

Algorithmen und Datenstrukturen I

Christian Schulz

Übungen:

Ernestine Grossmann

Tutoren:

**Julius Arnold, Tillmann Fehrenbach,
Anna-Sophie Sawitzki, Patrick Steil**

Algorithm Engineering Group

Web:

https://ae.ifi.uni-heidelberg.de/adsI_ss22.html

Organisatorisches

Die Vorlesung findet als in Präsenz statt.

Vorlesungen:

Mo: 09:30–11:00 Vorlesung

Di: 14:15–15:00 oder 15:45 Vorlesung,

Di: 15:00–15:45 Fragestunde und Beispiele (nach Bedarf)

Organisatorisches

Übungen:

Mi 16-18 SR 1 (Anna-Sophie) und SR 2 (Patrick)

Do 9-11 SR 5 (Julius) und SR 6 (Tillmann)

Do 11-13 SR 5 (Julius) und SR 6 (Tillmann)

Fr 11-13 SR 1 (Patrick)

Fr 14-16 Uhr SR 1 (Sophie)

Einteilung mittels Müsli (**Deadline** diesen Freitag!)

[https://muesli.mathi.uni-heidelberg.de/
lecture/view/1528](https://muesli.mathi.uni-heidelberg.de/lecture/view/1528)

Übungsblätter: wöchentlich

Ausgabe Montag, Abgabe Freitag (11 Tage nach Ausgabe)

Materialien

☐ Folien^a, Übungsblätter, Videos (z.Z. \geq Thema 1)

☐ Diskussionsforum

☐ Buch:

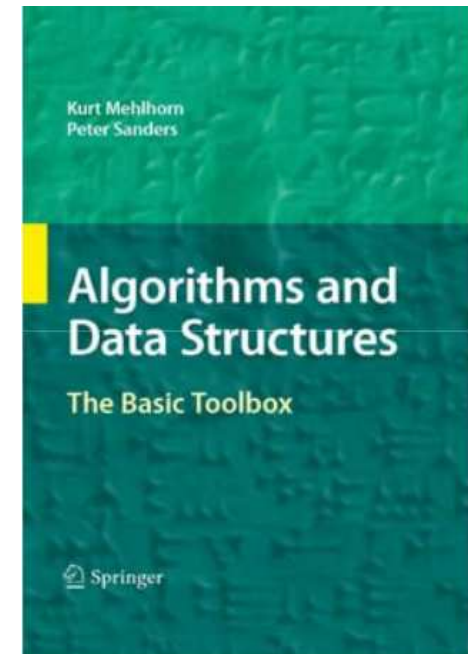
K. Mehlhorn, P. Sanders

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)



^agrößtenteils von P. Sanders

Übungen

- ☐ “Normale” Übungsaufgaben
- ☐ freiwilligen Programmieraufgaben (ohne Abgabe)
 ~> Hackerrank, Coding Competitions
- ☐ zwei Pflichtprogrammieraufgaben
- ☐ 50% der Übungspunkte, und 25% bei jedem
 Pflichtprogrammierblatt nötig um Klausur zu schreiben
- ☐ Gruppenabgaben bis zu 3 Personen pro Gruppe
 1. Gruppen können nicht gewechselt werden
 2. Reden Sie nach der VL, oder suchen Sie im Frageforum
- ☐ Besten zwei Gruppen erhalten jeweils eine Algorithm Engineering
 Tasse pro Person

Deutschsprachige Bücher

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

Algorithmus? Kann man das essen?

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

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

Das war damals “Elite” –

Machtzentrum des arabischen Kalifats auf seinem Höhepunkt.

Er hat ein Rechenlehrbuch geschrieben.

⇒ Algorithmus wurde zum Synonym für Rechenvorschrift.

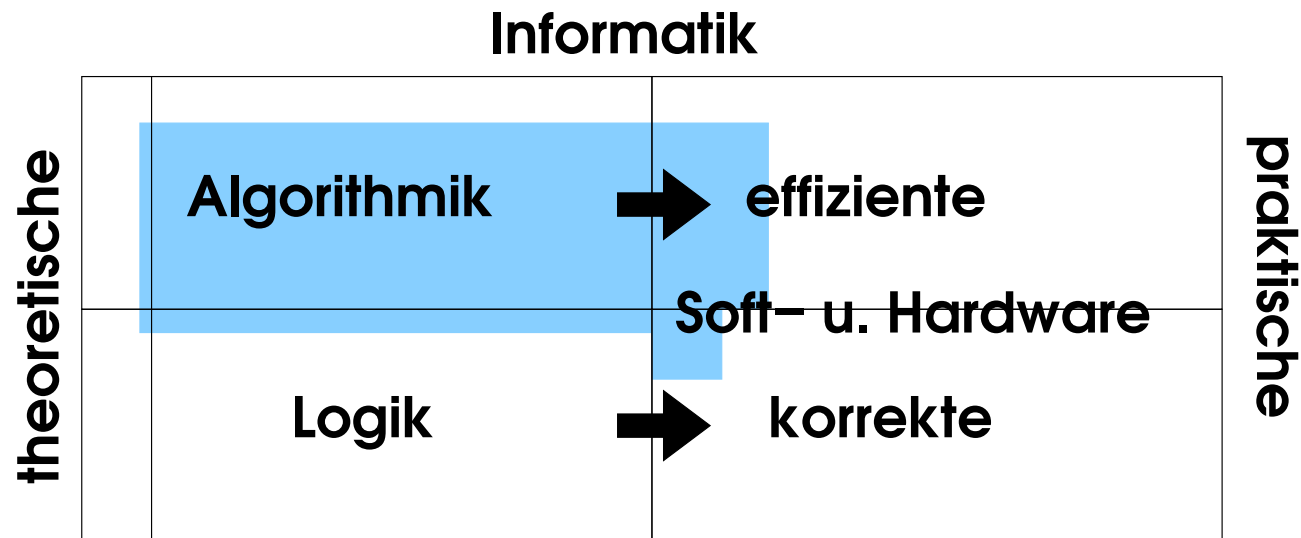


Moderne Definition (Wikipedia):

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



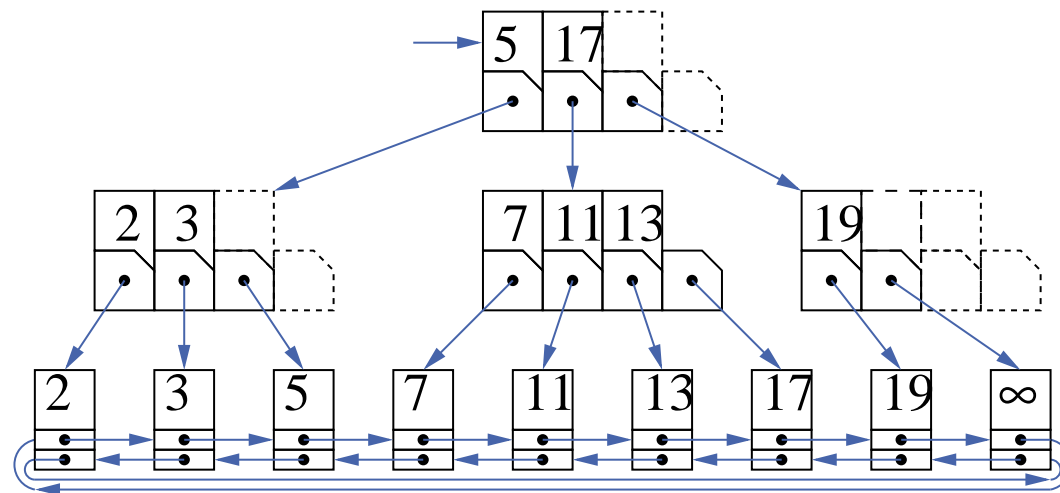
Datenstruktur

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 \rightsquigarrow neue Algorithmen
- ☐ Analysetechniken \rightsquigarrow Leistungsgarantien, objektiver
Algorithmenvergleich

Jeder Informatiker braucht das \rightsquigarrow Pflichtvorlesung

Inhaltsübersicht

0. Beweise	101
1. Amuse Geule	Appetithäppchen
2. Einführung	der Werkzeugkasten für den Werkzeugkasten
3. Folgen, Felder, Listen	Mütter und Väter aller Datenstrukturen
4. Hashing	Chaos als Ordnungsprinzip
5. Sortieren	Effizienz durch Ordnung
6. Prioritätslisten	immer die Übersicht behalten
7. Sortierte Liste	die eierlegende Wollmilchsau
8. Graphrepräsentation	Beziehungen im Griff haben
9. Graphtraversierung	globalen Dingen auf der Spur
10. Kürzeste Wege	schnellstens zum Ziel
11. Minimale Spannbäume	immer gut verbunden
12. Optimierung	noch mehr Entwurfsmethoden

0 Beweise

Observation

□ Viele Studenten haben Probleme einfache Aussagen zu beweisen

➡ Goal: Learn it (again)

Achtung: das ist nicht zu verstehen als full-fledged Repetitorium in Mathematik. We konzentrieren uns hier auf Beweisstrategien und praktische Aspekte.

Links etc..

Folien von Uwe Egly (Englisch)

Diese Folien basieren auf:

A. Wohlgemuth:

Introduction to Proof in Abstract Mathematics. Dover 2011.

Deductive Mathematics—an introduction to proof and discovery for
mathematics education

Download:

<http://andrew-wohlgemuth.com/DMmathed.pdf>

Statements

- Statement / Aussagen:

ein mathematischer Ausdruck der entweder wahr oder falsch ist.

- Beispiele: $2 \in \{x \in \mathbb{R} \mid x < 5\}$ (wahr) oder $3^2 + 5^2 = 8^2$ (falsch)

- Ausdrücke wie $0 < x < 1$ werden verwendet um Mengen zu definieren

$$A = \{x \in \mathbb{R} \mid 0 < x < 1\}$$

- Wichtig: Wahrheitswert eines **offenen** Ausdrucks $0 < x < 1$ hängt von gewähltem x ab. Bsp. wahr für $x = \frac{1}{2}$ und falsch für $x = 5$

- Wichtig: Die **Domäne**

Für \mathbb{N} , gibt es kein x s.t. $0 < x < 1$, aber es gibt welche für \mathbb{R}

Formal mathematical proofs

- ☐ Ein **formaler mathematischer Beweis** besteht aus einer nummerierten Sequenz von **wahren Aussagen**
 - ☐ Jede Aussage in einem Beweis ist eine **Annahme** oder ...
 - ☐ ... **folgt** aus vorherigen Aussagen durch eine Ableitungsregel/Inferenzregel (rule of inference)
 - ☐ Die letzte Aussage ist die die wir bewiesen haben.
- ➡ Offene Aussagen können in Beweisen nicht auftreten!

Beispiel einer Inferenzregel: set definition rule

Wenn ein Element in einer Menge ist, dann können wir die definierende Eigenschaft ableiten. Andererseits, wenn es die definierende Eigenschaft erfüllt, dann können wir ableiten das das Element in der Menge ist.

Set definition rule: Beispiel

Definiere $C = \{x \in \mathbb{R} \mid x < 2\}$

($x < 2 \wedge x \in \mathbb{R}$ ist die definierende Eigenschaft)

Zwei Möglichkeiten für Ableitungen

Möglichkeit 1

$$1. \quad a \in C$$

$$2. \quad a < 2 \wedge a \in \mathbb{R} \quad (1; \text{def } C)$$

Möglichkeit 2

$$1. \quad b < 2 \wedge b \in \mathbb{R}$$

$$2. \quad b \in C \quad (1; \text{def } C)$$

□ Jede Aussage in dem Beweis hat eine Nummer

□ Wir begründen wie wir eine Aussage ableiten, z.B. (1; def C)
bedeutet wir leiten die aktuelle Aussage aus Aussage 1 mit der Definition von C und der set definition rule ab.

Bemerkung: $\wedge b \in \mathbb{R}$ wird oft ausgelassen, wenn Kontext es zulässt.

Macro-steps in proofs

Problem: Schauen wir uns folgenden Beweis an:

(ass = Annahme (assumption) und prop = Eigenschaft (property))

Annahme:	1.	$X = \{x \in \mathbb{R} \mid x < 1\}$	
	2.	$a \in X$	
Zeige:		$a < 2$	
1.	$a \in X$	(ass 2)	
2.	$a < 1$	(1, ass 1; def X)	
3.	$1 < 2$	(prop \mathbb{R})	
4.	$a < 2$	(2, 3; prop \mathbb{R})	

Ist das ein akzeptabler Beweis? Akzeptanz von Makro-Schritten wie “prop \mathbb{R} ”

hängt von der Zielgruppe ab! Welche Eigenschaft von \mathbb{R} wurde benutzt?

Übersicht

- ☐ Beweisen und verwenden von Für Alle Aussagen
- ☐ Beweisen und verwenden von Oder Aussagen
- ☐ Beweisen und verwenden von Und Aussagen
- ☐ Theorems verwenden
- ☐ Beweisen und verwenden von Implikationen
- ☐ Law of Exluded Middle
- ☐ Äquivalenzen und iff Aussagen
- ☐ Beweis durch Widerspruch
- ☐ Existenzaussagen

Einfache Beweistechniken

Beweis durch Beispiel

Beispiel: Zeigen Sie es gibt eine Primzahl zwischen 80 und 90.

Idee: Zeugen angeben für die Primzahl (p) für die die Aussage gilt.

Beweis: Wähle $p = 83$.

Ist das ausreichend?

Eigentlich, **NEIN**. Wir müssen noch **zeigen** das 83 tatsächlich eine Primzahl ist.

Das können wir tun in dem wir alle Teiler ausprobieren.

Einfache Beweistechniken

Widerlegen von Behauptungen

Behauptung: Nimm an n ist eine Primzahl größer als 1. Dann ist $2^n - 1$ ebenfalls eine Primzahl.

Können Sie die Behauptung beweisen? Try hard ...

Wenn Sie es nicht können, dann sollten Sie drüber nachdenken die Behauptung zu widerlegen. Eine Primzahl n für die $2^n - 1$ nicht prim ist, ist genug!

Das Gegenbeispiel ist $n = 11$ da 11 prim ist, aber

$$2^{11} - 1 = 2047 = 23 \cdot 89$$

keine Primzahl ist!

∀ Aussagen beweisen

Inferenzregel für definierte Beziehungen

The definition rule

Angenommen, es wurde eine Beziehung definiert. Wenn die Beziehung gilt (in irgendeinem Beweisschritt oder Annahme), dann kann die definierende Eigenschaft abgeleitet werden. Andererseits, wenn die definierende Eigenschaft gilt, dann kann die Beziehung abgeleitet werden.

\forall Aussagen beweisen

Inferenzregel für definierte Beziehungen

Beispiel: Für Mengen A und B , definiere A ist **Teilmenge** von B , $A \subseteq B$, wenn für alle x mit $x \in A : x \in B$. Mit anderen Worten:

$A \subseteq B$ if and only if (iff) $\forall x ((x \in A) \rightarrow (x \in B))$ ist wahr

Möglichkeit 1:

1. $A \subseteq B$
2. für alle x s.t. $x \in A : x \in B$ (1; def \subseteq)

\forall Aussagen beweisen

Inferenzregel für definierte Beziehungen

Beispiel: Für Mengen A und B , definiere A ist **Teilmenge** von B , $A \subseteq B$, wenn für alle x mit $x \in A : x \in B$. Mit anderen Worten:

$A \subseteq B$ if and only if (iff) $\forall x ((x \in A) \rightarrow (x \in B))$ ist wahr

Möglichkeit 2:

1. für alle x s.t. $x \in A : x \in B$
2. $A \subseteq B$ (1; def \subseteq)

\forall Aussagen beweisen

Inferenzregel für \forall

- Sei \mathcal{P} eine Formel
- Beispiel: $\mathcal{P}(x)$ steht für $x \in A$ und $\mathcal{Q}(x)$ steht für $x \in B$
- Dann kann “für alle x s.t. $x \in A : x \in B$ ”
als “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ” geschrieben werden.

Regel um \forall Aussagen zu beweisen (pr \forall)

Um Aussage der Form “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ”, zu beweisen nimmt man an x sei ein beliebig gewähltest Element (eigenvariable) s.t. $\mathcal{P}(x)$ wahr ist. Dann zeigt man: $\mathcal{Q}(x)$ ist wahr.

Generalisierungen z.B. “für alle x, y s.t. $\mathcal{P}(x, y) : \mathcal{Q}(x, y)$ ” möglich

\forall Aussagen beweisen

Inferenzregel für \forall : Ein Beispiel

Sei $C = \{x \in \mathbb{R} \mid x < 1\}$ and $D = \{x \in \mathbb{R} \mid x < 2\}$. Zeige $C \subseteq D$!

Annahme: 1. $C = \{x \in \mathbb{R} \mid x < 1\}$

2. $D = \{x \in \mathbb{R} \mid x < 2\}$

Zeige: $C \subseteq D$

1. Sei $x \in C$ beliebig

2. $x < 1$ (1, ass 1; def C)

3. $x < 2$ (2; prop \mathbb{R})

4. $x \in D$ (3, ass 2; def D)

5. für alle $x \in C : x \in D$ (1 – 4; pr \forall)

6. $C \subseteq D$ (5; def \subseteq)

Wie können wir “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ” widerlegen?

\forall Aussagen beweisen

Inferenzregel für \forall : Bemerkungen

- ☐ Durch Einrückungen kennzeichnen wir **Teilbeweise** die von einer Annahme wie “Sei $x \in C$ beliebig” abhängen.
- ☐ Eine Annahme hat keine Begründung
- ☐ Teilbeweis 2–4 basiert auf Annahme in 1
- ☐ Schritte aus 1–4 können nicht in Begründungen auftauchen, sobald der Teilbeweis fertig ist. (d.h., **nach pr \forall in 5**)
- ☐ Wir schreiben oft “**für alle $x \in C : x \in D$** ” statt
“**für alle x s.t. $x \in C : x \in D$** ”

Verwenden von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Die Regel um \forall Aussagen in Beweisen zu verwenden (us \forall)

Wenn wir wissen das eine Aussage “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ” wahr ist und wir $\mathcal{P}(t)$ bereits als einen Schritt für eine Variable t im Beweis haben, dann können wir $\mathcal{Q}(t)$ ableiten.

Beispiel

1. $t \in A$
2. für alle x s.t. $x \in A : x \in B$
3. ? (1,2; us \forall)

Verwenden von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Die Regel um \forall Aussagen in Beweisen zu verwenden (us \forall)

Wenn wir wissen das eine Aussage “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ” wahr ist und wir $\mathcal{P}(t)$ bereits als einen Schritt für eine Variable t im Beweis haben, dann können wir $\mathcal{Q}(t)$ ableiten.

Beispiel

1. $t \in A$
2. für alle x s.t. $x \in A : x \in B$
3. $t \in B$ (1,2; us \forall)

Verwenden von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Die Regel um \forall Aussagen in Beweisen zu verwenden (us \forall)

Wenn wir wissen das eine Aussage “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ” wahr ist und wir $\mathcal{P}(t)$ bereits als einen Schritt für eine Variable t im Beweis haben, dann können wir $\mathcal{Q}(t)$ ableiten.

Beispiel

1. $|a| < |b|$
2. für alle x, y s.t. $|x| < |y| : x^2 < y^2$
3. ? (1,2; us \forall)

Verwenden von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Die Regel um \forall Aussagen in Beweisen zu verwenden (us \forall)

Wenn wir wissen das eine Aussage “für alle x s.t. $\mathcal{P}(x) : \mathcal{Q}(x)$ ” wahr ist und wir $\mathcal{P}(t)$ bereits als einen Schritt für eine Variable t im Beweis haben, dann können wir $\mathcal{Q}(t)$ ableiten.

Beispiel

1. $|a| < |b|$
2. für alle x, y s.t. $|x| < |y| : x^2 < y^2$
3. $a^2 < b^2$ (1,2; us \forall)

Verwenden von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden: Beispiel

Seien A, B, C Mengen. Zeige \subseteq ist transitiv, d.h., zeige wenn $A \subseteq B$ und $B \subseteq C$, dann $A \subseteq C$.

Annahmen: A, B, C Mengen

1. $A \subseteq B$

2. $B \subseteq C$

Zeige: $A \subseteq C$

1. Sei $x \in A$ beliebig
2. für alle $t \in A : t \in B$ (ass 1; def \subseteq)
3. $x \in B$ (1,2; us \forall)
4. für alle $t \in B : t \in C$ (ass 2; def \subseteq)
5. $x \in C$ (3,4; us \forall)
6. für alle $x \in A : x \in C$ (1 – 5; pr \forall)
7. $A \subseteq C$ (6; def \subseteq)

∨ Aussagen verwenden

Inferenzregel um ∨ Aussagen zu verwenden

Die Regel um ∨ Aussagen in Beweisen anzuwenden (us ∨)

Wenn wir wissen das “ \mathcal{P} oder \mathcal{Q} ” wahr ist und wir beweisen können das \mathcal{R} wahr ist wenn \mathcal{Q} gilt sowie das \mathcal{R} wahr ist wenn \mathcal{P} gilt, dann können wir ableiten das \mathcal{R} gilt.

➡ Dies nennt man auch **Fallunterscheidung!**

Definition 1. Gegeben Mengen A und B , die Vereinigung von A und B , $A \cup B$, ist definiert durch $A \cup B = \{x \mid x \in A \text{ oder } x \in B\}$.

∨ Aussagen verwenden

Inferenzregel um ∨ Aussagen zu verwenden: Beispiel

Zeige:

Für Mengen A, B, C , wenn $A \subseteq C$ und $B \subseteq C$, dann $(A \cup B) \subseteq C$.

Annahmen: A, B, C Mengen

1. $A \subseteq C$

2. $B \subseteq C$

Zeige: $(A \cup B) \subseteq C$

1. Sei $x \in A \cup B$ beliebig
2. $x \in A$ oder $x \in B$ (1; def \cup)
3. **Fall 1: Annahme $x \in A$**
4. für alle $t \in A : t \in C$ (ass 1; def \subseteq)
5. **$x \in C$** (3, 4; us \forall)
6. **Fall 2: Annahme $x \in B$**
7. für alle $t \in B : t \in C$ (ass 2; def \subseteq)
8. **$x \in C$** (6, 7; us \forall)
9. $x \in C$ (2, 3 – 8; us \vee)
10. für alle $x \in A \cup B : x \in C$ (1 – 9; pr \forall)
11. $(A \cup B) \subseteq C$ (10; def \subseteq)

Verwenden und beweisen von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Erweiterte Defintionsregel(def²)

Wenn eine Aussage \mathcal{P} die definierende Eigenschaft von einer Definition ist, ist es zulässig \mathcal{P} zu verwenden oder zu beweisen **ohne** \mathcal{P} selbst als Schritt aufzuführen. Als Begründung für den abgeleiteten Schritt gibt man die Defintion und nicht die Regel um \mathcal{P} zu verwenden oder zu beweisen.

➡ kürzere Beweise (**mit ausgelassenen Details**)

Verwenden und beweisen von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Beispiel

1. $a \in M$
2. $M \subseteq N$
3. ? (1,2; def² \subseteq)

Warnung: Später verwenden wir def und def² synonym!

Verwenden und beweisen von \forall Aussagen

Inferenzregel um \forall Aussagen zu verwenden

Beispiel

1. $a \in M$
2. $M \subseteq N$
3. $a \in N$ (1,2; def² \subseteq)

Warnung: Später verwenden wir def und def² synonym!

Verwenden und beweisen von \vee Aussagen

Inferenzregel um \vee Aussagen zu beweisen

Beweisregel für \vee (pr \vee)

Wenn \mathcal{P} als Schritt in einem Beweis etabliert wurde, dann kann “ \mathcal{P} oder \mathcal{Q} ” als neue Zeile geschrieben werden. Symmetrisch, wenn \mathcal{Q} als Schritt in einem Beweis etabliert wurde, dann kann “ \mathcal{P} or \mathcal{Q} ” als neue Zeile geschrieben werden.

Zeige: Für Mengen A, B, C : wenn $A \subseteq B$ oder $A \subseteq C$, dann $A \subseteq B \cup C$

Verwenden und beweisen von \vee Aussagen

Inferenzregel um \vee Aussagen zu beweisen: Beispiel

Zeige: Für Mengen A, B, C : wenn $A \subseteq B$ oder $A \subseteq C$, dann $A \subseteq B \cup C$

Annahmen: A, B, C Mengen

1. $A \subseteq B$ oder $A \subseteq C$

Zeige: $A \subseteq (B \cup C)$

1. Sei $x \in A$ beliebig
2. $A \subseteq B$ oder $A \subseteq C$ (ass 1)
3. Fall 1: Annahme $A \subseteq B$
4. $x \in B$ (1, 3; def² \subseteq)
5. $x \in B$ or $x \in C$ (4; pr \vee)
6. Fall 2: Annahme $A \subseteq C$
7. $x \in C$ (1, 6; def² \subseteq)
8. $x \in B$ or $x \in C$ (7; pr \vee)
9. $x \in B$ or $x \in C$ (2, 3 – 8; us \vee)
10. $x \in (B \cup C)$ (9; def \cup)
11. für alle $x \in A : x \in (B \cup C)$ (1 – 10; pr \forall)
12. $A \subseteq (B \cup C)$ (11; def \subseteq)

Verwenden und beweisen von \vee Aussagen

Generalisierung \vee Inferenzregeln

Regel um \vee in Beweisen zu verwenden (us \vee), final

Wenn wir wissen das “ \mathcal{P}_1 oder \mathcal{P}_2 oder \dots oder \mathcal{P}_n ” wahr ist und wir beweisen das \mathcal{R} in allen Fällen nicht zu einem Widerspruch führt, dann können wir ableiten das \mathcal{R} wahr ist.

Regel um \vee in Beweisen zu beweisen (us \vee), final

Wir können “ \mathcal{P}_1 oder \mathcal{P}_2 oder \dots oder \mathcal{P}_n ” als Schritt in einem Beweis aufführen falls wir einen von \mathcal{P}_1 bis \mathcal{P}_n im Beweis etabliert haben.

Verwenden und beweisen von \wedge Aussagen

Inferenzregel um \wedge Aussagen zu verwenden

Regel um \wedge zu verwenden (us \wedge)

Wenn “ \mathcal{P} und \mathcal{Q} ” ein Schritt in einem Beweis ist, dann können wir sowohl \mathcal{P} als auch \mathcal{Q} als Schritt aufführen.

Beispiel

1. $a < 1$ und $a \in A$
2. ? (1; us \wedge)

Verwenden und beweisen von \wedge Aussagen

Inferenzregel um \wedge Aussagen zu verwenden

Regel um \wedge zu verwenden (us \wedge)

Wenn “ \mathcal{P} und \mathcal{Q} ” ein Schritt in einem Beweis ist, dann können wir sowohl \mathcal{P} als auch \mathcal{Q} als Schritt aufführen.

Beispiel

1. $a < 1$ und $a \in A$
2. $a < 1$ (oder 2. $a \in A$) (1; us \wedge)

Verwenden und beweisen von \wedge Aussagen

Inferenzregel um \wedge Aussagen zu beweisen

Regel um \wedge zu beweisen (pr \wedge)

Um “ \mathcal{P} and \mathcal{Q} ” in einem Beweis zu zeigen, zeige \mathcal{P} und zeige ebenfalls \mathcal{Q} .

Verwenden und beweisen von \wedge Aussagen

Inferenzregel um \wedge Aussagen zu beweisen

Beispiel

$i.$	\mathcal{P}	
	\vdots	
$j.$	\mathcal{Q}	
	\vdots	
$k.$	$?$	$?$

Verwenden und beweisen von \wedge Aussagen

Inferenzregel um \wedge Aussagen zu beweisen

Beispiel

$i.$ \mathcal{P}

\vdots

$j.$ \mathcal{Q}

\vdots

$k.$ \mathcal{P} and \mathcal{Q} $(i, j; \text{pr } \wedge)$

Verwenden und beweisen von \wedge Aussagen

Beispiel: Zeige für Mengen A, B das gilt $A \cap B = B \cap A$

Annahme: A, B Mengen Zeige: $A \cap B = B \cap A$

1. Sei $x \in A \cap B$ beliebig
2. $x \in A$ und $x \in B$ (1; def \cap)
3. $x \in A$ (2; us \wedge)
4. $x \in B$ (2; us \wedge)
5. $x \in B$ und $x \in A$ (4, 3; pr \wedge)
6. $x \in B \cap A$ (5; def \cap)
7. für alle $x \in A \cap B : x \in B \cap A$ (1 – 6; pr \forall)
8. $A \cap B \subseteq B \cap A$ (7; def \subseteq)
9. $B \cap A \subseteq A \cap B$ (1 – 8; symmetry)
10. $A \cap B \subseteq B \cap A$ und $B \cap A \subseteq A \cap B$ (8, 9; pr \wedge)
11. $A \cap B = B \cap A$ (10; def $=$)

In 11, benutzen wir die Definition $=$, d.h., $A = B$ iff $A \subseteq B \wedge B \subseteq A$

Symmetrieregeln

Symmetrieregeln

Wenn $\mathcal{P}(A_1, B_1, \dots)$ eine Aussage ist die bewiesen wurde für beliebige A_1, B_1, \dots in den Annahmen und Hypothesen, und falls A_2, B_2, \dots eine Permutation von A_1, B_1, \dots ist, dann ist $\mathcal{P}(A_2, B_2, \dots)$ wahr. Dies lässt sich auch auf universelle Variable in für alle Aussage übertragen, d.h., wenn für alle A_1, B_1, \dots : $\mathcal{P}(A_1, B_1, \dots)$ wahr ist, dann ist für alle A_2, B_2, \dots : $\mathcal{P}(A_2, B_2, \dots)$ ebenfalls wahr.

Beispiel von oben:

$$A \cap B \subseteq B \cap A$$

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$

$$B \cap A \subseteq A \cap B$$

Tausche A durch B und B durch A

Theoreme verwenden

Substitutionen anwenden

Regel für Substitutionen (subs)

Jeder Name oder Repräsentat eines mathematischen Objects kann durch einen anderen Namen/Repräsentaten des gleichen Objekts ersetzt werden. Gleiche Namen für unterschiedliche Objekte dürfen nicht verwendet werden.

Zwei Beispiele

1. $A \cap B = C$
2. $A = D$
3. $D \cap B = C$ (1,2; subs)

Theoreme verwenden

Substitutionen anwenden

Regel für Substitutionen (subs)

Jeder Name oder Repräsentat eines mathematischen Objects kann durch einen anderen Namen/Repräsentaten des gleichen Objekts ersetzt werden. Gleiche Namen für unterschiedliche Objekte dürfen nicht verwendet werden.

Zwei Beispiele

1. $x^2 + x = 6$

2. $x = y + 1$

3. $(y + 1)^2 + y + 1 = 6$ (1,2; subs)

Theoreme verwenden

Theoremregel (thm)

Um ein Theorem auf Schritte in einem Beweis anzuwenden, finde eine Aussage \mathcal{P} die äquivalent zur Aussage vom Theorem ist. Dann kann \mathcal{P} als neuer Schritt im Beweis aufgeführt werden oder durch Substitution verwendet werden um einen Schritt zu verändern.

- ☐ Dies ist eine Möglichkeit Lemmas in Beweisen zu verwenden.
- ☐ Weitere Möglichkeiten \rightarrow später

Theoreme verwenden

Beispiel: Zeige für Mengen A, B, C : es gilt

$$A \cup (B \cup C) = (A \cup B) \cup C$$

Annahmen: A, B, C Mengen

Zeige: $A \cup (B \cup C) = (A \cup B) \cup C$

1. Sei $x \in (A \cup B) \cup C$ beliebig
2. $x \in (A \cup B)$ oder $x \in C$ (1; def \cup)
3. Fall 1: $x \in A \cup B$
4. $x \in A$ oder $x \in B$ (3; def \cup)
5. Fall 1a: $x \in A$
6. $x \in A \cup (B \cup C)$ (5; def \cup)

Theoreme verwenden

- | | | |
|-----|---|--|
| 7. | Fall 1b: $x \in B$ | |
| 8. | $x \in B \cup C$ | (7; def \cup) |
| 9. | $x \in A \cup (B \cup C)$ | (8; def \cup) |
| 10. | $x \in A \cup (B \cup C)$ | (4, 5–9; us \vee) |
| 11. | Fall 2: $x \in C$ | |
| 12. | $x \in B \cup C$ | (11; def \cup) |
| 13. | $x \in A \cup (B \cup C)$ | (12; def \cup) |
| 14. | $x \in A \cup (B \cup C)$ | (2, 3–13; us \vee) |
| 15. | $(A \cup B) \cup C \subseteq A \cup (B \cup C)$ | (1, 2–14; def \subseteq) |
| 16. | $C \cup (B \cup A) \subseteq (C \cup B) \cup A$ | (15; Thm $X \cup Y = Y \cup X$) |
| 17. | $A \cup (B \cup C) \subseteq (A \cup B) \cup C$ | (16; symmetry (AC)) |
| 18. | $A \cup (B \cup C) = (A \cup B) \cup C$ | (15, 17; def $=$) |

Regel um Implikationen zu beweisen ($\text{pr} \rightarrow$)

 $i - 1, \dots$

i. Annahme \mathcal{P}

1033

 $j. \quad 2$

$k.$ If \mathcal{P} , then \mathcal{Q} $(i - j; \text{pr} \rightarrow)$

Beweisen und verwenden von if-then Aussagen

Inferenzregel um if-then Aussagen zu beweisen: Beispiel

A, B, C Mengen. Zeige: If $A \subseteq B$, then $A \cap C \subseteq B \cap C$.

Annahmen: A, B, C Mengen

Zeige: If $A \subseteq B$, then $A \cap C \subseteq B \cap C$

1. Annahme $A \subseteq B$
2. Sei $x \in A \cap C$ beliebig
3. $x \in A$ (2; def \cap)
4. $x \in C$ (2; def \cap)
5. $x \in B$ (1, 3; def \subseteq)
6. $x \in B \cap C$ (5, 4; def \cap)
7. $A \cap C \subseteq B \cap C$ (2, 3 – 6; def \subseteq)
8. If $A \subseteq B$, then $A \cap C \subseteq B \cap C$ (1 – 7; pr \rightarrow)

Beweisen und verwenden von if-then Aussagen

Inferenzregel um if-then Aussagen zu verwenden

**Regel um Implikationen zu verwenden (us \rightarrow)
(oder modus ponens (MP))**

Wenn \mathcal{P} und “if \mathcal{P} , then \mathcal{Q} ” Schritte in einem Beweis sind,
dann können wir \mathcal{Q} ableiten und als Schritt schreiben.

$i.$	\mathcal{P}	
	\vdots	
$j.$	If \mathcal{P} , then \mathcal{Q}	
$j+1.$	\mathcal{Q}	$(i, j; \text{us } \rightarrow)$

Beweisen und verwenden von if-then Aussagen

Inferenzregel um if-then Aussagen zu verwenden

Beispiel

1. if $x < 2$, then $x \in A$
2. $x < 2$
3. $x \in A$ (1,2; us \rightarrow)

Beweisen und verwenden von Äquivalenzen

Beweisen von Äquivalenzen ($P \leftrightarrow Q$)

Um zu zeigen “ P ist äquivalent zu Q ”, nimm zuerst an P gilt und zeige Q , und dann nimm Q an und zeige P .

Verwenden von Äquivalenzen ($P \leftrightarrow Q$)

Eine Aussage darf durch eine äquivalente Aussage ersetzt werden.

iff Aussagen

- \mathcal{P} iff \mathcal{Q} gilt genau dann wenn $\mathcal{P} \leftrightarrow \mathcal{Q}$ ist wahr
- Zeige \mathcal{P} iff \mathcal{Q} : beweise “if \mathcal{P} , then \mathcal{Q} ” und “if \mathcal{Q} , then \mathcal{P} ”
- “If \mathcal{P} , then \mathcal{Q} ” wird typischerweise bewiesen durch Annahme von \mathcal{P} und Ableitung von \mathcal{Q}
- Oder “if \mathcal{P} , then \mathcal{Q} ” Kontraposition beweisen:
Zeige “if \mathcal{P} , then \mathcal{Q} ”: Annahme von $\neg \mathcal{Q}$ und Herleitung von $\neg \mathcal{P}$

Widerspruchsbeweise

Das Ableitungsschema

Idee: Nimm die Negation von \mathcal{P} an und leite einen Widerspruch her!

	\vdots	
$i.$	\mathcal{Q}	
	\vdots	
$j.$	Annahme $\neg \mathcal{P}$ (um einen Widerspruch zu erhalten)	
	\vdots	
$k.$	$\neg \mathcal{Q}$ (Widerspruch zu \mathcal{Q} bei $i.$)	
$k + 1.$	\mathcal{P}	$(j - k; \text{Widerspruch})$

Widerspruchsbeweise (Beispiel)

Zeige: For all $x \in \mathbb{R}$ mit $x \in [0, \pi/2]$: $\sin x + \cos x \geq 1$

1. Sei $x \in \mathbb{R} \wedge x \in [0, \pi/2]$ beliebig
2. $\sin x \geq 0$ and $\cos x \geq 0$ (1; prop sin, cos)
3. Annahme $\neg(\sin x + \cos x \geq 1)$, d.h., $\sin x + \cos x < 1$
4. $\sin x + \cos x < 1$ (3; prop \mathbb{R})
5. $(\sin x + \cos x)^2 < 1^2$ (4; prop \mathbb{R})
6. $\sin^2 x + 2 \sin x \cos x + \cos^2 x < 1^2$ (5; prop \mathbb{R})
7. $1 + 2 \sin x \cos x < 1^2$ (6; $\sin^2 + \cos^2 = 1$)
8. $\sin x \cos x < 0$ (7; prop \mathbb{R})
9. entweder $\sin x < 0$ oder $\cos x < 0$ (8; prop \mathbb{R})
Widerspruch zu 2.
10. $\sin x + \cos x \geq 1$ (3 – 9; Widerspruch)
11. $\forall x(x \in \mathbb{R} \wedge x \in [0, \pi/2]) : \sin x + \cos x \geq 1$ (1 – 10; pr \forall)

Existenzaussagen

Verwendung

Existenzaussagen verwenden (us \exists)

Um eine Aussage “ $\mathcal{P}(j)$ für ein $1 \leq j \leq n$ ” in einem Beweis zu verwenden, schreibe

“Wähle $1 \leq j_0 \leq n$ s.t. $\mathcal{P}(j_0)$ ”.

Das definiert das Symbol j_0 . Beide Ausdrücke $1 \leq j_0 \leq n$ und $\mathcal{P}(j_0)$ können später im Beweis verwendet werden.

Existenzaussagen

Verwendung

Beispiel: Für $i = 1, 2, \dots, 10$, definiere $A_i = \{t \in \mathbb{R} \mid 0 < t < \frac{1}{i}\}$

1. $x \in A_i$ für ein $1 \leq i \leq 10$
2. wähle $1 \leq j_0 \leq 10$ s.t. $x \in A_{j_0}$ (1; us \exists)
3. $1 \leq j_0 \leq 10$ (von Schritt 2.)
4. $x \in A_{j_0}$ (von Schritt 2.)

Existenzaussagen

Beweisen (Leichte Variante)

Beweisen von Existenzaussagen (pr \exists)

Wenn $1 \leq i \leq n$ und $\mathcal{P}(i)$ Schritte in einem Beweis sind, dann darf “für ein $1 \leq j \leq n : \mathcal{P}(j)$ ” als Schritt im Beweis geschrieben werden.

Beispiele

1. $x \in A_j$
2. $1 \leq j \leq n$
3. ?

Existenzaussagen

Beweisen (Leichte Variante)

Beweisen von Existenzaussagen (pr \exists)

Wenn $1 \leq i \leq n$ und $\mathcal{P}(i)$ Schritte in einem Beweis sind, dann darf “für ein $1 \leq j \leq n : \mathcal{P}(j)$ ” als Schritt im Beweis geschrieben werden.

Beispiele

1. $x \in A_j$
2. $1 \leq j \leq n$
3. for some $1 \leq i \leq n : x \in A_i$ (1, 2; pr \exists)

Existenzaussagen

Beweisen (Leichte Variante)

Beweisen von Existenzaussagen (pr \exists)

Wenn $1 \leq i \leq n$ und $\mathcal{P}(i)$ Schritte in einem Beweis sind, dann darf “für ein $1 \leq j \leq n : \mathcal{P}(j)$ ” als Schritt im Beweis geschrieben werden.

Beispiele

1. ?
2. $1 \leq 3 \leq 10$
3. Für ein $1 \leq i \leq 10 : x \in A_i$ (1, 2; pr \exists)

Existenzaussagen

Beweisen (Leichte Variante)

Beweisen von Existenzaussagen (pr \exists)

Wenn $1 \leq i \leq n$ und $\mathcal{P}(i)$ Schritte in einem Beweis sind, dann darf “für ein $1 \leq j \leq n : \mathcal{P}(j)$ ” als Schritt im Beweis geschrieben werden.

Beispiele

1. $x \in A_3$
2. $1 \leq 3 \leq 10$
3. Für ein $1 \leq i \leq 10 : x \in A_i$ (1, 2; pr \exists)

Erweiterte Existenzaussagen

Beweisen von Existenzaussagen (pr \exists)

Um eine Aussage “für ein $1 \leq j \leq n : \mathcal{P}(j)$ ”, definiere j im Beweis und zeige das $\mathcal{P}(j)$ und $1 \leq j \leq n$ für j gelten.

A_1, \dots, A_n Mengen. Zeige: Für alle $1 \leq j \leq n : A_j \subseteq \bigcup_{i=1}^n A_i$

1. Sei $1 \leq j \leq n$
2. Sei $x \in A_j$ beliebig
3. für ein $1 \leq i \leq n : x \in A_i$ (1, 2; pr \exists)
4. $x \in \bigcup_{i=1}^n A_i$ (3; def \cup)
5. $A_j \subseteq \bigcup_{i=1}^n A_i$ (2, 3 – 4; def \subseteq)
6. für alle $1 \leq j \leq n : A_j \subseteq \bigcup_{i=1}^n A_i$ (1 – 5; pr \forall)

Negierungen

Negierungsregel (\neg)

Die Negierung von “für alle $1 \leq i \leq n : \mathcal{P}(i)$ ” ist “für ein $1 \leq i \leq n : \neg \mathcal{P}(i)$ ”. Die Negierung von “für ein $1 \leq i \leq n : \mathcal{P}(i)$ ” ist “für alle $1 \leq i \leq n : \neg \mathcal{P}(i)$ ”.

Induktion

Wie können wir folgende Aussagen beweisen?

- ☐ Summenformeln wie $\sum_{i=0}^n 2^i = 2^{n+1} - 1$ für alle $n \in \mathbb{N}_0$
- ☐ Ungleichungen wie $2^n < n!$ für jede natürliche Zahl $n \geq 4$
- ☐ Teilbarkeiten wie $n^3 - n$ ist teilbar durch 3 für jedes $n \in \mathbb{N}$
- ☐ Resultate über Mengen wie jede Menge mit $n \in \mathbb{N}$ Elementen hat 2^n Teilmengen
- ☐ Resultate über Algorithmen wie Korrektheit oder Terminierung
Die Funktion `fac(n)` gibt $n!$ zurück für alle $n \in \mathbb{N}_0$

Induktionsprinipien

- Eine zentrale Beweistechnik in der Mathematik und Informatik!
- Formalisierung:

$$\left[\mathcal{P}(1) \wedge \forall k \in \mathbb{N} (\mathcal{P}(k) \rightarrow \mathcal{P}(k+1)) \right] \rightarrow \forall n \in \mathbb{N} \mathcal{P}(n)$$

- Es gibt verschiedene Arten
 - Mathematische Induktion (+Varianten)
 - Strong mathematical induction (not here)
 - Strukturelle Induktion (not here)
 - Noetherian (or well-founded) induction (not here)

How does it work behind the scene?

$$\left[\mathcal{P}(1) \wedge \forall k \in \mathbb{N} (\mathcal{P}(k) \rightarrow \mathcal{P}(k+1)) \right] \rightarrow \forall n \in \mathbb{N} \mathcal{P}(n) (1)$$

Wir müssen zeigen $\mathcal{P}(1) \wedge \forall k \in \mathbb{N} (\mathcal{P}(k) \rightarrow \mathcal{P}(k+1))$ dann gilt $\forall n \in \mathbb{N} \mathcal{P}(n)$ mit (1) durch modus ponens (MP).

Mathematische Induktion

Sei $\mathcal{P}(n)$ eine Aussage die eine Variable n verwendet. Angenommen

1. $\mathcal{P}(1)$ ist wahr ;
2. wenn $\mathcal{P}(k)$ wahr für eine natürliche Zahl $k \geq 1$,
dann ist $\mathcal{P}(k+1)$ ebenfalls wahr.

Then $\mathcal{P}(n)$ is true for all natural numbers $n = 1, 2, \dots$

- ☐ Im **Induktionsanfang** zeigen wir $\mathcal{P}(1)$ ist wahr.
- ☐ Es uns erlaubt $\mathcal{P}(k)$ als wahr anzunehmen für ein $k \geq 1$. Dies nennt man **Induktionshypothese**.
- ☐ Wir zeigen das $\mathcal{P}(k+1)$ gilt **Induktionsschritt**. In diesem Schritt wird in der Regel die Induktionshypothese verwendet.

Mathematische Induktion (Beispiel)

Theorem: Die Summe der ersten n positiven ungerade Zahlen ist n^2 .

Beweis. Wir wollen zeigen das für alle natürlichen Zahlen n , $\mathcal{P}(n)$ gilt, mit $\mathcal{P}(n)$ stehend für $\sum_{i=0}^{n-1} (2i+1) = n^2$.

IA. Wir zeigen $\mathcal{P}(1)$ ist wahr. Dies stimmt da

$$\sum_{i=0}^{n-1} (2i+1) = 1 = n^2.$$

IH. Annahme $\mathcal{P}(k)$ ist wahr für ein $k \geq 1$.

Induktionsschritt. Wir wollen $\mathcal{P}(k+1)$ ist wahr zeigen. Wir müssen also zeigen: $\sum_{i=0}^k (2i+1) = (k+1)^2$.

Mathematische Induktion (Beispiel)

Durch Induktionshypothese gilt $\mathcal{P}(k)$. Wir leiten ab

$\mathcal{P}(k)$ ist wahr iff

$$\sum_{i=0}^{k-1} (2i+1) = k^2 \quad \text{iff}$$

$$2k+1 + \sum_{i=0}^{k-1} (2i+1) = k^2 + 2k+1 \quad \text{iff}$$

$$\sum_{i=0}^k (2i+1) = (k+1)^2.$$

Also ist $\mathcal{P}(k+1)$ wahr.



1 Amuse Geule

Beispiel: Langzahl-Multiplikation

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

Ziffernfolge stellt die Zahl $\sum_{0 \leq i < n} a_i B^i$ dar

Wir zählen

Volladditionen und Ziffernmultiplikationen

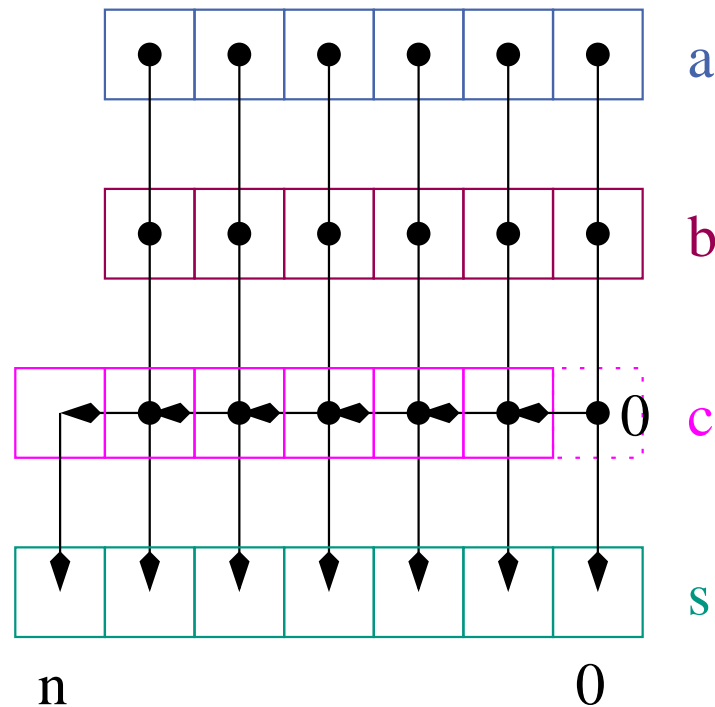
1.1 Addition

$c=0$: 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: $(c, s_i) := a_i + b_i + c$

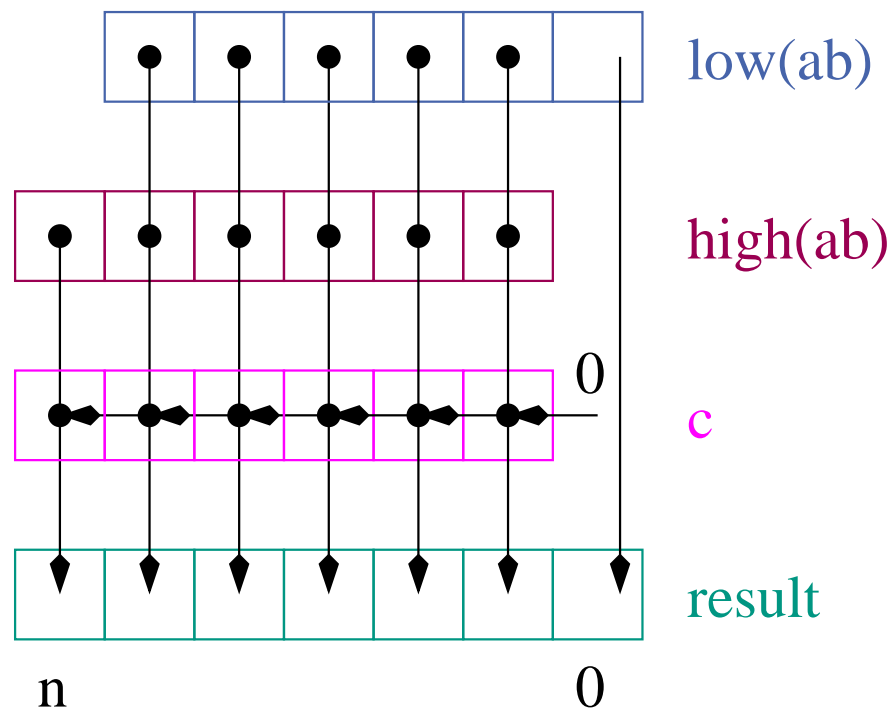
Schleifen: **for** , **while** , **repeat** ... **until** , ...

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
         $(c, \text{RESULT}[i]) := c + h' + \ell$         // Ziffernaddition
         $h' := h$ 
    result[n] :=  $c + h'$                             // Ziffernaddition, kein Überlauf?!
    return RESULT
```

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

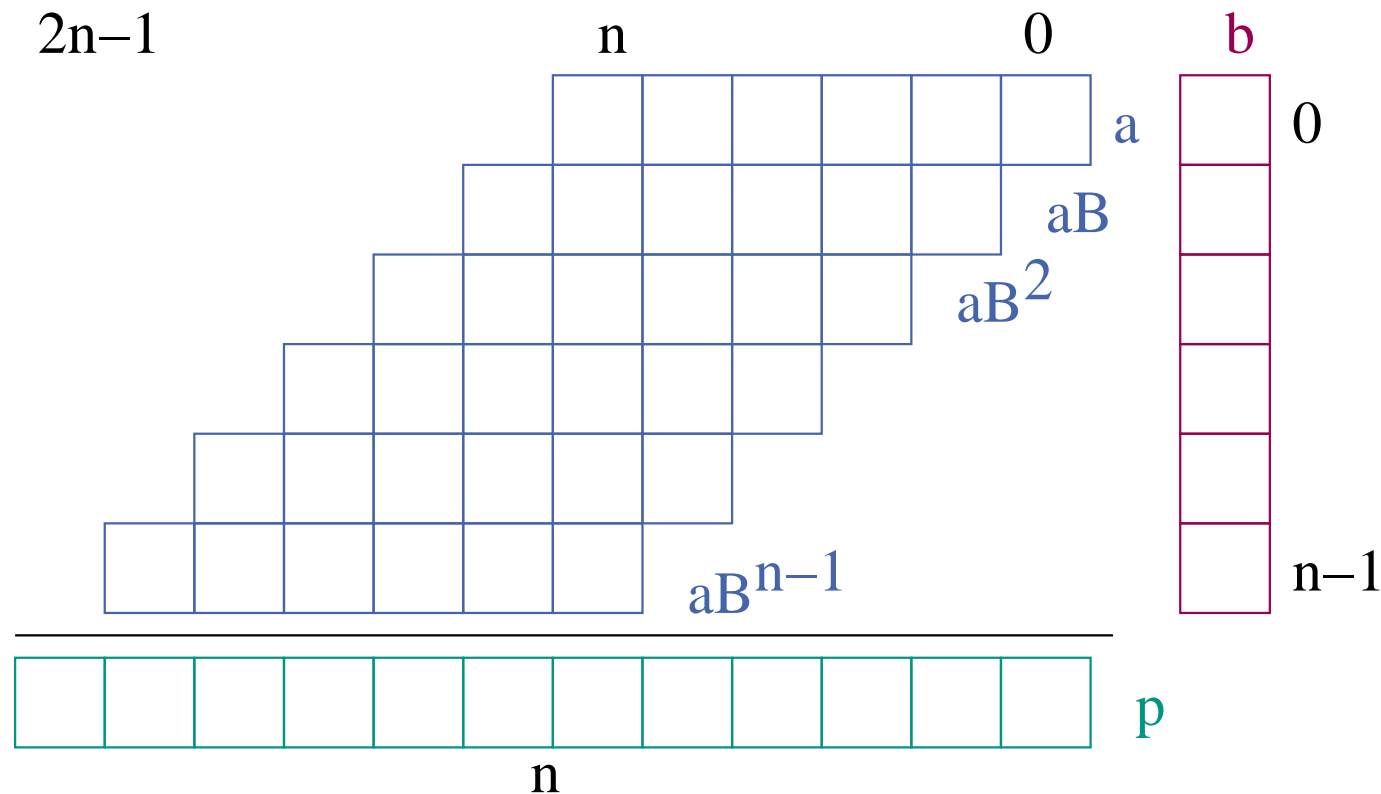
1.2 Schulmultiplikation

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

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

//Langzahladdition, Langzahl mal Ziffer, Schieben:

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



Schulmultiplikation Analyse

$p \leftarrow 0 : \mathbb{N}$

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

Exkurs O-Kalkül, die Erste

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

Idee: Konstante Faktoren (und Anfangsstück) ausblenden

+ Operationen zählen \rightsquigarrow Laufzeit welche Ops.?

+ Rechnungen vereinfachen

+ Interpretation vereinfachen

? Werfen wir zuviel Information weg ?

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

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 $a_1 \cdot B^k + a_0$

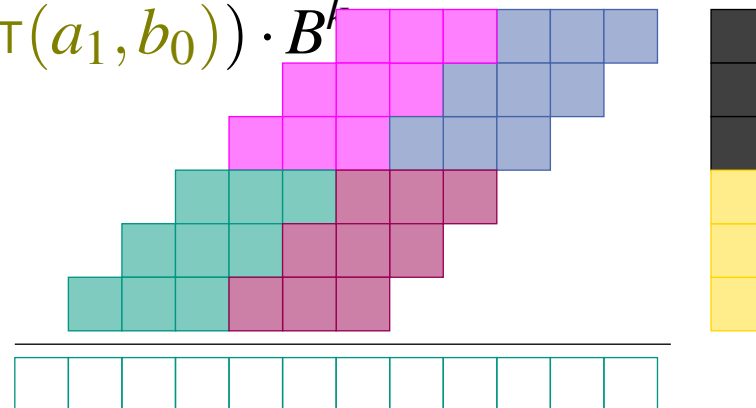
Schreibe b als $b_1 \cdot B^k + b_0$

return

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

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

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

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

→ (Master-Theorem, stay tuned)

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

Aufgabe:

Zeigen Sie durch vollständige Induktion, dass

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

, falls n eine Zweierpotenz ist

Exkurs: Algorithmen-Entwurfsmuster

Im Buch: siehe auch Index!

Schleife: z. B. Addition

Unterprogramm: z. B. Ziffernmultiplikation, Addition

Teile und Herrsche: (lat. divide et impera, engl. divide and conquer)

Aufteilen in eins oder mehrere, **kleinere** Teilprobleme,
oft rekursiv

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

1.5 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 $b_1 \cdot B^k + b_0$

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

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

return

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

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

$+ c_{00}$

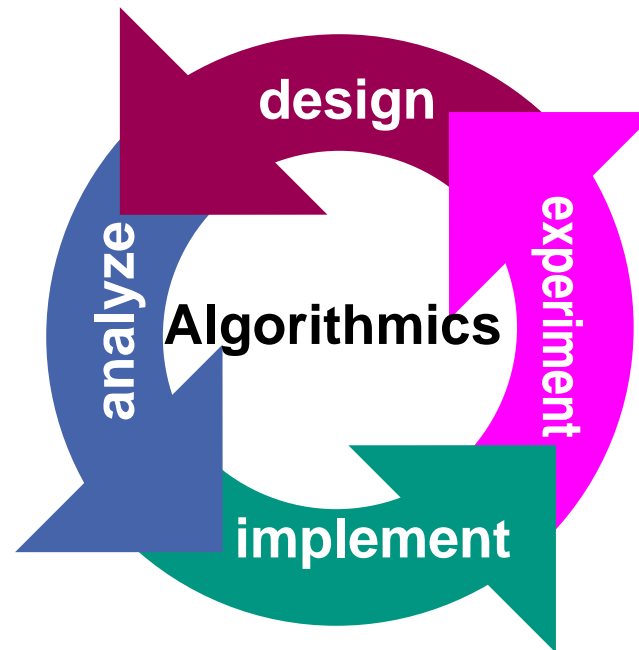
Analyse

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

→ (Master-Theorem)

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

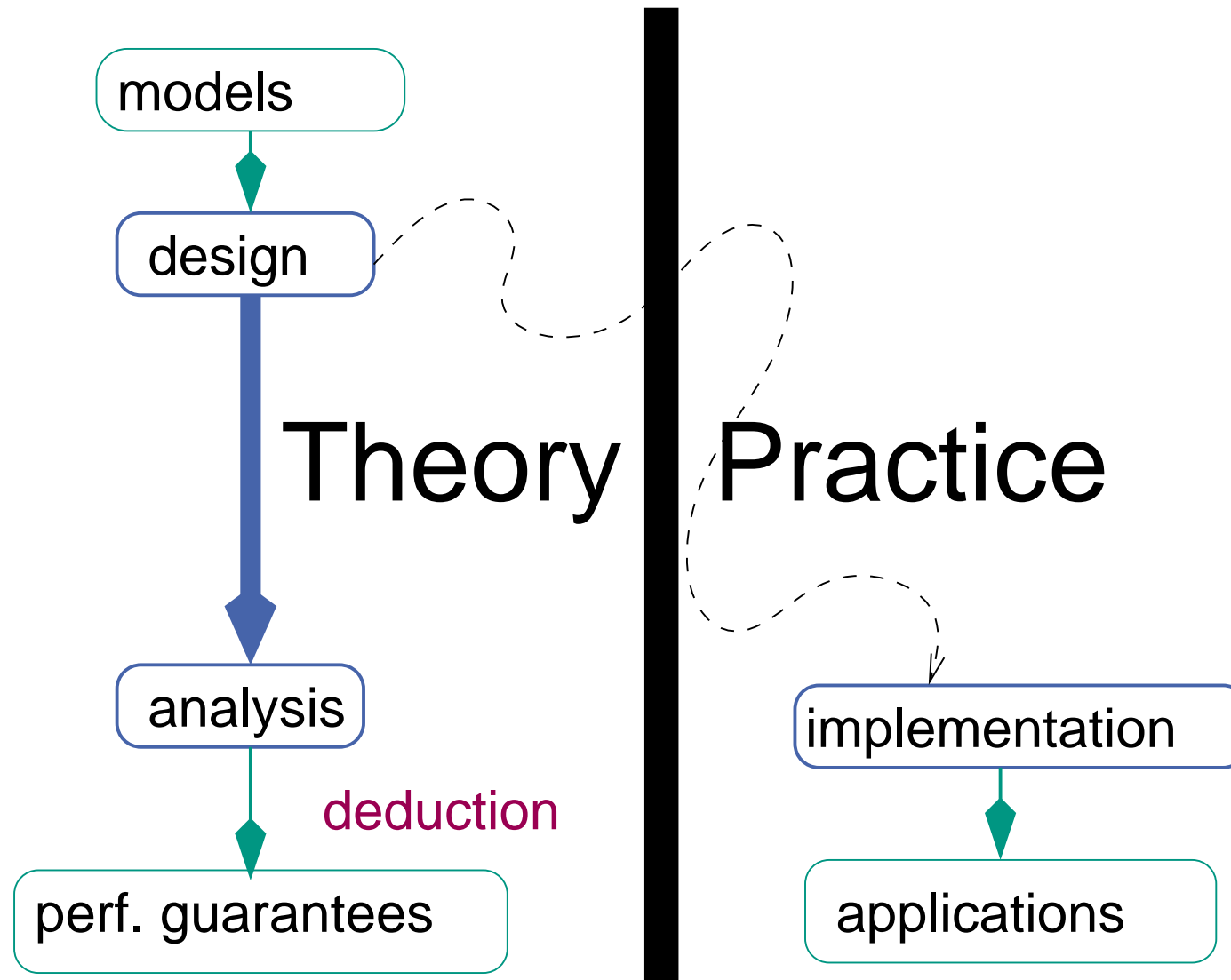
1.6 Algorithm Engineering – was hat das mit der Praxis zu tun?'



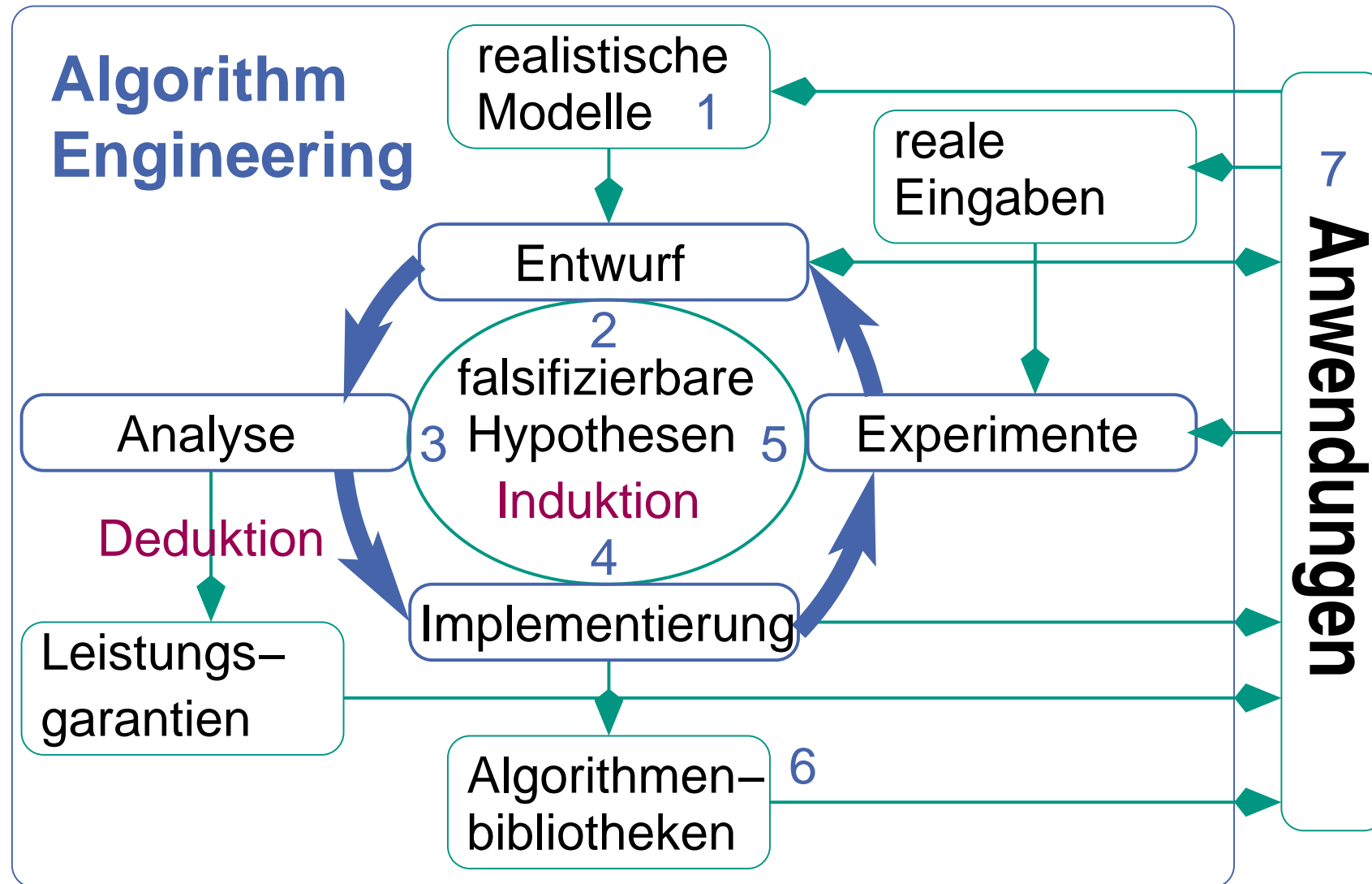
Mehr: DFG Schwerpunktprogram

www.algorithm-engineering.de

Algorithmmentheorie (Karikatur)



Algorithmik als Algorithm Engineering



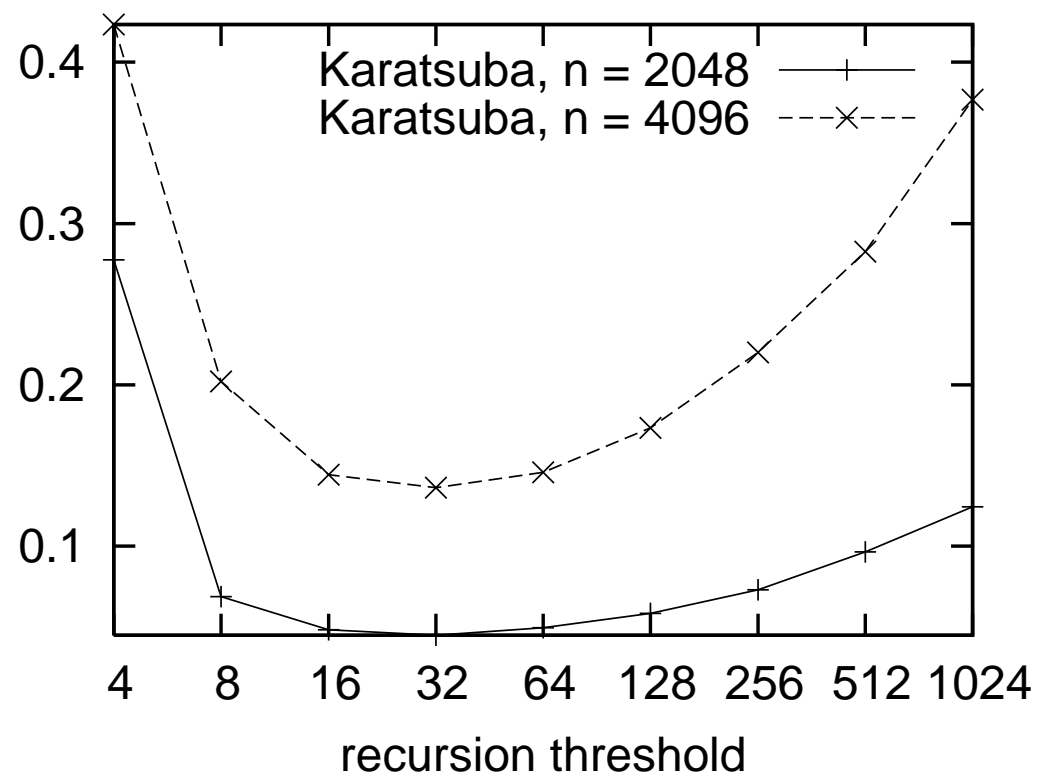
Zurück zur Langzahlmultiplikation

☐ Zifferngröße \leftrightarrow Hardware-Fähigkeiten

z. B. 32 Bit

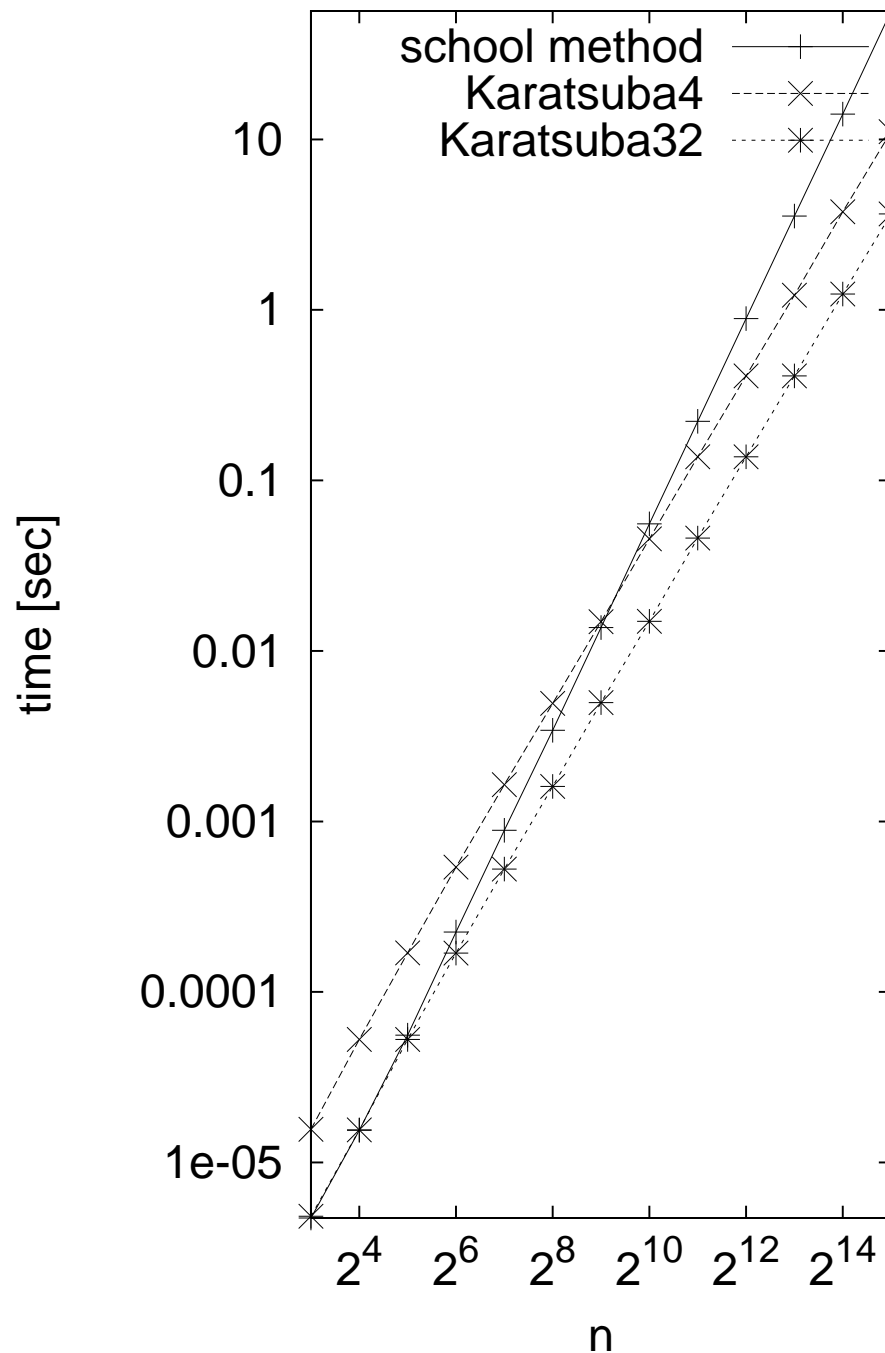
☐ Schulmultiplikation für kleine Eingaben

☐ Assembler, SIMD,...



Skalierung

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

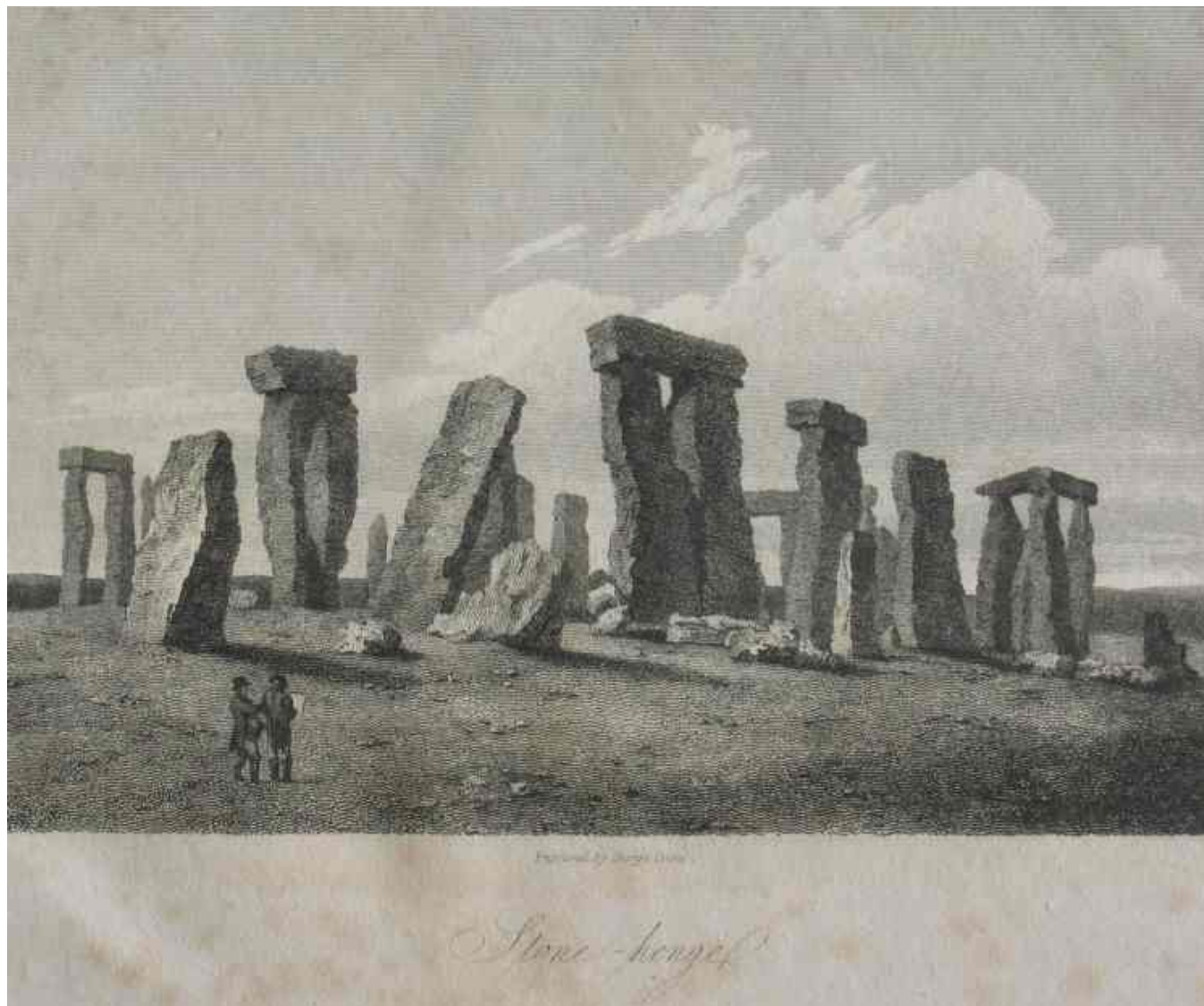


Blick über den Tellerrand

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

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

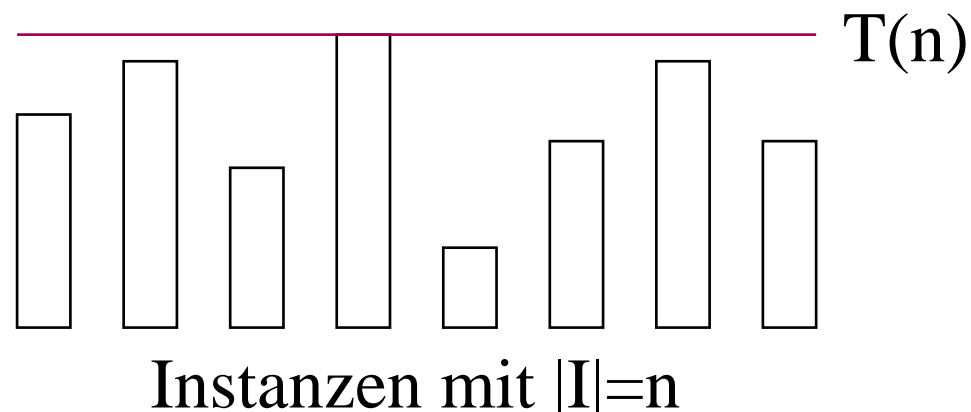
Gegeben: Ein Programm

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

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

„höchstens“

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

„mindestens“

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

„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 ' $\{f(n)\} \subseteq E$ '

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

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

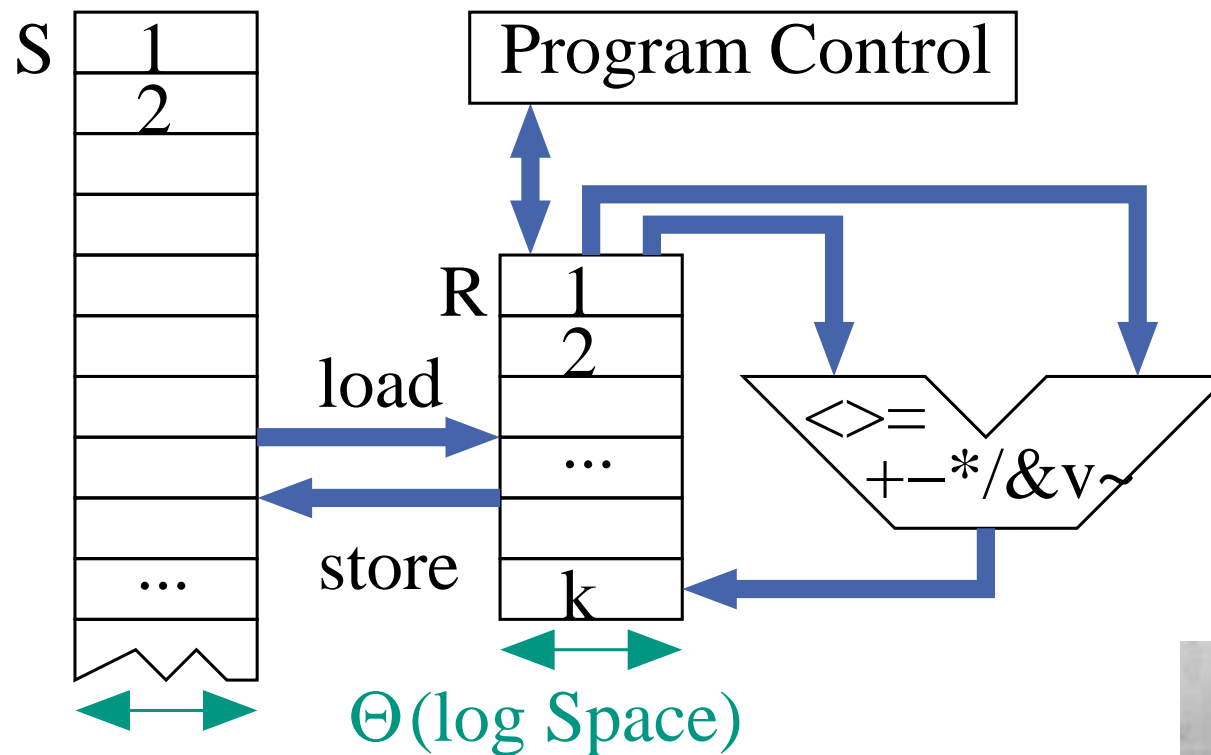
$$f(n) + g(n) = \Omega(f(n)),$$

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

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

u. s. w.

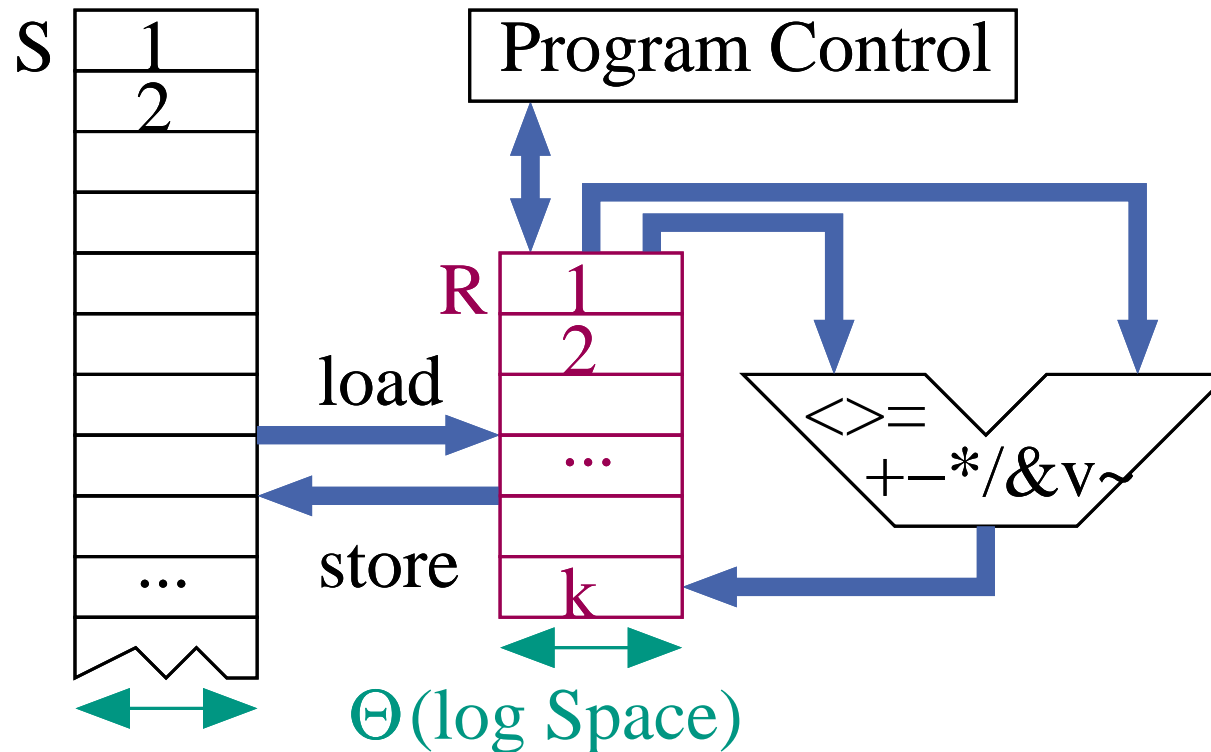
2.3 Maschinenmodell: RAM (Random Access Machine)



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



Register

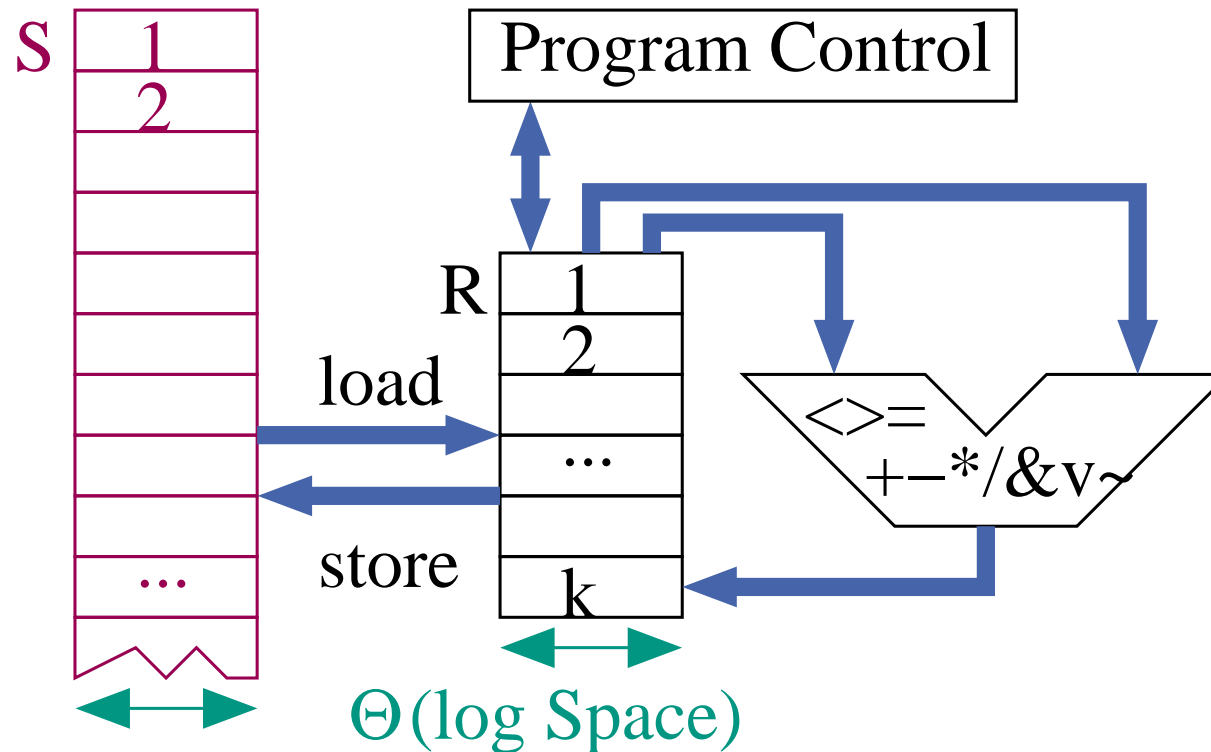


k (irgendeine Konstante) Speicher

R_1, \dots, R_k für

(kleine) ganze Zahlen

Hauptspeicher

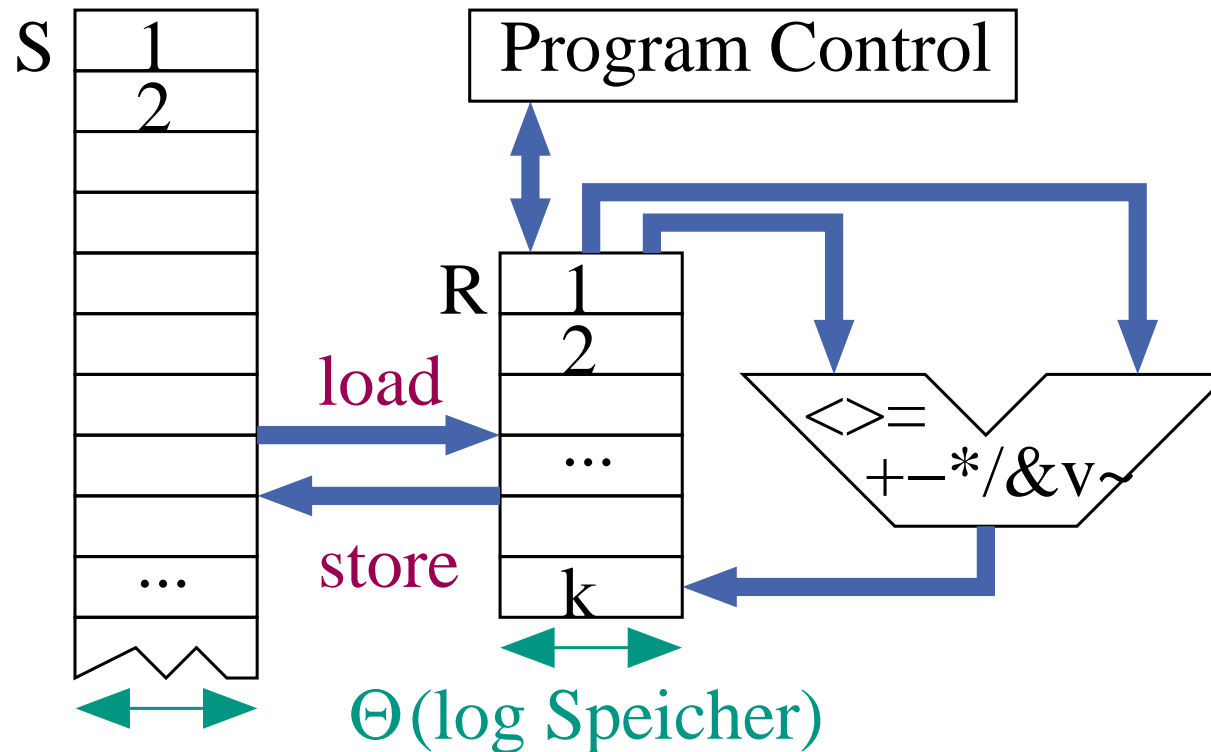


Unbegrenzter Vorrat an Speicherzellen

$S[1], S[2] \dots$ für

(kleine) ganze Zahlen

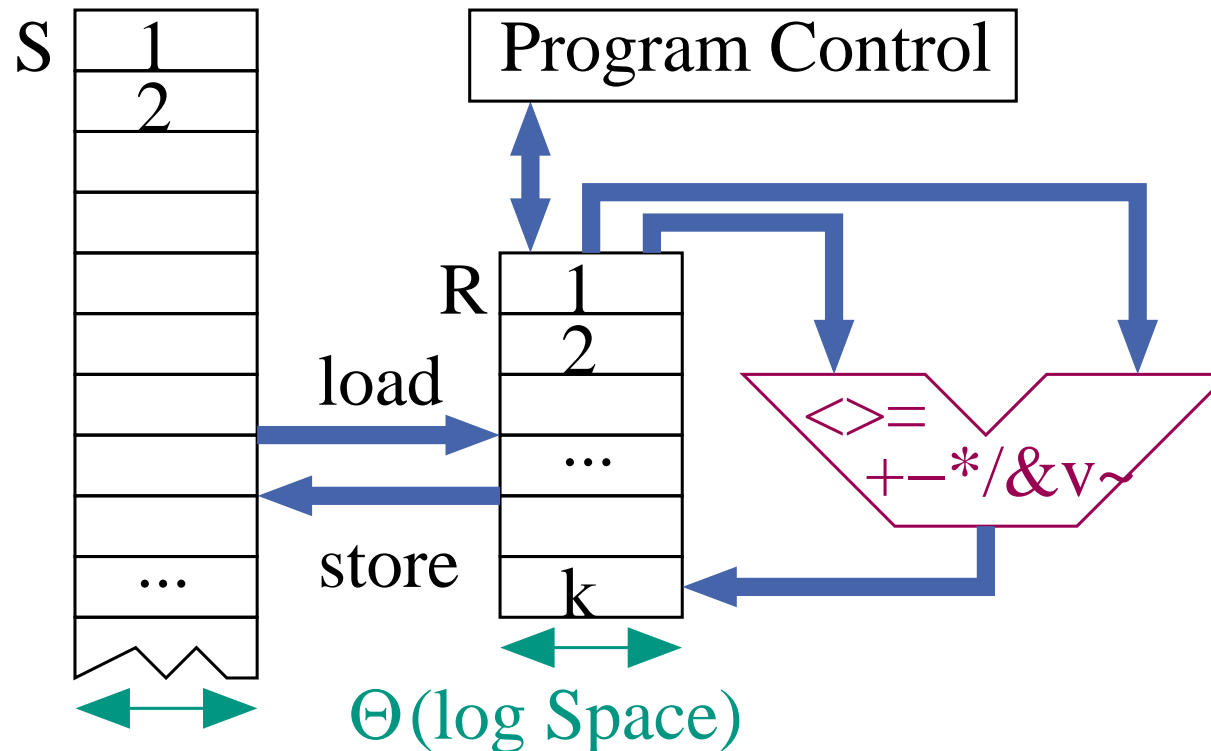
Speicherzugriff



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

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

Rechnen

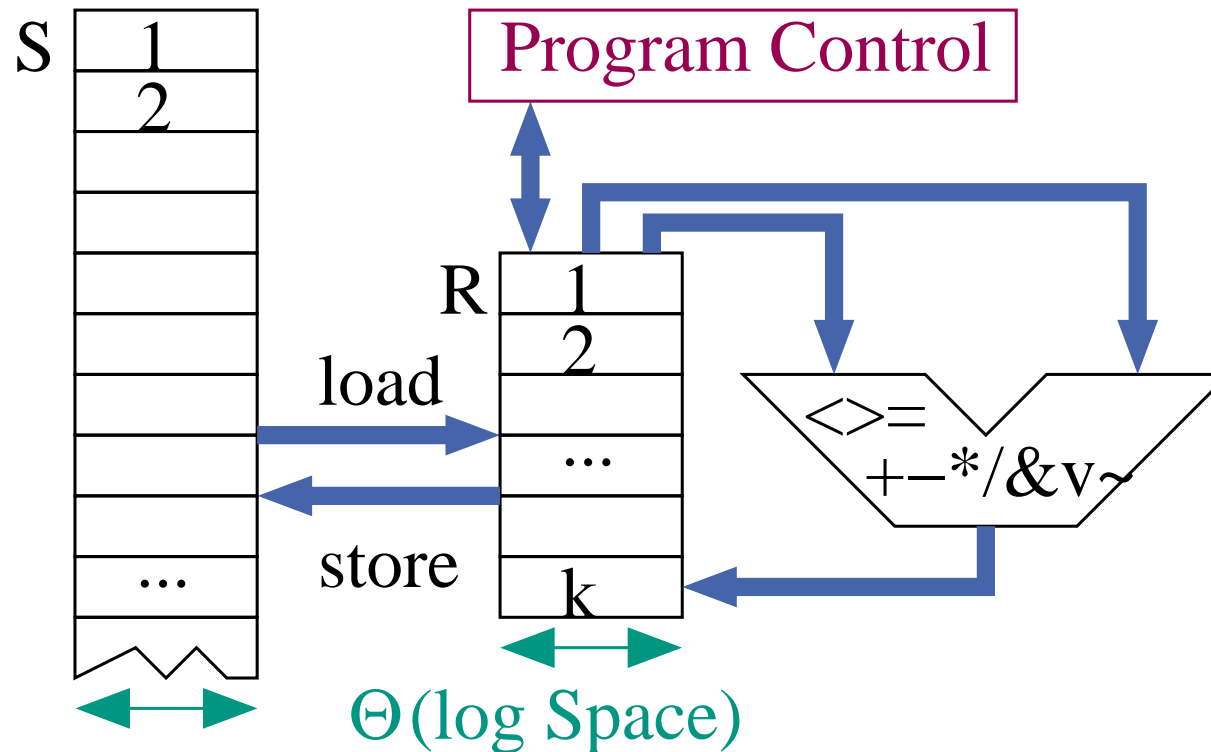


$R_i := R_j \odot R_\ell$ Registerarithmetik.

‘ \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 $O(\cdot)$ gerechtfertigt!

Ignoriert Cache, Pipeline, Parallelismus. . .

Platz: Etwas unklar:

☐ letzte belegte Speicherzelle?

☐ Anzahl benutzter Speicherzellen?

☐ Abhängigkeit von Speicherverwaltungsalgorithmen?

Hier: Es kommt eigentlich nie drauf an.

Mehr Maschinenmodell

Cache: schneller Zwischenspeicher

- begrenzte Größe

 - ↪ kürzlich/häufig zugegriffene Daten sind eher im Cache

- blockweiser Zugriff

 - ↪ Zugriff auf konsekutive Speicherbereiche sind schnell

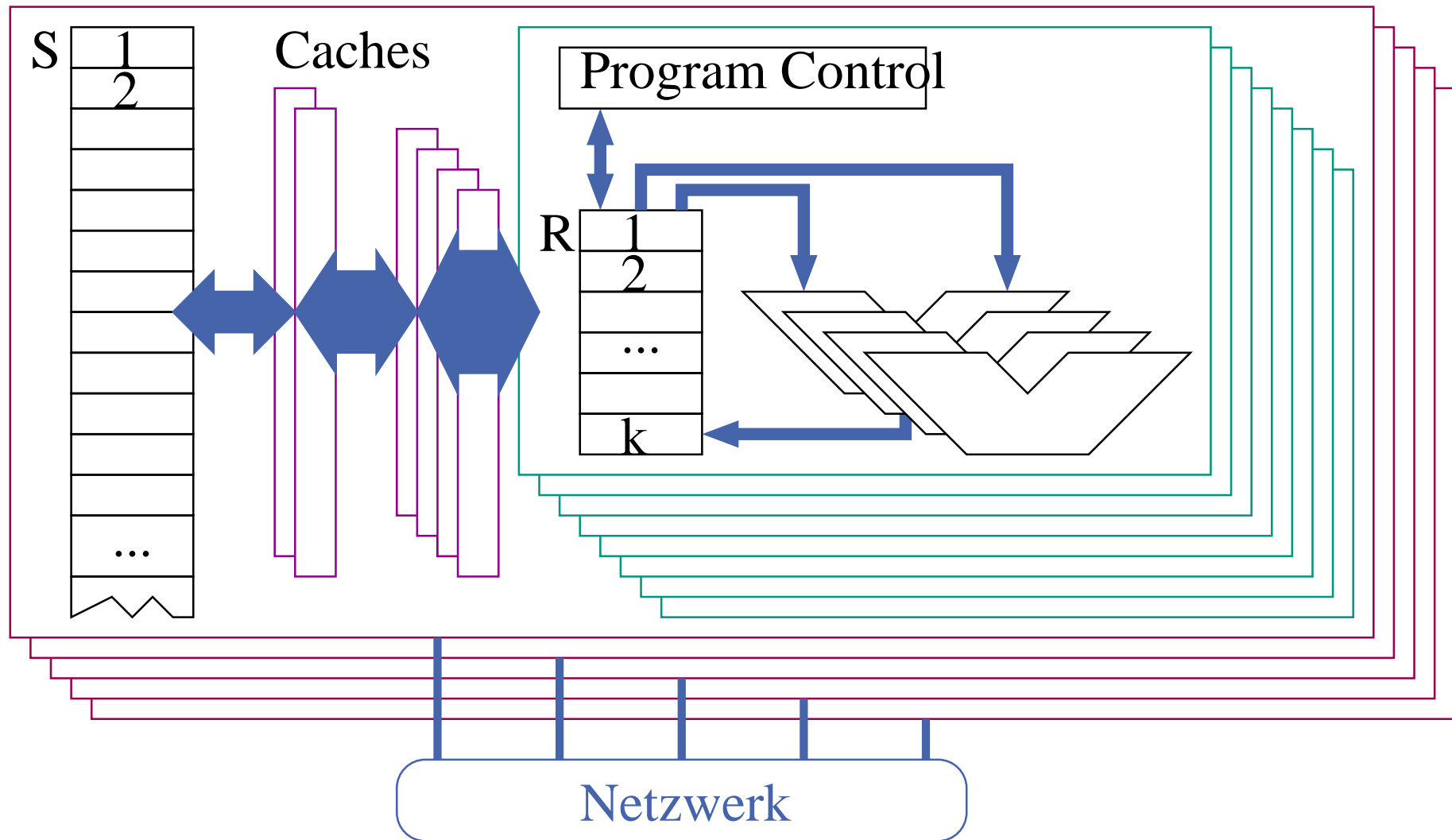
Parallelverarbeitung: Mehrere Prozessoren

- ↪ unabhängige Aufgaben identifizieren

...

mehr in TI, Algorithmen II, ...

Mehr Maschinenmodell



2.4 Pseudocode

just in time

Beispiel

Class Complex($x, y : \text{ELEMENT}$) **of** NUMBER

Number $r := x$

Number $i := y$

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

Function add($c' : \text{COMPLEX}$) : 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

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.

2.6 Beispiel (Ein anderes als im Buch)

Function $\text{power}(a : \mathbb{R}; n_0 : \mathbb{N}) : \mathbb{R}$

assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung

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

while $n > 0$ **do**

invariant $p^n r = a^{n_0}$ // Schleifeninvariante (*)

if n is odd **then** $n \leftarrow n - 1; r := r \cdot p$

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

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

return r

Beispiel

Function $\text{power}(a : \mathbb{R}; n_0 : \mathbb{N}) : \mathbb{R}$

assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung

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

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 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

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

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 $\xrightarrow{\quad}$ Maschinenbefehle

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

$$\square T(I; I') = T(I) + T(I').$$

$$\square T(\text{if } C \text{ then } I \text{ else } I') = O(T(C) + \max(T(I), T(I'))).$$

$$\square T(\text{repeat } I \text{ until } C) = O(\sum_i T(i\text{-te Iteration}))$$

Rekursion \rightsquigarrow Rekurrenzrelationen

2.7.1 Schleifenanalyse \rightsquigarrow Summen ausrechnen

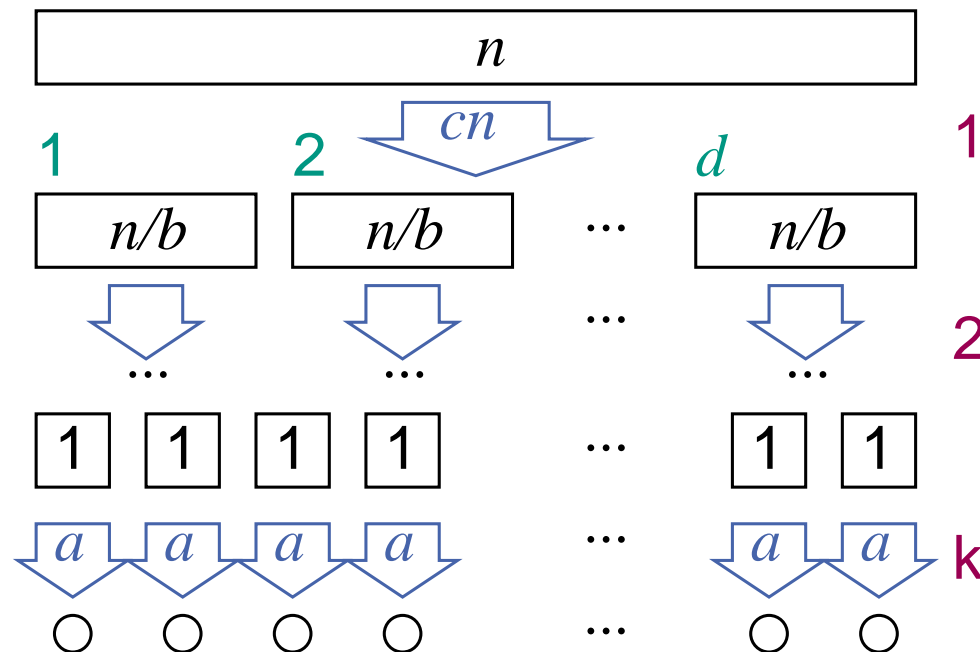
Das lernen Sie in Mathe

Beispiel: Schulumultiplikation

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

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

Beweisskizze

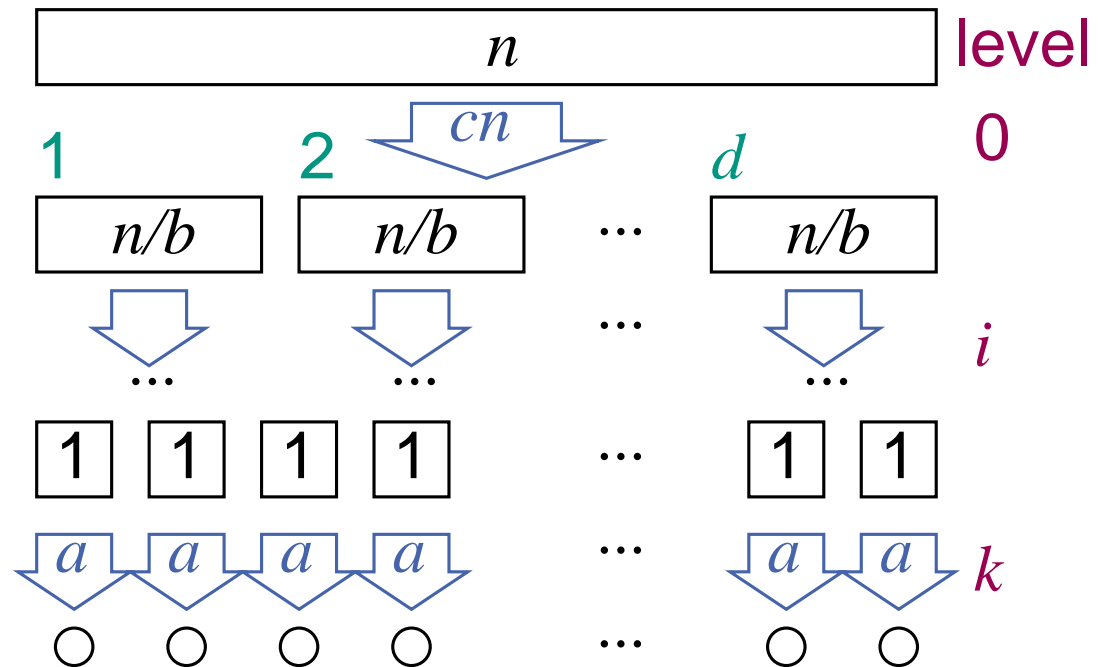
Auf Ebene i , haben wir d^i Probleme @ $n/b^i = b^{k-i}$

cost

cn

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

ad^k

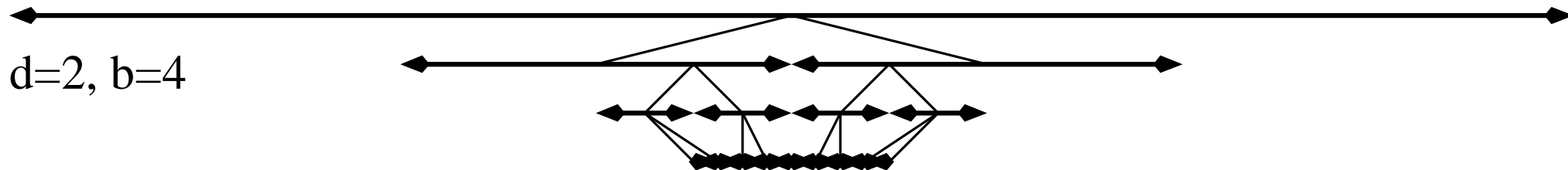


Beweisskizze Fall $d < b$

geometrisch schrumpfende Reihe

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

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

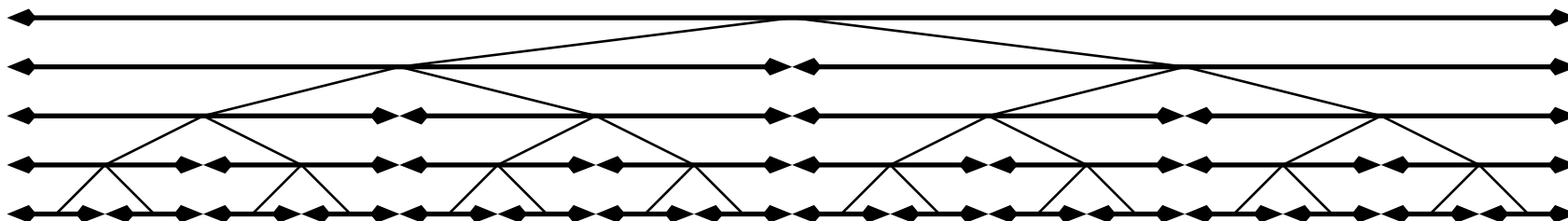


Beweisskizze Fall $d = b$

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

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

$d=b=2$



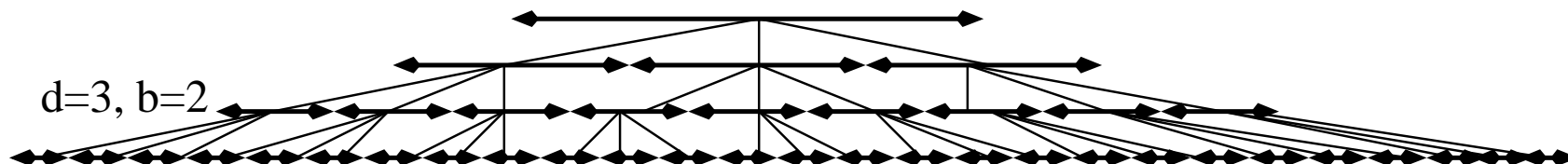
Beweisskizze Fall $d > b$

geometrisch wachsende Reihe

—→ letzte Rekursionsebene kostet konstanten Teil der Arbeit

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

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 $k \in \mathbb{N}$.

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

a schon gesehen, kommt noch, allgemeinerer Fall

$d < b$: Median bestimmen

$d = b$: mergesort, quicksort

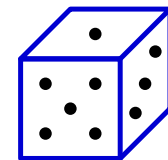
$d > b$: Schulumultiplikation, Karatsuba-Ofman-Multiplikation

2.8 Analyse im Mittel

später an Beispielen

2.9 Randomisierte Algorithmen

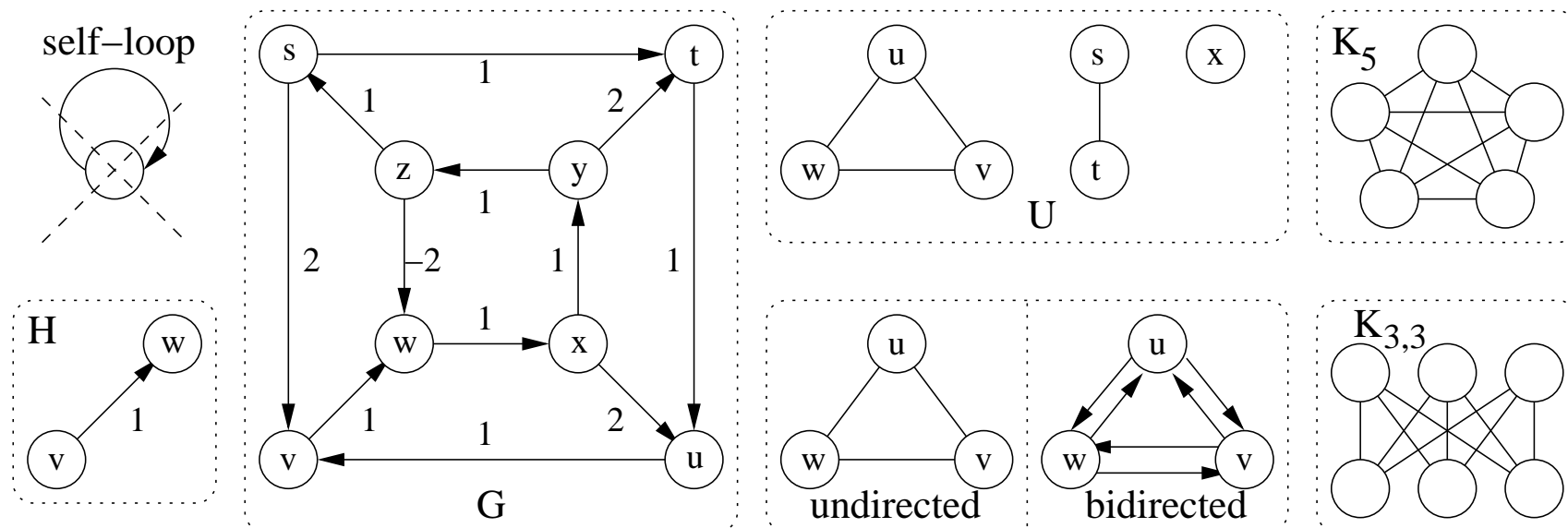
später an Beispielen



2.10 Graphen

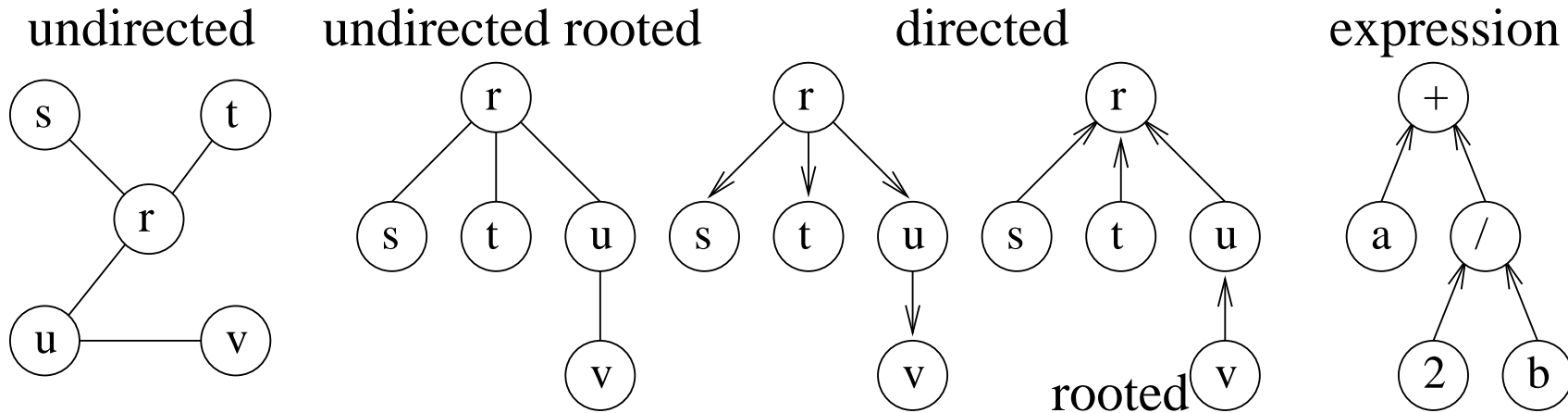
Sie kennen schon (?): **Relationen**, Knoten, Kanten, (un)gerichtete Graphen, Kantengewichte, Knotengrade, Kantengewichte, knoteninduzierte Teilgraphen.

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 $\exists v \in V : \text{OUTDEGREE}(v) = 0$ **do**

invariant G is a DAG iff the input graph is a DAG

$V := V \setminus \{v\}$

$E := E \setminus (\{v\} \times V \cup V \times \{v\})$

return $|V|=0$

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

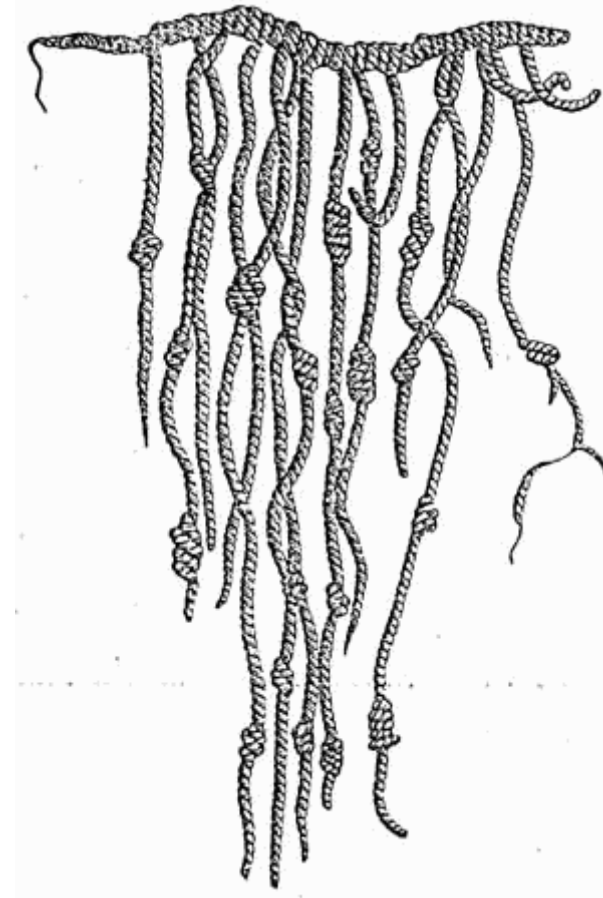
2.11 P und NP

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 $\langle 2, 3, 5, 7, 9, 11, \dots \rangle$

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	
$ \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

3.1 Verkettete Listen

3.1.1 Doppelt verkettete Listen



Listenglieder (Items)

Class Handle = **Pointer to** Item

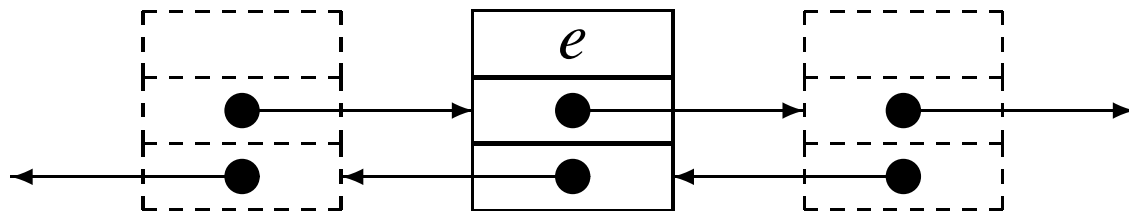
Class Item **of** Element // one link in a doubly linked list

e : Element

next : Handle //

prev : Handle

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

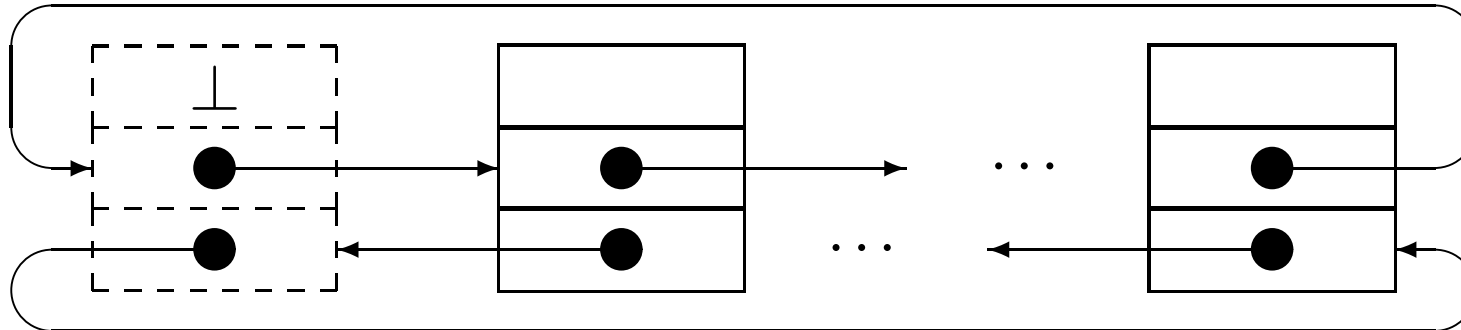


Problem:

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

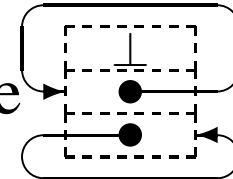
//and the successor of the last element.

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

// Pos. before any proper element

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

// init to empty sequence



// Simple access functions

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

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

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

⋮

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

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

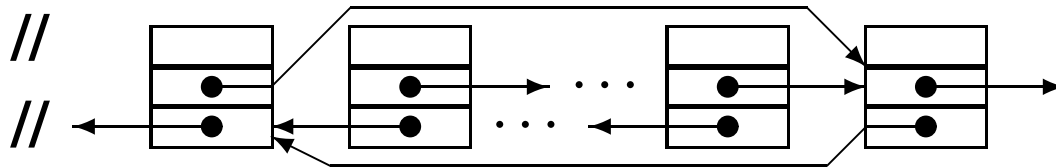
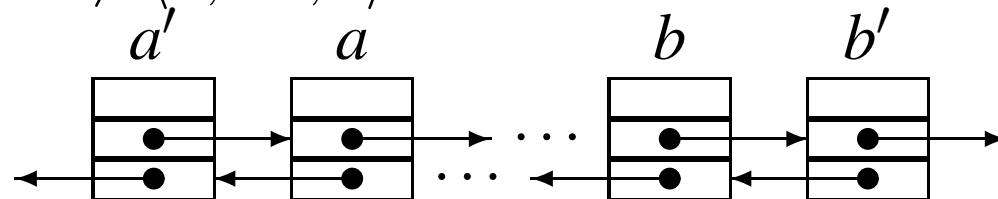
// Cut out $\langle a, \dots, b \rangle$

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

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

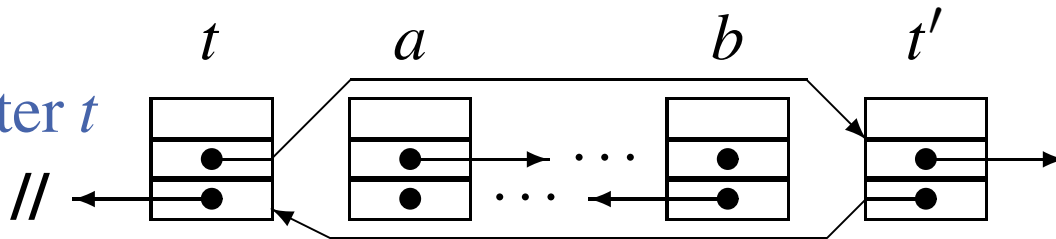
$a' \rightarrow \text{next} := b'$

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



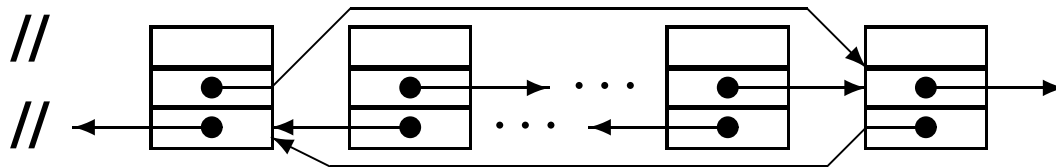
// insert $\langle a, \dots, b \rangle$ after t

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



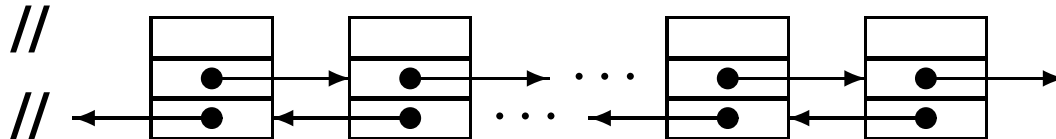
$b \rightarrow \text{next} := t'$

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



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

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



Der Rest sind Einzeiler (?)

// Moving elements around within a sequence.

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

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

Procedure moveToFront($b : \text{HANDLE}$) moveAfter(b, HEAD)

Procedure moveToBack($b : \text{HANDLE}$) moveAfter(b, LAST)

Oder doch nicht? Speicherverwaltung!

naiv / blauäugig / optimistisch:

Speicherverwaltung der Programmiersprache

~> potentiell sehr langsam

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

FREELIST enthält ungenutzte Items.

CHECKFREELIST stellt sicher, dass die nicht leer ist.

Reale Implementierungen:

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

Items löschen

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

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

Procedure popFront remove(FIRST)

Procedure popBack remove(LAST)

Elemente einfügen

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

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

checkFreeList // make sure FREELIST is nonempty.

$a' := \text{FREELIST.FIRST}$ // 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($e, b \rightarrow \text{prev}$)

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

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

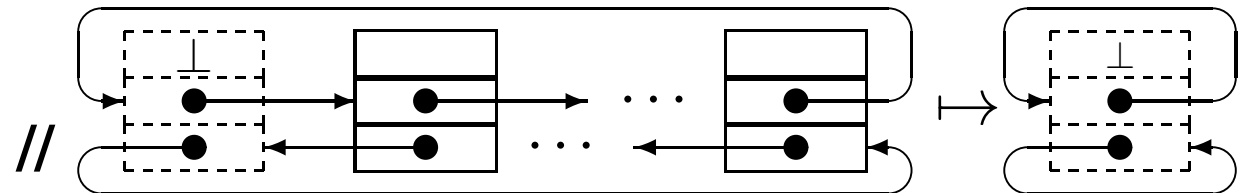
Ganze (Teil)Listen Manipulieren

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

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

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

Procedure makeEmpty
 freeList.concat(**this**)



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

Suchen

Trick: gesuchtes Element in Dummy-Item schreiben:

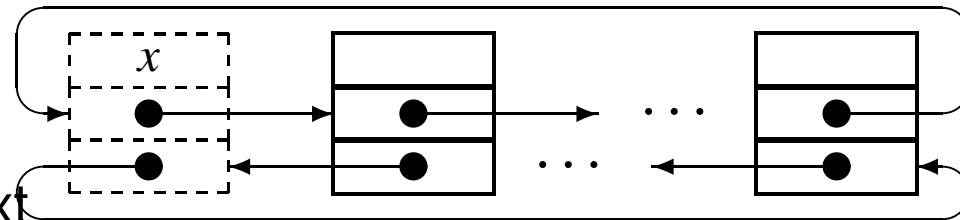
Function findNext(x : Element; from : Handle) : Handle

$h.e = x$ // Sentinel

while FROM $\rightarrow e \neq x$ **do**

FROM := FROM \rightarrow next

return from



Spart Sonderfallbehandlung.

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

\rightsquigarrow einfacher, schneller, ...

Funktionalität \leftrightarrow Effizienz

Beispiel: Listenlängen

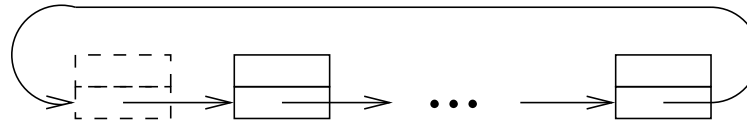
Verwalte zusätzliches Member **SIZE**.

Problem: inter-list **SPLICE** geht nicht mehr in konstanter Zeit

Die Moral von der Geschichte:

Es gibt nicht DIE Listenimplementierung.

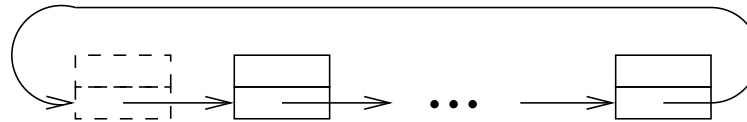
3.1.2 Einfach verkettete Listen



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
 $E = \{(u, v) : u \in \text{ITEM}, v = u \rightarrow \text{NEXT}\}$

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

Folge: Items bilden Kollektion von Kreisen

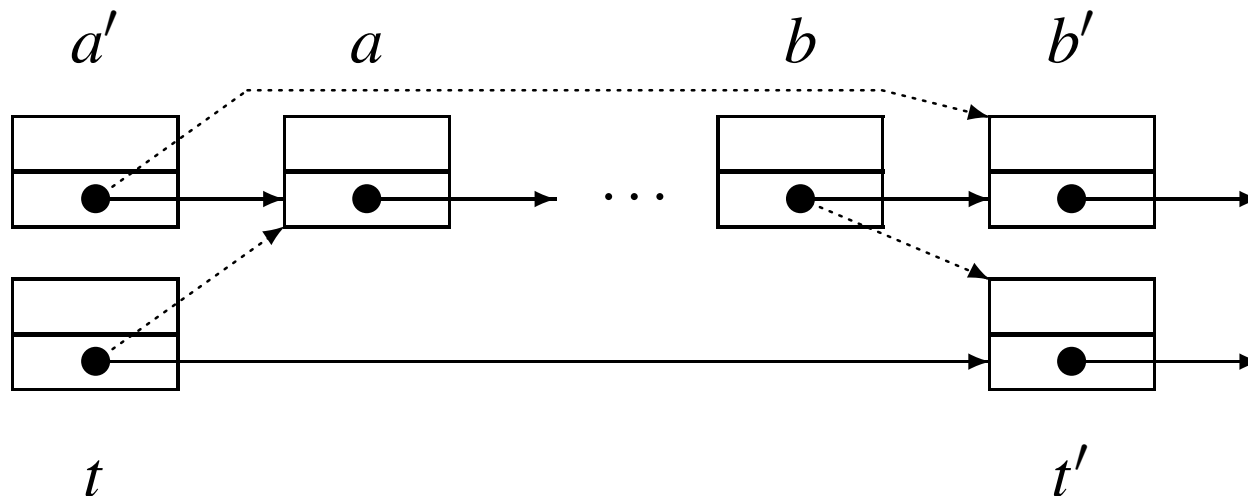
Einfach verkettete Listen – SPLICE

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

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

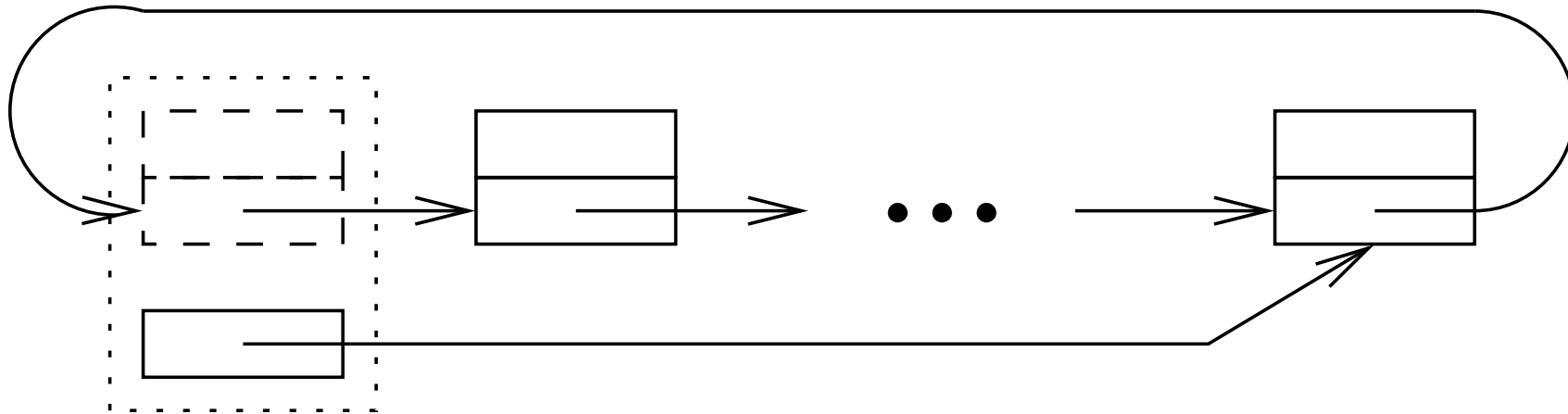
Procedure splice(a', b, t : SHandle)

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



Einfach verkettete Listen – PUSHBACK

Zeiger auf letztes Item erlaubt Operation **PUSHBACK**



Listen: Zusammenfassung, Verallgemeinerungen

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

Felder (Arrays)

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

Beschränkte Felder (Bounded Arrays)

Eingebaute Datenstruktur: Ein Stück Hauptspeicher +
Adressrechnung

Größe muss von Anfang an bekannt sein

3.2 Unbeschränkte Felder (Unbounded Arrays)

$$\langle e_0, \dots, e_n \rangle . \text{PUSHBACK}(E) \rightsquigarrow \langle e_0, \dots, e_n, e \rangle ,$$

$$\langle e_0, \dots, e_n \rangle . \text{POPBACK} \rightsquigarrow \langle e_0, \dots, e_{n-1} \rangle ,$$

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

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

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

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

// $b \rightarrow$ 

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

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**

 REALLOCATE($2n$)

$b[n] := e$

$n++$

// $b \rightarrow$

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

// $b \rightarrow$

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

// $b \rightarrow$

0	1	2	3	e			
---	---	---	---	-----	--	--	--

// $b \rightarrow$

0	1	2	3	e			
---	---	---	---	-----	--	--	--

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

$w := w'$

// $b \rightarrow$

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

$b' :=$ **allocate**

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

--	--	--	--	--	--	--	--

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

$(b[0], \dots, b[n - 1])$

// $b' \rightarrow$

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

von **PUSHBACK** oder **POPBACK** Operationen auf u

wird in **Zeit** $O(m)$ ausgeführt.

Sprechweise:

PUSHBACK und POPBACK haben **amortisiert** konstante

Ausführungszeit —

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

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

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

Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten	Typ
PUSHBACK	○○ (2 Token)	einzahlen
POPBACK	○ (1 Token)	einzahlen
REALLOCATE($2n$)	$n \times \circ$ (n Token)	abheben

Weitere Aufrufe von REALLOCATE:

$$\text{rauf: REALLOCATE}(2n) \underbrace{\geq n \times \text{PUSHBACK}}_{\geq n \times \circ\circ} \text{REALLOCATE}(4n)$$

$$\text{runter: REALLOCATE}(2n) \underbrace{\geq n/2 \times \text{POPBACK}}_{\geq n/2 \times \circ} \text{REALLOCATE}(n)$$

□

3.3 Amortisierte Analyse – allgemeiner

- Z : Menge von Operationen, z. B. $\{\text{PUSHBACK}, \text{POPBACK}\}$
- 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**: $s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \dots \xrightarrow{Op_n} s_n$

Die angenommenen amortisierten Kosten sind korrekt, wenn

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

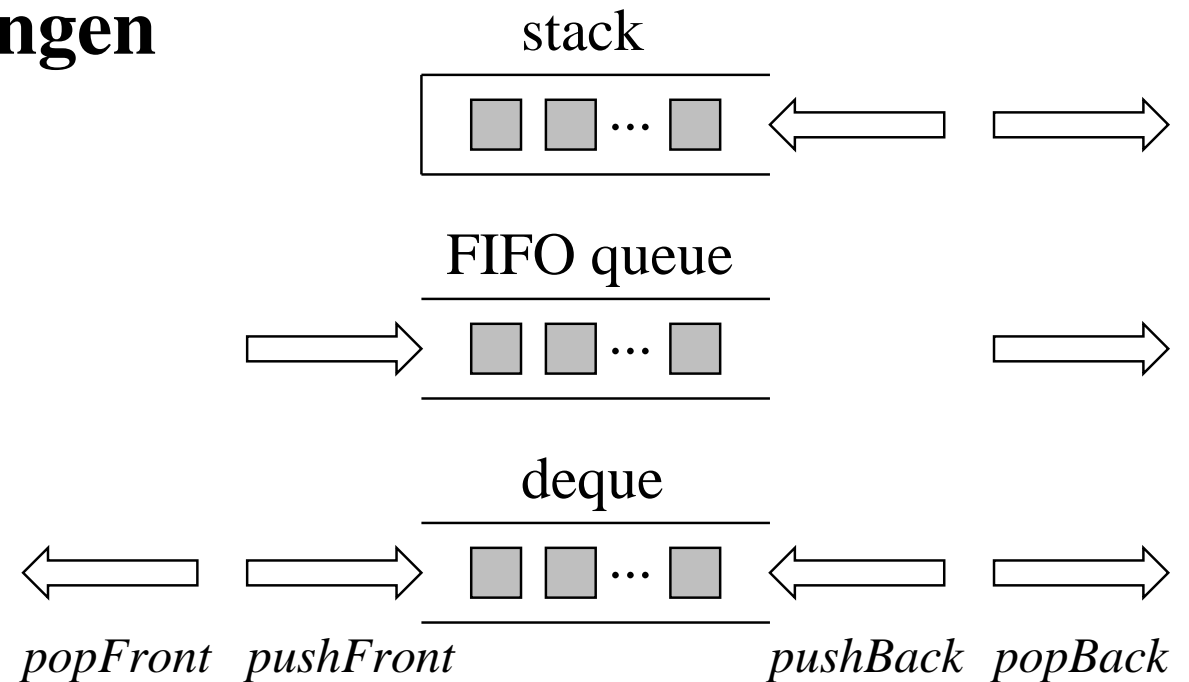
für eine Konstante c

Amortisierte Analyse – Diskussion

- ☐ Amortisierte Laufzeiten sind leichter zu garantieren als tatsächliche.
- ☐ Der Gesamtlauftzeit tut das keinen 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

$b : \mathbf{Array} [0..n] \mathbf{of}$ ELEMENT

$h=0 : \mathbb{N}$

$t=0 : \mathbb{N}$

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

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

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

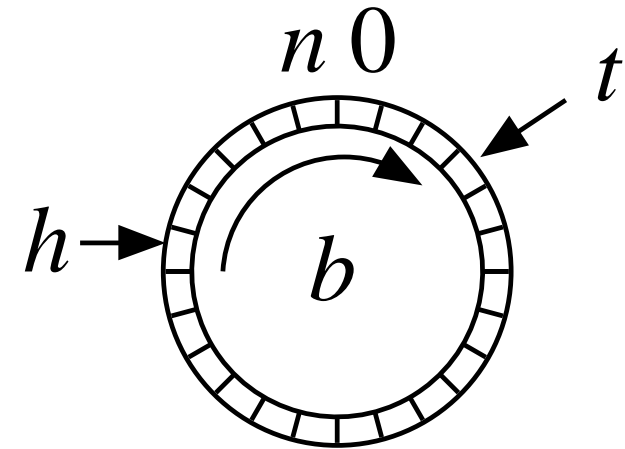
Procedure pushBack($x : \text{Element}$)

assert size $< n$

$b[t] := x$

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

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



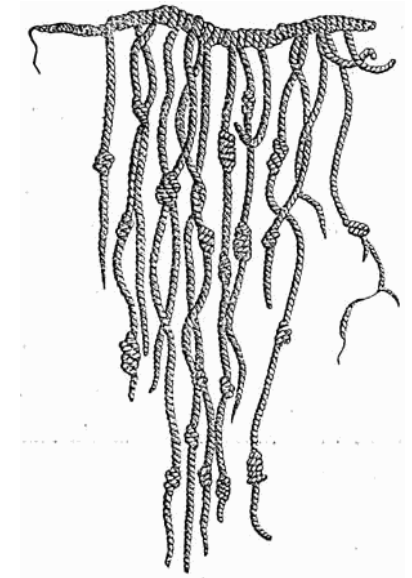
3.5 Vergleich: Listen – Felder

Vorteile von Listen

- ☐ flexibel
- ☐ REMOVE, SPLICE, . . .
- ☐ kein Verschnitt

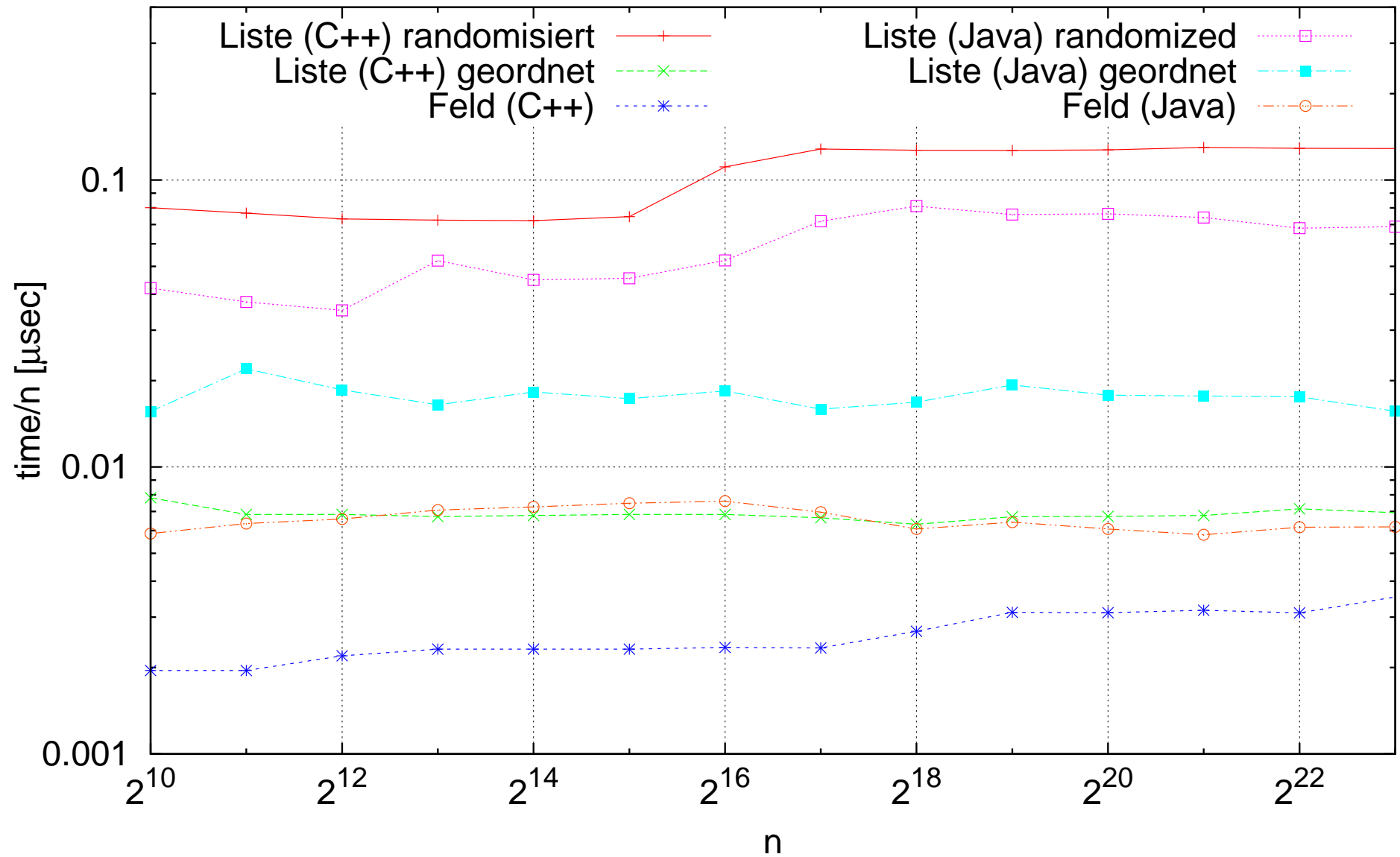
Vorteile von Feldern

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

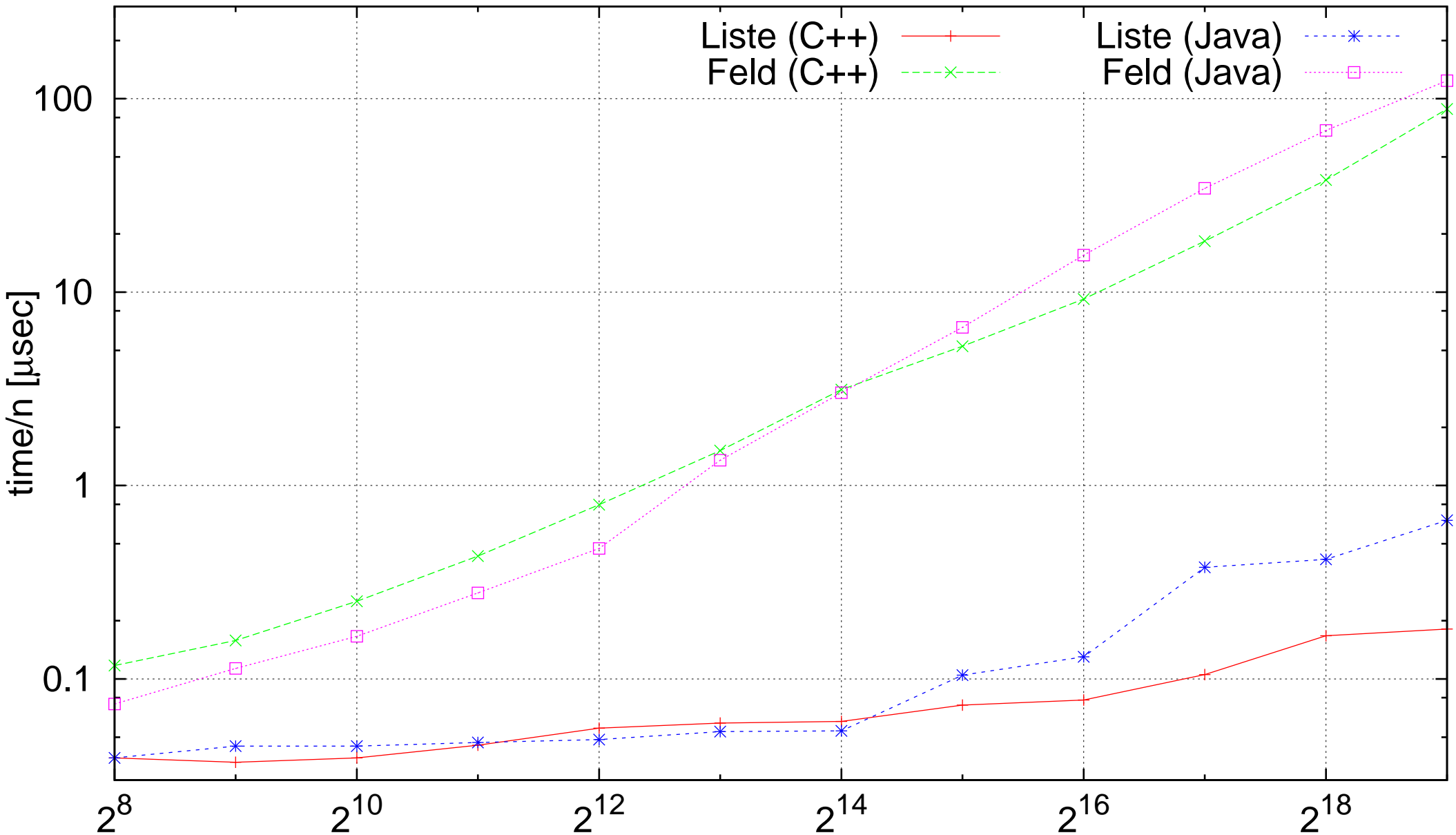


Operation	LIST	SLIST	UARRAY	CARRAY	explanation ‘*’
$[\cdot]$	n	n	1	1	
$ \cdot $	1^*	1^*	1	1	not with inter-list SPLICE
FIRST	1	1	1	1	
LAST	1	1	1	1	
INSERT	1	1^*	n	n	INSERTAFTER only
REMOVE	1	1^*	n	n	REMOVEAFTER only
PUSHBACK	1	1	1^*	1^*	amortized
PUSHFRONT	1	1	n	1^*	amortized
POPBACK	1	n	1^*	1^*	amortized
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



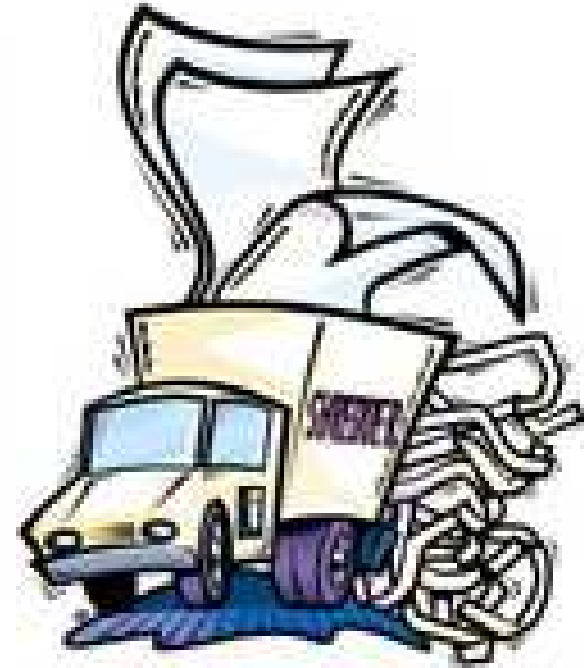
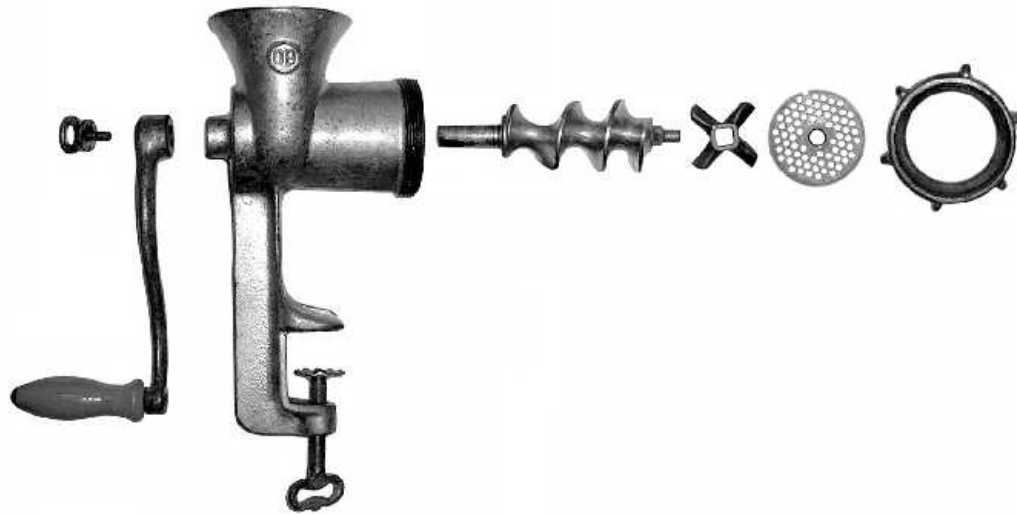
Ausblick: Weitere Repräsentationen von Folgen

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**” \approx “völlig **durcheinander** bringen”.

Paradoxerweise **hilft** das, Dinge wiederzufinden

Hashtabellen

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

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

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

$M.\text{INSERT}(e : \text{ELEMENT})$: $M := M \cup \{e\}$

$M.\text{REMOVE}(k : \text{KEY})$: $M := M \setminus \{e\}, e = k$

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

Anderes Interface: **map/partielle Funktion** $\text{KEY} \rightarrow \text{ELEMENT}$

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

Exkurs: Konventionen für Elemente

Viele Datenstrukturen repräsentieren Mengen

(engl. auch collection classes).

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

Elementvergleich hier gleichbedeutend mit Schlüsselvergleich.

$e < / > / = e'$ gdw. $\text{KEY}(e) < / > / = \text{KEY}(e')$.

Hashing: Anwendungen

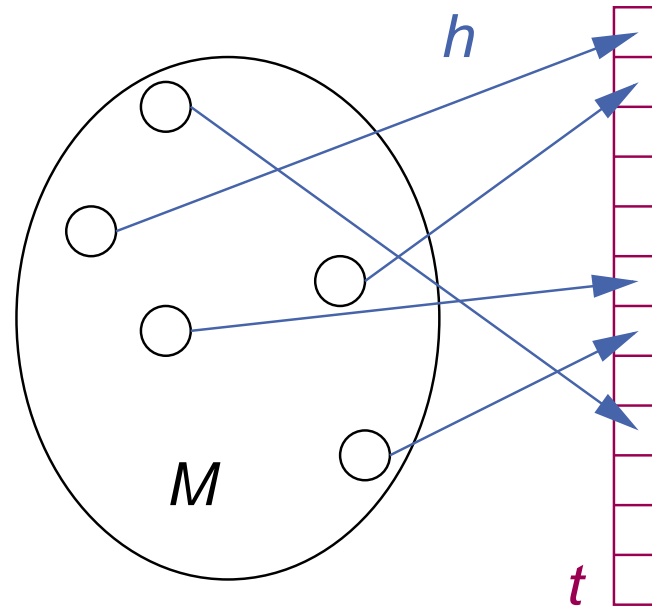
- ☐ Auslieferungsregale der UB Heidelberg
- ☐ Entfernen exakter **Duplikate**
- ☐ Schach (oder andere kombinatorische Suchprogramme):
welche Stellungen wurden **bereits durchsucht**?
- ☐ **Symboltabelle** bei Compilern
- ☐ **Assoziative Felder** bei Script-Sprachen wie PERL oder AWK
- ☐ Datenbank-Gleichheits-**Join**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

Überblick

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

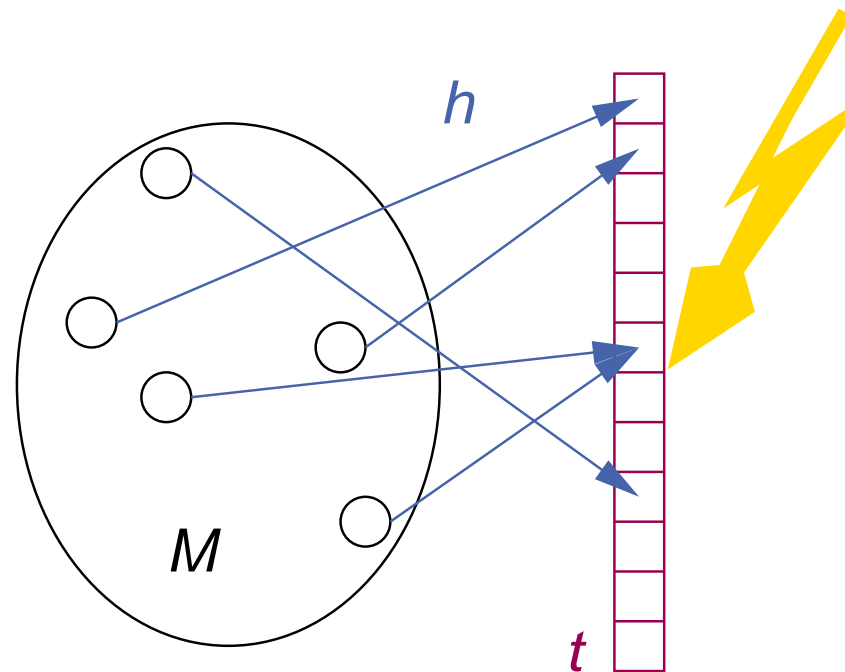
Ein (über)optimistischer Ansatz

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

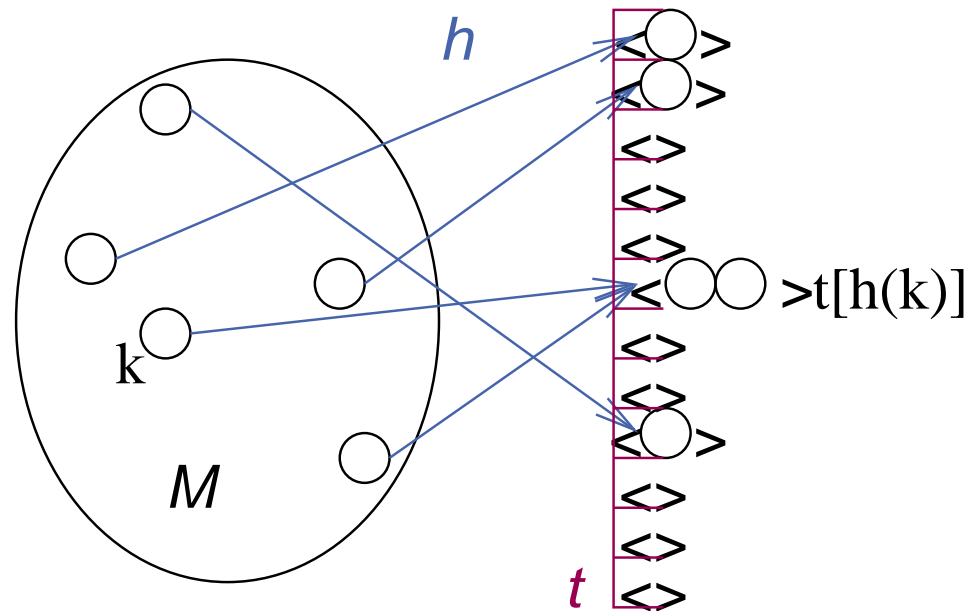


Beispiel: Geburtstagsparadox

Kollisionsauflösung

Beispiel **geschlossenes Hashing**

Tabelleneinträge: Elemente \rightsquigarrow **Folgen** von Elementen



4.1 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 $\text{KEY}(e) = k$ gefunden?

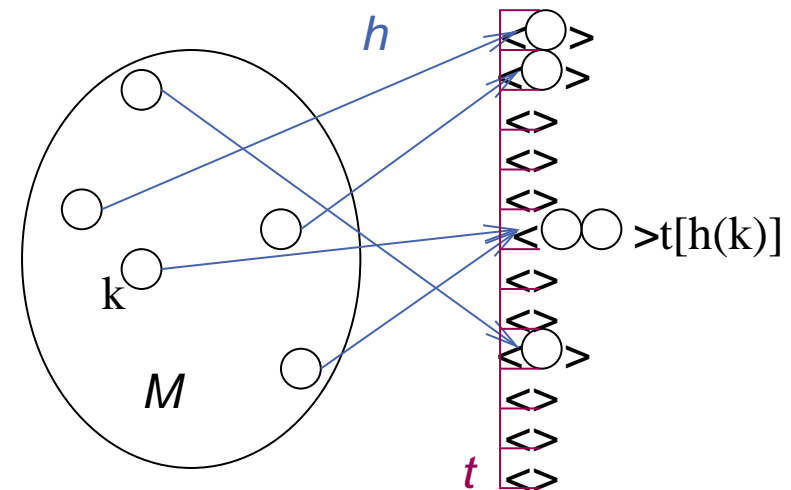
\rightsquigarrow löschen und zurückliefern.

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

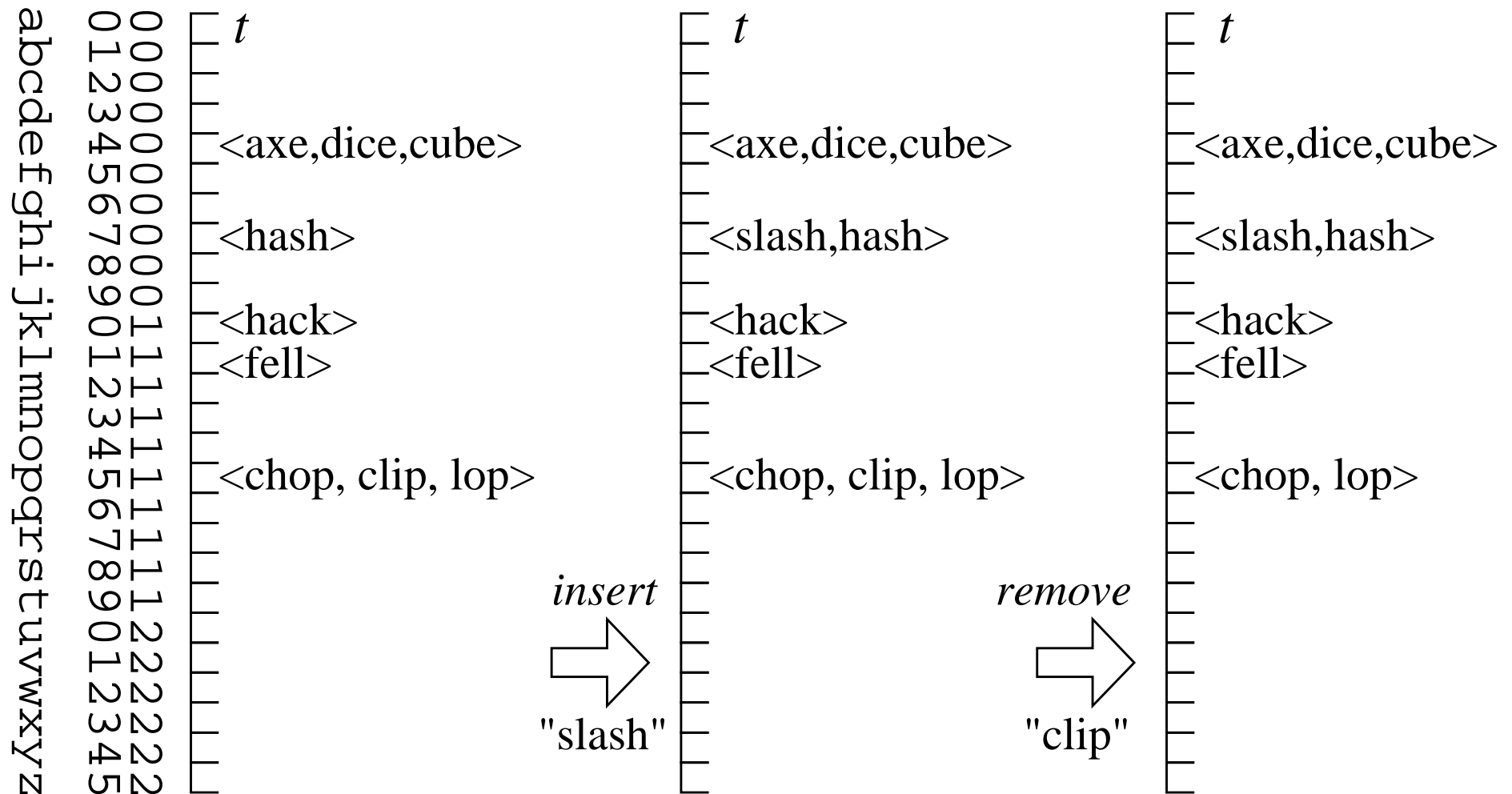
Element e mit $\text{KEY}(e) = k$ gefunden?

\rightsquigarrow zurückliefern.

Sonst: \perp zurückgeben.



Beispiel



Analyse

INSERT(e): konstante Zeit

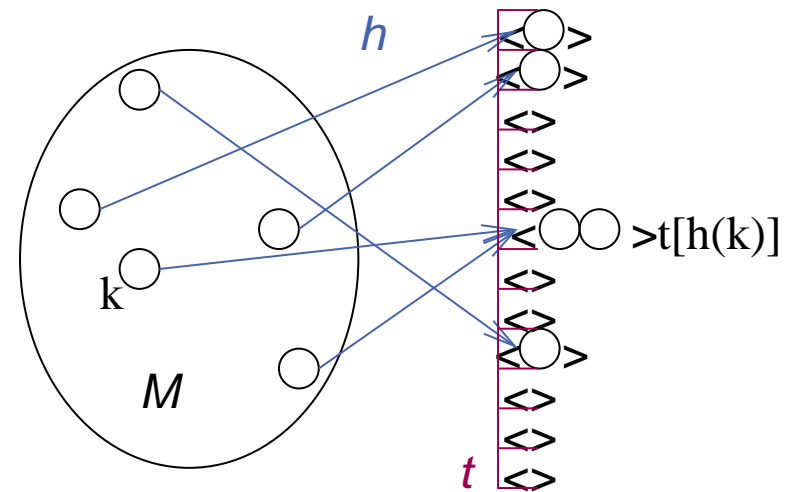
REMOVE(k): $O(\text{LISTENLÄNGE})$

FIND(k) : $O(\text{LISTENLÄNGE})$

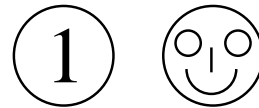
Aber wie lang werden die Listen?

Schlechtester Fall: $O(|M|)$

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

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

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

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

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

Zufallsvariable (ZV) $X_0 : \Omega \rightarrow \mathbb{R}$

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

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

Erwartungswert $E[X_0] = \sum_{y \in \Omega} p_y X(y)$

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

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

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: $X = \sum_{i=1}^n \sum_{j=i+1}^n I_{ij}$.

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

Mehr zum Geburtstagsparadoxon

Standardformulierung:

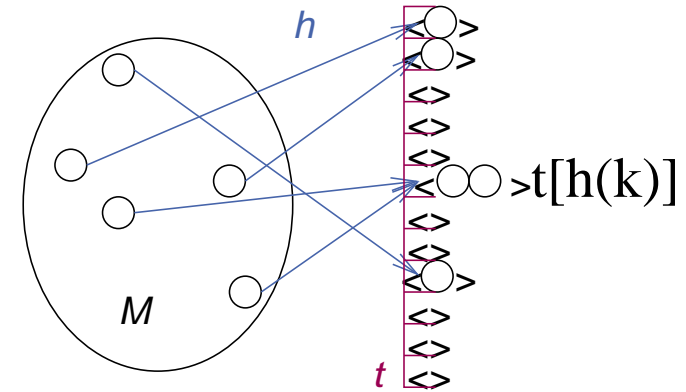
Ab wann lohnt es sich zu **wetten**, dass es zwei Gäste mit gleichem Geburtstag gibt? Etwas komplizierter. Antwort: $n \geq 23$

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

Riesige Platzverschwendung.

Analyse für zufällige Hash-Funktionen

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



Beweis. Für festen Schlüssel k definiere **Kollisionslänge X**

$X := |t[h(k)]| = |\{e \in M' : h(e) = h(k)\}|$ mit

$M' = \{e \in M : \text{KEY}(e) \neq k\}$.

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

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

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



Zufällige Hash-Funktionen?

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

⇒ **meist zu teuer**

Weniger naive Lösungen: kompliziert, immer noch viel Platz.

⇒ **meist unsinnig**

Zufällige Schlüssel?

⇒ **unrealistisch**

4.2 Universelles Hashing

Idee: nutze nur bestimmte “einfache” Hash-Funktionen

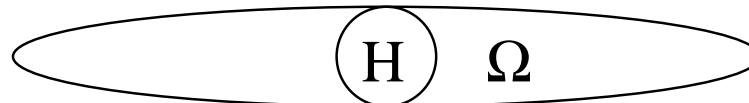
Definition 2. $\mathcal{H} \subseteq \{0..m-1\}^{\text{KEY}}$ ist *universell* falls für alle x, y in KEY mit $x \neq y$ und zufälligem $h \in \mathcal{H}$,

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

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

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

Der Rest geht wie vorher. □



Eine einfache universelle Familie

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

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

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

H ist eine universelle Familie von Hash-Funktionen

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

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

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

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

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

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

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



Bit-basierte Universelle Familien

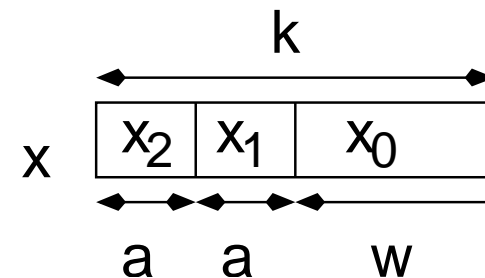
Sei $m = 2^w$, $\text{KEY} = \{0, 1\}^k$

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

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

Tabellenzugriff: $H^{\oplus \square} = \left\{ h_{(t_1, \dots, t_b)}^\oplus : t_i \in \{0..m-1\}^{\{0..w-1\}} \right\}$

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



4.3 Hashing mit Linearer Suche (Linear Probing)

Offenes Hashing: zurück zur Ursprungsidee.

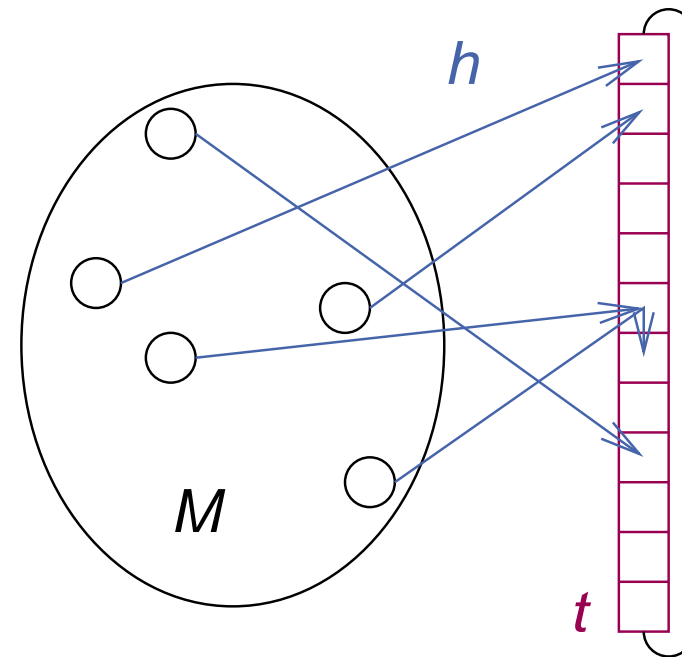
Elemente werden direkt in der Tabelle gespeichert.

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

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

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

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

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

assert $i < m + m' - 1$

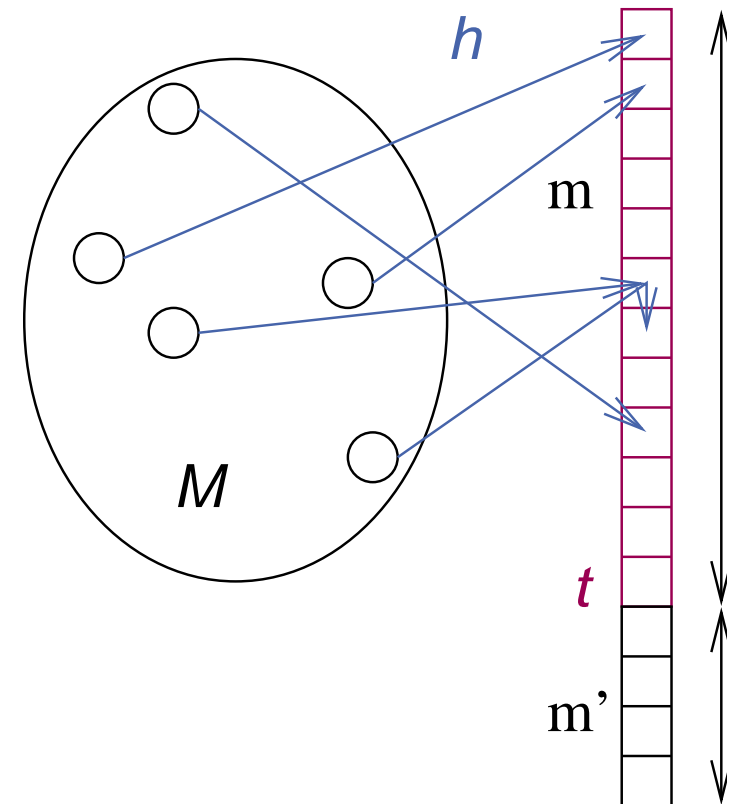
$t[i] := e$

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

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

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

return \perp



Remove

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

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

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

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

if $t[i] = \perp$ **then return** // nothing to do

// we plan for a hole at i .

for ($j := i + 1$; $t[j] \neq \perp$; $j++$)

// Establish invariant for $t[j]$.

if $h(t[j]) \leq i$ **then**


$t[i] := t[j]$ // Overwrite removed element

$i := j$ // move planned hole

$t[i] := \perp$ // erase freed entry

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

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

remove  clip

⊥	⊥	chop	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	⊥

4.4 Verketteten \leftrightarrow Lineare Suche

Volllaufen: Verketteten 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, Implementierungsdetails bei Verketteten
(shared dummy!, t speichert Zeiger oder item),
Speicherverwaltung bei Verketteten, beschränkt oder nicht,...

Referentielle Integrität: Nur bei Verketteten !

Leistungsgarantien: Universelles Hashing funktioniert so nur mit
Verketteten

4.5 Perfektes Hashing

hier nicht

Mehr Hashing

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

5 Sortieren & Co



Formaler

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

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

- ☐ s' ist Permutation von s
- ☐ $e'_1 \leq \dots \leq e'_n$ für eine 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.

Überblick

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

5.1 Einfache Sortialgorithmen

Sortieren durch Einfügen (insertion sort)

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

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

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

move $a[i]$ to the right place

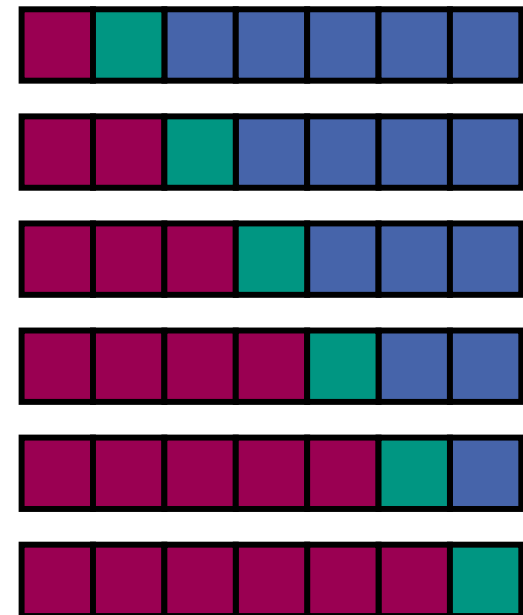
Beispiel:

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

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

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

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



Sentinels am Beispiel Sortieren durch Einfügen

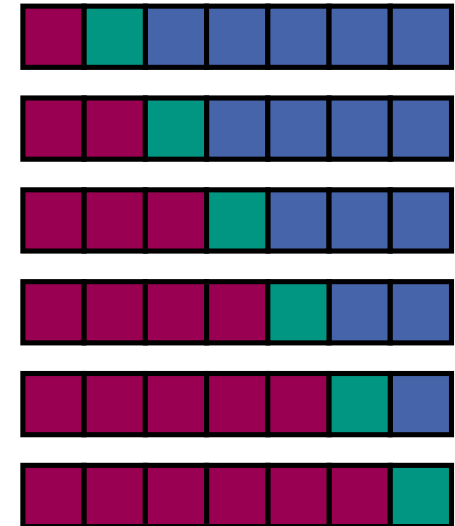
```
Procedure insertionSort( $a$  : Array  $[1..n]$  of Element)
  for  $i := 2$  to  $n$  do
    invariant  $a[1] \leq \dots \leq a[i-1]$ 
    // move  $a[i]$  to the right place
     $e := a[i]$ 
    if  $e < a[1]$  then                                     // new minimum
      for  $j := i$  downto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


Analyse

Schlechtester Fall

Die i -te Iteration braucht Zeit $O(i)$.

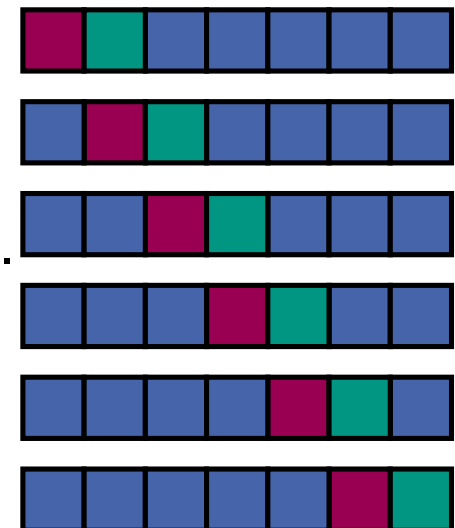
$$\sum_{i=2}^n i = \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$



Bester Fall

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

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



5.2 Sortieren durch Mischen

Idee: Teile und Herrsche

Function mergeSort($\langle e_1, \dots, e_n \rangle$) : Sequence **of** Element

if $n = 1$ **then return** $\langle e_1 \rangle$ // base case

else return merge(mergeSort($\langle e_1, \dots, e_{\lfloor n/2 \rfloor} \rangle$),
mergeSort($\langle e_{\lfloor n/2 \rfloor + 1}, \dots, e_n \rangle$))

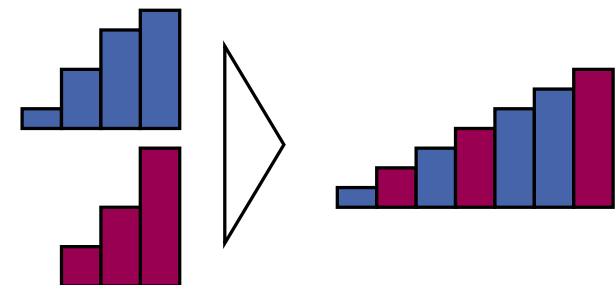
Mischen (merge)

Gegeben:

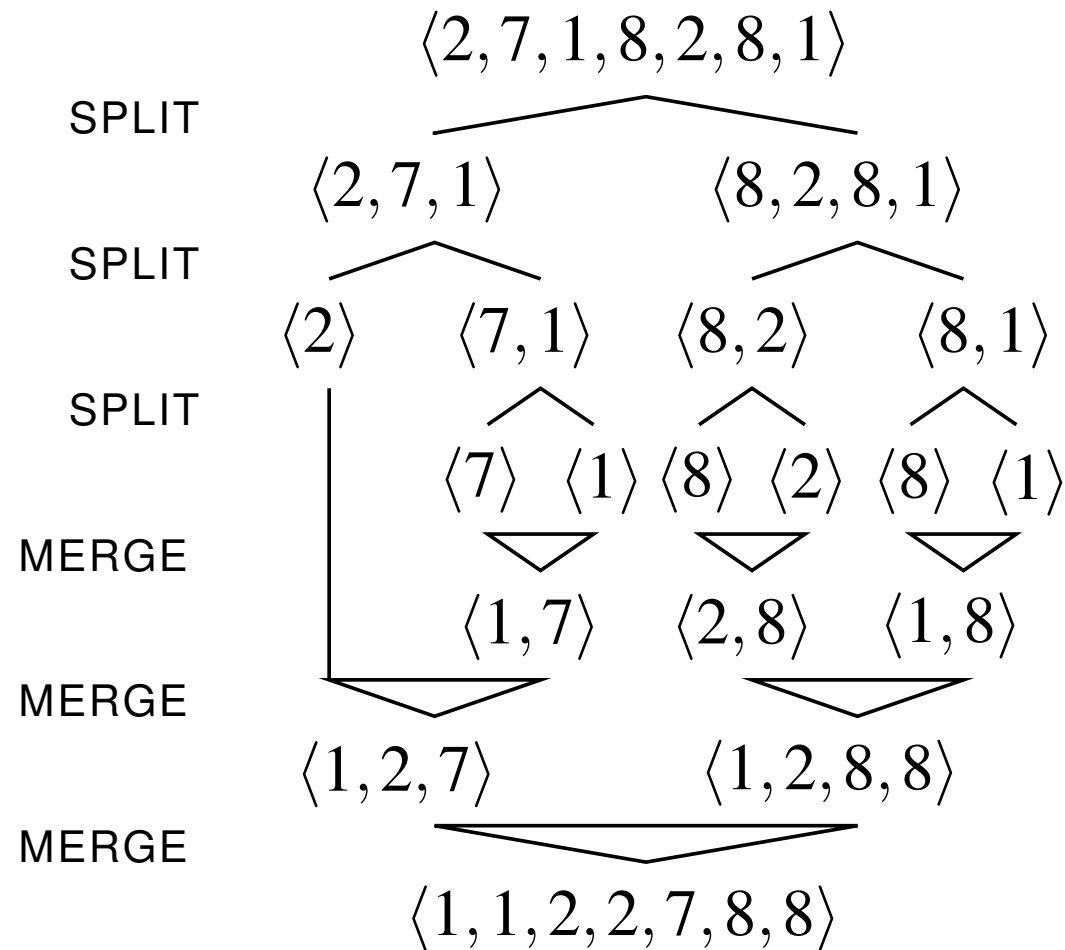
zwei **sortierte Folge** a und b

Berechne:

sortierte Folge der Elemente aus a und b



Beispiel



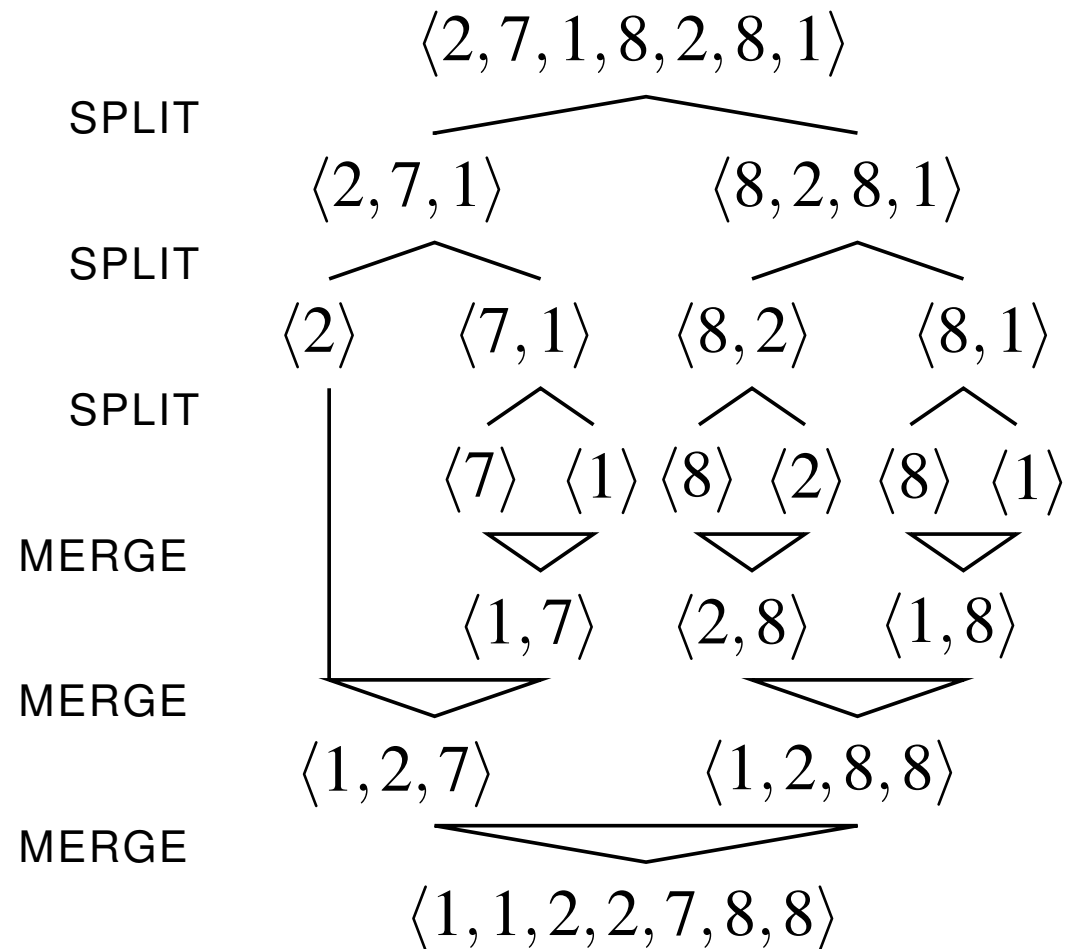
Mischen

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

Analyse

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

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 $O(n \log n)$

5.3 Untere Schranken

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

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

~>

einschränkende Annahmen

Eine vergleichsbasierte untere Schranke

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

Satz: Deterministische vergleichsbasierte Sortieralgorithmen
brauchen

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

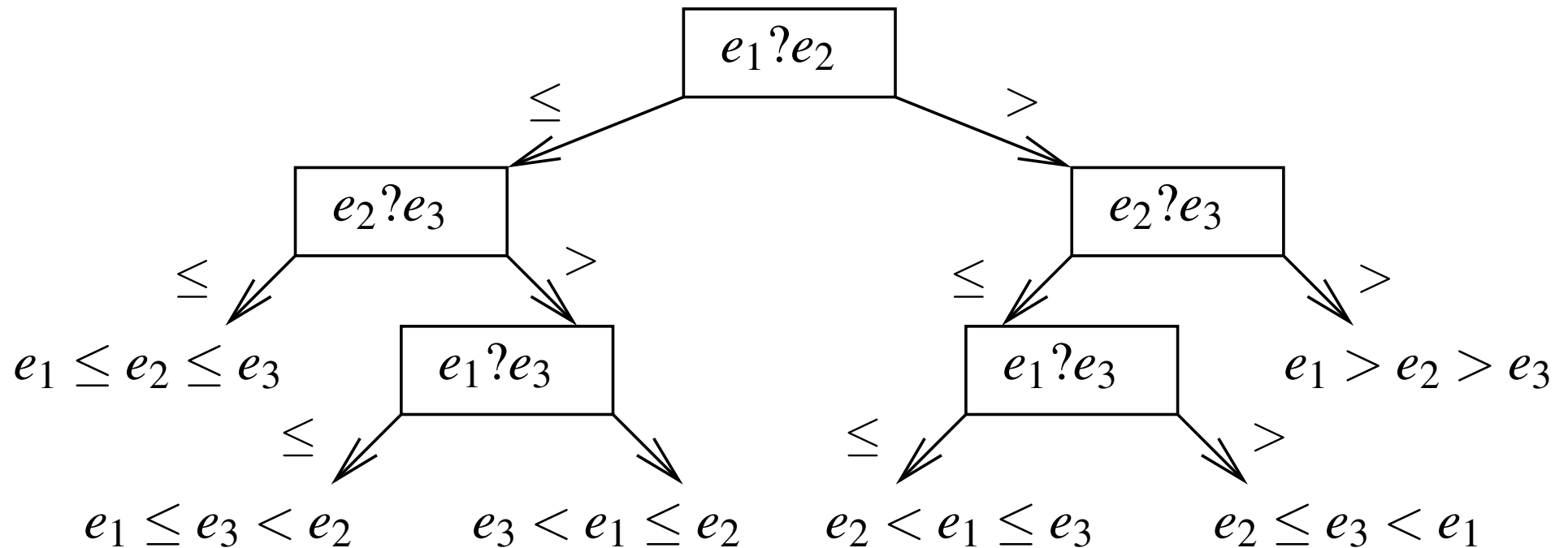
Vergleiche im schlechtesten Fall.

Beweis:

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

Es gibt genau $n!$ solche Permutationen.

Baumbasierte Sortierer-Darstellung



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

Ausführungszeit entspricht **Tiefe T**

Beweis

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

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

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

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

Beweis für **linken Teil**:

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

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

Randomisierung, Mittlere Ausführungszeit

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

Beweis: nicht hier.

5.4 Quicksort – erster Versuch

Idee: Teile-und-Herrsche aber verglichen mit mergesort „andersrum“.

Leiste Arbeit **vor** rekursivem Aufruf

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

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

pick **“some”** $p \in s$

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

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

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

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

Quicksort – Analyse im schlechtesten Fall

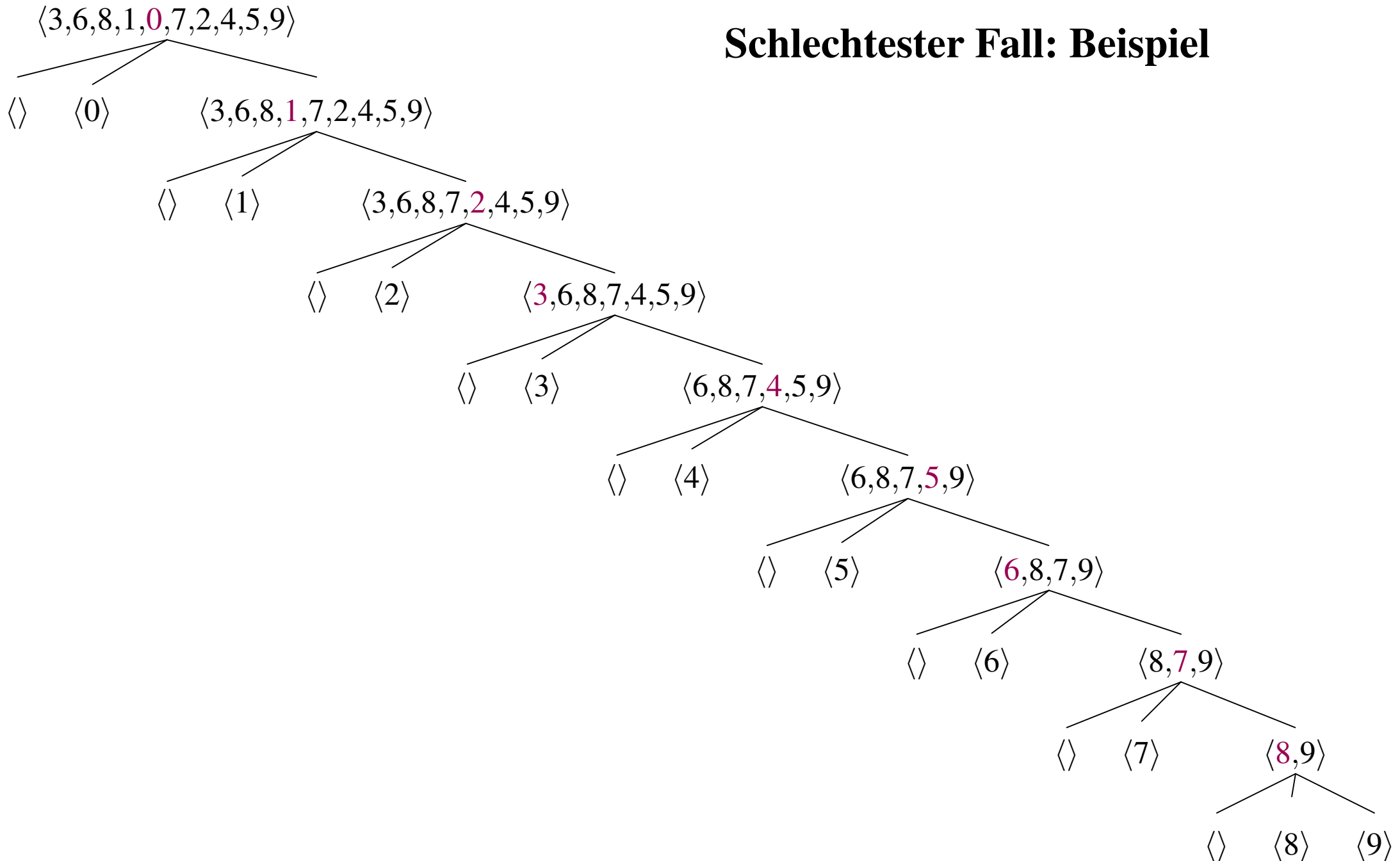
Annahme: Pivot ist immer Minimum (oder Max.) der Eingabe

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

\Rightarrow

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

Schlechtester Fall: Beispiel



Quicksort – Analyse im **besten** Fall

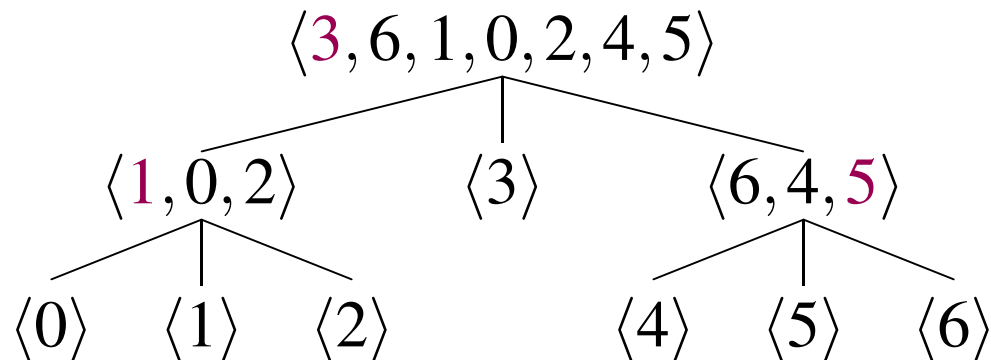
Annahme: Pivot ist immer **Median** der Eingabe

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

\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

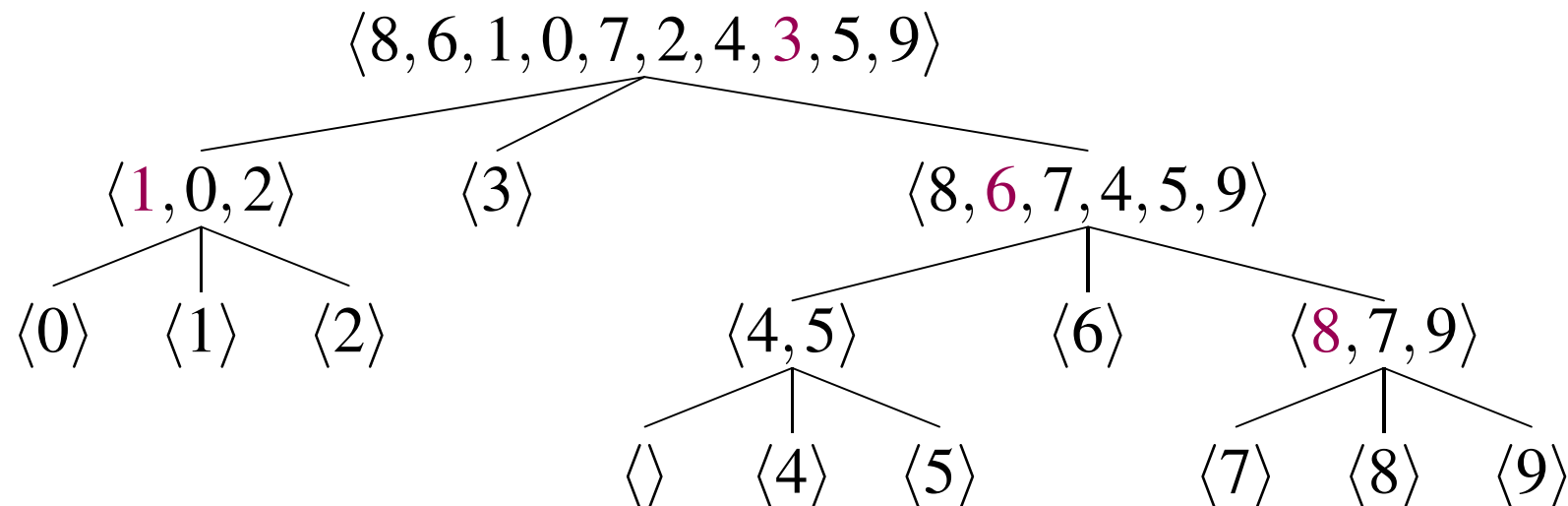
pick $p \in s$ uniformly at random

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

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

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

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



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

Annahme: alle Elemente verschieden

Warum 'OBdA'?

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

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

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

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

pick $p \in s$ uniformly at random

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

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

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

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

Beweisansatz 1: Rekurrenzen

Beweis:

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

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

Beweisansatz 2: Genauere, elegantere Analyse

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

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

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

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

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

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

Sortierte Eingabefolge:

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

$$X_{ij} = 1$$

\Leftrightarrow

e'_i wird mit e'_j verglichen

\Leftrightarrow

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

\Rightarrow

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

□

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

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

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

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

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

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

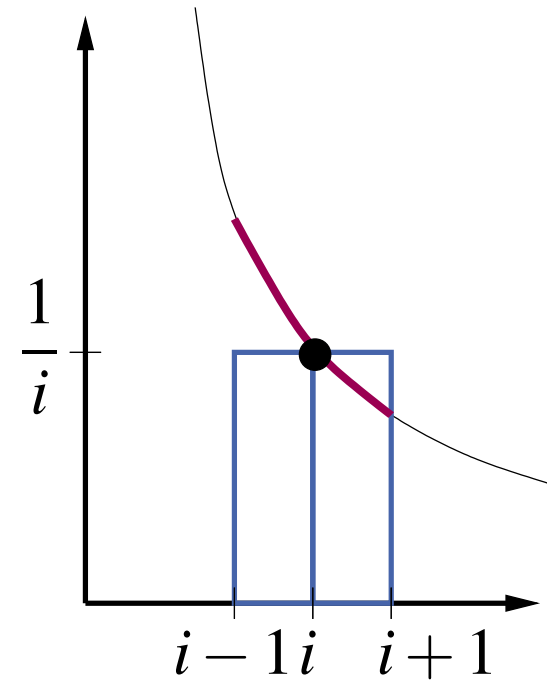
i	j	$\overbrace{j-i+1}^{=:k}$
1	2..n	2..n
2	3..n	2..n-1
3	4..n	2..n-2
\vdots	\vdots	\vdots
$n-1$	$n..n$	2..2
n	\emptyset	\emptyset

(harmonische Summe)

□

Exkurs: Harmonische Summe

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



Also

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

5.4.2 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)

ℓ									r
i	\rightarrow							\leftarrow	j
3	6	8	1	0	7	2	4	5	9
2	6	8	1	0	7	3	4	5	9
2	0	8	1	6	7	3	4	5	9
2	0	1	8	6	7	3	4	5	9
ℓ	\leftrightarrow	j	i			\leftrightarrow			r

stop

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 : \mathbb{N})

if $r - \ell + 1 \leq n_0$ **then** INSERTIONSORT($a[\ell..r]$)

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

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

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($a[i], a[j]$); $i++$; $j--$

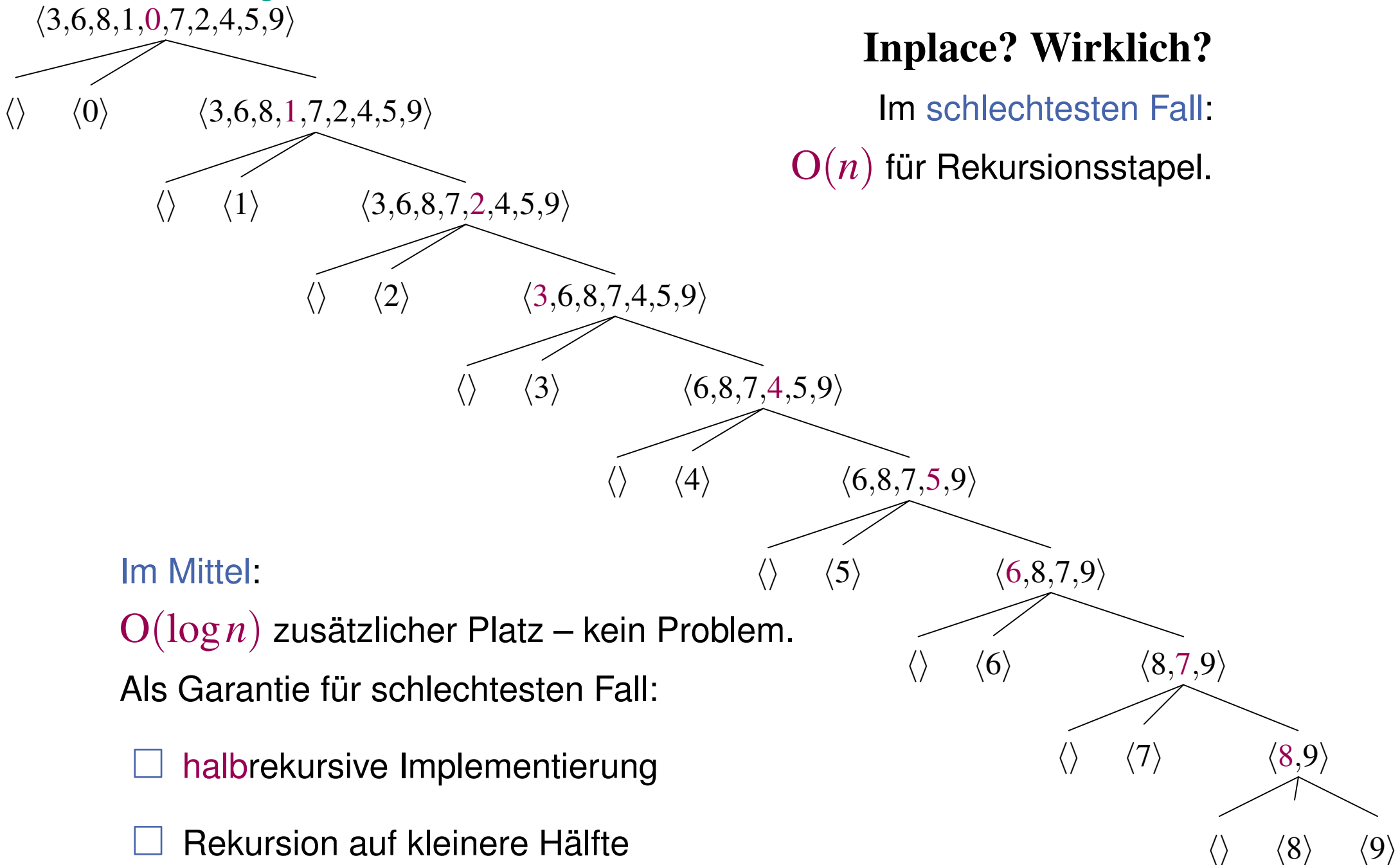
until $i > j$

// a :

ℓ	\leftrightarrow	j	i	\leftrightarrow	r
--------	-------------------	-----	-----	-------------------	-----

QSORT(a, ℓ, j)

QSORT(a, i, r)



Halbrekursive Implementierung

Procedure QSORT(a : **Array of** ELEMENT; ℓ, r : \mathbb{N})

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

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

// a : $\boxed{\ell \quad \leftrightarrow \quad j \quad i \quad \leftrightarrow \quad r}$

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; ℓ, r : \mathbb{N})

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

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

// a : $\boxed{\ell \quad \leftrightarrow \quad j \quad i \quad \leftrightarrow \quad r}$

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

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

qsort: möglich bei sorgfältiger Pivotwahl

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

qsort: leicht bei Aufgabe der inplace-Eigenschaft

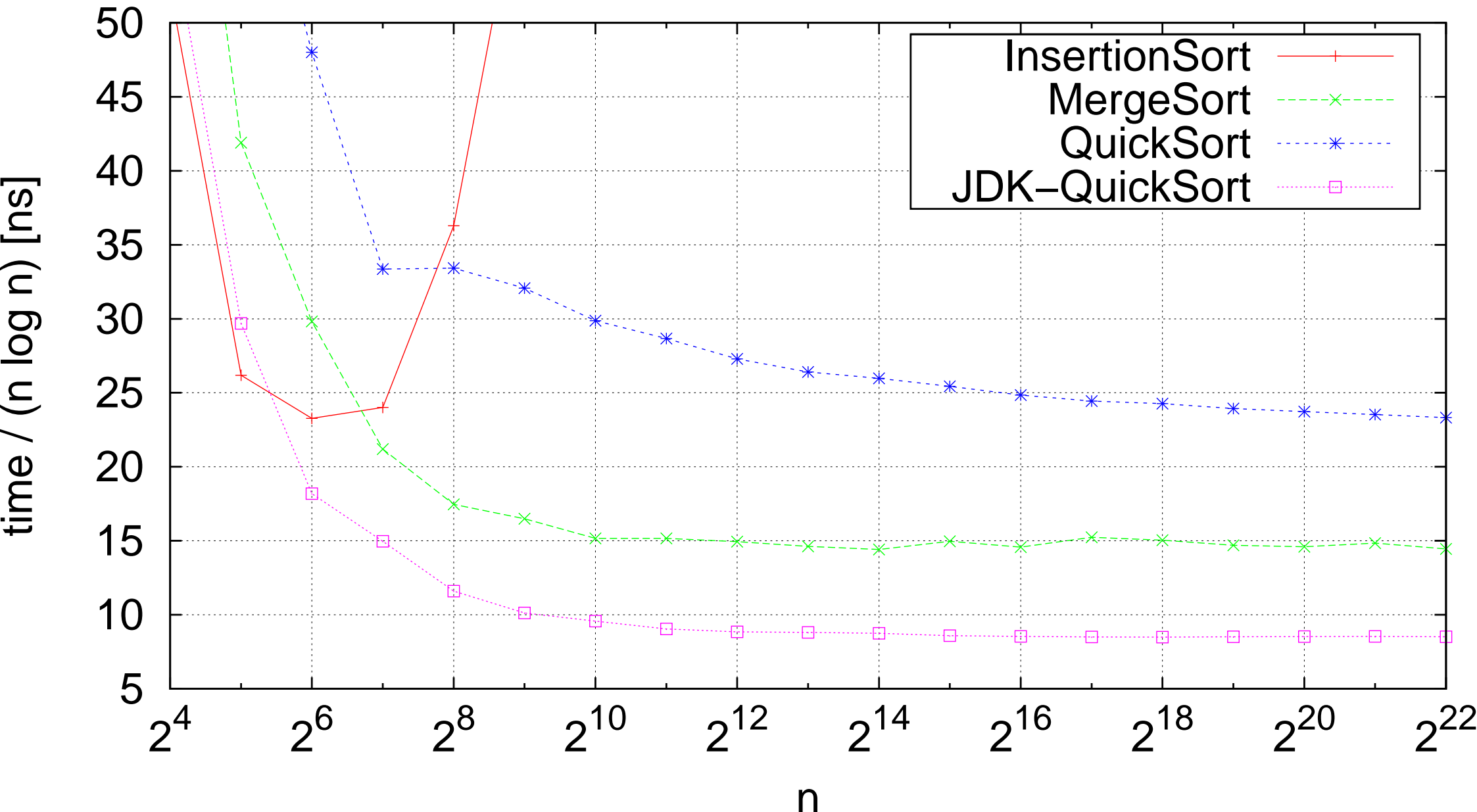
Pro Quicksort

- ☐ **inplace**

- ☐ Etwas schneller?

Benchmark

Sortieren einer zufaelligen Sequenz (int)



5.5 Auswahl (Selection)

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

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

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

Grob: $a[i]$ 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 : \mathbb{N}) : Element

assert $|s| \geq k$

Auswahl – Anwendungen

Statistik

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

Unterprogramm

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

Quickselect

≈ quicksort mit einseitiger Rekursion

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

assert $|s| \geq k$

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

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

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

a

 k

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

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

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

 k

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

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

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

 k

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

 k

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

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

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

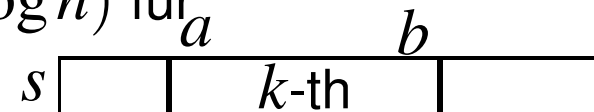
 k

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, ...
Beispiel: Optimale **Range Median** Berechnung
[B. Gfeller, P. Sanders, ICALP 2009].
Vorberechnungszeit $O(n \log n)$, Zeit $O(\log n)$ für
SELECT($\langle s[a], \dots, s[b] \rangle, k$)



5.6 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)

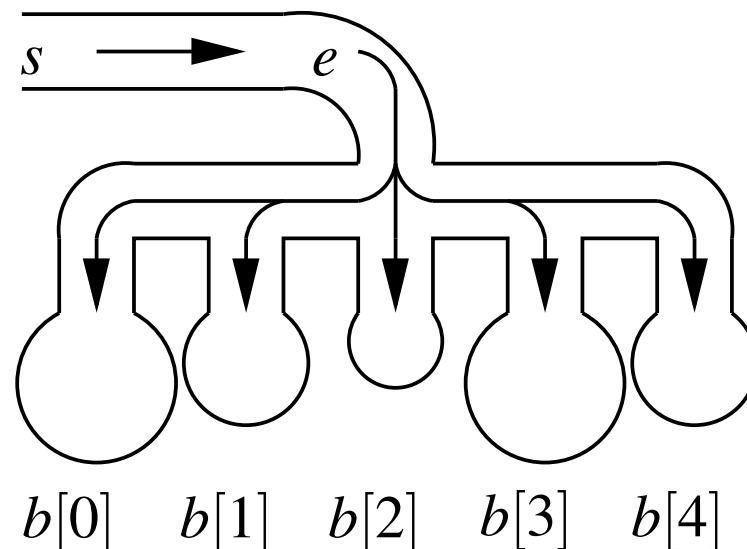
Procedure KSort(s : Sequence **of** Element)

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

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

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

Zeit: $O(n + K)$



Beispiel: $K = 4$ **Procedure** KSort(s : Sequence **of** Element) $b = \langle \langle \rangle, \dots, \langle \rangle \rangle$: **Array** $[0..K - 1]$ **of** Sequence **of** Element**foreach** $e \in s$ **do** $b[\text{KEY}(e)].\text{PUSHBACK}(e)$ $s :=$ concatenation of $b[0], \dots, b[K - 1]$ $s = \langle (3, a), (1, b), (2, c), (3, d), (0, e), (0, f), (3, g), (2, h), (1, i) \rangle$ verteilen \rightsquigarrow

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

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

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

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

$C := 1$

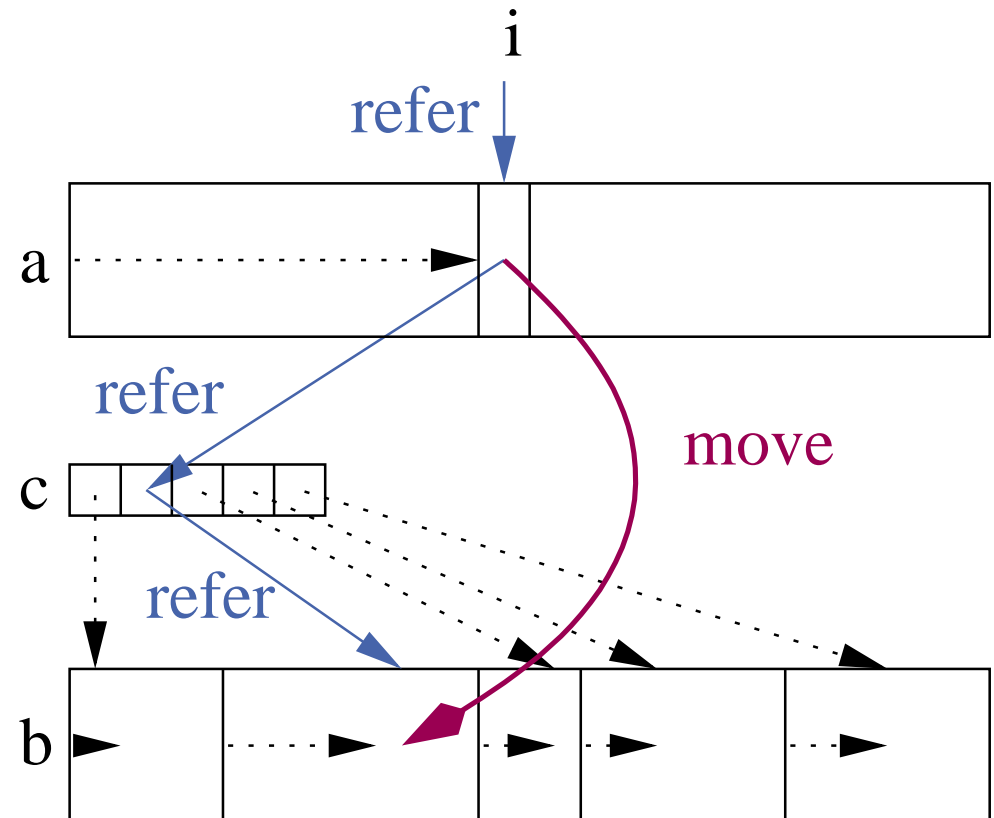
for $k := 0$ **to** $K-1$ **do**

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

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

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

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



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

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

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

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

$C := 1$

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

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

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

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

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

K^d Schlüssel –

Beobachtung: KSort ist **stabil**, d. h.,

Procedure LSDRadixSort(s : Sequence of Element)

for $i := 0$ **to** $d - 1$ **do**
$$\text{redefine KEY}(x) \text{ as } (x \mathbf{div} K^i) \mathbf{mod} K // x \begin{array}{|c|c|c|c|c|} \hline d-1 & \cdots & i & \cdots & 1 & 0 \\ \hline \end{array}$$
$$\text{KSort}(s)$$

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

Sortieren: vergleichsbasiert \leftrightarrow ganzzahlig

pro ganzzahlig:

- ☐ asymptotisch schneller

pro vergleichsbasiert

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

Mehr zum Sortieren

☐ Verfügbar in Algorithmenbibliotheken

☐ (binary) mergesort \rightsquigarrow

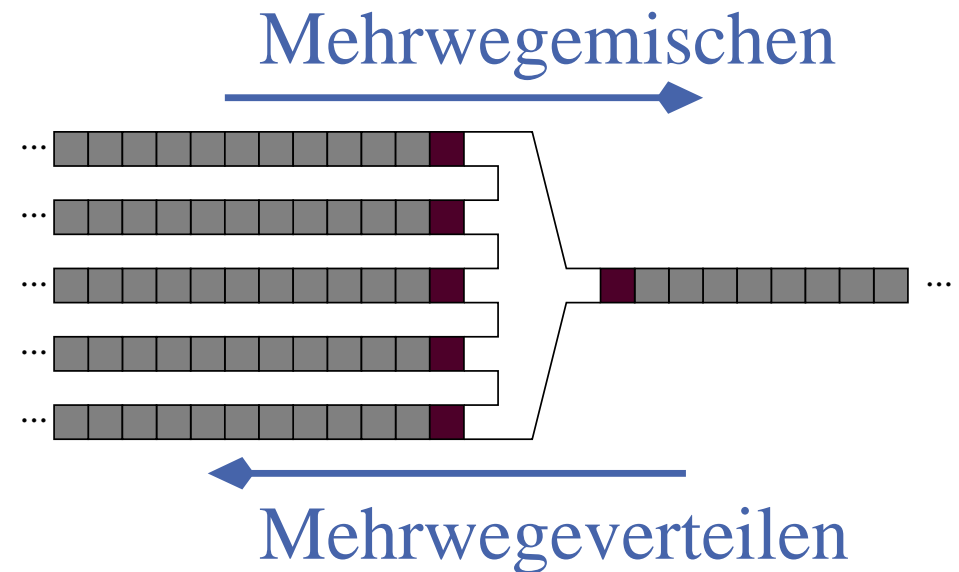
Mehrwegemischen

☐ quicksort \rightsquigarrow

Sortieren durch Mehrwegeverteilen

\rightsquigarrow Parallel

\rightsquigarrow 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

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

DeleteMin : return and remove $\min M$

Prioritätslisten – Anwendungen

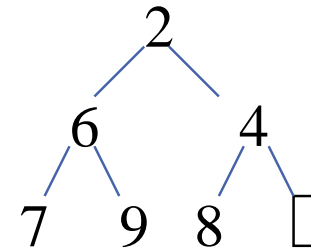
(ohne zusätzliche Operationen)

- | | |
|---|----------------|
| <input type="checkbox"/> Mehrwegemischen | (klein) |
| <input type="checkbox"/> Greedy Algorithmen (z. B., Scheduling) | (klein–mittel) |
| <input type="checkbox"/> Simulation diskreter Ereignisse | (mittel–groß) |
| <input type="checkbox"/> Branch-and-Bound Suche | (groß) |
| <input type="checkbox"/> run formation für externes Sortieren | (groß) |
| <input type="checkbox"/> Time forward processing | (riesig) |

6.1 Binäre Heaps

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

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



Beobachtung: **Minimum = Wurzel**

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

~>

INSERT, DELETEMIN brauchen **Zeit** $O(\log n)$

Implizite Baum-Repräsentation

□ Array $h[1..n]$

H:

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

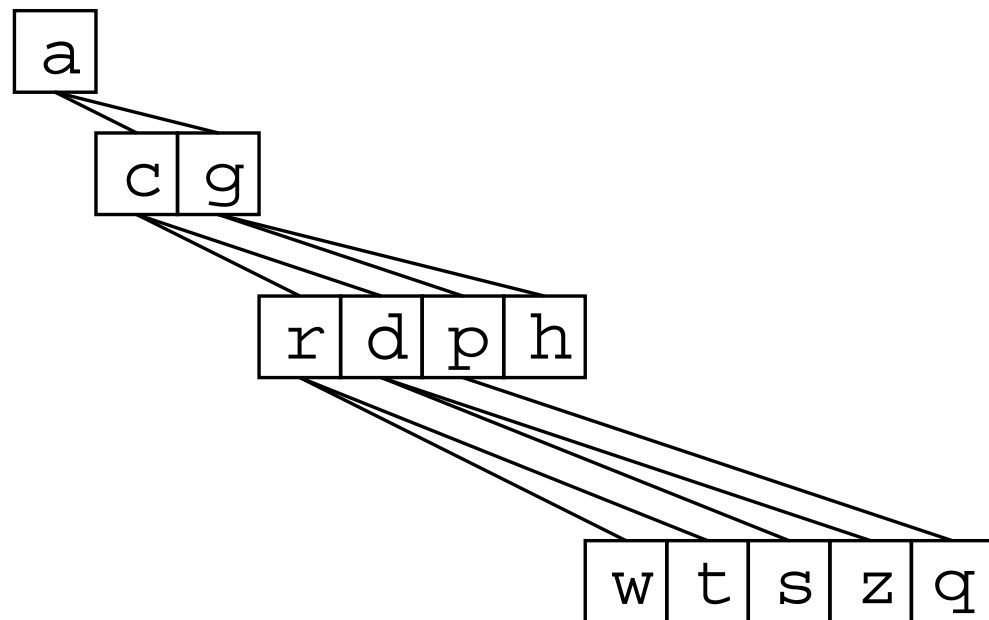
□ Schicht für Schicht

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

□ $\text{PARENT}(j) = \lfloor j/2 \rfloor$

□ linkes Kind(j): $2j$

□ rechtes Kind(j): $2j + 1$



Nicht nur nützlich für heaps:

z. B. Turnierbäume, statische Suchbäume

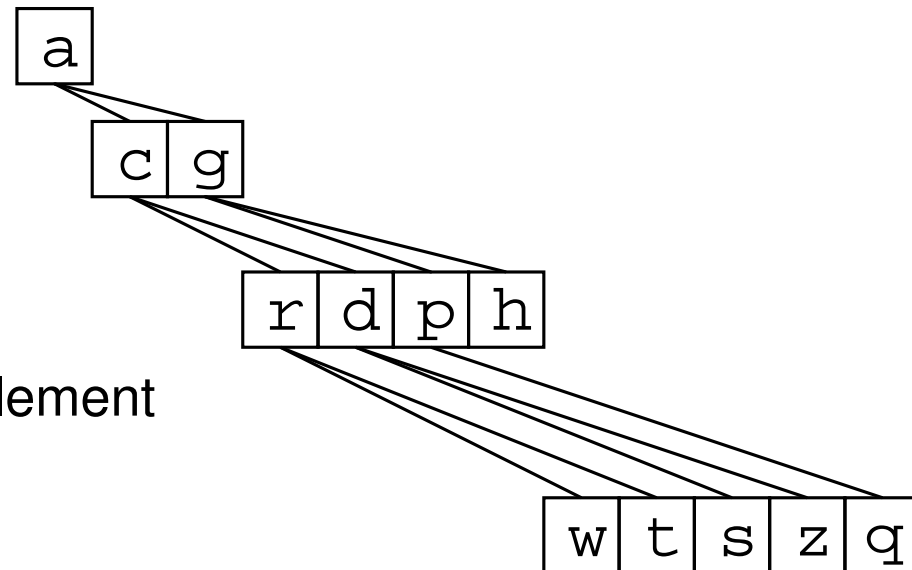
Pseudocode

(beschränkte PQ)

H:

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

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



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

$h : \mathbf{Array} [1..w] \mathbf{of}$ 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($e : \text{Element}$)

assert $n < w$

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

SIFTUP(n)

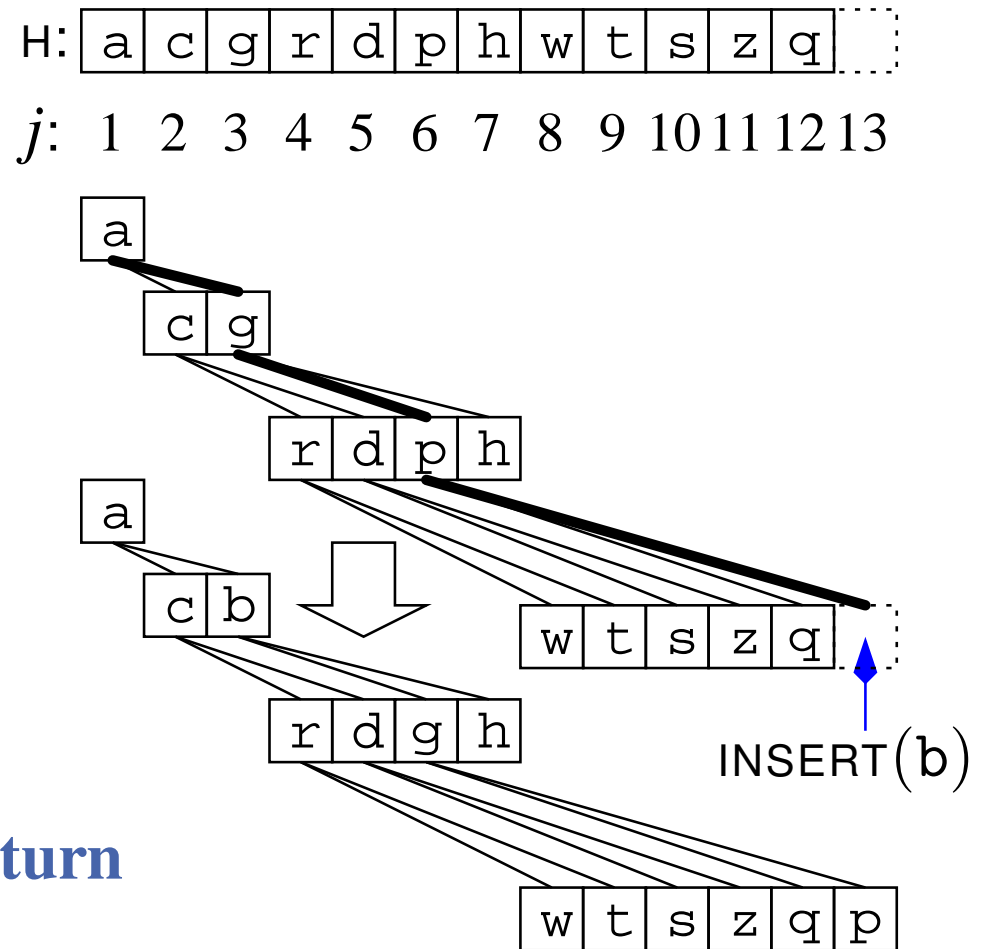
Procedure SIFTUP($i : \mathbb{N}$)

assert the heap property holds
except maybe at position i

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

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

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



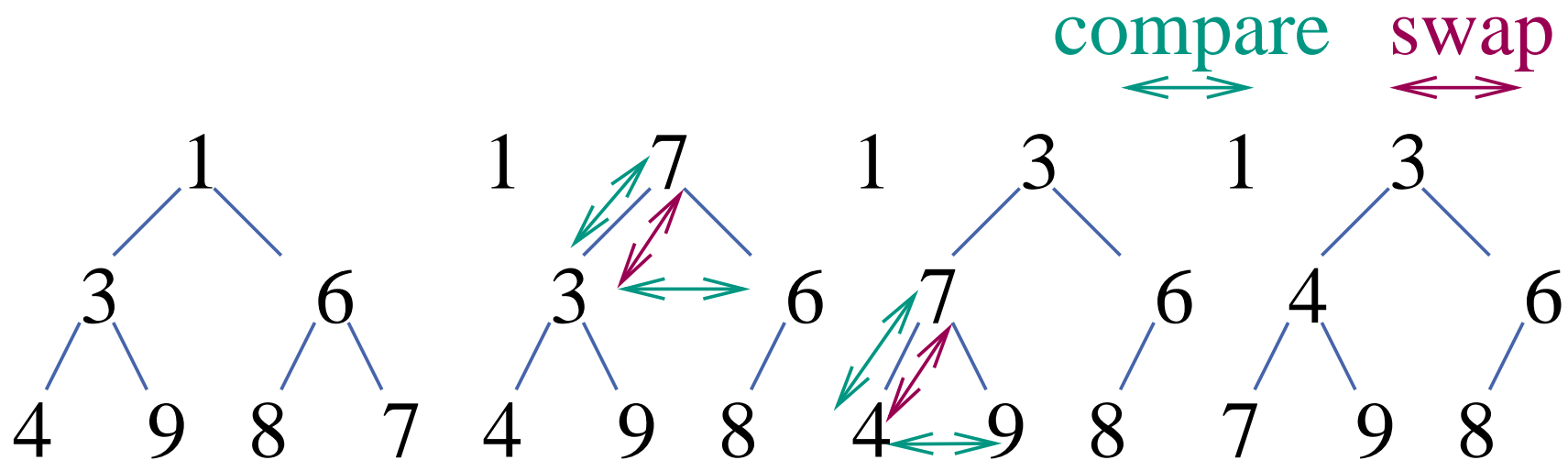
Function DELETEMIN : ELEMENT

RESULT = $h[1]$: ELEMENT

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

SIFTDOWN(1)

return RESULT



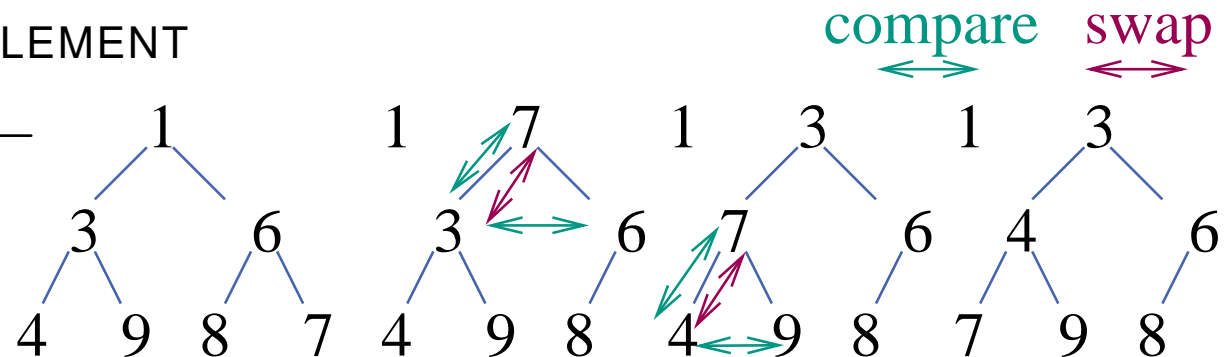
Function DELETEMIN : ELEMENT

RESULT = $h[1]$: ELEMENT

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

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 $\exists \text{SIBLING}(m) \vee h[\text{SIBLING}(m)] \geq h[m]$

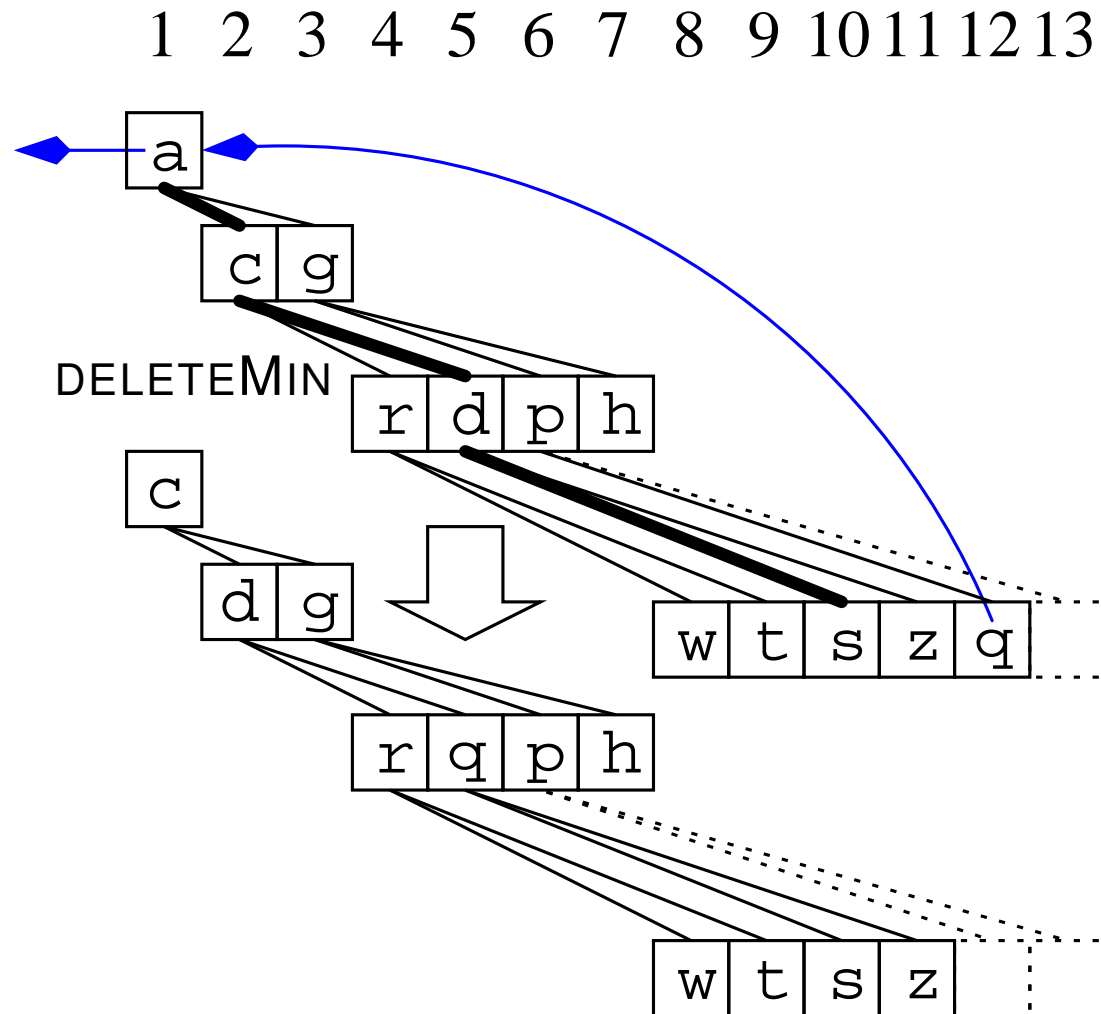
if $h[i] > h[m]$ **then** // heap property violated

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

SIFTDOWN(m)

assert the heap property holds for the subtree rooted at i

deleteMin: Beispiel



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($a[1..n]$) $h := a$; 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)

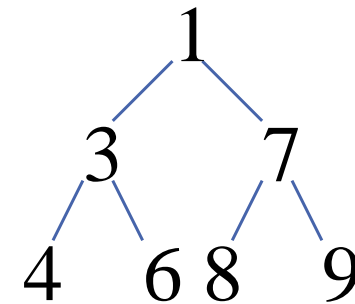
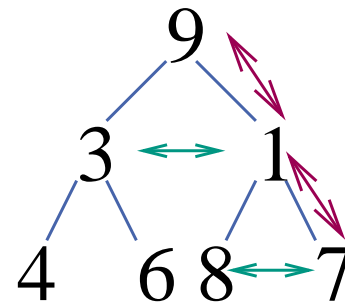
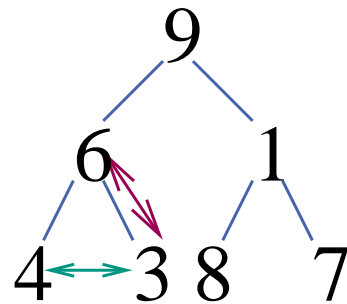
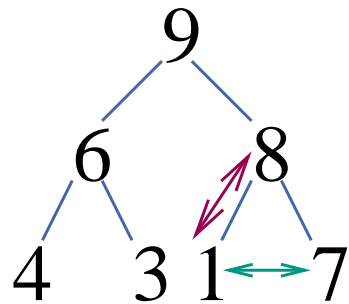
assert the heap property holds for the tree rooted at i

Procedure buildHeapBackwards

for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** SIFTDOWN(i)

Beispiel: Binärer Heap – Konstruktion

compare swap



Binärer Heap – Konstruktion

Satz: BUILDHEAP läuft in Zeit $O(n)$

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

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

□ 2^ℓ Aufrufe von SIFTDOWN

□ Kosten je $O(k - \ell)$. Insgesamt:

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

Ein nützlicher Rechentrick

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

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

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

Heapsort

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

 buildHeap(a)

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

$h[i] := \text{DELETMIN}$

Laufzeit: $O(n \log n)$

Andere Sichtweise: effiziente Implementierung von
Sortieren durch Auswahl

Frage: Wie sortiert man aufsteigend?

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

Kompromiss: z. B.

introspektives Quicksort der C++ Standardbibliothek:

Quicksort starten. Zu wenig Fortschritt? Umschalten auf Heapsort.

6.2 Adressierbare Prioritätslisten

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

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

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

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

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

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

Procedure decreaseKey($h : \text{HANDLE}, k : \text{KEY}$) **assert** $\text{KEY}(h) \geq k$; $\text{KEY}(h) := k$

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

Adressierbare Prioritätslisten: Anwendungen

Greedy-Algorithmus:

while solution not complete **do**

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

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

Beispiele:

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

Adressierbare Binäre Heaps

Problem: Elemente bewegen sich.

Dadurch werden Elementverweise ungültig.

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

Invariante: $\text{PROXY}(e)$ verweist auf Position von e .

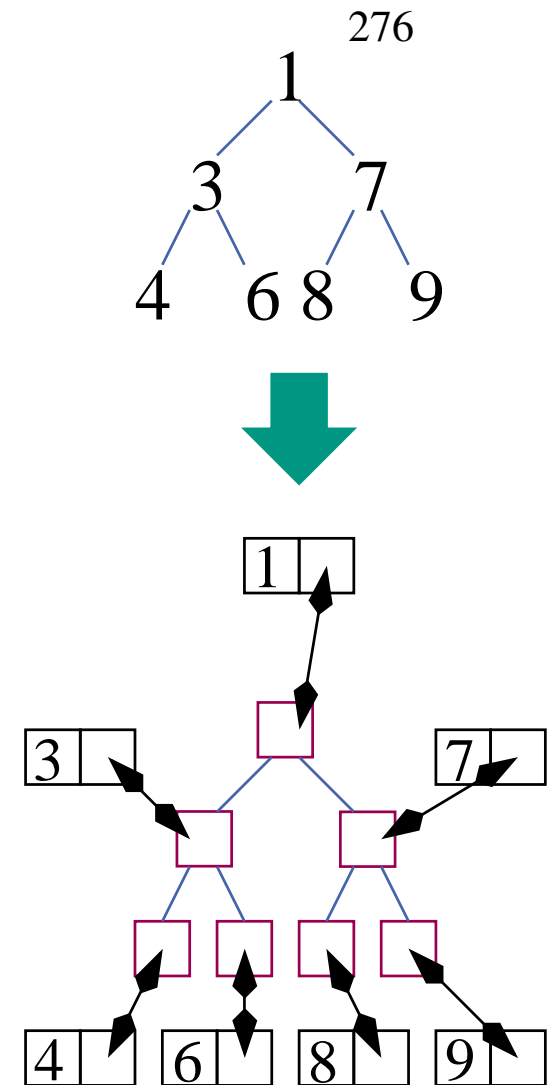
≈ Vermittler bei jeder Vertauschung aktualisieren.

≈ Rückverweis Element \rightarrow Vermittler

Laufzeit:

$O(\log n)$ für alle Operationen ausser MERGE und BUILDHEAP, die

$O(n)$ brauchen.



Adressierbare Prioritätslisten – Laufzeiten

Operation	Binary Heap	Fibonacci Heap (Buch)
build	$O(n)$	$O(n)$
size	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(\log n)$
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 (nichtadressierbar)
- ☐ parallel: Semantik?

Prioritätslisten: Zusammenfassung

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

7 Sortierte Folgen

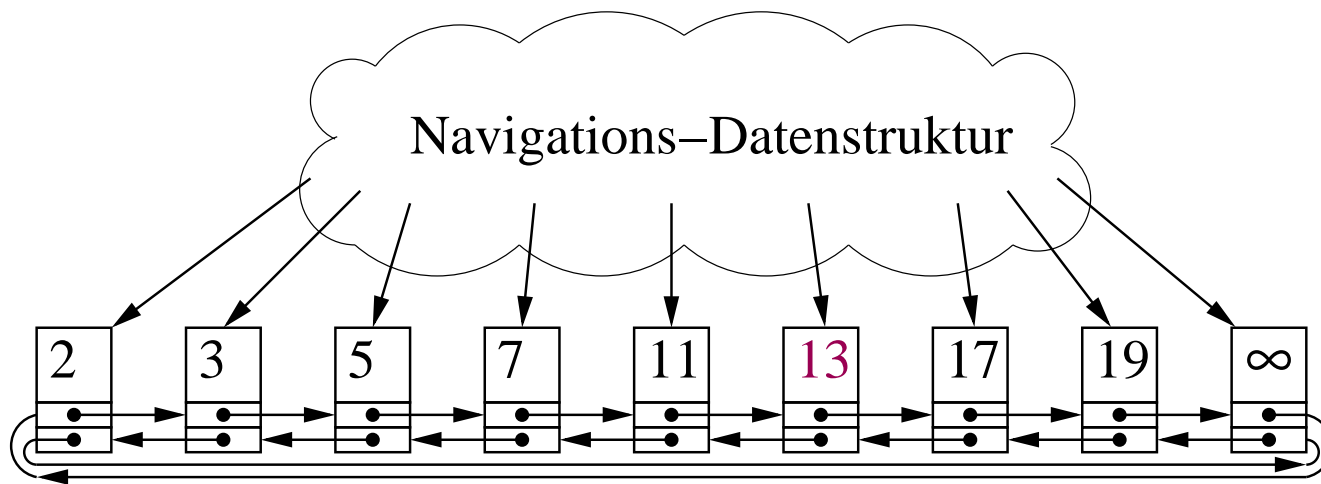


Sortierte Folgen:

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

„kennzeichnende“ Funktion:

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



Annahme: Dummy-Element mit Schlüssel ∞

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 $a[\ell..r-1] = [k, \dots, k]$, $a[\ell-1] < k$ und $a[r] > k$

Statisch: Sortiertes Feld mit binärer Suche

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

Function locate($a[1..n], k : \text{ELEMENT}$)

$(\ell, r) := (0, n+1)$ // 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($a[1..n], k : \text{ELEMENT}$)

$(\ell, r) := (0, n+1)$ // 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

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

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

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

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

Dynamische Sortierte Folgen – Grundoperationen

INSERT, REMOVE, UPDATE, LOCATE

$O(\log n)$

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

Mehr Operationen

 $\langle \text{min}, \dots, a, \dots, b, \dots, \text{max} \rangle$

min: Erstes Listenelement

Zeit $O(1)$

max: Letztes Listenelement

Zeit $O(1)$

RANGESearch(a, b)

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

RESULT := $\langle \rangle$

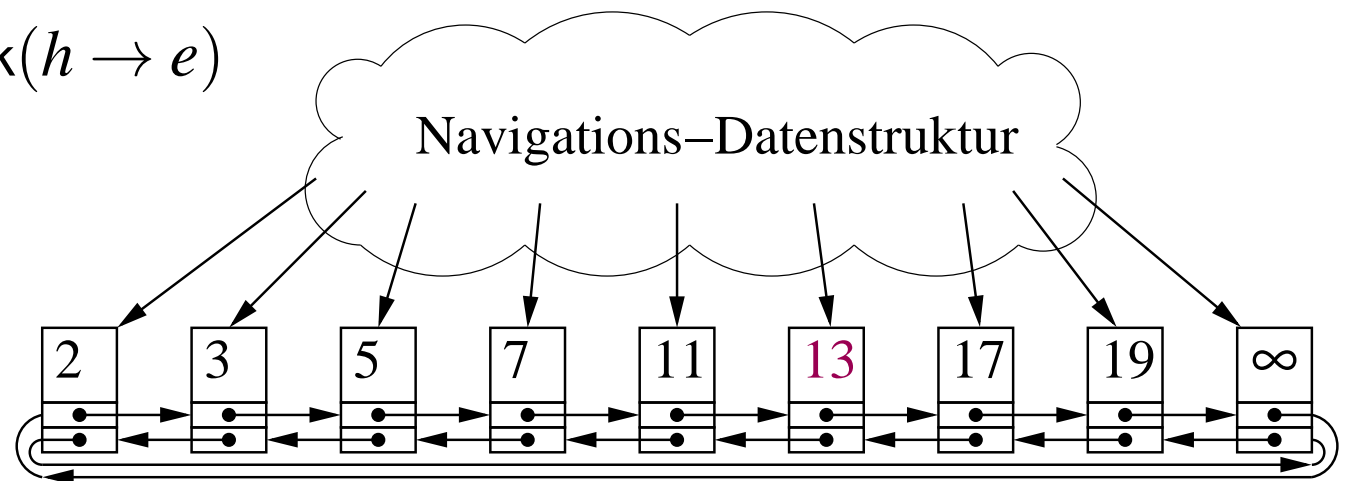
$h := \text{LOCATE}(a)$

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

 result.pushBack($h \rightarrow e$)

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

return result



Noch mehr Operationen

□ (re)**build**: Navigationstruktur für **sortierte** Liste aufbauen $O(n)$

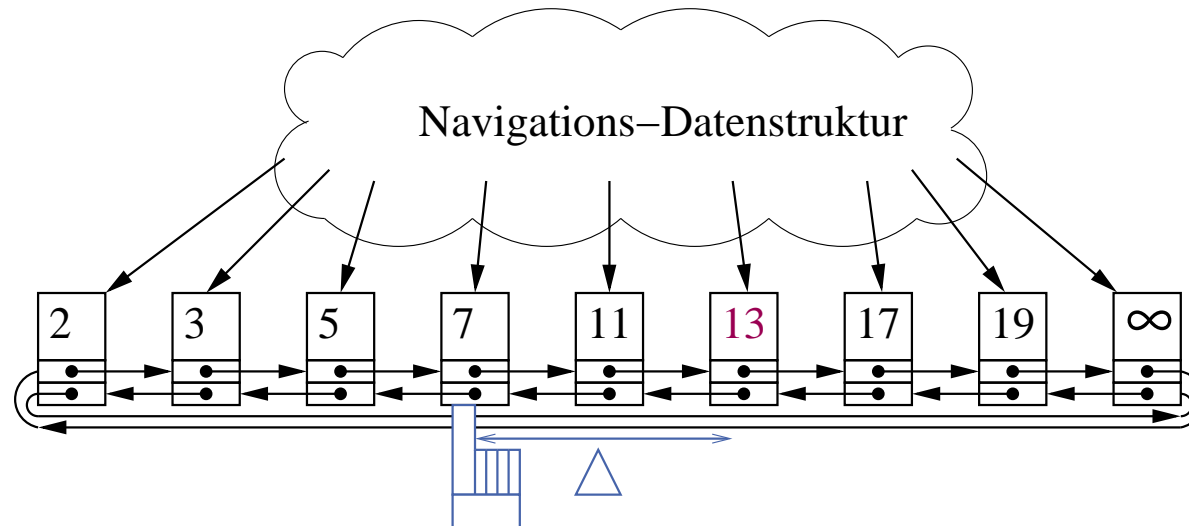
□ $\langle w, \dots, x \rangle.$ **concat** $(\langle y, \dots, z \rangle) = \langle w, \dots, x, y, \dots, z \rangle$ $O(\log n)$

□ $\langle w, \dots, x, y, \dots, z \rangle.$ **split** $(y) = (\langle w, \dots, x \rangle, \langle y, \dots, z \rangle)$ $O(\log n)$

Zählen: RANK, SELECT, RANGE SIZE $O(\log n)$

Fingersuche: Δ = Abstand zu **Finger**info

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



Abgrenzung

Hash-Tabelle: nur INSERT, REMOVE, FIND. Kein LOCATE, RANGEQUERY

Sortiertes Feld: nur bulk-Updates. Aber:

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

Prioritätsliste: nur INSERT, DELETEMIN, (DECREASEKEY, REMOVE).

Dafür: schnelles MERGE

Insgesamt: die eierlegende Wollmilchdatenstruktur.

„Etwas“ langsamer als speziellere Datenstrukturen

Sortierte Folgen – Anwendungen

- ☐ Best-First Heuristiken
- ☐ Alg. Geometrie: Sweep-line-Datenstrukturen
- ☐ Datenbankindex
- ☐ ...

Anwendungsbeispiel: Best Fit Bin Packing

Procedure binPacking(s)

B : SORTEDSEQUENCE // used bins sorted by free capacity

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

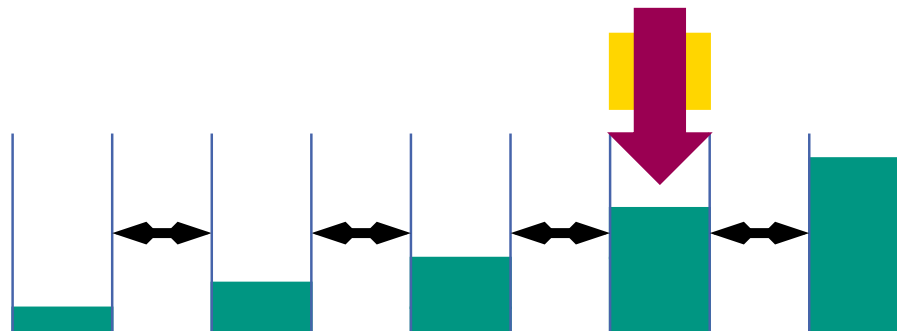
if $\neg \exists b \in B : \text{FREE}(b) > e$ **then** $B.\text{INSERT}(\text{new bin})$

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

insert e into bin b

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

Qualität: „gut“. Details: nicht hier



7.1 Binäre Suchbäume

Blätter: Elemente einer sortierten Folge.

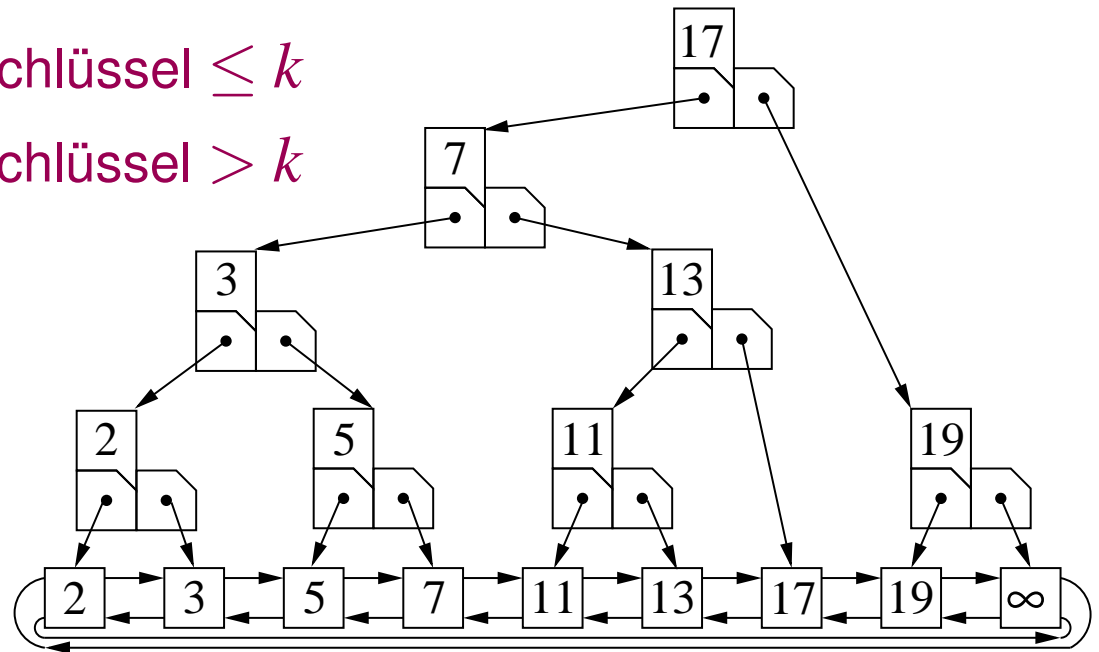
Innere Knoten $v = (k, \ell, r)$,

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

Invariante:

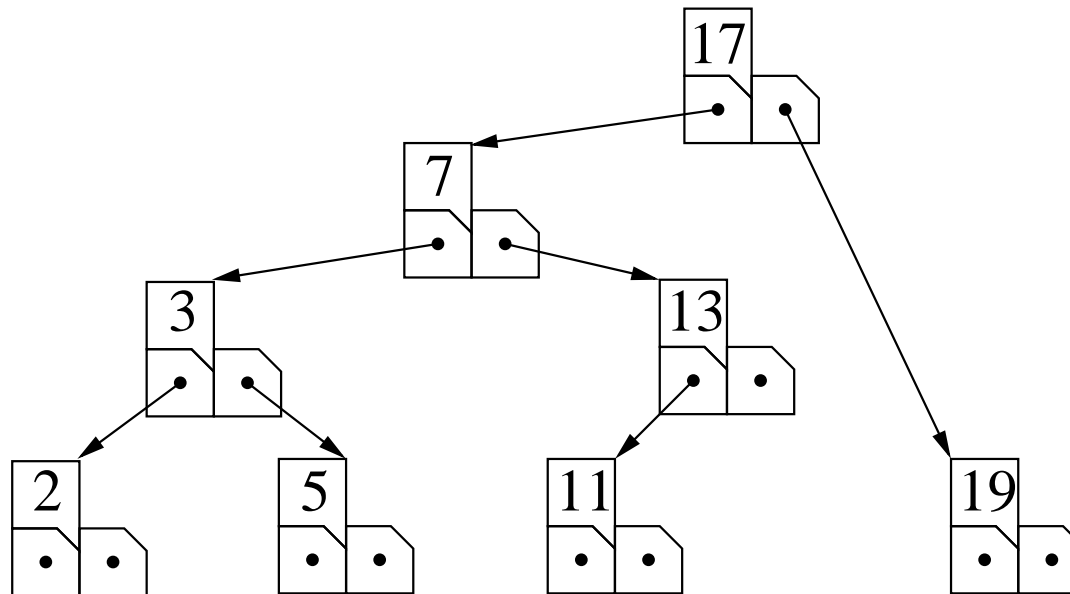
über ℓ erreichbare Blätter haben **Schlüssel** $\leq k$

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



Varianten, Bemerkungen

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



locate(k)

Idee: Benutze Spaltschlüssel x als Wegweiser.

Function locate(k, x)

if x is a leaf **then**

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

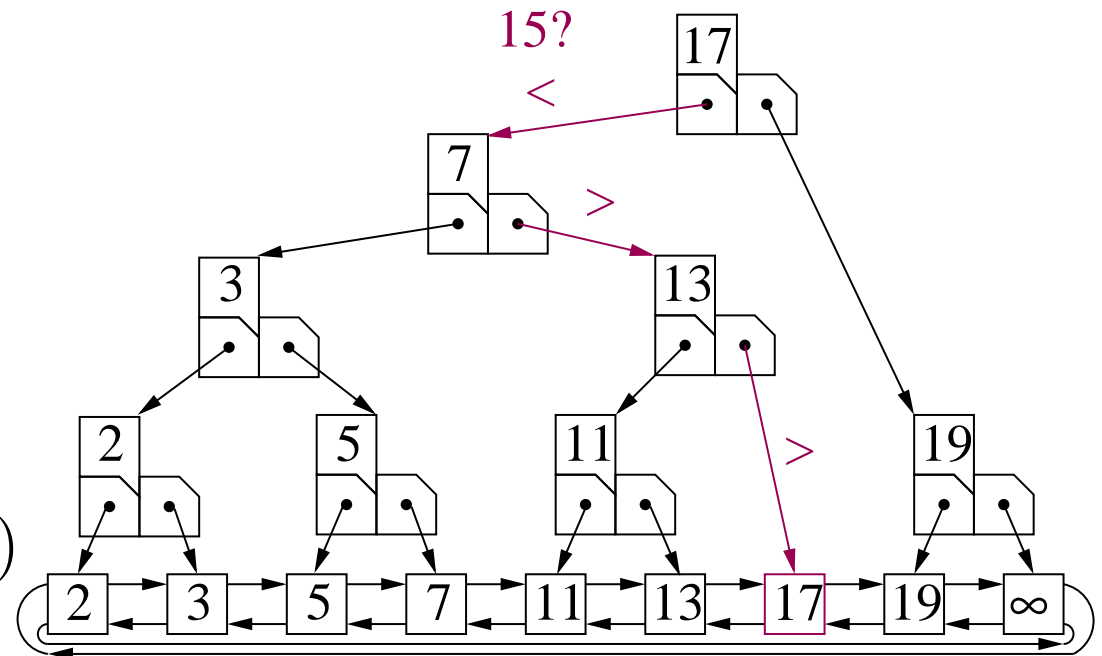
else return $x \rightarrow \text{next}$

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

return locate($k, x \rightarrow \text{left}$)

else

return locate($k, x \rightarrow \text{right}$)



Invariante von $\text{locate}(k)$

Function $\text{locate}(k, x)$

if x is a leaf **then**

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

else return $x \rightarrow \text{next}$

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

return $\text{locate}(k, x \rightarrow \text{left})$

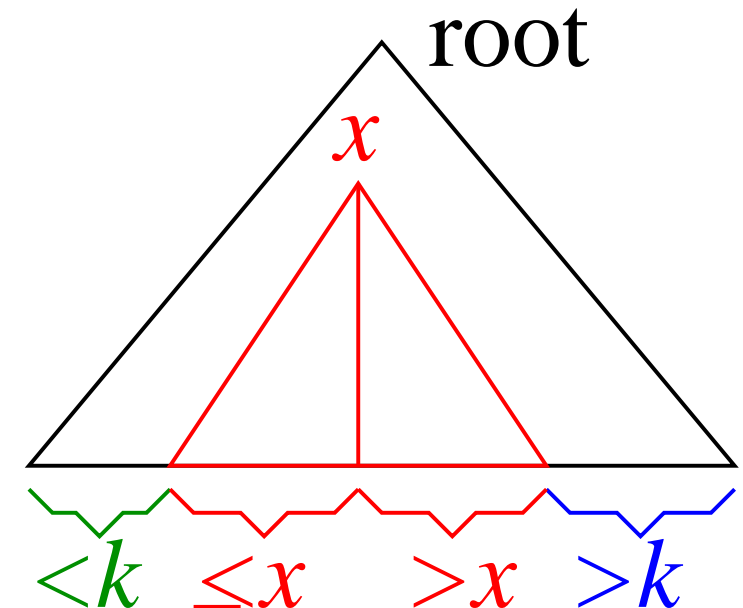
else

return $\text{locate}(k, x \rightarrow \text{right})$

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

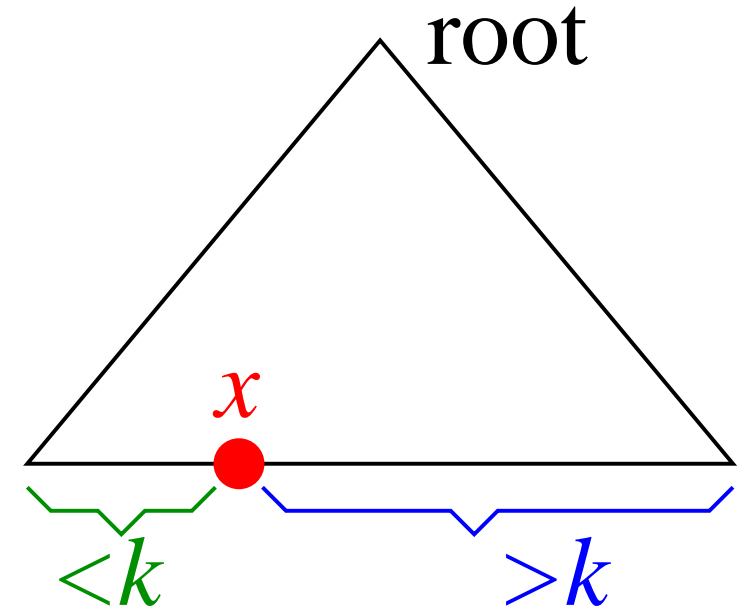
Listenelemente **links** von X sind $< k$

Listenelemente **rechts** von X sind $> k$



Ergebnisberechnung von $\text{locate}(k)$

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



Bingo!

links isses auch net.

das ist $> k$ und k gibts nich

Function $\text{locate}(k, x)$

if $k \leq x$ then return x

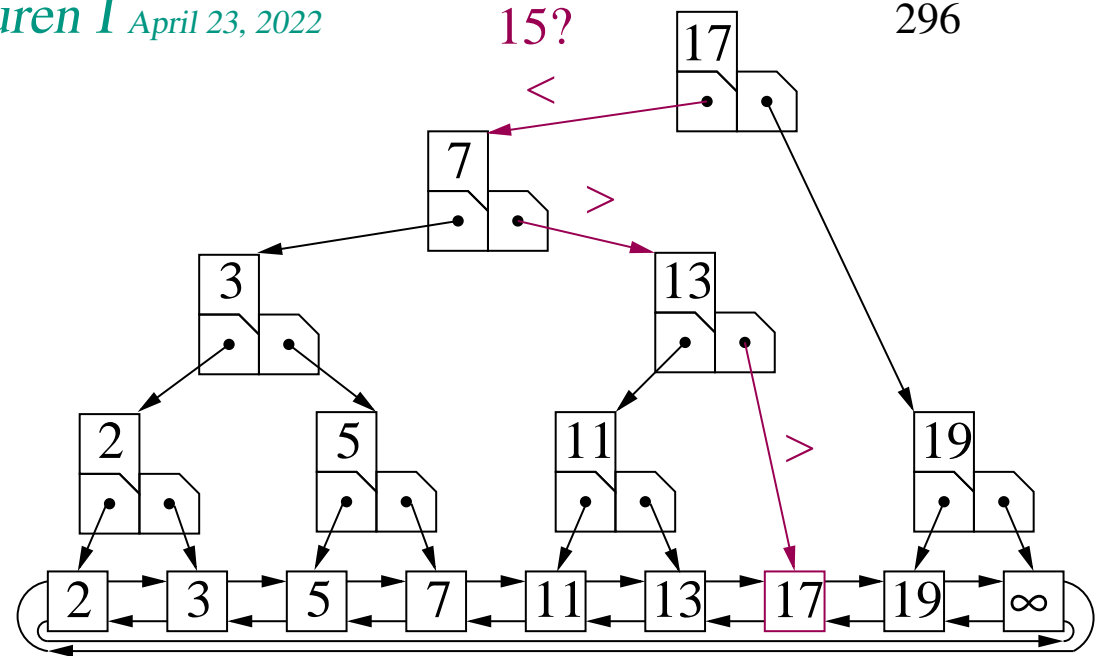
if $k \leq x$ then

```
return locate( $k, x \rightarrow \text{left}$ )
```

```
return locate( $k, x \rightarrow \text{right}$ )
```

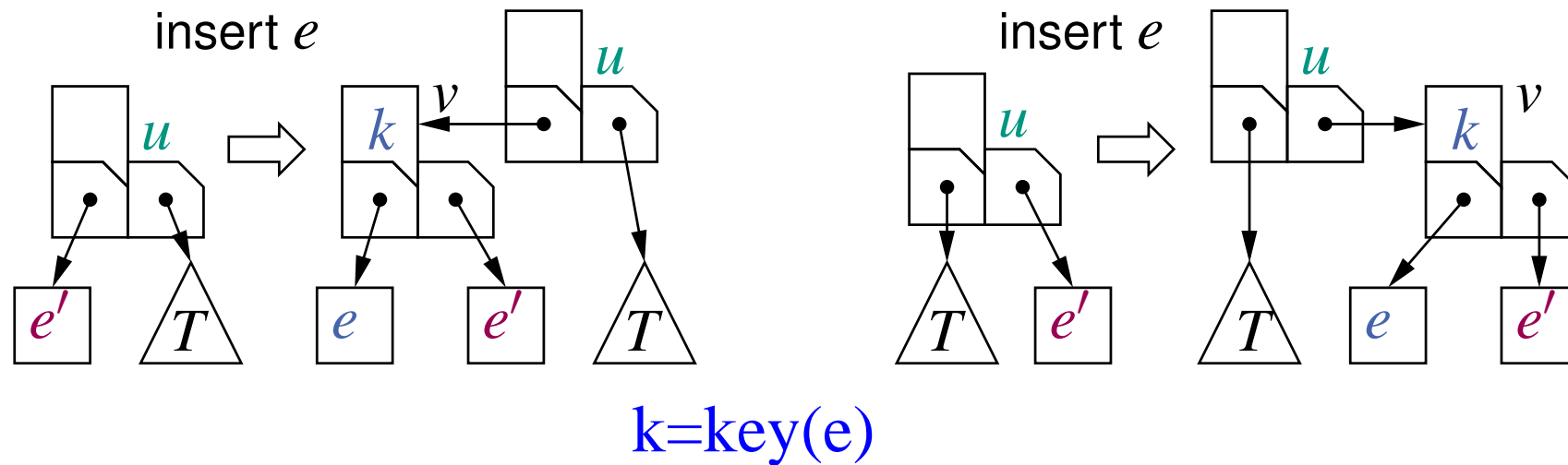
Bester Fall: perfekt balanciert, d. h. Tiefe = $\lceil \log n \rceil$

Schlechtester Fall: Höhe n



Naives Einfügen

Zunächst wie $\text{locate}(e)$. Sei e' gefundenes Element, u der Elterknoten



Suchbäume balancieren

perfekte Balance: schwer aufrechtzuerhalten

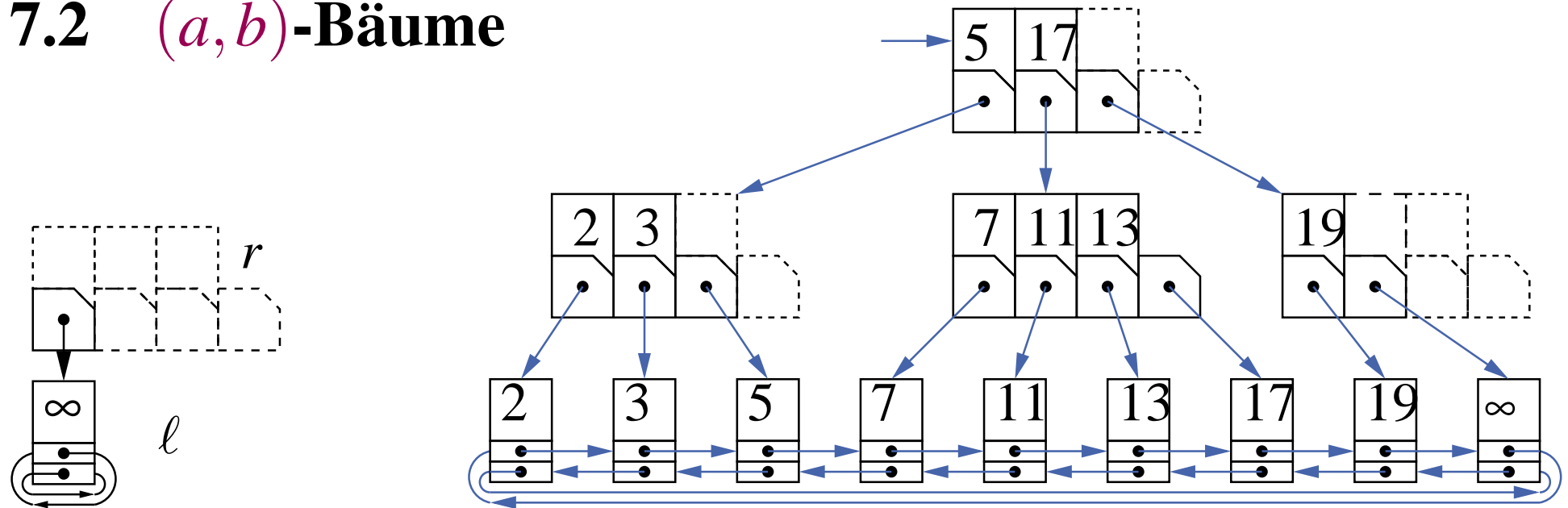
flexible Höhe $O(\log n)$: balancierte binäre Suchbäume.
nicht hier (Variantenzoo).

flexibler Knotengrad: (a, b) -Bäume.

\approx Grad zwischen a und b .

Höhe $\approx \log_a n$

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)

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

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

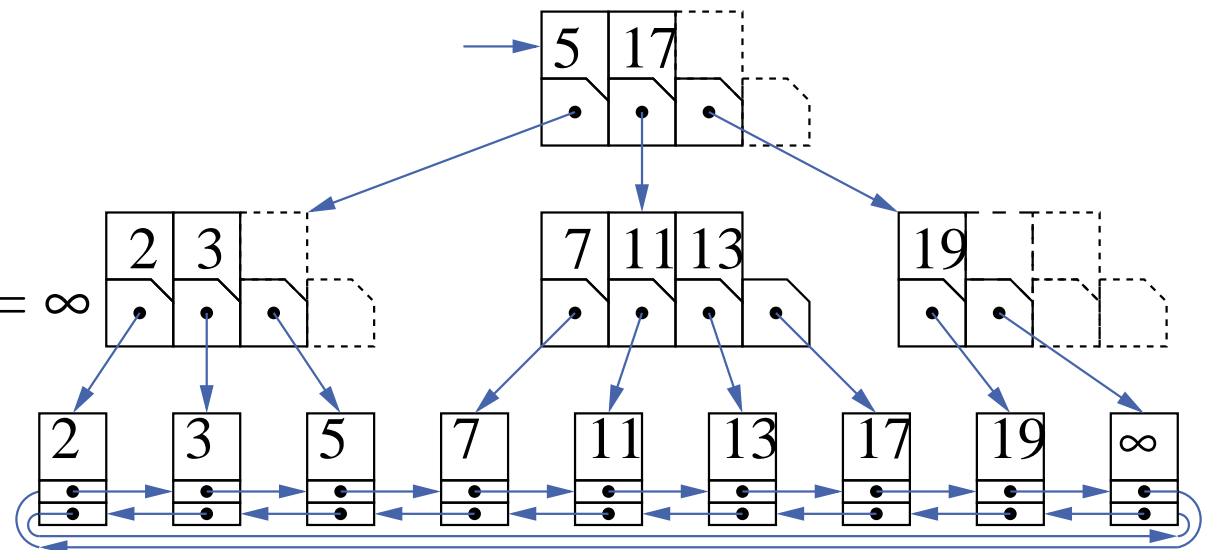
$c = \text{CHILDREN} : \text{Array } [1..b] \text{ of Handle}$

Invariante:

e über $c[i]$ erreichbar

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

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



Initialisierung

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

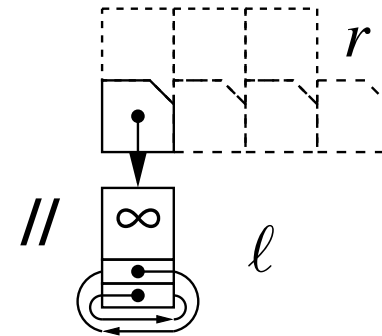
$\ell = \langle \rangle$: List **of** Element

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

height=1 : \mathbb{N}

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

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



Locate

Function ABItem::locateLocally(k : Key) : \mathbb{N}

return $\min \{i \in 1..d : k \leq s[i]\}$

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

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

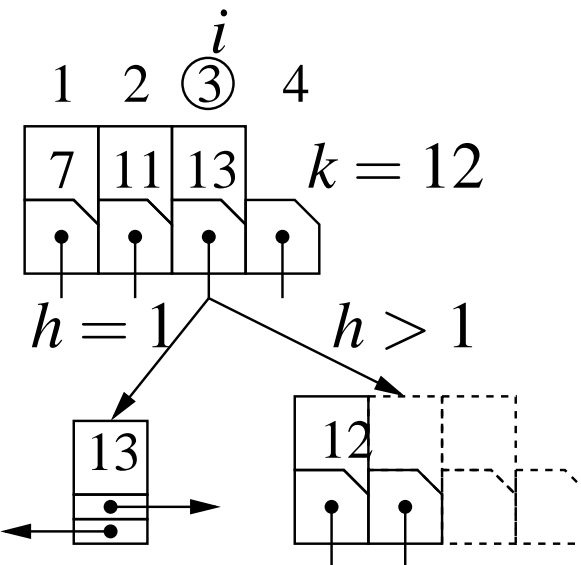
if $h = 1$ **then**

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

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

else

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



Invariante: analog binäre Suchbäume

Locate – Laufzeit

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

Übung: $b \rightarrow \log b$?

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

□

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**

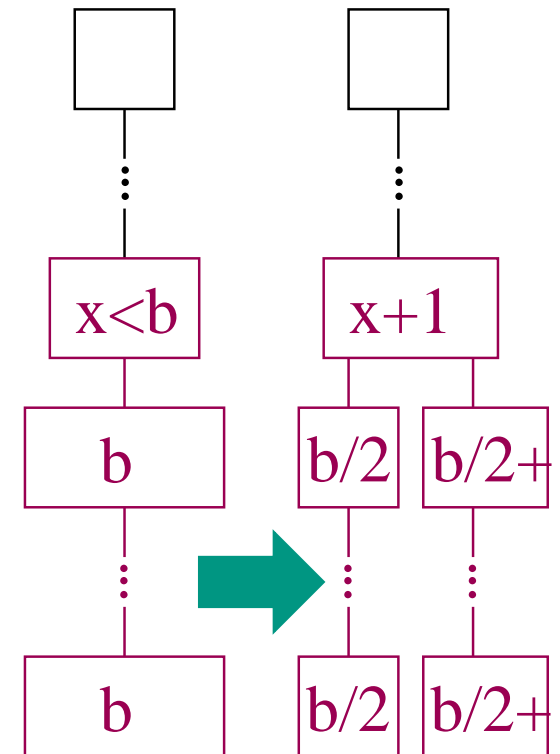
spalte u in 2 Knoten mit Graden

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

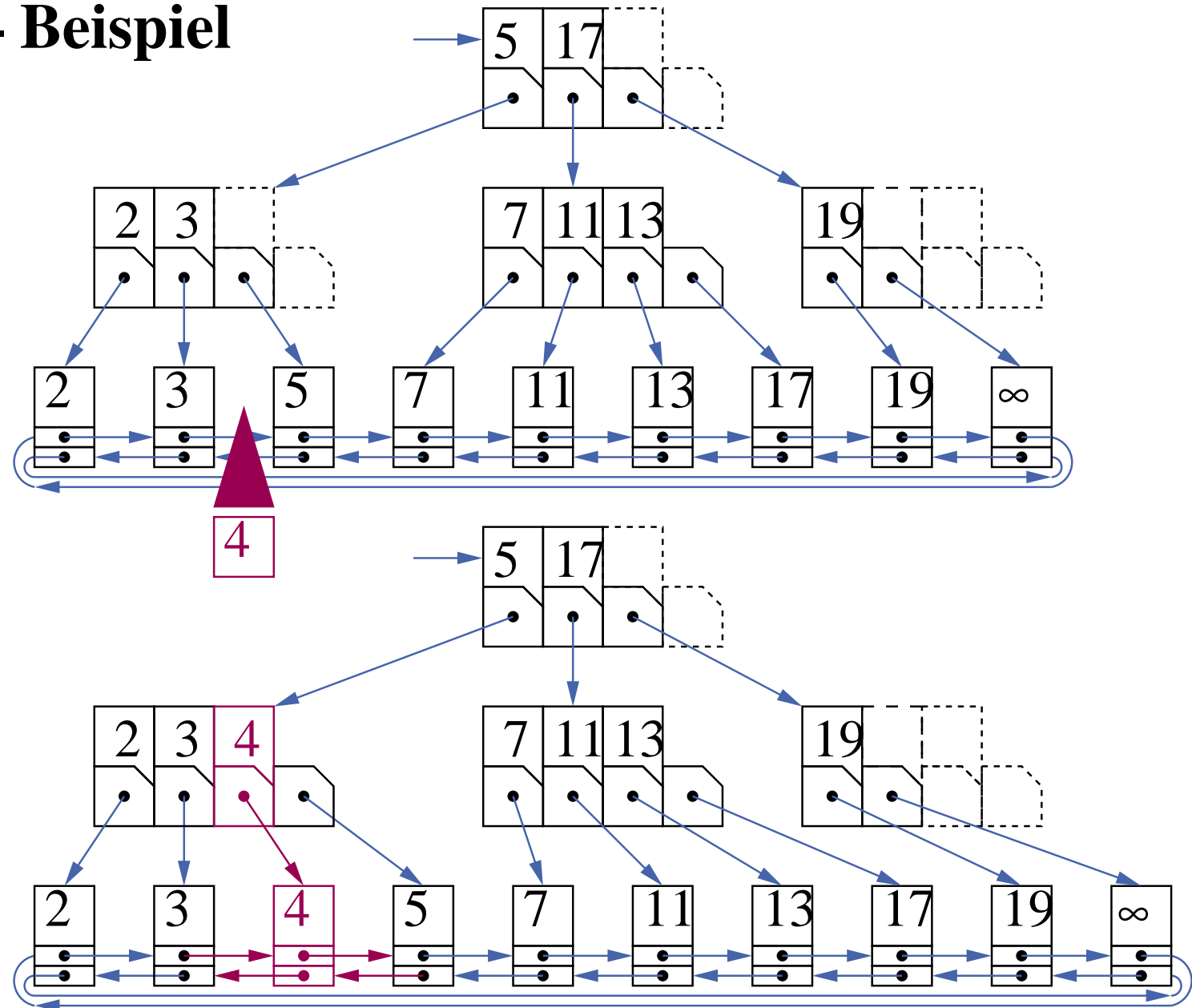
 Weiter oben einfügen, spalten

 ...

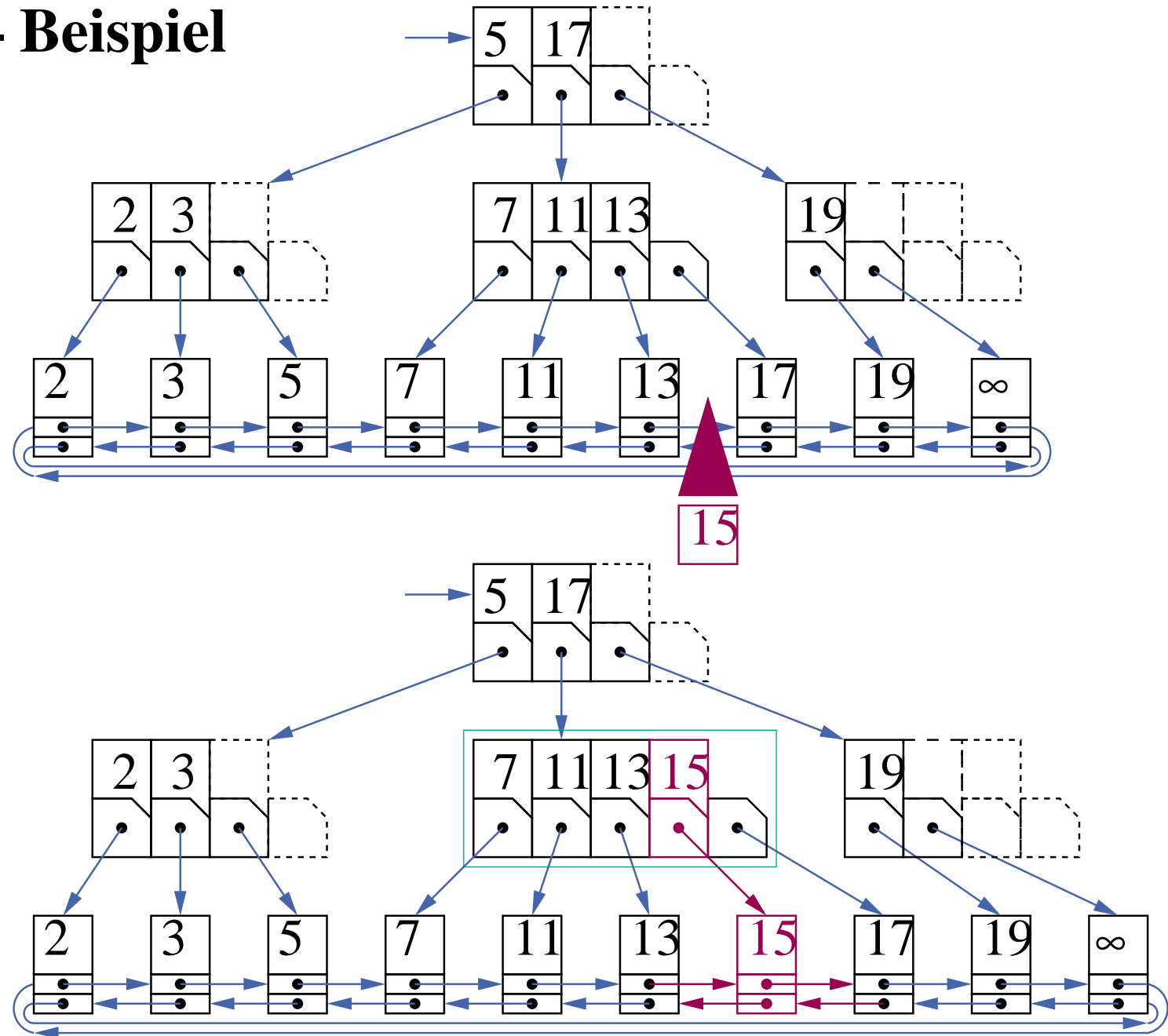
 ggf. neue Wurzel



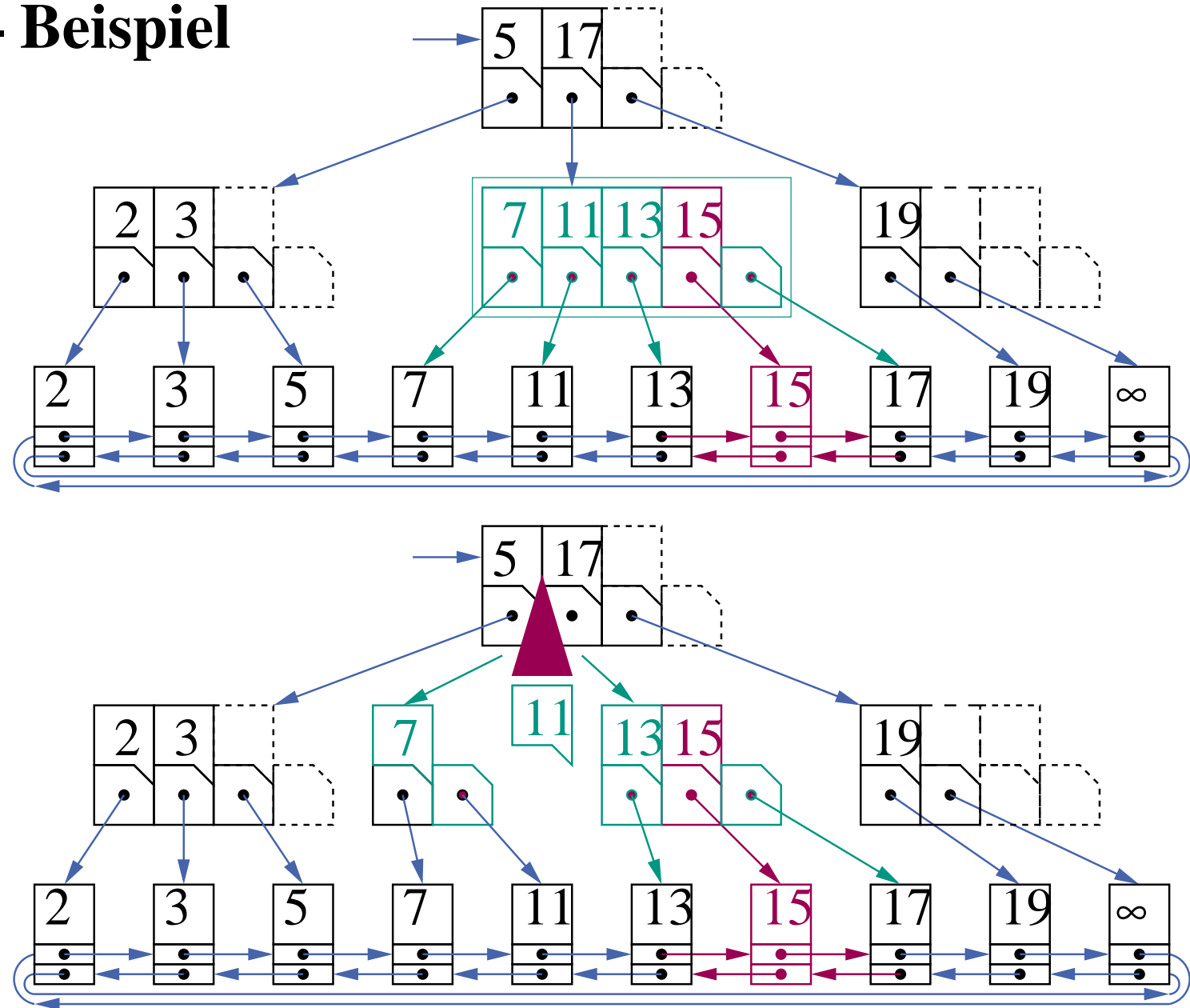
Einfügen – Beispiel



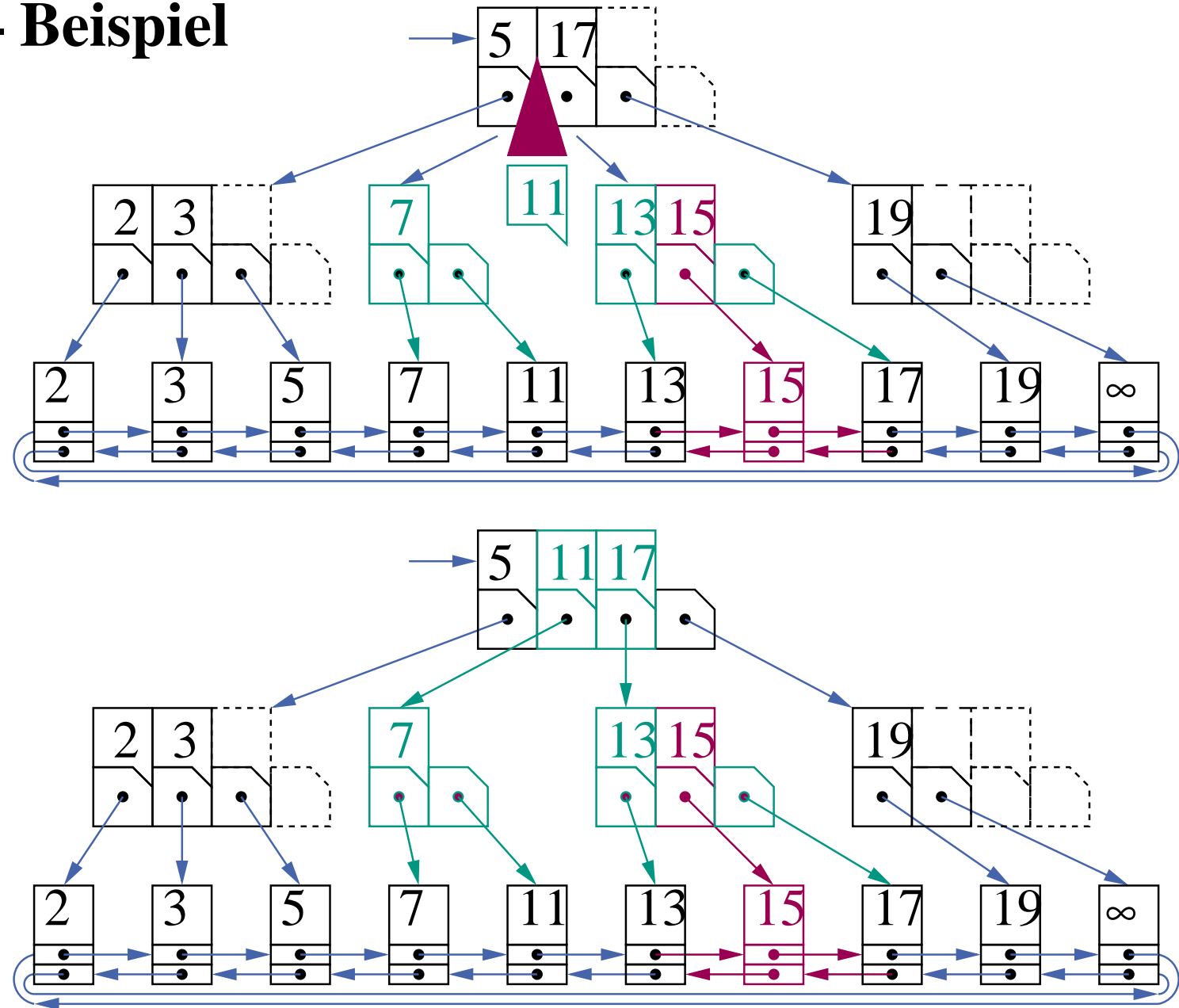
Einfügen – Beispiel



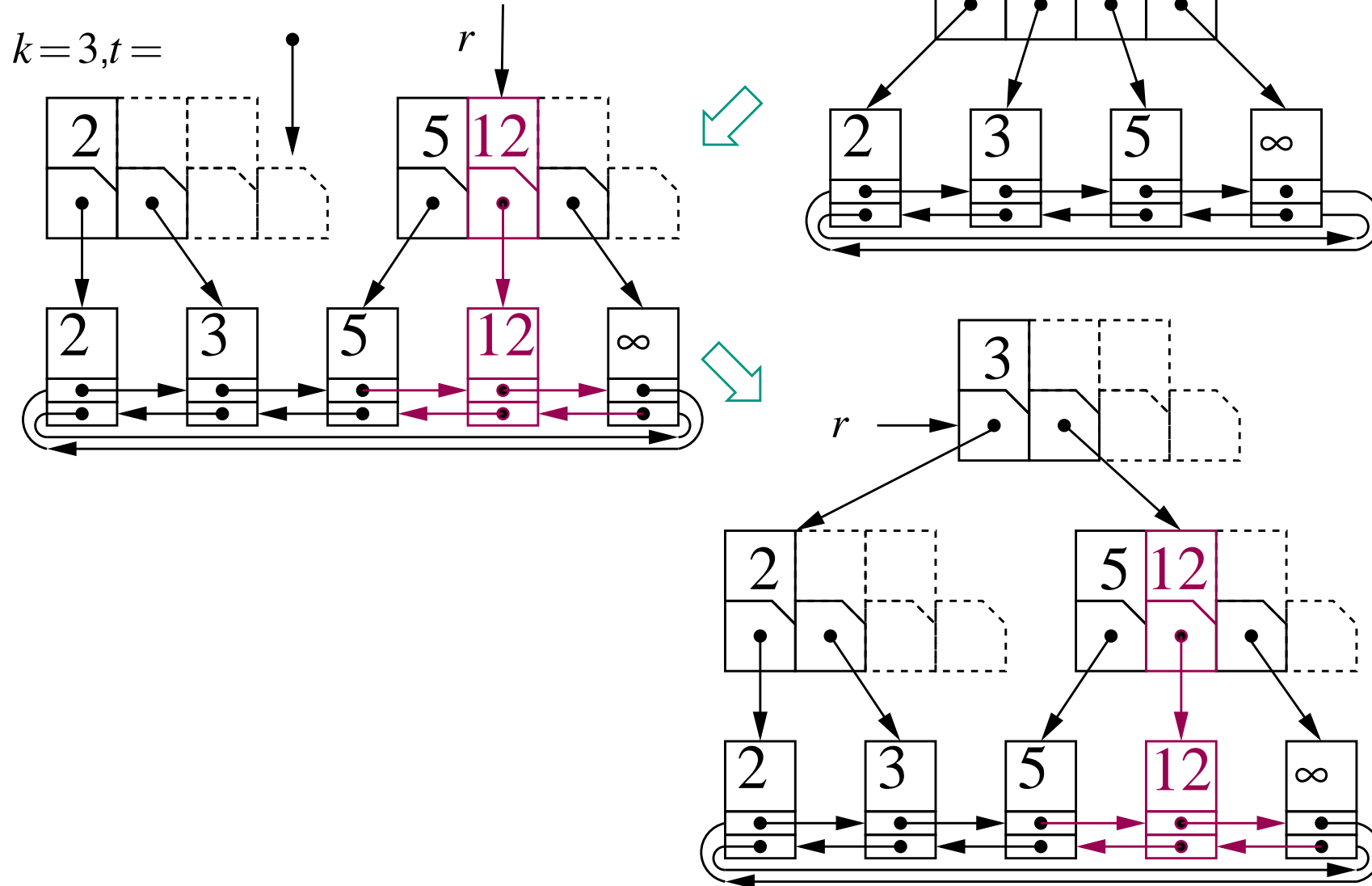
Einfügen – Beispiel



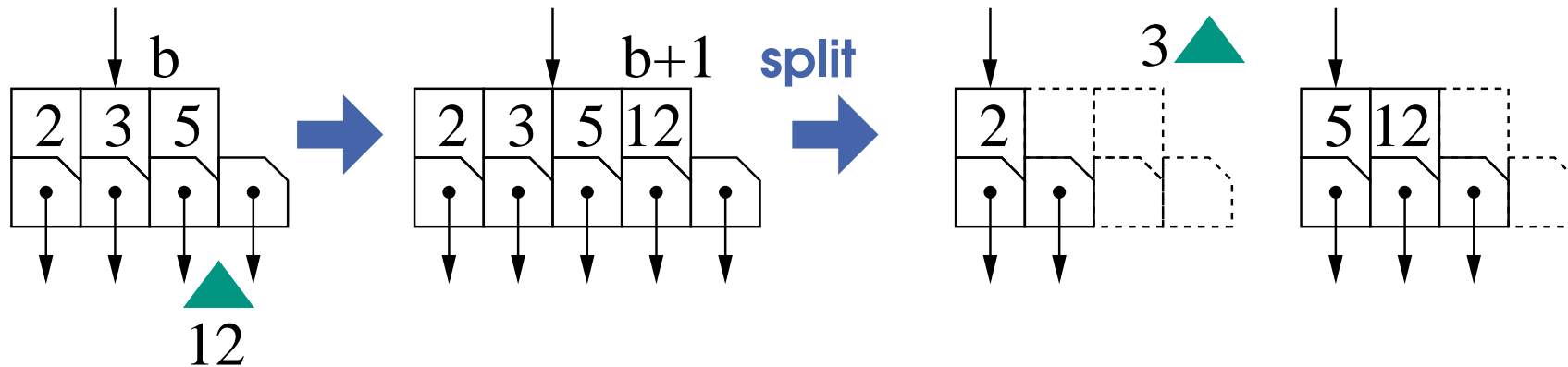
Einfügen – Beispiel



Einfügen – Beispiel



Einfügen – Korrektheit



Nach dem Split müssen zulässige Items entstehen:

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

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.

Entfernen – Algorithmenskizze

Procedure $\text{remove}(e)$

Finde Pfad Wurzel– e

$\ell.\text{REMOVE}(e)$

entferne $\text{KEY}(e)$ in Vorgänger u

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

finde Nachbarn u'

if $u'.d + a - 1 \leq b$ **then**

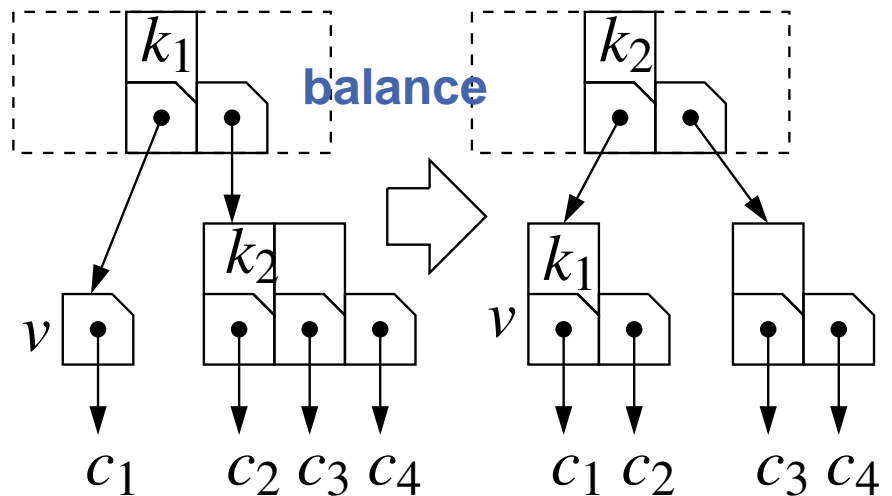
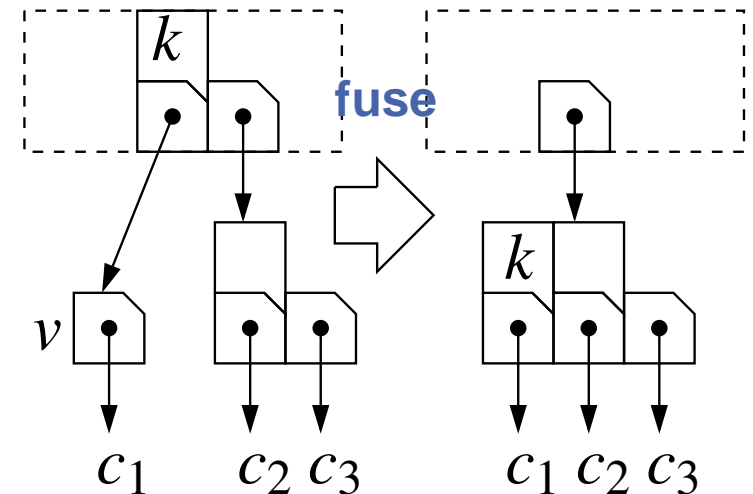
$\text{fuse}(u', u)$

Weiter oben splitter entfernen

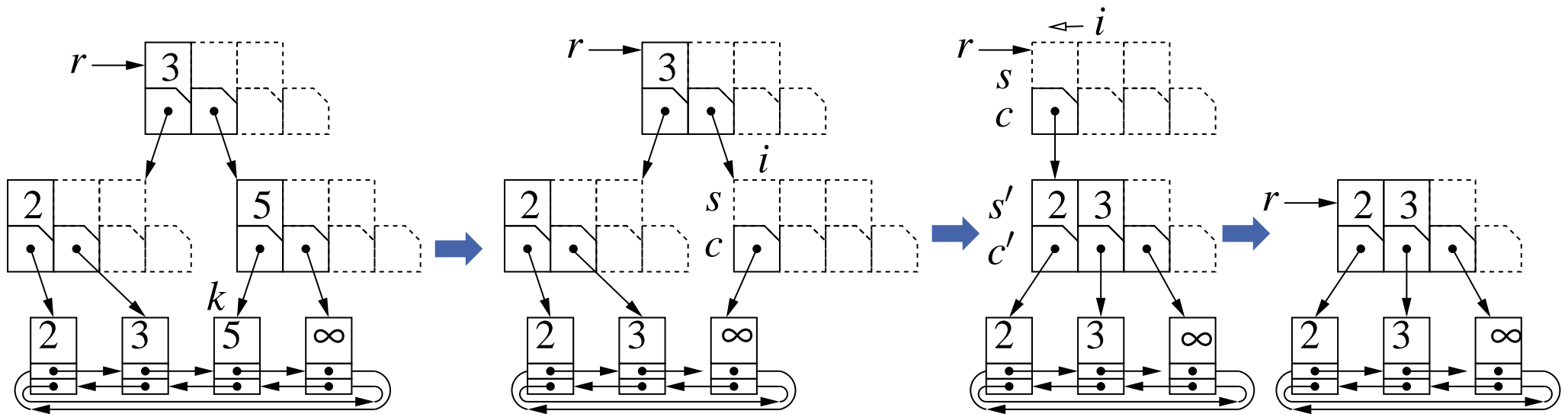
...

ggf. Wurzel entfernen

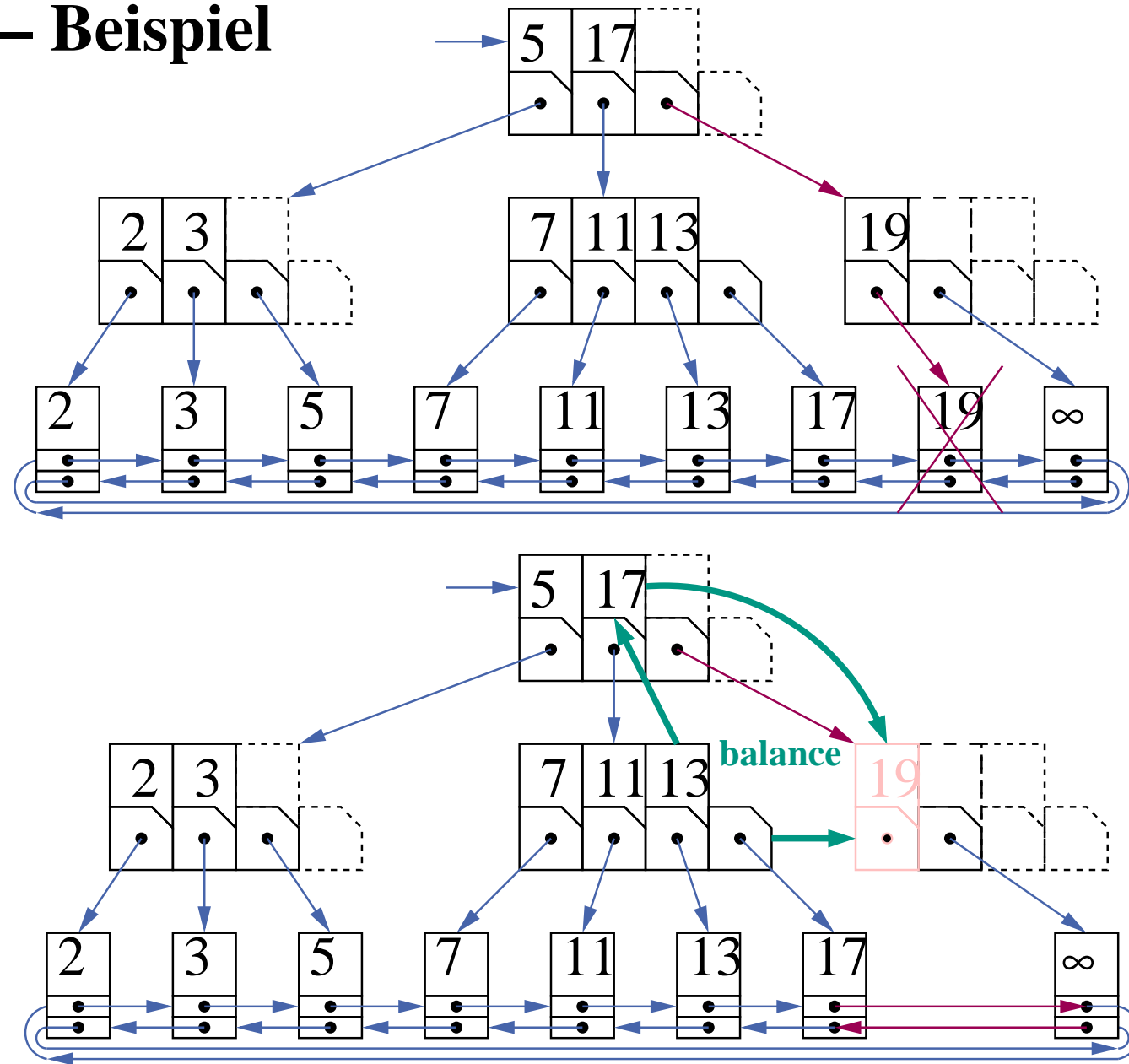
else $\text{balance}(u', u)$



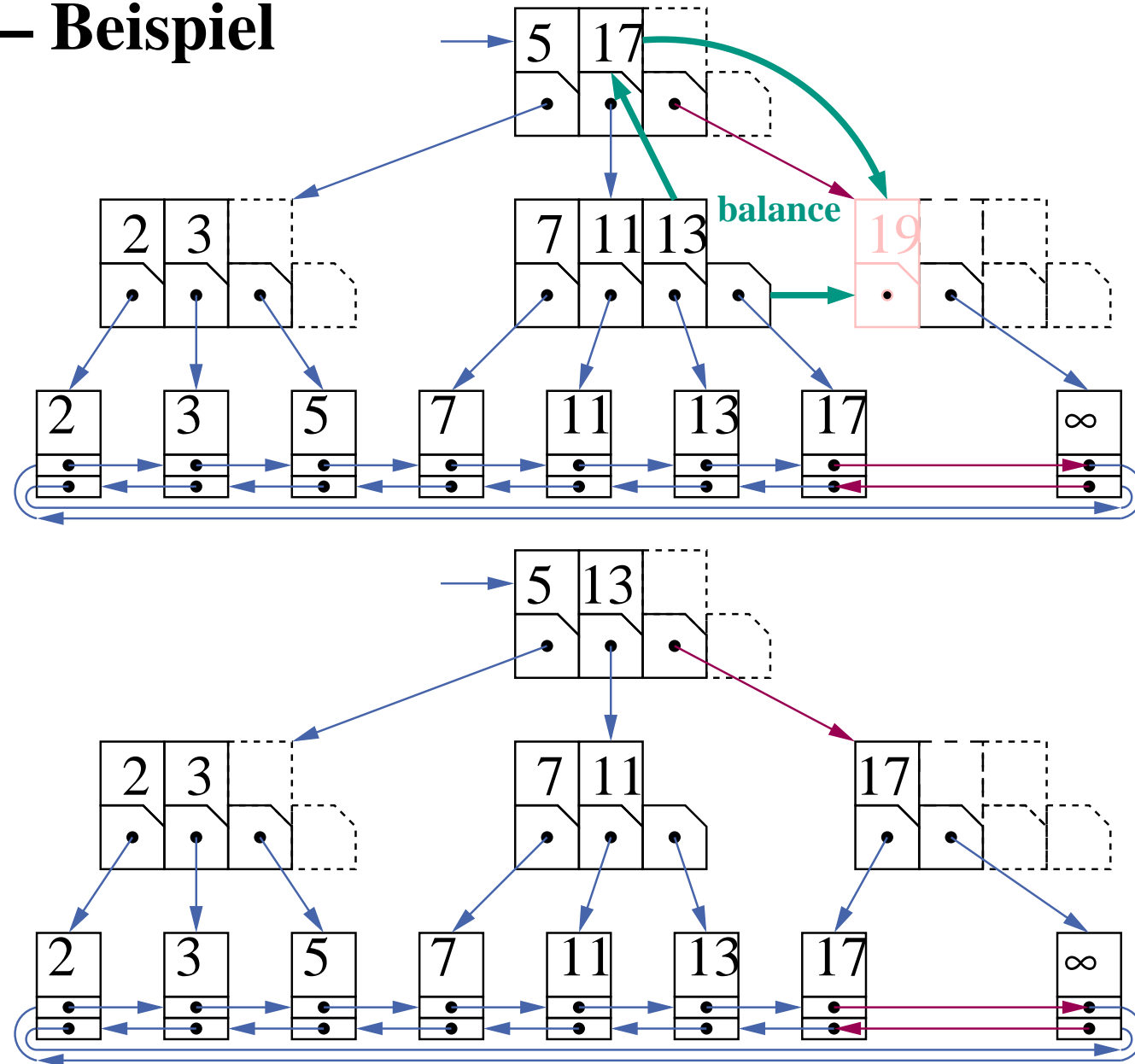
Entfernen – Beispiel



Entfernen – Beispiel



Entfernen – Beispiel



Entfernen – Korrektheit

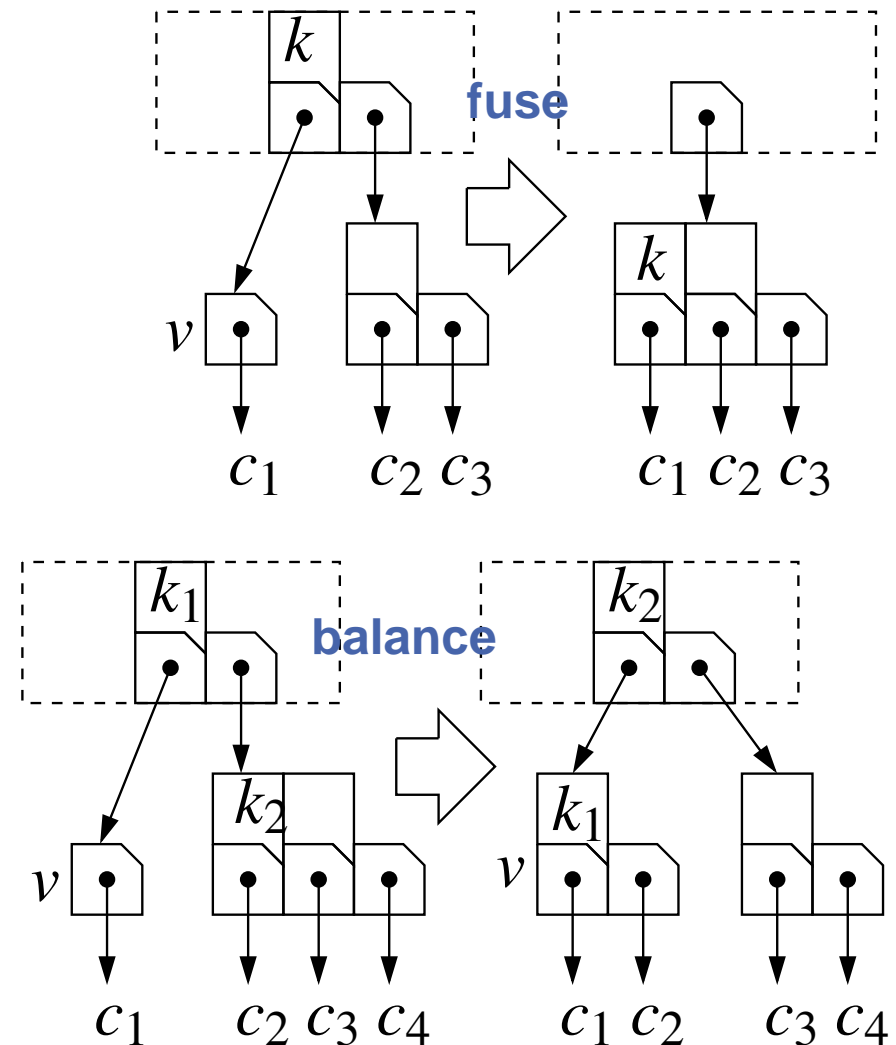
Balancieren: Kein Problem

Nach FUSE

müssen zulässige Items entstehen:

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

hatten wir schon!



Einfügen und Entfernen – Laufzeit

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

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

(a, b) -Bäume

Implementierungsdetails

Etwas kompliziert. . .

Wie merkt man sich das?

Gar nicht!

Man merkt sich:

□ Invarianten

Höhe, Knotengrade

□ Grundideen

split, balance, fuse

Den Rest **leitet** man
sich nach Bedarf **neu her**.

Procedure ABTree::remove(k : KEY)

r .removeRec(k , HEIGHT, ℓ)

if $r.d = 1 \wedge \text{HEIGHT} > 1$ **then** $r' := r$; $r := r'.c[1]$; **dispose** r'

Procedure ABItem::removeRec(k : Key, h : \mathbb{N} , ℓ : List of Element)

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

if $h = 1$ **then**

if $\text{KEY}(c[i] \rightarrow e) = k$ **then**

ℓ .REMOVE($c[i]$)

removeLocally(i)

else

$c[i] \rightarrow \text{REMOVEDREC}(e, h - 1, \ell)$

if $c[i] \rightarrow d < a$ **then**

if $i = d$ **then** $i --$

$s' := \text{CONCATENATE}(c[i] \rightarrow s, \langle s[i] \rangle, c[i + 1] \rightarrow s)$

$c' := \text{CONCATENATE}(c[i] \rightarrow c, c[i + 1] \rightarrow c)$

$d' := |c'|$

if $d' \leq b$ **then** // FUSE

$(c[i + 1] \rightarrow s, c[i + 1] \rightarrow c, c[i + 1] \rightarrow d) := (s', c', d')$

dispose $c[i]$; removeLocally(i)

else // BALANCE

$m := \lceil d' / 2 \rceil$

$(c[i] \rightarrow s, c[i] \rightarrow c, c[i] \rightarrow d) := (s'[1..m - 1], c'[1..m], m)$

$(c[i + 1] \rightarrow s, c[i + 1] \rightarrow c, c[i + 1] \rightarrow d) :=$

$(s'[m + 1..d' - 1], c'[m + 1..d'], d' - m)$

$s[i] := s'[m]$

Procedure ABItem::removeLocally(i : \mathbb{N})

$c[i..d - 1] := c[i + 1..d]$

$s[i..d - 2] := s[i + 1..d - 1]$

$d --$

7.3 Mehr Operationen

min, **max**, **RANGESearch**(a, b): $\langle \text{min}, \dots, a, \dots, b, \dots, \text{max} \rangle$

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: Abgesehen 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.

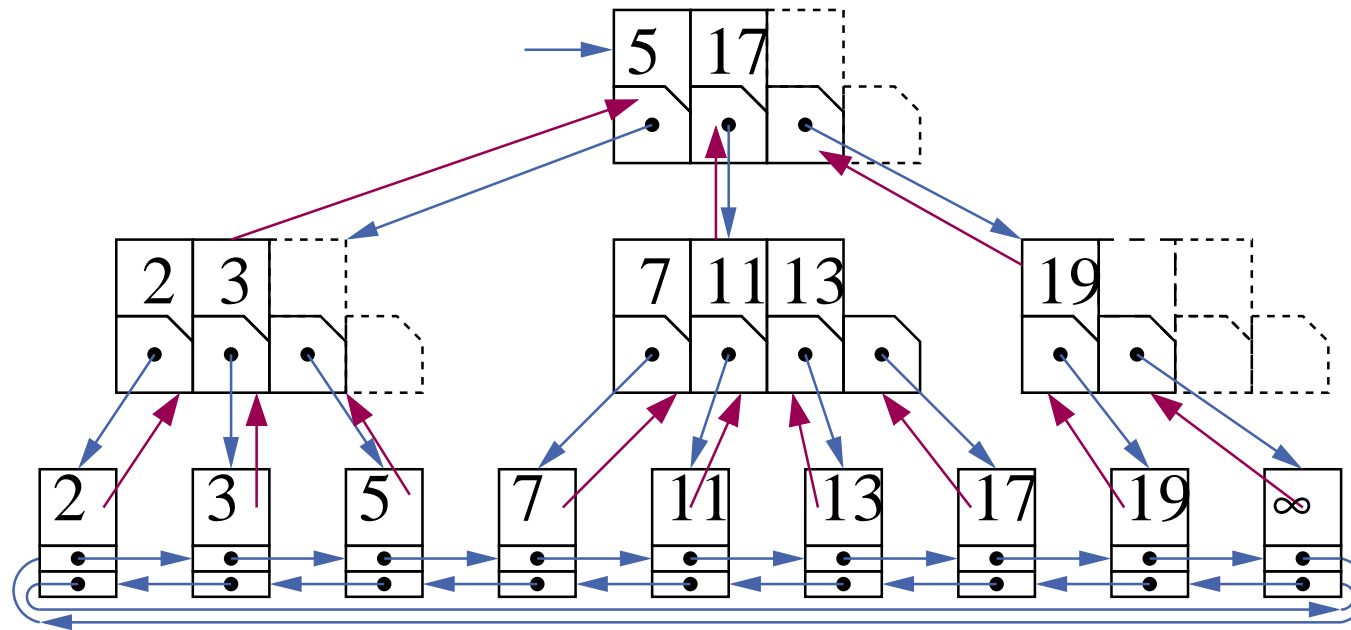
Nachteil: Zeit- und Platzverschwendung

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 $h \rightarrow \text{LEFTSIZE} \geq k$ **then return** select(ℓ, k)

else return select($r, k - \text{LEFTSIZE}$)

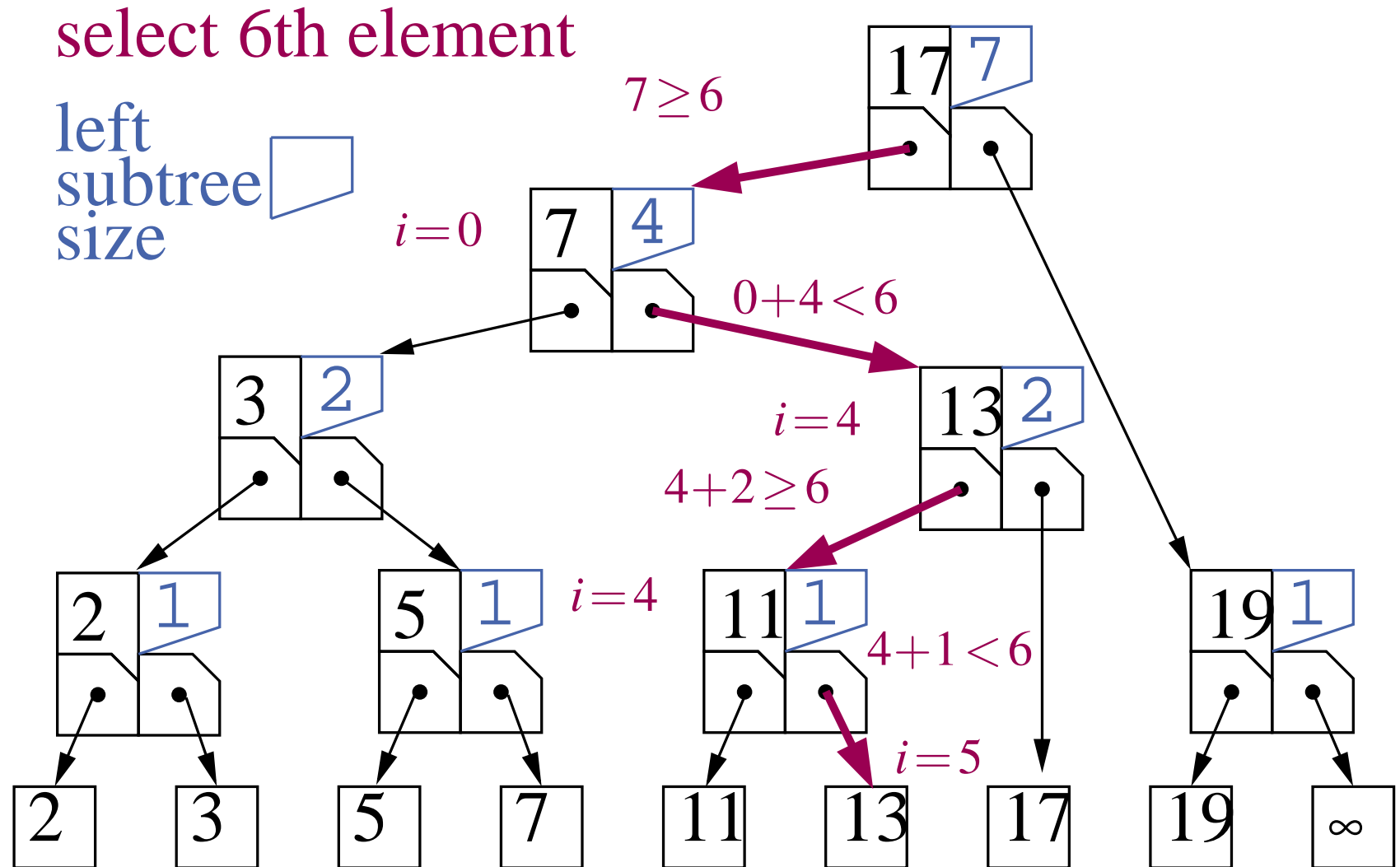
Zeit: $O(\log n)$

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

Übung: Rang eines Elements e bestimmen.

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

7.5.3 Beispiel



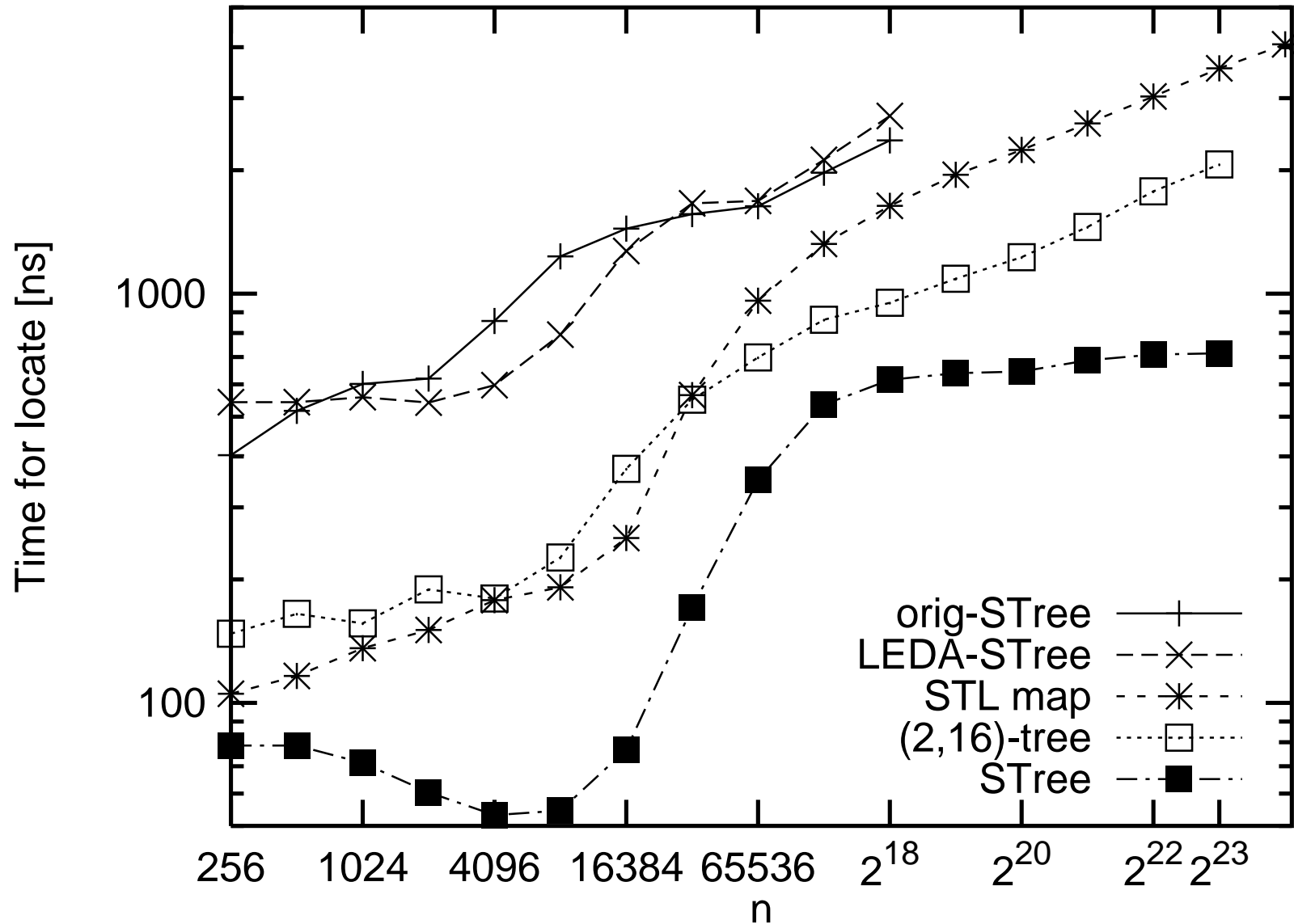
Zusammenfassung

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

Mehr zu sortierten Folgen

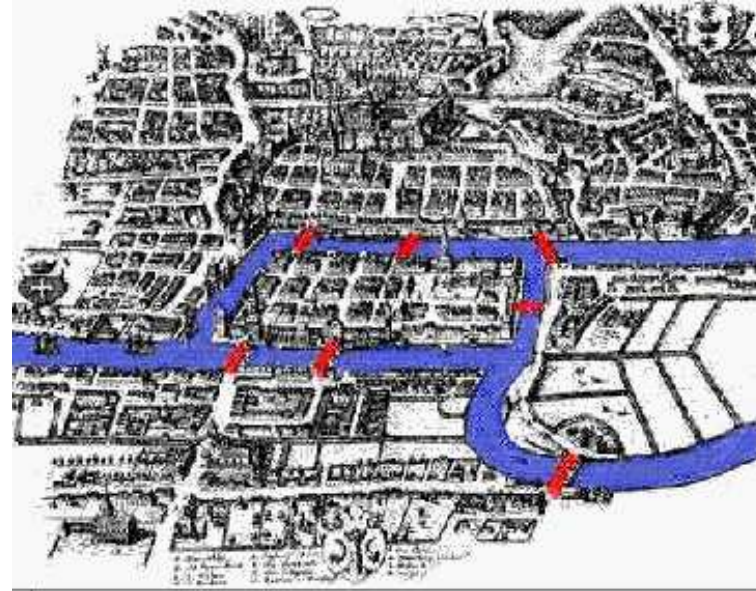
- ☐ Karteikasten \rightsquigarrow Array mit Löchern
- ☐ (a, b) -Bäume sind wichtig für externe Datenstrukturen
- ☐ Ganzzahlige Schlüssel aus $1..U$
 \rightsquigarrow Grundoperationen in Zeit $O(\log \log U)$
- ☐ Verallgemeinerungen: Zeichenketten, mehrdimensionale Daten

Ein paar Zahlen



8 Graphrepräsentation

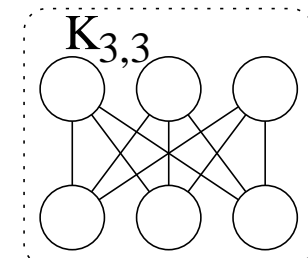
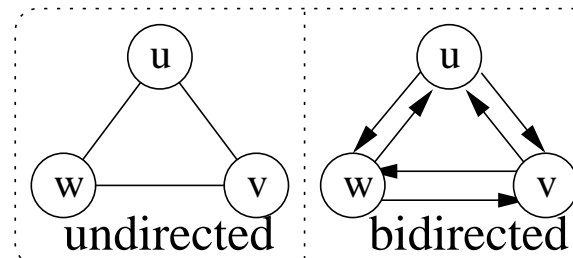
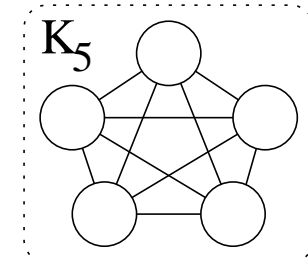
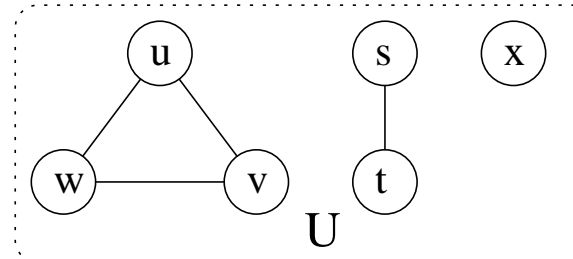
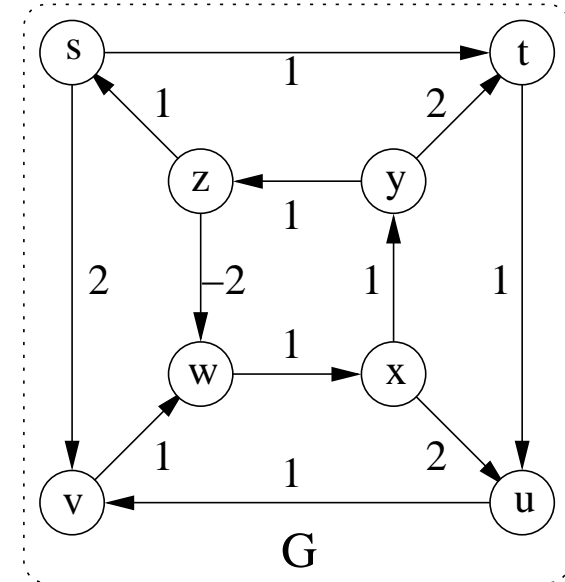
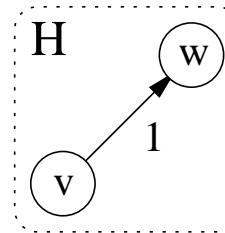
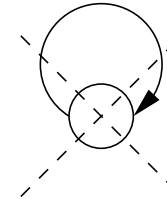
- ☐ 1736 fragt L. Euler die folgende “touristische” Frage:
- ☐ Straßen- oder Computernetzwerke
- ☐ Zugverbindungen (Raum und Zeit)
- ☐ Soziale Netzwerke (Freundschafts-, Zitier-, Empfehlungs-,...)
- ☐ Aufgabenabhängigkeiten \rightsquigarrow scheduling Probleme
- ☐ Werte und arithmetische Operationen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



Graphrepräsentation

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

self-loop



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.

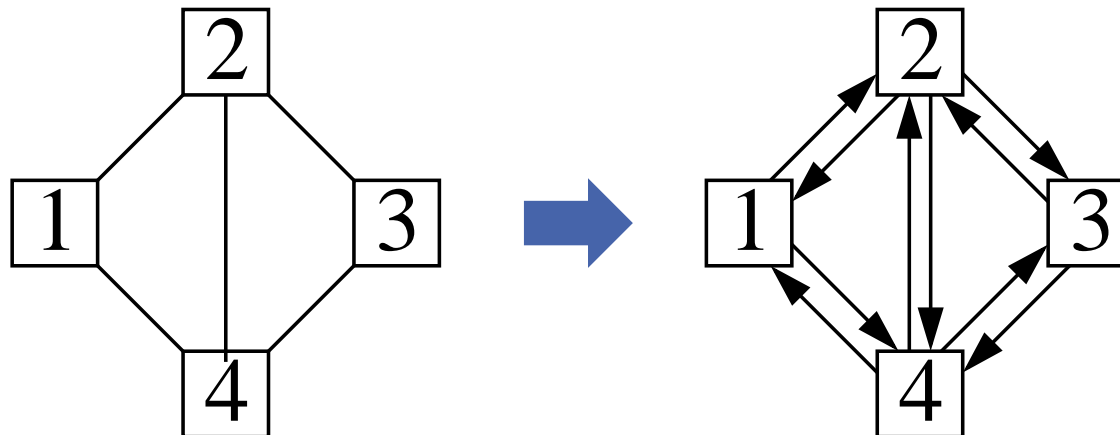
Das gilt für die ganze theoretische Informatik

Ungerichtete \rightarrow 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

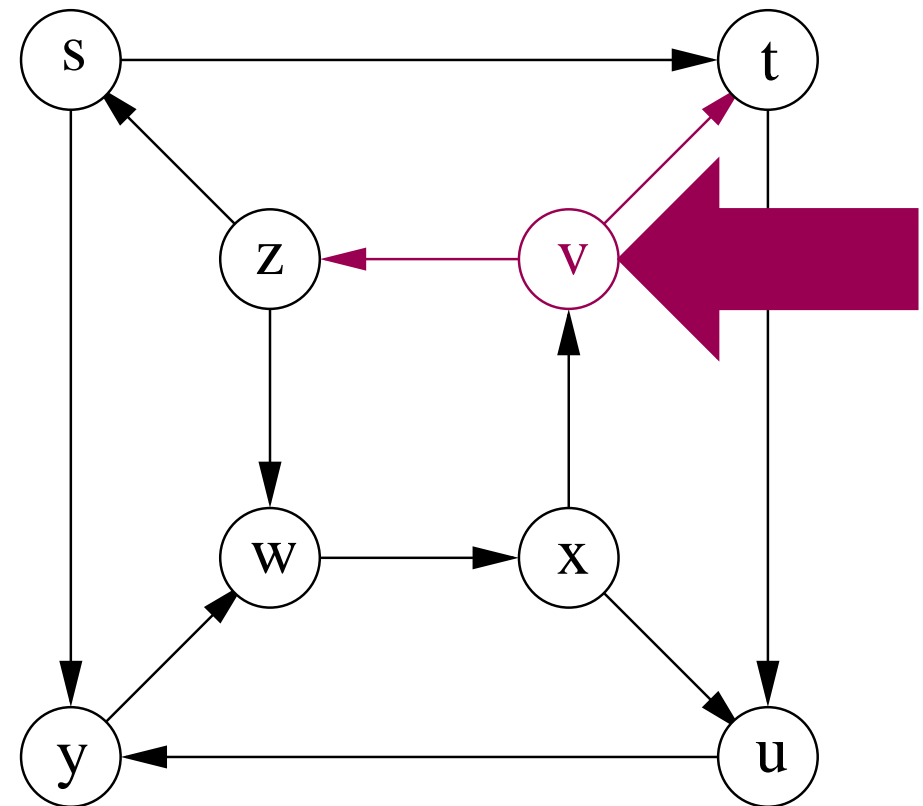
Konstruktion, Konversion und Ausgabe

($O(m + n)$ Zeit)

Navigation: Gegeben v ,
finde ausgehende Kanten.

Dynamische Graphen

Knoten/Kanten einfügen/löschen

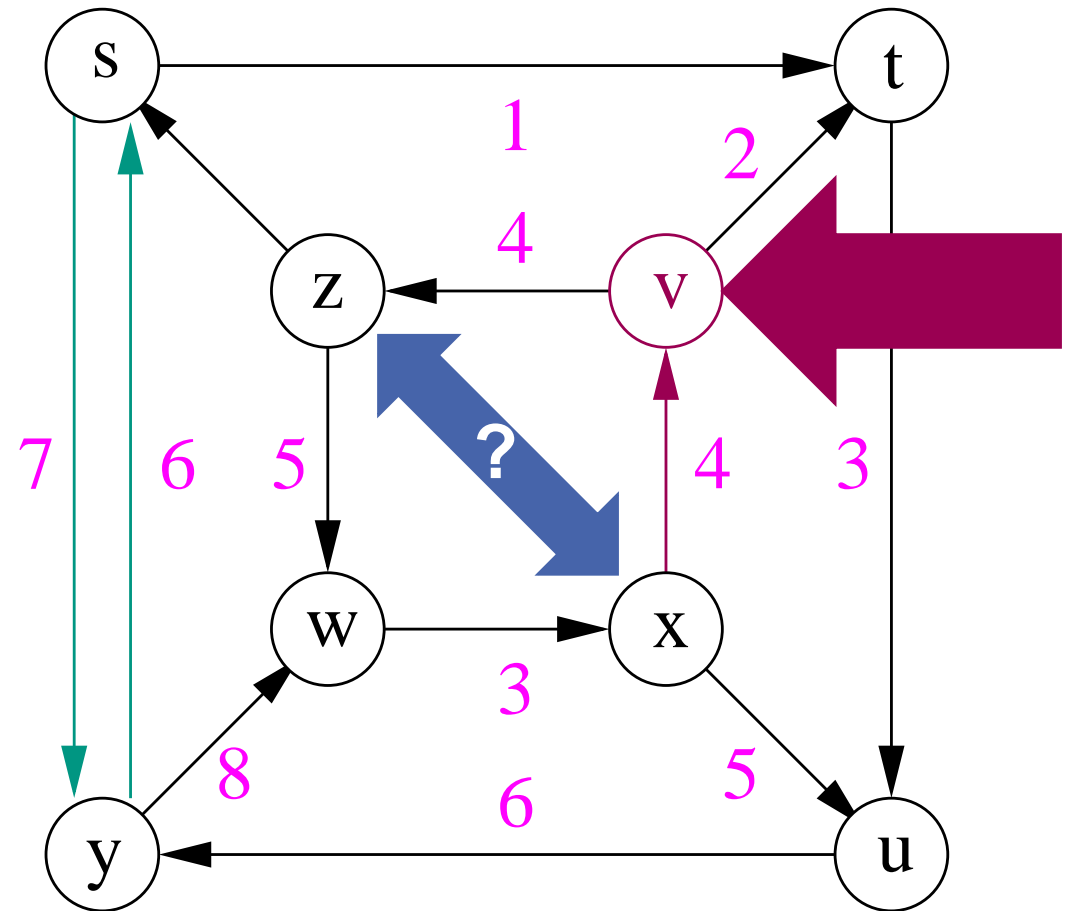


Weitere Operationen

Zugriff auf assoziierte Information

mehr Navigation: finde
eingehende Kanten

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



Kantenfolgenrepräsentation

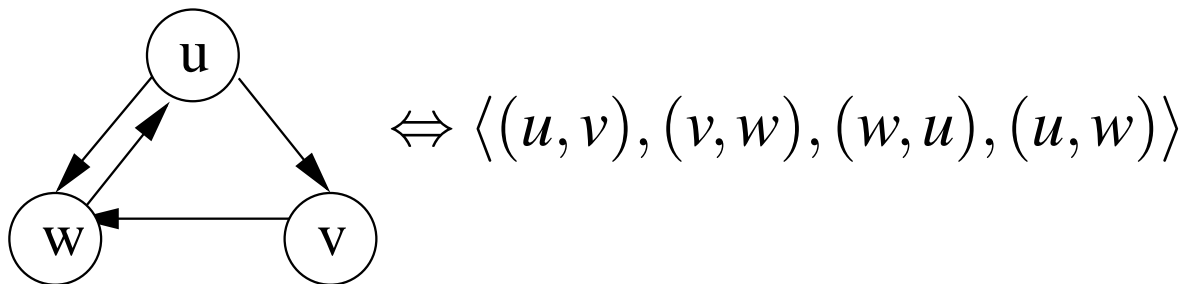
Folge von Knotenpaaren (oder Tripel mit Kantengewicht)

+ kompakt

+ gut für I/O

— Fast keine nützlichen Operationen ausser alle Kanten durchlaufen

Beispiele: Übung: isolierte Knoten suchen,
Kruskals MST-Algorithmus (später), Konvertierung.



Adjazenzfelder

$$V = 1..n$$

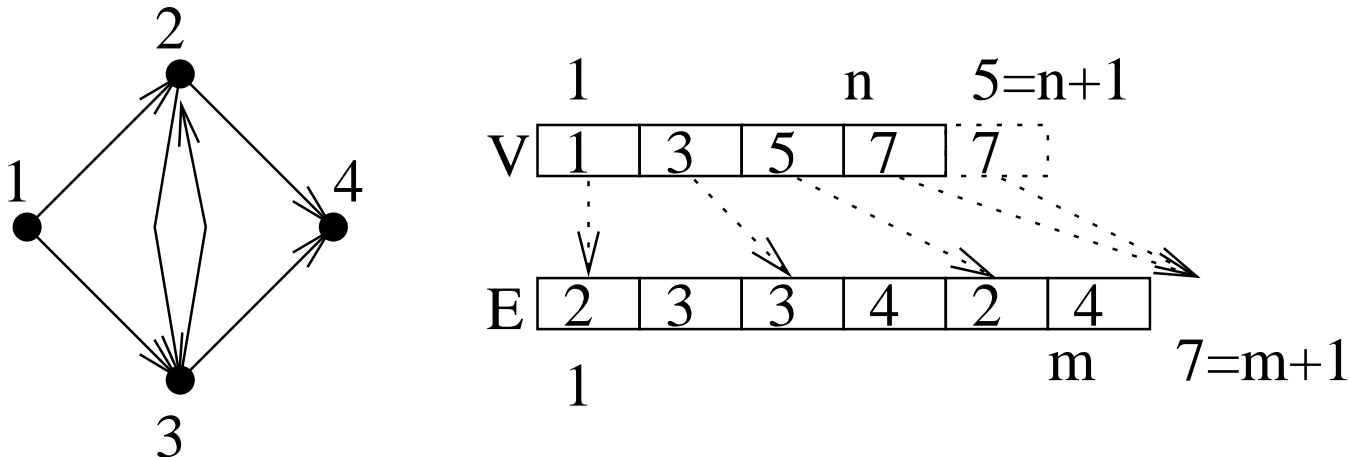
oder $0..n - 1$

Kantenfeld E speichert Ziele

gruppiert nach Startknoten

∇ speichert Index der ersten ausgehenden Kante

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



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

Kantenliste \rightarrow Adjazenzfeld

Zur Erinnerung: KSORT

Function adjacencyArray(EDGELIST)

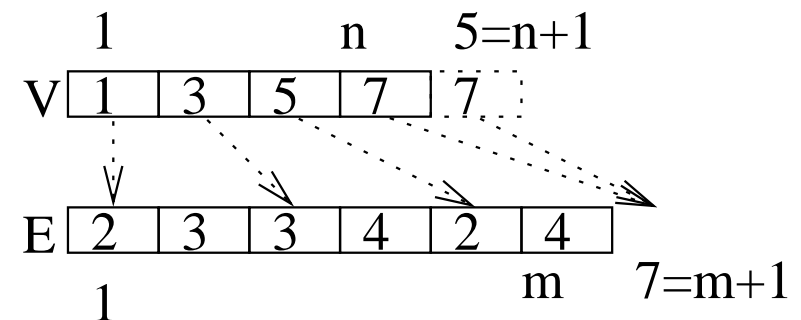
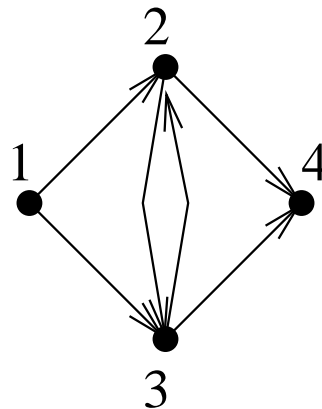
$V = \langle 1, 0, \dots, 0 \rangle$: **Array** $[1..n+1]$ **of** \mathbb{N}

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

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

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

return (V, E)



Operationen für Adjanzenzfelder

Navigation: einfach

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

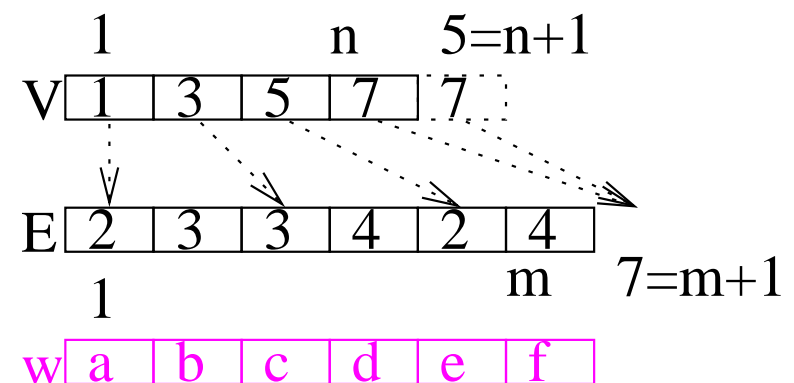
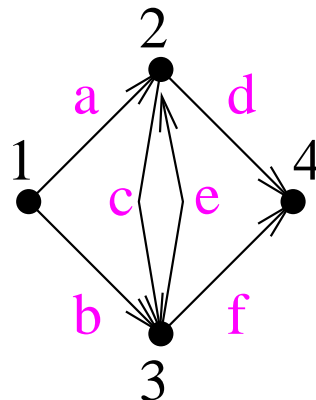
Knoteninfos: V wird Feld von Records (oder mehrere Felder)

Eingehende Kanten: umgedrehten Graphen speichern

Kanten löschen: explizite Endindizes

Batched Updates:

neu aufbauen



Kantenanfragen

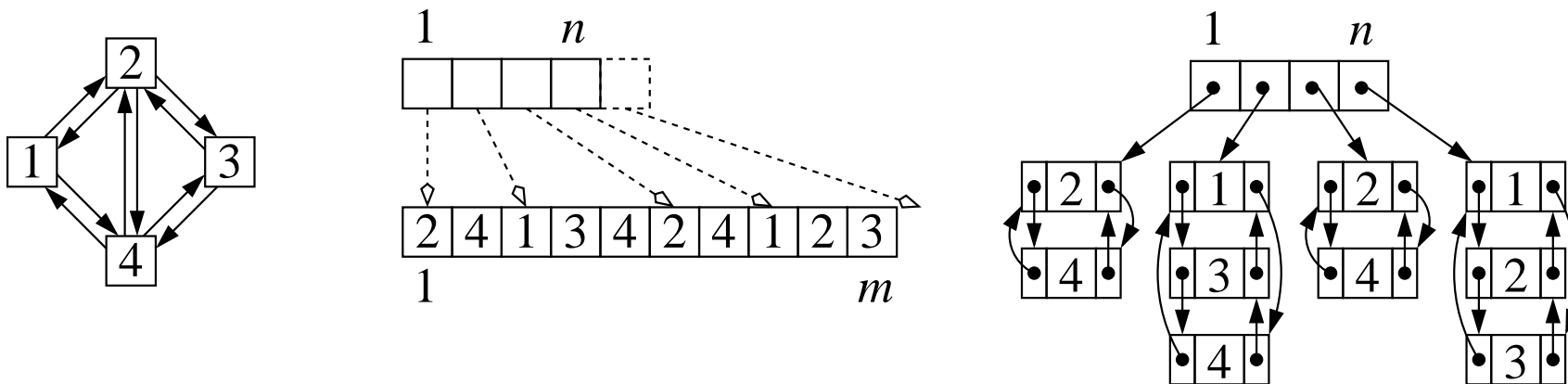
Hashtabelle H_E speichert (ggf. zusätzlich) alle Kanten.

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



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 = $\{v \in V : \text{INDEGREE}[v] = 0\}$: Stack

while DROPPABLE $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

$v := \text{DROPPABLE.POP}$

DROPPED ++

foreach edge $(v, w) \in E$ **do**

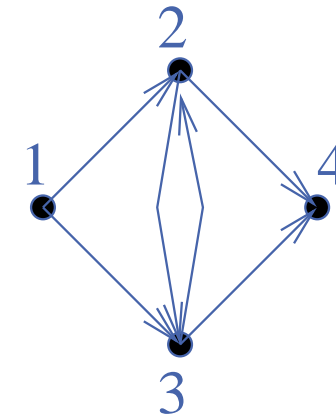
inDegree[w] --

if inDegree[w] = 0 **then** droppable.push(w)

return $|V| = \text{DROPPED}$

Adjazenz-Matrix

$$A \in \{0, 1\}^{n \times n} \text{ with } A(i, j) = [(i, j) \in E]$$



- + platzeffizient für sehr **dichte Graphen**
- — platz**ineffizient** 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

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Übung: zähle Pfade der Länge $\leq k$

Wichtige **Beschleunigungstechniken**:

$O(\log k)$ 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

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

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

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

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

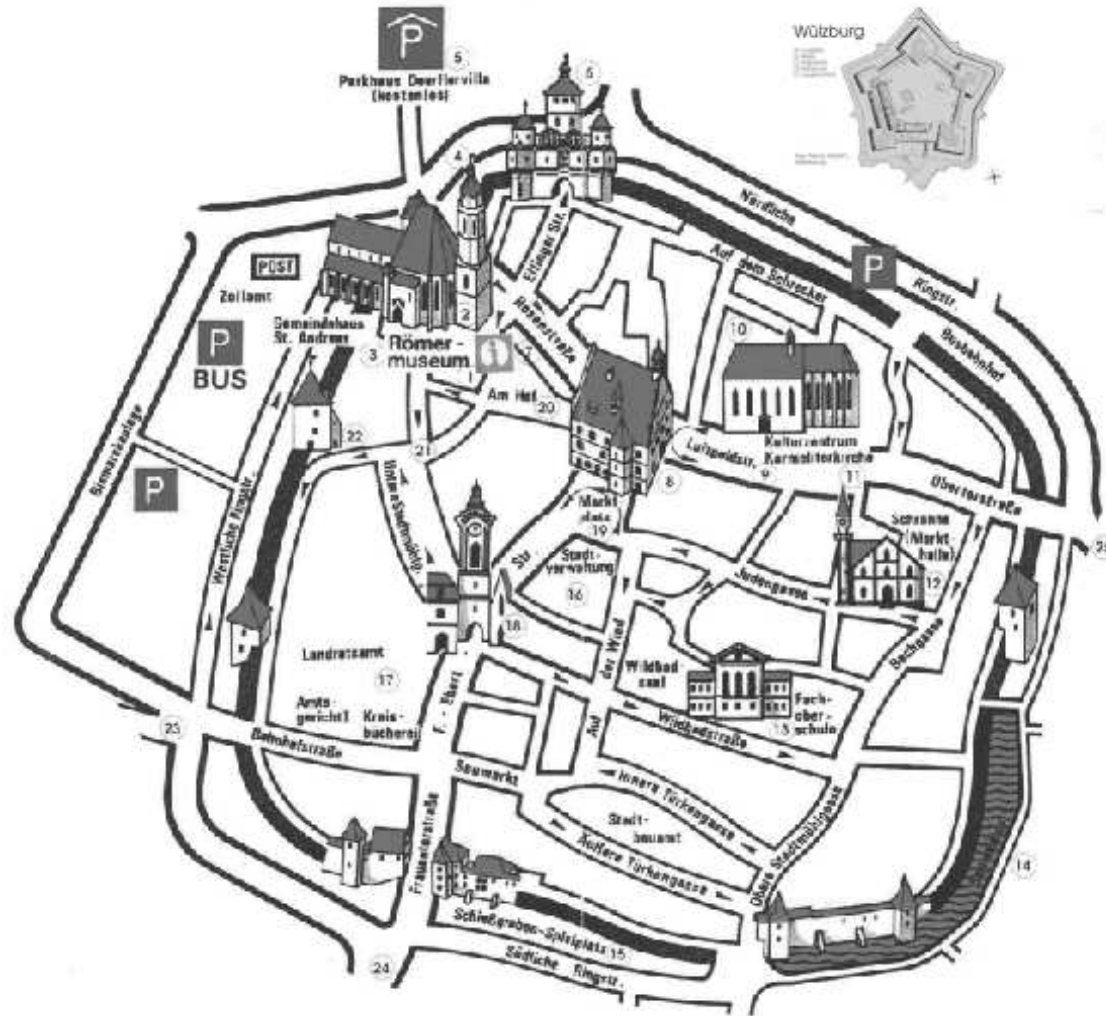
$O(n \log n)$ Algorithmus für bis zu $O(n^2)$ Kanten!

Übung: Zusammenhangskomponenten finden

Graphrepräsentation: Zusammenfassung

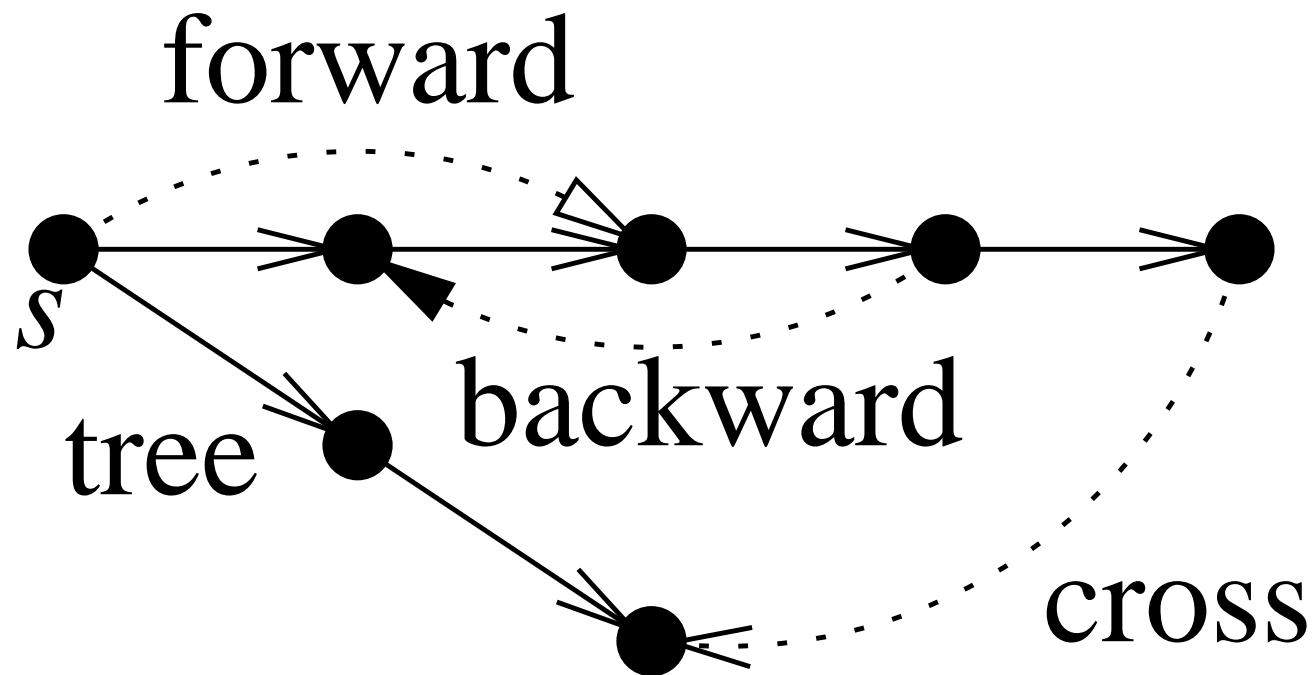
- ☐ Welche Operationen werden gebraucht?
- ☐ Wie oft?
- ☐ Adjazenzarrays gut für statische Graphen
- ☐ Pointer \rightsquigarrow flexibler aber auch teurer
- ☐ Matrizen eher konzeptionell interessant

9 Graphtraversierung



Ausgangspunkt oder Baustein fast jedes nichtrivialen Graphenalgorithmus

Graphtraversierung als Kantenklassifizierung

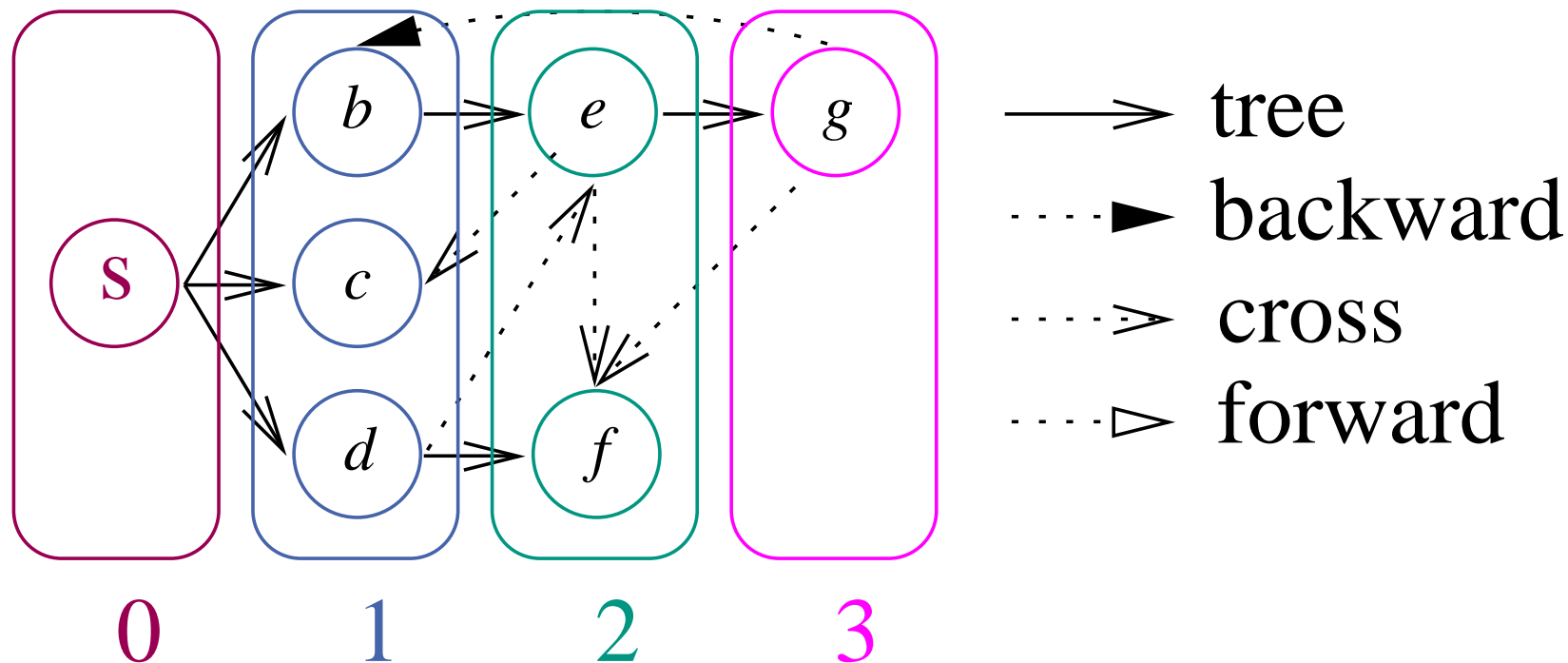


9.1 Breitensuche

Baue Baum von **Startknoten s**

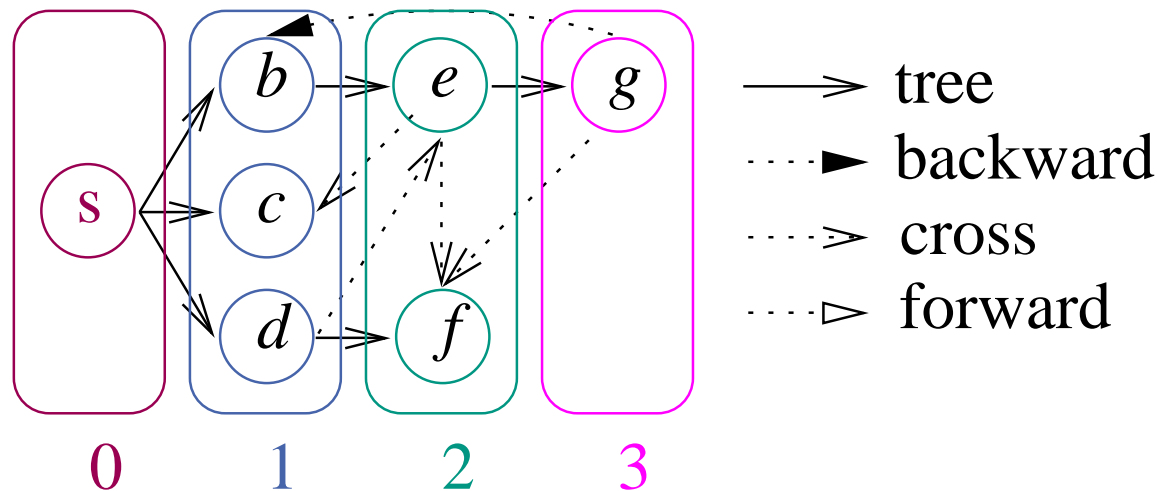
der alle von s erreichbaren Knoten

mit möglichst **kurzen** Pfaden erreicht. Berechne Abstände



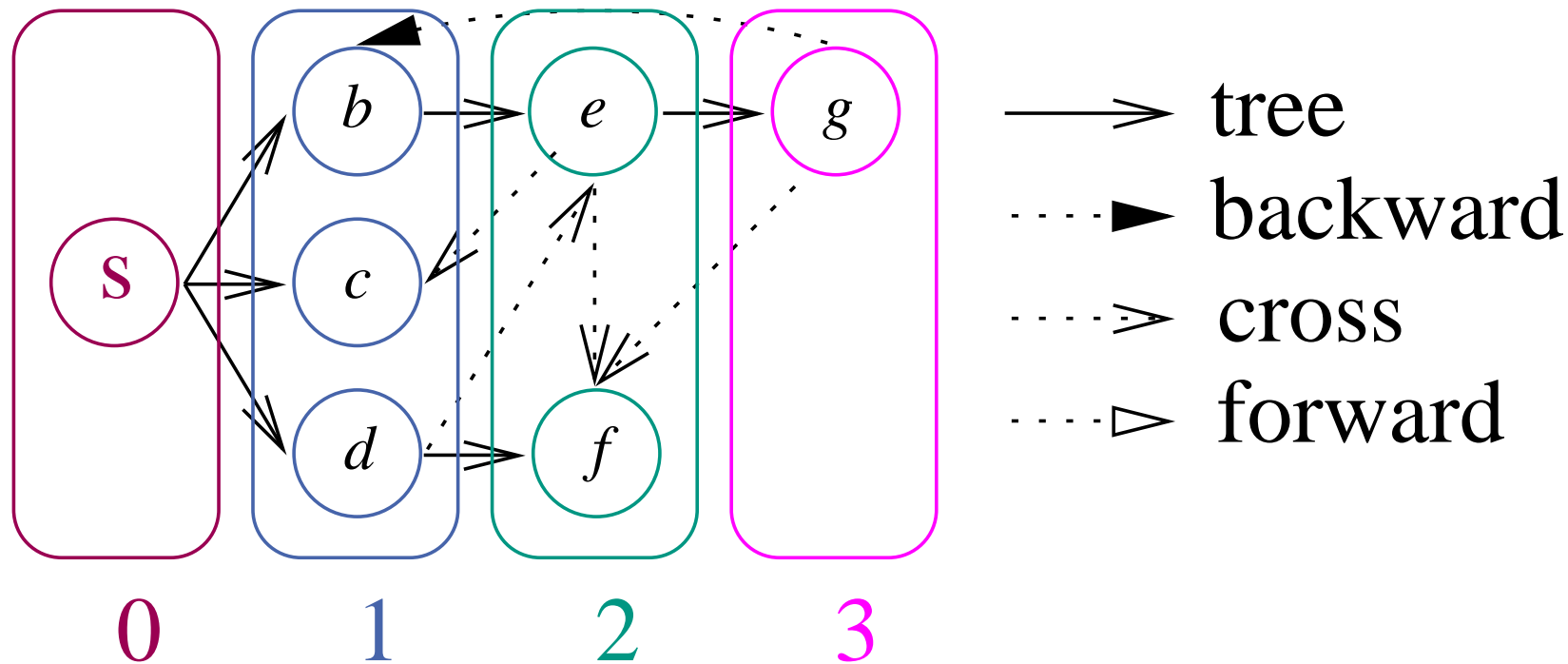
Breitensuche

- 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) :

$Q := \langle s \rangle$

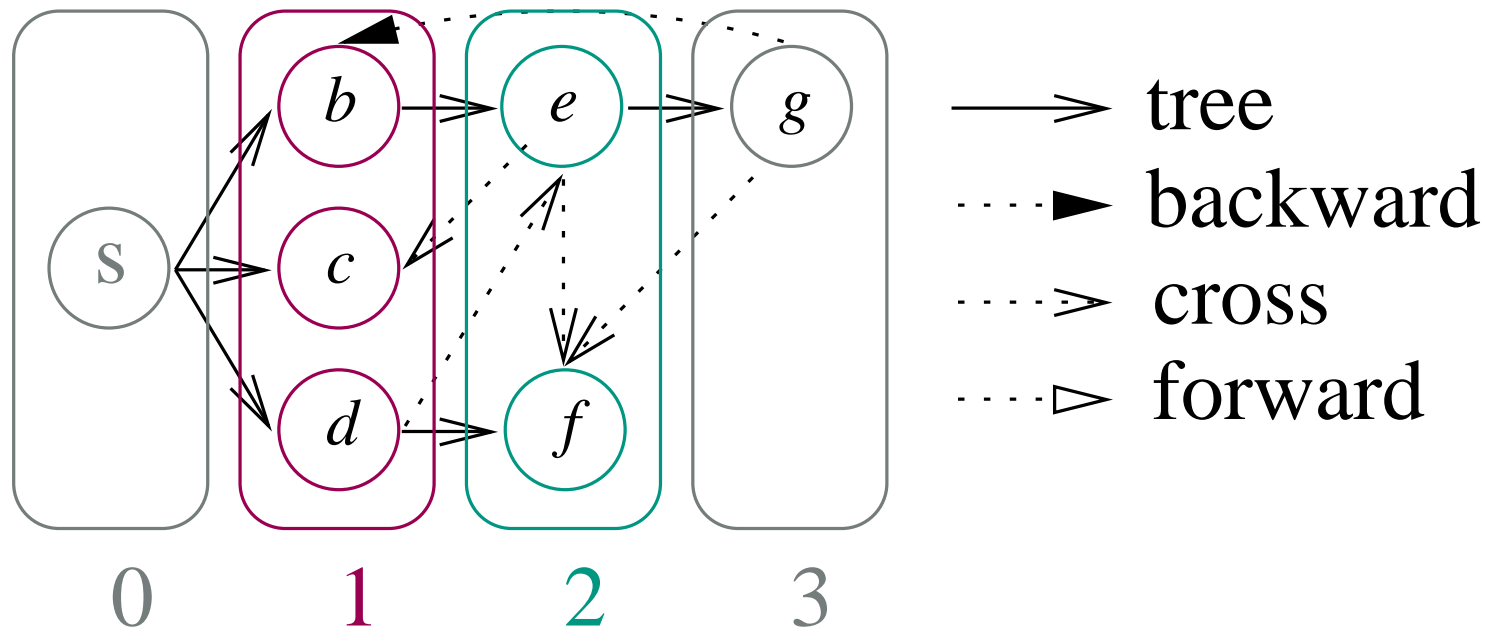
// aktuelle Schicht

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

exploriere Knoten in Q

merke dir Knoten der nächsten Schicht in Q'

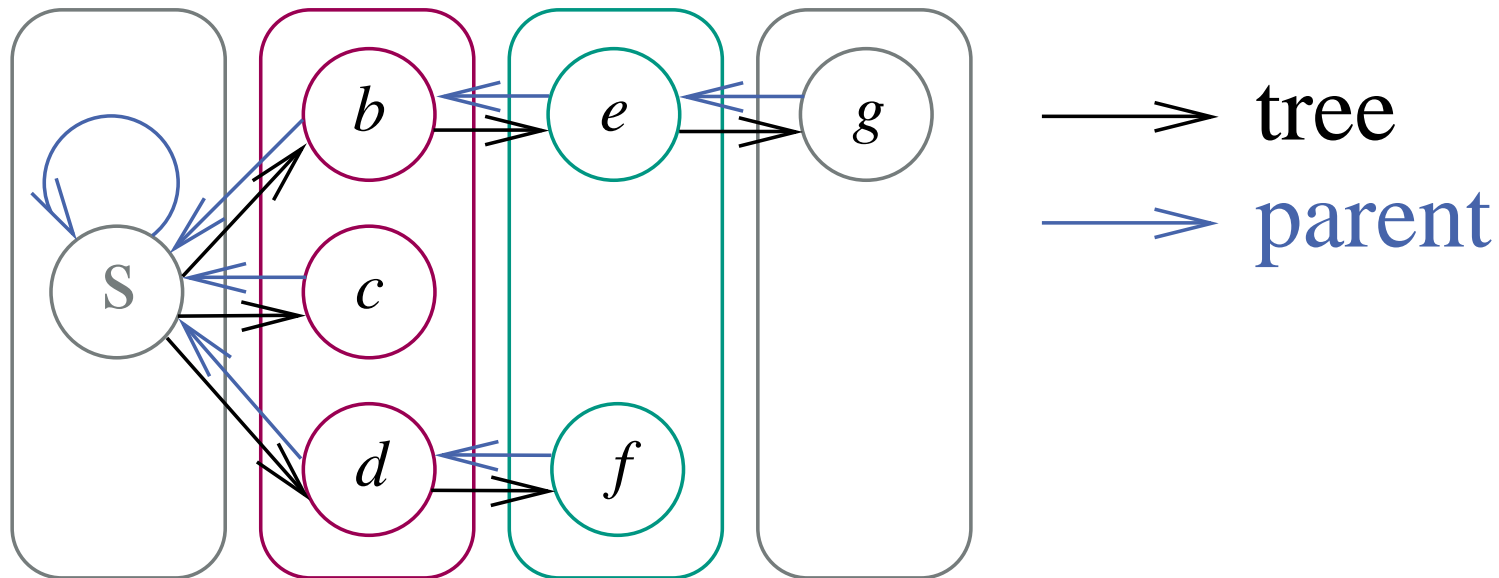
$Q := Q'$



Repräsentation des Baums

Feld **PARENT** speichert Vorgänger.

- noch nicht erreicht: $\text{PARENT}[v] = \perp$
- Startknoten/Wurzel: $\text{PARENT}[s] = s$



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

$d = \langle \infty, \dots, \infty \rangle$: NodeArray of $\mathbb{N}_0 \cup \{\infty\}$; $d[s] := 0$

parent = $\langle \perp, \dots, \perp \rangle$: NodeArray of NodeId; parent[s] := s

$Q = \langle s \rangle, Q' = \langle \rangle$: Set of NodeId // current, next layer

for ($\ell := 0$; $Q \neq \langle \rangle$; $\ell++$)

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) = \perp **then** // unexplored

$Q' := Q' \cup \{v\}$

$d[v] := \ell + 1$; parent(v) := u

$(Q, Q') := (Q', \langle \rangle)$ // next layer

return (PARENT, d) // BFS = $\{(v, w) : w \in V, v = \text{PARENT}(w)\}$

Repräsentation von Q und Q'

☐ 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

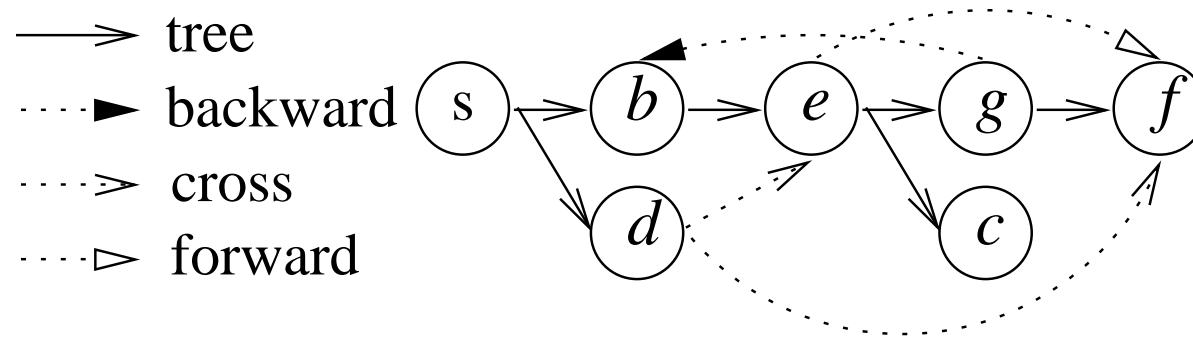
— Korrektheit weniger evident

Übung?

= Effizient (?)

Übung: ausprobieren?

9.2 Tiefensuche



Tiefensuchschema für $G = (V, E)$

unmark all nodes; **init**

foreach $s \in V$ **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($u, v : \text{NODEID}$) // Explore v coming from u .

foreach $(v, w) \in E$ **do**

if w is marked **then** **traverseNonTreeEdge**(v, w)

else **TRAVERSETREEEDGE**(v, w)

 mark w

 DFS(v, w)

backtrack(u, v) // return from v along the incoming edge

DFS Baum

INIT:

 $\text{PARENT} = \langle \perp, \dots, \perp \rangle : \text{NODEARRAY of NODEID}$
ROOT(s):
 $\text{PARENT}[s] := s$

 TRAVERSE TREE EDGE(v, w): $\text{PARENT}[w] := v$
 \longrightarrow tree

 \longrightarrow parent

mark s

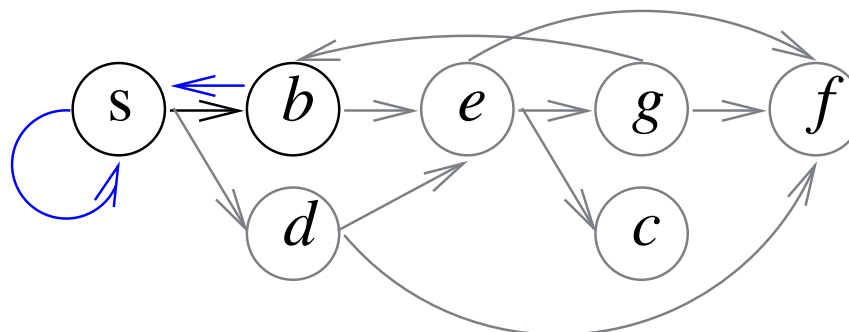
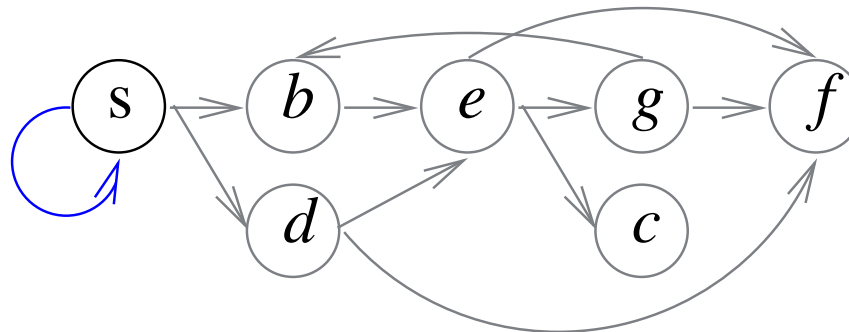
root(s)

dfs(s,s)

traverseTreeEdge(s,b)

mark b

dfs(s,b)



dfs(s,b)

traverseTreeEdge(b,e)

mark(e)

dfs(b,e)

traverseTreeEdge(e,g)

mark(g)

dfs(e,g)

traverseNonTreeEdge(g,b)

traverseTreeEdge(g,f)

mark(f)

dfs(g,f)

backtrack(g,f)

backtrack(e,g)

traverseNonTreeEdge(e,f)

traverseTreeEdge(e,c)

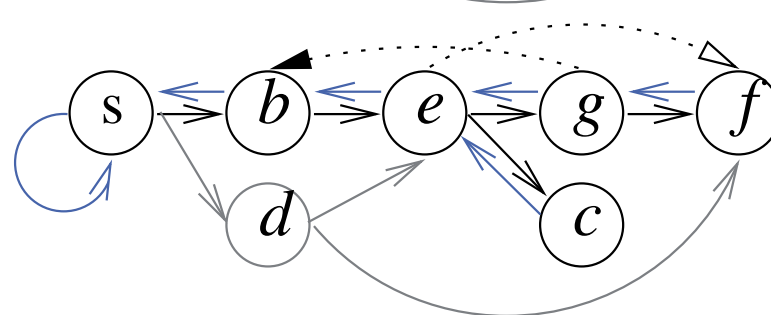
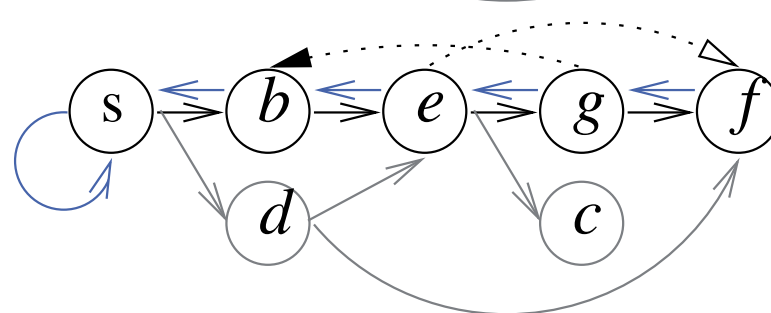
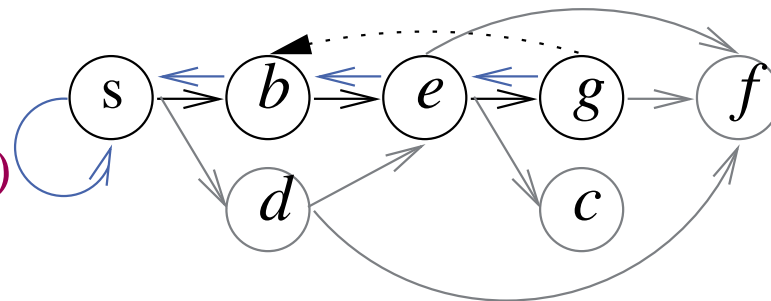
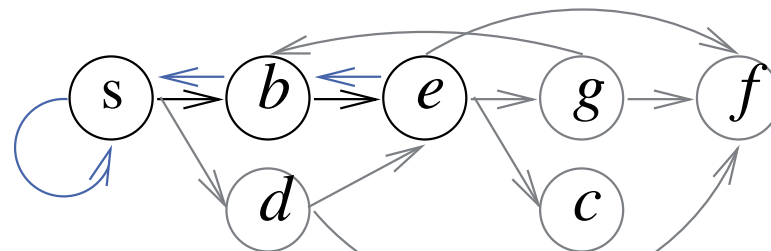
mark(c)

dfs(e,c)

backtrack(e,c)

backtrack(b,e)

backtrack(s,b)



traverseTreeEdge(s,d)

mark(d)

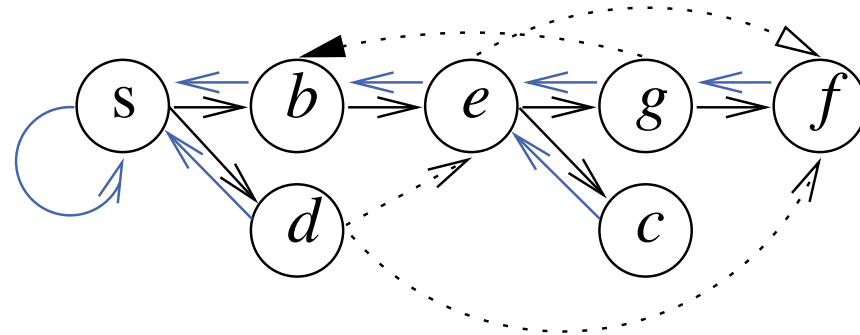
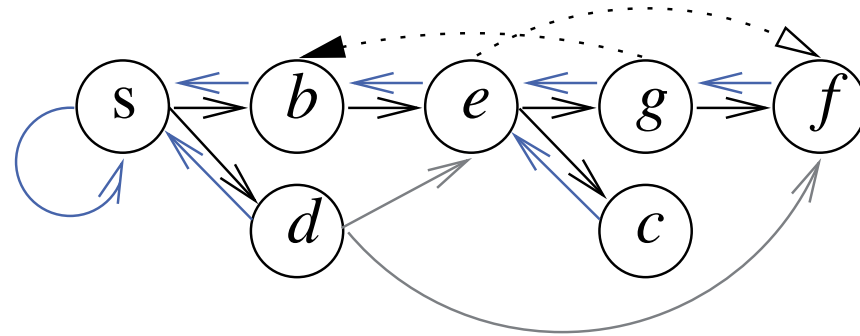
dfs(s,d)

traverseNonTreeEdge(d,e)

traverseNonTreeEdge(d,f)

backtrack(s,d)

backtrack(s,s)



DFS Nummerierung

INIT: DFSPOS = 1 : 1..n

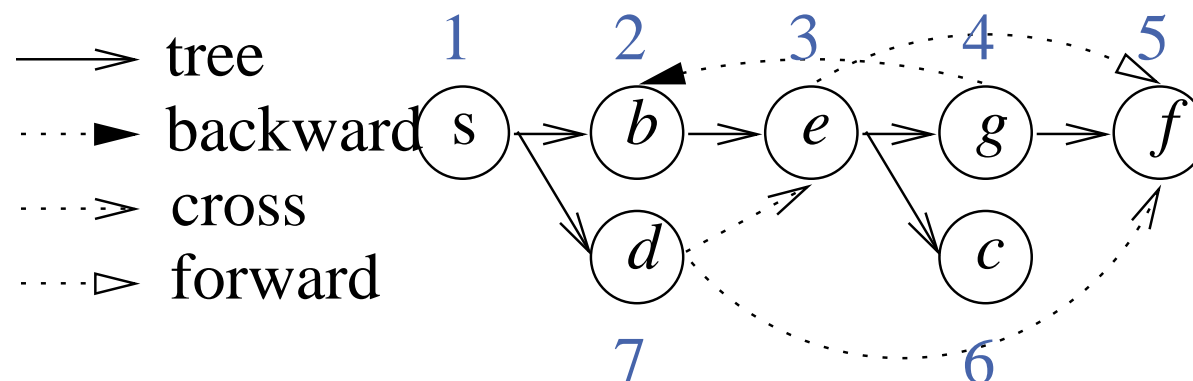
ROOT(s): DFSNUM[s] := DFSPOS++

TRAVERSE TREE EDGE(v, w): DFSNUM[w] := DFSPOS++

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

Beobachtung:

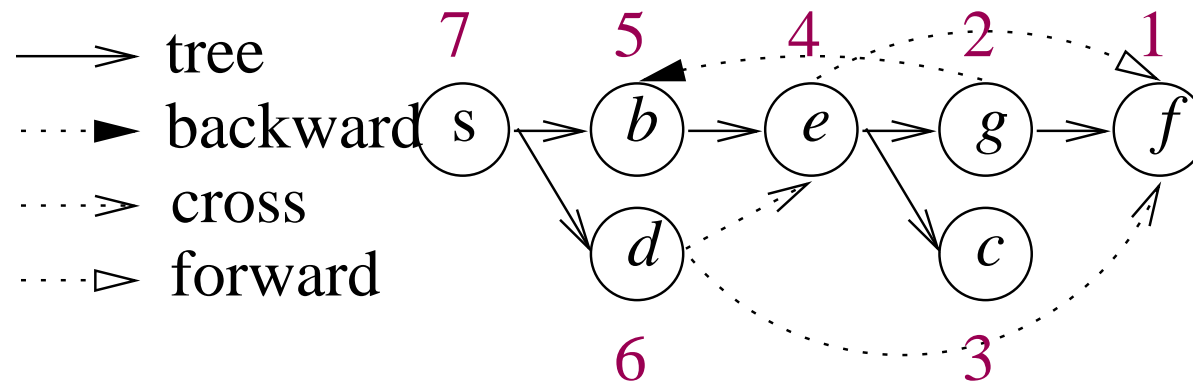
Knoten auf dem Rekursionsstapel sind bzgl., \prec sortiert



Fertigstellungszeit

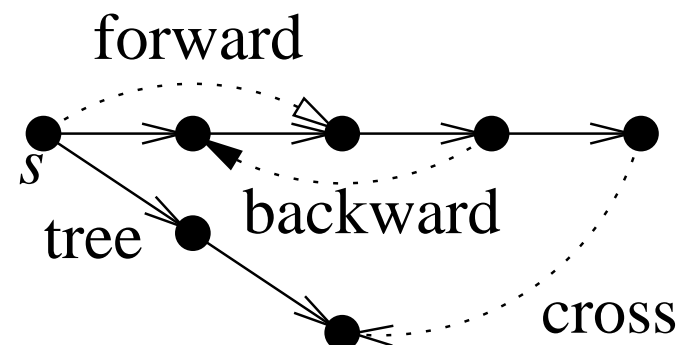
INIT: FINISHINGTIME = 1 : 1..n

BACKTRACK(u, v): **FINISHTIME**[v] := FINISHINGTIME++



Kantenklassifizierung bei DFS

type (v, w)	$\text{DFSNUM}[v] < \text{DFSNUM}[w]$	$\text{FINISHTIME}[w] < \text{FINISHTIME}[v]$	w is marked
tree	yes	yes	no
forward	yes	yes	yes
backward	no	no	yes
cross	no	yes	yes



Topologisches Sortieren mittels DFS

Satz:

G ist **kreisfrei (DAG)** \Leftrightarrow DFS findet keine Rückwärtskante.

In diesem Fall liefert

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

eine **topologische Sortierung**,

d. h.

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

Topologisches Sortieren mittels DFS

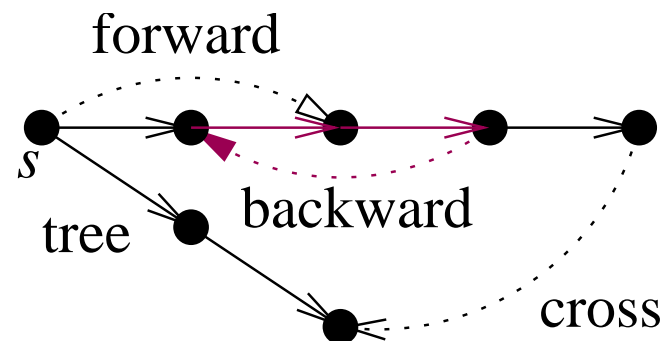
Satz: G **kreisfrei (DAG)** \Leftrightarrow DFS findet keine Rückwärtskante.

In diesem Fall liefert $t(v) := n - \text{FINISHTIME}[v]$ eine **topologische Sortierung**, d. h. $\forall (u, v) \in E : t(u) < t(v)$.

Beweis “ \Rightarrow ”: Annahme \exists Rückwärtskante.

Zusammen mit Baumkanten ergibt sich ein Kreis.

Widerspruch.



Topologisches Sortieren mittels DFS

Satz: G **kreisfrei (DAG)** \Leftrightarrow DFS findet keine Rückwärtskante.

In diesem Fall liefert $t(v) := n - \text{FINISHTIME}[v]$ eine **topologische Sortierung**, d. h. $\forall (u, v) \in E : t(u) < t(v)$.

Beweis “ \Leftarrow ”:

Keine Rückwärtskante

Kantenklassifizierung

$$\overset{\curvearrowright}{\Rightarrow} \quad \forall (v, w) \in E : \text{FINISHTIME}[v] > \text{FINISHTIME}[w]$$

\Rightarrow FINISHTIME definiert umgekehrte topologische Sortierung.

Starke Zusammenhangskomponenten

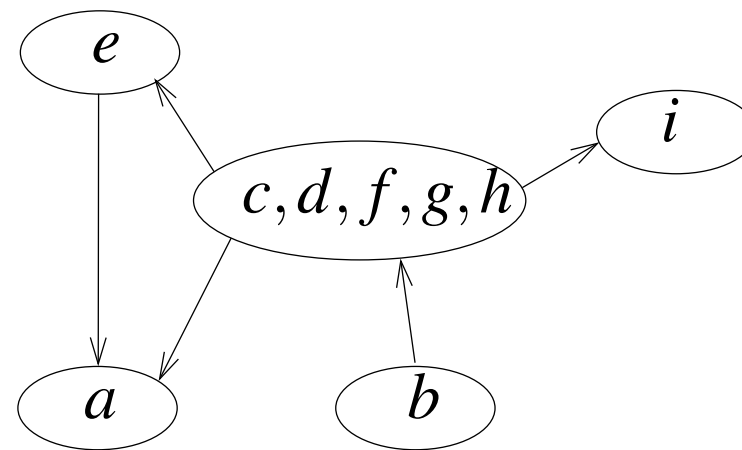
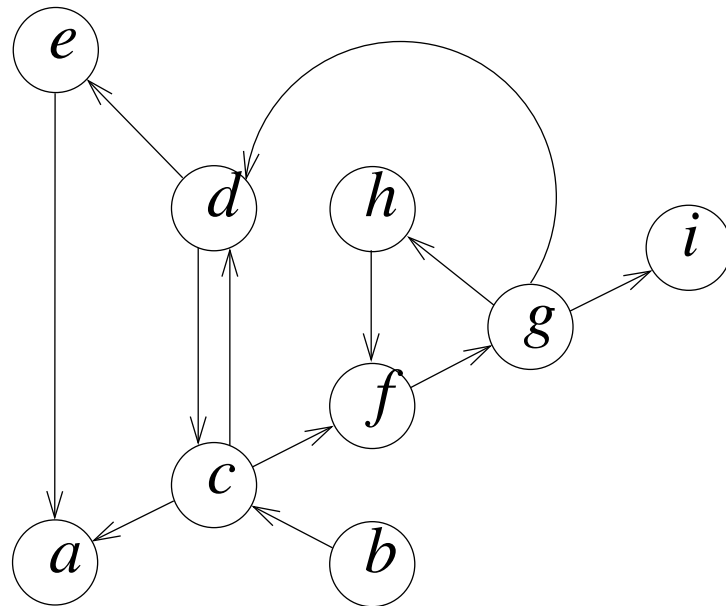
Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit

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

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

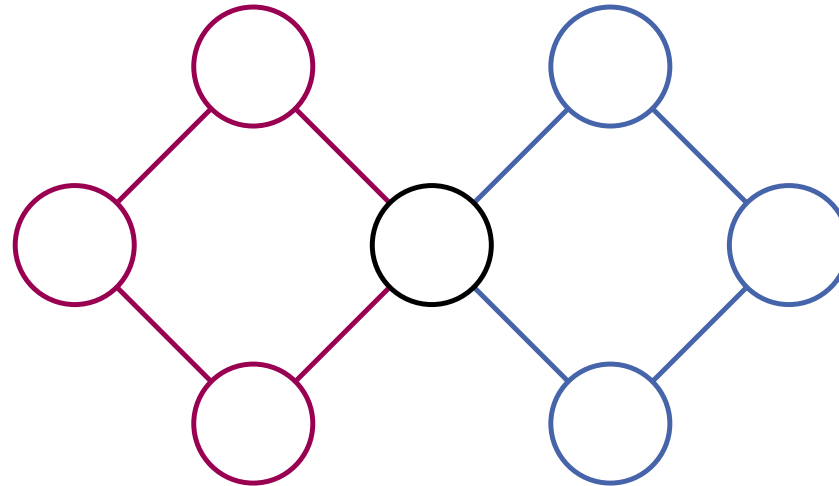
Übung

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

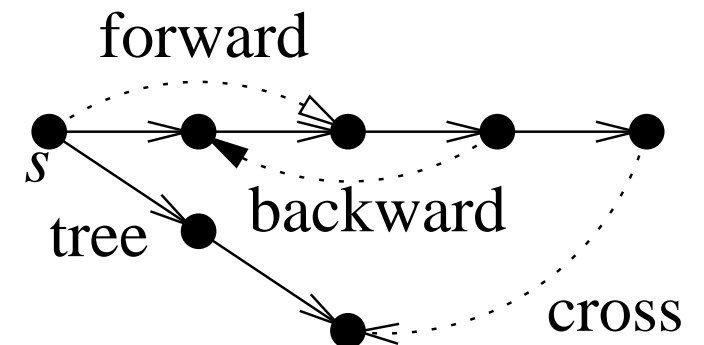
BFS \longleftrightarrow DFS

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 $c : E \rightarrow \mathbb{R}$

Anfangsknoten s .

Ausgabe: für alle $v \in V$

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

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

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

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



Anwendungen

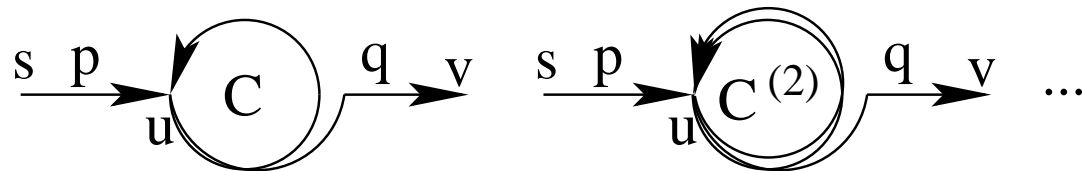
- ☐ Routenplanung
 - Strassennetze
 - Spiele
 - Kommunikationsnetze
- ☐ Unterprogramm
 - Flüsse in Netzwerken
 - ...
- ☐ Tippfehlerkorrektur
- ☐ Disk Scheduling
- ☐ ...



10.1 Grundlagen

Gibt es immer einen kürzesten Pfad?

Es kann **negative Kreise** geben!



weitere Grundlagen just in time

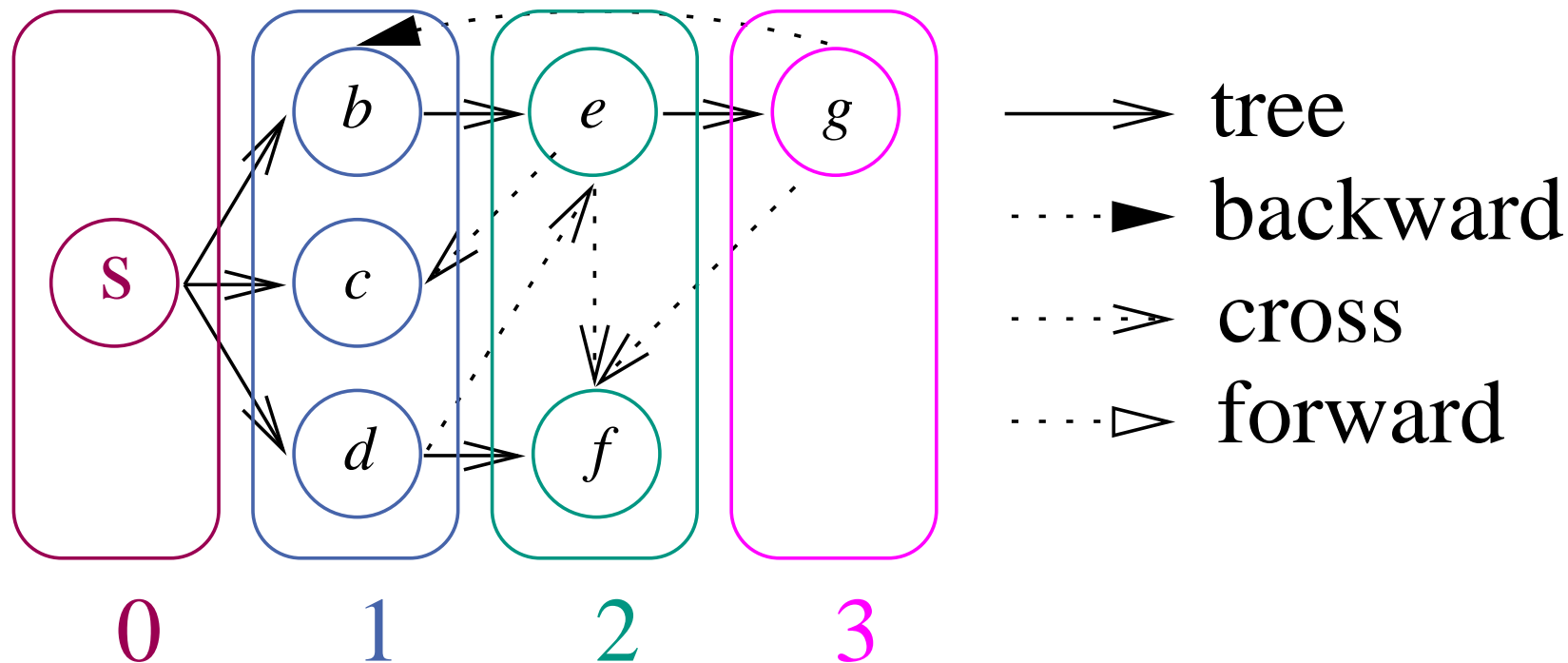
10.2 Azyklische Graphen

später

10.3 Kantengewichte ≥ 0

Alle Gewichte gleich:

Breitensuche (BFS)!



Dijkstra's Algorithmus

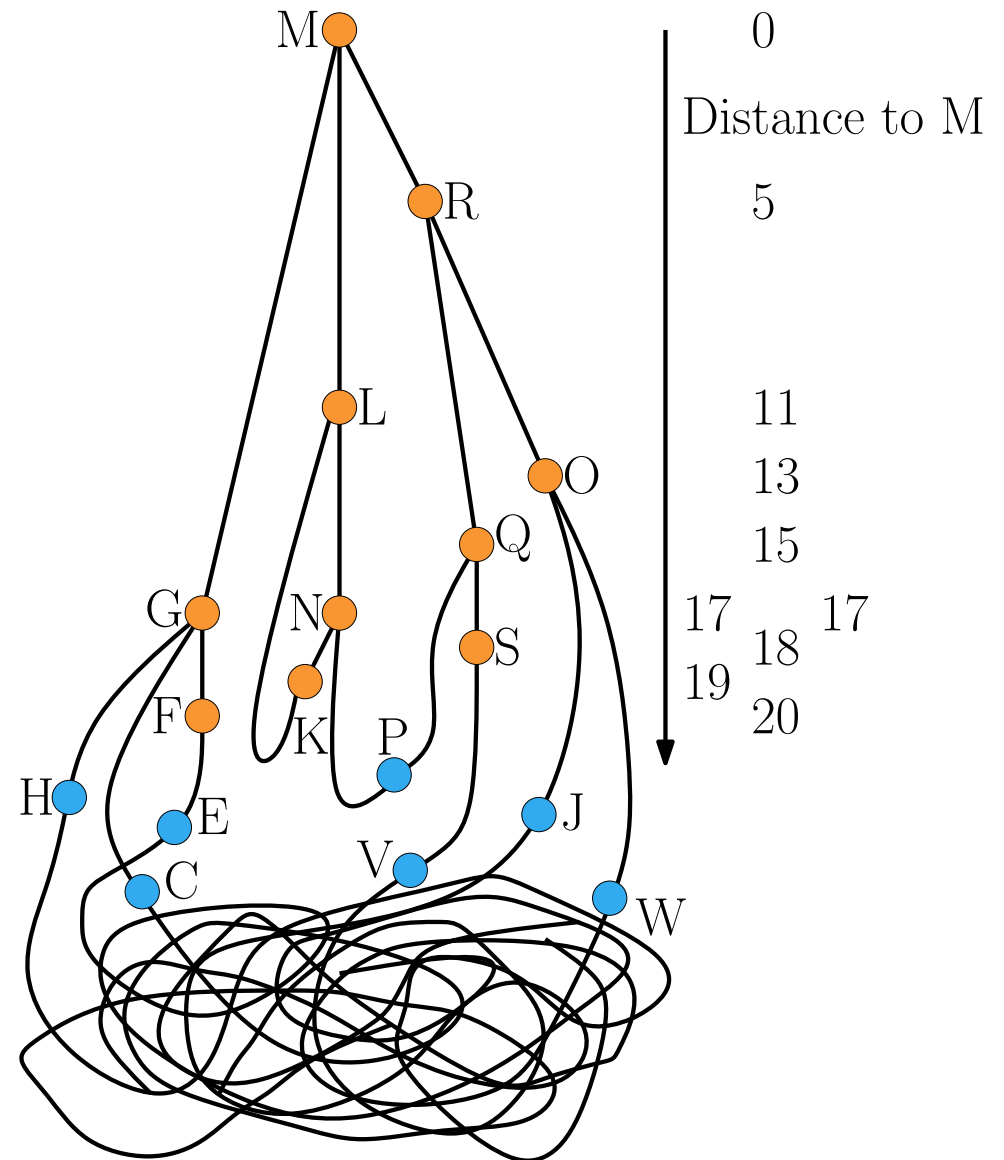
Allgemeine nichtnegative Gewichte

Lösung ohne Rechner:

Kanten → Fäden

Knoten → Knoten!

Am Startknoten anheben.



Korrekte Bindfäden

Betrachte beliebigen Knoten v

Mit Hängetiefe $d[v]$.

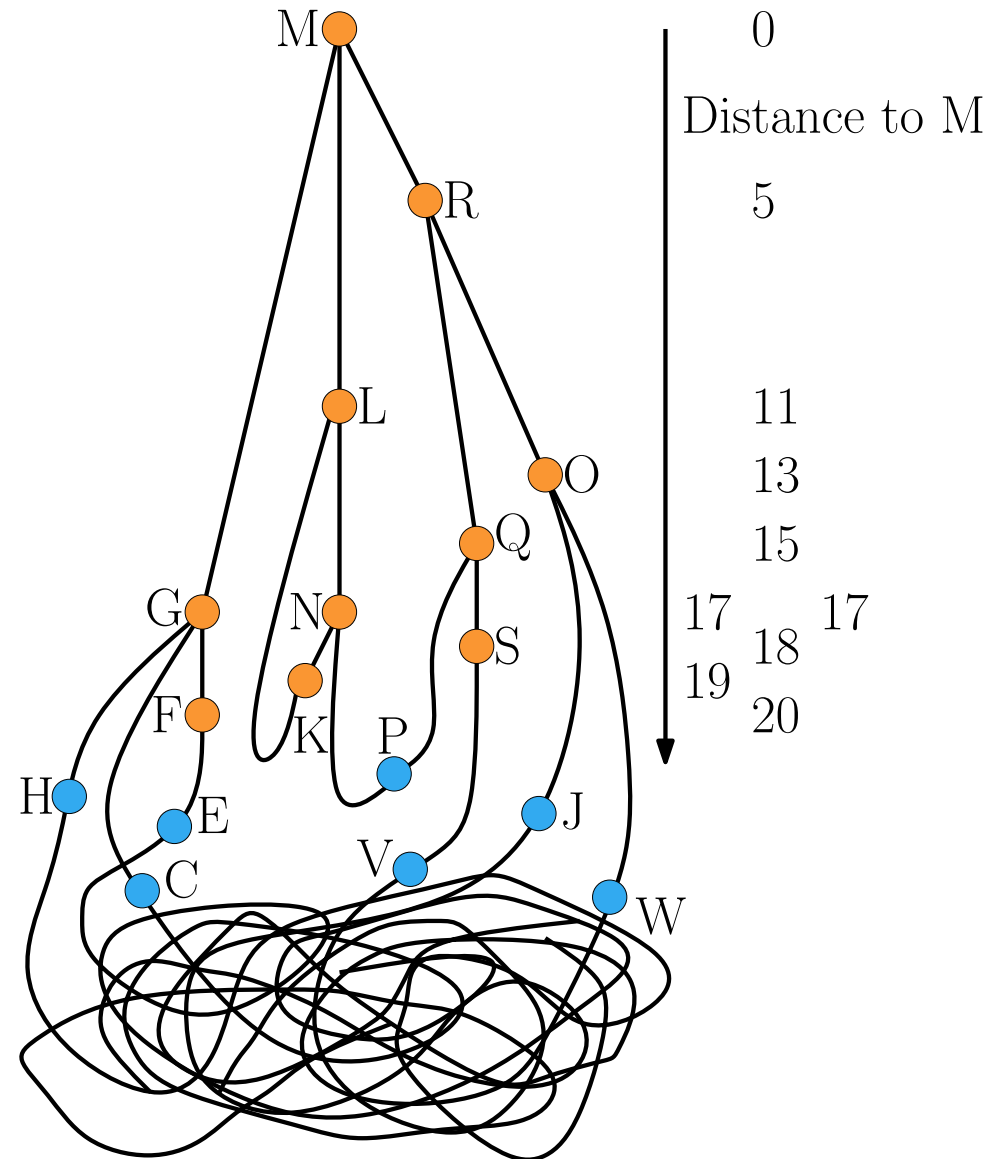
\exists **Pfad mit Hängetiefe:**

verfolge straffe Fäden

$\neg \exists$ **kürzerer Pfad:**

dann wäre einer seiner Fäden

zerrissen



Edsger Wybe Dijkstra



1972 ACM Turingpreis

THE: das erste Multitasking-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: $d[v] \geq \mu(v)$

□ $\text{PARENT}[v]$ = Vorgänger von v

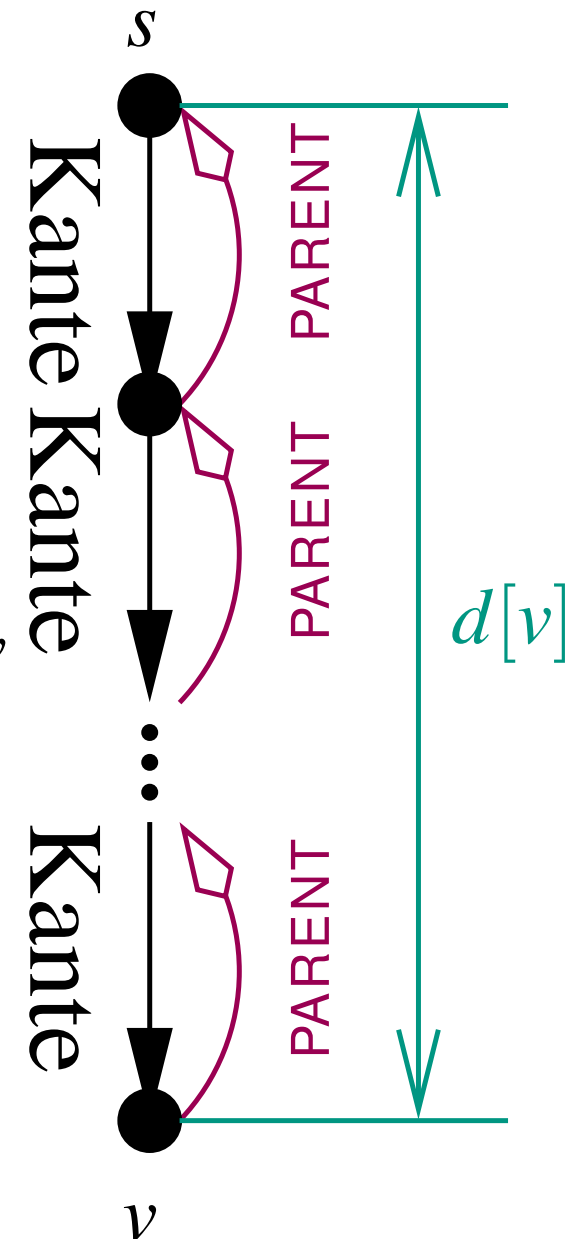
auf dem (vorläufigen) kürzesten Pfad von s nach v

Invariante: dieser Pfad bezeugt $d[v]$

Initialisierung:

$d[s] = 0$, $\text{PARENT}[s] = s$

$d[v] = \infty$, $\text{PARENT}[v] = \perp$



Kante (u, v) relaxieren

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

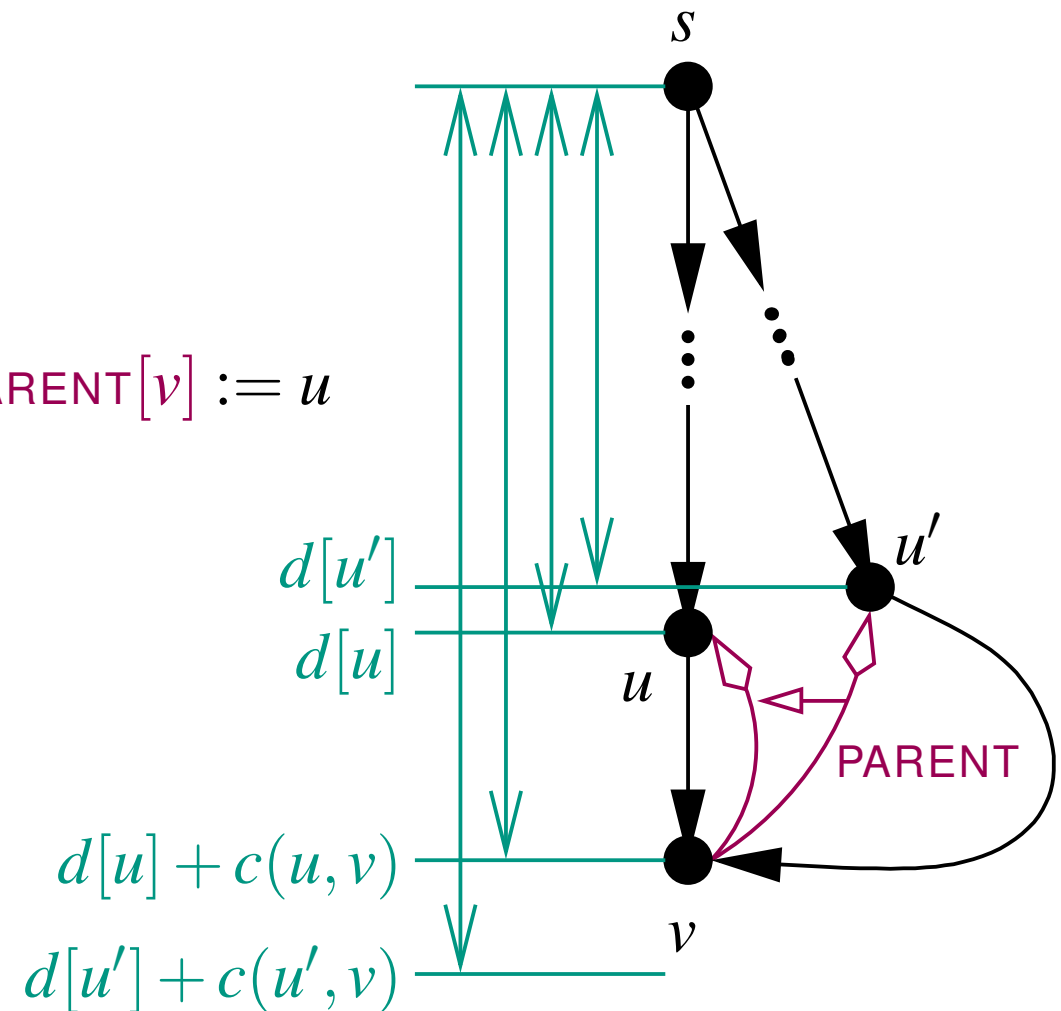
vielleicht $d[v] = \infty$

setze $d[v] := d[u] + c(u, v)$ und $\text{PARENT}[v] := u$

Invarianten bleiben erhalten!

Beobachtung:

$d[v]$ Kann sich mehrmals ändern!



Dijkstra's Algorithmus: Pseudocode

initialize d , PARENT

all nodes are non-scanned

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

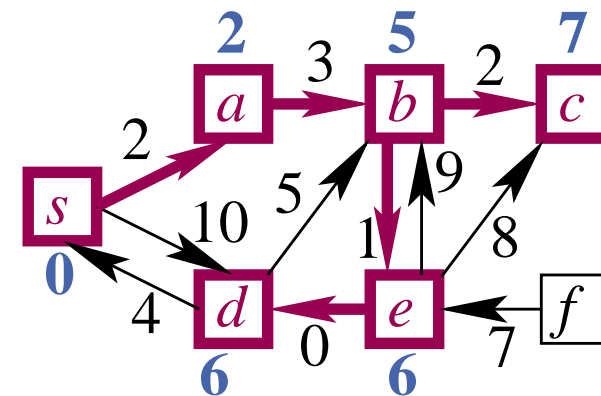
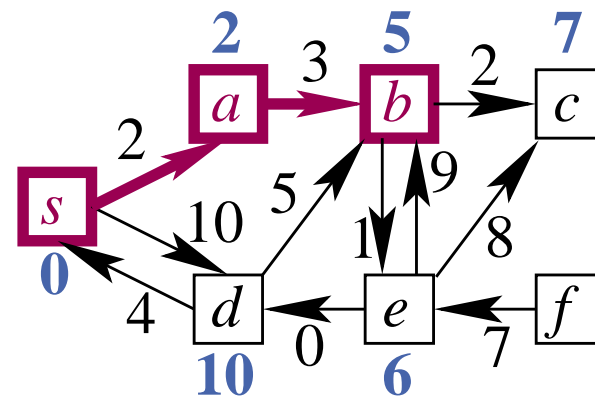
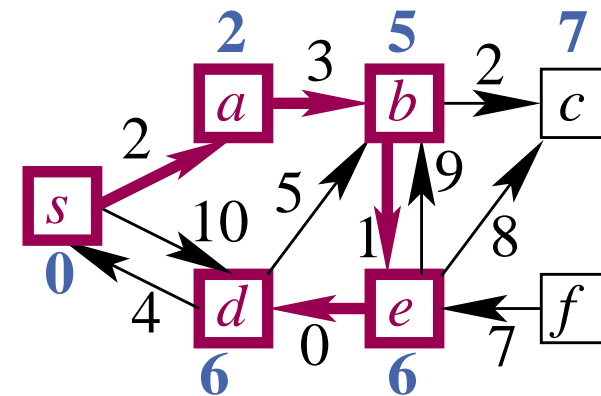
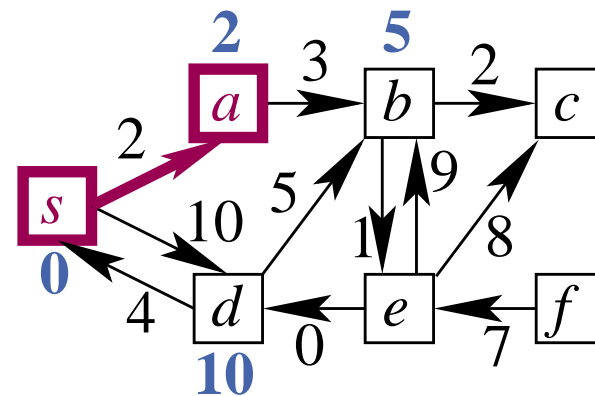
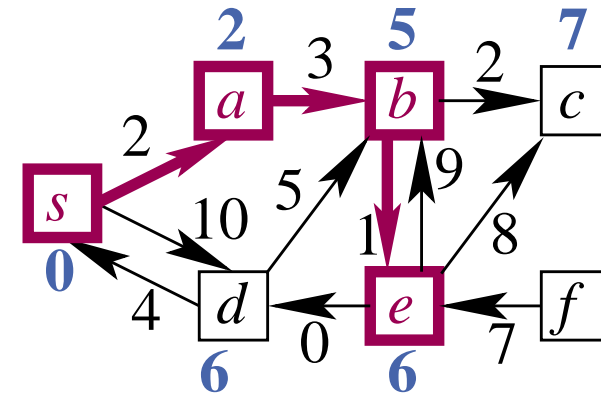
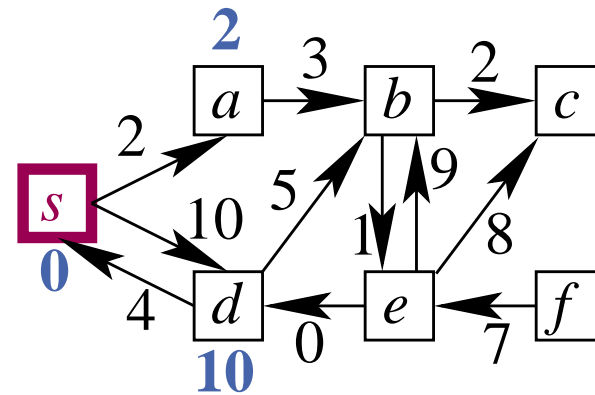
$u :=$ non-scanned node v with minimal $d[v]$

 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

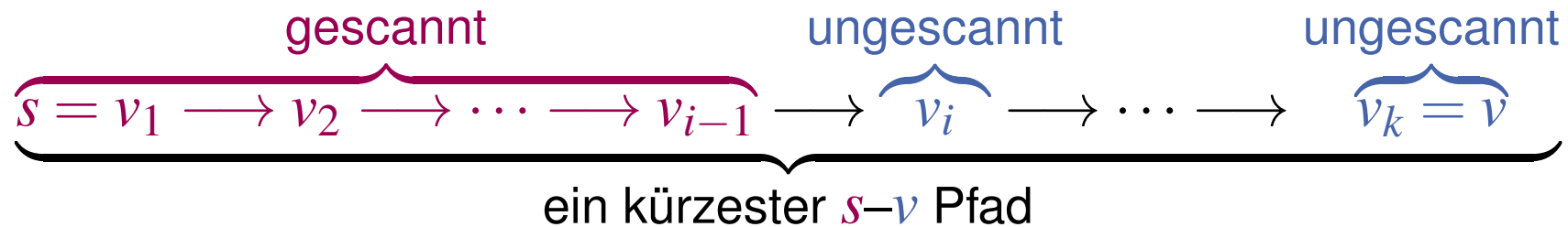
Annahme: alle Kosten nicht negativ!

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

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

v erreichbar $\implies v$ wird irgendwann gescannt

Annahme: v ist erreichbar aber wird nicht gescannt



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

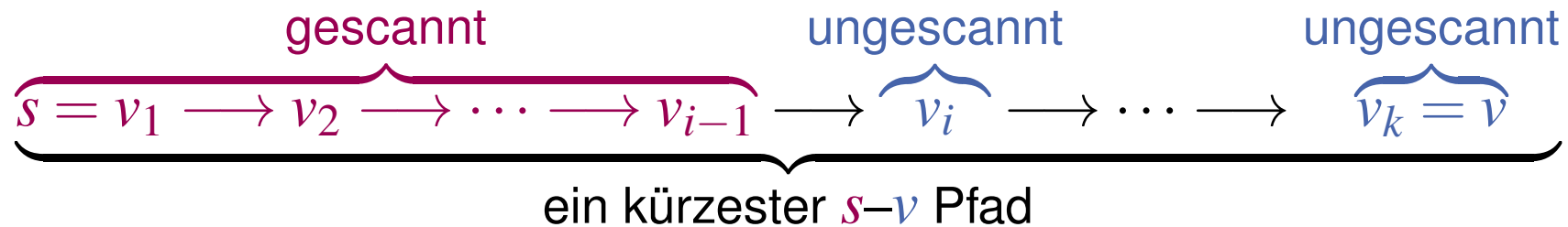
\implies Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert

$\implies d[v_i] < \infty$

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

v erreichbar $\implies v$ wird irgendwann gescannt

Annahme: v ist erreichbar aber wird nicht gescannt



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

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

$\implies d[v_i] < \infty$

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.

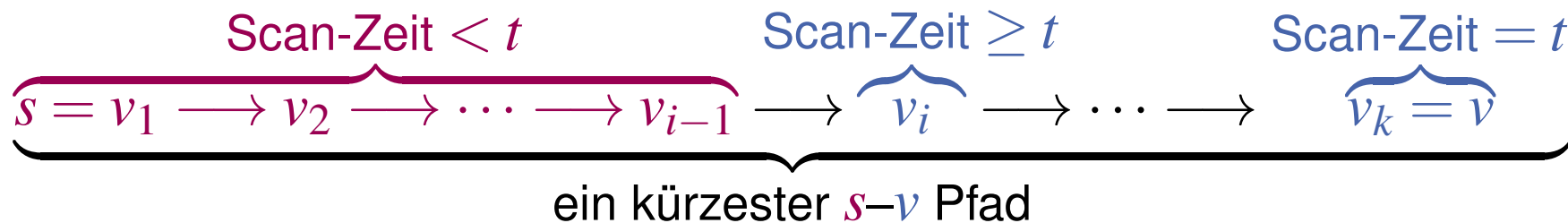
□

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

Annahme: v gescannt und $\mu(v) < d[v]$

OBdA: v ist der **erste** gescannte Knoten mit $\mu(v) < d[v]$.

$t :=$ Scan-Zeit von v



Also gilt zur Zeit t :

$$\mu(v_{i-1}) = d[v_{i-1}]$$

$v_{i-1} \rightarrow v_i$ wurde relaxiert

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

$\implies v_i$ wird vor v gescannt. **Widerspruch!**

Wieder: Spezialfall $i = 1$ unmöglich. □

Implementierung?

initialize d , PARENT

all nodes are non-scanned

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

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

relax all edges (u, v) out of u

u is scanned now

Wichtigste Operation: finde u

Prioritätsliste

Wir speichern **ungescannte erreichte Knoten** in
adressierbarer Prioritätsliste Q .

Schlüssel ist $d[v]$.

Knoten speichern handles.

oder gleich items

Implementierung \approx BFS mit PQ statt FIFO

Function Dijkstra($s : \text{NodeId}$) : $\text{NodeArray} \times \text{NodeArray}$
// returns (d, PARENT)

Initialisierung:

$d = \langle \infty, \dots, \infty \rangle : \text{NodeArray}$ **of** $\mathbb{R} \cup \{\infty\}$
// tentative distance from root

$\text{parent} = \langle \perp, \dots, \perp \rangle : \text{NodeArray}$ **of** NodeId

$\text{parent}[s] := s$ // self-loop signals root

$Q : \text{NodePQ}$ // unscanned reached nodes

$d[s] := 0; \quad Q.\text{insert}(s)$

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

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

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

$u := Q.deleteMin$

// scan u

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

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

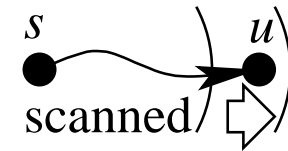
$d[v] := d[u] + c(e)$

parent[v] := u

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

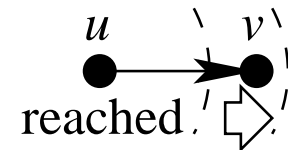
else $Q.insert(v)$

return (d , PARENT)

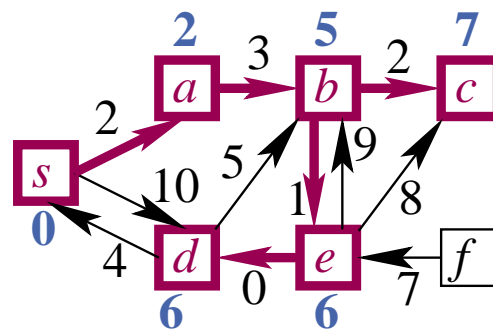
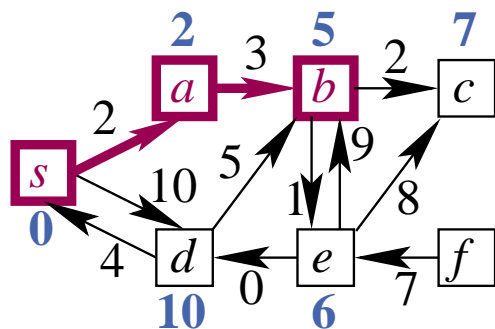
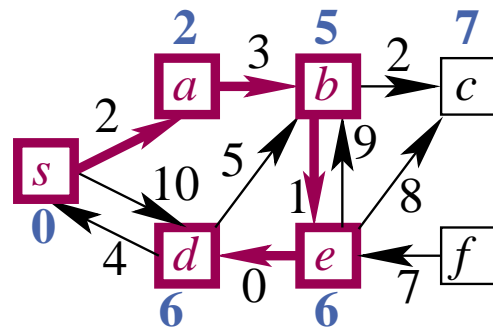
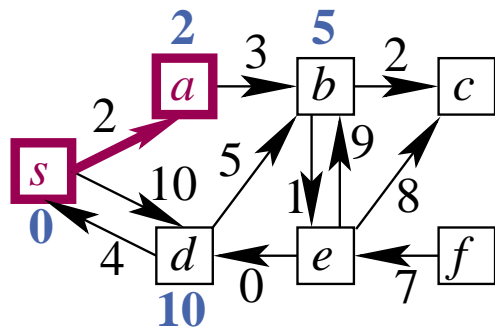
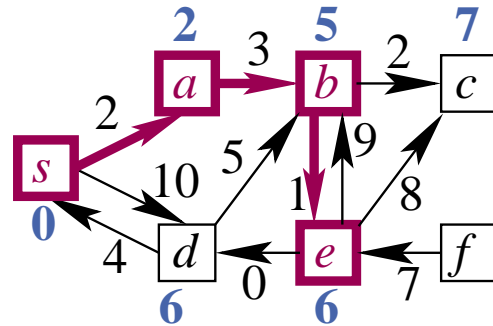
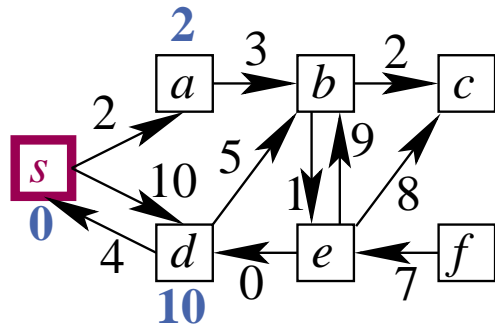


// relax

// update tree



Beispiel



Operation

INSERT(s)

DELETEMIN $\rightsquigarrow (s, 0)$

RELAX $s \xrightarrow{2} a$

RELAX $s \xrightarrow{10} d$

DELETEMIN $\rightsquigarrow (a, 2)$

RELAX $a \xrightarrow{3} b$

DELETEMIN $\rightsquigarrow (b, 5)$

RELAX $b \xrightarrow{2} c$

RELAX $b \xrightarrow{1} e$

DELETEMIN $\rightsquigarrow (e, 6)$

RELAX $e \xrightarrow{9} b$

RELAX $e \xrightarrow{8} c$

RELAX $e \xrightarrow{0} d$

DELETEMIN $\rightsquigarrow (d, 6)$

RELAX $d \xrightarrow{4} s$

RELAX $d \xrightarrow{5} b$

DELETEMIN $\rightsquigarrow (c, 7)$

Queue

$\langle (s, 0) \rangle$

$\langle \rangle$

$\langle (a, 2) \rangle$

$\langle (a, 2), (d, 10) \rangle$

$\langle (d, 10) \rangle$

$\langle (b, 5), (d, 10) \rangle$

$\langle (d, 10) \rangle$

$\langle (c, 7), (d, 10) \rangle$

$\langle (e, 6), (c, 7), (d, 10) \rangle$

$\langle (c, 7), (d, 10) \rangle$

$\langle (c, 7), (d, 10) \rangle$

$\langle (c, 7), (d, 10) \rangle$

$\langle (d, 6), (c, 7) \rangle$

$\langle (c, 7) \rangle$

$\langle (c, 7) \rangle$

$\langle (c, 7) \rangle$

$\langle \rangle$

Laufzeit

Function Dijkstra($s : \text{NodeId}$) : $\text{NodeArray} \times \text{NodeArray}$

Initialisierung:

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

$\text{parent} = \langle \perp, \dots, \perp \rangle : \text{NodeArray}$ **of** NodeId // $O(n)$

$\text{parent}[s] := s$

$Q : \text{NodePQ}$ // unscanned reached nodes, $O(n)$

$d[s] := 0; \quad Q.\text{insert}(s)$

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

$d = \{\infty, \dots, \infty\}$; PARENT[s] := s ; $d[s] := 0$; $Q.insert(s)$ // $O(n)$

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

$u := Q.deleteMin$ // $\leq n \times$

foreach edge $e = (u, v) \in E$ **do** // $\leq m \times$

if $d[u] + c(e) < d[v]$ **then** // $\leq m \times$

$d[v] := d[u] + c(e)$ // $\leq m \times$

parent[v] := u // $\leq m \times$

if $v \in Q$ **then** $Q.decreaseKey(v)$ // $\leq m \times$

else $Q.insert(v)$ // $\leq n \times$

return (d , PARENT)

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

$d = \{\infty, \dots, \infty\}$; PARENT[s] := s ; $d[s] := 0$; $Q.insert(s)$ // $O(n)$

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

$u := Q.deleteMin$ // $\leq n \times$

foreach edge $e = (u, v) \in E$ **do** // $\leq m \times$

if $d[u] + c(e) < d[v]$ **then** // $\leq m \times$

$d[v] := d[u] + c(e)$ // $\leq m \times$

parent[v] := u // $\leq m \times$

if $v \in Q$ **then** $Q.decreaseKey(v)$ // $\leq m \times$

else $Q.insert(v)$ // $\leq n \times$

return (d , PARENT)

Insgesamt

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{DECREASEKEY}}(n) + n \cdot (T_{\text{DELETETMIN}}(n) + T_{\text{INSERT}}(n)))$$

Laufzeit

Dijkstra's ursprüngliche Implementierung: „naiv“

□ insert $O(1)$

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

□ decreaseKey $O(1)$

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

□ deleteMin $O(n)$

d komplett durchsuchen

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{DECREASEKEY}}(n) + n \cdot (T_{\text{DELETETMIN}}(n) + T_{\text{INSERT}}(n)))$$

$$\begin{aligned} T_{\text{DIJKSTRA59}} &= O(m \cdot 1 + n \cdot (n + 1)) \\ &= O(m + n^2) \end{aligned}$$

Laufzeit

Bessere Implementierung mit **Binary-Heapprioritätslisten**:

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

$$\begin{aligned} T_{\text{Dijkstra}} &= O(m \cdot T_{\text{DECREASEKEY}}(n) + n \cdot (T_{\text{DELETETMIN}}(n) + T_{\text{INSERT}}(n))) \\ T_{\text{DIJKSTRABHEAP}} &= O(m \cdot \log n + n \cdot (\log n + \log n)) \\ &= O((m + n) \log n) \end{aligned}$$

Laufzeit

(Noch) besser mit **Fibonacci-Heapprioritätslisten**:

- ☐ insert $O(1)$
- ☐ decreaseKey $O(1)$ (amortisiert)
- ☐ deleteMin **$O(\log n)$** (amortisiert)

$$\begin{aligned} T_{\text{Dijkstra}} &= O(m \cdot T_{\text{DECREASEKEY}}(n) + n \cdot (T_{\text{DELETETMIN}}(n) + T_{\text{INSERT}}(n))) \\ T_{\text{DIJKSTRAFIB}} &= O(m \cdot 1 + n \cdot (\log n + 1)) \\ &= \mathbf{O(m + n \log n)} \end{aligned}$$

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

10.4 Analyse im Mittel

Modell: Kantengewichte sind „zufällig“ auf die Kanten verteilt

Dann gilt

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

Beweis: In Algorithmen II

10.5 Monotone ganzzahlige Prioritätslisten

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

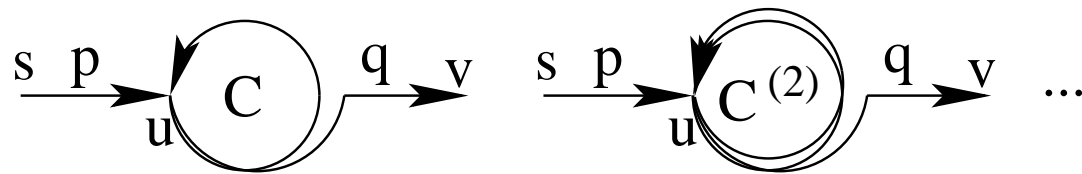
Das kann man ausnutzen. \rightsquigarrow schnellere Algorithmen
u.U. bis herunter zu $O(m + n)$.

Details: in Algorithmen II

10.6 Negative Kosten

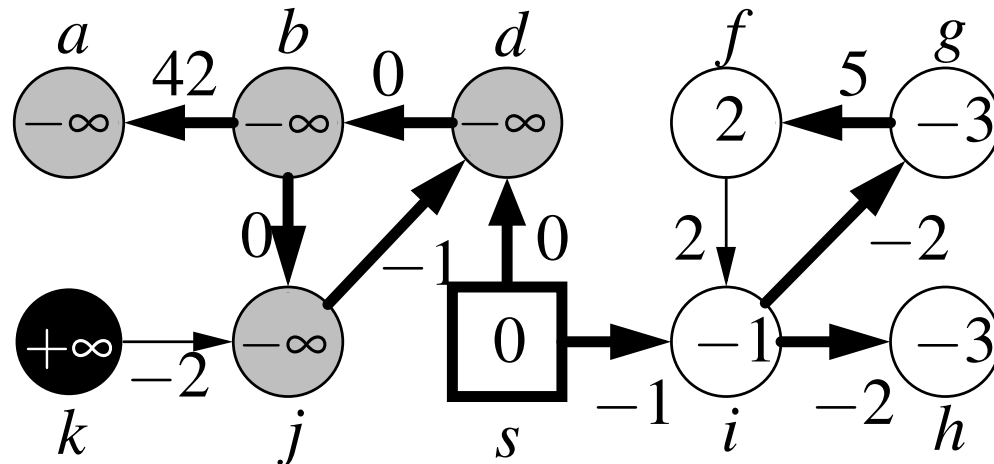
Was machen wir wenn es Kanten mit negativen Kosten gibt?

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)

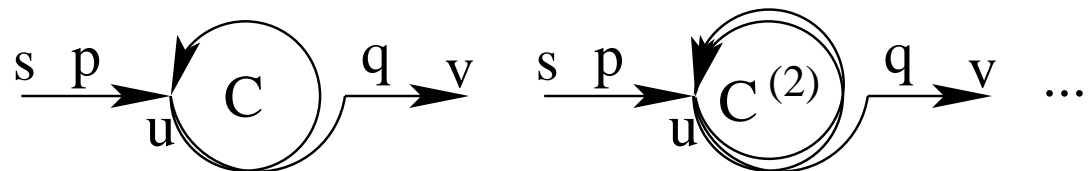
Lemma: \exists kürzesten $s-v$ -Pfad $P \implies P$ ist OBdA **einfach** (eng. simple)

Beweisidee:

Fall: v über negativen Kreis erreichbar?:

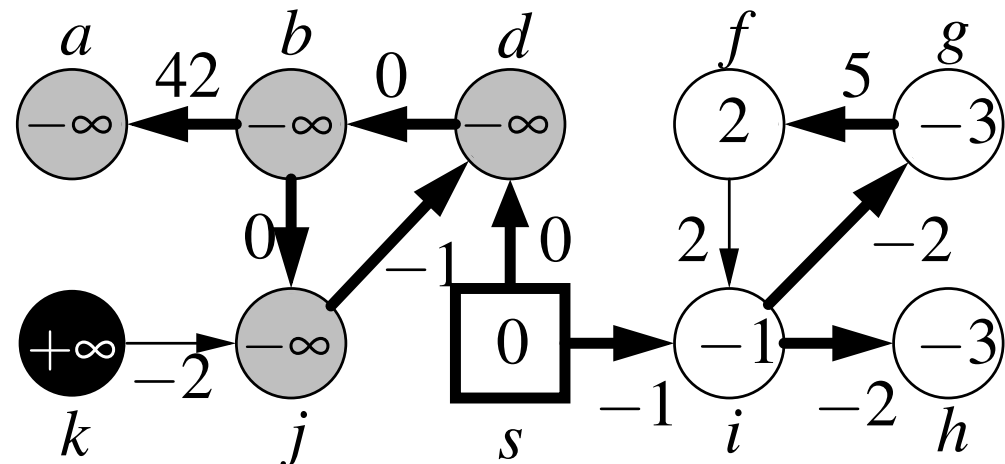
$\neg \exists$ kürzesten $s-v$ -Pfad

(sondern beliebig
viele immer kürzere)



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

Alle Kreise streichen \rightsquigarrow einfacher, nicht längerer Pfad. ■



Mehr Basiskonzepte

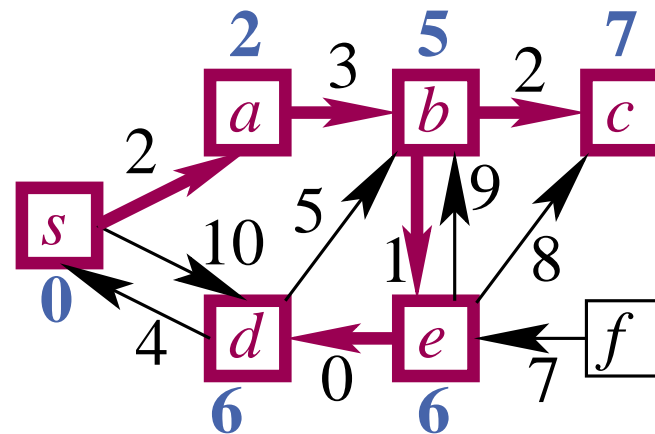
Übung, Zeige:

Teilpfade kürzester Pfade sind selbst kürzeste Pfade

$$a-b-c-d \rightsquigarrow a-b, b-c, c-d, a-b-c, b-c-d$$

Übung: Kürzeste-Wege-Baum

Alle 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
 $p = \langle e_1, e_2, \dots, e_k \rangle = \langle s, v_1, v_2, \dots, v_k \rangle$ ein kürzester Weg.
Dann gilt anschließend $d[v_k] = \mu(v_k)$

Beweisskizze: (Eigentlich Induktion über k)

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

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

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

...

$d[v_k] = \mu(v_k)$ nach Zeitpunkt t_k

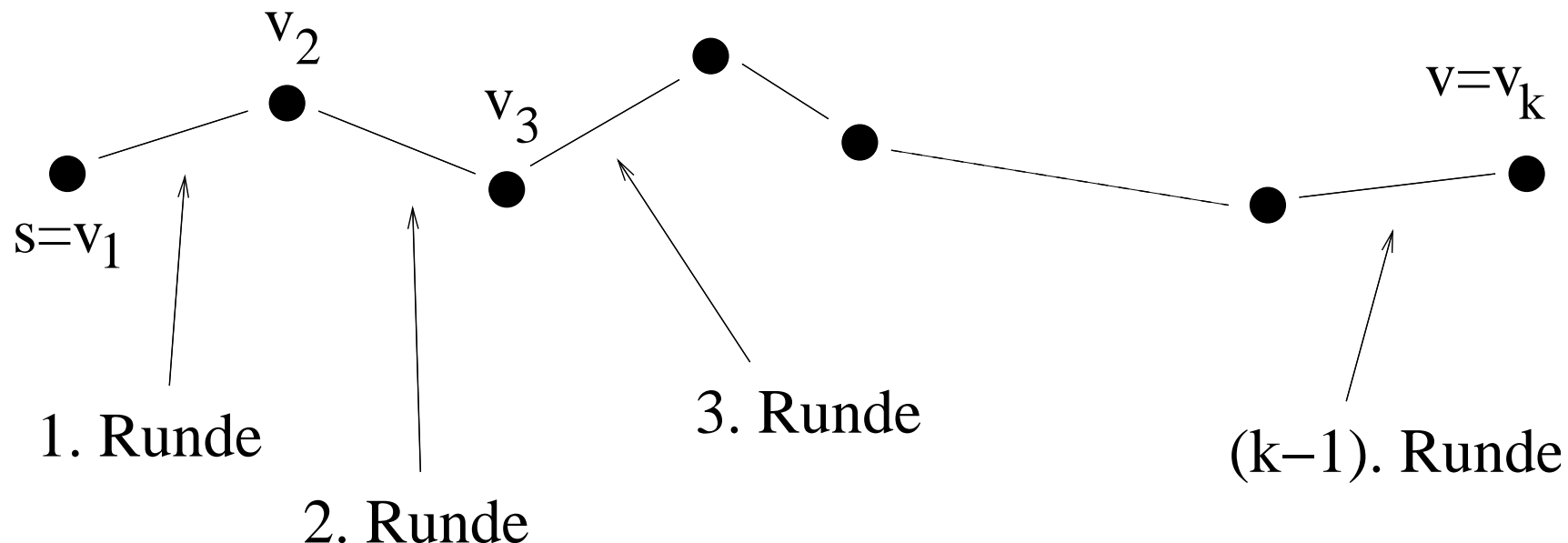


Algorithmen Brutal – Bellman-Ford-Algorithmus für beliebige Kantengewichte

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

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

Jeder kürzeste Pfad ist eine Teilfolge dieser Relaxierungen!



Negative Kreise Finden

Nach Ausführung von Bellman-Ford:

\forall negativen Kreise C :

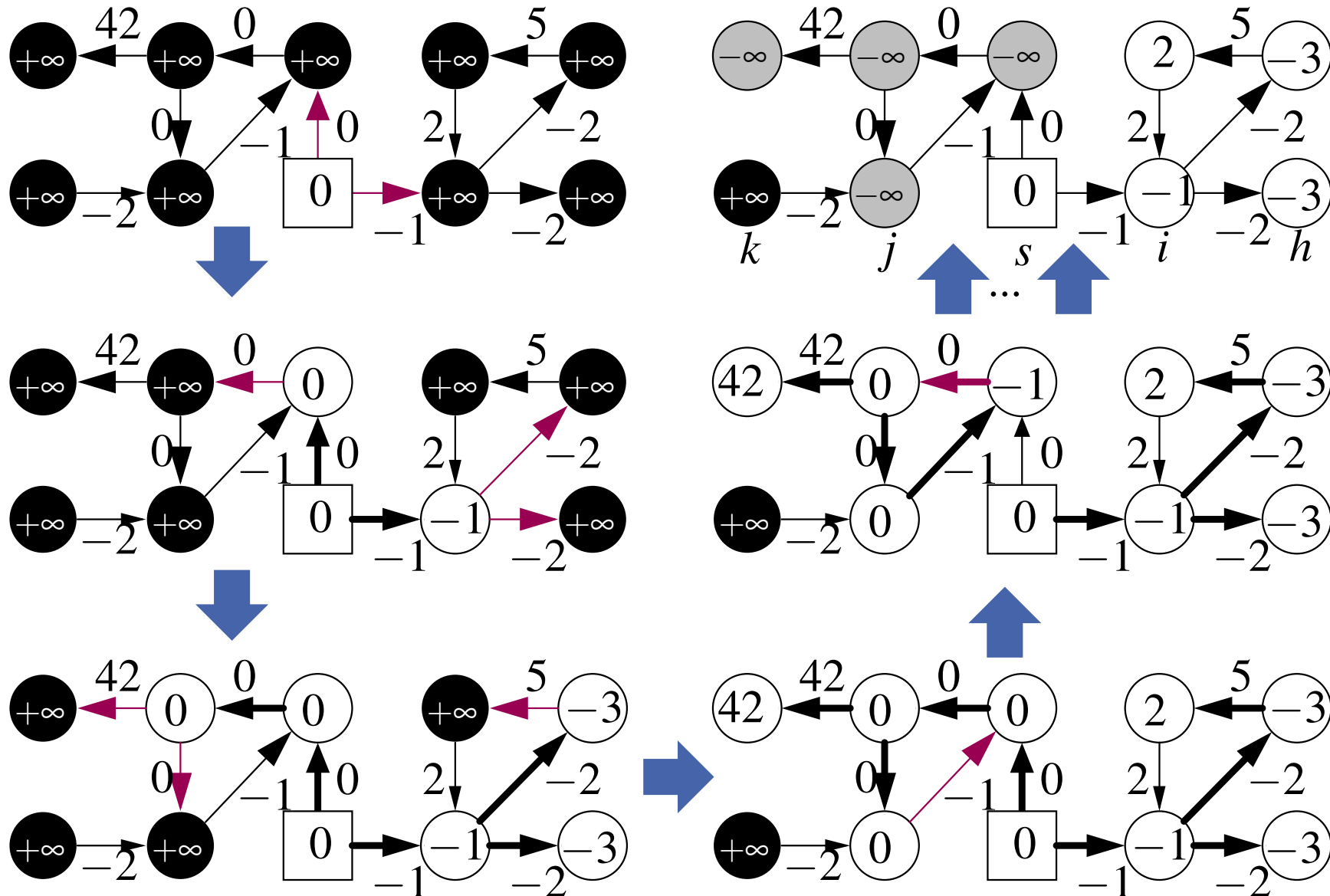
$\exists (u, v) \in C :$

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

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 \implies keine negativen Kreise !

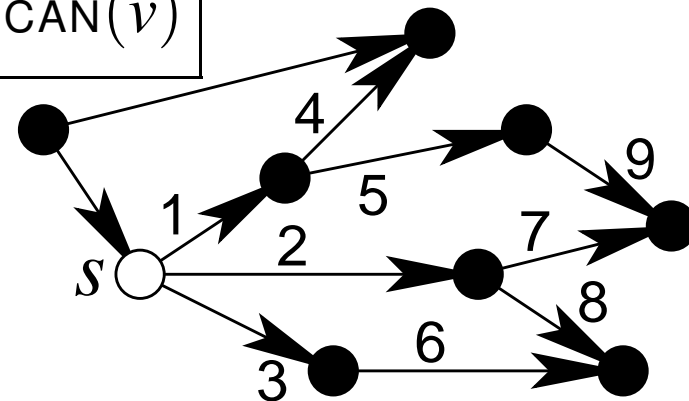
Für jeden (kürzesten) Pfad $\langle v_1, \dots, v_n \rangle$:

Die Kanten sind aufsteigend bzgl. jeder **topologischen Sortierung** !

initialize d , PARENT

foreach $v \in V$ in **topological order** **do** SCAN(v)

Laufzeit: $O(m + n)$



10.7 Von Überall nach Überall

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

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

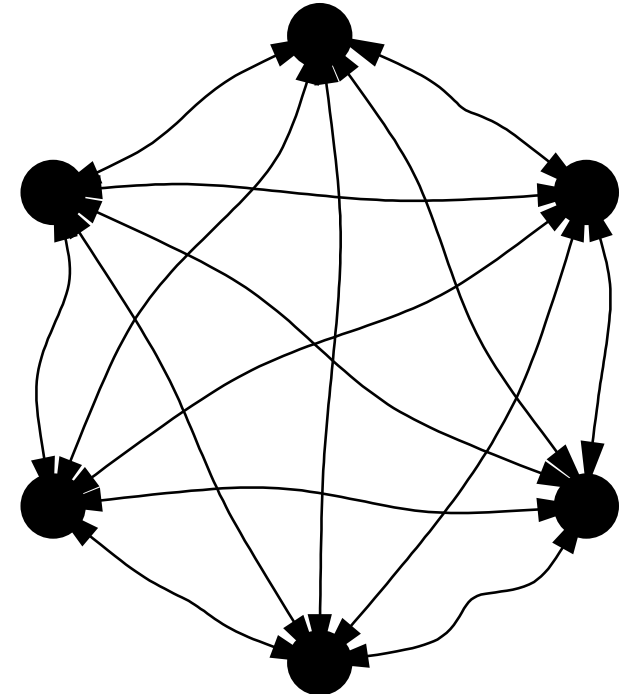
($n \times$ Dijkstra)

beliebige Kantengewichte: Zeit $O(n^2 m)$.

($n \times$ Bellman-Ford)

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

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



Kürzeste Wege: Zusammenfassung

- ☐ Einfache, effiziente Algorithmen für **nichtnegative** Kantengewichte und **azyklische** Graphen
- ☐ Optimale Lösungen bei nicht (ganz) trivialen Korrektheitsbeweisen
- ☐ Prioritätslisten sind wichtige Datenstruktur

Mehr zu kürzesten Wege

Viele Arbeiten zu besseren Prioritätslisten $\rightsquigarrow O(m + n \log \log n)$

[Thorup 2004]

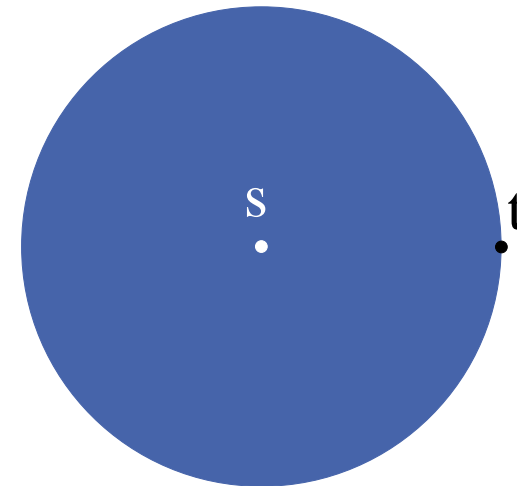
Verallgemeinerungen

- ☐ Mehrere Zielfunktionen abwägen
- ☐ Mehrere Ziele in beliebiger Reihenfolge anfahren siehe auch
Optimierungskapitel
- ☐ Mehrere disjunkte Wege

Fast alles schwierig (NP-hart)

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

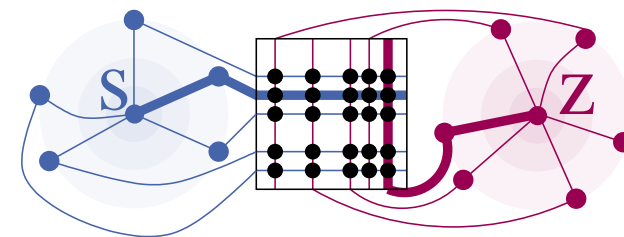
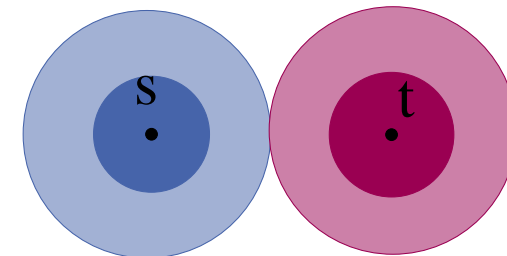
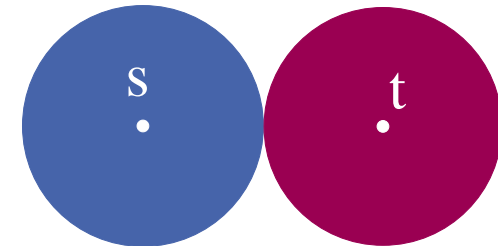
mehr in Algorithmen II, Algorithm Engineering

☐ Vorwärts + Rückwärtssuche

☐ Zielgerichtete Suche

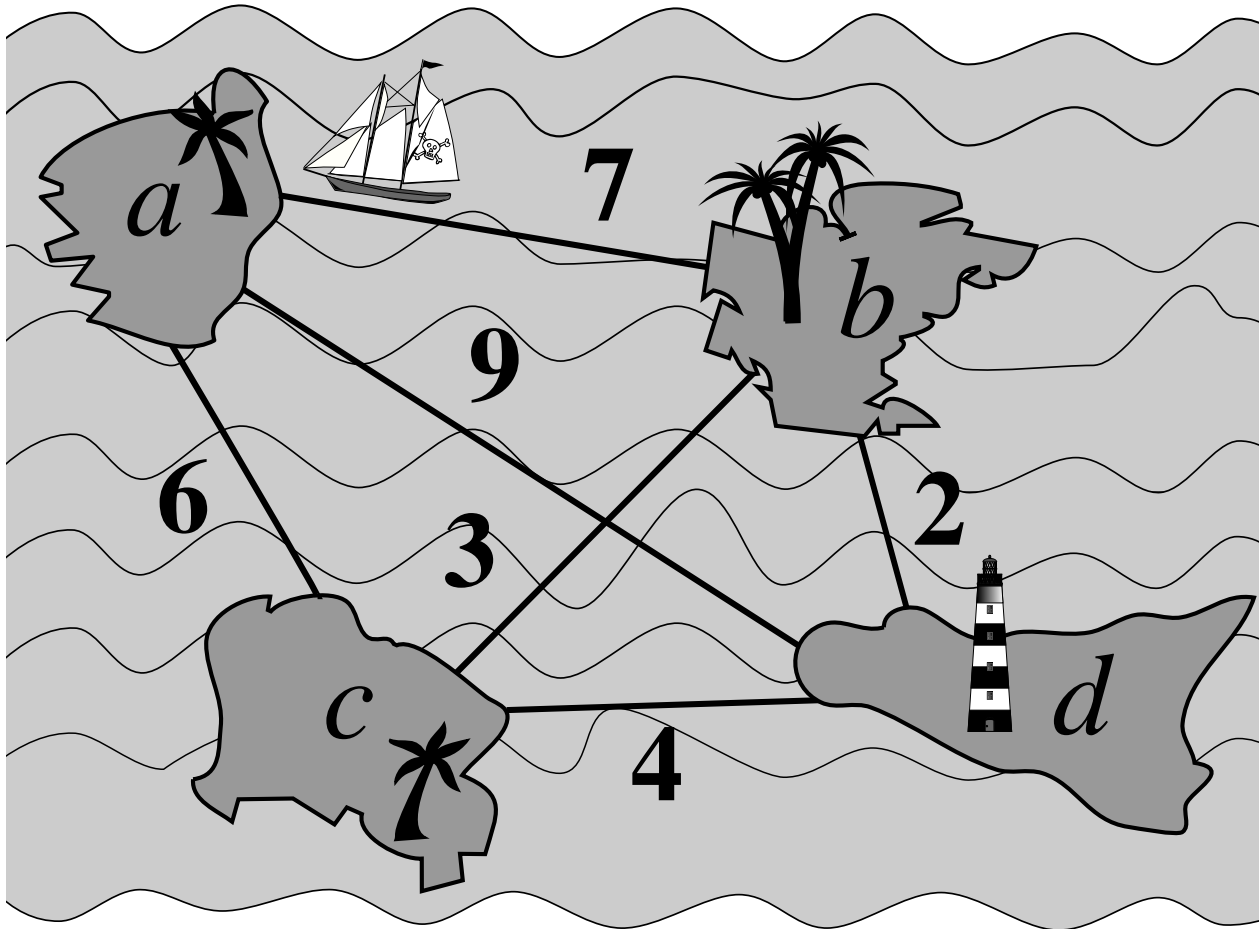
☐ Hierarchien ausnutzen

☐ Teilabschnitte tabellieren



Meist zentrale Idee: **Vorberechnung** amortisiert über viele Anfragen

11 Minimale Spannbäume



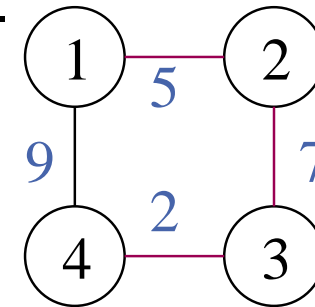
Minimale Spannbäume (MSTs)

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

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

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

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

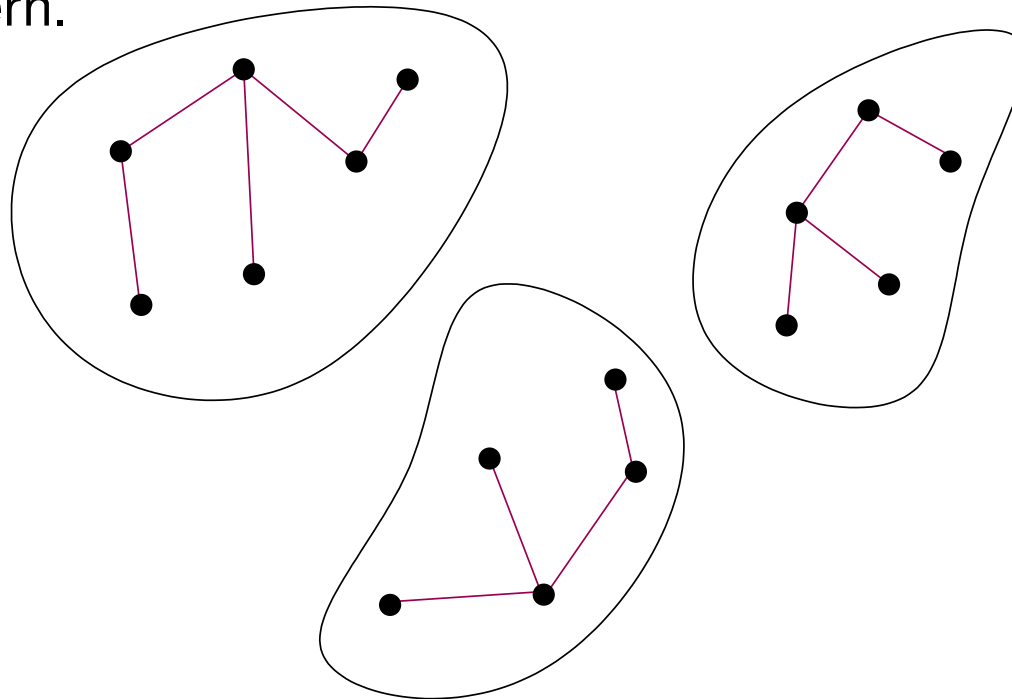


Finde Baum (V, T) mit **minimalem** Gewicht $\sum_{e \in T} c(e)$ der alle Knoten verbindet.

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

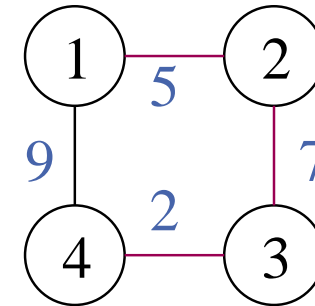
- Netzwerk-Entwurf

- 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

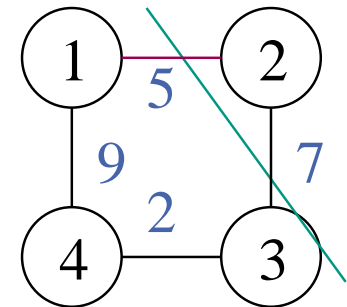
Die Schnitteigenschaft (Cut Property)

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 e in C
 kann in einem MST verwendet werden.

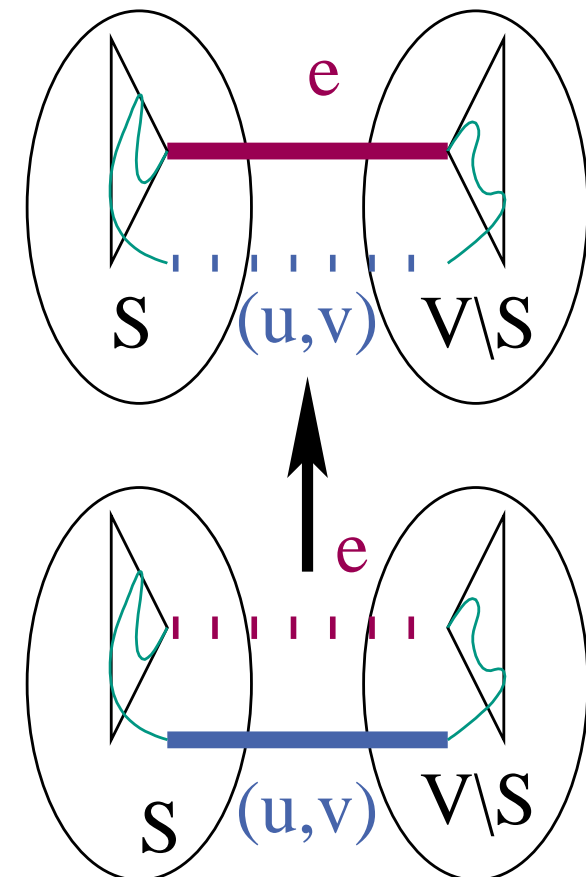
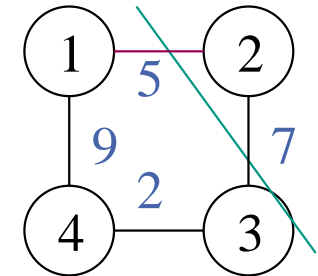
Beweis: Betrachte MST T' .

Fall $e \in T'$: Beweis fertig.

Sonst: $T' \cup \{e\}$ enthält **Kreis** K .

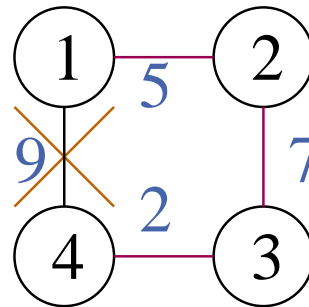
Betrachte eine Kante $\{u, v\} \in C \cap K \neq e$.

Dann ist $T = T' \setminus \{\{u, v\}\} \cup \{e\}$ ein Spannbaum
 der nicht schwerer ist.



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

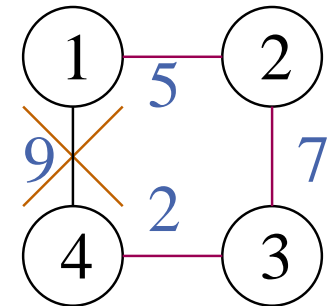
Beweis.

Angenommen MST T' benutzt die schwerste Kante e' auf Kreis C .

Wähle $e \in C$ mit $e \notin T'$.

Es gilt $c(e) \leq c(e')$.

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



□

11.2 Der Jarník-Prim Algorithmus

[Jarník 1930, Prim 1957]

Idee: Lasse einen Baum wachsen

$T := \emptyset$

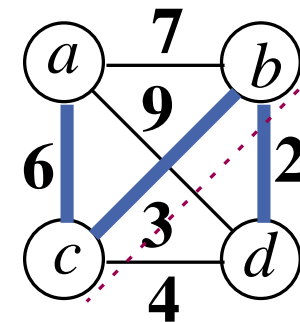
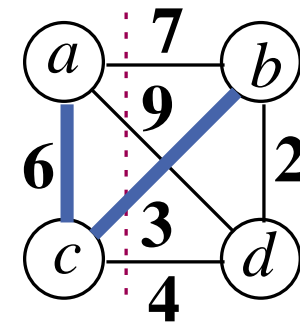
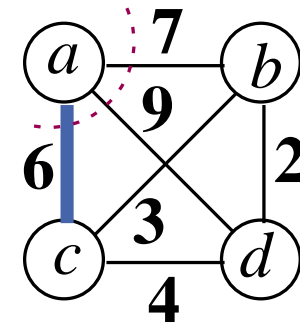
$S := \{s\}$ for arbitrary start node s

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 $s \in V$

$d = \{\infty, \dots, \infty\}$; PARENT[s] := s ; $d[s] := 0$; $Q.insert(s)$

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

$u := Q.deleteMin$

$d[u] := 0$

// scan u

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

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

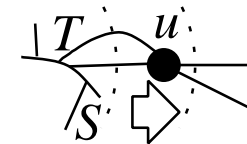
$d[v] := c(e)$

parent[v] := u

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

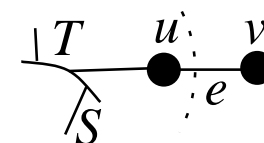
else $Q.insert(v)$

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



// relax

// update tree



Analyse

Praktisch identisch zu Dijkstra

□ $O(m + n)$ Zeit ausserhalb der PQ

□ n DELETETMIN (Zeit $O(n \log n)$)

□ $O(m)$ DECREASEKEY

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

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

Wichtigster Unterschied: monotone PQs reichen nicht

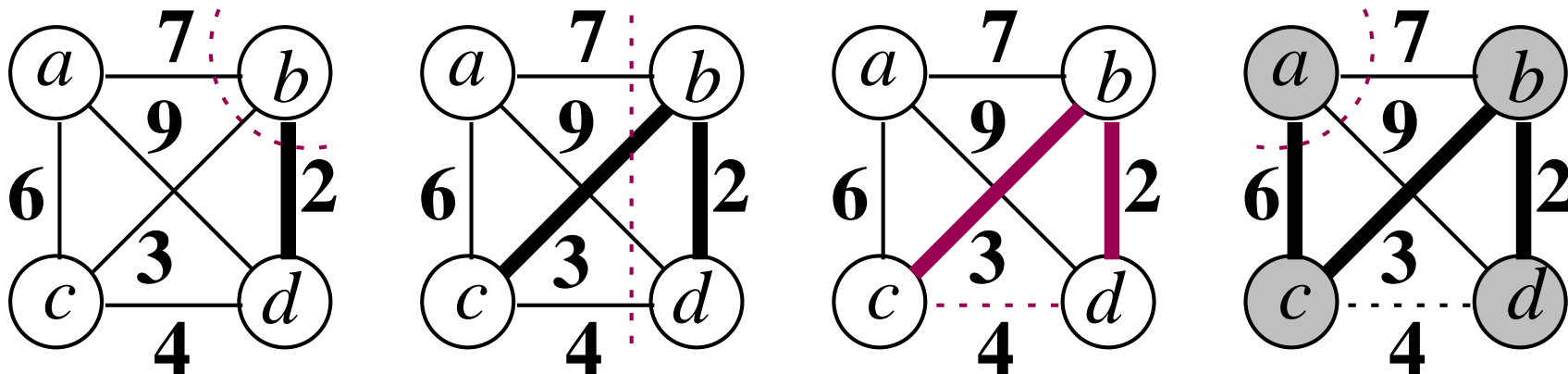
Warum?

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

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

$\implies (u, v) \in \text{MST}$

Sonst: benutze Kreiseigenschaft

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

$\implies (u, v) \notin \text{MST}$

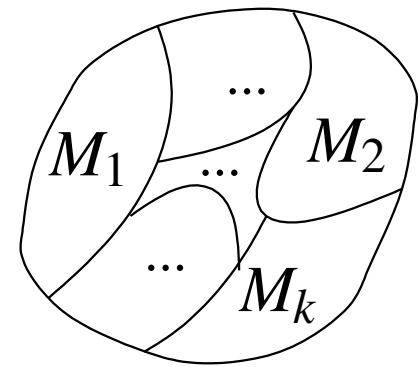


11.4 Union-Find Datenstruktur

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

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

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



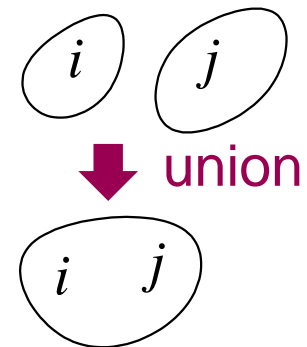
Class UnionFind($n : \mathbb{N}$)

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

join the blocks containing i and j to a single block.

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

return a unique identifier for the block containing i .



Union-Find Datenstruktur – Erste Version

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

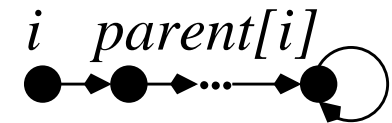
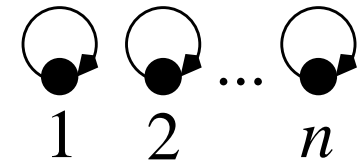
parent = $\langle 1, 2, \dots, n \rangle$: **Array** $[1..n]$ **of** $1..n$

invariant parent-refs lead to unique **Partition-Reps**

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

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

else return FIND(PARENT[i])

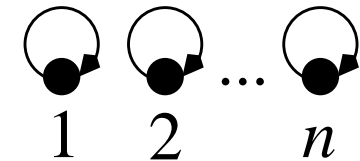


Union-Find Datenstruktur – Erste Version

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

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

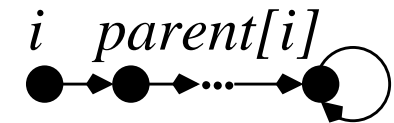
invariant parent-refs lead to unique **Partition-Reps**



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

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

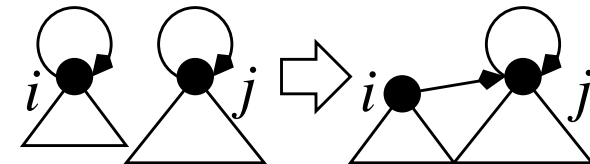
else return FIND(PARENT[i])



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

assert i and j are representatives of different blocks

parent[i] := j



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

if FIND(i) \neq FIND(j) **then** link(FIND(i), FIND(j))

Union-Find Datenstruktur – Erste Version

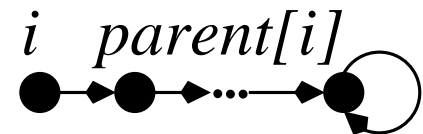
Analyse:

+: UNION braucht konstante Zeit

—: FIND braucht Zeit $\Theta(n)$ im schlechtesten Fall !

zu langsam.

Idee: FIND-Pfade kurz halten



Pfadkompression

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

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

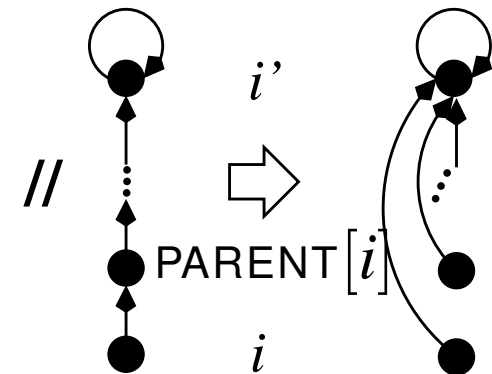
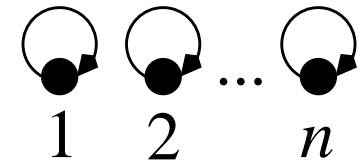
Function find($i : 1..n$) : $1..n$

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

else $i' := \text{find}(\text{parent}[i])$

parent[i] := i'

return i'



Union by Rank

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

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

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

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

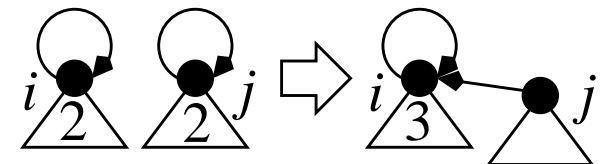
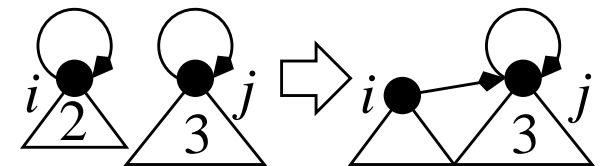
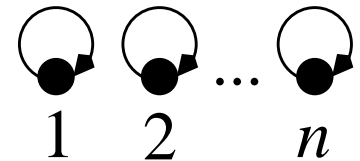
assert i and j are representatives of different blocks

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

else

parent[j] := i

if rank[i] = RANK[j] **then** rank[i] ++

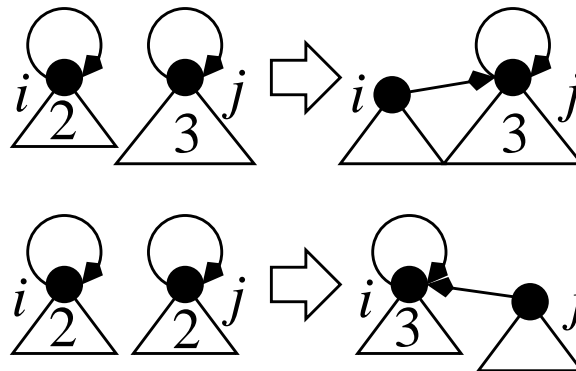


Analyse – nur Union-by-rank

invariant Der Pfad zum Repr. x hat Länge höchstens $\text{RANK}[x]$

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

Korollar: FIND braucht Zeit $O(\log n)$



Analyse – nur Pfadkompression

Satz: FIND braucht Zeit $O(\log n)$ (amortisiert)

Beweis: im Buch

Analyse – Pfadkompression + Union-by-rank

Satz: $m \times$ find + $n \times$ link brauchen Zeit $O(m\alpha_T(m, n))$ mit

$$\alpha_T(m, n) = \min \{i \geq 1 : A(i, \lceil m/n \rceil) \geq \log n\},$$

und

$$A(1, j) = 2^j \quad \text{for } j \geq 1,$$

$$A(i, 1) = A(i-1, 2) \quad \text{for } i \geq 2,$$

$$A(i, j) = A(i-1, A(i, j-1)) \quad \text{for } i \geq 2 \text{ and } j \geq 2.$$

Beweis: [Tarjan 1975, Seidel Sharir 2005]

A ist die Ackermannfunktion und α_T die inverse Ackermannfunktion.

$\alpha_T(m, n) = \omega(1)$ aber ≤ 4 für alle physikalisch denkbaren n, m .

Ackermannfunktion – Beispiele

$$A(1, j) = 2^j \quad \text{for } j \geq 1,$$

$$A(i, 1) = A(i - 1, 2) \quad \text{for } i \geq 2,$$

$$A(i, j) = A(i - 1, A(i, j - 1)) \quad \text{for } i \geq 2 \text{ and } j \geq 2.$$

$$A(2, 1) = A(1, 2) = 2^2$$

$$A(2, 2) = A(1, A(2, 1)) = 2^{2^2}$$

$$A(2, 3) = A(1, A(2, 2)) = 2^{2^{2^2}}$$

$$A(2, 4) = A(1, A(2, 3)) = 2^{2^{2^{2^2}}}$$

$$A(3, 1) = A(2, 2) = 2^{2^2}$$

$$A(3, 2) = A(2, A(3, 1)) = A(2, 16) = ???$$

$$A(4, 1) = A(3, 2) = ???$$

Kruskal mit Union-Find

Sei $V = 1..n$

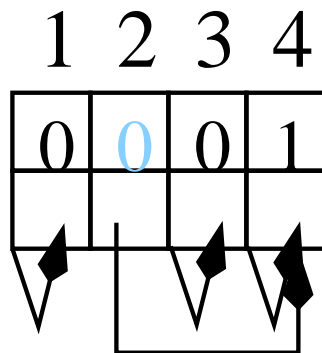
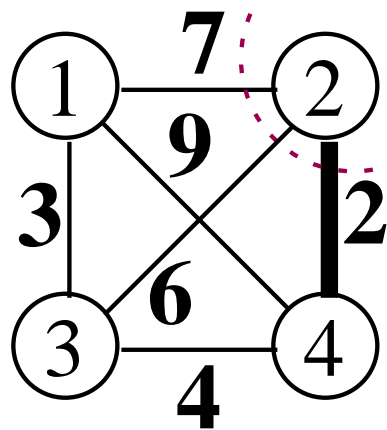
```
Tc : UnionFind(n)           // encodes components of forest T
foreach  $(u, v) \in E$  in ascending order of weight do           // sort
    if Tc.FIND( $u$ )  $\neq$  Tc.FIND( $v$ ) then
        output  $\{u, v\}$ 
        Tc.union( $u, v$ )           // LINK reicht auch
```

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

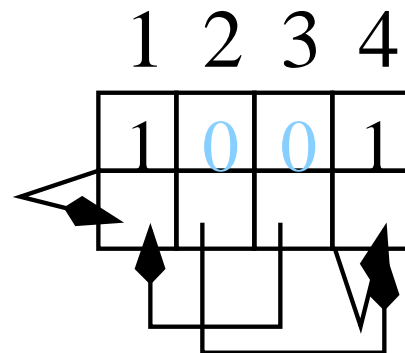
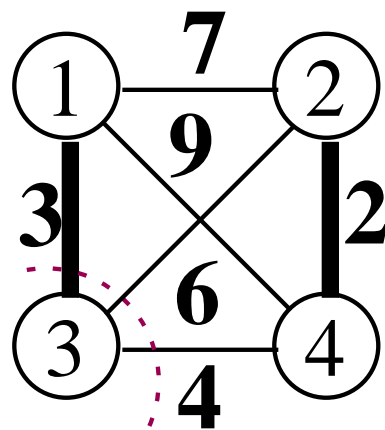
Graphrepräsentation: **Kantenliste**

Bäume im MSF \leftrightarrow Blöcke in Partition \rightarrow Wurzelbäume
aber mit **anderer Struktur** als die Bäume im MSF

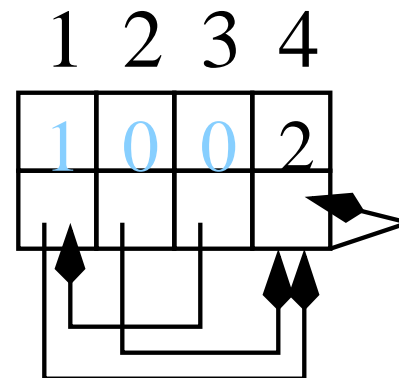
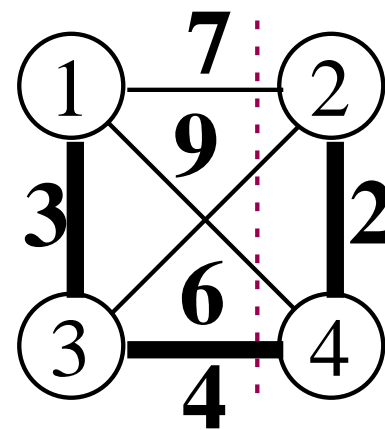
Beispiel



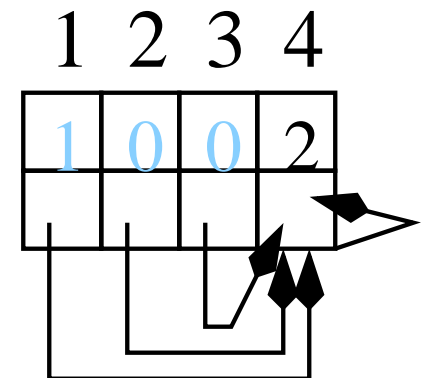
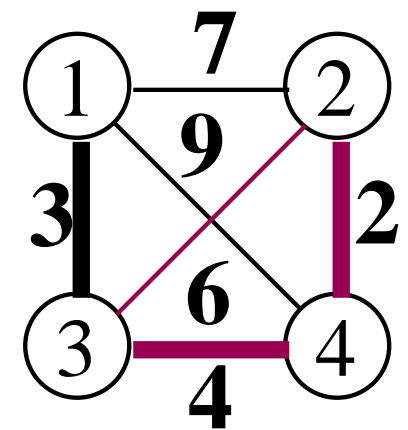
link



link



link



compress

Vergleich Jarník-Prim \leftrightarrow Kruskal

Pro Jarník-Prim

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

Pro Kruskal

- ☐ Gut für $m = O(n)$
- ☐ Braucht nur Kantenliste
- ☐ Profitiert von schnellen Sortierern (ganzzahlig, parallel, ...)
- ☐ Verfeinerungen auch gut für große m/n

Mehr MST-Algorithmen

- Zeit $O(m \log n)$ [Boruvka 1926]

Zutat vieler fortgeschrittener Algorithmen

- Erwartete Zeit $O(m)$ [Karger Klein Tarjan 1995],
parallelisierbar, externalisierbar

- Det. Zeit $O(m\alpha_T(m, n))$ [Chazelle 2000]

- “optimaler” det. Algorithmus [Pettie, Ramachandran 2000]

Verbesserung von Kruskal (parallelisierbar, weniger Sortieraufwand).

[Osipov Sanders Singler 2009]

Zusammenfassung

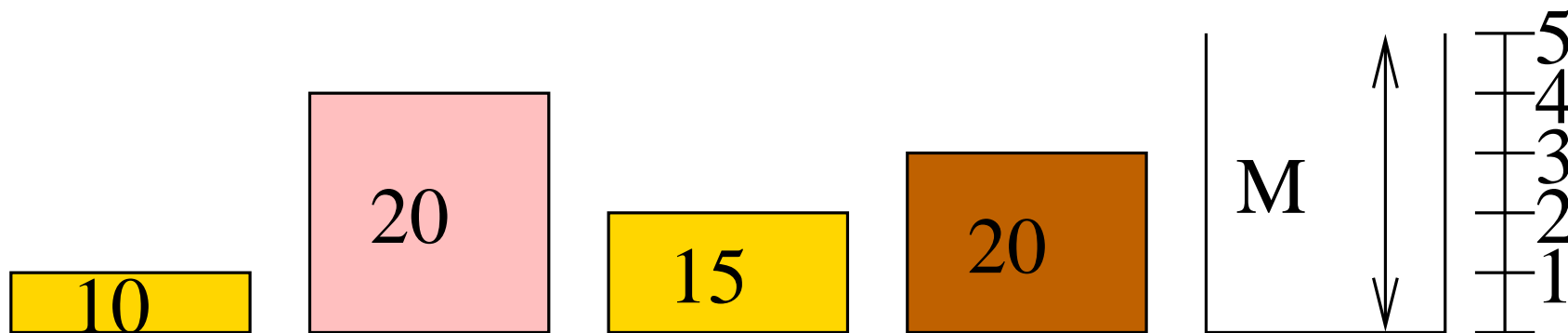
- ☐ **Schnitt-** und **Kreiseigenschaft** als Basis für abstrakte Algorithmen.
Entwurfsprinzip: benutze **abstrakten Problemeigenschaften**.
- ☐ Beweise mittels **Austauschargumenten**
- ☐ Implementierung braucht effiziente **Datenstrukturen**.
Auch ein Entwurfsprinzip...
- ☐ Dijkstra \approx JP.
Noch ein Entwurfsprinzip:
Greedy-Algorithmus effizient implementiert mittels **Prioritätsliste**
- ☐ **Union-Find**: effiziente Verwaltung von Partitionen mittels
Pfadkompression und **Union-by-rank**.
Beispiel für **einfache** Algorithmen mit **nichttrivialer** Analyse

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
- $f : \mathcal{L} \rightarrow \mathbb{R}$ Zielfunktion
- $\mathbf{x}^* \in \mathcal{L}$ ist optimale Lösung falls $f(\mathbf{x}^*) \geq f(\mathbf{x})$ für alle $\mathbf{x} \in \mathcal{L}$

Minimierungsprobleme: analog

Problem: variantenreich, meist NP-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**

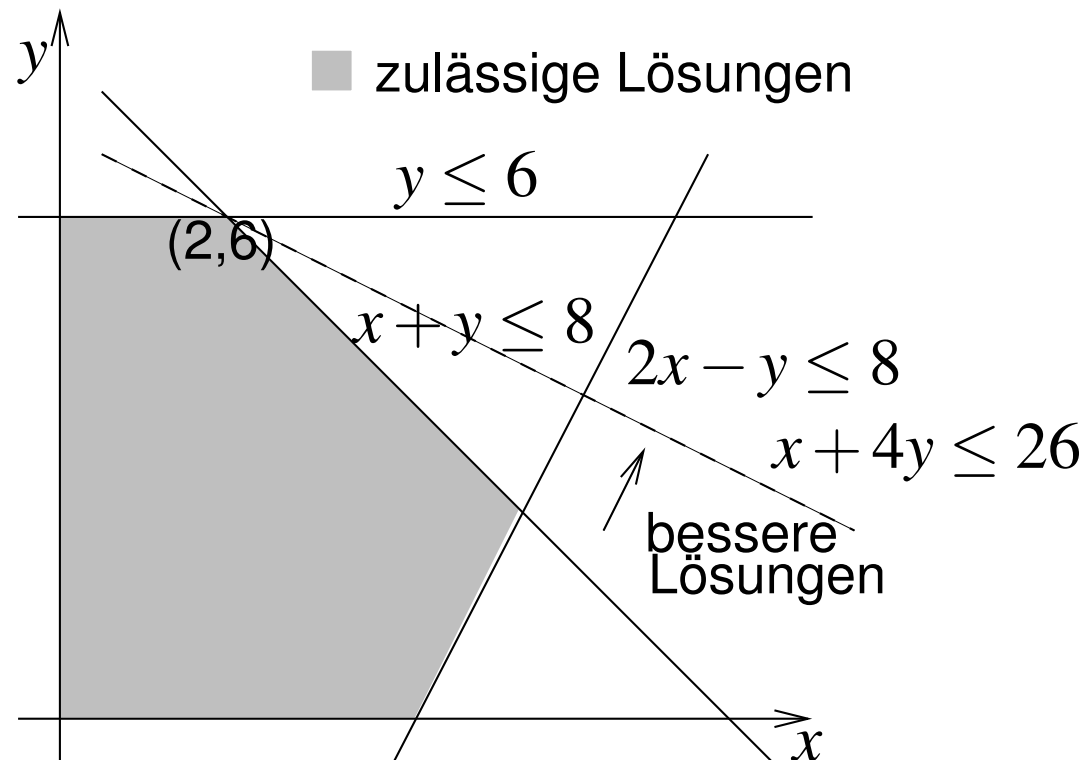
□ m constraints der Form $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$ mit $\bowtie_i \in \{\leq, \geq, =\}$, $\mathbf{a}_i \in \mathbb{R}^n$

Wir erhalten

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

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 abschnittsweise 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 $O\left(\max(m, n)^{\frac{7}{2}}\right)$
 - ☐ In der Praxis geht das viel schneller
 - ☐ Robuste, effiziente Implementierungen sind sehr aufwändig
- ~> Fertige freie und kommerzielle Pakete

Ganzzahlige Lineare Programmierung

ILP: Integer Linear Program, lineares Programm mit der zusätzlichen Bedingung $x_i \in \mathbb{N}$.
oft: 0/1 ILP mit $x_i \in \{0, 1\}$

MILP: Mixed Integer Linear Program, lineares Programm bei dem **einige** Variablen ganzzahlig sein müssen.

Lineare Relaxation: Entferne die Ganzzahligkeitsbedingungen eines (M)ILP

Beispiel: Rucksackproblem

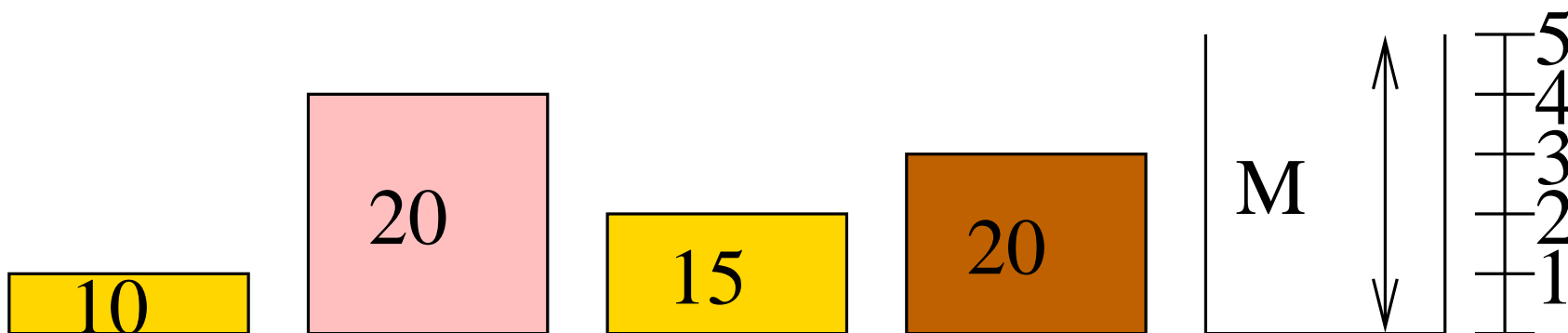
maximiere $\mathbf{p} \cdot \mathbf{x}$

so dass

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

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

0/1 Variablen sind typisch für ILPs



Umgang mit (M)ILPs

- NP-hart
- + 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 $\min \left\{ j : \sum_{i=1}^j w_i > M \right\}$

output items $1..j-1$

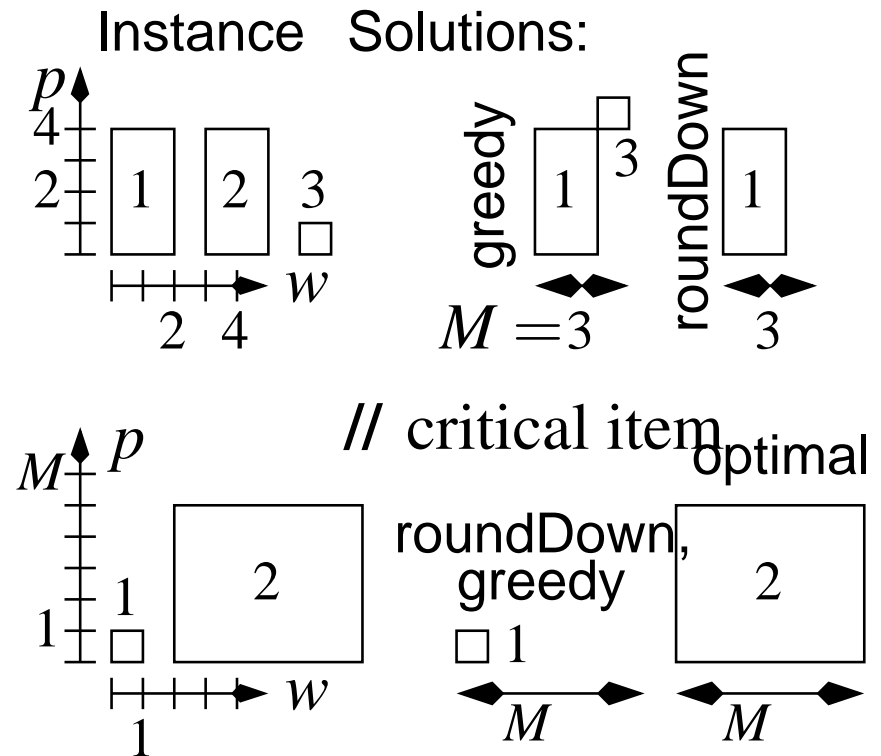
Procedure greedyKnapsack

sort items by **profit density** $\frac{p_i}{w_i}$

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

if there is room for item i **then**

insert it into the knapsack



12.3 Dynamische Programmierung – Aufbau aus Bausteinen

Anwendbar wenn, das **Optimalitätsprinzip** gilt:

- ☐ Optimale Lösungen bestehen aus optimalen Lösungen für Teilprobleme.
- ☐ Mehrere optimale Lösungen \Rightarrow es ist egal welche benutzt wird.

Beispiel: Rucksackproblem

Annahme: ganzzahlige Gewichte

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

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 $1, \dots, i$ bei Kap. C .

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

Beweis:

Sei \mathbf{x} optimale Lösung für Objekte $1..i$, Kapazität C ,
d.h. $\mathbf{c} \cdot \mathbf{x} = P(i, C)$.

Fall $x_i = 0$:

$\Rightarrow \mathbf{x}$ ist auch (opt.) Lösung für Objekte $1..i-1$, Kapazität C .

$\Rightarrow P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$

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

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

Beweis:

Sei \mathbf{x} optimale Lösung für Objekte $1..i$, Kapazität C ,
d.h. $\mathbf{c} \cdot \mathbf{x} = P(i, C)$.

Fall $x_i = 0$: $P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$

Fall $x_i = 1$:

\Rightarrow \mathbf{x} ohne i ist auch Lösung für Objekte $1..i-1$, Kapazität $C - w_i$.

Wegen Austauschbarkeit muß \mathbf{x} ohne i optimal für diesen Fall sein.

$$\Rightarrow P(i, C) - p_i = P(i-1, C - w_i)$$

$$\Leftrightarrow P(i, C) = P(i-1, C - w_i) + p_i$$

Insgesamt, wegen Optimalität von \mathbf{x} ,

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

□

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

Procedure knapsack(**p**, **c**, n , M)

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

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

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

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

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

if $P[C - w_i] + p_i > P[C]$ **then**

$P[C] := P[C - w_i] + p_i$

decision $[i, C] := 1$

Rekonstruktion der Lösung

```
 $C := M$   
array  $\mathbf{x}[1 \dots n]$   
for  $i := n$  downto 1 do  
     $\mathbf{x}[i] := \text{decision}[i, C]$   
    if  $\mathbf{x}[i] = 1$  then  $C := C - w_i$   
endfor  
return  $\mathbf{x}$ 
```

Analyse:

Zeit: $O(nM)$ pseudopolynomiell

Platz: $M + O(n)$ Maschinenwörter plus Mn bits.

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
- ☐ Verkettete Matrixmultiplikation Übung?
- ☐ Rucksackproblem Gegenstände $1..i$ füllen Teil des Rucksacks
- ☐ Geld wechseln Übung?

Gegenbeispiel: Teilproblemeigenschaft

Angenommen, die schnellste Strategie für 20 Runden auf dem Hockenheimring verbraucht den Treibstoff vollständig.

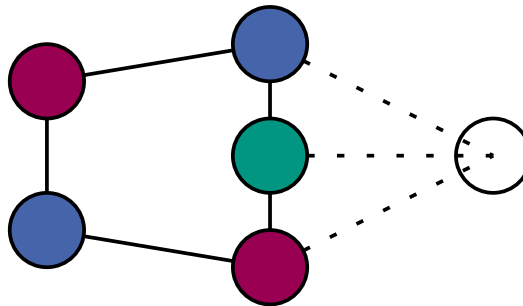


Keine gute Teilstrategie für 21 Runden.

Frage: Wie kann man “constrained shortest path” trotzdem mittels dynamischer Programmierung modellieren?

Gegenbeispiel: Austauschbarkeit

Optimale Graphfärbungen sind nicht austauschbar.



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 $\text{bbKnapsack}((p_1, \dots, p_n), (w_1, \dots, w_n), M) : \mathcal{L}$

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

$\hat{\mathbf{x}} = \text{heuristicKnapsack}((p_1, \dots, p_n), (w_1, \dots, w_n), M) : \mathcal{L}$

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

$\text{recurse}(0, M, 0)$

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

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

// $M' = M - \sum_{k < i} x_k w_k, P = \sum_{k < i} x_k p_k.$

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

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

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

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

if $u > \mathbf{p} \cdot \hat{\mathbf{x}}$ **then**

if $i > n$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

else // **Branch** on variable x_i

if $w_i \leq M'$ **then** $x_i := 1$; recurse($i + 1, M' - w_i, P + p_i$)

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

Schlechtester Fall: 2^n rekursive Aufrufe

Im Mittel: Linearzeit?

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),...

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 $\mathbf{x} \in \mathcal{L}$

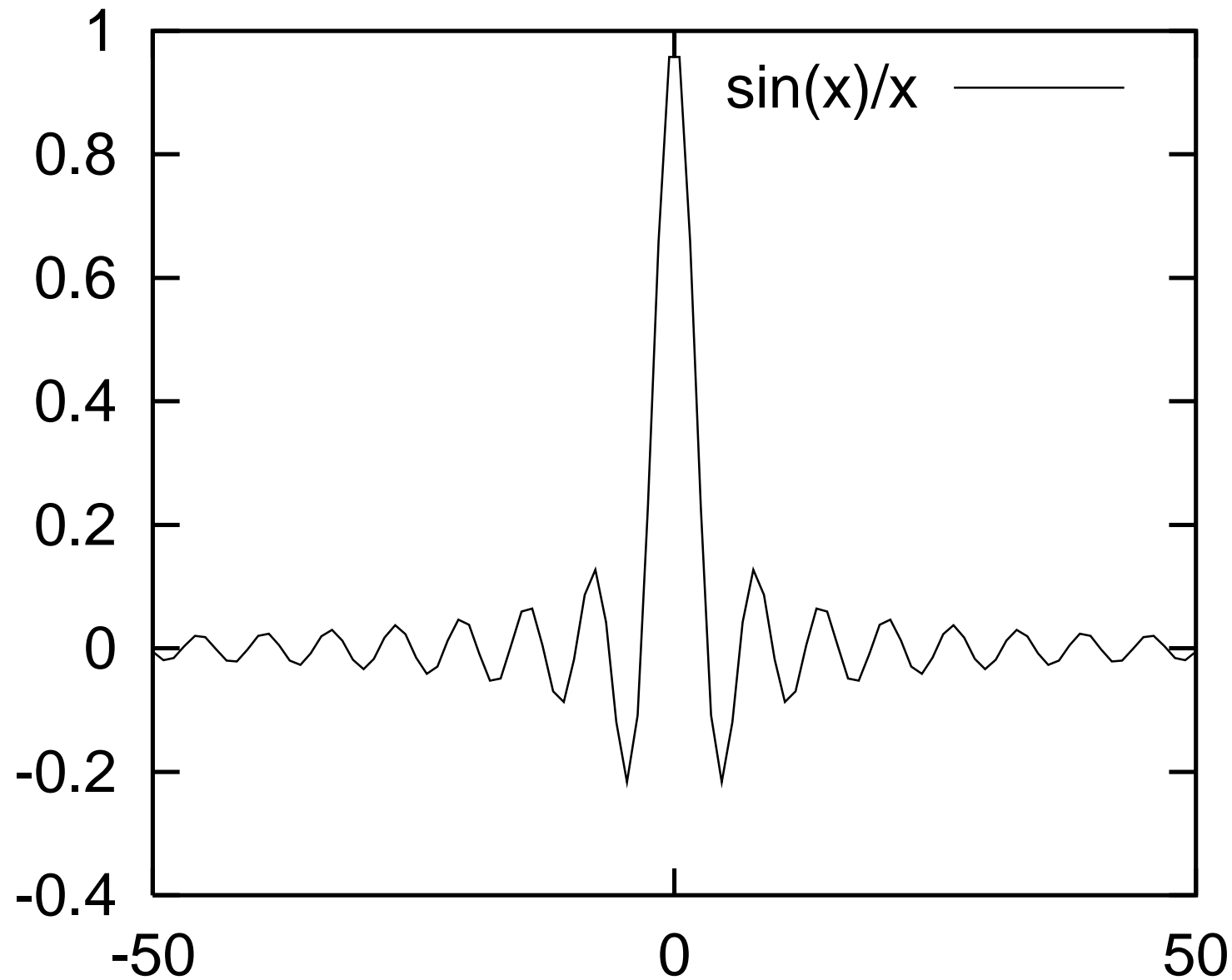
$\hat{\mathbf{x}} := \mathbf{x}$ // best solution found so far

loop

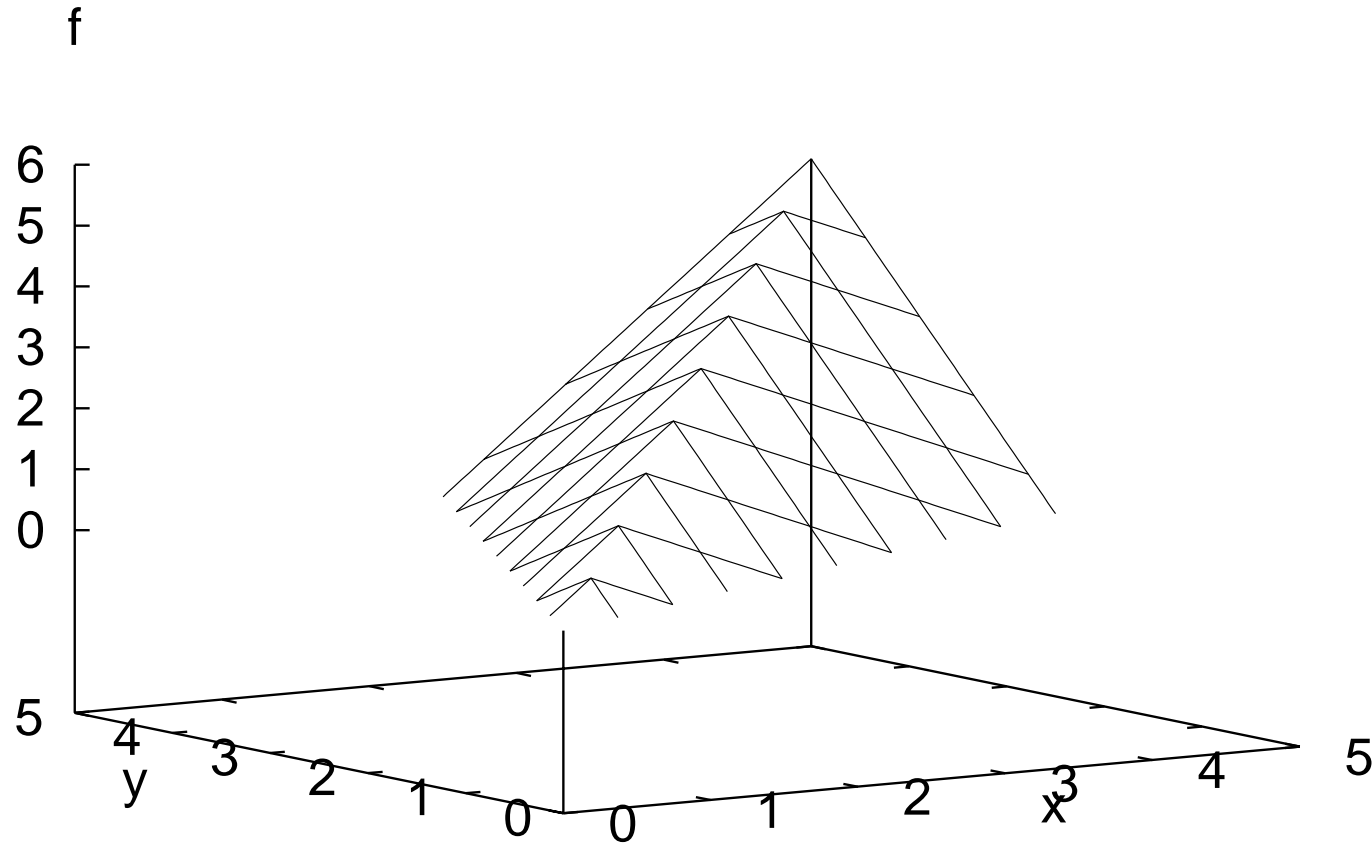
if $\exists \mathbf{x} \in \mathcal{N}(\mathbf{x}) \cap \mathcal{L} : f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

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

Problem: Lokale Optima



Warum die Nachbarschaft wichtig ist



Gegenbeispiel für Koordinatensuche

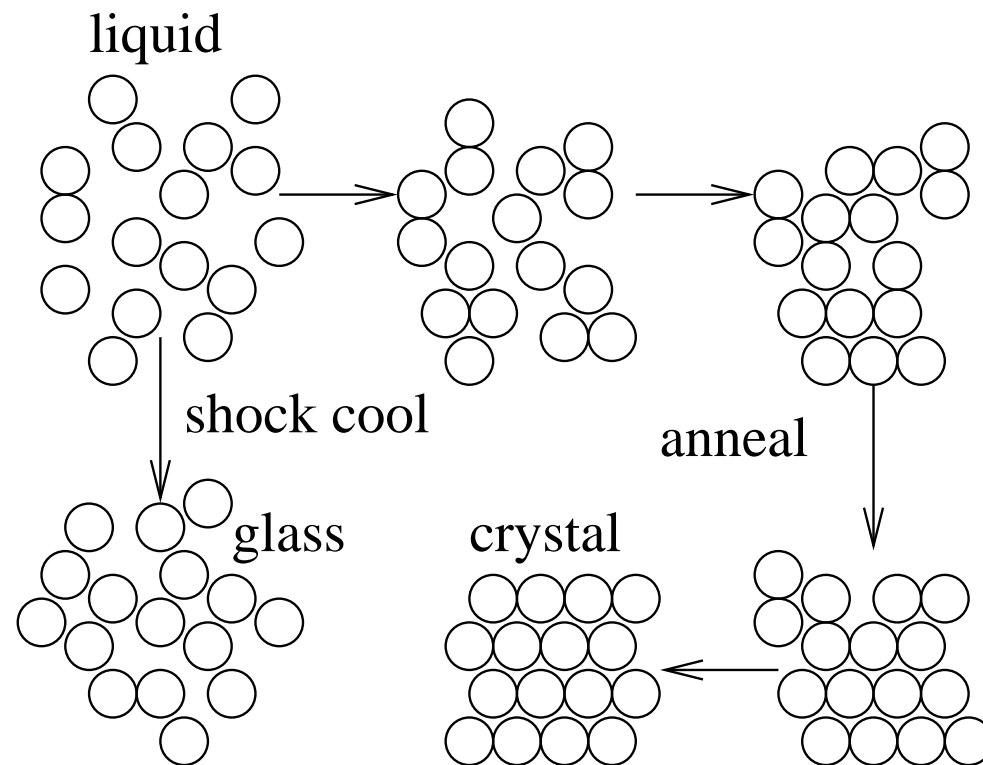
Jenseits von Hill Climbing

Auch Verschlechterungen akzeptieren.

☐ Simulated Annealing: physikalische Analogie

☐ Tabusuche

☐ ...



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 viel know how erfordern.

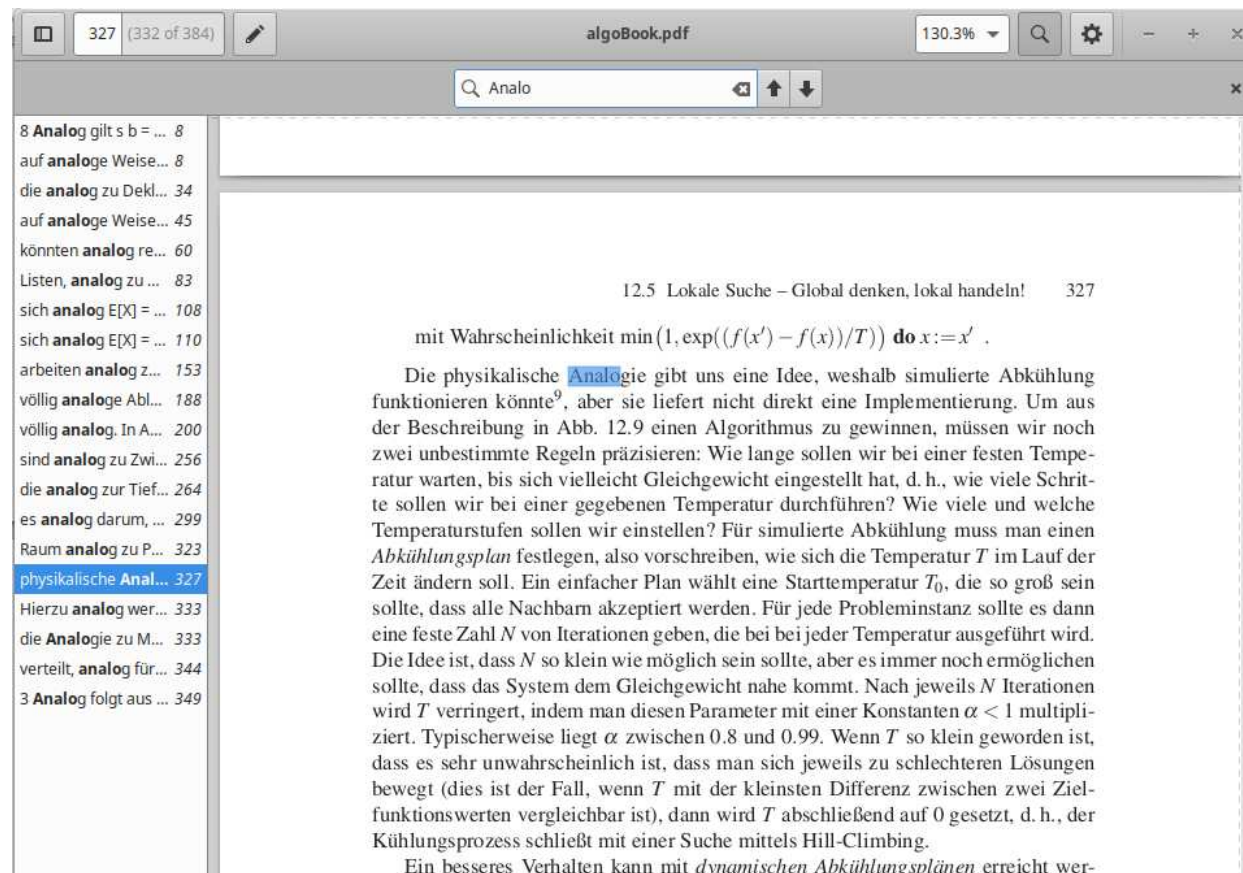
Lokale Suche: **Flexibel** und einfach. **Langsam** aber oft **gute Lösungen** für harte Probleme.

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.

13 String Algorithmen



Übersicht

- ☐ String Matching Algorithmen
 - Karp und Rabin
 - Knuth Morris Pratt (nicht hier)
- ☐ Suffix Bäume

Literatur: Cormen-Leiserson-Rivest-Stein [CLRS]: Introduction to Algorithms Chapter 32.2, 32.4

13.1 Volltextsuche von Langsam bis Superschnell

Gegeben: Text S ($n := |S|$), Muster (Pattern) P ($m := |P|$), $n \gg m$

Gesucht: Alle/erstes/nächstes Vorkommen von P in S

naiv: $O(nm)$

P vorverarbeiten: $O(n + m)$

Mit Fehlern: ???

S vorverarbeiten: Textindizes. Erstes Vorkommen:

Suffix Array: $O(m \log n) \dots O(m)$

Brute Force

Idee: vergleiche Pattern P mit jedem Teilstring $S[i..j]$ der Länge m .

```
Function naivePatternSearch( $P, S$ ) :  $\mathbb{N} \cup \{\infty\}$   
    for  $i := 0$  to  $n - m$  do  
         $j := 0$   
        while ( $j < m$  and  $S[i + j] = P[j]$ ) do  
             $j++$   
        if  $j = m$  then return  $i$   
    return  $\infty$ 
```

Laufzeit: $O(nm)$

Beispiel

→ Tafel

Schneller?

Idee: Pattern nicht direkt vergleichen, sondern erstmal Fingerprints

Fingerprint: kurze Label für große Objekte wie Strings

$F = \{f : \Omega \rightarrow \{0, 1\}^k\}$, so dass für jede endliche Teilmenge $S \subset \Omega$ gilt: wenn f zufällig und gleichverteilt aus F gezogen wird, dann entsteht mit hoher Wahrscheinlichkeit keine Kollision, d.h. für $x \in S$ und $y \in S, x \neq y$ gilt $f(x) \neq f(y)$

Karp, Rabin Algorithmus

Function KarpRabin(P, S)

$x := f(P)$ // f is the fingerprint function

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

$x' := f(S[i..i + m - 1])$

if $x' = x$ **then return** "Match?"

return "No Match"

Laufzeit: Naiv immer noch $O(nm)$, wenn Berechnung von f auf String der Größe m Zeit $O(m)$ braucht.

→ Neuberechnen von Fingerprints in jeder Iteration ist teuer!

Karp, Rabin Algorithmus

Function KarpRabin(P, S) : String

$x := f(P)$ // f is the fingerprint function

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

$x' := f(S[i..i + m - 1])$

if $x' = x$ **then return** "Match?"

return "No Match"

Laufzeit: Naiv immer noch $O(nm)$, wenn Berechnung von f auf String der Größe m Zeit $O(m)$ braucht.

→ Neuberechnen von Fingerprints in jeder Iteration ist teuer!

Idee: benutze cleverere Funktionen, z.B. so dass $f(S[i..i + m - 1])$ in konstanter Zeit von $f(S[i - 1..i + m - 2])$ berechnet werden kann

Karp, Rabin Algorithmus

Function KarpRabin(P, S) : String

$x := f(P)$ // f is the fingerprint function

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

$x' := f(S[i..i + m - 1])$ // update in $O(1)$

if $x' = x$ **then return** "Match?"

return "No Match"

Idee: benutze cleverere Funktionen, z.B. so dass $f(S[i..i + m - 1])$ in konstanter Zeit von $f(S[i - 1..i + m - 2])$ berechnet werden kann

Laufzeit dann: $O(m + n)$

Ausgabe Garantieren

```
Function KarpRabin( $P, S$ ) :  $\mathbb{N} \cup \{\infty\}$   
     $x := f(P)$  //  $f$  is the fingerprint function  
    for  $i := 0$  to  $n - m$  do  
         $x' := f(S[i..i + m - 1])$  // update in  $O(1)$   
        if  $x' = x$  then  
            if check=( $P, S[i..i + m - 1]$ ) then return  $i$   
    return  $\infty$ 
```

Garantieren das Pattern stimmt? $\rightarrow O(m)$ check einbauen

Worst case dann $O(nm)$

aber erwartete Anzahl Ausführungen vom check ist $O(1)$

\rightarrow erwarte Laufzeit $O(n + m)$

Ein Text, viele Muster

Bisher ein Pattern, ein Text. Was tut man, wenn man einen großen Text hat und diesen immer wieder nach Patterns durchsuchen möchte?

Anwendungen

- ☐ Volltextsuche
- ☐ Burrows-Wheeler Transformation (bzip2 Kompressor)
- ☐ Bioinformatik: Wiederholungen suchen,...

13.2 Strings Sortieren

multikey quicksort

Function mkqSort(s : Sequence **of** String, i : \mathbb{N}) : Sequence **of** String

assert $\forall e, e' \in s : e[1..i-1] = e'[1..i-1]$

if $|s| \leq 1$ **then return** s // base case

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

return concatenation of
 mkqSort($\langle e \in s : e[i] < p[i] \rangle, i$),
 mkqSort($\langle e \in s : e[i] = p[i] \rangle, i+1$), and
 mkqSort($\langle e \in s : e[i] > p[i] \rangle, i$)

□ Laufzeit: $O(|s| \log |s| + \sum_{t \in s} |t|)$

Suffixe Sortieren

Sortiere die Menge $\{S_0, S_1, \dots, S_{n-1}\}$
 von Suffixen des Strings S der Länge n
 (Alphabet $[1, n] = \{1, \dots, n\}$)
 in lexikographische Reihenfolge.

□ suffix $S_i = S[i, n]$ für $i \in [0..n-1]$

$S = \text{banana}$:

0	banana		5	a
1	anana		3	ana
2	nana	\Rightarrow	1	anana
3	ana		0	banana
4	na		4	na
5	a		2	nana

Volltextsuche

Suche **Muster (pattern)** $P[0..m)$ im Text $S[0..n)$
mittels Suffix-Tabelle SA of S .

Binäre Suche: $O(m \log n)$ gut für kurze Muster

Suffix-Baum: $O(m)$, Konstruktion in $O(n)$ Zeit

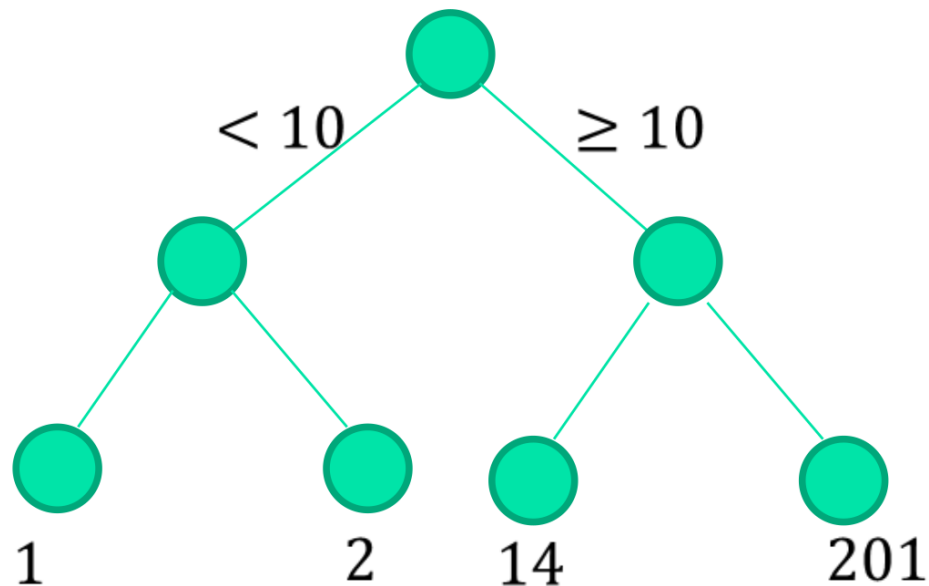
Versuch 1

Beispiel: banana

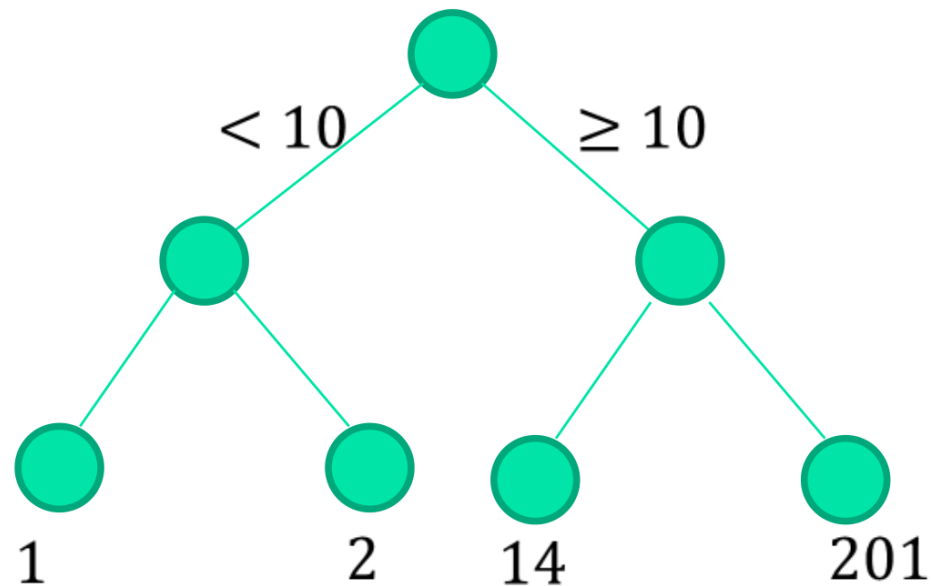
Teilstrings: "b", "a", "n", "ba", "an", ...

Konvertieren in Zahlen: "b"=02, "a"=01, "n"=14, "ba"=0201

Allgemein: Teilstring der Länge k wird Zahl der Länge $2k$



Versuch 1 – Aufbau



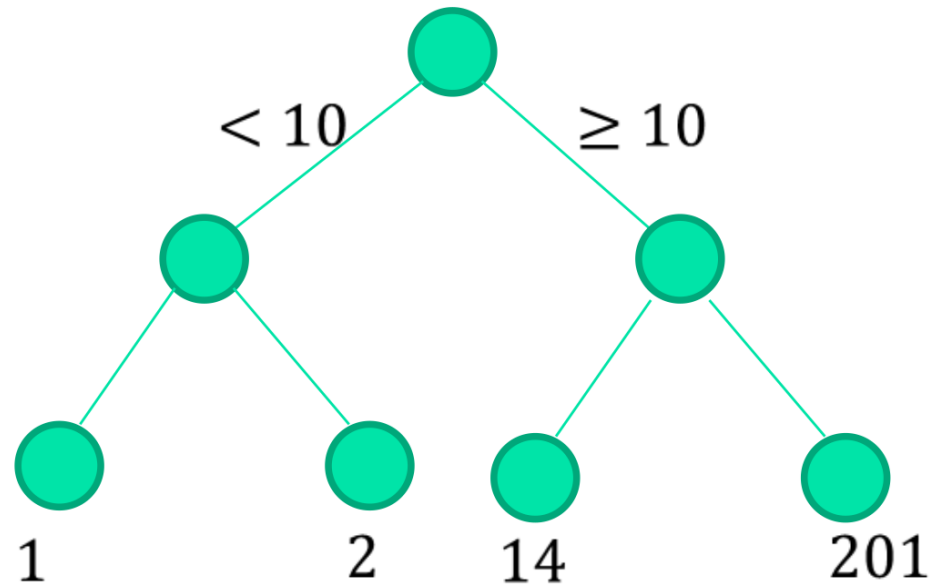
Für jeden Teilstring Wert W berechnen

W in Binärbaum einfügen

Beachte: $O(n^2)$ Teilstrings und $O(n)$ Arbeit pro Teilstring

Laufzeit für Konstruktion insgesamt $O(n^3)$, warum?

Versuch 1 – Aufbau

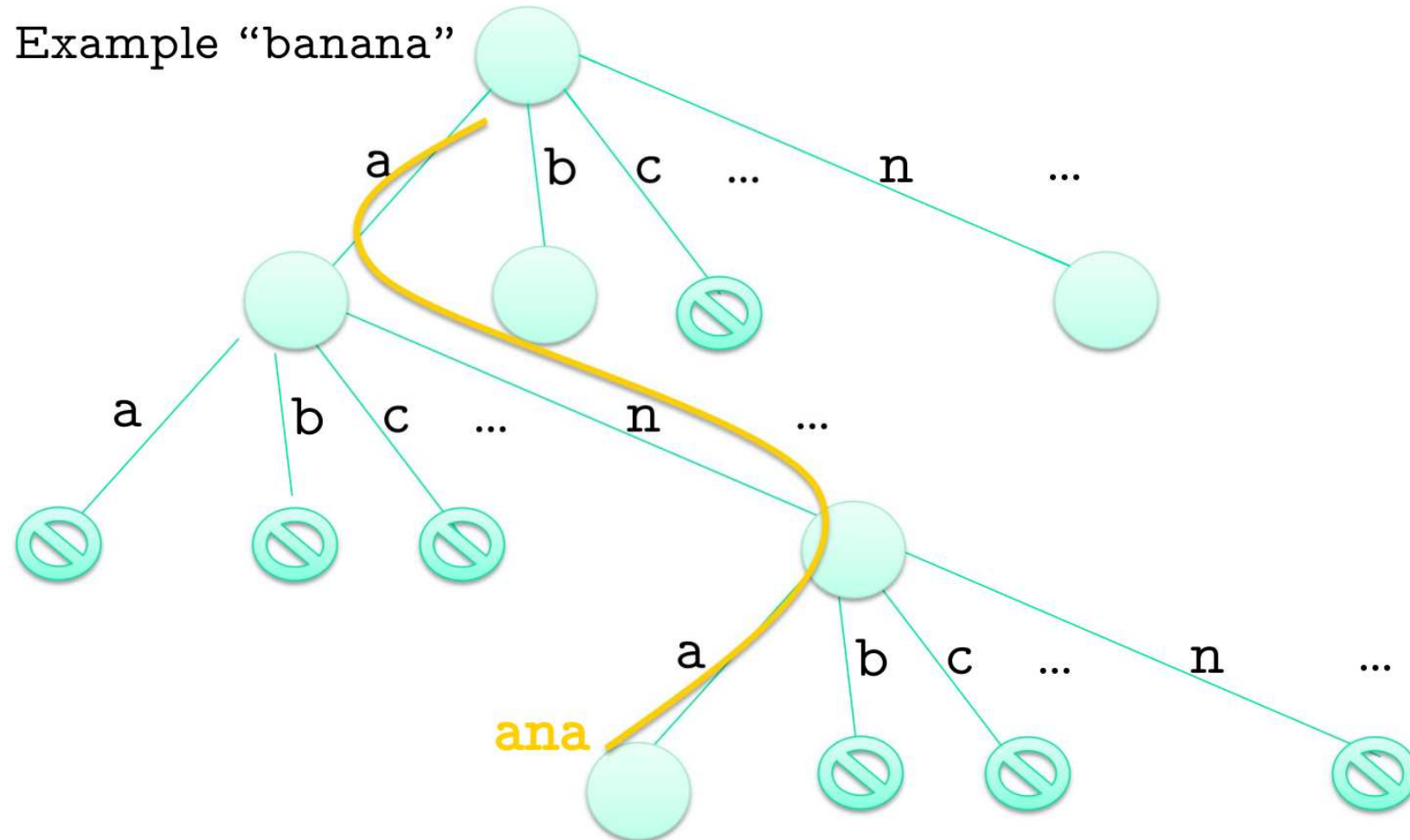


Queryzeit: $O(m + \log n)$

Platz: $O(n^2)$

Besser als Rabin-Karp für hohe Zahl von Queries

Example “banana”



Konstruktionszeit immer noch $O(n^3)$

Versuch 2

Idee: es genügt Suffixe in den Baum einzufügen!

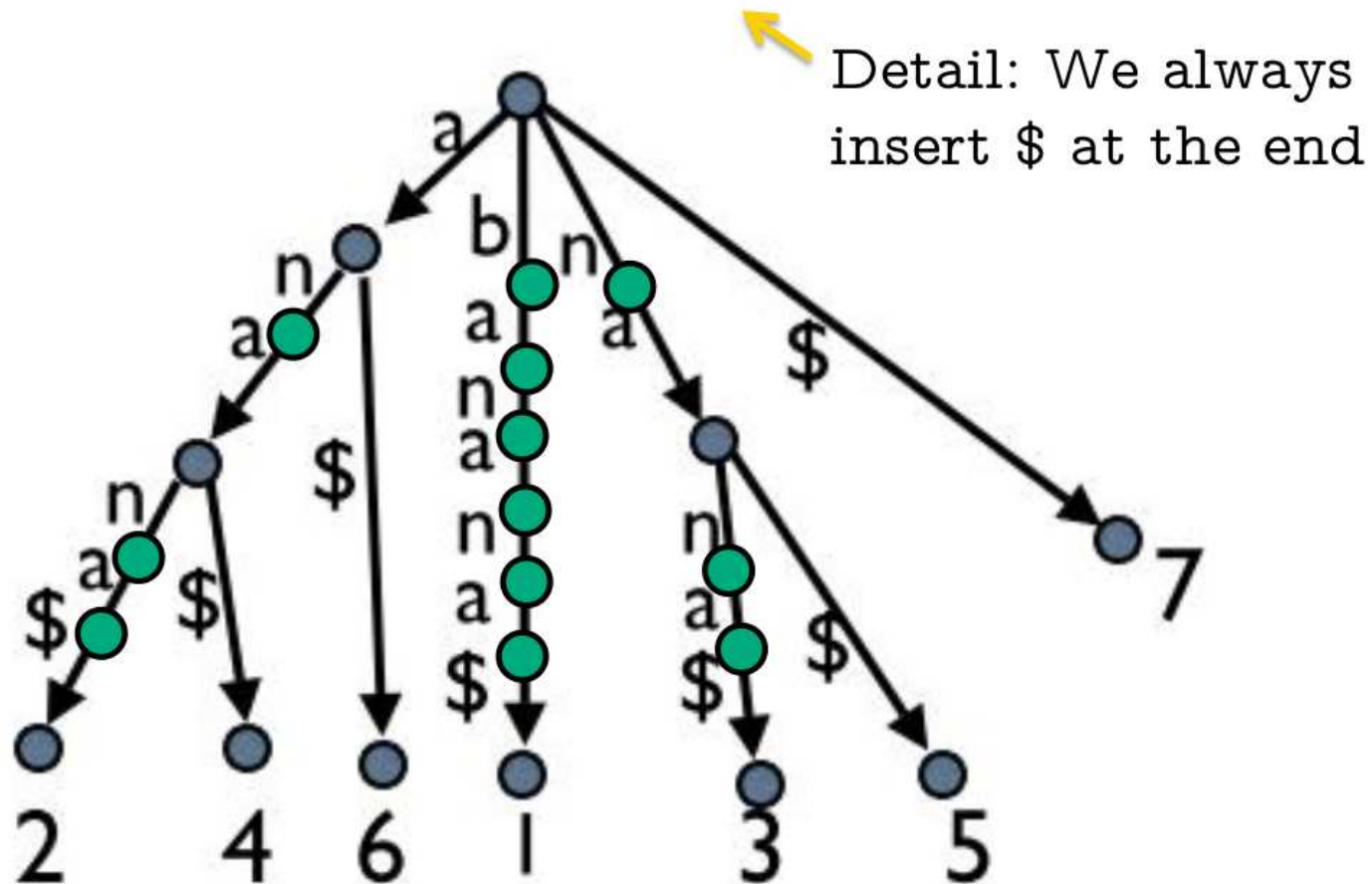
Es gibt nur $O(n)$ Suffixe (Länge $O(n)$)

Konstruktion: Für alle Suffixe S_i von S füge S_i in Trie ein

Diesen Baum nennt man auf Suffix-Tree

Suche: Pfade in Baum verfolgen. Wenn man in einem Knoten vom Baum endet ist das Pattern enthalten.

Beispiel: Suffix Tree für banana\$



Detail: We also write down the positions

Laufzeit

Query: $O(m)$, wie gehabt

Konstruktion: $O(n^2)$, Einfügen von $O(n)$ Suffixen

Platz: $O(n^2)$, es kann $O(n^2)$ Knoten geben

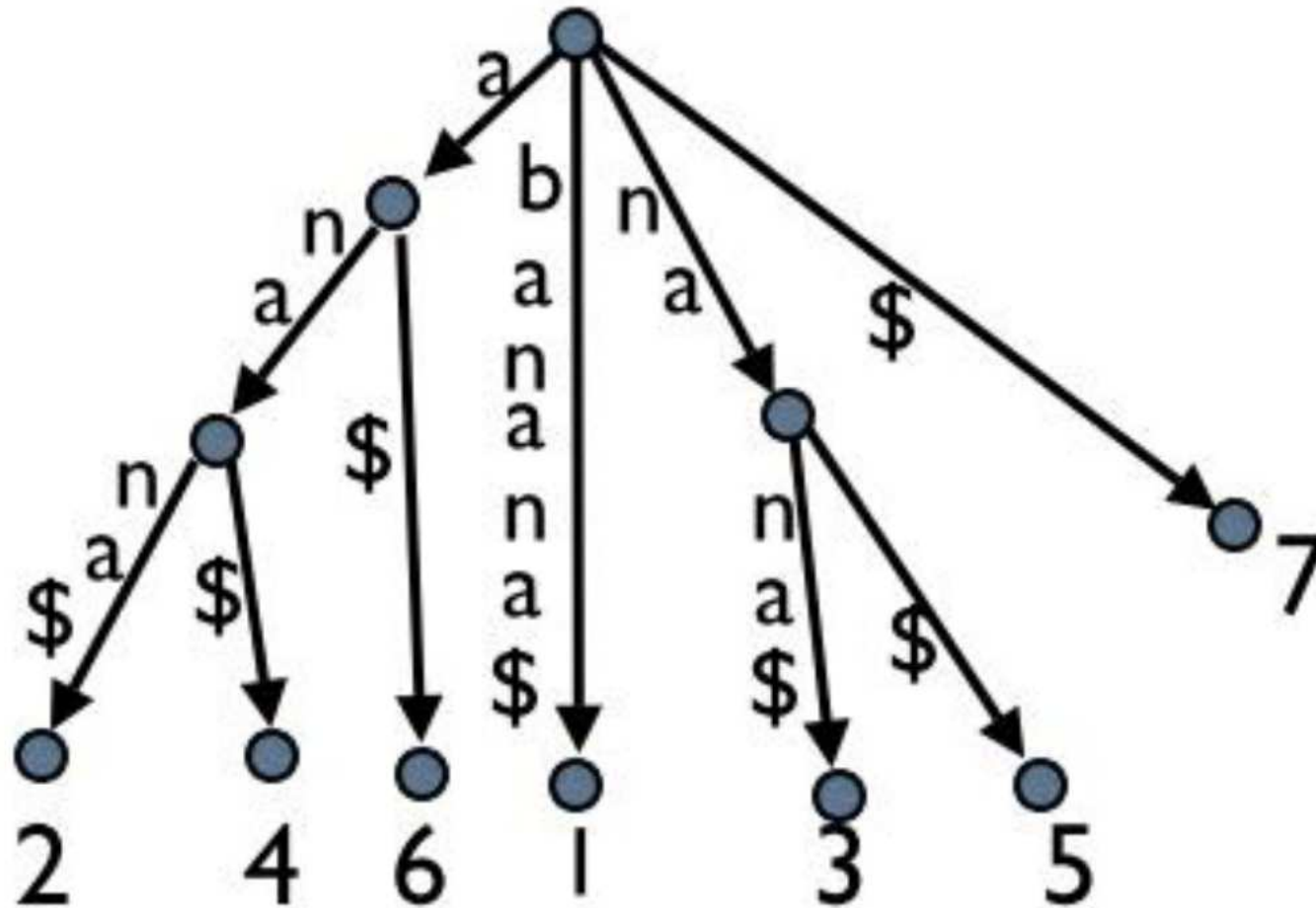
Detail: We always insert \$ at the end

Detail: We also write down the positions

[illegible]

Sicherstellen das alle Knoten $\text{Grad} \geq 2$ haben.

Anzahl Knoten reduzieren?

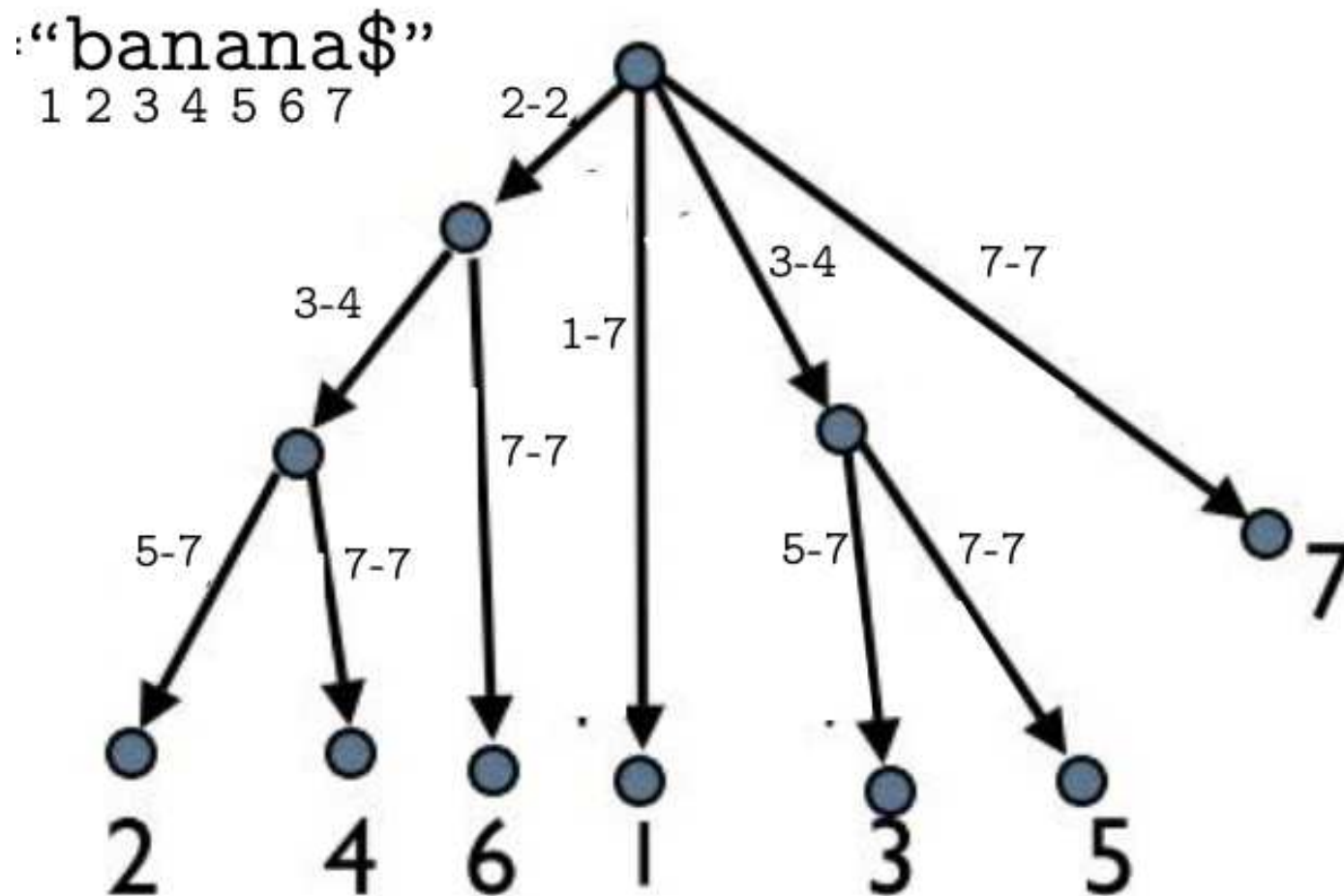


Blatt \leftrightarrow Suffix $\Rightarrow O(n)$ Knoten

A search tree diagram illustrating the construction of the word "banana". The root node is at the top. The first level of branches is labeled 'a', 'b', 'n', 'a', and '\$'. The second level of branches is labeled 'n', '\$', 'a', 'n', 'a', '\$', '\$', and '\$'. The third level of branches is labeled '\$', '\$', '\$', '\$', '\$', '\$', '\$', '\$', '\$', and '\$'. The leaf nodes are labeled 2, 4, 6, 1, 3, 5, and 7.

Problem: wir haben trotzdem noch $O(n^2)$ Charakter

Anzahl Knoten reduzieren?



Referenzen mit Start und Endpunkt in S verwenden! $\rightarrow O(n)$ Platz

Mehr Suffix-Bäume

- Suffix Bäume können sogar in Zeit $O(n)$ konstruiert werden, für konstant große Alphabete

E. Ukkonen. (1995). On-line construction of suffix trees. *Algorithmica* 14(3):249-260.

Zusammenfassung

- ☐ Datenstrukturen
- ☐ Algorithmen
- ☐ Entwurfstechniken
- ☐ Analysetechniken

Zusammenfassung – Datenstrukturen

- ☐ (doppelt) verkettete **Listen**, unbeschränkte (zyklische) **Felder**, Stapel, FIFOs, deques
- ☐ (beschränktes) **Hashing**: geschlossen (universell) / offen
- ☐ sortiertes Feld
- ☐ Prioritätslisten (**binärer Heap**) (adressierbar)
- ☐ Implizite Repräsentation vollständiger Bäume
- ☐ Suchbäume: binär, **(a, b) -Baum**
- ☐ Graphen: **Adjazenzfeld** / Listen / Matrix
- ☐ Union-Find

Zusammenfassung – Algorithmen

- ☐ Langzahlmultiplikation
- ☐ Insertion-, Merge-, Quick-, Heap-, Bucket-, Selektion
- ☐ BFS, DFS, topologisches Sortieren
- ☐ Kürzeste Wege: Dijkstra, Bellman-Ford
- ☐ MST: Jarník-Prim, Kruskal
- ☐ String-Algorithmen

Zusammenfassung – Entwurfstechniken

- ☐ Iteration/Induktion/Schleifen, Teile-und-Herrsche
- ☐ Schleifen- und Datenstruktur-Invarianten
- ☐ Randomisierung (universelles Hashing, Quicksort, . . .)
- ☐ Graphenmodelle
- ☐ Trennung Mathe \leftrightarrow Funktionalität \leftrightarrow Repräsentation \leftrightarrow
Algorithmus \leftrightarrow Implementierung
- ☐ Sonderfälle vermeiden
- ☐ Zeigerdatenstrukturen
- ☐ Datenstrukturen augmentieren (z.B. Teilbaumgrößen)
- ☐ Datenstrukturen unbeschränkt machen
- ☐ Implizite Datenstrukturen (z.B. Intervallgraphen)

☐ Algebra

(Karatsuba, univ. Hashfkt., Matrixmultiplikation für Graphen)

☐ Algorithmenschemata (z.B. DFS, lokale Suche)

☐ Verwendung abstrakter Problemeigenschaften

(z.B. Schnitt/Kreis-Eigenschaft bei MST)

☐ Black-Box-Löser (z.B. lineare Programmierung)

☐ Greedy

☐ Dynamische Programmierung

☐ Systematische Suche

☐ Metaheuristiken (z.B. Lokale Suche)

Zusammenfassung – Analysetechniken

- ☐ Summen, Rekurrenzen, Induktion, Master-Theorem, Abschätzung
- ☐ Asymptotik ($O(\cdot)$, \dots , $\omega(\cdot)$), einfache Modelle
- ☐ Analyse im Mittel
- ☐ Amortisierung (z.B. unbeschränkte Felder)
- ☐ Linearität des Erwartungswertes, Indikatorzufallsvariablen
- ☐ Kombinatorik (\approx Zählen): univ. Hashfunktionen, untere Sortierschranken (Informationsmenge)
- ☐ Integrale als Summenabschätzung
- ☐ Schleifen/Datenstruktur-(In)varianten
(z.B. (a, b) -Baum, Union-by-rank)

Zusammenfassung – weitere Techniken

- ☐ Algorithm Engineering
- ☐ Parameter Tuning (z.B. Basisfallgröße)
- ☐ High-Level Pseudocode
- ☐ Dummys und Sentinels (Listen, insertion sort, . . .)
- ☐ Speicherverwaltung