knoten hat Höhe/Tiefe  $\log_2(n + 1) - 1 \in \Theta(\log n)$ .

(b) Für vollständige Binärbäume ist die Formel  $\lceil \log_2(n + 1) - 1 \rceil \in \Theta(\log n)$ .

**Beweis:** Anzahl der Knoten in einem vollen Binärbaum der Tiefe  $h$  ist

$$1 + 2 + 4 + \dots + 2^h = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$

Damit gilt:

$$n = 2^{h+1} - 1 \Leftrightarrow h = \log_2(n + 1) - 1 \quad (\text{a})$$

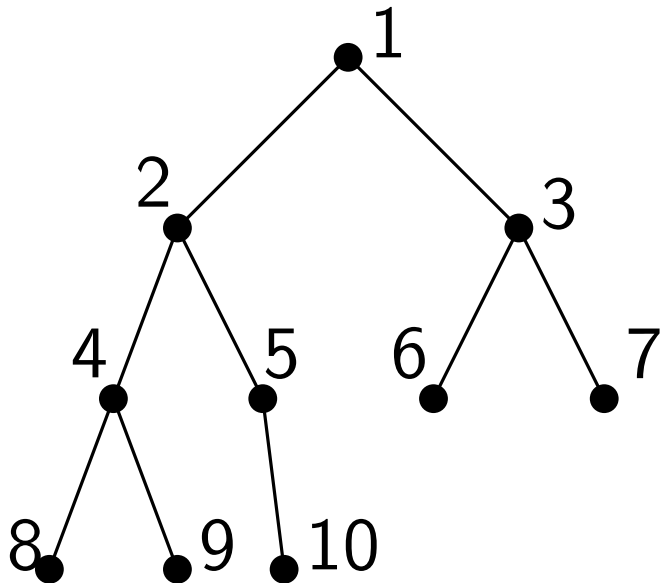
(b) Im Fall von vollständigen Binärbaum ist das letzte Level eventuell nicht ganz voll. Deshalb wird einfach aufgerundet:

$$h = \lceil \log_2(n + 1) \rceil - 1$$

# Wie werden Bäume implementiert ?

## Vollständige Binärbäume:

- Alle Blätter haben Tiefe  $k$  oder  $k + 1$ .
- Blätter auf Tiefe  $k + 1$  sitzen 'ganz links'.



Nummerierung schichtweise  
von links nach rechts

Für Knoten mit Nummer  $i$  gilt:

- Vater hat Nummer  $\lfloor i/2 \rfloor$
- Kinder haben Nummern  $2i$ ,  $2i + 1$ .

- wird als Array dargestellt, nur konzeptuell als Baum!  
Indizierungsregeln ergeben Vater-Kind-Relationen

## **Allgemeine Binärbäume:**

Jeder Knoten hat seinen Schlüsselwert, sowie Zeiger auf sein linkes und rechten Kindknoten, sowie einen Zeiger auf seinen Vaterknoten

Ist die Zahl der Kinder pro Knote beschränkt (z.B. 4), kann man das auch so machen.

## **Allgemeine Bäume:**

Geschwisterknoten sind nicht beschränkt, deshalb:

Kinder eines Knotens  $v$  werden in einer Liste organisiert.

$v$  hat einen Zeiger auf sein linkestes Kind. Jeder Knoten hat einen Zeiger auf seinen rechten Geschwisterknoten. Dies ergibt dann die Liste der Geschwister von  $v$ , von links nach rechts.

# Keller + Warteschlangen (Stacks + Queues)

## Keller:

Dynamische Datenstruktur, bei der immer nur das oberste Element zugreifbar ist.

Operationen: *Push*( $x$ ) fügt Element  $x$  oben auf den Keller ein.

*Pop* löscht das oberste Element vom Keller.

Prinzip: LIFO (Last in First out)

## Implementierung:

- Array mit Zeiger auf das augenblickliche Top-Element.

Operationen gehen in  $O(1)$ .

- Bei doppelt verkettete Liste: Zeiger einfach ans Ende der Liste. Dort steht das Top-Element.

- Bei einfacher Verkettung füge die Element vorne ein! Und halte den Zeiger auf das erste Element!

# Warteschlangen (Queues)

Dynamische Datenstruktur, die nach FIFO (First in first out) funktioniert.

Operationen: *Einfügen*( $x$ ) fügt Element  $x$  ans Ende der Queue (Enqueue). *Löschen* löscht das erste Element der Queue.

## **Implementierung:**

- Array mit Zeiger auf das augenblickliche End- sowie Top-Element. Operationen gehen in  $O(1)$ .
- Bei doppelt verkettete Liste ....
- Bei einfacher Verkettung ....

Heaps

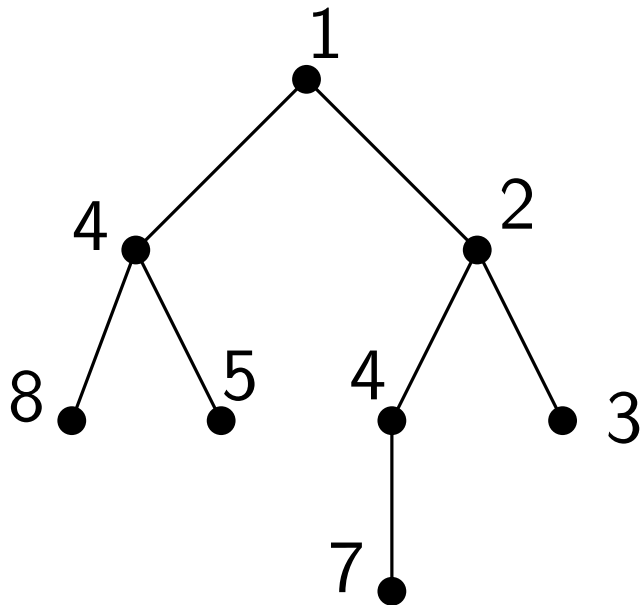
# Was sind Heaps ?

- Einfache baumartige Datenstruktur zur Verwaltung einer Menge mit Ordnungsrelation.
- Unterstützt Operationen wie Insert, IncreaseKey, DecreaseKey, ExtractMin
- Kompromiss zwischen ungeordnetem Array und geordnetem Array (Überlege Unterschiede, Vorteile, Nachteile)

# Heaps: Grundlagen

Speichern Menge  $S$  im Heap  $T$  ab.  
Knoten in  $T$  entsprechen Elementen in  $S$ .

**Heapeigenschaft:** Für Knoten  $u$  und  $v$  mit  $\text{Vater}(v) = u$  gilt:  
 $S[u] \leq S[v]$



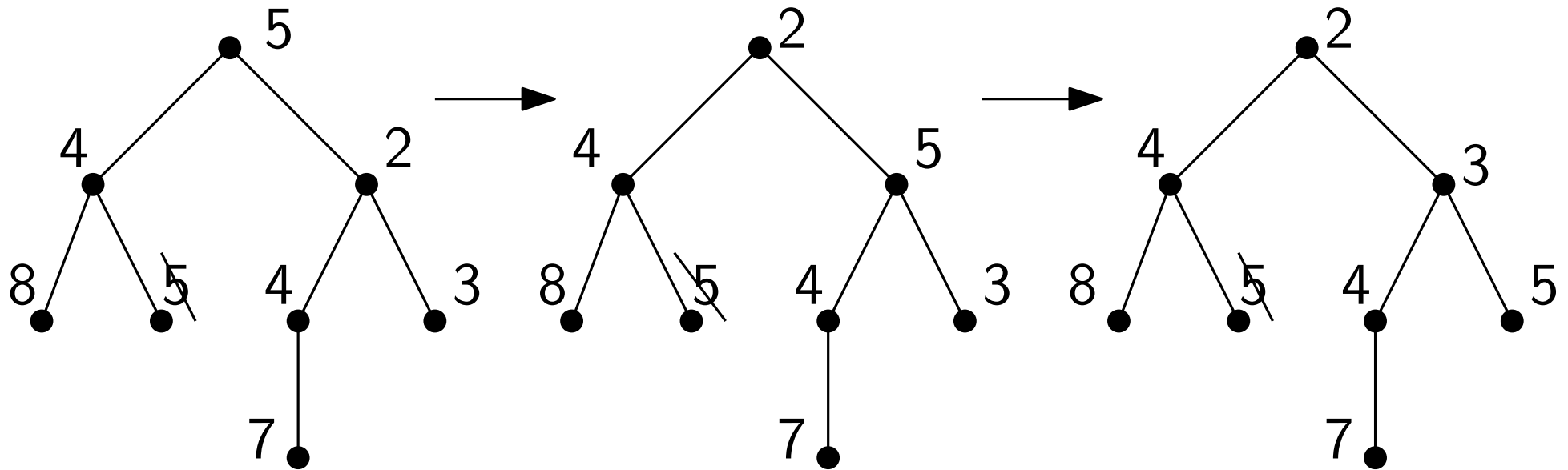
$$S = \{1, 2, 3, 4, 5, 4, 7, 8\}$$



# Operationen: ExtractMin

Im Heap steht Minimum in der Wurzel. Also:

1. **Suche Minimum**  $\rightarrow$  Wurzel  $O(1)$
2. **Entferne Minimum aus Heap. Repariere Heap:**
  - Nimm Blatt und füge es in Wurzel ein.
  - Lass es nach unten sinken (Vergleich mit kleinstem Kind)



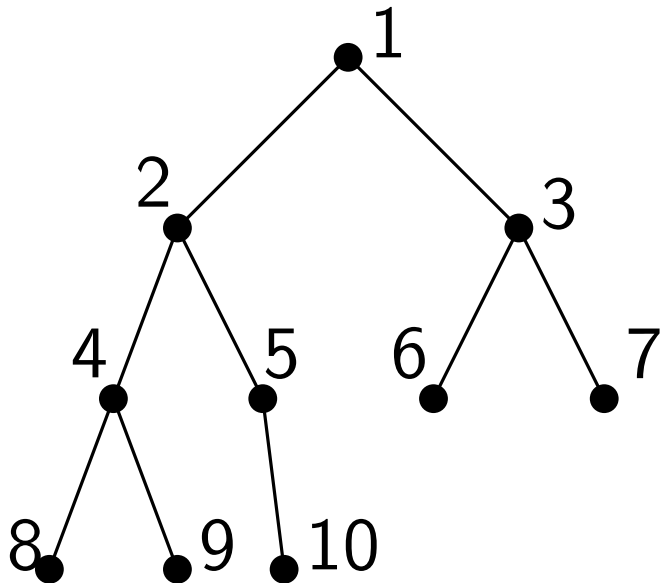
**Laufzeit:**  $O(\text{Höhe}(T))$

**Problem:** Mache Höhe klein!

# Wie werden Heaps implementiert ?

## Als vollständige Binärbäume:

- Alle Blätter haben Tiefe  $k$  oder  $k + 1$ .
- Blätter auf Tiefe  $k + 1$  sitzen 'ganz links'.



Nummerierung schichtweise  
von links nach rechts

Für Knoten mit Nummer  $i$  gilt:

- Vater hat Nummer  $\lfloor i/2 \rfloor$
- Kinder haben Nummern  $2i$ ,  $2i + 1$ .

- wird als **Array** dargestellt, nur konzeptuell als Baum!  
Indizierungsregeln ergeben Vater-Kind-Relationen

# HeapAufbau/Initialisierung: Algorithmus

## Initialisierung des Heaps $H$ :

für alle  $a \in S$  {**Insert**( $a, H$ )}

## **Insert**( $a, H$ ):

sei  $n$  die Größe von  $H$ ;

$n \leftarrow n + 1$ ;

$H(n) \leftarrow a$ ;

$i \leftarrow j \leftarrow n$ ;

**while** ( $i > 1$ ) {

$j \leftarrow \lfloor j/2 \rfloor$ ;

**if** ( $H[j] > H[i]$ ) {tausche  $H[i]$  und  $H[j]$ ;  $i \leftarrow j$ };

**else**  $i \leftarrow 1$ ;

};

**Beispiel!**

# HeapAufbau: Analyse

## Laufzeit:

**Initialisierung** (Einfügen, Hochsteigen lassen):  $O(n \cdot \text{Höhe}(H))$

$\text{Höhe}(H)$  ist  $O(\log n)$ , da Baum vollständig ist

## Lemma:

Ein ausgewogener Baum der Höhe  $k$  hat mindestens  $2^k$  Knoten.

## Beweis:

Der kleinste Baum der Höhe  $k$  hat nur einen Knoten auf Tiefe  $k$ .

Auf Stufe  $i$  gibt es  $2^i$  Knoten für  $0 \leq i < k$ .

$$\Rightarrow \sum_{i=0}^{k-1} 2^i + 1 = 2^k.$$

$\Rightarrow$  Für Heap mit  $n$  Knoten der Höhe  $k$  gilt  $n \geq 2^k$

$\Rightarrow \log n \geq k = \text{Höhe}(H)$

# Heapaufbau: Geht es besser ?

**Warum dieser Algorithmus nicht schneller sein kann:**

Füge Schlüssel in absteigender Reihenfolge an, also z.B.  $n, n - 1, n - 2, \dots, 2, 1$ . Bei jedem Hochsteigen muss der jeweilige Schlüssel bis ganz hoch in die Wurzel.

Das sind bei fast allen Schlüsseln mehr als  $1/2 \cdot \log n$  Schritte. Also insgesamt  $\Omega(n \log n)$ .

**Alternative:** Füge die Schlüssel nacheinander ein, OHNE gleich hochsteigen zu lassen.

Dann mache alle Unterbäume zu Heaps, startend bei den Blättern levelweise.

Genauer: Betrachte  $v$ . Dann sind beide Unterbäume von  $v$  schon Heaps. Nur wenn die Heapeigenschaft für  $v$  nicht gilt, lasse  $v$  runtersinken in Zeit  $O(h(v))$ , falls  $v$  die Höhe  $h$  hat. Danach ist Unterbaum mit Wurzel  $v$  ein Heap.

# Alternative hat nur $O(n)$ Laufzeit:

Überlegung: Behandeln eines Knotens auf Tiefe  $i$  hat Laufzeit  $O(\log n - i)$ .

Es gibt  $2^i$  Knoten davon. Also

Laufzeit:  $O(\sum_{i=0}^{\log n} (\log n - i) \cdot 2^i) = \dots = O(n)$ .

Wieder was verbessert, was offensichtlich vernünftig und nicht zu verbessern war !

# Operation: IncreaseKey

Sei  $H$  ein Heap von  $n$  Knoten. Jetzt wird der Schlüssel von Knoten  $v$  erhöht. Deshalb kann es sein, dass  $v$  jetzt die Heapeigenschaft verletzt.

Beachte: Alle anderen Knoten erfüllen die Heapeigenschaft. Lasse den Schlüsselwert nach unten sinken: Tausche den Platz mit dem Kind  $w$ , das den kleineren Schlüsselwert hat.

Beobachtung:  $v$  erfüllt die Heapeigenschaft. Eventuell erfüllt  $w$  immer noch nicht die Heapeigenschaft. Dann lasse den Schlüsselwert weiter nach unten sinken.

Laufzeit:  $O(h) = O(\log n)$  für Höhe  $h$  des Heaps

# Operation DecreaseKey

Sei  $H$  ein Heap von  $n$  Knoten. Jetzt wird der Schlüssel von Knoten  $v$  erniedrigt. Deshalb kann es sein, dass jetzt die Heapeigenschaft für  $\text{parent}(v)$  verletzt ist.

Beachte: Alle anderen Knoten erfüllen die Heapeigenschaft. Lasse den Schlüsselwert nach oben steigen: Tausche den Platz mit dem  $\text{parent}(v)$ .

Beobachtung:  $v$  erfüllt die Heapeigenschaft. Eventuell erfüllt  $\text{parent}(v)$  immer noch nicht die Heapeigenschaft. Dann lasse den Schlüsselwert weiter nach oben steigen.

Laufzeit:  $O(h) = O(\log n)$  für Höhe  $h$  des Heaps



**Anwendungen** später, z.B. HeapSort

**Einige Operationen gehen noch besser:** z.B.  
FibonacciHeaps (kürzeste Wege)

**Hat man viele billige Operationen, aber nur wenig teure,  
lohnt es sich oft, genauer hinzuschauen**

Jetzt : **Prioritätswarteschlangen**

# Algorithmen

Michael Kaufmann

17/11/2020 und 18/11/2020  
Prioritätswarteschlangen - Hashing

## **Prioritätswarteschlangen:**

Anwendung der bisherigen Datenstrukturen (Arrays, Listen, Heaps)

# Prioritätswarteschlangen (Priority Queues)

## **Szenario: Notaufnahme im Krankenhaus**

– Immer neue Patienten kommen rein. Jeder erhält eine 'Priorität', je nach Dringlichkeit. Die Fälle werden in der Reihenfolge ihrer Prioritäten behandelt.

Beachte dass Prioritäten sich auch ändern können (falls sich der Zustand eines Patienten verschlechtert) !

## **Szenario: Queue auf einem Clusterrechner**

– Jobs kommen unregelmäßig und andauernd an.

Normalerweise wird die Reihenfolge als Warteschlange organisiert. Manchmal gibt es aber Jobs mit höherer Priorität, die vorgezogen werden.

# Prioritätswarteschlangen

## Was wollen wir ? Formal:

- Datenstruktur, um eine dynamische Menge von Elementen zu verwalten.
- Jedes Element hat einen Prioritätswert.
- Operationen sind: Einfügen eines Elements, Erhöhen des Prioritätswertes, evtl. Erniedrigen des Prioritätswertes, Streichen des Element mit dem höchsten Prioritätswert

Wie realisieren wir diese Datenstruktur ?

# Realisierung

## **als ungeordnetes Array oder Liste**

- Einfügen geht in  $O(1)$  ans Ende der Liste/Arrays
- Streichen des Max-Elements geht in  $O(n)$  (Durchlaufen)
- DecreaseKey und IncreaseKey geht in  $O(1)$ , sofern die Position des Elements bekannt ist

# Realisierung

## als geordnetes Array oder Liste

- Erstmal Sortieren des Arrays in  $O(n \log n)$  (siehe später)
- Löschen des Max-Elements in  $O(1)$ , steht am Anfang/Ende des Arrays
- Einfügen muss an die richtige Stelle: In Liste mit Durchlaufen in  $O(n)$   
In Array mit binärer Suche in  $O(\log n)$ , dann aber alle restl. Einträge um 1 Stelle verschieben ( $O(n)$ ).
- DecreaseKey und IncreaseKey durch Verschieben des Elements an die richtige Stelle.  $O(n)$ .

# Realisierung

## als Heap

- Erstmal HeapAufbau in  $O(n)$
- Löschen des Max-Elements in  $O(1)$ , Reparieren in  $O(\log n)$
- Einfügen in  $O(\log n)$
- DecreaseKey und IncreaseKey in.  $O(\log n)$ .

## Zeigt Vorteile des Heaps

Ist irgendwie zwischen sortiertem und ungeordnetem Array  
Alle Operationen in  $O(\log n)$ , also bisschen teurer als  $O(1)$ ,  
aber keine richtig teuer ( $O(n)$ )



## 11.5 Hashing

# Hashing: Szenario

- Haben einen Online Shop und verwalten die Daten der Kunden, die in den letzten 20 Minuten auf unsere Seiten geklickt haben
- Kunden werden über IP Adressen identifiziert
- Menge an Personen, die online sind, ändert sich dauernd.
- Wollen schnellen Zugriff auf die Kundendaten

Erster Versuch:

- Allokiere ein Array für alle möglichen IP Adressen (je 32 Bits)
- Fülle Einträge für die Kunden, die gerade aktiv sind
- Dann geht der Zugriff in  $O(1)$

Nachteil: Anzahl aktiver Kunden ist VIEL kleiner als die Zahl aller möglichen IP Adressen. Speicherplatzverschwendung !!

# Verwaltung der aktiven Kunden

## **2. Versuch: Erzeuge Liste der IP Adressen**

– Effizient bzgl. Speicherplatz. Aber lange Zugriffszeit für einzelnen Kunden

## **3. Versuch: IP Adressen hintereinander in ein Array**

Aber: Welche Adresse steht wo? Zugriff auch ganz langsam.

**Lösung: Hashing**

# Hashing: 2. Szenario

## **Problem aus dem Programmierprojekt 'KI für Halma':**

Haben ca. 100 Felder und 10 Spielfiguren. Speichere alle erreichbaren Stellungen ab.

1. das sind viele
2. wollen schnellen Zugriff

**Methode der Wahl:** Hashing

# Hashing: Idee

Benutzen ein kleines, aber nicht zu kleines Array (Ansatz 3)

Aber anstatt fortlaufend Einträge zu schreiben, benutzen wir Funktion, die uns sagt, wo das Element steht. (HASH Funktion)

Beachte: Es können Fehler auftreten, nämlich wenn zwei Elemente den gleichen Hashfunktionswert haben.

Also: Mache das Array nicht zu klein ! Wähle Hashfunktion geschickt, so dass Fehler möglichst nicht oft auftreten

# Hashing

Gegeben Universum  $U = [0, \dots, N - 1]$  (alle möglichen IP Adr.)  
 $S \subset U$  sei die zu verwaltende Menge.  $|S| = n$  (Kunden online)

**Operationen:** Zugriff( $a, S$ ); Einfügen( $a, S$ ), Streichen( $a, S$ )

- Hashtafel  $T[0, \dots, m - 1]$  der Größe  $m$
- Hashfunktion  $h : U \rightarrow [0, \dots, m - 1]$  bildet Elemente  $a \in U$  auf Tafel  $T[h(a)]$  ab.
- $\frac{n}{m} = \beta$  heißt **Belegungsfaktor**

**Bsp:**  $N = 50, m = 3, S = \{2, 21\}, \quad h(x) = x \bmod 3$

**Problem:**

**Kollisionen!** Was tun falls  $x \neq y$  aber  $h(x) = h(y)$ ?

# Hashing mit Verkettung

Jedes Element  $T[i]$  der Hashtafel ( $0 \leq i \leq m - 1$ ) besteht aus einer Liste, die alle  $x \in S$  mit  $h(x) = i$  enthält.

**worst case:** alle  $x \in S$  in einer Liste  $\rightarrow O(n)$  Laufzeit

**Mittlere Analyse liefert was viel!!! besseres:**

**Wahrscheinlichkeitsannahmen:**

1.  $h(x)$  wird in  $O(1)$  berechnet.
2.  $h$  verteilt Elemente von  $U$  gleichmäßig auf  $T$ , also  $|h^{-1}(i)| = \frac{|U|}{m}$  für alle  $0 \leq i \leq m - 1$ .
3. Für  $n$  Operationen auf den Elementen gilt:  
 $prob(x = \text{Elem. der } j\text{-ten Operat}) = \frac{1}{N}$  für  $x \in U$   
(Operationen unabhängig, gleichverteilt)

Für  $x_k$  Arg, der  $k$ -ten Oper.:  $prob(h(x_k) = i) = \frac{1}{m}$   
(Hashwerte gleichverteilt)

# Hashing mit Verkettung

**Def.**  $\delta_h(x, y) = 1$  falls  $h(x) = h(y)$  für  $x \neq y$  und  
           $= 0$  sonst

$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$$

entspricht  $1 + \text{Länge der Liste in } T[h(x)]$

$\Rightarrow$  Kosten der Operation  $XYZ(x) = O(1 + \delta_h(x, S))$

**Satz:**

Erwartete Kosten von  $XYZ(x)$  sind  $1 + \beta = 1 + \frac{n}{m}$



# Erwartete Kosten bei Hashing mit Verkettung

## Beweis:

Sei  $h(x) = i$  und sei  $p_{ik}$  die Wahrsch., dass Liste  $i$  genau  $k$  Elemente enthält.

Dann ist  $p_{ik} = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k}$ .

Die erwarteten Kosten sind dann

$$\begin{aligned}\sum_k p_{ik}(1+k) &= \sum_k p_{ik} + \sum_k k \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k} \\ &= \dots \\ &= 1 + \frac{n}{m} \left(\frac{1}{m} + \left(1 - \frac{1}{m}\right)\right)^{n-1} \\ &= 1 + \frac{n}{m} = 1 + \beta\end{aligned}$$



**Also:** Im Mittel sind die Listen alle kurz.

Wie lang ist die längste Liste (erwartet)?

# Erwartete Länge der längsten Liste:

**Idee:** Wir wählen  $S$  zufällig aus  $U$ . Also

$$\text{prob}(h(x_k) = i) = \frac{1}{m} \text{ für } k \in S, i \in [0, \dots, m-1]$$

## Schritte:

1. Schätze ab, ob Liste  $i$  die Länge  $\geq j$  hat.
2. Schätze ab, ob die längste Liste die Länge  $\geq j$  hat
3. Bilde Erwartungswert der längsten Liste  $L$ .
4. Schätze ihn ab:  
 $\Rightarrow E(L) = O\left(\frac{\log n}{\log \log n}\right).$

**Wahl von  $\beta$ :** Wir wollen Laufzeit  $O(1)$ !

# Wahl von $\beta = n/m$

- Ist  $\beta \leq 1$ , ist die Laufzeit gut!
- Ist  $\beta \geq 1/4$ , ist die Platzausnutzung gut!
- Alles andere ist schlecht!  
→ schlechte Laufzeit oder schlechte Platzausnutzung

**Problem:** Durch Einfügen/Streichen verändert sich  $\beta$ .  
Vielleicht schnell zu klein oder zu gross.

**Rehashen!!!**

# Rehashing

- Benutze Folge von Hashtafeln  $T_0, T_1, \dots$  der Größe  $m, 2m, 4m, \dots$
- Ist  $\beta = 1$  bei Tafel  $T_i$  der Größe  $2^i m$ , kopiere Elemente in Tafel  $T_{i+1}$  der doppelten Größe  
→  $\beta \text{ nun} = 1/2$
- Ist  $\beta \leq 1/4$ , kopiere Elemente von  $T_i$  in  $T_{i-1}$   
→  $\beta \text{ nun} = 1/2$

**Beachte:** Nach Rehashen hat man eine Zeitlang Ruhe.  
Rehashen ist teuer! Danach kommen aber viele billige Operationen

→ **amortisierte Analyse!**

→  $n$  Operationen gehen erwartet in Zeit  $O(n)$

# Rehashing genauer: Nur Einfügungen

Nehmen an, wir sind grad von  $T_{i-1}$  zu  $T_i$  übergegangen und gehen nun von Tafel  $T_i$  der Größe  $2^i m$  zu Tafel  $T_{i+1}$ :

Also  $T_i$  hat  $\beta = 1/2$ .

Dazu fügen wir  $m2^i/2$  neue Elemente ein, bevor von  $\beta = 1/2$  auf  $\beta = 1$  steigt.

Also kostet Rehashen  $O(m2^{i+1})$  (teuer), aber dazwischen gibt es immer  $m2^{i-1} = O(m2^i)$  Operationen, die  $O(1)$  kosten, d.h. insgesamt kosten diese  $m2^{i-1} + 1$  Operationen bis inklusive der nächsten Rehashing-Operation dann  $O(m2^i)$ , also Linearzeit.

Rehashing hat Laufzeit maximal verdoppelt/vervierfacht.  
Das war amortisierte Analyse (siehe später).

# Offene Adressierung

**Ziel:** Alle Tafeleinträge werden höchstens einmal belegt, keine Listen!

Ist beim Einfügen ein Feld schon belegt, so probiere ein anderes:

→ Brauchen also eine Folge von Hashfunktionen

→ Funktioniert die erste nicht, nimm die zweite!

**Beispiele:**

- $h_i = (h(x) + i) \bmod m$

**Linear probing:** Probiere einfach das nächste Feld

- $h_i(x) = (h(x) + c_1 i + c_2 i^2) \bmod m$

**Quadratic probing**

Beispiel zu Linear Probing

# Linear Probing

**Prinzip:** Für Element  $k$ , berechne Hashwert  $h(k)$ . Versuche  $k$  dort abzulegen. Falls schon belegt, versuche  $h(k) + 1, h(k) + 2, \dots$  bis ein freies Feld gefunden ist.

**Invariante:** Falls  $k$  auf Feld  $h(k) + x$  gespeichert ist, sind alle Felder  $h(k), h(k) + 1, \dots, h(k) + x$  belegt.

**Zugriff auf Element  $k$ :** Springe zu Position  $h(k)$ , laufe los, bis entweder  $k$  gefunden oder freies Feld gefunden (dann gibt es  $k$  nicht).

**Wie geht Löschen ?** ... ohne Invariante zu verletzen

# Linear Probing - Löschen eines Elements

**1. Lösung:** Ersetze das Element durch Spezialsymbol, und behandle das Feld jeweils, also ob es belegt sei.

Nachteil: Hashtafel wird immer voller !

Ausweg: Reorganisiere die gesamte Menge von Zeit zu Zeit ohne die Spezialsymbole

**2. Lösung:** Nach dem Löschen teste die Einträge nach dem gelöschten Element. Wenn sie nicht korrekt stehen (Test mit Hashwert), dann schiebe sie um 1 nach links.

Höre auf, wenn ein Eintrag  $k'$  gefunden ist, der an der richtigen Stelle steht, also an  $h(k')$ .



## **Satz (Offene Adressierung)**

Unter der Annahme, dass alle Hashwerte gleichwahrscheinlich vorkommen, und dass  $\beta$  der Belegungsfaktor der Hashtafel angibt, gilt dass Einfügen/Streichen höchstens  $1/(1 - \beta)$  Schritte kosten in Erwarten.

Ein paar Werte eingesetzt:

$\beta = 0.5$  bedeutet  $1/(1 - \beta) = 2$

$\beta = 0.9$  bedeutet  $1/(1 - \beta) = 10$

**Merke:** Ist die Tashtafel relativ leer, ist offene Adressierung ne gute Idee. Falls zu voll, geht die Performanz in die Knie !

# Schlussbemerkungen:

Es gibt keine Hashfunktion, die für alle Mengen von Schlüsseln am besten funktioniert !

Wenn man die Menge der Schlüssel kennt:

1. Universelles Hashing: Für eine gegebene Datenmenge funktionieren die 'meisten' Hashfunktionen gut. Wähle eine zufällig !

2. Perfektes Hashing: Gegeben Menge von festen Schlüsseln, so kann man eine kollisionsfreie (injektive) Hashfunktion konstruieren.

Siehe dazu weiterführende VL über Algorithmen oder Bücher!

# Algorithmen

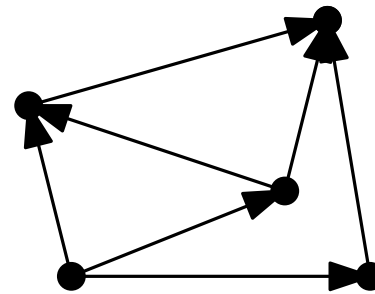
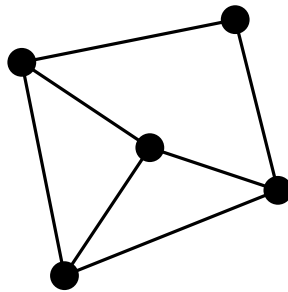
Michael Kaufmann

24/11/2020 – 7. Vorlesung  
Graphenalgorithmen -Grundbegriffe

### **III. Graphenalgorithmen**

# Grundbegriffe

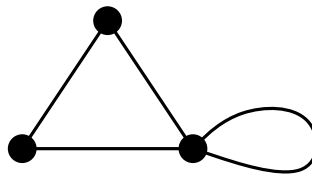
Ein Graph  $G = (V, E)$  besteht aus einer Menge von Knoten  $V$  und einer Menge von Kanten  $E \subset V \times V$ .



- Kanten können ungerichtet oder gerichtet sein. Zugehörige Notation  $\{u, v\}$  oder  $(u, v)$
- $u$  heißt adjazent (benachbart) zu  $v$ , falls es eine Kante zwischen  $u$  und  $v$  gibt. Die Kante  $\{u, v\}$  ist inzident zu  $u$  und  $v$ .

# Grundbegriffe

- In einem ungerichteten Graph schreiben wir  $u \sim v$ , falls  $u$  adjazent zu  $v$  ist
- In einem gerichteten Graph schreiben wir  $u \rightarrow v$ , falls es Kante  $(u, v) \in E$  gibt
- Sind Gewichte oder Kosten auf den Kanten, so hat jede Kante  $(u, v)$  ein Gewicht  $w(u, v)$  oder Kosten  $c(u, v)$ .
- Manchmal sehen wir ungewichtete Graphen als speziell gewichtete Graphen an, wo alle Kantengewichte entweder  $= 0$  (keine Kante) oder  $= 1$  (Kante existiert) sind.
- Eine Kante  $e = (u, v)$  ist eine Selbstschleife, wenn  $u = v$ .



# Grundbegriffe

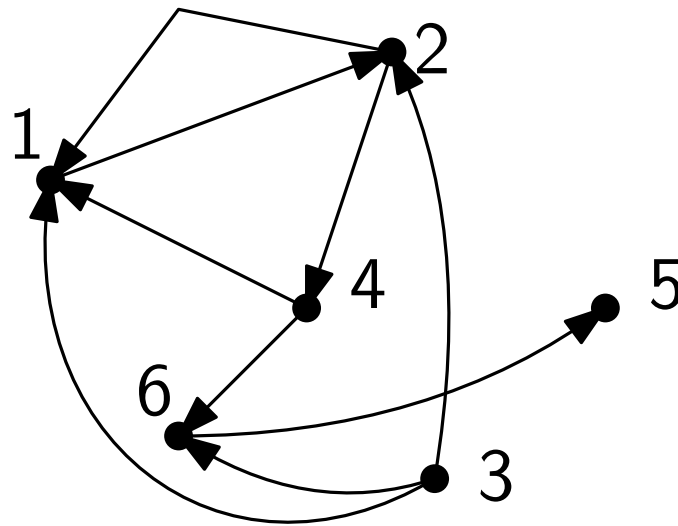
Ausgangsgrad eines Knoten  $v$  sind die Zahl seiner ausgehenden Kanten,

Formal:  $outdeg(v) = |\{w \in V \mid (v, w) \in E\}|$

Analog  $indeg(v) = |\{w \mid (w, v) \in E\}|$  Eingangsgrad

sowie  $deg(v) = indeg(v) + outdeg(v)$  bzw.

$= |\{w \mid \{v, w\} \in E\}|$  Grad von  $v$  (ungerichtet)



$$outdeg(2) = 2$$

$$indeg(1) = 3$$

$$deg(3) = 3$$

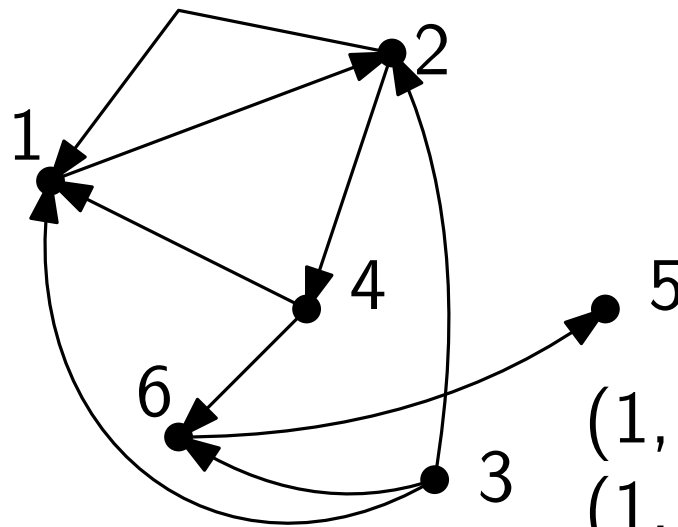
# Grundbegriffe

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

Eine Folge  $(v_0, v_1, \dots, v_k)$  heißt **Pfad**, falls für alle  $0 \leq i \leq k - 1$  Kanten  $(v_i, v_{i+1})$  existieren.

Der Pfad startet in  $v_0$ , endet in  $v_k$ .

Ein Pfad  $(v_0, v_1, \dots, v_k)$  heißt **Zykel**, falls  $v_k = v_0$ .



$(1, 2, 4, 6, 5)$  ist Pfad.

$(1, 2, 4, 1)$  ist ein Zykel

Kein Pfad von 1 nach 3

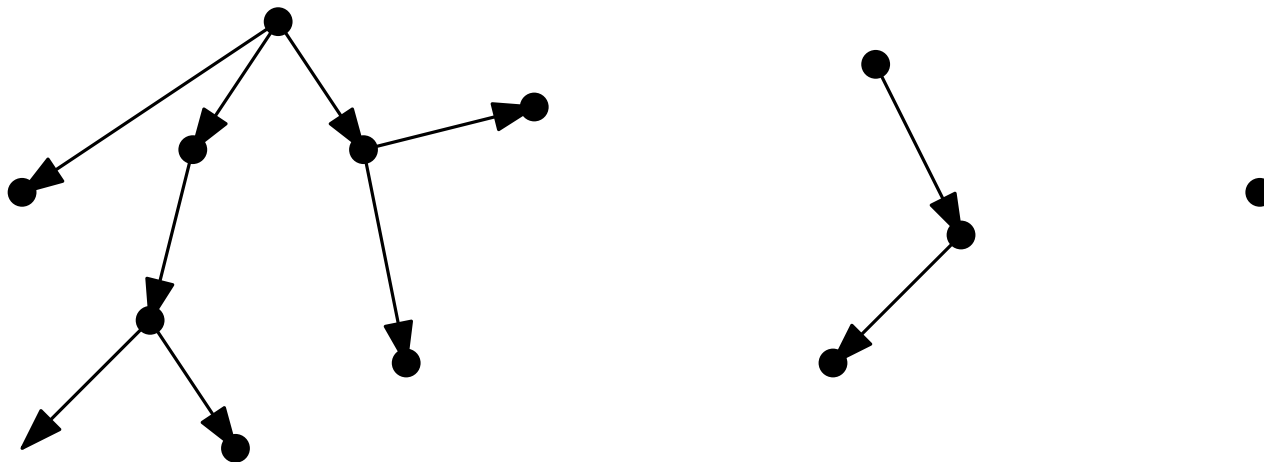


# Grundlegendes

Graph  $G = (V, E)$  heißt **Baum**, falls

- a)  $V$  enthält genau ein  $v_0$  mit  $\text{indeg}(v_0) = 0$
- b) Für alle  $v \in V \setminus \{v_0\} : \text{indeg}(v) = 1$
- c)  $G$  ist azyklisch (ohne Zykel)

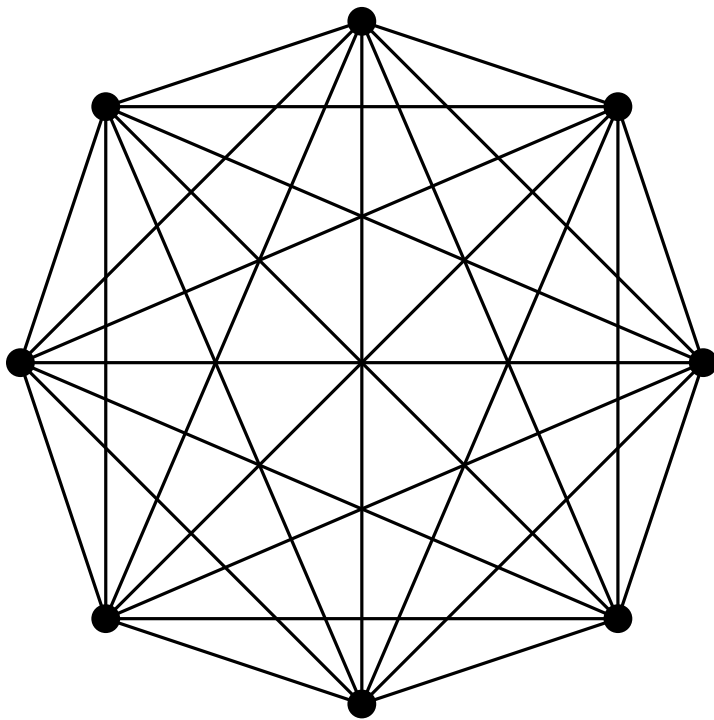
$G = (V, E)$  heißt **Wald**, falls  $G$  aus mehreren disjunkten Bäumen besteht. ( $G = G_1 \cup \dots \cup G_k$ , jedes  $G_i$  ist Baum)



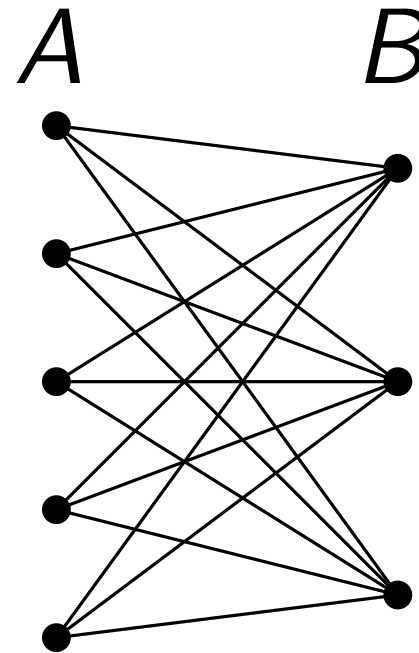
# Grundlegendes

## Vollständige Graphen

sind Graphen, für die für beliebige Knoten  $u, v \in V$  gilt:  
Kante  $\{u, v\} \in E$



$K_8$ : der vollst. Graph  
mit 8 Knoten

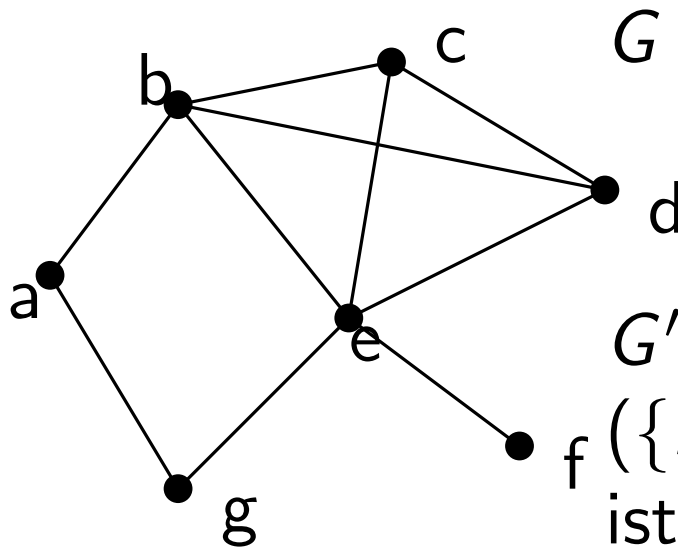


$K_{5,3}$ : der vollst. bipartite Graph  
mit 5 und 3 Knoten

# Noch mehr Begriffe

Ein Graph  $G = (V, E)$  heißt **bipartit**, wenn  $V = A \cup B$  und  $\emptyset \neq A \neq V$  und für alle Kanten  $u, v$  gilt: Falls  $u \in A$ , dann  $v \in B$ , sowie falls  $u \in B$ , dann  $v \in A$ .

Für einen Graph  $G = (V, E)$  heißt ein Teilgraph  $G' = (V', E')$  **induziert**, falls  $V' \subseteq V$  und für alle Kanten  $\{u, v\} \in E$  mit  $u, v \in V'$  gilt:  $\{u, v\} \in E'$



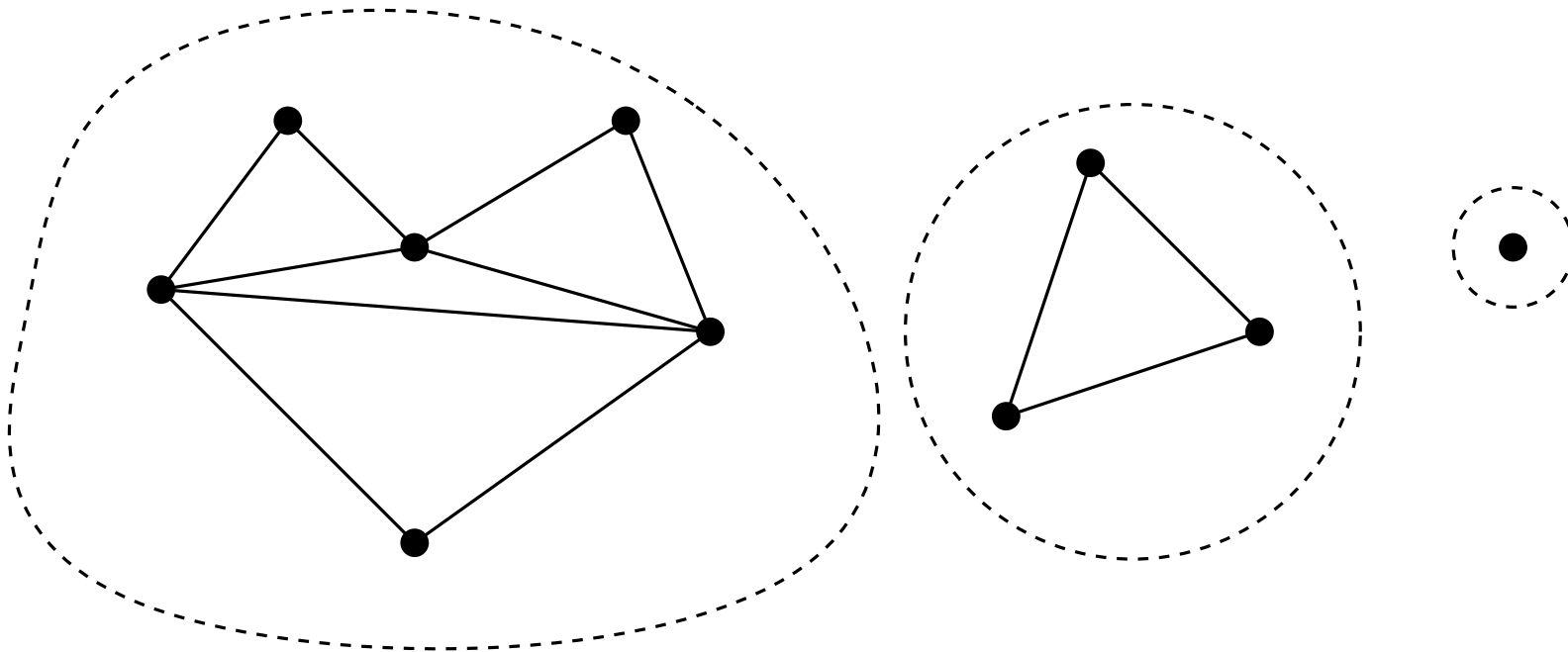
$G' =$

$(\{b, c, d, e\}, \{\{b, c\}, \{c, d\}, \{d, e\}, \{e, b\}\})$   
ist NICHT induziert für  $G$ .

## Noch mehr Begriffe (2)

Ein ungerichteter Graph  $(V, E)$  heißt **zusammenhängend**, falls es zwischen zwei beliebigen Knoten  $u, v \in V$  einen Pfad gibt mit den Endpunkten  $u$  und  $v$ .

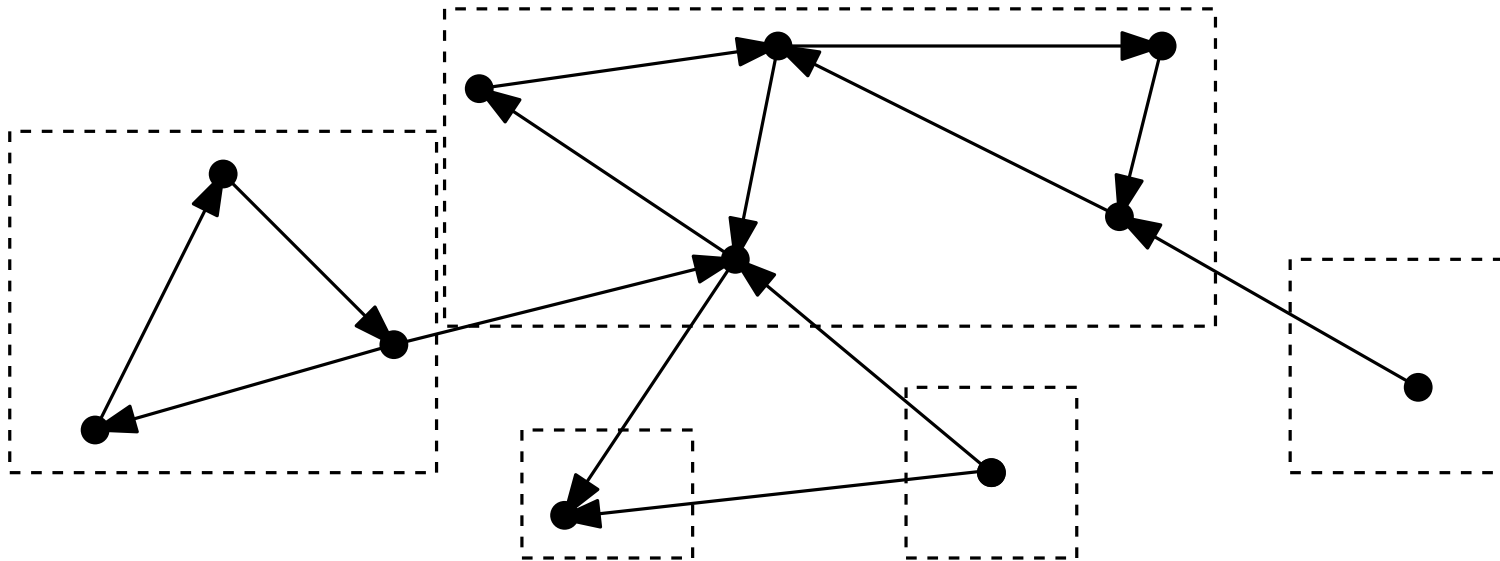
Eine **Zusammenhangskomponente** (ZK) eines ungerichteten Graphen  $G$  ist ein maximaler zusammenhängender Teilgraph von  $G$ .



# Noch mehr Begriffe (3)

Ein **gerichteter** Graph  $G = (V, E)$  heißt **stark zusammenhängend**, falls für alle Knoten  $u \neq v \in V$  gilt: Es gibt einen Pfad von  $u$  nach  $v$  UND es gibt einen Pfad von  $v$  nach  $u$ .

Eine **starke Zusammenhangskomponente** (SZK) ist ein maximaler stark zusammenhängender Teilgraph eines gerichteten Graphen  $G$

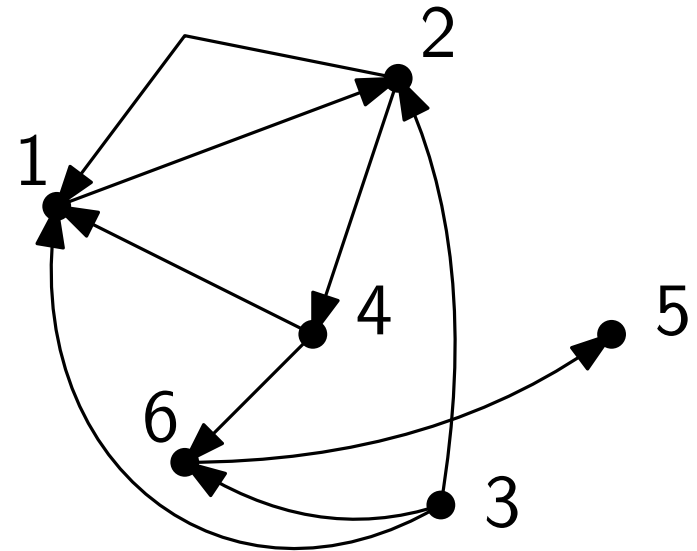


# Darstellung

## 1. Adjazenzmatrix $A = (a_{i,j})$

$$a_{i,j} = 1 \text{ falls } (i,j) \in E \\ = 0 \text{ sonst}$$

$$\begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$



Bei ungerichteten Graphen existieren beide Richtungen jeder Kante.

→ Matrix ist symmetrisch.

# Notationen und Adjazenzmatrix

Sei  $|V| = n$  und  $|E| = m$ .

**Platzbedarf:**  $O(n^2)$ .

**Zugriffszeit:**  $O(1)$ .

Platz eventuell nicht effizient. Insbesondere bei 'dünnen' Graphen, wo  $m = O(n)$ .

## Typische Fragen:

- Welcher Knoten hat die meisten eingehenden Kanten?  
(beliebteste Person)
- Welches Knotenpaar ist am weitesten auseinander?  
( 'Durchmesser' )
- Welcher Knoten ist 'zentral' ?

## 2. Adjazenzlisten

Speichern für jeden Knoten  $v$  die Nachbarknoten.

**Falls  $G$  gerichtet:**

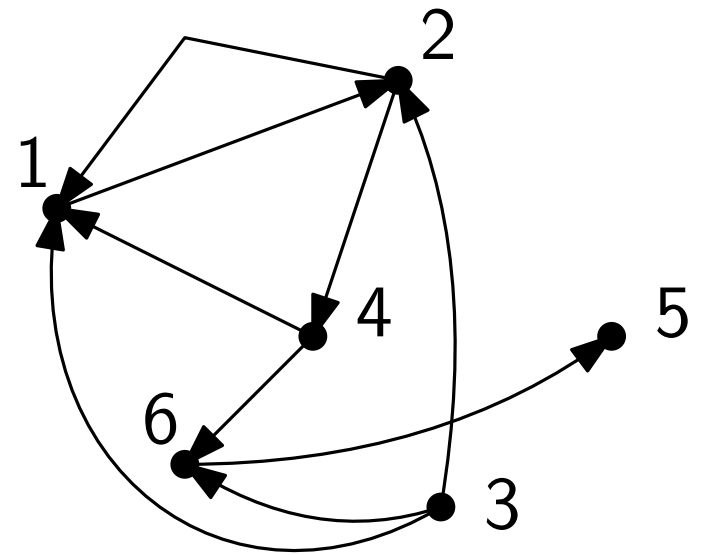
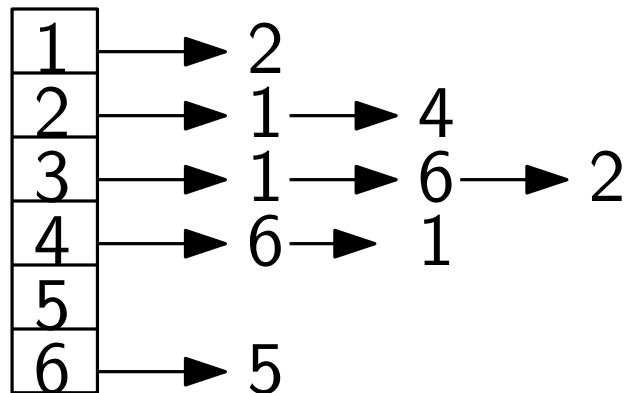
$$InAdj(v) = \{w \in V \mid (w, v) \in E\}$$

$$OutAdj(v) = \{w \in V \mid (v, w) \in E\}$$

**Falls  $G$  ungerichtet:**

$$Adj(v) = \{w \in V \mid \{v, w\} \in E\}$$

**Bsp:**  $OutAdj$  als Adjazenzliste



**Platz:**  $O(n + m)$



# Adjazenzlisten

Zugriff auf Kante  $(v, w)$  in  $O(outdeg(v))$ ,  
wobei  $outdeg(v) = |\{w \in V \mid (v, w) \in E\}|$

**Platzbedarf:**  $O(n + m)$ .

**Zugriffszeit:**  $O(outdeg(v))$ .

Platz sehr effizient. Insbesondere bei 'dünnen' Graphen,  
wo  $m = O(n)$  viel besser als Matrixdarstellung. Zugriff  
evtl. schlechter...

**Viele Algorithmen auf Adjazenzlistendarstellung  
ausgelegt.**

# Algorithmus: Topologisches Sortieren

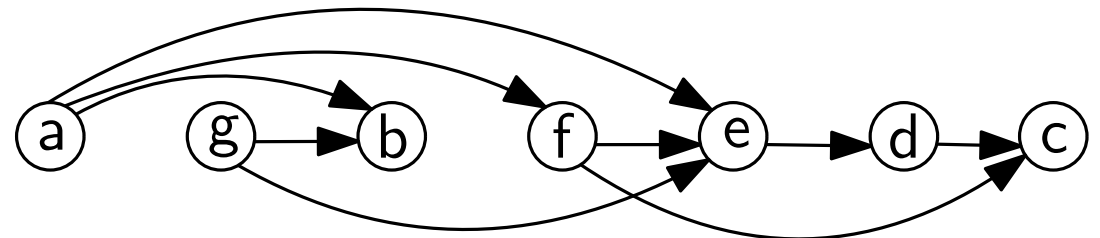
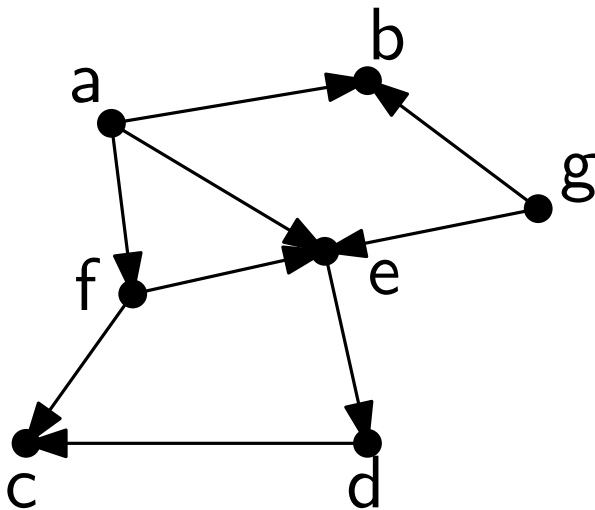
Sei  $G = (V, E)$  ein gerichteter Graph (DAG).

Eine Abbildung

$$num : V \rightarrow \{1, 2, \dots, n\}$$

mit  $n = |V|$  heißt topologische Sortierung, falls für alle  $(v, w) \in E$  gilt:

$$num(v) < num(w).$$



Pfeile alle nach rechts gerichtet.

# Topologisches Sortieren

## Lemma:

Graph  $G$  besitzt genau dann eine topologische Sortierung, wenn  $G$  azyklisch ist.

## Beweis:

' $\Rightarrow$ ': Annahme:  $G$  zyklisch. Dann sei  $(v_0, \dots, v_k)$  ein Zykel mit  $v_0 = v_k$ . Es muss gelten:

$$num(v_0) < num(v_1) < \dots < num(v_k) = num(v_0) \quad \text{Wid.!}$$

' $\Leftarrow$ ': Sei  $G$  azyklisch.

**Beh.:**  $G$  enthält Knoten  $v$  mit  $indeg(v) = 0$ .

$\rightarrow v$  kriegt die  $num(v) = 1$ . Lösche  $v$  und dann induktiv

**Bew.:** Starte bei belieb.  $w$ . Laufe eingeh. Kanten rückwärts. Nach spätestens  $n - 1$  Schritten ist  $v$  gefunden (oder Zykel)

# Algorithmus *TopSort*

**TopSort** $((V, E), i)$

**If**  $(|V| = 1)$

$num(v) \leftarrow i$  für  $v \in V$

**If**  $(|V| > 1)$  {

$v \leftarrow$  Knoten aus  $V$  mit indegree 0

$num(v) \leftarrow i$

**TopSort** $((V \setminus \{v\}, E \setminus \{(v, w) \mid (v, w) \in E\}), i + 1)$

}

# Algorithmus *TopSort*

```
count  $\leftarrow$  0
while ( $\exists v \in V$  mit  $\text{indeg}(v) = 0$ ) {
    count++
    num(v)  $\leftarrow$  count
    streiche v mit ausgehenden Kanten
}
if (count <  $|V|$ )
    return 'G zyklisch'
```

**Ziel:** Laufzeit  $O(n + m)$

# Implementierungsfragen

## 1. benutzen Adjazenzlisten: Platz $O(n + m)$

- Lösche  $v$  und Kanten mit Durchlaufen von  $OutAdj(v)$  in  $O(outdeg(v))$
- insgesamt Laufzeit  $\sum_v O(outdeg(v)) = O(n + m)$

## 2. Finden geeignetes $v$ :

- Benutze `inZaehler[]` für InDegrees und Menge ZERO
- Wird  $(v, w)$  gelöscht, dekrementiere `inZaehler[w]`.  
Nimm eventuell  $w$  in ZERO auf
- Geeignetes  $v$  wird in ZERO gezogen und gelöscht  $\rightarrow$  ZERO als Stack  $\rightarrow$  Laufzeit  $O(1)$

## 3. Initialisierung von `inZaehler[]` und ZERO:

- Durchlaufe Adj.listen. Erhöhe `inZaehler[w]` für Kante  $(v, w)$
- Initialisierung ZERO ?
- Insgesamt Laufzeit  $O(n + m)$

# TopSort

## Satz:

Gegeben ein gerichteter Graph  $G = (V, E)$ .

In Zeit  $O(n + m)$  kann festgestellt werden, ob  $G$  einen Zykel hat, und wenn nicht, kann  $G$  topologisch sortiert werden,

# Algorithmen

Michael Kaufmann

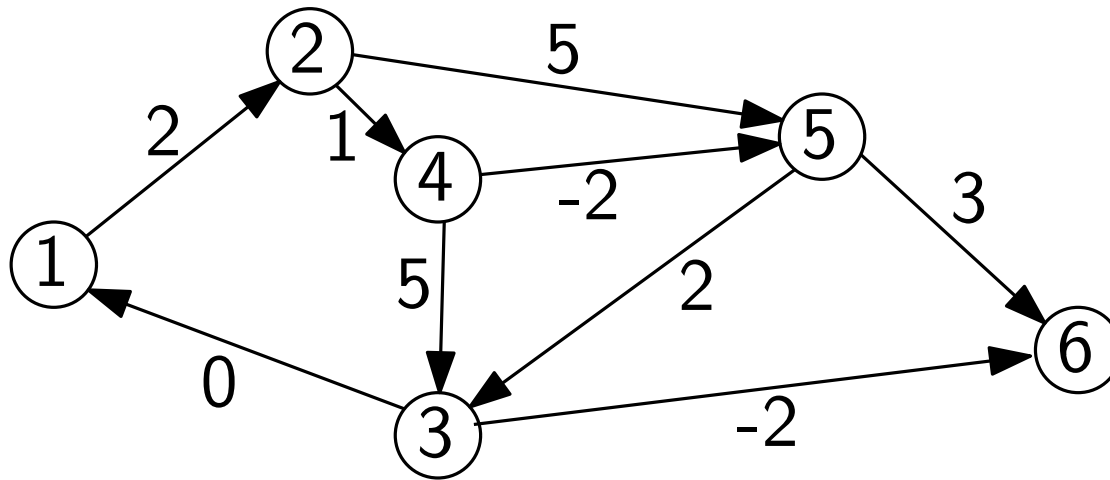
25/11/2020 und 2/12/2020  
III.b Kürzeste (billigste) Wege



# Definitionen

Gegeben sei Graph  $G = (V, E)$  sowie **Kantenkosten**  $c: E \rightarrow \mathbb{R}$ .

**Name:** **Netzwerk**  $(V, E, c)$ .



Kosten des Pfades  $c(v_0, v_1, \dots, v_k) := \sum_{i=1}^k c(v_{i-1}, v_i)$

Betrachten Varianten 'single source shortest paths' sowie 'all pairs shortest paths', zuerst SSSP.

# Single Source Shortest Paths

Sei  $s \in V$  der Startpunkt der Pfade (**Quelle/Source**).

Für  $u \in V$  sei  $P(s, u)$  die Menge aller Pfade von  $s$  nach  $u$ .

$$\text{Sei } \delta(u) := \begin{cases} \infty & \text{falls } P(s, u) = \emptyset \\ \inf \{c(p) \mid p \in P(s, u)\} & \text{sonst} \end{cases}$$

Ein zyklischer Pfad  $p$  mit  $c(p) < 0$  heißt '**negativer Zykel**'.

## **Lemma:**

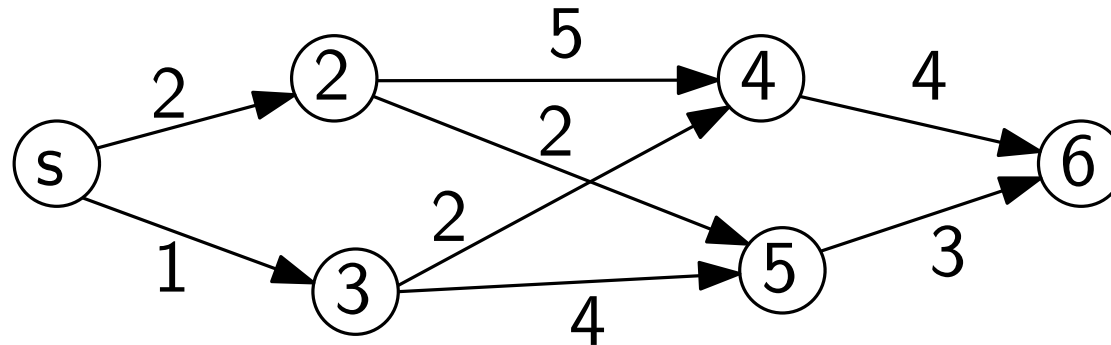
Sei  $u \in V$ .

a)  $\delta(u) = -\infty \Leftrightarrow u$  erreichbar von negativem Zykel,  
der von  $s$  aus erreichbar ist.

b)  $\delta(u) \in \mathbb{R} \Rightarrow \exists$  billigsten Weg von  $s$  nach  $u$  mit  
Kosten  $\delta(u)$ .

# 1. Graph ist DAG.

DAG  $\rightarrow$  topologische Sortierung (wird angenommen)



## Algorithmus:

Distanzen  $d(s) \leftarrow 0$ ; Pfad( $s$ )  $\leftarrow (s)$ ;

**for all** ( $v \in V \setminus \{s\}$ ) {  $d(v) \leftarrow \infty$ ; }

**for** ( $v \leftarrow s + 1$  to  $n$ ) {

$d(v) \leftarrow \min_{u \geq s} \{d(u) + c(u, v) \mid (u, v) \in E\}$ ;

Pfad( $v$ )  $\leftarrow \text{concat}(\text{Pfad}(u), v)$ ;

}

# DAG (Korrektheit, Laufzeit)

**Lemma 1:** Nach Ausführung des Algorithmus gilt  
 $d(v) = \delta(v)$  für alle  $v \in V$

**Beweis mit Fallunterscheidung:**

$v < s$ :  $\delta(v) = \infty = d(v)$

$v \geq s$ : Induktion über  $i$ :

**IA:**  $i = 0$ :  $\delta(s + i) = \delta(s) = 0 = d(s) = d(s + i)$

**IV:**  $\delta(s + j) = d(s + j)$  für alle  $j < i$

**IS:**  $\delta(s + i) = \min_{v \in V \setminus \{s+i\}} \{\delta(v) + c(v, s + i)\}$

$$= \min_{v < s+i} \{\delta(v) + c(v, s + i)\}$$

$$\stackrel{\text{IV}}{=} \min_{v < s+i} \{d(v) + c(v, s + i)\}$$

$$= d(s + i)$$

# DAG (Korrektheit, Laufzeit)

**Lemma 2:** Nach Ausführung des Algorithmus gilt:  
 $d(v) < \infty \Rightarrow \text{Pfad}(v)$  ist billigster Weg von  $s$  nach  $v$

**Beweis:**

Aus Lemma 1 folgt, dass  $d(v) = \delta(v) \forall v \in V$ .

Außerdem  $c(\text{Pfad}(v)) = d(v)$ . Also  $c(\text{Pfad}(v)) = \delta(v)$

**Laufzeit:**  $\sum_{v \in V} |\text{InAdj}(v)| = \sum_{v \in V} \text{indeg}(v) = O(m)$

Konkatenation jeweils in  $O(1)$ .

insgesamt:  $d$ -Werte in  $O(n + m)$  und Pfade in  $O(n)$ .

Ab sofort berechnen wir nur  $d$ -Werte. Pfade analog.

# SSSP auf DAGs

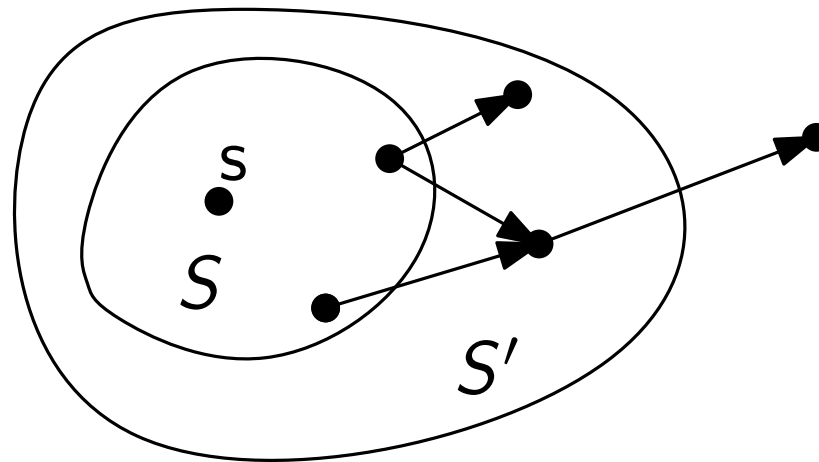
## **Satz:**

Distanzen von Single Source Shortest Paths können auf azyklischen Netzwerken in Zeit  $O(n + m)$  berechnet werden.

## 2. nichtneg. Kantenkosten ( $c(e) \geq 0 \forall e \in E$ )

**Name:** Dijkstra Algorithmus

- Starte an  $s$ . Halte Mengen  $S = \{v \in V \mid d(v) = \delta(v)\}$ , sowie  $S' = \{v \in V \setminus S \mid v \text{ hat Nachbar in } S\}$
- Vergrößere  $S$  sukzessive, bis  $S = V$



# Dijkstra Algorithmus

```
 $S \leftarrow \{s\}; d(s), d'(s) \leftarrow 0;$   
 $S' \leftarrow \text{OutAdj}(s); \text{for all } u \in S' \{d'(u) = c(s, u); \}$   
for all  $(u \in V \setminus (S' \cup \{s\})) \{d'(u) \leftarrow \infty; \}$   
  
while  $(S \neq V) \{$   
    wähle  $w \in S'$  geeignet // ('findmin')  
     $d(w) \leftarrow d'(w);$   
     $S \leftarrow S \cup \{w\}; S' \leftarrow S' \setminus \{w\};$   
  
    for all  $(u \in \text{OutAdj}(w)) \{$   
        if  $(u \notin (S \cup S')) \{S' \leftarrow S' \cup \{u\}; \}$   
         $d'(u) \leftarrow \min\{d'(u), d'(w) + c(w, u)\}$   
     $\}$   
 $\}$ 
```

$d'$  sind Distanzzwischenwerte



# Dijkstra Algorithmus: Korrektheit

## **Lemma:**

Wird  $w \in S'$  gewählt, dass  $d'(w)$  minimal, ist  $d'(w) = \delta(w)$ .

## **Beweis:**

Sei  $p = (s, s_1, \dots, s_k, w)$  billigster Weg von  $s$  nach  $w$  mit  $s_i \in S$ . Nimm an, es gäbe einen billigeren Weg  $q$  von  $s$  nach  $w$ .  $q$  hat ersten Knoten  $v$  in  $V \setminus S$ .

Es gilt:  $d'(v) \geq d'(w)$ .

Da alle Kantenkosten nichtnegativ, gilt:

$$c(q) \geq d'(v) \geq d'(w) = c(p).$$

Wid.

**Fazit:** Wir können uns also wirklich auf die Menge  $S'$  beschränken.

# Dijkstra Algorithmus: Laufzeit

**Wie implementieren wir  $S$  und  $S'$ ,  $d$  und  $d'$ ?**

**Antwort:**  $S$  und  $S'$  als Bitvektoren der Länge  $|V|$ ,  $d, d'$  als Integervektoren

'**for all** ( $u \in \text{OutAdj}(w)$ )': Durchlaufen der Adjazenzliste in  $O(\sum_w \text{OutDeg}(w)) = O(n + m)$

$n \times$  Minimumssuche in  $S'$ : insgesamt  $O(n^2)$

$\Rightarrow$  **Laufzeit:**  $O(n^2 + m)$ , ok für dichte Graphen.

**Verbesserung:**

Minimumssuche geht in  $O(\log n)$  jeweils mit  $S'$  als Heap.  
Einfügen, Minimumssuchen/Löschen in  $O(\log n)$

$d'$ -Werte müssen verringert werden! Decrease-Key! Geht!

**Laufzeit:**  $O(n \log n + m \log n)$

# Dijkstra Algorithmus

## **Satz:**

Single Source Shortest Paths können auf Netzwerken mit nichtnegativen Kantenkosten in Zeit  $O((n + m) \log n)$  berechnet werden.

**Bemerkung:** Es geht auch in  $O(n \log n + m)$ .

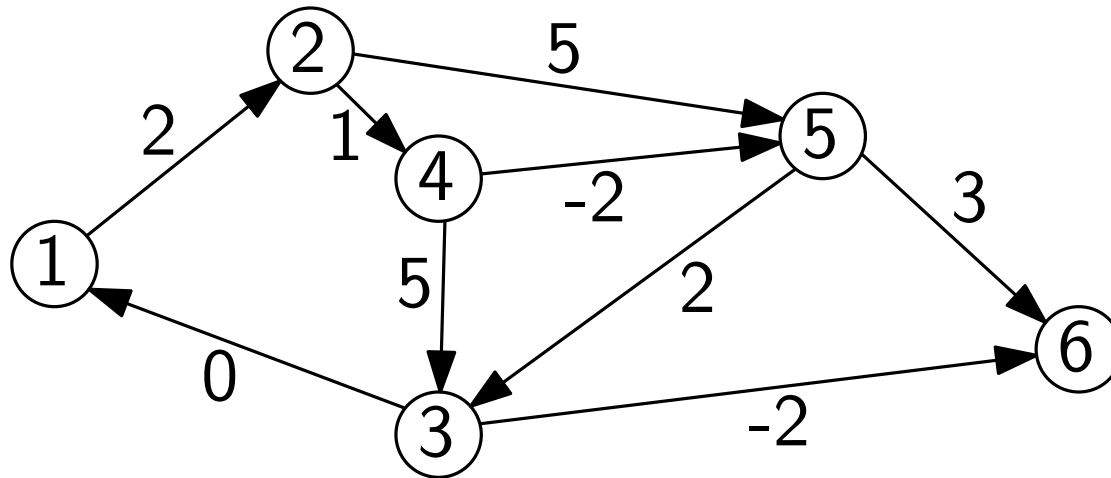
# Billigste Wege II

Michael Kaufmann

02/12/2020

# Rückblick:

Definition, negative Zykel, DAGs, Dijkstra



### 3. Neg. Kantenkosten erlaubt, keine Zykel

**Name:** Bellman/Ford - Algorithmus

**Idee:** benutze iterativ Operation  $Relax(v, w)$  durch  
$$d(w) \leftarrow \min\{d(w), d(v) + c(v, w)\}$$

**Beobachtung:**

- Relax-Operation erhöht keine  $d$ -Werte.
- Auch nach Relax-Operation gilt  $d(v) \geq \delta(v)$ , wenn es vorher galt.

**Algorithmus:**

- Sei  $d(v) = 0$ , falls  $v = s$ , und  $d(v) = \infty$  sonst, für alle  $v$
- **Iteriere:** Relaxiere Kanten

$d$ -Werte erniedrigen sich, bis  $\delta$ 's erreicht.

**Frage:** Wie schnell? Reihenfolge der Relaxierungen?

# Bellman-Ford Algorithmus

## Lemma:

Sei  $w \in V$  mit  $\delta(w) < \infty$ . Sei  $(v, w)$  die letzte Kante auf billigstem Pfad von  $s$  nach  $w$ . Dann gilt:  
Falls  $(v, w)$  relaxiert wird, und vorher  $d(v) = \delta(v)$ , ist danach  $d(w) = \delta(w)$ .

## Algorithmus:

```
 $d(s) \leftarrow 0$   
for all  $(v \neq s) \{d(v) \leftarrow \infty;\}$   
for  $(i \leftarrow 1 \text{ to } n - 1) \{$   
    for all  $((v, w) \in E) \{$   
        Relax $(v, w)$   
     $\}$   
 $\}$   
 $\}$ 
```

# Bellman-Ford Algorithmus

## Lemma:

Für  $i = 0, \dots, n - 1$  gilt: Nach Phase  $i$  ist  $d(w) = \delta(w)$  für alle  $w \in V$ , für die es einen billigsten Pfad von  $s$  nach  $w$  der Länge  $i$  gibt.

## Beweis: Per Induktion

$i = 0$ :  $d(s) = \delta(s) = 0$  (kein neg. Zykel)

$i \leftarrow i + 1$ :  $w$  ist Knoten mit billigstem Weg der Länge  $i + 1$  mit letzter Kante  $(v, w)$ . Also gibt es billigsten Weg von  $s$  nach  $v$  der Länge  $i$ .

Nach Induktionsannahme gilt nach Phase  $i$ :  $d(v) = \delta(v)$ .

In Phase  $i + 1$  wird  $(v, w)$  relaxiert:

$$d(w) = \min\{d(v) + c(v, w)\} = \delta(v) + c(v, w) = \delta(w).$$



# Bellman-Ford Algorithmus

## **Satz:**

Nach Phase  $n - 1$  gilt für alle  $v \in V$ :  $d(v) = \delta(v)$ .

**Laufzeit:**  $O(n \cdot m)$  (Implementierung ganz einfach!)

## 4. Negative Zykel erlaubt

### Idee:

Bei negativem Zykel erniedrigen sich  $\delta$ -Werte immer weiter.

1. Führe zuerst  $n - 1$  Phasen von Bellman-Ford aus, merke letzte  $d$ -Werte ( $d_1$ ).
2. Führe weitere  $n$  Phasen aus. Ergibt neue  $d$ -Werte ( $d_2$ ).

**Lemma:** Sei  $w \in V$ .

$$d_2(w) = d_1(w) \Rightarrow d_1(w) = \delta(w).$$

$$d_2(w) < d_1(w) \Rightarrow \delta(w) = -\infty.$$

### Frage:

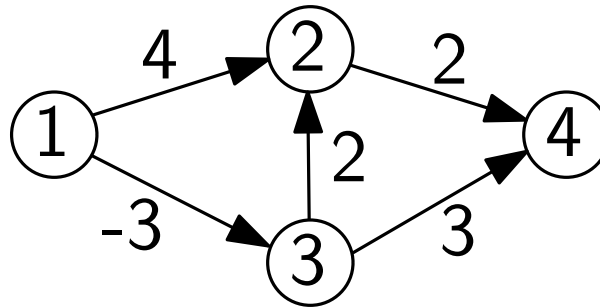
Warum im 2. Schritt  $n$  Iterationen und nicht bloss eine?

# All Pairs Shortest Paths (APSP)

**Name:** Floyd/Warshall

Sei  $(V, E, c)$  ein Netzwerk, ohne neg. Zykel und sei  $V = \{1, 2, \dots, n\}$ .

Wir definieren für Knoten  $i, j$  und  $0 \leq k \leq n$  die Kosten  $\delta_k(i, j)$  als die des billigsten Weges von  $i$  nach  $j$  mit inneren Knoten zwischen 0 und  $k$ .



Was ist  $\delta_0(1, 4)$ ?,  $\delta_2(1, 4)$ ?,  $\delta_4(1, 4)$ ?

$\delta_0(1, 4) = \infty$ ,  $\delta_2(1, 4) = 6$ ,  $\delta_4(1, 4) = 1$

# All Pairs Shortest Paths

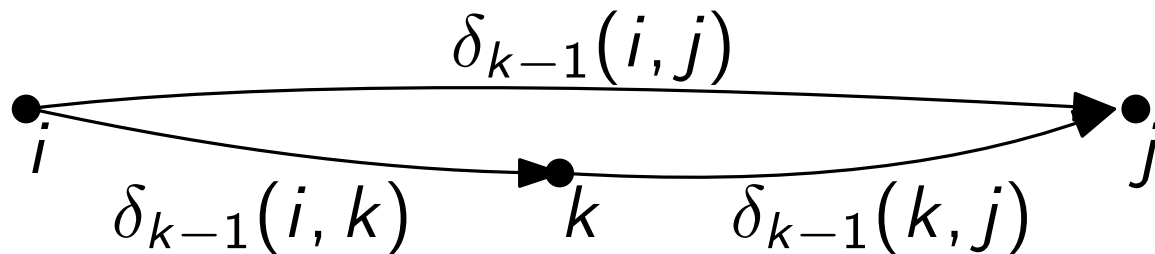
**Wollen  $\delta_k(i, j)$  berechnen:**

$$\text{Für } k = 0 \text{ initial: } \delta_0(i, j) = \begin{cases} c(i, j) & \text{falls } (i, j) \in E \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Für  $k = n$  gilt:  $\delta_n(i, j) = \delta(i, j)$  = Kosten des billigsten Weges von  $i$  nach  $j$

**Allgemein rekursiv:**

$$\delta_k(i, j) = \min\{\delta_{k-1}(i, j), \delta_{k-1}(i, k) + \delta_{k-1}(k, j)\}$$



# Floyd/Warshall's Algorithmus

```
for all  $(i \neq j \in V)$  {  
     $\delta_0(i, j) = c(i, j)$  falls  $(i, j) \in E$  und  $\delta_0(i, j) = \infty$  sonst  
}  
  
for all  $(i = j)$  {  
     $\delta_0(i, j) = 0$   
}  
  
for  $(k = 1 \text{ to } n)$  {  
    for all  $(i, j \in V)$  {  
         $\delta_k(i, j) = \min\{\delta_{k-1}(i, j), \delta_{k-1}(i, k) + \delta_{k-1}(k, j)\}$   
    }  
}
```

**Laufzeit:**  $O(n^3)$  (3-fache for-Schleife)

# Alternativer Algorithmus

1. **Mache  $n \times$  Bellman/Ford**, jeder Knoten ist 1x die Quelle  $s$   
**Laufzeit:**  $O(n \cdot n \cdot m)$

2. **Besser:**  $n \times$  Dijkstra  $\rightarrow$  Ziel:  $O(n \cdot (n \log n + m))$ .  
**Aber:** brauchen dazu nichtnegative Kantenkosten!

Es muss gelten für modifizierte Kosten  $c'$ :

**Lemma:**

Für alle  $u, v \in V$ : Pfad  $p$  zwischen  $u$  und  $v$  billigster Weg für originales  $c \Leftrightarrow p$  billigster Weg für Kosten  $c'$ .

# Wahl der Kantenkosten $c'$ :

Wählen  $c'(u, v) = c(u, v) + h(u) - h(v)$  für Kanten  $(u, v) \in E$ , und  $h : V \rightarrow R$ .

Für Pfad  $p = (v_0, \dots, v_k)$ :

$$\begin{aligned} c'(p) &= \sum_i c'(v_i, v_{i+1}) \\ &= \sum_i (c(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})) \\ &= c(p) + h(v_0) - h(v_k). \end{aligned}$$

Somit ist

$$c(p) = \delta(v_0, v_k) \Leftrightarrow c'(p) = \delta(v_0, v_k) + h(v_0) - h(v_k)$$

und für Zykel  $(v_0 = v_k)$ :  $c(p) = c'(p)$ .

Somit ist das Lemma ok.

Nun wähle  $h$ , so dass  $c' \geq 0$  für alle Kanten.

# Wahl von $c'$ :

## Wahl von $h$ :

- Erweitere  $G$  mit neuer Quelle  $s \notin V$  und Kanten  $(s, v)$  für alle  $v \in V$  mit  $c(s, v) = 0$ .
- Berechne SSSP von  $s$  aus und setze  $h(v) = \delta(s, v)$  für alle  $v \in V$ .

## Wieso ist das gut?

Dann gilt:  $h(v) \leq h(u) + c(u, v)$  für alle  $(u, v) \in E$ .

und somit  $0 \leq c(u, v) + h(u) - h(v) = c'(u, v)$ .

Berechne  $h$  mit  $1 \times$  Bellman/Ford in  $O(nm)$   
und dann  $n \times$  Dijkstra in  $O(n \cdot (n \log n + m))$ .



# Zusammenfassung: APSP

## **Satz:**

Das All-Pairs-Shortest-Paths Problem kann in Zeit  $O(n^2 \log n + nm)$  gelöst werden.

# Durchmusterung von Graphen

9+10/12/2020

# Durchmusterung genauer betrachtet !

Billigste Wege-Berechnung durchmustert Graph.  
Jetzt gehts ums Prinzip!

Gegeben  $G = (V, E)$ , sei  $s$  Startknoten,

sei  $S$  Menge der bekannten Knoten

$S \leftarrow \{s\};$

markiere alle Kanten als 'unbenutzt';

**while**  $(\exists v \in S \text{ mit unbenutzter Kante } (v, w) \in E) \{$

    sei  $e = (v, w)$  so eine Kante;

    markiere  $e$  als 'benutzt';

$S \leftarrow S \cup \{w\};$

$\}$

**Invariante:** Alle Knoten in  $S$  sind über benutzte Kanten von  $s$  aus erreichbar

# Durchmustern von Graph $G$

## Implementierungsfragen:

1. Wie realisiert man die Markierung 'benutzt'?
2. finde geeignete Kante  $(v, w)$ ?
3. Wie implementieren wir  $S$ ?

## Lösungen:

1. geht implizit:

benutze Adjazenzlisten. Durchlaufe die Kanten in Listenreihenfolge. Merke die Stelle, bis wohin die Kanten gesehen wurden. Danach kommen die 'unbenutzten'.

2. Merke in Zusatzmenge  $S' \subseteq S$  die Knoten, die noch unbenutzte Kanten haben.

3.  $S$  als bool'sches Array, da Operationen Init / Einfüge( $v$ ) / Frage  $w \in S$ ? gebraucht werden

# Durchmustern von $G$ : Fragen

## Was ist mit $S'$ ?

- Brauchen Operationen
  - Init
  - Test  $S' = \emptyset$
  - Einfüge( $w, S'$ )
  - wähle  $v \in S'$  beliebig
  - streiche gewähltes  $v \in S'$
- Implementieren  $S'$  als Stack oder als Queue.

## Name:

- Stack-Implementierung heisst **Tiefensuche** (DFS).
- Queue-Implementierung **Breitensuche** (BFS).

# Durchmustern von $G$ : Verfeinern

Gegeben  $G = (V, E)$ , Mengen  $S$  und  $S'$ . Start  $s$ .

$S \leftarrow S' \leftarrow s$ ;

**for all**  $(v \in V)$   $\{p(v) \leftarrow \text{adjlistenstart}(v)\}$

**while**  $(S' \neq \emptyset)$   $\{$

    sei  $v \in S'$  beliebig;

**if**  $(p(v)$  nicht am Listenende)  $\{$

$w \leftarrow p(v)$ ; verschiebe  $p(v)$ ;

**if**  $(w \notin S)$   $\{\text{einfüge}(w, S)$ ;  $\text{einfüge}(w, S')\}$

$\}$

**else** streiche  $(v, S')$

$\}$

**Name:** Explorefrom( $s$ )

**Laufzeit:** Operationen gehen jeweils in  $O(1)$   $\rightarrow$

# Explorefrom(s): Laufzeit

Laufzeit ist proportional zur Größe des erreichten Graphen  $G_s = (V_s, E_s)$ :

$$V_s = \{v \mid s \rightarrow^* v\} \text{ und } |V_s| = n_s$$

$$E_s = \{e = (v, w) \in E \mid v \text{ und } w \in V_s\} \text{ und } |E_s| = m_s.$$

**Name:**  $G_s$  ist der von  $V_s$  **induzierte Teilgraph** von  $G$

**Laufzeit:**  $O(n_s + m_s)$

# Explorefrom

Benutzen Explorefrom von allen noch nicht gesehenen Knoten aus:

## Pseudocode:

Initialisierung wie bisher;

```
for all ( $s \in V$ ) {  
    if ( $s \notin S$ ) then {Explorefrom( $s$ )}  
}
```

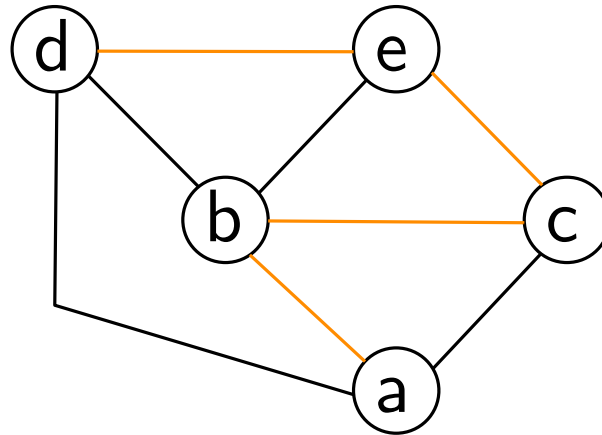
## Satz:

Ist der Graph  $G$  ungerichtet, so können in Zeit  $O(n + m)$  die Zusammenhangskomponenten berechnet werden.

**Frage:** Was waren nochmal Zusammenhangskomponenten ?



# Beispiel: Tiefensuche / Breitensuche



Betrachte Reihenfolge der betrachteten Knoten bei DFS(a) und bei BFS(a):

- **DFS**: a,b,c,e,d. mit Baum (a,b), (b,c), (c,e), (e,d)
- **BFS**: a,b,c,d,e mit Baum (a,b), (a,c), (a,d), (b,e)

**Vergleiche**: Reihenfolge bei Billigste-Wege-Algorithmen

# DFS rekursiv

$S'$  implizit als Aufrufkeller der Rekursion

*dfsnum, compnum* gibt Aufruf-/ Abschlussreihenfolge

→ **DFS**( $v$ ):

```
for all  $((v, w) \in E)$  {  
    if  $(w \notin S)$  {  
         $S \leftarrow S \cup \{w\}$   
         $dfsnum(w) \leftarrow count1; count1++$ ;  
        DFS( $w$ );  
         $compnum(w) \leftarrow count2; count2++$ ;  
        füge  $(v, w)$  zu  $T$  hinzu  
    }  
    else if  $(v \rightarrow_T^* w)$  {füge  $(v, w)$  zu  $F$ }  
    else if  $(w \rightarrow_T^* v)$  {füge  $(v, w)$  zu  $B$ }  
    else {füge  $(v, w)$  zu  $C$  }  
}
```

# DFS rekursiv

**main:**

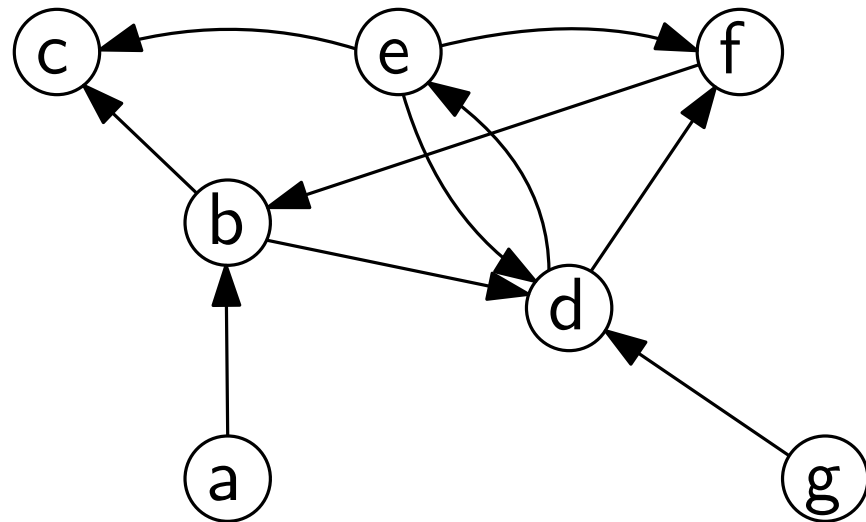
```
 $S \leftarrow \emptyset; count1 \leftarrow count2 \leftarrow 1;$   
for all ( $v \notin S$ ) {  
     $S \leftarrow S \cup \{v\};$   
     $dfsnum(v) \leftarrow count1; count1 ++;$   
    DFS( $v$ );  $compnum(v) \leftarrow count2; count2 ++;$   
}
```

$T$  Baumkanten

$F$  Vorwärtskanten

$B$  Rückwärtskanten

$C$  Querkanten



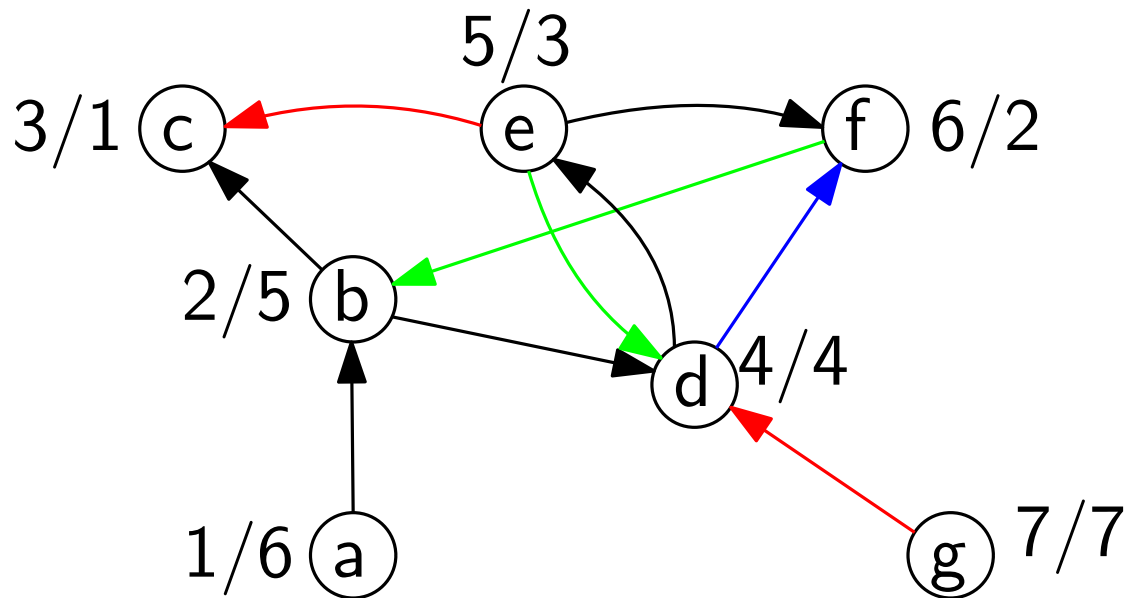
# DFS Kantenpartition

$T$  Baumkanten

$F$  Vorwärtskanten

$B$  Rückwärtskanten

$C$  Querkanten



**Aufruf: DFS( $a$ )**

*dfsnum/compnum*  
an den Knoten

# DFS Kantenpartition

**Lemma:** Sei  $G = (V, E)$  ein gerichteter Graph.

1. DFS auf Graph  $G$  hat Laufzeit  $O(n + m)$ .
2.  $T, F, B, C$  ist Partition der Kanten.
3.  $T$  entspricht dem Aufrufbaum der Rekursion.
4. Gibt es einen Pfad von Baumkanten von  $v$  nach  $w$ , so ist  $dfsnum(v) < dfsnum(w)$ .
5. Für alle Kanten  $(v, w) \in E$  gilt:
  - a)  $(v, w) \in T \cup F \Leftrightarrow dfsnum(v) < dfsnum(w)$
  - b)  $(v, w) \in B \Leftrightarrow dfsnum(v) > dfsnum(w)$  und  $compnum(w) > compnum(v)$
  - c)  $(v, w) \in C \Leftrightarrow dfsnum(v) > dfsnum(w)$  und  $compnum(w) < compnum(v)$

# DFS - Folgerungen

1. Einteilung der Kanten in  $T, F, B, C$  kann über  $dfsnum/compnum$  erfolgen.
2. Wendet man DFS auf azyklische gerichtete Graphen an, dann gibt es keine Rückwärtskanten.

Es gilt für alle Kanten  $(v, w) \in E$ , dass  $compnum(v) > compnum(w)$ .

Also ist  $n + 1 - compnum(v)$  für alle  $v \in V$  eine topologische Sortierung, und damit DFS ein alternativer Algorithmus zu TopSort.

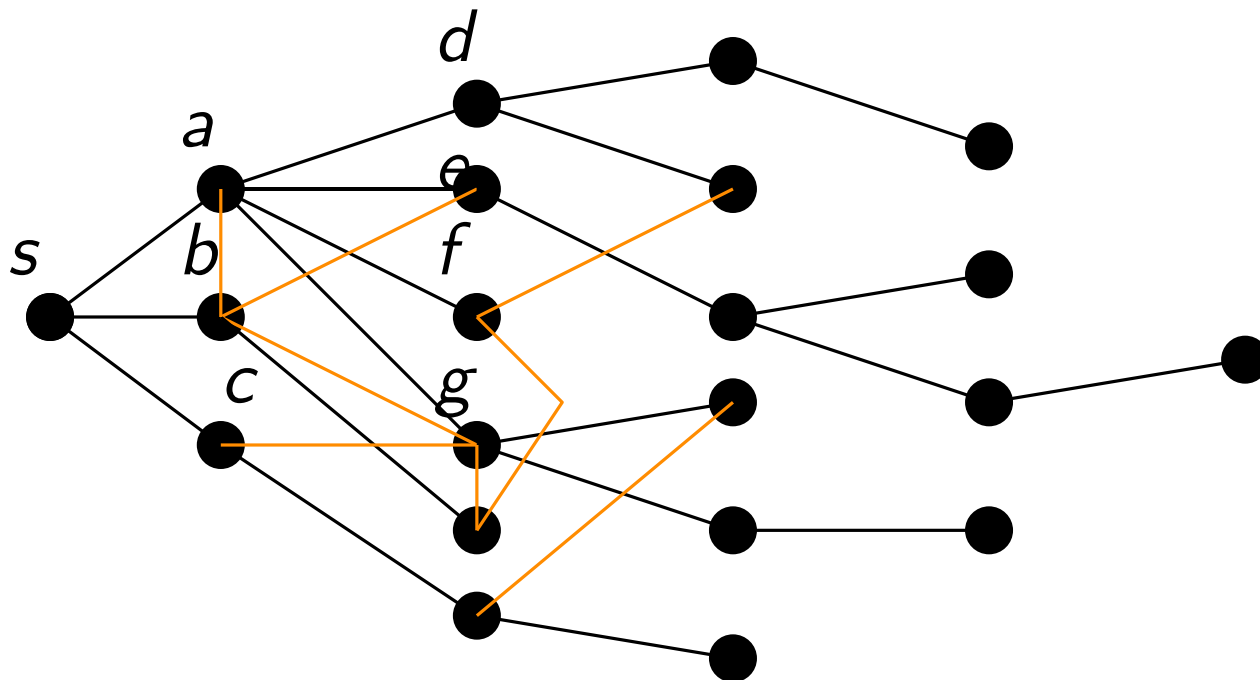
# BFS näher betrachtet

```
 $S \leftarrow S' \leftarrow s;$   
for all  $(v \in V)$   $\{p(v) \leftarrow \text{adjlistenstart}(v)\}$   
while  $(S' \neq \emptyset)$  {  
    sei  $v \in S'$  beliebig;  
    if  $(p(v)$  nicht am Listenende) {  
         $w \leftarrow p(v)$ ; verschiebe  $p(v)$ ;  
        if  $(w \notin S)$  {einfüge( $w, S$ ); einfüge( $w, S'$ )}  
    }  
    else streiche( $v, S'$ )  
}
```

$S'$  verwaltet die Knoten, von denen aus noch exploriert wird:  
In BFS wird  $S'$  als Queue realisiert.

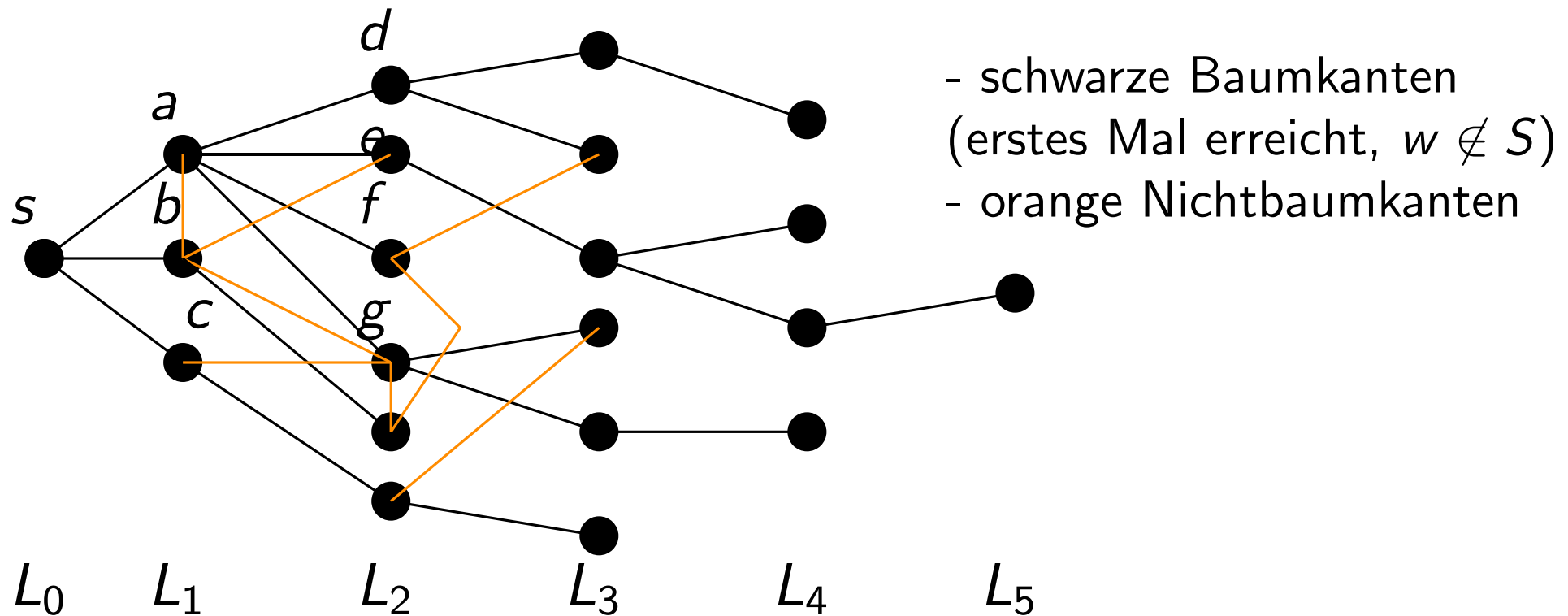
# BFS

1. Neuer Knoten  $w$  wird HINTEN eingefügt.
2.  $w$  wird erst exploriert, wenn alle anderen besuchten Knoten exploriert wurden.
3. Ist  $w$  an der Reihe, werden nacheinander alle Kanten zu den Nachbarn von  $w$  exploriert, erst dann andere Knoten.





# BFS



Durchmusterung ergibt Einteilung in Levels  $L_0, L_1, L_2, \dots$

Levels ergeben grad den Abstand zum Start  $s$ .

Baumkanten gehen von Level  $L_i$  zu  $L_{i+1}$  für  $i \geq 0$ .

Nichtbaumkanten ebenfalls und auch gleiches Level !

**Frage:** Wie ist das bei gerichteten Kanten ?

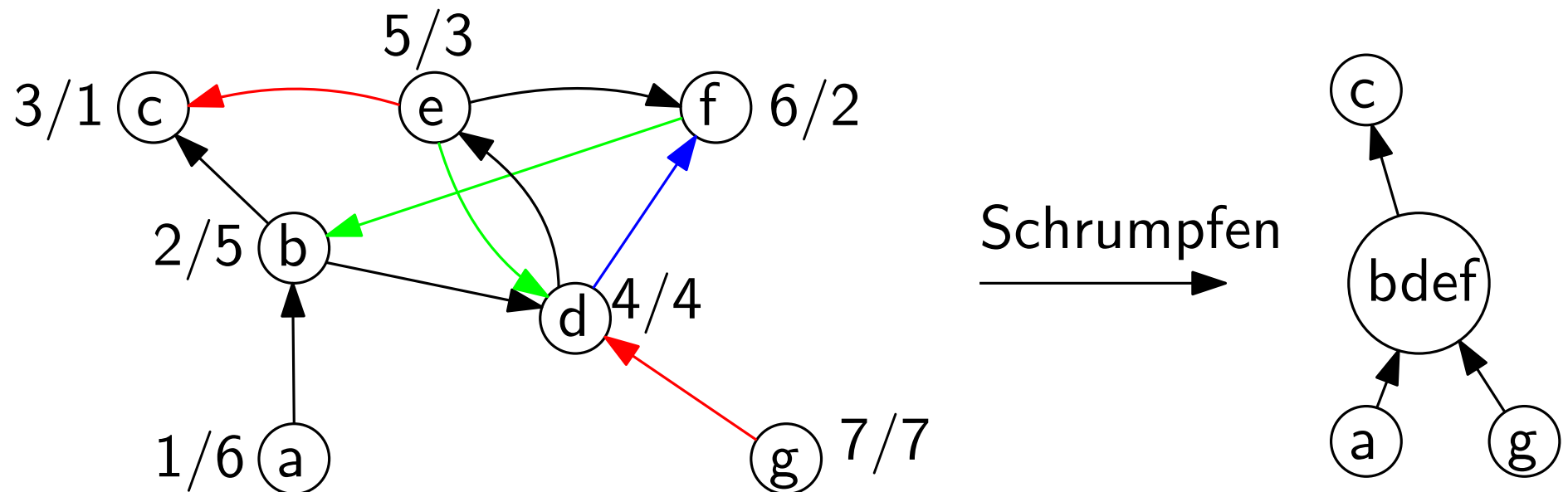
# DFS und einige Korollare

15+16/12/2020

# Starke Zusammenhangskomponenten

Gerichteter Graph  $G = (V, E)$  heißt **stark zusammenhängend**, gdw. für alle  $v, w \in V$  gilt:  $v \rightarrow^* w$ .

Eine **starke Zusammenhangskomponente** SZK ist ein maximaler stark zusammenhängender Teilgraph von  $G$ .



# SZKs: Konzepte

**Definition:** Sei  $(V', E')$  eine SZK von  $G$ . Knoten  $v \in V'$  heisst **Wurzel**, wenn er die minimale *dfsnum* der Knoten in  $V'$  hat.

**Lemma:** Sei  $(V', E')$  eine SZK mit Wurzel  $r$ . Dann gilt, dass alle Knoten der SZK von  $r$  aus über Baumkanten erreichbar sind.

**Idee:** Sei  $G_a = (V_a, E_a)$  der aktuell bekannte Graph. Wir verwalten die SZKs von  $G_a$ . Merke jeweils Wurzeln der SZKs.

Starte mit  $V_c = \{a\}$ ;  $E_c = \emptyset$ ;

betrachte nun Kante  $(v, w)$ :

- Ist  $(v, w) \in T$ , so kommt  $w$  zu  $V_c$  hinzu, bildet neue SZK.
- Ist  $(v, w) \notin T$ , mische mehrere SZK zu einer.

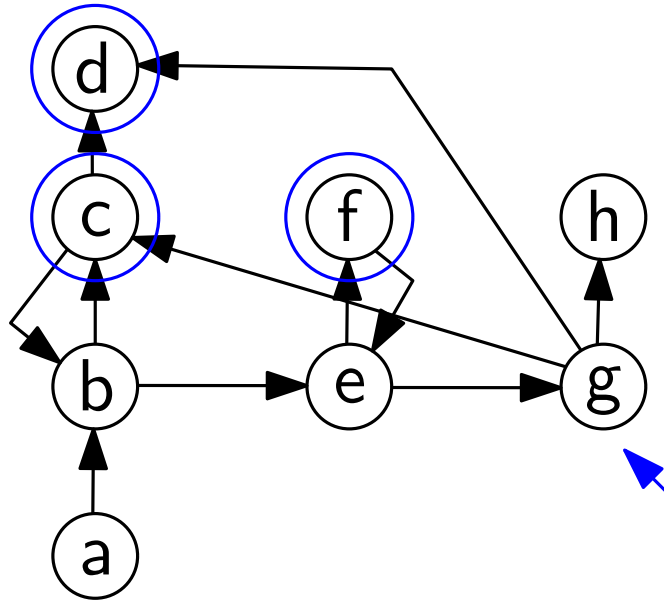
# SZK Konzepte

SZK  $K$  heisst **abgeschlossen**, falls die  $dfs(v)$  für alle  $v \in K$  abgeschlossen ist.

**Wir halten uns zwei Mengen (Stacks):**

- **Wurzeln**: Folge der Wurzeln in nicht abgeschlossenen Komponenten in aufsteigender  $dfsnum$
- **unfertig**: Folge der Knoten  $v$ , für die  $dfs(v)$  aufgerufen ist. SZK aber noch nicht abgeschlossen. Ebenfalls nach aufsteigender  $dfsnum$  geordnet.

# SZK Beispiel



**Unfertig:** a, b, c, e, g

**Wurzeln:** a, b, e, g

## Kanten aus g:

1. (g,d) ändert nichts. SZK von d abgeschlossen.
2. (g,c) vereinigt 3 SZKs mit Wurzeln b, e, g.  
Lösche e und g. Es bleibt Wurzel b.
3. (g,h)  $\in T$ . h ist SZK und wird in **unfertig** und **Wurzeln** eingefügt.

# SZK: DFS Pseudocode-Erweiterung

**in main:**

```
unfertig  $\leftarrow$  Wurzeln  $\leftarrow$  leerer Stack;  
  for all ( $v \in V$ ) {inunfertig( $v$ )  $\leftarrow$  false;}
```

**in  $dfs(v)$ :**

```
push( $v$ , unfertig); uninfertig( $v$ )  $\leftarrow$  true;  $count1++$ ;  
 $dfsnum(v) \leftarrow count1$ ;  $S \leftarrow S \cup \{v\}$ ; push( $v$ , Wurzeln);  
for all ( $(v, w) \in E$ ) {  
  if ( $w \notin S$ ) { $dfs(w)$ ;}  
  else if (inunfertig( $w$ )) {  
    while ( $dfsnum(w) < dfsnum(\text{top}(\text{Wurzeln}))$ ) {  
      pop(Wurzeln) }  
    }  
  }  
 $count2++$ ;  $compnum(v) \leftarrow count2$ ;
```

.....

# DFS Pseudocode-Erweiterung

```
if ( $v = \text{top}(\text{Wurzeln})$ ) {  
    repeat {  
         $w \leftarrow \text{top}(\text{unfertig})$ ;  
         $\text{inunfertig}(w) \leftarrow \text{false}$ ;  
         $\text{pop}(\text{unfertig})$   
    } until ( $w = v$ )  
    pop(Wurzeln);  
}
```

Schliesst SZK ab  
mit Wurzel  $v$ .

**Laufzeit:**  $O(n + m)$

## Satz:

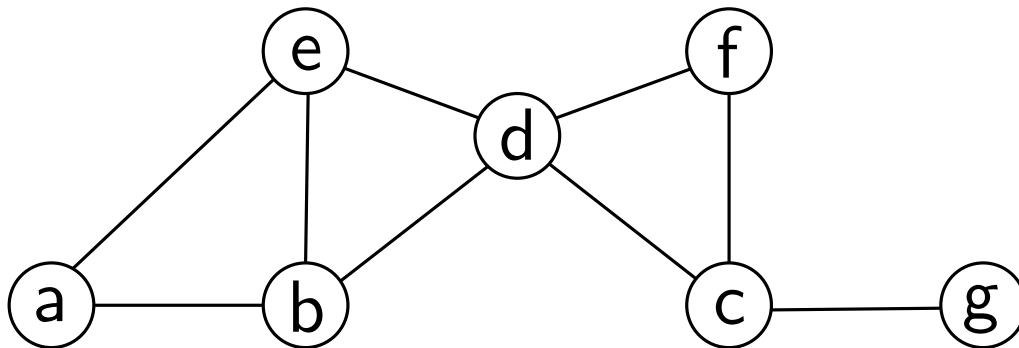
Sei  $G = (V, E)$  ein gerichteter Graph. Die starken Zusammenhangskomponenten von  $G$  können in Zeit  $O(n + m)$  mit einer Variante von DFS bestimmt werden.



# 2-fache Zusammenhangskomponenten (2ZK)

## Definitionen:

1. Ein ungerichteter Graph  $G = (V, E)$  heisst **2-fach zusammenhängend**, falls für alle  $v \in V$  auch  $G \setminus \{v\}$  zusammenhängend.
2. Eine **2ZK** eines ungerichteten Graphen ist ein maximaler 2-fach zusammenhängender Teilgraph.
3.  $v \in V$  heisst **Artikulationspunkt**, wenn  $G \setminus \{v\}$  nicht zusammenhängend ist.



## 2ZKs:

$\{a, b, e, d\}$ ,  $\{d, c, f\}$ ,  $\{c, g\}$

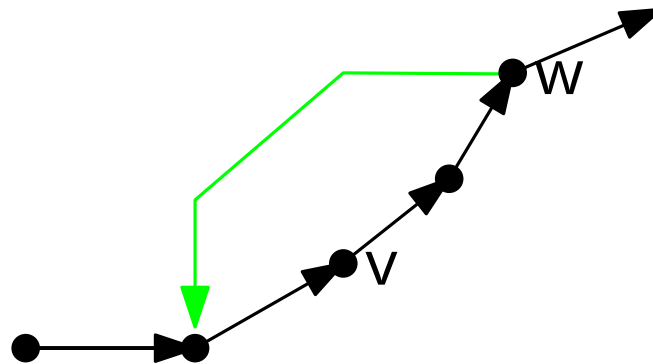
## Artikulationspunkte:

c, d

# 2ZKs: Berechnung

## DFS für ungerichtete Graphen:

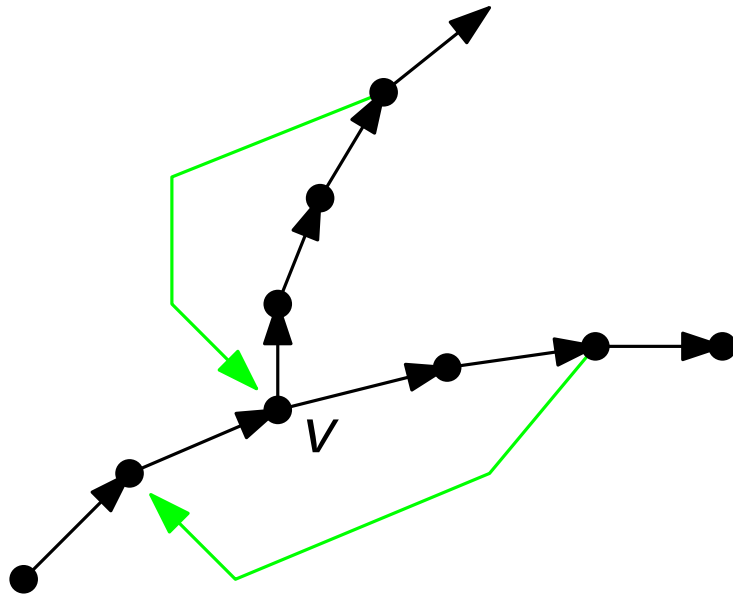
liefert Baumkanten und Rückwärtskanten, aber keine Vorwärtskanten und keine Querkanten.



**Idee:**  $v$  ist kein Artikulationspunkt, falls Baumnachfolger  $w$  eine Rückwärtskante 'vor'  $v$  hat.

→ '**vor**':  $dfsnum$  kleiner  $dfsnum(v)$

## 2ZKs: Artikulationspunkte berechnen



$v$  ist Artikulationspunkt, da es einen Zweig von  $v$  aus gibt, von wo es keine B-Kante 'vor'  $v$  gibt.

Für Wurzel  $v$  des Baums gilt, dass  $v$  Artikulationspunkt, wenn es mehr als einen Zweig gibt.

### Definiere

$$\text{low}(u) \leftarrow \min( \text{dfsnum}(u), \min_v \{ \text{dfsnum}(v) \text{ mit Pfad } u \xrightarrow{*}_T z \rightarrow_B v \} )$$

# 2ZK: Berechnung Art.-Punkte

**dfs( $v$ ) rekursiv:**

**for all**  $((v, w) \in E)$  {

**if**  $(w \notin S)$  {

$dfsnum(w) \leftarrow count1$ ;  $low(w) \leftarrow dfsnum(w)$ ;

$count1++$ ; **dfs**( $w$ );

→ **if**  $(low(w) < low(v))$  {  $low(v) \leftarrow low(w)$ ; }

→ **if**  $(low(w) \geq dfsnum(v))$  {  $artikulationspt(v) \leftarrow true$ ; }

}

→ **else** { **if**  $dfsnum(w) < low(v)$  {  $low(v) \leftarrow dfsnum(w)$ ; }

}

# 2ZKs: Art.-Punkte berechnen

## Erläuterungen:

- 1. if:** low-Wert von  $w$  wird an  $low(v)$  übergeben, falls er kleiner ist.
- 2. if:** erkennt, ob Zweig von  $v$  nach  $w$  keine Rückwärtskante 'vor'  $v$  enthält  $\Rightarrow v$  Artikulationspunkt
- 3. if:** Rückwärtskante von  $v$  nach  $w$  wird berücksichtigt.

**Spezialfall:**  $v$  Wurzel mit kleinster dfsnum

**if** ( $dfsnum(v) = 1$  und  $\exists w_1, w_2$  mit  $(v, w_1), (v, w_2) \in T$ ) {  
    artikulationspunkt( $v$ )  $\leftarrow$  true;}

# 2ZKs Conclusion

Das war Bestimmung der Artikulationspunkte in Zeit  $O(n + m)$ , durch Verfeinern von DFS. Eigentliche 2ZKs können (relativ leicht) daraus abgeleitet werden.

## **Satz:**

Für gegebenen ungerichteten Graphen  $G = (V, E)$  können in Zeit  $O(n + m)$  die 2-fachen Zusammenhangskomponenten mit modifiziertem DFS bestimmt werden.

# Minimale aufspannende Bäume (MST)

15/12/2020

# Minimale aufspannende Bäume

Sei  $G = (V, E)$  ein ungerichteter Graph mit  $n$  Knoten und  $m$  Kanten.

Ein Teilgraph  $T = (V', E')$  mit  $V' \subseteq V$  und  $E' \subseteq E$  heisst **aufspannender Baum**, falls  $T$  azyklisch und  $V' = V$  und  $|E'| = n - 1$ .

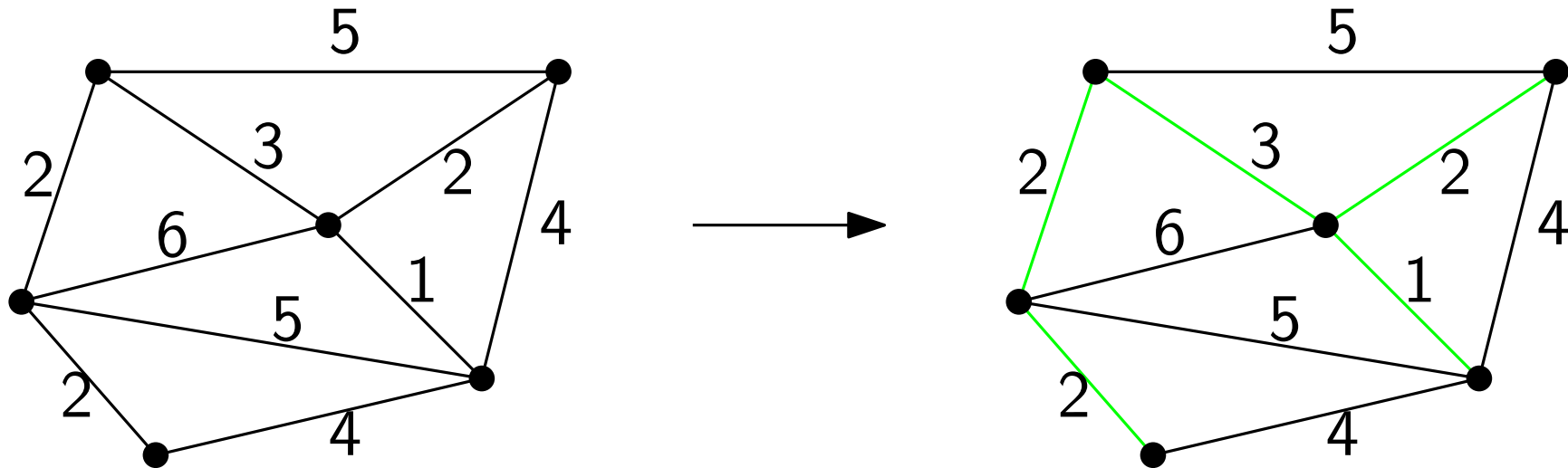
**Beachte:**  $T$  ist zusammenhängend (alle Pfade existieren).

Sei  $c$  **Kostenfunktion** auf den Kanten. Die Kosten  $c(T)$  des Baums  $T$  sind definiert als  $\sum_{e \in E'} c(e)$ .

Ein **MST** ist ein aufspannender Baum mit minimalen Kosten.



# Minimale aufspannende Bäume



Grün ist der MST. Warum ?

**Ideen:** Kurze Kanten zuerst! Zyklfreiheit!

# 1. Algorithmus: Greedy

Ordne Kanten  $e_1, \dots, e_m$  so dass  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$

Baumkanten  $E_T \leftarrow \emptyset$ ;

```
for all ( $i = 1, \dots, m$ ) {  
    if ( $E_T \cup \{e_i\}$  azyklisch) { $E_T \leftarrow E_T \cup \{e_i\}$ }  
}
```

**Name:** Kruskal's Algorithmus

**Korrektheit:**

**Idee:** Kantenmenge  $E'$  heisst **gut**, wenn sie zu einem MST erweiterbar ist.

**Zeige:**  $E_T$  bleibt in jeder Iteration gut. ( $E_T$  ist anfangs trivialerweise gut.)

# Kruskals's Algorithmus: Korrektheit.

## Wir folgen der Idee:

Sei  $E'$  eine gute Teilmenge und sei  $e$  die billigste Kante, so dass  $E' \cup e$  azyklisch ist.

**Beh.:** Dann ist  $E' \cup e$  ebenfalls gut.

**Bew.:** Sei  $E''$  die Erweiterung von  $E'$  zu einem MST.

Ist  $e \in E''$ , so ist  $E' \cup \{e\}$  auch gut. Beh. gilt.

Ist  $e \notin E''$ , betrachte  $H = (V, E' \cup E'' \cup \{e\})$ .

Graph  $H$  enthält Zykel, auf dem  $e$  liegt. Da auch  $E' \cup \{e\}$  azyklisch, gibt es in dem Zykel eine Kante  $e' \neq e$ , die nicht in  $E'$  liegt.

Nach Wahl von  $e$  gilt  $c(e) \leq c(e')$ . Da  $E' \cup E''$  MST, so erhalten wir auch MST, wenn  $e'$  durch  $e$  ersetzt wird.



# Kruskal's Algorithmus: Implementierung

Brauchen Test, ob Kante  $e_i$  einen Zykel bzgl.  $E_T$  schliesst.

**Idee:** Wir halten uns Partition  $V_1, \dots, V_k$  der Knoten  $V$ , so dass die Kanten aus  $E_T$  Bäume auf den Knotenmengen  $V_j$  induzieren. Für  $1 \leq i, j \leq k$  hängen die Teilmengen  $V_i, V_j$  nicht zusammen.

Starten mit Partitionen  $\{\{1\}, \{2\}, \dots, \{n\}\}$ .

## 2 Operationen:

- **Find**( $x$ ) liefert den Namen der Menge  $V_j$ , in der  $x$  liegt.
- **Union**( $A, B$ ) vereinigt die beiden Mengen  $A$  und  $B$

# Kruskals Algorithmus: 2. Version

.....

```
for ( $i = 1, \dots, m$ ) {  
    sei  $e_i = (v, w)$ ;  
     $A \leftarrow Find(v)$ ;  $B \leftarrow Find(w)$ ;  
    if ( $A \neq B$ ) {  $E_T \leftarrow E_T \cup \{e_i\}$ ;  $Union(A, B)$ ; }  
}
```

## Implementierung von 'Find' und 'Union':

1. merken uns ein Namensfeld  $R : V \rightarrow \{1, \dots, n\}$ , so dass  $R(x)$  den Namen der Menge, in der  $x$  liegt, angibt.

**Find**( $x$ ):      **return**  $R(x)$ ;

**Union**( $A, B$ ): **for** ( $i = 1$  to  $n$ ) {  
                  **if** ( $R(i) = A$ ) {  $R(i) \leftarrow B$  }  
                  }  
}

# Union-Find Implementierung

## Laufzeit:

- 'Find' geht in  $O(1)$ , 'Union' in  $O(n)$ .
- haben  $2m$  Finds und  $n - 1$  Unions. (wieso ?)
- also insgesamt  $O(m + n^2 + m \log m)$ .

**Verbesserung:** Union dauert zu lange (immer  $O(n)$ ).

Mache Namensänderung proportional zur Größe der kleineren Menge und behalte den Namen der größeren Menge.

# Union-Find Implementierung

**genauer:**

- Halte Mengen als Listen 'Elem(A)' und merke Größe
- 'size(A)'. Namensfeld R wie vorher.

**Initialisierung:** **for** ( $i = 1, \dots, n$ ) {  
                     $R(x) \leftarrow x; Elem(x) \leftarrow x; size(x) \leftarrow 1;$ }

**Find**( $x$ ): **return**  $R(x)$ ;

**Union**( $A, B$ ): **if** ( $size(A) < size(B)$ ) {tausche  $A$  und  $B$ ;}  
                    **for all** ( $x \in Elem(B)$ ) {  
                                 $R(x) \leftarrow A$ ; hängen( $x, Elem(A)$ );  
                            }

**Laufzeit von Union:**  $O(\min(|A|, |B|))$ .

# Union-Find Implementierung

**Laufzeit der insgesamt  $n - 1$  Unions:**  $O(\sum_i n_i)$ , wobei  $n_i$  die Größe der kleineren Menge in der  $i$ -ten Iteration ist

**Es gilt:**  $O(\sum_i n_i) = O(\sum_j r_j)$ , wobei  $r_j$  die Zahl der Mengenwechsel für  $j$  angibt.

**Beh.:**  $r_j \leq \log n$  für  $j = 1, \dots, n$ , also jedes Element wechselt  $\leq \log n$  mal die Menge.

**Bew.:** Wenn  $j$  Menge wechselt, und vorher in Menge mit  $k$  Elementen war, hat seine neue Menge Größe  $\geq 2k$ .

Nach  $l$  Wechseln ist damit die Größe  $\geq 2^l \leq n$   
 $\Rightarrow l \leq \log n$ .

**Laufzeit:**  $O(m + n \log n + m \log n)$ .



# Kruskal's Algorithmus: Laufzeit

## **Satz:**

In einem ungerichteten Graph mit Kantenkosten kann ein MST in Zeit  $O(m \log n + n \log n + m)$  berechnet werden.

**Bem.:** Das war amortisierte Analyse bei Union-Find.

# Alternative: Prim's Algorithmus

**Idee:** Baue nur einen Baum!

```
 $E_T \leftarrow \emptyset;$   
 $T \leftarrow \{v\};$   
while ( $T \neq V$ ) {  
    finde  $e = (u, v)$  mit  $u \in T, v \notin T$  und  $c(e)$  minimal;  
     $E_T \leftarrow E_T \cup \{e\};$   
     $T \leftarrow T \cup \{v\};$   
}
```

**Korrektheit:** Argumente wie vorher über 'gute' Kanten

**Laufzeit:** Da Algorithmus ähnlich wie Dijkstra, versuchen wir ähnliche Implementierung

# Prim's Algorithmus

## Implementierung:

Speichern Knoten  $w \notin T$  geordnet nach Kantenkosten in PQ als Menge  $\{c(w) \mid c(w) = \min\{c(u, w)\} \text{ mit } u \in T\}$ .

'Geeignete Kante  $e$  aussuchen' entspricht 'DeleteMin(PQ)'.

Wird dann  $w$  in  $T$  eingefügt, so mache Kostenupdate für alle  $x \in PQ$  mit  $(w, x) \in E$ :  $c(x) \leftarrow \min\{c(x), c(w, x)\}$ .

Entspricht 'Decreasekey' bei Dijkstra.

**Laufzeit:**  $n$  mal 'DeleteMin',  $m$  mal 'Decreasekey'

PQ als binärer Heap  $\rightarrow O(m \log n)$

oder  $(a, b)$ -Baum mit  $a = \max(2, \frac{m}{n})$ ,  $b = 2a$

$\rightarrow O(m \log \frac{n}{(m/n)})$ .

# Prim's Algorithmus und Satz

## Satz:

MST-Berechnung für ungerichteten Graph  $G = (V, E)$  geht mit Prim's Algorithmus in Zeit  $O(m \log \frac{n}{m/n})$  bzw.  $O(m + n \log n)$  (mit Fibonacci-Heaps).

# Algorithmen: Suchen in geordneten Mengen

12/1/2021

# Suchen in geordneten Mengen

**Modell:** Gegeben Universum  $U$  mit linearer Ordnung  $<$   
Gegeben: Teilmenge  $S \subset U$  “Schlüsselmenge”

Sei  $a \in U$ .

**Operationen:** Suche  $(a, S)$ .

Einfüge  $(a, S)$  / Streiche  $(a, S)$

Finde  $k$ -t größtes Element  $ord(k, S)$  in  $S$

$Sort(S)$

# Lineare Suche

geg.  $S$  als geordnetes Array  $S[0, \dots, n - 1]$

sei  $next$  Suchindex, initialisiert mit  $next \leftarrow 0$

```
while ( $a < S[next]$ )  
{  
     $next \leftarrow next + 1$ ;  
}  
if ( $a = S[next]$ ) return  $next$   
else return 'a nicht vorhanden';
```

**Laufzeit:**  $O(1)$  im besten Fall

Im schlechtesten Fall  $n$  Iterationen  $\rightarrow O(n)$

Im Mittel  $n/2$  Iterationen  $\rightarrow O(n)$

# Binäre Suche

geg.  $S$  als geordnetes Array  $S[0, \dots, n - 1]$ , gesucht:  $a$

3 Indexvariablen:  $next$ ,  $oben$ ,  $unten$  Grenzen des Suchbereichs

initial:  $oben \leftarrow n - 1$ ,  $unten \leftarrow 0$

```
while ( $oben - unten > 1$ )  
{  
     $next \leftarrow \lceil (oben - unten)/2 \rceil + unten$   
    Test  $S[next] = < > a; \dots$   
}
```

**Laufzeit:**  $T(n) = T(n/2) + c$  (worst case)  
               $= O(\log n)$  (Raten, Ausrechnen, Mastertheorem)



# Interpolationssuche

Suche im Telefonbuch !!

$$next \leftarrow \lfloor \frac{a - S[unten]}{S[oben] - S[unten]} \cdot (oben - unten) \rfloor + unten$$

Schätzen, wo  $a$  relativ zu  $S[unten]$ ,  $S[oben]$  liegt.

Laufzeit gut im **Mittel**:  $O(\log \log n)$

Laufzeit schlecht im **schlechtesten** Fall:  $O(n)$

*Wie kommt man auf  $O(\log \log n)$  Laufzeit?*

$$\begin{aligned} T(n) &= T(\sqrt{n}) + 1 \\ &= T(n^{1/2}) + 1 = T(n^{1/2^i}) + i = O(\log \log n) \end{aligned}$$

# Interpolationssuche

**Idee:** Reduziere pro Schritt den Suchbereich auf "Wurzel(Größe)"

## Algorithmus:

neue Elemente  $S[-1], S[n]$

Iteriere bis " $a$  gefunden oder oben-unten  $\leq 1$ "

$$next \leftarrow \left\lceil \frac{a - S[unten-1]}{S[oben+1] - S[unten-1]} \cdot (oben - unten + 1) \right\rceil + (unten - 1)$$

$$n \leftarrow oben - unten + 1$$

1. **Interpolation:** Vergleich  $a$  mit  $S[next]$

**If** " $=$ ": fertig

2. **If** ( $a > S[next]$ ) vergleiche  $S[next + \sqrt{n}], S[next + 2\sqrt{n}], \dots$   
bis  $a < S[next + (i - 1)\sqrt{n}]$

Nach  $i$  Vergl. Suchbereich  $[S[next + (i - 2)\sqrt{n}], S[next + (i - 1)\sqrt{n}]]$

# Interpolationssuche: Analyse

haben Problemgröße in  $i$  Schritten von  $n$  auf  $\sqrt{n}$  reduziert

Sei  $C$  die mittlere Anzahl von Vergleichen in einer Iteration

**Behauptung:**  $C \leq 2,4$

**Erhalten:**  $\bar{T}(2) = 2$   
 $\bar{T}(n) = \bar{T}(\sqrt{n}) + C \Rightarrow \bar{T}(n) = 2 + C \log \log n$

**Satz:** Interpolationssuche braucht  $\leq 2 + 2,4 \log \log n$   
Vergleiche **im Mittel**.

# Interpolationssuche: Bemerkungen

**Worst case:**  $\sqrt{n} + \sqrt{\sqrt{n}} + \dots = O(\sqrt{n})$ .

**Idee:** Besser, wenn binäre Suche und Interpolationssuche parallel laufen

$\Rightarrow$  worst case:  $O(\log n)$

**Beachte:** Gleichverteilungsannahme bei Analyse kritisch !

# Algorithmen: Sortieren

13/1/2021

# Gliederung in Themen

**I. Einführung**

**II. Grundlegende Datenstrukturen**

**III. Graphenalgorithmen**

**IV. Suchen**

---

**V. Sortieren**

**1. Elementare Sortieralg.**

**2. Quicksort, Heapsort, Mergesort**

**3. Bucketsort**

**VI. Datenstrukturen - 2**

**VII. Generische algorithmische Methoden**

**Zu IV. und V. Untere Schranken, Medianalgorithmus**

# Zeitplan bis zur Klausur am 3.3.

## **V. Sortieren** (19./20.1.)

– Elementare Sortialg, Heapsort, Mergesort, Bucketsort

## **VI. Datenstrukturen - 2** (26./27.1., 2/3.2., 9.2.)

– Suchbäume, AVL-Bäume, B-Bäume, (2,3)-Bäume, Anwendungen, Skiplists

## **VII. Generische algorithmische Methoden** (10.2., 16.2)

– Dynam. Programmieren, Branch+Bound

**Zu IV., V.** Untere Schranken, Medianalgorithmus (17.2., 23.2.)

Letztes Übungsblatt am 11.2., Abgabe 18.2., Bespr. 22.2.

Probeklausur am 23.2. (Helpdesk), Musterlösung 24.2.

Klausurvorbereitung: Tutorien 25./26.2., Helpdesk: 1.3.

# Einfache Verfahren

brauchen sortiertes Array!

Aus  $A : 17, 5, 8, 6, 2, 8, 12$  mache  $B : 2, 5, 6, 8, 8, 12, 17$

1. **Minimumssuche + Austausch** mit erstem Element, usw.

```
for  $i \leftarrow 1$  to  $n$  do
   $min \leftarrow i$ ;
  for  $j \leftarrow i + 1$  to  $n$  do
    if  $A[j] < A[min]$  then
       $min \leftarrow j$ 
    end
  end
  tausche( $A[i]$ ,  $A[min]$ );
end
```

Komplexität:  $T(n) = T(n - 1) + O(n) = O(n^2)$



# Einfache Verfahren

2. **Insertion Sort**: Füge immer an richtige Stelle ein. Mache Platz!

```
for  $i \leftarrow 1$  to  $n$  do  
  |  $key \leftarrow A[i]; j \leftarrow i;$   
  | while  $j > 1 \ \&\& \ A[j - 1] > key$  do  
  |   |  $A[j] \leftarrow A[j - 1];$   
  |   |  $j--;$   
  | end  
  |  $A[j] \leftarrow key;$   
end
```

Komplexität:  $T(n) = T(n - 1) + O(n) = O(n^2)$

## 2. Insertion Sort (Nachtrag)

### Korrektheit:

Invariante: Die ersten  $i - 1$  Einträge sind sortiert.

Argumentieren durch Induktion über  $i$ :

$i = 1$  : Vor der ersten Schleife ist noch nichts sortiert. Also ok.

$(i - 1) \rightarrow i$  : Wir betrachten den Eintrag  $A[i] = key$  und fügen ihn an die richtige Stelle ein.

Größere Einträge nach rechts verschieben !

Richtige Stelle finden in *while* Schleife.

Danach sind Einträge  $A[1], \dots, A[i]$  sortiert.

# Einfache Verfahren

## 3. Bubble Sort:

In  $i$ -ter Iteration sind die letzten  $n - (i - 1)$  Stellen schon sortiert. Jeweils größtes Element läuft von 1 nach  $i$  hoch.

```
for  $i \leftarrow n$  to 1 do
  | for  $j \leftarrow 1$  to  $i - 1$  do
  | | if  $A[j] > A[j + 1]$  then
  | | | tausche( $A[j], A[j + 1]$ );
  | | end
  | end
end
```

Rekursion?  $T(n) = \dots?$

# Quicksort (Divide and Conquer)

Eingabe:  $S = \{a_1, \dots, a_n\}$

1. Wähle Pivotelement  $a_1$ .

Teile  $S - \{a_1\}$  auf in

$A = \{a_i \mid a_i \leq a_1 \text{ für } i \geq 2\};$     sei  $|A| = j;$

$B = \{a_i \mid a_i > a_1 \text{ für } i \geq 2\};$

Speichere  $A$  in  $S[1], \dots, S[j]$ ,  $a_1$  in  $S[j + 1]$  und  
 $B$  in  $S[j + 2], \dots, S[n]$

2. Sortiere  $A$  und  $B$  rekursiv

Worst case:  $T(n) = O(n) + T(0) + T(n - 1)$  ( $a_1$  max./min.)  
 $= O(n^2)$

# Quicksort: Mittlere Analyse

**Wahrscheinlichkeitsmodell:** Elemente paarweise verschieden, alle Permutationen der Eingabe sind gleichwahrscheinlich  $= 1/n!$

- o.B.d.A.  $S = \{1, 2, 3, \dots, n\}$
- Also  $\text{prob}(S[1] = k) = 1/n$  für  $k = 1, 2, 3, \dots, n$
- Teilprobleme der Größen  $k - 1$  und  $n - k$  erfüllen W-Annahmen.

$\overline{QS}(n)$  sei mittlere Anzahl Vergleiche an Problemgröße  $n$

$$\overline{QS}(0) = \overline{QS}(1) = 0$$

$$\overline{QS}(n) = \dots$$

# Quicksort: Mittlere Analyse

$$\begin{aligned}\overline{QS}(n) &= n + E(QSort(A) + QSort(B)) \\ &= n + 1/n \cdot \sum_{k=1}^n (\overline{QS}(k-1) + \overline{QS}(n-k)) \\ &= n + 2/n \cdot \sum_{k=1}^{n-1} \overline{QS}(k)\end{aligned}$$

$$n \cdot \overline{QS}(n) = n^2 + 2 \cdot \sum_{k=1}^{n-1} \overline{QS}(k)$$

$$\text{und } (n+1)\overline{QS}(n+1) = (n+1)^2 + 2 \cdot \sum_{k=1}^n \overline{QS}(k)$$

$$(n+1)\overline{QS}(n+1) - n \cdot \overline{QS}(n) = (n+1)^2 - n^2 + 2\overline{QS}(n)$$

$$(n+1)\overline{QS}(n+1) = 2n+1 + (n+2)\overline{QS}(n)$$

$$\overline{QS}(n+1) \leq 2 + \frac{n+2}{n+1} \overline{QS}(n)$$

$$= 2 + \frac{n+2}{n+1} \left( 2 + \frac{n+1}{n} \overline{QS}(n-1) \right)$$

$$= 2 + (n+2) \left( \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \dots \right)$$

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

# Quicksort: Mittlere Analyse

$$\begin{aligned}\Rightarrow \overline{QS}(n) &\leq 2 + 2(n+1) \sum_{i=1}^n \frac{1}{i} \\ &\leq 2 + 2(n+1)(\ln n + 1) \\ &= O(n \log n)\end{aligned}$$

**Satz:** Die mittlere Laufzeit von Quicksort ist  $O(n \log n)$ .

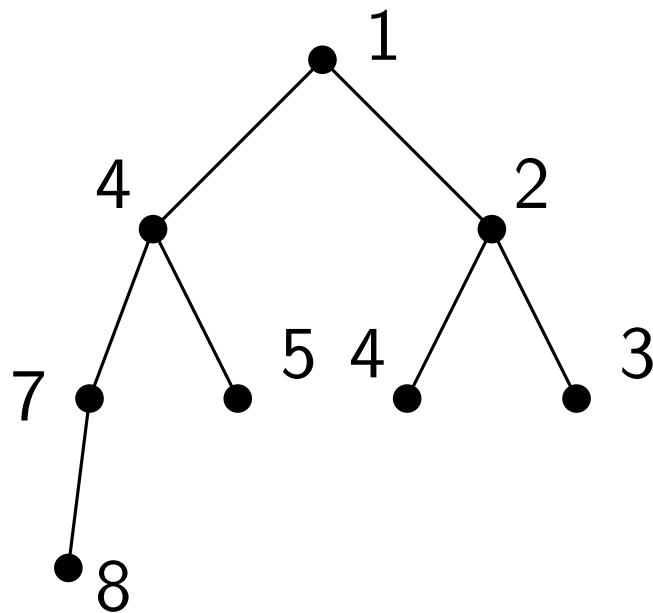
# Heapsort

Idee: Minimumsuche + Entfernen - aber effizient!!

Grundlage: Datenstruktur "binärer Baum"

**Heapeigenschaft:** Für Knoten  $u$  und  $v$  mit  $\text{Vater}(v) = u$  gilt:

$$S[u] \leq S[v]$$



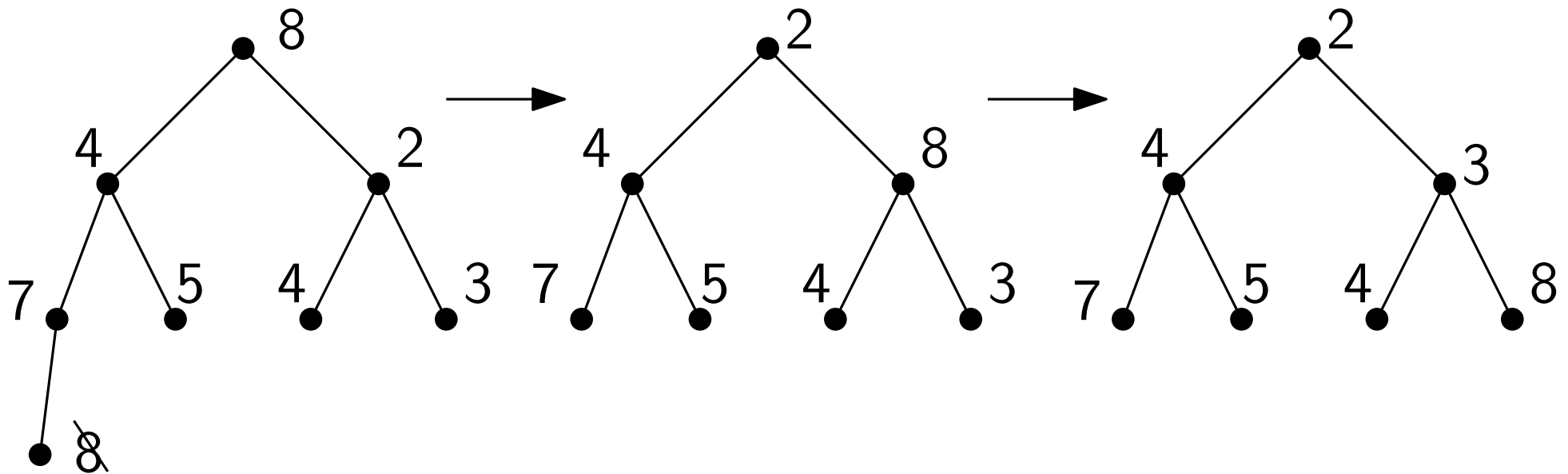
$$S = \{1, 2, 3, 4, 5, 4, 7, 8\}$$



# Heapsort: Idee

Im Heap steht Minimum in der Wurzel. Also:

1. **Suche Minimum**  $\rightarrow$  Wurzel  $O(1)$
2. **Entferne Minimum aus Heap. Repariere Heap:**
  - Nimm Blatt und füge es in Wurzel ein.
  - Lass es nach unten sinken (Vergleich mit kleinstem Kind)



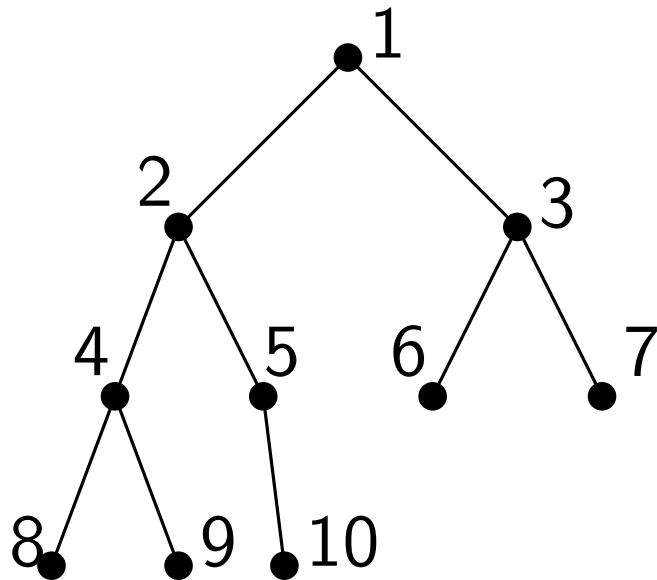
**Laufzeit:**  $O(\text{Höhe}(T))$

**Problem:** Mache Höhe klein!

# Heapsort

## Ausgewogene Bäume:

- Alle Blätter haben Tiefe  $k$  oder  $k + 1$ .
- Blätter auf Tiefe  $k + 1$  sitzen 'ganz links'.



Nummerierung schichtweise  
von links nach rechts

Für Knoten mit Nummer  $i$  gilt:

- Vater hat Nummer  $\lfloor i/2 \rfloor$
- Kinder haben Nummern  $2i$ ,  $2i + 1$ .

Heap wird als Array dargestellt, nur konzeptuell als Baum!  
Indizierungsregeln ergeben Vater-Kind-Relationen

# Heapsort: Algorithmus + Analyse

## Initialisierung des Heaps $H$ :

für alle  $a \in S$  {**Insert**( $a, H$ )}

## **Insert**( $a, H$ ):

sei  $n$  die Größe von  $H$ ;

$n \leftarrow n + 1$ ;

$H(n) \leftarrow a$ ;

$i \leftarrow j \leftarrow n$ ;

**while** ( $i > 1$ ) {

$j \leftarrow \lfloor j/2 \rfloor$ ;

**if** ( $H[j] > H[i]$ ) {tausche  $H[i]$  und  $H[j]$ ;  $i \leftarrow j$ };

**else**  $i \leftarrow 1$ ;

};

**Beispiel!**

# Heapsort: Analyse

## Laufzeit:

**Initialisierung** (Einfügen, Hochsteigen lassen):  $O(n \cdot \text{Höhe}(H))$

**Iteratives Minimumslöschen**:  $O(n \cdot \text{Höhe}(H))$

$\text{Höhe}(H)$  ist  $O(\log n)$ , da Baum ausgewogen ist.

**Satz:** Heapsort hat Laufzeit  $O(n \log n)$ , sogar im worst case.

# Heapsort

Überlege, ob Heapsort besser ist als Quicksort.  
Wann ist Bubblesort / Minimumssuche zu bevorzugen?

# Algorithmen: Mergesort, Bucketsort

19+20/1/2021

# Mergesort

gut für externe Speicher, wenn Datenmenge groß ist.

Mische jeweils 2 sortierte Folgen zusammen zu einer:

$\begin{array}{cccc} 1 & 2 & 4 & 7 \\ 5 & 6 & 8 & \end{array} > 1 \ 2 \ 4 \ 5 \ 6 \ 7 \ 8$

Vergleiche 2 **minimale Elemente**  
und schreibe das kleinste raus.

**Idee:** Starte mit Teilfolgen der Länge 1 (Einzelelemente)  
Mische zu Teilfolgen der Länge 2 ( $\leq n/2$  Teilfolgen),  
usw bis aus 2 Teilfolgen nur 1 werden

Es gibt  $\log n$  Iterationen, in jeder halbiert sich Anzahl der Teilfolgen. Jede Iteration kostet  $O(n)$ .

$\Rightarrow O(n \log n)$  Laufzeit.

# Merge Sort

## **Bemerkungen:**

- super simple. Kleine Konstanten !  
    Im wesentlichen beruht alles auf Algorithmus für Mischen.
- funktioniert auch mit Listen (greifen ganz streng sequentiell zu)
- formuliere als Pseudo Code !



# Mergesort -Variante

**Variante:**  $k$ -Wege-Mischen (mische  $k$  Teilfolgen zu einer)

**Vorteil:**  $\log_k n$  Iterationen

**Nachteil:** Mischen schwieriger. Minimumsbestimmung?

**Überlege:** Welches ist das optimale  $k$ ?

# Bucketsort (Fachverteilung)

geg.  $n$  Wörter über dem Alphabet  $\Sigma$ .  $|\Sigma| = m$ .  
Sortiere lexikographisch!

**Bsp:**  $a < aa < aba$ .

**Fall 1:** Alle Wörter haben Länge 1.

- Stelle  $m$  Fächer bereit (für  $a, b, \dots, z$ ).
- Wirf jedes Wort in entsprechendes Fach.
- Konkateniere die Inhalte der Fächer.

$a$	$b$		$d$				$z$
$a$	$b$		$d$	.....			$z$
$a$			$d$				$z$
$a$							

# Bucketsort (Fachverteilung)

geg.  $n$  Wörter über dem Alphabet  $\Sigma$ .  $|\Sigma| = m$ .  
Sortiere lexikographisch!

**Bsp:**  $a < aa < aba$ .

**Fall 1:** Alle Wörter haben Länge 1.

- Stelle  $m$  Fächer bereit (für  $a, b, \dots, z$ ).
- Wirf jedes Wort in entsprechendes Fach.  $O(n)$
- Konkateniere die Inhalte der Fächer.  $O(m)$

Einzelne Fächer werden als lineare Listen implementiert.  
Alle  $m$  Fächer sind zusammen ein Array der Größe  $m$ .

$\Rightarrow O(n + m)$

# Bucketsort (Fachverteilung)

**Fall 2:** Alle Wörter haben **gleiche** Länge  $k > 1$ .

Sei Wort  $a^i = a_1^i a_2^i \dots a_k^i$ .

**Idee:** Sortiere zuerst nach  $k$ -tem Zeichen, dann nach vorletztem.  
 Dann sind Elemente mit  $a_{k-1}^i = a_{k-1}^j$  automatisch richtig.  
 $\Rightarrow O((n + m)k)$

→ **Beispiel:** 124, 223, 324, 321, 123

Fächer	1	2	3	4
1.	321		223 123	124 324
2.		321 223 123 124 324		
3.	123 124	223	321 324	

Aufsammeln !

Aufsammeln !

Aufsammeln !

# Bucketsort (Fachverteilung)

**Fall 2:** Alle Wörter haben **gleiche** Länge  $k > 1$ .

Sei Wort  $a^i = a_1^i a_2^i \dots a_k^i$ .

**Problem:** Beim Listenauf sammeln werden leere Listen übersprungen.

- Jetzt:  $O((n + m)k)$
- Ziel:  $O(nk + m)$ , da  $nk$  Gesamtlänge der Wörter

**Lösung:** Lege  $k$  zusätzliche Fächer an. Merke im  $j$ -ten Fach die Buchst. sortiert, die an  $j$ -ter Stelle kommen.

Erzeuge Paare  $(j, a_j^i)$  für alle  $i$  und alle  $j$ . Sortiere nach 2., dann nach 1. Komponente.  $\Rightarrow O(nk)$

(dann haben wir für den  $j$ -ten Durchlauf im  $j$ -ten Fach alle Zeichen sortiert stehen, laufen nur durch und wissen, welches als nächstes kommt)

# Bucketsort: 3. Fall (allgemein)

Wort  $a^i$  hat Länge  $t_i$  und  $L = \sum_{i=1}^n t_i$ .

Sei  $r_{\max} = \max_i t_i$ .

**Idee:** Betrachte anfangs nur die langen Wörter. Sortiere!

## Algorithmus

1. **Erzeuge Paare**  $(t_i, \text{Verweis auf } a^i)$ .
2. **Sortiere Paare** nach 1. Komponente.  
 $L(k)$  sei Liste der Wörter der Länge  $k$ .
3. **Erzeuge  $L$  Paare**  $(j, a_j^i)$  für  $1 \leq i \leq n, 1 \leq j \leq t_i$   
und sortiere nach 2., dann nach 1. Komp.  
→ kennen alle nichtleeren Fächer für alle Stellen  $j$ .
4. **Sortiere Wörter**  $a^i$  durch Bucketsort,  
beachte dabei Wortlängen und Listen  $L(k)$

# Bucketsort

**Satz:** Bucketsort sortiert  $n$  Wörter der Gesamtlänge  $L$  über Alphabet  $\Sigma$  mit Alphabetgröße  $m$  in Zeit  $O(L + m)$ .

# Datenstrukturen 2: Balancierte Suchbäume

2+3/2/2021

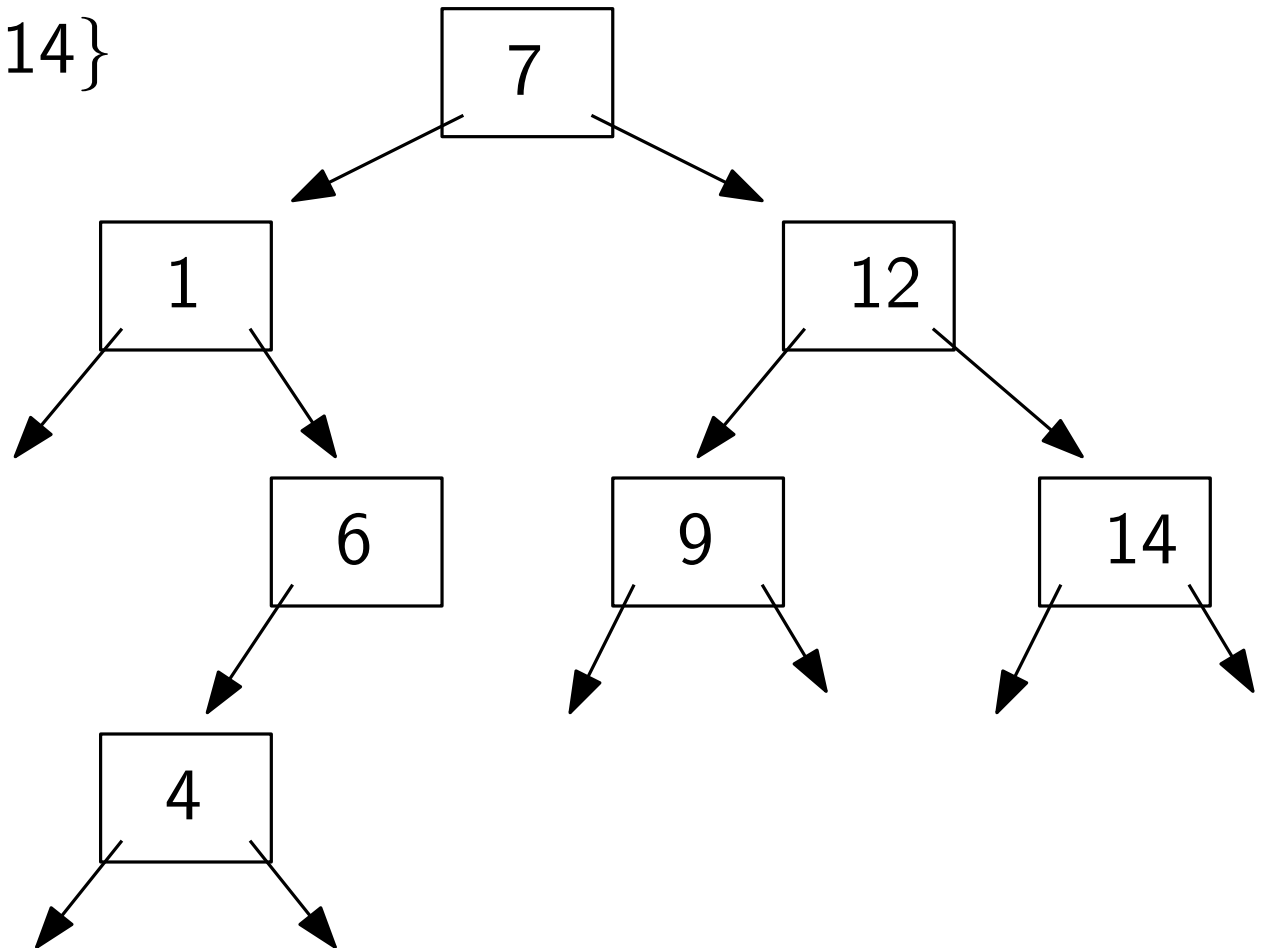


# Verwaltung dynamischer Mengen

## Operationen:

Suchen, Einfügen, Streichen, (Vereinigen, Aufspalten)

**Bsp:**  $\{1, 4, 6, 7, 9, 12, 14\}$



# Suchbäume

## Jeder Knoten $v$ hat:

- Schlüssel  $key(v)$
- Verweise auf Kinder  $lson(v), rson(v)$
- Verweis auf parent  $p(v)$

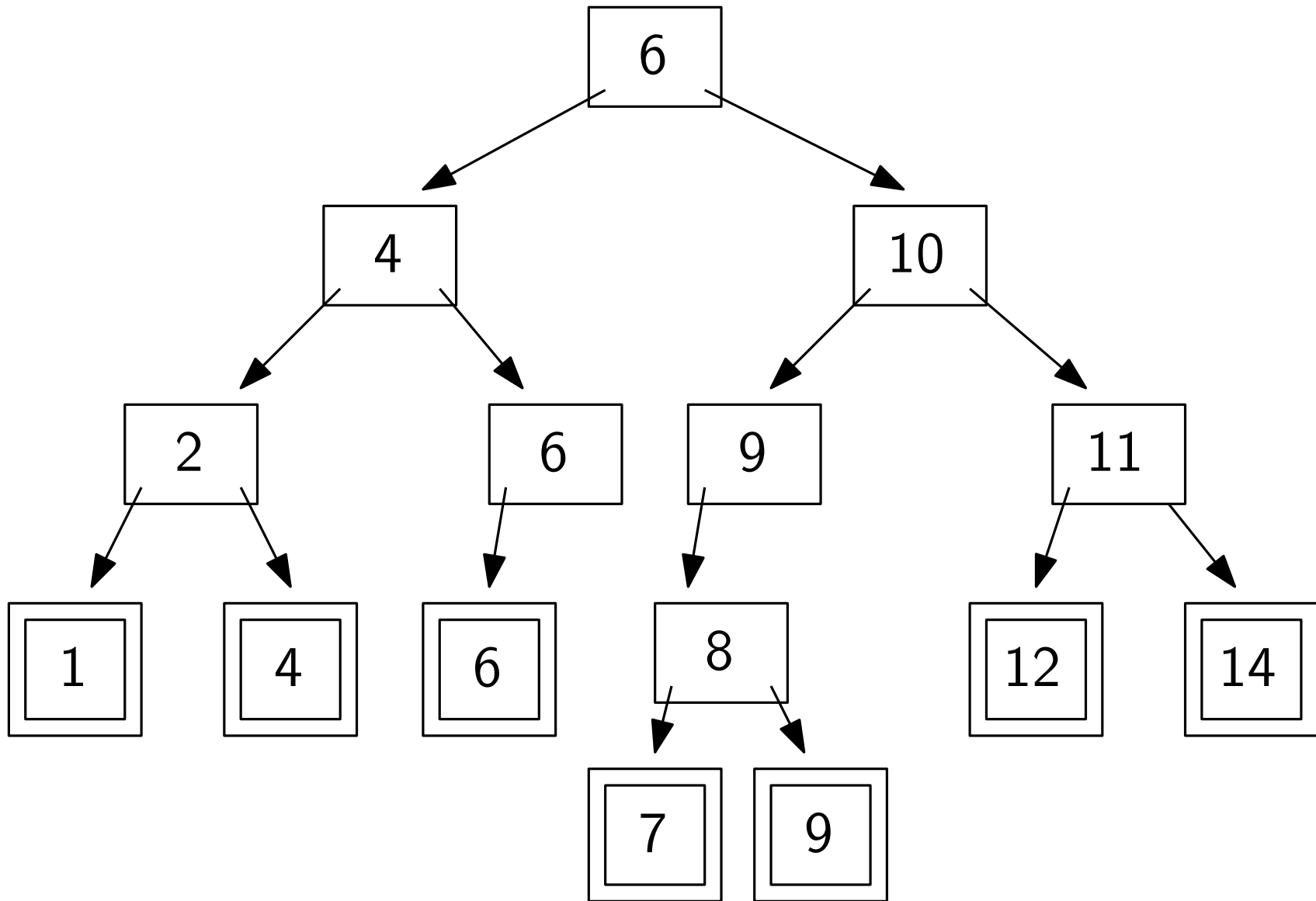
## Knotenorientierte Speicherung:

- speichern **pro Knoten** ein Element der Menge
- **Für Knoten  $v$  gilt:** Alle Schlüssel im linken TB von  $v$  sind kleiner als  $key(v)$ ; die aus rechtem TB von  $v$  sind größer.

## Blattorientierte Speicherung:

- Elemente nur **in Blättern**.
- In Knoten stehen Dummies.
- Schlüssel im linken TB von  $v \leq key(v) <$  Schlüssel rechts

# Bsp: Blattorientierte Speicherung



# knotenorientierte Speicherung: Suchen

**Suche( $x$ ):**

$u \leftarrow \text{Wurzel};$

$\text{found} \leftarrow \text{false};$

**while** ( $u$  existiert) **und** (not found) {  
    **if** ( $\text{key}(u) == x$ )  $\text{found} \leftarrow \text{true};$   
    **else if** ( $\text{key}(u) > x$ )  $u \leftarrow \text{lson}(u);$   
    **else**  $u \leftarrow \text{rson}(u);$   
}

# Einfügen in knotenorientierten Suchbaum

**Einfüge**( $x$ );

**Suche**( $x$ );

**if**  $x$  not found {

$u \leftarrow$  letzter besuchter Knoten in **Suche**( $x$ );

$u$  hat  $\leq 1$  Kind;

**if** ( $x < key(u)$ ) {

erzeuge neuen Knoten  $v \leftarrow lson(u)$  und  $key(v) \leftarrow x$ ;

}

**else** {

erzeuge Knoten  $v \leftarrow rson(u)$  und  $key(v) \leftarrow x$ ;

}

}

# Streichen im knotenorientierten Suchbaum

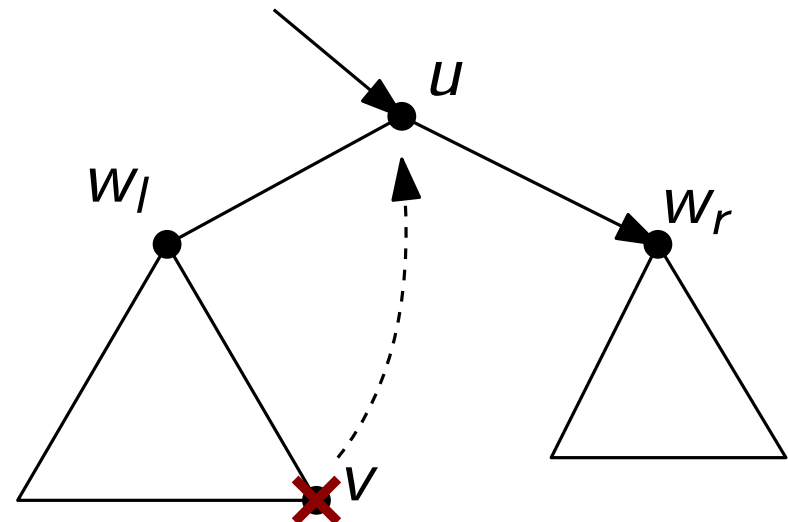
**Streiche**( $x$ );

**Suche**( $x$ ); ende in Knoten  $u$ , also  $key(u) = x$ ;

**if** ( $u$  Blatt) { [o.b.d.A. sei  $u = rson(p(u))$ ]  
    streiche  $u$ ;  $rson(p(u)) \leftarrow void$ ; }

**if** ( $u$  hat nur ein Kind  $w$ ) { [o.b.d.A. sei  $u = rson(p(u))$ ]  
     $rson(p(u)) \leftarrow w$ ; **Streiche**  $u$ ; }

**if** ( $u$  hat zwei Kinder  $w_l$  und  $w_r$ ) {  
    Suche  $v$  mit größtem  $key$  im linken Teilbaum von  $u$ ;  
    ersetze  $u$  durch  $v$ ;  
    streiche  $v$  unten;  
}



# Komplexität (Suchen, Einfügen, Streichen)

**Laufzeit:**  $O(\text{Höhe } h + 1)$

**Bem.:** Bei  $n$  Elementen ist  $h$  minimal  $\log n$ ,  
aber  $h$  kann auch  $= n$  sein

**Ziel:** Minimiere  $h$

1. Schlechter Fall kommt selten vor (zufällige Eingabe!)
2. Baue Baum ab und zu um  $\rightarrow$  Ausgewogenheit
3. balancierte Bäume

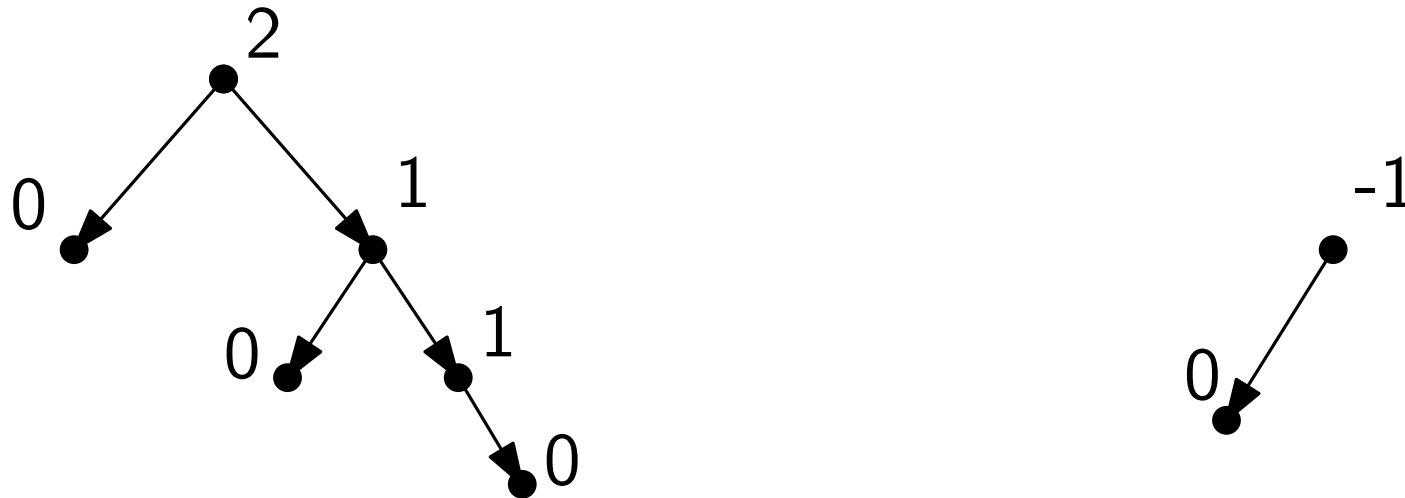
# Balancierte Bäume

## Höhenbalance:

Sei  $T$  Suchbaum mit Knoten  $u$  und Teilbäumen  $T_l$  und  $T_r$

$$Bal(u) \leftarrow \text{Höhe}(T_r) - \text{Höhe}(T_l)$$

Sei Höhe eines leeren Teilbaums =  $-1$ .

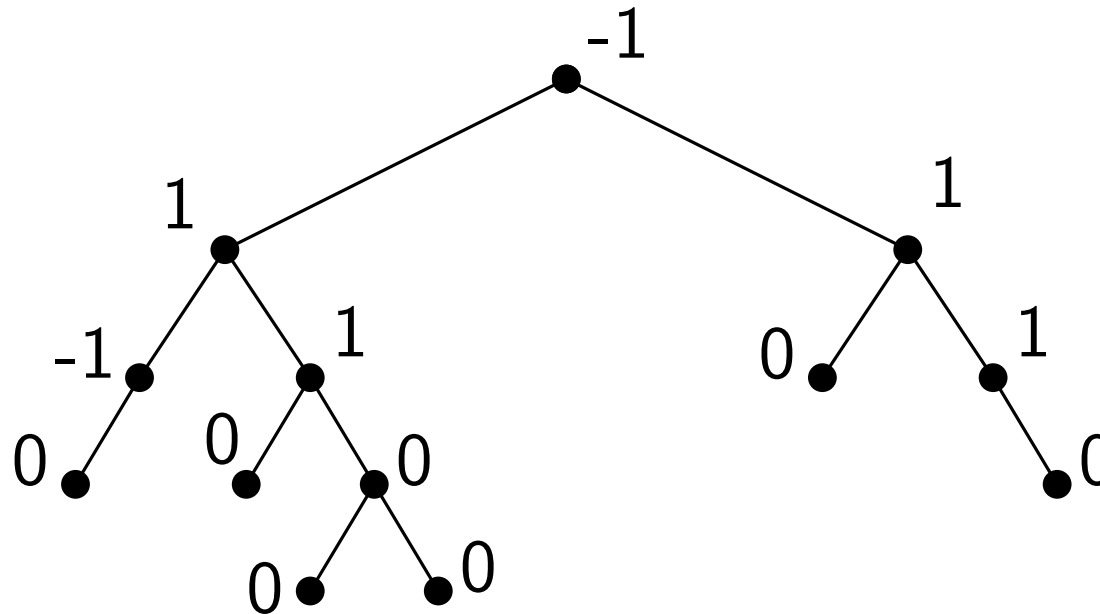




# AVL-Baum



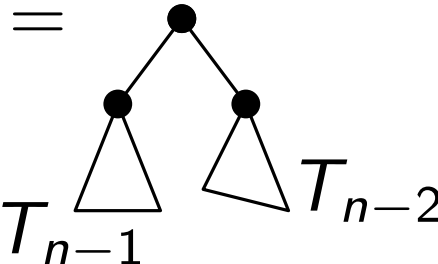
Suchbaum heißt **AVL-Baum**, falls  $|Bal(u)| \leq 1$  für alle Knoten  $u$ .

**Beispiel:**



# AVL-Bäume $\rightarrow$ Fibonacci-Bäume

**Fibonacci-Bäume**  $T_0, T_1, \dots$   
werden definiert durch:

- $T_0 =$  
- $T_1 =$  
- $T_n =$  

Vergleich mit  
**Fibonacci-Zahlen:**

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

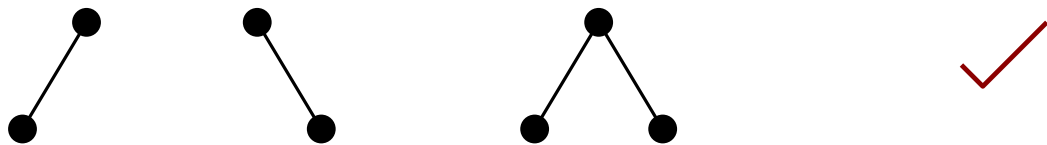
**Lemma:**  $T_h$  hat genau  $F_h$  Blätter. (Induktion!)

# Höhe von AVL-Bäumen

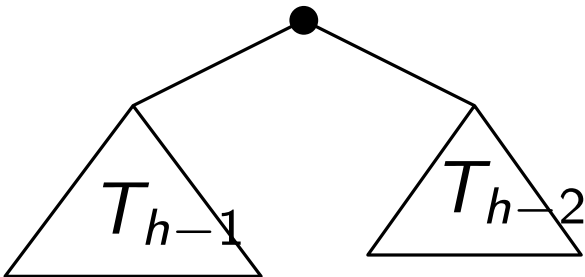
**Lemma:** AVL-Bäume der Höhe  $h$  haben mindestens  $F_h$  Blätter.

**Beweis:**

•  $h = 0$  : • ✓

•  $h = 1$  : 

•  $h \geq 2$  : Baum der Höhe  $h$  mit wenigsten Blättern setzt sich zusammen aus blattminimalen Bäumen der Höhen  $h - 1$  und  $h - 2$ .



$T_h$  hat nach Induktionsannahme  
 $\geq F_{h-1} + F_{h-2}$  Blätter.

# Höhe von AVL-Bäumen

**Fibonacci-Zahlen**  $F_h = \frac{\alpha^h - \beta^h}{\sqrt{5}}$ , mit  $\alpha = \frac{1+\sqrt{5}}{2}$  und  $\beta = \frac{1-\sqrt{5}}{2}$

**Lemma:** AVL-Bäume mit  $n$  Knoten haben Höhe  $O(\log n)$ .

**Beweis:** AVL-Baum  $T$  hat  $\leq n$  Blätter

$$\text{also } F_h = \frac{\alpha^h - \beta^h}{\sqrt{5}} \leq n$$

$$\text{weil } |\beta| < 1 \text{ ist gilt } \frac{\alpha^h - \beta^h}{\sqrt{5}} \geq \frac{\alpha^h}{2\sqrt{5}}$$

$$\Rightarrow \alpha^h \leq 2\sqrt{5} \cdot n$$

$$\Rightarrow h \log \alpha \leq \log(2\sqrt{5}) + \log n$$

$$\Rightarrow h \leq \frac{\log(2\sqrt{5}) + \log n}{\log \alpha} = O(\log n)$$



# Einfügen/Streichen in AVL-Bäumen

Durch **Einfüge**( $x$ ) können Balance einiger Vorfahren Werte  $\pm 2$  annehmen.

Sei  $u$  der tiefste dieser Vorfahren. Sei o.B.d.A.  $bal(u) = 2$ . Haben  $x$  also irgendwo rechts eingefügt.

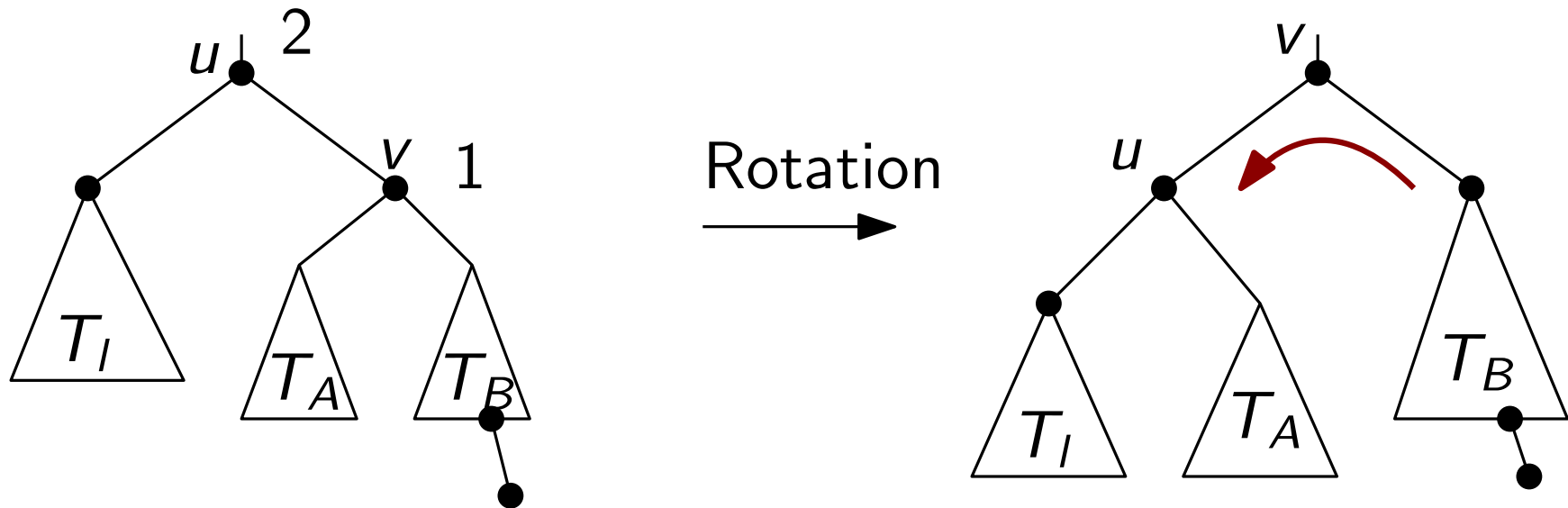
Betrachten  $u$  und sein rechtes Kind  $v$ .

Beachte:  $Bal(v) \neq 0$ , da  $Höhe(v)$  geändert.

**Fall 1:**  $Bal(v) = 1$  oder **Fall 2:**  $Bal(v) = -1$

# Rebalancieren in AVL-Bäumen

**Fall 1:**  $Bal(v) = 1$  (rechts eingefügt)



Rotation bewahrt links-rechts-Ordnung:  $T_L < u < T_A < v < T_B$

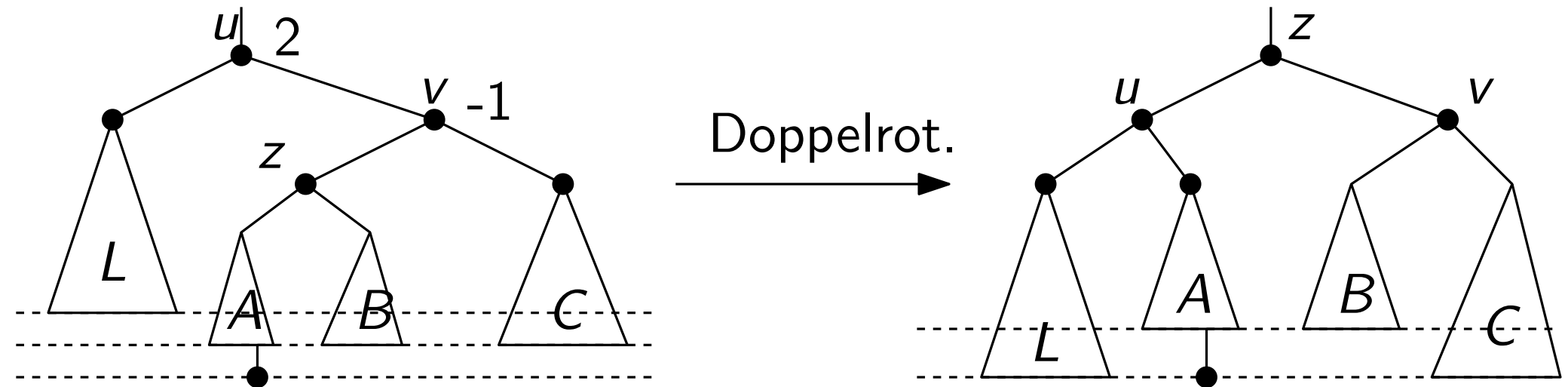
Balancen von  $u$  und  $v$  werden 0.

Balancen in Unterbäumen bleiben unverändert.

Höhe( $v$ ) nach Rotation = Höhe( $u$ ) vor **Einfüge**( $x$ )

# Rebalancieren in AVL-Bäumen

**Fall 2:**  $Bal(v) = -1$  (links eingefügt)



$z$  wird um 2 hochgehoben.  $Bal(z)$  danach 0.

$A$  und  $B$  könnten Rollen tauschen ( $x$  in  $B$  eingefügt).

# Einfügen + Rebalancieren

## Algorithmus:

- Füge  $x$  ein an einem neuen Blatt  $w$ . Folge dem Pfad von  $w$  zur Wurzel und berechne alle Balancen neu. (WIE?)
- Tritt Balance  $-2/+2$  auf, rebalanciere mit Rotation/Doppelrotation.

Wie oft macht man Rotation/Doppelrotation ?

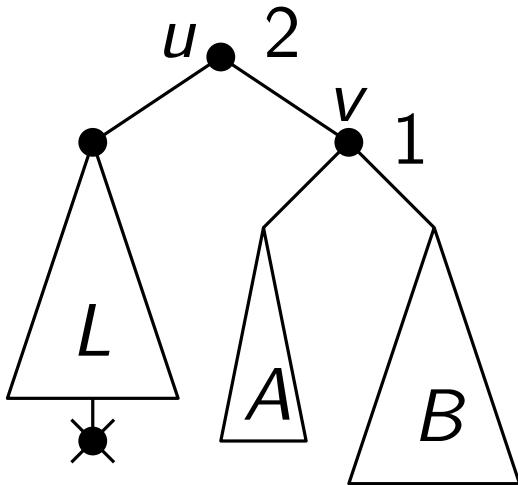
Höchstens einmal!



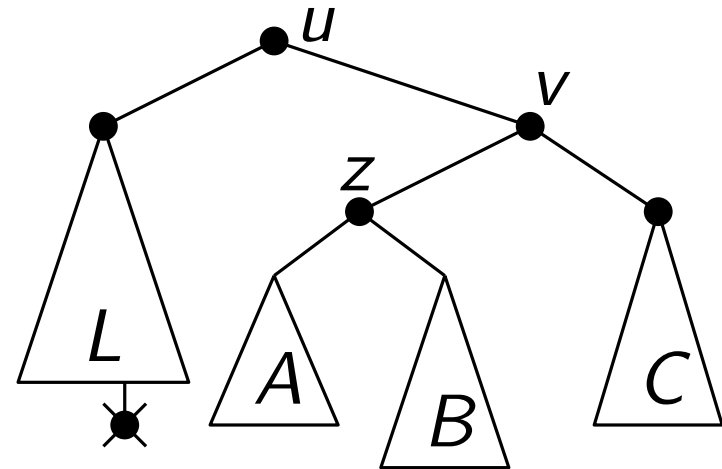
# Streichen in AVL-Bäumen

Wir streichen Knoten  $w$ . Ang. es gibt Knoten mit Balance  $\pm 2$ .  
Sei  $u$  der tiefste dieser Knoten (o.B.d.A.  $Bal(u) = 2$ ).  
Sei  $v$  das rechte Kind von  $u$ .

**1. Fall:**  $Bal(v) = 1$



**2. Fall:**  $Bal(v) = -1$



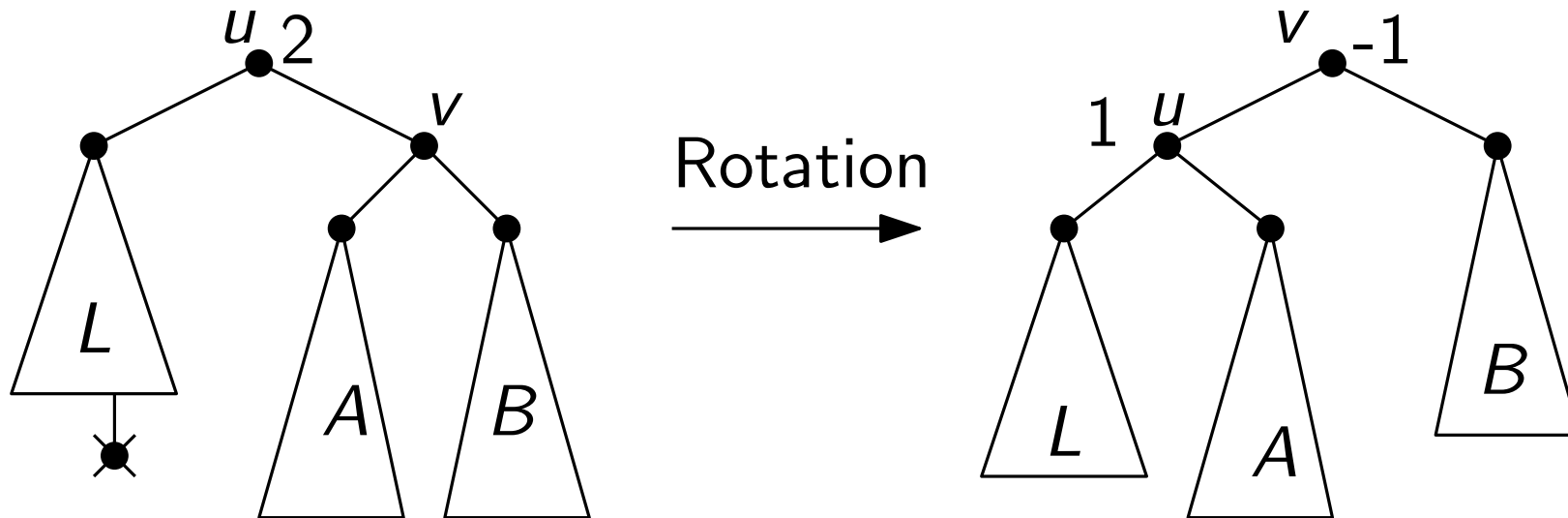
Rotation! Siehe Einfügen Fall 1. Doppelrotation! Siehe Fall 2.

**Beachte:** Höhen ändern sich.

Rebalancieren eventuell nach oben fortsetzen!

# Streichen in AVL-Bäumen

**Fall 3:**  $Bal(v) = 0$



Gesamthöhe hat sich nicht geändert.  
Keine Fortsetzung nach oben.

**Beachte:** Balance-Änderungen für  $u$  und  $v$

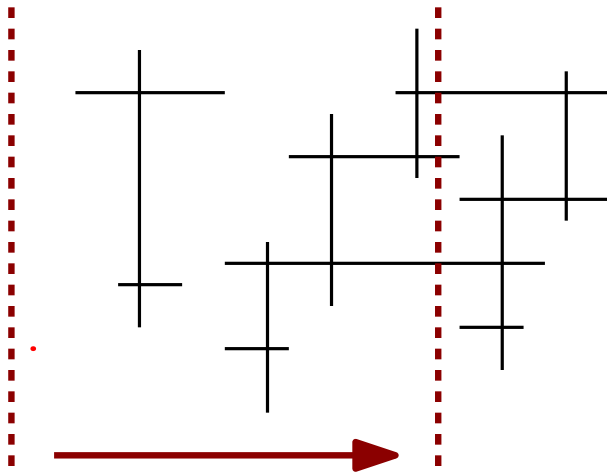
# Balancierte Bäume

## Satz:

Mit balancierten Bäumen kann man in Zeit  $O(\log n)$  die Operationen Suchen / Einfügen / Streichen durchführen, wobei  $n$  die Zahl der Elemente ist.

# Anwendung: Plane-Sweep-Algorithmus

Berechne Schnitte von  $n$  achsenparallelen Liniensegmenten



Bestimme die Zahl der  
Schnittpunkte!

## Annahmen (o.B.d.A):

allgemeine Lage, paarw. versch. Endpunkte

1. **Ansatz:** berechne Schnitt zwischen allen Paaren  $\rightarrow O(n^2)$
2. **Ansatz:** Plane-Sweep: Besenlinie überstreicht die Szene

# Plane-Sweep

## Idee:

Überstreiche die Szene mit einer vertikalen Linie  $L$   
Halte an best. Punkten an und berechne das nötige (Events)

## $x$ -Struktur:

**Event-Queue.** Halte geordnete Liste der Endpunkte  
Arbeite die Events nacheinander ab.  $\rightarrow$  statische Liste.  
an jedem Event  $x$  halte die Liste der aktiven Segmente.

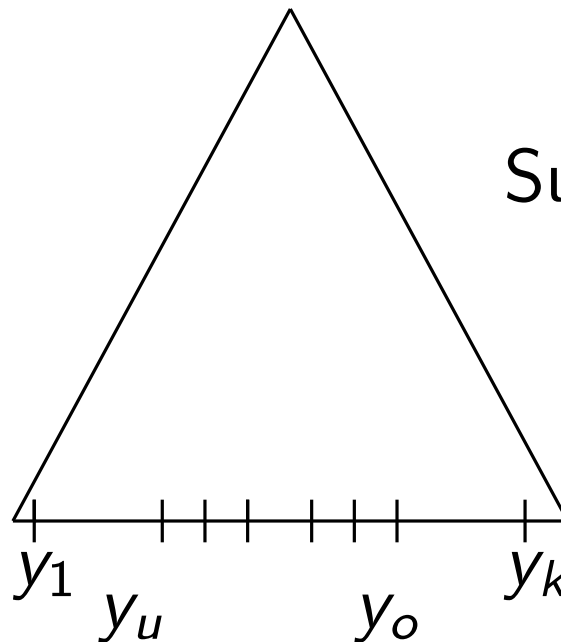
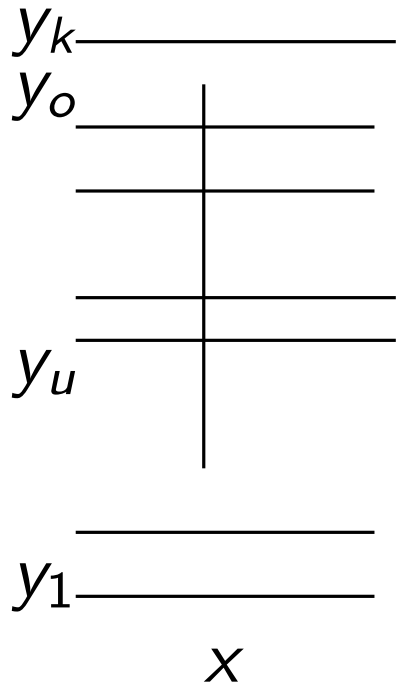
## $y$ -Struktur:

**Zustand der Sweep-Line  $L$  an Position  $x$**   
Speichere horizontale Segmente, die  $L$  gerade kreuzen.  
 $\rightarrow$  benutze AVL-Baum (Segm. nach  $y$  geordnet)  
 $\rightarrow$  Einfügen/Streichen in  $O(\log n)$

An vertikalen Segmenten passiert die Schnittpunktberechnung!

# Plane-Sweep

'Schnitt-Event' bei vertikalem Segment  $s(x, y_u, y_o)$



Suchbaum nach  $y$ -Koord.

blattorientiert

Wieviele Elemente gibt es zwischen  $y_u$  und  $y_o$ ?

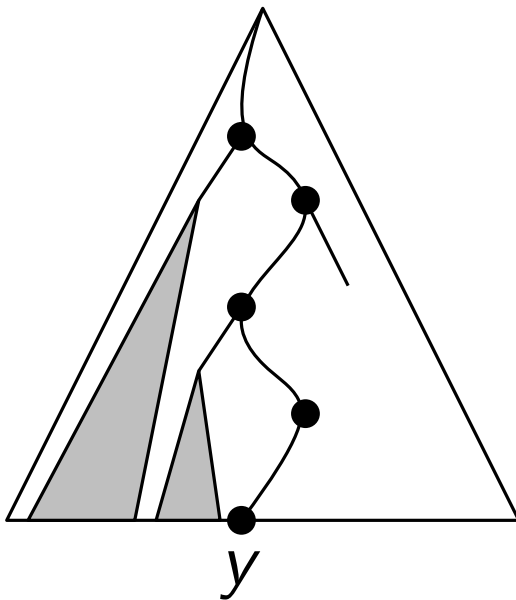
→ berechne Ränge von  $y_u$  und  $y_o$ !

# Berechne Ränge !

$\text{Rang}(y) = \# \text{ Segmente} < y$

wollen  $\#$  Schnittpunkte von Segm.  $s = \text{Rang}(y_o) - \text{Rang}(y_u)$

Bestimme  $\text{Rang}(y)$ .



Merke an jedem Knoten  $v$  die Zahl der Elemente im linken TB. Name:  $lcount(v)$

- Laufe Suchpfad nach  $y$  runter.
- Biegt man an Knoten  $v$  nach rechts:  
addiere  $lcount(v)$
- biegt man nach links: mache nix

das ergibt den  $\text{Rang}(y)$  in Zeit  $O(\log n)$

# Plane-Sweep

## Satz:

In Zeit  $O(n \log n)$  kann man mit Plane-Sweep die Anzahl der Schnittpunkte von  $n$  achsenparallelen Liniensegmenten in der Ebene bestimmen.

## Bemerkungen:

- Die Annahmen über allgemeine Lage der Segmente waren unnötig. Andernfalls wird's komplizierter.
- Die Segmente müssen nicht unbedingt achsenparallel sein.
- Plane-Sweep gibts auch z.B. für konvexe Hülle und auch in 3D - da heisst es dann Space-Sweep.



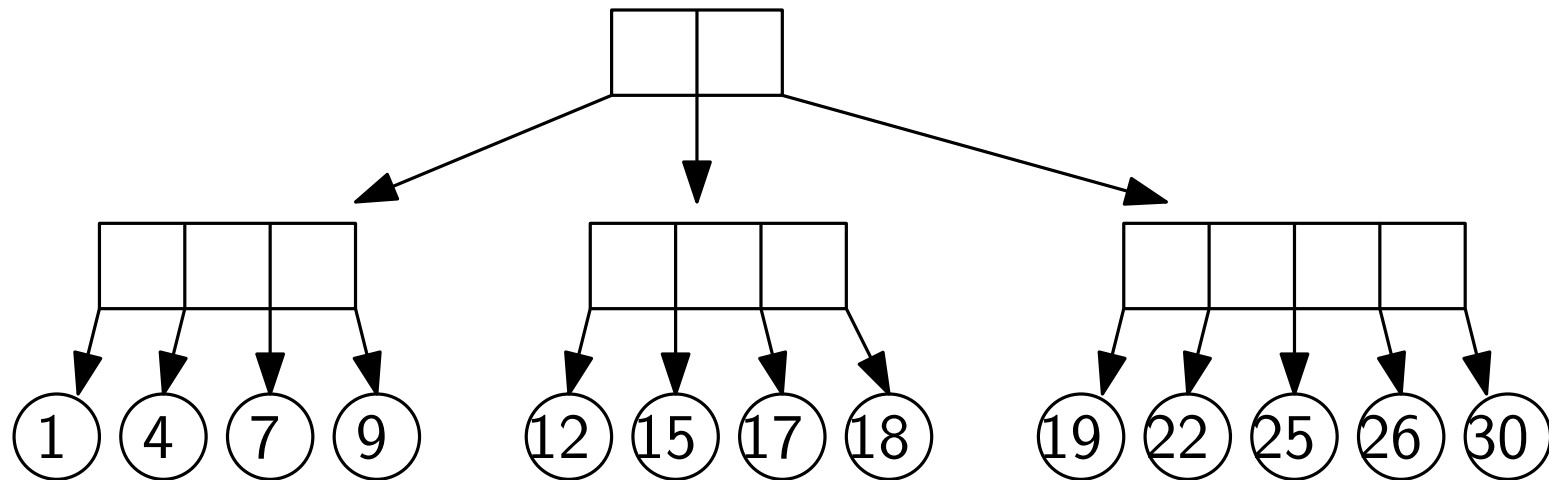
# Datenstrukturen 2: Balancierte Suchbäume II

2+3/2/2021

# B-Bäume

**B-Baum der Ordnung  $k \geq 2$**  ist ein Baum, mit folgenden Eigenschaften:

1. alle Blätter haben gleiche Tiefe
2. die Wurzel hat  $\geq 2$  Kinder, alle anderen Knoten haben  $\geq k$  Kinder
3. innere Knoten haben  $\leq 2k - 1$  Kinder



# B-Bäume

## Lemma:

Sei  $T$  ein B-Baum der Ordnung  $k$  mit  $n$  Blättern und Höhe  $h$ . Dann gilt:  $2k^{h-1} \leq n \leq (2k-1)^h$

## Beweis:

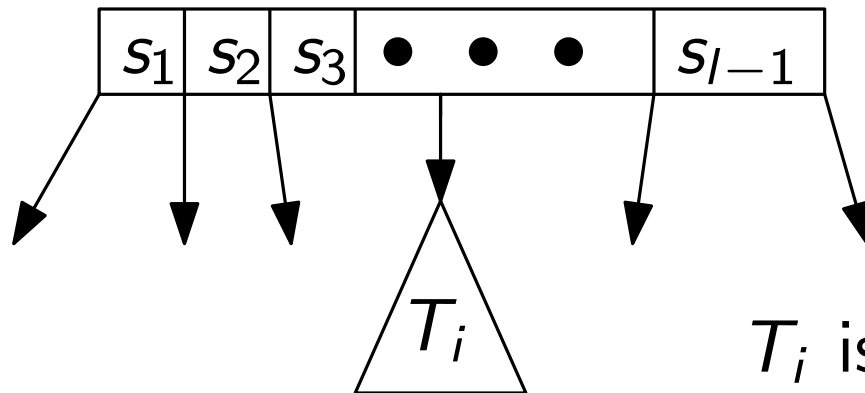
- Zahl der Blätter ist **minimal**, wenn innere Knoten Minimalzahl an Kindern haben.  $\rightarrow 2k^{h-1} \leq n$ .
- Blattzahl ist **maximal**, wenn Knoten Maximalzahl an Kinder haben.  $\rightarrow n \leq (2k-1)^h$ .

Also gilt (mit Logarithmieren):

$$\log_{(2k-1)} n \leq h \leq 1 + \log_k(n/2)$$

# B-Bäume

**Regeln:** Seien  $s_1, \dots, s_{l-1}$  Schlüssel in Knoten  $u$ .



$l - 1$  Schlüssel  
 $l$  Kinder

$T_i$  ist  $i$ -ter Unterbaum

Für alle  $v \in T_i$  und Schlüssel  $s$  in Knoten  $v$  gilt:

- falls  $i = 1$ , ist  $s \leq s_1$
- falls  $1 < i < l$ , ist  $s_{i-1} < s \leq s_i$
- falls  $i = l$ , ist  $s_{l-1} < s$

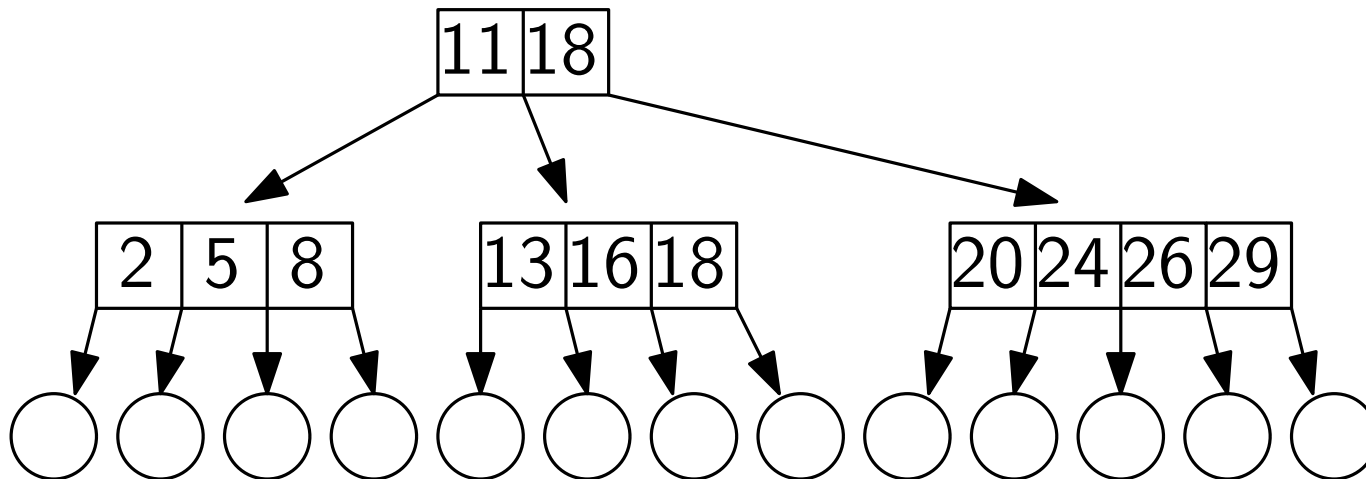
Speicherung knotenorientiert: keine Schlüssel in Blättern.

# B-Bäume

## Zugriff( $a, S$ ):

- Starte in Wurzel. Kommt  $a$  als Schlüssel vor, fertig.
- Wenn nicht, laufe in den richtigen Unterbaum, wo  $a$  stehen muß.
- Iteriere!

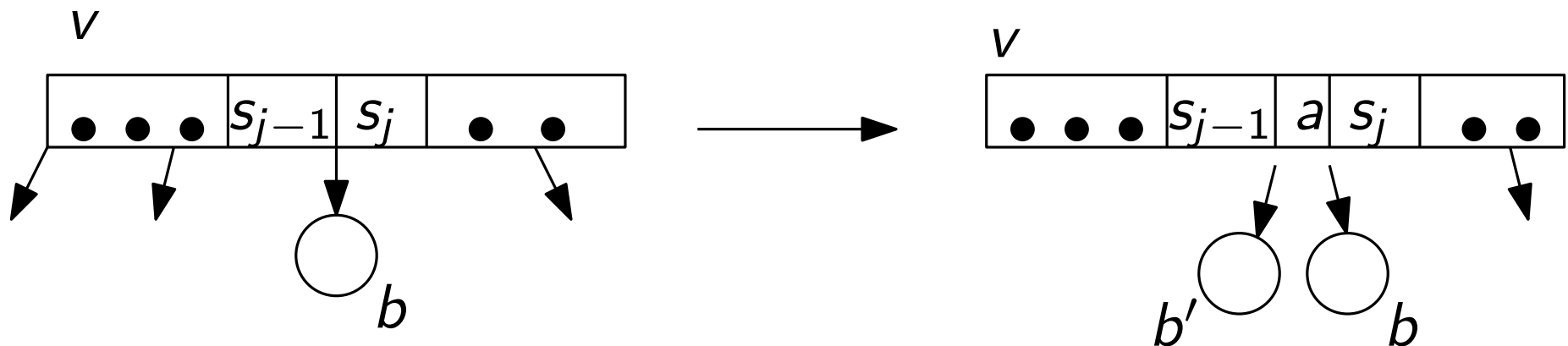
**Laufzeit:**  $O(k \cdot \log_k n)$



# B-Bäume

## Einfüge( $a, S$ ):

- Mache zuerst **Zugriff**( $a, S$ ). Am Blatt  $b$  mit parent  $v$  mit  $l$  Kindern endet Suche.
- Sei  $s_j$  der kleinste Schlüssel in  $v$  mit  $a < s_j$ , falls er existiert ( $b$  ist  $j$ -tes Blatt).
- Füge  $a$  als Schlüssel in  $v$  und Blatt  $b'$  links von  $a$

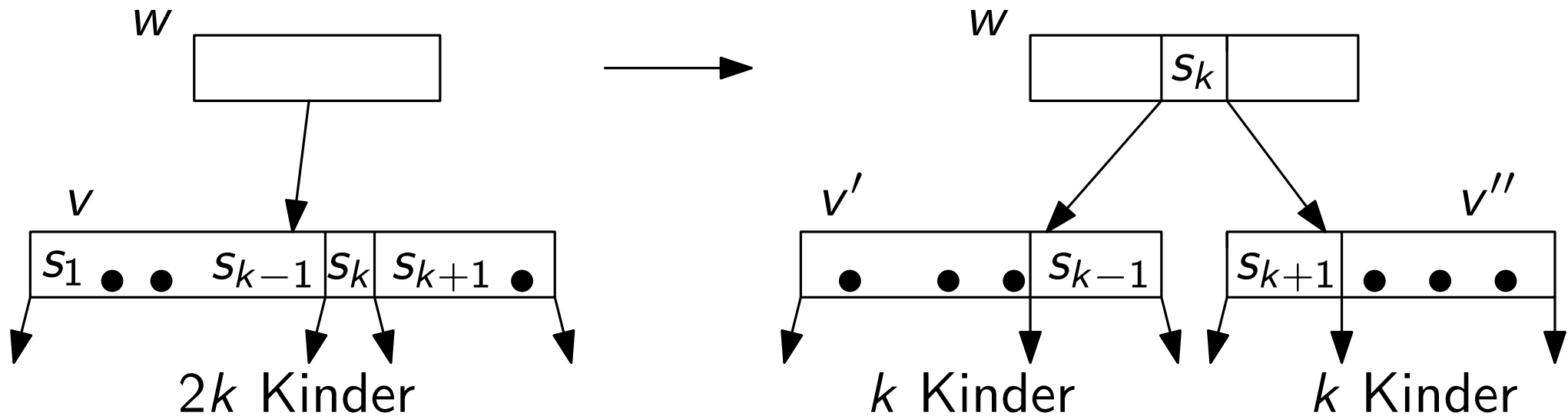


**Beachte:**  $v$  hat nun einen Schlüssel mehr!

# B-Bäume: Einfügen

- Falls  $v$  danach  $\leq 2k - 1$  Kinder hat, alles ok.
- Falls  $v$  danach  $2k$  Kinder hat, ist das zuviel!

**Idee:** Spalte  $v$  auf in 2 Knoten  $v'$  und  $v''$  mit jeweils  $k$  Kindern.



**Beachte:**  $w$  hat einen Schlüssel mehr! Iteriere nach oben!

**Wo endet das?**

# B-Bäume: Einfügen

- Iteration nach oben endet spätestens an der Wurzel.
- Wird diese aufgespalten, gibt es eine neue Wurzel mit 2 Kindern.
- Nur in diesem Fall wächst die Höhe des Baumes (um 1).



# B-Bäume: Streichen

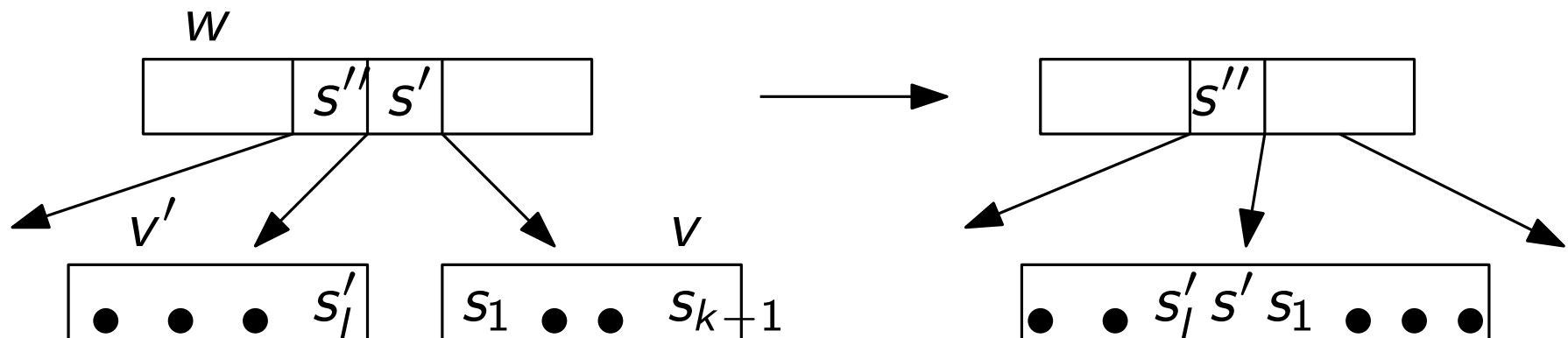
## **Streiche( $a, S$ ):**

- Mache **Zugriff**( $a, S$ ), und finde  $a$  in Knoten  $u$  mit  $j$ -tem Schlüssel. Sei  $T_j$  der Unterbaum links von  $a$ . Besteht  $T_j$  nur aus Blatt, lösche Schlüssel aus  $u$ .  
→ Kinderzahl von  $u$  wird kleiner.
- Ist  $T_j$  größer, so sei  $s$  der rechteste Schlüssel in  $T_j$ . Ersetze  $a$  durch  $s$ , und streiche  $s$  aus  $T_j$ . Auch hier wird Kinderzahl kleiner.

Was tun, wenn Kinderzahl von Knoten  $v$  kleiner als  $k$ ?

# B-Bäume: Streichen

- Ist  $v$  Wurzel mit nur noch einem Kind, streiche Wurzel.  
→ Höhe erniedrigt sich. Baum bleibt B-Baum.
- Ist  $v$  nicht Wurzel, so versuche  $v$  mit Geschwisterknoten  $v'$  zu vereinigen.



**Beachte:** neues  $v$  hat  $\leq 3k - 2$  Kinder. **Zuviele?**

**Ausweg:** Spalte  $v$  nochmals, so dass neue Knoten zulässig.

Geht das immer? **Ja!**

# B-Bäume

## **Satz:**

B-Bäume der Ordnung  $k$  unterstützen Zugriff, Einfügen und Streichen in Zeit  $O(k \log_k n)$  bei  $n$  Schlüsseln.

Wie groß wählt man  $k$ ?

**Kommt drauf an:** Wähle  $k$  so, dass bei Paging von Platte auf Hauptspeicher jeweils ein Knoten komplett auf eine Seite passt!

**Seitengröße:** typischerweise 4MB

# $(a, b)$ -Bäume, hier $(2, 4)$ -Bäume

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2, b \geq 2a - 1$ .

$T$  heißt  $(a, b)$ -Baum, falls gilt:

- (a) alle Blätter haben gleiche Tiefe
- (b) alle inneren Knoten haben  $\leq b$  Kinder
- (c) alle (außer Wurzel) haben  $\geq a$  Kinder
- (d) Wurzel  $r$  hat  $\geq 2$  Kinder.

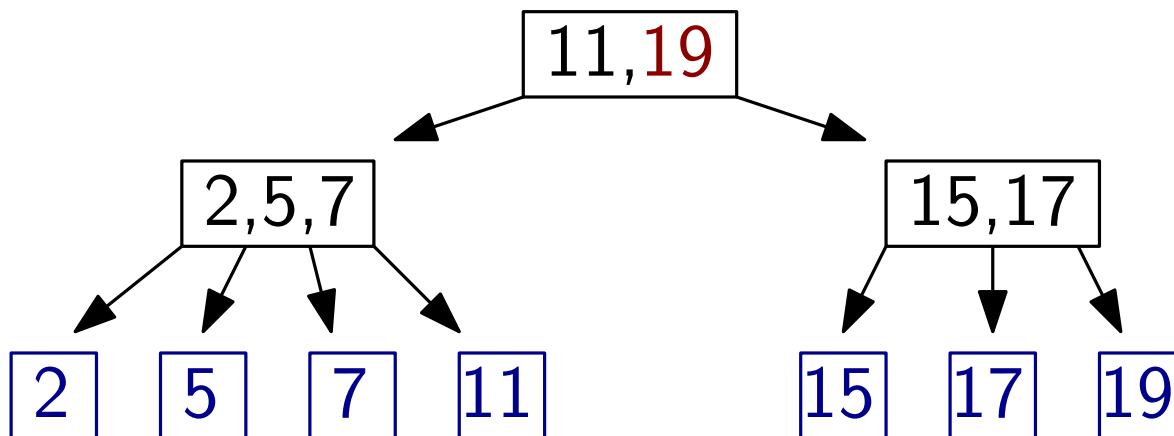
Betrachte  $(2, 3)$ -Bäume und  $(2, 4)$ -Bäume. Warum?

→ Vergleiche Ergebnisse B-Bäume.

# (2, 4)-Bäume

Speichern Menge  $S = \{x_1 < x_2 < \dots < x_n\}$  wie folgt:

- Speichere  $S$  in **Blättern** von links nach rechts (blattorientiert)
- Ein innerer Knoten  $v$  mit  $d$  Kindern hat  $s_1, \dots, s_{d-1}$  Schlüssel mit  $s_i$  ist Inhalt des rechtesten Blatts im  $i$ -ten Unterbaum.
- Speichere noch **maximales Element** von  $S$  in Wurzel (Zugriff).



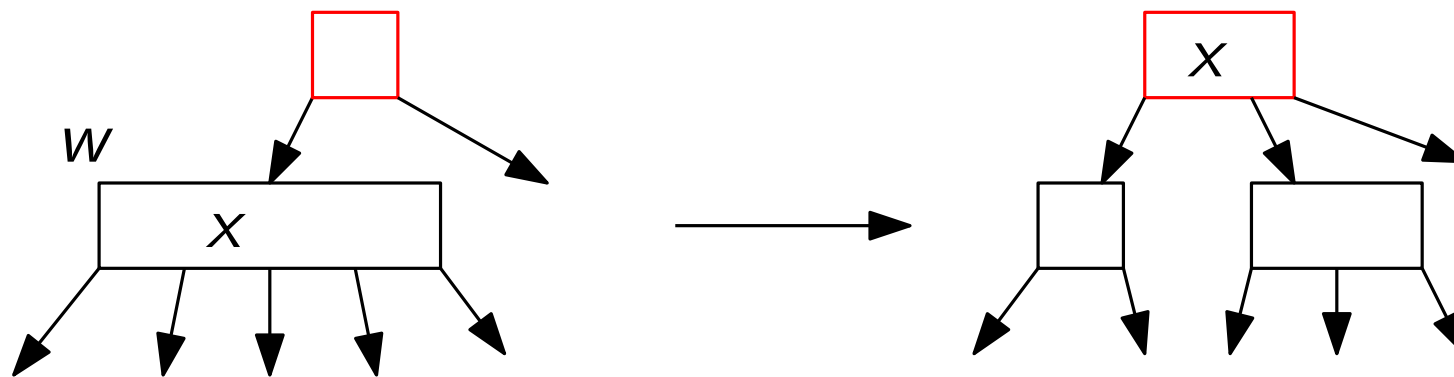
# (2, 4)-Bäume

**Lemma:** Für Höhe  $h$  eines (2, 4)-Baums mit  $n$  Blättern gilt:

$$1/2 \cdot \log n \leq h \leq \log n$$

**Operationen:** Zugriff( $a, S$ )  
Einfüge( $a, S$ )

Gibt es Knoten  $w$  mit 5 Kindern  $\rightarrow$  **Spalte**( $w$ )



**Laufzeit:**  $O(1 + \text{\#Spaltungen})$

## (2,4)-Bäume: Streiche( $a, S$ )

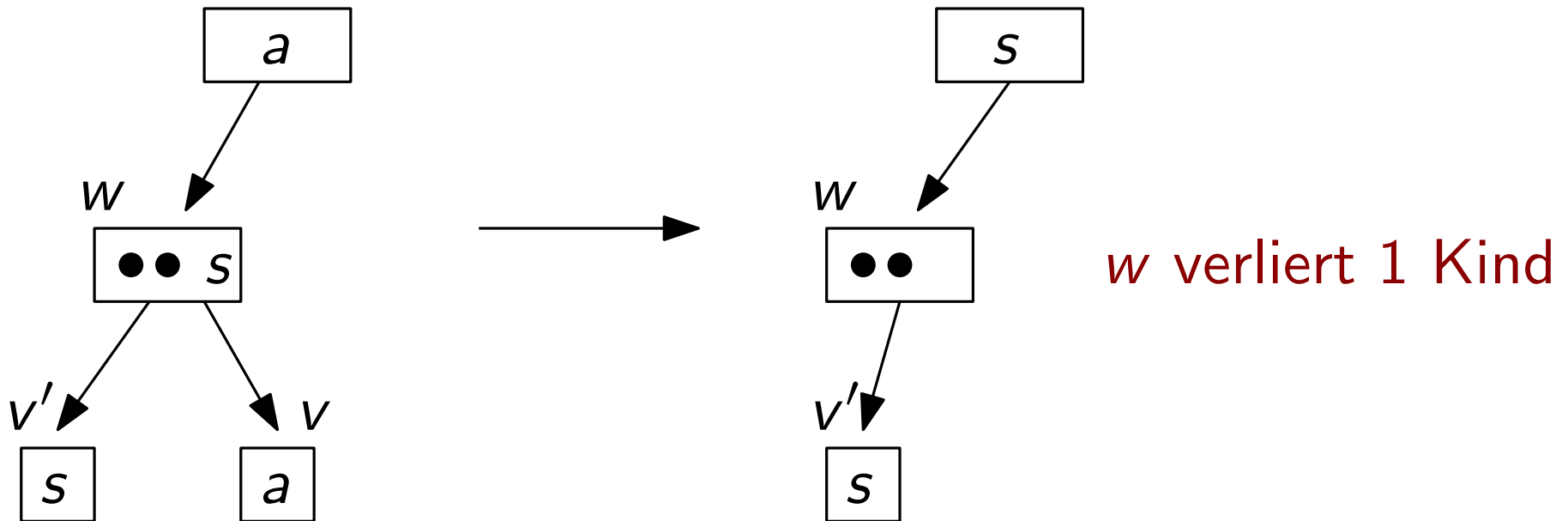
Mache zuerst **Zugriff**( $a, S$ ).  $\rightarrow a$  in Blatt  $v$

1.  $a$  steht auch in  $\text{parent}(v) = w$ .

Lösche  $v$ , streiche  $a$  aus  $w$ .

2.  $a$  steht nicht in  $\text{parent}(v) = w$  (rechtestes Kind).

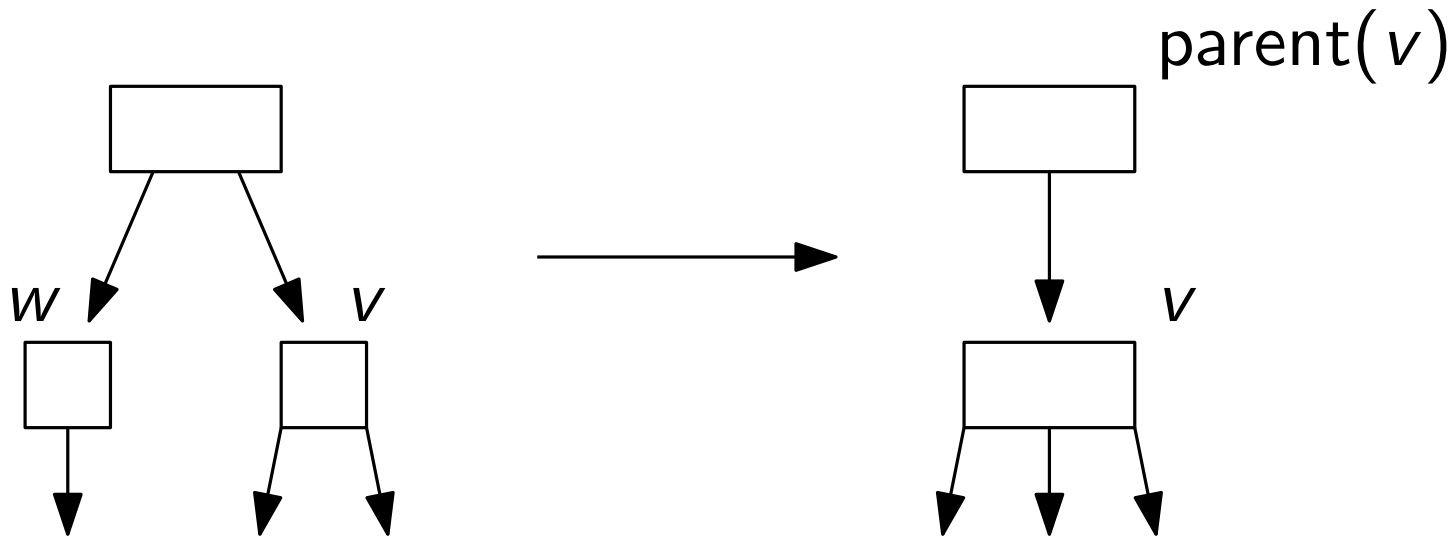
$v'$  sei linker Bruder von  $v$  mit Schlüssel  $s$



# (2,4)-Bäume: Streiche(a,S)

$w$  hat nur noch ein Kind  $\rightarrow$  Verschmelze oder Stehle

1. Geschwister  $v$  von  $w$  hat genau 2 Kinder.

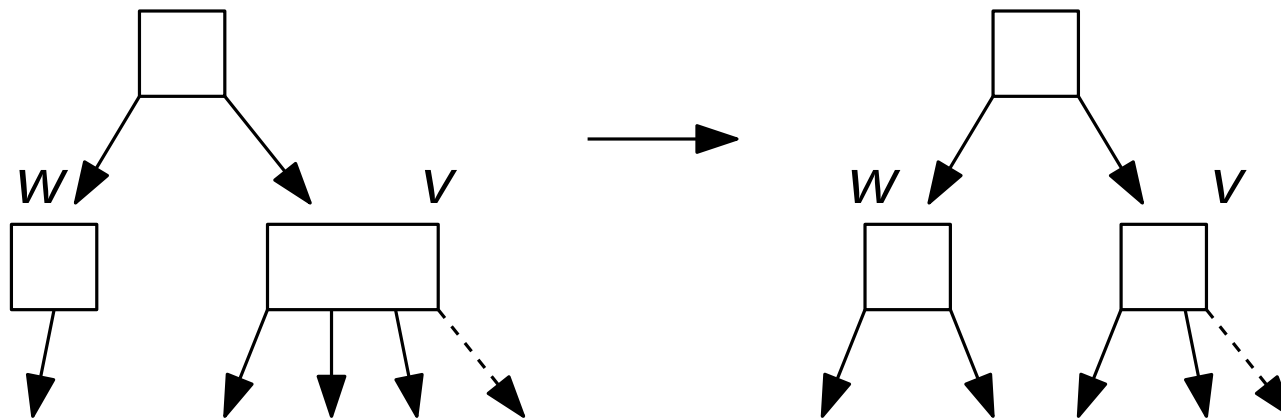


$\text{parent}(v)$  verliert ein Kind  
 $\rightarrow$  weiter nach oben



# (2,4)-Bäume: Streiche(a,S)

2. Geschwister  $v$  von  $w$  hat  $\geq 3$  Kinder  $\rightarrow$  **Stehle!**



keine Kinder verloren, kein Hochlaufen!

Laufzeit für Streichen:  $O(1 + \# \text{Verschmelzungen})$

# Amortisierte Analyse

## Beobachtung:

- Laufzeit  $T_{tat}(Op)$  schwankt zwischen  $O(1)$  und  $O(\log n)$ .
- Es hängt vom Zustand des Baums ab, wie hoch man laufen muss.

## Vorgehen:

- wir halten uns ein Bankkonto (je nach Zustand der Datenstr.)
- billige Operationen zahlen was ein (z.B. 1 Euro)
- teure Operationen heben ab (z.B.  $\log n$  Euro)

**Entscheidend:**  $T_{tat} + \Delta B \leq T_{amort}$

**Ziel:**  $T_{amort}$  vielleicht  $O(1)$

**Idee:** Wir verteilen Bankkontogeld auf dem Baum.

# Amortisierte Analyse von Einfügen/Streichen

**Lemma:** In (2,4)-Baum sind amortisierte Kosten von Einfügen bzw. Streichen  $O(1)$ .

**Beweis:**

Wähle Potential (Bankkontobetrag) wie folgt:

$$\begin{aligned} pot(T) = & 2 \cdot \# \text{Grad-1-Knoten} + 1 \cdot \# \text{Grad-2-Knoten} \\ & + 0 \cdot \# \text{Grad-3-Knoten} + 2 \cdot \# \text{Grad-4-Knoten} \\ & + 4 \cdot \# \text{Grad-5-Knoten} \end{aligned}$$

**Invariante:**

- $pot(T) \geq 0$  für alle Bäume  $T$
- Wird Spalte/Verschmelze/Stehle auf  $v$  angewendet, sind andere Knoten in Ordnung (2,3,4 Kinder)
- Vor Einfügen/Streichen sind alle Knoten in Ordnung.

# Amortisierte Analyse von Einfügen/Streichen

**Es gilt:** Spalte/Verschmelze/Stehle kosten jeweils  $O(1)$ .  
(tatsächl. Kosten einer Einzeloperation)

**Beh.:** Spalte/Verschmelze verringern Potential, Stehle erhöht es nicht.

**Stehle:**  $w$  von 2 auf 1,  $v$  von 0 auf 1 oder 2 auf 0.

**Spalte:**  $w$  von 4 auf 0 und 1.  $\text{parent}(w)$  geht um  $\leq 2$  hoch  
→ sinkt insgesamt.

**Verschmelze:**  $w$  von 2 und  $v$  von 1 auf insgesamt 0.  
 $\text{parent}(w)$  geht eventuell  $\leq 1$  hoch → sinkt insgesamt.

# Amortisierte Analyse: Einfügen/Streichen

**hier für Einfügen:**

amort. Laufzeit = tat. Kosten + Potentialerhöhung

1. tatsächliche Kosten(Einfügen) 1 + Potentialerhöhung  $\leq 2$

2. Folge von Spalte-Operationen

tatsächliche Kosten  $f$  + Potentialerhöhung  $\leq -f$

---

insgesamt  $\leq 3$

**Streichen?** Selbst machen!

# Anwendung: Sortieren fast-sortierter Listen

**Prinzip:** Sortieren durch Einfügen in (2,4)-Baum.

Starte mit leerem Baum.

Mache  $n \times$  Zugriff  $O(n \log n)$  und  $n \times$  Einfügen  $O(n)$ .

- Mache Zugriff schneller. Geht, wenn Eingabe 'bisschen' vorsortiert ist.
- Sei  $x_1, \dots, x_n$  Eingabefolge.
- $F = |\{(i, j) | i < j \text{ und } x_i > x_j\}|$  ist Anzahl der **Inversionen** und  $0 \leq F \leq n^2/2$ .

**Satz:** Sortieren geht in Zeit  $O(n \log(\frac{F}{n} + 1))$

**bedeutet:**  $F \approx n^2 \Rightarrow T = O(n \log n)$

$F \approx n \Rightarrow T = O(n)$

$F \approx n \log n \Rightarrow T = O(n \log \log n);$

$F \approx n^{3/2} \Rightarrow T = O(n \log n)$

# Sortieren fast-sortierter Listen

## Beweis:

Sei  $f_i = |\{j | i < j \text{ und } x_i > x_j\}|$

Dann ist  $F = \sum_i f_i$ .

Füge Elemente in umgekehrter Reihenfolge in den Baum, startend mit  $x_n$ .

Bei **Einfüge**( $x_i$ ) sind  $x_{i+1}, \dots, x_n$  schon drin. Starte ganz links, laufe hoch und drehe um und laufe runter.

Grund: Wenn  $x_i$  klein, läuft man nicht sehr hoch.

## Hochlaufen( $x_i$ ):

$v \leftarrow$  linkstes Blatt;

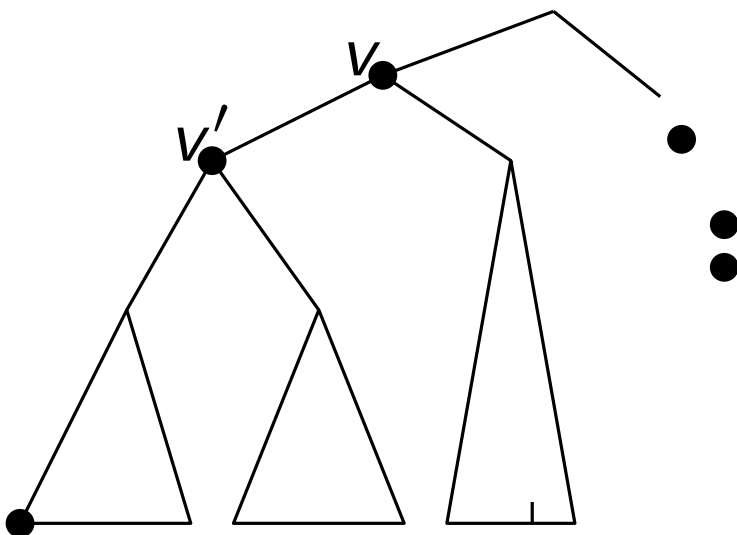
**while** ( $x_i > \text{Schlüssel}(\text{parent}(v))$ ) {  
     $v \leftarrow \text{parent}(v)$ ;}  
}

# Sortieren fast-sortierter Folgen

**Lemma:** **Einfüge** $(x_i)$  kostet amortisiert  $O(1 + \log f_i)$ .

**Beweis:** Wissen, eigentl. Einfügen kostet  $O(1)$  amortisiert.

- brauchen noch Lokalisieren, also Hochlaufen zur Höhe  $h$
- $x_i$  wird nach dem  $f_i$ -ten Element in die bisherige Liste eingefügt. Sei  $v$  der Umkehrpunkt, und  $v'$  das linkeste Kind von  $v$ .



Blätter unterhalb  $v'$   
sind kleiner  $x_i$ .

$$\Rightarrow 2^{h-1} \leq f_i$$

$$\Rightarrow h \leq 1 + \log f_i$$



# Sortieren fast-sortierter Folgen

⇒ **Gesamtkosten:**

$$\sum_{i=1}^n O(1 + \log f_i) = O(n + \sum_{i=1}^n \log f_i) = O(n + n \log \frac{F}{n})$$

**Satz:** Sortieren geht in Zeit  $O(n \log(\frac{F}{n} + 1))$ .

# Optimierungsmethoden: Dynamisches Programmieren

9+10/02/2021

# Dynamisches Programmieren

**Idee:** Optimale Lösung setzt sich zusammen aus optimalen Lösungen für Teilprobleme.

**Ansatz:** Löse Teilprobleme optimal. Kombiniere Lösungen und speichere beste Kombination. **Verwende gespeicherte Lösung für gleichartige Teilprobleme!**

**Beispiel:**

Sollen eine Stange  $s$  der Länge  $n$  zerschneiden.  
Preis für Stück der Länge  $i$  ist  $p_i$ .

**Maximiere Erlös  $r_n$ !**

Alternativen für Wahl des 1. Stückes:

1. Lass  $s$  ganz  $\rightarrow$  Erlös  $p_n$
2. 1. Stück Länge 1:  $\rightarrow$  Erlös  $p_1 + r_{n-1}$
3. 1. Stück Länge 2:  $\rightarrow$  Erlös  $p_2 + r_{n-2}$
- .... usw

**Maximiere über alle Möglichkeiten!**

# Dynam. Programmieren: Rekursion ?

Optimal ist also:  $Opt(n) = \max_k \{p_k + Opt(n - k)\}$

Berechne alle  $Opt(0), Opt(1), Opt(n - 1)$ .

Dann ist Berechnung von  $Opt(n)$  leicht.

→ Rekursiv: Eingabe ist Preisliste  $p[i], 1 \leq i \leq n$ .

**Schnitt( $n$ ):**

**if** ( $n = 0$ ) **return** 0;

$q \leftarrow -\infty$ ;

**for** ( $i = 1$  to  $n$ ) {

$q \leftarrow \max\{q, p[i] + \mathbf{Schnitt}(n - i)\}$ ;

**return**  $q$ ;

**Rekursion:**  $T(0) = 1, \quad T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

# Dynamisches Programmieren

## Rekursion:

$$T(0) = 1, T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + (2^n - 1) = 2^n$$

Beweisen durch Induktion

Jetzt **bottom-up**: 'von klein nach groß'

## Schnitt( $n$ ):

$r[0] \leftarrow 0;$

**for** ( $j = 1$  to  $n$ ) {

$q \leftarrow -\infty;$

**for** ( $i = 1$  to  $j$ ) {

$q \leftarrow \max\{q, p[i] + r[j - i]\};$

$r[j] \leftarrow q;$

**return**  $r[n];$

**Laufzeit:**  $T(n) = \sum_{j=1}^n j = \frac{n(n+1)}{2} = O(n^2)$

# Dynam. Programmieren: Von klein nach gross

## Rekonstruktion der optimalen Lösung:

Merke ersten Schnitt für Teilproblem der Länge  $j$ , also ersetze

$$q \leftarrow \max\{q, p[i] + r[j - i]\}$$

durch

```
if ( $q < p[i] + r[j - i]$ ) {  
     $q \leftarrow p[i] + r[j - i]$ ;  
     $s[j] \leftarrow i$ ;  
}
```

Also  $s[j] = i$ : Erster Schnitt an TP der Länge  $j$  ist  $i$  lang.

## Ausgabe:

```
while ( $n > 0$ ) {  
    print  $s[n]$ ;  
     $n \leftarrow n - s[n]$ ;  
}
```

# Dyn. Programmieren: Matrixmultiplikation

**Aufgabe:** Berechne  $A \cdot B$

**Voraussetzung:** #Spalten von  $A$  = #Zeilen von  $B$

```
for ( $i = 1$  to  $A$ .zeilen) {  
  for ( $j = 1$  to  $B$ .spalten) {  
     $c_{ij} \leftarrow 0$ ;  
    for ( $k = 1$  to  $A$ .spalten) {  
       $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$ ; }  
    }  
  }  
}  
return  $C$ ;
```

**Laufzeit:**  $O(A.\text{zeilen} \cdot B.\text{spalten} \cdot A.\text{spalten})$

**Jetzt:** Berechne  $A_1 \cdot A_2 \cdot \dots \cdot A_n$

# Matrixmultiplikation

**Jetzt:** Berechne  $A_1 \cdot A_2 \cdot \dots \cdot A_n$

**Gesucht:** Optimale Klammerung!

**Idee:** Problem  $A_i A_{i+1} \dots A_j$  geklammert  $(A_i \dots A_k)(A_{k+1} \dots A_j)$

Also für alle  $i \leq k \leq j$  berechne  $(A_i \dots A_k)$  und  $(A_{k+1} \dots A_j)$ , multipliziere  $(A_i \dots A_k)$  mit  $(A_{k+1} \dots A_j)$  und minimiere über  $k$ .

**Wichtig:**

Klammerung innerhalb beider Teilprobleme muss optimal sein.

**Also:**

- berechne min. Kosten für  $A_i \dots A_j$  für alle  $1 \leq i \leq j \leq n$
- $\min[i, j]$  ist Minimum aller Multiplikationen für  $A_i A_2 \dots A_j$
- Bestimme  $\min[1, n]$



# Matrixmultiplikation

Matrix  $A_i$  habe Dimension  $p_{i-1} \cdot p_i$ .

**rekursiv:**

- $\min[i, j] = \min_k \{ \min[i, k] + \min[k + 1, j] + p_{i-1} p_k p_j \}$
- $\min[i, j] = 0$  für  $i = j$

Speichere zusätzlich  $s[i, j] = k$  ab, mit  $k$  bestimmt  $\min[i, j]$

Formulierung mit 3-facher for-Schleife  $\longrightarrow$

# Matrixmultiplikation: Pseudocode

```
 $n \leftarrow \# \text{Matrizen} - 1;$   
for ( $i \leftarrow 1$  to  $n$ ) {  $\text{min}[i, i] \leftarrow 0;$  }  
for ( $l \leftarrow 2$  to  $n$ ) {  
  for ( $i \leftarrow 1$  to  $n - l + 1$ ) {  
     $j \leftarrow i + l - 1; \text{min}[i, j] \leftarrow \infty;$   
    for ( $k \leftarrow i$  to  $j + 1$ ) {  
       $q \leftarrow \text{min}[i, k] + \text{min}[k + 1, j] + p_{i-1}p_kp_j$   
      if ( $q < \text{min}[i, j]$ ) {  
         $\text{min}[i, j] \leftarrow q; s[i, j] \leftarrow k;$  }  
    }  
  }  
}  
return  $\text{min}, s;$ 
```

**Laufzeit:** 3 geschachtelte for-Schleifen  $\rightarrow O(n^3)$

# Matrixmult.: Laufzeit

**Genauer für  $l = 2, \dots, n - 1$ :**

$$\begin{aligned}\sum_{l=2}^{n-1} (l-1)(n-l) &= \sum_{l=1}^{n-2} l(n-l+1) = \\ &= \sum_{l=1}^{n-2} ln - l^2 - l = \dots \\ &= \dots\end{aligned}$$

# Greedy-Algorithmen

**Prinzip:** Entscheide lokal optimal

→ oft global optimal, oft aber auch nicht

**Beispiel 1:**

haben  $n$  Jobs  $a_1, \dots, a_n$  mit Startzeiten  $s_i$ , Endzeit  $e_i$ .

Jobs  $a_i, a_j$  kompatibel, falls Intervalle nicht überlappen

**dynam. Programm.:**

Löse Teilprobleme. Wähle beste Kombination!

**Greedy:**

Nicht alle Kombinationen, sondern nur **EINE** Lösung

# dynam. Programmieren

## Teilstruktur:

Seien  $S_{ij}$  die Jobs, die starten, nachdem  $a_i$  endet und enden, bevor  $a_j$  beginnt.

Suchen max. Anzahl kompatibler Jobs aus  $S_{ij}$ .

Sei  $A_{ij}$  so eine Menge, und sei  $a_k$  in  $A_{ij}$ .

Ist  $a_k$  in optimaler Teillösung, so ergeben sich Teilprobleme  $S_{ik}$  und  $S_{kj}$  mit

$$A_{ik} = A_{ij} \cap S_{ik} \text{ und } A_{kj} = A_{ij} \cap S_{kj}$$

$$\text{sowie } A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

Optimale Menge  $A_{ij}$  setzt sich zusammen aus einem  $a_k$  und den sich ergebenden Optima für  $S_{ik}$  und  $S_{kj}$ .

# dynam. Programmieren

Seien  $\min[i, j]$  opt. Anzahl von Jobs für  $S_{ij}$ . Dann:

- $\min[i, j] = 0$  falls  $S_{i,j} = \emptyset$
- $\min[i, j] = \max_{a_k \in S_{i,j}} \{ \min[i, k] + \min[k, j] + 1 \}$  sonst

## Greedy:

Intuitiv: Wähle Job als erstes, der als erstes endet. Also  $a_1$ .  
Lasse alle weg, die in Konflikt mit  $a_1$  stehen.  
Iteriere!

Einfach, wenn Jobs nach Endzeitpunkt sortiert sind.

# Jobauswahl:

## Korrektheit:

Zeige durch Widerspruch, dass allgemein für ein Teilproblem immer eine max. Lösung existiert, wo der Job mit dem frühesten Ende drin ist.

## Algorithmus:

gegeben Felder  $s, e$  (Start, Ende)

halte Index  $k$  (letzter gefundener Job)

Sortiere Job nach Endzeiten  $e$        $O(n \log n)$

weiter rekursiv oder iterativ:

# Jobauswahl

Jobauswahl(0) gibt fiktives  $a_0$  mit  $e_0 = 0$  zurück

**Jobauswahl( $k$ ):**

$i \leftarrow k + 1;$

**while** ( $i \leq n$  und  $s[i] < e[k]$ )  $\{i++;\}$

**if** ( $i \leq n$ ) **return**  $\{a_i\} \cup \text{Jobauswahl}(i);$

**else** **return**  $\emptyset; \}$

Laufzeit:  $O(n)$

**iterativ:**

$A \leftarrow \{a_1\}; k \leftarrow 1;$

**for** ( $i \leftarrow 2$  to  $n$ )  $\{$

**if** ( $s[i] \geq e[k]$ )  $\{$

$A \leftarrow A \cup \{a_i\};$

$k \leftarrow i;$

$\}$



# Greedy:

## Prinzip:

- Bestimme Teilstruktur
- Löse rekursiv  
bei Greedy: nur 1 Teilproblem
- zeige, dass Optimum gefunden wird
- rekursiv, dann iterativ

**Geht das immer?** → Rucksackproblem (gewichtet)

# Rucksackproblem

## Problemstellung:

- Wir haben  $n$  Gegenstände  $g_1, \dots, g_n$  mit Werten  $v_i$  und Gewichten  $w_i$  für  $1 \leq i \leq n$ .
- Träger kann  $\leq W$  Kilo tragen.
- Maximiere Gesamtwert der Ladung

## dynamisches Programmieren:

wertvollste Ladung  $\leq W$  Kilo besteht aus  $g_j$  und Rest.  
Rest ist wertvollste Ladung mit  $\leq W - w_j$  Kilo ohne  $g_j$   
Maximiere über  $j$ .

**Laufzeit:**  $O(n \cdot W)$

## Greedy:

Suche Gegenstand mit größtem Wert, kleinstem Gewicht, größtem Wert pro Kilo,...

# Rucksack:

## Variante mit teilbaren Gegenständen

→ **Greedy:**

```
Ordne Gegenstände nach Wert pro Kilo ( $g_1, \dots, g_n$ );  
 $i \leftarrow 1$ ;  
while ( $W$  noch nicht erreicht) {  
    stecke  $g_i$  oder Teil davon in Rucksack  
     $i++$ ;  
}
```

**Laufzeit:**  $O(n)$

# Enumeration

## **Definition:**

systematisches Aufzählen aller Lösungen durch Durchmustern des Lösungsraumes

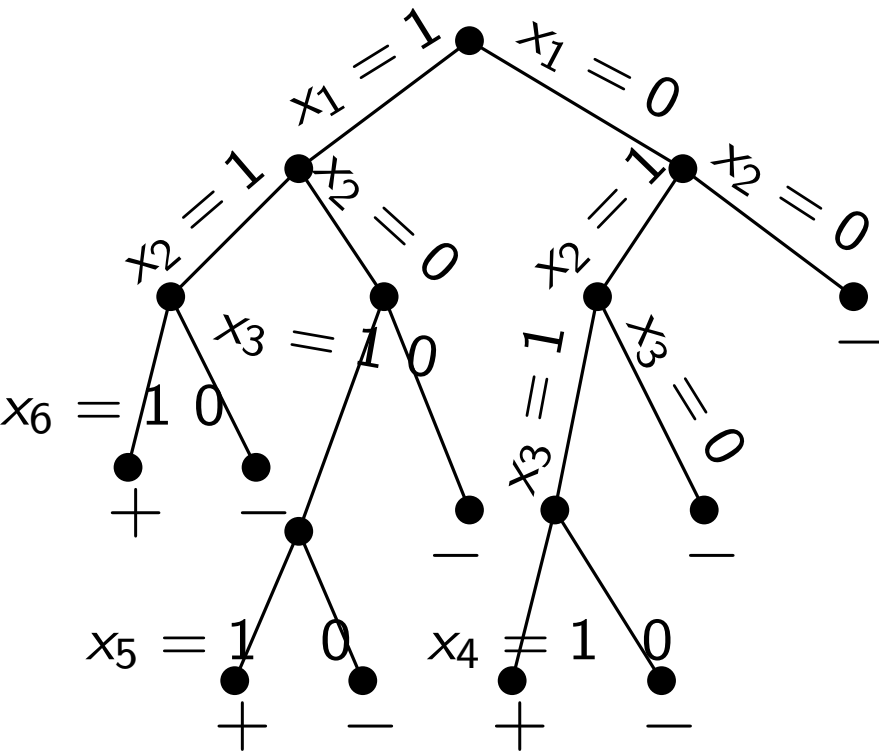
## **Beispiel:**

Zähle alle Möglichkeiten auf (+ Anzahl), die Zahl 9 durch Summe von 3 verschiedenen Zahlen darzustellen.

## → **Aufzählungsbaum:**

Variable  $x_i = 1$  gdw.  $i$  wird benutzt

# Aufzählungsbaum

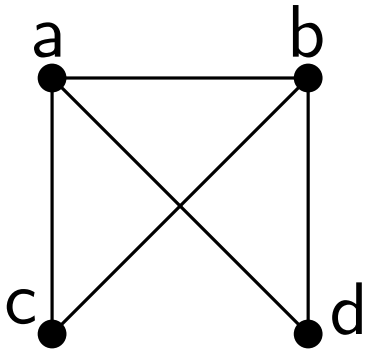


Kanten haben Restriktionen.  
Knoten haben Unterbäume, mit  
allen Kandidaten, die  
Restriktionen auf Pfad Wurzel zu  
Knoten bestimmen.

Beim **Backtracking** bricht man Durchforstung eines Unterbaums ab (weil es Argumente gibt, dass dort keine Lösung vorhanden).  
Gehe hoch, mache weiter!

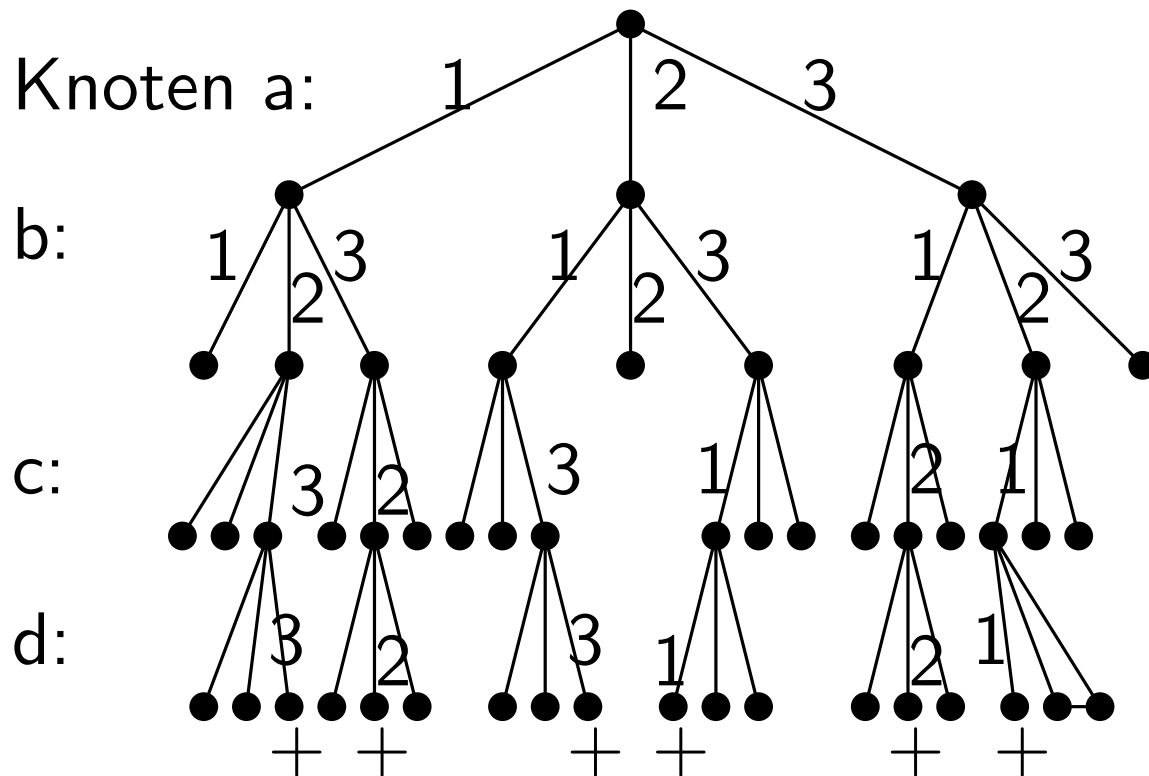
**Lösungen:** 126, 135, 234

# Enumeration: Knotenfärben



Färbe Graph  $G = (V, E)$  mit  $k$  Farben, so dass  $F(v) \neq F(w)$  für  $(v, w) \in E$

$k = 3$  möglich?



haben  $k$ -nären Baum  
der Höhe  $n + 1$ :  
→ Baumgröße  $O(k^n)$

# Branch & Bound

## Idee:

Ähnlich wie bei Enumeration:

### **Baum der Lösungen**

Exploriere immer an dem **vielversprechendsten Blatt**  
Das liefert eventuell Informationen (neue Bewertung,  
neue Schranken) für andere Knoten.

# TSP (Problem des Handlungsreisenden)

## Gegeben:

$n$  Städte  $\{0, 1, \dots, n - 1\}$  mit Kosten  $c_{ij} \in \mathbb{N}_0$  für Reise von  $i$  nach  $j$ .

## Gesucht:

Billigste Rundreise, die jede Stadt genau 1x besucht.

**Untere Schranke:** Jede Stadt muß betreten und verlassen werden

$$C_{opt} \geq \frac{1}{2} \sum_{i=0}^{n-1} \min_j c_{ji} + \min_k c_{ik},$$

da  $C_{opt} \geq \sum_i \min_j c_{ij}$  und  $C_{opt} \geq \sum_i \min_k c_{ik}$

## Beispiel!



# TSP mit dynam. Programmieren

**naiv:** Teste alle  $n!$  Permutationen ( $n! \leq n^n = 2^{n \log n}$ )

## dynamisches Programmieren:

Starte in Stadt 0.

Für  $S \subset \{1, 2, \dots, n-1\}$  und  $i \in S$  definiere  $OPT(S, i)$  als die Länge der kürzesten Tour ab 0, durch  $S$ , bis zu  $i$ .

Dann ist

$OPT(\{i\}, i) = c_{0i}$  für  $1 \leq i < n-1$  und

$OPT(S, i) = \min_{k \in S - \{i\}} OPT(S - \{i\}, k) + c_{ki}$  für  $|S| > 1$ .

Kante zurück zum Start:

$OPT = \min_i OPT(\{1, \dots, n-1\}, i) + c_{i0}$

Speichern OPT-Werte aller Teilmengen, das sind  $2^n$  Stück.

OPT-Werte jeweils in  $O(n)$   $\Rightarrow$  **Laufzeit:**  $O(n2^n)$