Karlsruhe Institute of Technology

11. Übung zu Algorithmen I

Institut für Theoretische Informatik, Prof. Sanders

KIT – die Kooperation von Forschungszentrum Karlsruhe GmbH und Universität Karlsruhe (TH)

Forschungszentrum Karlsruhe
in der Helmholtz-Gemeinschaft

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Preisverleihung

Karlsruhe Institute of Technology

- auf der nächsten Folie stehen die Gewinner
- bitte nach vorne kommen und die Tafel(n) Schokolade abholen
- jeweils für Klasse A und Klasse B gibt es eine Tafel Schokolade
- bitte unten für ein Gruppenbild bleiben

- Tutoriumsgewinner wurden vom Tutor bestimmt
- Gesamtgewinner wurde von den Übungsleitern bestimmt

Preisverleihung

Tutoriums-Sieger

Tutorium 1: Georg Hinkel AB
Tutorium 2: Tobias Schürg AB
Tutorium 5: Jan Aidel + Maximilian Schuler AB
Tutorium 6: Felix Schultze AB
Tutorium 7: Alexander Kruck + Mathis Bareis AB
Tutorium 8: Roland Kluge AB
Tutorium 9: Emanuel Poremba AB
Tutorium 11: Johann Böhler A, Tobias Dörr B
Tutorium 12: Marcus Georgi + Kevin Speckmaier AB
Tutorium 14: Alexandra Kiatipi A, Andreas Weber B
Tutorium 15: Patrick Hillert A, Jonas Fuchs + Alexander Weigel B
Tutorium 16: Max Baude + Max Peters AB
Tutorium 17: Stefan Walzer A, Georg Osang B
Tutorium 18: Markus Jung + Andreas Bauer AB
Tutorium 19: Michael Hamann AB
Tutorium 20: Bernhard Strähle + Johann Aydinbas A,
Anna Vekliuk + Marina Poimzew B
Tutorium 21: Luben Alexandrov AB
Tutorium 23: Carlo Hermanin de Reichenfeld A,
Alexander Unseld + Busra Bedir B
Tutorium 24: Malte Vesper AB
Tutorium 25: Sven Zühlsdorf A, Ralf Hauptmann + Andreas Staudt B
Tutorium 26: Alwin Karabiowski AB

Gesamtsieger

sowohl in Klasse `A` als auch in Klasse `B`

Michael Hamann

- Klasse A: 641 ms
- Klasse B: 69 ms bei $\epsilon = 0.0188$

KIT Karlsruhe Institute of Technology
Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825
KIT – die Kooperation von Forschungszentrum Karlsruhe GmbH und Universität Karlsruhe (TH)

Organisatorisches

Karlsruhe Institute of Technology

- Klausuranmeldung bis einschließlich Dienstag, 28.7.
- Übungsscheinanmeldung online!

Erlaubte Hilfsmittel für die Klausur:

- blauer Stift
- DIN-A4 Blatt handbeschrieben
- für die Identifikation: Studentenausweis

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825
KIT – die Kooperation von Forschungszentrum Karlsruhe GmbH und Universität Karlsruhe (TH)

Themen für die Klausur

- alles!
- außer
  - RAM-Code
  - Wahrscheinlichkeitsrechnung
- Die heutige Zusammenfassung umfasst NICHT annähernd alles für die Klausur!
  - Todays summary does not nearly contain everything necessary for the exam!

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825
KIT – die Kooperation von Forschungszentrum Karlsruhe GmbH und Universität Karlsruhe (TH)

Schultmultiplikation

p=0 $: \mathbb{N}$

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

$p := p$ // $n+j$ Ziffern (außer bei $j=0$)

$+$ // $n+1$ Ziffernadditionen (optimiert)

$a \cdot b[j]$ // 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

Karatsuba-Ofman Multiplikation[1962]

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

Function recMult(a,b)

assert a und b haben $n = 2k$ 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

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

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

$+c_{00}$

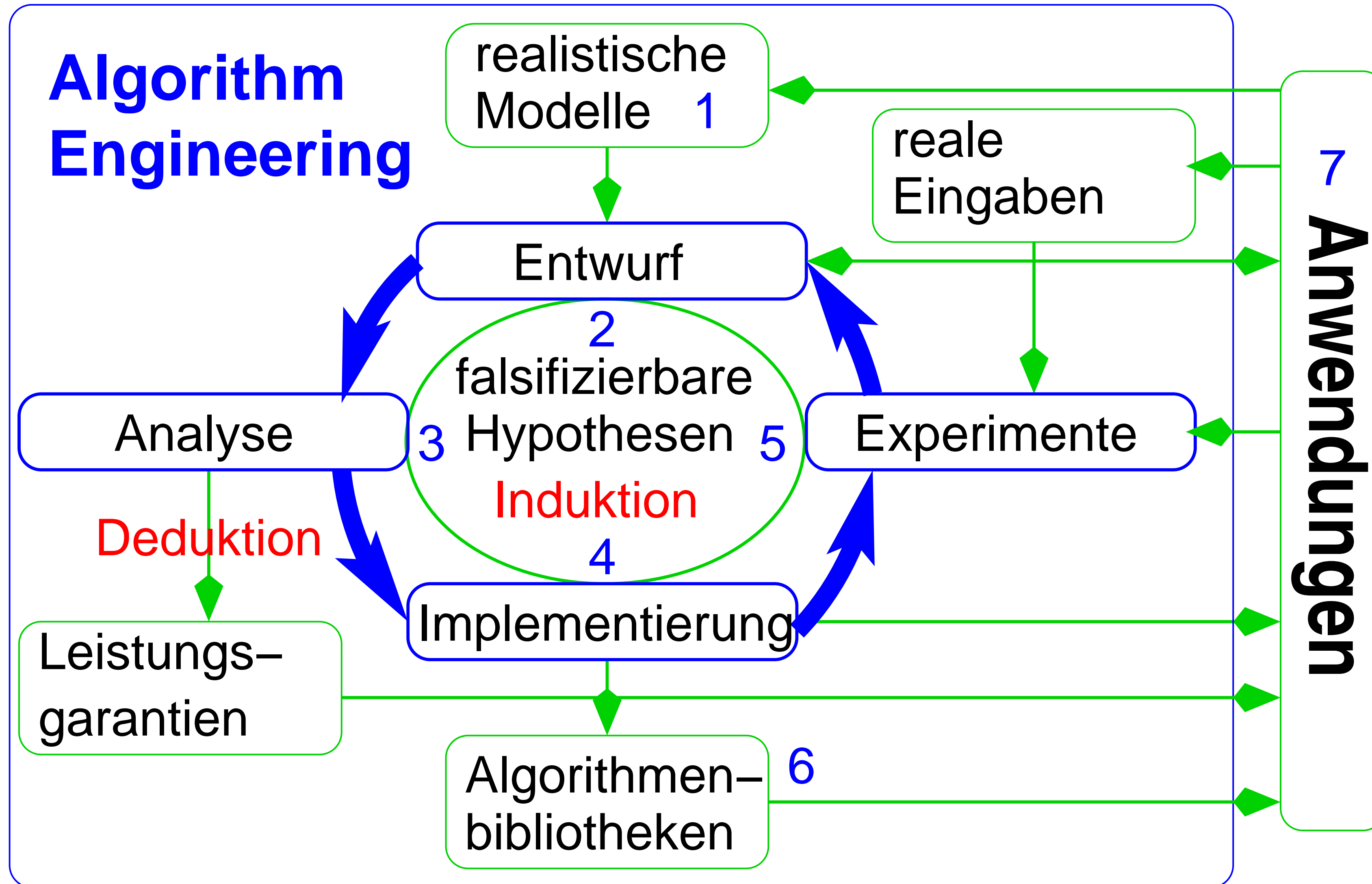
Analyse

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

$\longrightarrow$ (Master-Theorem)

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

Algorithmik als Algorithm Engineering



Zweite Vereinfachung: Asymptotik

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

„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"

$$\Theta(f(n)) = \mathrm{O}(f(n)) \cap \Omega(f(n))$$

„genau"

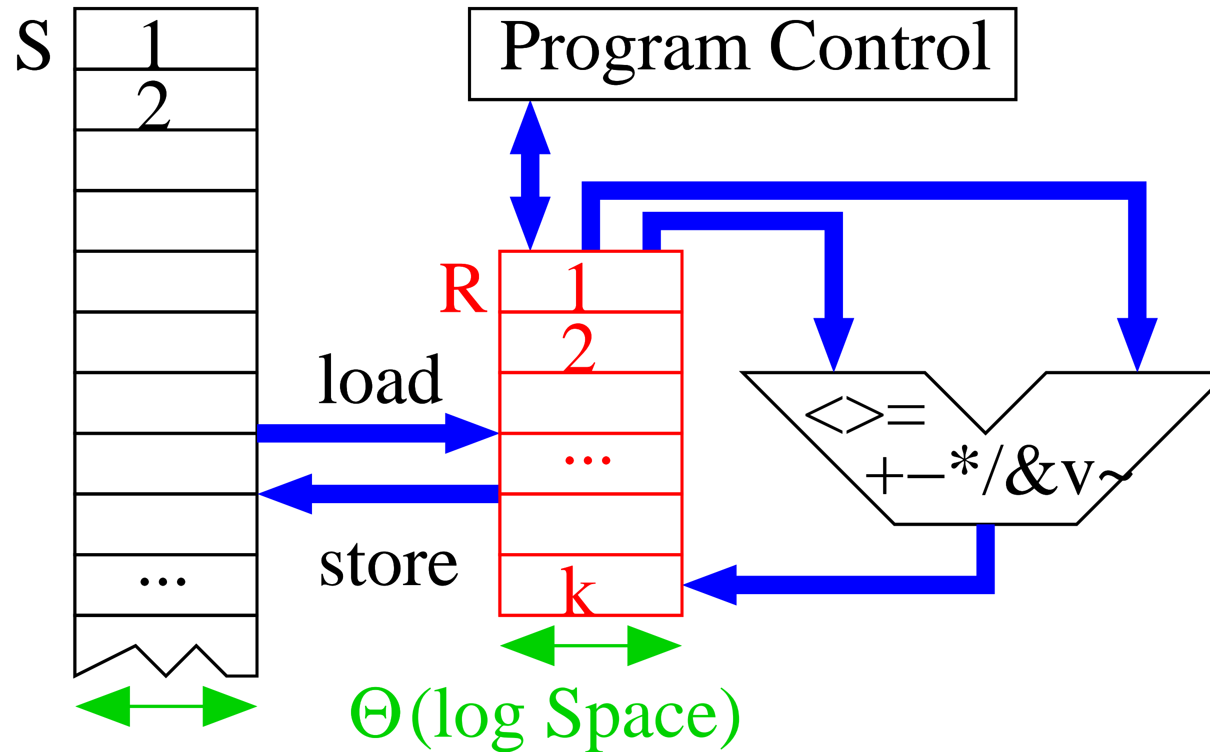
$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„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"

Maschinenmodell



Wortgröße: Genug um alle benutzten Speicherstellen zu adressieren.

Design by Contract / Schleifeninvarianten

assert: Aussage über Zustand der Programmausführung

Vorbedingung: Bedingung für korrektes Funktionieren einer Prozedur

Nachbedingung: Leistungsgarantie einer Prozedur,
falls Vorbedingung erfüllt

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

Schleifeninvariante: gilt vor / nach jeder Ausführung des
Schleifenkörpers

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

Hier: Invarianten als zentrales Werkzeug für Algorithmenentwurf und
Korrektheitsbeweis.

Schleifenanalyse $\rightsquigarrow$ Summen ausrechnen

Das lernen Sie in Mathe

Beispiel: Schulmultiplikation

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

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

$d < b$: z.B. Median bestimmen

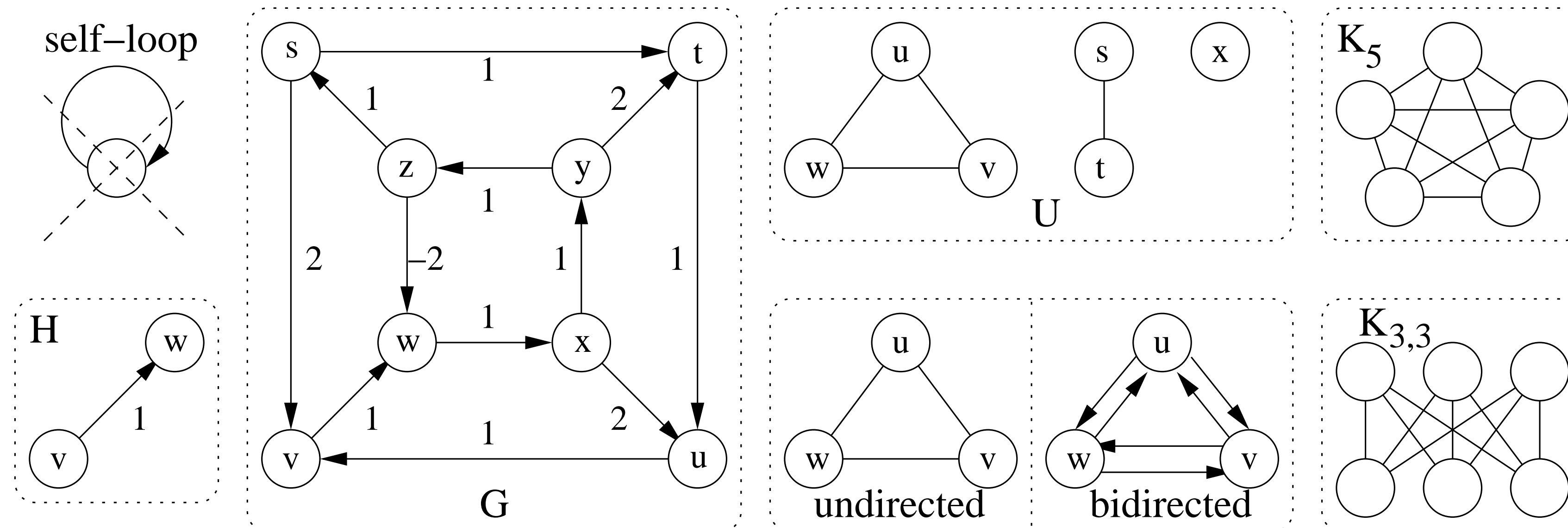
$d = b$: z.B. mergesort, quicksort

$d > b$: z.B. Schulmultiplikation, Karatsuba-Ofman-Multiplikation

Graphen

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

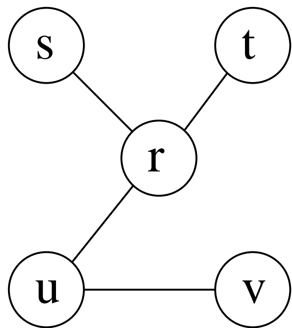
Pfade (einfach, Hamilton-), Kreise, DAGs

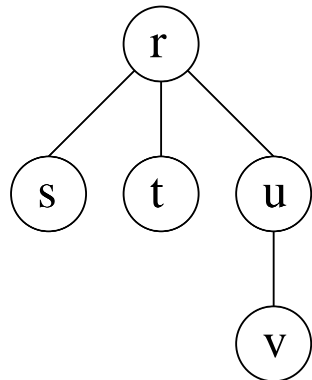


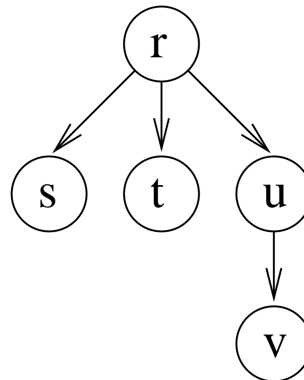
Bäume

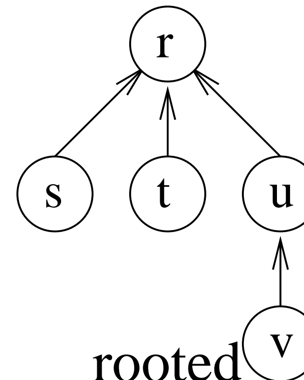
Zusammenhang, Bäume, Wurzeln, Wälder, Kinder, Eltern, . . .

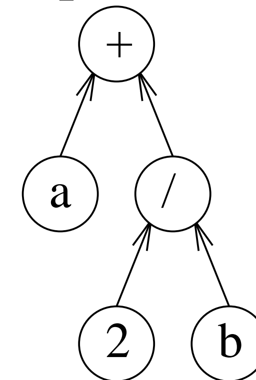
undirected undirected rooted directed expression











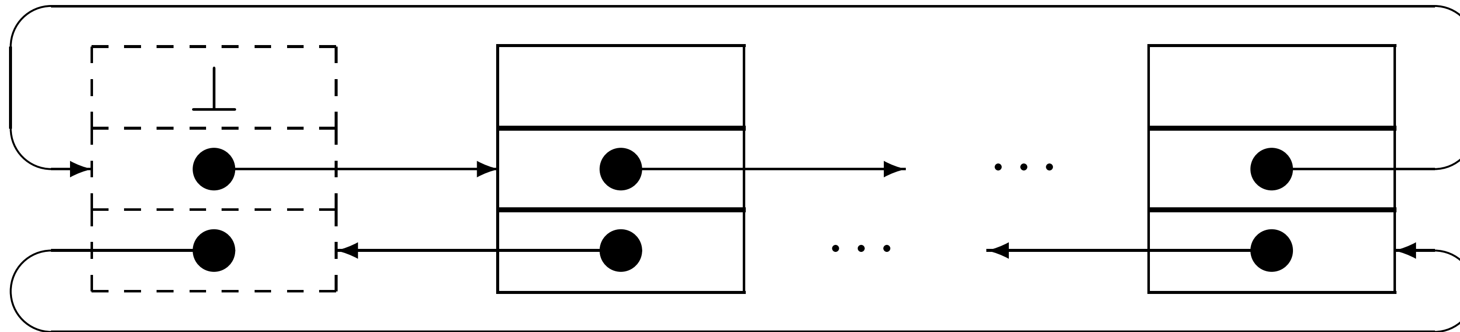
Form Follows Function

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

Verkettete Listen

Doppelt verkettete Listen

Trick: dummy header



+ Invariante immer erfüllt
+ Vermeidung vieler Sonderfälle
  - ⇝ einfach
  - ⇝ lesbar
  - ⇝ schnell
  - ⇝ testbar
  - ⇝ elegant

− Speicherplatz (irrelevant bei langen Listen)

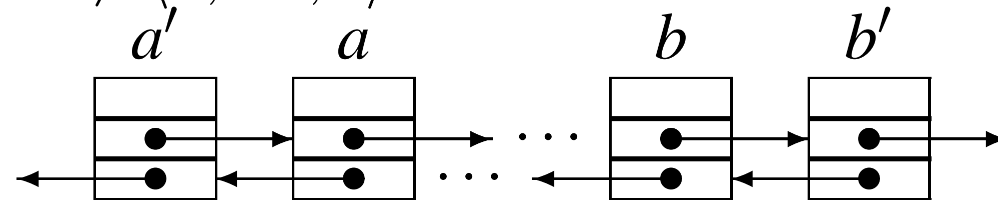
Procedure splice(a,b,t : Handle)//Cut out $\langle a,\ldots,b\rangle$ and insert after t

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

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

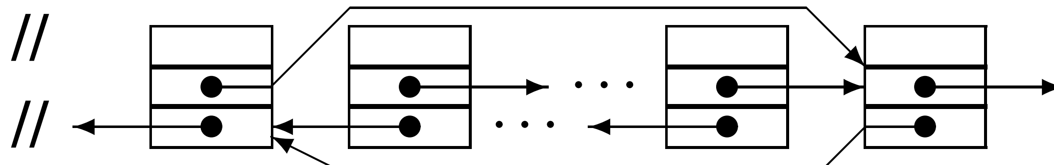
$a' := a \rightarrow$prev

$b' := b \rightarrow$next



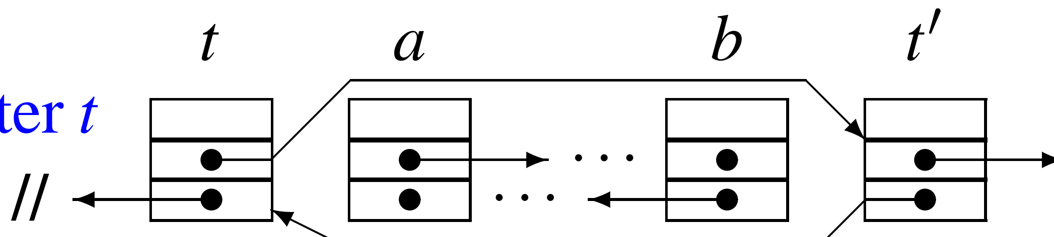
$a' \rightarrow$next $:= b'$ //

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



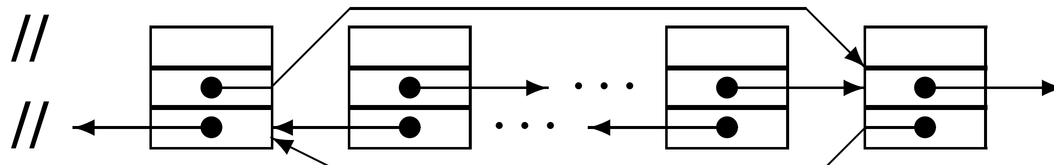
//insert $\langle a,\ldots,b\rangle$ after t

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



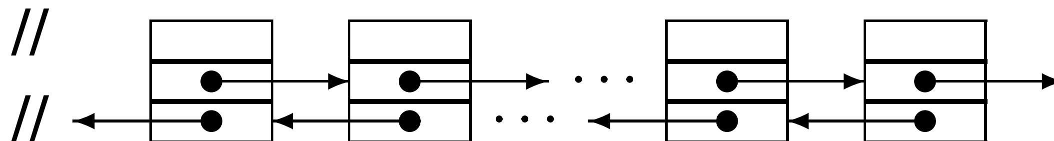
$b \rightarrow$next $:= t'$ //

$a \rightarrow$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.

Suchen

Trick: gesuchtes Element in Dummy-Item schreiben:

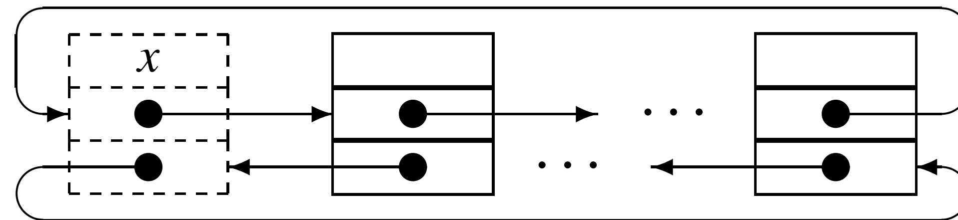
Function findNext(x : Element; from : Handle) : Handle

$h.e = x$ // 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.

$\leadsto$ einfacher, schneller,. . .

Unbeschränkte Felder (Unbounded Arrays)

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

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

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

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

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 $\mathrm{O}(m)$ ausgeführt.

Sprechweise:

pushBack und popBack haben amortisiert konstante Ausführungszeit

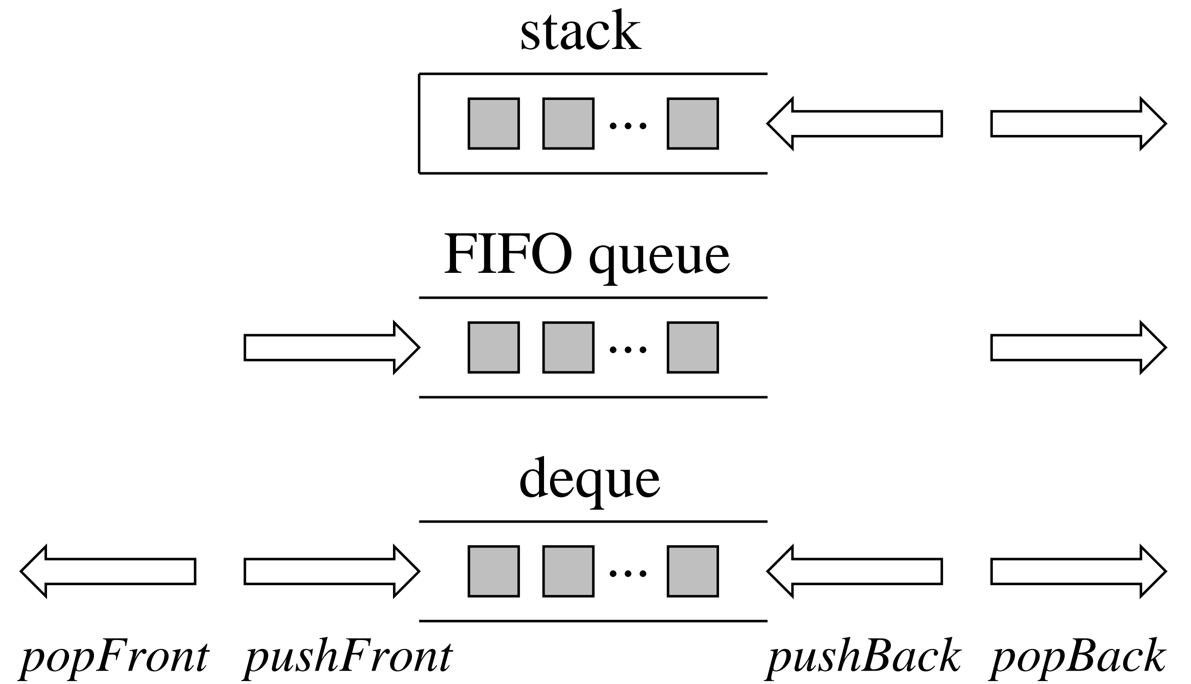
—

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

Beweis: Konto-Methode (oder Versicherung)

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

Stapel und Schlangen



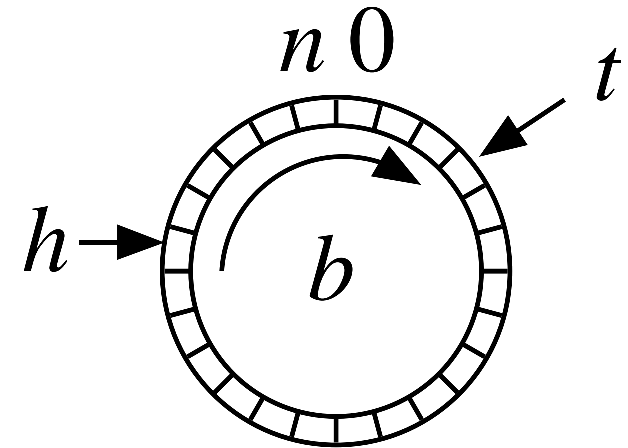
Class BoundedFIFO($n : \mathbb{N}$) **of** Element

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** $(n+1)$

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

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?

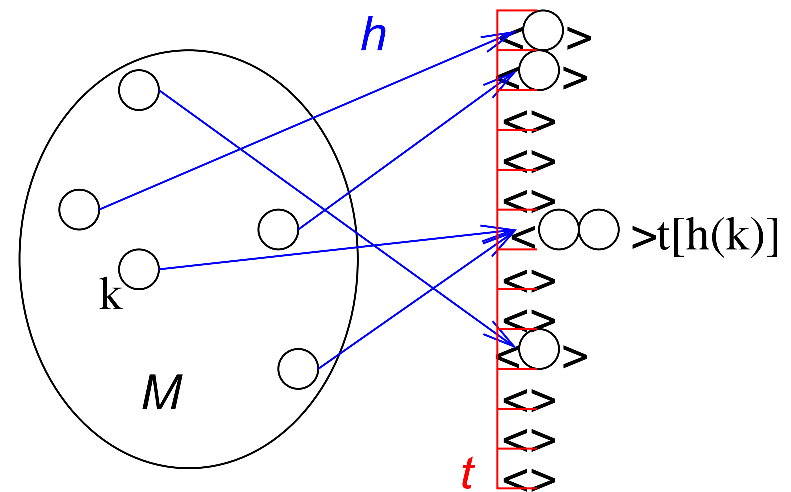
$\rightsquigarrow$ löschen und zurückliefern.

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

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

$\rightsquigarrow$ zurückliefern.

Sonst: $\perp$ zurückgeben.



Universelles Hashing

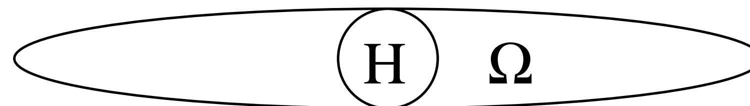
Idee: nutze nur bestimmte "einfache" Hash-Funktionen

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 1. *Für universelle Familien von Hashfunktionen ist die erwartete Listenlänge* $\mathrm{O}(1)$ *falls* $|M| = \mathrm{O}(m)$.

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 2. *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

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

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.

Remove

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

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

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

```
for (i := h(k);  k ≠ t[i];  i++ )                 // search k
    if t[i] = ⊥ then return                       // nothing to do
//we plan for a hole at i.
for (j := i+1;  t[j] ≠ ⊥;  j++ )
    //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
```

Sortieren & Co

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 '$\leq$'

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($\langle e_{\lfloor n/2 \rfloor + 1}, \ldots, e_n \rangle$))

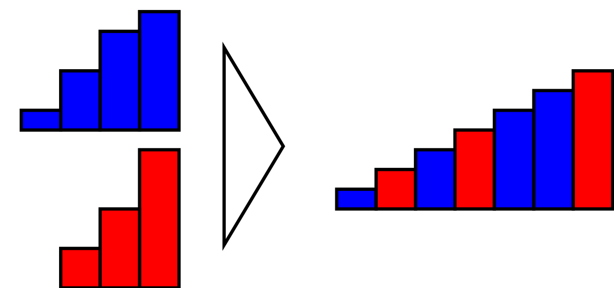
Mischen (merge)

Gegeben:

zwei sortierte Folge a und b

Berechne:

sortierte Folge der Elemente aus a und b



Eine vergleichsbasierte untere Schranke

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

Satz: Deterministische vergleichsbasierte Sortieralgorithmen brauchen

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

Vergleiche im schlechtesten Fall.

Beweis:

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

Es gibt genau $n!$ solche Permutationen...

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 Mittel und für zuf. Pivot

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

Quicksort: Effiziente Implementierung

- Array-Implementierung
- „inplace“
- generische Pivotwahl
- 2-Wegevergleiche
- größerer Basisfall
- Halbrekursive Implementierung

Quickselect

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)// [k in a]

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

if $|a| + |b| \geq k$ **then return** p // [a | $b = \langle p, \ldots, p \rangle$, k in b]

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

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

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

Beweis: hier nicht

K Schlüssel – Eimer-Sortieren (bucket sort)

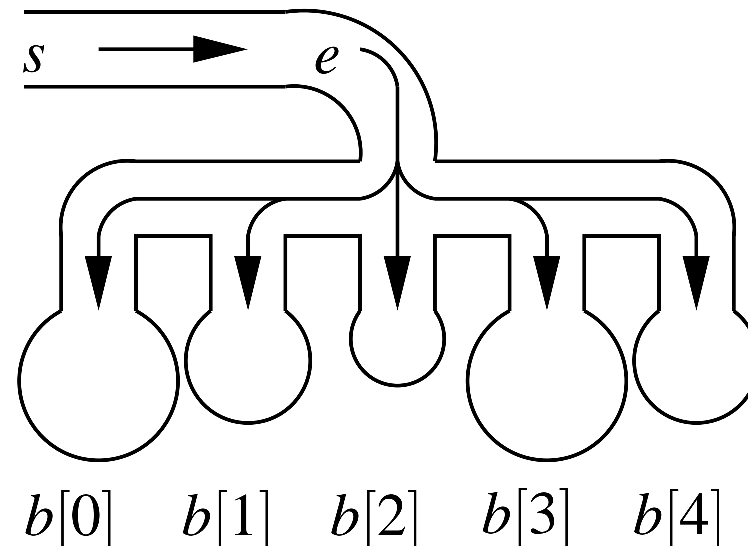
Procedure KSort(s : Sequence **of** Element)

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

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

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

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



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

$C := 1$

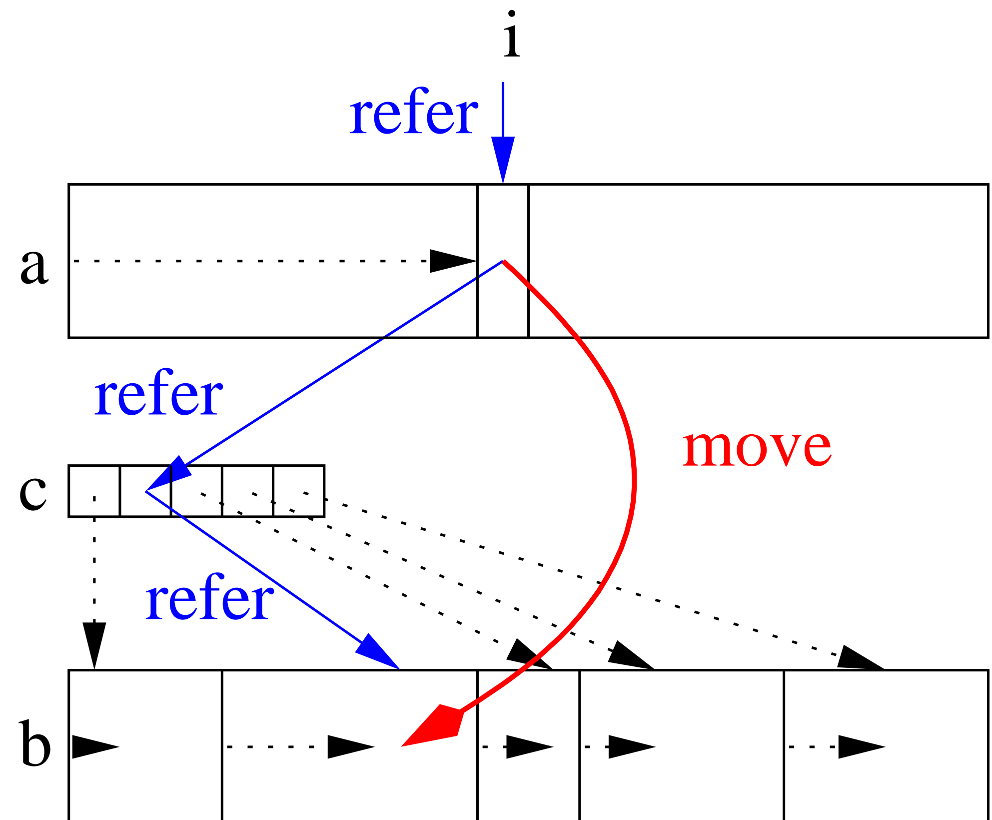
for $k := 0$ **to** $K-1$ **do**

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

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

$b[c[key(a[i])]] := a[i]$

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



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

Prioritätslisten

Prioritätslisten (priority queues)

Verwalte Menge M von Elementen mit Schlüsseln

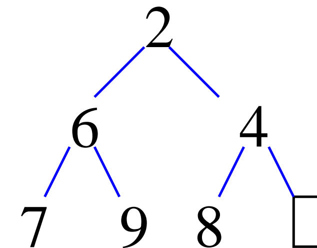
Insert(e): $M := M \cup e$

DeleteMin: return and remove $\min M$

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

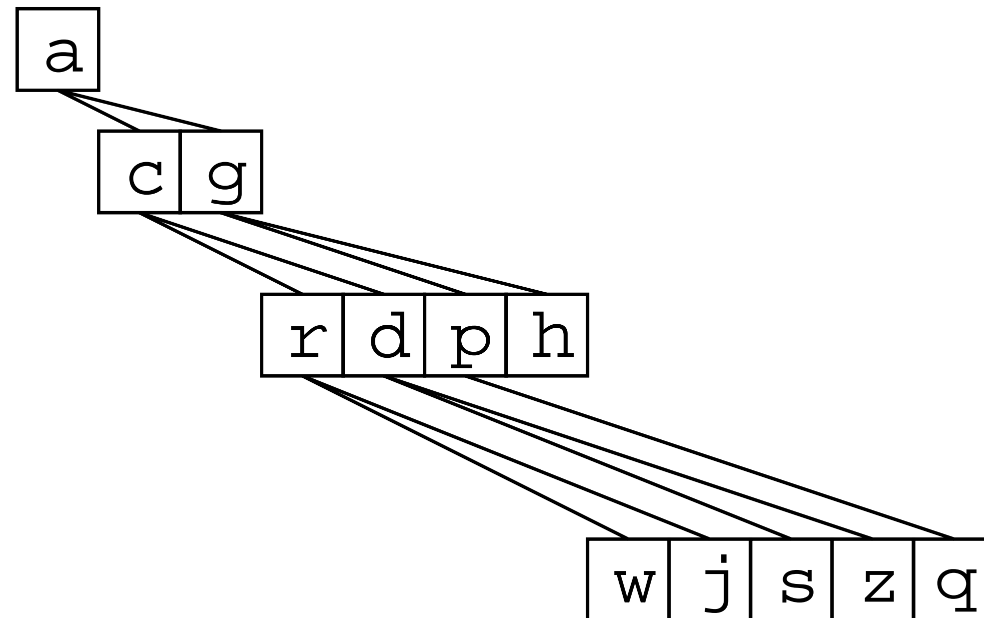
Idee: Änderungen nur entlang eines Pfades Wurzel–Blatt

$\rightsquigarrow$

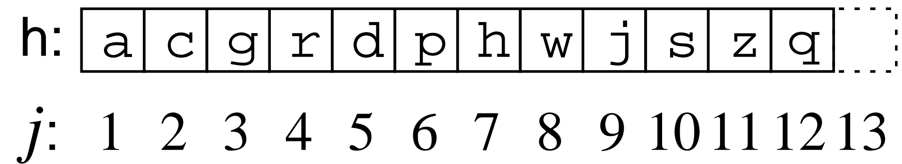
insert, deleteMin brauchen Zeit $\mathrm{O}(\log n)$

Implizite Baum-Repräsentation

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



Einfügen



Procedure insert$(e : Element)$

assert $n < w$

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

siftUp(n)

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

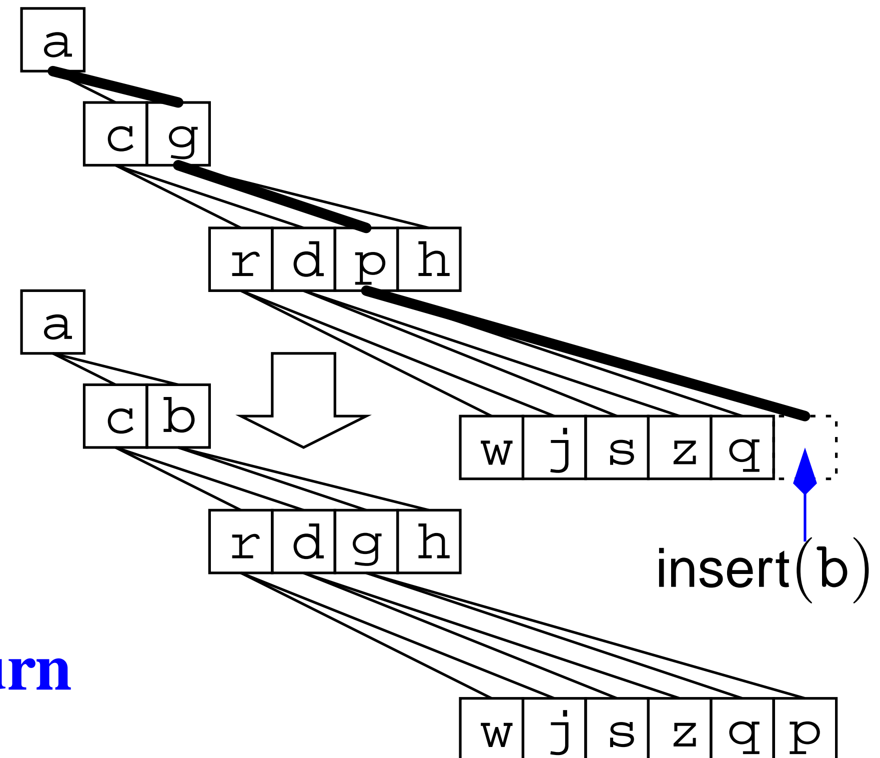
assert the heap property holds

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$(\lfloor i/2 \rfloor)$



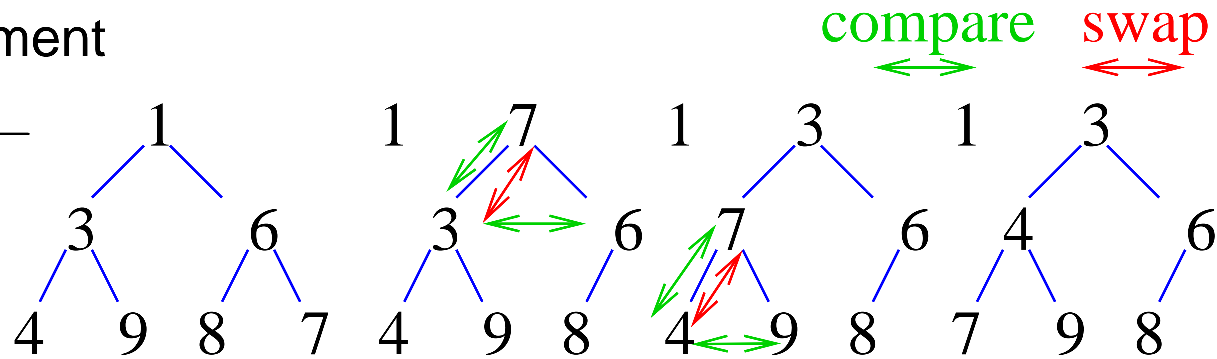
Function deleteMin : Element

result$=h[1]$: Element

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

siftDown(1)

return result



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

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

if $2i\leq n$ **then** // i is not a leaf

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

assert $\nexists$sibling$(m)\vee h[$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

Binäre Heap – Analyse

Satz: min dauert $\mathrm{O}(1)$. □

Lemma: Höhe ist $\lfloor \log n \rfloor$ □

Satz: insert dauert $\mathrm{O}(\log n)$.

Satz: deleteMin dauert $\mathrm{O}(\log n)$.

Beweis: Zeit $\mathrm{O}(1)$ pro Schicht. □

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** $\min M$

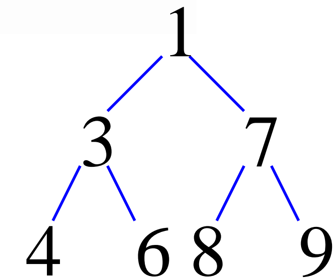
Function deleteMin $e{:=}\ \min M;$ $M{:=}\,M\setminus\{e\};$ **return** e

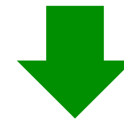
Function remove$(h:\text{Handle})$ $e{:=}\,h;$ $M{:=}\,M\setminus\{e\};$ **return** e

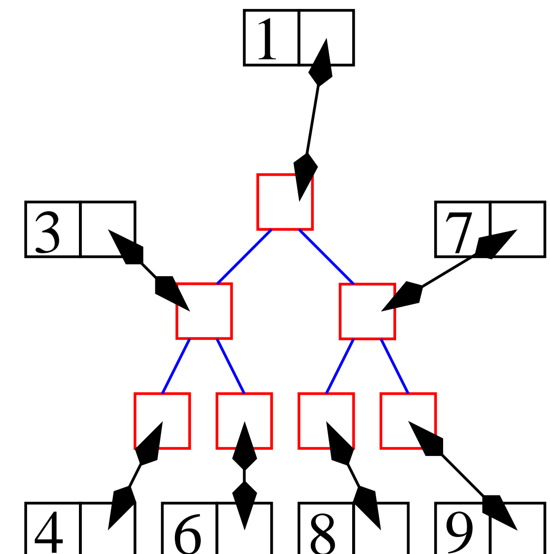
Procedure decreaseKey$(h:\text{Handle}, k:\text{Key})$ **assert** key$(h)\geq k;$ key$(h){:=}\,k$

Procedure merge(M') $M{:=}\,M\cup M'$

Adressierbare Binäre Heaps







Problem: Elemente bewegen sich.

Dadurch werden Elementverweise ungültig.

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

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

$\rightsquigarrow$ Vermittler bei jeder Vertauschung aktualisieren.

$\rightsquigarrow$ Rückverweis Element $\longrightarrow$ Vermittler

Laufzeit:

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

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 $a[r] > k$

Dynamische Sortierte Folgen – Grundoperationen

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

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

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

7.1 Binäre Suchbäume

Blätter: Elemente einer sortierten Folge.

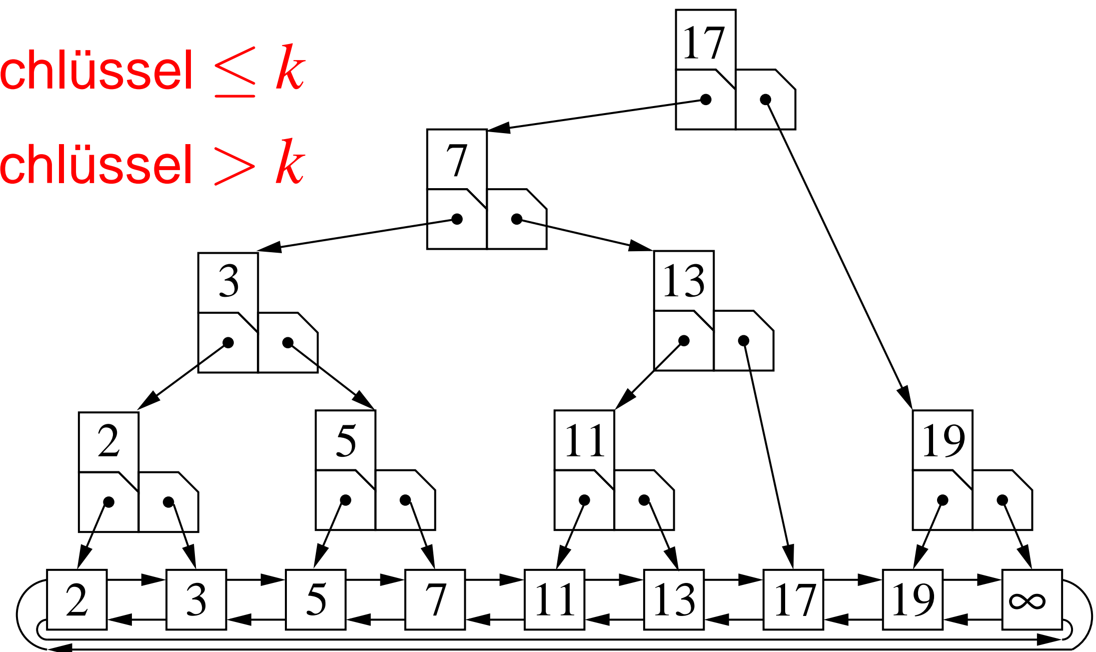
Innere Knoten $v = (k, \ell, r)$,

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

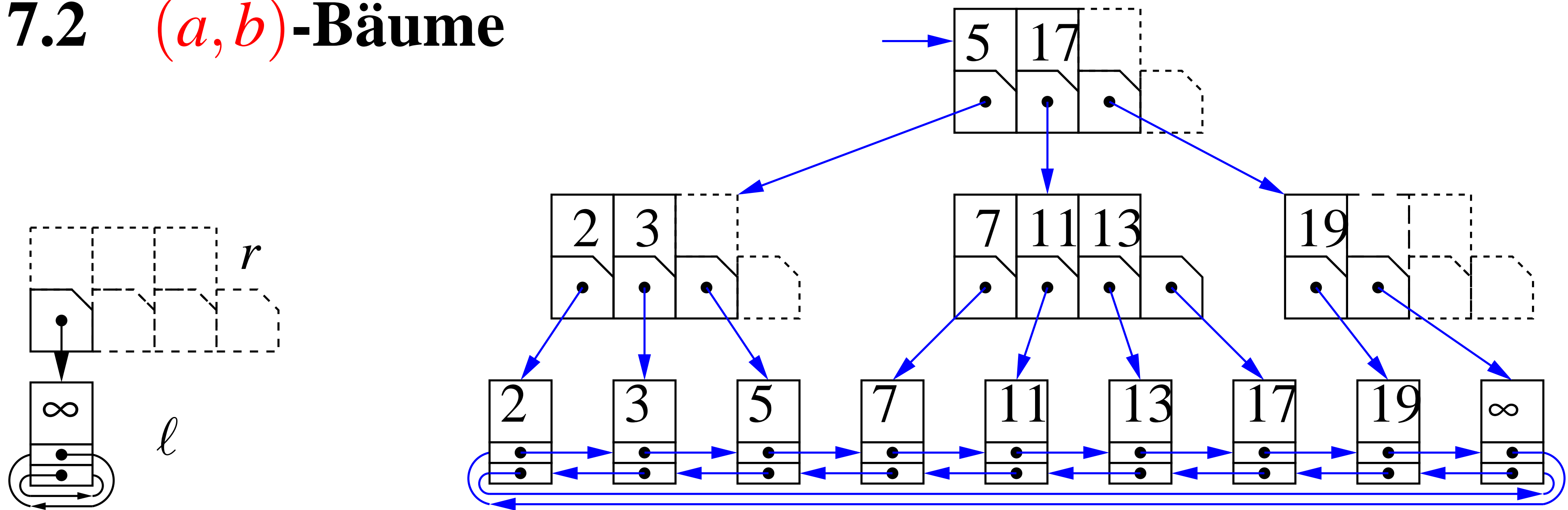
Invariante:

über ℓ erreichbare Blätter haben Schlüssel $\leq k$

über r erreichbare Blätter haben Schlüssel $> k$



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 $\langle\rangle$)

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**

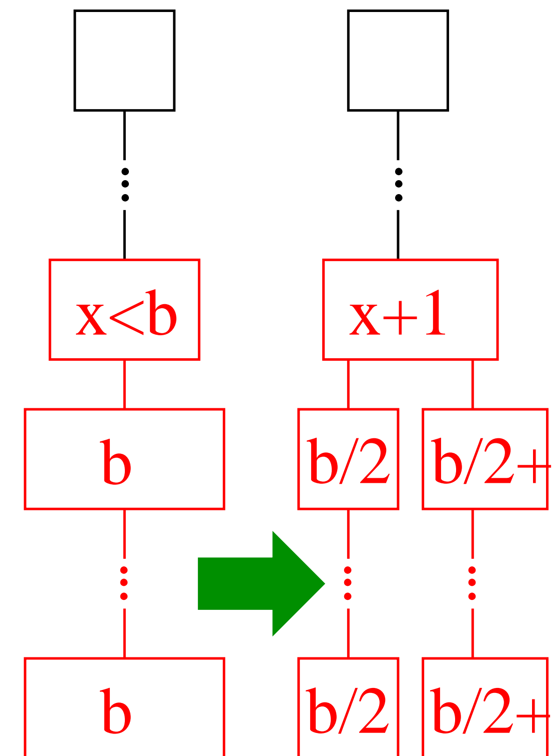
spalte u in 2 Knoten mit Graden

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

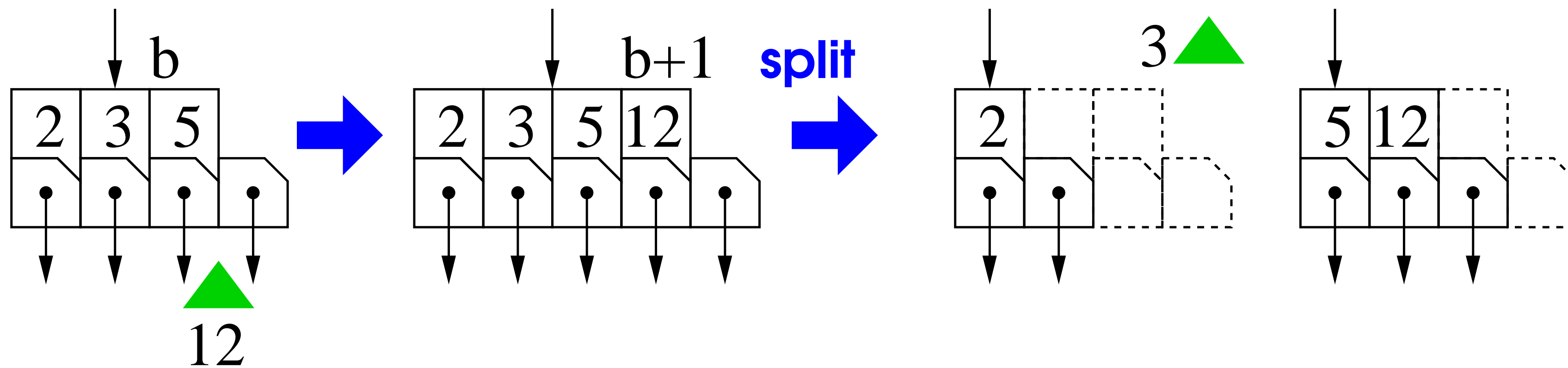
Weiter oben einfügen, spalten

…

ggf. neue Wurzel



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

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)

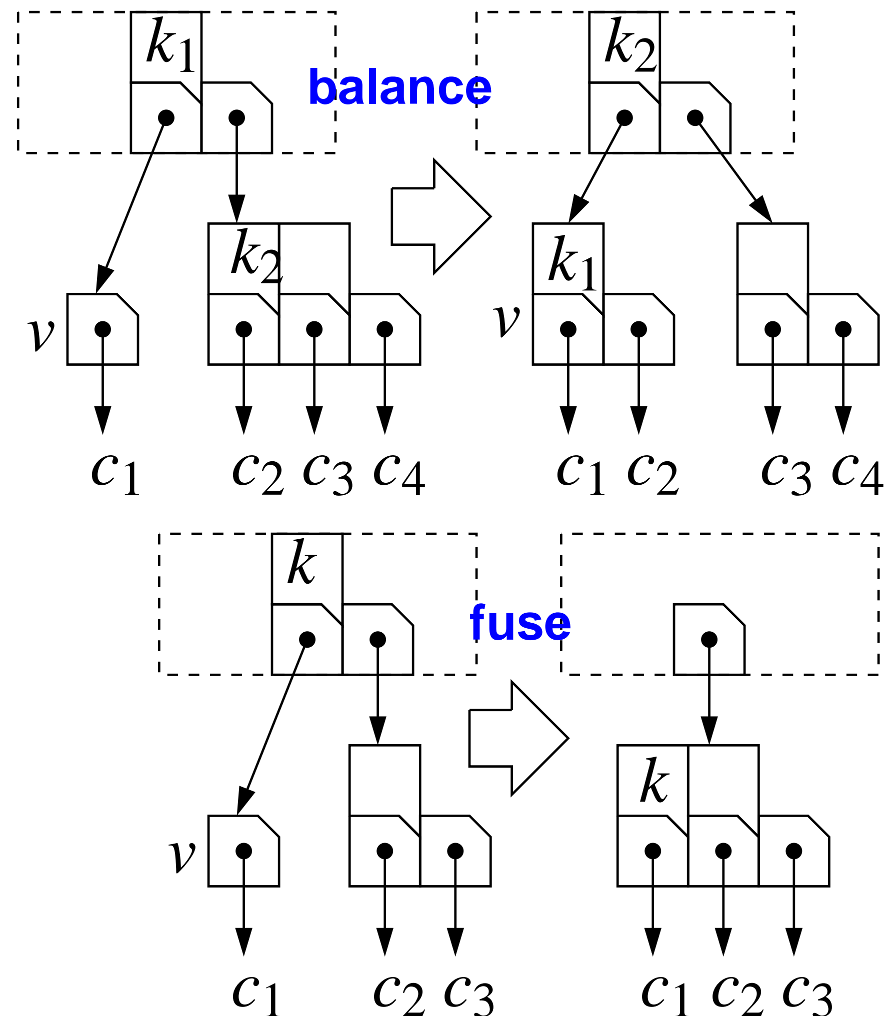
else

fuse(u', u)

Weiter oben splitter entfernen

…

ggf. Wurzel entfernen



(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 $r.d = 1 \wedge$ height > 1 **then** $r' := r$; $r := r'.c[1]$; **dispose** r'

Procedure ABItem::removeRec(k : Key, h : $\mathbb{N}$, ℓ : List **of** Element)
  $i :=$ locateLocally(k)
  if $h = 1$ **then**
    if key($c[i] \to e$) $= k$ **then**
      ℓ.remove($c[i]$)
      removeLocally(i)
  else
    $c[i] \to$ removeRec($e, h-1, \ell$)
    if $c[i] \to d < a$ **then**
      if $i = d$ **then** $i{-}{-}$
      $s' :=$ concatenate($c[i] \to s, \langle s[i] \rangle, c[i+1] \to s$))
      $c' :=$ concatenate($c[i] \to c, c[i+1] \to c$)
      $d' := |c'|$
      if $d' \leq b$ **then** // fuse
        $(c[i+1] \to s, c[i+1] \to c, c[i+1] \to d) := (s', c', d')$
        dispose $c[i]$; removeLocally(i)
      else // balance
        $m := \lceil d'/2 \rceil$
        $(c[i] \to s, c[i] \to c, c[i] \to d) := (s'[1..m-1], c'[1..m], m)$
        $(c[i+1] \to s, c[i+1] \to c, c[i+1] \to d) :=$
          $(s'[m+1..d'-1], c'[m+1..d'], d'-m)$
        $s[i] := s'[m]$

Procedure ABItem::removeLocally(i : $\mathbb{N}$)
  $c[i..d-1] := c[i+1..d]$
  $s[i..d-2] := s[i+1..d-1]$
  $d{-}{-}$

Zusammenfassung

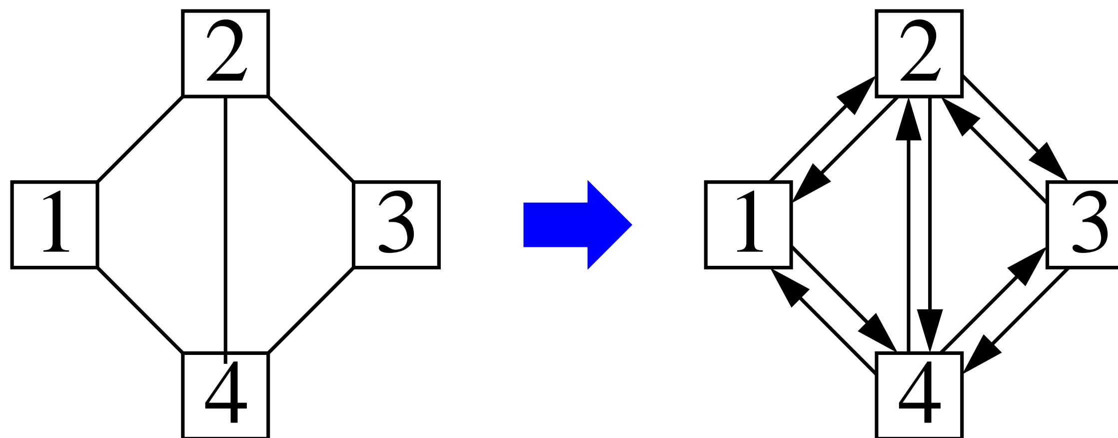
- Suchbäume erlauben viele effiziente Operationen auf sortierten Folgen.
- Oft logarithmische Ausführungszeit
- Der schwierige Teil: logarithmische Tiefe erzwingen.
- Augmentierungen $\rightsquigarrow$ zusätzliche Operationen

Ungerichtete → gerichtete Graphen

Meist repräsentieren wir

ungerichtete Graphen durch bigerichtete Graphen

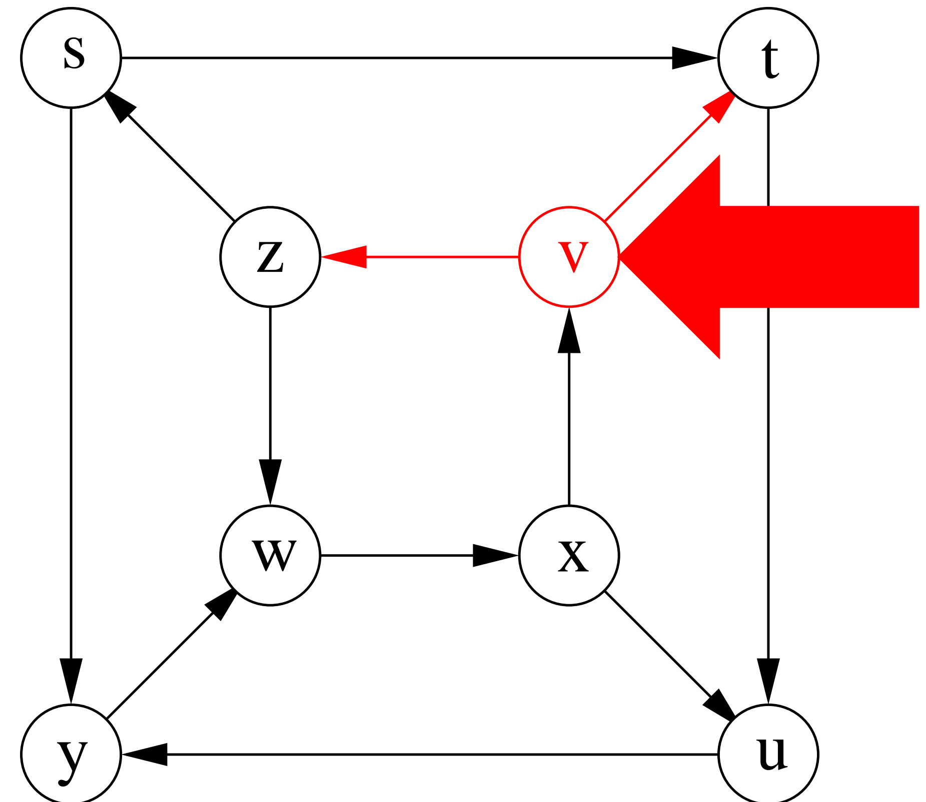
$\rightsquigarrow$ wir konzentrieren uns auf gerichtete Graphen



Operationen

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

Grundoperationen



Statische Graphen

Konstruktion, Konversion und Ausgabe

($O(m+n)$ Zeit)

Navigation: Gegeben v,

finde ausgehende Kanten.

Dynamische Graphen

Knoten/Kanten einfügen/löschen

Adjazenzfelder

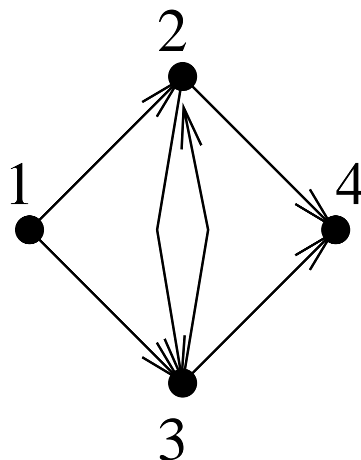
$V = 1..n$ oder $0..n-1$

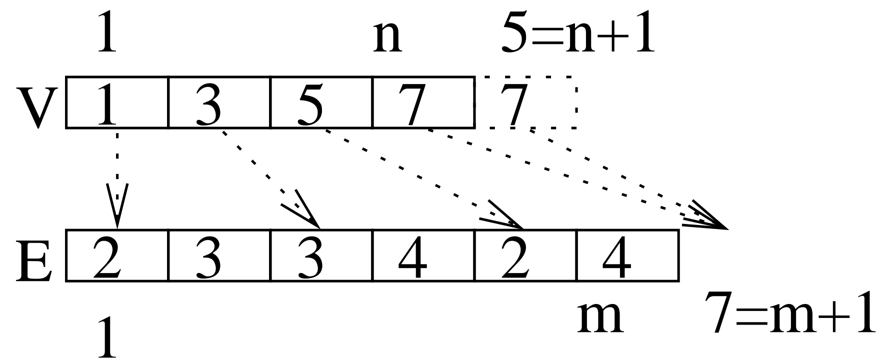
Kantenfeld E speichert Ziele

gruppiert nach Startknoten

V speichert Index der ersten ausgehenden Kante

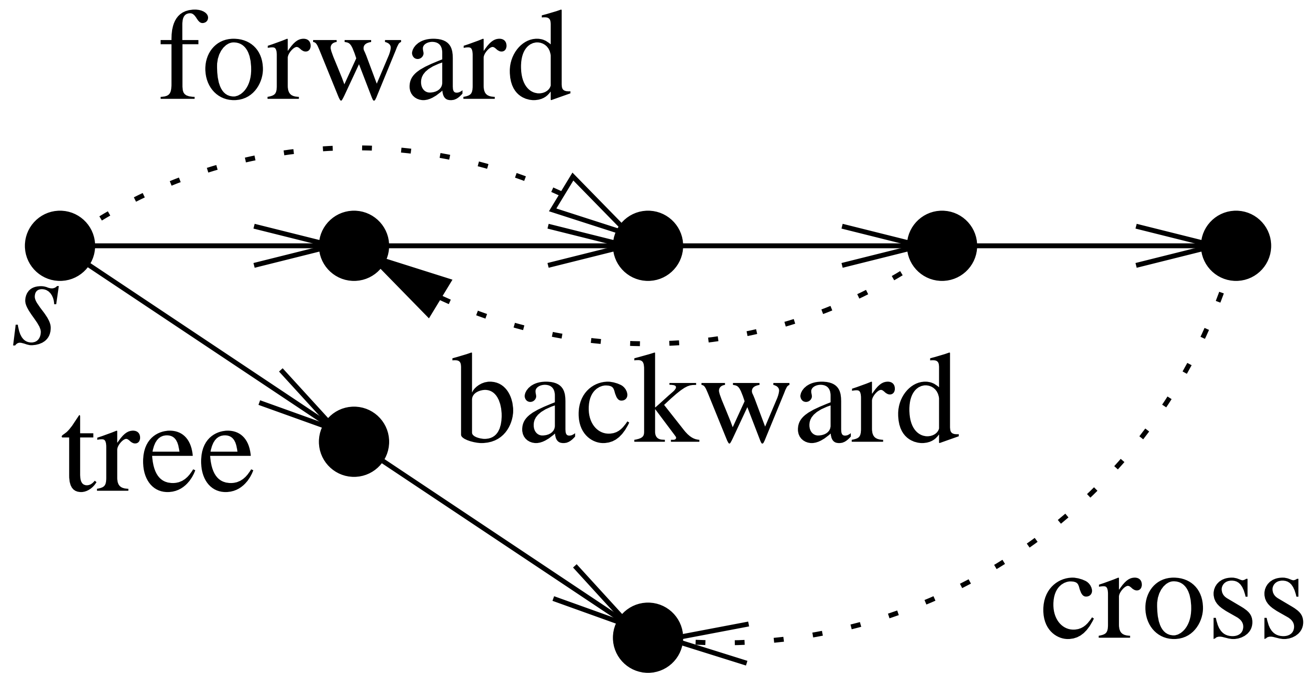
Dummy-Eintrag $V[n+1]$ speichert $m+1$





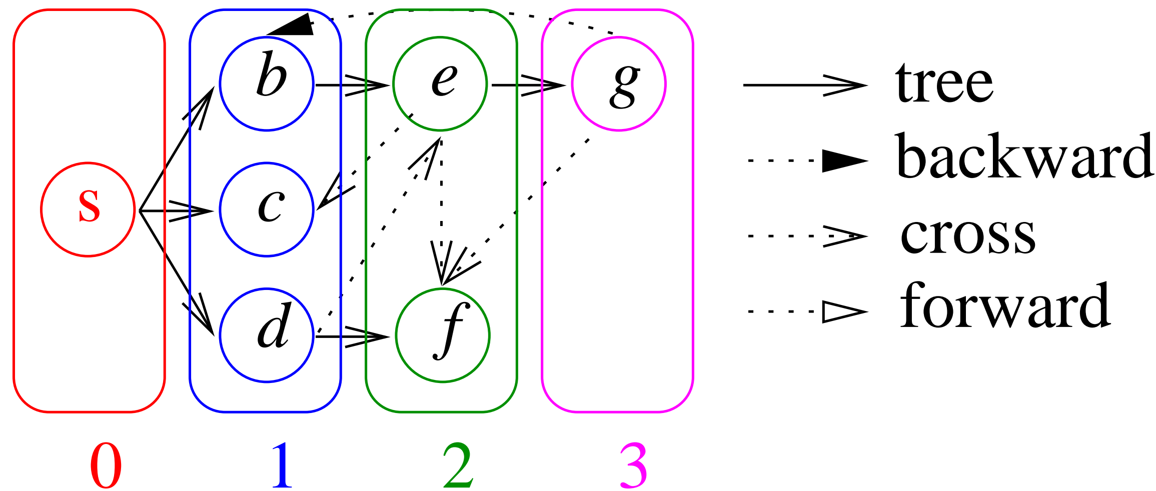
Beispiel: Ausgangsgrad$(v) = \mathtt{V}[v+1] - \mathtt{V}[v]$

Graphtraversierung als Kantenklassifizierung



Breitensuche

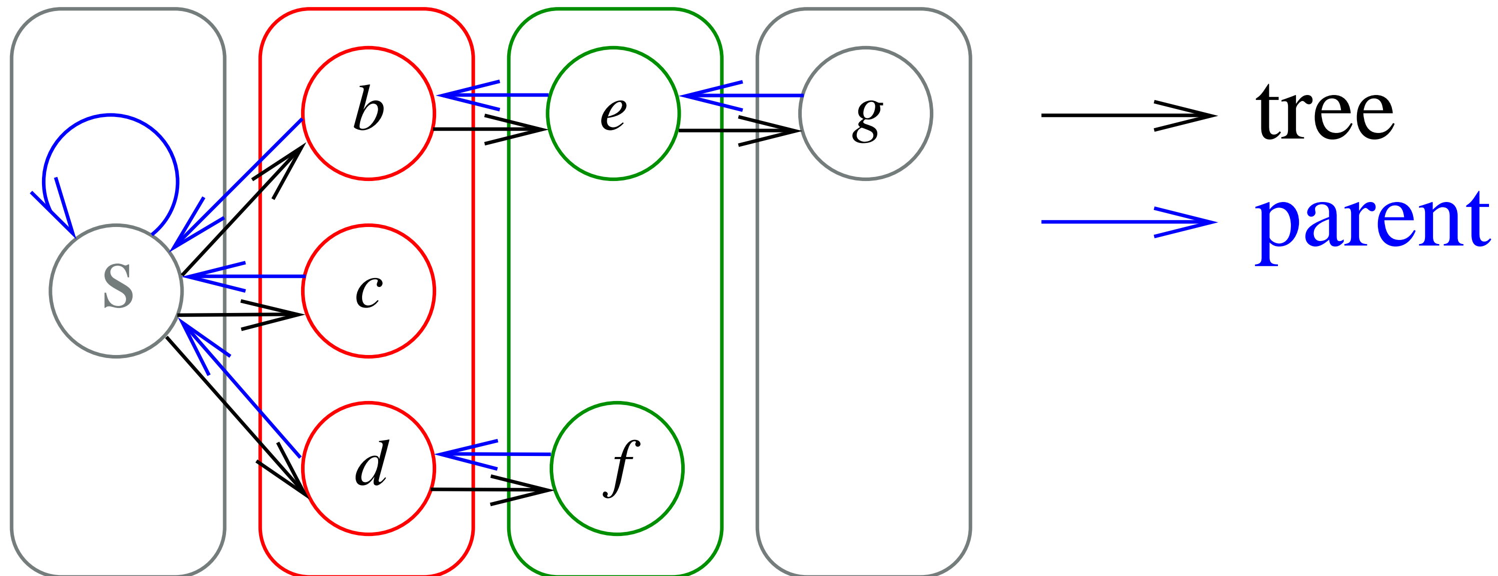
- Einfachste Form des kürzeste Wege Problems
- Umgebung eines Knotens definieren
  (ggf. begrenzte Suchtiefe)
- Einfache, effiziente Graphtraversierung
  (auch wenn Reihenfolge egal)



Repräsentation des Baums

Feld parent speichert Vorgänger.

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



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
```

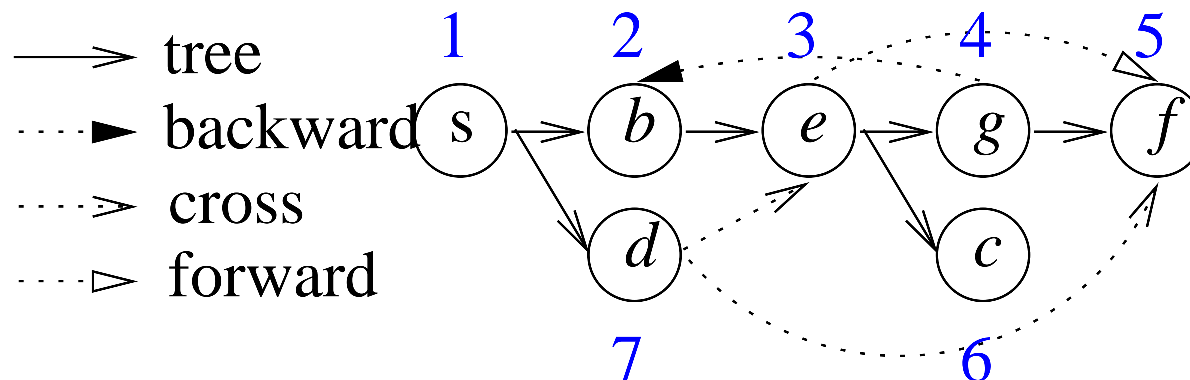
DFS Nummerierung

init:	dfsPos$=1$ $: 1..n$
root(s):	dfsNum$[s]$:= dfsPos$++$
traverseTreeEdge(v,w):	dfsNum$[w]$:= dfsPos$++$

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

Beobachtung:

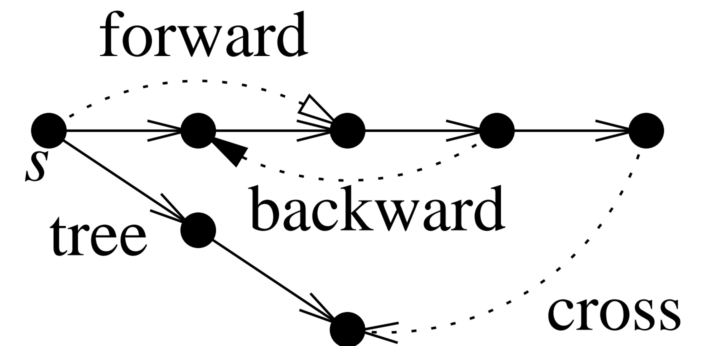
Knoten auf dem Rekursionsstapel sind bzgl., $\prec$ sortiert



BFS $\longleftrightarrow$ 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 $v\}$

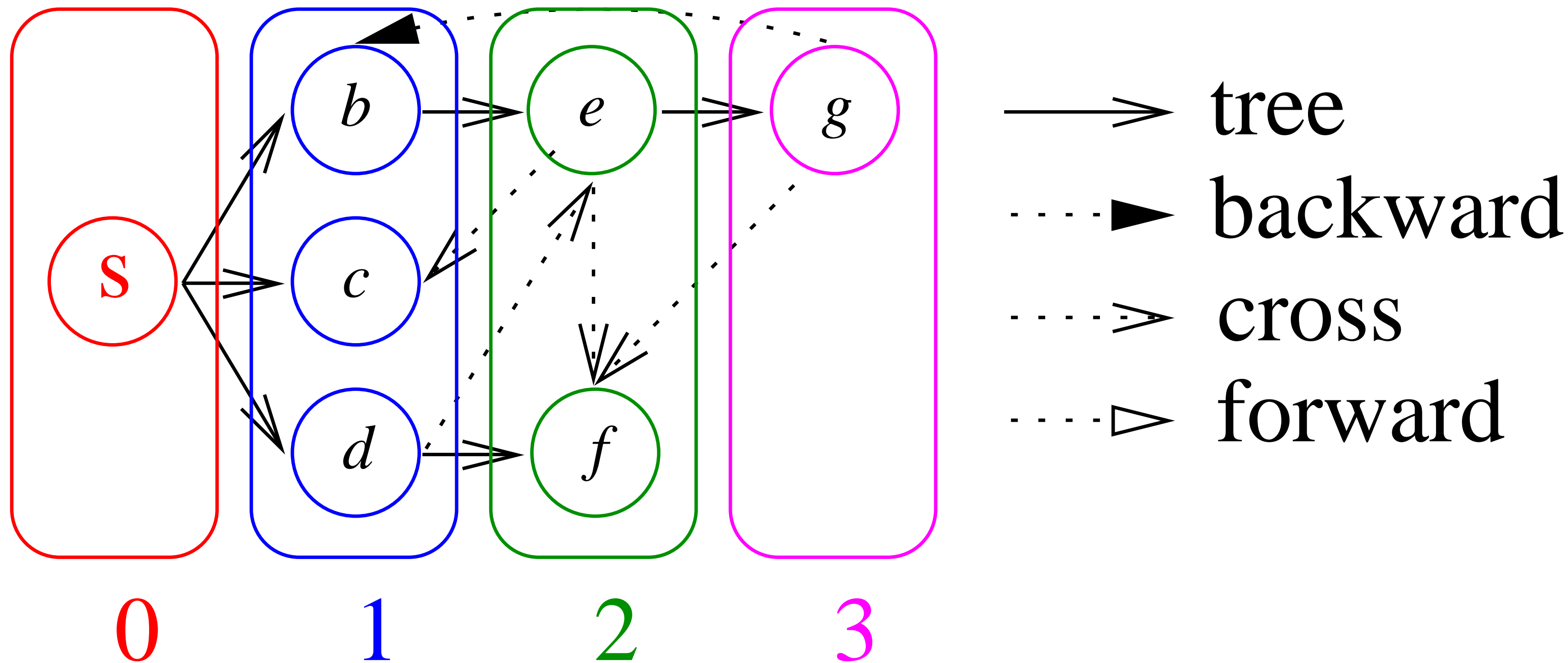
mit $c(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} c(e_i)$.

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

10.3 Kantengewichte ≥ 0

Alle Gewichte gleich:

Breitensuche (BFS)!



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

Laufzeit

(Noch) besser mit Fibonacci-Heapprioritätslisten:

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

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

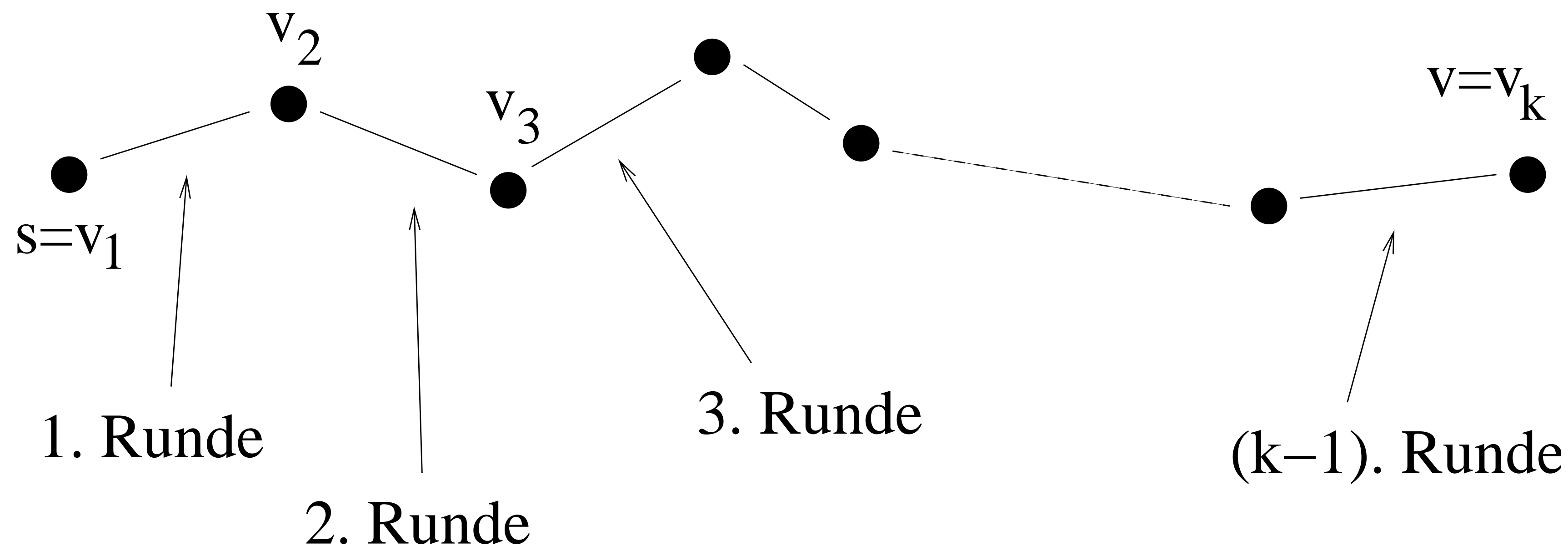
Aber: konstante Faktoren in $\mathrm{O}(\cdot)$ sind hier größer!

Algorithmen Brutal – Bellman-Ford-Algorithmus für beliebige Kantengewichte

Wir relaxieren alle Kanten (in irgendeiner Reihenfolge) $n-1$ mal

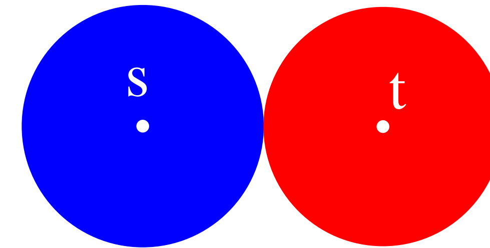
Alle kürzeste Pfade in G haben höchstens $n-1$ Kanten

Jeder kürzeste Pfad ist eine Teilfolge dieser Relaxierungen!

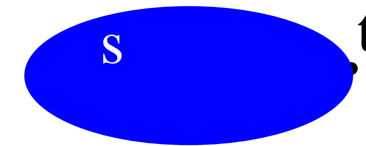


Ideen für Routenplanung

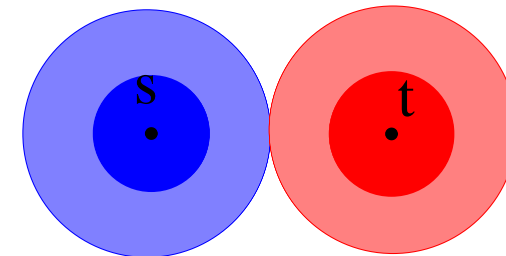
mehr in Algorithmen II, Algorithm Engineering



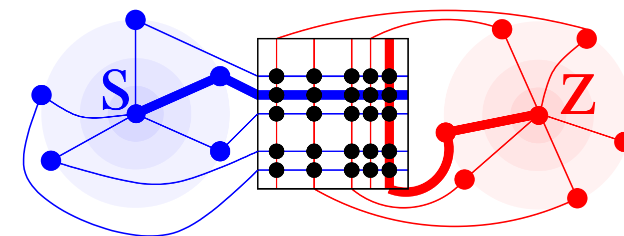
- Vorwärts + Rückwärtssuche



- Zielgerichtete Suche

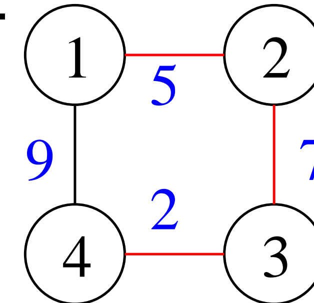


- Hierarchien ausnutzen



- Teilabschnitte tabellieren

Minimale Spannbäume (MSTs)



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

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

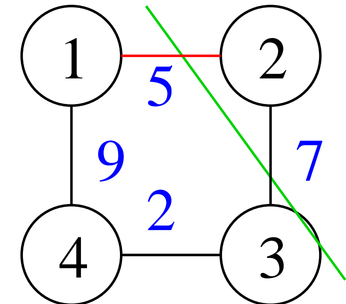
Kanten $e \in E$, $m = |E|$, two-element subsets of V.

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

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

11.1 MST-Kanten auswählen und verwerfen

Die Schnitteigenschaft (Cut Property)



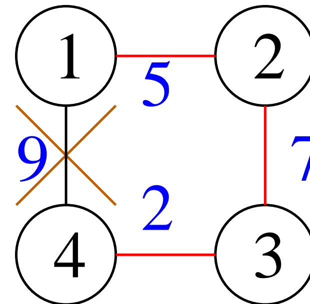
Für beliebige Teilmenge $S \subset V$ betrachte die Schnittkanten

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

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

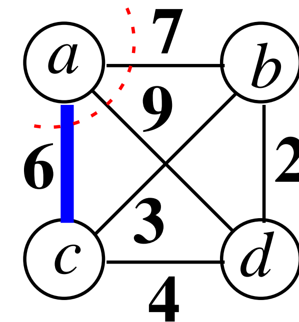
Die Kreiseigenschaft (Cycle Property)

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

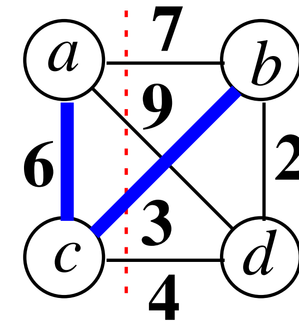


11.2 Der Jarník-Prim Algorithmus

[Jarník 1930, Prim 1957]



Idee: Lasse einen Baum wachsen



$T := \emptyset$

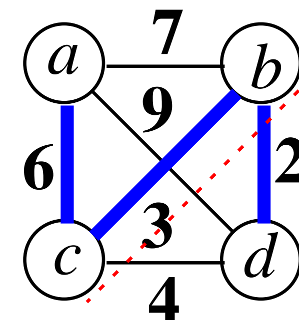
$S := \{s\}$ for arbitrary start node s

repeat $n-1$ times

find (u,v) fulfilling the cut property for S

$S := S \cup \{v\}$

$T := T \cup \{(u,v)\}$



11.3 Kruskals Algorithmus [1956]

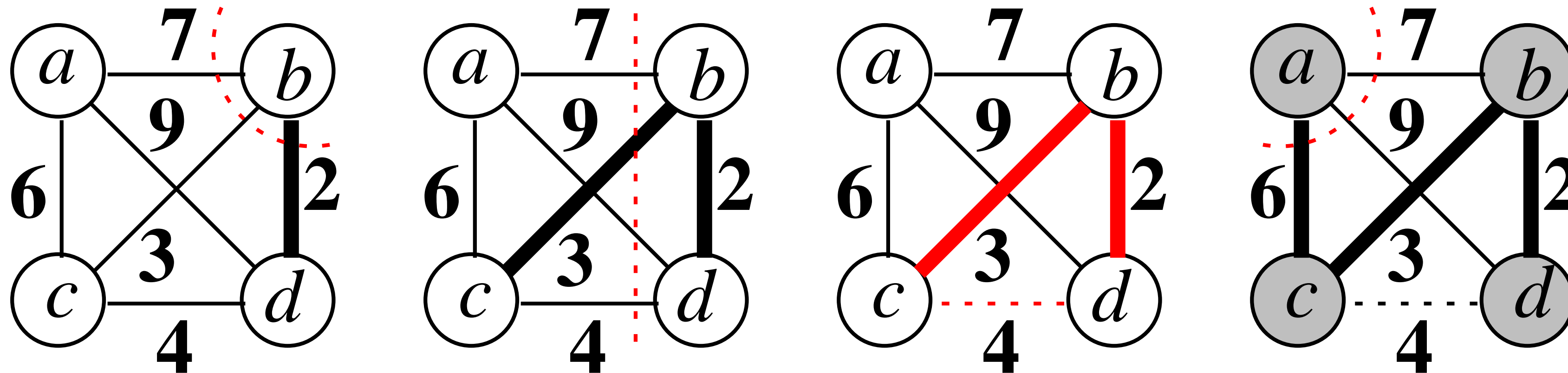
$T := \emptyset$ // subforest of the MST

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

  if u and v are in different subtrees of (V, T) **then**

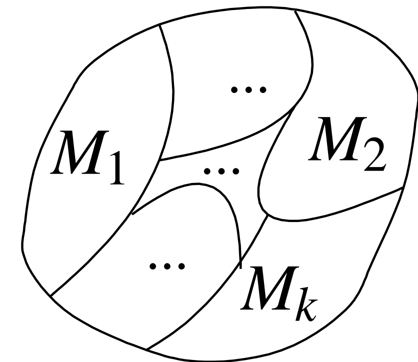
    $T := T \cup \{(u, v)\}$ // Join two subtrees

return T



11.4 Union-Find Datenstruktur

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



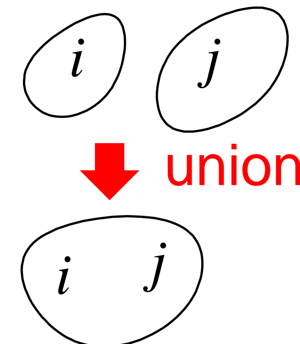
$$M_1 \cup \cdots \cup M_k = 1..n,$$

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

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.

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

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

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

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

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

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

Anwendungsspez. Suchraumbeschränkungen: Schnittebenen (ILP),
Lemma-Generierung (Logik),...

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.