Algorithmen I

Peter Sanders

Übungen:

Robert Geisberger und Johannes Singler

Institut für theoretische Informatik, Algorithmik II

Web:

`http://algo2.iti.uni-karlsruhe.de/AlgorithmenI.php`

Organisatorisches

Vorlesungen:

Mo: 14:00–15:30

Mi: 14:00–14:45

Saalübung:

Mi: 14:45–15:30

Tutorium: wöchentlich

Einteilung mittels Webinscribe

Übungsblätter: wöchentlich

Ausgabe Mittwoch nach der Übung

Abgabe Freitag 14 Uhr (9 Tage nach Ausgabe)

Organisatorisches

Sprechstunde:

- Peter Sanders, Dienstag 14–15 Uhr, Raum 217

Organisatorisches

Mittsemesterklausur: Mo 8.6. 14:00–15:45, Audimax. 10% der Note

Abschlussklausur:

3.8.2009, 11:00–13:00. 90% der Note

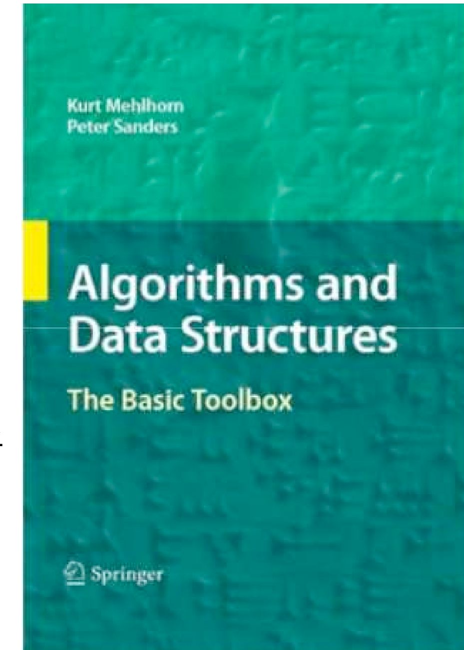
Materialien

- Folien, Übungsblätter
- Buch:

  K. Mehlhorn, P. Sanders

  Algorithms and Data Structures — The Basic Toolbox

  Springer 2008, ggf. einzelne Kapitel der deutschen Übersetzung von Prof. Martin Dietzfelbinger.



- Taschenbuch der Algorithmen

  Springer 2008 (Unterhaltung / Motivation)

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 $\approx 780..840$.

Das war damals “Elite” –

Machtzentrum des arabischen Kalifats auf seinem Höhepunkt.

Er hat ein Rechenlehrbuch geschrieben.

$\rightsquigarrow$Algorithmus wurde zum Synonym für Rechenvorschrift.

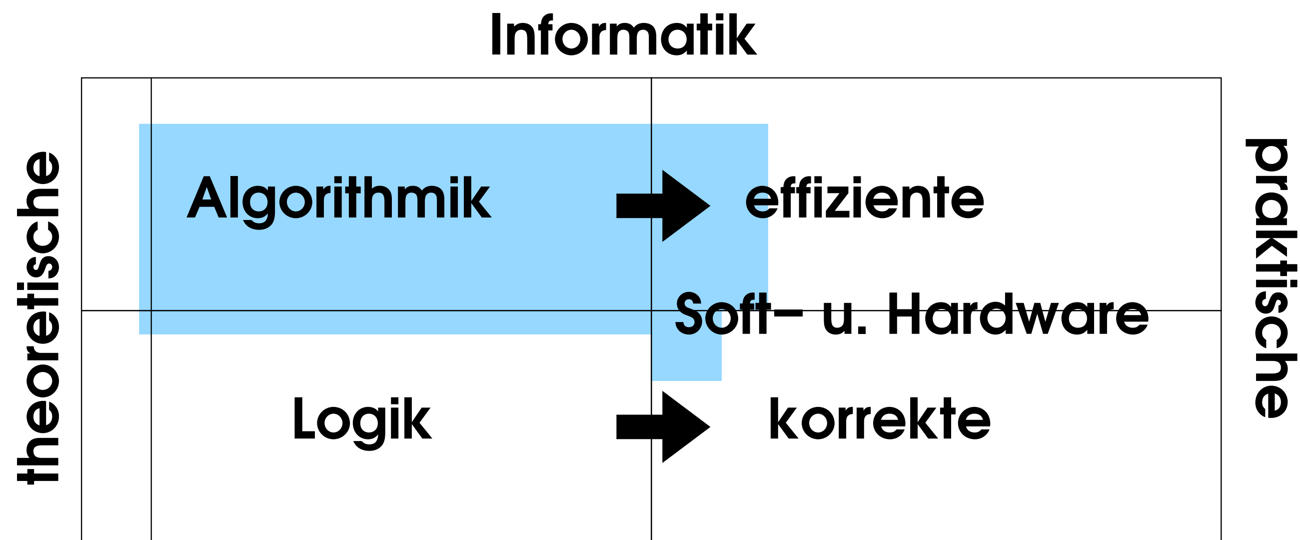
Moderne Definition (Wikipedia):

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

Algorithmik

Kerngebiet der (theoretischen) Informatik

mit direktem Anwendungsbezug



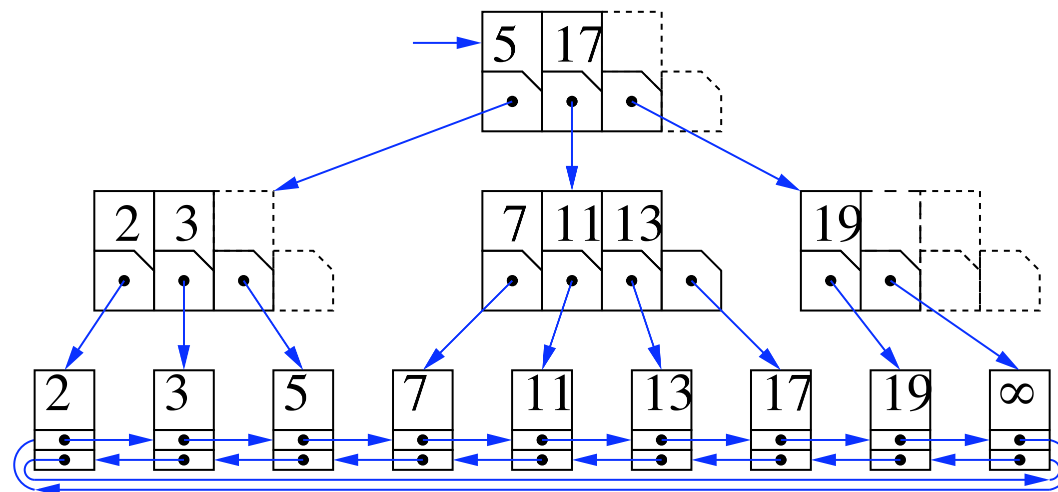
Datenstruktur

Ein Algorithmus bearbeitet Daten.

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

Immer wiederkehrende Datenstrukturen und dazugehörige Algorithmenteile

⇝ wichtiger Teil der Basic Toolbox



Themenauswahl: Werkzeugkasten

Immer wieder benötigte

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

Jeder Informatiker braucht das $\leadsto$ 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 I — 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

1 Amuse Geule

Beispiel: Langzahl-Multiplikation

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

Wir zählen

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

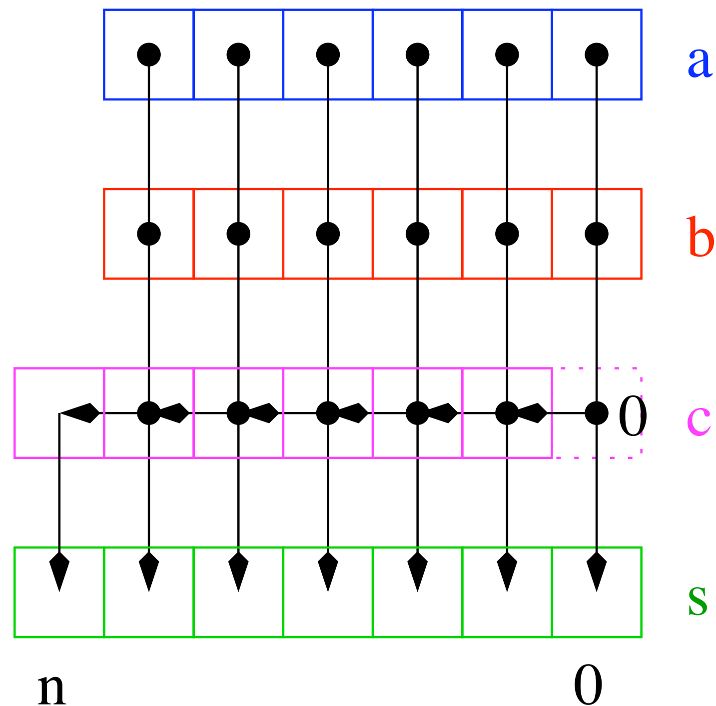
Ziffernmultiplikationen $(p', p) := a_i \cdot b_j$

1.1 Addition

c=0 : Digit // carry / Überlauf

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

$s_n := c$



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

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

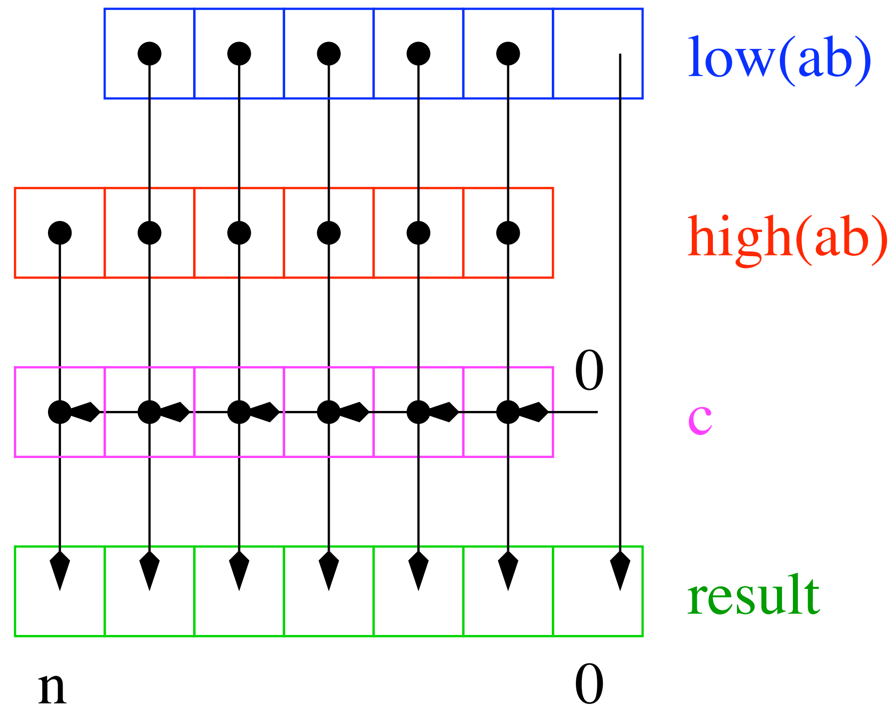
Schleifen: **for** , **while** , **repeat** . . . **until** ,. . .

uvam: Buch Abschnitt 2.3, hier: just in time und on demand

if , Datentypen, Klassen, Speicherverwaltung

Ziffernmultiplikation

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



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

    $(c,$ 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

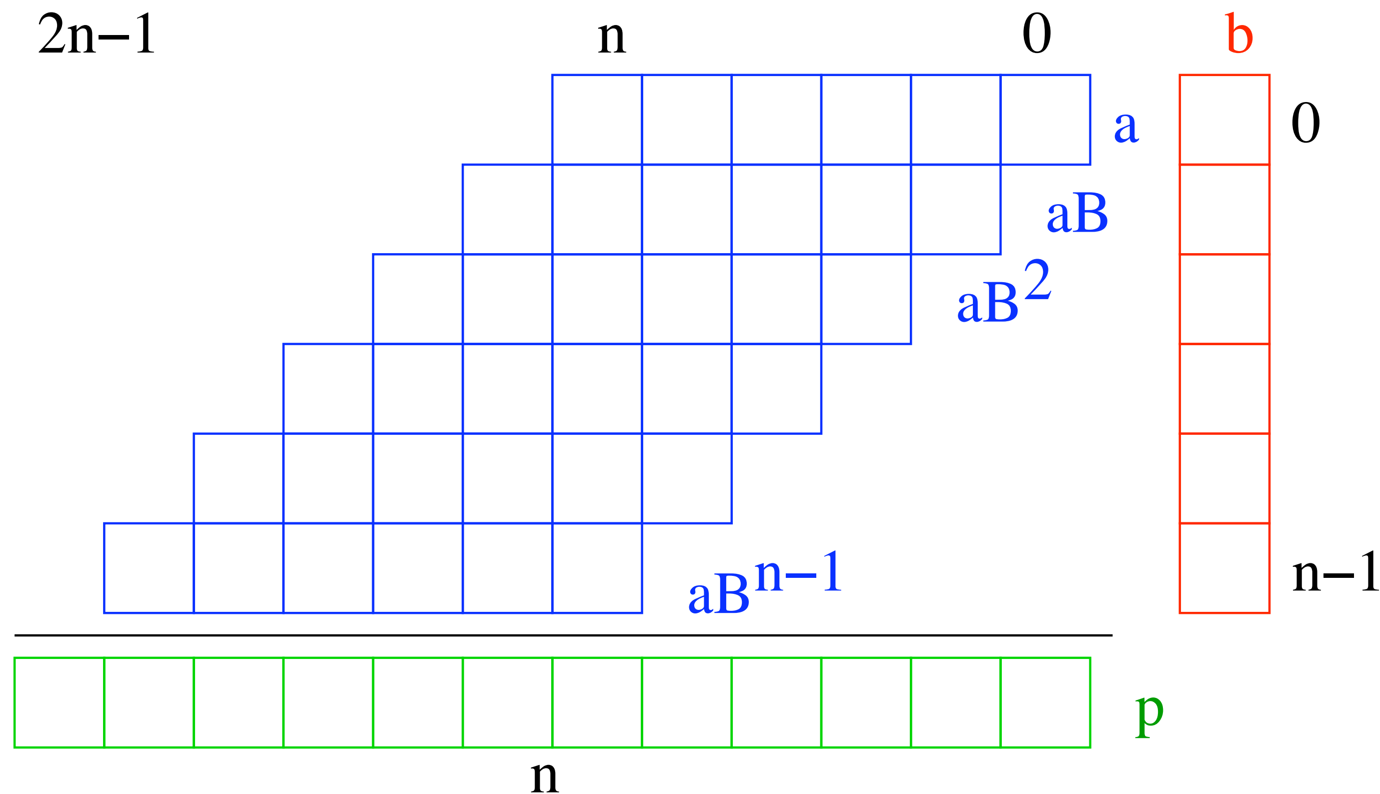
1.2 Schulmultiplikation

p=0 : $\mathbb{N}$ // Langzahl

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

//Langzahladdition, Langzahl mal Ziffer, Schieben:

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



Schultmultiplikation Analyse

p=0 : $\mathbb{N}$

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

$p := p$	// $n+j$ Ziffern (außer bei $j=0$)
$+$	// $n+1$ Ziffernadditionen (optimiert)
$a \cdot b[j]$	// je n Additionen/Multiplikationen
$\cdot B^j$	// schieben (keine Ziffernarithmetik)

Insgesamt:

n^2 Multiplikationen

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

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

Exkurs O-Kalkül, die Erste

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

Idee: Konstante Faktoren (und Anfangsstück) ausblenden

$+$ Operationen zählen $\rightsquigarrow$ Laufzeit welche Ops.?

$+$ Rechnungen vereinfachen

$+$ Interpretation vereinfachen

? Werfen wir zuviel Information weg ?

Beispiel: Schulmultiplikation braucht Zeit $\mathrm{O}(n^2)$

1.3 Ergebnisüberprüfung

später an Beispielen

1.4 Ein rekursiver Algorithmus

Function recMult(a,b)

assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

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

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

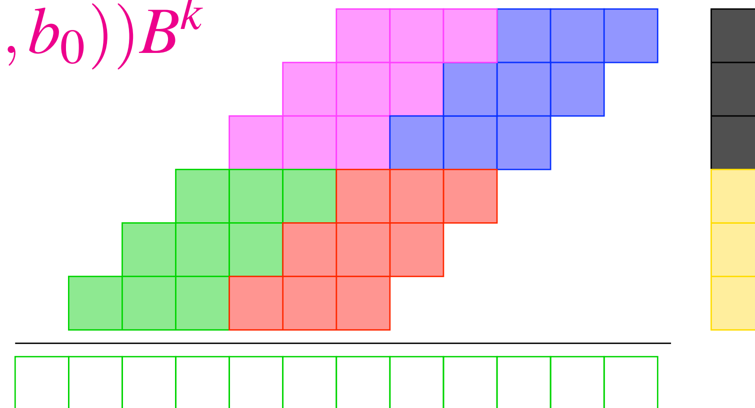
Schreibe b als $b_1 \cdot B^k + b_0$

return

recMult$(a_1, b_1) \cdot B^{2k} +$

(recMult(a_0, b_1)+recMult$(a_1, b_0))B^k$

+recMult(a_0, b_0)



Analyse

Function recMult(a,b) // $T(n)$ Ops

  assert a und b haben $n=2k$ Ziffern, n ist Zweierpotenz

  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?

Analyse

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

$\longrightarrow$ (Master-Theorem, stay tuned)

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

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,. . .

1.5 Karatsuba-Ofman Multiplikation[1962]

Beobachtung: $(a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_0 b_0 + a_1 b_0 + a_0 b_1$

Function recMult(a, b)

assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

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}:= recMult(a_1, b_1)

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

return

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

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

$+ c_{00}$

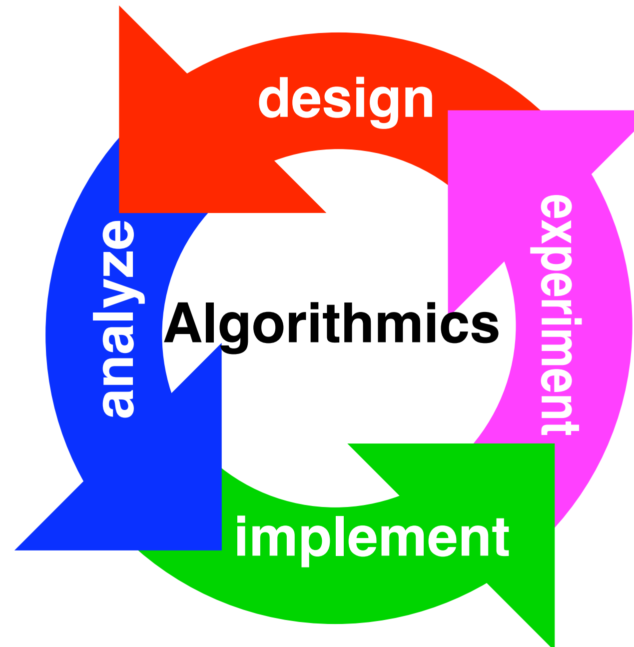
Analyse

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

$\longrightarrow$ (Master-Theorem)

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

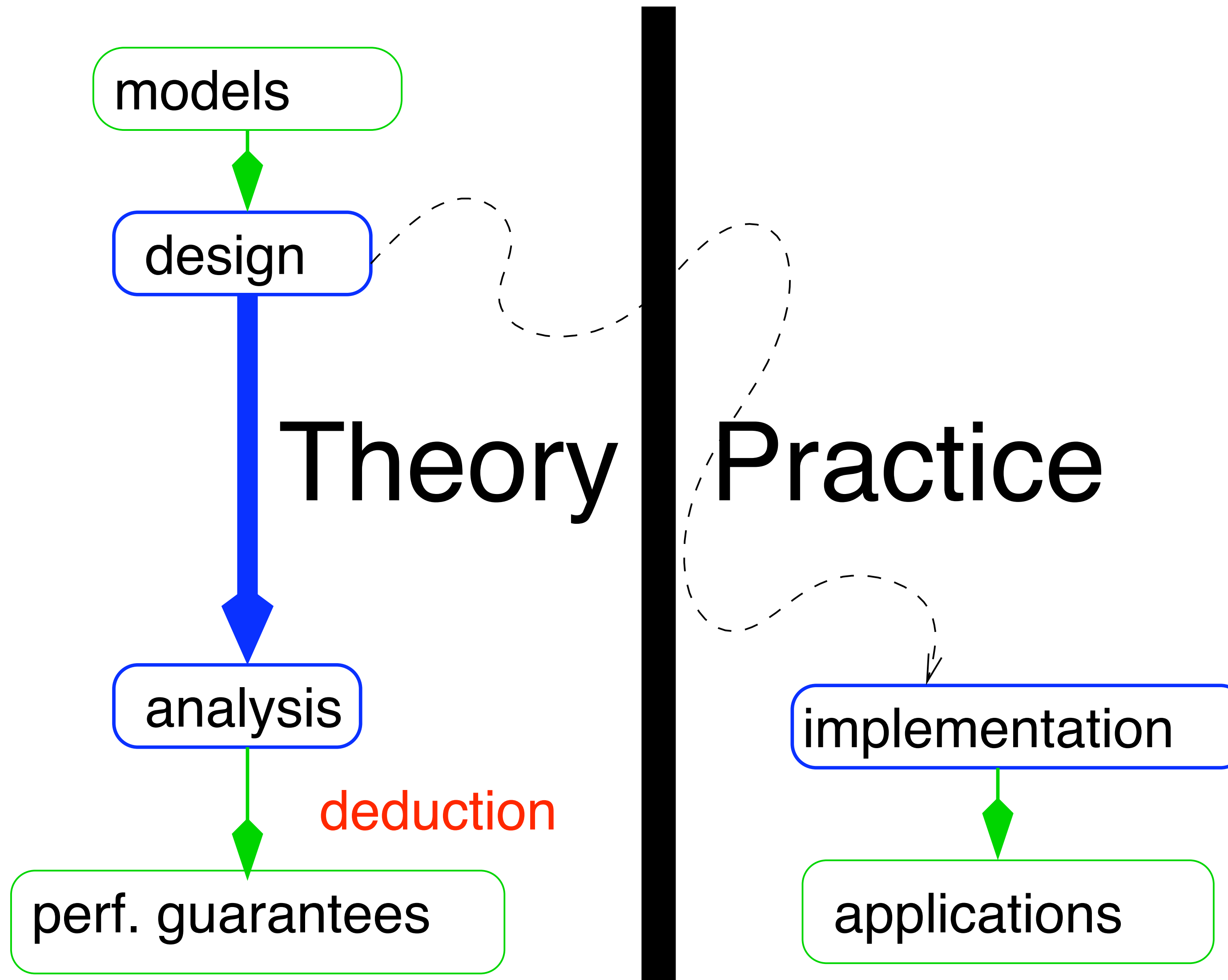
1.6 Algorithm Engineering – was hat das mit der Praxis zu tun?′



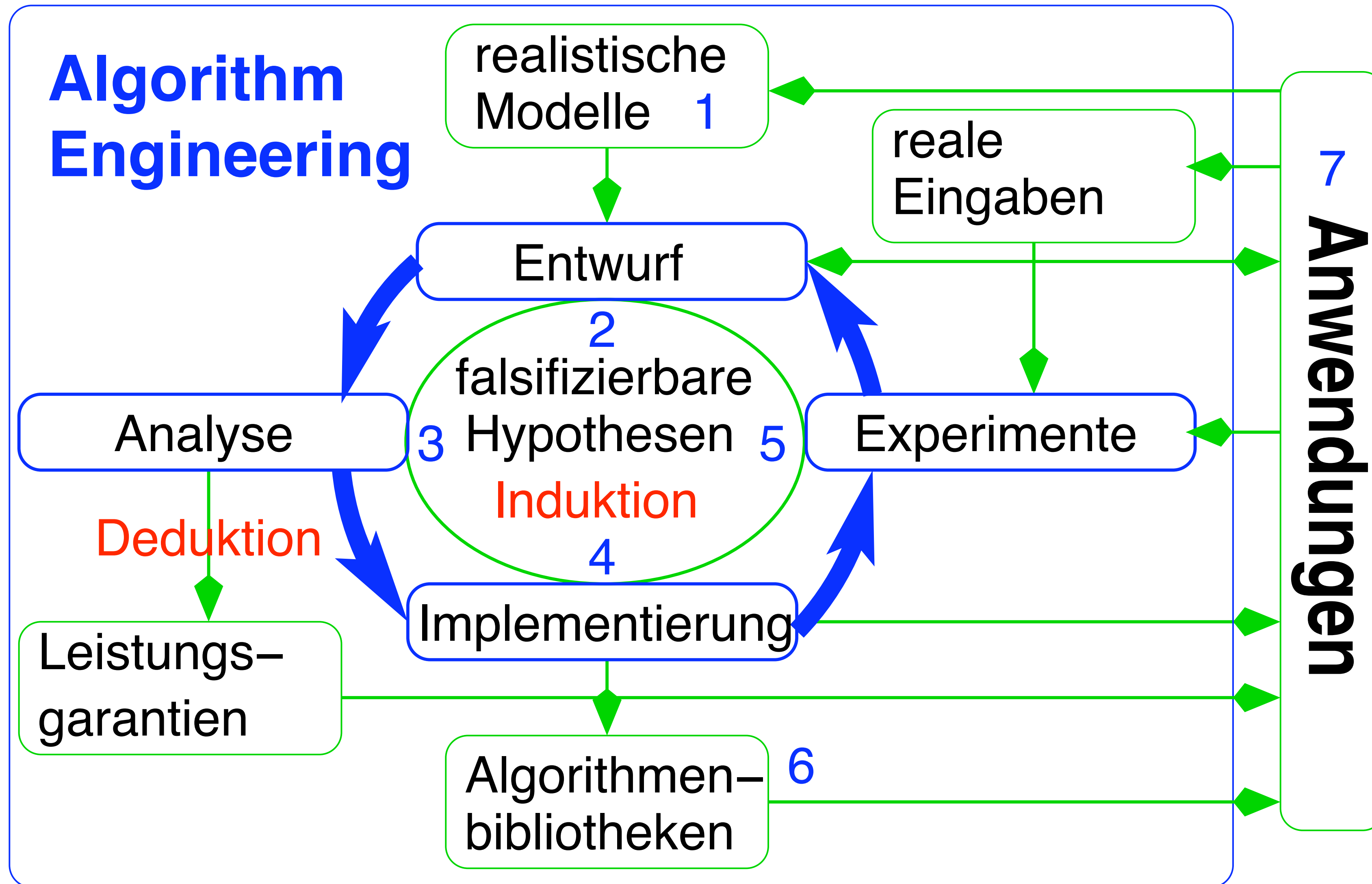
Mehr: DFG Schwerpunktprogram

`www.algorithm-engineering.de`

Algorithmentheorie (Karikatur)



Algorithmik als Algorithm Engineering



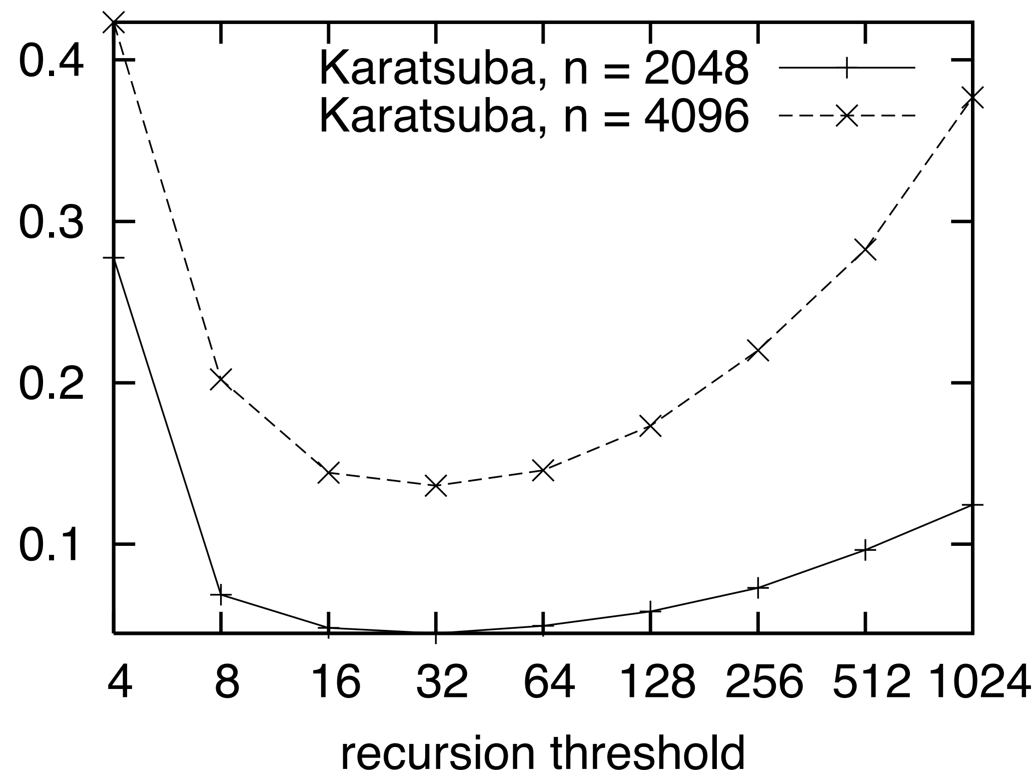
Zurück zur Langzahlmultiplikation

- Ziffterngröße $\leftrightarrow$ Hardware-Fähigkeiten

  z. B. 32 Bit

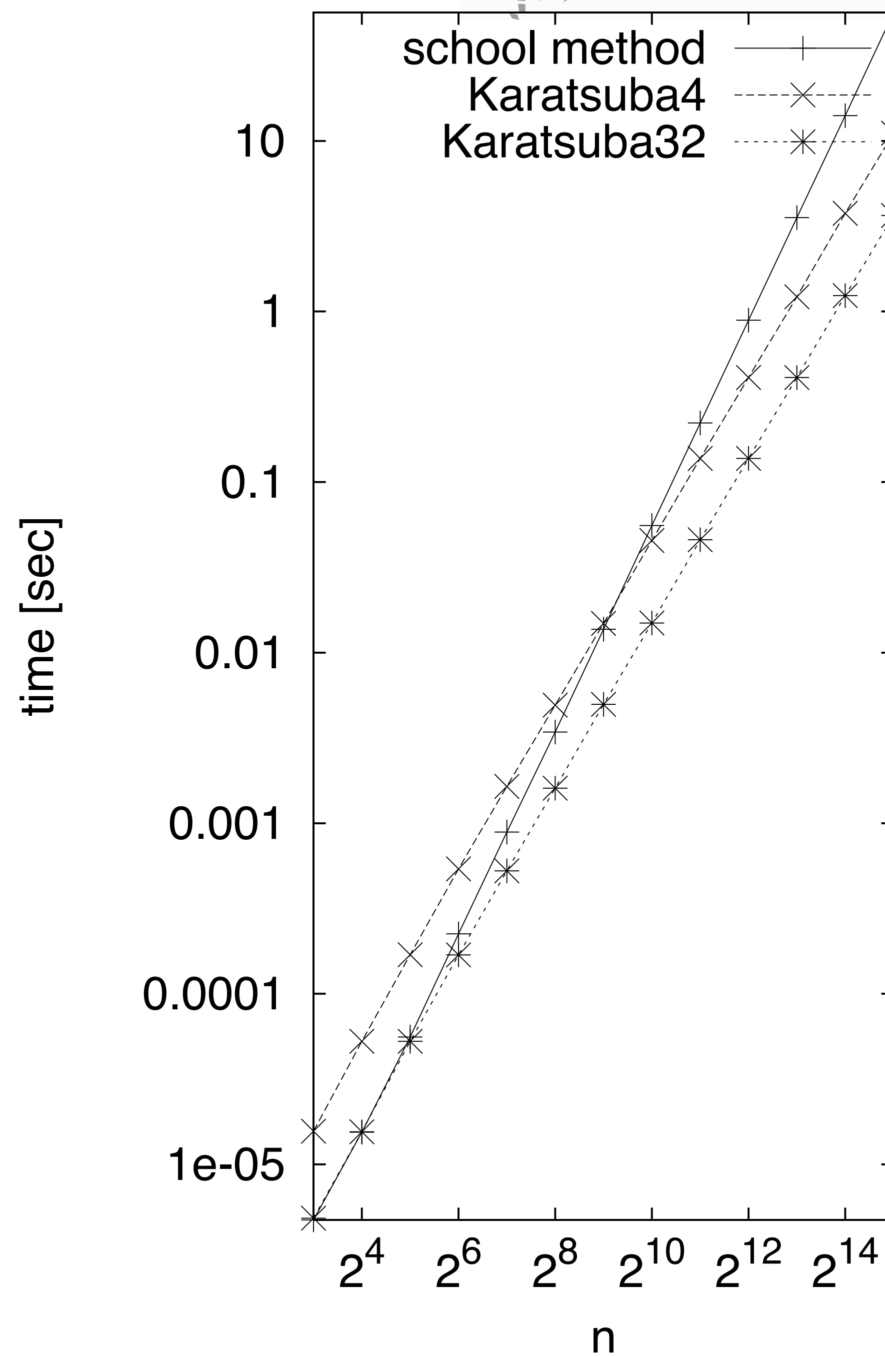
- Schulmultiplikation für kleine Eingaben

- 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
  $\rightsquigarrow \mathrm{O}(n)$ Multiplikationen von $\mathrm{O}(\log n)$-Bit Zahlen
- [Schönhage-Strassen 1971]: Bitkomplexität $\mathrm{O}(n \log n \log\log n)$
- [Fürer 2007]: Bitkomplexität $2^{\mathrm{O}(\log^* n)} n \log n$
- Praxis: Karatsuba-Multiplikation ist nützlich für Zahlenlängen aus der Kryptographie

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

2 Einführendes

2.1 Überblick

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

2.2 (Asymptotische) Algorithmenanalyse

Gegeben: Ein Programm

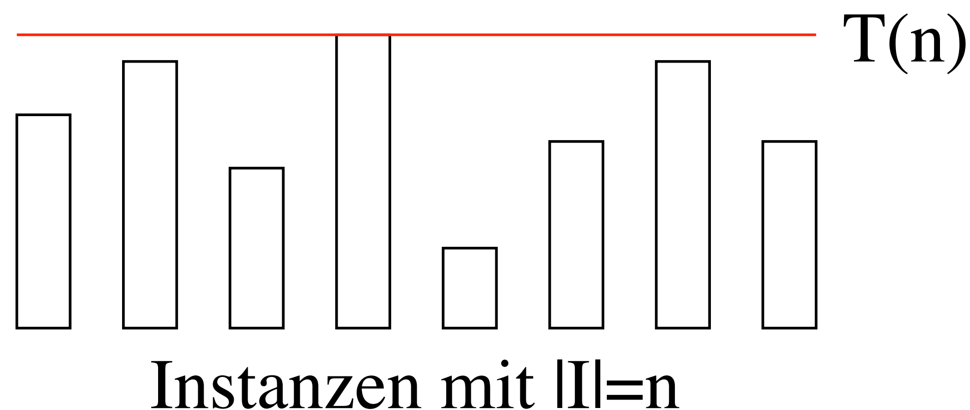
Gesucht: Laufzeit $T(I)$ (# Takte), eigentlich für alle Eingaben I (!)

(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 '$\{f(n)\} \subseteq E$'

$$cf(n) = \Theta(f(n)) \text{ für jede positive Konstante } c$$

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

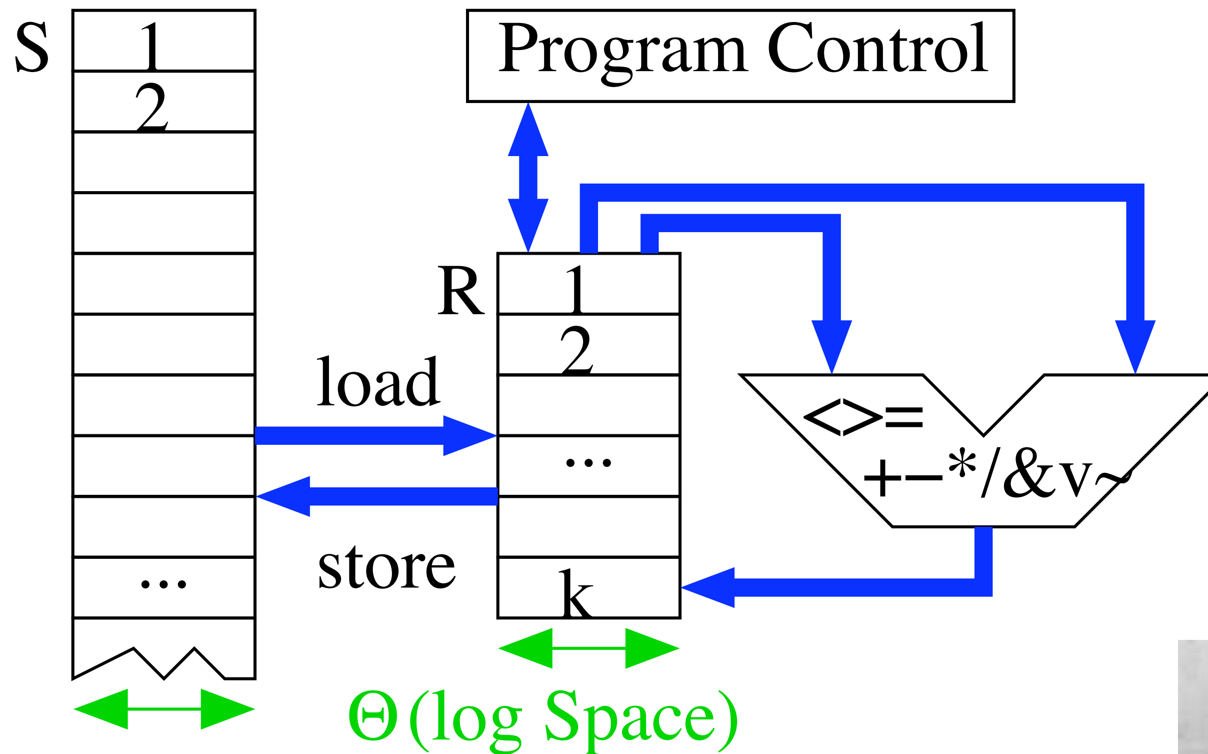
$$f(n) + g(n) = \Omega(f(n)),$$

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

$$\mathrm{O}(f(n)) \cdot \mathrm{O}(g(n)) = \mathrm{O}(f(n) \cdot g(n)).$$

u. s. w.

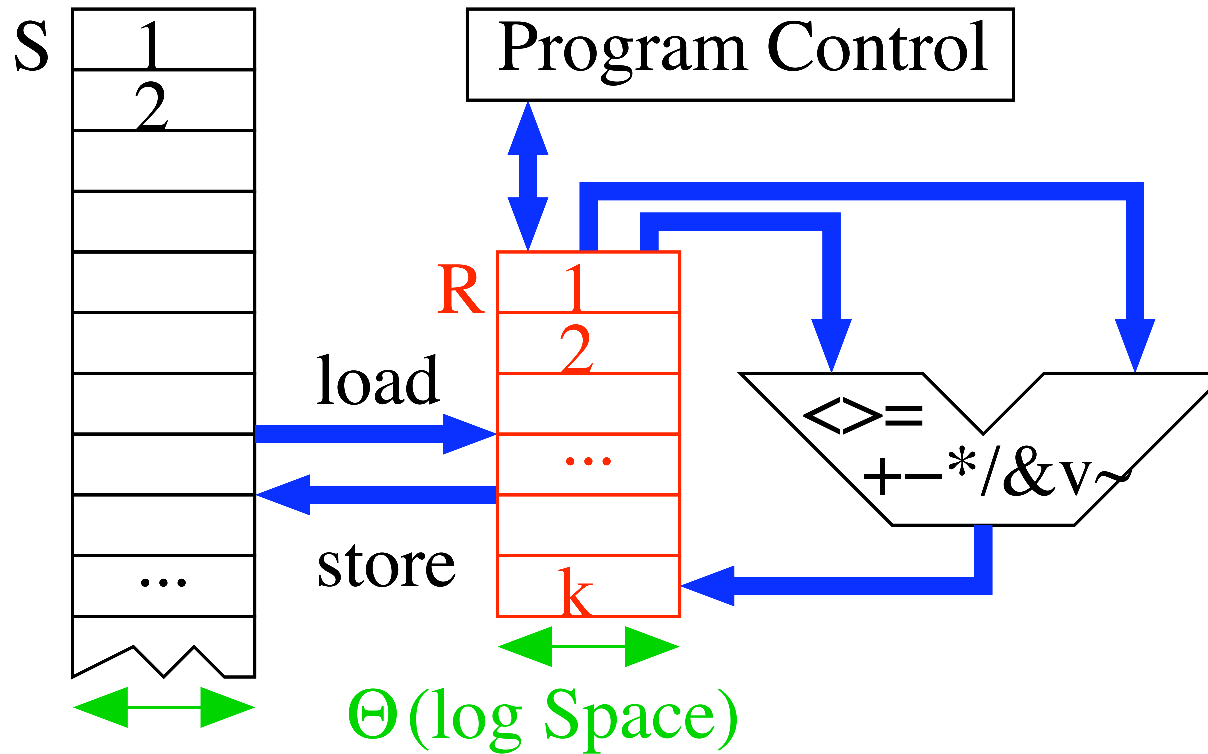
2.3 Maschinenmodell: RAM (Random Access Machine)



Moderne (RISC) Adaption des

von Neumann-Modells [von Neumann 1945]

Register

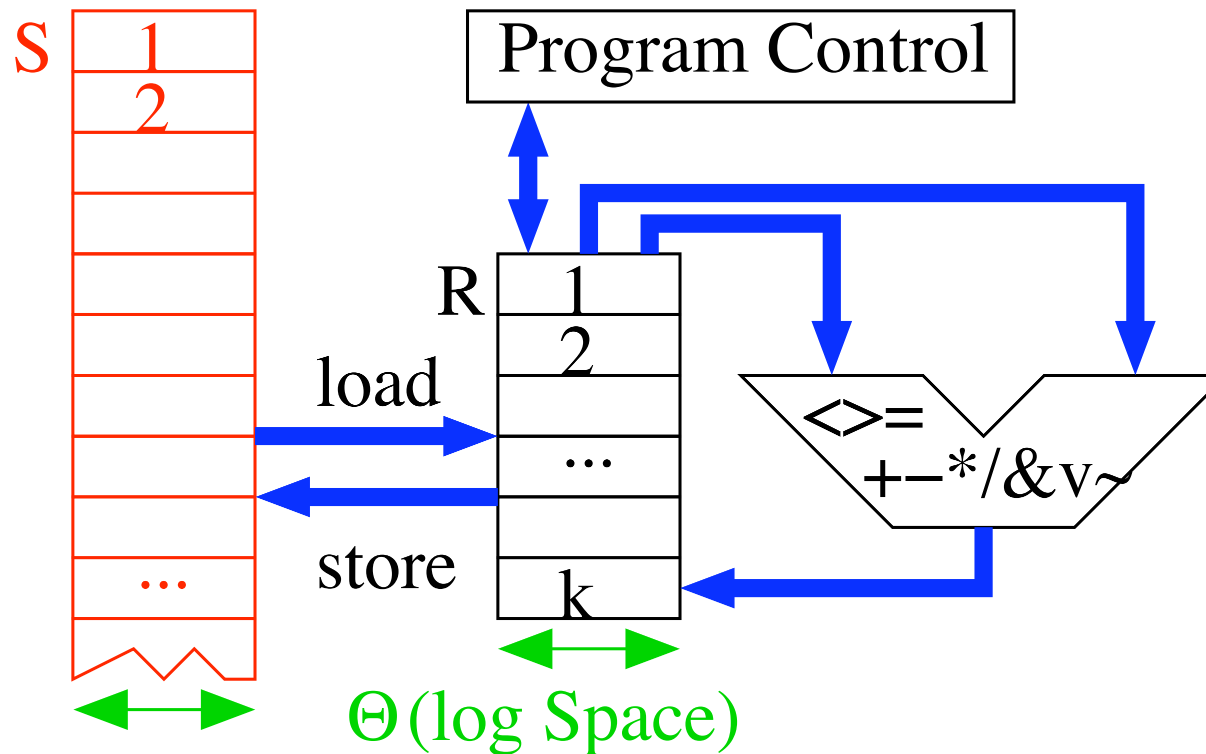


k (irgendeine Konstante) Speicher

$R_1,\ldots,R_k$ für

(kleine) ganze Zahlen

Hauptspeicher

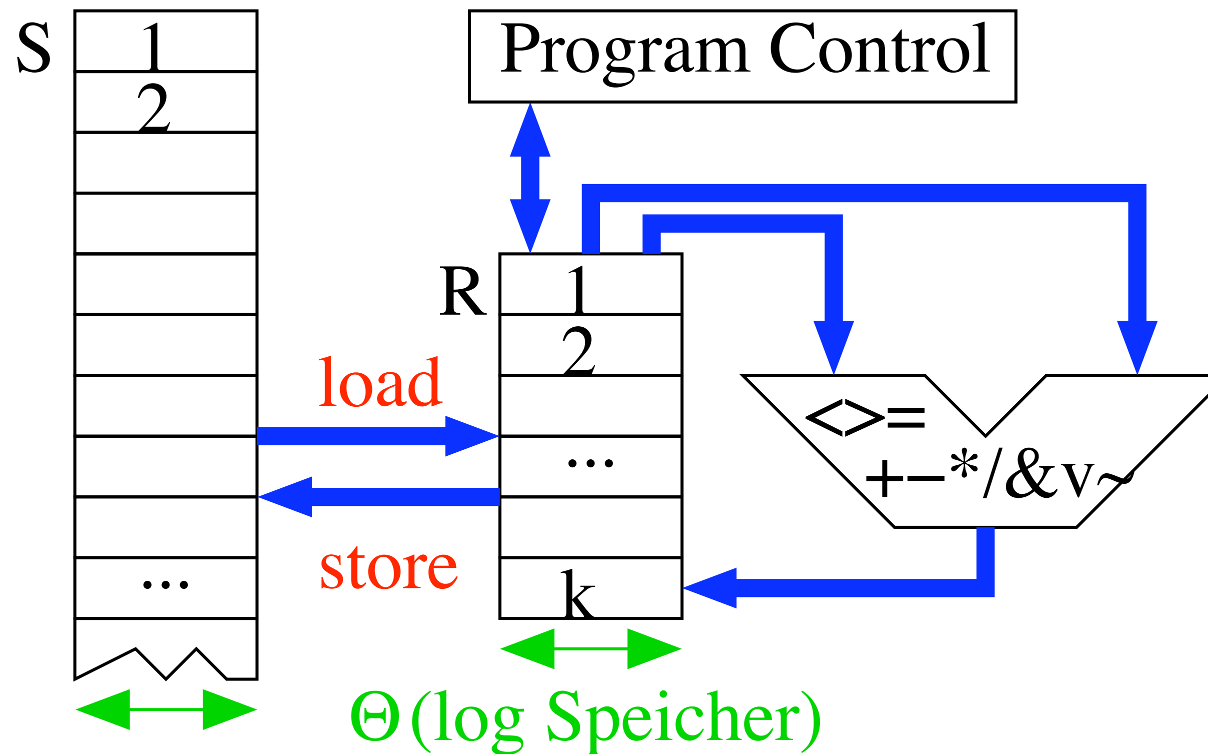


Unbegrenzter Vorrat an Speicherzellen

$S[1], S[2] \ldots$ für

(kleine) ganze Zahlen

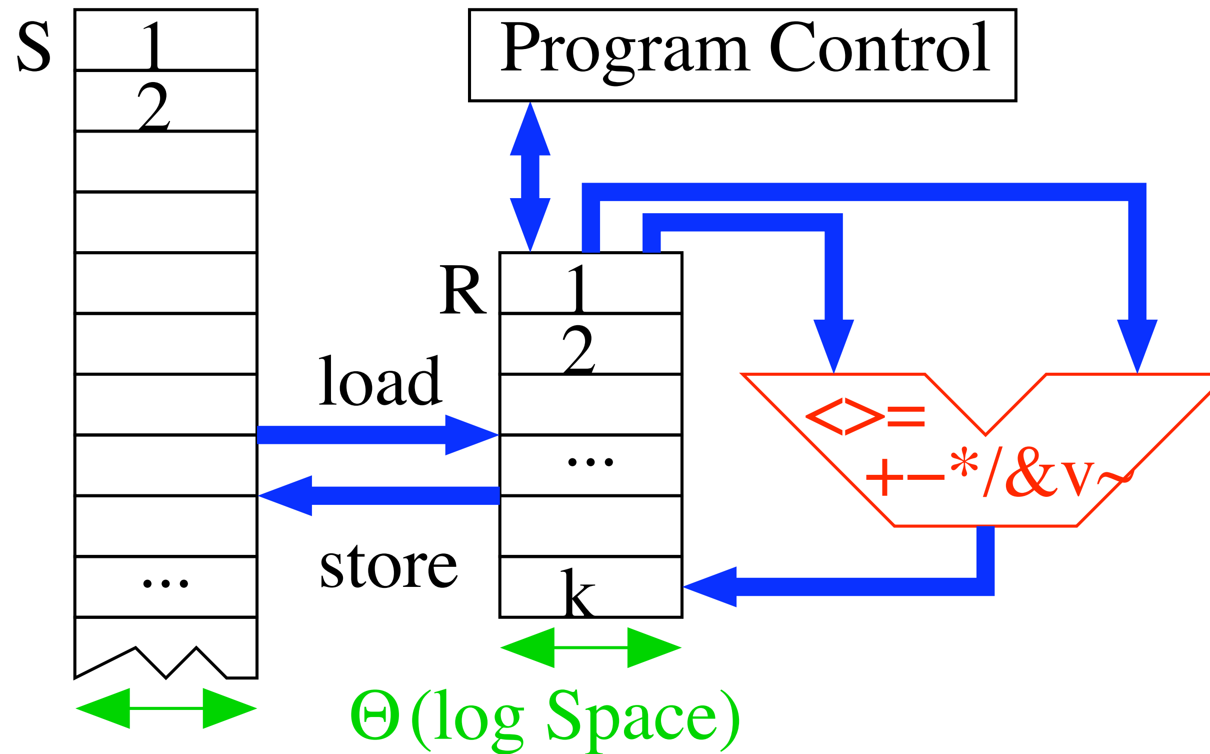
Speicherzugriff



$R_i := S[R_j]$ 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

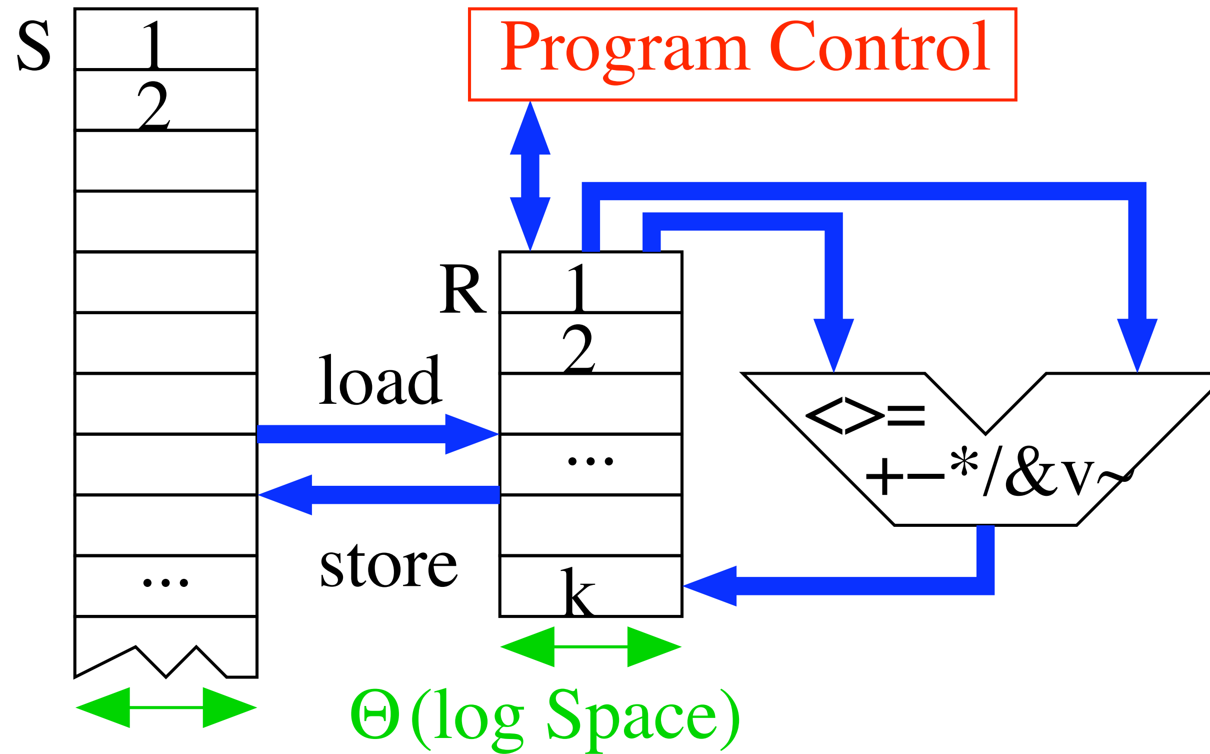


$R_i := R_j \odot R_\ell$ Registerarithmetik.

'$\odot$' 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 $\mathrm{O}(\cdot)$ 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

  $\leadsto$ kürzlich/häufig zugegriffene Daten sind eher im Cache

- blockweiser Zugriff

  $\leadsto$ Zugriff auf konsekutive Speicherbereiche sind schnell

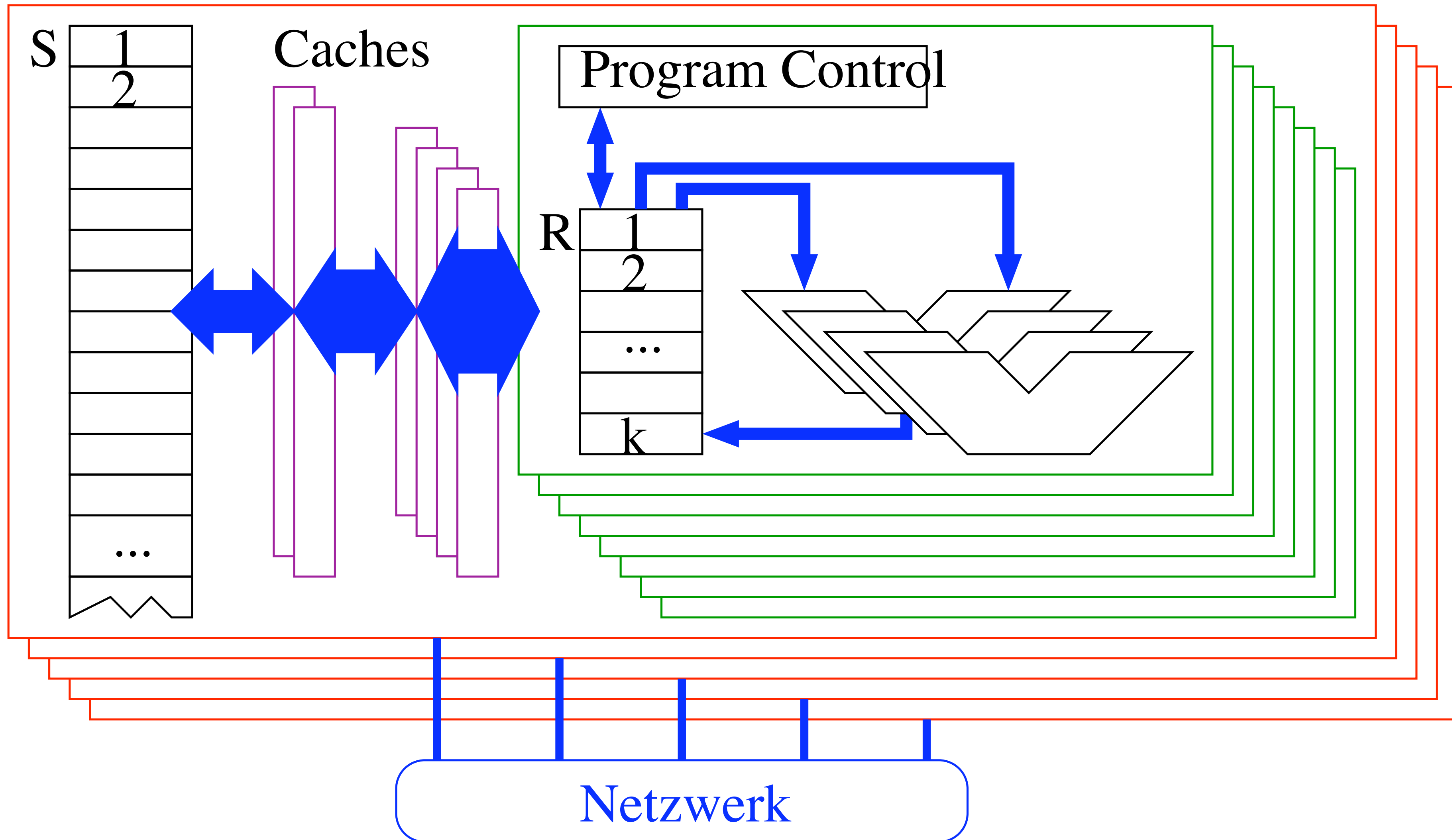
Parallelverarbeitung: Mehrere Prozessoren

$\leadsto$ unabhängige Aufgaben identifizieren

...

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

Mehr Maschinenmodell



2.4 Pseudocode

just in time

Beispiel

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

  Number r:= x

  Number i:= y

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

  Function add(c' : Complex) : Complex

    return Complex$(r + c'.r, i + c'.i)$

2.5 Design by Contract / Schleifeninvarianten

assert: Aussage über Zustand der Programmausführung

Vorbedingung: Bedingung für korrektes Funktionieren einer Prozedur

Nachbedingung: Leistungsgarantie einer Prozedur,
falls Vorbedingung erfüllt

invariant: 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.

2.6 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 $r = a^{n_0}$ // (*)$\wedge n = 0 \longrightarrow$Nachbedingung
  return r

Beispiel

```
Function power(a : ℝ; n_0 : ℕ) : ℝ
    assert n_0 ≥ 0 and ¬(a = 0 ∧ n_0 = 0)          // Vorbedingung
    p=a : ℝ;   r=1 : ℝ;   n=n_0 : 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 · p
        else (n, p):= (n/2, p · p)
    assert r = a^(n_0)                             // (*)∧n = 0 ⟶Nachbedingung
    return r
```

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

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 $r = a^{n_0}$ // (*)$\wedge n = 0 \longrightarrow$Nachbedingung

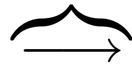
  return r

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

2.7 Programmanalyse

Die fundamentalistische Sicht: Ausgeführte RAM-Befehle zählen

einfache Übersetzungsregeln

Pseudo-Code  Maschinenbefehle

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

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

Rekursion $\rightsquigarrow$ Rekurrenzrelationen

2.7.1 Schleifenanalyse $\rightsquigarrow$ Summen ausrechnen

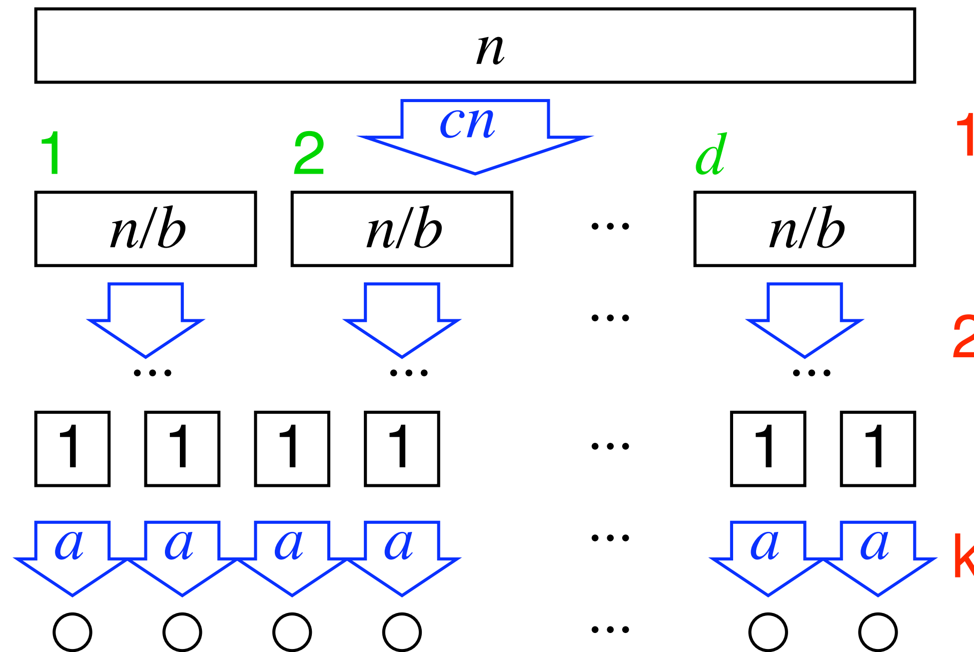
Das lernen Sie in Mathe

Beispiel: Schulmultiplikation

2.7.2 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

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

Beweisskizze

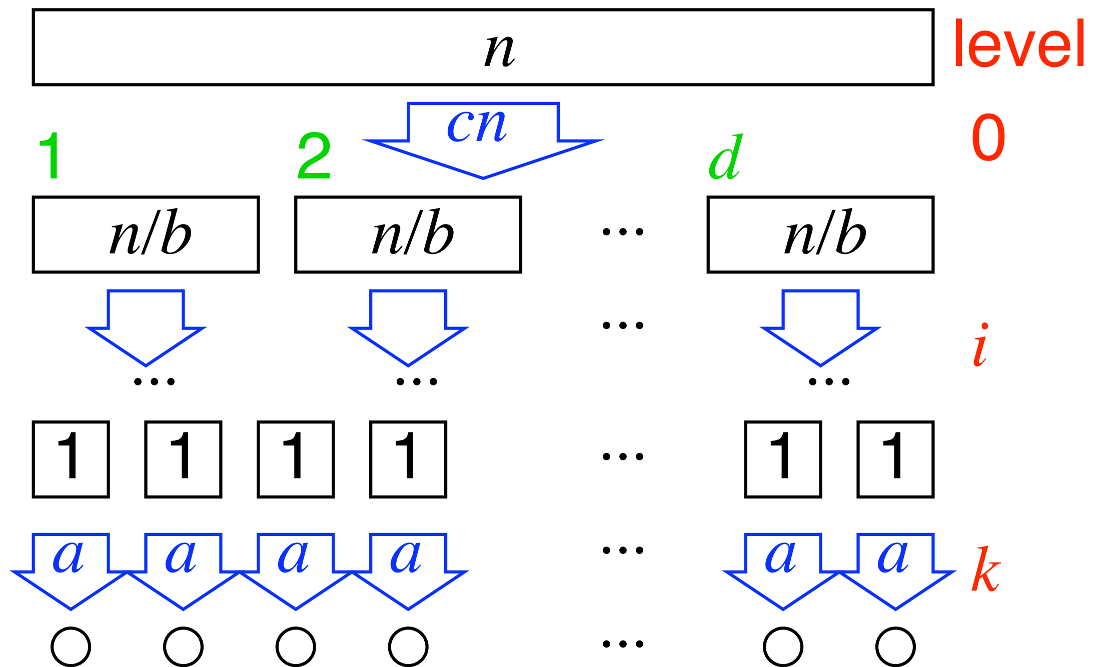
Auf Ebene i, haben wir d^i Probleme @ $n/b^i = b^{k-i}$

cost

cn

$$d^i \cdot c \cdot \frac{n}{b^i} = cn\left(\frac{d}{b}\right)^i$$

ad^k

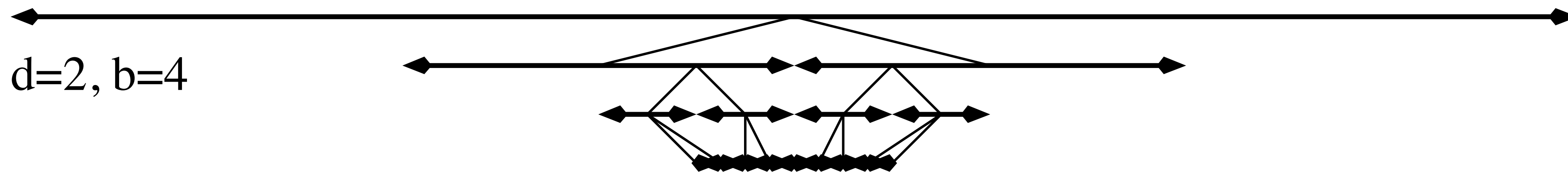


Beweisskizze Fall $d < b$

geometrisch schrumpfende Reihe

$\longrightarrow$ erste Rekursionsebene kostet konstanten Teil der Arbeit

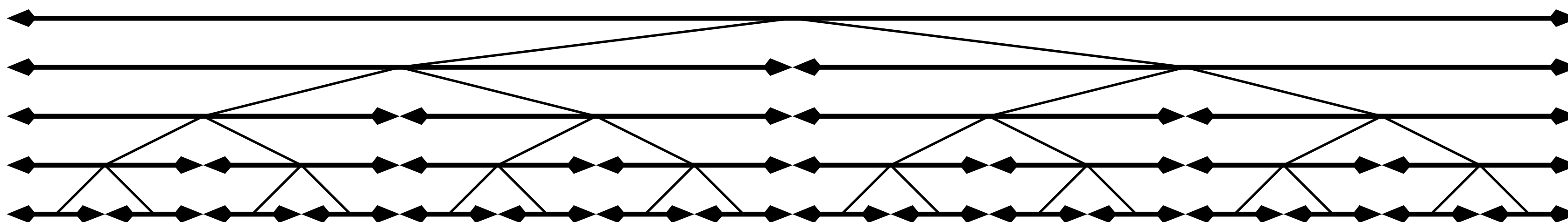
$$r(n) = \underbrace{a \cdot d^k}_{o(n)} + cn \cdot \underbrace{\sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i}_{O(1)} = \Theta(n)$$



Beweisskizze Fall $d = b$

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

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



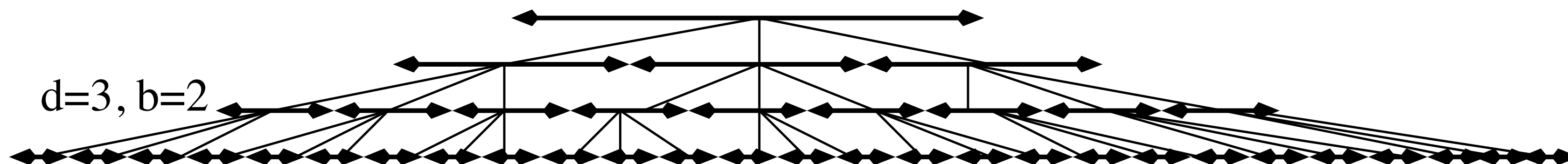
Beweisskizze Fall $d > b$

geometrisch wachsende Reihe

$\longrightarrow$ letzte Rekursionsebene kostet konstanten Teil der Arbeit

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

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

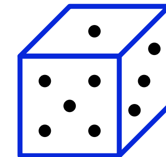
$d = b$: mergesort, quicksort

$d > b$: Schulmultiplikation, Karatsuba-Ofman-Multiplikation

2.8 Analyse im Mittel

später an Beispielen

2.9 Randomisierte Algorithmen

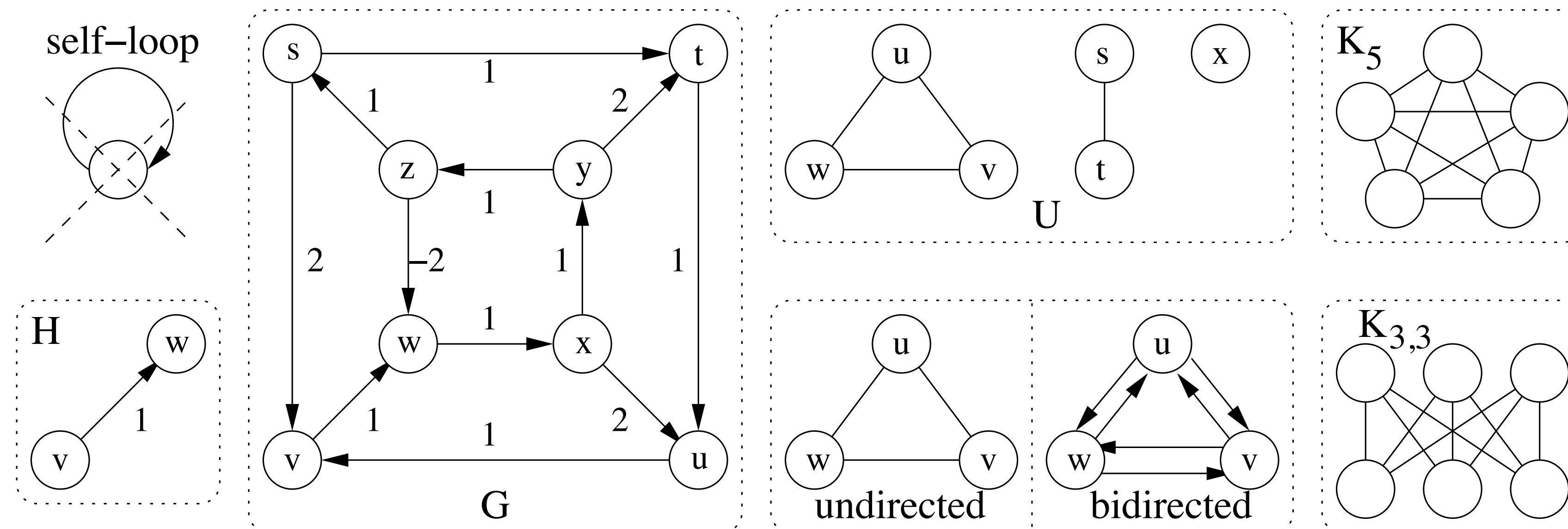


später an Beispielen

2.10 Graphen

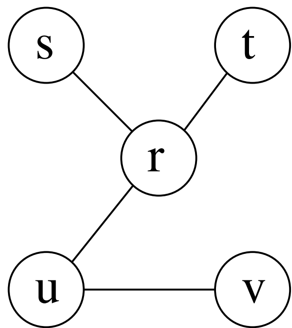
Sie kennen schon (?): Relationen, Knoten, Kanten, (un)gerichtete Graphen, Kantengewichte, Knotengrade, Kantengewichte, knoteninduzierte Teilgraphen.

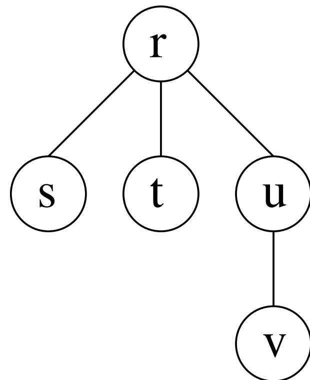
Pfade (einfach, Hamilton-), Kreise, DAGs

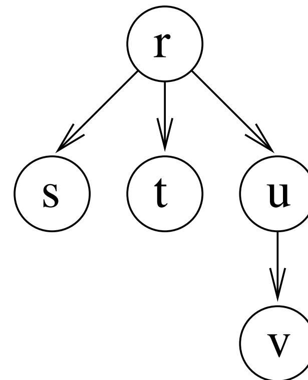


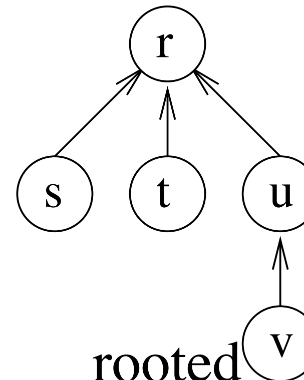
Bäume

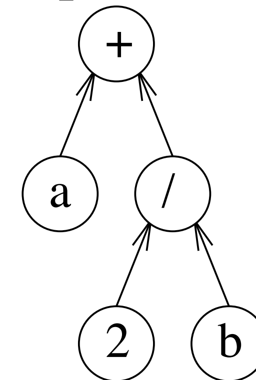
Zusammenhang, Bäume, Wurzeln, Wälder, Kinder, Eltern, . . .











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 $\exists v \in V$: 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 $\mathrm{O}(|V| + |E|)$.

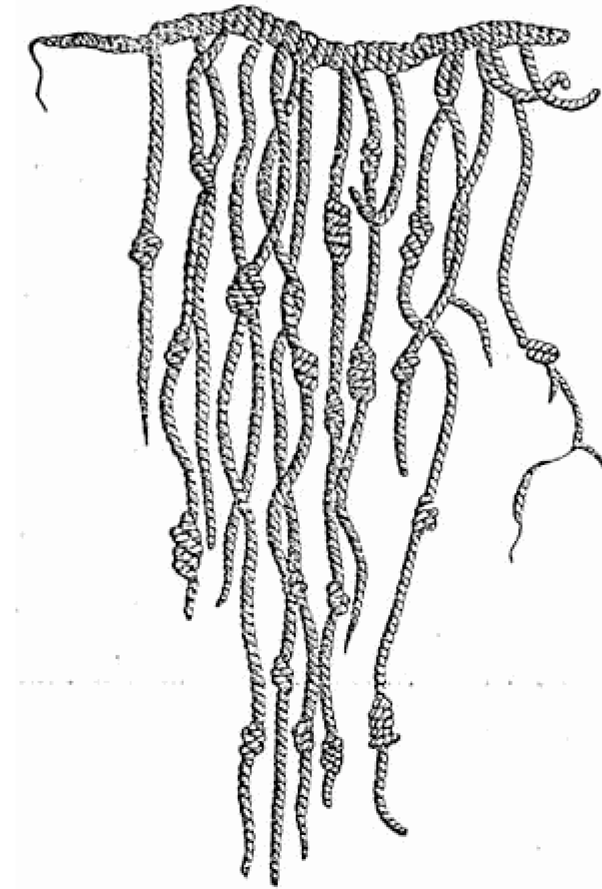
2.11 P und NP

das kommt in "Grundlagen der theoretischen 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/hart), bei denen es SEHR überraschend wäre, wenn sie in Polynomialzeit lösbar wären.

3 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 $\langle 2,3,5,7,9,11,\ldots\rangle$ — Mathe
- Funktionalität (stack, ...) — Softwaretechnik
- Repräsentation — 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 '$*$'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	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

3.1 Verkettete Listen

3.1.1 Doppelt verkettete Listen

Listenglieder (Items)

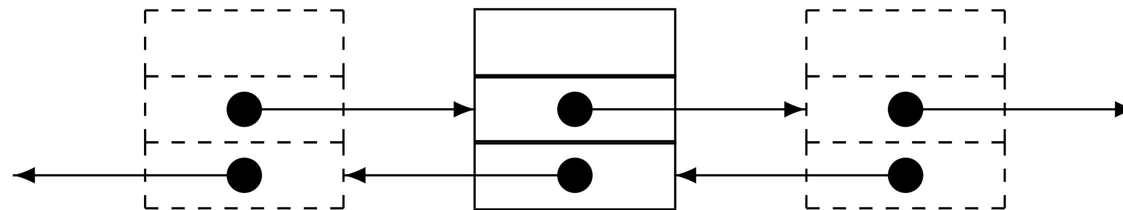
Class Handle = **Pointer to** Item

Class Item **of** Element // one link in a doubly linked list

e : Element

next : Handle //

prev : Handle



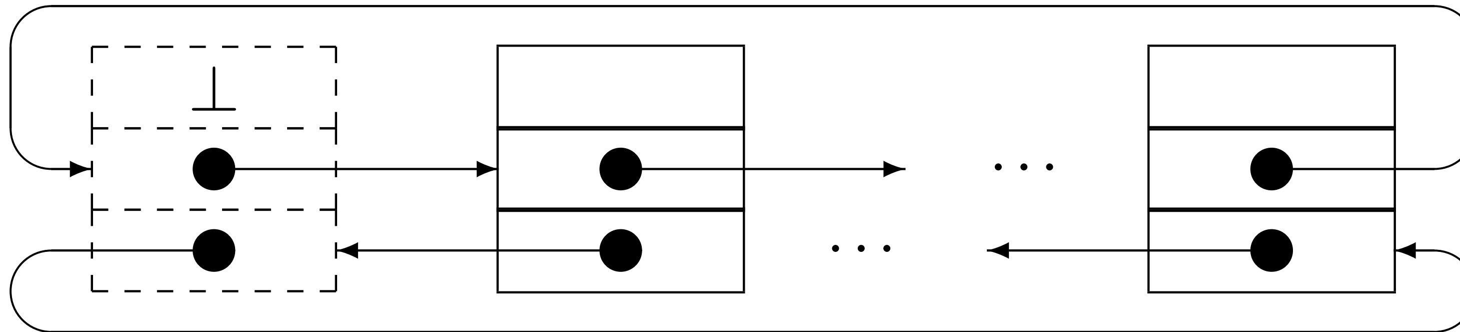
invariant next→prev=prev→next=**this**

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)

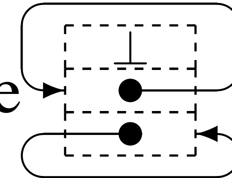
Die Listenklasse

Class List **of** Element

// Item h is the predecessor of the first element

// and the successor of the last element.

$h = \begin{pmatrix} \perp \\ \textbf{this} \\ \textbf{this} \end{pmatrix}$: Item // init to empty sequence



// Simple access functions

Function head : Handle; **return address of** h// Pos. before any proper ele

Function isEmpty : $\{0,1\}$; **return** h.next = **this** // $\langle\rangle$?

Function first : Handle; **assert** ¬isEmpty; **return** h.next

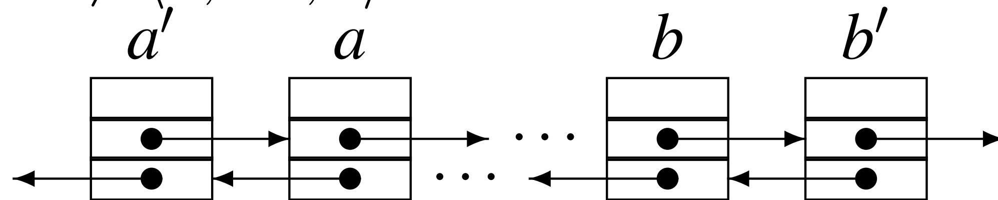
Function last : Handle; **assert** ¬isEmpty; **return** h.prev

⋮

Procedure splice(a,b,t : Handle)//Cut out $\langle a,\ldots,b\rangle$ and insert after t

assert b is not before $a \wedge t \notin \langle a,\ldots,b\rangle$

//Cut out $\langle a,\ldots,b\rangle$

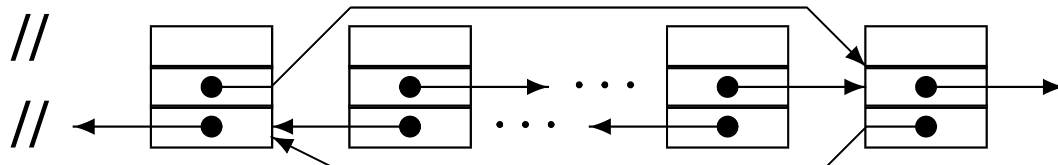


$a' := a \rightarrow$prev

$b' := b \rightarrow$next

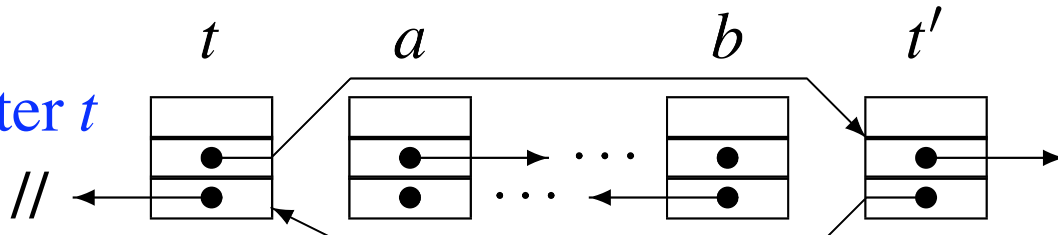
$a' \rightarrow$next $:= b'$ //

$b' \rightarrow$prev $:= a'$ //



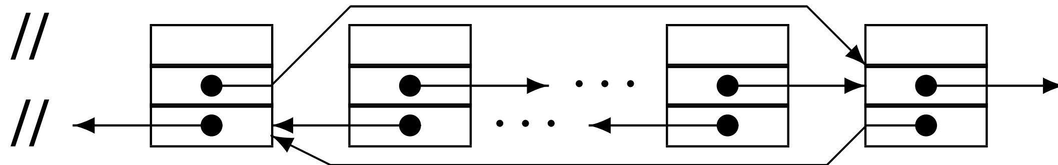
//insert $\langle a,\ldots,b\rangle$ after t

$t' := t \rightarrow$next //



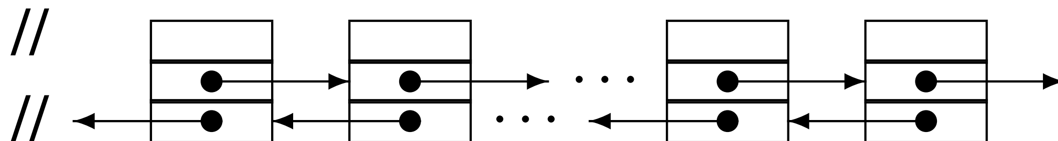
$b \rightarrow$next $:= t'$ //

$a \rightarrow$prev $:= t$ //



$t \rightarrow$next $:= a$ //

$t' \rightarrow$prev $:= b$ //



Der Rest sind Einzeiler (?)

// Moving elements around within a sequence.

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

Procedure moveAfter(b, a' : Handle) splice(b, b, a')

Procedure moveToFront$(b$: Handle$)$ moveAfter$(b,$ head$)$

Procedure moveToBack$(b$: Handle$)$ moveAfter$(b,$ last$)$

Oder doch nicht? Speicherverwaltung!

naiv / blauäugig /optimistisch:

Speicherverwaltung der Programmiersprache

$\rightsquigarrow$ 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

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

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

Procedure popFront remove(first)

Procedure popBack remove(last)

Elemente einfügen

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

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 at 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

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

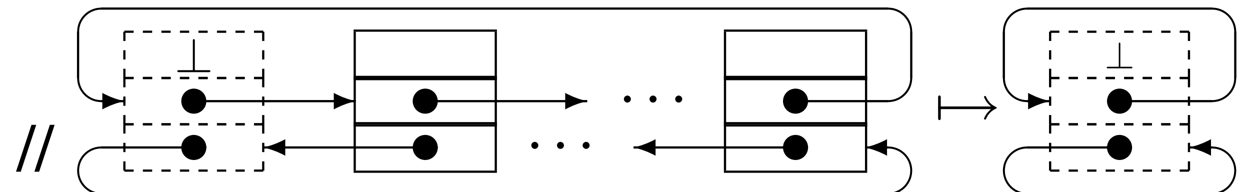
Procedure concat$(L'$: List$)$

splice(L'.first, L'.last, last)

// $\langle a,\ldots,b\rangle \mapsto \langle\rangle$

Procedure makeEmpty

freeList.concat(**this**) //



Das geht in konstanter Zeit – unabhängig von der Listenlänge!

Suchen

Trick: gesuchtes Element in Dummy-Item schreiben:

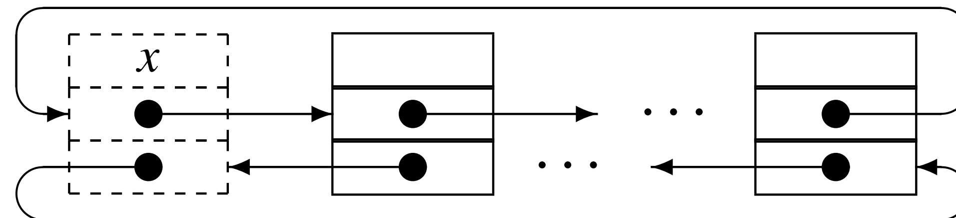
Function findNext(x : Element; from : Handle) : Handle

$h.e = x$ // 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 $\leftrightarrow$ Effizienz

Beispiel: Listenlängen

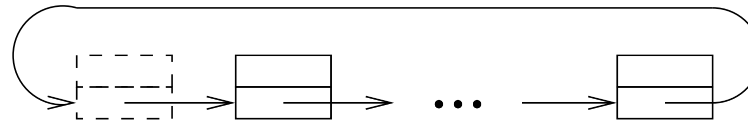
Verwalte zusätzliches Member size.

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

Die Moral von der Geschicht:

Es gibt nicht DIE Listenimplementierung.

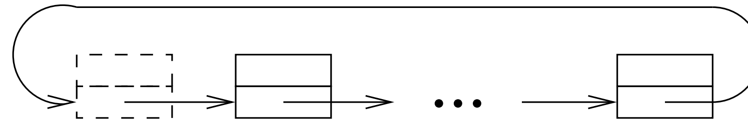
3.1.2 Einfach verkettete Listen



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

$E = \{(u,v) : u \in \text{Item}, v = u \rightarrow \text{next}\}$

- u.next zeigt immer auf ein Item
- $\forall u \in Item$: $\text{indegree}_G(u) = 1$. Wohl definiert obwohl nicht unbedingt leicht zu testen.

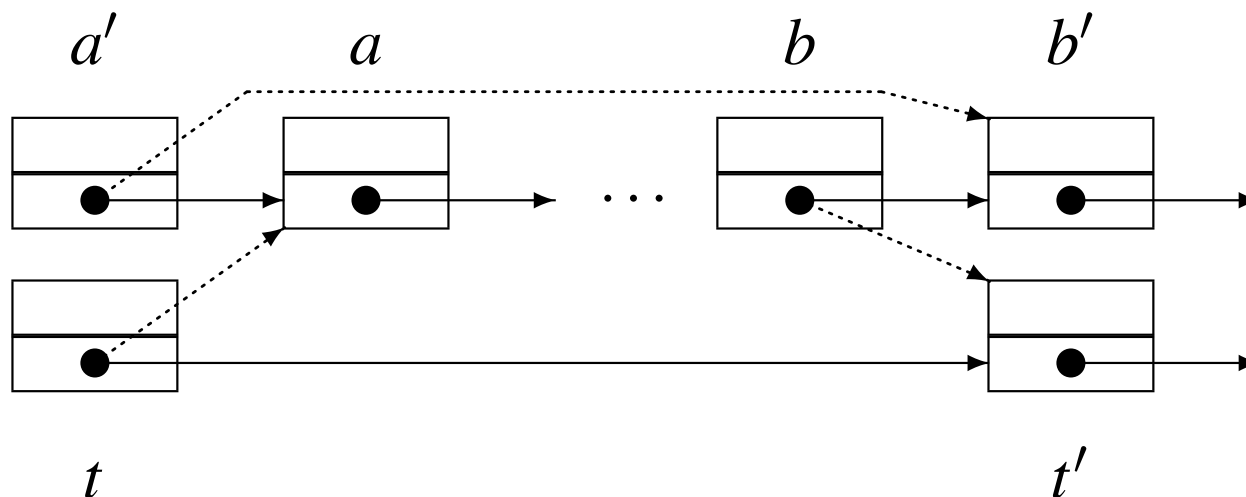
Folge: Items bilden Kollektion von Kreisen

Einfach verkettete Listen – splice

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

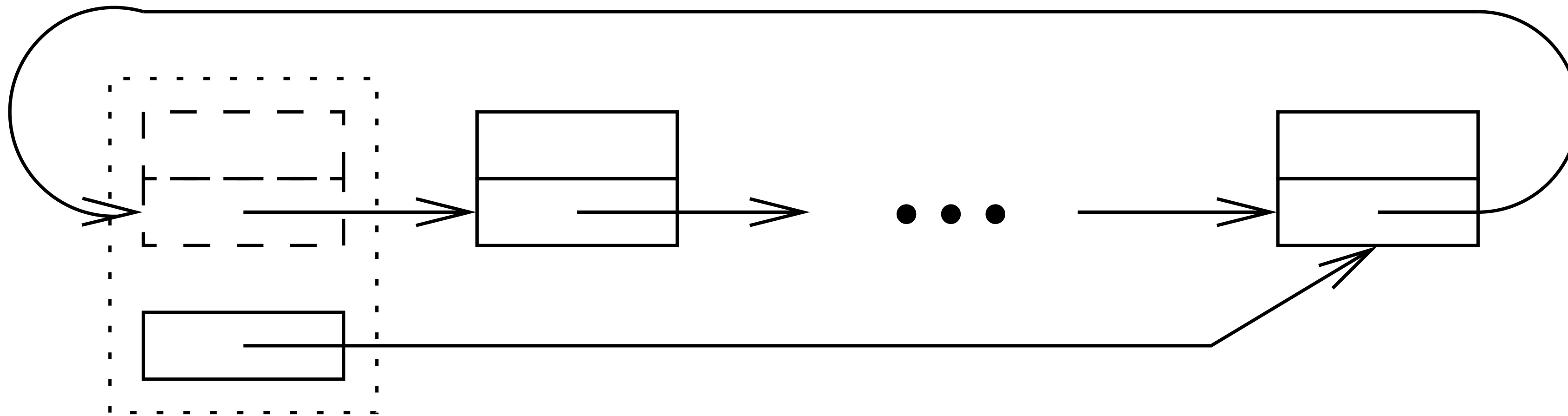
Procedure splice(a',b,t : SHandle)

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



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)

$A[i] = a_i$ falls $A = \langle a_1, \ldots, a_n \rangle$

Beschränkte Felder (Bounded Arrays)

Eingebaute Datenstruktur: Ein Stück Hauptspeicher $+$

Adressrechnung

Größe muss von Anfang an bekannt sein

3.2 Unbeschränkte Felder (Unbounded Arrays)

$$\langle e_0, \ldots, e_n \rangle.\text{pushBack}(e) \leadsto \langle e_0, \ldots, e_n, e \rangle,$$

$$\langle e_0, \ldots, e_n \rangle.\text{popBack} \leadsto \langle e_0, \ldots, e_{n-1} \rangle,$$

$$\text{size}(\langle e_0, \ldots, e_{n-1} \rangle) = n \ .$$

Unbeschränke Felder – Anwendungen

wenn man nicht weiß, wie lang das Feld wird.

Beispiele:

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

Unbeschränke 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

$\mathrm{O}(\sum_{i=1}^{n} i) = \mathrm{O}\left(n^2\right)$ Geht es schneller?

Unbeschränke Felder mit teilweise ungenutztem Speicher

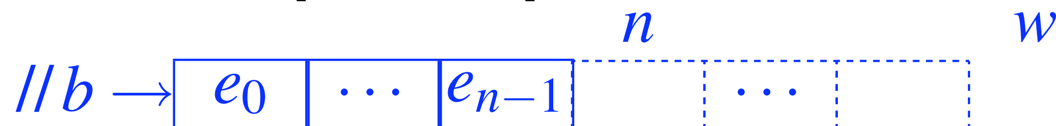
Class UArray **of** Element

w=1 : $\mathbb{N}$ // allocated size

n=0 : $\mathbb{N}$ // current size.

invariant $n \leq w < \alpha n$ or $n = 0$ and $w \leq 2$

b : **Array** $[0..w-1]$ **of** Element



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

assert $0 \leq i < n$

return $b[i]$

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

Procedure pushBack(e : Element) // Example for $n = w = 4$:

    if $n = w$ **then** // $b \rightarrow$ [0|1|2|3]

        reallocate($2n$) // $b \rightarrow$ [0|1|2|3| | | |]

    $b[n] := e$ // $b \rightarrow$ [0|1|2|3|e| | |]

    $n{+}{+}$ // $b \rightarrow$ [0|1|2|3|e| | |]

Procedure reallocate(w' : $\mathbb{N}$) // Example for $w = 4$, $w' = 8$:

    $w := w'$ // $b \rightarrow$ [0|1|2|3]

    $b' :=$ **allocate**

        Array $[0..w'-1]$ **of** Element // $b' \rightarrow$ [| | | | | | |]

    $(b'[0], \ldots, b'[n-1]) :=$

        $(b[0], \ldots, b[n-1])$ // $b' \rightarrow$ [0|1|2|3| | | |]

    dispose b // $b \rightarrow$ [~~0|1|2|3~~]

    $b := b'$ // 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?

3.2.1 Amortisierte Komplexität unbeschr. Felder

Sei u ein anfangs leeres, unbeschränktes Feld.

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

von pushBack oder popBack Operationen auf u

wird in Zeit $O(m)$ ausgeführt.

Sprechweise:

pushBack und popBack haben amortisiert konstante Ausführungszeit

—

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

Beweis: Konto-Methode (oder Versicherung)

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

Zu zeigen: keine Überziehungen

Erster Aufruf von reallocate: kein Problem

Beweis: Konto-Methode (oder Versicherung)

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

Weitere Aufrufe von reallocate:

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

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

$\square$

3.3 Amortisierte Analyse – allgemeiner

- Z: Menge von Operationen, z. B. $\{$pushBack, popBack$\}$
- s: Zustand der Datenstruktur
- $A_X(s)$: amortisierte Kosten von Operation $X \in Z$ in Zustand s
- $T_X(s)$: tatsächliche Kosten von Operation $X \in Z$ in Zustand s
- Berechnung: $s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \cdots \xrightarrow{Op_n} s_n$

Die angenommenen amortisierten Kosten sind korrekt, wenn

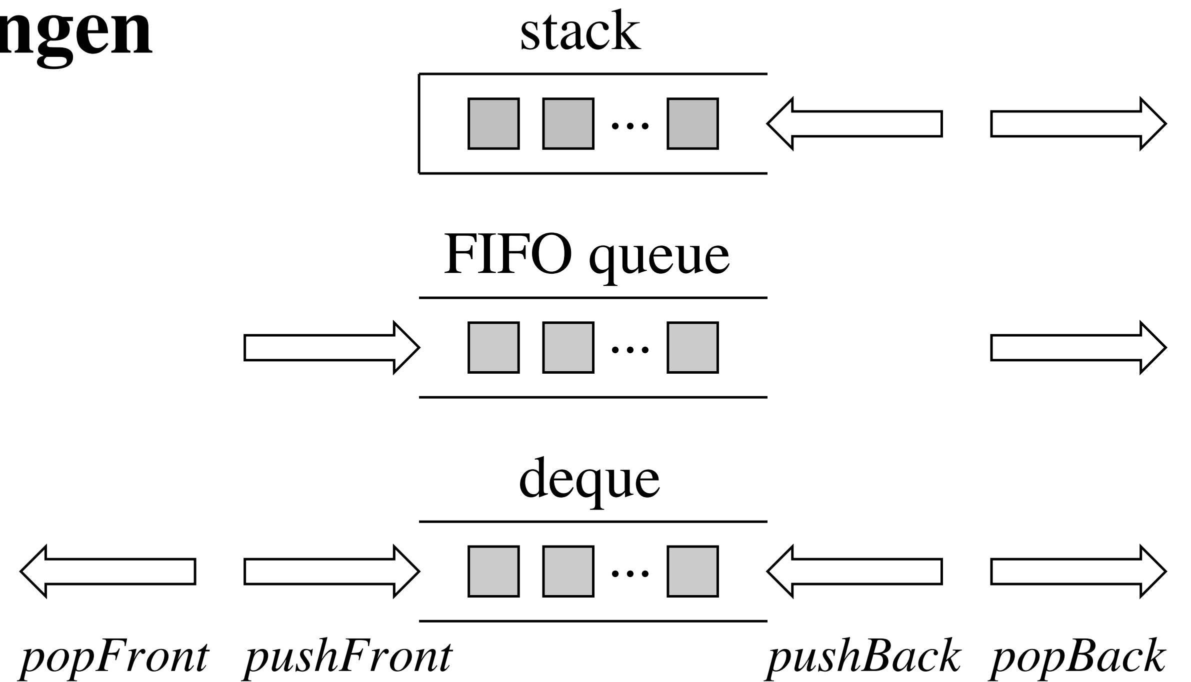
$$\underbrace{\sum_{1 \le i \le n} T_{Op_i}(s_{i-1})}_{\text{tatsächliche Gesamtkosten}} \le c + \underbrace{\sum_{1 \le i \le n} A_{Op_i}(s_{i-1})}_{\text{amortisierte Gesamtkosten}}$$

für eine Konstante c

Amortisierte Analyse – Diskussion

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

3.4 Stapel und Schlangen



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

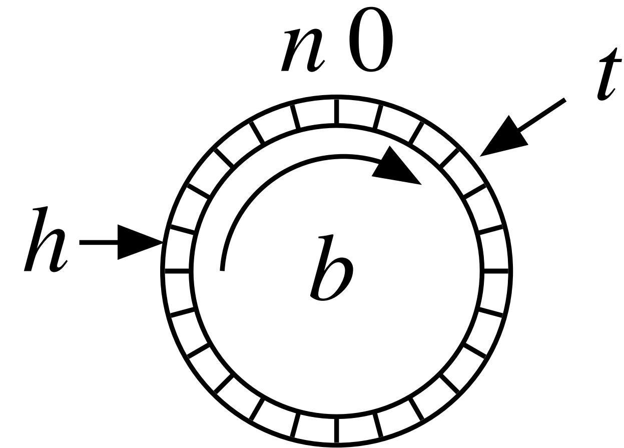
Class BoundedFIFO($n : \mathbb{N}$) **of** Element

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

h=0 : $\mathbb{N}$

t=0 : $\mathbb{N}$

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



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

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

Procedure pushBack(x : Element)

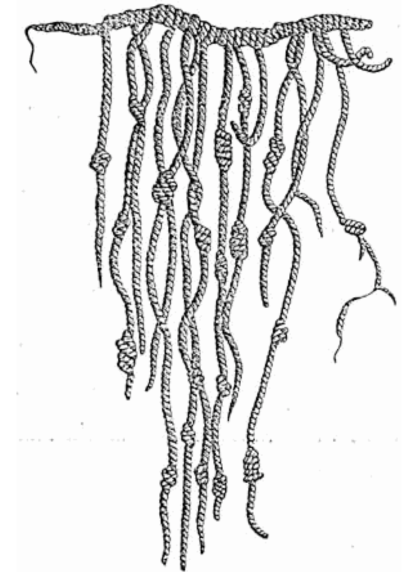
  assert size$< n$

  $b[t]$:= x

  t := $(t + 1)$ **mod** $(n + 1)$

Procedure popFront **assert** ¬isEmpty; h := $(h + 1)$ **mod** $(n + 1)$

3.5 Vergleich: Listen – Felder



Vorteile von Listen

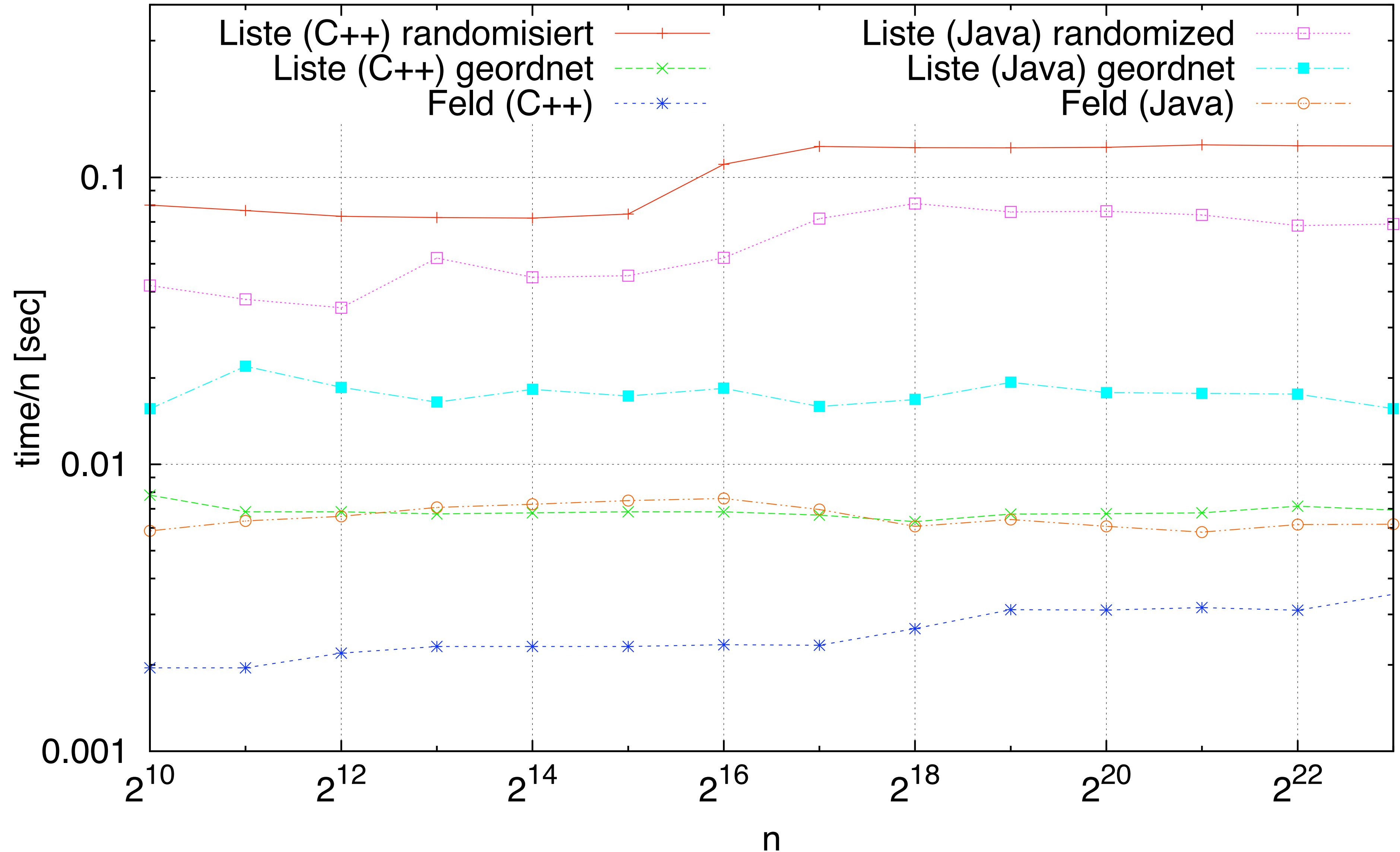
- flexibel
- remove, splice,. . .
- kein Verschnitt

Vorteile von Feldern

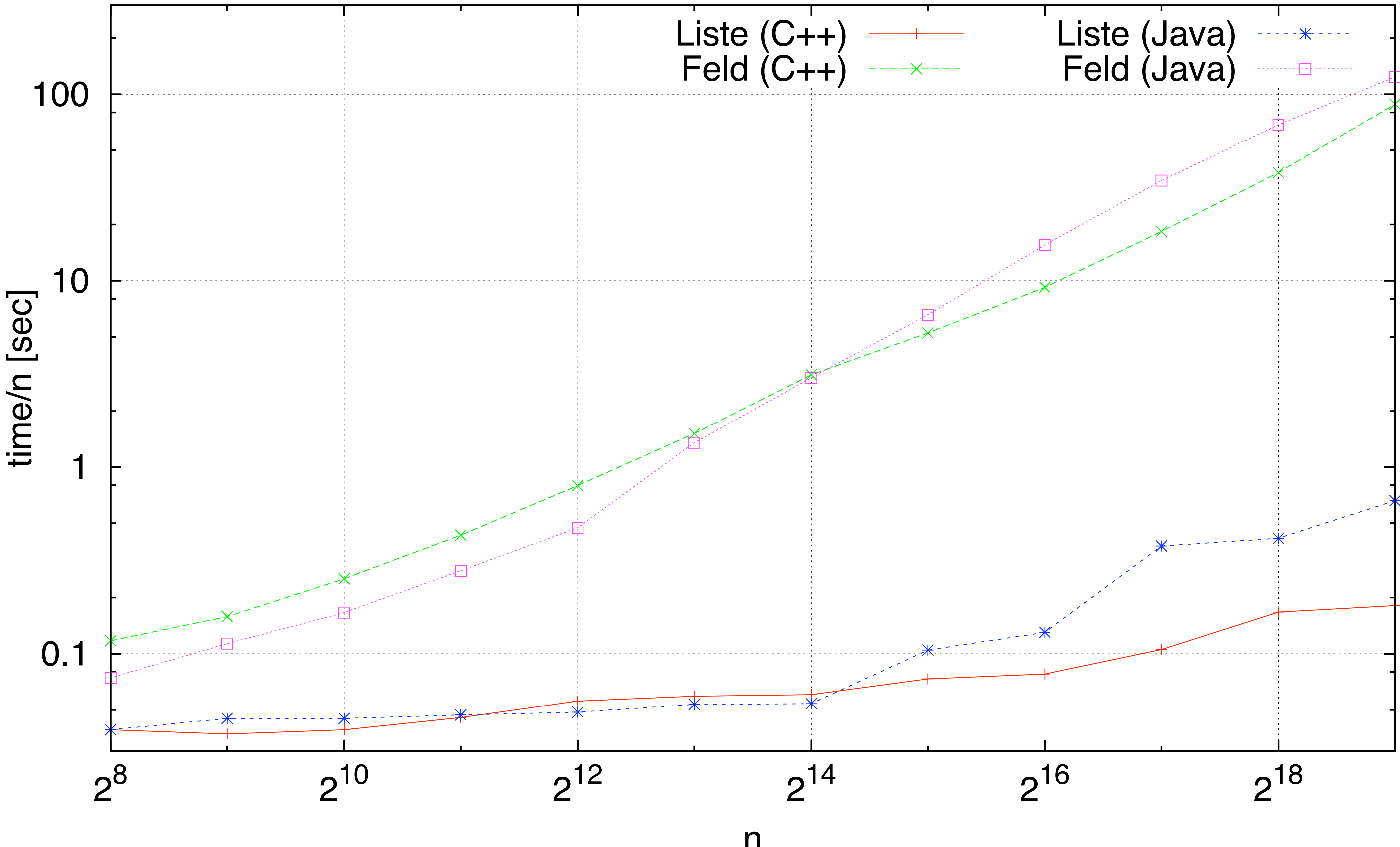
- beliebiger Zugriff
- einfach
- kein Overhead für Zeiger
- Cache-effizientes scanning

Operation	List	SList	UArray	CArray	explanation '$*$'
$[\cdot]$	n	n	1	1	
$\|\cdot\|$	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

Iterieren



Einfügen an zufälliger Position



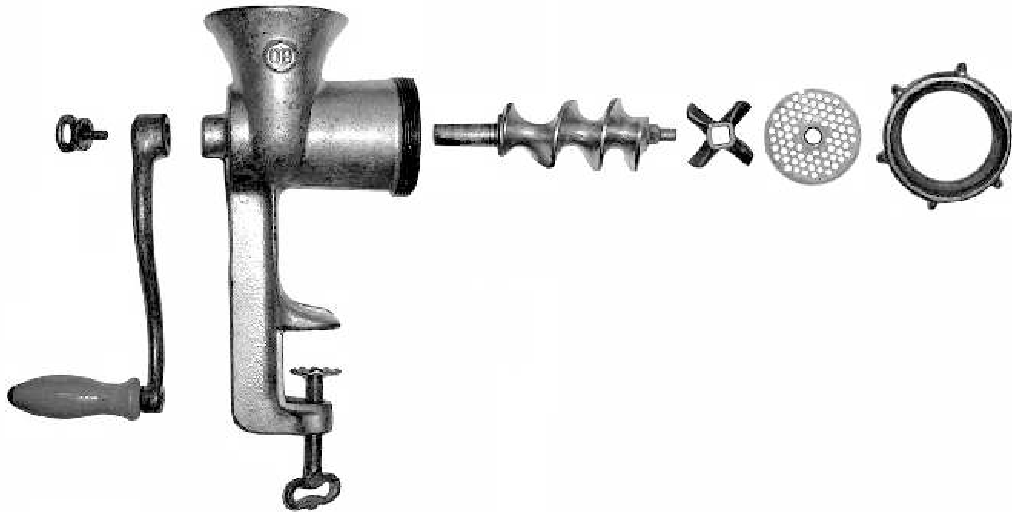
Ausblick: Weitere Repräsentationen von Folgen

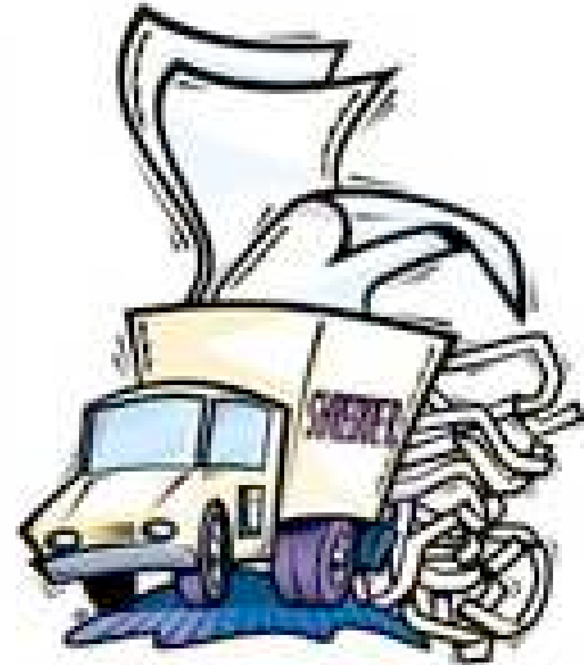
Hashtabellen: schnelles Einfügen, Löschen und Suchen — Kapitel 4

Prioritätslisten: schnelles Einfügen, Minimum Entfernen — Kapitel 6

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

4 Hashing (Streuspeicherung)





“to hash” ≈ “völlig durcheinander bringen”.

Paradoxerweise hilft das, Dinge wiederzufinden

Hashtabellen

speichere Menge $M \subseteq$ Element.

key(e) ist eindeutig für $e \in M$.

unterstütze Wörterbuch-Operationen in Zeit $\mathrm{O}(1)$.

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

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

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

Exkurs: Konventionen für Elemente

Viele Datenstrukturen repräsentieren Mengen

(engl. auch collection classes).

Die Mengenelemente e haben Schlüssel key(e).

Elementvergleich hier gleichbedeutend mit Schlüsselvergleich.

$e < / > / = e'$ gdw. key$(e) < / > / =$ key(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 awk
- Datenbank-Gleichheits-Join
  (wenn eine Tabelle in den Speicher passt)
- Unsere Routenplaner: Teilmengen von Knoten,
  z. B. Suchraum
- ...

4.1 Überblick

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

Ein (über)optimistischer Ansatz

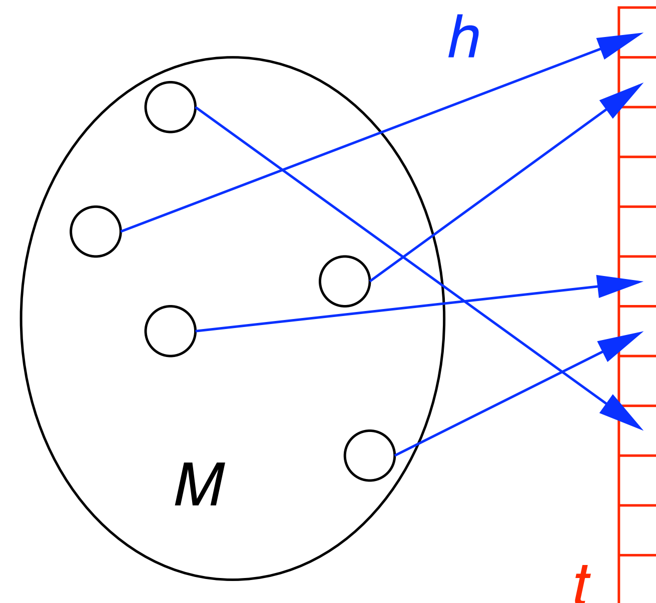
Eine perfekte Hash-Funktion h

bildet Elemente von M injektiv

auf eindeutige Einträge

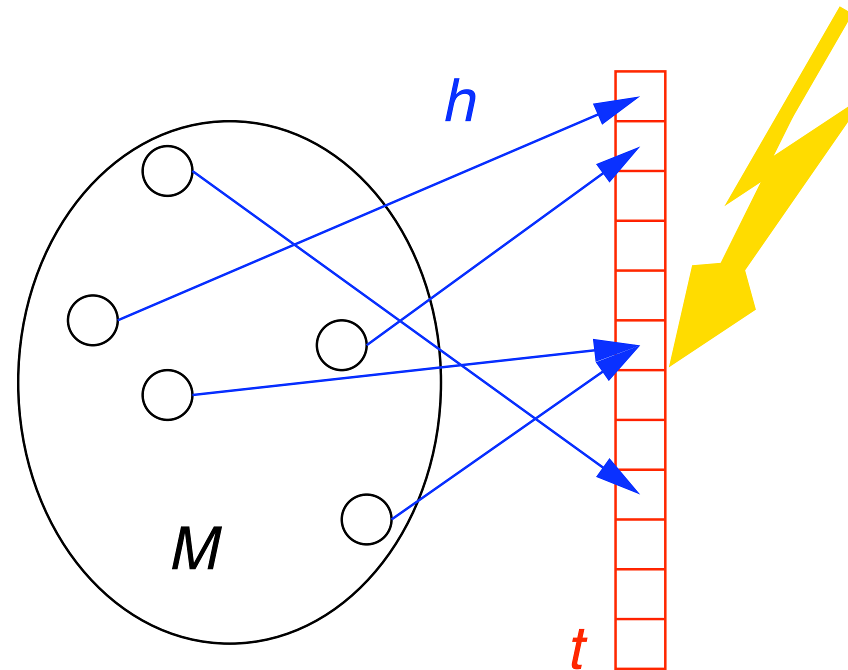
der Tabelle $t[0..m-1]$ ab, d. h.,

$t[h(\text{key}(e))] = e$



Kollisionen

Perfekte Hash-Funktionen sind schwer zu finden

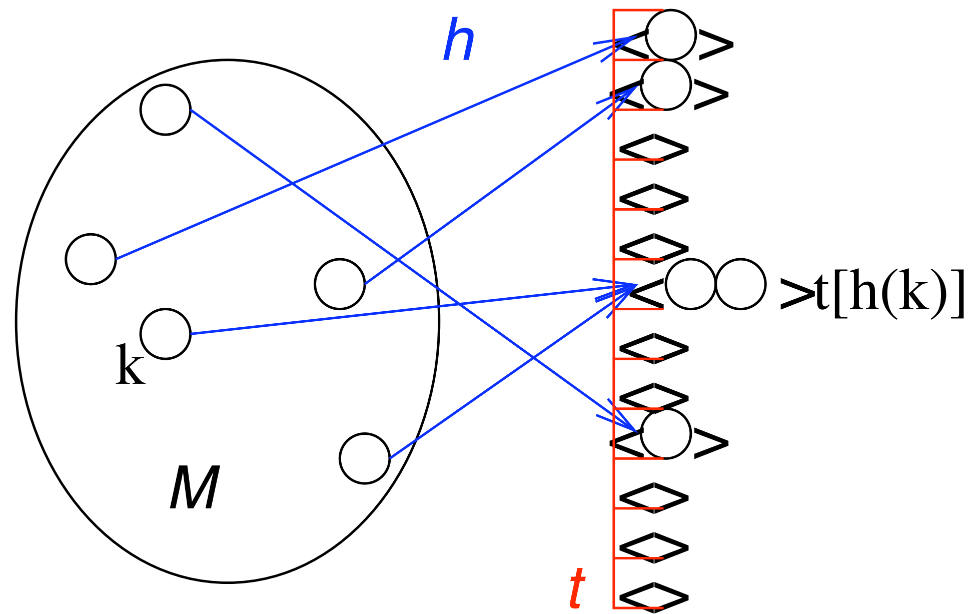


Beispiel: Geburtstagsparadox

Kollisionsauflösung

Beispiel geschlossenes Hashing

Tabelleneinträge: Elemente $\rightsquigarrow$ Folgen von Elementen



4.2 Hashing mit verketteten Listen

Implementiere die Folgen beim geschlossenen Hashing

durch einfach verkettete Listen

insert(e): Füge e am Anfang von $t[h(e)]$ ein.

remove(k): Durchlaufe $t[h(k)]$.

Element e mit $h(e) = k$ gefunden?

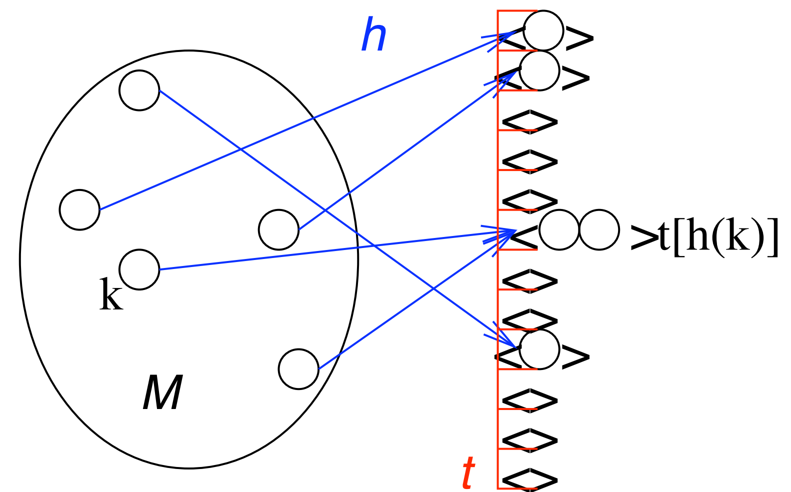
$\rightsquigarrow$ löschen und zurückliefern.

find(k) : Durchlaufe $t[h(k)]$.

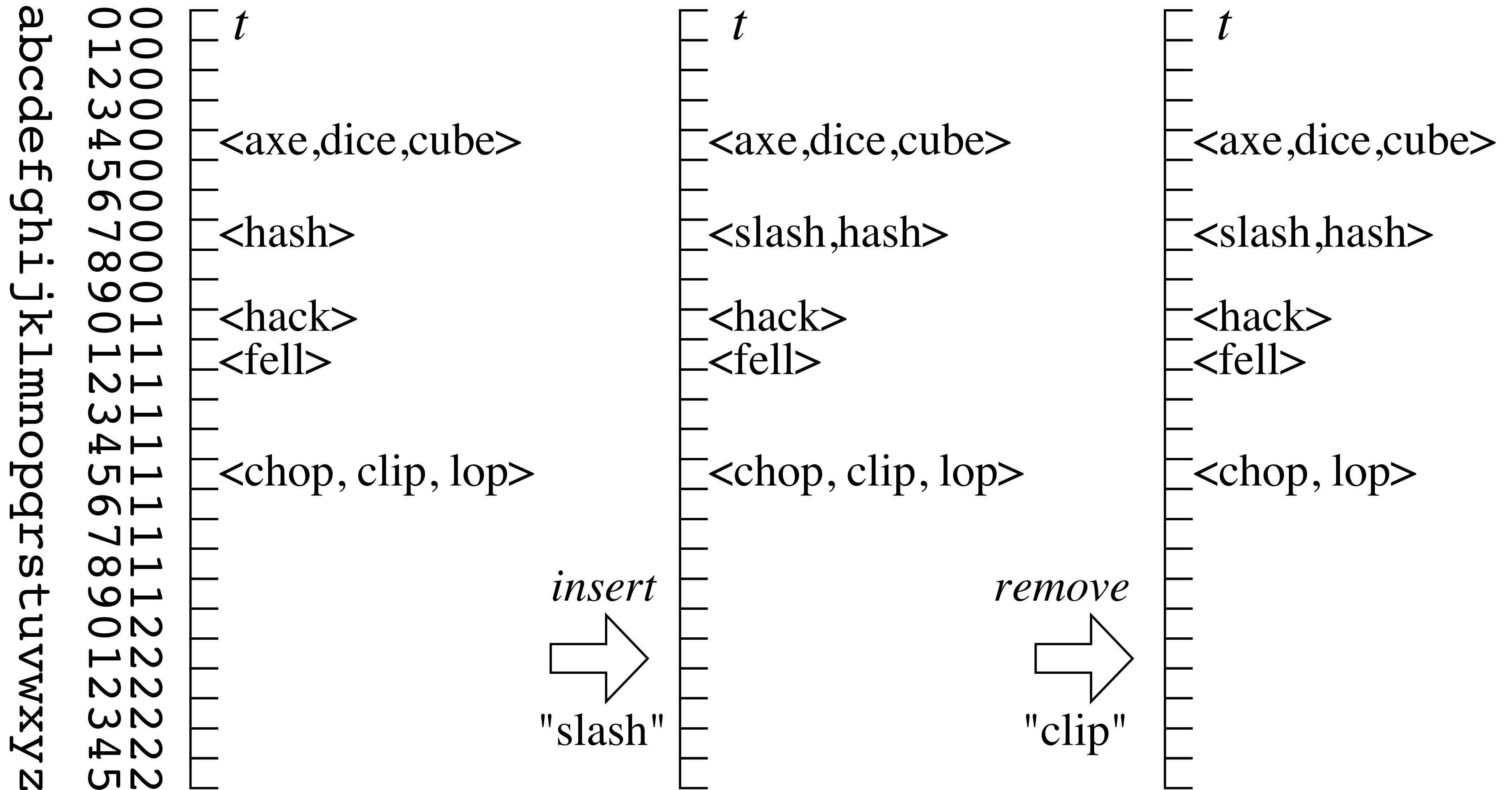
Element e mit $h(e) = k$ gefunden?

$\rightsquigarrow$ zurückliefern.

Sonst: $\perp$ zurückgeben.



Beispiel



Analyse

insert(e): konstante Zeit

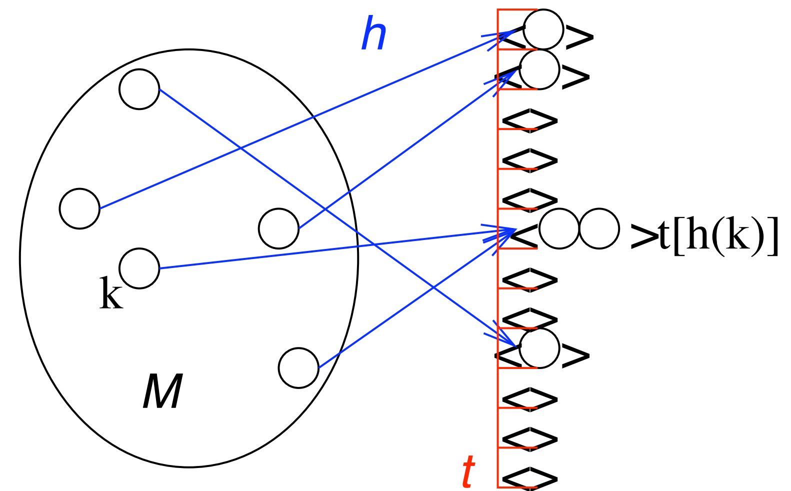
remove(k): $\mathrm{O}($Listenlänge$)$

find(k) : $\mathrm{O}($Listenlänge$)$

Aber wie lang werden die Listen?

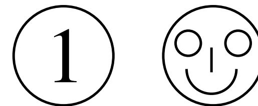
Schlechtester Fall: $\mathrm{O}(|M|)$

Besser wenn wir genug Chaos anrichten?



Etwas Wahrscheinlichkeitstheorie

für den Hausgebrauch



Hash-Beispiel

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

Ereignisse: Teilmengen von Ω — $\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$

p_x =Wahrscheinlichkeit von $x \in \Omega$

Gleichverteilung: $p_x = \frac{1}{|\Omega|}$ — $p_h = m^{-|\text{Key}|}$

$\mathbb{P}[\mathcal{E}] = \sum_{x \in E} p_x$ — $\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$

Zufallsvariable (ZV) $X_0 : \Omega \to \mathbb{R}$ — $X = |\{e \in M : h(e) = 0\}|$.

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

Erwartungswert $\mathrm{E}[X_0] = \sum_{y \in \Omega} p_y X(y)$ — $\mathrm{E}[X] = \frac{|M|}{m}$ (*)

Linearität des Erwartungswerts: $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$

Beispiel: Variante des Geburtstagsparadoxon

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

Gäste $1..n$.

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

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

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

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} I_{ij}\right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}[I_{ij}] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{P}\left[I_{ij} = 1\right] = \frac{n(n-1)}{2} \cdot \frac{1}{365} \\
&\stackrel{!}{=} 1 \Leftrightarrow n = -\frac{1}{2} + \sqrt{\frac{1}{2^2} + 730} \approx 26.52
\end{aligned}
$$

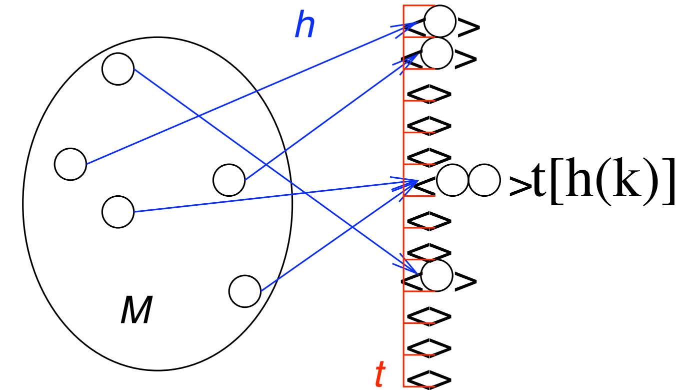
Mehr zum Geburtstagsparadoxon

Standardfomulierung:

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



Satz 1. *Die erwartete Listenlänge ist* $\mathrm{O}(1)$ *falls* $|M| = \mathrm{O}(m)$.

Beweis. Für festen Schlüssel k definiere Listenlänge
$X := |t[h(k)]| = |\{e \in M : h(e) = h(k)\}|$.
Betrachte die 0-1 ZV $X_e = 1$ für $h(e) = h(k)$, $e \in M$ und $X_e = 0$ sonst.

$$\mathrm{E}[X] = \mathrm{E}[\sum_{e \in M} X_e] = \sum_{e \in M} \mathrm{E}[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = \frac{|M|}{m}$$
$$= \mathrm{O}(1)$$

□

Das gilt unabhängig von der Eingabe M.

Zufällige Hash-Funktionen?

Naive Implementierung: ein Tabelleneintrag pro Schlüssel.

$\rightsquigarrow$ meist zu teuer

Weniger naive Lösungen: kompliziert, immer noch viel Platz.

$\rightsquigarrow$ meist unsinnig

Zufällige Schlüssel?

$\rightsquigarrow$ unrealistisch

4.3 Universelles Hashing

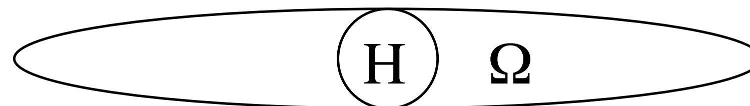
Idee: nutze nur bestimmte “einfache” Hash-Funktionen

Definition 1. $\mathscr{H} \subseteq \{0..m-1\}^{\text{Key}}$ *ist universell* *falls für alle x, y in* Key *mit* $x \neq y$ *und zufälligem* $h \in \mathscr{H}$,

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

Satz 2. *Theorem 1 gilt auch für universelle Familien von Hash-Funktionen.*

Beweis. Für $\Omega = \mathscr{H}$ 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, Key $\subseteq \{0,\ldots,m-1\}^k$

Satz 3. *Für* $\mathbf{a} = (a_1,\ldots,a_k) \in \{0,\ldots,m-1\}^k$ *definiere*

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

$H^{\cdot}$ ist eine universelle Familie von Hash-Funktionen

$$\left(x_1 * a_1 + x_2 * a_2 + x_3 * a_3 \bmod m \right) = h_a(x)$$

Beweis. Betrachte $\mathbf{x} = (x_1, \ldots, x_k)$, $\mathbf{y} = (y_1, \ldots, y_k)$ mit $x_j \neq y_j$
zähle $\mathbf{a}$-s mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
Für jede Wahl von a_is, $i \neq j$, $\exists$ genau ein a_j mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\sum_{1 \leq i \leq k} a_i x_i \equiv \sum_{1 \leq i \leq k} a_i y_i (\bmod m)$$

$$\Leftrightarrow a_j(x_j - y_j) \equiv \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) (\bmod m)$$

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

m^{k-1} Möglichkeiten a_i auszuwählen (mit $i \neq j$).
m^k ist die Gesamtzahl $\mathbf{a}$s, d. h.,

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

Bit-basierte Universelle Familien

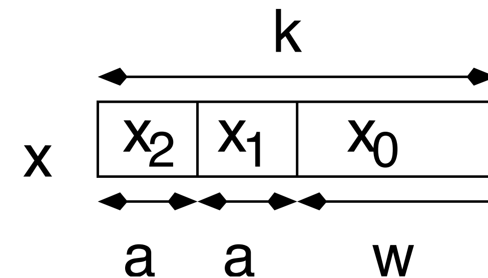
Sei $m = 2^w$, Key $= \{0,1\}^k$

Bit-Matrix Multiplikation: $H^{\oplus} = \left\{ h_{\mathbf{M}} : \mathbf{M} \in \{0,1\}^{w \times k} \right\}$

wobei $h_{\mathbf{M}}(\mathbf{x}) = \mathbf{M}x$ (Arithmetik $\bmod 2$, d. h., xor, and)

Tabellenzugriff: $H^{\oplus[]} = \left\{ h^{\oplus[]}_{(t_1,\ldots,t_b)} : t_i \in \{0..m-1\}^{\{0..w-1\}} \right\}$

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



4.4 Hashing mit Linearer Suche (Linear Probing)

Offenes Hashing: zurück zur Ursprungsidee.

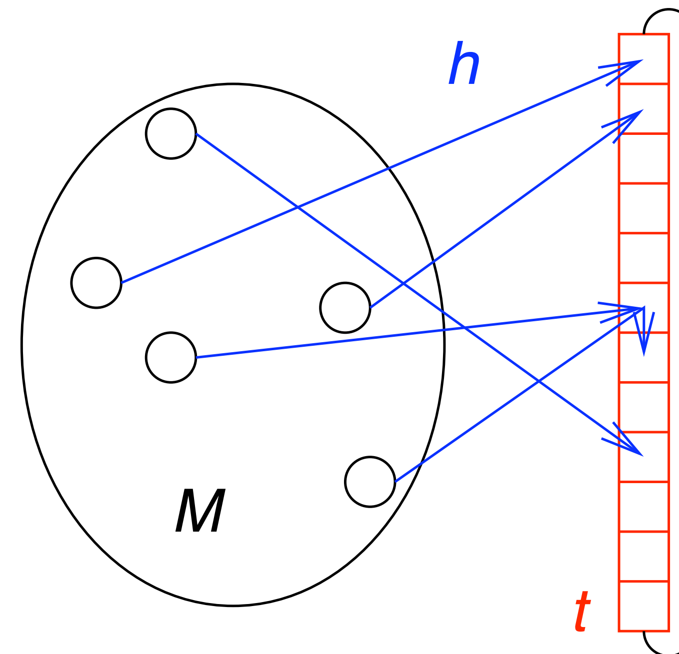
Elemente werden direkt in der Tabelle gespeichert.

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

$t = [\bot, \ldots, \bot]$: **Array** $[0..m+m'-1]$ **of** Element

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

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

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \bot$ **do** ;

assert $i < m + m' - 1$

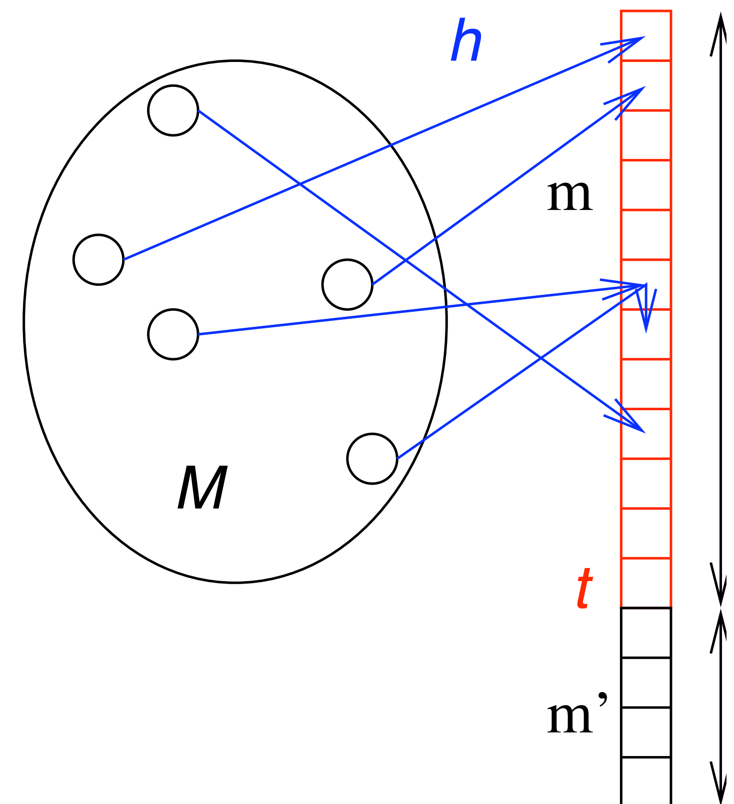
$t[i] := e$

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

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \bot$ **do**

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

return $\bot$



Remove

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

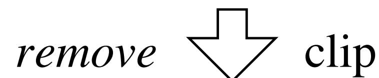
invariant $\forall i : t[i] \neq \bot \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[i] \neq \bot$

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

```
    for i := h(k) to ∞ while k ≠ t[i] do            // search k
        if t[i] = ⊥ then return                     // nothing to do
    // we plan for a hole at i.
    for j := i + 1 to ∞ while t[j] ≠ ⊥ do
        // Establish invariant for t[j].
        if h(t[j]) ≤ i then
            t[i] := t[j]                            // Overwrite removed element
            i := j                                  // move planned hole
    t[i] := ⊥                                       // 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
tt 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	⊥



an	bo	cp	dq	er	fs	gt	hu	iv	jw	kx	ly	mz
⊥	⊥	chop	~~clip~~	axe	cube	dice	hash	lop	slash	hack	fell	⊥
⊥	⊥	chop	lop	axe	cube	dice	hash	~~lop~~	slash	hack	fell	⊥
⊥	⊥	chop	lop	axe	cube	dice	hash	slash	~~slash~~	hack	fell	⊥
⊥	⊥	chop	lop	axe	cube	dice	hash	slash	⊥	hack	fell	⊥

4.5 Verketten $\leftrightarrow$ Lineare Suche

Volllaufen: Verketten weniger empfindlich.

Unbeschränktes offenes Hashing hat nur amortisiert konst. Einfügezeit

Cache: Lineare Suche besser. Vor allem für doall

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

Referentielle Integrität: Nur bei Verketten !

Leistungsgarantien: Universelles Hashing funktioniert so nur mit Verketten

4.6 Perfektes Hashing

hier nicht

Mehr Hashing

- Hohe Wahrscheinlichkeit und Garantien für den schlechtesten Fall
  $\rightsquigarrow$ höhere Anforderungen and die Hash-Funktionen
- Hashing als Mittel zur Lastverteilung z. B., storage servers,
  (peer to peer Netze,. . .)
- Verschiedene Plattengrößen und Geschwindigkeiten
- Platten hinzufügen / ersetzen
- $\mathrm{O}(1)$ find / perfektes Hashing

5 Sortieren & Co

Formaler

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

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

- s' ist Permutation von s
- $e'_1 \leq \cdots \leq e'_n$ für eine lineare Ordnung '$\leq$'

Anwendungsbeispiele

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

Beispiele aus Kurs/Buch

- Aufbau von Suchbäumen
- Kruskals MST-Algorithmus
- Verarbeitung von Intervallgraphen (z. B. Hotelbuchungen)
- 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.

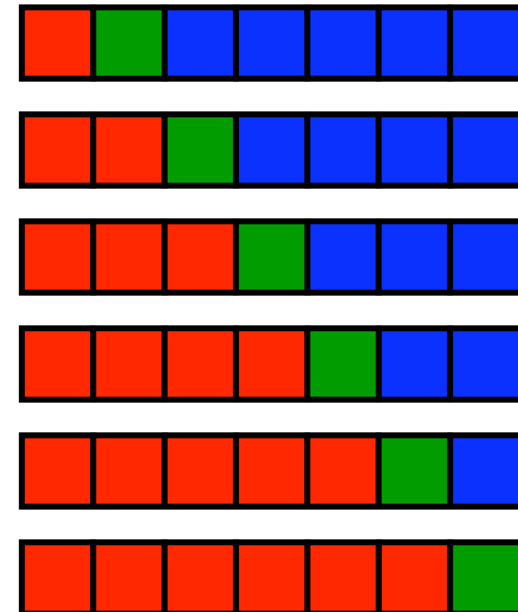
5.1 Überblick

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

5.2 Einfache Sortieralgorithmen

Sortieren durch Einfügen (insertion sort)

Procedure insertionSort(a : **Array** $[1..n]$ **of** Element)
  for $i := 2$ **to** n **do**
    invariant $a[1] \leq \cdots \leq a[i-1]$
    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 $a[1] \leq \cdots \leq a[i-1]$

    // move $a[i]$ to the right place

    $e := a[i]$

    if $e < a[1]$ **then**   // new minimum

      for $j := i$ **downto** 2 **do** $a[j] := a[j-1]$

      $a[1] := e$

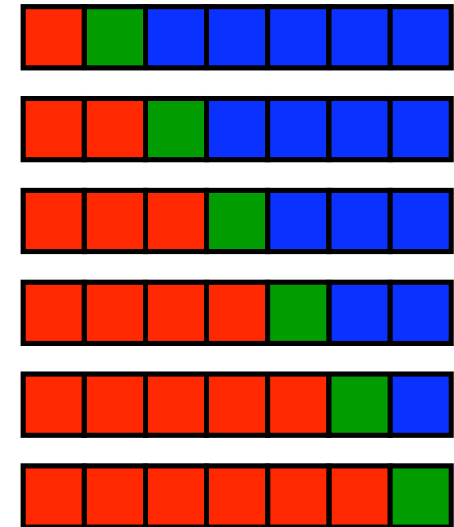
    else   // use $a[1]$ as a sentinel

      for $j := i$ **downto** $-\infty$ **while** $a[j-1] > e$ **do** $a[j] := a[j-1]$

      $a[j] := e$

Analyse

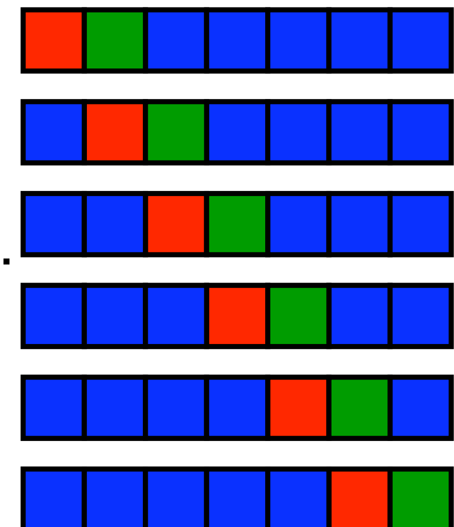
Schlechtester Fall



Die i-te Iteration braucht Zeit $\mathrm{O}(i)$.

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

Bester Fall



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

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

5.3 Sortieren durch Mischen

Idee: Teile und Herrsche

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

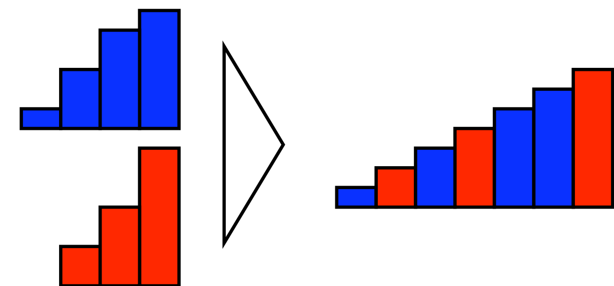
Mischen (merge)

Gegeben:

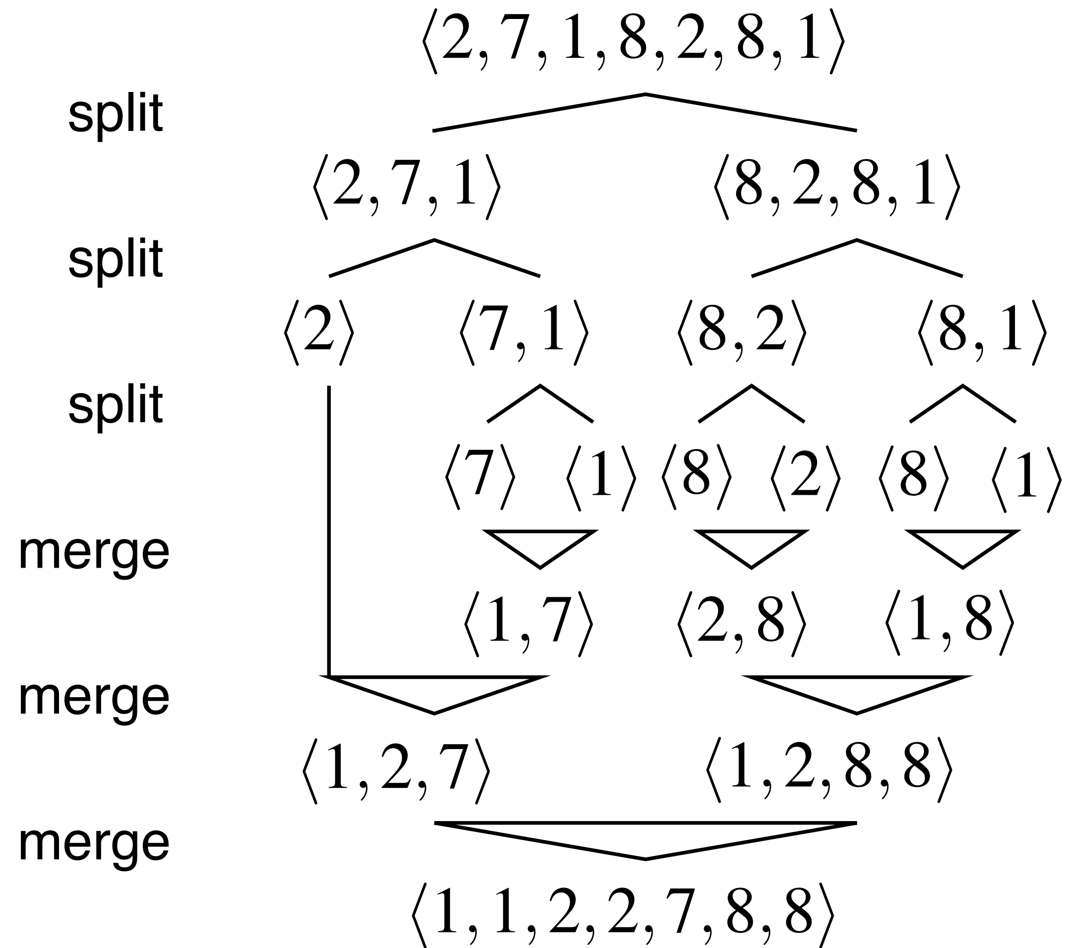
zwei sortierte Folge a und b

Berechne:

sortierte Folge der Elemente aus a und b



Beispiel

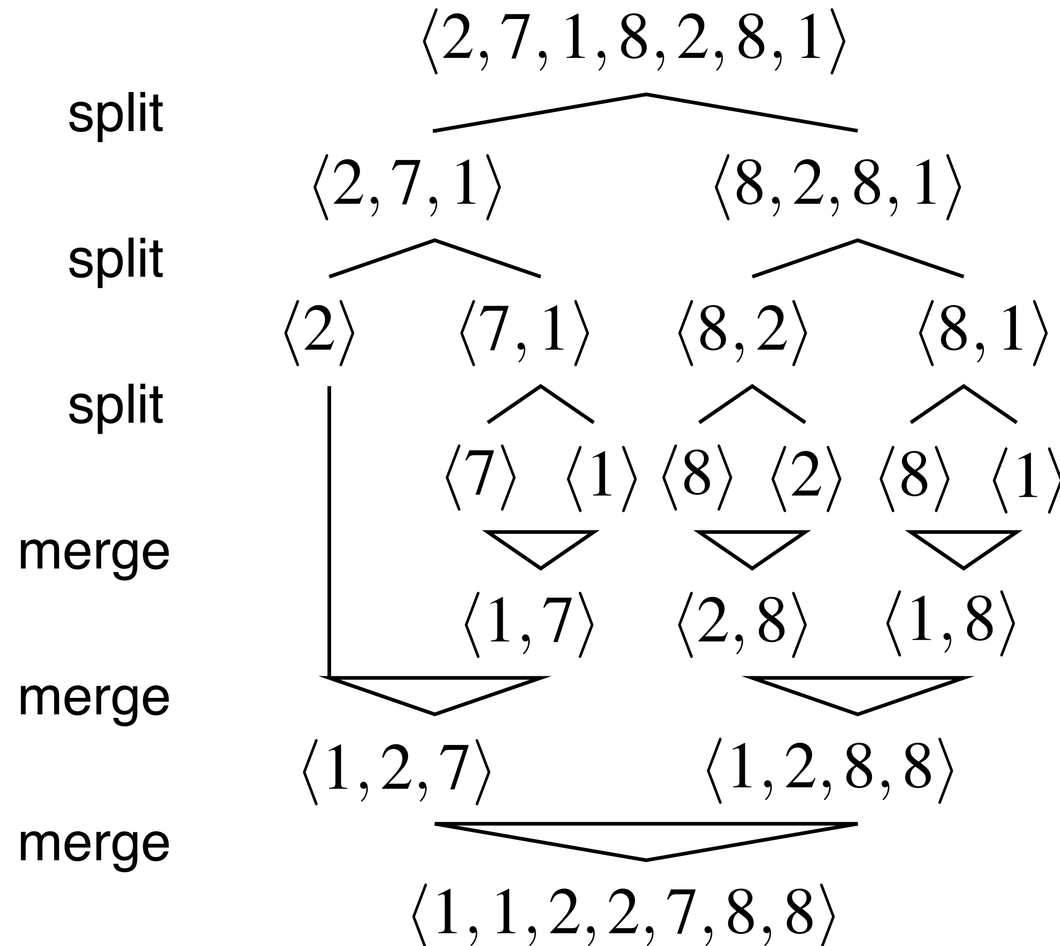


Mischen

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

a	b	c	operation
$\langle 1,2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle\rangle$	move a
$\langle 2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle 1\rangle$	move b
$\langle 2,7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1\rangle$	move a
$\langle 7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1,2\rangle$	move b
$\langle 7\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2\rangle$	move a
$\langle\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2,7\rangle$	concat b
$\langle\rangle$	$\langle\rangle$	$\langle 1,1,2,2,7,8,8\rangle$	

Analyse



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

Analyse

$$T(n) = \mathrm{O}(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$$

Problem: Runderei

Ausweg: genauer rechnen (siehe Buch)

Dirty trick:

Eingabe auf Zweierpotenz aufblasen

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

$\rightsquigarrow$

normales Master-Theorem anwendbar

Zeit $\mathrm{O}(n \log n)$

5.4 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.

$\rightsquigarrow$

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

$$n \log n - \mathrm{O}(n)$$

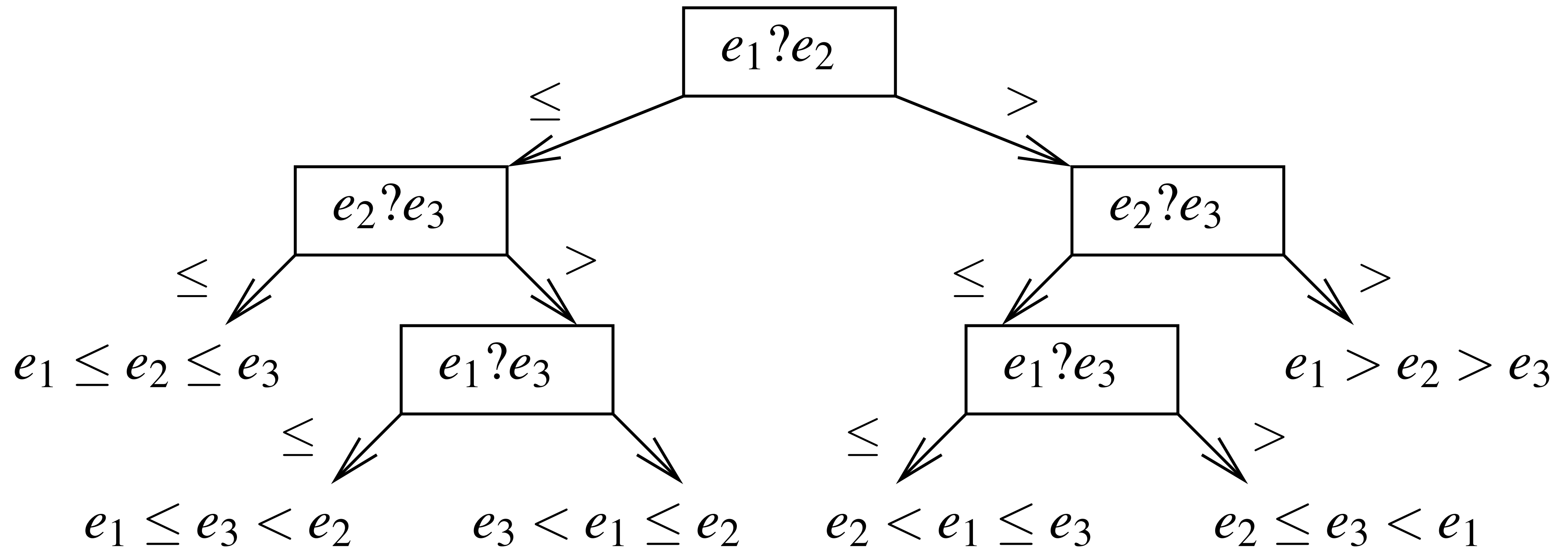
Vergleiche im schlechtesten Fall.

Beweis:

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

Es gibt genau $n!$ solche Permutationen.

Baumbasierte Sortierer-Darstellung



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

Ausführungszeit entspricht Tiefe T

Beweis

Baum der Tiefe T hat höchstens 2^T Blätter.

$\Rightarrow 2^T \geq n!$

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

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

Beweis für linken Teil:

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

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

Randomisierung, Mittlere Ausführungszeit

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

Beweis: nicht hier.

5.5 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

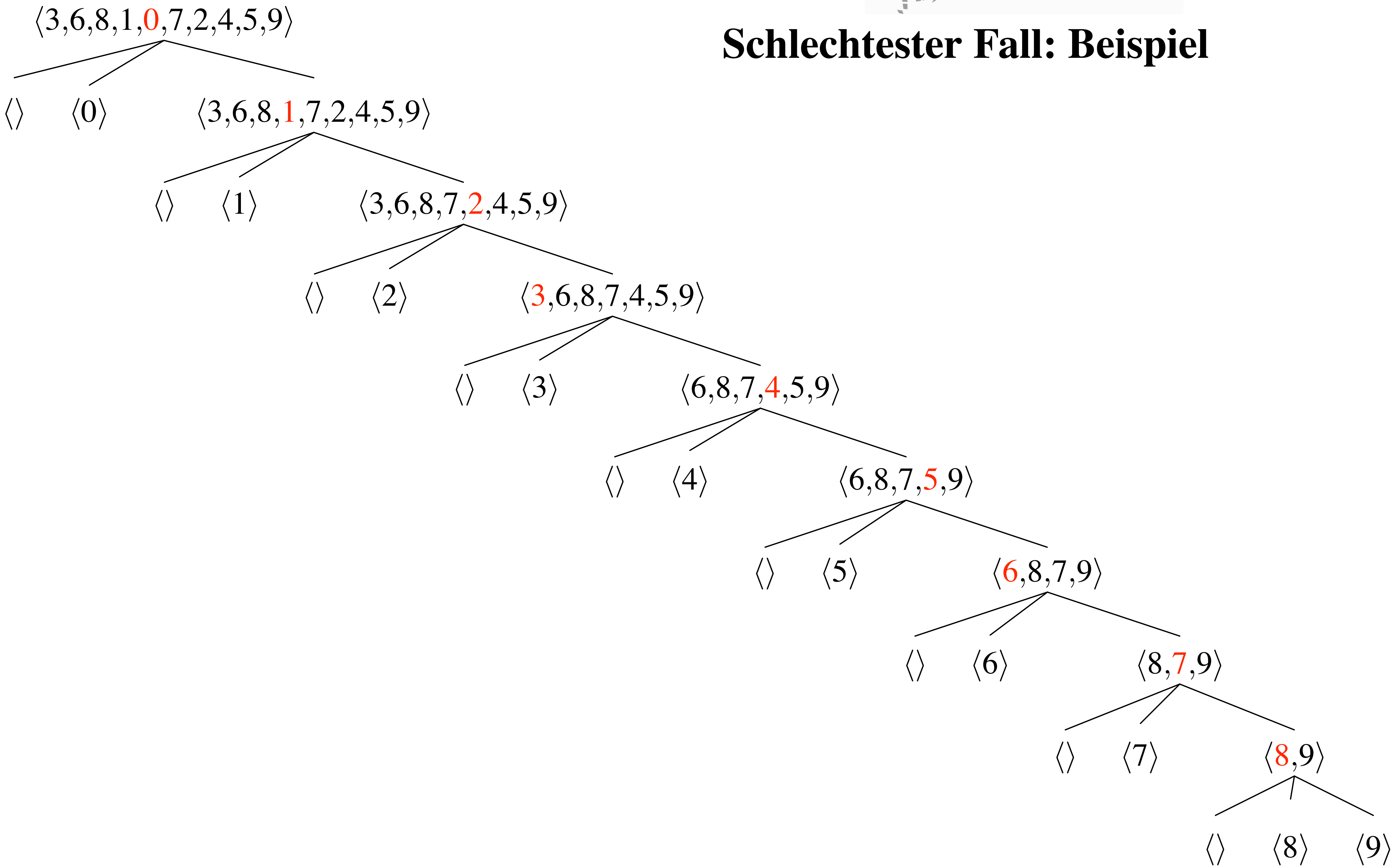
Annahme: Pivot ist immer Minimum (oder Max.) der Eingabe

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

$\Rightarrow$

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

Schlechtester Fall: Beispiel



Quicksort – Analyse im besten Fall

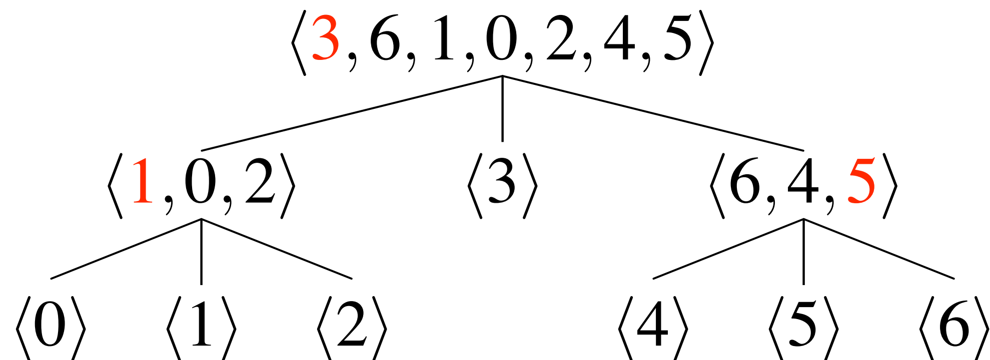
Annahme: Pivot ist immer Median der Eingabe

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

$\Rightarrow$ (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

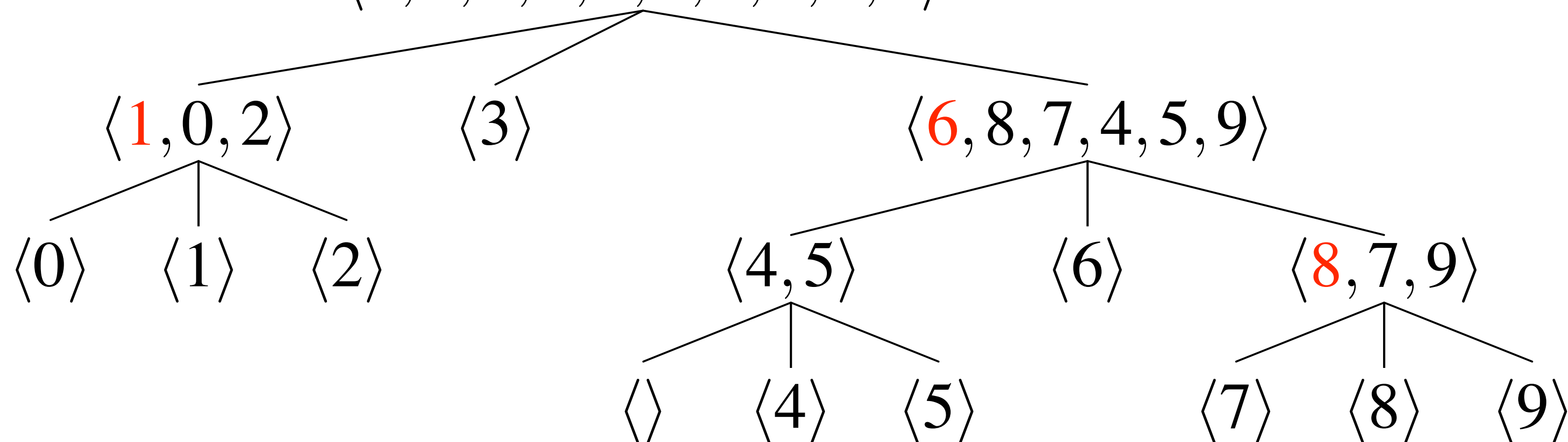
pick $p \in s$ uniformly at random

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

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

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

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



5.5.1 Satz: Quicksort hat erwartete Laufzeit $\mathrm{O}(n\log n)$

Annahme: alle Elemente verschieden — Warum 'OBdA'?

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

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

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
  if $|s| \leq 1$ **then return** s
  pick $p \in s$ uniformly at random
  $a := \langle e \in s : e < p \rangle$ // $|s|$
  $b := \langle e \in s : e = p \rangle$ // 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 $\bar{C}(n) = \mathrm{O}(n \log n)$ zu zeigen.

Beweisansatz 2: Genauere, elegantere Analyse

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

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

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

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

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

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

Sortierte Eingabefolge:

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

$X_{ij} := 1$

$\Leftrightarrow$

e'_i wird mit e'_j verglichen

$\Leftrightarrow$

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

$\Rightarrow$

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

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

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

$$= \sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{2}{k}$$

$$\leq \sum_{i=1}^{n}\sum_{k=2}^{n}\frac{2}{k}$$

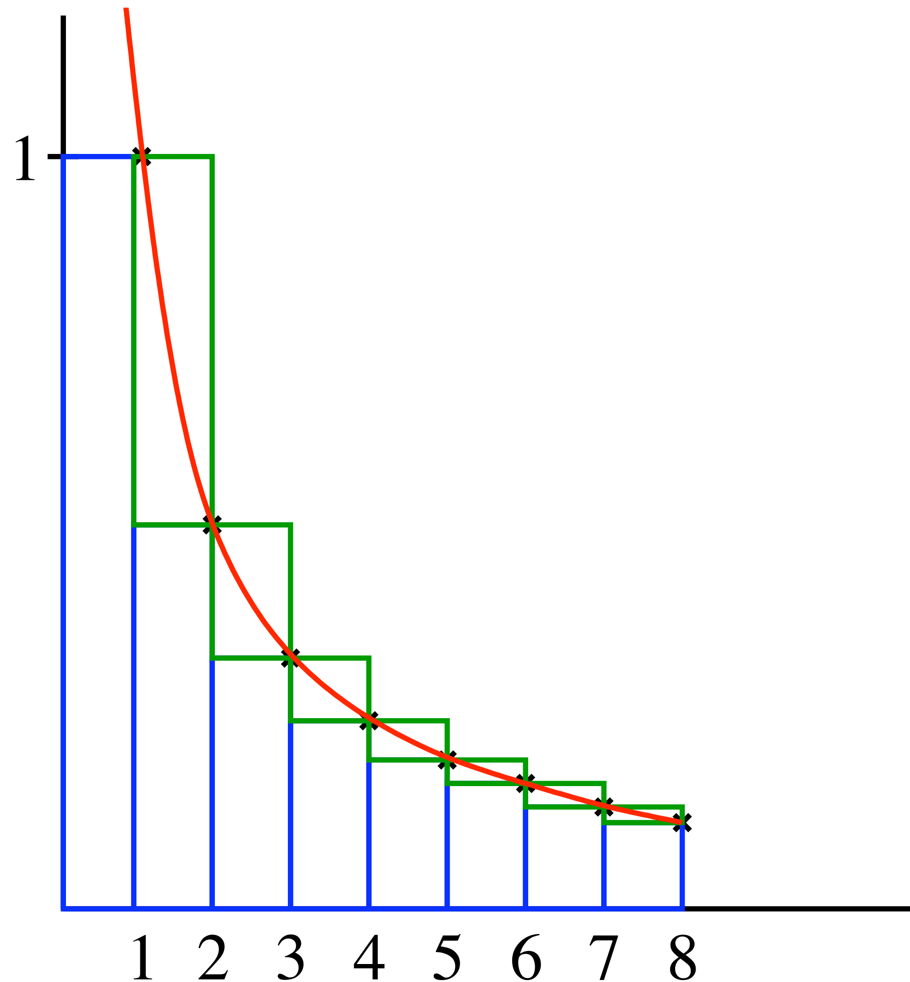
$$= 2n\sum_{k=2}^{n}\frac{1}{k} \qquad \text{(harmonische Summe)}$$

$$= 2n(H_n - 1) \leq 2n(1 + \ln n - 1) = 2n\ln n \ .$$

□

Exkurs: Harmonische Summe

$$\ln n = \int_1^n \frac{1}{x}\,dx \leq \sum_{1 \leq i \leq n} \frac{1}{i} \leq 1 + \int_1^n \frac{1}{x}\,dx = 1 + \ln n$$



5.5.2 Quicksort: Effiziente Implementierung

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

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

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

$p := a[\text{pickPivotPos}(a, \ell, r)]$

$i := \ell$; $j := r$

repeat // a: | ℓ | $i \rightarrow$ | $\leftarrow j$ | r |

while $a[i] < p \wedge i \leq j$ **do** $i{+}{+}$

while $a[j] > p \wedge i \leq j$ **do** $j{-}{-}$

if $i \leq j$ **then** swap($a[i], a[j]$); $i{+}{+}$; $j{-}{-}$

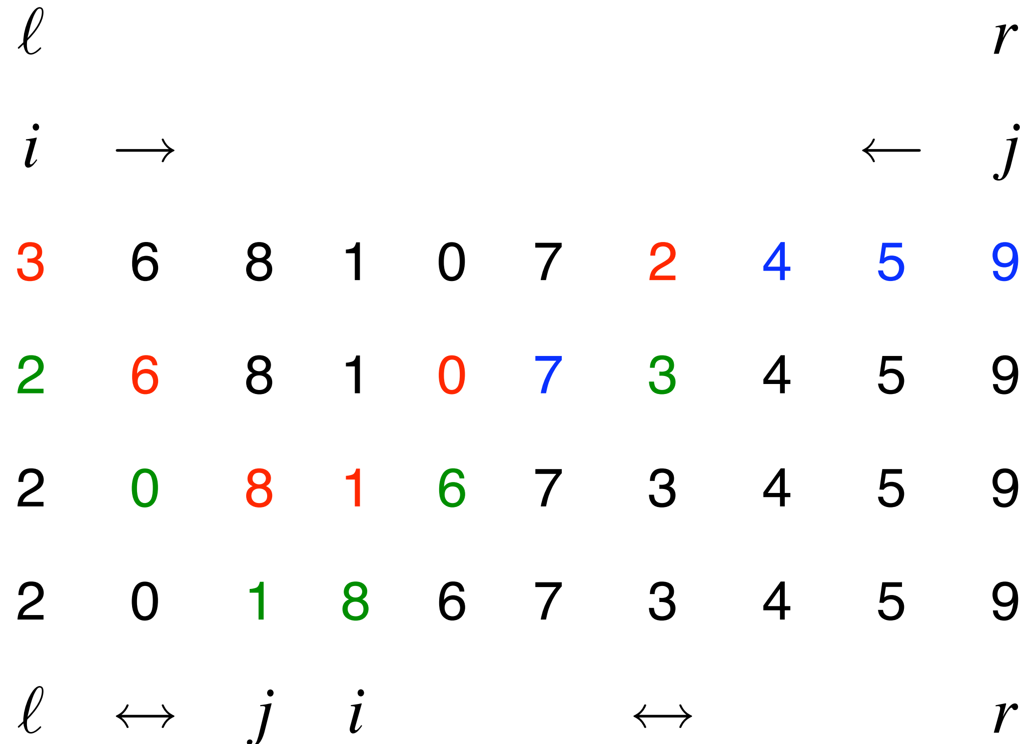
until $i > j$

// a: | ℓ | $\leftrightarrow$ | j | i | $\leftrightarrow$ | r |

qSort(a, ℓ, j)

qSort(a, i, r)

Beispiel: Partitionierung, $p = 3$



Beispiel: Rekursion

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

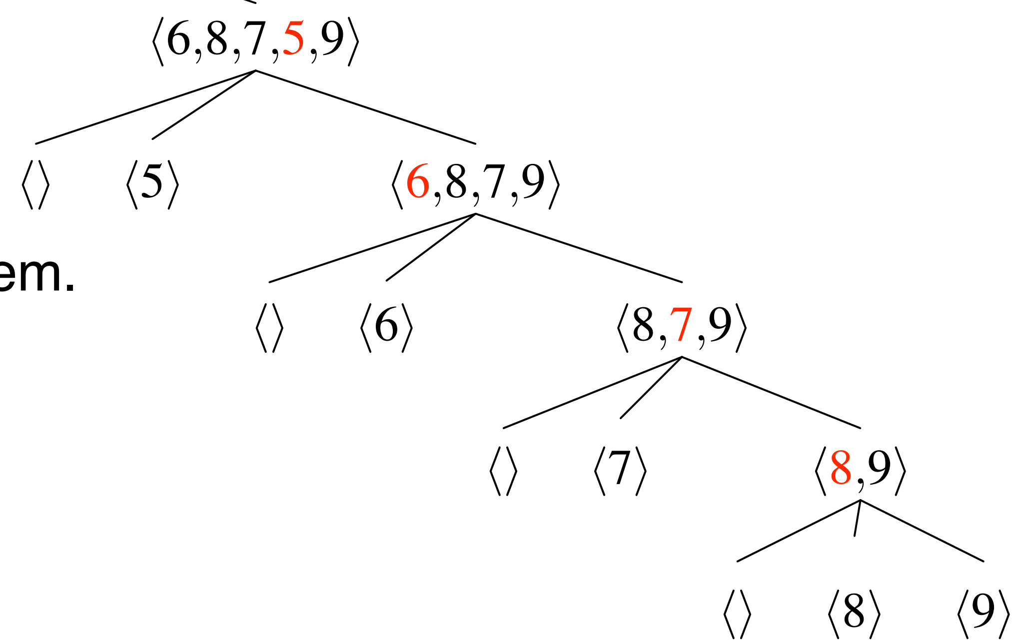
Größerer Basisfall

```
Procedure qSort(a : Array of Element; ℓ, r : ℕ)
    if r − ℓ + 1 ≤ n_0 then insertionSort(a[ℓ..r])
    p:= a[pickPivotPos(a, ℓ, r)]
    i:= ℓ;  j:= r
    repeat                                  // a: | ℓ      i→  ←j      r |
        while a[i] < p ∧ i ≤ j do i++
        while a[j] > p ∧ i ≤ j do j−−
        if i ≤ j then swap(a[i], a[j]);  i++ ;  j−−
    until i > j
    // a: | ℓ   ↔   j  i   ↔   r |
    qSort(a, ℓ, j)
    qSort(a, i, r)
```

Inplace? Wirklich?

Im schlechtesten Fall:

$O(n)$ für Rekursionsstapel.



Im Mittel:

$O(\log n)$ zusätzlicher Platz – kein Problem.

Als Garantie für schlechtesten Fall:

- halbrekursive Implementierung
- Rekursion auf kleinere Hälfte

Halbrekursive Implementierung

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  while $r - \ell + 1 > n_0$ **do**
    partition $a[\ell..r]$ using pivot $a[\text{pickPivotPos}(a, \ell, r)]$
    // a: | ℓ $\leftrightarrow$ j i $\leftrightarrow$ r |
    if $i < (\ell + r)/2$ **then** qSort(a, ℓ, j); $\ell := i$
    else qSort(a, i, r); $r := j$
  insertionSort($a[\ell..r]$)

Halbrekursive Implementierung

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

  while $r - \ell + 1 > n_0$ **do**

    partition $a[\ell..r]$ using pivot $a[\text{pickPivotPos}(a, \ell, r)]$

    // a: | ℓ | $\leftrightarrow$ | j | i | $\leftrightarrow$ | r |

    if $i < (\ell + r)/2$ **then** qSort(a, ℓ, j); $\ell := i$

    else qSort(a, i, r); $r := j$

  insertionSort($a[\ell..r]$)

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

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

Vergleich Quicksort $\leftrightarrow$ Mergesort

Pro Mergesort

- $O(n \log n)$ Zeit (deterministisch)
  qsort: $\exists$ det. Varianten (proportion extend sort [Chen 2001])
- $n \log n + O(n)$ 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?

5.6 Auswahl (Selection)

Definition: Rang der Elemente einer Folge s mit $|s| = n$:

Abbildung $r : 1..n \to 1..n$ mit

$\forall i, j : s[i] < s[j] \Rightarrow r(x) < r(y)$.

Grob: Das $a[i]$ ist das $r(i)$-te Element von a.

Frage: warum ist r nicht notwendig eindeutig?

```
//return an element of s with rank k
Function select(s : Sequence of Element; k : ℕ) : Element
    assert |s| ≥ k
```

Auswahl – Anwendungen

Statistik

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

5.6.1 Unterprogramm

z. B. Eingabe eingrenzen auf vielversprechendste Elemente

Quickselect

$\approx$ 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)//

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

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

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

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

Beispiel

s	k	p	a	b	c
$\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8\rangle$	6	2	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3,4,5,9,6,5,3,5,8\rangle$
$\langle 3,4,5,9,\mathbf{6},5,3,5,8\rangle$	4	6	$\langle 3,4,5,5,3,5\rangle$	$\langle 6\rangle$	$\langle 9,8\rangle$
$\langle 3,4,\mathbf{5},5,3,5\rangle$	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 … k]
  b := $\langle e \in s : e = p \rangle$
  if $|a| + |b| \geq k$ **then return** p // [a | $b = \langle p, \ldots, p \rangle$ … k]
  c := $\langle e \in s : e > p \rangle$
  return select$(c, k - |a| - |b|)$ // [a | b | c … k]

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

- partielles Sortieren (z. B. einfache Variante von quickselect) wer weiss wie es geht?

- Weitere Verallgemeinerungen:

  mehrere Ränge, teilweise sortierte Eingaben,. . .

5.7 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.

K Schlüssel – Eimer-Sortieren (bucket sort)

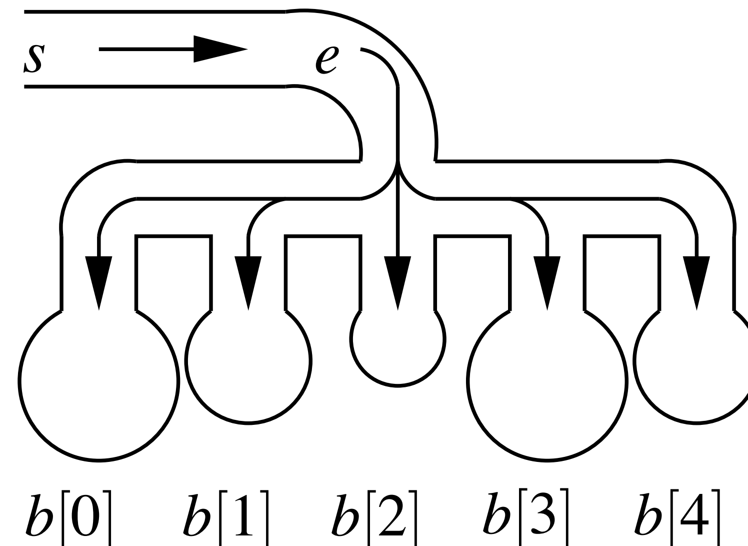
Procedure KSort(s : Sequence **of** Element)

  b=$\langle\langle\rangle,\ldots,\langle\rangle\rangle$: **Array** $[0..K-1]$ **of** Sequence **of** Element

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

  s := concatenation of $b[0],\ldots,b[K-1]$

Zeit: $\mathrm{O}(n+K)$



Beispiel: $K = 3$

Procedure KSort(s : Sequence **of** Element)

  b=$\langle\langle\rangle,\ldots,\langle\rangle\rangle$: **Array** $[0..K-1]$ **of** Sequence **of** Element

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

  s := concatenation of $b[0],\ldots,b[K-1]$

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

verteilen$\rightsquigarrow$

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

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)

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

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

  $C{:=}\,0$

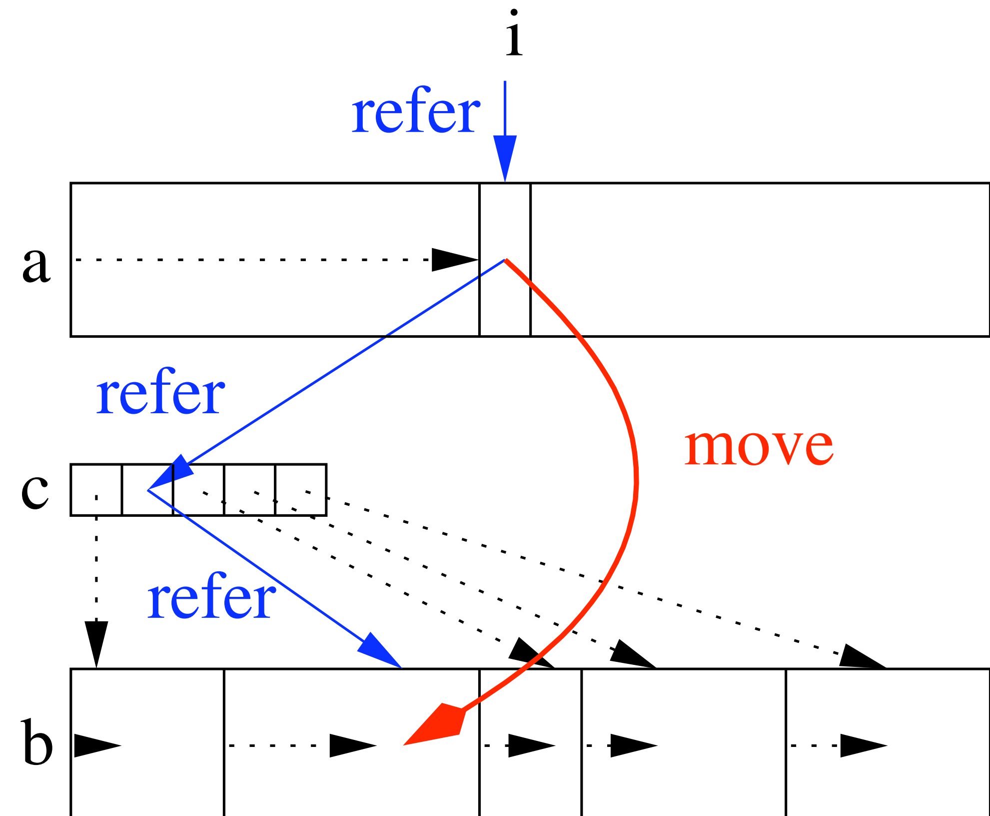
  for $k := 0$ **to** $K-1$ **do**

    $(C, c[k]) := (C + c[k], C)$

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

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

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



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 $(x \textbf{ div } K^i) \textbf{ mod } K$// x | $d-1$ | ··· | i | ··· | 1 | 0 |

digits

key(x)

KSort(s)

invariant

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

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

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

Sortieren: vergleichsbasiert ↔ ganzzahlig

pro ganzzahlig:

- asymptotisch schneller

pro vergleichsbasiert

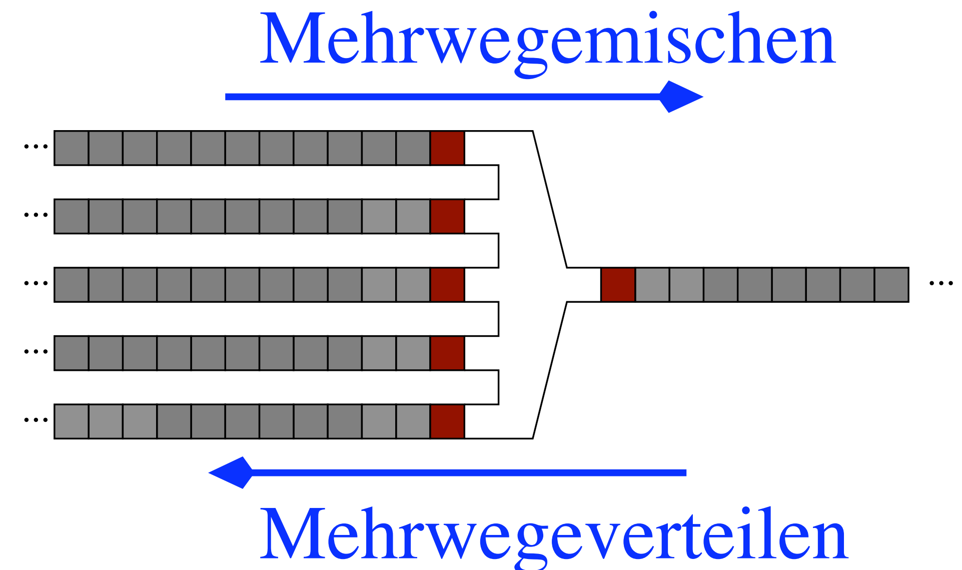
- weniger Annahmen
  (z. B. wichtig für Algorithmenbibliotheken)
- robust gegen beliebige Eingabeverteilungen
- Cache-Effizienz weniger schwierig

Mehr zum Sortieren

- Verfügbar in Algorithmenbibliotheken
- (binary) mergesort ⇝ Mehrwegemischen
- quicksort ⇝ Sortieren durch Mehrwegeverteilen

⇝ Parallel

⇝ Extern: oft noch wichtiger als intern



Verallgemeinerungen:

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

6 Prioritätslisten

Prioritätslisten (priority queues)

Verwalte Menge M von Elementen mit Schlüsseln

Insert(e): $M := M \cup e$

DeleteMin: return and remove $\min M$

Prioritätslisten – Anwendungen

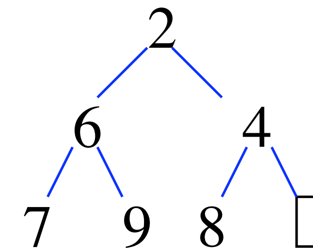
(ohne zusätzliche Operationen)

- 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)

6.1 Binäre Heaps

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

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



Beobachtung: Minimum $=$ Wurzel

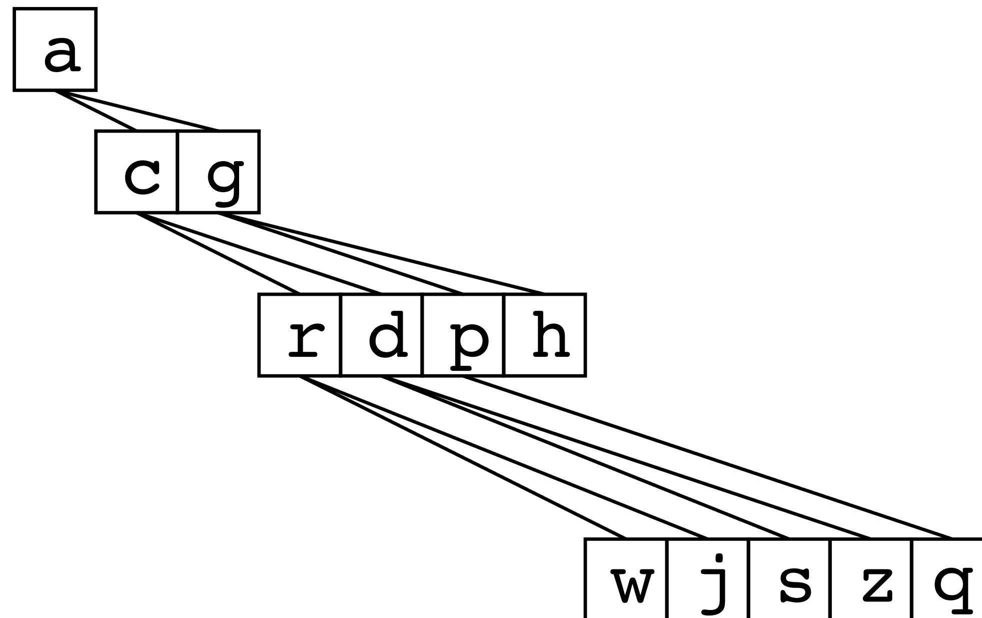
Idee: Änderungen nur entlang eines Pfades Wurzel–Blatt

$\rightsquigarrow$

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

Implizite Baum-Repräsentation

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

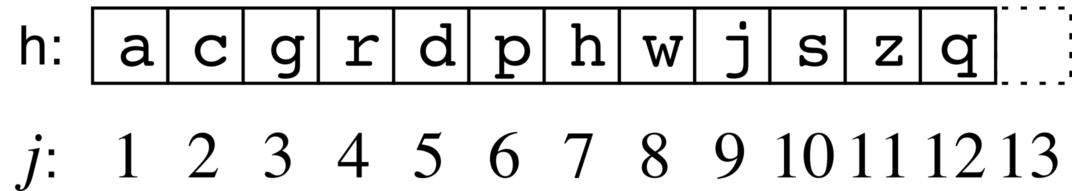


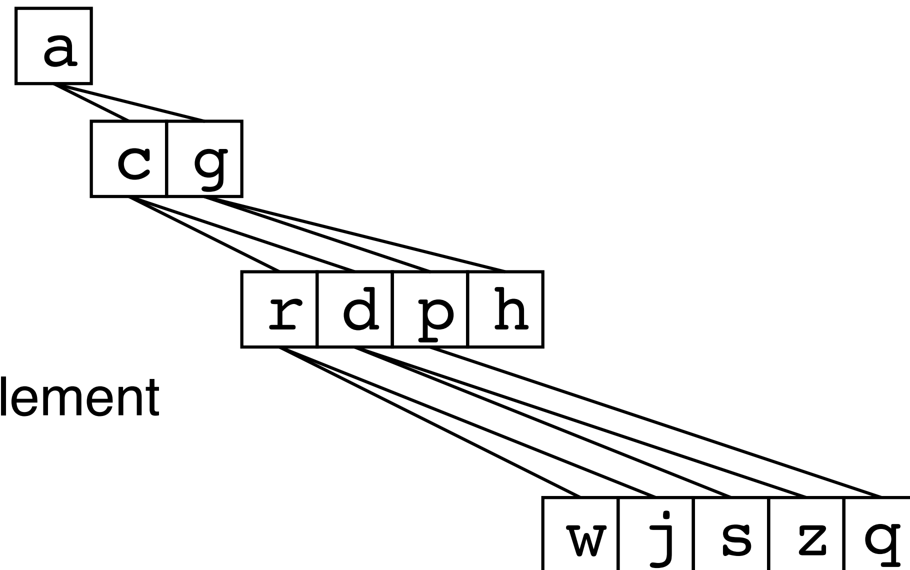
Nicht nur nützlich für heaps:

z. B. Turnierbäume, statische Suchbäume

Pseudocode

(beschränkte PQ)





Class BinaryHeapPQ$(w : \mathbb{N})$ **of** Element

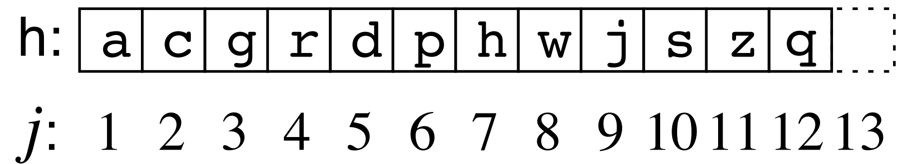
h : **Array** $[1..w]$ **of** Element

$n{=}0 \ : \mathbb{N}$

invariant $\forall j \in 2..n : h[\lfloor j/2 \rfloor] \leq h[j]$

Function min **assert** $n > 0$; **return** $h[1]$

Einfügen

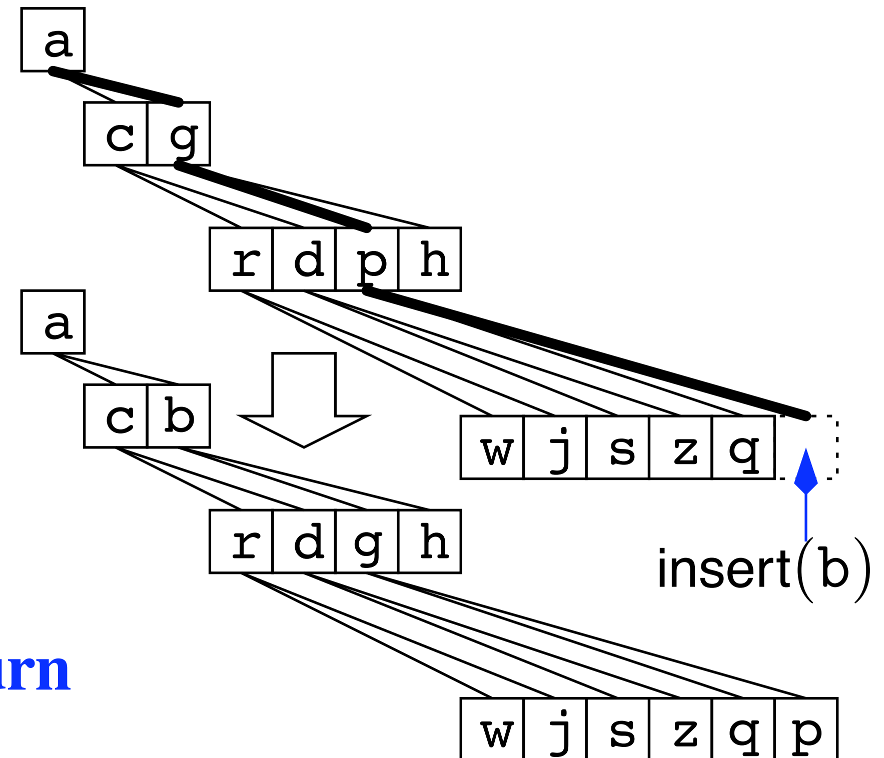


Procedure insert$(e : Element)$

  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$(\lfloor i/2 \rfloor)$

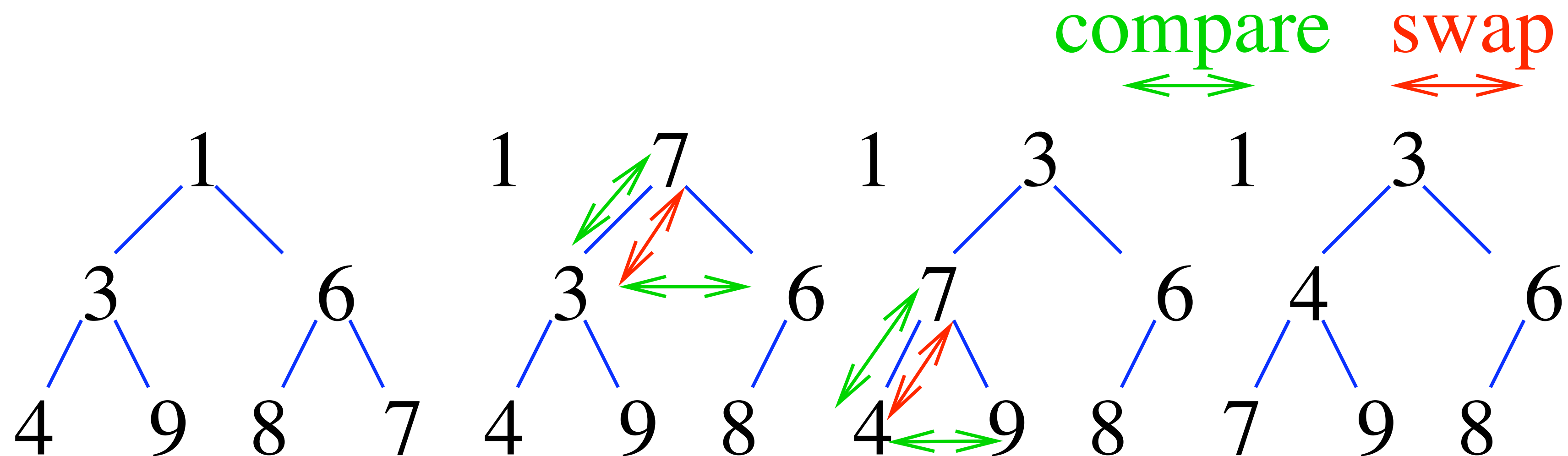
Function deleteMin : Element

result= $h[1]$: Element

$h[1] := h[n]$; n−−

siftDown(1)

return result



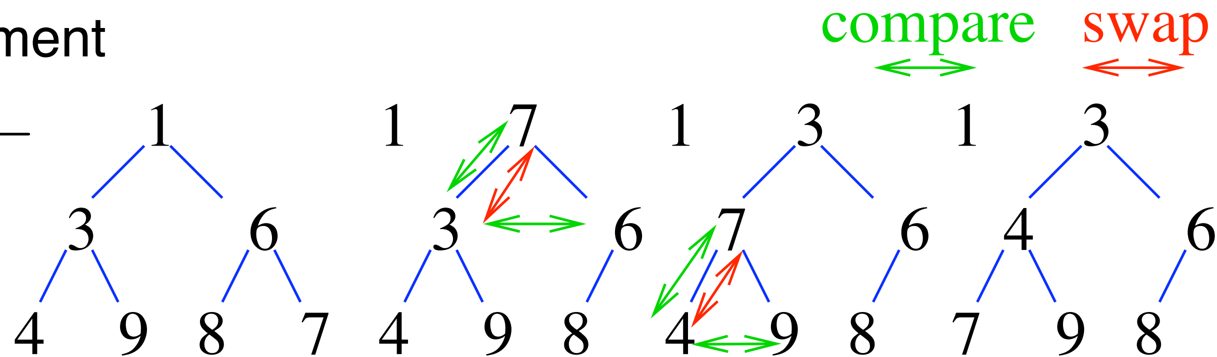
Function deleteMin : Element

result$=h[1]$: Element

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

siftDown(1)

return result



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

assert heap property @ subtrees rooted 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$(m)\vee h[$sibling$(m)]\geq h[m]$

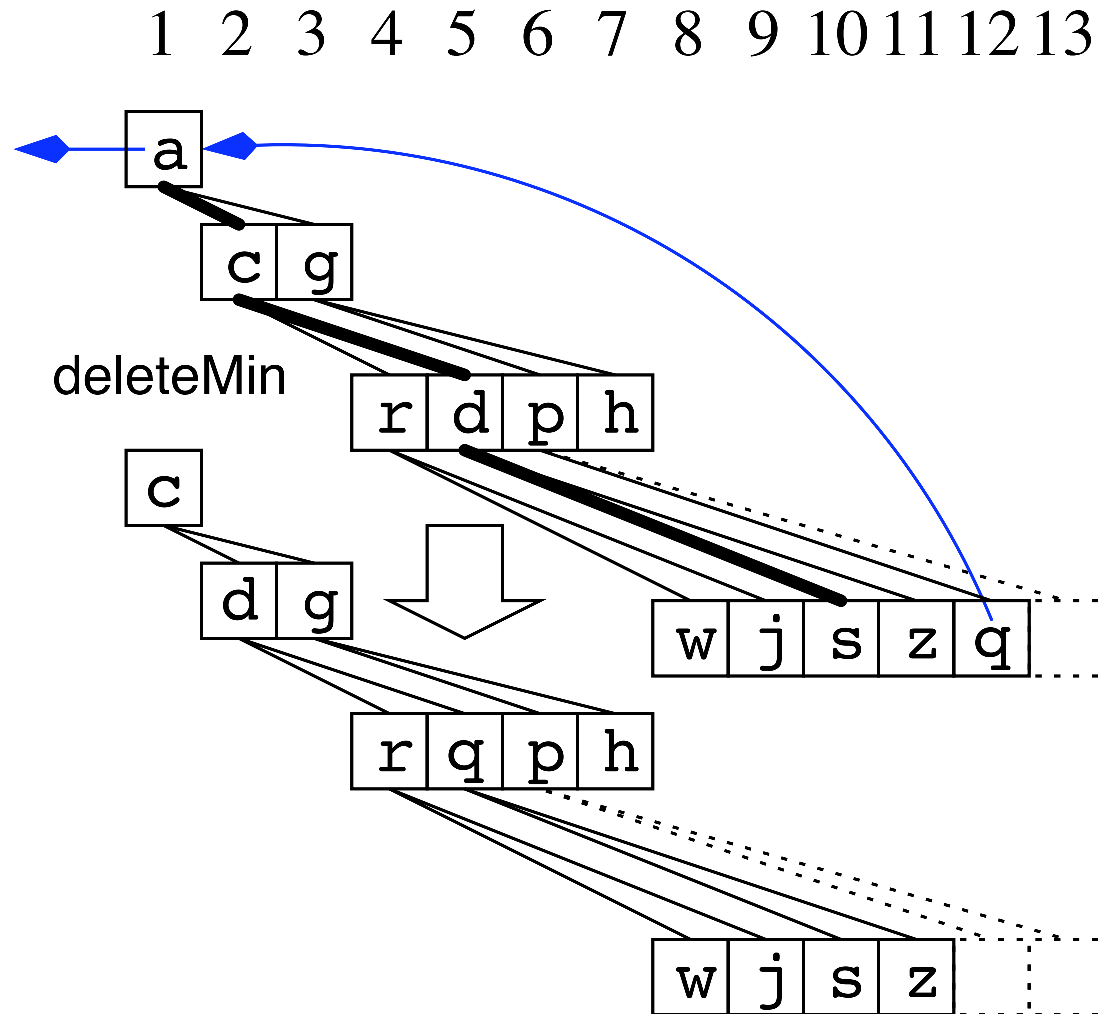
if $h[i]>h[m]$ **then** // heap property violated

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

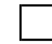
siftDown(m)

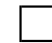
assert the heap property holds for the subtree rooted at i

deleteMin: Beispiel



Binäre 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 buildHeap buildHeapRecursive(1)

Procedure buildHeapRecursive$(i:\mathbb{N})$

 if $4i \leq n$ **then**

  buildHeapRecursive$(2i)$

  assert the heap property holds for the tree rooted at left child

  buildHeapRecursive$(2i+1)$

  assert the heap property holds for the tree rooted at right child

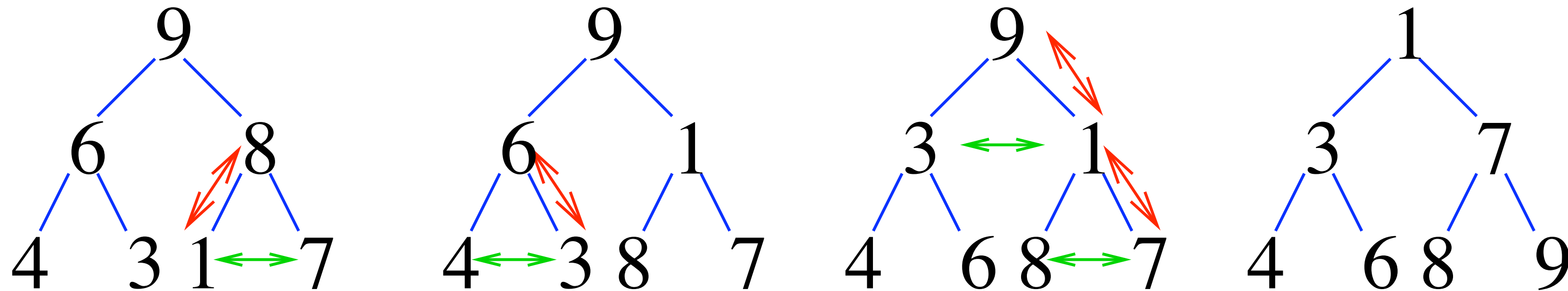
 siftDown(i)

 assert the heap property holds for the tree rooted at i

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 $\mathrm{O}(n)$

Beweis: Sei $k = \lfloor \log n \rfloor$.

In Tiefe $\ell \in 0..\lfloor \log n \rfloor$:

- 2^ℓ Aufrufe von siftDown
- Kosten je $\mathrm{O}(k-\ell)$. Insgesamt:

$$\mathrm{O}\left(\sum_{0\leq\ell<k} 2^\ell(k-\ell)\right) = \mathrm{O}\left(2^k \sum_{0\leq\ell<k} \frac{k-\ell}{2^{k-\ell}}\right) = \mathrm{O}\left(2^k \underbrace{\sum_{j\geq1} \frac{j}{2^j}}_{\mathrm{O}(1)!}\right)$$

$$= \mathrm{O}\left(2^k\right) = \mathrm{O}(n)$$

Ein nützlicher Rechentrick

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

```
1/2   +   1/4   +   1/8   +   1/16   +   ... = 1
          1/4   +   1/8   +   1/16   +   ... = 1/2
                    1/8   +   1/16   +   ... = 1/4
                              1/16   +   ... = 1/8
                               ...           = ...
-------------------------------------------------
1*1/2 + 2*1/4 + 3*1/8 + 4*1/16   +   ... =  2
```

Heapsort

Procedure heapSortDecreasing$(a[1..n])$

a.buildHeap

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

swap$(a[1], a[i])$

$n--$

siftDown(1)

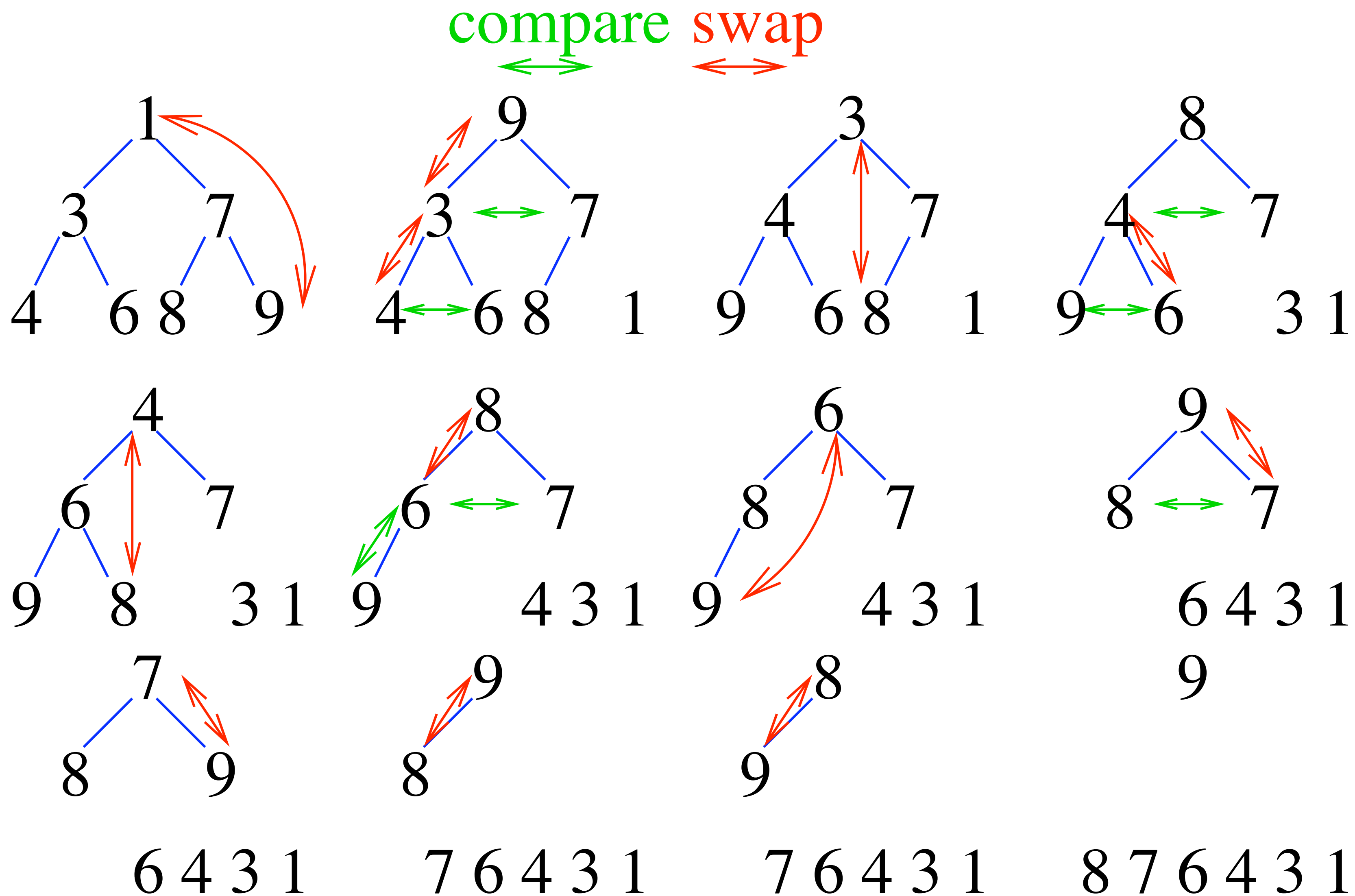
Laufzeit: $\mathrm{O}(n \log n)$

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	$O(n \log n)$	$O(\frac{n}{B} \log n)$	$O(\frac{n}{B} \log n)$
($B =$Blockgröße)			

Kompromiss: z. B.

introspektives Quicksort der C++ Standardbibliothek:

Quicksort starten. Zu wenig Fortschritt? Umschalten auf Heapsort.

6.2 Adressierbare Prioritätslisten

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

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

Procedure decreaseKey$(h:$ Handle$, k:$ Key$)$ **assert** key$(h)\geq k$; key$(h):=k$

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

Adressierbare Prioritätslisten: Anwendungen

Greedy-Algorithmus:

while solution not complete **do**

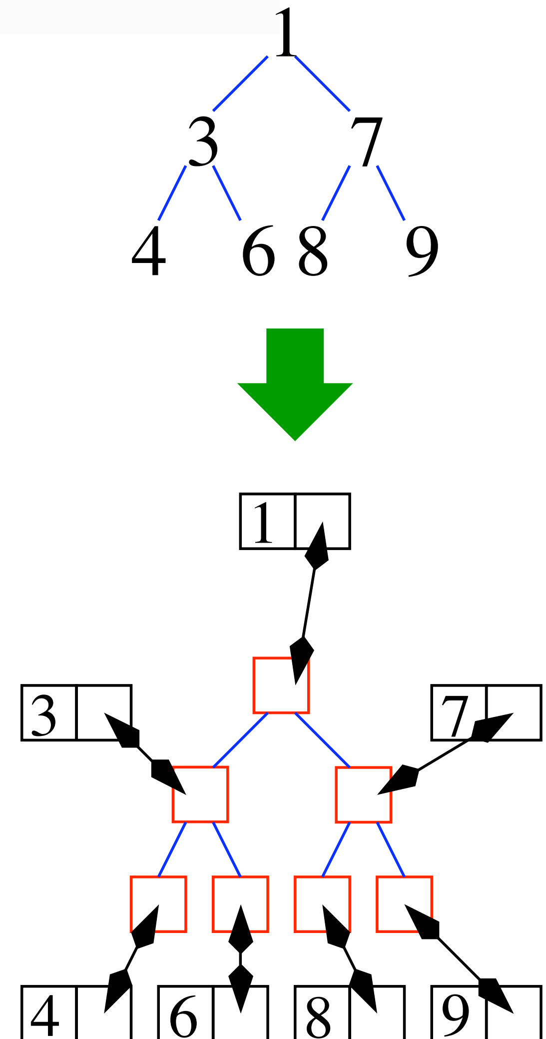
  add the best available "piece" to the solution

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

Beispiele:

- Dijkstras Algorithmus für kürzeste Wege
- Jarník-Prim Algorithmus für minimale Spannbäume
- Scheduling: Jobs $\longrightarrow$ am wenigsten belastete Maschine
- Hierarchiekonstruktion für Routenplanung

Adressierbare Binäre Heaps



Problem: Elemente bewegen sich.

Dadurch werden Elementverweise ungültig.

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

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

$\leadsto$ Vermittler bei jeder Vertauschung aktualisieren.

$\leadsto$ Rückverweis Element $\longrightarrow$ Vermittler

Laufzeit:

$\mathrm{O}(\log n)$ für alle Operationen ausser merge und buildHeap, die $\mathrm{O}(n)$ 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	$O(\log n)$	$O(\log n)$
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

- Untere Schranke $\Omega(\log n)$ für deleteMin, vergleichsbasiert.
  Beweis: Übung
- ganzzahlige Schlüssel (stay tuned)
- extern: Geht gut (nichtaddressierbar)
- parallel: Semantik?

Prioritätslisten: Zusammenfassung

- Häufig benötigte Datenstruktur
- Addressierbarkeit ist nicht selbstverständlich
- Binäre Heaps sind einfache, relativ effiziente Implementierung

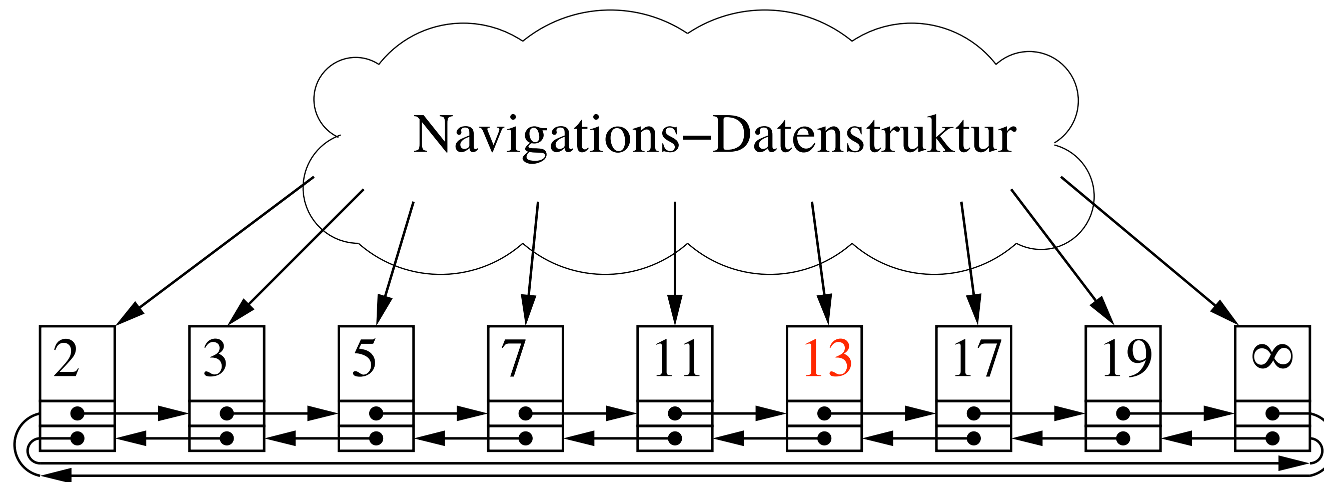
7 Sortierte Folgen

Sortierte Folgen:

$\langle e_1, \ldots, e_n \rangle$ mit $e_1 \leq \cdots \leq e_n$

„kennzeichnende“ Funktion:

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



Annahme: Dummy-Element mit Schlüssel ∞

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

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

Übung: Variante von binärer Suche:
bestimme ℓ, r so dass $a[\ell..r-1] = [k, \ldots, k]$, $a[\ell - 1] < k$ 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: $k = 15$

//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 $\ell + 1 < r$ **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** ℓ:= m

return r

$[-\infty, 2, 3, 5, 7, 11, 13, 17, 19, \infty]$

$[-\infty, 2, 3, 5, 7, 11, 13, 17, 19, \infty]$

$[-\infty, 2, 3, 5, 7, 11, 13, 17, 19, \infty]$

$[-\infty, 2, 3, 5, 7, 11, 13, \underline{17}, 19, \infty]$

Dynamische Sortierte Folgen – Grundoperationen

insert, remove, update, locate $O(\log n)$

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

Mehr Operationen

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

min: Erstes Listenelement — Zeit $O(1)$

max: Letztes Listenelement — Zeit $O(1)$

rangeSearch(a, b) — // $O(\log n + |\text{result}|)$

result:= $\langle\rangle$

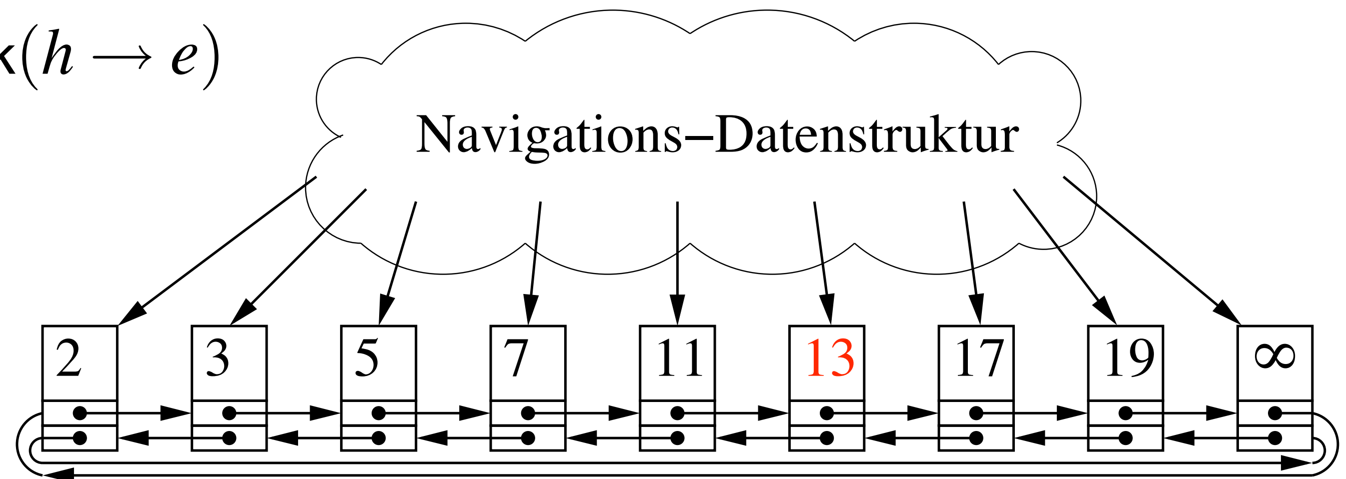
h:= locate(a)

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

result.pushBack$(h \to e)$

h:= h →next

return result



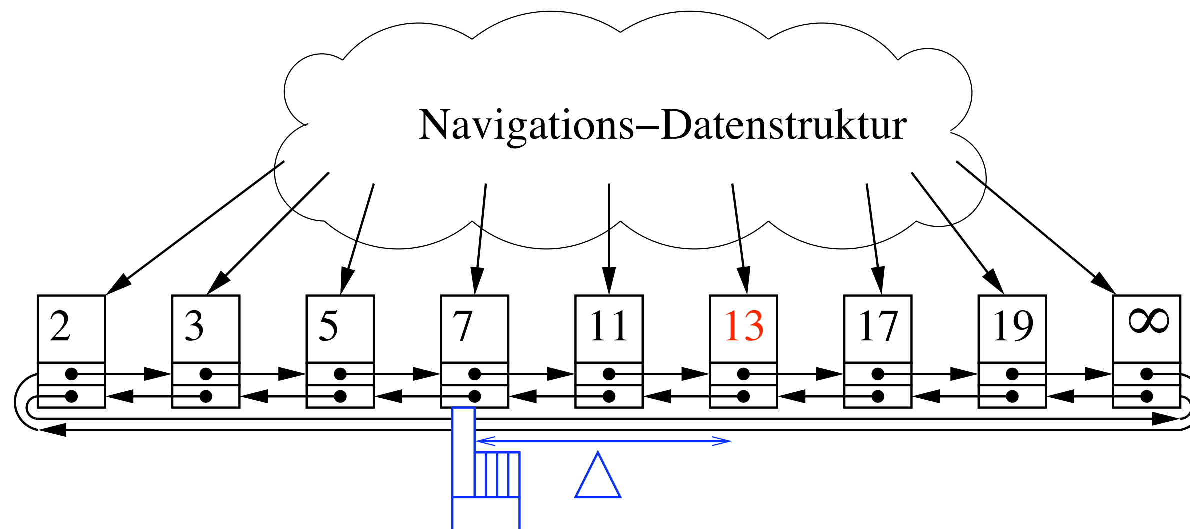
Noch mehr Operationen

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

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

Fingersuche: $\Delta =$ Abstand zu Fingerinfo

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

Insgesamt: die eierlegende Wollmilchdatenstruktur.

„Etwas“ langsamer als speziellere Datenstrukturen

Sortierte Folgen – Anwendungen

- Best-First Heuristiken
- Alg. Geometrie: Sweepline-Datenstrukturen
- Datenbankindex
- ...

Anwendungsbeispiel: Best First Bin Packing

Procedure binPacking(s)

B : SortedSequence // used bins sorted by free capacity

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

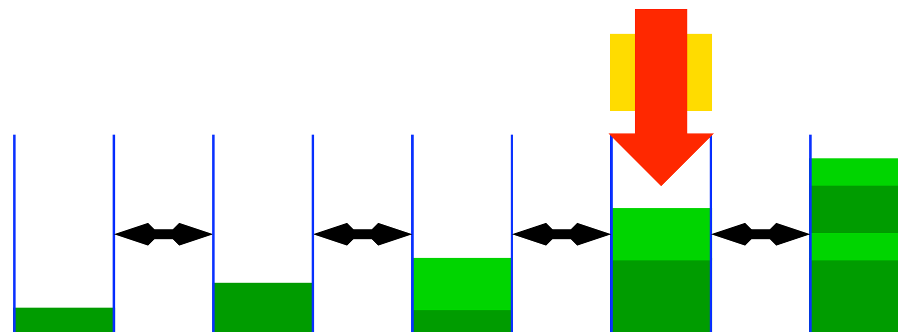
if $\neg\exists b \in B$: free$(b) > e$ **then** B.insert(new bin)

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

insert e into bin b

Zeit: $\mathrm{O}(|s| \log |s|)$

Qualität: „gut“. Details: nicht hier



7.1 Binäre Suchbäume

Blätter: Elemente einer sortierten Folge.

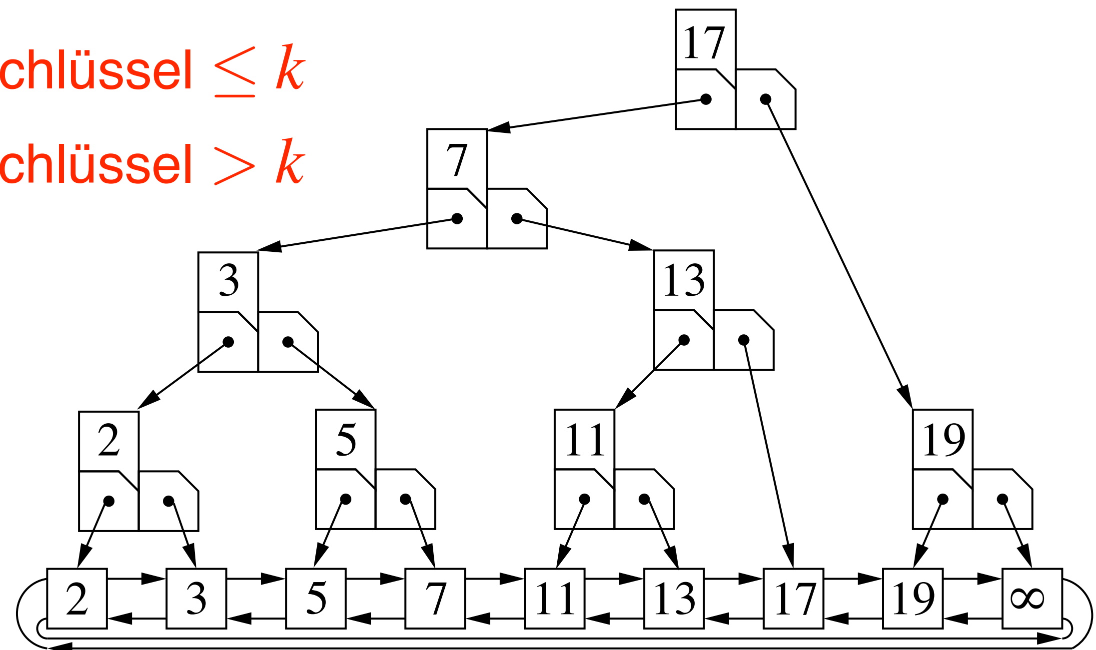
Innere Knoten $v = (k, \ell, r)$,

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

Invariante:

über ℓ erreichbare Blätter haben Schlüssel $\leq k$

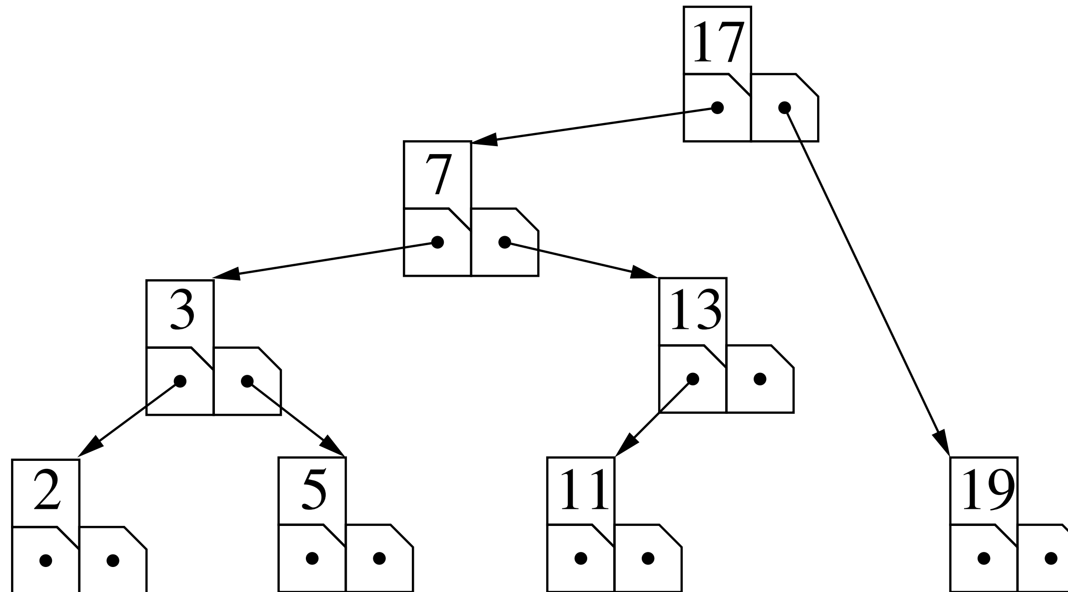
über r erreichbare Blätter haben Schlüssel $> k$



Varianten, Bemerkungen

- Dummy Element im Prinzip verzichtbar
- Oft speichern auch innere Knoten Elemente
- „Suchbaum" wird oft als Synomym 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 $x \geq k$ **then return** x

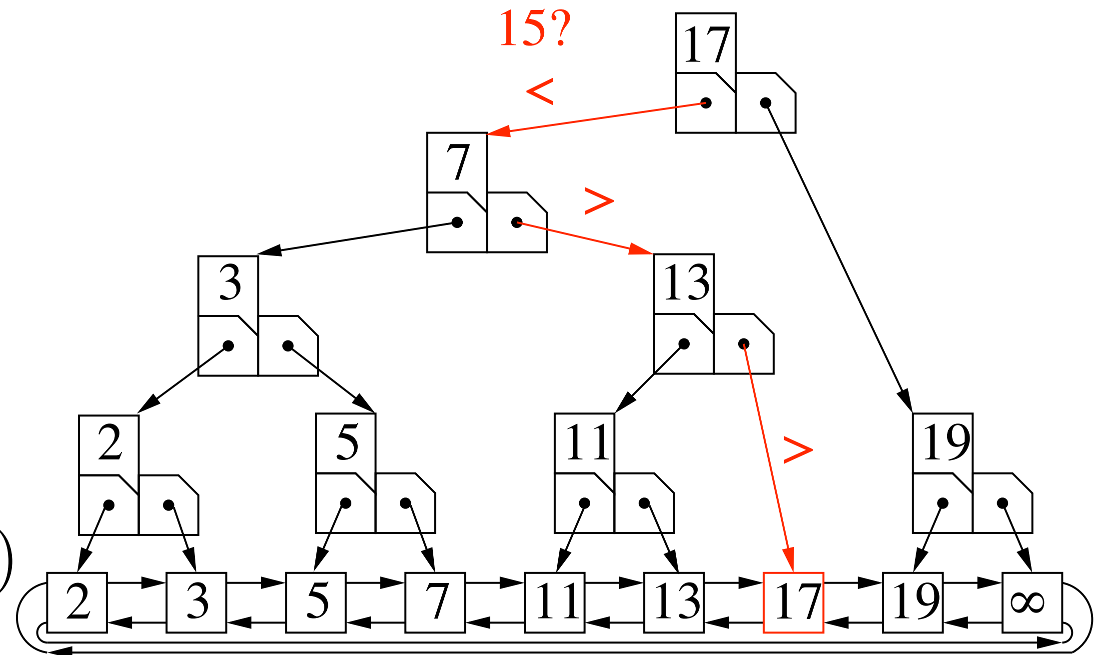
    else return $x \rightarrow$next

  if $k \leq x$ **then**

    return locate$(k, x \rightarrow$left$)$

  else

    return locate$(k, x \rightarrow$right$)$



Invariante: Sei X die Menge aller von x erreichbaren Listenelemente.

Listenelemente links von X sind $< k$

Listenelemente rechts von X sind $\geq k$

Laufzeit von locate(k)

Idee: Benutze Spaltschlüssel x als Wegweiser.

Function locate(k,x)

  if x is a leaf **then**

    if $x \geq k$ **then return** x

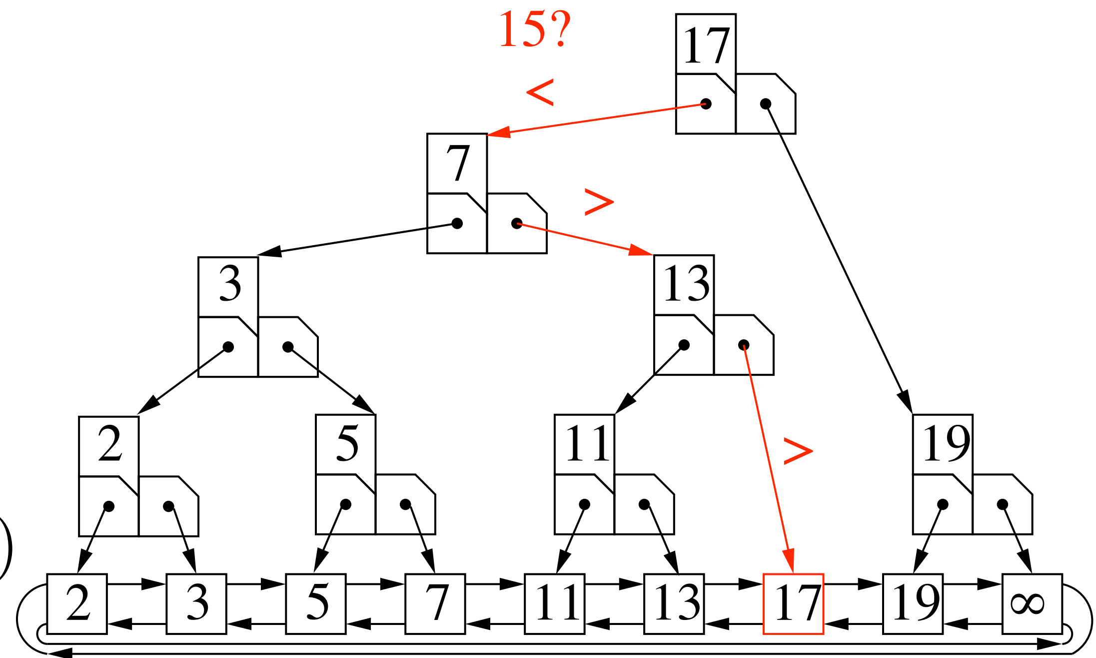
    else return $x \rightarrow$next

  if $k \leq x$ **then**

    return locate$(k, x \rightarrow$left$)$

  else

    return locate$(k, x \rightarrow$right$)$



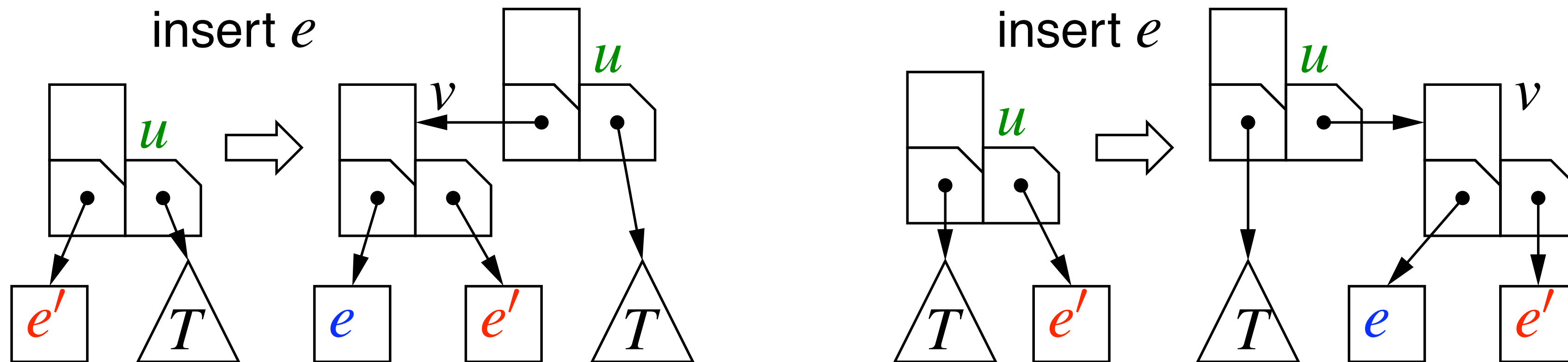
Laufzeit: O(Tiefe).

Bester Fall: perfekt balanciert, d. h. Tiefe $= \lceil \log n \rceil$

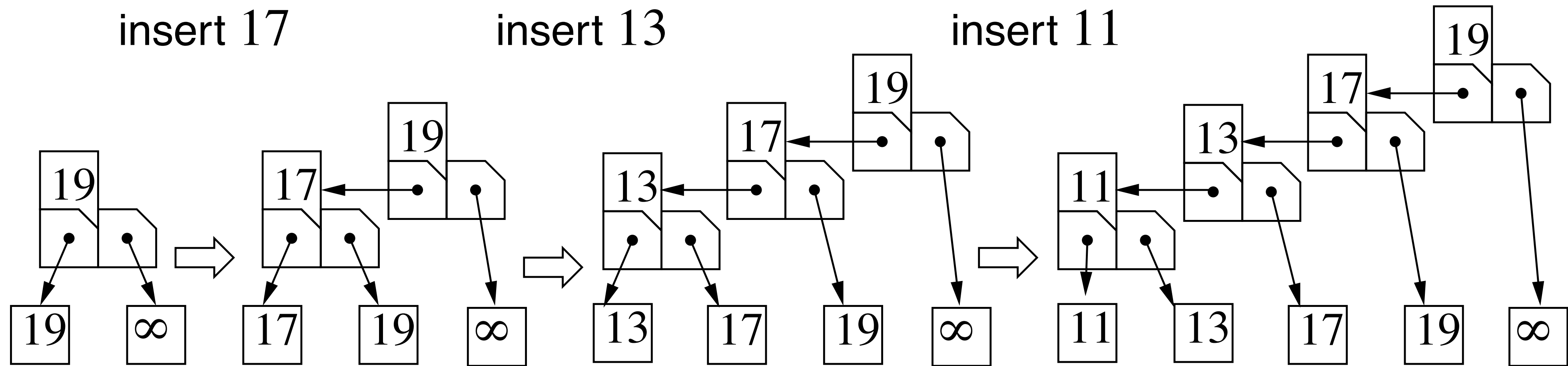
Schlechtester Fall: Tiefe n

Naives Einfügen

Zunächst wie locate(e). Sei e' gefundenes Element, u der Vorgänger



Beispiel



Problem: Der Baum wird beliebig unbalanciert.

$\rightsquigarrow$ langsam

Suchbäume balancieren

perfekte Balance: schwer aufreichzuerhalten

flexible Tiefe $O(\log n)$: balancierte binäre Suchbäume.

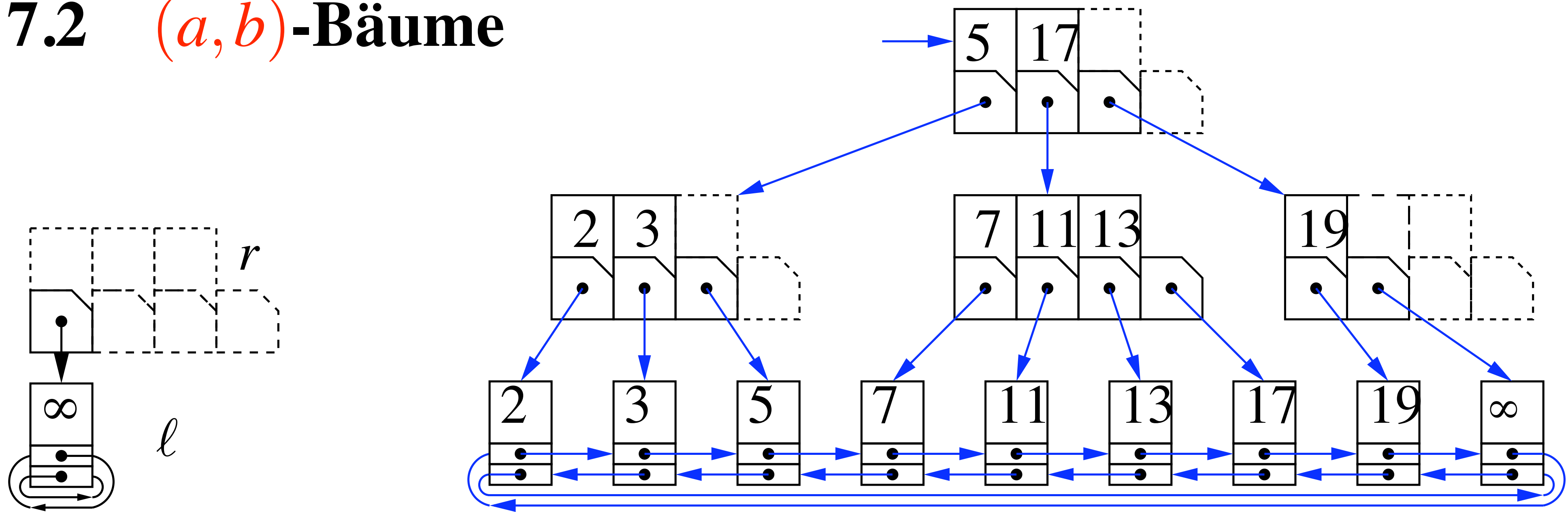
nicht hier (Variantenzoo).

flexibler Knotengrad: (a,b)-Bäume.

$\approx$ Grad zwischen a und b.

Tiefe $\approx \log_a n$

7.2 (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)

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

s=splitters : **Array** $[1..b-1]$ **of** Key

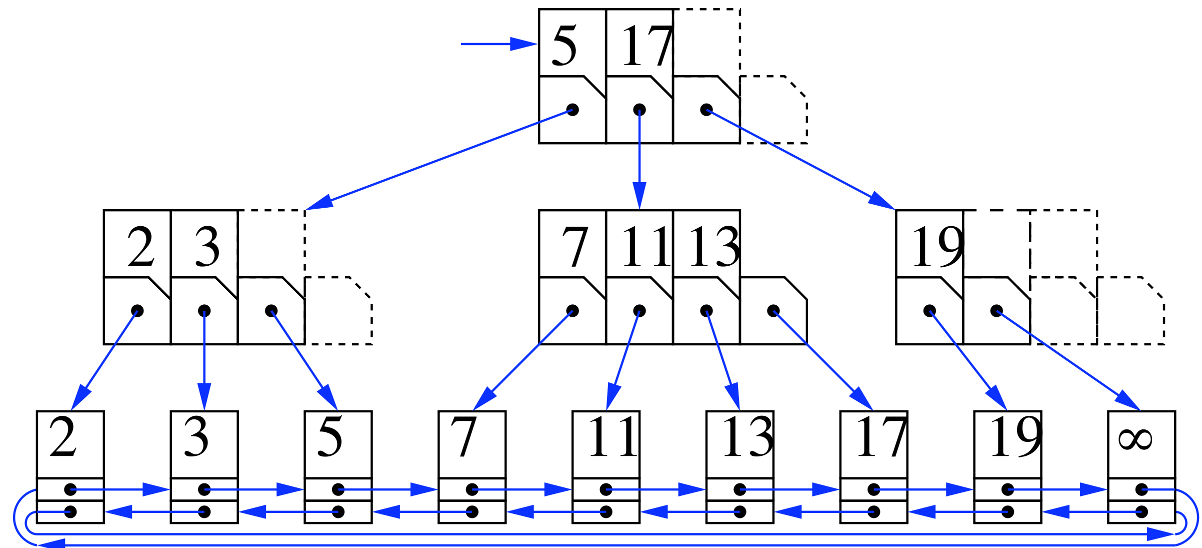
c=children : **Array** $[1..b]$ **of** Handle

Invariante:

e über $c[i]$ erreichbar

$\Rightarrow s[i-1] < e \leq s[i]$ mit

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



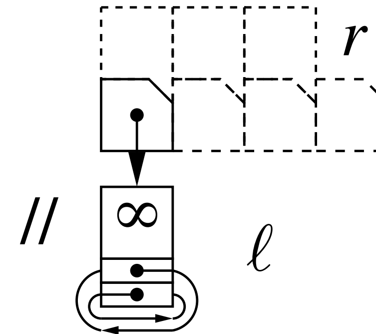
Initialisierung

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.locateRec(k, 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:= locateLocally(k)

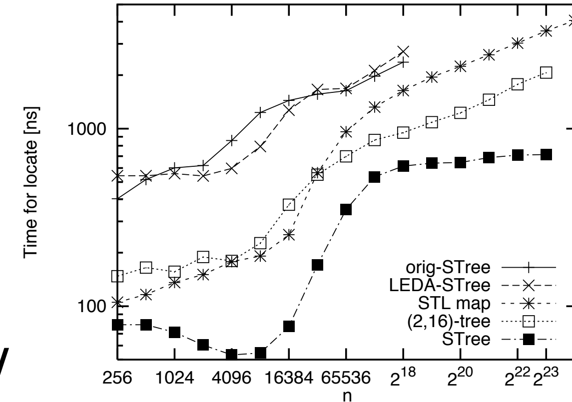
if $h = 1$ **then**

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

else return $c[i] \rightarrow$ next

else

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



Invariante: analog binäre Suchbäume

Locate – Laufzeit

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

Übung: $b \longrightarrow \log b$?

Lemma: $\text{height} = h \leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$

Beweis:

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

Fall $n > 1$:

Wurzel hat Grad ≥ 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 $\square$

Einfügen – Algorithmenskizze

Procedure insert(e)

Finde Pfad Wurzel–nächstes Element e'

ℓ.insertBefore(e, e')

füge key(e) als neuen Splitter in Vorgänger u

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

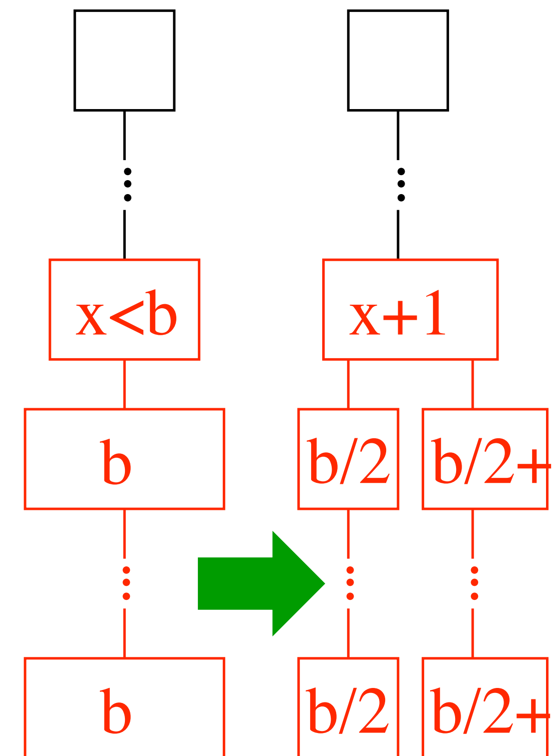
spalte u in 2 Knoten mit Graden

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

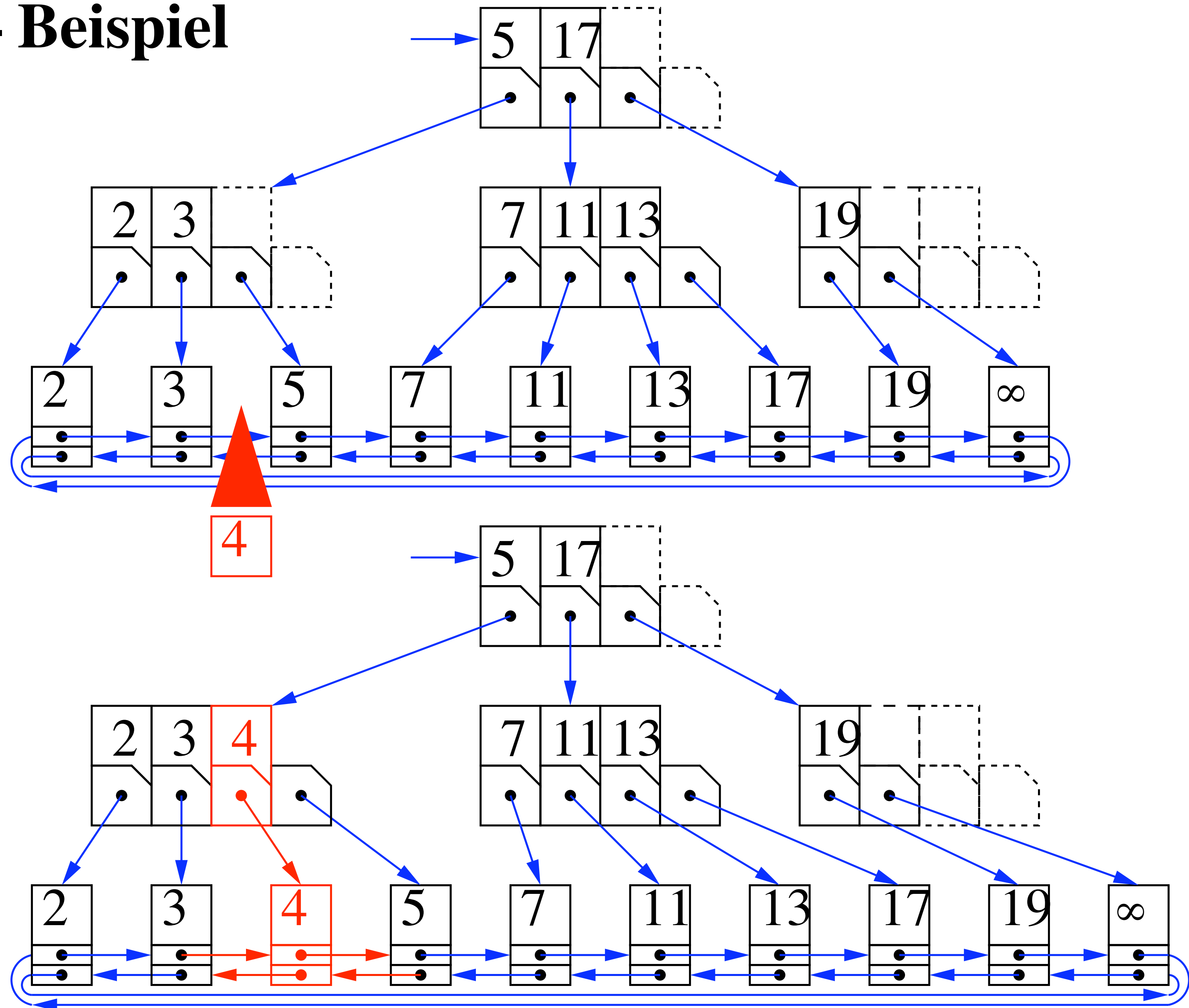
Weiter oben einfügen, spalten

. . .

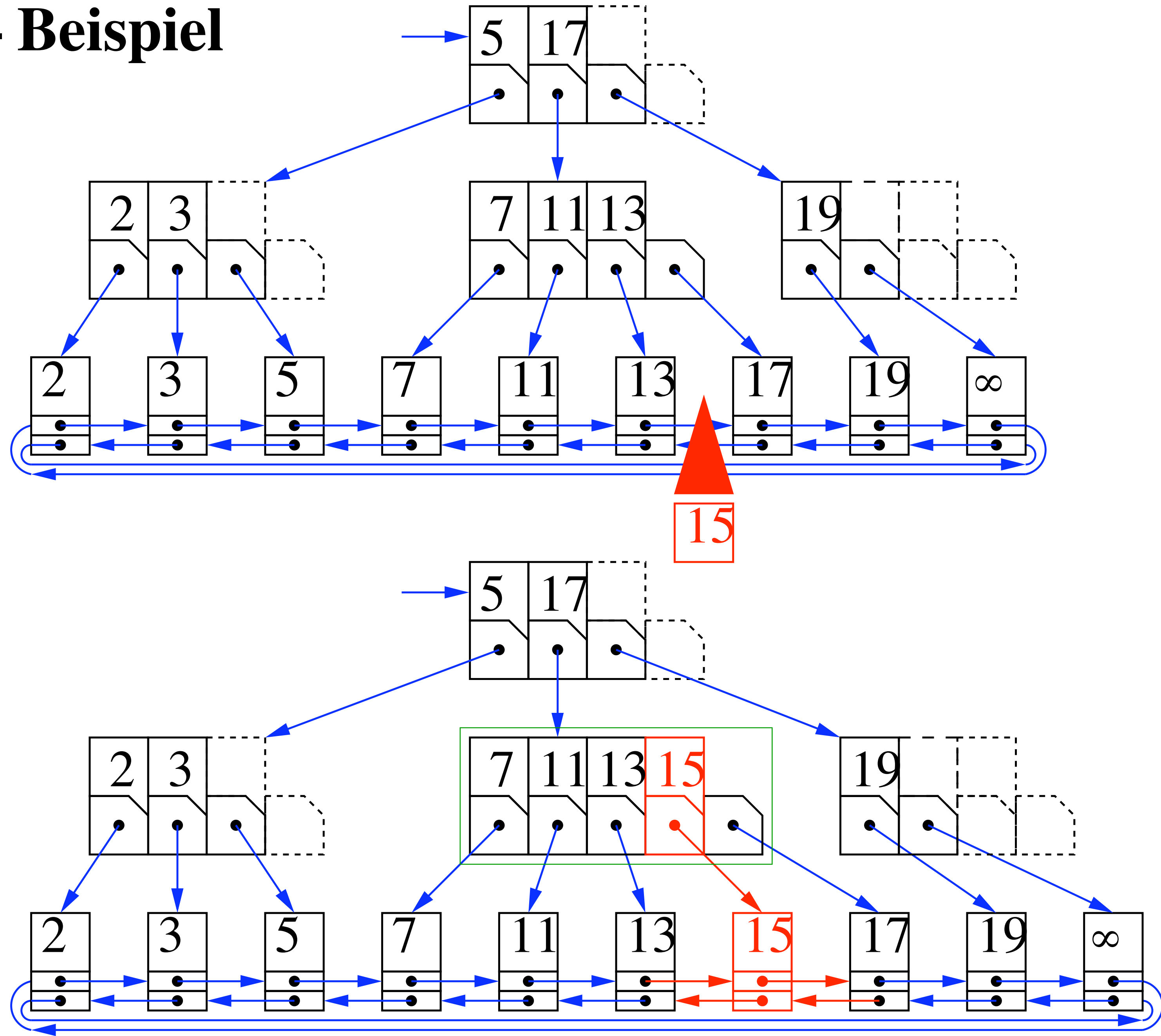
ggf. neue Wurzel



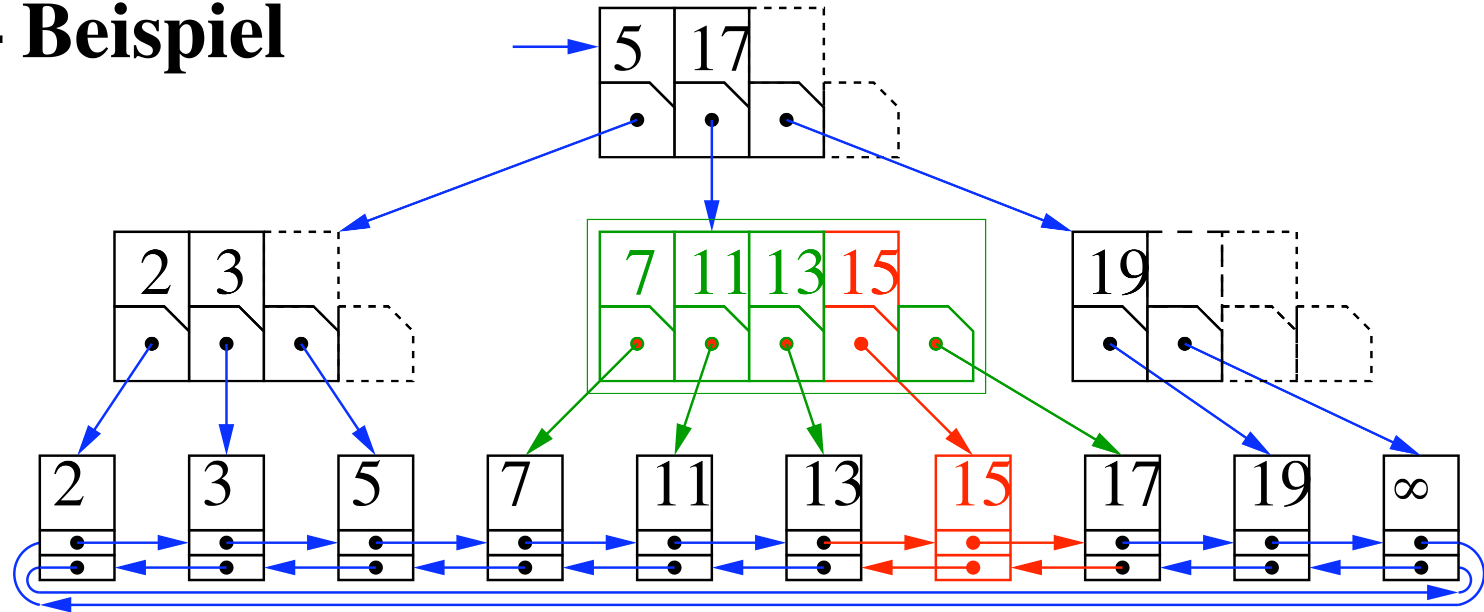
Einfügen – Beispiel

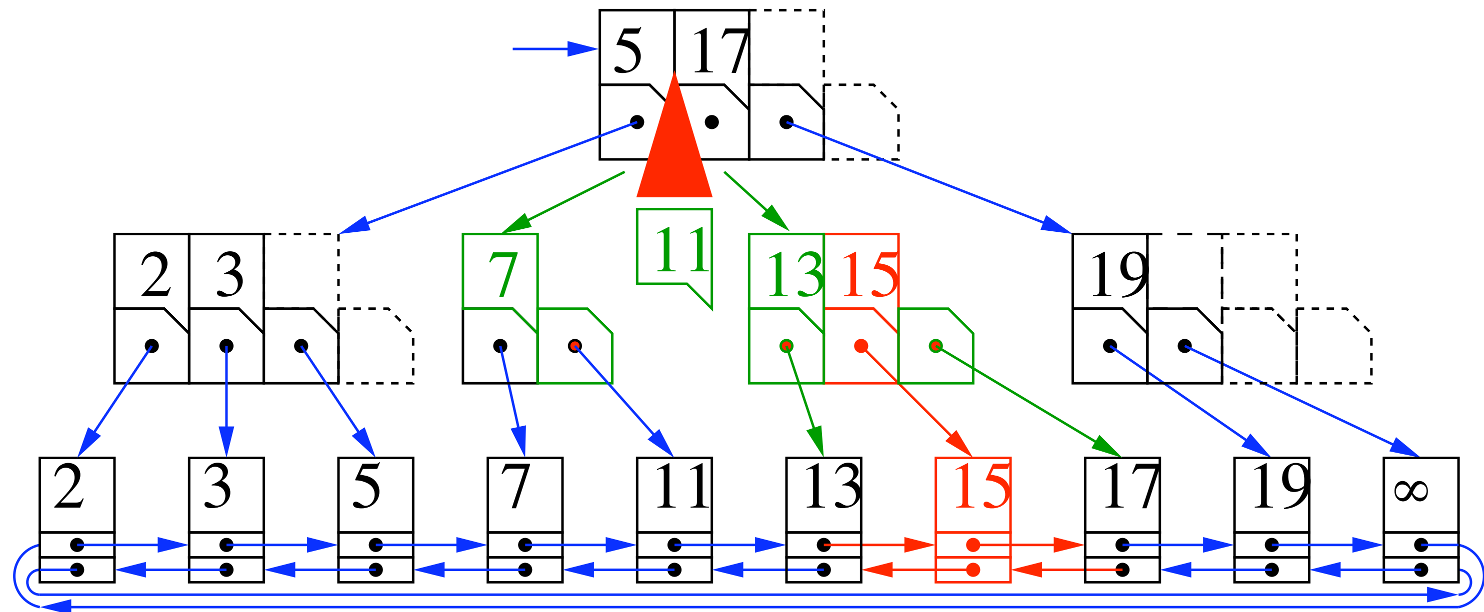


Einfügen – Beispiel

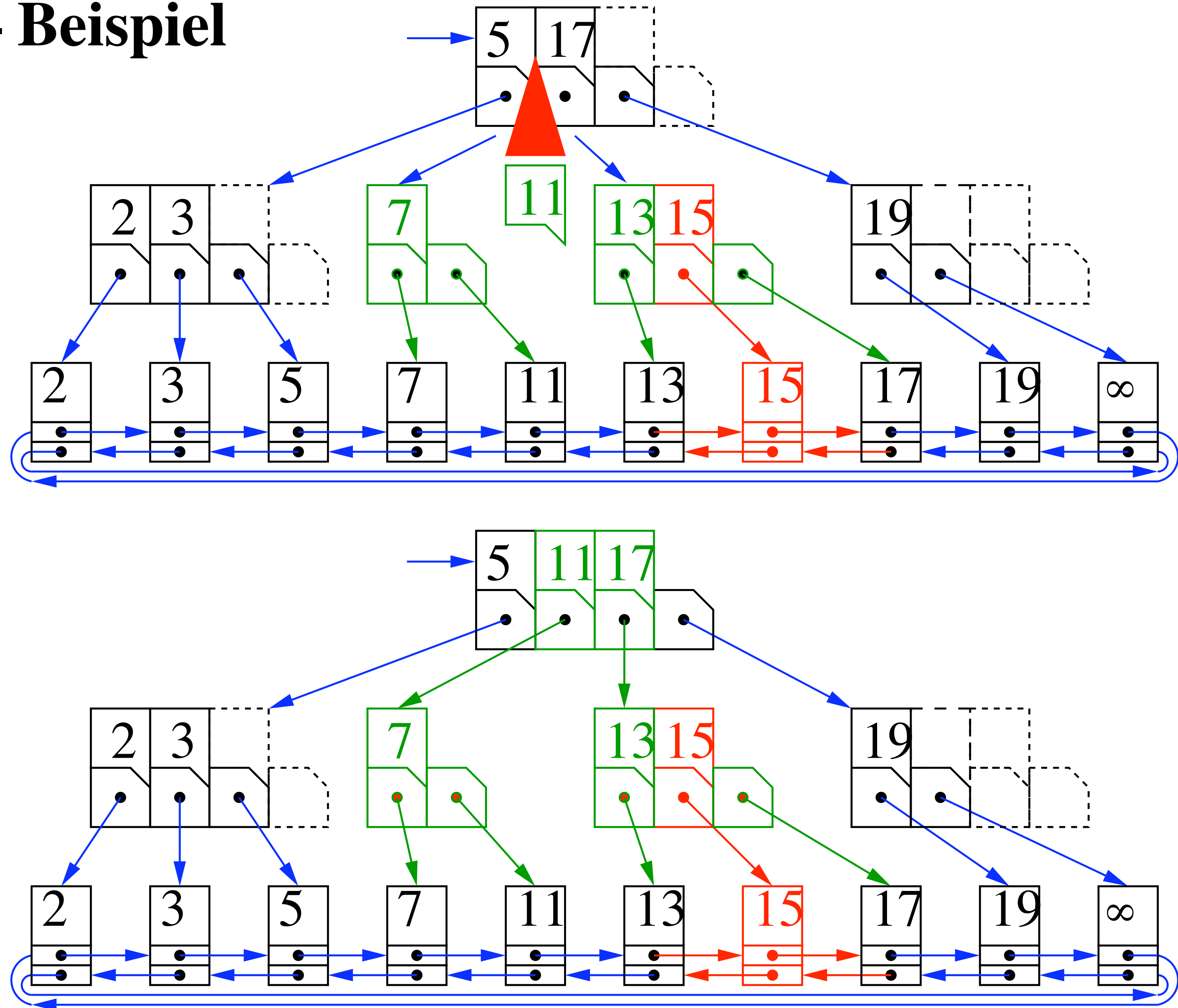


Einfügen – Beispiel

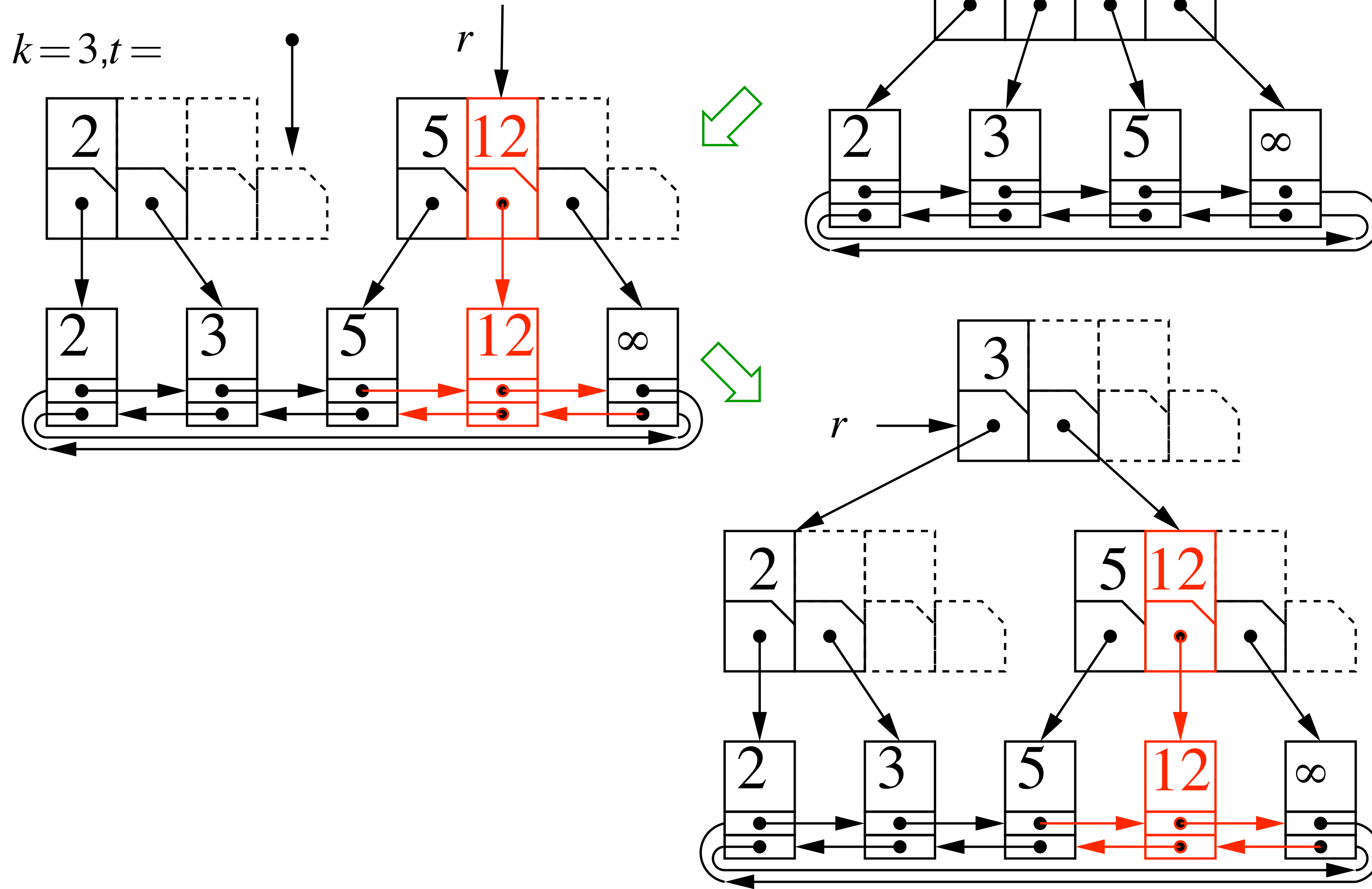




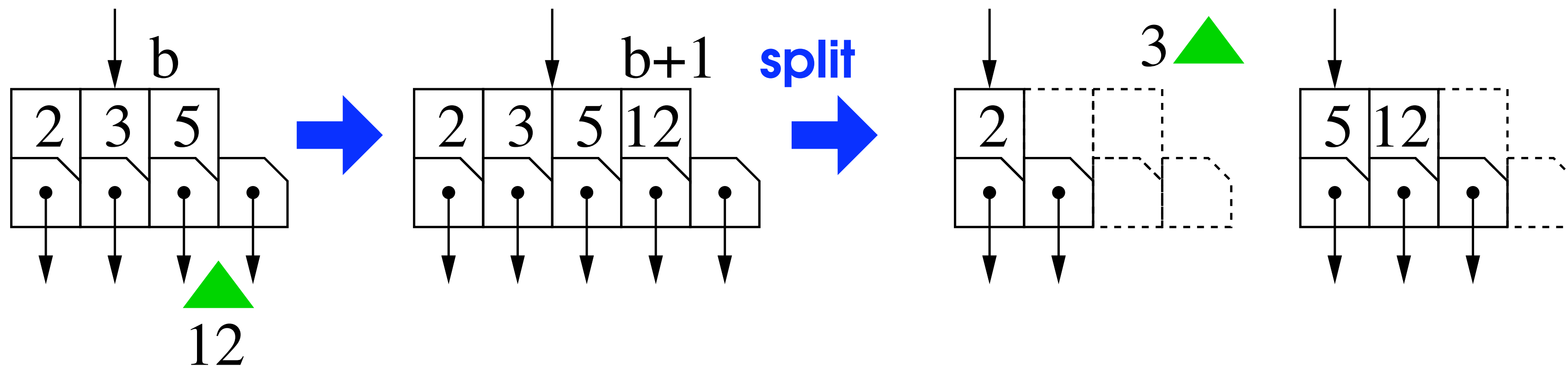
Einfügen – Beispiel



Einfügen – Beispiel



Einfügen – Korrektheit



Nach dem Spalten müssen zulässige Items entstehen:

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

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

Procedure ABTree::insert(e : $Element$)

(k,t):= r.insertRec(e, height, ℓ)

if $t \neq$ **null then**

r:= **allocate** ABItem($\langle k \rangle, \langle r, t \rangle$)

height$++$

```
Function ABItem::insertRec(e : Element, h : ℕ, ℓ : List of Element) : Key×ABHand
    i:= locateLocally(e)
    if h = 1 then (k,t):= (key(e), ℓ.insertBefore(e, c[i]))      // base
    else          (k,t):= c[i] → insertRec(e, h−1, ℓ)            // recurse
                  if t = null then return (⊥, null )
    s′:= ⟨s[1],...,s[i−1], k, s[i],...,s[d−1]⟩                    // new splitter
    c′:= ⟨c[1],...,c[i−1], t, c[i],...,c[d]⟩                      // new child
    if d < b then (s,c,d):= (s′,c′,d+1); return (⊥, null )
    else                                                         // split this node
        d:= ⌊(b+1)/2⌋
        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–e

ℓ.remove(e)

entferne key(e) in Vorgänger u

if $u.d = a - 1$ **then**

finde Nachbarn u'

if $u'.d > a$ **then** balance(u', u)

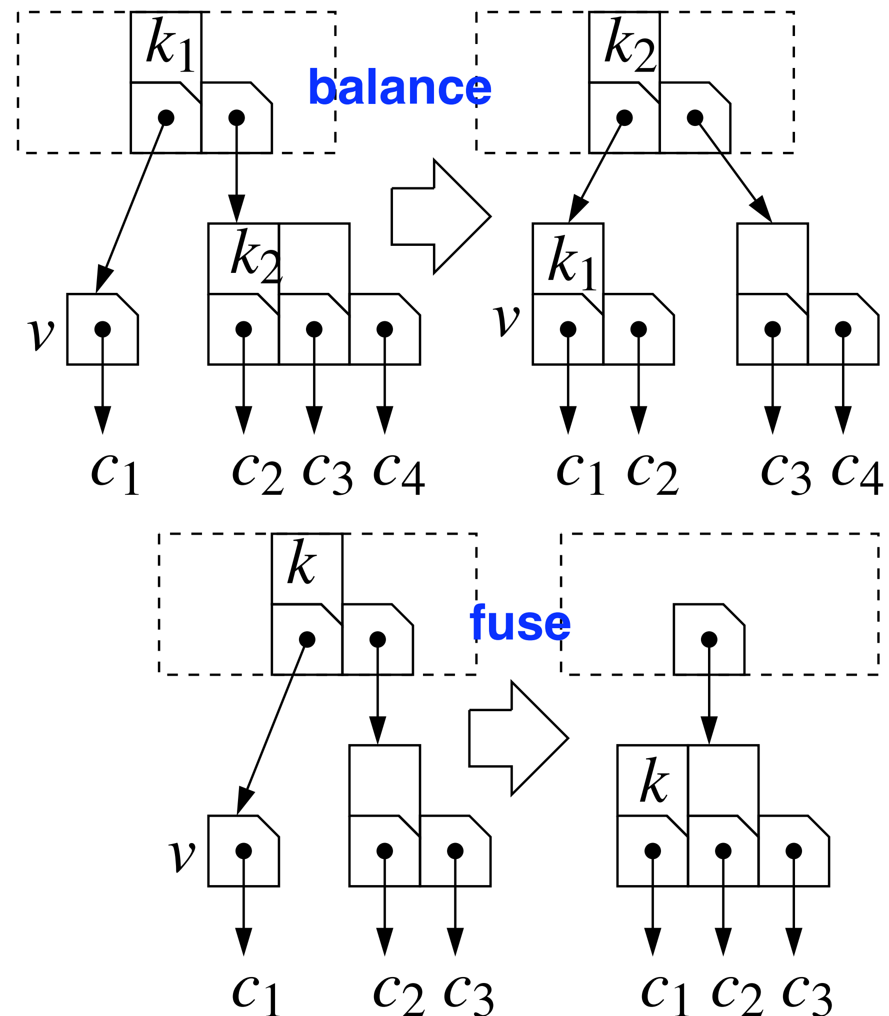
else

fuse(u', u)

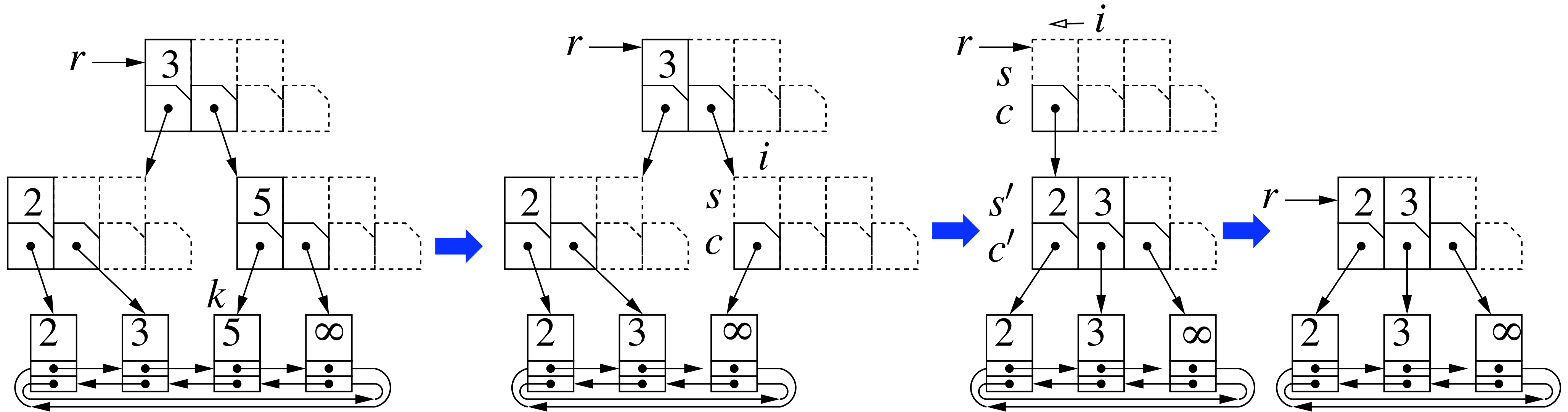
Weiter oben splitter entfernen

...

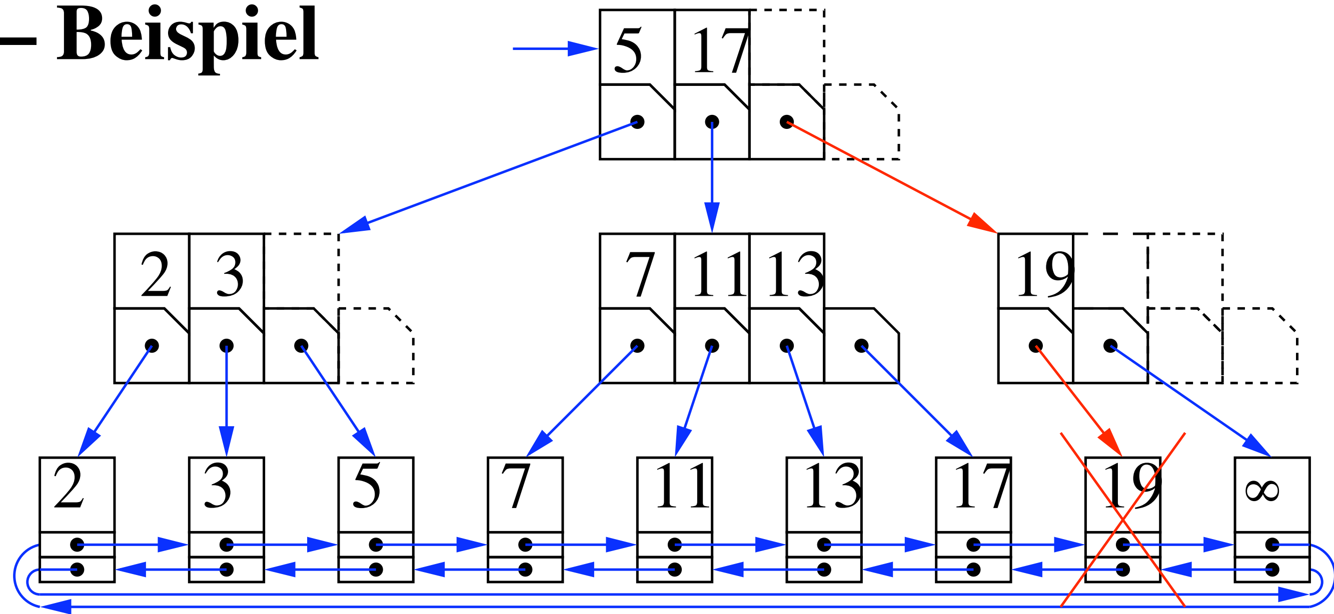
ggf. Wurzel entfernen

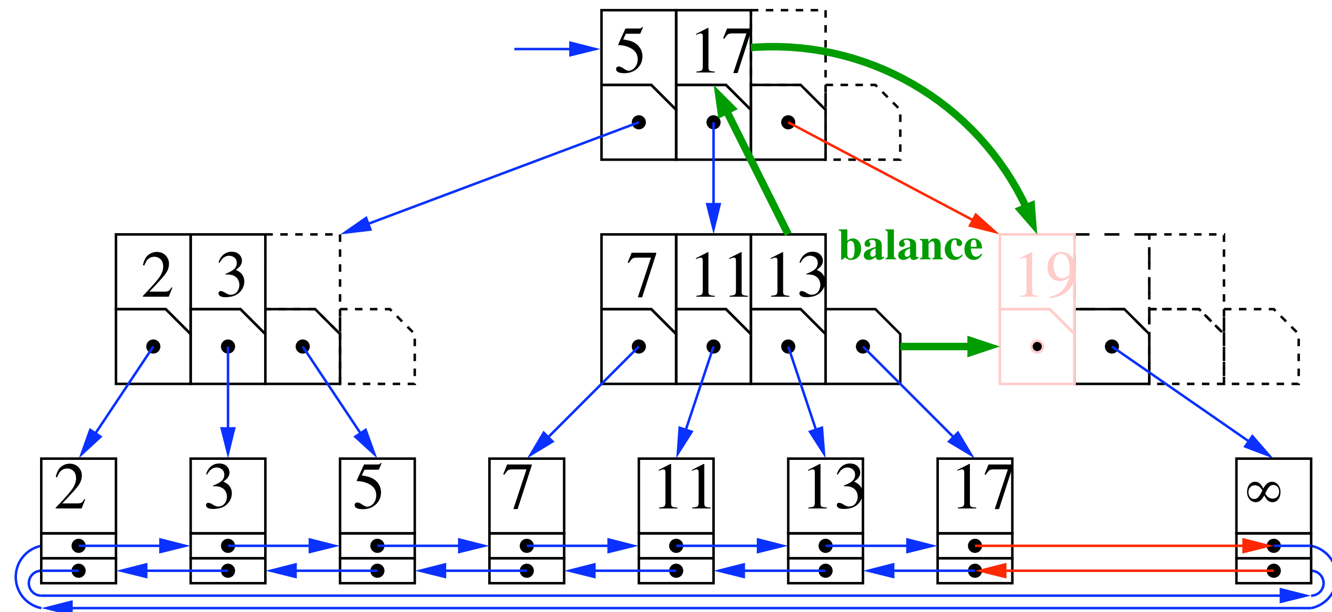


Entfernen – Beispiel

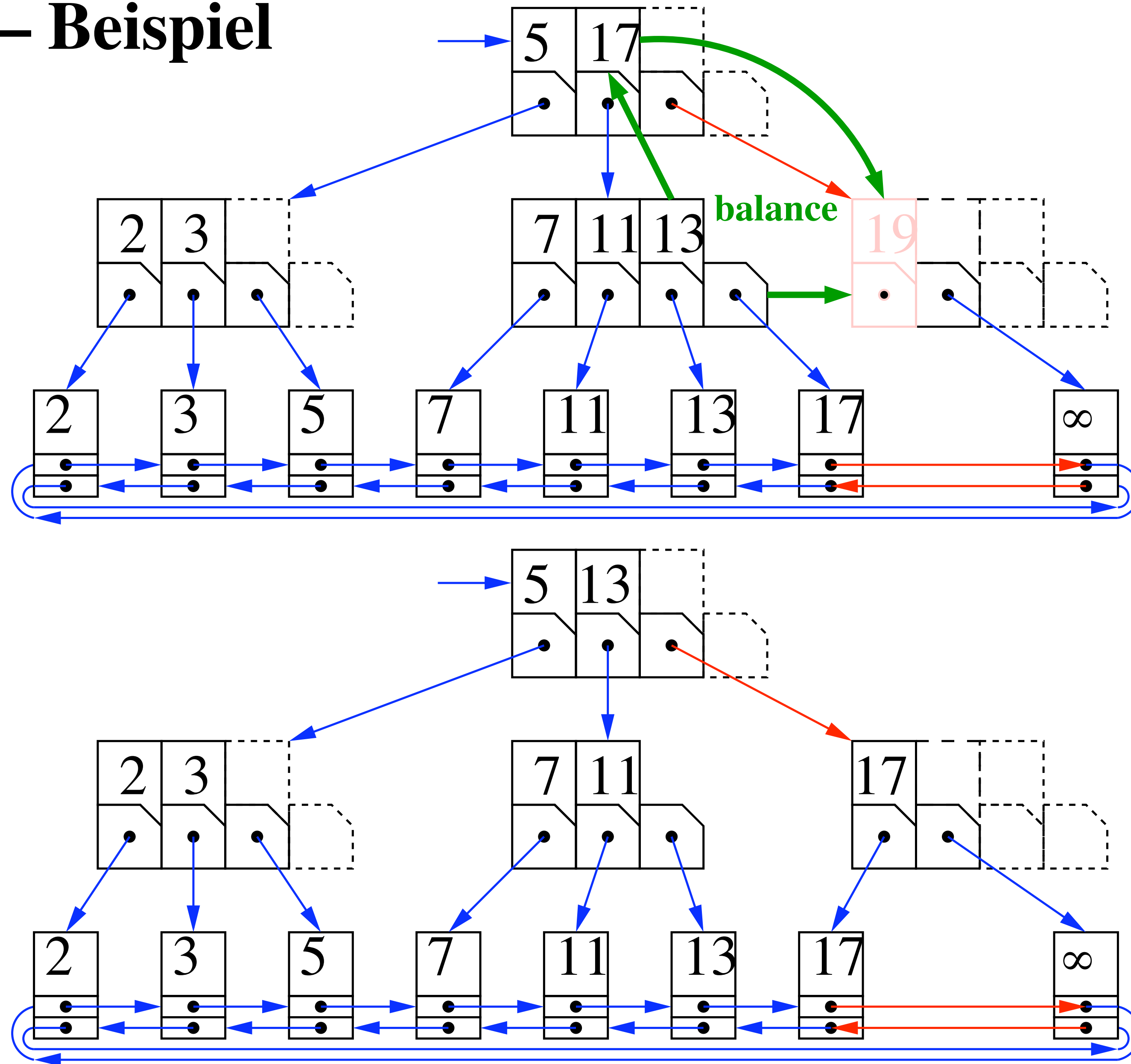


Entfernen – Beispiel





Entfernen – Beispiel



Entfernen – Korrektheit

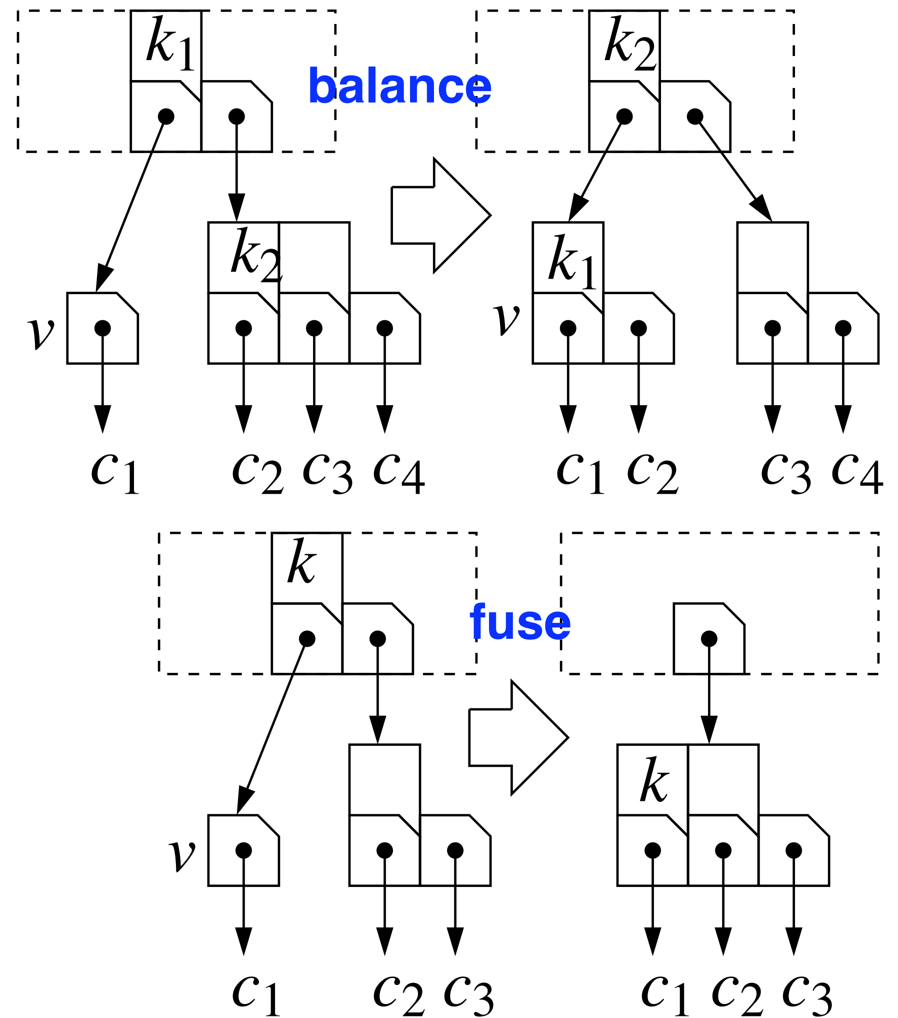
Balancieren: Kein Problem

Nach fuse

müssen zulässige Items entstehen:

$$a + (a-1) \overset{!}{\leq} b \Leftrightarrow b \geq 2a - 1$$

hatten wir schon!



Einfügen und Entfernen – Laufzeit

$$\mathrm{O}(b \cdot \text{Tiefe}) = \mathrm{O}(b \log_a n)$$

$$= \mathrm{O}(\log n) \text{ für } \{a,b\} \subseteq \mathrm{O}(1)$$

(a,b)-Bäume Implementierungsdetails

Etwas kompliziert. . .

Wie merkt man sich das?

Gar nicht!

Man merkt sich:

- Invarianten

  Tiefe, Knotengrade

- Grundideen

  split, balance, fuse

Den Rest leitet man

sich nach Bedarf neu her.

```
Procedure ABTree::remove(k : Key)
    r.removeRec(k, height, ℓ)
    if r.d = 1 ∧ height > 1 then r′:= r; r:= r′.c[1]; dispose r′
Procedure ABItem::removeRec(k : Key, h : ℕ, ℓ : List of Element)
    i:= locateLocally(k)
    if h = 1 then
        if key(c[i] → e) = k then
            ℓ.remove(c[i])
            removeLocally(i)
    else
        c[i] → removeRec(e, h − 1, ℓ)
        if c[i] → d < a then
            if i = d then i−−
            s′:= concatenate(c[i] → s, ⟨s[i]⟩, c[i+1] → s))
            c′:= concatenate(c[i] → c, c[i+1] → c)
            d′:= |c′|
            if d′ ≤ b then    // fuse
                (c[i+1] → s, c[i+1] → c, c[i+1] → d):= (s′, c′, d′)
                dispose c[i];   removeLocally(i)
            else              // balance
                m:= ⌈d′/2⌉
                (c[i] → s, c[i] → c, c[i] → d):= (s′[1..m−1], c′[1..m], m)
                (c[i+1] → s, c[i+1] → c,   c[i+1] → d) :=
                    (s′[m+1..d′−1], c′[m+1..d′],   d′ − m)
                s[i]:= s′[m]
Procedure ABItem::removeLocally(i : ℕ)
    c[i..d−1]:= c[i+1..d]
    s[i..d−2]:= s[i+1..d−1]
    d−−
```

7.3 Mehr Operationen

min, max, rangeSearch(a,b): $\langle \min, \ldots, a, \ldots, b, \ldots, \max \rangle$

hatten wir schon

build: Übung! Laufzeit $O(n)$!

(Navigationstruktur für sortierte Liste aufbauen)

concat, **split**: nicht hier. Zeit $O(\log n)$

Idee: Ganze Teilbäume umhängen

merge(N,M): sei $n = |N| \leq m = |M|$ Zeit $O\left(n \log \frac{m}{n}\right)$

nicht hier. Idee: z. B. Fingersuche

7.4 Amortisierte Analyse von insert und remove

nicht hier.

Grob gesagt: Abgsehen von der Suche fällt nur konstant viel Arbeit an (summiert über alle Operationsausführungen).

7.5 Erweiterte (augmentierte) Suchbäume

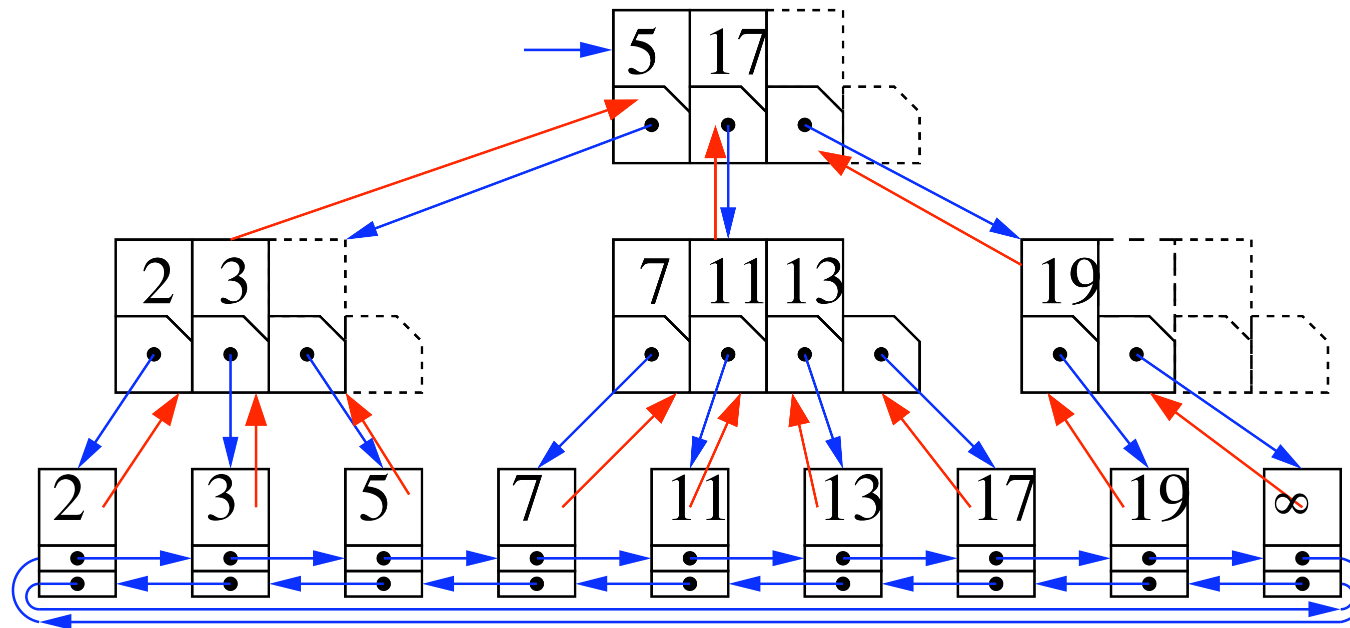
Idee: zusätzliche Infos verwalten $\rightsquigarrow$ mehr (schnelle) Operationen.

Nachteil: Zeit- und Platzverschwendung

wenn diese Operationen nicht wichtig sind. gold plating

7.5.1 Elternzeiger

Idee (Binärbaum): Knoten speichern Zeiger auf Elternknoten



Anwendungen: schnelleres remove, insertBefore, insertAfter,

wenn man ein handle des Elements kennt.

Man spart die Suche.

Frage: was speichert man bei (a,b)-Bäumen?

7.5.2 Teilbaumgrößen

Idee (Binärbaum): speichere wieviele Blätter von links erreichbar.

(Etwas anders als im Buch!)

// return k-th Element in subtree rooted at h

Function selectRec(h, k)

  if $h \rightarrow$ leftSize $> k$ **then return** select(ℓ, k)

  else return select$(r, k -$ leftSize$)$

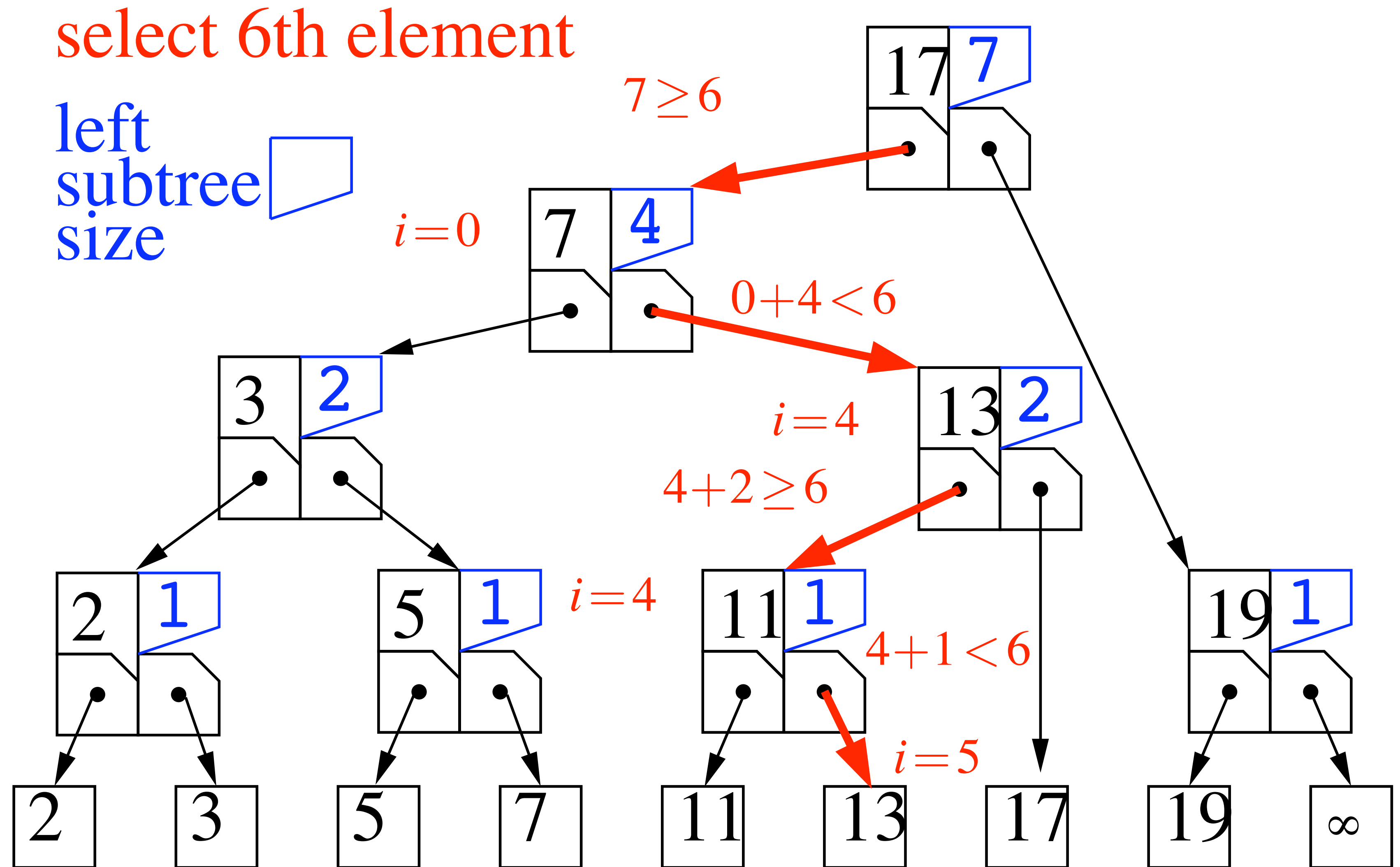
Zeit: $\mathrm{O}(\log n)$

Übung: Was ist anders bei (a, b)-Bäumen?

Übung: Rang eines Elements e bestimmen.

Übung: Größe eines Range $a..b$ bestimmen.

7.5.3 Beispiel



Zusammenfassung

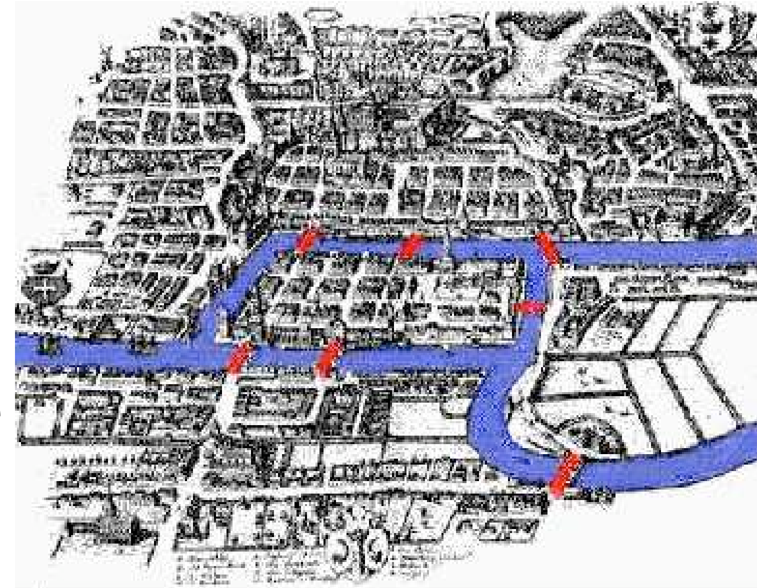
- Suchbäume erlauben viele effiziente Operationen auf sortierten Folgen.
- Oft logarithmische Ausführungszeit
- Der schwierige Teil: logarithmische Tiefe erzwingen.
- Augmentierungen $\rightsquigarrow$ zusätzliche Operationen

Mehr zu sortierten Folgen

- Karteikasten $\rightsquigarrow$ Array mit Löchern
- (a,b)-Bäume sind wichtig für externe Datenstrukturen
- Ganzzahliche Schlüssel aus $1..U$

  $\rightsquigarrow$ Grundoperationen in Zeit $\mathrm{O}(\log\log U)$
- Verallgemeinerungen: Zeichenketten, mehrdimensionale Daten

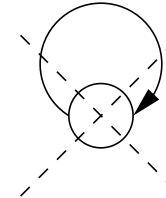
8 Graphrepräsentation

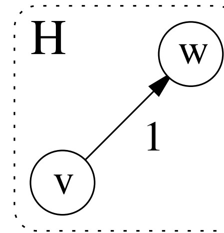


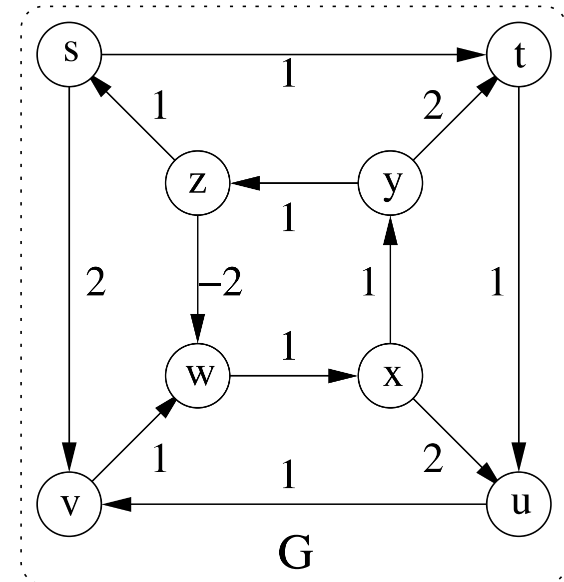
- 1736 fragt L. Euler die folgende "touristische" Frage:
- Straßen- oder Computernetzwerke
- Zugverbindungen (Raum und Zeit)
- Soziale Netzwerke (Freundschafts-, Zitier-, Empfehlungs-,. . .)
- Aufgabenabhängigkeiten ⇝ scheduling Probleme
- Werte und arithmetische Operationen ⇝ Compilerbau
- . . .

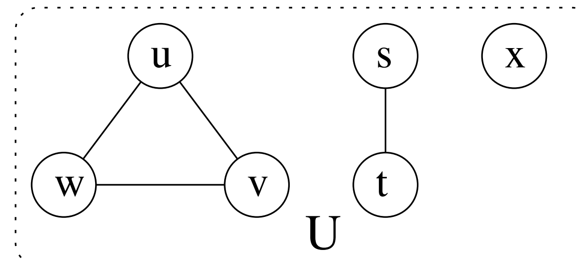
Graphrepräsentation

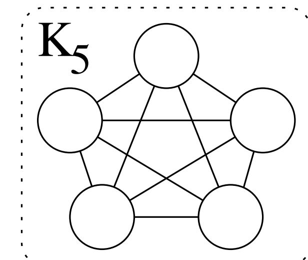
- Was zählt sind die Operationen
- Eine triviale Repräsentation
- Felder
- Verkettete Listen
- Matrizen
- Implizit
- Diskussion

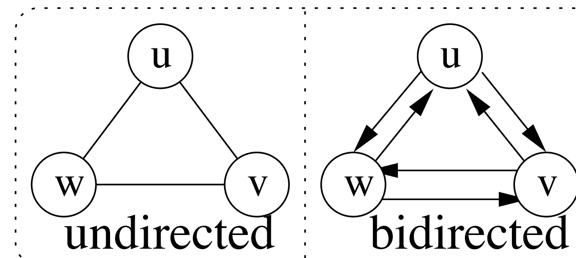


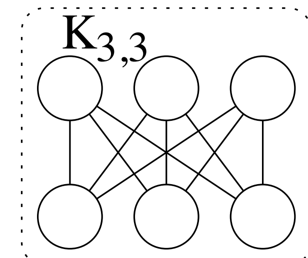












Notation und Konventionen

Graph $G = (\underbrace{V}_{\text{Knoten}}, \underbrace{E}_{\text{Kanten}})$:

$n = |V|$

$m = |E|$

Knoten: s, t, u, v, w, x, y, z

Kanten $e \in E$. Knotenpaare (manchmal Knotenmengen der Größe 2)

WICHTIG: Buchstabenzuordnungen $=$ unverbindliche Konvention

- Manchmal werden ganz andere Buchstaben verwendet.
- Im Zweifel immer genau sagen was was ist.

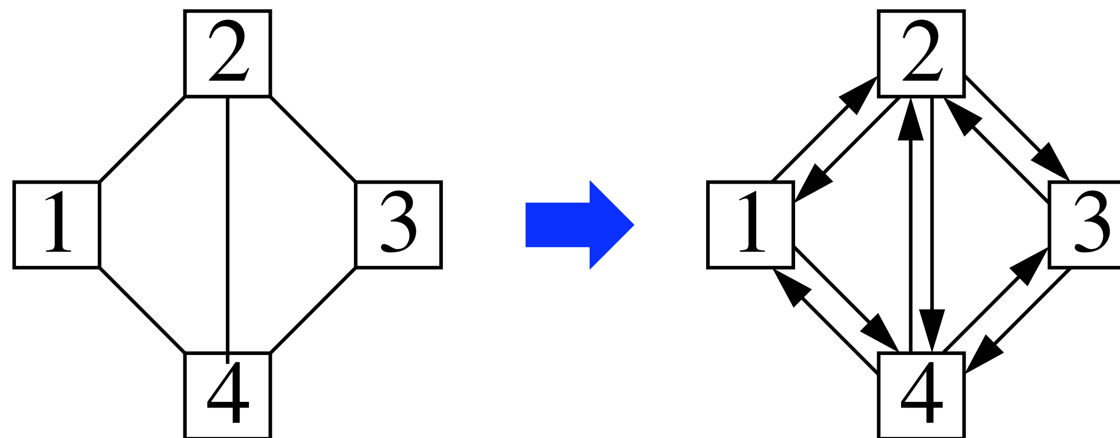
Das gilt für die ganze theoretische Informatik

Ungerichtete → gerichtete Graphen

Meist repräsentieren wir

ungerichtete Graphen durch bigerichtete Graphen

$\rightsquigarrow$ wir konzentrieren uns auf gerichtete Graphen



Operationen

Ziel: $\mathrm{O}(\text{Ausgabegröße})$ für alle Operationen

Grundoperationen

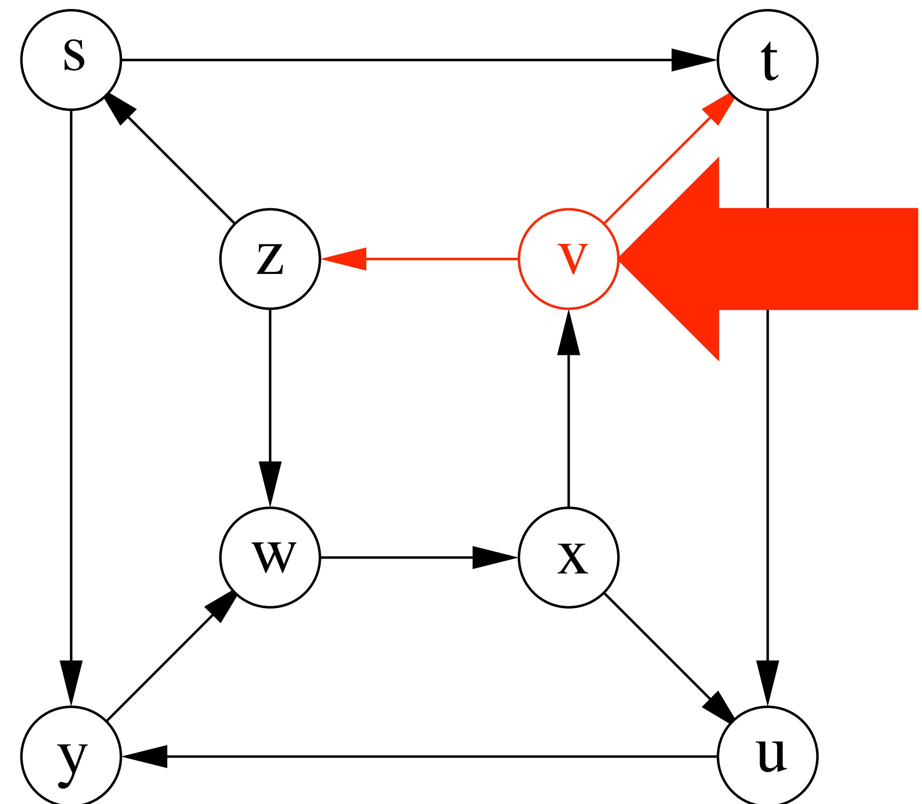
Statische Graphen

Konstruktion, Konversion und Ausgabe

($\mathrm{O}(m+n)$ Zeit)

Navigation: Gegeben v,

finde ausgehende Kanten.



Dynamische Graphen

Knoten/Kanten einfügen/löschen

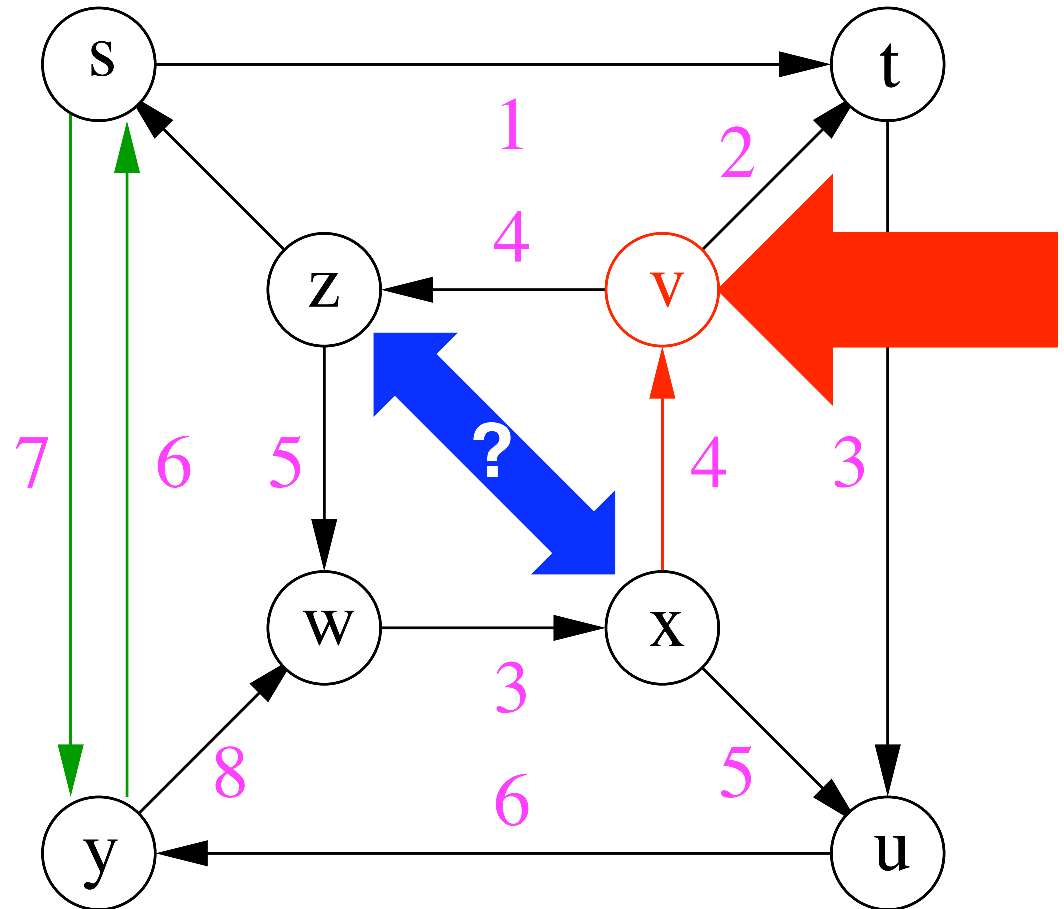
Weitere Operationen

Zugriff auf assoziierte Information

mehr Navigation: finde

eingehende Kanten

Kantenanfragen: $(z,x) \in E$?



Kantenfolgenrepräsentation

Folge von Knotenpaaren (oder Tripel mit Kantengewicht)

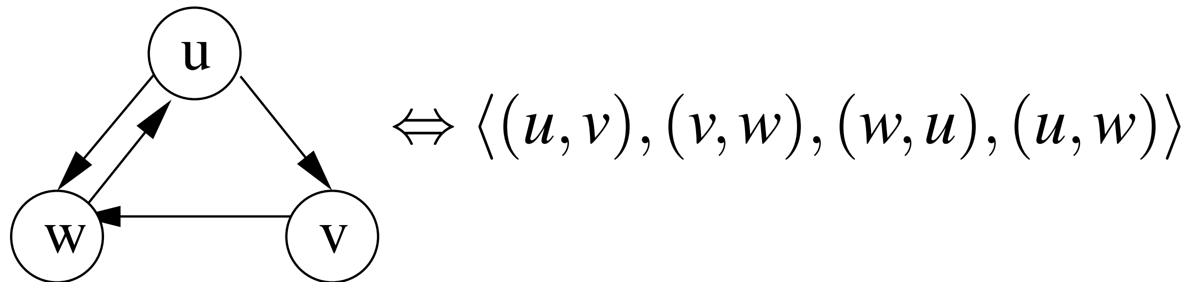
+ kompakt

+ gut für I/O

− Fast keine nützlichen Operationen ausser alle Kanten durchlaufen

Beispiele: Übung: isolierte Knoten suchen,

Kruskals MST-Algorithmus (später), Konvertierung.



$\Leftrightarrow \langle (u,v),(v,w),(w,u),(u,w) \rangle$

Adjazenzfelder

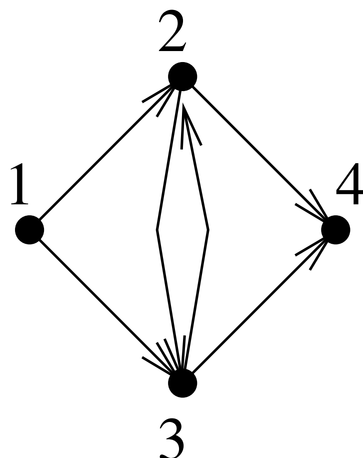
$V = 1..n$ oder $0..n-1$

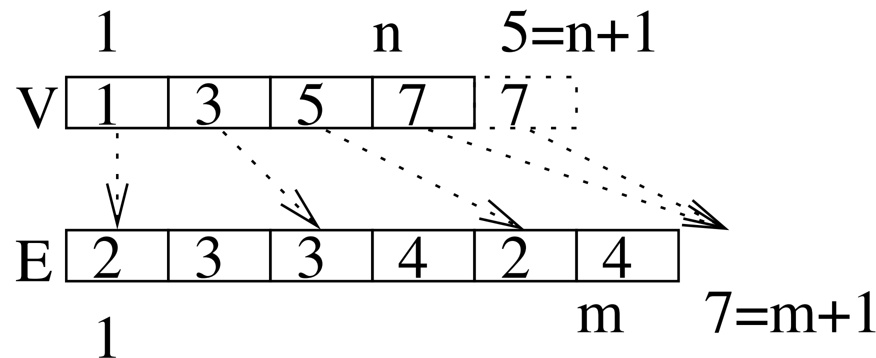
Kantenfeld E speichert Ziele

gruppiert nach Startknoten

V speichert Index der ersten ausgehenden Kante

Dummy-Eintrag $V[n+1]$ speichert $m+1$





Beispiel: Ausgangsgrad$(v) = \mathtt{V}[v+1] - \mathtt{V}[v]$

Kantenliste → Adjazenzfeld

Zur Erinnerung: KSort

Function adjacencyArray(EdgeList)

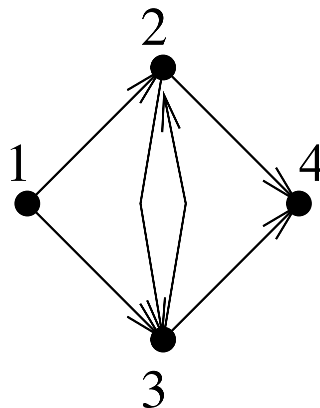
  V=$\langle 0,\ldots,0\rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$

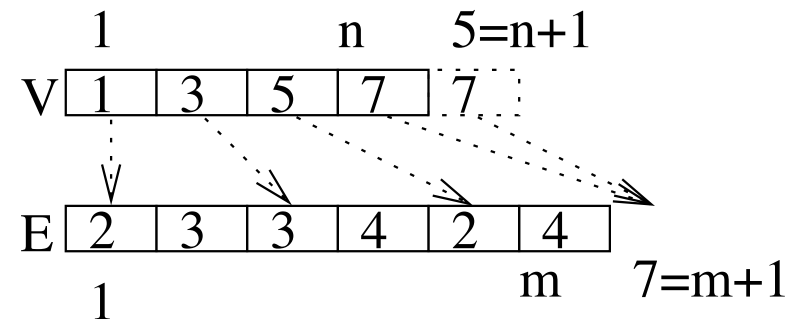
  foreach $(u,v) \in$ EdgeList **do** $V[u]$++ // count

  for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums

  foreach $(u,v) \in$ EdgeList **do** $E[--V[u]] = v$ // place

  return (V,E)





Operationen für Adjanzenzfelder

Navigation: einfach

Kantengewichte: E wird Feld von Records (oder mehrere Felder)

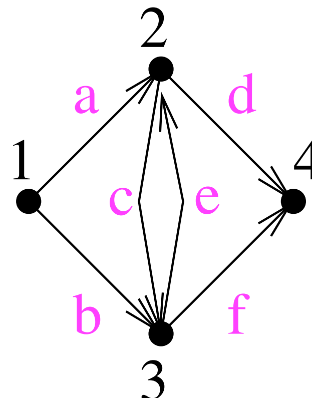
Knoteninfos: V wird Feld von Records (oder mehrere Felder)

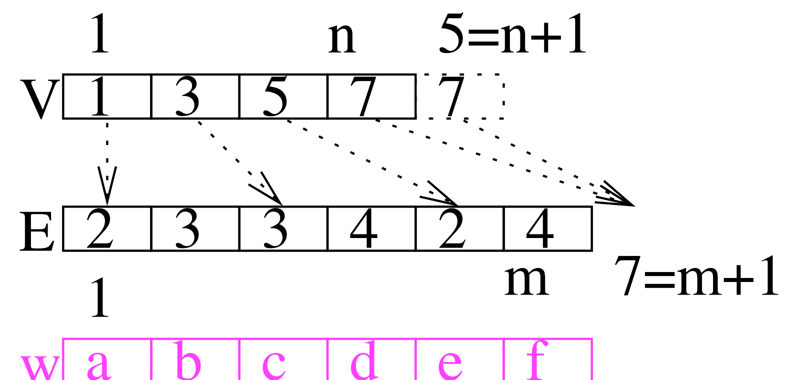
Eingehende Kanten: umgedrehten Graphen speichern

Kanten löschen: explizite Endindizes

Batched Updates:

neu aufbauen





Kantenanfragen

Hashtabelle H_E speichert (ggf. zusätzlich) alle Kanten.

Unabhängig von der sonstigen Graphrepräsentation

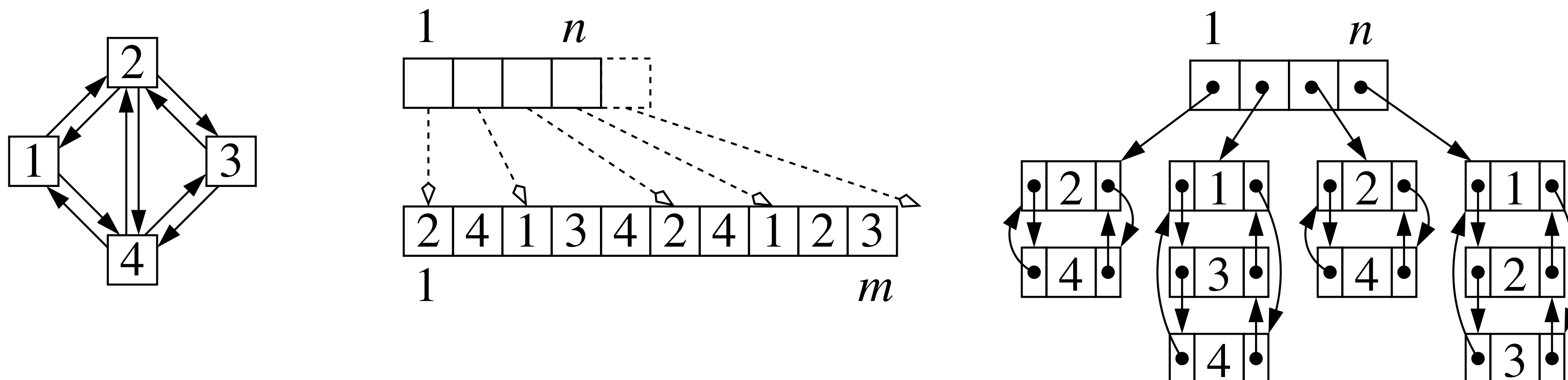
Adjazenzlisten

speichere (doppelt) verkettete Liste adjazenter Kanten für jeden Knoten.

+ einfaches Einfügen von Kanten
+ einfaches Löschen von Kanten (ordnungserhaltend)

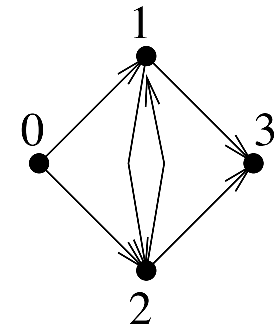
− mehr Platz (bis zu Faktor 3) als Adjazenzfelder

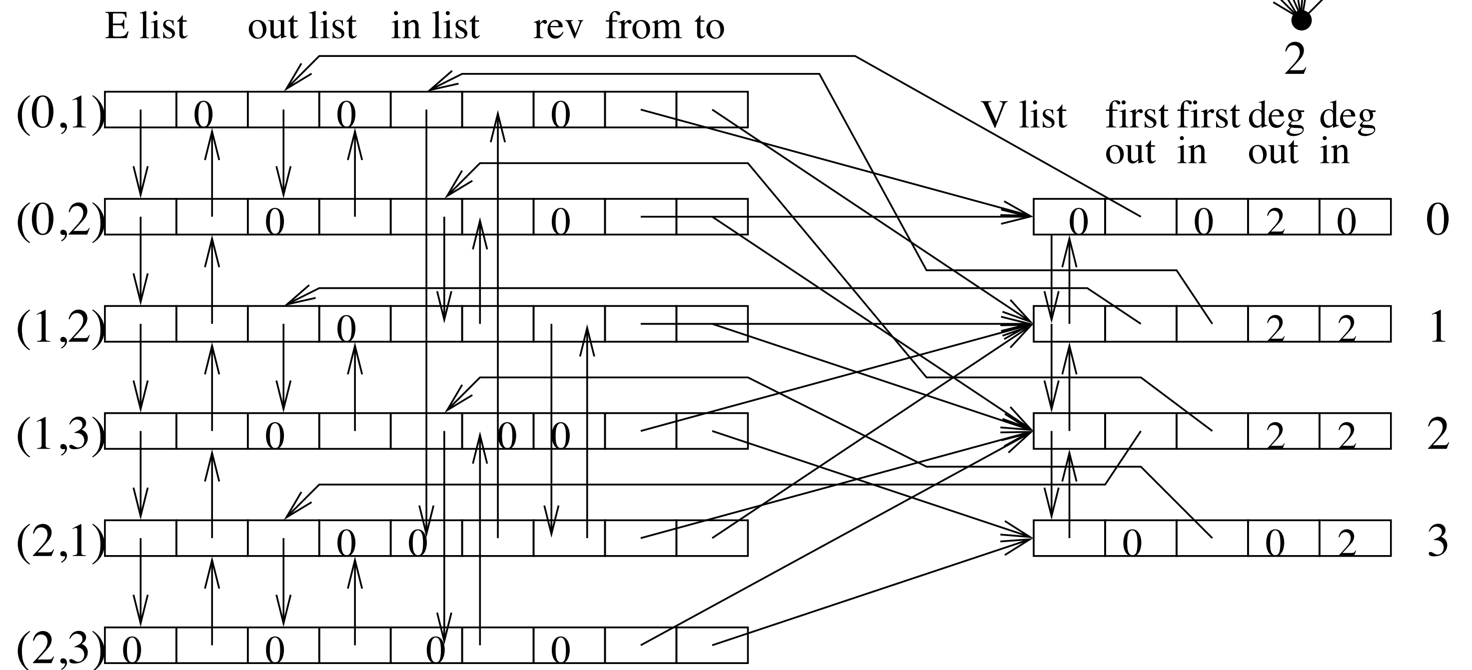
− mehr Cache-Misses



Adjazenzlisten aufrüsten

- Knotenlisten für Knotenupdates
- Ausgehende 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

Software Engineering Alptraum

Möglicher Ausweg: Trennung von Algorithmus und Repräsentation

Beispiel: DAG-Erkennung

```
Function isDAG(G = (V, E))                                   // Adjazenzarray!
    dropped:= 0
    compute array inDegree of indegrees of all nodes        // Zeit O(m)!
    droppable={v ∈ V : inDegree[v] = 0}  : Stack
    while droppable ≠ ∅ do
        invariant G is a DAG iff the input graph is a DAG
        v:= droppable.pop
        dropped++
        foreach edge (v, w) ∈ E do
            inDegree[w]−−
            if inDegree[w] = 0 then droppable.push(w)
    return |V| = dropped
```

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: $\mathbf{C} = \mathbf{A}^k$. $\mathbf{C}_{ij} =\#$ k-Kanten-Pfade von i nach j

Übung: zähle Pfade der Länge $\leq k$

Wichtige Beschleunigungstechniken:

$\mathrm{O}(\log k)$ Matrixmult. für Potenzberechnung

Matrixmultiplikation in subkubischer Zeit, z. B., Strassens Algorithmus

Beispiel wo Graphentheorie bei LA hilft

Problem: 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 so dass

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} .$$

Übung: Was wenn G ein DAG ist?

Implizite Repräsentation

Kompakte Repräsentation möglicherweise sehr dichter Graphen

Implementiere Algorithmen direkt mittels dieser Repr.

Beispiel: Intervall-Graphen

Knoten: Intervalle $[a,b] \subseteq \mathbb{R}$

Kanten: zwischen überlappenden Intervallen

Zusammenhangstest für Intervallgraphen

$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \left\{\left\{[a_i, b_i], [a_j, b_j]\right\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\right\}$

Idee: durchlaufe Intervalle von links nach rechts. Die Anzahl überlappender Intervalle darf nie auf Null sinken.

Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0, 1\}$

remove first element of L; overlap := 1
foreach $p \in L$ **do**
  if overlap$=0$ **return** 0
  if p is a start point **then** overlap$++$
  else overlap$--$ // end point
return 1

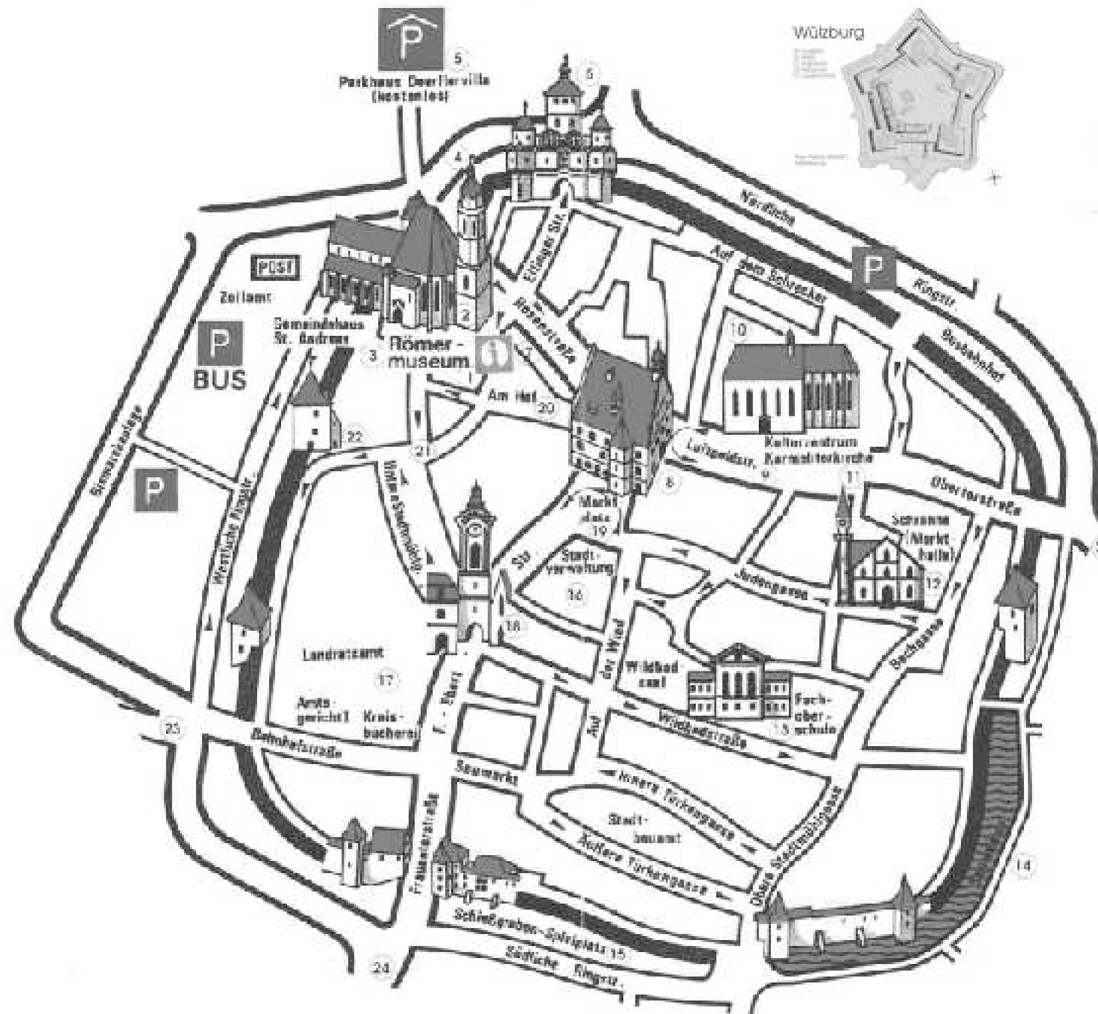
$O(n \log n)$ Algorithmus für bis zu $O\left(n^2\right)$ Kanten!

Übung: Zusammenhangskomponenten finden

Graphrepräsentation: Zusammenfassung

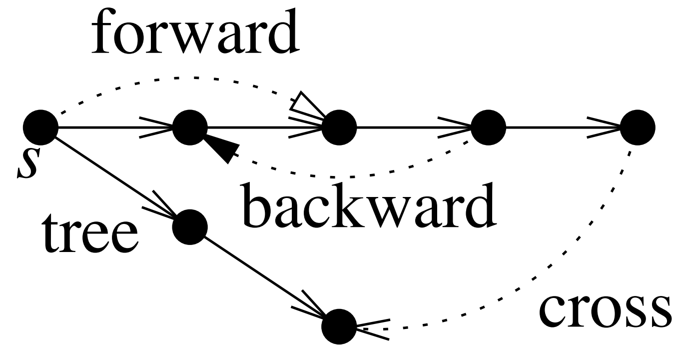
- Welche Operationen werden gebraucht?
- Wieoft?
- Adjazenzarrays gut für statische Graphen
- Pointer $\leadsto$ flexibler aber auch teurer
- Matrizen eher konzeptionell interessant

9 Graphtraversierung



Ausgangspunkt oder Baustein fast jedes nichtrivialen Graphenalgorithmus

Graphtraversierung als Kantenklassifizierung

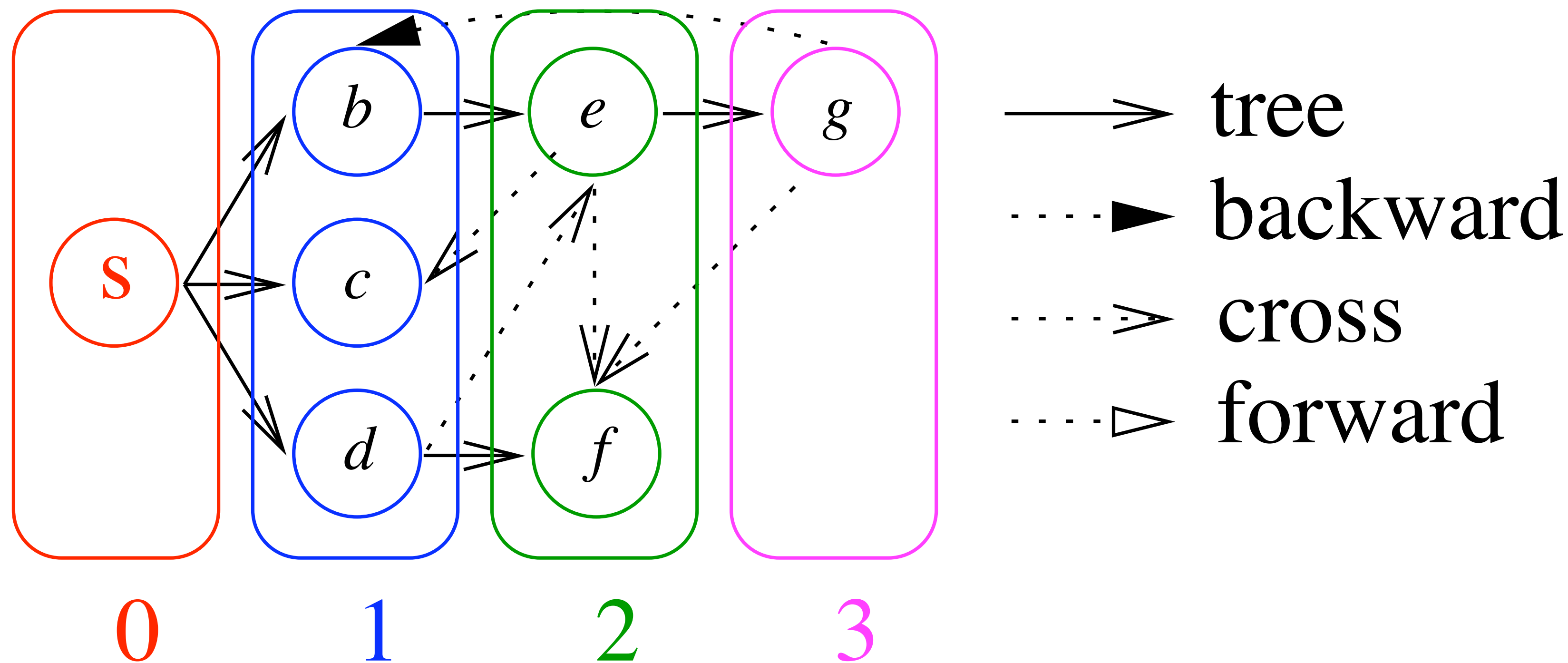


9.1 Breitensuche

Baue Baum von Startknoten s

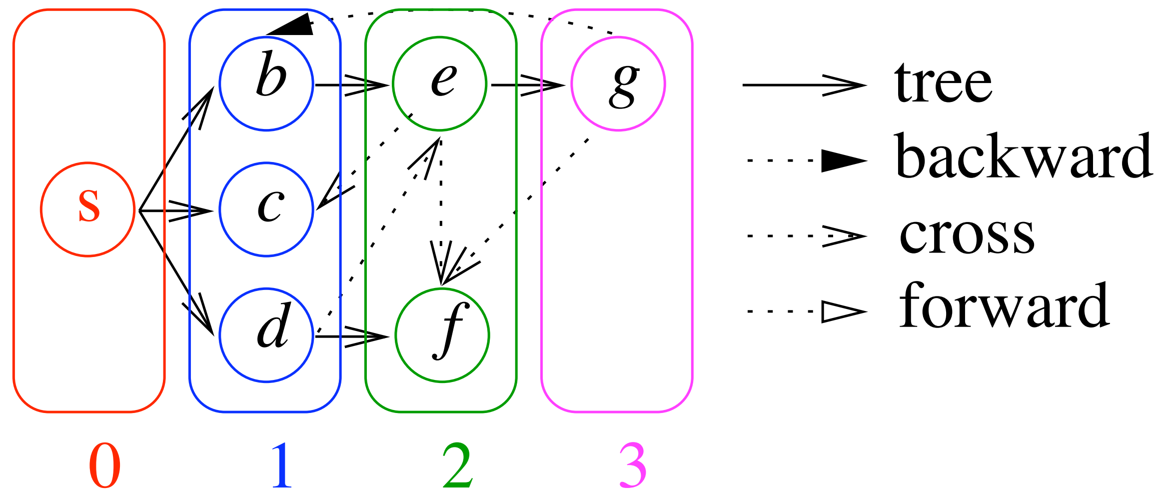
der alle von s erreichbaren Knoten

mit möglichst kurzen Pfaden erreicht. Berechne Abstände



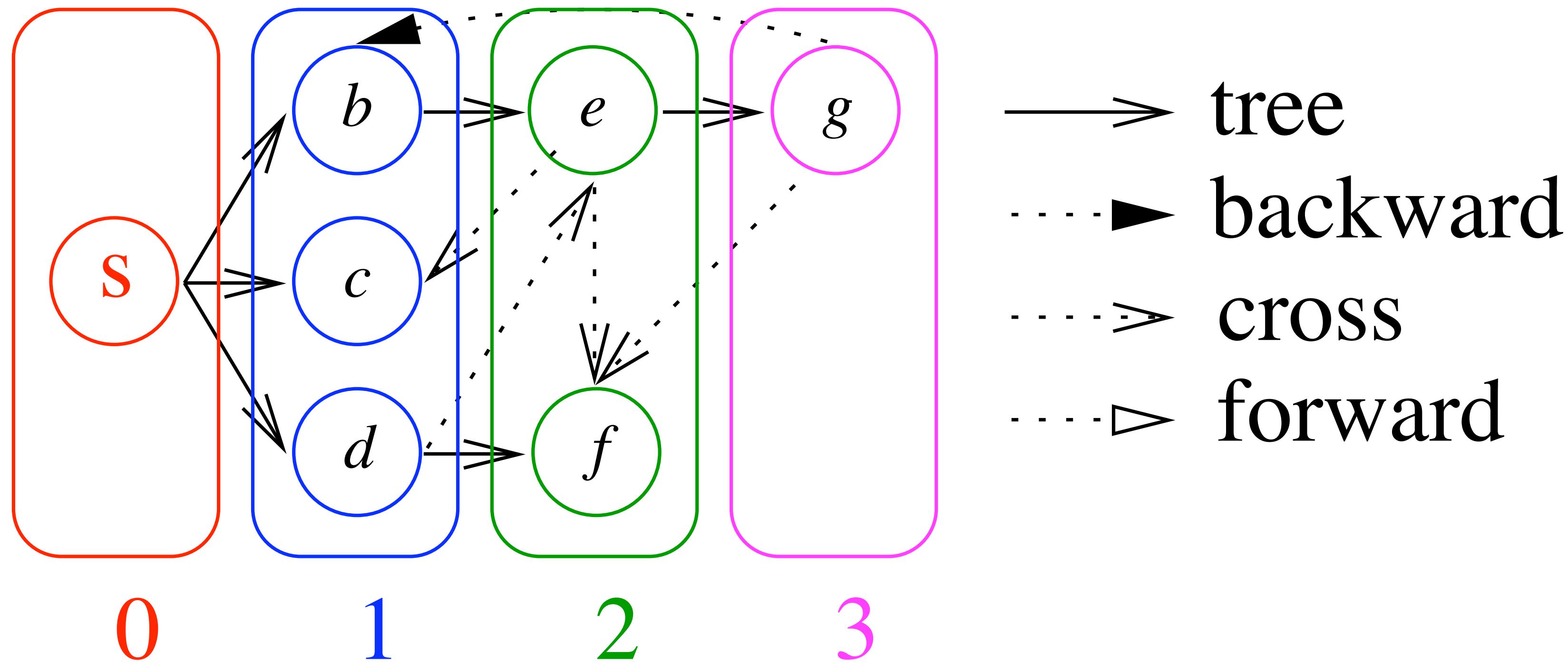
Breitensuche

- 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) :

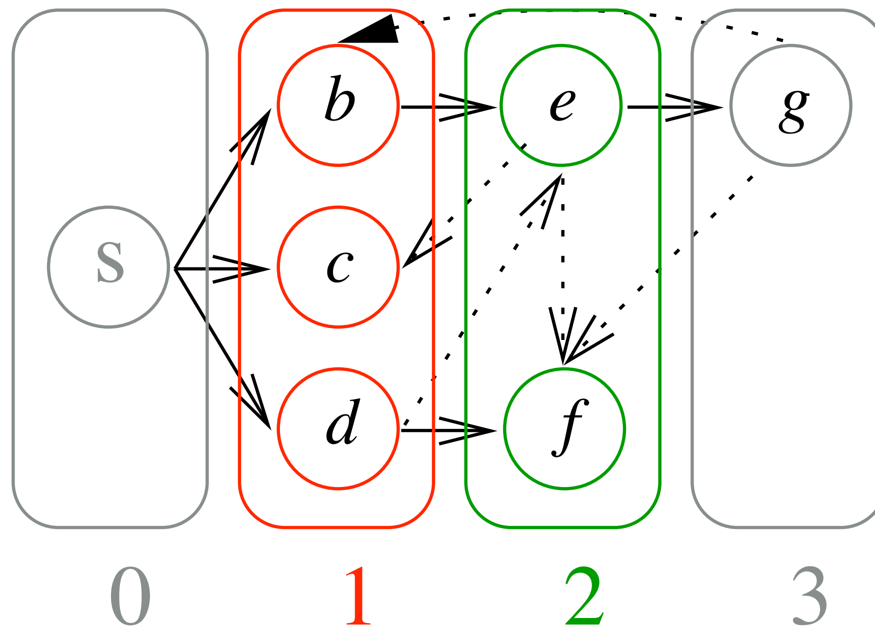
$Q := \langle s \rangle$ // aktuelle Schicht

while $Q \neq \langle\rangle$ **do**

exploriere Knoten in Q

merke dir Knoten der nächsten Schicht in Q'

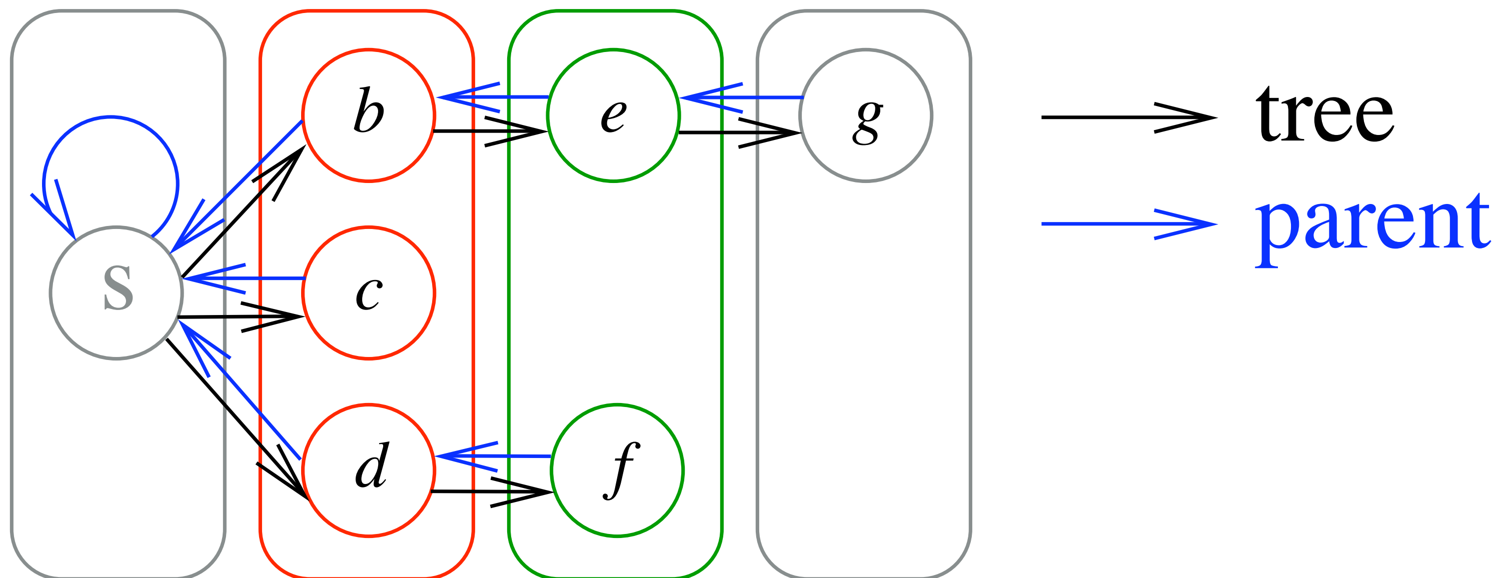
$Q := Q'$



Repräsentation des Baums

Feld parent speichert Vorgänger.

- noch nicht erreicht: parent$[v] = \bot$
- Startknoten/Wurzel: parent$[s] = s$



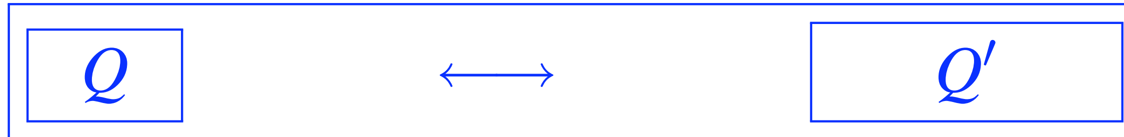
Function bfs(s : NodeId) : (NodeArray **of** NodeId) $\times$ (NodeArray **of** $\mathbb{N}_0 \cup \{\infty\}$)

    $d = \langle\infty,\ldots,\infty\rangle$: NodeArray **of** $\mathbb{N}_0 \cup \{\infty\}$; $d[s] := 0$
    parent$=\langle\bot,\ldots,\bot\rangle$: NodeArray **of** NodeId;   parent$[s] := s$
    $Q = \langle s\rangle, Q' = \langle\rangle$: Set **of** NodeId   // current, next layer
    for $\ell := 0$ **to** ∞ **while** $Q \neq \langle\rangle$ **do**
        invariant Q contains all nodes with distance ℓ from s
        foreach $u \in Q$ **do**
            foreach $(u,v) \in E$ **do**   // scan u
                if parent$(v) = \bot$ **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'

- Zwei Stapel
- Schleife $1\times$ ausrollen

  loop $Q \longrightarrow Q'; Q' \longrightarrow Q$
- Beide Stapel in ein Feld der Größe n

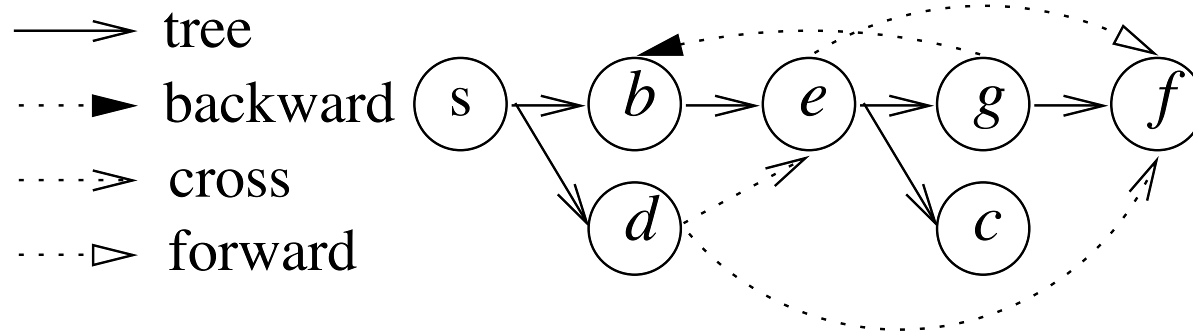


BFS mittels FIFO

$Q, Q' \longrightarrow$ einzelne FIFO Queue

- □ Standardimplementierung in anderen Büchern
- + „Oberflächlich“ einfacher
- − Korrektheit weniger evident — Übung?
- = Effizient (?) — Übung: ausprobieren?

9.2 Tiefensuche



Tiefensuchschema für $G = (V, E)$

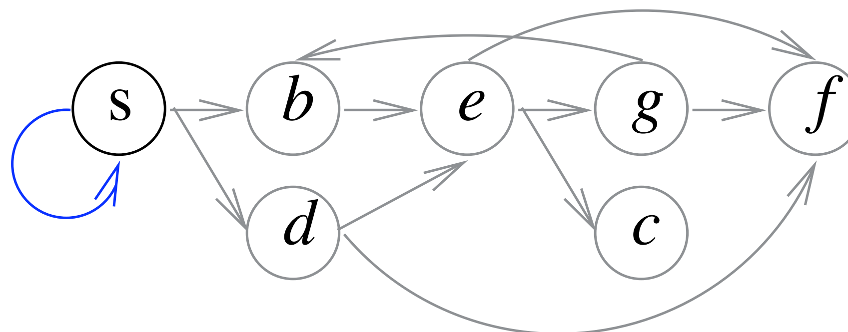
```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s                      // make s a root and grow
        root(s)                     // a new DFS-tree rooted at it.
        DFS(s, s)
Procedure DFS(u, v : NodeId)        // Explore v coming from u.
    foreach (v, w) ∈ 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:	parent$=\langle\perp,\ldots,\perp\rangle$: NodeArray **of** NodeId
root(s):	parent$[s]$:= s
traverseTreeEdge(v,w):	parent$[w]$:= v

→ tree

→ parent



mark s

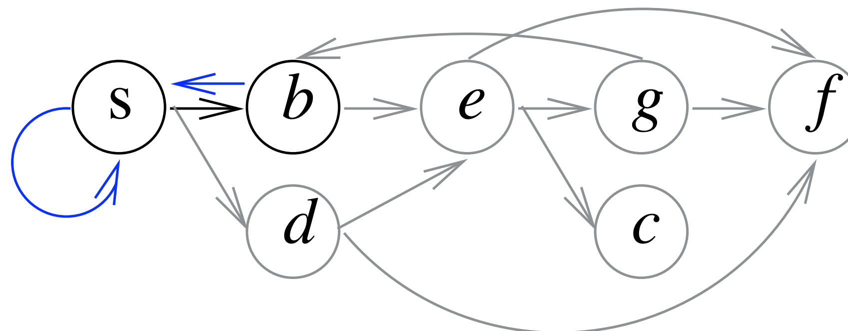
root(s)

dfs(s,s)

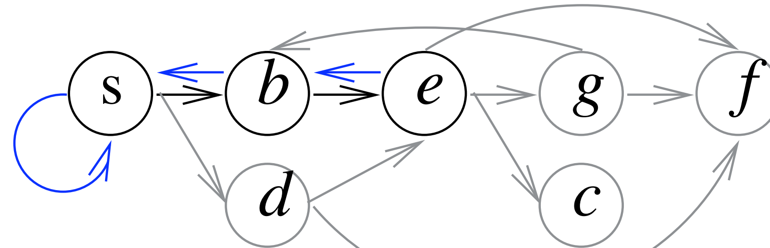
tranverseTreeEdge(s,b)

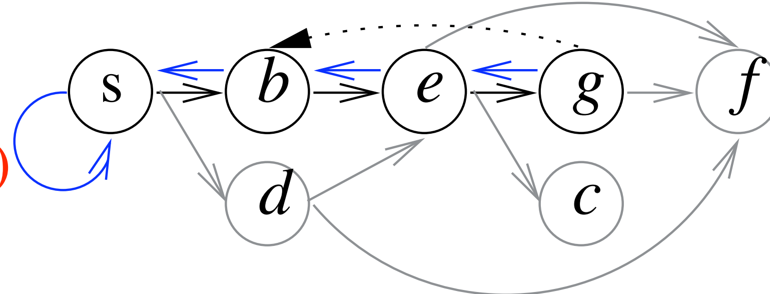
mark b

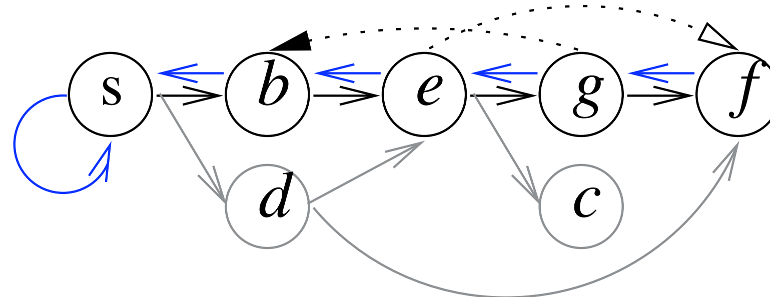
dfs(s,b)

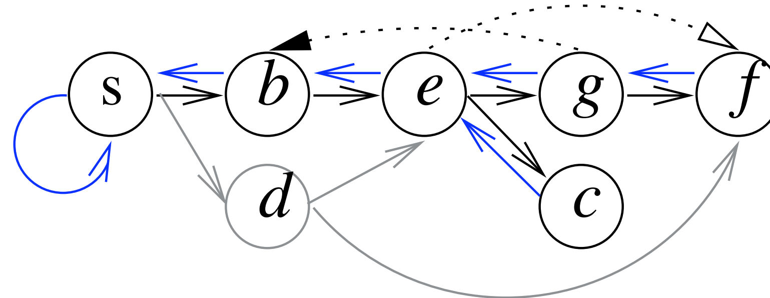


dfs(s,b)
  traverseTreeEdge(b,e)
  mark(e)
  dfs(b,e)
    traverseTreeEdge(e,g)
    mark(g)
    dfs(e,g)
      traverseNonTreeEdge(g,b)
      traverseTreeEdge(g,f)
      mark(f)
      dfs(g,f)
        backtrack(g,f)
      backtrack(e,g)
    traverseNonTreeEdge(e,f)
    traverseTreeEdge(e,c)
    mark(c)
    dfs(e,c)
      backtrack(e,c)
    backtrack(b,e)
  backtrack(s,b)

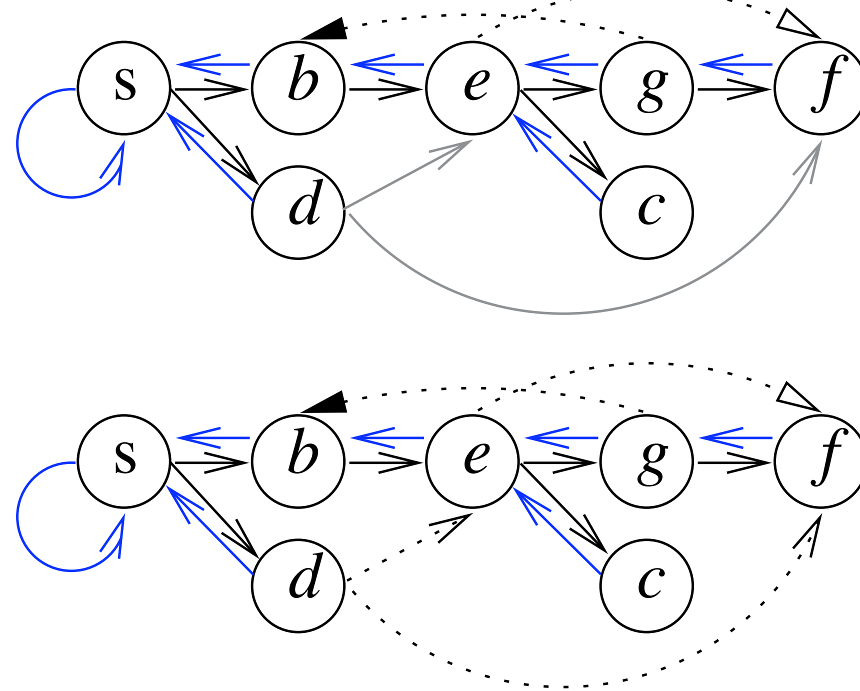








traverseTreeEdge(s,d)
mark(d)
dfs(s,d)
  traverseNonTreeEdge(d,e)
  traverseNonTreeEdge(d,f)
  backtrack(s,d)
backtrack(s,s)



DFS Nummerierung

init: dfsPos$=1$ $:1..n$

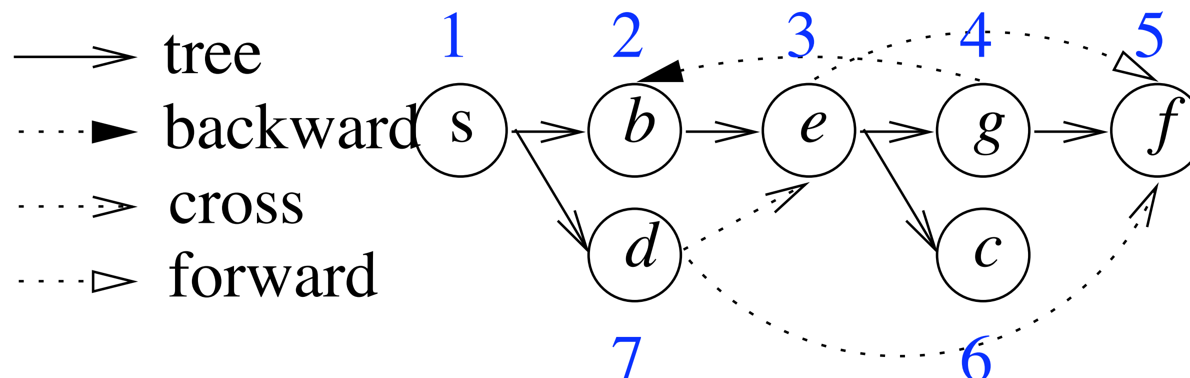
root(s): dfsNum$[s]$:= dfsPos$++$

traverseTreeEdge(v,w): dfsNum$[w]$:= dfsPos$++$

$u \prec v$ $:\Leftrightarrow$ dfsNum$[u] <$ dfsNum$[v]$.

Beobachtung:

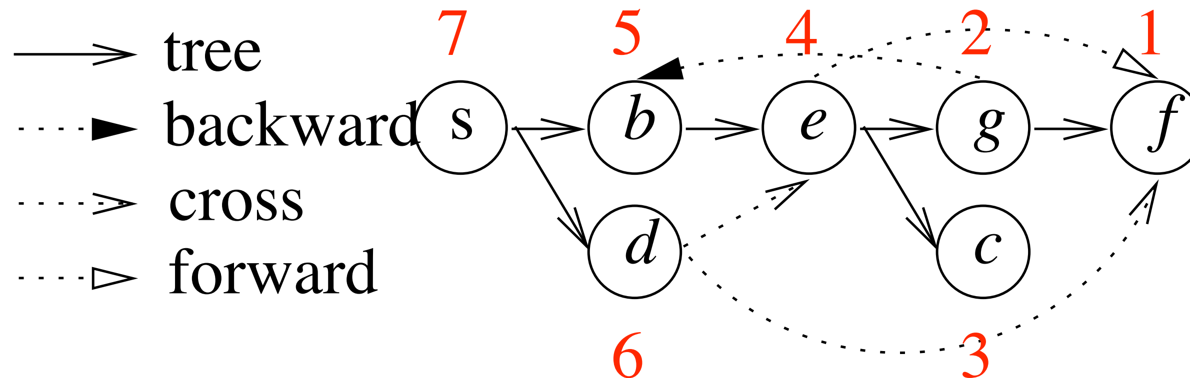
Knoten auf dem Rekursionsstapel sind bzgl., $\prec$ sortiert



Fertigstellungszeit

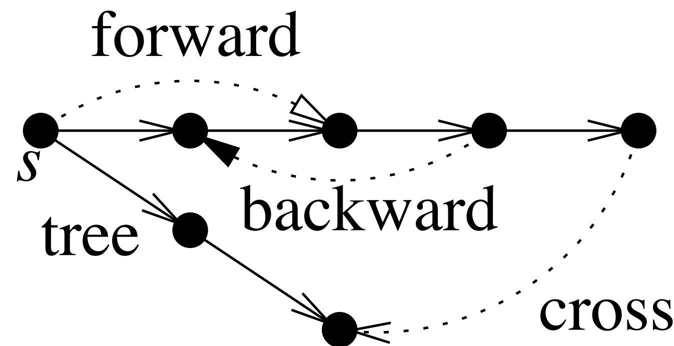
init: finishingTime$=1$: $1..n$

backtrack(u,v): finishTime$[v]$:= finishingTime$++$



Kantenklassifizierung bei DFS

type (v,w)	dfsNum$[v]<$ dfsNum$[w]$	finishTime$[w]<$ finishTime$[v]$	w is marked
tree	yes	yes	no
forward	yes	yes	yes
backward	no	no	yes
cross	no	yes	yes



Topologisches Sortieren mittels DFS

Satz:

G ist kreisfrei (DAG) $\Leftrightarrow$ DFS finded keine Rückwärtskante.

In diesem Fall liefert

$$t(v) := n - \text{finishTime}[v]$$

eine topologische Sortierung,

d. h.

$$\forall (u,v) \in E : t(u) < t(v)$$

Topologisches Sortieren mittels DFS

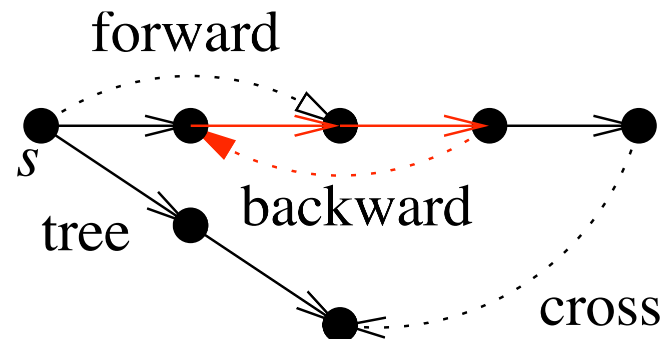
Satz: G kreisfrei (DAG) $\Leftrightarrow$ DFS finded keine Rückwärtskante.

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 "⇒": Annahme $\exists$ Rückwärtskante.

Zusammen mit Baumkanten ergibt sich ein Kreis.

Widerspruch.



Topologisches Sortieren mittels DFS

Satz: G kreisfrei (DAG) $\Leftrightarrow$ DFS finded 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

$$\overbrace{\Rightarrow}^{\text{Kantenklassifizierung}} \quad \forall (v,w) \in E : \text{finishTime}[v] > \text{finishTime}[w]$$

$\Rightarrow$ finishTime definiert umgekehrte topologische Sortierung.

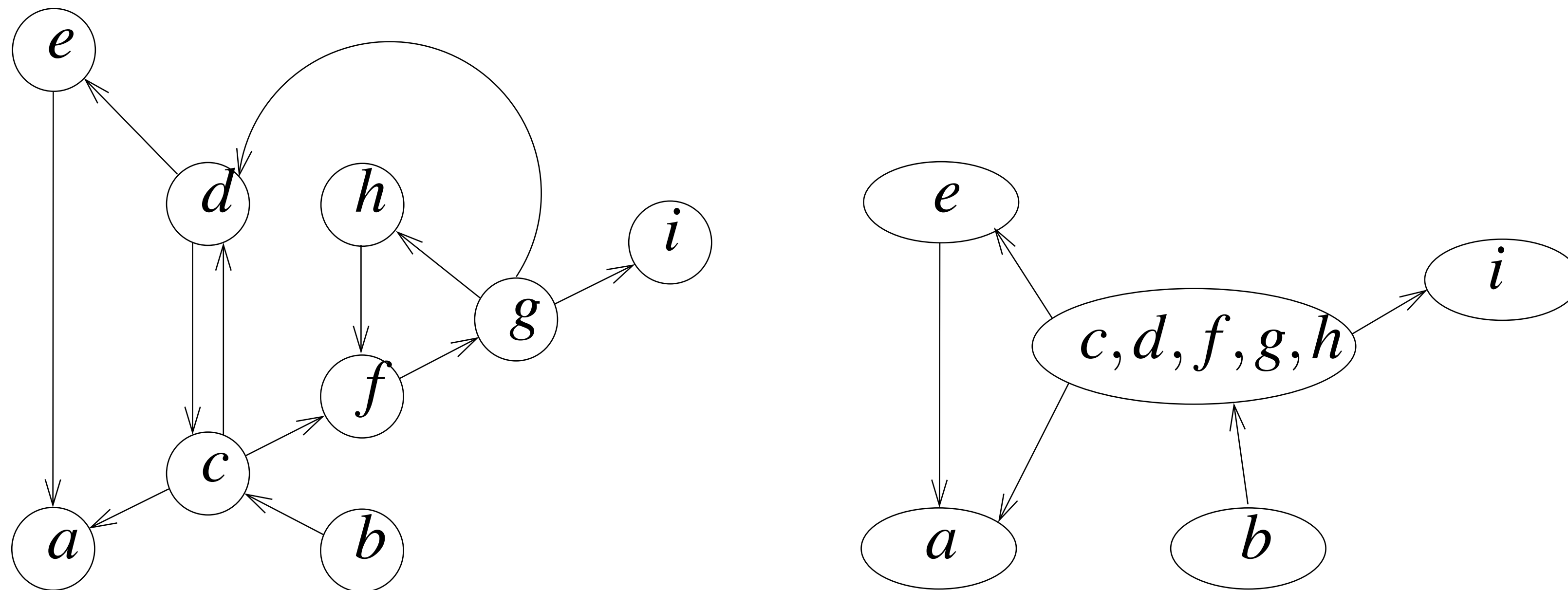
Starke Zusammenhangskomponenten

Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit

$u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle u, \ldots, v \rangle$ und $\exists$ Pfad $\langle v, \ldots, u \rangle$.

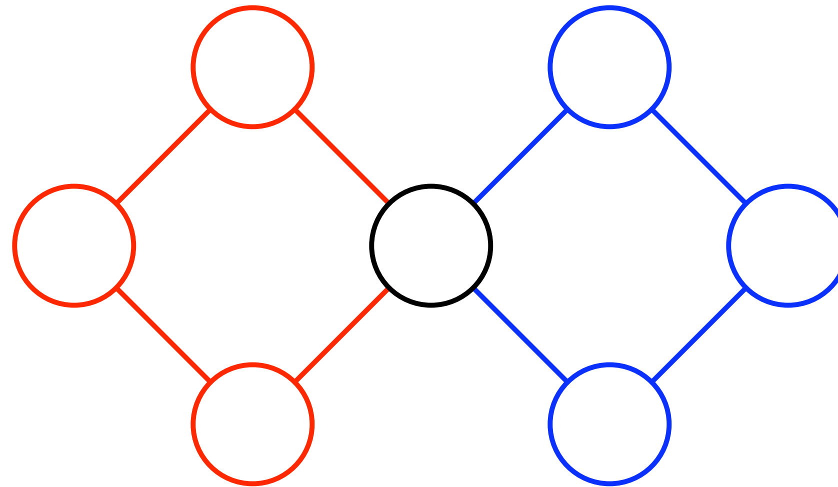
Beobachtung: $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation Übung

Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.



DFS-basierter Linearzeitalgorithmus $\longrightarrow$ Algorithmen II

Mehr DFS-basierte Linearzeitalgorithmen

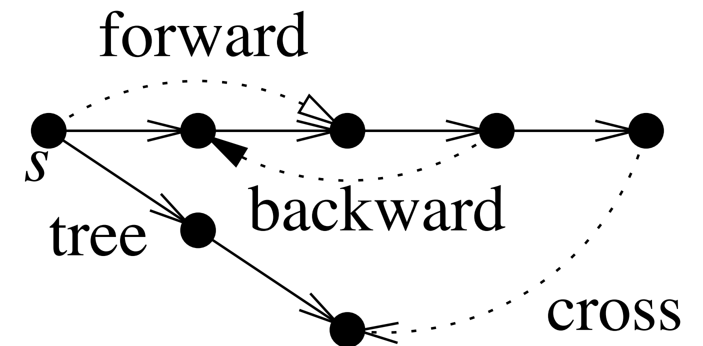


- 2-zusammenhängende Komponenten: bei Entfernen eines Knotens aus einer Komponente bleibt diese zusammenhängend (ungerichtet)
- 3-zusammenhängende Komponenten
- Planaritätstest (läßt sich der Graph kreuzungsfrei zeichnen?)
- Einbettung planarer Graphen

BFS ⟷ DFS

pro BFS:

- nichtrekursiv
- keine Vorwärtskanten
- kürzeste Wege, „Umgebung“



pro DFS

- keine explizite TODO-Datenstruktur (Rekursionsstapel)
- Grundlage vieler Algorithmen

10 Kürzeste Wege

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

Kostenfunktion/Kantengewicht $c : E \to \mathbb{R}$

Anfangsknoten s.

Ausgabe: für alle $v \in V$

Länge $\mu(v)$ des kürzesten Pfades von s nach v,

$\mu(v) := \min\{c(p) : p$ ist Pfad von s nach $v\}$

mit $c(\langle e_1,\ldots,e_k\rangle) := \sum_{i=1}^{k} c(e_i)$.

Oft wollen wir auch „geeignete“ Repräsentation der kürzesten Pfade.

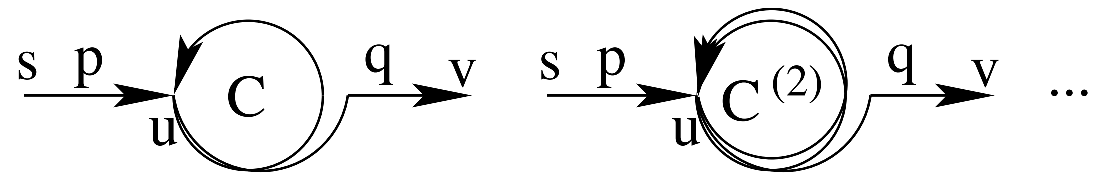
Anwendungen

- Routenplanung
  - Strassennetze
  - Spiele
  - Kommunikationsnetze
- Unterprogramm
  - Flüsse in Netzwerken
  - ...
- Tippfehlerkorrektur
- Disk Scheduling
- ...

10.1 Grundlagen

Gibt es immer einen kürzesten Pfad?

Es kann negative Kreise geben!



weitere Grundlagen just in time

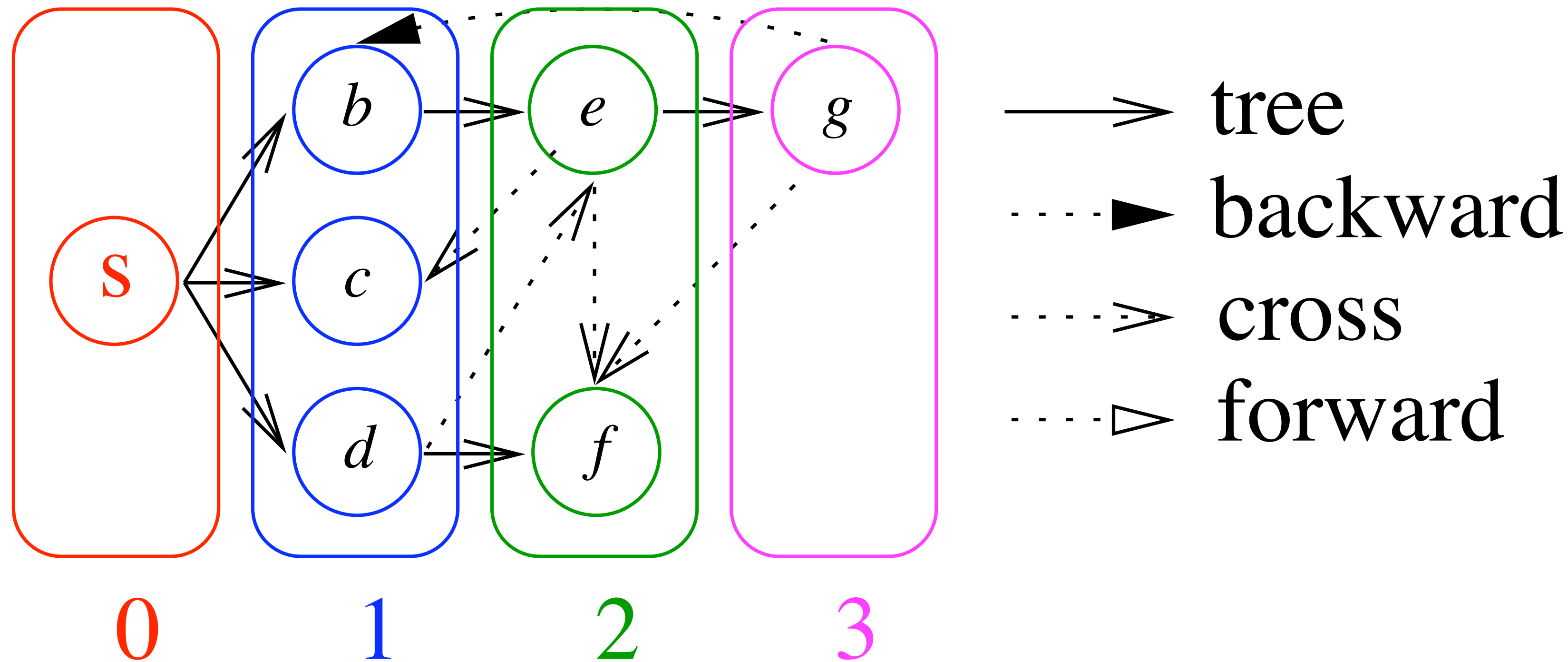
10.2 Azyklische Graphen

später

10.3 Kantengewichte ≥ 0

Alle Gewichte gleich:

Breitensuche (BFS)!



Dijkstra's Algorithmus

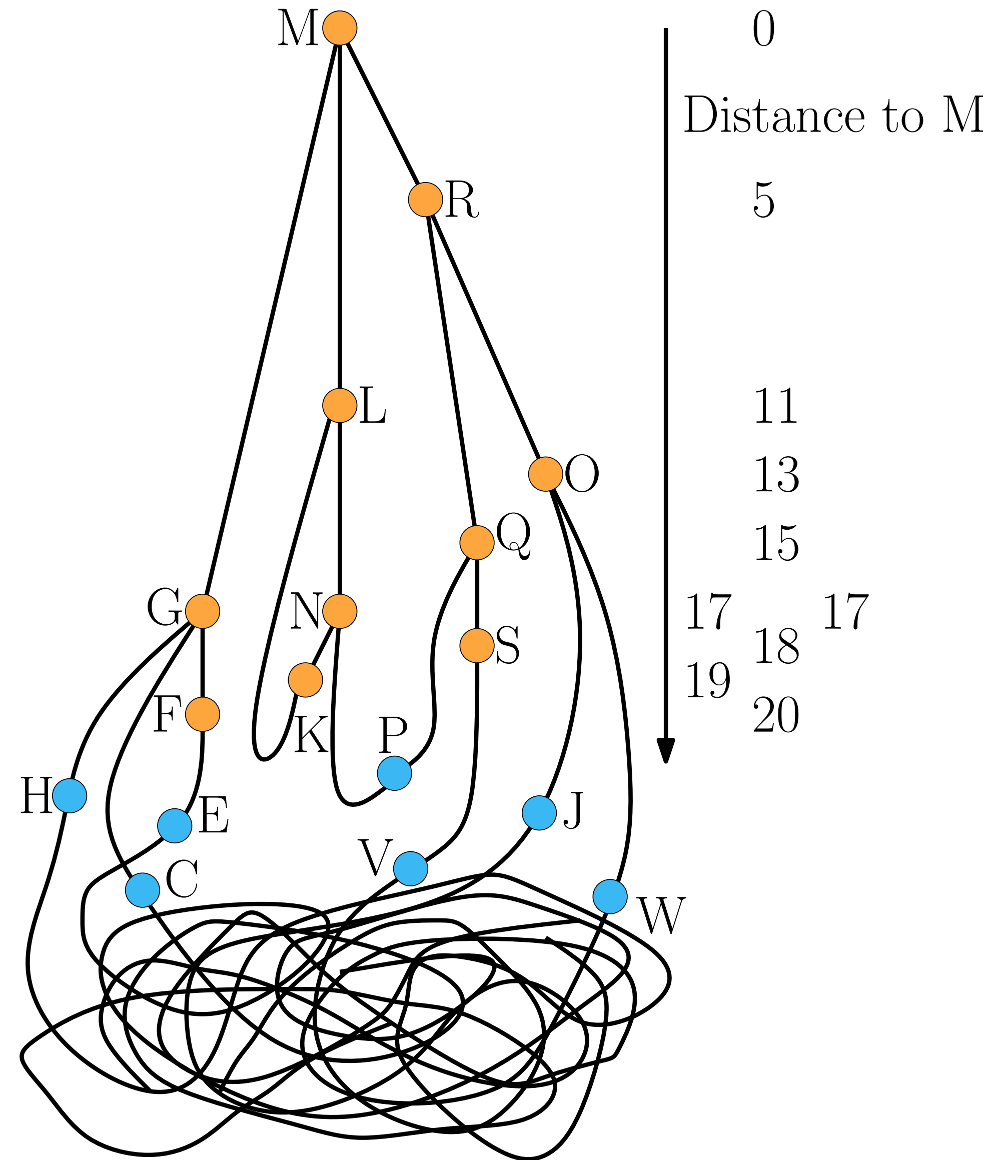
Allgemeine nichtnegative Gewichte

Lösung ohne Rechner:

Kanten $\longrightarrow$ Fäden

Knoten $\longrightarrow$ Knoten!

Am Startknoten anheben.



Korrekte Bindfäden

Betrachte beliebigen Knoten v

Mit Hängetiefe $d[v]$.

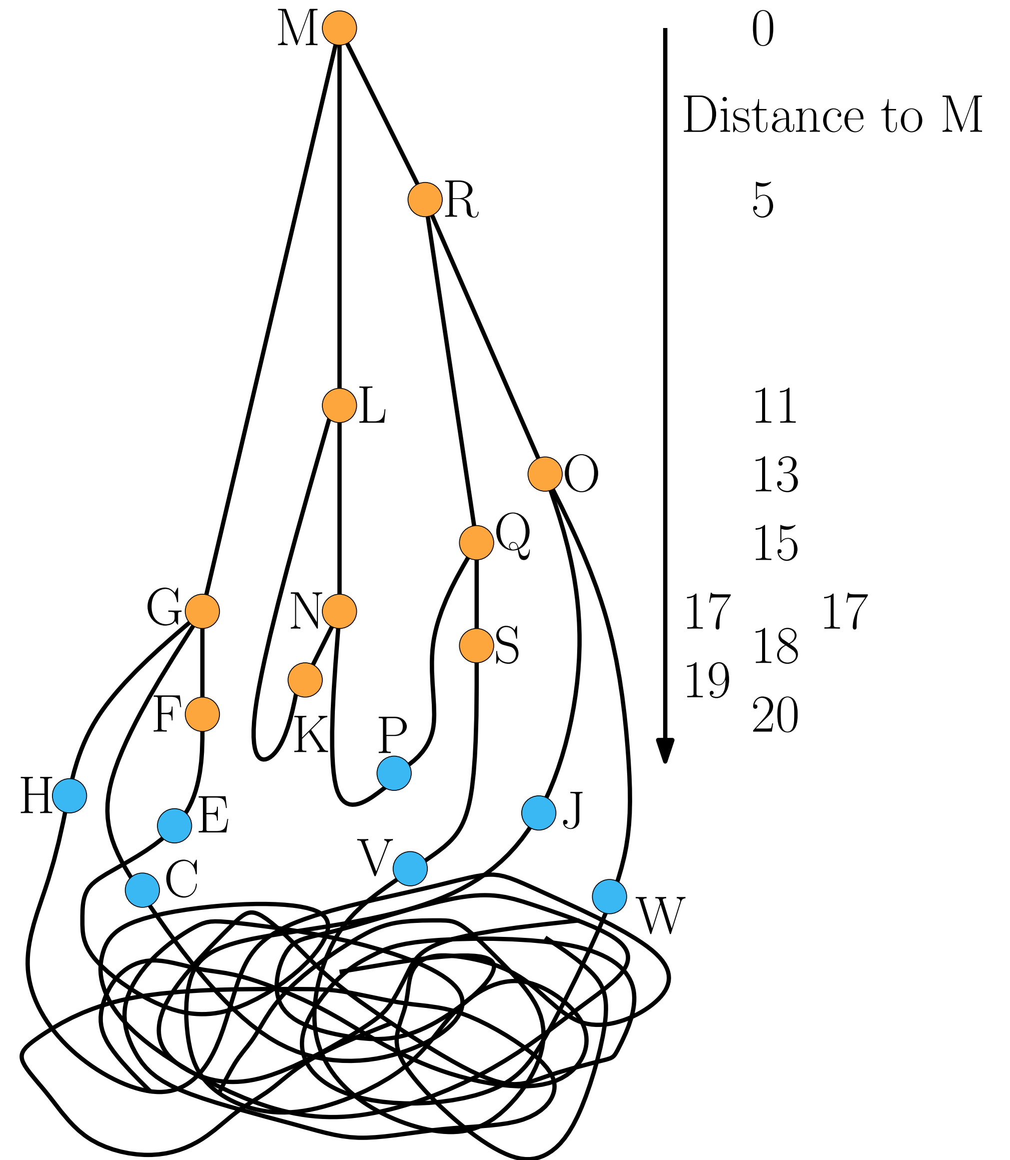
∃ Pfad mit Hängetiefe:

verfolge straffe Fäden

¬∃ kürzerer Pfad:

dann wäre einer seiner Fäden

zerrissen



Edsger Wybe Dijkstra

1972 ACM Turingpreis

THE: das erste Mulitasking-OS

Semaphor

Selbst-stabilisierende Systeme

GOTO Statement Considered Harmful

Allgemeine Definitionen

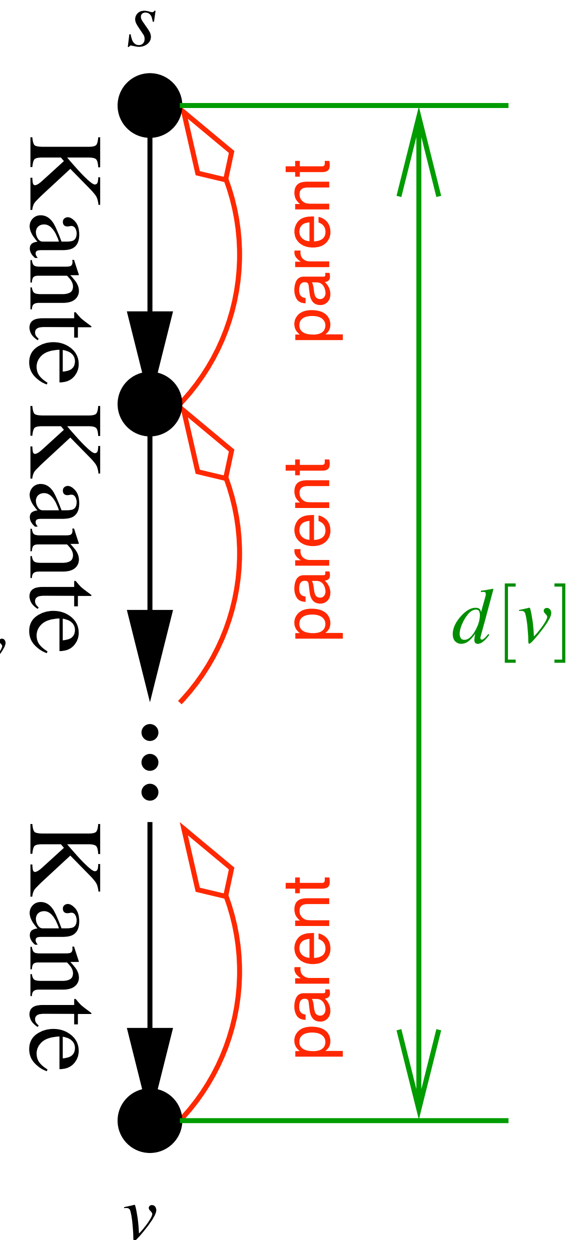
Wie bei BFS benutzen wir zwei Knotenarrays:

- $d[v]$ = aktuelle (vorläufige) Distanz von s nach v
  Invariante: $d[v] \geq \mu(v)$
- parent$[v]$ = Vorgänger von v
  auf dem (vorläufigen) kürzesten Pfad von s nach v
  Invariante: dieser Pfad bezeugt $d[v]$

Initialisierung:

$d[s] = 0$, parent[s] $= s$

$d[v] = \infty$, parent$[v] = \perp$



Kante (u, v) relaxieren

falls $d[u] + c(u, v) < d[v]$

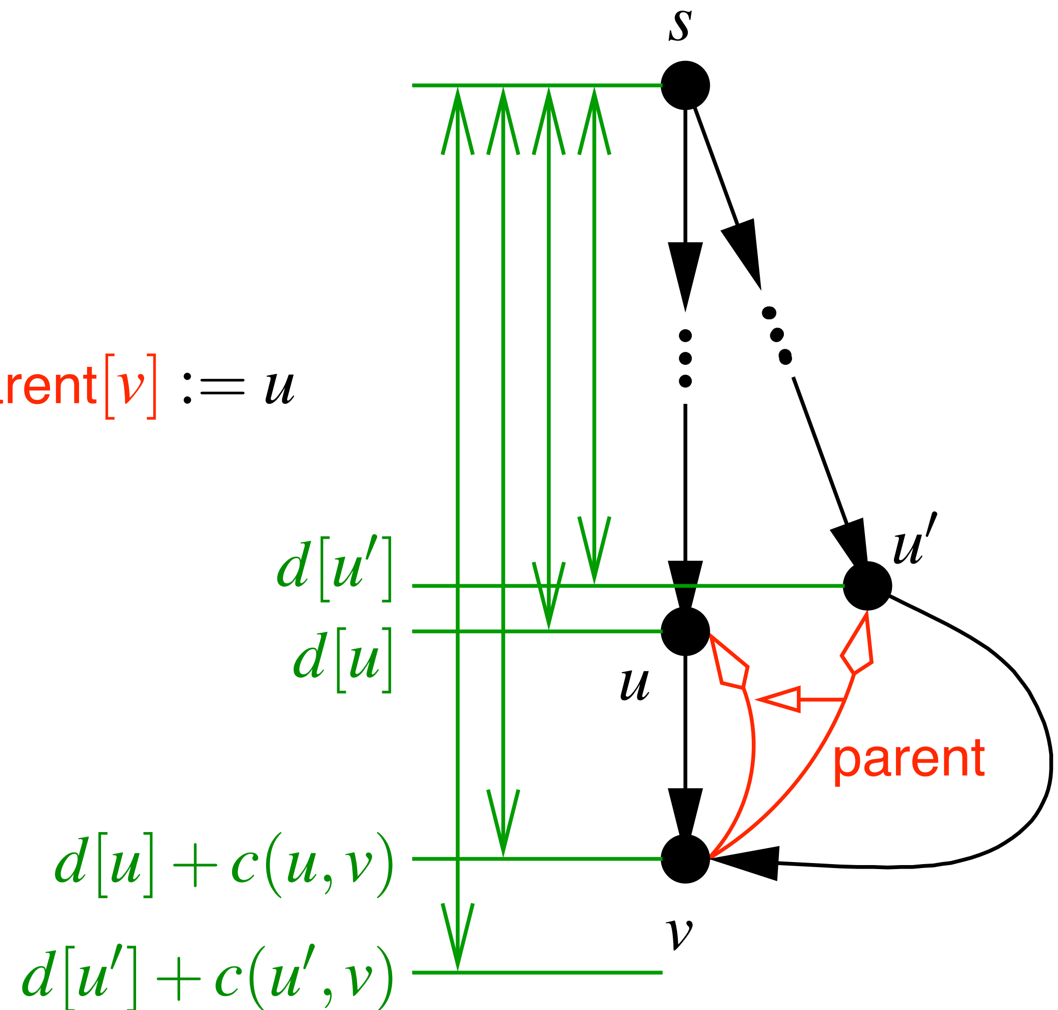
vielleicht $d[v] = \infty$

setze $d[v] := d[u] + c(u, v)$ und parent$[v] := u$

Invarianten bleiben erhalten!

Beobachtung:

$d[v]$ Kann sich mehrmals ändern!



Dijkstra's Algorithmus: Pseudocode

initialize d, parent

all nodes are non-scanned

while $\exists$ non-scanned node u with $d[u] < \infty$

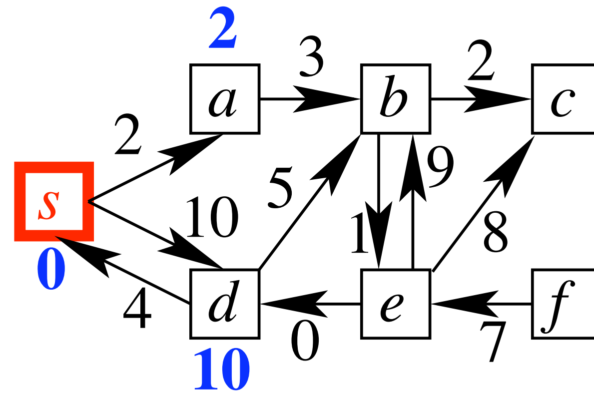
    $u :=$ non-scanned node v with minimal $d[v]$

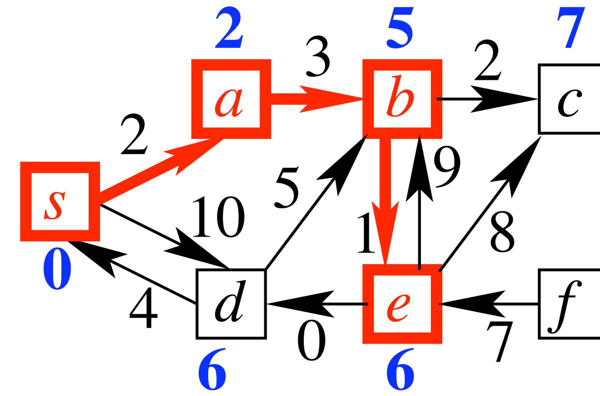
    relax all edges (u,v) out of u

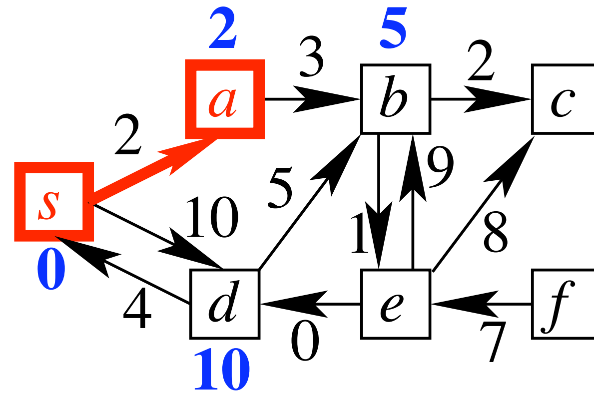
    u is scanned now

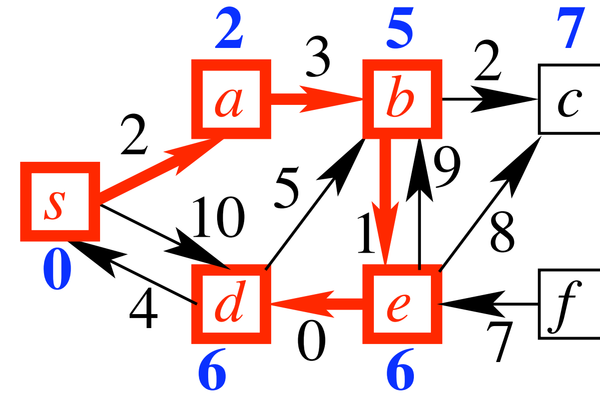
Behauptung: Am Ende definiert d die optimalen Entfernungen und parent die zugehörigen Wege

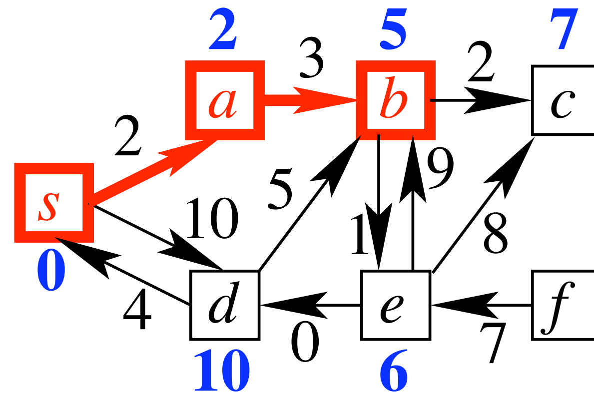
Beispiel

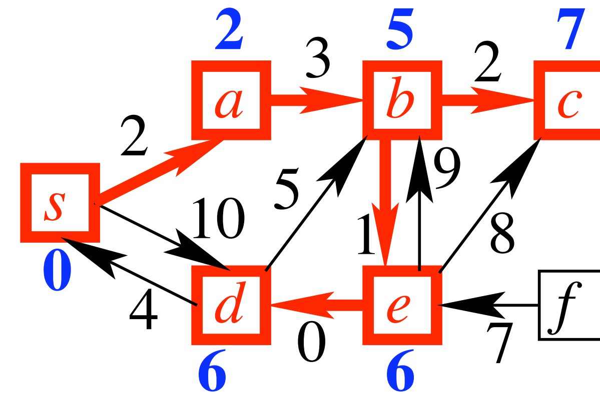












Korrektheit

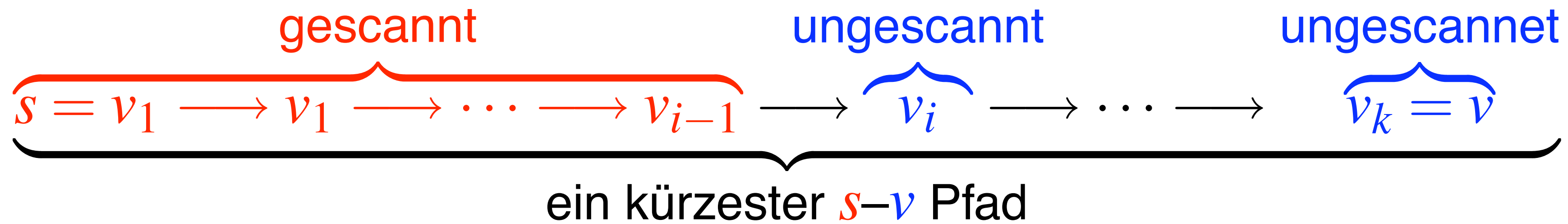
Annahme: alle Kosten nicht negativ!

Wir zeigen: $\forall v \in V$:

- v erreichbar $\Longrightarrow$ v wird irgendwann gescannt
- v gescannt $\Longrightarrow$ $\mu(v) = d[v]$

v erreichbar $\Longrightarrow$ v wird irgendwann gescannt

Annahme: v ist erreichbar aber wird nicht gescannt



$\Longrightarrow$ v_{i-1} wird gescannt

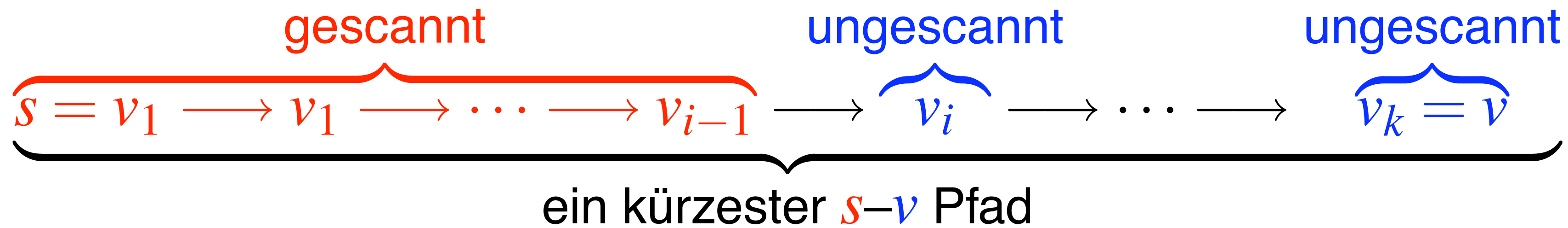
$\Longrightarrow$ Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert

$\Longrightarrow d[v_i] < \infty$

Widerspruch – nur Knoten x mit $d[x] = \infty$ werden nie gescannt $\square$?

v erreichbar $\Longrightarrow$ v wird irgendwann gescannt

Annahme: v ist erreichbar aber wird nicht gescannt



$\Longrightarrow$ v_{i-1} wird gescannt

$\Longrightarrow$ Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert

$\Longrightarrow d[v_i] < \infty$

Widerspruch – nur Knoten x mit $d[x] = \infty$ werden nie gescannt

Oops: Spezialfall $i = 1$?

Kann auch nicht sein.

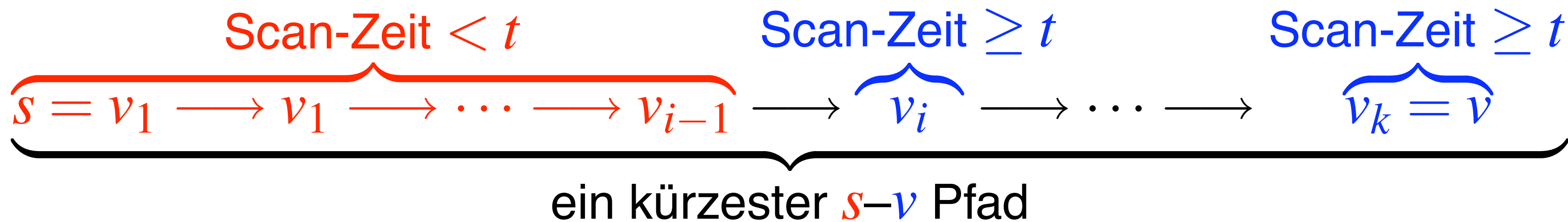
$v_1 = s$ wird bei Initialisierung gescannt. $\square$

v gescannt $\Longrightarrow \mu(v) = d[v]$

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

$\overbrace{\Longrightarrow} d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) = \mu(v_i) \leq \mu(v) < d[v]$

$\Longrightarrow v_i$ wird vor v gescannt. Widerspruch!

Wieder: Spezialfall $i = 1$ unmöglich. $\square$

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 addressierbarer Prioritätsliste Q.

Schlüssel ist $d[v]$.

Knoten speichern handles.

oder gleich items

Implementierung $\approx$ BFS mit PQ statt FIFO

Function Dijkstra(s : NodeId) : NodeArray$\times$NodeArray

// returns (d, parent)

Initialisierung:

$d = \langle\infty, \ldots, \infty\rangle$: NodeArray **of** $\mathbb{R} \cup \{\infty\}$

// tentative distance from root

parent$=\langle\bot, \ldots, \bot\rangle$: NodeArray **of** NodeId

parent$[s]$:= s // self-loop signals root

Q : NodePQ // unscanned reached nodes

$d[s]$:= 0; Q.insert(s)

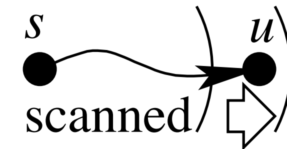
Function Dijkstra(s : NodeId) : NodeArray×NodeArray

$d = \langle\infty,\ldots,\infty\rangle$; parent$[s]$:= s; $d[s]$:= 0; Q.insert(s)

while $Q \neq \emptyset$ **do**

u := Q.deleteMin

// scan u



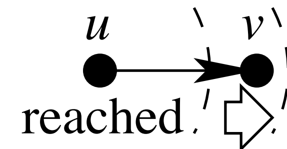
foreach edge $e = (u, v) \in E$ **do**

if $d[u] + c(e) < d[v]$ **then** // relax

$d[v]$:= $d[u] + c(e)$

parent$[v]$:= u // update tree

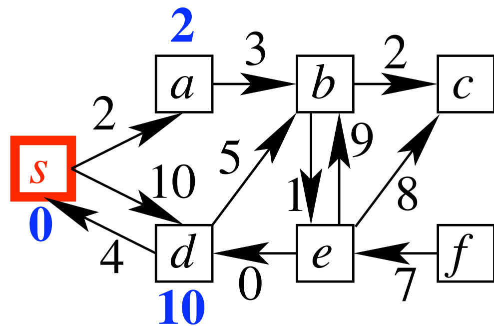
if $v \in Q$ **then** Q.decreaseKey(v)

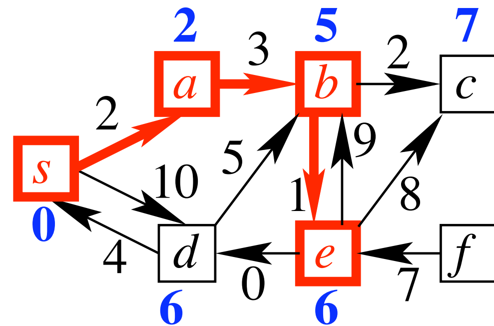


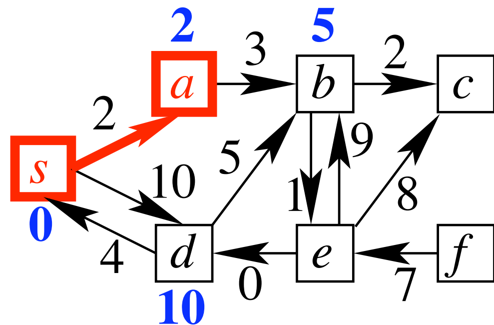
else Q.insert(v)

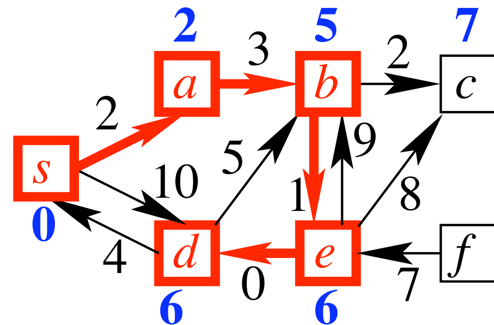
return $(d,$parent$)$

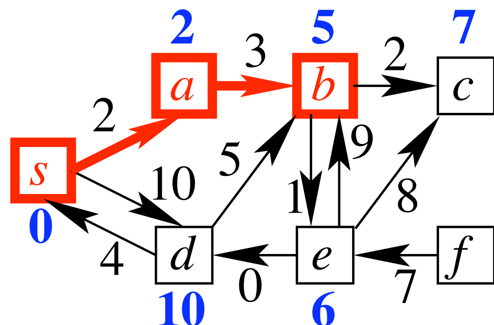
Beispiel

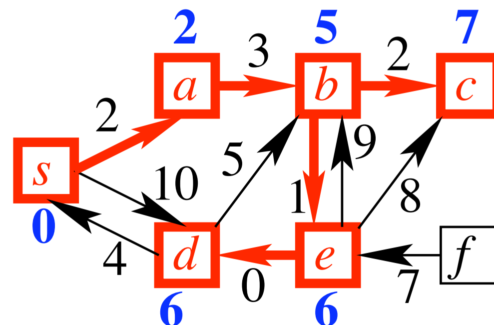












Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow (s,0)$	$\langle\rangle$
relax $s \stackrel{2}{\to} a$	$\langle(a,2)\rangle$
relax $s \stackrel{10}{\to} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow (a,2)$	$\langle(d,10)\rangle$
relax $a \stackrel{3}{\to} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow (b,5)$	$\langle(d,10)\rangle$
relax $b \stackrel{2}{\to} c$	$\langle(c,7),(d,10)\rangle$
relax $b \stackrel{1}{\to} e$	$\langle(e,6),(c,7),(d,10)\rangle$
deleteMin$\rightsquigarrow (e,6)$	$\langle(c,7),(d,10)\rangle$
relax $e \stackrel{9}{\to} b$	$\langle(c,7),(d,10)\rangle$
relax $e \stackrel{8}{\to} c$	$\langle(c,7),(d,10)\rangle$
relax $e \stackrel{0}{\to} d$	$\langle(d,6),(c,7)\rangle$
deleteMin$\rightsquigarrow (d,6)$	$\langle(c,7)\rangle$
relax $d \stackrel{4}{\to} s$	$\langle(c,7)\rangle$
relax $d \stackrel{5}{\to} b$	$\langle(c,7)\rangle$
deleteMin$\rightsquigarrow (c,7)$	$\langle\rangle$

Laufzeit

Function Dijkstra(s : NodeId) : NodeArray$\times$NodeArray

Initialisierung:

$d = \langle\infty,\ldots,\infty\rangle$: NodeArray **of** $\mathbb{R}\cup\{\infty\}$ // $O(n)$

parent$=\langle\perp,\ldots,\perp\rangle$: NodeArray **of** NodeId // $O(n)$

parent$[s]$:= s

Q : NodePQ // unscanned reached nodes, $O(n)$

$d[s]$:= 0; Q.insert(s)

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = {∞,...,∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)    // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                      // ≤ n×
        foreach edge e = (u,v) ∈ E do                         // ≤ m×
            if d[u] + c(e) < d[v] then                        // ≤ m×
                d[v]:= d[u] + c(e)                            // ≤ m×
                parent[v] := u                                // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                // ≤ m×
                else Q.insert(v)                              // ≤ n×
    return (d, parent)
```

Function Dijkstra(s : NodeId) : NodeArray×NodeArray

```
    d = {∞,...,∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)     // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                       // ≤ n×
        foreach edge e = (u,v) ∈ E do                          // ≤ m×
            if d[u] + c(e) < d[v] then                         // ≤ m×
                d[v]:= d[u] + c(e)                             // ≤ m×
                parent[v] := u                                 // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                 // ≤ m×
                else Q.insert(v)                               // ≤ n×
    return (d, parent)
```

Insgesamt

$$T_{\text{Dijkstra}} = \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

Laufzeit

Dijkstra's ursprüngliche Implementierung: „naiv“

- insert $\mathrm{O}(1)$ — $d[v] := d[u] + c(u,v)$
- decreaseKey $\mathrm{O}(1)$ — $d[v] := d[u] + c(u,v)$
- deleteMin $\mathrm{O}(n)$ — d komplett durchsuchen

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{Dijkstra59}} &= \mathrm{O}(m \cdot 1 + n \cdot (n+1)) \\
&= \mathrm{O}\left(m + n^2\right)
\end{aligned}
$$

Laufzeit

Bessere Implementierung mit Binary-Heapprioritätslisten:

- insert $O(\log n)$
- decreaseKey $O(\log n)$
- deleteMin $O(\log n)$

$$
\begin{aligned}
T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{DijkstraBHeap}} &= O(m \cdot \log n + n \cdot (\log n + 1)) \\
&= O((m+n)\log n)
\end{aligned}
$$

Laufzeit

(Noch) besser mit Fibonacci-Heapprioritätslisten:

- insert $\mathrm{O}(1)$
- decreaseKey $\mathrm{O}(1)$ (amortisiert)
- deleteMin $\mathrm{O}(\log n)$ (amortisiert)

$$
\begin{aligned}
T_{\mathrm{Dijkstra}} &= \mathrm{O}(m \cdot T_{\mathsf{decreaseKey}}(n) + n \cdot (T_{\mathsf{deleteMin}}(n) + T_{\mathsf{insert}}(n))) \\
T_{\mathsf{DijkstraFib}} &= \mathrm{O}(m \cdot 1 + n \cdot (\log n + 1)) \\
&= \mathrm{O}(m + n \log n)
\end{aligned}
$$

Aber: konstante Faktoren in $\mathrm{O}(\cdot)$ sind hier größer!

10.4 Analyse im Mittel

Modell: Kantengewichte sind „zufällig“ auf die Kanten verteilt

Dann gilt

$$\mathrm{E}[T_{\text{DijkstraBHeap}}] = \mathrm{O}\left(m + n \log n \log \frac{m}{n}\right)$$

Beweis: In Algorithmen II

10.5 Monotone ganzzahlige Prioritätslisten

Beobachtung: In Dijkstra's Algorithmus steigt das Minimum in der Prioritätsliste monoton.

Das kann man ausnutzen. $\leadsto$ schnellere Algorithmen

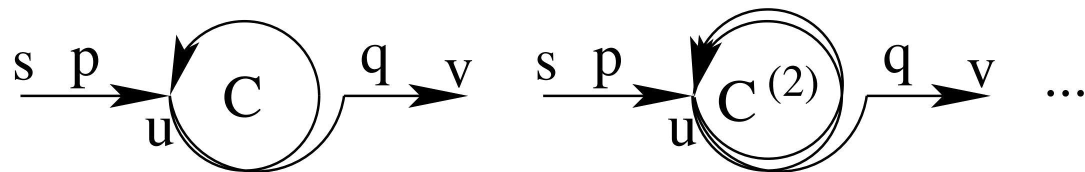
u.U. bis herunter zu $O(m+n)$.

Details: in Algorithmen II

10.6 Negative Kosten

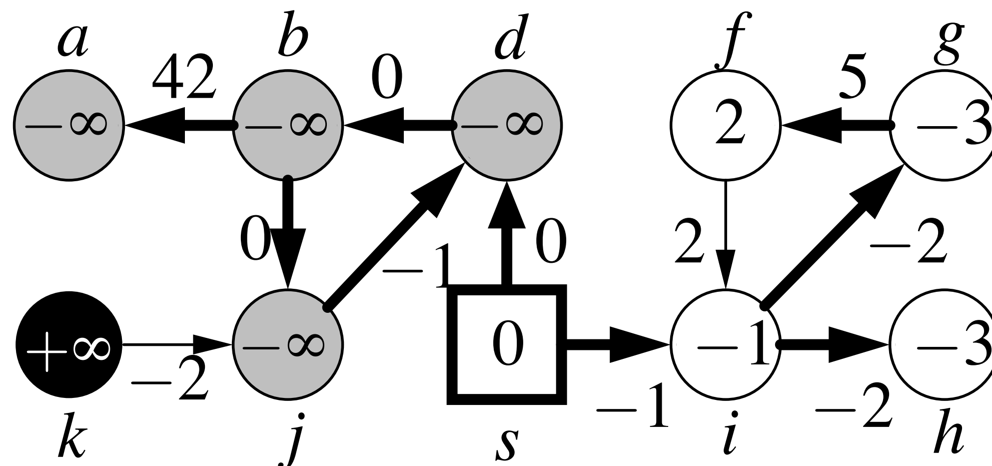
Was machen wir wenn es Kanten mit negativen Kosten gibt?

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: $\exists$ kürzesten s–v-Pfad $P \Longrightarrow P$ ist OBdA einfach (eng. simple)

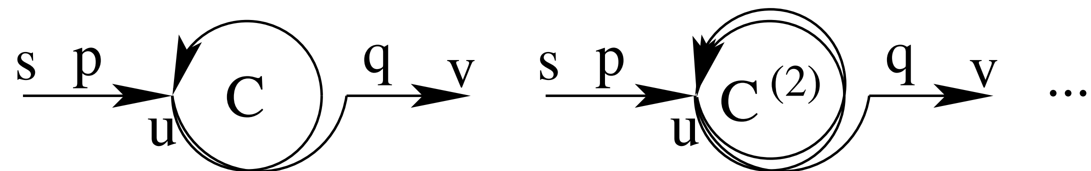
Beweisidee:

Fall: v über negativen Kreis erreichbar?**:**

$\neg\exists$ kürzesten s–v-Pfad

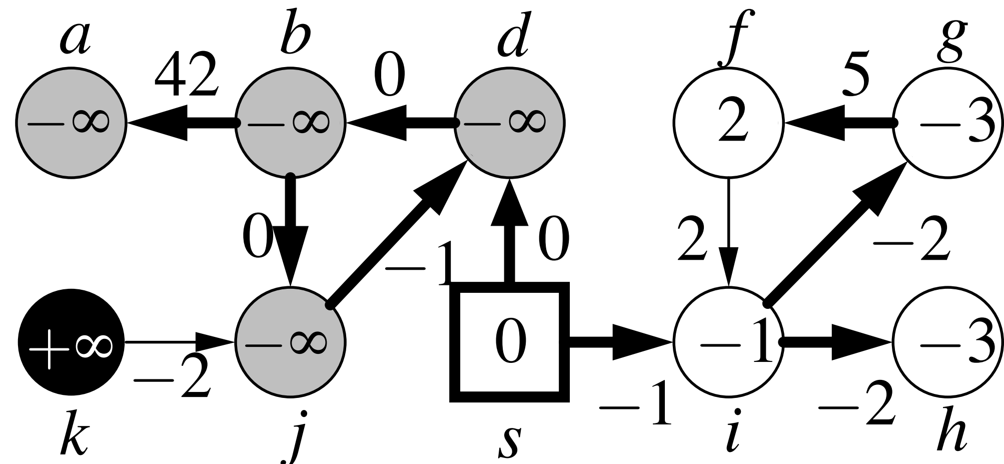
(sondern beliebig

viele immer kürzere)



Sonst: betrachte beliebigen nicht-einfachen s–v-Pfad.

Alle Kreise streichen $\leadsto$ einfacher, nicht längerer Pfad. $\blacksquare$



Mehr Basiskonzepte

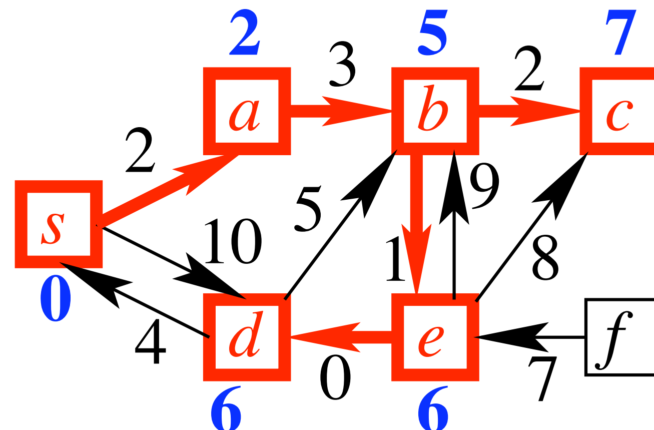
Übung, Zeige:

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 kürzeste Pfade von s aus zusammen bilden einen Baum falls es keine negativen Kreise gibt.



Allgemeines Korrektheitskriterium

Sei $R = \langle \cdots \overbrace{\text{relax}(e_1)}^{t_1} \cdots \overbrace{\text{relax}(e_2)}^{t_2} \cdots \overbrace{\text{relax}(e_k)}^{t_k} \cdots \rangle$

eine Folge von Relaxierungsoperationen und

$p = \langle e_1, e_2, \ldots, e_k \rangle = \langle s, v_1, v_2, \ldots, v_k \rangle$ ein kürzester Weg.

Dann gilt anschließend $d[v_k] = \mu(v_k)$

Beweisskizze: (Eigentlich Induktion über k)

$$d[s] = \mu(s) \text{ bei Initialisierung}$$

$$d[v_1] = \mu(v_1) \text{ nach Zeitpunkt } t_1$$

$$d[v_2] = \mu(v_2) \text{ nach Zeitpunkt } t_2$$

$$\ldots$$

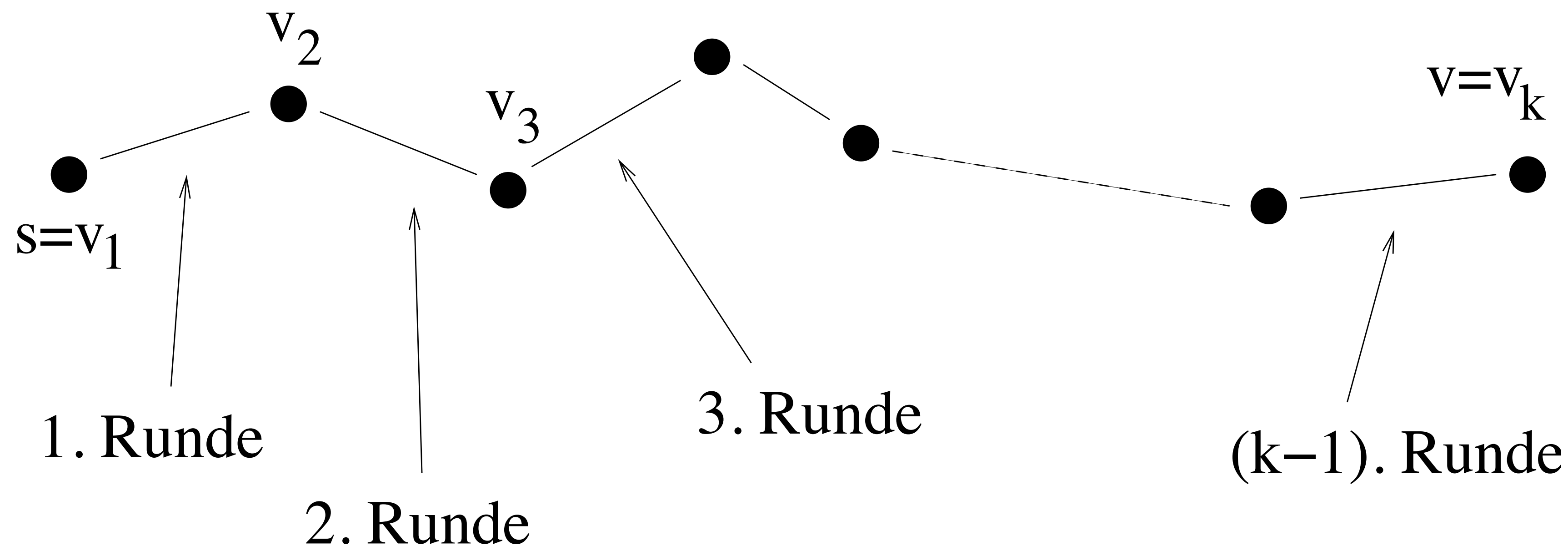
$$d[v_k] = \mu(v_k) \text{ nach Zeitpunkt } t_k \quad \blacksquare$$

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 kürzeste Pfad ist eine Teilfolge dieser Relaxierungen!



Negative Kreise Finden

Nach Ausführung von Bellman-Ford:

$\forall$ negativen Kreise C:

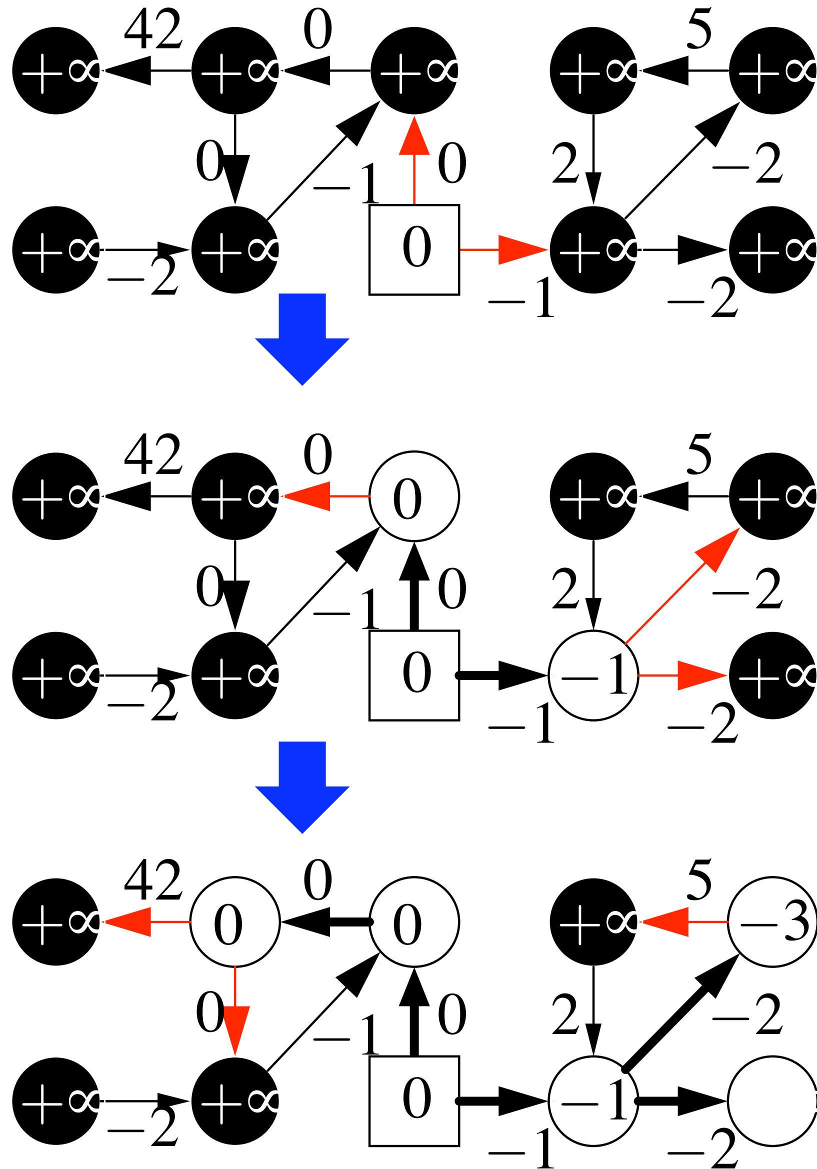
$\exists(u,v) \in C :$

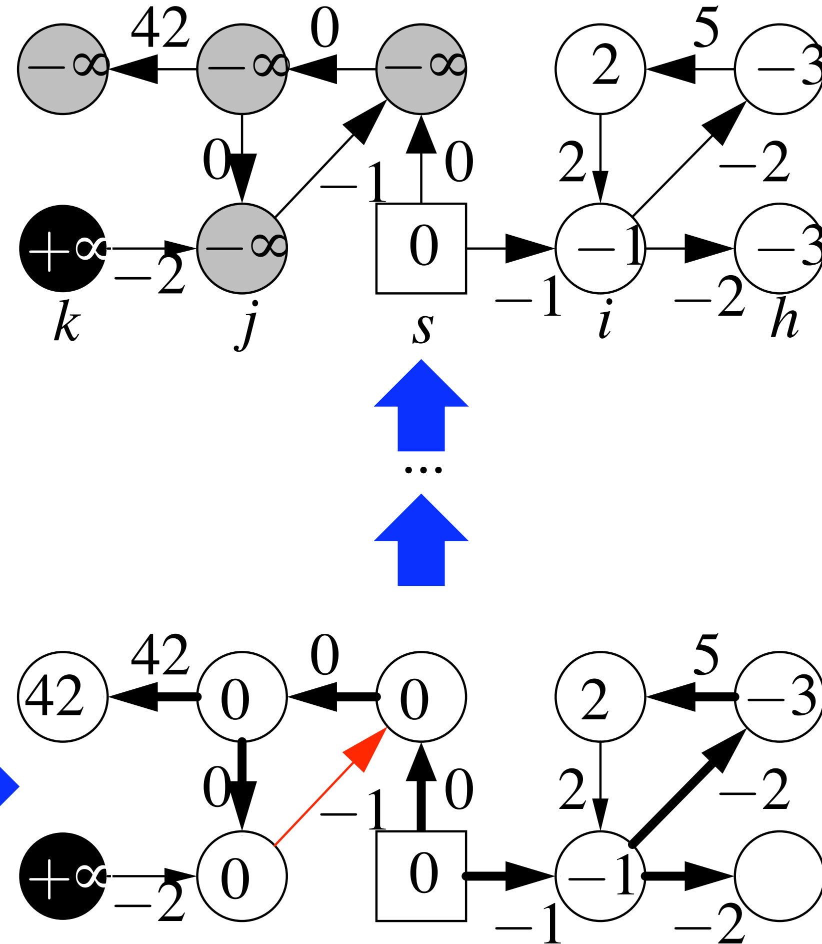
$d[u] + c(e) < d[v]$

Beweis: Übung

v und alle von v erreichbaren Knoten x haben $\mu(x) = -\infty$

Beispiel





Bellman-Ford – Laufzeit

$O(nm)$ viel langsamer als Dijkstra !

Es gibt Algorithmenvarianten mit viel besserem best case.

Azyklische Graphen (10.2 im Buch)

Beobachtungen:

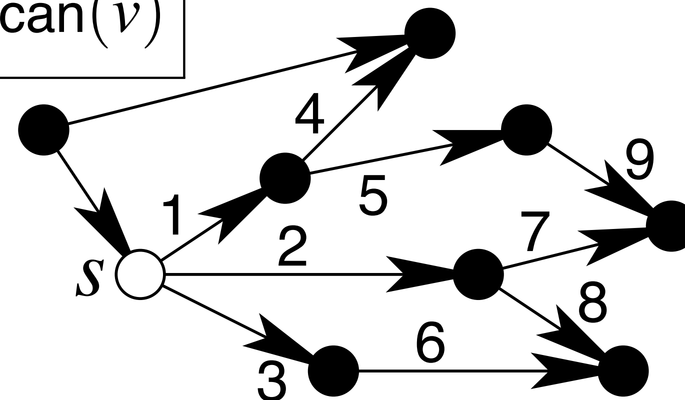
Keine (gerichteten) Kreise $\Longrightarrow$ keine negativen Kreise !

Für jeden (kürzesten) Pfad $\langle v_1, \ldots, v_n \rangle$:

Die Knoten sind aufsteigend bzgl. jeder topologischen Sortierung !

```
initialize d, parent
foreach v ∈ V in topological order do scan(v)
```

Laufzeit: $O(m+n)$



10.7 Von Überall nach Überall

Im Prinzip: $n\times$ von s nach Überall

nichtnegative Kantengewichte: Zeit $\mathrm{O}(n(m+n\log n))$.

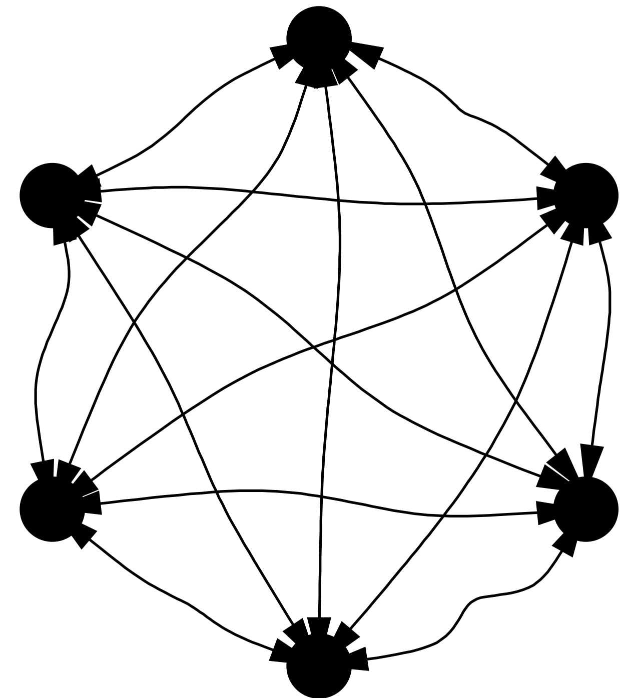
($n\times$ Dijkstra)

beliebige Kantengewichte: Zeit $\mathrm{O}\left(n^2 m\right)$.

($n\times$ Bellman-Ford)

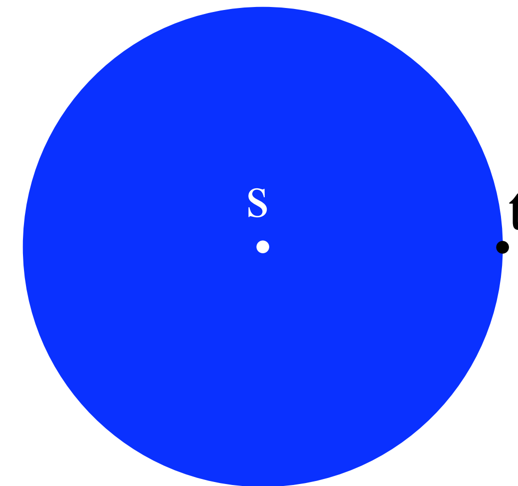
In Algorithmen II: Zeit $\mathrm{O}(n(m+n\log n))$.

($1\times$ Bellman-Ford $+$ $n\times$ Dijkstra)



10.8 Distanz zu einem Zielknoten t

Was machen wir, wenn wir nur die Distanz von s zu einem bestimmten Knoten t wissen wollen?



Trick 0:

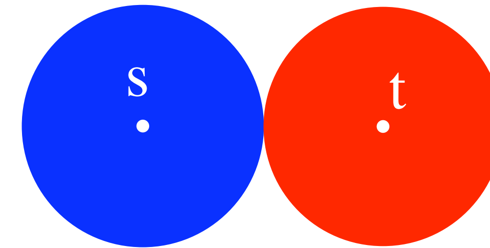
Dijkstra hört auf, wenn t aus Q entfernt wird

Spart "im Durchschnitt" Hälfte der Scans

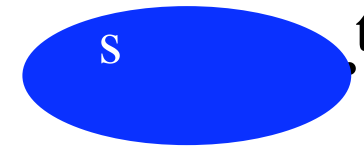
Frage: Wieviel spart es (meist) beim Europa-Navi?

Ideen für Routenplanung

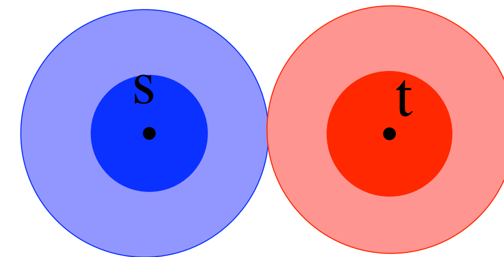
mehr in Algorithmen II, Algorithm Engineering



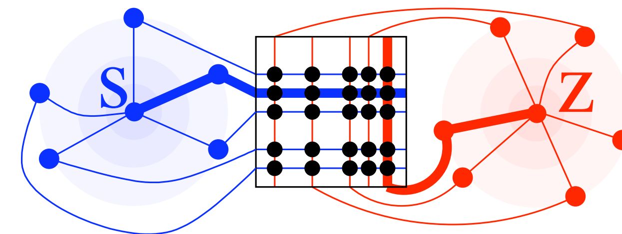
- Vorwärts + Rückwärtssuche



- Zielgerichtete Suche

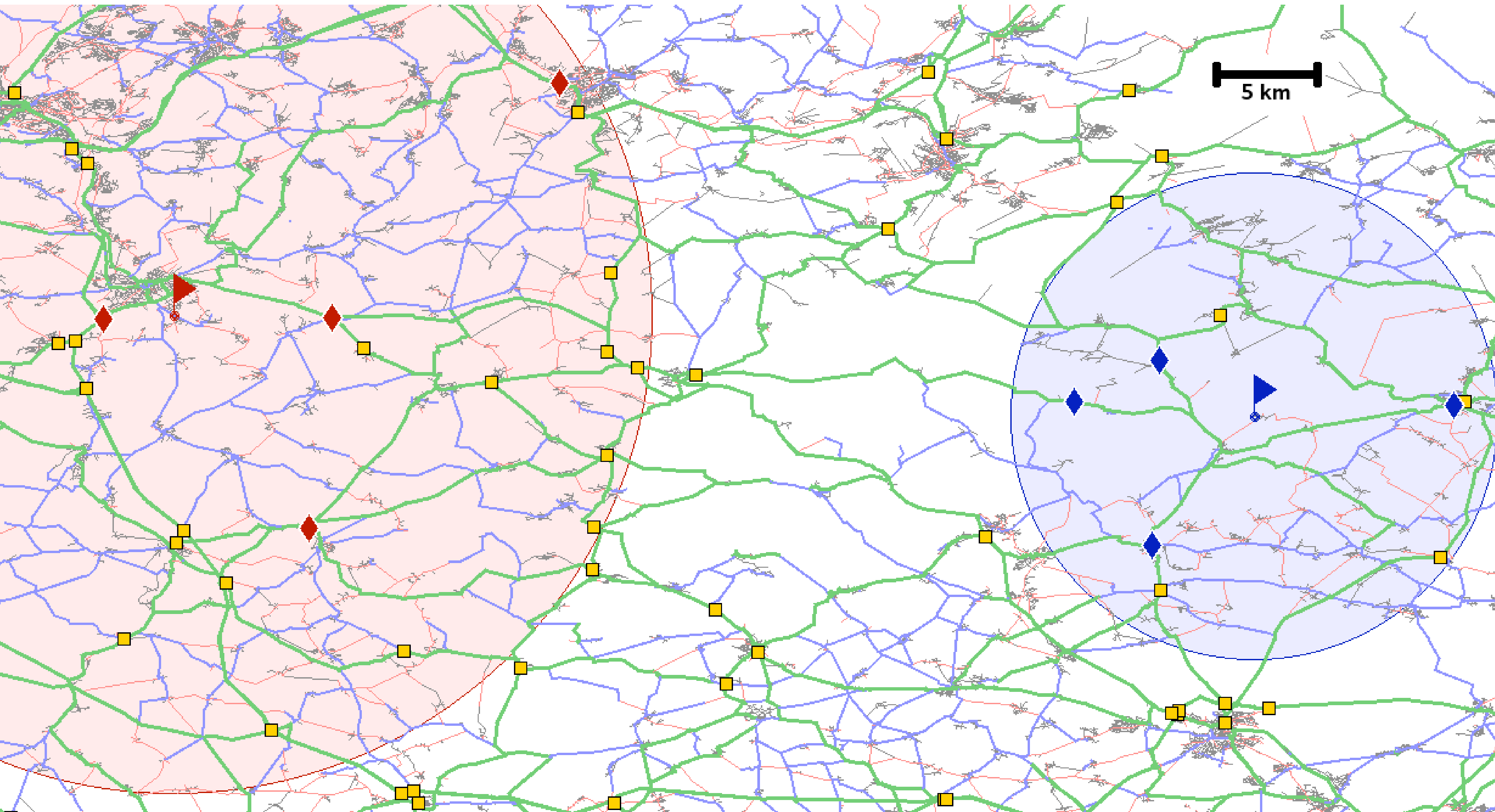


- Hierarchien ausnutzen



- Teilabschnitte tabellieren

Transit-Node Routing



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 Wege

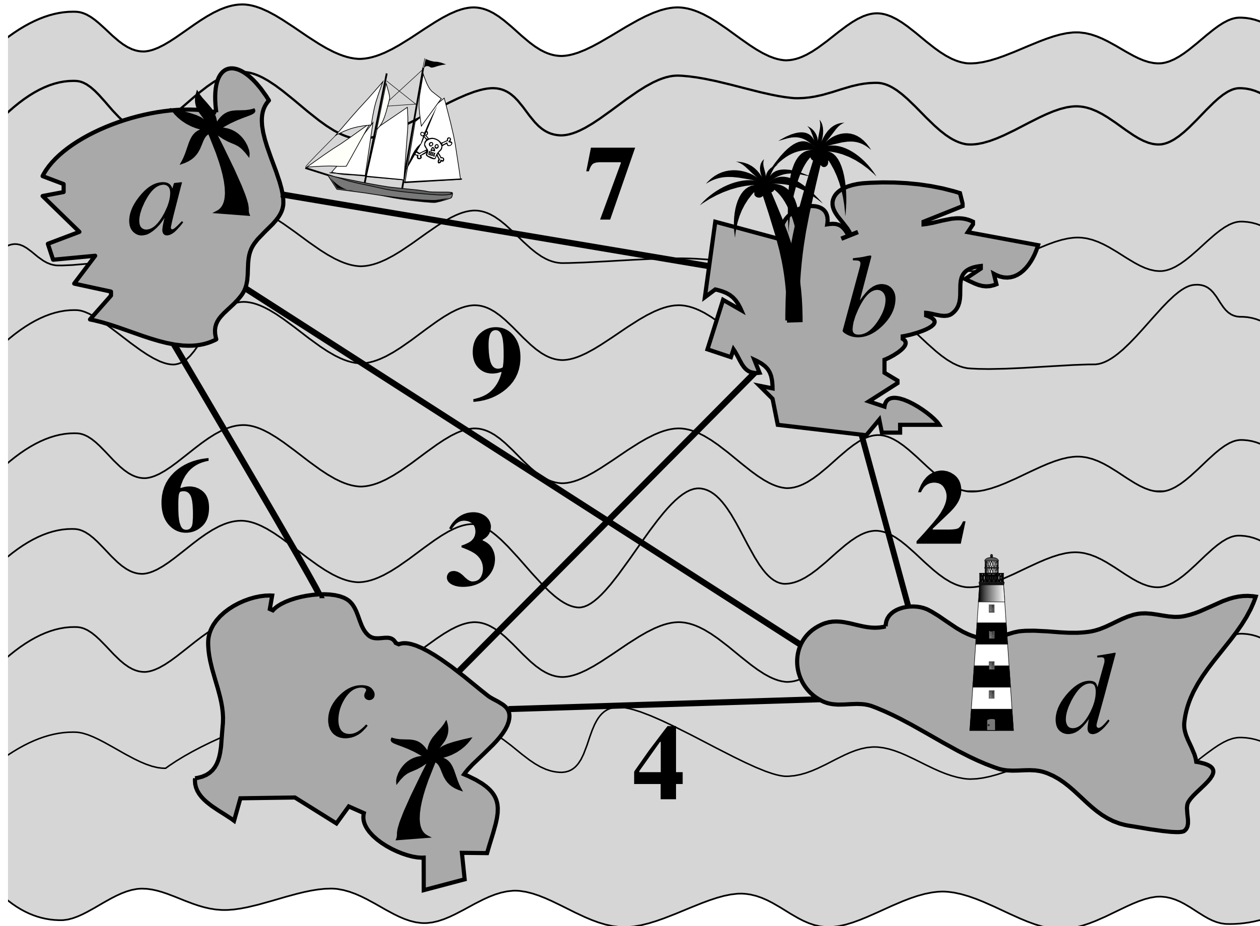
Viele Arbeiten zu besseren Prioritätslisten $\rightsquigarrow \mathrm{O}(m + n \log\log n)$
[Thorup 2004]

Verallgemeinerungen

- Mehre Zielfunktionen abwägen
- Mehre Ziele in beliegiger Reihenfolge anfahren siehe auch Optimierungskapitel
- Mehrere disjunkte Wege

Fast alles schwierig (NP-hart)

11 Minimale Spannbäume



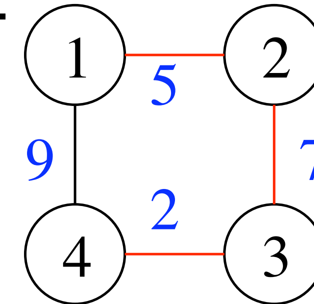
Minimale Spannbäume (MSTs)

ungerichteter (zusammenhängender) Graph $G = (V, E)$.

Knoten V, $n = |V|$, e.g., $V = \{1, \ldots, n\}$

Kanten $e \in E$, $m = |E|$, two-element subsets of V.

Kantengewichte $c(e) \in \mathbb{R}_+$.

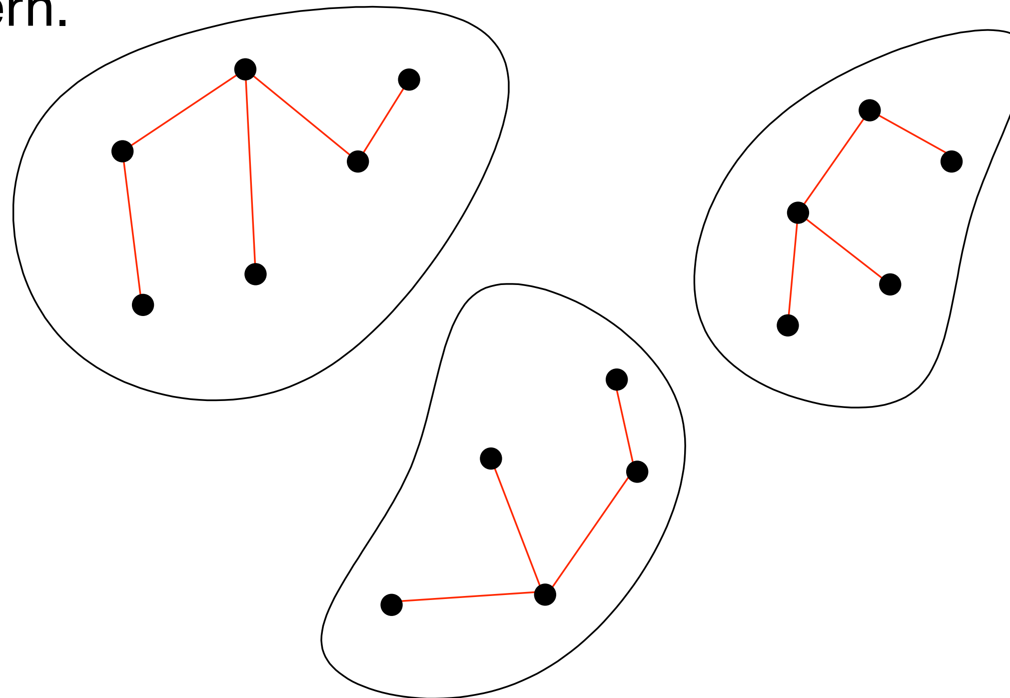


Finde Baum (V, T) mit minimalem Gewicht $\sum_{e \in T} c(e)$ der alle Knoten verbindet.

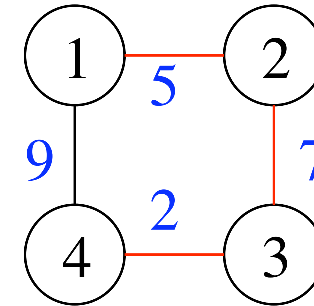
Minimale spannende Wälder (MSF)

Falls G nicht zusammenhängend, finde
minimalen spannenden Wald T der alle
Zusammenhangskomponenten von G aufspannt.

MST Algorithmen lassen sich leicht zu MSF Algorithmen
verallgemeinern.



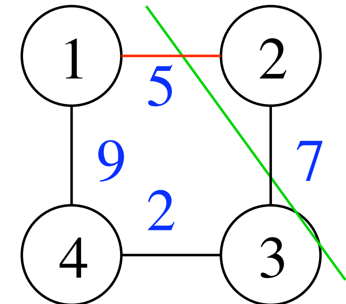
Anwendungen



- Netzwerk-Entwurf
- Bottleneck-Shortest-Paths:
  Suche s–t-Pfad,
  dessen max. Kantengewicht minimal ist.
  Dies ist der Pfad im MST!
- Clustering: Lass schwere MST-Kanten weg. Teilbäume definieren Cluster. Konkret z. B. Bildsegmentierung
- Näherungslösungen für schwere Probleme, z. B. Handlungsreisendenproblem, Steinerbaumproblem.
  Siehe Buch, VL G. theoretischer Informatik, Algorithmen II.

11.1 MST-Kanten auswählen und verwerfen

Die Schnitteigenschaft (Cut Property)

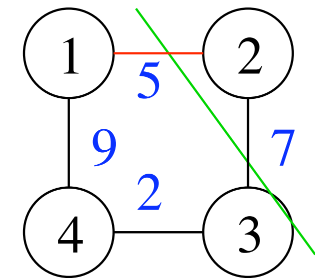


Für beliebige Teilmenge $S \subset V$ betrachte die Schnittkanten

$$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$$

Die leichteste Kante in C kann in einem MST verwendet werden.

Die Schnitteigenschaft (Cut Property)

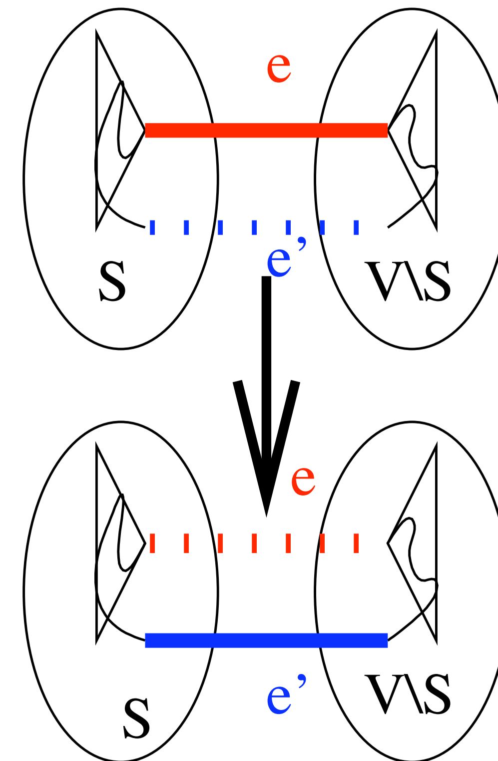


Für beliebige Teilmenge $S \subset V$ betrachte die Schnittkanten

$C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$

Die leichteste Kante in C

kann in einem MST verwendet werden.



Beweis:

Angenommen MST T' benutzt Kante e' zwischen

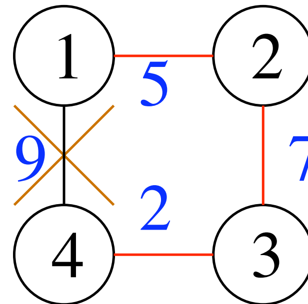
S und $V \setminus S$ mit $c(e) \leq c(e')$.

Dann ist $T = T' \setminus \{e'\} \cup \{e\}$ auch ein MST.

(Es gilt sogar, $c(e) = c(e')$.)

Die Kreiseigenschaft (Cycle Property)

Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt



Die Kreiseigenschaft (Cycle Property)

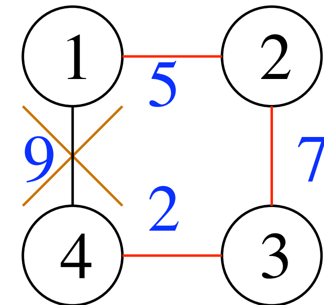
Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt

Beweis.

Angenommen MST T' benutzt die schwerste Kante e' auf Kreis C mit $c(e) \leq c(e')$.

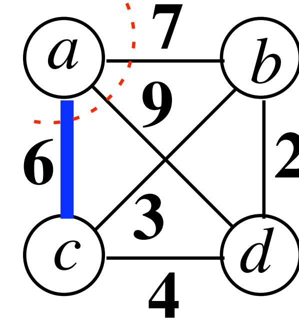
Dann ist $T = T' \setminus \{e'\} \cup \{e\}$ auch ein MST.

(Es gilt sogar, $c(e) = c(e')$.) $\square$



11.2 Der Jarník-Prim Algorithmus

[Jarník 1930, Prim 1957]



Idee: Lasse einen Baum wachsen

$T := \emptyset$

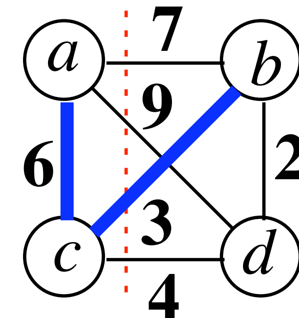
$S := \{s\}$ for arbitrary start node s

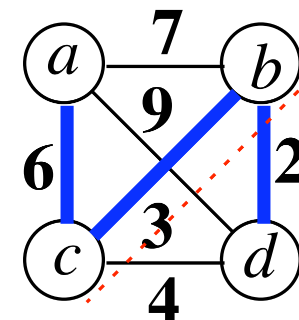
repeat $n-1$ times

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

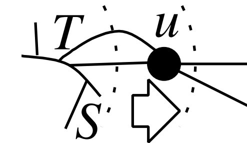
pick any $s \in V$

$d = \{\infty, \ldots, \infty\}$; 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**

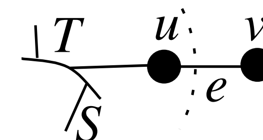
if $c(e) < d[v]$ **then** // relax

$d[v]$:= $c(e)$

parent$[v]$:= u // update tree

if $v \in Q$ **then** Q.decreaseKey(v)

else Q.insert(v)



return $\{(v, \text{parent}[v]) : v \in V \setminus \{s\}\}$

Analyse

Praktisch identisch zu Dijkstra

- $O(m+n)$ Zeit ausserhalb der PQ
- n deleteMin (Zeit $O(n \log n)$)
- $O(m)$ decreaseKey

$\rightsquigarrow$ $O((m+n)\log n)$ mit binären Heaps

$\rightsquigarrow$ $O(m+n\log n)$ mit Fibonacci Heaps

Wichtigster Unterschied: monotone PQs reichen nicht   Warum?

11.3 Kruskals Algorithmus [1956]

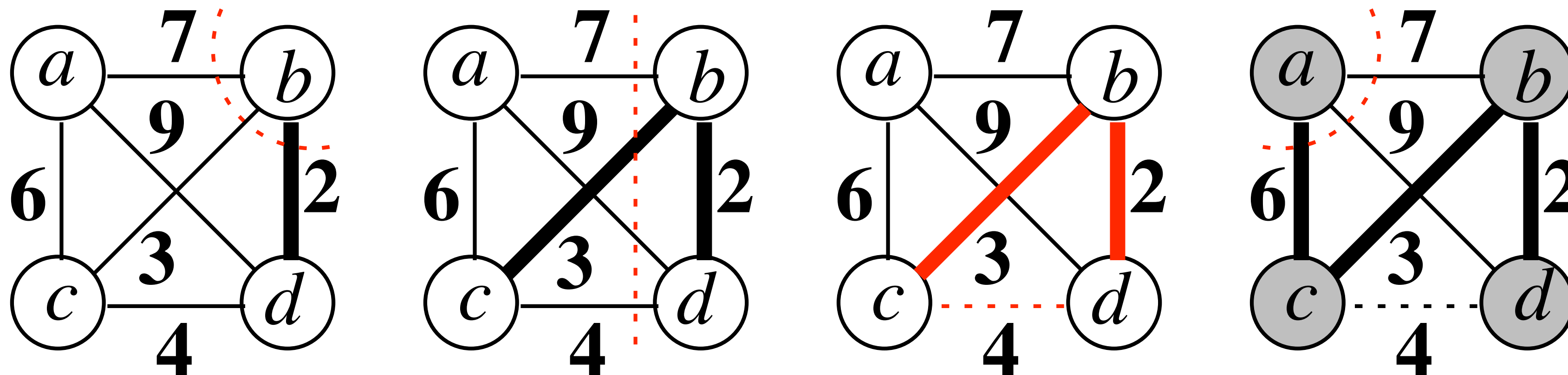
$T := \emptyset$ // subforest of the MST

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

$\Longrightarrow (u,v)$ ist leichteste Kante im cut $(\text{Komponente}(u), V \setminus \text{Komponente}(u))$

$\Longrightarrow (u,v) \in$ MST

Sonst: benutze Kreiseigenschaft

$\Longrightarrow (u,v)$ ist schwerste Kante im Kreis $\langle u, v, v\text{–}u\text{-Pfad in } T \rangle$

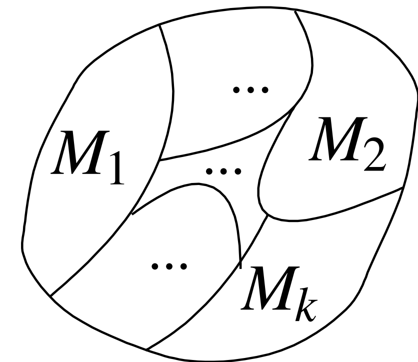
$\Longrightarrow (u,v) \notin$ MST ■

11.4 Union-Find Datenstruktur

Verwalte Partition der Menge $1..n$, d. h., Mengen (Blocks) $M_1,\ldots,M_k$ mit

$$M_1 \cup \cdots \cup M_k = 1..n,$$

$$\forall i \neq j : M_i \cap M_j = \emptyset$$



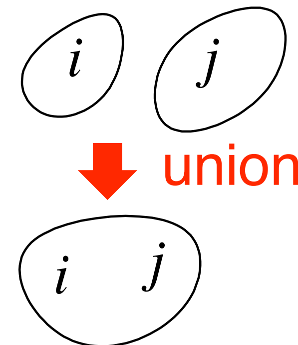
Class UnionFind$(n : \mathbb{N})$

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.

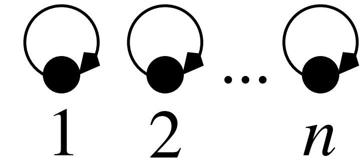


Union-Find Datenstruktur – Erste Version

Class UnionFind$(n : \mathbb{N})$

parent$=\langle 1, 2, \ldots, n\rangle$: **Array** $[1..n]$ **of** $1..n$

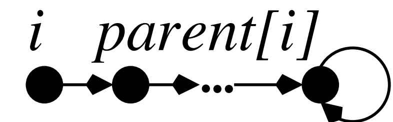
invariant parent-refs lead to unique Partition-Reps



Function find$(i : 1..n) : 1..n$

if parent$[i] = i$ **then return** i

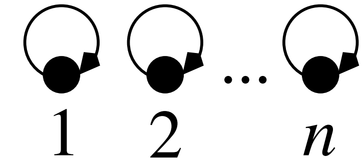
else return find(parent$[i]$)



Union-Find Datenstruktur – Erste Version

Class UnionFind$(n : \mathbb{N})$

parent$=\langle 1,2,\ldots,n\rangle$: **Array** $[1..n]$ **of** $1..n$

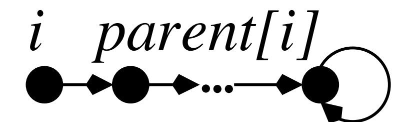


invariant parent-refs lead to unique Partition-Reps

Function find$(i : 1..n) : 1..n$

if parent$[i] = i$ **then return** i

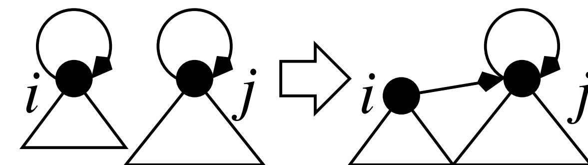
else return find(parent$[i]$)



Procedure link$(i, j : 1..n)$

assert i and j are representatives of different blocks

parent$[i]$:= j



Procedure union$(i, j : 1..n)$

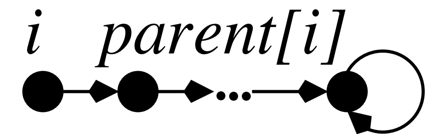
if find$(i) \neq$ find(j) **then** link(find(i), find(j))

Union-Find Datenstruktur – Erste Version

Analyse:

$+$: union braucht konstante Zeit

$-$: find braucht Zeit $\Theta(n)$ im schlechtesten Fall !



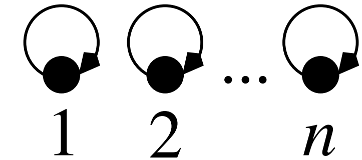
zu langsam.

Idee: find-Pfade kurz halten

Pfadkompression

Class UnionFind$(n : \mathbb{N})$

parent$=\langle 1,2,\ldots,n\rangle$: **Array** $[1..n]$ **of** $1..n$



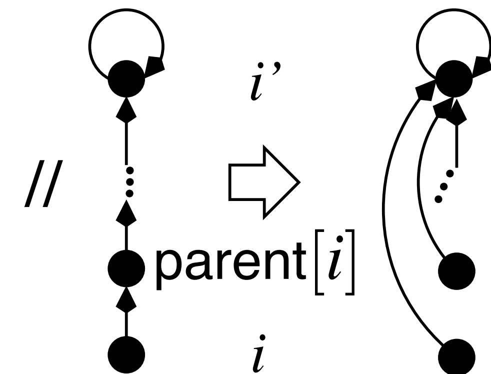
Function find$(i : 1..n) : 1..n$

if parent$[i] = i$ **then return** i

else i' := find(parent$[i]$)

parent$[i]$:= i' //

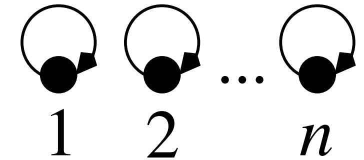
return i'



Union by Rank

Class UnionFind$(n : \mathbb{N})$

parent$=\langle 1, 2, \ldots, n \rangle$: **Array** $[1..n]$ **of** $1..n$

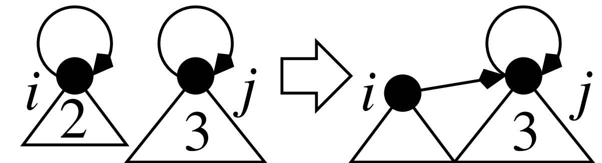


rank$=\langle 0, \ldots, 0 \rangle$: **Array** $[1..n]$ **of** $0..\log n$

Procedure link$(i, j : 1..n)$

assert i and j are representatives of different blocks

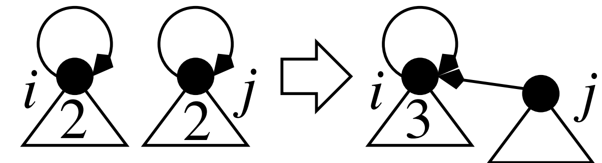
if rank$[i] <$ rank$[j]$ **then** parent$[i] := j$



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 $\mathrm{gen}[x]$

invariant x ist Repr. $\Rightarrow$ x's Menge hat Größe mindestens $2^{\mathrm{gen}[x]}$

Korollar: find braucht Zeit $\mathrm{O}(\log n)$

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 $\mathrm{O}(m\alpha_T(m,n))$ 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 α_T die inverse Ackermannfunktion.
$\alpha_T(m,n) = \omega(1)$ aber ≤ 5 für alle physikalisch denkbaren n, m.

Kruskal mit Union-Find

Sei $V = 1..n$

Tc : UnionFind(n) // encodes components of forest T

foreach $(u, v) \in E$ in ascending order of weight **do** // sort

  if Tc.find$(u) \neq$ Tc.find(v) **then**

    output $\{u, v\}$

    Tc.union(u, v) // link reicht auch

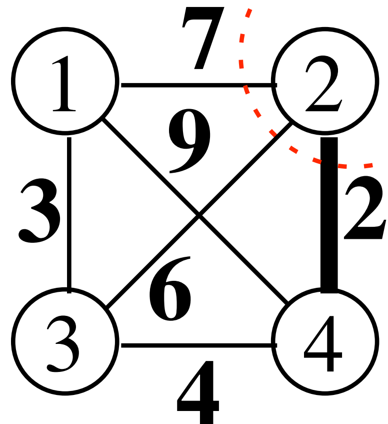
Zeit $O(m \log m)$. Schneller für ganzzahlige Gewichte.

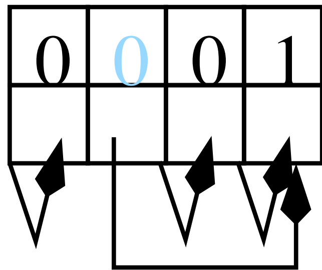
Graphrepräsentation: Kantenliste

Bäume im MSF $\leftrightarrow$ Blöcke in Partition $\longrightarrow$ Wurzelbäume

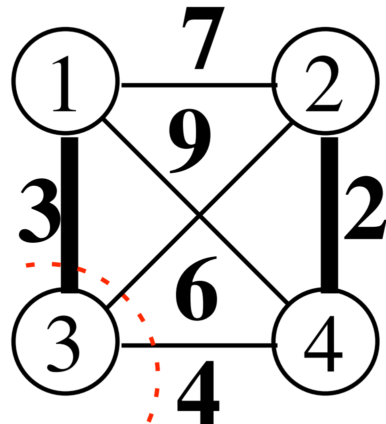
aber mit anderer Struktur als die Bäume im MSF

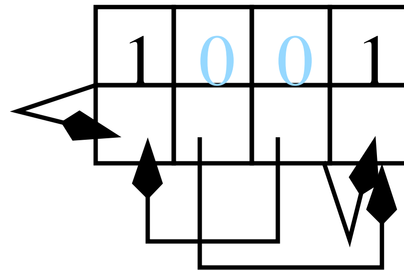
Beispiel



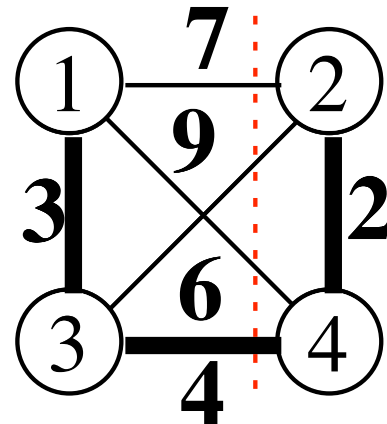


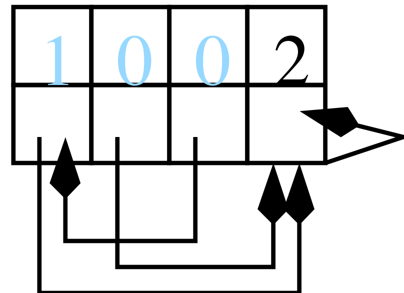
link



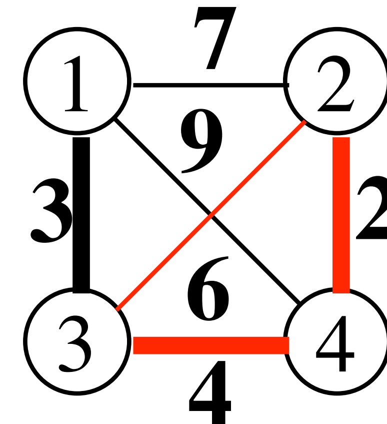


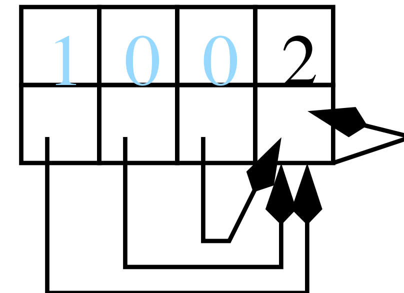
link





link





compress

Vergleich Jarník-Prim $\leftrightarrow$ Kruskal

Pro Jarník-Prim

- Asymptotisch gut für alle m, n
- Sehr schnell für $m \gg n$

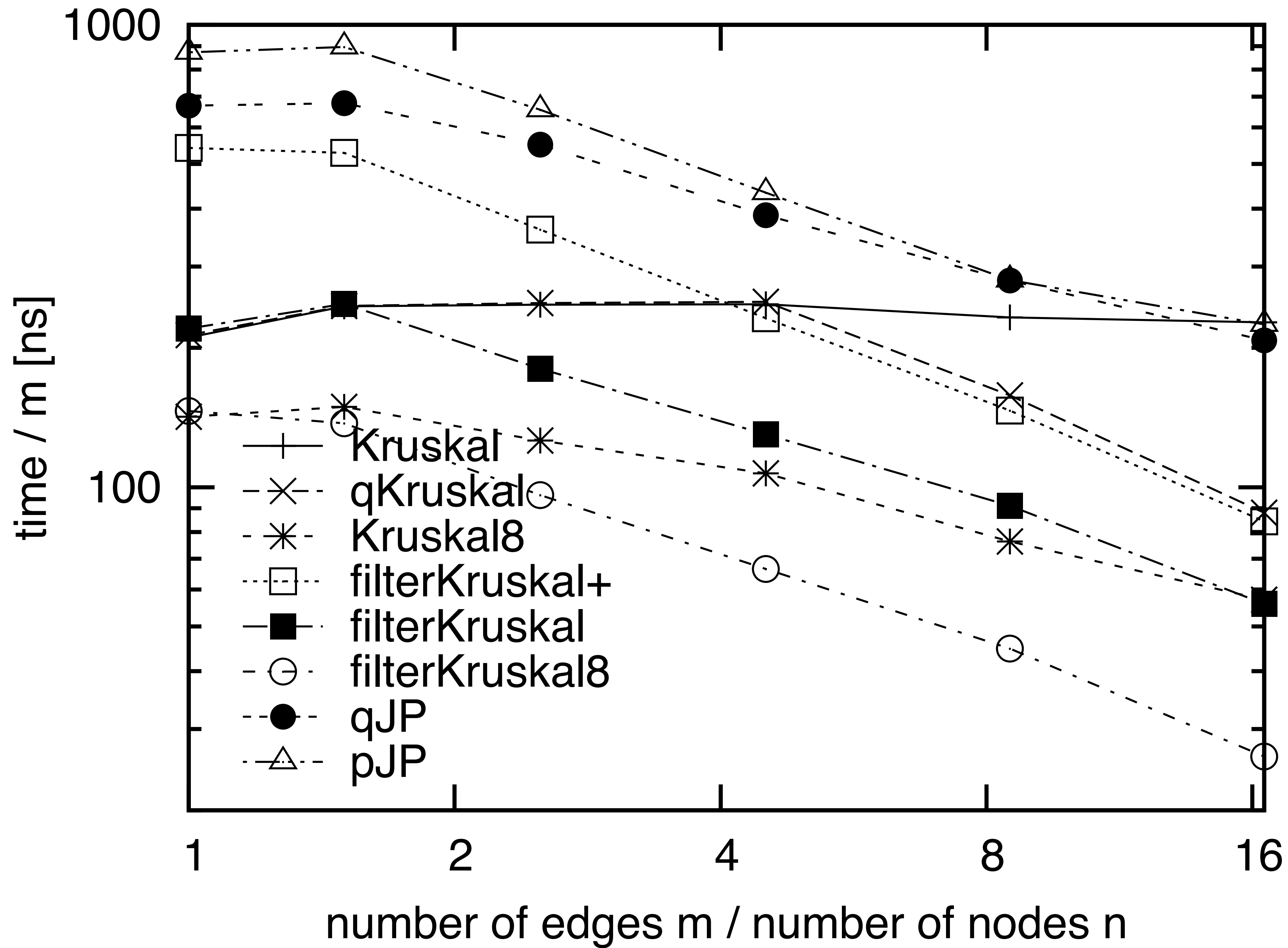
Pro Kruskal

- 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 here":
  Praktikabler externer Algorithmus [Sanders Schultes Sibeyn 2004]
  Verbesserung von Kruskal (parallelisierbar, weniger Sortieraufwand).
  [Osipov Sanders Singler 2009]

Messungen, Zufallsgraph, $n = 2^{22}$



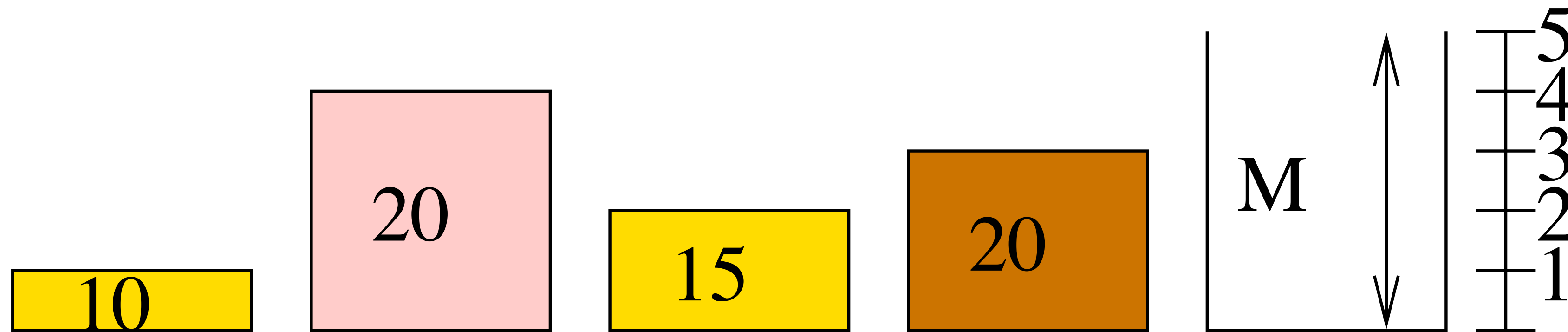
Zusammenfassung

- Schnitt- und Kreiseigenschaft als Basis für abstrakte Algorithmen.
  Entwurfsprinzip:
  Suche nach einfachen, abstrakten Problemeigenschaften.
- 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

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 $\mathbf{x}$ von Gegenständen
- so dass $\sum_{i \in \mathbf{x}} w_i \leq W$ und
- maximiere den Profit $\sum_{i \in \mathbf{x}} p_i$

Allgemein: Maximierungsproblem $(\mathcal{L}, f)$

- $\mathcal{L} \subseteq \mathcal{U}$: zulässige Lösungen
- $f : \mathcal{L} \to \mathbb{R}$ 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-hart

12.1 Black-Box-Löser

- (Ganzzahlige) Lineare Programmierung
- Aussagenlogik
- Constraint-Programming $\approx$ Verallgemeinerung von beidem

Lineare Programmierung

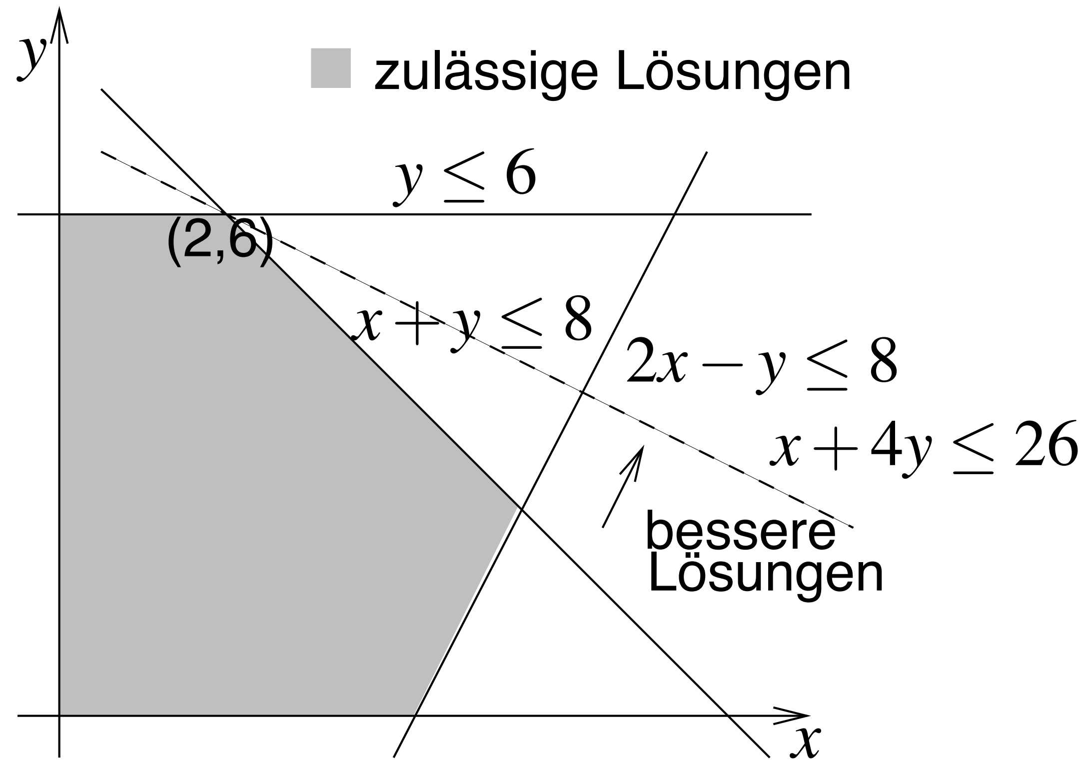
Ein lineares Programm mit n Variablen und m Constraints wird durch das folgende Minimierungs/Maximierungsproblem definiert

- 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

$$\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n : \forall 1 \leq i \leq m : x_i \geq 0 \wedge \mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i\} \ .$$

Sei a_{ij} die j-te Komponente von Vektor $\mathbf{a}_i$.

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

- 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 abschnittweise lineare Kostenfunktionen (z. B. mit Abstand wachsende Transportkosten)

Grenzen

- 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 $\mathrm{O}\left(\max(m,n)^{\frac{7}{2}}\right)$
- 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

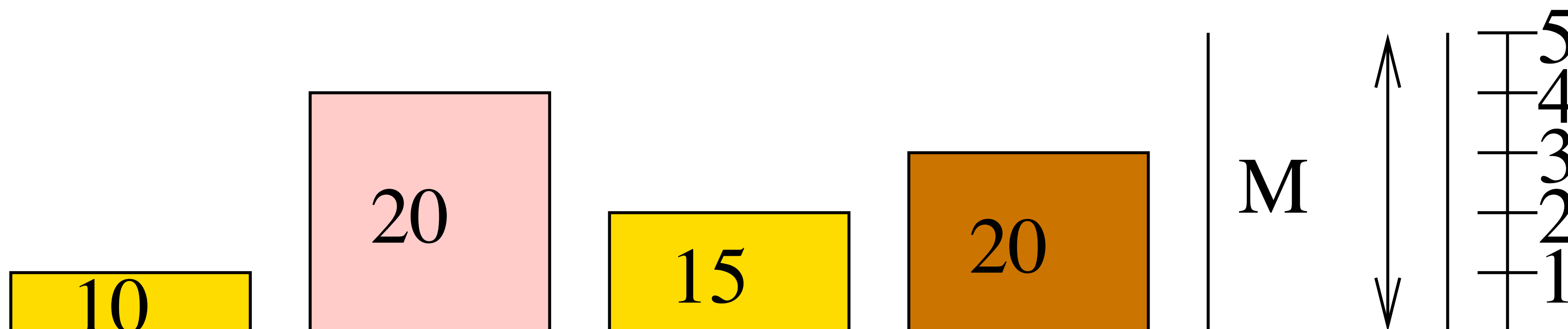
$$\text{maximiere } \mathbf{p} \cdot \mathbf{x}$$

so dass

$$\mathbf{w} \cdot x \leq M, x_i \in \{0,1\} \text{ for } 1 \leq i \leq n \ .$$

$x_i = 1$ gdw Gegenstand i in den Rucksack kommt.

0/1 Variablen sind typisch für ILPs



Umgang mit (M)ILPs

– NP-hard

+ Ausdrucksstarke Modellierungssprache

+ Es gibt generische Lösungsansätze, die manchmal gut funktionieren

+ Viele Möglichkeiten für Näherungslösungen

+ Die Lösung der linearen Relaxierung hilft oft, z. B. einfach runden.

+ Ausgefeilte Softwarepakete

Beispiel: Beim Rucksackproblem gibt es nur eine fraktionale Variable in der linearen Relaxierung – abrunden ergibt zulässige Lösung.

Annähernd optimal falls Gewichte und Profite $\ll$ Kapazität

12.2 Nie Zurückschauen – Greedy-Algorithmen

(deutsch: gierige Algorithmen, wenig gebräuchlich)

Idee: treffe jeweils eine lokal optimale Entscheidung

Optimale Greedy-Algorithmen

- Dijkstra's Algorithmus für kürzeste Wege
- Minimale Spannbäume
  - Jarník-Prim
  - Kruskal
- Selection-Sort (wenn man so will)

Näherungslösungen mit Greedy-Algorithmen

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 $\max \left\{ j : \sum_{i=1}^{j} \right\} > M$

output items $1..j-1$

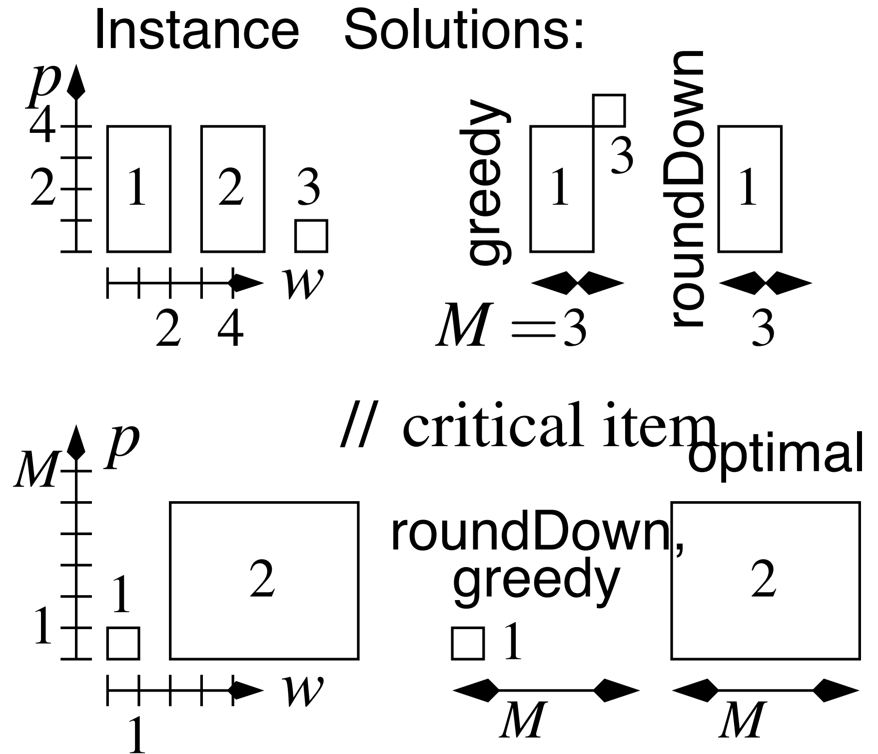
Procedure greedyKnapsack

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



12.3 Dynamische Programmierung – Aufbau aus Bausteinen

Anwendbar wenn, das **Optimalitätsprinzip** gilt:

- Optimale Lösungen bestehen aus optimalen Löungen für Teilprobleme.
- Mehrere optimale Lösungen $\Rightarrow$ es is egal welches benutzt wird.

Beispiel: Rucksackproblem

Annahme: ganzzahlige Gewichte

$P(i,C) :=$ optimaler Profit für Gegenstände $1,\ldots,i$ unter Benutzung von Kapatzität $\leq C$.

Lemma:

$$\forall 1 \leq i \leq n : P(i,C) = \max(P(i-1,C), \\ P(i-1,C-w_i)+p_i)$$

$P(i,C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ bei Kap. C.

Lemma: $P(i,C) = \max(P(i-1,C), P(i-1,C-w_i)+p_i)$

Beweisanfang:

$P(i,C) \geq P(i-1,C)$: Setze $x_i = 0$, benutze optimale Teillösung.

$P(i,C) \geq P(i-1,C-w_i)+p_i$: Setze $x_i = 1$ ...

$P(i,C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ bei Kap. C.

Lemma: $P(i,C) = \max(P(i-1,C), P(i-1,C-w_i)+p_i)$

Beweis fortgesetzt:

$$P(i,C) \leq \max(P(i-1,C), P(i-1,C-w_i)+p_i)$$

angenommen das stimmt nicht $\longrightarrow$

$\exists \mathbf{x} : \mathbf{x}$ ist is optimal für das Teilprolem so dass

$$\begin{aligned} P(i-1,C) &< \mathbf{p}\cdot\mathbf{x} \wedge \\ P(i-1,C-w_i)+p_i &< \mathbf{p}\cdot\mathbf{x} \end{aligned}$$

$P(i,C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ bei Kap. C.

Lemma: $P(i,C) = \max(P(i-1,C), P(i-1,C-w_i)+p_i)$

Beweis weiter fortgesetzt:

$P(i,C) \leq \max(P(i-1,C), P(i-1,C-w_i)+p_i)$

angenommen das stimmt nicht $\longrightarrow$

$\exists \mathbf{x} : \mathbf{x}$ ist is optimal für das Teilprolem so dass

$$\begin{aligned} P(i-1,C) &< \mathbf{p}\cdot\mathbf{x} \wedge \\ P(i-1,C-w_i)+p_i &< \mathbf{p}\cdot\mathbf{x} \end{aligned}$$

Fall $x_i = 0$: $\mathbf{x}$ ebenfalls zulässig für $P(i-1,C)$. Außerdem gilt, $P(i-1,C) \geq \mathbf{p}\cdot\mathbf{x}$. Widerspruch

Fall $x_i = 1$: Mit $x_i = 0$ erhalten wir eine zulässige Lösung $\mathbf{x}'$ für $P(i-1,C-w_i)$ mit Profit $\mathbf{p}\cdot\mathbf{x}' = \mathbf{p}\cdot\mathbf{x} - p_i$. Außerdem, $P(i-1,C-w_i)+p_i \geq \mathbf{p}\cdot\mathbf{x}$. Widerspruch

Berechung von $P(i,C)$ elementweise:

Procedure knapsack($\mathbf{p}$, $\mathbf{c}$, n, M)

  array P$[0\ldots M] = [0,\ldots,0]$

  bitarray decision$[1\ldots n, 0\ldots M] = [(0,\ldots,0),\ldots,(0,\ldots,0)]$

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

    // invariant: $\forall C \in \{1,\ldots,M\} : \mathrm{P}[C] = P(i-1,C)$

    for C := M **downto** w_i **do**

      if $\mathrm{P}[C-w_i] + p_i > \mathrm{P}[C]$ **then**

        P$[C]$:= $\mathrm{P}[C-w_i] + p_i$

        decision$[i,C]$:= 1

Rekonstruktion der Lösung

$C := M$

array $\mathbf{x}[1\ldots n]$

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

  $\mathbf{x}[i]$:= decision$[i, C]$

  if $\mathbf{x}[i] = 1$ **then** $C := C - w_i$

endfor

return $\mathbf{x}$

Analyse:

Zeit: $\mathrm{O}(nM)$ pseudopolynomiell

Space: $M + \mathrm{O}(n)$ Maschinenwörter plus Mn bits.

Beispiel

maximiere $(10,20,15,20)\cdot\mathbf{x}$

so dass $(1,3,2,4)\cdot\mathbf{x}\le 5$

$P(i,C),(\text{decision}[i,C])$

$i\setminus C$	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

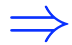
1. Was sind die Teilprobleme? Kreativität!
2. Wie setzen sich optimale Lösungen aus Teilproblemlösungen zusammen? Beweisnot
3. Bottom-up Aufbau der Lösungstabelle einfach
4. Rekonstruktion der Lösung einfach
5. Verfeinerungen:
   Platz sparen, Cache-effizient, Parallelisierung Standard-Trickkiste

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.

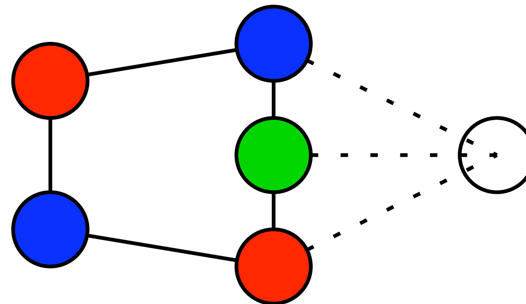


Keine gute Teilstrategie für 21 Runden.

Frage: Wie kann man constrained shortest path trotzdem als dynamischen Programmier-problem modellieren?

Gegenbeispiel: Austauschbarkeit

Optimale Graphfärbungen sind nicht austauschbar.



12.4 Systematische Suche

Idee: Alle (sinnvollen) Möglichkeiten ausprobieren.

Anwendungen:

- Integer Linear Programming (ILP)
- Constraint Satisfaction
- SAT (Aussagenlogik)
- Theorembeweiser (Prädikatenlogik,. . .)
- konkrete NP-harte Probleme
- Strategiespiele
- Puzzles

Beispiel: Branch-and-Bound für das Rucksackproblem

Function bbKnapsack$((p_1,\ldots,p_n),(w_1,\ldots,w_n),M) : \mathscr{L}$

assert $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$

$\hat{\mathbf{x}}$=heuristicKnapsack$((p_1,\ldots,p_n),(w_1,\ldots,w_n),M) \;: \mathscr{L}$

$\mathbf{x} : \mathscr{L}$

recurse$(1,M,0)$

return $\hat{\mathbf{x}}$

// Find solutions assuming $x_1,\ldots,x_{i-1}$ are fixed,

// $M' = M - \sum_{k<i} x_i w_i, P = \sum_{k<i} x_i p_i.$

Procedure recurse$(i,M',P : \mathbb{N})$

$\mathbf{x}$ // current Solution

$\hat{\mathbf{x}}$ // best solution so far

Procedure recurse$(i, M', P : \mathbb{N})$

$u := P + \text{upperBound}((p_i, \ldots, p_n), (w_i, \ldots, w_n), M')$

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$(i + 1, M' - w_i, P + p_i)$

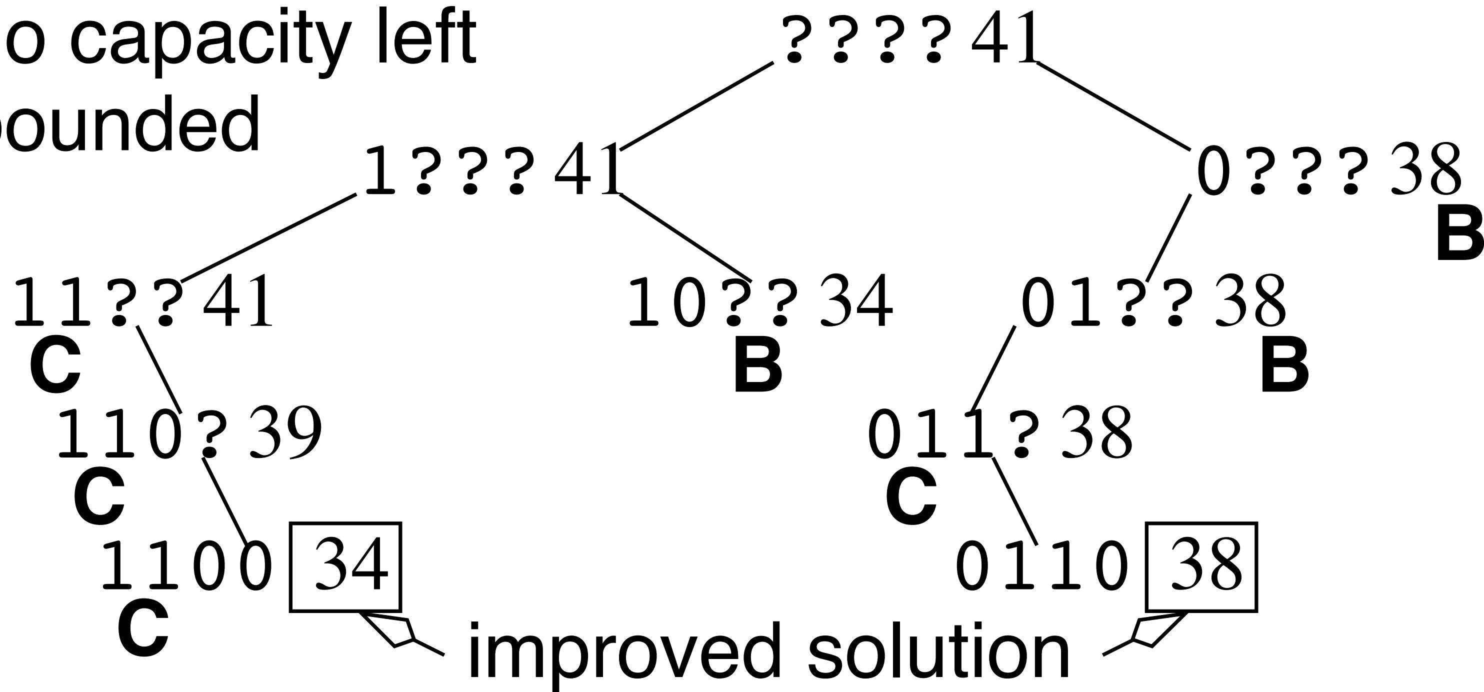
if $u > \mathbf{p} \cdot \hat{\mathbf{x}}$ **then** $x_i := 0$; recurse$(i + 1, M', P)$

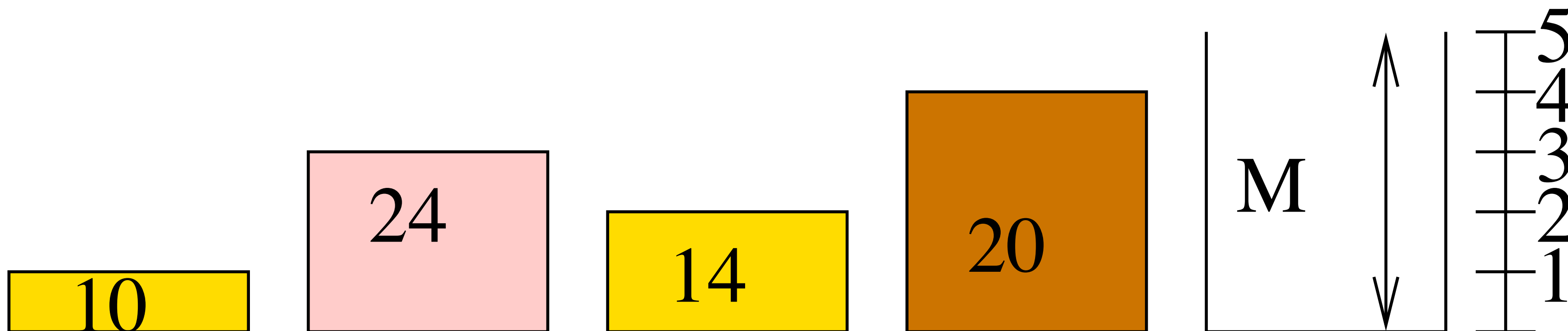
Schlechtester Fall: 2^n rekursive Aufrufe

Im Mittel: Linearzeit?

Beispielrechnung

C no capacity left
B bounded





Branch-and-Bound – allgemein

Branching (Verzweigen): Systematische Fallunterscheidung,

z. B. rekursiv (Alternative, z. B. Prioritätsliste)

Verweigungsauswahl: 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.

Duplikatelimination: Einmal suchen reicht (unnötig bei ILP)

Anwendungsspez. Suchraumbeschränkungen: Schnittebenen (ILP),

Lemma-Generierung (Logik),. . .

12.5 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 $\mathcal{N}(\mathbf{x}) \cap \mathcal{S}$

  if $f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

Hill Climbing

Find some feasible solution $\mathbf{x} \in \mathcal{L}$

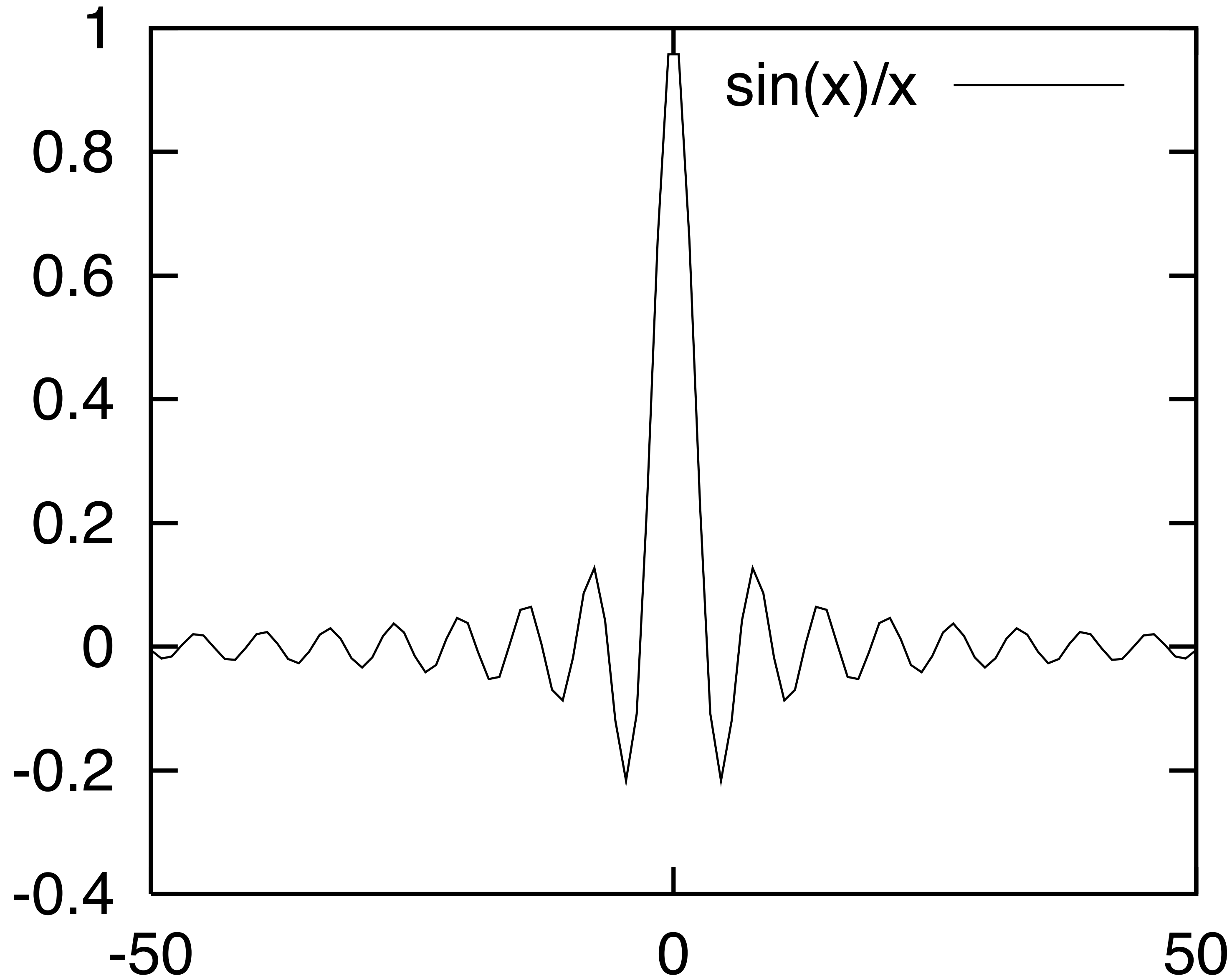
$\hat{\mathbf{x}} := \mathbf{x}$ // best solution found so far

loop

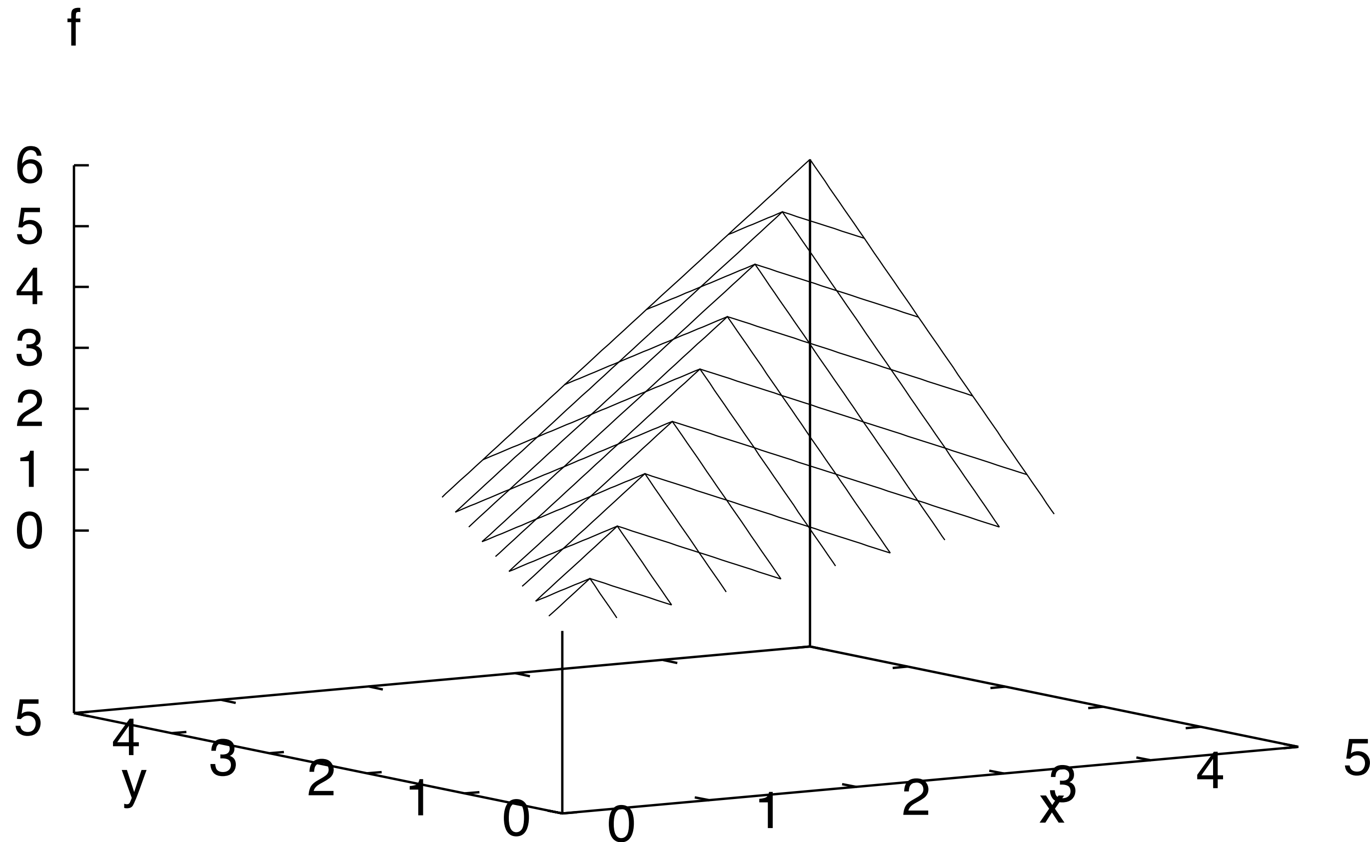
  if $\exists \mathbf{x} \in \mathcal{N}(\mathbf{x}) \cap \mathcal{L} : f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

  else return $\hat{\mathbf{x}}$ // local optimum found

Problem: Lokale Optima



Warum die Nachbarschaft wichtig ist

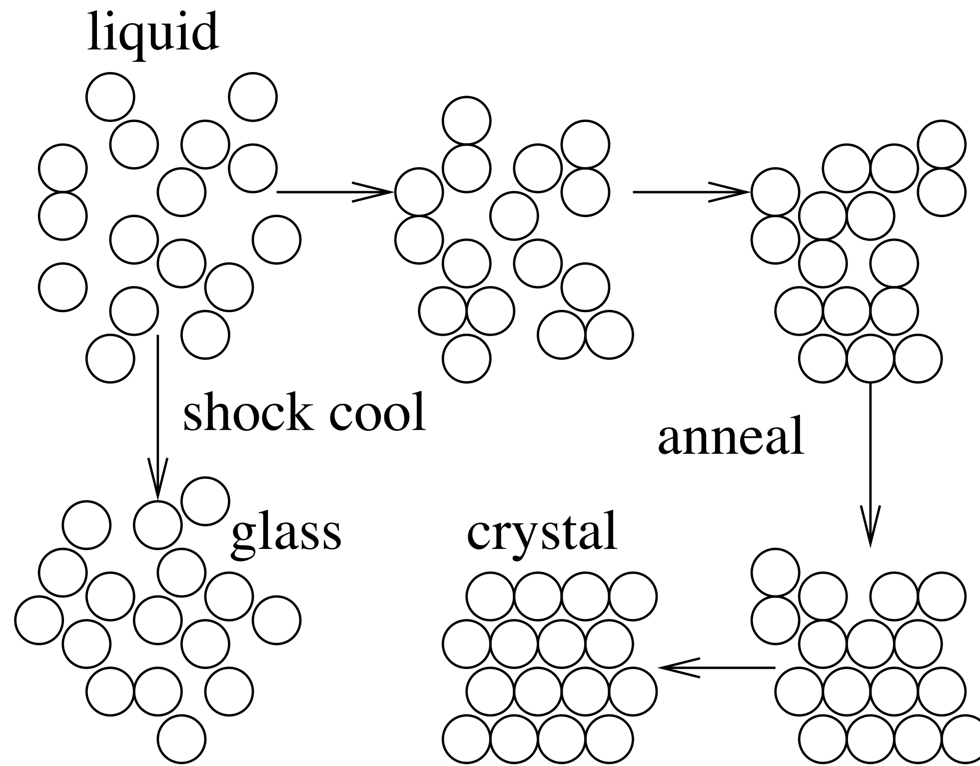


Gegenbeispiel für Koordinatensuche

Jenseits von Hill Climbing

Auch Verschlechterungen akzeptieren.

- Simulated Annealing: physikalische Analogie
- Tabusuche
- ...


liquid
shock cool
anneal
glass
crystal

12.6 Evolutionäre Algorithmen

Ausführliche Behandlung würde den Rahmen sprengen.

Verallgemeinerung von lokaler Suche:

- $\mathbf{x} \longrightarrow$ Population von Lösungskandidaten
- Reproduktion fitter Lösungen
- Mutation ähnlich lokaler Suche
- zusätzlich: geschlechtliche Vermehrung.

  Idee: erben guter Eigenschaften beider Eltern

Zusammenfassung Vor- und Nachteile

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.

Linear Programming: Einfach und einigermaßen schnell. Optimal falls das Modell passt. Rundungsheuristiken ergeben Näherungslösungen

Dynamische Programmierung: Optimale Lösungen falls Teilprobleme optimal und austauschbar sind. Hoher Platzverbrauch.

Integer Linear Programming: Leistungsfähiges Werkzeug für optimale Lösungen. Gute Formulierungen können Einiges know how erfordern.

Lokale Suche: Flexibel und einfach. Langsam aber oft gute Lösungen für harte 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.