Karlsruhe Institute of Technology

Algorithmen II

Peter Sanders, Thomas Worsch, Simon Gog

Übungen:

Demian Hespe, Yaroslav Akhremtsev

Institut für Theoretische Informatik, Algorithmik II

Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS17.php`

Organisatorisches

Vorlesungen:

Mo 09:45–11:15 HS Neue Chemie

Di 15:45–16:30 HS Neue Chemie

Saalübung:

Di 16:30–17:15 HS Neue Chemie

Übungsblätter:

14-tägig, jeweils Dienstags, Musterlösung 9 Tage später

1. Blatt: 24.10.2017

Ilias Forum:

Fragen zum Vorlesungsinhalt (auch anonym möglich)

Link auf Vorlesungswebseite

Organisatorisches

Sprechstunden:

- Peter Sanders, Dienstag 13:45–14:45 Uhr, Raum 217
- Thomas Worsch, Freitag 8:00–9:00 Uhr, Raum 230
- Simon Gog, Nach Vereinbarung, Raum 220 (erst ab November)
- Demian Hespe, Nach Vereinbarung, Raum 210
- Yarsolav Akhremtsev, Donnerstag 14:00–15:00 Uhr, Raum 221

Letzte Vorlesung: 06. Februar 2018

Klausur: Mittwoch, 21. Februar 2018, 11:30 Uhr

Materialien

- Folien
- Übungsblätter
- Buch:

  K. Mehlhorn, P. Sanders

  Algorithms and Data Structures — The Basic Toolbox

  Springer 2008. Ca. 40 % der Vorlesung.
- Skript: Minimalistischer Aufschrieb der Sachen, die nicht im Buch stehen, mit Verweisen auf Originalliteratur

  Sprache: Deutsch.

  Aber Literatur und ein Teil der Folien auf Englisch

Zum Weiterlesen

- [Mehlhorn, Näher] Algorithm Engineering, Flows, Geometrie

  The LEDA Platform of Combinatorial and Geometric Computing.

- [Ahuja, Magnanti, Orlin] Network Flows

- [de Berg, Cheong, van Kreveld, Overmars] Geometrie

  Computational Geometry: Algorithms and Applications

- [Gonzalo Navarro] Succinct Data Structures

  Compact Data Structures: A Practical Approach

- [R. Niedermeier] Invitation to Fixed-Parameter Algorithms

Inhaltsübersicht I

- Algorithm Engineering
- Fortgeschrittene Datenstrukturen am Beispiel von Prioritätslisten
  - addressierbar
  - ganzzahlige Schlüssel
  - (Externspeicher)
- Fortgeschrittene Graphenalgorithmen
  - Kürzeste Wege II: negative Kreise, Potentialmethode
  - Starke Zusammenhangskomponenten
  - Maximale Flüsse und Matchings

Inhaltsübersicht II

- “Bindestrichalgorithmiken”
  - Randomisierte Algorithmen
  - Externe Algorithmen
  - Parallele Algorithmen
  - Stringalgorithmen: sorting, indexing,. . .
  - Geometrische Algorithmen
  - Approximationsalgorithmen
  - Fixed-Parameter-Algorithmen
  - Onlinealgorithmen

Karlsruhe Institute of Technology

1 Algorithm Engineering

A detailed definition

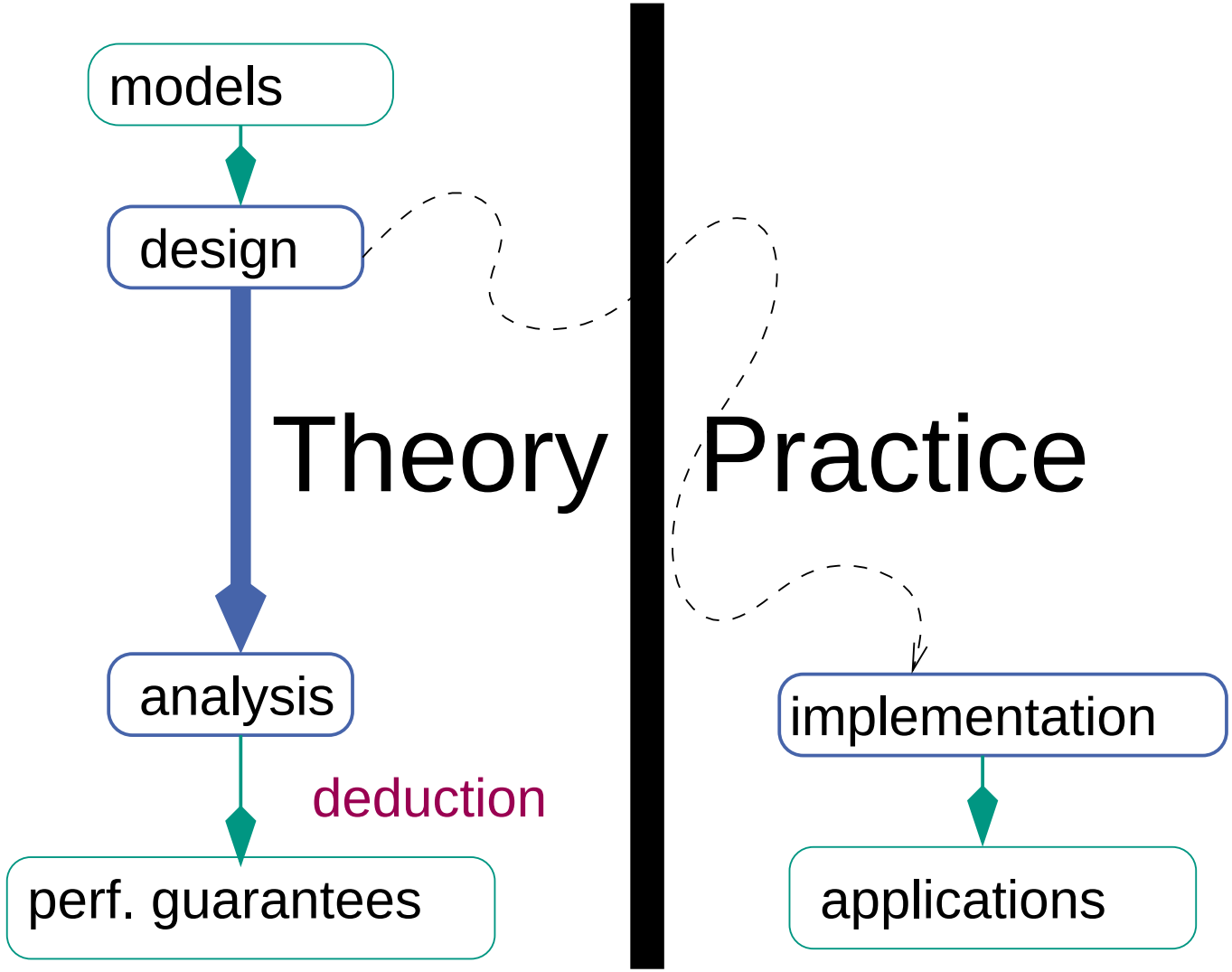
- in general

  [with Kurt Mehlhorn, Rolf Möhring, Petra Mutzel, Dorothea Wagner]

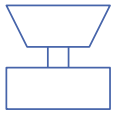
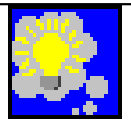
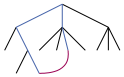
- A few examples, usually sorting

- A little bit on experimental methodology

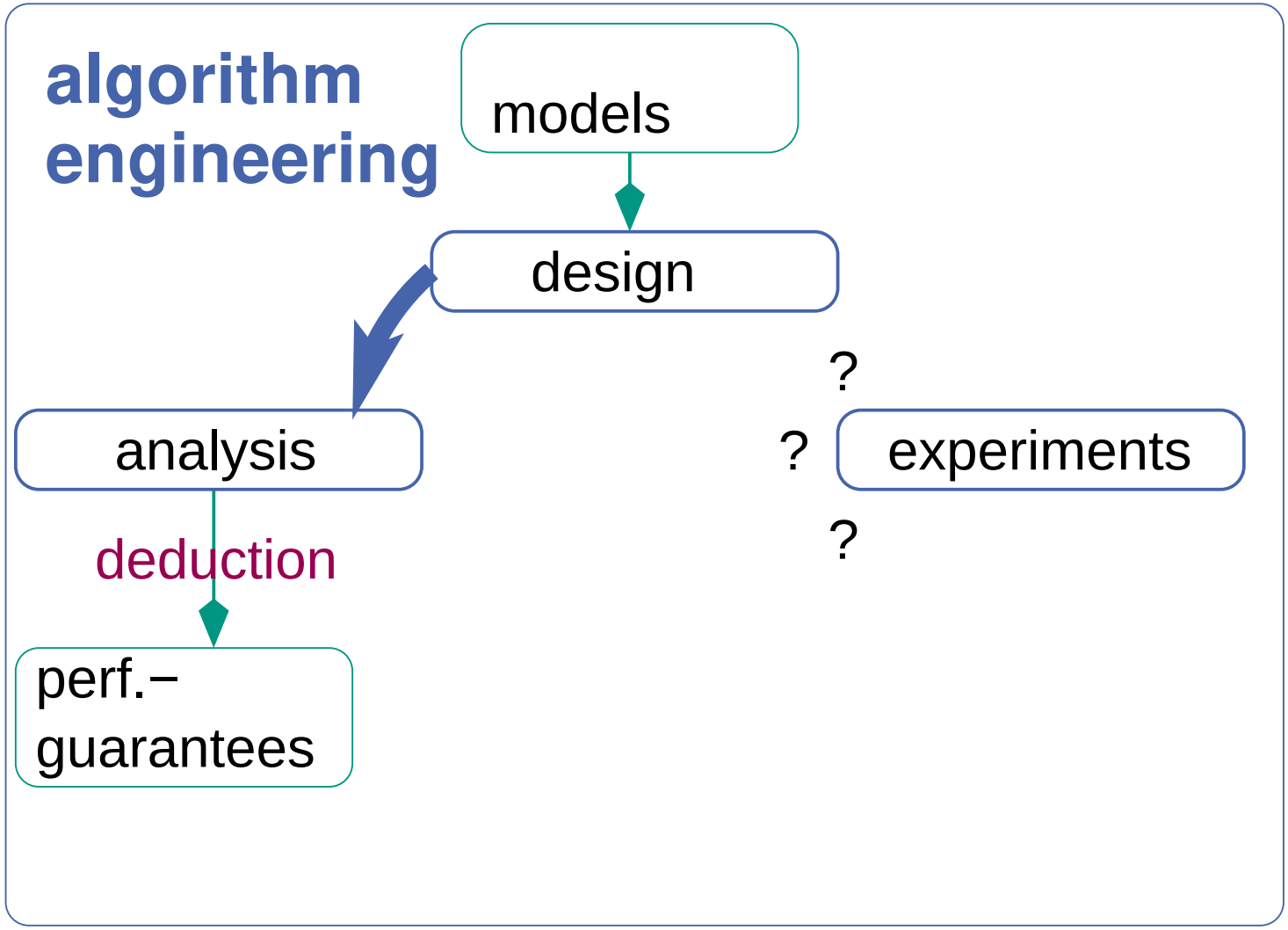
(Caricatured) Traditional View: Algorithm Theory



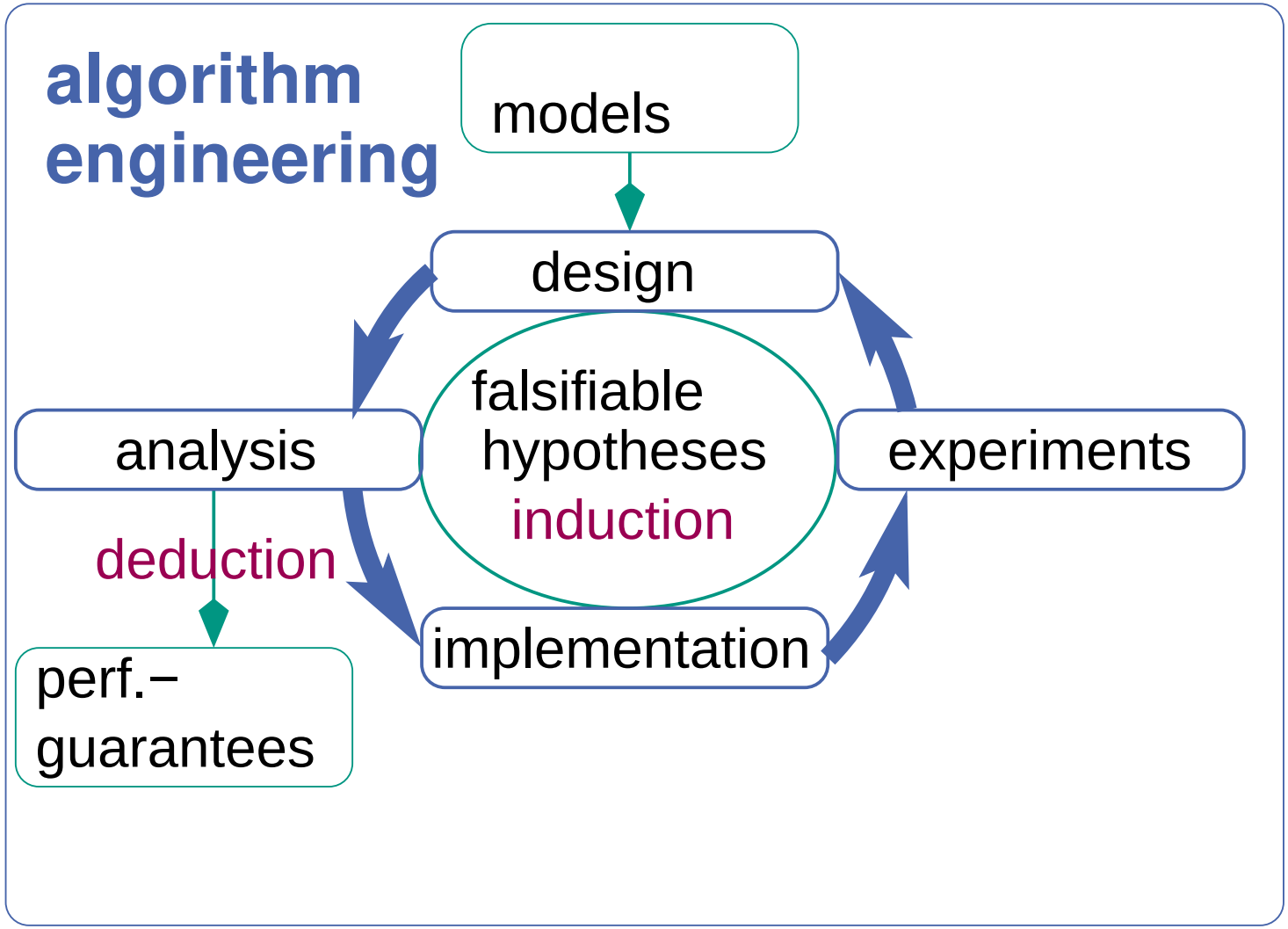
Gaps Between Theory & Practice

Theory	$\longleftrightarrow$	**Practice**
simple 	**appl. model**	 complex
simple 	**machine model**	 real
complex 	**algorithms**	FOR simple
advanced 	**data structures**	 arrays,. . .
worst case max	**complexity measure**	 inputs
asympt. $O(\cdot)$	**efficiency**	42% constant factors

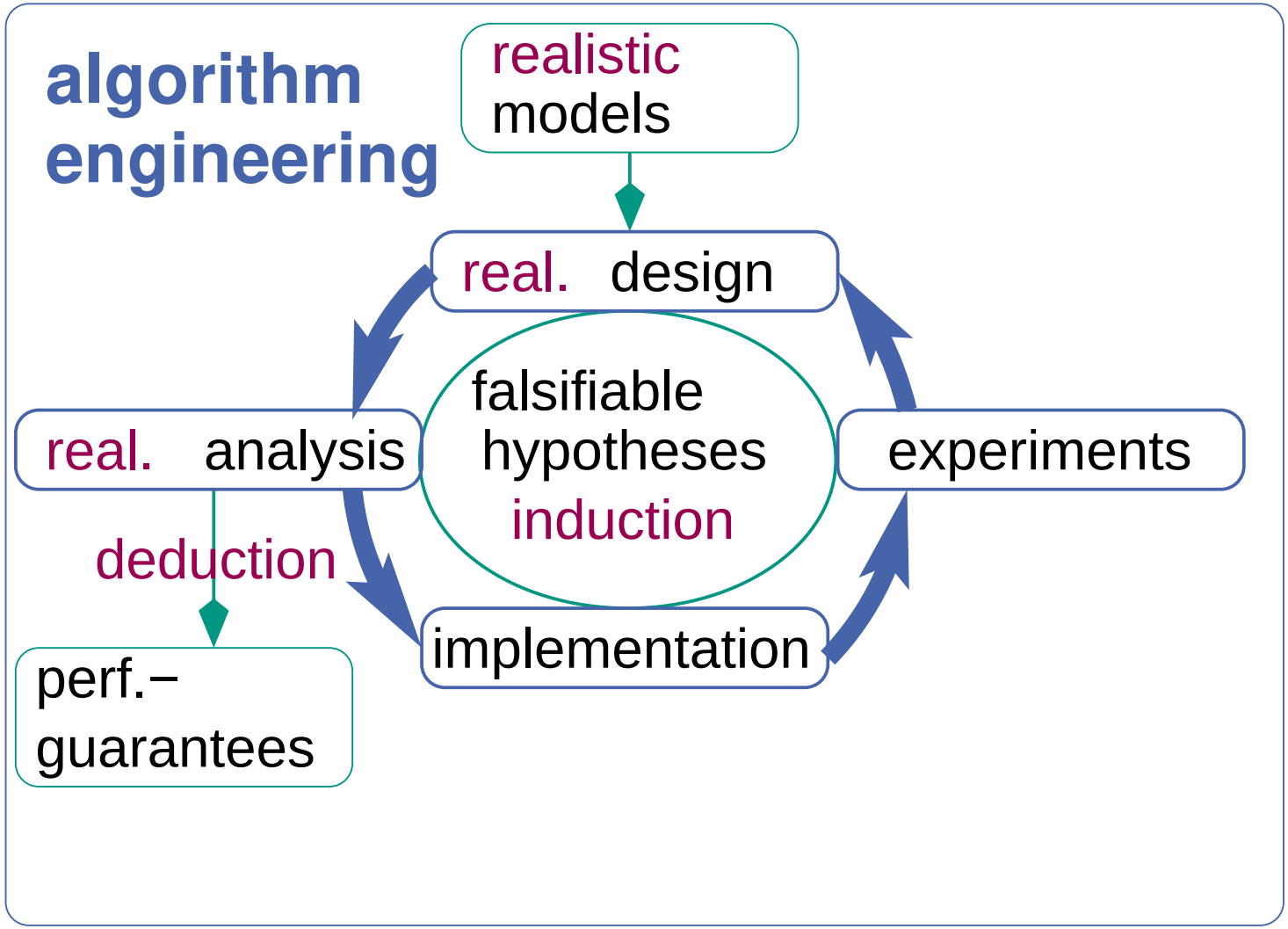
Algorithmics as Algorithm Engineering



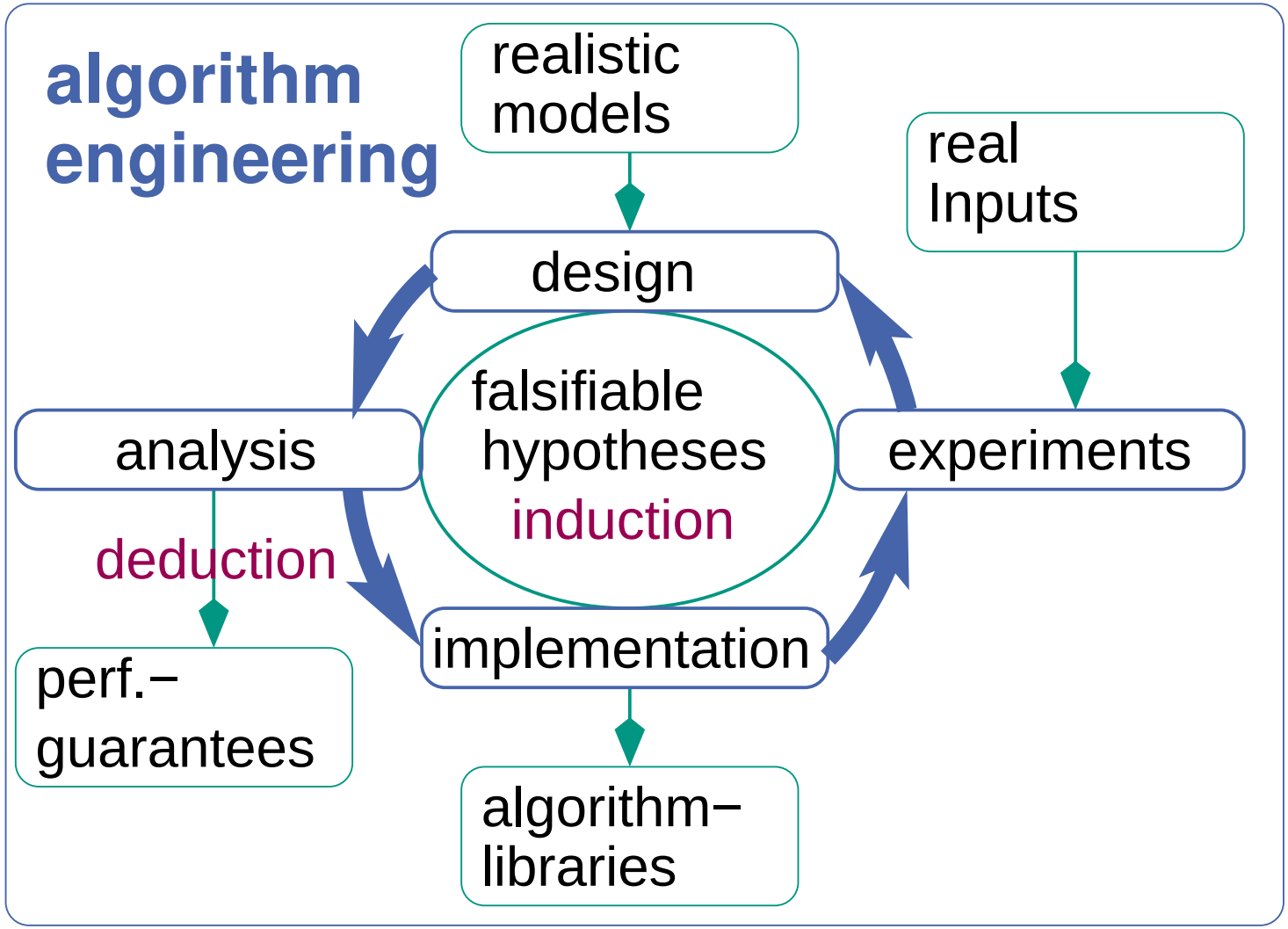
Algorithmics as Algorithm Engineering



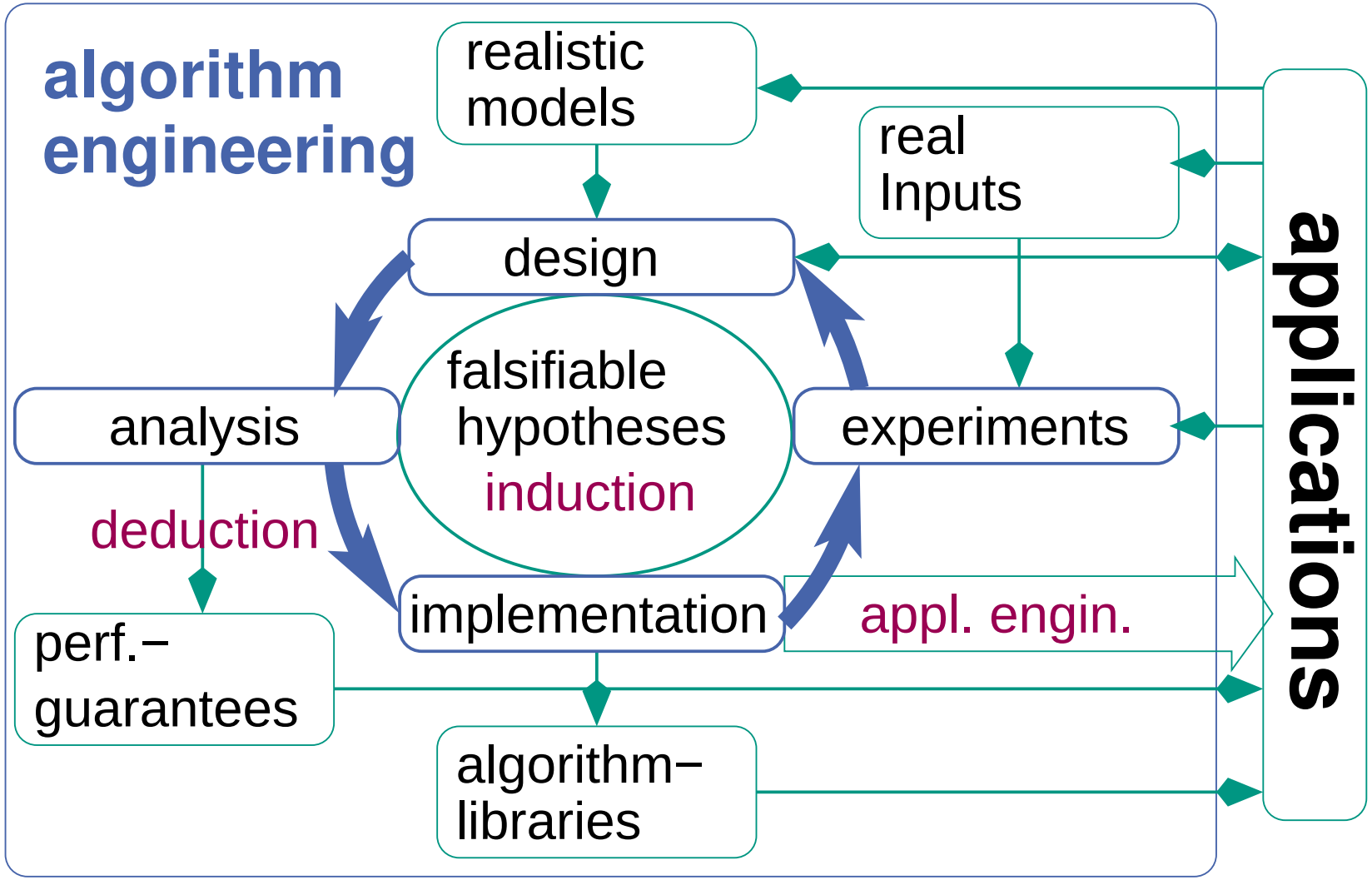
Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering



Bits of History

1843– Algorithms in theory and practice

1950s,1960s Still infancy

1970s,1980s Paper and pencil algorithm theory.

Exceptions exist, e.g., [D. Johnson], [J. Bentley]

1986 Term used by [T. Beth],

lecture "Algorithmentechnik" in Karlsruhe.

1988– Library of Efficient Data Types and Algorithms (LEDA) [K. Mehlhorn]

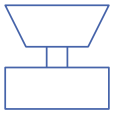
1997– Workshop on Algorithm Engineering

$\rightsquigarrow$ ESA applied track [G. Italiano]

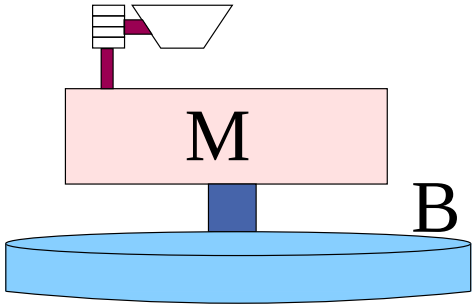
1997 Term used in US policy paper [Aho, Johnson, Karp, et. al]

1998 Alex workshop in Italy $\rightsquigarrow$ ALENEX

Realistic Models

Theory	$\longleftrightarrow$	**Practice**
simple 	**appl. model**	 complex
simple 	**machine model**	 real

- Careful refinements
- Try to preserve (partial) analyzability / simple results



Design

of algorithms that work well in practice

- simplicity
- reuse
- constant factors
- exploit easy instances

Analysis

- Constant factors matter

  Beispiel: quicksort

- Beyond worst case analysis

- Practical algorithms might be difficult to analyze

  (randomization, meta heuristics,. . .)

Implementation

sanity check for algorithms !

Challenges

Semantic gaps:

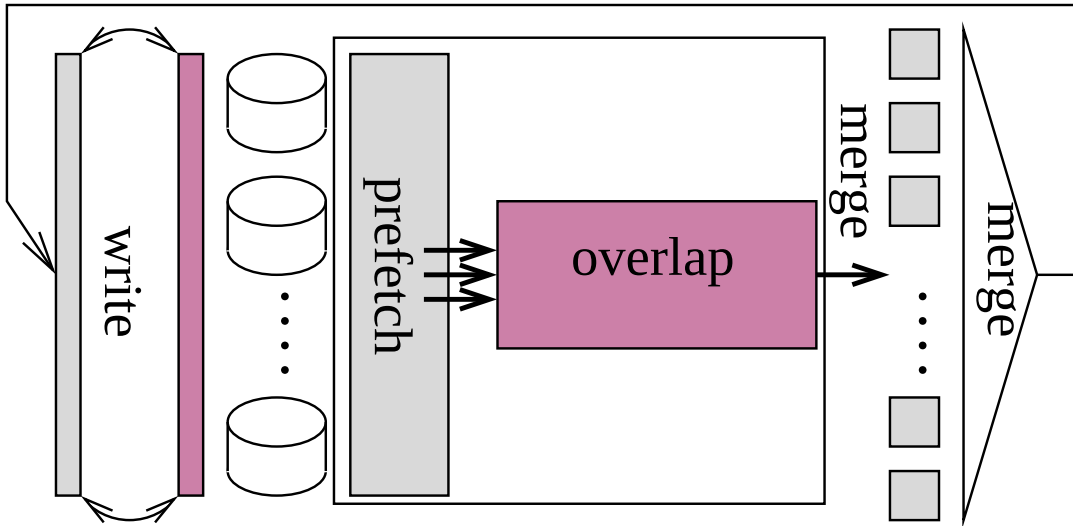
Abstract algorithm

$\leftrightarrow$

C++...

$\leftrightarrow$

hardware



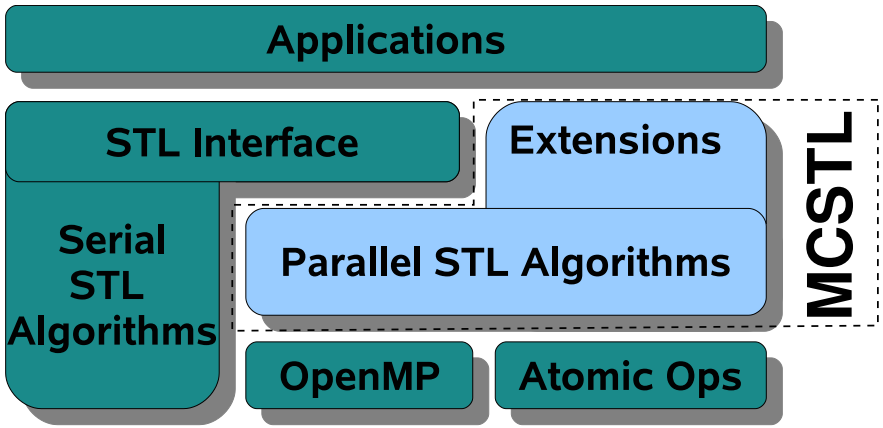
Experiments

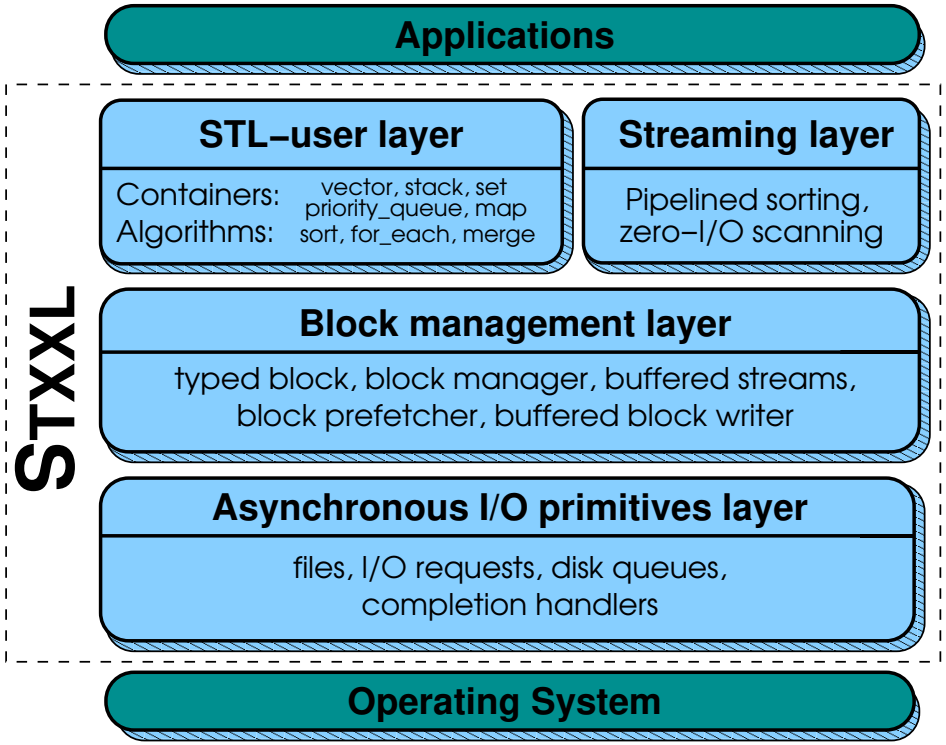
- sometimes a good surrogate for analysis
- too much rather than too little output data
- reproducibility (10 years!)
- software engineering

Stay tuned.

Algorithm Libraries — Challenges

- software engineering , e.g. CGAL
- standardization, e.g. java.util, C++ STL and BOOST
- performance ↔ generality ↔ simplicity
- applications are a priori unknown
- result checking, verification

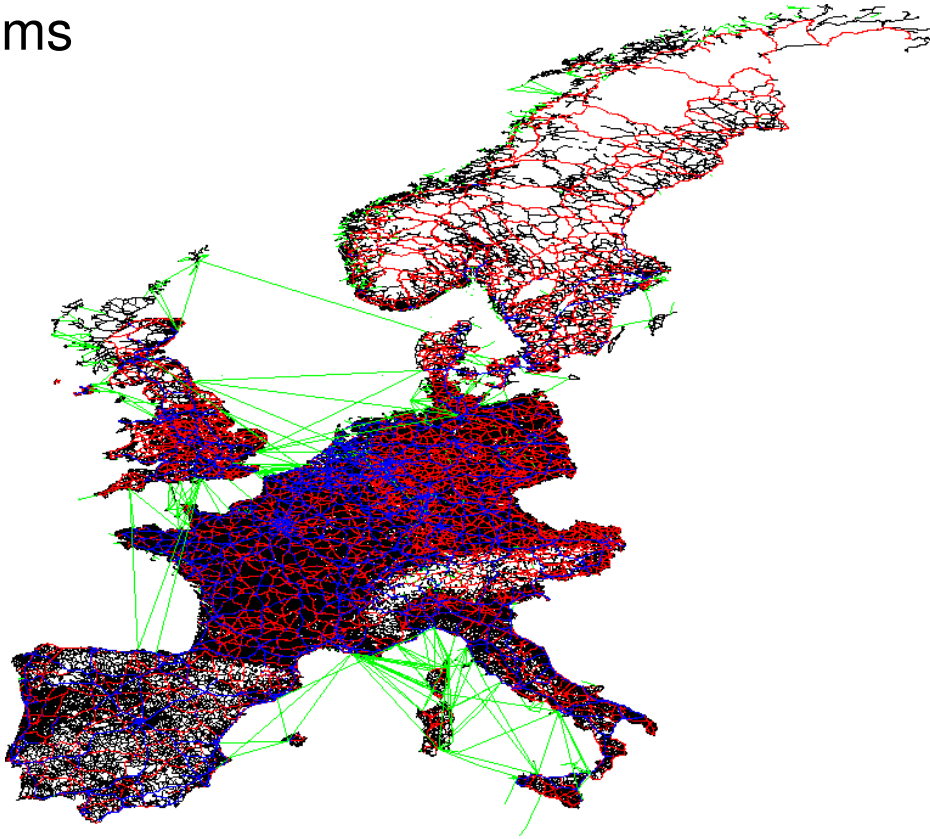




Problem Instances

Benchmark instances for NP-hard problems

- TSP
- Steiner-Tree
- SAT
- set covering
- graph partitioning
- ...



have proved essential for development of practical algorithms

Strange: much less real world instances for polynomial problems (MST, shortest path, max flow, matching...)

Example: Sorting Benchmark (Indy)

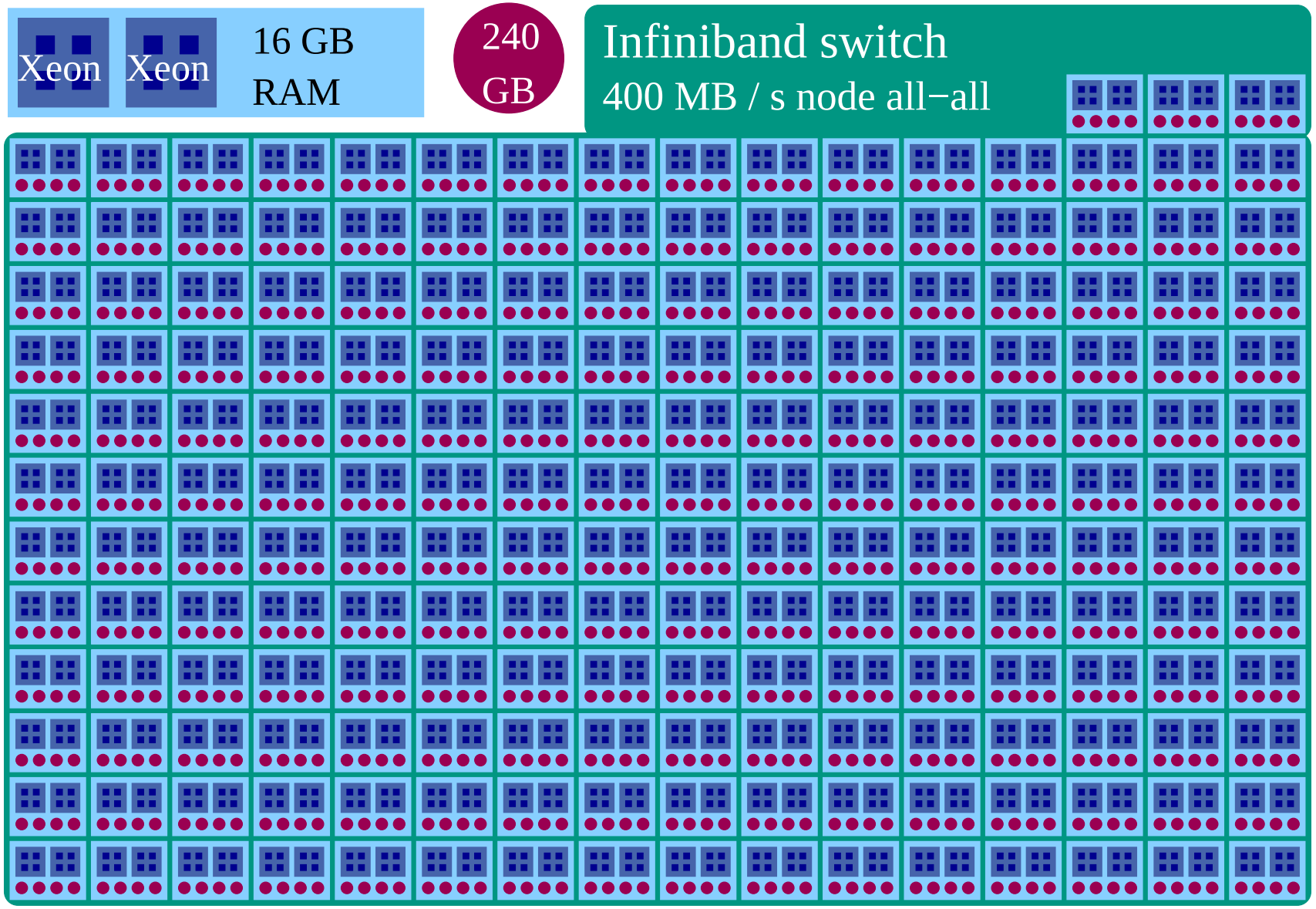
100 byte records, 10 byte random keys, with file I/O

Category	data volume	performance	improvement
GraySort	100 000 GB	564 GB / min	$17\times$
MinuteSort	955 GB	955 GB / min	$> 10\times$
JouleSort	100 000 GB	3 400 Recs/Joule	$???\times$
JouleSort	1 000 GB	17 500 Recs/Joule	$5.1\times$
JouleSort	100 GB	39 800 Recs/Joule	$3.4\times$
JouleSort	10 GB	43 500 Recs/Joule	$5.7\times$

Also: PennySort

Karlsruhe Institute of Technology

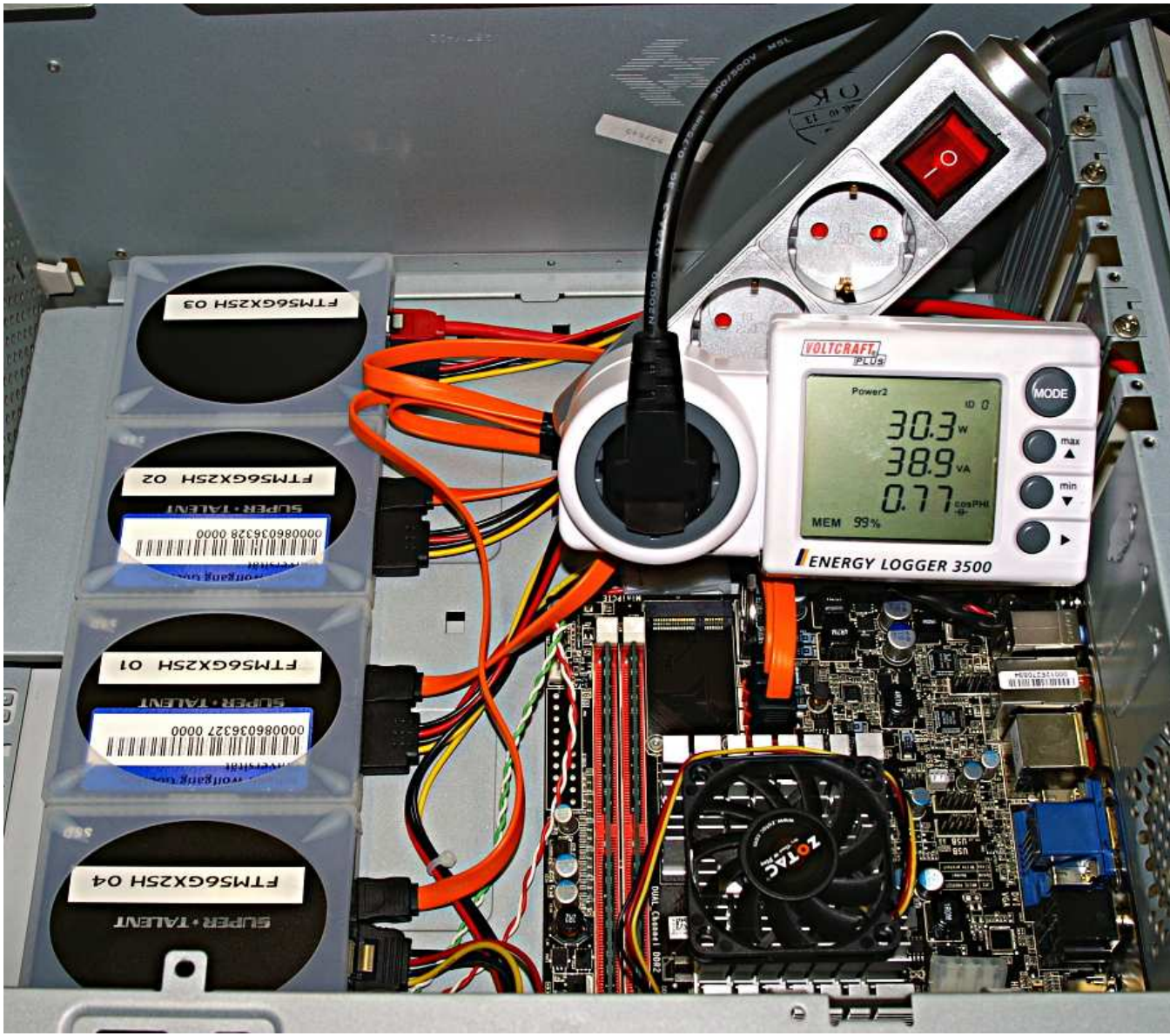
GraySort: inplace multiway mergesort, exact splitting



JouleSort

- Intel Atom N330
- 4 GB RAM
- 4×256 GB SSD (SuperTalent)

Algorithm similar to GraySort



Applications that "Change the World"

Algorithmics has the potential to SHAPE applications

(not just the other way round) [G. Myers]

Bioinformatics: sequencing, proteomics, phylogenetic trees,. . .

Information Retrieval: Searching, ranking,. . .

Traffic Planning: navigation, flow optimization,

adaptive toll, disruption management

Geographic Information Systems: agriculture, environmental protection,

disaster management, tourism,. . .

Communication Networks: mobile, P2P, cloud, selfish users,. . .

Conclusion:

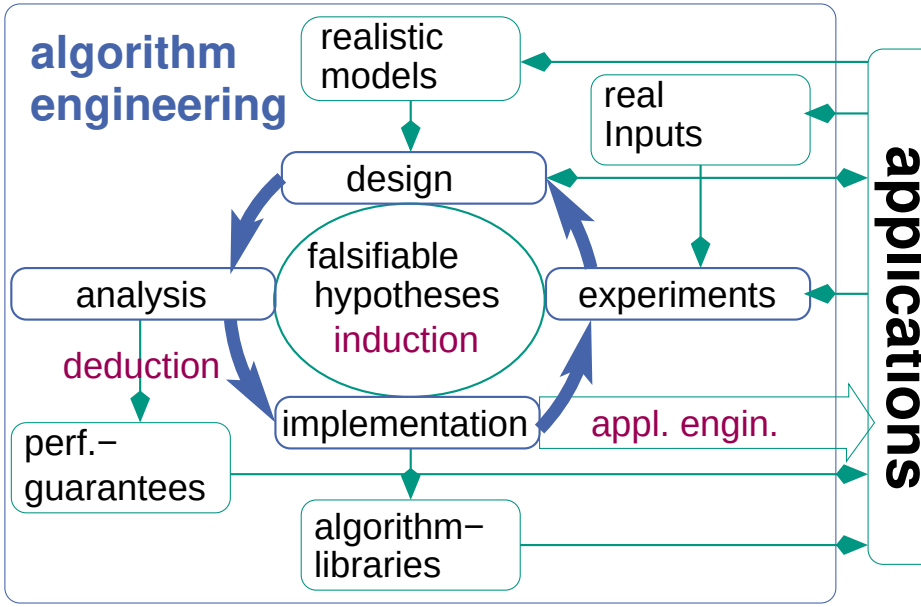
Algorithm Engineering $\leftrightarrow$ Algorithm Theory

- algorithm engineering is a wider view on algorithmics
  (but no revolution. None of the ingredients is really new)
- rich methodology
- better coupling to applications
- experimental algorithmics $\ll$ algorithm engineering
- algorithm theory $\subset$ algorithm engineering
- sometimes different theoretical questions
- algorithm theory may still yield the strongest, deepest and most persistent results within algorithm engineering

More On Experimental Methodology

Scientific Method:

- Experiment need a possible outcome that falsifies a hypothesis
- Reproducible
  - keep data/code for at least 10 years
    + documentation (aka laboratory journal (Laborbuch))
  - clear and detailed
    description in papers / TRs
  - share instances and code



Quality Criteria

- Beat the state of the art, globally – (not your own toy codes or the toy codes used in your community!)
- Clearly demonstrate this !
  - both codes use same data ideally from accepted benchmarks (not just your favorite data!)
  - comparable machines or fair (conservative) scaling
  - Avoid uncomparabilities like:
    “Yeah we have worse quality but are twice as fast”
  - real world data wherever possible
  - as much different inputs as possible
  - its fine if you are better just on some (important) inputs

Not Here but Important

- describing the setup
- finding sources of measurement errors
- reducing measurement errors (averaging, median,unloaded machine...)
- measurements in the creative phase of experimental algorithmics.

The Starting Point

- (Several) Algorithm(s)
- A few quantities to be measured: time, space, solution quality, comparisons, cache faults,... There may also be measurement errors.
- An unlimited number of potential inputs. $\rightsquigarrow$ condense to a few characteristic ones (size, $|V|$, $|E|$, ... or problem instances from applications)

Usually there is not a lack but an abundance of data $\neq$ many other sciences

The Process

Waterfall model?

1. Design
2. Measurement
3. Interpretation

Perhaps the paper should at least look like that.

The Process

- Eventually stop asking questions (Advisors/Referees listen !)
- build measurement tools
- automate (re)measurements
- Choice of Experiments driven by risk and opportunity
- Distinguish mode

  explorative: many different parameter settings, interactive, short turnaround times

  consolidating: many large instances, standardized measurement conditions, batch mode, many machines

Of Risks and Opportunities

Example: Hypothesis = my algorithm is the best

big risk: untried main competitor

small risk: tuning of a subroutine that takes 20 % of the time.

big opportunity: use algorithm for a new application

$\rightsquigarrow$ new input instances

Überblick

Sichtweisen für randomisierte Algorithmen

Erweiterung des Begriffs eines deterministischen Algorithmus:

- neuer *algorithmischer Elementarschritt*
  Beschaffung eines zufälligen Wertes
  RANDINT($c : \mathbb{N}$) liefere
  zufällig gleichverteilt eine ganze Zahl $x \in \{0, 1, \ldots, c-1\}$

alternative Sichtweisen (nicht in dieser Vorlesung)

- „quantifizierter Nichtdeterminismus“
  mit (schönen!) Wahrscheinlichkeiten
- zufällige Auswahl eines deterministischen Algorithmus

Fundamentale Änderung

Mehrere Ausführungen

des *gleichen* randomisierten Algorithmus
für die *gleiche* Eingabe

können verschieden sein!

Beispiel: Randomisierter Quicksort

RANDQS(s) mit $s = \langle e_0, \ldots, e_{n-1} \rangle$

1 **if** $n \leq 1$
2 **then** return s
3 $i \leftarrow$ RANDINT(n); $pvt \leftarrow e_i$
4 $a \leftarrow \langle e \in s : e < pvt \rangle$
5 $b \leftarrow \langle e \in s : e = pvt \rangle$
6 $c \leftarrow \langle e \in s : e > pvt \rangle$
7 **return** CONCAT(RANDQS(a), b, RANDQS(c))

Zufallsvariablen überall

Selbst für eine einzelne festgehaltene Eingabe sind

- die benötigte Laufzeit
- der benötigte Speicherplatz
- das berechnete Ergebnis

Zufallsvariablen.

Erinnerung an W-Theorie

- Ω Menge der *Elementarereignisse*
- $E \subseteq 2^\Omega$ Menge von *Ereignissen* (σ-Algebra)
  - *diskrete* σ-Algebra: $E = 2^\Omega$ und $|\Omega| \leq |\mathbb{N}|$
- *W-Maß* $\mathbf{Pr} : E \to$ (σ-additiv, $\mathbf{Pr}[\Omega] = 1$)
- *Zufallsvariable* $X : \Omega \to \mathbb{R}$ mit Eigenschaften …
  (erfüllt, falls σ-Algebra diskret)
- Schreibweisen $\mathbf{Pr}[X \leq x]$ statt $\mathbf{Pr}[\{\omega \mid X(\omega) \leq x\}]$
  $\mathbf{Pr}[X = x]$ statt $\mathbf{Pr}[\{\omega \mid X(\omega) = x\}]$

Standardbeispiel: Würfeln

- $\Omega = \{\boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}, \boxed{5}, \boxed{6}\}$
- $E = 2^{\Omega}$ diskret
  - z. B. $g = \{\boxed{2}, \boxed{4}, \boxed{6}\}$ «gerade Zahl gewürfelt»
- «fairer» Würfel:
  $\forall \omega \in \Omega : \mathbf{Pr}[\omega] = 1/6$
  - $\mathbf{Pr}[g] = 1/2$
  - allgemein: $\mathbf{Pr}[e] = |e|/6$
- $X(\omega) = \begin{cases} 0, & \text{falls } \omega \text{ Produkt zweier Primfaktoren} \\ 1, & \text{sonst} \end{cases}$
  - z. B. $\mathbf{Pr}[X = 1] = 2/3$

Erinnerung an W-Theorie (2)

- Zufallsvariable $X : \Omega \to \mathbb{R}$, nur abzählbar viele $X(\omega) \neq 0$
- *Erwartungswert* von X: $\mathbf{E}[X] = \sum_{x \in \mathbb{R}} x \cdot \mathbf{Pr}[X = x]$

  (falls das absolut konvergiert)
- immer: $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$

- *unabhängige* ZV: für alle $x, y \in \mathbb{R}$:
  $\mathbf{Pr}[X = x \wedge Y = y] = \mathbf{Pr}[X = x] \cdot \mathbf{Pr}[Y = y]$
- falls X, Y unabhängig: $\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$

- *Indikator-ZV:* 0 und 1 einzig mögliche Funktionswerte

Standardbeispiel: Würfeln

- $\Omega = \{\boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}, \boxed{5}, \boxed{6}\}$
- $E = 2^{\Omega}$
- Indikator-ZV $X(\omega) = 1$, falls ω prim
- Indikator-ZV $Y(\omega) = 1$, falls ω gerade
- X und Y *nicht* unabhängig:

$$\mathbf{Pr}[X = 1] = 1/2$$
$$\mathbf{Pr}[Y = 1] = 1/2$$
$$\mathbf{Pr}[X = 1] \cdot \mathbf{Pr}[Y = 1] = 1/4$$
$$\mathbf{Pr}[X = 1 \wedge Y = 1] = 1/6$$

Beispiel: Ausführung randomisierter Algorithmen

zu randomisiertem Algorithmus R und Eingabe x

- $\Omega = \{$«Programmlauf» von R für Eingabe $x\}$
  - Programmlauf: Folge der durchlaufenen globalen Speicherzustände
- $E = 2^{\Omega}$
- Beispiele möglicher Zufallsvariablen:
  - X(Prog.lauf) = Anzahl Schritte
  - X(Prog.lauf) = Ausgabe

Algorithmen mit unbekannter Laufzeit

- Will man das?
- «Wie unbekannt?»
  - *Quantifizierung?*

Algorithmen, die «variierende Ausgaben» liefern

- vielleicht erträglich bei Optimierungsproblemen
  - *Quantifizierung?*
- Erzeugung «zufälliger Objekte»
  - *Eigenschaften?*

Algorithmen, die «falsche Ausgaben» liefern

- Will man das?
- Schlägt in diesem Hörsaal gleich ein Meteorit ein?
- Ist $2^{400} - 593$ die größte Primzahl kleiner als 2^{400}?
- Soll die Ampel jetzt auf grün schalten?

- *Fehlerwahrscheinlichkeit?*

Vorteile randomisierter Algorithmen

Sie sind manchmal

- leichter zu formulieren und zu implementieren
- «schneller»
- «besser»
- die einzige Möglichkeit

Vorteile randomisierter Algorithmen

Sie sind manchmal

- leichter zu formulieren und zu implementieren
- «schneller»
- «besser»
- die einzige Möglichkeit

Aber: Man zahlt einen Preis!

Überblick

Motivation

auch wenn sie unrealistisch ist …

- gegeben: Listen $\langle e_1, \ldots, e_n \rangle$ und $\langle a_1, \ldots, a_n \rangle$
  - die Werte seien Elemente eines Körpers $\mathbb{F}$
  - z. B. Ein- und Ausgabe eines (Sortier?)Algorithmus
- Frage: Beinhalten die Listen die gleichen Elemente?
- naiver Algorithmus: quadratische Laufzeit

Polynome

- definiere $e(z) = \prod_{i=1}^{n}(z - e_i)$ und $a(z) = \prod_{i=1}^{n}(z - a_i)$
  - Ist $e(z) - a(z)$ das Nullpolynom?

Polynome

- definiere $e(z) = \prod_{i=1}^{n}(z - e_i)$ und $a(z) = \prod_{i=1}^{n}(z - a_i)$
  - Ist $e(z) - a(z)$ das Nullpolynom?
- Algorithmus

$c \leftarrow$ zufällig gleichverteilt $\in \mathbb{F}$
$y \leftarrow e(c) - a(c)$
if $y = 0$
  then return «$e = a$»
  else return «$e \neq a$»

Polynome

- definiere $e(z) = \prod_{i=1}^{n}(z - e_i)$ und $a(z) = \prod_{i=1}^{n}(z - a_i)$
  - Ist $e(z) - a(z)$ das Nullpolynom?
- Algorithmus

  $c \leftarrow$ zufällig gleichverteilt $\in \mathbb{F}$
  $y \leftarrow e(c) - a(c)$
  if $y = 0$
  then return «$e = a$»
  else return «$e \neq a$»

- Fehlerwahrscheinlichkeit
  $\mathbf{Pr}\left[e(c) - a(c) = 0 \wedge e(z) - a(z) \text{ nicht Nullpolynom}\right] \leq \frac{n}{|\mathbb{F}|}$

Ausblick: polynomial identity testing

Verallgemeinerung auf Polynome mit mehreren Veränderlichen

- gegeben: $P(z_1, \ldots, z_n)$ und $Q(z_1, \ldots, z_n)$
  Frage: Ist $P = Q$?
- dafür gibt es Anwendungen
- Idee von eben anpassbar
- kein deterministischer Polynomialzeit-Algorithmus bekannt!
  - man multipliziere $\prod_{i=1}^{n-1}(z_i + z_{i+1})$ aus … oder nicht …

Überblick

Randomisierter Quicksort

RANDQS(s) mit $s = \langle e_0, \ldots, e_{n-1} \rangle$, e_i paarweise verschieden

1 **if** $n \leq 1$
2 **then** return s
3 $i \leftarrow$ RANDINT(n); $pvt \leftarrow e_i$
4 $a \leftarrow \langle e \in s : e < pvt \rangle$
5 $b \leftarrow \langle e \in s : e = pvt \rangle$
6 $c \leftarrow \langle e \in s : e > pvt \rangle$
7 **return** CONCAT(RANDQS(a), b, RANDQS(c))

randQS: Anzahl Vergleiche

Erwartungswert (aus Algorithmen 1)

- es sei $\textsc{randQS}(e_1, \ldots, e_n) = (a_1, \ldots, a_n)$
- Laufzeit-Analyse nutzt

$$X_{ij} = \begin{cases} 1, & \text{falls } a_i \text{ und } a_j \text{ miteinander verglichen} \\ 0, & \text{sonst} \end{cases}$$

- Anzahl Vergleiche $= \mathbf{E}\left[\sum_{i<j} X_{ij}\right]$:

$$\mathbf{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbf{E}\left[X_{ij}\right]$$

$$= \sum_{i<j} \mathbf{Pr}\left[X_{ij} = 1\right] = \sum_{i<j} \frac{2}{j-i+1} \leq 2n \ln n$$

randQS: Anzahl Vergleiche (2)

mit hoher Wahrscheinlichkeit wie klein?

- Anzahl Vergleiche immer zwischen $n \ln n$ und n^2
- erwartete Laufzeit in $\Theta(n \ln n)$.
- zufrieden?

randQS: Anzahl Vergleiche (2)

mit hoher Wahrscheinlichkeit wie klein?

- Anzahl Vergleiche immer zwischen $n \ln n$ und n^2
- erwartete Laufzeit in $\Theta(n \ln n)$.
- zufrieden?
- Wahrscheinlichkeit für Laufzeit «nahe am Erwartungswert» z. B. $\leq 28n \ln n$?

randQS: Anzahl Vergleiche mit hoher Wkt.

- e_i beliebiges Element
- L_j Liste auf Rekursionsstufe j, die e_i enthält
- Zufallsvariable Y_j

$$Y_j = \begin{cases} 1, & \text{falls } \frac{1}{4}|L_j| \leq |L_{j+1}| \leq \frac{3}{4}|L_j| \\ 0, & \text{sonst} \end{cases}$$

- $\mathbf{E}\left[Y_j\right] = \mathbf{Pr}\left[Y_j = 1\right] = \frac{1}{2}$
- die Y_j sind unabhängig voneinander

randQS: Anzahl Vergleiche mit hoher Wkt.

- $Y_j = 1 \quad$ falls $\frac{1}{4}|L_j| \leq |L_{j+1}| \leq \frac{3}{4}|L_j|$
- nenne Rekursionsstufe *erfolgreich*, falls $Y_j = 1$
- wenn r von k Stufen erfolgreich, dann $|L_k| \leq \left(\frac{3}{4}\right)^r n$
- setze $r = 3.5 \ln n$, denn

$$r = 3.5 \ln n \implies \ln n \leq r \ln \frac{4}{3} \implies n \leq \left(\frac{4}{3}\right)^r$$

$$\implies \left(\frac{3}{4}\right)^r n \leq 1 \implies |L_k| \leq 1$$

- nach r erfolgreichen Stufen ist e_i in einelementiger Liste

randQS: Anzahl Vergleiche mit hoher Wkt.

- sei nun $k = 8r = 28 \ln n$
- $Y = \sum_{j=1}^{k} Y_j$, $\mathbf{E}[Y] = k/2$
- e_i höchstens dann *nicht* in einelementiger Liste, falls $Y < r$
- Wie groß ist
  $\mathbf{Pr}[Y < r] = \mathbf{Pr}\left[Y < \frac{k}{8}\right] = \mathbf{Pr}\left[Y < (1 - \frac{3}{4})\mathbf{E}[Y]\right]$?
- gleich *Chernoff-Schranken*
  liefern z. B. $\mathbf{Pr}[Y < r] \leq e^{-\frac{k}{4}\cdot(\frac{3}{4})^2} = n^{-63/16}$
- nach k Rekursionsstufen
  - Wkt. ein bestimmtes e_i nicht in Einerliste: $\leq n^{-63/16}$
  - Wkt. mindestens ein e_i nicht in Einerliste:
    $\leq n \cdot n^{-63/16} = n^{-47/16}$

randQS: Anzahl Vergleiche mit hoher Wkt.

- $k = 8r = 28 \ln n$
- nach k Rekursionsstufen
  - Wkt. ein bestimmtes e_i nicht in Einerliste: $\leq n^{-63/16}$
  - Wkt. mindestens ein e_i nicht in Einerliste: $\leq n \cdot n^{-63/16} = n^{-47/16}$
- mit Wkt. $1 - n^{-47/16}$ nach $k = 28 \ln n$ Rekursionsstufen *alle* Elemente in Einerlisten
- mit Wkt. $1 - n^{-47/16}$ reichen $28n \ln n$ Vergleiche

Überblick

Einfache Schranken

Markov- und Chebyshev-Ungleichung

- *Markov-Ungleichung:* ZV $Y \geq 0$, Erwartungswert μ_Y.
  Dann gilt für alle $t, k \in \mathbb{R}_+$:

$$\mathbf{Pr}\left[Y \geq t\right] \leq \frac{\mu_Y}{t} \quad \text{bzw.} \quad \mathbf{Pr}\left[Y \geq k\mu_Y\right] \leq \frac{1}{k}$$

- *Chebyshev-Ungleichung:* ZV X mit
  Erwartungswert μ_X, Standardabweichung σ_X.
  Dann gilt für alle $t \in \mathbb{R}_+$:

$$\mathbf{Pr}\left[|X - \mu_X| \geq t\sigma_X\right] \leq \frac{1}{t^2} \quad \text{bzw.} \quad \mathbf{Pr}\left[|X - \mu_X| \geq t\right] \leq \frac{\sigma_X^2}{t^2}$$

Chernoff-Schranken

Es seien

- $X_1, \ldots, X_n$ *unabhängige 0-1-ZV* mit $\mathbf{Pr}[X_i = 1] = p_i$
- $X = \sum_{i=1}^{n} X_i$ und $\mu = \mathbf{E}[X] = \sum p_i$

Dann gilt

- für alle $0 \leq \delta$:

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}$$

- für alle $1 > \delta \geq 0$ also $0 < 1 - \delta \leq 1$:

$$\mathbf{Pr}[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}} \right)^{\mu}$$

Chernoff-Schranken: Beweis von Teil 1

- sei t positiv; Markov-Ungleichung liefert:

$$\mathbf{Pr}\left[X \geq (1+\delta)\mu\right] = \mathbf{Pr}\left[e^{tX} \geq e^{t(1+\delta)\mu}\right] \leq \frac{\mathbf{E}\left[e^{tX}\right]}{e^{t(1+\delta)\mu}}$$

- mit den X_i sind auch die e^{tX_i} unabhängig:

$$\mathbf{E}\left[e^{tX}\right] = \mathbf{E}\left[e^{t\sum X_i}\right] = \mathbf{E}\left[\prod e^{tX_i}\right] = \prod \mathbf{E}\left[e^{tX_i}\right]$$

- $\mathbf{E}\left[e^{tX_i}\right] = p_i \cdot e^t + (1-p_i)\cdot 1 = 1 + p_i(e^t - 1) \leq e^{p_i(e^t-1)}$
- $\mathbf{Pr}\left[X \geq (1+\delta)\mu\right] \leq \dfrac{\prod e^{p_i(e^t-1)}}{e^{t(1+\delta)\mu}} = \dfrac{e^{\sum p_i(e^t-1)}}{e^{t(1+\delta)\mu}}$

$$= \frac{e^{\mu(e^t-1)}}{e^{t(1+\delta)\mu}} = \left(\frac{e^{(e^t-1)}}{e^{t(1+\delta)}}\right)^{\mu}$$

- wähle $t = \ln(1+\delta)$ (positiv!)

Chernoff-Schranken: Vereinfachungen

- statt $\left(\frac{e^{\delta}}{(1\pm\delta)^{(1\pm\delta)}}\right)^{\mu}$ betrachte $f(\delta) = \delta - (1 \pm \delta)\ln(1 \pm \delta)$
- je nach Bereich, aus dem δ stammt,
  kann man $f(\delta)$ nach oben abschätzen in der Form $-\delta^2/c$

Chernoff-Schranken: Korollare

obige Abschätzungen manchmal etwas unhandlich
diverse Vereinfachungen, unter anderem:

- Für $1 > \delta \geq 0$ gilt:

$$\mathbf{Pr}\left[X \leq (1-\delta)\mu\right] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} \leq e^{-\delta^2\mu/2}$$

Überblick

Und-Oder-Bäume

T_k vollständiger binärer Baum der Höhe $2k$

- innere Knoten abwechselnd mit "$\wedge$" und "$\vee$" markiert
- Wurzel von T_1 ist $\wedge$-Knoten mit zwei $\vee$-Knoten als Nachfolger
- Jeder dieser Knoten hat zwei Blätter als Nachfolger.
- T_k entsteht aus T_1, indem die Blätter durch Kopien von T_{k-1} ersetzt werden

- T_k hat $n = 4^k$ Blätter
- im folgenden $x_1, \ldots, x_{4^k}$ genannt

Auswertung von Und-Oder-Bäumen

- boolesche Werte an allen Blättern
  $\rightsquigarrow$ boolscher Wert für Wurzel (und andere innere Knoten)
  - (für die naheliegende Definition von Auswertung …)
- Problem:
  - gegeben: Werte $x_1, \ldots, x_n$ an den Blättern
  - gesucht: Wert $T_k(x_1 \ldots, x_n)$ der Wurzel
- Berechnung des Wurzelwertes „bottom up“ durch
  - Besuch aller $n = 4^k$ Blätter und
  - Berechnung der Werte aller inneren Knoten möglich
- **Frage:** Geht es besser?

Satz

Für jeden *deterministischen* Alg. A und jedes $k \geq 1$ gilt:

- Es gibt eine Folge $x_1, \ldots, x_n$ von Bits so, dass A bei Berechnung von $T_k(x_1, \ldots, x_n)$ *alle* $n = 4^k$ Blätter besucht.

Satz

Für jeden *deterministischen* Alg. A und jedes $k \geq 1$ gilt:

- Es gibt eine Folge $x_1, \ldots, x_n$ von Bits so, dass A bei Berechnung von $T_k(x_1, \ldots, x_n)$ *alle* $n = 4^k$ Blätter besucht.
- Wert der Wurzel = Wert des zuletzt besuchten Blattes
- sowohl Wurzelwert $= 0$ als auch Wurzelwert $= 1$ kann erzwungen werden

Beweis

Induktion über k

- Induktionsanfang $k = 1$:
  - A muss mindestens ein Blatt besuchen, o. B. d. A. x_1.
  - setze $x_1 = 0 \quad \leadsto$ kein innerer Wert festgelegt
  - A muss ein weiteres Blatt besuchen.
  - O. B. d. A. zwei Möglichkeiten:
    1. Zweites besuchtes Blatt ist x_2. Setze $x_2 = 1$.
       Damit nur Wert des $\vee$-Knotens klar, Wurzelwert nicht.
       A muss weiteres Blatt besuchen, o. B. d. A. x_3.
       Setze $x_3 = 0$. A muss x_4 besuchen $\leadsto$ Wurzelwert.
    2. Zweites besuchtes Blatt ist x_3. Setze $x_3 = 0$.
       Kein innerer Wert festgelegt
       A muss weiteres Blatt besuchen, o. B. d. A. x_2.
       Setze $x_2 = 1$. A muss x_4 besuchen $\leadsto$ Wurzelwert

Beweis (2)

- Induktionsschritt $k-1 \rightsquigarrow k$:
  - Fasse T_k als T_1-Baum auf, dessen Blätter durch T_{k-1}-Bäume ersetzt sind.
  - Bezeichne die „Blätter“ von T_1 mit $y_1, \ldots, y_4$.
  - Analog Induktionsanfang kann durch geeignete Wahl der y_i erzwungen werden, dass A *alle 4* Werte ermitteln muss.
  - Induktionsvoraussetzung: für jeden T_{k-1}-Baum gibt es Blattwerte, die gewünschtes y_i liefern und erzwingen, dass A alle darunter liegenden Blätter besuchen muss.
  - Also muss A in diesem Fall alle Blätter überhaupt besuchen.

Zwischenüberlegung

- Jede Liste $x_1, \ldots, x_{4^k}$ der Länge $4^k = n$
  enthält eine Teilfolge $y_1, \ldots, y_{2^k}$ der Länge $2^k = \sqrt{n}$,
  die schon den Wurzelwert festlegt!
  - Beweis: Induktion …
- aber deterministisch manchmal *alle* Knoten benötigt
- «Die y_i sind schwer zu finden.»

- Ausweg Randomisierung … ? …

Algorithmus: randomisierte UOB-Auswertung

Algorithmus: randomisierte UOB-Auswertung

```
proc AndNodeEval(T)
if IsLeaf(T) then
        return value(T) fi
⟨andernfalls:⟩
T' ← ⟨zufälliger T-U.baum⟩
r ← OrNodeEval(T')
if r = 0 then
        return 0
else
        T'' ← ⟨and. T-U.baum⟩
        return OrNodeEval(T'')
fi
```

```
proc OrNodeEval(T)


T' ← ⟨zufälliger T-U.baum⟩
r ← AndNodeEval(T')
if r = 1 then
        return 1
else
        T'' ← ⟨and. T-U.baum⟩
        return AndNodeEval(T'')
fi
```

```
AndNodeEval(root)
```

Satz

Für die randomisierte Auswertung von UOB gilt:
Für *jede Folge* $x_1, \ldots, x_{4^k}$
ist Erwartungswert für die Anzahl besuchter Blätter
$\leq 3^k = n^{\log_4 3} \approx n^{0.792\ldots}$.

(Das ist übrigens beweisbar optimal.)

Beweis

Induktion

- E: ein Erwartungswert für die Anzahl besuchter Blätter
- Induktionsanfang $k = 1$: Überprüfen aller 16 möglichen Kombinationen $x_1, \ldots, x_4$. Z. B. 0100 (Rest analog):
  1. Falls erst linker Teilbaum: gleich wahrscheinlich wird
     erst und nur die 1 oder
     erst die 0 und dann die 1 besucht,
     anschließend im rechten Teilbaum beide Blätter.
     $E = 1/2 \cdot 1 + 1/2 \cdot 2 + 2 = 7/2$.
  2. Falls erst rechter Teilbaum; nach Besuch beider Blätter klar: der T_1-Baum den liefert Wert 0.
     $E = 2$.

  Beide Fälle gleich wahrscheinlich, also insgesamt
  $E = 1/2 \cdot 7/2 + 1/2 \cdot 2 = 11/4 < 3 \ (= 3^1 = 3^k)$.

Beweis (2)

Induktionsschritt $k-1 \rightsquigarrow k$:

- Betrachte zunächst $\vee$-Knoten mit zwei T_{k-1}-Bäumen darunter. Zwei Fälle:

  O1. $\vee$-Knoten wird 1 liefern:

  - Dann muss mindestens ein T_{k-1}-Baum dies auch tun.
  - Mit Wahrscheinlichkeit $p \geq 1/2$ wird ein Unterbaum untersucht, der 1 liefert.
  - Mit Wahrscheinlichkeit $1-p \leq 1/2$ werden beide Unterbäume untersucht.
  - $E \leq p \cdot 3^{k-1} + (1-p) \cdot 2 \cdot 3^{k-1} = (2-p) \cdot 3^{k-1} \leq 3/2 \cdot 3^{k-1}$.

  O2. Der $\vee$-Knoten wird 0 liefern:

  - Dann müssen beide T_{k-1}-Bäume dies auch tun.
  - Nach Induktionsvoraussetzung $E \leq 2 \cdot 3^{k-1}$ ist.

Beweis (3)

Induktionsschritt $k-1 \leadsto k$: Teil 2a

- Betrachte Wurzel des T_k-Baumes mit zwei eben untersuchten Bäumen darunter. Zwei Fälle:
  U1. Der $\wedge$-Knoten wird 0 liefern:
  - Dann muss mindestens ein U.baum dies auch tun.
  - Mit Wahrscheinlichkeit $p \geq 1/2$ wird ein Unterbaum untersucht, der 0 liefert.
  - Mit Wahrscheinlichkeit $1-p \leq 1/2$ werden beide Unterbäume untersucht.
  - Gemäß O1 und O2 ist folglich

$$\begin{aligned} E &\leq p \cdot 2 \cdot 3^{k-1} + (1-p) \cdot (3/2 \cdot 3^{k-1} + 2 \cdot 3^{k-1}) \\ &= 7/2 \cdot 3^{k-1} - p \cdot 3/2 \cdot 3^{k-1} \\ &\leq 11/4 \cdot 3^{k-1} \\ &\leq 3^k \end{aligned}$$

Beweis (4)

Induktionsschritt $k-1 \rightsquigarrow k$: Teil 2b

- Betrachte Wurzel des T_k-Baumes mit zwei eben untersuchten Bäumen darunter. Fall:
  - U2. Der $\wedge$-Knoten wird 1 liefern:
    - Dann müssen beide Unterbäume dies auch tun.
    - Gemäß O1 ist daher $E \leq 2 \cdot 3/2 \cdot 3^{k-1} \leq 3^k$.

Überblick

Erdős-Rényi-Zufallsgraphen

$G(n, p)$

1 $V \leftarrow \{1, \ldots, n\}$
2 $E \leftarrow \{\}$
3 **for** $i \leftarrow 1$ **to** $n - 1$ **do**
4 **for** $j \leftarrow i + 1$ **to** n **do**
5 with probability p: add edge $\{i, j\}$ to E
6 **return** (V, E)

mitunter: p von n abhängig, z. B. $p = p(n) = \frac{\ln n}{n}$

ER-Graphen: einfache Beobachtungen

- Wkt. für bestimmten Graphen mit n Knoten und m Kanten: $p^m(1-p)^{\binom{n}{2}-m}$
  - $p = 1/2$, n fest: alle Graphen gleichwahrscheinlich
- erwartete Anzahl Kanten: $p\binom{n}{2}$
- erwarteter Knotengrad: $p(n-1)$
- Wkt. g_d für Knotengrad d
  - $\binom{n-1}{d}$ mögliche «Nachbarmengen»
  - jede passiert mit Wkt. $p^d(1-p)^{n-1-d}$
  - insgesamt $g_d = \binom{n-1}{d}p^d(1-p)^{n-1-d}$
    Binomialverteilung

Manchmal interessieren sehr große n — und asymptotische Eigenschaften

- mitunter interessant: $n \to \infty$
- *monotone increasing property* $\mathcal{P}$
  - Hinzufügen von Kanten erhält die Eigenschaft
  - nichttrivial (leerer Graph: nein, vollständiger Graph: ja)
- $p^*(n)$ ist *Grenzfunktion*, falls
  $\lim_{n\to\infty} \mathbf{Pr}\left[G(n,p) \text{ hat } \mathcal{P}\right]$
  $$= \begin{cases} 0, & \text{falls } \lim_{n\to\infty} p(n)/p^*(n) = 0 \\ 1, & \text{falls } \lim_{n\to\infty} p(n)/p^*(n) = \infty \end{cases}$$
- *«Phasenübergang»*
  - passiert für jede monotone increasing property

Manchmal interessieren sehr große n — und asymptotische Eigenschaften

- $p^*(n)$ ist *scharfe* Grenzfunktion, falls für jedes $\varepsilon > 0$ gilt:

$$\lim_{n\to\infty} \mathbf{Pr}\left[G(n,p) \text{ hat } \mathcal{P}\right] = \begin{cases} 0, & \text{falls } p(n)/p^*(n) \leq 1-\varepsilon \\ 1, & \text{falls } p(n)/p^*(n) \geq 1+\varepsilon \end{cases}$$

- existiert *nicht* für jede monotone increasing property

ER-Graphen: Durchmesser ≤ 2

- $\mathcal{D}$=«hat Durchmesser ≤ 2» ist monotone increasing property
- **Satz** $p(n) = \sqrt{2}\sqrt{\frac{\ln n}{n}}$ ist scharfe Grenzfunktion für $\mathcal{D}$.
- $X_{ij} = \begin{cases} 1, & \text{falls Abstand zwischen } i \text{ und } j \text{ größer 2} \\ 0, & \text{sonst} \end{cases}$
- betrachte $X = \sum X_{ij}$
  - Durchmesser ≤ 2 genau dann, wenn $X = 0$
- $\lim_{n\to\infty} \mathbf{E}[X] = ?$
- $\mathbf{E}[X] = \binom{n}{2}(1-p)(1-p^2)^{n-2}$

ER-Graphen: Durchmesser ≤ 2

- $\mathbf{E}[X] = \binom{n}{2}(1-p)(1-p^2)^{n-2}$
- setze $p = c\sqrt{\frac{\ln n}{n}}$
- dann
$$\mathbf{E}[X] = \frac{n(n-1)}{2}\left(1 - c\sqrt{\frac{\ln n}{n}}\right)\left(1 - c^2\frac{\ln n}{n}\right)^{n-2}$$
$$\leq \frac{n^2}{2}\left(1 - \frac{c^2 \ln n}{n}\right)^n$$
$$\lim_{n\to\infty} \mathbf{E}[X] = \frac{n^2}{2}e^{-c^2 \ln n} = \frac{1}{2}n^{2-c^2}$$
- $c > \sqrt{2} \implies \lim_{n\to\infty} \mathbf{E}[X] = 0$ $\quad$ ($c < \sqrt{2}$ nicht hier …)

Zusammenhangskomponenten

Satz von Erdős- und Rényi

$t(n) = \frac{\ln n}{n}$ ist Grenzfunktion für Abwesenheit isolierter Knoten:

- Wenn $\lim_{n\to\infty} p(n)/t(n) = 0$,
  dann geht Wkt. für Existenz isolierter Knoten gegen 1
- $\lim_{n\to\infty} p(n)/t(n) \to \infty$,
  dann hat ER-Graph mit hoher Wkt. keine isolierten Knoten
  und sogar nur eine Zusammenhangskomponente.

Zusammenhangskomponenten

weitere Ergebnisse

- wenn $np < 1$,
  dann fast nie Zhk. größer als $O(\log n)$
- wenn $np = 1$,
  dann fast immer Zhk. größer als $O(\log n)$
- wenn $np \to c > 1$,
  dann fast immer eindeutige riesige Zhk., alle anderen nur $O(\log n)$ groß
- wenn $np > (1 + \varepsilon) \ln n \, (\varepsilon > 0)$,
  dann fast immer eine Zhk. der Größe n

Erwartet konstanter Knotengrad

- $p(n-1) = c$ konstant $\rightsquigarrow p(n) = \frac{c}{n-1}$
- dann für $n \to \infty$

$$
\begin{aligned}
g_d &= \binom{n-1}{d} p^d (1-p)^{n-1-d} \\
&= \frac{(n-1)!}{(n-1-d)!d!} \left(\frac{c}{n-1}\right)^d \left(1 - \frac{c}{n-1}\right)^{n-1-d} \\
&\approx \frac{(n-1-d)! \prod_{i=1}^{d}(n-i)}{(n-1-d)!(n-1)^d} \cdot \frac{c^d}{d!} \cdot e^{-c} \\
&\approx \frac{c^d}{d!} e^{-c} \qquad \text{Poisson-Verteilung}
\end{aligned}
$$

- große Graphen aus der Realität haben andere Verteilung …

Andere Anforderungen?

- z. B. so genannte skalenfreie Netzwerke «scale-free graphs»
- *Skalenfreiheit:* Anteil der Knoten mit Grad d «etwa proportional» zu $d^{-\alpha}$ für ein $\alpha > 0$
  - in Graphen aus der Realität beobachtbar
  - ER-Graphen im allgemeinen *nicht* skalenfrei
- Barabási-Albert Modell: $\alpha = 3$

Barabási-Albert Modell

- beginne mit einem «kleinen» zusammenhängenden Graphen G_{m_0} mit m_0 Knoten
- wähle $m < m_0$
- aus G_i erzeuge Graphen G_{i+1} so
  - $V_{i+1} = V_i \cup \{v_{i+1}\}$ neuer Knoten
  - wähle zufällig m Kanten $\{v_{i+1}, v_j\}$ mit $j \leq i$ wobei v_j gewählt mit Wkt. proportional zu seinem Grad in G_i
  - «preferential attachment»

Überblick

Suchprobleme

- für jede Probleminstanz $I \in M_I$ gibt es Menge $M_S(I)$ möglicher Lösungen $S \in M_S(I)$
- *Zielfunktion* $f : \bigcup_{I \in M_I} M_S(I) \to \mathbb{R}_+$
- *Minimierungsproblem:* für jedes $I \in M_I$ existiere
  - $f^*(I) = \min\{f(S) \mid S \in M_S(I)\}$
  - gegeben: I
  - gesucht: S mit $f(S) = f^*(I)$
- *Maximierungspoblem:* analog

Approximation bei Suchproblemen

- Lösungen S mit $f(S) = f^*(I)$ sind oft schwer zu finden
  - z. B. **NP**-schwer

Auswege:

- naiv: alle möglichen Lösungen betrachten
  - oft zu aufwändig
- ad-hoc Heuristiken
  - Qualität der Antwort eventuell unklar
- *Approximationsalgorithmus* A
  - $f(A(I))$ «möglichst nahe an» $f^*(I)$
  - *Optimierungsaufgabe*
  - gerne polynomielle Laufzeit
  - mit Garantie für Lösungsgüte

Approximation bei «Zählproblemen»

- für jede Probleminstanz $I \in M_I$ gibt es Menge $M_S(I)$ möglicher Lösungen $S \in M_S(I)$
- *gesucht:* Algorithmus A
  - mit $A(I)$ «möglichst nahe an» $|M_S(I)|$,
  - der die Anzahl der Lösungen annähert
  - Beispiel: bestimme die Anzahl perfekter Matchings eines bipartiten Graphen

- nicht in dieser Vorlesung

Turing-Reduzierbarkeit

Das kam in «Theoretische Grundlagen der Informatik» vor:

- Suchproblem Π **NP**-*schwer* oder **NP**-*hart*,
  falls ein **NP**-vollständiges Entscheidungsproblem L
  existiert mit $L \leq_T^p \Pi$.
- das heißt:
  - es gibt *Orakel-Turingmaschine* für L
  - mit Orakel für Π,
    - jede Befragung des Orakels braucht 1 Schritt
  - die polynomielle Laufzeit hat,
- Turing-Reduktion in Polynomialzeit
- das heißt:
  wenn Π in Polynomialzeit lösbar, dann auch L

Überblick

Job Scheduling: Aufgabenstellung

Problem: *Job Shop Scheduling* oder *Makespan Scheduling*

- *Maschinen* $M_1, \ldots, M_m$
- *Jobs* $J_1, \ldots, J_n$
  - Job J_j benötigt Zeit $t_j \geq 0$
- *Lösung* $S : \{1, \ldots, n\} \longrightarrow \{1, \ldots m\}$
- Last von Maschine i: $L_i = \sum_{S(j)=i} t_j$
- Zielfunktion: minimiere *Makespan* $L_{\max} = \max_i L_i$
  - Wann ist die letzte Maschine fertig?
- einfacher Fall
  - identische Maschinen
  - unabhängige Jobs
- Problem ist **NP**-hart

naheliegender Algorithmus: LISTSCHEDULING

naheliegender Algorithmus: LISTSCHEDULING

```
1  LISTSCHEDULING(n, m, t_{1...n})
2  each L_i ← 0                          ▷ load of machine i, 1 ≤ i ≤ m
3  each S_j ← 0                          ▷ machine for job j, 1 ≤ j ≤ n
4  for each j in range(1, n)
5       do pick k from {i | L_i is currently minimal }
6          S_j ← k
7          L_k ← L_k + t_j
8  return S
```

im folgenden kurz *LS* statt LISTSCHEDULING

Eigenschaften des Algorithmus (1)

Lemma

Es sei $T = \sum_j t_j$. Für jede Lösung S des Algorithmus gilt:
Wenn J_ℓ ein Job ist, der zuletzt fertig wird, dann ist

$$L_{\max} \leq \frac{T}{m} + \frac{m-1}{m} t_\ell$$

Beweis

$t = L_{\max} - t_\ell$ Startzeitpunkt von J_ℓ

Eigenschaften des Algorithmus (1)

Lemma

Es sei $T = \sum_j t_j$. Für jede Lösung S des Algorithmus gilt:
Wenn J_ℓ ein Job ist, der zuletzt fertig wird, dann ist

$$L_{\max} \leq \frac{T}{m} + \frac{m-1}{m} t_\ell$$

Beweis

$t = L_{\max} - t_\ell$ Startzeitpunkt von J_ℓ

- bis hierhin *alle* beschäftigt, mit Jobs $\neq J_\ell$,
  also $mt \leq T - t_\ell$, also

$$L_{\max} = t + t_\ell \leq \frac{T}{m} - \frac{t_\ell}{m} + t_\ell$$

Approximationsfaktor

Approximationsalgorithmus A
für Minimierungsproblem mit Zielfunktion f erzielt
Approximationsfaktor ρ,
falls für alle Probleminstanzen $I \in M_i$ gilt:

$$\frac{f(A(I))}{f^*(I)} \leq \rho$$

$\rho = 1$: A liefert stets eine optimale Lösung

Eigenschaften des Algorithmus (2)

Satz

LS erzielt Aproximationsfaktor $\rho \leq 2 - \frac{1}{m}$.

Beweis

- stets: $f^*(I) \geq T/m$ und für jedes j: $f^*(I) \geq t_j$
- sei I beliebig:

$$\frac{f(LS(I))}{f^*(I)} \leq \frac{T/m}{f^*(I)} + \frac{m-1}{m}\frac{t_\ell}{f^*(I)}$$
$$\leq \frac{T/m}{T/m} + \frac{m-1}{m}\frac{t_\ell}{t_\ell} = 2 - \frac{1}{m}$$

□

Eigenschaften des Algorithmus (3)

Lemma

LS erzielt Aproximationsfaktor $\rho \geq 2 - \frac{1}{m}$.

Beweis

Eigenschaften des Algorithmus (3)

Lemma

LS erzielt Aproximationsfaktor $\rho \geq 2 - \frac{1}{m}$.

Beweis

Probleminstanzen bei denen

- $f(LS(I)) = 2m - 1$ und $f^*(I) = m$:
  - $m(m-1)$ Jobs mit $t_1 = \cdots = t_{n-1} = 1$
  - letzter Job mit $t_n = m$
- LS: alle m Maschinen bis $t = m - 1$ nur Minijobs, dann der große $\quad \rightsquigarrow L_{\max} = m - 1 + m = 2m - 1$
- optimal: eine Maschine nur den großen Job, $m - 1$ machen je m Minijobs $\quad \rightsquigarrow L_{\max} = m$ □

Überblick

Erinnerung: TSP-Suchproblem

- Probleminstanz
  - vollständiger Graph $G = (V, E = V \times V)$ und
  - Längenfunktion $c : E \to \mathbb{Z}_+$
- für Permutation π von V:

$$f(\pi) = \sum_{i=1}^{n-1} c(\pi(i), \pi(i+1)) + c(\pi(n), \pi(1))$$

- $f^*(G, c) = \min_\pi f(\pi)$
- gesucht: Permutation π mit minimalem $f(\pi) = f^*(G, c)$

TSP-a-Approximations-Suchproblem

gegeben: $a \geq 1$

- Probleminstanz
  - vollständiger Graph $G = (V, E = V \times V)$ und
  - Längenfunktion $c : E \longrightarrow \mathbb{Z}_+$
- für Permutation π von V:

$$f(\pi) = \sum_{i=1}^{n-1} c(\pi(i), \pi(i+1)) + c(\pi(n), \pi(1))$$

- gesucht: Permutation π mit $f(\pi) \leq a \cdot f^*(G, c)$

- kurz a-APPROXTSP

Erinnerung: Hamiltonkreis (Entscheidungsproblem)

- Probleminstanz:
  - Graph $G = (V, E)$
- Frage: Gibt es einen Hamiltonkreis in G?
  - i. e. Permutation π so, dass $(\pi(1), \pi(2), \pi(n), \pi(1))$ Kreis

- HAMILTONIANCIRCUIT ist **NP**-vollständig

Schwere Approximierbarkeit des TSP

Satz

Für jede Konstante $a \geq 1$ ist das
TSP-a-Approximations-Suchproblem **NP**-hart.

Beweisidee

Schwere Approximierbarkeit des TSP

Satz

Für jede Konstante $a \geq 1$ ist das
TSP-a-Approximations-Suchproblem **NP**-hart.

Beweisidee

zeige:

- wenn a-APPROXTSP in Polynomialzeit lösbar,
- dann auch HAMILTONIANCIRCUIT

Schwere Approximierbarkeit des TSP: Konstruktion

- gegeben: $G = (V, E)$ mit $n = |V|$
- definiere: $G' = (V, V \times V)$ und Längenfunktion c

Schwere Approximierbarkeit des TSP: Konstruktion

- gegeben: $G = (V, E)$ mit $n = |V|$
- definiere: $G' = (V, V \times V)$ und Längenfunktion c mit

$$c(u, v) = \begin{cases} 1, & \text{falls } (u, v) \in E \\ an + 1, & \text{sonst} \end{cases}$$

- sei AATSP Algorithmus für a-APPROXTSP Suchproblem

Schwere Approximierbarkeit des TSP: Konstruktion

- gegeben: $G = (V, E)$ mit $n = |V|$
- definiere: $G' = (V, V \times V)$ und Längenfunktion c mit

$$c(u, v) = \begin{cases} 1, & \text{falls } (u, v) \in E \\ an + 1, & \text{sonst} \end{cases}$$

- sei AATSP Algorithmus für a-APPROXTSP Suchproblem
- wenn G Hamiltonkreis hat,
  dann hat G' Kreis der Länge n

Schwere Approximierbarkeit des TSP: Konstruktion

- gegeben: $G = (V, E)$ mit $n = |V|$
- definiere: $G' = (V, V \times V)$ und Längenfunktion c mit

$$c(u, v) = \begin{cases} 1, & \text{falls } (u, v) \in E \\ an + 1, & \text{sonst} \end{cases}$$

- sei AATSP Algorithmus für a-APPROXTSP Suchproblem
- wenn G Hamiltonkreis hat,
  dann hat G' Kreis der Länge n
  AATSP(G', c) liefert Lösung $\leq an$

Schwere Approximierbarkeit des TSP: Konstruktion

- gegeben: $G = (V, E)$ mit $n = |V|$
- definiere: $G' = (V, V \times V)$ und Längenfunktion c mit

$$c(u, v) = \begin{cases} 1, & \text{falls } (u, v) \in E \\ an + 1, & \text{sonst} \end{cases}$$

- sei AATSP Algorithmus für a-APPROXTSP Suchproblem
- wenn G Hamiltonkreis hat,
  dann hat G' Kreis der Länge n
  AATSP(G', c) liefert Lösung $\leq an$
- wenn G keinen Hamiltonkreis hat,
  dann hat jeder Kreis von G' Länge $\geq an + 1 + n - 1 > an$

□

Schwere Approximierbarkeit des TSP: Konstruktion

- gegeben: $G = (V, E)$ mit $n = |V|$
- definiere: $G' = (V, V \times V)$ und Längenfunktion c mit

$$c(u, v) = \begin{cases} 1, & \text{falls } (u, v) \in E \\ an + 1, & \text{sonst} \end{cases}$$

- sei AATSP Algorithmus für a-APPROXTSP Suchproblem
- wenn G Hamiltonkreis hat,
  dann hat G' Kreis der Länge n
  AATSP(G', c) liefert Lösung $\leq an$
- wenn G keinen Hamiltonkreis hat,
  dann hat jeder Kreis von G' Länge $\geq an + 1 + n - 1 > an$
  AATSP(G', c) liefert Lösung $> an$

□

Überblick

METRIC TSP

- wie bei TSP, aber zusätzlich wird verlangt, dass c der *Dreiecksungleichung* genügt:
  - für alle $x, y, z \in V$: $c(x, y) + c(y, z) \geq c(x, z)$

- *metrische «Vervollständigung»* $mv(G, c)$ von (G, c):
  $mv(G, c) = (G, c')$ mit

  $$c'(x, y) = \text{ Länge kürzester Wege von } x \text{ nach } y$$

  - «Vervollständigung» hier irreführendes Wort
  - woher kommt das wohl?

- Konstruktion eben:
  Dreiecksungleichung im allgemeinen *verletzt*

Satz

Für Instanzen des METRICTSP Suchproblems kann man in Polynomialzeit eine 2-Approximation berechnen.

Beweisidee

```
1  metricTSPApprox(G, c)
2  T ← MST(G, c)                          ▷ w(T) ≤ f*(G, c)
3  T' ← T with every edge doubled         ▷ w(T') ≤ 2f*(G, c)
4  T'' ← EULERTOUR(T')                    ▷ w(T'') ≤ 2f*(G, c)
5  S ← remove duplicate nodes from T''    ▷ w(S) ≤ 2f*(G, c)
6  return S
```

Überblick

Pseudopolynomielle Laufzeit

Wovon hängt die Laufzeit eines Algorithmus ab?

- Laufzeit $t(I)$ eines Algorithmus für eine Eingabe I abhängig von *Größe* $n(I)$ der Repräsentation von I

üblich:

- $n(I)$ ist Anzahl $n_2(I)$ der Bits um I binär zu codieren
  - Codierung von $k \in \mathbb{N}_+$ braucht $\Theta(\log_2 k)$ Bits
- *polynomielle Laufzeit* $t(n)$:
  exisitert Polynom $p(n)$ für alle I gilt: $t(I) \leq p(n_2(I))$

alternativ:

- $n(I)$ ist Anzahl $n_1(I)$ der Bits um I *unär* zu codieren
  - Codierung von $k \in \mathbb{N}_+$ braucht k Bits
- *pseudopolynomielle Laufzeit* $t(n)$:
  exisitert Polynom $p(n)$ für alle I gilt: $t(I) \leq p(n_1(I))$

Zwei kleine Warnungen

- Algorithmus mit pseudopolynomieller Laufzeit: tatsächliche Form der Eingabe?

- pseudopolynomielle Laufzeit nicht verwechseln mit *quasipolynomieller Laufzeit:*

$$t(n) = 2^{O((\log n)^c)}$$

für eine Konstante $c > 0$

  - Beispiel: $c = 2 \rightsquigarrow n^{\log n}$

KNAPSACK Suchproblem

Probleminstanzen

- endliche Menge $M = \{1, \ldots, n\}$ von *Gegenständen*
- *Maximalgröße* $W \in \mathbb{N}_+$ «Rucksackgröße»
- *Größen* $w_i \in \mathbb{N}_+$ o. B. d. A. jedes $w_i \leq W$
- *Profite* $p_i \in \mathbb{N}_+$

Lösungen

- Teilmenge $M' \subseteq M$ mit
- $w(M') = \sum_{i \in M'} w_i \leq W$

gesucht

- Teilmengen mit möglichst großem Profit
- Zielfunktion $f(M') = p(M') = \sum_{i \in M'} p_i$
- Maximierungsproblem o. B. d. A. $f^*(I) \geq \max p_i$

KNAPSACK: Codierungen der Eingabe

Teil T von I	$u_2(T)$	$u_1(T)$
$\{1, \ldots, n\}$	$\log n$	n
W	$\log W$	W
$\langle w_1, \ldots, w_n \rangle$	$n \log W$	nW
$\langle p_1, \ldots, p_n \rangle$	$n \log \hat{P}$	$n\hat{P}$
insgesamt	$\Theta(n \log W + n \log \hat{P})$	$\Theta(nW + n\hat{P})$

wobei $\hat{P} = \sum_i p_i$

Schwere des KNAPSACK Suchproblems

- **NP**-schwer
  - bei «normaler» Messung der Eingabegröße
  - also Polynomialzeit-Algorithmen unbekannt
- aber: pseudopolynomielle Laufzeit erreichbar
- solche Probleme heißen *schwach* **NP***-schwer*

KNAPSACK: pseudopolynomielle Laufzeit

mit dynamischer Programmierung

- es sei

$$C(i, P) = \min\{w(M') \mid M' \subseteq \{1, \ldots, i\} \wedge p(M') \geq P\}$$

  falls eine solche Teilmenge M' existiert, und ∞ sonst.

- Beobachtung:

$$C(1, P) = \begin{cases} w_1, & \text{falls } p_1 \geq P \\ \infty, & \text{sonst} \end{cases}$$

$$C(i+1, P) = \min(\, C(i, P),\ w_{i+1} + C(i, P - p_{i+1})\,)$$

KNAPSACK: pseudopolynomielle Laufzeit

```
1  dynProgKnapsack(n, W, ⟨w_1, ..., w_n⟩, ⟨p_1, ..., p_n⟩)
2  for P ← 1 to P̂ do
3      C(1, P) ← ...
4  for i ← 1 to n − 1 do
5      for P ← 1 to P̂ do
6          C(i + 1, P) ← min( C(i, P), w_{i+1} + C(i, P − p_{i+1}) )
7  return max{P | C(n, P) ≤ W}
```

Erweiterung

- speichere in $C(i, P)$ auch, welche der Objekte $1, \ldots, i$ benutzt
- Lösung: sogar konkrete Teilmenge/Bitvektor $\mathbf{x}$
- Skalarprodukt $\mathbf{p} \cdot \mathbf{x}$ ist maximaler Profit

Polynomielle Approximationsschemata

Polynomial Time Approximation Scheme (PTAS)

- es sei Π ein $\left\{\begin{matrix}\text{Minimierungs-}\\ \text{Maximierungs-}\end{matrix}\right\}$ Problem
- es sei $\mathcal{A}$ Algorithmus mit Paaren (I, ε) als Eingaben, wobei $\varepsilon > 0$ reell und $I \in \Pi_I$ ist

- $\mathcal{A}$ *polynomielles Approximationsschema (engl. PTAS)*, wenn für jedes $\varepsilon > 0$ ein Polynom $p(n)$ so existiert, dass für jede Instanz I gilt

$$f(\mathcal{A}(I, \varepsilon)) \begin{matrix}\leq\\ \geq\end{matrix} \begin{pmatrix}1+\varepsilon\\ 1-\varepsilon\end{pmatrix} \cdot f^*(I)$$

und Laufzeit $t(I, \varepsilon) \leq p(n_2(I))$

Voll polynomielle Approximationsschemata

Fully Polynomial Time Approximation Scheme (FPTAS)

- es sei Π ein $\left\{\begin{matrix}\text{Minimierungs-}\\ \text{Maximierungs-}\end{matrix}\right\}$ Problem
- es sei $\mathcal{A}$ mit Paaren (I, ε) als Eingaben, wobei $\varepsilon > 0$ reell und $I \in M_I$ ist

- $\mathcal{A}$ *voll polynomielles Approximationsschema (engl. PTAS)*, wenn ein Polynom $p(n, x)$ so existiert, dass für jede Instanz I und jedes $\varepsilon > 0$ gilt

$$f(\mathcal{A}(I, \varepsilon)) \begin{matrix}\leq\\ \geq\end{matrix} \begin{pmatrix}1+\varepsilon\\ 1-\varepsilon\end{pmatrix} \cdot f^*(I)$$

und Laufzeit $t(I, \varepsilon) \leq p(n_2(I), 1/\varepsilon)$

PTAS versus FPTAS

- jedes FPTAS ist ein PTAS
  - $(\frac{n}{\varepsilon})^5 + n^2 + \varepsilon^{-42}$
- aber nicht umgekehrt:
  PTAS- aber nicht FPTAS-Laufzeiten z. B.
  - $n + 2^{1/\varepsilon}$
  - $n^{1/\varepsilon}$
  - $2^{-\log n \log \varepsilon}$

Von pseudopolynomieller zu polynomieller Laufzeit

- $n_2(I) \in \Theta(n \log W + n \log \hat{P})$
- pseudopolynomielle Laufzeit $\in \Theta(n\hat{P})$

Von pseudopolynomieller zu polynomieller Laufzeit

- $n_2(I) \in \Theta(n \log W + n \log \hat{P})$
- pseudopolynomielle Laufzeit $\in \Theta(n\hat{P})$
- polynomielle Laufzeit,
  - wenn die Profite kleine Werte sind,
    z. B. für konstantes k,

$$\hat{P} \in O(n) \cdot 2^k$$

  - z. B. jedes $0 < p_i < 2^k$

FPTAS für KNAPSACK

Schreibweise $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$, etc.

```
1  EPSAPPROXKNAPSACK(ε, n, W, w, p)
2  P ← max_i p_i
3  K ← εP/n
4  each p'_i ← ⌊p_i/K⌋
5  x' ← DYNPROGKNAPSACK(n, W, w, p')
6  return x'
```

FPTAS für KNAPSACK

Schreibweisen

- $\mathbf{x}^*$ optimale Lösung für ursprüngliches $\mathbf{p}$
- $\mathbf{x}'$ optimale Lösung für $\mathbf{p}'$
- dann

$$\mathbf{p} \cdot \mathbf{x}^* = \sum_{i, x_i^* = 1} p_i = \text{ maximaler Profit des Originalproblems}$$

- *Frage:* Wie gut ist $\mathbf{p} \cdot \mathbf{x}'$ im Vergleich zu $\mathbf{p} \cdot \mathbf{x}^*$?

EPSAPPROXKNAPSACK ist FPTAS

Satz

EPSAPPROXKNAPSACK ist ein voll polynomielles Approximationsschema für KNAPSACK.

Lemma

Mit den Bezeichnungen wie bisher

$$\mathbf{p} \cdot \mathbf{x}' \geq (1 - \varepsilon)\mathbf{p} \cdot \mathbf{x}^*$$

EPSAPPROXKNAPSACK ist FPTAS: Approximation

- $K = \varepsilon P/n$ und $p'_i = \lfloor p_i/K \rfloor$, also $Kp'_i \leq p_i$

$$
\begin{aligned}
\mathbf{p} \cdot \mathbf{x}' \geq K\mathbf{p}' \cdot \mathbf{x}' &\geq K\mathbf{p}' \cdot \mathbf{x}^* && \text{weil } \mathbf{x}' \text{ optimal für } \mathbf{p}' \\
&= \sum_{i \in \mathbf{x}^*} K \left\lfloor \frac{p_i}{K} \right\rfloor \\
&\geq \sum_{i \in \mathbf{x}^*} K \left(\frac{p_i}{K} - 1 \right) && = \sum_{i \in \mathbf{x}^*} p_i - \sum_{i \in \mathbf{x}^*} K \\
&\geq \mathbf{p} \cdot \mathbf{x}^* - nK \\
&= \mathbf{p} \cdot \mathbf{x}^* - \varepsilon P \\
&\geq \mathbf{p} \cdot \mathbf{x}^* - \varepsilon \mathbf{p} \cdot \mathbf{x}^* && \text{weil Optimum } \geq \max p_i = P \\
&= (1 - \varepsilon)\mathbf{p} \cdot \mathbf{x}^*
\end{aligned}
$$

EPSAPPROXKNAPSACK ist FPTAS: Polynomialzeit

Lemma

Die Laufzeit von EPSAPPROXKNAPSACK ist in $O(n^3 \cdot \frac{1}{\varepsilon})$.

- dynamischen Programmierung für $\mathbf{p}'$-Problem dominiert: Laufzeit $n\hat{P}'$
- es ist (mit $P = \max_i p_i$)

$$n\hat{P}' = n \sum_i p'_i \leq n^2 \max_i p'_i = n^2 \left\lfloor \frac{P}{K} \right\rfloor = n^2 \left\lfloor \frac{Pn}{\varepsilon P} \right\rfloor \leq n^3 \cdot \frac{1}{\varepsilon}$$

Varianten von Approximation: zwei Beispiele

- bestes bekanntes FPTAS linear in n
- in der Praxis erreicht man «fast Linearzeit»

4 Stringology (Zeichenkettenalgorithmen)

- Strings sortieren
- Patterns suchen
  - Pattern vorverarbeiten
  - Text vorverarbeiten
    - Invertierte Indizes
    - Suffix Trees / Suffix Arrays
- Datenkompression
- Pattern suchen in komprimierten Indizes

Strings Sortieren

multikey quicksort / ternary quicksort

Function mkqSort(S : Sequence **of** String, $\ell : \mathbb{N}$) : Sequence **of** String

assert $\forall e, e' \in S : e[1..\ell-1] = e'[1..\ell-1]$

if $|S| \leq 1$ **then return** S // base case

pick $p \in S$ uniformly at random // pivot string

return concatenation of mkqSort($\langle e \in S : e[\ell] < p[\ell] \rangle, \ell$),
mkqSort($\langle e \in S : e[\ell] = p[\ell] \rangle, \ell+1$), and
mkqSort($\langle e \in S : e[\ell] > p[\ell] \rangle, \ell$)

- Laufzeit: $O(|S| \log |S| + \sum_{t \in s} |t|)$
- genauer: $O(|S| \log |S| + d)$ (d: Summe der eindeutigen Präfixe)
- Übung: in-place!

Strings Sortieren

S A A L
B I E N E
E H R E
H A U S
A R M
M I E T E
T A S S E
M O R D
H A N D
S E E
H U N D
H A L L E
N A C H T

Strings Sortieren

```
      SAAL
      BIENE
      EHRE
      HAUS
      ARM
      MIETE
      TASSE
      MORD
p     HAND
      SEE
      HUND
      HALLE
      NACHT
```

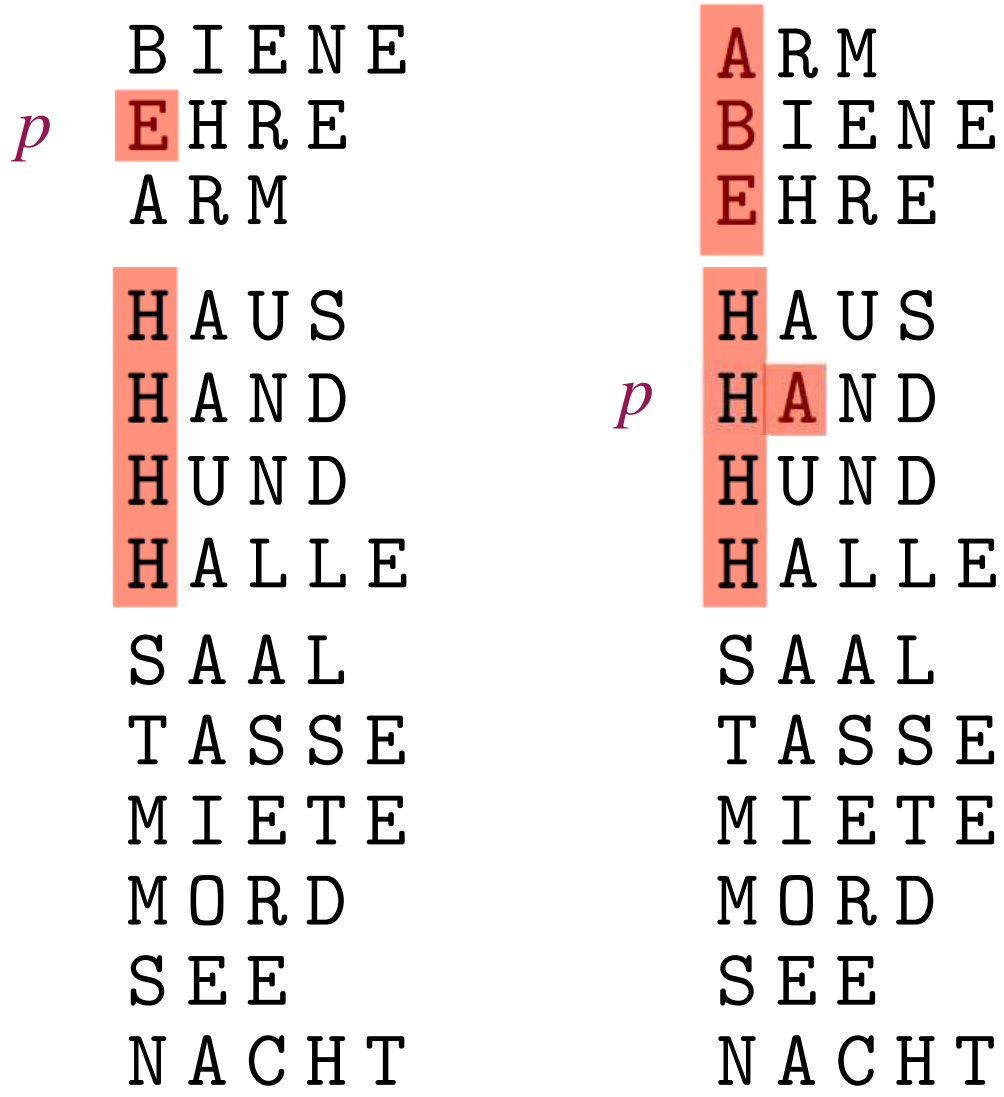
Strings Sortieren

BIENE
EHRE
ARM

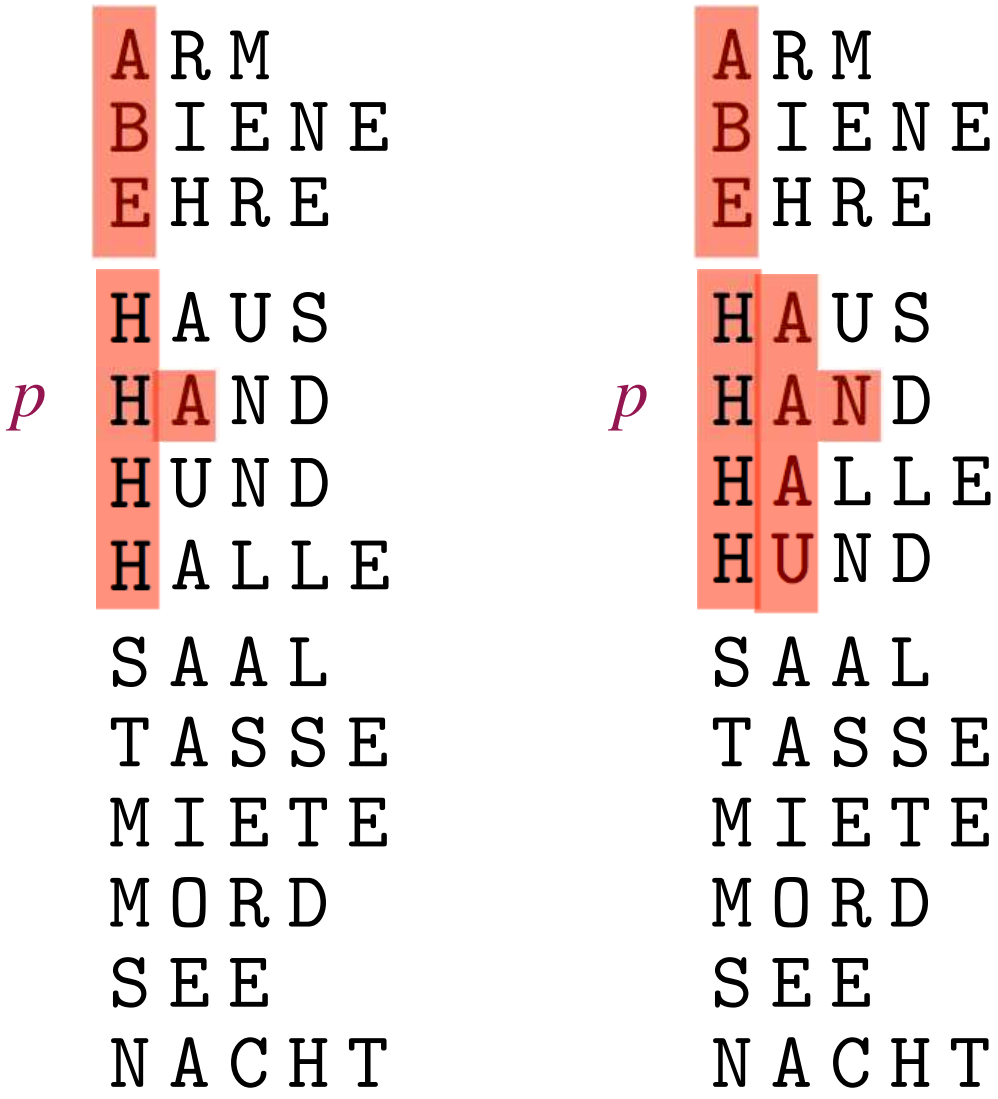
HAUS
HAND
HUND
HALLE

SAAL
TASSE
MIETE
MORD
SEE
NACHT

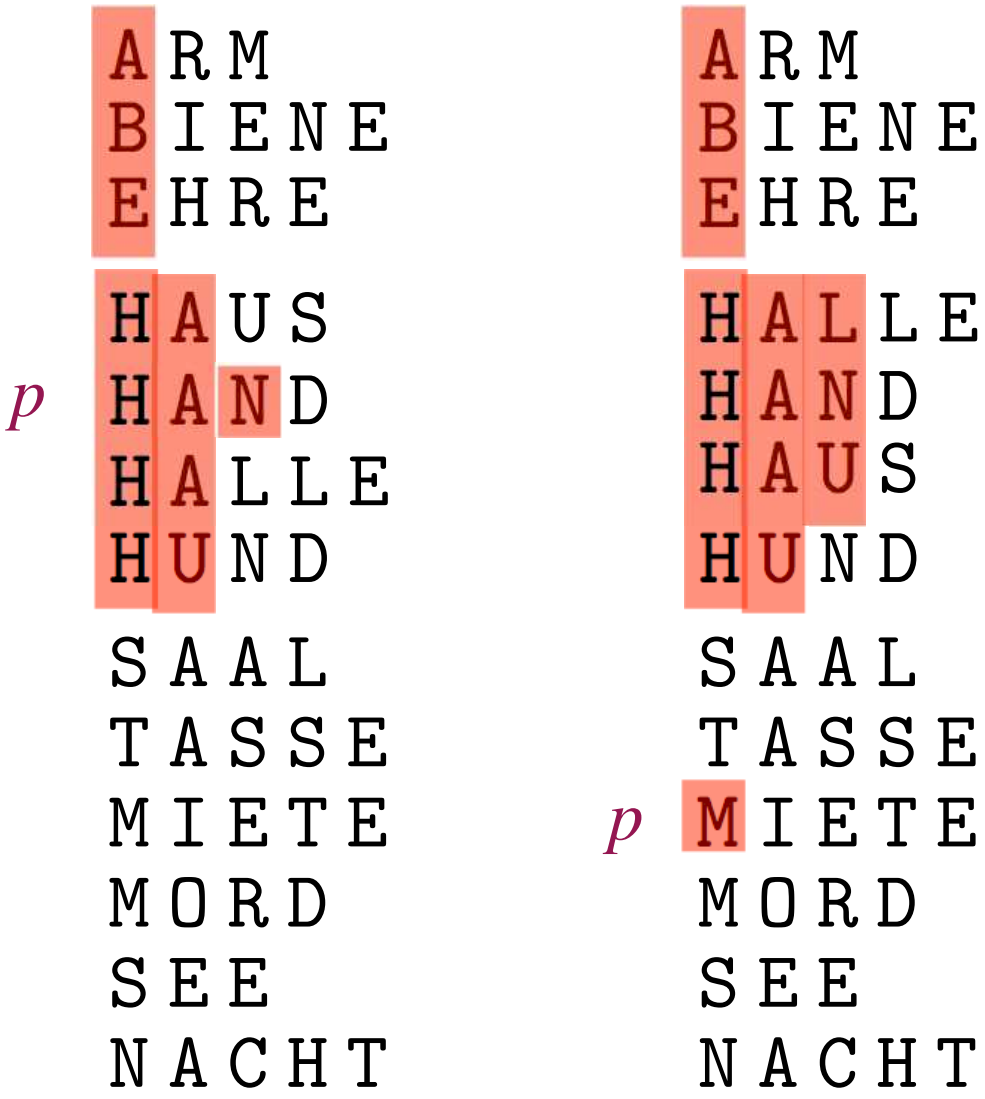
Strings Sortieren



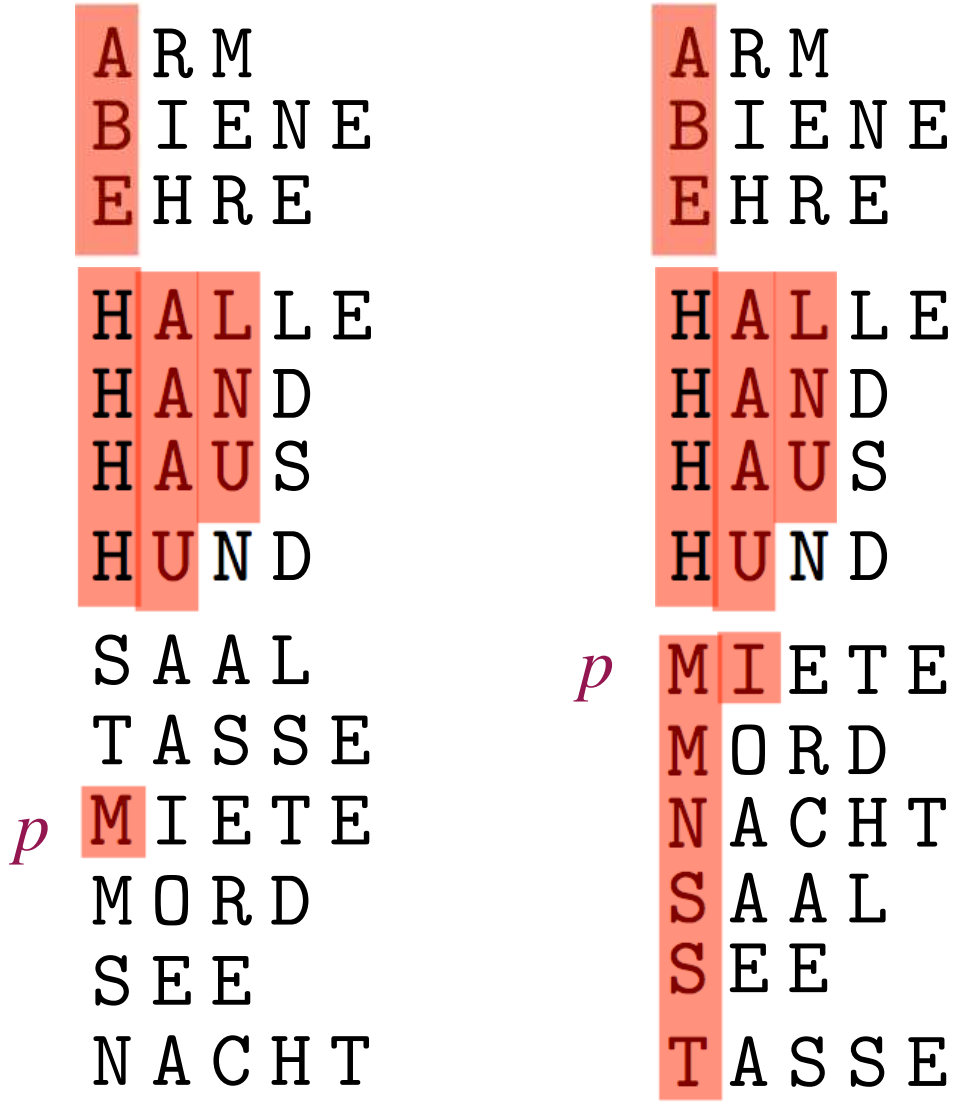
Strings Sortieren



Strings Sortieren



Strings Sortieren



Strings Sortieren

Strings Sortieren

Function mkqSort(S : Sequence **of** String, $\ell : \mathbb{N}$) : Sequence **of** String

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

$S_\perp \leftarrow \langle e \in S : |e| = \ell \rangle$; $S \leftarrow S \setminus S_\perp$

select pivot $p \in S$

$S_< \leftarrow \langle e \in S : e[\ell] < p[\ell] \rangle$

$S_= \leftarrow \langle e \in S : e[\ell] = p[\ell] \rangle$

$S_> \leftarrow \langle e \in S : e[\ell] > p[\ell] \rangle$

return concatenation of $S_\perp$,
mkqSort$(S_<, \ell)$,
mkqSort$(S_=, \ell+1)$, and
mkqSort$(S_>, \ell)$

Strings Sortieren – Laufzeitanalyse

Hauptarbeit in den Buchstabenvergleichen. Zwei Fälle:

- $e[\ell] = p[\ell]$: Ordne den Vergleich dem Zeichen $e[\ell]$ zu.
  - $e[\ell]$ wird danach nicht mehr betrachtet (Rekursion mit $\ell + 1$)
  - Maximale Vergleichsanzahl pro String e? Maximale Länge des längstes gemeinsamen Präfix von e mit $e' \in S$.
- $e[\ell] \neq p[\ell]$: Ordne den Vergleich dem String e zu.
  - e wird zu $S_<$ oder $S_>$ zugeordnet. Mit optimaler Pivotwahl sind beide Mengen höchstens $|S|/2$.
  - Nach höchstens $\log |S|$ Schritten ist e richtig sortiert.

Naives Pattern Matching

- Aufgabe: Finde alle Vorkommen von P in T
  - n: Länge von T
  - m: Länge von P
- naiv in $O(nm)$ Zeit

$i, j := 1$ // indexes in T and P

while $i \leq n - m + 1$

  while $j \leq m$ and $t_{i+j-1} = p_j$ **do** j++ // compare characters

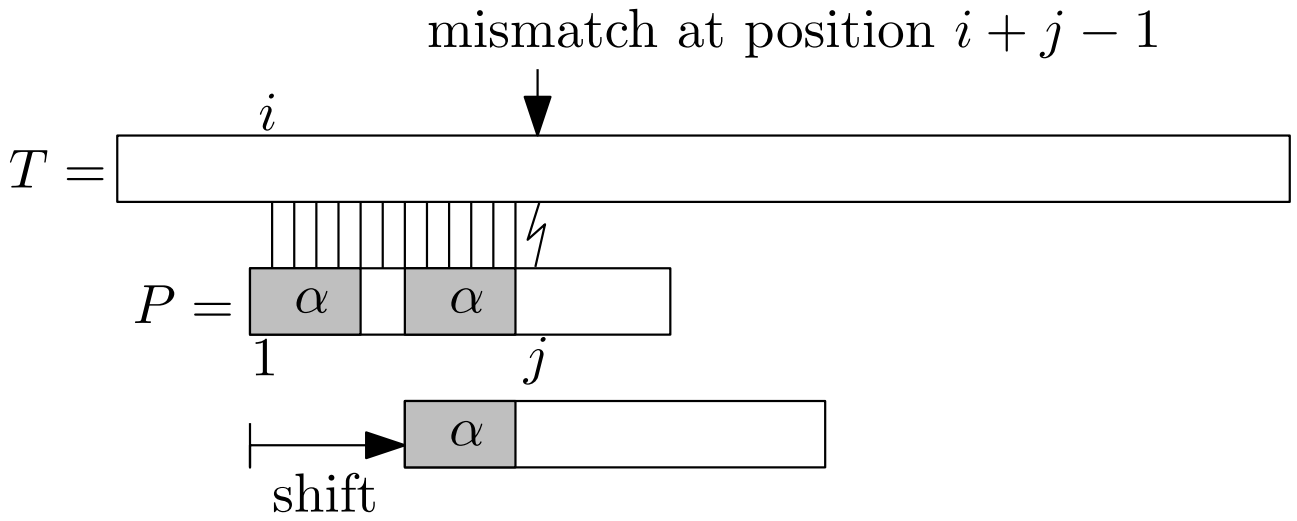
  if $j > m$ **then return** "P occurs at position i in T"

  i++ // advance in T

  $j := 1$ // restart

Knuth-Morris-Pratt (1977)

- besserer Algorithmus in $O(n+m)$ Zeit
- Idee: beachte bereits gematchten Teil



- $\text{border}[j] =$ längstes echtes Präfix von $P_{1\ldots j-1}$, das auch (echtes) Suffix von $P_{1\ldots j-1}$ ist

Knuth-Morris-Pratt (1977)

$i := 1$ // index in T

$j := 1$ // index in P

while $i \leq n - m + 1$

  while $j \leq m$ and $t_{i+j-1} = p_j$ **do** j++ // compare characters

  if $j > m$ **then**

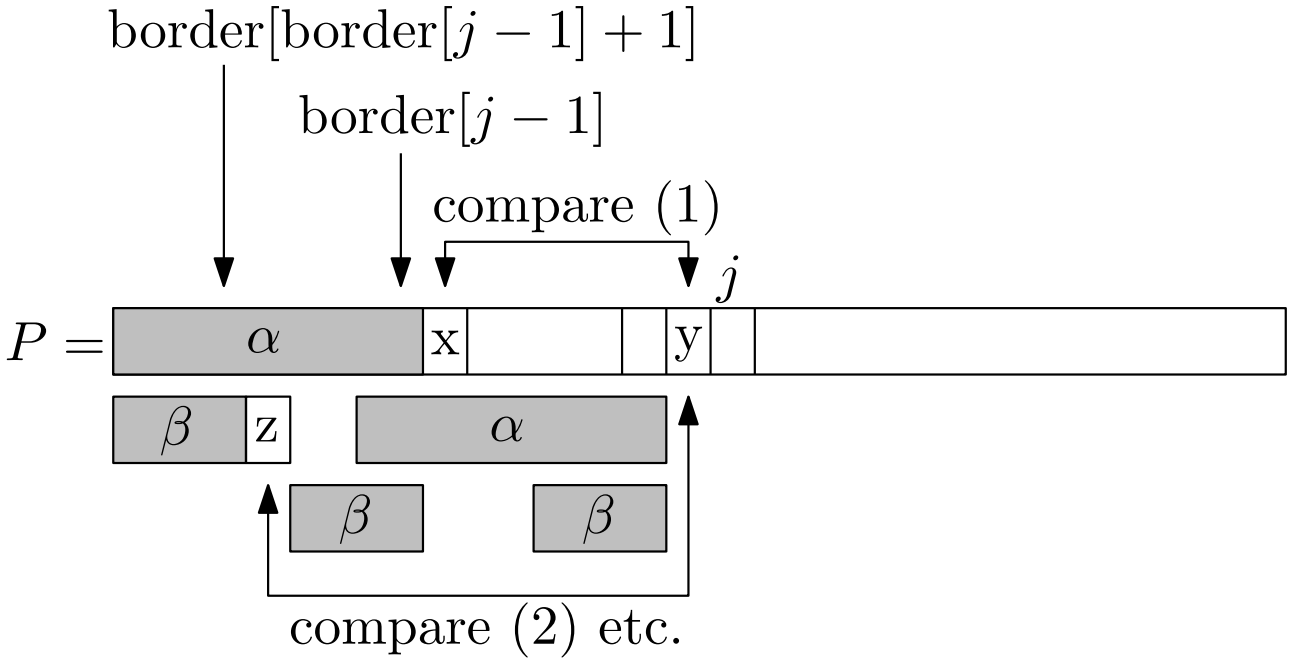
  return "P occurs at position i in T"

  $i := i + j - \text{border}[j] - 1$ // advance in T

  $j := \max\{1, border[j] + 1\}$ // skip first border$[j]$ characters of P

Berechnung des Border-Arrays

- seien die Werte bis zur Position $j-1$ bereits berechnet



Berechnung des Border-Arrays

- in $O(m)$ Zeit:

$\text{border}[1] := -1$

$i := \text{border}[1]$ // position in P

for $j = 2, \ldots, m$

  while $i \geq 0$ and $p_{i+1} \neq p_{j-1}$ **do** $i = \text{border}[i+1]$

  i++

  $\text{border}[j] := i$

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:

Invertierter Index: gute heuristik

Suffix Array: $O(m \log n) \ldots O(m)$

Suffixtabellen

aus

Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Seiten 1–19, Nummer 6, Band 53.

Etwas “Stringology”-Notation

String S: Array $S[0..n) := S[0..n-1] := [S[0], \ldots, S[n-1]]$

von Buchstaben

Suffix: $S_i := S[i..n)$

Endmarkierungen: $S[n] := S[n+1] := \cdots := 0$

0 ist kleiner als alle anderen Zeichen

Suffixe Sortieren

Sortiere die Menge $\{S_0, S_1, \ldots, S_{n-1}\}$
von Suffixen des Strings S der Länge n
(Alphabet $[1,n] = \{1,\ldots,n\}$)
in lexikographische Reihenfolge.

- suffix $S_i = S[i,n]$ für $i \in [0..n-1]$

$S =$ banana:

0	banana
1	anana
2	nana
3	ana
4	na
5	a

$\Longrightarrow$

5	a
3	ana
1	anana
0	banana
4	na
2	nana

Anwendungen

- Volltextsuche
- Burrows-Wheeler Transformation (`bzip2` Kompressor)
- Ersatz für kompliziertere Suffixbäume
- Bioinformatik: Wiederholungen suchen,. . .

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

Binäre Suche mit lcp: $O(m + \log n)$ falls wir die

längsten gemeinsamen (common) Präfixe

zwischen verglichenen Zeichenketten vorberechnen

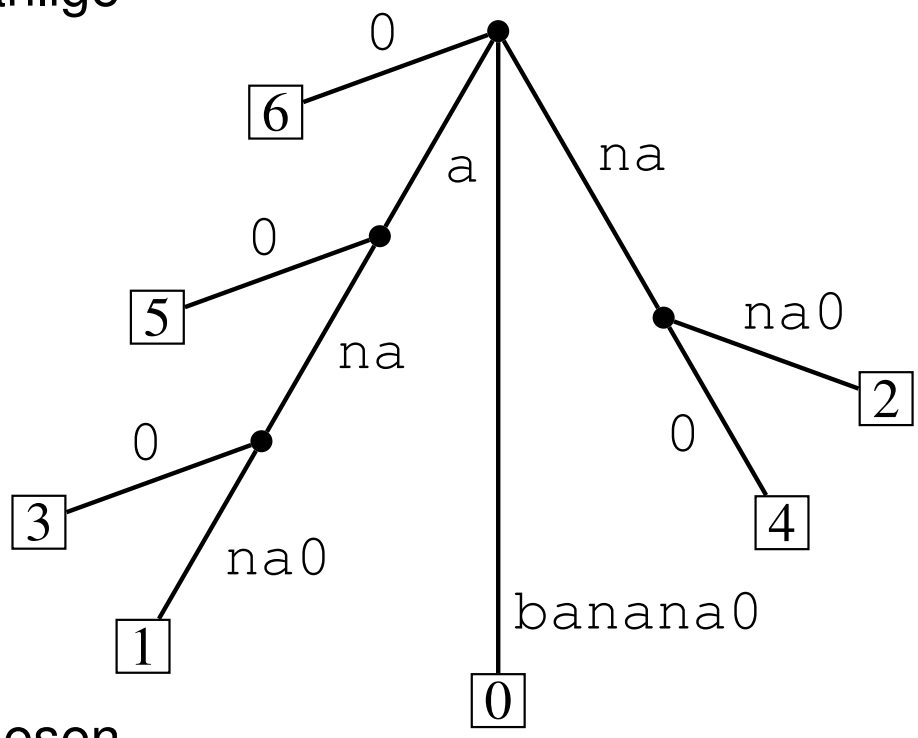
Suffix-Baum: $O(n)$ kann aus SA berechnet werden

Suffix-Baum

[Weiner '73][McCreight '76]

- ☐ kompaktierter Trie der Suffixe
- \+ Zeit $O(n)$ [Farach 97] für ganzzahlige Alphabete
- \+ Mächtigstes Werkzeug der Stringology?
- − Hoher Platzverbrauch
- − Effiziente direkte Konstruktion ist kompliziert
- ☐ kann aus SA in Zeit $O(n)$ abgelesen werden

$S =$ banana0



Alphabet-Modell

Geordnetes Alphabet: Zeichen können nur verglichen werden

Konstante Alphabetgröße: endliche Menge

deren Größe nicht von n abhängt.

Ganzzahliges Alphabet: Alphabet ist $\{1,\ldots,\sigma\}$

für eine ganze Zahl $\sigma \geq 2$

Geordnetes $\rightarrow$ ganzzahliges Alphabet

Sortiere die Zeichen von S

Ersetze $S[i]$ durch seinen Rang

```
012345     135024
banana -> aaabnn
213131 <- 111233
```

Verallgemeinerung: Lexikographische Namen

Sortiere die k-Tupel $S[i..i+k)$ für $i \in 1..n$

Ersetze $S[i]$ durch den Rang von $S[i..i+k)$ unter den Tupeln

Ein erster Teile-und-Herrsche-Ansatz

1. $SA^1 =$sort $\{S_i : i$ ist ungerade$\}$ (Rekursion)
2. $SA^0 =$sort $\{S_i : i$ ist gerade$\}$ (einfach mittels SA^1)
3. Mische SA^0 und SA^1 (schwierig)

Problem: wie vergleicht man gerade und ungerade Suffixe?

[Farach 97] hat einen Linearzeitalgorithmus für Suffix-Baum-Konstruktion entwickelt, der auf dieser Idee beruht.

Sehr kompliziert.

Das war auch der einzige bekannte Algorithmus für Suffix-Tabellen (läßt sich leicht aus S-Baum ablesen.)

SA^1 berechnen

- Erstes Zeichen weglassen.

  banana $\rightarrow$ anana

- Ersetze Buchstabenpaare durch Ihre lexikographischen Namen

  an | an | a0 $\rightarrow$ 221

- Rekursion

  $\langle 1, 21, 221 \rangle$

- Rückübersetzen

  $\langle$a, ana, anana$\rangle$

Berechne SA^0 aus SA^1

1	anana
3	ana
5	a

$\Longrightarrow$

5	a
3	ana
1	anana

Ersetze S_i, $i \bmod 2 = 0$ durch $(S[i], r(S_{i+1}))$

mit $r(S_{i+1})$:= Rang von S_{i+1} in SA^1

0	b 3(anana)
2	n 2(ana)
4	n 1(a)

$\Longrightarrow$

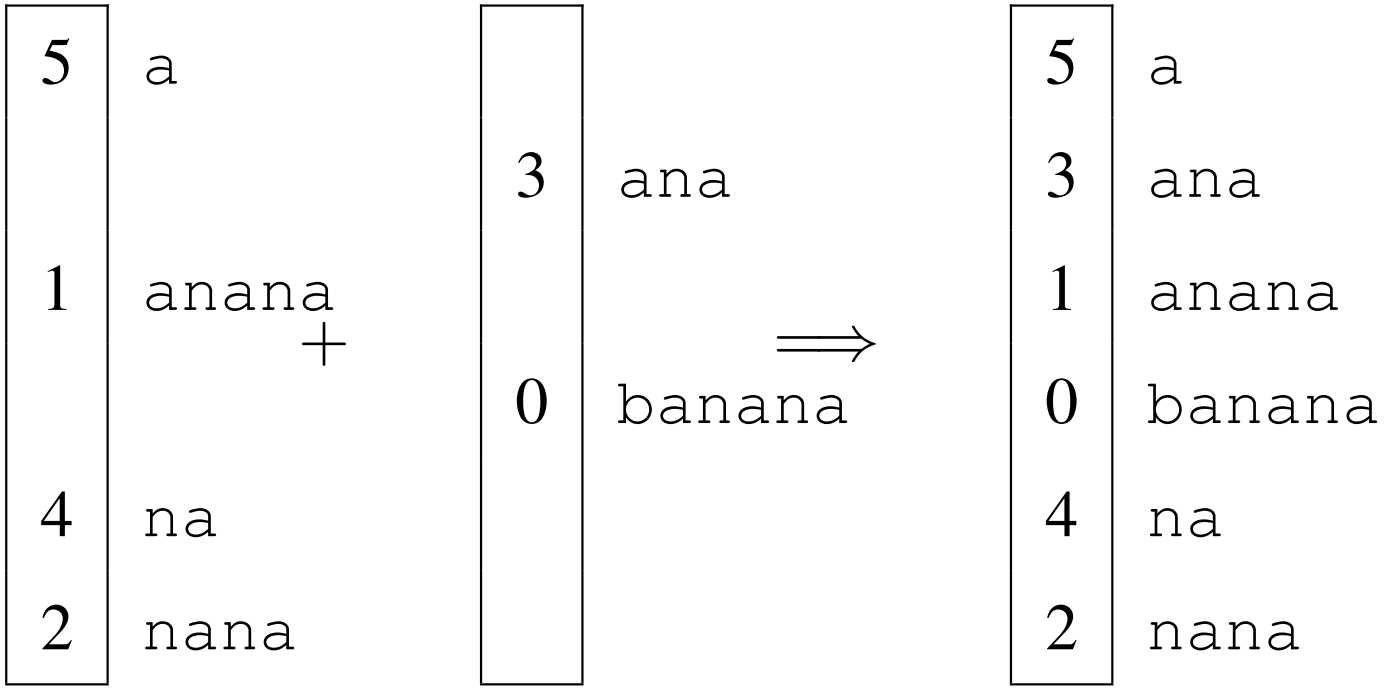
0	banana
4	na
2	nana

Radix-Sort

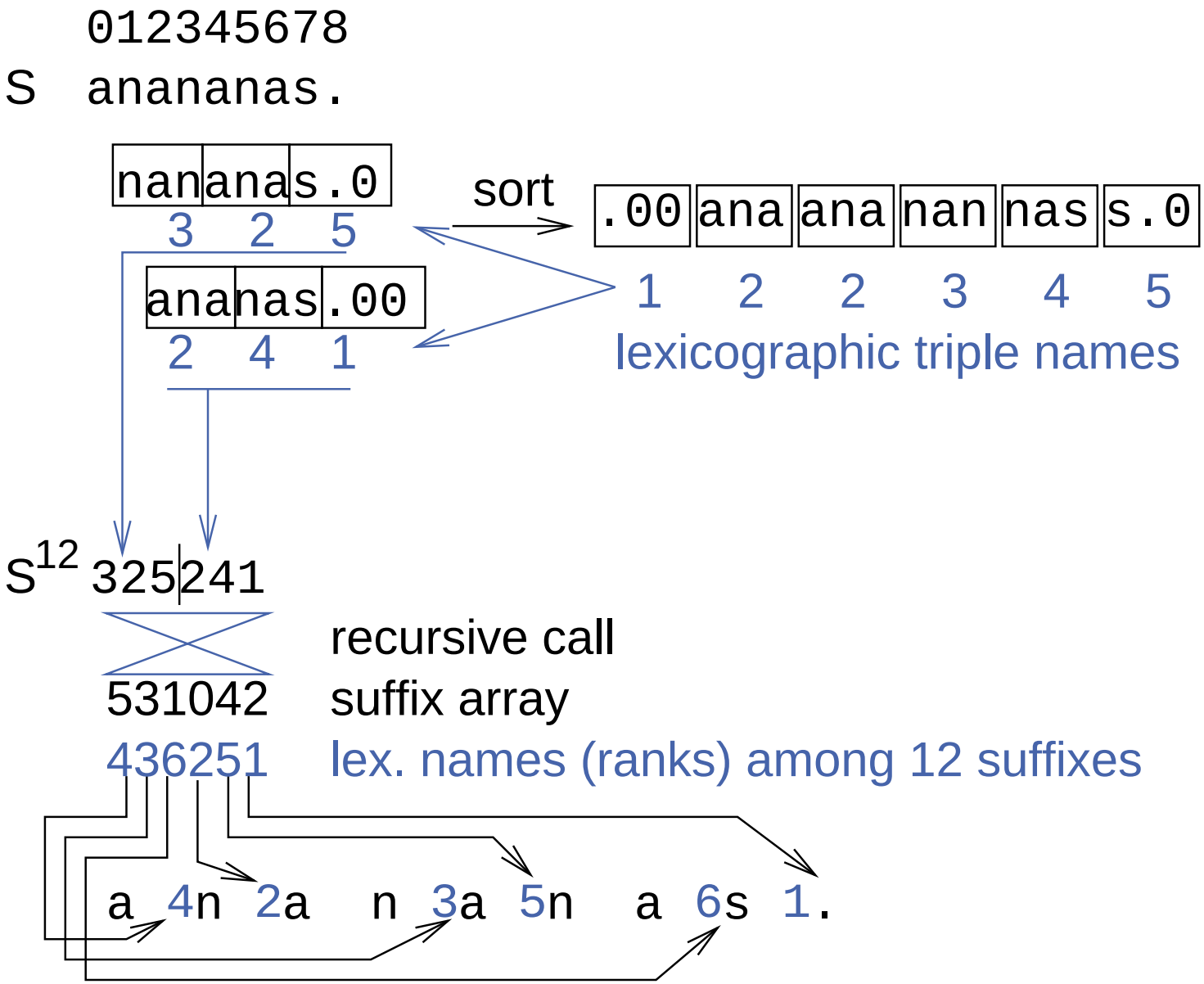
Asymmetrisches Divide-and-Conquer

1. SA^{12} =sort $\{S_i : i \bmod 3 \neq 0\}$ (Rekursion)
2. SA^{0} =sort $\{S_i : i \bmod 3 = 0\}$ (einfach mittels SA^{12})
3. Mische SA^{12} und SA^{0} (einfach!)

$S =$ banana



Rekursion, Beispiel



Rekursion

- sortiere Tripel $S[i..i+2]$ für $i \bmod 3 \neq 0$
  (LSD Radix-Sortieren)
- Finde lexikographische Namen $S'[1..2n/3]$ der Tripel,
  (d.h., $S'[i] < S'[j]$ gdw $S[i..i+2] < S[j..j+2]$)
- $S^{12} = [S'[i] : i \bmod 3 = 1] \circ [S'[i] : i \bmod 3 = 2]$,
  Suffix S^{12}_i von S^{12} repräsentiert S_{3i+1}
  Suffix $S^{12}_{n/3+i}$ von S^{12} repräsentiert S_{3i+2}

- Rekursion auf (S^{12}) (Alphabetgröße $\leq 2n/3$)
- Annotiere die 12-Suffixe mit ihrer Position in rek. Lösung

Least Significant Digit First Radix Sort

Hier: Sortiere n 3-Tupel von ganzen Zahlen $\in [0..n]$ in lexikographische Reihenfolge

Sortiere nach 3. Position

Elemente sind nach Pos. 3 sortiert

Sortiere stabil nach 2. Position

Elemente sind nach Pos. 2,3 sortiert

Sortiere stabil nach 1. Position

Elemente sind nach Pos. 1,2,3 sortiert

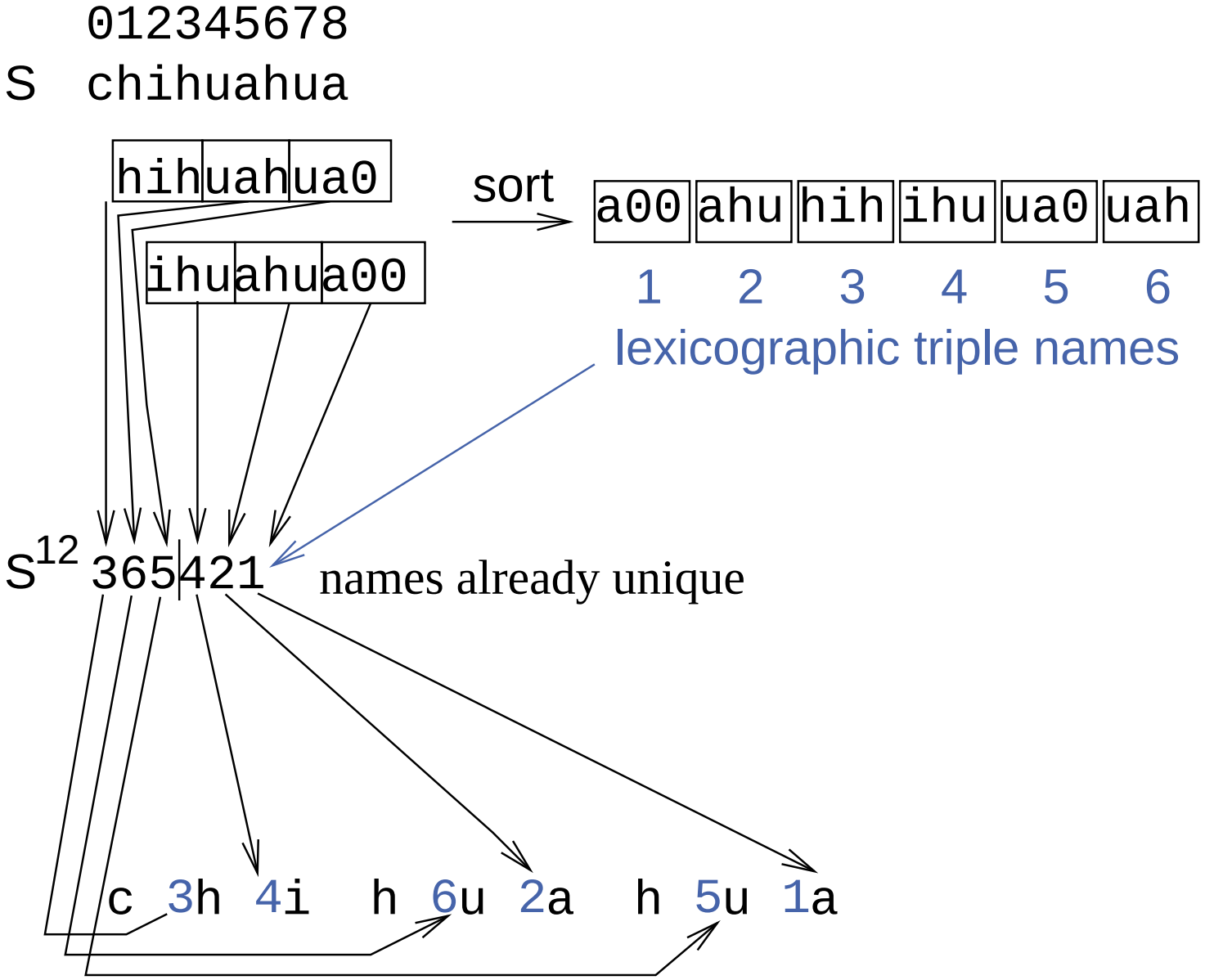
Stabiles Ganzzahliges Sortieren

Sortiere $a[0..n)$ nach $b[0..n)$ mit $\text{key}(a[i]) \in [0..n]$

$c[0..n] := [0,\ldots,0]$	Zähler
for $i \in [0..n)$ do $c[a[i]]{+}{+}$	zähle
$s := 0$	
for $i \in [0..n)$ do $(s, c[i]) := (s + c[i], s)$	Präfixsummen
for $i \in [0..n)$ do $b[c[a[i]]{+}{+}] := a[i]$	bucket sort

Zeit $O(n)$!

Rekursions-Beispiel: Einfacher Fall



Sortieren der mod 0 Suffixe

0	c 3(h 4i h 6u 2a h 5u 1a)
1	
2	
3	h 6(u 2a h 5u 1a)
4	
5	
6	h 5(u 1a)
7	
8	

Benutze Radix-Sort (LSD-Reihenfolge bereits bekannt)

Mische SA^{12} und SA^{0}

$0 < 1 \Leftrightarrow c\,n < c\,n$

$0 < 2 \Leftrightarrow cc\,n < cc\,n$

```
3: h 6u 2(ahua)
6: h 5u 1(a)
0: c 3h 4(ihuahua)
```

```
4: (6)u 2(ahua)
7: (5)u 1(a)
2: (4)i  h 6(uahua)
1: (3)h 4(ihuahua)
5: (2)a  h 5(ua)
8: (1)a 00 0(0)
```

$\Downarrow$

```
8: a
5: ahua
0: chihuahua
1: hihuahua
6: hua
3: huahua
2: ihuahua
7: ua
4: uahua
```

Analyse

1. Rekursion: $T(2n/3)$ plus

   Tripel extrahieren: $\mathrm{O}(n)$ (forall $i, i \bmod 3 \neq 0$ do ...)

   Tripel sortieren: $\mathrm{O}(n)$

   (e.g., LSD-first radix sort — 3 Durchgänge)

   Lexikographisches benenennen: $\mathrm{O}(n)$ (scan)

   Rekursive Instanz konstruieren: $\mathrm{O}(n)$ (forall names do ...)

2. SA^0 =sortiere $\{S_i : i \bmod 3 = 0\}$: $\mathrm{O}(n)$

   (1 Radix-Sort Durchgang)

3. mische SA^{12} and SA^0: $\mathrm{O}(n)$

   (gewöhnliches Mischen mit merkwürdiger Vergleichsfunktion)

Insgesamt: $T(n) \leq cn + T(2n/3)$

$\Rightarrow T(n) \leq 3cn = \mathrm{O}(n)$

Implementierung: Vergleichs-Operatoren

```
inline bool leq(int a1, int a2,   int b1, int b2) {
  return(a1 < b1 || a1 == b1 && a2 <= b2);
}
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3) {
  return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));
}
```

Implementierung: Radix-Sortieren

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];                          // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;          // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;     // count occurences
  for (int i = 0, sum = 0;  i <= K;  i++) { // exclusive prefix sums
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
  delete [] c;
}
```

Implementierung: Tripel Sortieren

```
void suffixArray(int* s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  // generate positions of mod 1 and mod  2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);
```

Implementierung: Lexikographisches Benennen

```
// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0;  i < n02;  i++) {
  if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
    name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
  }
  if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; } // left half
  else                  { s12[SA12[i]/3 + n0] = name; } // right half
```

Implementierung: Rekursion

```
// recurse if names are not yet unique
if (name < n02) {
  suffixArray(s12, SA12, n02, name);
  // store unique names in s12 using the suffix array
  for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
} else // generate the suffix array of s12 directly
  for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;
```

Implementierung: Sortieren der mod 0 Suffixe

```
for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i
radixPass(s0, SA0, s, n0, K);
```

Implementierung: Mischen

```
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    { // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) { // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
      }
    } else {
      SA[k] = j;  p++;
      if (p == n0)  { // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
      }
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0; }
```

Verallgemeinerung: Differenzenüberdeckungen

Ein Differenzenüberdeckung D modulo v ist eine Teilmenge von $[0,v)$, so dass $\forall i \in [0,v) : \exists j,k \in D : i \equiv k - j \pmod{v}$.

Beispiel:

$\{1,2\}$ ist eine Differenzenüberdeckung modulo 3.

$\{1,2,4\}$ ist eine Differenzenüberdeckung modulo 7.

- ☐ Führt zu platzeffizienterer Variante
- ☐ Schneller für kleine Alphabete

Verbesserungen / Verallgemeinerungen

- tuning
- größere Differenzenüberdeckungen
- Kombiniere mit den besten Alg. für einfache Eingaben

  [Manzini Ferragina 02, Schürmann Stoye 05, Yuta Mori 08]

Suffixtabellenkonstruktion: Zusammenfassung

- einfache, direkte, Linearzeit für Suffixtabellenkonstruktion
- einfach anpassbar auf fortgeschrittene Berechnungsmodelle
- Verallgemeinerung auf Diff-Überdeckungen ergibt platzeffiziente Implementierung

Suche in Suffix Arrays

$l := 1; r := n+1$

while $l < r$ **do** // search left index

  $q := \lfloor \frac{l+r}{2} \rfloor$

  if $P >_{\text{lex}} T_{\text{SA}[q]\ldots\min\{\text{SA}[q]+m-1,n\}}$

    then $l := q+1$ **else** $r := q$

$s := l; l--; r := n$

while $l < r$ **do** // search right index

  $q := \lceil \frac{l+r}{2} \rceil$

  if $P =_{\text{lex}} T_{\text{SA}[q]\ldots\min\{\text{SA}[q]+m-1,n\}}$

    then $l := q$ **else** $r := q-1$

return $[s,l]$

- Zeit $O(m \log n)$ (geht auch: $O(m + \log n)$)

LCP-Array

speichert Längen der längsten gemeinsamen Präfixe lexikographisch benachbarter Suffixe!

$S =$ banana:

0	banana
1	anana
2	nana
3	ana
4	na
5	a

SA =

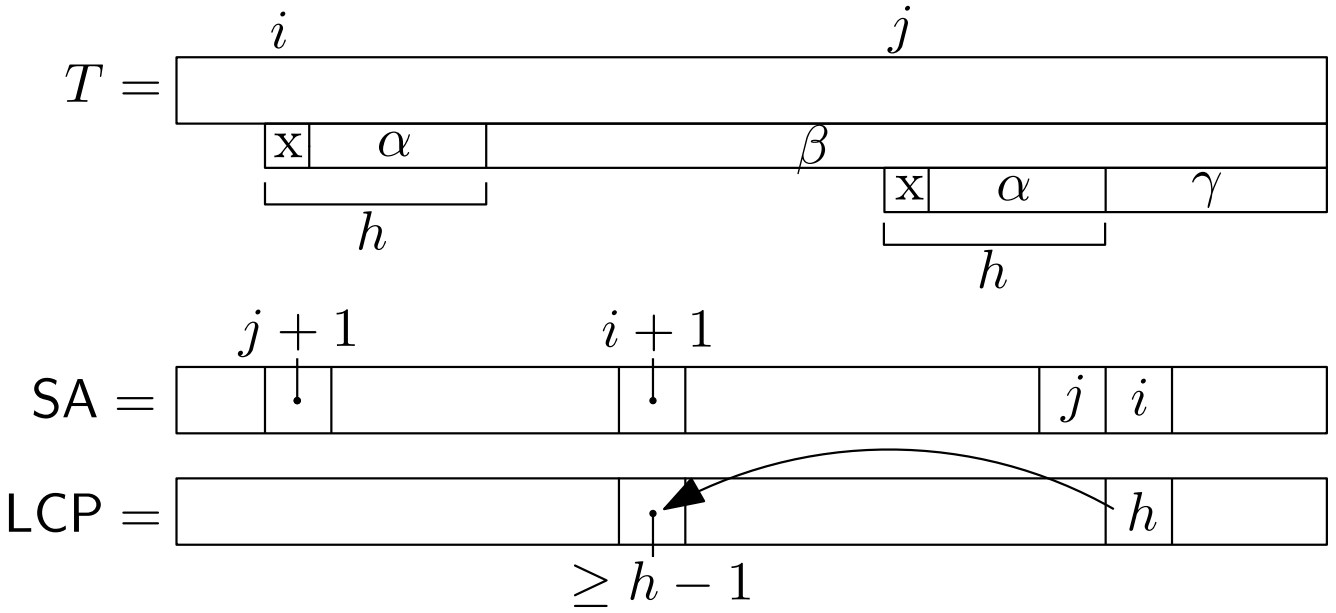
5	a
3	ana
1	anana
0	banana
4	na
2	nana

LCP =

0	a
1	**a**na
3	**ana**na
0	banana
0	na
2	**na**na

LCP-Array: Berechnung

- naiv $O(n^2)$
- inverses Suffix-Array: $\mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$ (wo steht i in SA?)
- For all $1 \leq i < n$: $\mathrm{LCP}[\mathrm{SA}^{-1}[i+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[i]] - 1$.



LCP-Array: Berechnung

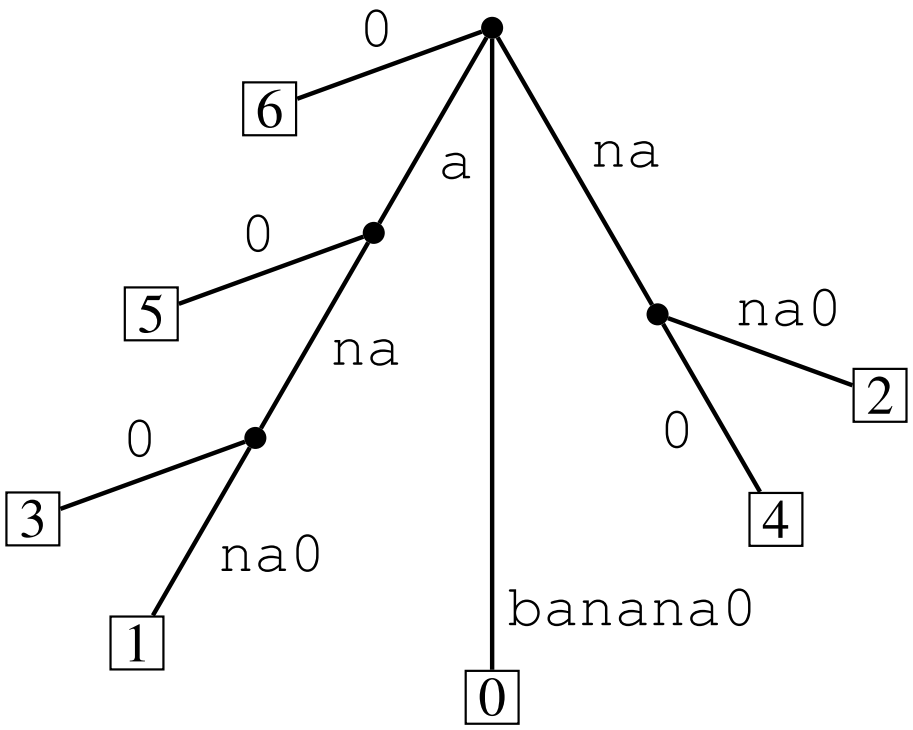
- For all $1 \leq i < n$: $\mathrm{LCP}[\mathrm{SA}^{-1}[i+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[i]] - 1$.

$h := 0, \mathrm{LCP}[1] := 0$
for $i = 1, \ldots, n$ **do**
  if $\mathrm{SA}^{-1}[i] \neq 1$ **then**
    while $t_{i+h} = t_{\mathrm{SA}[\mathrm{SA}^{-1}[i]-1]+h}$ **do** $h++$
    $\mathrm{LCP}[\mathrm{SA}^{-1}[i]] := h$
    $h := \max(0, h-1)$

- Zeit: $O(n)$

Suffix-Baum aus SA und LCP

$S =$ banana0



- naiv: $O(n^2)$
- mit Suffix-Array: in lexikographischer Reihenfolge

Suffix-Baum aus SA und LCP

- LCP-Werte helfen!
- Betrachte nur rechtesten Pfad!
- Finde tiefsten Knoten mit String-Tiefe $\leq \mathrm{LCP}[i] \rightsquigarrow$ Einfügepunkt!

	0	1	2	3	4	5	6
$\mathrm{SA} =$	6	5	3	1	0	4	2
$\mathrm{LCP} =$	0	0	1	3	0	0	2

- Zeit $O(n)$

Suche in Suffix-Bäumen

- Suche (alle/ein) Vorkommen von $P_{1..m}$ in T:
- Wenn ausgehende Kanten Arrays der Größe $|\Sigma|$:
  - $O(m)$ Suchzeit
  - $O(n|\Sigma|)$ Gesamtplatz
- Wenn ausgehende Kanten Arrays der Größe prop. zur #Kinder:
  - $O(m \log |\Sigma|)$ Suchzeit
  - $O(n)$ Platz

Datenkompression

Problem: bei naiver Speicherung verbrauchen Daten sehr viel Speicherplatz / Kommunikationsbandbreite. Das läßt sich oft reduzieren.

Varianten:

- Verlustbehaftet (mp3, jpg, ...) / **Verlustfrei** (Text, Dateien, Suchmaschinen, Datenbanken, medizin. Bildverarbeitung, Profifotografie, ...)
- **1D** (text, Zahlenfolgen,...) / 2D (Bilder) / 3D (Filme)
- **nur Speicherung** / mit Operationen ($\rightsquigarrow$ succinct data structures)

Verlustfreie Textkompression

Gegeben: Alphabet Σ

String $S = \langle s_1, \ldots, s_n \rangle \in \Sigma^*$

Textkompressionsalgorithms $f : S \to f(S)$ mit $|f(S)|$ (z.B. gemessen in bits) möglichst klein.

Theorie Verlustfreier Textkompression

Informationstheorie. Zum Beispiel

Entropie: $H(S) = \sum_{c \in \Sigma} p(c) \log(1/p(c))$ wobei

$p(c) = |\{s_i : s_i = c\}|/n$ die relative Häufigkeit von c ist.

untere Schranke für # bits pro Zeichen falls Text einer Zufallsquelle entspränge.

$\rightsquigarrow$ Huffman-Coding ist annähernd optimal! (Entropiecodierung) ????

Schon eher:

Entropie höherer Ordnung betrachte Teilstrings fester Länge

“Ultimativ”: Kolmogorov Komplexität. Leider nicht berechenbar.

Theorie Verlustfreier Textkompression

Entropie höherer Ordnung: Gegeben ein Text S der Länge n über dem Alphabet Σ. Wir definieren $N(\omega, S)$ als Konkatenation aller Zeichen, die in S auf Vorkommen von $\omega \in \Sigma^k$ folgen. Wir definieren die empirische Entropie der Ordung k wie folgt:

$$H_k(S) = \sum_{\omega \in \Sigma^k} \frac{|N(\omega, S)|}{n} H(N(\omega, S))$$

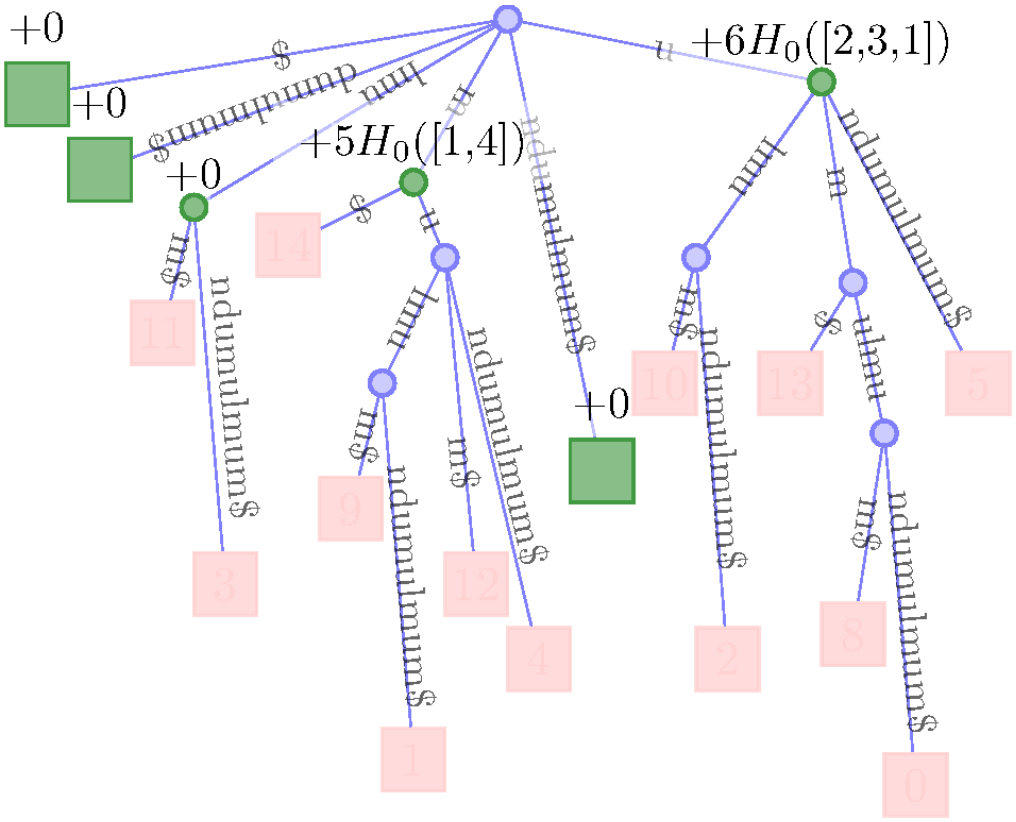
Beispiel: $S = \texttt{ananas}$, $k = 2$

$N(\texttt{an}, S) = \texttt{aa}$, $N(\texttt{na}, S) = \texttt{ns}$, $N(\texttt{as}, S) = \varepsilon$.

$H_2(\texttt{ananas}) = \frac{2}{6} H(\texttt{ns}) = \frac{1}{3}$ bits

Theorie Verlustfreier Textkompression

H_k: Berechnung mittels Suffixbaumes (Beispiel: H_1)



Theorie Verlustfreier Textkompression

Werte der empirischen Entropie in der Praxis

$H_k(S)$ in bits und (# eindeutiger Kontexte/$|S|$ in Prozent)

k	dblp.xm		DNA		english		proteins	
0	5.26	0.0	1.97	0.0	4.53	0.0	4.20	0.0
1	3.48	0.0	1.93	0.0	3.62	0.0	4.18	0.0
2	2.17	0.0	1.92	0.0	2.95	0.0	4.16	0.0
3	1.43	0.1	1.92	0.0	2.42	0.0	4.07	0.0
4	1.05	0.4	1.91	0.0	2.06	0.3	3.83	0.1
5	0.82	1.3	1.90	0.0	1.84	1.0	3.16	1.7
6	0.70	2.7	1.88	0.0	1.67	2.7	1.50	17.4

Wörterbuchbasierte Textkompression

Grundidee: wähle $\Sigma' \subseteq \Sigma^*$ und ersetze $S \in \Sigma^*$ durch $S' = \langle s'_1, \ldots, s'_k \rangle \in \Sigma'^*$, so dass $S = s'_1 \cdot s'_2 \cdots s'_k$. (mit '$\cdot$'= Zeichenkettenkonkatenation.

Platz $n \lceil \log \Sigma \rceil \rightarrow k \lceil \log \Sigma' \rceil$ mit Entropiecodierung der Zeichen aus Σ' sogar $k\text{Entropie}(S')$

Problem: zusätzlicher Platz für Wörterbuch.

OK für sehr große Datenbestände.

Wörterbuchbasierte Textkompression – Beispiel

Volltextsuchmaschinen verwenden oft $\Sigma' :=$ durch Leerzeichen (etc.) separierte Wörter der zugrundeliegenden natürlichen Sprache.

Spezialbehandlung von Trennzeichen etc.

`Gallia est omnis divisa in partes tres,...`

$\rightarrow$ gallia | est | omnis | divisa | in | partes | tres ...

Lempel-Ziv Kompression (LZ)

Idee: baue Wörterbuch “on the fly” bei Codierung und Decodierung.

Ohne explizite Speicherung!

Naive Lempel-Ziv Kompression (LZ)

Procedure naiveLZCompress$(\langle s_1, \ldots, s_n \rangle, \Sigma)$

$D := \Sigma$ // Init Dictionary

$p := s_1$ // current string

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

if $p \cdot s_i \in D$ **then** $p := p \cdot s_i$

else

output code for p

$D := D \cup p \cdot s_i$

$p := s_i$

output code for p

Naive LZ Dekompression

Procedure naiveLZDecode$(\langle c_1, \ldots, c_k \rangle)$

$D := \Sigma$

output decode(c_1)

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

if $c_i \in D$ **then**

$D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_i)[1]$

else

$D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_{i-1})[1]$

output decode(c_i)

LZ Beispiel: abracadabra

#	p	output	input	$D \cup =$
1	$\perp$	-	a	a,b,c,d,r
2	a	a	b	ab
3	b	b	r	br
4	r	r	a	ra
5	a	a	c	ac
6	c	c	a	ca
7	a	a	d	ad
8	d	d	a	da
9	a	-	b	-
10	ab	ab	r	abr
11	r	-	a	-
-	ra	ra	-	-

LZ-Verfeinerungen

- Wörterbuchgröße begrenzen, z.B. $|D| \leq 4096 \rightsquigarrow$ 12bit codes.
- Von vorn wenn Wörterbuch voll $\rightsquigarrow$ Blockweise arbeiten
- Kodierung mit variabler Zahl Bits (z.B. Huffman, arithmetic coding)
- Selten benutzte Wörterbucheinträge löschen ???
- Wörterbuch effizient implementieren:
  (universelles) hashing

LCP zwischen beliebigen Suffixen

- Wie berechnet man die Länge des längste gemeinsame Präfix zweier Suffixe S_x und S_y mit $(x \neq y)$?
- Falls LCP Array schon vorliegt:
  - Sei $\ell = min(SA^{-1}[x], SA^{-1}[y])$ und $r = max(SA^{-1}, SA^{-1}[y])$
  - $LCP(S_x, S_y) = min_{\ell < i \leq r}\{LCP[i]\}$
  - Frage: Wie bestimmt man die Position des Minimums in einem Array A effizient?

Motivation für Range-Minimum-Query (RMQ) Datenstrukturen.

Range minimum queries (RMQs)

Karlsruhe Institute of Technology

Definition

Given an array A of length n containing elements from a totally ordered set. A range minimum query $rmq_A(\ell, r)$ returns the *position* of the minimal element in the sub-array $A[\ell, r]$:

$$rmq_A(\ell, r) = \arg\min_{\ell \leq k \leq r} A[k]$$

Example

$A =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	2	4	7	1	9	3	5	7	4	6	4	3	1	4	8

Range minimum queries (RMQs)

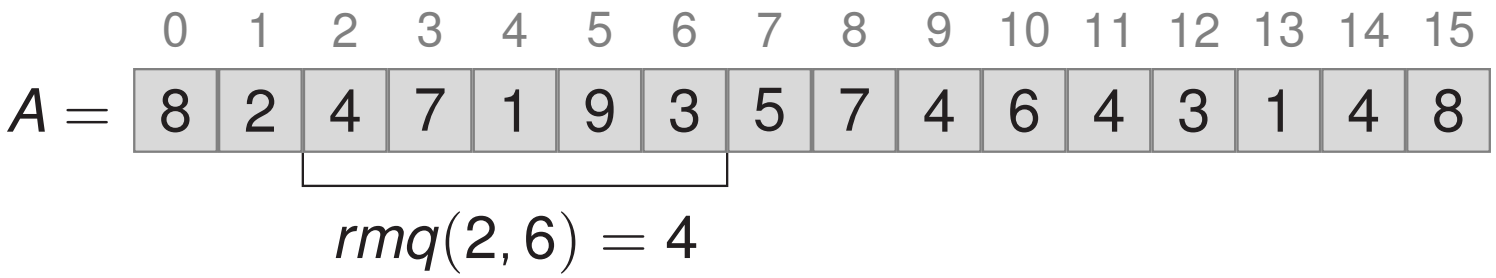
Karlsruhe Institute of Technology

Definition

Given an array A of length n containing elements from a totally ordered set. A range minimum query $rmq_A(\ell, r)$ returns the *position* of the minimal element in the sub-array $A[\ell, r]$:

$$rmq_A(\ell, r) = \arg\min_{\ell \leq k \leq r} A[k]$$

Example



Range minimum queries (RMQs)

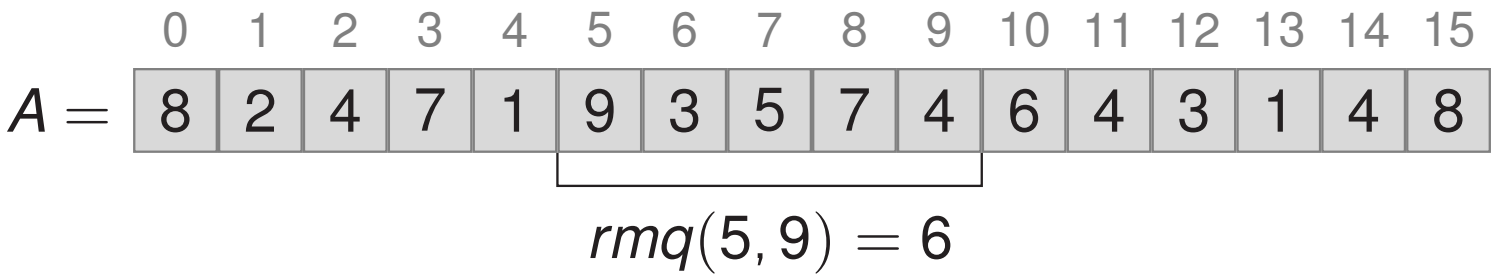
Karlsruhe Institute of Technology

Definition

Given an array A of length n containing elements from a totally ordered set. A range minimum query $rmq_A(\ell, r)$ returns the *position* of the minimal element in the sub-array $A[\ell, r]$:

$$rmq_A(\ell, r) = \arg\min_{\ell \leq k \leq r} A[k]$$

Example



Range minimum queries (RMQs)

Karlsruhe Institute of Technology

Definition

Given an array A of length n containing elements from a totally ordered set. A range minimum query $rmq_A(\ell, r)$ returns the *position* of the minimal element in the sub-array $A[\ell, r]$:

$$rmq_A(\ell, r) = \arg\min_{\ell \leq k \leq r} A[k]$$

Example

$A =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	2	4	7	1	9	3	5	7	4	6	4	3	1	4	8

$$rmq(0, 15) = 4$$

Overview

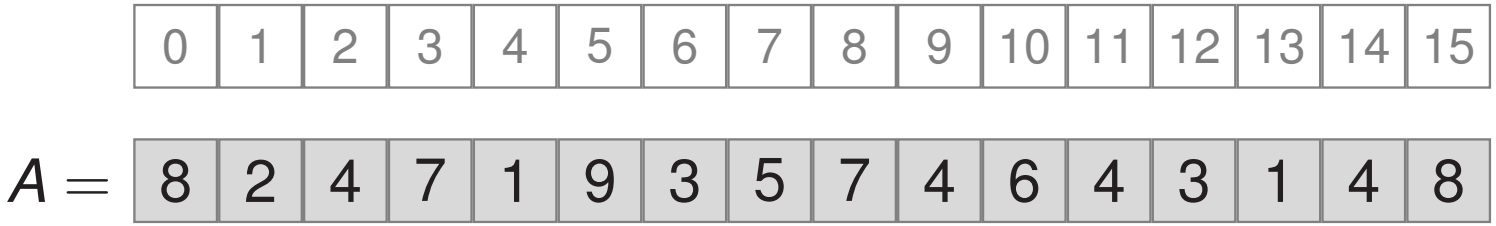
- Notation: Complexity of an algorithm is denoted with $\langle f(n), g(n) \rangle$, where $f(n)$ is preprocessing time and $g(n)$ query time.
- Different solutions:
  - naïve approach 1: $\langle O(n^2), O(1) \rangle$ using $O(n^2)$ words of space
  - naïve approach 2: $\langle O(1), O(n) \rangle$
  - $\langle O(n), O(\log n) \rangle$ using $O(n)$ words of space
  - $\langle O(n \log n), O(1) \rangle$ using $O(n \log n)$ words of space
  - $\langle O(n \log \log n), O(1) \rangle$ using $O(n \log \log n)$ words of space
  - $\langle O(n), O(1) \rangle$ using $O(n)$ words of space
  - $\langle O(n), O(1) \rangle$ using $4n + o(n)$ bits of space
  - $\langle O(n), O(1) \rangle$ using $2n + o(n)$ bits of space

Note

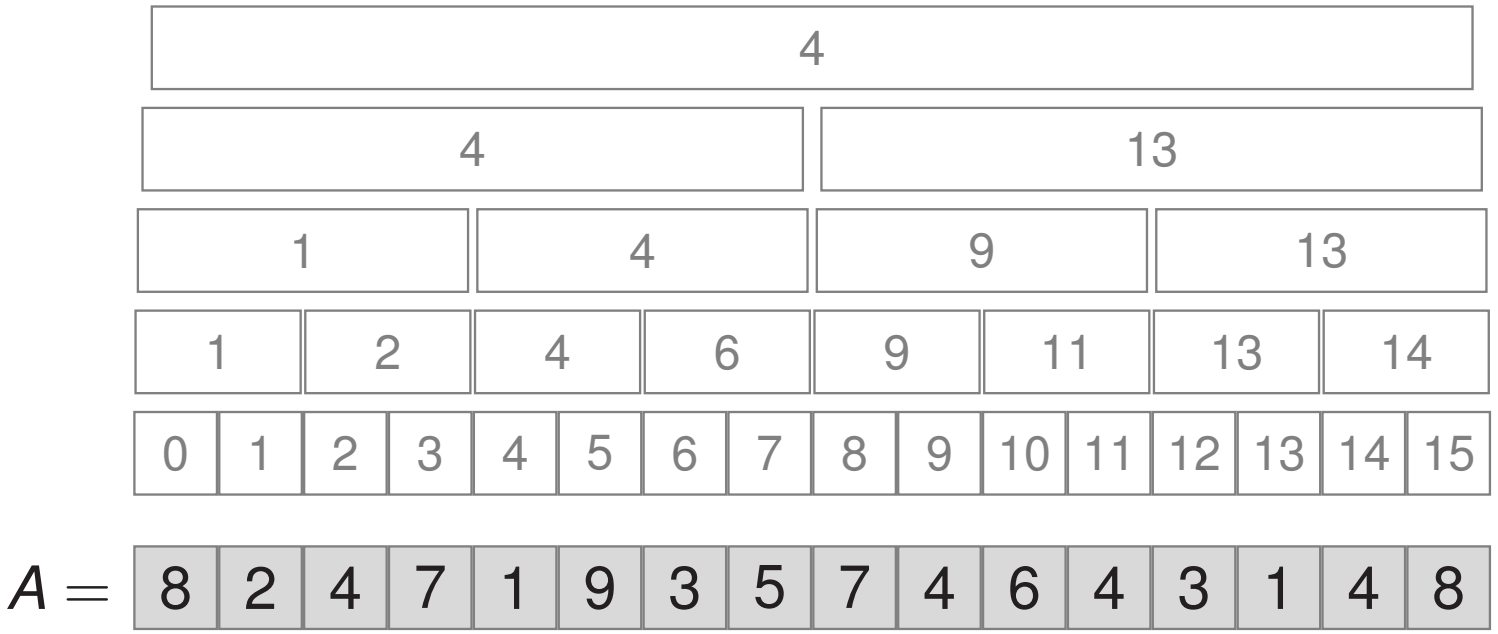
The last two solutions do not require access to the original array A.

$\langle O(n), O(\log n) \rangle$ – solution #1

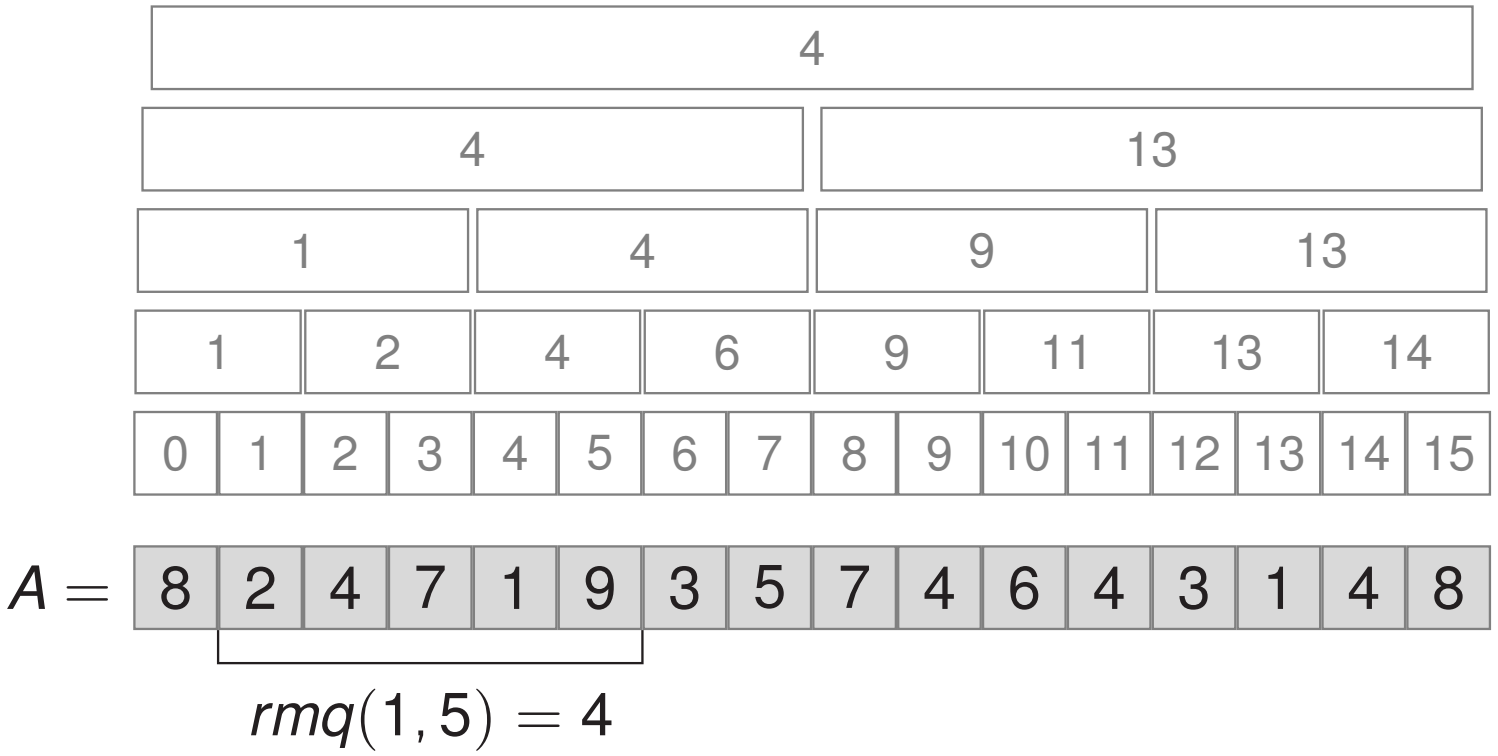
Karlsruhe Institute of Technology



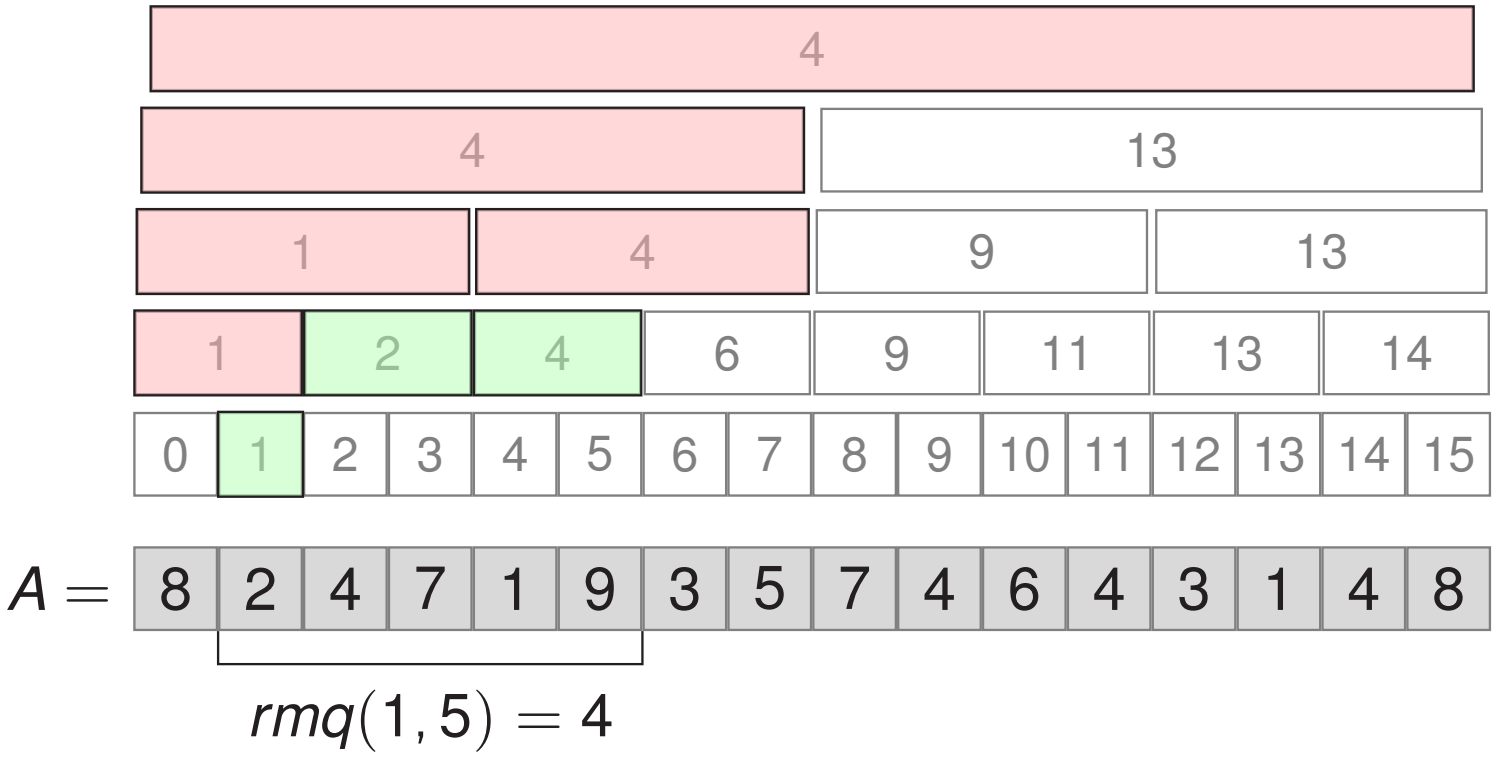
$\langle O(n), O(\log n) \rangle$ – solution #1



$\langle O(n), O(\log n) \rangle$ – solution #1



$\langle O(n), O(\log n) \rangle$ – solution #1



$\langle O(n), O(\log n) \rangle$ – solution #1

Karlsruhe Institute of Technology

- Store index of minimum in binary interval tree.
- Tree has $O(n)$ nodes.
- Follow all nodes which overlap with the query interval but are not fully contained in it (at most 2 per level).
- So not more than $2 \log n$ such nodes in total.
- Select all children of these nodes which are fully contained in the query interval.
- From these nodes select the index with minimal value.

$\langle O(n \log n), O(1) \rangle$ – solution #2

- For each item $A[i]$ store an array $M_i[0, \log n]$.
- $M_i[j] = rmq_A(i, i + 2^j - 1)$
- Space is $O(n \log n)$ words
- How long does pre-computation take?

Querying

Find the largest k with $2^k \leq \ell - r + 1$. Then

$$rmq_A(\ell, r) = \begin{cases} M_i[k] & \text{if } A[M_i[k]] < A[M_{j-2^k+1}[k]] \\ M_{j-2^k+1}[k] & \text{otherwise} \end{cases}$$

Question: How can k be determined in constant time?

$\langle O(n \log\log n), O(1) \rangle$ solution

- Split A into $t = \frac{n}{\log n}$ blocks $B_0, ..., B_{t-1}$. B spans $O(\log n)$ items of A.
- Create an array $S[0, t-1]$ with $S[i] = min\{x \in B_i\}$
- Build rmq structure #2 for S
- For each block B_i of $O(\log n)$ elements build rmq structure #2
- Total space: $O(n) + O(n \log\log n)$

Querying

- Determine blocks $B_{\ell'}$, $B_{r'}$ which contain ℓ and r
- Calculate $m = rmq_S(\ell' + 1, r' - 1)$
- Let k_0, k_1, k_2 be the results of the RMQs in blocks ℓ', r', and m relative to A
- Return $\arg\min_{k_i} A[k_i]$ for $0 \leq k \leq 3$

$\langle O(n), O(1) \rangle$ solution

Karlsruhe Institute of Technology

Definition

The Cartesian Tree C of an array is defined as follows:

- The root of C is the (leftmost) minimum element of the array and is labeled with its position
- Removing the root splits the array into two pieces
- The left and right children of the root are recursively constructed Cartesian trees of the left and right subarray
- C can be constructed in linear time

$\langle O(n), O(1) \rangle$ solution

Solution overview:

- Partition the array into blocks of size s
- Each block corresponds to a Cartesian Tree of size s
- Precompute the s^2 answers for all $\frac{1}{s+1}\binom{2s}{s}$ possible Cartesian Trees of size s in a table P.
- P requires $O(2^{2s}s^2)$ words of space
- For $s = \frac{\log n}{4}$ P requires $o(n)$ words of space
- Build structure #2 for array A' consisting of the block minima of A. This takes $O(n)$ construction time and uses $O(n)$ words of space.

LCA & $\pm 1 RMQ$

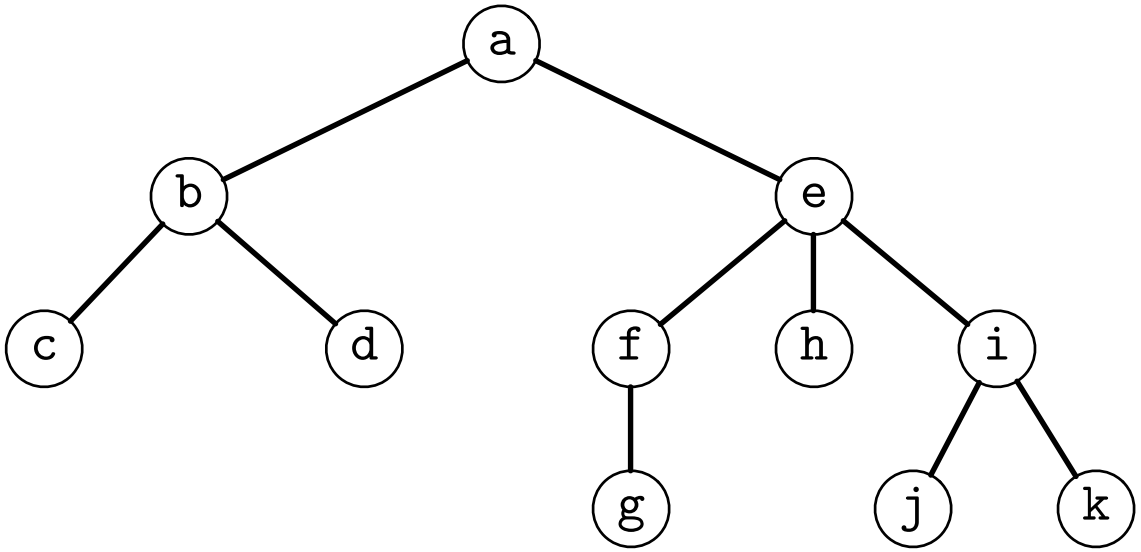
LCA (Lowest Common Ancestor)

Given a rooted tree T of n nodes. For nodes v and w of T the query $LCA_T(v, w)$ returns the *lowest common ancestor* of u and v in T. I.e. the node which is (1) ancestor of v and w and (2) maximizes the distance to the root.

LCA & $\pm 1 RMQ$

LCA (Lowest Common Ancestor)

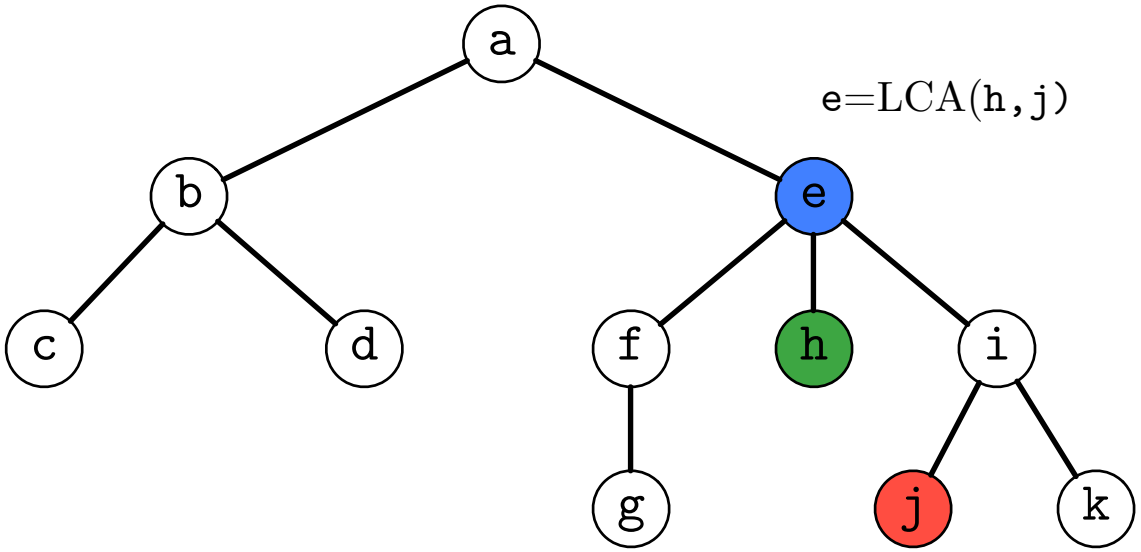
Given a rooted tree T of n nodes. For nodes v and w of T the query $LCA_T(v, w)$ returns the *lowest common ancestor* of u and v in T. I.e. the node which is (1) ancestor of v and w and (2) maximizes the distance to the root.



LCA & $\pm 1 RMQ$

LCA (Lowest Common Ancestor)

Given a rooted tree T of n nodes. For nodes v and w of T the query $LCA_T(v, w)$ returns the *lowest common ancestor* of u and v in T. I.e. the node which is (1) ancestor of v and w and (2) maximizes the distance to the root.



LCA & $\pm 1 RMQ$

Karlsruhe Institute of Technology

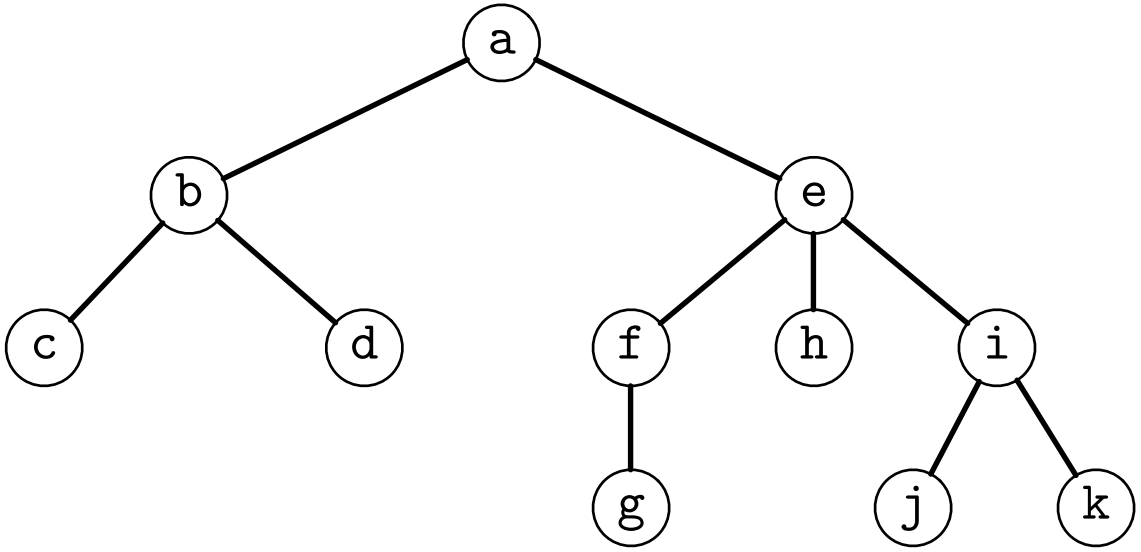
Lemma

If there is a $\langle f(n), g(n) \rangle$-time solution for RMQ, then there is a $\langle f(2n-1) + O(n), g(2n-1) + O(1) \rangle$-time solution for LCA.

- Let T be the Cartesian Tree of array A
- Let array $E[0, \ldots, 2n-2]$ store the nodes visited in an DFS Euler Tour of T
- Let array $L[0, \ldots, 2n-2]$ store the corresp. levels of the nodes in E
- Let $R[0, \ldots, n-1]$ be an array which stores a representative $R[i] = \min\{j \mid E[j] = i\}$ for each node i of T

LCA & $\pm 1 RMQ$

Example




```
i = 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
E = a b c b d b a e f g f e h e i j i k i e a
L = 0 1 2 1 2 1 0 1 2 3 2 1 2 1 2 3 2 3 2 1 0

    a b c d e f g  h  i  j  k
R = 0 1 2 4 7 8 9 12 14 15 17
```

LCA & $\pm 1 RMQ$

Example

```
i = 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
E = a b c b d b a e f g f e h e i j i k i e a
L = 0 1 2 1 2 1 0 1 2 3 2 1 2 1 2 3 2 3 2 1 0

    a b c d e f g  h  i  j  k
R = 0 1 2 4 7 8 9 12 14 15 17
```

$$LCA_T(v, w) = E[RMQ_L(min(R[v], R[w]), max(R[v], R[w]))]$$

LCA & $\pm 1 RMQ$

Example

```
i = 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
E = a b c b d b a e f g f e h e i j i k i e a
L = 0 1 2 1 2 1 0 1 2 3 2 1 2 1 2 3 2 3 2 1 0

    a b c d e f g  h  i  j  k
R = 0 1 2 4 7 8 9 12 14 15 17
```

$$LCA_T(v, w) = E[RMQ_L(min(R[v], R[w]), max(R[v], R[w]))]$$

Note: $(L[i] - L[i+1]) \in \{-1, +1\}$. So we only need to solve RMQs over arrays with this ± 1 restriction. This is called $\pm 1 RMQ$.

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

Karlsruhe Institute of Technology

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

											1	
i	=	0	1	2	3	4	5	6	7	8	9	0	1
A	=	4	6	3	5	1	4	6	4	5	2	6	3

(1) Build Cartesian Tree

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

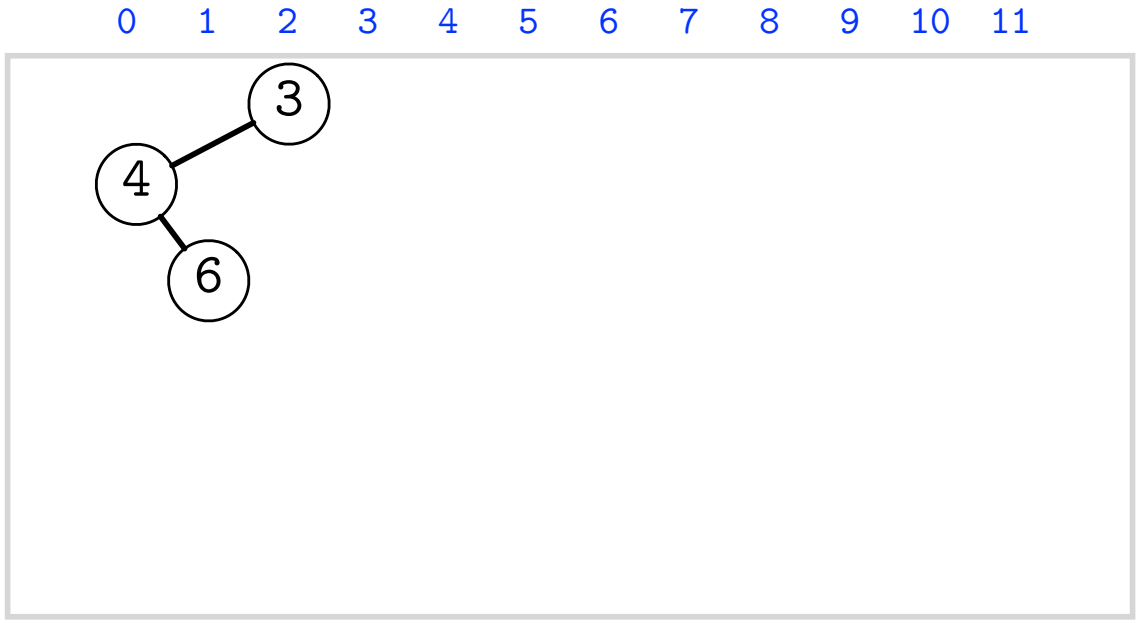
											1	
i	=	0	1	2	3	4	5	6	7	8	9	0	1
A	=	4	6	3	5	1	4	6	4	5	2	6	3

(1) Build Cartesian Tree

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

											1	
i	=	0	1	2	3	4	5	6	7	8	9	0	1
A	=	4	6	3	5	1	4	6	4	5	2	6	3

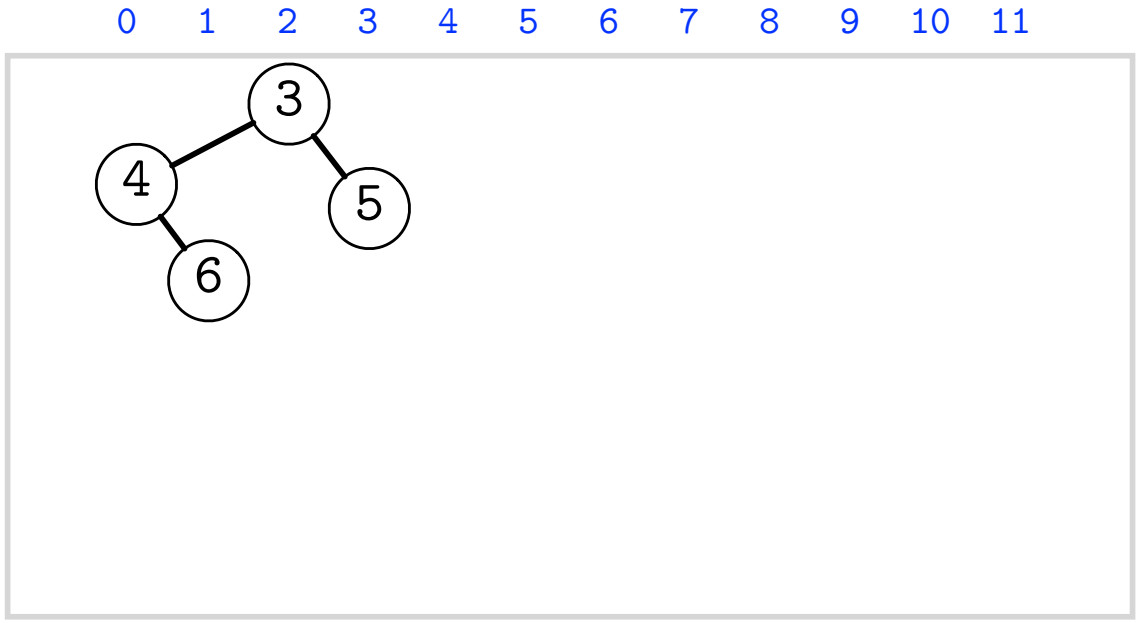
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

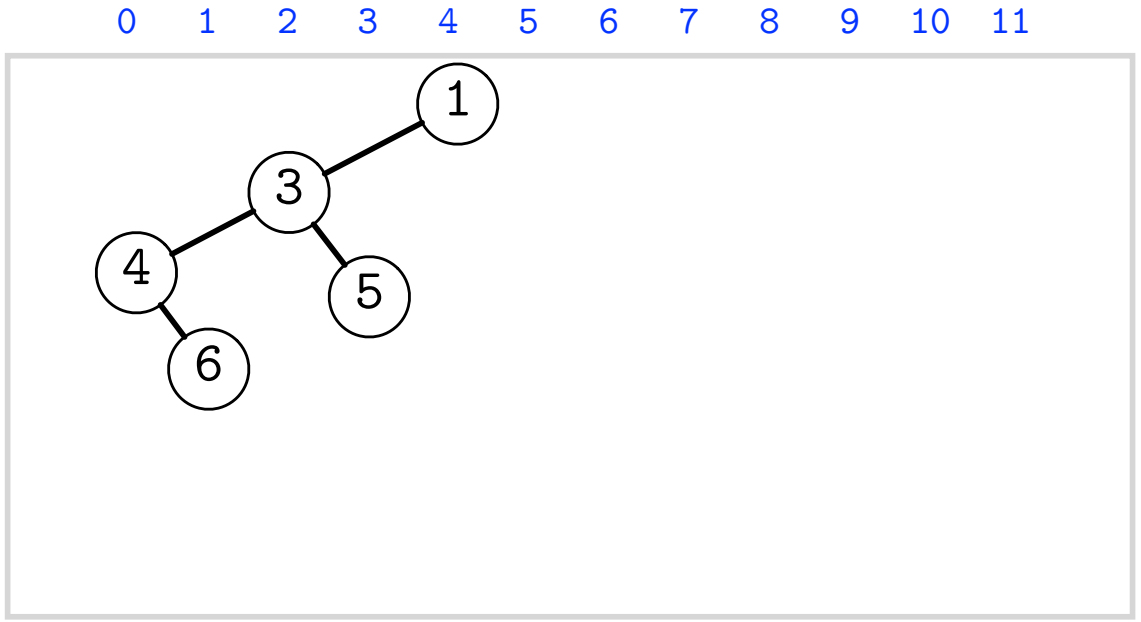
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

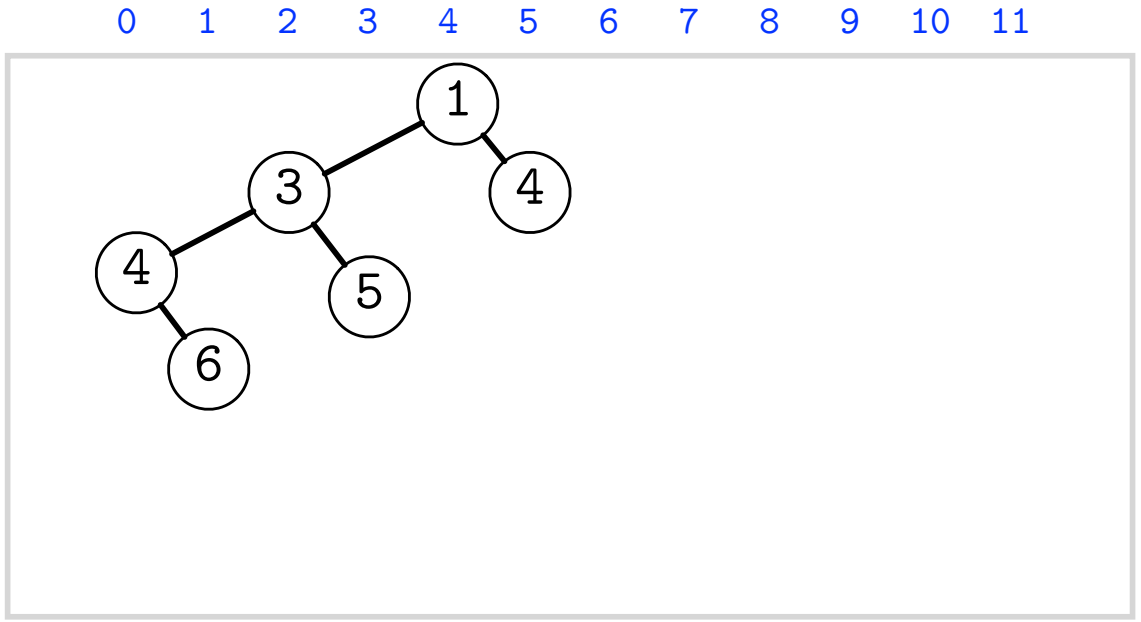
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

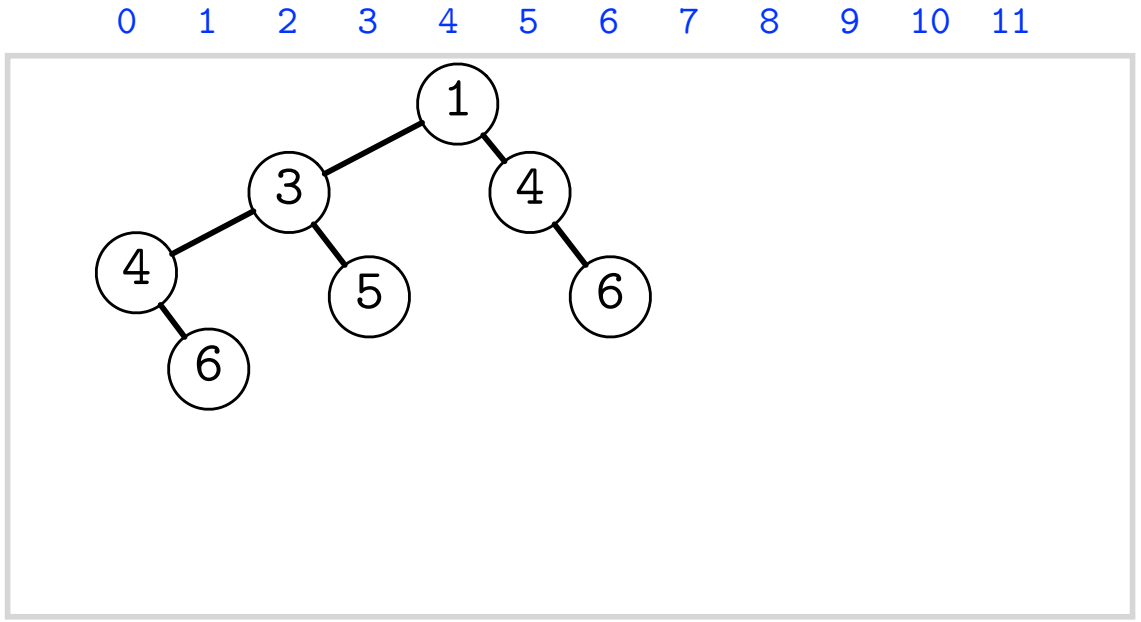
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

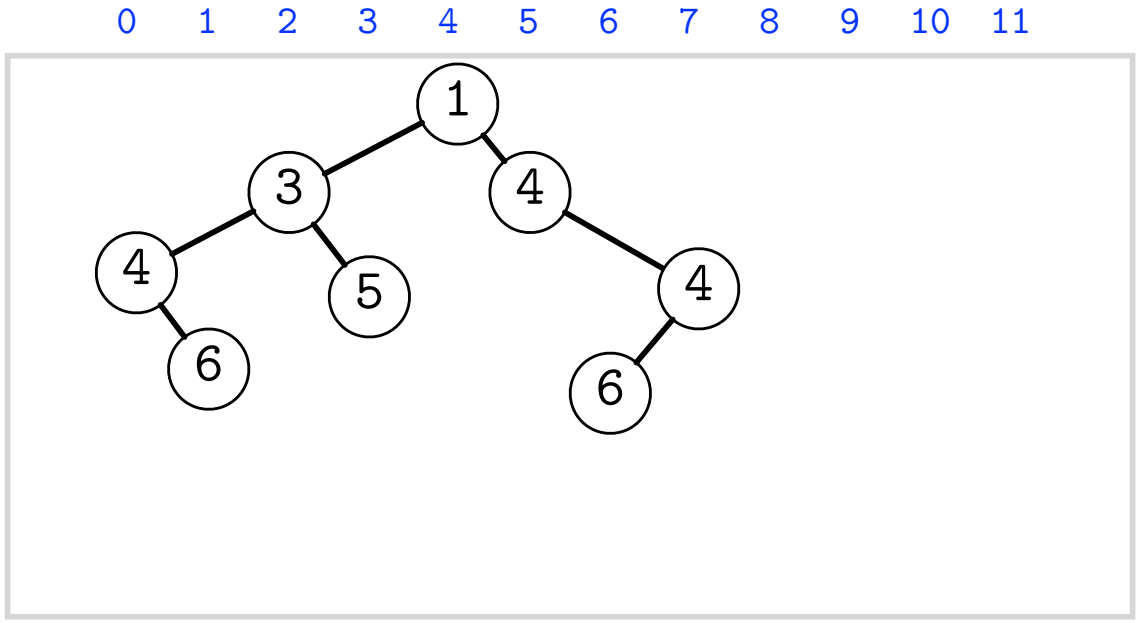
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

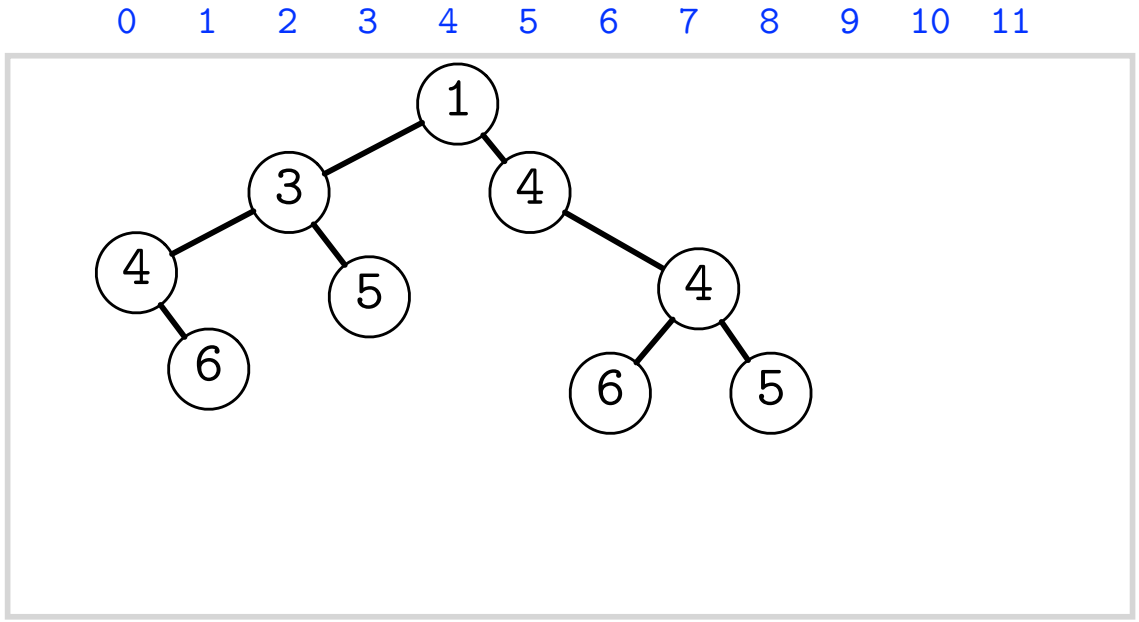
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

(1) Build Cartesian Tree

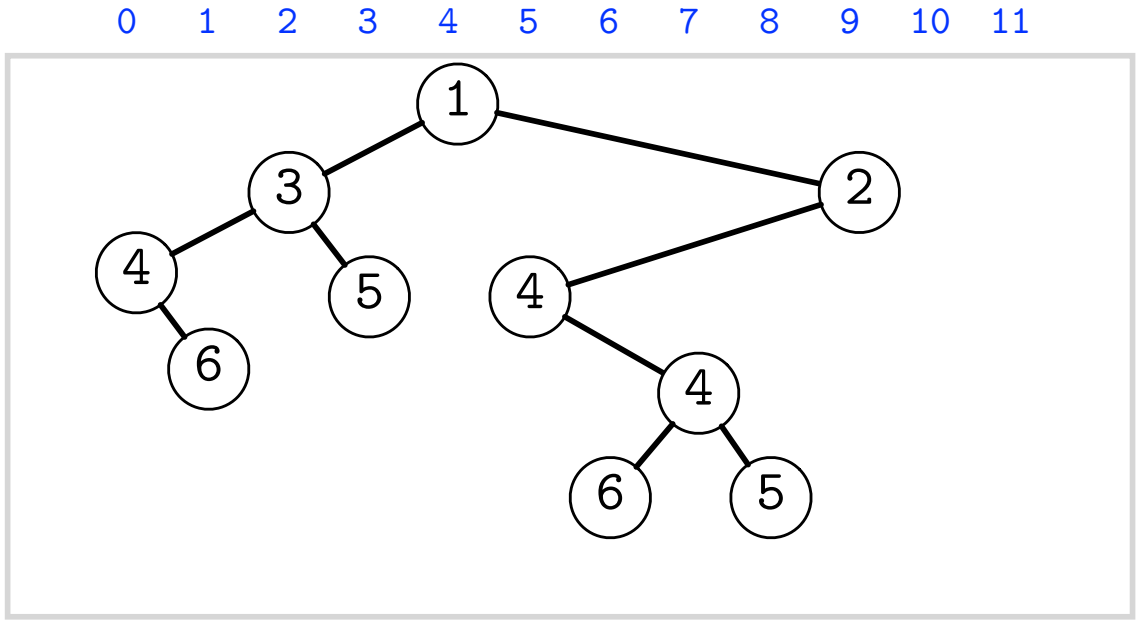


$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

Karlsruhe Institute of Technology

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

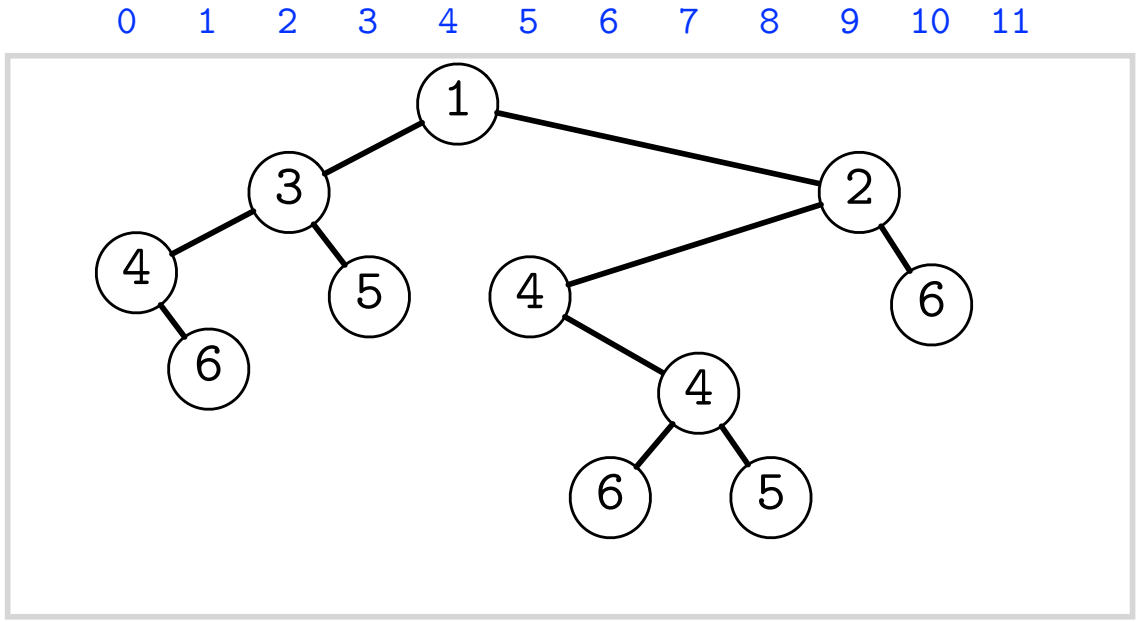
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

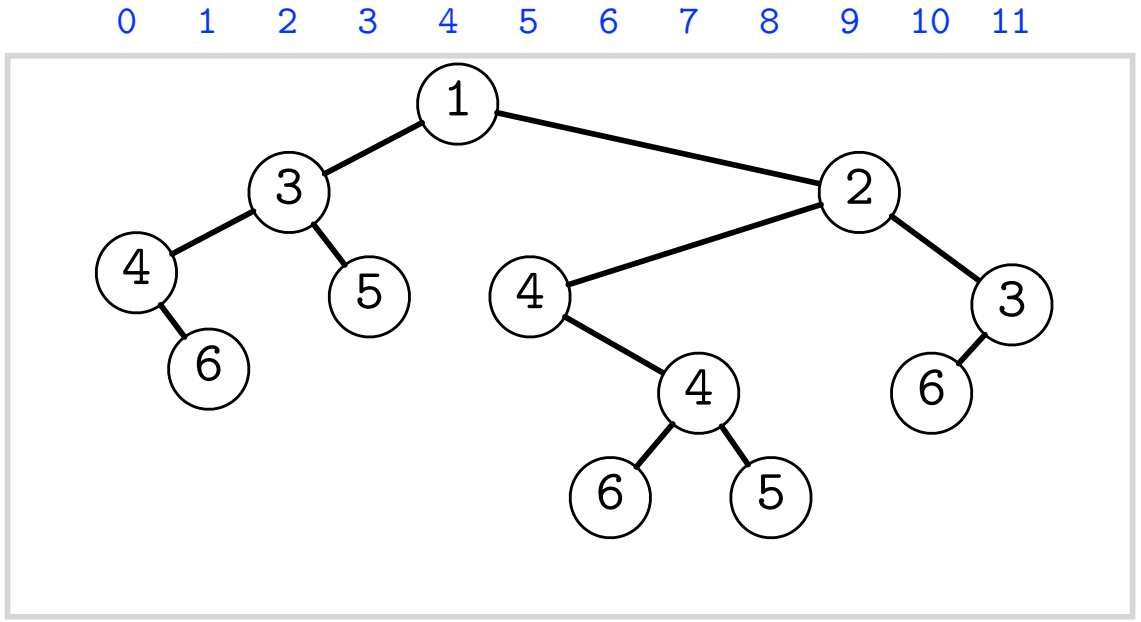
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

```
                                  1
i = 0 1 2 3 4 5 6 7 8 9 0 1
A = 4 6 3 5 1 4 6 4 5 2 6 3
```

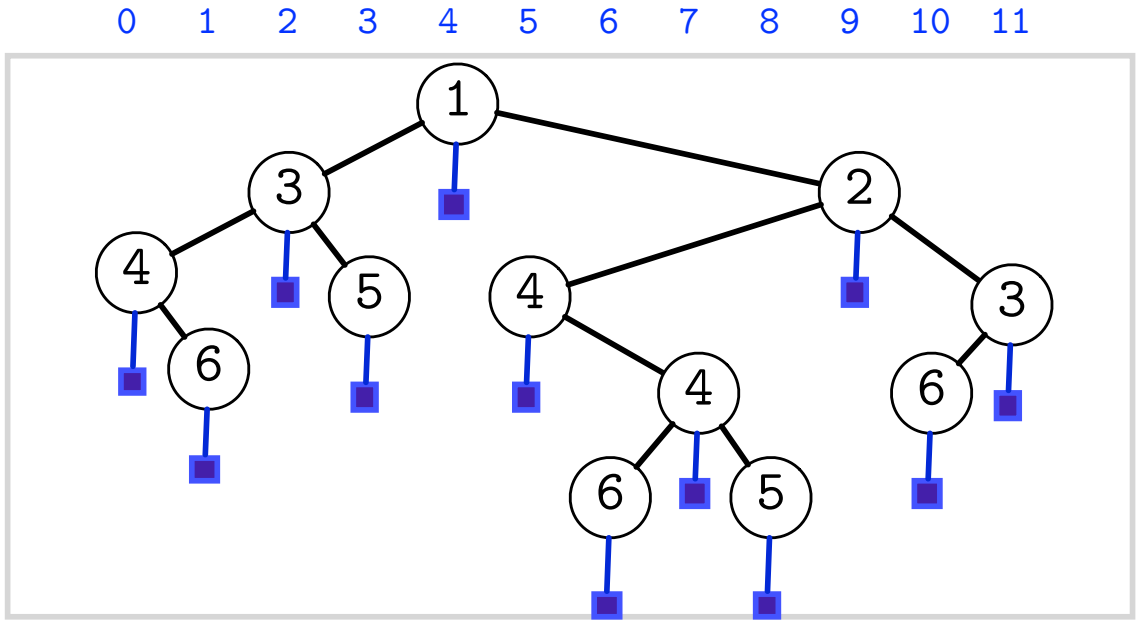
(1) Build Cartesian Tree



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

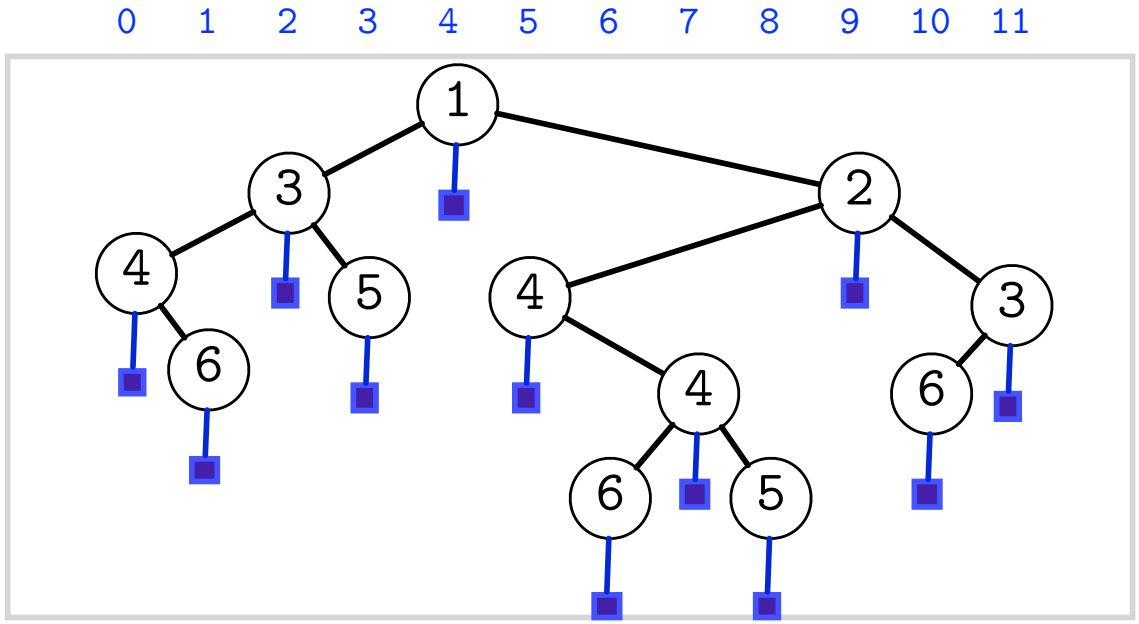
											1	
i	=	0	1	2	3	4	5	6	7	8	9	0	1
A	=	4	6	3	5	1	4	6	4	5	2	6	3

(2) Add a leaf to each node



$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

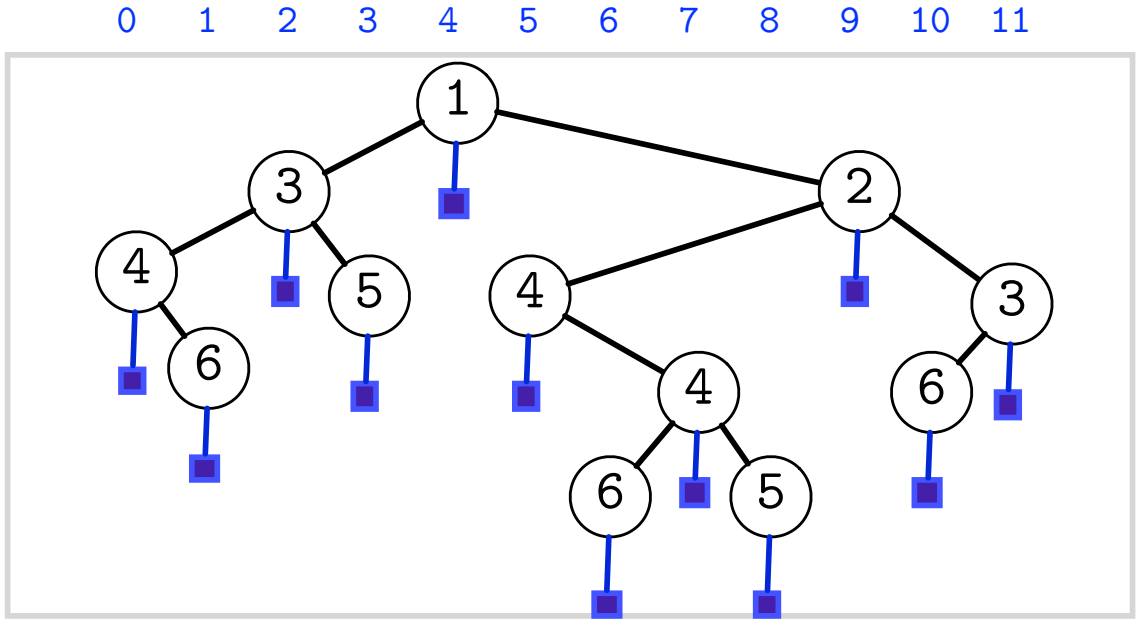


(3) DFS traversal to construct balanced parentheses sequence

$BP_{ext} =$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

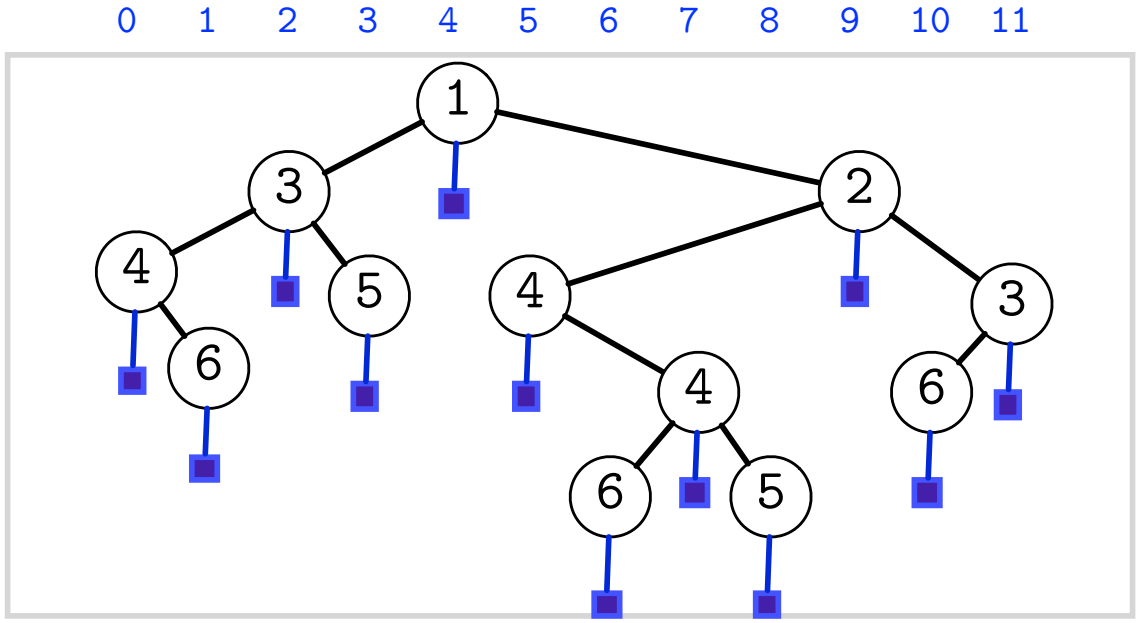


(3) DFS traversal to construct balanced parentheses sequence

$BP_{ext} = \overset{1}{(}$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

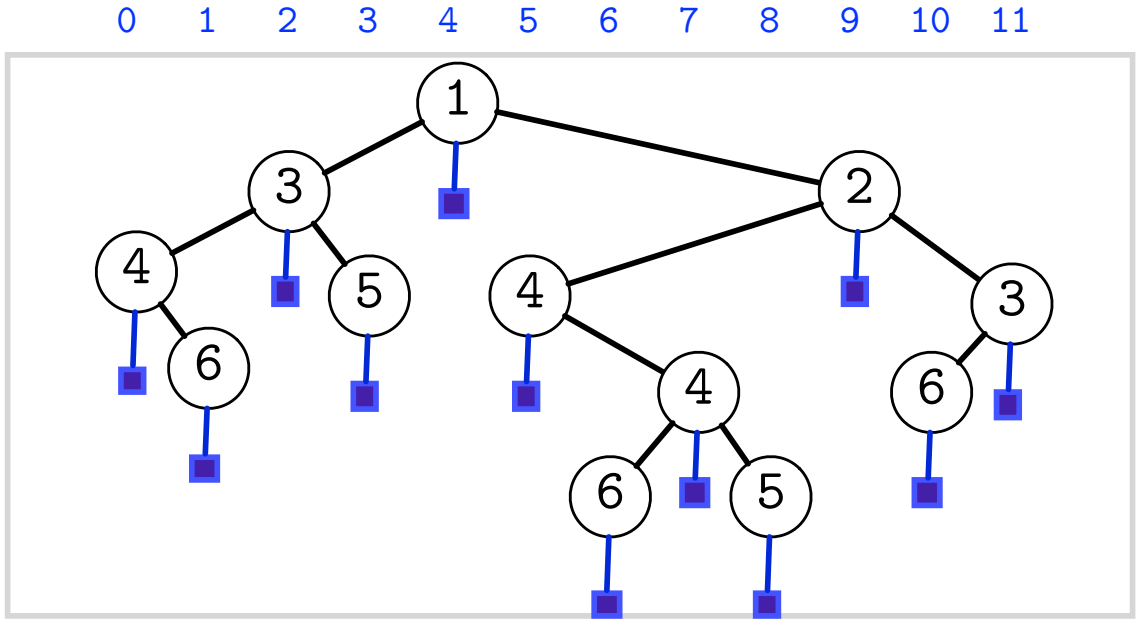


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1}{(}\overset{3}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

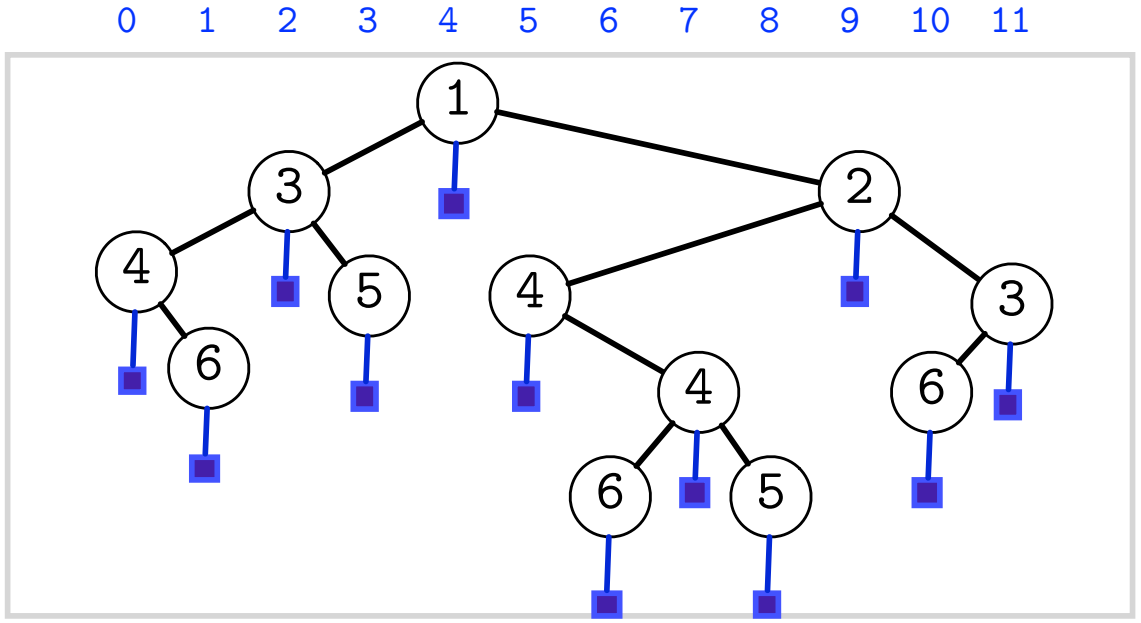


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(\,(\,(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

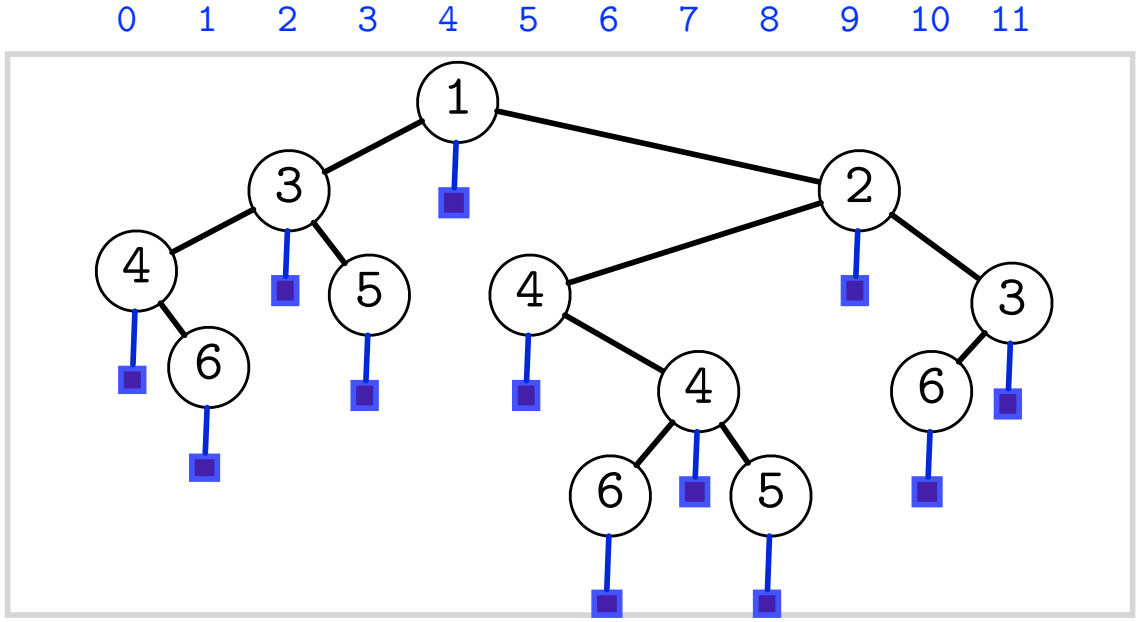


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(\,(\,(\,(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



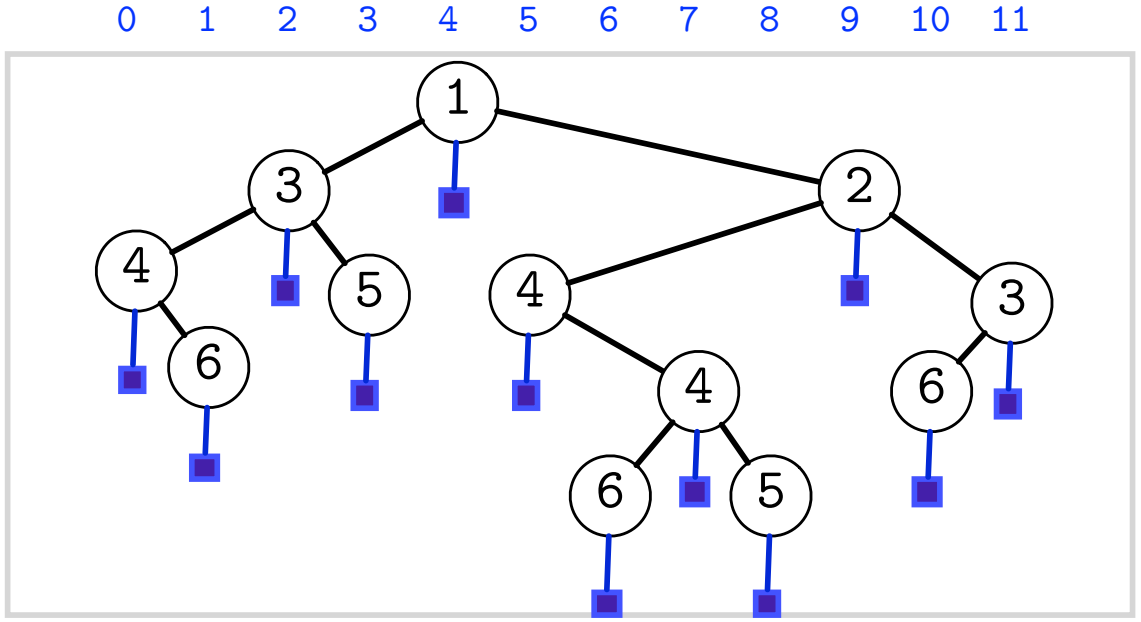
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(\,(\,(\,(\,)\,)}$$

0

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



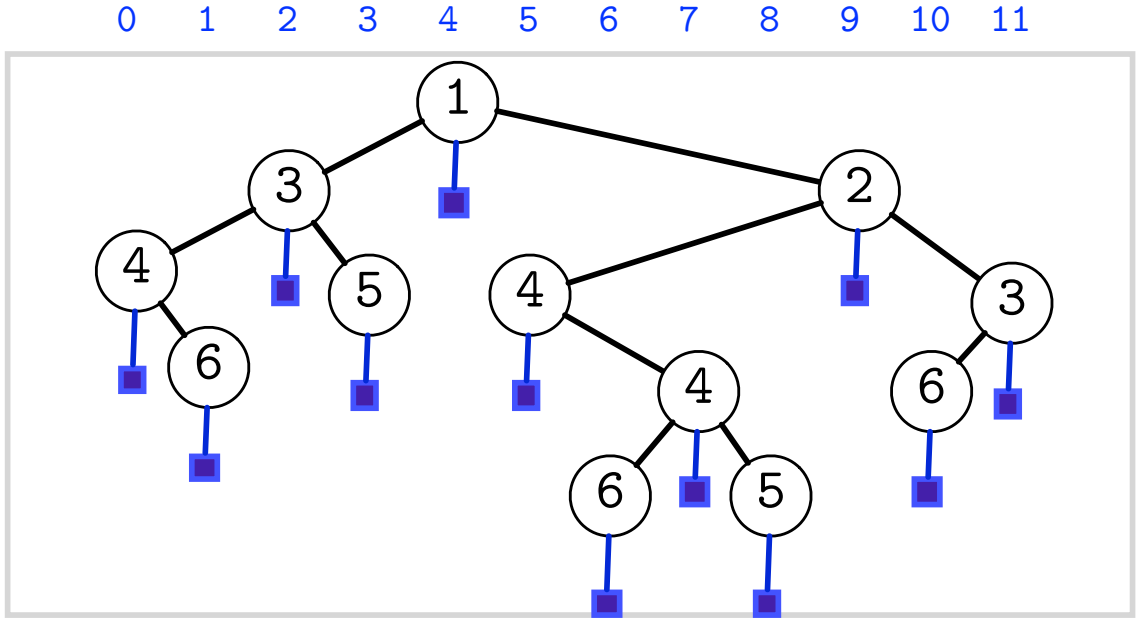
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,(\,(\,(\,)\,(\,)\,)\,)\,\overset{6}{(}$$

0

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



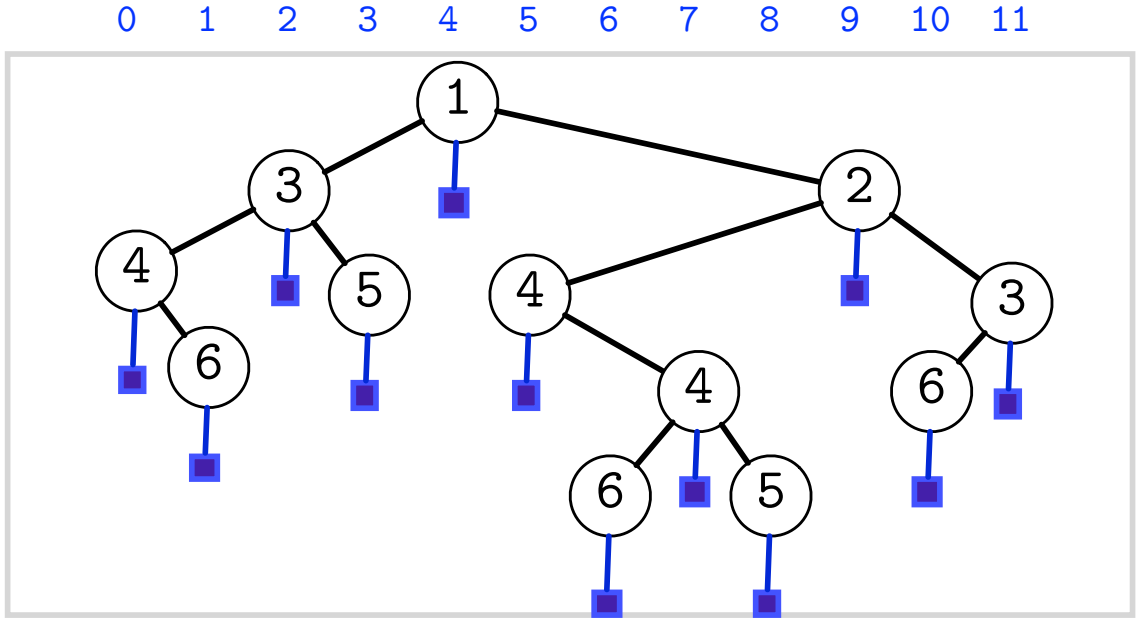
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,(\,(\,(\,)\,(\,\overset{6}{(}$$

0

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



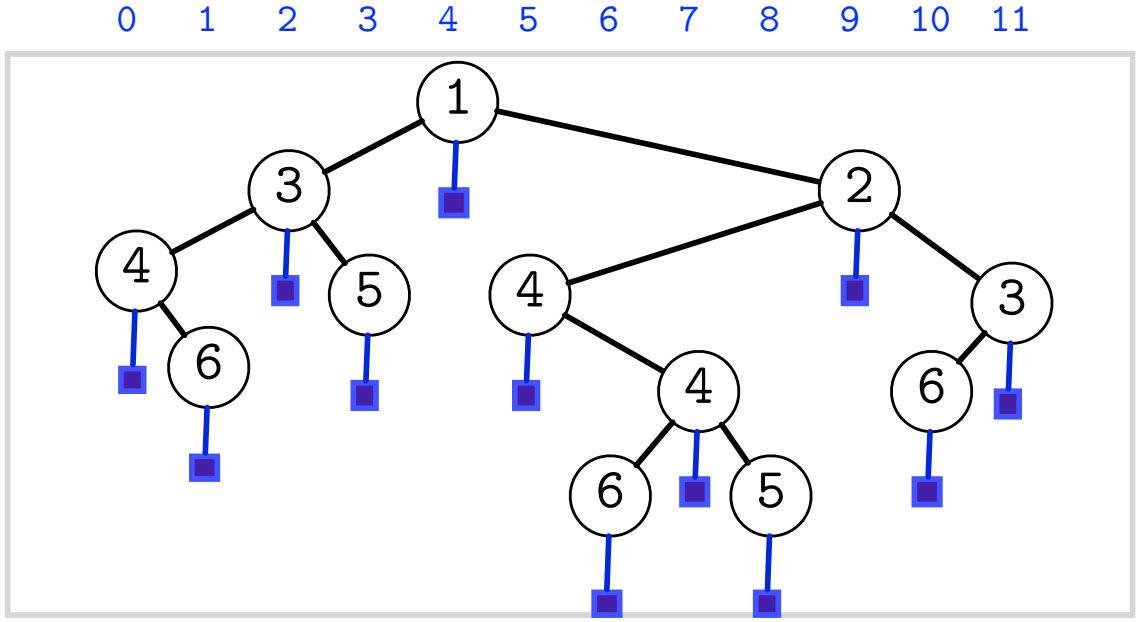
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,(\,(\,(\,)\,(\,\overset{6}{(}\,)$$

0 1

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

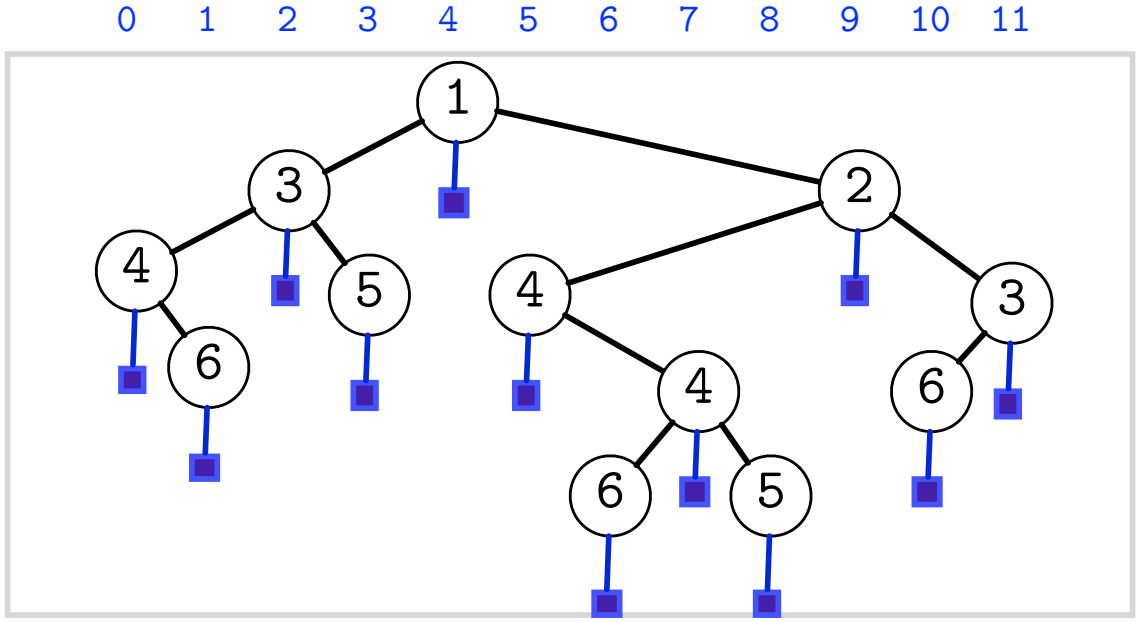


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{0}{(\,)}\,\overset{6}{(}\,\overset{1}{(\,)}\,\overset{6}{)}\,)$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

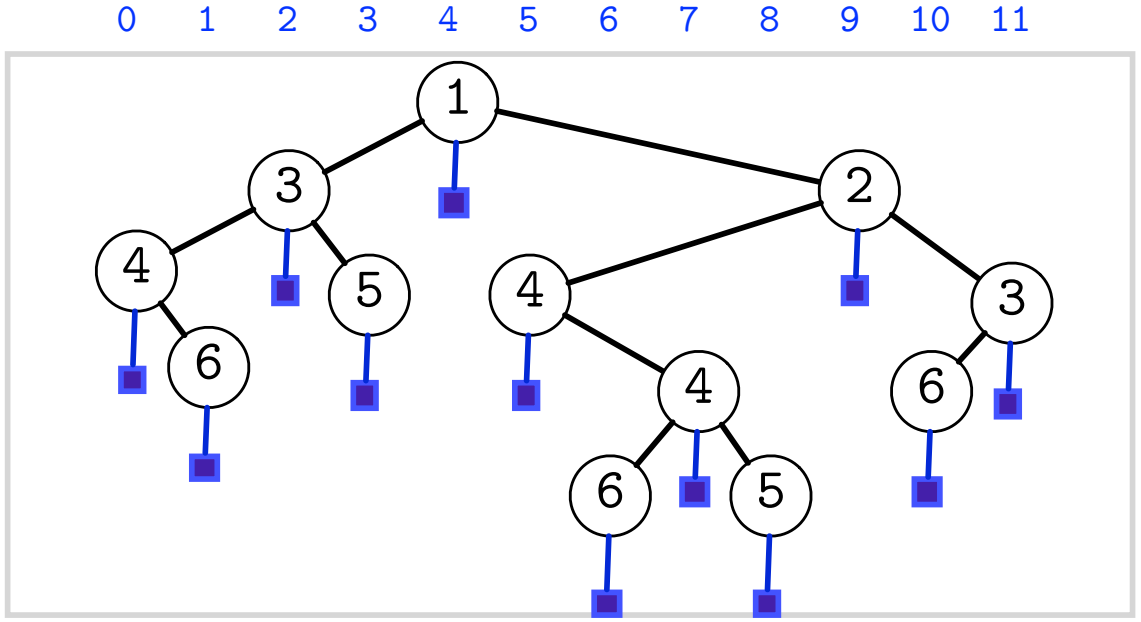


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{0}{(\,)}\,\overset{6}{(}\,\overset{1}{(\,)}\,\overset{6\ 4}{)}\,)\,)$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



(3) DFS traversal to construct balanced parentheses sequence

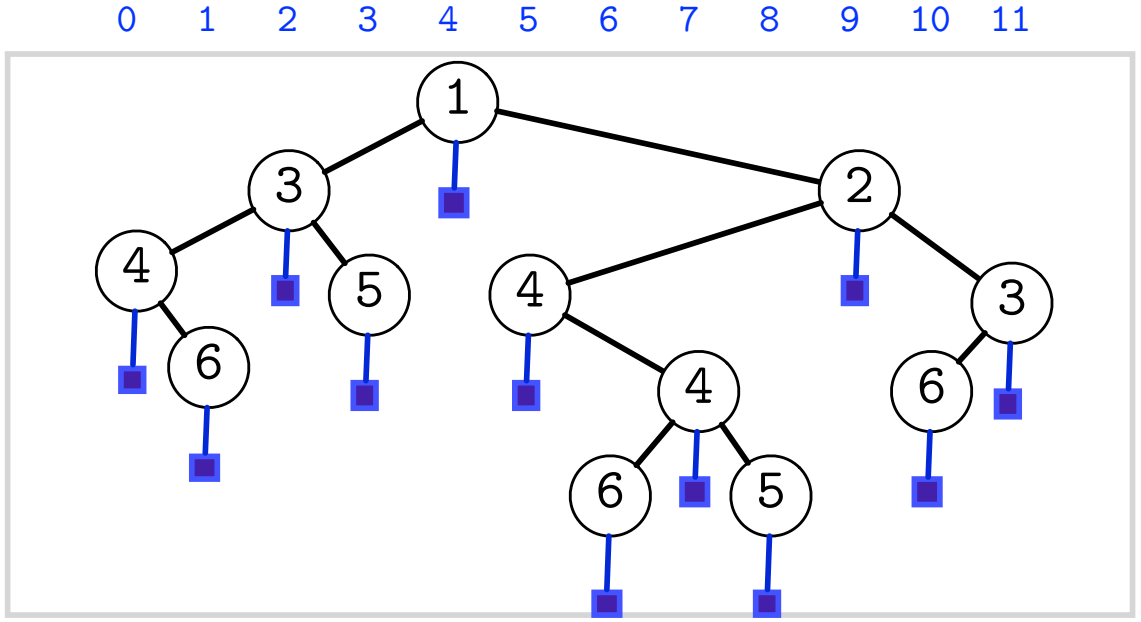
$$BP_{ext} = (((()(())))($$

1 3 4 6 6 4

0 1

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

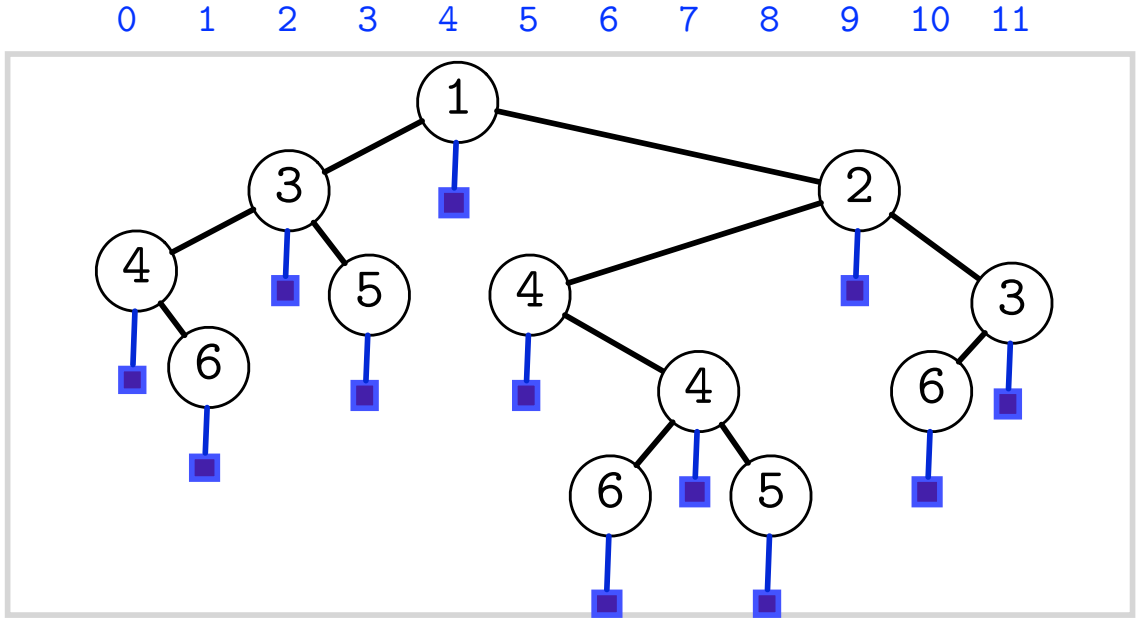


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{0}{(\,)}\,\overset{6}{(}\,\overset{1}{(\,)}\,\overset{}{)}\,\overset{6\ 4}{)}\,\overset{}{)}\,\overset{2}{(\,)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



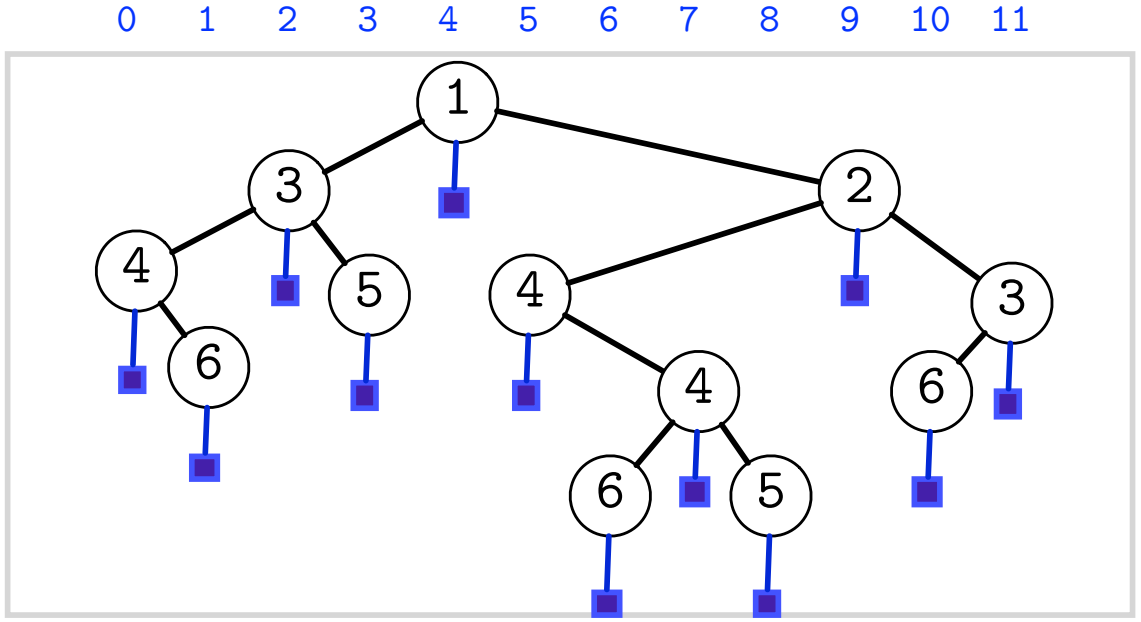
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,(\,(\,\overset{}{(}\,)\,\overset{6}{(}\,\overset{}{(}\,)\,)\,)\,\overset{6\ 4}{(}\,)\,\overset{5}{(}$$

0 1 2

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

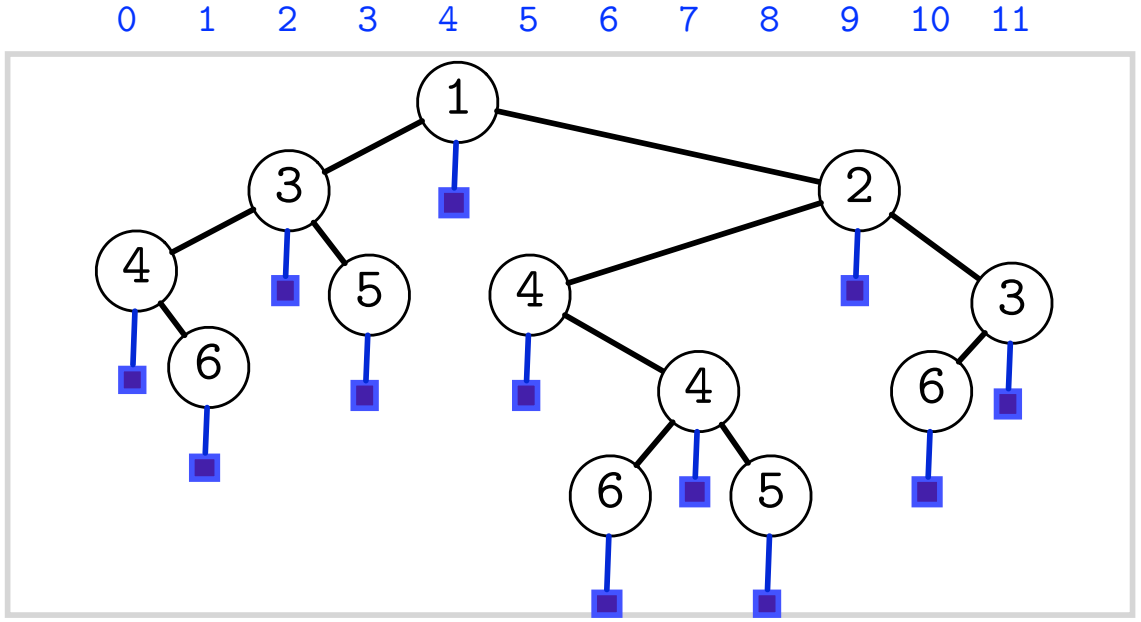


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{0}{)}\,\overset{}{(}\,\overset{6}{(}\,\overset{1}{)}\,\overset{}{)}\,\overset{6\ 4}{)}\,\overset{2}{(}\,\overset{}{)}\,\overset{5}{(}\,\overset{}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

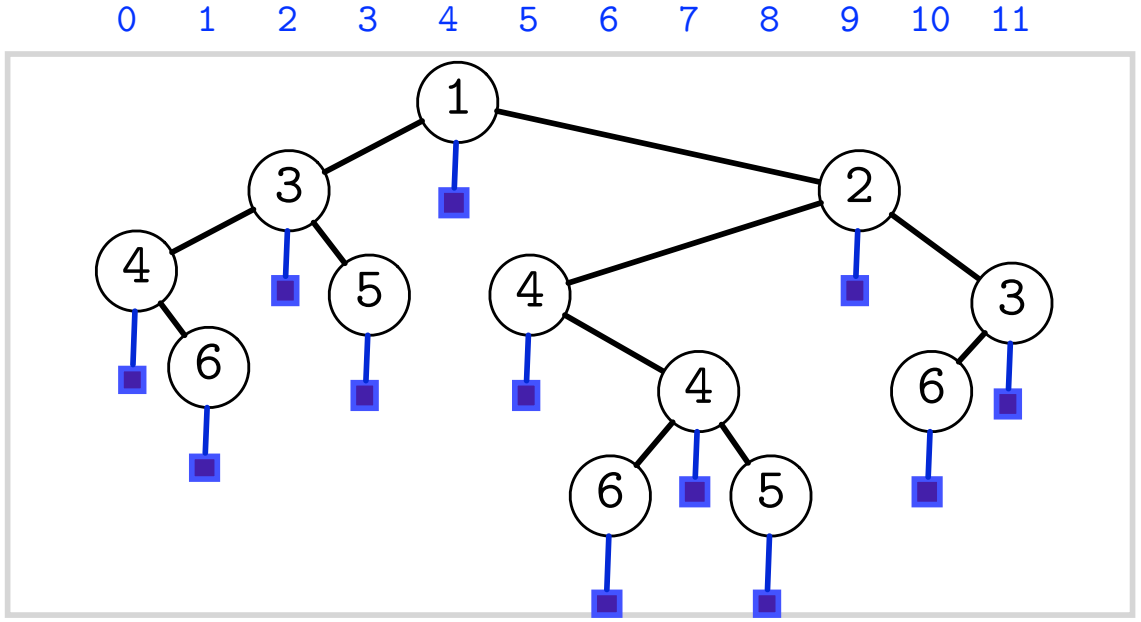


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{0}{)}\,\overset{}{(}\,\overset{1}{)}\,\overset{6}{)}\,\overset{6\ 4}{)}\,\overset{2}{(}\,\overset{}{)}\,\overset{5}{(}\,\overset{3}{(}\,\overset{}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

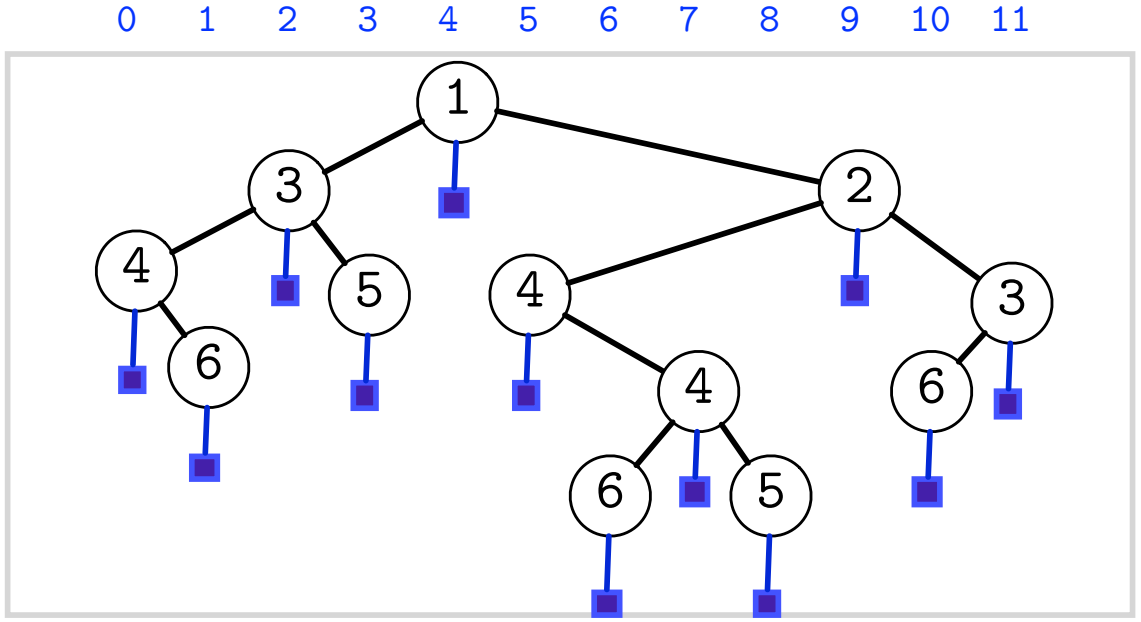


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{0}{()}\overset{6}{(}\overset{1}{()}\overset{6\ 4}{)}\overset{}{)}\overset{2}{()}\overset{5}{(}\overset{3}{()}\overset{5}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

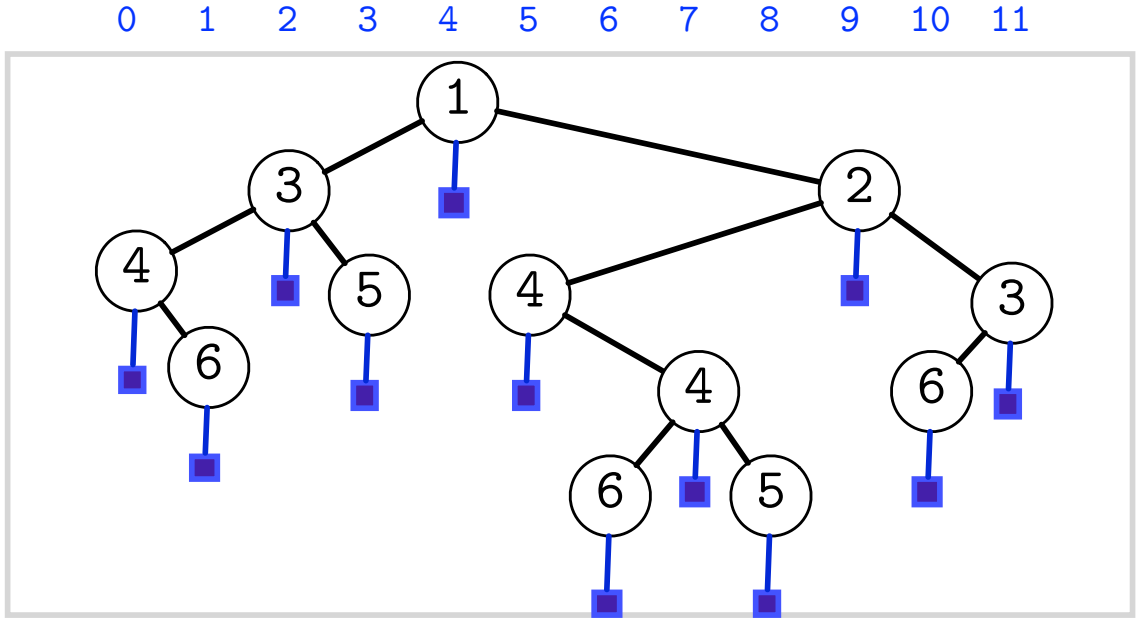


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{(}\overset{0}{)}\overset{}{(}\overset{6}{(}\overset{1}{)}\overset{}{)}\overset{6\ 4}{)}\overset{2}{(}\overset{}{)}\overset{5}{(}\overset{}{(}\overset{3}{)}\overset{}{)}\overset{5\ 3}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



(3) DFS traversal to construct balanced parentheses sequence

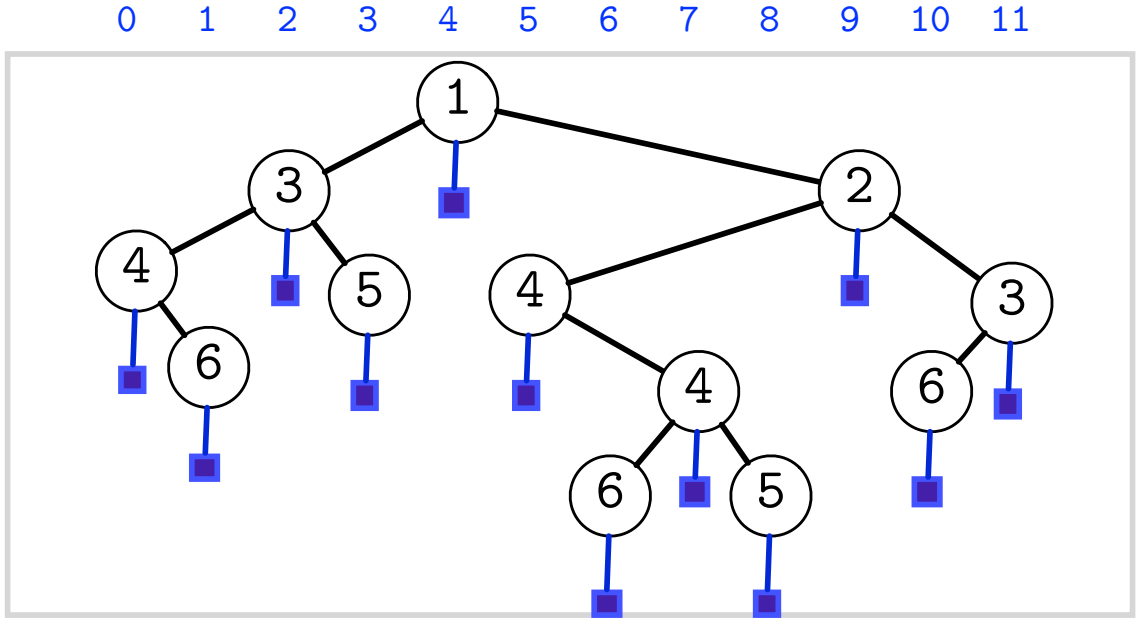
$$BP_{ext} = (((()(()))()(()))($$

1 3 4 6 6 4 5 5 3

0 1 2 3

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

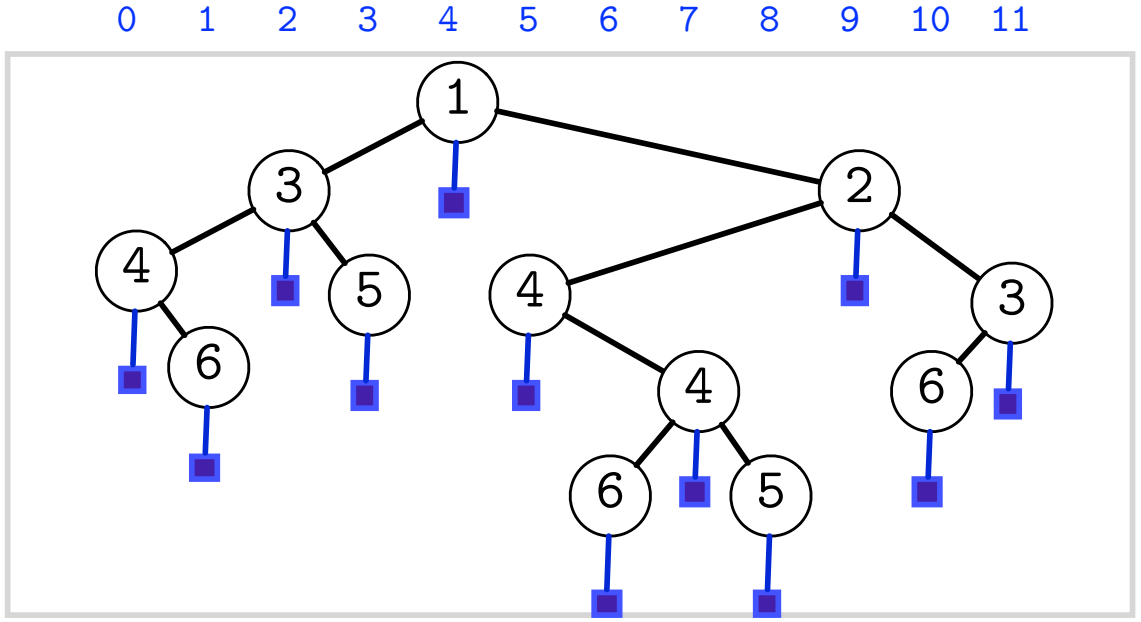


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\,\overset{}{(}\,\overset{}{(}\,\overset{0}{(\,)}\,\overset{6}{(}\,\overset{1}{(\,)}\,\overset{6\ 4}{)}\,\overset{2}{(\,)}\,\overset{5}{(}\,\overset{3}{(\,)}\,\overset{5\ 3}{)}\,\overset{4}{(\,)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



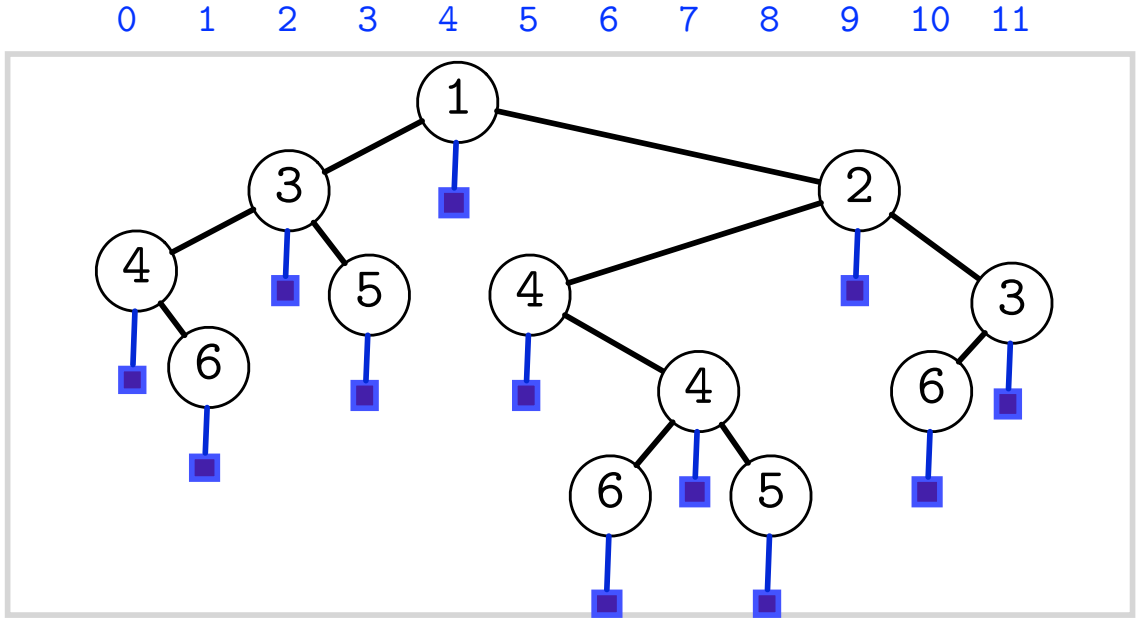
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1}{(}\overset{3}{(}\overset{4}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{(}\overset{}{)}\overset{6}{)}\overset{4}{)}\overset{}{(}\overset{}{)}\overset{5}{(}\overset{}{(}\overset{}{)}\overset{5}{)}\overset{3}{)}\overset{}{(}\overset{}{)}\overset{2}{(}$$

$$\qquad\qquad\quad 0 \quad 1 \quad\quad 2 \quad 3 \quad\quad 4$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



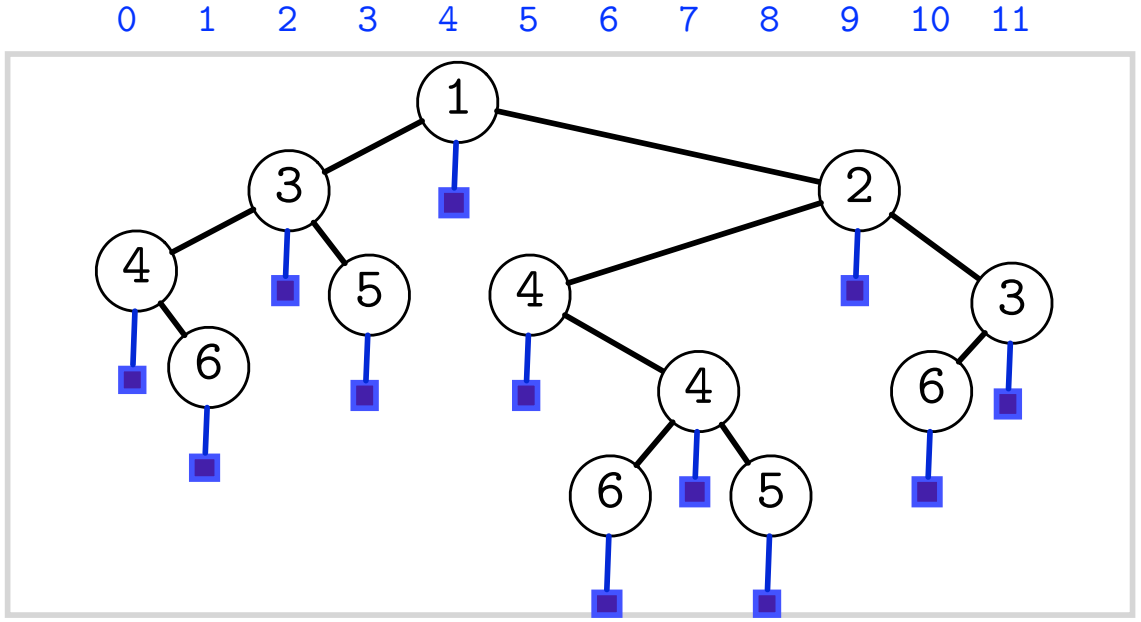
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{}{)}\overset{5}{(}\overset{}{(}\overset{5\ 3}{)}\overset{}{)}\overset{2\ 4}{(}\overset{}{)}\overset{}{(}\overset{}{(}$$

$$\phantom{BP_{ext} = (((}\ 0\ \ 1\ \ \ 2\ \ 3\ \ \ 4$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



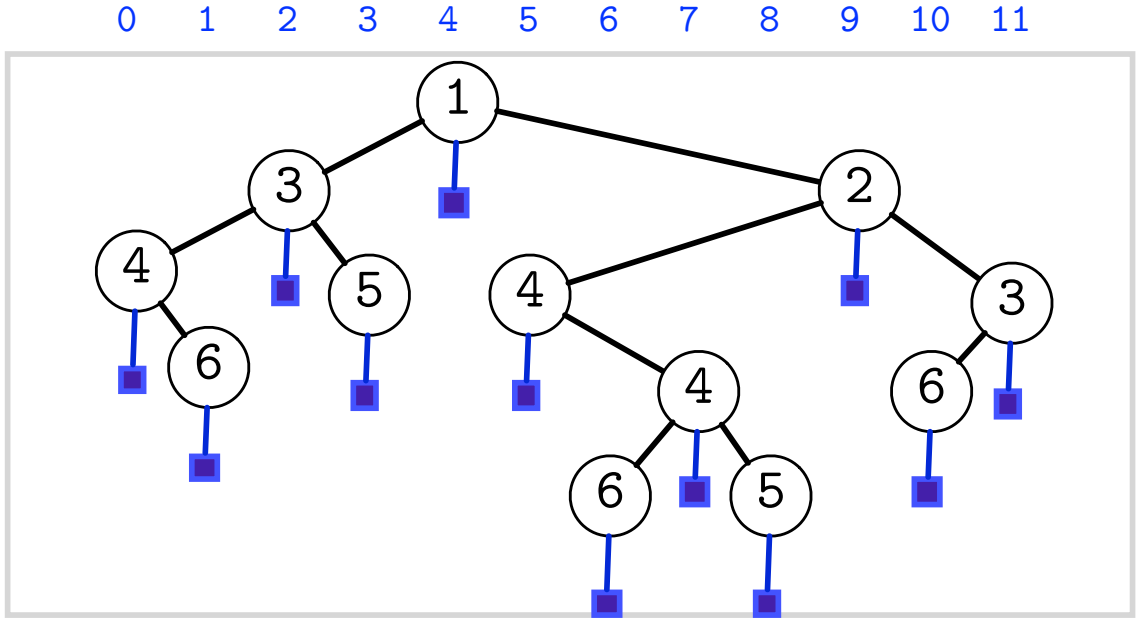
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{}{)}\overset{5}{(}\overset{}{(}\overset{5\ 3}{)}\overset{}{)}\overset{}{)}\overset{2\ 4}{(}\overset{}{(}\overset{}{(}$$

$$0 \quad 1 \quad 2 \quad 3 \quad 4$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

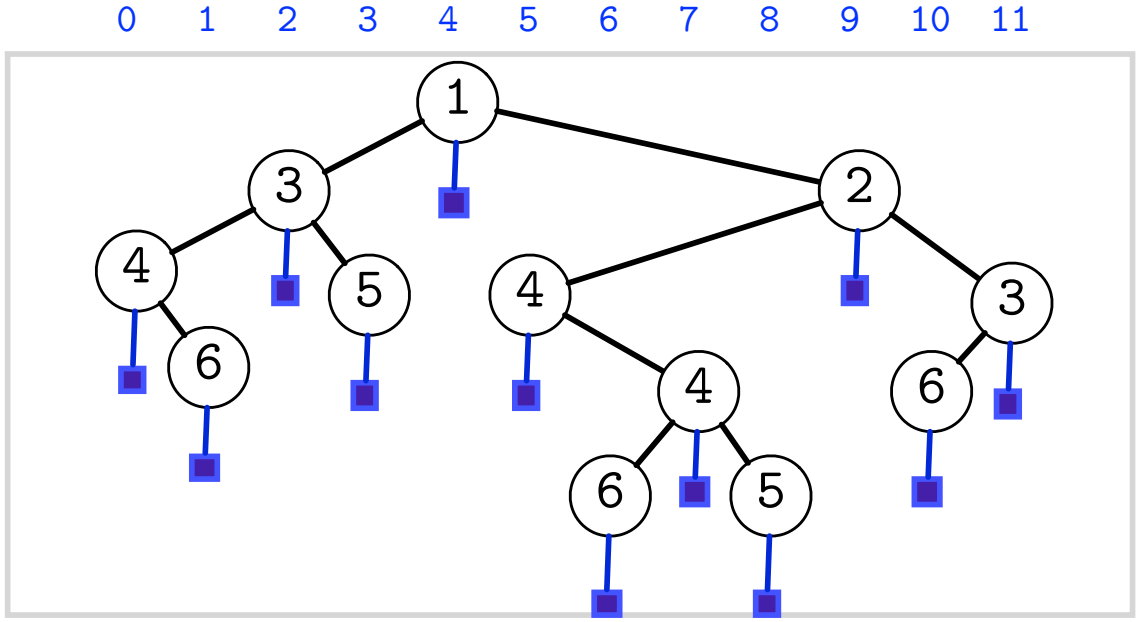


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{0}{()}\overset{6}{(}\overset{1}{()}\overset{6\ 4}{)}\overset{2}{()}\overset{5}{(}\overset{3}{()}\overset{5\ 3}{)}\overset{4}{()}\overset{2\ 4}{(}\overset{}{(}\overset{5}{()}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



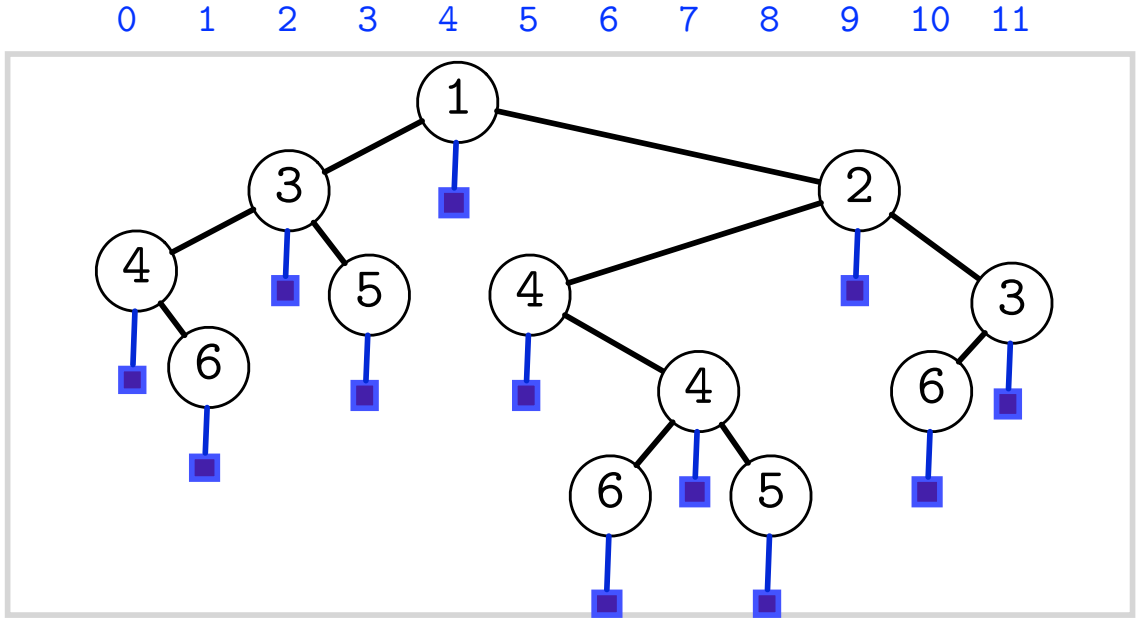
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{}{)}\overset{5}{(}\overset{}{(}\overset{5\ 3}{)}\overset{}{)}\overset{}{)}\overset{2\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{4}{(}$$

$$\qquad\qquad\quad 0 \quad 1 \quad\quad 2 \quad 3 \quad\quad 4 \quad\quad 5$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



(3) DFS traversal to construct balanced parentheses sequence

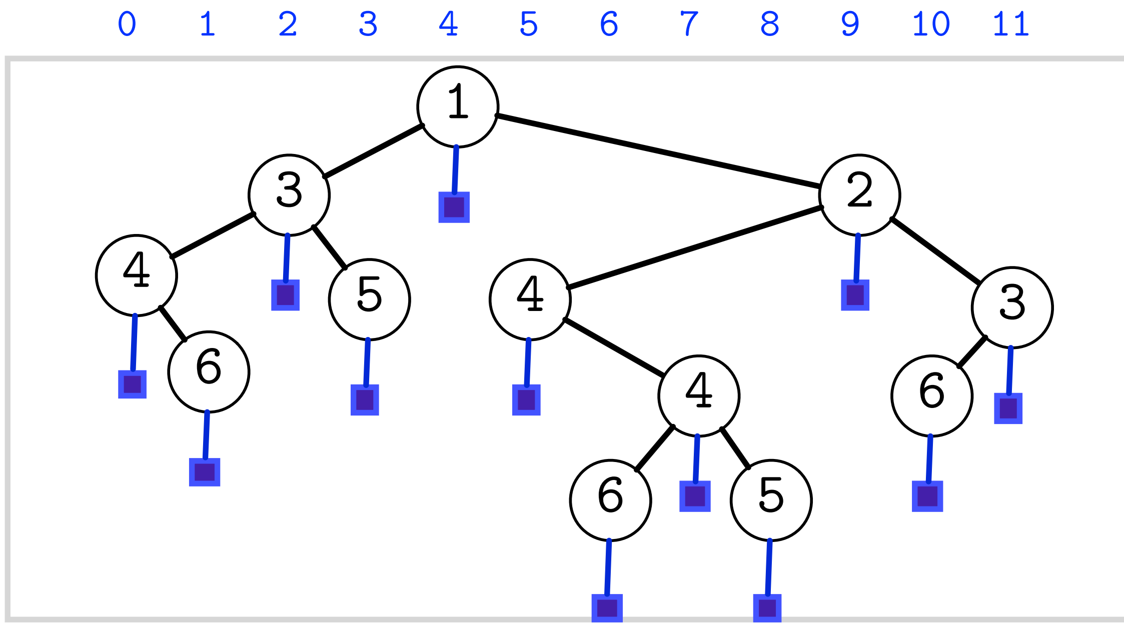
$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{(}\overset{6}{)}\overset{}{(}\overset{}{(}\overset{6\ 4}{)}\overset{}{)}\overset{}{(}\overset{5}{)}\overset{}{(}\overset{}{(}\overset{5\ 3}{)}\overset{}{)}\overset{}{)}\overset{2\ 4}{(}\overset{}{(}\overset{}{(}\overset{4\ 6}{)}\overset{}{(}\overset{}{(}$$

$$\qquad\qquad\qquad\quad 0 \quad 1 \quad\quad 2 \quad 3 \quad\quad 4 \quad\quad 5$$

$\langle O(n), O(1)\rangle$ solution ($4n + o(n)$ bits)

Karlsruhe Institute of Technology

(2) Add a leaf to each node



(3) DFS traversal to construct balanced parentheses sequence

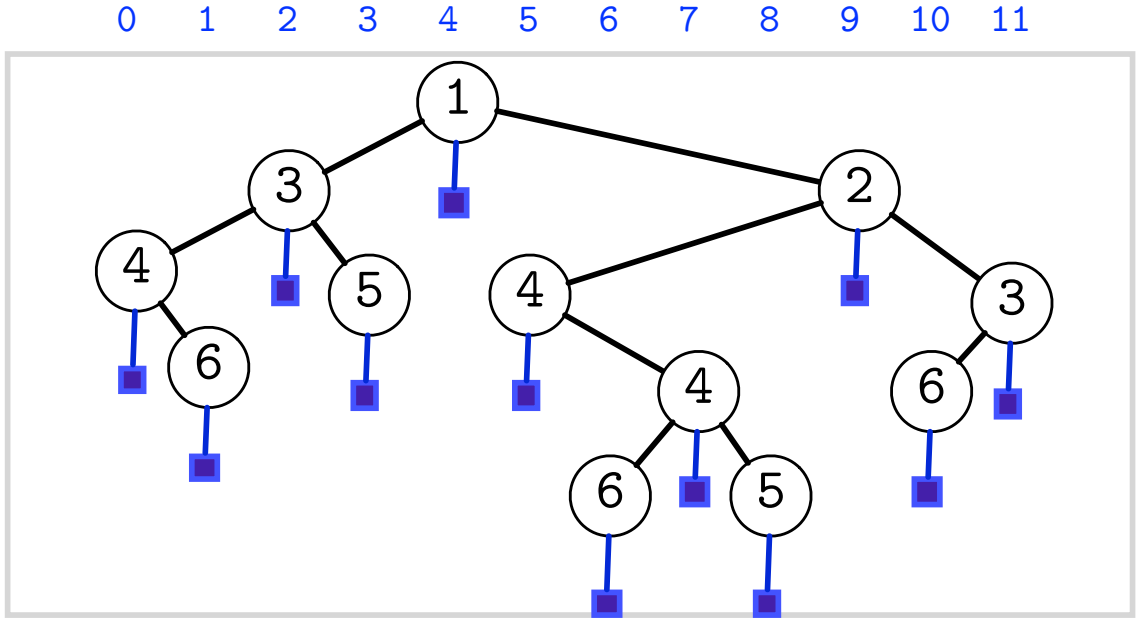
1 3 4 6 6 4 5 5 3 2 4 4 6

BP_{ext} = (((( ) (( ))) ( ) (( ))) ( ) ((( ) (((

0 1 2 3 4 5

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



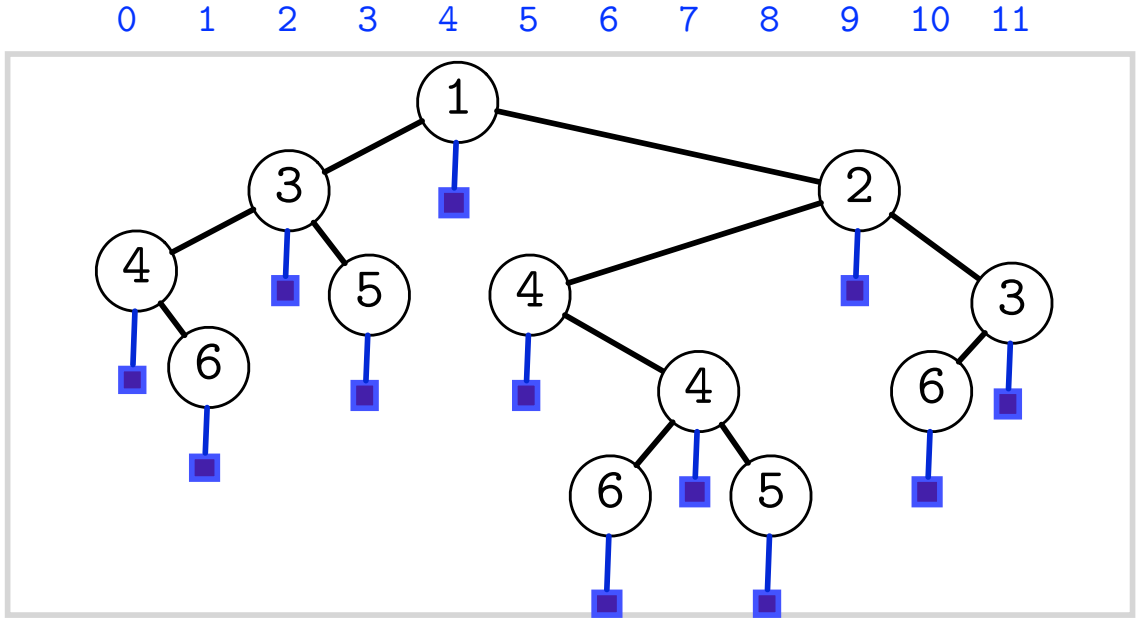
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{}{)}\overset{}{(}\overset{5}{(}\overset{}{)}\overset{5\ 3}{)}\overset{}{)}\overset{}{(}\overset{2\ 4}{(}\overset{}{(}\overset{}{)}\overset{4\ 6}{(}\overset{}{(}\overset{}{)}$$

$$\qquad\qquad\quad 0 \quad 1 \quad\quad 2 \quad 3 \quad\quad 4 \quad\quad 5 \quad\quad 6$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



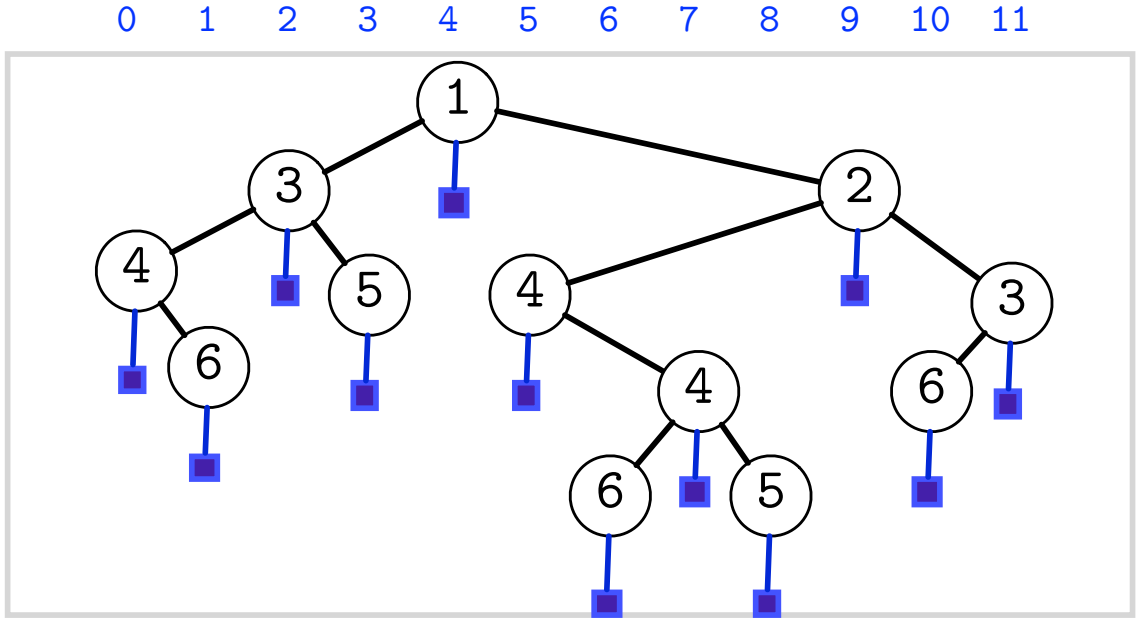
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1}{(}\overset{3}{(}\overset{4}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{(}\overset{}{)}\overset{6}{)}\overset{4}{)}\overset{}{(}\overset{5}{)}\overset{}{(}\overset{}{(}\overset{}{)}\overset{5}{)}\overset{3}{)}\overset{}{(}\overset{2}{)}\overset{4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{4}{(}\overset{6}{(}\overset{}{(}\overset{}{)}\overset{6}{)}$$

$$\qquad\qquad\qquad\qquad\qquad\quad 0 \quad 1 \quad\quad 2 \quad 3 \quad\quad 4 \quad\quad 5 \quad\quad 6$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



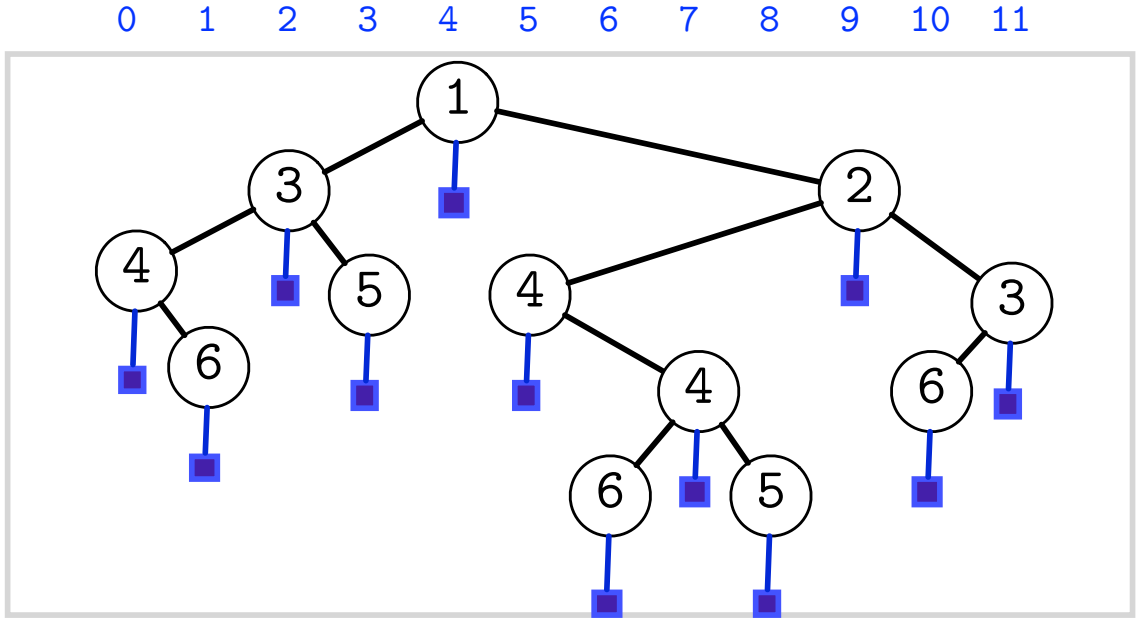
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{5}{)}\overset{}{(}\overset{}{(}\overset{5\ 3}{)}\overset{}{)}\overset{2\ 4}{(}\overset{}{(}\overset{4\ 6}{)}\overset{6}{(}\overset{}{(}\overset{}{)}\overset{}{)}\overset{}{(}$$

$$\phantom{BP_{ext} = (}\ 0\ \ 1\ \ \ 2\ \ 3\ \ \ 4\ \ \ 5\ \ \ 6$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



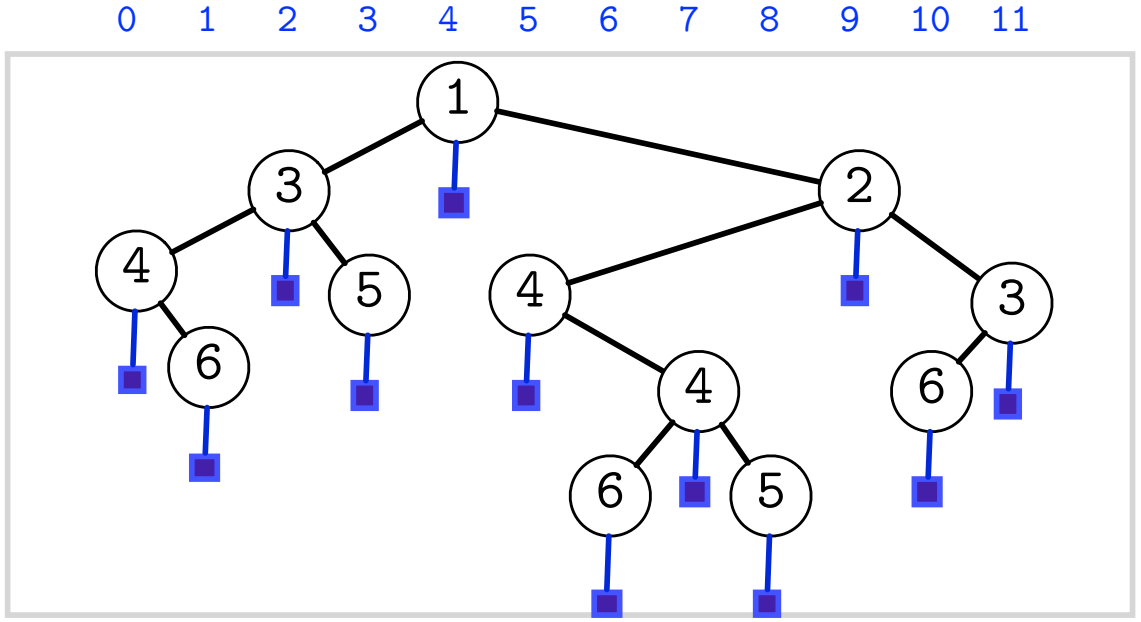
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{}{)}\overset{}{(}\overset{5}{(}\overset{}{)}\overset{5\ 3}{)}\overset{}{)}\overset{2\ 4}{(}\overset{}{(}\overset{}{)}\overset{4\ 6}{(}\overset{}{(}\overset{}{)}\overset{6}{)}\overset{}{)}\overset{}{(}\overset{}{)}$$

$$\qquad\qquad\quad 0\quad 1\quad\ 2\quad 3\quad\ 4\quad\ 5\quad\ 6\quad 7$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



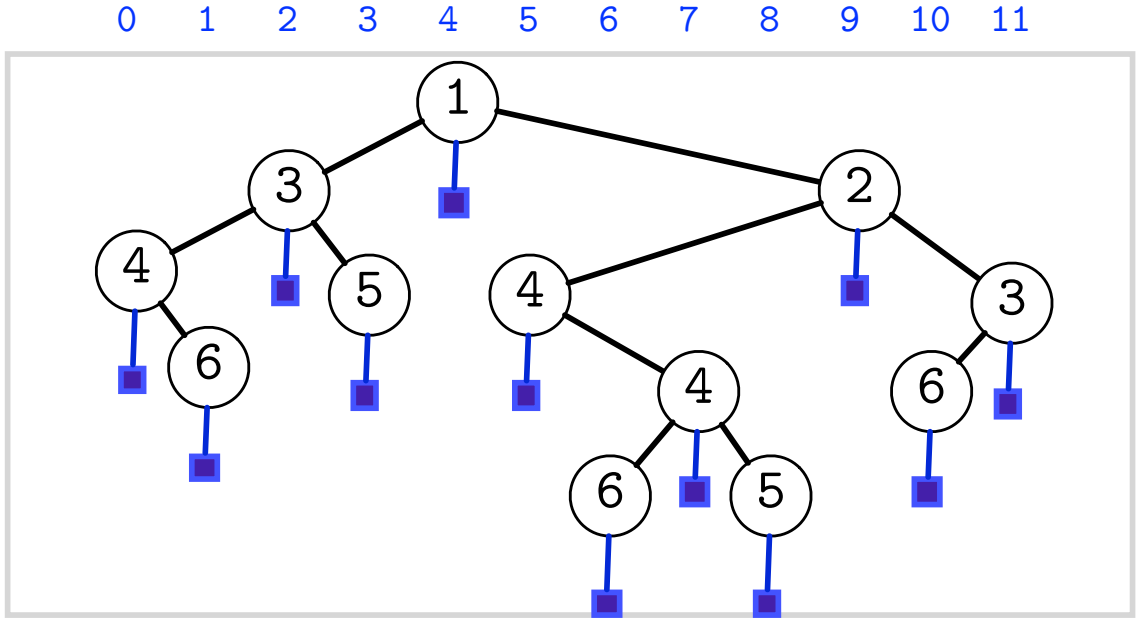
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1\ 3\ 4}{(}\overset{}{(}\overset{}{(}\overset{}{)}\overset{6}{(}\overset{}{)}\overset{6\ 4}{)}\overset{5}{)}\overset{}{(}\overset{5\ 3}{(}\overset{}{)}\overset{}{)}\overset{2\ 4}{)}\overset{}{(}\overset{}{(}\overset{}{)}\overset{4\ 6}{(}\overset{}{(}\overset{}{)}\overset{6}{)}\overset{}{(}\overset{5}{)}\overset{}{(}$$

$$\qquad\qquad\qquad\quad 0\quad 1\qquad 2\quad 3\qquad 4\qquad 5\qquad 6\quad 7$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

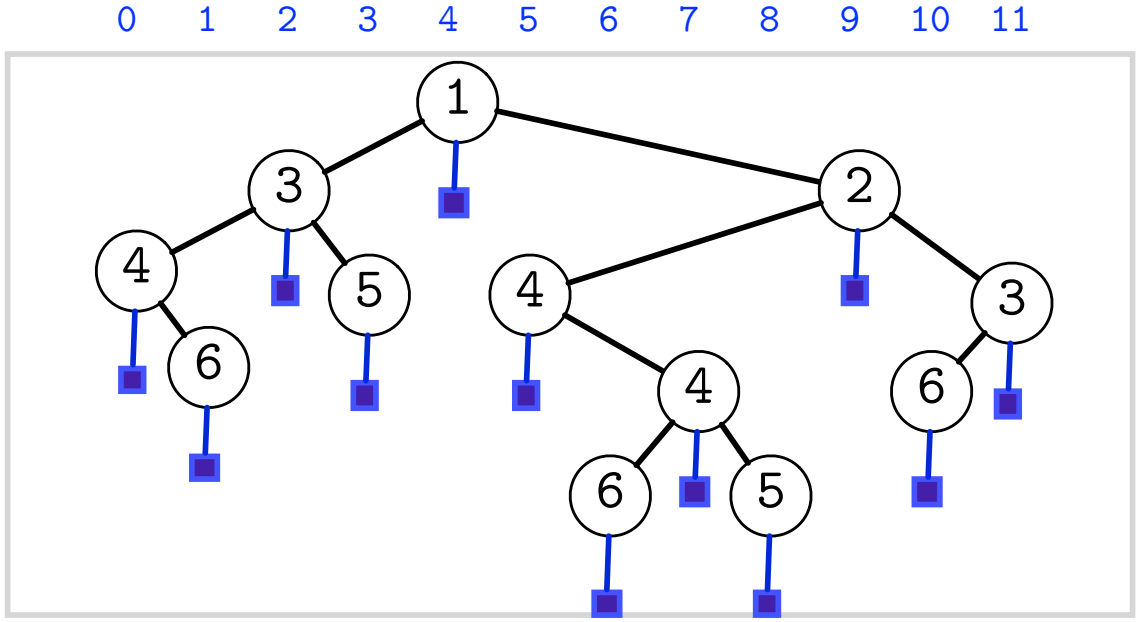


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(} \underset{\substack{1 \\ 1}}{(} \underset{\substack{6 \\ 2}}{(} \underset{\substack{6\ 4 \\ 3}}{(} \underset{\substack{5 \\ 4}}{)} \underset{\substack{5\ 3 \\ 5}}{(} \underset{\substack{2\ 4 \\ 6}}{)} \underset{\substack{4\ 6 \\ 7}}{)} \underset{\substack{6 \\ 8}}{)} \underset{\substack{5 \\ 9}}{)} \underset{\substack{ \\ 10}}{)} \underset{\substack{ \\ 11}}{(} \underset{\substack{ \\ 12}}{(} \underset{\substack{ \\ 13}}{(} \underset{\substack{ \\ 14}}{)} \underset{\substack{ \\ 15}}{(} \underset{\substack{ \\ 16}}{(} \underset{\substack{ \\ 17}}{)} \underset{\substack{ \\ 18}}{)} \underset{\substack{ \\ 19}}{(} \underset{\substack{ \\ 20}}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

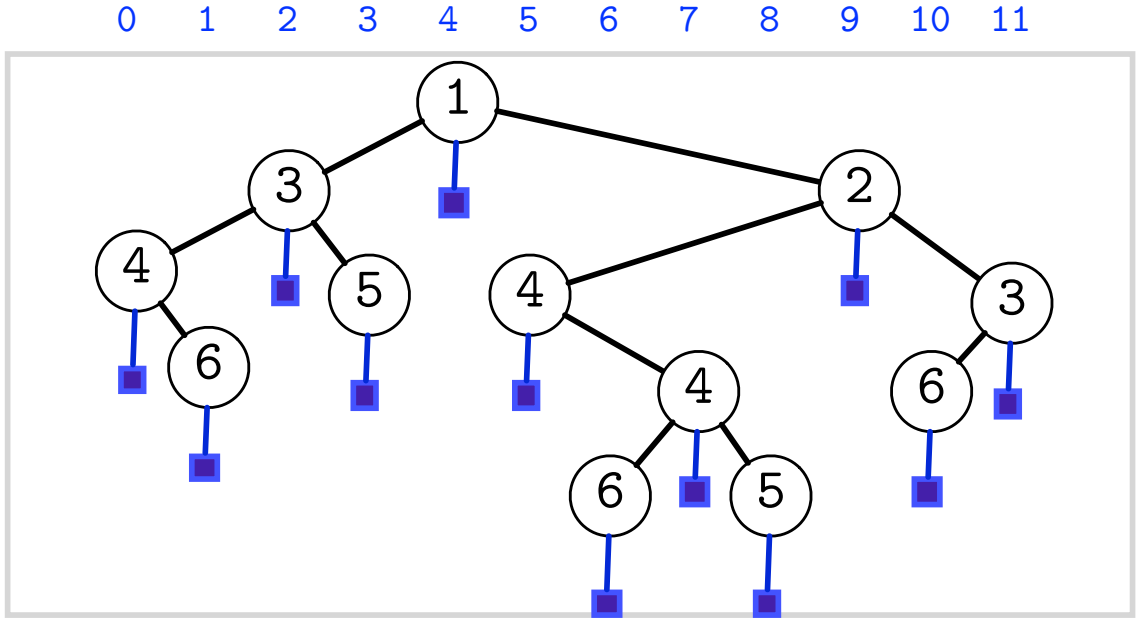


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \overset{1}{(}\overset{3}{(}\overset{4}{(}\underset{0}{()}\overset{6}{(}\underset{1}{()}\overset{6}{)}\overset{4}{)}\underset{2}{()}\overset{5}{(}\underset{3}{()}\overset{5}{)}\overset{3}{)}\underset{4}{()}\overset{2}{(}\overset{4}{(}\underset{5}{()}\overset{4}{(}\overset{6}{(}\underset{6}{()}\overset{6}{)}\underset{7}{()}\overset{5}{)}\overset{4}{(}\underset{8}{()}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



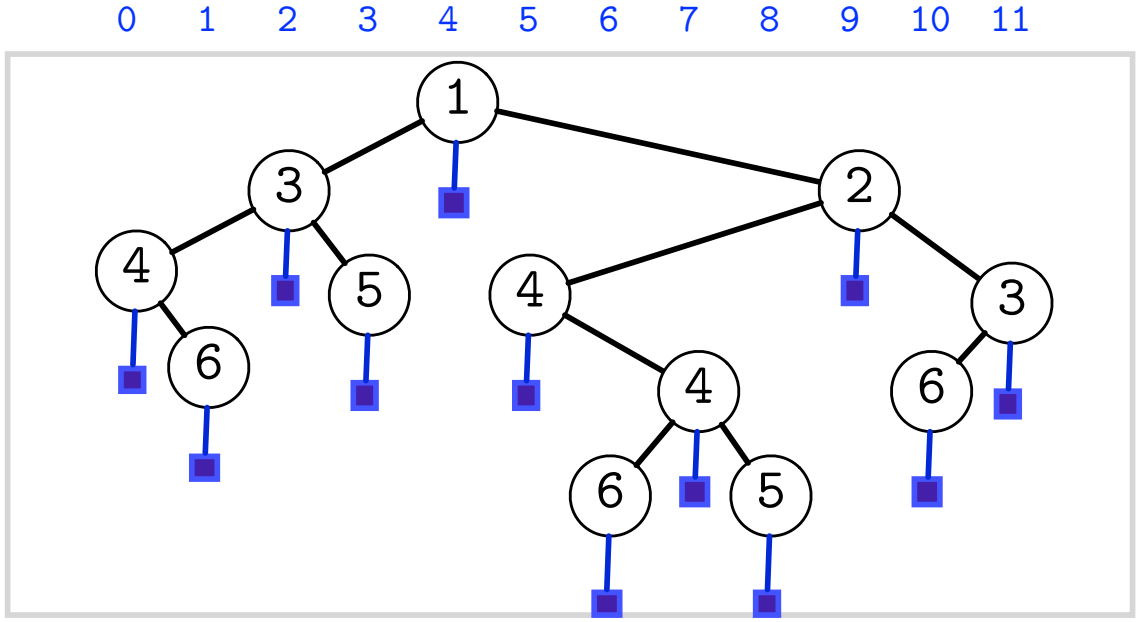
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{1\ 3\ 4}{(}\underset{6}{(}\underset{6\ 4}{(}\underset{5}{(}\underset{5\ 3}{)}\underset{2\ 4}{)}\underset{4\ 6}{(}\underset{6}{(}\underset{5}{(}\underset{5}{)}$$

$$\underset{0}{(}\underset{1}{(}\underset{2}{)}\underset{3}{(}\underset{4}{)}\underset{5}{(}\underset{6}{(}\underset{7}{)}\underset{8}{(}\underset{}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



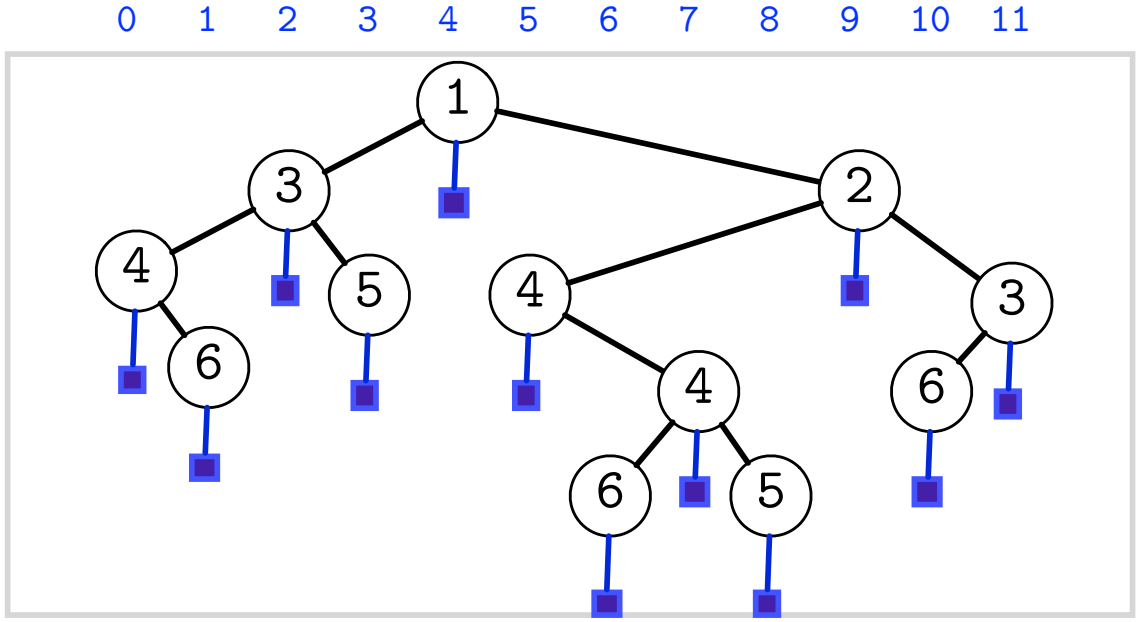
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{1\ 3\ 4}{(}\underset{6}{(}\underset{6\ 4}{(}\underset{5}{(}\underset{5\ 3}{)}\underset{2\ 4}{)}\underset{4\ 6}{(}\underset{6}{(}\underset{5}{(}\underset{5\ 4}{)}\underset{}{)}\underset{}{)}\underset{}{)}\underset{}{)}\underset{}{)}\underset{}{)}\underset{}{)}\underset{}{)}$$

$$\underset{0}{(}\underset{1}{(}\underset{2}{)}\underset{3}{)}\underset{4}{(}\underset{5}{(}\underset{6}{)}\underset{7}{)}\underset{8}{(}\underset{}{)}\underset{}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



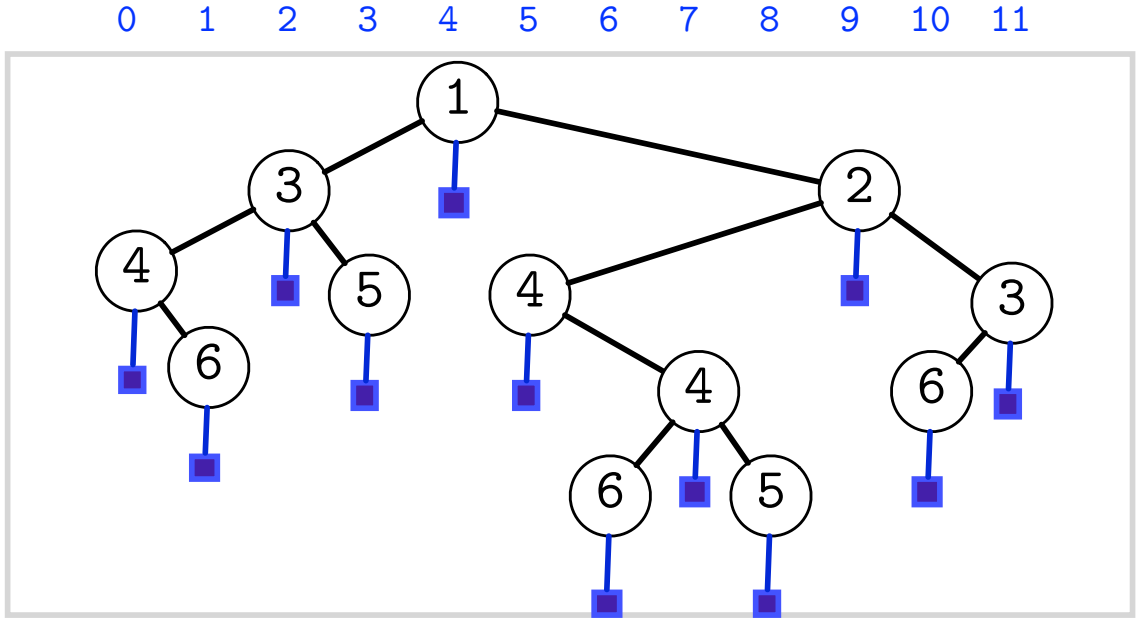
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{1\ 3\ 4}{(}\underset{6}{(}\underset{6\ 4}{(}\underset{5}{(}\underset{5\ 3}{)}\underset{2\ 4}{)}\underset{4\ 6}{(}\underset{6}{(}\underset{5}{)}\underset{5\ 4\ 4}{)}))$$

$$\underset{0}{(}\underset{1}{(}\underset{2}{)}\underset{3}{(}\underset{4}{)}\underset{5}{(}\underset{6}{(}\underset{7}{)}\underset{8}{(}))$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

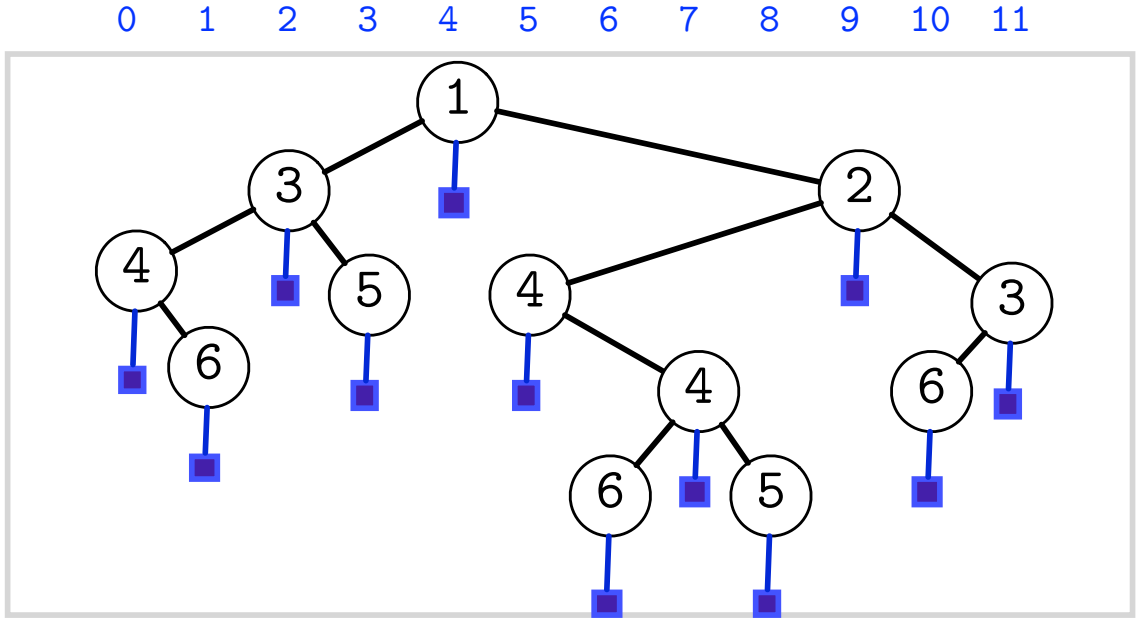


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = (((()(()))()(()))()((()((()()(()))(
\begin{array}{cccccccccccccccccccccccc}
1 & 3 & 4 & & 6 & & 6 & 4 & & 5 & & 5 & 3 & & 2 & 4 & & 4 & 6 & & 6 & & 5 & & 5 & 4 & 4 \\
 & & & 0 & & 1 & & & 2 & & 3 & & & 4 & & & 5 & & & 6 & & 7 & & 8 & & &
\end{array}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



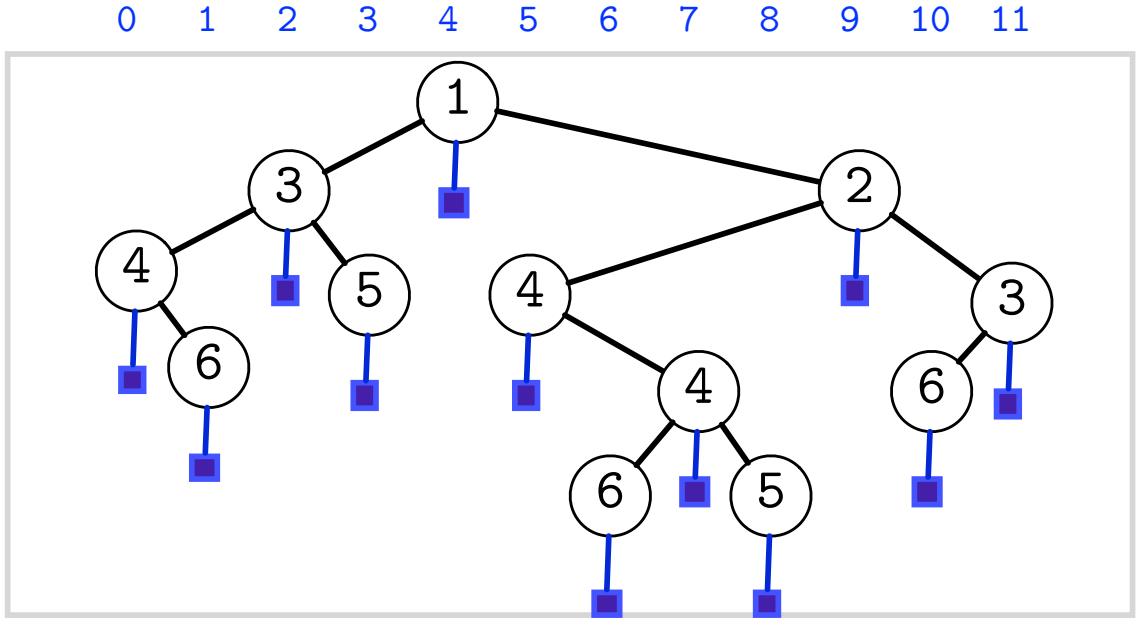
(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{1\ 3\ 4}{(}\underset{6}{(}\underset{6\ 4}{(}\underset{5}{(}\underset{5\ 3}{)}\underset{2\ 4}{)}\underset{4\ 6}{(}\underset{6}{(}\underset{5}{)}\underset{5\ 4\ 4}{)}\underset{}{)}$$

$$\underset{0}{(}\underset{1}{)}\underset{2}{(}\underset{3}{)}\underset{4}{(}\underset{5}{(}\underset{6}{)}\underset{7}{(}\underset{8}{)}\underset{9}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

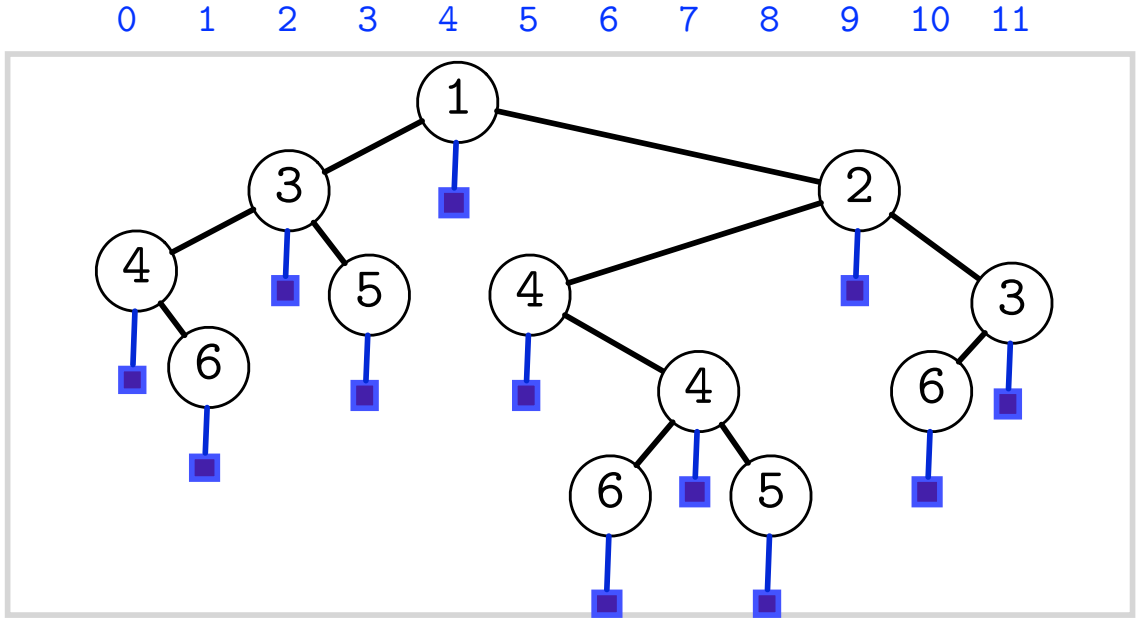


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1 \\ 1}}{(}\underset{\substack{6 \\ 2}}{(}\underset{\substack{ \\ 3}}{)}\underset{\substack{6\ 4 \\ 4}}{(}\underset{\substack{ \\ 5}}{)}\underset{\substack{ \\ 6}}{)}\underset{\substack{5 \\ 7}}{)}\underset{\substack{5\ 3 \\ 8}}{(}\underset{\substack{ \\ 9}}{)}\underset{\substack{2\ 4 \\ 10}}{)}\underset{\substack{4\ 6 \\ 11}}{(}\underset{\substack{ \\ 12}}{(}\underset{\substack{6 \\ 13}}{)}\underset{\substack{ \\ 14}}{(}\underset{\substack{5 \\ 15}}{(}\underset{\substack{ \\ 16}}{)}\underset{\substack{5\ 4\ 4 \\ 17}}{)}\underset{\substack{3 \\ 18}}{)}\underset{\substack{ \\ 19}}{)}\underset{\substack{ \\ 20}}{(}\underset{\substack{ \\ 21}}{)}\underset{\substack{ \\ 22}}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

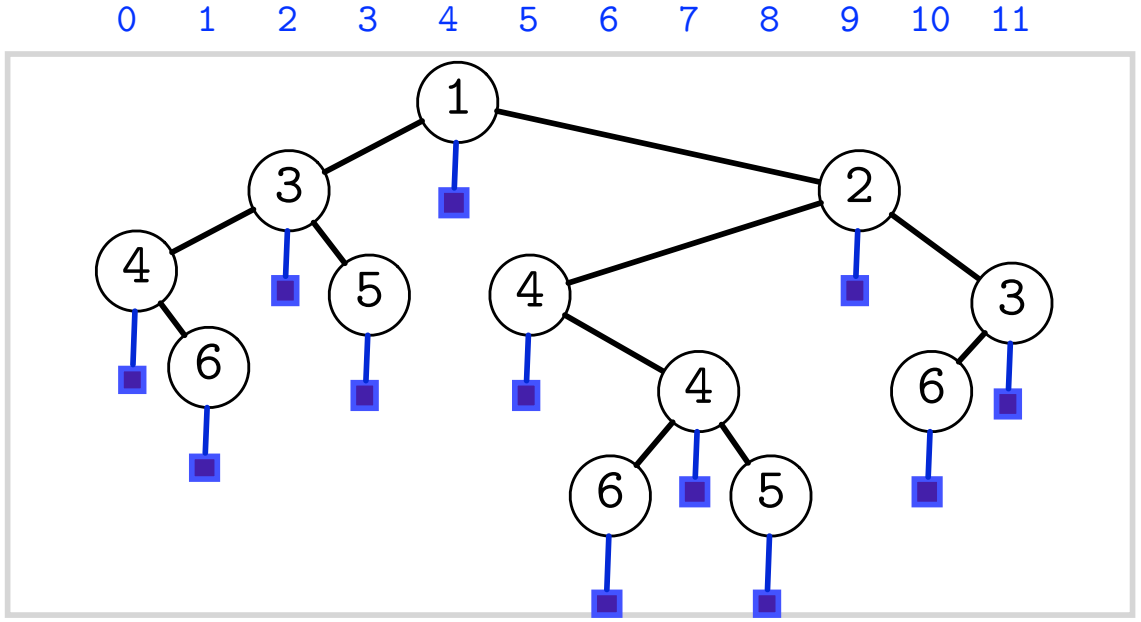


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1 \\ 1}}{(}\underset{\substack{6 \\ 2}}{(}\underset{\substack{6\ 4 \\ 3}}{(}\underset{\substack{5 \\ 4}}{)}\underset{\substack{5\ 3 \\ 5}}{)}\underset{\substack{2\ 4 \\ 6}}{)}\underset{\substack{4\ 6 \\ 7}}{(}\underset{\substack{6 \\ 8}}{(}\underset{\substack{5 \\ 9}}{)}\underset{\substack{5\ 4\ 4 \\ 10}}{)}\underset{\substack{3\ 6 \\ 11}}{)}\underset{\substack{ \\ 12}}{(}\underset{\substack{ \\ 13}}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

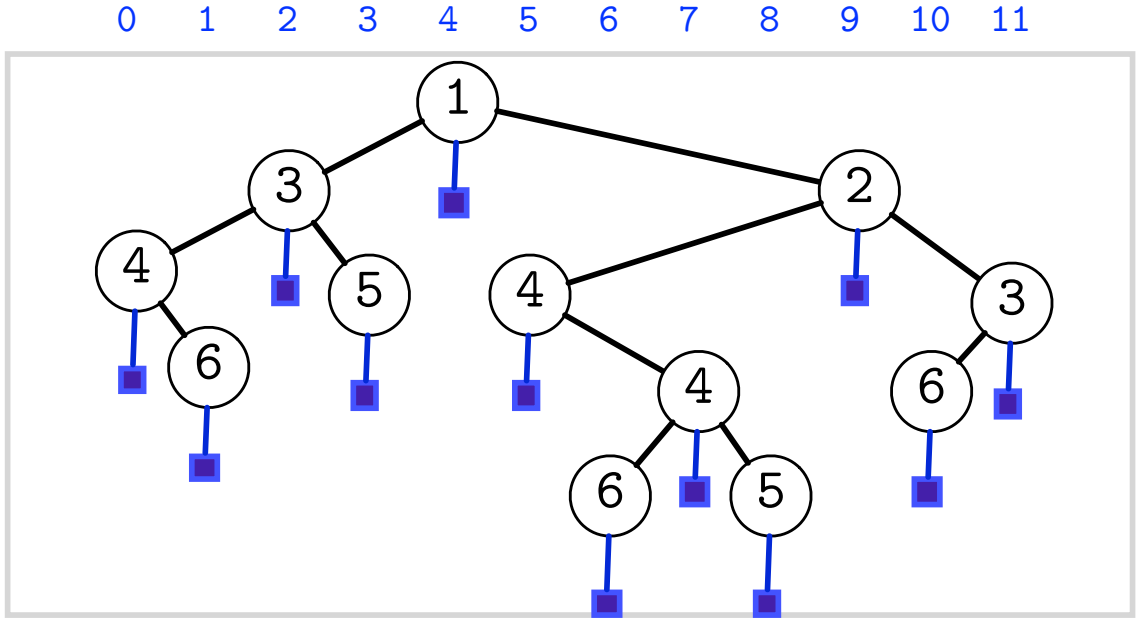


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{1\ 3\ 4}{(}\underset{6}{(}\underset{6\ 4}{(}\underset{5}{(}\underset{5\ 3}{)}\underset{2\ 4}{)}\underset{4\ 6}{(}\underset{6}{(}\underset{5}{)}\underset{5\ 4\ 4}{)}\underset{3\ 6}{)}(((\underset{0}{)}\underset{1}{(}\underset{2}{)}\underset{3}{(}\underset{4}{)}\underset{5}{)}\underset{6}{(}\underset{7}{(}\underset{8}{)}\underset{9}{(}))$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

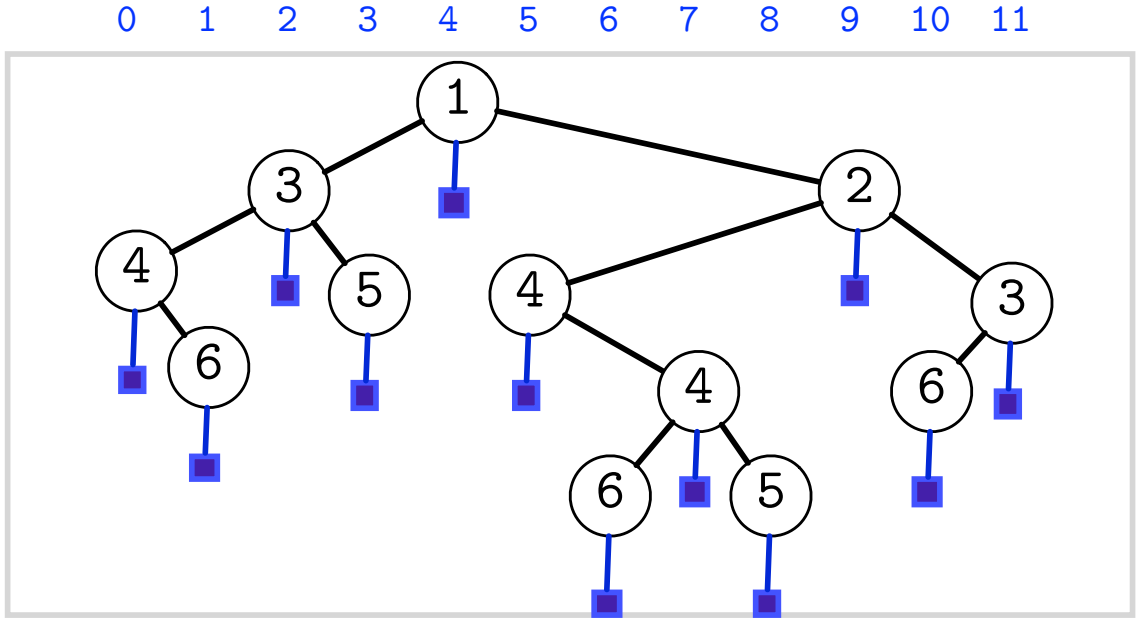


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1\ 3\ 4 \\ 1}}{(}\underset{\substack{1\ 3\ 4 \\ 2}}{(}\underset{\substack{6 \\ 3}}{(}\underset{\substack{6 \\ 4}}{)}\underset{\substack{1\ 3\ 4 \\ 5}}{)}\underset{\substack{6\ 4 \\ 6}}{(}\underset{\substack{6\ 4 \\ 7}}{)}\underset{\substack{1\ 3\ 4 \\ 8}}{)}\underset{\substack{5 \\ 9}}{(}\underset{\substack{5 \\ 10}}{)}\underset{\substack{5\ 3 \\ 11}}{)}\underset{\substack{1\ 3\ 4 \\ 12}}{(}\underset{\substack{2\ 4 \\ 13}}{(}\underset{\substack{2\ 4 \\ 14}}{)}\underset{\substack{4\ 6 \\ 15}}{(}\underset{\substack{4\ 6 \\ 16}}{(}\underset{\substack{6 \\ 17}}{)}\underset{\substack{4\ 6 \\ 18}}{(}\underset{\substack{6 \\ 19}}{)}\underset{\substack{5 \\ 20}}{)}\underset{\substack{5 \\ 21}}{(}\underset{\substack{5 \\ 22}}{)}\underset{\substack{5\ 4\ 4 \\ 23}}{)}\underset{\substack{5\ 4\ 4 \\ 24}}{)}\underset{\substack{3\ 6 \\ 25}}{(}\underset{\substack{3\ 6 \\ 26}}{(}\underset{\substack{3\ 6 \\ 27}}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

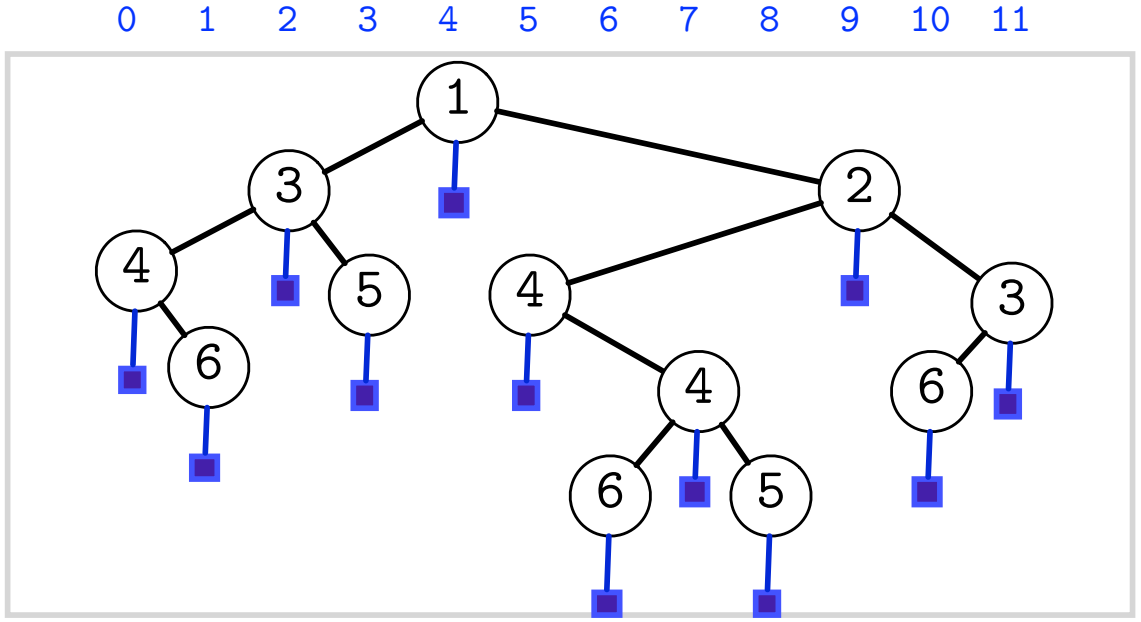


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1\ 3\ 4 \\ 1}}{(}\underset{\substack{1\ 3\ 4 \\ 2}}{(}\underset{\substack{6 \\ 3}}{(}\underset{\substack{6 \\ 4}}{)}\underset{\substack{6\ 4 \\ 5}}{(}\underset{\substack{6\ 4 \\ 6}}{)}\underset{\substack{1\ 3\ 4 \\ 7}}{)}\underset{\substack{5 \\ 8}}{(}\underset{\substack{5 \\ 9}}{)}\underset{\substack{5\ 3 \\ 10}}{(}\underset{\substack{5\ 3 \\ 11}}{(}\underset{\substack{2\ 4 \\ 12}}{)}\underset{\substack{2\ 4 \\ 13}}{(}\underset{\substack{4\ 6 \\ 14}}{(}\underset{\substack{6 \\ 15}}{(}\underset{\substack{6 \\ 16}}{)}\underset{\substack{4\ 6 \\ 17}}{(}\underset{\substack{6 \\ 18}}{)}\underset{\substack{5 \\ 19}}{(}\underset{\substack{5 \\ 20}}{)}\underset{\substack{5\ 4\ 4 \\ 21}}{)}\underset{\substack{5\ 4\ 4 \\ 22}}{)}\underset{\substack{3\ 6 \\ 23}}{)}\underset{\substack{3\ 6 \\ 24}}{(}\underset{\substack{6 \\ 25}}{(}\underset{\substack{6 \\ 26}}{)}\underset{\substack{6 \\ 27}}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

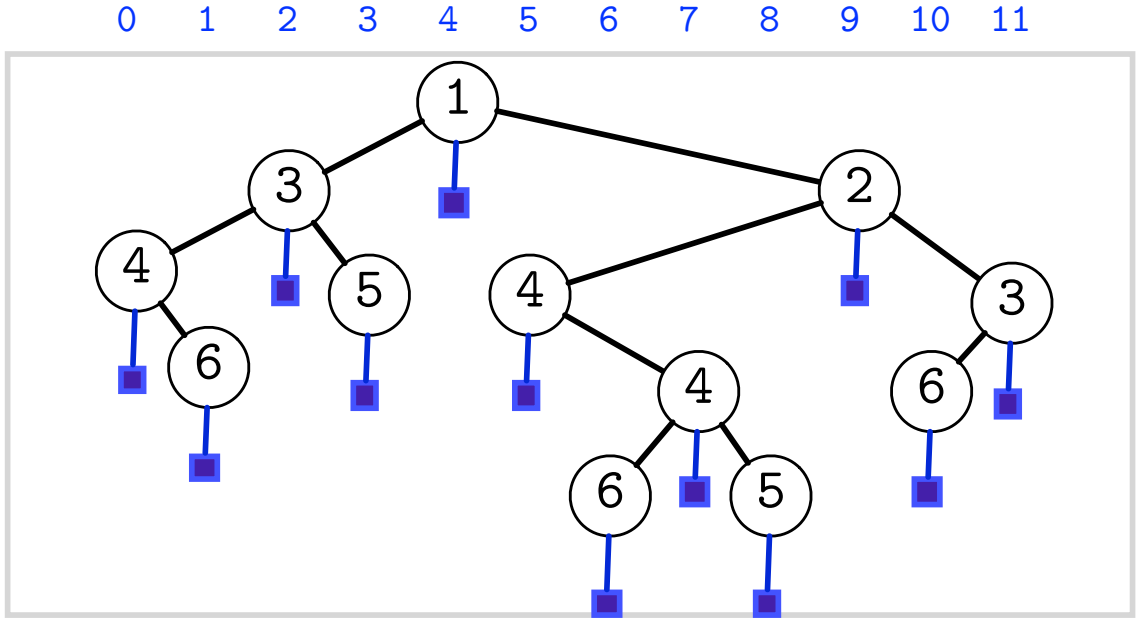


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1 \\ 1}}{(}\underset{\substack{6 \\ 2}}{(}\underset{\substack{ \\ 3}}{)}\underset{\substack{6\ 4 \\ 4}}{(}\underset{\substack{ \\ 5}}{)}\underset{\substack{ \\ 6}}{)}\underset{\substack{5 \\ 7}}{)}\underset{\substack{ \\ 8}}{(}\underset{\substack{5\ 3 \\ 9}}{(}\underset{\substack{ \\ 10}}{)}\underset{\substack{ \\ 11}}{)}\underset{\substack{2\ 4 \\ 12}}{(}\underset{\substack{ \\ 13}}{(}\underset{\substack{4\ 6 \\ 14}}{)}\underset{\substack{6 \\ 15}}{(}\underset{\substack{ \\ 16}}{(}\underset{\substack{ \\ 17}}{)}\underset{\substack{5 \\ 18}}{(}\underset{\substack{ \\ 19}}{)}\underset{\substack{5\ 4\ 4 \\ 20}}{(}\underset{\substack{ \\ 21}}{)}\underset{\substack{ \\ 22}}{)}\underset{\substack{3\ 6 \\ 23}}{)}\underset{\substack{ \\ 24}}{(}\underset{\substack{6 \\ 25}}{(}\underset{\substack{ \\ 26}}{)}\underset{\substack{ \\ 27}}{)}\underset{\substack{ \\ 28}}{(}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

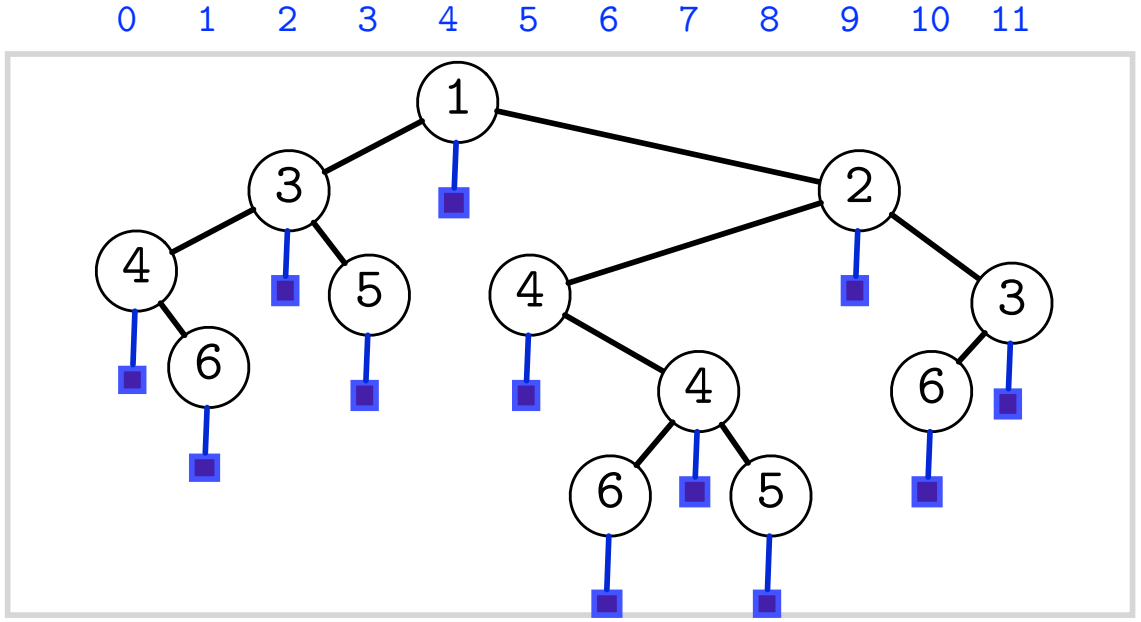


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(} \underset{\substack{1\ 3\ 4 \\ 1}}{(} \underset{\substack{1\ 3\ 4 \\ 2}}{(} \underset{\substack{6 \\ 3}}{(} \underset{\substack{6 \\ 4}}{)} \underset{\substack{6\ 4 \\ 5}}{(} \underset{\substack{6 \\ 6}}{)} \underset{\substack{6\ 4 \\ 7}}{)} \underset{\substack{5 \\ 8}}{)} \underset{\substack{5\ 3 \\ 9}}{(} \underset{\substack{5 \\ 10}}{)} \underset{\substack{5\ 3 \\ 11}}{)} \underset{\substack{2\ 4 \\ 12}}{(} \underset{\substack{2\ 4 \\ 13}}{(} \underset{\substack{4\ 6 \\ 14}}{(} \underset{\substack{6 \\ 15}}{(} \underset{\substack{6 \\ 16}}{)} \underset{\substack{4\ 6 \\ 17}}{(} \underset{\substack{6 \\ 18}}{)} \underset{\substack{4\ 6 \\ 19}}{)} \underset{\substack{5 \\ 20}}{)} \underset{\substack{5\ 4 \\ 21}}{(} \underset{\substack{5 \\ 22}}{)} \underset{\substack{5\ 4 \\ 23}}{)} \underset{\substack{4 \\ 24}}{)} \underset{\substack{4 \\ 25}}{)} \underset{\substack{3\ 6 \\ 26}}{(} \underset{\substack{3\ 6 \\ 27}}{(} \underset{\substack{6 \\ 28}}{(} \underset{\substack{6 \\ 29}}{)} \underset{\substack{6 \\ 30}}{)} \underset{\substack{6 \\ 31}}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

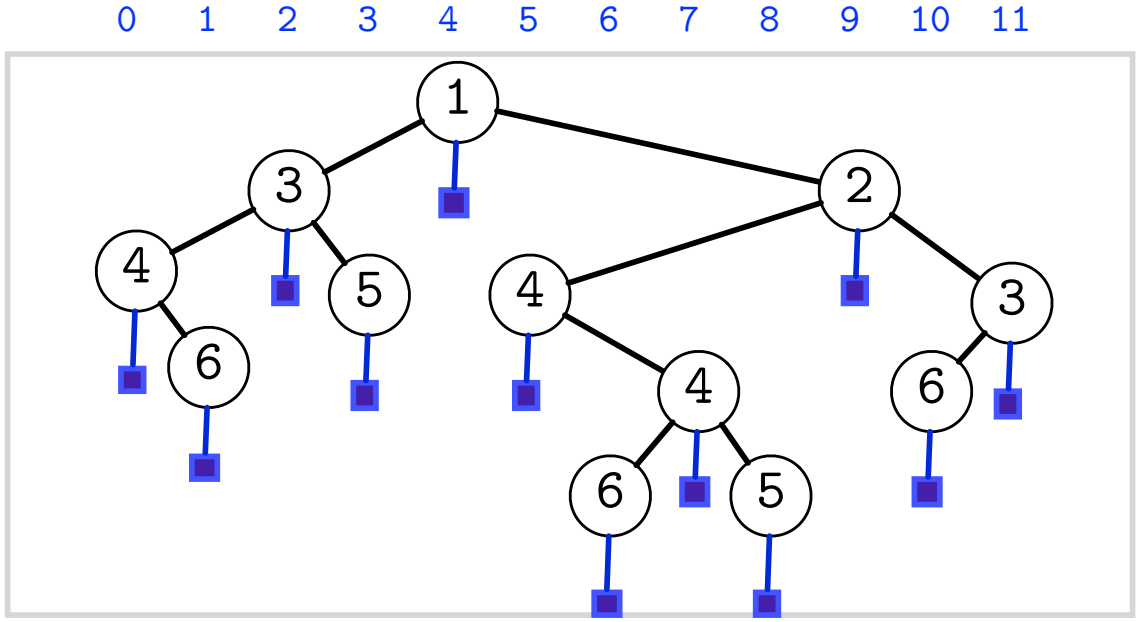


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1\ 3\ 4 \\ 1}}{(}\underset{\substack{1\ 3\ 4 \\ 2}}{(}\underset{\substack{6 \\ 3}}{(}\underset{\substack{6 \\ 4}}{)}\underset{\substack{1\ 3\ 4 \\ 5}}{)}\underset{\substack{6\ 4 \\ 6}}{(}\underset{\substack{6\ 4 \\ 7}}{)}\underset{\substack{1\ 3\ 4 \\ 8}}{)}\underset{\substack{5 \\ 9}}{(}\underset{\substack{5 \\ 10}}{)}\underset{\substack{5\ 3 \\ 11}}{)}\underset{\substack{2\ 4 \\ 12}}{(}\underset{\substack{2\ 4 \\ 13}}{(}\underset{\substack{4\ 6 \\ 14}}{(}\underset{\substack{4\ 6 \\ 15}}{)}\underset{\substack{6 \\ 16}}{(}\underset{\substack{6 \\ 17}}{(}\underset{\substack{6 \\ 18}}{)}\underset{\substack{5 \\ 19}}{(}\underset{\substack{5 \\ 20}}{)}\underset{\substack{5\ 4\ 4 \\ 21}}{)}\underset{\substack{5\ 4\ 4 \\ 22}}{)}\underset{\substack{1\ 3\ 4 \\ 23}}{)}\underset{\substack{3\ 6 \\ 24}}{(}\underset{\substack{3\ 6 \\ 25}}{(}\underset{\substack{6 \\ 26}}{(}\underset{\substack{6 \\ 27}}{)}\underset{\substack{3 \\ 28}}{)}\underset{\substack{3 \\ 29}}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node

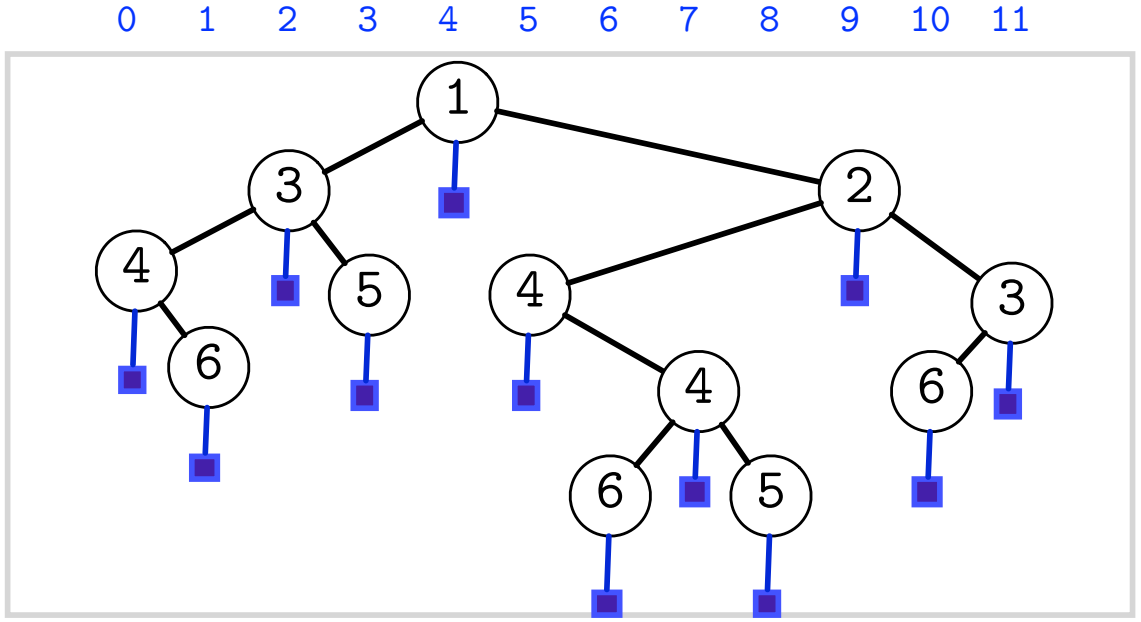


(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = \underset{\substack{1\ 3\ 4 \\ 0}}{(}\underset{\substack{1\ 3\ 4 \\ 1}}{(}\underset{\substack{1\ 3\ 4 \\ 2}}{(}\underset{\substack{6 \\ 3}}{(}\underset{\substack{6 \\ 4}}{)}\underset{\substack{6\ 4 \\ 5}}{(}\underset{\substack{6\ 4 \\ 6}}{)}\underset{\substack{6\ 4 \\ 7}}{)}\underset{\substack{5 \\ 8}}{)}\underset{\substack{5\ 3 \\ 9}}{(}\underset{\substack{5\ 3 \\ 10}}{)}\underset{\substack{5\ 3 \\ 11}}{)}\underset{\substack{2\ 4 \\ 12}}{)}\underset{\substack{2\ 4 \\ 13}}{(}\underset{\substack{4\ 6 \\ 14}}{(}\underset{\substack{4\ 6 \\ 15}}{)}\underset{\substack{6 \\ 16}}{(}\underset{\substack{6 \\ 17}}{(}\underset{\substack{6 \\ 18}}{)}\underset{\substack{5 \\ 19}}{(}\underset{\substack{5 \\ 20}}{)}\underset{\substack{5 \\ 21}}{)}\underset{\substack{5\ 4\ 4 \\ 22}}{)}\underset{\substack{5\ 4\ 4 \\ 23}}{)}\underset{\substack{3\ 6 \\ 24}}{(}\underset{\substack{3\ 6 \\ 25}}{(}\underset{\substack{6 \\ 26}}{(}\underset{\substack{6 \\ 27}}{)}\underset{\substack{3\ 6 \\ 28}}{)}\underset{\substack{3\ 2 \\ 29}}{(}\underset{\substack{3\ 2 \\ 30}}{)}\underset{\substack{3\ 2 \\ 31}}{)}$$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

(2) Add a leaf to each node



(3) DFS traversal to construct balanced parentheses sequence

$$BP_{ext} = (((()(()))()(()))()((()((()()(()))))()(((()())))))$$

1 3 4 6 6 4 5 5 3 2 4 4 6 6 5 5 4 4 3 6 6 3 2 1

0 1 2 3 4 5 6 7 8 9 10 11

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

- The extended Cartesian Tree contains $2n$ nodes
- The balanced parentheses sequence consists of $4n$ bits
- The position of each leaf corresponds to the inorder number of its parent in the Cartesian tree
- The inorder number corresponds to the array index of the element
- Let $excess(i) = rank(i+1, 1, BP_{ext}) - rank(i+1, 0, BP_{ext})$

For $0 \leq i \leq j < n$ we get:

00 **rmq**$_A(i,j)$
01 $ipos \leftarrow select(i+1, (), BP_{ext})$
02 $jpos \leftarrow select(j+1, (), BP_{ext})$
03 **return** $rank(rmq^{\pm 1}_{excess}(ipos, jpos+1), (), BP_{ext})$

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

- Added leaves are used to navigate to inorder index nodes
- Select on a pattern „`()`" of fixed size can be done in constant time after precomputing a $o(n)$-space structure
- Let v be the $(i+1)$th leaf node
- Let w be the $(j+1)$th leaf node
- $rmq^{\pm 1}_{excess}(ipos, jpos + 1)$ is the position of closing parenthesis of the leaf node z added to $LCA(v, w)$
- In-order number of $LCA(v, w)$ corresponds to index of minimum in $A[i, j]$ and can be determined by a rank operation on pattern „`()`"

Next: $o(n)$ data structure to support $rmq^{\pm 1}_{excess}$ queries on BP_{ext}

$\langle O(n), O(1) \rangle$ solution ($4n + o(n)$ bits)

- Divide the (conceptional) array *excess* in blocks of size $\log^3 n$
- $S[0, n/\log^3 n]$ stores the minima of the blocks
- Solution #2 for S requires $O(\frac{n}{\log^3 n} \cdot \log^2 n) = o(n)$ bits
- Divide each block in subblocks of size $\frac{1}{2} \log n$
- Apply again solution #2 on subblocks ($n' = \log^2 n$). I.e. total space $O(\frac{n}{\log n} \log n' \cdot \log n') = O(\frac{n}{\log n} \log^2 \log n) = o(n)$
- Lookup table for blocks of size $\frac{1}{2} \log n$ is also in $o(n)$

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

Observation

Adding the leaf nodes enables inorder indexing of the nodes but doubles the space.

Next: Approach which does not require additional nodes.

- Cartesian Tree (CT) is a binary tree of n nodes
- Transform CT into general tree of $n + 1$ nodes

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

Observation

Adding the leaf nodes enables inorder indexing of the nodes but doubles the space.

Next: Approach which does not require additional nodes.

- Cartesian Tree (CT) is a binary tree of n nodes
- Transform CT into general tree of $n + 1$ nodes

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

Observation

Adding the leaf nodes enables inorder indexing of the nodes but doubles the space.

Next: Approach which does not require additional nodes.

- Cartesian Tree (CT) is a binary tree of n nodes
- Transform CT into general tree of $n + 1$ nodes

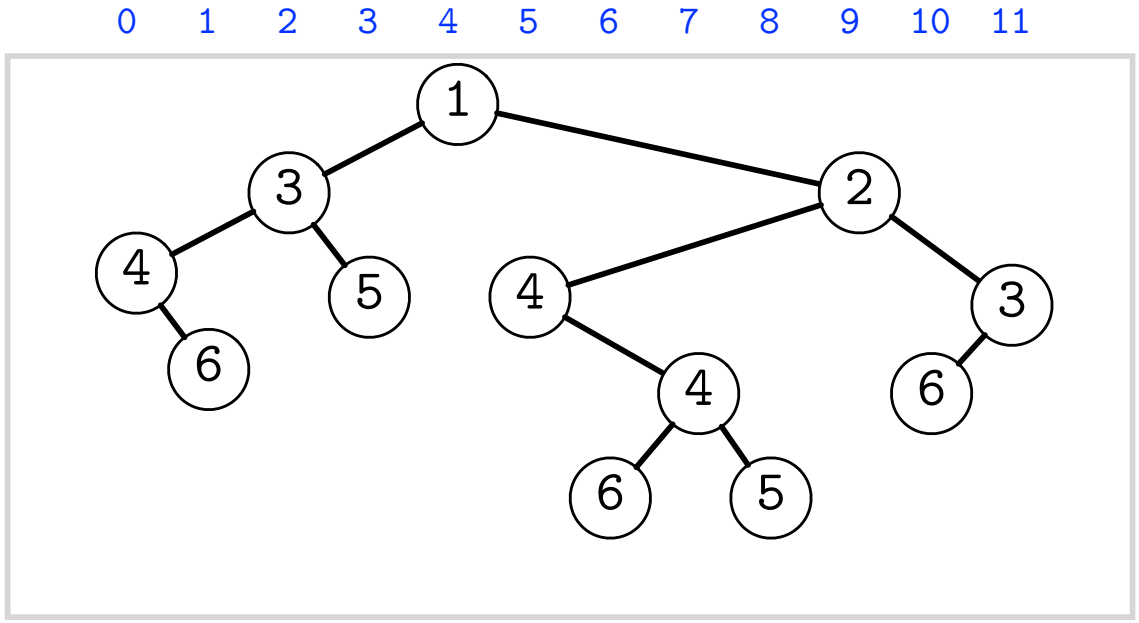
$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Observation

Adding the leaf nodes enables inorder indexing of the nodes but doubles the space.

Next: Approach which does not require additional nodes.

- Cartesian Tree (CT) is a binary tree of n nodes
- Transform CT into general tree of $n + 1$ nodes



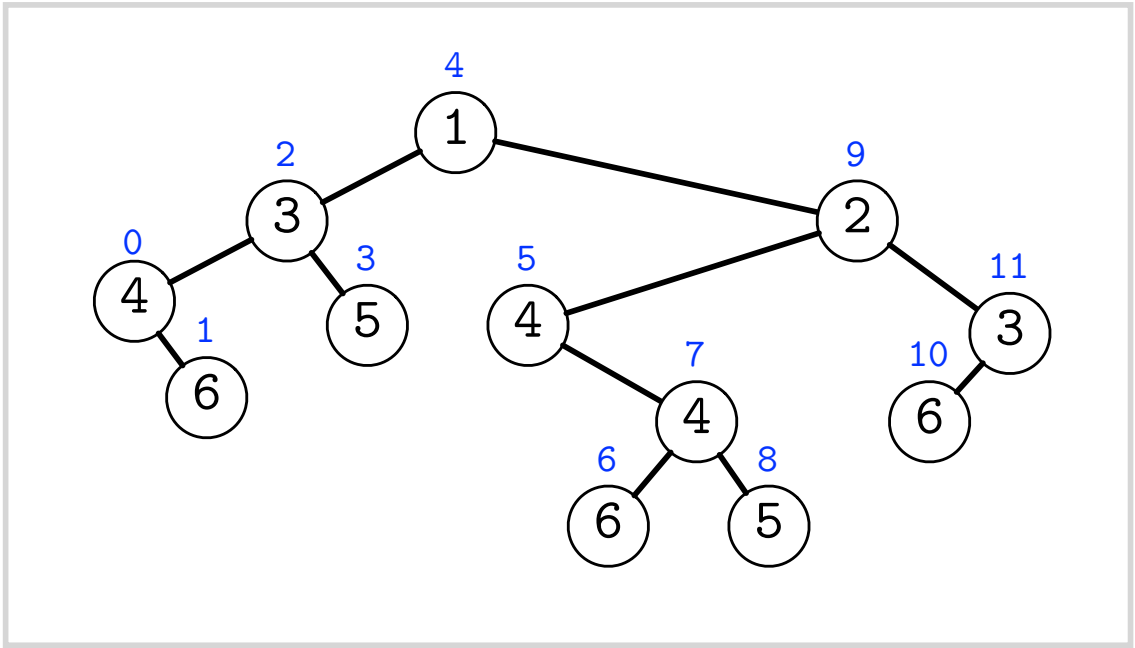
$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Observation

Adding the leaf nodes enables inorder indexing of the nodes but doubles the space.

Next: Approach which does not require additional nodes.

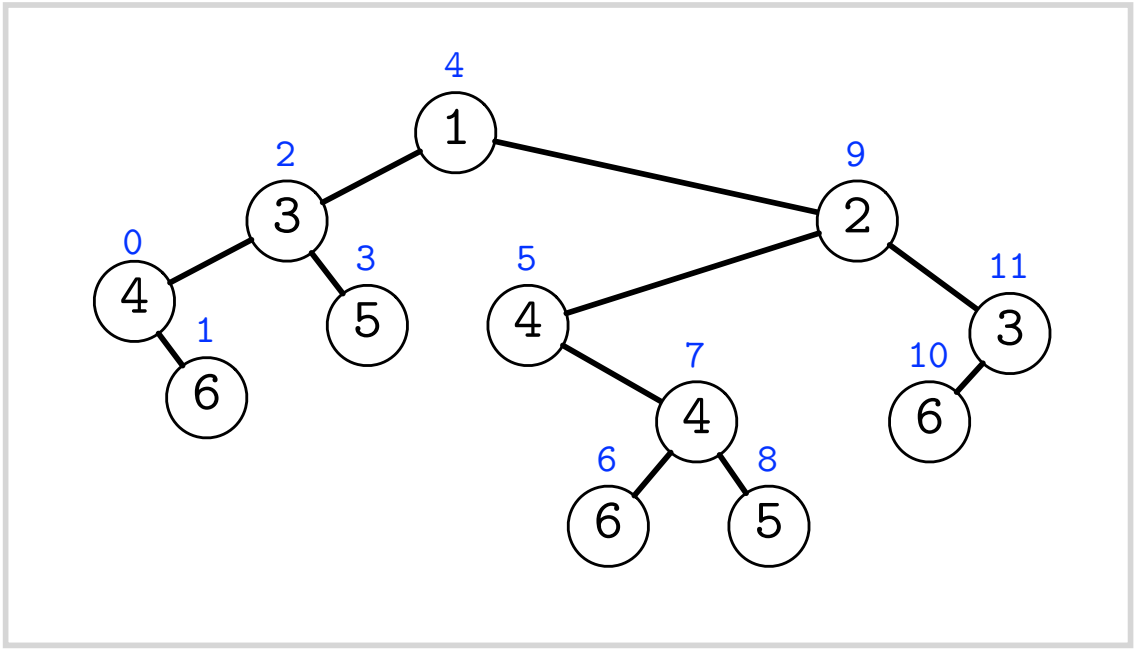
- Cartesian Tree (CT) is a binary tree of n nodes
- Transform CT into general tree of $n + 1$ nodes



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

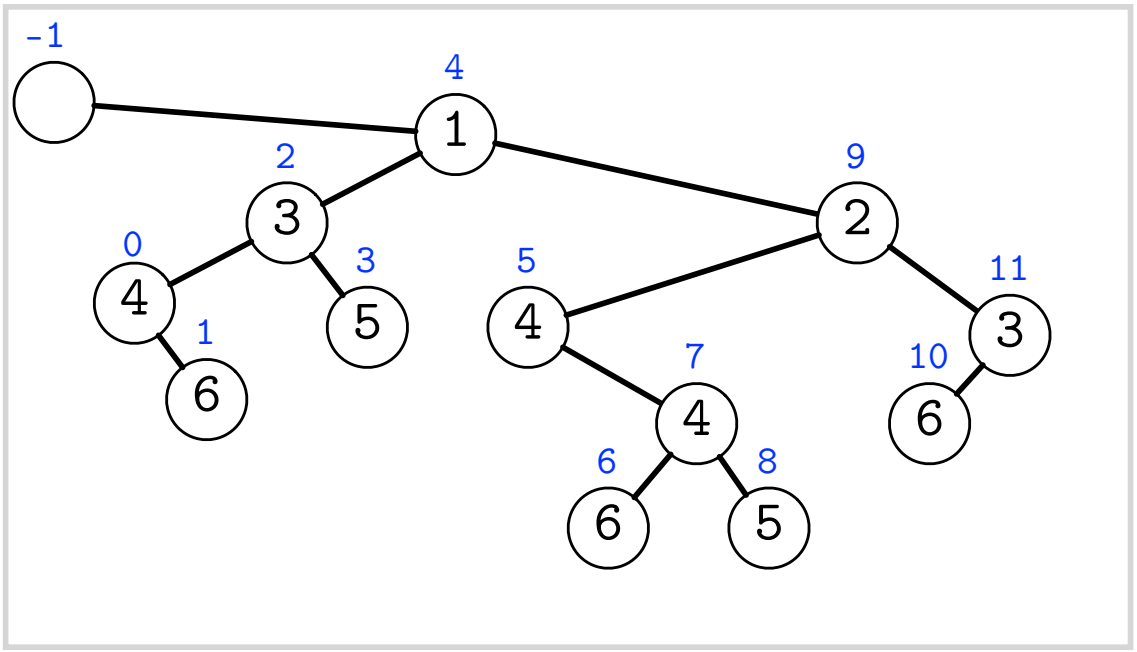
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

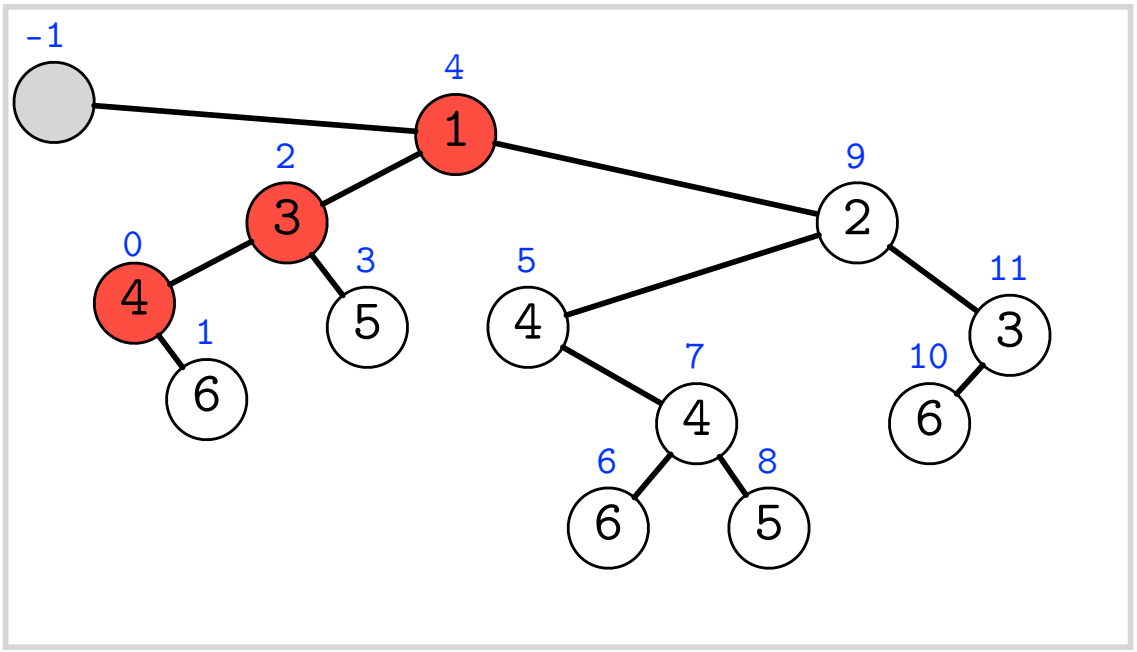
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

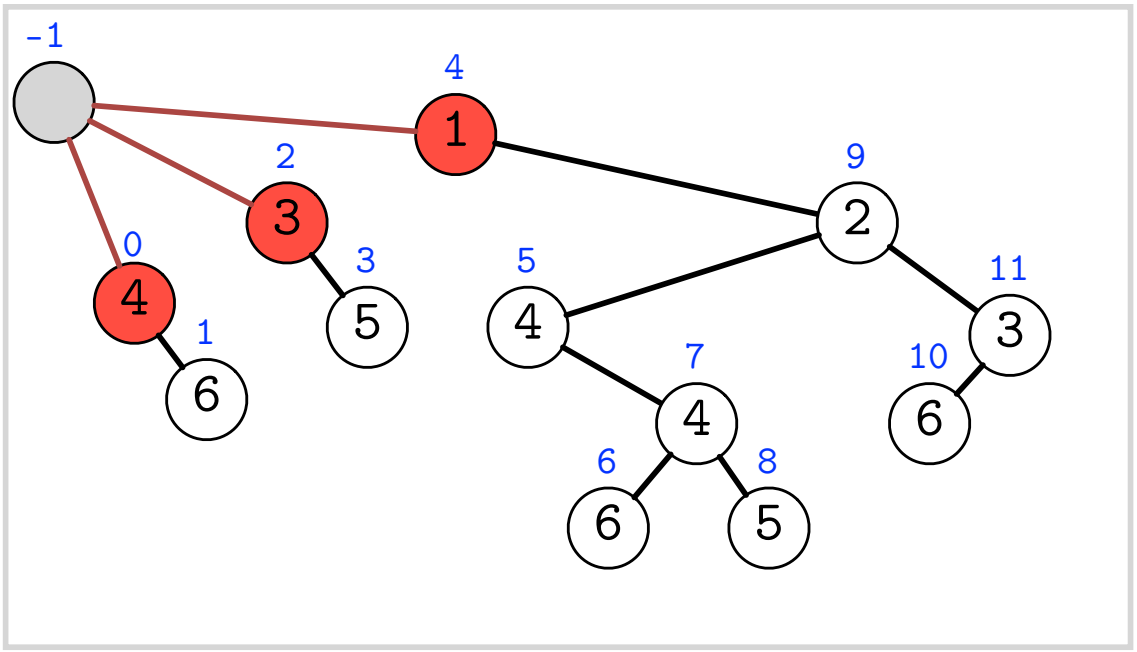
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

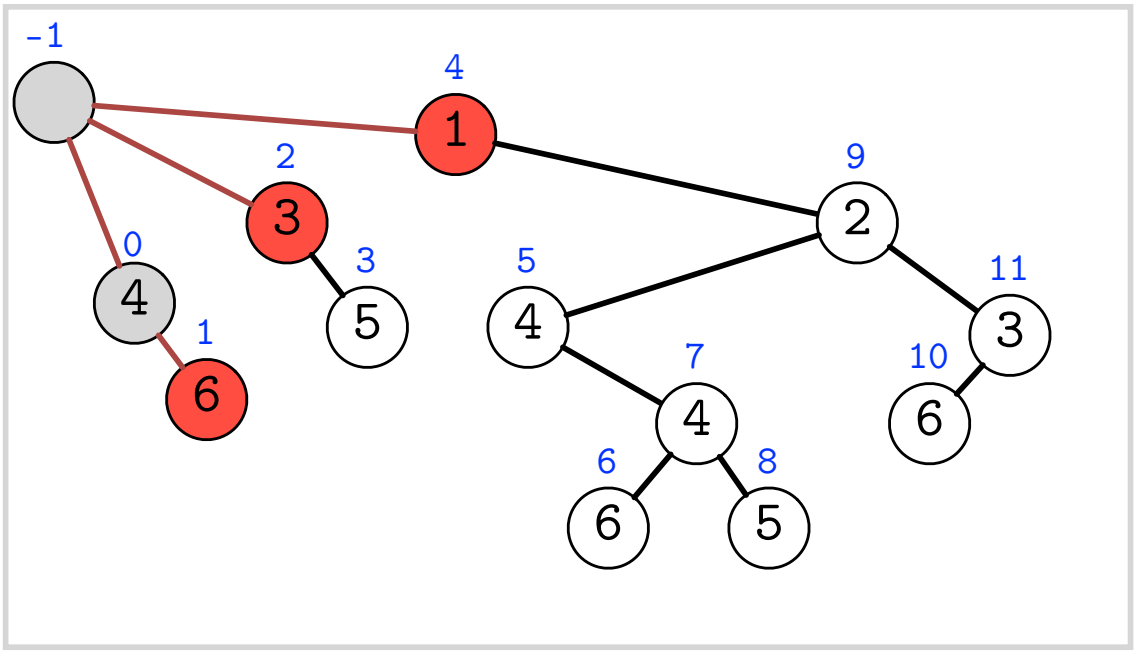
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

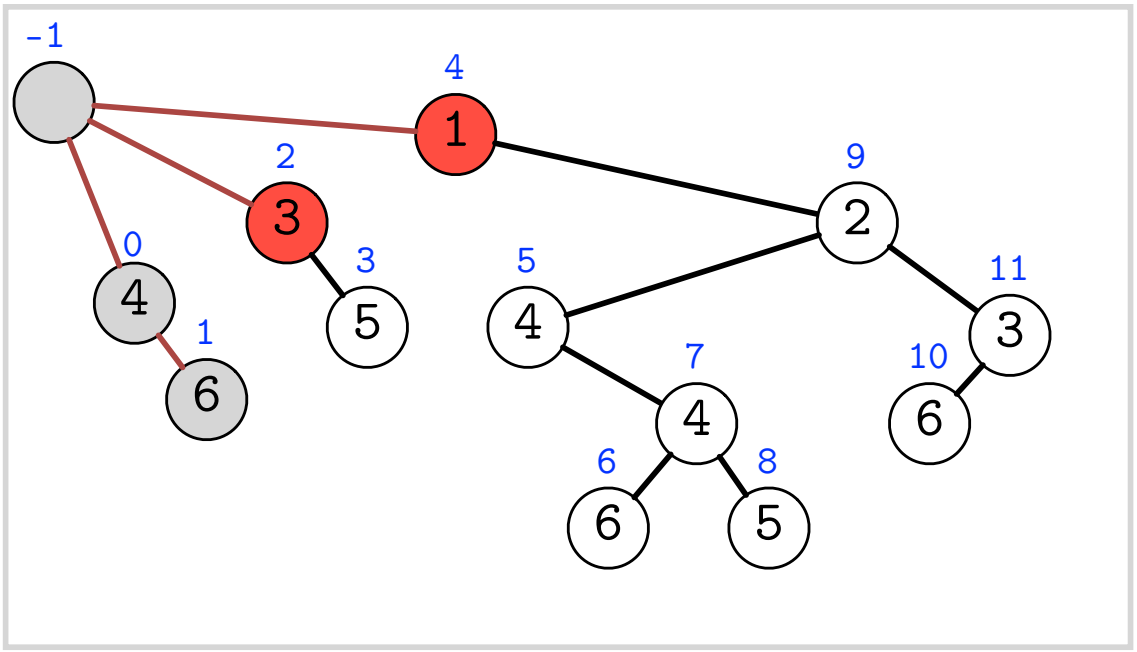
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

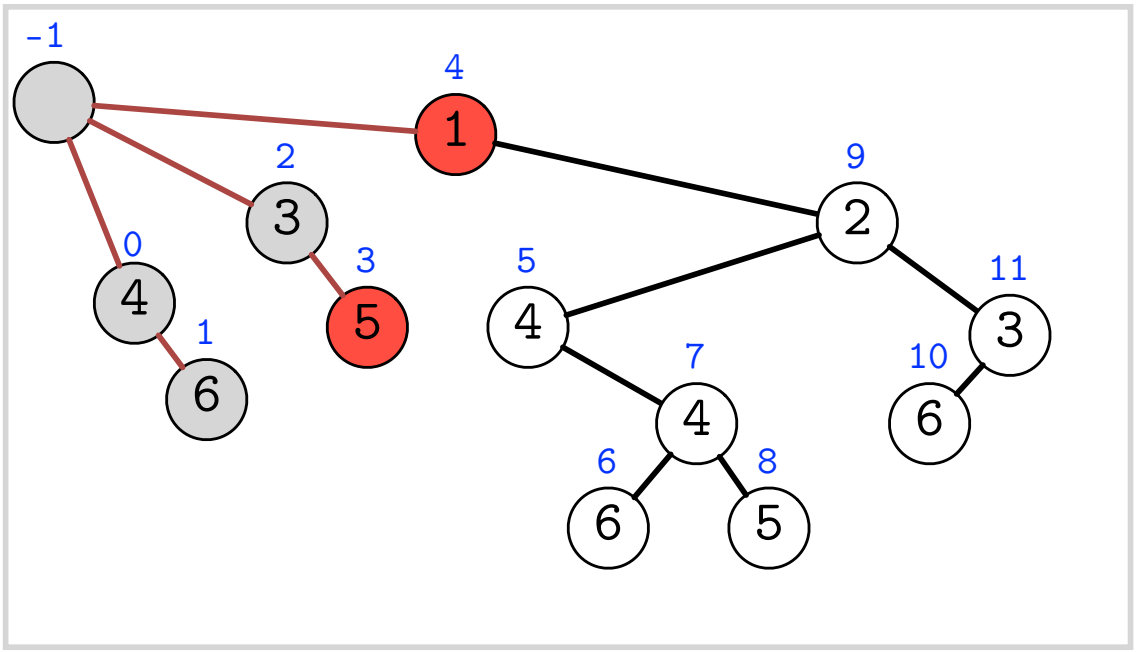
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

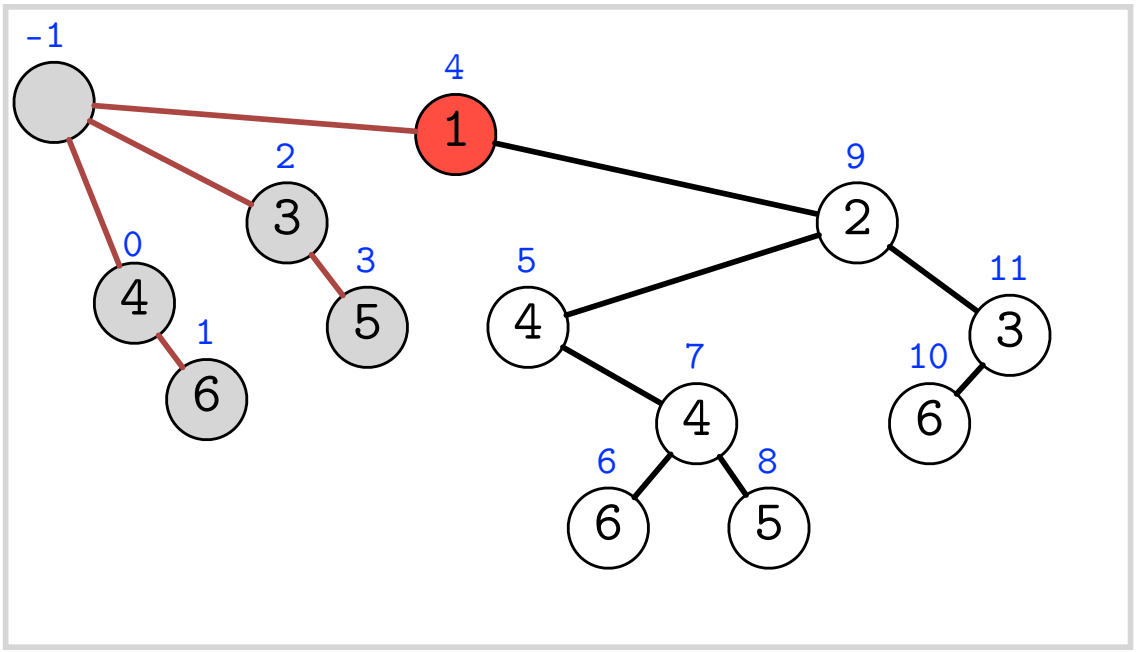
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

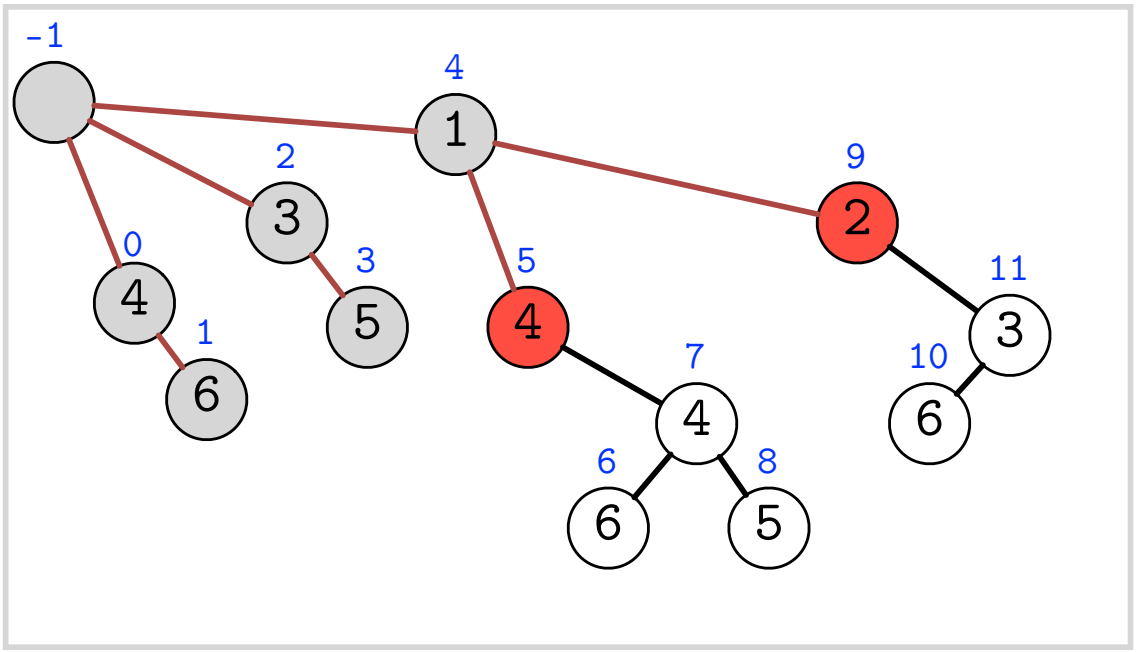
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

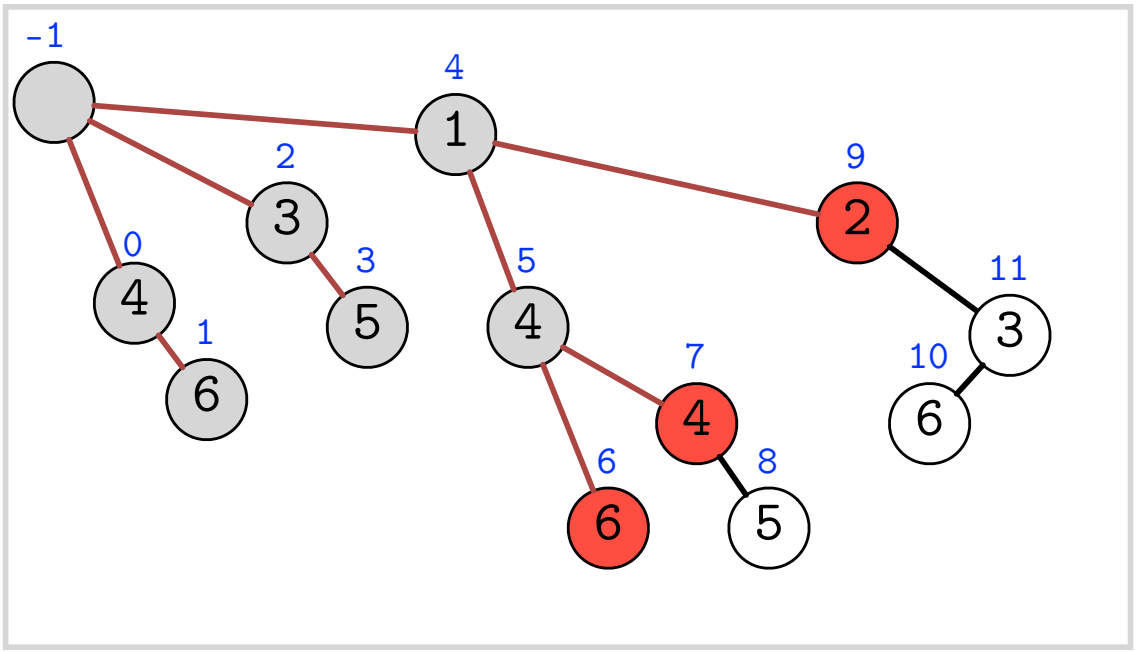
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

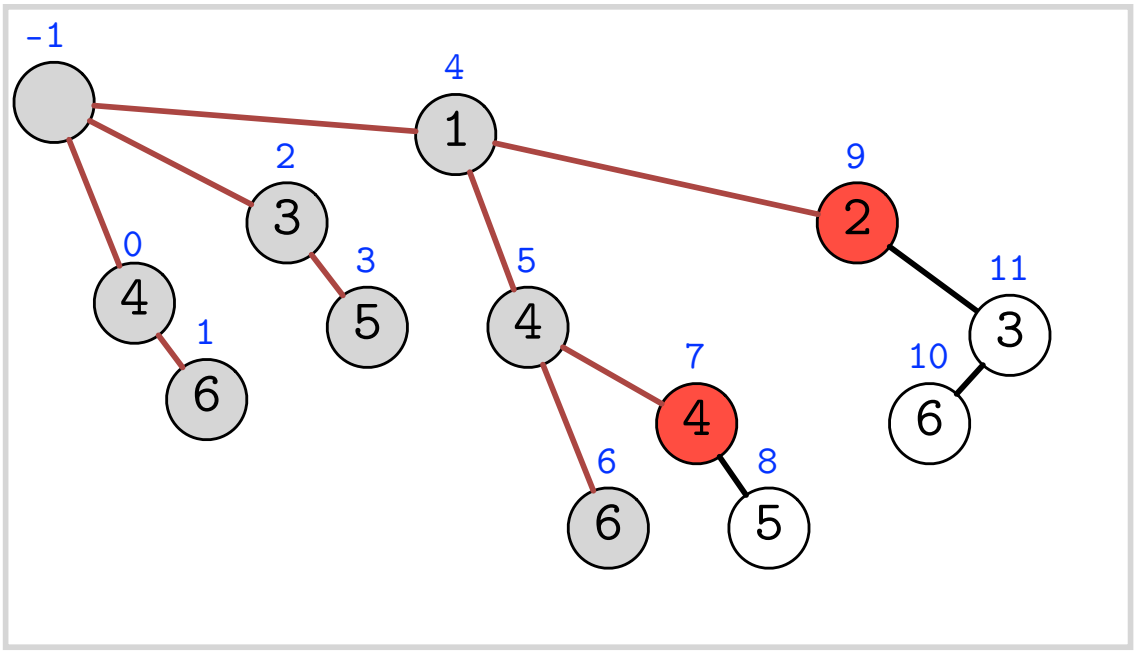
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

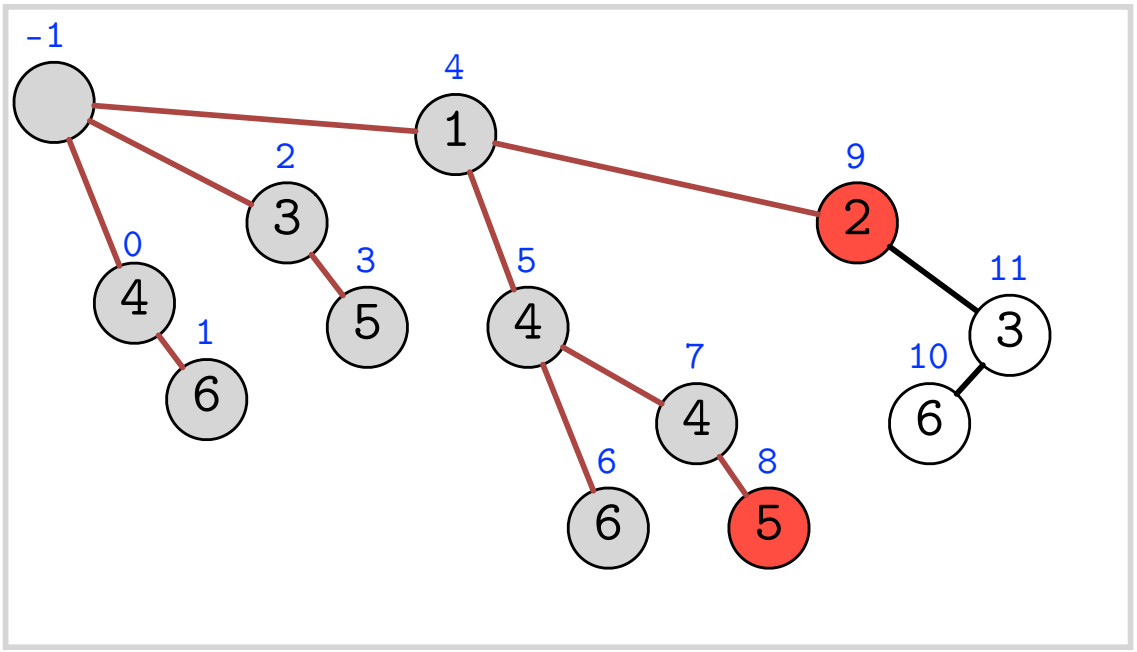
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

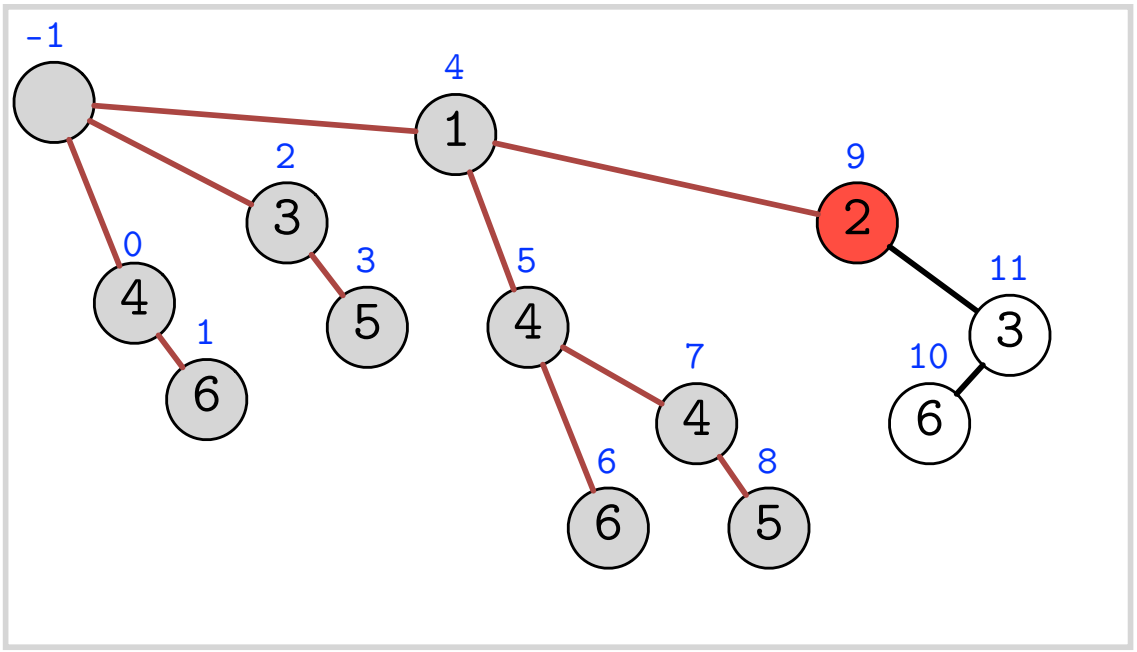
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

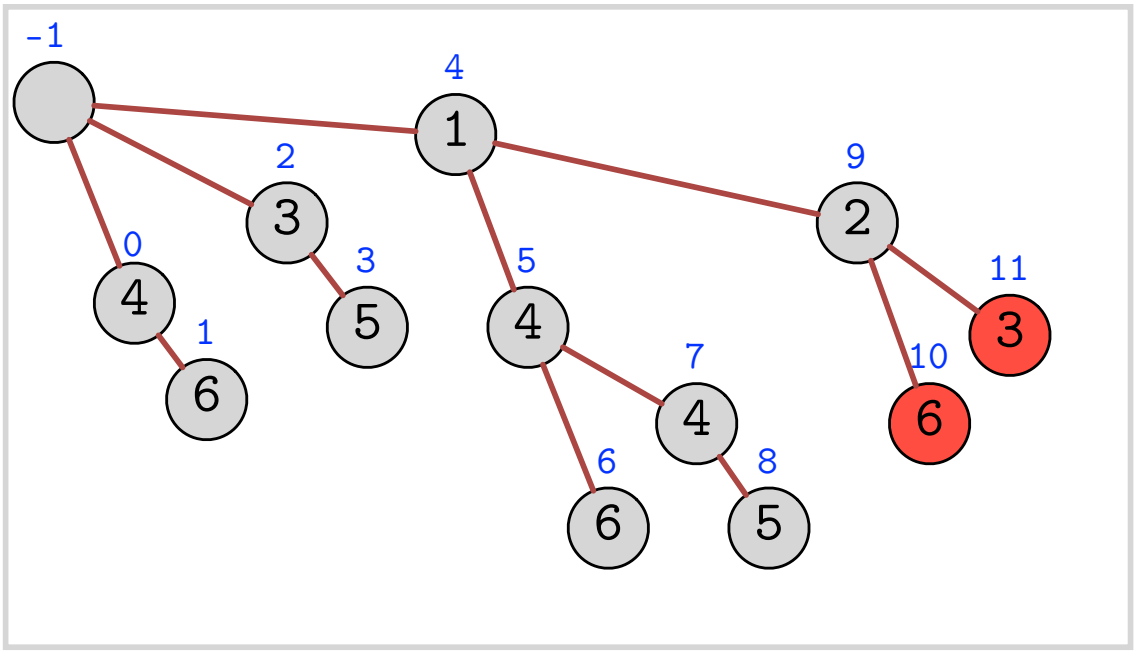
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

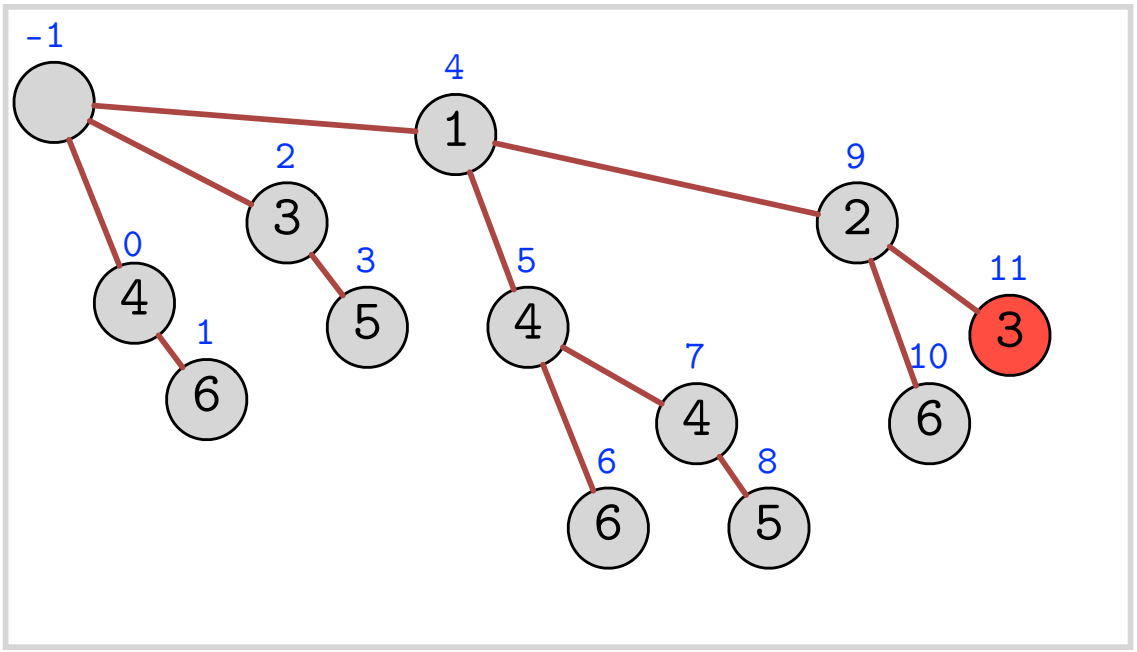
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

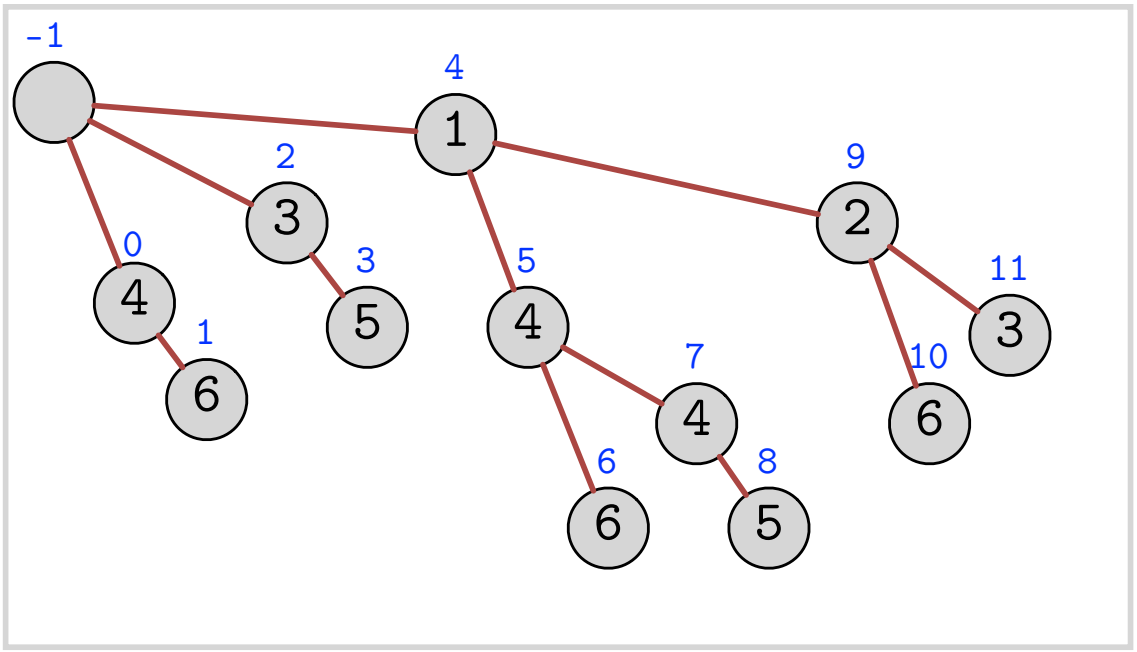
- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Transformation

- Add a new root node to the leaf of the original root
- For each node v (starting at the root) take the right child w, and add the nodes on the leftmost path from w as children of v



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

- Node with inorder i in CT becomes node with preorder $i + 1$ in the general tree
- So we can identify the node of array element i by selecting the $(i + 2)$th opening parenthesis in the balanced parentheses sequence of the general tree (+1 for the added root node, +1 for index shift by one)
- Let BP be the balanced parenthes sequence of the general tree

For $0 \leq i \leq j < n$ we get:

00 **rmq**$_A(i,j)$
01 $ipos \leftarrow select(i + 2, (, BP)$
02 $jpos \leftarrow select(j + 2, (, BP)$
03 **return** $rank(rmq^{\pm 1}_{excess}(ipos - 1, jpos), (, BP) - 1$

Where $rmq^{\pm 1}_{excess}$ returns the rightmost position of the minimal value.

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

- Node with inorder i in CT becomes node with preorder $i + 1$ in the general tree
- So we can identify the node of array element i by selecting the $(i + 2)$th opening parenthesis in the balanced parentheses sequence of the general tree (+1 for the added root node, +1 for index shift by one)
- Let BP be the balanced parenthes sequence of the general tree

For $0 \leq i \leq j < n$ we get:

```
00  rmq_A(i,j)
01      ipos ← select(i + 2, (, BP)
02      jpos ← select(j + 2, (, BP)
03      return rank(rmq^{±1}_{excess}(ipos − 1, jpos), (, BP) − 1
```

Where $rmq^{\pm 1}_{excess}$ returns the rightmost position of the minimal value.

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

- Node with inorder i in CT becomes node with preorder $i + 1$ in the general tree
- So we can identify the node of array element i by selecting the $(i + 2)$th opening parenthesis in the balanced parentheses sequence of the general tree (+1 for the added root node, +1 for index shift by one)
- Let BP be the balanced parenthes sequence of the general tree

For $0 \leq i \leq j < n$ we get:

```
00  rmq_A(i,j)
01      ipos ← select(i + 2, (, BP)
02      jpos ← select(j + 2, (, BP)
03      return rank(rmq^{±1}_excess(ipos − 1, jpos), (, BP) − 1
```

Where $rmq^{\pm 1}_{excess}$ returns the rightmost position of the minimal value.

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

- Node with inorder i in CT becomes node with preorder $i + 1$ in the general tree
- So we can identify the node of array element i by selecting the $(i + 2)$th opening parenthesis in the balanced parentheses sequence of the general tree (+1 for the added root node, +1 for index shift by one)
- Let BP be the balanced parenthes sequence of the general tree

For $0 \leq i \leq j < n$ we get:

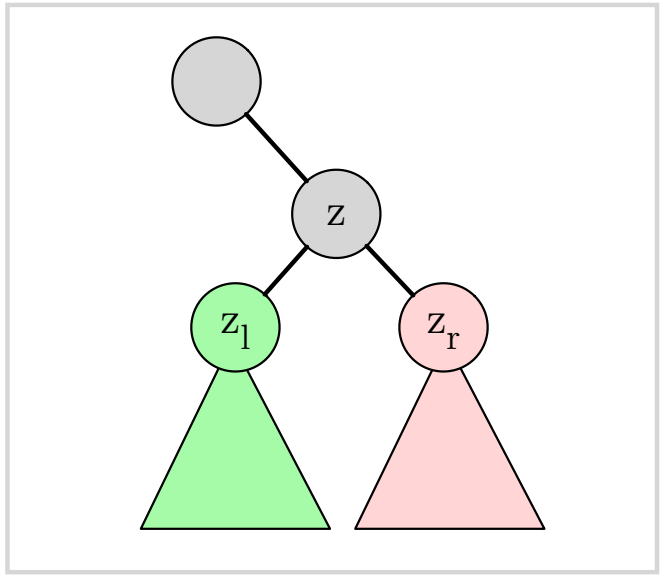
00 **rmq**$_A(i,j)$
01 $ipos \leftarrow select(i + 2, (, BP)$
02 $jpos \leftarrow select(j + 2, (, BP)$
03 **return** $rank(rmq^{\pm 1}_{excess}(ipos - 1, jpos), (, BP) - 1$

Where $rmq^{\pm 1}_{excess}$ returns the rightmost position of the minimal value.

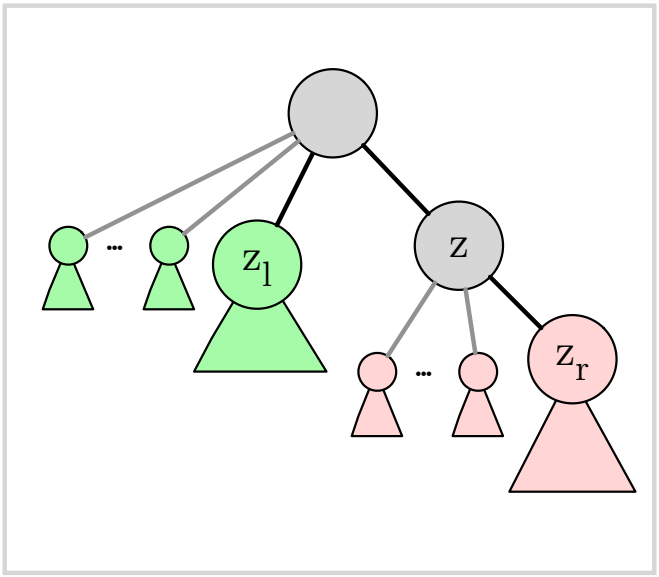
$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Proof sketch: Let v and w be the nodes of $A[i]$ and $A[j]$ and z be the LCA of v and w in the Cartesian Tree. Node v is in the left subtree and w in the right subtree of z.

Situation in binary tree



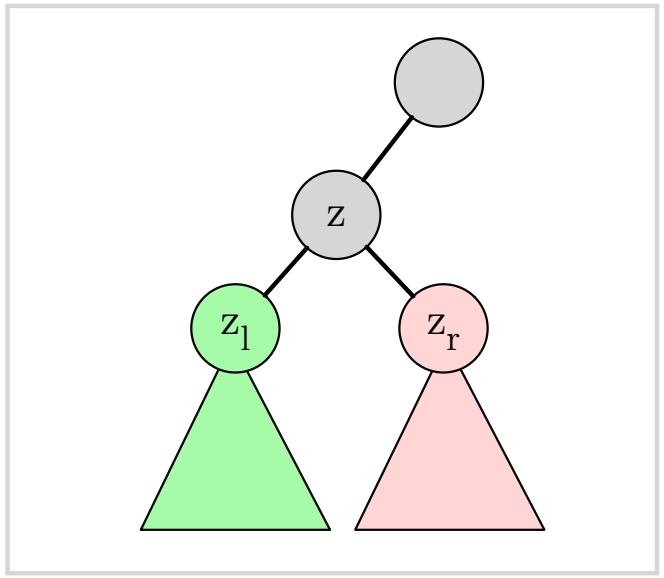
Situation in general tree



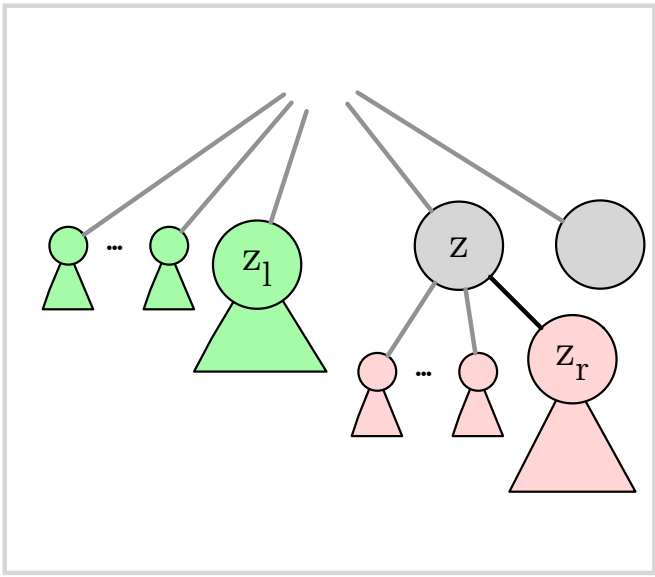
$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Proof sketch: Let v and w be the nodes of $A[i]$ and $A[j]$ and z be the LCA of v and w in the Cartesian Tree. Node v is in the left subtree and w in the right subtree of z.

Situation in binary tree



Situation in general tree



$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

First case: $v \neq z$ and $w \neq z$

- v's opening parenthesis is the green area
- w's opening parenthesis is the the red area
- the (righmost) $\pm 1 RMQ$ will return the position p of the closing parenthesis of z_l
- z's opening parenthesis is at position $p + 1$ by construction
- $r = rank(p) - 1$ corresponds to the preorder number of z in the general tree
- which in turns corresponds to the inorder number in CT
- which in turns corresponds to the index of the minimum in $A[i, j]$

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

Second case: $v \neq z$ and $w = z$:

- v's opening parenthesis is the green area
- w's opening parenthesis is z's opening parenthesis now
- the (righmost) $\pm 1 RMQ$ will return the position p of the closing parenthesis of z_l
- z's opening parenthesis is at position $p + 1$ by construction
- $r = rank(p) - 1$ corresponds to the preorder number of z in the general tree
- which in turns corresponds to the inorder number in CT
- which in turns corresponds to the index of the minimum in $A[i, j]$

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

Third case: $v = z$ and $w \neq z$:

- v's opening parenthesis is z's opening parenthesis now
- w's opening parenthesis is the the red area
- the (righmost) $\pm 1 RMQ$ will return the position p of the closing parenthesis of z_l
- z's opening parenthesis is at position $p + 1$ by construction
- $r = rank(p) - 1$ corresponds to the preorder number of z in the general tree
- which in turns corresponds to the inorder number in CT
- which in turns corresponds to the index of the minimum in $A[i, j]$

$\langle O(n), O(1) \rangle$ solution ($2n + o(n)$ bits)

Karlsruhe Institute of Technology

Third case: $v = z$ and $w = z$:

- v's opening parenthesis is z's opening parenthesis now
- w's opening parenthesis is z's opening parenthesis now
- the (righmost) $\pm 1 RMQ$ will return the position p of the closing parenthesis of z_l
- z's opening parenthesis is at position $p + 1$ by construction
- $r = rank(p) - 1$ corresponds to the preorder number of z in the general tree
- which in turns corresponds to the inorder number in CT
- which in turns corresponds to the index of the minimum in $A[i, j]$

Conclusion

Range Minimum Queries (RMQ)s over an array A can be answered in constant time after prepocessing a $2n + o(n)$ space data structure in linear time.
Applications:

- Compressed suffix trees
- Document retrieval
- Weighted query completion
- ...

Literature

- M.A. Bender, M. Farach-Colton: The LCA Problem Revisited. (LATIN 2000)
- K. Sadakane: Compressed Suffix Trees with Full Functionality. (TCS 2007)
- H. Ferrada, G. Navarro: Improved Range Minimum Queries (DCC 2016)

Burrows-Wheeler-Transformation

Einführung

Karlsruher Institut für Technologie

Burrows, Wheeler (1983, 1994)

- "nur" eine Umordnung des Textes,
  $\rightarrow$ gruppiert Zeichen mit ähnlichem Kontext, reversibel
- ursprünglich eingeführt zur Textkompression
  $\rightarrow$ Teilschritt von bzip2
- später auch für Textindizierung verwendet
  $\rightarrow$ z.B. Rückwärtssuche, Berechnung des LCP-Array

Burrows-Wheeler-Transformation

Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$

Definition

- $\mathrm{SA}[i] \mathrel{\hat{=}}$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)

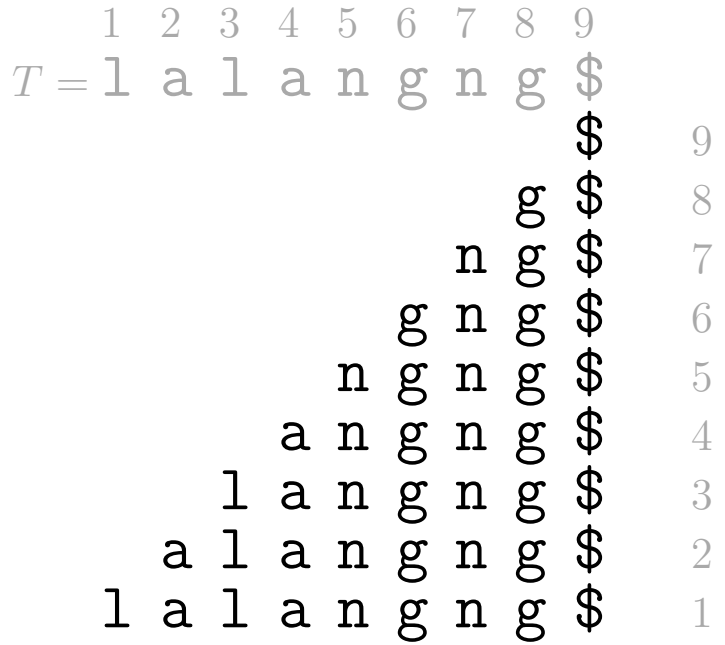
Burrows-Wheeler-Transformation

Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)

Definition

- $\mathrm{SA}[i] \mathrel{\hat{=}}$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)

Definition

- $\mathrm{SA}[i] \mathrel{\hat{=}}$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)

	1	2	3	4	5	6	7	8	9	
$T =$	l	a	l	a	n	g	n	g	$	
									$	9
		a	l	a	n	g	n	g	$	2
			a	n	g	n	g	$		4
							g	$		8
						g	n	g	$	6
	l	a	l	a	n	g	n	g	$	1
			l	a	n	g	n	g	$	3
							n	g	$	7
					n	g	n	g	$	5

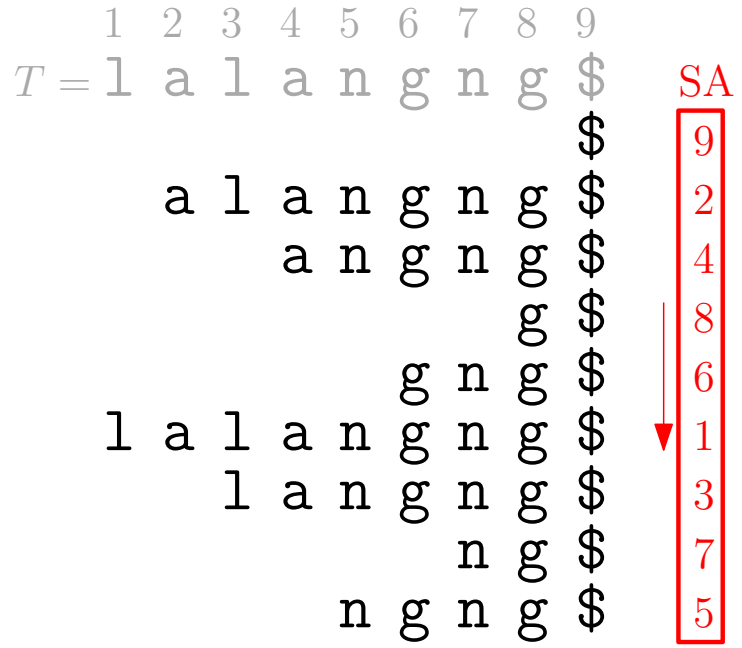
Burrows-Wheeler-Transformation

Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)

Definition

- $\mathrm{SA}[i] \mathrel{\hat{=}}$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

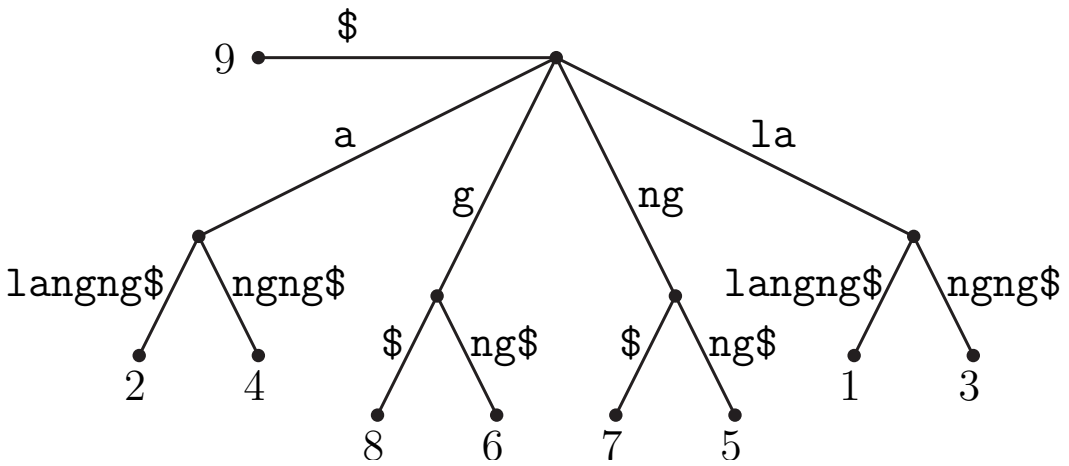
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)

	1	2	3	4	5	6	7	8	9	
$T =$	l	a	l	a	n	g	n	g	$	SA
									$	9
		a	l	a	n	g	n	g	$	2
			a	n	g	n	g	$		4
							g	$		8
					g	n	g	$		6
	l	a	l	a	n	g	n	g	$	1
		l	a	n	g	n	g	$		3
						n	g	$		7
				n	g	n	g	$		5

Definition

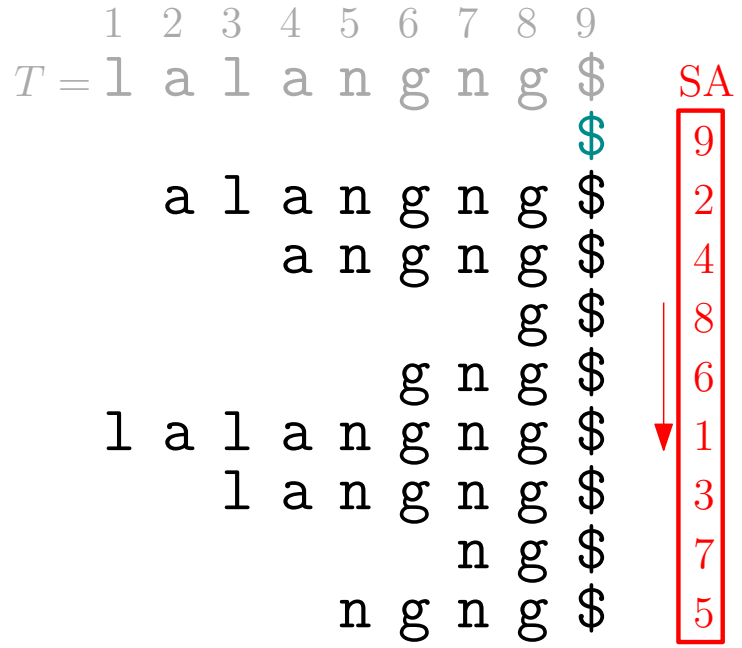
- $\mathrm{SA}[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

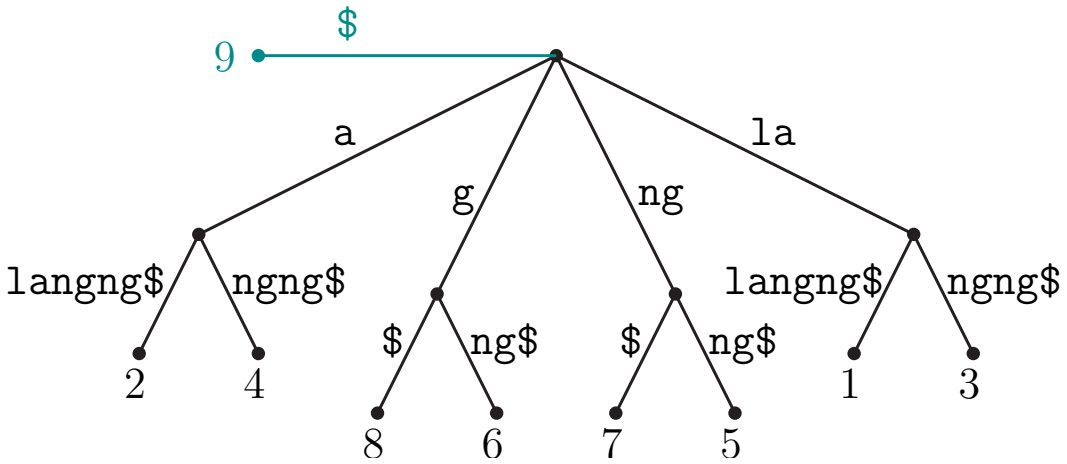
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)



Definition

- SA$[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

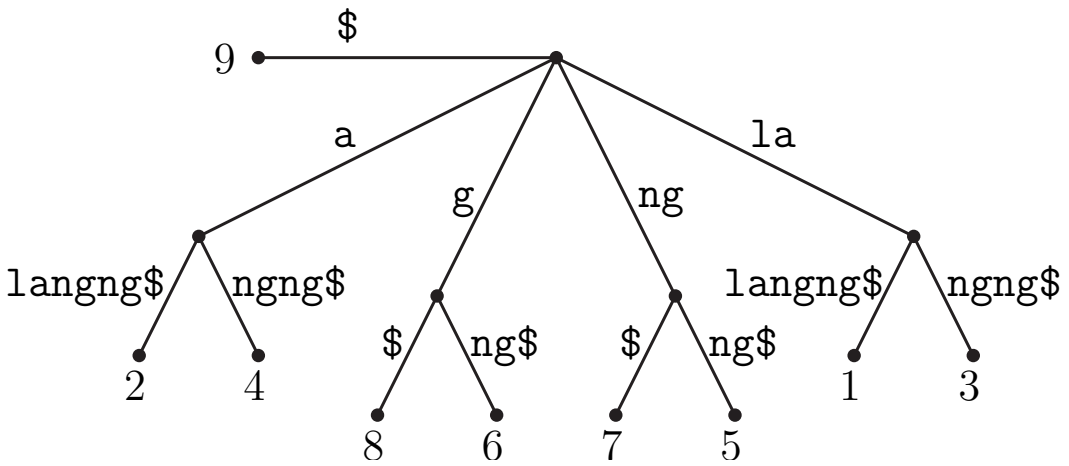
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)

	1	2	3	4	5	6	7	8	9	
$T =$	l	a	l	a	n	g	n	g	$	SA
									$	9
				a	l	a	n	g	n	g	$	2
					a	n	g	n	g	$	4
								g	$	8
							g	n	g	$	6
	l	a	l	a	n	g	n	g	$	1
			l	a	n	g	n	g	$	3
							n	g	$	7
					n	g	n	g	$	5

Definition

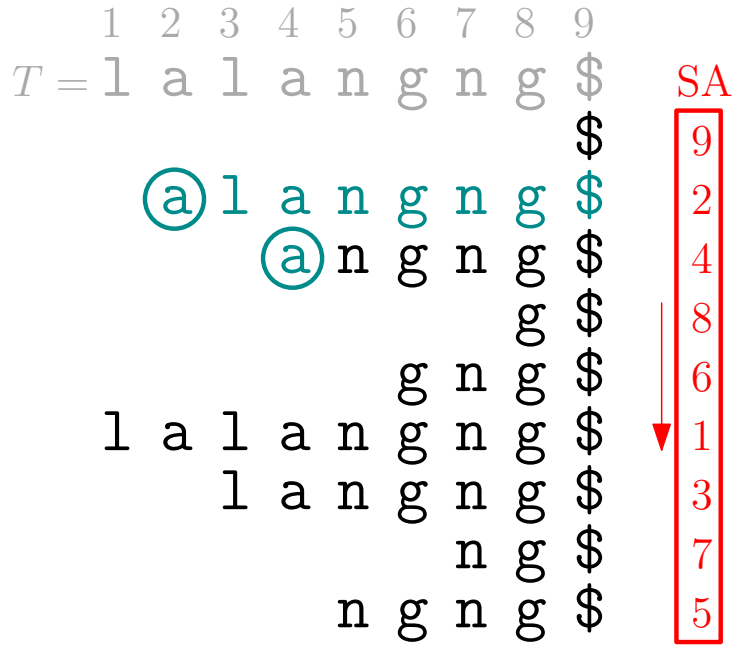
- $\mathrm{SA}[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

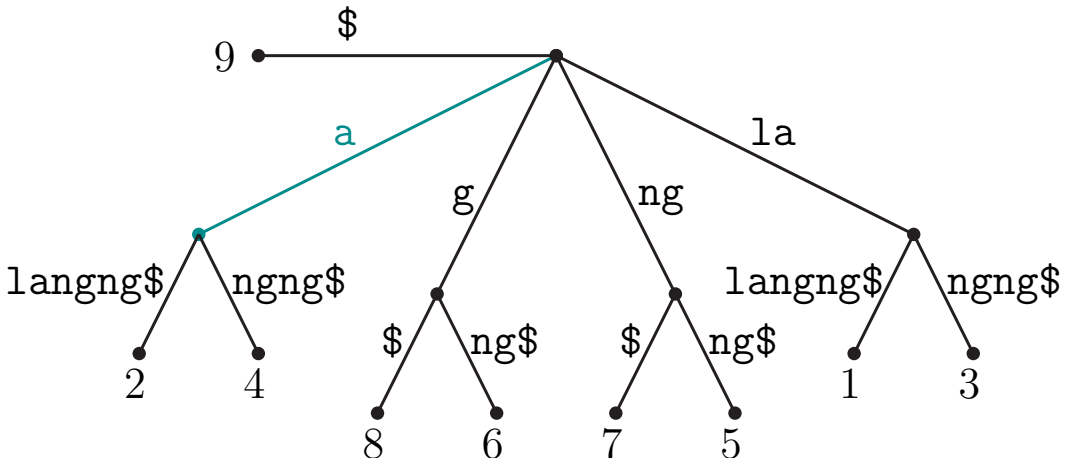
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)



Definition

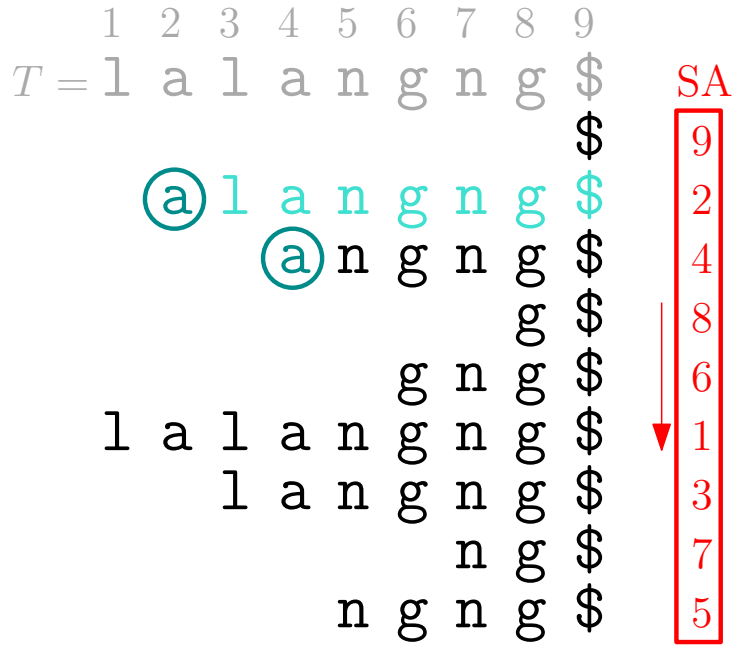
- SA$[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

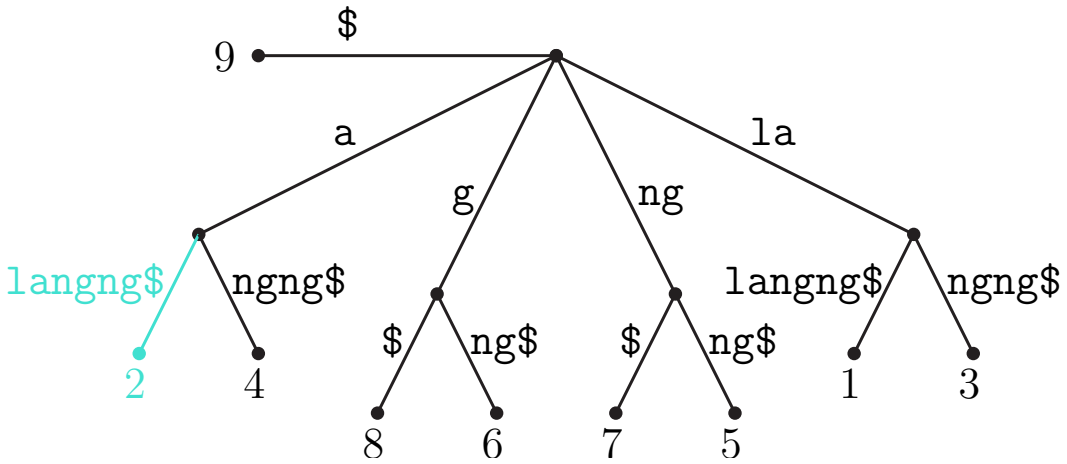
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)



Definition

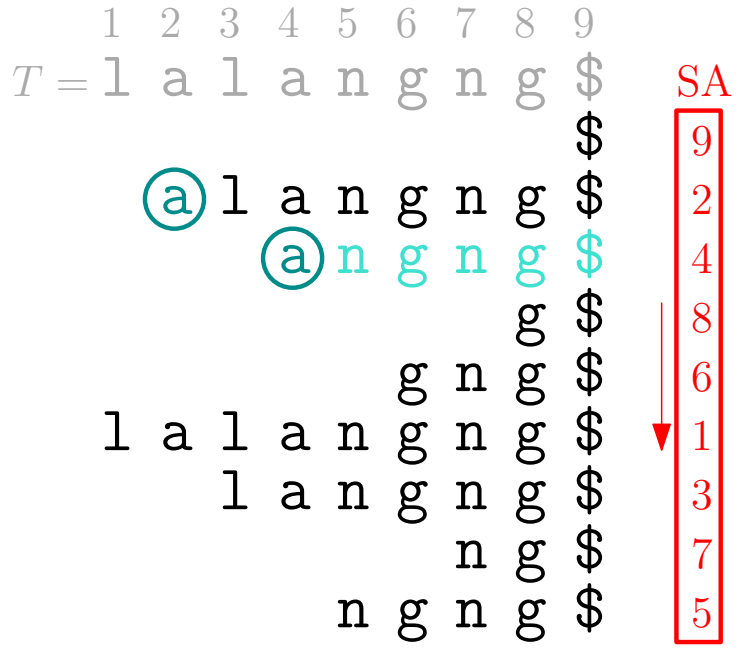
- SA$[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

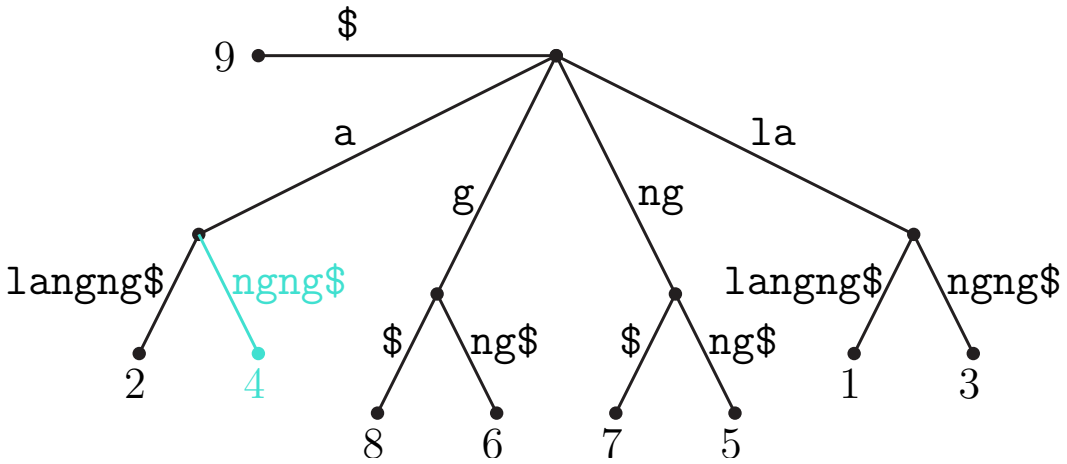
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)



Definition

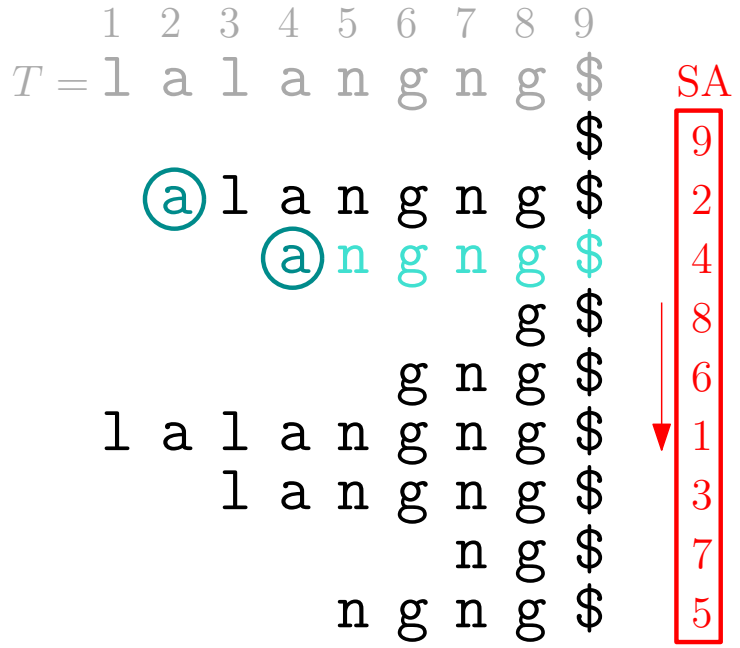
- SA$[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



Burrows-Wheeler-Transformation

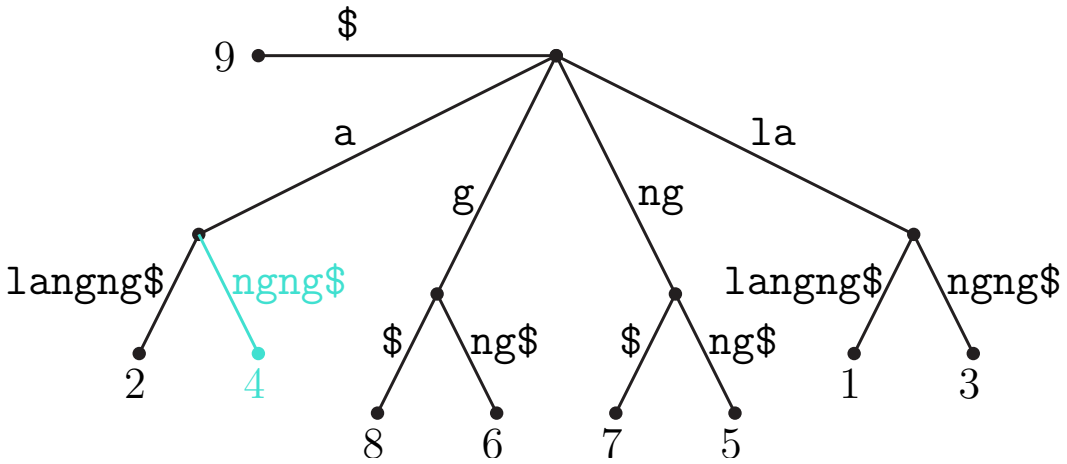
Wiederholung: Suffix-Arrays (und Suffixbäume)

- Textindizierung
  $\rightarrow$ schnelle Suche in Texten
- kann in $O(n)$ erzeugt werden
  (DC3-Algorithmus)



Definition

- SA$[i] \triangleq$ Index des i-ten sortierten Suffix
  (einer Zeichenkette T)



USW.

Burrows-Wheeler-Transformation

Transformation

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$

Burrows-Wheeler-Transformation

Transformation

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$
	a	l	a	n	g	n	g	$	l

Burrows-Wheeler-Transformation

Transformation

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$
	a	l	a	n	g	n	g	$	l
	l	a	n	g	n	g	$	l	a

Burrows-Wheeler-Transformation

Transformation

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$
	a	l	a	n	g	n	g	$	l
	l	a	n	g	n	g	$	l	a
	a	n	g	n	g	$	l	a	l

Burrows-Wheeler-Transformation

Transformation

$$
T = \begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\mathtt{l} & \mathtt{a} & \mathtt{l} & \mathtt{a} & \mathtt{n} & \mathtt{g} & \mathtt{n} & \mathtt{g} & \mathtt{\$} \\
\mathtt{a} & \mathtt{l} & \mathtt{a} & \mathtt{n} & \mathtt{g} & \mathtt{n} & \mathtt{g} & \mathtt{\$} & \mathtt{l} \\
\mathtt{l} & \mathtt{a} & \mathtt{n} & \mathtt{g} & \mathtt{n} & \mathtt{g} & \mathtt{\$} & \mathtt{l} & \mathtt{a} \\
\mathtt{a} & \mathtt{n} & \mathtt{g} & \mathtt{n} & \mathtt{g} & \mathtt{\$} & \mathtt{l} & \mathtt{a} & \mathtt{l} \\
 & & & & \vdots & & & &
\end{array}
$$

Burrows-Wheeler-Transformation

Transformation

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$
	a	l	a	n	g	n	g	$	l
	l	a	n	g	n	g	$	l	a
	a	n	g	n	g	$	l	a	l
	n	g	n	g	$	l	a	l	a
	g	n	g	$	l	a	l	a	n
	n	g	$	l	a	l	a	n	g
	g	$	l	a	l	a	n	g	n
	$	l	a	l	a	n	g	n	g

Burrows-Wheeler-Transformation

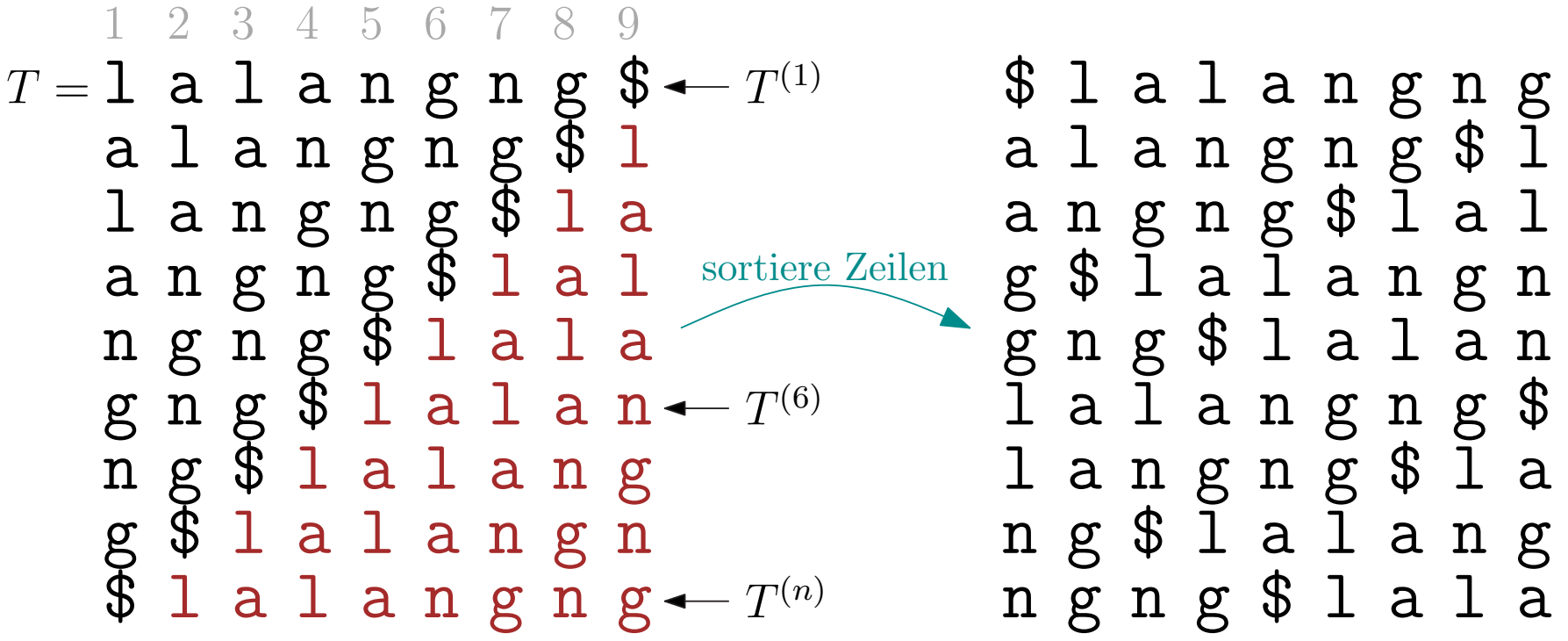
Transformation

	1	2	3	4	5	6	7	8	9	
$T =$	l	a	l	a	n	g	n	g	$	$\leftarrow T^{(1)}$
	a	l	a	n	g	n	g	$	l	
	l	a	n	g	n	g	$	l	a	
	a	n	g	n	g	$	l	a	l	
	n	g	n	g	$	l	a	l	a	
	g	n	g	$	l	a	l	a	n	$\leftarrow T^{(6)}$
	n	g	$	l	a	l	a	n	g	
	g	$	l	a	l	a	n	g	n	
	$	l	a	l	a	n	g	n	g	$\leftarrow T^{(n)}$

- $T^{(i)} \triangleq T$ zyklisch ab Position i, Länge $n = |T|$

Burrows-Wheeler-Transformation

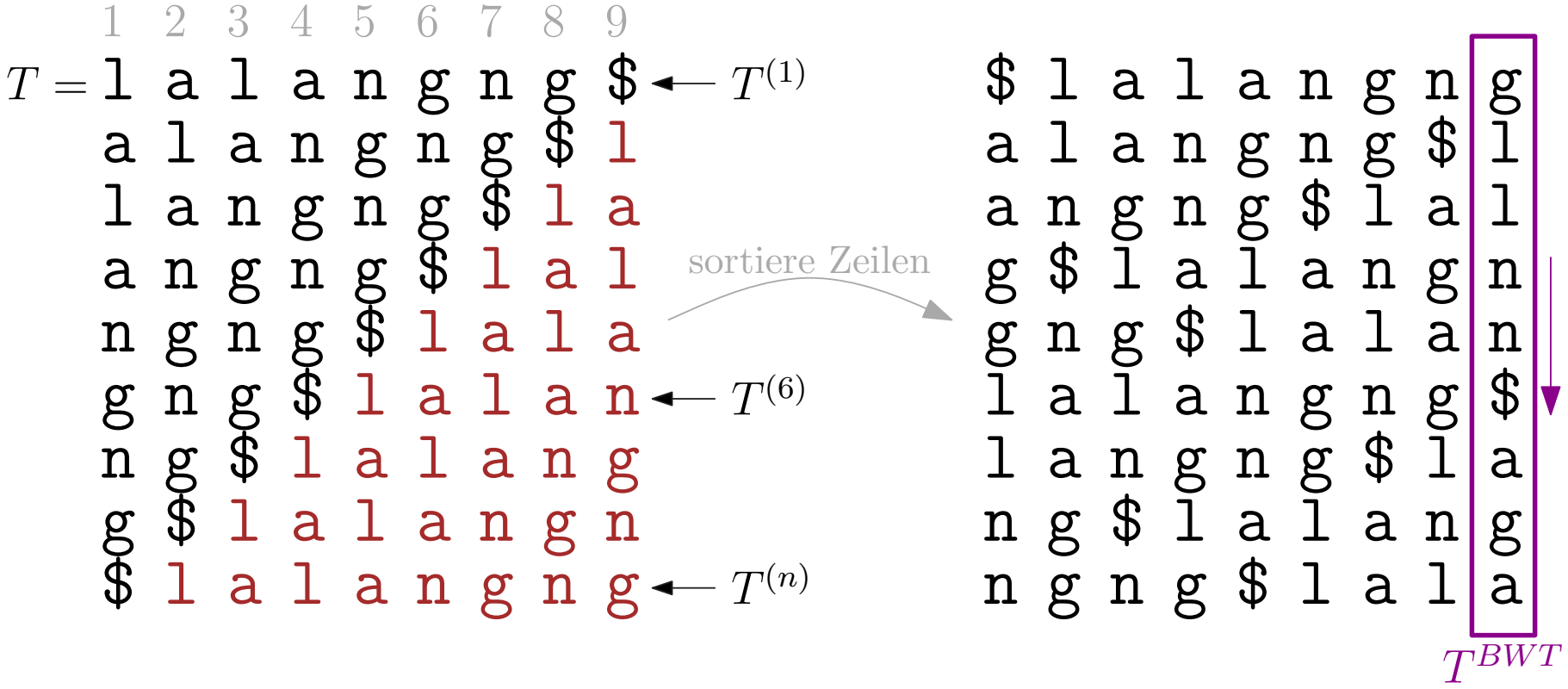
Transformation



- $T^{(i)} \triangleq T$ zyklisch ab Position i, Länge $n = |T|$

Burrows-Wheeler-Transformation

Transformation

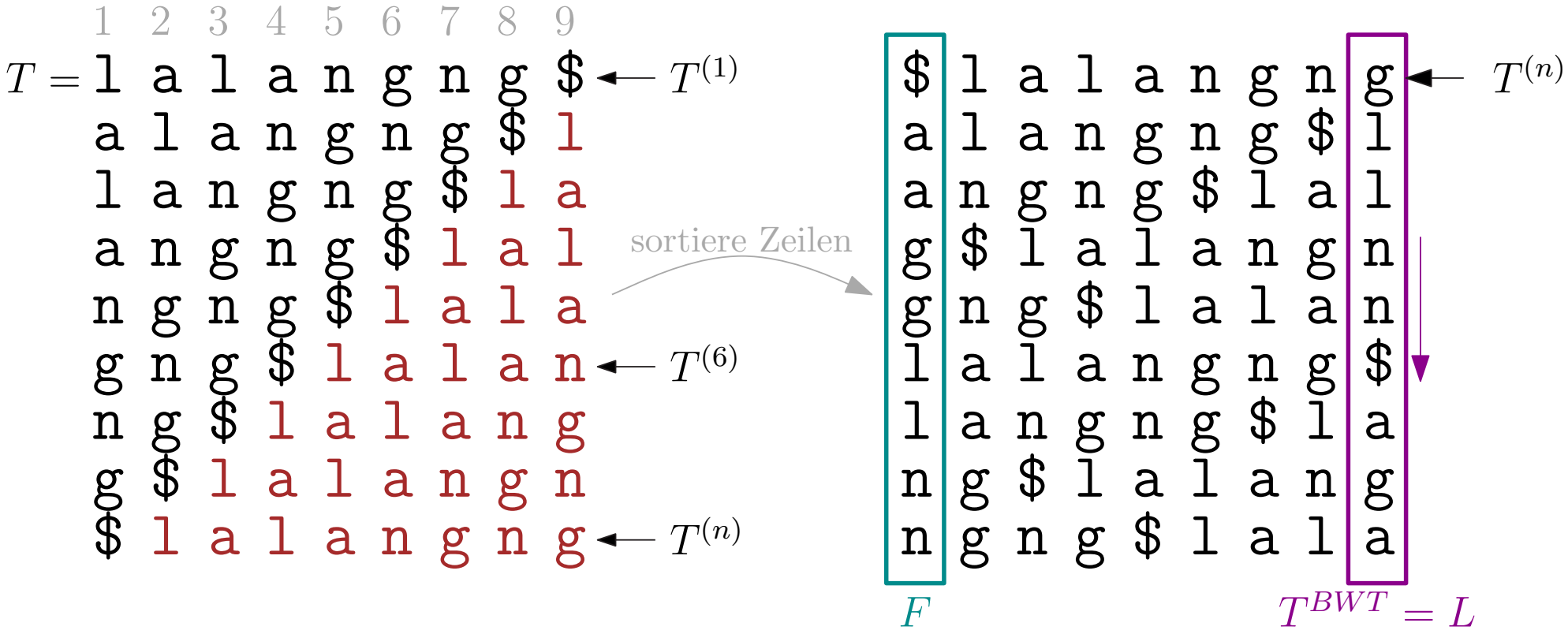


- $T^{(i)} \triangleq T$ zyklisch ab Position i, Länge $n = |T|$
- $T = \texttt{lalangng\$} \quad \rightarrow \quad T^{BWT} = \texttt{gllnn\$aga}$

$\mathcal{O}(n^2 + n \log n)$

Burrows-Wheeler-Transformation

Transformation



- $T^{(i)} \triangleq T$ zyklisch ab Position i, Länge $n = |T|$
- $T = \texttt{lalangng\$} \quad \rightarrow \quad T^{BWT} = \texttt{gllnn\$aga}$

$\mathcal{O}(n^2 + n \log n)$

Burrows-Wheeler-Transformation

Eigenschaften

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$

- BWT in $\mathcal{O}(n)$ berechenbar
- Zeilen enthalten sortierte Suffixe
- Zeichen in letzer Spalte entspricht Zeichen vor Suffix in T

$	l	a	l	a	n	g	n	g	1
a	l	a	n	g	n	g	$	l	2
a	n	g	n	g	$	l	a	l	3
g	$	l	a	l	a	n	g	n	4
g	n	g	$	l	a	l	a	n	5
l	a	l	a	n	g	n	g	$	6
l	a	n	g	n	g	$	l	a	7
n	g	$	l	a	l	a	n	g	8
n	g	n	g	$	l	a	l	a	9

Burrows-Wheeler-Transformation

Eigenschaften

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$

- BWT in $\mathcal{O}(n)$ berechenbar
- Zeilen enthalten sortierte Suffixe
- Zeichen in letzer Spalte entspricht Zeichen vor Suffix in T

$	l	a	l	a	n	g	n	g	1
a	l	a	n	g	n	g	$	l	2
a	n	g	n	g	$	l	a	l	3
g	$	l	a	l	a	n	g	n	4
g	n	g	$	l	a	l	a	n	5
l	a	l	a	n	g	n	g	$	6
l	a	n	g	n	g	$	l	a	7
n	g	$	l	a	l	a	n	g	8
n	g	n	g	$	l	a	l	a	9

Burrows-Wheeler-Transformation

Eigenschaften

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$

- BWT in $\mathcal{O}(n)$ berechenbar
- Zeilen enthalten sortierte Suffixe
- Zeichen in letzer Spalte entspricht Zeichen vor Suffix in T

$	l	a	l	a	n	g	n	g	1
a	l	a	n	g	n	g	$	l	2
a	n	g	n	g	$	l	a	l	3
g	$	l	a	l	a	n	g	n	4
g	n	g	$	l	a	l	a	n	5
l	a	l	a	n	g	n	g	$	6
l	a	n	g	n	g	$	l	a	7
n	g	$	l	a	l	a	n	g	8
n	g	n	g	$	l	a	l	a	9

Burrows-Wheeler-Transformation

Eigenschaften

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$

- BWT in $\mathcal{O}(n)$ berechenbar
- Zeilen enthalten sortierte Suffixe
- Zeichen in letzer Spalte entspricht Zeichen vor Suffix in T

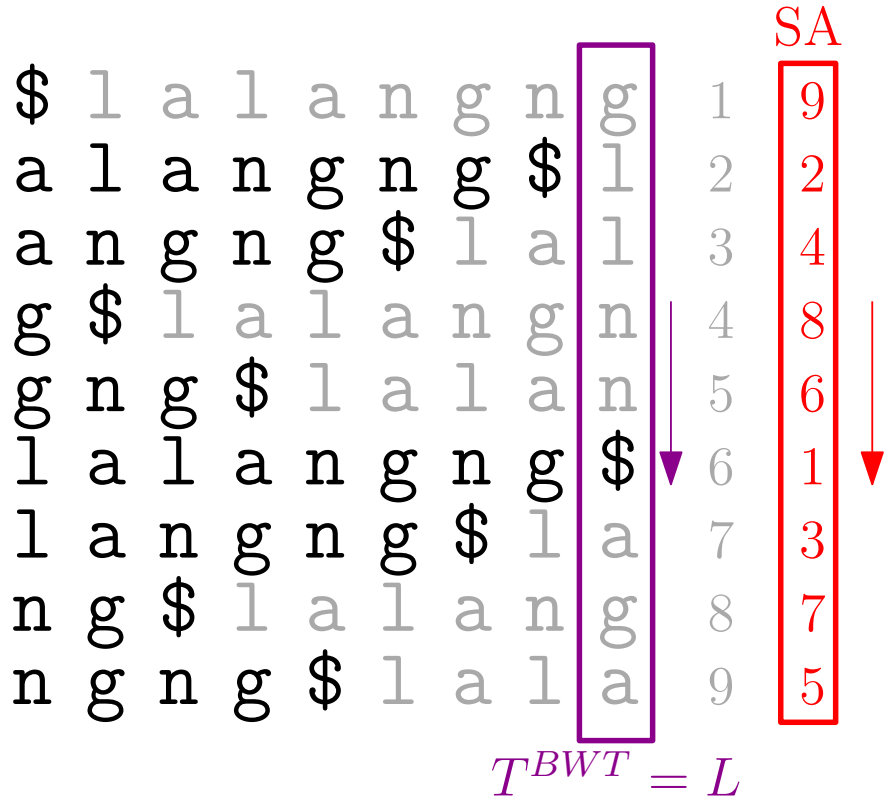
$	l	a	l	a	n	g	n	g	1
a	l	a	n	g	n	g	$	l	2
a	n	g	n	g	$	l	a	l	3
g	$	l	a	l	a	n	g	n	4
g	n	g	$	l	a	l	a	n	5
l	a	l	a	n	g	n	g	$	6
l	a	n	g	n	g	$	l	a	7
n	g	$	l	a	l	a	n	g	8
n	g	n	g	$	l	a	l	a	9

$T^{BWT} = L$

Burrows-Wheeler-Transformation

Eigenschaften

	1	2	3	4	5	6	7	8	9
$T =$	l	a	l	a	n	g	n	g	$



- BWT in $\mathcal{O}(n)$ berechenbar
- Zeilen enthalten sortierte Suffixe
- Zeichen in letzer Spalte entspricht Zeichen vor Suffix in T
- $T^{BWT}[i] = L[i] = T[SA[i] - 1] = T^{(SA[i])}[n]$
  ($T^{BWT}[i]$ ist das Zeichen vor dem i-ten Suffix in T)

Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)

\$	l	a	l	a	n	g	n	g
a	l	a	n	g	n	g	\$	l
a	n	g	n	g	\$	l	a	l
g	\$	l	a	l	a	n	g	n
g	n	g	\$	l	a	l	a	n
l	a	l	a	n	g	n	g	\$
l	a	n	g	n	g	\$	l	a
n	g	\$	l	a	l	a	n	g
n	g	n	g	\$	l	a	l	a

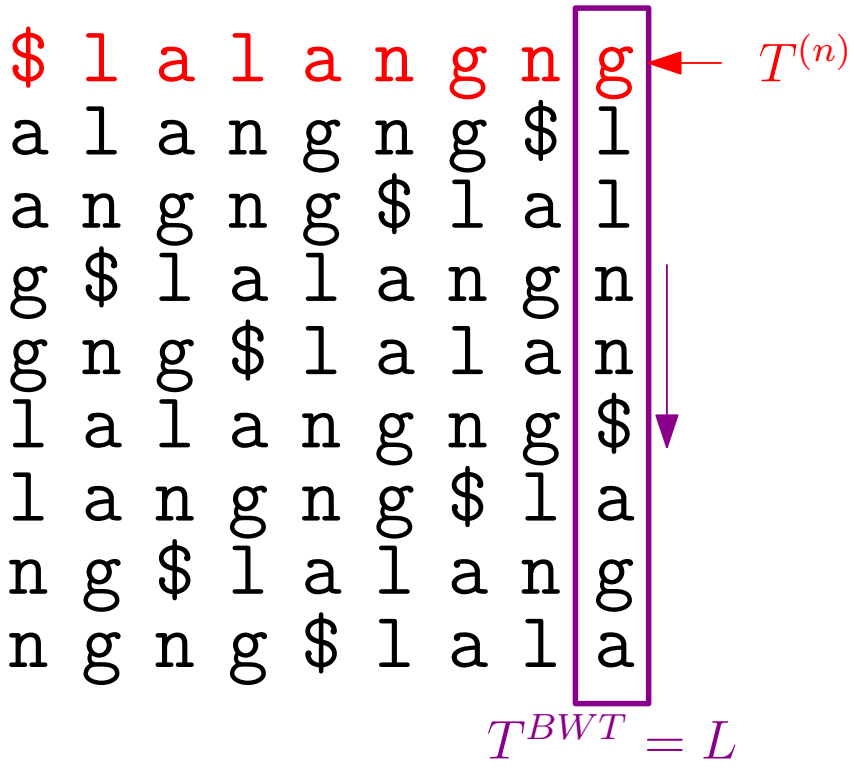
$T^{BWT} = L$

Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)



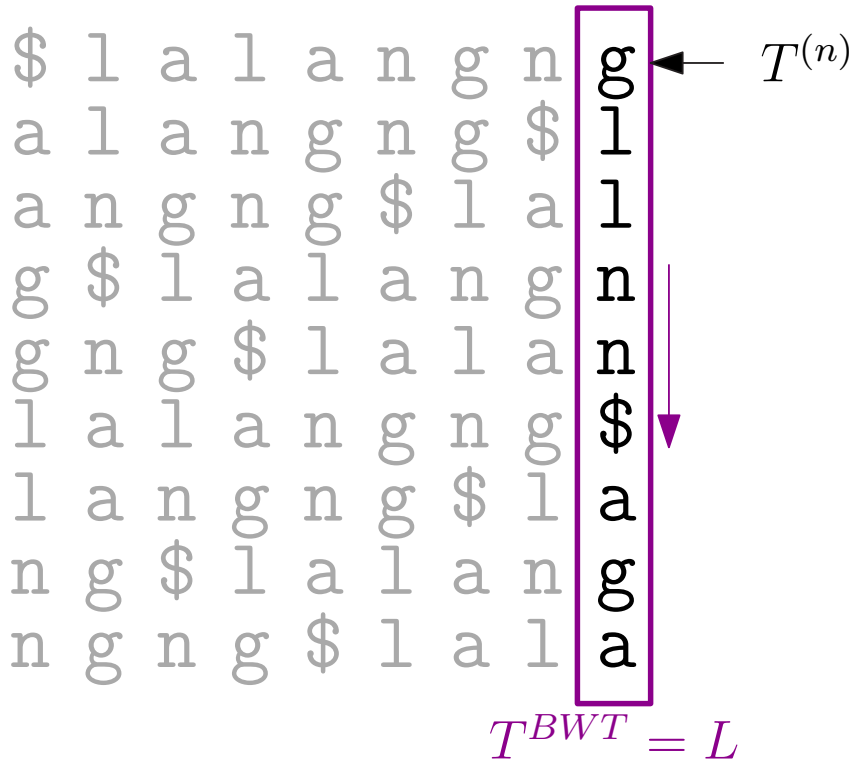
Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

Karlsruher Institut für Technologie

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)

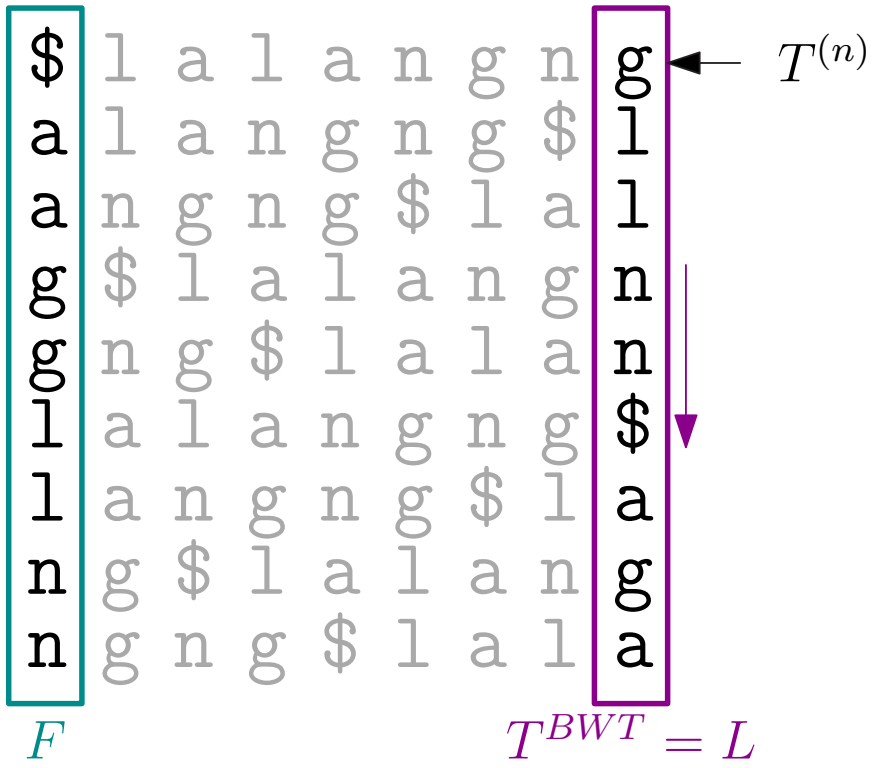


Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

$$T^{BWT} = \underset{1}{\texttt{g}}\ \underset{2}{\texttt{l}}\ \underset{3}{\texttt{l}}\ \underset{4}{\texttt{n}}\ \underset{5}{\texttt{n}}\ \underset{6}{\texttt{\$}}\ \underset{7}{\texttt{a}}\ \underset{8}{\texttt{g}}\ \underset{9}{\texttt{a}}$$

- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)

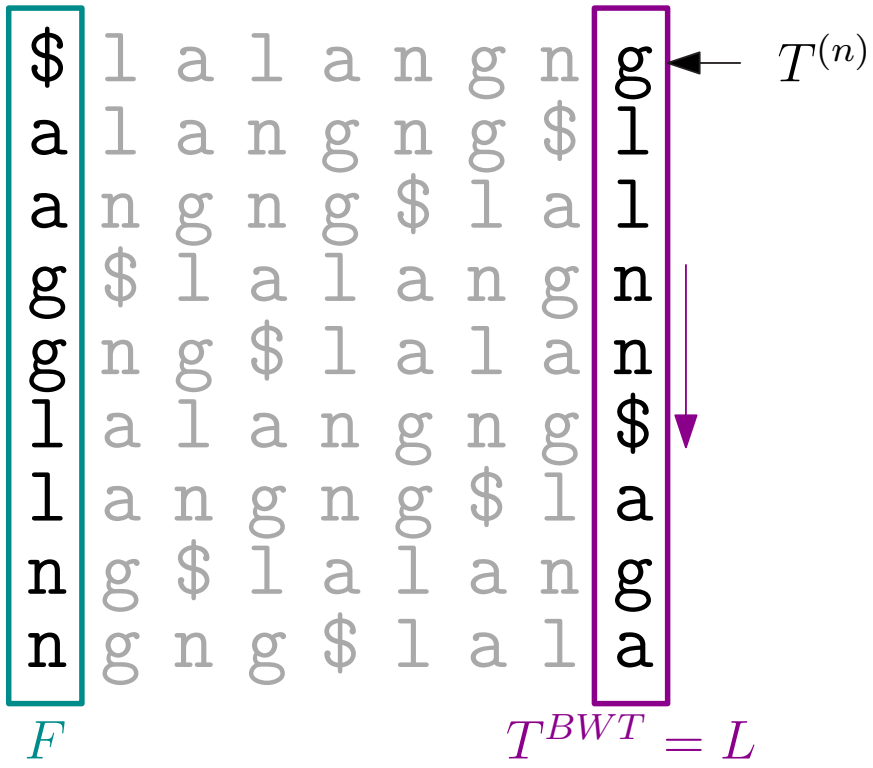


Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

$$T^{BWT} = \underset{1}{\texttt{g}}\ \underset{2}{\texttt{l}}\ \underset{3}{\texttt{l}}\ \underset{4}{\texttt{n}}\ \underset{5}{\texttt{n}}\ \underset{6}{\texttt{\$}}\ \underset{7}{\texttt{a}}\ \underset{8}{\texttt{g}}\ \underset{9}{\texttt{a}}$$

- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)



Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)

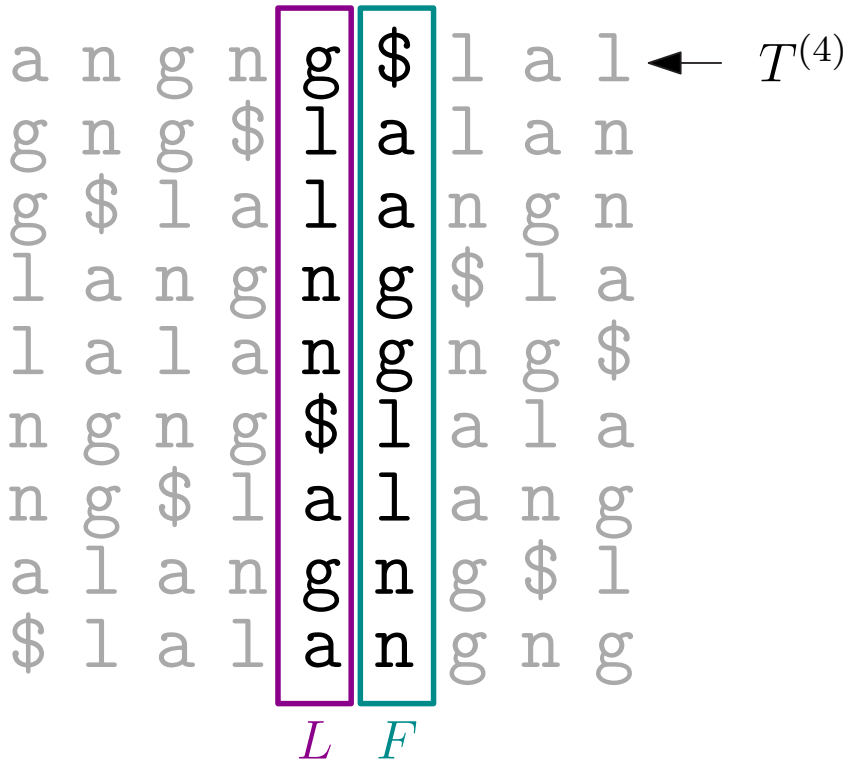
							L	F	
l	a	l	a	n	g	n	g	\$	← $T^{(1)}$
l	a	n	g	n	g	\$	l	a	
n	g	n	g	\$	l	a	l	a	
\$	l	a	l	a	n	g	n	g	
n	g	\$	l	a	l	a	n	g	
a	l	a	n	g	n	g	\$	l	
a	n	g	n	g	\$	l	a	l	
g	\$	l	a	l	a	n	g	n	
g	n	g	\$	l	a	l	a	n	

Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

Karlsruher Institut für Technologie

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

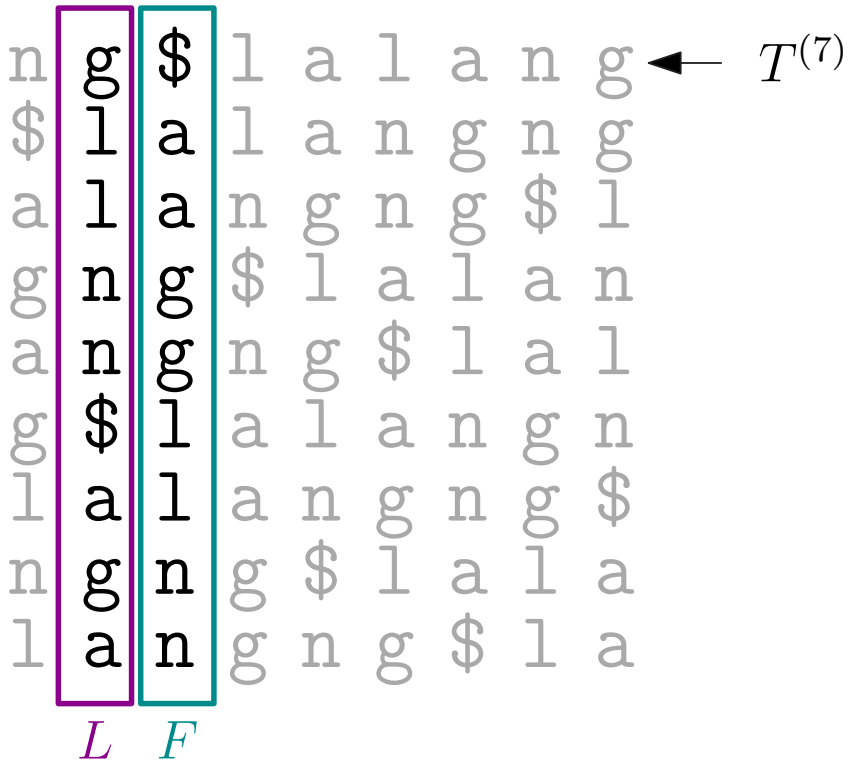


- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)

Burrows-Wheeler-Transformation

Rücktransformation – Vorüberlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$



- betrachte Matrix aus Transformation
  - erste Zeile enthält Lösung $T^{(n)}$
  - $T^{BWT} = L$ gegeben
  - F leicht zu bestimmen (sortiere L)
  - L ist *immer* Spalte vor F (zyklisch)

Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

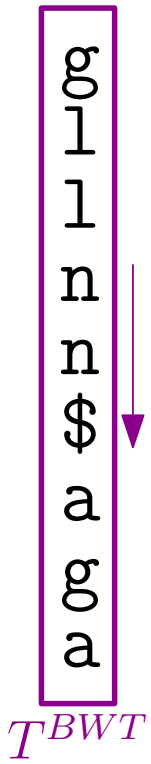
- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert



T^{BWT}

Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

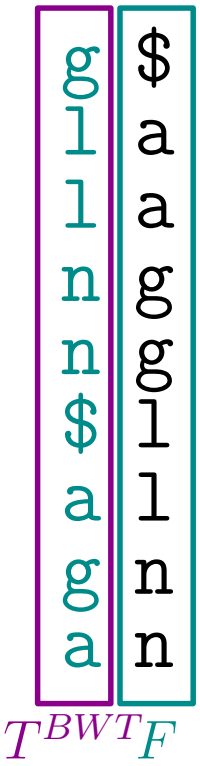
- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

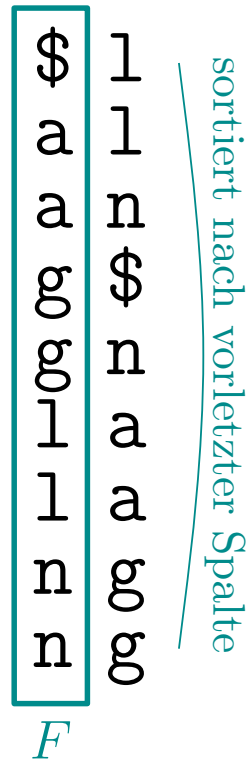


Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

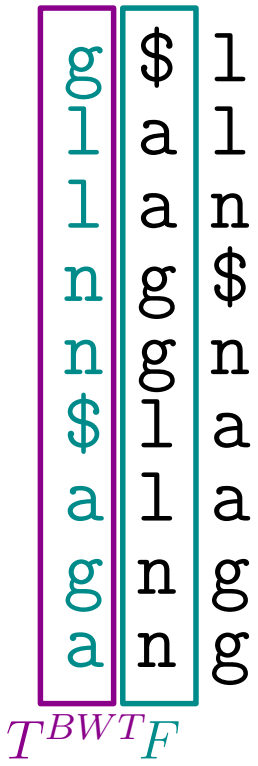


Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert



Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

$$\begin{array}{|c|cc} \hline \mathtt{\$} & \mathtt{l} & \mathtt{a} \\ \mathtt{a} & \mathtt{l} & \mathtt{a} \\ \mathtt{a} & \mathtt{n} & \mathtt{g} \\ \mathtt{g} & \mathtt{\$} & \mathtt{l} \\ \mathtt{l} & \mathtt{n} & \mathtt{g} \\ \mathtt{l} & \mathtt{a} & \mathtt{l} \\ \mathtt{n} & \mathtt{a} & \mathtt{n} \\ \mathtt{n} & \mathtt{g} & \mathtt{\$} \\ \mathtt{n} & \mathtt{g} & \mathtt{n} \\ \hline \end{array}$$

F

sortiert nach drittletzter Spalte

Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

T^{BWT}	F		
g	$	l	a
l	a	l	a
l	a	n	g
n	g	$	l
n	g	n	g
$	l	a	l
a	l	a	n
g	n	g	$
a	n	g	n

Burrows-Wheeler-Transformation

Rücktransformation

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

$$\ldots \quad \begin{array}{|c|c|}\hline \texttt{g} & \texttt{\$} \\ \texttt{l} & \texttt{a} \\ \texttt{l} & \texttt{a} \\ \texttt{n} & \texttt{g} \\ \texttt{n} & \texttt{g} \\ \texttt{\$} & \texttt{l} \\ \texttt{a} & \texttt{l} \\ \texttt{g} & \texttt{n} \\ \texttt{a} & \texttt{n} \\ \hline \end{array} \begin{array}{l} \texttt{l a} \\ \texttt{l a} \\ \texttt{n g} \\ \texttt{\$ l} \\ \texttt{n g} \\ \texttt{a l} \\ \texttt{a n} \\ \texttt{g \$} \\ \texttt{g n} \end{array}$$

T^{BWT} F

Burrows-Wheeler-Transformation

Rücktransformation

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

```
$ l a l a n g n g
a l a n g n g $ l
a n g n g $ l a l
g $ l a l a n g n
g n g $ l a l a n
l a l a n g n g $
l a n g n g $ l a
n g $ l a l a n g
n g n g $ l a l a
```

Burrows-Wheeler-Transformation

Rücktransformation

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a

$	l	a	l	a	n	g	n	g	← $T^{(n)}$
a	l	a	n	g	n	g	$	l	
a	n	g	n	g	$	l	a	l	
g	$	l	a	l	a	n	g	n	
g	n	g	$	l	a	l	a	n	
l	a	l	a	n	g	n	g	$	
l	a	n	g	n	g	$	l	a	
n	g	$	l	a	l	a	n	g	
n	g	n	g	$	l	a	l	a	

- schreibe T^{BWT} in Spaltenform
- sortiere zeilenweise
- schreibe T^{BWT} in Spaltenform davor
- wiederhole bis $|T^{BWT}|$ mal sortiert

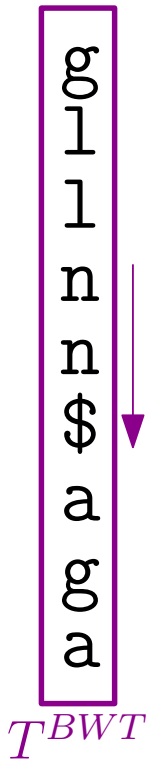
- $T^{BWT} = \texttt{gllnn\$aga} \quad \rightarrow \quad T = \texttt{lalangng\$}$ $\mathcal{O}(n^2 \log n)$

Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

$$\begin{array}{rccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ T^{BWT} = & \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \\ T = & \multicolumn{4}{c}{\text{———————}} & ? & \multicolumn{4}{c}{\text{———————}} \end{array}$$

- Wie lautet das Vorgängerzeichen?
  $\rightarrow$ *last-to-front mapping* $LF[\cdot]$
  (Position in $L[\cdot]$ an der Vorgänger steht)

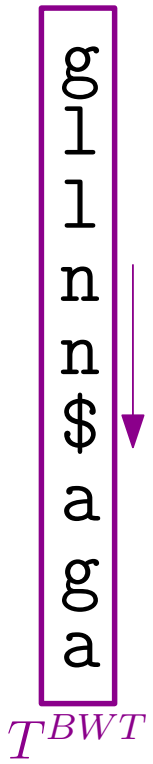


Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$									$

- Wie lautet das Vorgängerzeichen?
  $\rightarrow$ *last-to-front mapping* $LF[\cdot]$
  (Position in $L[\cdot]$ an der Vorgänger steht)



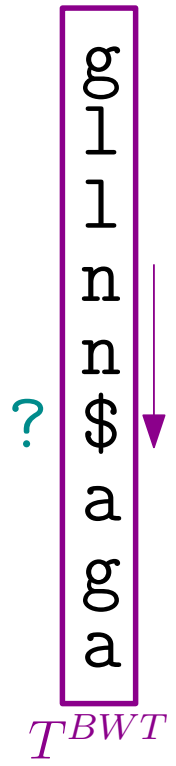
T^{BWT}

Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$								?	$

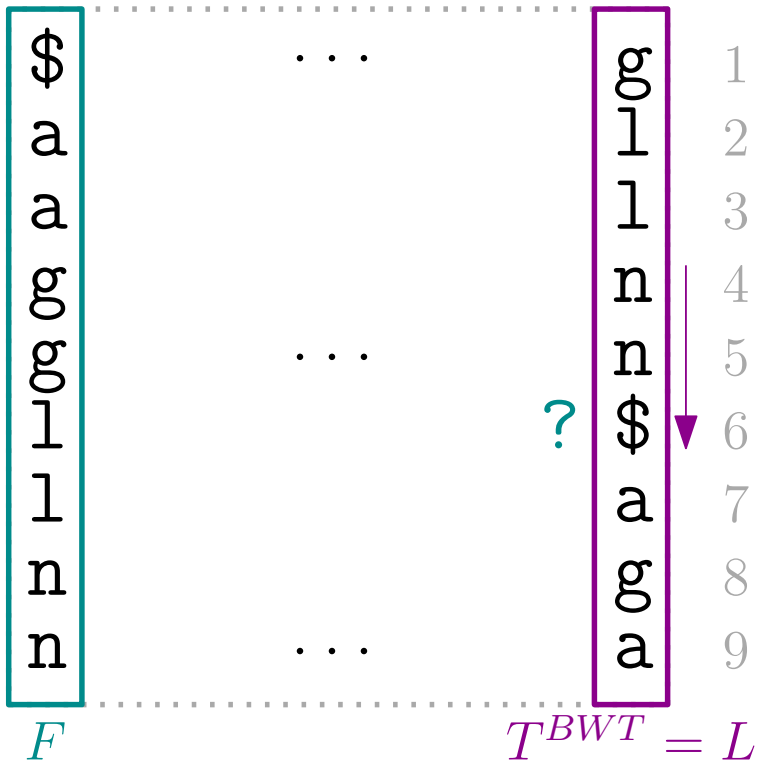
- Wie lautet das Vorgängerzeichen?
  $\rightarrow$ *last-to-front mapping* $LF[\cdot]$
  (Position in $L[\cdot]$ an der Vorgänger steht)



Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$								?	$



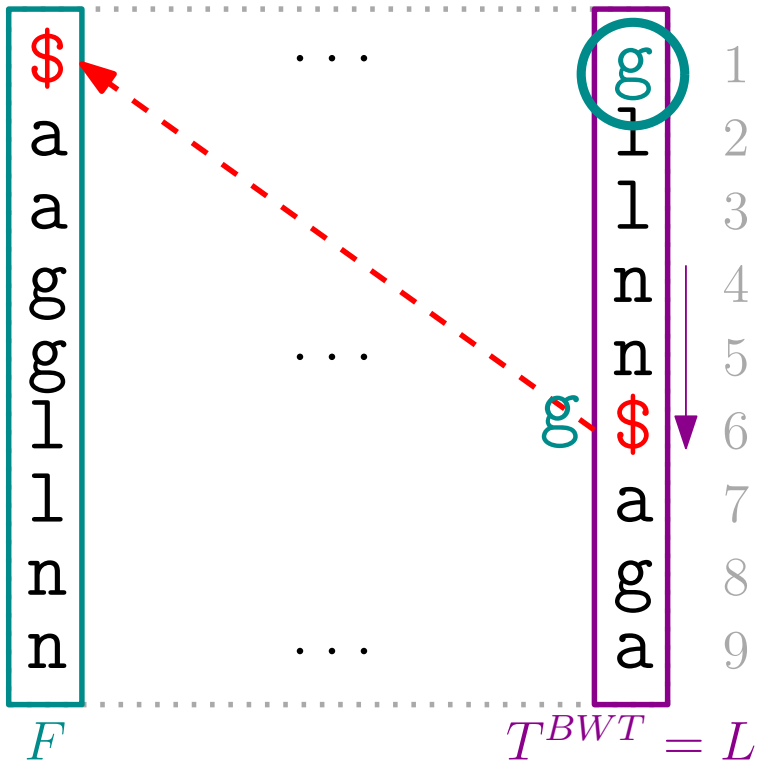
- Wie lautet das Vorgängerzeichen?
  → *last-to-front mapping* $LF[\cdot]$
  (Position in $L[\cdot]$ an der Vorgänger steht)

- $LF[6] = ?$

Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$								?	$



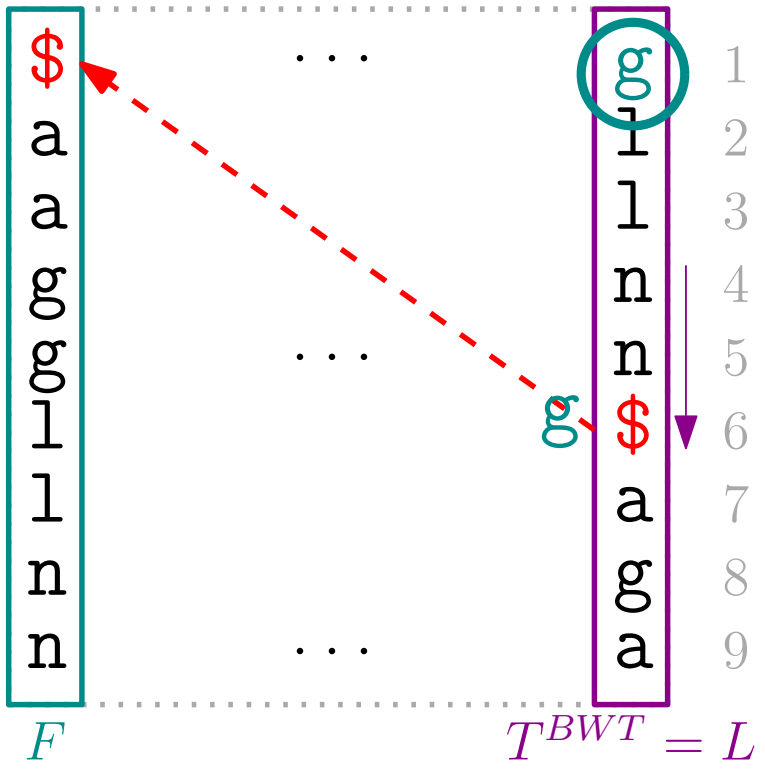
- Wie lautet das Vorgängerzeichen?
  $\rightarrow$ *last-to-front mapping* $LF[\cdot]$
  (Position in $L[\cdot]$ an der Vorgänger steht)

- $LF[6] = 1$
  ($LF[6]$ ist die Position in $F[\cdot]$ an der $L[6]$ steht)

Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

$$\begin{array}{rccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ T^{BWT} = & \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \\ T = & & & & & & & & ? & \mathtt{\$} \end{array}$$



- Wie lautet das Vorgängerzeichen?
  $\rightarrow$ *last-to-front mapping* $LF[\cdot]$
  (Position in $L[\cdot]$ an der Vorgänger steht)

- $LF[i] = j \Leftrightarrow T^{(SA[j])} = (T^{(SA[i])})^{(n)} \Leftrightarrow SA[i] = SA[j] - 1 \pmod n$
  ($LF[i]$ ist die Position in $F[\cdot]$ an der $L[i]$ steht)

Burrows-Wheeler-Transformation

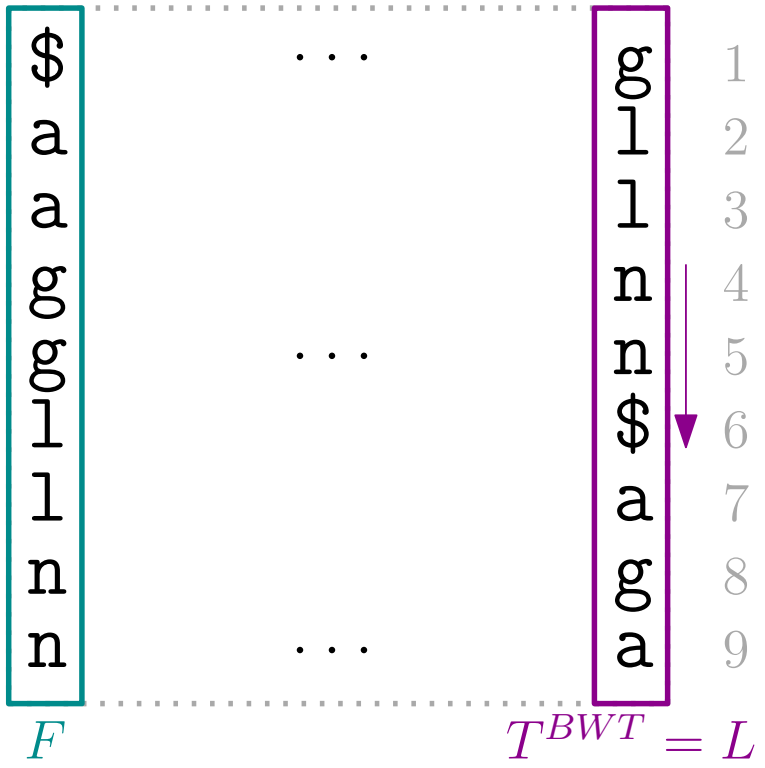
Rücktransformation – weitere Überlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- T^{BWT} hat Struktur, so dass gilt

  → gleiche Zeichen besitzen gleiche Reihenfolge in $F[\cdot]$ und $L[\cdot]$

  → falls $L[i] = L[j]$ mit $i < j$, dann $LF[i] < LF[j]$



- Grund: $\alpha < \beta$ lexikographisch (nach Konstruktion)

Burrows-Wheeler-Transformation

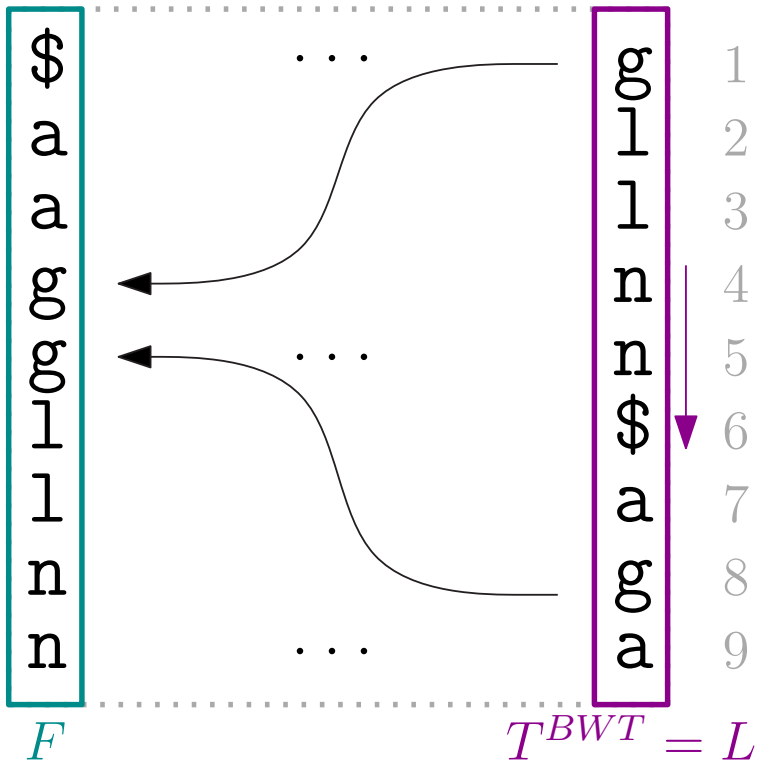
Rücktransformation – weitere Überlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- T^{BWT} hat Struktur, so dass gilt

  $\rightarrow$ gleiche Zeichen besitzen gleiche Reihenfolge in $F[\cdot]$ und $L[\cdot]$

  $\rightarrow$ falls $L[i] = L[j]$ mit $i < j$, dann $LF[i] < LF[j]$

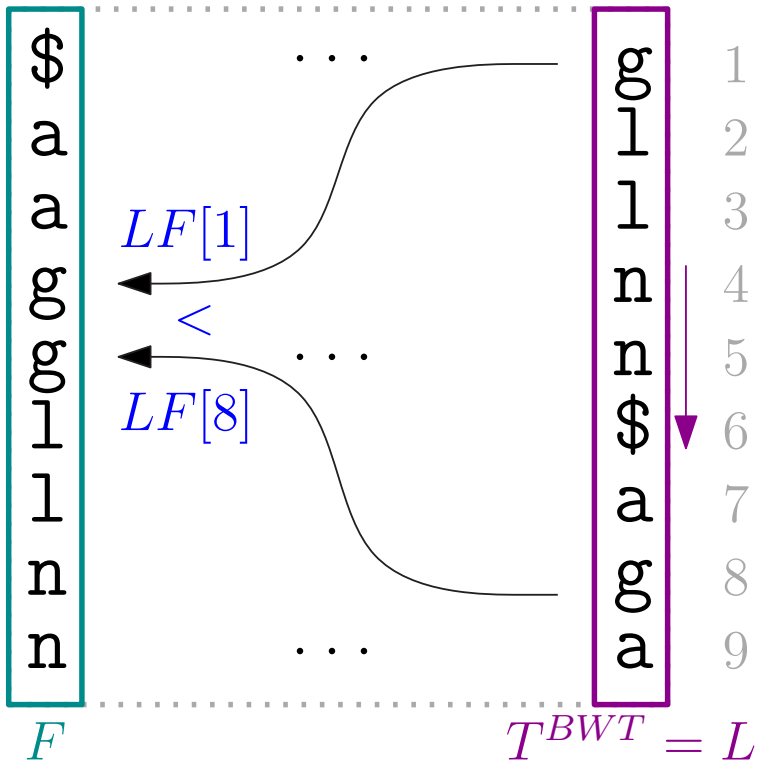


- Grund: $\alpha < \beta$ lexikographisch (nach Konstruktion)

Burrows-Wheeler-Transformation

Rücktransformation – weitere Überlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$



- T^{BWT} hat Struktur, so dass gilt

  → gleiche Zeichen besitzen gleiche Reihenfolge in $F[\cdot]$ und $L[\cdot]$

  → falls $L[i] = L[j]$ mit $i < j$, dann $LF[i] < LF[j]$

- Grund: $\alpha < \beta$ lexikographisch (nach Konstruktion)

Burrows-Wheeler-Transformation

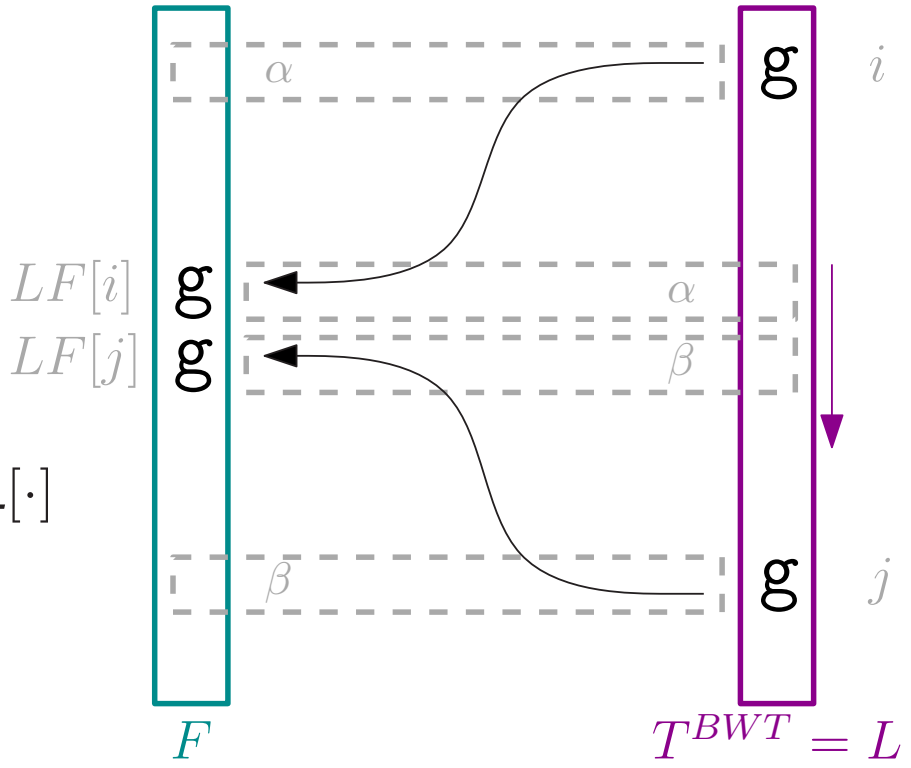
Rücktransformation – weitere Überlegungen

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \end{array}$$

- T^{BWT} hat Struktur, so dass gilt

  $\rightarrow$ gleiche Zeichen besitzen gleiche Reihenfolge in $F[\cdot]$ und $L[\cdot]$

  $\rightarrow$ falls $L[i] = L[j]$ mit $i < j$, dann $LF[i] < LF[j]$



- Grund: $\alpha < \beta$ lexikographisch (nach Konstruktion)

Burrows-Wheeler-Transformation

Berechnung von $LF[\cdot]$

$$T^{BWT} = \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \texttt{g} & \texttt{l} & \texttt{l} & \texttt{n} & \texttt{n} & \texttt{\$} & \texttt{a} & \texttt{g} & \texttt{a} \end{array}$$

- $LF[\cdot]$ nur aus $T^{BWT}[\cdot]$ berechenbar
- $LF[i] = C(L[i]) + occ[i]$
  - $C(a)$ $= \#_{\text{Zeichen}}$ kleiner als a
  - $occ[i] = \#_{\text{Zeichen}}$ gleich $L[i]$ in $L[1..i]$

  ($LF[i]$ ist Position in $F[\cdot]$, an der $L[i]$ steht)

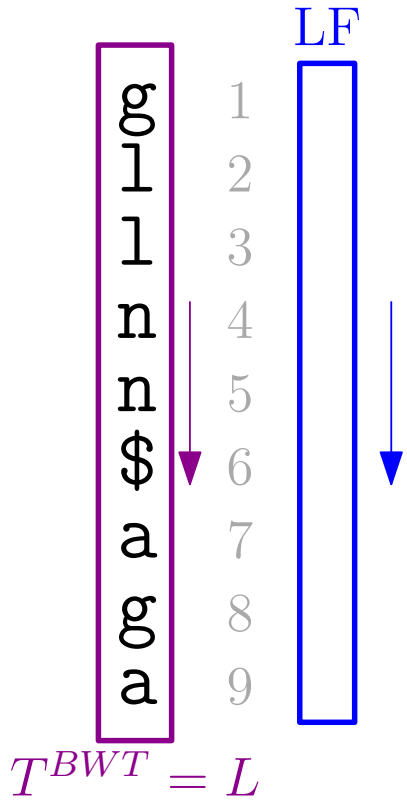
Burrows-Wheeler-Transformation

Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a

- $LF[\cdot]$ nur aus $T^{BWT}[\cdot]$ berechenbar
- $LF[i] = C(L[i]) + occ[i]$
  - $C(a)$ $= \#_{\text{Zeichen}}$ kleiner als a
  - $occ[i] = \#_{\text{Zeichen}}$ gleich $L[i]$ in $L[1..i]$

($LF[i]$ ist Position in $F[\cdot]$, an der $L[i]$ steht)



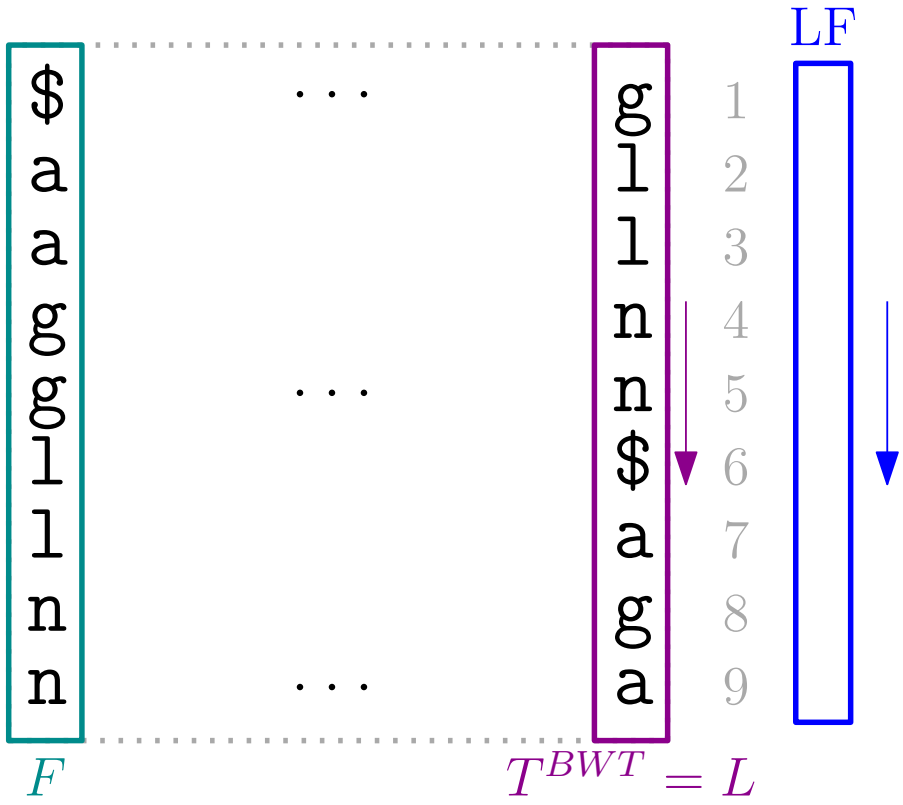
Burrows-Wheeler-Transformation

Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a

- $LF[\cdot]$ nur aus $T^{BWT}[\cdot]$ berechenbar
- $LF[i] = C(L[i]) + occ[i]$
  - $C(a)$ $= \#_{\text{Zeichen}}$ kleiner als a
  - $occ[i] = \#_{\text{Zeichen}}$ gleich $L[i]$ in $L[1..i]$

  ($LF[i]$ ist Position in $F[\cdot]$, an der $L[i]$ steht)



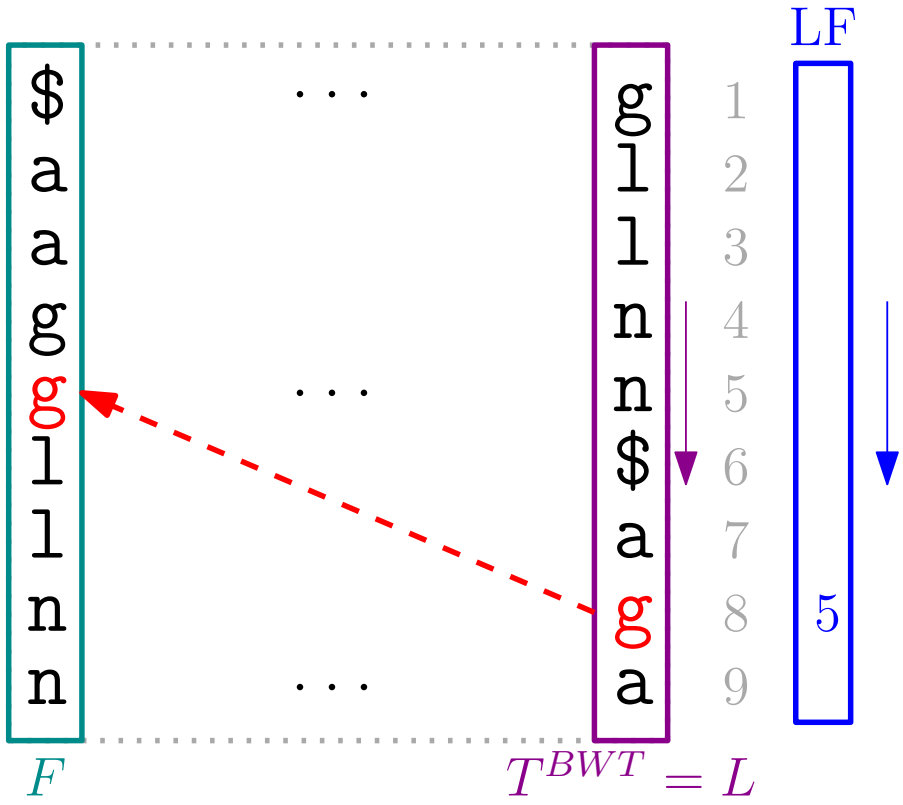
Burrows-Wheeler-Transformation

Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a

- $LF[\cdot]$ nur aus $T^{BWT}[\cdot]$ berechenbar
- $LF[i] = C(L[i]) + occ[i]$
  - $C(a)$ $= \#_{\text{Zeichen}}$ kleiner als a
  - $occ[i] = \#_{\text{Zeichen}}$ gleich $L[i]$ in $L[1..i]$

  ($LF[i]$ ist Position in $F[\cdot]$, an der $L[i]$ steht)



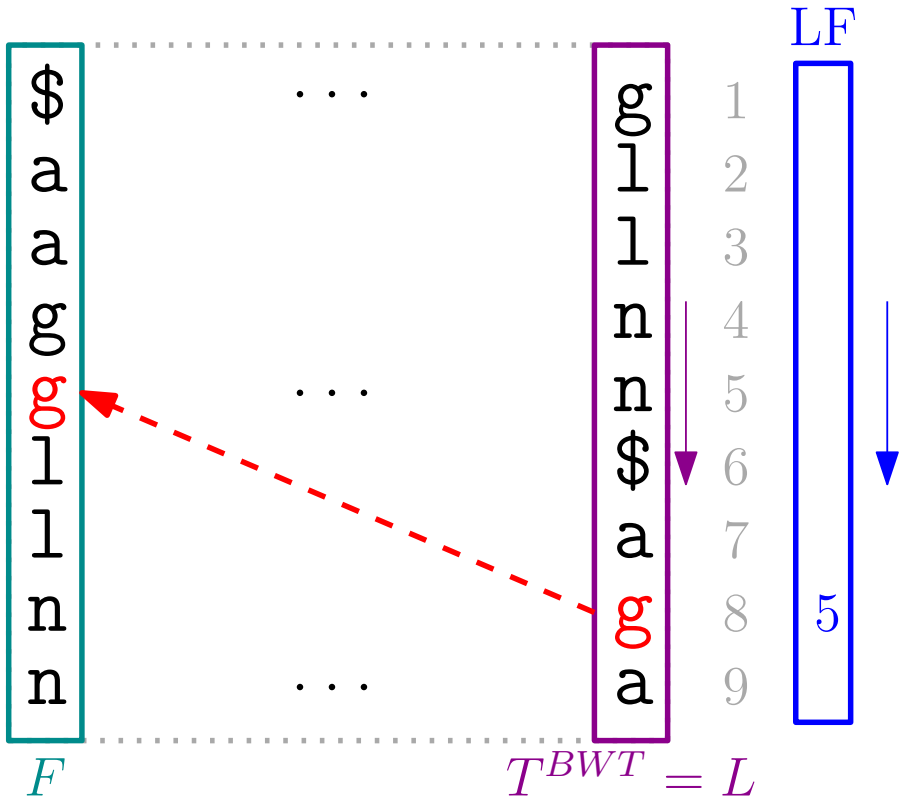
Burrows-Wheeler-Transformation

Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a

- $LF[\cdot]$ nur aus $T^{BWT}[\cdot]$ berechenbar
- $LF[i] = C(L[i]) + occ[i]$
  - $C(a)$ $= \#_{\text{Zeichen}}$ kleiner als a
  - $occ[i] = \#_{\text{Zeichen}}$ gleich $L[i]$ in $L[1..i]$

  ($LF[i]$ ist Position in $F[\cdot]$, an der $L[i]$ steht)



- $C(\cdot)$, $occ[\cdot]$ in $\mathcal{O}(n)$ berechenbar $\rightarrow$ $LF[\cdot]$ auch in $\mathcal{O}(n)$ berechenbar

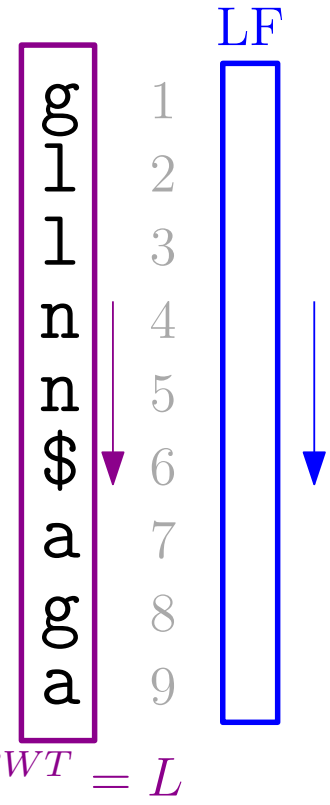
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

$$\begin{array}{rccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ T^{BWT} = & \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \\ occ = & & & & & & & & & \end{array}$$

$$\begin{array}{rccccc} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{l} & \mathtt{n} \\ h = & 0 & 0 & 0 & 0 & 0 \end{array}$$

(zählt Zeichen)



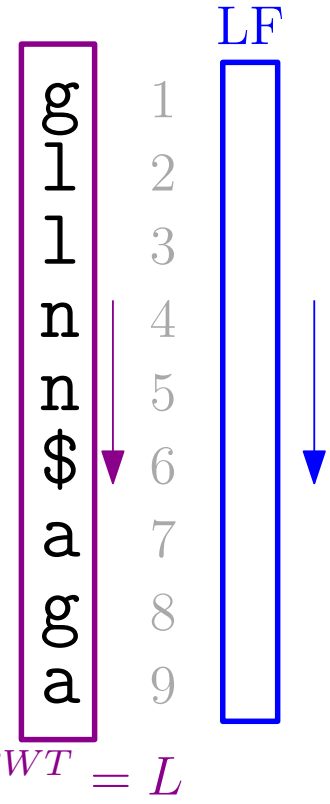
- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$									

	$	a	g	l	n
$h =$ (zählt Zeichen)	0	0	0	0	0



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

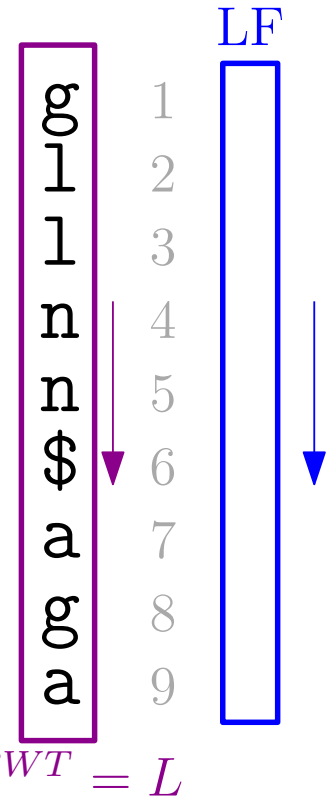
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$									

	$	a	g	l	n
$h =$	0	0	1	0	0

(zählt Zeichen)



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

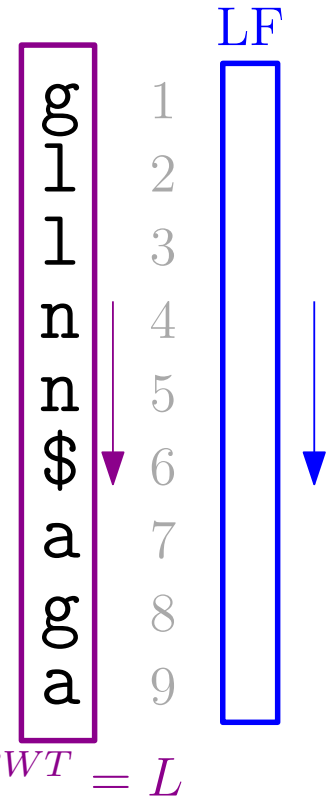
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1								

	$	a	g	l	n
$h =$	0	0	1	0	0

(zählt Zeichen)



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

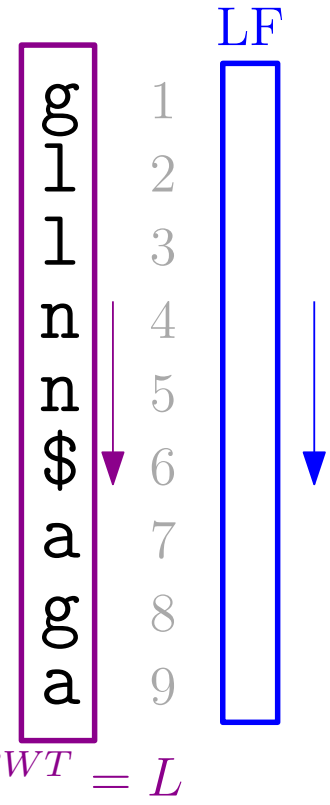
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1	1							

	$	a	g	l	n
$h =$	0	0	1	1	0

(zählt Zeichen)



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

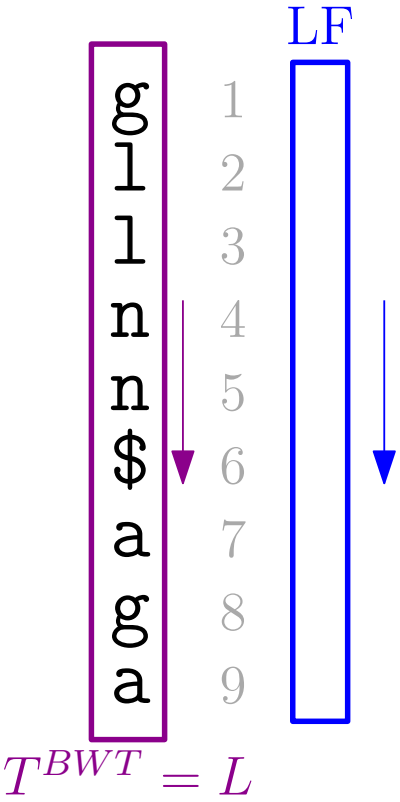
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1	1	2						

	$	a	g	l	n
$h =$	0	0	1	2	0

(zählt Zeichen)



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

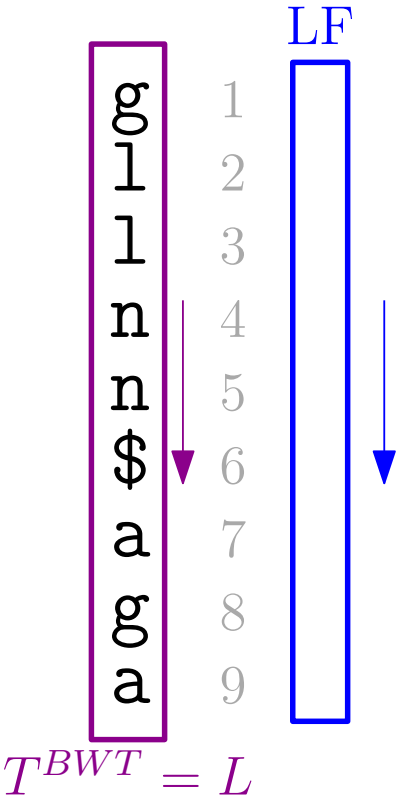
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1	1	2	...					

	$	a	g	l	n
$h =$	0	0	1	2	0

(zählt Zeichen)



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

Burrows-Wheeler-Transformation

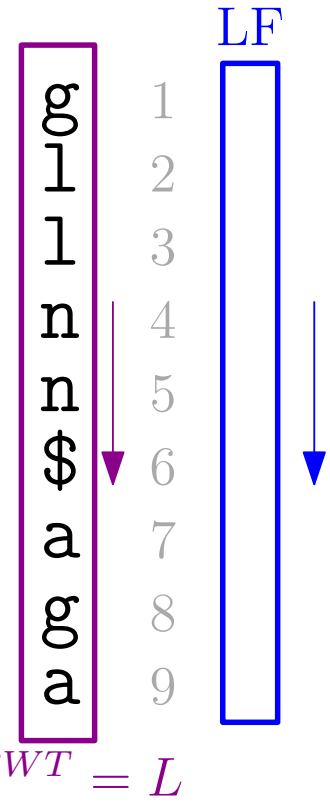
Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1	1	2	1	2	1	1	2	2

	$	a	g	l	n
$h =$	1	2	2	2	2

(zählt Zeichen)

- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$



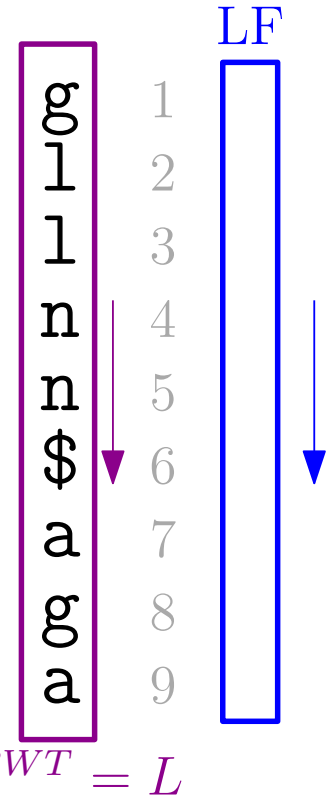
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1	1	2	1	2	1	1	2	2

	$	a	g	l	n
$h =$	1	2	2	2	2
$C =$	0	1	3	5	7

(Präfixsumme von h)



- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$

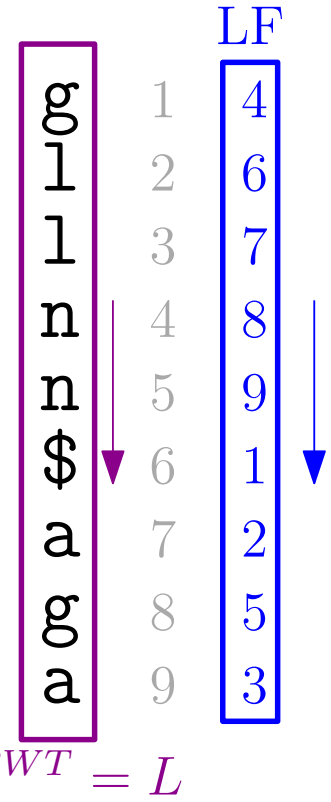
Burrows-Wheeler-Transformation

Ablauf der Berechnung von $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$occ =$	1	1	2	1	2	1	1	2	2

	$	a	g	l	n
$h =$	1	2	2	2	2
$C =$	0	1	3	5	7

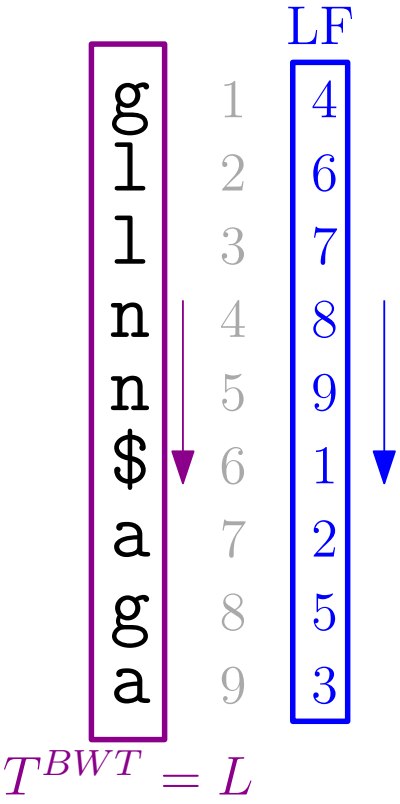
- initialisiere *occ* und *h*
- laufe durch $T^{BWT} = L$ $(i = 1..n)$
  - $h(\ L[i]\ ) = h(\ L[i]\ ) + 1$
  - $occ(\ L[i]\ ) = h(\ L[i]\ )$
- bilde exkl. Präfixsumme von $h \rightarrow C$
- $LF[i] = C(L[i]) + occ[i]$



Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

$$\begin{array}{rccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ T^{BWT} = & \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \\ T = & & & & & & & & & \end{array}$$

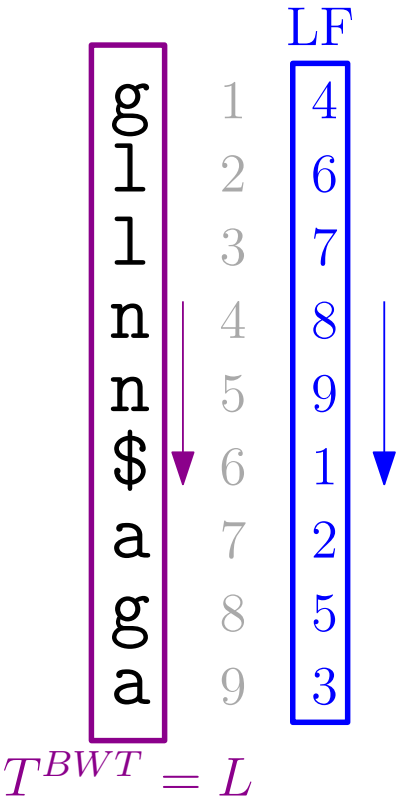


- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...

Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

$$
\begin{array}{rccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
T^{BWT} = & \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{a} \\
T = & & & & & & & & & \mathtt{\$}
\end{array}
$$



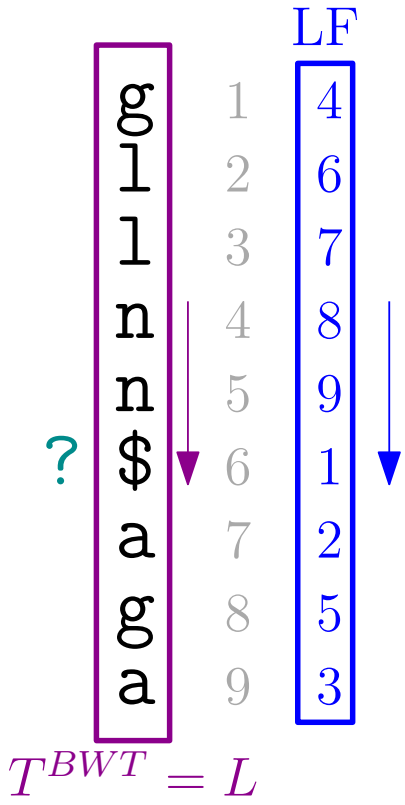
- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...

Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$								?	$

- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...

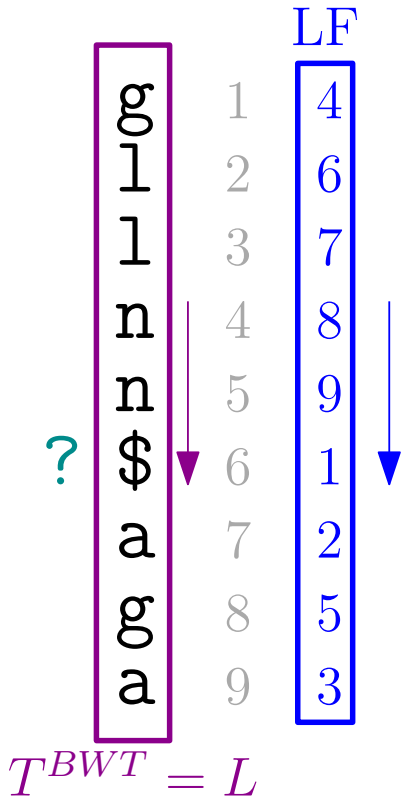


Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$								?	$

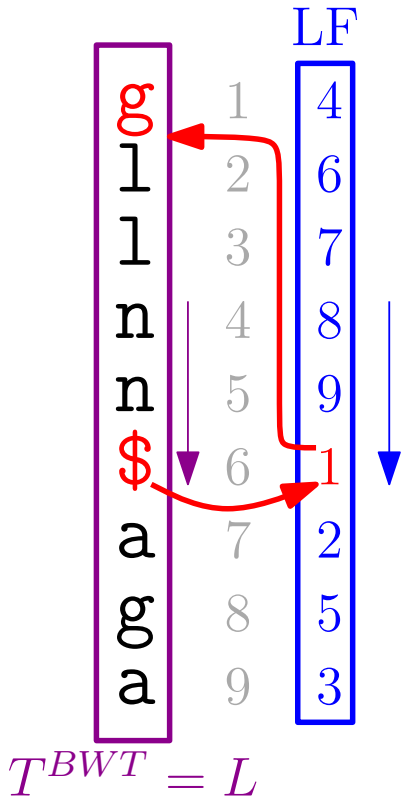
- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...



Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$								g	$

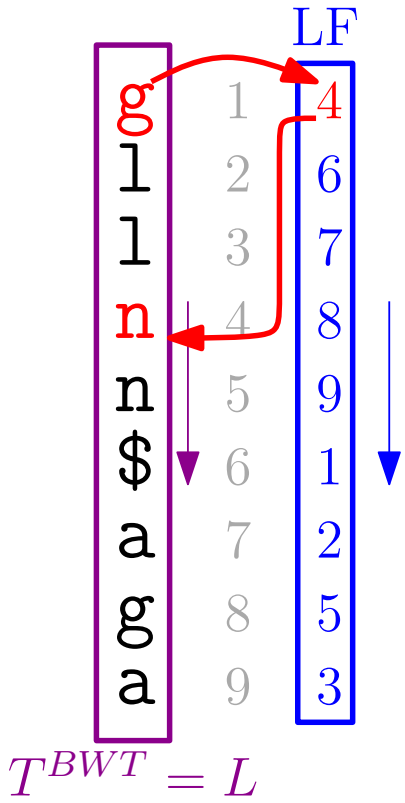


- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...

Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$							n	g	$



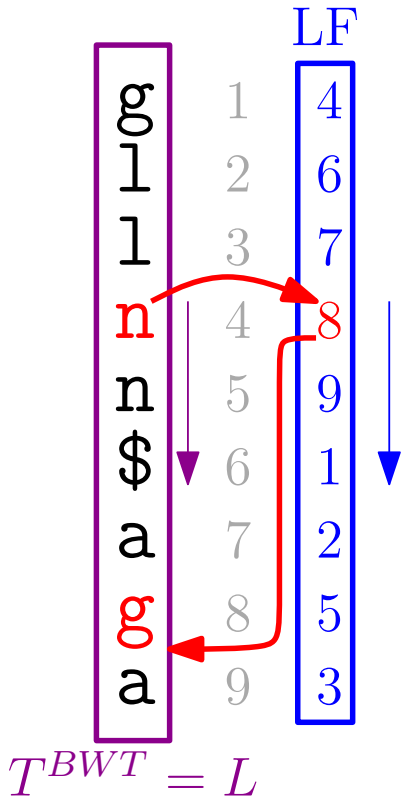
- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...

Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$					...	g	n	g	$

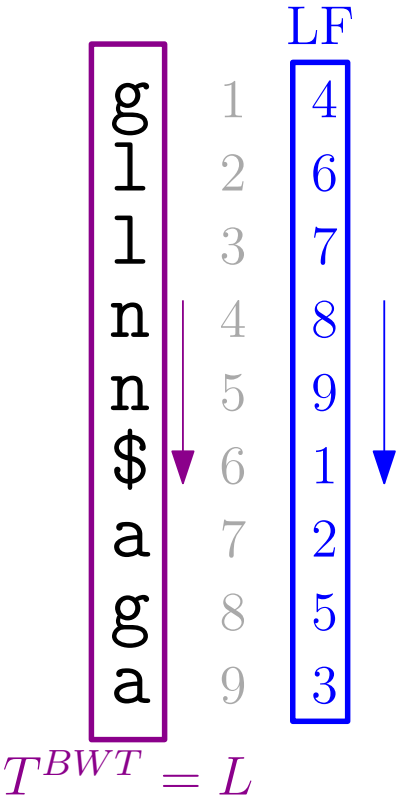
- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...



Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$	l	a	l	a	n	g	n	g	$

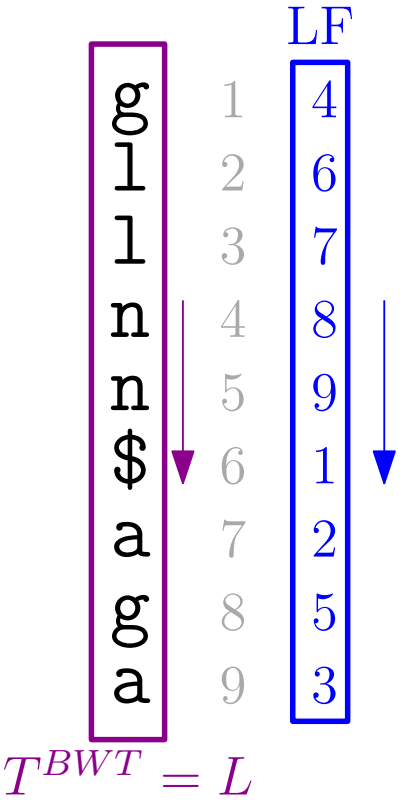


- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...
- Allgemein: $T[n-i] = L[LF(LF(\ldots(LF(1))\ldots))]$ $(i-1)$ LF Anwendungen

Burrows-Wheeler-Transformation

Rücktransformation mit $LF[\cdot]$

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	$	a	g	a
$T =$	l	a	l	a	n	g	n	g	$



- Berechnung von T von rechts nach links
  - Initialisierung: $T[n] = \$ \Rightarrow LF(?) = 1$ (immer!)
  - $L[1] = \mathtt{g} \Rightarrow T[n-1] = g \Rightarrow LF(1) = 4$
  - $L[4] = \mathtt{n} \Rightarrow T[n-2] = n \Rightarrow LF(4) = 8$
  - ...
- Allgemein: $T[n-i] = L[LF(LF(\ldots(LF(1))\ldots))]$ $(i-1)$ LF Anwendungen
- Rücktransformation in $\mathcal{O}(n)$ (ohne Zusatzinformationen)

Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit
  vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)

(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit
  vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)

(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit
  vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)

(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

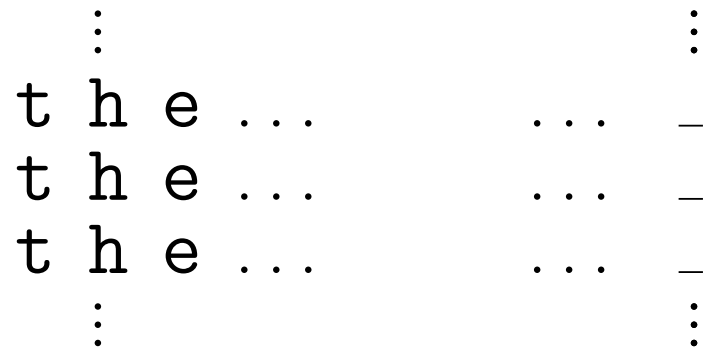
Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit
  vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)



(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

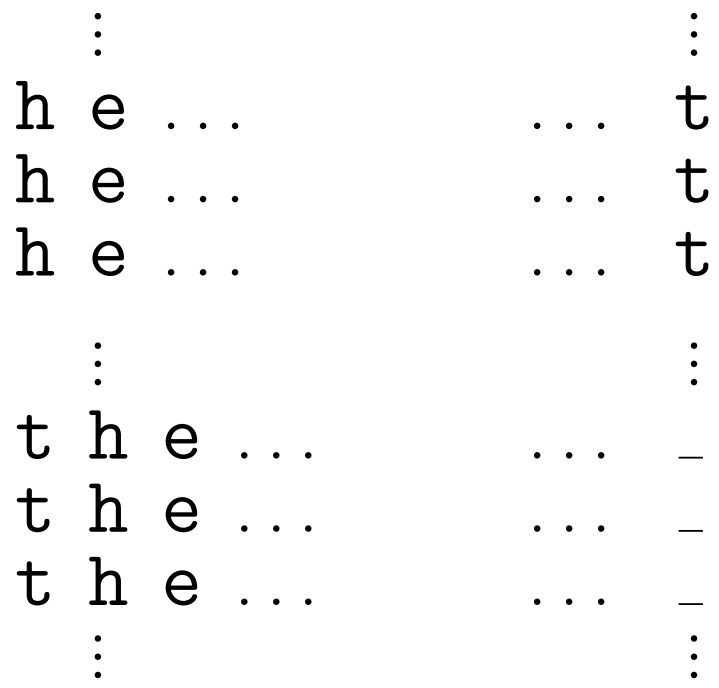
Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)



(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

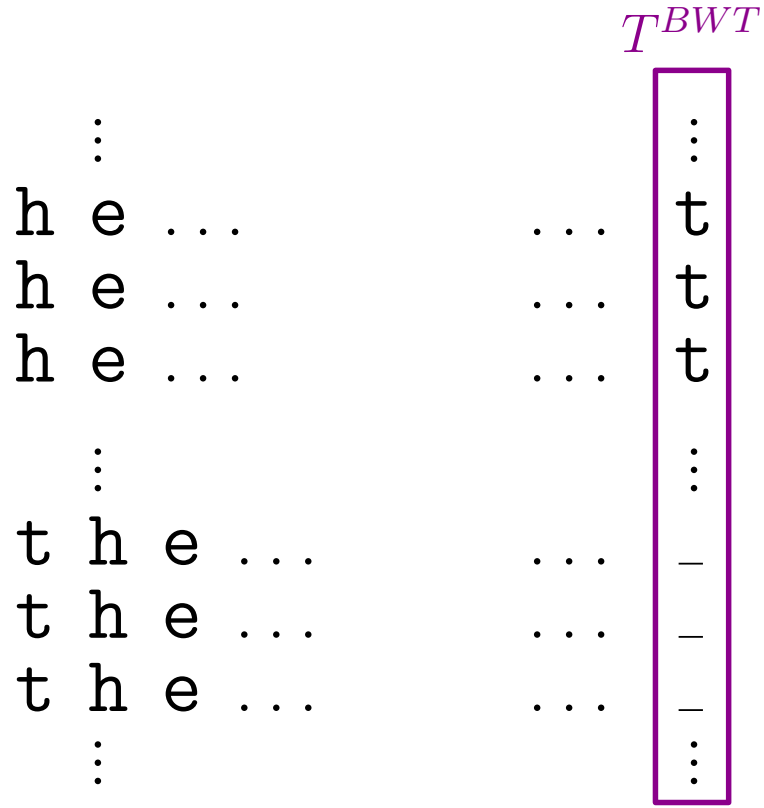
Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)



(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

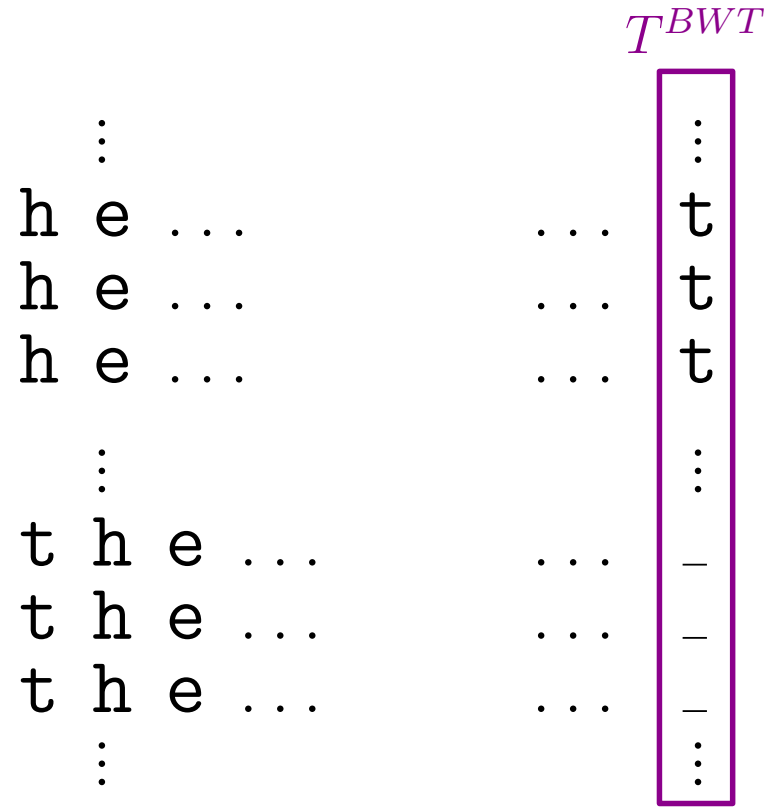
Burrows-Wheeler-Transformation

Was bringt die BWT?

- benötigt gleichen Platz, verwendet gleiche Zeichen wie T
  $\rightarrow$ scheinbar keine Vorteile ?!?

- Permutation einfach umkehrbar
  (benötigt keine Zusatzinformationen, $\mathcal{O}(n)$)
- Zeichen mit ähnlichem Kontext gruppiert
  $\rightarrow$ vereinfacht Komprimierung

- besonders gut auf Texten mit vielen gleichen Substrings
  $\rightarrow$ Beispiel: englischer Text
  (u.a. viele “the”, . . .)



(Außerdem: Vorgänger von Suffixen einfach bestimmbar $\rightarrow$ Indizierung / Suche)

Burrows-Wheeler-Transformation

Kompression

gegeben: Text T
gesucht : komprimierter Text C

bzip2 (1996)

- (Huffmann Kodierung)
- erzeuge *Burrows-Wheeler-Transformation*
- *Move-To-Front* (MTF) Kodierung
- Huffmann Kodierung
  (eigentliche Kompression)

bzip2

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$									

	1	2	3	4	5
$Y =$	$	a	g	l	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$

$R = 3$

	1	2	3	4	5
$Y =$	$	a	g	l	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$	3								

	1	2	3	4	5
	$	a	g	l	n
$Y =$	g	$	a	l	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$	3	4							

	1	2	3	4	5
	$	a	g	l	n
$Y =$	g	$	a	l	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i]$ = Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$	3	4							

	1	2	3	4	5
	$	a	g	l	n
	g	$	a	l	n
$Y =$	l	g	$	a	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i]$ = Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$	3	4	1						

	1	2	3	4	5
	$	a	g	l	n
	g	$	a	l	n
$Y =$	l	g	$	a	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$	3	4	1						

	1	2	3	4	5
	$	a	g	l	n
	g	$	a	l	n
	l	g	$	a	n
$Y =$	l	g	$	a	n

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i]$ = Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

$$
\begin{array}{rccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
T^{BWT} = & \mathtt{g} & \mathtt{l} & \mathtt{l} & \mathtt{n} & \mathtt{n} & \mathtt{a} & \mathtt{g} & \mathtt{a} & \mathtt{\$} \\
R = & 3 & 4 & 1 & \ldots & & & & &
\end{array}
$$

$$
\begin{array}{rccccc}
 & 1 & 2 & 3 & 4 & 5 \\
 & \mathtt{\$} & \mathtt{a} & \mathtt{g} & \mathtt{l} & \mathtt{n} \\
 & \mathtt{g} & \mathtt{\$} & \mathtt{a} & \mathtt{l} & \mathtt{n} \\
 & \mathtt{l} & \mathtt{g} & \mathtt{\$} & \mathtt{a} & \mathtt{n} \\
Y = & \mathtt{l} & \mathtt{g} & \mathtt{\$} & \mathtt{a} & \mathtt{n}
\end{array}
$$

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9
$T^{BWT} =$	g	l	l	n	n	a	g	a	$
$R =$	3	4	1	5	1	5	4	2	5

	1	2	3	4	5
	$	a	g	l	n
	g	$	a	l	n
	l	g	$	a	n
	l	g	$	a	n
	n	l	g	$	a
	n	l	g	$	a
	a	n	l	g	$
	g	a	n	l	$
	a	g	n	l	$
$Y =$	$	a	g	n	l

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9	10	11	12	13
$T^{BWT} =$	a	a	a	a	b	b	b	b	c	c	c	c	d
$R =$													

	1	2	3	4
$Y =$	a	b	c	d

Burrows-Wheeler-Transformation

Kompression: *Move-To-Front* (MTF) Kodierung

- nutzt lokale Redundanz
- erzeugt kleine Zahlen für gleiche Zeichen, die nahe beieinander sind

Ablauf

- initialisiere Y mit Alphabet von T^{BWT}
- durchlaufe T^{BWT} $(i = 1..n)$, generiere $R[1..n]$
  - $R[i] =$ Position von $T^{BWT}[i]$ in Y
  - Schiebe $T^{BWT}[i]$ an den Anfang von Y

	1	2	3	4	5	6	7	8	9	10	11	12	13
$T^{BWT} =$	a	a	a	a	b	b	b	b	c	c	c	c	d
$R =$	1	1	1	1	2	1	1	1	3	1	1	1	4

	1	2	3	4
	a	b	c	d
	⋮			
	b	a	c	d
	⋮			
	c	b	a	d
	⋮			
$Y =$	d	c	b	a

Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

- präfixfreie Codes variabler Länge
- können greedy konstruiert werden

Ablauf

- erzeuge binären Baum *bottom-up*
  - nimm seltenste 2 Zeichen(-gruppen)
  - erzeuge neuen Knoten, der beide Zeichen(-gruppen) repräsentiert, neue Häufigkeit $\hat{=}$ Summe beider Häufigkeiten
- Beschriftungen der Baumkanten (links:0, rechts:1) ergeben Zeichenkodierungen

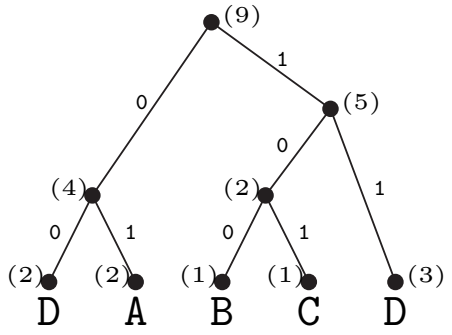
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

- präfixfreie Codes variabler Länge
- können greedy konstruiert werden

Ablauf

- erzeuge binären Baum *bottom-up*
  - nimm seltenste 2 Zeichen(-gruppen)
  - erzeuge neuen Knoten, der beide Zeichen(-gruppen) repräsentiert, neue Häufigkeit $\widehat{=}$ Summe beider Häufigkeiten
- Beschriftungen der Baumkanten (links:0, rechts:1) ergeben Zeichenkodierungen



Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit
1	2
2	1
3	1
4	2
5	3

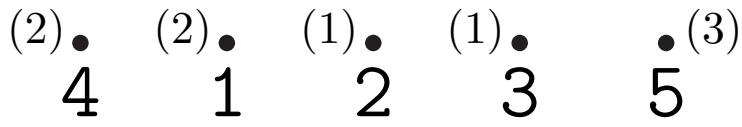
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit
1	2
2	1
3	1
4	2
5	3



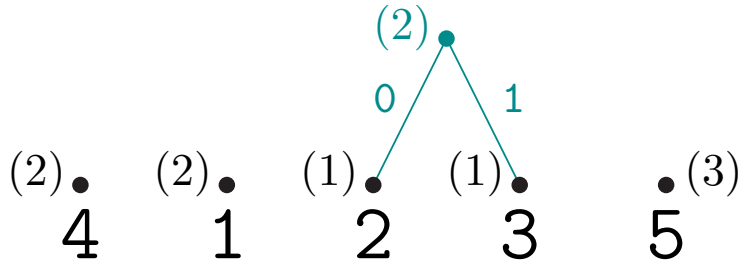
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit
1	2
2	1
3	1
4	2
5	3



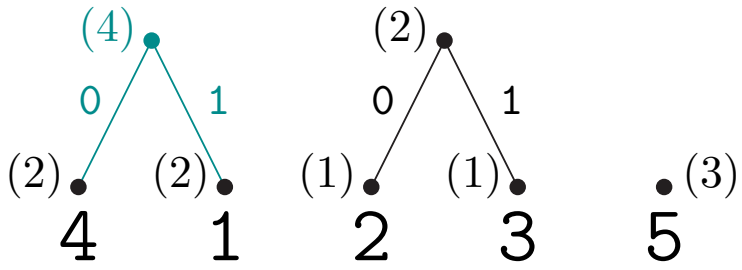
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit
1	2
2	1
3	1
4	2
5	3

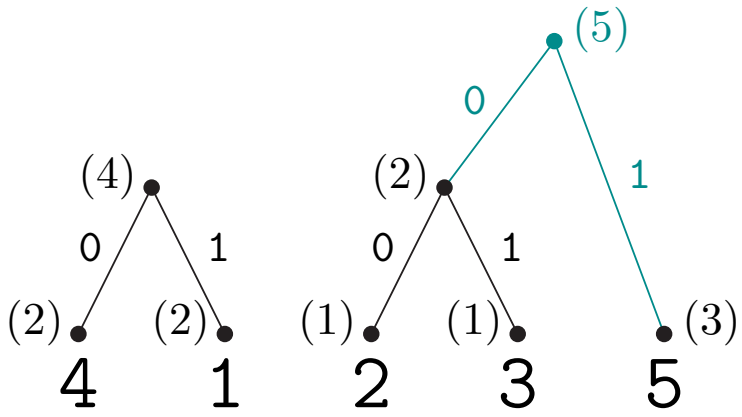


Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $
$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit
1	2
2	1
3	1
4	2
5	3



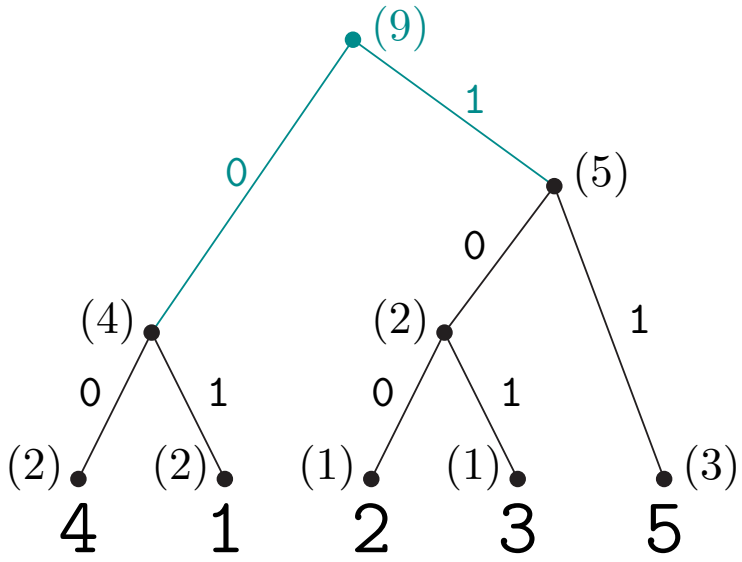
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit
1	2
2	1
3	1
4	2
5	3



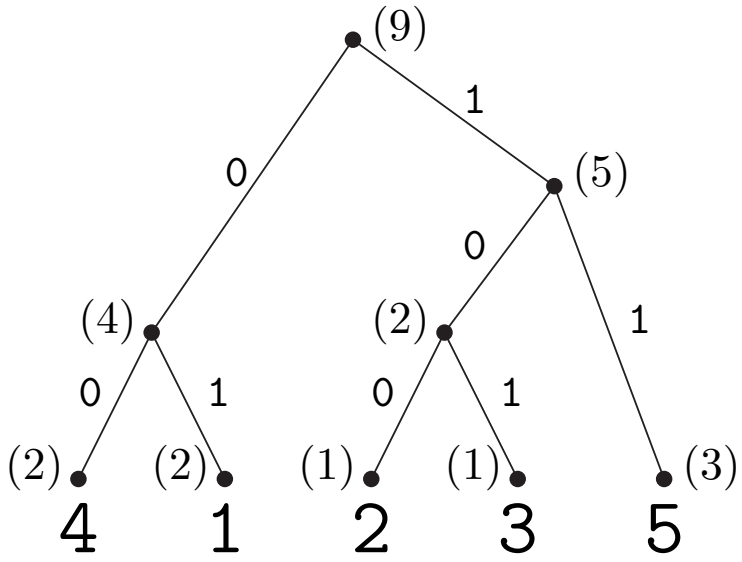
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

Symbol	Häufigkeit	Code
1	2	01
2	1	100
3	1	101
4	2	00
5	3	11



Burrows-Wheeler-Transformation

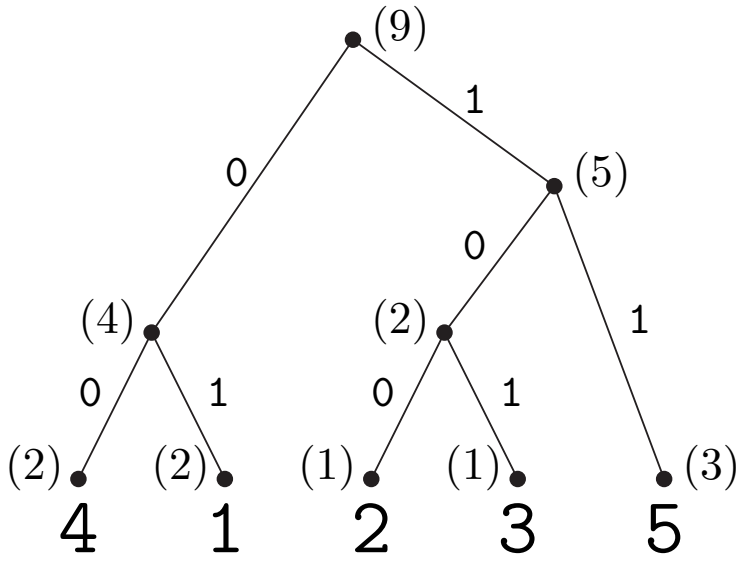
Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

$R =$ 3 4 1 5 1 5 4 2 5

101 00 01 11 01 11 00 100 11 20 Bits

Symbol	Häufigkeit	Code
1	2	01
2	1	100
3	1	101
4	2	00
5	3	11



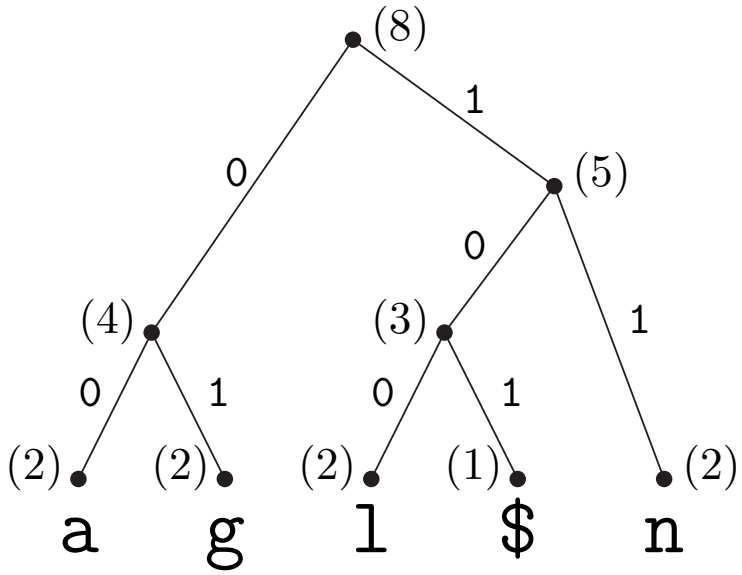
Burrows-Wheeler-Transformation

Kompression: Huffmann Kodierung

$T =$ l a l a n g n g $

100 00 100 00 11 01 11 01 101 21 Bits

Symbol	Häufigkeit	Code
$	1	101
a	2	00
g	2	01
l	2	100
n	2	11



Burrows-Wheeler-Transformation

Zusammenfassung

- erzeugt (sinnvolle) Permutation der Eingabe
  (gruppiert Zeichen mit ähnlichem Kontext nahe beieinander)
- keine Zusatzinformation für Rücktransformation nötig
  (alle Informationen in Struktur der Permutation)
- Hin- und Rücktransformation in $\mathcal{O}(n)$
  (einfache Papier-und-Bleistift-Methode existiert auch)
- Vorverarbeitung (statischer) Texte
  (Komprimierung, Indizierung, Suche)

Suche in der Burrwos-Wheeler-Transformation

Karlsruher Institut für Technologie

Ferragina &Manzini (2000)

- Index basierend auf der Burrows-Wheeler-Transformation (BWT)
- Vergleich des Musters von rechts nach links
- Zeitkomplexität: $\mathcal{O}(m \log \sigma)$

BWT

- $BWT[i] = \mathcal{T}[SA[i] - 1 \bmod n]$
- Unkomprimierte Größe: $n \log \sigma$ Bits
- Komprimierte Größe: $nH_k(\mathcal{T})$ Bits (+Kontextinformation)

Backward Search

i	$SA[i]$	BWT	$\mathcal{T}[SA[i]..n-1]$
0	18	a	\$
1	17	r	a\$
2	10	r	abarbara\$
3	7	d	abrabarbara\$
4	0	\$	abracadabrabarbara\$
5	3	r	acadabrabarbara\$
6	5	c	adabrabarbara\$
7	15	b	ara\$
8	12	b	arbara\$
9	14	r	bara\$
10	11	a	barbara\$
11	8	a	brabarbara\$
12	1	a	bracadabrabarbara\$
13	4	a	cadabrabarbara\$
14	6	a	dabrabarbara\$
15	16	a	ra\$
16	9	b	rabarbara\$
17	2	b	racadabrabarbara\$
18	13	a	rbara\$

- $BWT[i] = \mathcal{T}[SA[i]-1]$, for $SA[i] > 0$
- $BWT[i] = \mathcal{T}[n-1]$, for $SA[i] = 0$
- I.e. $BWT[i]$ is the character preceding suffix $SA[i]$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

Array C contains for each $c \in \Sigma$ the position of the first suffix in SA which starts with c:

$	a	b	c	d	r	r+1
0	1	9	13	14	15	19

- Operation $rank(i, X, BWT)$ returns how often character $X \in \Sigma$ occurs in the prefix $BWT[0..i-1]$.
- Example: search for $\mathcal{P} = bar$.

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = C[r] + rank(sp_0, r, BWT)$
  $ep_1 = C[r] + rank(ep_0 + 1, r, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + rank(0, r, BWT)$
  $ep_1 = 15 + rank(19, r, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + 0$
  $ep_1 = 15 + rank(19, r, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15+0 = 15$
  $ep_1 = 15+4-1 = 18$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = C[a] + rank(sp_1 - 1, a, BWT)$
  $ep_2 = C[a] + rank(ep_1, a, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a, BWT)$
  $ep_2 = 1 + rank(ep_1, a, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a, BWT)$
  $ep_2 = 1 + rank(ep_1, a, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + 6$
  $ep_2 = 1 + rank(19, a, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1+6 = 7$
  $ep_2 = 1+8-1 = 8$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = C[b] + rank(sp_2, b, BWT)$
  $ep_3 = C[b] + rank(ep_2 + 1, b, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b, BWT)$
  $ep_3 = 9 + rank(ep_1, b, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b, BWT)$
  $ep_3 = 9 + rank(ep_1, b, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + 0$
  $ep_3 = 9 + rank(9, b, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9+0 = 9$
  $ep_3 = 9+2-1 = 10$

Backward Search

Summary

- Only C and a data structure R supporting the *rank* operation on *BWT* are required for existence and count queries.
- Space: $\sigma \log n$ bits for C + space for R
- Time: $\mathcal{O}(m \cdot t_{rank})$, where t_{rank} is time for one rank operation. Independent from n?
- Next: How to implement *rank*?

Rank operation

- Constant time and $o(n)$ extra space solution on bitvectors (Jacobson 1989)
- Solution on general sequences: Wavelet Tree (Grossi & Vitter 2003)

Backward Search

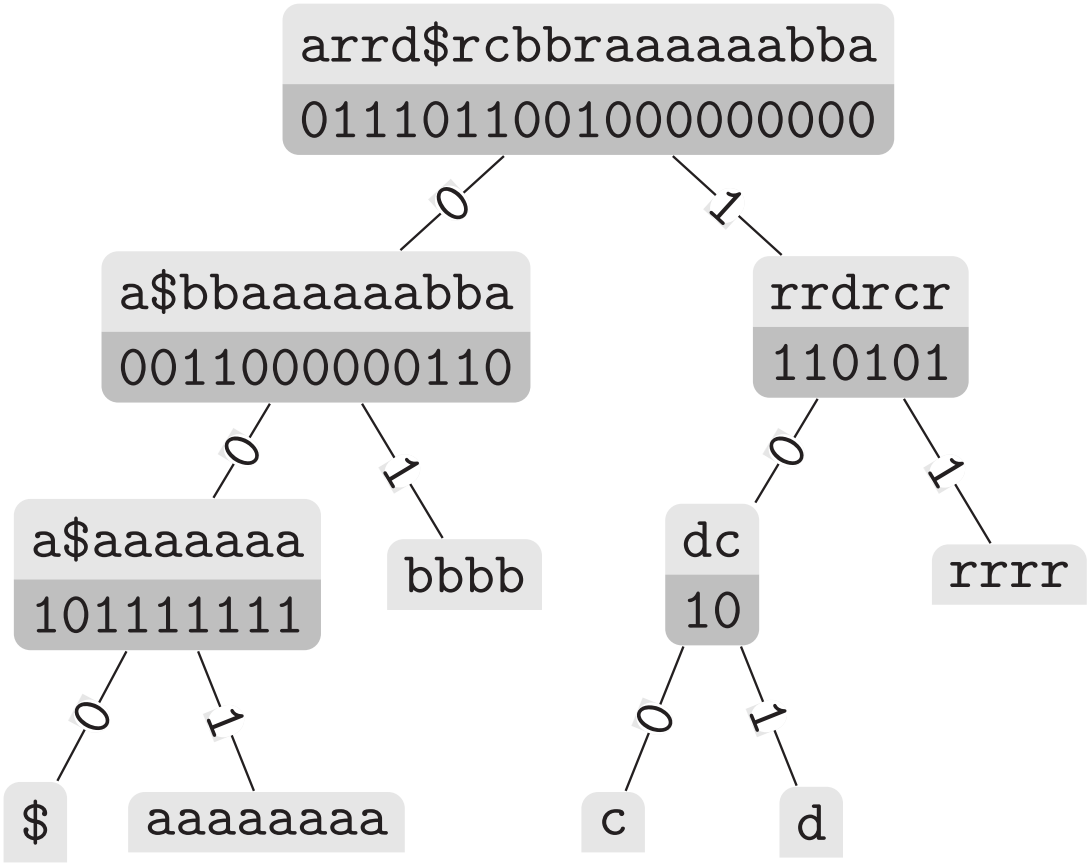
Summary

- Only C and a data structure R supporting the *rank* operation on *BWT* are required for existence and count queries.
- Space: $\sigma \log n$ bits for C + space for R
- Time: $\mathcal{O}(m \cdot t_{rank})$, where t_{rank} is time for one rank operation. Independent from n? If t_{rank} is independent from n
- Next: How to implement *rank*?

Rank operation

- Constant time and $o(n)$ extra space solution on bitvectors (Jacobson 1989)
- Solution on general sequences: Wavelet Tree (Grossi & Vitter 2003)

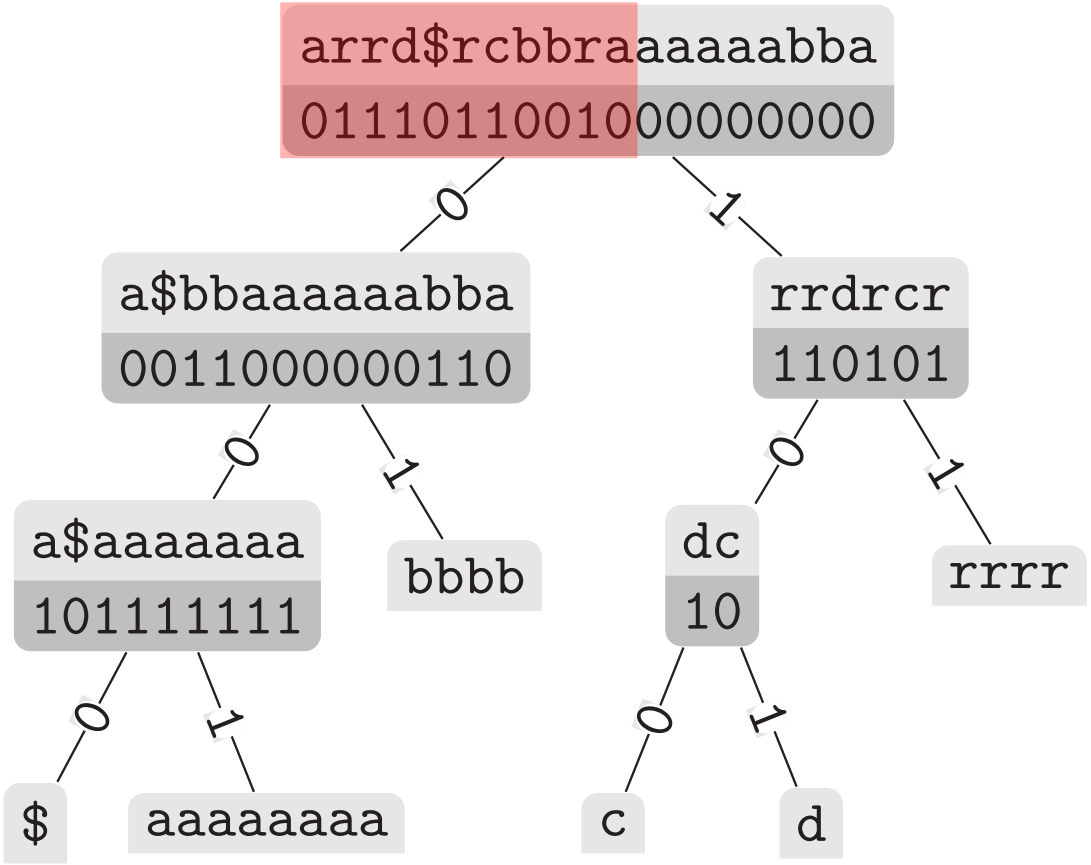
Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

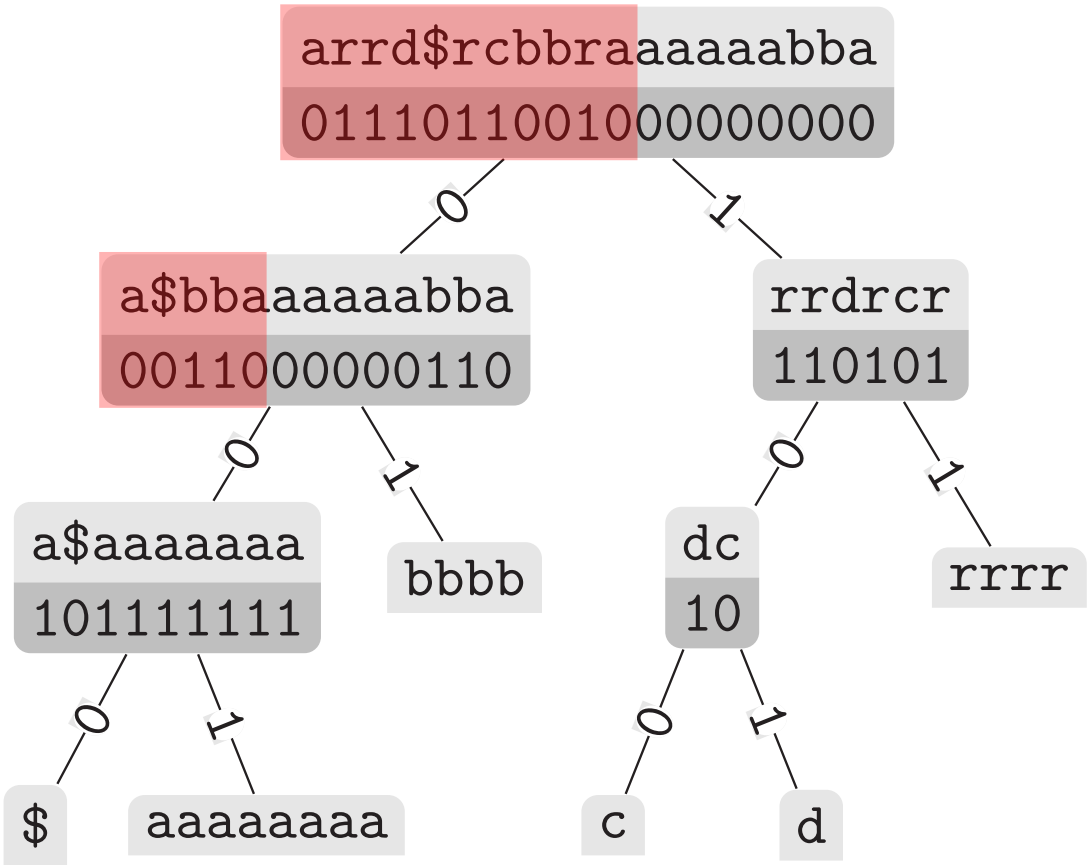
Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

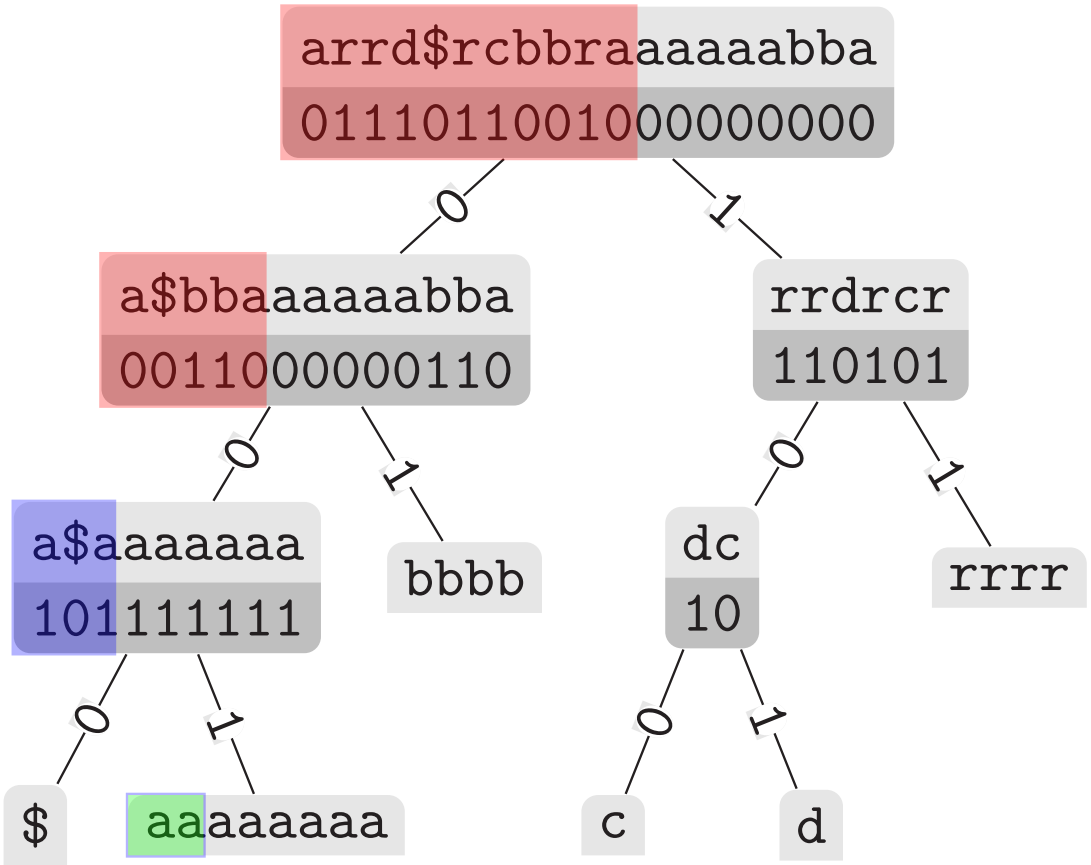
Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

$o(n)$ space / constant query time

- Given bitvector B of length n bits
- Divide B into superblocks of size L
- For each superblock SB_j store $\Sigma_{i=0}^{(j-1)L-1} B[i]$ in $\log n$ bits
- Divide each superblock into blocks of size S
- For each block B_k of superblock j store $\Sigma_{i=(j-1)L}^{(j-1)L+kS-1} B[i]$ in $\log L$ bits

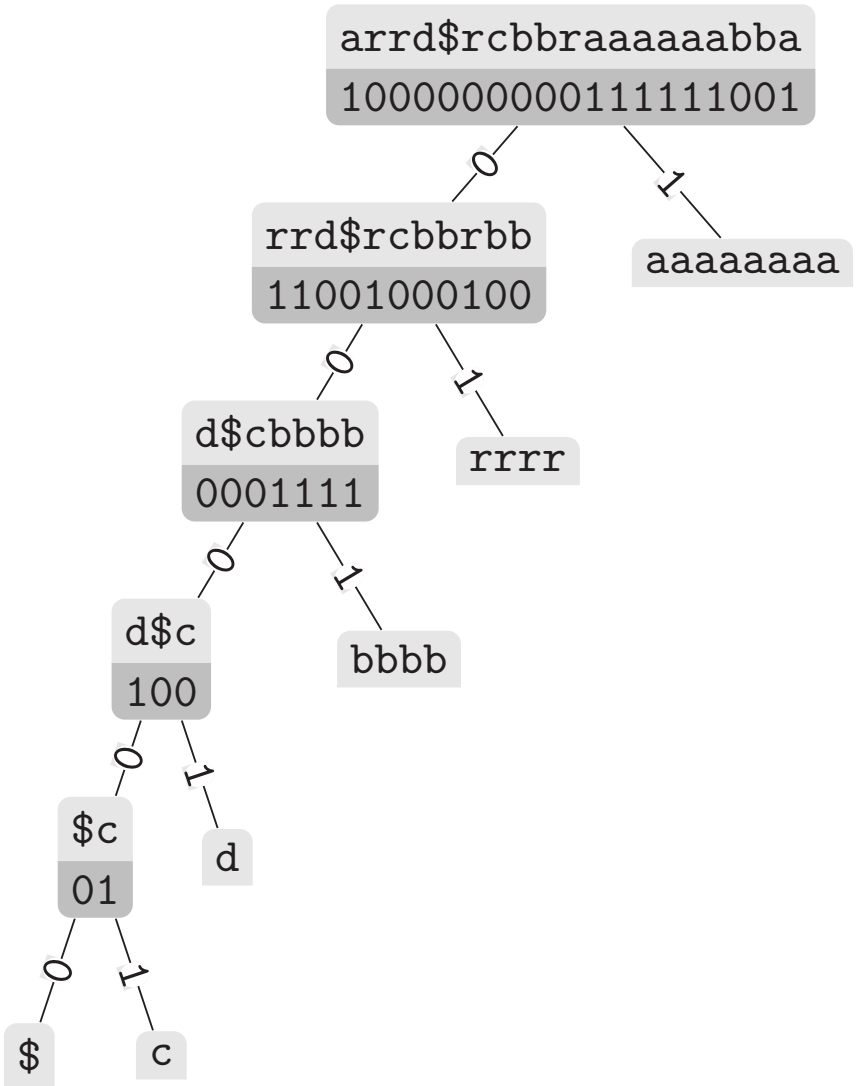
If blocks are small enough, we can pre-compute a table which stores all answers for every block and position (four Russian-Tick).

Solution

- $L = \log^2 n$
- $S = \frac{1}{2} \log n$

Final space: $\mathcal{O}(\frac{n}{\log n} + \frac{n \log\log n}{\log n} + \sqrt{n} \log n \log\log n)$ bits

Huffman-shaped Wavelet Tree



Char c	$codeword(c)$
$	00000
a	1
b	001
c	00001
d	0001
r	01

Avg. depth: $H_0(BWT)$. Total space: $\approx nH_0 + 2\sigma \log n$ bits

Practical Performance of FM-Index

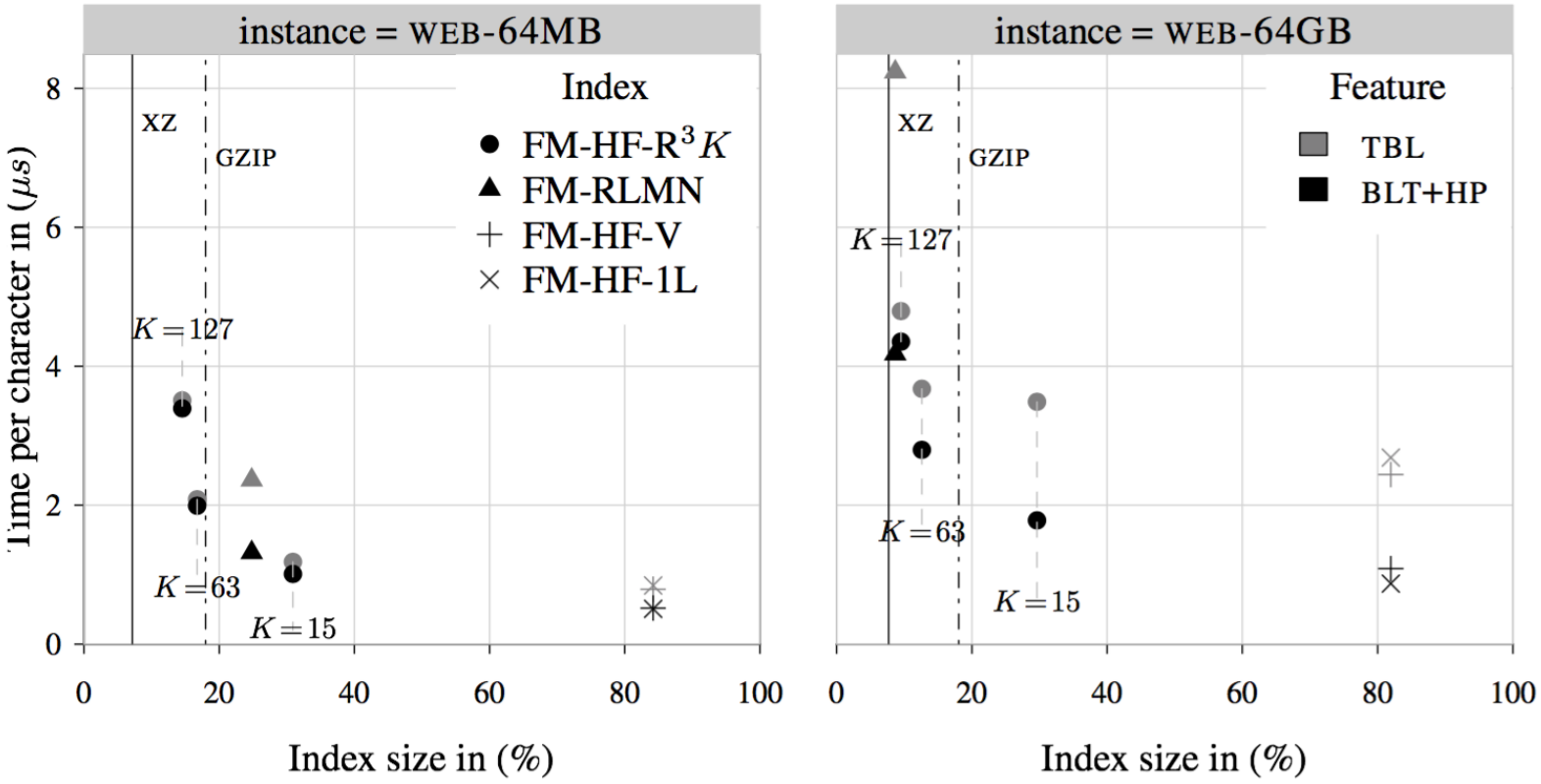


Figure 10. Count time and space of our index implementations on input instances of different size with compression effectiveness baselines using standard compression utilities XZ and GZIP with option `--best`.

Succinct Data Structures

Karlsruher Institut für Technologie

Definition

A *succinct data structure* uses space „close” to the information-theoretical lower bound, but still supports operations time-efficiently.

Let L be the information-theoretical lower bound to represent a class of objects. Then a data structure which still supports time-efficient operations is called

- *implicit*, if it takes $L + O(1)$ bits of space
- *succinct*, if it takes $L + o(L)$ bits of space
- *compact*, if it takes $O(L)$ bits of space

Succinct representation of trees

- First consider binary trees
- Number of n-node binary trees: $C_n = \frac{1}{n+1}\binom{2n}{n}$
- We need $\log C_n = 2n + o(n)$ bits (using Sterling's Approximation)
- Operations: $\mathit{parent}(v)$, $\mathit{leftchild}(v)$, $\mathit{rightchild}(v)$

Succinct representation of trees

- First consider binary trees
- Number of n-node binary trees: $C_n = \frac{1}{n+1}\binom{2n}{n}$
- We need $\log C_n = 2n + o(n)$ bits (using Sterling's Approximation)
- Operations: $\mathit{parent}(v)$, $\mathit{leftchild}(v)$, $\mathit{rightchild}(v)$

Succinct representation of trees

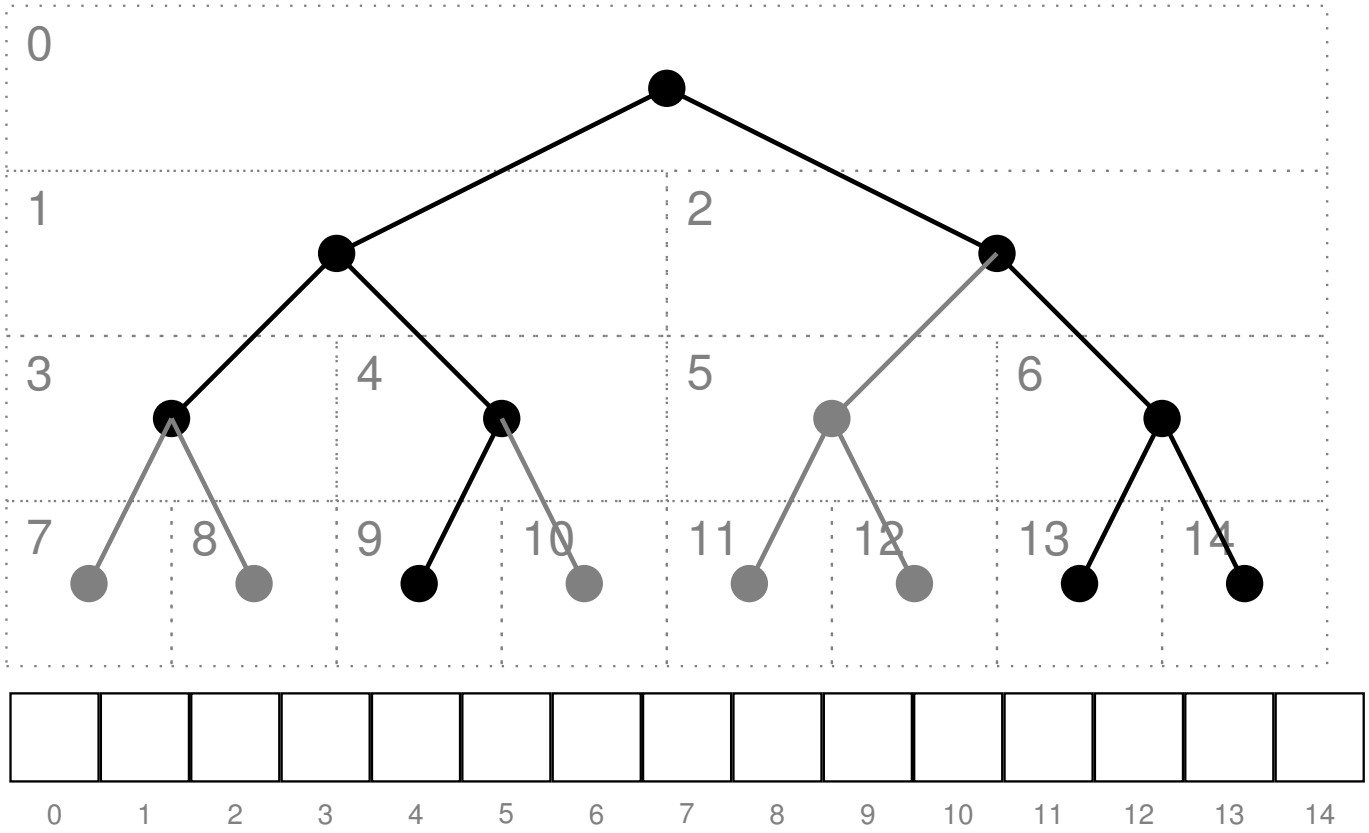
- First consider binary trees
- Number of n-node binary trees: $C_n = \frac{1}{n+1}\binom{2n}{n}$
- We need $\log C_n = 2n + o(n)$ bits (using Sterling's Approximation)
- Operations: $\mathit{parent}(v)$, $\mathit{leftchild}(v)$, $\mathit{rightchild}(v)$

Succinct representation of trees

- First consider binary trees
- Number of n-node binary trees: $C_n = \frac{1}{n+1}\binom{2n}{n}$
- We need $\log C_n = 2n + o(n)$ bits (using Sterling's Approximation)
- Operations: $\mathit{parent}(v)$, $\mathit{leftchild}(v)$, $\mathit{rightchild}(v)$

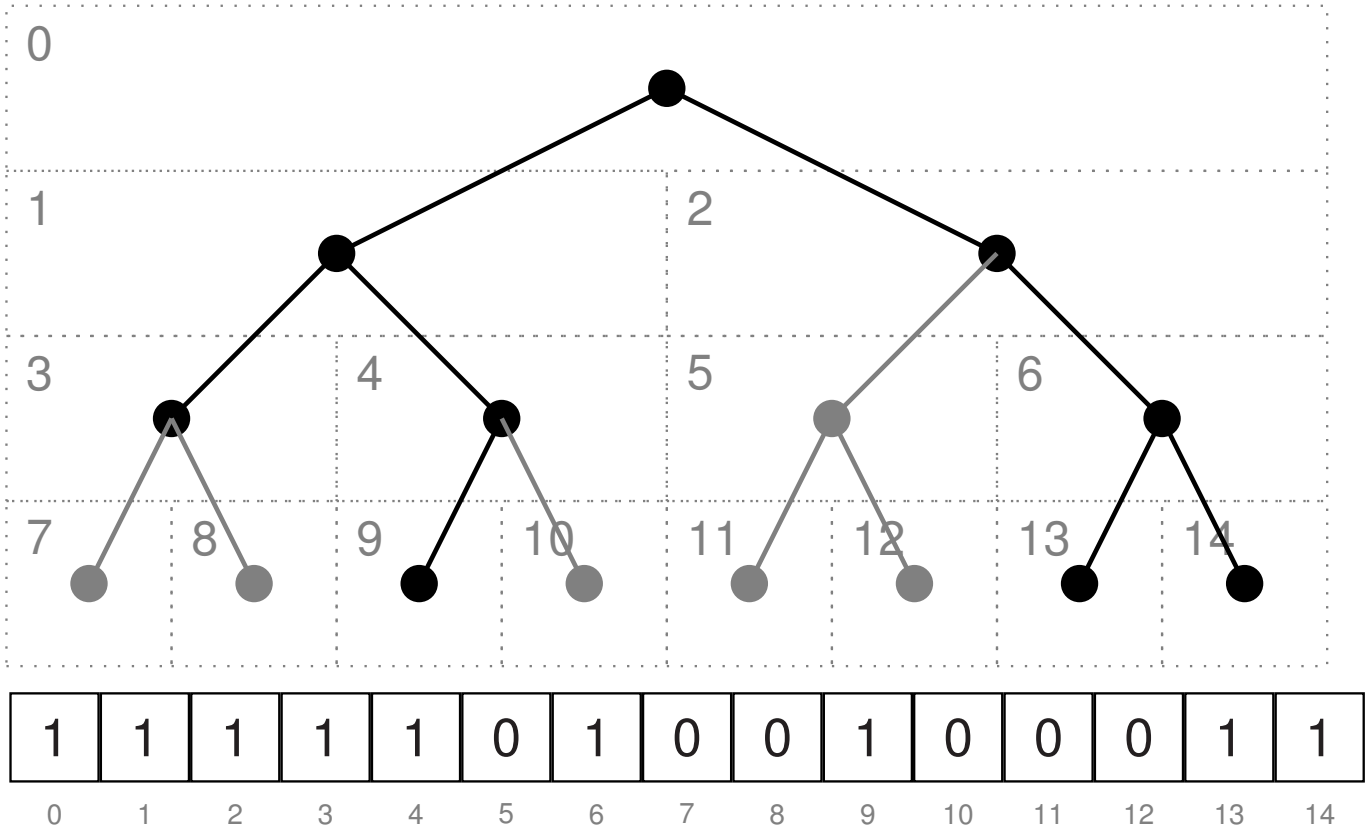
Succinct representation of trees

- First consider binary trees
- Number of n-node binary trees: $C_n = \frac{1}{n+1}\binom{2n}{n}$
- We need $\log C_n = 2n + o(n)$ bits (using Sterling's Approximation)
- Operations: $parent(v)$, $leftchild(v)$, $rightchild(v)$



Succinct representation of trees

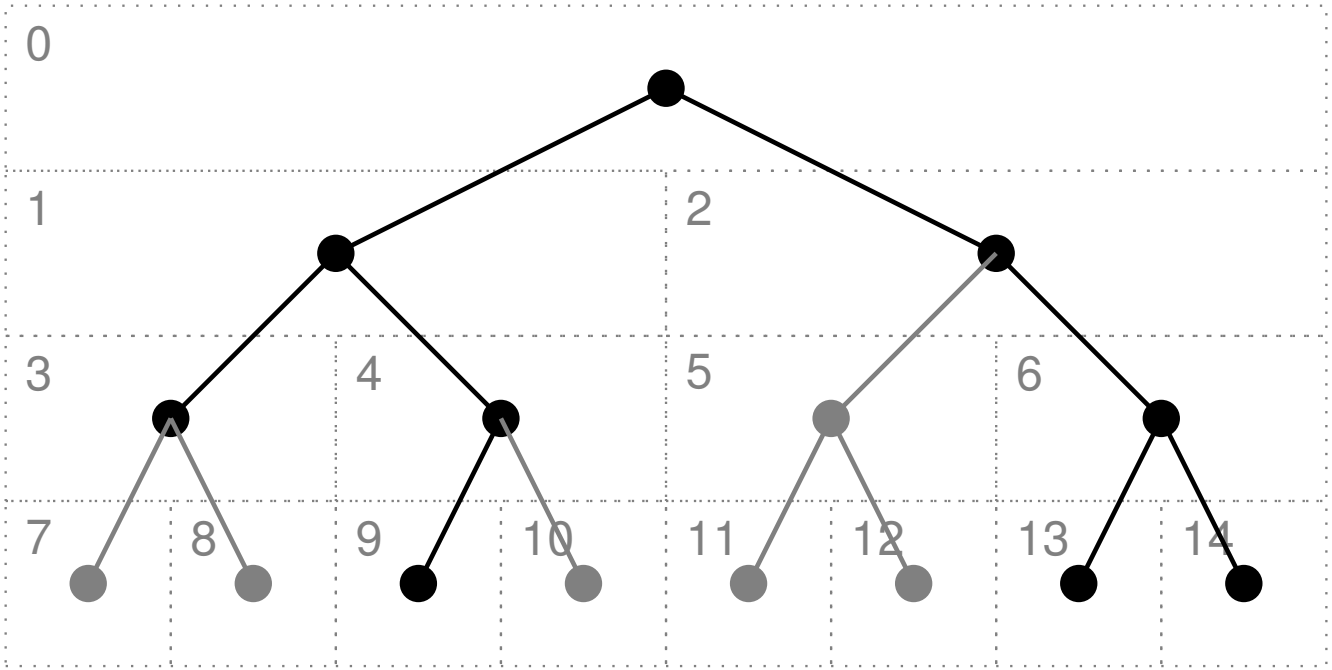
- First consider binary trees
- Number of n-node binary trees: $C_n = \frac{1}{n+1}\binom{2n}{n}$
- We need $\log C_n = 2n + o(n)$ bits (using Sterling's Approximation)
- Operations: $parent(v)$, $leftchild(v)$, $rightchild(v)$



Succinct representation of trees

- In a very balanced binary tree (like in a heap) operations are easy
- Let 0 be the root identifier
  - $\mathit{parent}(v) = \lfloor \frac{v-1}{2} \rfloor$ for $v > 0$
  - $\mathit{leftchild}(v) = 2v + 1$
  - $\mathit{rightchild}(v) = 2v + 2$

Child operation in detail

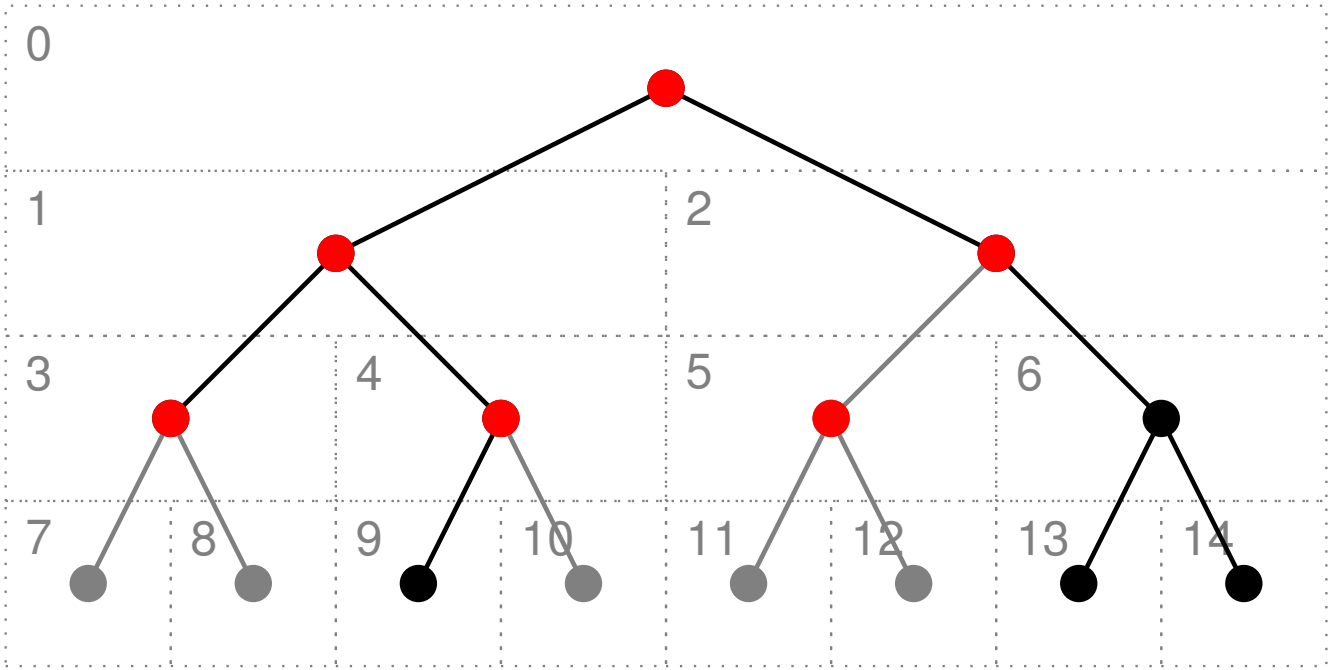


w=leftchild(v)

- Mark $P(v)$
- $P(w) = \{$children of $P(v)\} \cup$
- w is at position $|P(w)| = 2P(v) + 1 = 2v + 1$

- Let $\delta(v)$ be the distance of a node v to the root node
- $P(v) := \{w \mid \delta(w) \leq \delta(v) \wedge w < v\}$
- Note: $|P(v)| = v$

Child operation in detail

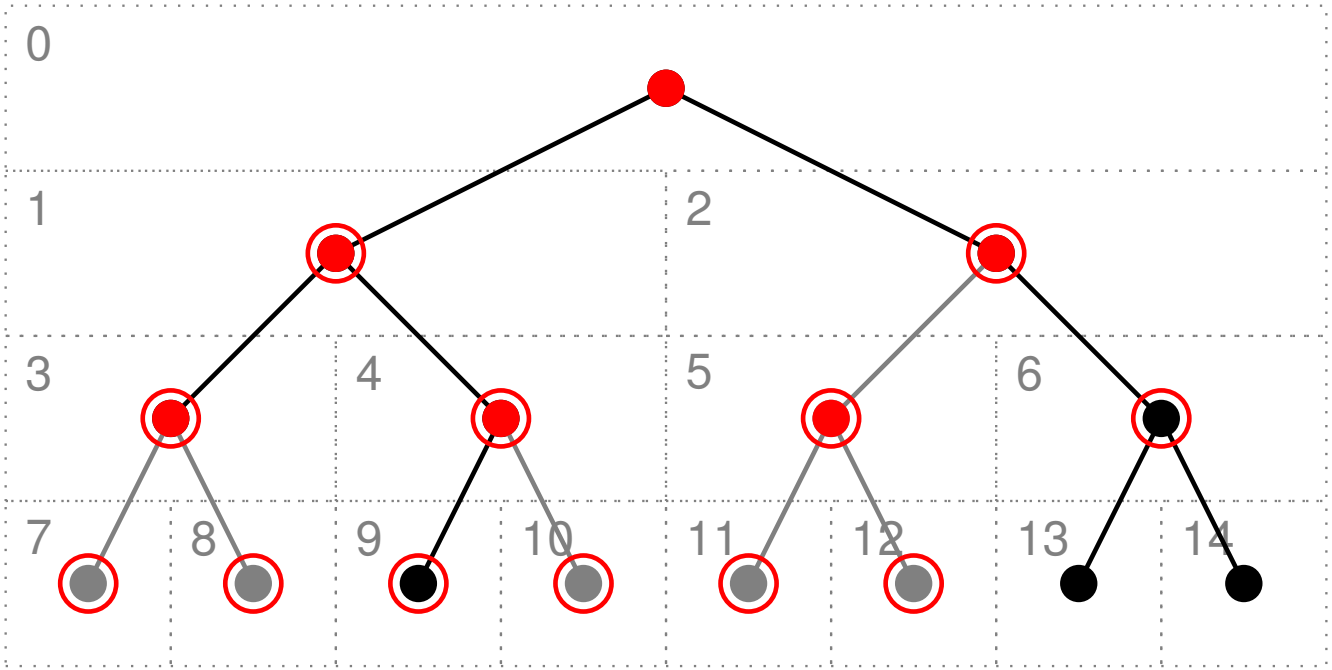


w=leftchild(v)

- Mark $P(v)$
- $P(w) = \{$children of $P(v)\} \cup$
- w is at position $|P(w)| = 2P(v) + 1 = 2v + 1$

- Let $\delta(v)$ be the distance of a node v to the root node
- $P(v) := \{w \mid \delta(w) \leq \delta(v) \wedge w < v\}$
- Note: $|P(v)| = v$

Child operation in detail

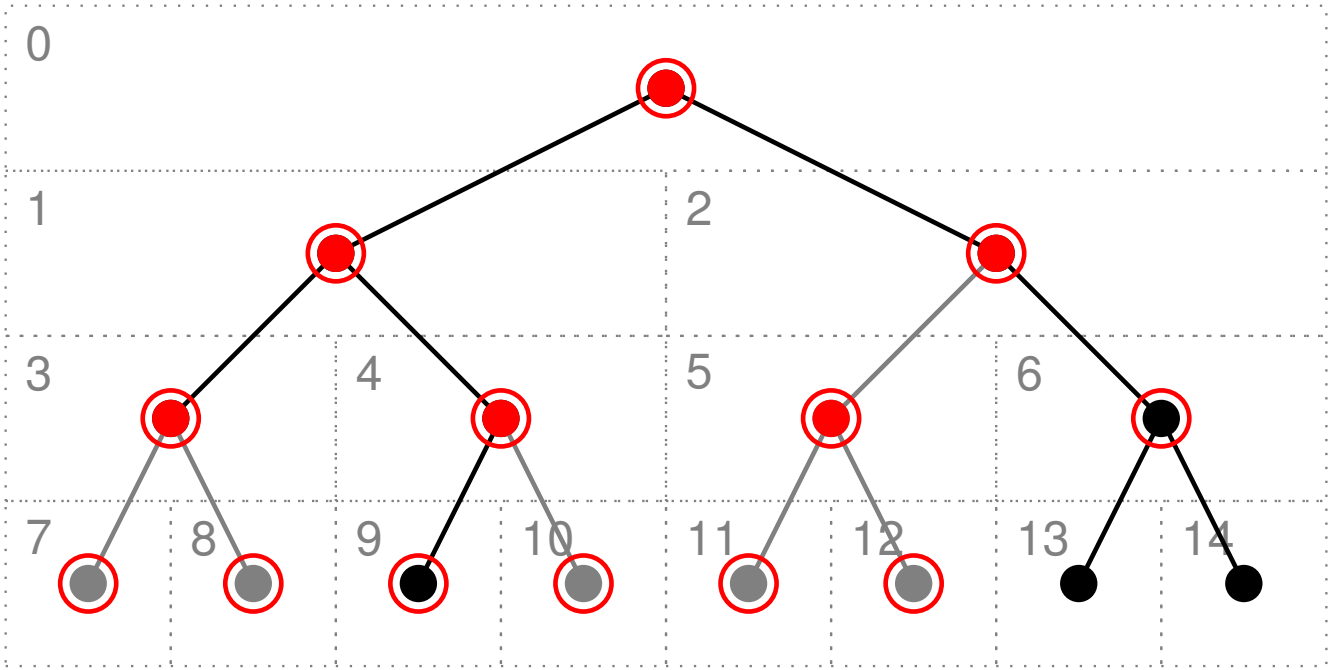


w=leftchild(v)

- Mark $P(v)$
- $P(w) = \{$children of $P(v)\} \cup$
- w is at position $|P(w)| = 2P(v) + 1 = 2v + 1$

- Let $\delta(v)$ be the distance of a node v to the root node
- $P(v) := \{w \mid \delta(w) \leq \delta(v) \wedge w < v\}$
- Note: $|P(v)| = v$

Child operation in detail

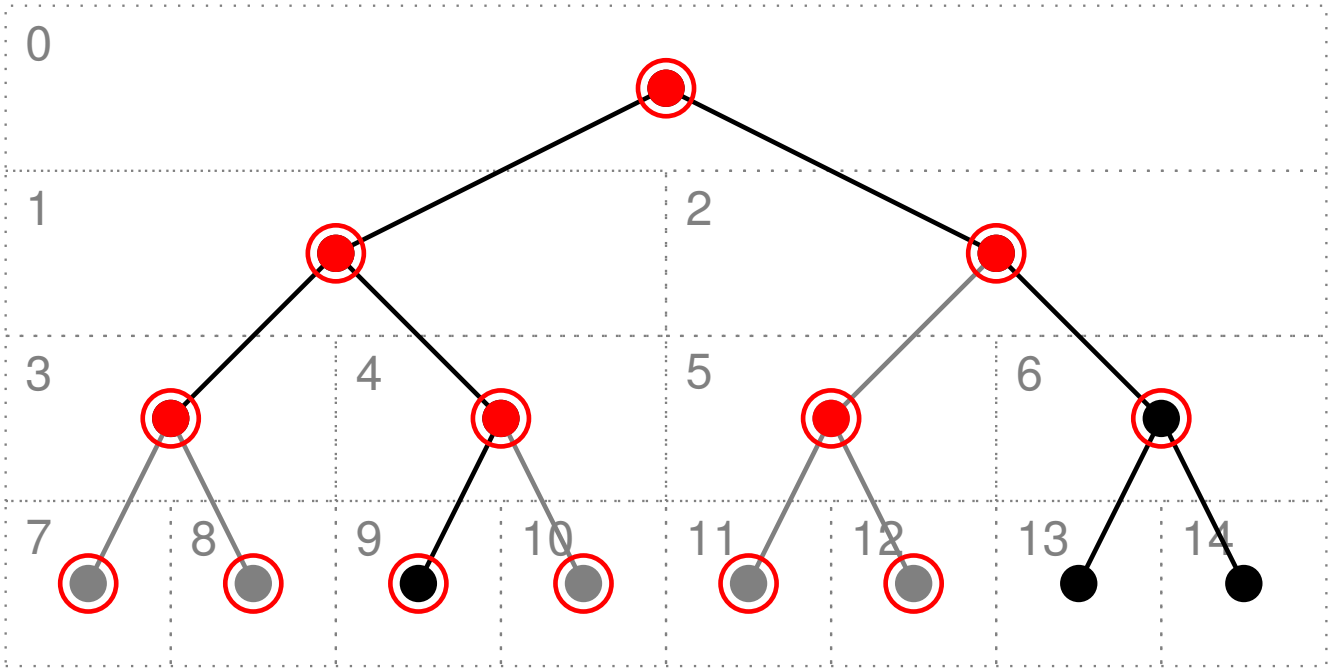


w=leftchild(v)

- Mark $P(v)$
- $P(w) = \{$children of $P(v)\} \cup \{root\}$
- w is at position $|P(w)| = 2P(v) + 1 = 2v + 1$

- Let $\delta(v)$ be the distance of a node v to the root node
- $P(v) := \{w \mid \delta(w) \leq \delta(v) \wedge w < v\}$
- Note: $|P(v)| = v$

Child operation in detail



w=leftchild(v)

- Mark $P(v)$
- $P(w) = \{$children of $P(v)\} \cup \{root\}$
- w is at position $|P(w)| = 2P(v) + 1 = 2v + 1$

- Let $\delta(v)$ be the distance of a node v to the root node
- $P(v) := \{w \mid \delta(w) \leq \delta(v) \wedge w < v\}$
- Note: $|P(v)| = v$

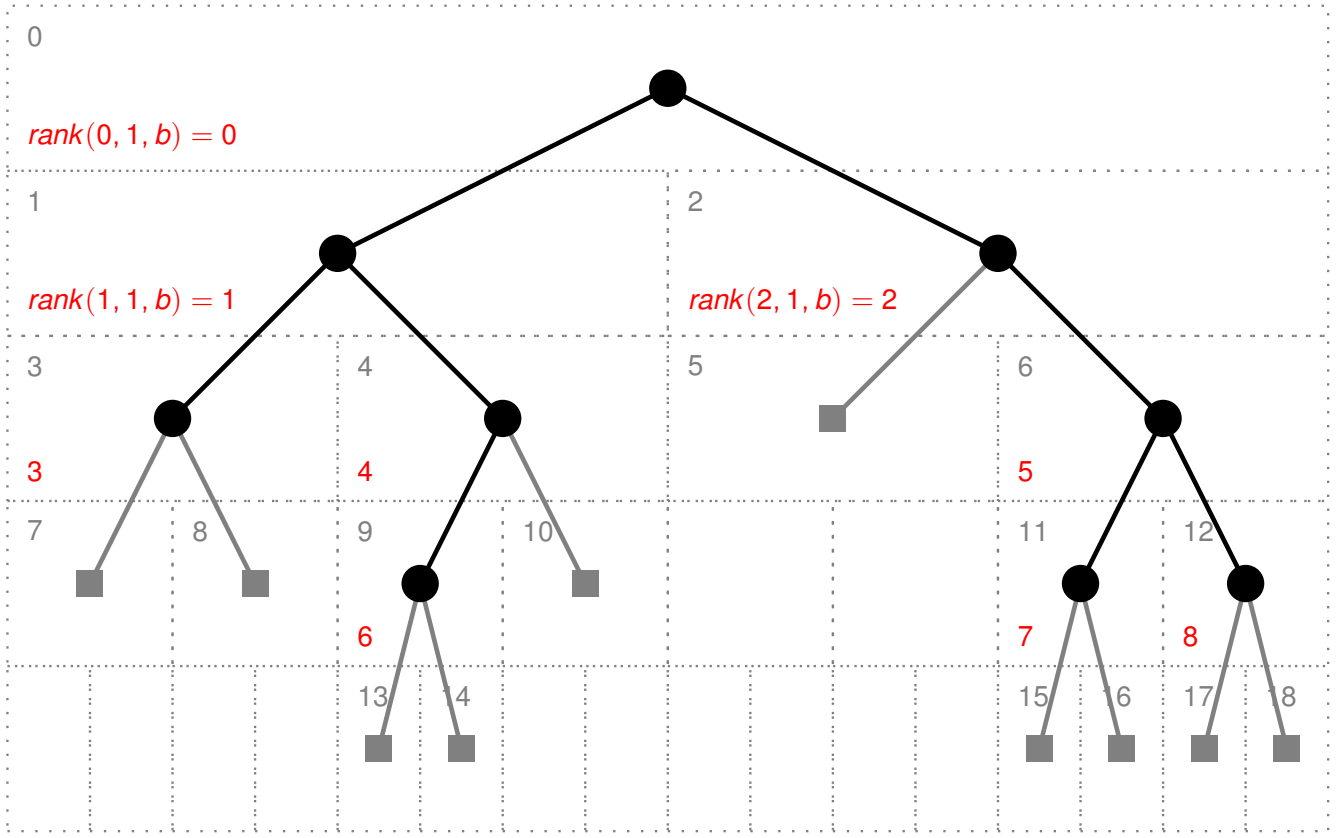
Succinct representation of trees

Probelm with previous approach:
For unbalanced trees the space would be 2^d bits, where d is the maximum depth of a node.

Proposal of Jacobson (FOCS 1989):

1. Mark all the nodes of the tree with a 1.
2. Add external nodes to the tree, and mark them all with 0-bits.
3. Read off the bits marking the nodes of the tree in (left-to-right) level-order.

Succinct representation of trees



$b =$

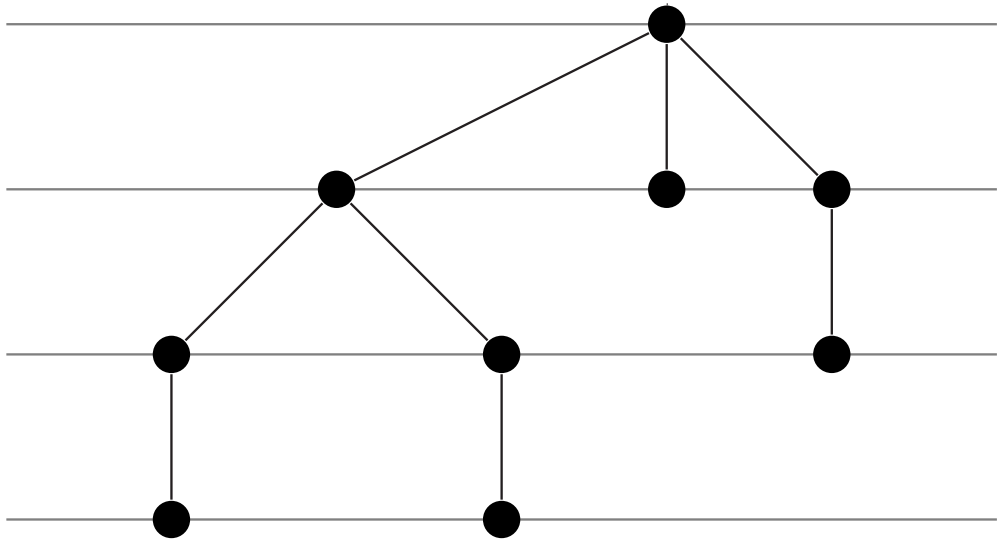
1	1	1	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Succinct representation of trees

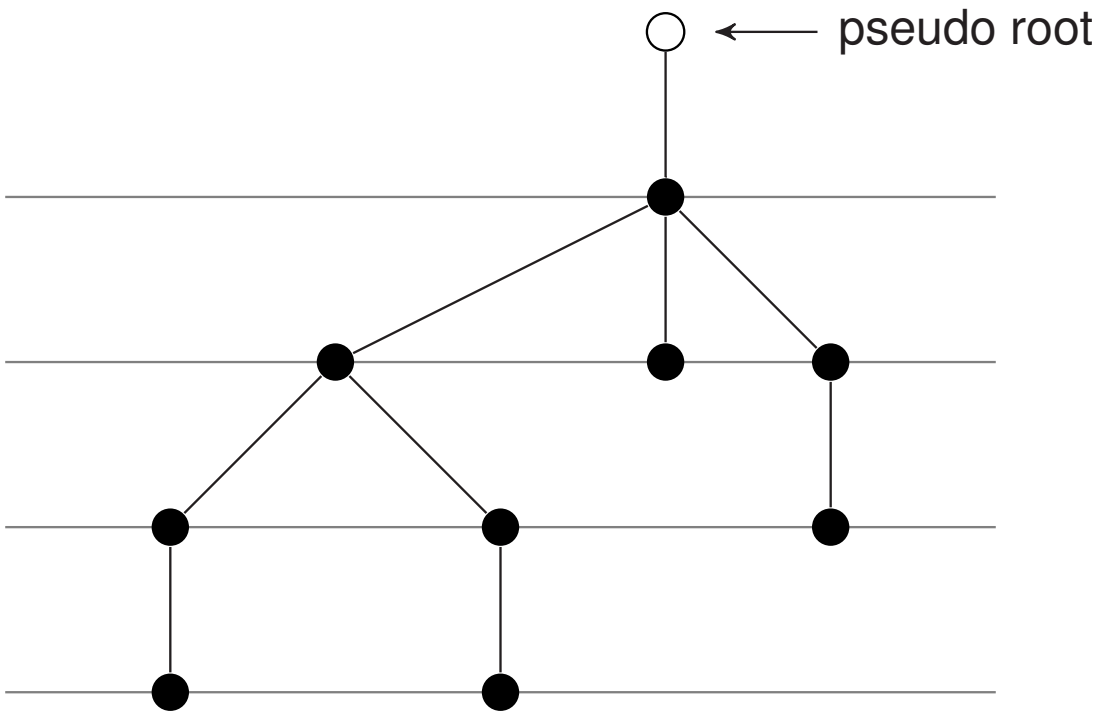
- b contains n set bits and is of length $2n + 1$.
- A node is represented by the positon of its corresponing 1-bit in b.
  - $leftchild(v) = 2 \cdot rank(v) + 1$
  - $rightchild(v) = 2 \cdot rank(v) + 2$
  - $parent(v)$ also possible in constant time (homework)
- Total space (including rank and select): $2n + o(n)$ bits

Jacobson also considered rooted, ordered tree with degree higher than 2.

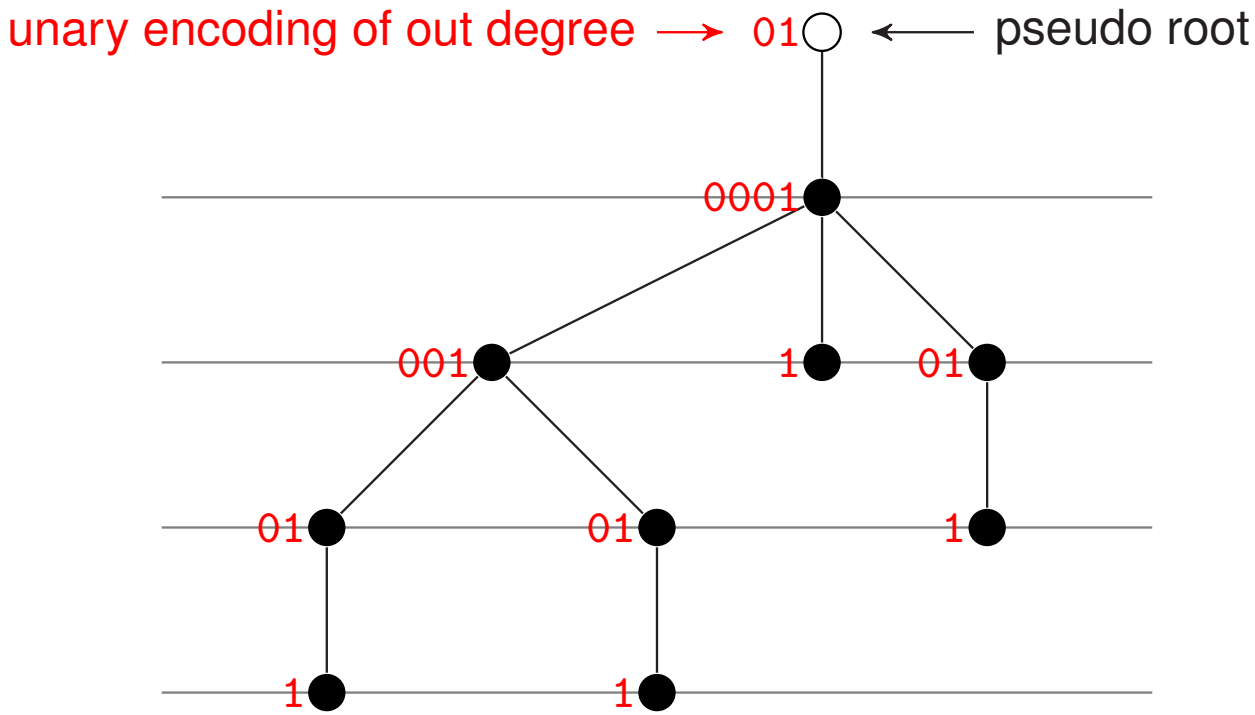
LOUDS – level order unary degree sequence



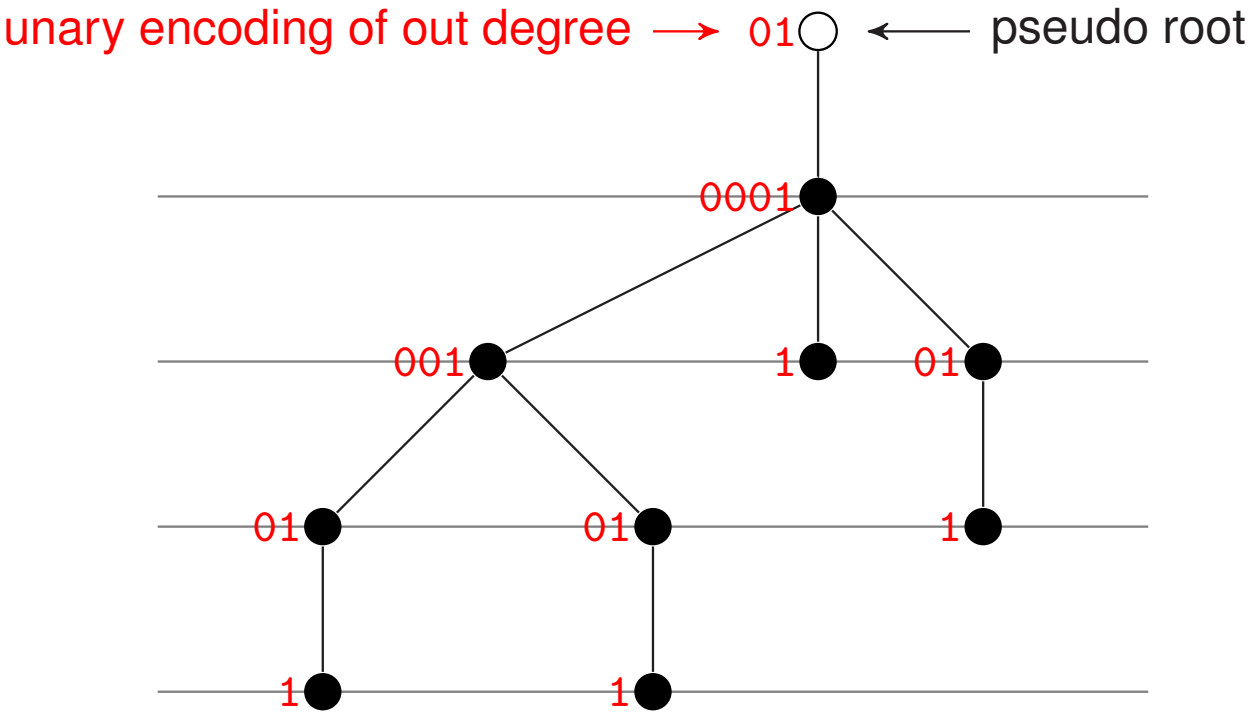
LOUDS – level order unary degree sequence



LOUDS – level order unary degree sequence



LOUDS – level order unary degree sequence



LOUDS sequence = 01000100110101011111
(concatenation of unary codes in level order)

LOUDS – level order unary degree sequence

- Each node (excpet the pseudo root) is represented twice
  - Once as „0” in the child list of its parent
  - Once as the terminal („1”) in its child list
- Represent node *v* by the index of its corresponding „0”
- I.e. *root* corresponds to „0”

LOUDS – level order unary degree sequence

Karlsruher Institut für Technologie

```
00  is_leaf(v)
01      id ← rank(v, 0, LOUDS)
02      p ← select(id + 2, 1, LOUDS)
03      if LOUDS[p − 1] = 1 then
04          return true
05      return false

00  out_degree(v)
01      if is_leaf(v) then
02          return 0
03      id ← rank(v, 0, LOUDS)
04      return select(id + 2, 1, LOUDS) − select(id + 1, 1, LOUDS) − 1
```

LOUDS – level order unary degree sequence

Get i-th child ($i \in [1, out\_degree(v)]$) of v and parent:

```
00  child(v,i)
01      if i > out_degree(v) then
02          return ⊥
03      id ← rank(v, 0, LOUDS)
04      return select(id + 1, 1, LOUDS) + i

00  parent(v)
01      if is_root(v) then
02          return ⊥
01      pid ← rank(v, 1, LOUDS)
04      return select(pid, 0, LOUDS)
```

LOUDS – level order unary degree sequence

Karlsruher Institut für Technologie

Conclusion

- Total space for LOUDS representation is $2n + 1 + o(n)$ bits
- All operations take constant time

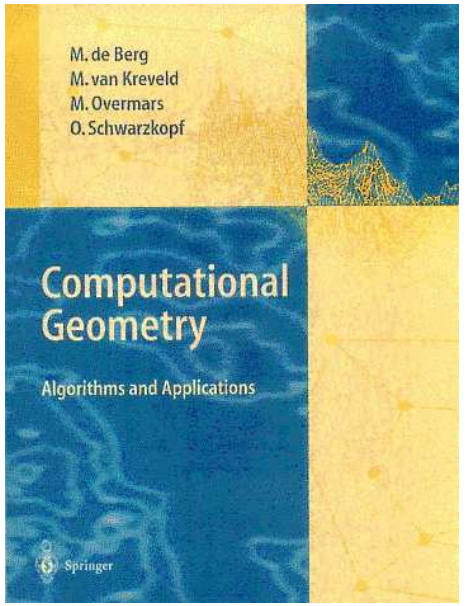
Karlsruhe Institute of Technology

6 Geometrische Algorithmen

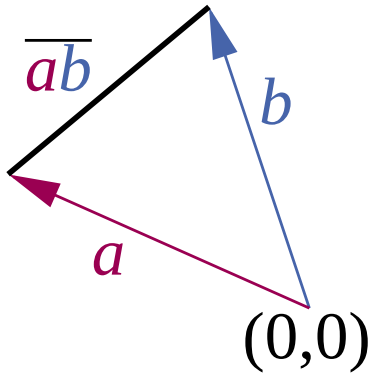
- Womit beschäftigen sich geom. Algorithmen?
- Schnitt von Strecken: Bentley-Ottmann-Algorithmus
- Konvexe Hüllen
- Kleinste einschließende Kugel
- Range Search

Quelle:

[Computational Geometry – Algorithms and Applications
de Berg, van Kreveld, Overmars, Schwartzkopf
Springer, 1997]



Elementare Geometrische Objekte



Punkte: $x \in \mathbb{R}^d$

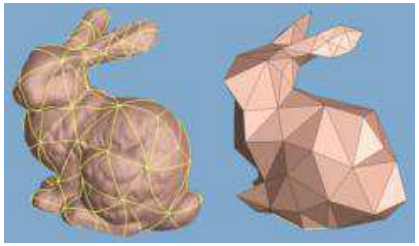
Strecken: $\overline{ab} := \{\alpha a + (1-\alpha)b : \alpha \in [0,1]\}$

uvam: Halbräume, Ebenen, Kurven,...

Dimension d:

1: Oft trivial. Gilt i. allg. nicht als geometrisches Problem

2: Geogr. Informationssysteme (GIS), Bildverarbeitung,...



3: Computergrafik, Simulationen,...

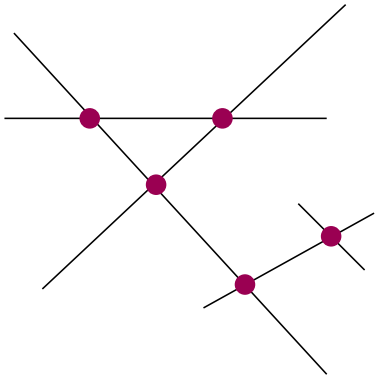
≥ 4: Optimierung, Datenbanken, maschinelles Lernen,...

curse of dimensionality!

n: Anzahl vorliegender Objekte

Typische Fragestellungen

- Schnittpunkte zwischen n Strecken



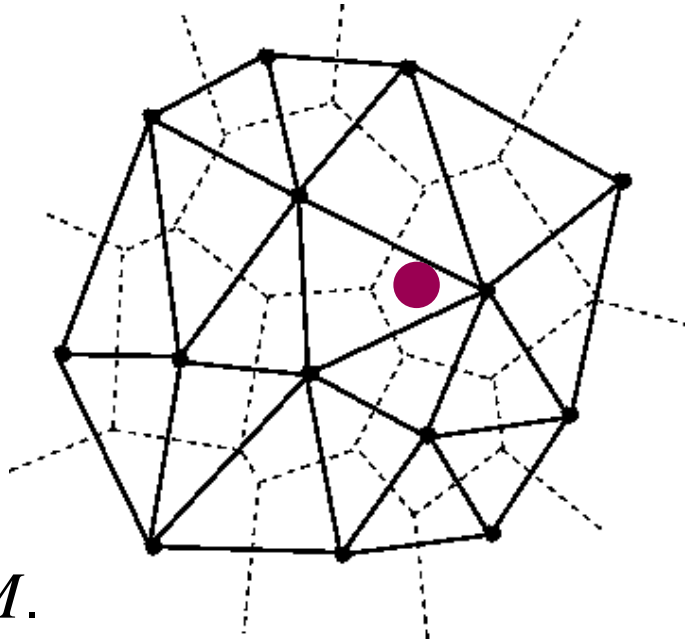
Typische Fragestellungen

- Schnittpunkte zwischen n Strecken
- Konvexe Hülle
- Triangulation von Punktmengen

  (2D, verallgemeinerbar)

  z.B. Delaunaytriangulierung:

  Kein Dreieck enthält weiteren Punkt



- Voronoi-Diagramme: Sei $x \in M \subseteq \mathbb{R}^d$.

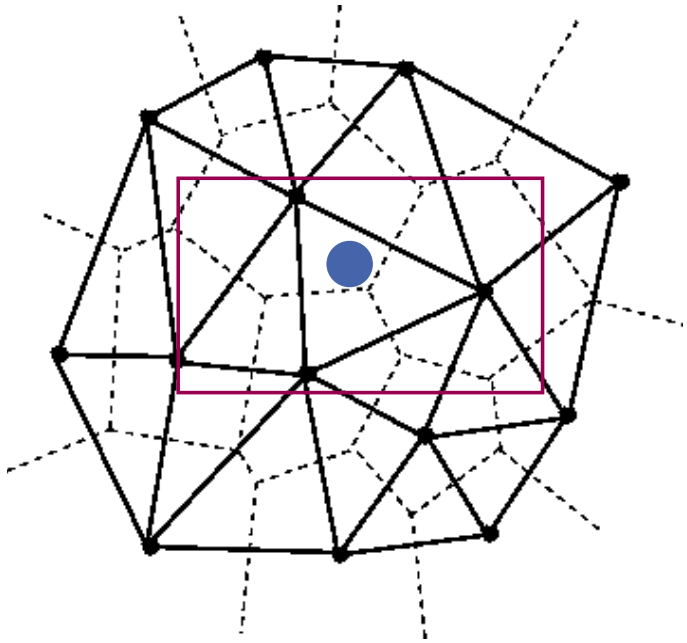
  $\forall y \in \mathbb{R}^d$ bestimme nächstes Element aus M.

  (Unterteilung von M in n Voronoizellen)
- Punklokalisierung: Geg. Unterteilung von R^d, $x \in R^d$:

  in welchem Teil liegt x ?

Datenstrukturen für Punktmengen

- nächsten Nachbarn berechnen
- Bereichsanfragen
- ...



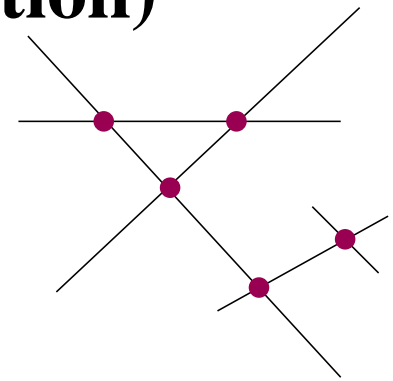
Mehr Fragestellungen

- Sichtbarkeitsberechnungen
- Lineare Programmierung
- Geometrische Versionen von Optimierungsproblemen
  - Kürzeste Wege, z.B.

    energieffiziente Kommunikation in Radionetzwerken
  - minimale Spannbäume

    reduzierbar auf Delaunay-Triangulierung + Graphalgorithmus
  - Matchings
  - Handlungsreisendenproblem
  - ...
- ...

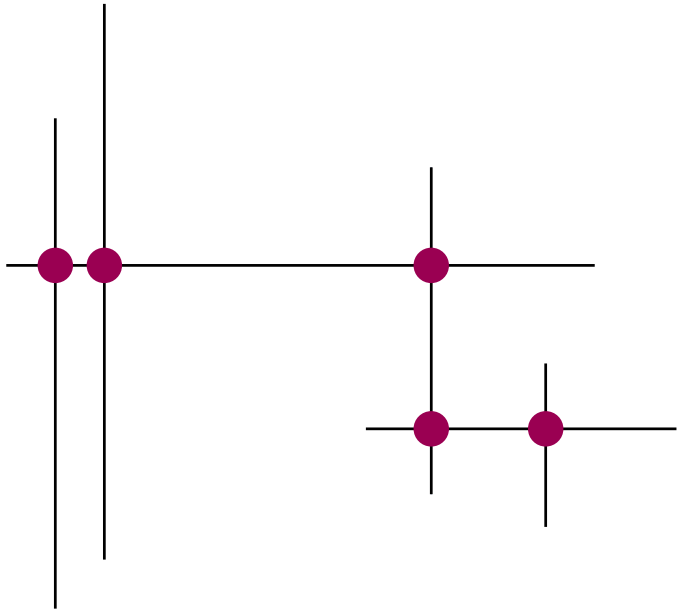
6.1 Streckenschnitt (line segment intersection)



Gegeben: $S = \{s_1, \ldots, s_n\}$, n Strecken

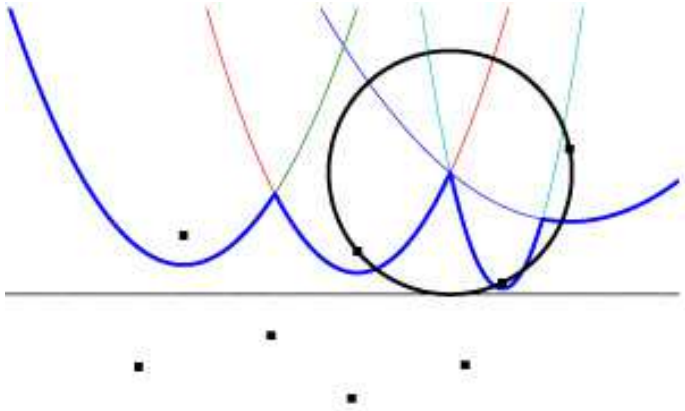
Gesucht: Schnittpunkte $\bigcup_{s,t \in S} s \cap t$

Zum Warmwerden: Orthogonaler Streckenschnitt – die Strecken sind parallel zur x- oder y-Achse



Streckenschnitt: Anwendungen

- Schaltungsentwurf: wo kreuzen sich Leiterbahnen?
- GIS: Strassenkreuzungen, Brücken,. . .
- Erweiterungen: z.B. Graphen benachbarter Strecken/Flächen aufbauen/verarbeiten
- Noch allgemeiner: Plane-Sweep-Algorithmen für andere Fragestellungen (z.B. Konstruktion von konvexen Hüllen oder Voronoi-diagrammen)



Streckenschnitt: Naiver Algorithmus

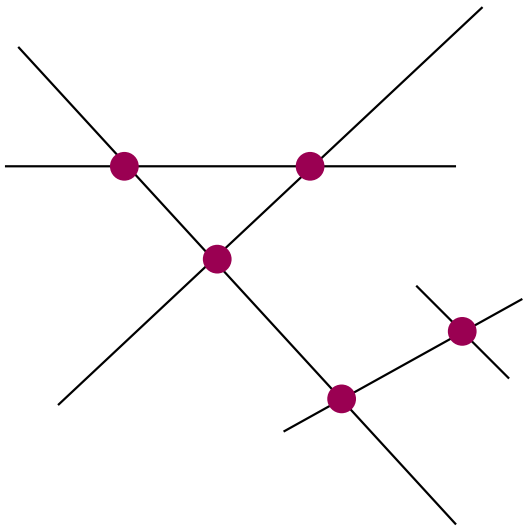
foreach $\{s,t\} \subseteq S$ **do**

  if $s \cap t \neq \emptyset$ **then**

    output $\{s,t\}$

Problem: Laufzeit $\Theta\left(n^2\right)$.

Zu langsam für große Datenmengen

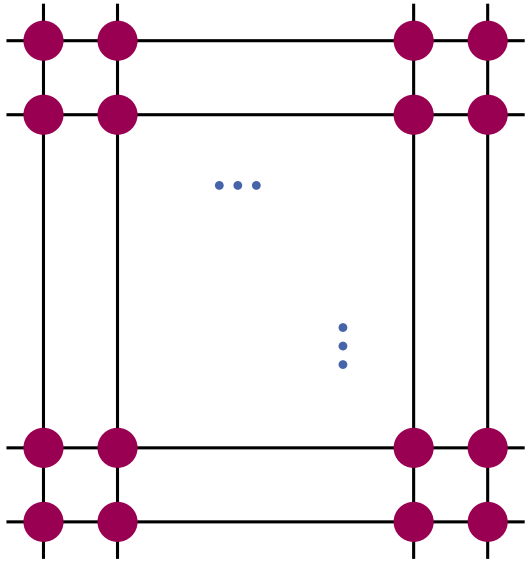


Streckenschnitt: Untere Schranke

$\Omega(n+k)$ mit k:= Anzahl ausgegebener Schnitte.

Vergleichsbasiert: $\Omega(n \log n + k)$ (Beweis: nicht hier)

Beobachtung $k = \Theta(n^2)$ ist möglich, aber reale Eingaben haben meist $k = \mathrm{O}(n)$.

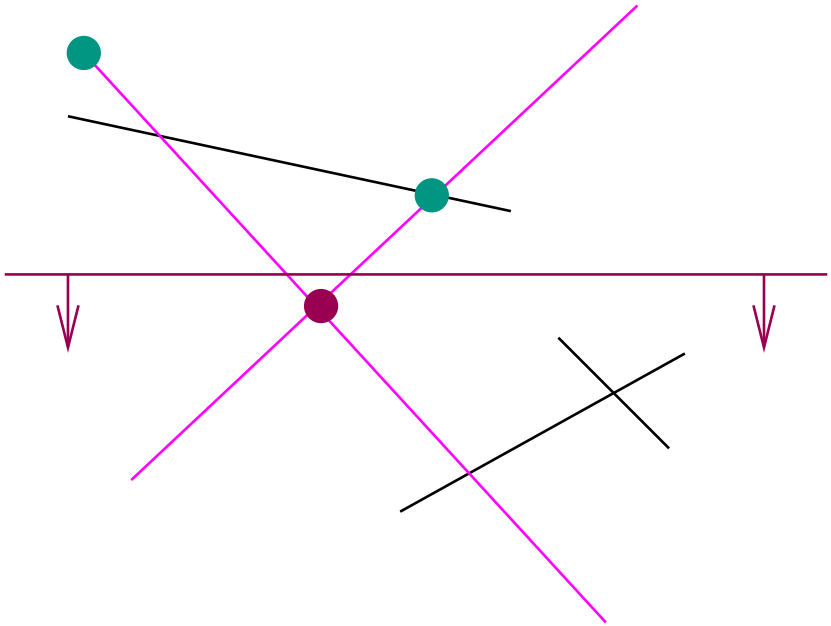


Idee: Plane-Sweep-Algorithmen

(Waagereche) Sweep-Line ℓ läuft von oben nach unten.

Invariante: Schnittpunkte oberhalb von ℓ wurden korrekt ausgegeben.

Ansatz: speichere jeweils Segmente, die ℓ schneiden und finde deren Schnittpunkte.



Plane-Sweep für orth. Streckenschnitt

Erstmal nur: Schnitte zwischen vertikalen und horizontalen Strecken.

$T = \langle\rangle$: SortedSequence **of** Segment

invariant T stores the vertical segments intersecting ℓ

Q:= sort($\langle (y,s) : \exists$hor. seg. s at y or $\exists$vert. seg. s starting/ending at $y \rangle$)

// tie breaking: vert. starting events first, vert. finishing events last

foreach $(y,s) \in Q$ in descending order **do**

if s is a vertical segment and starts at y **then** T.insert(s)

else if s is a vertical segment and ends at y **then** T.remove(s)

else // we have a horizontal segment $s = \overline{(x_1,y)(x_2,y)}$

foreach $t = \overline{(x,y_1)(x,y_2)} \in T$ with $x \in [x_1,x_2]$ **do**

output $\{s,t\}$

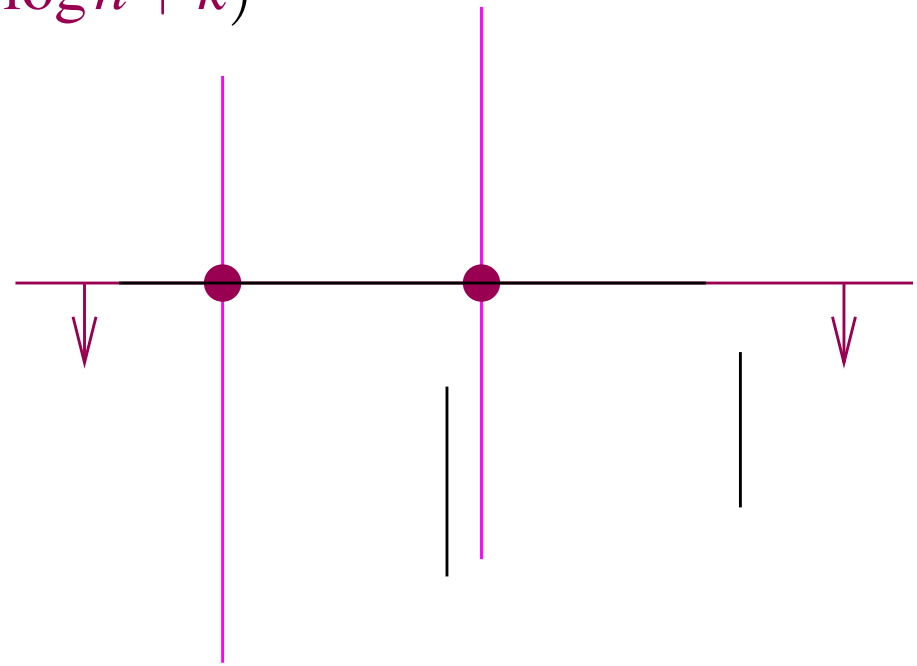
Analyse orth. Streckenschnitt

insert: $O(\log n)$ $(\leq n\times)$

remove: $O(\log n)$ $(\leq n\times)$

rangeQuery: $O(\log n + k_s)$, k_s Schnitte mit hor. Segment s

Insgesamt: $O(n\log n + \sum_s k_s) = O(n\log n + k)$



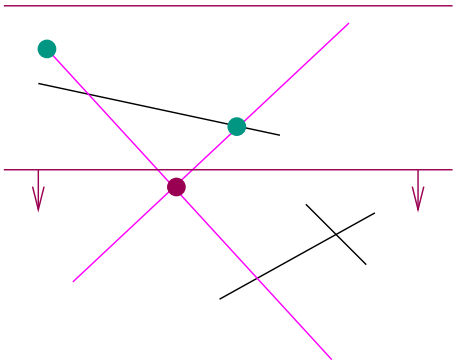
Verallgemeinerung – aber erstmal “nicht ganz”

Annahme zunächst:

Allgemeine Lage, d.h. hier

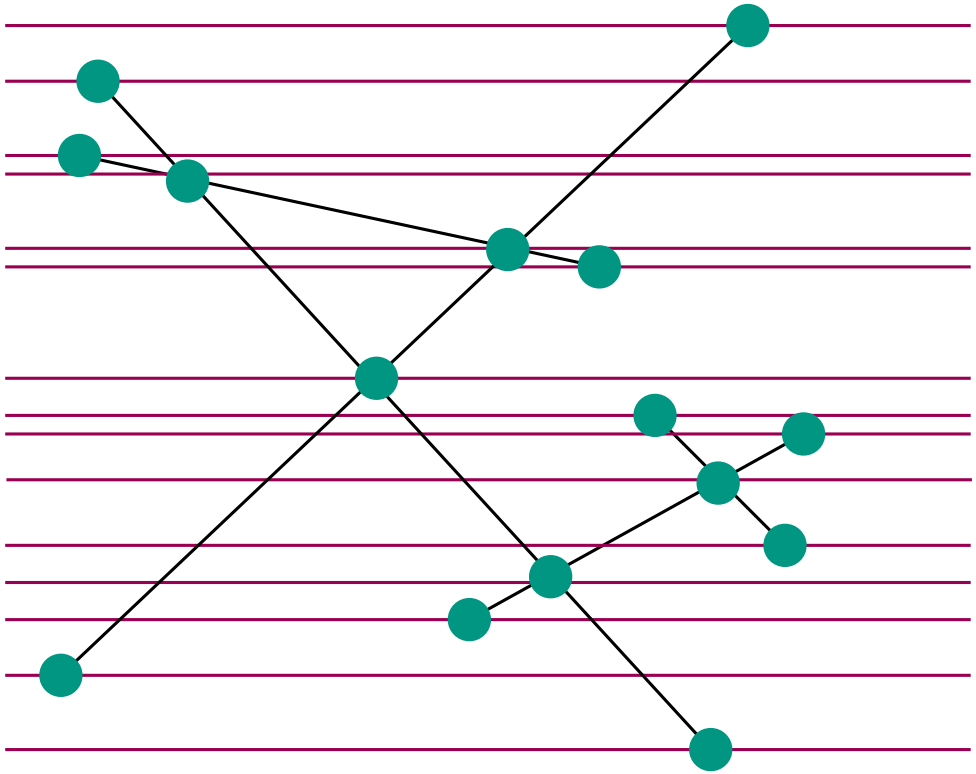
- Keine horizontalen Strecken
- Keine Überlappungen
- Schnittpunkte jeweils im Inneren von genau zwei Strecken

Beobachtung: kleine zuf. Perturbationen produzieren allg. Lage.



Verallgemeinerung – Grundidee

- Plane-Sweep mit Sweep-Line ℓ
- Status T := nach x geordnete Folge der ℓ schneidenden Strecken
- Ereignis:= Statusänderung
  - Startpunkte
  - Endpunkte
  - Schnittpunkte
- Schnitttest nur für Segmente, die an einem Ereignispunkt in T benachbart sind.



Verallgemeinerung – Korrektheit

Lemma:

$s \cap t = \{(x,y)\} \longrightarrow \exists$ Ereignis : s,t werden Nachbarn auf ℓ

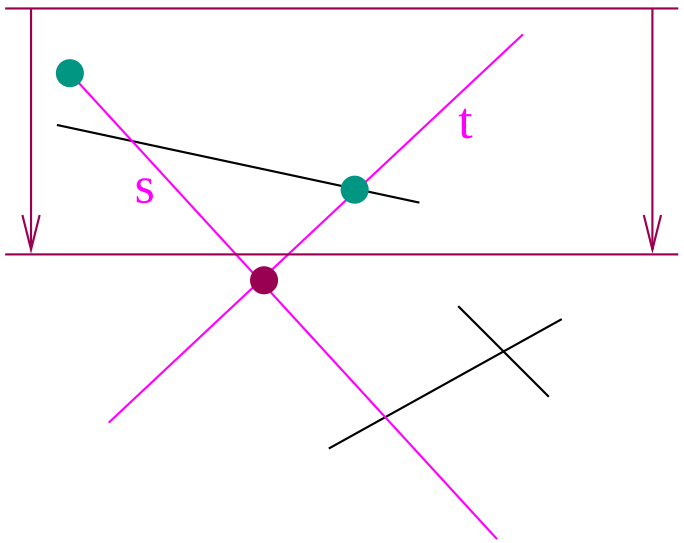
Beweis:

Anfangs : $T = \langle\rangle \longrightarrow s,t$ sind nicht in ℓ benachbart.

@$y+\varepsilon$: s,t sind in ℓ benachbart.

$\longrightarrow$

$\exists$ Ereignis bei dem s und t Nachbarn werden.



Verallgemeinerung – Implementierung

$T = \langle\rangle$: SortedSequence **of** Segment

invariant T stores the relative order of the segments intersecting ℓ

Q : MaxPriorityQueue

$Q := Q \cup \left\{ (\max\{y, y'\}, \text{start}, s) : s = \overline{(x,y)(x',y')} \in S \right\}$ // $O(n \log n)$

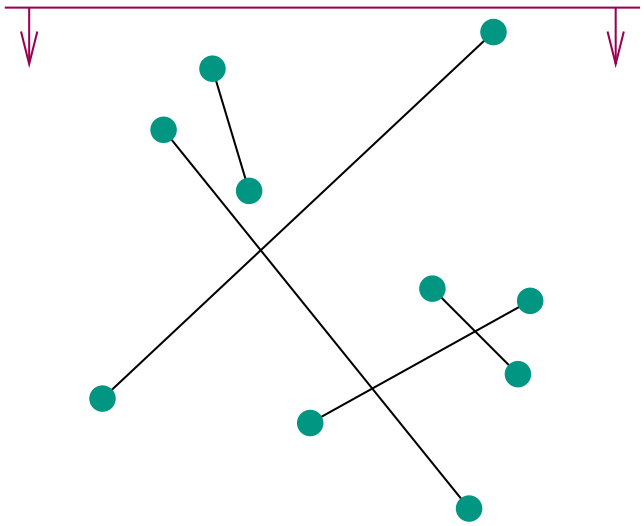
$Q := Q \cup \left\{ (\min\{y, y'\}, \text{finish}, s) : s = \overline{(x,y)(x',y')} \in S \right\}$ // $O(n \log n)$

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

$(y, \text{type}, s) := Q$.deleteMax

// $O((n+k)\log n)$

handleEvent$(y, \text{type}, s, T, Q)$



handleEvent$(y, \text{start}, s, T, Q)$ // $n\times$

  h:= T.insert(s) // $O(\log n)$

  prev:= pred(h) // $O(1)$

  next:= succ(h) // $O(1)$

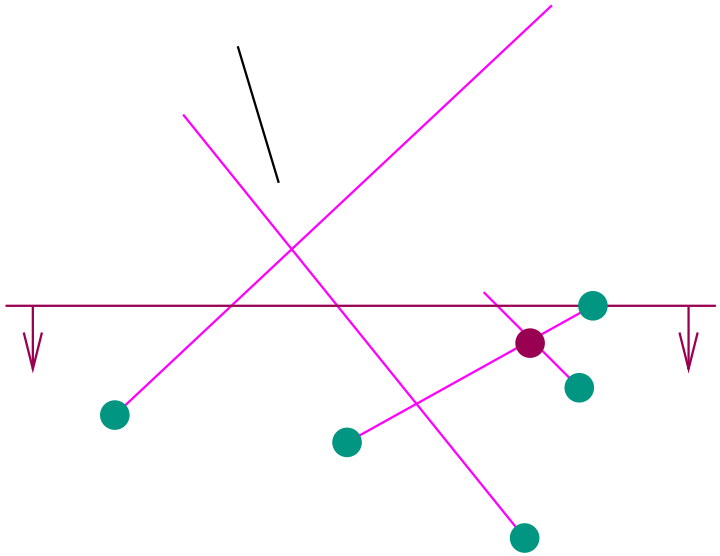
  findNewEvent(prev, h)

  findNewEvent(h, next)

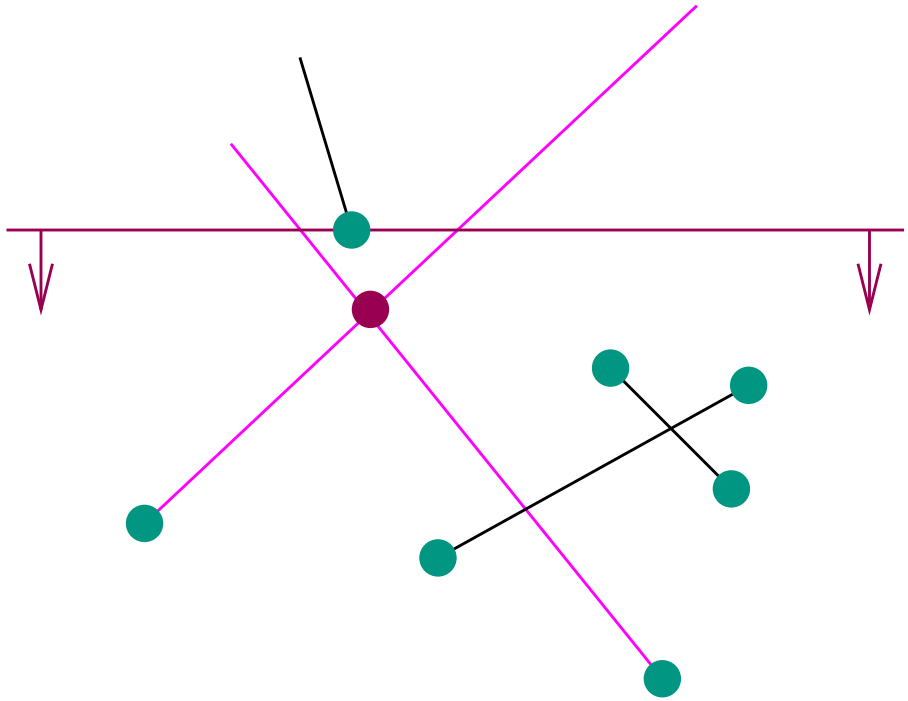
Procedure findNewEvent(s,t) // $O(1+\log n)$

  if $*s$ and $*t$ intersect at $y' < y$ **then**

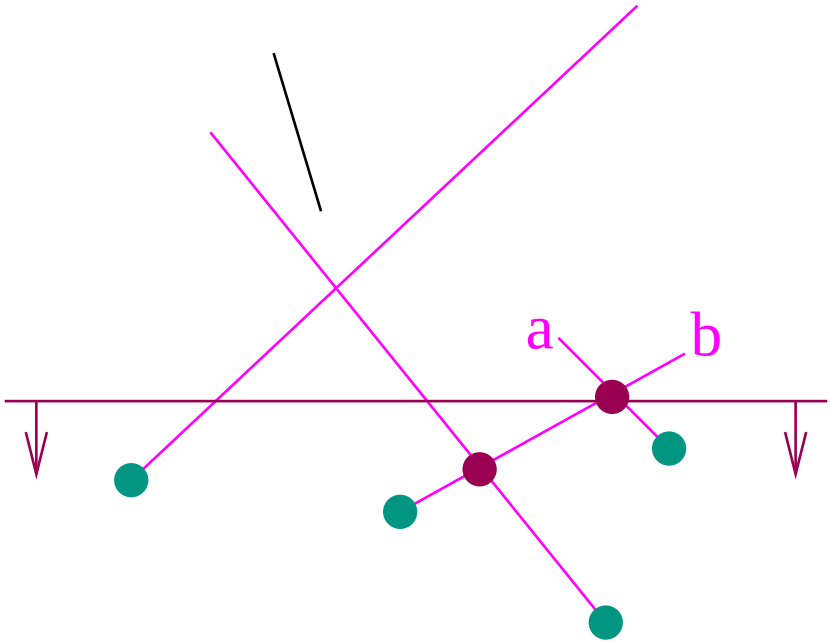
    Q.insert$((y', \text{intersection}, (s,t)))$



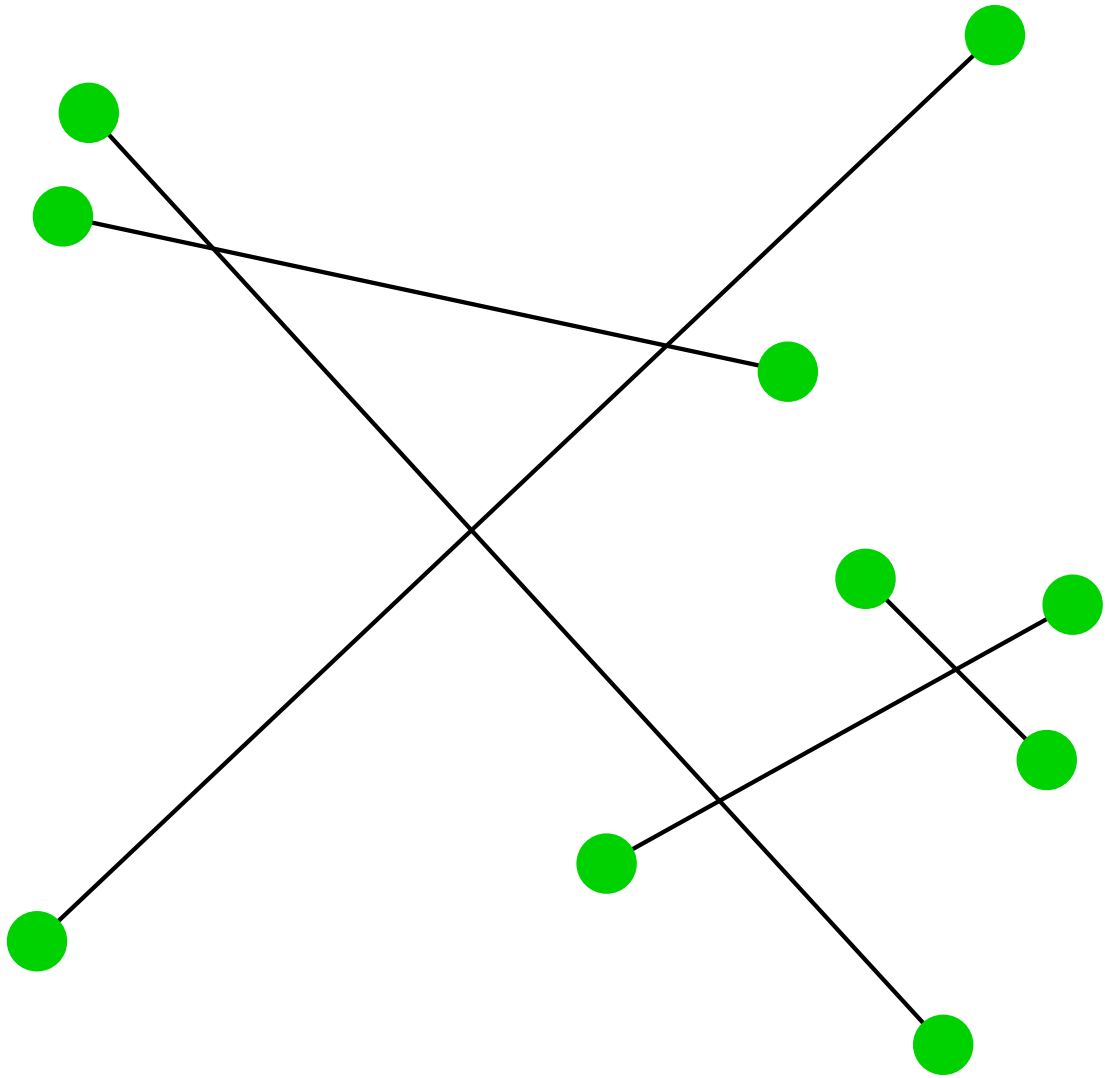
```
handleEvent(y, finish, s, T, Q)                    // n×
    h:= T.locate(s)                                // O(log n)
    prev:= pred(h)                                 // O(1)
    next:= succ(h)                                 // O(1)
    T.remove(s)                                    // O(log n)
    findNewEvent(prev, next)
```



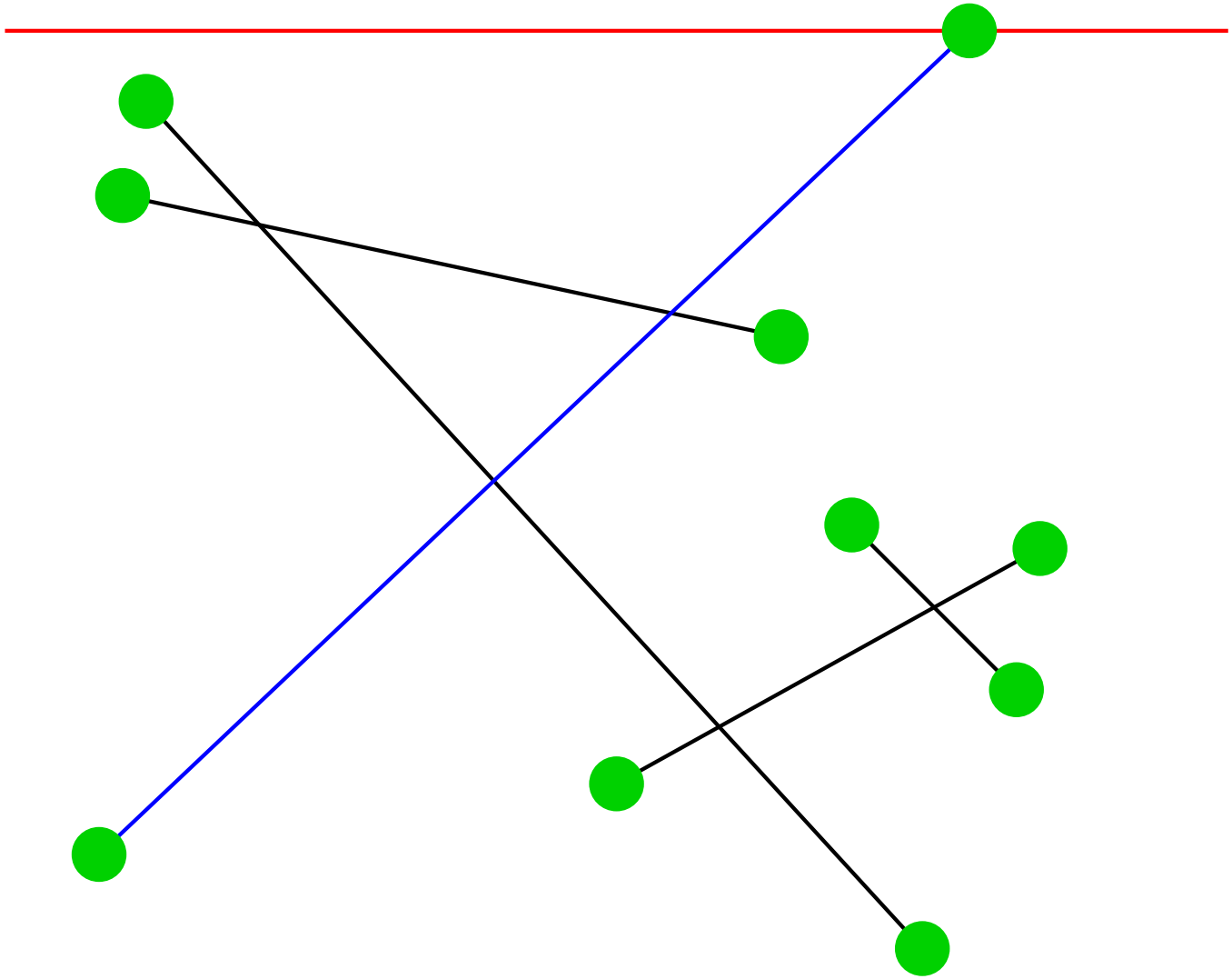
```
handleEvent(y, intersection, (a,b), T, Q)          // k×
    output (∗s ∩ ∗t)                               // O(1)
    T.swap(a,b)                                    // O(log n)
    prev:= pred(b)                                 // O(1)
    next:= succ(a)                                 // O(1)
    findNewEvent(prev, b)
    findNewEvent(a, next)
```



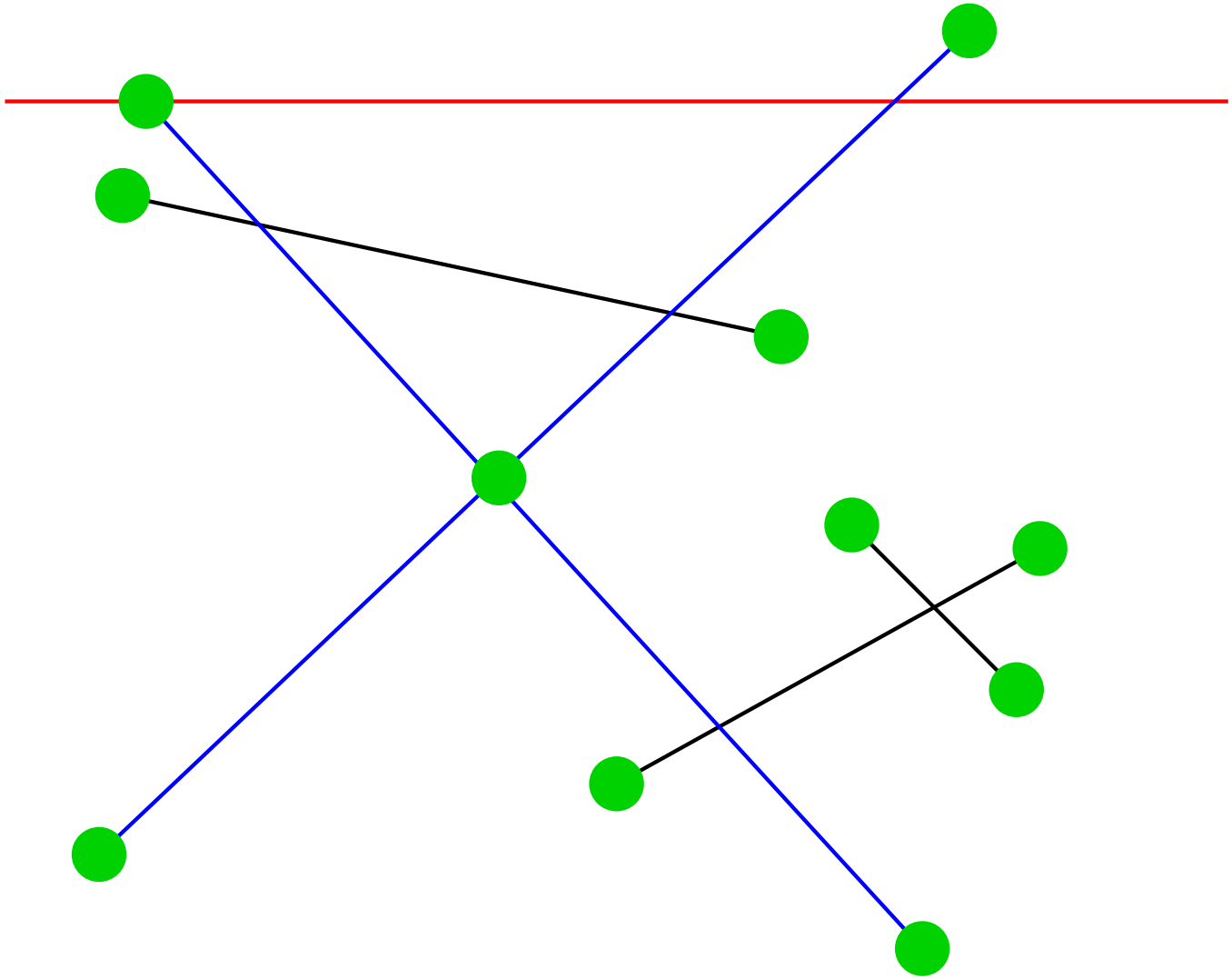
Verallgemeinerung – Beispiel



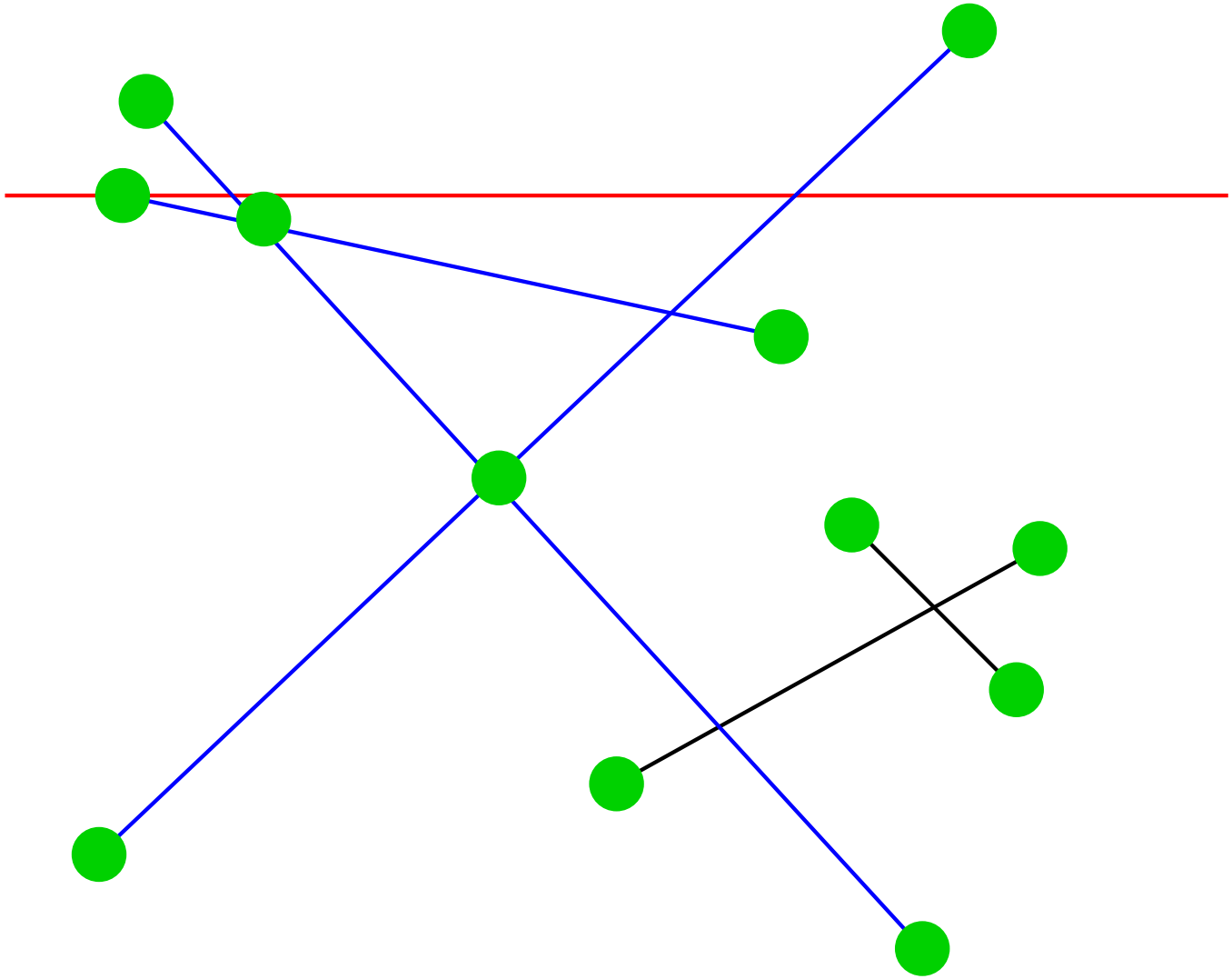
Verallgemeinerung – Beispiel



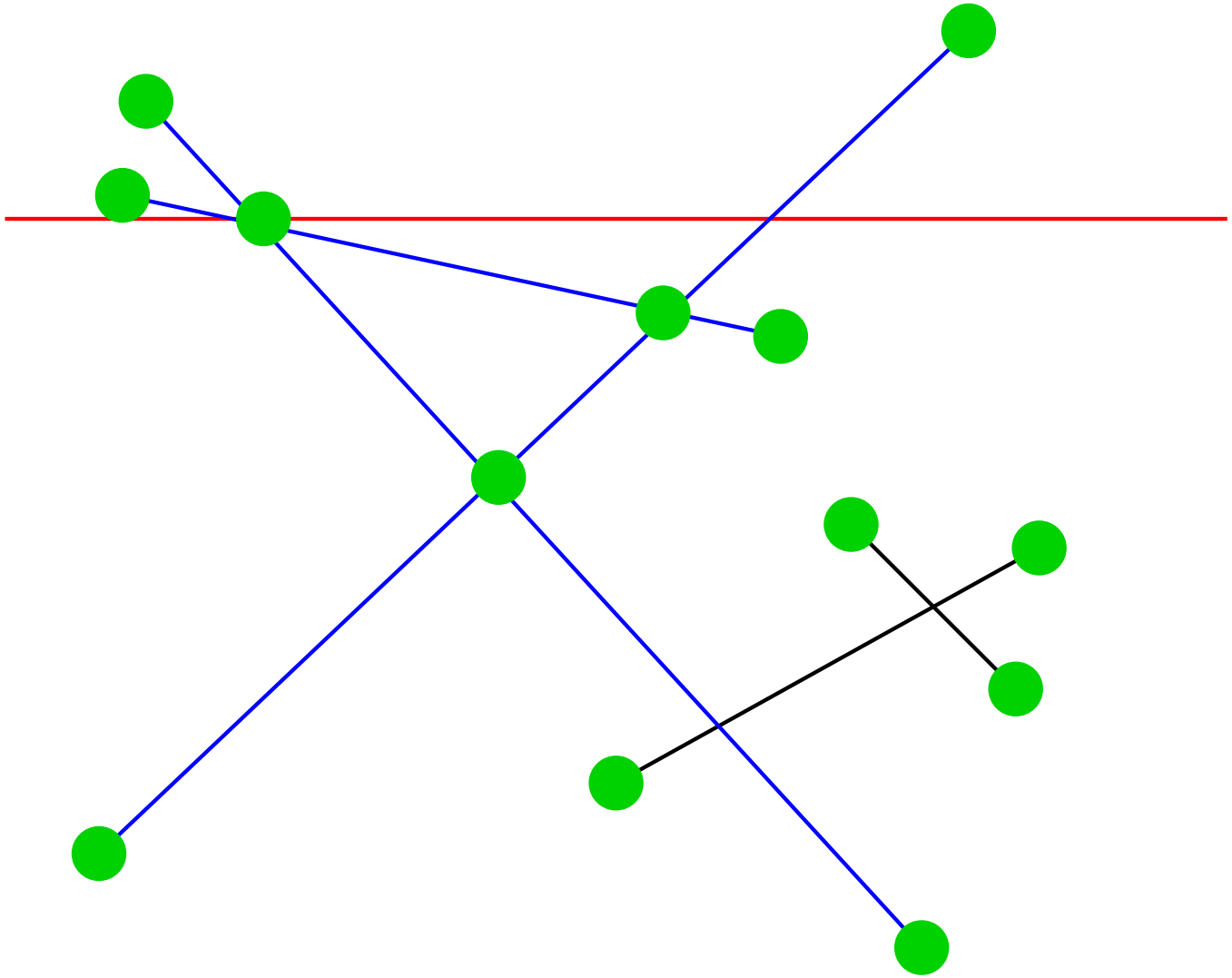
Verallgemeinerung – Beispiel



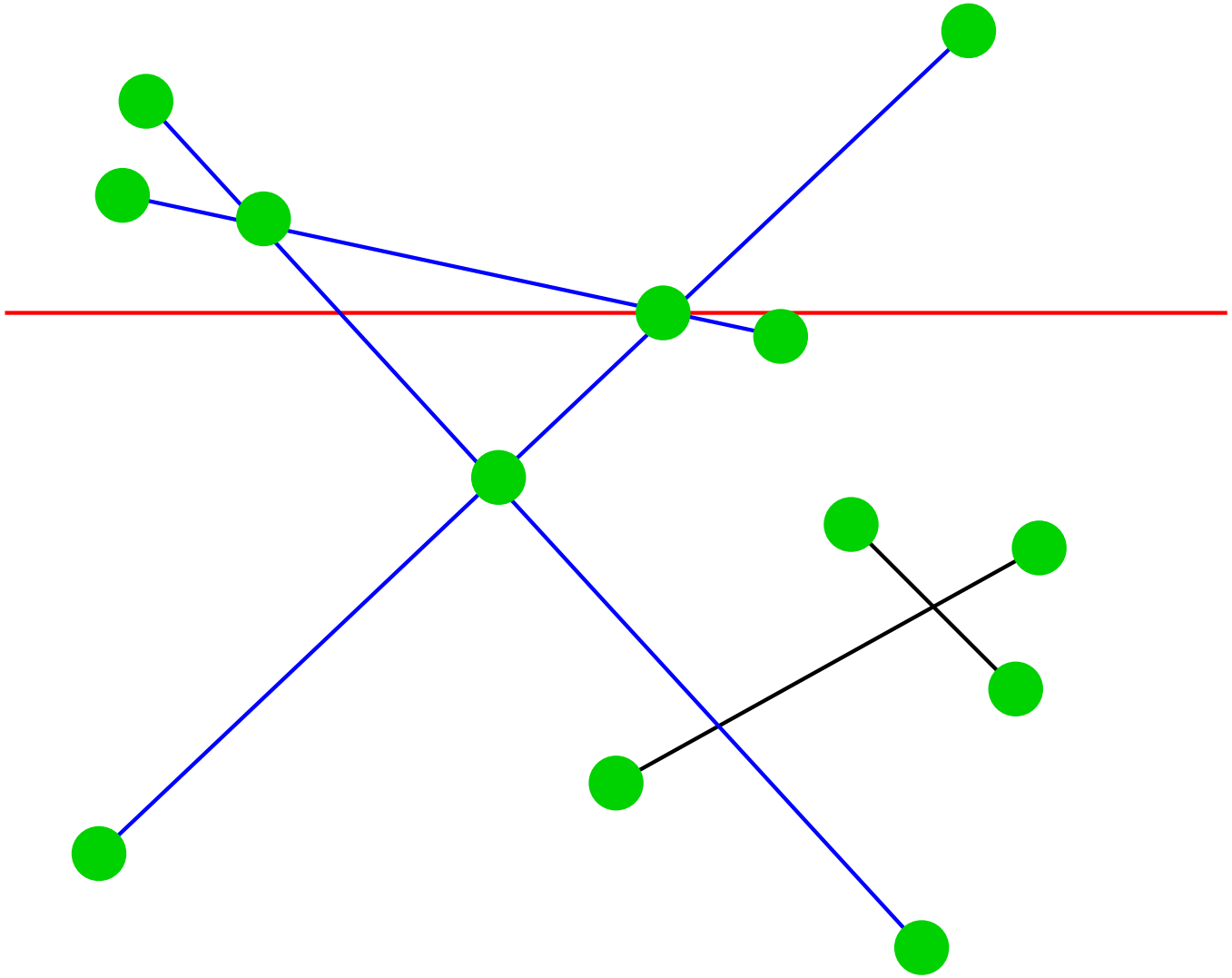
Verallgemeinerung – Beispiel



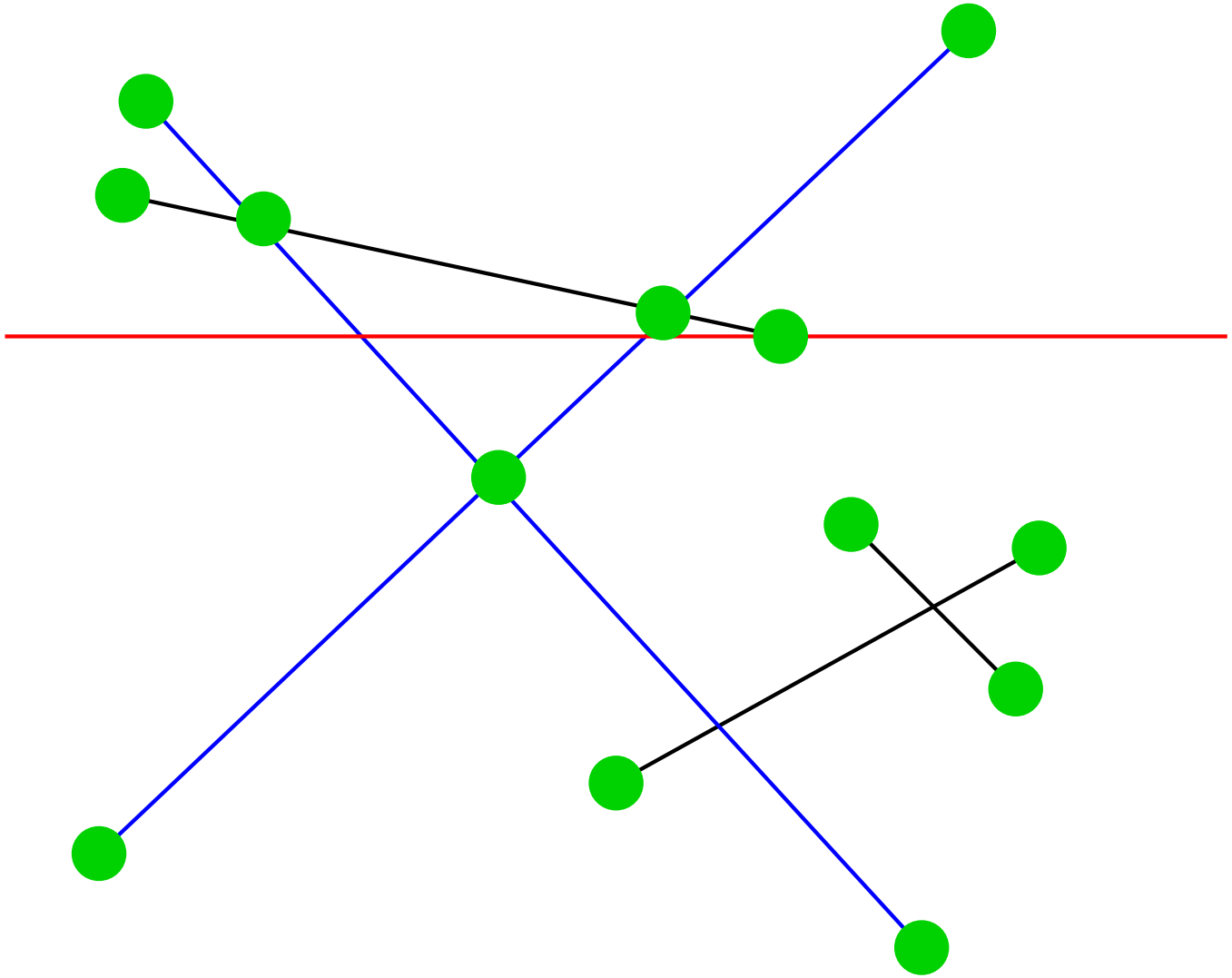
Verallgemeinerung – Beispiel



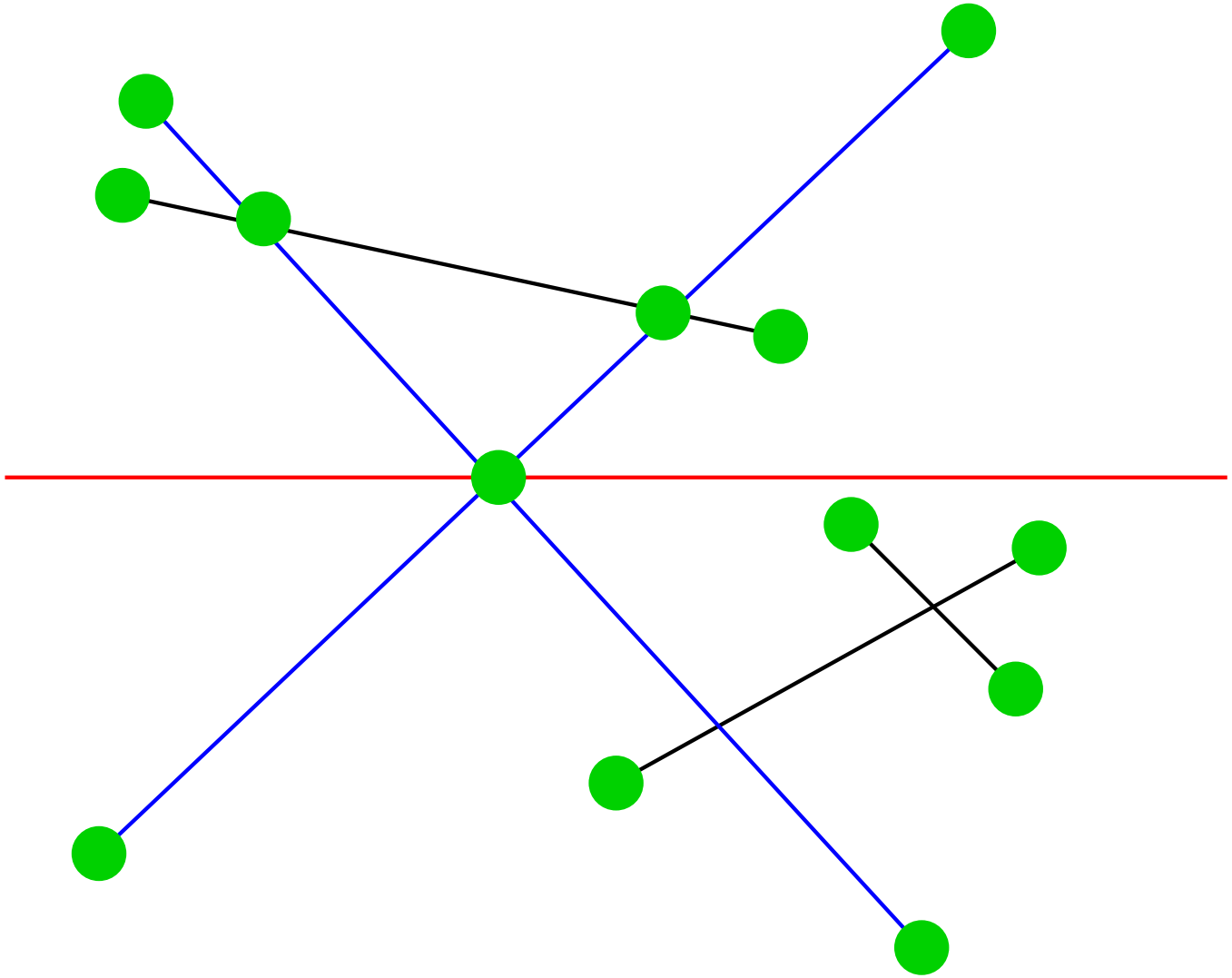
Verallgemeinerung – Beispiel



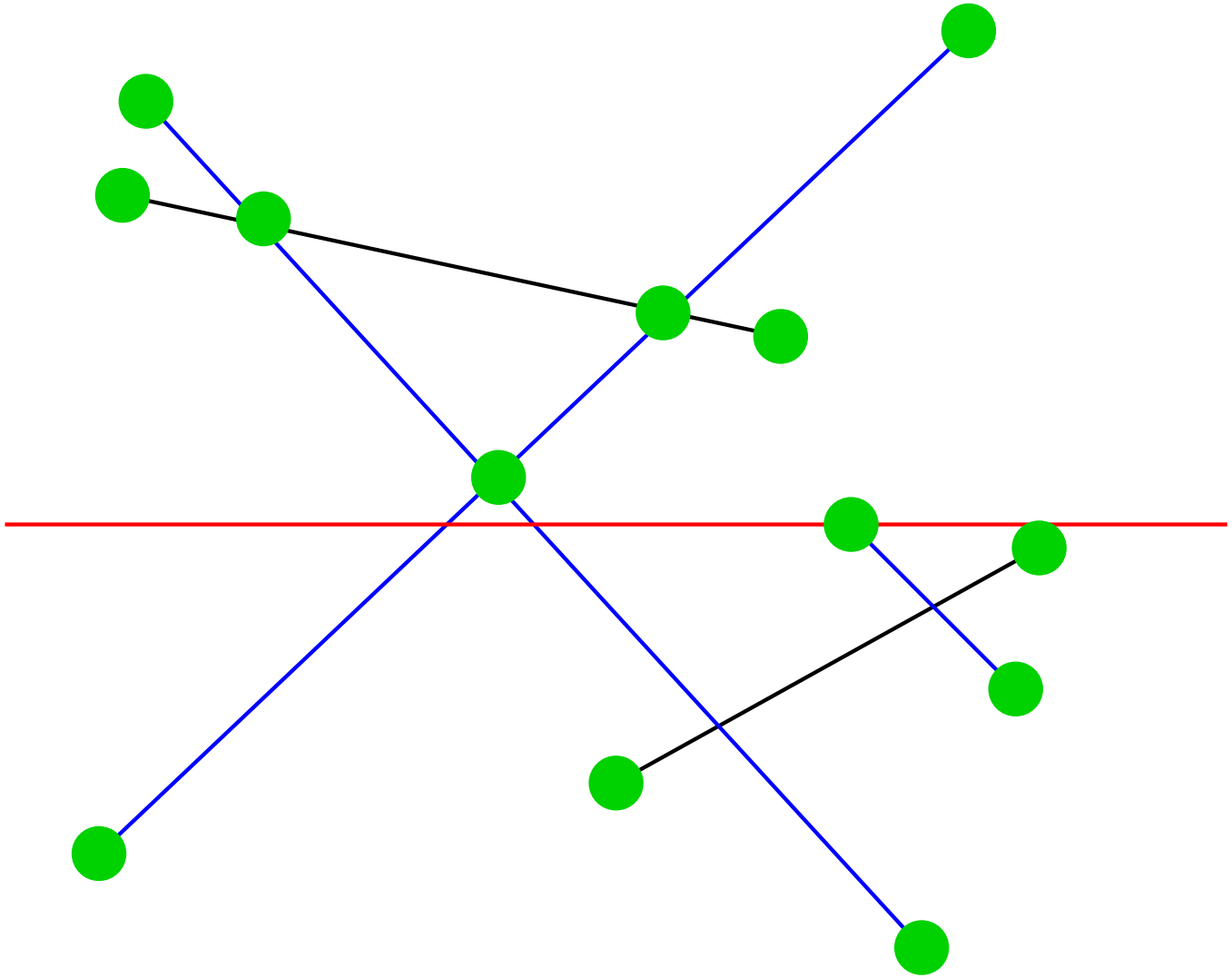
Verallgemeinerung – Beispiel



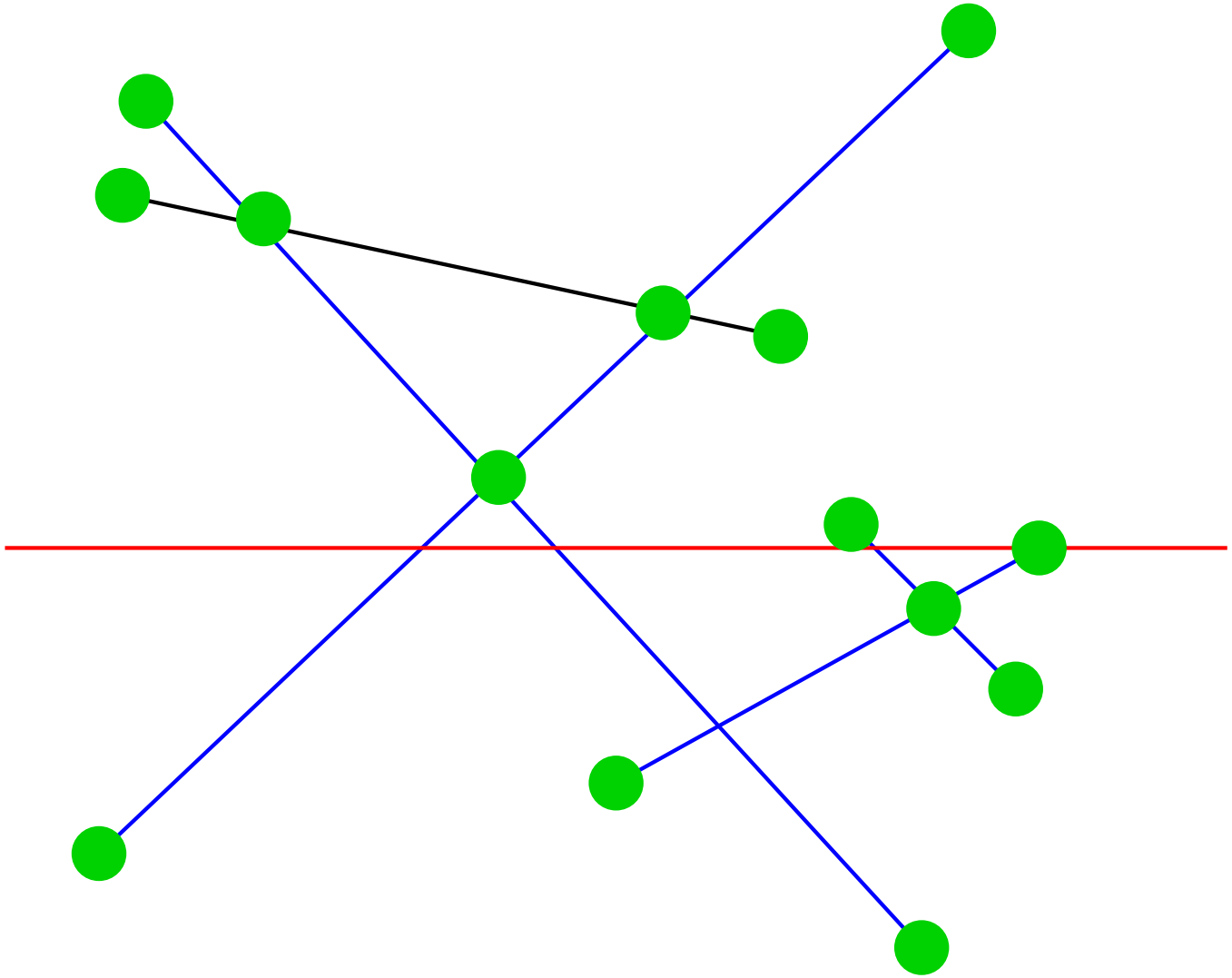
Verallgemeinerung – Beispiel



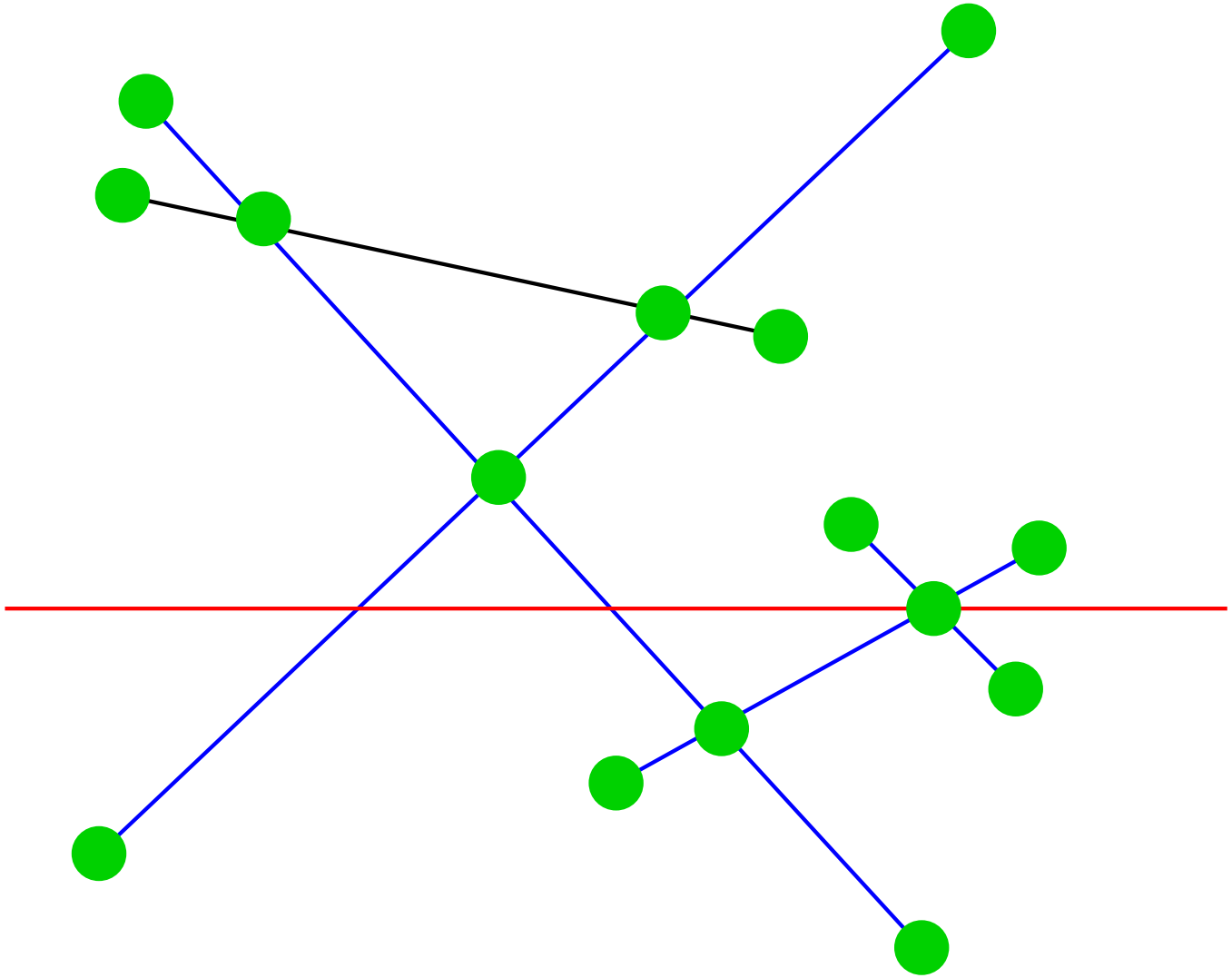
Verallgemeinerung – Beispiel



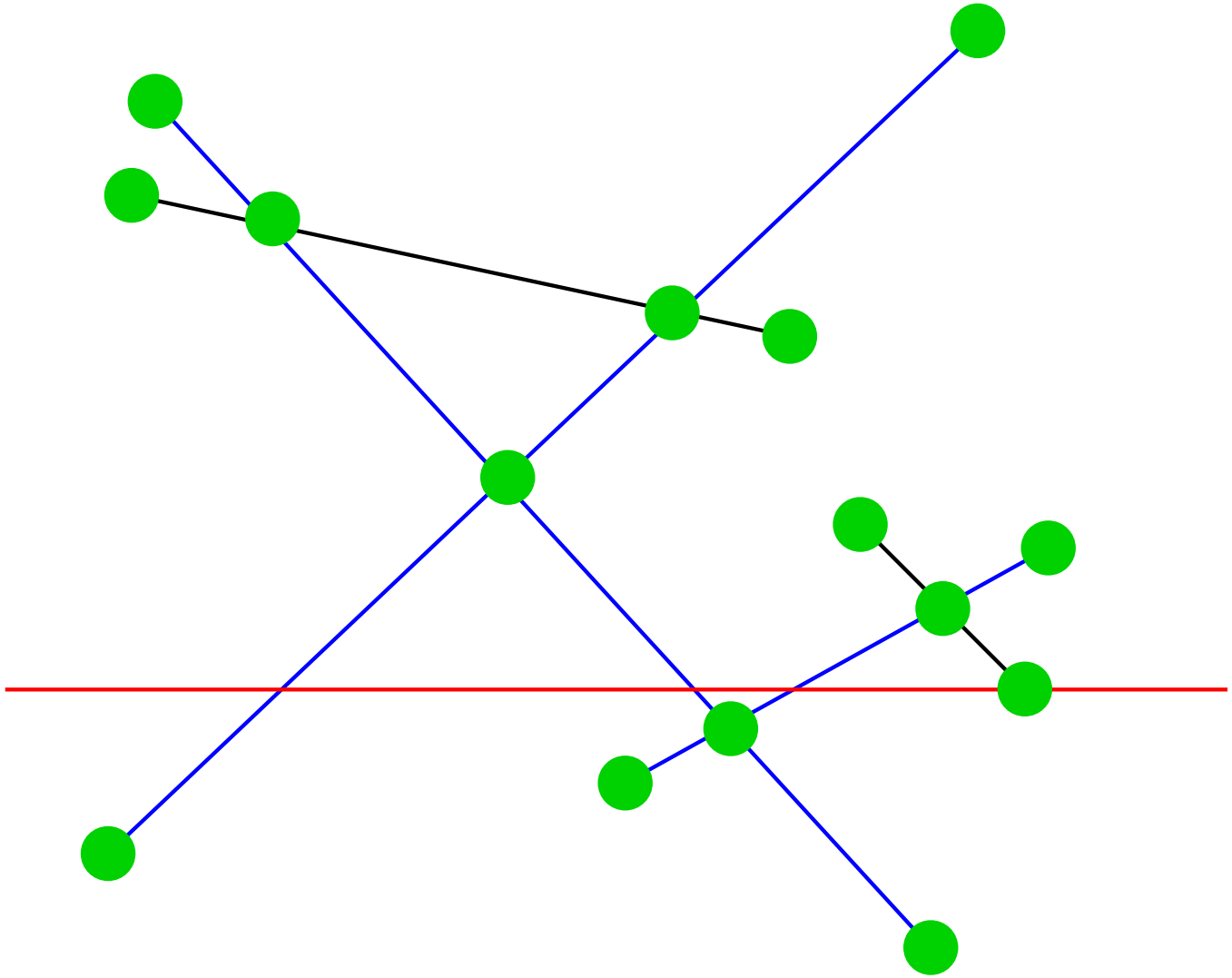
Verallgemeinerung – Beispiel



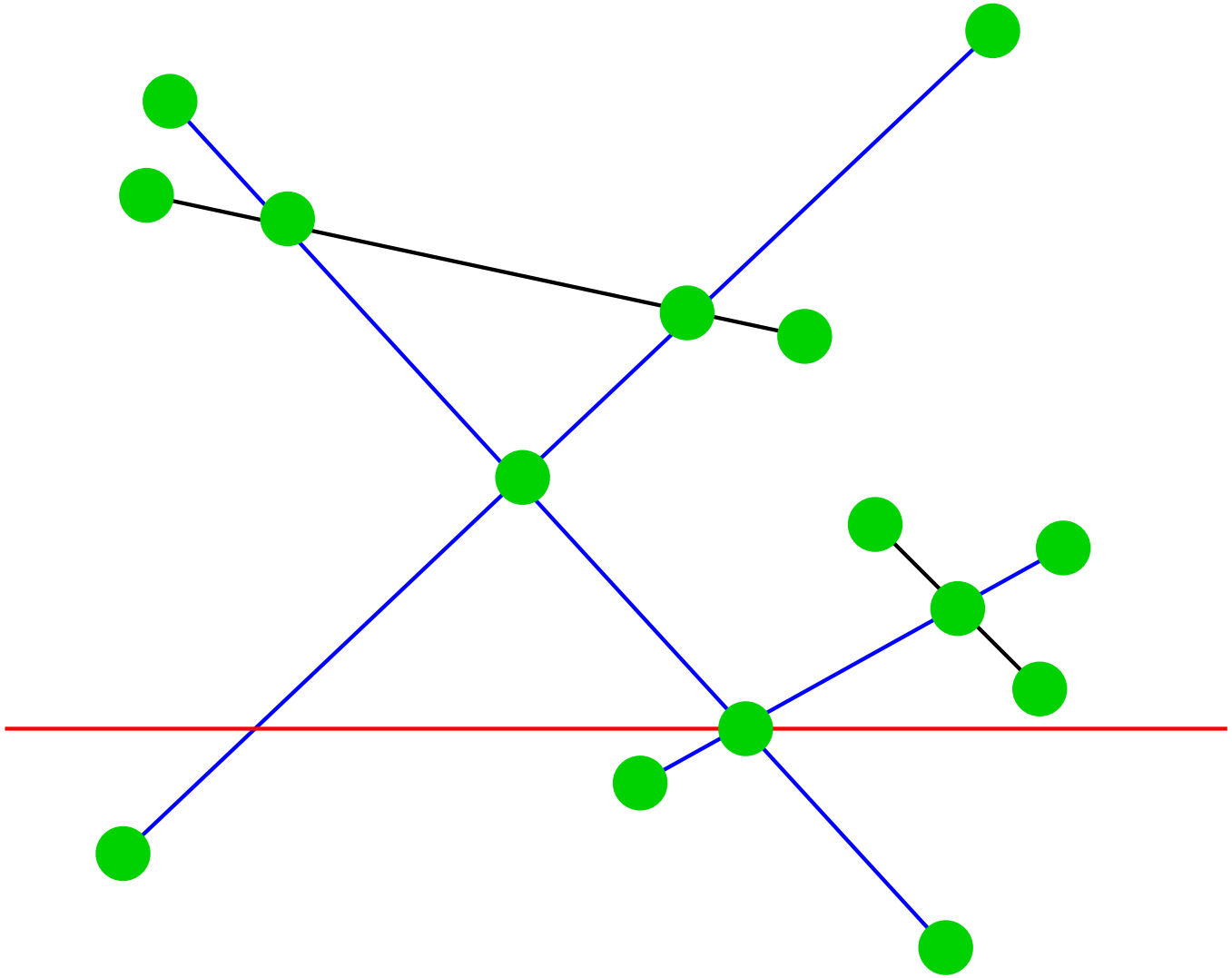
Verallgemeinerung – Beispiel



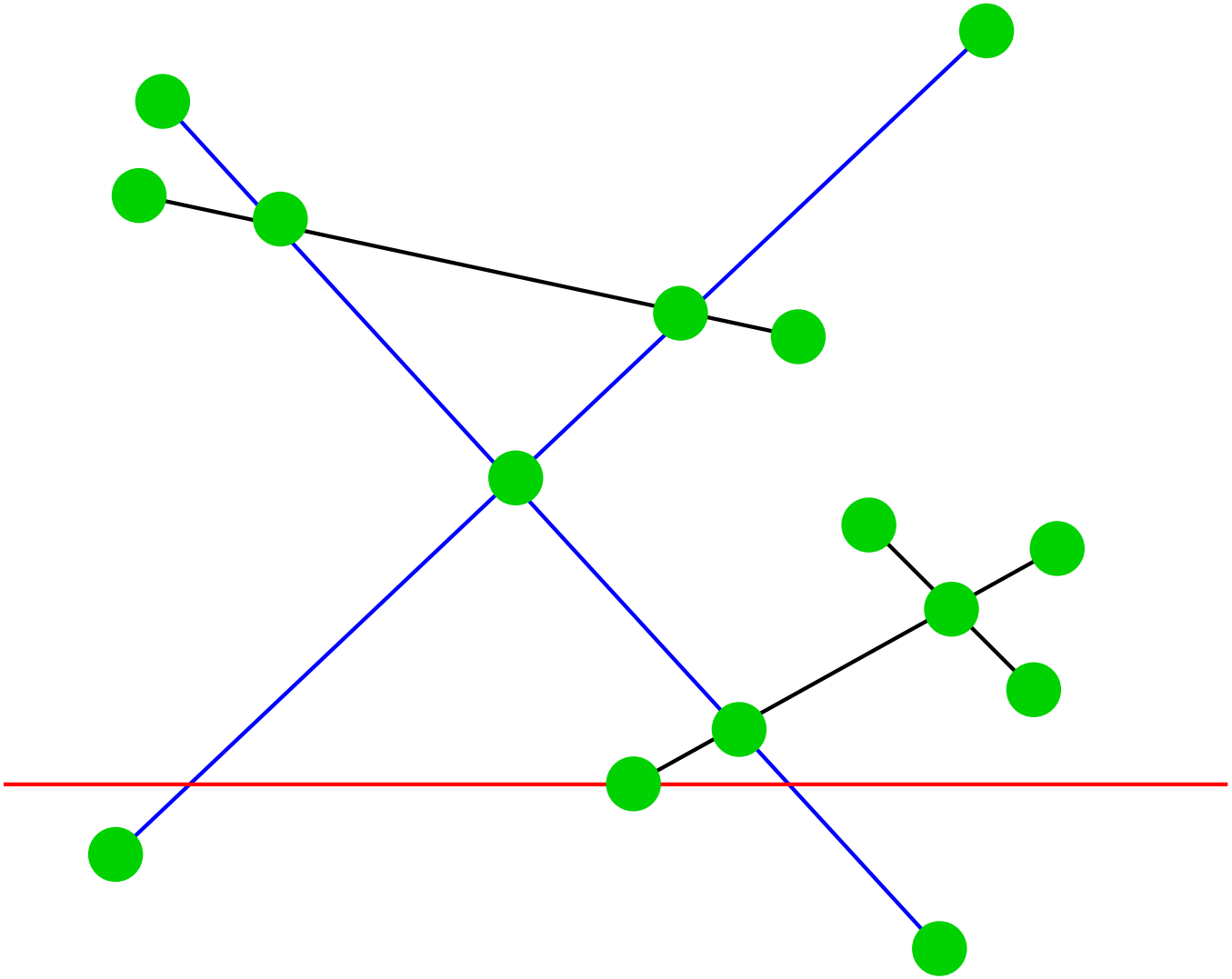
Verallgemeinerung – Beispiel



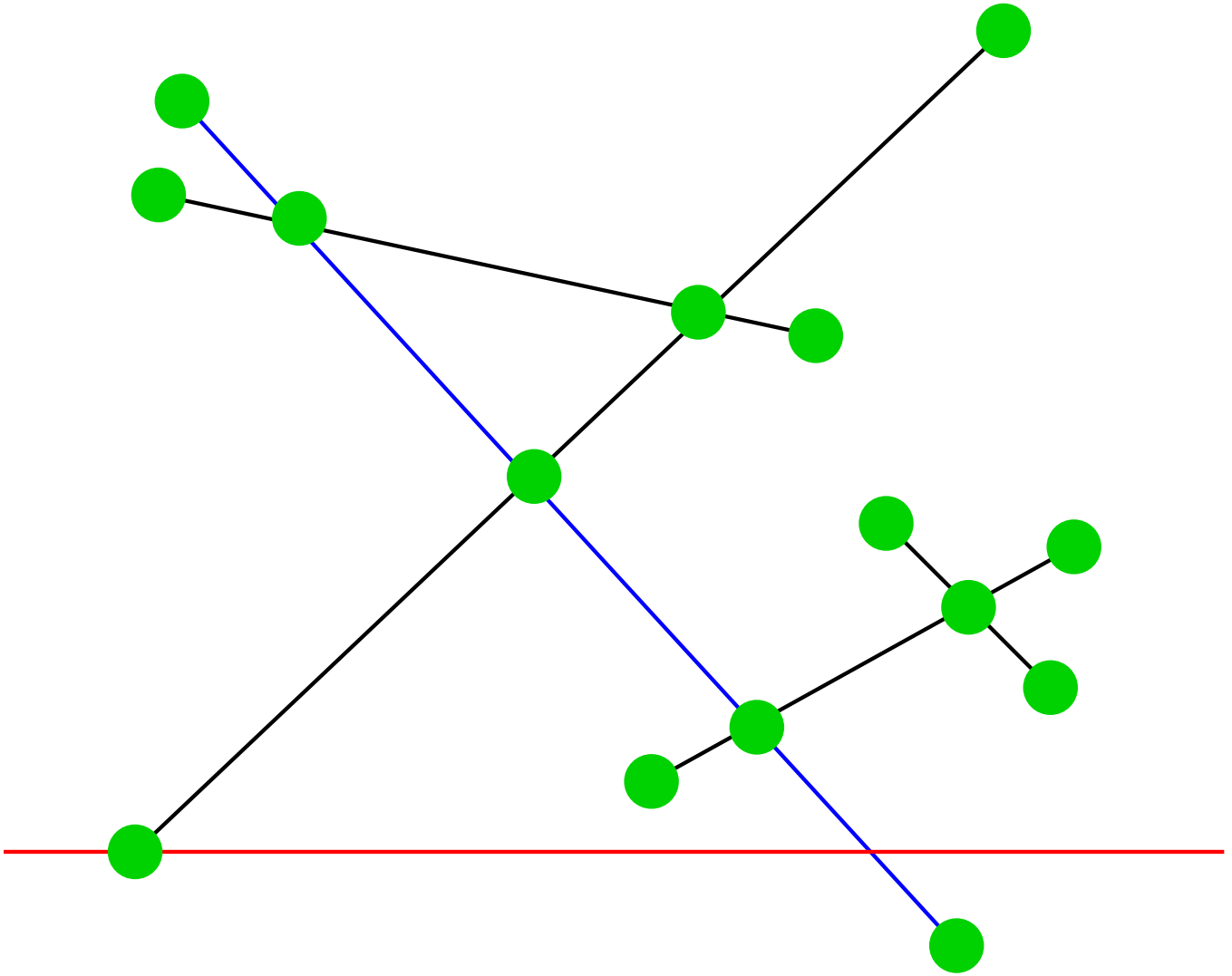
Verallgemeinerung – Beispiel



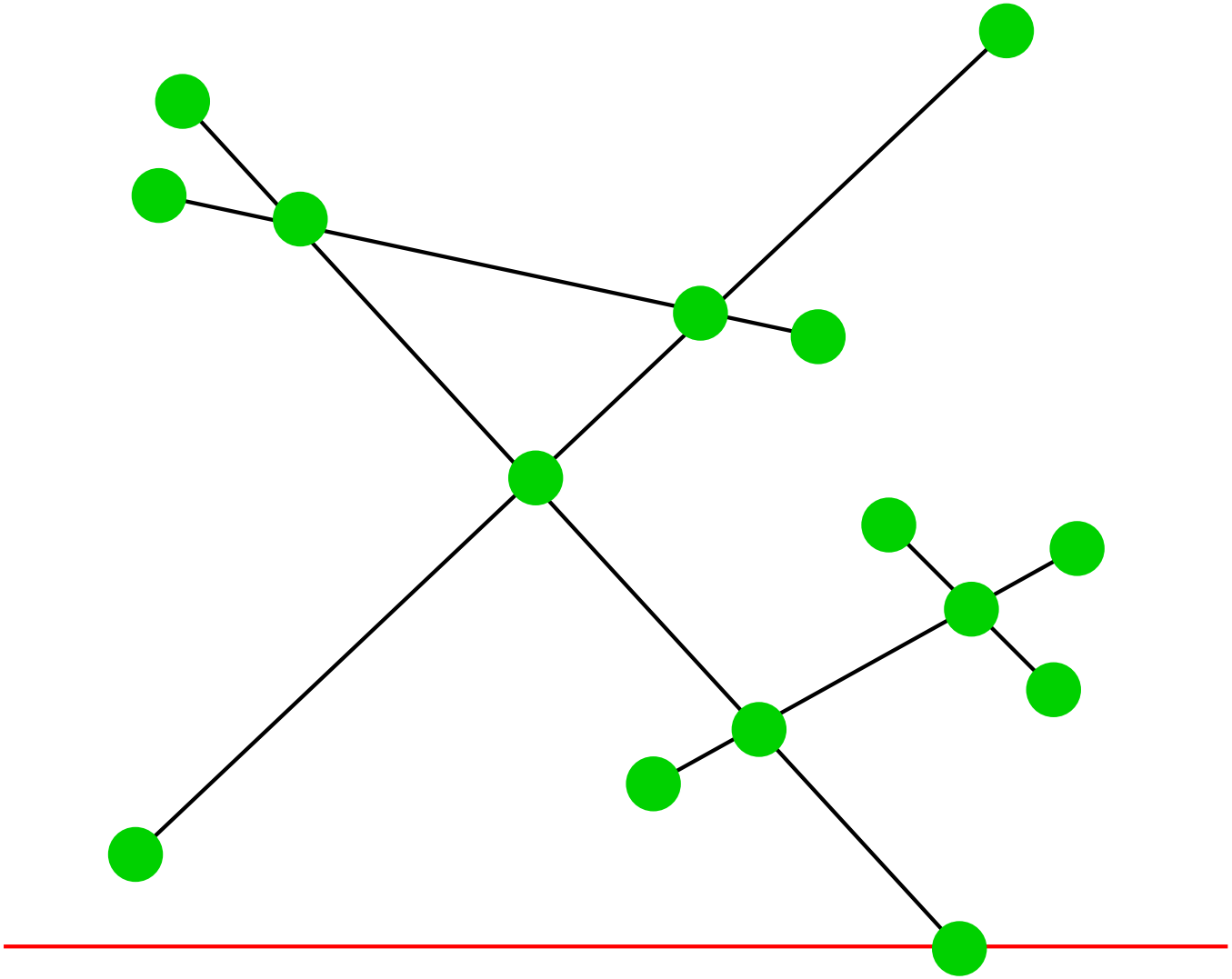
Verallgemeinerung – Beispiel



Verallgemeinerung – Beispiel



Verallgemeinerung – Beispiel



Verallgemeinerung – Analyse

Insgesamt: $\mathrm{O}((n+k)\log n)$

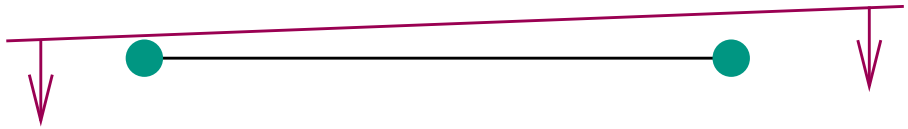
Verallgemeinerung – jetzt (fast) wirklich

Verbleibende Annahme: Keine Überlappungen

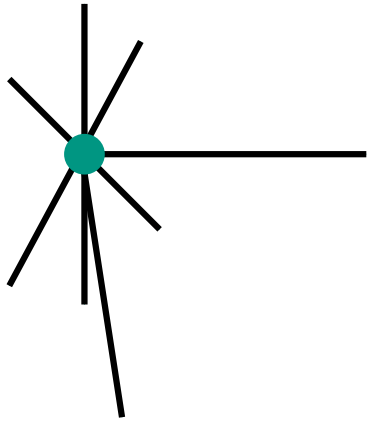
Ordnung für Q : $(x,y) \prec (x',y') \Leftrightarrow y > y' \vee y = y' \wedge x < x'$

(verquere lexikographische Ordnung)

Interpretation: infinitesimal ansteigende Sweep-Line



Q speichert Mengen von Ereignissen mit gleichem (x,y)



handleEvent$(p = (x, y))$

U:= segments starting at p // from Q

C:= segments with p in their interior // from T

L:= segments finishing at p // from Q

if $|U| + |C| + |L| \geq 2$ **then** report intersection @ p

T.remove$(L \cup C)$

T.insert$(C \cup U)$ such that order just below p is correct

if $U \cup C = \emptyset$ **then**

findNewEvent$(T$.findPred$(p), T$.findSucc$(p), p)$

else

findNewEvent$(T$.findPred$(p), T$.findLeftmost$(p), p)$

findNewEvent$(T$.findRightmost$(p), T$.findSucc$(p), p)$

findNewEvent(s,t,p)

if s and t intersect at a point $p' \succ p$ **then**

if $p' \notin Q$ **then** Q.insert(p')

Überlappungen finden

Für jede Strecke s berechne die Gerade $g(s)$, auf der s liegt

Sortiere S nach $g(s)$

1D Überlappungsproblem für jede auftretende Gerade.

Platzverbrauch

Im Moment: $\Theta(n+k)$

Reduktion auf $\mathrm{O}(n)$:

lösche Schnittpunkte zwischen nicht benachbarten Strecken aus T.

Die werden ohnehin wieder eingefügt wenn sie wieder benachbart werden.

Mehr Linienschnitt

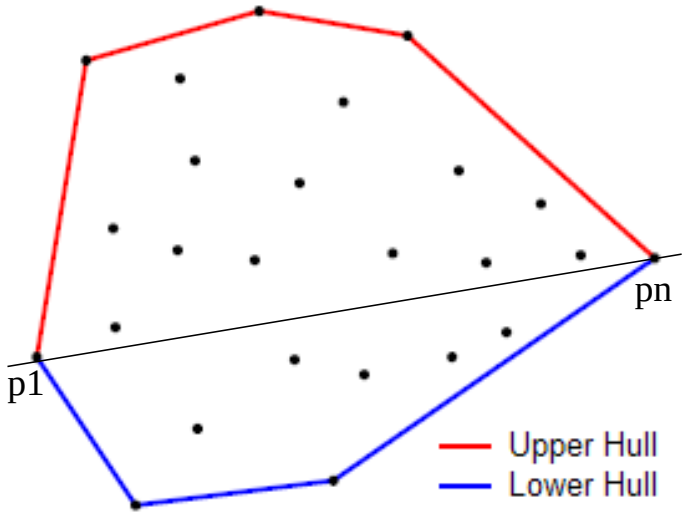
- [Bentley Ottmann 1979] Zeit $O((n+k)\log n)$
- [Chazelle Edelsbrunner 1988] Zeit $O(n\log n+k)$
- [Pach Sharir 1991] Zeit $O((n+k)\log n)$, Platz $O(n)$
- [Mulmuley 1988] erwartete Zeit $O(n\log n+k)$, Platz $O(n)$
- [Balaban 1995] Zeit $O(n\log n+k)$, Platz $O(n)$

6.2 2D Konvexe Hülle

Gegeben: Menge $P = \{p_1, \ldots, p_n\}$ von Punkten in $\mathbb{R}^2$

Gesucht: Konvexes Polygon K mit Eckpunkten aus P und $P \subseteq K$.

Wir geben einen einfachen Algorithmus, der in Zeit $O(\text{sort}(n))$ läuft.



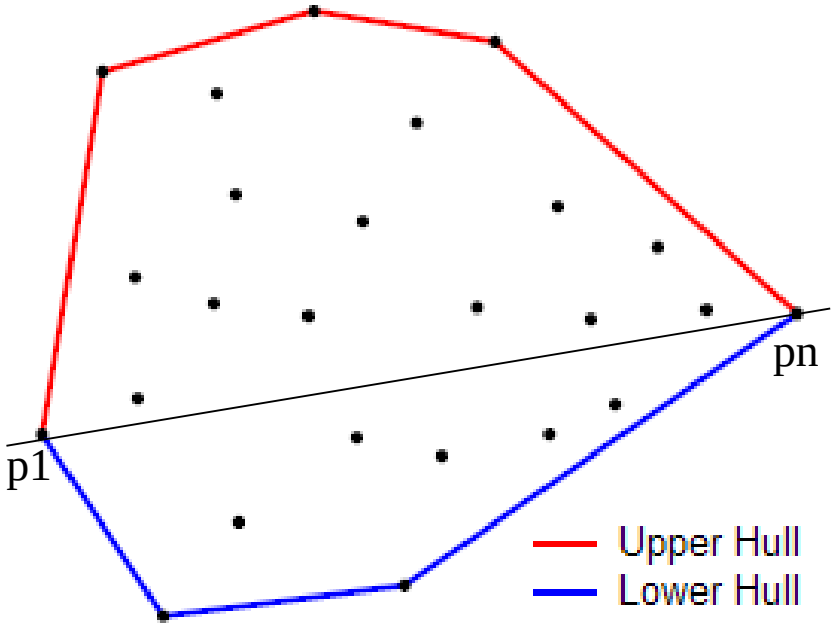
Konvexe Hülle

sortiere P lexikographisch nach (x, y), d.h., ab jetzt

$p_1 < p_2 < \cdots < p_n$

OBdA:

Wir berechnen nur die obere Hülle von Punkten oberhalb von $\overline{p_1 p_n}$



Graham's Scan [Graham 1972, Andrew 1979]

Function upperHull$(p_1, \ldots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$: Stack **of** Point

invariant L is the upper hull of $\langle p_n, p_1, \ldots, p_i \rangle$

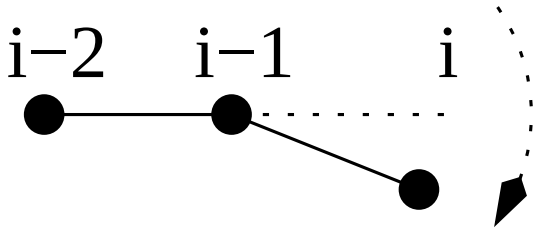
for $i := 3$ **to** n **do**

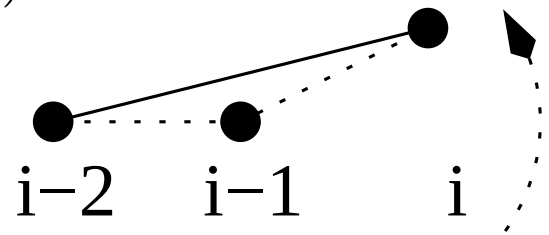
while $\neg$rightTurn$(L.\text{secondButlast}, L.\text{last}, p_i)$ **do**

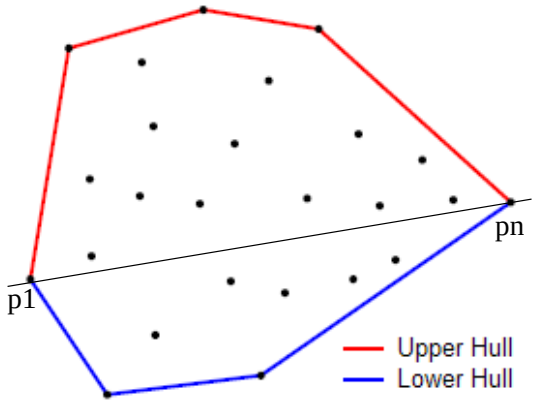
L.pop

$L := L \circ \langle p_i \rangle$

return L







Graham's Scan – Beispiel

Function upperHull$(p_1, \ldots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$: Stack **of** Point

invariant L is the upper hull of $\langle p_n, p_1, \ldots, p_i \rangle$

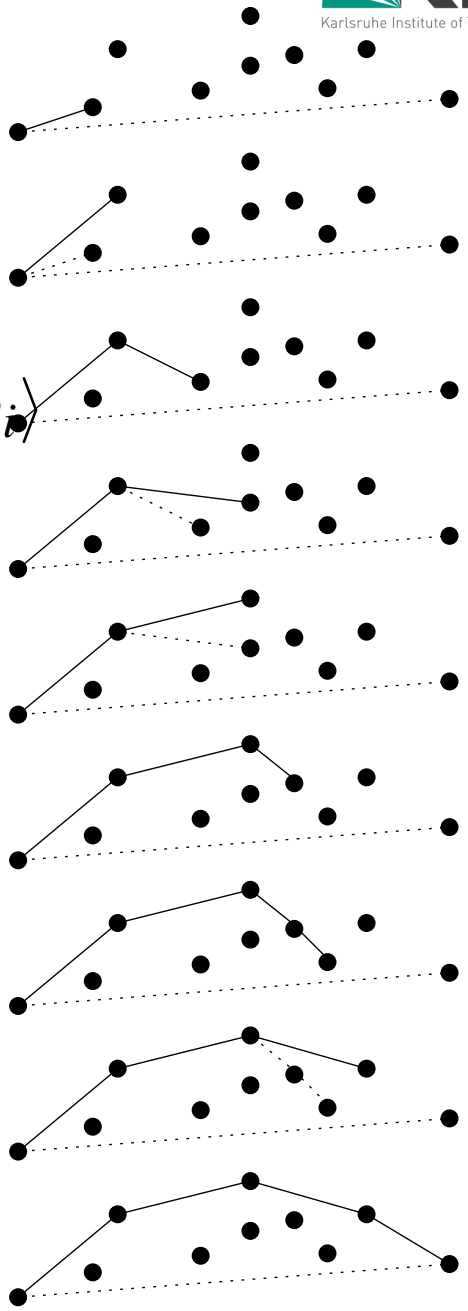
for $i := 3$ **to** n **do**

while $\neg$rightTurn$(L$.secondButlast, L.last, $p_i)$ **do**

L.pop

$L := L \circ \langle p_i \rangle$

return L



Graham's Scan – Analyse

Function upperHull$(p_1, \ldots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$: Stack **of** Point

invariant L is the upper hull of $\langle p_n, p_1, \ldots, p_i \rangle$

for $i := 3$ **to** n **do**

while $\neg$rightTurn$(L.\text{secondButlast}, L.\text{last}, p_i)$ **do**

L.pop

$L := L \circ \langle p_i \rangle$

return L

Sortieren $+\mathrm{O}(n)$

Wieviele Iterationen der While-Schleife insgesamt?

3D Konvexe Hülle

Geht in Zeit $\mathrm{O}(n \log n)$ [Preparata Hong 1977]

Konvexe Hülle, $d \geq 4$

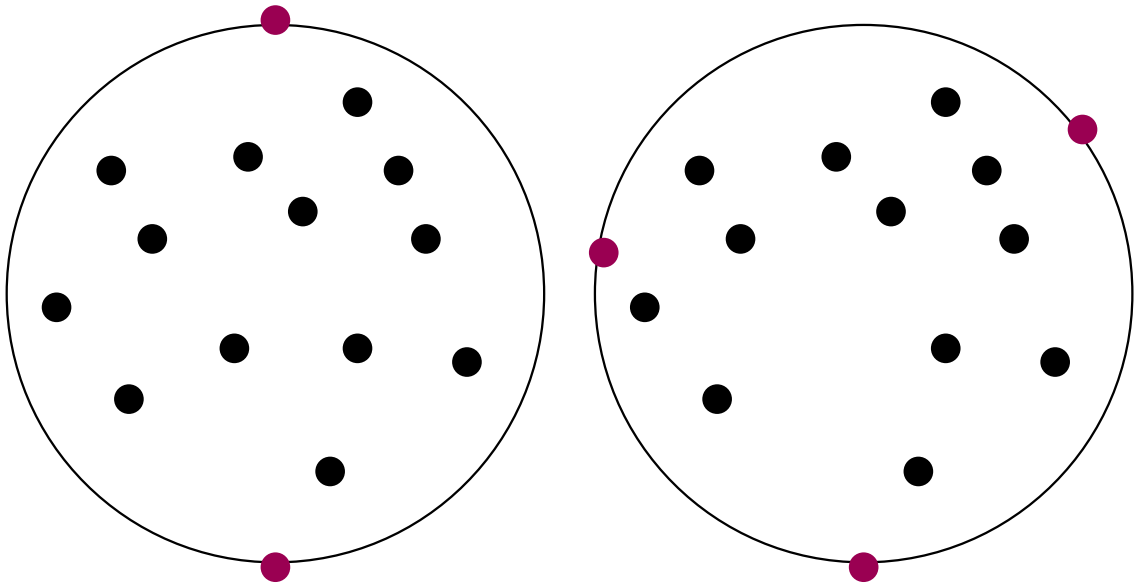
Ausgabekomplexität $\mathrm{O}\left(n^{\lfloor d/2 \rfloor}\right)$

6.3 Kleinste einschließende Kugel

Gegeben: Menge $P = \{p_1, \ldots, p_n\}$ von Punkten in $\mathbb{R}^d$

Gesucht: Kugel K mit minimalem Radius, so dass $P \subseteq K$.

Wir geben einen einfachen Algorithmus, der in erwarteter Zeit $O(n)$ läuft. [Welzl 1991].



Function smallestEnclosingBallWithPoints(P, Q)

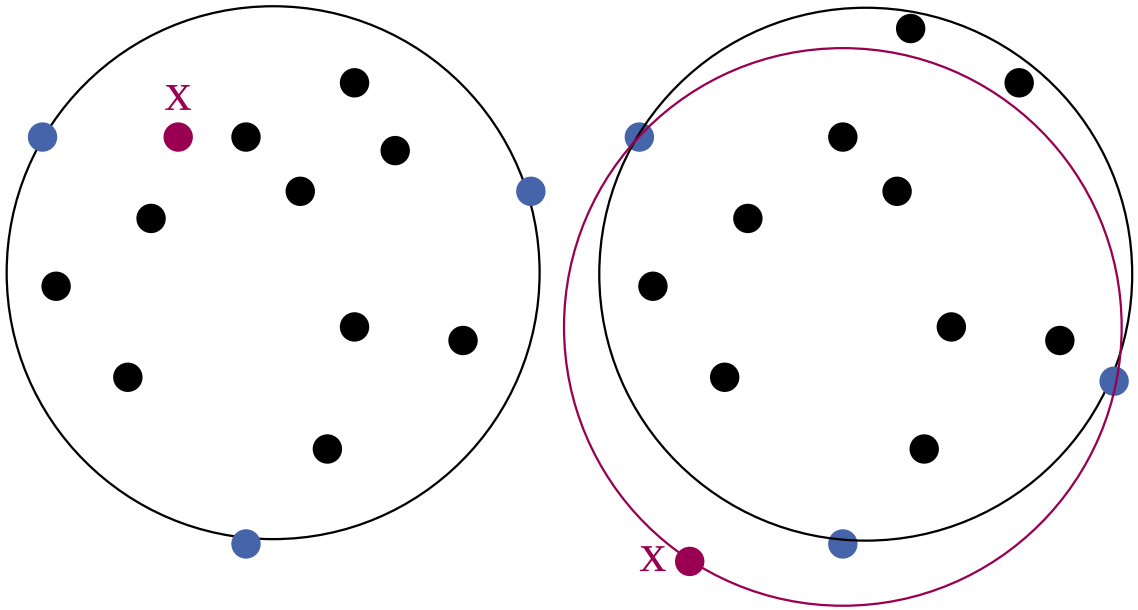
if $|P| = 0 \vee |Q| = d+1$ **then return** ball(Q)

pick random $x \in P$

B:= smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q)$

if $x \in B$ **then return** B

return smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q \cup \{x\})$



Kleinste einschließende Kugel – Korrektheit

Function smallestEnclosingBallWithPoints(P, Q)

  if $|P| = 1 \vee |Q| = d+1$ **then return** ball(Q)

  pick random $x \in P$

  B:= smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q)$

  if $x \in B$ **then return** B

  return smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q \cup \{x\})$

z.Z.: $x \notin B \rightarrow x$ ist auf dem Rand von sEB(P)

Wir zeigen Kontraposition:

x nicht auf dem Rand von sEB(P)

$\longrightarrow$ sEB(P) = sEB$(P \setminus \{x\}) = B$ z.Z.: sEBs sind eindeutig!

Also $x \in B$

Lemma: sEB(P) ist eindeutig bestimmt.

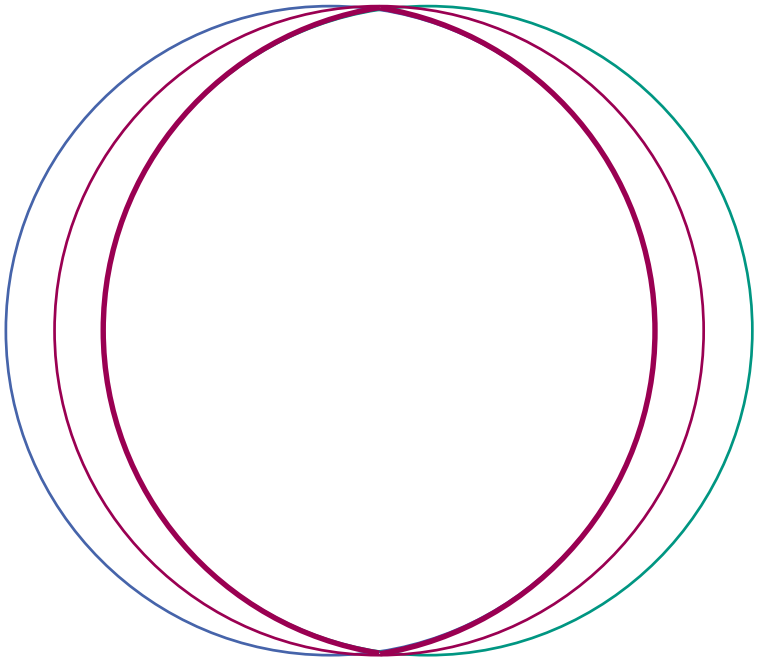
Beweis: Annahme, $\exists$sEBs $B_1 \neq B_2$

$\longrightarrow P \subseteq B_1 \wedge P \subseteq B_2$

$\longrightarrow P \subseteq B_1 \cap B_2 \subseteq \text{sEB}(B_1 \cap B_2) =: B$

Aber dann ist radius$(B) <$ radius(B_1)

Widerspruch zur Annahme, dass B_1 eine sEB ist. $\square$



Kleinste einschließende Kugel – Analyse

Wir zählen die erwartete Anzahl der Tests $x \in B$, $T(p,q)$.

$$T(p,d+1) = T(1,p) = 0 \qquad \text{Basis der Rekurrenz}$$

$$\begin{aligned} T(p,q) &\leq 1 + T(p-1,q) + \mathbb{P}[x \notin B]\, T(p,q+1) \\ &\leq 1 + T(p-1,q) + \frac{d+1-q}{p} T(p,q+1) \end{aligned}$$

Kleinste einschließende Kugel – Analyse, $d = 2$

$T(p, d+1) = T(1, p) = 0$

$$T(p,q) \leq 1 + T(p-1,q) + \frac{d+1-q}{p} T(p,q+1)$$

$$T(p,2) \leq 1 + T(p-1,2) + \frac{1}{p} T(p,3) \leq 1 + T(p-1,2) \leq p$$

$$T(p,1) \leq 1 + T(p-1,1) + \frac{2}{p} T(p,2)$$

$$\leq 1 + T(p-1,1) + \frac{2}{p} p = 3 + T(p-1,1) \leq 3p$$

$$T(p,0) \leq 1 + T(p-1,0) + \frac{3}{p} T(p,1)$$

$$\leq 1 + T(p-1,0) + \frac{3}{p} 3p = 10 + T(p-1,0) \leq 10p$$

Kleinste einschließende Kugel – Analyse

d	$T(p,0)$
1	$3n$
2	$10n$
3	$41n$
4	$206n$

Allgemein $T(p,0) \geq d!n$

Karlsruhe Institute of Technology

Ähnliche Randomisierte Linearzeitalgorithmen

- Lineare Programmierung mit konstantem d [Seidel 1991]
- Kleinster einschließender Ellipsoid, Kreisring,. . .
- Support-Vector-Machines (maschinelles Lernen)
- Alles wo (LP-type problem [Sharir Welzl 1992])
  - $O(1)$ Objekte das Optimum festlegen
  - Objekt x hinzufügen

    $\longrightarrow$ Lösung bleibt gleich oder ist an Lösungsdef. beteiligt

6.4 2D Bereichssuche (range search)

Daten: $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^2$

Anfragen: achsenparallele Rechtecke $Q = [x, x'] \times [y, y']$

finde $P \cap Q$ (range reporting)

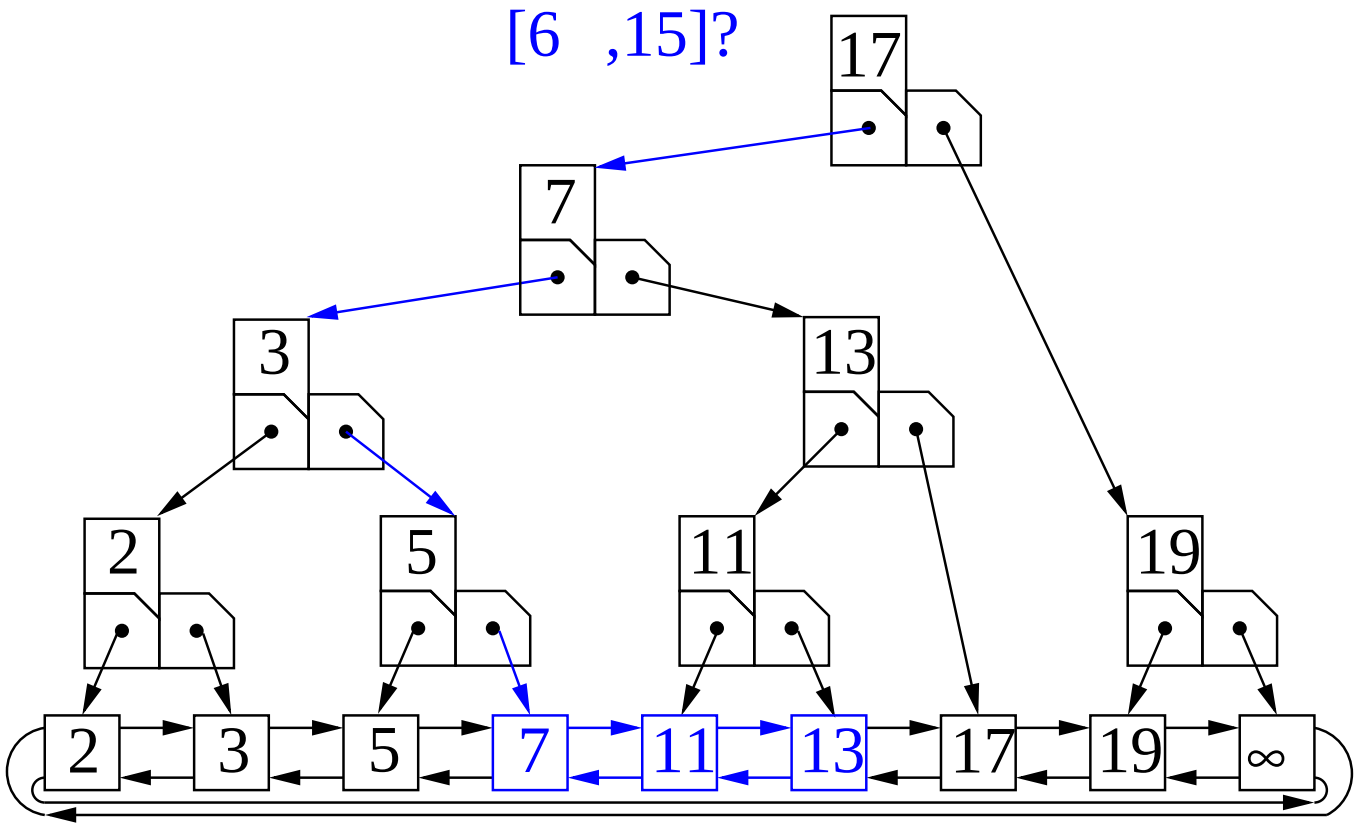
oder $k = |P \cap Q|$ (range counting)

Vorverarbeitung erlaubt.

- ☐ Vorverarbeitungszeit? $O(n \log n)$
- ☐ Platz? $O(n)$?
- ☐ Anfragebearbeitung?
  - Counting: $O(\log n)$
  - Reporting: $O(k + \log n)$ oder wenigstens $O(k \cdot \log n)$

1D Bereichssuche

Suchbaum



Zählanfragen: Teilbaumgrößen speichern

Sogar dynamisch !

Reduktion auf $1..n \times 1..n$

vereinfachende Annahme: Koordinaten paarweise verschieden.

Ersetze Koordinaten $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ durch ihren Rang:

$x_i \rightarrow$ Rang von x_i in $\{x_1, \ldots, x_n\}$

$y_i \rightarrow$ Rang von y_i in $\{y_1, \ldots, y_n\}$

Reduktion auf $1..n \times 1..n$

Ersetze Koordinaten $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ durch ihren Rang:

P_x:= sort$(\langle x_1, \ldots, x_n \rangle)$; P_y:= sort$(\langle y_1, \ldots, y_n \rangle)$

P:= $\{(\text{binarySearch}(x, P_x), \text{binarySearch}(y, P_y)) : (x, y) \in P\}$

Function rangeQuery$([x, x'] \times [y, y'])$

x:= binarySearchSucc(x, P_x); x':= binarySearchPred(x', P_x)

y:= binarySearchSucc(y, P_y); y':= binarySearchPred(y', P_y)

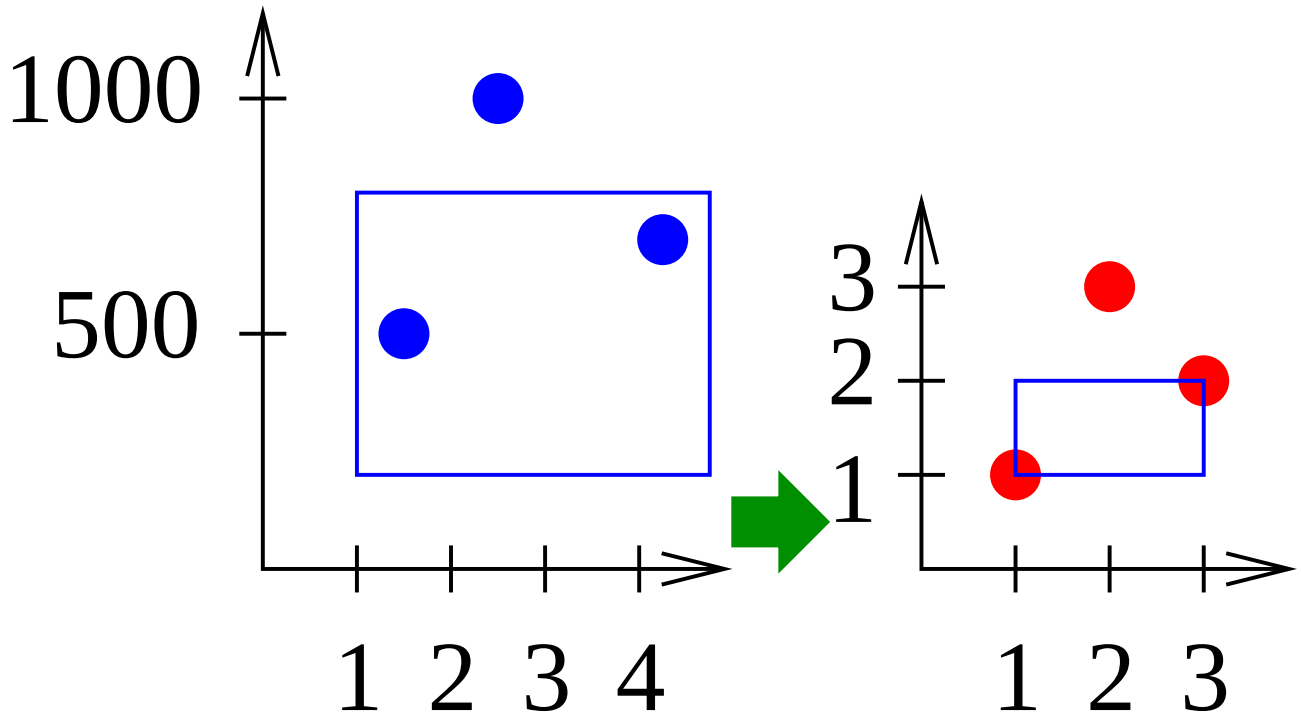
R:= intRangeQuery$([x, x'] \times [y, y'])$

return $\{(P_x[x], P_y[y]) : (x, y) \in R\}$

Zeit $O(n \log n)$

Beispiel

$\{(2.5,1000),(1.4,500),(4.2,700)\} \rightarrow \{(2,3),(1,1),(3,2)\}$



Wavelet Tree

[Chazelle 1988, Grossi/Gupta/Vitter 2003, Mäkinen/Navarro 2007]

Class WaveletTree($X = \langle x_1, \ldots, x_n \rangle$)// represents $(x_1, 1), \ldots, (x_n, n)$

//Constructor:

if $n < n_0$ **then** store X directly

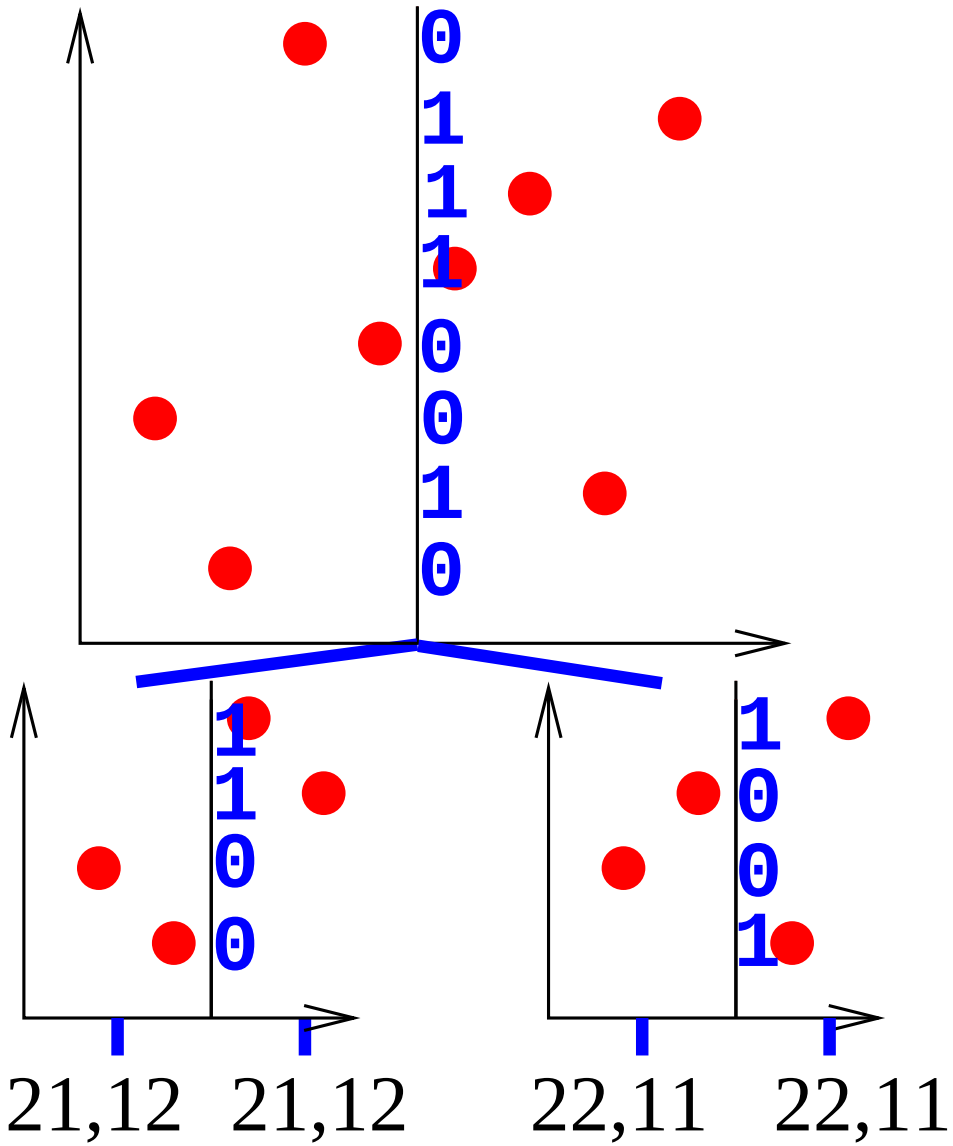
else

store bitvector b with $b[i] = 1$ iff $x_i > \lfloor n/2 \rfloor$

ℓ:= WaveletTree($\langle x_i : x_i \leq \lfloor n/2 \rfloor \rangle$)

r:= WaveletTree($\langle x_i - \lfloor n/2 \rfloor : x_i > \lfloor n/2 \rfloor \rangle$)

Beispiel



Wavelet Tree Counting Query

Function intRangeCount$([x,x'] \times [y,y'])$

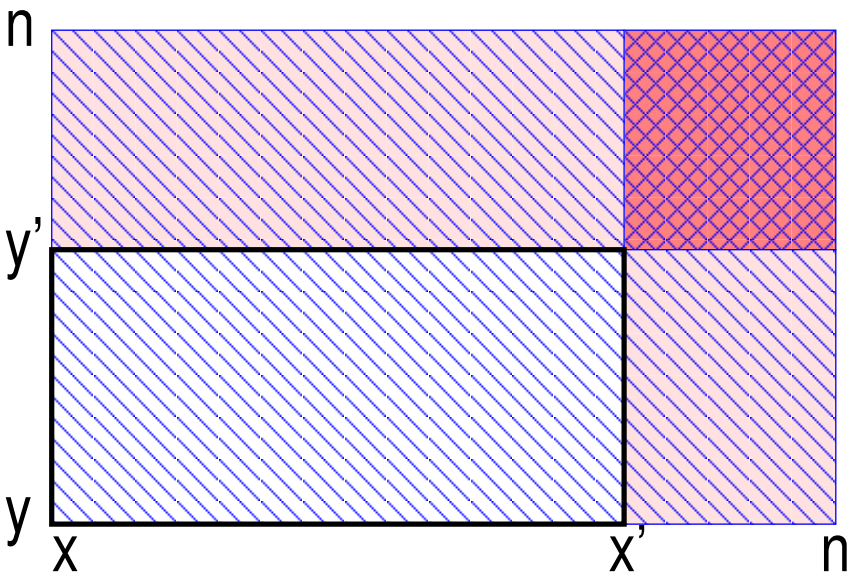
return

intDominanceCount$(x,y)-$

intDominanceCount$(x',y)-$

intDominanceCount$(x,y')+$

intDominanceCount(x',y')



Wavelet Tree Dominance Counting Query

Function intDominanceCount(x, y) // $|[x,n] \times [y,n] \cap P|$

  if $n \leq n_0$ **then return** $|[x,n] \times [y,n] \cap P|$ // brute force

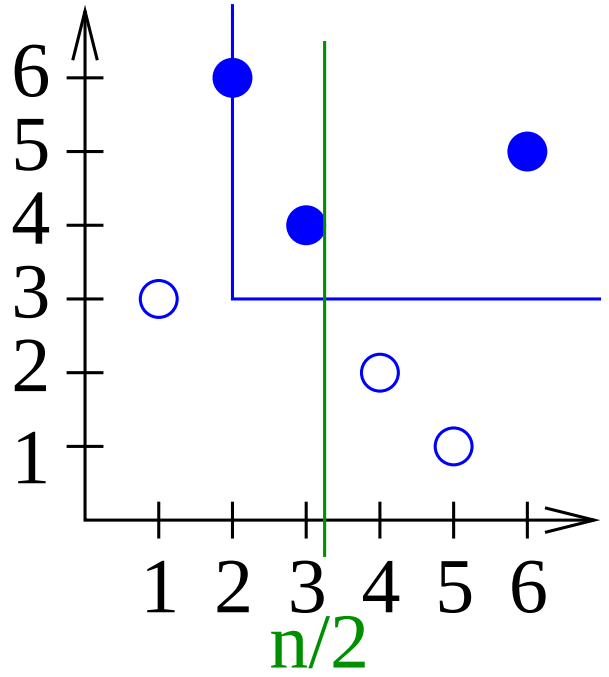
  y_r:= b.rank(y) // Number of els $\leq y$ in right half

  if $x \leq \lfloor n/2 \rfloor$ **then**

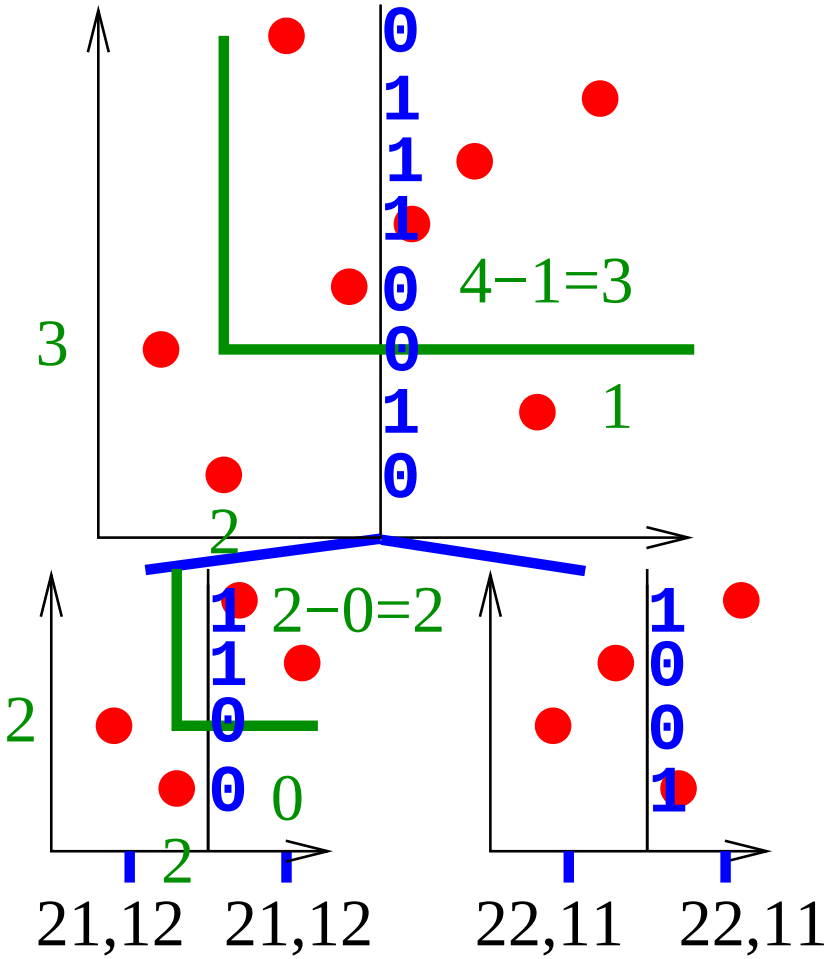
    return ℓ.intDominanceCount$(x, y - y_r) + \lceil n/2 \rceil - y_r$

  else

    return r.intDominanceCount$(x - \lfloor n/2 \rfloor, y_r)$



Beispiel



Analyse

Nur ein rekursiver Aufruf.

Rekursionstiefe $\mathrm{O}(\log n)$.

rank in konstanter Zeit (s.u.)

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

Wavelet Tree Dominance Reporting Query

Function intDominanceReporting(x, y) // $[x,n] \times [y,n] \cap P$

  if $n \leq n_0$ **then return** $[x,n] \times [y,n] \cap P$ // brute force

  $R := \emptyset$ // Result

  $y_r := b$.rank(y) // Number of els $\leq y$ in right half

  if $x \leq \lfloor n/2 \rfloor$ **then** // Both halves interesting

    if $y - y_r < \frac{n}{2}$ **then** $R := R \cup \ell$.intDominanceReporting$(x, y - y_r)$

    if $y_r < \frac{n}{2}$ **then** $R := R \cup r$.oneSidedReporting(y_r)

  else if $y_r < \frac{n}{2}$ **then** $R := R \cup r$.intDominanceReporting$(x - \lfloor n/2 \rfloor, y_r)$

  return R

Function oneSidedReporting(y) // $[1,n] \times [y,n] \cap P$

  if $n \leq n_0$ **then return** $[1,n] \times [y,n] \cap P$ // brute force

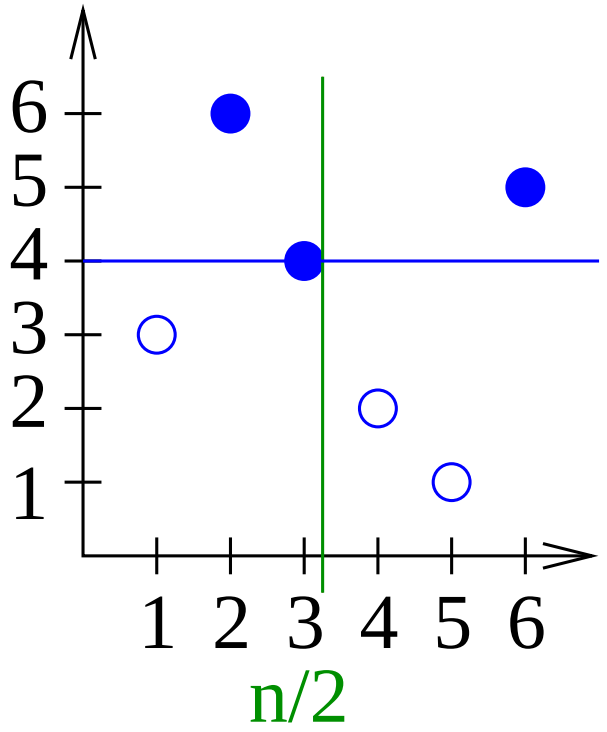
  $y_r := b$.rank(y) // Number of els $\leq y$ in right half

  $R := \emptyset$

  if $y_r < \frac{n}{2}$ **then** $R := R \cup r$.oneSidedReporting(y_r)

  if $y - y_r < \frac{n}{2}$ **then** $R := R \cup \ell$.oneSidedReporting$(y - y_r)$

  return R



Analyse

Rekurrenz

$T(n_0,0) = \mathrm{O}(1)$

$T(n,0) = T(n/2,0) + O(1) \Longrightarrow T(n,0) = \mathrm{O}(\log n)$

$T(n,k) = T(n/2,k') + T(n/2,k-k') + O(1)$ also

$T(n,k) \leq ck' \log n + c(k-k')\log n + c = c(1+k\log n) =$

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

Zeit $\mathrm{O}(k+\log n)$ braucht zusätzlichen Faktor $\log n$ Platz.

Z.B. komplette Listen auf allen Ebenen speichern

Übungsaufgabe?

Allgemeine Reporting Query

4-seitig

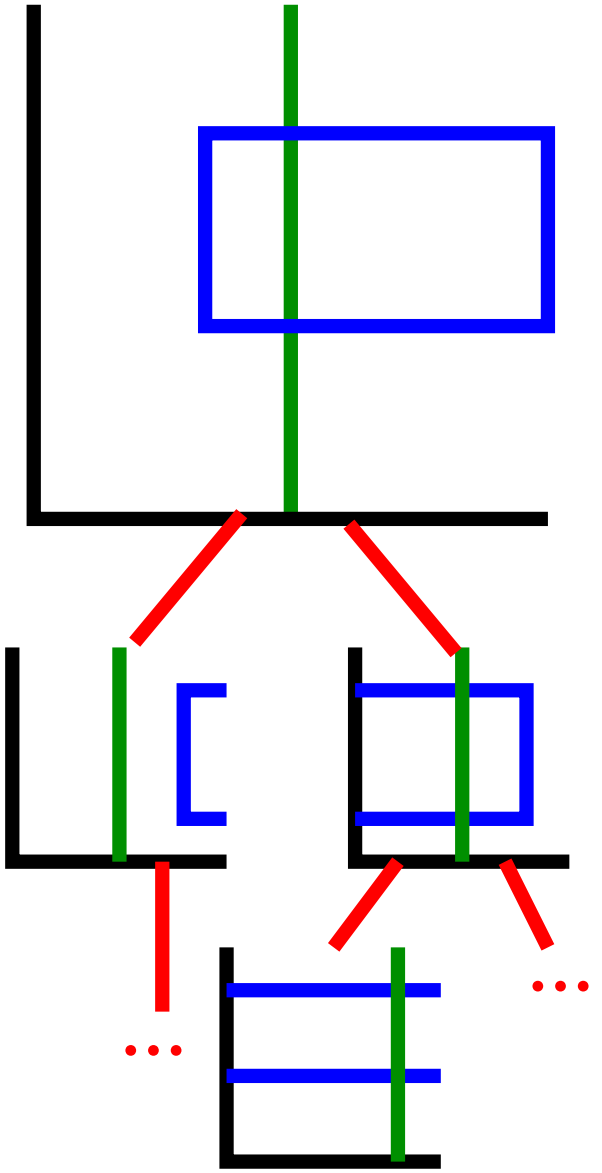
$\rightsquigarrow$

3-seitig (2 Varianten)

$\rightsquigarrow$

y-range

Analog oneSidedReporting (zwei Ranks statt einem)



Bitvektoren v mit rank in $O(1)$

Wähle $B = \Theta(\log n)$.

Vorberechnung bRank$[i]$:= v.rank(iB)

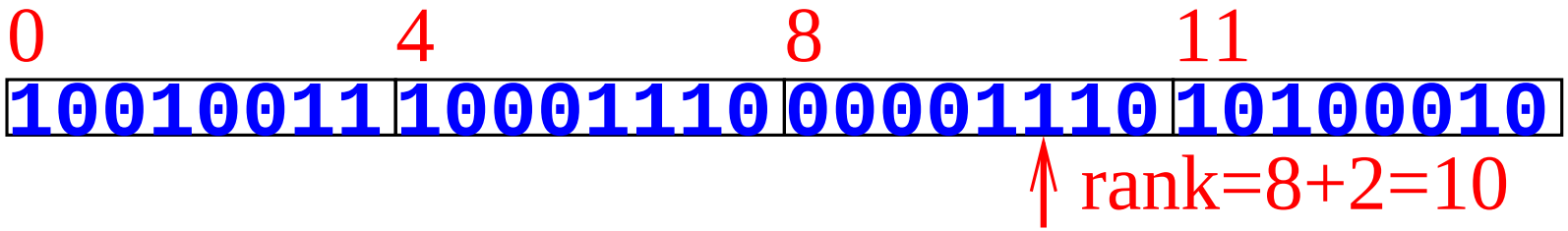
Zeit $O(n)$, Platz $O(n)$ bits

Reduktion auf logarithmische Eingabegröße:

Function rank(j) **return** bRank$[j$ **div** $B] +$ rank$(v[B(j$ **div** $B)..j])$

Logarithmische Größe:

Maschinenbefehl (population count) oder Tabellenzugriff (z.B. Größe $\sqrt{n}$ Zahlen)

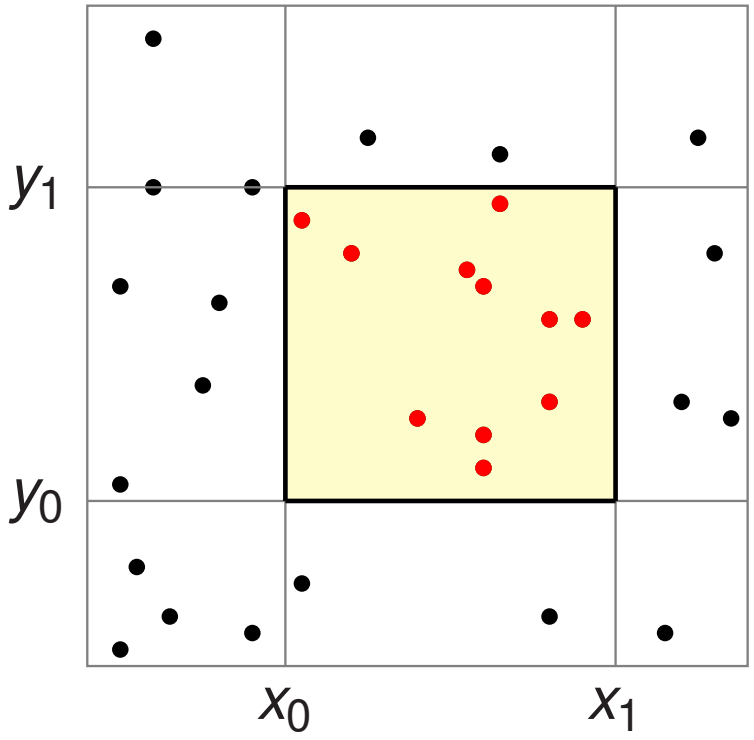


Mehr zu Bitvektoren

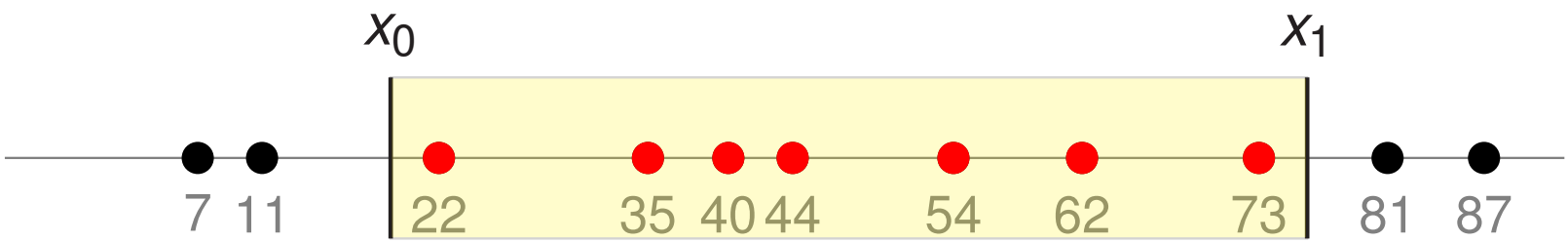
- weitere wichtige Operation $b.\text{select}(i) :=$ Position des i-ten 1-bits. Ebenfalls $\mathrm{O}(1)$
- Informationstheoretisch asympt. optimaler Platz $n + o(n)$ bits möglich.
- Grundlage für weitere succinct data structures
- Bespiel: Baum mit Platz $2n + o(n)$ bits und Navigation in konstanter Zeit.

Orthogonal range searching

Classical OLAP queries: „Find all users aged between 30 and 35 who are connected to at least 100 and at most 200 other users”



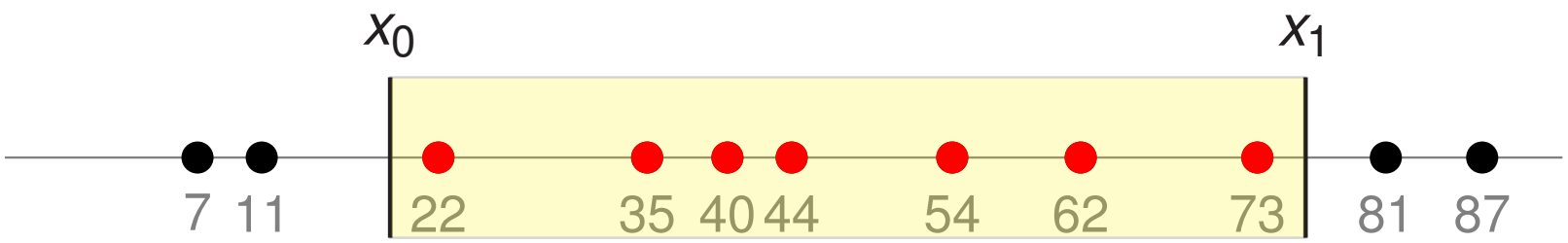
Orthogonal range searching – 1D



One dimensional case ($d = 1$). Example ($x_0 = 19$, $x_1 = 76$):

- count(x_0, x_1)=7
- report(x_0, x_1)=$\{22, 35, 40, 44, 54, 62, 73\}$

Orthogonal range searching – 1D

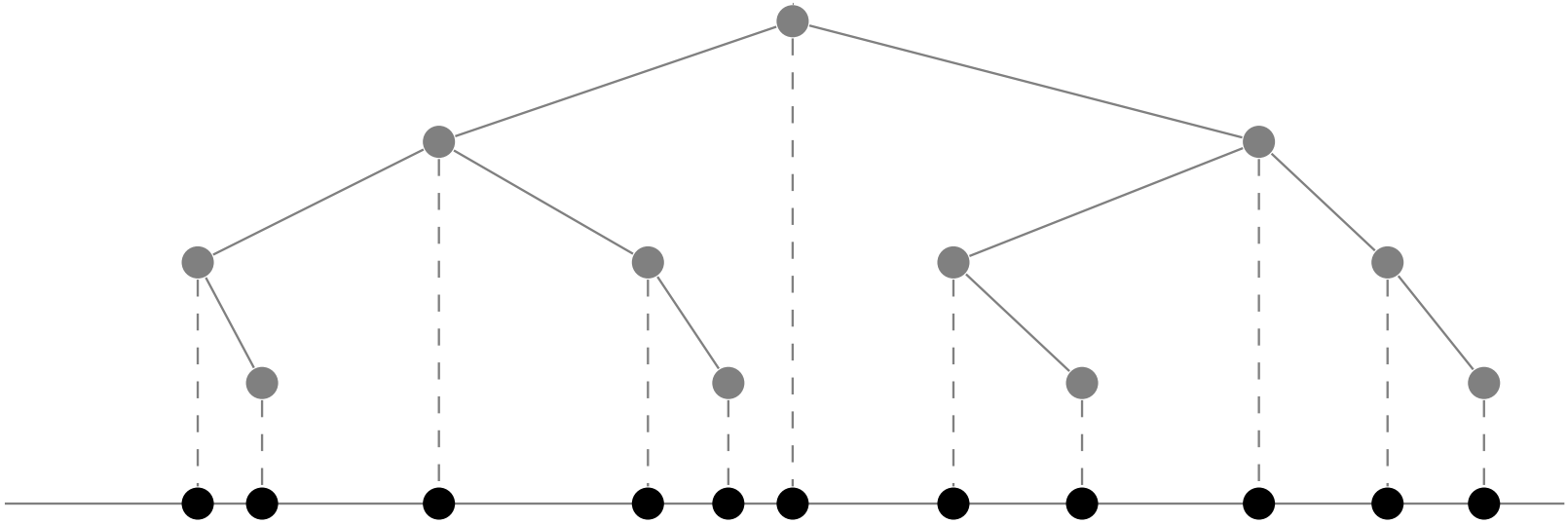


Simple solution

- Sort points according to x-coordinates ($O(n \log n)$) and store them in array A
- Calculate successor x_0' of x_0 and predecessor x_1' of x_1
- Let $i'\,(j')$ be the index of x_0' (x_1') in A
- Method *count* returns $k = j' - i' + 1$ (in $O(\log n)$ time)
- Method *report* returns subarray $A[i', j']$ (in $O(\log n + k)$ time)

Orthogonal range searching – 1D

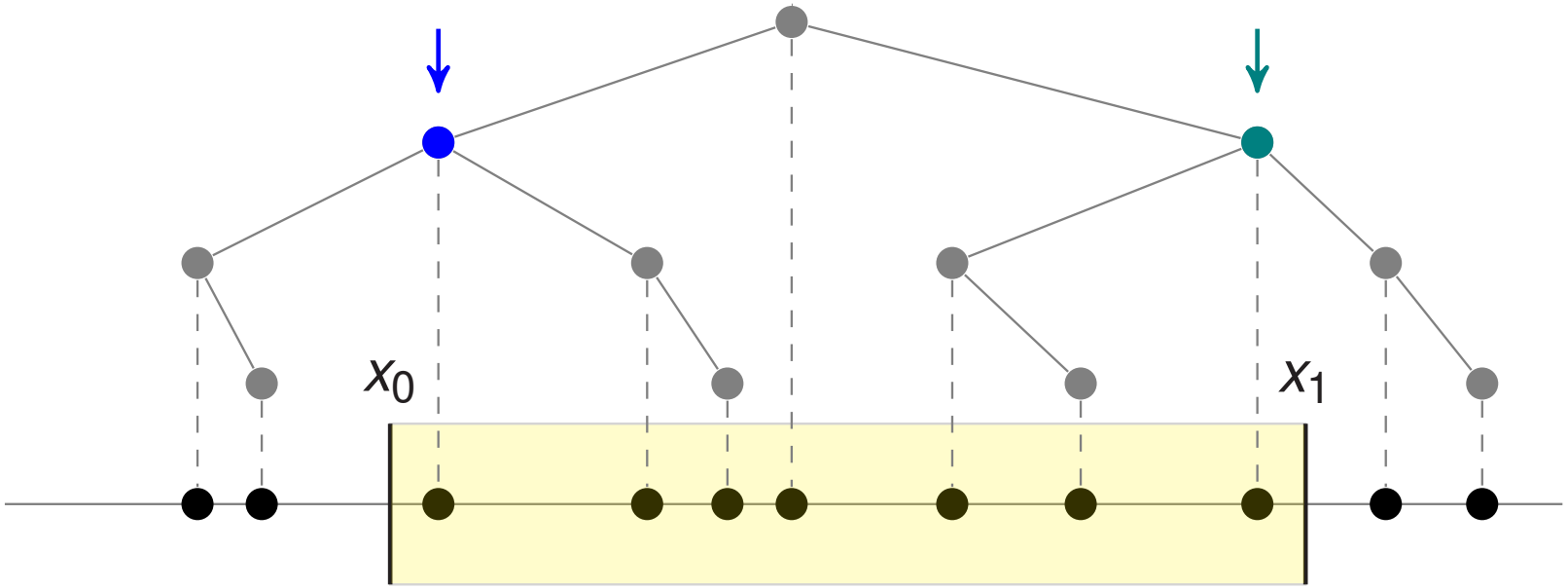
Alternative solution: balanced binary search trees



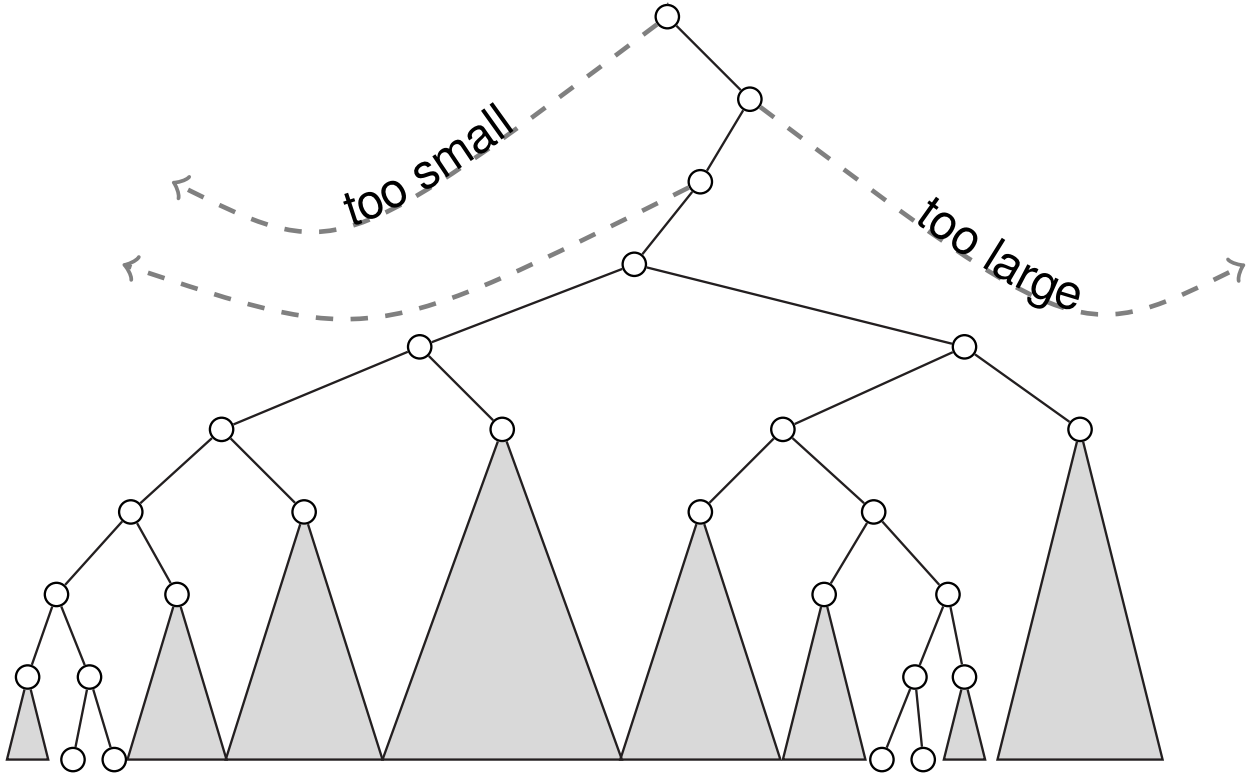
- Find point in middle, split set and recurse on both half (pick left point if set size is even)
- Depth is $\log n$, construction time is bounded by sorting ($O(n \log n)$)

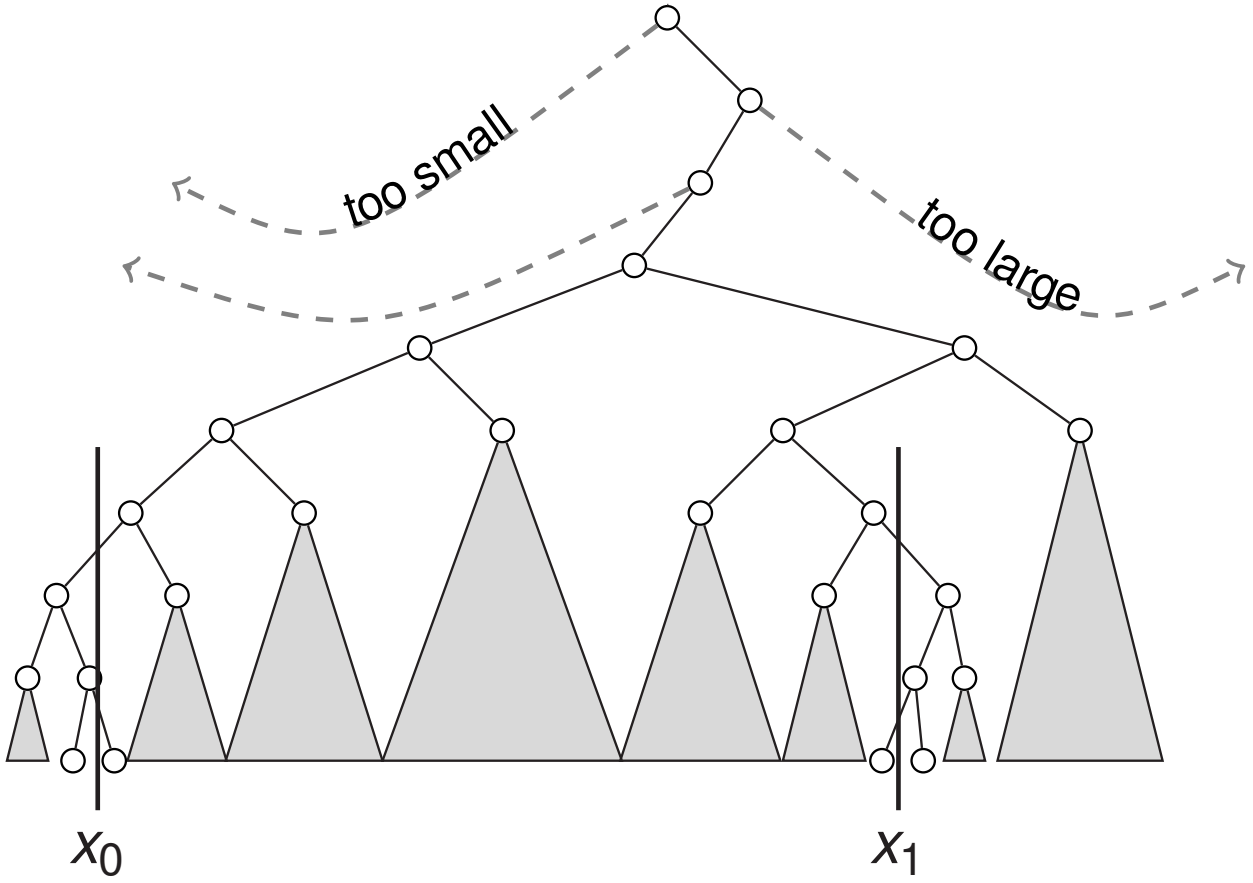
Orthogonal range searching – 1D

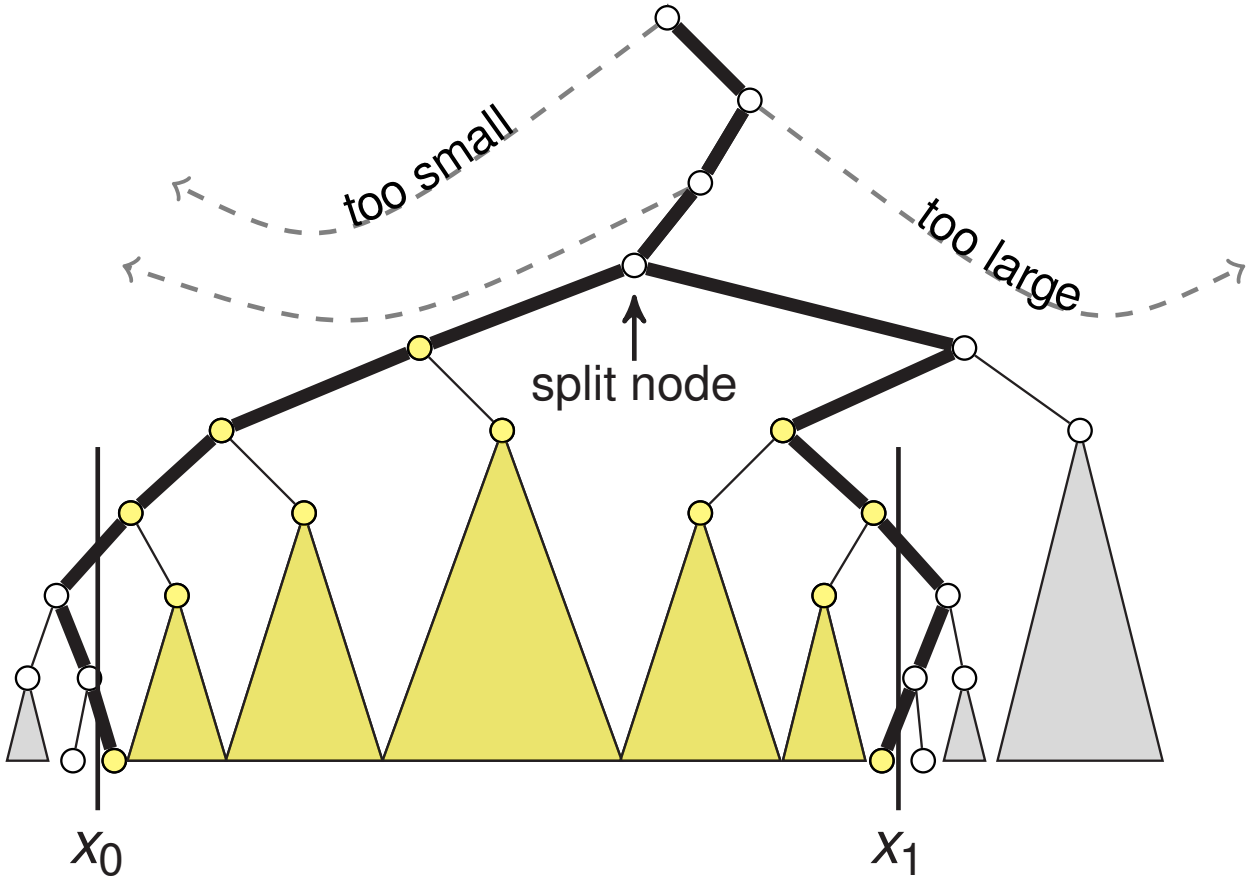
Alternative solution: balanced binary search trees

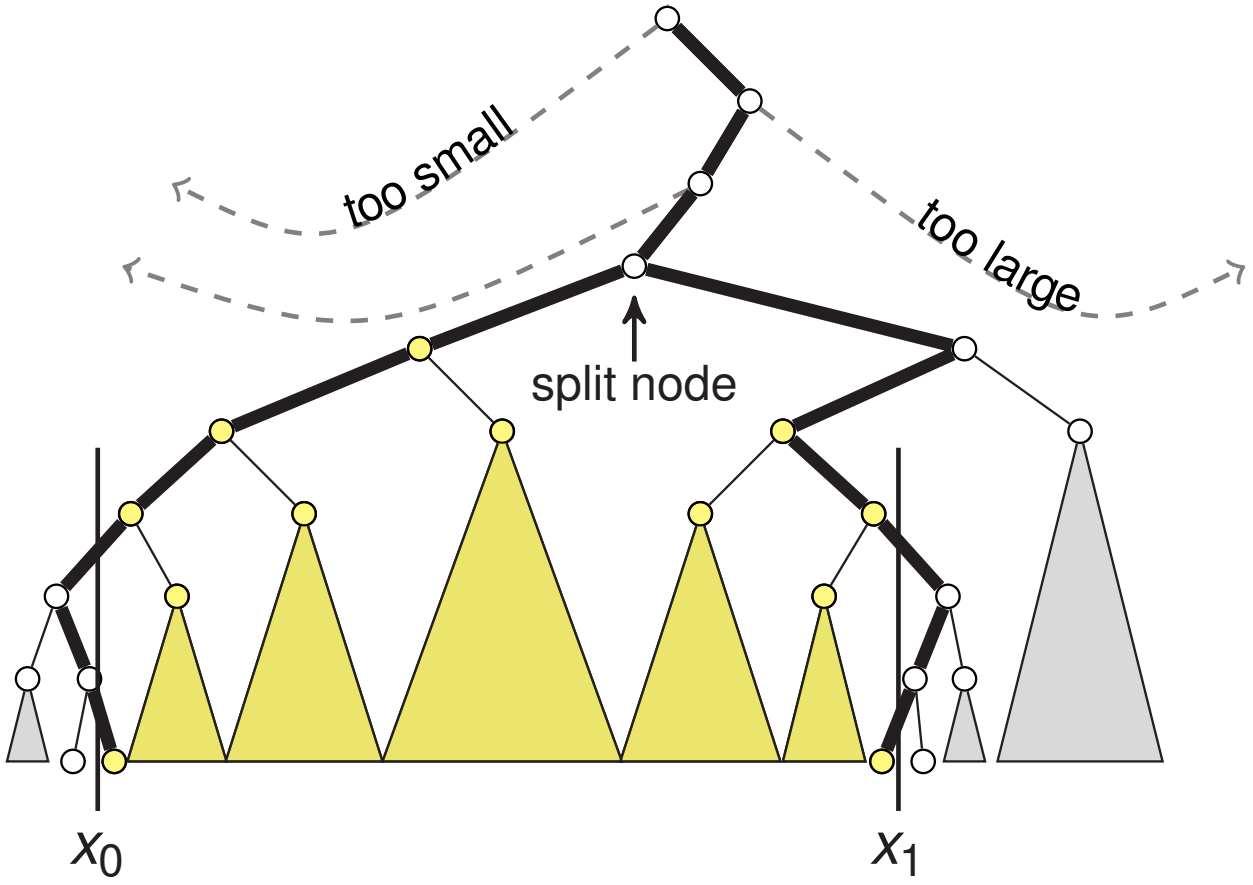


- Find successor (predecessor) of x_0 (x_1) again


too small
too large

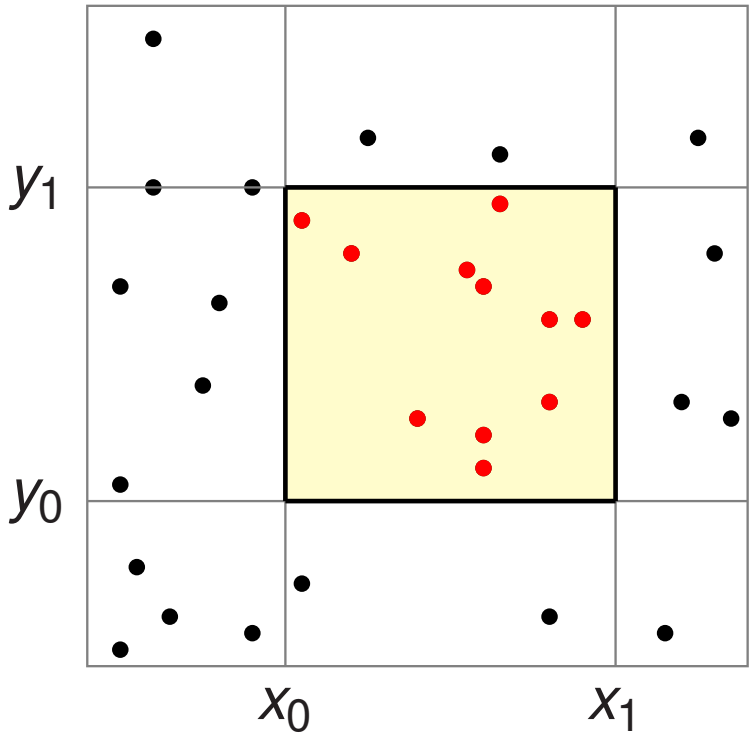

too small
too large
x_0
x_1


too small
too large
split node
x_0
x_1



- Subtrees of off-path edges are either included or excluded form result
- Result can be implicitly represented using included off-path subtrees (there are at most $O(\log n)$ of them)

Orthogonal range searching – 2D



Two dimensional case ($d = 2$). Example ($x_0 = 12$, $x_1 = 32$, $y_0 = 10$,$y_1 = 29$):

- $count(x_0, x_1, y_0, y_1) = 11$
- $report(x_0, x_1, y_0, y_1) = (19, 40), (23, 39), (22, 49), \cdots$

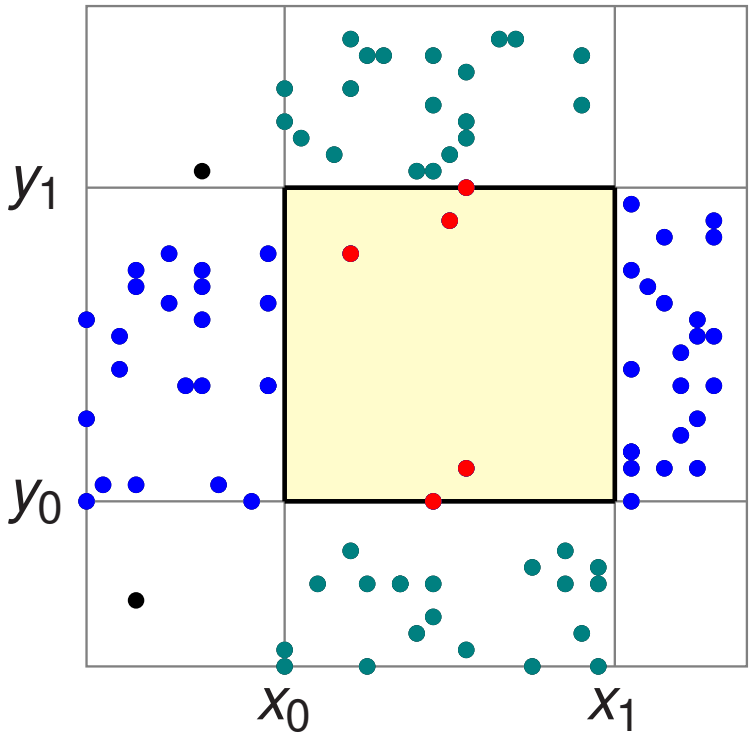
Orthogonal range searching – 2D

Karlsruher Institut für Technologie

First attempt of a solution

- Store points in array A_x and A_y
- Sort points in A_x according to x-coordinate (A_y according to y-coordinate)
- Let $k_x = count(x_0, x_1)$ in A_x, i.e all points with $x_0 \leq x \leq x_1$
- Let $k_y = count(y_1, y_1)$ in A_y, i.e. all points with $y_0 \leq y \leq y_1$
- Check smaller point list for both constaints
- Time complexity for this approach: $O(\log n) + O(min(k_x, k_y))$
- Well, there are cases...

Orthogonal range searching – 2D

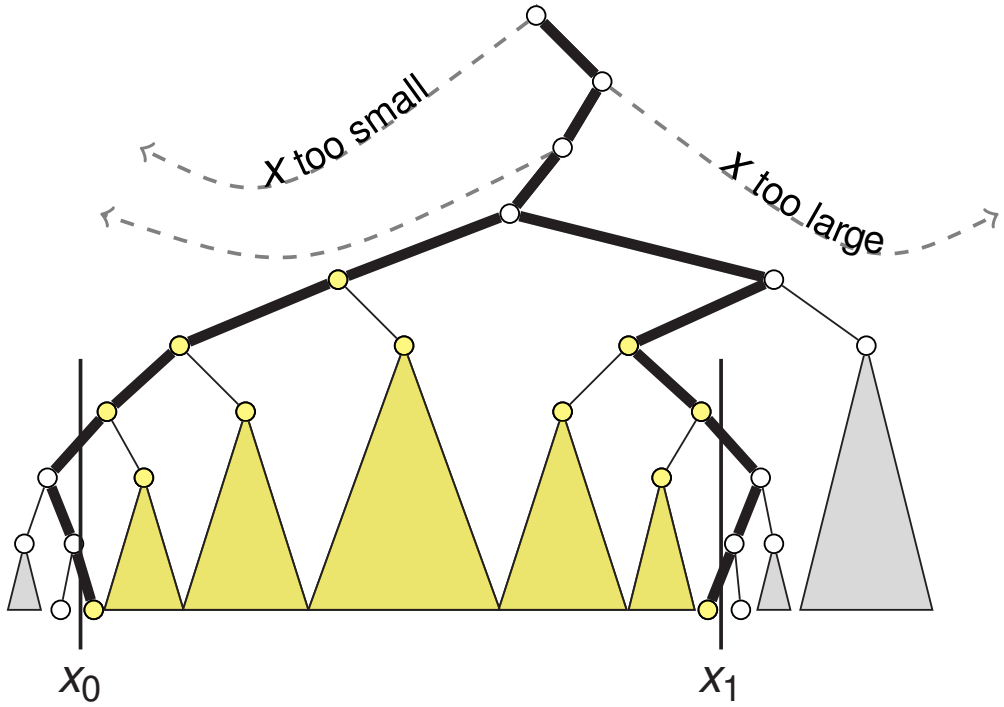


$k_x = 45$
$k_y = 49$
$k = 5$
$n = 91$

Orthogonal range search – 2D

Second attempt

- Build a balanced binary tree using the x coordinates
- Calculate the $O(\log n)$ subtrees which contain all points with $x_0 \leq x \leq x_1$
- Idea: Filter these subtrees by y-coordinate



Orthogonal range searching – 2D

Karlsruher Institut für Technologie

How to filter by y-coordinate

- For each node v in the tree build a 1D range searching structure on the y-coordinates of all points in v's subtree
- This can be done during the preprocessing
- How does the query process change?
  - Determine paths to successor and predecessor of x_0 and x_1
  - Determine the root nodes of the $O(\log n)$ included off-path subtrees
  - For each such root node v_i retrieve all points which are in $[y_0, y_1]$ in $O(\log n + k_i)$ time, where k_i is the number of matching points

Total time complexity: $O(\log^2 n + k)$

- At most $O(\log n)$ subtrees for x
- Retrieval time for each subtree $O(\log n + k_i)$
- Points from two different subtrees are *distinct*

Orthogonal range searching – 2D

Karlsruher Institut für Technologie

How to filter by y-coordinate

- For each node v in the tree build a 1D range searching structure on the y-coordinates of all points in v's subtree
- This can be done during the preprocessing
- How does the query process change?
  - Determine paths to successor and predecessor of x_0 and x_1
  - Determine the root nodes of the $O(\log n)$ included off-path subtrees
  - For each such root node v_i retrieve all points which are in $[y_0, y_1]$ in $O(\log n + k_i)$ time, where k_i is the number of matching points

Total time complexity: $O(\log^2 n + k)$

- At most $O(\log n)$ subtrees for x
- Retrieval time for each subtree $O(\log n + k_i)$
- Points from two different subtrees are *distinct*

Orthogonal range searching – 2D

Karlsruher Institut für Technologie

How much space is used?

- Tree height is $O(\log n)$
- On each level ℓ each point is represented in only one node
- $\sum_{i=0}^{c_1 \log n} c_2 n = O(n \log n)$ words

How long does the preprocessing take?

- Problem: points have to be sorted according to y-coordinate in each node
- Solution: bottom-up construction
  - Start at the leaves
  - Merge the (already sorted) lists of the two children of a node
  - I.e. $O(n \log n)$ construction time

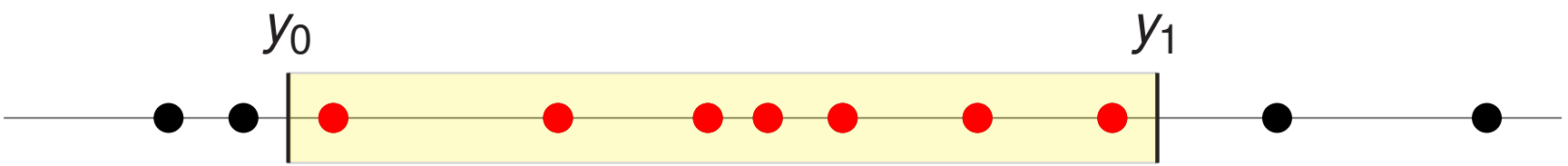
Orthogonal range searching – 2D

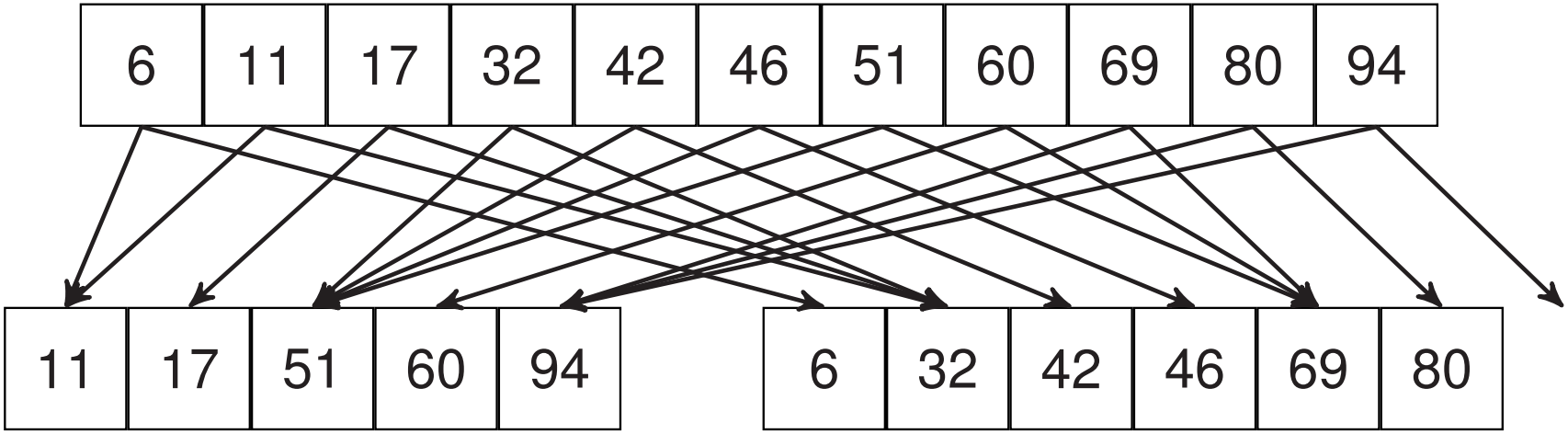
Karlsruher Institut für Technologie

2d-range searching in $O(\log n + k)$ time

- Idea: Avoid expensive calcuation of successor/predecessor in all $O(\log n)$ 1d-range structures for y-coordinates
- Determine successor/predecessor in root node and map result into child nodes
- Technique known as *fractional cascading*
- More detailed: For each node v and entry of the y-range searching structure store a pointer to the corresponding successor in v's left and right child

Orthogonal range searching – 2D





Überblick

Quellen

- Skript zu dieser Vorlesung
- Unterlagen von R. van Stee zur Vorlesung «Approximations- und Online-Algorithmen» (KIT)
- Buch Borodin/El-Yaniv: Online Computation and Competitive Analysis
- Unterlagen von F. Kuhn zur Vorlesung «Algorithm Theory» (Uni Freiburg)
- Unterlagen von G. Schnitger und A. Kovacs (beide Uni Frankfurt/Main) zur Vorlesung «Effiziente Algorithmen»
- Unterlagen zur Vorlesung «Online and Approximation Algorithms» von Susanne Albers (TU München)
- und viele andere mehr …

Eine Reihe von Beispielen

auf manche werden wir genauer eingehen:

- Job-Scheduling
- Skiausleihe
- Speicherverwaltung
- Auswahl von «Experten»
- selbstorganisierende Datenstrukturen

Beispiel Job-Scheduling

ähnlich wie im Kapitel zu Approximationsalgorithmen

- *Maschinen* $M_1, \ldots, M_m$
- *Anfrage:* ein Job J_i, der Zeit $t_j \geq 0$ benötigt
- *Antwort:* Zuordnung von J_i zu einer Maschine M_j
- Wie kann man den *Makespan* minimieren?
  (Wann ist die letzte Maschine fertig?)

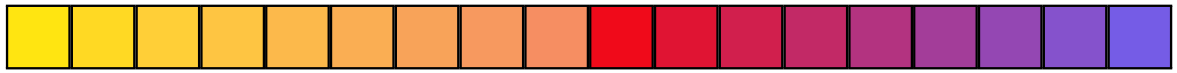
- Entscheidung für jedes J_i ohne Kenntnis der zukünftigen Jobs

Beispiel Skiausleihe

- *Skiurlaub:* solange das Wetter gut ist, jeden Morgen
  Anforderung: «Skier besorgen!»
  - sobald Wetter schlecht, Aufforderung: «Heimfahren!»
- zwei mögliche *«Antworten»*
  - Ski für einen Tag ausleihen; Kosten k €
  - Ski für ganzen Urlaub kaufen; Kosten K €, $K > k$
- Was tun, um Gesamtkosten klein zu halten?
  (in welchem Sinne?)
- Entscheidung ohne Kenntnis des zukünftigen Wetters

Beispiel Speicherverwaltung

- langsamer *Hauptspeicher* Größe N



schneller *Cache* Größe $k < N$

- *Anforderung:* eine Seite P des Hauptspeichers: 
  - P im Cache: alles gut
- P nicht im Cache: *Cache Miss, Strafpunkt*, *Seitenfehler*
  «Antwort» des Systems:
  - verdränge an einer Stelle a im Cache
  - die dort gespeicherte Seite durch P
- Welche Verdrängungsstrategie minimiert die «Kosten»?
- Auswahl der verdrängten Seiten ohne Kenntnis der zukünftigen Anforderungen

Auswahl von Experten

- in jeder von mehreren Runden:
  1. jeder von n Experten: eine Ja/Nein-Empfehlung
     - Experten können irren!
  2. eigene Ja/Nein-Entscheidung
  3. Mitteilung, welche Entscheidung richtig gewesen wäre
- Wie kann man die Anzahl Fehlentscheidungen minimieren? (in welchem Sinne?)
- eigene Entscheidungen ohne Kenntnis der zukünftigen Qualität der Expertenantworten

Beispiel Selbstorganisierende Datenstrukturen

- einfach verkettete Liste
- *Anforderung:* ein Element x der Liste
  Kosten: Position von x in Liste
- mögliche *Reaktionen:*
  - ohne weitere Kosten: angefragtes Element weiter nach vorne
  - mit Kosten 1: Vertauschung zweier aufeinanderfolgender Elemente
- Welche Listen-Verwaltung minimiert die Kosten?
- eventuelle Umordnungen ohne Kenntnis der zukünftigen Anforderungen

Online-Algorithmus

Formalisierung (Borodin/El-Yaniv folgend)

- «Eingabe» als *Folge von Anforderungen* (engl. *requests*)
  $\sigma = (r_1, r_2, \ldots, r_n) \in R^n$
- auf Anforderung r_i muss bearbeitet werden
  - *Antwort* $a_i = g_i(r_1, \ldots, r_i) \in A$
  - *ohne Wissen der Zukunft* $(r_{i+1}, \ldots)$
    - auch keine probabilistischen Annahmen
  - *unwiderruflich*
- also ALG festgelegt durch $g_1, g_2, \ldots$
- *Kosten* festgelegt durch $\text{cost}_n : R^n \times A^n \longrightarrow \mathbb{R}_{>0}$ (nie 0)
- für $\sigma \in R^n$ produziert ALG Ausgabe
  $\text{ALG}[\sigma] = (g_1(r_1), g_2(r_1, r_2), \ldots, g_n(r_1, \ldots, r_n))$
  mit Kosten $\text{ALG}(\sigma) = \text{cost}_n(\sigma, \text{ALG}[\sigma])$

Competitive Analysis

Warum man nicht einfach die Kosten schlimmster Instanzen betrachten kann

- Beispiel Speicherverwaltung
  - schlimmste Anforderungsfolgen $\sigma \in R^n$ erzwingen
  - *in jedem Schritt* Seitenfehler
  - *für jeden Onlinealgorithmus*
- schlimmste Instanzen sind immer gleich schlimm

Competitive Analysis

Betrachte alle Instanzen und vergleiche mit «optimalem» Wettbewerber

- betrachte *«optimalen Offline-‹Algorithmus› OPT»*
- (jedenfalls, falls A endlich ist, existiert)
  für $\sigma \in R^n$ $\text{OPT}(\sigma) = \min\{\text{cost}_n(\sigma, \tau) \mid \tau \in A^n\}$
- nur bei Kenntnis der *ganzen* Folge σ bestimmbar

- Aufgabe: Vergleiche ALG mit OPT!
- Frage: Wie?

- Beachte: hier nur *Minimierungsprobleme*

Wettbewerbsfaktor

für Onlinealgorithmus ALG heist

$$c_{\text{ALG}} = \sup\{\text{ALG}(\sigma)/\text{OPT}(\sigma) \mid \sigma \in R^+\}$$

der *Wettbewerbsfaktor* (engl. *competitive ratio*) von ALG

- sofern nicht ∞

Strikte c-Kompetitivität

der technisch angenehmere Fall

- Onlinealgorithmus ALG *strikt c-kompetitiv*, falls

$$\forall \sigma \in R^+ : \text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma)$$

- da wir stets $\text{cost}(\sigma) > 0$ voraussetzen, äquivalent zu

$$\forall \sigma \in R^+ : \text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c$$

- Wenn ALG strikt c-kompetitiv, dann
  - $c \geq 1$
  - ALG strikt c'-kompetitiv für jedes $c' \geq c$

- Wie klein kann man c machen?

Wettbewerbsfaktor und strikte Kompetitivität

Lemma

Es sei ALG ein strikt c-kompetitiver Onlinealgorithmus und $C = \{ c \mid \text{ALG ist strikt } c\text{-kompetitiv} \}$.
Dann ist

$$c_{\text{ALG}} = \inf C \quad \text{und} \quad c_{\text{ALG}} \in C \ .$$

Wettbewerbsfaktor und strikte Kompetitivität

$C = \{\, c \mid \text{ALG ist strikt } c\text{-kompetitiv} \,\}$ und $c^{\inf} = \inf C$

zeige: $c_{\text{ALG}} = c^{\inf}$ und nebenbei auch $c_{\text{ALG}} \in C$

Beweis

$\geq$: für alle σ ist $\text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c_{\text{ALG}}$
also $\text{ALG}(\sigma) \leq c_{\text{ALG}} \cdot \text{OPT}(\sigma)$
also ist ALG auch c_{ALG}-kompetitiv
also ist $c_{\text{ALG}} \in C$ und $c_{\text{ALG}} \geq c^{\inf}$

Wettbewerbsfaktor und strikte Kompetitivität

$C = \{\, c \mid \text{ALG ist strikt } c\text{-kompetitiv} \,\}$ und $c^{\inf} = \inf C$
zeige: $c_{\text{ALG}} = c^{\inf}$ und nebenbei auch $c_{\text{ALG}} \in C$

Beweis

$\geq$: für alle σ ist $\text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c_{\text{ALG}}$
also $\text{ALG}(\sigma) \leq c_{\text{ALG}} \cdot \text{OPT}(\sigma)$
also ist ALG auch c_{ALG}-kompetitiv
also ist $c_{\text{ALG}} \in C$ und $c_{\text{ALG}} \geq c^{\inf}$

$\leq$: indirekt: angenommen $c_{\text{ALG}} - c^{\inf} = d > 0$
also $c' = c^{\inf} + d/2 \in C$
also $\forall\sigma$: $\text{ALG}(\sigma) \leq c' \cdot \text{OPT}(\sigma)$
also $\sup_\sigma \text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c'$
im Widerspruch zu $c' = c_{\text{ALG}} - d/2 < c_{\text{ALG}}$

Nicht-strikte c-Kompetitivität

- Onlinealgorithmus ALG *c-kompetitiv* für ein $c \in \mathbb{R}_{>0}$, falls Konstante $b \in \mathbb{R}$ (unabhängig von σ) existiert mit

$$\forall \sigma \in R^+ : \text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + b$$

- im Vergleich zu *strikter* c-Kompetitivität erlaubt man Ausnahmen

- Achtung:
  - wir *vermeiden* hier den Begriff Wettbewerbsfaktor bzw. competitive ratio
  - andere benutzen ihn weiter, *in unterschiedlichen Bedeutungen*

Überblick

listScheduling ist (fast) ein Onlinealgorithmus

Approximationsalgorithmus

```
1  listScheduling(n, m, t_{1...n})
2  each L_i ← 0                          ▷ load of machine i, 1 ≤ i ≤ m
3  each S_j ← 0                          ▷ machine for job j, 1 ≤ j ≤ n
4  for each j in range(1, n)
5       do pick k from {i | L_i is currently minimal }
6          S_j ← k
7          L_k ← L_k + t_j
8
9  return S
```

frühere Analyse: Wettbewerbsfaktor höchstens 2

listScheduling ist (fast) ein Onlinealgorithmus

Onlinealgorithmus

```
1  listScheduling(   m      )
2  each L_i ← 0                    ▷ load of machine i, 1 ≤ i ≤ m
3
4  for each t_j in σ
5       do pick k from {i | L_i is currently minimal }
6          a_j ← k
7          L_k ← L_k + t_j
8          assign Job j to machine a_j
9
```

frühere Analyse: Wettbewerbsfaktor höchstens 2

Überblick

Skiausleihe

- Ski für einen Tag ausleihen kostet k €
- Ski für ganzen Urlaub kaufen kostet K €, $K > k$
- Man weiß nicht, wie lange der Urlaub dauert.
- Am wievielten Tag sollte man die Ski kaufen?

Ähnlich (was ist anders?)

- Nach wievielen Bahnfahrten eine Bahncard kaufen?

Optimale Kosten?

- Anforderungsfolge: t mal «Skier ausleihen!»
- Optimale Kosten:
  - falls Urlaubsdauer $t \leq K/k$ Tage: t mal Ausleihe
    - Kosten tk
  - falls Urlaubsdauer $t \geq K/k$ Tage: sofort kaufen
    - Kosten K

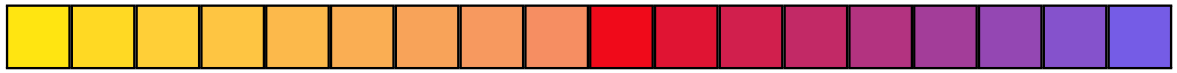
Deterministische Entscheidung für Skikauf

- Entscheidung hängt nur von k und K ab.
- *Wettbewerbsfaktor* 2
  - wenn Kauf an Tag K/k
  - besser gehts nicht
  - Details in der Übung

Überblick

Problemstellung

- langsamer *Hauptspeicher* Größe N



schneller *Cache* Größe $k < N$

- *Anforderung:* eine Seite P des Hauptspeichers: 
  - P im Cache: alles gut
- P nicht im Cache: *Cache Miss, Page Fault, Strafpunkt*
  Reaktion des Systems:
  - verdränge an einer Stelle a im Cache
  - dort gespeicherte Seite durch P
- Welche Verdrängungsstrategie minimiert die «Kosten»?
- Online-Entscheidungen ohne Kenntnis der Zukunft

Plan für diesen Abschnitt

- Offlinealgorithmus LFD und seine Optimalität
- deterministische Onlinealgorithmen:
  untere Schranke k für Wettbewerbsfaktor
- wird von LRU erreicht
- kurz: resource augmentation
- randomisierter Onlinealgorithmus:
  RANDMARK hat Wettbewerbsfaktor $2H_k$

Überblick

Longest Forward Distance (LFD)

- Aus dem Cache wird ein Eintrag verdrängt, der *am spätesten in der Zukunft* wieder angefordert wird.
- Beispiel:
  vor Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	4	2				

Longest Forward Distance (LFD)

- Aus dem Cache wird ein Eintrag verdrängt, der *am spätesten in der Zukunft* wieder angefordert wird.
- Beispiel:
  vor Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	4	2				

  nach Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

  nach Bearbeitung der zweiten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der zweiten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der zweiten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

  nach Bearbeitung der dritten Anforderung

σ	3	1	5	3	2	4	1
Cache	5	3	2				

Optimalität von LFD

Satz

LFD ist optimal.

Optimalität von LFD — Beweisskizze (1)

Satz

LFD ist optimal.

Beweisidee

- zeige: Ein optimaler Algorithmus OPT kann schrittweise zu LFD umgebaut werden.
- für induktiven Schritt folgendes Lemma

Lemma

Gegeben ALG, $\sigma = (r_1, \ldots, r_n)$ und $i \leq n$ arbeite $\text{LFD}_i(\text{ALG})$ so:

- $\text{LFD}_i(\text{ALG})$ verarbeitet $(r_1, \ldots, r_{i-1})$ wie ALG
- Bei Cache Miss für r_i entfert $\text{LFD}_i(\text{ALG})$ den am spätestens in der Zukunft angeforderten Eintrag.

Dann gilt: $\text{LFD}_i(\text{ALG})(\sigma) \leq \text{ALG}(\sigma)$.

Optimalität von LFD: Beweisskizze (2)

- angenommen, Lemma wäre gezeigt, also $\text{LFD}_i(\text{ALG})(\sigma) \leq \text{ALG}(\sigma)$
- zu gegebenem optimalem Algorithmus OPT definiere:
  - $\text{OPT}_0 = \text{OPT}$
  - $\text{OPT}_1 = \text{LFD}_1(\text{OPT}_0)$
  - $\text{OPT}_2 = \text{LFD}_2(\text{OPT}_1)$
  - ...
  - d. h. allgemein für $1 \leq i \leq n$: $\text{OPT}_i = \text{LFD}_i(\text{OPT}_{i-1})$
- dann per Induktion
  - alle OPT_i sind optimal
  - $\text{OPT}_n = \text{LFD}$

Optimalität von LFD: Beweisskizze (3)

bleibt zu zeigen: $\text{LFD}_i(\text{ALG})(\sigma) \leq \text{ALG}(\sigma)$

nicht hier

Überblick

Deterministische Onlinealgorithmen

FIFO	first in first out
LIFO	last in first out
LRU	least recently used
LFU	least frequently used

- LIFO ist nicht kompetitiv: betrachte
  $\langle 1, 2, \ldots, k-1, k, k+1, k, k+1, k, k+1, k, k+1, \ldots \rangle$
- LFU ist nicht kompetitiv: betrachte
  $\langle 1^m, 2^m, \ldots, (k-1)^m \rangle, \langle k, k+1 \rangle^{m-1}$
- LRU und FIFO sind k-kompetitiv
  - in der Praxis nimmt man LRU

Untere Schranke für den Wettbewerbsfaktor (det. Seitenwechsel)

Satz

Jeder deterministische Onlinealgorithmus für das Seitenwechselproblem mit Cachegröße k hat einen Wettbewerbsfaktor $c \geq k$.

Untere Schranke für den Wettbewerbsfaktor (det. Seitenwechsel) — Beweisskizze (1)

Satz

Jeder deterministische Onlinealgorithmus für das Seitenwechselproblem mit Cachegröße k hat einen Wettbewerbsfaktor $c \geq k$.

Beweisskizze

- Hauptspeicher Größe $N = k + 1$
- ALG beliebiger Onlinealgorithmus für Speicherverwaltung
- betrachte σ mit der Eigenschaft:
  - Jede Anforderung verursacht Seitenfehler bei ALG und
  - $|\sigma| > k$
- Zeige: LFD hat höchstens alle k Anforderungen einen Seitenfehler.

Beweisskizze (2)

- o. B. d. A.
  - zu Beginn ALG und LFD mit gleichem Cache-Inhalt
  - Cache Miss bei erster Anforderung
- *Phase:*
  - beginnt mit Anforderung, bei der LFD Cache Miss hat
  - endet unmittelbar vor der nächsten
- Länge einer Phase: *mindestens k*, denn
  - wenn LFD einen Cache Miss hat,
  - dann wird die Seite verdrängt,
  - vor der jede der anderen $k - 1$ Seiten,
  - die gerade im Cache sind,
  - mindestens einmal angefordert wird.
- Anzahl Cache Misses pro Phase:
  - ALG Länge der Phase viele, also $\geq k$
  - LFD nur 1 zu Beginn der Phase

Überblick

LRU — Beispiel

live

LRU ist k-kompetitiv

Satz

LRU ist k-kompetitiv.

LRU ist k-kompetitiv — Beweisskizze (1)

Satz

LRU ist k-kompetitiv.

Beweisskizze

- σ beliebige Anforderungsfolge
- Aufteilung von σ in Phasen P_0, P_1, etc.
  - P_0 beginnt mit erster Anforderung
  - jedes P_{i+1} beginnt unmittelbar nach Ende von P_i
  - jede P_i umfasst maximal viele Anforderungen, die nur k Seiten betreffen
- sei außerdem Q_i die Folge P_i ohne deren erstes Element, aber mit dem ersten Element von P_{i+1} angehängt

Beweisskizze (2)

- zeige:
  - LRU hat höchstens k Cache Misses pro Phase P_i
  - LFD hat mindestens 1 Cache Miss pro Q_i
- dann bei insgesamt m Phasen
  - $\text{LRU}(\sigma) \leq km$
  - $\text{LFD}(\sigma) \geq m - 1$
  - also $\text{LRU}(\sigma) \leq k \cdot \text{LFD}(\sigma) + k$

Beweisskizze (3)

zeige: LRU hat höchstens k Cache Miss pro P_i

- andernfalls hätte LRU in einer Phase
  für eine Seite p *zweimal* einen Cache Miss
- wegen «least recently used» hätten zwischen den beiden Cache Misses k *andere* Seiten angefordert werden müssen
- d. h. es wären in einer Phase $k + 1$ verschiedene Seiten angefordert worden
- im Widerspruch zur Definition der Phasen

Beweisskizze (4)

zeige: LFD hat mindestens 1 Cache Miss pro Q_i

- seien $p_1^i, \ldots, p_k^i$ die der Reihe nach in P_i auftretenden verschiedenen Seiten
- betrachte $p_2^i, \ldots, p_k^i, p_1^{i+1}$
- das sind k verschiedene Seiten
- wenn von $p_2^i, \ldots, p_k^i$ keine einen Cache Miss verursacht, dann p_1^{i+1} (weil $p_1^i, \ldots, p_k^i$ im Cache sind)
- also hat LRU mindestens in jeder Phase außer der ersten mindestens einen Cache Miss

Überblick

Speicherverwaltung

- Offline: LFD ist optimal
- Deterministisch: bestenfalls k-kompetitiv
- Deterministisch: LRU ist k-kompetitiv
- Resource Augmentation: (h, k)-Seitenwechsel
- Randomisiert: RANDMARK ist $2H_k$-kompetitiv

Resource Augmentation

- *(h, k)-Seitenwechselproblem*
- vergleiche Onlinealgorithmus ALG_k mit Cachegröße k mit
- LFD_h mit Cachegröße $h < k$

- Onlinealgorithmus heißt *konservativ*, wenn er
  - für Anforderungsfolgen mit höchsten k verschiedenen Seiten
  - höchstens k Cache Misses hat
- Beispiele: LRU, FIFO

Resource Augmentation

Satz

Jeder konservative Onlinealgorithmus ist $\frac{k}{k-h+1}$-kompetitiv.

Beweisskizze

- Verallgemeinerung von «LRU ist k-kompetitiv».
- konservativer Alg. hat in P_i höchstens k Cache Misses
- Nach erster Anforderung von P_i
- enthält der Cache von LFD_h noch $h-1$ andere Seiten,
- die in P_{i-1} angefordert wurden.
- In Q_i folgen aber noch k andere verschiedene Seiten,
- von denen $\geq k-(h-1)$ einen Cache Miss verursachen

Beispiel $h = k/2$: $\rightsquigarrow \text{ALG}_k(\sigma) \leq 2\text{LFD}_h(\sigma) + b$

Überblick

Randomisierte Onlinealgorithmen

Bei randomisierten Online-Algorithmus R ist die Zahl der Cache Misses eine Zufallsvariable $f_R(r_1, \ldots, r_n)$.

etwas andere Sicht auf c-Kompetitivität sinnvoll

Widersacher: verschieden miese Typen

- Widersacher
  - bekommen Eingabe n
  - erzeugen für R «schlimme» Anforderungsfolgen der Länge n
  - die sie aber auch selbst verarbeiten müssen
- Wieviel Information ist über die Zufallsbits bekannt?
  - *unwissender Widersacher* W (engl. *oblivious adversary*)
    - *kein* Wissen über erzeugte Zufallsbits
    - Zu R und n erzeugt W immer gleiches $(r_1, \ldots, r_n)$.
  - *adaptiver Widersacher*:
    - arbeitet gegen eine konkrete Abarbeitung von R,
    - kennt die von R bei Abarbeitung von $(r_1, \ldots, r_i)$ erzeugten Zufallsbits,
    - und folglich auch immer den aktuellen Cachezustand von R.

Widersacher (2)

Womit vergleicht man die Zahl der Cache Misses von R?

- *unwissende Widersacher:* $\text{OPT}(r_1, \ldots, r_n)$
- *adaptive Widersacher:* zwei Varianten
  - adaptiver Online-Widersacher:
    - muss erzeugte Anforderungsfolge selbst auch irgendwie online verwalten
    - muss also sofort nach Erzeugung eines r_i entscheiden, wie er in seinem Cache damit umgeht $\rightsquigarrow$ Kosten
  - adaptiver Offline-Widersacher:
    - kann zunächst vollständig $(r_1, \ldots, r_n)$ erzeugen und
    - bekommt nur $\text{OPT}(r_1, \ldots, r_n)$ in Rechnung gestellt
  - In beiden Fällen ist Anzahl der Cache Misses von W Zufallsvariable (Abhängigkeit von den Zufallsbits von R)

Wettbewerbsfaktor

- R ist *c-kompetitiv gegen unwissende Widersacher*, wenn es ein von n unabhängiges b gibt so, dass für jede Anforderungsfolge $(r_1, \ldots, r_n)$ gilt:

$$\mathbf{E}\left[f_R(r_1, \ldots, r_n)\right] - c \cdot \mathrm{OPT}(r_1, \ldots, r_n) \leq b$$

  nur das werden wir hier betrachten

- R ist *c-kompetitiv gegen einen adaptiven Online-* resp. *Offline-Widersacher*, wenn es ein von n unabhängiges b gibt so, dass gilt:

$$\mathbf{E}\left[f_R(r_1, \ldots, r_n) - c \cdot f_W(r_1, \ldots, r_n)\right] \leq b$$

resp.
$$\mathbf{E}\left[f_R(r_1, \ldots, r_n) - c \cdot f_O(r_1, \ldots, r_n)\right] \leq b$$

randMark Algorithmus (Fiat et al., 1991)

randMark Algorithmus (Fiat et al., 1991)

$\langle$*Cache: cache*$[i]$, *Markierungsbits mark*$[i]$, $1 \leq i \leq k\rangle$
for $i \leftarrow 1$ **to** k **do** *mark*$[i] \leftarrow 0$ **od**
while $\langle$*noch weitere Anforderungen*$\rangle$ **do**
    r $\leftarrow \langle$*nächste Anforderung*$\rangle$
    if $\langle$*memory*$[r]$ *nicht in cache*$\rangle$ **then**
        if $\langle$*alle mark*$[i] = 1\rangle$ **then** $\langle$*alle mark*$[i] \leftarrow 0\rangle$ **fi**
        $i \leftarrow \langle$*zufälliges j mit mark*$[j] = 0\rangle$
        cache$[i] \leftarrow$ *memory*$[r]$
    else
        $i \leftarrow \langle$*Index mit cache*$[i] =$ *memory*$[r]\rangle$
    fi
    mark$[i] \leftarrow 1$
od

randMark Algorithmus (Fiat et al., 1991)

$\langle$*Cache: cache*$[i]$, *Markierungsbits mark*$[i]$, $1 \leq i \leq k\rangle$
for $i \leftarrow 1$ **to** k **do** *mark*$[i] \leftarrow 0$ **od**
while $\langle$*noch weitere Anforderungen*$\rangle$ **do**
    r $\leftarrow \langle$*nächste Anforderung*$\rangle$
    if $\langle$*memory*$[r]$ *nicht in cache*$\rangle$ **then**
        if $\langle$*alle mark*$[i] = 1\rangle$ **then** $\langle$*alle mark*$[i] \leftarrow 0\rangle$ **fi**
        $i \leftarrow \langle$*zufälliges j mit mark*$[j] = 0\rangle$
        cache$[i] \leftarrow$ *memory*$[r]$
    else
        $i \leftarrow \langle$*Index mit cache*$[i] =$ *memory*$[r]\rangle$
    fi
    mark$[i] \leftarrow 1$
od

Kompetitivität von randMark

Satz

randMark ist $2H_k$-kompetitiv gegen unwissende Widersacher.

Beachte:

- $2H_k \in \Theta(\log k)$
- jeder det. Online-Alg. ist bestenfalls k-kompetitiv

Beweis

- Algorithmus zerfällt *Phasen*:
  - Phase beginnt mit Zurücksetzen aller Markierungsbits auf 0
  - und endet unmittelbar vor der nächsten.
- Betrachte LFD (optimal) und RANDMARK für $r_1, r_2, r_3 \ldots$
- anfangs gleiche Cacheinhalte und r_1 führe zu Cache Miss
- also
  - jede Phase beginnt mit Cache Miss,
  - umfasst maximale Teilfolge $r_i, \ldots, r_j$
  - mit genau k verschiedenen Adressen,
  - denn jede in der Teilfolge «neue» Adresse führt dazu,
  - dass ein Markierungsbit auf 1 gesetzt wird, und
  - Phase endet direkt vor $(k + 1)$-ter neuer Adresse.

Beweis (2)

Zeige:

1. LFD hat im Mittel pro Phase $\geq \ell/2$ Cache Misses.
2. RANDMARK hat erwartet pro Phase ℓH_k Cache Misses.

Definition von ℓ kommt gleich

Beweis (3)

betrachte einzelne Phase

ein Datum heiße

- *veraltet*, wenn es *zu Beginn der Phase* im Cache ist
- *sauber*, wenn es zu Beginn der Phase *nicht* im Cache ist
- *markiert*, wenn es *zum betrachteten Zeitpunkt* an einer markierten Stelle des Caches liegt

Beweis (4)

- randMark entfernt aus dem Cache stets ein nicht markiertes, veraltetes Datum und
- das den Cache Miss verursachende Datum ist ab dem Zeitpunkt, zu dem es geladen wird, markiert.
- Innerhalb einer Phase führt die erste Anforderung jedes sauberen Datums zu Cache Miss.
- ℓ: Anzahl sauberer Datenelemente, die durch randMark im Laufe der Phase (evtl. mehrfach) angefordert werden.

Beweis (5)

Anforderung eines veralteten Datums kann zu Cache Miss führen

- wenn es zwischenzeitlich (aufgrund eines anderen Cache Miss) aus dem Cache entfernt worden war.
- Das kann aber *nur einmal* innerhalb einer Phase zu einem Cache Miss führen, da es beim erneuten Laden markiert wird.
- Außerdem wird hierdurch wieder ein (nicht markiertes, also) veraltetes Datum verdrängt.

Beweis (6)

Abschätzung Cache Misses bei LFD

- S_O: Menge der Cacheelemente von LFD
  S_R: Menge der Cacheelemente von randMark
- d_a: Größe von $S_O \setminus S_R$ am Anfang der Phase
  d_e: Größe von $S_O \setminus S_R$ am Ende der Phase
- m_O: Anzahl der Cache Misses von LFD während der Phase.

Beweis (6)

Abschätzung Cache Misses bei LFD

- S_O: Menge der Cacheelemente von LFD
  S_R: Menge der Cacheelemente von RANDMARK
- d_a: Größe von $S_O \setminus S_R$ am Anfang der Phase
  d_e: Größe von $S_O \setminus S_R$ am Ende der Phase
- m_O: Anzahl der Cache Misses von LFD während der Phase.

Zu Beginn der Phase:

- ℓ später angeforderte saubere Datenelemente nicht in S_R
- höchstens d_a von ihnen sind in S_O
- also $m_O \geq \ell - d_a$

Beweis (7)

Abschätzung Cache Misses von LFD (2)

Am Ende der Phase:

- S_R enthält genau die k markierten, also insbesondere auch während der Phase angeforderten Elemente.
- Davon sind d_e am Ende nicht mehr in S_O, sie sind also vom optimalen Algorithmus verdrängt worden.
- Das kann nur durch den Cache Miss eines anderen Elementes geschehen sein.
- Als muss der optimale Algorithmus mindestens d_e Cache Misses erzeugt haben.
- Also $m_O \geq d_e$

Beweis (8)

Abschätzung Cache Misses von LFD (3)

Insgesamt:

- $m_O \geq \max\{\ell - d_a, d_e\} \geq (\ell - d_a + d_e)/2$.

Beweis (8)

Abschätzung Cache Misses von LFD (3)

Insgesamt:

- $m_O \geq \max\{\ell - d_a, d_e\} \geq (\ell - d_a + d_e)/2$.

Summation über alle Phasen:

- d_e am Ende einer Phase ist d_a zu Beginn der nächsten.
- die Summanden für d_a und d_e heben sich auf,
  - außer zu Beginn der ersten Phase: $d_a = 0$
  - und am Ende der letzten Phase d_e
- Vorstellung:
  - schiebe Cache Misses in „benachbarte Phasen“
  - immer noch $m_O \geq \ell/2$ Cache Misses in jeder Phase

Beweis (9)

Abschätzung Cache Misses von randMark (1)

- jeweils erste Anforderung eines der ℓ sauberen Elemente
  - führt zu Cache Miss
  - übrige Anforderungen sauberer Elemente nicht
- alle anderen Anforderungen: veraltete Elemente
- am Ende der Phase sind $k - \ell$ davon im Cache.
- erwartete Anzahl dadurch erzwungener Cache Misses maximieren: erst alle sauberen Elemente angefordern.
- danach fehlen im Cache ℓ veraltete Elemente
- das bleibt bis zum Ende der Phase so, denn
  - durch Anforderung eines fehlenden veralteten Datums x wird stets ein anderes veraltetes Datum x' verdrängt,
  - aber x wird bis zum Ende der Phase nicht mehr verdrängt.

Beweis (10)

Abschätzung Cache Misses von randMark (2)

- Seien $x_1, \ldots, x_{k-\ell}$ die am Ende der Phase im Cache befindlichen veralteten Elemente.
- x_1 zuerst angefordert, dann x_2, usw.
- W.keit für Cache Miss bei 1. Anforderung von x_i, $i \leq k - \ell$: gleich der W.keit, ein nicht im Cache vorhandenes veraltetes Element auszuwählen aus den veralteten Elementen, die in dieser Phase noch nicht angefordert wurden.
- stets ℓ veraltete Elemente nicht im Cache
- bei erster Anforderung von x_i wurden $k - (i - 1)$ veraltete Elemente noch nicht angefordert
- relative Häufigkeit eines Cache Miss also $\ell/(k - i + 1)$

Beweis (11)

Abschätzung Cache Misses von randMark (3)

- Es ist

$$\sum_{i=1}^{k-\ell} \frac{\ell}{(k-i+1)} = \ell\left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{\ell+1}\right) = \ell(H_k - H_\ell) \ .$$

- also erwartete Anzahl Cache Misses kleiner oder gleich

$$\ell + \ell(H_k - H_\ell) = \ell H_k - (H_\ell - 1)\ell \leq \ell H_k \ .$$

Überblick

Überblick

Problemstellung

- in jeder von mehreren Runden:
  1. jeder von k «Experten»: eine Ja/Nein-Empfehlung
     - Experten können irren!
  2. eigene Ja/Nein-Entscheidung
  3. Mitteilung, welche Entscheidung richtig gewesen wäre

Ziel:

- *Anforderung:* k-Tupel, J/N-Empfehlungen der Experten
- *Antwort:* eigene J/N-Entscheidung
- *Kosten:* Anzahl eigener Fehlentscheidungen
- Ziel: eigene Kosten nahe an Kosten der besten Experten

Auswahl von Experten

Wie man es nicht machen sollte:

- Wähle die Antwort eines Experten, der bisher am besten war.

denn

- für jedes k Menge von k Experten konstruierbar, für die gilt:
  - Die obige Strategie irrt *immer*.
  - Jeder Experte hat nach t Runden höchsten t/k Fehler gemacht.
- $\rightsquigarrow$ Übung

Wie dann?

Auswahl von Experten: der deterministische Weighted Majority Algorithm (WMA)

- Experte i hat ein «Gewicht» w_i
- initial: alle $w_i \leftarrow 1$
- in jeder Runde
  1. Experten geben Empfehlungen $x_i \in \{ \text{ja}, \text{nein} \}$
  2. eigene Entscheidung fällen:

$$\begin{cases} \text{ja}, & \text{falls } \sum_{i, x_i = \text{ja}} w_i \geq \sum_{i, x_i = \text{nein}} w_i \\ \text{nein}, & \text{falls } \sum_{i, x_i = \text{ja}} w_i < \sum_{i, x_i = \text{nein}} w_i \end{cases}$$

  3. Expertengewichte anpassen

$$w_i^{t+1} = \begin{cases} w_i^t, & \text{falls } x_i \text{ richtig war} \\ w_i^t / 2, & \text{falls } x_i \text{ falsch war} \end{cases}$$

Qualität von WMA

Satz

WMA ist $1/\log_2(4/3)$-kompetitiv. Genauer

$$\text{WMA}(\sigma) \leq \frac{1}{\log_2(4/3)} (\text{OPTXPRT}(\sigma) + \log_2 k)$$

Beweis

in einer Runde t

- W_t: Summe der w_i zu Beginn der Runde
  - initial $W_1 = n$
- x_i die Expertenempfehlungen, y die richtige Antwort
- $R_t = \sum_{i,x_i=y} w_i$ und $F_t = \sum_{i,x_i \neq y} w_i$
  - $W_t = R_t + F_t$

wenn Entscheidung falsch, dann

- $R_t < W_t/2$
- $W_{t+1} = \frac{F_t}{2} + R_t = \frac{F_t}{2} + \frac{R_t}{2} + \frac{R_t}{2} \leq \frac{W_t}{2} + \frac{W_t}{4} = \frac{3}{4} W_t$

Beweis (2)

bis zu Beginn von Runde t

- wenn eigene Entscheidungen f mal falsch
  dann $W_t \leq \left(\frac{3}{4}\right)^f \cdot W_1 = \left(\frac{3}{4}\right)^f \cdot k$
- wenn bester Experte i nur f_{opt} mal falsch
  dann $W_t \geq w_i = \left(\frac{1}{2}\right)^{f_{opt}}$
- also $\left(\frac{1}{2}\right)^{f_{opt}} \leq \left(\frac{3}{4}\right)^f \cdot k$
- also $f \leq \frac{1}{\log_2(4/3)} \left(f_{opt} + \log_2 k\right)$
  - $\frac{1}{\log_2(4/3)} \approx 2.409$

Überblick

Verallgemeinerte Problemstellung

- in Runde t:
  1. Experten: Empfehlungen $x_1, \ldots, x_k$
  2. eigene Wahl einer Empfehlung
  3. Mitteilung, welche Entscheidung richtig gewesen wäre
  4. Beurteilung der Empfehlungen durch «Noten» c_i^t
     - beste Note: 0
     - schlechteste Note: 1
  5. eigene *Kosten:* Note der gewählten Empfehlung

Randomisiert: RandWMA$_\varepsilon$

- es sei $0 < \varepsilon < 1/2$
- Experte i hat ein «Gewicht» w_i
- initial: alle $w_i \leftarrow 1$
- in jeder Runde $t = 1, 2, \ldots$
  1. wähle Empfehlung von Experte i mit Wahrscheinlichkeit $p_i = \frac{w_i}{\sum_1^k w_i}$
  2. Benotung
  3. setze jedes $w_i \leftarrow w_i(1 - \varepsilon c_i^t)$

Qualität von RANDWMA$_\varepsilon$

Satz

Es sei k die Anzahl der Experten und $0 < \varepsilon < 1/2$.
Wenn RANDWMA$_\varepsilon$ nach einer Anzahl Runden erwartete Gesamtkosten K hat und die besten Experten Empfehlungen mit (minimalen) Gesamtkosten K_{opt} gegeben haben, dann ist

$$K \leq (1 + \varepsilon) \cdot K_{opt} + \frac{\ln n}{\varepsilon}$$

Beweis

- setze
  - w_i^t Gewicht von Experte i zu Beginn von Runde t
  - $W_t = \sum_{i=1}^{k} w_i$ Gesamtgewicht
  - $K_t = \sum_{i=1}^{k} \frac{w_i^t}{W_t} c_i^t$ erwartete eigene Kosten in Runde t
- $$W_{t+1} = \sum_{i=1}^{k} w_i^t(1 - \varepsilon c_i^t) = \sum_{i=1}^{k} w_i^t - \varepsilon \sum_{i=1}^{k} w_i^t c_i^t$$
  $$= W_t - \varepsilon K_t W_t = W_t(1 - \varepsilon K_t)$$
- also $W_{t+1} = W_1 \cdot \prod_{\tau=1}^{t}(1 - \varepsilon K_\tau)$
- also $\ln W_{t+1} \leq \ln n - \sum_{\tau=1}^{t} \varepsilon K_\tau = \ln n - \varepsilon K$
  - wegen $\ln(1 - x) \leq -x$ für $0 \leq x \leq 1/2$

Beweis (2)

- andererseits für jeden Experten i (auch den opt.):
  $W_{t+1} \geq w_i^{t+1} = \prod_{\tau=1}^{t}(1 - \varepsilon c_i^\tau)$
- wegen $\ln(1-x) \geq -x - x^2$ für $0 \leq x \leq 1/2$ daher
  $\ln W_{t+1} \geq \sum_{\tau=1}^{t} \ln(1 - \varepsilon c_i^\tau) \geq -\sum_{\tau=1}^{t}(\varepsilon c_i^\tau + (\varepsilon c_i^\tau)^2)$
- wegen $(c_i^\tau)^2 \leq c_i^\tau$ weiter:
  $\ln W_{t+1} \geq -\sum_{\tau=1}^{t}(\varepsilon c_i^\tau + \varepsilon^2 c_i^\tau) = -(\varepsilon + \varepsilon^2)K(i)$
- insgesamt insbesondere für K_{opt}:

$$-(\varepsilon + \varepsilon^2)K_{opt} \leq \ln W_{t+1} \leq \ln k - \varepsilon \cdot K$$

$$\text{also} \qquad K \leq (1+\varepsilon) \cdot K_{opt} + \frac{\ln k}{\varepsilon}$$

Überblick

Warum Parallelverarbeitung

Warum Parallelverarbeitung

Zeitersparnis: p Computer, die gemeinsam an einem Problem arbeiten, lösen es bis zu p mal so schnell.

- *aber* gute Koordinationsalgorithmen nötig

Warum Parallelverarbeitung

Zeitersparnis: p Computer, die gemeinsam an einem Problem arbeiten, lösen es bis zu p mal so schnell.

- *aber* gute Koordinationsalgorithmen nötig

Kommunikationsersparnis: wenn Daten verteilt anfallen, kann man sie auch verteilt (vor)verarbeiten

Energieersparnis: zwei Prozessoren mit halber Taktfrequenz brauchen weniger als eine voll getakteter Prozessor.

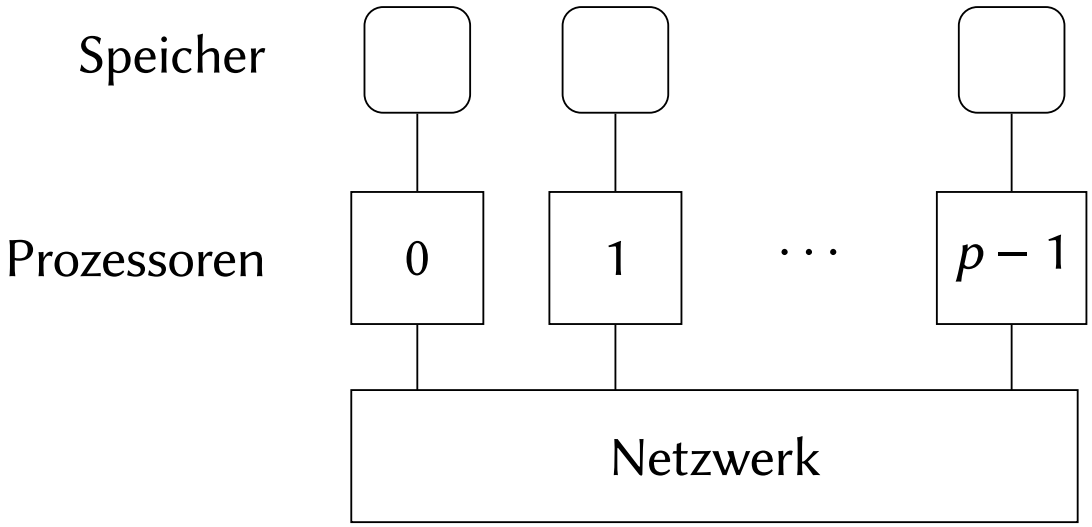
Speicherbeschränkung mehr Prozessoren haben mehr Hauptspeicher, mehr Cache, …

Verschiedene Modelle für Parallelverarbeitung

- es gibt *viiiiiiiiele*
  - konzeptionell sehr unterschiedlich (auf den ersten Blick)
  - Parallelverarbeitung zum Teil «versteckt»
- in dieser Vorlesung zwei Standardmodelle
  - Prozessornetzwerk mit *Nachrichtengekopplung*
  - *parallele Registermaschinen* mit Speicherkopplung

Nachrichtengekoppelter Parallelrechner

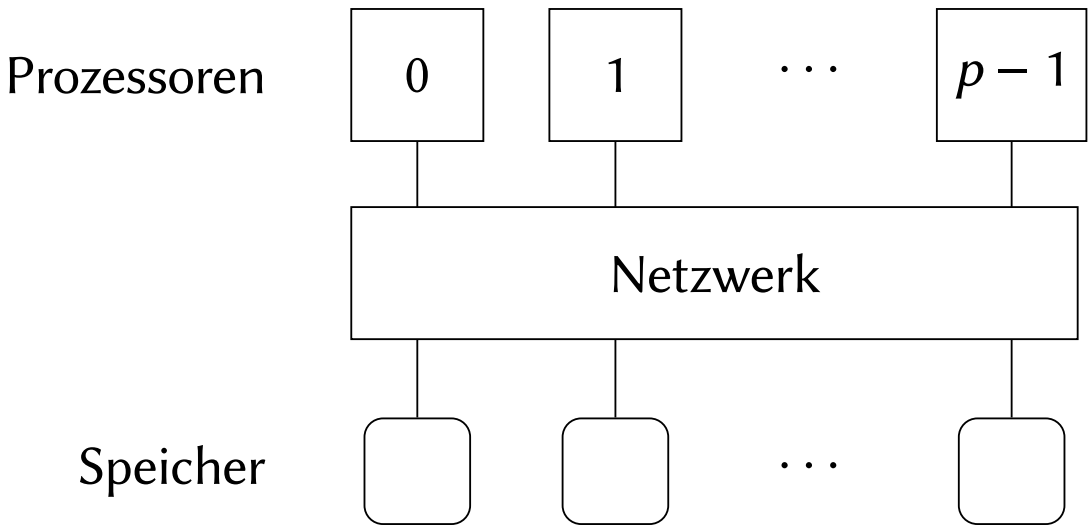
- Prozessoren: *random access machines*
  - «normale CPUs» mit lokalem Speicher
  - jedes Speichermodul nur von «seinem» Prozessor zugreifbar
  - nennen wir auch *PE (processing element)*
- *Datenaustausch*
  - über Netzwerk
  - *Nachrichten* von einem Prozessor zu einem anderen



Parallelrechner mit globalem Speicher

ein möglicher Aufbau

- Prozessoren wie üblich
  - aber kein «lokaler» Speicher
  - jedes Speichermodul von «allen» Prozessoren zugreifbar
- *Datentransport*
  - über Netzwerk
  - zwischen einer CPU und einem Speicher



Nachrichtenkopplung versus Speicherkopplung

potenzielle Nachteile von Nachrichtenkopplung

- für Datentransport Beteilgung von *zwei* Prozessoren
  - für Empfänger mitunter «unpassend»
- Parallelismus muss explizit programmiert werden

potenzielle Nachteile von Speicherkopplung

- *Skalierbarkeit:* große Prozessorzahlen sinnvoll?
- Kostenmaß bei *Speicherzugriffskonflikten?*

daher oft gute Strategie:

- Entwurf für verteilten Speicher
  - deckt viel breiteren Bereich ab
- Implementierung u. U. für gemeinsamen Speicher

Parallele Beispielalgorithmen

einfache, aber wichtige, algorithmische Probleme

- Datenumverteilung
- Präfixsummen
- Sortieren
- …

Überblick

Überblick

Modell für Nachrichtenaustausch

ähnlich wie bei MPI

- Netzwerk
  - Punkt-zu-Punkt
  - vollständig verknüpft
  - Nachrichten überholen sich nicht
  - voll-duplex
- jeder Prozessor kann maximal gleichzeitig
  - eine Nachricht an einen beliebigen Empfänger senden
    - SEND(*smsg*, *to*)
  - eine Nachricht von einem beliebigen Sender empfangen
    - *rmsg* ← RECV(*from*)
  - oder beides gleichzeitig
    - *rmsg* ← SENDRECV(*smsg*, *to*, *from*)
- wer MPI kennt, möchte mehr spezifiziert haben …

Kostenmodell für Nachrichtenaustausch

Dauer für Senden oder Empfangen von ℓ Bytes

$$T_{comm}(\ell) = T_{start} + \ell \cdot T_{byte}$$

- in der Praxis meist $T_{byte} \ll T_{start}$
- ignoriert z. B. «Abstand» zwischen Sender und Empfänger
  - auf gleicher CPU?
  - auf gleichem Mainboard?
  - in gleichem Rack?

Programmiermodell

- *SPMD:* single program multiple data
  - alle PEs führen das gleiche Programm aus
  - Unterscheidung durch «Ränge»
    - paarweise verschiedene PE-Nummern
- Beispiel:

```
parHelloWorld()
1  p = commSize()
2  iPE = commRank()
3  print("I am PE", iPE, "of", p, "PEs")
```

mögliche Ausgabe

"*I am PE* 1 *of* 3 *PEs*"
"*I am PE* 2 *of* 3 *PEs*"
"*I am PE* 0 *of* 3 *PEs*"

Überblick

Reduktion

- $\otimes$ sei binäre assoziative Operation auf Menge M
  - nicht notwendig kommutativ
  - Beispiele $x \otimes y = x + y$, $x \otimes y = \min(x, y)$, $x \otimes y = x$, $x \otimes y = y$
- für $x = (x_0, \ldots, x_{p-1}) \in M$ definiere $R_\otimes(x) = \bigotimes_{i<p} x_i = x_0 \otimes \cdots \otimes x_{p-1}$

Parallele Berechnung für Reduktion

Satz

Wenn $\otimes$ eine binäre assoziative Operation ist und
p Elemente $x_0, \ldots, x_{p-1}$ auf p PEs verteilt sind,
dann kann man $\bigotimes_{i<p} x_i$ in Zeit $\Theta(\log p)$ auf PE 0 berechnen.

Parallele Reduktion: Algorithmusidee

- für $0 \leq i < p$ liege x_i auf PE i
- Idee für den Fall $p = 2^k$:

$$\bigotimes_{i<p} x_i = \bigotimes_{i<p/2} x_i \otimes \bigotimes_{i<p/2} x_{i+p/2}$$

  und Rekursion
- (Bild live)

Parallele Reduktion: Algorithmus

```
1   active ← 1
2   sum ← x_i
3   for k ← 0 to ⌊log p⌋
4       if active
5           if bit k of i
6               SEND(sum, i − 2^k)  // bit k is 0
7               active ← 0
8           else if i + 2^k < p
9               sum′ ← RECV(i + 2^k)
10              sum ← sum ⊗ sum′
11  // result is in sum of PE 0
```

zu jedem SEND ein RECV und umgekehrt

Parallele Reduktion: Algorithmus (2)

Bild live

Parallele Reduktion: Analyse

Reduktion von n Elementen

- sequenziell: Zeit $\Theta(n)$ (optimal)
- auf $p = n$ PEs
  - Laufzeit $\Theta(\log n)$
  - Beschleunigung $\Theta(\frac{n}{\log n})$
    - «ideal» wäre Beschleunigung $\Theta(p) = \Theta(n)$
- wie kann man Beschleunigung noch verbessern?

Parallele Reduktion mit $p < n$

- jedes PE i hat n/p Datenelemente
- berechnet zuerst $s_i = \bigotimes$ der lokalen Elemente
- anschließend parallele Reduktion der s_i

- Laufzeit $\Theta(n/p) + \Theta(\log p)$
- Beschleunigung

$$\frac{\Theta(n)}{\Theta(n/p + \log p)}$$

- für $p \in \Theta(n/\log n)$ ist die Beschleunigung $\Theta(p)$

Anmerkungen

- Bäume sind des öfteren nützlich
- *Brent's Prinzip:* «Durch Verringerung der Prozessorzahl p kann man die Beschleunigung in die Nähe von p bringen.»
  - gleich noch etwas präziser

Überblick

Analyse paralleler Programme (1)

was interessiert?

- Laufzeit
- Beschleunigung
- verrichtete Arbeit

Analyse paralleler Programme (1)

was interessiert?

- Laufzeit
- Beschleunigung
- verrichtete Arbeit

- $T_{par}(I, p)$ Laufzeit Probleminstanz I bearbeitet mit p PEs
- $T_{seq}(I)$ Laufzeit Probleminstanzen Größe n mit «bestem (bekannten) Algorithmus»
- «im allgemeinen» $T_{seq}(I) < T_{par}(I, 1)$
  - aber par. Alg. kann leicht (?) modifiziert werden:
    if $p = 1$ **then** $A_{seq}(I)$ **else** $A_{par}(I)$ **fi**

Analyse paralleler Programme (2)

- *Beschleunigung* (*speedup*)

$$S(I,p) = \frac{T_{seq}(I)}{T_{par}(I,p)}$$

- Beschleunigung vergröbert

$$S(n,p) = \inf\{S(I,p) \mid n = |I|\}$$

- bequemer Spezialfall: für Instanzen I, I' gleicher Größe n ist $T_{par}(I,p) = T_{par}(I',p) = T_{par}(n,p)$ und $T_{seq}(I) = T_{seq}(I') = T_{seq}(n)$
- dann $S(n,p) = T_{seq}(n)/T_{par}(n,p)$
  - *Im allgemeinen gilt das nicht!* (Übung)

Analyse paralleler Programme (3)

- Simulation von p Prozessoren durch 1 Prozessor:
  - reihum immer «ein Stückchen» von Prozessor i
- «möglichst guter» seq. Alg. nie langsamer als $O(p \cdot T_{par}(I, p))$
- also stets $\frac{T_{seq}(I)}{T_{par}(I,p)} \in O(p)$ und $S(n, p) \in O(p)$

Analyse paralleler Programme (4)

- *Effizienz:*

$$E(n,p) = \frac{S(n,p)}{p}$$

- wegen $S(n,p) \in O(p)$ stets $E(n,p) \in O(1)$
- kurios, aber möglich: $E(p) > 1$ «superlinearer Speedup»
  - wie kann das sein?
- Ziel: $E(p) \in \Theta(1)$
-

$$E(n,p) = \frac{T_{par}(n,p)}{p \cdot T_{seq}(n)}$$

Analyse paralleler Programme (5)

- *Arbeit:*

$$W(n,p) = p \cdot T_{par}(n,p)$$

- dann

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{seq}(n)}{T_{par}(n,p) \cdot p} = \frac{T_{seq}(n)}{W(n,p)}$$

Analyse der parallelen Reduktion mit $p < n$

jetzt präziser formuliert

- sequenzielle Reduktion von n/p Elementen
- anschließend parallele Reduktion der p partiellen Summen

- Laufzeit $T_{par}(n,p) = \Theta(n/p) + \Theta(\log p)$
- Beschleunigung $S(n,p) = \frac{\Theta(n)}{\Theta(n/p+\log p)}$
- Effizienz $E(n,p) = \frac{\Theta(n)}{\Theta(n+p\log p)}$
- für $p = n$:
  $S(n,n) \in \Theta(\frac{n}{\log n})$ und $E(n,n) \in \Theta(\frac{1}{\log n})$
- für $p \in \Theta(n/\log n)$:
  $S(n,p) = \Theta(p)$ und $E(n,p) \in \Theta(1)$
- größere Effizienz für kleinere p Brents Prinzip

Überblick

Präfixsummen

- $\otimes$ sei binäre assoziative Operation auf Menge M
  - nicht notwendig kommutativ
  - evtl. neutrales Element e
- für $x = (x_0, \ldots, x_{p-1}) \in M$ definiere
  $P_\otimes(x) = y = (y_0, \ldots, y_{p-1})$ wobei für $0 \leq i < p$:
  - $y_i = \bigotimes_{k \leq i} x_k$ oder
  - $y_i = \bigotimes_{k < i} x_k$ (falls neutrales Element e; $\bigotimes_{k<0} x_i = e$)
- also
  - $y_0 = x_0$ und $y_{i+1} = y_i \otimes x_{i+1}$,
    also
  - $y_1 = x_0 \otimes x_1$
  - $y_2 = x_0 \otimes x_1 \otimes x_2$
  - $y_3 = x_0 \otimes x_1 \otimes x_2 \otimes x_3$
  - …

Präfixsummen: Hyperwürfel-Algorithmus

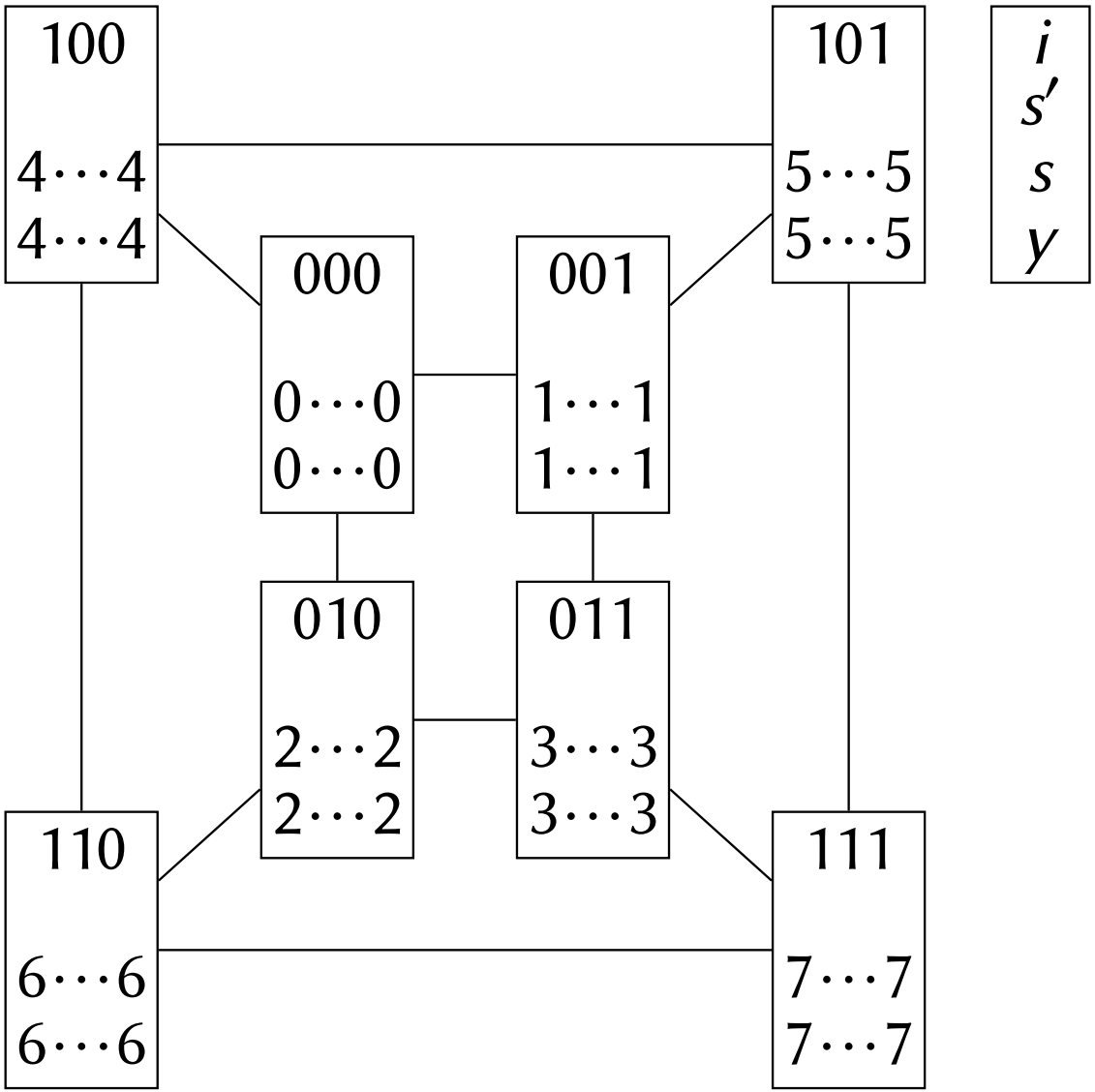
- um es bequem zu haben:
  - $p = n = 2^d$
  - kommutatives $\otimes$
- PE-«Koordinaten» $0 \leq i < 2^d$
  - repräsentiert als Bitvektoren $i = (i_{d-1} \cdots i_0)$ mit $i_j \in \{0, 1\}$
    - i_k: das k-te Bit von rechts in i
  - Kommunikation $i \leftrightarrow i'$ nur bei Hammingdistanz 1, also $i' = i \oplus 2^k \pmod{2^d}$
    - $\rightsquigarrow$ Hyperwürfel

Präfixsummen: Hyperwürfel-Algorithmus

für den Fall $p = n = 2^d$ und kommutatives $\otimes$

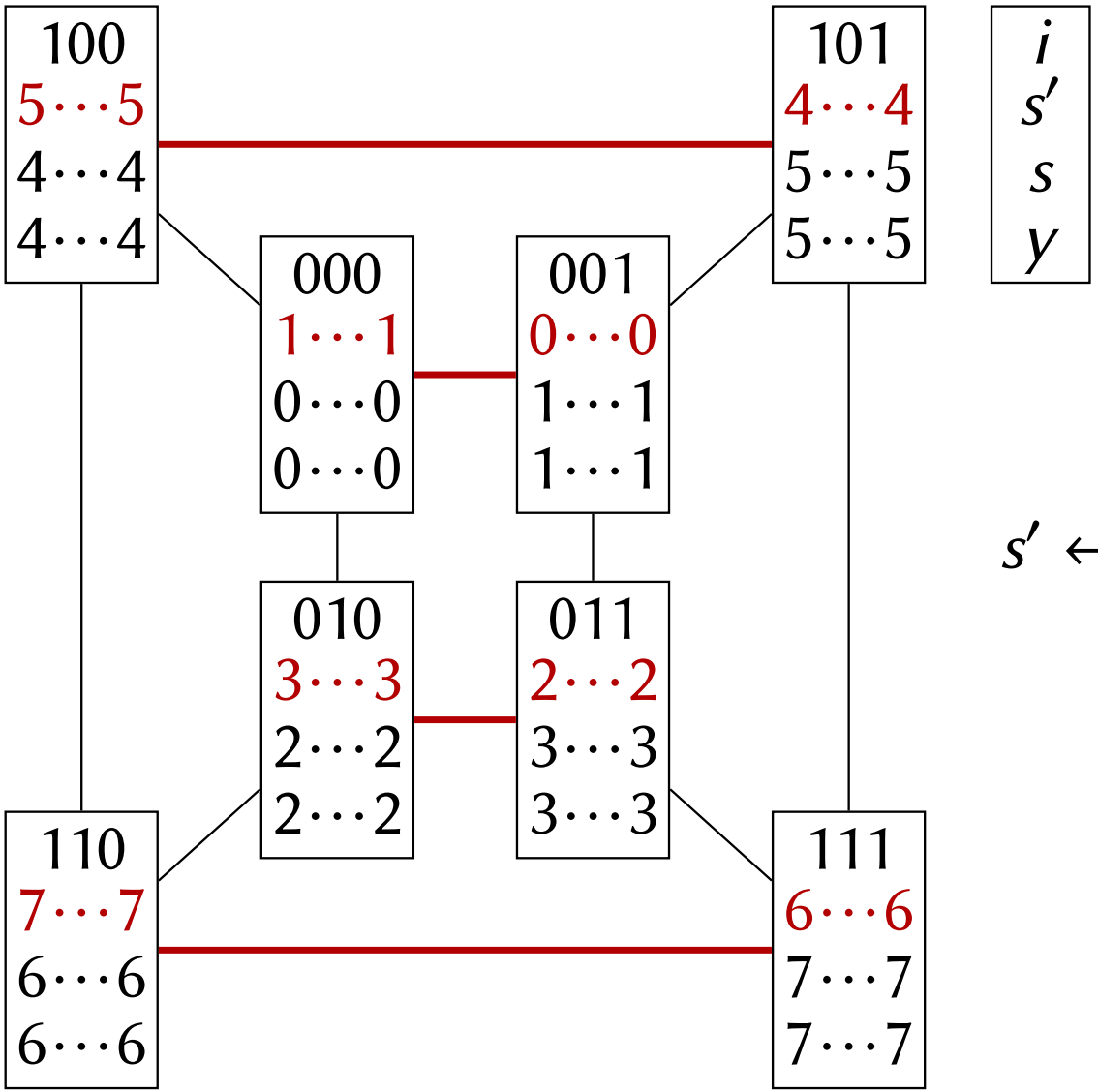
```
PREFIXSUM(x, ⊗)
1  y ← x  // auf PE i liegt x_i
2  s ← x  // für Summe von Elementen in Unterwürfel
3  for k ← 0 to d − 1
4        s' ← SENDRECV(s, i ⊕ 2^k, i ⊕ 2^k)
5        s ← s ⊗ s'
6        if i_k = 1
7              y ← y ⊗ s'
8  // auf PE i liegt y_i = x_0 ⊗ ··· ⊗ x_i
```

Präfixsummen: Hyperwürfel-Algorithmus (Init)



$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

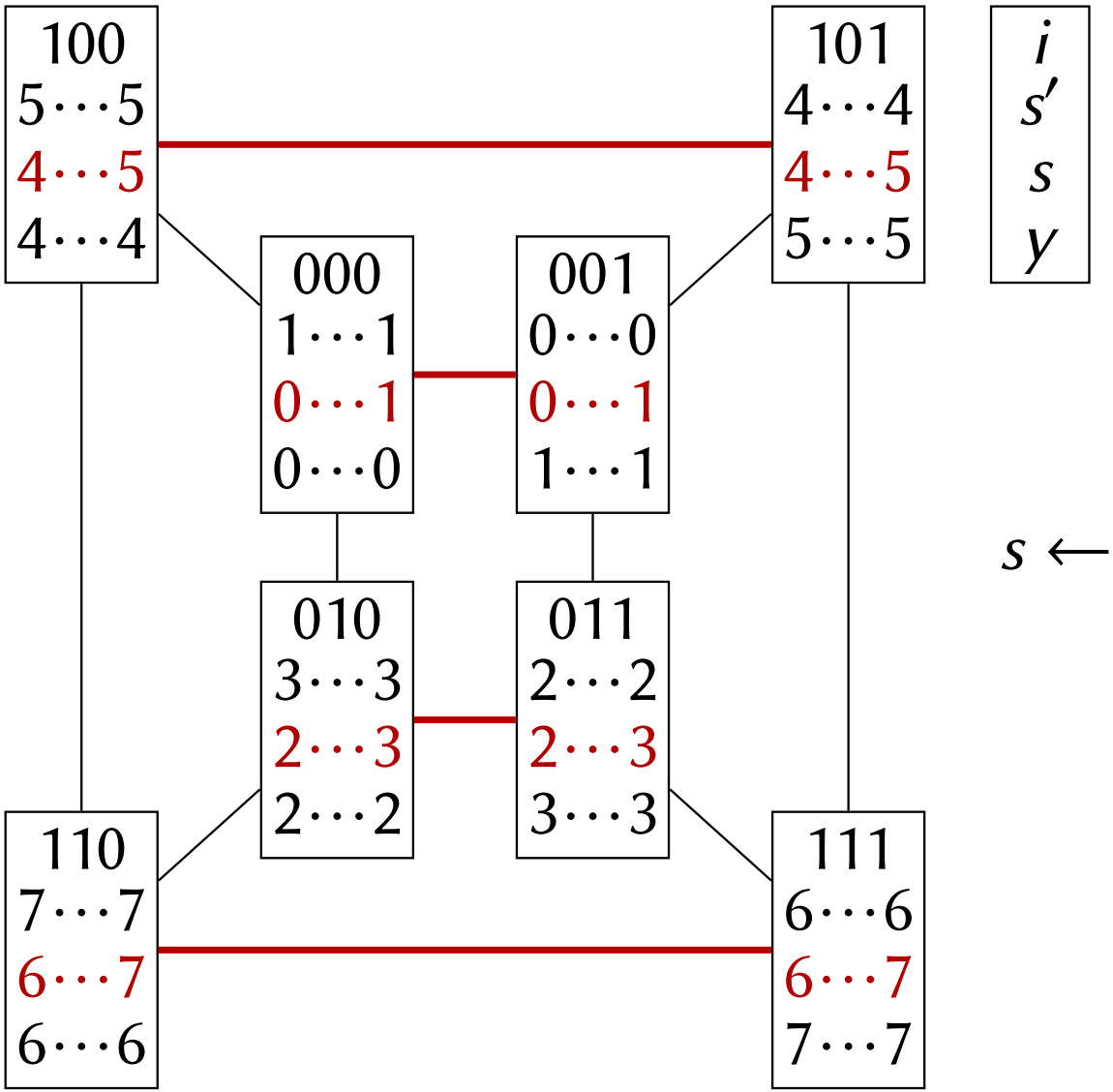
Präfixsummen: Hyperwürfel-Algorithmus ($k = 0$)



$s' \leftarrow \text{SENDRECV}(s, i \oplus 2^0, i \oplus 2^0)$

$k \cdots m = \bigotimes_{j=k}^{m} x_j$

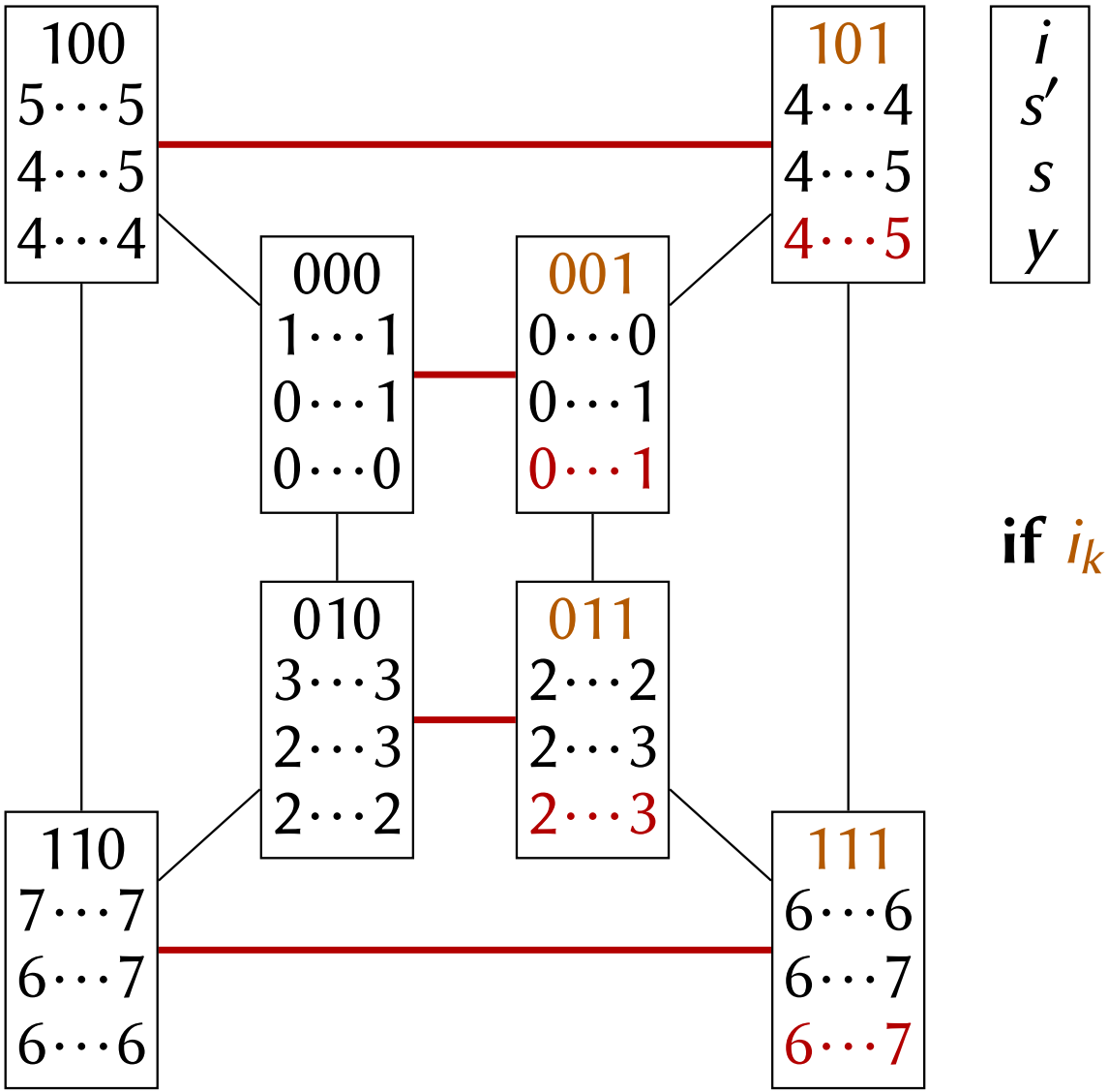
Präfixsummen: Hyperwürfel-Algorithmus ($k = 0$)



$s \leftarrow s \otimes s'$

$k \cdots m = \bigotimes_{j=k}^{m} x_j$

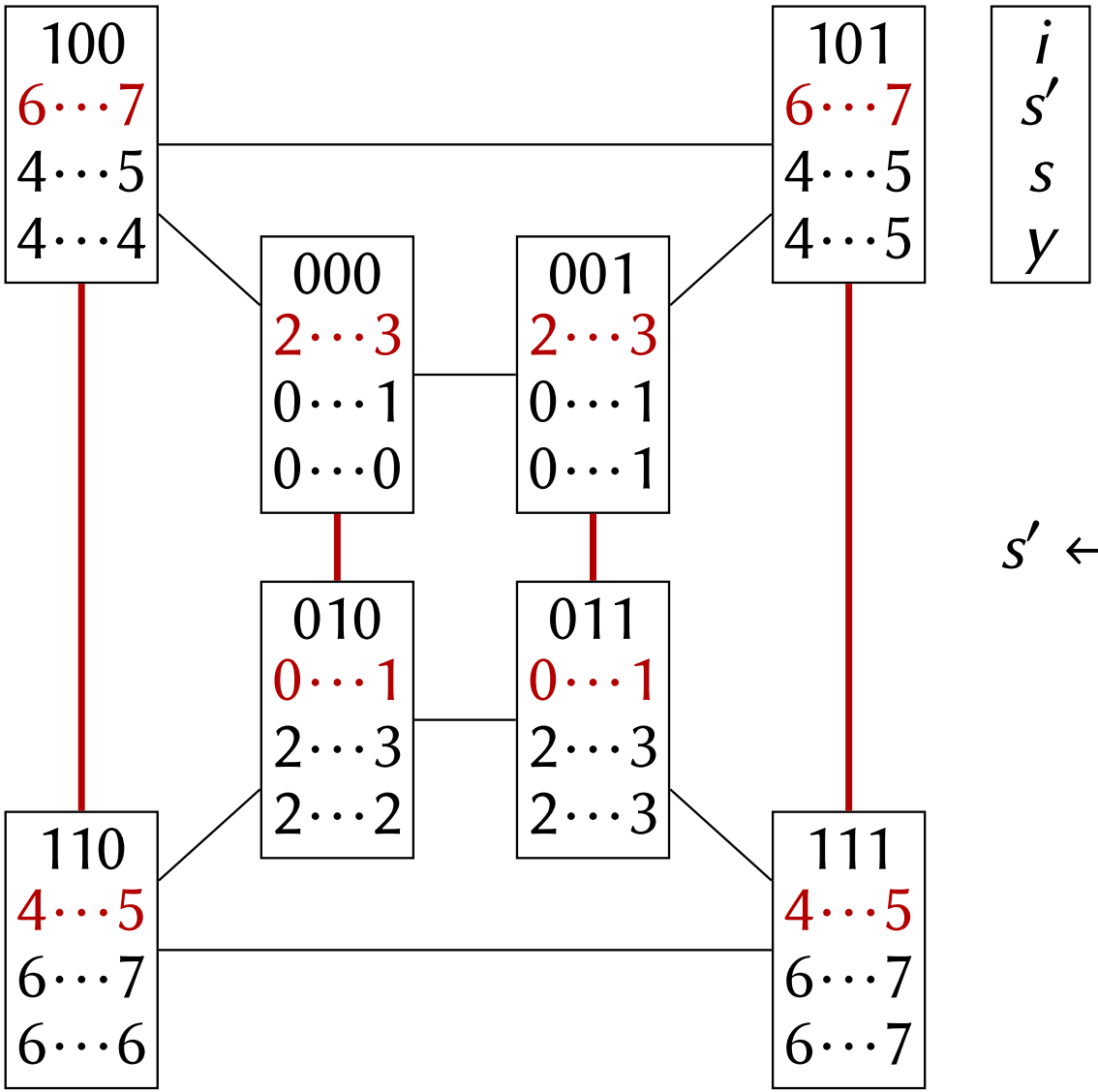
Präfixsummen: Hyperwürfel-Algorithmus ($k = 0$)



if $i_k = 1$ **then** $y \leftarrow y \otimes s'$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

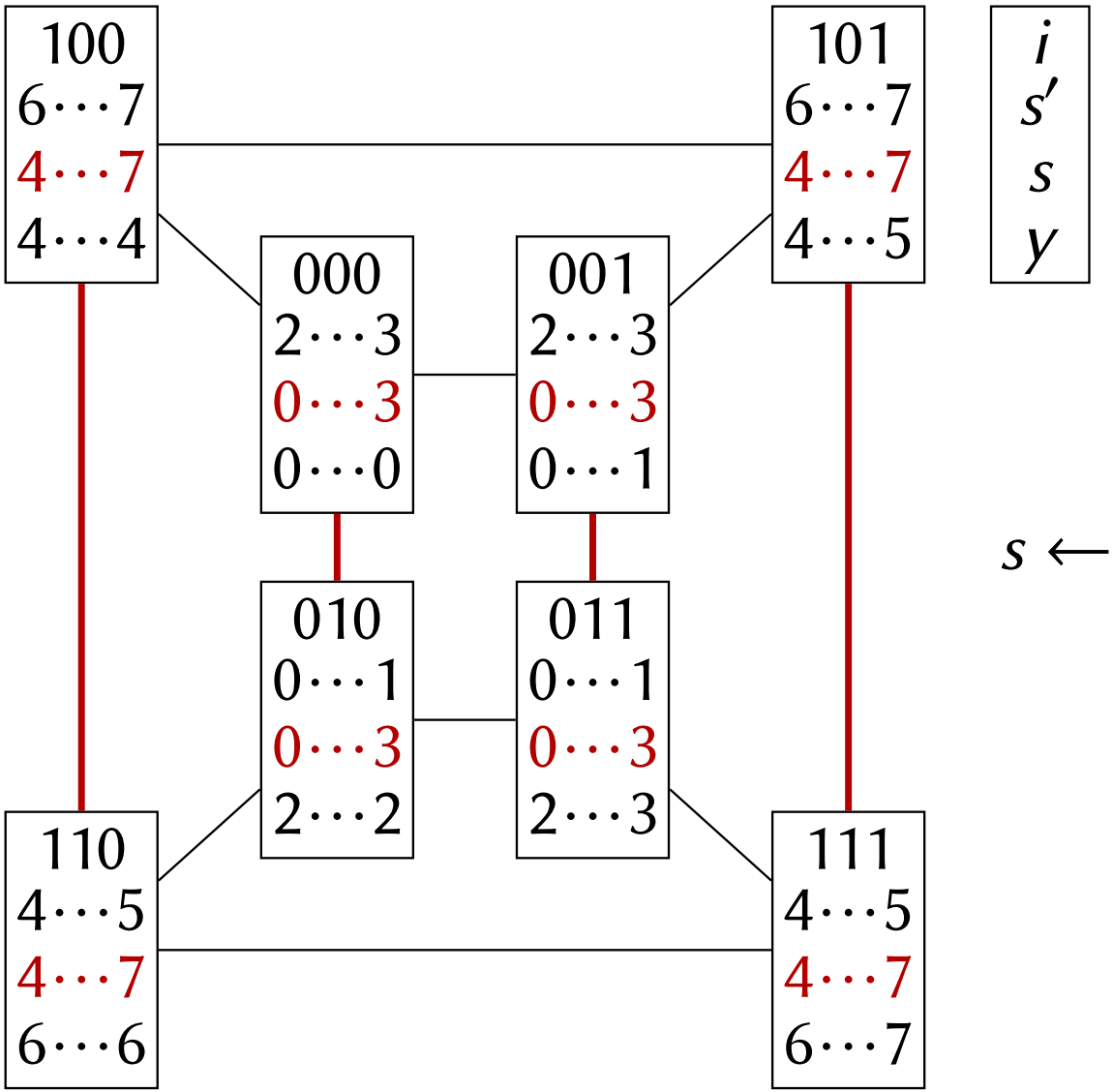
Präfixsummen: Hyperwürfel-Algorithmus ($k = 1$)



$s' \leftarrow \text{SENDRECV}(s, i \oplus 2^1, i \oplus 2^1)$

$k \cdots m = \bigotimes_{j=k}^{m} x_j$

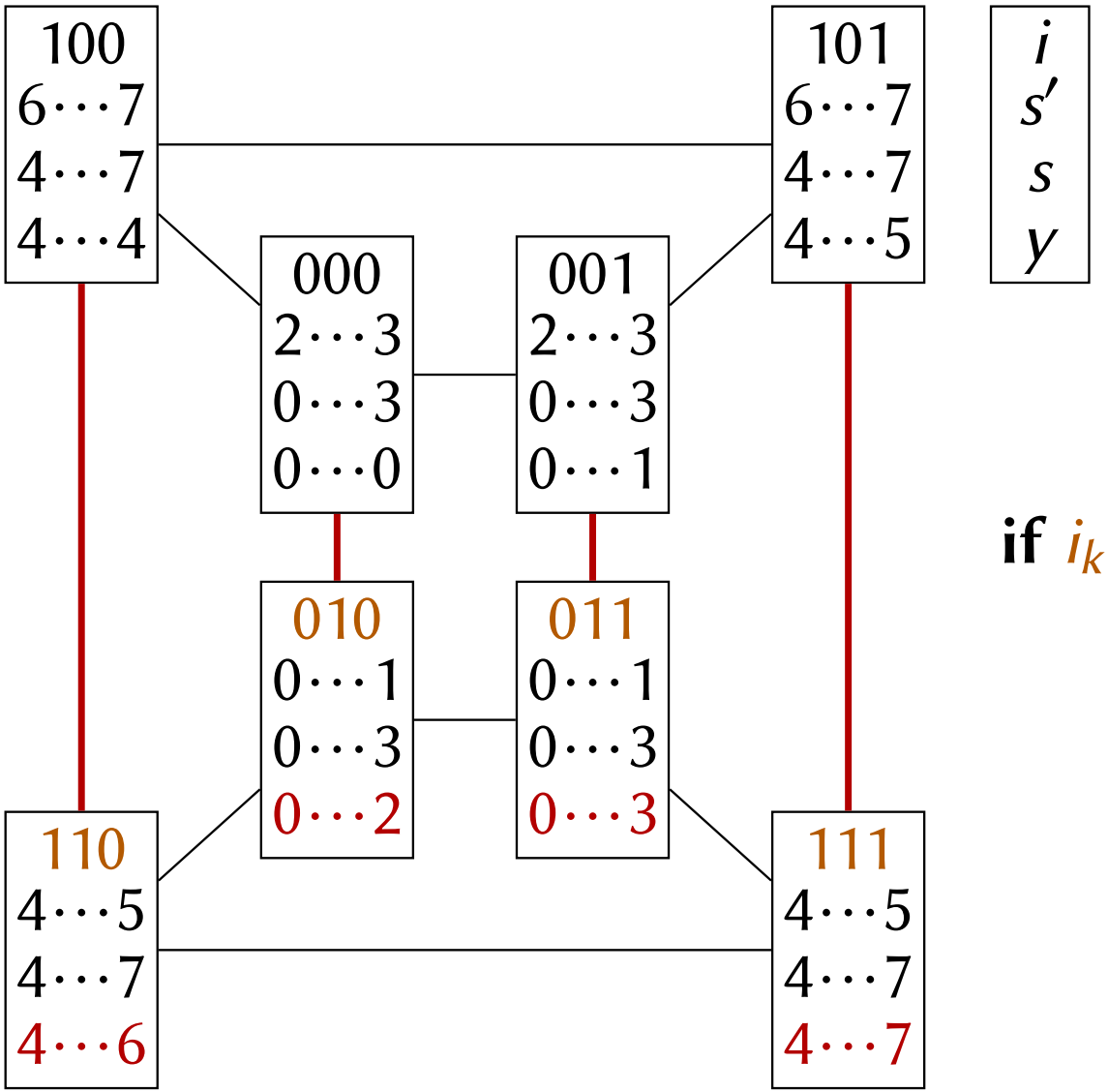
Präfixsummen: Hyperwürfel-Algorithmus ($k = 1$)



$$s \leftarrow s \otimes s'$$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

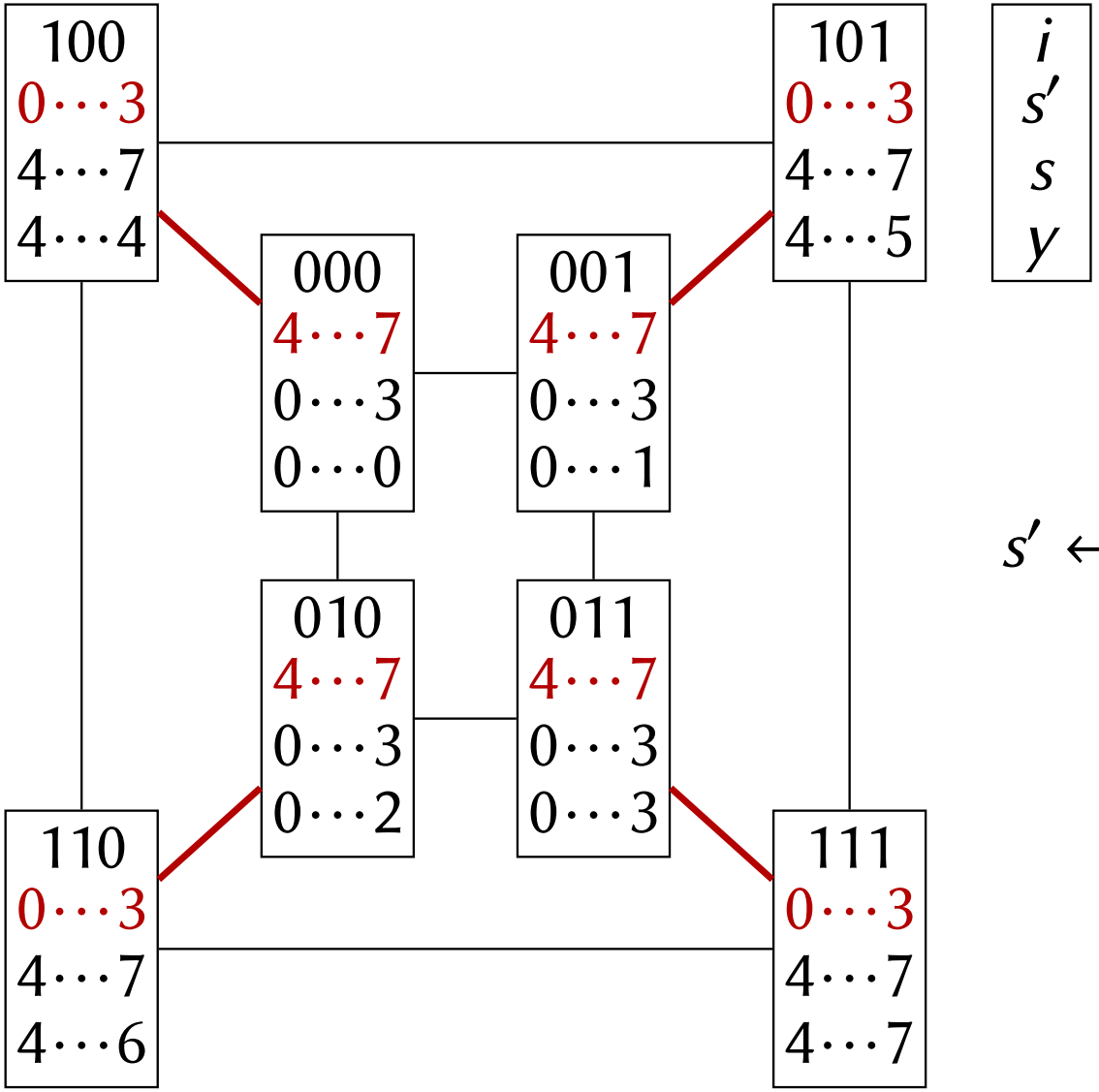
Präfixsummen: Hyperwürfel-Algorithmus ($k = 1$)



if $i_k = 1$ **then** $y \leftarrow y \otimes s'$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

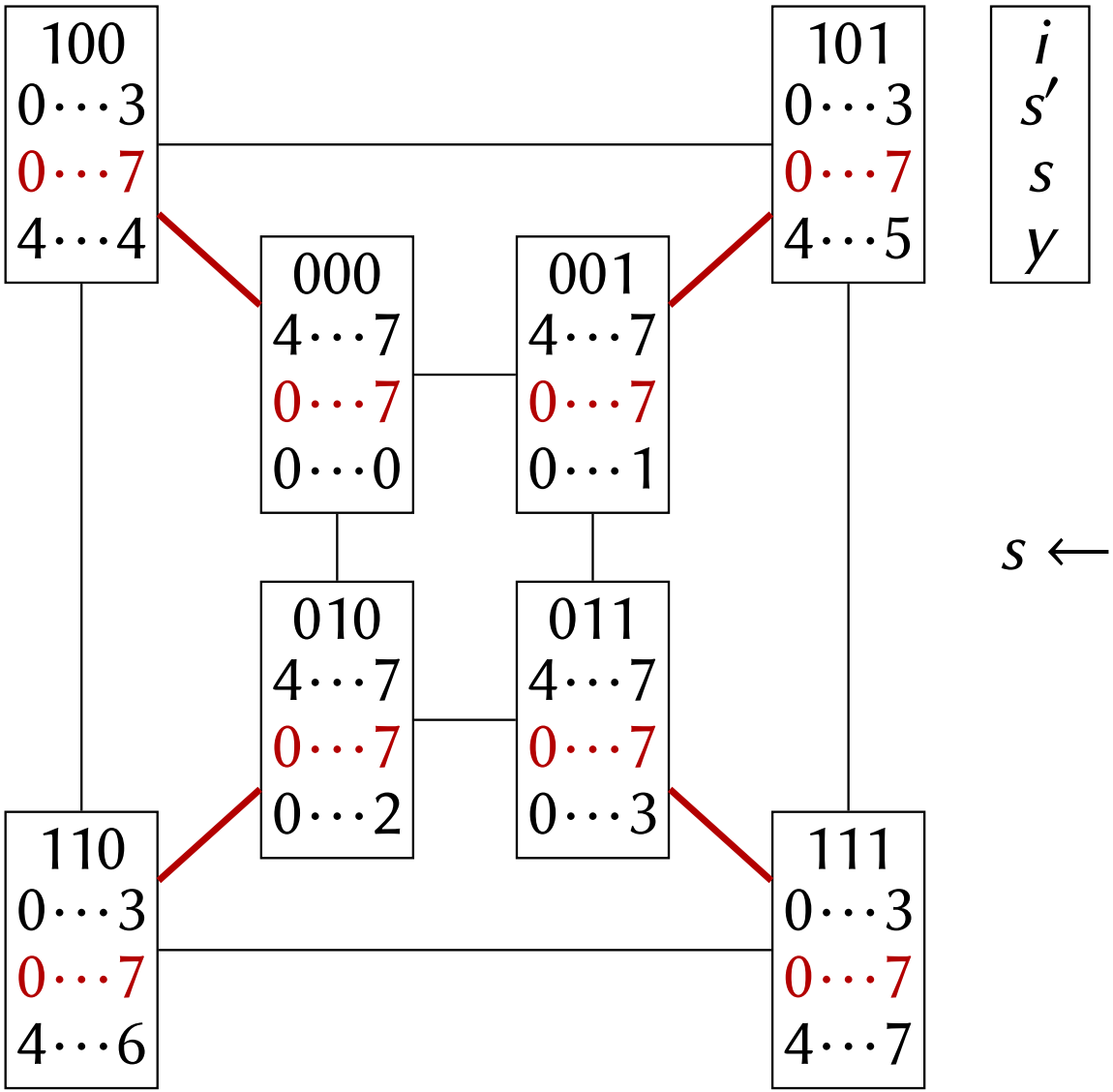
Präfixsummen: Hyperwürfel-Algorithmus ($k = 2$)



$$s' \leftarrow \text{SENDRECV}(s, i \oplus 2^2, i \oplus 2^2)$$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

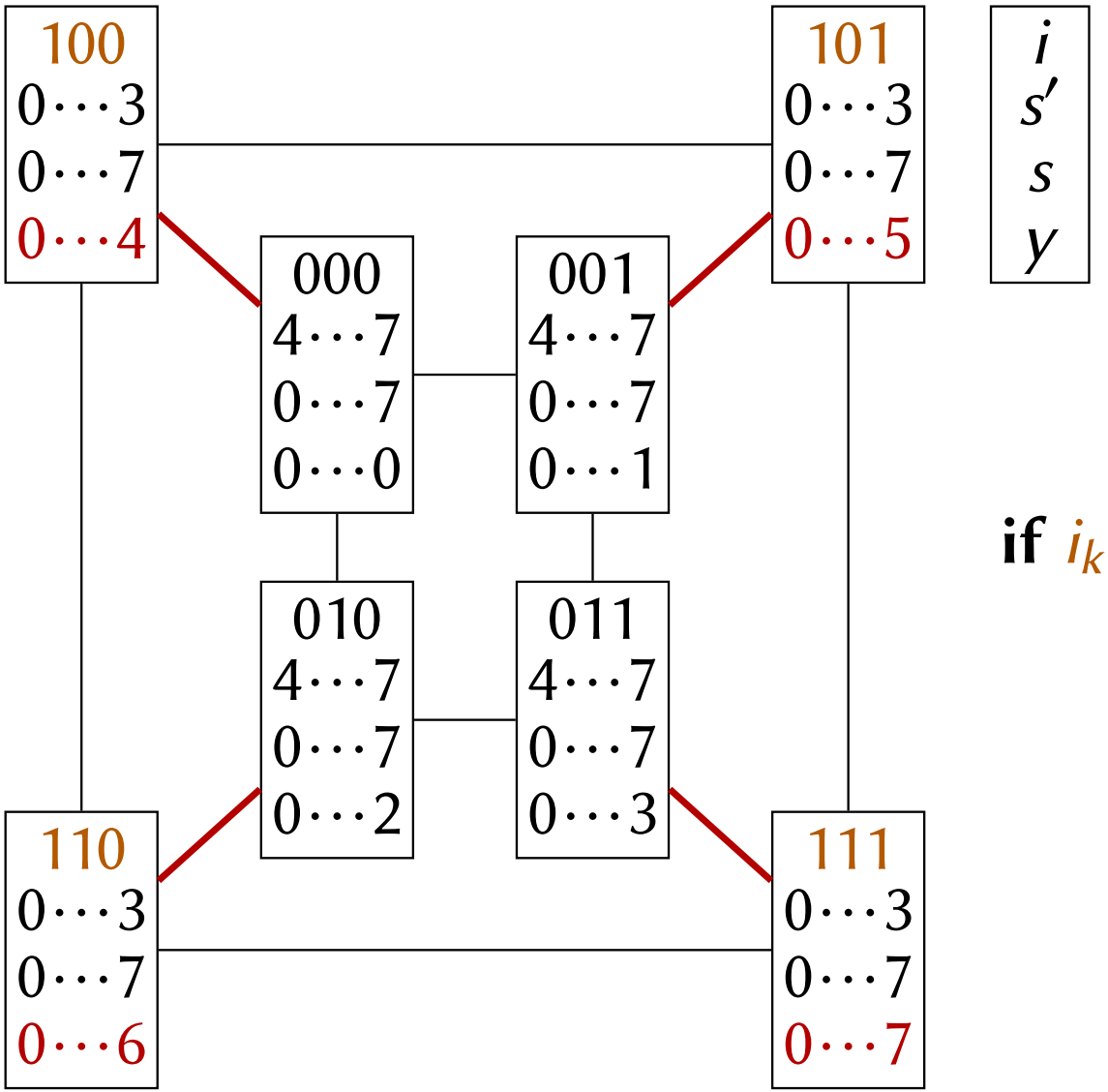
Präfixsummen: Hyperwürfel-Algorithmus ($k = 2$)



$$s \leftarrow s \otimes s'$$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

Präfixsummen: Hyperwürfel-Algorithmus ($k = 2$)



if $i_k = 1$ **then** $y \leftarrow y \otimes s'$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

Hyperwürfel-Algorithmus: Korrektheit

schreibe $i|0^k$ für $(i_{d-1}, \ldots, i_k, 0, \ldots, 0)$ und
analog $i|1^k$ für $(i_{d-1}, \ldots, i_k, 1, \ldots, 1)$

1 $y \leftarrow x$ // auf PE i liegt x_i
2 $s \leftarrow x$ // für Summe von Elementen in Unterwürfel
3 **for** $k \leftarrow 0$ **to** $d-1$
// Invariante: $s_i = \bigotimes_{a_i}^{b_i} x_i$ und $y_i = \bigotimes_{a_i}^{i} x_i$
// mit $a_i = i|0^k$, $b_i = i|1^k$
4 $s' \leftarrow \text{SENDRECV}(s, i \oplus 2^k, i \oplus 2^k)$
5 $s \leftarrow s \otimes s'$
6 **if** $i_k = 1$
7 $y \leftarrow y \otimes s'$
8 // auf PE i liegt $y_i = x_0 \otimes \cdots \otimes x_i$

Hyperwürfel-Algorithmus: Analyse

- Laufzeit

$$T_{prefix} \in O((T_{start} + \ell \cdot T_{byte}) \cdot \log p)$$

- für $T_{start} \ll \ell T_{byte}$ nicht optimal
  - $\Theta(T_{start} \cdot \log p + \ell \cdot T_{byte})$ erreichbar
  - $\rightsquigarrow$ Vorlesung «Parallele Algorithmen»

- analog übrigens auch schon bei Reduktion

Parallele Präfixsummen für nicht-kommutatives $\otimes$

kleine Korrekturen an zwei Stellen:

1 $y \leftarrow x$ // auf PE i liegt x_i
2 $s \leftarrow x$ // für Summe von Elementen in Unterwürfel
3 **for** $k \leftarrow 0$ **to** $d-1$
4 $s' \leftarrow \text{SENDRECV}(s, i \oplus 2^k, i \oplus 2^k)$
// statt $s \leftarrow s \otimes s'$
5 **if** $i_k = 1$
6 $s \leftarrow s' \otimes s$
7 **else** $s \leftarrow s \otimes s'$
8 **if** $i_k = 1$
9 $y \leftarrow s' \otimes y$ // statt $y \otimes s'$
10 // auf PE i liegt $y_i = x_0 \otimes \cdots \otimes x_i$

Parallele Präfixsummen für nicht-kommutatives $\otimes$

- $x \otimes y = x$: parallele Präfixsumme *eine* Möglichkeit
  - in Zeit $\log p$
  - x_0 an alle Prozessoren zu verteilen
- *Broadcast*
  - $rval \leftarrow \textsc{bcast}(val, from)$
  - Anwendung «wie in MPI»
  - muss von *allen* Prozessoren aufgerufen werden

Überblick

Aufgabenvarianten

wo liegen am Anfang die Daten

- alle n Elemente auf PE 0
  - jedes Element von PE 0 mindestens einmal angefasst
  - $\rightsquigarrow$ Laufzeit in $\Omega(n)$
- je n/p Elemente auf PE i, $\quad 0 \leq i < p$
  - interessanter

Verteilte Eingabedaten

- zunächst einfacher Fall $p = n$
  - Prozessor i hat Eingabeelement x_i
- Grundidee von Quicksort beibehalten
  - ein Element pv als Pivot wählen
  - Elemente umverteilen
    - Elemente kleiner als pv auf Prozessoren mit kleinen Rängen
    - Elemente größer als pv auf Prozessoren mit großen Rängen
  - parallele Rekursion: gleichzeitig
    - Elemente kleiner als pv rekursiv analog und
    - Elemente größer als pv rekursiv analog sortieren

Theoretiker-Quicksort

Theoretiker-Quicksort Teil 0: Vorbereitungen

theoQSort0(x)

```
1  i ← commRank()
2  p ← commSize()
   // PE i hat Element x_i
3  theoQSort(x, 0, p − 1)
```

Theoretiker-Quicksort

Theoretiker-Quicksort Teil 1: kleine Elemente zählen

THEOQSORT$(x, \ldots)$

1 **if** $i = 0$
2 $\quad ipv \leftarrow$ RANDINT$(0, p)$ // expected log. recursion levels
3 $ipv \leftarrow$ BCAST$(ipv, 0)$
4 $pv \leftarrow$ BCAST(x, ipv)
5 $small \leftarrow x \leq pv$ // 1 iff $x \leq pv$; 0 otherwise
6 $j \leftarrow$ PREFIXSUM$(small, +)$
7 $p' \leftarrow$ BCAST$(j, p-1)$ // nr of elements $\leq pv$

Theoretiker-Quicksort

Theoretiker-Quicksort Teil 2: Datenumverteilung und Rekursion

```
    small ← x ≤ pv                          // 1 iff x ≤ pv; 0 otherwise
    j ← PREFIXSUM(small, +)                 // «inclusive» prefix sum
                                            // as described before
    p' ← BCAST(j, p − 1)                    // nr of elements ≤ pv
1   if small = 1
2       SEND(x, j − 1)                      // assume nonblocking SEND
3   else SEND(x, p' + i − j)                // assume nonblocking SEND
                                            // i − j small elements left of i
4   x ← RECV(ANY)
5   recursive THEOQSORT of «left»/«right» part
```

Theoretiker-Quicksort: Laufzeit

- erwartete Rekursionstiefe $O(\log p)$
- jeweils Zeit $O(T_{start} \log p)$
- insgesamt erwartete Zeit $O(T_{start}(\log p)^2)$

11 Fortgeschrittene Datenstrukturen

Hier am Beispiel von Prioritätslisten.

Weitere Beispiele:

- Monotone ganzzahlige Prioritätslisten — Kapitel kürzeste Wege
- perfektes Hashing — siehe Buch
- Suchbäume mit fortgeschrittenen Operationen — siehe Buch
- Externe Priortätslisten — Kapitel externe Algorithmen
- Geometrische Datenstrukturen — Kapitel geom. Algorithmen

11.1 Adressierbare Prioritätslisten

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

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

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

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

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

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

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

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

Adressierbare Prioritätslisten: Anwendungen

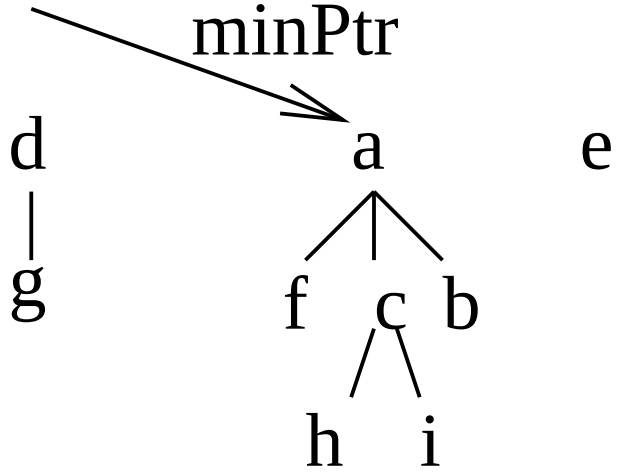
- Dijkstras Algorithmus für kürzeste Wege
- Jarník-Prim-Algorithmus für minimale Spannbäume
- Bei uns: Hierarchiekonstruktion für Routenplanung
- Bei uns: Graphpartitionierung
- Bei uns: disk scheduling

Allgemein:

Greedy-Algorithmen, bei denen sich Prioritäten (begrenzt) ändern.

Grundlegende Datenstruktur

Ein Wald heap-geordneter Bäume

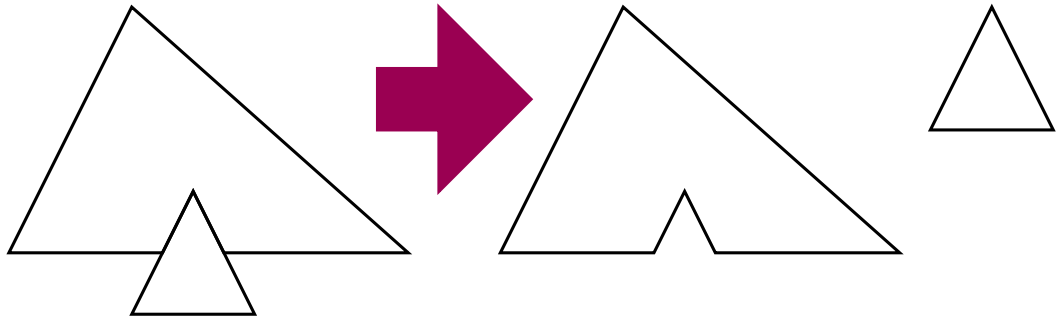


Verallgemeinerung gegenüber binary heap:

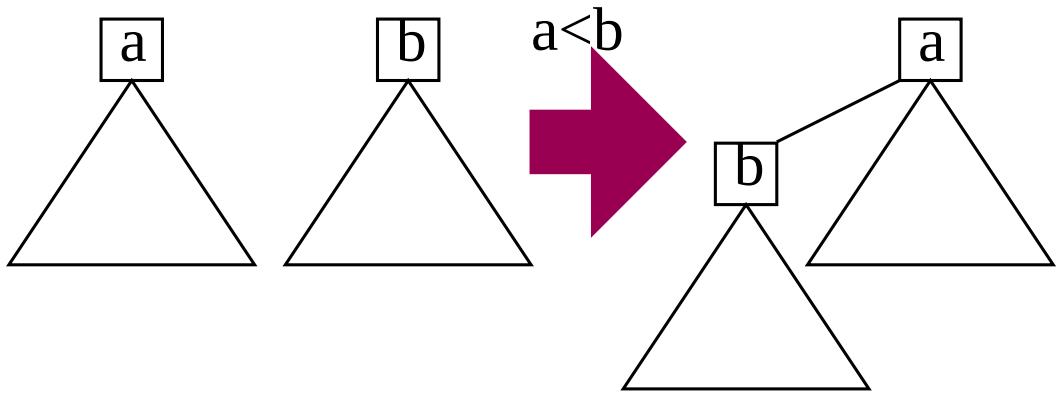
- Baum $\rightarrow$ Wald
- binär $\rightarrow$ beliebige Knotengrade

Wälder Bearbeiten

Cut:



Link:



union(a,b): link$(\min(a,b),\max(a,b))$

Pairing Heaps (Paarungs-Haufen??)

[Fredman Sedgewick Sleator Tarjan 1986]

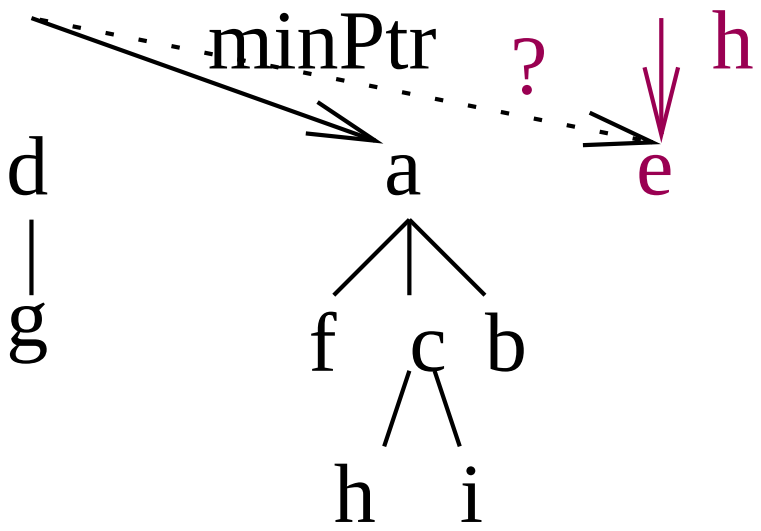
Procedure insertItem(h : Handle)

newTree(h)

Procedure newTree(h : Handle)

forest:= forest $\cup \{h\}$

if $*h < \min$ **then** minPtr:= h



Pairing Heaps

Procedure decreaseKey(h : Handle, k : Key)

key(h):= k

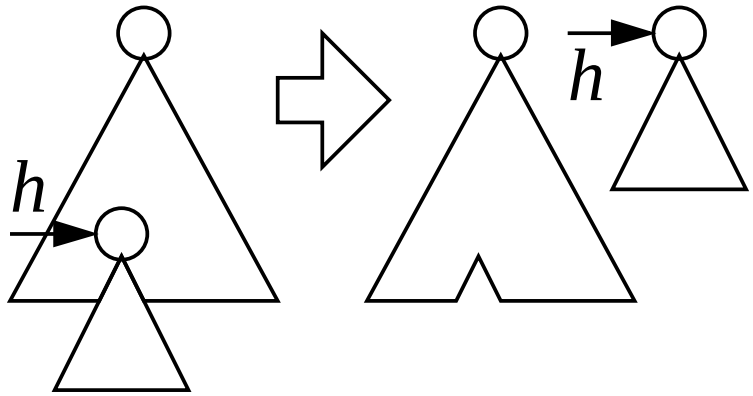
if h is not a root **then** cut(h)

else update minPtr

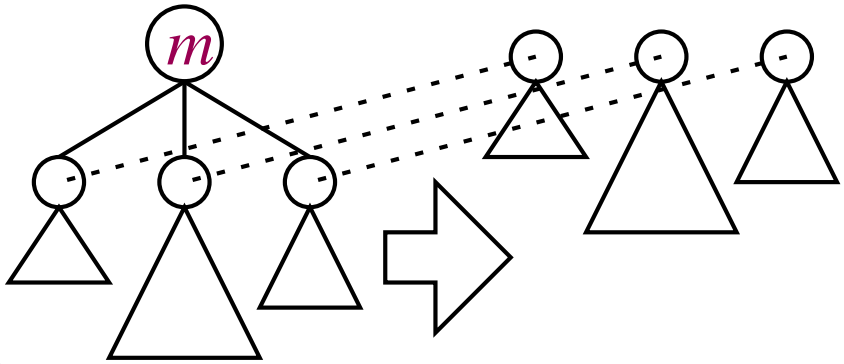
Procedure cut(h : Handle)

remove the subtree rooted at h

newTree(h)



Pairing Heaps



Function deleteMin : Handle

m:= minPtr

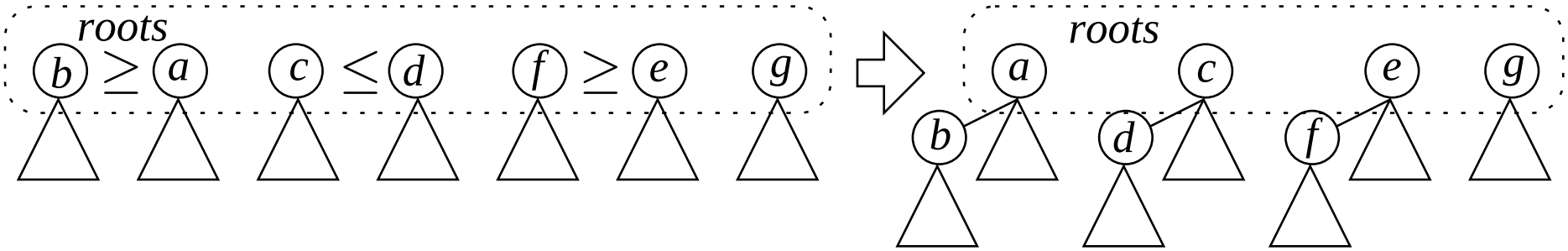
forest:= forest $\setminus \{m\}$

foreach child h of m **do** newTree(h)

perform pair-wise union operations on the roots in forest

update minPtr

return m



Pairing Heaps

Procedure merge(o : AdressablePQ)

if *minPtr $>$ $*$(o.minPtr) **then** minPtr:= o.minPtr

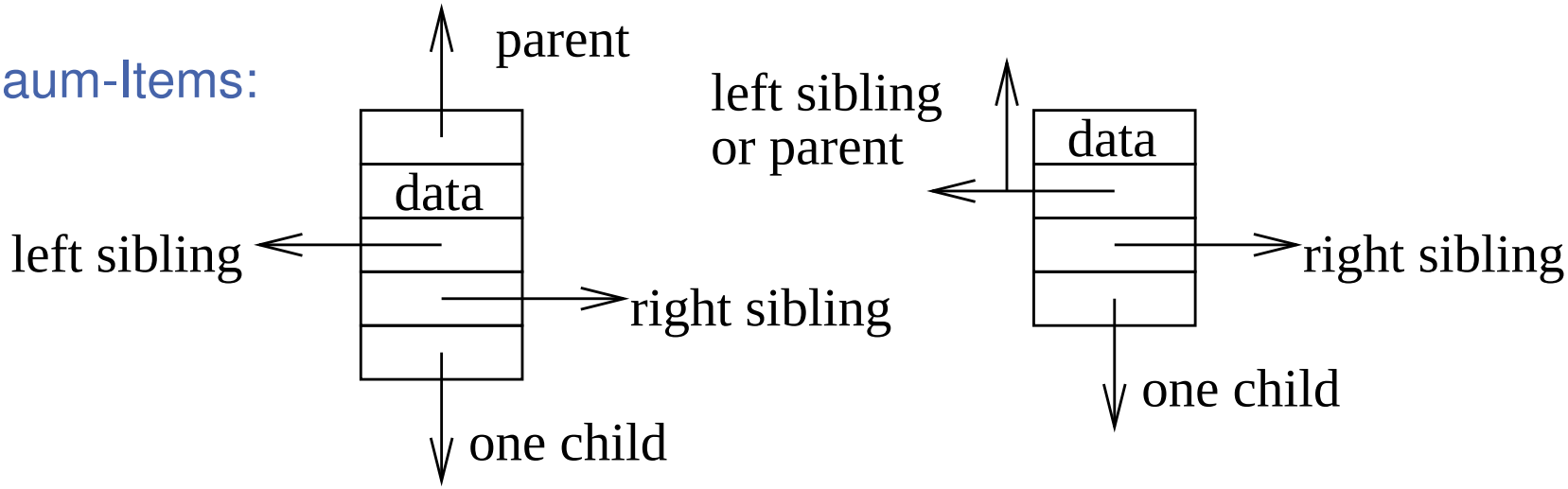
forest:= forest $\cup$ o.forest

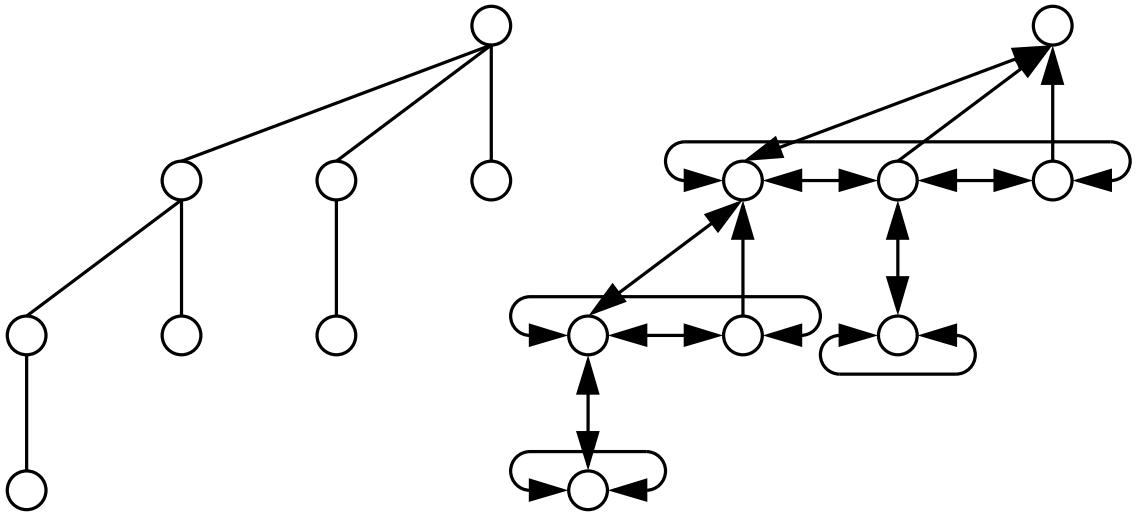
o.forest:= $\emptyset$

Pairing Heaps – Repräsentation

Wurzeln: Doppelt verkettete Liste

Baum-Items:





Pairing Heaps – Analyse

insert, merge: $O(1)$

deleteMin, remove: $O(\log n)$ amortisiert

decreaseKey: unklar! $O(\log\log n) \leq T \leq O(\log n)$ amortisiert.

In der Praxis sehr schnell.

Beweise: nicht hier.

Fibonacci Heaps [Fredman Tarjan 1987]

Rang: Anzahl (direkter) Kinder speichern

Vereinigung nach Rang: Union nur für gleichrangige Wurzeln

Markiere Knoten, die ein Kind verloren haben

Kaskadierende Schnitte: Schneide markierte Knoten

(die also 2 Kinder verloren haben)

Satz: Amortisierte Komplexität $O(\log n)$ für deleteMin, remove und $O(1)$ für alle anderen Operationen

(d.h. $Gesamtzeit = O(o + d \log n)$ falls

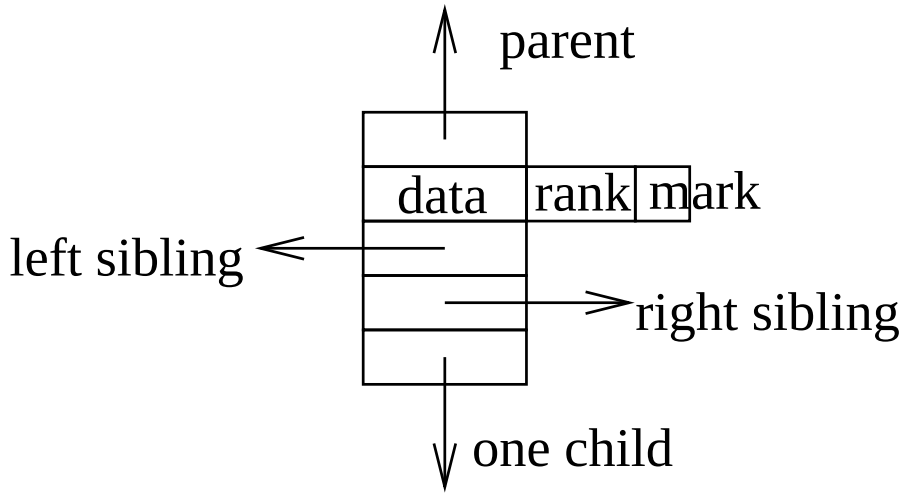
$d =$#deleteMin, $o =$#otherOps, $n = \max |M|$)

Repräsentation

Wurzeln: Doppelt verkettete Liste

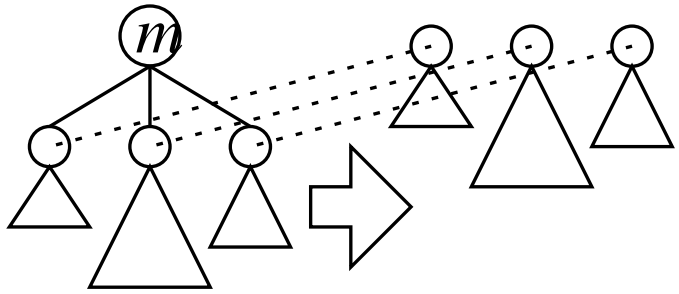
(und ein tempräres Feld für deleteMin)

Baum-Items:



insert, merge: wie gehabt. Zeit $O(1)$

deleteMin mit Union-by-Rank



Function deleteMin : Handle

m:= minPtr

forest:= forest $\setminus \{m\}$

foreach child h of m **do** newTree(h)

while $\exists a, b \in$ forest : rank$(a) =$ rank(b) **do**

union(a, b) // increments rank of surviving root

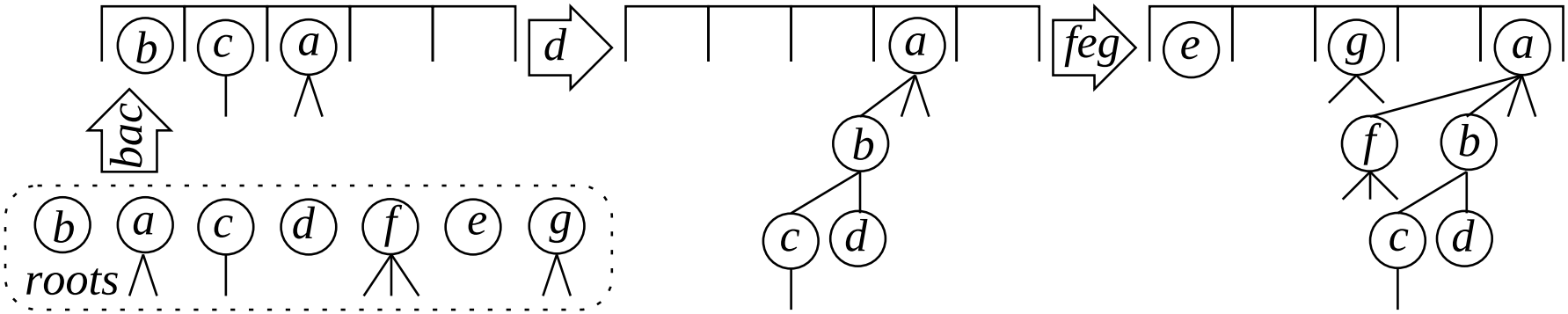
update minPtr

return m

Schnelles Union-by-Rank

Durch rank adressiertes Feld.

Solange link durchführen bis freier Eintrag gefunden.



Analyse: Zeit $O(\#\text{unions} + |\text{forest}|)$

Amortisierte Analyse von deleteMin

$$\text{maxRank} := \max_{a \in \text{forest}} \text{rank}(a) \text{ (nachher)}$$

Lemma: $T_{\text{deleteMin}} = O(\text{maxRank})$

Beweis: Kontomethode. Ein Token pro Wurzel

$\text{rank}(\text{minPtr}) \leq \text{maxRank}$

$\rightsquigarrow$ Kosten $O(\text{maxRank})$ für newTrees und neue Token.

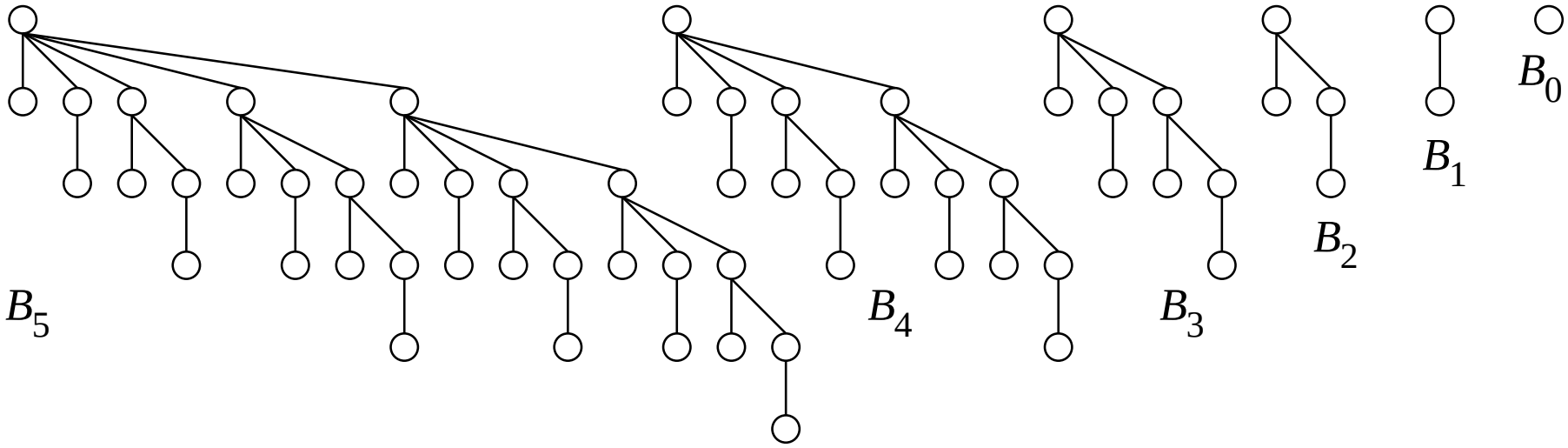
Union-by-rank: Token zahlen für

- union Operationen (ein Token wird frei) und
- durchlaufen alter und neuer Wurzeln.

Am Ende gibt es $\leq$ maxRank Wurzeln.

Warum ist maxRank logarithmisch? – Binomialbäume

$2^k + 1 \times$ insert, $1 \times$ deleteMin $\rightsquigarrow$ rank k



[Vuillemin 1978] PQ nur mit Binomialbäumen, $T_{\text{decreaseKey}} = \mathrm{O}(\log n)$.

Problem: Schnitte können zu kleinen hochrangigen Bäumen führen

Kaskadierende Schnitte

Procedure decreaseKey(h : Handle, k : Key)

key(h):= k

update minPtr

cascadingCut(h)

Procedure cascadingCut(h)

if h is not a root **then**

p:= parent(h)

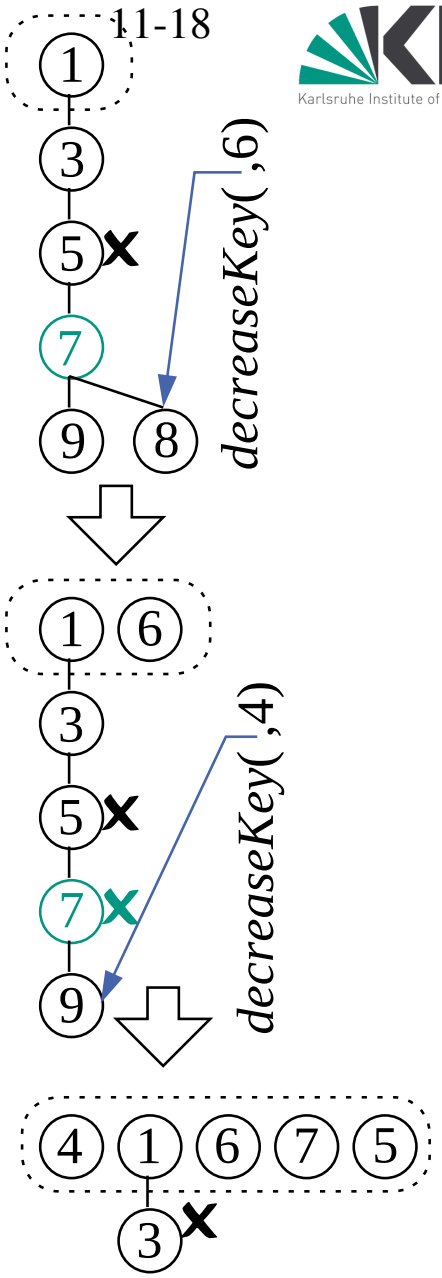
unmark h

cut(h)

if p is marked **then**

cascadingCut(p)

else mark p



Wir werden zeigen: kaskadierende Schnitte halten maxRank logarithmisch

Lemma: decreaseKey hat amortisierte Komplexität $O(1)$

Kontomethode: ($\approx$ 1 Token pro cut oder union)

1 Token für jede Wurzel

2 Token für jeden markierten Knoten

betrachte decreaseKey mit k konsekutiven markierten Vorgängern:

2k Token werden frei (unmarked nodes)

2 Token für neue Markierung

k+1 Token für Ausstattung der neuen Wurzeln

k+1 Token für Schnitte

Bleiben 4 Token $+O(1)$ Kosten für decreaseKey

Auftritt Herr Fibonacci

$$F_i := \begin{cases} 0 & \text{für i=0} \\ 1 & \text{für i=1} \\ F_{i-2} + F_{i-1} & \text{sonst} \end{cases}$$

Bekannt: $F_{i+2} \geq ((1+\sqrt{5})/2)^i \geq 1.618^i$ for all $i \geq 0$.

Wir zeigen:

Ein Teilbaum mit Wurzel v mit $\text{rank}(v) = i$ enthält $\geq F_{i+2}$ Elemente.

$\Rightarrow$

logarithmische Zeit für deleteMin.

Beweis:

Betrachte Zeitpunkt als das j-te Kind w_j von v hinzugelinkt wurde:

w_j und v hatten gleichen Rang $\geq j-1$ (v hatte schon $j-1$ Kinder)

$\text{rank}(w_j)$ hat höchsten um eins abgenommen (cascading cuts)

$\Rightarrow \text{rank}(w_j) \geq j-2$ und $\text{rank}(v) \geq j-1$

S_i:= untere Schranke für # Knoten mit Wurzel vom Rang i:

$S_0 = 1$

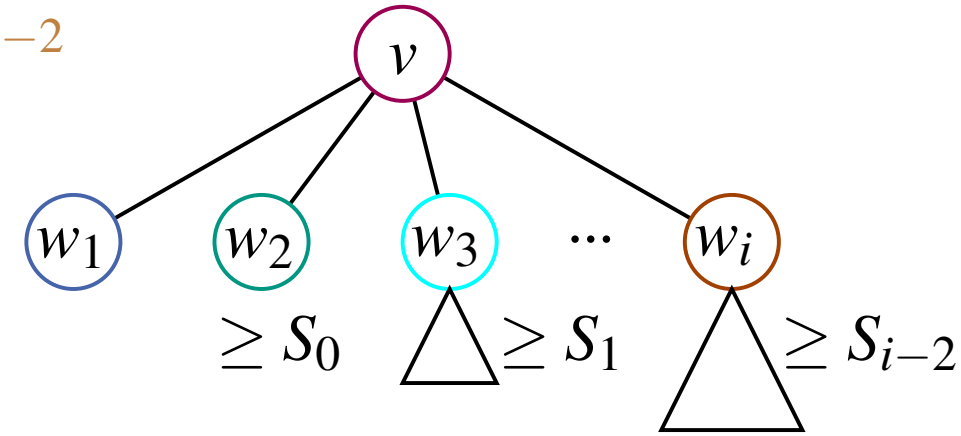
$S_1 = 2$

$S_i \geq 1 + 1 + S_0 + S_1 + \cdots + S_{i-2}$

für $i \geq 2$

Diese Rekurrenz

hat die Lösung $S_i \geq F_{i+2}$



Addressable Priority Queues: Mehr

- Untere Schranke $\Omega(\log n)$ für deleteMin, vergleichsbasiert.
  Beweis: Übung
- Worst case Schranken: nicht hier
- Monotone PQs mit ganzzahligen Schlüsseln (stay tuned)

Offene Probleme:

Analyse Pairing Heap, Vereinfachung Fibonacci Heap.

Zusammenfassung Datenstrukturen

- In dieser Vorlesung Fokus auf Beispiel Prioritätslisten
  (siehe auch kürzeste Wege, externe Algorithmen)
- Heapkonzept trägt weit
- Geschwisterzeiger erlauben Repräsentation beliebiger Bäume mit konstanter Zahl Zeiger pro Item.
- Fibonacci heaps als nichttriviales Beispiel für amortisierte Analyse

Fortgeschrittene Graphenalgorithmen

12 Kürzeste Wege

Folien teilweise von Rob van Stee

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

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

Anfangsknoten s.

Ausgabe: für alle $v \in V$

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

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

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

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

Allgemeine Definitionen

Wie bei BFS benutzen wir zwei Knotenarrays:

- $d[v] =$ aktuelle (vorläufige) Distanz von s nach v

  Invariante: $d[v] \geq \mu(v)$

- parent$[v] =$ Vorgänger von v

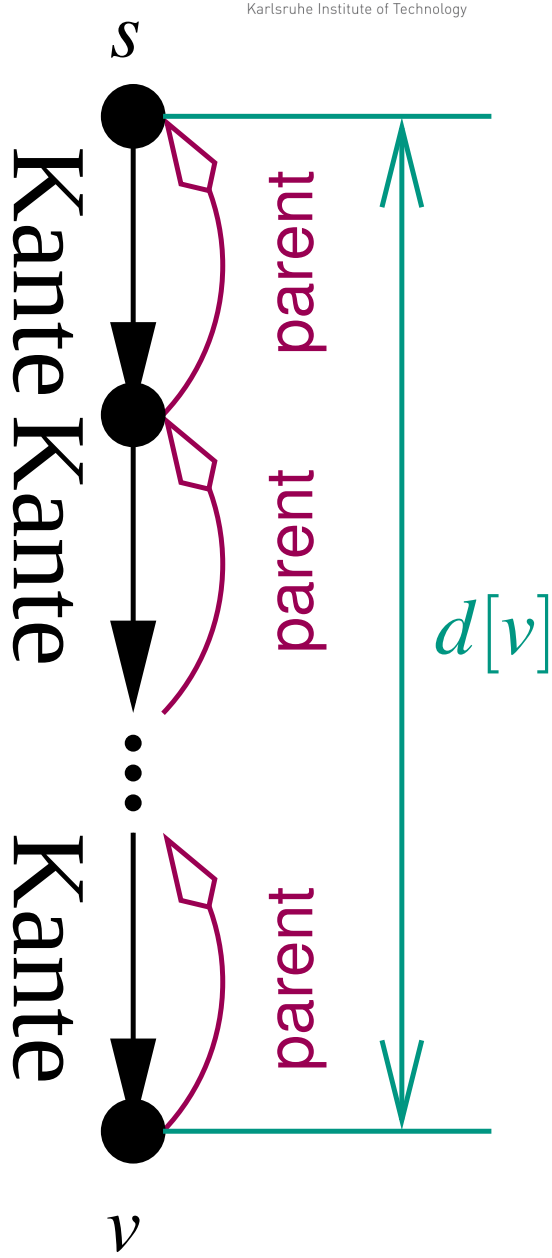
  auf dem (vorläufigen) kürzesten Pfad von s nach v

  Invariante: dieser Pfad bezeugt $d[v]$

Initialisierung:

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

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



Kante (u,v) relaxieren

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

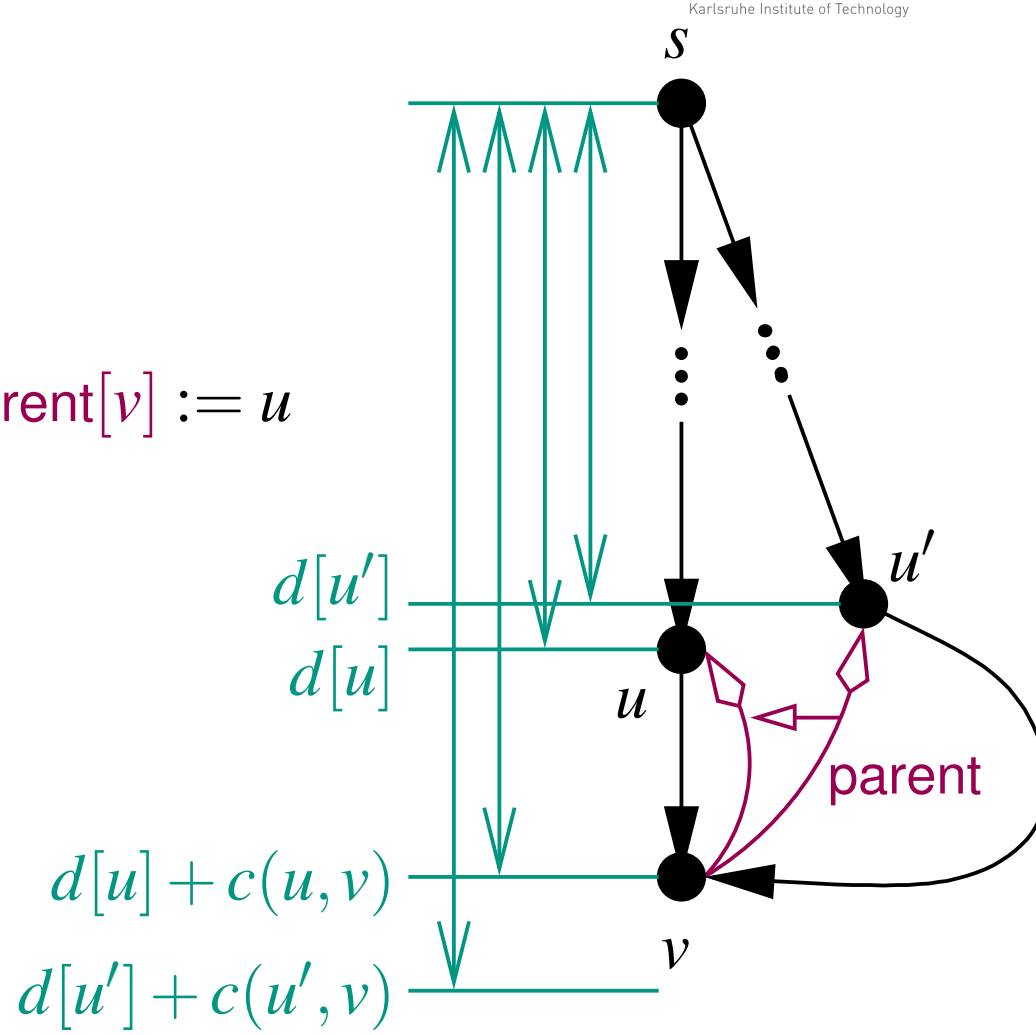
vielleicht $d[v]=\infty$

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

Invarianten bleiben erhalten!

Beobachtung:

$d[v]$ Kann sich mehrmals ändern!



Dijkstra's Algorithmus: Pseudocode

initialize d, parent

all nodes are non-scanned

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

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

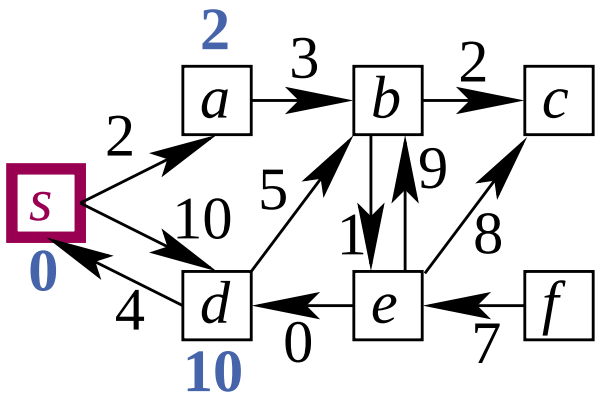
relax all edges (u,v) out of u

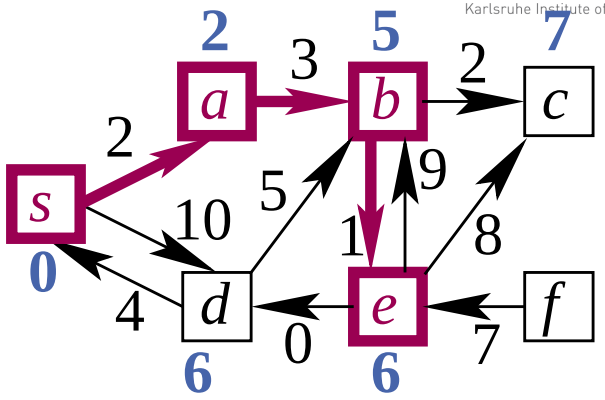
u is scanned now

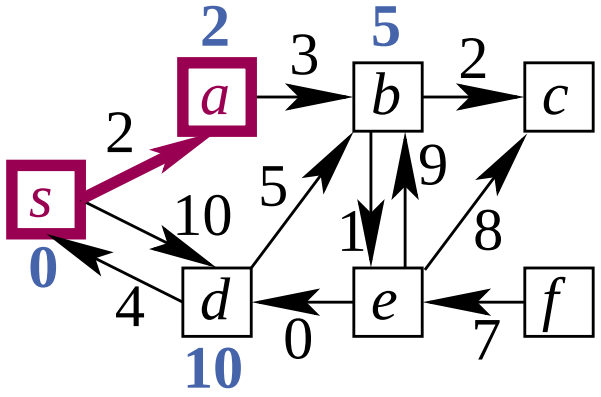
Behauptung: Am Ende definiert d die optimalen Entfernungen und parent die zugehörigen Wege (siehe Algo I:)

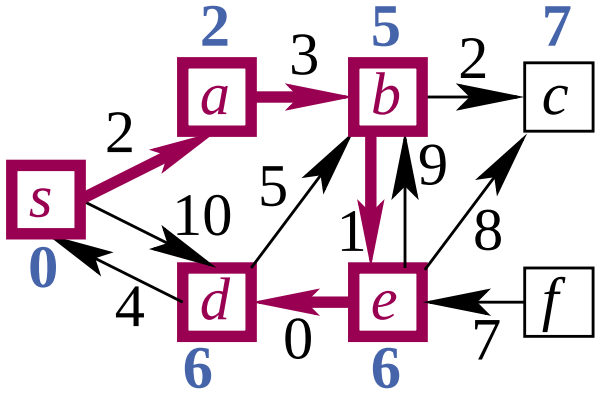
- v erreichbar $\Longrightarrow$ v wird irgendwann gescannt
- v gescannt $\Longrightarrow \mu(v) = d[v]$

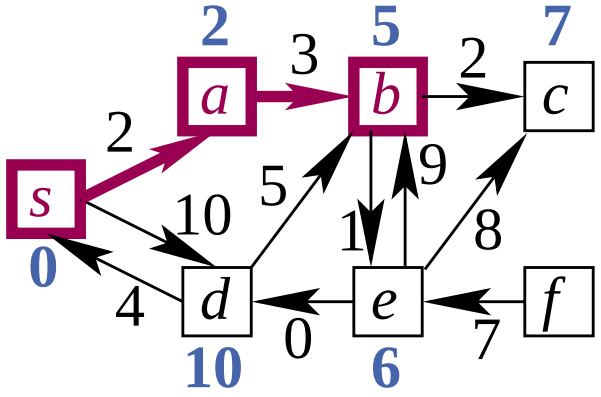
Beispiel

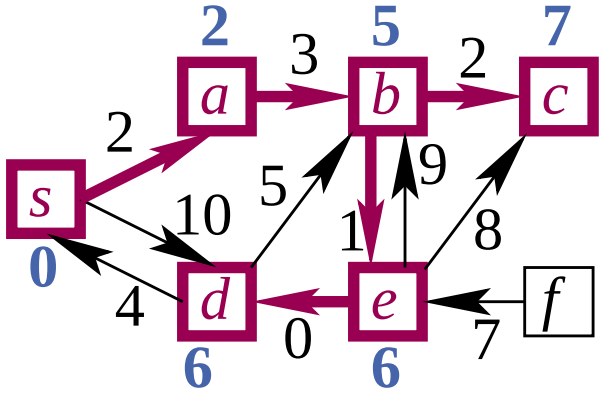

2
a
3
b
2
c
2
5
9
10
1
8
s
d
e
f
0
4
0
7
10


2
5
7
a
3
b
2
c
2
5
9
10
1
8
s
d
e
f
0
4
0
7
6
6


2
5
a
3
b
2
c
2
5
9
10
1
8
s
d
e
f
0
4
0
7
10


2
5
7
a
3
b
2
c
2
5
9
10
1
8
s
d
e
f
0
4
0
7
6
6


2
5
7
a
3
b
2
c
2
5
9
10
1
8
s
d
e
f
0
4
0
7
10
6


2
5
7
a
3
b
2
c
2
5
9
10
1
8
s
d
e
f
0
4
0
7
6
6

Laufzeit

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

Mit Fibonacci-Heapprioritätslisten:

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

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

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

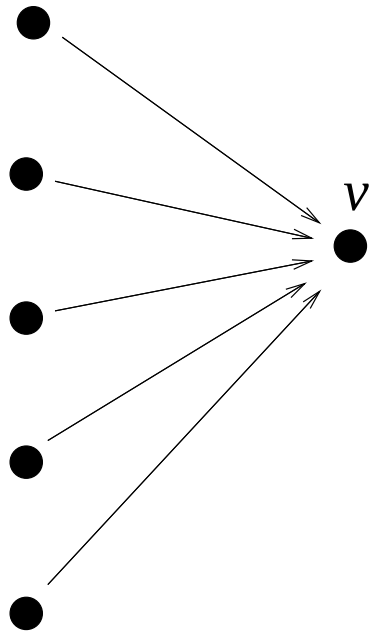
Laufzeit im Durchschnitt

Bis jetzt: $\leq m$ decreaseKeys ($\leq 1\times$ pro Kante)

Wieviel decreaseKeys im Durchschnitt?

Modell:

- Beliebiger Graph G
- Beliebiger Anfangsknoten s
- Beliebige Mengen $C(v)$
  von Kantengewichten für
  eingehende Kanten von Knoten v



Gemittelt wird über alle Zuteilungen

$C(v) \rightarrow$ eingehende Kanten von v

Beispiel: alle Kosten unabhängig identisch verteilt

Laufzeit im Durchschnitt

Probabilistische Sichtweise: Zufällige gleichverteilte Auswahl einer der zu mittelnden Eingaben.

$\Longrightarrow$

wir suchen den Erwartungswert der Laufzeit

Frage: Unterschied zu erwarteter Laufzeit bei randomisierten Algorithmen ?

Laufzeit im Durchschnitt

Satz 1. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Dann

$$\begin{aligned}
\mathbb{E}(T_{\text{DijkstraBHeap}}) &= \mathrm{O}\left(m + n\log\frac{m}{n} \cdot T_{\text{decreaseKey}}(n)\right. \\
&\qquad \left. + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\right) \\
&= \mathrm{O}\left(m + n\log\frac{m}{n}\log n + n\log n\right) \\
&= \mathrm{O}\left(m + n\log\frac{m}{n}\log n\right)
\end{aligned}$$

(wir hatten vorher $T_{\text{DijkstraBHeap}} = \mathrm{O}((m+n)\log n)$)

($T_{\text{DijkstraFib}} = \mathrm{O}(m + n\log n)$ schlechtester Fall)

Lineare Laufzeit für dichte Graphen

$m = \Omega(n \log n \log\log n) \Rightarrow$ lineare Laufzeit.

(nachrechnen)

Also hier u. U. besser als Fibonacci heaps

Satz 1. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Satz 1. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

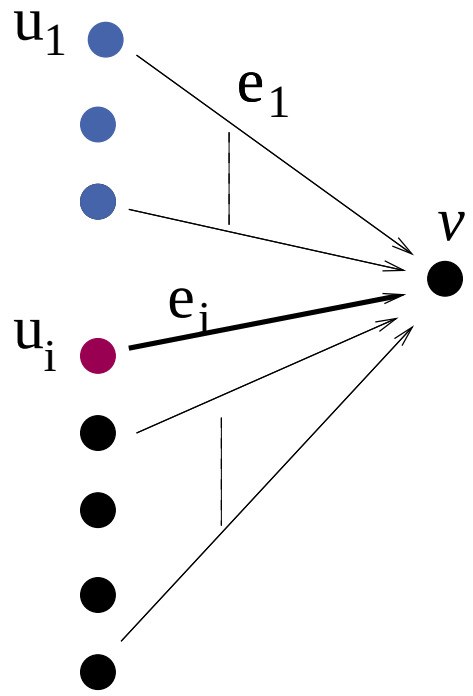
decreaseKey bei Bearbeitung von e_i nur wenn

$$\mu(u_i) + c(e_i) < \min_{j<i}(\mu(u_j) + c(e_j)).$$

Aber $\mu(u_i) \geq \mu(u_j)$ für $j < i$, also muss gelten:

$$c(e_i) < \min_{j<i} c(e_j).$$

Präfixminimum



Satz 1. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Kosten in $C(v)$ erscheinen in zufälliger Reihenfolge

Wie oft findet man ein neues Minimum bei zufälliger Reihenfolge?

Harmonische Zahl H_k (Sect. 2.8, s.u.)

Erstes Minimum: führt zu insert(v).

Also $\leq H_k - 1 \leq (\ln k + 1) - 1 = \ln k$ erwartete decreaseKeys

Satz 1. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Für Knoten $v \leq H_k - 1 \leq \ln k$ decreaseKeys (erwartet) mit $k = \text{indegree}(v)$.

Insgesamt

$$\sum_{v \in V} \ln \text{indegree}(v) \leq n \ln \frac{m}{n}$$

(wegen Konkavität von $\ln x$)

Präfixminima einer Zufallsfolge

Definiere Zufallsvariable M_n als Anzahl Präfixminima einer Folge von n verschiedenen Zahlen (in Abhängigkeit von einer Zufallspermutation)

Definiere Indikatorzufallsvariable $I_i := 1$ gdw. die i-te Zahl ein Präfixminimum ist.

$$\mathrm{E}[M_n] = \mathrm{E}[\sum_{i=1}^{n} I_i] \overset{\text{Lin. E}[\cdot]}{=} \sum_{i=1}^{n} \mathrm{E}[I_i]$$

$$= \sum_{i=1}^{n} \frac{1}{i} = H_n \quad \text{wegen } \mathbb{P}[I_i = 1] = \frac{1}{i}$$

$$\underbrace{x_1, \ldots, x_{i-1}}_{<x_i?}, x_i, x_{i+1}, \ldots, x_n$$

Monotone ganzzahlige Prioritätslisten

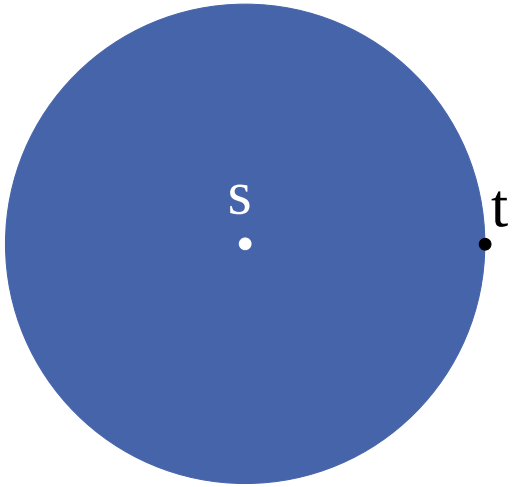
Grundidee: Datenstruktur auf Anwendung zuschneiden

Dijkstra's Algorithmus benutzt die Prioritätsliste monoton:

Operationen insert und decreaseKey benutzen Distanzen der Form

$d[u] + c(e)$

Dieser Wert nimmt ständig zu



Monotone ganzzahlige Prioritätslisten

Annahme: Alle Kantengewichte sind ganzzahlig und im Interval $[0, C]$

$\Longrightarrow \forall v \in V : d[v] \leq (n-1)C$

Es gilt sogar:

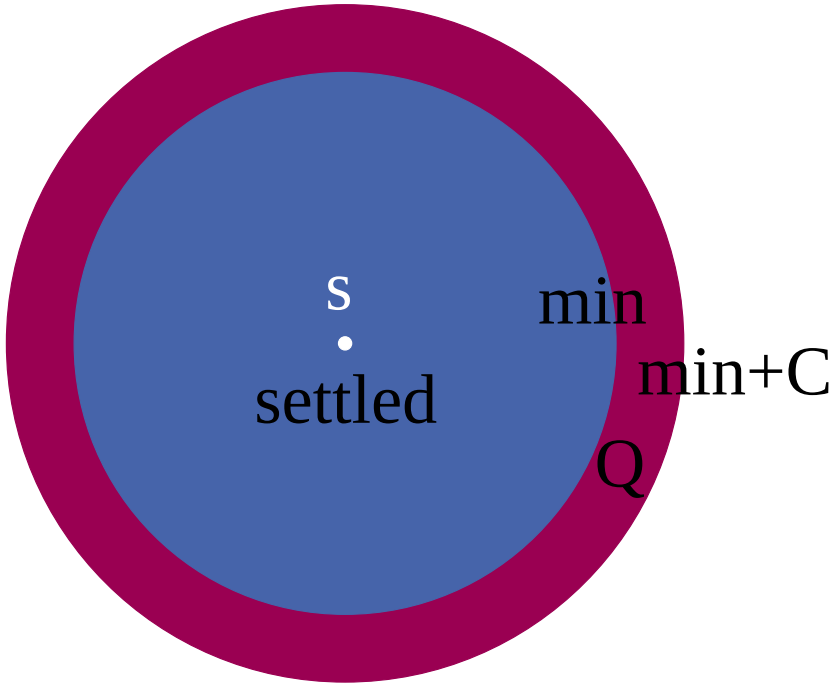
Sei min der letzte Wert,

der aus Q entfernt wurde.

In Q sind immer

nur Knoten mit Distanzen im

Interval $[min, min + C]$.

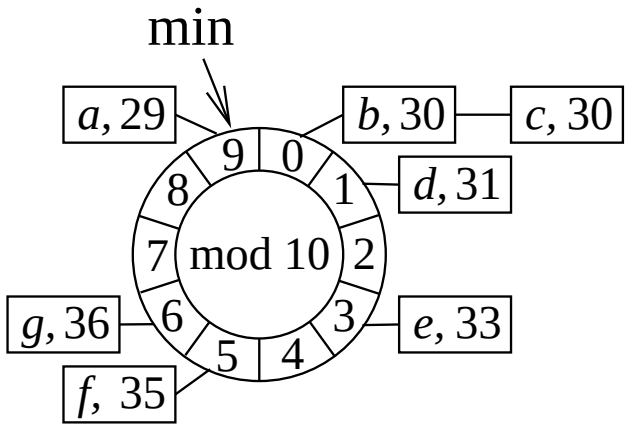


Bucket-Queue

Zyklisches Array B von $C+1$ doppelt verketteten Listen

Knoten mit Distanz $d[v]$ wird in $B[d[v] \mod (C+1)]$ gespeichert.

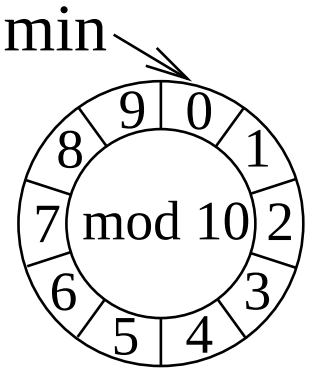
Bucket queue with C = 9



Content=

<(a,29), (b,30), (c,30), (d,31)
(e,33), (f,35), (g,36)>

Operationen



Initialisierung: $C+1$ leere Listen, $min = 0$

insert(v): fügt v in $B[d[v] \mod (C+1)]$ ein $O(1)$

decreaseKey(v): entfernt v aus seiner Liste und fügt es ein in $B[d[v] \mod (C+1)]$ $O(1)$

deleteMin: fängt an bei Bucket $B[min \mod (C+1)]$. Falls der leer ist, $min := min+1$, wiederhole. erfordert Monotonizität!

min nimmt höchstens nC mal zu, höchstens n Elemente werden insgesamt aus Q entfernt $\Rightarrow$

Gesamtkosten deleteMin-Operationen $= O(n+nC) =$ $O(nC)$.

Genauer: $O(n+\text{maxPathLength})$

Laufzeit Dijkstra mit Bucket-Queues

$$
\begin{aligned}
T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) \\
&\quad + \text{Kosten deleteMin-Operationen} \\
&\quad + n \cdot T_{\text{insert}}(n))) \\
T_{\text{DijkstraBQ}} &= O(m \cdot 1 + nC + n \cdot 1)) \\
&= O(m + nC) \text{ oder auch} \\
&= O(m + \text{maxPathLength})
\end{aligned}
$$

Mit Radix-Heaps finden wir sogar $T_{\text{DijkstraRadix}} = O(m + n \cdot \log C)$

Idee: nicht alle Buckets gleich groß machen

Radix-Heaps

Wir verwenden die Buckets -1 bis K, für $K = 1 + \lfloor \log C \rfloor$

$min =$ die zuletzt aus Q entfernte Distanz

Für jeden Knoten $v \in Q$ gilt $d[v] \in [min, \ldots, min + C]$.

Betrachte binäre Repräsentation der möglichen Distanzen in Q.

Nehme zum Beispiel $C = 9$, binär 1001. Dann $K = 4$.

Beispiel 1: $min = 10000$, dann $\forall v \in Q : d[v] \in [10000, 11001]$

Beispiel 2: $min = 11101$, dann $\forall v \in Q : d[v] \in [11101, 100110]$

Speichere v in Bucket $B[i]$ falls $d[v]$ und min sich zuerst an der $i$ten Stelle unterscheiden, (in $B[K]$ falls $i > K$, in $B[-1]$ falls sie sich nicht unterscheiden)

Definition $msd(a,b)$

Die Position der höchstwertigen Binärziffer wo a und b sich unterscheiden

a	$1100\mathbf{1}010$	$1010\mathbf{1}00$	1110110
b	$1100\mathbf{0}101$	$1010\mathbf{1}10$	1110110
$\text{msd}(a,b)$	3	1	-1

$msd(a,b)$ können wir mit Maschinenbefehlen sehr schnell berechnen

Radix-Heap-Invariante

v ist gespeichert in Bucket $B[i]$ wo $i = \min(msd(min, d[v]), K)$.

Beispiel 1: $min = 10000$, $C = 9$, $K = 4$

Bucket	$d[v]$ binär	$d[v]$
-1	10000	16
0	10001	17
1	1001*	18,19
2	101**	20–23
3	11***	24–25
4	-	-

(In Bucket 4 wird nichts gespeichert)

Radix-Heap-Invariante

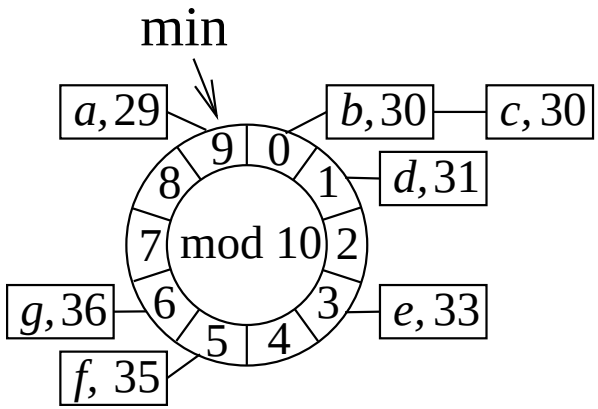
v ist gespeichert in Bucket $B[i]$ wo $i = \min(msd(min, d[v]), K)$.

Beispiel 2: $min = 11101$, $C = 9$, dann $K = 4$

Bucket	$d[v]$ binär	$d[v]$
-1	11101	29
0	-	-
1	1111*	30,31
2	-	-
3	-	-
4	100000 und höher	32 und höher

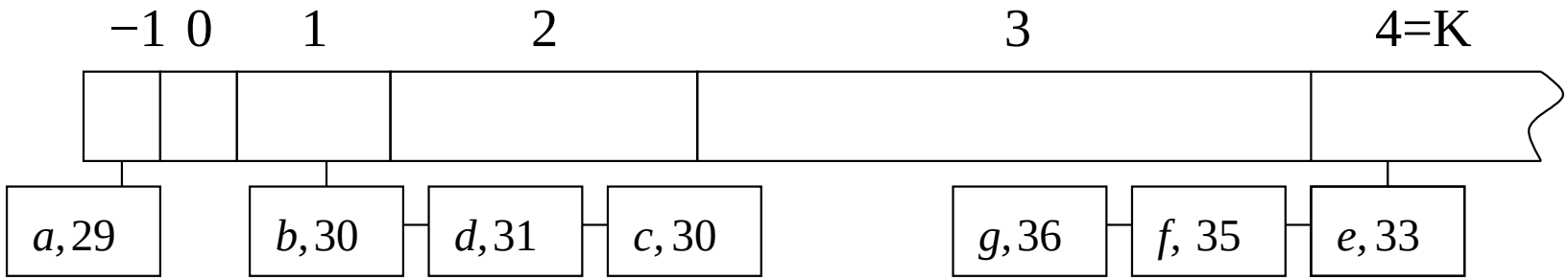
Falls $d[v] \geq 32$, dann $msd(min, d[v]) > 4$!

Bucket-Queues und Radix-Heaps



Bucket queue with C = 9

Content=
<(a,29), (b,30), (c,30), (d,31), (e,33), (f,35), (g,36)>



Binary Radix Heap

Radix Heap: deleteMin

Function deleteMin: Element

  if $B[-1] = \emptyset$

    $i := \min\{j \in 0..K : B[j] \neq \emptyset\}$

    move $\min B[i]$ to $B[-1]$ and to min

    foreach $e \in B[i]$ **do** // exactly here invariant is violated !

      move e to $B[\min(msd(min, d[v]), K)]$

  result:= $B[-1]$.popFront

  return result

$B[0], \ldots, B[i-1]$: leer, also nichts zu tun.

$B[i+1], \ldots, B[K]$: msd bleibt erhalten, weil altes und neues min **gleich** für alle Bits $j > i$

Buckets $j > i$ bei Änderung von min

Beispiel: $min = 10000$, $C = 9$, $K = 4$.

Neues $min = 10010$, war in Bucket 1

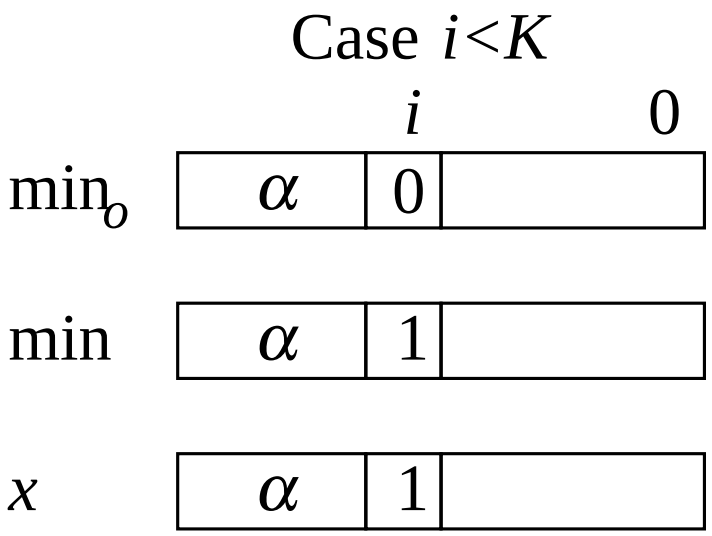
	$min = 10000$		$min = 10010$	
Bucket	$d[v]$ binär	$d[v]$	$d[v]$ binär	$d[v]$
-1	10000	16	10010	18
0	-	-	10011	19
1	1001*	18,19	-	-
2	101**	20–23	101**	20-23
3	11***	24–25	11***	24-27
4	-	-	-	-

Bucket $B[i]$ bei Änderung von *min*

Lemma: Elemente x aus $B[i]$ gehen zu Buckets mit kleineren Indices

Wir zeigen nur den Fall $i < K$.

Sei $\min_o$ der alte Wert von min.



Kosten der deleteMin-Operationen

Bucket $B[i]$ finden: $\mathrm{O}(i)$

Elemente aus $B[i]$ verschieben: $\mathrm{O}(|B[i]|)$

Insgesamt $\mathrm{O}(K + |B[i]|)$ falls $i \geq 0$, $\mathrm{O}(1)$ falls $i = -1$

Verschiebung erfolgt immer nach kleineren Indices

Wir zahlen dafür schon beim insert (amortisierte Analyse):

es gibt höchstens K Verschiebungen eines Elements

Laufzeit Dijkstra mit Radix-Heaps

Insgesamt finden wir amortisiert

- $T_{\text{insert}}(n) = \mathrm{O}(K)$
- $T_{\text{deleteMin}}(n) = \mathrm{O}(K)$
- $T_{\text{decreaseKey}}(n) = \mathrm{O}(1)$

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

Lineare Laufzeit für zufällige Kantengewichte

Vorher gesehen: Dijkstra mit Bucket-Queues hat lineare Laufzeit für dichte Graphen ($m > n \log n \log\log n$)

Letzte Folie: $T_{\text{DijkstraRadix}} = \mathrm{O}(m + n \cdot \log C)$

Jetzt: Dijkstra mit Radix-Heaps hat lineare Laufzeit ($\mathrm{O}(m+n)$) falls Kantenkosten identisch uniform verteilt in $0..C$

–wir brauchen nur eine kleine Änderung im Algorithmus

Änderung im Algorithmus für zufällige Kantengewichte

Vorberechnung von $c^{in}_{\min}(v) := \min\{c((u,v) : (u,v) \in E\}$ leichtestes eingehendes Kantengewicht.

Beobachtung: $d[v] \leq min + c^{in}_{\min}(v)$

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

$\Longrightarrow$ schiebe v in Menge F ungescannter Knoten mit korrekter Distanz

Knoten in F werden bei nächster Gelegenheit gescannt.

($\approx F$ als Erweiterung von $B[-1]$.)

Analyse

Ein Knoten v kommt nie in einem Bucket i mit $i < \log c^{in}_{\min}(v)$

Also wird v höchstens $K + 1 - \log c^{in}_{\min}(v)$ mal verschoben

Kosten von Verschiebungen sind dann insgesamt höchstens

$$\sum_v (K - \log c^{\text{in}}_{\min}(v) + 1) = n + \sum_v (K - \log c^{\text{in}}_{\min}(v)) \leq n + \sum_e (K - \log c(e)).$$

$K - \log c(e) =$ Anzahl Nullen am Anfang der binären Repräsentation von $c(e)$ als K-Bit-Zahl.

$$\mathbb{P}(K - \log c(e) = i) = 2^{-i} \quad \Rightarrow \quad \mathbb{E}(K - \log c(e)) = \sum_{i \geq 0} i 2^{-i} \leq 2$$

Laufzeit $= \mathrm{O}(m + n)$

All-Pairs Shortest Paths

Bis jetzt gab es immer einen bestimmten Anfangsknoten s

Wie können wir kürzeste Pfade für alle Paare (u, v) in G bestimmen?

Annahme: negative Kosten erlaubt, aber keine negativen Kreise

Lösung 1: n mal Bellman-Ford ausführen

... Laufzeit $O(n^2 m)$

Lösung 2: Knotenpotentiale

... Laufzeit $O(nm + n^2 \log n)$, deutlich schneller

Knotenpotentiale

Jeder Knoten bekommt ein Potential $\text{pot}(v)$

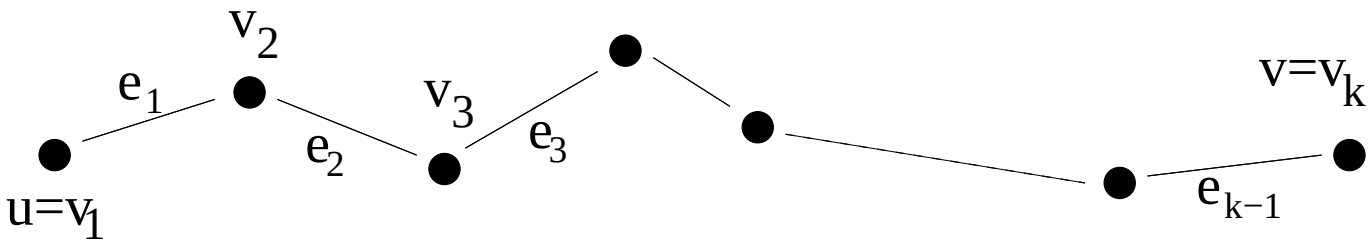
Mit Hilfe der Potentiale definieren wir reduzierte Kosten $\bar{c}(e)$ für Kante $e = (u, v)$ als

$$\bar{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v).$$

Mit diesen Kosten finden wir die gleichen kürzesten Pfade wie vorher!

Gilt für alle möglichen Potentiale – wir können sie also frei definieren

Knotenpotentiale



Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

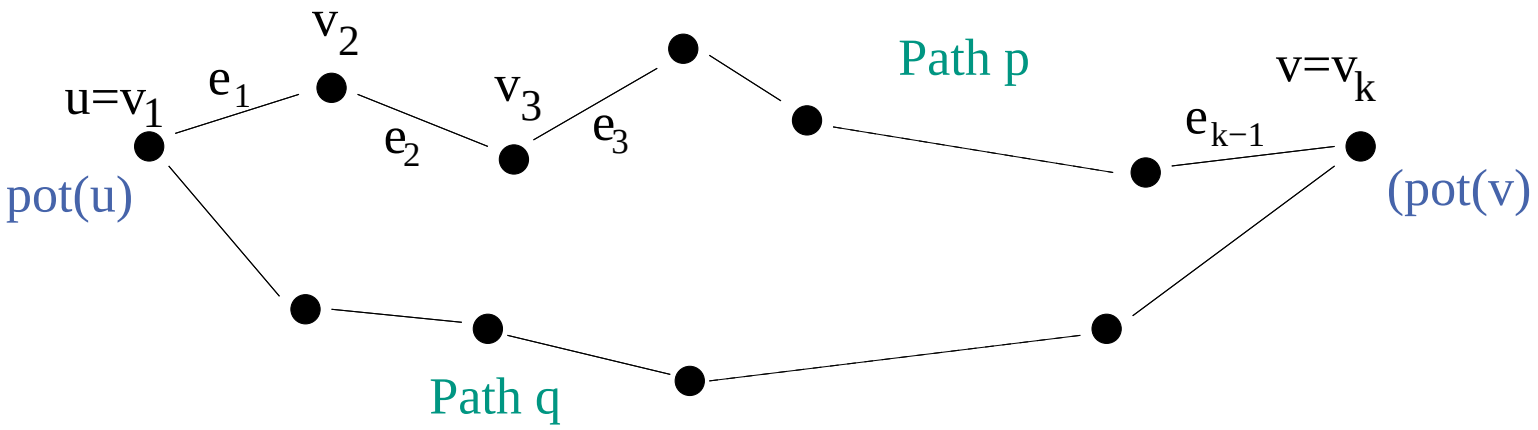
$$\begin{aligned}
\bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) = \sum_{i=1}^{k-1} (\mathrm{pot}(v_i) + c(e_i) - \mathrm{pot}(v_{i+1})) \\
&= \mathrm{pot}(v_1) + \sum_{i=1}^{k-1} c(e_i) - \mathrm{pot}(v_k) \\
&= \mathrm{pot}(v_1) + c(p) - \mathrm{pot}(v_k).
\end{aligned}$$

Knotenpotentiale

Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

$$\bar{c}(p) = \text{pot}(v_1) + c(p) - \text{pot}(v_k).$$

Sei q ein anderer u-v-Pfad, dann $c(p) \leq c(q) \Leftrightarrow \bar{c}(p) \leq \bar{c}(q)$.



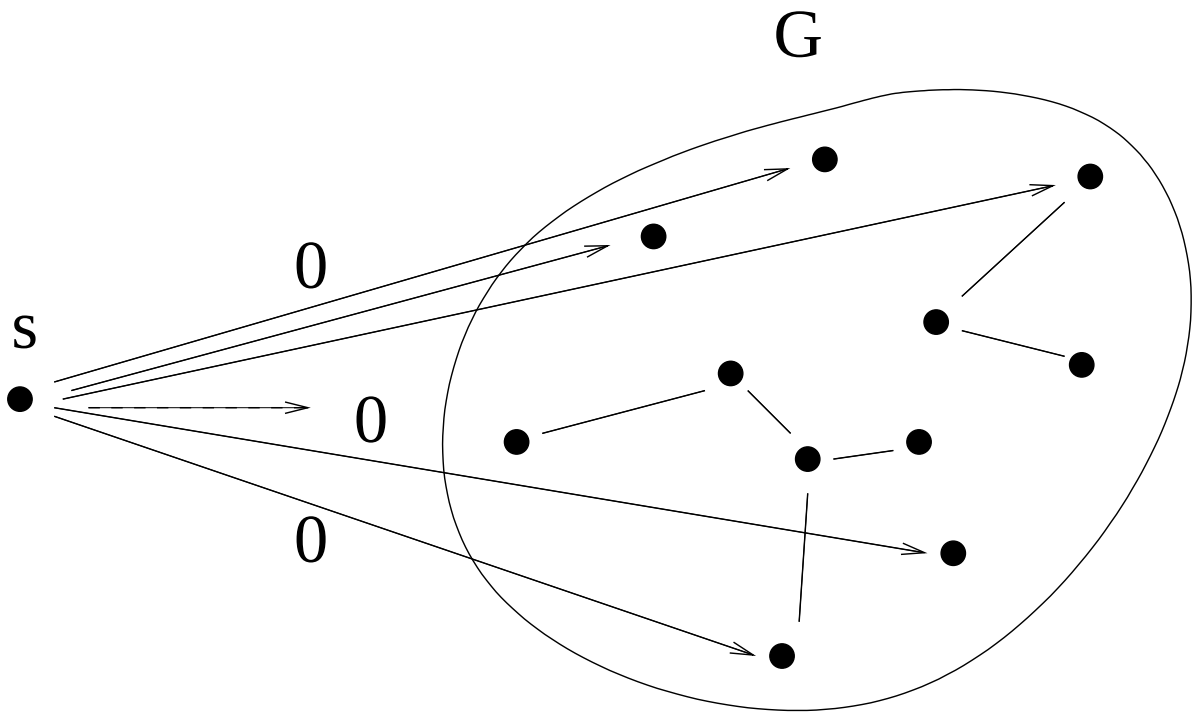
Definition: $\mu(u,v) =$ kürzeste Distanz von u nach v

Hilfsknoten

Wir fügen einen Hilfsknoten s an G hinzu

Für alle $v \in V$ fügen wir eine Kante (s, v) hinzu mit Kosten 0

Berechne kürzeste Pfade von s aus mit Bellman-Ford



Definition der Potentiale

Definiere $\text{pot}(v) := \mu(v)$ für alle $v \in V$

Jetzt sind die reduzierten Kosten alle nicht negativ: also können wir Dijkstra benutzen! (Evtl. s wieder entfernen...)

- Keine negativen Kreise, also $\text{pot}(v)$ wohldefiniert
- Für beliebige Kante (u,v) gilt

$$\mu(u) + c(e) \geq \mu(v)$$

deshalb

$$\bar{c}(e) = \mu(u) + c(e) - \mu(v) \geq 0$$

Algorithmus

All-Pairs Shortest Paths in the Absence of Negative Cycles

neuer Knoten s

foreach $v \in V$ **do** füge Kante (s,v) ein (Kosten 0) // $O(n)$

pot:= μ:= BellmanFordSSSP(s,c) // $O(nm)$

foreach Knoten $x \in V$ **do** // $O(n(m+n\log n))$

$\bar{\mu}(x,\cdot)$:= DijkstraSSSP$(x,\bar{c})$

// zurück zur ursprünglichen Kostenfunktion

foreach $e=(v,w) \in V \times V$ **do** // $O(n^2)$

$\mu(v,w) := \bar{\mu}(v,w) + \text{pot}(w) - \text{pot}(v)$

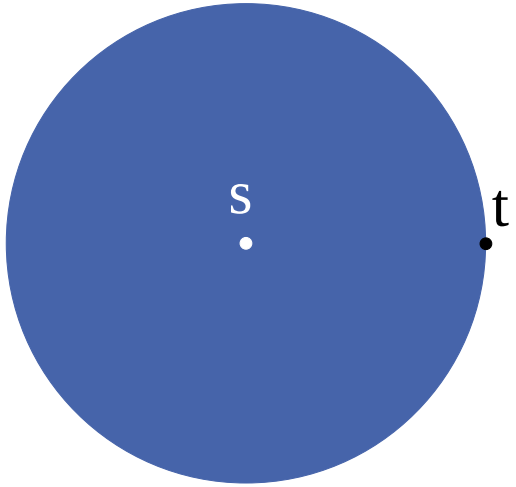
Laufzeit

- s hinzufügen: $\mathrm{O}(n)$
- Postprocessing: $\mathrm{O}\left(n^2\right)$ (zurück zu den ursprünglichen Kosten)
- n mal Dijkstra dominiert

Laufzeit $\mathrm{O}(n(m+n\log n)) = \mathrm{O}\left(nm+n^2\log n\right)$

Distanz zu einem Zielknoten t

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



Trick 0:

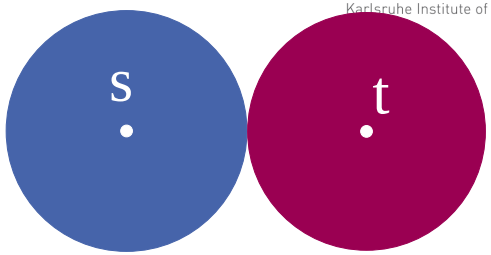
Dijkstra hört auf, wenn t aus Q entfernt wird

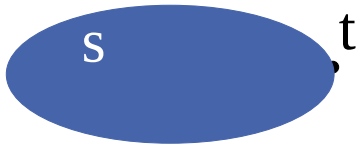
Spart “im Durchschnitt” Hälfte der Scans

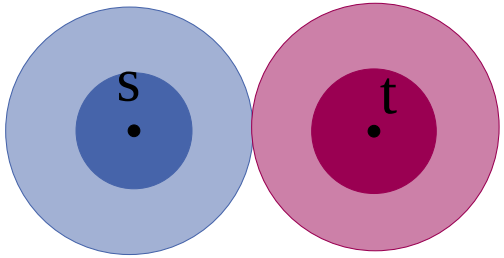
Frage: Wieviel spart es (meist) beim Europa-Navi?

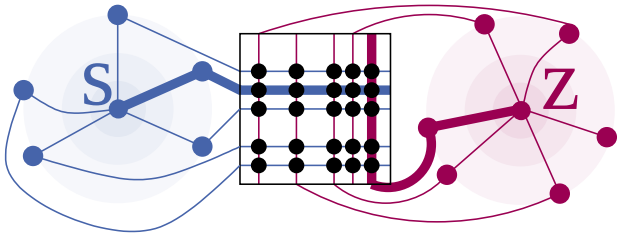
Ideen für Routenplanung

- Vorwärts + Rückwärtssuche
- **Zielgerichtete Suche**
- Hierarchien ausnutzen 
- Teilabschnitte tabellieren 

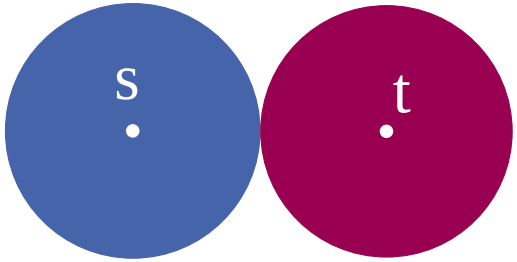








Bidirektionale Suche



Idee: Suche abwechselnd s und t

Vorwärtssuche auf normalem Graph $G = (V, E)$

Rückwärtssuche auf Rückwärtsgraph $G^r = (V, E^r)$

(Suchrichtungen wechseln in jedem Schritt)

Vorläufige kürzeste Distanz wird in jedem Schritt gespeichert:

$d[s,t] = min(d[s,t],\ d_{forward}[u] + d_{backward}[u])$

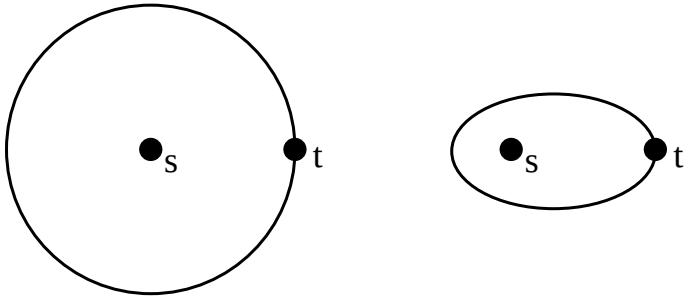
Abbruchkriterium:

Suche scannt Knoten, der in anderer Richtung bereits gescannt wurde.

$d[s,t] \Rightarrow\ \mu(s,t)$

A^*-Suche

Idee: suche "in die Richtung von t"

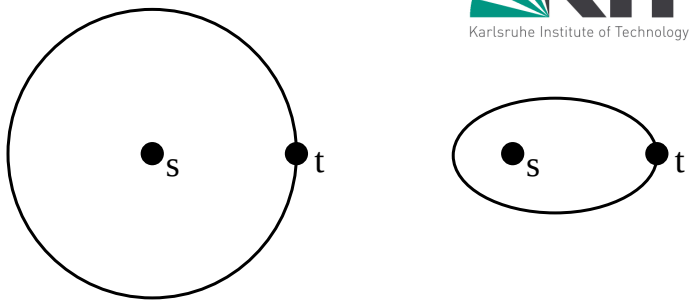


Annahme: Wir kennen eine Funktion $f(v)$ die $\mu(v,t)$ schätzt $\forall v$

Definiere pot$(v) = f(v)$ und $\bar{c}(u,v) = c(u,v) + f(v) - f(u)$

[Oder: in Dijkstra's Algorithmus, entferne nicht v mit minimalem $d[v]$ aus Q, sondern v mit minimalem $d[v] + f[v]$]

A^*-Suche



Idee: suche “in die Richtung von t”

Annahme: wir kennen eine Funktion $f(v)$ die $\mu(v,t)$ schätzt $\forall v$

Definiere pot$(v) = f(v)$ und $\bar{c}(u,v) = c(u,v) + f(v) - f(u)$

Beispiel: $f(v) = \mu(v,t)$.

Dann gilt: $\bar{c}(u,v) = c(u,v) + \mu(v,t) - \mu(u,t) = 0$ falls (u,v) auf

dem kürzesten Pfad von s nach t liegt.

$$s \to \cdots \to \underbrace{u \xrightarrow{c(u,v)} \overbrace{v \to \cdots \to t}^{\mu(v,t)}}_{\mu(u,t)}$$

Also scannt Dijkstra nur die Knoten auf diesem Pfad!

Benötigte Eigenschaften von $f(v)$

- Konsistenz (reduzierte Kosten nicht negativ):
  $c(e) + f(v) \geq f(u)\ \forall e = (u,v)$
- $f(v) \leq \mu(v,t)\ \forall v \in V$. Dann gilt $f(t) = 0$ und wir können aufhören wenn t aus Q entfernt wird.

Sei p irgendein Pfad von s nach t.

Alle Kanten auf p sind relaxiert? $\Rightarrow d[t] \leq c(p)$.

Sonst: $\exists v \in p \cap Q$, und $d[t] + f(t) \leq d[v] + f(v)$ weil t schon entfernt wurde. Deshalb

$$d[t] = d[t] + f(t) \leq d[v] + f(v) \leq d[v] + \mu(v,t) \leq c(p)$$

Wie finden wir $f(v)$?

Wir brauchen Heuristiken für $f(v)$.

Strecke im Straßennetzwerk: $f(v) =$ euklidischer Abstand $||v-t||_2$

bringt deutliche aber nicht überragende Beschleunigung

Fahrzeit: $\dfrac{||v-t||_2}{\text{Höchstgeschwindigkeit}}$ praktisch nutzlos

Noch besser aber mit Vorberechnung: Landmarks

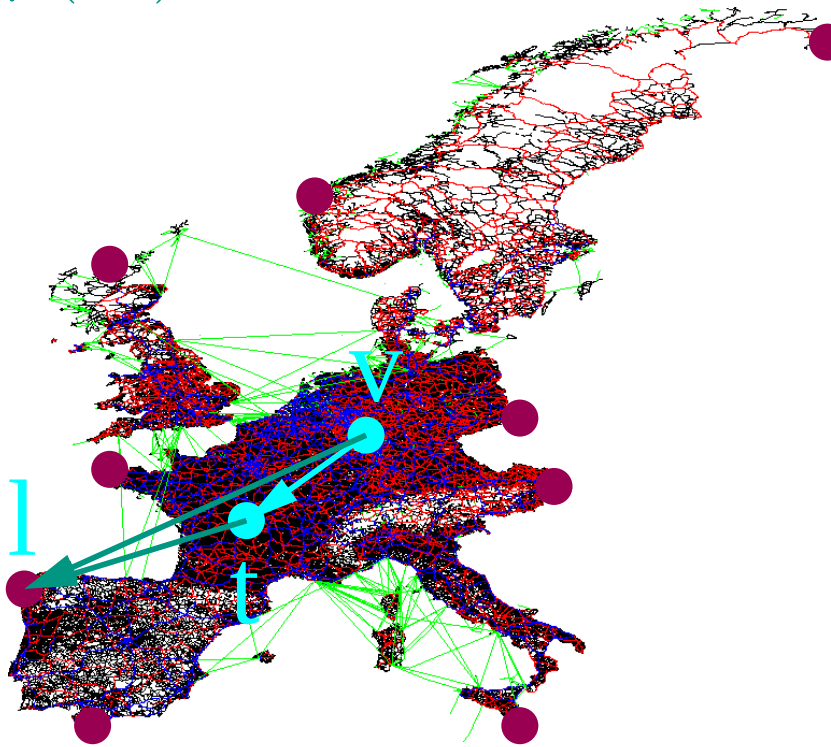
Landmarks [Goldberg Harrelson 2003]

Vorberechnung: Wähle Landmarkmenge L. $\forall \ell \in L, v \in V$ berechne/speichere $\mu(v,\ell)$.

Query: Suche Landmark $\ell \in L$ "hinter" dem Ziel.

Benutze untere Schranke $f_\ell(v) = \mu(v,\ell) - \mu(t,\ell)$

- $+$ Konzeptuell einfach
- $+$ Erhebliche Beschleunigungen
  ($\approx$ Faktor 20 im Mittel)
- $+$ Kombinierbar mit anderen Techniken
- $-$ Landmarkauswahl kompliziert
- $-$ hoher Platzverbrauch



Zusammenfassung Kürzeste Wege

- Nichtriviale Beispiele für Analyse im Mittel. Ähnlich für MST
- Monotone, ganzzahlige Prioritätslisten als Beispiel wie Datenstrukturen auf Algorithmus angepasst werden
- Knotenpotentiale allgemein nützlich in Graphenalgorithmen
- Aktuelle Forschung trifft klassische Algorithmik

Karlsruhe Institute of Technology

13 Anwendungen von DFS

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$(u, v$: 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 Nummerierung

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

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

Beobachtung:

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

Fertigstellungszeit

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

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

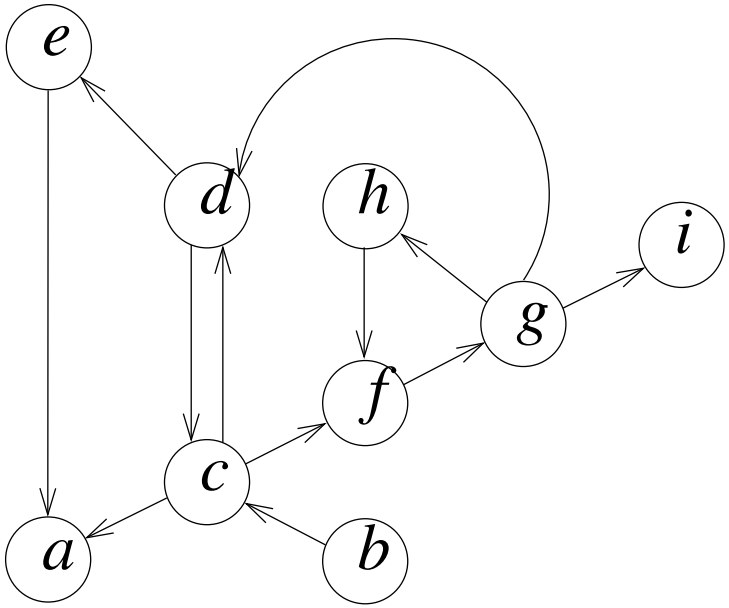
Starke Zusammenhangskomponenten

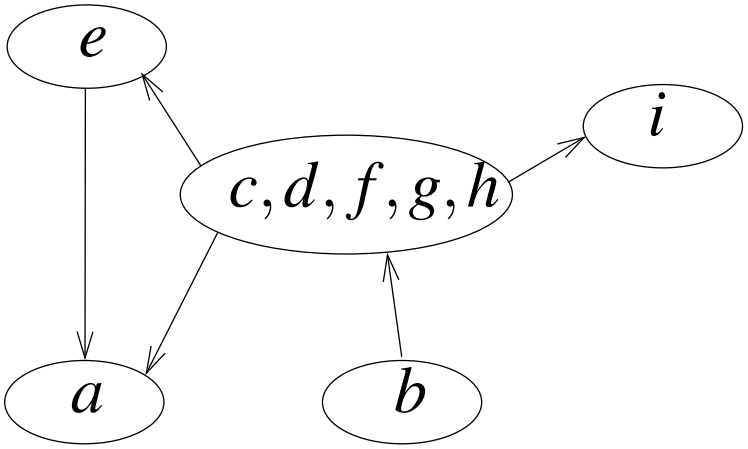
Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit

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

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

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





Starke Zusammenhangskomponenten – Abstrakter Algorithmus

$G_c := (V, \emptyset = E_c)$

foreach edge $e \in E$ **do**

  invariant SCCs of G_c are known

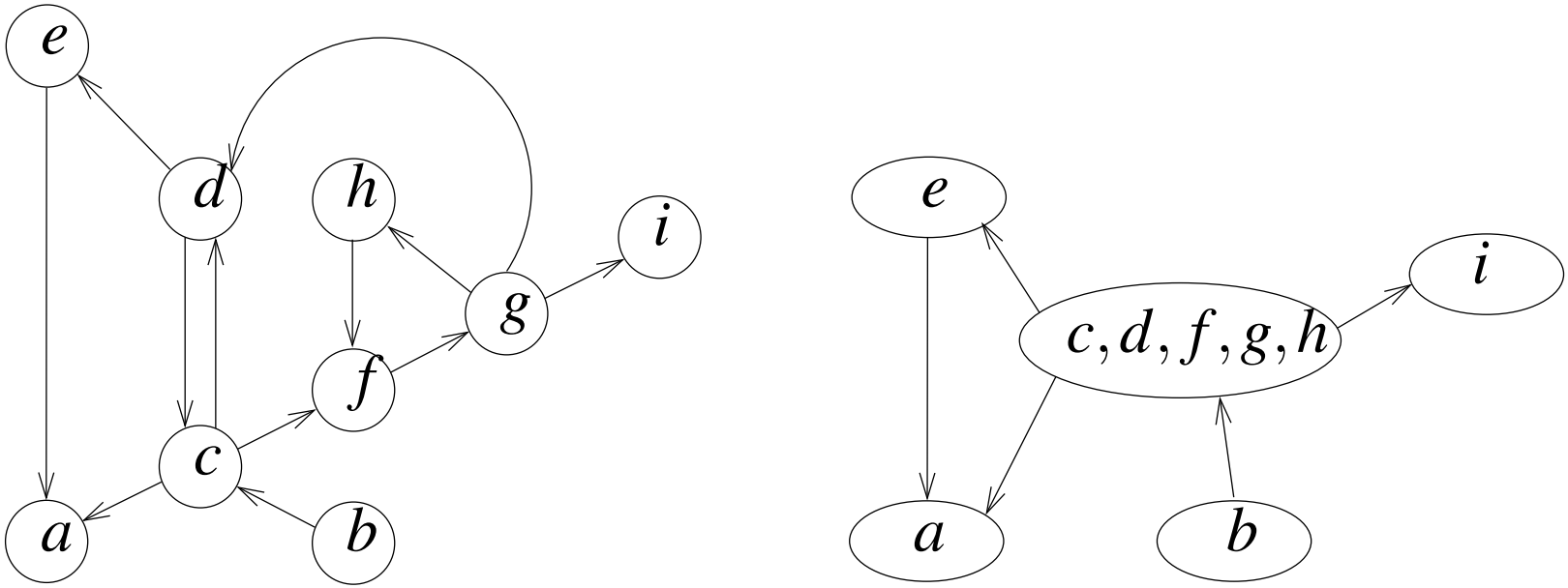
  $E_c := E_c \cup \{e\}$

Schrumpfgraph

$G_c^s = (V^s, E_c^s)$

Knoten: SCCs von G_c.

Kanten: $(C,D) \in E_c^s \Leftrightarrow \exists (c,d) \in E_c : c \in C \wedge d \in D$



Beobachtung: Der Schrumpfgraph ist azyklisch

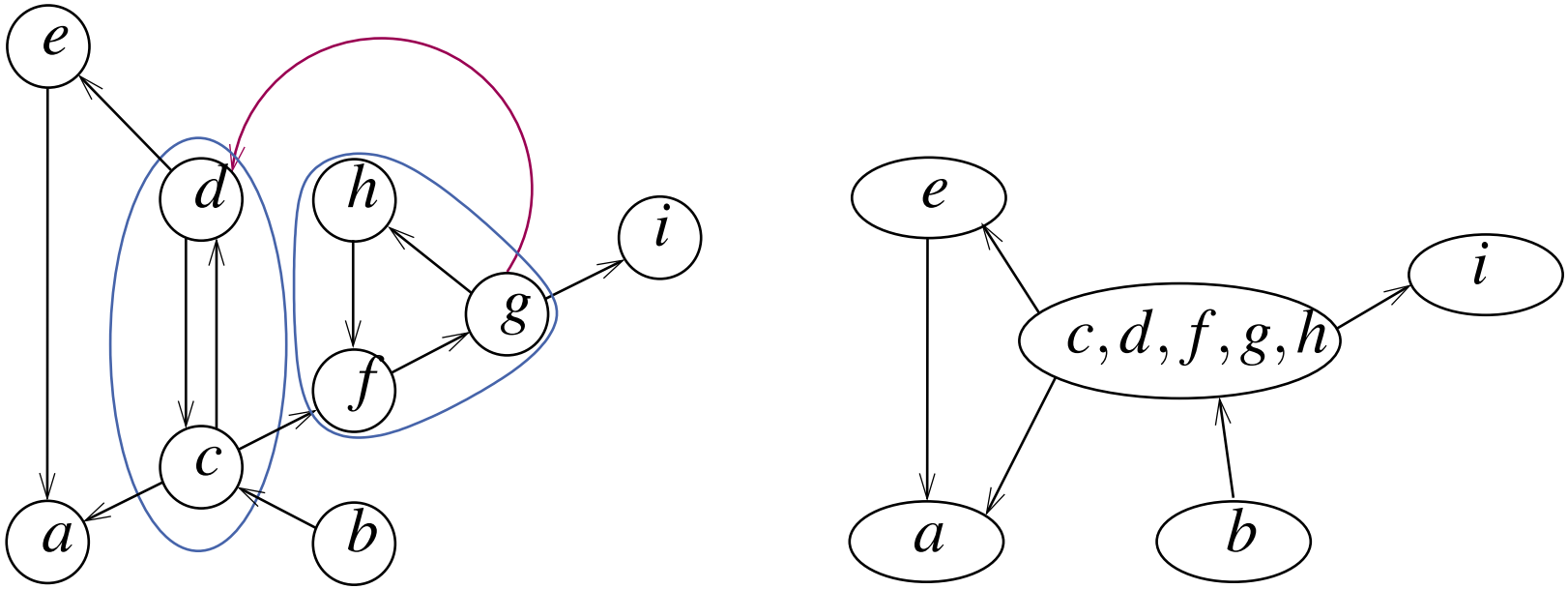
Auswirkungen einer neuen Kante e auf G_c, G_c^s

SCC-intern: Nichts ändert sich

zwischen zwei SCCs:

Kein Kreis: Neue Kante in G_c^s

Kreisschluss: SCCs auf Kreis kollabieren.



Konkreter: SCCs mittels DFS

[Cheriyan/Mehlhorn 96, Gabow 2000]

$V_c =$ markierte Knoten

$E_c =$ bisher explorierte Kanten

Aktive Knoten: markiert aber nicht finished.

SCCs von G_c:

nicht erreicht: Unmarkierte Knoten

offen: enthält aktive Knoten

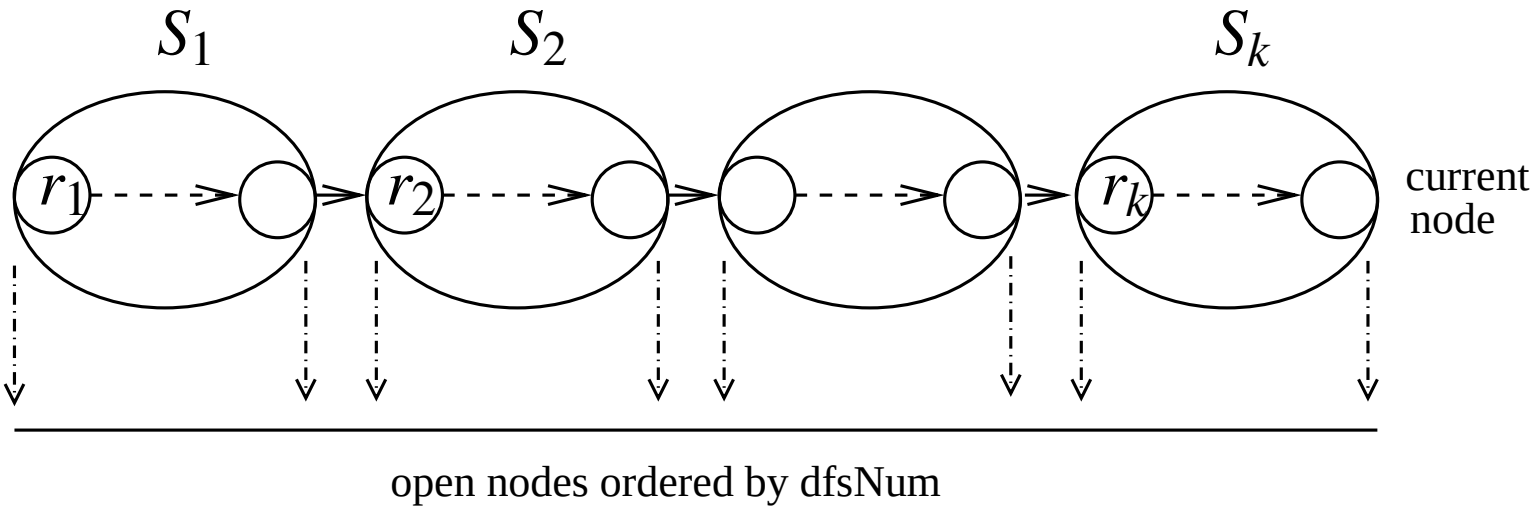
abgeschlossen: alle Knoten finished

component[w] gibt Repräsentanten einer SCC an.

Knoten von offenen (abgeschl.) Komponenten heißen offen (abgeschl.)

Invarianten von G_c

1. Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten
2. Offene Komponenten $S_1,\ldots,S_k$ bilden Pfad in G_c^s.
3. Repräsentanten partitionieren die offenen Komponenten bzgl. ihrer dfsNum.



Lemma: Abgeschlossene SCCs von G_c sind SCCs von G

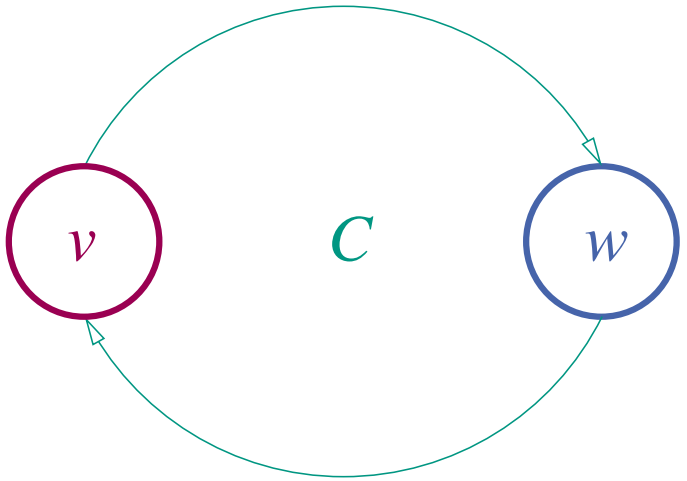
Betrachte abgeschlossenen Knoten v

und beliebigen Knoten w

in der SCC von v bzgl. G.

z.Z.: w ist abgeschlossen und

in der gleichen SCC von G_c wie v.



Betrachte Kreis C durch v, w.

Inv. 1: Knoten von C sind abgeschlossen.

Abgeschl. Knoten sind finished.

Kanten aus finished Knoten wurden exploriert.

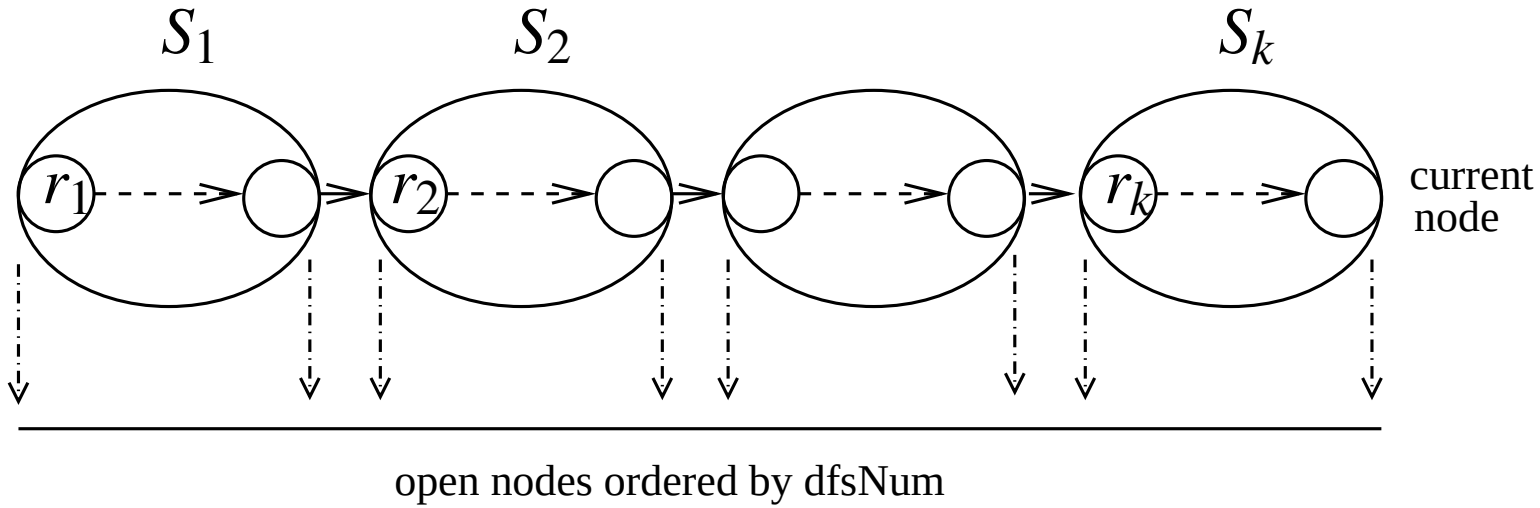
Also sind alle Kanten von C in G_c. □

Repräsentation offener Komponenten

Zwei Stapel aufsteigend sortiert nach dfsNum

oReps: Repräsentanten offener Komponenten

oNodes: Alle offenen Knoten



Karlsruhe Institute of Technology

init

component : NodeArray **of** NodeId // SCC representatives

oReps=$\langle\rangle$: Stack **of** NodeId // representatives of open SCCs

oNodes=$\langle\rangle$: Stack **of** NodeId // all nodes in open SCCs

Alle Invarianten erfüllt.

(Weder offene noch geschlossene Knoten)

root(s)

oReps.push(s) // new open

oNodes.push(s) // component

$\{s\}$ ist die einzige offene Komponente.

Alle Invarianten bleiben gültig

traverseTreeEdge(v, w)

oReps.push(w) // new open

oNodes.push(w) // component

$\{w\}$ ist neue offene Komponente.

dfsNum$(w) >$ alle anderen.

$\rightsquigarrow$Alle Invarianten bleiben gültig

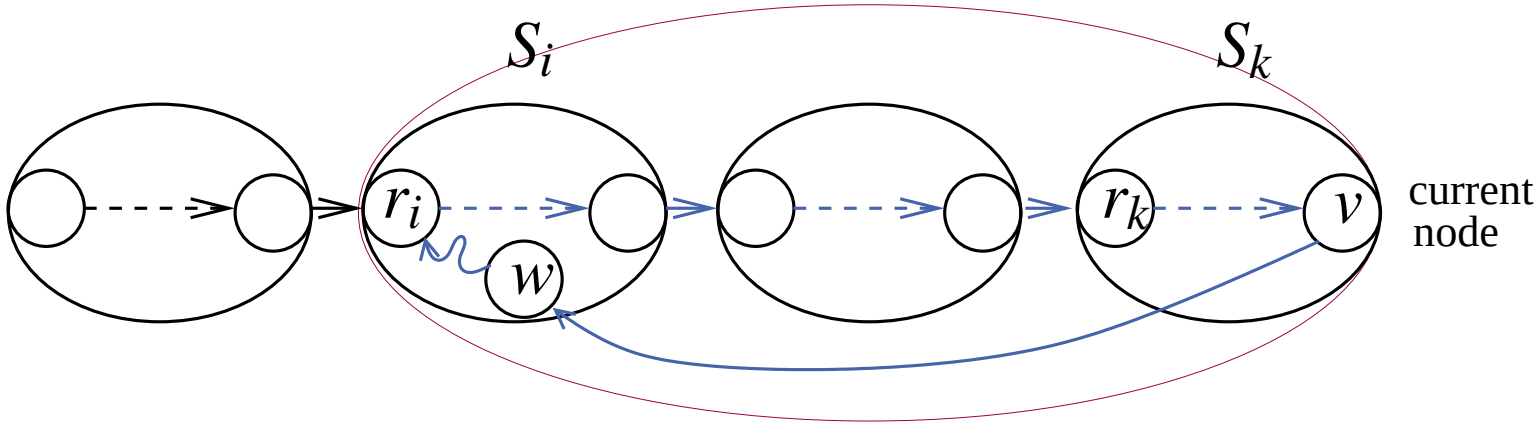
traverseNonTreeEdge(v, w)

if $w \in$ oNodes **then**

while $w \prec$ oReps.top **do** oReps.pop

$w \notin$ oNodes $\rightsquigarrow$ w is abgeschlossen $\overset{Lemma(*)}{\rightsquigarrow}$ Kante uninteressant

$w \in$ oNodes: kollabiere offene SCCs auf Kreis



backtrack(u, v)

if $v =$ oReps.top **then**

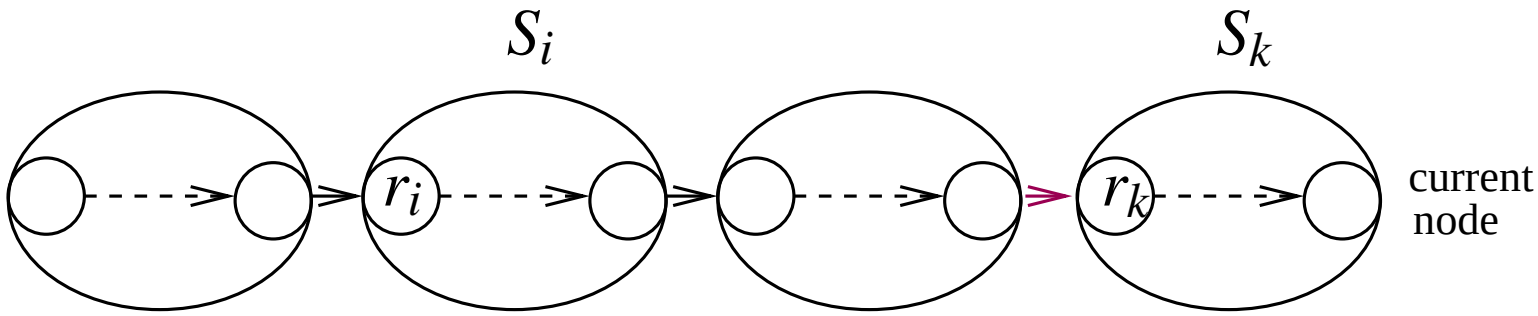
oReps.pop // close

repeat // component

w:= oNodes.pop

component$[w]$:= v

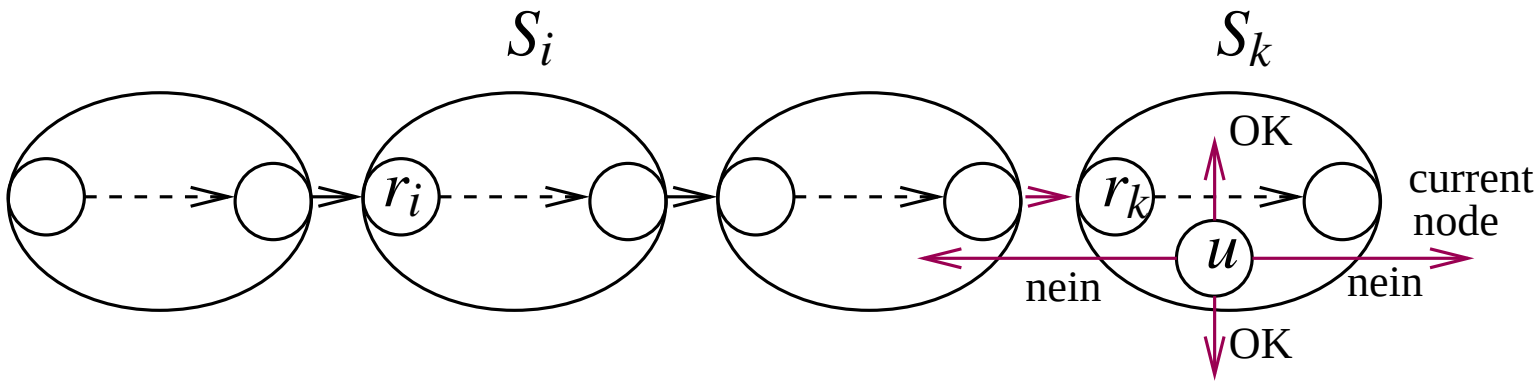
until $w = v$



z.Z. Invarianten bleiben erhalten...

```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

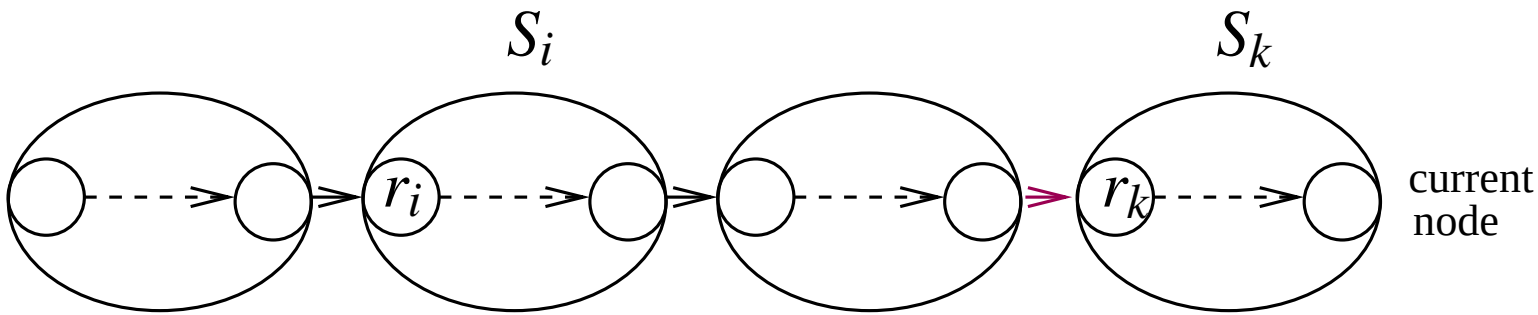
Inv. 1: Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten.



```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

Inv. 2: Offene Komponenten $S_1,\ldots,S_k$ bilden Pfad in G_c^s

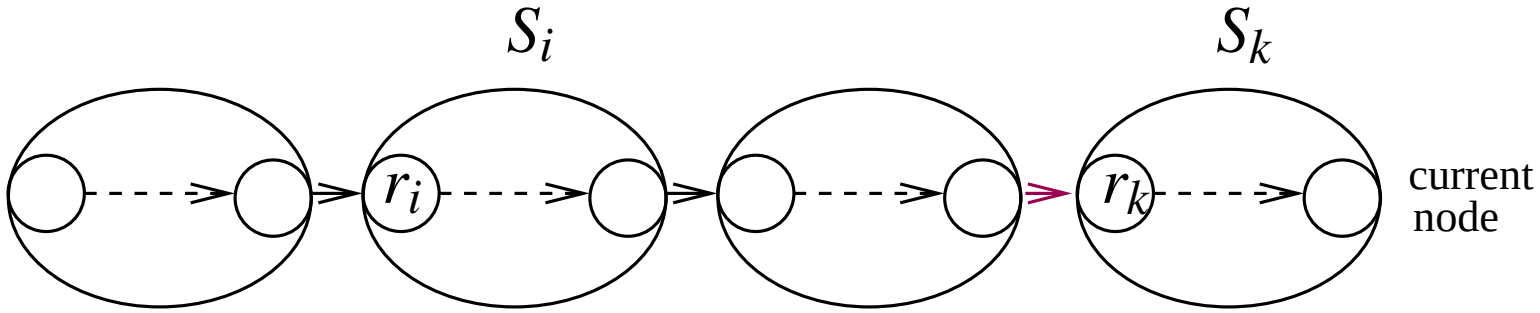
OK. (S_k wird ggf. entfernt)



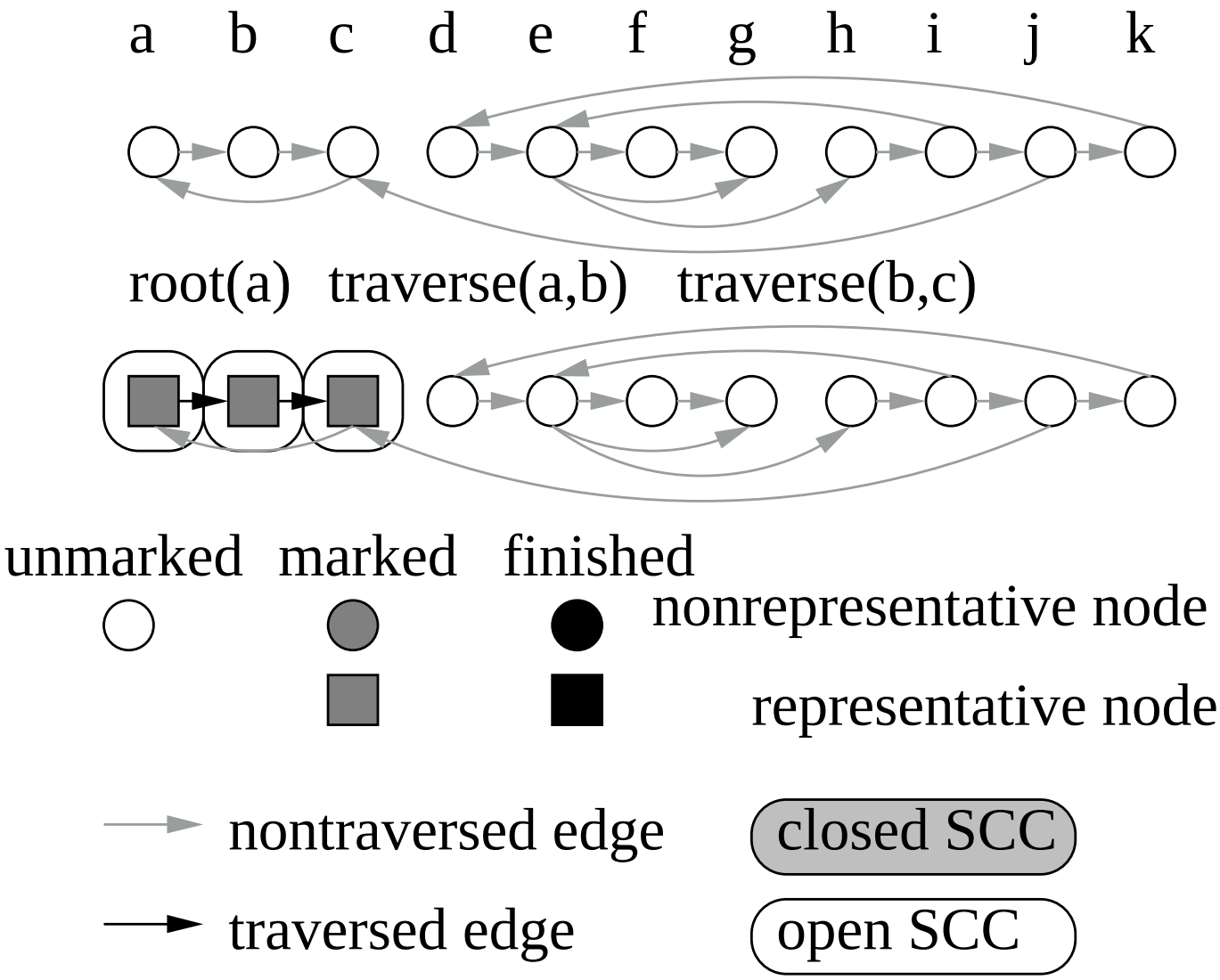
```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

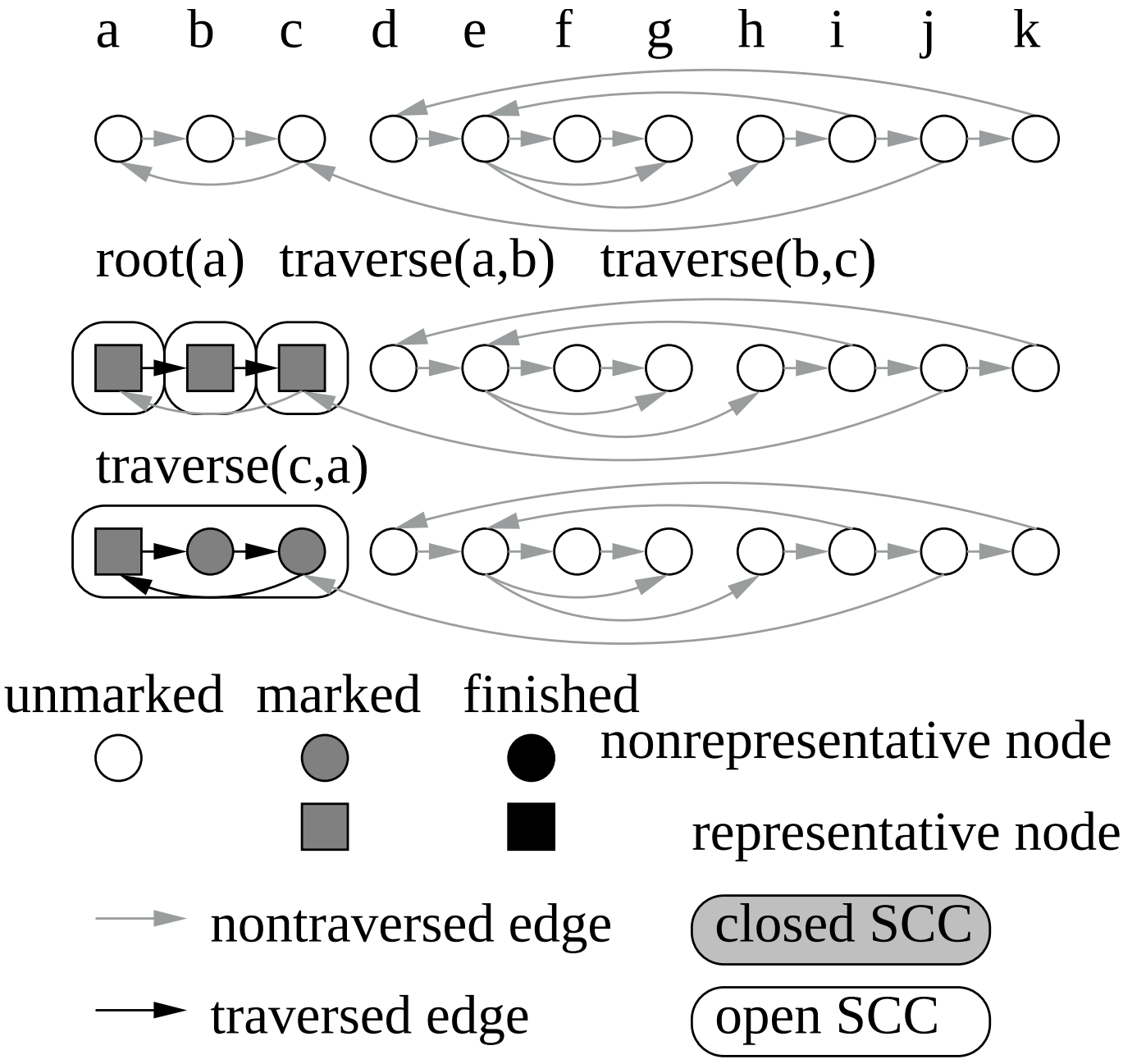
Inv. 3: Repräsentanten partitionieren die offenen Komponenten bzgl. ihrer dfsNum.

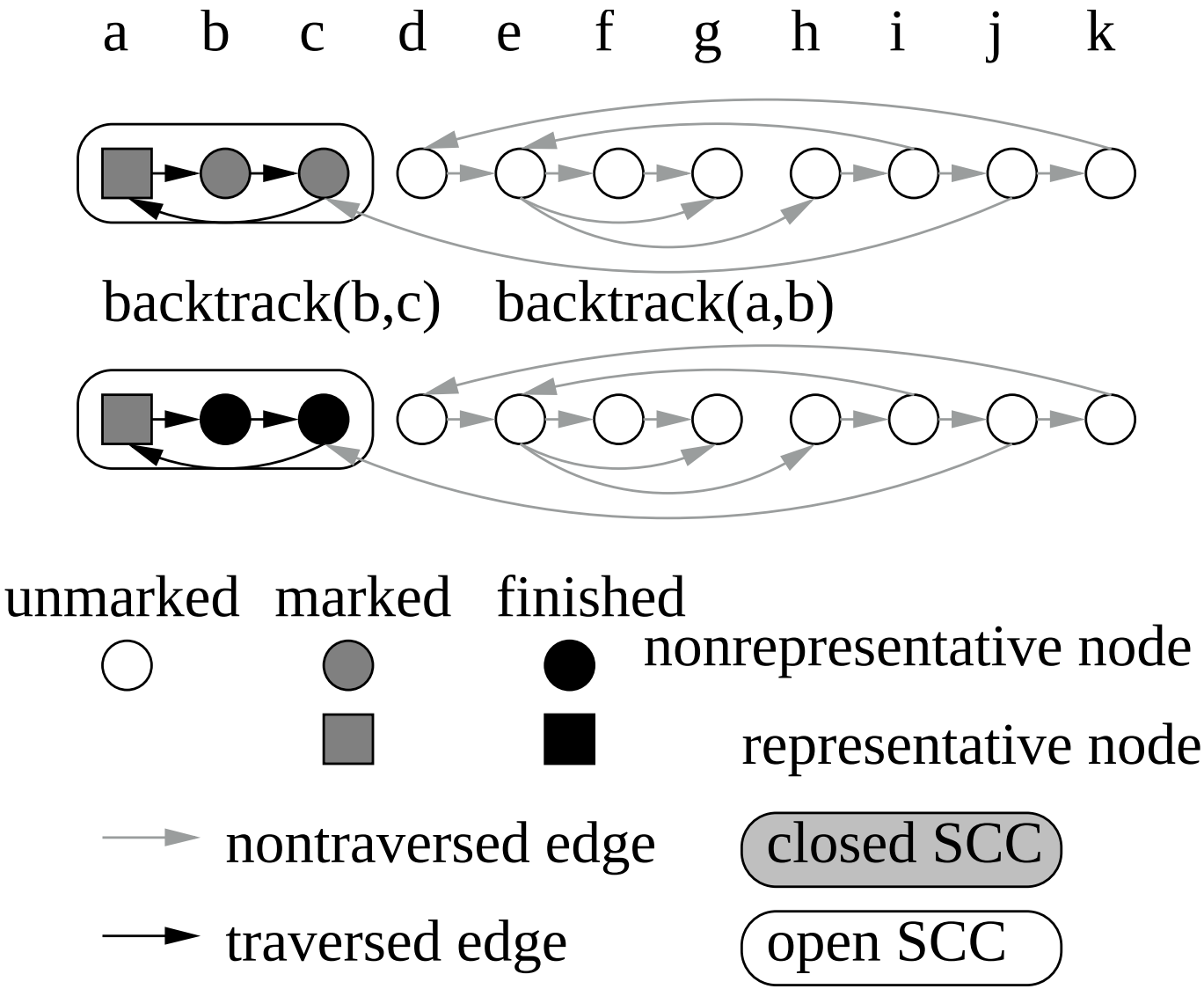
OK. (S_k wird ggf. entfernt)

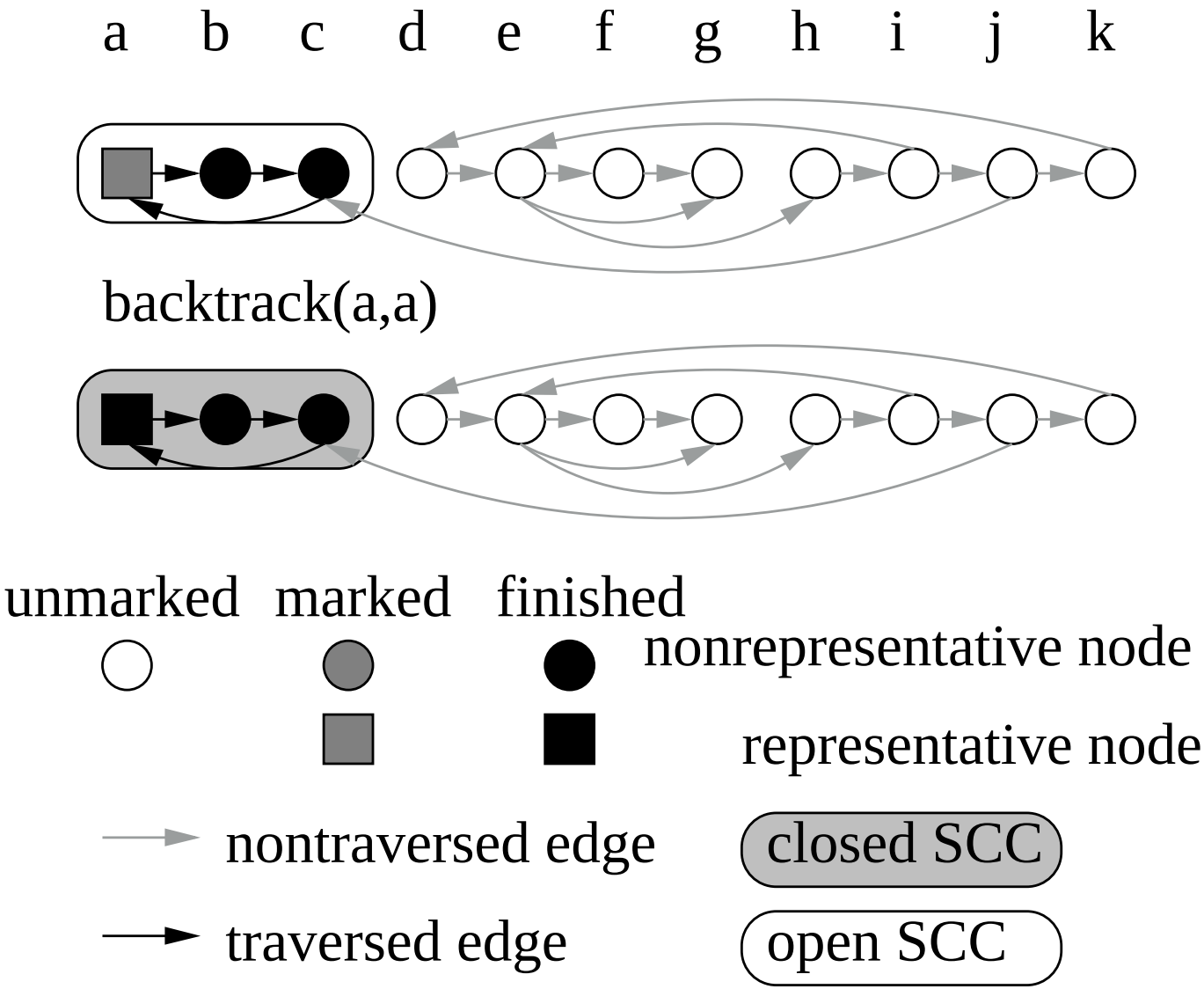


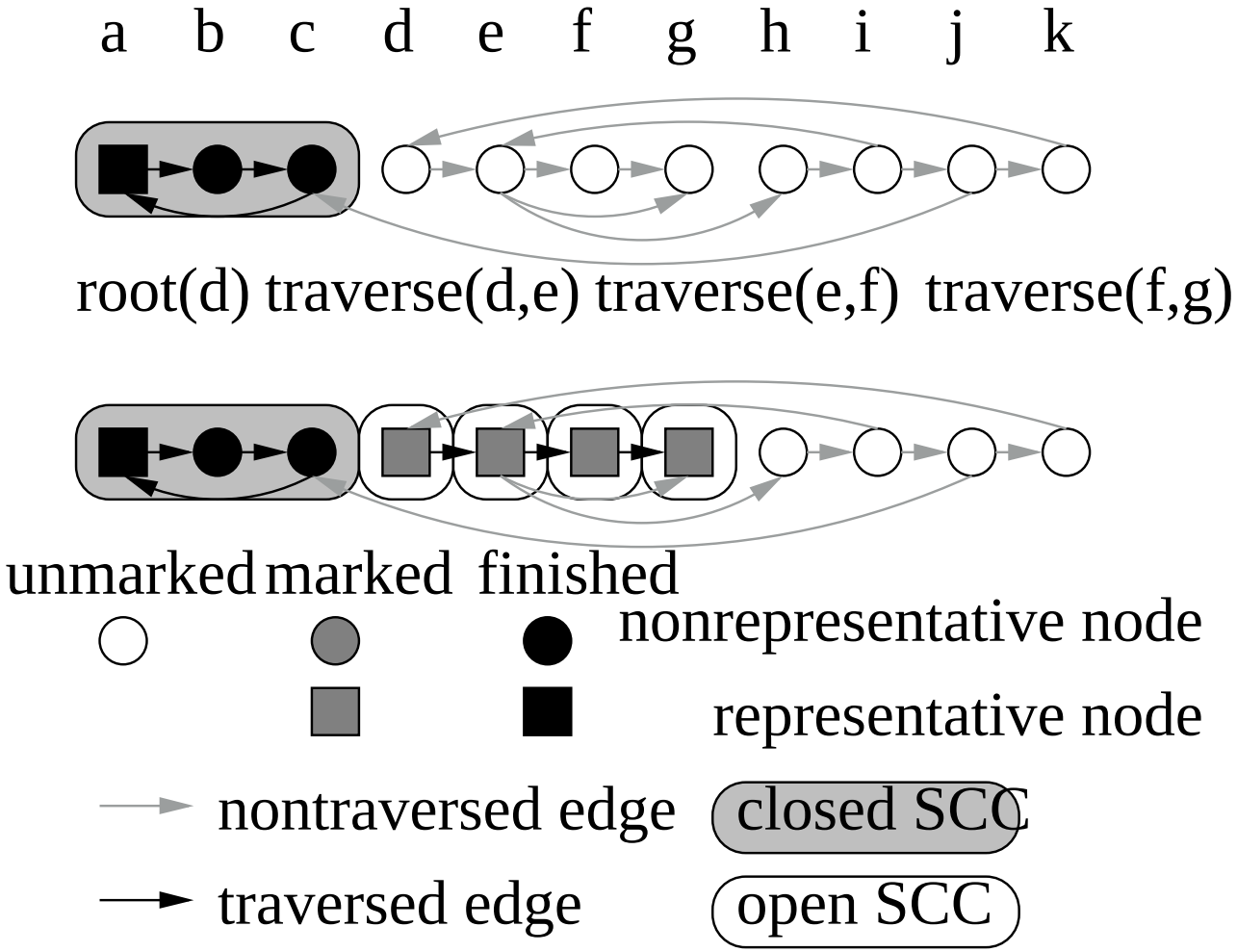
Beispiel

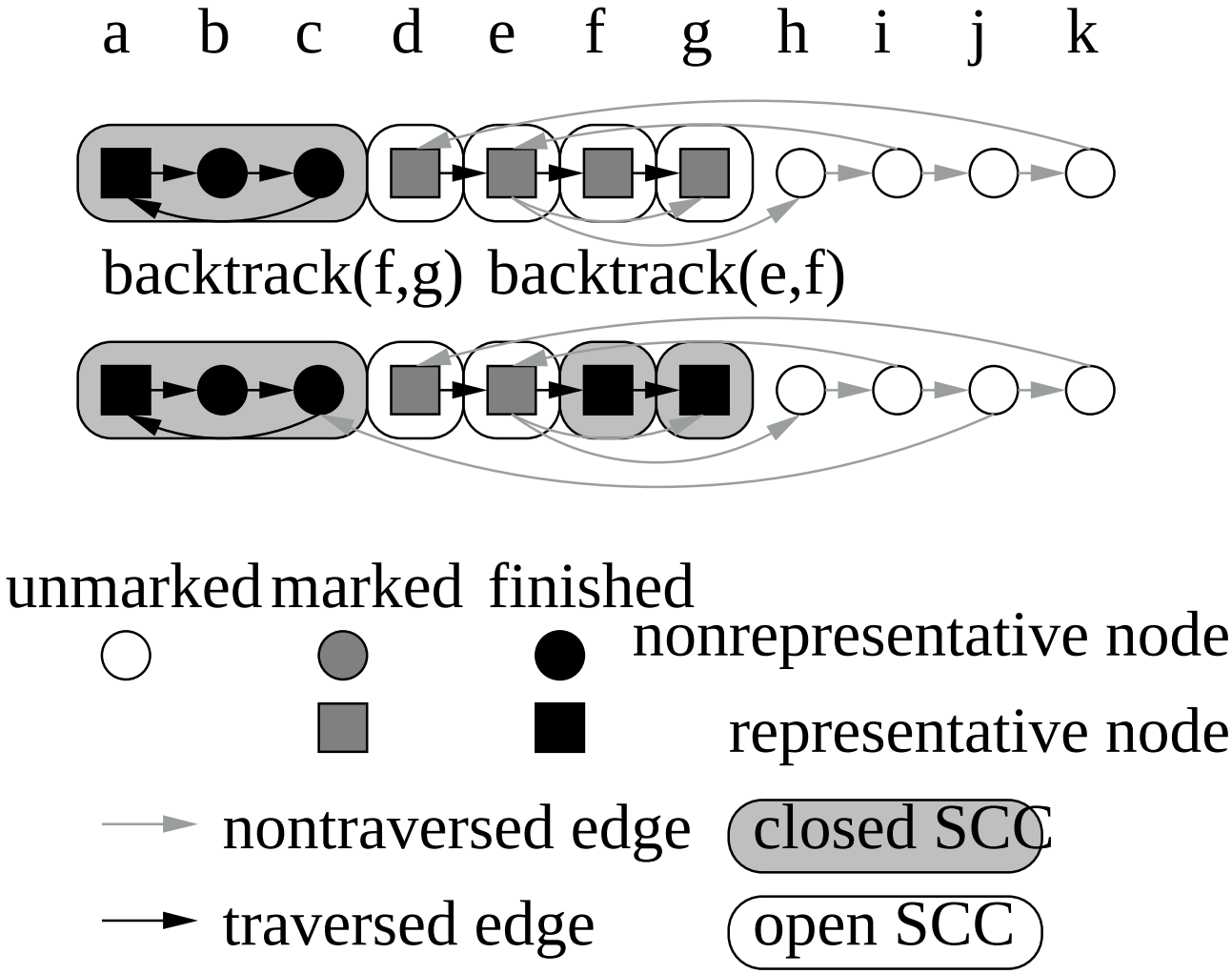

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

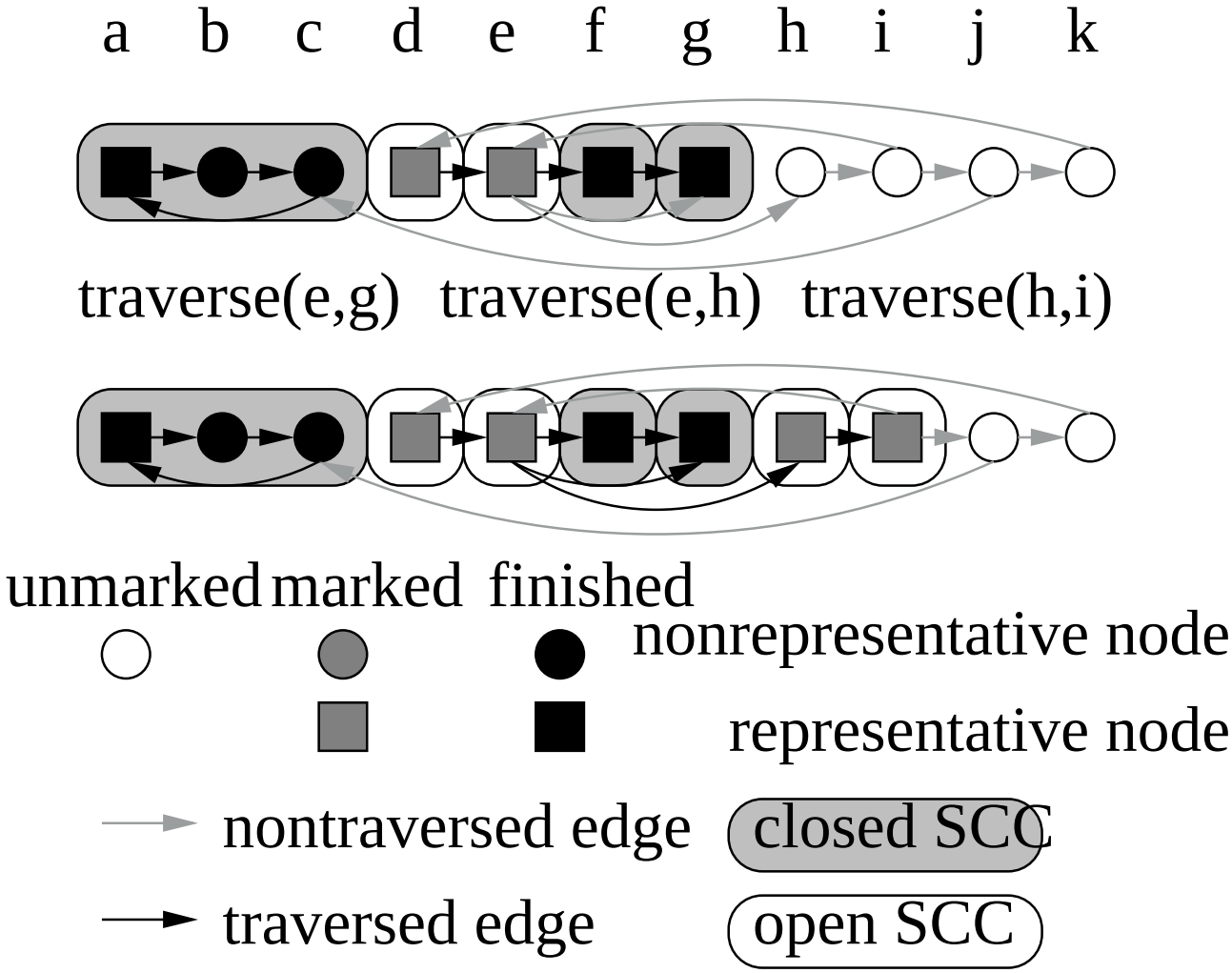

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
traverse(c,a)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

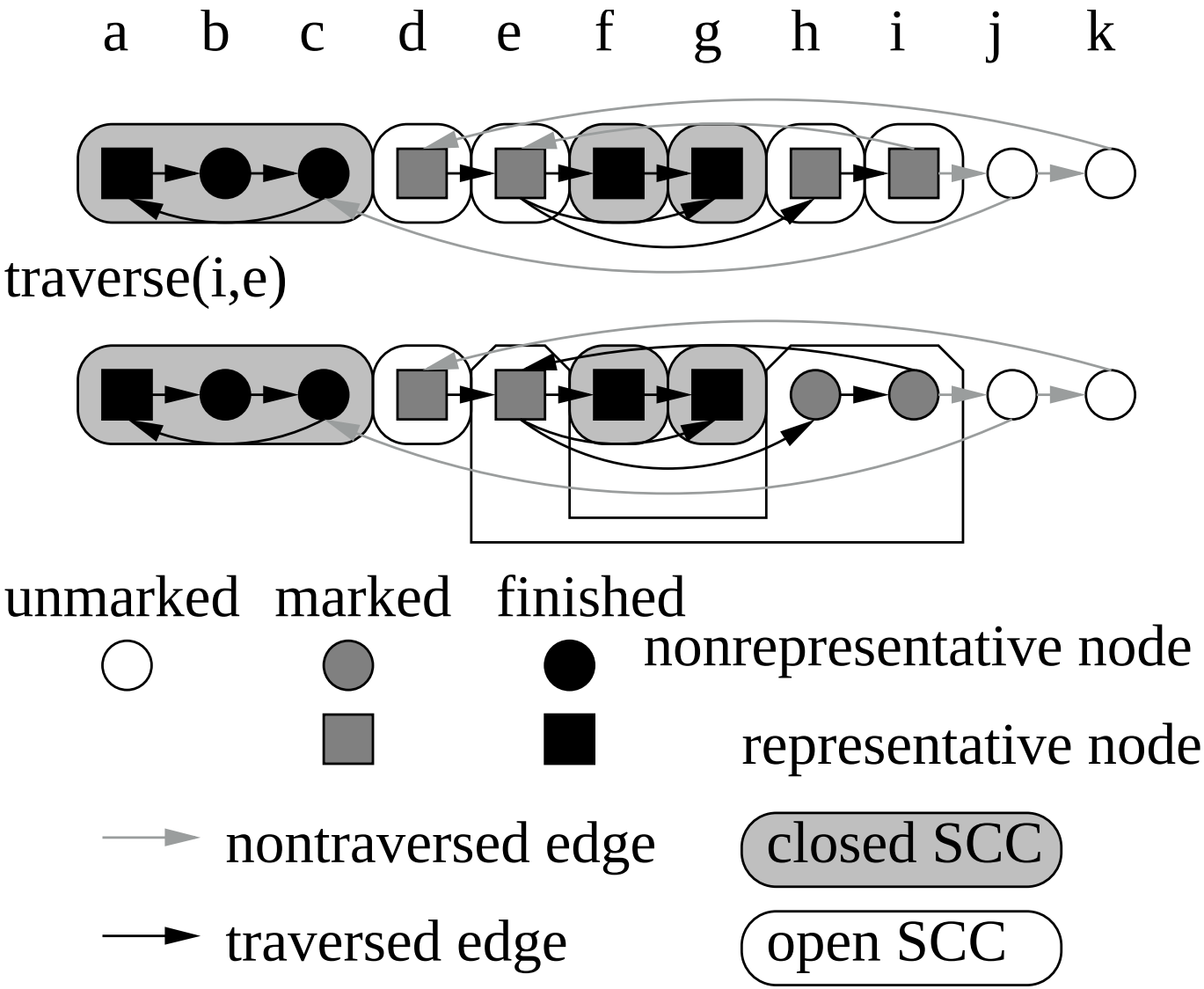

a b c d e f g h i j k
backtrack(b,c)
backtrack(a,b)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

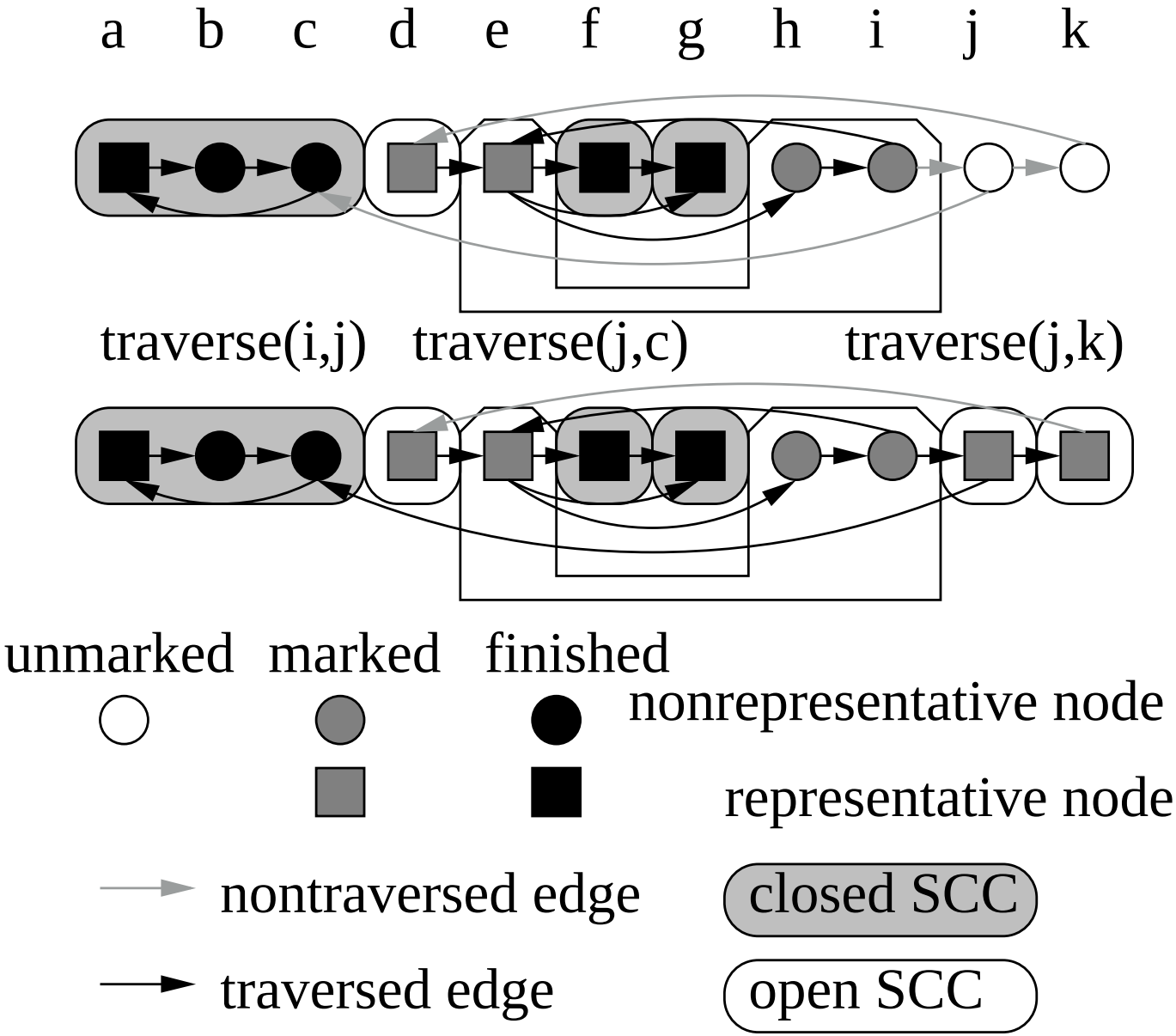

a b c d e f g h i j k
backtrack(a,a)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

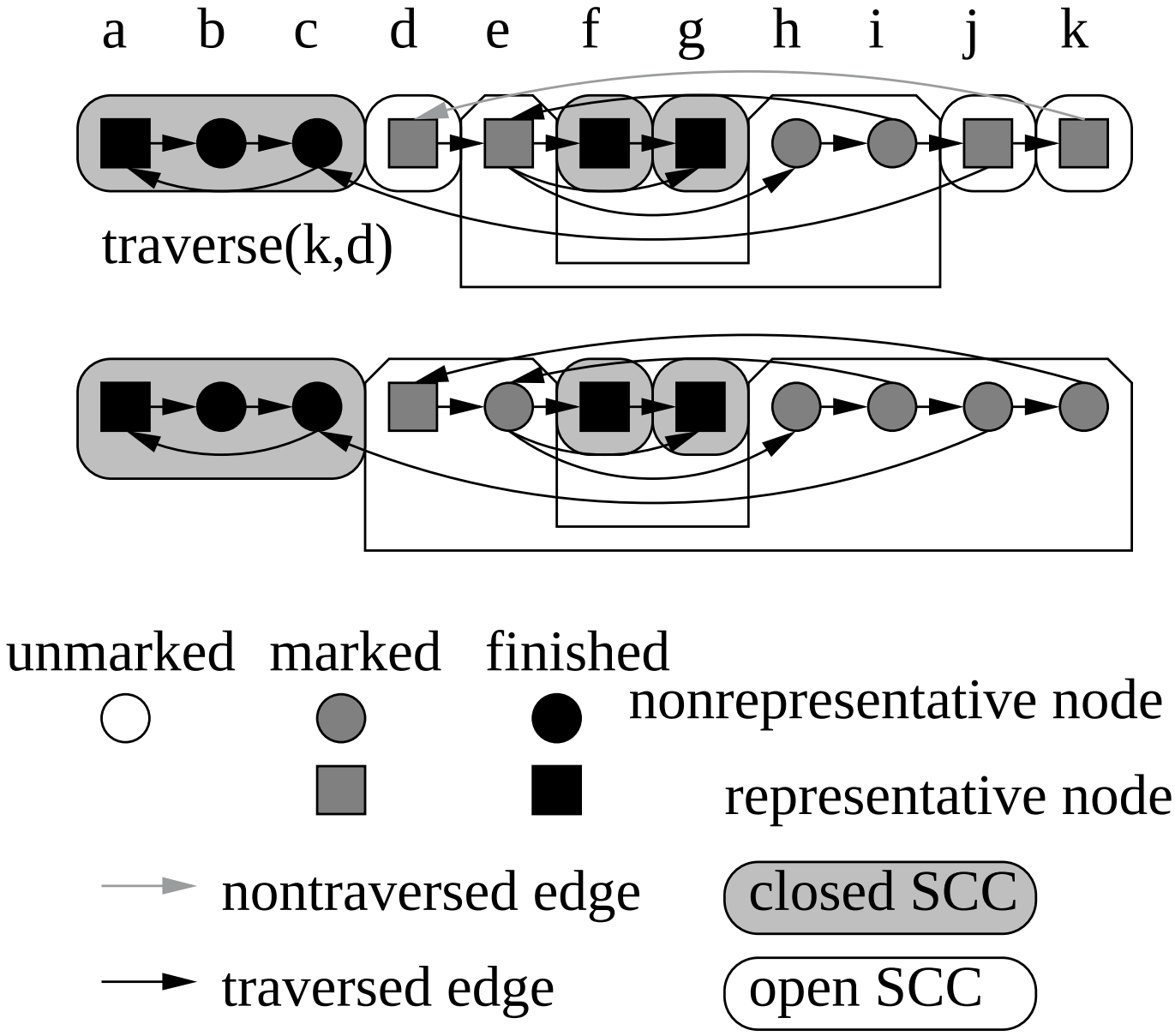

a b c d e f g h i j k
root(d) traverse(d,e) traverse(e,f) traverse(f,g)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

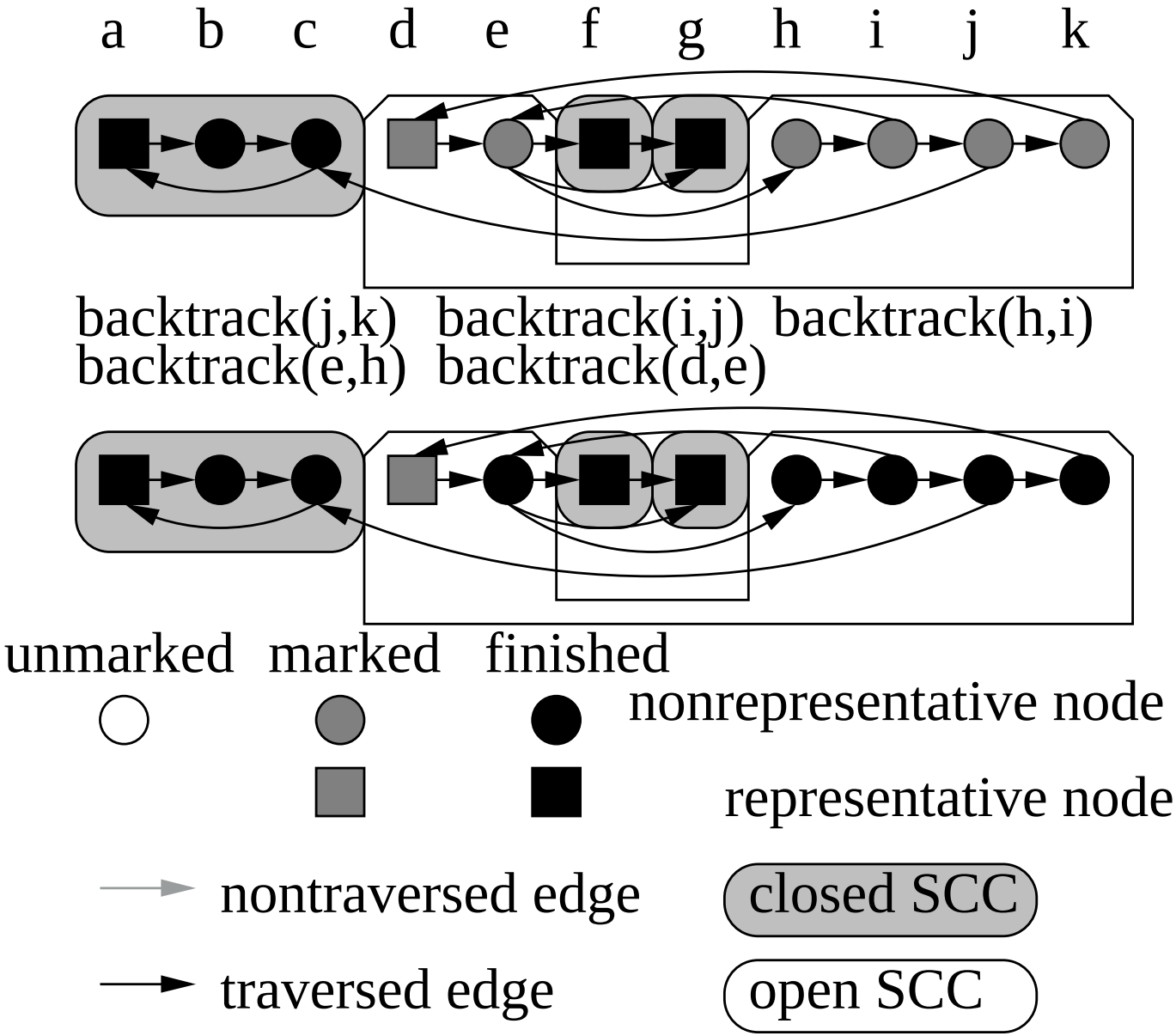

a b c d e f g h i j k
backtrack(f,g)
backtrack(e,f)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

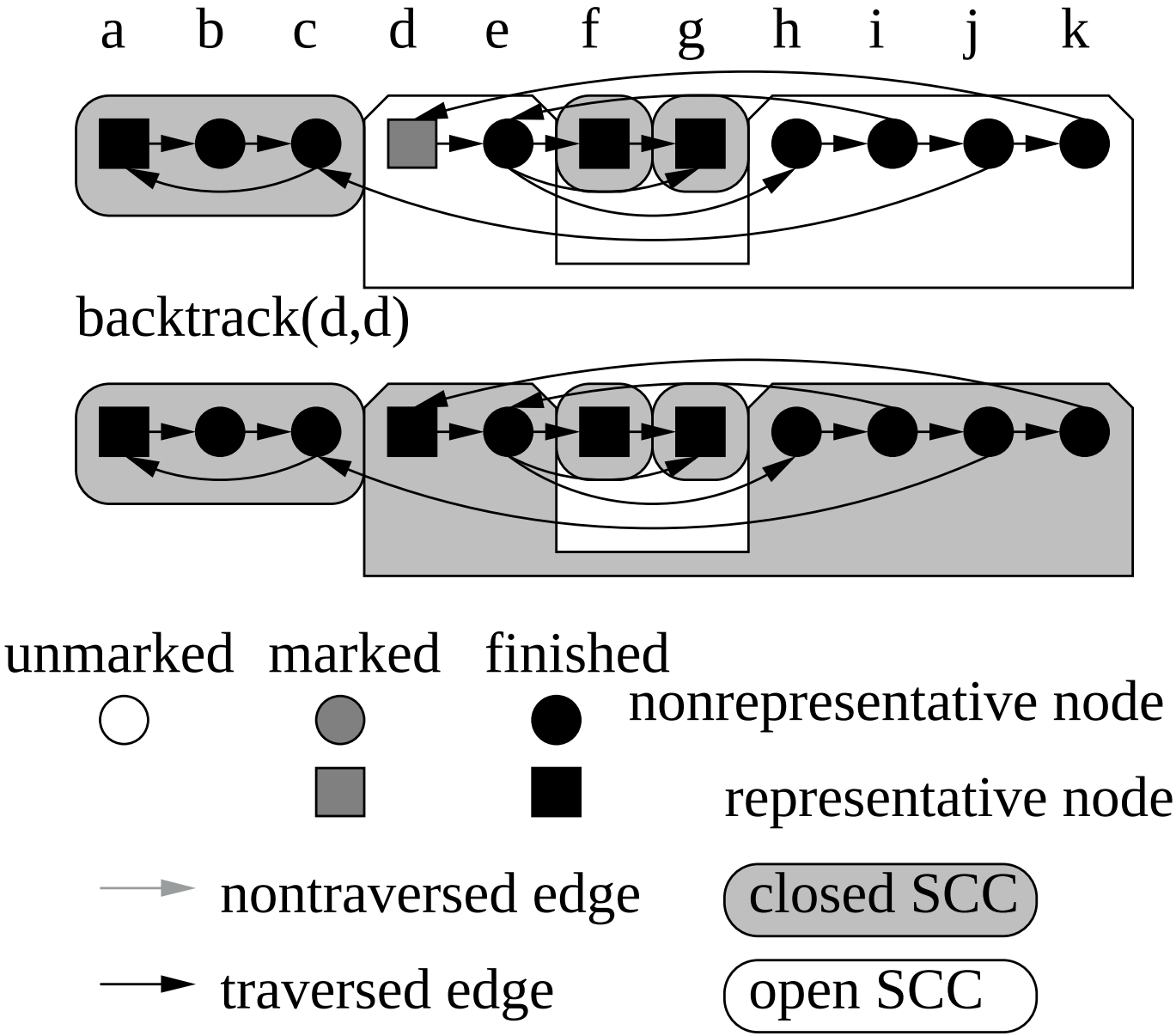

a b c d e f g h i j k
traverse(e,g) traverse(e,h) traverse(h,i)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(i,e)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(i,j)
traverse(j,c)
traverse(j,k)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(k,d)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
backtrack(j,k) backtrack(i,j) backtrack(h,i)
backtrack(e,h) backtrack(d,e)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
backtrack(d,d)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

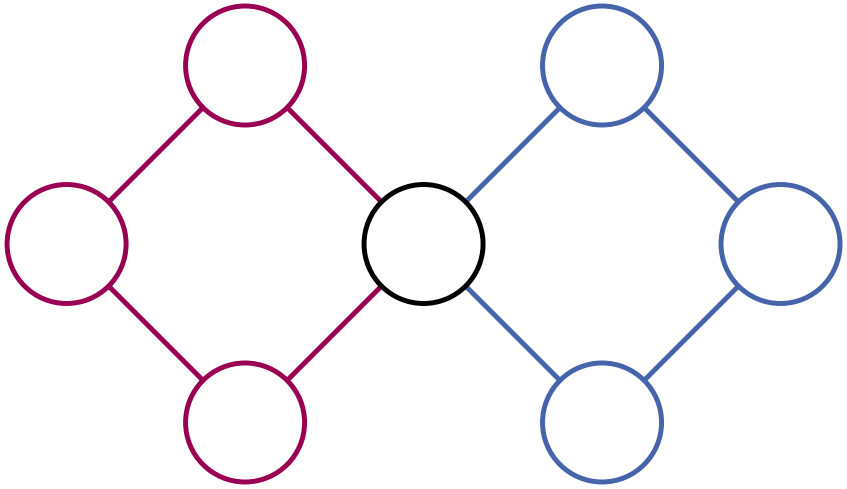
Zusammenfassung: SCC Berechnung

- Einfache Instantiierung des DFS-Musters
- Nichttrivialer Korrektheitsbeweis
- Laufzeit $\mathrm{O}(m+n)$: (Jeweils max. n push/pop Operationen)
- Ein einziger Durchlauf

2-zusammenhängende Komponenten (ungerichtet)

Bei entfernen eines Knotens bleibt die Komponente zusammenhängend.

(Partitionierung der Kanten)



Geht in Zeit $\mathrm{O}(m+n)$ mit Algorithmus ähnlich zu SCC-Algorithmus

Mehr DFS-basierte Linearzeitalgorithmen

- 3-zusammenhängende Komponenten
- Planaritätstest
- Einbettung planarer Graphen

14 Maximum Flows and Matchings

[mit Kurz Mehlhorn, Rob van Stee]

Folien auf Englisch

Literatur:

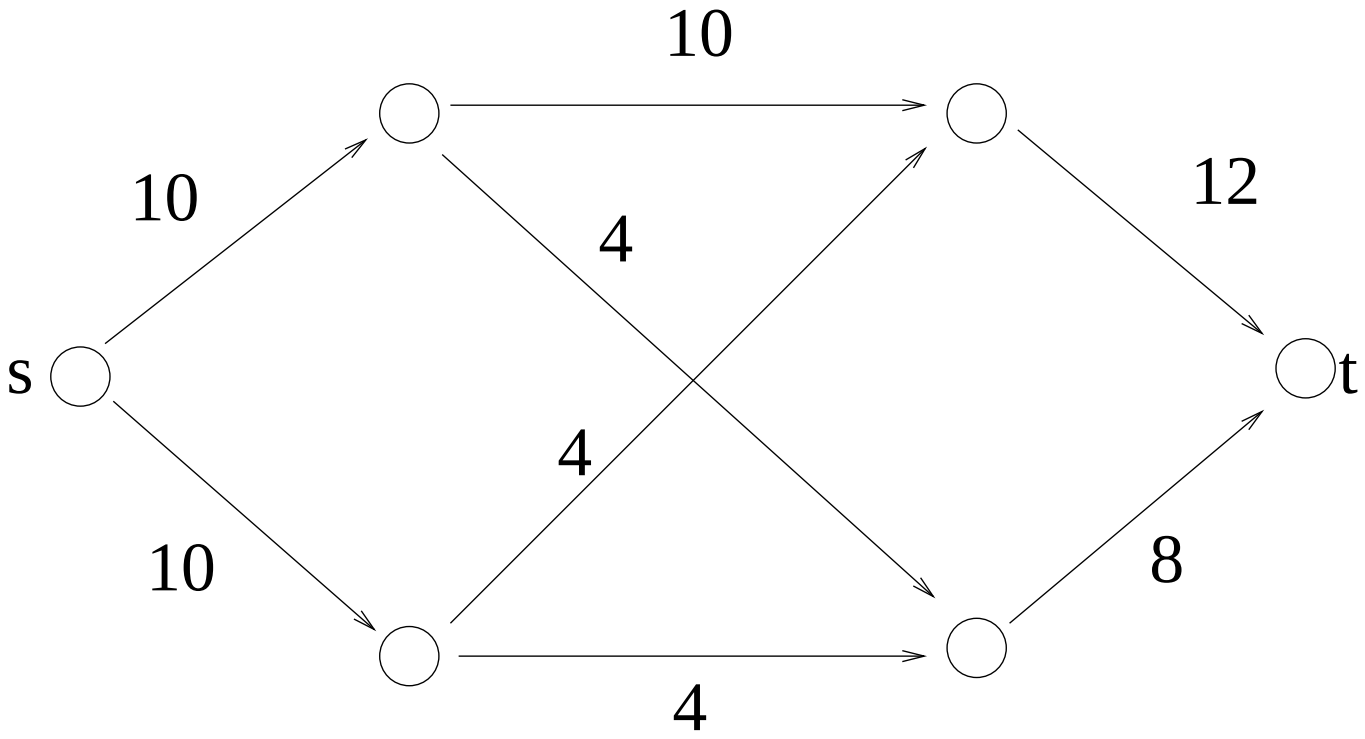
[Mehlhorn / Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999]

`http://www.mpi-inf.mpg.de/~mehlhorn/ftp/LEDAbook/Graph_alg.ps`

[Ahuja, Magnanti, Orlin, Network Flows, Prentice Hall, 1993]

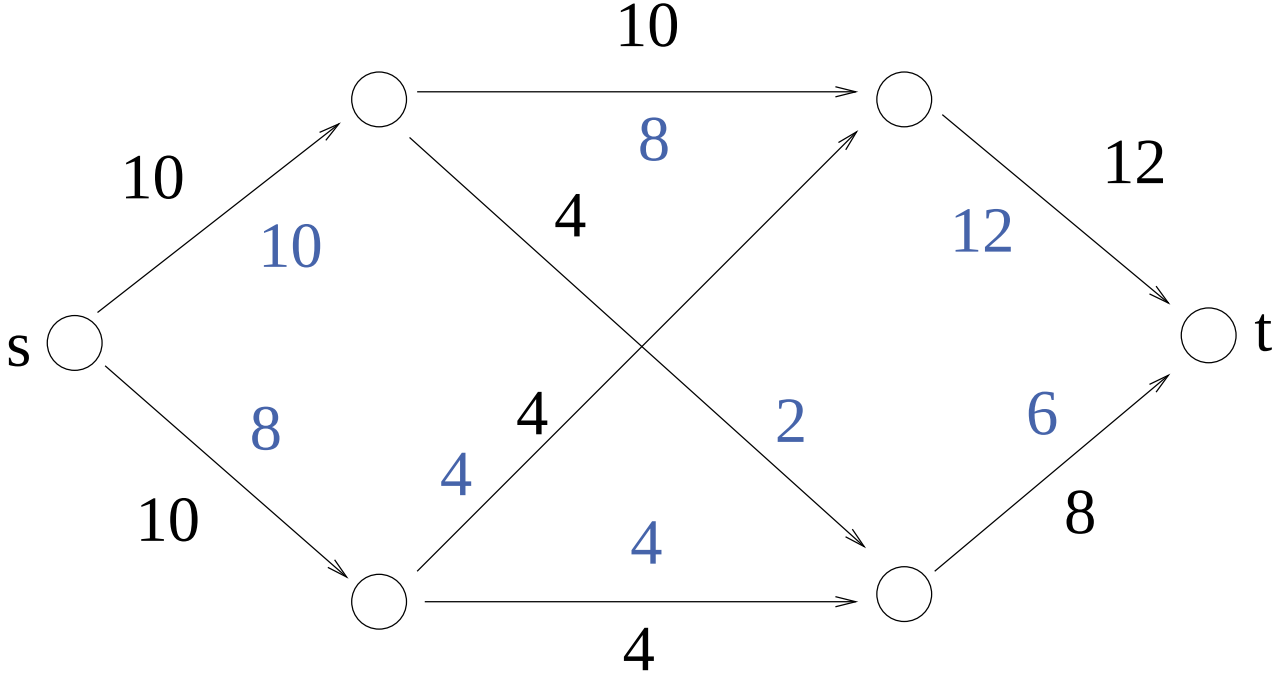
Definitions: Network

- Network = directed weighted graph with
  source node s and sink node t
- s has no incoming edges, t has no outgoing edges
- Weight c_e of an edge e = capacity of e (nonnegative!)



Definitions: Flows

- Flow = function f_e on the edges, $0 \leq f_e \leq c_e \forall e$
  $\forall v \in V \setminus \{s,t\}$: total incoming flow = total outgoing flow
- Value of a flow $\mathbf{val}(f)$ = total outgoing flow from s = total flow going into t
- Goal: find a flow with maximum value

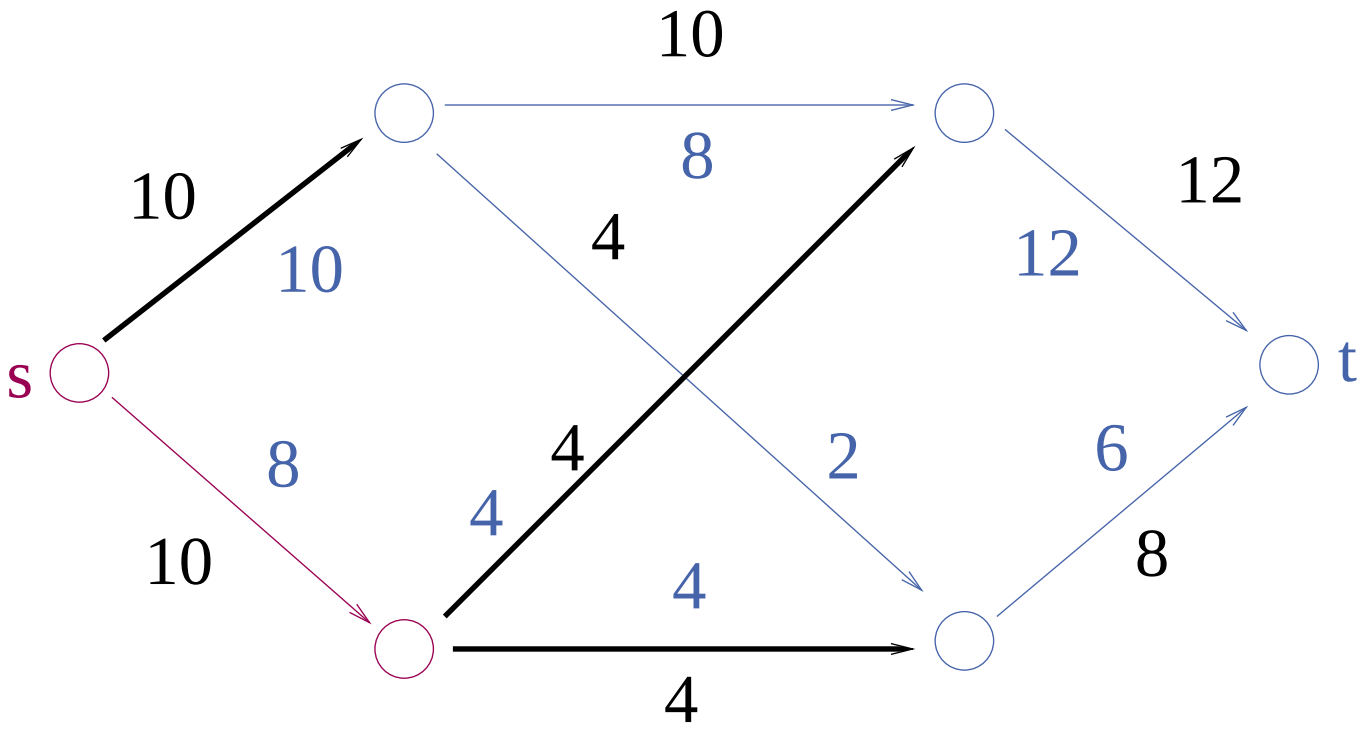


Definitions: (Minimum) s-t Cuts

An s-t cut is partition of V into S and T with $s \in S$ and $t \in T$.

The capacity of this cut is:

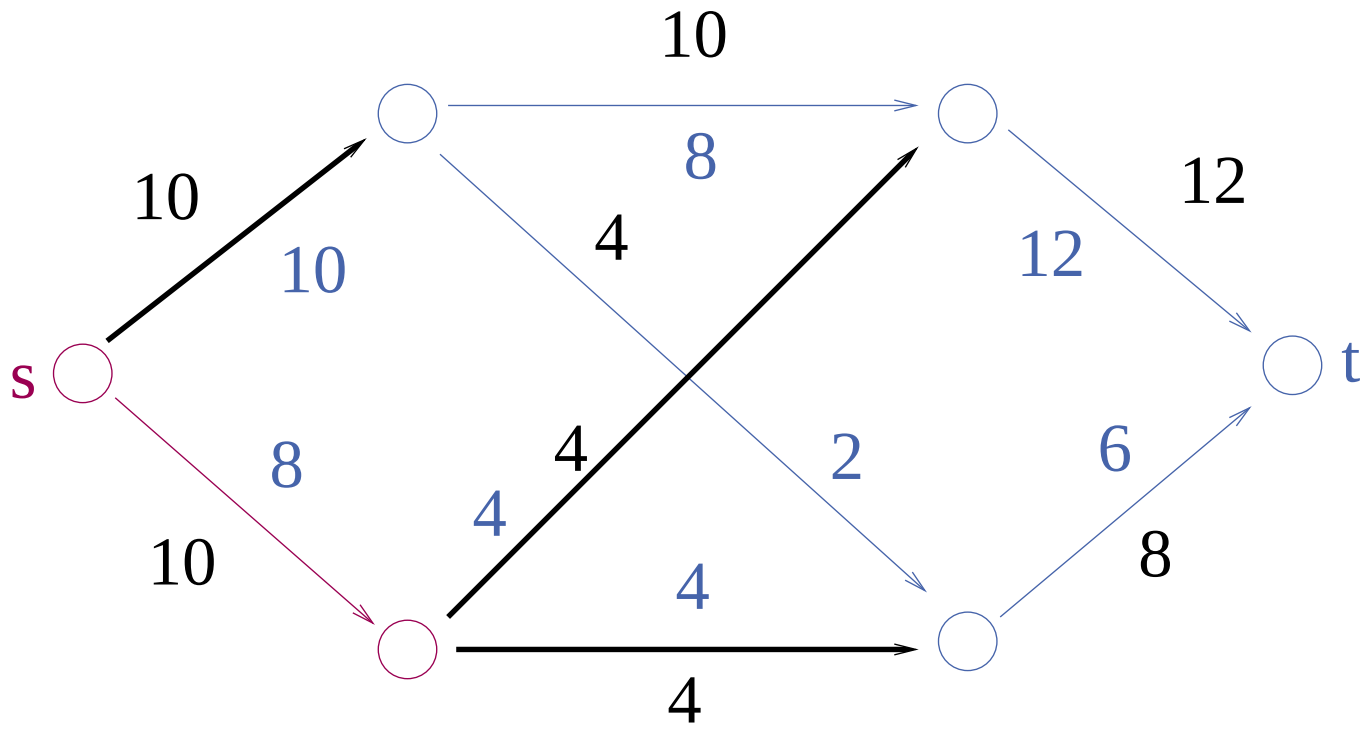
$$\sum \{c_{(u,v)} : u \in S, v \in T\}$$



Duality Between Flows and Cuts

Theorem:[Elias/Feinstein/Shannon, Ford/Fulkerson 1956]

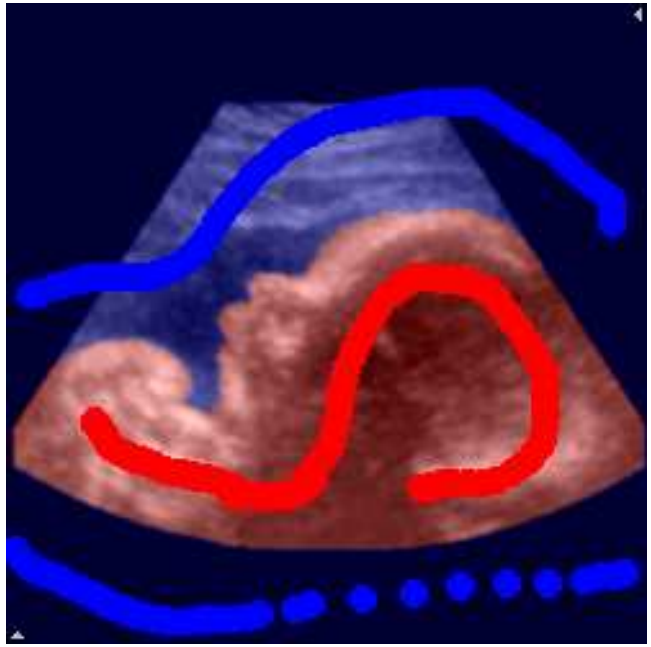
Value of an s-t max-flow $=$ minimum capacity of an s-t cut.



Proof: later

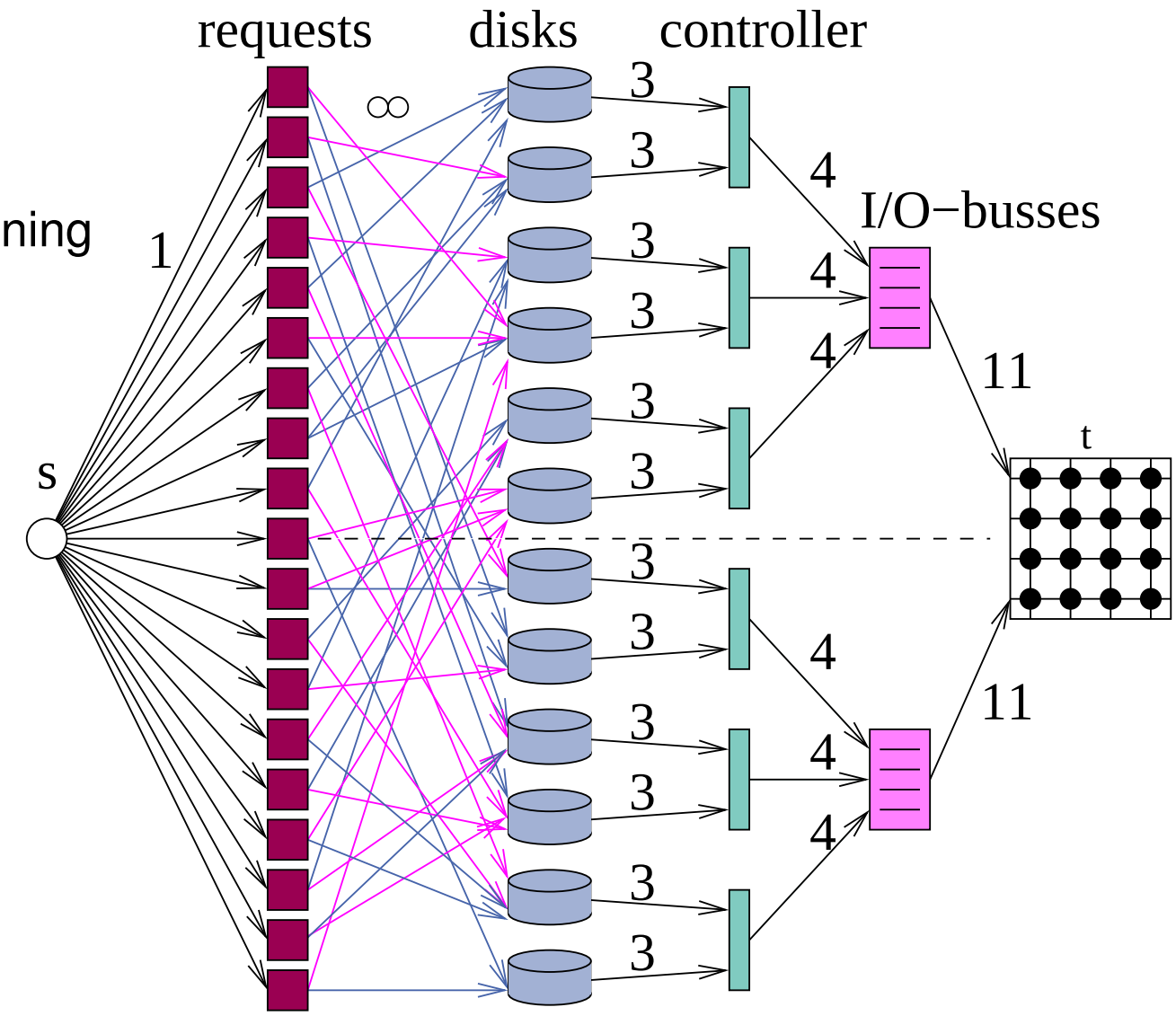
Applications

- Oil pipes
- Traffic flows on highways
- Image Processing `http://vision.csd.uwo.ca/maxflow-data`
  - segmentation
  - stereo processing
  - multiview reconstruction
  - surface fitting
- disk/machine/tanker scheduling
- matrix rounding
- ...



Applications in our Group

- multicasting using network coding
- balanced k partitioning
- disk scheduling



Option 1: linear programming

- Flow variables x_e for each edge e
- Flow on each edge is at most its capacity
- Incoming flow at each vertex = outgoing flow from this vertex
- Maximize outgoing flow from starting vertex

We can do better!

Algorithms 1956–now

Year	Author	Running time
1956	Ford-Fulkerson	$O(mnU)$
1969	Edmonds-Karp	$O(m^2n)$
1970	Dinic	$O(mn^2)$
1973	Dinic-Gabow	$O(mn\log U)$
1974	Karzanov	$O(n^3)$
1977	Cherkassky	$O(n^2\sqrt{m})$
1980	Galil-Naamad	$O(mn\log^2 n)$
1983	Sleator-Tarjan	$O(mn\log n)$

$n =$ number of nodes

$m =$ number of arcs

$U =$ largest capacity

Year	Author	Running time
1986	Goldberg-Tarjan	$O(mn\log(n^2/m))$
1987	Ahuja-Orlin	$O(mn+n^2\log U)$
1987	Ahuja-Orlin-Tarjan	$O(mn\log(2+n\sqrt{\log U}/m))$
1990	Cheriyan-Hagerup-Mehlhorn	$O(n^3/\log n)$
1990	Alon	$O(mn+n^{8/3}\log n)$
1992	King-Rao-Tarjan	$O(mn+n^{2+e})$
1993	Philipps-Westbrook	$O(mn\log n/\log\frac{m}{n}+n^2\log^{2+\varepsilon} n)$
1994	King-Rao-Tarjan	$O(mn\log n/\log\frac{m}{n\log n})$ if $m \geq 2n\log$
1997	Goldberg-Rao	$O(\min\{m^{1/2},n^{2/3}\}m\log(n^2/m)\log$

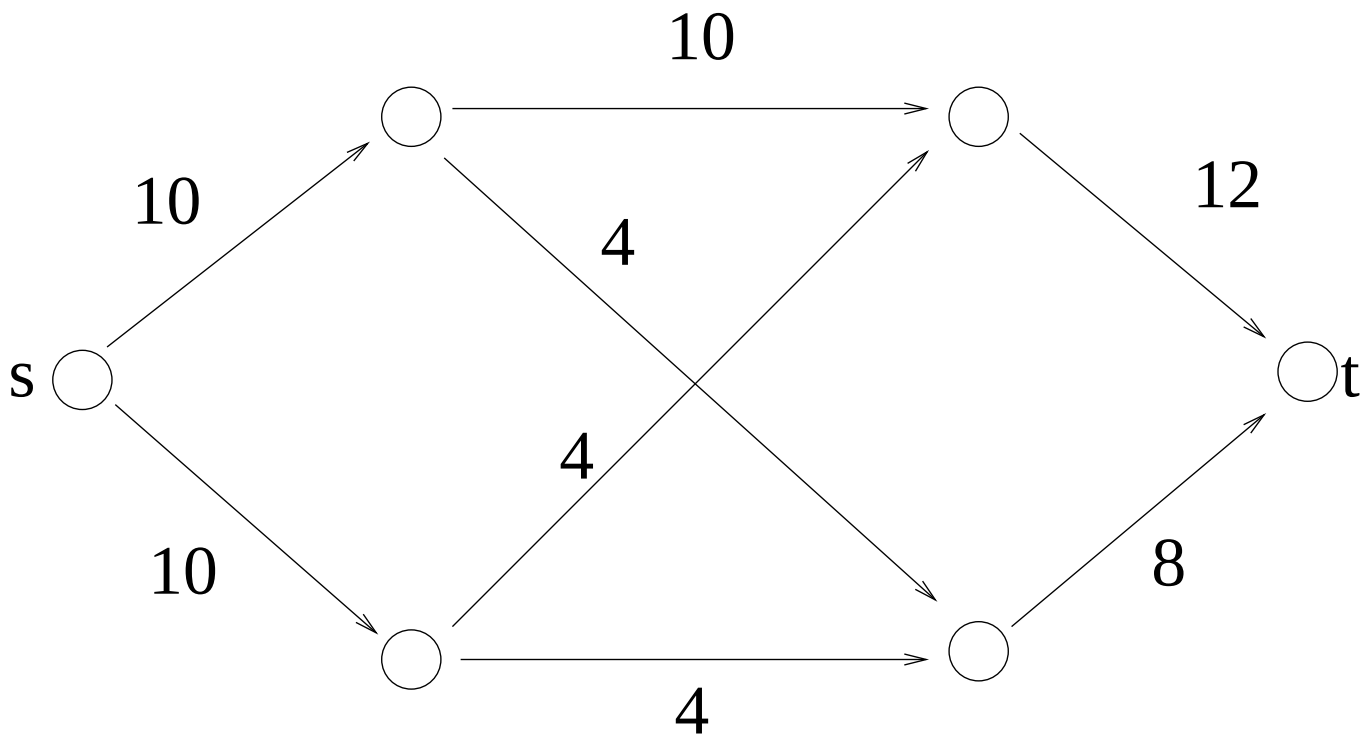
Augmenting Paths (Rough Idea)

Find a path from s to t such that each edge has some spare capacity

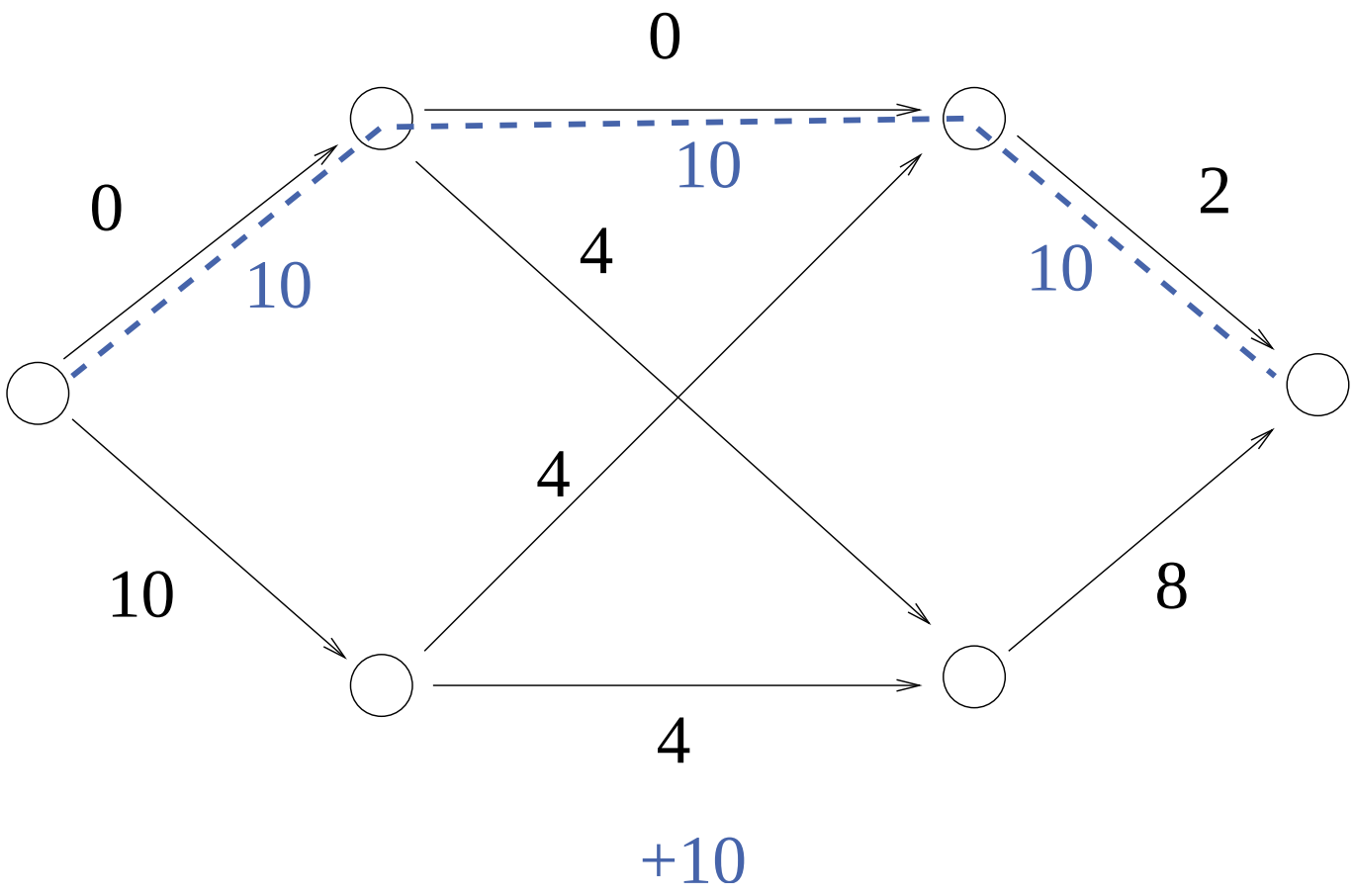
On this path, saturate the edge with the smallest spare capacity

Adjust capacities for all edges (create residual graph) and repeat

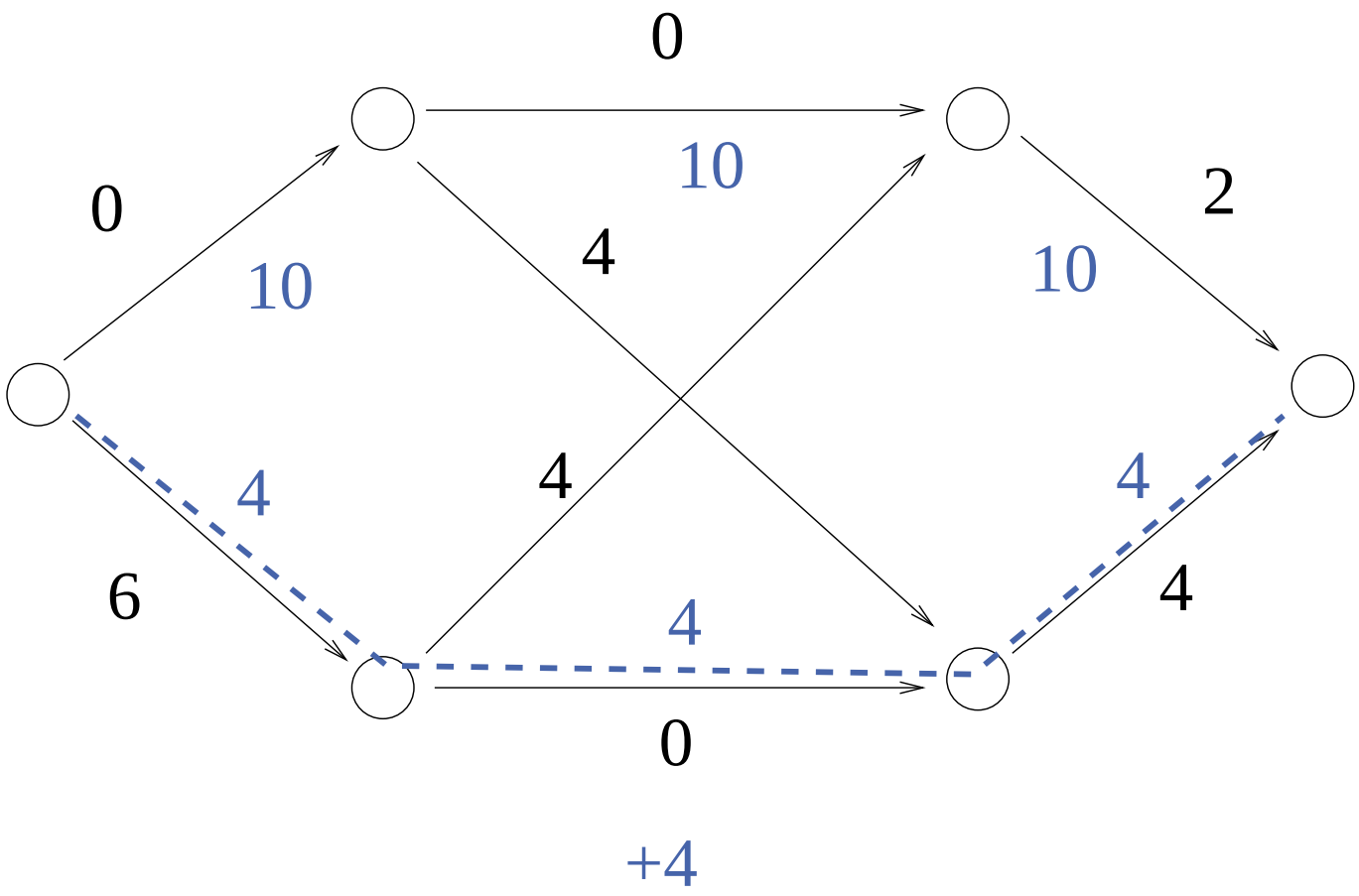
Example



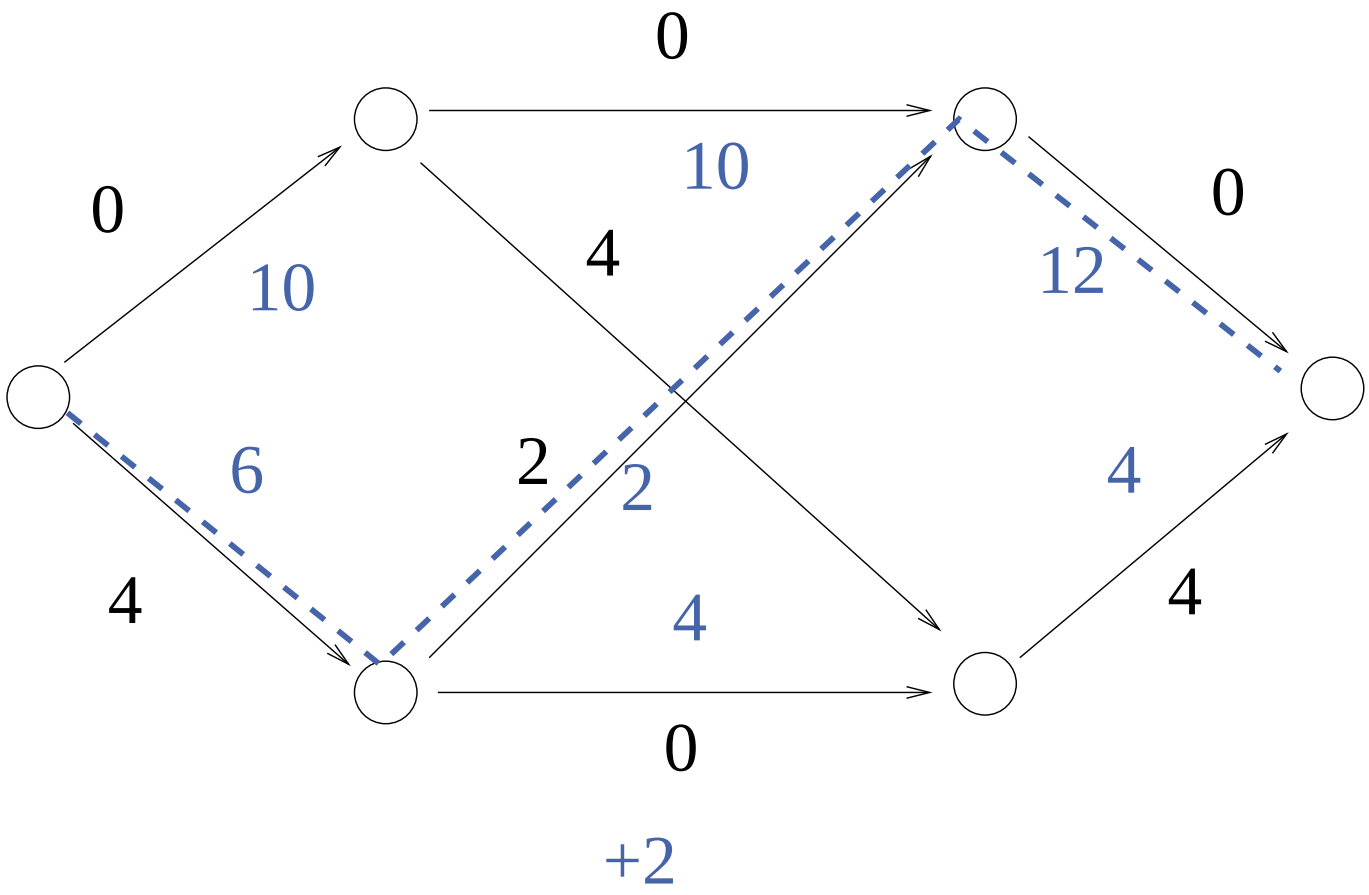
Example



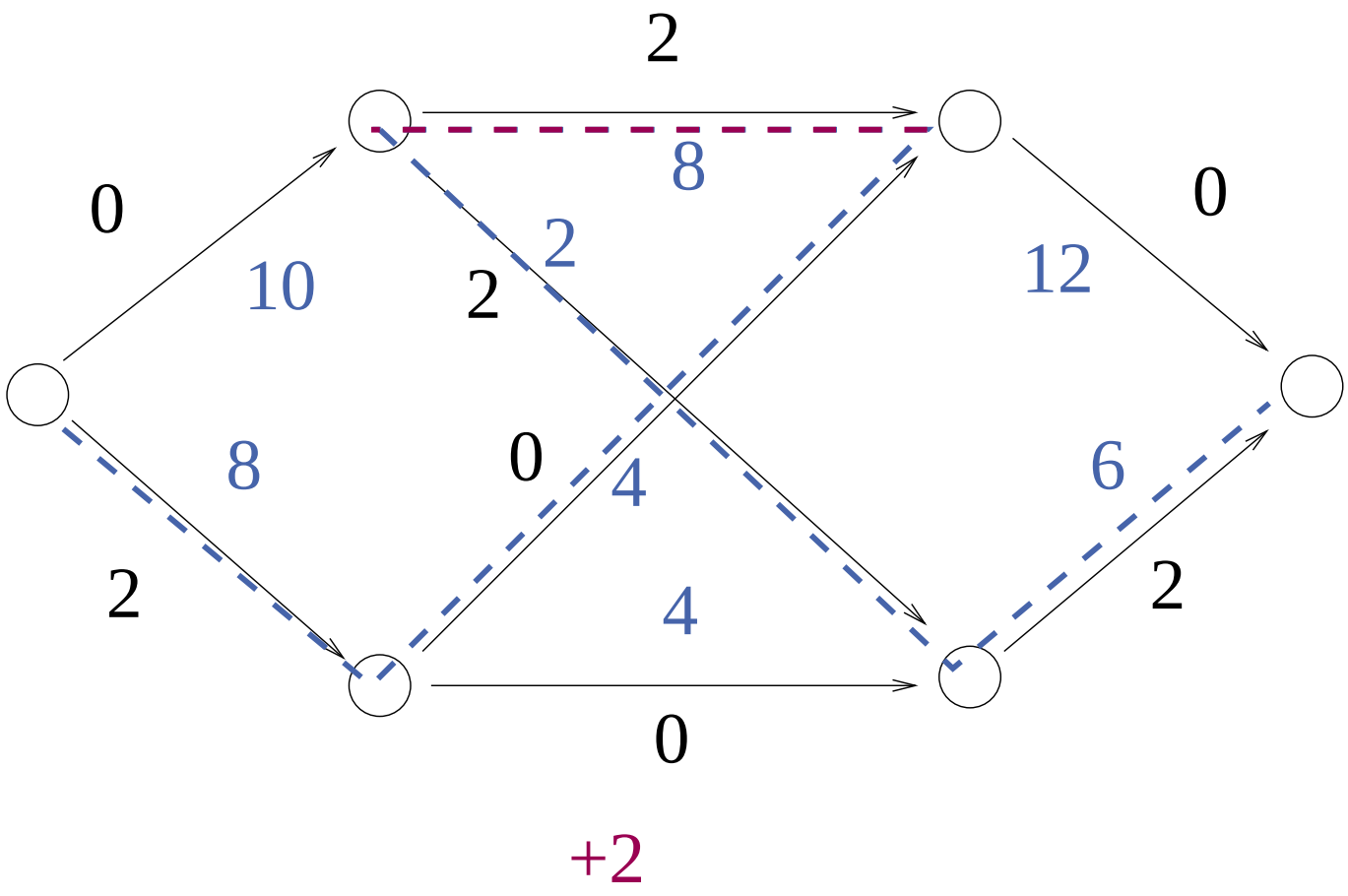
Example



Example



Example

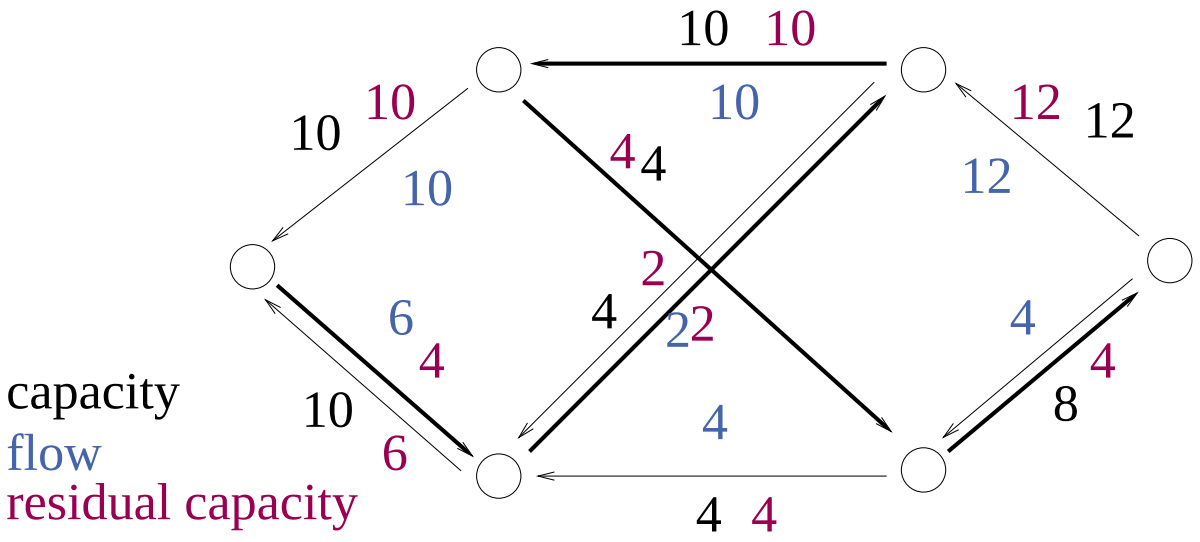


Residual Graph

Given, network $G = (V, E, c)$, flow f

Residual graph $G_f = (V, E_f, c^f)$. For each $e \in E$ we have

$$\begin{cases} e \in E_f \text{ with } c_e^f = c_e - f(e) & \text{if } f(e) < c(e) \\ e^{\text{rev}} \in E_f \text{ with } c_{e^{\text{rev}}}^f = f(e) & \text{if } f(e) > 0 \end{cases}$$



Augmenting Paths

Find a path p from s to t such that each edge e has nonzero residual capacity c_e^f

$\Delta f := \min_{e \in p} c_e^f$

foreach $(u,v) \in p$ **do**

  if $(u,v) \in E$ **then** $f_{(u,v)} += \Delta f$

  else $f_{(v,u)} -= \Delta f$

Ford Fulkerson Algorithm

Function FFMaxFlow($G = (V,E), s, t, \mathsf{c} : E \to \mathbb{N}) : E \to \mathbb{N}$

$f := 0$

while $\exists$path $p = (s, \ldots, t)$ in G_f **do**

augment f along p

return f

time $O(m\mathrm{val}(f))$

Ford Fulkerson – Correctness

"Clearly" FF computes a feasible flow f. (Invariant)

Todo: flow value is maximal

At termination: no augmenting paths in G_f left.

Consider cut $(S, V \setminus S)$ with
$S := \{v \in V : v \text{ reachable from } s \text{ in } G_f\}$

Some Basic Observations

Lemma 1: For any cut (S,T):

$$\mathbf{val}(f) = \overbrace{\sum_{e \in E \cap S \times T} f_e}^{S \to T \text{ edges}} - \overbrace{\sum_{e \in E \cap T \times S} f_e}^{T \to S \text{ edges}} .$$

Lemma 2: $\forall (u,v) \in E : c^f_{(u,v)} = 0 \Rightarrow f_{(v,u)} = 0$

Ford Fulkerson – Correctness

Todo: $\mathbf{val}(f)$ is maximal when no augmenting paths in G_f left.

Consider cut $(S, V \setminus S)$ with S:= $\{v \in V : v \text{ reachable from } s \text{ in } G_f\}$.

Observation: $\forall (u,v) \in E \cap S \times T : c_e^f = 0$ and hence $f_{(v,u)} = 0$

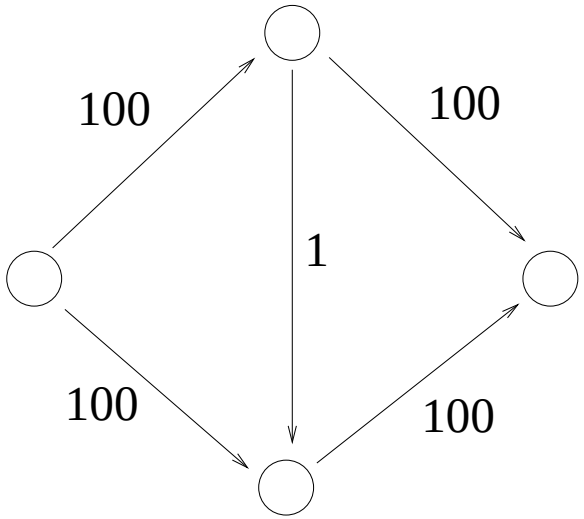
Lemma 2.

Now, by Lemma 1,

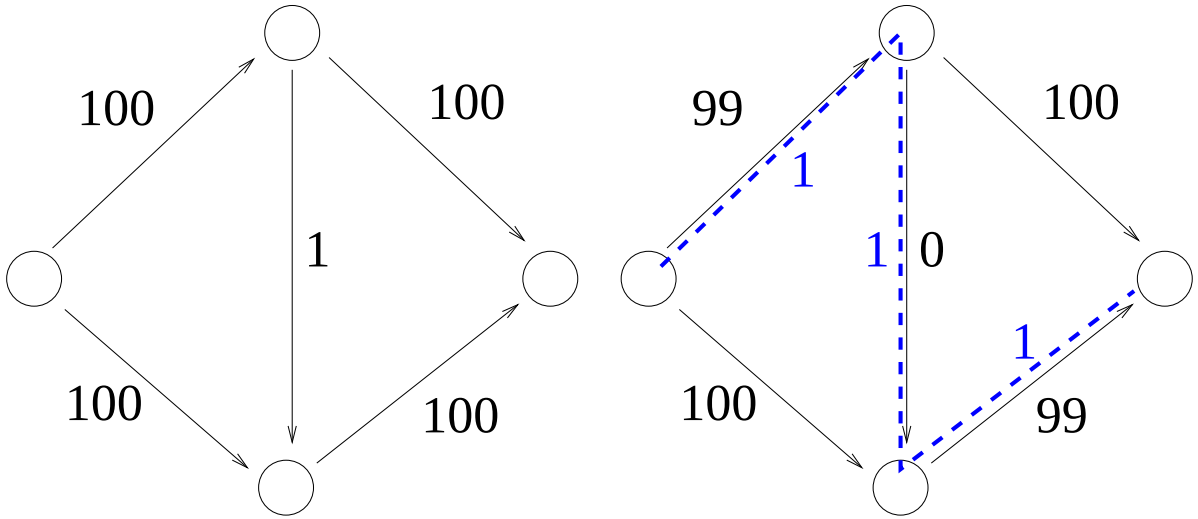
$$\begin{aligned}\mathbf{val}(f) &= \sum_{e \in E \cap S \times T} f_e - \sum_{e \in E \cap T \times S} f_e \\ &= \sum_{e \in E \cap S \times T} f_e = \text{cut capacity} \\ &\geq \text{max flow}\end{aligned}$$

Corollary: max flow = min cut

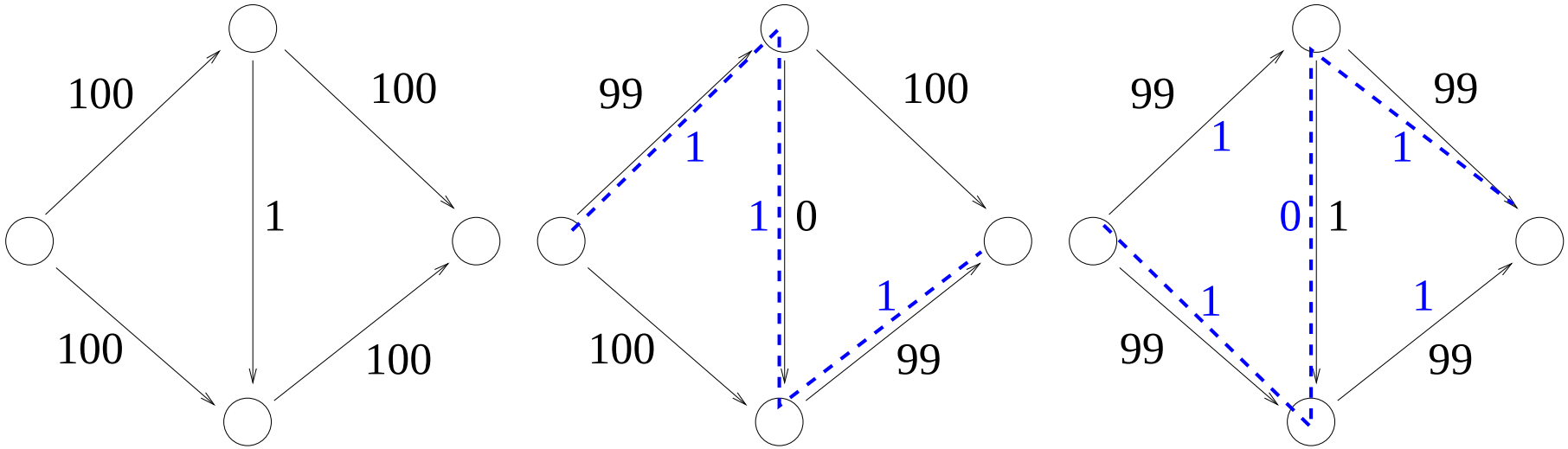
A Bad Example for Ford Fulkerson



A Bad Example for Ford Fulkerson



A Bad Example for Ford Fulkerson

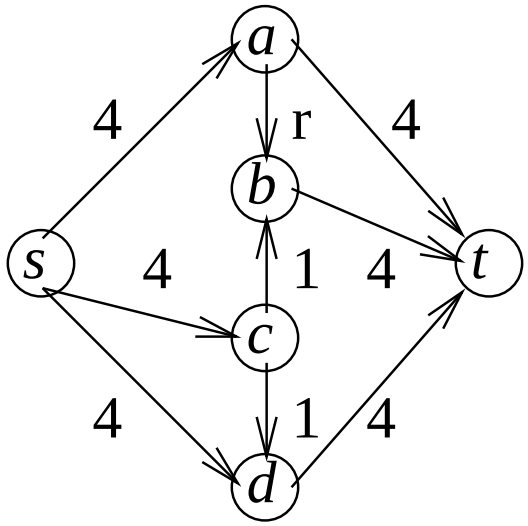


An Even Worse Example for Ford Fulkerson

[U. Zwick, TCS 148, p. 165–170, 1995]

Let $r = \frac{\sqrt{5}-1}{2}$.

Consider the graph



And the augmenting paths

$p_0 = \langle s, c, b, t \rangle$

$p_1 = \langle s, a, b, c, d, t \rangle$

$p_2 = \langle s, c, b, a, t \rangle$

$p_3 = \langle s, d, c, b, t \rangle$

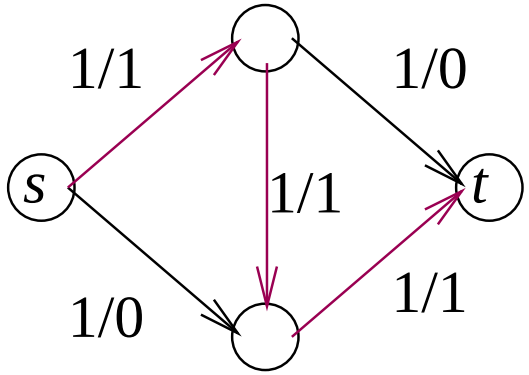
The sequence of augmenting paths $p_0(p_1, p_2, p_1, p_3)^*$ is an infinite sequence of positive flow augmentations.

The flow value does not converge to the maximum value 9.

Blocking Flows

f_b is a blocking flow in H if

$$\forall \text{paths } p = \langle s, \ldots, t \rangle : \exists e \in p : f_b(e) = c(e)$$



Dinitz Algorithm

Function DinitzMaxFlow($G = (V,E), s, t, c : E \to \mathbb{N}) : E \to \mathbb{N}$

$f := 0$

while $\exists$path $p = (s, \ldots, t)$ in G_f **do**

$d = G_f$.reverseBFS(t) $: V \to \mathbb{N}$

$L_f = (V, \{(u,v) \in E_f : d(v) = d(u) - 1\})$ // layer graph

find a blocking flow f_b in L_f

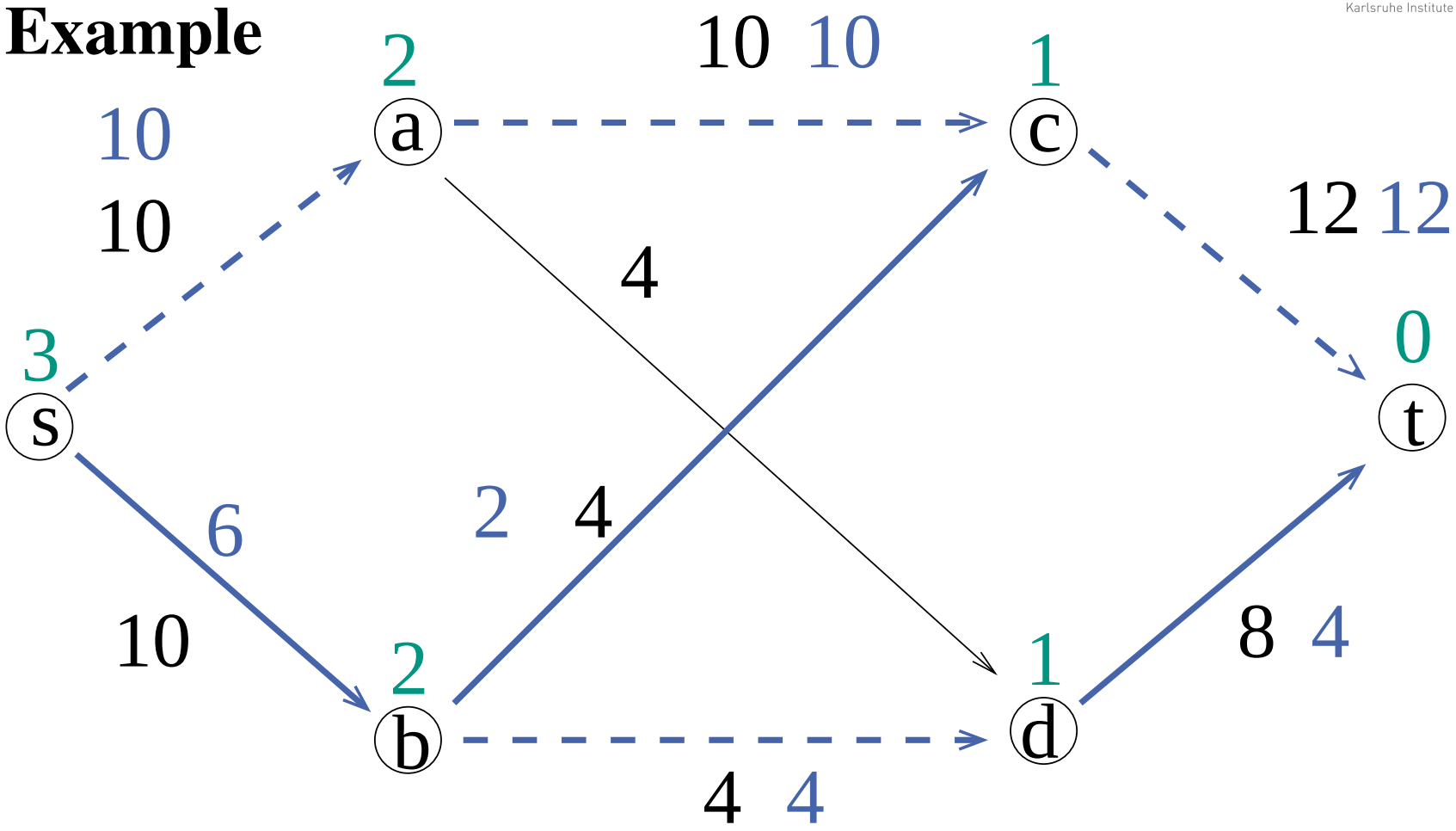
augment f += f_b

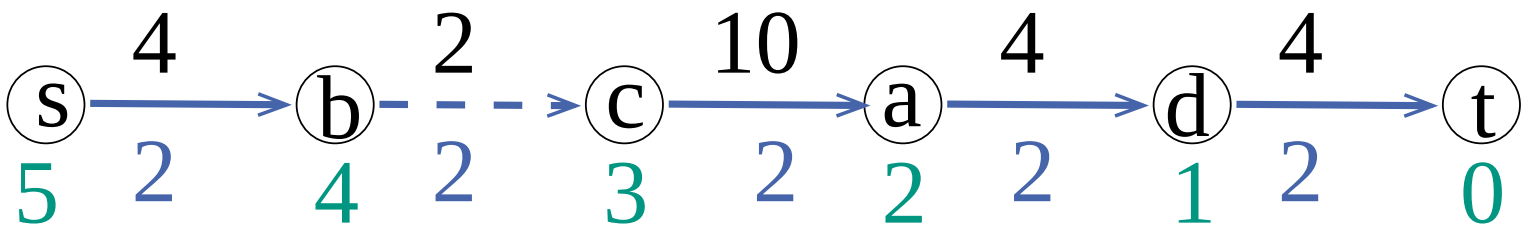
return f

Dinitz – Correctness

analogous to Ford-Fulkerson

Example


10 10
2
1
10
10
12 12
4
3
0
s
t
6
2 4
10
2
1
8 4
b
d
4 4


4
2
10
4
4
s
b
c
a
d
t
5 2
4
2
3
2
2 2
1 2
0

Computing Blocking Flows

Idee: wiederholte DFS nach augmentierenden Pfaden

Function blockingFlow($L_f = (V,E)$) : $E \to \mathbb{N}$

p=$\langle s \rangle$: Path; f_b=0 : Flow

loop // Round

v:= p.last()

if $v = t$ **then** // breakthrough

δ:= min $\{c(e) - f_b(e) : e \in p\}$

foreach $e \in p$ **do**

$f_b(e)$+=δ

if $f_b(e) = c(e)$ **then** remove e from E

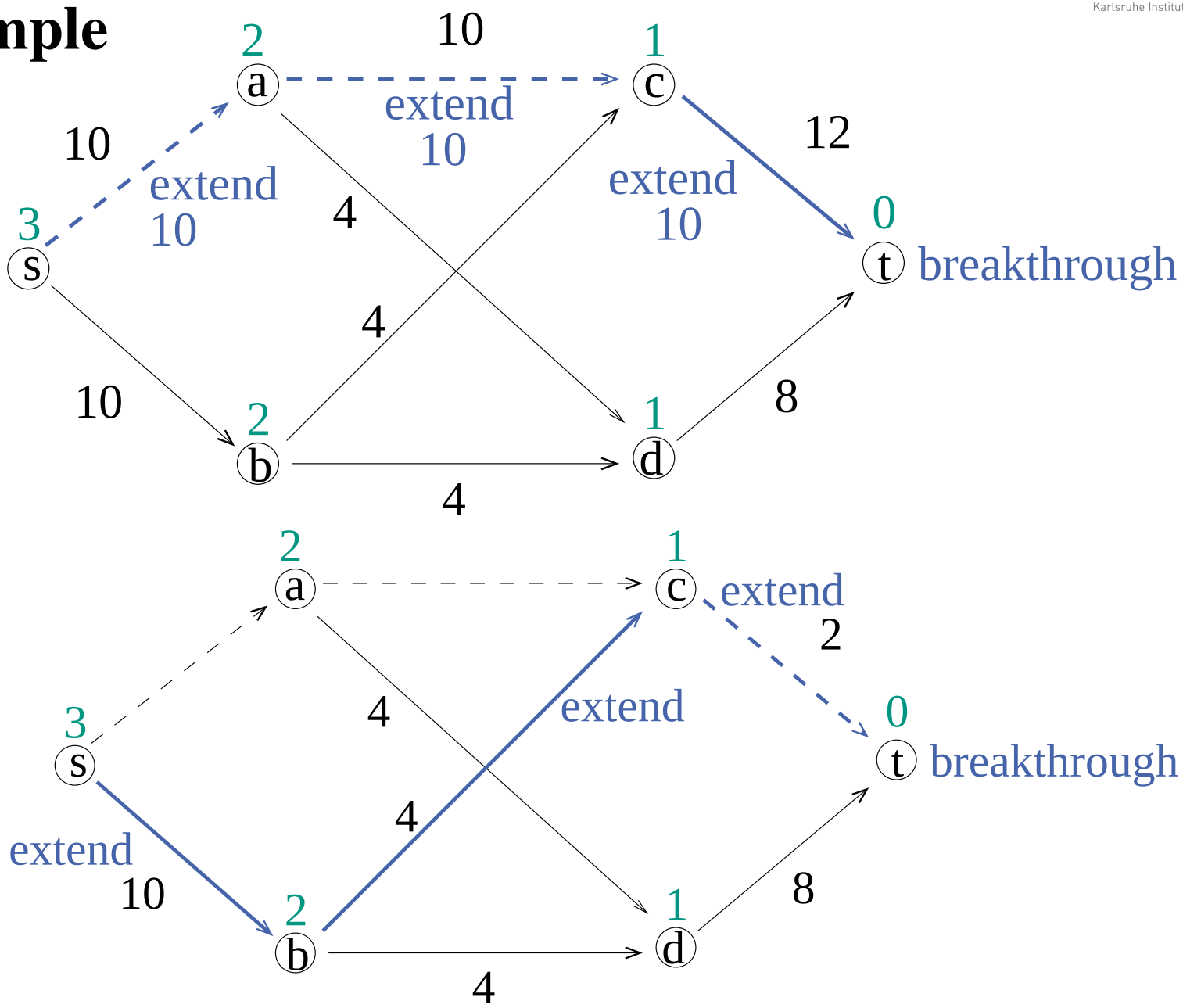
p:= $\langle s \rangle$

else if $\exists e = (v,w) \in E$ **then** p.pushBack(w) // extend

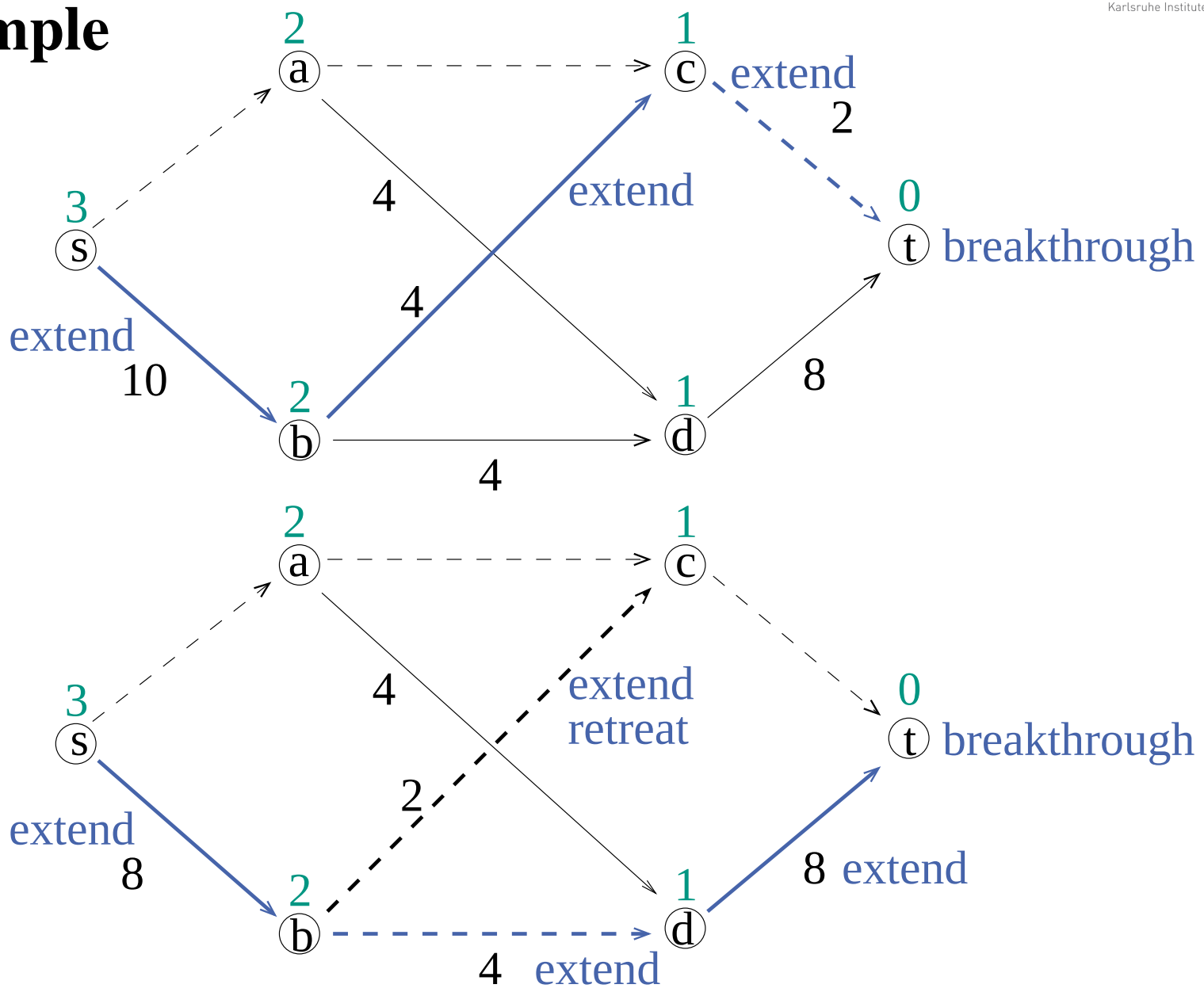
else if $v = s$ **then return** f_b // done

else delete the last edge from p in p and E // retreat

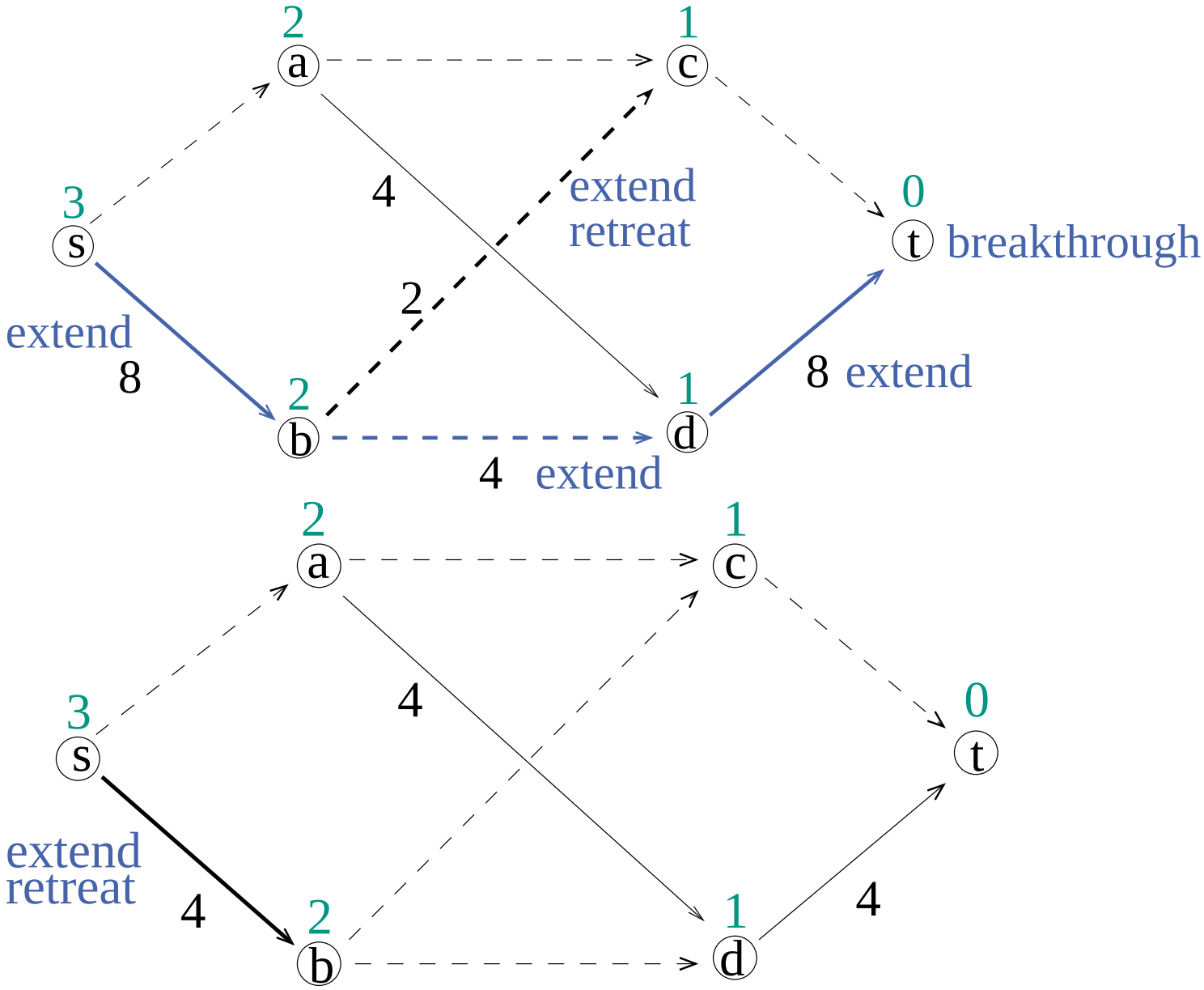
Example


2
a
10
1
c
extend
10
10
extend
10
12
extend
10
3
s
4
0
t
breakthrough
4
10
2
b
1
d
8
4
2
a
1
c
extend
2
3
s
4
extend
0
t
breakthrough
4
extend
10
2
b
1
d
8
4

Example


2
a
1
c
extend
2
3
s
4
extend
0
t
breakthrough
4
extend
10
2
b
1
d
8
4
2
a
1
c
3
s
4
extend
retreat
0
t
breakthrough
2
extend
8
2
b
1
d
8 extend
4 extend

Example


2
a
1
c
3
s
4
extend
retreat
0
t
breakthrough
2
extend
8
2
b
1
d
8 extend
4 extend
2
a
1
c
3
s
4
0
t
extend
retreat
4
2
b
1
d
4

Blocking Flows Analysis 1

- running time $\#_{extends} + \#_{retreats} + n \cdot \#_{breakthroughs}$
- $\#_{breakthroughs} \leq m$ – ≥ 1 edge is saturated
- $\#_{retreats} \leq m$ – one edge is removed
- $\#_{extends} \leq \#_{retreats} + n \cdot \#_{breakthroughs}$
  – a retreat cancels 1 extend, a breakthrough cancels $\leq n$ extends

time is $O(m + nm) = O(nm)$

Blocking Flows Analysis 2

Unit capacities:

breakthroughs saturates all edges on p, i.e., amortized constant cost per edge.

time $O(m+n)$

Blocking Flows Analysis 3

Dynamic trees: breakthrough (!), retreat, extend in time $O(\log n)$

time $O((m+n)\log n)$

"Theory alert": In practice, this seems to be slower

(few breakthroughs, many retreat, extend ops.)

Dinitz Analysis 1

Lemma 1. *$d(s)$ increases by at least one in each round.*

Beweis. not here □

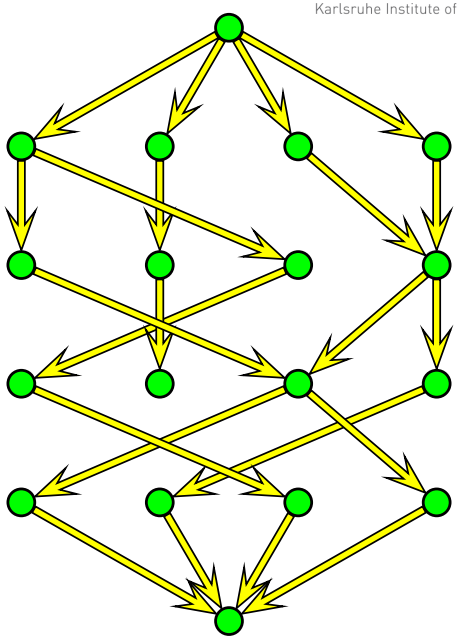
Dinitz Analysis 2

- $\leq n$ rounds
- time $\mathrm{O}(mn)$ each

time $\mathrm{O}(mn^2)$ (strongly polynomial)

time $\mathrm{O}(mn \log n)$ with dynamic trees

Dinitz Analysis 3 – Unit Capacities



Lemma 2. *At most $2\sqrt{m}$ BF computations:*

Beweis. Consider iteration $k = \sqrt{m}$.
Cut in layergraph induces cut in residual graph of capacity at most $\sqrt{m}$.
At most $\sqrt{m}$ additional phases.

□

Total time: $O((m+n)\sqrt{m})$

more detailed analysis: $O\left(m \min\left\{m^{1/2}, n^{2/3}\right\}\right)$

Dinitz Analysis 4 – Unit Networks

Unit capacity $+ \forall v \in V : \min\{\text{indegree}(v), \text{outdegree}(v)\} = 1$:

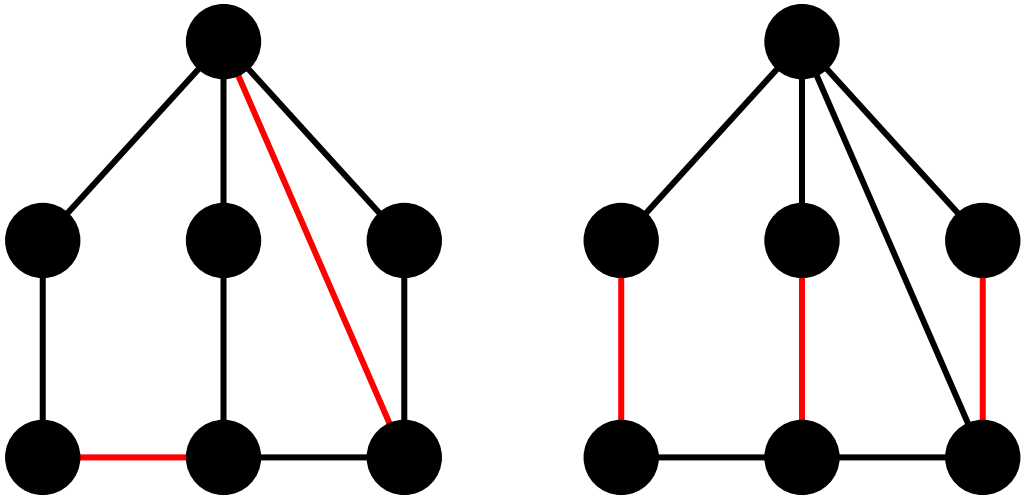
time: $O((m+n)\sqrt{n})$

Matching

$M \subseteq E$ is a matching in the undirected graph $G = (V, E)$ iff (V, M) has maximum degree ≤ 1.

M is maximal if $\nexists e \in E \setminus M : M \cup \{e\}$ is a matching.

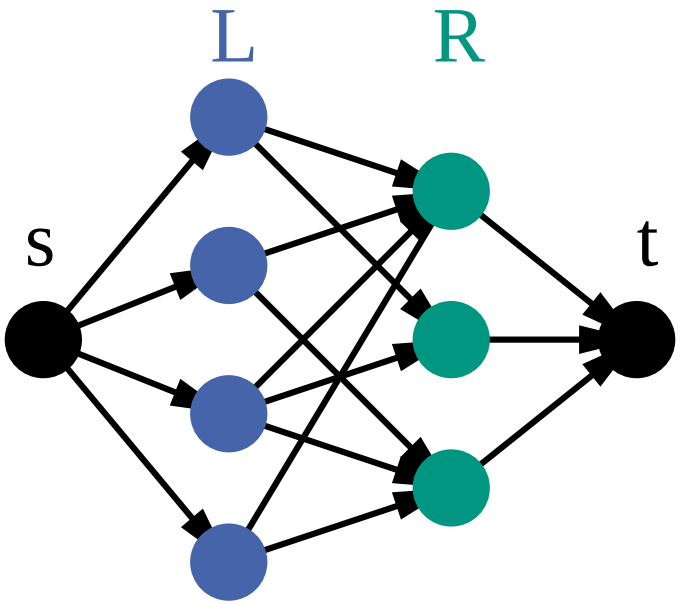
M has maximum cardinality if $\nexists$matching $M' : |M'| > |M|$



Maximum Cardinality Bipartite Matching

in $(L \cup R, E)$. Model as a unit network maximum flow problem

$$(\{s\} \cup L \cup R \cup \{t\}, \{(s,u) : u \in L\} \cup E \cup \{(v,t) : v \in R\})$$

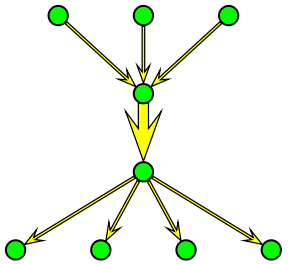


Dinitz algorithm yields $O((n+m)\sqrt{n})$ algorithm

Similar Performance for Weighted Graphs?

time: $O\left(m \min\left\{m^{1/2}, n^{2/3}\right\} \log C\right)$ [Goldberg Rao 97]

Problem: Fat edges between layers ruin the argument



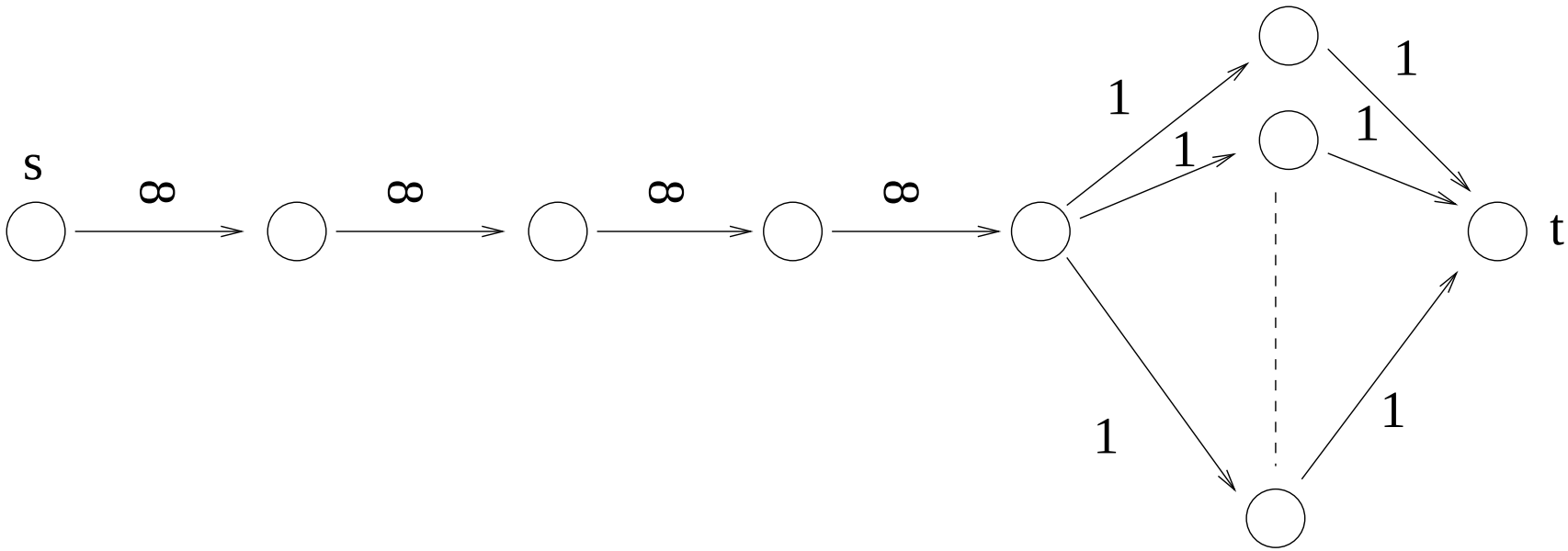
Idea: scale a parameter Δ from small to large

contract SCCs of fat edges (capacity $> \Delta$)

Experiments [Hagerup, Sanders Träff 98]:

Sometimes best algorithm usually slower than preflow push

Disadvantage of augmenting paths algorithms



Preflow-Push Algorithms

Preflow f: a flow where the flow conservation constraint is relaxed to

$$\text{excess}(v) := \overbrace{\sum_{(u,v)\in E} f_{(u,v)}}^{\text{inflow}} - \overbrace{\sum_{(v,w)\in E} f_{(v,w)}}^{\text{outflow}} \geq 0 \ .$$

$v \in V \setminus \{s,t\}$ is active iff $\text{excess}(v) > 0$

Procedure push$(e = (v,w), \delta)$

assert $\delta > 0 \quad \wedge \quad \text{excess}(v) \geq \delta$

assert residual capacity of $e \geq \delta$

$\text{excess}(v) -= \delta$

$\text{excess}(w) += \delta$

if e is reverse edge **then** $f(\text{reverse}(e)) -= \delta$

else $f(e) += \delta$

Level Function

Idea: make progress by pushing towards t

Maintain

an approximation $d(v)$ of the BFS distance from v to t in G_f.

invariant $d(t) = 0$

invariant $d(s) = n$

invariant $\forall (v,w) \in E_f : d(v) \leq d(w) + 1$ // no steep edges

Edge directions of $e = (v,w)$

steep: $d(w) < d(v) - 1$

downward: $d(w) < d(v)$

horizontal: $d(w) = d(v)$

upward: $d(w) > d(v)$

Procedure genericPreflowPush(G=(V,E), f)

forall $e = (s,v) \in E$ **do** push$(e, c(e))$ // saturate

$d(s) := n$

$d(v) := 0$ for all other nodes

while $\exists v \in V \setminus \{s,t\}$: excess$(v) > 0$ **do** // active node

if $\exists e = (v,w) \in E_f : d(w) < d(v)$ **then** // eligible edge

choose some $\delta \leq \min\left\{\text{excess}(v), c_e^f\right\}$

push(e, δ) // no new steep edges

else $d(v)$++ // relabel. No new steep edges

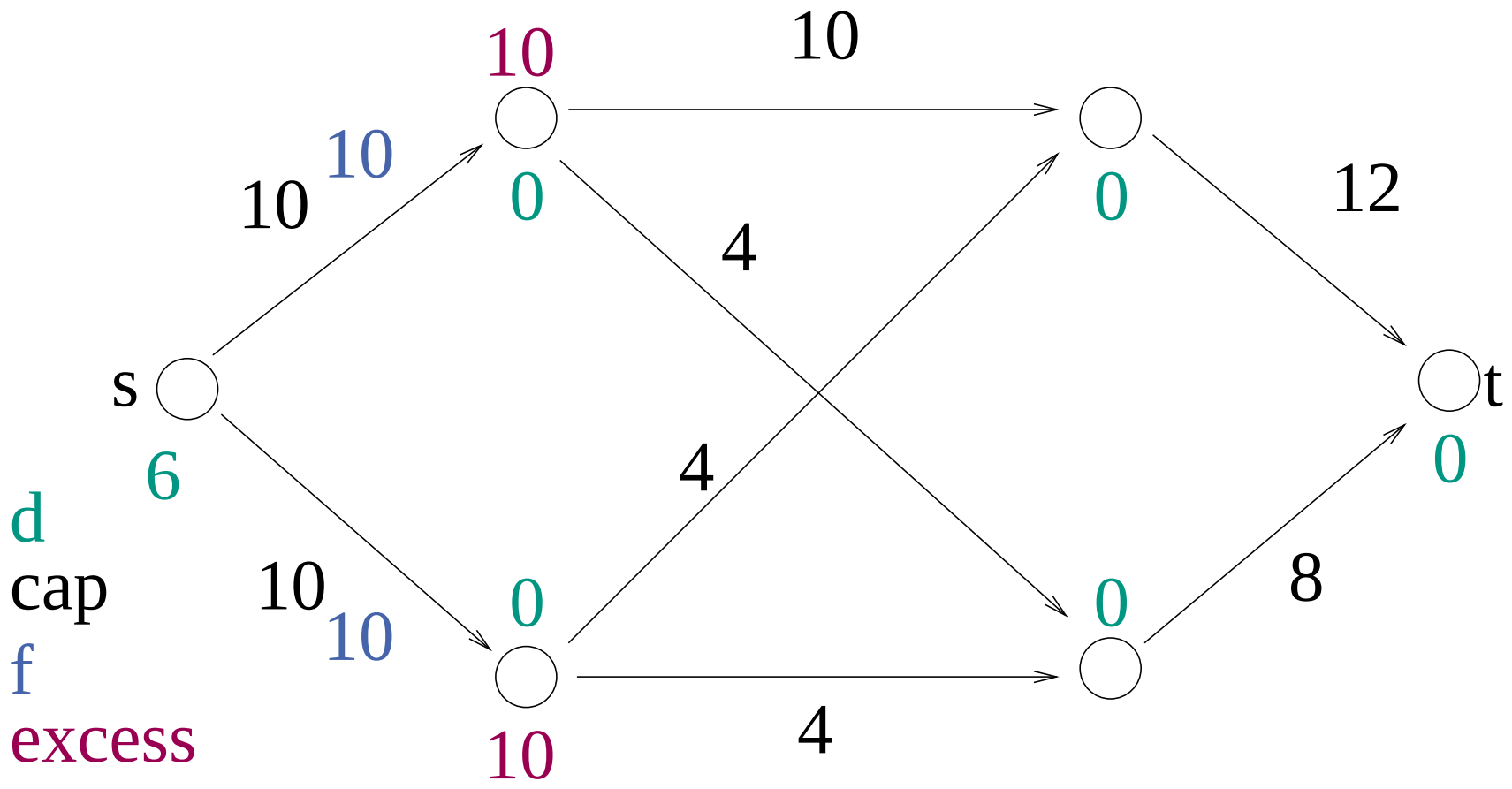
Obvious choice for δ : $\delta = \min\left\{\text{excess}(v), c_e^f\right\}$

Saturating push: $\delta = c_e^f$

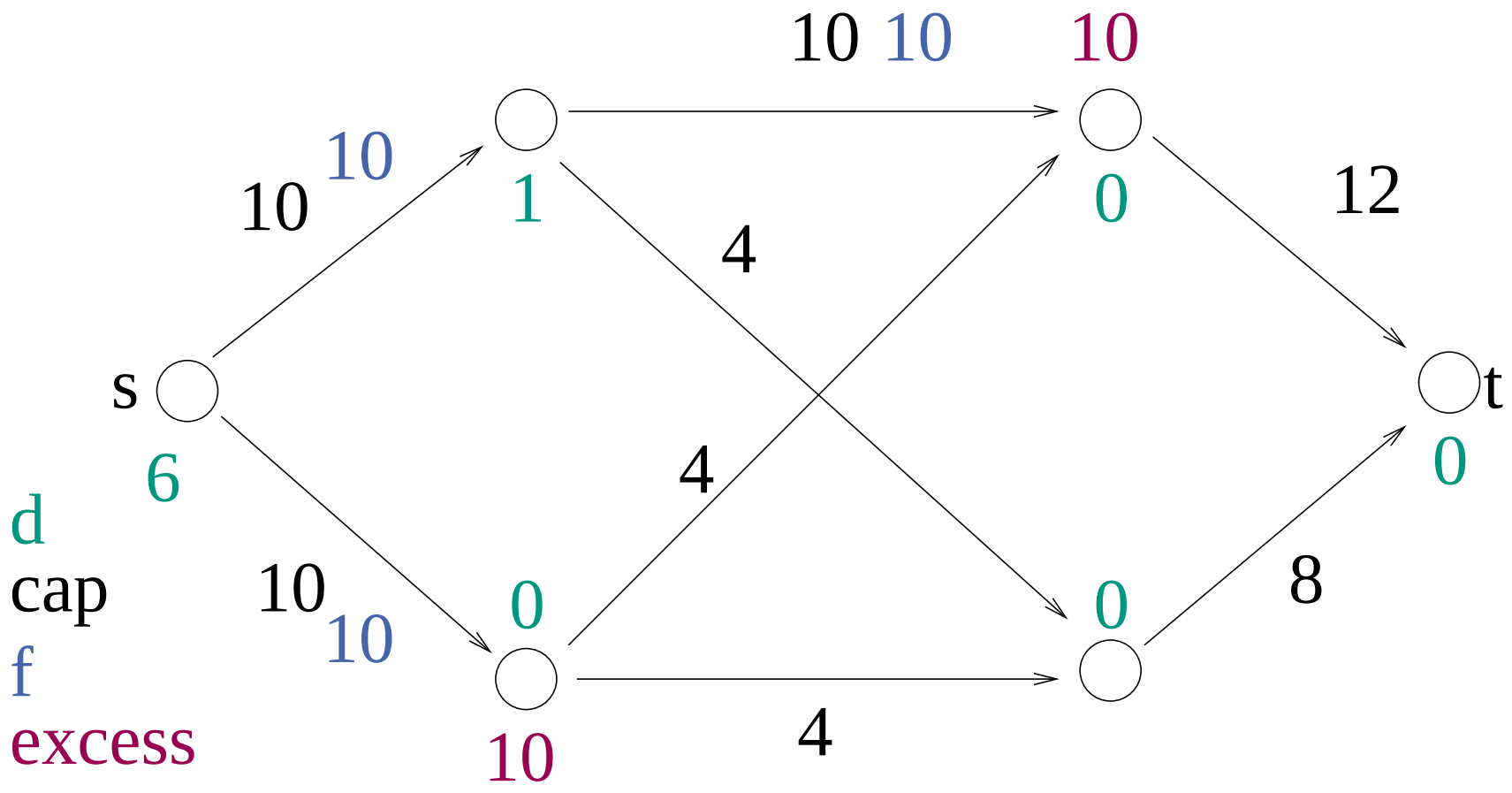
nonsaturating push: $\delta < c_e^f$

To be filled in: How to select active nodes and eligible edges?

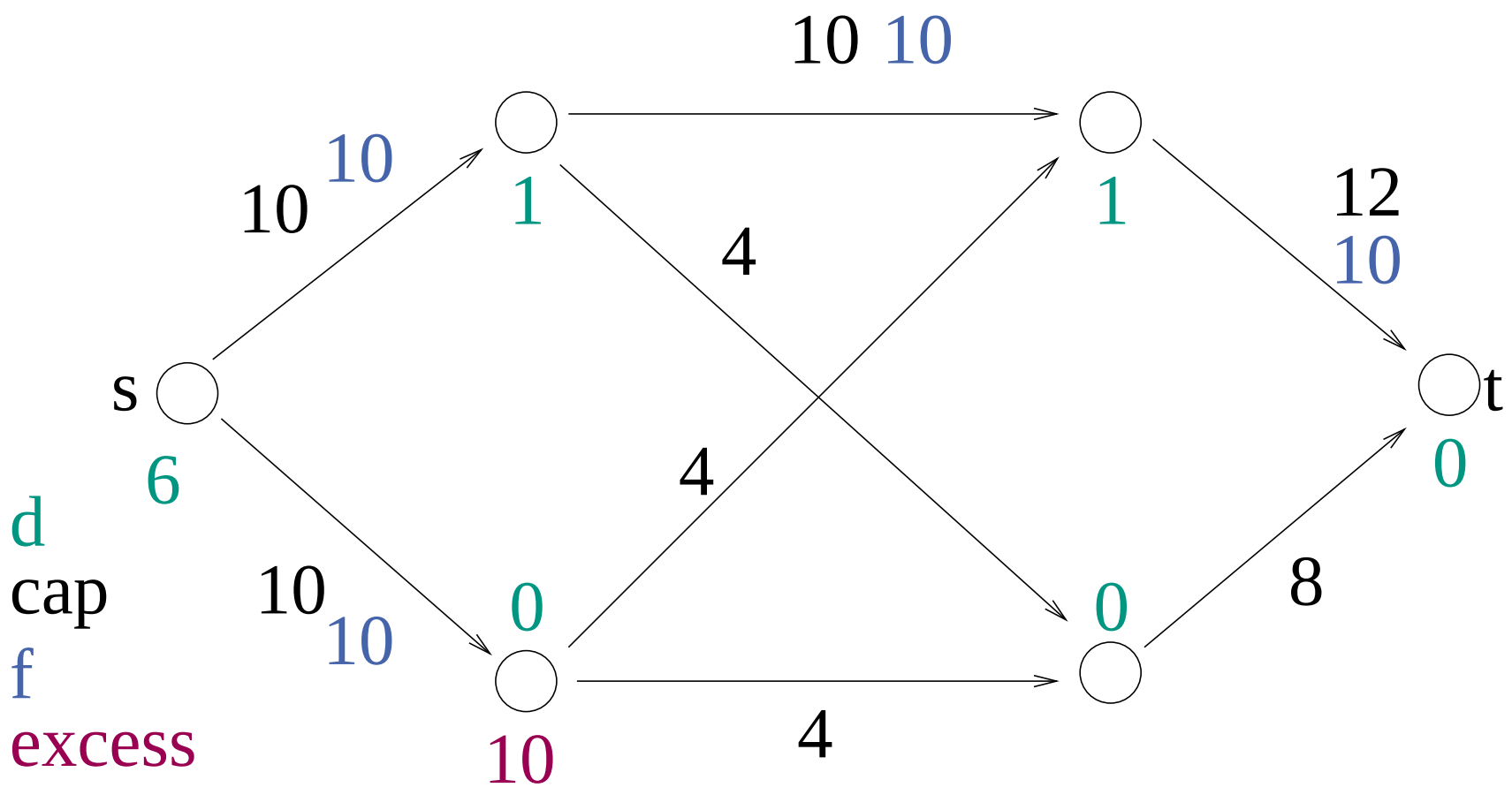
Example



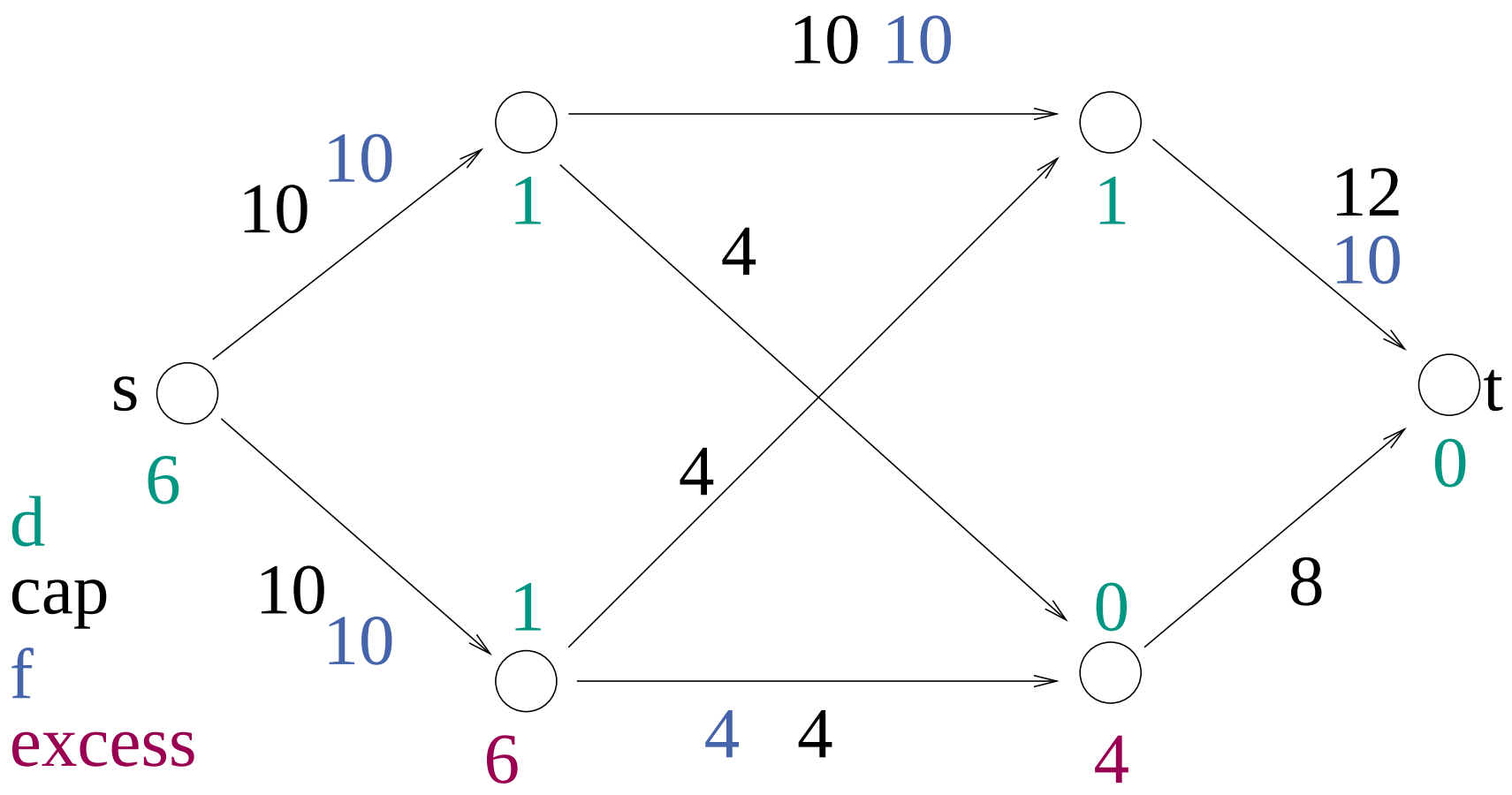
Example



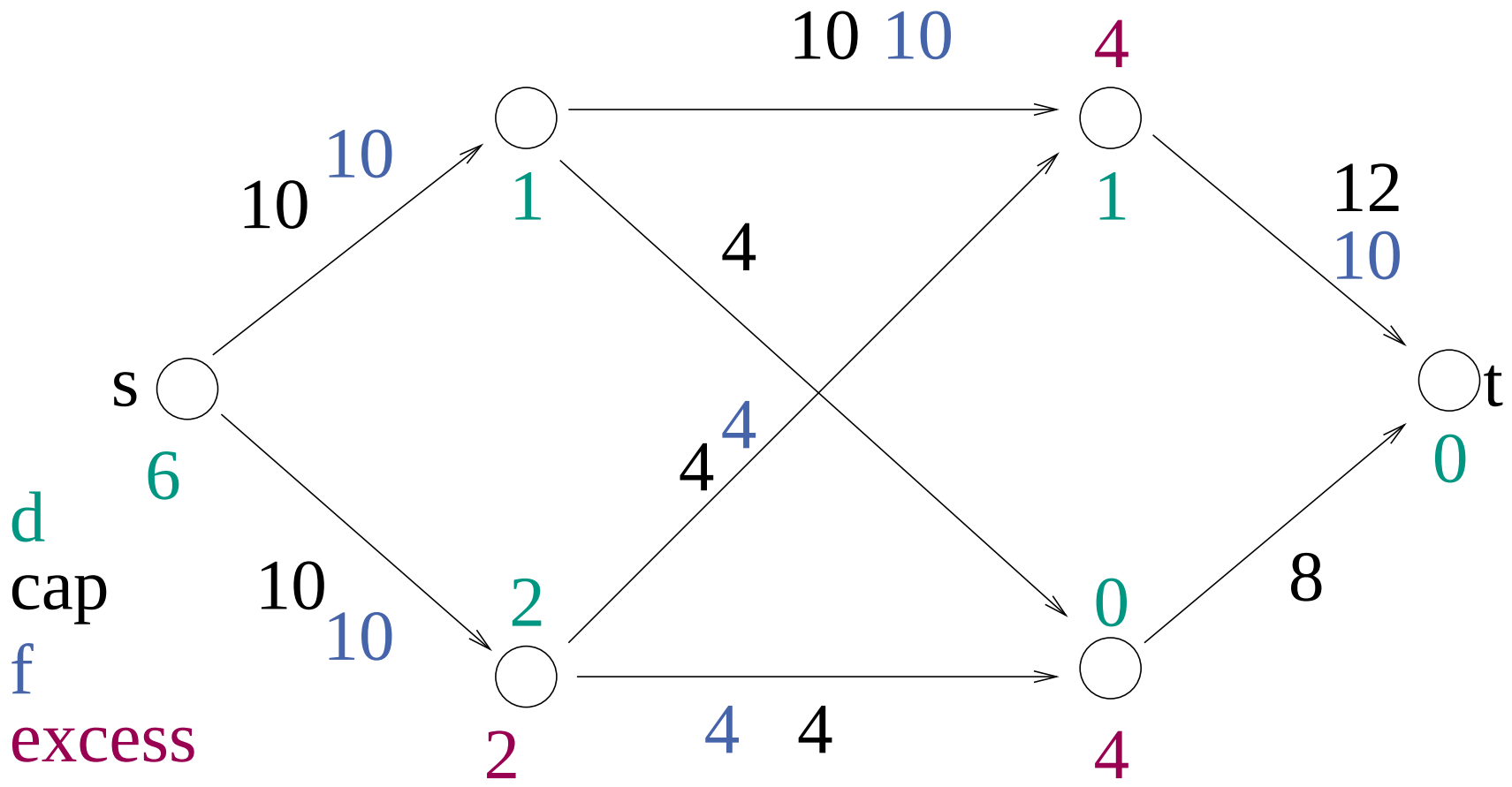
Example



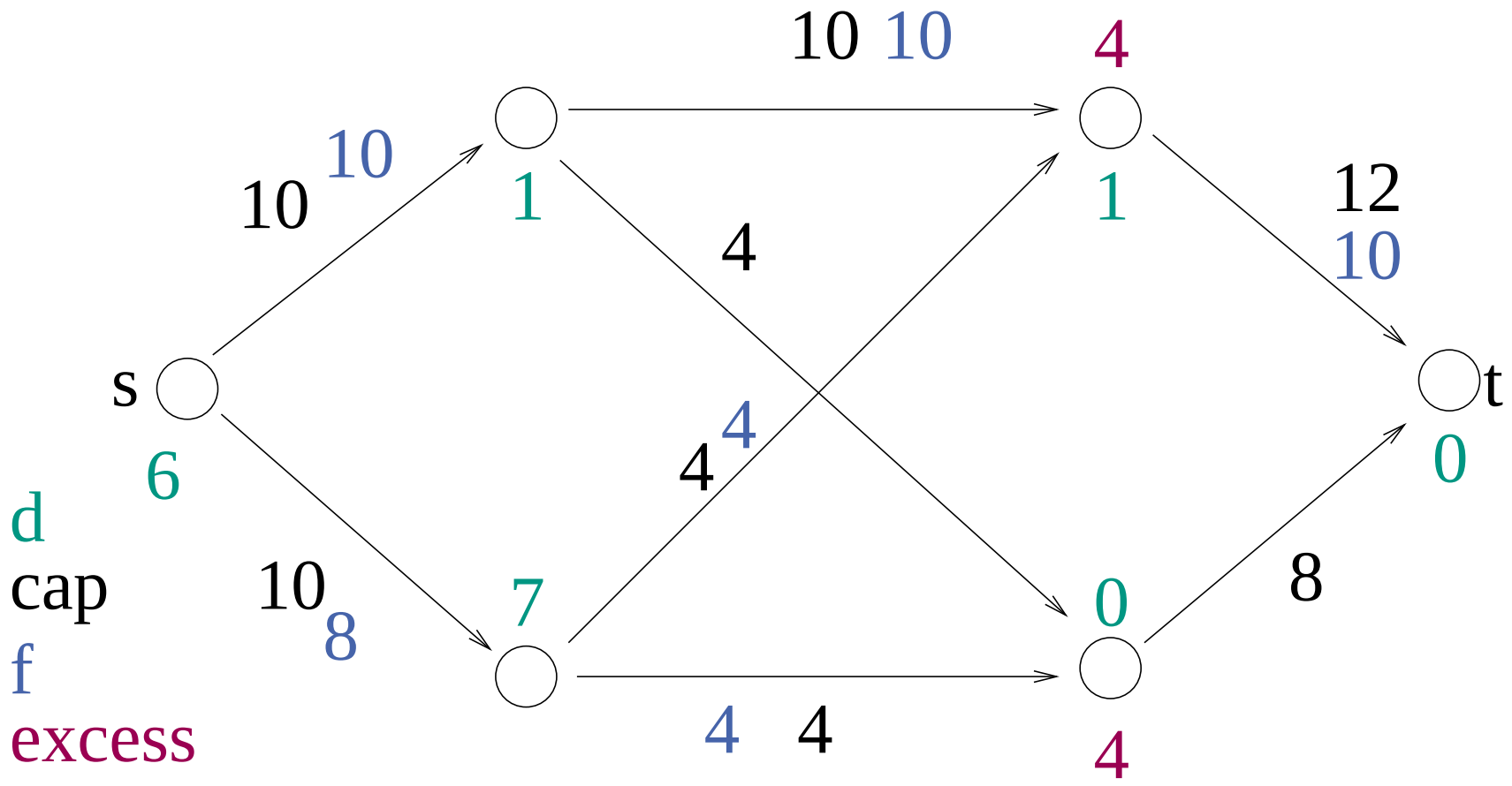
Example



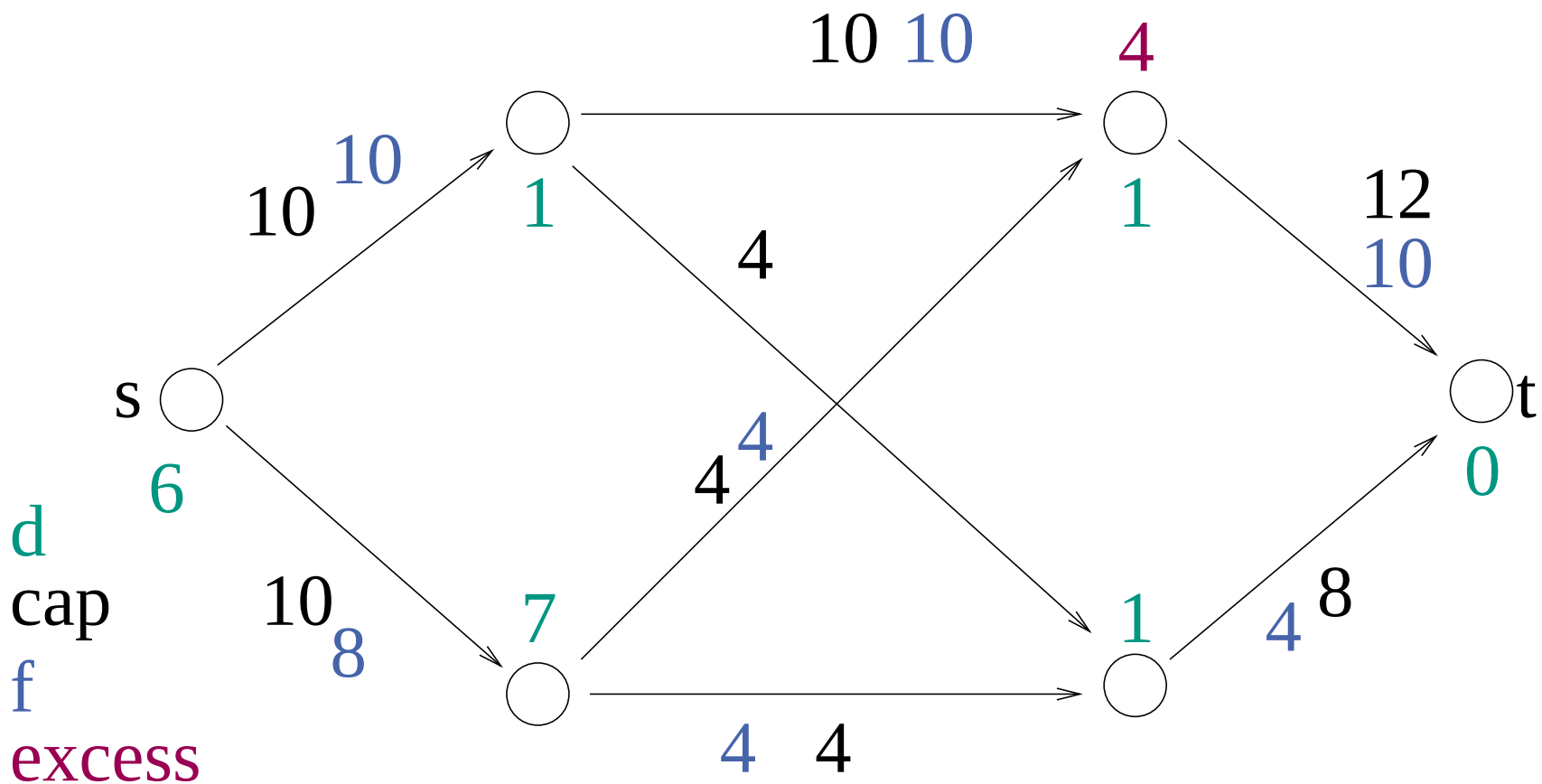
Example



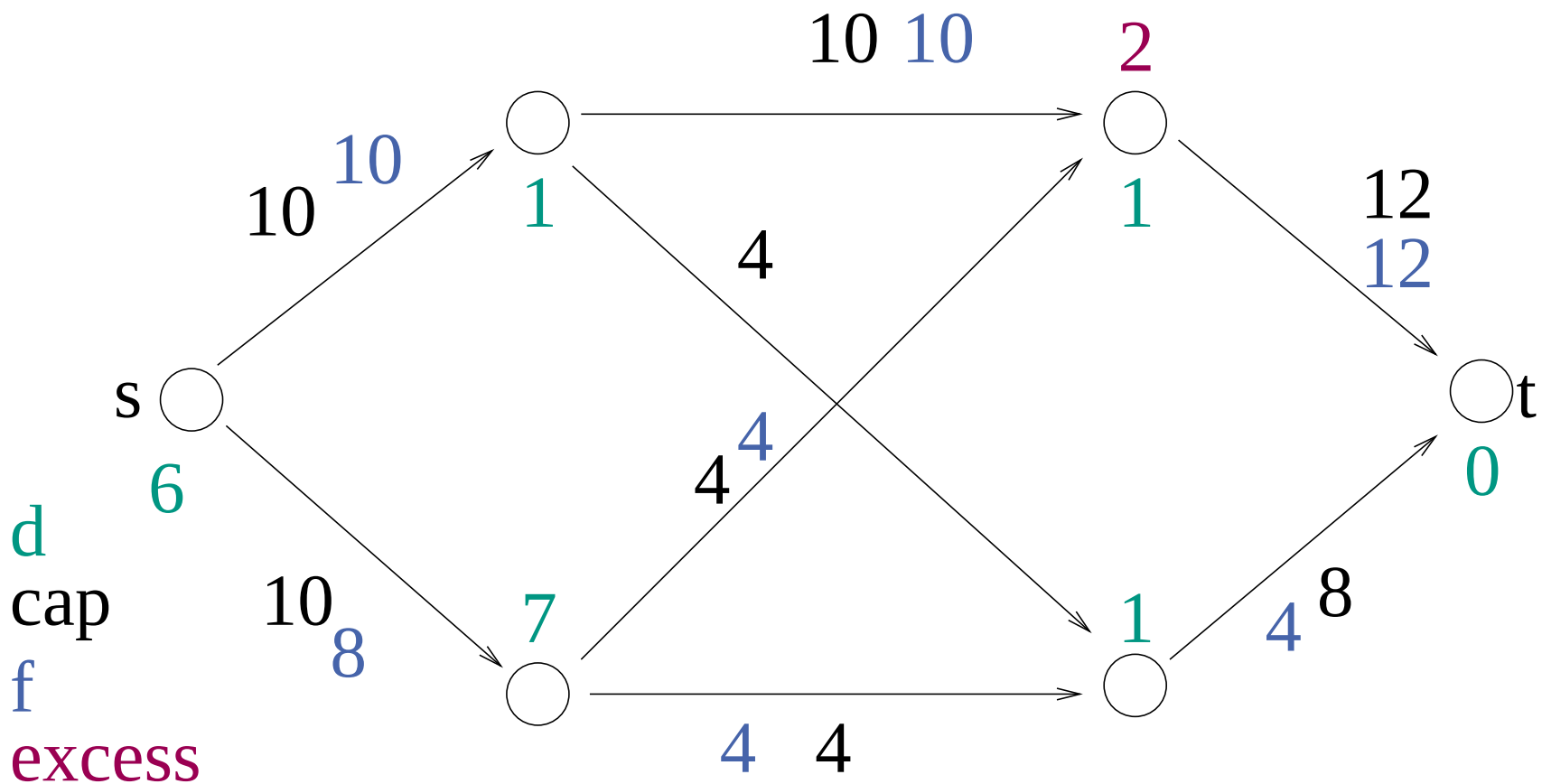
Example



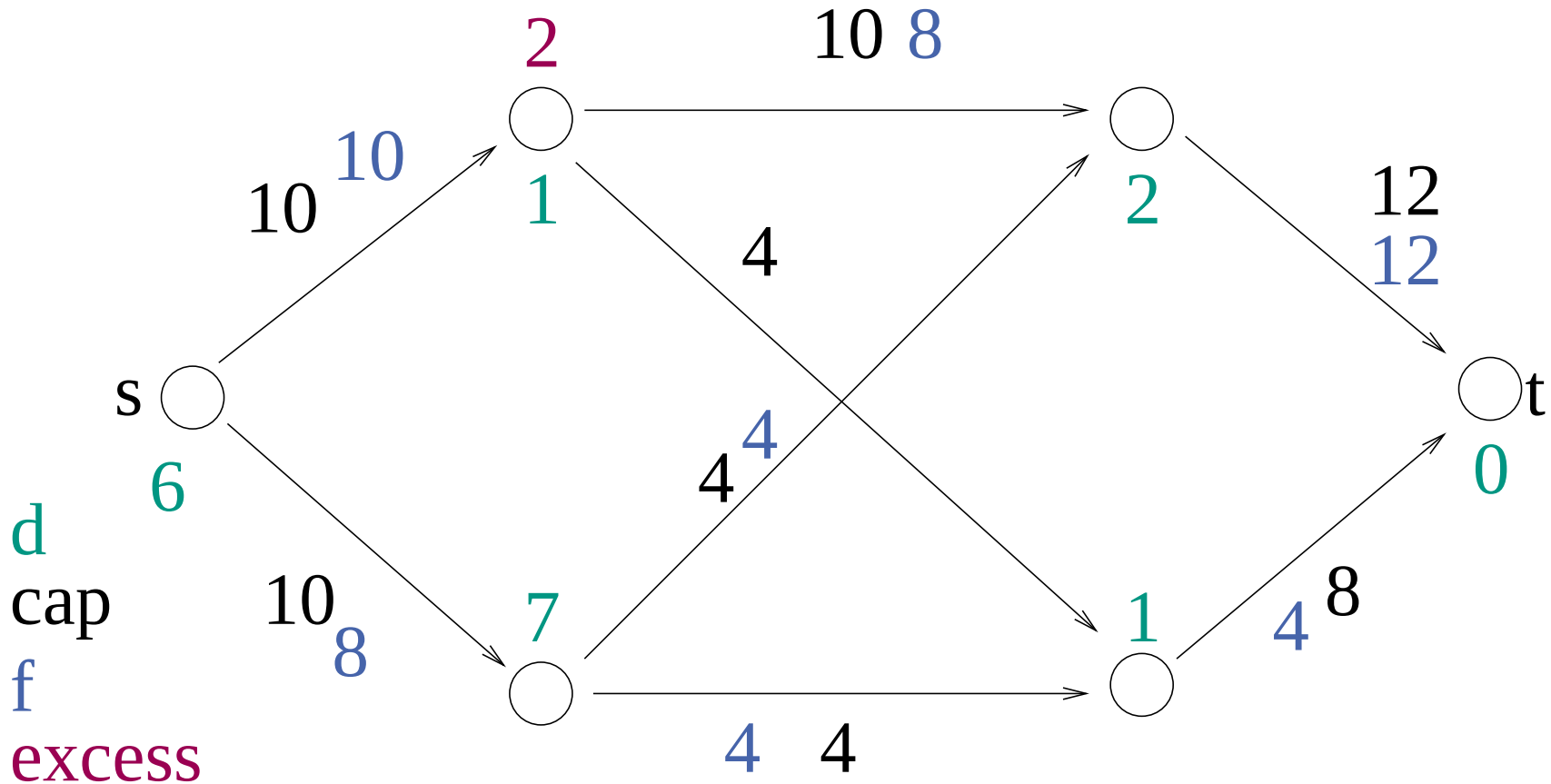
Example



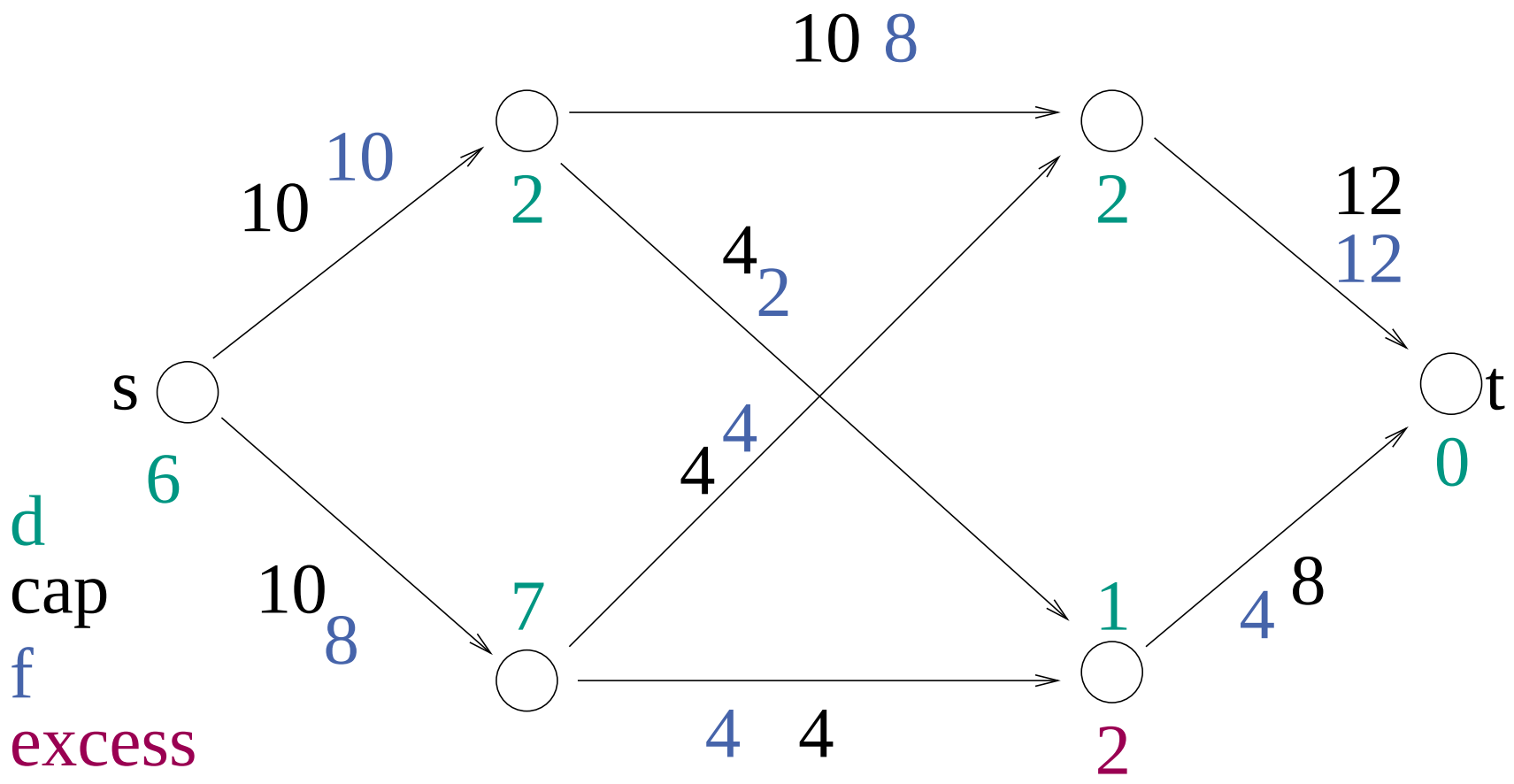
Example



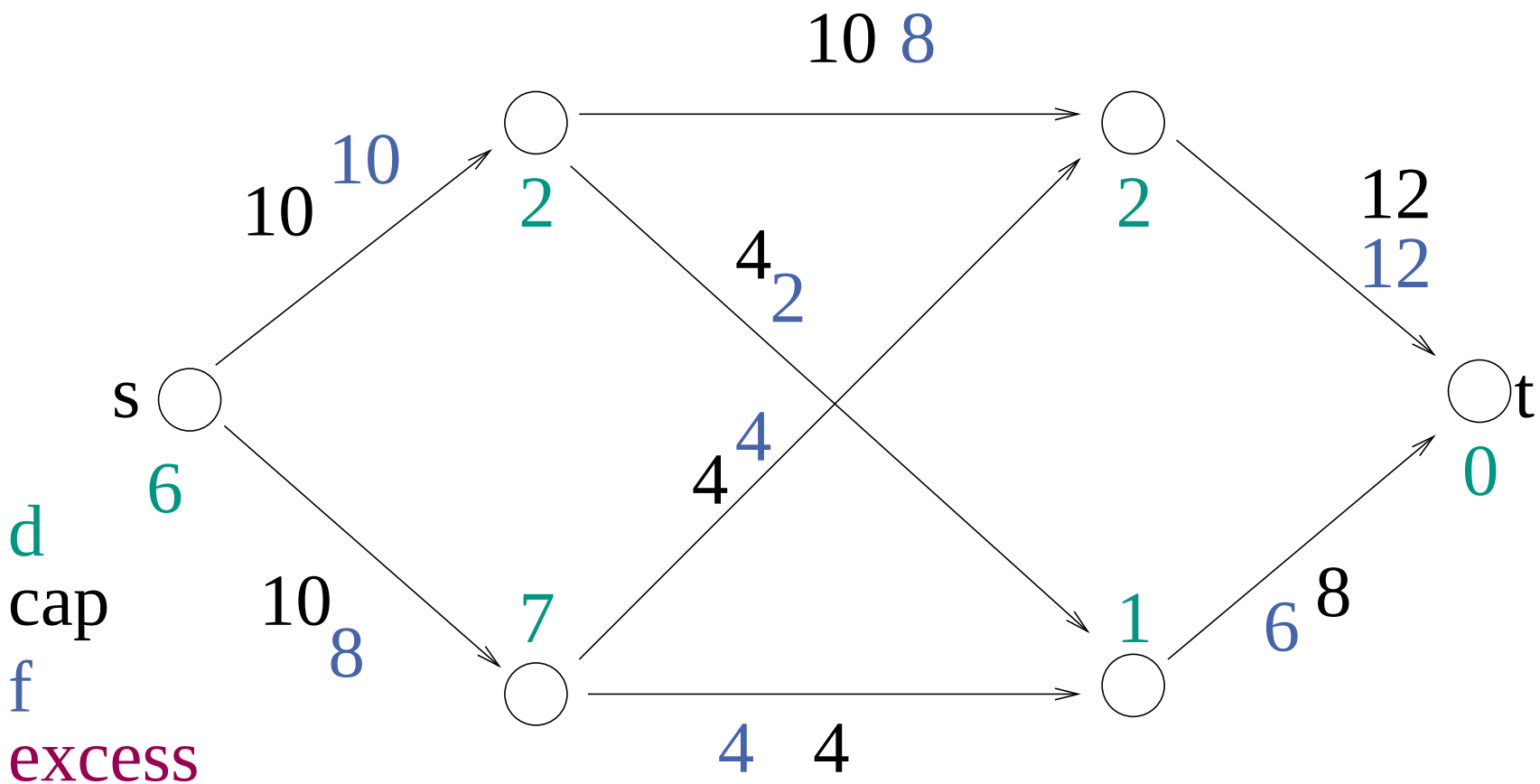
Example



Example



Example



12 pushes in total

Partial Correctness

Lemma 3. *When* genericPreflowPush *terminates*
f *is a* *maximal flow*.

Beweis.
f is a flow since $\forall v \in V \setminus \{s,t\} : \mathsf{excess}(v) = 0$.

To show that f is maximal, it suffices to show that
$\nexists$ path $p = \langle s, \ldots, t \rangle \in G_f$ (Max-Flow Min-Cut Theorem):
Since $d(s) = n$, $d(t) = 0$, p would have to contain steep edges.
That would be a contradiction. □

Lemma 4. *For any cut* (S,T),

$$\sum_{u\in S} excess(u) = \sum_{e\in E\cap(T\times S)} f(e) - \sum_{e\in E\cap(S\times T)} f(e),$$

Proof:

$$\sum_{u\in S} excess(u) = \sum_{u\in S}\left(\sum_{(v,u)\in E} f((v,u)) - \sum_{(u,v)\in E} f((u,v))\right)$$

Contributions of edge e to sum:

S to T: $-f(e)$

T to S: $f(e)$

within S: $f(e)-f(e)=0$

within T: 0

■

Lemma 5.

$\forall$ *active nodes* v : excess$(v) > 0 \Rightarrow \exists$ *path* $\langle v, \ldots, s \rangle \in G_f$

Intuition: what got there can always go back.

Beweis. $S := \{u \in V : \exists \text{ path } \langle v, \ldots u \rangle \in G_f\}$, $T := V \setminus S$. Then

$$\sum_{u \in S} excess(u) = \sum_{e \in E \cap (T \times S)} f(e) - \sum_{e \in E \cap (S \times T)} f(e),$$

$\forall (u,w) \in E_f : u \in S \Rightarrow w \in S$ by Def. of G_f, S

$\Rightarrow \forall e = (u,w) \in E \cap (T \times S) : f(e) = 0$ Otherwise $(w,u) \in E_f$

Hence, $\sum_{u \in S} excess(u) \leq 0$

Only the negative excess of s can outweigh excess$(v) > 0$.

Hence $s \in S$. $\square$

Lemma 6.

$\forall v \in V : d(v) < 2n$

Beweis.

Suppose v is lifted to $d(v) = 2n$.

By the Lemma 2, there is a (simple) path p to s in G_f.

p has at most $n-1$ nodes

$d(s) = n$.

Hence $d(v) < 2n$. Contradiction (no steep edges). □

Lemma 7. *# Relabel operations* $\leq 2n^2$

Beweis. $d(v) \leq 2n$, i.e., v is relabeled at most $2n$ times.
Hence, at most $|V| \cdot 2n = 2n^2$ relabel operations. □

Lemma 8. *# saturating pushes* $\leq nm$

Beweis.

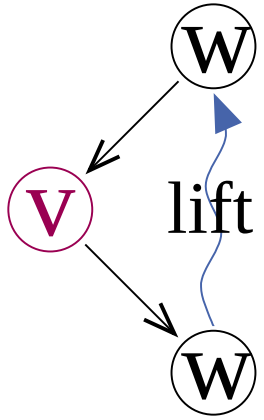
We show that there are at most n sat. pushes over any edge $e = (v, w)$.

A saturating push(e, δ) removes e from E_f.

Only a push on (w, v) can reinsert e into E_f.

For this to happen, w must be lifted at least two levels.

Hence, at most $2n/2 = n$ saturating pushes over (v, w)



□

Lemma 9. *# nonsaturating pushes* $= O(n^2 m)$
if $\delta = \min\left\{\text{excess}(v), c_e^f\right\}$
for arbitrary node and edge selection rules.
(arbitrary-preflow-push)

Beweis. $\Phi := \sum_{\{v : v \text{ is active}\}} d(v).$ (Potential)

$\Phi = 0$ initially and at the end (no active nodes left!)

Operation	$\Delta(\Phi)$	How many times?	Total effect
relabel	1	$\leq 2n^2$	$\leq 2n^2$
saturating push	$\leq 2n$	$\leq nm$	$\leq 2n^2 m$
nonsaturating push	≤ -1		

$\Phi \geq 0$ always. □

Searching for Eligible Edges

Every node v maintains a currentEdge pointer to its sequence of outgoing edges in G_f.

invariant no edge $e = (v, w)$ to the left of currentEdge is eligible

Reset currentEdge at a relabel $(\leq 2n\times)$

Invariant cannot be violated by a push over a reverse edge $e' = (w, v)$

since this only happens when e' is downward,

i.e., e is upward and hence not eligible.

Lemma 10.

Total cost for searching $\leq \sum_{v \in V} 2n \cdot \text{degree}(v) = 4nm = \mathrm{O}(nm)$

Satz 11. *Arbitrary Preflow Push finds a maximum flow in time* $O(n^2 m)$.

Beweis.

Lemma 3: partial correctness

Initialization in time $O(n+m)$.

Maintain set (e.g., stack, FIFO) of active nodes.

Use reverse edge pointers to implement push.

Lemma 7: $2n^2$ relabel operations

Lemma 8: nm saturating pushes

Lemma 9: $O(n^2 m)$ nonsaturating pushes

Lemma 10: $O(nm)$ search time for eligible edges

Total time $O(n^2 m)$ □

FIFO Preflow push

Examine a node: Saturating pushes until nonsaturating push or relabel.

Examine all nodes in phases (or use FIFO queue).

Theorem: time $O(n^3)$

Proof: not here

Highest Level Preflow Push

Always select active nodes that maximize $d(v)$

Use bucket priority queue (insert, increaseKey, deleteMax)

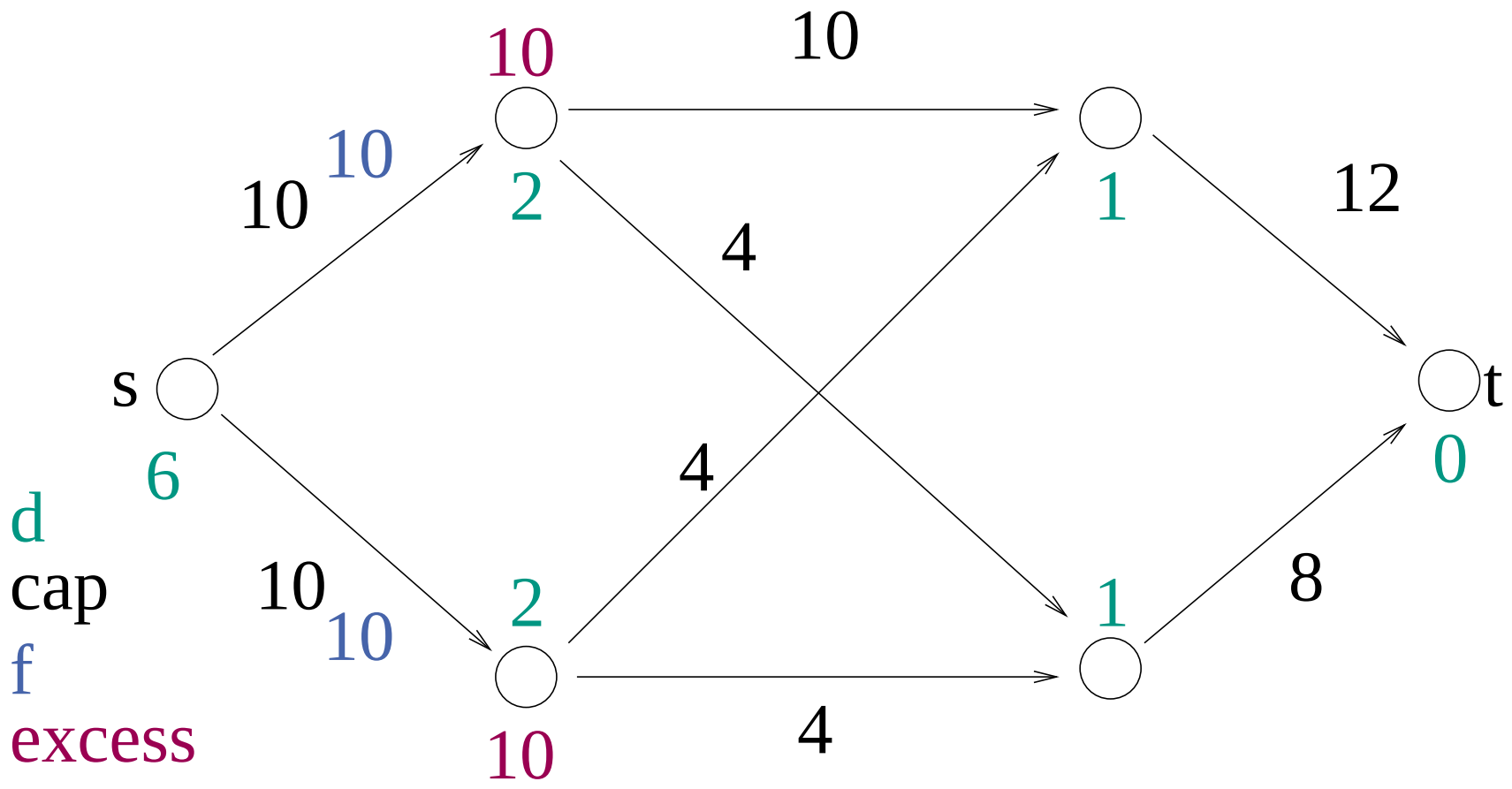
not monotone (!) but relabels "pay" for scan operations

Lemma 12. *At most $n^2\sqrt{m}$ nonsaturating pushes.*

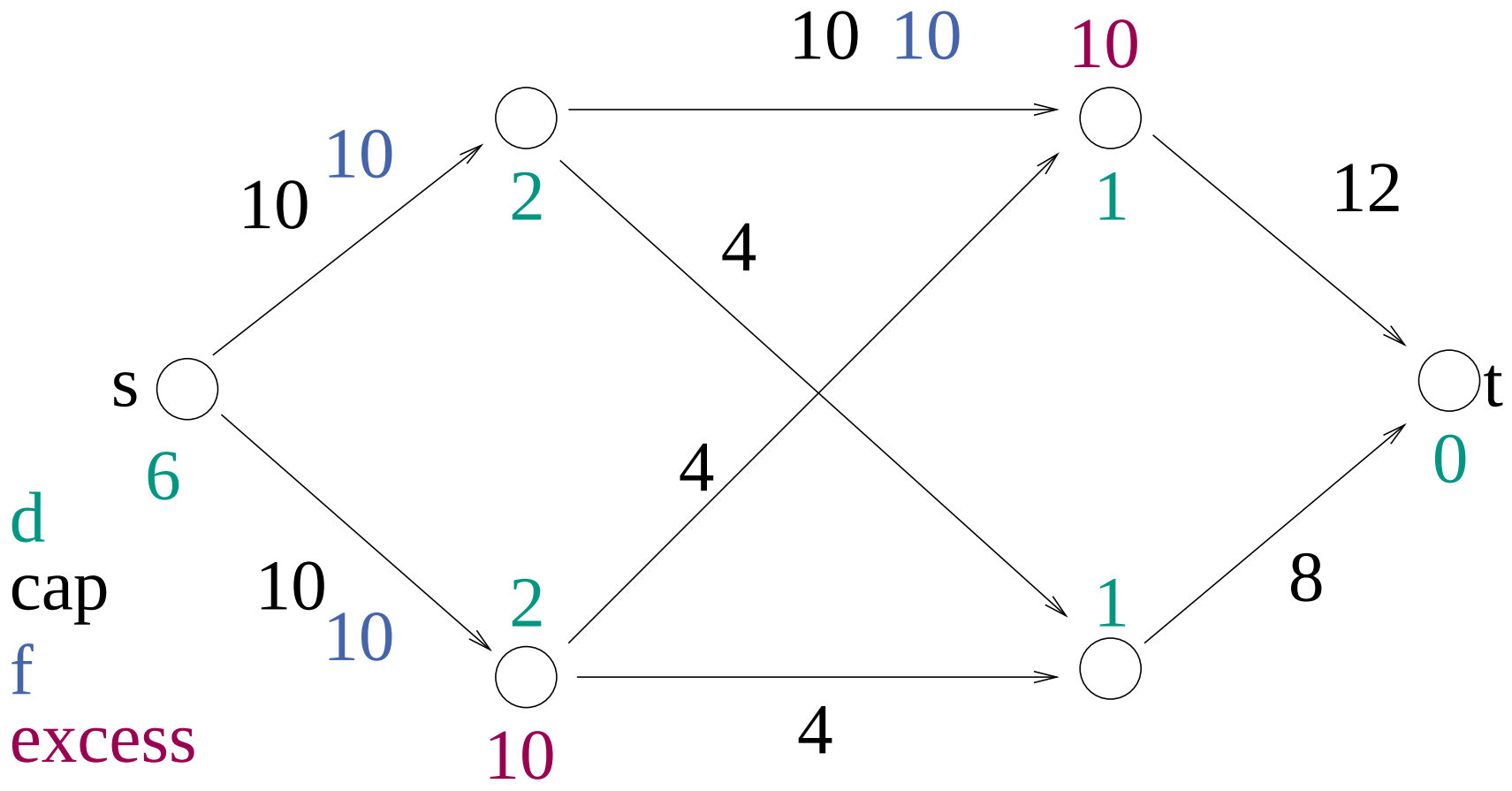
Beweis. later □

Satz 13. *Highest Level Preflow Push finds a maximum flow in time $O\left(n^2\sqrt{m}\right)$.*

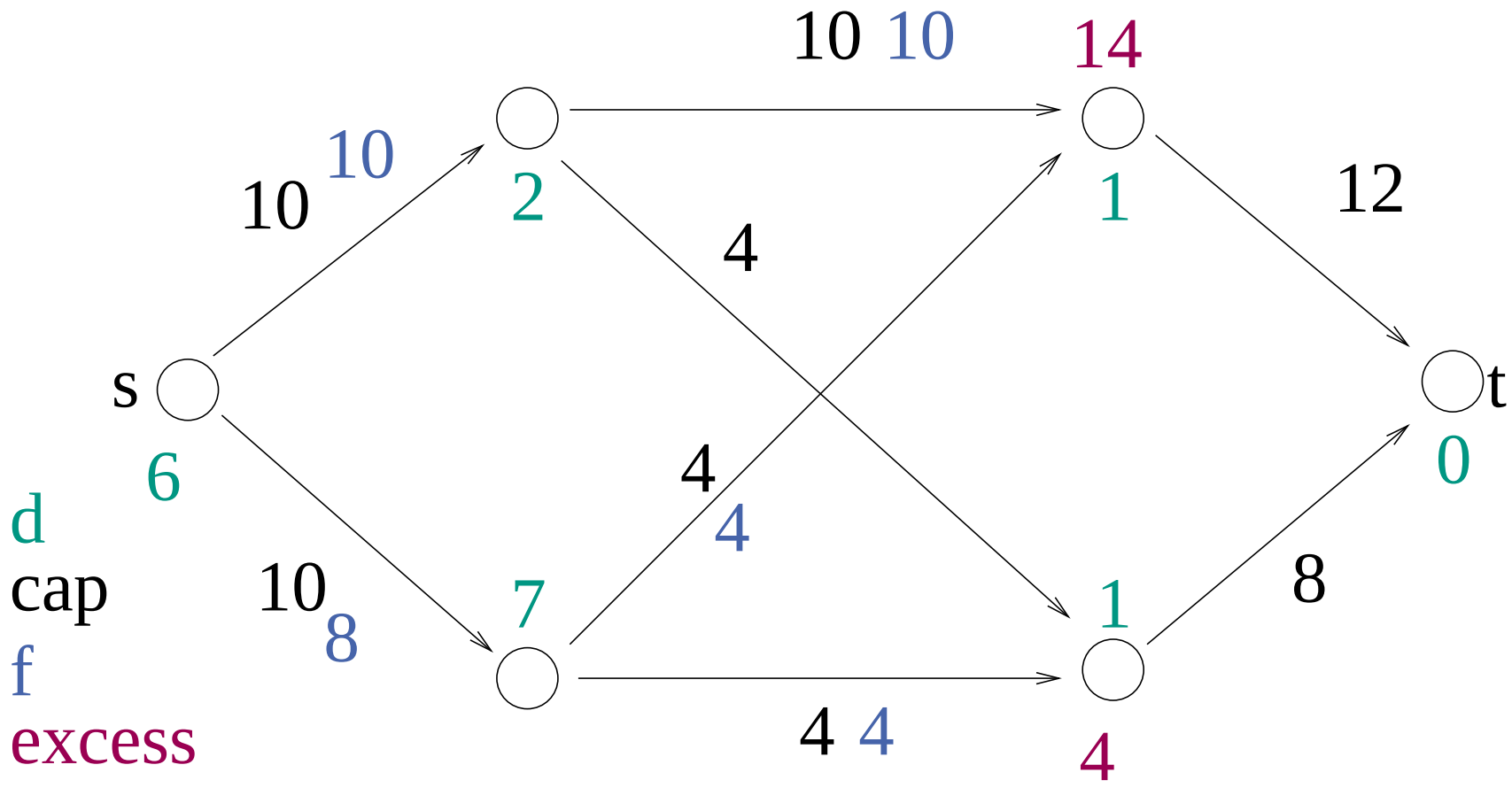
Example



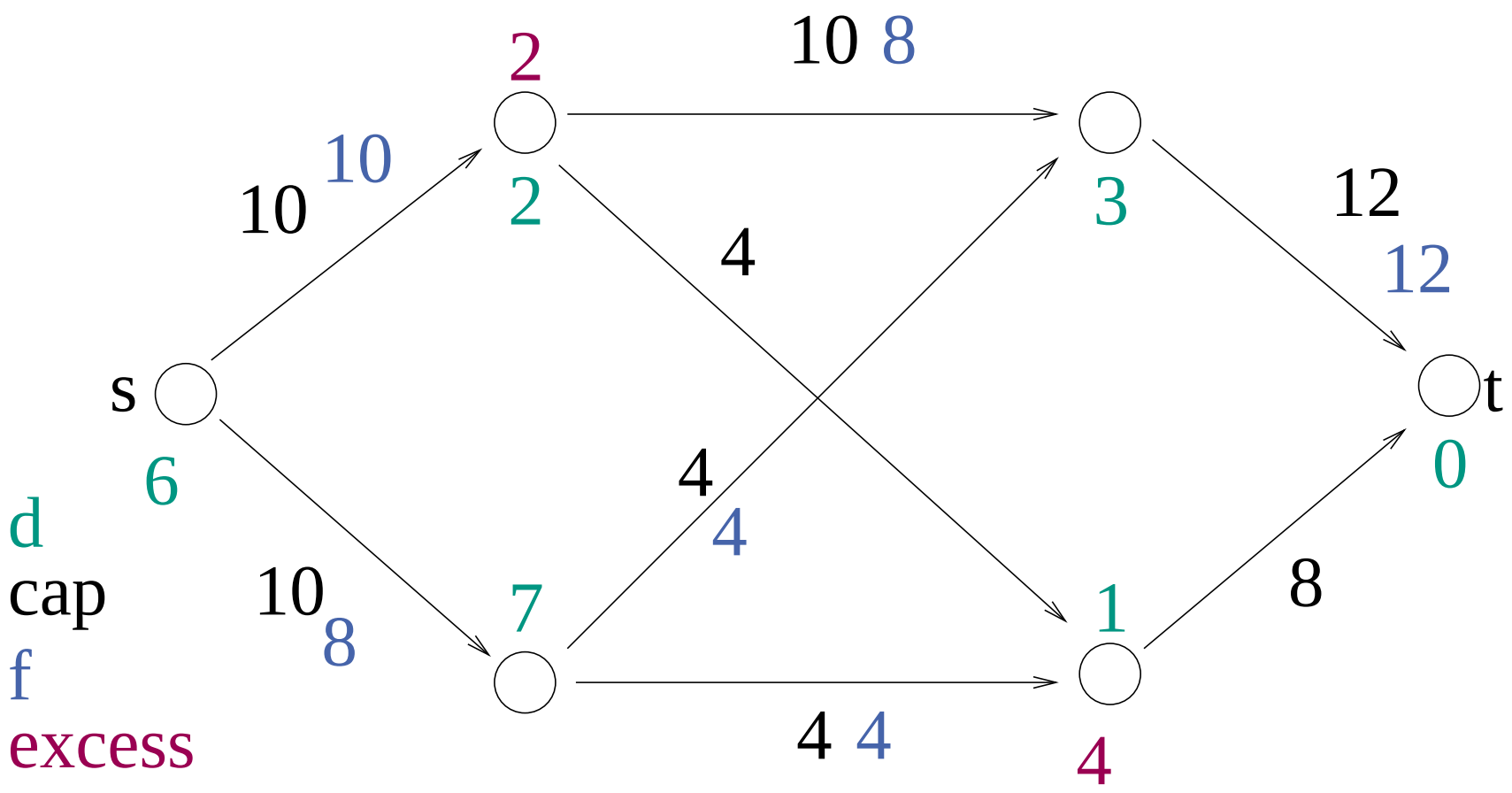
Example



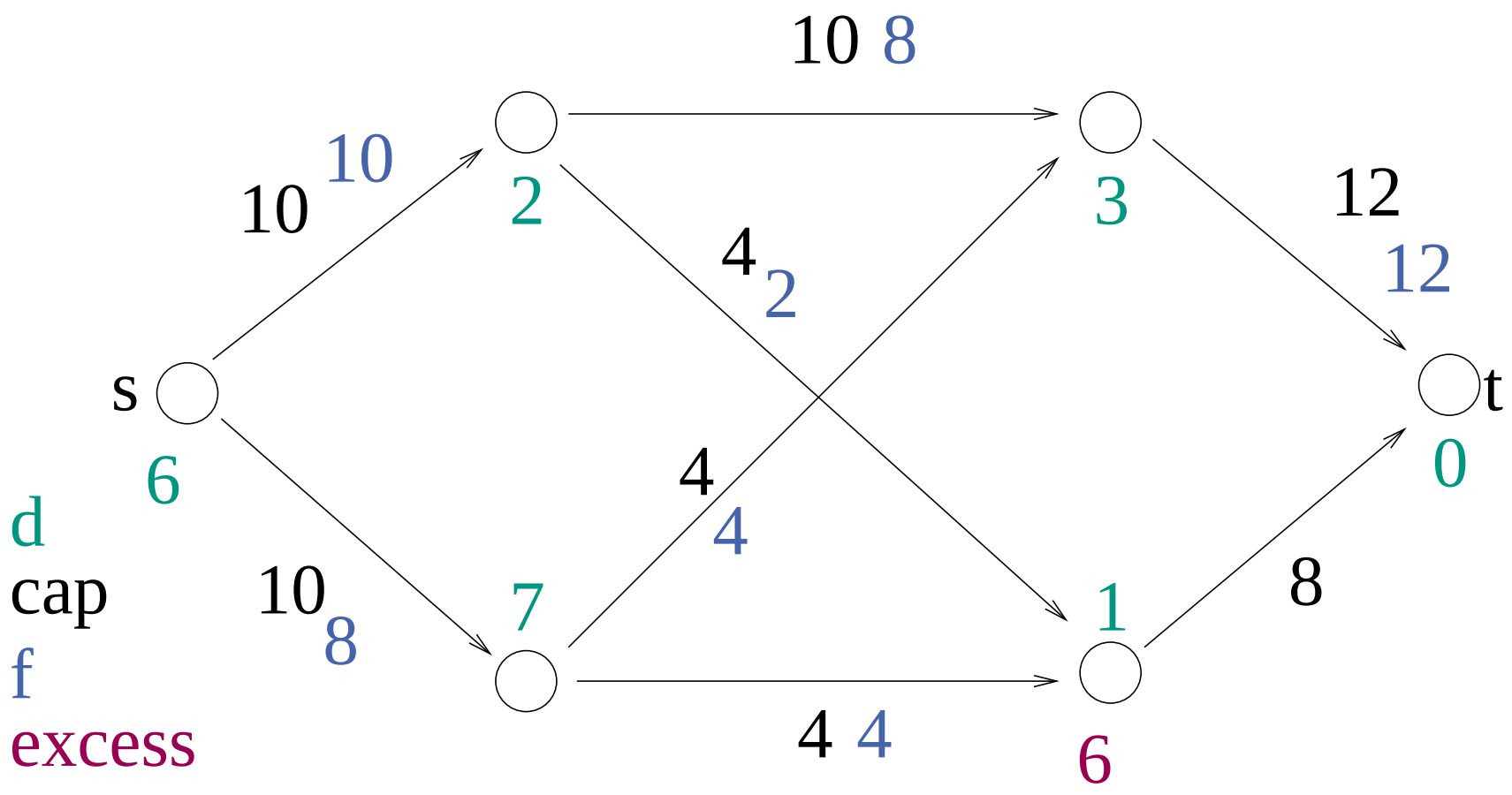
Example



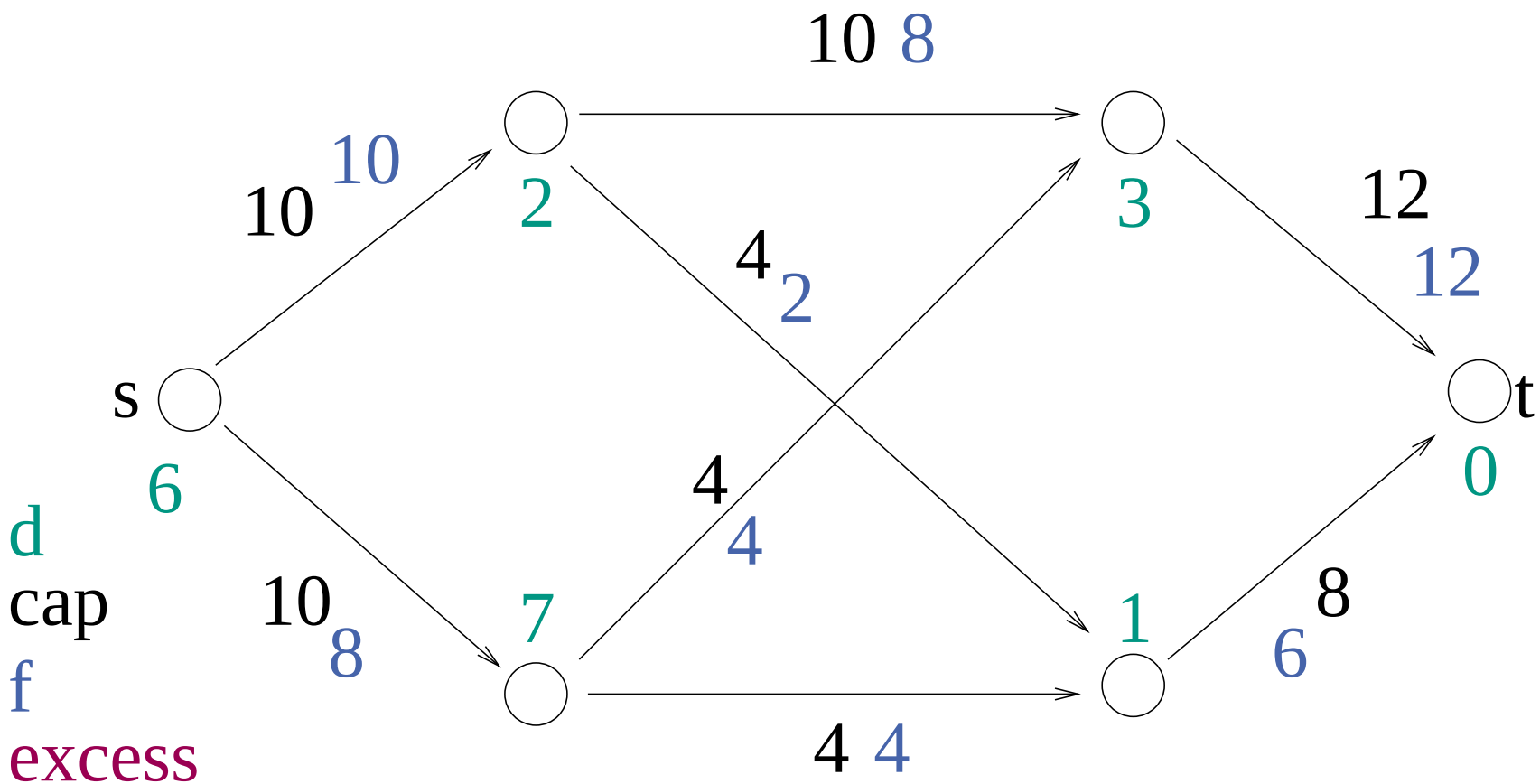
Example



Example



Example



9 pushes in total, 3 less than before

Proof of Lemma 12

$K := \sqrt{m}$ tuning parameter

$d'(v) := \frac{|\{w : d(w) \leq d(v)\}|}{K}$ scaled number of dominated nodes

$\Phi := \sum_{\{v : v \text{ is active}\}} d'(v).$ (Potential)

$d^* := \max\{d(v) : v \text{ is active}\}$ (highest level)

phase:= all pushes between two consecutive changes of d^*

expensive phase: more than K pushes

cheap phase: otherwise

Claims:

1. $\leq 4n^2K$ nonsaturating pushes in all cheap phases together
2. $\Phi \geq 0$ always, $\Phi \leq n^2/K$ initially (obvious)
3. a relabel or saturating push increases Φ by at most n/K.
4. a nonsaturating push does not increase Φ.
5. an expensive phase with $Q \geq K$ nonsaturating pushes decreases Φ by at least Q.

Operation	Amount
Relabel	$2n^2$
Sat.push	nm

Lemma 7+Lemma 8+2.+3.+4.:$\Rightarrow$

total possible decrease $\leq (2n^2 + nm)\frac{n}{K} + \frac{n^2}{K}$

This $+5.$:$\leq \frac{2n^3+n^2+mn^2}{K}$ nonsaturating pushes in expensive phases

This $+1.$:$\leq \frac{2n^3+n^2+mn^2}{K} + 4n^2K = \mathrm{O}\left(n^2\sqrt{m}\right)$ nonsaturating pushes overall for $K = \sqrt{m}$ □

Claims:

1. $\leq 4n^2K$ nonsaturating pushes in all cheap phases together

We first show that there are at most $4n^2$ phases

(changes of $d^* = \max\{d(v) : v \text{ is active}\}$).

$d^* = 0$ initially, $d^* \geq 0$ always.

Only relabel operations increase d^*, i.e.,

$\leq 2n^2$ increases by Lemma 7 and hence

$\leq 2n^2$ decreases

$\leq 4n^2$ changes overall

By definition of a cheap phase, it has at most K pushes.

Claims:

1. $\leq 4n^2K$ nonsaturating pushes in all cheap phases together
2. $\Phi \geq 0$ always, $\Phi \leq n^2/K$ initially (obvious)
3. a relabel or saturating push increases Φ by at most n/K.

Let v denote the relabeled or activated node.

$$d'(v) := \frac{|\{w : d(w) \leq d(v)\}|}{K} \leq \frac{n}{K}$$

A relabel of v can increase only the d'-value of v.

A saturating push on (u, w) may activate only w.

Claims:

1. $\leq 4n^2K$ nonsaturating pushes in all cheap phases together
2. $\Phi \geq 0$ always, $\Phi \leq n^2/K$ initially (obvious)
3. a relabel or saturating push increases Φ by at most n/K.
4. a nonsaturating push does not increase Φ.

v is deactivated (excess(v) is now 0)

w may be activated

but $d'(w) \leq d'(v)$ (we do not push flow away from the sink)

Claims:

1. $\leq 4n^2K$ nonsaturating pushes in all cheap phases together
2. $\Phi \geq 0$ always, $\Phi \leq n^2/K$ initially (obvious)
3. a relabel or saturating push increases Φ by at most n/K.
4. a nonsaturating push does not increase Φ.
5. an expensive phase with $Q \geq K$ nonsaturating pushes decreases Φ by at least Q.

During a phase d^* remains constant

Each nonsat. push decreases the number of nodes at level d^*

Hence, $|\{w : d(w) = d^*\}| \geq Q \geq K$ during an expensive phase

Each nonsat. push across (v,w) decreases Φ by

$\geq d'(v) - d'(w) \geq |\{w : d(w) = d^*\}|/K \geq K/K = 1$ ■

Claims:

1. $\leq 4n^2K$ nonsaturating pushes in all cheap phases together
2. $\Phi \geq 0$ always, $\Phi \leq n^2/K$ initially (obvious)
3. a relabel or saturating push increases Φ by at most n/K.
4. a nonsaturating push does not increase Φ.
5. an expensive phase with $Q \geq K$ nonsaturating pushes decreases Φ by at least Q.

Operation	Amount
Relabel	$2n^2$
Sat.push	nm

Lemma 7+Lemma 8+2.+3.+4.:$\Rightarrow$

total possible decrease $\leq (2n^2 + nm)\frac{n}{K} + \frac{n^2}{K}$

This $+5.$:$\leq \frac{2n^3+n^2+mn^2}{K}$ nonsaturating pushes in expensive phases

This $+1.$:$\leq \frac{2n^3+n^2+mn^2}{K} + 4n^2K = \mathrm{O}\left(n^2\sqrt{m}\right)$ nonsaturating pushes overall for $K = \sqrt{m}$ □

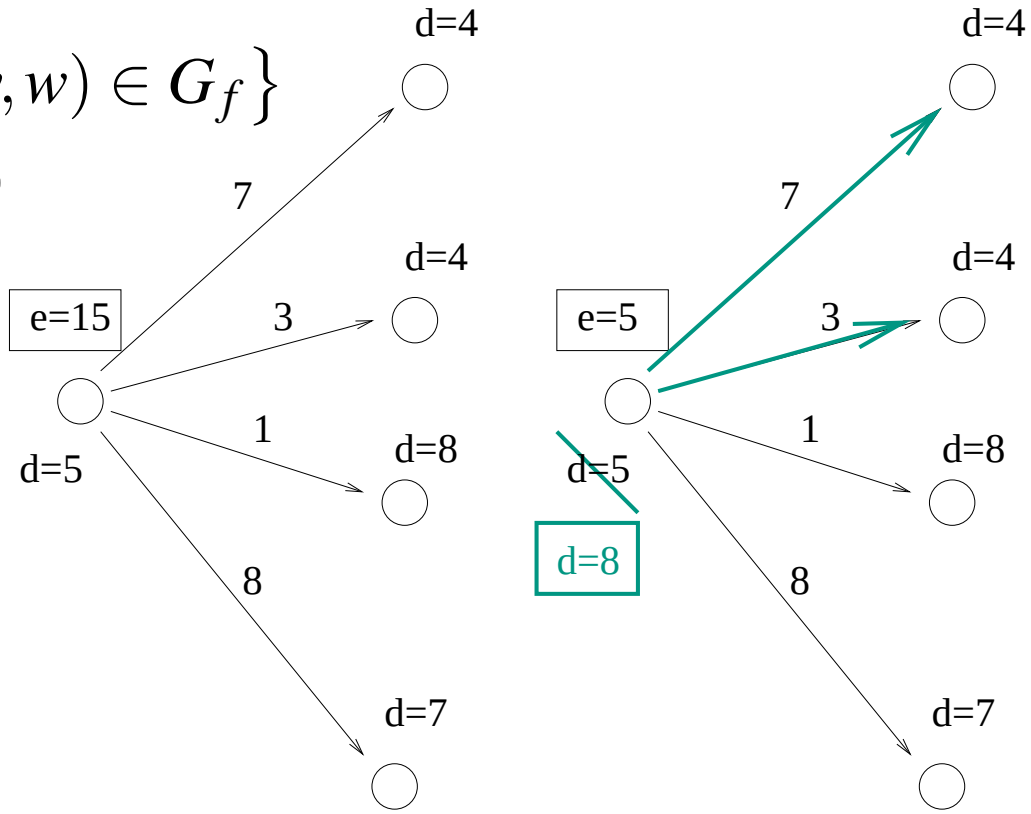
Heuristic Improvements

Naive algorithm has best case $\Omega\left(n^2\right)$. Why? We can do better.

aggressive local relabeling:

$$d(v) := 1 + \min\left\{d(w) : (v,w) \in G_f\right\}$$

(like a sequence of relabels)



Heuristic Improvements

Naive algorithm has best case $\Omega\left(n^2\right)$. Why?

We can do better.

aggressive local relabeling: $d(v) := 1 + \min\left\{d(w) : (v,w) \in G_f\right\}$

(like a sequence of relabels)

global relabeling: (initially and every $\mathrm{O}(m)$ edge inspections):

$d(v) := G_f$.reverseBFS(t) for nodes that can reach t in G_f.

Special treatment of nodes with $d(v) \geq n$. (Returning flow is easy)

Gap Heuristics. No node can connect to t across an empty level:

if $\{v : d(v) = i\} = \emptyset$ **then foreach** v with $d(v) > i$ **do** $d(v) := n$

Experimental results

We use four classes of graphs:

- Random: n nodes, $2n+m$ edges; all edges (s,v) and (v,t) exist
- Cherkassky and Goldberg (1997) (two graph classes)
- Ahuja, Magnanti, Orlin (1993)

Timings: Random Graphs

Rule	BASIC	Ln	LRH	GRH	GAP	LEDA
FF	5.84	6.02	4.75	0.07	0.07	—
	33.32	33.88	26.63	0.16	0.17	—
HL	6.12	6.3	4.97	0.41	0.11	0.07
	27.03	27.61	22.22	1.14	0.22	0.16
MF	5.36	5.51	4.57	0.06	0.07	—
	26.35	27.16	23.65	0.19	0.16	—

$n \in \{1000, 2000\}, m = 3n$

FF=FIFO node selection, HL=hightest level, MF=modified FIFO

Ln= $d(v) \geq n$ is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Timings: CG1

Rule	BASIC	Ln	LRH	GRH	GAP	LEDA
FF	3.46	3.62	2.87	0.9	1.01	—
	15.44	16.08	12.63	3.64	4.07	—
HL	20.43	20.61	20.51	1.19	1.33	0.8
	192.8	191.5	193.7	4.87	5.34	3.28
MF	3.01	3.16	2.3	0.89	1.01	—
	12.22	12.91	9.52	3.65	4.12	—

$n \in \{1000, 2000\}, m = 3n$

FF=FIFO node selection, HL=hightest level, MF=modified FIFO

Ln= $d(v) \geq n$ is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Timings: CG2

Rule	BASIC	Ln	LRH	GRH	GAP	LEDA
FF	50.06	47.12	37.58	1.76	1.96	—
	239	222.4	177.1	7.18	8	—
HL	42.95	41.5	30.1	0.17	0.14	0.08
	173.9	167.9	120.5	0.36	0.28	0.18
MF	45.34	42.73	37.6	0.94	1.07	—
	198.2	186.8	165.7	4.11	4.55	—

$n \in \{1000, 2000\}, m = 3n$

FF=FIFO node selection, HL=hightest level, MF=modified FIFO

Ln= $d(v) \geq n$ is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Timings: AMO

Rule	BASIC	Ln	LRH	GRH	GAP	LEDA
FF	12.61	13.25	1.17	0.06	0.06	—
	55.74	58.31	5.01	0.1399	0.1301	—
HL	15.14	15.8	1.49	0.13	0.13	0.07
	62.15	65.3	6.99	0.26	0.26	0.14
MF	10.97	11.65	0.04999	0.06	0.06	—
	46.74	49.48	0.1099	0.1301	0.1399	—

$n \in \{1000, 2000\}, m = 3n$

FF=FIFO node selection, HL=hightest level, MF=modified FIFO

Ln= $d(v) \geq n$ is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Karlsruhe Institute of Technology

Asymptotics, $n \in \{5000, 10000, 20000\}$

Gen	Rule	GRH			GAP			LEDA		
rand	FF	0.16	0.41	1.16	0.15	0.42	1.05	—	—	—
	HL	1.47	4.67	18.81	0.23	0.57	1.38	0.16	0.45	1.09
	MF	0.17	0.36	1.06	0.14	0.37	0.92	—	—	—
CG1	FF	3.6	16.06	69.3	3.62	16.97	71.29	—	—	—
	HL	4.27	20.4	77.5	4.6	20.54	80.99	2.64	12.13	48.52
	MF	3.55	15.97	68.45	3.66	16.5	70.23	—	—	—
CG2	FF	6.8	29.12	125.3	7.04	29.5	127.6	—	—	—
	HL	0.33	0.65	1.36	0.26	0.52	1.05	0.15	0.3	0.63
	MF	3.86	15.96	68.42	3.9	16.14	70.07	—	—	—
AMO	FF	0.12	0.22	0.48	0.11	0.24	0.49	—	—	—
	HL	0.25	0.48	0.99	0.24	0.48	0.99	0.12	0.24	0.52
	MF	0.11	0.24	0.5	0.11	0.24	0.48	—	—	—

Zusammenfassung Flows und Matchings

- Natürliche Verallgemeinerung von kürzesten Wegen:
  ein Pfad $\rightsquigarrow$ viele Pfade
- viele Anwendungen
- “schwierigste/allgemeinste” Graph-Probleme, die sich mit kombinatorischen Algorithmen in Polynomialzeit lösen lassen
- Beispiel für nichttriviale Algorithmenanalyse
- Potentialmethode ($\neq$ Knotenpotentiale)
- Algorithm Engineering: practical case $\neq$ worst case.
  Heuristiken/Details/Eingabeeigenschaften wichtig
- Datenstrukturen: bucket queues, graph representation, (dynamic trees)

Karlsruhe Institute of Technology

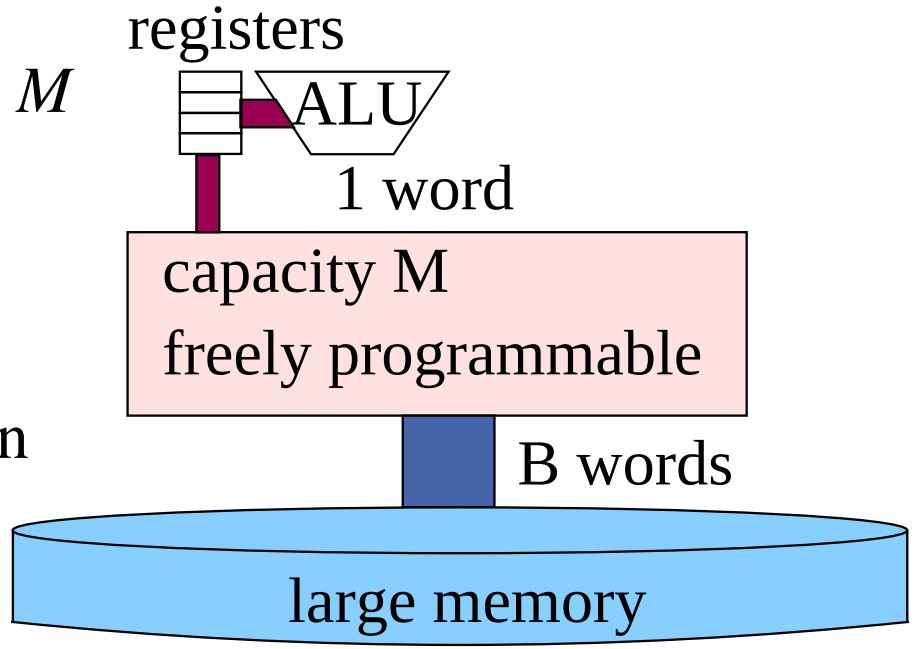
15 Externe Algorithmen

15.1 Das Sekundärspeichermodell

M: Schneller Speicher der Größe M

B: Blockgröße

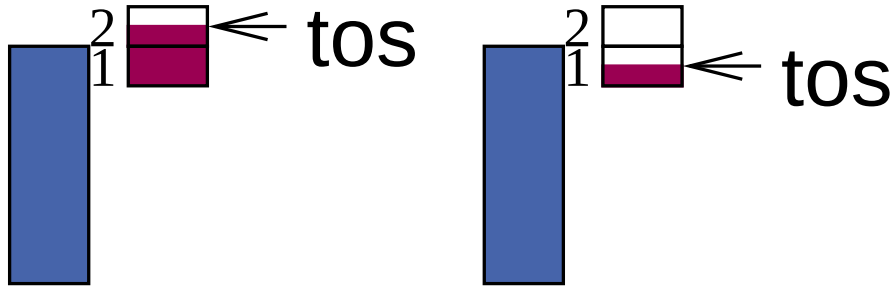
Analyse: Blockzugriffe zählen



15.2 Externe Stapel

Datei mit Blöcken

2 interne Puffer



push: Falls Platz, in Puffer.

Sonst schreibe Puffer eins in die Datei (push auf Blockebene)

Umbenennung: Puffer 1 und 2 tauschen die Plätze

pop: Falls vorhanden, pop aus Puffer.

Sonst lese Puffer eins aus der Datei (pop auf Blockebene)

Analyse: amortisiert $O(1/B)$ I/Os pro Operation

Aufgabe 1: Beweis.

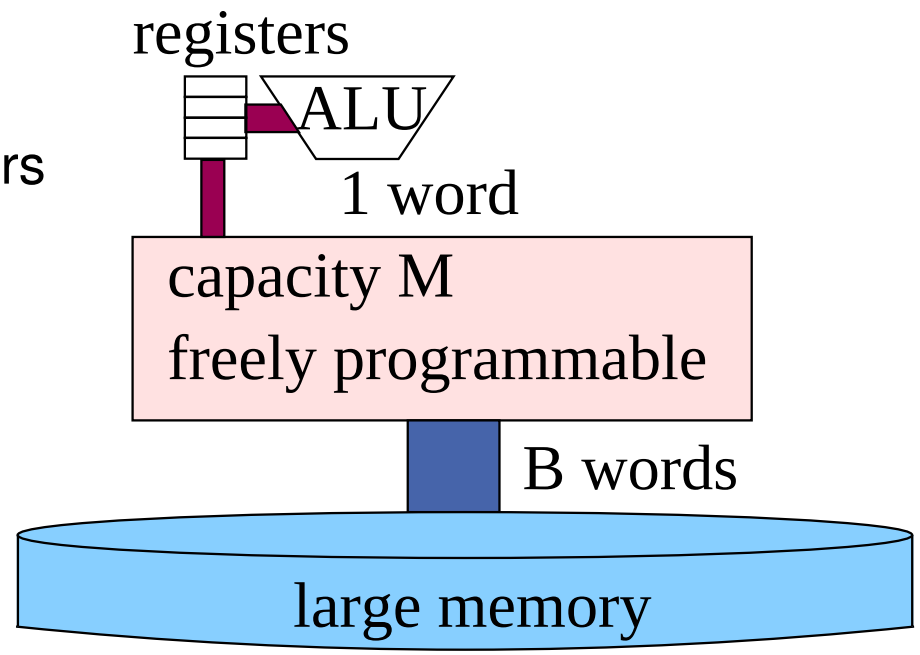
Aufgabe 2: effiziente Implementierung ohne überflüssiges Kopieren

15.3 Externes Sortieren

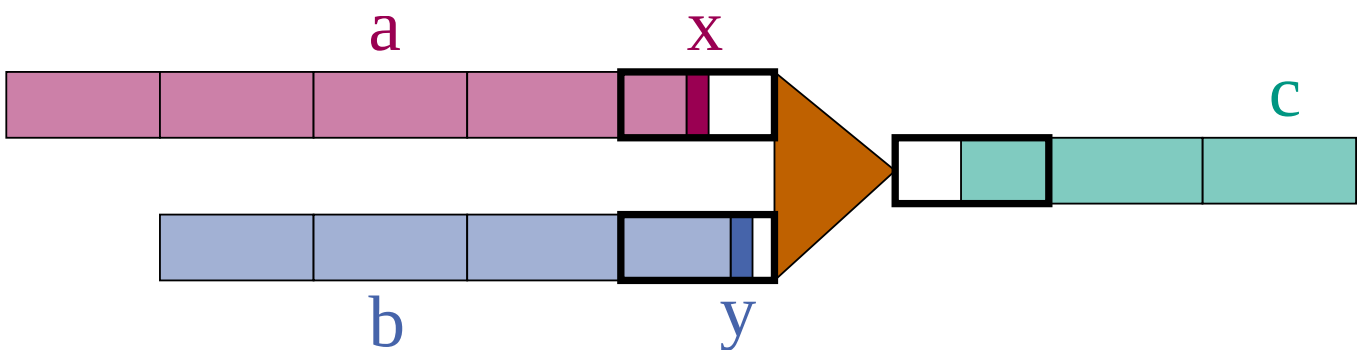
n: Eingabegröße

M: Größe des schnellen Speichers

B: Blockgröße



```
Procedure externalMerge(a, b, c :File of Element)
    x := a.readElement        // Assume emptyFile.readElement= ∞
    y := b.readElement
    for j := 1 to |a| + |b| do
        if x ≤ y then    c.writeElement(x);   x := a.readElement
        else             c.writeElement(y);   y := b.readElement
```



Externes (binäres) Mischen – I/O-Analyse

Datei a lesen: $\lceil |a|/B \rceil \leq |a|/B + 1$.

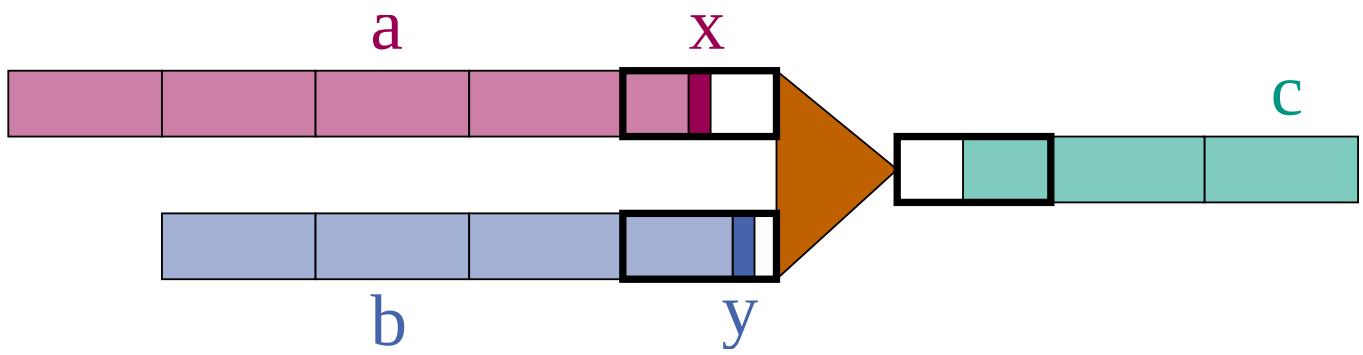
Datei b lesen: $\lceil |b|/B \rceil \leq |b|/B + 1$.

Datei c schreiben: $\lceil (|a| + |b|)/B \rceil \leq (|a| + |b|)/B + 1$.

Insgesamt:

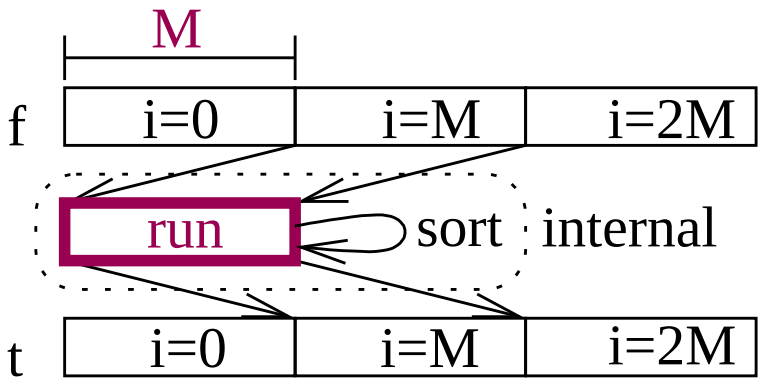
$$\leq 3 + 2\frac{|a| + |b|}{B} \approx 2\frac{|a| + |b|}{B}$$

Bedingung: Wir brauchen 3 Pufferblöcke, d.h., $M > 3B$.



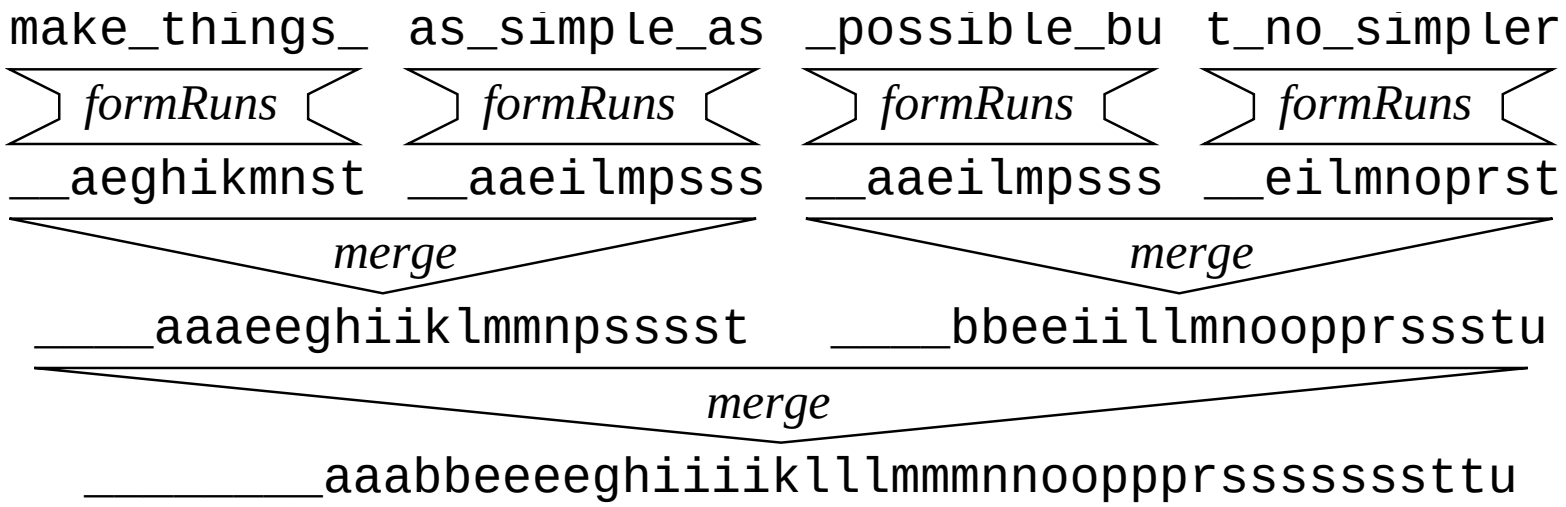
Run Formation

Sortiere Eingabeportionen der Größe M



I/Os: $\approx 2\frac{n}{B}$

Sortieren durch Externes Binäres Mischen



Procedure externalBinaryMergeSort // I/Os: $\approx$

run formation // $2n/B$

while more than one run left **do** // $\lceil \log \frac{n}{M} \rceil \times$

merge pairs of runs // 2n/B

output remaining run // $\sum : 2\frac{n}{B}\left(1 + \left\lceil \log \frac{n}{M} \right\rceil\right)$

Zahlenbeispiel: PC 2010

$n = 2^{39}$ Byte

$M = 2^{33}$ Byte

$B = 2^{22}$ Byte

I/O braucht 2^{-4} s

Zeit: $2\frac{n}{B}\left(1+\left\lceil \log \frac{n}{M} \right\rceil\right) = 2 \cdot 2^{17} \cdot (1+6) \cdot 2^{-4}$ s $= 2^{16}$ s≈ 32 h

Idee: 7 Durchläufe $\rightsquigarrow$ 2 Durchläufe

Mehrwegemischen

Procedure multiwayMerge$(a_1, \ldots, a_k, c$:File **of** Element$)$

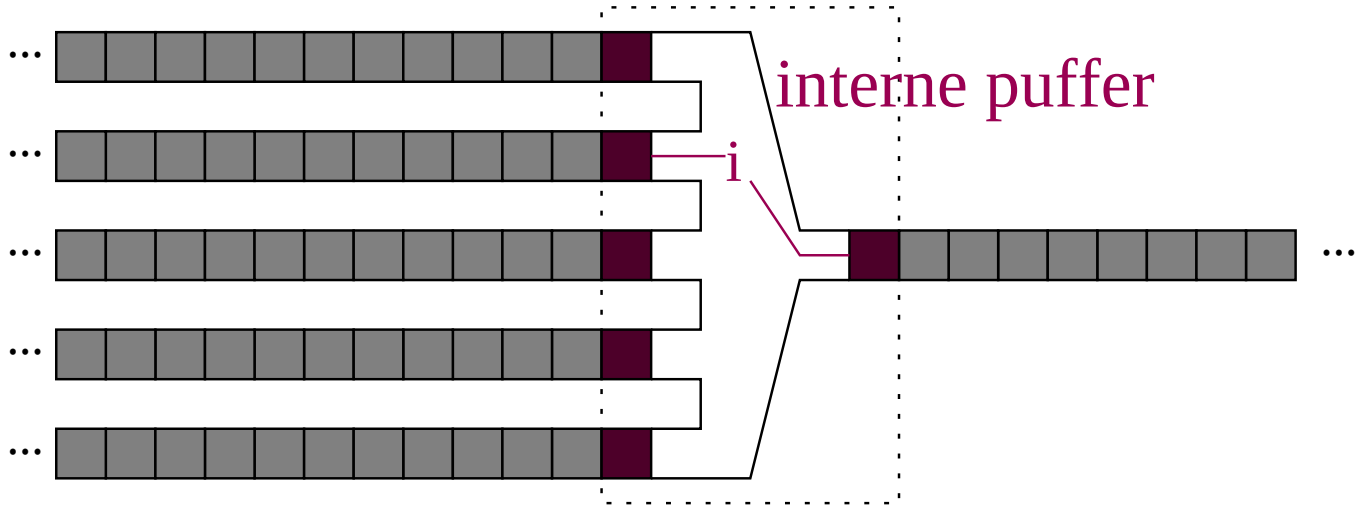
for $i := 1$ **to** k **do** $x_i := a_i$.readElement

for $j := 1$ **to** $\sum_{i=1}^{k} |a_i|$ **do**

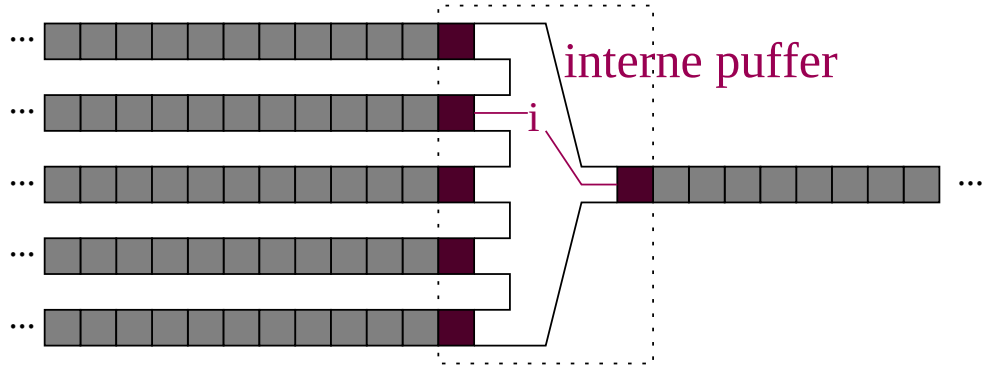
find $i \in 1..k$ that minimizes x_i // no I/Os!, $O(\log k)$ time

c.writeElement(x_i)

$x_i := a_i$.readElement



Mehrwegemischen – Analyse



I/Os: Datei a_i lesen: $\approx |a_i|/B$.

Datei c schreiben: $\approx \sum_{i=1}^{k} |a_i|/B$

Insgesamt:

$$\leq\approx 2\frac{\sum_{i=1}^{k} |a_i|}{B}$$

Bedingung: Wir brauchen $k+1$ Pufferblöcke, d.h., $k+1 < M/B$

(im Folgenden vereinfacht zu $k < M/B$)

Interne Arbeit: (benutze Prioritätsliste !)

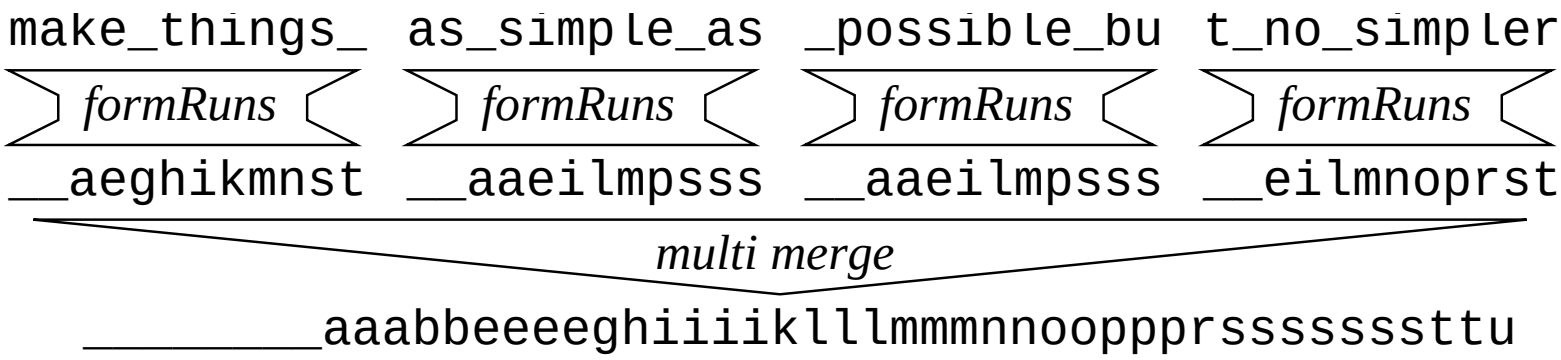
$$\mathrm{O}\left(\log k \sum_{i=1}^{k} |a_i|\right)$$

Sortieren durch Mehrwege-Mischen

- Sortiere $\lceil n/M \rceil$ runs mit je M Elementen — $2n/B$ I/Os
- Mische jeweils M/B runs — $2n/B$ I/Os
- bis nur noch ein run übrig ist — $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ Mischphasen

Insgesamt

$$\text{sort}(n) := \frac{2n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) \text{ I/Os}$$



Sortieren durch Mehrwege-Mischen

Interne Arbeit:

$$\mathrm{O}\left(\overbrace{n\log M}^{\text{run formation}} + \underbrace{n\log\frac{M}{B}}_{\text{PQ access per phase}} \overbrace{\left\lceil\log_{M/B}\frac{n}{M}\right\rceil}^{\text{phases}}\right) = \mathrm{O}(n\log n)$$

Mehr als eine Mischphase?:

Nicht für Hierarchie Hauptspeicher, Festplatte.

Grund $\overbrace{\frac{M}{B}}^{>1000} > \overbrace{\frac{\text{RAM Euro/bit}}{\text{Platte Euro/bit}}}^{\approx 400}$

Mehr zu externem Sortieren

Untere Schranke $\approx \frac{2^{(?)}n}{B}\left(1+\left\lceil \log_{M/B}\frac{n}{M}\right\rceil\right)$ I/Os

[Aggarwal Vitter 1988]

Obere Schranke $\approx \frac{2n}{DB}\left(1+\left\lceil \log_{M/B}\frac{n}{M}\right\rceil\right)$ I/Os (erwartet)

für D parallele Platten

[Hutchinson Sanders Vitter 2005, Dementiev Sanders2003]

Offene Frage: deterministisch?

Externe Prioritätslisten

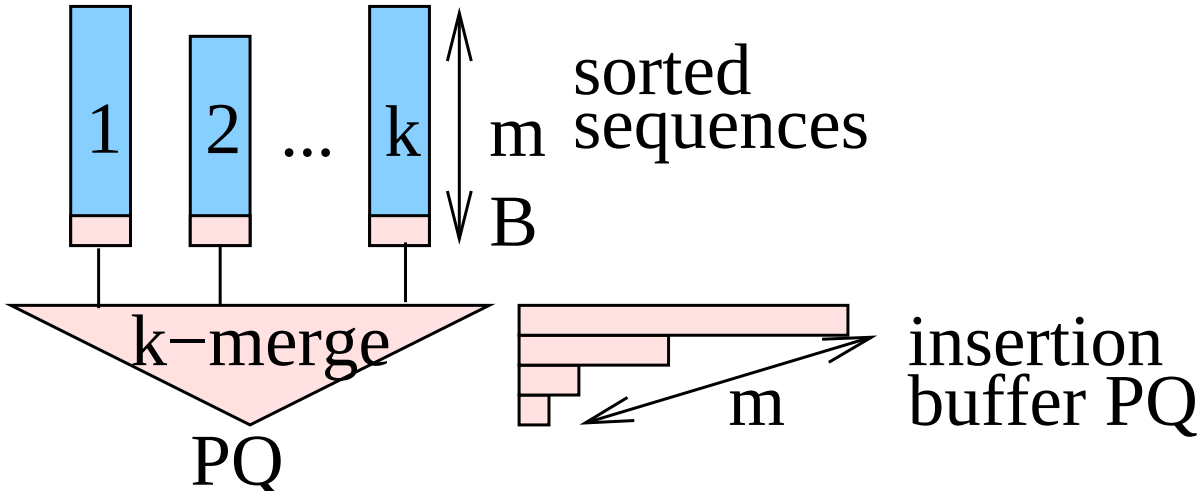
Problem: Binary heaps brauchen

$\Theta\left(\log \frac{n}{M}\right)$ I/Os pro deleteMin

Wir hätten gerne:

$\Theta\left(\frac{1}{B} \log_{M/B} \frac{n}{M}\right)$ I/Os amortisiert

Mittelgroße PQs – $km \ll M^2/B$ Einfügungen



Insert: Anfangs in insertion buffer.

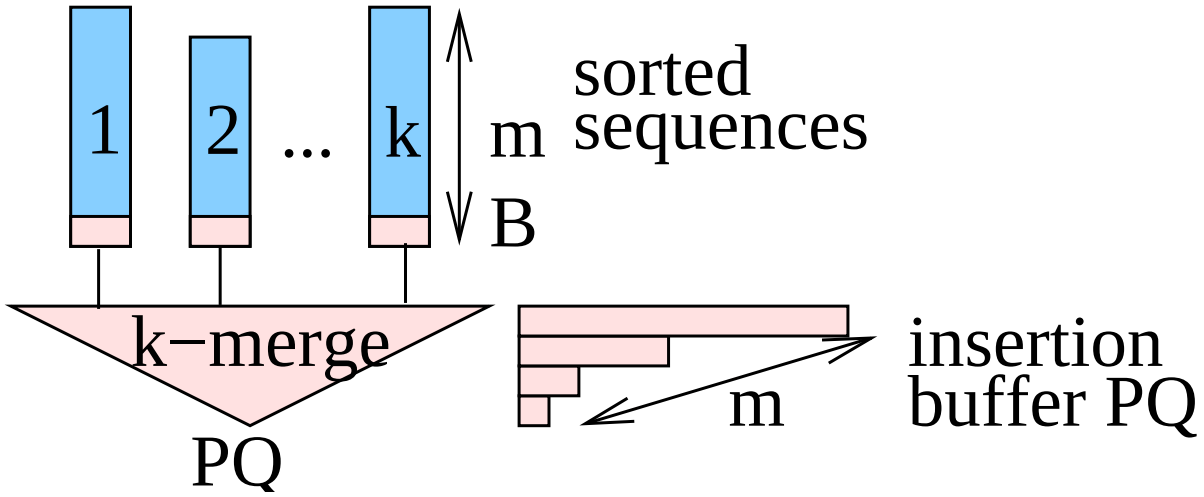
Überlauf $\longrightarrow$

sort; flush; kleinster Schlüssel in merge-PQ

Delete-Min: deleteMin aus der PQ mit kleinerem $\min$

Analyse – I/Os

deleteMin: jedes Element wird $\leq 1\times$ gelesen, zusammen mit B anderen – amortisiert $1/B$ penalty für insert.



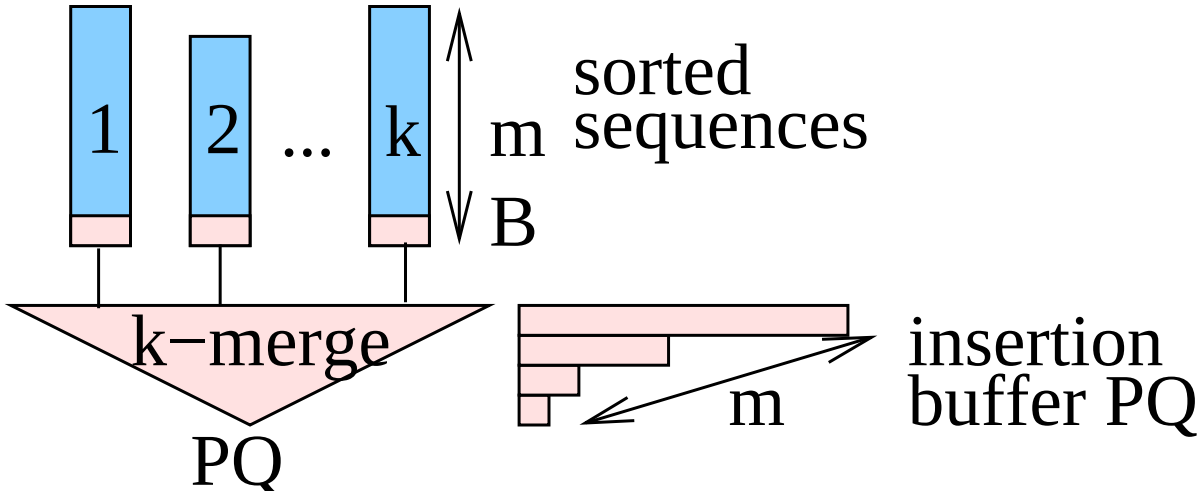
Analyse – Vergleiche (Maß für interne Arbeit)

deleteMin: $1 + \mathrm{O}(\max(\log k, \log m)) = \mathrm{O}(\log m)$

genauere Argumentation: amortisiert $1 + \log k$ bei geeigneter PQ

insert: $\approx m \log m$ alle m Ops. Amortisiert $\log m$

Insgesamt nur $\log km$ amortisiert !



Große Queues

$$\approx \frac{2n}{B}\left(1+\left\lceil \log_{M/B}\frac{n}{M}\right\rceil\right)$$

I/Os für n Einfügeoperationen

$O(n \log n)$ Arbeit.

[Sanders 1999].

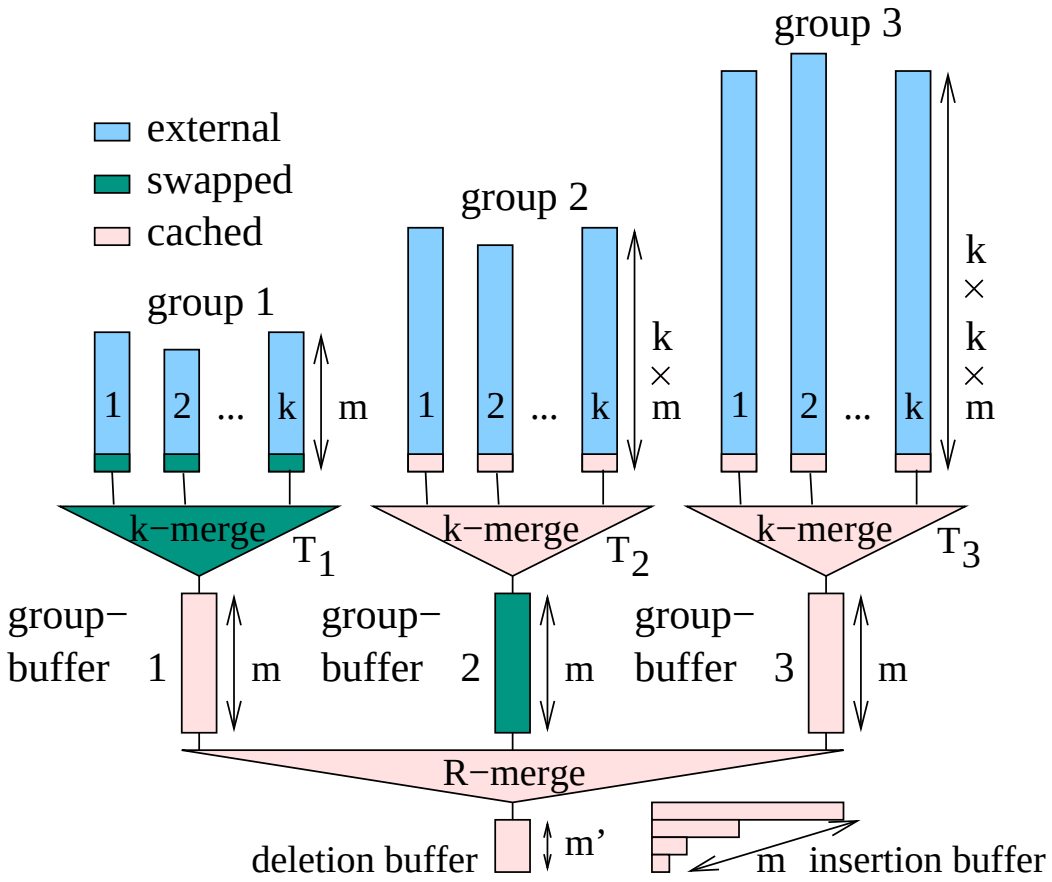
deleteMin:

“amortisiert umsonst”.

Details:

Vorlesung

Algorithm Engineering.



Experiments

Keys: random 32 bit integers

Associated information: 32 dummy bits

Deletion buffer size: 32

Group buffer size: 256

Merging degree k: 128

Near optimal

: performance on

all machines tried!

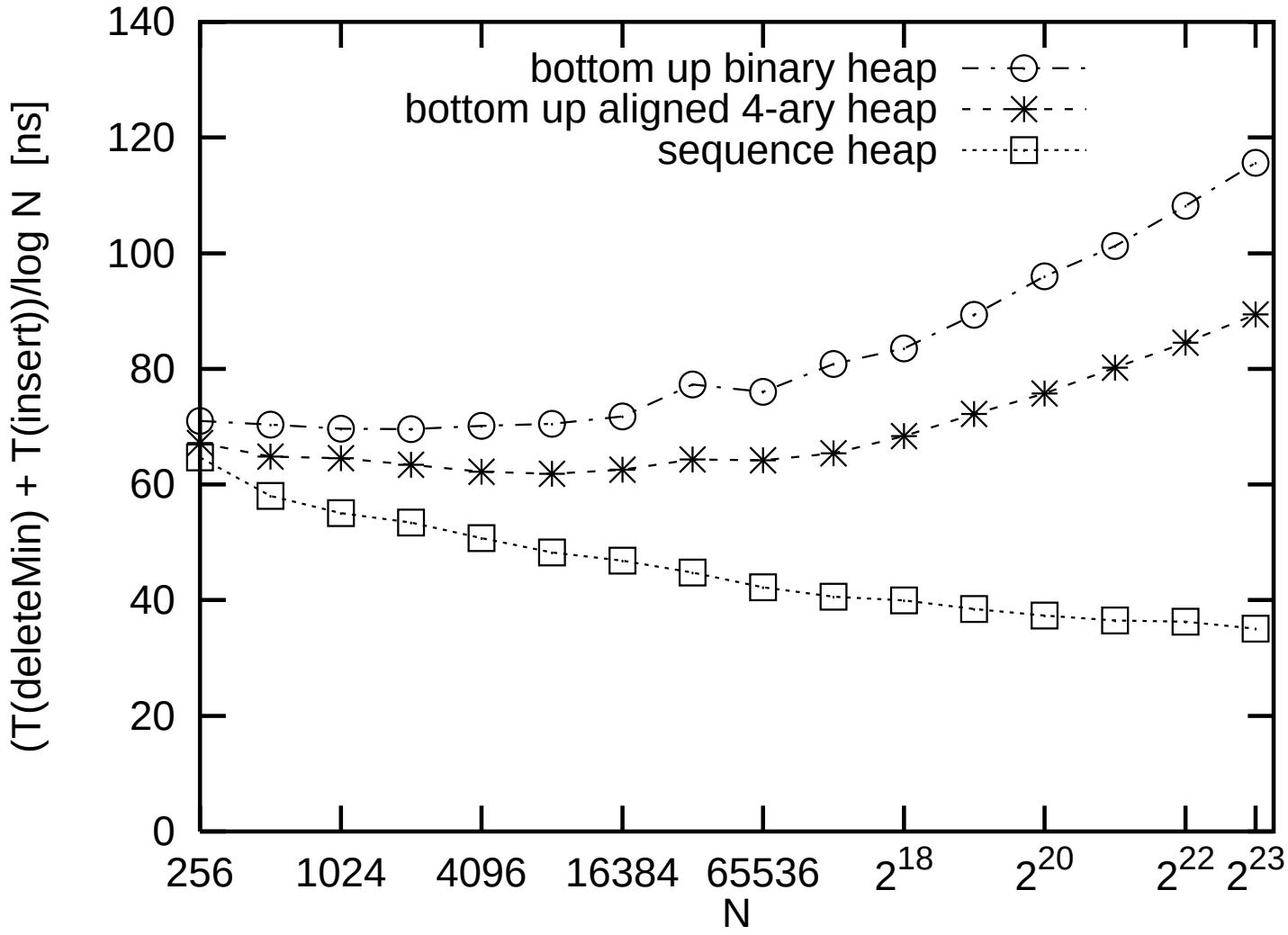
Compiler flags: Highly optimizing, nothing advanced

Operation Sequence:

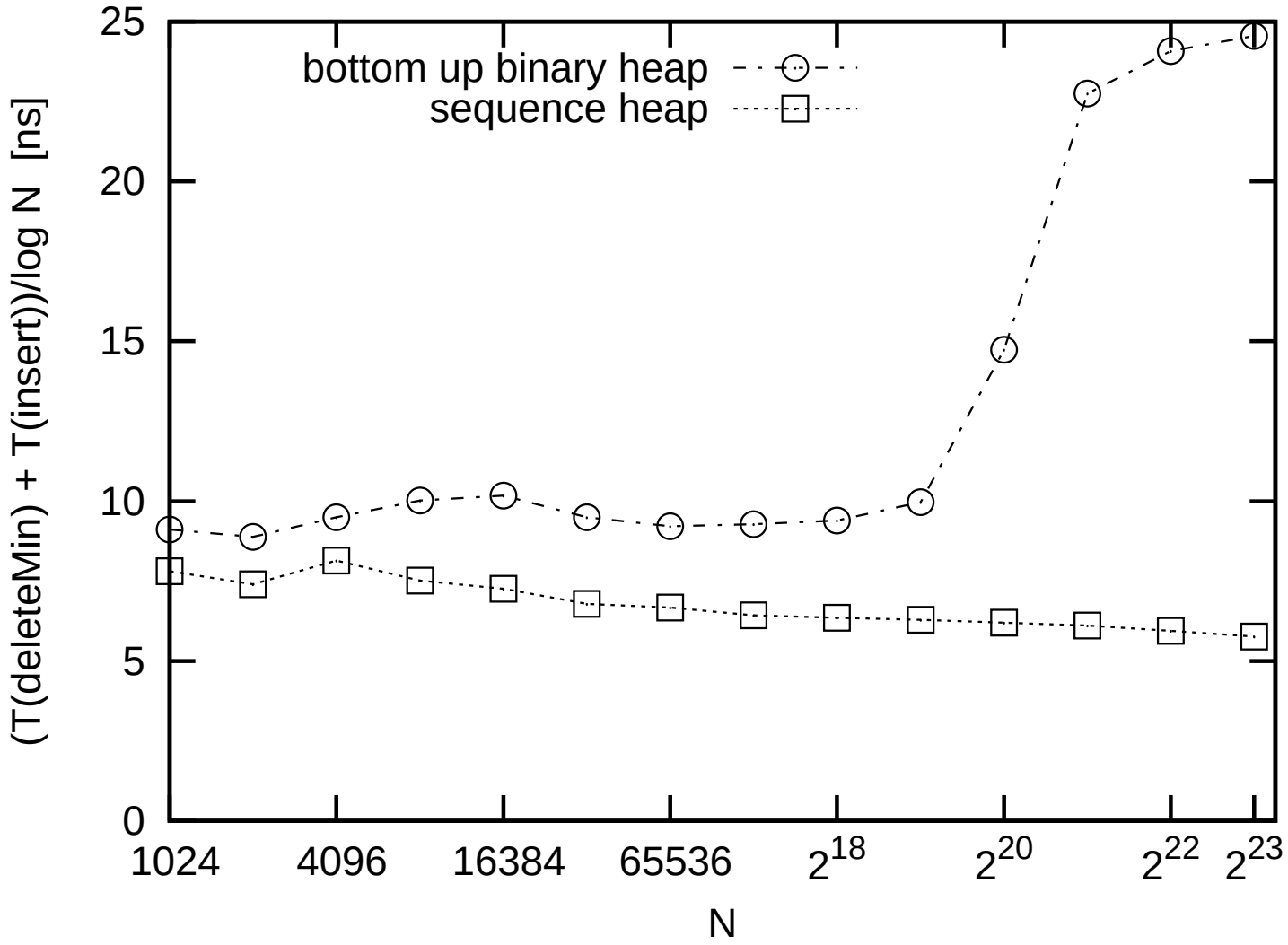
$$\big(\text{Insert-DeleteMin-Insert}\big)^N\big(\text{DeleteMin-Insert-DeleteMin}\big)^N$$

Near optimal performance on all machines tried!

Alpha-21164, 533 MHz



Core2 Duo Notebook, 1.??? GHz



Minimale Spannbäume

Semiexterner Kruskal

Annahme: $M = \Omega(n)$ konstant viele Maschinenworte pro Knoten

Procedure seKruskal$(G = (1..n, E))$

sort E by decreasing weight // sort(m) I/Os

Tc : UnionFind(n)

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

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

output $\{u,v\}$

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

Externe MST-Berechnung

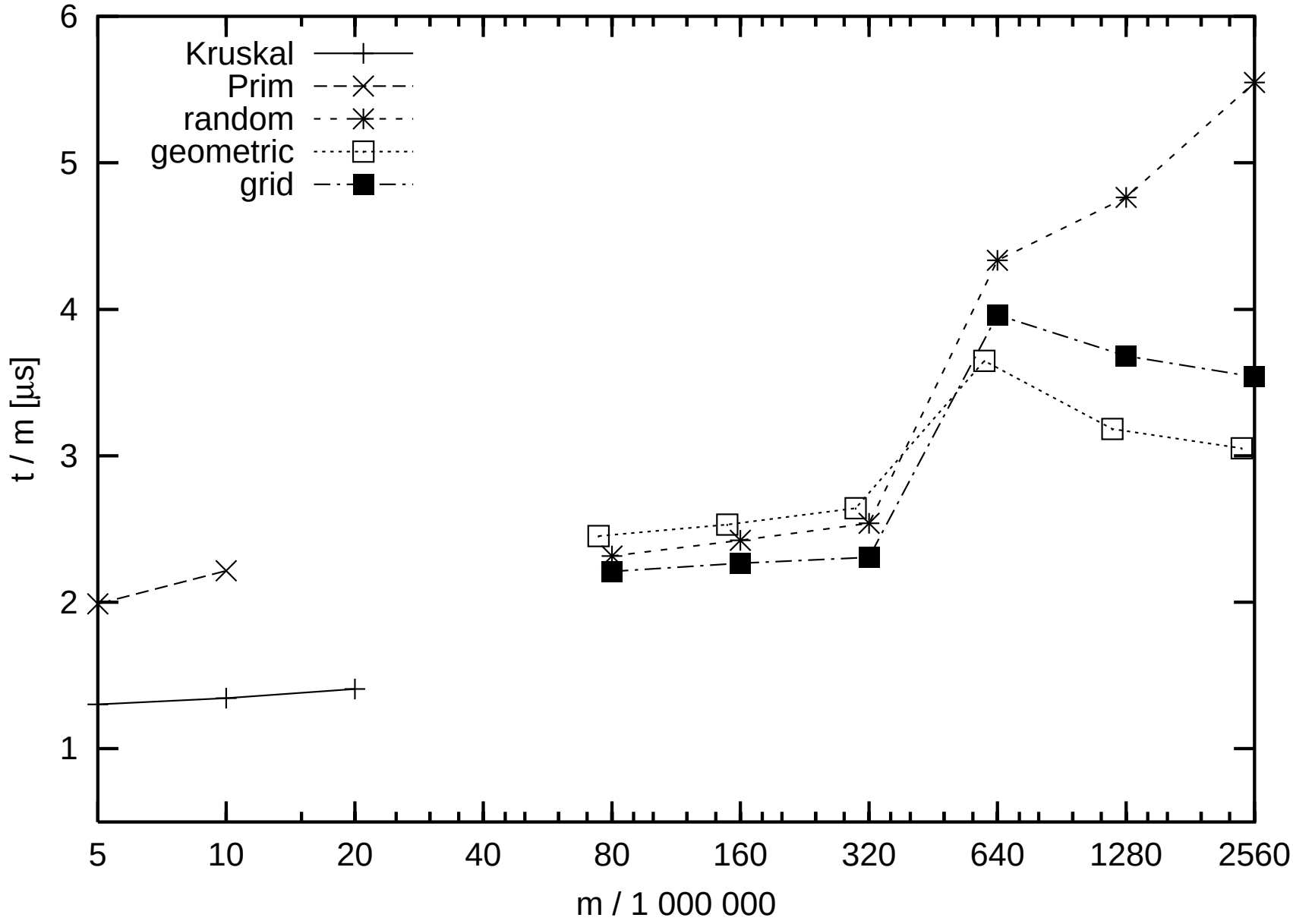
- Reduziere Knotenzahl mittels Kontraktion von MST-Kanten

  Details: Vorlesung Algorithm Engineering, Sibeyn's Algorithmus.

  Implementierung $\approx$ Sortierer $+$ ext. Prioritätsliste $+$ 1 Bildschirmseite. (STXXL Bibliothek)

- benutze semiexternen Algorithmus sobald $n < M$.

Beispiel, Sibeyn's algorithm, $m \approx 2n$



Mehr zu externen Algorithmen – Basic Toolbox ?

Externe Hashtabellen: geht aber 1 I/O pro Zugriff

Suchbäume: $(a, 2a)$-Bäume mit $a = \Theta(B) \rightsquigarrow \log_B n$ I/Os für Basisoperationen. Brot-und-Butter-Datenstruktur für Datenbanken. Inzwischen auch in Dateisystemen. Viel Tuning: Große Blätter, Caching, ...

BFS: OK bei kleinem Graphdurchmesser

DFS: noch schwieriger. Heuristiken für den semiexternen Fall

kürzeste Wege: ähnlich BFS.