

# Algorithmen I

Dennis Hofheinz

Übungen:

Lukas Barth und Lisa Kohl

Institut für theoretische Informatik

Web:

<https://crypto.iti.kit.edu/algo-bose16>

(Folien von Peter Sanders)

# Organisatorisches

## Vorlesungen:

Mo: 15:45–17:15

Mi: 14:00–14:45

## Saalübung:

Mi: 14:45–15:30

**Tutorium:** wöchentlich, ab nächster Woche  
Einteilung mittels Webinscribe  
<https://webinscribe.ira.uka.de/>

**Übungsblätter:** wöchentlich  
Ausgabe Mittwoch nach der Vorlesung/Übung  
Abgabe Freitag 12:45 Uhr (9 Tage nach Ausgabe)

# Organisatorisches

**Sprechstunde:** Donnerstag, 10.30–11.30 Uhr  
(jederzeit bei offener Tür oder nach Vereinbarung)

- ▶ Dennis Hofheinz, Raum 279

**Mittsemesterklausur:** zur Kontrolle

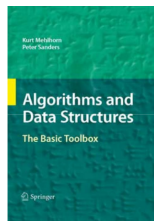
**Abschlussklausur:**

05.10.2016, 100% der Note

nächste Versuchsmöglichkeit: nach dem WS 16/17

# Materialien

- ▶ Folien, Übungsblätter
- ▶ Diskussionsforum: Link siehe Homepage
- ▶ Buch:  
K. Mehlhorn, P. Sanders  
Algorithms and Data Structures — The Basic Toolbox
- ▶ Taschenbuch der Algorithmen  
Springer 2008 (Unterhaltung / Motivation)



## Weitere Bücher

- ▶ [Algorithmen - Eine Einführung](#) von Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, und Clifford Stein von Oldenbourg
- ▶ [Algorithmen und Datenstrukturen](#) von Thomas Ottmann und Peter Widmayer von Spektrum Akademischer Verlag
- ▶ [Algorithmen kurz gefasst](#) von Uwe Schöning von Spektrum Akad. Vlg., Hdg.

# Algorithmus? Kann man das essen?

Pseudogriechische Verballhornung eines **Namens**,  
der sich aus einer **Landschaftsbezeichnung** ableitet:

**Al-Khwarizmi** war persischer/usbekischer  
Wissenschaftler (aus **Khorasan**) aber lebte in  
Bagdad  $\approx$  780..840.

Machtzentrum des arabischen Kalifats  
auf seinem Höhepunkt.

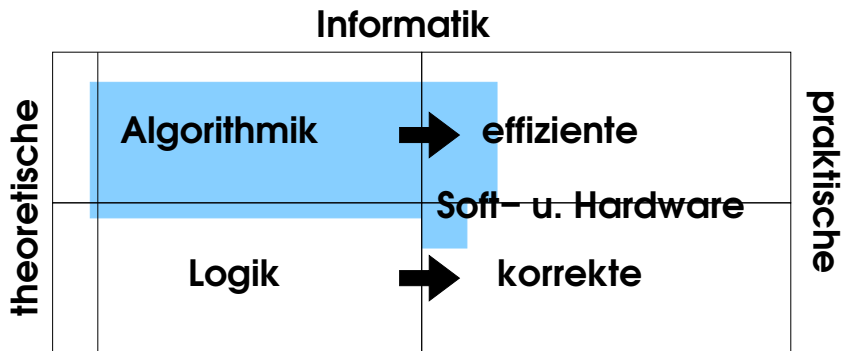


Er hat ein **Rechenlehrbuch** geschrieben.

$\rightsquigarrow$  **Algorithmus** wurde zum Synonym für **Rechenvorschrift**.

Unter einem **Algorithmus** versteht man eine **genau** definierte  
**Handlungsvorschrift** zur Lösung eines Problems oder einer bestimmten  
Art von Problemen in **endlich vielen Schritten**.

Kerngebiet der (theoretischen) Informatik  
mit direktem Anwendungsbezug



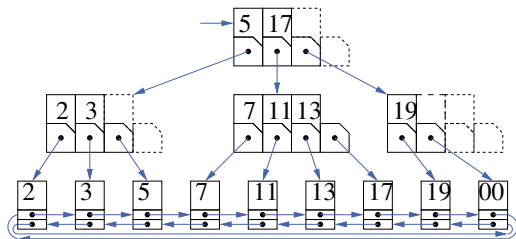
# Datenstruktur

Ein Algorithmus bearbeitet **Daten**.

Wenn ein Teil dieser Daten eine (**interessante**) **Struktur** haben, nennen wir das **Datenstruktur**.

Immer wiederkehrende Datenstrukturen und dazugehörige Algorithmenteile

↔ wichtige **Grundwerkzeuge** (**Basic Toolbox**)





# Themenauswahl: Werkzeugkasten

Immer wieder benötigte

- ▶ Datenstrukturen
- ▶ Algorithmen
- ▶ Entwurfstechniken  $\rightsquigarrow$  neue Algorithmen
- ▶ Analysetechniken  $\rightsquigarrow$  Leistungsgarantien, objektiver Algorithmenvergleich

Jeder Informatiker braucht das  $\rightsquigarrow$  Pflichtvorlesung

# Inhaltsübersicht

1. Amuse Geule Appetithäppchen
2. Einführung der Werkzeugkasten für den Werkzeugkasten
3. Folgen, Felder, Listen Mütter und Väter aller Datenstrukturen
4. Hashing Chaos als Ordnungsprinzip
5. Sortieren Effizienz durch Ordnung
6. Prioritätslisten immer die Übersicht behalten
7. Sortierte Liste die eierlegende Wollmilchsau
8. Graphrepräsentation Beziehungen im Griff haben
9. Graphtraversierung globalen Dingen auf der Spur
10. Kürzeste Wege schnellstens zum Ziel
11. Minimale Spannbäume immer gut verbunden
12. Optimierung noch mehr Entwurfsmethoden

# Amuse Geule

## Beispiel: Langzahl-Multiplikation

Schreibe Zahlen als **Ziffernfolgen**  $a = (a_{n-1} \dots a_0)$ ,  $a_i \in 0..B - 1$ .

Wir zählen

**Volladditionen:**  $(c', s) := a_i + b_j + c$

$$\text{Beispiel } (B = 10): 9 + 9 + 1 = (1, 9)$$

**Ziffernmultiplikationen:**  $(p', p) := a_i \cdot b_j$

$$\text{Beispiel } (B = 10): 9 \cdot 9 = (8, 1)$$

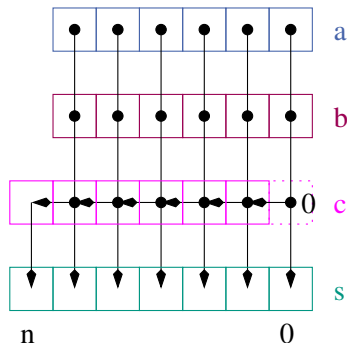
# Addition

$c=0$  : Digit

for  $i := 0$  to  $n-1$  do  $(c, s_i) := a_i + b_i + c$

$s_n := c$

// carry / Überlauf



Satz: Addition von  $n$ -Ziffern-Zahlen braucht  $n$  Ziffern-Additionen.

# Beispiel

$c=0$  : Digit

**for**  $i := 0$  **to**  $n-1$  **do**  $(c, s_i) := a_i + b_i + c$

$s_n := c$

4	5	6
---	---	---

 a

7	8	2
---	---	---

 b

1	1	0
---	---	---

 c

1	2	3	8
---	---	---	---

 s

// carry / Überlauf

# Exkurs: Pseudocode

- ▶ Kein C/C++/Java    Menschenlesbarkeit vor Maschinenlesbarkeit
- ▶ Eher Pascal + Mathe – begin/end    Einrückung trägt Bedeutung

Zuweisung: **:=**

Kommentar: **//**

Ausdrücke: volle Mathepower     $\{i \geq 2 : \neg \exists a, b \geq 2 : i = ab\}$

Deklarationen: **c=0** : Digit

Tupel:  $(c, s_i) := a_i + b_i + c$

Schleifen: **for** , **while** , **repeat ...until** , ...

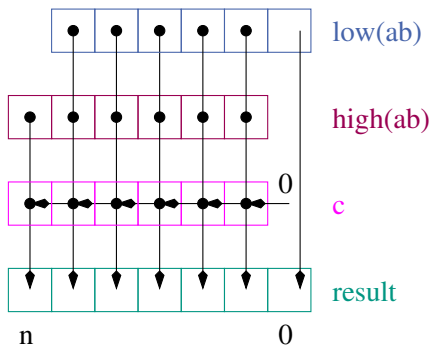
uvam: Buch Abschnitt 2.3, hier: just in time und **on demand**  
**if** , Datentypen, Klassen, Speicherverwaltung

# Exkurs vom Exkurs: Wieso nicht C++/Java-like?

- ▶ Klare Unterscheidung von **Programmcode**
- ▶ viele **redundante** `() [] ;`
- ▶ C for ist sehr **low level**
- ▶ `==` ist unschön während `:=` für Zuweisung klarer ist
- ▶ C **Logik/Bit**operatoren sind kryptischer als `^` etc.
- ▶ Wir verwendenen C++/Java-Notation wo dies sinnvoll ist  
`// ++ --, - + = - =`
- ▶ **Mathe**notation ist oft mächtiger

# Ziffernmultiplikation

**Function** numberTimesDigit( $a$  : **Array**  $[0..n-1]$  of Digit,  $b$  : Digit)





# Beispiel

numberTimesDigit(256,4)

8	0	4
---	---	---

 low(ab)

0	2	2
---	---	---

 high(ab)

1	0	0
---	---	---

 c

1	0	2	4
---	---	---	---

 result

# Ziffernmultiplikation

```
Function numberTimesDigit( $a$  : Array [0.. $n-1$ ] of Digit,  $b$  : Digit)
  result : Array [0.. $n$ ] of Digit
   $c=0$  : Digit // carry / Überlauf
  ( $h', \ell$ ) :=  $a[0] \cdot b$  // Ziffernmultiplikation
  result[0] :=  $\ell$ 
  for  $i$  := 1 to  $n-1$  do //  $n-1$  Iterationen
    ( $h, \ell$ ) :=  $a[i] \cdot b$  // Ziffernmultiplikation
    ( $c, \text{result}[i]$ ) :=  $c + h' + \ell$  // Ziffernaddition
     $h' := h$ 
  result[ $n$ ] :=  $c + h'$  // Ziffernaddition, kein Überlauf?!
  return result
```

Analyse:  $1 + (n-1) = n$  Multiplikationen,  $(n-1) + 1 = n$  Additionen

# Schulmultiplikation

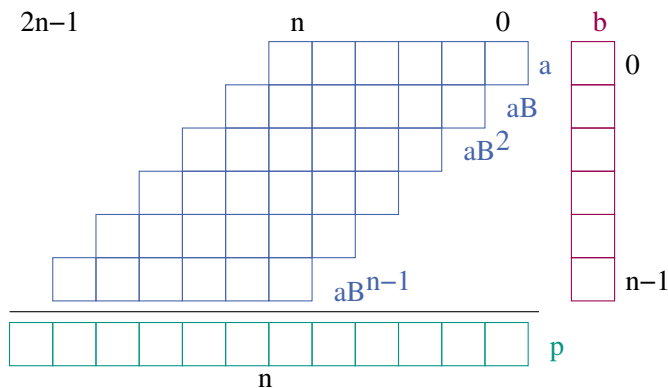
$p=0 : \mathbb{N}$

for  $j := 0$  to  $n-1$  do

// Langzahladdition, Langzahl mal Ziffer, Schieben:

$p := p + a \cdot b[j] \cdot B^j$

// Langzahl



# Schulmultiplikation Beispiel

$p=0 : \mathbb{N}$

for  $j := 0$  to  $n-1$  do

// Langzahladdition, Langzahl mal Ziffer, Schieben:

$p := p + a \cdot b[j] \cdot B^j$

// Langzahl

32\*64

					<b>b</b>
		1	2	8	4
1	9	2	aB		6
<hr/>					
2	0	4	8		<b>p</b>

# Schulmultiplikation Analyse

```
p:=0 :  $\mathbb{N}$ 
for j := 0 to n-1 do
    p:= p // n+j Ziffern (außer bei j = 0)
    + // n+1 Ziffernadditionen (optimiert)
    a · b[j] // je n Additionen/Multiplikationen
    · B^j // schieben (keine Zifferarithmetik)
```

Insgesamt:

$n^2$  Multiplikationen

$n^2 + (n-1)(n+1) = 2n^2 - 1$  Additionen

---

$3n^2 - 1 \leq 3n^2$  Zifferoperationen

# Exkurs O-Kalkül, die Erste

$$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

Idee: Konstante Faktoren (und Anfangsstück) ausblenden

- + Operationen zählen  $\rightsquigarrow$  Laufzeit welche Ops.?
- + Rechnungen vereinfachen
- + Interpretation vereinfachen
- ? Werfen wir **zuviel** Information weg ?

Beispiel: Schulumultiplikation braucht **Zeit**  $O(n^2)$

# Ergebnisüberprüfung

später an Beispielen

# Ein rekursiver Algorithmus

**Function** `recMult(a,b)`

**assert**  $a$  und  $b$  haben  $n$  Ziffern, sei  $k = \lceil n/2 \rceil$

**if**  $n = 1$  **then return**  $a \cdot b$

Schreibe  $a$  als  $a_1 \cdot B^k + a_0$

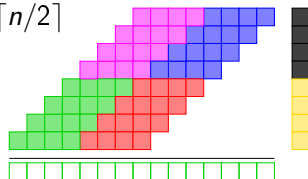
Schreibe  $b$  als  $b_1 \cdot B^k + b_0$

**return**

`recMult(a1, b1)` ·  $B^{2k}$  +

`(recMult(a0, b1) + recMult(a1, b0))` ·  $B^k$  +

`recMult(a0, b0)`





# Beispiel

$$\begin{array}{|c|c|} \hline 10 & 01 \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline 19 & 84 \\ \hline \end{array} = \\ 10 \cdot 19 \cdot 10000 + \\ (10 \cdot 84 + 1 \cdot 19) \cdot 100 + \\ 1 \cdot 84 = \\ 1985984$$

# Analyse

```
Function recMult( $a, b$ ) //  $T(n)$  Ops
  assert  $a$  und  $b$  haben  $n$  Ziffern, sei  $k = \lceil n/2 \rceil$ 
  if  $n = 1$  then return  $a \cdot b$  // 1 Op
  Schreibe  $a$  als  $a_1 \cdot B^k + a_0$  // 0 Ops
  Schreibe  $b$  als  $b_1 \cdot B^k + b_0$  // 0 Ops
  return
    recMult( $a_1, b_1$ )  $\cdot B^{2k} +$  //  $T(n/2) + 2n$  Ops
    (recMult( $a_0, b_1$ ) + recMult( $a_1, b_0$ ))  $B^k +$  //  $2T(n/2) + 2n$  Ops
    recMult( $a_0, b_0$ ) //  $T(n/2) + 2n$  Ops
```

Also  $T(n) \leq 4T(n/2) + 6n$

Übung: Wo kann man hier  $\approx 2n$  Ops sparen?

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 6 \cdot n & \text{if } n \geq 2. \end{cases}$$

→ (Master-Theorem, stay tuned)

$$T(n) \in \Theta\left(n^{\log_2 4}\right) \subseteq O(n^2)$$

**Aufgabe:**

Zeigen Sie durch vollständige Induktion, dass

$$T(n) \leq 7n^2 - 6n$$

, falls  $n$  eine Zweierpotenz ist

# Exkurs: Algorithmen-Entwurfsmuster

Im Buch: siehe auch Index!

**Schleife:** z. B. Addition

**Unterprogramm:** z. B. Ziffernmultiplikation, Addition

**Teile und Herrsche:** (lat. divide et impera, engl. divide and conquer)  
Aufteilen in eins oder mehrere, **kleinere** Teilprobleme,  
oft rekursiv

Es kommen noch mehr: greedy, dynamische Programmierung,  
Metaheuristiken, Randomisierung, . . .

# Karatsuba-Ofman Multiplikation[1962]

Beobachtung:  $(a_1 + a_0)(b_1 + b_0) = a_1b_1 + a_0b_0 + a_1b_0 + a_0b_1$

**Function** recMult( $a, b$ )

**assert**  $a$  und  $b$  haben  $n$  Ziffern, sei  $k = \lceil n/2 \rceil$

**if**  $n = 1$  **then return**  $a \cdot b$

Schreibe  $a$  als  $a_1 \cdot B^k + a_0$

Schreibe  $b$  als  $b_1 \cdot B^k + b_0$

$c_{11} := \text{recMult}(a_1, b_1)$

$c_{00} := \text{recMult}(a_0, b_0)$

**return**

$c_{11} \cdot B^{2k} +$

$(\text{recMult}((a_1 + a_0), (b_1 + b_0)) - c_{11} - c_{00})B^k$

$+ c_{00}$

# Beispiel

$$\boxed{10} \boxed{01} \cdot \boxed{19} \boxed{84} =$$

$$10 \cdot 19 \cdot 10000 +$$

$$((10 + 1) \cdot (19 + 84) - 10 \cdot 19 - 1 \cdot 84) \cdot 100 +$$
$$1 \cdot 84 =$$

$$1985984$$

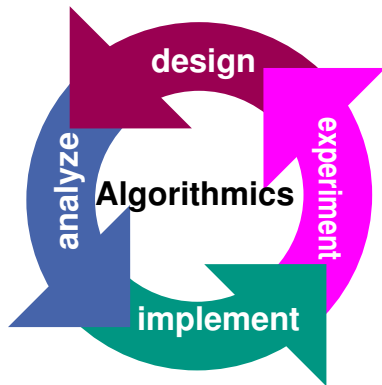
# Analyse

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 3 \cdot T(\lceil n/2 \rceil) + 10 \cdot n & \text{if } n \geq 2. \end{cases}$$

→ (Master-Theorem)

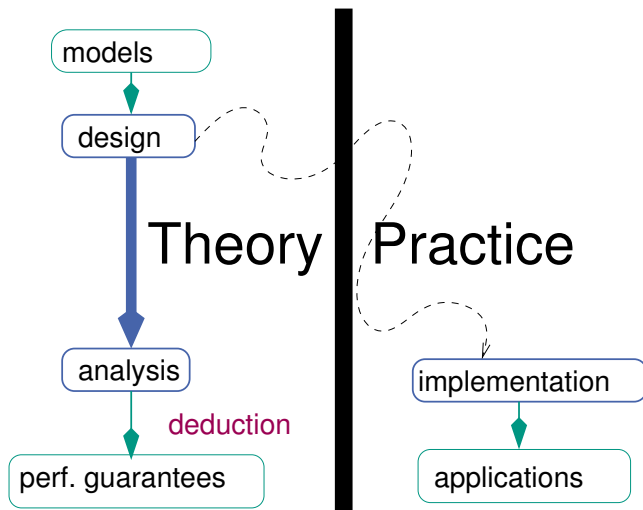
$$T(n) = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.58}\right)$$

# Algorithm Engineering – was hat das mit der Praxis zu tun?'

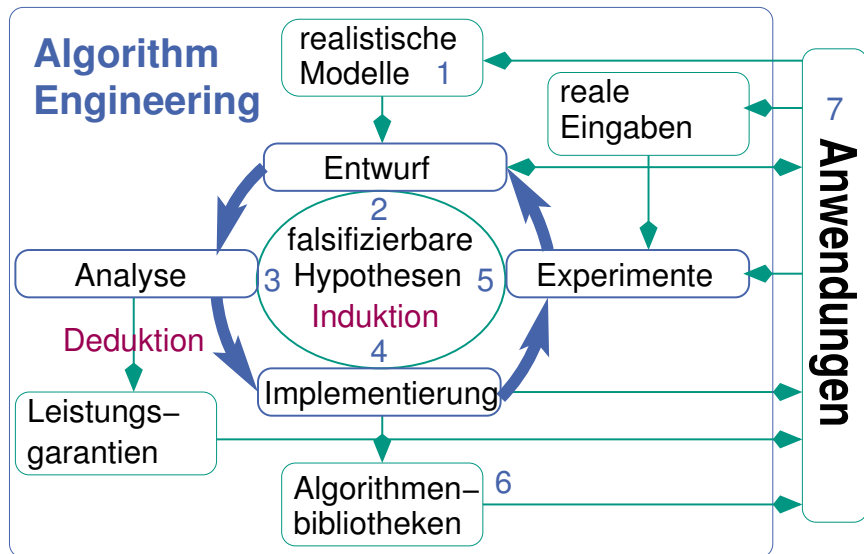




# Algorithmentheorie (Karikatur)

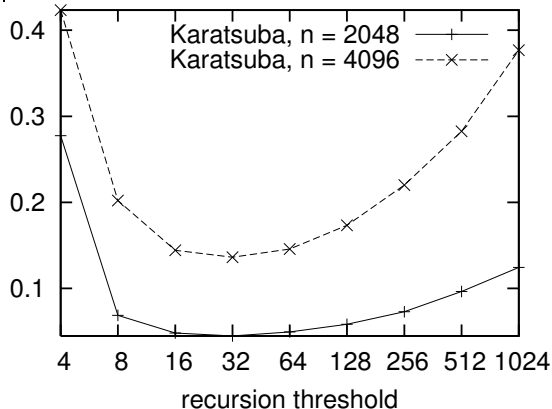


# Algorithmik als Algorithm Engineering



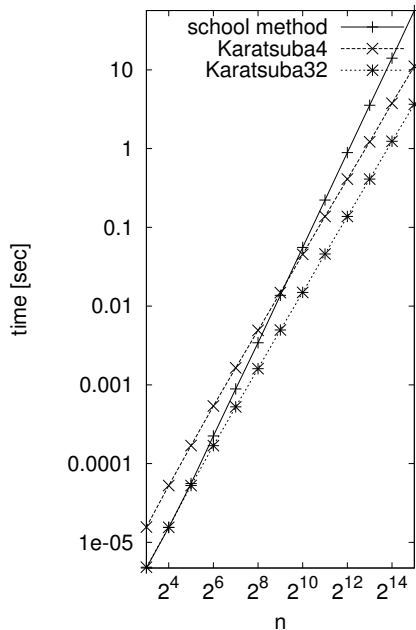
## Zurück zur Langzahlmultiplikation

- ▶ Zifferngröße  $\leftrightarrow$  Hardware-Fähigkeiten  
z. B. 64 Bit
- ▶ Schulmultiplikation für kleine Eingaben
- ▶ Assembler, SIMD, ...



# Skalierung

- ▶ Asymptotik setzt sich durch
- ▶ Konstante Faktoren oft Implementierungsdetail

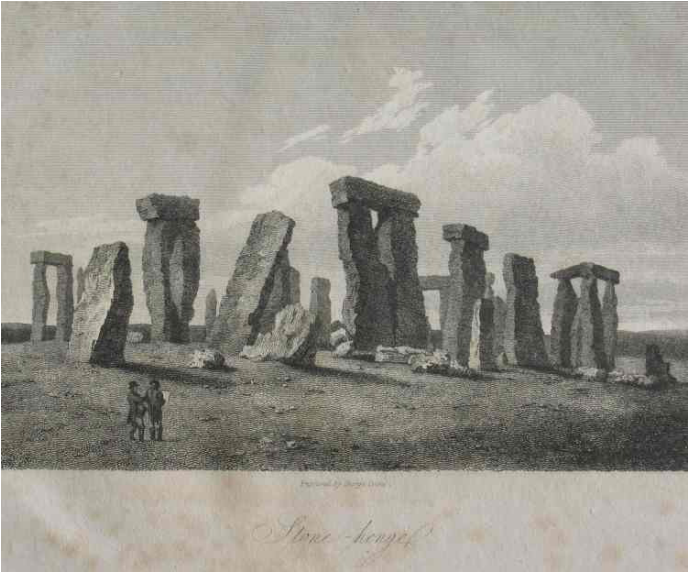


# Blick über den Tellerrand

- ▶ Bessere Potenzen durch Aufspalten in **mehr Teile**
- ▶ **Schnelle Fourier Transformation**  
     $\rightsquigarrow O(n)$  Multiplikationen von  $O(\log n)$ -Bit Zahlen
- ▶ [Schönhage-Strassen 1971]: Bitkomplexität  $O(n \log n \log \log n)$
- ▶ [Fürer 2007]: Bitkomplexität  $2^{O(\log^* n)} n \log n$
- ▶ Praxis: Karatsuba-Multiplikation ist nützlich für Zahlenlängen aus der **Kryptographie**
- ▶ GnuPG, OpenSSL verwenden Karatsuba (ab best. Bitlänge)

Iterierter Logarithmus: 
$$\log^* n = \begin{cases} 0 & \text{falls } n \leq 1 \\ 1 + \log^* \log n & \text{sonst} \end{cases}$$

# Einführendes



# Überblick

- ▶ Algorithmenanalyse
- ▶ Maschinenmodell
- ▶ Pseudocode
- ▶ Codeannotationen
- ▶ Mehr Algorithmenanalyse
- ▶ Graphen

# (Asymptotische) Algorithmenanalyse

Gegeben:

Ein Programm

**Gesucht:** Laufzeit  $T(I)$  (# Takte), eigentlich für **alle Eingaben  $I$  (!)**  
(oder auch Speicherverbrauch, Energieverbrauch, ...)

**Erste Vereinfachung:** **Worst case:**  $T(n) = \max_{|I|=n} T(I)$

(Später mehr:

average case, best case, die Rolle des Zufalls, mehr Parameter)





## Zweite Vereinfachung: Asymptotik

$$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„höchstens“

$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

„mindestens“

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

„genau“

$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„weniger“

$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

„mehr“

# O-Kalkül Rechenregeln

Schludrigkeit: implizite Mengenklammern.

Lese ' $f(n) = E$ ' als ' $\{f(n)\} \subseteq E$ '

$cf(n) \in \Theta(f(n))$  für jede positive Konstante  $c$

$$\sum_{i=0}^k a_i n^i \in O(n^k)$$

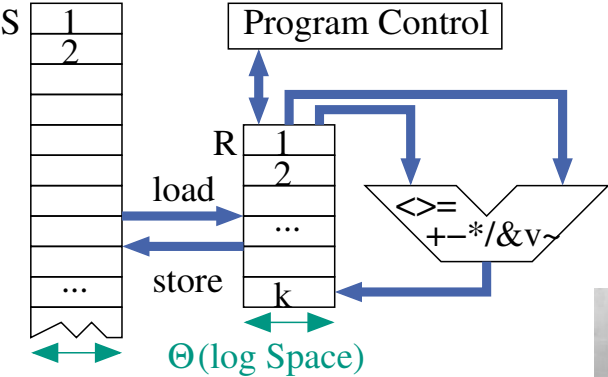
$$f(n) + g(n) \in \Omega(f(n)),$$

$$f(n) + g(n) \in O(f(n)) \text{ falls } g(n) = O(f(n)),$$

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)).$$

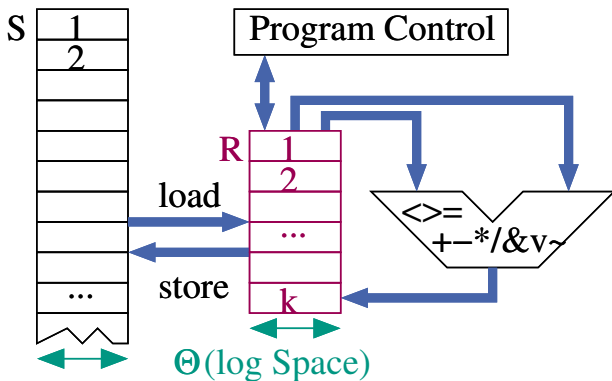
u. s. w.

# Maschinenmodell: RAM (Random Access Machine)



Moderne (RISC) Adaption des  
von Neumann-Modells [von Neumann 1945]

# Register

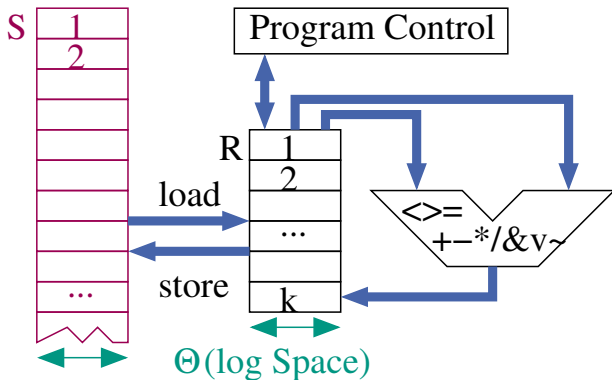


$k$  (irgendeine Konstante) Speicher

$R_1, \dots, R_k$  für

(kleine) ganze Zahlen

# Hauptspeicher

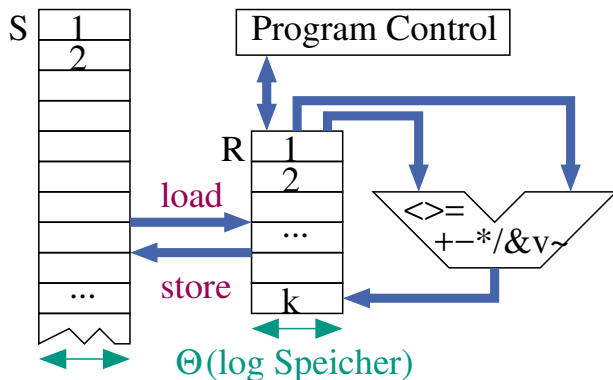


Unbegrenzter Vorrat an Speicherzellen

$S[1], S[2], \dots$  für

(kleine) ganze Zahlen

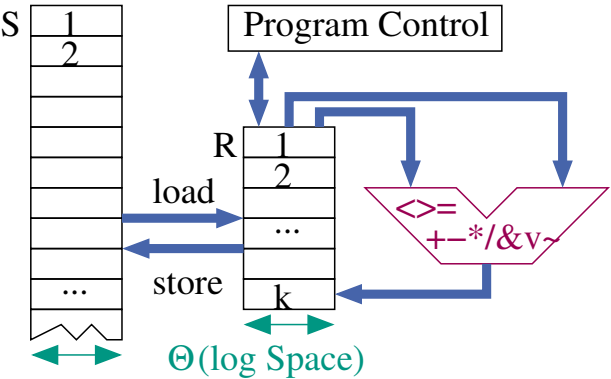
# Speicherzugriff



$R_i := S[R_j]$  **lädt** Inhalt von Speicherzelle  $S[R_j]$  in Register  $R_i$ .

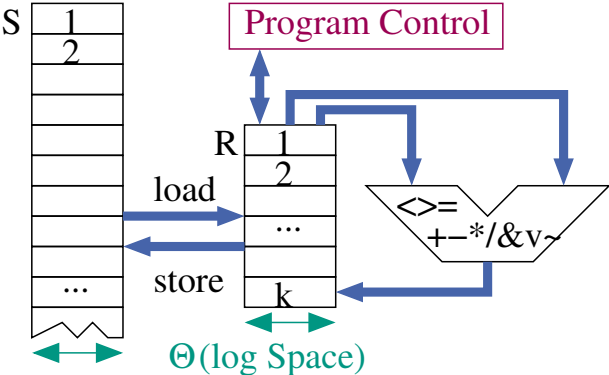
$S[R_j] := R_i$  **speichert** Register  $R_i$  in Speicherzelle  $S[R_j]$ .

# Rechnen



$R_j := R_j \odot R_\ell$  Registerarithmetik.  
 '⊙' ist Platzhalter für eine Vielzahl von Operationen  
 Arithmetik, Vergleich, Logik

# Bedingte Sprünge



$JZ j, R_i$  Setze Programmausführung an Stelle  $j$  fort falls  $R_i = 0$



# „Kleine“ ganze Zahlen?

Alternativen:

**Konstant viele Bits (64?):** theoretisch unbefriedigend, weil nur endlich viel Speicher adressierbar  $\rightsquigarrow$  endlicher Automat

**Beliebige Genauigkeit:** viel zu optimistisch für vernünftige **Komplexitätstheorie**. Beispiel:  $n$ -maliges Quadrieren führt zu einer Zahl mit  $\approx 2^n$  Bits.

**OK für Berechenbarkeit**

**Genug um alle benutzten Speicherstellen zu adressieren:** bester Kompromiss.

# Algorithmenanalyse im RAM-Modell

**Zeit:** Ausgeführte Befehle zählen,  
d. h. Annahme 1 Takt pro Befehl.  
Nur durch späteres  $O(\cdot)$  gerechtfertigt!  
Ignoriert Cache, Pipeline, Parallelismus. . .

**Platz:** Etwas unklar:

- ▶ letzte belegte Speicherzelle?
- ▶ Anzahl benutzter Speicherzellen?
- ▶ Abhängigkeit von Speicherverwaltungsalgorithmen?

Hier: Es kommt eigentlich nie drauf an.

# Mehr Maschinenmodell

Cache: schneller Zwischenspeicher

- ▶ begrenzte Größe
  - ↪ kürzlich/häufig zugriffene Daten sind eher im Cache
- ▶ blockweiser Zugriff
  - ↪ Zugriff auf konsekutive Speicherbereiche sind schnell

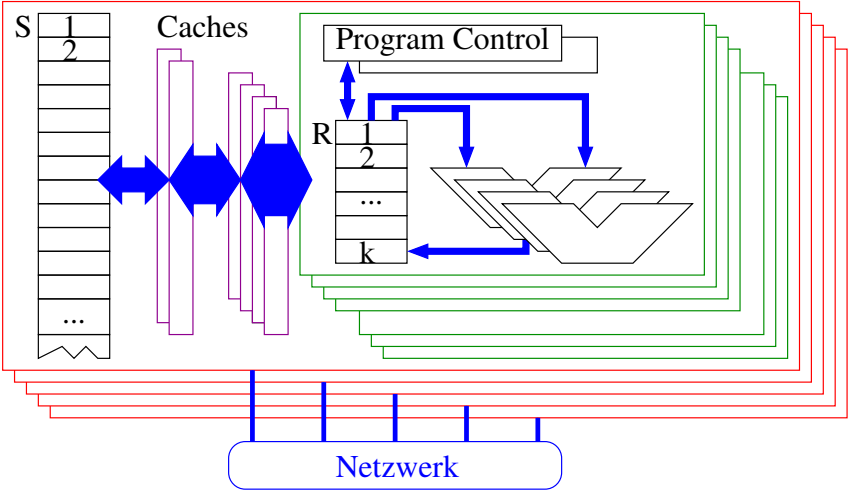
Parallelverarbeitung: Mehrere Prozessoren

↪ unabhängige Aufgaben identifizieren

...

mehr in TI, Algorithmen II, Programmierparadigmen,...

# Mehr Maschinenmodell



# Pseudocode

just in time

## Beispiel:

**Class** Complex( $x, y$  : Number) **of** Number

Number  $r := x$

Number  $i := y$

**Function** abs : Number **return**  $\sqrt{r^2 + i^2}$

**Function** add( $c'$  : Complex) : Complex  
**return** Complex( $r + c'.r, i + c'.i$ )

# Design by Contract / Schleifeninvarianten

**assert:** Aussage über Zustand der Programmausführung

**Vorbedingung:** Bedingung für korrektes Funktionieren einer Prozedur

**Nachbedingung:** Leistungsgarantie einer Prozedur,  
falls Vorbedingung erfüllt

**Invariante:** Aussage, die an „vielen“ Stellen im Programm gilt

**Schleifeninvariante:** gilt vor / nach jeder Ausführung des  
Schleifenkörpers

**Datenstrukturinvariante:** gilt vor / nach jedem Aufruf einer Operation  
auf abstraktem Datentyp

Hier: **Invarianten** als zentrales Werkzeug für Algorithmenentwurf und Korrektheitsbeweis.

## Beispiel (Ein anderes als im Buch)

```
Function power( $a : \mathbb{R}; n_0 : \mathbb{N}$ ) :  $\mathbb{R}$   
   $p = a : \mathbb{R}; r = 1 : \mathbb{R}; n = n_0 : \mathbb{N}$   
  while  $n > 0$  do  
    if  $n$  is odd then  $n--$  ;  $r := r \cdot p$   
    else  $(n, p) := (n/2, p \cdot p)$   
  return  $r$ 
```

## Beispiel (Ein anderes als im Buch)

```
Function power( $a : \mathbb{R}; n_0 : \mathbb{N}$ ) :  $\mathbb{R}$   
  assert  $n_0 \geq 0$  and  $\neg(a = 0 \wedge n_0 = 0)$  // Vorbedingung  
   $p = a : \mathbb{R}; r = 1 : \mathbb{R}; n = n_0 : \mathbb{N}$  //  $p^n r = a^{n_0}$   
  while  $n > 0$  do  
    invariant  $p^n r = a^{n_0}$  // Schleifeninvariante (*)  
    if  $n$  is odd then  $n--$ ;  $r := r \cdot p$   
    else  $(n, p) := (n/2, p \cdot p)$   
  assert  $r = a^{n_0}$  // (*)  $\wedge n = 0 \rightarrow$  Nachbedingung  
  return  $r$ 
```



## Rechenbeispiel: $2^5$

$p=a=2 : \mathbb{R}; \quad r=1 : \mathbb{R}; \quad n=n_0=5 : \mathbb{N}$

//  $2^5 \cdot 1 = 2^5$

**while**  $n > 0$  **do**

**if**  $n$  is odd **then**  $n--$  ;  $r:=r \cdot p$

**else**  $(n,p):=(n/2, p \cdot p)$

Iteration	$p$	$r$	$n$	$p^n r$
0	2	1	5	32
1	2	2	4	32
2	4	2	2	32
3	16	2	1	32
4	16	32	0	<b>32</b>

# Beispiel

```
Function power( $a : \mathbb{R}; n_0 : \mathbb{N}$ ) :  $\mathbb{R}$   
  assert  $n_0 \geq 0$  and  $\neg(a = 0 \wedge n_0 = 0)$  // Vorbedingung  
   $p = a : \mathbb{R}; r = 1 : \mathbb{R}; n = n_0 : \mathbb{N}$  //  $p^n r = a^{n_0}$   
  while  $n > 0$  do  
    invariant  $p^n r = a^{n_0}$  // Schleifeninvariante (*)  
    if  $n$  is odd then  $n-- ; r := r \cdot p$   
    else  $(n, p) := (n/2, p \cdot p)$   
  assert  $r = a^{n_0}$  // (*)  $\wedge n = 0 \rightarrow$  Nachbedingung  
  return  $r$ 
```

Fall  $n$  ungerade: Invariante erhalten wegen  $p^n r = \overbrace{p^{n-1}}^{\text{neues } n} \underbrace{pr}_{\text{neues } r}$

# Beispiel

```
Function power( $a : \mathbb{R}; n_0 : \mathbb{N}$ ) :  $\mathbb{R}$   
  assert  $n_0 \geq 0$  and  $\neg(a = 0 \wedge n_0 = 0)$  // Vorbedingung  
   $p = a : \mathbb{R}; r = 1 : \mathbb{R}; n = n_0 : \mathbb{N}$  //  $p^n r = a^{n_0}$   
  while  $n > 0$  do  
    invariant  $p^n r = a^{n_0}$  // Schleifeninvariante (*)  
    if  $n$  is odd then  $n--$  ;  $r := r \cdot p$   
    else  $(n, p) := (n/2, p \cdot p)$   
  assert  $r = a^{n_0}$  //  $(*) \wedge n = 0 \rightarrow$  Nachbedingung  
  return  $r$ 
```

Fall  $n$  gerade: Invariante erhalten wegen  $p^n = \underbrace{(p \cdot p)}_{\text{neues } p} \overbrace{n/2}^{\text{neues } n}$

# Programmanalyse

Die fundamentalistische Sicht: Ausgeführte RAM-Befehle zählen

einfache Übersetzungsregeln

Pseudocode



Maschinenbefehle

Idee:  $O(\cdot)$ -Notation vereinfacht die direkte Analyse des Pseudocodes.

- ▶  $T(I; I') = T(I) + T(I')$ .
- ▶  $T(\text{if } C \text{ then } I \text{ else } I') \in O(T(C) + \max(T(I), T(I')))$ .
- ▶  $T(\text{repeat } I \text{ until } C) \in O(\sum_i T(i\text{-te Iteration}))$

Rekursion  $\rightsquigarrow$  Rekurrenzrelationen

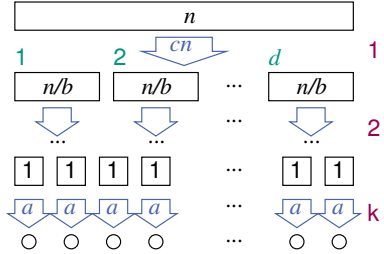
# Schleifenanalyse $\rightsquigarrow$ Summen ausrechnen

Das lernen Sie in Mathe  
Beispiel: Schulmultiplikation

# Eine Rekurrenz für Teile und Herrsche

Für positive Konstanten  $a, b, c, d$ , sei  $n = b^k$  für ein  $k \in \mathbb{N}$ .

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$



# Master Theorem (Einfache Form)

Für positive Konstanten  $a, b, c, d$ , sei  $n = b^k$  für ein  $k \in \mathbb{N}$ .

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

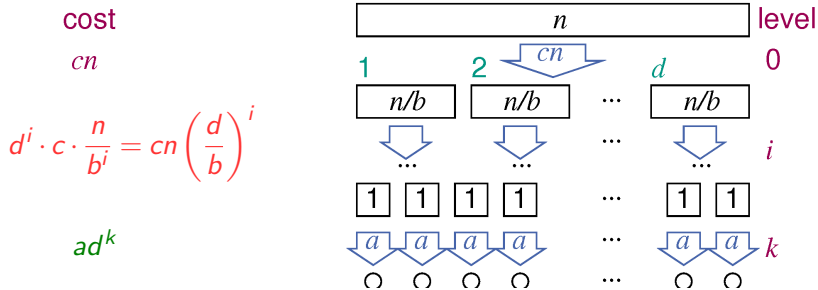
Es gilt

$$r(n) \in \begin{cases} \Theta(n) & \text{falls } d < b \\ \Theta(n \log n) & \text{falls } d = b \\ \Theta(n^{\log_b d}) & \text{falls } d > b. \end{cases}$$

**Übung:** dasselbe gilt, wenn wir beliebige  $n \in \mathbb{N}$  zulassen und  $r(n) = cn + dr(\lceil n/b \rceil)$  für  $n > 1$  setzen.

# Beweisskizze

Auf Ebene  $i$  haben wir  $d^i$  Probleme @  $n/b^i = b^{k-i}$





# Beweisskizze Fall $d < b$

geometrisch schrumpfende Reihe

→ **erste** Rekursionsebene kostet konstanten Teil der Arbeit

$$r(n) = \underbrace{a \cdot d^k}_{o(n)} + \underbrace{cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i}_{O(1)} \in \Theta(n)$$

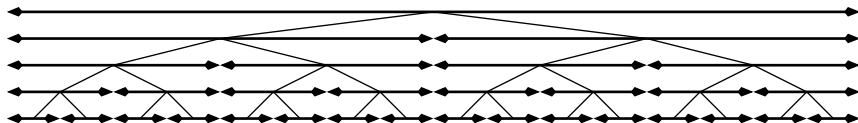


# Beweisskizze Fall $d = b$

gleich viel Arbeit auf **allen**  $k = \log_b(n)$  Ebenen.

$$r(n) = an + cn \log_b n \in \Theta(n \log n)$$

$d=b=2$



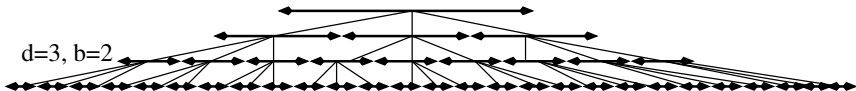
# Beweisskizze Fall $d > b$

geometrisch wachsende Reihe

→ letzte Rekursionsebene kostet konstanten Teil der Arbeit

$$r(n) = ad^k + cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i \in \Theta\left(n^{\log_b d}\right)$$

beachte:  $d^k = 2^{k \log d} = 2^{k \frac{\log b}{\log b} \log d} = b^{k \frac{\log d}{\log b}} = b^{k \log_b d} = n^{\log_b d}$



# Master Theorem Beispiele

Für positive Konstanten  $a$ ,  $b$ ,  $c$ ,  $d$ , sei  $n = b^k$  für ein  $k \in \mathbb{N}$ .

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

**schon gesehen**, kommt noch, **allgemeinerer Fall**

$d < b$ : Median bestimmen

$d = b$ : mergesort, **quicksort**

$d > b$ : **rekursive Multiplikation, Karatsuba-Ofman-Multiplikation**

später an Beispielen

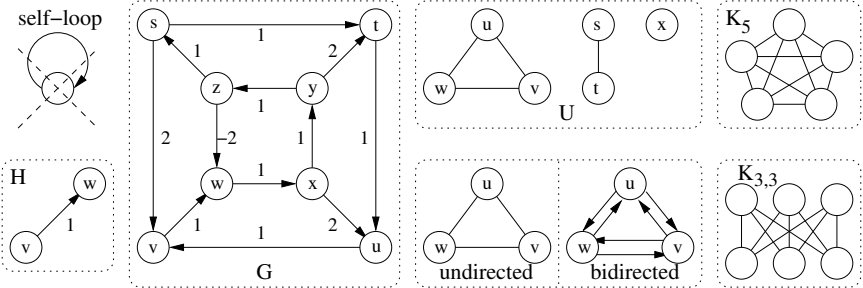
## **Randomisierte Algorithmen**

später an Beispielen



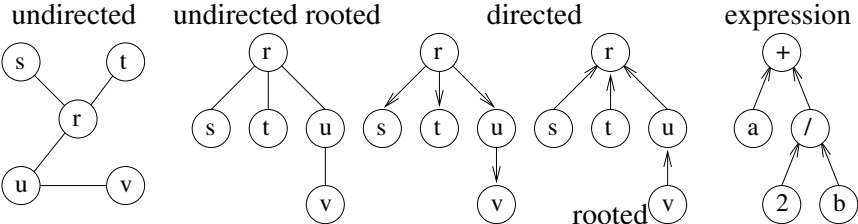
# Graphen

Sie kennen schon (?): **Relationen**, Knoten, Kanten, (un)gerichtete Graphen, Kantengewichte, Knotengrade, knoteninduzierte Teilgraphen. Pfade (einfach, Hamilton-), Kreise, DAGs



# Bäume

Zusammenhang, Bäume, Wurzeln, Wälder, Kinder, Eltern, ...



# Ein erster Graphalgorithmus

Ein **DAG** (directed acyclic graph, gerichteter azyklischer Graph) ist ein gerichteter Graph, der keine Kreise enthält.

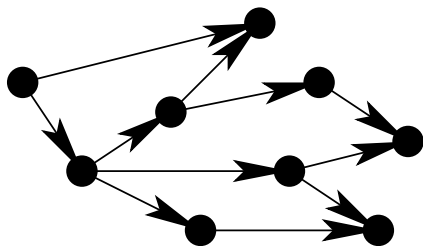
```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
    invariant  $G$  is a DAG iff the input graph is a DAG  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V|=0$ 
```

Analyse: kommt auf **Repräsentation** an (Kapitel 8), geht aber in  $O(|V| + |E|)$ .



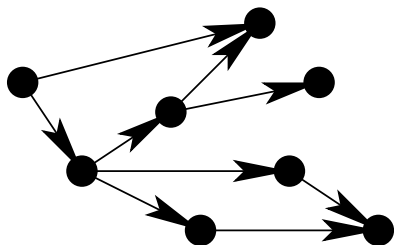
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



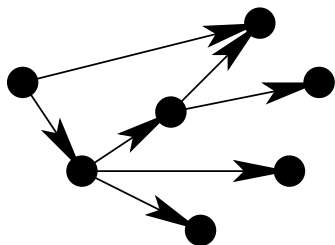
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



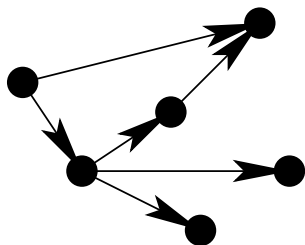
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



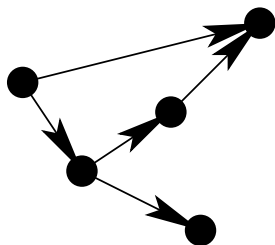
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



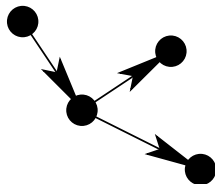
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



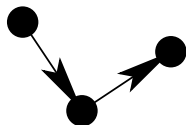
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



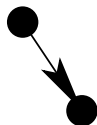
# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```





# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```



# Beispiel

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{outdegree}(v) = 0$  do  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V| = 0$ 
```

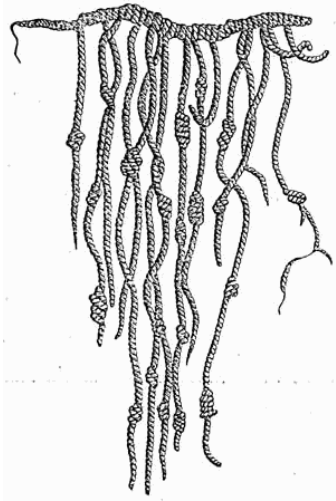
Leerer Graph.

das kommt in „Theoretische Grundlagen der Informatik“

Ganz kurz:

- ▶ Es gibt einigermaßen gute Gründe, „effizient“ mit „polynomiell“ gleichzusetzen (d.h. Laufzeit  $n^{O(1)}$ ).
- ▶ Es gibt viele algorithmische Probleme (NP-vollständig/-schwer), bei denen es SEHR überraschend wäre, wenn sie in Polynomialzeit lösbar wären.

# Folgen als Felder und Listen



# Folgen

spielen in der Informatik eine überragende Rolle.

Das sieht man schon an der Vielzahl von Begriffen:

Folge, **Feld**, Schlange, **Liste**, Datei, Stapel, Zeichenkette, Log. . .

(sequence, array, queue, list, file, stack, string, log. . .).

Wir unterscheiden:

- ▶ **abstrakter** Begriff  $\langle 2, 3, 5, 7, 9, 11, \dots \rangle$
- ▶ **Funktionalität** (stack, . . .)
- ▶ **Repräsentation** und Implementierung

Mathe  
Softwaretechnik  
Algorithmik

# Anwendungen

- ▶ Ablegen und Bearbeiten von Daten aller Art
- ▶ Konkrete Repräsentation abstrakterer Konzepte wie Menge, **Graph** (Kapitel 8),...

# Form Follows Function

Operation	List	SList	UArray	CArray	explanation '*'
[.]	$n$	$n$	1	1	
[. ]	1*	1*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1*	$n$	$n$	insertAfter only
remove	1	1*	$n$	$n$	removeAfter only
pushBack	1	1	1*	1*	amortized
pushFront	1	1	$n$	1*	amortized
popBack	1	$n$	1*	1*	amortized
popFront	1	1	$n$	1*	amortized
concat	1	1	$n$	$n$	
splice	1	1	$n$	$n$	
findNext,...	$n$	$n$	$n^*$	$n^*$	cache-efficient

# Verkettete Listen





# Listenglieder (Items)

**Class** Handle = **Pointer to** Item

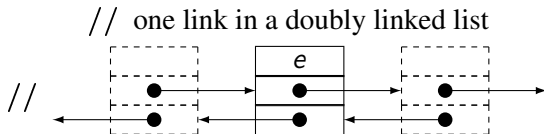
**Class** Item of Element

e : Element

next : Handle

prev : Handle

**invariant** next → prev = prev → next = **this**

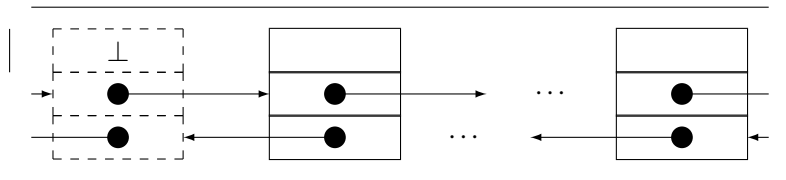


Problem:

Vorgänger des ersten Listenelements?

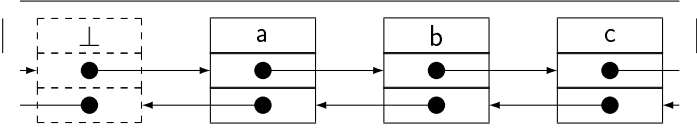
Nachfolger des letzten Listenelements?

## Trick: dummy header



- + **Invariante** immer erfüllt
- + Vermeidung vieler **Sonderfälle**
  - ↪ einfach
  - ↪ lesbar
  - ↪ schnell
  - ↪ testbar
  - ↪ elegant
- Speicherplatz (irrelevant bei langen Listen)

# Dummy header – Beispiel $\langle a, b, c \rangle$



# Die Listenklasse

## Class List of Element

// Item  $h$  is the predecessor of the first element

// and the successor of the last element.

**Function** head : Handle; **return address of**  $h$

// Pos. before any proper element

$h = \begin{pmatrix} \perp \\ \text{head} \\ \text{head} \end{pmatrix} : \text{Item}$

// init to empty sequence



// Simple access functions

**Function** isEmpty : {0, 1}; **return**  $h.\text{next} = \text{head}$

//  $\langle \rangle$ ?

**Function** first : Handle; **assert**  $\neg \text{isEmpty}$ ; **return**  $h.\text{next}$

**Function** last : Handle; **assert**  $\neg \text{isEmpty}$ ; **return**  $h.\text{prev}$

⋮

**Procedure** splice( $a, b, t : \text{Handle}$ ) // **Cut out**  $\langle a, \dots, b \rangle$  and insert after  $t$   
**assert**  $b$  is not before  $a \wedge t \notin \langle a, \dots, b \rangle$

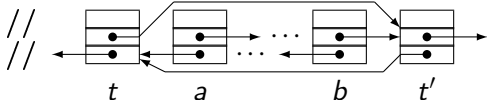
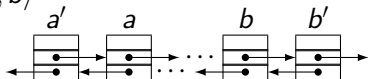
// **Cut out**  $\langle a, \dots, b \rangle$

$a' := a \rightarrow \text{prev}$

$b' := b \rightarrow \text{next}$

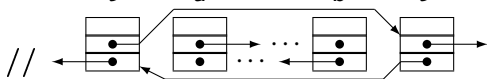
$a' \rightarrow \text{next} := b'$

$b' \rightarrow \text{prev} := a'$



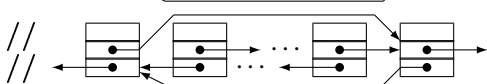
// **insert**  $\langle a, \dots, b \rangle$  after  $t$

$t' := t \rightarrow \text{next}$



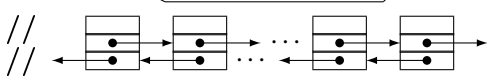
$b \rightarrow \text{next} := t'$

$a \rightarrow \text{prev} := t$



$t \rightarrow \text{next} := a$

$t' \rightarrow \text{prev} := b$



## Splice Beispiel

$$\langle 1, 2, 3, 4, \overset{a}{\underbrace{5, 6, 7, 8}}_b, 9, 10 \rangle, \langle u, v, \overset{t}{\underbrace{w}}, x, y, z \rangle$$

⇓

$$\langle 1, 2, 3, 4, 9, 10 \rangle, \langle u, v, w, 5, 6, 7, 8, x, y, z \rangle$$

## Der Rest sind Einzeiler (?)

// Moving elements around within a sequence.

//  $\langle \dots, a, b, c, \dots, a', c', \dots \rangle \mapsto \langle \dots, a, c, \dots, a', b, c', \dots \rangle$

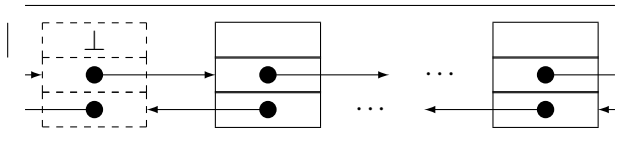
**Procedure** moveAfter( $b, a' : \text{Handle}$ ) splice( $b, b, a'$ )

//  $\langle x, \dots, a, b, c, \dots \rangle \mapsto \langle b, x, \dots, a, c, \dots \rangle$

**Procedure** moveToFront( $b : \text{Handle}$ ) moveAfter( $b, \text{head}$ )

//  $\langle \dots, a, b, c, \dots, z \rangle \mapsto \langle \dots, a, c, \dots, z, b \rangle$

**Procedure** moveToBack( $b : \text{Handle}$ ) moveAfter( $b, \text{last}$ )



# Oder doch nicht? Speicherverwaltung!

naiv / blauäugig / optimistisch:

Speicherverwaltung der Programmiersprache

↪ potentiell sehr langsam

Hier: einmal existierende Variable (z. B. static member in Java)

**freeList** enthält ungenutzte Items.

**checkFreeList** stellt sicher, dass die nicht leer ist.

Reale Implementierungen:

- ▶ naiv aber mit guter Speicherverwaltung
- ▶ verfeinerte Freelistkonzepte (klassenübergreifend, Freigabe,...)
- ▶ anwendungsspezifisch, z. B. wenn man weiß wieviele Items man insgesamt braucht



# Items löschen

//  $\langle \dots, a, b, c, \dots \rangle \mapsto \langle \dots, a, c, \dots \rangle$

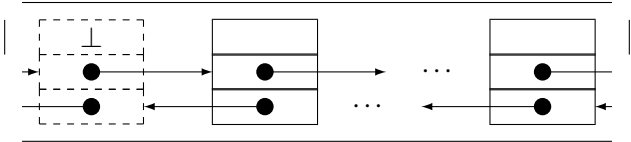
**Procedure** remove( $b$  : Handle) moveAfter(  $b$ , freeList.head)

//  $\langle a, b, c, \dots \rangle \mapsto \langle b, c, \dots \rangle$

**Procedure** popFront remove(first)

//  $\langle \dots, a, b, c \rangle \mapsto \langle \dots, a, b \rangle$

**Procedure** popBack remove(last)



## Elemente einfügen

//  $\langle \dots, a, b, \dots \rangle \mapsto \langle \dots, a, e, b, \dots \rangle$

**Function** insertAfter( $x$  : Element;  $a$  : Handle) : Handle

```
checkFreeList           // make sure freeList is nonempty.
 $a' :=$  freeList.first    // Obtain an item  $a'$  to hold  $x$ ,
moveAfter( $a', a$ )       // put it to the right place.
 $a' \rightarrow e := x$   // and fill it with the right content.
return  $a'$ 
```

**Function** insertBefore( $x$  : Element;  $b$  : Handle) : Handle

```
return insertAfter( $e, b \rightarrow$ prev)
```

**Procedure** pushFront( $x$  : Element) insertAfter( $x$ , head)

**Procedure** pushBack( $x$  : Element) insertAfter( $x$ , last)

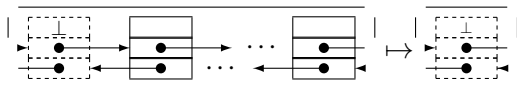
# Ganze (Teil)Listen Manipulieren

```
// ( $\langle a, \dots, b \rangle, \langle c, \dots, d \rangle$ )  $\mapsto$  ( $\langle a, \dots, b, c, \dots, d \rangle, \langle \rangle$ )
```

```
Procedure concat( $L' : \text{List}$ )  
    splice( $L'.\text{first}$ ,  $L'.\text{last}$ , last)
```

```
//  $\langle a, \dots, b \rangle \mapsto \langle \rangle$ 
```

```
Procedure makeEmpty  
    freeList.concat(this) //
```



Das geht in **konstanter Zeit** – unabhängig von der Listenlänge!

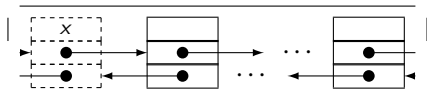
# Suchen

Trick: gesuchtes Element in Dummy-Item schreiben:

**Function** findNext( $x$  : Element; from : Handle) : Handle

$h.e = x$  // Sentinel

**while** from  $\rightarrow e \neq x$  **do**  
    from := from  $\rightarrow$  next  
**return** from



Spart Sonderfallbehandlung.

Allgemein: ein **Wächter-Element** (engl. **Sentinel**) fängt Sonderfälle ab.

$\rightsquigarrow$  einfacher, schneller, ...

# Funktionalität ↔ Effizienz

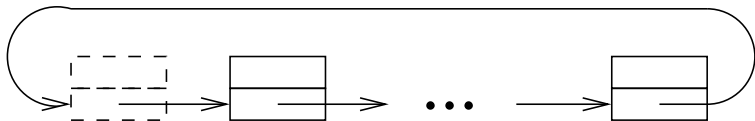
Verwalte zusätzliches Member `size`.

Problem: inter-list `splice` geht nicht mehr in konstanter Zeit

Die Moral von der Geschichte:

Es gibt nicht DIE Listenimplementierung.

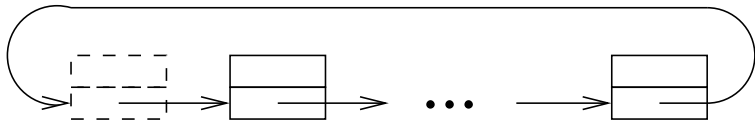
# Einfach verkettete Listen



## Vergleich mit doppelt verketteten Listen

- ▶ weniger Speicherplatz
- ▶ Platz ist oft auch Zeit
- ▶ eingeschränkter, z. B. kein remove
- ▶ merkwürdige Benutzerschnittstelle, z. B. removeAfter

# Einfach verkettete Listen – Invariante?



Betrachte den Graphen  $G = (\text{Item}, E)$  mit  
 $E = \{(u, v) : u \in \text{Item}, v = u.\text{next}\}$

- ▶  $u.\text{next}$  zeigt immer auf ein Item
- ▶  $\forall u \in \text{Item} : \text{indegree}_G(u) = 1.$

Wohldefiniert obwohl nicht unbedingt leicht zu testen.

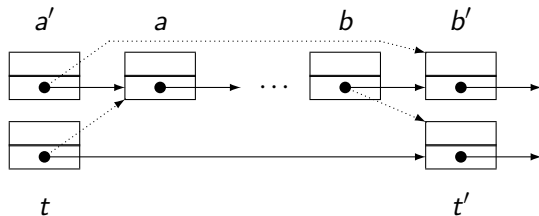
Folge: Items bilden Kollektion von Kreisen

# Einfach verkettete Listen – splice

//( $\langle \dots, a', a, \dots, b, b' \dots \rangle, \langle \dots, t, t', \dots \rangle$ )  $\mapsto$   
//( $\langle \dots, a', b' \dots \rangle, \langle \dots, t, a, \dots, b, t', \dots \rangle$ )

**Procedure** splice( $a', b, t : SHandle$ )

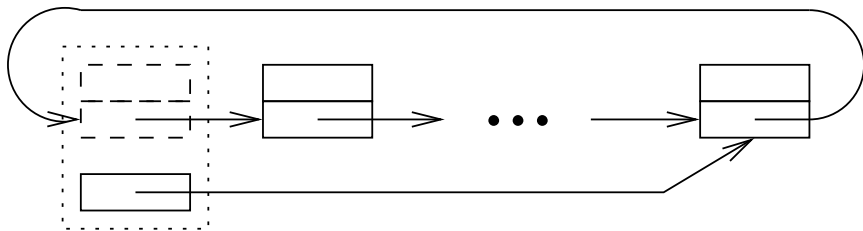
$$\begin{pmatrix} a' \rightarrow \text{next} \\ t \rightarrow \text{next} \\ b \rightarrow \text{next} \end{pmatrix} := \begin{pmatrix} b \rightarrow \text{next} \\ a' \rightarrow \text{next} \\ t \rightarrow \text{next} \end{pmatrix}$$





# Einfach verkettete Listen – pushBack

Zeiger auf letztes Item erlaubt Operation `pushBack`



# Listen: Zusammenfassung, Verallgemeinerungen

- ▶ **Zeiger** zwischen **Items** ermöglichen flexible, **dynamische Datenstrukturen**  
später: Bäume, Prioritätslisten
- ▶ (einfache) **Datenstrukturinvarianten** sind Schlüssel zu einfachen, effizienten Datenstrukturen
- ▶ **Dummy-Elemente**, **Wächter**, ... erlauben Einsparung von Sonderfällen
- ▶ Einsparung von **Sonderfällen** machen Programme, einfacher, lesbarer, testbarer und schneller

## Felder (Arrays)

$$A[i] = a_i \text{ falls } A = \langle a_0, \dots, a_{n-1} \rangle$$

### Beschränkte Felder (Bounded Arrays)

Eingebaute Datenstruktur: Ein Stück Hauptspeicher + Adressrechnung  
Größe muss von Anfang an bekannt sein

### Unbeschränkte Felder (Unbounded Arrays)

$$\begin{aligned} \langle e_0, \dots, e_n \rangle.\text{pushBack}(e) &\rightsquigarrow \langle e_0, \dots, e_n, e \rangle, \\ \langle e_0, \dots, e_n \rangle.\text{popBack} &\rightsquigarrow \langle e_0, \dots, e_{n-1} \rangle, \\ \text{size}(\langle e_0, \dots, e_{n-1} \rangle) &= n . \end{aligned}$$

# Unbeschränkte Felder – Anwendungen

wenn man nicht weiß, wie lang das Feld wird.

Beispiele:

- ▶ Datei zeilenweise einlesen
- ▶ später: Stacks, Queues, Prioritätslisten, ...

# Unbeschränkte Felder – Grundidee

wie beschränkte Felder: Ein Stück Hauptspeicher

**pushBack:** Element anhängen,  $size++$   
Kein Platz?: umkopieren und (größer) neu anlegen

**popBack:**  $size--$   
Zuviel Platz?: umkopieren und (kleiner) neu anlegen

Immer passender Platzverbrauch?

$n$  pushBack Operationen brauchen Zeit

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

Geht es schneller?

# Unbeschränkte Felder mit teilweise ungenutztem Speicher

**Class** UArray **of** Element

$w=1 : \mathbb{N}$

$n=0 : \mathbb{N}$

**invariant**  $n \leq w < \alpha n$  or  $(n = 0 \text{ and } w \leq 2)$

$b : \text{Array } [0..w-1] \text{ of Element}$

//  $b \rightarrow$ 

$e_0$	$\dots$	$e_{n-1}$	$\dots$	$\dots$
-------	---------	-----------	---------	---------

**Operator**  $[i : \mathbb{N}] : \text{Element}$

**assert**  $0 \leq i < n$

**return**  $b[i]$

**Function** size :  $\mathbb{N}$     **return**  $n$

// allocated size

// current size

// e.g.,  $\alpha = 4$

```

Procedure pushBack( $e$  : Element)
  if  $n = w$  then
    reallocate( $2n$ )
   $b[n] := e$ 
   $n++$ 

```

```

// Example for  $n = w = 4$ :
//  $b \rightarrow$ 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
|---|---|---|---|


//  $b \rightarrow$ 

|   |   |   |   |  |  |
|---|---|---|---|--|--|
| 0 | 1 | 2 | 3 |  |  |
|---|---|---|---|--|--|


//  $b \rightarrow$ 

|   |   |   |   |     |  |
|---|---|---|---|-----|--|
| 0 | 1 | 2 | 3 | $e$ |  |
|---|---|---|---|-----|--|


//  $b \rightarrow$ 

|   |   |   |   |     |  |
|---|---|---|---|-----|--|
| 0 | 1 | 2 | 3 | $e$ |  |
|---|---|---|---|-----|--|


```

```

Procedure reallocate( $w'$  :  $\mathbb{N}$ )
   $w := w'$ 
   $b' :=$  allocate
    Array  $[0..w' - 1]$  of Element
  ( $b'[0], \dots, b'[n - 1]$ ) :=
    ( $b[0], \dots, b[n - 1]$ )
  dispose  $b$ 
   $b := b'$ 

```

```

// Example for  $w = 4, w' = 8$ :
//  $b \rightarrow$ 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
|---|---|---|---|


//  $b' \rightarrow$ 

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|


//  $b' \rightarrow$ 

|   |   |   |   |  |  |  |  |
|---|---|---|---|--|--|--|--|
| 0 | 1 | 2 | 3 |  |  |  |  |
|---|---|---|---|--|--|--|--|


//  $b \rightarrow$  

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
|---|---|---|---|


// pointer assignment  $b \rightarrow$ 

|   |   |   |   |  |  |  |  |
|---|---|---|---|--|--|--|--|
| 0 | 1 | 2 | 3 |  |  |  |  |
|---|---|---|---|--|--|--|--|


```

# Kürzen

```
Procedure popBack // Example for  $n = 5, w = 16$ :
  assert  $n > 0$  //  $b \rightarrow$ 

|   |   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|
| 0 | 1 | 2 | 3 | 4 |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|

 $n--$  //  $b \rightarrow$ 

|   |   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|
| 0 | 1 | 2 | 3 | 4 |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|


  if  $4n \leq w \wedge n > 0$  then // reduce waste of space
    reallocate( $2n$ ) //  $b \rightarrow$ 

|   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0 | 1 | 2 | 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|


```

Was geht schief, wenn man auf passende Größe kürzt?



# Amortisierte Komplexität unbeschr. Felder

Sei  $u$  ein anfangs leeres, unbeschränktes Feld.

Jede Operationenfolge  $\sigma = \langle \sigma_1, \dots, \sigma_m \rangle$

von **pushBack** oder **popBack** Operationen auf  $u$

wird in **Zeit**  $O(m)$  ausgeführt.

Sprechweise:

pushBack und popBack haben **amortisiert** konstante Ausführungszeit —

$$O\left(\underbrace{\quad}_{c \cdot m}^{\text{Gesamtzeit}} / \underbrace{\quad}_m^{\text{\#Ops}}\right) = O(1) .$$

## Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten	Typ
pushBack	oo (2 Token)	einzahlen
popBack	o (1 Token)	einzahlen
reallocate(2n)	$n \times o$ (n Token)	abheben

Zu zeigen: keine Überziehungen

Erster Aufruf von reallocate: kein Problem

( $n = 2, \geq 2$ tes pushBack)

# Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten	Typ
pushBack	oo (2 Token)	einzahlen
popBack	o (1 Token)	einzahlen
reallocate(2n)	n × o (n Token)	abheben

Weitere Aufrufe von reallocate:

$$\text{rauf: } \text{reallocate}(2n) \underbrace{\geq n \times \text{pushBack}}_{\geq n \times oo} \text{reallocate}(4n)$$

$$\text{runter: } \text{reallocate}(2n) \underbrace{\geq n/2 \times \text{popBack}}_{\geq n/2 \times o} \text{reallocate}(n)$$



# Amortisierte Analyse – allgemeiner

- ▶  $\mathcal{O}$ : Menge von Operationen, z. B. {pushBack, popBack}
- ▶  $s$  (für state): **Zustand** der Datenstruktur
- ▶  $A_{Op}(s)$ : **amortisierte Kosten** von Operation  $Op \in \mathcal{O}$  in Zustand  $s$
- ▶  $T_{Op}(s)$ : **tatsächliche Kosten** von Operation  $Op \in \mathcal{O}$  in Zustand  $s$
- ▶ **Berechnung**:  $s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \dots \xrightarrow{Op_n} s_n$

Die angenommenen amortisierten Kosten sind korrekt, wenn

$$\underbrace{\sum_{1 \leq i \leq n} T_{Op_i}(s_{i-1})}_{\text{tatsächliche Gesamtkosten}} \leq c + \underbrace{\sum_{1 \leq i \leq n} A_{Op_i}(s_{i-1})}_{\text{amortisierte Gesamtkosten}}$$

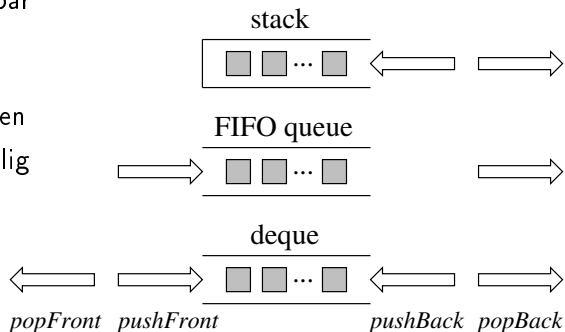
für eine Konstante  $c$

# Amortisierte Analyse – Diskussion

- ▶ **Amortisierte** Laufzeiten sind leichter zu garantieren als **tatsächliche**.
- ▶ Der **Gesamtlaufzeit** tut das keinen Abbruch.
- ▶ **Deamortisierung** oft möglich, aber kompliziert und teuer
  - ▶ Wie geht das mit unbeschränkten Feldern?
  - ▶ Anwendung: **Echtzeitsysteme**
  - ▶ Anwendung: **Parallelverarbeitung**

# Stapel und Schlangen

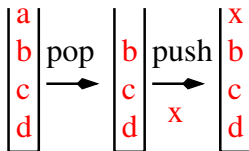
- ▶ einfache Schnittstellen
- ▶ vielseitig einsetzbar
- ▶ austauschbare, effiziente Implementierungen
- ▶ wenig fehleranfällig



# Stapel

## Operationen:

push/pop, entsprechen  
pushFront/popFront oder pushBack/popBack für Folgen



# Stapel – Implementierungsvarianten

## Operationen:

**push/pop**, entsprechen

**pushFront/popFront** oder **pushBack/popBack** für Folgen

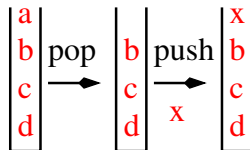
**List**: OK aber doppelte Verkettung ist overkill

**SList**: mittels **pushFront/popFront**.

Endezeiger unnötig, dummy item unnötig

**UArray**: mittels **pushBack/popBack**. Cache-effizient aber nur amortisierte konstante Laufzeit pro Operation

In der Vorlesung Algorithm Engineering lernen wir bessere Implementierungen kennen.





# Stapel – Anwendungen

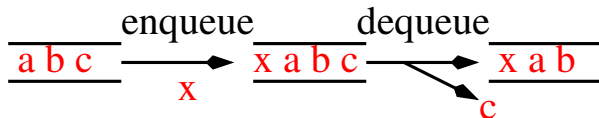
- ▶ Rekursion
- ▶ Klammerstrukturen, . . . , Parser
- ▶ Daten “irgendwie” ablegen und wieder herausholen

# Warteschlangen / First-In-First-Out / FIFO

Operationen:

`enqueue/dequeue`, entsprechen  
`pushFront/popBack` oder `pushBack/popFront` für Folgen

Beispiel:



# FIFO – Implementierungsvarianten

## Operationen:

`enqueue/dequeue`, entsprechen

`pushFront/popBack` oder `pushBack/popFront` für Folgen

`List`: OK aber doppelte Verkettung ist overkill

`SList`: mittels `pushBack/popFront`. `Endezeiger` wichtig, dummy item unnötig

`Array,UArray`: scheinbar nicht effizient möglich

`CArray`: “zyklisches” Array

Übung: unbounded cyclic array

In der Vorlesung Algorithm Engineering lernen wir bessere Implementierungen kennen.

**Class** BoundedFIFO( $n : \mathbb{N}$ ) **of** Element

$b$  : **Array** [0.. $n$ ] **of** Element // **CArray**

$h=0$  :  $\mathbb{N}$  // **head**

$t=0$  :  $\mathbb{N}$  // **tail**

**Function** isEmpty :  $\{0, 1\}$ ; **return**  $h = t$

**Function** first : Element; **assert**  $\neg$ isEmpty; **return**  $b[h]$

**Function** size :  $\mathbb{N}$ ; **return**  $(t - h + n + 1) \bmod (n + 1)$

**Procedure** pushBack( $x$  : Element)

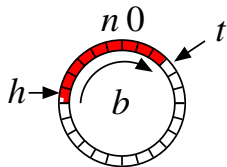
**assert** size <  $n$

$b[t] := x$

$t := (t + 1) \bmod (n + 1)$

**Procedure** popFront **assert**  $\neg$ isEmpty;  $h := (h + 1) \bmod (n + 1)$

Einsparung von einem Element durch gesonderte Variable  $s$  (für size)



# Warteschlangen – Anwendungen

- ▶ Datenpuffer für
  - ▶ Netzwerke
  - ▶ Pipeline-Verarbeitung
- ▶ Job-Queues (Fairness...)
- ▶ Breitensuche in Graphen (siehe Kapitel 9.1)

# Deque – Double-Ended Queues

Aussprache wie “dek”.

List: OK

SList: Nein (aber push/pop-Front und pushBack OK)

Array,UArray: Nein

CArray: Ja

Übung: Pseudocode für Deque mittels CArray

# Deque – Anwendungen

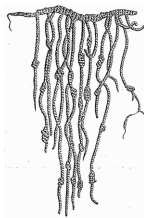
relativ selten. Oft werden nur 3 der vier Operationen benötigt.

- ▶ Work Stealing Load Balancing
- ▶ Undo/Redo Operationspuffer

# Vergleich: Listen – Felder

## Vorteile von Listen

- ▶ flexibel
- ▶ remove, splice, ...
- ▶ kein Verschnitt



## Vorteile von Feldern

- ▶ beliebiger Zugriff
- ▶ einfach
- ▶ kein Overhead für Zeiger
- ▶ Cache-effizientes scanning





Operation	List	SList	UArray	CArray	explanation '*'
[.]	$n$	$n$	1	1	
.	1*	1*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1*	$n$	$n$	insertAfter only
remove	1	1*	$n$	$n$	removeAfter only
pushBack	1	1	1*	1*	amortized
pushFront	1	1	$n$	1*	amortized
popBack	1	$n$	1*	1*	amortized
popFront	1	1	$n$	1*	amortized
concat	1	1	$n$	$n$	
splice	1	1	$n$	$n$	
findNext,...	$n$	$n$	$n^*$	$n^*$	cache-efficient

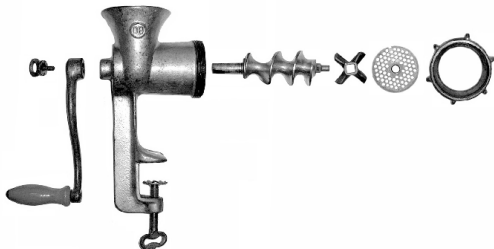
# Ausblick: Weitere Repräsentationen von Folgen

Hashtabellen: schnelles Einfügen, Löschen und Suchen Kapitel 4

Prioritätslisten: schnelles Einfügen, Minimum Entfernen Kapitel 6

Suchbäume, . . . : sortierte Folgen – einfügen, löschen, suchen,  
Bereichsanfragen, . . . Kapitel 7

# Hashing (Streuspeicherung)



“to **hash**”  $\approx$  “völlig **durcheinander** bringen”.  
Paradoxerweise **hilft** das, Dinge wiederzufinden

# Hashtabellen

speichere Menge  $M \subseteq \text{Element}$ .

$\text{key}(e)$  ist eindeutig für  $e \in M$ .

unterstütze **Wörterbuch**-Operationen in Zeit  $O(1)$ .

$M.\text{insert}(e : \text{Element})$ :  $M := M \cup \{e\}$

$M.\text{remove}(k : \text{Key})$ :  $M := M \setminus \{e\}$ ,  $\text{key}(e) = k$

$M.\text{find}(k : \text{Key})$ : return  $e \in M$  with  $\text{key}(e) = k$ ;  $\perp$  falls nichts gefunden

Anderes Interface: **map/partielle Funktion**  $\text{Key} \rightarrow \text{Element}$

$M[k] = M.\text{find}(k)$

## Exkurs: Konventionen für Elemente

Viele Datenstrukturen repräsentieren Mengen  
(engl. auch *collection classes*).

Die Mengenelemente  $e$  haben Schlüssel  $\text{key}(e)$ .

Elementvergleich hier gleichbedeutend mit Schlüsselvergleich.

$e = e'$  gdw.  $\text{key}(e) = \text{key}(e')$  (analog für  $e < e'$  und  $e > e'$ ).

# Hashing: Anwendungen

- ▶ Auslieferungsregale der UB Karlsruhe
- ▶ Entfernen exakter **Duplikate**
- ▶ Schach (oder andere kombinatorische Suchprogramme):  
**welche** Stellungen wurden **bereits durchsucht**?
- ▶ **Symboltabelle** bei Compilern
- ▶ **Assoziative Felder** bei Script-Sprachen wie perl oder python
- ▶ Datenbank-Gleichheits-**Join**  
(wenn eine Tabelle in den Speicher passt)
- ▶ Routenplaner: **Teilmengen** von Knoten, z. B. Suchraum
- ▶ ...

# Überblick

- ▶ Grundidee
- ▶ Hashing mit verketteten Listen
- ▶ Analyse
- ▶ Hashing mit Arrays

# Erste Ideen zu Implementierungen

speichere Menge  $M \subseteq$  Element.

$\text{key}(e)$  ist eindeutig für  $e \in M$ .

unterstütze **Wörterbuch**-Operationen in Zeit  $O(1)$ .

**Implementierung mit Listen**: Wörterbuchoperationen zu aufwändig

**Implementierung mit Feldern**: Elemente **wo** ablegen?

$\rightsquigarrow$   $\text{key}(e)$  legt fest, wo  $e$  abgelegt wird



# Ein (über)optimistischer Ansatz

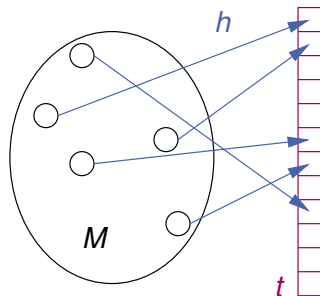
Eine perfekte **Hash-Funktion**  $h$   
bildet Elemente von  $M$  **injektiv**  
auf eindeutige Einträge  
der **Tabelle**  $t[0..m-1]$  ab, d. h.,  
 $t[h(\text{key}(e))] = e$

**Datenstrukturinvariante:**

$$\forall e \in M : t[h(\text{key}(e))] = e$$

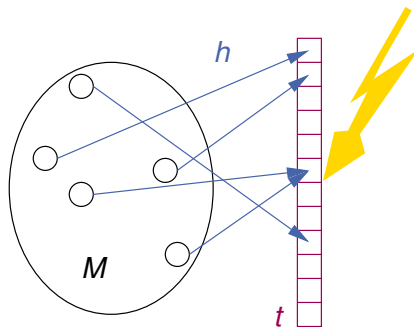
$\wedge$

$$\forall 0 \leq i < m : t[i] \in M \cup \{\perp\}$$



# Kollisionen

Perfekte Hash-Funktionen sind schwer zu finden

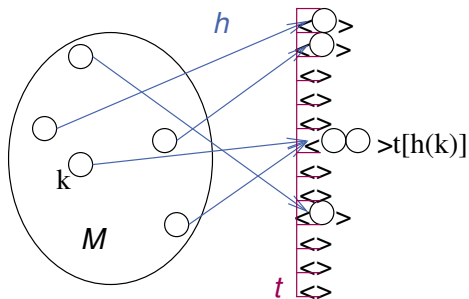


Beispiel: Geburtstagsparadox

# Kollisionsauflösung

Eine Möglichkeit:

Tabelleneinträge: Elemente  $\rightsquigarrow$  Folgen von Elementen



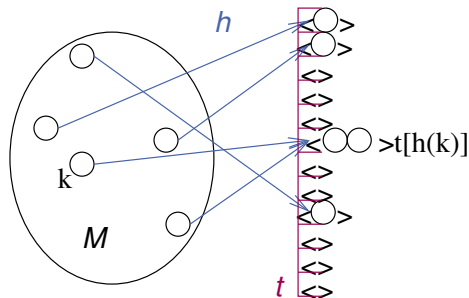
# Hashing mit verketteten Listen

Implementiere die Folgen in den Tabelleneinträgen  
durch **einfach verkettete Listen**

**Datenstrukturinvariante:**

$$\forall e \in M : e \in t[h(\text{key}(e))]$$

$$\wedge \\ \forall 0 \leq i < m : t[i] \subseteq M$$



# Hashing mit verketteten Listen

Implementiere die Folgen in den Tabelleneinträgen durch **einfach verkettete Listen**

$\text{insert}(e)$ : Füge  $e$  am Anfang von  $t[h(\text{key}(e))]$  ein.

$\text{remove}(k)$ : Durchlaufe  $t[h(k)]$ .

Element  $e$  mit  $\text{key}(e) = k$  gefunden?

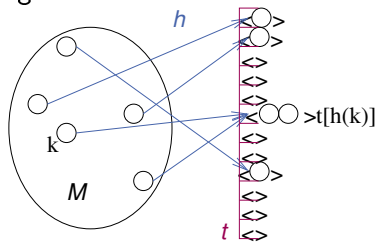
$\rightsquigarrow$  löschen und zurückliefern.

$\text{find}(k)$ : Durchlaufe  $t[h(k)]$ .

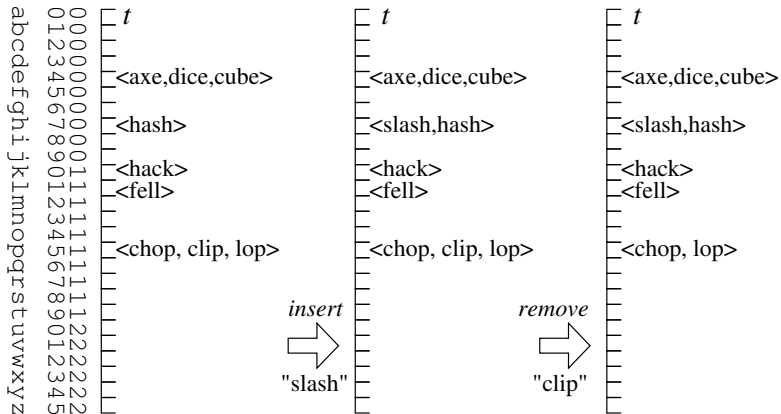
Element  $e$  mit  $\text{key}(e) = k$  gefunden?

$\rightsquigarrow$  zurückliefern.

Sonst:  $\perp$  zurückgeben.



# Beispiel



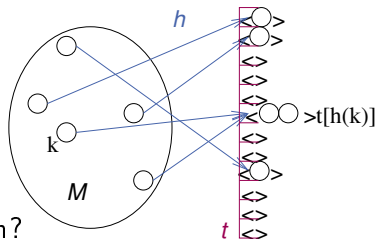
# Analyse

$\text{insert}(e)$ : konstante Zeit  
 $\text{remove}(k)$ :  $O(\text{Listenlänge})$   
 $\text{find}(k)$ :  $O(\text{Listenlänge})$

Aber wie lang werden die Listen?

Schlechtester Fall:  $O(|M|)$

Besser wenn wir genug Chaos anrichten?



- ▶ Hashtabellen (mit verketteten Listen)
  - ▶ Datenstruktur für Mengen mit schnellen Wörterbuchoperationen
  - ▶ Schnelles Wiederfinden von Elementen durch möglichst chaotische Anordnung im Speicher
  - ▶ Hashfunktion (angewandt auf Key) sagt, wo Element gespeichert
  - ▶ Bei Kollisionen: speichere (Zeiger auf) Liste von Elementen
  - ▶ (Idee auf nächster Folie)
- ▶ **Heute:**
  - ▶ Wie groß muss der Speicherbereich für Hashtabelle sein?
  - ▶ Gute Hashfunktionen?
  - ▶ Hierzu: etwas Wahrscheinlichkeitstheorie
  - ▶ Alternative: Hashing mit linearer Suche



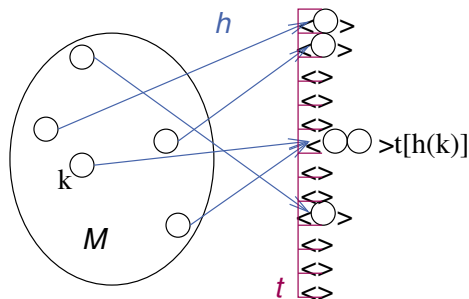
# Hashing mit verketteten Listen (Wdh.)

Implementiere die Folgen in den Tabelleneinträgen durch **einfach verkettete Listen**

**Datenstrukturinvariante:**

$$\forall e \in M : e \in t[h(\text{key}(e))]$$

$$\wedge \\ \forall 0 \leq i < m : t[i] \subseteq M$$



# Etwas Wahrscheinlichkeitstheorie für den Hausgebrauch



Elementarereignisse  $\Omega$

Ereignisse: Teilmengen von  $\Omega$

$p_x =$  Wahrscheinlichkeit von  $x \in \Omega$ .  $\sum_x p_x = 1$  !

Gleichverteilung:  $p_x = \frac{1}{|\Omega|}$

$\mathbb{P}[\mathcal{E}] = \sum_{x \in \mathcal{E}} p_x$

Zufallsvariable (ZV)  $X : \Omega \rightarrow \mathbb{R}$

0-1-Zufallsvariable (Indikator-ZV)  $I : \Omega \rightarrow \{0, 1\}$

Erwartungswert  $E[X] = \sum_{y \in \Omega} p_y X(y)$

Linearität des Erwartungswerts:  $E[X + Y] = E[X] + E[Y]$

Hash-Beispiel

Hash-Funktionen  $\{0..m-1\}^{\text{Key}}$

$\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$

$p_h = m^{-|\text{Key}|}$

$\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$

$X = |\{e \in M : h(e) = 0\}|$

$E[X] = \frac{|M|}{m}$

## Beispiel: Variante des Geburtstagsparadoxon

Wieviele Gäste muss eine Geburtstagsparty “im Mittel” haben, damit mindestens zwei Gäste den gleichen Geburtstag haben?

Gäste (Keys)  $1..n$ .

Elementarereignisse:  $h \in \Omega = \{0..364\}^{\{1..n\}}$ .

Definiere Indikator-ZV  $I_{ij} = 1$  gdw  $h(i) = h(j)$ .

Anzahl Paare mit gleichem Geburtstag:  $X = \sum_{i=1}^n \sum_{j=i+1}^n I_{ij}$ .

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^n \sum_{j=i+1}^n I_{ij}\right] = \sum_{i=1}^n \sum_{j=i+1}^n E[I_{ij}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{P}[I_{ij} = 1] = \frac{n(n-1)}{2} \cdot \frac{1}{365} \\ &\stackrel{!}{=} 1 \Leftrightarrow n = \frac{1}{2} + \sqrt{\frac{1}{2^2} + 730} \approx 27.52 \end{aligned}$$

# Mehr zum Geburtstagsparadoxon

Standardformulierung:

Ab wann lohnt es sich zu **wetten**, dass es zwei Gäste mit gleichem Geburtstag gibt? Etwas komplizierter. Antwort:  $n \geq 23$

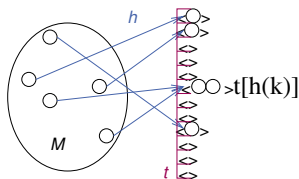
Verallgemeinerung: Jahreslänge  $m =$  Hashtabelle der Größe  $m$ : eine zufällige Hashfunktion  $h : 1..n \rightarrow 0..m-1$  ist nur dann mit vernünftiger Wahrscheinlichkeit **perfekt** wenn  $m = \Omega(n^2)$ .

Riesige Platzverschwendung.

# Analyse für zufällige Hash-Funktionen

## Theorem 1

$\forall k$ : die erwartete Anzahl kollidierender Elemente ist  $O(1)$  falls  $|M| \in O(m)$ .



## Beweis.

Für festen Schlüssel  $k$  definiere **Kollisionslänge**  $X$

$X := |\{e \in M' : h(e) = h(k)\}|$  mit  $M' = \{e \in M : \text{key}(e) \neq k\}$ .

Betrachte die 0-1 ZV  $X_e = 1$  für  $h(e) = h(k)$ ,  $e \in M'$  und  $X_e = 0$  sonst.

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in M'} X_e\right] = \sum_{e \in M'} \mathbb{E}[X_e] = \sum_{e \in M'} \mathbb{P}[X_e = 1] = \frac{|M'|}{m} \\ &\in O(1) \end{aligned}$$

Das gilt **unabhängig** von der Eingabe  $M$ . □

# Zufällige Hash-Funktionen?

Naive Implementierung: ein **Tabelleneintrag pro Schlüssel**.

↪ **meist zu teuer**

Weniger naive Lösungen: kompliziert, immer noch viel Platz.

↪ **meist unsinnig**

↪ **unrealistisch**

# Universelles Hashing

Idee: nutze nur bestimmte “einfache” Hash-Funktionen

## Definition 2

$\mathcal{H} \subseteq \{0..m-1\}^{\text{Key}}$  ist *universell*

falls für alle  $x, y$  in Key mit  $x \neq y$  und zufälligem  $h \in \mathcal{H}$ ,

$$\mathbb{P}[h(x) = h(y)] = \frac{1}{m} .$$

## Theorem 3

*Theorem 1 gilt auch für universelle Familien von Hash-Funktionen.*

## Beweis.

Für  $\Omega = \mathcal{H}$  haben wir immer noch  $\mathbb{P}[X_e = 1] = \frac{1}{m}$ .

Der Rest geht wie vorher. □



# Eine einfache universelle Familie

$m$  sei eine Primzahl,  $\text{Key} \subseteq \{0, \dots, m-1\}^k$

## Theorem 4

Für  $\mathbf{a} = (a_1, \dots, a_k) \in \{0, \dots, m-1\}^k$  definiere

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \bmod m, \quad H = \{h_{\mathbf{a}} : \mathbf{a} \in \{0, \dots, m-1\}^k\}.$$

$H$  ist eine universelle Familie von Hash-Funktionen

$$\left( \begin{array}{|c|c|c|} \hline x_1 & x_2 & x_3 \\ \hline * & * & * \\ \hline a_1 & a_2 & a_3 \\ \hline \end{array} \right) \bmod m = h_{\mathbf{a}}(\mathbf{x})$$



## Beispiel für $H$

Für  $\mathbf{a} = (a_1, \dots, a_k) \in \{0, \dots, m-1\}^k$  definiere  
 $h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \bmod m$ ,  $H = \{h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k\}$ .

$k = 3$ ,  $m = 11$

wähle  $\mathbf{a} = (8, 1, 5)$ .

$h_{\mathbf{a}}((1, 1, 2)) = (8, 1, 5) \cdot (1, 1, 2) = 8 \cdot 1 + 1 \cdot 1 + 5 \cdot 2 = 19 \equiv 8 \bmod 11$

## Beweis.

Betrachte  $\mathbf{x} = (x_1, \dots, x_k)$ ,  $\mathbf{y} = (y_1, \dots, y_k)$  mit  $x_j \neq y_j$   
zähle  $\mathbf{a}$  mit  $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$ .

Für jede Wahl der  $a_i$ ,  $i \neq j$ ,  $\exists$  genau ein  $a_j$  mit  $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$ :

$$\begin{aligned}\sum_{1 \leq i \leq k} a_i x_i &\equiv \sum_{1 \leq i \leq k} a_i y_i \pmod{m} \\ \Leftrightarrow a_j(x_j - y_j) &\equiv \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) \pmod{m} \\ \Leftrightarrow a_j &\equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) \pmod{m}\end{aligned}$$

$m^{k-1}$  Möglichkeiten die  $a_i$  (mit  $i \neq j$ ) auszuwählen.

$m^k$  ist die Gesamtzahl der  $\mathbf{a}$ , d. h.,

$$\mathbb{P}[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}.$$



# Bit-basierte Universelle Familien

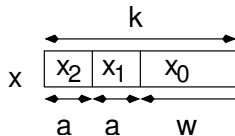
Sei  $m = 2^w$ ,  $\text{Key} = \{0, 1\}^k$

**Bit-Matrix Multiplikation:**  $H^\oplus = \{h_M : M \in \{0, 1\}^{w \times k}\}$

wobei  $h_M(x) = Mx$  (Arithmetik mod 2, d. h., xor, and)

**Tabellenzugriff:**  $H^{\oplus[]} = \{h_{(t_1, \dots, t_b)}^\oplus : t_i \in \{0..m-1\}^{\{0..2^a-1\}}\}$

wobei  $h_{(t_1, \dots, t_b)}^\oplus((x_0, x_1, \dots, x_b)) = x_0 \oplus \bigoplus_{i=1}^b t_i[x_i]$



# Hashing mit Linearer Suche (Linear Probing)

Zurück zur Ursprungsidee.

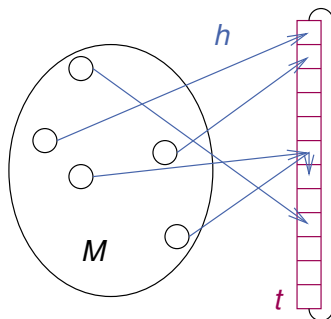
Elemente werden direkt in der Tabelle gespeichert.

Kollisionen werden durch Finden anderer Stellen aufgelöst.

**linear probing**: Suche nächsten freien Platz.

Am Ende fange von vorn an.

- ▶ einfach
- ▶ platz-effizient
- ▶ cache-effizient



# Der einfache Teil

**Class** BoundedLinearProbing( $m, m' : \mathbb{N}; h : \text{Key} \rightarrow 0..m-1$ )

$t = [\perp, \dots, \perp] : \mathbf{Array}$   $[0..m+m'-1]$  of Element

**invariant**  $\forall i : t[i] \neq \perp \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[j] \neq \perp$

**Procedure** insert( $e : \text{Element}$ )

**for** ( $i := h(e); t[i] \neq \perp; i++$ ) ;

**assert**  $i < m + m' - 1$

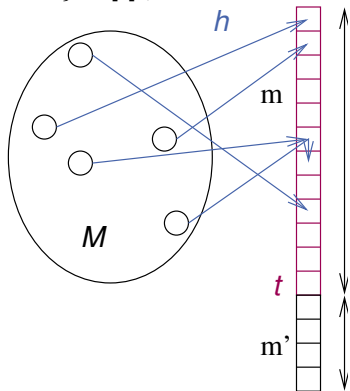
$t[i] := e$

**Function** find( $k : \text{Key}$ ) : Element

**for** ( $i := h(k); t[i] \neq \perp; i++$ )

**if**  $t[i] = k$  **then return**  $t[i]$

**return**  $\perp$



## Remove

Beispiel:  $t = [\dots, \underset{h(z)}{x}, y, z, \dots]$ ,  $\text{remove}(x)$

**invariant**  $\forall i : t[i] \neq \perp \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[j] \neq \perp$

**Procedure**  $\text{remove}(k : \text{Key})$


```
for (  $i := h(k)$ ;  $k \neq t[i]$ ;  $i++$  ) // search  $k$   
    if  $t[i] = \perp$  then return // nothing to do
```

// we plan for a hole at  $i$ .

```
for (  $j := i + 1$ ;  $t[j] \neq \perp$ ;  $j++$  )  
    // Establish invariant for  $t[j]$ .  
    if  $h(t[j]) \leq i$  then  
         $t[i] := t[j]$  // Overwrite removed element  
         $i := j$  // move planned hole  
 $t[i] := \perp$  // erase freed entry
```

*insert* : axe, chop, clip, cube, dice, fell, hack, hash, lop, slash

	an	bo	cp	dq	er	fs	gt	hu	iv	jw	kx	ly	mz
<i>tt</i>	0	1	2	3	4	5	6	7	8	9	10	11	12
	⊥	⊥	⊥	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	slash	hack	fell	⊥

*remove*  clip

	⊥	⊥	chop	<del>clip</del>	axe	cube	dice	hash	lop	slash	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	<del>lop</del>	slash	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	slash	<del>slash</del>	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	slash	⊥	hack	fell	⊥

# Verketteten $\leftrightarrow$ Lineare Suche

**Volllaufen:** Verketteten weniger empfindlich.  
Unbeschränktes Hashing mit lin. Suche hat nur  
amortisiert konst. Einfügezeit

**Cache:** Lineare Suche besser. Vor allem für **doall**

**Platz/Zeit Abwägung:** Kompliziert! Abhängig von  $n$ , **Füllgrad**,  
**Elementgröße**, Implementierungsdetails bei Verketteten  
(shared dummy!,  $t$  speichert Zeiger oder item),  
Speicherverwaltung bei Verketteten, beschränkt oder  
nicht,...

**Referentielle Integrität:** Nur bei Verketteten !

**Leistungsgarantien:** Universelles Hashing funktioniert so nur mit  
Verketteten



# Perfektes Hashing

hier nicht

# Mehr Hashing

- ▶ Hohe Wahrscheinlichkeit, Garantien für den schlechtesten Fall, Garantien für linear probing  
     $\rightsquigarrow$  höhere Anforderungen an die Hash-Funktionen
- ▶ Hashing als Mittel zur Lastverteilung z. B., storage servers, (peer to peer Netze, . . . )
- ▶ Zufallsextraktion durch universelles Hashing
- ▶  $O(1)$  find / perfektes Hashing

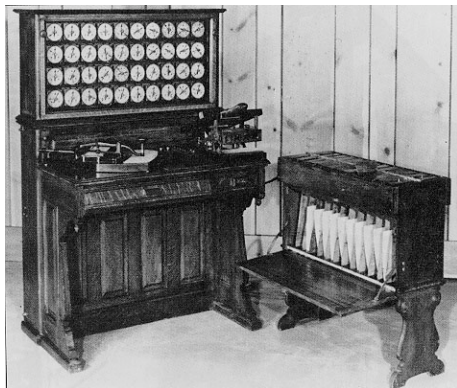
# Hashtabellen für assoziative Arrays

- ▶ In Java: `java.util.Hashtable`, in Python: `dict`
- ▶ Beispiel (Python): `args["username"]="fred"` (CGI-Skripte)
- ▶ Beispiel für Sicherheitsproblem (gelöst seit 2012):
  - ▶ Webserver beantwortet HTTP-Anfragen mit Python-Skript
  - ▶ HTTP-Parameter werden für Skript in `dict` eingelesen
  - ▶ Problem: Hashfunktion  $h$  deterministisch (d.h. fest, bekannt!)
  - ▶ Böartige Anfrage: viele HTTP-Parameter  $x \neq y$  mit gleichem  $h(x)$
  - ▶ Konsequenz: quadratischer Aufwand bei Einlesen/Verarbeiten von Skriptparametern → **Denial of Service**
- ▶ Wie reparieren?

# Kryptographische Hashfunktionen

- ▶ Universelle Hashfunktionen: kurze Ausgabe (typisch:  $\leq 32$  Bit), Kollisionen „gleichmäßig“
- ▶ Kryptographische Hashfunktionen: längere Ausgabe (typisch: 128–512 Bit), Kollisionen existieren, sind aber schwer zu finden
- ▶ Anwendungen für kryptographische Hashfunktionen:
  - ▶ „Fingerabdruck“ von großen Datenmengen
  - ▶ Zertifikate (z.B. für Webserver)
  - ▶ Nutzerauthentifikation (Unix: /etc/passwd)
- ▶ Beispiele: MD5 (unsicher), SHA-1 (unsicher), SHA-256, SHA-3
- ▶ Mehr in Sicherheits-Vorlesung

# Sortieren & Co



# Formaler

Gegeben: Elementfolge  $s = \langle e_1, \dots, e_n \rangle$

Gesucht:  $s' = \langle e'_1, \dots, e'_n \rangle$  mit

- ▶  $s'$  ist Permutation von  $s$
- ▶  $e'_1 \leq \dots \leq e'_n$  für eine **Totalordnung** ' $\leq$ '

# Anwendungsbeispiele

- ▶ Allgemein: Vorverarbeitung
- ▶ Suche: **Telefonbuch**  $\leftrightarrow$  unsortierte Liste
- ▶ Gruppieren (Alternative Hashing?)

# Beispiele aus Kurs/Buch

- ▶ Aufbau von Suchbäumen
- ▶ Kruskals MST-Algorithmus
- ▶ Rucksackproblem
- ▶ Scheduling, die schwersten Probleme zuerst
- ▶ Sekundärspeicheralgorithmen, z. B. Datenbank-**Join**

Viele verwandte Probleme. Zum Beispiel **Transposition** dünner Matrizen, **invertierten Index** aufbauen, Konversion zwischen Graphrepräsentationen.



# Überblick

- ▶ Einfache Algorithmen / kleine Datenmengen
- ▶ **Mergesort** – ein erster effizienter Algorithmus
- ▶ Eine passende **untere Schranke**
- ▶ **Quicksort**
- ▶ das Auswahlproblem
- ▶ ganzzahlige Schlüssel – jenseits der unteren Schranke

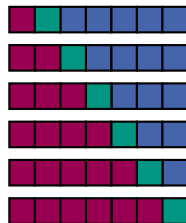
# Einfache Sortieralgorithmen

**Procedure** insertionSort( $a$  : **Array** [1.. $n$ ] of Element)

**for**  $i := 2$  **to**  $n$  **do**

**invariant**  $a[1] \leq \dots \leq a[i-1]$

    move  $a[i]$  to the right place



Beispiel:

$\langle 4 \rangle, \langle 7, 1, 1 \rangle \rightsquigarrow \langle 4, 7 \rangle, \langle 1, 1 \rangle \rightsquigarrow \langle 1, 4, 7 \rangle, \langle 1 \rangle \rightsquigarrow \langle 1, 1, 4, 7 \rangle, \langle \rangle$

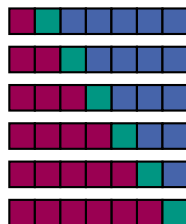
## Sentinels am Beispiel Sortieren durch Einfügen

```
Procedure insertionSort( $a$  : Array [1.. $n$ ] of Element)
  for  $i$  := 2 to  $n$  do
    invariant  $a[1] \leq \dots \leq a[i-1]$ 
    // move  $a[i]$  to the right place
     $e := a[i]$ 
    if  $e < a[1]$  then // new minimum
      for  $j$  :=  $i$  downto 2 do  $a[j] := a[j-1]$ 
       $a[1] := e$ 
    else // use  $a[1]$  as a sentinel
      for ( $j := i$ ;  $a[j-1] > e$ ;  $j--$ )  $a[j] := a[j-1]$ 
       $a[j] := e$ 
```

# Analyse

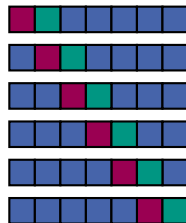
Die  $i$ -te Iteration braucht Zeit  $\Theta(i)$ .

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



Die  $i$ -te Iteration braucht Zeit  $O(1)$  z. B. (beinahe) sortiert.

$$\sum_{i=2}^n O(1) \in O(n)$$



# Sortieren durch Mischen

Idee: Teile und Herrsche

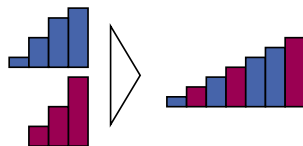
```
Function mergeSort( $\langle e_1, \dots, e_n \rangle$ ) : Sequence of Element  
  if  $n = 1$  then return  $\langle e_1 \rangle$  // base case  
  else return merge( mergeSort( $\langle e_1, \dots, e_{\lfloor n/2 \rfloor} \rangle$ ),  
                     mergeSort( $\langle e_{\lfloor n/2 \rfloor + 1}, \dots, e_n \rangle$ ))
```

Gegeben:

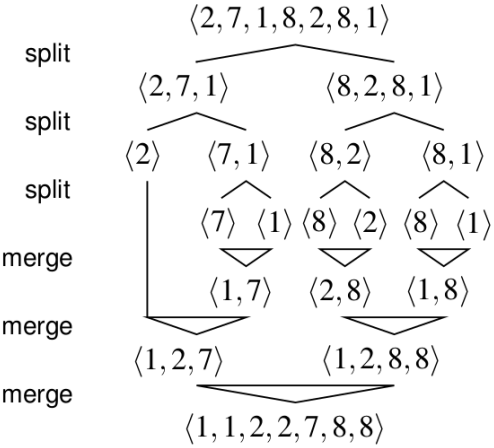
zwei **sortierte Folgen**  $a$  und  $b$

Berechne:

sortierte Folge der Elemente aus  $a$  und  $b$



# Beispiel



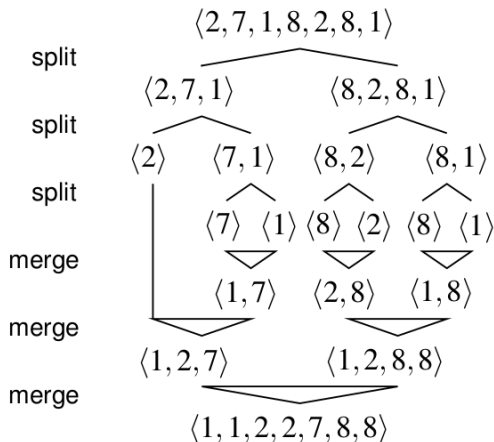
# Mischen

Jeweils  $\min(a, b)$  in die Ausgabe schieben.

Zeit  $O(n)$

<i>a</i>	<i>b</i>	<i>c</i>	operation
$\langle 1, 2, 7 \rangle$	$\langle 1, 2, 8, 8 \rangle$	$\langle \rangle$	move <i>a</i>
$\langle 2, 7 \rangle$	$\langle 1, 2, 8, 8 \rangle$	$\langle 1 \rangle$	move <i>b</i>
$\langle 2, 7 \rangle$	$\langle 2, 8, 8 \rangle$	$\langle 1, 1 \rangle$	move <i>a</i>
$\langle 7 \rangle$	$\langle 2, 8, 8 \rangle$	$\langle 1, 1, 2 \rangle$	move <i>b</i>
$\langle 7 \rangle$	$\langle 8, 8 \rangle$	$\langle 1, 1, 2, 2 \rangle$	move <i>a</i>
$\langle \rangle$	$\langle 8, 8 \rangle$	$\langle 1, 1, 2, 2, 7 \rangle$	concat <i>b</i>
$\langle \rangle$	$\langle \rangle$	$\langle 1, 1, 2, 2, 7, 8, 8 \rangle$	

# Analyse



Analyse:  $T(n) = O(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) = O(n \log n)$ .



$$T(n) = \Theta(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$$

Problem: **Runderei**

Ausweg: genauer rechnen (siehe Buch)

Dirty trick:

Eingabe auf **Zweierpotenz** aufblasen

(z. B.  $(2^{\lceil \log n \rceil} - n) \times \infty$  anhängen)

↪

normales Master-Theorem anwendbar

Zeit  $\Theta(n \log n)$

# Untere Schranken

Geht es schneller als  $\Theta(n \log n)$ ?

**Unmöglichkeit** einer Verbesserung i.allg. **schwer zu beweisen** –  
sie erfordert eine Aussage über alle **denkbaren** Algorithmen.

↔

einschränkende Annahmen

# Eine vergleichsbasierte untere Schranke

Vergleichsbasiertes Sortieren: Informationen über Elemente nur durch Zwei-Wege-Vergleich  $e_i \leq e_j$ ?

**Satz:** Deterministische vergleichsbasierte Sortieralgorithmen brauchen

$$n \log n - O(n)$$

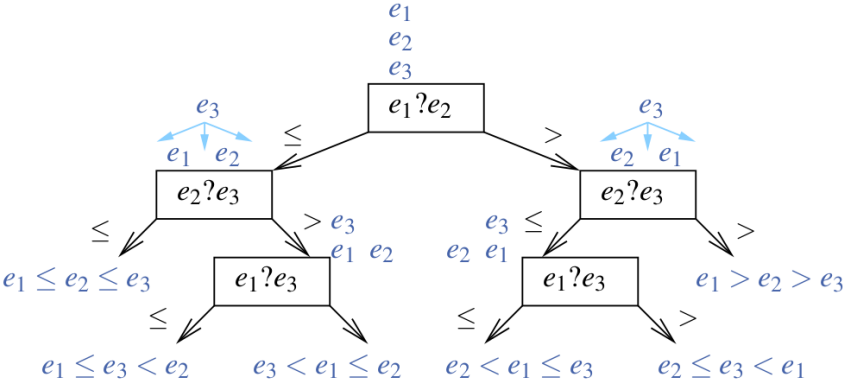
Vergleiche im schlechtesten Fall.

**Beweis:**

Betrachte Eingaben, die Permutationen von  $1..n$  sind.

Es gibt genau  $n!$  solche Permutationen.

# Baumbasierte Sortierer-Darstellung



Mindestens ein Blatt pro Permutation von  $e_1, \dots, e_n$

Ausführungszeit entspricht **Tiefe  $T$**

# Beweis

Baum der **Tiefe**  $T$  hat höchstens  $2^T$  Blätter.

$$\Rightarrow 2^T \geq n!$$

$$\Leftrightarrow T \geq \underbrace{\log n!}_{\geq \left(\frac{n}{e}\right)^n} \geq \log \left(\frac{n}{e}\right)^n = n \log n - n \log e = n \log n - O(n)$$

Einfache Approximation der Fakultät:  $\left(\frac{n}{e}\right)^n \leq n! \leq n^n$

Beweis für **linken Teil**:

$$\ln n! = \sum_{2 \leq i \leq n} \ln i \geq \int_1^n \ln x \, dx = \left[ x(\ln x - 1) \right]_{x=1}^{x=n} \geq n(\ln n - 1) .$$

$$\Rightarrow n! \geq e^{n(\ln n - 1)} = \frac{e^{n \ln n}}{e^n} = \frac{n^n}{e^n} = \left(\frac{n}{e}\right)^n$$

# Randomisierung, Mittlere Ausführungszeit

Satz: immer noch  $n \log n - O(n)$  Vergleiche.

Beweis: nicht hier.

# Erinnerung: Mergesort

Idee: Teile und Herrsche

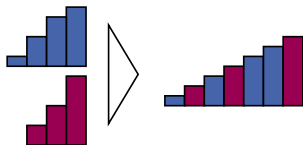
```
Function mergeSort( $\langle e_1, \dots, e_n \rangle$ ) : Sequence of Element  
  if  $n = 1$  then return  $\langle e_1 \rangle$  // base case  
  else return merge( mergeSort( $\langle e_1, \dots, e_{\lfloor n/2 \rfloor} \rangle$ ),  
                     mergeSort( $\langle e_{\lfloor n/2 \rfloor + 1}, \dots, e_n \rangle$ ))
```

Gegeben:

zwei **sortierte Folgen**  $a$  und  $b$

Berechne:

sortierte Folge der Elemente aus  $a$  und  $b$



# Quicksort – erster Versuch

Idee: Teile-und-Herrsche aber verglichen mit mergesort „andersrum“.  
Leiste Arbeit **vor** rekursivem Aufruf

```
Function quickSort( $s$  : Sequence of Element) : Sequence of Element
  if  $|s| \leq 1$  then return  $s$ 
  pick “some”  $p \in s$ 
   $a := \langle e \in s : e < p \rangle$ 
   $b := \langle e \in s : e = p \rangle$ 
   $c := \langle e \in s : e > p \rangle$ 
  return concatenation of quickSort( $a$ ),  $b$ , and quickSort( $c$ )
```



# Quicksort – Analyse im schlechtesten Fall

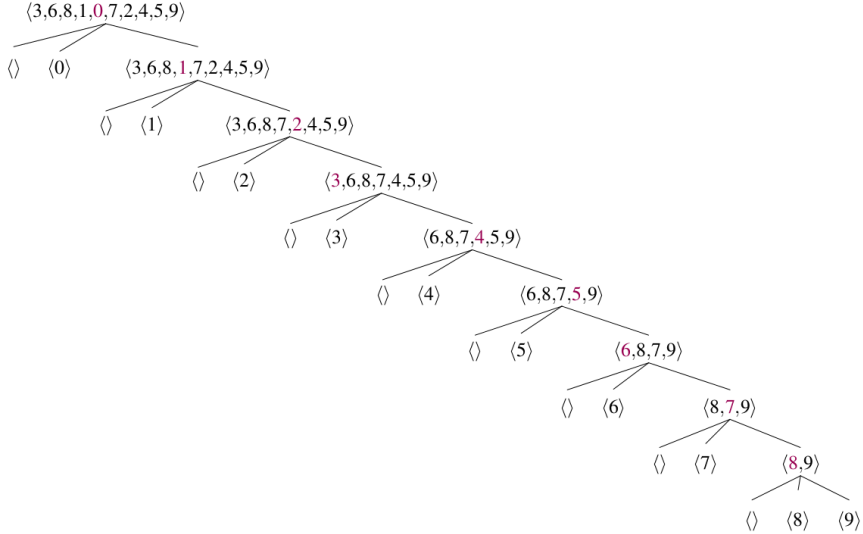
Annahme: Pivot ist immer **Minimum** (oder Max.) der Eingabe

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \Theta(n) + T(n-1) & \text{if } n \geq 2. \end{cases}$$

$\Rightarrow$

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

# Schlechtester Fall: Beispiel



# Quicksort – Analyse im besten Fall

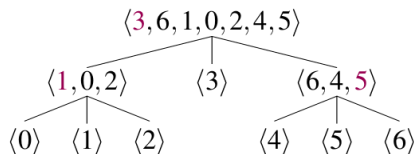
Annahme: Pivot ist immer **Median** der Eingabe

$$T(n) \leq \begin{cases} O(1) & \text{if } n = 1, \\ O(n) + 2T(\lfloor n/2 \rfloor) & \text{if } n \geq 2. \end{cases}$$

⇒ (Master-Theorem)

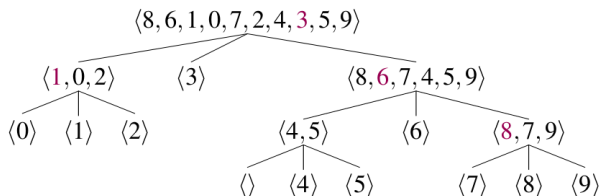
$$T(n) = O(n \log n)$$

Problem: Median bestimmen ist nicht so einfach



# Quicksort – zufälliger Pivot

**Function** quickSort( $s$  : Sequence of Element) : Sequence of Element  
**if**  $|s| \leq 1$  **then return**  $s$   
**pick**  $p \in s$  **uniformly at random**  
 $a := \langle e \in s : e < p \rangle$   
 $b := \langle e \in s : e = p \rangle$   
 $c := \langle e \in s : e > p \rangle$   
**return** concatenation of quickSort( $a$ ),  $b$ , and quickSort( $c$ )



## Satz: Quicksort hat erwartete Laufzeit $O(n \log n)$

Annahme: alle Elemente **verschieden**

Warum 'OBdA' ?

Es genügt, die **3-Wege** Vergleiche ( $<, =, >$ )  $C(n)$  zu zählen.

Genauer: wir bestimmen  $\bar{C}(n) = E[C(n)]$

**Function** quickSort( $s$  : Sequence of Element) : Sequence of Element

**if**  $|s| \leq 1$  **then return**  $s$

**pick**  $p \in s$  **uniformly at random**

$a := \langle e \in s : e < p \rangle$  //  $|s| - 1$

$b := \langle e \in s : e = p \rangle$  // 3-Wege-

$c := \langle e \in s : e > p \rangle$  // Vergleiche

**return** concatenation of quickSort( $a$ ),  $b$ , and quickSort( $c$ )

# Beweisansatz 1: Rekurrenzen

## Beweis:

Im Buch wird bewiesen, dass mit Wahrscheinlichkeit  $1/2$  das Aufspaltverhältnis nicht schlechter als  $\frac{1}{4} : \frac{3}{4}$  ist.

Das genügt um  $\bar{C}(n) = O(n \log n)$  zu zeigen.

**Satz:**  $\bar{C}(n) \leq 2n \ln n \leq 1.45n \log n$

Satz:  $\bar{C}(n) \leq 2n \ln n \leq 1.45n \log n$

Sei  $s' = \langle e'_1, \dots, e'_n \rangle$  sortierte Eingabefolge.

Indikatorzufallsvariable:  $X_{ij} := 1$  gdw.  $e'_i$  wird mit  $e'_j$  verglichen.

$$\bar{C}(n) = E \left[ \sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{P}[X_{ij} = 1] .$$

$$\text{Lemma: } \mathbb{P}[X_{ij} = 1] = \frac{2}{j-i+1}$$

Sortierte Eingabefolge:

$$s' = \langle e'_1, \dots, e'_{i-1}, \underbrace{e'_i, e'_{i+1}, \dots, e'_{j-1}, e'_j}_{j-i+1 \text{ Elemente}}, e'_{j+1}, \dots, e'_n \rangle$$

$$X_{ij} = 1$$

$\Leftrightarrow$

$e'_i$  wird mit  $e'_j$  verglichen

$\Leftrightarrow$

$e'_i$  oder  $e'_j$  wird Pivot bevor ein Pivot aus  $\langle e'_{i+1}, \dots, e'_{j-1} \rangle$  gewählt wird.

$\Rightarrow$

$$\mathbb{P}[X_{ij} = 1] = \frac{2}{j-i+1}$$

□



Satz:  $\bar{C}(n) \leq 2n \ln n \leq 1.45n \log n$

$$\begin{aligned}
 \bar{C}(n) &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\
 &= \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{2}{k} \\
 &\leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \\
 &= 2n \sum_{k=2}^n \frac{1}{k} \\
 &= 2n(H_n - 1) \leq 2n(1 + \ln n - 1) = 2n \ln n .
 \end{aligned}$$

$i$	$j$	$\overbrace{j-i+1}^{=:k}$
1	2..n	2..n
2	3..n	2..n-1
3	4..n	2..n-2
$\vdots$	$\vdots$	$\vdots$
$n-1$	$n..n$	2..2
$n$	$\emptyset$	$\emptyset$

(harmonische Summe)

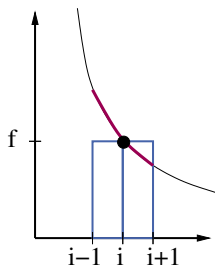


## Exkurs: Harmonische Summe

$$\int_i^{i+1} \frac{1}{x} dx \leq \frac{1}{i} \leq \int_{i-1}^i \frac{1}{x} dx$$

Also

$$\begin{aligned} \ln n &= \int_1^n \frac{1}{x} dx = \sum_{i=1}^{n-1} \int_i^{i+1} \frac{1}{x} dx \leq \sum_{i=1}^{n-1} \frac{1}{i} \leq \sum_{i=1}^n \frac{1}{i} = 1 + \sum_{i=2}^n \frac{1}{i} \\ &\leq 1 + \sum_{i=2}^n \int_{i-1}^i \frac{1}{x} dx = 1 + \int_1^n \frac{1}{x} dx = 1 + \ln n \end{aligned}$$



# Quicksort: Effiziente Implementierung

- ▶ Array-Implementierung
- ▶ „inplace“
- ▶ 2-Wegevergleiche

# Quicksort: Effiziente Implementierung

**Procedure** qSort( $a$  : **Array of** Element;  $\ell, r$  :  $\mathbb{N}$ )

**if**  $\ell \geq r$  **then return**

$k :=$  pickPivotPos( $a, \ell, r$ )

$m :=$  partition( $a, \ell, r, k$ )

qSort( $a, \ell, m - 1$ )

qSort( $a, m + 1, r$ )

# Quicksort: Effiziente Implementierung

**Function** partition( $a$  : **Array of** Element;  $\ell, r, k$  :  $\mathbb{N}$ )

$p := a[k]$

// pivot

swap( $a[k], a[r]$ )

$i := \ell$

**for**  $j := \ell$  **to**  $r-1$  **do**

**invariant**

$\ell$	$i$	$j$	$r$
$\leq p$	$> p$	?	$p$

**if**  $a[j] \leq p$  **then**

swap( $a[i], a[j]$ )

$i++$

**assert**

$\ell$	$i$	$r$
$\leq p$	$> p$	$p$

swap( $a[i], a[r]$ )

**assert**

$\ell$	$i$	$r$
$\leq p$	$p$	$> p$

**return**  $i$

# Beispiel: Partitionierung, $k = 1$

$p, \bar{i}, \underline{j}$	3	6	8	1	0	7	2	4	5	9
	<u>9</u>	6	8	1	0	7	2	4	5	3
	9	<u>6</u>	8	1	0	7	2	4	5	3
	9	6	<u>8</u>	1	0	7	2	4	5	3
	9	6	8	<u>1</u>	0	7	2	4	5	3
	1	<u>6</u>	8	9	<u>0</u>	7	2	4	5	3
	1	0	<u>8</u>	9	6	<u>7</u>	2	4	5	3
	1	0	<u>8</u>	9	6	7	<u>2</u>	4	5	3
	1	0	2	<u>9</u>	6	7	8	<u>4</u>	5	3
	1	0	2	<u>9</u>	6	7	8	4	<u>5</u>	3
	1	0	2	<u>9</u>	6	7	8	4	5	<u>3</u>
	1	0	2	<u>3</u>	6	7	8	4	5	9

# Beispiel: Rekursion

```
3 6 8 1 0 7 2 4 5 9
1 0 2 | 3 | 6 7 8 4 5 9
0 | 1 | 2 |   | 4 5 | 6 | 9 7 8
          |   | 4 | 5 |   | 8 7 | 9 |
          |   |   |   |   |   | 7 | 8 |
```

# Größerer Basisfall

**Procedure** qSort( $a$  : **Array of** Element;  $l, r$  :  $\mathbb{N}$ )

**if**  $r - l + 1 \leq n_0$  **then**

    insertionSort( $a[l..r]$ )

**else**

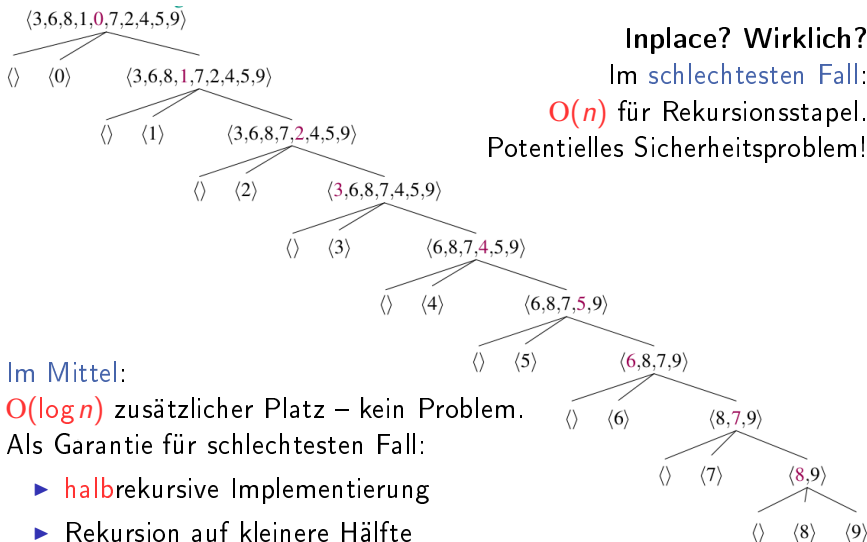
$k :=$  pickPivotPos( $a, l, r$ )

$m :=$  partition( $a, l, r, k$ )

    qSort( $a, l, m - 1$ )

    qSort( $a, m + 1, r$ )





# Halbrekursive Implementierung

**Procedure** qSort( $a$  : **Array of** Element;  $l, r$  :  $\mathbb{N}$ )

**while**  $r - l + 1 > n_0$  **do**

$k :=$  pickPivotPos( $a, l, r$ )

$m :=$  partition( $a, l, r, k$ )

**if**  $m < (l + r) / 2$  **then**

**else**

        qSort( $a, l, m - 1$ );  $l := m + 1$

        qSort( $a, m + 1, r$ );  $r := m - 1$

insertionSort( $a[l..r]$ )

# Halbrekursive Implementierung

**Procedure** qSort( $a$  : **Array of** Element;  $l, r$  :  $\mathbb{N}$ )

**while**  $r - l + 1 > n_0$  **do**

$k :=$  pickPivotPos( $a, l, r$ )

$m :=$  partition( $a, l, r, k$ )

**if**  $m < (l + r) / 2$  **then**

**else**

insertionSort( $a[l..r]$ )

qSort( $a, l, m - 1$ );  $l := m + 1$

qSort( $a, m + 1, r$ );  $r := m - 1$

**Satz:** Rekursionstiefe  $\leq \left\lceil \log \frac{n}{n_0} \right\rceil$

**Beweisidee:** Induktion. Teilproblemgröße halbiert sich (mindestens) mit jedem rekursiven Aufruf

# Quadratische Komplexität bei gleichen Elementen?

- ▶ Variante aus dem Buch verwenden
- ▶ oder doch Drei-Wege-Partitionierung

```
Procedure qSortTernary( $a$  : Array of Element;  $l, r$  :  $\mathbb{N}$ )  
  if  $l \geq r$  then return  
   $p := \text{key}(a[\text{pickPivotPos}(a, l, r)])$   
   $(m, m') := \text{partitionTernary}(a, l, r, p)$   
  qSortTernary( $a, l, m - 1$ )  
  qSortTernary( $a, m' + 1, r$ )
```

**Function** partitionTernary( $a$  : **Array of** Element;  $l, r$  :  $\mathbb{N}$ ;  $p$  : Key)

$i := l, j := l, k := r$

**invariant**

$\ell$	$i$	$j$	$k$	$r$
$< p$	$> p$	?	$= p$	

**while** ( $j \leq k$ )

**if**  $a[j] = p$       **then** swap( $a[j], a[k]$ ),  $k--$  ;

**else if**  $a[j] < p$  **then** swap( $a[j], a[i]$ ),  $i++$ ,  $j++$  ;

**else**  $j++$  ;

**assert**

$\ell$	$i$	$k$	$r$
$< p$	$> p$	$= p$	

**if**  $r - k \leq k - i + 1$  **then** swap( $a[i..i + (r - k) - 1], a[k + 1..r]$ )

**else** swap( $a[i..k], a[r - (k - i)..r]$ )

**assert**

$\ell$	$i$	$r$
$< p$	$= p$	$> p$

**return** ( $i, i + (r - k) - 1$ )

# Vergleich Quicksort $\leftrightarrow$ Mergesort

## Pro Mergesort

- ▶  $O(n \log n)$  Zeit (**deterministisch**)

qsort:  $\exists$  det. Varianten

- ▶  $n \log n + O(n)$  Elementvergleiche ( $\approx$  untere Schranke)

qsort: möglich bei sorgfältiger Pivotwahl

- ▶ **Stabil** (gleiche Elemente behalten Reihenfolge bei)

qsort: leicht bei Aufgabe der inplace-Eigenschaft

## Pro Quicksort

- ▶ **inplace**
- ▶ Etwas schneller?

# Auswahl (Selection)

**Definition:** Rang  $k$  eines Elements  $e$  einer Folge  $s =$   
Position von  $e$  in  $\text{sort}(s)$   
(angefangen bei 1).

Frage: warum ist  $k$  nicht notwendig eindeutig?

// return an element of  $s$  with rank  $k$

**Function** `select`( $s$  : Sequence of Element;  $k$  :  $\mathbb{N}$ ) : Element  
`assert`  $|s| \geq k$

Vorsicht: Es gibt **verschiedene Definitionen** von "Rang"



# Beispiel

$\langle 3, 1, 4, 5, 9, 2, 6, 5, 3, 5, 8 \rangle$

$\rightsquigarrow$  sortieren

$\langle 1, 2, 3, 3, 4, 5, 5, 5, 6, 8, 9 \rangle$

mögliche Ränge:

$\langle \overset{1}{1}, \overset{2}{2}, \overset{3,4}{3,3}, \overset{5}{4}, \overset{6,7,8}{5,5,5}, \overset{9}{6}, \overset{10}{8}, \overset{11}{9} \rangle$

# Auswahl – Anwendungen

- ▶ Spezialfall **Median**auswahl:  $k = \lceil |s|/2 \rceil$
- ▶ allgemeinere **Quantile** (10 % , ...)

z. B. Eingabe **eingrenzen** auf vielversprechendste Elemente

# Quickselect

≈ quicksort mit einseitiger Rekursion

**Function** select( $s$  : Sequence of Element;  $k$  :  $\mathbb{N}$ ) : Element

**assert**  $|s| \geq k$

pick  $p \in s$  uniformly at random // pivot key

$a := \langle e \in s : e < p \rangle$

**if**  $|a| \geq k$  **then return** select( $a, k$ ) // 

$a$
-----

$b := \langle e \in s : e = p \rangle$

**if**  $|a| + |b| \geq k$  **then return**  $p$  // 

$a$	$b = \langle p, \dots, p \rangle$
-----	-----------------------------------

$c := \langle e \in s : e > p \rangle$

**return** select( $c, k - |a| - |b|$ ) // 

$a$	$b$	$c$
-----	-----	-----

# Beispiel

$s$	$k$	$p$	$a$	$b$	$c$
$\langle 3, 1, 4, 5, 9, 2, 6, 5, 3, 5, 8 \rangle$	6	2	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle 3, 4, 5, 9, 6, 5, 3, 5, 8 \rangle$
$\langle 3, 4, 5, 9, 6, 5, 3, 5, 8 \rangle$	4	6	$\langle 3, 4, 5, 5, 3, 5 \rangle$	$\langle 6 \rangle$	$\langle 9, 8 \rangle$
$\langle 3, 4, 5, 5, 3, 5 \rangle$	4	5	$\langle 3, 4, 3 \rangle$	$\langle 5, 5, 5 \rangle$	$\langle \rangle$

# Quickselect – Analyse

**Function** select( $s$  : Sequence of Element;  $k$  :  $\mathbb{N}$ ) : Element

**assert**  $|s| \geq k$

pick  $p \in s$  uniformly at random // pivot key

$a := \langle e \in s : e < p \rangle$

**if**  $|a| \geq k$  **then return** select( $a, k$ ) // 

$a$
-----

$b := \langle e \in s : e = p \rangle$

**if**  $|a| + |b| \geq k$  **then return**  $p$  // 

$a$	$b = \langle p, \dots, p \rangle$
-----	-----------------------------------

$c := \langle e \in s : e > p \rangle$

**return** select( $c, k - |a| - |b|$ ) // 

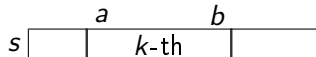
$a$	$b$	$c$
-----	-----	-----

**Satz:** quickselect hat erwartete Ausführungszeit  $O(|s|)$

**Beweis:** hier nicht

# Mehr zum Auswahlproblem

- ▶ Tuning (array, inplace, 2-Wege-Vergleiche, iterativ)  
analog quicksort
- ▶ **Deterministische** Auswahl: quickselect mit spezieller det. Pivotwahl
- ▶  $k$  Elemente mit Rang  $\leq k$   
ggf. sortiert. wer weiss wie es geht?
- ▶ Weitere **Verallgemeinerungen**:  
mehrere Ränge, teilweise sortierte Eingaben, ...  
Beispiel: Optimale **Range Median** Berechnung  
[B. Gfeller, P. Sanders, ICALP 2009].  
Vorberechnungszeit  $O(n \log n)$ , Zeit  $O(\log n)$  für  
 $\text{select}(\langle s[a], \dots, s[b] \rangle, k)$



# Durchbrechen der unteren Schranke – Ganzzahliges Sortieren

Untere Schranke = schlechte Nachricht?

Nein: u.U. **Hinweis, welche Annahmen man in Frage stellen muss.**

Beim Sortieren:

Mehr mit den Schlüsseln machen als nur Vergleichen.

# Schlüssel $0..K - 1$ – Eimer-Sortieren (Bucketsort)

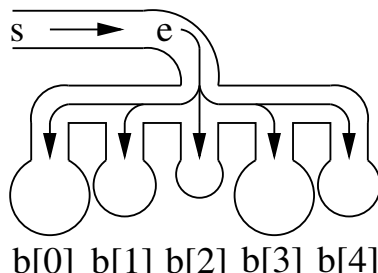
**Procedure**  $K\text{Sort}(s : \text{Sequence of Element})$

$b = \langle \langle \rangle, \dots, \langle \rangle \rangle : \text{Array } [0..K - 1] \text{ of Sequence of Element}$

**foreach**  $e \in s$  **do**  $b[\text{key}(e)].\text{pushBack}(e)$

$s := \text{concatenation of } b[0], \dots, b[K - 1]$

Zeit:  $O(n + K)$





Beispiel:  $K = 4$

**Procedure**  $K\text{Sort}(s : \text{Sequence of Element})$

$b = \langle \langle \rangle, \dots, \langle \rangle \rangle : \mathbf{Array} [0..K - 1] \mathbf{of Sequence of Element}$

**foreach**  $e \in s$  **do**  $b[\text{key}(e)].\text{pushBack}(e)$

$s := \text{concatenation of } b[0], \dots, b[K - 1]$

$s = \langle (3, a), (1, b), (2, c), (3, d), (0, e), (0, f), (3, g), (2, h), (1, i) \rangle$

verteilen  $\rightsquigarrow$

$b = \langle \langle (0, e), (0, f) \rangle \mid \langle (1, b), (1, i) \rangle \mid \langle (2, c), (2, h) \rangle \mid \langle (3, a), (3, d), (3, g) \rangle \rangle$

aneinanderhängen  $\rightsquigarrow$

$s = \langle (0, e), (0, f), (1, b), (1, i), (2, c), (2, h), (3, a), (3, d), (3, g) \rangle$ .

# Array-Implementierung

**Procedure** KSortArray( $a, b$  : **Array** [1.. $n$ ] of Element)

$c = \langle 0, \dots, 0 \rangle$  : **Array** [0.. $K-1$ ] of  $\mathbb{N}$

**for**  $i := 1$  **to**  $n$  **do**  $c[\text{key}(a[i])]++$

$C := 1$

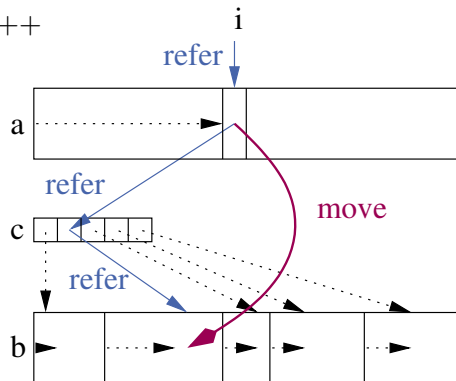
**for**  $k := 0$  **to**  $K-1$  **do**

$$\begin{pmatrix} C \\ c[k] \end{pmatrix} := \begin{pmatrix} C + c[k] \\ C \end{pmatrix}$$

**for**  $i := 1$  **to  $n$  **do****

$b[c[\text{key}(a[i])]] := a[i]$

$c[\text{key}(a[i])]++$



Beispiel:  $a = [3, 1, 2, 3, 0, 0, 3, 2, 1]$ ,  $K = 4$

**Procedure** KSortArray( $a, b : \text{Array } [1..n]$  of Element)

$c = \langle 0, \dots, 0 \rangle : \text{Array } [0..K-1]$  of  $\mathbb{N}$

**for**  $i := 1$  **to**  $n$  **do**  $c[\text{key}(a[i])]++$  //  $c := [2, 2, 2, 3]$

$C := 1$

**for**  $k := 0$  **to**  $K-1$  **do**

$\binom{C}{c[k]} := \binom{C+c[k]}{C}$  //  $c := [1, 3, 5, 7]$

**for**  $i := 1$  **to**  $n$  **do**

$b[c[\text{key}(a[i])]] := a[i]$  //  $b := [0, 0, 1, 1, 2, 2, 3, 3, 3]$

$c[\text{key}(a[i])]++$  // bei  $i = [5, 6, 2, 9, 3, 8, 1, 4, 7]$

# $K^d$ Schlüssel –

## Least-Significant-Digit Radix-Sortieren

Beobachtung: KSort ist **stabil**, d. h.,  
Elemente mit gleichem Schlüssel behalten ihre relative Reihenfolge.

**Procedure** LSDRadixSort( $s$  : Sequence of Element)

**for**  $i := 0$  **to**  $d - 1$  **do**

    redefine key( $x$ ) as  $(x \text{ div } K^i) \bmod K$  //  $x$ 

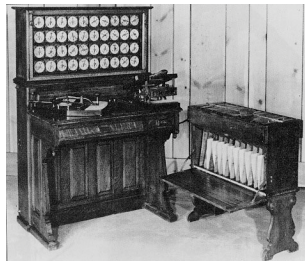
$d-1$	...	$i$	...	1	0
-------	-----	-----	-----	---	---

    KSort( $s$ )

**invariant**

$s$  is sorted with respect to digits  $i..0$

Zeit:  $O(d(n + K))$



# LSD-Radix-Sort Beispiel

11 73 23 45 76 34 78 23 65 13 67 05 14 75

Ziffer 0:

11 73 23 23 13 34 14 45 65 05 75 76 67 78

Ziffer 1:

05 11 13 14 23 23 34 45 65 67 73 75 76 78

# Mehr zu ganzzahligem Sortieren

- ▶ Nicht (ohne weiteres) inplace
- ▶ MSD-Radix-Sort: Wichtigste Ziffer zuerst.  
im Mittel Cache-effizienter aber Probleme mit schlechtestem Fall
- ▶ Kleineres  $K$  kann besser sein. (Cache-Misses, TLB-Misses)

## Mehr Theorie:

Zeit  $O\left(n\sqrt{\log\log n}\right)$  (erwartet) für ganzzahlige Schlüssel, die in ein Maschinenwort passen. [[Han Thorup 2002](#)]

# Sortieren: vergleichsbasiert $\leftrightarrow$ ganzzahlig

pro ganzzahlig:

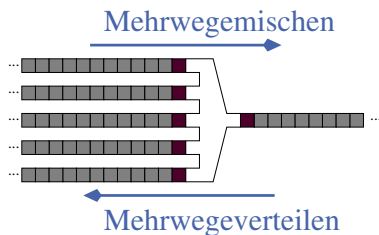
- ▶ **asymptotisch** schneller

pro vergleichsbasiert

- ▶ weniger Annahmen  
(z. B. wichtig für **Algorithmenbibliotheken**)
- ▶ robust gegen beliebige **Eingabeverteilungen**
- ▶ **Cache**-Effizienz weniger schwierig
- ▶ bei **langen Schlüsseln** oft schneller

# Mehr zu Sortieren

- ▶ Verfügbar in **Algorithmenbibliotheken**
- ▶ (binary) mergesort  $\rightsquigarrow$   
**Mehrwegemischen**
- ▶ quicksort  $\rightsquigarrow$   
Sortieren durch **Mehrwegeverteilen**
- $\rightsquigarrow$  **Parallel**
- $\rightsquigarrow$  **Extern**: oft noch wichtiger als intern



- ▶ **Prioritätslisten** (kommen als nächstes)
- ▶ **Dynamische sortierte Listen** (als übernächstes)



# Was haben wir jenseits von Sortieren gelernt?

- ▶ **Auswahl**problem (Quickselect)
- ▶ Konzept: **Inplace**-Algorithmen (Quicksort)
- ▶ Analysetechniken: **randomisierte** Algorithmen (Quicksort)
- ▶ **Untere Schranken** (und wie man sie durchbricht)  
(vergleichsbasiertes Sortieren, Radix-Sortieren)
- ▶ Mehr **Teile-und-Herrsche** Algorithmen (Mergesort, Quicksort, Quickselect)
- ▶ Mehr Beispiele für **Invarianten** (Schleifen, Datenstrukturen)
- ▶ Bezug Summen  $\leftrightarrow$  **Integrale**
- ▶ Harmonische Summe  $H_n$
- ▶ Algorithm Engineering: **Effizienz und Robustheit** der Implementierung (Quicksort)

# Prioritätslisten



# Prioritätslisten (priority queues)

Verwalte Menge  $M$  von Elementen mit Schlüsseln

$\text{Insert}(e)$ :  $M := M \cup e$

$\text{DeleteMin}$ : return and remove  $\min M$

# Prioritätslisten – Anwendungen

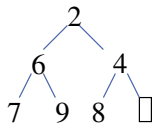
(ohne zusätzliche Operationen)

- ▶ Mehrwegemischen (klein)
- ▶ Greedy Algorithmen (z. B., Scheduling) (klein–mittel)
- ▶ Simulation diskreter Ereignisse (mittel–groß)
- ▶ Branch-and-Bound-Suche (groß)
- ▶ run formation für externes Sortieren (groß)
- ▶ Time forward processing (riesig)

# Binäre Heaps

**Heap-Eigenschaft:** Bäume (oder Wälder) mit  $\forall v : \text{parent}(v) \leq v$

**Binärer Heap:** Binärbaum, Höhe  $\lfloor \log n \rfloor$ , fehlende Blätter rechts unten.



Beobachtung: **Minimum** steht an der **Wurzel**

Idee: Änderungen nur entlang eines **Pfades** Wurzel–Blatt

↔

insert, deleteMin brauchen **Zeit  $O(\log n)$**

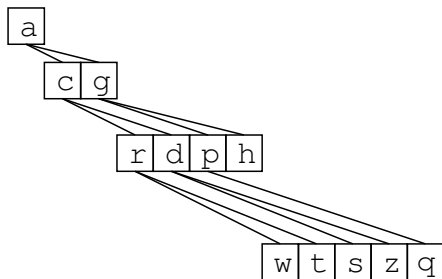
# Implizite Baum-Repräsentation

- ▶ Array  $h[1..n]$
- ▶ Schicht für Schicht
- ▶  $\text{parent}(j) = \lfloor j/2 \rfloor$
- ▶ linkes Kind( $j$ ):  $2j$
- ▶ rechtes Kind( $j$ ):  $2j+1$

h: 

a	c	g	r	d	p	h	w	t	s	z	q
---	---	---	---	---	---	---	---	---	---	---	---

j: 1 2 3 4 5 6 7 8 9 10 11 12 13

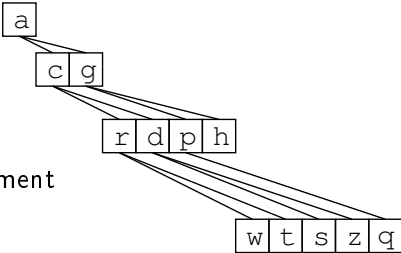


Nicht nur nützlich für heaps:  
z. B. Turnierbäume, statische Suchbäume

# Pseudocode

(beschränkte PQ)

h: [a | c | g | r | d | p | h | w | t | s | z | q] |  
j: 1 2 3 4 5 6 7 8 9 10 11 12 13



```
Class BinaryHeapPQ(w : ℕ) of Element  
  h : Array [1..w] of Element  
  n=0 : ℕ  
  invariant  $\forall j \in 2..n : h[\lfloor j/2 \rfloor] \leq h[j]$   
  Function min assert n > 0 ; return h[1]
```

# Einfügen

**Procedure** insert( $e : \text{Element}$ )

**assert**  $n < w$

$n++ ; h[n] := e$

siftUp( $n$ )

**Procedure** siftUp( $i : \mathbb{N}$ )

**assert** the heap property holds

except maybe at position  $i$

**if**  $i = 1 \vee h[\lfloor i/2 \rfloor] \leq h[i]$  **then return**

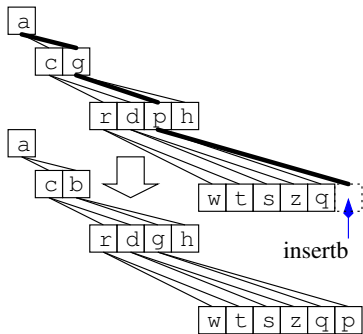
swap( $h[i], h[\lfloor i/2 \rfloor]$ )

siftUp( $\lfloor i/2 \rfloor$ )

h: 

a	c	g	r	d	p	h	w	t	s	z	q	
---	---	---	---	---	---	---	---	---	---	---	---	--

  
j: 1 2 3 4 5 6 7 8 9 10 11 12 13





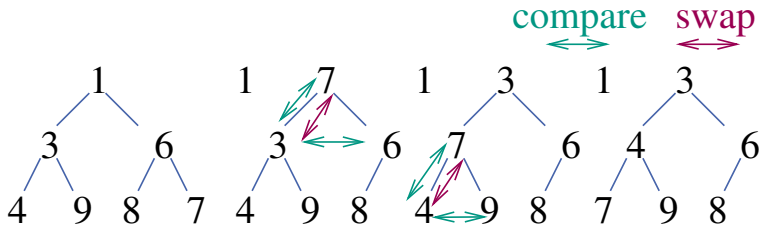
**Function** deleteMin : Element

result =  $h[1]$  : Element

$h[1] := h[n]$ ;  $n \leftarrow$

siftDown(1)

**return** result



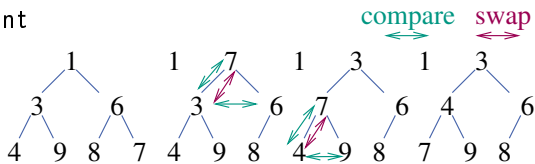
**Function** deleteMin : Element

result = h[1] : Element

h[1] := h[n]; n--

siftDown(1)

return result



**Procedure** siftDown( $i : \mathbb{N}$ )

assert heap property except, possibly at  $j = 2i$  and  $j = 2i + 1$

if  $2i \leq n$  then

//  $i$  is not a leaf

if  $2i + 1 > n \vee h[2i] \leq h[2i + 1]$  then  $m := 2i$  else  $m := 2i + 1$

assert  $\nexists$  sibling( $m$ )  $\vee h[\text{sibling}(m)] \geq h[m]$

if  $h[i] > h[m]$  then

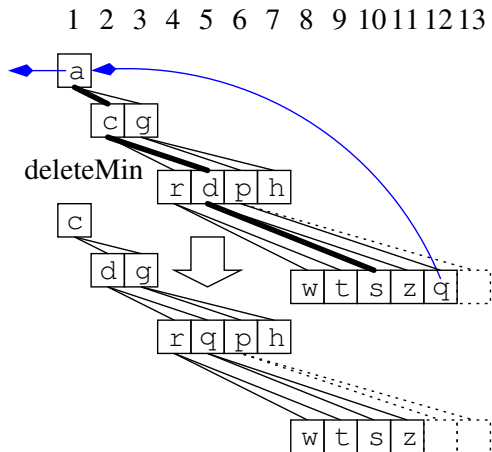
// heap property violated

swap( $h[i], h[m]$ )

siftDown( $m$ )

assert the heap property holds for the subtree rooted at  $i$

# deleteMin: Beispiel



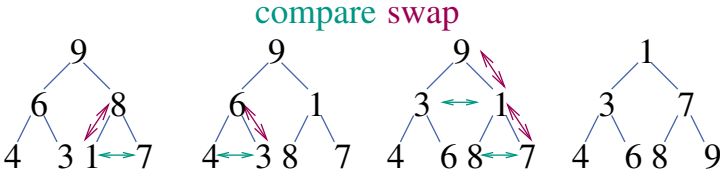
# Binärer Heap – Analyse

- ▶ **Satz:** min dauert  $O(1)$ .
- ▶ **Lemma:** Höhe ist  $\lfloor \log n \rfloor$
- ▶ **Satz:** insert dauert  $O(\log n)$ .
- ▶ **Satz:** deleteMin dauert  $O(\log n)$ .
- ▶ **Beweis:** Zeit  $O(1)$  pro Schicht.

# Binärer Heap – Konstruktion

```
Procedure buildHeapBackwards  
  for  $i := \lfloor n/2 \rfloor$  downto 1 do siftDown( $i$ )
```

# Beispiel: Binärer Heap – Konstruktion



# Binärer Heap – Konstruktion

- ▶ **Satz:** buildHeap läuft in Zeit  $O(n)$
- ▶ **Beweis:** Sei  $k = \lfloor \log n \rfloor$ .  
In Tiefe  $\ell \in 0.. \lfloor \log n \rfloor$ :
  - ▶  $2^\ell$  Aufrufe von siftDown
  - ▶ Kosten je  $O(k - \ell)$ . Insgesamt:

$$\begin{aligned} O\left(\sum_{0 \leq \ell < k} 2^\ell (k - \ell)\right) &= O\left(2^k \sum_{0 \leq \ell < k} \frac{k - \ell}{2^{k - \ell}}\right) = O\left(2^k \underbrace{\sum_{j \geq 1} \frac{j}{2^j}}_{O(1)}\right) \\ &= O(2^k) = O(n) \end{aligned}$$

## Ein nützlicher Rechentrick

$$\begin{aligned}\sum_{j \geq 1} j \cdot 2^{-j} &= \sum_{j \geq 1} 2^{-j} + \sum_{j \geq 2} 2^{-j} + \sum_{j \geq 3} 2^{-j} + \dots \\ &= (1 + 1/2 + 1/4 + 1/8 + \dots) \cdot \sum_{j \geq 1} 2^{-j} \\ &= 2 \cdot 1 = 2\end{aligned}$$

$$\begin{array}{rcccccc} 1/2 & + & 1/4 & + & 1/8 & + & 1/16 & + & \dots & = & 1 \\ & & 1/4 & + & 1/8 & + & 1/16 & + & \dots & = & 1/2 \\ & & & & 1/8 & + & 1/16 & + & \dots & = & 1/4 \\ & & & & & & 1/16 & + & \dots & = & 1/8 \\ & & & & & & & & \dots & & = \dots \end{array}$$

---

$$1 \cdot 1/2 + 2 \cdot 1/4 + 3 \cdot 1/8 + 4 \cdot 1/16 + \dots = 2$$



# Heapsort

```
Procedure heapSortDecreasing( $a[1..n]$ )  
  buildHeap( $a$ )  
  for  $i := n$  downto 2 do  
     $h[i] :=$  deleteMin
```

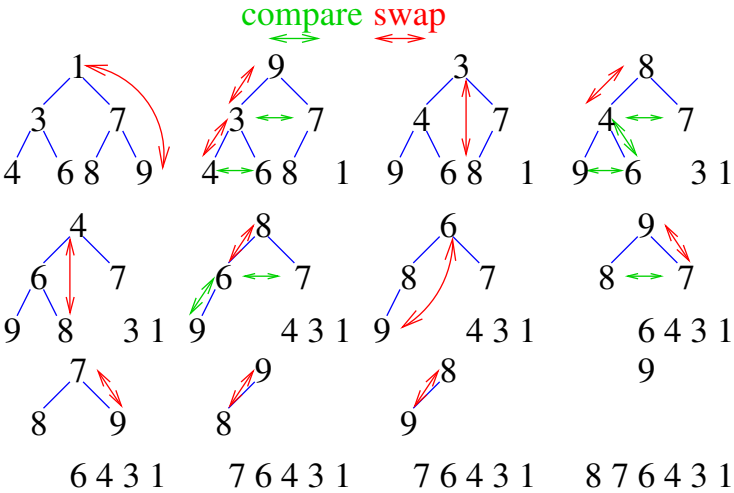
Laufzeit:  $O(n \log n)$

Andere Sichtweise: effiziente Implementierung von

Sortieren durch Auswahl

Frage: Wie sortiert man aufsteigend?

# Heapsort: Beispiel



# Heapsort $\leftrightarrow$ Quicksort $\leftrightarrow$ Mergesort

	Heapsort	Quicksort	Mergesort
Vergleiche	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
E[Vergleiche]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
zusätzl. Platz	$O(1)$	$O(\log n)$	$O(n)$
Cachezugriffe ( $B$ = Blockgröße)	$O(n \log n)$	$O(\frac{n}{B} \log n)$	$O(\frac{n}{B} \log n)$

Kompromiss: z. B.

introspektives Quicksort der C++ Standardbibliothek:

Quicksort starten. Zu wenig Fortschritt? Umschalten auf Heapsort.

# Adressierbare Prioritätslisten

**Procedure** build( $\{e_1, \dots, e_n\}$ )  $M := \{e_1, \dots, e_n\}$

**Function** size **return**  $|M|$

**Procedure** insert( $e$ )  $M := M \cup \{e\}$

**Function** min **return**  $\min M$

**Function** deleteMin  $e := \min M$ ;  $M := M \setminus \{e\}$ ; **return**  $e$

**Function** remove( $h : \text{Handle}$ )  $e := h$ ;  $M := M \setminus \{e\}$ ; **return**  $e$

**Procedure** decreaseKey( $h : \text{Handle}, k : \text{Key}$ )

**assert**  $\text{key}(h) \geq k$ ;  $\text{key}(h) := k$

**Procedure** merge( $M'$ )  $M := M \cup M'$

# Adressierbare Prioritätslisten: Anwendungen

## Greedy-Algorithmus:

**while** solution not complete **do**

add the **best** available “piece” to the solution

**update** piece priorities // e.g., using addressable priority queue

Beispiele:

- ▶ Dijkstras Algorithmus für **kürzeste Wege**
- ▶ Jarník-Prim Algorithmus für **minimale Spannbäume**
- ▶ **Scheduling**: Jobs → am wenigsten belastete Maschine
- ▶ Hierarchiekonstruktion für **Routenplanung**
- ▶ Suche nach erfüllenden Belegungen **aussagenlog.** Formeln?

# Adressierbare Binäre Heaps

**Problem:** Elemente bewegen sich.

Dadurch werden Elementverweise ungültig.

(Ein) **Ausweg:** Unbewegliche **Vermittler-Objekte**.

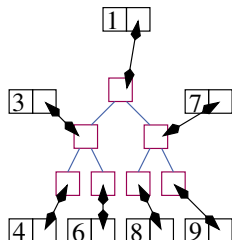
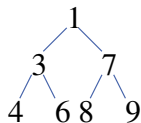
**Invariante:** **proxy(e)** verweist auf Position von e.

↪ **Vermittler** bei jeder Vertauschung **aktualisieren**.

↪ **Rückverweis** Element → **Vermittler**

**Laufzeit:**

$O(\log n)$  für alle Operationen ausser merge und buildHeap, die  $O(n)$  brauchen.



# Adressierbare Prioritätslisten – Laufzeiten

Operation	Binary Heap	Fibonacci-Heap (Buch)
build	$O(n)$	$O(n)$
size	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$
remove	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(1)$ am.
merge	$O(n)$	$O(1)$

# Prioritätslisten: Mehr

- ▶ **ganzzahlige** Schlüssel (stay tuned)
- ▶ extern: Geht gut (nichtadressierbar)
- ▶ parallel: Semantik?



# Prioritätslisten: Zusammenfassung

- ▶ Häufig benötigte Datenstruktur
- ▶ Adressierbarkeit ist nicht selbstverständlich
- ▶ **Binäre Heaps** sind einfache, relativ effiziente Implementierung

# Was haben wir jenseits von Prioritätslisten gelernt?

- ▶ **implizites** Layout von Binärbäumen
- ▶  $\sum_j j2^j$
- ▶ **Heapsort** (inplace!)

# Sortierte Folgen

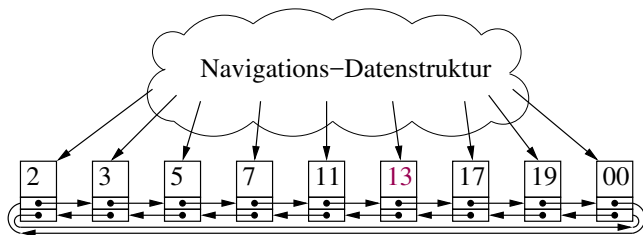


## Sortierte Folgen:

$\langle e_1, \dots, e_n \rangle$  mit  $e_1 \leq \dots \leq e_n$

„kennzeichnende“ Funktion:

$M.\text{locate}(k) := \text{address of } \min \{ e \in M : e \geq k \}$



**Annahme:** Dummy-Element mit Schlüssel  $\infty$

**Achtung:** In Abbildungen sieht  $\infty$  wie 00 aus

## Statisch: Sortiertes Feld mit binärer Suche

```
// Find  $\min \{i \in 1..n+1 : a[i] \geq k\}$ 
```

```
Function locate( $a[1..n]$ ,  $k$  : Element)
```

```
  ( $l, r$ ) := (0,  $n+1$ )           // Assume  $a[0] = -\infty$ ,  $a[n+1] = \infty$ 
```

```
  while  $l+1 < r$  do
```

```
    invariant  $0 \leq l < r \leq n+1$  and  $a[l] < k \leq a[r]$ 
```

```
     $m := \lfloor (r+l)/2 \rfloor$            //  $l < m < r$ 
```

```
    if  $k \leq a[m]$  then  $r := m$  else  $l := m$ 
```

```
  return  $r$ 
```

**Übung:** Müssen die Sentinels  $\infty$  /  $-\infty$  tatsächlich vorhanden sein?

**Übung:** Variante von binärer Suche:

bestimme  $l, r$  so dass  $a[l..r-1] = [k, \dots, k]$ ,  $a[l-1] < k$  und  $a[r] > k$

## Statisch: Sortiertes Feld mit binärer Suche

```
// Find  $\min \{i \in 1..n+1 : a[i] \geq k\}$   
Function locate( $a[1..n], k$  : Element)  
  ( $\ell, r$ ) := (0,  $n+1$ )           // Assume  $a[0] = -\infty, a[n+1] = \infty$   
  while  $\ell + 1 < r$  do  
    invariant  $0 \leq \ell < r \leq n+1$  and  $a[\ell] < k \leq a[r]$   
     $m := \lfloor (r + \ell) / 2 \rfloor$            //  $\ell < m < r$   
    if  $k \leq a[m]$  then  $r := m$  else  $\ell := m$   
  return  $r$ 
```

Zeit:  $O(\log n)$

Beweisidee:  $r - \ell$  „halbiert“ sich in jedem Schritt

## Binäre Suche – Beispiel: $k = 15$

```
Function locate( $a[1..n], k$  : Element) //  $\min \{i \in 1..n+1 : a[i] \geq k\}$   
  ( $\ell, r$ ):= (0,  $n+1$ ) // Assume  $a[0] = -\infty, a[n+1] = \infty$   
  while  $\ell + 1 < r$  do  
    invariant  $0 \leq \ell < r \leq n+1$  and  $a[\ell] < k \leq a[r]$   
     $m$ :=  $\lfloor (r + \ell) / 2 \rfloor$  //  $\ell < m < r$   
    if  $k \leq a[m]$  then  $r$ :=  $m$  else  $\ell$ :=  $m$   
  return  $r$ 
```

Indizes:	[0,	1,	2,	3,	4,	5,	6,	7,	8,	9]
Einträge:	$[-\infty,$	2,	3,	5,	7,	11,	13,	17,	19,	$\infty$ ]
	$[-\infty,$	2,	3,	5,	7,	11,	13,	17,	19,	$\infty$ ]
	$[-\infty,$	2,	3,	5,	7,	11,	13,	17,	19,	$\infty$ ]
	$[-\infty,$	2,	3,	5,	7,	11,	13,	17,	19,	$\infty$ ]

# Dynamische Sortierte Folgen – Grundoperationen

insert, remove, update, locate

$(M.\text{locate}(k) := \min \{e \in M : e \geq k\})$

$O(\log n)$



# Mehr Operationen

$\langle \min, \dots, a, \dots, b, \dots, \max \rangle$

**min**: Erstes Listenelement

Zeit  $O(1)$

**max**: Letztes Listenelement

Zeit  $O(1)$

**rangeSearch**( $a, b$ )

//  $O(\log n + |\text{result}|)$

result :=  $\langle \rangle$

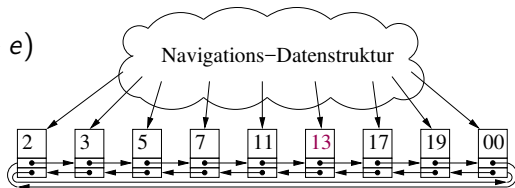
$h := \text{locate}(a)$

**while**  $h \rightarrow e \leq b$  **do**

    result.pushBack( $h \rightarrow e$ )

$h := h \rightarrow \text{next}$

**return** result



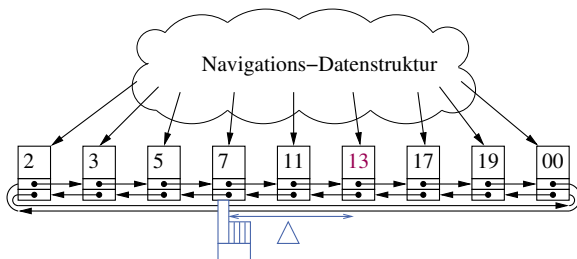
# Noch mehr Operationen

- ▶ **(re)build**: Navigationsstruktur für **sortierte** Liste aufbauen  $O(n)$
- ▶  $\langle w, \dots, x \rangle.\text{concat}(\langle y, \dots, z \rangle) = \langle w, \dots, x, y, \dots, z \rangle$   $O(\log n)$
- ▶  $\langle w, \dots, x, y, \dots, z \rangle.\text{split}(y) = (\langle w, \dots, x \rangle, \langle y, \dots, z \rangle)$   $O(\log n)$

**Zählen**: rank, select, rangeSize  $O(\log n)$

**Fingersuche**:  $\Delta =$  Abstand zu **Finger**info

zusätzlicher Parameter für insert, remove, locate, ...  $O(\log n) \rightarrow \log \Delta$



# Abgrenzung

Hash-Tabelle: nur insert, remove, find. Kein locate, rangeQuery

Sortiertes Feld: nur bulk-Updates. Aber:

Hybrid-Datenstruktur oder  $\log \frac{n}{M}$  geometrisch wachsende  
statische Datenstrukturen

Prioritätsliste: nur insert, deleteMin, (decreaseKey, remove). Dafür:  
schnelles merge

Sortierte Folgen allgemein: die eierlegende Wollmilchdatenstruktur.  
„Etwas“ langsamer als speziellere Datenstrukturen

# Sortierte Folgen – Anwendungen

- ▶ Best-First Heuristiken
- ▶ Algorithmische Geometrie: Sweepline-Datenstrukturen
- ▶ Datenbankindex
- ▶ ...

# Anwendungsbeispiel: Best Fit Bin Packing

**Procedure** binPacking( $s$ )

$B$  : SortedSequence // used bins sorted by free capacity

**foreach**  $e \in s$  by decreasing element size // sort

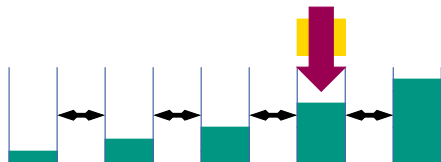
**if**  $\neg \exists b \in B : \text{free}(b) \geq e$  **then**  $B.\text{insert}(\text{new bin})$

locate  $b \in B$  with smallest  $\text{free}(b) \geq e$

insert  $e$  into bin  $b$

Zeit:  $O(|s| \log |s|)$

Qualität: „gut“. Details: nicht hier



# Binäre Suchbäume

## Blätter: Elemente

einer sortierten Folge.

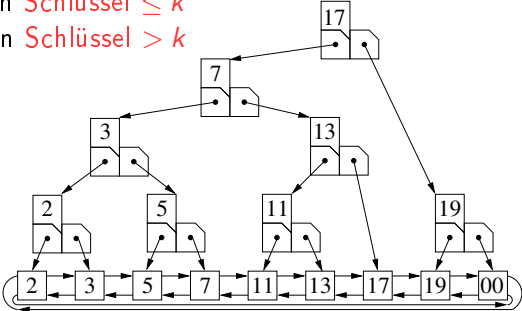
**Innere Knoten**  $v = (k, \ell, r)$ ,

(Spalt-Schlüssel, linker Teilbaum, rechter Teilbaum).

**Invariante:**

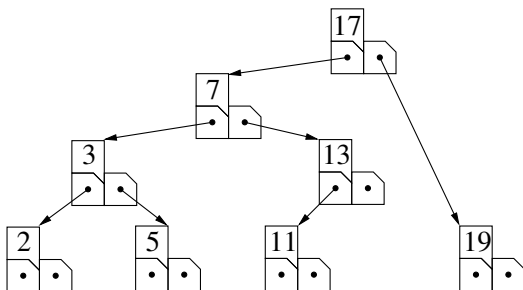
über  $\ell$  erreichbare Blätter haben **Schlüssel**  $\leq k$

über  $r$  erreichbare Blätter haben **Schlüssel**  $> k$



# Varianten, Bemerkungen

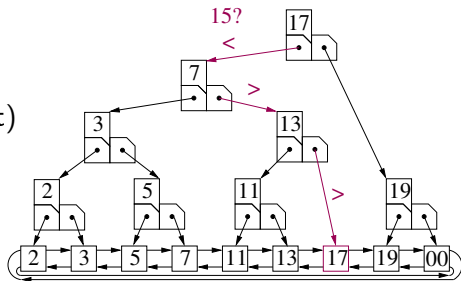
- ▶ **Dummy**-Element im Prinzip verzichtbar
- ▶ Oft speichern auch **innere Knoten Elemente**
- ▶ „**Suchbaum**“ wird oft als Synonym für „**sortierte Folge**“ verwendet.  
(Aber das vermischt (eine) **Implementierung** mit der **Schnittstelle**)



# locate( $k$ )

Idee: Benutze Spaltschlüssel  $x$  als Wegweiser.

```
Function locate( $k, x$ )  
  if  $x$  is a leaf then  
    if  $k \leq x$  then return  $x$   
    else return  $x \rightarrow \text{next}$   
  if  $k \leq x$  then  
    return locate( $k, x \rightarrow \text{left}$ )  
  else  
    return locate( $k, x \rightarrow \text{right}$ )
```

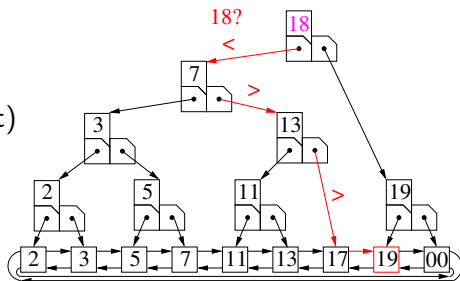




# locate( $k$ ) – anderes Beispiel

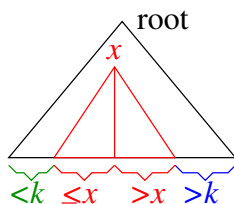
Idee: Benutze Spaltschlüssel  $x$  als Wegweiser.

```
Function locate( $k, x$ )  
  if  $x$  is a leaf then  
    if  $k \leq x$  then return  $x$   
    else return  $x \rightarrow \text{next}$   
  if  $k \leq x$  then  
    return locate( $k, x \rightarrow \text{left}$ )  
  else  
    return locate( $k, x \rightarrow \text{right}$ )
```



# Invariante von locate( $k$ )

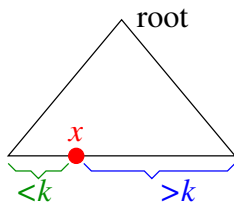
```
Function locate( $k, x$ )  
  if  $x$  is a leaf then  
    if  $k \leq x$  then return  $x$   
    else return  $x \rightarrow \text{next}$   
  if  $k \leq x$  then  
    return locate( $k, x \rightarrow \text{left}$ )  
  else  
    return locate( $k, x \rightarrow \text{right}$ )
```



**Invariante:** Sei  $X$  die Menge aller von  $x$  erreichbaren Listenelemente.  
Listenelemente **links** von  $X$  sind  $< k$   
Listenelemente **rechts** von  $X$  sind  $> k$

# Ergebnisberechnung von locate( $k$ )

```
Function locate( $k, x$ )  
  if  $x$  is a leaf then  
    if  $k \leq x$  then return  $x$   
    else return  $x \rightarrow \text{next}$   
  if  $k \leq x$  then  
    return locate( $k, x \rightarrow \text{left}$ )  
  else  
    return locate( $k, x \rightarrow \text{right}$ )
```



**Fall**  $k = x$ : return  $x$

**Fall**  $k < x$ : return  $x$

**Fall**  $k > x$ : return  $x \rightarrow \text{next}$

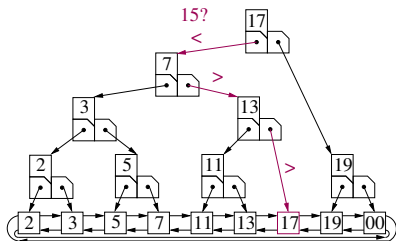
Bingo!

links ist es auch nicht

nächstes ist  $> k$  und  $k$  gibt es nicht

# Laufzeit von locate( $k$ )

```
Function locate( $k, x$ )  
  if  $x$  is a leaf then  
    if  $k \leq x$  then return  $x$   
    else return  $x \rightarrow \text{next}$   
  if  $k \leq x$  then  
    return locate( $k, x \rightarrow \text{left}$ )  
  else  
    return locate( $k, x \rightarrow \text{right}$ )
```



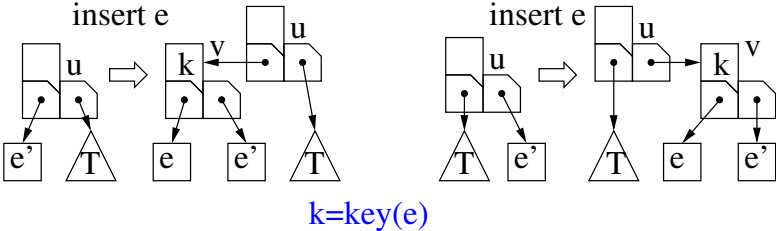
Laufzeit:  $O(\text{Höhe})$ .

Bester Fall: **perfekt balanciert, d. h.** Tiefe =  $\lfloor \log n \rfloor$

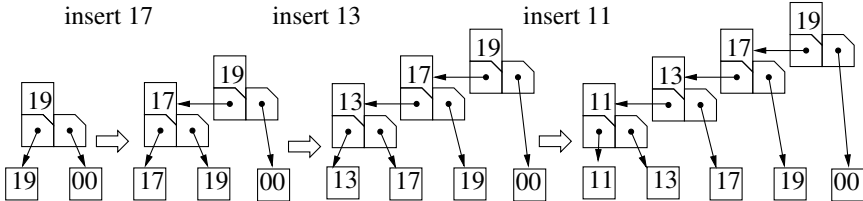
Schlechtester Fall: Höhe  $n$

# Naives Einfügen

Zunächst wie `locate(e)`. Sei  $e'$  gefundenes Element,  $u$  der Elterknoten



# Beispiel



Problem: Der Baum wird beliebig unbalanciert.

↪ langsam

# Suchbäume balancieren

Perfekte Balance: schwer aufrechtzuerhalten

Flexible Höhe  $O(\log n)$ : balancierte **binäre** Suchbäume.

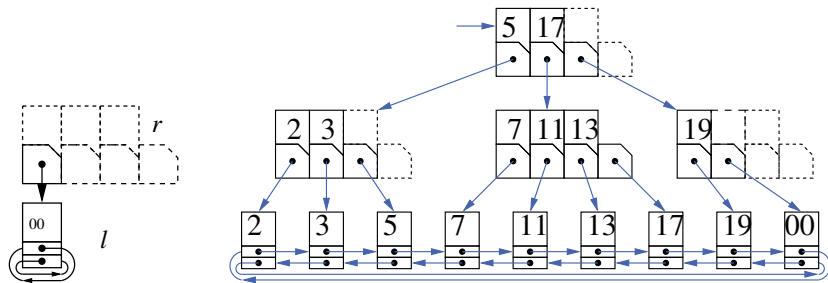
Nicht hier (Variantenzoo).

Flexibler Knotengrad:  **$(a, b)$ -Bäume**.

$\approx$  Grad zwischen  $a$  und  $b$ .

Höhe  $\approx \log_a n$

# $(a, b)$ -Bäume



**Blätter:** Listenelemente (wie gehabt). Alle mit **gleicher Tiefe!**

**Innere Knoten:** Grad  $a..b$

**Wurzel:** Grad  $2..b$ , (Grad 1 für  $\langle \rangle$ )



# Items

**Class** ABHandle : **Pointer** to ABItem or Item

**Class** ABItem (splitters : Sequence of Key, children : Sequence of ABHandle)

$d = |\text{children}| : 1..b$  // outdegree

$s = \text{splitters} : \text{Array } [1..b-1] \text{ of Key}$

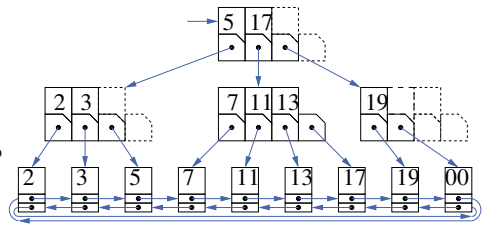
$c = \text{children} : \text{Array } [1..b] \text{ of Handle}$

## Invariante:

$e$  über  $c[i]$  erreichbar

$\Rightarrow s[i-1] < \text{key}(e) \leq s[i]$  mit

$s[0] = -\infty, s[d] = s[d+1] = \infty$



# Initialisierung

**Class** ABTree( $a \geq 2 : \mathbb{N}$ ,  $b \geq 2a - 1 : \mathbb{N}$ ) **of** Element

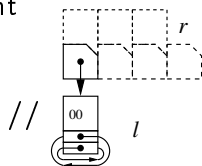
$\ell = \langle \rangle$  : List **of** Element

$r$  : ABItem( $\langle \rangle$ ,  $\langle \ell.\text{head} \rangle$ )

height=1 :  $\mathbb{N}$

// Locate the smallest Item with key  $k' \geq k$

**Function** locate( $k$  : Key) : Handle **return**  $r.\text{locateRec}(k, \text{height})$



# Locate

**Function** ABItem::locateLocally( $k$  : Key) :  $\mathbb{N}$   
    **return**  $\min \{i \in 1..d : k \leq s[i]\}$

**Function** ABItem::locateRec( $k$  : Key,  $h$  :  $\mathbb{N}$ ) : Handle

$i := \text{locateLocally}(k)$

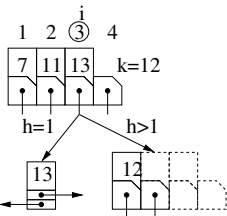
**if**  $h = 1$  **then**

**if**  $c[i] \rightarrow e \geq k$  **Then** **return**  $c[i]$

**else** **return**  $c[i] \rightarrow \text{next}$

**else**

**return**  $c[i] \rightarrow \text{locateRec}(k, h - 1)$  //



**Invariante:** im Wesentlichen analog zu binären Suchbäumen

# Locate – Laufzeit

$O(b \cdot \text{height})$

**Lemma:**  $\text{height} = h \leq 1 + \left\lceil \log_a \frac{n+1}{2} \right\rceil$

**Beweis:**

**Fall**  $n = 1$ :  $\text{height} = 1$ .

**Fall**  $n > 1$ :

Wurzel hat Grad  $\geq 2$  und

Innere Knoten haben Grad  $\geq a$ .

$\Rightarrow \geq 2a^{h-1}$  Blätter.

Es gibt  $n+1$  Blätter.

Also  $n+1 \geq 2a^{h-1}$

$\Rightarrow h \leq 1 + \log_a \frac{n+1}{2}$

Rundung folgt, weil  $h$  eine ganze Zahl ist. □

Übung:  $b \rightarrow \log b?$

# Einfügen – Algorithmenskizze

## Procedure insert( $e$ )

Finde Pfad Wurzel  $\rightsquigarrow$  nächstes Element  $e'$

$\ell$ .insertBefore( $e, e'$ )

füge key( $e$ ) als neuen Splitter in Vorgänger  $u$

**if**  $u.d = b+1$  **then**

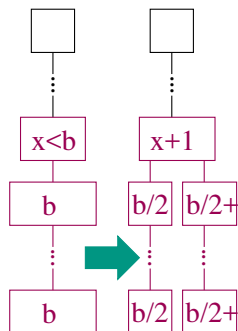
**spalte**  $u$  in 2 Knoten mit Graden

$\lfloor (b+1)/2 \rfloor, \lceil (b+1)/2 \rceil$

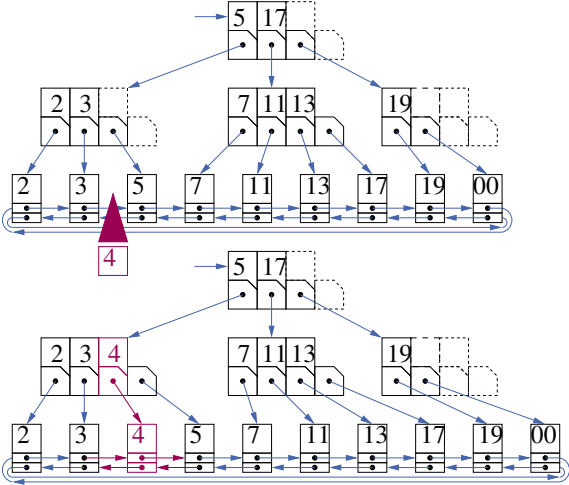
    Weiter oben einfügen, spalten

    ...

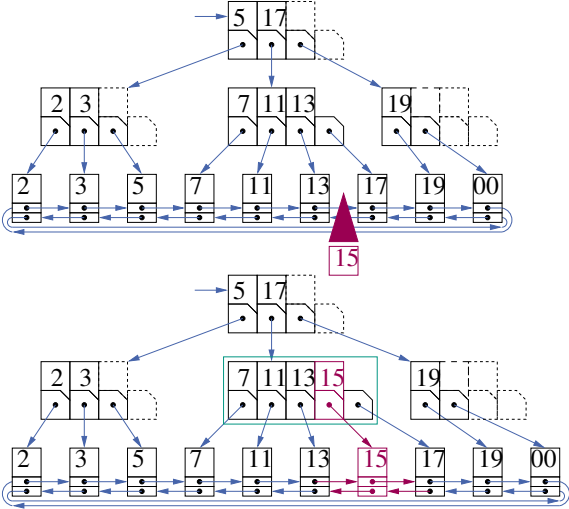
    ggf. neue Wurzel



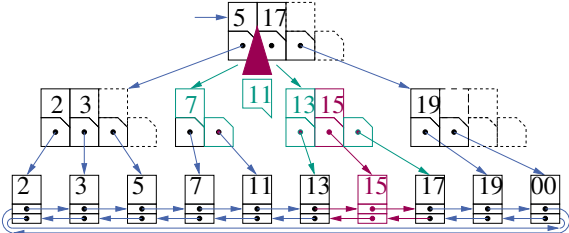
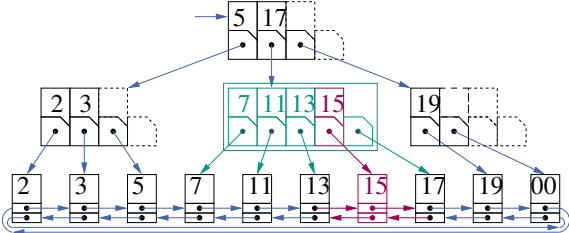
# Einfügen – Beispiel



# Einfügen – Beispiel

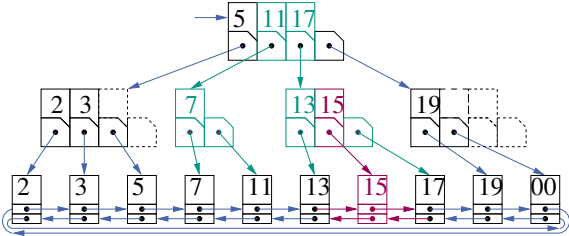
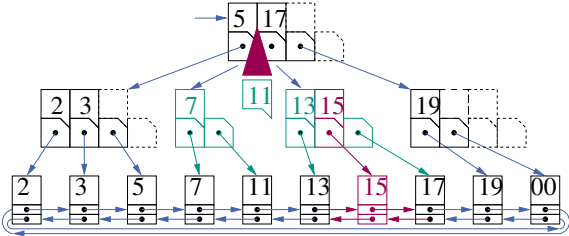


# Einfügen – Beispiel



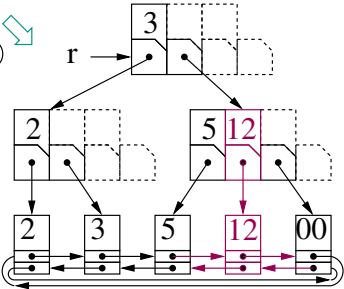
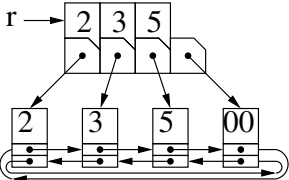
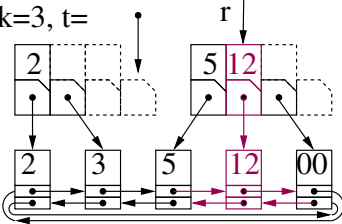


# Einfügen – Beispiel

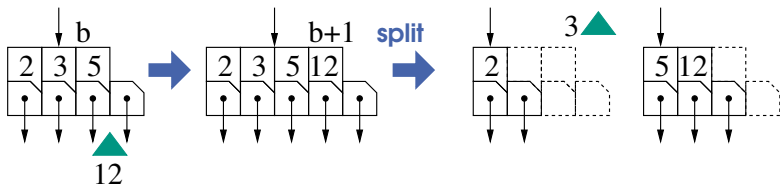


# Einfügen – Beispiel

$k=3, t=$



# Einfügen – Korrektheit



Nach dem Spalten müssen zulässige Items entstehen:

$$\left\lfloor \frac{b+1}{2} \right\rfloor \geq a \Leftrightarrow b \geq 2a - 1$$

Weil  $\left\lfloor \frac{(2a-1)+1}{2} \right\rfloor = \left\lfloor \frac{2a}{2} \right\rfloor = a$

# Einfügen – Implementierungsdetails

- ▶ Spalten pflanzt sich **von unten** nach oben fort. Aber wir speichern nur Zeiger **nach unten**.  
**Lösung:** **Rekursionsstapel** speichert Pfad.
- ▶ Einheitlicher Itemdatentyp mit **Kapazität für  $b$**  Nachfolger.  
einfacher, schneller, Speicherverwaltung!
- ▶ Baue nie explizit temporäre Knoten mit  **$b+1$**  Nachfolgern.

# Einfügen – Pseudocode

```
//  $\ell$ : “the list”
```

```
//  $r$ : root
```

```
// height (of tree)
```

```
Procedure ABTree::insert( $e : \textit{Element}$ )
```

```
  ( $k, t$ ) :=  $r$ .insertRec( $e, \text{height}, \ell$ )
```

```
  if  $t \neq \text{null}$  then
```

```
     $r := \text{allocate}$  ABItem( $\langle k \rangle, \langle r, t \rangle$ )
```

```
    height++
```

```

Function ABItem::insertRec( $e$  : Element,  $h$  :  $\mathbb{N}$ ,  $\ell$  : List of Element) :
    Key  $\times$  ABHandle
 $i :=$  locateLocally( $e$ )
if  $h = 1$  then ( $k, t :=$  (key( $e$ ),  $\ell$ .insertBefore( $e, c[i]$ ))) // base
else ( $k, t := c[i] \rightarrow$  insertRec( $e, h - 1, \ell$ ) // recurse
        if  $t = \text{null}$  then return ( $\perp, \text{null}$ ))
 $s' :=$   $\langle s[1], \dots, s[i - 1], k, s[i], \dots, s[d - 1] \rangle$  // new splitter
 $c' :=$   $\langle c[1], \dots, c[i - 1], t, c[i], \dots, c[d] \rangle$  // new child
if  $d < b$  then ( $s, c, d := (s', c', d + 1)$ ; return ( $\perp, \text{null}$ ))
else // split this node
     $d := \lfloor (b + 1) / 2 \rfloor$ 
     $s := s'[b + 2 - d..b]$ 
     $c := c'[b + 2 - d..b + 1]$ 
    return ( $s'[b + 1 - d],$ 
            allocate ABItem( $s'[1..b - d], c'[1..b + 1 - d]$ ))

```

# Entfernen – Algorithmenskizze

## Procedure remove( $e$ )

Finde Pfad Wurzel  $\rightarrow e$

$l.remove(e)$

entferne  $key(e)$  in Vorgänger  $u$

**if**  $u.d = a - 1$  **then**

finde Nachbarn  $u'$

**if**  $u'.d + a - 1 \leq b$  **then**

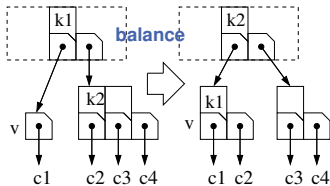
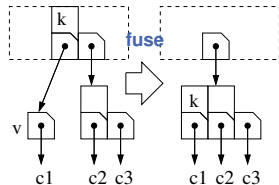
**fuse**( $u', u$ )

Weiter oben splitter entfernen

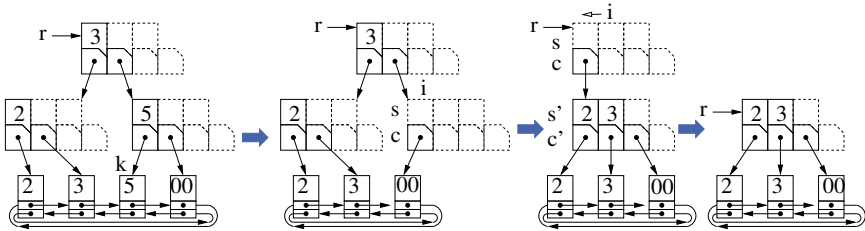
...

ggf. Wurzel entfernen

**else** **balance**( $u', u$ )

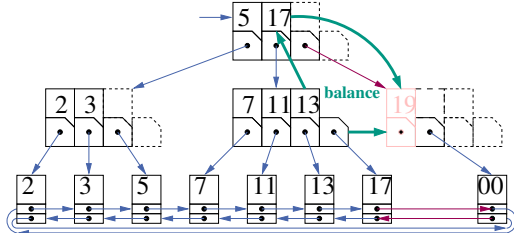
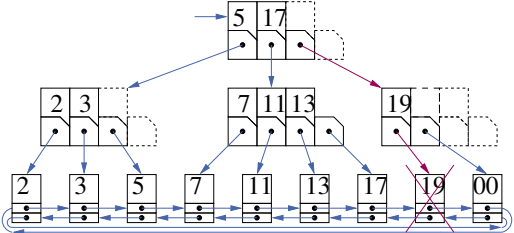


# Entfernen – Beispiel

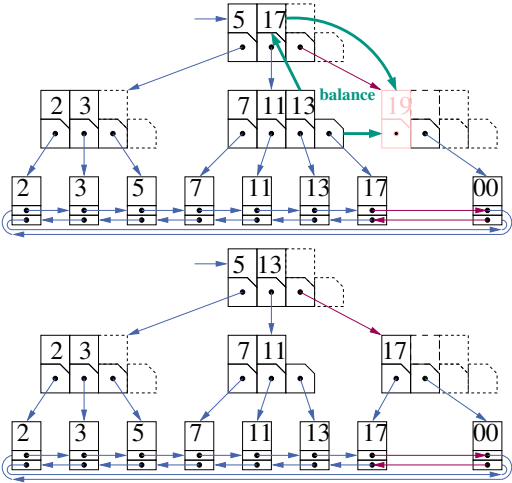




# Entfernen – Beispiel



# Entfernen – Beispiel



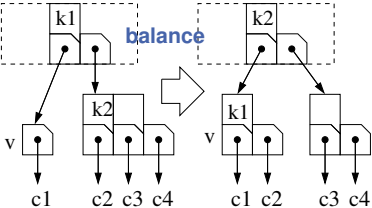
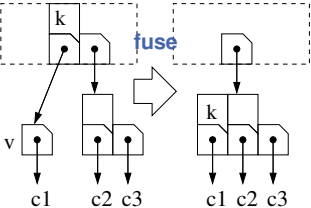
# Entfernen – Korrektheit

Nach **fuse**

müssen zulässige Items entstehen:

$$a + (a - 1) \stackrel{!}{\leq} b \Leftrightarrow b \geq 2a - 1$$

hatten wir schon!



# Einfügen und Entfernen – Laufzeit

$$\begin{aligned} O(b \cdot \text{Höhe}) &= O(b \log_a n) \\ &= O(\log n) \text{ für } \{a, b\} \subseteq O(1) \end{aligned}$$

# $(a, b)$ -Bäume

## Implementierungsdetails

Etwas kompliziert. . .

Wie merkt man sich das?

Gar nicht!

Man merkt sich:

- ▶ **Invarianten**  
Höhe, Knotengrade
- ▶ **Grundideen**  
split, balance, fuse

Den Rest **leitet** man  
sich nach Bedarf **neu her**.

```
Procedure ABTree::remove( $k$  : Key)
   $r$ .removeRec( $k$ , height,  $\ell$ )
  if  $r.d = 1 \wedge \text{height} > 1$  then  $r' := r$ ;  $r := r'.c[1]$ ; dispose  $r'$ 
Procedure ABItem::removeRec( $k$  : Key,  $h$  :  $\mathbb{N}$ ,  $\ell$  : List of Element)
   $i := \text{locateLocally}(k)$ 
  if  $h = 1$  then
    if key( $c[i] \rightarrow e$ ) =  $k$  then
       $\ell$ .remove( $c[i]$ )
      removeLocally( $i$ )
  else
     $c[i] \rightarrow \text{removeRec}(e, h - 1, \ell)$ 
    if  $c[i] \rightarrow d < a$  then
      if  $i = d$  then  $i --$ 
         $s' := \text{concatenate}(c[i] \rightarrow s, (s[i]), c[i + 1] \rightarrow s)$ 
         $c' := \text{concatenate}(c[i] \rightarrow c, c[i + 1] \rightarrow c)$ 
         $d' := \lfloor c' \rfloor$ 
        if  $d' \leq b$  then // fuse
          ( $c[i + 1] \rightarrow s, c[i + 1] \rightarrow c, c[i + 1] \rightarrow d$ ) := ( $s', c', d'$ )
          dispose  $c[i]$ ; removeLocally( $i$ )
        else // balance
           $m := \lceil d' / 2 \rceil$ 
          ( $c[i] \rightarrow s, c[i] \rightarrow c, c[i] \rightarrow d$ ) := ( $s'[1..m - 1], c'[1..m], m$ )
          ( $c[i + 1] \rightarrow s, c[i + 1] \rightarrow c, c[i + 1] \rightarrow d$ ) :=
            ( $s'[m + 1..d' - 1], c'[m + 1..d']$ ,  $d' - m$ )
           $s[i] := s'[m]$ 
Procedure ABItem::removeLocally( $i$  :  $\mathbb{N}$ )
   $c[i..d - 1] := c[i + 1..d]$ 
   $s[i..d - 2] := s[i + 1..d - 1]$ 
   $d --$ 
```

# Mehr Operationen

**min**, **max**, **rangeSearch**( $a, b$ ):

hatten wir schon

$\langle \text{min}, \dots, a, \dots, b, \dots, \text{max} \rangle$

**build**:

(Navigationstruktur für **sortierte** Liste aufbauen)

Übung! Laufzeit  $O(n)$ !

**concat**, **split**: nicht hier.

Zeit  $O(\log n)$

Idee: Ganze Teilbäume umhängen

**merge**( $N, M$ ): sei  $n = |N| \leq m = |M|$

Zeit  $O(n \log \frac{m}{n})$

nicht hier. Idee: z. B. Fingersuche

# Amortisierte Analyse von insert und remove

nicht hier.

Grob gesagt: Abgesehen von der Suche fällt nur konstant viel Arbeit an (summiert über alle Operationsausführungen).

# Erweiterte (augmentierte) Suchbäume

Idee: zusätzliche Infos verwalten  $\rightsquigarrow$  mehr (schnelle) Operationen.

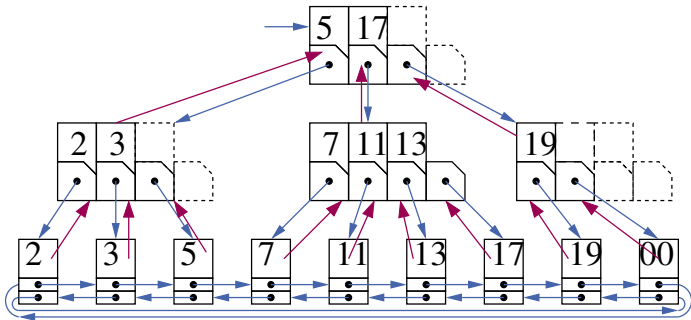
Nachteil: Zeit- und Platzverschwendung,  
wenn diese Operationen nicht wichtig sind.

gold plating



# Elternzeiger

Idee: Knoten speichern Zeiger auf Elternknoten



Anwendungen: schnelleres **remove**, **insertBefore**, **insertAfter**,  
**falls** man ein **handle** des Elements kennt.  
Man spart die Suche.

**Frage:** was speichert man bei  $(a, b)$ -Bäumen (zusätzlich)?

# Teilbaumgrößen

Idee (Binärbaum): speichere, wie viele Blätter von links erreichbar.  
(Etwas anders als im Buch!)

// return  $k$ -th Element in subtree rooted at  $h$

**Function** selectRec( $h, k$ )

**if**  $h \rightarrow \text{leftSize} \geq k$  **then return** select( $\ell, k$ )

**else return** select( $r, k - \text{leftSize}$ )

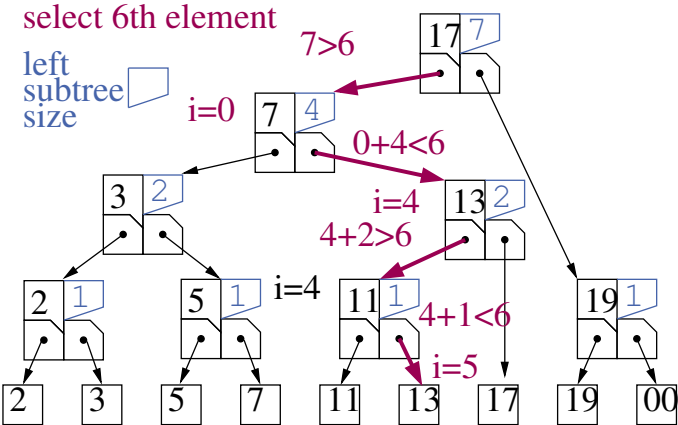
Zeit:  $O(\log n)$

Übung: Was ist anders bei  $(a, b)$ -Bäumen?

Übung: Rang eines Elements  $e$  bestimmen.

Übung: Größe eines Bereichs  $a..b$  bestimmen.

# Beispiel



# Zusammenfassung

- ▶ **Suchbäume** erlauben viele effiziente Operationen auf **sortierten Folgen**.
- ▶ Oft **logarithmische** Ausführungszeit
- ▶ Der schwierige Teil: logarithmische Höhe erzwingen.
- ▶ Augmentierungen  $\rightsquigarrow$  zusätzliche Operationen

# Mehr zu sortierten Folgen

- ▶ **Karteikasten**  $\rightsquigarrow$  Array mit Löchern
- ▶  $(a, b)$ -Bäume sind wichtig für **externe** Datenstrukturen
- ▶ Ganzzahlige Schlüssel aus  $1..U$   
 $\rightsquigarrow$  Grundoperationen in Zeit  $O(\log \log U)$
- ▶ Verallgemeinerungen: **Zeichenketten**, **mehrdimensionale** Daten

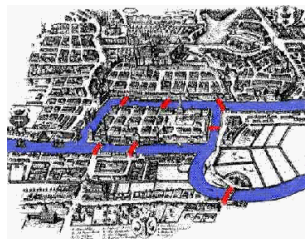
# Was haben wir noch gelernt?

- ▶ **Invarianten**, Invarianten, Invarianten
- ▶ Komplexe **verzeigerte Datenstrukturen**
- ▶ Datenstruktur-**Augmentierung**
- ▶ Unterschied **Interface** ↔ **Repräsentation**
- ▶ Tradeoff Array, sortierte Liste, Hash-Tabelle

# Kap. 8: Repräsentation von Graphen

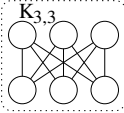
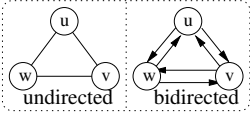
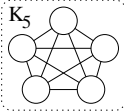
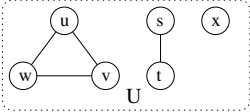
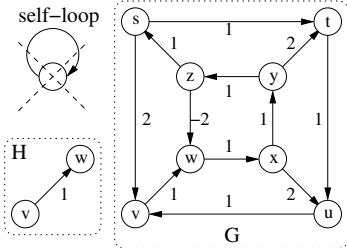
## Einleitung

- ▶ 1736 stellt L. Euler die folgende “touristische” Frage:
- ▶ Straßen- oder Computernetzwerke
- ▶ Zugverbindungen (Raum und Zeit)
- ▶ Soziale Netzwerke (Freundschafts-, Zitier-, Empfehlungs-, . . .)
- ▶ Aufgabenabhängigkeiten  $\rightsquigarrow$  Scheduling-Probleme
- ▶ Werte und arithmetische Operationen  $\rightsquigarrow$  Compilerbau
- ▶ . . .



# Repräsentation von Graphen

- ▶ Was zählt, sind die **Operationen!**
- ▶ Eine **triviale** Repräsentation
- ▶ **Felder**
- ▶ Verkettete **Listen**
- ▶ **Matrizen**
- ▶ **Implizit**
- ▶ **Diskussion**





# Notation und Konventionen

▶ Graph  $G = ( \underbrace{V}, \underbrace{E} )$ :  
Knoten Kanten

▶  $n = |V|$

▶  $m = |E|$

▶ Knoten:  $s, t, u, v, w, x, y, z$

▶ Kanten  $e \in E$ .

Oder: Knotenpaare (manchmal Knotenmengen der Größe 2)

# Notation und Konventionen

▶ Graph  $G = ( \underbrace{V}, \underbrace{E} )$ :  
Knoten Kanten

▶  $n = |V|$

▶  $m = |E|$

▶ Knoten:  $s, t, u, v, w, x, y, z$

▶ Kanten  $e \in E$ .

Oder: Knotenpaare (manchmal Knotenmengen der Größe 2)

**WICHTIG:** Buchstabenzuordnungen sind **unverbindliche** Konvention

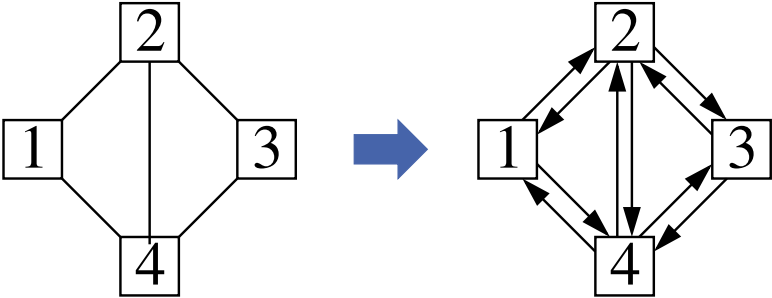
▶ Manchmal werden ganz andere Buchstaben verwendet.

▶ Im Zweifel immer genau sagen, was was ist.

Das gilt für die ganze theoretische Informatik!

# Ungerichtete $\rightarrow$ gerichtete Graphen

Meist repräsentieren wir ungerichtete Graphen durch **doppelt gerichtete** Graphen  $\rightsquigarrow$  wir konzentrieren uns auf gerichtete Graphen



# Operationen

**Ziel:**  $O(\text{Ausgabegröße})$  für alle Operationen

Grundoperationen:

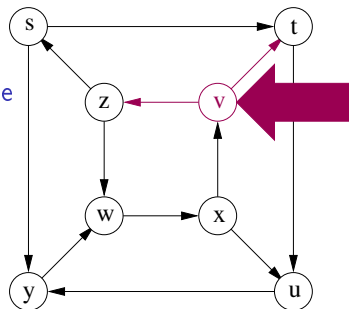
- ▶ Statische Graphen:

Konstruktion, Konversion und Ausgabe  
( $O(m+n)$  Zeit)

**Navigation:** Gegeben  $v$ , finde  
ausgehende Kanten.

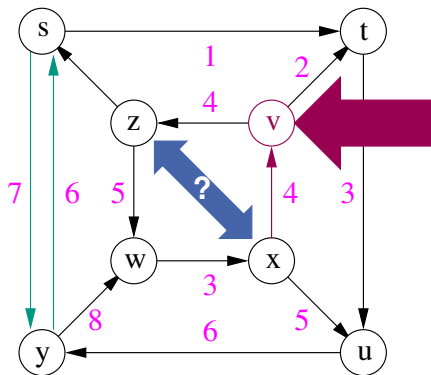
- ▶ Dynamische Graphen:

Knoten/Kanten einfügen/löschen



# Weitere Operationen

- ▶ Zugriff auf assoziierte Information
- ▶ Mehr Navigation: Finde eingehende Kanten
- ▶ Kantenanfragen:  $(z,x) \in E?$

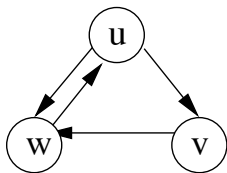


# Kantenfolgenrepräsentation

Folge von Knotenpaaren (oder Tripel mit Kantengewicht)

- + kompakt
- + gut für I/O
- Fast keine nützlichen Operationen – außer alle Kanten zu durchlaufen

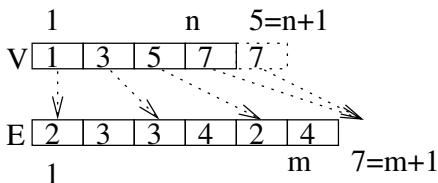
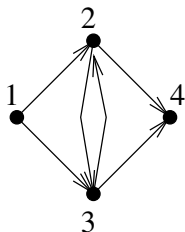
**Beispiele:** Übung: isolierte Knoten suchen,  
Kruskals MST-Algorithmus (später), Konvertierung.



$$\Leftrightarrow \langle (u, v), (v, w), (w, u), (u, w) \rangle$$

# Adjazenzfelder

- ▶  $V = 1..n$  oder  $0..n-1$
- ▶ **Kantenfeld**  $E$  speichert **Ziele** und zwar **gruppiert** nach Startknoten
- ▶  $V$  speichert Index der ersten ausgehenden Kante
- ▶ **Dummy**-Eintrag  $V[n+1]$  speichert  $m+1$



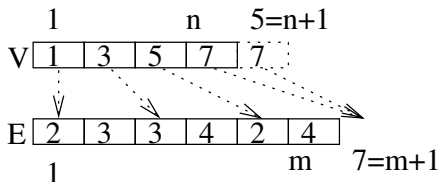
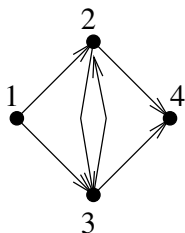
**Beispiel:**  $\text{Ausgangsgrad}(v) = V[v+1] - V[v]$

# Kantenliste $\rightarrow$ Adjazenzfeld

Zur Erinnerung: KSort (BucketSort)

**Function** adjacencyArray(EdgeList)

```
V =  $\langle 1, 0, \dots, 0 \rangle$  : Array [1..n+1] of  $\mathbb{N}$   
foreach  $(u, v) \in \text{EdgeList}$  do  $V[u]++$  // count  
for  $v := 2$  to  $n+1$  do  $V[v] += V[v-1]$  // prefix sums  
foreach  $(u, v) \in \text{EdgeList}$  do  $E[--V[u]] = v$  // place  
return  $(V, E)$ 
```

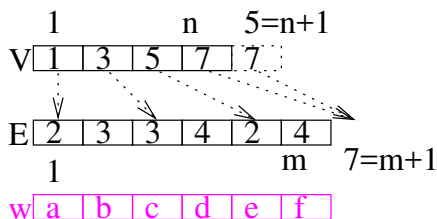
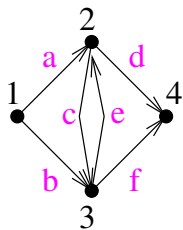






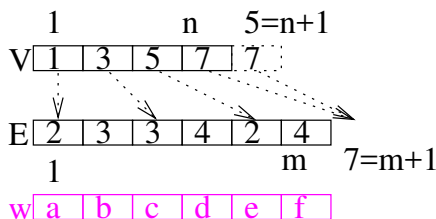
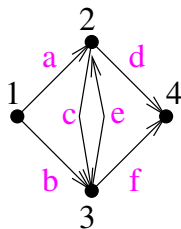
# Operationen für Adjanzenzfelder

- ▶ Navigation: einfach
- ▶ **Kantengewichte**:  $E$  wird Feld von Records (oder mehrere Felder)
- ▶ **Knoteninfos**:  $V$  wird Feld von Records (oder mehrere Felder)



# Operationen für Adjanzenzfelder

- ▶ Navigation: einfach
- ▶ **Kantengewichte**:  $E$  wird Feld von Records (oder mehrere Felder)
- ▶ **Knoteninfos**:  $V$  wird Feld von Records (oder mehrere Felder)
- ▶ **Eingehende Kanten**: umgedrehten Graphen speichern
- ▶ **Kanten löschen**: explizite Endindizes
- ▶ **Batched Updates**:  
neu aufbauen



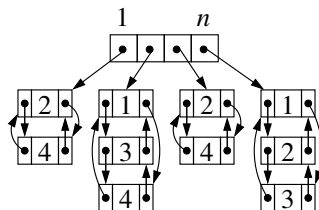
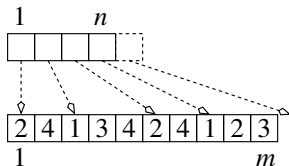
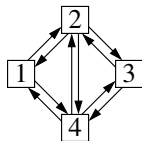
# Kantenanfragen

Hashtabelle  $H_E$  speichert (ggf. zusätzlich) alle Kanten.  
Unabhängig von der sonstigen Graphrepräsentation

# Adjanzlisten

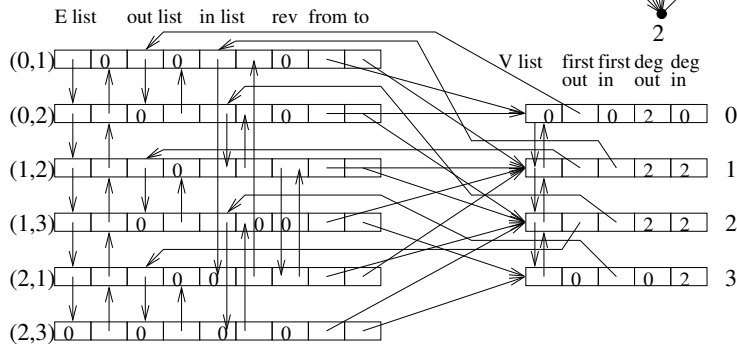
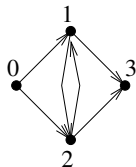
speichere (doppelt) verkettete **Liste** adjazenter Kanten für jeden Knoten.

- + einfaches **Einfügen** von Kanten
- + einfaches **Löschen** von Kanten (ordnungserhaltend)
- mehr Platz (bis zu Faktor 3) als Adjanzfelder
- mehr Cache-Misses



# Adjazenzlisten aufrüsten

- ▶ **Knotenlisten** für Knotenupdates
- ▶ Eingehende Kanten
- ▶ Kantenobjekte (in globaler Kantenliste)
- ▶ Zeiger auf Umkehrkante



# Customization (Zuschneiden)

Anpassen der (Graph)Datenstruktur an die Anwendung.

- ▶ Ziel: schnell, kompakt.
- ▶ benutze Entwurfsprinzip: **make the common case fast**
- ▶ Listen vermeiden

**Mögliches Problem:** **Software-Engineering**-Alptraum

Möglicher Ausweg: Trennung von Algorithmus und Repräsentation

# Beispiel: DAG-Erkennung

Beispiel aus Notations-Kapitel (generisch, leicht variiert):

```
Function isDAG( $G = (V, E)$ )  
  while  $\exists v \in V : \text{indegree}(v) = 0$  do  
    invariant  $G$  is a DAG iff the input graph is a DAG  
     $V := V \setminus \{v\}$   
     $E := E \setminus (\{v\} \times V \cup V \times \{v\})$   
  return  $|V|=0$ 
```



## Beispiel: DAG-Erkennung

```
Function isDAG( $G = (V, E)$ ) // Adjazenzarray!  
  dropped := 0  
  compute array inDegree of indegrees of all nodes // Zeit  $O(m + n)$ !  
  droppable = {  $v \in V : \text{inDegree}[v] = 0$  } : Stack  
  while droppable  $\neq \emptyset$  do  
    invariant  $G$  is a DAG iff the input graph is a DAG  
     $v := \text{droppable.pop}$   
    dropped ++  
    foreach edge  $(v, w) \in E$  do  
      inDegree[ $w$ ] --  
      if inDegree[ $w$ ] = 0 then droppable.push( $w$ )  
  return  $|V| = \text{dropped}$ 
```

**Laufzeit:**  $O(m + n)$  (auch ohne dynamische Graphdatenstruktur!)

# Adjazenz-Matrix

$A \in \{0,1\}^{n \times n}$  with  $A(i,j) = [(i,j) \in E]$

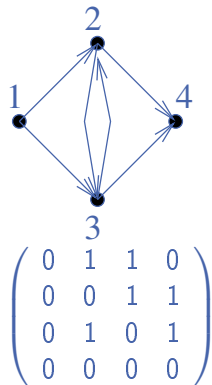
- + platzeffizient für sehr **dichte Graphen**
- platzineffizient sonst.
- Übung: was bedeutet "sehr dicht" hier?
- + einfache **Kantenanfragen**
- langsame Navigation
- ++ verbindet **lineare Algebra** und Graphentheorie

Beispiel:  $C = A^k$ .

$C_{ij} = \#$   $k$ -Kanten-Pfade von  $i$  nach  $j$

Wichtige **Beschleunigungstechniken**:

- ▶  $O(\log k)$  Matrixmult. für Potenzberechnung
- ▶ Matrixmultiplikation in subkubischer Zeit, z. B., **Strassens** Algorithmus



# Pfade zählen mittels LA

## Adjanzenzmatrix:

$A \in \{0, 1\}^{n \times n}$  mit  $A(i, j) = [(i, j) \in E]$

Sei  $C := A^k$ .

**Behauptung:**  $C_{ij} = \#$   $k$ -Kanten-Pfade von  $i$  nach  $j$ .

**Beweis:** IA ( $k = 1$ )  $C = A^1 = A$  stimmt nach Definition von  $A$ .

Schluss  $k \rightsquigarrow k + 1$ :  $C_{ij} = (A^k A)_{ij} = \sum_{\ell} A_{i\ell}^k A_{\ell j}$

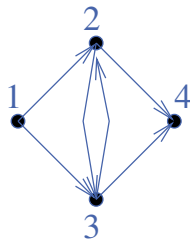
$A_{i\ell}^k = \#k$ -Kanten-Pfade von  $i$  nach  $\ell$  (nach IV).

$A_{i\ell}^k A_{\ell j} = \#k + 1$ -Kanten-Pfade von  $i$  nach  $j$

mit  $(\ell, j)$  als letzter Kante.

Jede mögliche letzte Kante wird genau einmal gezählt.

□



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Übung: zähle Pfade der Länge  $\leq k$

## Beispiel, wo Graphentheorie bei LA hilft

**Problemstellung:** löse  $\mathbf{Bx} = \mathbf{c}$

Sei  $G = (1..n, E = \{\{i,j\} : B_{ij} \neq 0\})$

Nehmen wir an,  $G$  habe zwei **Zusammenhangskomponenten**

$\Rightarrow$  tausche Zeilen und Spalten derart, dass

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix}$$

zu lösen bleibt.

Übung: Was passiert, wenn  $(1..n, E = \{(i,j) : B_{ij} \neq 0\})$  ein DAG ist?

# Implizite Repräsentation

**Kompakte** Repräsentation möglicherweise sehr dichter Graphen  
**Implementiere** Algorithmen **direkt** mittels dieser Repräsentation

**Beispiel:** Intervall-Graphen

Knoten: Intervalle  $[a, b] \subseteq \mathbb{R}$

Kanten: zwischen überlappenden Intervallen

# Zusammenhangstest für Intervallgraphen

$$V = \{[a_1, b_1], \dots, [a_n, b_n]\}$$

$$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$$

**Idee:** **durchlaufe** Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf null sinken.

**Annahme:** Startpunkte in Sortierung vor Endpunkten!

**Function** isConnected( $L$  : SortedListOfIntervalEndPoints) : {0, 1}

remove first element of  $L$ ; overlap := 1

**foreach**  $p \in L$  **do**

**if** overlap = 0 **return** false

**if**  $p$  is a start point **then** overlap++

**else** overlap--

// end point

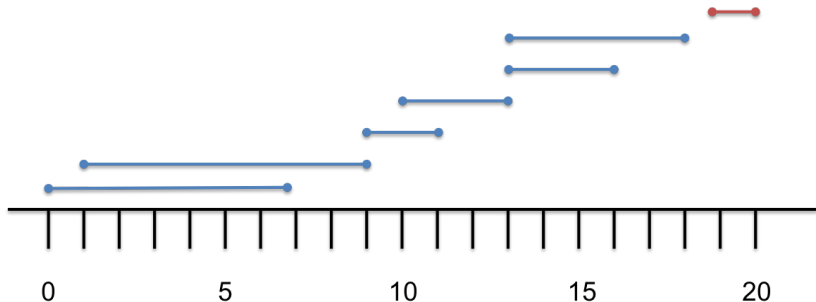
**return** true

$O(n \log n)$  Algorithmus für bis zu  $O(n^2)$  Kanten!

Übung: Zusammenhangskomponenten finden

# Beispiel

```
Function isConnected( $L$  : SortedListOfIntervalEndPoints) : {0,1}  
  remove first element of  $L$ ;  overlap := 1  
  foreach  $p \in L$  do  
    if overlap = 0 return false  
    if  $p$  is a start point then overlap++  
    else overlap-- // end point  
  return true
```



# Graphrepräsentation: Zusammenfassung

- ▶ Welche **Operationen** werden gebraucht?
- ▶ **Wie oft?**
- ▶ Adjazenz**arrays** gut für statische Graphen
- ▶ Pointer  $\rightsquigarrow$  flexibler, aber auch teurer
- ▶ Matrizen eher konzeptionell interessant

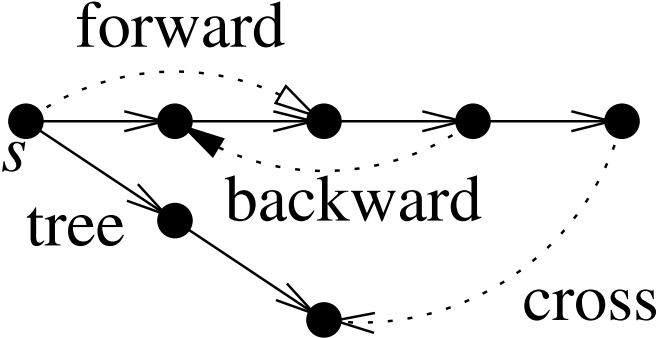


# Kap. 9: Graphtraversierung



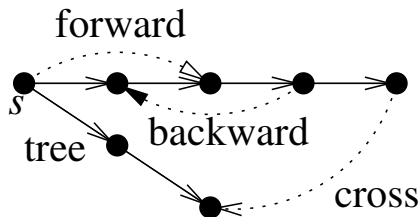
Ausgangspunkt oder Baustein fast jedes nichttrivialen Graphenalgorithmus

# Graphtraversierung als Kantenklassifizierung



# Graphtraversierung als Kantenklassifizierung

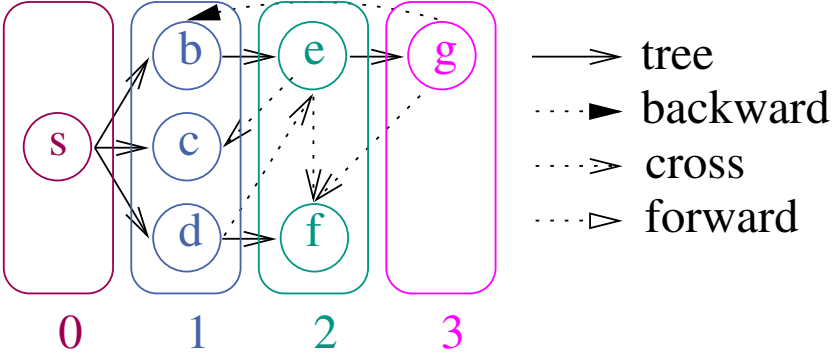
- ▶ Baumkanten: Elemente des Waldes, der bei der Suche gebaut wird
- ▶ Vorwärtskanten: verlaufen **parallel** zu Wegen aus Baumkanten
- ▶ Rückwärtskanten: verlaufen **antiparallel** zu Wegen aus Baumkanten
- ▶ Querkanten: alle übrigen



# Breitensuche

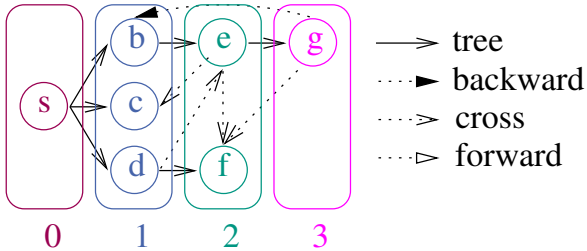
Baue Baum von **Startknoten s**,  
der alle von s erreichbaren Knoten  
mit möglichst **kurzen** Pfaden erreicht.

Berechne Abstände:



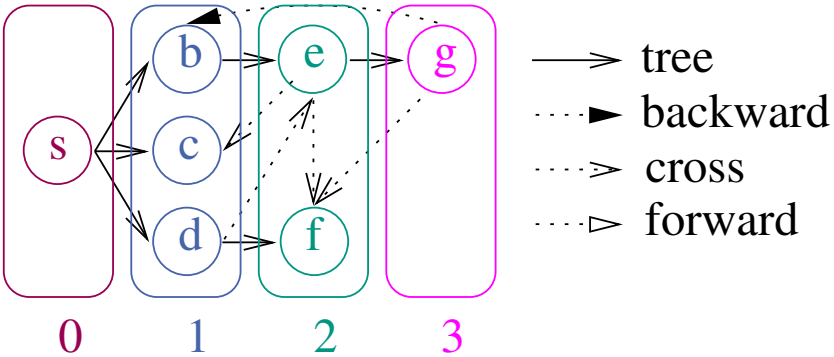
# Breitensuche

- ▶ Einfachste Form des **Kürzeste-Wege-Problems**
- ▶ **Umgebung** eines Knotens definieren (ggf. begrenzte Suchtiefe)
- ▶ Einfache, effiziente Graphtraversierung (auch wenn Reihenfolge egal)



# Breitensuche

Algorithmenidee: Baum **Schicht für Schicht** aufbauen



**Function** bfs(s) :

$Q := \langle s \rangle$

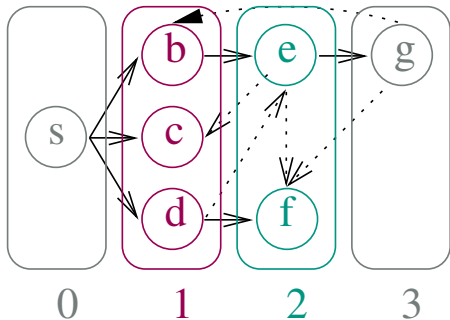
// aktuelle Schicht

**while**  $Q \neq \langle \rangle$  **do**

exploriere Knoten in  $Q$

merke dir Knoten der nächsten Schicht in  $Q'$

$Q := Q'$

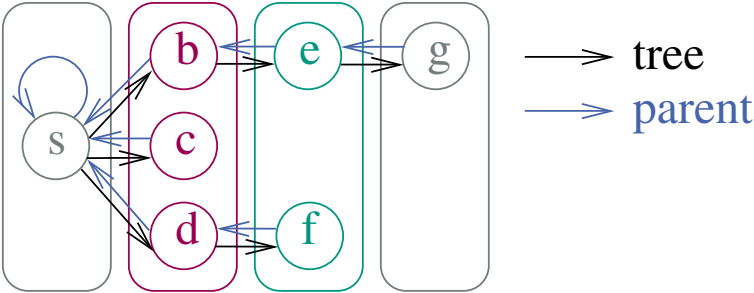


- tree
- ...▶ backward
- ...▷ cross
- ...▷ forward

# Repräsentation des Baums

Feld **parent** speichert Vorgänger.

- ▶ noch nicht erreicht:  $\text{parent}[v] = \perp$
- ▶ Startknoten/Wurzel:  $\text{parent}[s] = s$





**Function** bfs( $s$  : Nodeld) :

```
(NodeArray of Nodeld) × (NodeArray of  $\mathbb{N}_0 \cup \{\infty\}$ )  
 $d = \langle \infty, \dots, \infty \rangle$  : NodeArray of  $\mathbb{N}_0 \cup \{\infty\}$ ;  $d[s] := 0$   
parent =  $\langle \perp, \dots, \perp \rangle$  : NodeArray of Nodeld; parent[s] :=  $s$   
 $Q = \langle s \rangle, Q' = \langle \rangle$  : Set of Nodeld // current, next layer  
for ( $\ell := 0$ ;  $Q \neq \langle \rangle$ ;  $\ell++$ )  
  invariant  $Q$  contains all nodes with distance  $\ell$  from  $s$   
  foreach  $u \in Q$  do  
    foreach  $(u, v) \in E$  do // scan  $u$   
      if parent( $v$ ) =  $\perp$  then // unexplored  
         $Q' := Q' \cup \{v\}$   
         $d[v] := \ell + 1$ ;  
        parent( $v$ ) :=  $u$   
   $(Q, Q') := (Q', \langle \rangle)$  // next layer  
return (parent,  $d$ ) // BFS =  $\{(v, w) : w \in V, v = \text{parent}(w)\}$ 
```

# Repräsentation von $Q$ und $Q'$ mittels FIFO

$Q, Q' \rightarrow$  einzelne FIFO-Queue

- ▶ Standardimplementierung in anderen Büchern
- + „Oberflächlich“ **einfach**
- **Korrektheit** mglw. weniger evident
- = Effizient (?)

Übung!

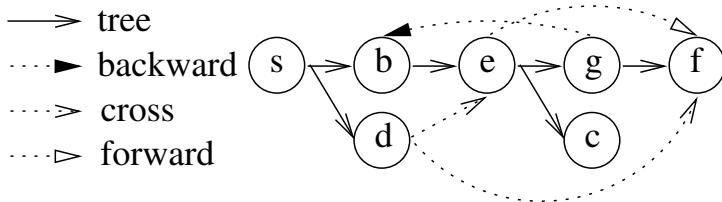
Übung: ausprobieren!

# Alternative Repräsentation von $Q$ und $Q'$

- ▶ Zwei Stapel
- ▶ Schleife  $1\times$  ausrollen  
`loop  $Q \rightarrow Q'$ ;  $Q' \rightarrow Q$`
- ▶ Beide Stapel in **ein Feld** der Größe  $n$



# Tiefensuche



# Tiefensuchschema für $G = (V, E)$

unmark all nodes;

init

foreach  $s \in V$  do

  if  $s$  is not marked then

    mark  $s$

    root( $s$ )

    DFS( $s, s$ )

    // make  $s$  a root and grow  
    // a new DFS tree rooted at  $s$

Procedure DFS( $u, v : \text{NodeId}$ )

// Explore  $v$  coming from  $u$

  foreach  $(v, w) \in E$  do

    if  $w$  is marked then traverseNonTreeEdge( $v, w$ )

    else traverseTreeEdge( $v, w$ )

      mark  $w$

      DFS( $v, w$ )

  backtrack( $u, v$ )

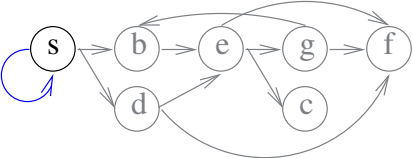
    // return from  $v$  along the incoming edge

# DFS-Baum

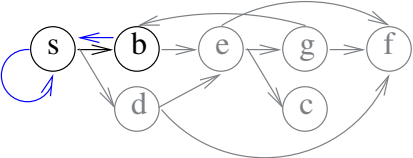
init:  $\text{parent} = \langle \perp, \dots, \perp \rangle$  : NodeArray of NodeId  
 root(s):  $\text{parent}[s] := s$   
 traverseTreeEdge(v, w):  $\text{parent}[w] := v$

→ tree  
 → parent

mark s  
 root(s)  
 dfs(s,s)  
 traverseTreeEdge(s,b)



mark b  
 dfs(s,b)



dfs(s,b)

traverseTreeEdge(b,e)

mark(e)

dfs(b,e)

traverseTreeEdge(e,g)

mark(g)

dfs(e,g)

traverseNonTreeEdge(g,b)

traverseTreeEdge(g,f)

mark(f)

dfs(g,f)

backtrack(g,f)

backtrack(e,g)

traverseNonTreeEdge(e,f)

traverseTreeEdge(e,c)

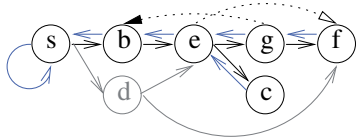
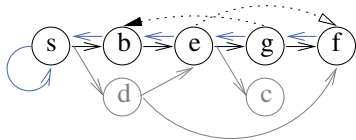
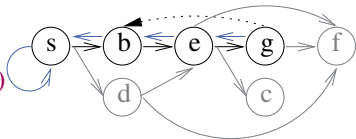
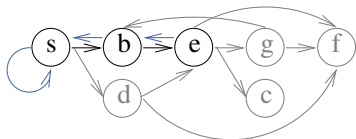
mark(c)

dfs(e,c)

backtrack(e,c)

backtrack(b,e)

backtrack(s,b)



traverseTreeEdge(s,d)

mark(d)

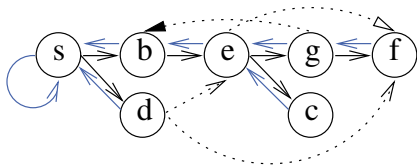
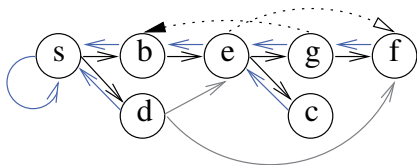
dfs(s,d)

traverseNonTreeEdge(d,e)

traverseNonTreeEdge(d,f)

backtrack(s,d)

backtrack(s,s)





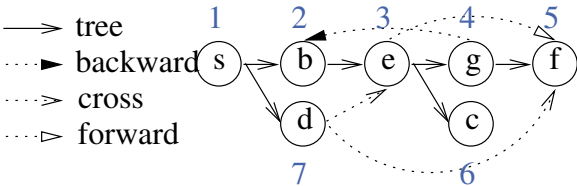
# DFS-Nummerierung

init:  $\text{dfsPos} = 1 : 1..n$   
root( $s$ ):  $\text{dfsNum}[s] := \text{dfsPos}++$   
traverseTreeEdge( $v, w$ ):  $\text{dfsNum}[w] := \text{dfsPos}++$

$$u \prec v \Leftrightarrow \text{dfsNum}[u] < \text{dfsNum}[v] .$$

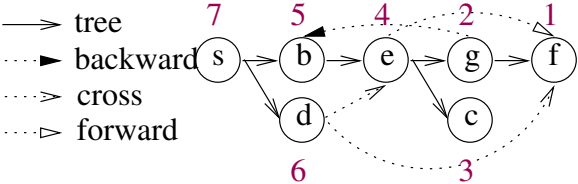
## Beobachtung:

Knoten auf dem Rekursionsstapel sind bzgl.  $\prec$  sortiert



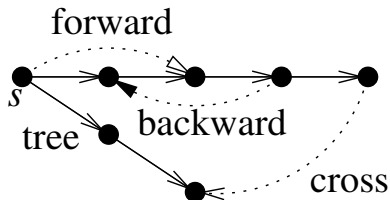
# Fertigstellungszeit

```
init:      finishingTime=1 : 1..n  
backtrack(u, v):  finishTime[v]:= finishingTime++
```



# Kantenklassifizierung bei DFS

type ( $v, w$ )	$\text{dfsNum}[v] < \text{dfsNum}[w]$	$\text{finishTime}[w] < \text{finishTime}[v]$	$w$ is marked
tree	yes	yes	no
forward	yes	yes	yes
backward	no	no	yes
cross	no	yes	yes



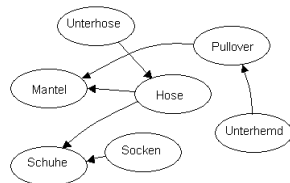
# Topologische Sortierung

## Definition 5

Eine **lineare Anordnung**  $t$  der Knoten eines DAGs  $G = (V, E)$ , in der alle Kanten von “kleineren” zu “größeren” Knoten verlaufen, heißt **topologische Sortierung**, d. h.

$$\forall (u, v) \in E : t(u) < t(v).$$

## Beispiel:



Kleidergraph, Quelle: Wikipedia



topologisch sortierter Kleidergraph, Quelle: Wikipedia

# Topologisches Sortieren mittels DFS

## Theorem 6

$G$  ist *kreisfrei (DAG)*  $\Leftrightarrow$  DFS findet keine Rückwärtskante.  
In diesem Fall liefert

$$t(v) := n - \text{finishTime}[v]$$

eine *topologische Sortierung*.

# Topologisches Sortieren mittels DFS

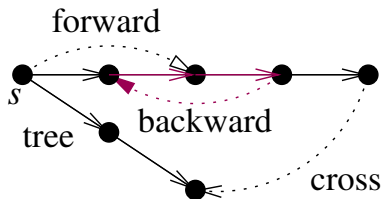
## Theorem 6

$G$  ist *kreisfrei (DAG)*  $\Leftrightarrow$  DFS findet keine Rückwärtskante.  
In diesem Fall liefert

$$t(v) := n - \text{finishTime}[v]$$

eine *topologische Sortierung*.

**Beweis “ $\Rightarrow$ ”:** Annahme:  $\exists$  Rückwärtskante.  
Zusammen mit Baumkanten ergibt sich ein Kreis.  
Widerspruch.



# Topologisches Sortieren mittels DFS

**Satz:**  $G$  **kreisfrei (DAG)**  $\Leftrightarrow$  DFS findet keine Rückwärtskante.  
In diesem Fall liefert  $t(v) := n - \text{finishTime}[v]$  eine **topologische Sortierung**, d. h.  $\forall (u, v) \in E : t(u) < t(v)$ .

**Beweis “ $\Leftarrow$ ”:**

Keine Rückwärtskante  
Kantenklassifizierung



$$\forall (v, w) \in E : \text{finishTime}[v] > \text{finishTime}[w]$$

$\Rightarrow G$  ist kreisfrei und

$\text{finishTime}$  definiert umgekehrte topologische Sortierung.

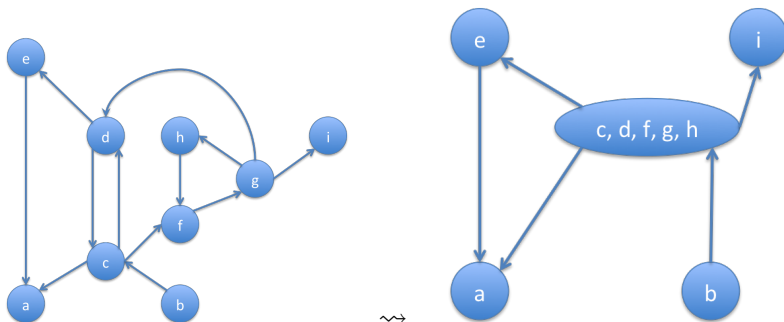
# Starke Zusammenhangskomponenten

Betrachte die Relation  $\overset{*}{\leftrightarrow}$  mit  
 $u \overset{*}{\leftrightarrow} v$  falls  $\exists$  Pfad  $\langle u, \dots, v \rangle$  und  $\exists$  Pfad  $\langle v, \dots, u \rangle$ .

**Beobachtung:**  $\overset{*}{\leftrightarrow}$  ist Äquivalenzrelation

Die **Äquivalenzklassen** von  $\overset{*}{\leftrightarrow}$  bezeichnet man als **starke Zusammenhangskomponenten**.

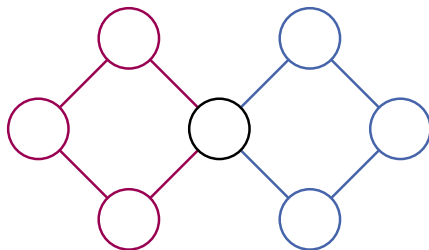
Übung



DFS-basierter Linearzeitalgorithmus  $\rightarrow$  Algorithmen II



# Mehr DFS-basierte Linearzeitalgorithmen



- ▶ 2-zusammenhängende Komponenten: bei Entfernen eines Knotens aus einer Komponente bleibt diese zusammenhängend (ungerichtet)
- ▶ 3-zusammenhängende Komponenten
- ▶ Planaritätstest (lässt sich der Graph kreuzungsfrei zeichnen?)
- ▶ Einbettung planarer Graphen

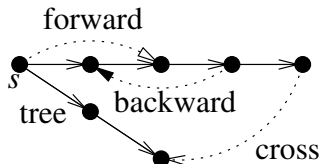
# BFS $\longleftrightarrow$ DFS

## pro BFS:

- ▶ nichtrekursiv
- ▶ keine Vorwärtskanten
- ▶ kürzeste Wege, „Umgebung“

## pro DFS:

- ▶ keine explizite Datenstruktur (Rekursionsstapel) für ToDos, daher mglw. einfacher
- ▶ Grundlage vieler Algorithmen



# Kap. 10: Kürzeste Wege

## Eingabe:

- ▶ Graph  $G = (V, E)$  mit
- ▶ Kostenfunktion/Kantengewicht  $c : E \rightarrow \mathbb{R}$  sowie
- ▶ Startknoten  $s$ .



## Ausgabe: für alle $v \in V$ :

- ▶ Länge  $\mu(v)$  des kürzesten Pfades von  $s$  nach  $v$ ,
- ▶  $\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$   
mit  $c(\langle e_1, \dots, e_k \rangle) := \sum_{i=1}^k c(e_i)$ .

# Kap. 10: Kürzeste Wege

## Eingabe:

- ▶ Graph  $G = (V, E)$  mit
- ▶ Kostenfunktion/Kantengewicht  $c : E \rightarrow \mathbb{R}$  sowie
- ▶ Startknoten  $s$ .



## Ausgabe: für alle $v \in V$ :

- ▶ Länge  $\mu(v)$  des kürzesten Pfades von  $s$  nach  $v$ ,
- ▶  $\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$   
mit  $c(\langle e_1, \dots, e_k \rangle) := \sum_{i=1}^k c(e_i)$ .

Oft wollen wir auch „geeignete“ Repräsentation der kürzesten Pfade.

# Anwendungen

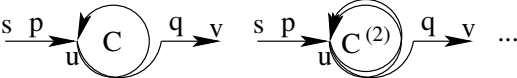
- ▶ Routenplanung
  - ▶ Straßennetze
  - ▶ Spiele
  - ▶ Kommunikationsnetze
- ▶ Unterprogramm
  - ▶ Flüsse in Netzwerken
  - ▶ ...
- ▶ Tippfehlerkorrektur
- ▶ Disk Scheduling
- ▶ ...



# Grundlagen

Gibt es immer einen kürzesten Pfad?

Es kann **negative Kreise** geben!



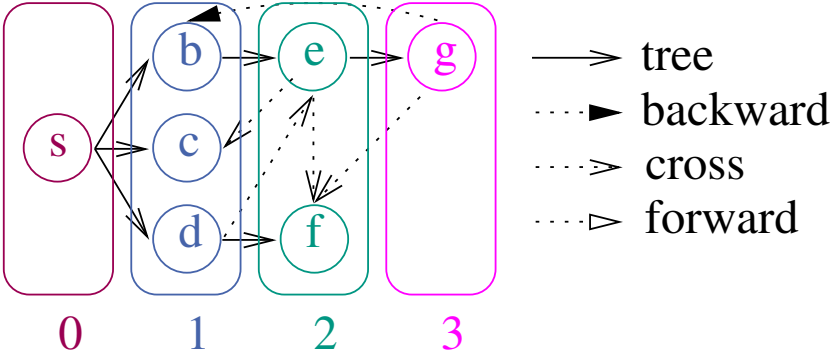
weitere Grundlagen just in time

# Azyklische Graphen

später

Kantengewichte  $\geq 0$

Alle Gewichte gleich: Breitensuche (BFS)!



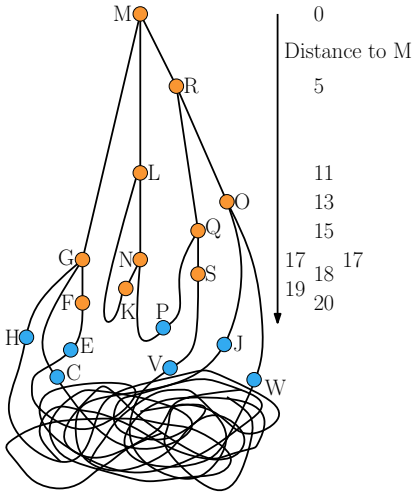


# Dijkstras Algorithmus

**Nun:** Beliebige nichtnegative Kantengewichte

## Lösung ohne Rechner:

- ▶ Kanten → Fäden
- ▶ Kantengewicht → Fadenlänge
- ▶ Knoten → Knoten
- ▶ **Dann:** Am Startknoten anheben.

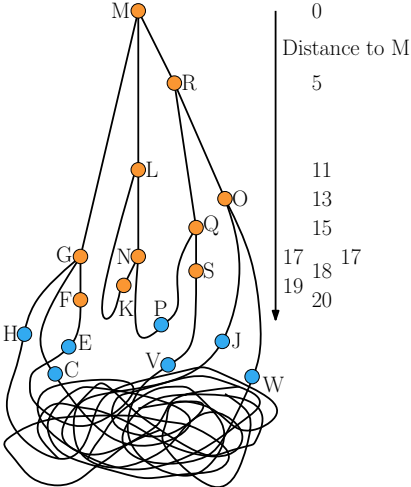


# Korrektheit der Bindfäden

Betrachte beliebigen Knoten  $v$  mit Hängetiefe  $d[v]$ .

$\exists$  **Pfad mit Hängetiefe:**  
verfolge straffe Fäden

$\neg \exists$  **kürzerer Pfad:**  
falls es einen solchen Pfad  
gäbe, wäre einer seiner Fäden  
**zerrissen!**



# Edsger Wybe Dijkstra 1930–2002



Bildquelle: Wikipedia

- ▶ 1972 ACM Turing Award
- ▶ THE: das erste Multitasking-OS
- ▶ Semaphore
- ▶ Selbst-stabilisierende Systeme
- ▶ GOTO Statement Considered Harmful

# Allgemeine Definitionen

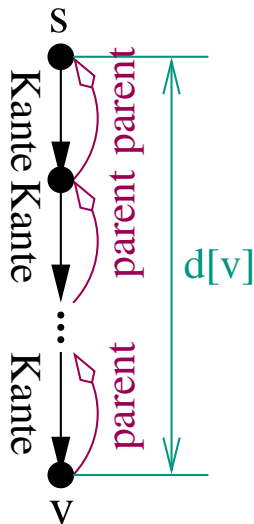
Wie bei BFS benutzen wir zwei Knotenarrays:

- ▶  $d[v]$  = aktuelle (vorläufige) Distanz von  $s$  nach  $v$   
**Invariante:**  $d[v] \geq \mu(v)$
- ▶  $\text{parent}[v]$  = Vorgänger von  $v$  auf dem (vorläufigen) kürzesten Pfad von  $s$  nach  $v$   
**Invariante:** dieser Pfad bezeugt  $d[v]$

**Initialisierung:**

$$d[s] = 0, \text{parent}[s] = s$$

$$d[v] = \infty, \text{parent}[v] = \perp$$



# Kante $(u, v)$ relaxieren

Falls  $d[u] + c(u, v) < d[v]$   
(vielleicht  $d[v] = \infty$ ),

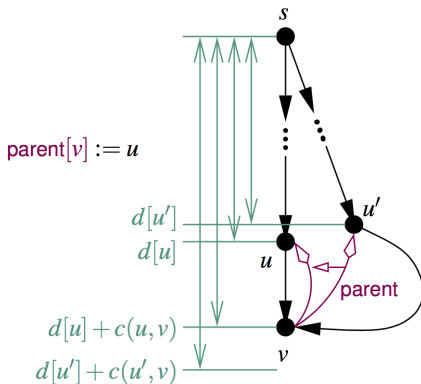
setze

- ▶  $d[v] := d[u] + c(u, v)$   
und
- ▶  $\text{parent}[v] := u$

Invarianten bleiben erhalten!

**Beobachtung:**

$d[v]$  kann sich mehrmals  
ändern!



# Dijkstras Algorithmus: Pseudocode

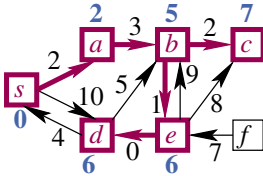
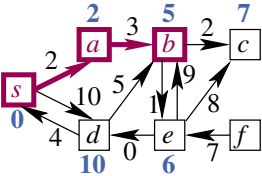
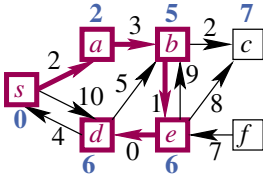
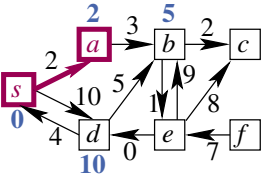
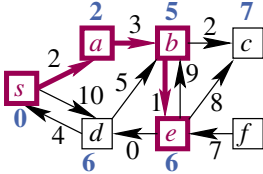
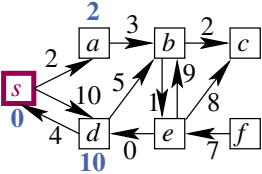
initialize  $d$ , parent

all nodes are non-scanned

**while**  $\exists$  non-scanned node  $u$  with  $d[u] < \infty$   
     $u :=$  non-scanned node  $v$  with minimal  $d[v]$   
    relax all edges  $(u, v)$  out of  $u$   
     $u$  is scanned now

**Behauptung:** Am Ende definiert  $d$  die optimalen Entfernungen  
und parent die zugehörigen Wege

# Beispiel



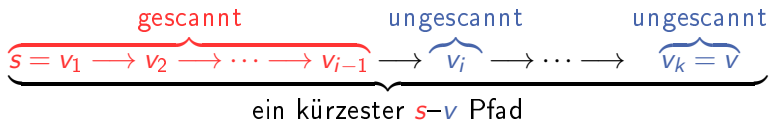
Wir zeigen:  $\forall v \in V$ :

- ▶  $v$  erreichbar  $\implies v$  wird irgendwann gescannt
- ▶  $v$  gescannt  $\implies \mu(v) = d[v]$



$v$  erreichbar  $\implies v$  wird irgendwann gescannt

Annahme:  $v$  ist erreichbar, aber wird nicht gescannt



$\implies v_{i-1}$  wird gescannt

$\implies$  Kante  $v_{i-1} \rightarrow v_i$  wird relaxiert

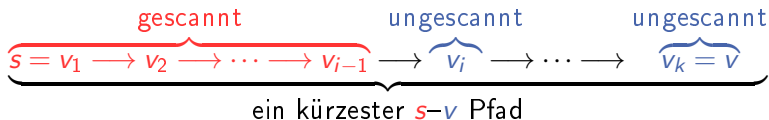
$\implies d[v_i] < \infty$

Widerspruch – nur Knoten  $x$  mit  $d[x] = \infty$  werden nie gescannt

□?

$v$  erreichbar  $\implies v$  wird irgendwann gescannt

Annahme:  $v$  ist erreichbar, aber wird nicht gescannt



$\implies v_{i-1}$  wird gescannt

$\implies$  Kante  $v_{i-1} \rightarrow v_i$  wird relaxiert

$\implies d[v_i] < \infty$

Widerspruch – nur Knoten  $x$  mit  $d[x] = \infty$  werden nie gescannt

Ups: Spezialfall  $i = 1$ ?

Kann auch nicht sein.

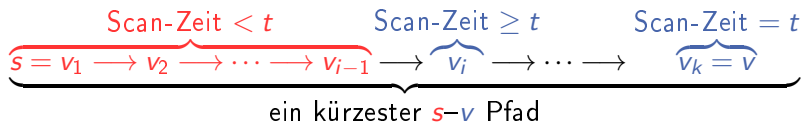
$v_1 = s$  wird nach Initialisierung gescannt. □

$v$  gescannt  $\implies \mu(v) = d[v]$

**Annahme:**  $v$  gescannt und  $\mu(v) < d[v]$

**OBdA:**  $v$  ist der **erste** gescannte Knoten mit  $\mu(v) < d[v]$ .

$t :=$  Scan-Zeit von  $v$



Also gilt zur Zeit  $t$ :

$$\mu(v_{i-1}) = d[v_{i-1}]$$

$v_{i-1} \rightarrow v_i$  wurde relaxiert

$$\implies d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) = \mu(v_i) \leq \mu(v) < d[v]$$

$\implies v_i$  wird vor  $v$  gescannt. **Widerspruch!**

**Wieder:** Spezialfall  $i = 1$  unmöglich. □

# Dijkstra: Implementierung?

initialize  $d$ , parent

all nodes are non-scanned

**while**  $\exists$  non-scanned node  $u$  with  $d[u] < \infty$

$u :=$  non-scanned node  $v$  with minimal  $d[v]$

relax all edges  $(u, v)$  out of  $u$

$u$  is scanned now

Wichtigste Operation: **finde  $u$**

# Prioritätsliste

Wir speichern **ungescannte erreichte Knoten** in **adressierbarer Prioritätsliste  $Q$** .

**Schlüssel** ist  $d[v]$ .

Knoten speichern handles.

oder gleich items

# Implementierung $\approx$ BFS mit PQ statt FIFO

**Function** Dijkstra( $s : \text{NodeId}$ ) :  $\text{NodeArray} \times \text{NodeArray}$   
// returns ( $d, \text{parent}$ )

Initialisierung:

$d = \langle \infty, \dots, \infty \rangle : \text{NodeArray}$  of  $\mathbb{R} \cup \{\infty\}$  // tentative distance from root  
 $\text{parent} = \langle \perp, \dots, \perp \rangle : \text{NodeArray}$  of  $\text{NodeId}$   
 $\text{parent}[s] := s$  // self-loop signals root  
 $Q : \text{NodePQ}$  // unscanned reached nodes  
 $d[s] := 0; Q.\text{insert}(s)$

```

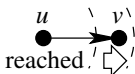
Function Dijkstra( $s$  : NodeId) : NodeArray  $\times$  NodeArray
 $d = \langle \infty, \dots, \infty \rangle$ ; parent[ $s$ ] :=  $s$ ;  $d[s] := 0$ ;  $Q.insert(s)$ 
while  $Q \neq \emptyset$  do
     $u := Q.deleteMin$ 
    // scan  $u$ 
    foreach edge  $e = (u, v) \in E$  do
        if  $d[u] + c(e) < d[v]$  then
             $d[v] := d[u] + c(e)$ 
            parent[ $v$ ] :=  $u$ 
            if  $v \in Q$  then  $Q.decreaseKey(v)$ 
            else  $Q.insert(v)$ 
return ( $d, parent$ )

```

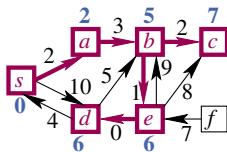
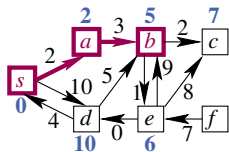
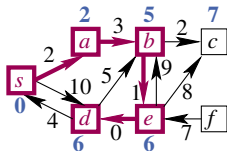
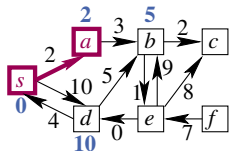
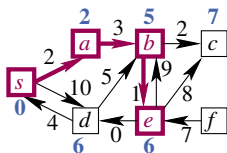
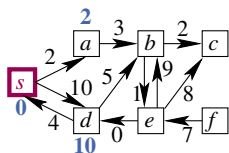


// relax

// update tree



# Beispiel



Operation

insert(s)

deleteMin  $\rightsquigarrow$  (s,0)

relax  $s \xrightarrow{2} a$

relax  $s \xrightarrow{10} d$

deleteMin  $\rightsquigarrow$  (a,2)

relax  $a \xrightarrow{3} b$

deleteMin  $\rightsquigarrow$  (b,5)

relax  $b \xrightarrow{2} c$

relax  $b \xrightarrow{1} e$

deleteMin  $\rightsquigarrow$  (e,6)

relax  $e \xrightarrow{9} b$

relax  $e \xrightarrow{8} c$

relax  $e \xrightarrow{0} d$

deleteMin  $\rightsquigarrow$  (d,6)

relax  $d \xrightarrow{4} s$

relax  $d \xrightarrow{5} b$

deleteMin  $\rightsquigarrow$  (c,7)

Queue

$\langle (s,0) \rangle$

$\langle \rangle$

$\langle (a,2) \rangle$

$\langle (a,2), (d,10) \rangle$

$\langle (d,10) \rangle$

$\langle (b,5), (d,10) \rangle$

$\langle (d,10) \rangle$

$\langle (c,7), (d,10) \rangle$

$\langle (e,6), (c,7), (d,10) \rangle$

$\langle (c,7), (d,10) \rangle$

$\langle (c,7), (d,10) \rangle$

$\langle (d,6), (c,7) \rangle$

$\langle (c,7) \rangle$

$\langle (c,7) \rangle$

$\langle \rangle$



# Dijkstra: Laufzeit

**Function** Dijkstra( $s : \text{NodeId}$ ) :  $\text{NodeArray} \times \text{NodeArray}$

Initialisierung:

$d = \langle \infty, \dots, \infty \rangle$  :  $\text{NodeArray}$  of  $\mathbb{R} \cup \{\infty\}$  //  $O(n)$

$\text{parent} = \langle \perp, \dots, \perp \rangle$  :  $\text{NodeArray}$  of  $\text{NodeId}$  //  $O(n)$

$\text{parent}[s] := s$

$Q : \text{NodePQ}$  // unscanned reached nodes,  $O(n)$

$d[s] := 0$ ;  $Q.\text{insert}(s)$

# Dijkstra: Laufzeit

```
Function Dijkstra( $s$  : NodeId) : NodeArray  $\times$  NodeArray
 $d = \{\infty, \dots, \infty\}$ ; parent[ $s$ ] :=  $s$ ;  $d[s] := 0$ ;  $Q.insert(s)$  //  $O(n)$ 
while  $Q \neq \emptyset$  do
     $u := Q.deleteMin$  //  $\leq n \times$ 
    foreach edge  $e = (u, v) \in E$  do //  $\leq m \times$ 
        if  $d[u] + c(e) < d[v]$  then //  $\leq m \times$ 
             $d[v] := d[u] + c(e)$  //  $\leq m \times$ 
            parent[ $v$ ] :=  $u$  //  $\leq m \times$ 
            if  $v \in Q$  then  $Q.decreaseKey(v)$  //  $\leq m \times$ 
            else  $Q.insert(v)$  //  $\leq n \times$ 
return ( $d, parent$ )
```

# Dijkstra: Laufzeit

```
Function Dijkstra( $s$  : NodeId) : NodeArray  $\times$  NodeArray
 $d = \{\infty, \dots, \infty\}$ ; parent[ $s$ ] :=  $s$ ;  $d[s] := 0$ ;  $Q.insert(s)$  //  $O(n)$ 
while  $Q \neq \emptyset$  do
     $u := Q.deleteMin$  //  $\leq n \times$ 
    foreach edge  $e = (u, v) \in E$  do //  $\leq m \times$ 
        if  $d[u] + c(e) < d[v]$  then //  $\leq m \times$ 
             $d[v] := d[u] + c(e)$  //  $\leq m \times$ 
            parent[ $v$ ] :=  $u$  //  $\leq m \times$ 
            if  $v \in Q$  then  $Q.decreaseKey(v)$  //  $\leq m \times$ 
            else  $Q.insert(v)$  //  $\leq n \times$ 
return ( $d, parent$ )
```

Insgesamt:

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

Dijkstras ursprüngliche Implementierung: „naiv“

- ▶ insert:  $O(1)$   $d[v] := d[u] + c(u, v)$
- ▶ decreaseKey:  $O(1)$   $d[v] := d[u] + c(u, v)$
- ▶ deleteMin:  $O(n)$   $d$  komplett durchsuchen

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

$$T_{\text{Dijkstra59}} = O(m \cdot 1 + n \cdot (n + 1))$$

$$= O(m + n^2)$$

Bessere Implementierung mit **Binary-Heap-Prioritätslisten**:

- ▶ insert:  $O(\log n)$
- ▶ decreaseKey:  $O(\log n)$
- ▶ deleteMin:  $O(\log n)$

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

$$\begin{aligned} T_{\text{DijkstraBHp}} &= O(m \cdot \log n + n \cdot (\log n + \log n)) \\ &= O((m + n) \log n) \end{aligned}$$

(Noch) besser mit **Fibonacci-Heapprioritätslisten**:

- ▶ insert:  $O(1)$
- ▶ decreaseKey:  $O(1)$  (amortisiert)
- ▶ deleteMin:  $O(\log n)$  (amortisiert)

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

$$\begin{aligned} T_{\text{DijkstraFib}} &= O(m \cdot 1 + n \cdot (\log n + 1)) \\ &= O(m + n \log n) \end{aligned}$$

Aber: konstante Faktoren in  $O(\cdot)$  sind hier größer!

# Analyse im Mittel

Modell: Kantengewichte sind „zufällig“ auf die Kanten verteilt  
Dann gilt:

$$E[T_{\text{DijkstraBH}(ea)_p}] = O\left(m + n \log n \log \frac{m}{n}\right)$$

**Beweis:** In Algorithmen II

# Monotone ganzzahlige Prioritätslisten

**Beobachtung:** In Dijkstras Algorithmus steigt das Minimum in der Prioritätsliste monoton.

Das kann man ausnutzen.  $\rightsquigarrow$  **schnellere Algorithmen**  
u.U. bis herunter zu  **$O(m+n)$** .

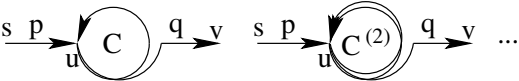
**Details:** in Algorithmen II



# Negative Kosten

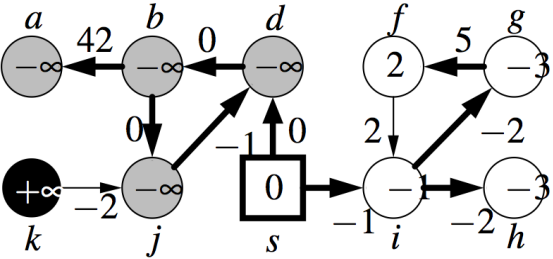
Was machen wir, wenn es Kanten mit negativen Kosten gibt?

Es kann Knoten geben mit  $d[v] = -\infty$



Wie finden wir heraus, welche das sind?

Endlosschleifen vermeiden!



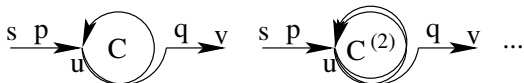
## Zurück zu Basiskonzepten (Abschnitt 10.1 im Buch)

**Lemma:**  $\exists$  kürzester  $s$ - $v$ -Pfad  $P \implies P$  ist OBdA **einfach** (eng. simple)

**Beweisidee:** (Kontraposition)

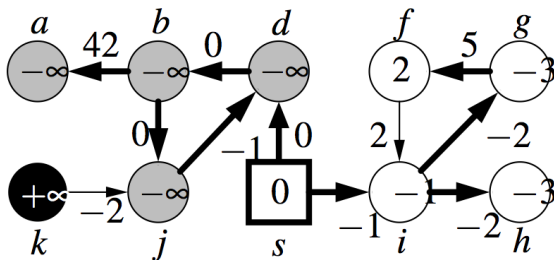
**Fall:**  $v$  über negativen Kreis erreichbar  $\implies$

$\neg \exists$  kürzester  $s$ - $v$ -Pfad (sondern beliebig viele immer kürzere)



**Sonst:** betrachte beliebigen nicht-einfachen  $s$ - $v$ -Pfad.

**Alle Kreise streichen**  $\rightsquigarrow$  einfacher, nicht längerer Pfad. ■



# Mehr Basiskonzepte

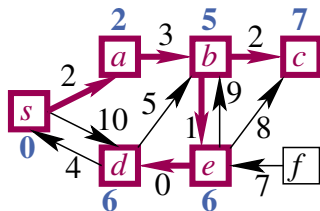
Übung, zeige:

Teilpfade kürzester Pfade sind selbst kürzeste Pfade

$$a-b-c-d \rightsquigarrow a-b, b-c, c-d, a-b-c, b-c-d$$

## Übung: Kürzeste-Wege-Baum

Alle kürzesten Pfade von  $s$  aus zusammen (einer pro Zielknoten) bilden einen Baum, falls es keine negativen Kreise gibt.



# Allgemeines Korrektheitskriterium

Sei  $R = \langle \dots \overbrace{\text{relax}(e_1)}^{t_1} \dots \overbrace{\text{relax}(e_2)}^{t_2} \dots \overbrace{\text{relax}(e_k)}^{t_k} \dots \rangle$

eine Folge von Relaxationsoperationen und

$p = \langle e_1, e_2, \dots, e_k \rangle = \langle s, v_1, v_2, \dots, v_k \rangle$  ein kürzester Weg.

Dann gilt anschließend:  $d[v_k] = \mu(v_k)$

**Beweisskizze:** (Eigentlich Induktion über  $k$ )

$d[s] = \mu(s)$  bei Initialisierung

$d[v_1] = \mu(v_1)$  nach Zeitpunkt  $t_1$

$d[v_2] = \mu(v_2)$  nach Zeitpunkt  $t_2$

...

$d[v_k] = \mu(v_k)$  nach Zeitpunkt  $t_k$

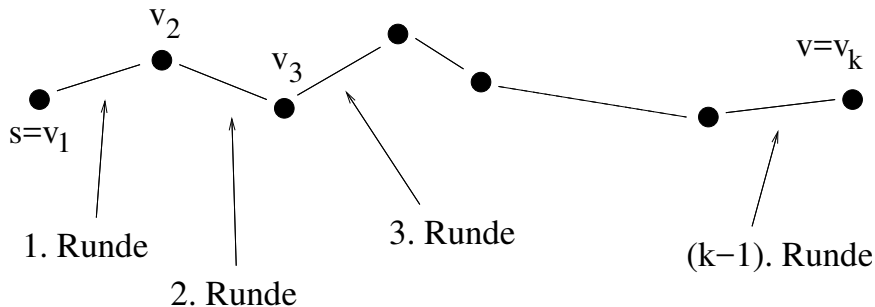


# Algorithmen brutal – Bellman-Ford-Algorithmus für beliebige Kantengewichte

Wir relaxieren alle Kanten (in irgendeiner Reihenfolge)  $n - 1$  mal.

Alle kürzeste Pfade in  $G$  haben höchstens  $n - 1$  Kanten.

⇒ **Jeder** (einfache) kürzeste Pfad ist eine Teilfolge dieser Relaxationen!



# Negative Kreise finden

Nach Ausführung von Bellman-Ford:

$\forall$  negativen Kreise  $C$ :

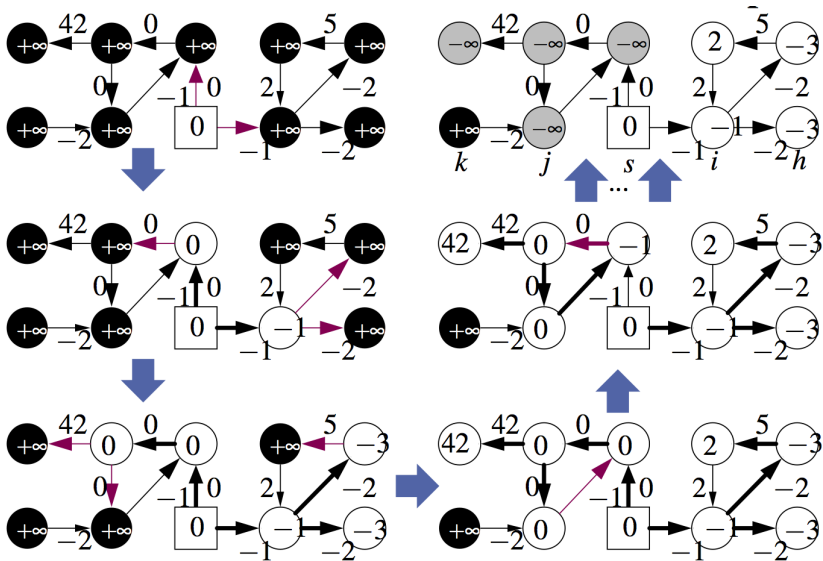
$\exists (u, v) \in C$ :

$$d[u] + c(e) < d[v]$$

**Beweis:** Übung

$v$  und alle von  $v$  **erreichbaren** Knoten  $x$  haben  $\mu(x) = -\infty$

# Beispiel



# Bellman-Ford – Laufzeit

$O(nm)$ , also viel langsamer als Dijkstra!

Es gibt Algorithmenvarianten mit viel besserem **best case**.



# Azyklische Graphen (10.2 im Buch)

## Beobachtungen:

Keine (gerichteten) Kreise  $\implies$  keine negativen Kreise!

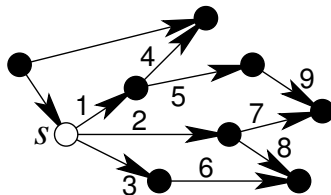
Für jeden (kürzesten) Pfad  $\langle v_1, \dots, v_n \rangle$ :

Die Kanten sind aufsteigend bzgl. jeder **topologischen Sortierung**!

```
initialize  $d$ , parent
```

```
foreach  $v \in V$  in topological order do scan( $v$ )
```

Laufzeit:  $O(m + n)$



## Von überall nach überall

Im Prinzip:  $n \times$  von  $s$  nach überall

**nichtnegative Kantengewichte:** Zeit  $O(n(m + n \log n))$ .

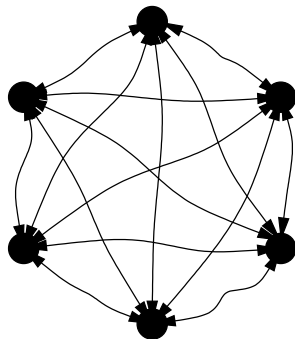
( $n \times$  Dijkstra)

**beliebige Kantengewichte:** Zeit  $O(n^2 m)$ .

( $n \times$  Bellman-Ford)

In Algorithmen II: Zeit  $O(n(m + n \log n))$ .

( $1 \times$  Bellman-Ford +  $n \times$  Dijkstra)



# Kürzeste Wege: Zusammenfassung

- ▶ Einfache, effiziente Algorithmen für **nichtnegative** Kantengewichte und **azyklische** Graphen
- ▶ Optimale Lösungen bei nicht (ganz) trivialen Korrektheitsbeweisen
- ▶ Prioritätslisten sind wichtige Datenstruktur

# Mehr zu kürzesten Wegen

Viele Arbeiten zu besseren Prioritätslisten

$\rightsquigarrow O(m + n \log \log n)$  [Thorup 2004]

- ▶ Mehrere **Zielfunktionen** abwägen
- ▶ Mehrere **Ziele** in beliebiger Reihenfolge anfahren  
siehe auch **Optimierungskapitel**
- ▶ Mehrere **disjunkte Wege**

Fast alles schwierig (NP-schwer)

# Exkurs: Routing in Straßennetzwerken

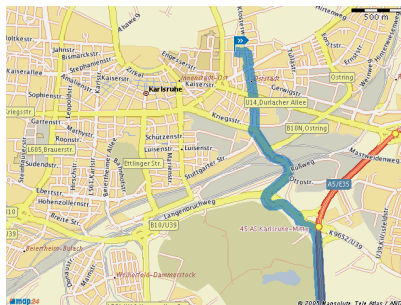
Start: Beobachtungen zu Eigenschaften von Straßennetzwerken

- ▶ **groß**, z.B.  $n = 18\,000\,000$  Knoten für Westeuropa
- ▶ **dünn besetzt**, z.B.,  $m = \Theta(n)$  Kanten
- ▶ beinahe **planar**, d.h., wenige Kanten kreuzen sich (Brücken)
- ▶ inhärente **Hierarchie**, schnellste Pfade benutzen **wichtige** Straßen

# Straßennetzwerke

## Gängige **Anwendungen**:

- ▶ Routenplanungssysteme im Internet, (z. B. `maps.google.com`)
- ▶ Fahrzeugnavigationssysteme
- ▶ Logistik
- ▶ Verkehrssimulationen

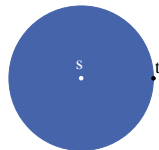


## Distanz zu einem Zielknoten $t$

Was machen wir, wenn wir nur die Distanz von  $s$  zu einem bestimmten Knoten  $t$  wissen wollen?

**Trick 0:** Dijkstra hört auf, wenn  $t$  aus  $Q$  entfernt wird.

Spart "im Durchschnitt" Hälfte der Scans.



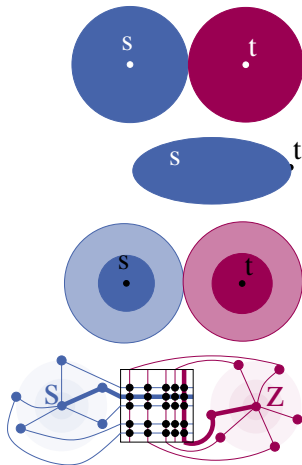
Frage: Wieviel spart es (meist) beim Europa-Navi?



# Ideen für Routenplanung

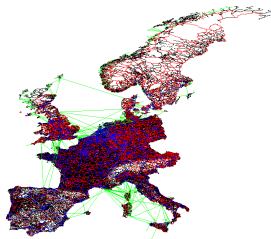
mehr in Algorithmen II, Algorithm Engineering

- ▶ Vorwärts- + Rückwärtsuche
- ▶ Zielgerichtete Suche
- ▶ Hierarchien ausnutzen
- ▶ Teilabschnitte tabellieren



Meist zentrale Idee: **Vorbereitung** amortisiert über viele Anfragen



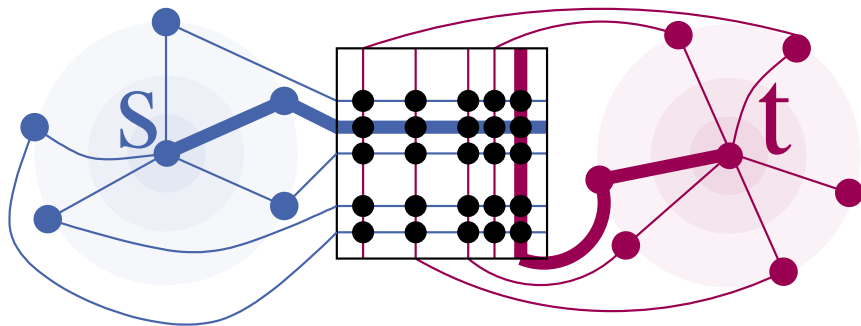


Wir konzentrieren uns auf  
Straßennetzwerke.

- ▶ mehrere **nützliche Eigenschaften**, die sich ausnutzen lassen
  - ▶ viele **reale** Anwendungen
- 
- ▶ einige Techniken: anwendbar für **öffentliche Verkehrsmittel**
  - ▶ die meisten Techniken: **unklar**, wie nützlich sie für weitere Graphtypen sind

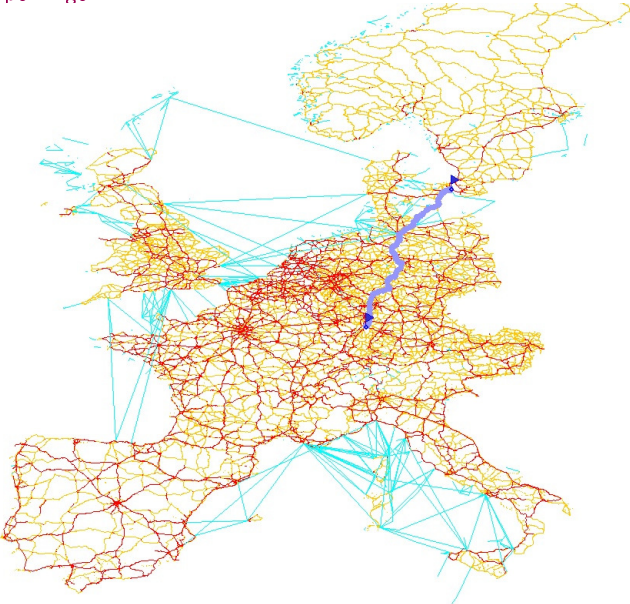
# Approach: Transit-Node Routing

[Bast, Funke, Matijevic, Sanders, Schultes]



# Beispiel

Karlsruhe → Copenhagen



# Beispiel

Karlsruhe → Berlin



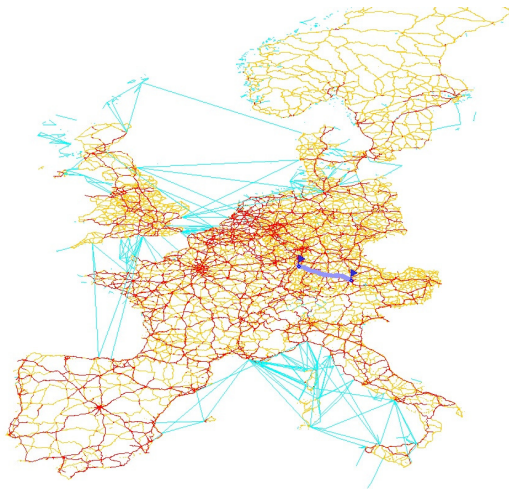
# Beispiel

Karlsruhe → Vienna



# Beispiel

Karlsruhe → Munich



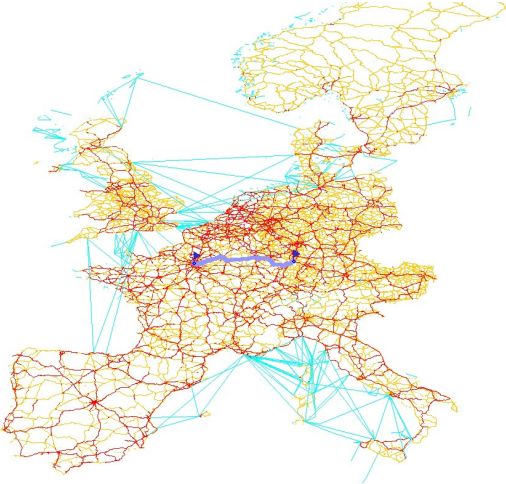
# Beispiel

Karlsruhe → Rome



# Beispiel

Karlsruhe → Paris





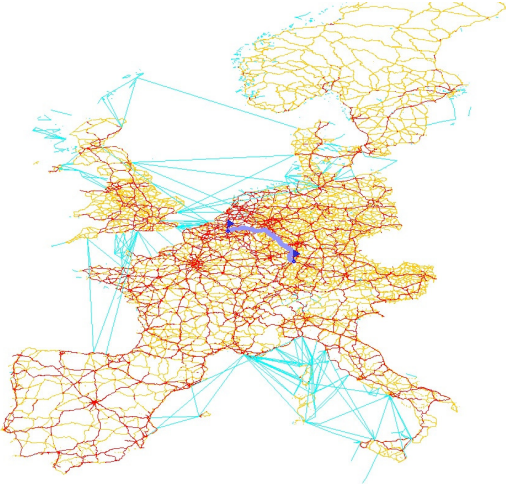
# Beispiel

Karlsruhe → London



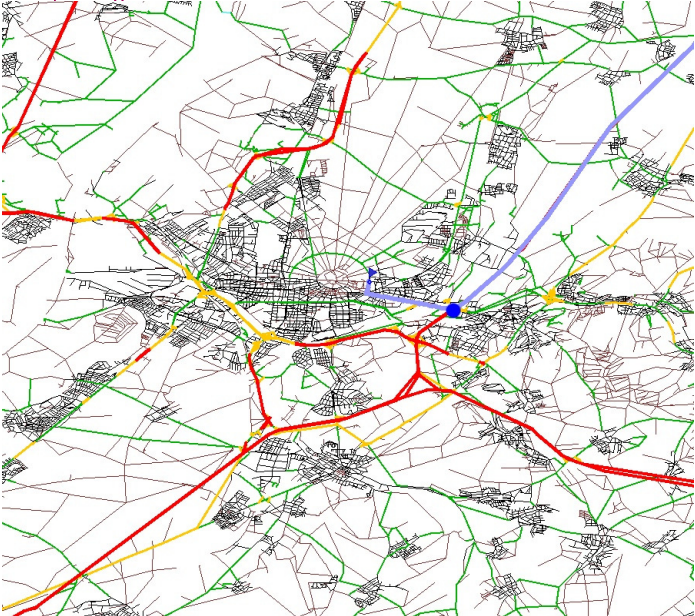
# Beispiel

Karlsruhe → Brussels



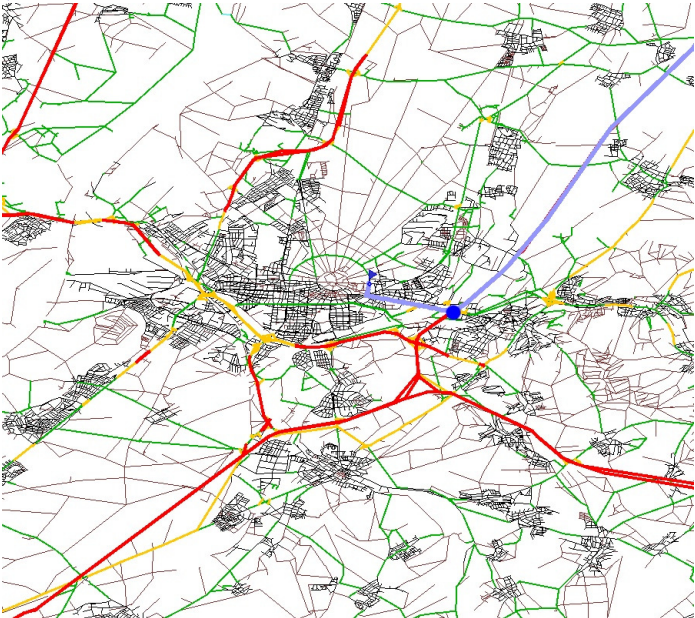
# Beispiel

Karlsruhe → Copenhagen



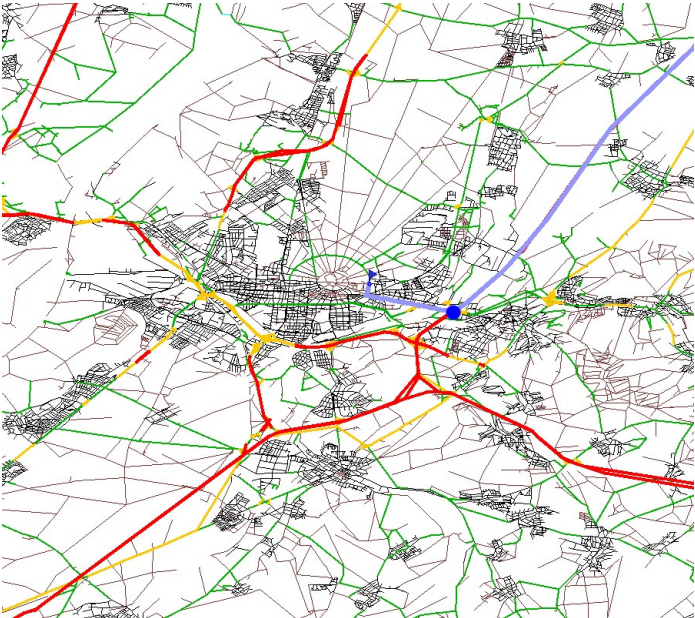
# Beispiel

Karlsruhe → Berlin



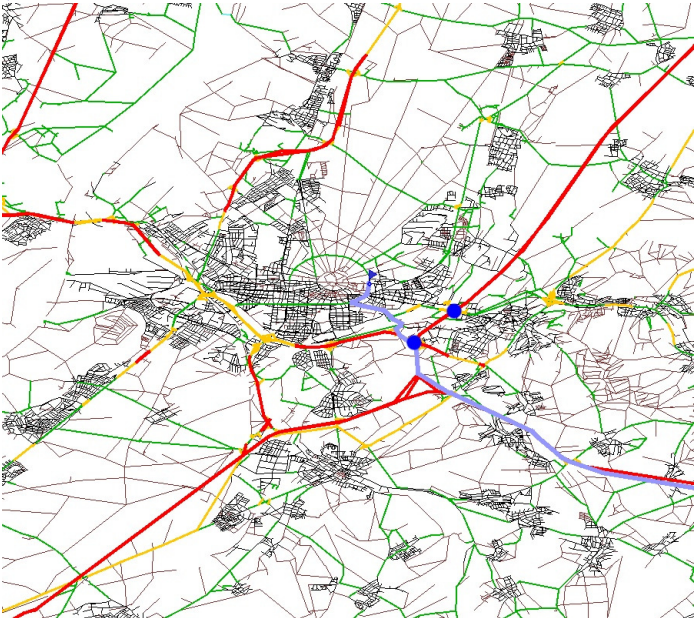
# Beispiel

Karlsruhe → Vienna



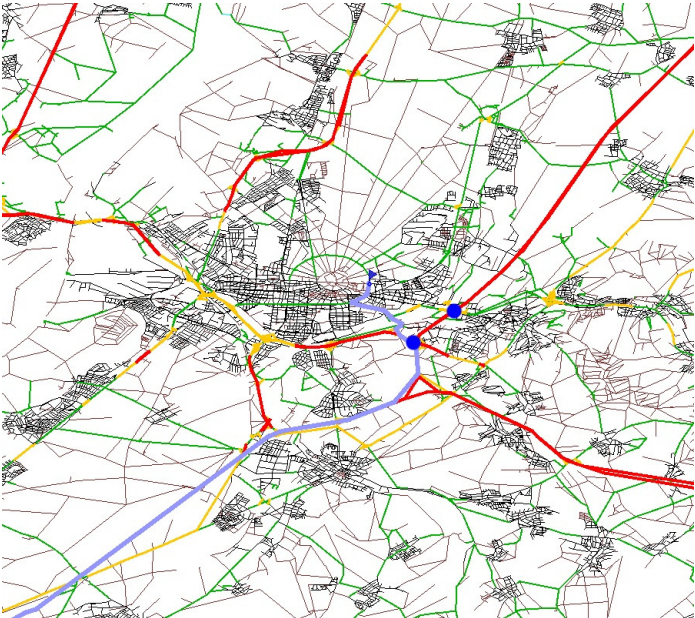
# Beispiel

Karlsruhe → Munich



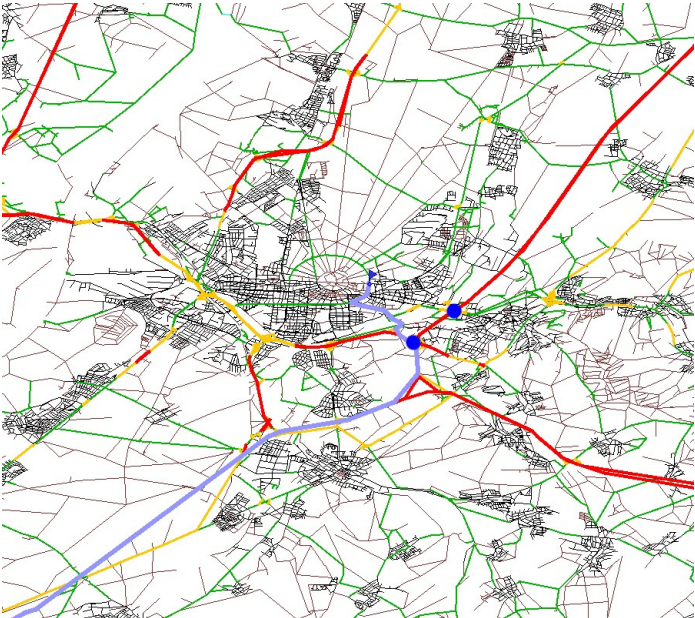
# Beispiel

Karlsruhe → Rome



# Beispiel

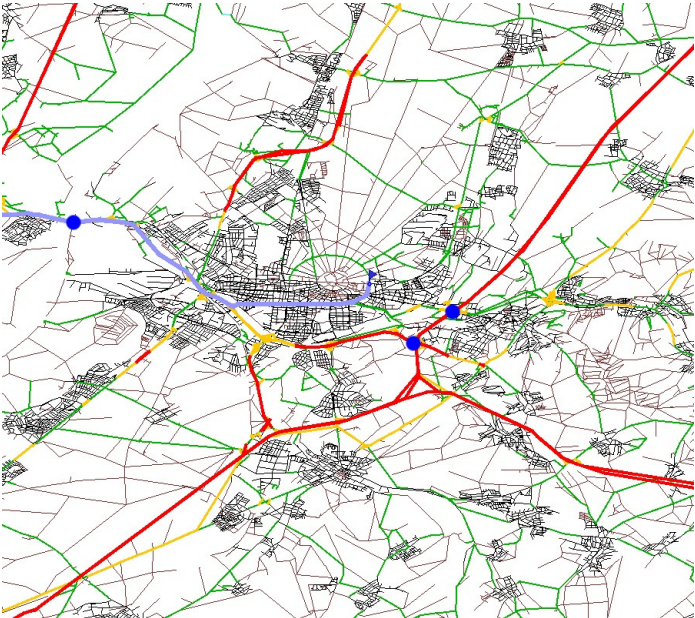
Karlsruhe → Paris





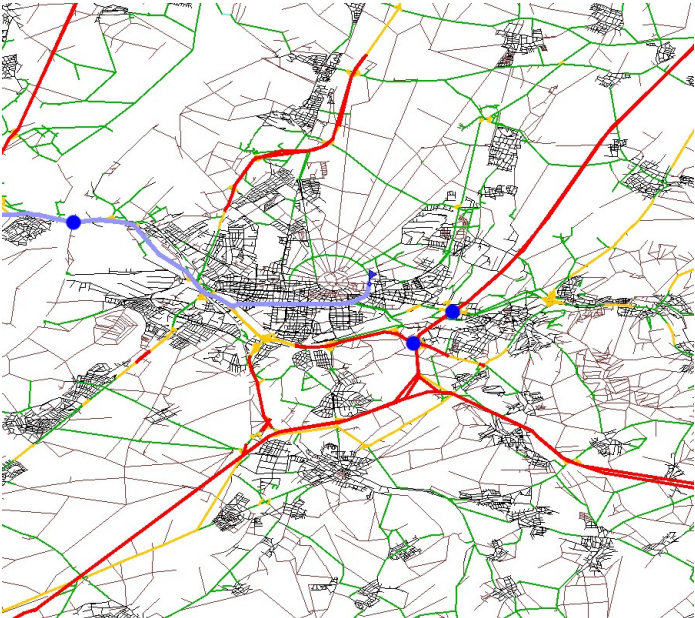
# Beispiel

Karlsruhe → London



# Beispiel

Karlsruhe → Brussels



# Erste Beobachtung

## Lange Strecken benutzen

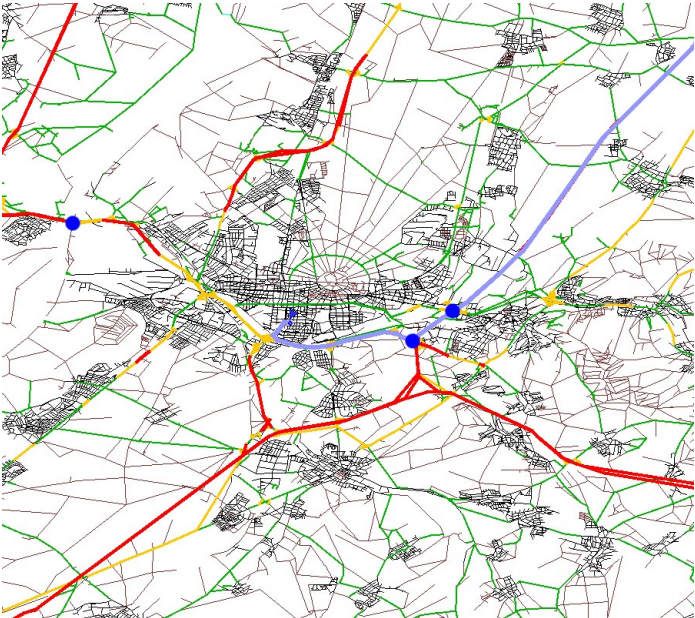
nur wenige 'wichtige' Zugänge zum Fernverkehrsnetzwerk,  
sog. **access points**

( $\rightsquigarrow$  wir können alle Zugangspunkte vorberechnen)

[in Europa: etwa 10 Zugangspunkte pro Knoten im Mittel]

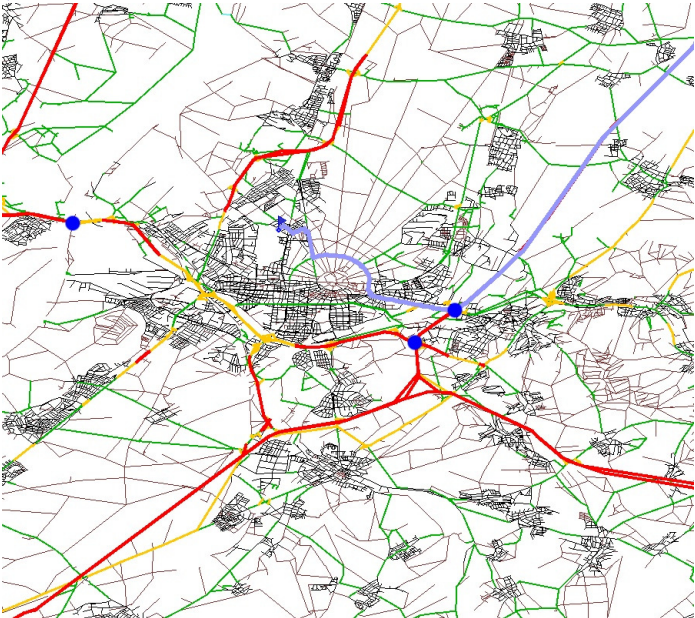
# Beispiel

Karlsruhe → Berlin



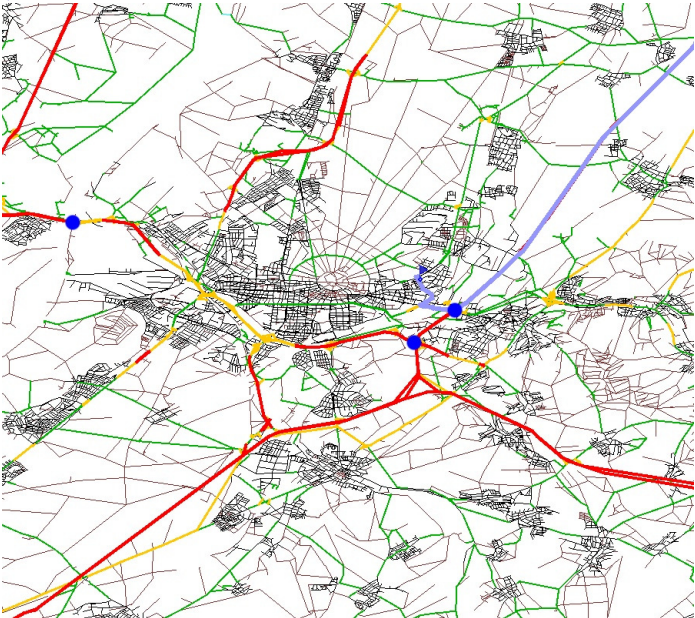
# Beispiel

Karlsruhe → Berlin



# Beispiel

Karlsruhe → Berlin



## Zweite Beobachtung

Jeder Zugangspunkt ist für mehrere Knoten relevant.  $\rightsquigarrow$

**Gesamtmenge** aller Zugangspunkte ist **klein**,  
**Transitknotenmenge**

( $\rightsquigarrow$  wir können alle Abstände zwischen allen Transitknoten speichern)

[in Europa:  $\approx$  10 000 Transitknoten]

# Transit-Node Routing

## Preprocessing:

- ▶ Identifiziere **Transitknoten**  $\mathcal{T} \subseteq V$
- ▶ Berechne  $|\mathcal{T}| \times |\mathcal{T}|$  **Abstandstabelle**
- ▶ Für jeden Knoten: identifiziere **Zugangsknoten** (Abbildung  $A : V \rightarrow 2^{\mathcal{T}}$ ),  
speichere **Abstände**

**Query** (geg. Start  $s$  und Ziel  $t$ ): berechne

$$d_{\text{top}}(s, t) := \min \{d(s, u) + d(u, v) + d(v, t) : u \in A(s), v \in A(t)\}$$



# Transit-Node Routing

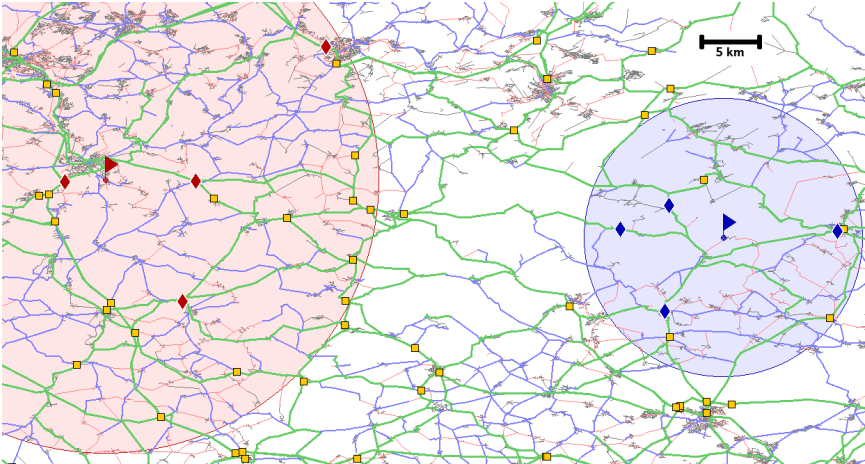
## Lokalitätsfilter:

lokale Fälle ausfiltern ( $\rightsquigarrow$  Spezialbehandlung)

$L : V \times V \rightarrow \{\text{true}, \text{false}\}$

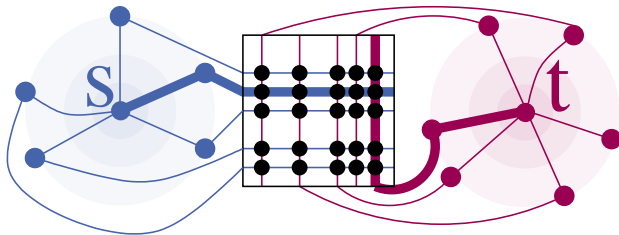
$\neg L(s, t)$  impliziert  $d(s, t) = d_{\text{top}}(s, t)$

# Beispiel: Transitknoten



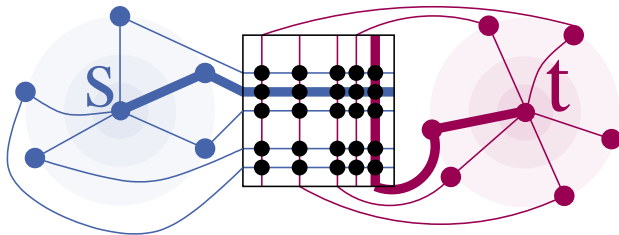
# Experimente

- ▶ sehr schnelle Anfragen (queries)  
( $4 \mu\text{s}$ ,  $> 1\,000\,000$  mal schneller als Dijkstra)
- ▶ **Gewinner** der 9. DIMACS Implementation Challenge
- ▶ erträglich: Vorberechnungszeiten ( $1:15 \text{ h}$ ) und Speicherbedarf ( $247 \text{ bytes/Knoten}$ )



# Experimente

- ▶ sehr schnelle Anfragen (queries)  
( $4 \mu\text{s}$ ,  $> 1\,000\,000$  mal schneller als Dijkstra)
- ▶ **Gewinner** der 9. DIMACS Implementation Challenge
- ▶ erträglich: Vorberechnungszeiten ( $1:15 \text{ h}$ ) und Speicherbedarf ( $247 \text{ bytes/Knoten}$ )
- ▶ Neuere Werte:  $< 2 \mu\text{s}$ , 5 Minuten PP, 150 Bytes/Knoten



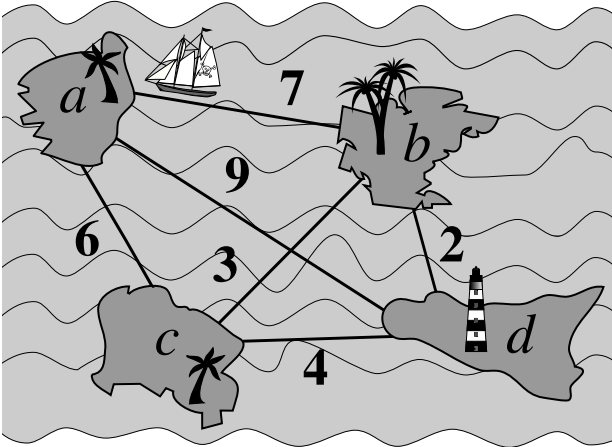
# Offene Fragen

- ▶ Wie bestimmt man die **Transitknoten**?
- ▶ Wie bestimmt man die **Zugangsknoten** effizient?
- ▶ Wie bestimmt man die **Lokalitätsfilter**?
- ▶ Wie handhabt man **lokale Anfragen**?

## Antwort:

- ▶ Andere Routenplanungstechniken benutzen!

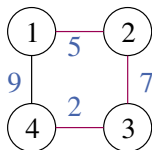
# Kap. 11: Minimale Spann



# Minimale Spann bäume (MST)

## Eingabe:

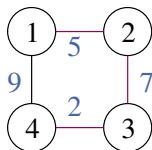
- ▶ ungerichteter (zusammenhängender) Graph  $G = (V, E)$ .
- ▶ Knoten  $V$ ,  $n = |V|$ , z. B.  $V = \{1, \dots, n\}$
- ▶ Kanten  $e \in E \subseteq V \times V$ ,  $m = |E|$
- ▶ Kantengewichte  $c(e) \in \mathbb{R}_+$ .



# Minimale Spannbäume (MST)

## Eingabe:

- ▶ ungerichteter (zusammenhängender) Graph  $G = (V, E)$ .
- ▶ Knoten  $V$ ,  $n = |V|$ , z. B.  $V = \{1, \dots, n\}$
- ▶ Kanten  $e \in E \subseteq V \times V$ ,  $m = |E|$
- ▶ Kantengewichte  $c(e) \in \mathbb{R}_+$ .



## Aufgabe:

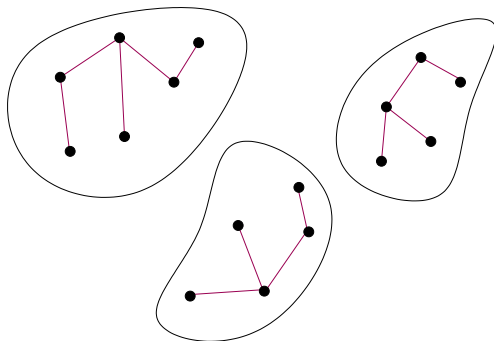
Finde Baum  $(V, T)$  mit **minimalem** Gewicht  $\sum_{e \in T} c(e)$ ,  
der alle Knoten verbindet.



# Minimale aufspannende Wälder (MSF)

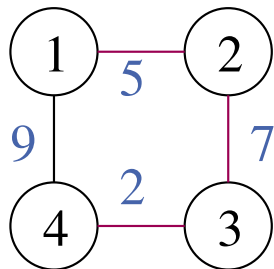
Falls  $G$  nicht zusammenhängend ist,  
finde **minimalen aufspannenden Wald**  $T$ , der alle  
Zusammenhangskomponenten von  $G$  aufspannt.

MST-Algorithmen lassen sich leicht zu MSF-Algorithmen  
verallgemeinern.



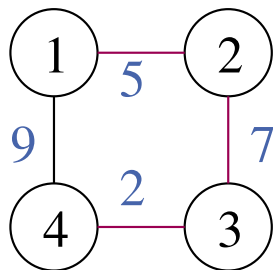
# Anwendungen

- ▶ Netzwerk-Entwurf
- ▶ **Bottleneck-Shortest-Paths:**  
Suche  $s-t$ -Pfad, dessen max. Kantengewicht minimal ist.  
Dies ist der Pfad im MST!



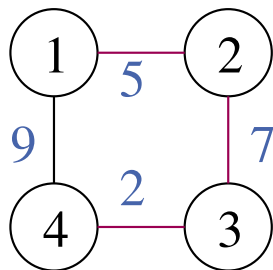
# Anwendungen

- ▶ Netzwerk-Entwurf
- ▶ **Bottleneck-Shortest-Paths:**  
Suche  $s$ - $t$ -Pfad, dessen max. Kantengewicht minimal ist.  
Dies ist der Pfad im MST!
- ▶ **Clustering:** Lass schwere MST-Kanten weg. Teilbäume definieren Cluster.  
Konkret z. B. **Bildsegmentierung**
- ▶ **Näherungslösungen** für schwere Probleme, z. B. **Handlungsreisenden-, Steinerbaumproblem.**  
Siehe Buch, VL G. theoretischer Informatik, Algorithmen II.



# Anwendungen

- ▶ Netzwerk-Entwurf
- ▶ **Bottleneck-Shortest-Paths:**  
Suche  $s-t$ -Pfad, dessen max. Kantengewicht minimal ist.  
Dies ist der Pfad im MST!
- ▶ **Clustering:** Lass schwere MST-Kanten weg. Teilbäume definieren Cluster.  
Konkret z. B. **Bildsegmentierung**
- ▶ **Näherungslösungen** für schwere Probleme, z. B. **Handlungsreisenden-, Steinerbaumproblem.**  
Siehe Buch, VL G. theoretischer Informatik, Algorithmen II.
- ▶ Irrgärten (Beispiel von Wikipedia)



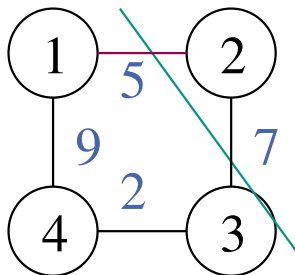
# MST-Kanten auswählen und verwerfen

## Die Schnitteigenschaft (Cut Property)

Für beliebige Teilmenge  $S \subset V$  betrachte die Schnittkanten

$$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$$

Die **leichteste** Kante in  $C$  kann in einem MST verwendet werden.



# MST-Kanten auswählen und verwerfen

## Die Schnitteigenschaft (Cut Property)

Für beliebige Teilmenge  $S \subset V$  betrachte die Schnittkanten

$$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$$

Die **leichteste** Kante in  $C$  kann in einem MST verwendet werden.

**Beweis:** Betrachte MST  $T'$ .

**Fall  $e \in T'$ :** Beweis fertig.

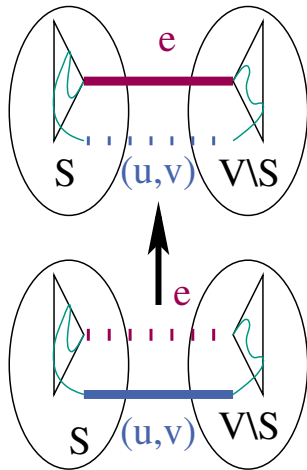
**Sonst:**  $T' \cup \{e\}$  enthält **Kreis  $K$** .

Betrachte eine Kante

$$\{u, v\} \in (C \cap K) \setminus \{e\}.$$

Dann ist  $T = T' \setminus \{\{u, v\}\} \cup \{e\}$  ein Spannbaum, der nicht schwerer ist.

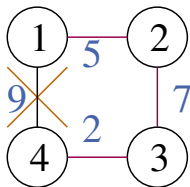
Denn:  $c(e) \leq c(\{u, v\})$



# MST-Kanten auswählen und verwerfen

Die Kreiseigenschaft (Cycle Property)

Die **schwerste** Kante auf einem Kreis wird nicht für einen MST benötigt.



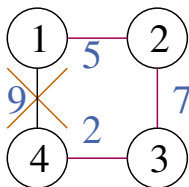
# MST-Kanten auswählen und verwerfen

## Die Kreiseigenschaft (Cycle Property)

Die **schwerste** Kante auf einem Kreis wird nicht für einen MST benötigt.

### Beweis.

Angenommen, MST  $T'$  benutzt die schwerste Kante  $e'$  auf Kreis  $C$ .  
Dann ex.  $e \in C \setminus T'$ , so dass  $T = T' \setminus \{e'\} \cup \{e\}$  auch ein MST ist.  
(Es gilt  $c(e) \leq c(e')$ .)





# Der Jarník-Prim-Algorithmus

[Jarník 1930, Prim 1957]

Idee: Lasse einen Baum wachsen

$T := \emptyset$

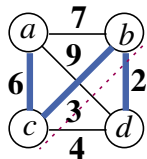
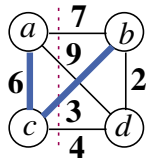
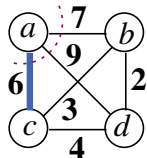
$S := \{s\}$  for arbitrary start node  $s$

repeat  $n - 1$  times

find  $(u, v)$  fulfilling the cut property for  $S$

$S := S \cup \{v\}$

$T := T \cup \{(u, v)\}$



**Function** jpMST : Set of Edge // weitgehend analog zu Dijkstra

pick any  $s \in V$

$d = \{\infty, \dots, \infty\}$ ; parent[s]:= s;  $d[s] := 0$ ; Q.insert(s)

**while**  $Q \neq \emptyset$  **do**

$u := Q.deleteMin$

$d[u] := 0$

    // scan  $u$

**foreach** edge  $e = (u, v) \in E$  **do**

**if**  $c(e) < d[v]$  **then**

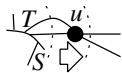
$d[v] := c(e)$

            parent[v] :=  $u$

**if**  $v \in Q$  **then** Q.decreaseKey(v)

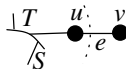
**else** Q.insert(v)

**return**  $\{(v, parent[v]) : v \in V \setminus \{s\}\}$



// relax

// update tree



▶  $d[u] = 0 \Leftrightarrow u \in S$

▶  $\Rightarrow$  am Ende jeder **while**-Iteration: endliches  $d[v] > 0$  speichert  
**Gewicht der leichtesten Kante von  $v \notin S$  über den Schnitt**

# Analyse

Praktisch identisch zu Dijkstra

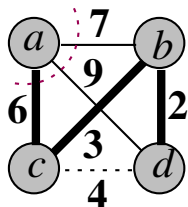
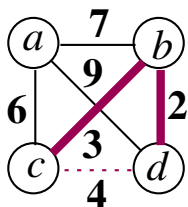
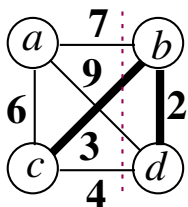
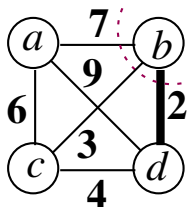
- ▶  $O(m+n)$  Zeit ausserhalb der PQ
- ▶  $n \times$  deleteMin (Zeit  $O(n \log n)$ )
- ▶  $O(m) \times$  decreaseKey
- ↪  $O((m+n) \log n)$  mit binären Heaps
- ↪  $O(m+n \log n)$  mit Fibonacci Heaps

Wichtigster Unterschied: monotone PQs reichen nicht

Warum?

# Kruskals Algorithmus

```
 $T := \emptyset$  // subforest of the MST  
foreach  $(u, v) \in E$  in ascending order of weight do  
  if  $u$  and  $v$  are in different subtrees of  $(V, T)$  then  
     $T := T \cup \{(u, v)\}$  // Join two subtrees  
return  $T$ 
```



# Kruskals Algorithmus – Korrektheit

```
 $T := \emptyset$  // subforest of the MST
foreach  $(u, v) \in E$  in ascending order of weight do
    if  $u$  and  $v$  are in different subtrees of  $(V, T)$  then
         $T := T \cup \{(u, v)\}$  // Join two subtrees
return  $T$ 
```

**Fall  $u, v$  in verschiedenen Teilbäumen:** benutze Schnitteigenschaft  
 $\implies (u, v)$  ist **leichteste** Kante im  $\text{cut}(\text{Komponente}(u), V \setminus \text{Komponente}(u))$   
 $\implies (u, v) \in \text{MST}$

**Sonst:** benutze Kreiseigenschaft  
 $\implies (u, v)$  ist **schwerste** Kante im **Kreis**  $\langle u, v, v-u\text{-Pfad in } T \rangle$   
 $\implies (u, v) \notin \text{MST}$  ■

# Union-Find Datenstruktur

⇒ Wir brauchen eine effiziente Methode für  
“Verbindet die Kante verschiedene Teilbäume?”

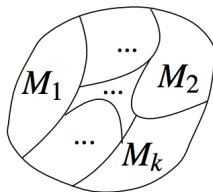
# Union-Find Datenstruktur

⇒ Wir brauchen eine effiziente Methode für  
“Verbindet die Kante verschiedene Teilbäume?”

Verwalte **Partition** der Menge  $1..n$ ,  
d. h., Mengen (Blöcke)  $M_1, \dots, M_k$  mit

$$M_1 \cup \dots \cup M_k = 1..n,$$

$$\forall i \neq j : M_i \cap M_j = \emptyset$$



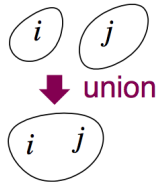
**Class** UnionFind( $n : \mathbb{N}$ )

**Procedure** union( $i, j : 1..n$ )

join the blocks containing  $i$  and  $j$   
to a single block.

**Function** find( $i : 1..n$ ) :  $1..n$

**return** a unique identifier  
for the block containing  $i$ .



# Union-Find Datenstruktur – Erste Version

**Class** UnionFind( $n : \mathbb{N}$ )

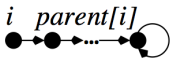
parent= $\langle 1, 2, \dots, n \rangle$  : **Array**  $[1..n]$  **of**  $1..n$

**invariant** parent-refs lead to unique **Partition-Reps**

**Function** find( $i : 1..n$ ) :  $1..n$

**if** parent[ $i$ ] =  $i$  **then return**  $i$

**else return** find(parent[ $i$ ])





# Union-Find Datenstruktur – Erste Version

**Class** UnionFind( $n : \mathbb{N}$ )

parent =  $\langle 1, 2, \dots, n \rangle$  : **Array** [1..n] of 1..n

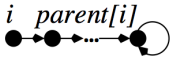
**invariant** parent-refs lead to unique **Partition-Reps**



**Function** find( $i : 1..n$ ) : 1..n

**if** parent[ $i$ ] =  $i$  **then return**  $i$

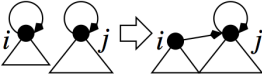
**else return** find(parent[ $i$ ])



**Procedure** link( $i, j : 1..n$ )

**assert**  $i$  and  $j$  are representatives of different blocks

parent[ $i$ ] :=  $j$



**Procedure** union( $i, j : 1..n$ )

**if** find( $i$ )  $\neq$  find( $j$ ) **then** link(find( $i$ ), find( $j$ ))

# Union-Find Datenstruktur – Erste Version

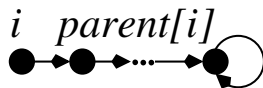
## Analyse

+: **union** braucht konstante Zeit

–: **find** braucht Zeit  $\Theta(n)$  im schlechtesten Fall!

Zu langsam!

Idee: **find-Pfade kurz halten**



# Pfadkompression

**Class** UnionFind( $n : \mathbb{N}$ )

parent =  $\langle 1, 2, \dots, n \rangle$  : **Array** [1..n] of 1..n

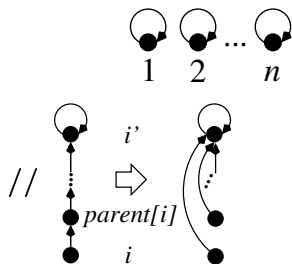
**Function** find( $i : 1..n$ ) : 1..n

**if** parent[ $i$ ] =  $i$  **then return**  $i$

**else**  $i' :=$  find(parent[ $i$ ])

    parent[ $i$ ] :=  $i'$

**return**  $i'$



# Union by Rank

**Class** UnionFind( $n : \mathbb{N}$ )

parent= $\langle 1, 2, \dots, n \rangle$  : **Array** [1.. $n$ ] of 1.. $n$

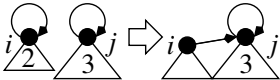
rank= $\langle 0, \dots, 0 \rangle$  : **Array** [1.. $n$ ] of 0..log  $n$



**Procedure** link( $i, j : 1..n$ )

**assert**  $i$  and  $j$  are representatives of different blocks

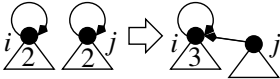
**if** rank[ $i$ ] < rank[ $j$ ] **then** parent[ $i$ ] :=  $j$



**else**

parent[ $j$ ] :=  $i$

**if** rank[ $i$ ] = rank[ $j$ ] **then** rank[ $i$ ]++

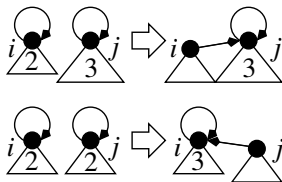


# Analyse – nur Union by rank

**invariant** Der Pfad zum Repr.  $x$  hat Länge höchstens  $\text{rank}[x]$

**invariant**  $x$  ist Repr.  $\Rightarrow x$ 's Menge hat Größe mindestens  $2^{\text{rank}[x]}$

**Korollar:** find braucht Zeit  $O(\log n)$



# Analyse – nur Pfadkompression

**Satz:** find braucht Zeit  $O(\log n)$  (amortisiert)

**Beweis:** im Buch

# Analyse – Pfadkompression + Union by rank

**Satz:**  $m \times$  find +  $n \times$  link brauchen Zeit  $O(m\alpha_T(m, n))$  mit

$$\alpha_T(m, n) = \min \{i \geq 1 : A(i, \lceil m/n \rceil) \geq \log n\}$$

und

$$\begin{aligned} A(1, j) &= 2^j && \text{for } j \geq 1, \\ A(i, 1) &= A(i-1, 2) && \text{for } i \geq 2, \\ A(i, j) &= A(i-1, A(i, j-1)) && \text{for } i \geq 2 \text{ and } j \geq 2. \end{aligned}$$

**Beweis:** [Tarjan 1975, Seidel Sharir 2005]

$A$  ist die **Ackermannfunktion** und  $\alpha_T$  die **inverse Ackermannfunktion**.  
 $\alpha_T(m, n) = \omega(1)$  aber  $\leq 5$  für alle **physikalisch denkbaren**  $n, m$ .

## Ackermannfunktion – Beispiele

$$A(1, j) = 2^j \quad \text{for } j \geq 1,$$

$$A(i, 1) = A(i-1, 2) \quad \text{for } i \geq 2,$$

$$A(i, j) = A(i-1, A(i, j-1)) \quad \text{for } i \geq 2 \text{ and } j \geq 2.$$

---

$$A(2, 1) = A(1, 2) = 2^2$$

$$A(2, 2) = A(1, A(2, 1)) = 2^{2^2}$$

$$A(2, 3) = A(1, A(2, 2)) = 2^{2^{2^2}}$$

$$A(2, 4) = A(1, A(2, 3)) = 2^{2^{2^{2^2}}}$$

$$A(3, 1) = A(2, 2) = 2^{2^2}$$

$$A(3, 2) = A(2, A(3, 1)) = A(2, 16) = ???$$

$$A(4, 1) = A(3, 2) = ???$$



# Kruskal mit Union-Find

Sei  $V = 1..n$

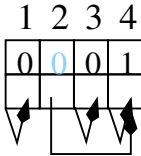
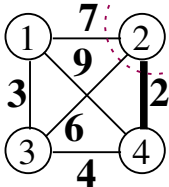
```
Tc : UnionFind(n) // encodes components of forest T
foreach (u, v) ∈ E in ascending order of weight do // sort
  if Tc.find(u) ≠ Tc.find(v) then
    output {u, v}
    Tc.union(u, v) // link reicht auch
```

Zeit  $O(m \log m)$ . Schneller für ganzzahlige Gewichte.

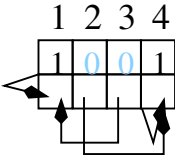
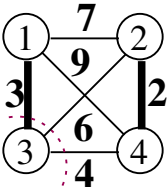
Graphrepräsentation: **Kantenliste**

Bäume im MSF  $\leftrightarrow$  Blöcke in Partition  $\rightarrow$  Wurzelbäume,  
aber mit **anderer Struktur** als die Bäume im MSF!

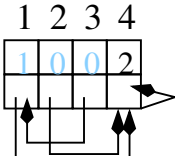
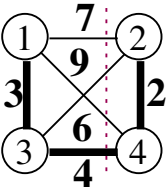
# Beispiel



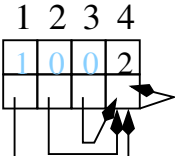
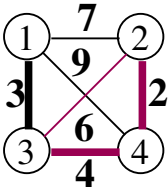
link



link



link



compress

# Vergleich Jarník-Prim $\leftrightarrow$ Kruskal

## Pro Jarník-Prim

- ▶ **Asymptotisch** gut für alle  $m, n$
- ▶ Sehr schnell für  $m \gg n$

## Pro Kruskal

- ▶ Gut für  $m = O(n)$
- ▶ Braucht nur **Kantenliste**
- ▶ Profitiert von schnellen Sortierern (**ganzzahlig, parallel, ...**)
- ▶ **Verfeinerungen** auch gut für große  $m/n$

# Mehr MST-Algorithmen

- ▶ Zeit  $O(m \log n)$  [Boruvka 1926]  
Zutat vieler fortgeschrittener Algorithmen
- ▶ Erwartete Zeit  $O(m)$  [Karger Klein Tarjan 1995],  
parallelisierbar, externalisierbar
- ▶ Det. Zeit  $O(m\alpha_T(m, n))$  [Chazelle 2000]
- ▶ “optimaler” det. Algorithmus [Pettie, Ramachandran 2000]
- ▶ “Invented at ITI”:  
Praktikabler **externer** Algorithmus [Sanders Schultes Sibeyn 2004]  
Verbesserung von Kruskal (parallelisierbar, weniger Sortieraufwand).  
[Osipov Sanders Singler 2009]

# Zusammenfassung

- ▶ **Schnitt-** und **Kreiseigenschaft** als Basis für abstrakte Algorithmen.  
Entwurfsprinzip: benutze **abstrakte Problemeigenschaften**.
- ▶ Beweise mittels **Austauschargumenten**
- ▶ Implementierung braucht effiziente **Datenstrukturen**.  
Auch ein Entwurfsprinzip. . .

# Zusammenfassung

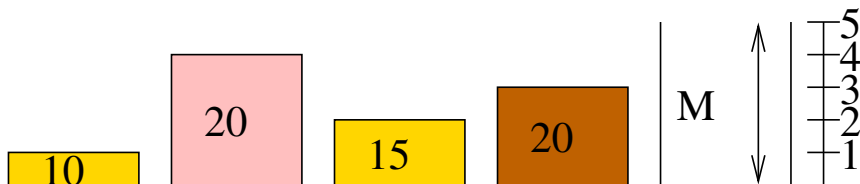
- ▶ **Schnitt-** und **Kreiseigenschaft** als Basis für abstrakte Algorithmen.  
Entwurfsprinzip: benutze **abstrakte Problemeigenschaften**.
- ▶ Beweise mittels **Austauschargumenten**
- ▶ Implementierung braucht effiziente **Datenstrukturen**.  
Auch ein Entwurfsprinzip. . .
- ▶ Dijkstra  $\approx$  JP.  
Noch ein Entwurfsprinzip:  
**Greedy**-Algorithmus effizient implementiert mittels **Prioritätsliste**
- ▶ **Union-Find**: effiziente Verwaltung von Partitionen mittels **Pfadkompression** und **Union-by-rank**.  
Beispiel für **einfache** Algorithmen mit **nichttrivialer** Analyse

## Kap. 12: Generische Optimierungsansätze

- ▶ Black-Box-Löser
- ▶ Greedy
- ▶ Dynamische Programmierung
- ▶ Systematische Suche
- ▶ Lokale Suche
- ▶ Evolutionäre Algorithmen



## Durchgehendes Beispiel: Rucksackproblem



- ▶  $n$  Gegenstände mit Gewicht  $w_i \in \mathbb{N}$  und Profit  $p_i$
- ▶ Wähle eine Teilmenge  $x$  von Gegenständen derart, dass  $\sum_{i \in x} w_i \leq M$  und
- ▶ maximiere den Profit  $\sum_{i \in x} p_i$



# Allgemein: Maximierungsproblem $(\mathcal{L}, f)$

- ▶  $\mathcal{L} \subseteq \mathcal{U}$ : zulässige Lösungen
- ▶  $f: \mathcal{L} \rightarrow \mathbb{R}$  Zielfunktion
- ▶  $\mathbf{x}^* \in \mathcal{L}$  ist optimale Lösung falls  $f(\mathbf{x}^*) \geq f(\mathbf{x})$  für alle  $\mathbf{x} \in \mathcal{L}$

Minimierungsprobleme: analog

Problem: variantenreich, meist NP-schwer

# Black-Box-Löser

- ▶ (Ganzzahlige) Lineare Programmierung
- ▶ Aussagenlogik
- ▶ Constraint-Programming  $\approx$  Verallgemeinerung von beidem

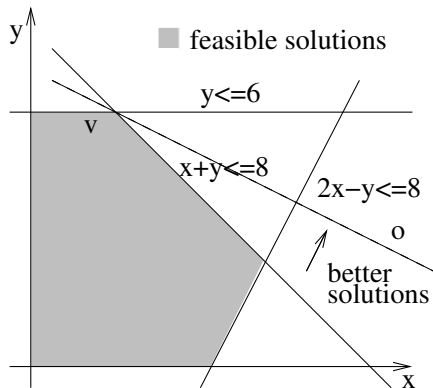
# Lineare Programmierung

Ein **lineares Programm** mit  $n$  **Variablen** und  $m$  **Constraints** (NB) wird durch das folgende Minimierungs-/Maximierungsproblem definiert:

- ▶ Kostenfunktion  $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$   
 $\mathbf{c}$  ist der **Kostenvektor**
- ▶  $m$  **Constraints** der Form  $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$  mit  $\bowtie_i \in \{\leq, \geq, =\}$ ,  $\mathbf{a}_i \in \mathbb{R}^n$ .  
Wir erhalten:

$$\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n : \forall j \in 1..n : x_j \geq 0 \wedge \forall i \in 1..m : \mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i\} .$$

# Ein einfaches Beispiel



## Beispiel: Kürzeste Wege

maximiere

$$\sum_{v \in V} d_v$$

so dass

$$d_s = 0$$

$$d_w \leq d_v + c(v, w) \quad \text{für alle } (v, w) \in E$$

# Eine Anwendung – Tierfutter

- ▶  $n$  Futtersorten,  
Sorte  $i$  kostet  $c_i$  Euro/kg.
- ▶  $m$  Anforderungen an gesunde Ernährung (Kalorien, Proteine, Vitamin C, ...).  
Sorte  $i$  enthält  $a_{ji}$  Prozent des täglichen Bedarfs pro kg bzgl. Anforderung  $j$
- ▶ Sei  $a_{ji}$  die  $i$ -te Komponente von Vektor  $\mathbf{a}_j$ .



# Eine Anwendung – Tierfutter

- ▶  $n$  Futtersorten,  
Sorte  $i$  kostet  $c_i$  Euro/kg.
- ▶  $m$  Anforderungen an gesunde Ernährung (Kalorien, Proteine, Vitamin C, ...).  
Sorte  $i$  enthält  $a_{ji}$  Prozent des täglichen Bedarfs pro kg bzgl. Anforderung  $j$
- ▶ Sei  $a_{ji}$  die  $i$ -te Komponente von Vektor  $\mathbf{a}_j$ .
- ▶ Definiere  $x_i$  als zu beschaffende Menge von Sorte  $i$
- ▶ LP-Lösung gibt eine kostenoptimale “gesunde” Mischung.



# Verfeinerungen

- ▶ Obere Schranken (Radioaktivität, Cadmium, Kuhhirn, ...)
- ▶ Beschränkte Reserven (z. B. eigenes Heu)
- ▶ bestimmte abschnittsweise lineare Kostenfunktionen (z. B. mit Abstand wachsende Transportkosten)
- ▶ Minimale Abnahmemengen
- ▶ die meisten nichtlinearen Kostenfunktionen
- ▶ Ganzzahlige Mengen (für wenige Tiere)
- ▶ Garbage in, Garbage out



# Algorithmen und Implementierungen

- ▶ LPs lassen sich in **polynomieller Zeit lösen** [Khachiyan 1979]
  - ▶ Worst case  $O\left(\max(m, n)^{\frac{7}{2}}\right)$
  - ▶ In der Praxis geht das viel schneller
  - ▶ Robuste, effiziente Implementierungen sind sehr aufwändig
- ↪ Fertige freie und kommerzielle Pakete

# Ganzzahlige Lineare Programmierung

**ILP:** Integer Linear Program, lineares Programm mit der zusätzlichen Bedingung  $x_i \in \mathbb{N}$ .  
oft: 0/1 ILP mit  $x_i \in \{0,1\}$

**MILP:** Mixed Integer Linear Program, lineares Programm bei dem **einige** Variablen ganzzahlig sein müssen.

**Lineare Relaxation:** Entferne die Ganzzahligkeitsbedingungen eines (M)ILP

# Beispiel: Rucksackproblem

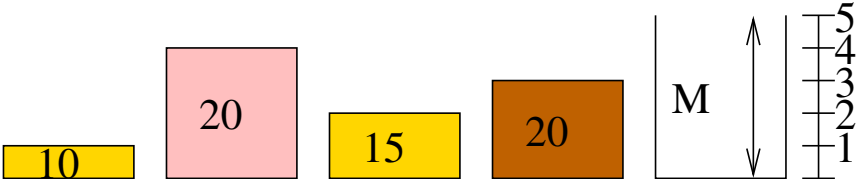
maximiere  $\mathbf{p} \cdot \mathbf{x}$

derart, dass

$$\mathbf{w} \cdot \mathbf{x} \leq M, x_i \in \{0, 1\} \text{ for } 1 \leq i \leq n .$$

$x_i = 1$  gdw. Gegenstand  $i$  in den Rucksack kommt.

0/1 Variablen sind typisch für ILPs



# Umgang mit (M)ILPs

- NP-schwer
- + Ausdrucksstarke Modellierungssprache
- + Es gibt generische Lösungsansätze, die manchmal gut funktionieren
- + Viele Möglichkeiten für Näherungslösungen
- + Die Lösung der linearen Relaxation hilft oft, z. B. einfach **runden**.
- + Ausgefeilte Softwarepakete

**Beispiel:** Beim **Rucksackproblem** gibt es nur **eine** fraktionale Variable in der linearen Relaxation – Abrunden ergibt zulässige Lösung. Annähernd optimal, falls Gewichte und Profite  $\ll$  Kapazität

# Nie zurückschauen – Greedy-Algorithmen

(deutsch: **gierige** Algorithmen, Ausdruck wenig gebräuchlich)

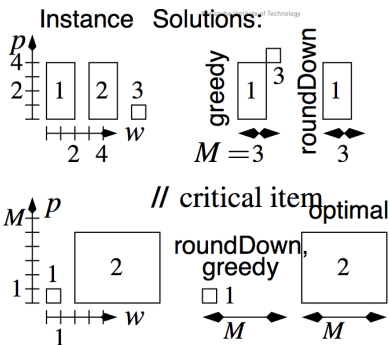
Idee: treffe jeweils eine **lokal** optimale Entscheidung

# Optimale Greedy-Algorithmen

- ▶ Dijkstras Algorithmus für **kürzeste Wege**
- ▶ **Minimale Spannbäume**
  - ▶ Jarník-Prim
  - ▶ Kruskal
- ▶ Selection-Sort (wenn man so will)

Viel häufiger, z.T. mit Qualitätsgarantien.  
Mehr: Vorlesungen Algorithmen II und  
Approximations- und Onlinealgorithmen

# Beispiel: Rucksackproblem



**Procedure** roundDownKnapsack  
sort items by profit density  $\frac{p_i}{w_i}$   
find  $\min \left\{ j : \sum_{i=1}^j w_j > M \right\}$   
output items  $1..j-1$

**Procedure** greedyKnapsack  
sort items by profit density  $\frac{p_i}{w_i}$   
**for**  $i := 1$  **to**  $n$  **do**  
    **if** there is room for item  $i$  **then**  
        insert it into the knapsack

# Dynamische Programmierung – Aufbau aus Bausteinen

Anwendbar, wenn das **Optimalitätsprinzip** gilt:

- ▶ Optimale Lösungen bestehen aus optimalen Lösungen für Teilprobleme.
- ▶ Mehrere optimale Lösungen  $\Rightarrow$  es ist egal, welche benutzt wird.



# Beispiel: Rucksackproblem

**Annahme:** ganzzahlige Gewichte

$P(i, C)$  := optimaler Profit für Gegenstände  $1, \dots, i$   
unter Benutzung von Kapazität  $\leq C$ .

$$P(0, C) := 0$$

**Lemma:**

$$\forall 1 \leq i \leq n : P(i, C) = \max(P(i-1, C), \\ P(i-1, C - w_i) + p_i)$$

# Dynamische Programmierung

auszufüllende Tabelle

**Wdh. Lemma:**

$$\forall 1 \leq i \leq n : P(i, C) = \max(P(i-1, C), \\ P(i-1, C - w_i) + p_i)$$

i / c	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

## Beweis des Lemmas

$P(i, C)$  := optimaler Profit für Gegenstände  $1, \dots, i$  bei Kap.  $C$ .

**Lemma:**  $P(i, C) = \max(P(i-1, C), P(i-1, C - w_i) + p_i)$

**Beweis:**

Sei  $x$  optimale Lösung für Objekte  $1..i$ , Kapazität  $C$ , d. h.

$c \cdot x = P(i, C)$ .

## Beweis des Lemmas

$P(i, C)$  := optimaler Profit für Gegenstände  $1, \dots, i$  bei Kap.  $C$ .

**Lemma:**  $P(i, C) = \max(P(i-1, C), P(i-1, C - w_i) + p_i)$

**Beweis:**

Sei  $\mathbf{x}$  optimale Lösung für Objekte  $1..i$ , Kapazität  $C$ , d. h.

$$\mathbf{c} \cdot \mathbf{x} = P(i, C).$$

**Fall**  $x_i = 0$ :

$\Rightarrow \mathbf{x}$  ist auch (opt.) Lösung für Objekte  $1..i-1$ , Kapazität  $C$ .

$$\Rightarrow P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$$

## Beweis des Lemmas

$P(i, C)$  := optimaler Profit für Gegenstände  $1, \dots, i$  bei Kap.  $C$ .

**Lemma:**  $P(i, C) = \max(P(i-1, C), P(i-1, C - w_i) + p_i)$

**Beweis:**

Sei  $\mathbf{x}$  optimale Lösung für Objekte  $1..i$ , Kapazität  $C$ , d. h.

$$\mathbf{c} \cdot \mathbf{x} = P(i, C).$$

**Fall**  $x_i = 0$ :

$\Rightarrow \mathbf{x}$  ist auch (opt.) Lösung für Objekte  $1..i-1$ , Kapazität  $C$ .

$$\Rightarrow P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$$

**Fall**  $x_i = 1$ :

$\Rightarrow \mathbf{x}$  ohne  $i$  ist auch Lösung für Objekte  $1..i-1$ , Kapazität  $C - w_i$ .

Wegen Austauschbarkeit muß  $\mathbf{x}$  ohne  $i$  optimal für diesen Fall sein.

$$\Rightarrow P(i, C) - p_i = P(i-1, C - w_i)$$

$$\Leftrightarrow P(i, C) = P(i-1, C - w_i) + p_i$$

## Beweis des Lemmas

$P(i, C)$  := optimaler Profit für Gegenstände  $1, \dots, i$  bei Kap.  $C$ .

**Lemma:**  $P(i, C) = \max(P(i-1, C), P(i-1, C - w_i) + p_i)$

**Beweis:**

Sei  $\mathbf{x}$  optimale Lösung für Objekte  $1..i$ , Kapazität  $C$ , d. h.

$$\mathbf{c} \cdot \mathbf{x} = P(i, C).$$

**Fall**  $x_i = 0$ :

$\Rightarrow \mathbf{x}$  ist auch (opt.) Lösung für Objekte  $1..i-1$ , Kapazität  $C$ .

$$\Rightarrow P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$$

**Fall**  $x_i = 1$ :

$\Rightarrow \mathbf{x}$  ohne  $i$  ist auch Lösung für Objekte  $1..i-1$ , Kapazität  $C - w_i$ .

Wegen Austauschbarkeit muß  $\mathbf{x}$  ohne  $i$  optimal für diesen Fall sein.

$$\Rightarrow P(i, C) - p_i = P(i-1, C - w_i)$$

$$\Leftrightarrow P(i, C) = P(i-1, C - w_i) + p_i$$

**Insgesamt**, wegen Optimalität von  $\mathbf{x}$ ,

$$P(i, C) = \max(P(i-1, C), P(i-1, C - w_i) + p_i)$$

□

## Berechnung von $P(i, C)$ elementweise:

$$P(i, C) = \max(P(i-1, C), P(i-1, C - w_i) + p_i)$$

i/C	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

**Procedure** knapsack( $\mathbf{p}$ ,  $\mathbf{c}$ ,  $n$ ,  $M$ )

array  $P[0 \dots M] = [0, \dots, 0]$

bitarray decision $[1 \dots n, 0 \dots M] = [(0, \dots, 0), \dots, (0, \dots, 0)]$

**for**  $i := 1$  **to**  $n$  **do**

    // invariant:  $\forall C \in \{1, \dots, M\} : P[C] = P(i-1, C)$

**for**  $C := M$  **downto**  $w_i$  **do**

**if**  $P[C - w_i] + p_i > P[C]$  **then**

$P[C] := P[C - w_i] + p_i$

            decision $[i, C] := 1$

# Rekonstruktion der Lösung

```
 $C := M$   
array  $\mathbf{x}[1 \dots n]$   
for  $i := n$  downto 1 do  
     $\mathbf{x}[i] := \text{decision}[i, C]$   
    if  $\mathbf{x}[i] = 1$  then  $C := C - w_i$   
endfor  
return  $\mathbf{x}$ 
```

Analyse:

Zeit:  $O(nM)$  pseudopolynomiell

Platz:  $M + O(n)$  Maschinenwörter plus  $nM$  bits.



# Beispiel

Maximiere  $(10, 20, 15, 20) \cdot \mathbf{x}$ ,  
so dass  $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2						
3						
4						

# Beispiel

Maximiere  $(10, 20, 15, 20) \cdot \mathbf{x}$ ,  
so dass  $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3						
4						

# Beispiel

Maximiere  $(10, 20, 15, 20) \cdot \mathbf{x}$ ,  
so dass  $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3	0, (0)	10, (0)	15, (1)	25, (1)	30, (0)	35, (1)
4						

# Beispiel

Maximiere  $(10, 20, 15, 20) \cdot \mathbf{x}$ ,  
so dass  $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3	0, (0)	10, (0)	15, (1)	25, (1)	30, (0)	35, (1)
4	0, (0)	10, (0)	15, (0)	25, (0)	30, (0)	35, (0)

# Algorithmenentwurf mittels dynamischer Programmierung

1. Was sind die **Teilprobleme**? Kreativität!
2. Wie setzen sich optimale Lösungen aus Teilproblemlösungen zusammen? Beweisnot
3. Bottom-up Aufbau der **Lösungstabelle** einfach
4. **Rekonstruktion** der Lösung einfach
5. Verfeinerungen:  
Platz sparen, Cache-effizient, Parallelisierung Standard-Trickkiste

# Anwendungen dynamischer Programmierung

- ▶ Bellman-Ford-Alg. für kürzeste Wege Teilpfade
- ▶ Edit distance/approx. string matching Algorithmen II?
- ▶ Verkettete Matrixmultiplikation Übung?
- ▶ Rucksackproblem Gegenstände 1..i füllen Teil des Rucksacks
- ▶ Geld wechseln Übung?

## Gegenbeispiel: Teilproblemeigenschaft

Angenommen, die schnellste Strategie für 20 Runden auf dem Hockenheimring verbraucht den Treibstoff vollständig.

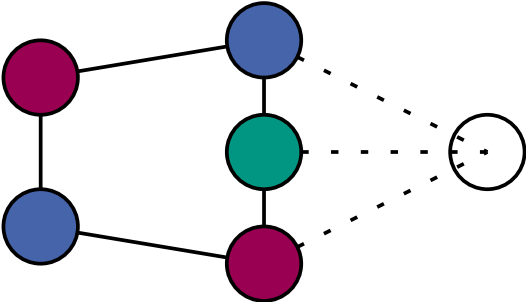


Keine gute Teilstrategie für 21 Runden.

**Frage:** Wie kann man “constrained shortest path” trotzdem mittels dynamischer Programmierung modellieren?

# Gegenbeispiel: Austauschbarkeit

Optimale Graphfärbungen sind nicht austauschbar.





# Systematische Suche

Idee:

Alle (sinnvollen) Möglichkeiten ausprobieren.

## Anwendungen:

- ▶ Ganzzahlige lineare Programmierung (Integer Linear Programming) (ILP)
- ▶ Constraint Satisfaction
- ▶ SAT (Aussagenlogik)
- ▶ Theorembeweiser (Prädikatenlogik, ...)
- ▶ konkrete NP-schwere Probleme
- ▶ Strategiespiele
- ▶ Puzzles

# Beispiel: Branch-and-Bound für das Rucksackproblem

**Function** bbKnapsack( $(p_1, \dots, p_n), (w_1, \dots, w_n), M$ ) :  $\mathcal{L}$

**assert**  $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$

$\hat{x}$  = heuristicKnapsack( $(p_1, \dots, p_n), (w_1, \dots, w_n), M$ ) :  $\mathcal{L}$

$x$  :  $\mathcal{L}$

recurse(1,  $M$ , 0)

**return**  $\hat{x}$

# Beispiel: Branch-and-Bound für das Rucksackproblem

**Function** bbKnapsack( $(p_1, \dots, p_n), (w_1, \dots, w_n), M$ ) :  $\mathcal{L}$

**assert**  $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$

$\hat{x}$  = heuristicKnapsack( $(p_1, \dots, p_n), (w_1, \dots, w_n), M$ ) :  $\mathcal{L}$

$x$  :  $\mathcal{L}$

recurse(1,  $M$ , 0)

**return**  $\hat{x}$

// Find solutions assuming  $x_1, \dots, x_{i-1}$  are fixed,

//  $M' = M - \sum_{k < i} x_k w_k, P = \sum_{k < i} x_k p_k$ .

**Procedure** recurse( $i, M', P$  :  $\mathbb{N}$ )

// Find solutions assuming  $x_1, \dots, x_{i-1}$  are fixed,  
 //  $M' = M - \sum_{k < i} x_k w_k, P = \sum_{k < i} x_k p_k.$

**x** // current solution  
 **$\hat{x}$**  // best solution so far

**Procedure** recurse( $i, M', P : \mathbb{N}$ )

$u := P + \text{upperBound}((p_i, \dots, p_n), (w_i, \dots, w_n), M')$

**if**  $u > \mathbf{p} \cdot \hat{\mathbf{x}}$  **then**

**if**  $i > n$  **then**  $\hat{\mathbf{x}} := \mathbf{x}$

**else**

// **Branch** on variable  $x_i$

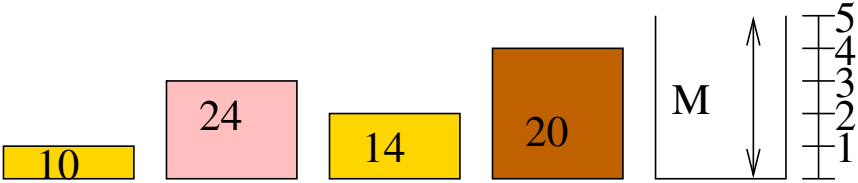
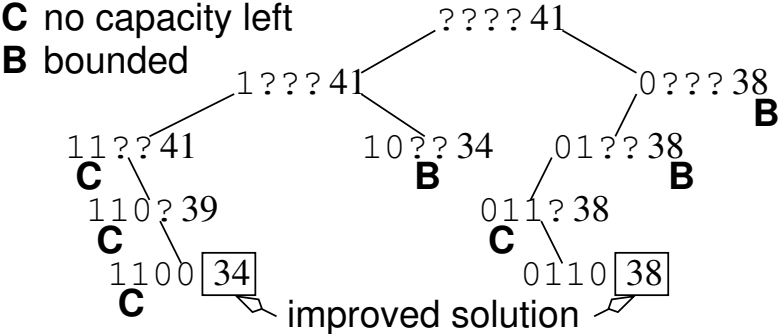
**if**  $w_i \leq M'$  **then**  $x_i := 1$ ; recurse( $i + 1, M' - w_i, P + p_i$ )

**if**  $u > \mathbf{p} \cdot \hat{\mathbf{x}}$  **then**  $x_i := 0$ ; recurse( $i + 1, M', P$ )

Schlechtester Fall:  $2^n$  rekursive Aufrufe

Im Mittel: Linearzeit?

# Beispielrechnung



# Branch-and-Bound – allgemein

Branching (Verzweigen): Systematische **Fallunterscheidung**,  
z. B. **rekursiv** (Alternative, z. B. **Prioritätsliste**)

Verzweigungsauswahl: Wonach soll die Fallunterscheidung stattfinden?  
(z. B. welche Variable bei ILP)

Reihenfolge der Fallunterscheidung: Zuerst vielversprechende Fälle  
(lokal oder global)

Bounding: Nicht weitersuchen, wenn **optimistische** Abschätzung der  
erreichbaren Lösungen schlechter als **beste** (woanders)  
**gefundene Lösung**.

Duplikateliminierung: Einmal suchen reicht

Anwendungsspez. Suchraumbeschränkungen: Schnittebenen (ILP),  
Lemma-Generierung (Logik),...

# Lokale Suche – global denken, lokal handeln

find some feasible solution  $\mathbf{x} \in \mathcal{S}$

$\hat{\mathbf{x}} := \mathbf{x}$

//  $\hat{\mathbf{x}}$  is best solution found so far

**while** not satisfied with  $\hat{\mathbf{x}}$  **do**

$\mathbf{x} :=$  some **heuristically** chosen element from  $\mathcal{N}(\hat{\mathbf{x}}) \cap \mathcal{S}$

**if**  $f(\mathbf{x}) > f(\hat{\mathbf{x}})$  **then**  $\hat{\mathbf{x}} := \mathbf{x}$

# Hill Climbing

Find some feasible solution  $\mathbf{x} \in \mathcal{L}$

$\hat{\mathbf{x}} := \mathbf{x}$  // best solution found so far

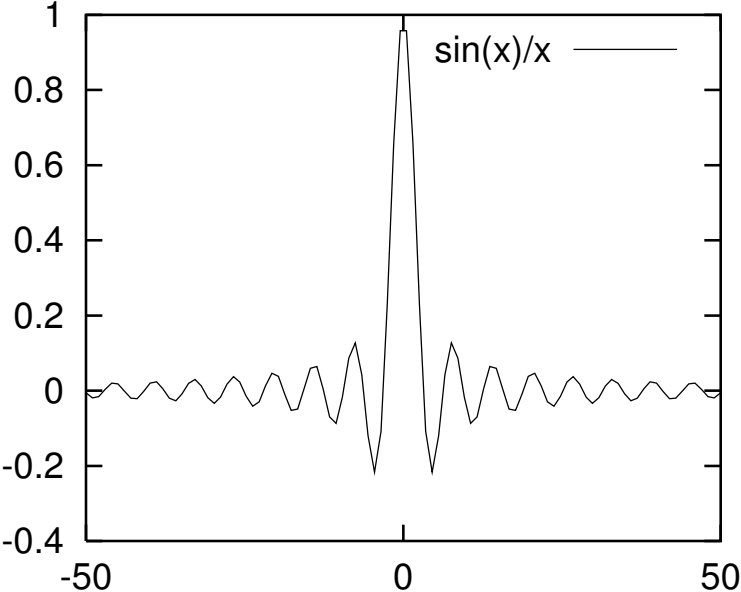
**loop**

**if**  $\exists \mathbf{x} \in \mathcal{N}(\hat{\mathbf{x}}) \cap \mathcal{L} : f(\mathbf{x}) > f(\hat{\mathbf{x}})$  **then**  $\hat{\mathbf{x}} := \mathbf{x}$

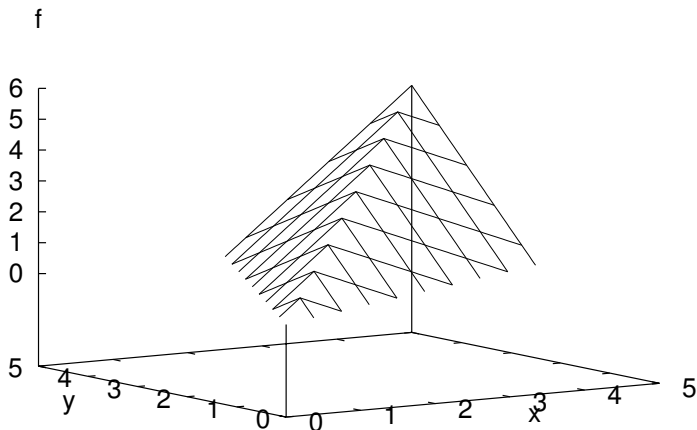
**else return**  $\hat{\mathbf{x}}$  // local optimum found



# Problem: Lokale Optima



# Warum die Nachbarschaft wichtig ist

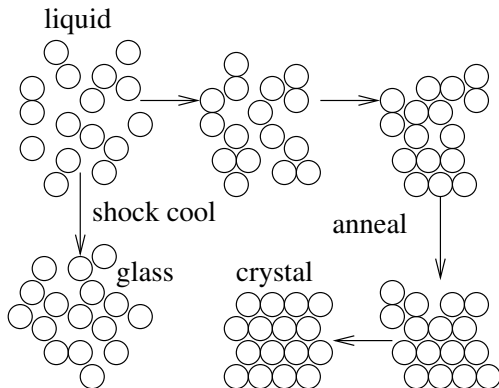


Gegenbeispiel für Koordinatensuche ( $\min(3x - y, 3y - x)$ )

# Jenseits von Hill Climbing

Auch Verschlechterungen akzeptieren.

- ▶ Simulated Annealing: physikalische Analogie



# Jenseits von Hill Climbing

Auch Verschlechterungen akzeptieren.

- ▶ Simulated Annealing: physikalische Analogie
- ▶ Tabusuche
- ▶ ...

# Evolutionäre Algorithmen

Ausführliche Behandlung würde den Rahmen sprengen.  
Verallgemeinerung von lokaler Suche:

- ▶  $x \rightarrow$  **Population** von Lösungskandidaten
- ▶ Reproduktion fitter Lösungen
- ▶ Mutation ähnlich lokaler Suche
- ▶ zusätzlich: geschlechtliche Vermehrung.  
Idee: erben von guten Eigenschaften der Eltern

# Zusammenfassung

## Vor- und Nachteile generischer Optimierungsmethoden

Greedy: **Einfach** und **schnell**. Selten optimal. Manchmal **Approximationsgarantien**.

Systematische Suche: Einfach mit **Werkzeugen** z. B. (I)LP, SAT, constraint programming. Selbst gute Implementierungen mögen nur mit **kleinen** Instanzen funktionieren.

# Zusammenfassung

## Vor- und Nachteile generischer Optimierungsmethoden

**Greedy:** Einfach und schnell. Selten optimal. Manchmal Approximationsgarantien.

**Systematische Suche:** Einfach mit Werkzeugen z. B. (I)LP, SAT, constraint programming. Selbst gute Implementierungen mögen nur mit kleinen Instanzen funktionieren.

**Lineare Programmierung:** Einfach und einigermaßen schnell. Optimal, falls das Modell passt. Rundungsheuristiken ergeben Näherungslösungen

**Dynamische Programmierung:** Optimale Lösungen, falls Teilprobleme optimal und austauschbar sind. Hoher Platzverbrauch.

**Ganzzahlige lineare Programmierung:** Leistungsfähiges Werkzeug für optimale Lösungen. Gute Formulierungen können viel Know-how erfordern. Kann rechenintensiv sein.

# Zusammenfassung

## Vor- und Nachteile generischer Optimierungsmethoden

**Lokale Suche:** **Flexibel** und einfach. **Langsam**, aber oft **gute Lösungen** für schwierige Probleme.

**Hill climbing:** einfach, aber leidet an **lokalen Optima**.

**Simulated Annealing und Tabu Search:** **Leistungsfähig**, aber langsam. Tuning kann unschön werden.

**Evolutionäre Algorithmen:** Ähnliche Vor- und Nachteile wie lokale Suche. Durch geschl. Vermehrung potentiell mächtiger, aber auch langsamer und schwieriger, gut hinzukriegen. Weniger zielgerichtet.



# Kap. 13: Zusammenfassung

- ▶ Datenstrukturen
- ▶ Algorithmen
- ▶ Entwurfstechniken
- ▶ Analysetechniken

# Zusammenfassung – Datenstrukturen

- ▶ (doppelt) verkettete **Listen**, unbeschränkte (zyklische) **Felder**, Stapel, FIFOs, deques
- ▶ (beschränktes) **Hashing**: verketteten (universell) / lin. Suche
- ▶ sortiertes Feld
- ▶ Prioritätslisten (**binärer Heap**) (adressierbar)
- ▶ Implizite Repräsentation vollständiger Bäume
- ▶ Suchbäume: binär, **(a, b)-Baum**
- ▶ Graphen: **Adjazenzfeld** / Listen / Matrix
- ▶ Union-Find

# Zusammenfassung – Algorithmen

- ▶ Langzahlmultiplikation
- ▶ Insertion-, Merge-, Quick-, Heap-, Bucket-, Radix-sort, Selektion
- ▶ BFS, DFS, topologisches Sortieren
- ▶ Kürzeste Wege: Dijkstra, Bellman-Ford
- ▶ **MST**: Jarník-Prim, Kruskal

# Zusammenfassung – Entwurfstechniken I

- ▶ Iteration/Induktion/Schleifen, **Teile-und-Herrsche**
- ▶ Schleifen- und Datenstruktur-**Invarianten**
- ▶ **Randomisierung** (universelles Hashing, Quicksort, ...)
- ▶ **Graphenmodelle**
- ▶ Trennung Mathe  $\leftrightarrow$  **Funktionalität**  $\leftrightarrow$  **Repräsentation**  $\leftrightarrow$  **Algorithmus**  $\leftrightarrow$  Implementierung
- ▶ Sonderfälle vermeiden
- ▶ **Zeiger**datenstrukturen
- ▶ Datenstrukturen **augmentieren** (z.B. Teilbaumgrößen)
- ▶ Datenstrukturen **unbeschränkt** machen
- ▶ **Implizite** Datenstrukturen (z.B. Intervallgraphen)

# Zusammenfassung – Entwurfstechniken II

- ▶ **Algebra**  
(Karatsuba, univ. Hashfkt., Matrixmultiplikation für Graphen)
- ▶ Algorithmen**schemata** (z.B. DFS, lokale Suche)
- ▶ Verwendung abstrakter **Problemeigenschaften**  
(z.B. Schnitt/Kreis-Eigenschaft bei MST)
- ▶ Black-Box-Löser (z.B. lineare Programmierung)
- ▶ **Greedy**
- ▶ **Dynamische Programmierung**
- ▶ Systematische Suche
- ▶ Metaheuristiken (z.B. Lokale Suche)

# Zusammenfassung – Analysetechniken

- ▶ Summen, Rekurrenzen, Induktion, Master-Theorem, Abschätzung
- ▶ Asymptotik ( $O(\cdot)$ ,  $\dots$ ,  $\omega(\cdot)$ ), einfache Modelle
- ▶ Analyse im Mittel
- ▶ Amortisierung (z.B. unbeschränkte Felder)
- ▶ Linearität des Erwartungswertes, Indikatorzufallsvariablen
- ▶ Kombinatorik ( $\approx$  Zählen): univ. Hashfunktionen, untere Sortierschranken (Informationsmenge)
- ▶ Integrale als Summenabschätzung
- ▶ Schleifen/Datenstruktur-(In)varianten (z.B.  $(a, b)$ -Baum, Union-by-rank)

# Zusammenfassung – weitere Techniken

- ▶ Algorithm Engineering
- ▶ Parameter-Tuning (z.B. Basisfallgröße)
- ▶ High-Level-Pseudocode
- ▶ Dummies und Sentinels (Listen, insertion sort, ...)
- ▶ Speicherverwaltung