

Informatik II

Vorlesung am D-BAUG der ETH Zürich

Felix Friedrich & Hermann Lehner

FS 2019

Willkommen!

Vorlesungshomepage:

<http://lec.inf.ethz.ch/baug/informatik2/2019/>

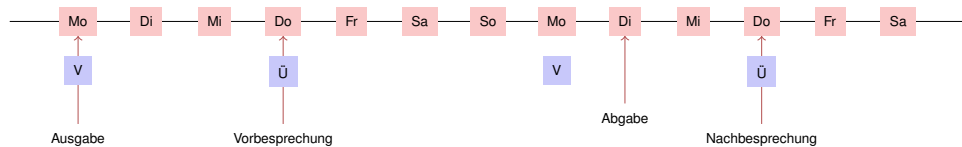
Das Team:

Dozenten	Felix Friedrich Hermann Lehner
Assistenten	Patrick Gruntz Aristeidis Mastoras Chris Wendler Manuel Winkler
Back-Office	Katja Wolff

1

2

Übungsbetrieb



- Übungsblattausgabe zur Vorlesung (online).
- Vorbereitung in der folgenden Übung.
- Bearbeitung der Übung bis spätestens 2 Tage vor der nächsten Übungsstunde (23:59h).
- Nachbereitung der Übung in der nächsten Übungsstunde. Feedback zu den Abgaben innerhalb einer Woche nach Nachbereitung.

Zu den Übungen

- Bearbeitung der wöchentlichen Übungsserien ist freiwillig, wird aber *dringend* empfohlen!

3

4

Es ist so einfach!

Für die Übungen verwenden wir eine Online-Entwicklungsumgebung, benötigt lediglich einen Browser, Internetverbindung und Ihr ETH Login.

Falls Sie keinen Zugang zu einem Computer haben: in der ETH stehen an vielen Orten öffentlich Computer bereit.

5

Relevantes für die Prüfung

Prüfungsstoff für die Endprüfung schliesst ein

- Vorlesungsinhalt (Vorlesung, Handout) und
- Übungsinhalte (Übungsstunden, Übungsaufgaben).

Prüfung ist schriftlich

Es wird sowohl praktisches Wissen (Kenntnis von Algorithmen, Programmierfähigkeit) als auch theoretisches Wissen (Hintergründe, Systematik) geprüft.

7

Literatur

Algorithmen und Datenstrukturen, T. Ottmann, P. Widmayer, Spektrum-Verlag, 5. Auflage, 2011

Algorithmen - Eine Einführung, T. Cormen, C. Leiserson, R. Rivest, C. Stein, Oldenbourg, 2010

Introduction to Algorithms, T. Cormen, C. Leiserson, R. Rivest, C. Stein, 3rd ed., MIT Press, 2009

Algorithmen Kapiern, Aditya Y. Bhargava, MITP, 2019.

6

Unser Angebot

- Bearbeitung der wöchentlichen Übungsserien → Bonus von maximal 0.25 Notenpunkten für die Prüfung.
- Bonus proportional zur erreichten Punktzahl von **speziell markierten Bonus-Aufgaben**. Volle Punktzahl $\hat{=}$ 0.25.
- **Zulassung** zu speziell markierten Bonusaufgaben kann von der erfolgreichen Absolvierung anderer Übungsaufgaben abhängen.

8

Unser Angebot (konkret)

- Insgesamt 3 Bonusaufgaben; 2/3 der Punkte reichen für 0.25 Bonuspunkte für die Prüfung
- Sie können also z.B. 2 Bonusaufgaben zu 100% lösen, oder 3 Bonusaufgaben zu je 66%, oder ...
- Bonusaufgaben müssen durch erfolgreich gelöste Übungsserien freigeschaltet (→ Experience Points) werden
- Es müssen wiederum nicht alle Übungsserien vollständig gelöst werden, um eine Bonusaufgabe freizuschalten
- Details: Übungsstunden, Online-Übungssystem (Code Expert)

Akademische Lauterkeit

Regel: Sie geben nur eigene Lösungen ab, welche Sie selbst verfasst und verstanden haben.

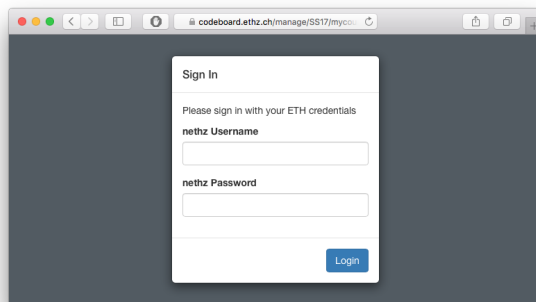
Wir prüfen das (zum Teil automatisiert) nach und behalten uns disziplinarische Massnahmen vor.

9

10

Einschreibung in Übungsgruppen - I

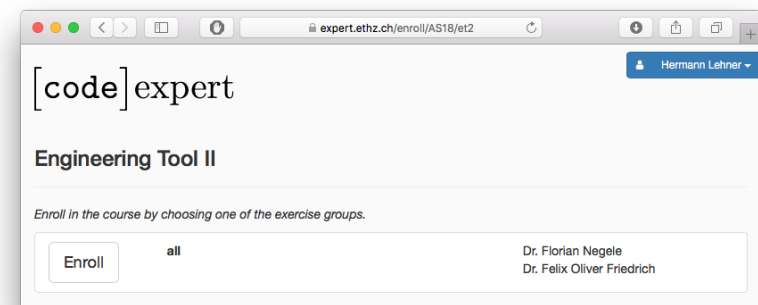
- Besuchen Sie <http://expert.ethz.ch/enroll/SS19/ifbaug2>
- Loggen Sie sich mit Ihrem nethz Account ein.



11

Einschreibung in Übungsgruppen - II

Schreiben Sie sich im folgenden Dialog in eine Übungsgruppe ein.



12

Übersicht

The screenshot shows the [code]expert web interface. At the top, it displays the user's name 'Felix Oliver Friedrich' and the semester 'Autum 2017'. Below this, there are tabs for 'Enrolled Courses', 'My Exercise Groups', and 'My Courses'. The main content area is titled 'Demo Course' and 'Demo Group - Dr. Hermann Lehner'. It features a table with columns for 'Coding Demo Exercise', 'Earned XP', 'Submissions', 'Handout Date', and 'Due Date'. One exercise, 'Quadratic Equations In C++', is highlighted in yellow and shows 100% completion. Below the table, there are links for 'Basic Markdown Syntax' and 'Code Blocks and Inline Code'.

Programmierübung

The screenshot shows a code editor with C++ code for a min-max problem. The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

Annotations on the screenshot include:

- A red box labeled 'D: Beschreibung' pointing to the code.
- A red box labeled 'E: History' pointing to the right sidebar.
- A red box labeled 'A: compile' pointing to the compile button (A).
- A red box labeled 'B: run' pointing to the run button (B).
- A red box labeled 'C: test' pointing to the test button (C).

13

14

Testen und Abgeben

Wo ist der Save Knopf?

The screenshot shows the code editor with test results and submission options. The test results are as follows:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first failed
max_middle failed
unique passed

Input:
100250 45 0 0 1 -1000001 45 -25065 1
expected output:
-1000001/100251
actual output:
-1000001/100250
.....
max_test passed
max_middle passed
unique passed

Tests result: passed 6 of 7 / score: 86% [ ]
```

Annotations on the screenshot include:

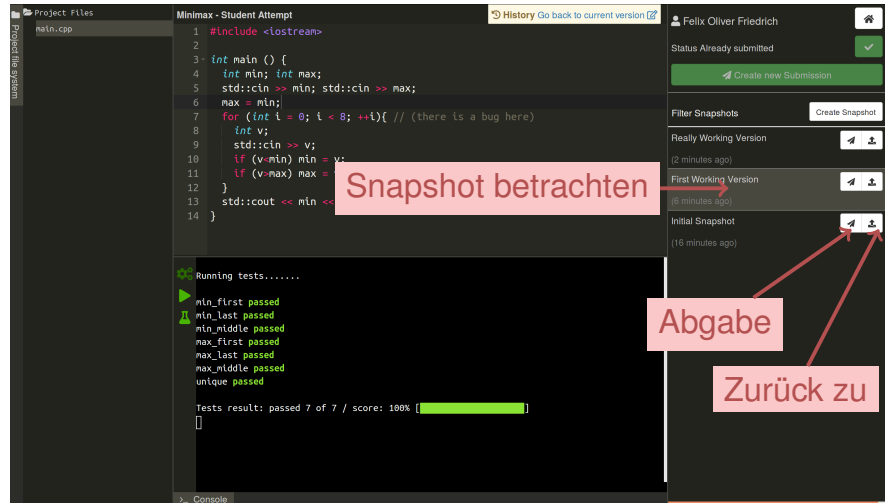
- A red box labeled 'Abgeben' pointing to the 'Submit' button.
- A red box labeled 'Testen' pointing to the test results.

15

16

- Das Filesystem ist transaktionsbasiert und es wird laufend gespeichert („autosave“). Beim Öffnen eines Projektes findet man immer den zuletzt gesehenen Zustand wieder.
- Der derzeitige Stand kann als (benannter) *Snapshot* festgehalten werden. Zu gespeicherten Snapshots kann jederzeit zurückgekehrt werden.
- Der aktuelle Stand kann als Snapshot abgegeben (submitted) werden. Zudem kann jeder gespeicherte Snapshot abgegeben werden.

Snapshots



1. Einführung

Algorithmen und Datenstrukturen, Korrektheit, erstes Beispiel

Wenn es Probleme gibt ...

- mit dem Kursinhalt
 - unbedingt alle Übungen besuchen
 - dort Fragen stellen
 - und/oder Übungsleiter kontaktieren
- alle weiteren Probleme
 - Email an Dozenten (Felix Friedrich, Hermann Lehner)
- Wir helfen gerne!

Ziele der Vorlesung

- Verständnis des Entwurfs und der Analyse grundlegender Algorithmen und Datenstrukturen.
- Fähigkeit, korrekte und ausreichend effiziente Programme entwickeln, um eine klar formulierte Problemstellung zu lösen.

Inhalte der Vorlesung

Datenstrukturen / Algorithmen

Begriff der Invariante, Kostenmodell, Landau Symbole

Algorithmenentwurf, Induktion

Suchen und Auswahl, Sortieren

Wörterbücher: Hashing und Suchbäume, Balancierte Bäume

Dynamische Programmierung

Fundamentale Graphenalgorithmen, Kürzeste Wege, Maximaler Fluss

Software Engineering

Python Introduction

Python Datastructures

1.1 Algorithmen

[Cormen et al, Kap. 1; Ottman/Widmayer, Kap. 1.1]

21

22

Algorithmus

Algorithmus: wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (*input*) Ausgabedaten (*output*) berechnet.

23

Beispielproblem

Input: Eine Folge von n Zahlen (a_1, a_2, \dots, a_n)

Output: Eine Permutation $(a'_1, a'_2, \dots, a'_n)$ der Folge $(a_i)_{1 \leq i \leq n}$, so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Mögliche Eingaben

$(1, 7, 3), (15, 13, 12, -0.5), (1) \dots$

Jedes Beispiel erzeugt eine *Probleminstanz*.

Die Performanz (Geschwindigkeit) des Algorithmus hängt üblicherweise ab von der Probleminstanz. Es gibt oft "gute" und "schlechte" Instanzen.

24

Beispiele für Probleme in der Algorithmik

- **Tabellen und Statistiken**: Suchen, Auswählen und Sortieren
- **Routenplanung**: Kürzeste Wege Algorithmus, Heap Datenstruktur
- **DNA Matching**: Dynamic Programming
- **Auswertungsreihenfolge**: Topologische Sortierung
- **Autovervollständigung**: Wörterbücher/Bäume
- **Schnelles Nachschlagen** : Hash-Tabellen
- **Der Handlungsreisende**: Dynamische Programmierung, Minimal aufspannender Baum, Simulated Annealing,

25

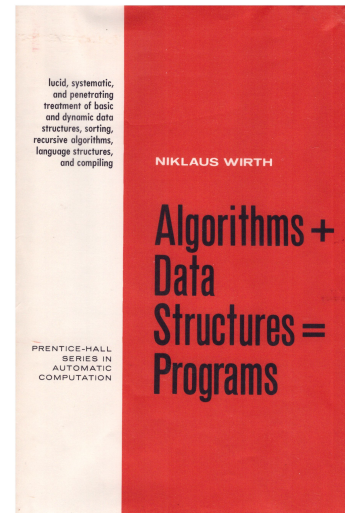
Charakteristik

- Extrem grosse Anzahl potentieller Lösungen
- Praktische Anwendung

26

Datenstrukturen

- Eine Datenstruktur *organisiert Daten* so in einem Computer, dass man sie (in den darauf operierenden Algorithmen) *effizient nutzen* kann.
- Programme = Algorithmen + Datenstrukturen.



27

Effizienz

Illusion:

- Wären Rechner unendlich schnell und hätten unendlich viel Speicher ...
- ... dann bräuchten wir die Theorie der Algorithmen (nur) für Aussagen über Korrektheit (incl. Terminierung).

Realität: Ressourcen sind beschränkt und nicht umsonst:

- Rechenzeit → Effizienz
- Speicherplatz → Effizienz

Eigentlich geht es in diesem Kurs nur um Effizienz.

28

2. Effizienz von Algorithmen

Effizienz von Algorithmen, Random Access Machine Modell, Funktionenwachstum, Asymptotik [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Effizienz von Algorithmen

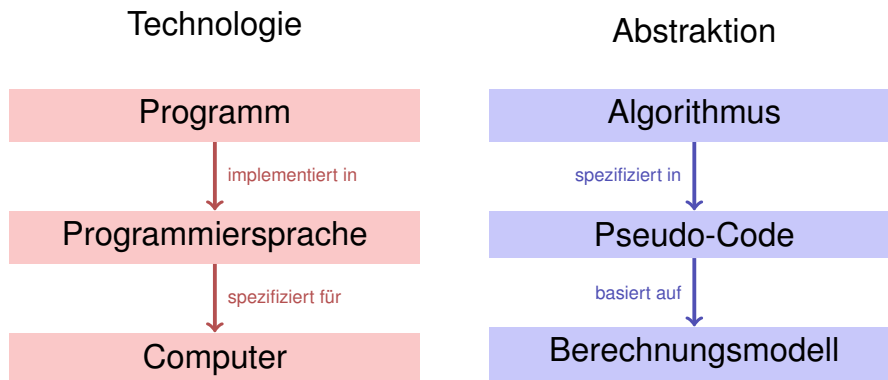
Ziele

- Laufzeitverhalten eines Algorithmus maschinenunabhängig quantifizieren.
- Effizienz von Algorithmen vergleichen.
- Abhängigkeit von der Eingabegrösse verstehen.

29

30

Programme und Algorithmen



31

Technologiemodell

Random Access Machine (RAM)

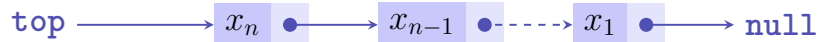
- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit (grosses Array)
- Elementare Operationen: Rechenoperation (+, -, ·, ...), Vergleichsoperationen, Zuweisung / Kopieroperation auf Maschinenworten (Registern), Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.
- Datentypen: Fundamentaltypen wie grössenbeschränkte Ganzzahl oder Fließkommazahl.

32

Pointer Machine Modell

Wir nehmen an

- Objekte beschränkter Grösse können dynamisch erzeugt werden in konstanter Zeit 1.
- Auf Felder (mit Wortgrösse) der Objekte kann in konstanter Zeit 1 zugegriffen werden.



33

Asymptotisches Verhalten

Genauere Laufzeit eines Algorithmus lässt sich selbst für kleine Eingabedaten kaum voraussagen.

- Betrachten das asymptotische Verhalten eines Algorithmus.
- Ignorieren alle konstanten Faktoren.

Beispiel

Eine Operation mit Kosten 20 ist genauso gut wie eine mit Kosten 1. Lineares Wachstum mit Steigung 5 ist genauso gut wie lineares Wachstum mit Steigung 1.

34

Algorithmen, Programme und Laufzeit

Programm: Konkrete Implementation eines Algorithmus.

Laufzeit des Programmes: messbarer Wert auf einer konkreten Maschine. Kann sowohl nach oben, wie auch nach unten abgeschätzt werden.

Beispiel

Rechner mit 3 GHz. Maximale Anzahl Operationen pro Taktzyklus (z.B. 8). \Rightarrow untere Schranke.
Einzelne Operation dauert mit Sicherheit nie länger als ein Tag \Rightarrow obere Schranke.

Hinsichtlich des *asymptotischen Verhaltens* des Programmes spielen die Schranken keine Rolle.

35

2.2 Funktionenwachstum

\mathcal{O} , Θ , Ω [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

36

Oberflächlich

Verwende die asymptotische Notation zur Kennzeichnung der Laufzeit von Algorithmen

Wir schreiben $\Theta(n^2)$ und meinen, dass der Algorithmus sich für grosse n wie n^2 verhält: verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

Genauer: Asymptotische obere Schranke

Gegeben: Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

Definition:¹

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

Schreibweise:

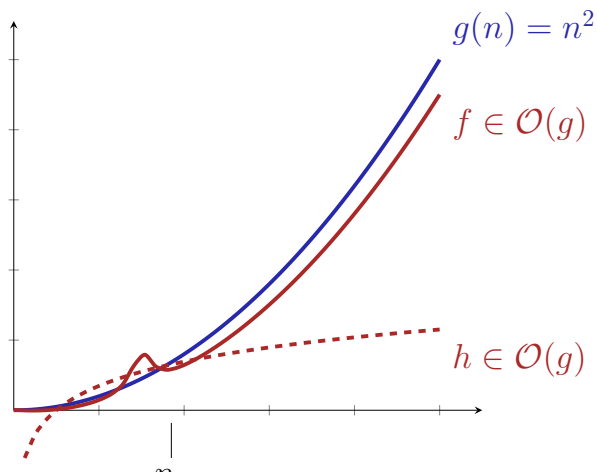
$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

¹Ausgesprochen: Menge aller reellwertiger Funktionen $f : \mathbb{N} \rightarrow \mathbb{R}$ für die gilt: es gibt ein (reellwertiges) $c > 0$ und ein $n_0 \in \mathbb{N}$ so dass $0 \leq f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

37

38

Anschaung



39

Beispiele

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

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

40

Eigenschaft

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

Umkehrung: Asymptotische untere Schranke

Gegeben: Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

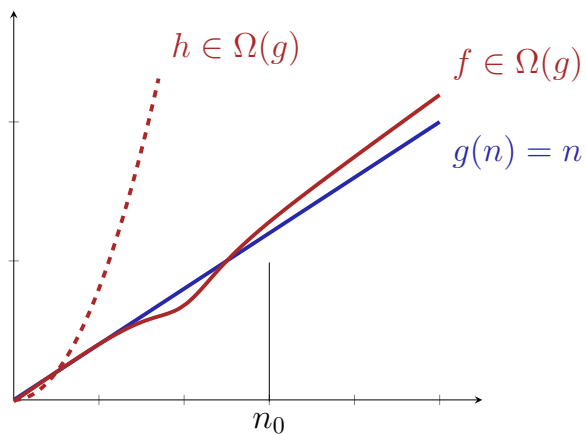
Definition:

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

41

42

Beispiel



43

Asymptotisch scharfe Schranke

Gegeben Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

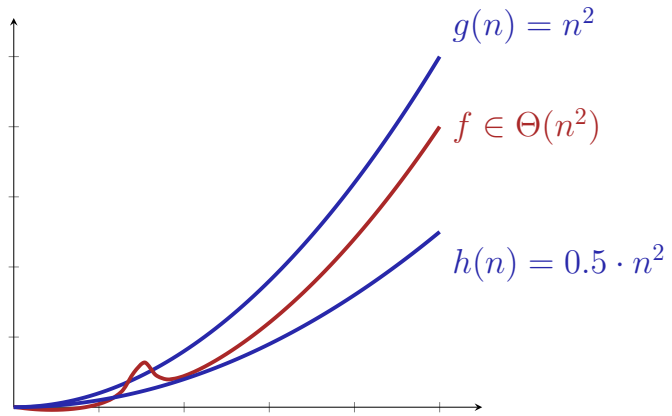
Definition:

$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Einfache, geschlossene Form: Übung.

44

Beispiel



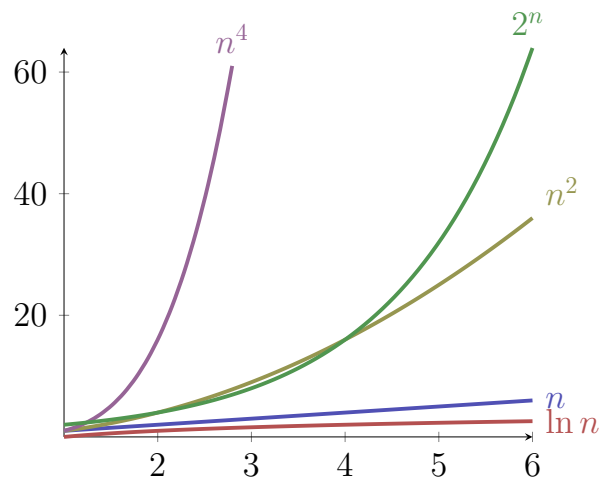
45

Wachstumsbezeichnungen

$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log \log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n \log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(2^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

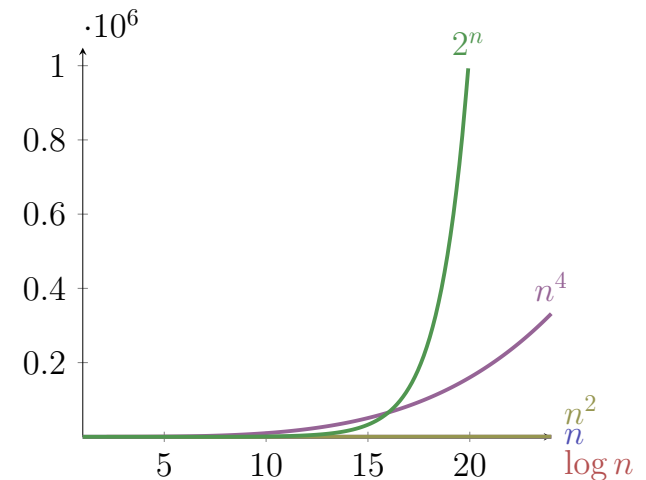
46

Kleine n



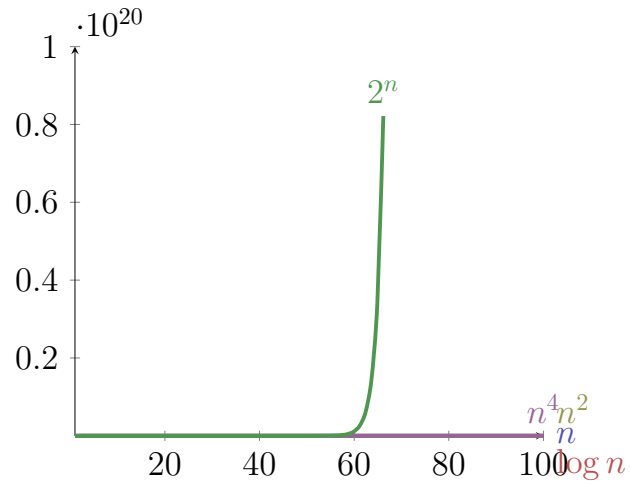
47

Größere n



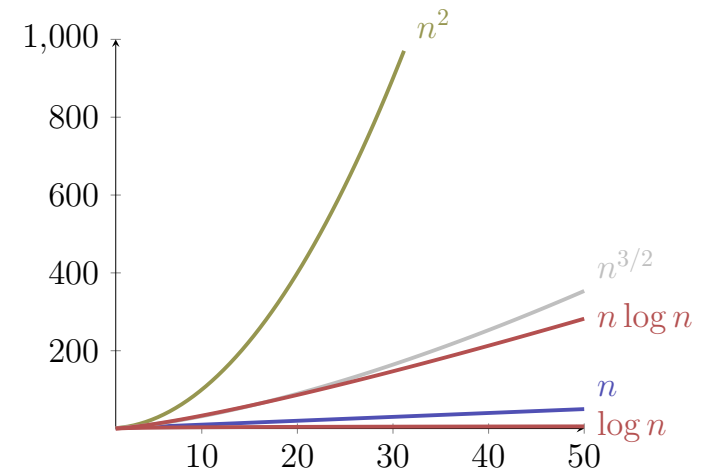
48

“Grosse” n



49

Logarithmen!



50

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$	10^{14} Jahrh.	$\approx \infty$	$\approx \infty$	$\approx \infty$

51

Nützlich

Theorem

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ zwei Funktionen. Dann gilt:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subsetneq \mathcal{O}(g)$.
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C > 0$ (C konstant) $\Rightarrow f \in \Theta(g)$.
- $\frac{f(n)}{g(n)} \xrightarrow{n \rightarrow \infty} \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

52

Zur Notation

Übliche informelle Schreibweise

$$f = \mathcal{O}(g)$$

ist zu verstehen als $f \in \mathcal{O}(g)$.

Es gilt nämlich

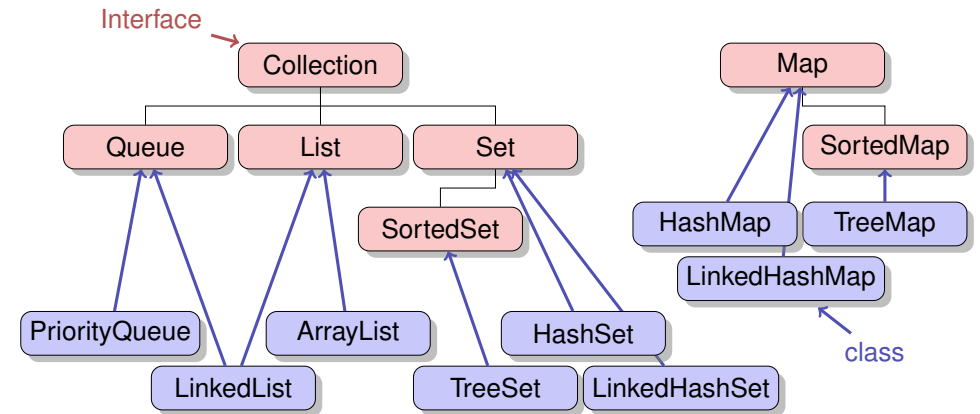
$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

Beispiel

$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$ aber natürlich $n \neq n^2$.

Wir vermeiden diese Schreibweise, wo sie zu Mehrdeutigkeiten führen könnte.

Erinnerung: Java Collections / Maps



53

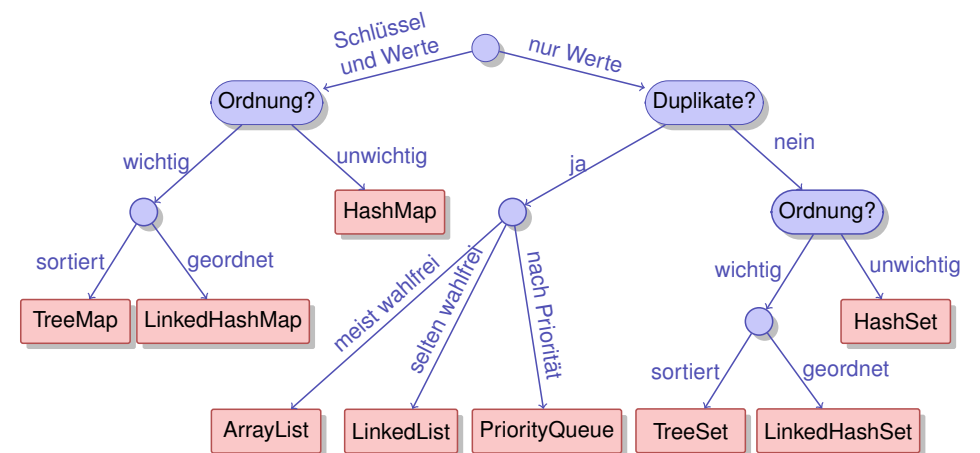
54

ArrayList versus LinkedList

Laufzeitmessungen für 10000 Operationen (auf [code]expert)

	ArrayList	LinkedList
Einfügen am Ende	469 μ s	1787 μ s
Einfügen am Anfang	37900 μ s	761 μ s
Iterieren	1840 μ s	2050 μ s
Wahlfreier Zugriff	426 μ s	110600 μ s
Einfügen in der Mitte	31ms	301ms
Enthält (erfolgreich)	38ms	141ms
Enthält (erfolglos)	228ms	1080ms
Entfernen am Ende	648 μ s	757 μ s
Entfernen am Anfang	58075 μ s	609 μ s

Erinnerung: Entscheidungshilfe



55

56

Asymptotische Laufzeiten

Mit unserer neuen Sprache (Ω , \mathcal{O} , Θ) können wir das *Verhalten der Datenstrukturen und ihrer Algorithmen präzisieren*.

Asymptotische Laufzeiten (Vorgriff!)

Datenstruktur	Wahlfreier Zugriff	Einfügen	Nächstes	Einfügen nach Element	Suchen
ArrayList	$\Theta(1)$	$\Theta(1) A$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
LinkedList	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
TreeSet	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
HashSet	–	$\Theta(1) P$	–	–	$\Theta(1) P$

A = amortisiert, P = erwartet, sonst schlechtester Fall („worst case“)

57

3. Suchen

Lineare Suche, Binäre Suche [Ottman/Widmayer, Kap. 3.2, Cormen et al, Kap. 2: Problems 2.1-3,2.2-3,2.3-5]

58

Das Suchproblem

Gegeben

- Menge von Datensätzen.

Beispiele

Telefonverzeichnis, Wörterbuch, Symboltabelle

- Jeder Datensatz hat einen Schlüssel k .
- Schlüssel sind vergleichbar: eindeutige Antwort auf Frage $k_1 \leq k_2$ für Schlüssel k_1, k_2 .

Aufgabe: finde Datensatz nach Schlüssel k .

59

Suche in Array

Gegeben

- Array A mit n Elementen ($A[1], \dots, A[n]$).
- Schlüssel b

Gesucht: Index k , $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

22	20	32	10	35	24	42	38	28	41
1	2	3	4	5	6	7	8	9	10

60

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- **Bestenfalls** 1 Vergleich.
- **Schlimmstenfalls** n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. **Erwartete** Anzahl Vergleiche für die erfolgreiche Suche:

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

Suche im sortierten Array

Gegeben

- Sortiertes Array A mit n Elementen ($A[1], \dots, A[n]$) mit $A[1] \leq A[2] \leq \dots \leq A[n]$.
- Schlüssel b

Gesucht: Index k , $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

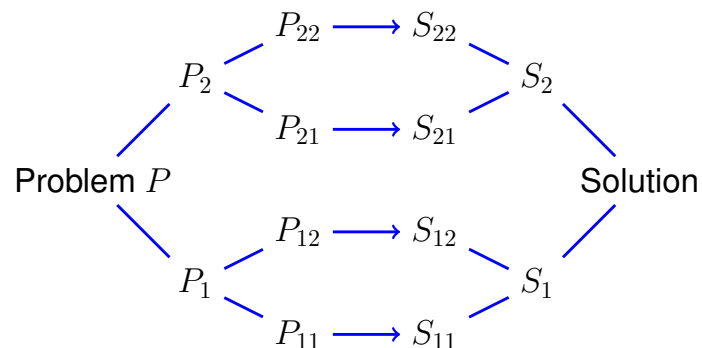
61

62

divide et impera

Teile und (be)herrsche (engl. divide and conquer)

Zerlege das Problem in Teilprobleme, deren Lösung zur vereinfachten Lösung des Gesamtproblems beitragen.



Divide and Conquer!

Suche $b = 23$.

10	20	22	24	28	32	35	38	41	42	$b < 28$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	$b > 20$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	$b > 22$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	$b < 24$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	erfolglos
1	2	3	4	5	6	7	8	9	10	

63

64

Binärer Suchalgorithmus $BSearch(A[l..r], b)$

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b . Bereichsgrenzen

$1 \leq l \leq r \leq n$ oder $l > r$ beliebig.

Output: Index des gefundenen Elements. 0, wenn erfolglos.

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if $l > r$ **then** // erfolglose Suche

return *NotFound*

else if $b = A[m]$ **then** // gefunden

return m

else if $b < A[m]$ **then** // Element liegt links

return $BSearch(A[l..m-1], b)$

else // $b > A[m]$: Element liegt rechts

return $BSearch(A[m+1..r], b)$

Analyse (schlechtester Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:²

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \dots \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n) \end{aligned}$$

²Versuche eine geschlossene Form zu finden, indem die Rekurrenz, ausgehend von $T(n)$, wiederholt eingesetzt wird.

65

66

Analyse (schlechtester Fall)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Vermutung : $T(n) = d + c \cdot \log_2 n$

Beweis durch Induktion:

■ Induktionsanfang: $T(1) = d$.

■ Hypothese: $T(n/2) = d + c \cdot \log_2 n/2$

■ Schritt ($n/2 \rightarrow n$)

$$T(n) = T(n/2) + c = d + c \cdot (\log_2 n - 1) + c = d + c \log_2 n.$$

67

68

Resultat

Theorem

Der Algorithmus zur binären sortierten Suche benötigt $\Theta(\log n)$ Elementarschritte.

Iterativer binärer Suchalgorithmus

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b .

Output: Index des gefundenen Elements. 0, wenn erfolglos.

$l \leftarrow 1; r \leftarrow n$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if $A[m] = b$ **then**

return m

else if $A[m] < b$ **then**

$l \leftarrow m + 1$

else

$r \leftarrow m - 1$

return *NotFound*;

4. Sortieren

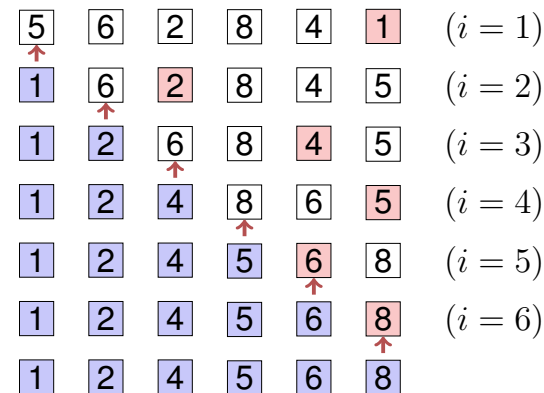
Einfache Sortierverfahren, Quicksort, Mergesort

Problemstellung

Eingabe: Ein Array $A = (A[1], \dots, A[n])$ der Länge n .

Ausgabe: Eine Permutation A' von A , die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.

Sortieren durch Auswahl



- Auswahl des kleinsten Elementes durch Suche im unsortierten Teil $A[i..n]$ des Arrays.
- Tausche kleinstes Element an das erste Element des unsortierten Teiles.
- Unsortierter Teil wird ein Element kleiner ($i \rightarrow i + 1$). Wiederhole bis alles sortiert. ($i = n$)

Algorithmus: Sortieren durch Auswahl

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output: Sortiertes Array A

```
for  $i \leftarrow 1$  to  $n - 1$  do
   $p \leftarrow i$ 
  for  $j \leftarrow i + 1$  to  $n$  do
    if  $A[j] < A[p]$  then
       $p \leftarrow j$ 
  swap( $A[i], A[p]$ )
```

4.1 Mergesort

[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$

73

74

Mergesort (Sortieren durch Verschmelzen)

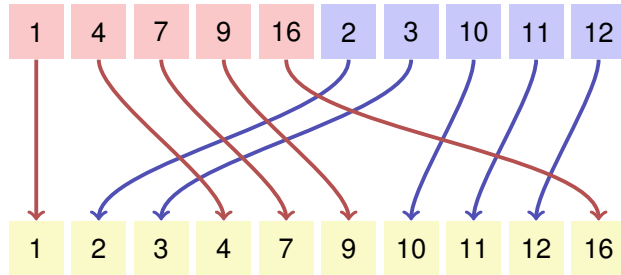
Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Füge die beiden vorsortierten Hälften von A zusammen in $\mathcal{O}(n)$.

75

76

Merge



Algorithmus Merge(A, l, m, r)

Input: Array A der Länge n , Indizes $1 \leq l \leq m \leq r \leq n$.
 $A[l, \dots, m], A[m + 1, \dots, r]$ sortiert

Output: $A[l, \dots, r]$ sortiert

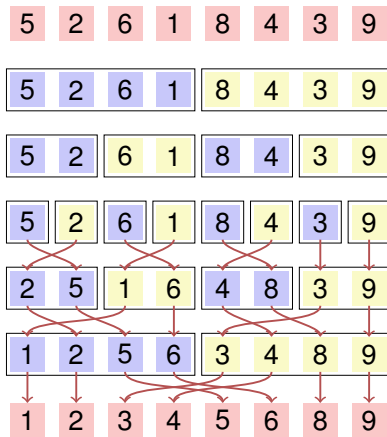
```

1  $B \leftarrow \text{new Array}(r - l + 1)$ 
2  $i \leftarrow l; j \leftarrow m + 1; k \leftarrow 1$ 
3 while  $i \leq m$  and  $j \leq r$  do
4   if  $A[i] \leq A[j]$  then  $B[k] \leftarrow A[i]; i \leftarrow i + 1$ 
5   else  $B[k] \leftarrow A[j]; j \leftarrow j + 1$ 
6    $k \leftarrow k + 1;$ 
7 while  $i \leq m$  do  $B[k] \leftarrow A[i]; i \leftarrow i + 1; k \leftarrow k + 1$ 
8 while  $j \leq r$  do  $B[k] \leftarrow A[j]; j \leftarrow j + 1; k \leftarrow k + 1$ 
9 for  $k \leftarrow l$  to  $r$  do  $A[k] \leftarrow B[k - l + 1]$ 
    
```

77

78

Mergesort



Split

Split

Split

Merge

Merge

Merge

Algorithmus (Rekursives 2-Wege) Mergesort(A, l, r)

Input: Array A der Länge n . $1 \leq l \leq r \leq n$

Output: Array $A[l, \dots, r]$ sortiert.

```

if  $l < r$  then
   $m \leftarrow \lfloor (l + r) / 2 \rfloor$  // Mittlere Position
  Mergesort( $A, l, m$ ) // Sortiere vordere Hälfte
  Mergesort( $A, m + 1, r$ ) // Sortiere hintere Hälfte
  Merge( $A, l, m, r$ ) // Verschmelzen der Teilfolgen
    
```

79

80

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

4.2 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Herleitung für $n = 2^k$

Sei $n = 2^k$, $k > 0$. Rekurrenz

$$T(n) = \begin{cases} d & \text{falls } n = 1 \\ 2T(n/2) + cn & \text{falls } n > 1 \end{cases}$$

Teleskopieren

$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn \\ &= 2(2(T(n/8) + cn/4) + cn/2) + cn = \dots \\ &= 2(2(\dots(2(2T(n/2^k) + cn/2^{k-1})\dots) + cn/2^2) + cn/2^1) + cn \\ &= 2^k T(1) + \underbrace{2^{k-1}cn/2^{k-1} + 2^{k-2}cn/2^{k-2} + \dots + 2^{k-k}cn/2^{k-k}}_{k \text{ Terme}} \\ &= nd + cnk = nd + cn \log_2 n \in \Theta(n \log n). \end{aligned}$$

81

82

Quicksort

❓ Was ist der Nachteil von Mergesort?

⚠️ Benötigt zusätzlich $\Theta(n)$ Speicherplatz für das Verschmelzen.

❓ Wie könnte man das Verschmelzen einsparen?

⚠️ Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

❓ Wie?

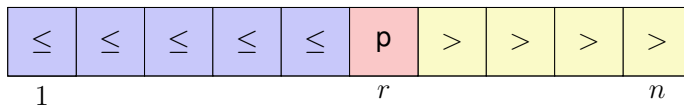
⚠️ Pivotieren und Aufteilen!

83

84

Pivotieren

- 1 Wähle ein (beliebiges) Element p als Pivotelement
- 2 Teile A in zwei Teile auf: einen Teil L der Elemente mit $A[i] \leq p$ und einen Teil R der Elemente mit $A[i] > p$.
- 3 Quicksort: Rekursion auf Teilen L und R



Algorithmus Partition($A[l..r], p$)

Input: Array A , welches den Pivot p im Intervall $[l, r]$ mindestens einmal enthält.

Output: Array A partitioniert in $[l..r]$ um p . Rückgabe der Position von p .

```

while  $l \leq r$  do
  while  $A[l] < p$  do
     $l \leftarrow l + 1$ 
  while  $A[r] > p$  do
     $r \leftarrow r - 1$ 
  swap( $A[l], A[r]$ )
  if  $A[l] = A[r]$  then
     $l \leftarrow l + 1$ 

```

return $l-1$

85

86

Algorithmus Quicksort($A[l, \dots, r]$)

Input: Array A der Länge n . $1 \leq l \leq r \leq n$.

Output: Array A , sortiert zwischen l und r .

if $l < r$ then

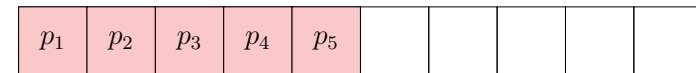
```

  Wähle Pivot  $p \in A[l, \dots, r]$ 
   $k \leftarrow$  Partition( $A[l, \dots, r], p$ )
  Quicksort( $A[l, \dots, k-1]$ )
  Quicksort( $A[k+1, \dots, r]$ )

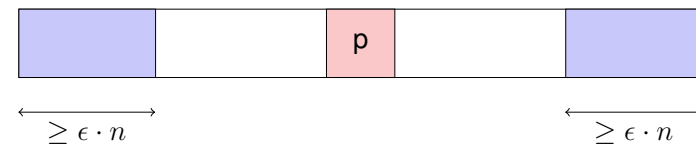
```

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$



Ein guter Pivot hat linear viele Elemente auf beiden Seiten.

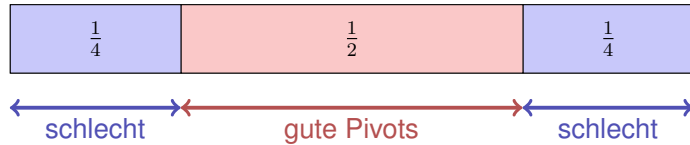


87

88

Wahl des Pivots?

Der Zufall hilft uns (Tony Hoare, 1961). Wähle in jedem Schritt einen zufälligen Pivot.



Wahrscheinlichkeit für guten Pivot nach einem Versuch: $\frac{1}{2} =: \rho$.

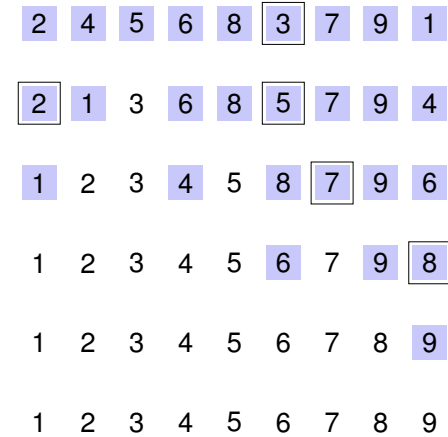
Wahrscheinlichkeit für guten Pivot nach k Versuchen: $(1 - \rho)^{k-1} \cdot \rho$.

Erwartete Anzahl Versuche³: $1/\rho = 2$

³Erwartungswert der geometrischen Verteilung:

89

Quicksort (willkürlicher Pivot)



90

Analyse: Anzahl Vergleiche

Schlechtester Fall. Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n - 1) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \Theta(n^2)$$

91

Analyse (Randomisiertes Quicksort)

Theorem

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log n)$ Vergleiche.

(ohne Beweis.)

92

Praktische Anmerkungen

- Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.

93

5. Von Java nach Python

Erstes Python Programm, Transfer Java → Python, Dynamische Datenstrukturen in Python

94

Learning Objectives

- see a new programming language (Python) and learn how to transfer from one programming language to another
- learn the most important differences between Java and Python, both from a syntactical and semantical point of view
- learn about the basic data types of Python (list, set, dict, tuple) and operations leveraging the use of such data types
- get used to the new programming language and environment (Python) by re-implementing known algorithms

95

Erstes Java Programm

```
public class Hello {  
    public static void main (String[] args) {  
        System.out.print("Hello World!");  
    }  
}
```

96

Erstes Python Programm

```
print("Hello World!")
```

97

Kommentare

Kommentare werden mit einem # vorangestellt:

```
# prints 'Hello World!' to the console  
print("Hello World!")
```

98

Formattierung ist Relevant: Anweisungen

- Whitespace (Space, Enter, Tab) ist relevant!
- Jede Zeile steht für eine Anweisung
- Also *genau eine* Anweisung pro Zeile
- Kommentare starten mit einem #

Beispiel-Programm mit zwei Anweisungen:

```
# two print-statements  
print("Hurray, finally ...")  
print("... no Semicolons!")
```

99

Formattierung ist Relevant: Blöcke

- Blöcke müssen eingerückt werden!
- Alle eingerückten Anweisungen gehören zum Block, Block endet, wo die Einrückung endet.
- Start eines Blocks ist markiert durch ein Doppelpunkt ":"

```
# in Python  
while i > 0:  
    x = x + 1 / i  
    i = i - 1  
  
print(x)
```

```
// in Java  
while (i > 0) {  
    x = x + 1.0 / i;  
    i = i - 1;  
}  
System.out.print(x)
```

100

Literale: Zahlen

- **integer:** 42, -5, 0x1b, 0o33, 7729684762313578932578932
Beliebig genaue Ganzzahlen
- **float:** -0.1, 34.567e-4
Wie **double** in Java, aber Präzision ist abhängig von der Plattform (CPU/Betriebssystem)
- **complex:** 2 + 3j, (0.21 - 1.2j)
Komplexe Zahlen in der Form $a+bj$. Optional mit runden Klammern.

101

Literale: Wahrheitswerte

- True
- False

102

Literale: Zeichenketten

- 'a single quoted string\nand a second line'
- "a double quoted string\nand a second line"
- Multi-line strings (triple double quotes):

```
"""a multiline string  
and a second line"""
```

103

Literale: Sequenzen

- **arrays:** Es gibt keine Arrays in Python
- **lists:** [17, True, "abc"], []
Mutierbare geordnete Sequenz von 0 oder mehr Werten mit beliebigen Typen.
- **tuples:** (17, True, "abc"), (42,)
Nicht-änderbare geordnete Sequenz von 1 oder mehr Werten mit beliebigen Typen.

104

Literale: Kollektionen

- **dicts:** { "a": 42, "b": 27, False: 0 }, {}
Mutierbarer Key-Wert Store. Keys und Werte mit beliebigen Typen.
- **sets:** {17, True, "abc"}, {42}
Mutierbare ungeordnete Sequenz von 0 oder mehr Werten mit beliebigen Typen. Keine Duplikate.

Variablen

- Variablen werden automatisch erstellt bei der ersten Zuweisung
- Der Typ einer Variable wird bei Wertzuweisungen nicht geprüft. D.h. es können Werte von unterschiedlichen Typen einer Variable zugewiesen werden über Zeit.
- Zuweisung von Werten mittels Zuweisungsoperator: =
- Zuweisung an mehrere Variablen gleichzeitig mit Tupeln

```
a = "Ein Text"
print(a) # prints: Ein Text
a = 42
print(a) # prints: 42
```

```
x, y = 4, 5
print(x) # prints: 4
print(y) # prints: 5
```

105

106

Variablen

- Variablen müssen immer erst zugewiesen werden, bevor diese ausgelesen werden können.

Angenommen, b wurde noch nie ein Wert zugewiesen.

```
a = b
```

Gibt folgenden Fehler

NameError: name 'b' is not defined

Numerische und Boolean Operatoren

- Numerische Operatoren wie bei Java: +, -, *, /, %, **, //
- Achtung: "/" resultiert immer in einer Fließkommazahl
- **: Potenzierung, $a**b = a^b$.
- //: Ganzzahl-Division, $5//2$ results in 2.5.
- Vergleichsoperatoren wie bei Java: ==, >=, <=, >, <, !=
- Logische Operatoren: and, or, not
- Membership Operator: "in" Gibt an, ob ein Wert in einer Liste, einem Set oder einer Zeichenkette ist.
- Identitäts Operator: "is" Prüft, ob zwei Variablen auf das gleiche Objekt zeigen.

107

108

Eingabe/Ausgabe

- Lesen von Eingaben mittels `input()`
- Uebergabe eines Prompts möglich.
- Ausgabe mittels `print(...)`
- `print` nimmt eine oder mehrere Argumente entgegen und druckt diese mit einem Leerschlag dazwischen

```
name = input("What is your name: ")
print("Hello", name)
```

109

Eingabe/Ausgabe

- Eingabe wird immer als Zeichenkette gelesen.
- Um eine Zahl einzulesen, muss die Eingabe in eine Zahl konvertiert werden
- Es finden keine impliziten Konvertierungen statt
- Explizite Konvertierung mittels:
`int()`, `float()`, `complex()`, `list()`, ...

```
i = int(input("Enter a number: "))
print("The", i, "th power of two is", 2**i)
```

110

Bedingungen

- Keine Klammern nötig um den Test
- `elif` um einen weiteren Fall zu testen
- Einrückung beachten!

```
a = int(input("Enter a number: "))
if a == 42:
    print("Naturally, the answer")
elif a == 28:
    print("A perfect number, good choice")
else:
    print(a, "is just some boring number")
```

111

While-Schleifen

Die bekannte Collaz-Folge

```
a = int(input("Enter a number: "))
while a != 1:
    if a % 2 == 0:
        a = a // 2
    else:
        a = a * 3 + 1
    print(a, end=' ')

```

112

For-Schleifen

- For-Schleifen funktionieren anders als in Java
- Iteriert über die Elemente der angegebenen Menge

```
some_list = [14, 'lala', 22, True, 6]
total = 0;
for item in some_list:
    if type(item) == int:
        total += item
print("Total of the numbers is", total)
```

113

For-Schleifen über einen Wertebereich

- Die Funktion `range(start, end, step)` erstellt eine Liste von Werten, beginnend mit `start`, bis zu `end` - exklusive. Schrittgrösse `step`.
- Schrittgrösse ist 1, wenn das dritte Argument weggelassen wird.

```
# the following loop prints "1 2 3 4"
for i in range(1,5):
    print(i, end=' ')
```

```
# the following loop prints "10 8 6 4 2"
for i in range(10, 0, -2):
    print(i, end=' ')
```

114

Methoden

- Der Keks-Rechner revisited

```
def readInt(prompt, atleast = 1):
    """Prompt for a number greater 0 (or min, if specified)"""
    number = 0;
    while number < atleast:
        number = int(input(prompt))
        if (number < atleast):
            print("Too small, pick a number larger than", atleast)
    return number
```

```
kids = readInt("Kids: ")
cookies = readInt("Cookies: ", atleast=kids)
print("Each Kid gets", cookies // kids, "cookies.")
print("Papa gets", cookies % kids, "cookies.")
```

115

Listen: Basisoperationen

- Elementzugriff (0-basiert): `a[2]` zeigt auf das dritte Element.
- Negative Indizes zählen von hinten!

```
a = [ 3, 7, 4]
print(a[-1]) # prints '4'
```

- Wert hinten hinzufügen: `a.append(12)`
- Test ob ein Element in einer Collection ist:

```
if 12 in a:
    print('12 is in the list, we just added it before')
```

- Anzahl Elemente in einer Collection: `len(a)`

116

Listen: Slicing

- Slicing: Teilbereiche adressieren: `a[start:end]`
- `a` und/oder `b` sind positive oder negative Indizes.
- `end` ist nicht inklusive

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[2:4])    # [3, 4]
print(a[3:-3])  # [4, 5, 6]
print(a[-3:-1]) # [7, 8]
print(a[5:])     # [6, 7, 8, 9]
print(a[:3])    # [1, 2, 3]
```

117

Dictionaries

Dictionaries sind sehr wichtige primitive Datenstrukturen in Python

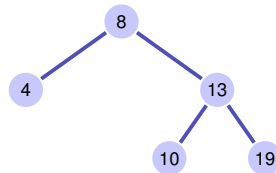
- Einfache und effiziente Möglichkeit, mehrere Datenfelder zu benennen und zusammenzufassen
- Aufbau von hierarchischen Datenstrukturen durch Verschachtelung
- Zugriff auf Elemente mittels `[]` Operator

```
record = { 'Firstname': 'Hermann', 'Lastname': 'Lehner',
           'Salary': 420000, 'Mac User': True }
record['Salary'] = 450000
if record['Mac User']:
    print('... one more thing!')
```

118

Dynamische Datenstrukturen mit Dicts

```
tree = {
    'key': 8,
    'left': {
        'key': 4, 'left': None, 'right': None
    },
    'right': {
        'key': 13,
        'left': {
            'key': 10, 'left': None, 'right': None
        },
        'right': {
            'key': 19, 'left': None, 'right': None
        }
    }
}
```



119

Dynamische Datenstrukturen mit Dicts

- Arbeiten mit Dicts (Beispiele)

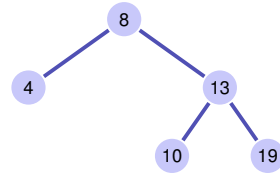
```
l = tree['left'] # assign left subtree to variable l
l['key'] = 6     # changes key from 4 to 6

if l['left'] is None: # proper way to test against None
    print("There is no left child here...")
else:
    print("Value of left subtree is", l['left']['key'])
```

120

Dynamische Datenstrukturen mit Klassen

```
class Node:
    def __init__(self, k, l=None, r=None):
        self.key, self.left, self.right = k, l, r
```



```
# create the tree depicted on the right
rightSubtree = Node(13, l=Node(10), r=Node(19))
tree = Node(8, l=Node(4), r=rightSubtree)
```

```
# an example query
print(tree.right.right.key) # prints: 19
```

121

Module

Python hat eine extrem grosse Anzahl Libraries in Form von Modulen, die importiert werden können.

■ Importieren eines ganzen Modules:

```
import math
x = math.sqrt(4)
```

```
from math import *
x = sqrt(4)
```

■ Importieren von Teilen eines Modules:

```
from datetime import date
t = date.today()
```

122

6. Weiterführende Python Konzepte

Eingebaute Funktionen, Bedingte Ausdrücke, List und Dict Comprehension, File IO, Fehlerbehandlung

Eingebaute Funktionen: Aufzählungen mit Index

Manchmal möchte man durch Listen iterieren, inklusive Index für jedes Element. Dies geht mit `enumerate(...)`

```
data = [ 'Spam', 'Eggs', 'Ham' ]

for index, value in enumerate(data):
    print(index, ":", value)
```

Output:

```
0 : Spam
1 : Eggs
2 : Ham
```

123

124

Eingebaute Funktionen: Listen kombinieren

Es gibt eine einfache Möglichkeit, Listen element-weise zusammenzuführen (wie ein Reißverschluss!): `zip(...)`

```
places = [ 'Zurich', 'Basel', 'Bern' ]
plz = [ 8000, 4000, 3000, ]

list(zip(places, plz))
# [('Zurich', 8000), ('Basel', 4000), ('Bern', 3000)]

dict(zip(places, plz))
# {'Zurich': 8000, 'Basel': 4000, 'Bern': 3000}
```

125

Bedingte Ausdrücke

In Python kann der Wert eines Ausdrucks von einer Bedingung abhängen (als Teil des Ausdrucks!)

Beispiel: Collaz Folge

```
while a != 1:
    a = a // 2 if a % 2 == 0 else a * 3 + 1
```

Beispiel: Textformatierung

```
print('I see', n, 'mouse' if n == 1 else 'mice')
```

126

List Comprehension

- Python bietet eine sehr angenehme Möglichkeit an, Listen deklarativ zu erstellen
- Ähnliche Denkweise wie bei 'map' und 'filter' bei funktionalen Sprachen

Beispiel: Eine Sequenz von Zahlen einlesen

```
line = input('Enter some numbers: ')
s_list = line.split()
n_list = [ int(x) for x in s_list ]
```

Das selbe kombiniert in einem Ausdruck

```
n_list = [ int(x) for x in input('Enter some numbers: ').split() ]
```

127

List Comprehension

Beispiel: Leerschläge vorne und hinten eliminieren

```
line = [ ' some eggs ', ' slice of ham ', ' a lot of spam ' ]
cleaned = [ item.strip() for item in line ]
```

```
# cleaned == [ 'some eggs', 'slice of ham', 'a lot of spam' ]
```

128

Dict Comprehension

- Wie bei Listen, aber mit key/value Paaren

Beispiel: Daten extrahieren aus einem Dict

```
data = {
    'Spam' : { 'Amount' : 12, 'Price': 0.45 },
    'Eggs' : { 'Price': 0.8 },
    'Ham'  : { 'Amount': 5, 'Price': 1.20 }
}

total_prices = { item : record['Amount'] * record['Price']
                 for item, record in data.items()
                 if 'Amount' in record }

# total_prices == {'Spam': 5.4, 'Ham': 6.0}
```

129

File IO

- Files können mit dem Befehl `open` geöffnet werden.
- Damit Files automatisch wieder geschlossen werden, muss das innerhalb eines `with` Blocks geschehen.

Beispiel: CSV File einlesen

```
import csv

with open('times.csv', mode='r') as csv_file:
    csv_lines = csv.reader(csv_file)
    for line in csv_lines:
        # do something for each record
```

Schreiben geht ähnlich. Siehe Python Doku.

130

Ausnahmebehandlung

Gegeben folgender Code:

```
x = int(input('A number please: '))
```

Wenn keine Zahl eingegeben wird, stürzt das Programm ab:

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    x = int(input('A number please: '))
ValueError: invalid literal for int() with base 10: 'a'
```

Wir können diesen Fehler auffangen und entsprechend reagieren.

Ausnahmebehandlung

```
try:
    x = int(input('A number please: '))
except ValueError:
    print('Oh boy, that was no number...')
    x = 0
print('x:', x)
```

Ausgabe, wenn `'spam'` eingegeben wird statt einer Zahl:

```
Oh boy, that was no number...
x: 0
```

131

132

7. Java Input/Output

Benutzereingaben/Konsolenausgabe, Datei Ein- und Ausgabe (I/O)

Benutzereingaben (die halbe Wahrheit)

- z.B. einer Zahl: `int i = In.readInt();`
- Unsere `In` Klasse stellt diverse solche Methoden zur Verfügung.
- Manche dieser Methoden müssen mit falschen Eingaben umgehen können: Was passiert mit `readInt()` bei folgendem Input?

"spam"

133

134

Benutzereingaben (die halbe Wahrheit)

```
public class Main {
    public static void main(String[] args) {
        Out.print("Number: ");
        int i = In.readInt();
        Out.print("Your number: " + i);
    }
}
```

Anscheinend passiert nicht viel!

```
Number: spam
Your number: 0
```

135

Benutzereingaben (die ganze Wahrheit)

- z.B. einer Zahl, mittels der Klasse `Scanner`

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Out.print("Number: ");
        Scanner input = new Scanner(System.in);
        int i = input.nextInt();
        Out.print("Your number: " + i);
    }
}
```

Was passiert bei folgendem Input?

"spam"

136

Benutzereingaben (die ganze Wahrheit)

Number: spam

```
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Main.main(Main.java:7)
    at TestRunner.main(TestRunner.java:330)
```

Oh! Wir schauen uns das im naechsten Kapitel an...

Konsolenausgabe

- Bisher kannten Sie: `Out.print("Hi")` oder `Out.println("Hi")`
- Ohne unsere Out Klasse:

```
System.out.print("The answer is: ");
System.out.println(42);
System.out.println("What was the question?!");
```

Dies ergibt die folgende Ausgabe:

```
The answer is: 42
What was the question?!
```

137

138

Also: Benutzereingabe/Konsolenausgabe

- Lesen von Eingaben via dem Eingabestrom `System.in`
- Schreiben von Ausgaben via dem Ausgabestrom `System.out`

139

Dateien (zeilenweise) Lesen/Schreiben

- Dateien können mittels der Klasse `java.io.FileReader` byte für byte gelesen werden
- Um ganze Zeilen zu lesen, verwenden wir ausserdem einen `java.io.BufferedReader`
- Dateien können mittels der Klasse `java.io.FileWriter` byte für byte geschrieben werden
- Um ganze Zeilen zu schreiben, verwenden wir ausserdem einen `java.io.BufferedWriter`

140

Dateien (zeilenweise) Lesen

```
import java.io.FileReader;
import java.io.BufferedReader;

public class Main {
    public static void main(String[] args) {
        FileReader fr = new FileReader("gedicht.txt");
        BufferedReader bufr = new BufferedReader(fr);
        String line;
        while ((line = bufr.readLine()) != null){
            System.out.println(line);
        }
    }
}
```

141

... deshalb nun ...

Dateien (zeilenweise) Lesen

Wir erhalten folgenden Kompilier-Fehler:

```
./Main.java:6: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
    FileReader fr = new FileReader("gedicht.txt");
                        ^
./Main.java:9: error: unreported exception IOException; must be
caught or declared to be thrown
    while ((line = bufr.readLine()) != null){
                        ^
```

2 errors

Offensichtlich müssen wir mehr verstehen zum Thema "Exceptions"

142

8. Fehler und Ausnahmen

Fehler, Systemausnahmen, Benutzerausnahmen, Behandeln von Ausnahmen, Spezialfall Ressourcen

143

144

Fehler und Ausnahmen in Java

Fehler und Ausnahmen unterbrechen die normale Programmausführung abrupt und stellen ein *nicht geplantes Ereignis* dar.



Ausnahmen sind böse, oder doch nicht?

- Java ermöglicht es, solche Ereignisse abzufangen und zu behandeln (als Alternative zum Programmabsturz).
- Nicht behandelte Fehler und Ausnahmen werden durch den Aufrufstapel hochgereicht.

145

Fehler (Errors)



Hier ist nichts mehr zu machen

Fehler treten in der virtuellen Maschine von Java auf und sind *nicht reparierbar*.

Beispiele

- Kein Speicher mehr verfügbar
- Zu hoher Aufrufstapel (→ Rekursion)
- Fehlende Programmbibliotheken
- Bug in der virtuellen Maschine
- Hardwarefehler

146

Ausnahmen (Exceptions)

Ausnahmen werden von der virtuellen Maschine oder vom Programm selbst ausgelöst und können meist behandelt werden um die *Normalsituation wiederherzustellen*



Aufwischen und neu einschenken

Beispiele

- Dereferenzierung von `null`
- Division durch 0
- Schreib/Lesefehler (Dateien)
- Businesslogik Fehler

147

Arten von Ausnahmen

Systemausnahmen
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

Benutzerausnahmen
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

148

Beispiel einer Systemausnahme

```
1 import java.util.Scanner;
2 class ReadTest {
3     public static void main(String[] args){
4         int i = readInt("Number");
5     }
6     private static int readInt(String prompt){
7         System.out.print(prompt + ": ");
8         Scanner input = new Scanner(System.in);
9         return input.nextInt();
10    }
11 }
```

Eingabe: `Number: asdf`

149

Nicht behandelte Fehler und Ausnahmen

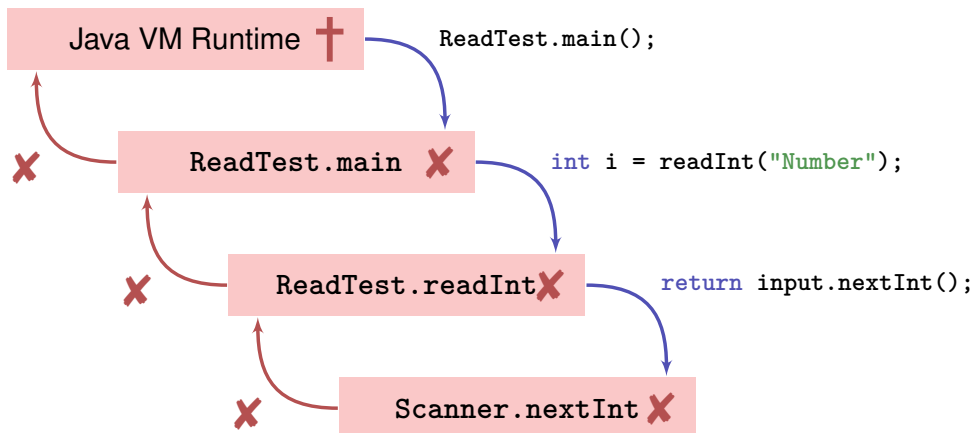
Das Programm stürzt ab und hinterlässt auf der Konsole eine "Aufrufstapelzurückverfolgung" 😊 (ab jetzt: *Stacktrace*). Darin sehen wir, wo genau das Programm abgebrochen wurde.

```
Exception in thread "main" java.util.InputMismatchException
    [...]
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at ReadTest.readInt(ReadTest.java:9)
    at ReadTest.main(ReadTest.java:4)
```

⇒ Forensische Nachforschungen mit Hilfe dieser Information.

150

Ausnahme propagiert durch Aufrufstapel



151

Stacktraces verstehen

Ausgabe:

```
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at ReadTest.readInt(ReadTest.java:9)
    at ReadTest.main(ReadTest.java:4)
```

Eine unpassende Eingabe ...

... in Methode readInt auf Zeile 9 ...

aufgerufen durch Methode main auf Zeile 4

152

Stacktraces verstehen

```
1 import java.util.Scanner;
2 class ReadTest {
3     public static void main(String[] args){
4         int i = readInt("Number");
5     }
6     private static int readInt(String prompt){
7         System.out.print(prompt + ": ");
8         Scanner input = new Scanner(System.in);
9         return input.nextInt();
10    }
11 }
```

at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)

153

Systemausnahme: Bug im Programm?!

Wo ist der Fehler?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

Nicht garantiert, dass als nächstes ein int anliegt.

⇒ Die Scanner Klasse bietet ein Test dafür an

154

Systemausnahme: Bugfix!

Erst prüfen!

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    if (input.hasNextInt()){
        return input.nextInt();
    } else {
        return 0; // or do something else ...?!
    }
}
```

155

Erste Erkenntnis: Oft keine Ausnahmesituation

Oft sind die "Sonderfälle" gar kein besonderes Ereignis, sondern absehbar. Hier sollten *keine* Ausnahmen verwendet werden!



Kinder kippen Becher um.
Man gewöhnt sich daran.

Beispiele

- Falsche Credentials beim Einloggen
- Leere Pflichtfelder in Eingabemasken
- Nicht verfügbare Internet-Ressourcen
- Timeouts

156

Zweite Erkenntnis: Ausnahmen verhindern



Problem gelöst.

Statt eine Systemausnahme abzuwarten *aktiv verhindern*, dass diese überhaupt auftreten kann.

Beispiele

- Usereingaben frühzeitig prüfen
- Optionale Typen verwenden
- Timeout Situationen voraussehen
- Plan B für nicht verfügbare Ressourcen

157

Arten von Ausnahmen

Systemausnahmen
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

Benutzerausnahmen
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

158

Beispiel einer Benutzerausnahme

```
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

Compiler Fehler:

```
./Root/Main.java:9: error: unreported exception FileNotFoundException; must be caught or declared to be
    FileReader fr = new FileReader(filename);
    ~
./Root/Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
    String line = bufr.readLine();
    ~
```

159

Kurzer Blick in die Javadoc

readLine

```
public String readLine()
    throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

See Also:

Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)

160

Warum eine Benutzerausnahme?

Folgende Situationen rechtfertigen Benutzerausnahmen:

- Fehlerfall ist *unwahrscheinlich aber prinzipiell möglich* – und kann durch geeignete Massnahmen zur Laufzeit behoben werden können.

Der Aufrufer einer Methode mit einer deklarierten Benutzerausnahme wird gezwungen, sich damit zu beschäftigen – behandeln oder weiterreichen.

Behandeln von Ausnahmen

```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

← *Geschützter Bereich*

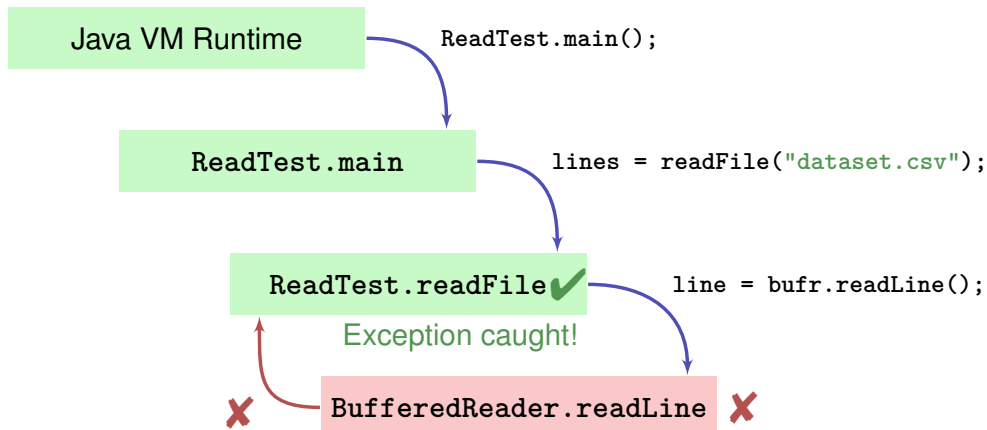
← *Massnahmen zur Wiederherstellung der Normalsituation*

← *Wird in jedem Fall am Schluss ausgeführt, immer!*

161

162

Behandlung von Ausnahmen: Propagieren stoppen!



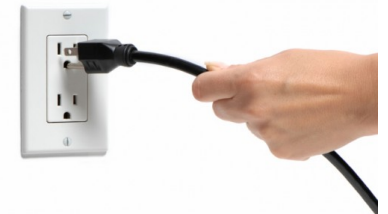
163

Finally: Ressourcen schliessen!

In Java müssen *Ressourcen* unbedingt geschlossen werden nach Gebrauch. Ansonsten wird Speicher nicht freigegeben.

Ressourcen:

- Dateien
- Datenströme
- GUI Elemente
- ...



164

Try-With-Resources Anweisung

Spezifische Syntax, um Ressourcen *automatisch* zu schliessen:

```
private static String[] readFile(String filename){
    try (FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

Resources werden hier geöffnet

Resources werden hier automatisch geschlossen

165

9. Hashing

Hashtabellen, Pre-Hashing, Hashing, Kollisionsauflösung durch Verkettung, Einfaches gleichmässiges Hashing, Gebräuchliche Hashfunktionen, Tabellenvergrößerung, offene Addressierung: Sondieren [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

166

Motivierendes Beispiel

Ziel: Effiziente Verwaltung einer Tabelle aller n ETH-Studenten

Mögliche Anforderung: Schneller Zugriff (Einfügen, Löschen, Finden) von Datensätzen nach Name.

Wörterbuch (Dictionary)

Abstrakter Datentyp (ADT) D zur Verwaltung einer Menge von Einträgen⁴ i mit Schlüsseln $k \in \mathcal{K}$. Operationen

- $D.insert(i)$: Hinzufügen oder Überschreiben von i im Wörterbuch D .
- $D.delete(i)$: Löschen von i aus dem Wörterbuch D . Nicht vorhanden \Rightarrow Fehlermeldung.
- $D.search(k)$: Liefert Eintrag mit Schlüssel k , wenn er existiert.

⁴Schlüssel-Wert Paare (k, v) , im Folgenden betrachten wir hauptsächlich die Schlüssel.

167

168

Wörterbücher in Python

```
Wörterbuch → fruits = {  
    "banana": 2.95, "kiwi": 0.70,  
    "pear": 4.20, "apple": 3.95  
}
```

```
Einfügen → fruits["melon"] = 3.95  
Verändern → fruits["banana"] = 1.90  
Suchen → print("banana", fruits["banana"])  
         print("melon in fruits", "melon" in  
         fruits) print("onion in fruits"  
         , "onion" in fruits)  
Löschen → del fruits["strawberry"]  
Iterieren → for name, price in fruits.items():  
             print(name, "→", price)
```

169

Wörterbücher in Java

```
Wörterbuch → Map<String, Double> fruits =  
             new HashMap<String, Double>();
```

```
Einfügen → fruits.put("banana", 2.95);  
          fruits.put("kiwi", 0.70);  
          fruits.put("strawberry", 9.95);  
          fruits.put("pear", 4.20);  
          fruits.put("apple", 3.95);  
Verändern → fruits.put("banana", 2.90);  
Suchen → Out.println("banana " + fruits.get("banana"));  
Löschen → fruits.remove("banana");  
Iterieren → for (String s: fruits.keySet())  
            Out.println(s + " " + fruits.get(s));
```

170

Motivation / Verwendung

Wahrscheinlich *die* gängigste Datenstruktur

- Unterstützt in vielen Programmiersprachen (C++, Java, Python, Ruby, Javascript, C# ...)
- Offensichtliche Verwendung
 - Datenbanken / Tabellenkalkulation
 - Symboltabellen in Compilern und Interpretern
- Weniger offensichtlich
 - Substring Suche (Google, grep)
 - Ähnlichkeit von Texten (Dokumentenvergleich, DNA)
 - Dateisynchronisation
 - Kryptographie: Filetransfer / Identifikation

171

1. Idee: Direkter Zugriff (Array)

Index	Eintrag
0	-
1	-
2	-
3	[3, wert(3)]
4	-
5	-
⋮	⋮
k	[k, wert(k)]
⋮	⋮

Probleme

- 1 Schlüssel müssen nichtnegative ganze Zahlen sein
- 2 Grosser Schlüsselbereich ⇒ grosses Array

172

Lösung zum ersten Problem: Pre-hashing

Prehashing: Bilde Schlüssel ab auf positive Ganzzahlen mit einer Funktion $ph : \mathcal{K} \rightarrow \mathbb{N}$

- Theoretisch immer möglich, denn jeder Schlüssel ist als Bitsequenz im Computer gespeichert
- Theoretisch auch: $x = y \Leftrightarrow ph(x) = ph(y)$
- In der Praxis: APIs bieten Funktionen zum pre-hashing an. (Java: `object.hashCode()`, C++: `std::hash<>`, Python: `hash(object)`)
- APIs bilden einen Schlüssel aus der Schlüsselmenge ab auf eine Ganzzahl mit beschränkter Grösse.⁵

⁵Somit gilt die Implikation $ph(x) = ph(y) \Rightarrow x = y$ nicht mehr für alle x, y .

173

Prehashing Beispiel: String

Zuordnung Name $s = s_1s_2 \dots s_{l_s}$ zu Schlüssel

$$ph(s) = \left(\sum_{i=1}^{l_s} s_{l_s-i+1} \cdot b^i \right) \bmod 2^w$$

b so, dass verschiedene Namen möglichst verschiedene Schlüssel erhalten.

w Wortgrösse des Systems (z.B. 32 oder 64).

Beispiel (Java), mit $b = 31$, $w = 32$ Ascii-Werte s_i .

Anna \mapsto 2045632

Jacqueline \mapsto 2042089953442505 $\bmod 2^{32} = 507919049$

174

Implementation Prehashing (String) in Java

$$ph_{b,m}(s) = \left(\sum_{i=0}^{l-1} s_{l-i} \cdot b^i \right) \bmod m$$

Mit $b = 31$ und $m = 2^{32}$ erhält man in Java⁶

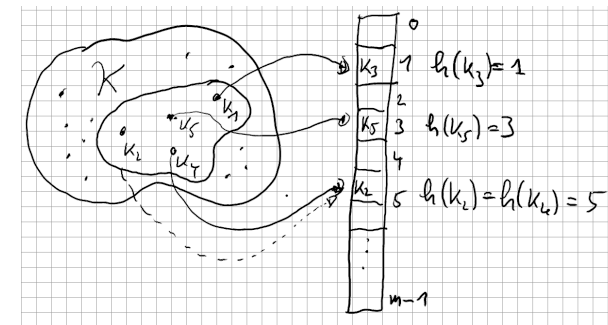
```
int prehash(String s){
    int h = 0;
    for (int k = 0; k < s.length(); ++k){
        h = h * b + s.charAt(k);
    }
    return h;
}
```

⁶Machen Sie sich klar, warum das funktioniert

175

Lösung zum zweiten Problem: Hashing

Reduziere des Schlüsseluniversum: Abbildung (Hash-Funktion) $h : \mathcal{K} \rightarrow \{0, \dots, m-1\}$ ($m \approx n =$ Anzahl Einträge in der Tabelle)



Kollision: $h(k_i) = h(k_j)$.

176

Nomenklatur

Hashfunktion h : Abbildung aus der Menge der Schlüssel \mathcal{K} auf die Indexmenge $\{0, 1, \dots, m - 1\}$ eines Arrays (**Hashtabelle**).

$$h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

Meist $|\mathcal{K}| \gg m$. Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (**Kollision**).

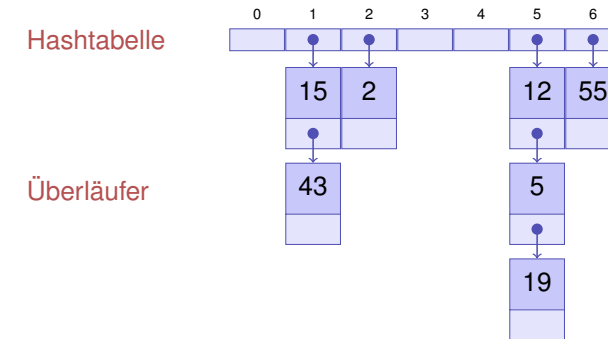
Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

Behandlung von Kollisionen: Verkettung

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Schlüssel 12, 55, 5, 15, 2, 19, 43

Direkte Verkettung der Überläufer



177

178

Algorithmen zum Hashing mit Verkettung

- **insert**(i) Prüfe ob Schlüssel k vom Eintrag i in Liste an Position $h(k)$. Falls nein, füge i am Ende der Liste ein; andernfalls ersetze das Element durch i .
- **find**(k) Prüfe ob Schlüssel k in Liste an Position $h(k)$. Falls ja, gib die Daten zum Schlüssel k zurück. Andernfalls Rückgabe eines leeren Elements **null**.
- **delete**(k) Durchsuche die Liste an Position $h(k)$ nach k . Wenn Suche erfolgreich, entferne das entsprechende Listenelement.

Worst-case Analyse

Schlechtester Fall: alle Schlüssel werden auf den gleichen Index abgebildet.

⇒ $\Theta(n)$ pro Operation im schlechtesten Fall. 😞

179

180

Einfaches Gleichmässiges Hashing

Starke Annahmen: Jeder beliebige Schlüssel wird

- mit gleicher Wahrscheinlichkeit (Uniformität)
- und unabhängig von den anderen Schlüsseln (Unabhängigkeit)

auf einen der m verfügbaren Slots abgebildet.

Einfaches Gleichmässiges Hashing

Unter der Voraussetzung von einfachem gleichmässigem Hashing:
Erwartete Länge einer Kette, wenn n Elemente in eine Hashtabelle mit m Elementen eingefügt werden

$$\begin{aligned}\mathbb{E}(\text{Länge Kette } j) &= \mathbb{E}\left(\sum_{i=0}^{n-1} \mathbb{1}(k_i = j)\right) = \sum_{i=0}^{n-1} \mathbb{P}(k_i = j) \\ &= \sum_{i=1}^n \frac{1}{m} = \frac{n}{m}\end{aligned}$$

$\alpha = n/m$ heisst **Belegungsfaktor** oder **Füllgrad** der Hashtabelle.

181

182

Einfaches Gleichmässiges Hashing

Theorem

Sei eine Hashtabelle Verkettung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$.
Unter der Annahme vom einfachen gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von $\leq 1 + \alpha$.

Folgerung: ist die Anzahl der Slots m der Hashtabelle immer mindestens proportional zur Anzahl Elemente n in der Hashtabelle, $n \in \mathcal{O}(m) \Rightarrow$ Erwartete Laufzeit der Operationen Suchen, Einfügen und Löschen ist $\mathcal{O}(1)$.

Vor und Nachteile der Verkettung

Vorteile der Strategie:

- Belegungsfaktoren $\alpha > 1$ möglich
- Entfernen von Schlüsseln einfach

Nachteile

- Speicherverbrauch der Verkettung

183

184

Ein Beispiel einer gebräuchlichen Hashfunktion

Divisionsmethode

$$h(k) = k \bmod m$$

Ideal: m Primzahl, nicht zu nahe bei Potenzen von 2 oder 10

Aber oft: $m = 2^k - 1$ ($k \in \mathbb{N}$)

Andere Methode: Multiplikationsmethode (siehe Cormen et al, Kap. 11.3).

185

Tabellenvergrößerung

- Wissen nicht a priori, wie gross n sein wird.
- Benötigen $m = \Theta(n)$ zu jeder Zeit.

Grösse der Tabelle muss angepasst werden. Hash-Funktion ändert sich \Rightarrow **Rehashing**

- Alloziere Array A' mit Grösse $m' > m$
- Füge jeden Eintrag von A erneut in A' ein (mit erneutem Hashing)
- Setze $A \leftarrow A'$.
- Kosten: $\mathcal{O}(n + m + m')$.

Wie wählt man m' ?

186

Tabellenvergrößerung

- 1. Idee $n = m \Rightarrow m' \leftarrow m + 1$
Bei jedem Einfügen vergrössern. Kosten
 $\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2)$ 😞
- 2. Idee $n = m \Rightarrow m' \leftarrow 2m$ Vergrössern nur wenn $m = 2^i$:
 $\Theta(1 + 2 + 4 + 8 + \dots + n) = \Theta(n)$
Einige Einfügeoperationen kosten lineare Zeit, aber im
Durchschnitt kosten sie $\Theta(1)$ 😊

Jede Operation vom Hashing mit Verketteten hat erwartet amortisierte Kosten $\Theta(1)$.

(\Rightarrow Amortisierte Analyse)

187

Amortisierte Analyse

Generelles Vorgehen bei dynamischen Arrays (z.B. Java: `ArrayList`, Python: `List`)

- Die Datenstruktur speichert neben dem Datenarray zwei Grössen: Grösse des Arrays (Kapazität m) und Anzahl verwendete Einträge (Grösse n).
- Grösse verdoppeln und Einträge kopieren, wenn die Liste voll ist
 $n = m \Rightarrow m \leftarrow 2n$. Kosten $\Theta(m)$.
- Laufzeitkosten bei $n = 2^k$ Einfügeoperationen:
 $\Theta(1 + 2 + 4 + 8 + \dots + 2^k) = \Theta(2^{k+1} - 1) = \Theta(n)$.

Kosten pro Operation **gemittelt über alle Operationen** = **amortisierte Kosten** = $\Theta(1)$ pro Einfügeoperation.

188

Offene Addressierung⁷

Speichere die Überläufer direkt in der Hashtabelle mit einer **Sondierungsfunktion** $s : \mathcal{K} \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$ Tabellenposition des Schlüssels entlang der **Sondierungsfolge**

$$S(k) := (s(k, 0), s(k, 1), \dots, s(k, m - 1)) \pmod{m}$$

Sondierungsfolge muss für jedes $k \in \mathcal{K}$ eine Permutation sein von $\{0, 1, \dots, m - 1\}$

⁷Begriffsklärung: Dieses Verfahren nutzt **offene Addressierung** (Positionen in der Hashtabelle nicht fixiert), ist aber **ein geschlossenes Hashverfahren** (Einträge bleiben in der Hashtabelle)

189

Algorithmen zur offenen Addressierung

- **insert**(i) Suche Schlüssel k von i in der Tabelle gemäss Sondierungssequenz $S(k)$. Ist k nicht vorhanden, füge k an die erste freie Position in der Sondierungsfolge ein. Andernfalls Fehlermeldung.
- **find**(k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib die zu k gehörenden Daten zurück. Andernfalls Rückgabe eines leeres Elements **null**.
- **delete**(k) Suche k in der Tabelle gemäss $S(k)$. Wenn k gefunden, ersetze k durch den speziellen Schlüssel **removed**.

190

Lineares Sondieren

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \pmod{m}$.
Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

191

Diskussion

Beispiel $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 200 Tabelleneinträge! (hier ohne Herleitung).

❓ Grund für die schlechte Performance?

⚠️ **Primäre Häufung**: Ähnliche Hashadressen haben ähnliche Sondierungsfolgen \Rightarrow lange zusammenhängende belegte Bereiche.

192

Quadratisches Sondieren

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \pmod m$.
Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
19	15	2		5	12	55

193

Diskussion

Beispiel $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 22 Tabelleneinträge
(hier ohne Herleitung)

❓ Grund für die schlechte Performance?

⚠️ **Sekundäre Häufung:** Synonyme k und k' (mit $h(k) = h(k')$)
durchlaufen dieselbe Sondierungsfolge.

194

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.
 $S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \pmod m$

Beispiel:
 $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \pmod 7$, $h'(k) = 1 + k \pmod 5$.
Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

195

Double Hashing

- Sondierungsreihenfolge muss Permutation aller Hashadressen bilden. Also $h'(k) \neq 0$ und $h'(k)$ darf m nicht teilen, z.B. garantiert mit m prim.
- h' sollte möglichst unabhängig von h sein (Vermeidung sekundärer Häufung).

Unabhängigkeit weitgehend erfüllt von $h(k) = k \pmod m$ und
 $h'(k) = 1 + k \pmod (m-2)$ (m prim).

196

Gleichmässiges Hashing

Starke Annahme: Die Sondierungssequenz $S(k)$ eines Schlüssels k ist mit gleicher Wahrscheinlichkeit eine der $m!$ vielen Permutationssequenzen von $\{0, 1, \dots, m - 1\}$.

(Double Hashing kommt dem am ehesten nahe)

10. Wiederholung Natürliche Suchbäume und Heaps

[Ottman/Widmayer, Kap. 2.3, 5.1, Cormen et al, Kap. 6, 12.1 - 12.3]

Analyse gleichmässiges Hashing mit offener Addressierung

Theorem

Sei eine Hashtabelle mit offener Addressierung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. Unter der Annahme vom gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von $\leq \frac{1}{1-\alpha}$.

Ohne Beweis, siehe z.B. Cormen et al, Kap. 11.4

197

198

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit.

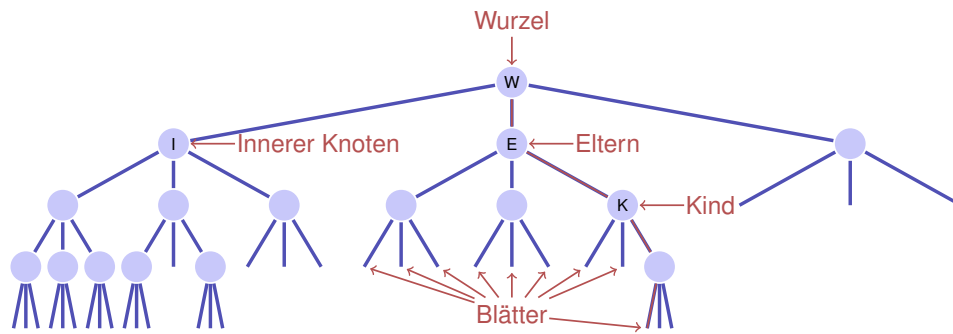
Manche Operationen gar nicht unterstützt:

- Aufzählen von Schlüssel in aufsteigender Anordnung
- Nächst kleinerer Schlüssel zu gegebenem Schlüssel

199

200

Nomenklatur



- Ordnung des Baumes: Maximale Anzahl Kindknoten, hier: 3
- Höhe des Baumes: maximale Pfadlänge Wurzel – Blatt (hier: 4)

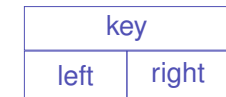
201

Binäre Bäume

Ein binärer Baum ist entweder

- ein Blatt, d.h. ein leerer Baum, oder
- ein innerer Knoten mit zwei Bäumen T_l (linker Teilbaum) und T_r (rechter Teilbaum) als linken und rechten Nachfolger.

In jedem Knoten v speichern wir



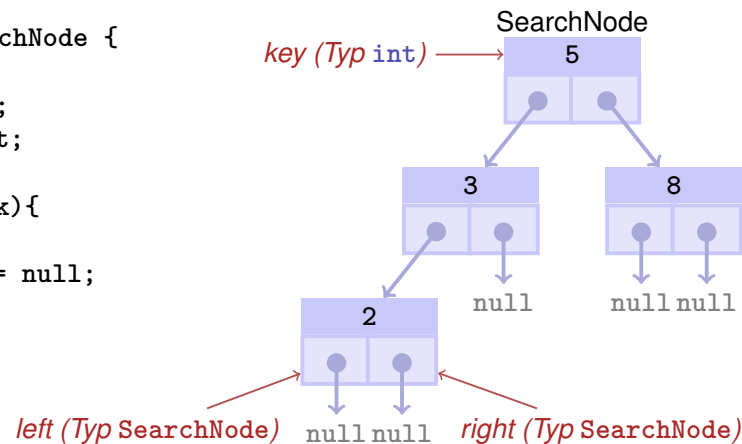
- einen Schlüssel $v.key$ und
- zwei Zeiger $v.left$ und $v.right$ auf die Wurzeln der linken und rechten Teilbäume.
- Ein Blatt wird durch den **null**-Zeiger repräsentiert

202

Baumknoten in Java

```
public class SearchNode {
    int key;
    SearchNode left;
    SearchNode right;

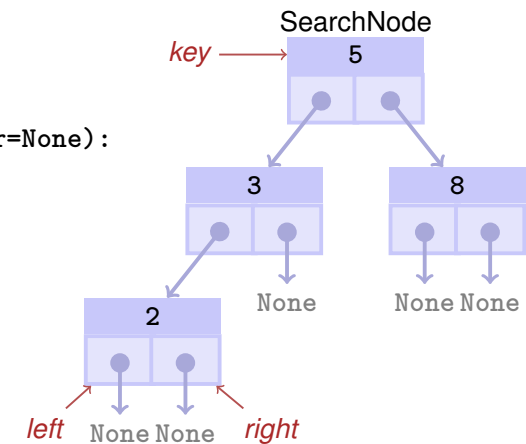
    SearchNode(int k){
        key = k;
        left = right = null;
    }
}
```



203

Baumknoten in Python

```
class SearchNode:
    def __init__(self, k, l=None, r=None):
        self.key = k
        self.left, self.right = l, r
        self.flagged = False
```

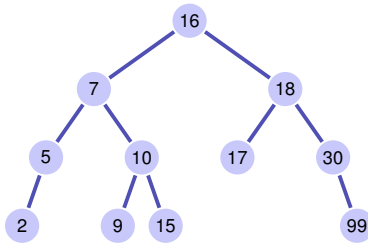


204

Binärer Suchbaum

Ein binärer Suchbaum ist ein binärer Baum, der die *Suchbaumeigenschaft* erfüllt:

- Jeder Knoten v speichert einen Schlüssel
- Schlüssel im linken Teilbaum $v.left$ kleiner als $v.key$
- Schlüssel im rechten Teilbaum $v.right$ grösser als $v.key$



205

Suchen

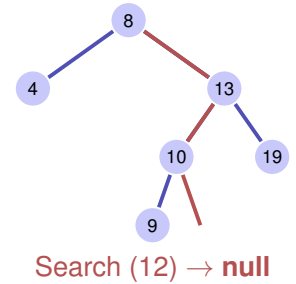
Input: Binärer Suchbaum mit Wurzel r ,
Schlüssel k

Output: Knoten v mit $v.key = k$ oder **null**

$v \leftarrow r$

```
while  $v \neq \text{null}$  do
  if  $k = v.key$  then
    return  $v$ 
  else if  $k < v.key$  then
     $v \leftarrow v.left$ 
  else
     $v \leftarrow v.right$ 
```

return null

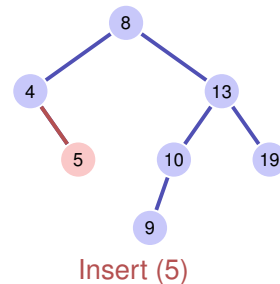


206

Einfügen eines Schlüssels

Einfügen des Schlüssels k

- Suche nach k .
- Wenn erfolgreich:
Fehlerausgabe
- Wenn erfolglos: Einfügen des Schlüssels am erreichten Blatt.



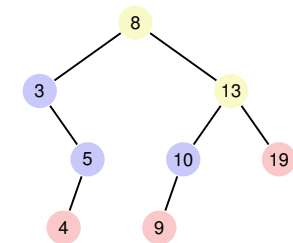
207

Knoten entfernen

Drei Fälle möglich

- Knoten hat keine Kinder
- Knoten hat ein Kind
- Knoten hat zwei Kinder

[Blätter zählen hier nicht]

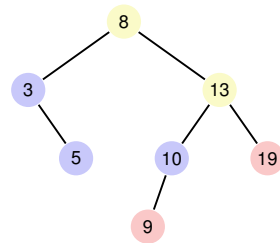
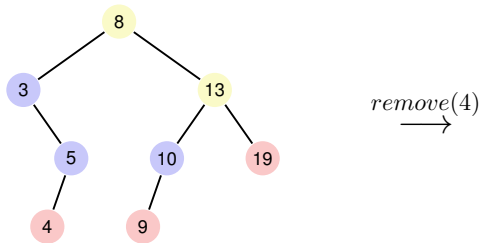


208

Knoten entfernen

Knoten hat keine Kinder

Einfacher Fall: Knoten durch Blatt ersetzen.

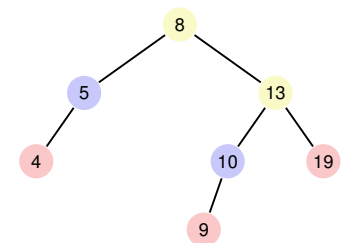
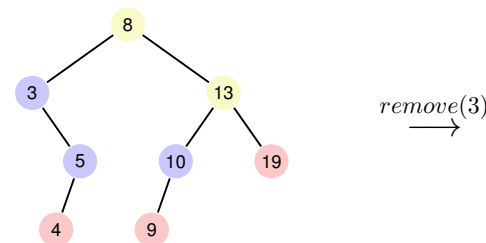


209

Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



210

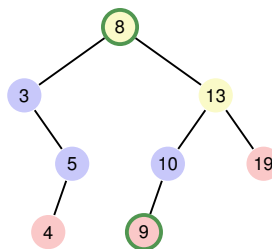
Knoten entfernen

Knoten v hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum $v.right$ (der *symmetrische Nachfolger* von v)

- ist kleiner als alle Schlüssel in $v.right$
- ist grösser als alle Schlüssel in $v.left$
- und hat kein linkes Kind.

Lösung: ersetze v durch seinen symmetrischen Nachfolger

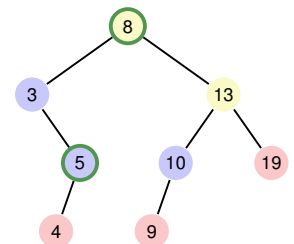


211

Aus Symmetriegründen...

Knoten v hat zwei Kinder

Auch möglich: ersetze v durch seinen symmetrischen Vorgänger



212

Algorithmus SymmetricSuccessor(v)

Input: Knoten v eines binären Suchbaumes

Output: Symmetrischer Nachfolger von v

$w \leftarrow v.\text{right}$

$x \leftarrow w.\text{left}$

while $x \neq \text{null}$ **do**

$w \leftarrow x$

$x \leftarrow x.\text{left}$

return w

Traversierungsarten

■ Hauptreihenfolge (preorder): v , dann $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$.

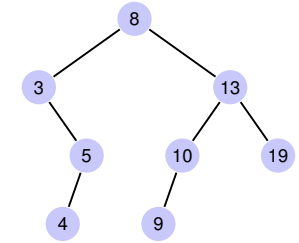
8, 3, 5, 4, 13, 10, 9, 19

■ Nebenreihenfolge (postorder): $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$, dann v .

4, 5, 3, 9, 10, 19, 13, 8

■ Symmetrische Reihenfolge (inorder): $T_{\text{left}}(v)$, dann v , dann $T_{\text{right}}(v)$.

3, 4, 5, 8, 9, 10, 13, 19



213

214

Höhe eines Baumes

Die Höhe $h(T)$ eines Baumes T mit Wurzel r ist gegeben als

$$h(r) = \begin{cases} 0 & \text{falls } r = \text{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist somit im schlechtesten Fall $\mathcal{O}(h(T))$

Analyse

Suchen, Einfügen und Löschen eines Elementes v aus einem Baum T benötigt im schlechtesten Fall $\mathcal{O}(h(T))$ Elementarschritte.

215

216

Mögliche Höhen

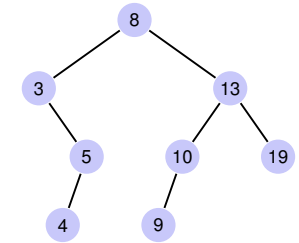
- 1 Die maximale Höhe h_n eines Baumes mit n inneren Knoten ergibt sich durch $h_1 = 1$ und $h_{n+1} \leq 1 + h_n$ als $h_n \geq n$.
- 2 Die minimale Höhe h_n eines (ideal balancierten) Baumes mit n inneren Knoten erfüllt $n \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$.

Somit

$$\lceil \log_2(n + 1) \rceil \leq h \leq n$$

Weitere unterstützte Operationen

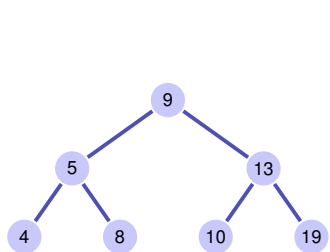
- $\text{Min}(T)$: Auslesen des Minimums in $\mathcal{O}(h)$
- $\text{ExtractMin}(T)$: Auslesen und Entfernen des Minimums in $\mathcal{O}(h)$
- $\text{List}(T)$: Ausgeben einer sortierten Liste der Elemente von T
- $\text{Join}(T_1, T_2)$: Zusammenfügen zweier Bäume mit $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.



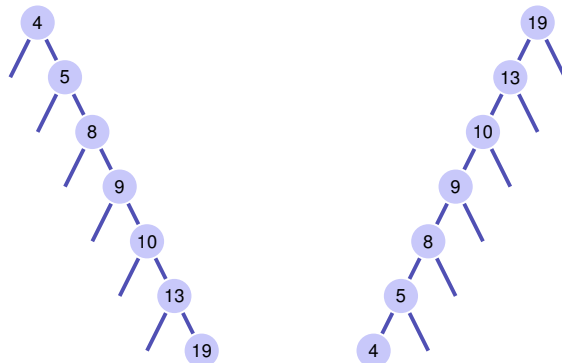
217

218

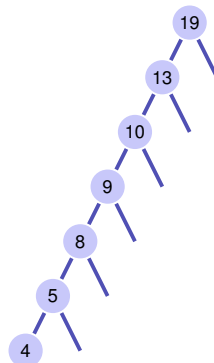
Degenerierte Suchbäume



Insert 9,5,13,4,8,10,19
bestmöglich
balanciert



Insert 4,5,8,9,10,13,19
Lineare Liste



Insert 19,13,10,9,8,5,4
Lineare Liste

[Probabilistisch]

Ein Suchbaum, welcher aus einer zufälligen Sequenz von Zahlen erstellt wird hat erwartete Pfadlänge von $\mathcal{O}(\log n)$.

Achtung: das gilt nur für Einfügeoperation. Wird der Baum zufällig durch Einfügen und Entfernen gebildet, ist die erwartete Pfadlänge $\mathcal{O}(\sqrt{n})$.

Balancierte Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine $\mathcal{O}(\log n)$ Worst-Case-Garantie.

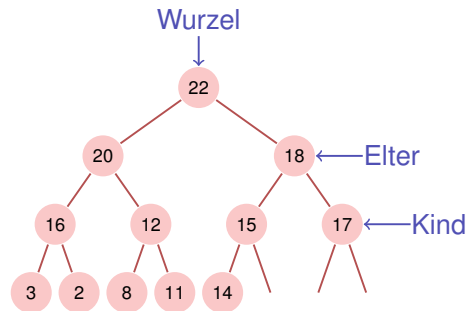
219

(not shown in class) 220

[Max-]Heap⁸

Binärer Baum mit folgenden Eigenschaften

- 1 vollständig, bis auf die letzte Ebene
- 2 Lücken des Baumes in der letzten Ebene höchstens rechts.
- 3 **Heap-Bedingung:** Max-(Min-)Heap: Schlüssel eines Kindes kleiner (größer) als der des Elternknotens

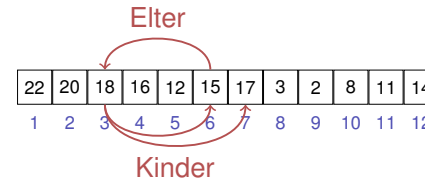


⁸Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

Heap und Array

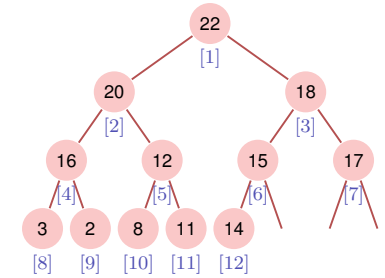
Baum → Array:

- $Kinder(i) = \{2i, 2i + 1\}$
- $Elter(i) = \lfloor i/2 \rfloor$



Abhängig von Startindex!⁹

⁹Für Arrays, die bei 0 beginnen: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$



Höhe eines Heaps

Vollständiger binärer Baum der Höhe¹⁰ h hat

$$1 + 2 + 4 + 8 + \dots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Knoten. Somit gilt für einen Heap der Höhe h :

$$2^{h-1} - 1 < n \leq 2^h - 1$$

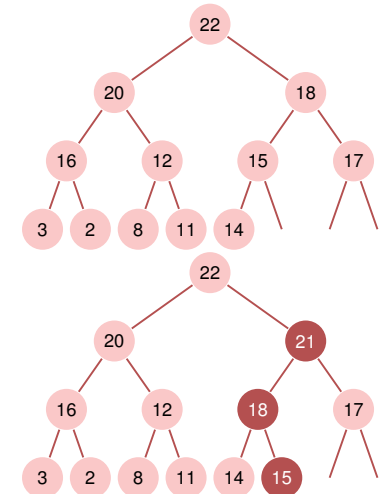
$$\Leftrightarrow 2^{h-1} < n + 1 \leq 2^h$$

Also insbesondere $h(n) = \lceil \log_2(n + 1) \rceil$ und $h(n) \in \Theta(\log n)$.

¹⁰Hier: Anzahl Kanten von der Wurzel zu einem Blatt

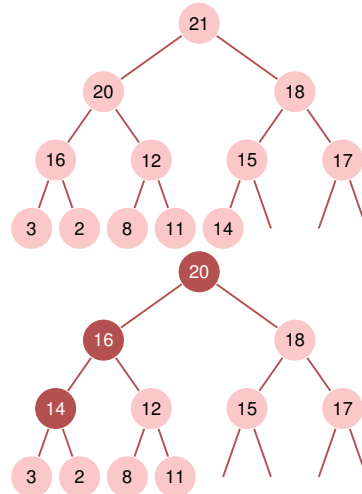
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



225

Algorithmus Versickern(A, i, m)

Input: Array A mit Heapstruktur für die Kinder von i . Letztes Element m .

Output: Array A mit Heapstruktur für i mit letztem Element m .

```

while  $2i \leq m$  do
     $j \leftarrow 2i$ ; //  $j$  linkes Kind
    if  $j < m$  and  $A[j] < A[j + 1]$  then
         $j \leftarrow j + 1$ ; //  $j$  rechtes Kind mit grösserem Schlüssel
    if  $A[i] < A[j]$  then
        swap( $A[i], A[j]$ )
         $i \leftarrow j$ ; // weiter versickern
    else
         $i \leftarrow m$ ; // versickern beendet
    
```

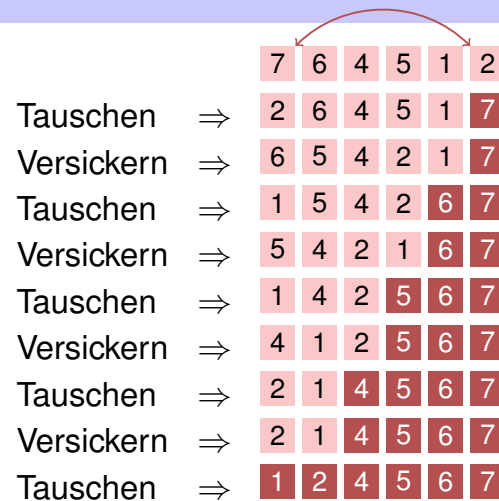
226

Heap Sortieren

$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

- swap($A[1], A[n]$)
- Versickere($A, 1, n - 1$);
- $n \leftarrow n - 1$



227

Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

228

Algorithmus HeapSort(A, n)

Input: Array A der Länge n .

Output: A sortiert.

// Heap Bauen.

for $i \leftarrow n/2$ **downto** 1 **do**

└ Versickere(A, i, n);

// Nun ist A ein Heap.

for $i \leftarrow n$ **downto** 2 **do**

└ swap($A[1], A[i]$)

└ Versickere($A, 1, i - 1$)

// Nun ist A sortiert.

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. \Rightarrow Heap Sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.

229

230

[Analyse: Heap bauen]

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade sind aber im Mittel viel kürzer:

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level } l} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{Höhe Heaps auf Level } l} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k \\ &\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(n) \end{aligned}$$

mit $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ($0 < x < 1$)¹¹ und $s(\frac{1}{2}) = 2$

¹¹ $f(x) = \frac{1}{1-x} = 1 + x + x^2 \dots \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + \dots$

(not shown in class) 231

11. AVL Bäume

Balancierte Bäume [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

232

Ziel

Suchen, Einfügen und Entfernen eines Schlüssels in Baum mit n Schlüsseln, welche in zufälliger Reihenfolge eingefügt wurden im Mittel in $\mathcal{O}(\log_2 n)$ Schritten.

Schlechtester Fall jedoch: $\Theta(n)$ (degenerierter Baum).

Ziel: Verhinderung der Degenerierung. Künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes

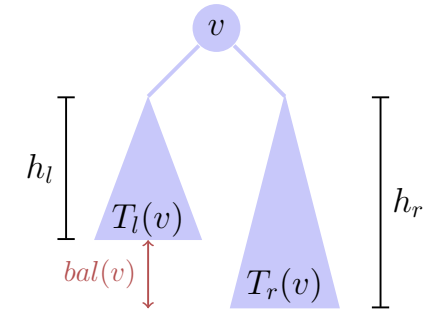
Balancierung: garantiere, dass ein Baum mit n Knoten stets eine Höhe von $\mathcal{O}(\log n)$ hat.

Adelson-Venskii und Landis (1962): AVL-Bäume

Balance eines Knotens

Die **Balance** eines Knotens v ist definiert als die Höhendifferenz seiner beiden Teilbäume $T_l(v)$ und $T_r(v)$

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$

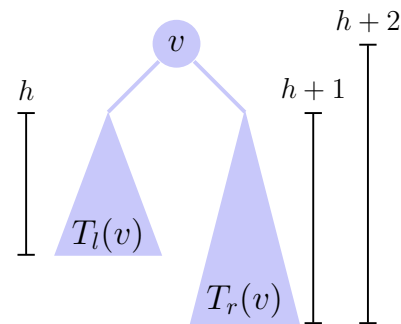


233

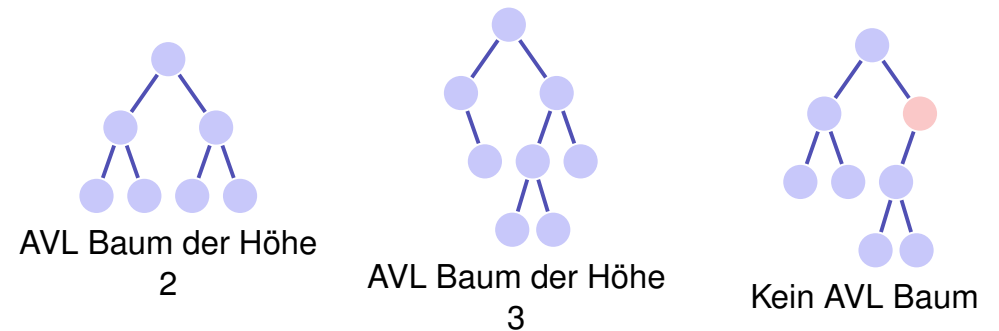
234

AVL Bedingung

AVL Bedingung: für jeden Knoten v eines Baumes gilt $\text{bal}(v) \in \{-1, 0, 1\}$



(Gegen-)Beispiele



235

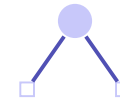
236

Anzahl Blätter

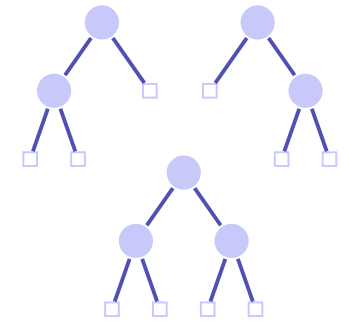
- 1. Beobachtung: Ein Suchbaum mit n Schlüsseln hat genau $n + 1$ Blätter. Einfaches Induktionsargument.
 - Der Suchbaum mit $n = 0$ Schlüsseln hat $m = 1$ Blätter
 - Wird ein Schlüssel (Knoten) hinzugefügt ($n \rightarrow n + 1$), so ersetzt er ein Blatt und fügt zwei Blätter hinzu ($m \rightarrow m - 1 + 2 = m + 1$).
- 2. Beobachtung: untere Grenze für Anzahl Blätter eines Suchbaums zu gegebener Höhe erlaubt Abschätzung der maximalen Höhe eines Suchbaums zu gegebener Anzahl Schlüssel.

237

Untere Grenze Blätter



AVL Baum der Höhe 1 hat
 $N(1) := 2$ Blätter



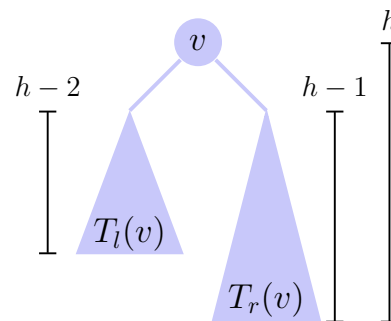
AVL Baum der Höhe 2 hat
mindestens $N(2) := 3$
Blätter

238

Untere Grenze Blätter für $h > 2$

- Höhe eines Teilbaums $\geq h - 1$.
 - Höhe des anderen Teilbaums $\geq h - 2$.
- Minimale Anzahl Blätter $N(h)$ ist

$$N(h) = N(h - 1) + N(h - 2)$$



Insgesamt gilt $N(h) = F_{h+2}$ mit **Fibonacci-Zahlen** $F_0 := 0, F_1 := 1, F_n := F_{n-1} + F_{n-2}$ für $n > 1$.

239

Fibonacci Zahlen, geschlossene Form

Es gilt

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$$

mit den Wurzeln $\phi, \hat{\phi}$ der Gleichung vom goldenen Schnitt
 $x^2 - x - 1 = 0$:

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

240

[Fibonacci Zahlen, Induktiver Beweis]

$$F_i \stackrel{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \quad [*] \quad \left(\phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2} \right).$$

1 Klar für $i = 0, i = 1$.

2 Sei $i > 2$ und Behauptung [*] wahr für alle $F_j, j < i$.

$$\begin{aligned} F_i &\stackrel{def}{=} F_{i-1} + F_{i-2} \stackrel{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2}) \\ &= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1) \end{aligned}$$

($\phi, \hat{\phi}$ erfüllen $x + 1 = x^2$)

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

(not shown in class) 241

Baumhöhe

Da $|\hat{\phi}| < 1$, gilt insgesamt

$$N(h) \in \Theta \left(\left(\frac{1+\sqrt{5}}{2} \right)^h \right) \subseteq \Omega(1.618^h)$$

und somit

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow h &\leq 1.44 \log_2 n + c'. \end{aligned}$$

Ein AVL Baum ist asymptotisch nicht mehr als 44% höher als ein perfekt balancierter Baum.¹²

¹²Ein perfekt balancierter Baum hat Höhe $\lceil \log_2 n + 1 \rceil$

242

Einfügen

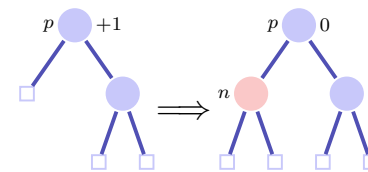
Balancieren

- Speichern der Balance für jeden Knoten
- Baum rebalancieren bei jeder Update-Operation

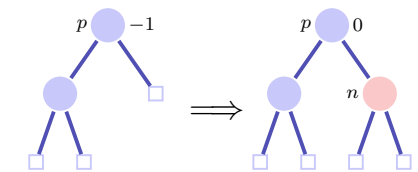
Neuer Knoten n wird eingefügt:

- Zuerst einfügen wie bei Suchbaum.
- Prüfe die Balance-Bedingung für alle Knoten aufsteigend von n zur Wurzel.

Balance am Einfügeort



Fall 1: $\text{bal}(p) = +1$



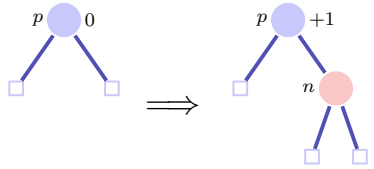
Fall 2: $\text{bal}(p) = -1$

Fertig in beiden Fällen, denn der Teilbaum ist nicht gewachsen.

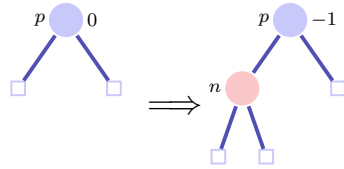
243

244

Balance am Einfügeort



Fall 3.1: $\text{bal}(p) = 0$ rechts



Fall 3.2: $\text{bal}(p) = 0$, links

In beiden Fällen noch nicht fertig. Aufruf von `upin(p)`.

245

246

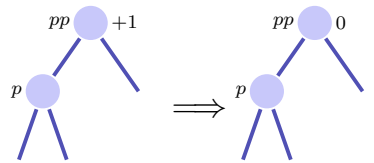
upin(p) - Invariante

Beim Aufruf von `upin(p)` gilt, dass

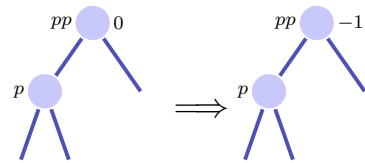
- der Teilbaum ab p gewachsen ist und
- $\text{bal}(p) \in \{-1, +1\}$

upin(p)

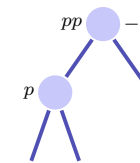
Annahme: p ist linker Sohn von pp ¹³



Fall 1: $\text{bal}(pp) = +1$, fertig.



Fall 2: $\text{bal}(pp) = 0$, `upin(pp)`



Fall 3: $\text{bal}(pp) = -1$,

In beiden Fällen gilt nach der Operation die AVL-Bedingung für den Teilbaum ab pp

upin(p)

Annahme: p ist linker Sohn von pp

Dieser Fall ist problematisch: das Hinzufügen von n im Teilbaum ab pp hat die AVL-Bedingung verletzt. Rebalancieren!

Zwei Fälle $\text{bal}(p) = -1$, $\text{bal}(p) = +1$

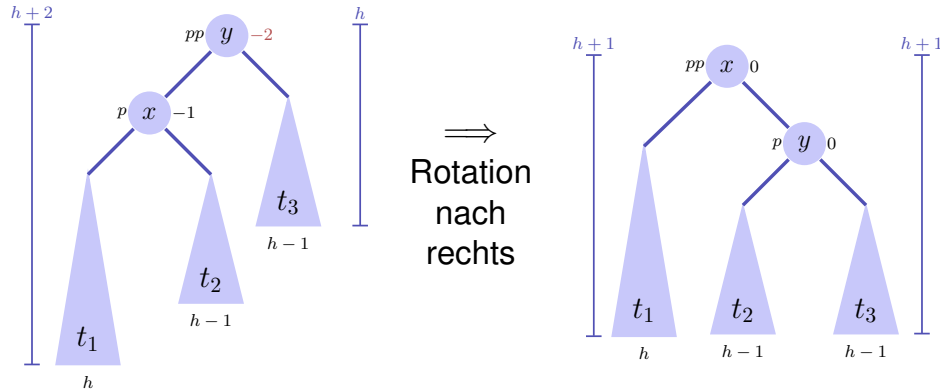
¹³Ist p rechter Sohn: symmetrische Fälle unter Vertauschung von $+1$ und -1

247

248

Rotationen

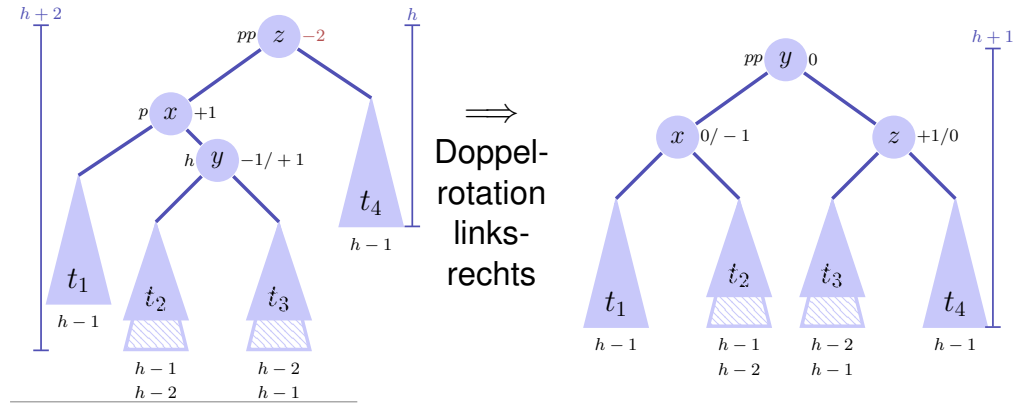
Fall 1.1 $\text{bal}(p) = -1$.¹⁴



¹⁴ p rechter Sohn $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$, Linksrotation

Rotationen

Fall 1.2 $\text{bal}(p) = +1$.¹⁵



¹⁵ p rechter Sohn $\Rightarrow \text{bal}(pp) = +1$, $\text{bal}(p) = -1$, Doppelrotation rechts links

Analyse

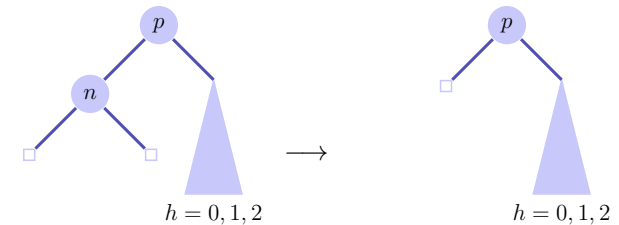
- Höhe des Baumes: $\mathcal{O}(\log n)$.
- Einfügen wie beim binären Suchbaum.
- Balancieren durch Rekursion vom Knoten zur Wurzel. Maximale Pfadlänge $\mathcal{O}(\log n)$.

Das Einfügen im AVL-Baum hat Laufzeitkosten von $\mathcal{O}(\log n)$.

Löschen

Fall 1: Knoten n hat zwei Blätter als Kinder Sei p Elternknoten von n .
 \Rightarrow Anderer Teilbaum hat Höhe $h' = 0, 1$ oder 2

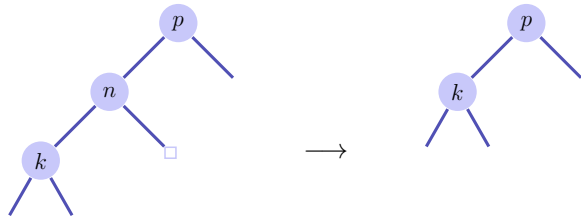
- $h' = 1$: $\text{bal}(p)$ anpassen.
- $h' = 0$: $\text{bal}(p)$ anpassen. Aufruf `upout(p)`.
- $h' = 2$: Rebalancieren des Teilbaumes. Aufruf `upout(p)`.



Löschen

Fall 2: Knoten n hat einen inneren Knoten k als Kind

- Ersetze n durch k . **upout (k)**



253

Löschen

Fall 3: Knoten n hat zwei inneren Knoten als Kinder

- Ersetze n durch symmetrischen Nachfolger. **upout (k)**
- Löschen des symmetrischen Nachfolgers wie in Fall 1 oder 2.

254

upout (p)

Sei pp der Elternknoten von p

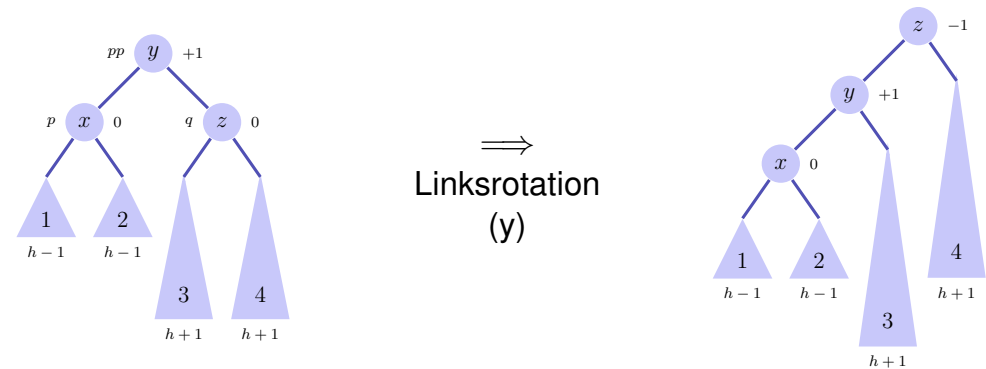
(a) p linkes Kind von pp

- 1 $\text{bal}(pp) = -1 \Rightarrow \text{bal}(pp) \leftarrow 0$. **upout (pp)**
- 2 $\text{bal}(pp) = 0 \Rightarrow \text{bal}(pp) \leftarrow +1$.
- 3 $\text{bal}(pp) = +1 \Rightarrow$ nächste Folien.

(b) p rechtes Kind von pp : Symmetrische Fälle unter Vertauschung von $+1$ und -1 .

upout (p)

Fall (a).3: $\text{bal}(pp) = +1$. Sei q Bruder von p
 (a).3.1: $\text{bal}(q) = 0$.¹⁶



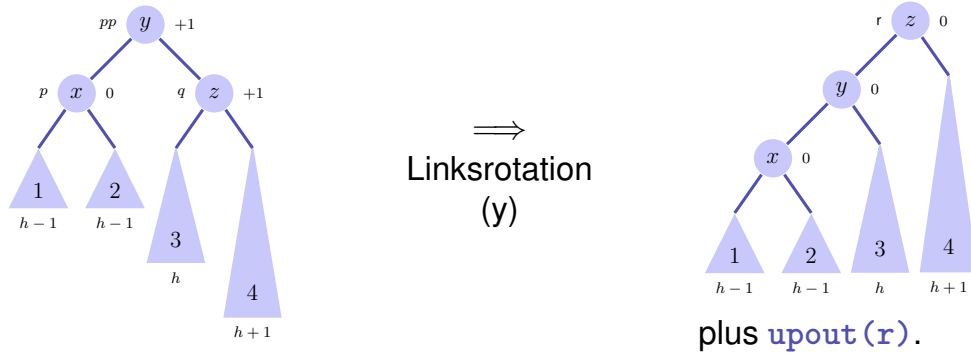
255

¹⁶(b).3.1: $\text{bal}(pp) = -1, \text{bal}(q) = -1$, Rechtsrotation.

256

upout (p)

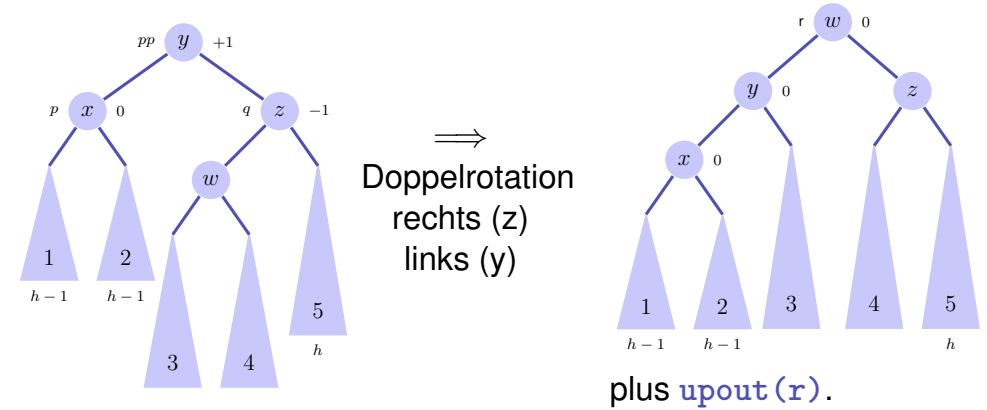
Fall (a).3: $\text{bal}(pp) = +1$. (a).3.2: $\text{bal}(q) = +1$.¹⁷



¹⁷(b).3.2: $\text{bal}(pp) = -1$, $\text{bal}(q) = +1$, Rechtsrotation+upout

upout (p)

Fall (a).3: $\text{bal}(pp) = +1$. (a).3.3: $\text{bal}(q) = -1$.¹⁸



¹⁸(b).3.3: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, Links-Rechts-Rotation + upout

Zusammenfassung

- AVL-Bäume haben asymptotische Laufzeit von $\mathcal{O}(\log n)$ (schlechtester Fall) für das Suchen, Einfügen und Löschen von Schlüsseln
- Einfügen und Löschen ist verhältnismässig aufwändig und für kleine Probleme relativ langsam.

12. Dynamische Programmierung

Memoisieren, Optimale Substruktur, Überlappende Teilprobleme, Abhängigkeiten, Allgemeines Vorgehen. Beispiele: Schneiden von Eisenstangen, Kaninchen, Editierdistanz

[Ottman/Widmayer, Kap. 7.1, 7.4, Cormen et al, Kap. 15]

Fibonacci Zahlen



(schon wieder)

$$F_n := \begin{cases} n & \text{wenn } n < 2 \\ F_{n-1} + F_{n-2} & \text{wenn } n \geq 2. \end{cases}$$

Analyse: warum ist der rekursive Algorithmus so langsam.

261

Algorithmus FibonacciRecursive(n)

Input: $n \geq 0$

Output: n -te Fibonacci Zahl

if $n < 2$ **then**

$f \leftarrow n$

else

$f \leftarrow \text{FibonacciRecursive}(n - 1) + \text{FibonacciRecursive}(n - 2)$

return f

262

Analyse

$T(n)$: Anzahl der ausgeführten Operationen.

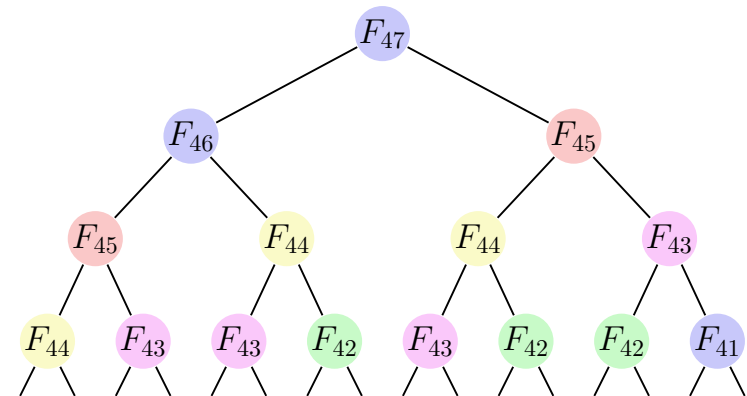
■ $n = 0, 1$: $T(n) = \Theta(1)$

■ $n \geq 2$: $T(n) = T(n - 2) + T(n - 1) + c$.

$$T(n) = T(n - 2) + T(n - 1) + c \geq 2T(n - 2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Algorithmus ist *exponentiell (!)* in n .

Grund, visualisiert



Knoten mit denselben Werten werden (zu) oft ausgewertet.

263

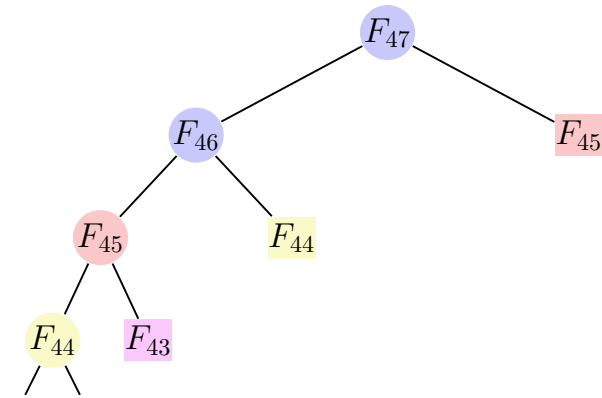
264

Memoization

Memoization (sic) Abspeichern von Zwischenergebnissen.

- Bevor ein Teilproblem gelöst wird, wird Existenz eines entsprechenden Zwischenergebnis geprüft.
- Existiert ein gespeichertes Zwischenergebnis bereits, so wird dieses verwendet.
- Andernfalls wird der Algorithmus ausgeführt und das Ergebnis wird entsprechend gespeichert.

Memoization bei Fibonacci



Rechteckige Knoten wurden bereits ausgewertet.

265

266

Algorithmus FibonacciMemoization(n)

Input: $n \geq 0$

Output: n -te Fibonacci Zahl

if $n \leq 2$ **then**

$f \leftarrow 1$

else if $\exists \text{memo}[n]$ **then**

$f \leftarrow \text{memo}[n]$

else

$f \leftarrow \text{FibonacciMemoization}(n-1) + \text{FibonacciMemoization}(n-2)$
 $\text{memo}[n] \leftarrow f$

return f

Analyse

Berechnungsaufwand:

$$T(n) = T(n-1) + c = \dots = \mathcal{O}(n).$$

denn nach dem Aufruf von $f(n-1)$ wurde $f(n-2)$ bereits berechnet.

Das lässt sich auch so sehen: Für jedes n wird $f(n)$ maximal einmal rekursiv berechnet. Laufzeitkosten: n Aufrufe mal $\Theta(1)$ Kosten pro Aufruf $n \cdot c \in \Theta(n)$. Die Rekursion verschwindet aus der Berechnung der Laufzeit.

Algorithmus benötigt $\Theta(n)$ Speicher.¹⁹

¹⁹Allerdings benötigt der naive Algorithmus auch $\Theta(n)$ Speicher für die Rekursionsverwaltung.

267

268

Genauer hingesehen ...

... berechnet der Algorithmus der Reihe nach die Werte $F_1, F_2, F_3,$
... verkleidet im *Top-Down* Ansatz der Rekursion.

Man kann den Algorithmus auch gleich *Bottom-Up* hinschreiben.
Das ist charakteristisch für die *dynamische Programmierung*.

Algorithmus FibonacciBottomUp(n)

Input: $n \geq 0$
Output: n -te Fibonacci Zahl

```
F[1] ← 1
F[2] ← 1
for i ← 3, ..., n do
  F[i] ← F[i - 1] + F[i - 2]
return F[n]
```

269

270

Dynamische Programmierung: Idee

- Aufteilen eines komplexen Problems in eine vernünftige Anzahl kleinerer Teilprobleme
- Die Lösung der Teilprobleme wird zur Lösung des komplexeren Problems verwendet
- Identische Teilprobleme werden nur einmal gerechnet

Dynamische Programmierung: Konsequenz

Identische Teilprobleme werden nur einmal gerechnet
⇒ Resultate werden zwischengespeichert



Wir tauschen Laufzeit
gegen Speicherplatz

271

272

Dynamic Programming: Beschreibung

- 1 Verwalte *DP-Tabelle* mit Information zu den Teilproblemen.
Dimension der Tabelle? Bedeutung der Einträge?
- 2 Berechnung der *Randfälle*.
Welche Einträge hängen nicht von anderen ab?
- 3 *Berechnungsreihenfolge* bestimmen.
In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?
- 4 Auslesen der *Lösung*.
Wie kann sich Lösung aus der Tabelle konstruieren lassen?

Laufzeit (typisch) = Anzahl Einträge der Tabelle mal Aufwand pro Eintrag.

273

Dynamic Programming: Beschreibung am Beispiel

- 1 Dimension der Tabelle? Bedeutung der Einträge?
Tabelle der Grösse $n \times 1$. n -ter Eintrag enthält n -te Fibonacci Zahl.
- 2 Welche Einträge hängen nicht von anderen ab?
Werte F_1 und F_2 sind unabhängig einfach "berechenbar".
- 3 In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?
 F_i mit aufsteigenden i .
- 4 Wie kann sich Lösung aus der Tabelle konstruieren lassen?
 F_n ist die n -te Fibonacci-Zahl.

274

Dynamic Programming = Divide-And-Conquer ?

- In beiden Fällen ist das Ursprungsproblem (einfacher) lösbar, indem Lösungen von Teilproblemen herangezogen werden können. Das Problem hat *optimale Substruktur*.
- Bei Divide-And-Conquer Algorithmen (z.B. Mergesort) sind Teilprobleme unabhängig; deren Lösungen werden im Algorithmus nur einmal benötigt.
- Beim DP sind Teilprobleme nicht unabhängig. Das Problem hat *überlappende Teilprobleme*, welche im Algorithmus mehrfach gebraucht werden.
- Damit sie nur einmal gerechnet werden müssen, werden Resultate tabelliert. Dafür darf es *zwischen Teilproblemen keine zirkulären Abhängigkeiten* geben.

275

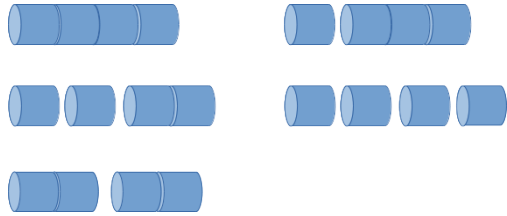
Schneiden von Eisenstäben

- Metallstäbe werden zerschnitten und verkauft.
- Metallstäbe der Länge $n \in \mathbb{N}$ verfügbar. Zerschneiden kostet nichts.
- Für jede Länge $l \in \mathbb{N}$, $l \leq n$ bekannt: Wert $v_l \in \mathbb{R}^+$
- Ziel: Zerschneide die Stange so (in $k \in \mathbb{N}$ Stücke), dass

$$\sum_{i=1}^k v_{l_i} \text{ maximal unter } \sum_{i=1}^k l_i = n.$$

276

Schneiden von Eisenstäben: Beispiel



Arten, einen Stab der Länge 4 zu zerschneiden (ohne Permutationen)

Länge	0	1	2	3	4
Preis	0	2	3	8	9

⇒ Bester Schnitt: 3 + 1 mit Wert 10.

277

Wie findet man den DP Algorithmus

- 0 Genaue Formulierung der gesuchten Lösung
- 1 Definiere Teilprobleme (und bestimme deren Anzahl)
- 2 Raten / Aufzählen (und bestimme die Laufzeit für das Raten)
- 3 Rekursion: verbinde die Teilprobleme
- 4 Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme
- 5 Lösung des Problems
Laufzeit = #Teilprobleme × Zeit/Teilproblem

278

Struktur des Problems

- 0 **Gesucht:** r_n = maximal erreichbarer Wert von (ganzem oder geschnittenem) Stab mit Länge n .
- 1 **Teilprobleme:** maximal erreichbarer Wert r_k für alle $0 \leq k < n$
- 2 **Rate** Länge des ersten Stückes
- 3 **Rekursion**

$$r_k = \max \{v_i + r_{k-i} : 0 < i \leq k\}, \quad k > 0$$

$$r_0 = 0$$

- 4 **Abhängigkeit:** r_k hängt (nur) ab von den Werten $v_i, l \leq i \leq k$ und den optimalen Schnitten $r_i, i < k$
- 5 **Lösung** in r_n

279

Algorithmus RodCut(v, n)

Input: $n \geq 0$, Preise v

Output: bester Wert

```

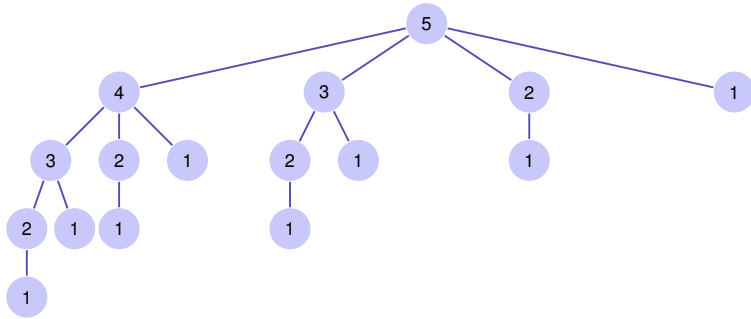
q ← 0
if n > 0 then
  for i ← 1, ..., n do
    q ← max{q, vi + RodCut(v, n - i)};
return q
    
```

Laufzeit $T(n) = \sum_{i=0}^{n-1} T(i) + c \Rightarrow^{20} T(n) \in \Theta(2^n)$

²⁰ $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

280

Rekursionsbaum



Algorithmus RodCutMemoized(m, v, n)

Input: $n \geq 0$, Preise v , Memoization Tabelle m

Output: bester Wert

```

q ← 0
if n > 0 then
  if ∃ m[n] then
    q ← m[n]
  else
    for i ← 1, ..., n do
      q ← max{q, vi + RodCutMemoized(m, v, n - i)};
    m[n] ← q
return q
  
```

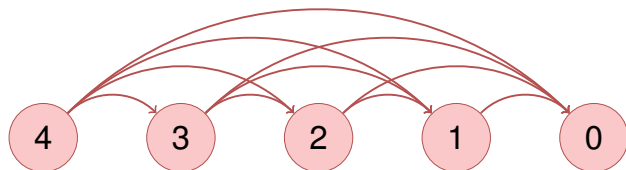
Laufzeit $\sum_{i=1}^n i = \Theta(n^2)$

281

282

Teilproblem-Graph

beschreibt die Abhängigkeiten der Teilprobleme untereinander



und darf keine Zyklen enthalten

Konstruktion des optimalen Schnittes

- Während der (rekursiven) Berechnung der optimalen Lösung für jedes $k \leq n$ bestimmt der rekursive Algorithmus die optimale Länge des ersten Stabes
- Speichere die Länge des ersten Stabes für jedes $k \leq n$ in einer Tabelle mit n Einträgen.

283

284

Bottom-Up Beschreibung am Beispiel

Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle der Grösse $n \times 1$. n -ter Eintrag enthält besten Wert eines Stabes der Länge n .

Welche Einträge hängen nicht von anderen ab?

Wert r_0 ist 0.

In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?

$r_i, i = 1, \dots, n$.

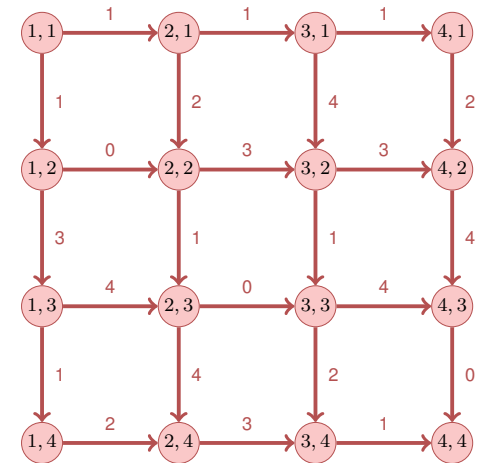
Wie kann sich Lösung aus der Tabelle konstruieren lassen?

r_n ist der beste Wert für eine Stange der Länge n

285

Kaninchen!

Ein Kaninchen sitzt auf Platz $(1, 1)$ eines $n \times n$ Gitters. Es kann nur nach Osten oder nach Süden gehen. Auf jedem Wegstück liegt eine Anzahl Rüben. Wie viele Rüben sammelt das Kaninchen maximal ein?



286

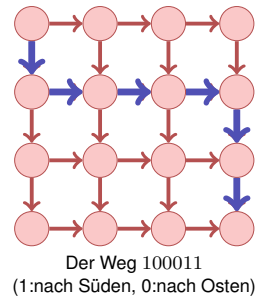
Kaninchen!

Anzahl mögliche Pfade?

- Auswahl von $n - 1$ Wegen nach Süden aus $2n - 2$ Wegen insgesamt.

$$\binom{2n-2}{n-1} \in \Omega(2^n)$$

⇒ Naiver Algorithmus hat keine Chance



287

Rekursion

Gesucht: $T_{0,0} = \text{Maximale Anzahl Rüben von } (0, 0) \text{ nach } (n, n)$.

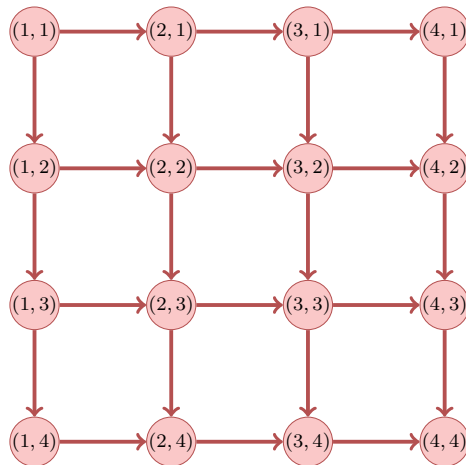
Sei $w_{(i,j)-(i',j')}$ Anzahl Rüben auf Kante von (i, j) nach (i', j') .

Rekursion (maximale Anzahl Rüben von (i, j) nach (n, n))

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i < n, j < n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i = n, j < n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

288

Teilproblemabhängigkeitsgraph



Bottom-Up Beschreibung am Beispiel

Dimension der Tabelle? Bedeutung der Einträge?

1 Tabelle T der Grösse $n \times n$. Eintrag bei i, j enthält die maximale Anzahl Rüben von (i, j) nach (n, n) .

2 Welche Einträge hängen nicht von anderen ab?

Wert $T_{n,n}$ ist 0.

3 In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?

$T_{i,j}$ mit $i = n \searrow 1$ und für jedes $i: j = n \searrow 1$, (oder umgekehrt: $j = n \searrow 1$ und für jedes $j: i = n \searrow 1$).

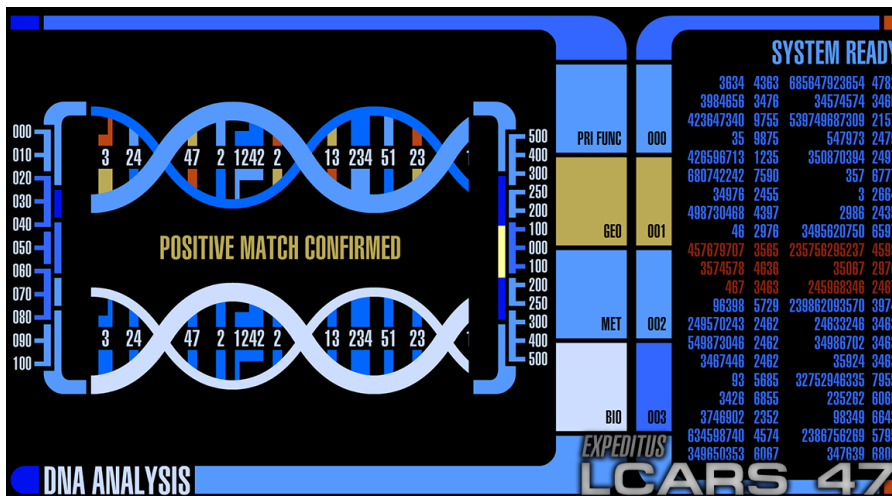
4 Wie kann sich Lösung aus der Tabelle konstruieren lassen?

$T_{1,1}$ enthält die maximale Anzahl Rüben

289

290

DNA - Vergleich (Star Trek)



DNA - Vergleich

- DNA besteht aus Sequenzen von vier verschiedenen Nukleotiden **Adenin Guanin Thymin Cytosin**
- DNA-Sequenzen (Gene) werden mit Zeichenketten aus A, G, T und C beschrieben.
- Ein möglicher Vergleich zweier Gene: Bestimme **Längste gemeinsame Teilfolge**

Das Problem, die längste gemeinsame Teilfolge zu finden ist ein Spezialfall der minimalen Editierdistanz. Die folgenden Folien werden daher in der Vorlesung nicht behandelt.

291

292

[Längste Gemeinsame Teilfolge]

Teilfolgen einer Zeichenkette:

Teilfolgen(KUH): (), (K), (U), (H), (KU), (KH), (UH), (KUH)

Problem:

- **Eingabe:** Zwei Zeichenketten $A = (a_1, \dots, a_m)$, $B = (b_1, \dots, b_n)$ der Längen $m > 0$ und $n > 0$.
- **Gesucht:** Eine längste gemeinsame Teilfolge (LGT) von A und B .

(not shown in class) 293

[Längste Gemeinsame Teilfolge]

Beispiele:

$LGT(IGEL, KATZE) = E$, $LGT(TIGER, ZIEGE) = IGE$

Ideen zur Lösung?

T I G E R
Z I E G E

(not shown in class) 294

[Rekursives Vorgehen]

Annahme: Lösungen $L(i, j)$ bekannt für $A[1, \dots, i]$ und $B[1, \dots, j]$ für alle $1 \leq i \leq m$ und $1 \leq j \leq n$, jedoch nicht für $i = m$ und $j = n$.

T I G E R
Z I E G E

Betrachten Zeichen a_m, b_n . Drei Möglichkeiten:

- 1 A wird um ein Leerzeichen erweitert. $L(m, n) = L(m, n - 1)$
- 2 B wird um ein Leerzeichen erweitert. $L(m, n) = L(m - 1, n)$
- 3 $L(m, n) = L(m - 1, n - 1) + \delta_{mn}$ mit $\delta_{mn} = 1$ wenn $a_m = b_n$ und $\delta_{mn} = 0$ sonst

(not shown in class) 295

[Rekursion]

$L(m, n) \leftarrow \max \{L(m - 1, n - 1) + \delta_{mn}, L(m, n - 1), L(m - 1, n)\}$

für $m, n > 0$ und Randfälle $L(\cdot, 0) = 0$, $L(0, \cdot) = 0$.

	\emptyset	Z	I	E	G	E
\emptyset	0	0	0	0	0	0
T	0	0	0	0	0	0
I	0	0	1	1	1	1
G	0	0	1	1	2	2
E	0	0	1	2	2	3
R	0	0	1	2	2	3

(not shown in class) 296

[Dynamic Programming Algorithmus LGT]

Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle $L[0, \dots, m][0, \dots, n]$. $L[i, j]$: Länge einer LGT der Zeichenketten (a_1, \dots, a_i) und (b_1, \dots, b_j)

Berechnung eines Eintrags

- 2 $L[0, i] \leftarrow 0 \forall 0 \leq i \leq m$, $L[j, 0] \leftarrow 0 \forall 0 \leq j \leq n$. Berechnung von $L[i, j]$ sonst mit $L[i, j] = \max(L[i-1, j-1] + \delta_{ij}, L[i, j-1], L[i-1, j])$.

(not shown in class) 297

[Dynamic Programming Algorithmus LGT]

Berechnungsreihenfolge

- 3 Abhängigkeiten berücksichtigen: z.B. Zeilen aufsteigend und innerhalb von Zeilen Spalten aufsteigend.

Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4 Beginne bei $j = m, i = n$. Falls $a_i = b_j$ gilt, gib a_i aus und fahre fort mit $(j, i) \leftarrow (j-1, i-1)$; sonst, falls $L[i, j] = L[i, j-1]$ fahre fort mit $j \leftarrow j-1$; sonst, falls $L[i, j] = L[i-1, j]$ fahre fort mit $i \leftarrow i-1$. Terminiere für $i = 0$ oder $j = 0$.

(not shown in class) 298

[Analyse LGT]

- Anzahl Tabelleneinträge: $(m+1) \cdot (n+1)$.
- Berechnung jeweils mit konstanter Anzahl Zuweisungen und Vergleichen. Anzahl Schritte $\mathcal{O}(mn)$
- Bestimmen der Lösung: jeweils Verringerung von i oder j . Maximal $\mathcal{O}(n+m)$ Schritte.

Laufzeit insgesamt:

$$\mathcal{O}(mn).$$

(not shown in class) 299

Minimale Editierdistanz

Editierdistanz von zwei Zeichenketten $A_n = (a_1, \dots, a_m)$, $B_m = (b_1, \dots, b_m)$.

Editieroperationen:

- Einfügen eines Zeichens
- Löschen eines Zeichens
- Änderung eines Zeichens

Frage: Wie viele Editieroperationen sind mindestens nötig, um eine gegebene Zeichenkette A in eine Zeichenkette B zu überführen.

TIGER ZIGER ZIEGER ZIEGE

300

Minimale Editierdistanz

Gesucht: Günstigste zeichenweise Transformation $A_n \rightarrow B_m$ mit Kosten

Operation	Levenshtein	LGT ²¹	allgemein
c einfügen	1	1	ins(c)
c löschen	1	1	del(c)
Ersetzen $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

T	I	G	E	R	T	I	_	G	E	R	T→Z	+E	-R
Z	I	E	G	E	Z	I	E	G	E	_	Z→T	-E	+R

²¹Längste gemeinsame Teilfolge – Spezialfall des Editierproblems

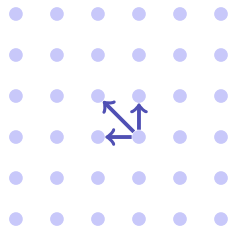
DP

- 0 $E(n, m)$ = minimale Anzahl Editieroperationen (ED Kosten) für $a_{1..n} \rightarrow b_{1..m}$
- 1 Teilprobleme $E(i, j)$ = ED von $a_{1..i} \cdot b_{1..j}$. #TP = $n \cdot m$
- 2 Raten/Probieren Kosten $\Theta(1)$
 - $a_{1..i} \rightarrow a_{1..i-1}$ (löschen)
 - $a_{1..i} \rightarrow a_{1..i}b_j$ (einfügen)
 - $a_{1..i} \rightarrow a_{1..i_1}b_j$ (ersetzen)
- 3 Rekursion

$$E(i, j) = \min \begin{cases} \text{del}(a_i) + E(i-1, j), \\ \text{ins}(b_j) + E(i, j-1), \\ \text{repl}(a_i, b_j) + E(i-1, j-1) \end{cases}$$

DP

4 Abhängigkeiten



⇒ Berechnung von links oben nach rechts unten. Zeilen- oder Spaltenweise.

5 Lösung steht in $E(n, m)$

Beispiel (Levenshteinabstand)

$$E[i, j] \leftarrow \min \{ E[i-1, j] + 1, E[i, j-1] + 1, E[i-1, j-1] + \mathbb{1}(a_i \neq b_j) \}$$

	∅	Z	I	E	G	E
∅	0	1	2	3	4	5
T	1	1	2	3	4	5
I	2	2	1	2	3	4
G	3	3	2	2	2	3
E	4	4	3	2	3	2
R	5	5	4	3	3	3

Editierschritte: von rechts unten nach links oben, der Rekursion folgend. Bottom-Up Beschreibung des Algorithmus: Übung

Bottom-Up DP Algorithmus ED]

Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle $E[0, \dots, m][0, \dots, n]$. $E[i, j]$: Minimaler Editierabstand der Zeichenketten (a_1, \dots, a_i) und (b_1, \dots, b_j)

Berechnung eines Eintrags

- 2 $E[0, i] \leftarrow i \forall 0 \leq i \leq m, E[j, 0] \leftarrow j \forall 0 \leq j \leq n$. Berechnung von $E[i, j]$ sonst mit $E[i, j] = \min\{\text{del}(a_i) + E(i-1, j), \text{ins}(b_j) + E(i, j-1), \text{repl}(a_i, b_j) + E(i-1, j-1)\}$

Bottom-Up DP Algorithmus ED

Berechnungsreihenfolge

- 3 Abhängigkeiten berücksichtigen: z.B. Zeilen aufsteigend und innerhalb von Zeilen Spalten aufsteigend.

Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4 Beginne bei $j = m, i = n$. Falls $E[i, j] = \text{repl}(a_i, b_j) + E(i-1, j-1)$ gilt, gib $a_i \rightarrow b_j$ aus und fahre fort mit $(j, i) \leftarrow (j-1, i-1)$; sonst, falls $E[i, j] = \text{del}(a_i) + E(i-1, j)$ gib $\text{del}(a_i)$ aus fahre fort mit $j \leftarrow j-1$; sonst, falls $E[i, j] = \text{ins}(b_j) + E(i, j-1)$, gib $\text{ins}(b_j)$ aus und fahre fort mit $i \leftarrow i-1$. Terminiere für $i = 0$ und $j = 0$.

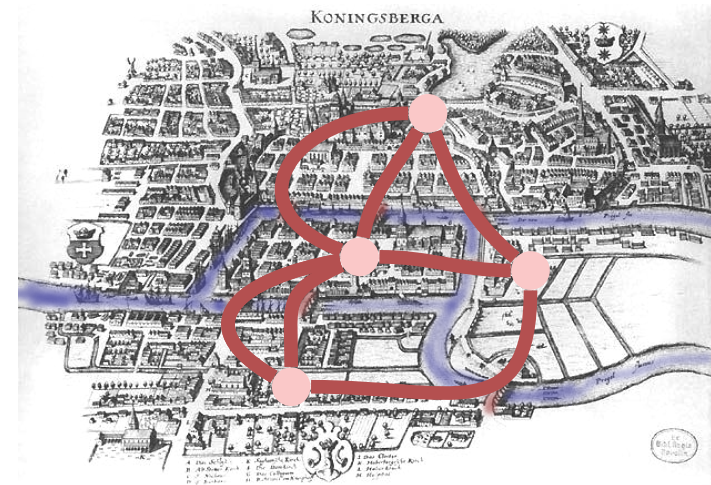
305

306

13. Graphen

Notation, Repräsentation, Traversieren (DFS, BFS), Topologisches Sortieren [Ottman/Widmayer, Kap. 9.1 - 9.4, Cormen et al, Kap. 22]

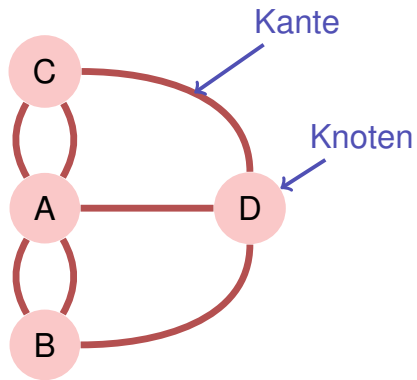
Königsberg 1736



307

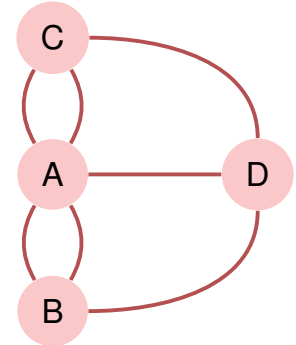
308

[Multi]Graph



Zyklen

- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst *Eulerscher Kreis*.
- Eulerzyklus \Leftrightarrow jeder Knoten hat gerade Anzahl Kanten (jeder Knoten hat einen *geraden Grad*).

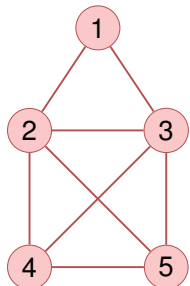


" \Rightarrow " ist sofort klar, " \Leftarrow " ist etwas schwieriger, aber auch elementar.

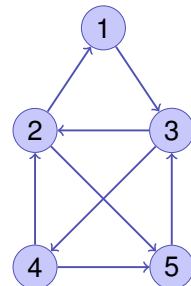
309

310

Notation



ungerichtet



gerichtet

$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$$

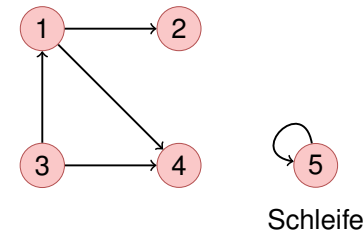
$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 3), (2, 1), (2, 5), (3, 2), (3, 4), (4, 2), (4, 5), (5, 3)\}$$

311

Notation

Ein *gerichteter Graph* besteht aus einer Menge $V = \{v_1, \dots, v_n\}$ von Knoten (*Vertices*) und einer Menge $E \subseteq V \times V$ von Kanten (*Edges*). Gleiche Kanten dürfen nicht mehrfach enthalten sein.

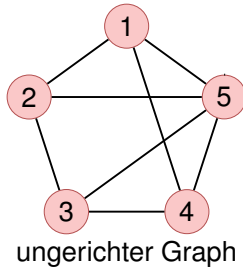


Schleife

312

Notation

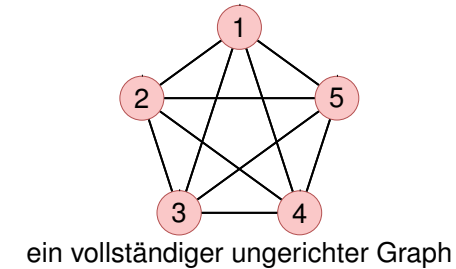
Ein *ungerichteter Graph* besteht aus einer Menge $V = \{v_1, \dots, v_n\}$ von Knoten und einer Menge $E \subseteq \{\{u, v\} | u, v \in V\}$ von Kanten. Kanten dürfen nicht mehrfach enthalten sein.²²



²²Im Gegensatz zum Eingangsbeispiel – dann Multigraph genannt.

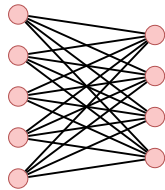
Notation

Ein ungerichteter Graph $G = (V, E)$ ohne Schleifen in dem jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist, heisst *vollständig*.



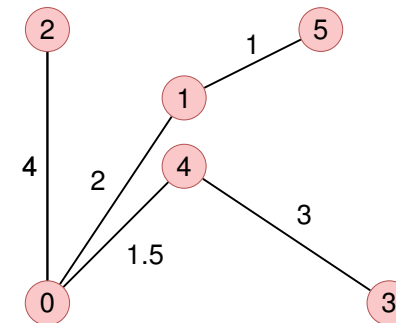
Notation

Ein Graph, bei dem V so in disjunkte U und W aufgeteilt werden kann, dass alle $e \in E$ einen Knoten in U und einen in W haben heisst *bipartit*.



Notation

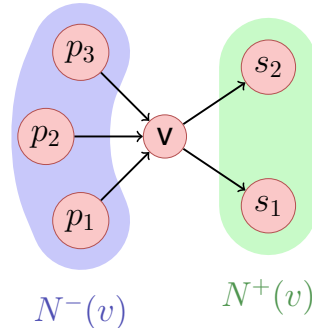
Ein *gewichteter Graph* $G = (V, E, c)$ ist ein Graph $G = (V, E)$ mit einer *Kantengewichtsfunktion* $c : E \rightarrow \mathbb{R}$. $c(e)$ heisst *Gewicht* der Kante e .



Notation

Für gerichtete Graphen $G = (V, E)$

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $(v, w) \in E$
- **Vorgängermenge** von $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
- **Nachfolgermenge**: $N^+(v) := \{u \in V | (v, u) \in E\}$

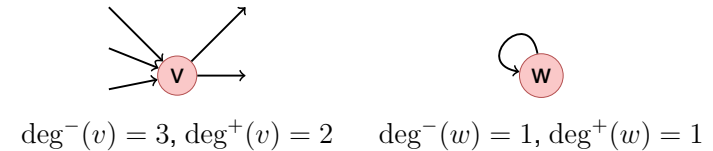


317

Notation

Für gerichtete Graphen $G = (V, E)$

- **Eingangsgrad**: $\deg^-(v) = |N^-(v)|$,
- **Ausgangsgrad**: $\deg^+(v) = |N^+(v)|$

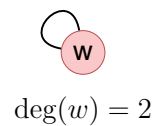
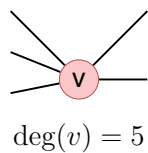


318

Notation

Für ungerichtete Graphen $G = (V, E)$:

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $\{v, w\} \in E$
- **Nachbarschaft** von $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$
- **Grad** von v : $\deg(v) = |N(v)|$ mit Spezialfall Schleifen: erhöhen Grad um 2.



319

Beziehung zwischen Knotengraden und Kantenanzahl

In jedem Graphen $G = (V, E)$ gilt

- 1 $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, falls G gerichtet
- 2 $\sum_{v \in V} \deg(v) = 2|E|$, falls G ungerichtet.

320

Wege

- **Weg**: Sequenz von Knoten $\langle v_1, \dots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \dots k\}$ eine Kante von v_i nach v_{i+1} existiert.
- **Länge** des Weges: Anzahl enthaltene Kanten k .
- **Gewicht** des Weges (in gewichteten Graphen): $\sum_{i=1}^k c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^k c(\{v_i, v_{i+1}\})$)
- **Pfad** (auch: einfacher Pfad): Weg der keinen Knoten mehrfach verwendet.

321

Zusammenhang

- Ungerichteter Graph heisst **zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **stark zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **schwach zusammenhängend**, wenn der entsprechende ungerichtete Graph zusammenhängend ist.

322

Einfache Beobachtungen

- Allgemein: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- Zusammenhängender Graph: $|E| \in \Omega(|V|)$
- Vollständiger Graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (ungerichtet)
- Maximal $|E| = |V|^2$ (gerichtet), $|E| = \frac{|V| \cdot (|V|+1)}{2}$ (ungerichtet)

323

Zyklen

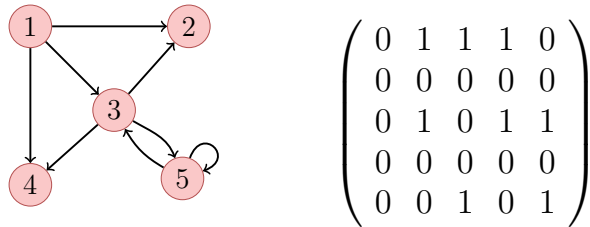
- **Zyklus**: Weg $\langle v_1, \dots, v_{k+1} \rangle$ mit $v_1 = v_{k+1}$
- **Kreis**: Zyklus mit paarweise verschiedenen v_1, \dots, v_k , welcher keine Kante mehrfach verwendet.
- **Kreisfrei (azyklisch)**: Graph ohne jegliche Kreise.

Eine Folgerung: Ungerichtete Graphen können keinen Kreis der Länge 2 enthalten (Schleifen haben Länge 1).

324

Repräsentation mit Matrix

Graph $G = (V, E)$ mit Knotenmenge v_1, \dots, v_n gespeichert als **Adjazenzmatrix** $A_G = (a_{ij})_{1 \leq i, j \leq n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j .

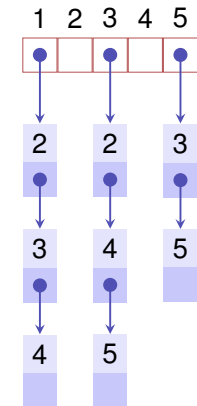
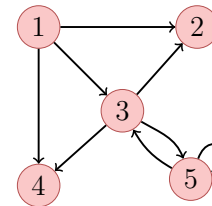


Speicherbedarf $\Theta(|V|^2)$. A_G ist symmetrisch, wenn G ungerichtet.

325

Repräsentation mit Liste

Viele Graphen $G = (V, E)$ mit Knotenmenge v_1, \dots, v_n haben deutlich weniger als n^2 Kanten. Repräsentation mit **Adjazenzliste**: Array $A[1], \dots, A[n]$, A_i enthält verkettete Liste aller Knoten in $N^+(v_i)$.



Speicherbedarf $\Theta(|V| + |E|)$.

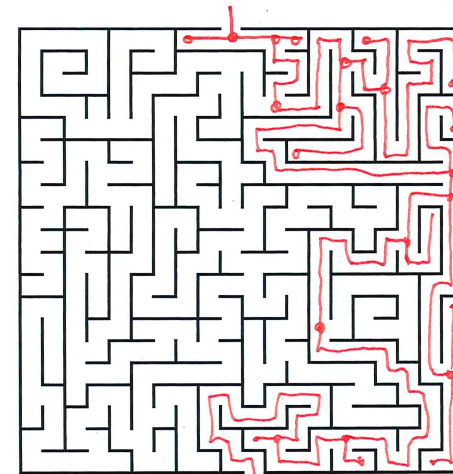
326

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen	$\Theta(1)$	$\Theta(\deg^+ v)$

327

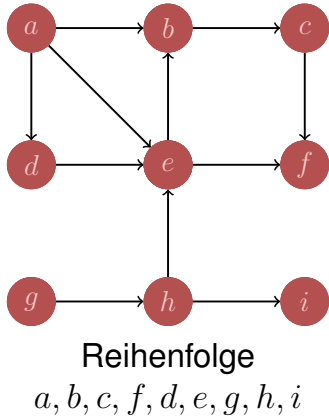
Tiefensuche



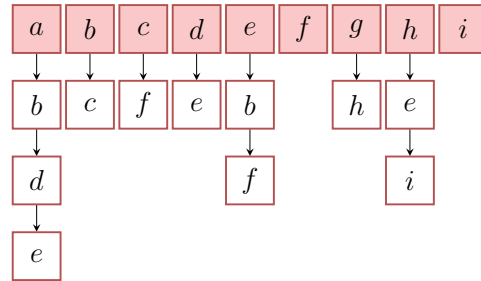
328

Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



Adjazenzliste



329

Farben

Konzeptuelle Färbung der Knoten

- **Weiss:** Knoten wurde noch nicht entdeckt.
- **Grau:** Knoten wurde entdeckt und zur Traversierung vorgemerkt / in Bearbeitung.
- **Schwarz:** Knoten wurde entdeckt und vollständig bearbeitet

330

Algorithmus Tiefensuche DFS-Visit(G, v)

Input: Graph $G = (V, E)$, Knoten v .

```

v.color ← grey
foreach w ∈ N+(v) do
  if w.color = white then
    DFS-Visit(G, w)
v.color ← black
    
```

Tiefensuche ab Knoten v . Laufzeit (ohne Rekursion): $\Theta(\deg^+ v)$

331

Algorithmus Tiefensuche DFS-Visit(G)

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

```

foreach v ∈ V do
  v.color ← white
foreach v ∈ V do
  if v.color = white then
    DFS-Visit(G, v)
    
```

Tiefensuche für alle Knoten eines Graphen. Laufzeit
 $\Theta(|V| + \sum_{v \in V} (\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

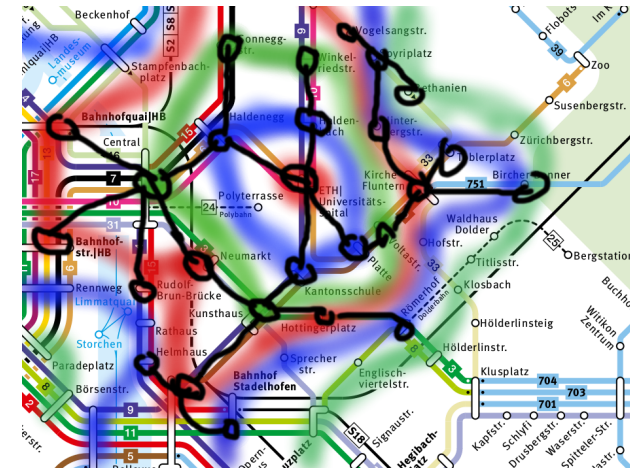
332

Interpretation der Farben

Beim Traversieren des Graphen wird ein Baum (oder Wald) aufgebaut. Beim Entdecken von Knoten gibt es drei Fälle

- Weisser Knoten: neue Baumkante
- Grauer Knoten: Zyklus („Rückwärtskante“)
- Schwarzer Knoten: Vorwärts-/Seitwärtskante

Breitensuche

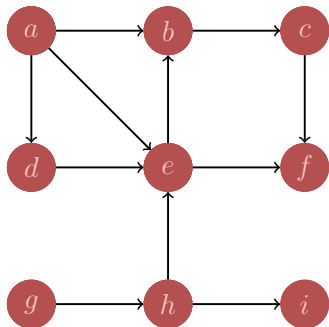


333

334

Graphen Traversieren: Breitensuche

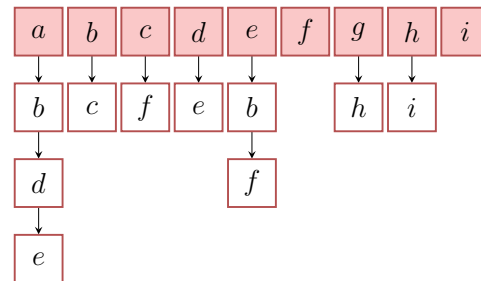
Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Reihenfolge

$a, b, d, e, c, f, g, h, i$

Adjazenzliste



(Iteratives) BFS-Visit(G, v)

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

```

Queue  $Q \leftarrow \emptyset$ 
 $v.color \leftarrow grey$ 
enqueue( $Q, v$ )
while  $Q \neq \emptyset$  do
     $w \leftarrow dequeue(Q)$ 
    foreach  $c \in N^+(w)$  do
        if  $c.color = white$  then
             $c.color \leftarrow grey$ 
            enqueue( $Q, c$ )
     $w.color \leftarrow black$ 
    
```

Algorithmus kommt mit $\mathcal{O}(|V|)$ Extraplatz aus.

335

336

Rahmenprogramm BFS-Visit(G)

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

```

foreach  $v \in V$  do
   $v.color \leftarrow white$ 
foreach  $v \in V$  do
  if  $v.color = white$  then
    BFS-Visit( $G, v$ )
    
```

Breitensuche für alle Knoten eines Graphen. Laufzeit $\Theta(|V| + |E|)$.

Topologisches Sortieren

	A	B	C	D	E	F	G	H	I
1		Task 1	Task 2	Task 3	Task 4	Total		Note	
2	TOTAL	8	8	10	10	36			
3	Arleen	4	5	6	9	24		4	
4	Hans	1	3	2	3	9		1.5	
5	Mike	2	7	5	4	18		3	
6	Selina	6	5	8	2	21		3.5	
7									
8				Durchschnitt		18		3	
9									
10									
11									
12									
13									
14									

Auswertungsreihenfolge?

Topologische Sortierung

Topologische Sortierung eines azyklischen gerichteten Graphen

$G = (V, E)$:

Bijektive Abbildung

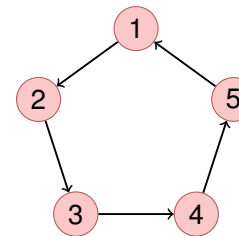
$$\text{ord} : V \rightarrow \{1, \dots, |V|\}$$

so dass

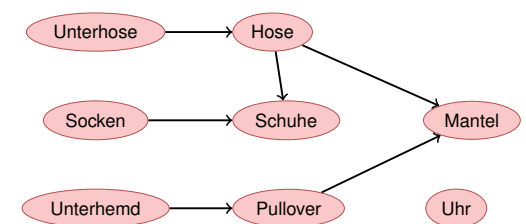
$$\text{ord}(v) < \text{ord}(w) \quad \forall (v, w) \in E.$$

Identifizieren Wert i mit dem Element $v_i := \text{ord}^{-1}(i)$. Topologische Sortierung $\hat{=} \langle v_1, \dots, v_{|V|} \rangle$.

(Gegen-)Beispiele



Zyklischer Graph: kann nicht topologisch sortiert werden.



Eine mögliche Topologische Sortierung des Graphen: Unterhemd, Pullover, Unterhose, Uhr, Hose, Mantel, Socken, Schuhe

Beobachtung

Theorem

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist

Beweis " \Rightarrow ": Wenn G einen Kreis besitzt, so besitzt er keine topologische Sortierung. Denn in einem Kreis $\langle v_{i_1}, \dots, v_{i_m} \rangle$ gälte $v_{i_1} < \dots < v_{i_m} < v_{i_1}$.

341

Induktiver Beweis Gegenrichtung

- Anfang ($n = 1$): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze $\text{ord}(v_1) = 1$.
- Hypothese: Graph mit n Knoten kann topologisch sortiert werden.
- Schritt ($n \rightarrow n + 1$):
 - 1 G enthält einen Knoten v_q mit Eingangsgrad $\text{deg}^-(v_q) = 0$. Andernfalls verfolge iterativ Kanten rückwärts – nach spätestens $n + 1$ Iterationen würde man einen Knoten besuchen, welcher bereits besucht wurde. Widerspruch zur Zyklenfreiheit.
 - 2 Graph ohne Knoten v_q und ohne dessen Eingangskanten kann nach Hypothese topologisch sortiert werden. Verwende diese Sortierung, setze $\text{ord}(v_i) \leftarrow \text{ord}(v_i) + 1$ für alle $i \neq q$ und setze $\text{ord}(v_q) \leftarrow 1$.

342

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

- 1 Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
- 2 Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.
- 3 Setze $\text{ord}(v_q) \leftarrow d$.
- 4 Entferne v_q und seine Kanten von G .
- 5 Wenn $V \neq \emptyset$, dann $d \leftarrow d + 1$, gehe zu Schritt 1.

Laufzeit im schlechtesten Fall: $\Theta(|V|^2)$.

343

Verbesserung

Idee?

Berechne die Eingangsgrade der Knoten im Voraus und durchlaufe dann jeweils die Knoten mit Eingangsgrad 0 die Eingangsgrade der Nachfolgeknoten korrigierend.

344

Algorithmus Topological-Sort(G)

Input: Graph $G = (V, E)$.

Output: Topologische Sortierung ord

Stack $S \leftarrow \emptyset$

foreach $v \in V$ **do** $A[v] \leftarrow 0$

foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Eingangsgrade berechnen

foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Merke Nodes mit Eingangsgrad 0

$i \leftarrow 1$

while $S \neq \emptyset$ **do**

$v \leftarrow \text{pop}(S)$; ord[v] $\leftarrow i$; $i \leftarrow i + 1$ // Wähle Knoten mit Eingangsgrad 0

foreach $(v, w) \in E$ **do** // Verringere Eingangsgrad der Nachfolger

$A[w] \leftarrow A[w] - 1$

if $A[w] = 0$ **then** push(S, w)

if $i = |V| + 1$ **then return** ord **else return** "Cycle Detected"

345

Algorithmus Korrektheit

Theorem

Sei $G = (V, E)$ ein gerichteter, kreisfreier Graph. Der Algorithmus $\text{TopologicalSort}(G)$ berechnet in Zeit $\Theta(|V| + |E|)$ eine topologische Sortierung ord für G .

Beweis: folgt im wesentlichen aus vorigem Theorem:

- 1 Eingangsgrad verringern entspricht Knotenentfernen.
- 2 Im Algorithmus gilt für jeden Knoten v mit $A[v] = 0$ dass entweder der Knoten Eingangsgrad 0 hat oder dass zuvor alle Vorgänger einen Wert ord[u] $\leftarrow i$ zugewiesen bekamen und somit ord[v] $>$ ord[u] für alle Vorgänger u von v . Knoten werden nur einmal auf den Stack gelegt.
- 3 Laufzeit: Inspektion des Algorithmus (mit Argumenten wie beim Traversieren).

346

Algorithmus Korrektheit

Theorem

Sei $G = (V, E)$ ein gerichteter, nicht kreisfreier Graph. Der Algorithmus $\text{TopologicalSort}(G)$ terminiert in Zeit $\Theta(|V| + |E|)$ und detektiert Zyklus.

Beweis: Sei $\langle v_{i_1}, \dots, v_{i_k} \rangle$ ein Kreis in G . In jedem Schritt des Algorithmus bleibt $A[v_{i_j}] \geq 1$ für alle $j = 1, \dots, k$. Also werden k Knoten nie auf den Stack gelegt und somit ist zum Schluss $i \leq |V| + 1 - k$.

Die Laufzeit des zweiten Teils des Algorithmus kann kürzer werden, jedoch kostet die Berechnung der Eingangsgrade bereits $\Theta(|V| + |E|)$.

347

14. Kürzeste Wege

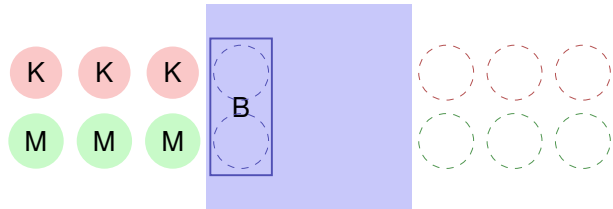
Motivation, Dijkstras Algorithmus auf Distanzgraphen, Algorithmus von Bellman-Ford

[Ottman/Widmayer, Kap. 9.5.1-9.5.2 Cormen et al, Kap. 24.1-24.3]

348

Flussüberquerung (Missionare und Kannibalen)

Problem: Drei Kannibalen und drei Missionare stehen an einem Ufer eines Flusses. Ein dort bereitstehendes Boot fasst maximal zwei Personen. Zu keiner Zeit dürfen an einem Ort (Ufer oder Boot) mehr Kannibalen als Missionare sein. Wie kommen die Missionare und Kannibalen möglichst schnell über den Fluss? ²³



²³Es gibt leichte Variationen dieses Problems, es ist auch äquivalent zum Problem der eifersüchtigen Ehemänner

349

Formulierung als Graph

Zähle alle erlaubten Konfigurationen als Knoten auf und verbinde diese mit einer Kante, wenn Überfahrt möglich ist. Das Problem ist dann ein Problem des kürzesten Pfades

Beispiel

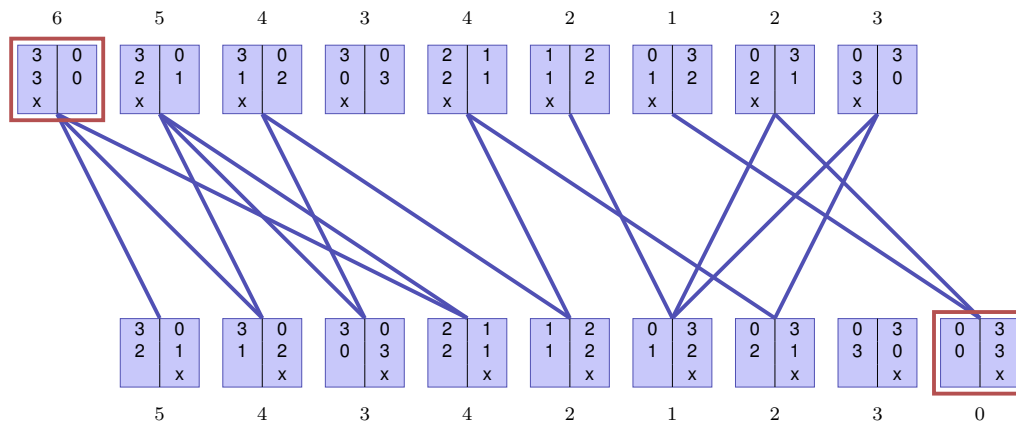
	links	rechts		links	rechts	
Missionare	3	0	Überfahrt möglich	Missionare	2	1
Kannibalen	3	0		Kannibalen	2	1
Boot	x			Boot		x

6 Personen am linken Ufer

4 Personen am linken Ufer

350

Das ganze Problem als Graph



351

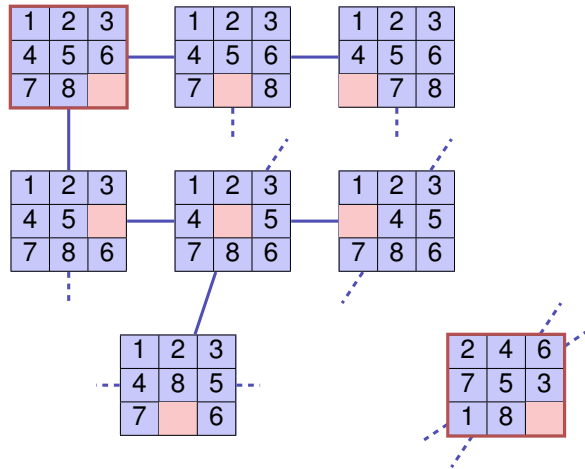
Anderes Beispiel: Schiebepuzzle

Wollen die schnellste Lösung finden für



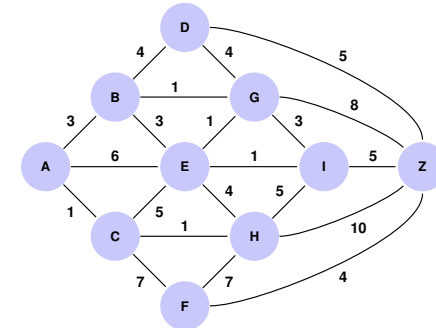
352

Problem als Graph



Routenfinder

Gegeben Städte A - Z und Distanzen zwischen den Städten.



Was ist der kürzeste Weg von A nach Z?

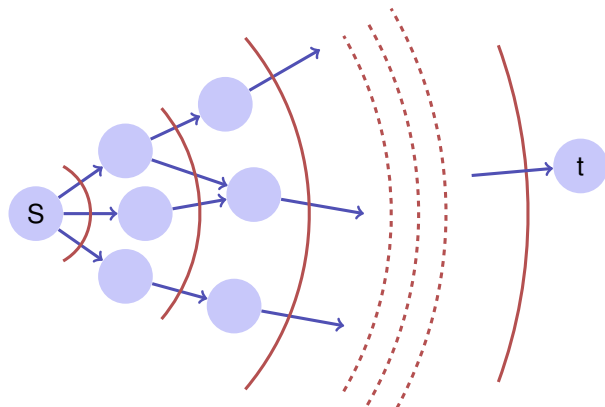
353

354

Einfachster Fall

Konstantes Kantengewicht 1 (oBdA)

Lösung: Breitensuche



355

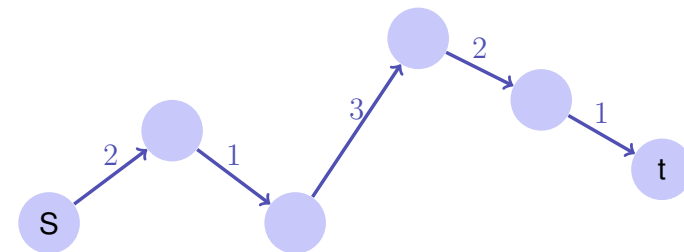
Gewichtete Graphen

Gegeben: $G = (V, E, c)$, $c : E \rightarrow \mathbb{R}$, $s, t \in V$.

Gesucht: Länge (Gewicht) eines kürzesten Weges von s nach t .

Weg: $p = \langle s = v_0, v_1, \dots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ ($0 \leq i < k$)

Gewicht: $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Weg mit Gewicht 9

356

Kürzeste Wege

Notation: Wir schreiben

$$u \overset{p}{\rightsquigarrow} v \quad \text{oder} \quad p : u \rightsquigarrow v$$

und meinen einen Weg p von u nach v

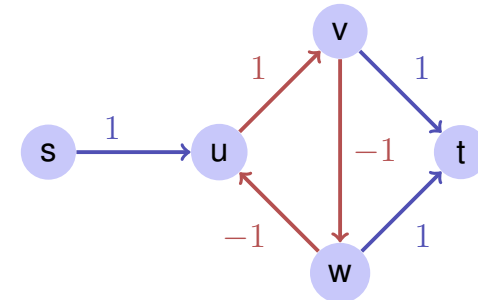
Notation: $\delta(u, v)$ = Gewicht eines kürzesten Weges von u nach v :

$$\delta(u, v) = \begin{cases} \infty & \text{kein Weg von } u \text{ nach } v \\ \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{sonst} \end{cases}$$

357

Beobachtungen (1)

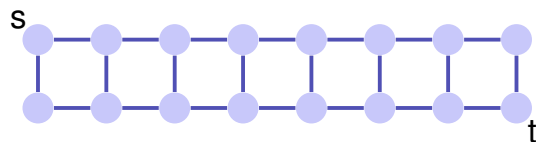
Es gibt Situationen, in denen kein kürzester Weg existiert: negative Zyklen könnten auftreten.



358

Beobachtungen (2)

Es kann exponentiell viele Wege geben.



(mindestens $2^{|V|/2}$ Wege von s nach t)

⇒ Alle Wege probieren ist zu ineffizient.

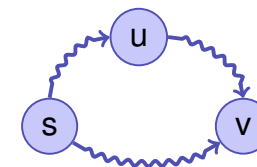
359

Beobachtungen (3)

Dreiecksungleichung

Für alle $s, u, v \in V$:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$$



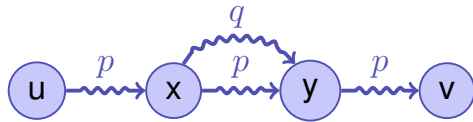
Ein kürzester Weg von s nach v (ohne weitere Einschränkungen) kann nicht länger sein als ein kürzester Weg von s nach v , der u enthalten muss.

360

Beobachtungen (4)

Optimale Substruktur

Teilpfade von kürzesten Pfaden sind kürzeste Pfade: Sei $p = \langle v_0, \dots, v_k \rangle$ ein kürzester Pfad von v_0 nach v_k . Dann ist jeder der Teilpfade $p_{ij} = \langle v_i, \dots, v_j \rangle$ ($0 \leq i < j \leq k$) ein kürzester Pfad von v_i nach v_j .



Wäre das nicht so, könnte man einen der Teilpfade kürzen, Widerspruch zur Voraussetzung.

361

Beobachtungen (5)

Kürzeste Wege enthalten keine Zyklen

- 1 Kürzester Weg enthält negativen Zyklus: es existiert kein kürzester Weg. Widerspruch.
- 2 Weg enthält positiven Zyklus: Weglassen des positiven Zyklus kann den Weg verkürzen: Widerspruch
- 3 Weg enthält Zyklus vom Gewicht 0: Weglassen des Zyklus verändert das Pfadgewicht nicht. Weglassen (Konvention).

362

Zutaten für einen Algorithmus

Gesucht: Kürzeste Wege von einem Startknoten s aus.

- Gewicht des kürzesten bisher gefundenen Pfades

$$d_s : V \rightarrow \mathbb{R}$$

Zu Beginn: $d_s[v] = \infty$ für alle Knoten $v \in V$.

Ziel: $d_s[v] = \delta(s, v)$ für alle $v \in V$.

- Vorgänger eines Knotens

$$\pi_s : V \rightarrow V$$

Zu Beginn $\pi_s[v]$ undefiniert für jeden Knoten $v \in V$

363

Allgemeiner Algorithmus

- 1 Initialisiere d_s und π_s : $d_s[v] = \infty$, $\pi_s[v] = \text{null}$ für alle $v \in V$
- 2 Setze $d_s[s] \leftarrow 0$
- 3 Wähle eine Kante $(u, v) \in E$
Relaxiere (u, v) :
if $d_s[v] > d_s[u] + c(u, v)$ then
 $d_s[v] \leftarrow d_s[u] + c(u, v)$
 $\pi_s[v] \leftarrow u$
- 4 Wiederhole 3 bis nichts mehr relaxiert werden kann.
(bis $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

364

Relaxieren ist sicher

Zu jeder Zeit gilt in obigem Algorithmus

$$d_s[v] \geq \delta(s, v) \quad \forall v \in V$$

Im Relaxierschritt:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v) \quad \text{[Dreiecksungleichung].}$$

$$\delta(s, u) \leq d_s[u] \quad \text{[Induktionsvoraussetzung].}$$

$$\delta(u, v) \leq c(u, v) \quad \text{[Minimalität von } \delta \text{]}$$

$$\Rightarrow d_s[u] + c(u, v) \geq \delta(s, v)$$

$$\Rightarrow \min\{d_s[v], d_s[u] + c(u, v)\} \geq \delta(s, v)$$

365

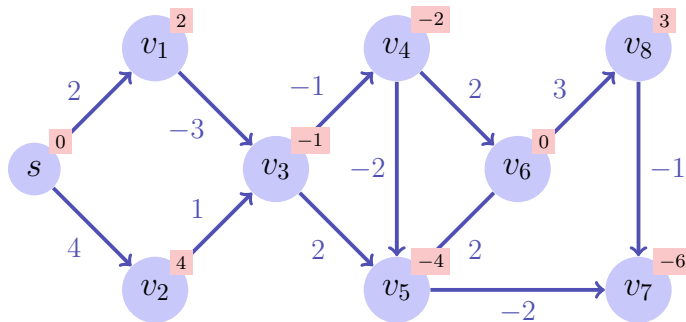
Zentrale Frage

Wie / in welcher Reihenfolge wählt man die Kanten in obigem Algorithmus?

366

Spezialfall: Gerichteter Azyklischer Graph (DAG)

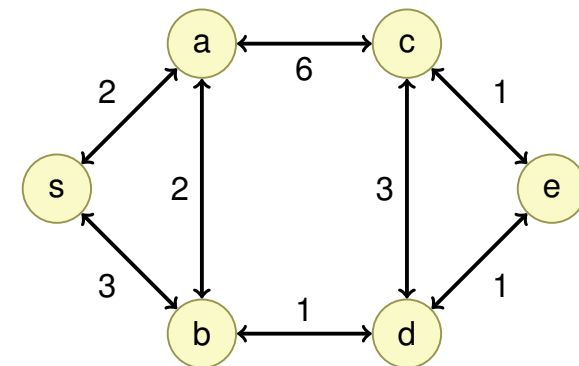
DAG \Rightarrow Topologische Sortierung liefert optimale Besuchsreihenfolge



Top. Sortieren: \Rightarrow Reihenfolge $s, v_1, v_2, v_3, v_4, v_6, v_5, v_8, v_7$.

367

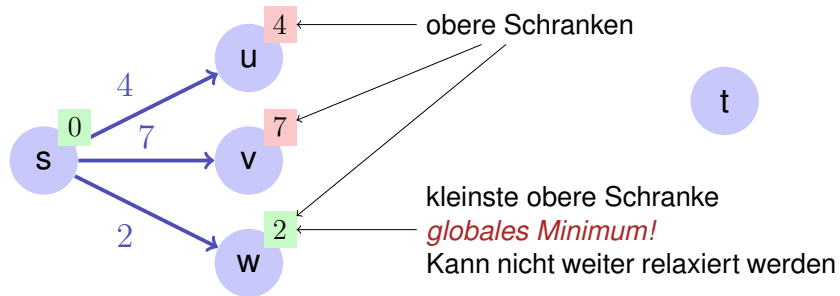
Annahme (vorübergehend)



Alle Gewichte von G sind *positiv*.

368

Beobachtung (Dijkstra)

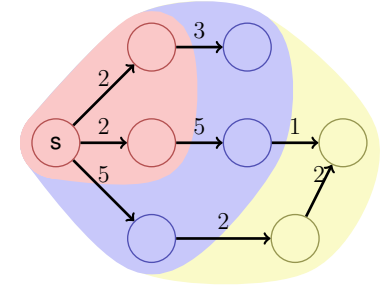


369

Grundidee

Menge V aller Knoten wird unterteilt in

- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \bigcup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt wurden.

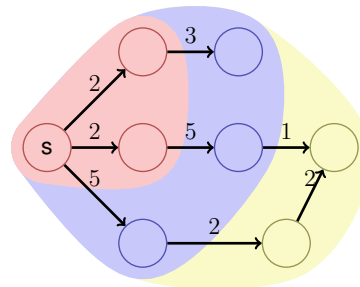


370

Induktion

Induktion über $|M|$: Wähle Knoten aus R mit kleinster oberer Schranke. Nimm r zu M hinzu, und update R und U .

Korrektheit: Ist innerhalb einer "Wellenfront" einmal ein Knoten mit minimalem Pfadgewicht w gefunden, kann kein Pfad über später gefundene Knoten (mit Gewicht $\geq w$) zu einer Verbesserung führen.



371

Algorithmus Dijkstra(G, s)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$

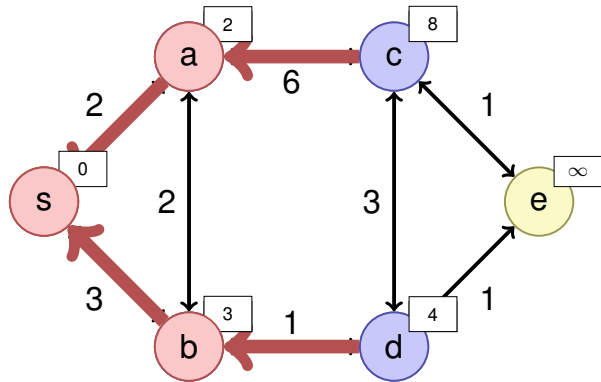
Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

```

foreach  $u \in V$  do
   $d_s[u] \leftarrow \infty$ ;  $\pi_s[u] \leftarrow \text{null}$ 
 $d_s[s] \leftarrow 0$ ;  $R \leftarrow \{s\}$ 
while  $R \neq \emptyset$  do
   $u \leftarrow \text{ExtractMin}(R)$ 
  foreach  $v \in N^+(u)$  do
    if  $d_s[u] + c(u, v) < d_s[v]$  then
       $d_s[v] \leftarrow d_s[u] + c(u, v)$ 
       $\pi_s[v] \leftarrow u$ 
       $R \leftarrow R \cup \{v\}$ 
    
```

372

Beispiel



$$M = \{s, a, b\}$$

$$R = \{c, d\}$$

$$U = \{e\}$$

Zur Implementation: Datenstruktur für R ?

Benötigte Operationen:

- Insert (Hinzunehmen zu R)
- ExtractMin (über R) und DecreaseKey (Update in R)

```

foreach  $v \in N^+(u)$  do
  if  $d_s[u] + c(u, v) < d_s[v]$  then
     $d_s[v] \leftarrow d_s[u] + c(u, v)$ 
     $\pi_s[v] \leftarrow u$ 
    if  $v \in R$  then
      DecreaseKey( $R, v$ )           // Update eines  $d(v)$  im Heap zu  $R$ 
    else
       $R \leftarrow R \cup \{v\}$     // Einfügen eines neuen  $d(v)$  im Heap zu  $R$ 
    
```

MinHeap!

373

374

DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in $\mathcal{O}(\log |V|)$
- Position im Heap?
 - Möglichkeit (a): Speichern am Knoten
 - Möglichkeit (b): Hashtabelle über Knoten
 - Möglichkeit (c): Knoten nach erfolgreichem Relaxieren erneut einfügen. Knoten beim Entnehmen als "deleted" kennzeichnen (Lazy Deletion).²⁴

²⁴Für die lazy deletion benötigt man ein Paar von Kante (oder Zielknoten) und Distanz

Laufzeit

- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert oder DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- Insgesamt: $\mathcal{O}(|E| \log |V|)$.

375

376

Allgemeine Bewertete Graphen

Verbesserungsschritt wie bisher, aber mit Rückgabewert:

$\text{Relax}(u, v)$ ($u, v \in V, (u, v) \in E$)

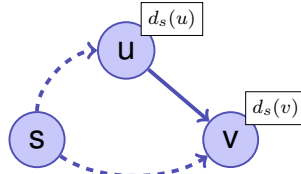
if $d_s[u] + c(u, v) < d_s[v]$ **then**

$d_s[v] \leftarrow d_s[u] + c(u, v)$

$\pi_s[v] \leftarrow u$

return true

return false



Problem: Zyklen mit negativen Gewichten können Weg verkürzen:
es muss keinen kürzesten Weg mehr geben

377

Dynamic Programming Ansatz (Bellman)

Induktion über Anzahl Kanten. $d_s[i, v]$: Kürzeste Weglänge von s nach v über maximal i Kanten.

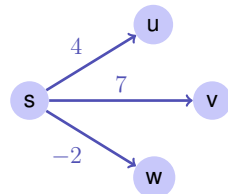
$$d_s[i, v] = \min\{d_s[i-1, v], \min_{(u,v) \in E} (d_s[i-1, u] + c(u, v))\}$$

$$d_s[0, s] = 0, d_s[0, v] = \infty \quad \forall v \neq s.$$

378

Dynamic Programming Ansatz (Bellman)

	s	\dots	v	\dots	w
0	0	∞	∞	∞	∞
1	0	∞	7	∞	-2
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$n-1$	0	\dots	\dots	\dots	\dots



Algorithmus: Iteriere über letzte Zeile bis die Relaxationsschritte keine Änderung mehr ergeben, maximal aber $n-1$ mal. Wenn dann noch Änderungen, dann gibt es keinen kürzesten Pfad.

379

Algorithmus Bellman-Ford(G, s)

Input: Graph $G = (V, E, c)$, Startpunkt $s \in V$

Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

foreach $u \in V$ **do**

$d_s[u] \leftarrow \infty; \pi_s[u] \leftarrow \text{null}$

$d_s[s] \leftarrow 0;$

for $i \leftarrow 1$ **to** $|V|$ **do**

$f \leftarrow \text{false}$

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

$f \leftarrow f \vee \text{Relax}(u, v)$

if $f = \text{false}$ **then return true**

return false;

380

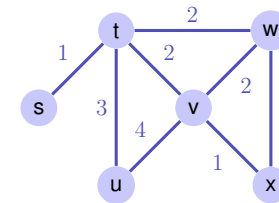
15. Minimale Spannbäume

Motivation, Greedy, Algorithmus von Kruskal, Allgemeine Regeln, Union-Find Struktur, Algorithmus von Jarnik, Prim, Dijkstra, [Ottman/Widmayer, Kap. 9.6, 6.2, 6.1, Cormen et al, Kap. 23, 19]

Problem

Gegeben: Ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$.

Gesucht: Minimaler Spannbaum $T = (V, E')$: zusammenhängender, zyklenfreier Teilgraph $E' \subset E$, so dass $\sum_{e \in E'} c(e)$ minimal.



381

382

Beispiele von Anwendungen

- Netzwerk-Design: finde das billigste / kürzeste Netz oder Leitungssystem, welches alle Knoten miteinander verbindet.
- Approximation einer Lösung des Travelling-Salesman Problems: finde einen möglichst kurzen Rundweg, welcher jeden Knoten einmal besucht. ²⁵

Greedy Verfahren

- Gierige Verfahren berechnen eine Lösung schrittweise, indem lokal beste Lösungen gewählt werden.
- Die meisten Probleme sind nicht mit einer greedy Strategie lösbar.
- Das Problem des Minimalen Spannbaumes kann mit einem gierigen Verfahren effizient gelöst werden.

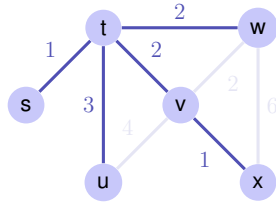
²⁵Der beste bekannte Algorithmus zur exakten Lösung des TS Problems hat exponentielle Laufzeit

383

384

Greedy Idee (Kruskal, 1956)

Konstruiere T indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

Algorithmus MST-Kruskal(G)

Input: Gewichteter Graph $G = (V, E, c)$

Output: Minimaler Spannbaum mit Kanten A .

Sortiere Kanten nach Gewicht $c(e_1) \leq \dots \leq c(e_m)$

$A \leftarrow \emptyset$

for $k = 1$ **to** $|E|$ **do**

if $(V, A \cup \{e_k\})$ kreisfrei **then**

$A \leftarrow A \cup \{e_k\}$

return (V, A, c)

385

386

[Korrektheit]

Zu jedem Zeitpunkt ist (V, A) ein Wald, eine Menge von Bäumen.

MST-Kruskal betrachtet jede Kante e_k einmal und wählt e_k oder verwirft e_k

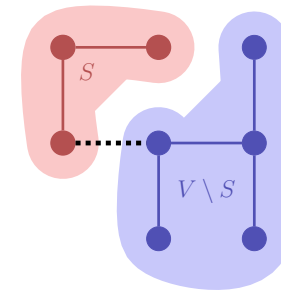
Notation (Momentaufnahme im Algorithmus)

- A : Menge gewählte Kanten
- R : Menge verworfener Kanten
- U : Menge der noch unentschiedenen Kanten

[Schnitt]

Ein Schnitt von G ist eine Partition $S, V \setminus S$ von V . ($S \subseteq V$).

Eine Kante kreuzt einen Schnitt, wenn einer Ihrer Endpunkte in S und der andere in $V \setminus S$ liegt.



(not shown in class) 387

(not shown in class) 388

[Regeln]

- 1 Auswahlregel: Wähle einen Schnitt, den keine gewählte Kante kreuzt. Unter allen unentschiedenen Kanten, welche den Schnitt kreuzen, wähle die mit minimalem Gewicht.
- 2 Verwerfregel: Wähle einen Kreis ohne verworfene Kanten. Unter allen unentschiedenen Kanten im Kreis verwerfe die mit maximalem Gewicht.

(not shown in class) 389

[Regeln]

Kruskal wendet beide Regeln an:

- 1 Ein gewähltes e_k verbindet zwei Zusammenhangskomponenten, sonst würde ein Kreis erzeugt werden. e_k ist beim Verbinden minimal, man kann also einen Schnitt wählen, den e_k mit minimalem Gewicht kreuzt.
- 2 Ein verworfenes e_k ist Teil eines Kreises. Innerhalb des Kreises hat e_k maximales Gewicht.

(not shown in class) 390

[Korrektheit]

Theorem

Jeder Algorithmus, welcher schrittweise obige Regeln anwendet bis $U = \emptyset$ ist korrekt.

Folgerung: MST-Kruskal ist korrekt.

(not shown in class) 391

[Auswahlvariante]

Invariante: Es gibt stets einen minimalen Spannbaum, der alle gewählten und keine der verworfenen Kanten enthält.

Wenn die beiden Regeln die Invariante erhalten, dann ist der Algorithmus sicher korrekt. Induktion:

- Zu Beginn: $U = E, R = A = \emptyset$. Invariante gilt offensichtlich.
- Invariante bleibt nach jedem Schritt des Algorithmus erhalten.
- Am Ende: $U = \emptyset, R \cup A = E \Rightarrow (V, A)$ ist Spannbaum.

Beweis des Theorems: zeigen nun, dass die beiden Regeln die Invariante erhalten.

(not shown in class) 392

[Auswahlregel erhält Invariante]

Es gibt stets einen minimalen Spannbaum T , der alle gewählten und keine der verworfenen Kanten enthält.

Wähle einen Schnitt, den keine gewählte Kante kreuzt. Unter allen unentschiedenen Kanten, welche den Schnitt kreuzen, wähle eine Kante e mit minimalem Gewicht.

- Fall 1: $e \in T$ (fertig)
- Fall 2: $e \notin T$. Dann hat $T \cup \{e\}$ einen Kreis, der e enthält. Kreis muss eine zweite Kante e' enthalten, welche den Schnitt auch kreuzt.²⁶ Da $e' \notin T$ ist $e' \in U$. Somit $c(e) \leq c(e')$ und $T' = T \setminus \{e'\} \cup \{e\}$ ist auch minimaler Spannbaum (und $c(e) = c(e')$).

²⁶Ein solcher Kreis enthält mindestens einen Knoten in S und einen in $V \setminus S$ und damit mindestens zwei Kanten zwischen S und $V \setminus S$.

(not shown in class) 393

[Verwerfregel erhält Invariante]

Es gibt stets einen minimalen Spannbaum T , der alle gewählten und keine der verworfenen Kanten enthält.

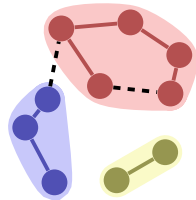
Wähle einen Kreis ohne verworfene Kanten. Unter allen unentschiedenen Kanten im Kreis verwerfe die Kante e mit maximalem Gewicht.

- Fall 1: $e \notin T$ (fertig)
- Fall 2: $e \in T$. Entferne e von T , Das ergibt einen Schnitt. Diesen Schnitt muss eine weitere Kante e' aus dem Kreis kreuzen. Da $c(e') \leq c(e)$ ist $T' = T \setminus \{e\} \cup \{e'\}$ auch minimal (und $c(e) = c(e')$).

(not shown in class) 394

Zur Implementation

Gegeben eine Menge von Mengen $i \equiv A_i \subset V$. Zur Identifikation von Schnitten und Kreisen: Zugehörigkeit der beiden Endpunkte einer Kante zu einer der Mengen.



395

Zur Implementation

Allgemeines Problem: Partition (Menge von Teilmengen) z.B. $\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$

Benötigt: Abstrakter Datentyp „Union-Find“ mit folgenden Operationen

- Make-Set(i): Hinzufügen einer neuen Menge i .
- Find(e): Name i der Menge, welche e enthält.
- Union(i, j): Vereinigung der Mengen mit Namen i und j .

396

Union-Find Algorithmus MST-Kruskal(G)

Input: Gewichteter Graph $G = (V, E, c)$

Output: Minimaler Spannbaum mit Kanten A .

Sortiere Kanten nach Gewicht $c(e_1) \leq \dots \leq c(e_m)$

$A \leftarrow \emptyset$

for $k = 1$ **to** $|V|$ **do**

 MakeSet(k)

for $k = 1$ **to** m **do**

$(u, v) \leftarrow e_k$

if Find(u) \neq Find(v) **then**

 Union(Find(u), Find(v))

$A \leftarrow A \cup e_k$

else

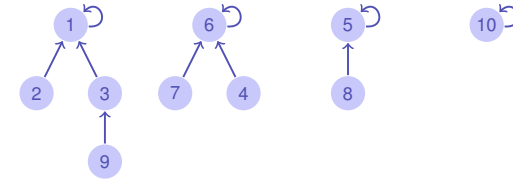
// konzeptuell: $R \leftarrow R \cup e_k$

return (V, A, c)

Implementation Union-Find

Idee: Baum für jede Teilmenge in der Partition, z.B.

$\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$

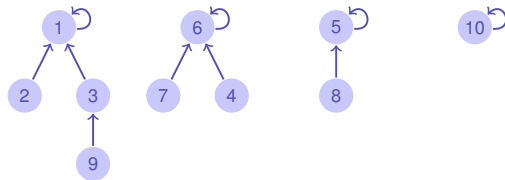


Baumwurzeln = Namen (Stellvertreter) der Mengen,
Bäume = Elemente der Mengen

397

398

Implementation Union-Find



Repräsentation als Array:

Index	1	2	3	4	5	6	7	8	9	10
Parent	1	1	1	6	5	6	5	5	3	10

Implementation Union-Find

Index	1	2	3	4	5	6	7	8	9	10
Parent	1	1	1	6	5	6	5	5	3	10

Make-Set(i) $p[i] \leftarrow i$; **return** i

Find(i) **while** ($p[i] \neq i$) **do** $i \leftarrow p[i]$
 return i

Union(i, j)²⁷ $p[j] \leftarrow i$;

²⁷ i und j müssen Namen (Wurzeln) der Mengen sein. Andernfalls verwende Union(Find(i),Find(j))

399

400

Optimierung der Laufzeit für Find

Baum kann entarten: Beispiel Union(8, 7), Union(7, 6), Union(6, 5), ...

Index	1	2	3	4	5	6	7	8	..
Parent	1	1	2	3	4	5	6	7	..

Laufzeit von Find im schlechtesten Fall in $\Theta(n)$.

401

Optimierung der Laufzeit für Find

Idee: Immer kleineren Baum unter grösseren Baum hängen.
Benötigt zusätzliche Grösseninformation (Array) g

Make-Set(i) $p[i] \leftarrow i; g[i] \leftarrow 1; \text{return } i$

Union(i, j)
 $\text{if } g[j] > g[i] \text{ then swap}(i, j)$
 $p[j] \leftarrow i$
 $\text{if } g[i] = g[j] \text{ then } g[i] \leftarrow g[i] + 1$

\Rightarrow Baumtiefe (und schlechteste Laufzeit für Find) in $\Theta(\log n)$

402

[Beobachtung]

Theorem

Obiges Verfahren Vereinigung nach Grösse konserviert die folgende Eigenschaft der Bäume: ein Baum mit Höhe h hat mindestens 2^h Knoten.

Unmittelbare Folgerung: Laufzeit Find = $\mathcal{O}(\log n)$.

(not shown in class) 403

[Beweis]

Induktion: nach Voraussetzung haben Teilbäume jeweils mindestens 2^{h_i} Knoten. ObdA: $h_2 \leq h_1$.

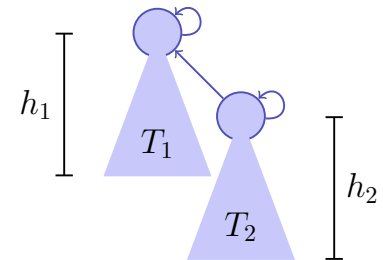
■ $h_2 < h_1$:

$$h(T_1 \oplus T_2) = h_1 \Rightarrow g(T_1 \oplus T_2) \geq 2^{h_1}$$

■ $h_2 = h_1$:

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$

$$\Rightarrow g(T_1 \oplus T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h(T_1 \oplus T_2)}$$



(not shown in class) 404

Weitere Verbesserung

Bei jedem Find alle Knoten direkt an den Wurzelknoten hängen.

Find(i):

```
 $j \leftarrow i$   
while ( $p[i] \neq i$ ) do  $i \leftarrow p[i]$   
while ( $j \neq i$ ) do  
   $t \leftarrow j$   
   $j \leftarrow p[j]$   
   $p[t] \leftarrow i$   
return  $i$ 
```

Laufzeit: amortisiert *fast* konstant (Inverse der Ackermannfunktion).²⁸

²⁸Wird hier nicht vertieft.

Laufzeit des Kruskal Algorithmus

- Sortieren der Kanten: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$.²⁹
 - Initialisieren der Union-Find Datenstruktur $\Theta(|V|)$
 - $|E| \times \text{Union}(\text{Find}(x), \text{Find}(y))$: $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.
- Insgesamt $\Theta(|E| \log |V|)$.

²⁹da G zusammenhängend: $|V| \leq |E| \leq |V|^2$

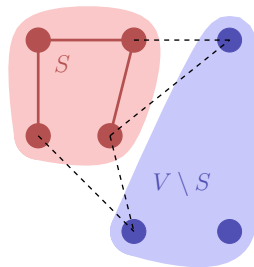
405

406

Algorithmus von Jarnik (1930), Prim, Dijkstra (1959)

Idee: Starte mit einem $v \in V$ und lasse von dort unter Verwendung der Auswahlregel einen Spannbaum wachsen:

```
 $A \leftarrow \emptyset$   
 $S \leftarrow \{v_0\}$   
for  $i \leftarrow 1$  to  $|V|$  do  
  Wähle billigste  $(u, v)$  mit  $u \in S, v \notin S$   
   $A \leftarrow A \cup \{(u, v)\}$   
   $S \leftarrow S \cup \{v\}$  // (Färbung)
```



Anmerkung: man benötigt keine Union-Find Datenstruktur. Es genügt, Knoten zu färben, sobald sie zu S hinzugenommen werden.

Laufzeit

Trivial $\mathcal{O}(|V| \cdot |E|)$.

Verbesserung (wie bei Dijkstras Kürzeste Pfade):

- Mit Min-Heap, Kosten:
 - Initialisierung (Knotenfärbung) $\mathcal{O}(|V|)$
 - $|V| \times \text{ExtractMin} = \mathcal{O}(|V| \log |V|)$,
 - $|E| \times \text{Insert oder DecreaseKey} = \mathcal{O}(|E| \log |V|)$,

$\mathcal{O}(|E| \cdot \log |V|)$

407

408

Motivation

16. Flüsse in Netzen

Flussnetzwerk, Maximaler Fluss, Schnitt, Restnetzwerk, Max-flow Min-cut Theorem, Ford-Fulkerson Methode, Edmonds-Karp Algorithmus, Maximales Bipartites Matching [Ottman/Widmayer, Kap. 9.7, 9.8.1], [Cormen et al, Kap. 26.1-26.3]

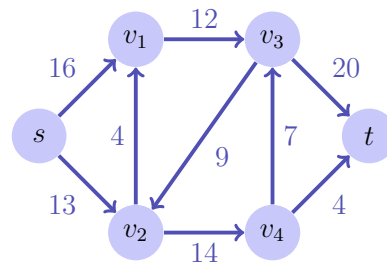
- Modelliere Fluss von Flüssigkeiten, Bauteile auf Fließbändern, Strom in elektrischen Netzen oder Information in Kommunikationsnetzwerken.
- Konnektivität von Kommunikationsnetzwerken, Bipartites Matching, Zirkulationen, Scheduling, Image Segmentation, Baseball Elimination...

409

410

Flussnetzwerk

- **Flussnetzwerk** $G = (V, E, c)$: gerichteter Graph mit **Kapazitäten**
- Antiparallele Kanten verboten: $(u, v) \in E \Rightarrow (v, u) \notin E$.
- Fehlen einer Kante (u, v) auch modelliert durch $c(u, v) = 0$.
- **Quelle** s und **Senke** t : spezielle Knoten. Jeder Knoten v liegt auf einem Pfad zwischen s und t : $s \rightsquigarrow v \rightsquigarrow t$



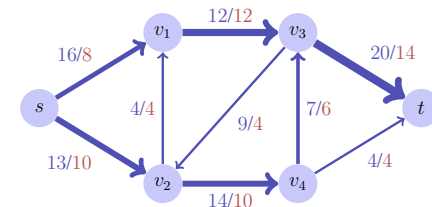
411

Fluss

Ein **Fluss** $f : V \times V \rightarrow \mathbb{R}$ erfüllt folgende Bedingungen:

- **Kapazitätsbeschränkung:** Für alle $u, v \in V$: $f(u, v) \leq c(u, v)$.
- **Schiefssymmetrie:** Für alle $u, v \in V$: $f(u, v) = -f(v, u)$.
- **Flusserhaltung:** Für alle $u \in V \setminus \{s, t\}$:

$$\sum_{v \in V} f(u, v) = 0.$$



Wert w des Flusses:
 $|f| = \sum_{v \in V} f(s, v)$.
 Hier $|f| = 18$.

412

Wie gross kann ein Fluss sein?

Begrenzende Faktoren: Schnitte

- ***s* von *t* trennender Schnitt**: Partitionierung von V in S und T mit $s \in S, t \in T$.
- **Kapazität** eines Schnittes: $c(S, T) = \sum_{v \in S, v' \in T} c(v, v')$
- **Minimaler Schnitt**: Schnitt mit minimaler Kapazität.
- **Fluss über Schnitt**: $f(S, T) = \sum_{v \in S, v' \in T} f(v, v')$

413

Implizites Summieren

Notation: Seien $U, U' \subseteq V$

$$f(U, U') := \sum_{\substack{u \in U \\ u' \in U'}} f(u, u'), \quad f(u, U') := f(\{u\}, U')$$

Somit

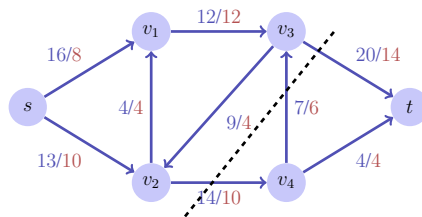
- $|f| = f(s, V)$
- $f(U, U) = 0$
- $f(U, U') = -f(U', U)$
- $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$, wenn $X \cap Y = \emptyset$.
- $f(R, V) = 0$ wenn $R \cap \{s, t\} = \emptyset$. [Flusserhaltung!]

414

Wie gross kann ein Fluss sein?

Es gilt für jeden Fluss und jeden Schnitt, dass $f(S, T) = |f|$:

$$\begin{aligned} f(S, T) &= f(S, V) - \underbrace{f(S, S)}_0 = f(S, V) \\ &= f(s, V) + \underbrace{f(S - \{s\}, V)}_{\neq t, \neq s} = |f|. \end{aligned}$$



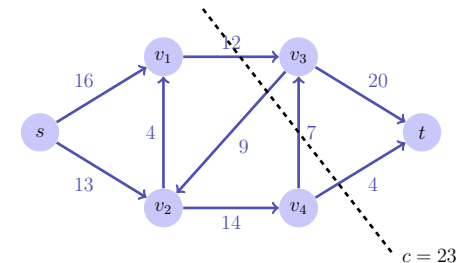
415

Maximaler Fluss ?

Es gilt insbesondere für alle Schnitte (S, T) von V .

$$|f| \leq \sum_{v \in S, v' \in T} c(v, v') = c(S, T)$$

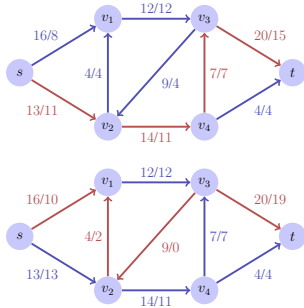
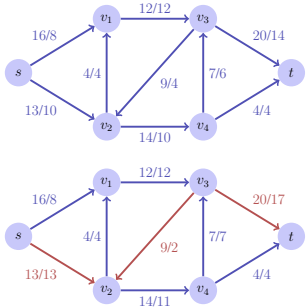
Werden sehen, dass Gleichheit gilt für $\min_{S, T} c(S, T)$.



416

Maximaler Fluss ?

Naives Vorgehen:



Folgerung: Greedy Flusserhöhung löst das Problem nicht.

Die Ford-Fulkerson Methode

- Starte mit $f(u, v) = 0$ für alle $u, v \in V$
- Bestimme Restnetzwerk* G_f und Erweiterungspfad in G_f
- Erhöhe Fluss über den Erweiterungspfad*
- Wiederholung bis kein Erweiterungspfad mehr vorhanden.

$$G_f := (V, E_f, c_f)$$

$$c_f(u, v) := c(u, v) - f(u, v) \quad \forall u, v \in V$$

$$E_f := \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

*Wird im Folgenden erklärt

Flusserhöhung, negativ

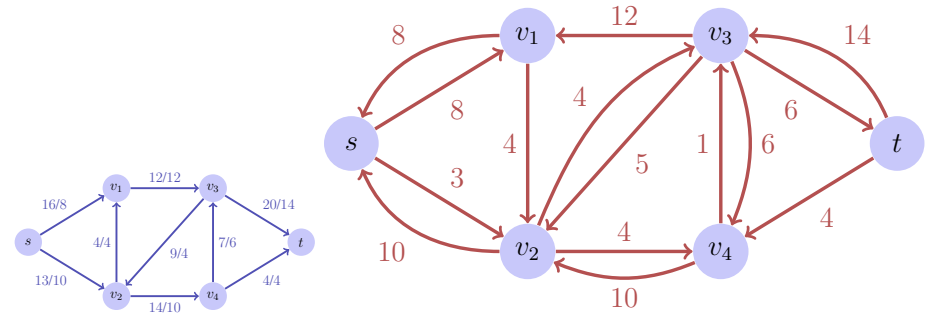
Sei ein Fluss f im Netzwerk gegeben.

Erkenntnis:

- Flusserhöhung in Richtung einer Kante möglich, wenn Fluss entlang der Kante erhöht werden kann, also wenn $f(u, v) < c(u, v)$.
Restkapazität $c_f(u, v) = c(u, v) - f(u, v) > 0$.
- Flusserhöhung *entgegen* der Kantenrichtung möglich, wenn Fluss entlang der Kante verringert werden kann, also wenn $f(u, v) > 0$.
Restkapazität $c_f(v, u) = f(u, v) > 0$.

Restnetzwerk

Restnetzwerk G_f gegeben durch alle Kanten mit Restkapazität:



Restnetzwerke haben dieselben Eigenschaften wie Flussnetzwerke, außer dass antiparallele Kapazitäten-Kanten zugelassen sind.

Beobachtung

Theorem

Sei $G = (V, E, c)$ ein Flussnetzwerk mit Quelle s und Senke t und f ein Fluss in G . Sei G_f das dazugehörige Restnetzwerk und sei f' ein Fluss in G_f . Dann definiert $f \oplus f'$ mit

$$(f \oplus f')(u, v) = f(u, v) + f'(u, v)$$

einen Fluss in G mit Wert $|f| + |f'|$.

Beweis

$f \oplus f'$ ist ein Fluss in G :

- Kapazitätsbeschränkung

$$(f \oplus f')(u, v) = f(u, v) + \underbrace{f'(u, v)}_{\leq c(u, v) - f(u, v)} \leq c(u, v)$$

- Schiefsymmetrie

$$(f \oplus f')(u, v) = -f(v, u) + -f'(v, u) = -(f \oplus f')(v, u)$$

- Flusserhaltung $u \in V - \{s, t\}$:

$$\sum_{v \in V} (f \oplus f')(u, v) = \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) = 0$$

421

422

Beweis

Wert von $f \oplus f'$

$$\begin{aligned} |f \oplus f'| &= (f \oplus f')(s, V) \\ &= \sum_{u \in V} f(s, u) + f'(s, u) \\ &= f(s, V) + f'(s, V) \\ &= |f| + |f'| \end{aligned}$$

■

423

Erweiterungspfade

Erweiterungspfad p : einfacher Pfad von s nach t im Restnetzwerk G_f .

Restkapazität $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ Kante in } p\}$

424

Fluss in G_f

Theorem

Die Funktion $f_p : V \times V \rightarrow \mathbb{R}$,

$$f_p(u, v) = \begin{cases} c_f(p) & \text{wenn } (u, v) \text{ Kante in } p \\ -c_f(p) & \text{wenn } (v, u) \text{ Kante in } p \\ 0 & \text{sonst} \end{cases}$$

ist ein Fluss in G_f mit dem Wert $|f_p| = c_f(p) > 0$.

f_p ist ein Fluss (leicht nachprüfbar). Es gibt genau einen Knoten $u \in V$ mit $(s, u) \in p$. Somit $|f_p| = \sum_{v \in V} f_p(s, v) = f_p(s, u) = c_f(p)$.

425

Folgerung

Strategie für den Algorithmus:

Mit einem Erweiterungspfad p in G_f definiert $f \oplus f_p$ einen neuen Fluss mit Wert $|f \oplus f_p| = |f| + |f_p| > |f|$.

426

Max-Flow Min-Cut Theorem

Theorem

Wenn f ein Fluss in einem Flussnetzwerk $G = (V, E, c)$ mit Quelle s und Senke t ist, dann sind folgende Aussagen äquivalent:

- 1 f ist ein maximaler Fluss in G
- 2 Das Restnetzwerk G_f enthält keine Erweiterungspfade
- 3 Es gilt $|f| = c(S, T)$ für einen Schnitt (S, T) von G .

427

[Beweis]

- (3) \Rightarrow (1):
Es gilt $|f| \leq c(S, T)$ für alle Schnitte S, T . Aus $|f| = c(S, T)$ folgt also $|f|$ maximal.
- (1) \Rightarrow (2):
 f maximaler Fluss in G . Annahme: G_f habe einen Erweiterungspfad. Dann gilt $|f \oplus f_p| = |f| + |f_p| > |f|$. Widerspruch.

(not shown in class) 428

[Beweis (2) \Rightarrow (3)]

Annahme: G_f habe keinen Erweiterungsfad

Definiere $S = \{v \in V : \text{es existiert Pfad } s \rightsquigarrow v \text{ in } G_f\}$.

$(S, T) := (S, V \setminus S)$ ist ein Schnitt: $s \in S, t \in T$.

Sei $u \in S$ und $v \in T$. Dann $c_f(u, v) = 0$, also

$c_f(u, v) = c(u, v) - f(u, v) = 0$. Somit $f(u, v) = c(u, v)$.

Somit

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) = \sum_{u \in S} \sum_{v \in T} c(u, v) = C(S, T).$$

(not shown in class) 429

Algorithmus Ford-Fulkerson(G, s, t)

Input: Flussnetzwerk $G = (V, E, c)$

Output: Maximaler Fluss f .

for $(u, v) \in E$ **do**

$f(u, v) \leftarrow 0$

while Existiert Pfad $p : s \rightsquigarrow t$ im Restnetzwerk G_f **do**

$c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$

foreach $(u, v) \in p$ **do**

$f(u, v) \leftarrow f(u, v) + c_f(p)$

$f(v, u) \leftarrow f(v, u) - c_f(p)$

Praktische Anmerkung

In einer Implementation des Ford-Fulkerson Algorithmus werden die negativen Flusskanten normalerweise nicht gespeichert, da ihr Wert sich stets als der negierter Wert der Gegenkante ergibt.

$f(u, v) \leftarrow f(u, v) + c_f(p)$

$f(v, u) \leftarrow f(v, u) - c_f(p)$

wird dann zu

if $(u, v) \in E$ **then**

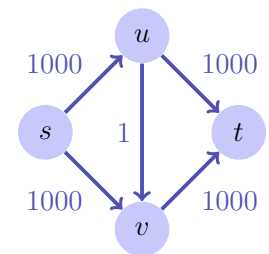
$f(u, v) \leftarrow f(u, v) + c_f(p)$

else

$f(v, u) \leftarrow f(v, u) - c_f(p)$

Analyse

- Der Ford-Fulkerson Algorithmus muss für irrationale Kapazitäten nicht einmal terminieren! Für ganze oder rationale Zahlen terminiert der Algorithmus.
- Für ganzzahligen Fluss benötigt der Algorithmus maximal $|f_{\max}|$ Durchläufe der While-Schleife (denn der Fluss erhöht sich mindestens um 1). Suche einzelner zunehmender Weg (z.B. Tiefensuche oder Breitensuche) $\mathcal{O}(|E|)$. Also $\mathcal{O}(f_{\max}|E|)$.



Bei schlecht gewählter Strategie benötigt der Algorithmus hier bis zu 2000 Iterationen.

Edmonds-Karp Algorithmus

Wähle in der Ford-Fulkerson-Methode zum Finden eines Pfades in G_f jeweils einen Erweiterungspfad kürzester Länge (z.B. durch Breitensuche).

Edmonds-Karp Algorithmus

Theorem

Wenn der Edmonds-Karp Algorithmus auf ein ganzzahliges Flussnetzwerk $G = (V, E)$ mit Quelle s und Senke t angewendet wird, dann ist die Gesamtanzahl der durch den Algorithmus angewendete Flusserhöhungen in $\mathcal{O}(|V| \cdot |E|)$.

\Rightarrow Gesamte asymptotische Laufzeit: $\mathcal{O}(|V| \cdot |E|^2)$

[Ohne Beweis]

433

434

Anwendung: Maximales bipartites Matching

Gegeben: bipartiter ungerichteter Graph $G = (V, E)$.

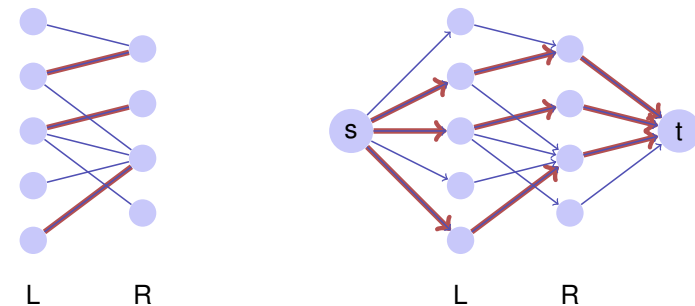
Matching M : $M \subseteq E$ so dass $|\{m \in M : v \in m\}| \leq 1$ für alle $v \in V$.

Maximales Matching M : Matching M , so dass $|M| \geq |M'|$ für jedes Matching M' .



Korrespondierendes Flussnetzwerk

Konstruiere zur einer Partition L, R eines bipartiten Graphen ein korrespondierendes Flussnetzwerk mit Quelle s und Senke t , mit gerichteten Kanten von s nach L , von L nach R und von R nach t . Jede Kante bekommt Kapazität 1.



435

436

[Ganzzahligkeitstheorem]

Theorem

Wenn die Kapazitäten eines Flussnetzwerks nur ganzzahlige Werte annehmen, dann hat der durch Ford-Fulkerson erzeugte maximale Fluss die Eigenschaft, dass der Wert von $f(u, v)$ für alle $u, v \in V$ eine ganze Zahl ist.

[ohne Beweis]

Folgerung: Ford Fulkerson erzeugt beim zum bipartiten Graph gehörenden Flussnetzwerk ein maximales Matching

$$M = \{(u, v) : f(u, v) = 1\}.$$