

Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 1a, Dienstag, 25. April 2017
(Gesamtüberblick, Sortieren, Kurssysteme)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

- Überblick über die gesamte Veranstaltung
 - Inhalt: kurze Übersicht
 - Ablauf: Vorlesungen, Übungen, Stil, Aufwand
 - Klausur: Zulassungskriterien, Termin, Note
- Sortieren, Coding Standards, Kurssysteme
 - Sortieren: ein erster Algorithmus + Programm dazu
 - Coding Standards: Unit Tests, Stil, Build Framework
 - Kurssysteme: Daphne, Forum, SVN, Jenkins
 - **ÜB 1:** Implementieren Sie MergeSort (kommt morgen dran) und vergleichen Sie die Laufzeit mit MinSort (wie heute)

■ Ausgangsbasis

- Sie können schon kleinere Programme schreiben

Zum Beispiel: berechnen, ob eine gegebene Zahl prim ist

- In Python oder Java oder C++ (Ihnen überlassen!)

Die meisten von Ihnen haben im 1. Semester **Python** gelernt

Parallel lernen die meisten jetzt im 2. Semester **Java**

- Verständnis einiger Grundkonzepte des Programmierens, z.B. Variablen, Funktionen, Schleifen, Ein- und Ausgabe, Rekursion

Wer da noch Lücken hat, kann auch mitmachen, aber es wird dann deutlich mehr Aufwand ... siehe Folie 11 zum Aufwand

Inhalt 2/4

■ Grundbausteine, die man immer wieder braucht

- Zum Beispiel:

Sortieren

Dynamische Felder / Assoziative Felder / Hash Maps

Prioritätswürgeschlangen

Suchbäume

Berechnen kürzester Wege in Netzwerken

Suchen von Mustern in Zeichenketten

- In Python, Java, C++ gibt es dafür Bibliotheken ... in dieser Vorlesung lernen Sie, was dahinter steckt

Für komplexere Projekte braucht man oft eine Variante von den Grundbausteinen ... oder neue Bausteine, die es noch gar nicht gibt

Inhalt 3/4

■ Effizienz

- In der "Informatik I" haben Fragen der Effizienz noch kaum eine Rolle gespielt ... in dieser Veranstaltung ist es ein ganz wesentlicher Aspekt:

Wie schnell läuft mein Programm / eine Funktion?

Wie kann ich es schneller machen?

Wie weiß ich, ob es noch schneller geht?

Manchmal geht es auch um andere Ressourcen, zum Beispiel Speicherverbrauch, aber meistens geht es um **Geschwindigkeit**

Inhalt 4/4

■ Methoden

- Laufzeitanalyse (O-Notation)

Zum Beispiel: mein Programm läuft in Linearzeit (= gut)

Oder: mein Programm hat quadratische Laufzeit (= böse)

- Den einen oder anderen Korrektheitsbeweis

Zum Beispiel: dynamische Felder mit konstanter Laufzeit pro Operation (im Durchschnitt)

- Die Mathematik, die wir in diesem Kurs verwenden, ist sehr "basic", aber es ist schon Mathematik, nicht nur "Rechnen"

Wir machen hier keine Theorie um der Theorie willen, sondern nur da wo hilfreich oder nötig, dann aber gerne !

Ablauf 1/5

■ Vorlesungen

- **Di 14:15 – 15:45 Uhr und Mi 16:15 – 17:45 Uhr, im HS 026**
- Insgesamt 25 Vorlesungstermine (der letzte am 26. Juli)

Keine Vorlesung am 6. + 7. Juni (Pfingstpause)
- Die Vorlesungen werden aufgezeichnet

Folien + Audio + Video ... Schnitt: Alexander Monneret
- Auf unserem Wiki finden Sie alle Kursmaterialien

Aufzeichnungen, Folien, Übungsblätter, Code aus der Vorlesung + ggf. zusätzliche Hinweise, Musterlösungen
- Auch im **SVN**, Unterordner **/public** (außer die Aufzeichnungen)

Ablauf 2/5

■ Übungen

- Die Übungen sind der wichtigste Teil der Veranstaltung
- Sie bekommen jede Woche ein Übungsblatt, insgesamt **12**
- Das können Sie machen wo und wann Sie wollen (im Rahmen der Abgabefrist)
- Aber Sie müssen es **selber** machen

Sie können gerne zusammen über die Übungsblätter nachdenken, diskutieren, etc. ... aber die Lösungen bzw. Programme müssen Sie dann **100% selber** schreiben

Auch das teilweise Übernehmen gilt als Täuschungsversuch, siehe auch die Erklärungen auf Seite 2 vom 1. Übungsblatt

Ablauf 3/5

■ Übungsgruppen

- Wie in allen unseren Lehrveranstaltungen läuft der Übungsbetrieb komplett **online**
- Sie bekommen jede Woche Feedback zu Ihren Aufgaben, von Ihrem Tutor bzw. Ihrer Tutorin:

Maya Schöchlin, Sebastian Holler, Daniel Tischner,
Daniel Bindemann, Danny Stoll, Simon Selg

- Assistent der Vorlesung ist: **Axel Lehmann**
- Für Fragen aller Art gibt es ein **Forum** (siehe Wiki)

Es wird in der zweiten oder dritten Woche auch eine Fragestunde geben ... erfahrungsgemäß wollen die alle haben und dann geht aber keiner hin

Ablauf 4/5

■ Stil der Veranstaltung

- Vorlesungen: viele Beispiele + viele Programme (live)

Die Details müssen Sie sich selber aneignen, sonst lernt man sie nicht, u.a. dafür sind die Übungsblätter da

Die Vorlesungen enthalten alles, was man dafür braucht

Am Ende jedes Foliensatzes: weiterführende Literatur

■ Praxisbezug

- Die Übungsblätter sind zu 1/3 Theorie, zu 2/3 Praxis

Möglichst praxisnahe Übungsaufgaben, damit Sie sehen, wozu man das alles braucht ... später mehr, weil komplexer

Theorie genau da wo nötig bzw. hilfreich zum Verständnis

Ablauf 5/5

■ Aufwand / ECTS Punkte

- Veranstaltung zählt 8 ECTS Punkte = 240 Arbeitsstunden
- Insgesamt 25 Vorlesungen = 50 (relaxte) Arbeitsstunden
- Außerdem 12 Übungsblätter

Sorgfältige Bearbeitung der Übungsblätter + gründliches Verständnis dahinter = beste Vorbereitung auf die Klausur

- Optionen für Ihr Zeitmanagement ÜB = Übungsblatt
 - A. 9-12 Std / ÜB, wenig lernen für Klausur **EMPFOHLEN**
 - B. 4-6 Std / ÜB, viel lernen für Klausur **MINIMUM**
 - C. 0 Std / ÜB, ??? lernen für Klausur **UNMÖGLICH**

■ Zulassung

- Für jedes Übungsblatt gibt es maximal **20 Punkte**
12 Übungsblätter → maximal 240 Punkte
- Davon müssen Sie insgesamt mindestens die Hälfte (120 Punkte) erreichen, um zur Klausur zugelassen zu werden
Das können alle schaffen, die kontinuierlich mitarbeiten
- Für das Ausfüllen des Evaluationsbogens am Ende gibt es noch mal 20 Punkte, mit denen Sie die Punktzahl Ihres schlechtesten Übungsblattes ersetzen können
U.a. als Joker für Kranksein oder sonstige Umstände

■ Termin

- Am [Wochentag], den [Tag]. [Monat] 2017 von 14 – 17 Uhr
- 6 Aufgaben à 20 Punkte, wir zählen die besten 5
- Also maximal 100 Punkte

■ Die Endnote

- ... ergibt sich linear aus der Punktzahl in der Klausur
- | | | | | | |
|-----------|------|----------|------|----------|-----|
| 50 – 54: | 4.0; | 55 – 59: | 3.7; | 60 – 64: | 3.3 |
| 65 – 69: | 3.0; | 70 – 74: | 2.7; | 75 – 79: | 2.3 |
| 80 – 84: | 2.0; | 85 – 89: | 1.7; | 90 – 94: | 1.3 |
| 95 – 100: | 1.0 | | | | |

Sortieren 1/4

■ Problemdefinition

- **Eingabe:** eine Folge von n Elementen x_1, \dots, x_n
Sowie ein transitiver Operator \leq auf diesen Elementen
Transitiv: $x \leq y$ und $y \leq z \Rightarrow x \leq z$

- **Ausgabe:** die n Elemente in gemäß diesem Operator sortierter Reihenfolge, zum Beispiel

Eingabe: $17, 4, 32, 19, 8, 44, 65, 19$

Ausgabe: $4, 8, 17, 19, 19, 32, 44, 65$

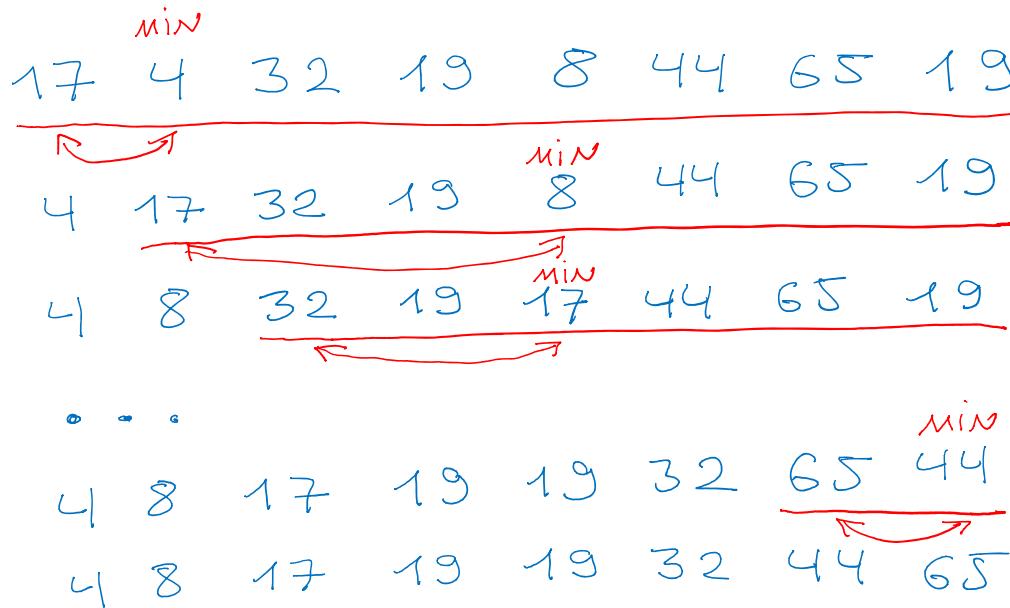
■ Wo braucht man Sortieren?

- In praktisch **jedem** größeren Programm
Beispiel: Bauen eines Indexes für eine Suchmaschine

Sortieren 2/4

■ MinSort: Informale Beschreibung

- Finde das Minimum und tausche es an die erste Stelle
- Finde das Minimum im Rest und tausche es an die zweite Stelle
- Finde das Minimum im Rest und tausche es an die dritte Stelle
- usw.



■ MinSort: Programm

- Ich mache es heute in Python vor
- Morgen zeige ich auch Code in Java und C++

Im Laufe der Vorlesung plane ich öfter mal hin- und her zu wechseln, je nach Problemstellung

Sie können sich auch für jedes ÜB neu entscheiden
(es gibt keine Abhängigkeiten der ÜB untereinander)

Es gibt aber evtl. einige wenige ÜB, wo Python nicht erlaubt oder möglich sein wird (weil manche Effizienz-Betrachtungen in Python einfach keinen Sinn machen)

■ MinSort: Laufzeit

- Wir testen das mal für verschiedene Eingabegrößen

Beobachtung: Es wird "unverhältnismäßig" langsamer,
je mehr Zahlen sortiert werden

- Um das genauer zu machen, malen wir ein Schaubild:

x-Achse: Eingabegröße, y-Achse: Laufzeit

Beobachtung: Die Laufzeit "wächst schneller als linear"

Das heißt, für doppelt so viele Zahlen braucht es (viel)
mehr als doppelt so viel Zeit

- Nächste Woche machen wir das präziser, diese Woche
bleiben wir erst mal noch bei Schaubildern

Für das 1. Übungsblatt sollen Sie auch eins malen

Coding Standards 1/3

■ Was und warum

- Sie sollen auch (weiter) **gutes Programmieren** lernen
- Dazu gehören ... für: Python / Java / C++

Unit Tests für jede nicht-triviale Funktion

Für korrekten Code ... mit: doctest / junit / gtest

Einheitlicher Stil Ihrer Codes

Für verständlichen Code ... mit: flake8 / checkstyle / cpplint

Build-Framework zum automatischen Testen

Damit wir nicht bei jeder Abgabe getrennt schauen müssen,
wie man den Code testet etc ... mit: make / ant / make

- **Ich werde das jetzt mal für Python vormachen**

Java und C++ Code im SVN unter public/code/vorlesung-01

Coding Standards 2/3

■ Warum Unit Tests

- **Grund 1:** Eine nicht-triviale Methode ohne Unit Test ist mit hoher Wahrscheinlichkeit nicht korrekt
- **Grund 2:** Macht das Debuggen von größeren Programmen viel leichter und angenehmer
- **Grund 3:** Wir und Sie selber können automatisch testen ob Ihr Code das tut was er soll

Am Anfang nervt es vielleicht etwas, aber mit der Zeit kapiert man, wofür es gut ist und irgendwann macht es dann sogar Spaß!

Coding Standards 3/3

■ Mindestanforderungen für diese Veranstaltung

- Ein Unit Test für **jede nicht-triviale Funktion**
- Für mindestens **eine typische** Eingabe
- Für mindestens **einen kritischen** "Grenzfall", wenn es einen solchen gibt ... z.B. leeres Feld beim Sortieren

Das ist in der Regel sehr wenig Arbeit ... mit sehr großem Nutzen (für Sie und für uns)

Siehe auch noch mal Rückseite vom 1. Übungsblatt

■ Daphne, unser Kursverwaltungssystem

- Link auf der Wiki-Seite zum Kurs, **bitte anmelden!**
- In Daphne haben Sie eine Übersicht über folgende Infos
 - Wer Ihr Tutor ist
 - Ihre Punkte in den Übungsblättern
 - Link zu den **Coding Standards** ... siehe letzte Folien
 - Link zum **Forum** ... siehe nächste Folie
 - Link zum **SVN** ... siehe übernächste Folie
 - Link zu **Jenkins** ... siehe überübernächste Folie

Kurssysteme 2/4

■ Forum zur Veranstaltung

- Link dazu auf dem Wiki und auf Ihrer Daphne Seite
- Bitte fragen Sie, wann immer etwas nicht klar ist !

Aber bitte möglichst **konkret** fragen: siehe Anleitung
dazu auf dem Wiki

Und fragen Sie uns bitte nicht einzeln, sondern auf dem
Forum ... praktisch immer interessiert das auch andere

- Entweder **Ich** oder **Axel Lehmann** oder eine*r der
Tutor*innen wird dann möglichst schnell antworten

- **SVN** = Subversion <http://subversion.apache.org/>
 - Dateien liegen bei uns auf einem zentralen Server
 - Typische Operationen: **svn add**, **svn commit**, **svn update**
 - Vollständige Historie von allen Änderungen an den Dateien
 - Kurze Anleitung auf dem Wiki
 - Wir benutzen das hier für fast alles:
 - Die Abgaben Ihrer Übungsblätter (Code + alles andere)
 - Das Feedback von Ihrem Tutor / Ihrer Tutorin
 - Folien / Vorlesungsdateien / Übungsblätter / Musterlösungen
 - **Ich werde das jetzt mal kurz vormachen**

Kurssysteme 4/4

■ Jenkins ist unser automatisches Build System

- Damit können Sie schauen, ob Ihr Code, so wie Sie ihn in unser SVN hochgeladen haben, kompiliert und läuft

Insbesondere ob die **Unit Tests** alle durchlaufen

Und ob der **Stil** in Ordnung ist

- **Zeige ich Ihnen auch gerade mal**
- Wichtig: Abgaben, für die "compile" fehlschlägt oder die keine Tests haben werden **nicht** korrigiert

Es wäre sonst unzumutbar viel Arbeit für die Tutoren

Literatur / Links

■ Weiterführende Literatur

- Mehlhorn / Sanders: Algorithms and Data Structures, The Basic Toolbox

Neueres Lehrbuch zu Algorithmen und Datenstrukturen, mit praktischerer Ausrichtung als ältere Lehrbücher. Online:

<http://www.mpi-inf.mpg.de/~mehlhorn/Toolbox.html>

- Für fast alle grundlegenden Algorithmen und Datenstrukturen gibt es inzwischen gute bis sehr gute Wikipedia Artikel

Und wenn man etwas rumgoogelt, findet man auch die eine oder andere hilfreiche Animation oder Live-Demo

Eigentlich sind die Vorlesungen aber self-contained, d.h. Sie kriegen alles erklärt, was Sie für die Übungsblätter brauchen

Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 1b, Mittwoch, 26. April 2017
(MergeSort, Divide and Conquer, Rekursion)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

■ Organisatorisches

- Code aus dem [/public](#) Verzeichnis im SVN

Diese Code dürfen Sie grundsätzlich verwenden wie Sie möchten, siehe ausführliche Post dazu auf dem **Forum**

■ Inhalt

- [MergeSort](#): ein besserer Sortieralgorithmus

Wir betrachten eine iterative und eine rekursive Variante

Für das ÜB1 sollen Sie die iterative Variante nehmen

- [Divide and Conquer](#): das allgemeine Prinzip dahinter
- [Rekursion](#): kurze Wiederholung für die, die es wieder vergessen oder noch nie richtig verstanden haben

MergeSort 1/6

■ Mischen (Merge), Definition

- Wir betrachten erstmal folgendes Teilproblem:
- Gegeben zwei sortierte Folgen **A** und **B**
- Berechne eine sortierte Folge **C** mit den Elementen aus **A** und **B**
- Das ist einfacher als das Ganze neu zu sortieren, weil **A** und **B** ja schon sortiert sind

A: 17 19 44 65

B: 4 7 18 31

C: 4 7 17 18 19 31 44 65



MergeSort 2/6

■ Mischen, Algorithmus

- Wir merken uns in jedem Feld eine Position (i und j)
- Zu Beginn $i = j = 0$
- Jetzt gehen wir immer in dem Feld weiter nach rechts wo das kleinere Element von $A[i]$ und $B[j]$ steht
- **Das schauen wir uns erst an einem Beispiel an und implementieren es dann zusammen**

A: $\begin{matrix} 17 \\ \downarrow \\ 17 \end{matrix}$ $\begin{matrix} 19 \\ \downarrow \\ 19 \end{matrix}$ $\begin{matrix} 44 \\ \downarrow \\ 44 \end{matrix}$ 65

B: $\begin{matrix} 4 \\ \uparrow \\ 4 \end{matrix}$ $\begin{matrix} 7 \\ \uparrow \\ 7 \end{matrix}$ $\begin{matrix} 18 \\ \uparrow \\ 18 \end{matrix}$ $\begin{matrix} 31 \\ \uparrow \\ 31 \end{matrix}$

C: $4 \ 7 \ 17 \ 18 \ 19 \ \dots$

■ MergeSort, iterativer Algorithmus

- **Grundidee:** wir benutzen Mischen, um aus kleinen sortierten Teilstücken größere sortierte Teilstücke zu machen
- Wir beginnen mit Teilstücken der Größe 1, die wir paarweise zu sortierten Teilstücken der Größe 2 mischen
- Diese sortierten Teilstücke der Größe 2 mischen wir dann paarweise zu sortierten Teilstücken der Größe 4
- Und so weiter ... bis das ganze Feld sortiert ist
- Beispiel auf der nächsten Folie, mit **n = Zweierpotenz**

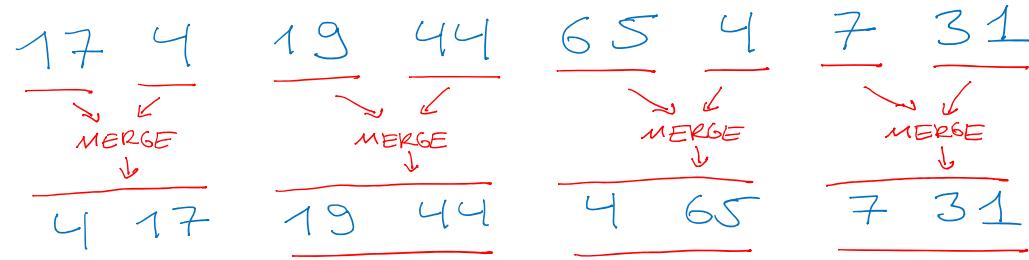
Für das ÜB1 müssen Sie sich überlegen, wie es auch für beliebige n (= nicht unbedingt eine Zweierpotenz) geht

MergeSort 4/6

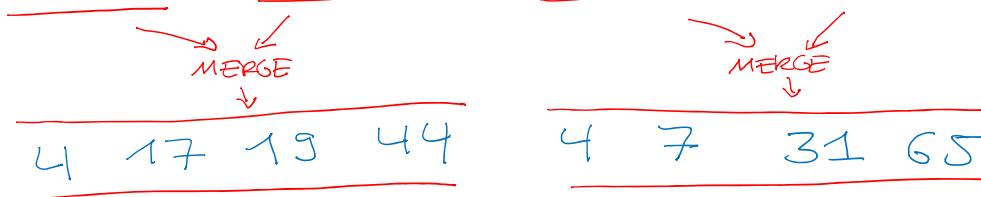
$m = 8$
3 Runden

MergeSort, iterativer Algorithmus, Beispiel

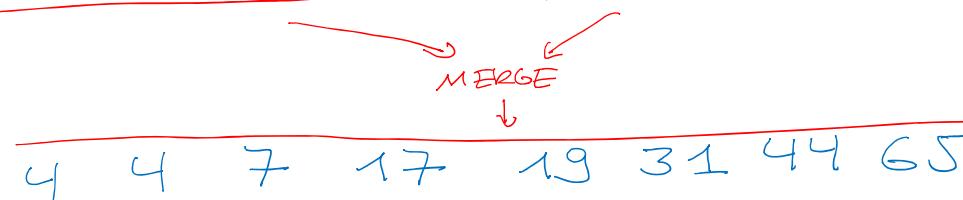
Runde 1



Runde 2



Runde 3



■ MergeSort, rekursiver Algorithmus

- Dasselbe Prinzip lässt sich auch rekursiv anwenden:

Teile das Eingabefeld in zwei (fast) gleichgroße Teile

Für jedes der beiden Teilstücke: falls größer als 1,
sortiere das Teilstück rekursiv (= mit derselben Funktion)

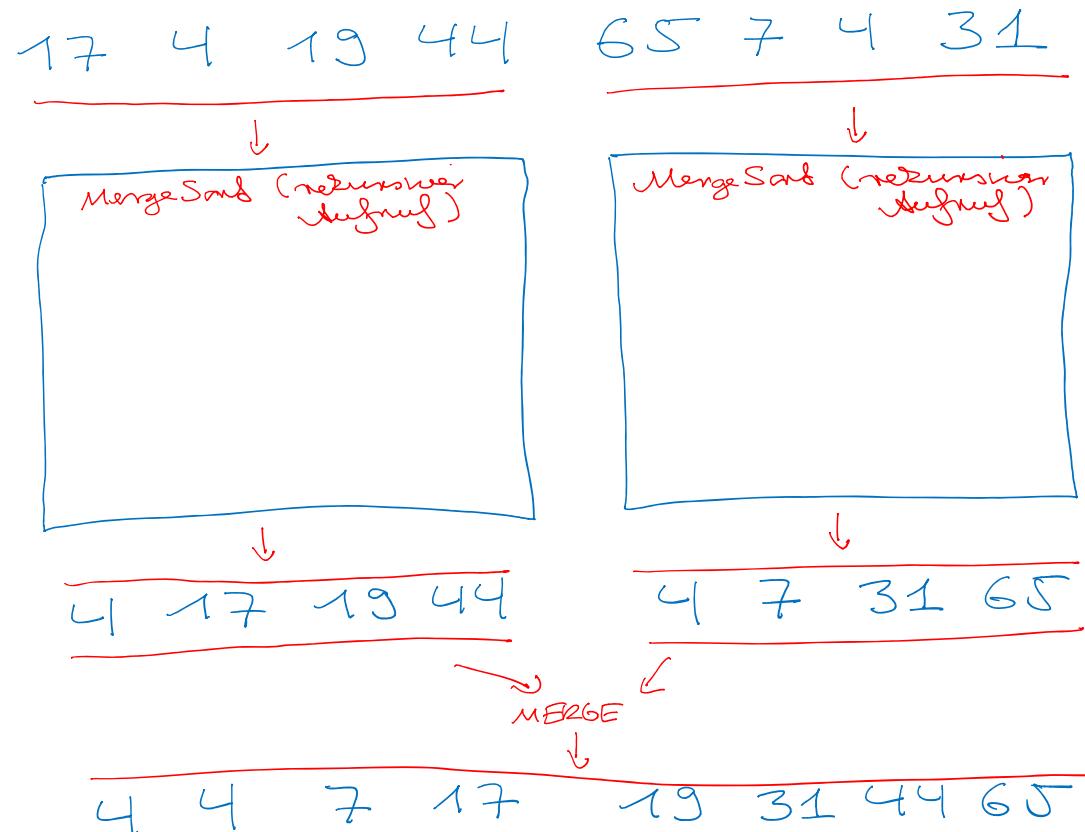
Mische die beiden sortierten Hälften

- Von der Implementierung her ist das **einfacher**
- Aber es ist schwerer zu verstehen, was konkret passiert
und es ist auch leichter (doofe) Fehler zu machen

Deswegen bleiben wir für das ÜB1 erstmal bei der
(aufwändigeren aber klareren) iterativen Variante

MergeSort 6/6

■ MergeSort, rekursiver Algorithmus, Beispiel



■ Allgemeines Prinzip

- Teile das gegebene Problem in kleinere Teile
- Setze Lösungen für größere Teile aus Lösungen für kleinere Teile zusammen
- Zwei Eigenschaften sind notwendig, damit das klappt:
 1. Das Zusammensetzen von Teillösungen ist einfacher, als das Problem von Grund auf zu lösen
 2. Das Problem ist für kleinste Teile einfach(er) zu lösen
- Das führt in der Regel zu einer **rekursiven** Funktion
- Man kann es aber auch (immer) **iterativ** implementieren
- Je nach Implementierung, kann die rekursive oder die iterative Variante schneller sein

■ Etymologie

- Latein: Divide et impera
- Deutsch: Teile und herrsche
- Französisch: Diviser pour régner
- Spanisch: Divide y vencerás
- Griechisch: Διαιρεῖ καὶ βασίλευε
- Usw.
- Das kommt eigentlich aus der Kriegsführung

So wie der Campus hier übrigens auch ... wir versuchen,
das Beste daraus zu machen

- Ist im Prinzip ganz einfach
 - Eine Funktion ruft sich als Teil Ihres Codes selber wieder auf
 - Man muss nur darauf achten, dass das Ganze irgendwann aufhört, so wie das hier z.B. **nicht**
- ```
def eternalDamnation():
 eternalDamnation()
```

$$1, 1, 2, 3, 5, 8, 13, \dots$$
$$F_1, F_2, F_3, F_4, F_5, F_6, F_7, \dots$$

## ■ Ein einfaches Beispiel

- **Fibonacci-Zahlen** ... die sind ja rekursiv definiert:

$$F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2} \text{ für alle } n \geq 3$$

- Die programmieren wir jetzt rekursiv ... und lassen uns dabei zum Verständnis die Rekursionsstufen ausgeben

- Die rekursive Implementierung ist rein zu Lehrzwecken und sehr ineffizient, aus **zwei** Gründen:

1. Durch die doppelte Rekursion wird derselbe Wert viele Male berechnet → iterativ wäre hier effizienter

2. Es gibt außerdem eine geschlossene Formel für  $F_n$  mit einer konstanten Anzahl von Operationen

$$F_n = \frac{1}{\sqrt{5}} (\varphi^n - (1-\varphi)^n) \dots \varphi = \frac{1+\sqrt{5}}{2} = 1.61803\dots$$

Solenne  
Santtt

# Literatur / Links

---

## ■ MergeSort

- Wikipedia: [Merge sort](#)
- Mehlhorn/Sanders: [5.2 Mergesort](#)

## ■ Divide and Conquer

- Wikipedia: [Divide and conquer algorithm](#)
- Mehlhorn/Sanders: [über das ganze Buch verteilt](#)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 2a, Dienstag, 2. Mai 2017  
(Laufzeitanalyse MinSort und MergeSort)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Organisatorisches

- Ihre Erfahrungen mit dem Ü1 (Drumherum + Sortieren)

## ■ Laufzeitanalyse

- Allgemein                wie fasst man Laufzeit mathematisch?
- MinSort                 "quadratische" Laufzeit
- MergeSort              besser, aber auch nicht ganz "linear"
- Auffrischung           vollständige Induktion, Logarithmus
- ÜB2: vier wunderschöne Theorieaufgaben zum Einüben dieser Grundtechniken und Konzepte

## ■ Zusammenfassung / Auszüge

- Wie üblich beim ÜB1 sehr große Unterschiede in den Bearbeitungszeiten: bei manchen war es schnell gemacht, andere haben ewig daran gesessen
- Einige fanden das iterative MergeSort kompliziert
- Einige kamen mit rekursiv vs. iterativ durcheinander

Das Bild von Vorlesung 1b, Folie 6 ist aber sehr klar

- Es konnte viel Code aus der VL übernommen werden
- Viele Fragen auf dem Forum zum "Drumherum"

Das ist auch normal für das erste Übungsblatt

# Erfahrungen mit dem ÜB1 2/2

---

## ■ Ergebnisse

- MergeSort ist **viel** schneller als MinSort
- Die Laufzeit auf dem Schaubild sieht "linear" aus

## ■ Wie lange laufen unsere bisherigen Programme?

- Für MinSort und MergeSort hatten wir dazu bisher zwei Schaubilder und Folgendes beobachtet

**MinSort:** Laufzeit wird "unproportional" langsamer, und damit schon bei mittleren Eingabegrößen sehr langsam

Doppelt so große Eingabe → mehr als doppelt so langsam

**MergeSort:** Laufzeit wird "proportional" langsamer und bei mittleren Eingabegrößen viel schneller als MinSort

Doppelt so große Eingabe → ca. doppelt so langsam

## ■ Wie können wir das präziser fassen

- **Idealerweise:** eine Formel, die uns für eine bestimmte Eingabe sagt, wie lange das Programm darauf läuft
- **Problem:** Laufzeit hängt auch noch von vielen anderen Umständen ab, insbesondere
  - auf was für einem Rechner wird den Code ausführen
  - was sonst gerade noch auf dem Rechner läuft
  - was für eine Programmiersprache benutzt wurde
  - welchen Compiler wir benutzt haben
  - Jahreszeit, Mondphase, Raum-Zeit-Verkrümmung, ...

# Laufzeitanalyse allgemein 3/5

## ■ Abstraktion 1: Anzahl Operationen statt Laufzeit

- Intuitiv: eine Operation = eine Zeile Code, zum Beispiel:

|                           |                 |
|---------------------------|-----------------|
| Arithmetische Operationen | $a + b$         |
| Variablenzuweisungen      | $x = y$         |
| Verzweigungen             | if ... else ... |
| Sprung zu einer Funktion  | min_sort(...)   |

- Genauer wäre: eine Zeile Maschinencode ... oder noch genauer: ein Prozessorzyklus

Wir sehen später noch, dass es nicht so wichtig ist, wie genau wir diese Operationen definieren

Wichtig ist, dass die Anzahl Operationen ungefähr **proportional** zur tatsächlichen Laufzeit ist

## ■ Abstraktion 2: Abschätzung statt genau zählen

- Meistens Abschätzung nach oben ("obere Schranke")

Dann wissen wir, wie lange ein Programm höchstens braucht

- Seltener Abschätzung nach unten ("untere Schranke")

Dann wissen wir, wie lange ein Programm mindestens braucht

Schätzen statt genau zählen erleichtert die Aufgabe

Und wir haben ja eh schon abstrahiert von der exakten Laufzeit zu der Anzahl Operationen

## ■ Abstraktion 3: Schranken pro Eingabegröße $n$

- Oft hängt die Laufzeit vor allem von der Größe der Eingabe ab, und nur wenig davon, wie die Eingabe genau aussieht

Zum Beispiel hängt die Laufzeit von MinSort für eine Eingabegröße  $n$  nur minimal von der genauen Eingabe ab

- Wir schreiben deswegen für die Laufzeit oft  $T(n)$

Wenn wir obere Schranken berechnen wollen, ist damit die **maximale** Laufzeit für eine Eingabe der Größe  $n$  gemeint

Wenn wir untere Schranken berechnen wollen, ist damit die **minimale** Laufzeit für eine Eingabe der Größe  $n$  gemeint

Diese Notation ist mathematisch etwas unpräzise, aber es ist in aller Regel klar, was gemeint ist

# Laufzeitanalyse MinSort 1/4

für irgendeine Konstante A und B

UNI  
FREIBURG

- Es gilt:  $T(n) \leq C_1 \cdot n^2$  ... für irgendeine Konstante  $C_1$

- MinSort hat eine äußere und eine innere Schleife
  - Für jede Iteration der äußeren Schleife, schätzen wir die Anzahl Operationen der inneren Schleife ab

$$\text{Iteration 1: } \leq A \cdot (n-1) + B$$

$$\text{Iteration 2: } \leq A \cdot (n-2) + B$$

...

$$\text{Iteration } n-1: \leq A + B + B \cdot (n-1)$$

$$\text{Insgesamt: } T(n) \leq A \cdot \underbrace{(n-1 + n-2 + \dots + 1)}_{\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)} + B \cdot (n-1)$$

$$\leq \frac{1}{2}A \cdot \underbrace{n(n-1)}_{\leq n^2} + B \cdot \underbrace{(n-1)}_{\leq n^2}$$

$$\leq \frac{1}{2}A \cdot n^2 + B \cdot n^2$$

$$\leq (\underbrace{A/2 + B}_{=: C_1}) \cdot n^2$$

# Laufzeitanalyse MinSort 2/4

für irgendeine  
Konstanten  $A'$  und  $B'$

UNI  
FREIBURG

- Es gilt auch:  $T(n) \geq \underline{C_2} \cdot n^2 \dots$  für eine Konst.  $C_2 < C_1$

$$\text{Iteration 1 : } \geq A' \cdot (n-1) + B' \quad \text{für } n \geq 2$$

$$\text{Iteration 2 : } \geq A' \cdot (n-2) + B'$$

$$\dots$$
  
$$\text{Iteration } n-1 : \geq A' + B'$$

$$\text{Insgesamt : } T(n) \geq A' \cdot (\underbrace{n-1 + n-2 + \dots + 1}_{= \frac{1}{2}n(n-1)}) + B' \cdot (n-1)$$

$$\geq A'/2 \cdot n \cdot \underbrace{(n-1)}_{\geq \frac{m}{2}} + B' \cdot (n-1) \geq 0$$

$$\geq \underbrace{A'/4}_{=: C_2} \cdot n^2$$



## ■ Quadratische Laufzeit

- Es gibt Konstanten  $C_1$  und  $C_2$ , so dass gilt

$$C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$$

- Dann gilt insbesondere

Doppelt so große Eingabe  $\rightarrow$  viermal so große Laufzeit

$$T(2m) = C \cdot (2m)^2 = 4 \cdot C \cdot m^2 = 4 \cdot T(m)$$

Laufzeitanalyse MinSort 4/4

## ■ Quadratische Laufzeit, Rechenbeispiel

- Unabhängig von den Konstanten wird das schnell sehr teuer  
 $10^{-9}$
  - Annahme:  $C = 1 \text{ ns}$  (1 einfache Anweisung  $\approx 1$  Nanosekunde)
  - Beispiel 1:  $n = 10^6$  (1 Millionen Zahlen = 4 MB, 4 Bytes/Zahl)  
... dann  $C \cdot n^2 = \underline{10^{-9}} \cdot 10^{12} \text{ s} = 10^3 \text{ s} = 16.7 \text{ Minuten}$
  - Beispiel 2:  $n = 10^9$  (1 Milliarde Zahlen = 4 GB)  
... dann  $C \cdot n^2 = 10^{-9} \cdot 10^{18} \text{ s} = 10^9 \text{ s} = 31.7 \text{ Jahre}$   
 $\downarrow \times 1 \text{ Millionen}$

**Quadratische Laufzeit = "große" Probleme unlösbar**

## ■ Analyse iterativer MergeSort

- Annahme:  $n$  ist eine Zweierpotenz und Mischen von zwei Feldern der Größe  $m$  geht in höchstens  $A \cdot m$  Zeit

**Iteration 1:**  $n$  Teilfelder der Größe jeweils 1

$$n / 2 \text{ mal Mischen} \rightarrow A \cdot \frac{n}{2} = A/2 \cdot n$$

**Iteration 2:**  $n/2$  Teilfelder der Größe jeweils 2

$$n / 4 \text{ mal Mischen} \rightarrow A \cdot \frac{n}{4} = A/2 \cdot n$$

...

**Iteration k:** 2 Teilfelder der Größe jeweils  $n / 2^k$

$$1 \text{ mal Mischen} \rightarrow A \cdot \frac{n}{2^k} = A/2 \cdot n$$

## ■ Analyse iterativer MergeSort ... Fortsetzung

- Sei  $k$  die Anzahl der Iterationen
- Dann ist die Laufzeit insgesamt  $T(n) \leq A/2 \cdot n \cdot k$
- Wie groß ist  $k$  ?
- Die Teilstücke in Iteration  $k$  sind  $2^{k-1} = n / 2$  groß
- Also  $k - 1 = \log_2 (n/2) = (\log_2 n) - 1 \leq \log_2 n \Rightarrow \underline{\log_2 n + 1}$
- Es ist also  $T(n) \leq A/2 \cdot n \cdot (1 + \log_2 n)$       ÜBRIGENS :  $\log_2 1 = 0$
- Frage: kommt bei der rekursiven Implementierung auch etwas mit  $n \cdot \log_2 n$  heraus?

## ■ Analyse rekursiver MergeSort

- Annahme wieder:  $n$  ist eine Zweierpotenz und Mischen von zwei Feldern der Größe  $m$  geht in  $\leq A \cdot m$  Zeit
- Nach dem Bild von Vorlesung 1b, Folie 8 :

$$T(n) \leq T(n/2) + T(n/2) + A \cdot n / 2$$

rec. Aufruf  
links Hälfte      rec. Aufruf  
rechte Hälfte      Mischen

Das gilt aber nur, wenn wir die Rekursion tatsächlich ausführen, also für  $n \geq 2$ ; für  $n = 1$  haben wir einfach:

$$T(1) \leq A$$

- Solch eine rekursive Gleichung ist typisch bei der Analyse der Laufzeit von einem rekursiven Algorithmus

ohne  $T$  auf der nächsten Seite

Wie kommen wir damit auf eine obere Schranke für  $T(n)$  ?

# Laufzeitanalyse MergeSort 4/6

## ■ Auflösung der rekursiven Gleichung

$$\begin{aligned} \text{Annahme: } m &= \text{Zweipotenz} \\ &= 2^k \end{aligned}$$

$$\begin{aligned} T(m) &\leq 2 \cdot T(m/2) + A \cdot m/2 & (*) \quad \forall m \\ &\stackrel{(*)}{\leq} 2 \cdot [2 \cdot T(m/4) + A \cdot m/4] + A \cdot m/2 \\ &\stackrel{\text{gib } m/2}{=} 4 \cdot T(m/4) + 2 \cdot A \cdot m/2 \\ &\stackrel{(*)}{\leq} 4 \cdot [2 \cdot T(m/8) + A \cdot m/8] + 2 \cdot A \cdot m/2 \\ &\stackrel{\text{gib } m/4}{=} 8 \cdot T(m/8) + 3 \cdot A \cdot m/2 \\ &\vdots \\ &\leq 2^k \cdot T(m/2^k) + k \cdot A \cdot m/2 \end{aligned}$$

$$\begin{aligned} 2^k = m &\Leftrightarrow k = \log_2 m \\ &\leq m \cdot \underbrace{T(1)}_{\leq A} + \log_2 m \cdot A \cdot \underbrace{m/2}_{\leq m} \\ &\leq A \cdot m + A \cdot m \cdot \log_2 m \\ &\leq A \cdot m \cdot (1 + \log_2 m) \end{aligned}$$

■

## ■ Laufzeit $n \cdot \log n$

- Es gibt Konstanten  $C_1$  und  $C_2$ , so dass gilt

$$C_1 \cdot n \cdot \log_2 n \leq T(n) \leq C_2 \cdot n \cdot \log_2 n \text{ für } n \geq 2$$

- Dann gilt insbesondere

Doppelt so große Eingabe  $\rightarrow$  etwas mehr als doppelt so lange

$$T(2m) = C \cdot 2 \cdot m \cdot \underbrace{\log_2(2m)}_{\log_2 2 + \log_2 m} = 2 \cdot C \cdot m \cdot \underbrace{(1 + \log_2 m)}_{\approx \log_2 m} \approx 2 \cdot T(m)$$

# Laufzeitanalyse MergeSort 6/6

## ■ Laufzeit $n \cdot \log n$ , Rechenbeispiel

- Annahme:  $C = 1 \text{ ns}$  ( $1 \text{ einfache Anweisung} \approx 1 \text{ Nanosekunde}$ )
- Beispiel 1:  $n = 2^{20}$  ( $\approx 1 \text{ Millionen Zahlen} = 4 \text{ MB}$ )  
... dann  $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{20} \cdot 20 \text{ s} = 21 \text{ Millisekunden}$
- Beispiel 2:  $n = 2^{30}$  ( $\approx 1 \text{ Milliarde Zahlen} = 4 \text{ GB}$ )  
... dann  $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{30} \cdot 30 \text{ s} = 32 \text{ Sekunden}$

Laufzeit  $n \cdot \log n$  ist also fast so gut wie linear!

## ■ Vollständige Induktion, Prinzip

- Man möchte beweisen, dass eine Aussage für alle natürlichen Zahlen gilt, also:  $A(n)$  gilt für alle  $n \in \mathbb{N}$
- **Induktionsanfang:** Wir zeigen, dass  $A(1), \dots, A(k)$  gelten  
*Meistens reicht  $k = 1$ , aber manchmal braucht man mehr*
- **Induktionsschritt:** Wir nehmen für ein beliebiges  $n > k$  an, dass  $A(1), \dots, A(n-1)$  gelten, und zeigen: dann gilt auch  $A(n)$   
*Meistens reicht  $A(n-1)$ , aber manchmal auch  $A(n-2), \dots$*
- Wenn wir die beiden Sachen gezeigt haben, haben wir nach dem Prinzip der **vollständigen Induktion** gezeigt, dass  $A(n)$  für alle natürlichen Zahlen  $n$  gilt

# Auffrischung 2/4

## ■ Vollständige Induktion, Beispiel

- Wir haben vorhin benutzt:  $\sum_{i=1..n} i = \frac{1}{2} \cdot n \cdot (n + 1)$

Induktionsanfang :  $\sum_{i=1}^1 i = 1 = 1 = \frac{1}{2} \cdot 1 \cdot \underbrace{(1+1)}_{=2}$  OK

Induktionsdritte  
 $1, \dots, n \rightarrow n+1$ :  
für bel.  $n \geq 1$

$$\begin{aligned}\sum_{i=1}^{n+1} i &= \underbrace{\sum_{i=1}^n i}_{\text{from Ind. Hyp.}} + n+1 \\ &= \frac{1}{2} n(n+1) \text{ nach Induktions-} \\ &\quad \text{voraussetzung} \\ &= \frac{1}{2} n(n+1) + n+1 \\ &= (\frac{1}{2} n + 1)(n+1) \\ &= \frac{1}{2} (n+2)(n+1)\end{aligned}$$

OK

# Auffrischung 3/4

## ■ Der Logarithmus ( $\neq$ Algorithmus)

- Der "Logarithmus zur Basis b" ist gerade die inverse Funktion zu "b hoch"

Formal:  $\log_b n = x \Leftrightarrow b^x = n$

Beispiel:  $\log_2 1024 = 10 \Leftrightarrow 2^{10} = 1024$

- Die Rechenregeln ergeben sich dann einfach aus den Rechenregeln für das Potenzieren

- Zum Beispiel:  $\log_b(x \cdot y) = (\underbrace{\log_b x}_{=: z_1}) + (\underbrace{\log_b y}_{=: z_2})$

$$z = \log_b(x \cdot y) \Rightarrow b^z = x \cdot y$$

$$z_1 = \log_b x \Rightarrow b^{z_1} = x$$

$$z_2 = \log_b y \Rightarrow b^{z_2} = y$$

$$b^z = x \cdot y = b^{z_1} \cdot b^{z_2} = b^{z_1 + z_2}$$

$$\Rightarrow z = z_1 + z_2$$

$$\Rightarrow \log_b(x \cdot y) = \log_b x + \log_b y$$



# Auffrischung 4/4

## ■ Der Logarithmus, Fortsetzung

- Der Logarithmus kommt in der Informatik **sehr häufig** vor, insbesondere bei der Analyse von Laufzeiten

$\log_2 n$  ist gerade: wie oft man  $n$  halbieren muss, bis man bei 1 ankommt ... oder umgekehrt: wie oft man 1 verdoppeln muss, bis man bei  $n$  ankommt

- Es kommt auch öfter mal vor, dass der Logarithmus in einer Potenz auftaucht, aber mit einer anderen Basis

Zum Beispiel:  $3^{\log_2 n}$

$$(b^x)^y = b^{x \cdot y} = (b^y)^x$$

In welcher Größenordnung liegt das?

$$3 = 2^{\log_2 3} \Rightarrow 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n}$$

$$= 2^{\log_2 3 \cdot \log_2 n}$$

$$\Rightarrow = 2^{\log_2 n \cdot \log_2 3} = (\underbrace{2^{\log_2 n}}_{\approx n})^{\log_2 3}$$

# Literatur / Links

---

## ■ Laufzeitanalyse

- Mehlhorn/Sanders: [2.6 Basic Program Analysis](#)
- Wikipedia: [Vollständige Induktion](#)
- Stupidea: [Unvollständige Induktion](#)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 2b, Mittwoch, 3. Mai 2017  
(Andere Sortierverfahren, Sortieren von Objekten,  
Sortieren in Linearzeit, Untere Schranke  $n \cdot \log n$ )

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Inhalt

- Andere Sortierverfahren QuickSort, HeapSort, ...
- Sortieren von Objekten Und nicht nur Zahlen
- Sortieren in Linearzeit 0-1-Sort und CountingSort
- Untere Schranke vergleichsbasiert geht es nicht besser als  $n \cdot \log n$

## ■ QuickSort, Grundprinzip

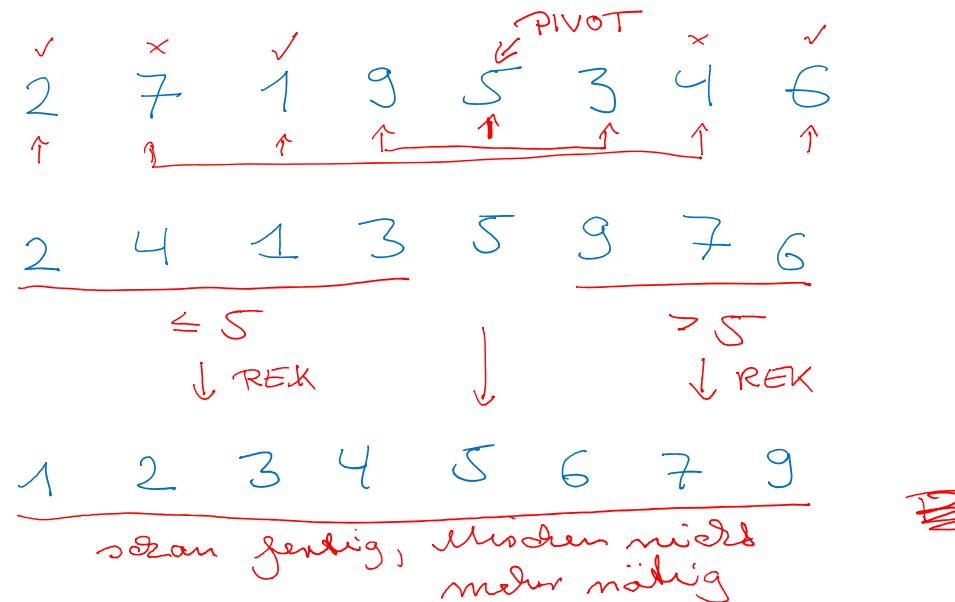
- Ähnlich wie dem rekursiven MergeSort wird das Feld in zwei Teile aufgeteilt, die dann rekursiv sortiert werden
- **Unterschied 1:** die Aufteilung ist so, dass alle Elemente im linken Teil  $\leq$  alle Elemente im rechten Teil sind  
Teilergebnisse müssen dann nicht mehr gemischt werden
- **Unterschied 2:** die Teile können sehr unterschiedlich groß sein, im schlechtesten Fall hat ein Teil nur Größe 1  
Die Rekursionstiefe kann so viel größer als  $\log_2 n$  werden

# Andere Sortierverfahren 2/8

## ■ QuickSort, Aufteilen + Beispiel

- Zum Aufteilen wird ein Element P des Feldes gewählt (z.B. das erste Element oder ein zufälliges Element)
- Das Feld wird dann so aufgeteilt, dass links alle Elemente  $\leq P$  stehen und rechts alle Elemente  $\geq P$

Das geht für ein Feld der Größe n in Zeit  $\leq A \cdot n$



## ■ QuickSort, Laufzeit

- Im besten Fall werden die Felder immer in zwei (fast) gleich große Hälften aufgeteilt, wie bei MergeSort

Die Laufzeit ist dann  $\leq C_1 \cdot n \cdot \log_2 n$  wie bei MergeSort

Aber in der Praxis schneller als MergeSort, weil das Mischen wegfällt und keine Felder kopiert werden müssen

- Im schlechtesten Fall wird das Feld pro Rekursionsstufe immer nur eins kleiner und die Laufzeit ist  $\geq C_2 \cdot n^2$
- Wenn das Pivot-Element immer zufällig gewählt wird, ist der Erwartungswert der Laufzeit auch  $\leq C_3 \cdot n \cdot \log_2 n$

## ■ HeapSort

- HeapSort benutzt einen **binären Heap** zum Sortieren

Das ist eine Datenstruktur, die aus einer Menge von  $n$  Elementen mit  $\leq C \cdot \log n$  Operationen das kleinste Element extrahieren und entfernen kann

Dazu in einer späteren Vorlesung mehr !

- HeapSort schafft damit auch **in jedem Fall**

$T(n) \leq C \cdot n \cdot (1 + \log_2 n)$  für eine Konstante  $C > 0$

- In der Praxis:

HeapSort etwas besser als MergeSort ... kleineres  $C$

HeapSort etwas schlechter als QuickSort ... größeres  $C$

## ■ Intelligent Design Sort

- Annahme: die Eingabezahlen sind alle verschieden
- Dann gibt es  $n!$  mögliche Permutationen dieser Zahlen
- Mit Wahrscheinlichkeit  $1/n!$  ist die Eingabe also sortiert
- Weil diese Wahrscheinlichkeit so klein ist, ist es absurd zu denken, dass dies zufällig passiert ist
- Es muss durch einen Intelligenten Sortierer erfolgt sein
- Man kann deswegen annehmen, dass die Eingabe schon optimal sortiert war ... in einer Reihenfolge, die unser weltliches Verständnis von "sortiert" transzendiert

## ■ BogoSort

- **Schritt 1:** Prüfe, ob die Eingabe bereits sortiert ist

Das geht in Zeit  $\leq C_1 \cdot n$ , für eine Konstante  $C_1$
- **Schritt 2:** Falls nicht, permutiere die Zahlen zufällig und gehe zu Schritt 1

Das geht ebenfalls in Zeit  $\leq C_2 \cdot n$ , für eine Konstante  $C_2$
- Die erwartete Laufzeit ist  $\geq C_3 \cdot n \cdot n!$ 

Die Frage ist: geht es noch langsamer?

## ■ BogoBogoSort

- Ersetze Schritt 1 (Prüfung, ob Eingabe sortiert) durch einen rekursiven Algorithmus wie folgt:
  1. Mache eine Kopie des Feldes
  2. Sortiere die ersten  $n - 1$  Elemente rekursiv
  3. Prüfe, ob das  $n$ -te Element größer ist, als das das letzte Element der rekursiv sortierten  $n - 1$  Elemente
  4. Falls nicht, permutiere die Element zufällig und gehe zu Schritt 2
- ÜB2 Zusatzaufgabe: schätzen Sie die Laufzeit ab  
**Schon auf Eingaben der Größe 7 wird es nicht fertig**

## ■ DropSort

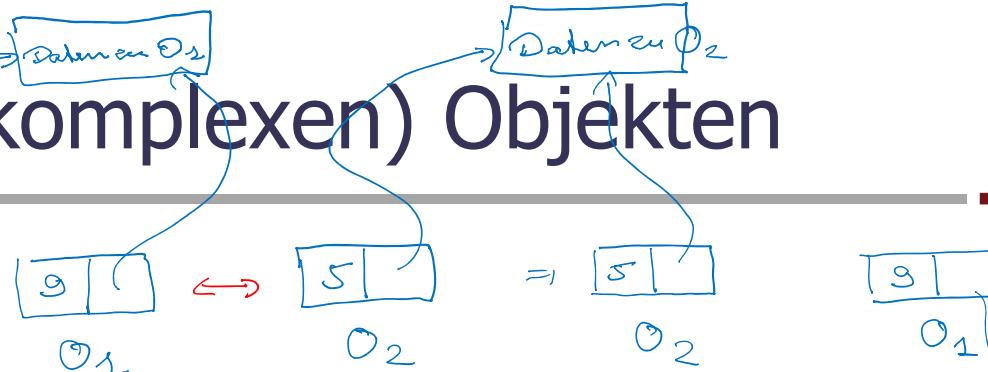
- Gehe von links nach rechts durch das Feld
- Wenn eine Element kleiner ist als das vorhergehende, wird es einfach nicht mit ausgegeben
- Die Eingabe ist dann garantiert immer sortiert, es fehlen nur vielleicht ein paar Elemente
- Einen solchen Algorithmus nennt man "**lossy**"

Das Prinzip wird zum Beispiel auch bei der Kompression von Bildern verwendet (JPEG Format)

Warum also nicht auch beim Sortieren?

# Sortieren von (komplexen) Objekten

## Motivation



- Meistens will man nicht einfach nur Zahlen sortieren, sondern komplexere Objekte nach bestimmten Werten

Zum Beispiel: Studierende nach Punktzahl

- Dann muss man aufpassen, dass man nicht bei jedem Vergleich zwei (evtl. große) Objekte hin- und her kopiert
- Lösung: in dem Objekt steht außer dem Wert, nach dem sortiert wird, nur **ein Zeiger** auf die anderen Daten

Dann müssen bei jedem Vertauschen nur die beiden Werte und die beiden Zeiger kopiert werden

# Sortieren in Linearzeit 1/3

## ■ ZeroOneSort

- Geht Sortieren auch schneller als  $n \cdot \log n$  ?
- Ja, zum Beispiel wenn alle Elemente nur 0 oder 1 sind
- Dann kann man einfach die 0en und 1en zählen

Dazu schreiben wir gerade ein Programm ZeroOneSort

0, 1, 1, 0, 0, 1, 0

#0 = 4, #1 = 3

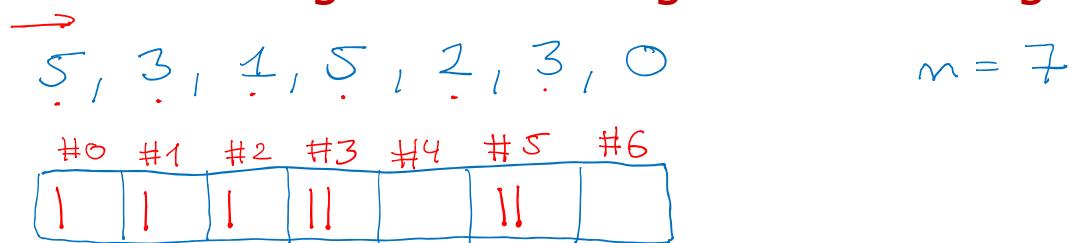
0, 0, 0, 0, 1, 1, 1  
4 mal                    3 mal

# Sortieren in Linearzeit 2/3

## ■ CountingSort

- Die Idee klappt auch noch, wenn die Elemente aus dem Bereich  $0 \dots n - 1$  sind

Dazu schreiben wir gerade ein Programm CountingSort



↓ Ausgabe

$\underbrace{0}_{1} \underbrace{1}_{1} \underbrace{2}_{1} \underbrace{33}_{11} \underbrace{55}_{11}$



# Sortieren in Linearzeit 3/3

## ■ Laufzeit

- Für beide Verfahren gilt  $T(n) \leq C \cdot n \dots$  für ein  $C > 0$

Man geht einmal über das Eingabefeld zum "Zählen",  
und dann gibt man die (gleich große) Ausgabe aus

- Ist das vielleicht sogar für beliebige Eingaben möglich?

CountingSort braucht ein Feld der Größe m für Zahlen  
aus dem Bereich 0 .. m-1

Für  $m \gg n$  ist die Laufzeit (und der Platzverbrauch)  
dann proportional zu m, und nicht zu n

## ■ Vergleichsbasiertes Sortieren

- ZeroOneSort und CountingSort sortieren die Elemente nicht durch "Umsortieren", sondern durch "Zählen"
- Wir wollen jetzt zeigen: wenn man nur "Umsortieren" zulässt, geht es tatsächlich nicht schneller als  $n \cdot \log n$
- Dazu müssen wir erst mal genauer fassen, was es heißt, dass ein Algorithmus "nur umsortiert"

## ■ Vorberachtung 1

- Wir werden uns bei unserem Beweis auf Algorithmen **von einer bestimmten Art** beschränken

Wir werden sehen: weil das die Argumentation erleichtert

- Nehmen wir an, ein Algorithmus **A** ist nicht von dieser Art, aber es gibt einen Algorithmus **A'** von der Art für den gilt:

**A** ist braucht höchstens  $\leq C_1 \cdot n$  Operationen mehr als **A'**

Die Ausgabe von **A** kann mit  $\leq C_2 \cdot n$  Operationen in die Ausgabe von **A'** überführt werden

- Dann gilt  $T_{A'}(n) \geq n \cdot \log n \Rightarrow T_A(n) \geq n \cdot \log n$

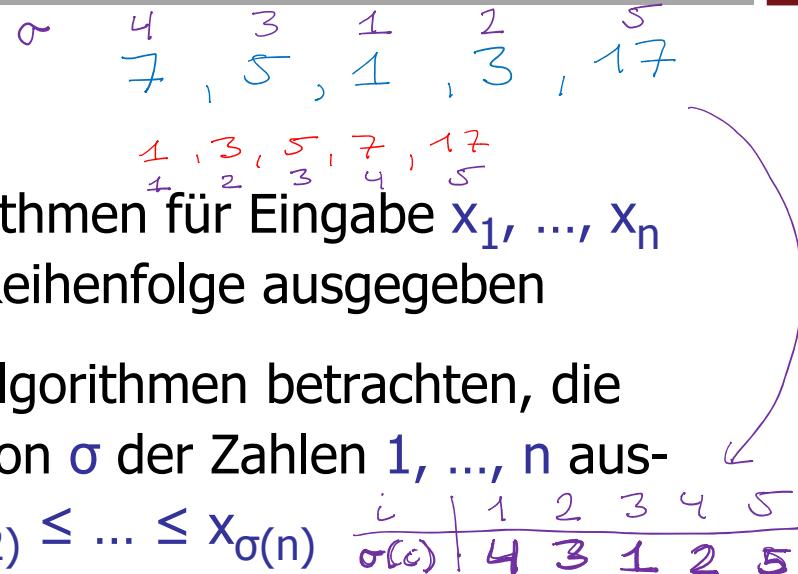
Wäre **A** schneller, könnten wir auch **A'** schneller machen

# Untere Schranke Sortieren 3/12

## ■ Vorberichtigung 2

- Bisher haben unsere Algorithmen für Eingabe  $x_1, \dots, x_n$  diese Zahlen in sortierter Reihenfolge ausgegeben
- Wir wollen im Folgenden Algorithmen betrachten, die stattdessen eine Permutation  $\sigma$  der Zahlen  $1, \dots, n$  ausgeben, so dass  $x_{\sigma(1)} \leq x_{\sigma(2)} \leq \dots \leq x_{\sigma(n)}$
- **Beobachtung:** alle unseren bisherigen Algorithmen können dahingehend abgewandelt werden, ohne dass sich die Anzahl Operationen um mehr als  $C \cdot n$  ändert

Das gilt insbesondere für ZeroOneSort und CountingSort



## ■ Vor betrachtung 3

- In einem Programm (Python, Java, C++) können an diversen Stellen Verzweigungen auftreten

`while ( ... ) { ... }`

`for ( ... ) { ... }`

`if (...) { ... } else { ... }`

- Ohne Beschränken der Allgemeinheit seien all diese Verzweigungen von der Form `if (...) { ... } else { ... }`

- Zum Beispiel ist `while (EXPR) { ... }` äquivalent zu:

`while (true) { if (EXPR) { ... } else { break; } }`

## ■ Vor betrachtung 4

- Betrachten wir die Folge von Entscheidungen in den `if (...) { ... } else { ... }` Teilen im Ablauf eines Programms
- Dann entspricht jeder Ablauf einer Folge **IEEIIIEIIIIE...**  
**I** = `if`-Teil wird ausgeführt, **E** = `else`-Teil wird ausgeführt
- Ein Algorithmus heißt **vergleichsbasiert**, wenn die I/E Folge die Ergebnispermutation **eindeutig** bestimmt
- `MinSort`, `MergeSort`, `QuickSort`, `HeapSort` sind allesamt vergleichsbasiert bzw. können dazu gemacht werden
- `ZeroOneSort` und `CountingSort` sind es nicht, und können auch nicht dahingehend abgeändert werden

## ■ ZeroOneSort ist nicht "vergleichsbasiert"

- Hier sind zwei Eingaben mit gleicher I/E Folge aber verschiedenen Ergebnispermutationen

Wir lassen dabei die I und E von den "for" Schleifen weg, die hängen ja sowieso nicht von der Eingabe ab

$$\sigma_1 \quad \begin{matrix} 1 & 4 & 5 & 2 & 3 \\ \textcolor{blue}{0}, 1, 1, \textcolor{blue}{0}, \textcolor{blue}{0} \end{matrix} \quad \textcolor{red}{I I I E E}$$

$$\sigma_2 \quad \begin{matrix} 1 & 4 & 2 & 3 & 5 \\ \textcolor{blue}{0}, 1, \textcolor{blue}{0}, \textcolor{blue}{0}, 1 \end{matrix} \quad \textcolor{red}{I I I E E}$$

- MinSort ist "vergleichsbasiert"
    - Beispiel: zwei Eingaben mit gleicher I/E Folge und gleicher Permutation, und dritte Eingabe mit anderer I/E Folge und anderer Permutation
- Wieder ohne die Is und Es von den "for" Schleifen

## ■ Beweis untere Schranke, Teil 1

- Wir betrachten jetzt einen beliebigen vergleichsbasierten Algorithmus **A**, der für eine Eingabe der Größe  $n$  eine sortierende Permutation  $\sigma$  der Zahlen  $1, \dots, n$  ausgibt

- Sei  $T(n)$  eine obere Schranke für die Anzahl der von **A** benötigten Operationen auf einer Eingabe der Größe  $n$

Dann gibt es höchstens  $T(n)$  Verzweigungs-Anweisungen

- Der Algorithmus gibt also für Eingabegröße  $n$  höchstens  $2^{T(n)}$  verschiedene Permutationen aus

*I E I I E E ... I*  
 $\underbrace{\quad\quad\quad}_{x \leq T(n) \text{ m鰂le}}$

$2^x \leq 2^{T(n)}$  M鰎glid -  
zeilen

## ■ Beweis untere Schranke, Teil 2

- Ein korrekter Algorithmus muss für Eingabegröße  $n$  alle möglichen Permutationen erzeugen können, das sind  $n!$   
*Wenn er eine Permutation nicht erzeugen könnte, würde er für die Eingabe, die genau diese Permutation zum Sortieren benötigt, nicht das richtige Ergebnis liefern*
- Auf der vorherigen Folie hatten wir gesehen, dass bei Laufzeit  $\leq T(n)$  höchstens  $2^{T(n)}$  Permutationen erzeugt werden können
- Wäre  $2^{T(n)} < n!$ , würden nicht alle Permutationen erzeugt werden können und der Algorithmus wäre nicht korrekt
- Es muss also  $2^{T(n)} \geq n!$  sein, oder äquivalent  $T(n) \geq \log_2 (n!)$

## ■ Abschätzung von $\log_2(n!)$

- Dafür wird oft die Stirling-Formel benutzt:

$$n! \geq \sqrt{2\pi n} \cdot (n/e)^n$$

- Wir können das aber auch (weniger genau, aber ausreichend) elementar-mathematisch abschätzen:

$$n! \geq (n/2)^{n/2}$$

- Daraus folgt:

$$\log_2(n!) \geq \log_2((n/2)^{n/2}) \geq n/2 \cdot \log_2(n/2)$$

$$6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \geq 3^3 \geq 3 \cdot 3 \cdot 3 \geq 3^3 \geq \frac{1}{2} \cdot \log_2 n + \frac{n}{4} \quad \forall n \geq 4$$

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \geq 2.5^3 \geq 2.5^{2.5} \geq 2.5 \cdot 2.5 \geq 2.5$$

## ■ Beweis untere Schranke, Zusammenfassung

- Wir haben gezeigt:

Sei  $T(n)$  eine obere Schranke für einen Algorithmus, der  $n$  Elemente vergleichsbasiert sortiert

Dann ist  $T(n) \geq \log_2 (n!) \geq \frac{1}{4} \cdot n \cdot \log_2 n$  für  $n \geq 4$

## ■ Fazit

- Unter den vergleichsbasierten Algorithmen sind also QuickSort, MergeSort, HeapSort alle optimal !
- Aber in der Praxis zählen auch (unter anderem):

**Konstante Faktoren:** zum Beispiel ist  $10 \cdot n \cdot \log n$  offensichtlich 10 mal langsamer als  $n \cdot \log n$

Komplexe oder komplizierte Implementierungen haben typischerweise viel höhere Konstanten

**Cache-Effizienz:** der Zugriff auf aufeinanderfolgende Elemente im Speicher ist billiger als wenn "verstreut"

Spielt vorherrschende Rolle bei großen Datenmengen

- Zu diesen Aspekten in einer späteren Vorlesung mehr !

# Literatur / Links

---

## ■ Untere Schranke

- Mehlhorn/Sanders: [5.3 A Lower Bound \[for Sorting\]](#)

## ■ Laufzeitanalysen von QuickSort

- Wikipedia: [QuickSort#Formal\\_analysis](#)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 3a, Dienstag, 9. Mai 2017  
(O-Notation, Teil 1)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- Korrektur vom ÜB1 Wie, wann, was
  - Fragestunde Teil 2 der VL morgen
  - Was zählt als Plagiat Erinnerung + Klarstellung
  - Erfahrungen mit dem ÜB2 Laufzeitanalyse

## ■ O-Notation

- Motivation, Definition, Beispiele  $O, \Omega, \Theta, o, \omega$
  - ÜB3: ein paar Rechenaufgaben zu  $O$  und  $\Theta$  ... und die Laufzeit von drei Programmen als  $\Theta(\dots)$  bestimmen

# Korrektur vom ÜB1 1/2

---

## ■ Das funktioniert so

- Im SVN in einer Datei im zum ÜB gehörigen Ordner, z.B.  
`xy123/blatt-01/feedback-tutor.txt`
- Machen Sie einfach **svn update** in Ihrer Arbeitskopie
- Sie erhalten Ihre Korrektur in der Regel spätestens am Freitag nach Abgabe, allerspätestens am Wochenende  
Manchmal aber auch schon Mittwoch oder Donnerstag
- Falls Sie Wünsche oder Abneigungen in Bezug auf die Korrektur haben, sprechen Sie einfach mit Ihrem Tutor  
Das hat in der Vergangenheit immer sehr gut funktioniert

## ■ Rückmeldung vom Tutorentreffen

- Es haben trotz mehrfacher Warnung einige ein MergeSort mit quadratischer Laufzeit produziert
- Variablennamen sollten selbst-dokumentierend sein

Einbuchstabige Laufvariablen bei Schleifen OK, aber nur da

- Einige Leute sagen, sie hätten es nicht verstanden, aber haben sich keine Hilfe auf dem Forum geholt

Man kann seinen Tutor auch um ein Treffen bitten !

- Dokumentation sollte nicht sagen, was der Code macht (sieht man ja am Code), sondern welches Problem er löst

Aber nicht sowas schreiben wie: "Das steht hier, weil es sonst einen Index-Out-Of-Range Fehler gibt"

# Plagiat

---

## ■ Erinnerung und Klarstellung

- Es wurde in der Vorlesung 1a besprochen, stand auf den Folien und in rot und fett auf dem 1. Übungsblatt ... und trotzdem gab es schon wieder einige Plagiatsfälle
- Deswegen hier nochmal zur Klarstellung:

Auch das Übernehmen von **Lösungen oder Code aus dem Internet**, und sei es nur teilweise, gilt als **Plagiat**

Sie können miteinander diskutieren und recherchieren und googeln so viel Sie möchten

Aber den Code bzw. Ihre Lösungen müssen Sie dann zu **100% selber schreiben**

Ausnahme: alles aus dem SVN /public dürfen Sie benutzen

# Fragestunde morgen

---

## ■ Nach der Vorlesung morgen

- Die Vorlesung morgen wird nur ca. 1 Stunde dauern
- Danach machen wir eine Frage(halbe)stunde
- Sie können dort Fragen aller Art rund um den Vorlesungsstoff (und die Übungen dazu) stellen

**Überlegen Sie sich was !**

## ■ Zusammenfassung / Auszüge

- Manche haben generelle Probleme mit dem Beweisen
- Fehler bei der Aufgabenstellung von Aufgabe 1 ( $\varphi = y/x$ )  
*Wurde im Forum sehr schnell (Mittwoch 12:22 Uhr) geklärt!*
- Probleme mit dem Verständnis von Aufgabe 4
- Quartische Gleichungen haben noch eine Lösungsformel  
*Keine allgemeine Lösungsformel erst **ab Grad 5***
- Einige haben bei Aufgabe 4 Beweis aus der VL wiederholt  
*Bitte die Aufgaben sorgfältig lesen und verstehen und nicht einfach nur halbautomatisch Output produzieren*
- Hurra, die Fragen sind zurück + die ÜB haben wieder Sinn!

# Erfahrungen mit dem ÜB2

2/3

$$\sqrt{\frac{1}{4} + 1} = \sqrt{\frac{1}{4} + \frac{4}{4}} = \sqrt{\frac{5}{4}} = \frac{\sqrt{5}}{2}$$

UNI  
FREIBURG

## Lösungsskizze Aufgabe 1+2

AUFGABE 1 :  $\frac{x}{y} = \frac{y}{x+y}$

$$\Rightarrow \frac{y}{x} = \frac{x+y}{y} = \frac{x}{y} + 1 \quad \stackrel{y}{\Rightarrow} \quad = y \quad = 1/y$$

$$z^2 - z - 1 = 0 \quad \Rightarrow \quad z_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1 \pm \sqrt{5}}{2}$$



$$g := \frac{y}{x} \quad \text{z.z. } g^2 = g + 1$$

$$g^2 = 1 + g \quad \blacksquare$$

$$g > 1 \Rightarrow g = \frac{1 + \sqrt{5}}{2}$$

andere Lösung  
 $\psi = \frac{1 - \sqrt{5}}{2}$   
 $= 1 - g$

AUFGABE 2 : z.z.  $F_m = \frac{1}{\sqrt{5}} (g^m - \psi^m) \quad \forall m \in \mathbb{N}$

Induktionsanfang :  $m=1 \Rightarrow \frac{1}{\sqrt{5}} (g - \psi) = \frac{1}{\sqrt{5}} \cdot \sqrt{5} = 1 \quad \blacksquare$

$$m=2 \Rightarrow \frac{1}{\sqrt{5}} (g^2 - \psi^2) = \dots = 1 \quad \blacksquare$$

Induktionsgeschritt :  $m, m+1 \rightarrow m+2$

$$\begin{aligned} F_{m+2} &= \frac{1}{\sqrt{5}} (g^{m+1} + g^m - (\psi^{m+1} + \psi^m)) \\ &\stackrel{\text{maar Definition der } F_m}{=} F_{m+1} + F_m \\ &\stackrel{\text{maar I.V.}}{=} \frac{1}{\sqrt{5}} (g^{m+1} - \psi^{m+1}) + \frac{1}{\sqrt{5}} (g^m - \psi^m) \\ &= \frac{1}{\sqrt{5}} (g^m(g+1) - \psi^m(\psi+1)) \\ &= \frac{1}{\sqrt{5}} (g^m(g+1) - \psi^m(g+1)) \\ &= \frac{1}{\sqrt{5}} g^{m+2} - \psi^{m+2} \quad \blacksquare \end{aligned}$$

## ■ Lösungsskizze Aufgabe 4 (untere Schranke)

- In Vorlesung 2b wurde gezeigt: sei  $T(n)$  eine obere Schranke für die Laufzeit eines vergleichsbasierten Algorithmus für Eingabegröße  $n$ , dann  $T(n) \geq \log_2(n!)$
- Das heißt: für **mindestens eine Eingabe** der Größe  $n$  muss die Laufzeit  $\geq \log_2(n!)$  sein
- Das heißt **nicht**, dass **für alle Eingaben** der Größe  $n$  die Laufzeit  $\geq \log_2(n!)$  sein muss
- Es gilt auch nicht, weil man jedem Sortieralgorithmus, einen einfachen Test voranstellen kann, der in  $\leq A \cdot n$  Zeit prüft, ob die Eingabe schon sortiert ist  
Für sortierte Eingaben ist die Laufzeit dann  $\leq A \cdot n$

## ■ Erinnerung

- Wir haben jetzt mehrfach die Laufzeit  $T(n)$  in Abhängigkeit von der Eingabegröße abgeschätzt
- Die Werte der Konstanten waren dabei sekundär ... und auch, wenn die Schranken erst ab  $n \geq$  irgendeinem  $n_0$  galten

Für sehr kleine Eingaben sind Programme ja sowieso schnell

- Zum Beispiel hatte wir, für  $n \geq$  irgendeinem  $n_0$  :
  - Die Laufzeit von MinSort ist "irgendwas mal"  $n^2$
  - Die Laufzeit von MergeSort ist "irgendwas mal"  $n \cdot \log n$
  - Die Laufzeit von CountingSort ist "irgendwas mal"  $n$
  - Vergleichsbasiertes Sortieren dauert "irgendwas mal"  $n \cdot \log n$

## ■ Motivation

- Genau das wollen wir jetzt formaler machen, damit wir in Zukunft präzise sagen bzw. schreiben können

Die Laufzeit von MinSort ist  $\Theta(n^2)$

Die Laufzeit von MergeSort ist  $\Theta(n \cdot \log n)$

Die Laufzeit von CountingSort ist  $\Theta(n)$

Vergleichsbasiertes Sortieren hat Laufzeit  $\Omega(n \cdot \log n)$

## ■ Vorberichtigung

- Wir betrachten Funktionen  $f : \mathbb{N} \rightarrow \mathbb{R}$

$\mathbb{N}$  = die natürlichen Zahlen ... typisch: Eingabegröße

$\mathbb{R}$  = die reellen Zahlen ... typisch: Laufzeit

Uns reicht, wenn  $f(n) > 0$  für  $n \geq n_0$  ... darunter darf  $f$  negativ sein, und das kommt bei Abschätzungen auch manchmal raus

- Beispiele

$$f(n) = 3 \cdot n + 3$$

$$f(n) = 2 \cdot n \cdot (\log_2 n - 5)$$

für  $m \leq 32$  :  $g(m) \leq 0$  ; für  $m > 32$  :  
 $= 2^5$   $g(m) > 0$

$$f(n) = n^2 / 10$$

$$f(n) = n^2 + 3 \cdot n \cdot \log_2 n - 4 \cdot n$$

## ■ Groß-O, Definition

- Seien  $g$  und  $f$  zwei Funktionen  $\mathbf{N} \rightarrow \mathbf{R}$
- **Intuitiv:** Man sagt  $g$  ist Groß-O von  $f$  ...  
wenn  $g$  "höchstens so stark wächst wie"  $f$
- **Informal:** Man schreibt  $g = O(f)$  ...  
wenn ab irgendeinem Wert  $n_0$  für all  $n \geq n_0$   
 $g(n) \leq C \cdot f(n)$  für irgendeine Konstante  $C$
- **Formal:** für eine Funktion  $f : \mathbf{N} \rightarrow \mathbf{R}$  ist ...  
*INTUITIV: alle Funktionen, die nicht stärker wachsen als  $f$*   
 $O(f) = \{ g : \mathbf{N} \rightarrow \mathbf{R} \mid \exists n_0 \in \mathbf{N} \ \exists C > 0 \ \forall n \geq n_0 \ g(n) \leq C \cdot f(n) \}$   
dabei heißt  $\exists$  = "es existiert ..." und  $\forall$  = "für alle ..."

eigentlich wäre ziemlich:  
 $g \in O(g)$

## ■ Groß-O, Beispiel

- Sei  $g(n) = 5 \cdot n + 7$  und  $f(n) = n$
- Dann ist  $g = O(f)$  bzw. man schreibt  $5 \cdot n + 7 = O(n)$
- **Intuitiv:**  $5 \cdot n + 7$  wächst höchstens "linear"
- Beweis unter Verwendung der Definition von O :

zu zeigen:  $5 \cdot n + 7 \leq C \cdot n$  für "irgendem"  $C$ ,  
 für  $n \geq$  "irgendem"  $n_0$

Beweis 1:  $5 \cdot n + 7 \leq 5 \cdot n + 7 \cdot n = \underbrace{12 \cdot n}_{=:C}$

$\leq 7 \cdot n$   
 für  $n \geq 1$   
 $=:n_0$

□

Beweis 2:  $5 \cdot n + 7 \leq 5 \cdot n + n = \underbrace{6 \cdot n}_{=:C}$

$\leq n$   
 für  $n \geq 7$   
 $=:n_0$

□

- Es zählt "Wachstumsrate", nicht absolute Werte

- Für zwei Funktionen kann ohne Probleme gelten

$$g = O(f)$$

g wächst **nicht stärker** als f

$$g > f$$

g ist überall **echt größer** als f

- Zum Beispiel **g** und **f** von der Folie vorher

$$g(m) = 5 \cdot m + 7 ; \quad g(m) = m$$

$$\Rightarrow g(m) > g(m) \quad \forall m \geq 1 \quad \text{also} \quad g > g$$

TROTZDEM:  $g = O(g)$

## ■ Groß-Omega, Definition + Beispiel

- **Intuitiv:** Man sagt  $g$  ist Groß-Omega von  $f$  ...  
... wenn  $g$  "mindestens so stark wächst wie"  $f$   
Also wie Groß-O, nur mit "mindestens" statt "höchstens"
- **Formal:** Für eine Funktion  $f : \mathbf{N} \rightarrow \mathbf{R}$  ist

$$\Omega(f) = \{ g : \mathbf{N} \rightarrow \mathbf{R} \mid \exists n_0 \in \mathbf{N} \ \exists C > 0 \ \forall n \geq n_0 \ g(n) \geq \underline{C \cdot f(n)} \}$$

- Zum Beispiel  $5 \cdot n + 7 = \Omega(n)$
- Beweis unter Verwendung der Definition von  $\Omega$  :

zu zeigen:  $5 \cdot n + 7 \geq C \cdot n$  für "irgendem"  $C$ ,  
für  $n \geq$  "irgendem"  $n_0$

Beweis:  $5 \cdot n + 7 \geq \underbrace{5 \cdot n}_{\geq 0} \geq \underbrace{C \cdot n}_{= C}$

sogar für alle  $n$

## ■ Groß-Theta, Definition + Beispiel

- **Intuitiv:** Man sagt  $g$  ist Theta von  $f$  ...  
... wenn  $g$  "genauso so stark wächst wie"  $f$
- **Formal:** Für eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}$  ist  
 $\Theta(f) = O(f) \cap \Omega(f) =$  die Schnittmenge von  $O(f)$  und  $\Omega(f)$   
Wächst "höchstens so stark" **und** "mindestens so stark"
- Zum Beispiel  $5 \cdot n + 7 = \Theta(n)$
- Beweis unter Verwendung der Definition von  $\Theta$  :

$$\text{Folie 14} \Rightarrow 5 \cdot n + 7 = O(n)$$

$$\text{Folie 16} \Rightarrow 5 \cdot n + 7 = \Omega(n)$$

$$\Rightarrow 5 \cdot n + 7 = \Theta(n)$$

qed

- Es gibt auch noch  $o$  (Klein-O) und  $\omega$  (Klein-Omega)

- Die braucht man in der Informatik viel seltener
  - Hier kurz die Definitionen für  $f : \mathbf{N} \rightarrow \mathbf{R}$

$$o(f) = \{ g : \underline{\forall C > 0} \exists n_0 \in \mathbf{N} \ \underline{\forall n \geq n_0} \ g(n) \leq C \cdot f(n) \}$$

einiger Unterschied zu  $O(g)$

$$\omega(f) = \{ g : \underline{\forall C > 0} \exists n_0 \in \mathbf{N} \ \underline{\forall n \geq n_0} \ g(n) \geq C \cdot f(n) \}$$

einiger Unterschied zu  $\Omega(g)$

- Intuitiv:

$g = o(f)$  :  $g$  wächst (strikt) langsamer als  $f$

$g = \omega(f)$  :  $g$  wächst (strikt) schneller als  $f$

Insbesondere ist die Schnittmenge leer:  $o(f) \cap \omega(f) = \emptyset$

## ■ Intuitive Zusammenfassung

- Die Operatoren  $\mathcal{O}$ ,  $\Omega$ ,  $\Theta$ ,  $\circ$ ,  $\omega$  sind auf Funktionen, was die Operatoren  $\leq$ ,  $\geq$ ,  $=$ ,  $<$ ,  $>$  auf Zahlen sind:

|               |            |        |                                                                                                          |
|---------------|------------|--------|----------------------------------------------------------------------------------------------------------|
| $\mathcal{O}$ | entspricht | $\leq$ | WICHTIG: es macht keinen Sinn zu sagen, dass ein Algorithmus Laufzeit mindestens $\mathcal{O}(n^2)$ hat. |
| $\Omega$      | entspricht | $\geq$ |                                                                                                          |
| $\Theta$      | entspricht | $=$    |                                                                                                          |
| $\circ$       | entspricht | $<$    | wenn schon, dann mindestens $\Theta(n^2)$                                                                |
| $\omega$      | entspricht | $>$    |                                                                                                          |

## ■ Weitere Eigenschaften

- Viele Eigenschaften von  $\leq$ ,  $\geq$ ,  $=$ ,  $<$ ,  $>$  gelten auch sinngemäß genauso für  $O$ ,  $\Omega$ ,  $\Theta$ ,  $o$ ,  $\omega$

- Zum Beispiel: Transitivität

$$\cancel{x < y} \wedge \cancel{y \leq z} \Rightarrow \cancel{x < z}$$

$$f = o(g) \wedge g = O(h) \Rightarrow f = o(h)$$

- Zum Beispiel: Additivität

$$\cancel{x_1 \leq y_1} \wedge \cancel{x_2 \leq y_2} \Rightarrow \cancel{x_1 + x_2 = y_1 + y_2}$$

$$f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(g_1 + g_2)$$

Gute Zusatzaufgabe für die, die vom ÜB unterfordert sind

# Literatur / Links

---

## ■ O-Notation / $\Omega$ -Notation / $\Theta$ -Notation

- In Mehlhorn/Sanders:

2.1 Asymptotic Notation

- In Wikipedia

[http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation)

<http://de.wikipedia.org/wiki/Landau-Symbole>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 3b, Dienstag, 10. Mai 2017  
(O-Notation, Teil 2)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Drumherum

- Klausurtermin
- Astrologie
- Fragestunde

**29. August ... 14 – 17 Uhr**

Was ist dran?

oder auch nur halbe Stunde

## ■ Inhalt

- O-Notation und Grenzwerte
- Diskussion

Bestimmung via  $\lim_{n \rightarrow \infty}$

Sinn und Grenzen,  
mehrere Variablen

# Klausurtermin

Dienstag

## ■ Der Termin steht jetzt fest

- Es ist der **29. August 2017**, von 14 Uhr bis max. 17 Uhr
- Es wird geschrieben im **HS 026 + 036** und bei Bedarf auch im SR 00 10/14 und SR 01 09/13 (alles hier in 101)
- Zur Aufteilung sagen wir dann noch rechtzeitig was, siehe die Seite zur Klausur auf dem Wiki der Veranstaltung
- Das Horoskop für den Termin: Sonne in Jungfrau im 9. Haus, Aszendent in Skorpion, Mond in Schütze im 1. Haus  
**"Fleiß + nicht zur Ruhe kommen, bis das Soll erreicht ist"**  
**Achtung: Aszendent wechselt kurz vor 15 Uhr zu Schütze !**  
**"Suche nach Wahrheit + von den eigenen Ideen überzeugt"**

## ■ Auszug aus Ihren Rückmeldungen

- "Da muss ich erst mal meine Glaskugel befragen"
- "Astrologie ist wahr: meine Katze ist Sternzeichen Fisch und sie isst Fisch auch sehr gerne (Aszendent Löwe)"
- "Ich glaube nicht an Astrologie, wir Widder sind misstrauisch!"
- "Die Sterne versprechen mir volle Punktzahl für dieses Blatt"
- "Gutes Gesprächsthema, wenn man Gothic-Mädchen daten will"
- "Was ist dran an der Realität?"
- "Laut Horoskop bin ich als Skorpion bekannt für meine sexuelle Ausdauer. Das stimmt eindeutig nicht ..."
- The positions of the stars and planets will ...

# Was ist dran an Astrologie 2/4

## ■ Studie aus dem Jahr 1968 ... von Michel Gauquelin

- 150 Personen (zufällig ausgewählt über ein Zeitungsinserat) bekommen ihr ganz persönliches Horoskop
- Sie werden gefragt, wie sehr sie sich darin wiedererkennen

94% sagen: ja, es passt; 90% sagen: sogar **sehr** passend

Alle Personen erhielten denselben Text, erstellt von einem Profi-Astrologen für eine (ihm nicht bekannte) Person X

Die Person X war der Serienmörder Marcel Petiot

- Das ist der sogenannte "Barnum-Effekt":  
Die Neigung vage bzw. allgemeingültige Aussagen über die eigene Person als zutreffend zu interpretieren

# Was ist dran an Astrologie 3/4

---

## ■ Weitere Studien

- S. Carlson: [A Double-Blind Test in Astrology](#), Nature 1985

Detaillierte Persönlichkeitsprofile von 128 Testpersonen

Ein Team von 26 Profi-Astrologen, die von jedem der 128 nur das Geburtsdatum (und damit das Horoskop) kannten

Aufgabe für die Astrologen: die Personen auf der Grundlage der Horoskope den Persönlichkeitsprofilen zuordnen

Das Design des Experiments wurde vorher im Detail mit dem Team von Astrologen abgestimmt

**Ergebnis: die "Trefferquote" war nicht signifikant höher als bei einer zufälligen Zuordnung**

# Was ist dran an Astrologie 4/4

---

## ■ Pro Astrologie

- Die ersten Lebensmonate sind sehr prägend, insbesondere für die Gehirnentwicklung
- Die Geburtsdaten und der Ort sind ein Hinweis auf die zu dieser Zeit vorherrschenden Witterungsverhältnisse
- Man kann sich schon vorstellen, dass das einen Einfluss hat
- Allerdings: Persönlichkeitsprofile von Menschen können sich über die Spanne ihres Lebens **sehr** stark verändern

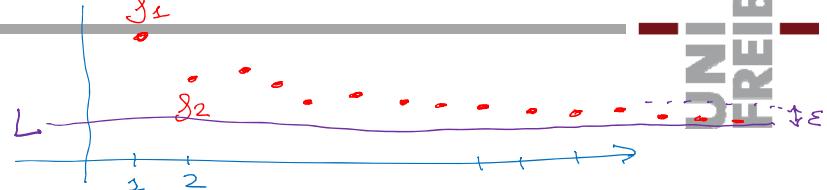
[Personality Stability from Age 14 to Age 77](#), Psych & Aging '16

174 Testpersonen, 6 Persönlichkeitsmerkmale, starke Veränderung über einen Zeitraum von 63 Lebensjahren

etwas Stabilität nur bei: "Stetigkeit" und "Gewissenhaftigkeit"

# O-Notation – Grenzwerte 1/7

## ■ Grenzwertbegriff



- Die Definitionen von der Vorlesung 3a erinnern sehr stark an den **Grenzwertbegriff** aus der **Analysis**
- **Definition:** Eine unendliche Folge  $f_1, f_2, f_3, \dots$  hat einen Grenzwert  $L$ , wenn für alle  $\varepsilon > 0$  ein  $n_0 \in \mathbb{N}$  existiert so dass für alle  $n \geq n_0$  gilt dass  $|f_n - L| \leq \varepsilon$
- In Symbolen schreibt man dann  $\lim_{n \rightarrow \infty} f_n = L$
- Eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}$  kann man genauso gut als Folge  $f(1), f(2), f(3), \dots$  auffassen und schreibt  $\lim_{n \rightarrow \infty} f(n) = L$

# O-Notation – Grenzwerte 2/7

- Beispiel für einen Beweis von einem Grenzwert  
(sollten Sie eigentlich in Mathe 1 schon mal gesehen haben)

– Es gilt:  $\lim_{n \rightarrow \infty} 1/n = 0$

$$\text{zu zeigen: } \forall \varepsilon > 0 \quad \exists m_0 \in \mathbb{N} \quad \forall m \geq m_0 \quad \left| \frac{1}{m} - 0 \right| \leq \varepsilon$$

$= L$   
 $= \frac{1}{m}$

Beweis: sei  $\varepsilon > 0$  beliebig

Finde ein  $m_0 \in \mathbb{N}$ :  $\forall m \geq m_0 \quad \frac{1}{m} \leq \varepsilon$

$$\text{z.B. } \varepsilon = \frac{1}{1000} \quad m_0 = \underline{1000} ? \quad m \geq m_0 \Rightarrow \frac{1}{m} \leq \frac{1}{m_0} \leq \frac{1}{1000}$$

$$\text{Allgemein } m_0 = \underline{\lceil 1/\varepsilon \rceil} ? \quad m \geq m_0 \Rightarrow \frac{1}{m} \leq \frac{1}{m_0} \leq \frac{1}{\lceil 1/\varepsilon \rceil} \leq \varepsilon$$

$\geq \frac{1}{\varepsilon}$   
 $\leq \frac{1}{\lceil 1/\varepsilon \rceil}$

# O-Notation – Grenzwerte 3/7

## ■ Satz

- Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}$  und der Grenzwert  $\lim_{n \rightarrow \infty} f(n)/g(n)$  existiert (evtl. ist er  $\infty$ ) ... dann gelten:

$$(1) \quad f = O(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) < \infty$$

$$(2) \quad f = \Omega(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) > 0$$

$$(3) \quad f = \Theta(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) > 0 \text{ und } < \infty$$

$$(4) \quad f = o(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) = 0$$

$$(5) \quad f = \omega(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) = \infty$$

- Wir beweisen auf der nächsten Folie Aussage (1)

Die Beweise für die anderen Aussagen gehen analog

# O-Notation – Grenzwerte 4/7

## ■ Beweis von: $f = O(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) < \infty$

Beweis von „ $\Rightarrow$ “ (Hinwendung):

$$f = O(g) \Rightarrow \exists C > 0 \ \exists m_0 \in \mathbb{N} \ \forall n \geq m_0 : \underbrace{f(n)}_{\leq C \cdot g(n)} \leq C \cdot g(n)$$

Annahme:  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

$$\Rightarrow \forall D > 0 \ \exists m_1 \in \mathbb{N} \ \forall n \geq m_1 : \frac{f(n)}{g(n)} \geq D$$

insbesondere  $\forall n \geq m_1 : \frac{f(n)}{g(n)} \geq C + 1$

Wir haben also:  $\forall n \geq m_0 : \frac{f(n)}{g(n)} \leq C ; \forall n \geq m_1 : \frac{f(n)}{g(n)} \geq C + 1$



Beweis von „ $\Leftarrow$ “ (Rückwendung):

$$\lim_{n \rightarrow \infty} f(n)/g(n) = C < \infty \Rightarrow \forall \varepsilon > 0 \ \exists m_0 \in \mathbb{N} \ \forall n \geq m_0 : \frac{f(n)}{g(n)} \leq C + \varepsilon$$

zum Beispiel für  $\varepsilon = 1$ :  $\exists m_0 \in \mathbb{N} \ \forall n \geq m_0 : f(n) \leq (C+1) \cdot g(n)$

*noch*

$$\Rightarrow f = O(g)$$

Deg. vom  
 $O(\dots)$

# O-Notation – Grenzwerte 5/7

$$\begin{aligned} 1' &= 0 \\ m' &= 1 \end{aligned}$$

$$\lim_{m \rightarrow \infty} \frac{1}{m} = \lim_{m \rightarrow \infty} \frac{0}{1} = 0$$

## ■ Variante 1: "zu Fuß"

- Dafür hatten wir gerade das Beispiel

$$\lim_{n \rightarrow \infty} \frac{1}{n} = 0$$

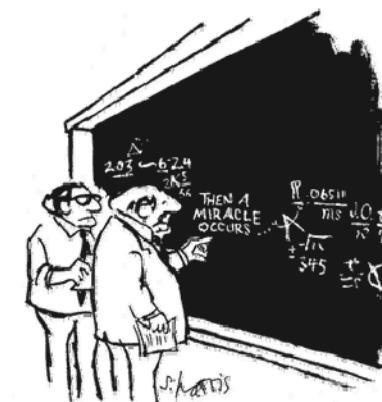
## ■ Variante 2: Regel von L'Hôpital

- Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}$  wie gehabt
- Es existieren die ersten Ableitungen  $f'$  und  $g'$ , sowie der Grenzwert  $\lim_{n \rightarrow \infty} f'(n)/g'(n)$  ... dann gilt

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

## ■ Variante 3: "sieht man doch"

- Erst mit Professur erlaubt ...



# O-Notation – Grenzwerte 6/7

$$\ln n = \log e^n \\ e = 2.71828\dots$$

## ■ Beispiel: Grenzwert mit L'Hôpital

- Was ist  $\lim_{n \rightarrow \infty} (\ln n)/n$  ?

$$\lim_{m \rightarrow \infty} \ln m = \infty \\ \lim_{m \rightarrow \infty} m = \infty \\ \frac{\infty}{\infty} \rightarrow ?$$

SEMIKOLON  
↙  
 $g(m) = \ln m ; g'(m) = \frac{1}{m}$

$$g(m) = m ; g'(m) = 1$$

$$\lim_{m \rightarrow \infty} \frac{\ln m}{m} \stackrel{\text{L'Hôpital}}{=} \lim_{m \rightarrow \infty} \frac{1/m}{1} = \lim_{m \rightarrow \infty} \frac{1}{m} = 0$$



Daraus folgt unabschiede  $\ln m = O(m)$

aber nicht  $\ln m = \Theta(m)$

## ■ Was darf man ohne Beweis annehmen?

- Gute Frage !!! Da gibt es keine klare Regel

Im Zweifelsfall immer mehr beweisen als weniger

- **Beispiel 1:**  $\lim_{n \rightarrow \infty} 1/n = 0$

Brauchen Sie nicht mehr weiter zu beweisen

- **Beispiel 2:**  $\lim_{n \rightarrow \infty} 1/n^2 = 0$

$$\lim_{n \rightarrow \infty} \frac{1}{n^2} = \left( \lim_{n \rightarrow \infty} \frac{1}{n} \right)^2 = 0$$

$\underbrace{\phantom{0}}_0 = 0$

Kann man leicht auf Beispiel 1 zurückführen

- **Beispiel 3:**  $\lim_{n \rightarrow \infty} (\log n)/n = 0$

Das sollte man beweisen, zum Beispiel mit L'Hôpital

## ■ Asymptotische Analyse

- Die  $\mathcal{O}$ -Notation schaut sich das Verhalten der Funktionen an, wenn  $n \rightarrow \infty$  geht (es interessieren nur die  $n \geq n_0$ )
- Wenn man Laufzeiten o.ä. als  $\mathcal{O}(\dots)$  etc. ausdrückt, spricht man daher von **asymptotischer Analyse**
- Vorsicht: asymptotische Analyse sagt nichts über das Verhalten bei "kleinen" Eingabegrößen ( $n < n_0$ ) aus
- Für  $n < 2$  oder  $n < 10$  ist das egal, da wird schon nichts Schlimmes passieren
- Aber das  $n_0$  ist nicht immer so klein ... **siehe nächste Folie**

# O-Notation – Diskussion 2/3

## ■ Beispiel wo das $n_0$ nicht ganz so klein ist

- Algorithmus A hat Laufzeit  $f(n) = 80 \cdot n$  „linear“  $\Theta(n)$
- Algorithmus B hat Laufzeit  $g(n) = 2 \cdot n \cdot \log_2 n$   $\Theta(n \cdot \log n)$
- Dann ist  $f = O(g)$  und sogar  $f = o(g)$  aber nicht  $g = \Theta(f)$

Insbesondere für alle  $n \geq$  irgendein  $n_0$ :  $f(n) \leq g(n)$

Das heißt, A ist asymptotisch echt schneller als B

### – Allerdings:

$$n_0 = 2^{40} = (2^{10})^4 \approx 1000^4 = 10^{12} = 1 \text{ Billionen} \quad \text{„TERA“}$$
$$n < 2^{40}: \quad g(n) = 2 \cdot n \cdot \underbrace{\log_2 n}_{\log_2 2^{40} = 40} < 80 \cdot n = f(n)$$

erst ab  $n \geq 2^{40}$  ist  $g(n) \geq f(n)$

# O-Notation – Diskussion 3/3

## ■ Mehrere Variablen

auf leicht auf  $\mathbb{N}^3, \mathbb{N}^4, \dots$   
verallgemeinbar

- Es kommt öfter mal vor, dass die Laufzeit (oder eine andere Größe) von mehr als einer Variablen abhängt
- Zum Beispiel von der Eingabegröße  $n$  und der Anzahl  $m$  der verschiedenen Elemente in der Eingabe
- Dann würden wir auch gerne sowas schreiben wie

$$O(n + m \cdot \log m)$$

$\mathbb{N} \times \mathbb{N}$

- Wir können die Definitionen aus Vorlesung 3a leicht auf Funktionen  $\mathbb{N}^2 \rightarrow \mathbb{R}$  verallgemeinern, zum Beispiel:

$$O(f) = \{ g : \mathbb{N}^2 \rightarrow \mathbb{R} \mid \exists n_0 \in \mathbb{N} \ \exists C > 0 \ \forall n, m \geq n_0 \\ g(n, m) \leq C \cdot f(n, m) \}$$

# Frageviertelstunde

---

## ■ Fragen Sie !

Sie dürfen verwenden :  $\log_x y = \frac{\ln y}{\ln x} = \frac{\log_2 y}{\log_2 x}$

# Literatur / Links

---

## ■ O-Notation / $\Omega$ -Notation / $\Theta$ -Notation

- In Mehlhorn/Sanders:

2.1 Asymptotic Notation

- In Wikipedia

[http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation)

<http://de.wikipedia.org/wiki/Landau-Symbole>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 4a, Dienstag, 16. Mai 2017  
(Assoziative Felder aka Maps)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- Erfahrungen mit dem ÜB3 O-Notation
- Kommunikation mit Tutor RZ-Kürzel
- Update Jenkins Python checkstyle

## ■ Inhalt

- Assoziative Felder Definition + Beispiele
- Anwendungsbeispiel MapCountingSort
- Bibliotheken dafür in Python / Java / C++
- ÜB4: Implementieren Sie ein assoziatives Feld + benutzen Sie es für eine konkrete Anwendung (Wortfrequenzen)

## ■ Zusammenfassung / Auszüge

- Für die meisten gut machbar (besser als das ÜB2)
- "Sogar für mathe-phobe Studierende gut lösbar"
- Beispiele in der VL seien einfacher als auf dem ÜB
- Aufgabe 3 hat Spaß gemacht
- Wie zeigt man, dass eine Funktion **nicht** in  $\Omega(\dots)$  ist?
- "Ich kann die Landau Notation langsam nicht mehr sehen"
- "Diese Landau-Notation ist wirklich eine feine Sache"
- "Unsicherheit: Wann hat man etwas komplett bewiesen"
- Bereitstellung der Videoaufnahmen großer Luxus!
- Rhythmus der Vorlesung dürfe gerne schneller werden

# Erfahrungen mit dem ÜB3 2/3

## ■ Wie zeigt man $g \neq \Omega(f)$

- Zum Beispiel  $n \neq \Omega(n^2)$  ... also  $g(n) = n$ ,  $f(n) = n^2$
- Zur Wiederholung, noch einmal die Definition von  $\Omega(f)$   
$$\{ g : \mathbf{N} \rightarrow \mathbf{R} \mid \exists n_0 \in \mathbf{N} \ \exists C > 0 \ \forall n \geq n_0 \ g(n) \geq C \cdot f(n) \}$$
- Da steht  $\exists n_0 \in \mathbf{N}$  und  $\exists C > 0$
- Also nehmen wir doch mal an, dass die existieren, und versuchen, das zu einem Widerspruch zu führen

$$\forall m \geq m_0 : \underbrace{m \geq C \cdot m}_{\Leftrightarrow 1 \geq C \cdot m \Leftrightarrow \frac{1}{C} \geq m}$$

$$\forall m \geq m_0 : m \leq \underbrace{\frac{1}{C}}_{\text{Konstante}}$$

## ■ Laufzeiten der Programme von Aufgabe 3

- Funktion **quad(n)** ... Rückgabewert  $n^2$

Anzahl Durchläufe der inneren Schleife = Endwert von "result" =  $n + (n - 1) + \dots + 1 = n \cdot (n + 1) / 2 = \Theta(n^2)$

- Funktion **id(n)** ... Rückgabewert n

Anzahl der Durchläufe der inneren Schleife = Summe aller Zähler in "counts" = n ... weil: in der ersten Schleife wird n mal genau einer der Zähler um genau 1 erhöht

- Funktion **rev(n)** ... Rückgabewert [n, ..., 1]

Die Laufzeit der Funktion index auf einem Feld der Größe m ist **nicht** konstant, sondern bis zu  $\Theta(m)$

→ Gesamlaufzeit hier  **$\Theta(n^2)$** , trotz einfacher Schleife

# Kommunikation mit Tutor

---

## ■ Erinnerung

- Sie können Ihrem Tutor / Ihrer Tutorin bei Fragen oder Problemen gerne eine Mail schreiben
- Geben Sie dabei bitte das Kürzel von Ihrem RZ-Account mit an (Initialen + Zahl)
- Ihr Tutor / Ihre Tutorin findet Sie dann leichter im System wieder

# Jenkins Update

---

## ■ Für den Python checkstyle

- Die neue Version prüft nun auch die Benennung von Funktions- und Variablennamen
- Außerdem muss die Importreihenfolge jetzt alphabetisch sein + Systemheader müssen vor anderen stehen
- Lokal bekommt man diese Änderung mit:  
`pip install pep8-naming==0.4.1 flake8-import-order==0.12`
- Auf Jenkins ist das schon installiert

Keine Sorte: das ist alles Standard und keine große Sache

## ■ Definition

- Verwalten einer Menge von  $n$  Elementen, jedes mit einem eindeutigen Schlüssel  $S$  aus einer beliebigen Menge  $U$

Terminologie: Schlüssel = **Key**, Element = **Value**

- Uns interessieren insbesondere folgende Operationen:

`insert(key, value)` Einfügen von value mit Schlüssel key

`lookup(key)` Ist key  $\in S$  und wenn ja mit welchem Wert

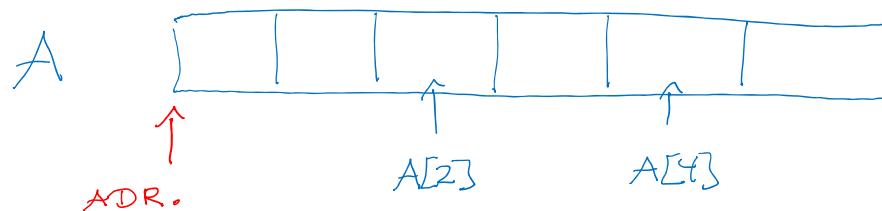
`erase(key)` Falls key  $\in S$ , dann das Element löschen

Wir schauen uns heute erstmal nur "insert" und "lookup" an

- Terminologie: ein assoziatives Feld heißt in den gängigen Programmiersprachen meistens **Map** oder **Dictionary**

## ■ Anwendungsbeispiel 1

- Das normale Feld ist ein Spezialfall mit  $S = \{0, \dots, n - 1\}$
  - Der Schlüssel von einem Element ist dann gerade die Position in dem Feld, in dem Fall genannt "Index"
  - Dann bekommt man
    - Laufzeit insert:  $\Theta(1)$
    - Laufzeit lookup:  $\Theta(1)$
    - Platzverbrauch:  $\Theta(n)$
- Einfügen von  $A[i]$ :  
 $ADR + i \cdot \text{Elementgröße}$*
- sitzt*
- besser geht's nicht
- besser geht's nicht
- besser geht's nicht



## ■ Anwendungsbeispiel 2

- Datensätze aller Informatik Studierenden mit Matrikelnummer

|         |         |
|---------|---------|
| 1234567 | Studi X |
| 3276432 | Studi Y |
| 2334523 | Studi Z |

- Die Matrikelnummer ist eindeutig pro Datensatz, von daher ist das ein geeigneter Schlüssel
- Die Schlüsselmenge S (Matrikelnummern der Infostudis) ist viel kleiner als die Menge U aller möglichen Matrikelnummern  
$$U = \{0, \dots, 9.999.999\} \dots \text{ bei bis zu 7-stelligen Nummern}$$

EINGABE:

12, 17, 17, 12, 12, 4, 4, 4, 12,  
4, 4, 17, 4, 4, 12

## ■ Anwendungsbeispiel 3

- Die verschiedenen Zahlen aus der Eingabe zu einem Sortieralgorithmus, und wie oft sie jeweils vorkommen

|    |       |
|----|-------|
| 12 | 5 mal |
| 17 | 3 mal |
| 4  | 7 mal |

$U = \{\text{Menge aller möglicher Zahlen}\}$

- Die Schlüsselmenge  $S$  ist hier ebenfalls viel kleiner als  $U$

Dazu schreiben wir nachher zusammen ein Programm

## ■ Anwendungsbeispiel 4

- Worthäufigkeiten in einem gegebenen Text

|                       |                  |
|-----------------------|------------------|
| umfang                | kommt 16 mal vor |
| zuständig             | kommt 4 mal vor  |
| fachprüfungsausschuss | kommt 88 mal vor |

- Ähnlich wie Beispiel 3, nur dass die Schlüssel jetzt Zeichenketten sind, dazu mehr in der Vorlesung morgen
- Auf dem ÜB4 sollen gerade Worthäufigkeiten berechnet werden, und zwar für die Prüfungsordnungen der Informatik

## ■ Anwendungsbeispiel 5

- Wie Beispiel 3, nur mit einem kleineren  $U$  und abstrakten Elementen, als handliches Beispiel für die nächsten Folien

5            Element 1

14          Element 2

12          Element 3

- Die Schlüssel kommen aus der Menge  $U = \{0, \dots, 19\}$
- Die genau Beschaffenheit der Elemente (Values) wird auf den nächsten Folien keine Rolle spielen: es sind einfach nur Daten, die an der betreffenden Stelle gespeichert werden

## ■ Realisierung 1

- Feld  $A$  der Größe  $|U|$ , für jeden Schlüssel  $i$  steht an  $A[i]$  das zugehörige Element, sonst ein spezieller Wert  $\times$
- Dann bekommen wir:

GUT

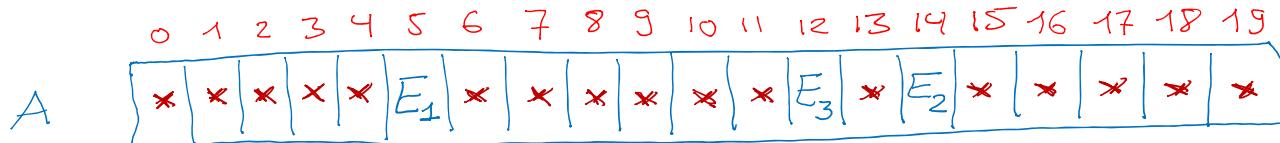
Laufzeit insert:  $\Theta(1)$  ... d.h. konstante Zeit

GUT

Laufzeit lookup:  $\Theta(1)$  # Elemente

FÜRCHTBAR

Platzverbrauch:  $\Theta(|U|)$  ... mit  $\Theta(|S|)$



## ■ Realisierung 2

- Feld der Größe  $|S|$  = Anzahl Schlüssel, an Stelle  $i$  steht einfach das  $i$ -te Element zusammen mit seinem Schlüssel
- Dann bekommen wir:

GUT

Laufzeit insert:  $\Theta(1)$  ... einfach am Ende anfügen

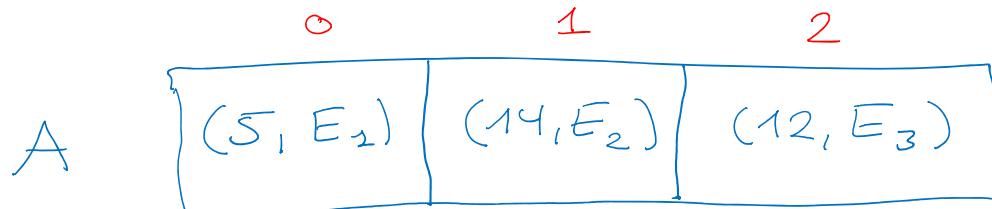
*ohne nachzuprüfen, ob Element  
schon da ist*

SCHLECHT

Laufzeit lookup:  $\Theta(|S|)$  ... kann am Anfang stehen,  
aber auch am Ende

GUT

Platzverbrauch:  $\Theta(|S|)$



5 : E<sub>1</sub>

14 : E<sub>2</sub>

12 : E<sub>3</sub>

## ■ Realisierung 3

- Wie Realisierung 2, aber sortiert

Dann geht lookup schneller (mit binärer Suche), aber insert langsamer (die Sortierung muss gewahrt bleiben)

- Dann bekommen wir:

kann bis zu  $\Theta(|S|)$  kosten

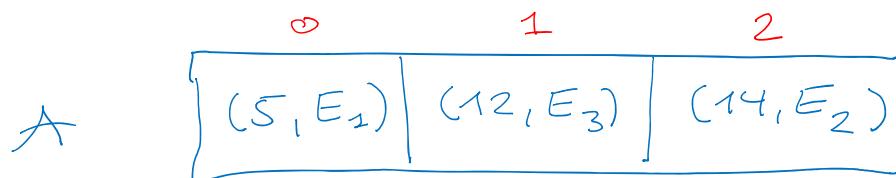
Laufzeit insert:  $\Theta(\log |S|) + \text{Einfügen in ein Feld}$

OK

Laufzeit lookup:  $\Theta(\log |S|)$  ... binäre Suche

GUT

Platzverbrauch:  $\Theta(|S|)$



## ■ Realisierung 4 (Wunsch)

- Wir hätten gerne eine Datenstruktur, mit der wir für eine Menge von  $n$  Elementen mit beliebigen Schlüsseln aus einer beliebigen Menge  $U$  bekommen:

Laufzeit insert:  $\Theta(1)$

Laufzeit lookup:  $\Theta(1)$

Platzverbrauch:  $\Theta(n)$

- Das wäre das Optimum, den besser geht es nicht
- Mit einer sog. "Hash Map" kriegt man das tatsächlich hin !  
**Dazu morgen mehr, heute erstmal, wie man das benutzt**

# MapCountingSort 1/3

## ■ Unterschied zu CountingSort

- Mit CountingSort: sortieren mit Zeit  $O(n)$  und Platz  $O(m)$  wenn die  $n$  Zahlen aus dem Bereich  $1..m$  waren
- Mit einer Map können wir das jetzt verallgemeinern zu:  
 $n$  beliebige Zahlen, aber höchstens  $m$  verschiedene

Beispiel: 17 17 3 3 17 3 3 3 3 12 12 3

Lösung:  $\underbrace{3, 3, 3, 3, 3, 3, 3}_{7 \times}, \underbrace{12, 12}_{2 \times}, \underbrace{17, 17, 17}_{3 \times}$

- Das programmieren wir jetzt zusammen in Python und benutzen dabei Pythons assoziatives Feld (= Dictionary)

# MapCountingSort 2/3

## ■ Algorithmus am Beispiel

Beispiel: 17 17 3 3 17 3 3 3 3 12 12 3

- **Schritt 1:** Mit einer Map die Anzahl Vorkommen zählen

Beispiel: 17: 3 mal , 3: 7 mal , 12: 2 mal

- **Schritt 2:** Diese Anzahlen nach den Werten sortieren

Beispiel: (3, 7) , (12, 2) , (17, 3)

Zum Sortieren in ein Feld von Paaren (Wert, Anzahl)  
schreiben, dieses Feld dann nach den Werten sortieren

- **Schritt 3:** Dann die Ausgabe schreiben wie gehabt

Beispiel: 3, 3, . . . , 3, 12, 12, 17, 17, 17

# MapCountingSort 3/3

## ■ Laufzeitanalyse

- Die Laufzeit ist  $\Theta(n \cdot M + m \cdot \log m)$ , wobei  $M$  = die durchschnittliche Kosten einer Map Operation

Schritt 1:  $\Theta(n \cdot M)$

n Map Operationen

Schritt 2:  $\Theta(m \cdot \log m)$

z.B. mit MergeSort

Schritt 3:  $\Theta(n)$

die n Werte ausgeben

- Das ist gut, wenn  $M$  klein und  $m \ll n$
- Insbesondere: linear, wenn  $M = O(1)$  und  $m = O(n / \log n)$

$$\Rightarrow m \cdot \log m \leq n$$

$m / \log m \leq \log m$

$$m \leq m / \log m \leq n$$

Rechenbeispiel für  $n$  und  $m$ :  $m = 2^{40} \approx 10^{12} = 1 \text{ Tera}$

$$\Rightarrow m / \log m = \frac{1 \text{ Tera}}{40} .$$

## ■ Python ... da heißt ein assoziatives Feld **Dictionary**

- Grundoperationen

`map = {}`

keine Typinformation nötig

`map[key] = value`

Erzeuge leere Map

`value = map[key]`

Einfügen von key mit Wert value

`if key in map: ...`

Wert für key

`del map[key]`

Fragen, ob key enthalten ist

`list(map.items())`

Löschen von key

Alle key, value Paare als Feld

## ■ Java ... da heißt ein assoziatives Feld **Map**

- Grundoperationen

K = key type, V = value type

Map<K, V> map = ...

Erzeuge leere Map

map.put(key, value);

Einfügen von key mit Wert value

value = map.get(key);

Wert für key

if (map.containsKey(key)) ...

Fragen, ob key enthalten ist

map.remove(key);

Löschen von key

map.entrySet();

Alle key, value Paare

## ■ C++ ... da heißt ein assoziatives Feld ebenfalls **Map**

- Grundoperationen

K = key type, V = value type

std::map<K, V> map;

Erzeuge leere Map

map[key] = value;

Einfügen von key mit Wert value

value = map[key];

Wert für key

if (map.count(key)) ...

Fragen, ob key enthalten ist

map.erase(key)

Löschen von key

for (auto item : map) ...

Iteration über alle key, value Paare

## ■ C++

- Achtung bei folgendem Nebeneffekt:

`if (map[key]) ...`

Fügt key mit Wert 0 ein, falls  
key bisher nicht in map war

`if (map.count(key) > 0) ...`

Fragt nur, ob key in map ist

- Das ist gefährlich, kann aber auch nützlich sein, z.B.

`map[key]++;`

Erhöht den Wert von key;  
falls key noch nicht in map,  
wird vorher mit 0 initialisiert

## ■ Effizienz

- Hängt von der Implementierung ab; effizient sind

Hashtabellen

Vorlesungen 4b, 5a, 5b

Suchbäume

Vorlesungen 8a und 8b

- In den diversen Programmiersprachen:

**Java:** `java.util.HashMap` und `java.util.TreeMap`

**C++11:** `std::unordered_map` und `std::map`

**Python:** ist dem Compiler überlassen

# Literatur / Links

---

## ■ Assoziative Arrays

- In Mehlhorn/Sanders:  
[4 Hash Tables and Associative Arrays](#) (das führt schon weiter)
- In Wikipedia
  - [http://de.wikipedia.org/wiki/Assoziatives\\_Feld](http://de.wikipedia.org/wiki/Assoziatives_Feld)
  - [http://en.wikipedia.org/wiki/Associative\\_array](http://en.wikipedia.org/wiki/Associative_array)
- In Python, Java, C++
  - <http://docs.python.org/tutorial>
  - <http://docs.oracle.com/javase>
  - <http://www.cplusplus.com/reference>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 4b, Mittwoch, 17. Mai 2017  
(Hash Maps, Rehash, Cuckoo Hashing)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Drumherum

- Placebo-Effekt

Einbildung?

## ■ Inhalt

- Hashtabellen

Prinzip + Beispiele

- Dynamisches Hashing

Rehash

- Cuckoo Hashing

Prinzip + Beispiel

# Placebo-Effekt 1/4

---

## ■ Ist der Placebo-Effekt Einbildung?

- "Meine Mama sagt immer: Einbildung ist auch ne Bildung"
- "Das ist wie einen leeren aber mit blatt-xx beschrifteten Ordner committen: man bekommt trotzdem 0 Punkte"
- "Der Placebo-Effekt ist in dem Moment real, wenn dadurch irgendetwas bewirkt wurde"
- "Er ist nicht Einbildung, er basiert auf Einbildung"
- "Wenn es durch die Anwendung eines Placebos zu einer tatsächlichen Besserung beim Patienten kommt, ist es irreführend, diese Wirkung als Einbildung zu bezeichnen"
- "Placebo-Effekt ist Einbildung, Medikamente helfen wirklich"
- "Sind wir Einbildungen?"

# Placebo-Effekt 2/4

---

- Eine bekannte und vielzitierte Studie von 2002
  - B. Moseley et al: [A Controlled Trial of Arthroscopic Surgery for Osteoarthritis of the Knee](#), N Engl J Med 2002
  - Hintergrund: in den USA 650.000 Knie-OPs / Jahr wegen Knie-Arthrose und der damit einhergehenden Schmerzen
  - Doppelblinder Vergleich von drei Behandlungen (je 60 Pat.)

**Lavage:** Spülung des Gelenkes

**Débridement:** Wundreinigungs-OP

**Placebo:** Simulation einer Wundreinigungs-OP inklusive Schnitte und Narben, aber ohne die eigentliche Maßnahme

- Ergebnis: Funktionsfähigkeit und Schmerzlinderung waren in der Placebo-Gruppe zeitweise am besten !

# Placebo-Effekt 3/4

---

## ■ Ist Placebo besser als Nicht-Behandlung?

- Auch hierzu gibt es zahlreiche Studien
- Die Ergebnislage ist gemischt

Bei vielen Problemen (z.B. Depression, Bluthochdruck, Schlaflosigkeit) wird in der Regel kein Unterschied zwischen Placebo und Nicht-Behandlung gefunden

Bei Schmerz-Problemen (z.B. Arthrose) aber teilweise deutliche Unterschiede bezüglich Schmerz **und** Funktion

Interessant: im letzteren Fall sind Schnitte, Injektionen, Nadeln, etc. effektiver als Pillen

- Ein Fazit daraus ist definitiv: anstatt sich darüber zu wundern, sollte es sich die Medizin zu Nutze machen!

# Placebo-Effekt 4/4

## ■ Nutzen in der Medizin

- Bei alternativen Heilmethoden heißt es oft abwertend: "Das ist doch nur Placebo"
- Allerdings ist die positive Wirkung vieler klassischer Maßnahmen (Medikamente / OPs) auch "nur Placebo"

Nur dass oft noch Nebenwirkungen dazu kommen

- Ärzte geben oft Quasi-Placebos ("harmlose" Medikamente) wenn der Patient unbedingt eine Behandlung wünscht
- Nicht-Intervention ist wohl bei einem Großteil aller Behandlungsanliegen (natürlich nicht bei allen) die beste Therapie

Voltaire: "Das Geheimnis der Medizin besteht darin, den Patienten abzulenken, während die Natur sich selber hilft"

# Hash Map 1/8

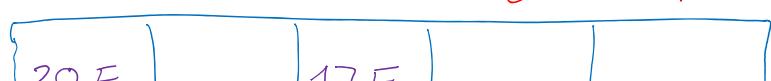
## ■ Grundidee

- **Hash-Tabelle:** ein Feld  $A$  der Größe  $m$
  - **Hash-Funktion:** eine Funktion  $h : U \rightarrow \{0, \dots, m - 1\}$
  - Speichere Element mit Schlüssel  $x$  unter  $A[h(x)]$

Problem: Kollisionen  $\rightarrow x_1 \neq x_2$  mit  $h(x_1) = h(x_2)$

  - Beispiel mit  $m = 5$  und  $h(x) = x \bmod 5$
  - Wir fügen nacheinander ein:  $(17, E_1), (20, E_2), (12, E_3)$

|   | 0         | 1 | 2         | 3 | 4 | x  | $\varrho(x)$ |
|---|-----------|---|-----------|---|---|----|--------------|
| A | $20, E_2$ |   | $17, E_1$ |   |   | 17 | 2            |
|   |           |   |           |   |   | 20 | 0            |
|   |           |   |           |   |   | 12 | 2            |
|   |           |   |           |   |   |    |              |


  
 ↗  $\varrho(17) = 2$    ↗  $\varrho(12) = 2$

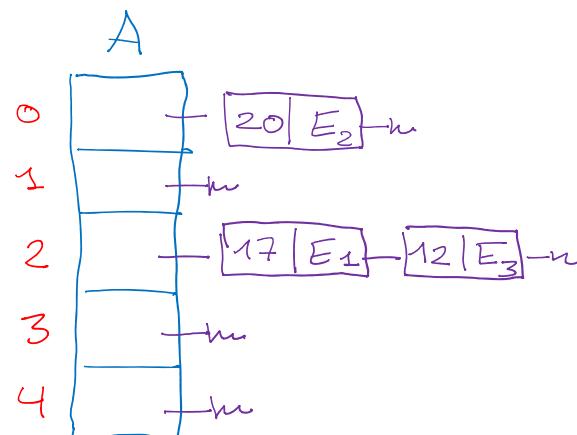
Kollision  
 warum?

# Hash Map 2/8

## ■ Kollisionen, Lösung 1

- Hashing mit **Verkettung**
  - Jeder Eintrag der Hashtabelle kann nicht nur ein key-value Paar speichern, sondern eine Menge davon
- Array<Array<KeyValuePair>> hashTable;
- Beispiel mit  $m = 5$  und  $h(x) = x \bmod 5$  und Operationen:

$\begin{array}{l} h(17)=2 \\ \text{insert}(17, E_1) \\ h(20)=0 \\ \text{insert}(20, E_2) \\ h(12)=2 \\ \text{insert}(12, E_3) \\ h(32)=2 \\ \text{lookup}(17) \rightarrow E_1 \\ \text{lookup}(32) \rightarrow \text{nix} \end{array}$



# Hash Map 3/8

→ ERASE geht nicht so ohne Weiteres!

## ■ Kollisionen, Lösung 2

- Hashing mit offener Adressierung
- Wenn eine Zelle schon besetzt ist, solange "ein Zelle weiter" gehen, bis man eine freie Zelle findet
- Man braucht dafür einen speziellen Schlüssel "Zelle ist frei" *nix*
- Beispiel mit  $m = 5$  und  $h(x) = x \bmod 5$  und Operationen:

$h_2(x) = 2$   
 $\text{insert}(17, E_1)$

$h_2(x) = 0$   
 $\text{insert}(20, E_2)$

$h_2(x) = 2$   
 $\text{insert}(12, E_3)$

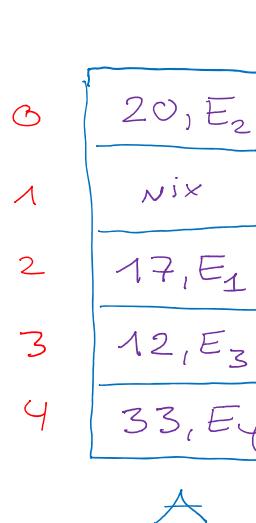
$A[2]$  ausdrucken  
 $\text{lookup}(17) \rightarrow E_1$

$A[2]$  ausdrucken  
 $A[3]$  ausdrucken  
 $\text{lookup}(32) \rightarrow \text{nix}$

$A[4]$  ausdrucken  
 $\text{insert}(33, E_4)$

$A[2], A[3], A[4]$   
 $A[0], A[1]$   
 $\text{ausdrucken}$   
 $\text{lookup}(32) \rightarrow \text{nix}$

Z.B.  $i \rightarrow (i+1) \bmod 5$   
 es genüge aber auch jede  
 andere deterministische  
 Formel



Gespezifisch  
 nur bei  
 $\leq m$  Schlüssel

# Hash Map 4/8

---

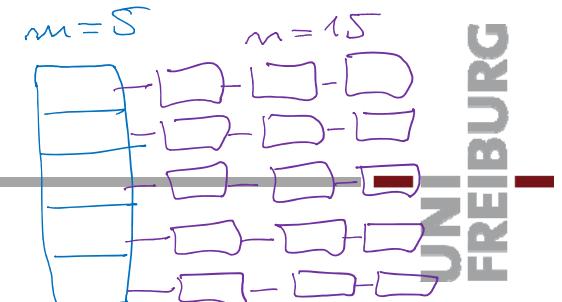
## ■ Vorgehen insert / lookup / erase

- Im schlechtesten Fall muss man alle Elemente durchgehen, deren Schlüssel auf denselben Wert abgebildet werden

Bei lookup kann man aufhören, wenn man den Schlüssel gefunden hat (mit Glück schon am Anfang)

Bei Hashing mit Verkettung, kann man bei insert einfach am Ende anfügen

# Hash Map 5/8



## ■ Laufzeit ... bei Hashing mit Verkettung

- **Best case:** die  $n$  Schlüssel werden von der Hashfunktion gleichmäßig verteilt

Dann gehen insert und lookup in Zeit  $O(n / m)$

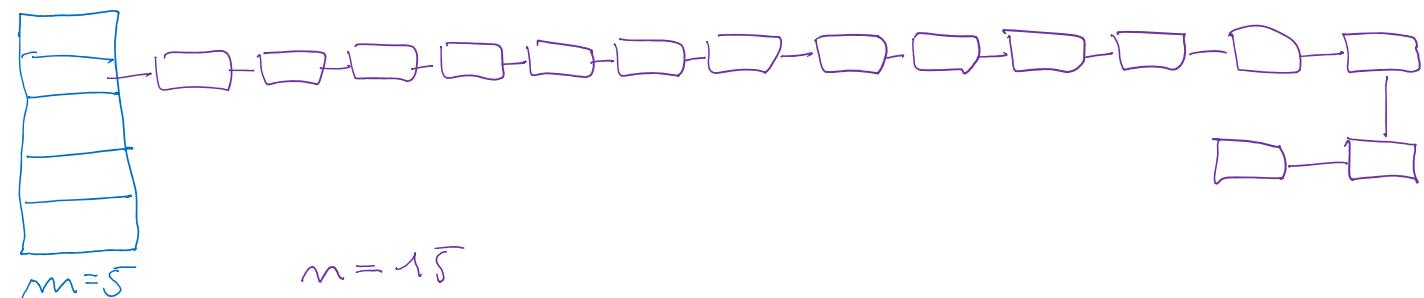
Falls  $n = O(m)$  also in Zeit  $O(1)$

$$\text{z.B. } \frac{m = 1.000.000}{m = 100.000} \Rightarrow m/m = 10$$

- **Worst case:** alle  $n$  Schlüssel werden von der Hashfunktion auf denselben Wert abgebildet

Dann braucht lookup im schlechtesten Fall  $\Theta(n)$

Wie bei Realisierung 2 (Folie 15 von Vorlesung 4a gestern)



# Hash Map 6/8

## ■ Wahl der Hashfunktion, zufällige Schlüssel

- Bei zufällig verteilten Schlüsseln gibt die einfache Funktion  $h(x) = x \bmod m$  schon die bestmögliche Verteilung

**Intuitiv:** für zufälliges  $x$  ist auch  $x \bmod m$  zufällig aus  $\{0, \dots, m - 1\}$ , und so bekommt jede Zelle der Hashtabelle im Erwartungswert gleich viele Schlüssel (und zwar  $n / m$ )

Das machen wir nächste Woche genauer

Ich gebe Ihnen da auch einen kleinen Auffrischungskurs in Wahrscheinlichkeitsrechnung ([#endlichwiedermathe](#))

# Hash Map 7/8

---

## ■ Wahl der Hashfunktion, nicht-zufällige Schlüssel

- Bei nicht-zufällig verteilten Schlüsseln kann die Hashfunktion  $h(x) = x \bmod m$  beliebig schlecht sein
- Beispiel:  $m = 10$  und Schlüssel 21, 11, 51, 71, 61, ...
- Was man dagegen macht, sehen wir nächste Woche

Für das ÜB4 können Sie trotzdem einfach  $h(x) = x \bmod m$  verwenden ... und feste daran glauben, dass es klappt

# Hash Map 8/8

$$\begin{aligned} h("dog") &= \\ (100 + 111 + 111 + 102) \bmod 5 &= 4 \end{aligned}$$

$$m = 5$$

## ■ Schlüssel, die keine Zahlen sind

- **Option 1:** Jedes im Rechner repräsentierte Objekt kann als Zahl aufgefasst werden, zum Beispiel

Sei Objekt in  $k$  Bytes  $B_0 \dots B_{k-1}$  repräsentiert, dann entspricht der Inhalt dieser Bytes eindeutig der Zahl  $\sum_{j=0, \dots, k-1} B_j \cdot 256^j$

- **Option 2:** Objekt direkt auf  $\{0, \dots, m - 1\}$  abbilden (= "hashen"), ohne Umweg über eine Zahl, z.B. für string  $s$

$h(s)$  = Summe der ASCII-Codes der Zeichen mod  $m$

Das können Sie zum Beispiel für das ÜB4 verwenden

Aber wieder keine Garantie, dass es gut verteilt ist, die gibt es erst ab nächster Woche

# Rehash 1/4

## ■ Bisherige Annahme

- Die Schlüsselmenge  $S$  ist vorher bekannt  
*bei Verretung; bei off. Addr.  $m \geq 2n$  oder so*
- Dann kann man leicht die Größe der Hashtabelle als  $m = \Theta(n)$  wählen, so dass die Anzahl Schlüssel, die auf denselben Wert abgebildet werden im besten Fall  $\Theta(1)$  ist

Dann gehen auch insert und lookup in Zeit  $\Theta(1)$

- Es können aber zwei Dinge passieren
    - Es kommen Schlüssel dazu (und wir wissen vorher nicht, wie viele) und die Hashtabelle wird zu klein
    - Wir haben Pech und es werden übermäßig viele Schlüssel auf denselben Wert abgebildet
- Das Gute daran: beides kann man leicht feststellen

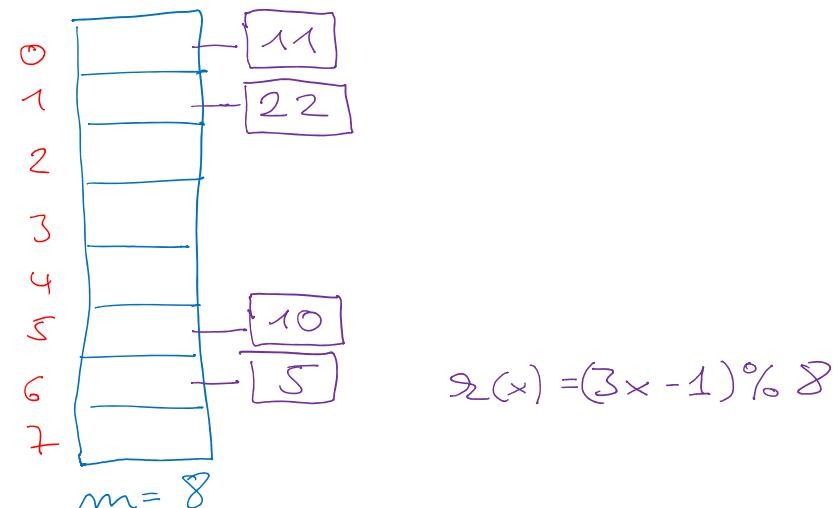
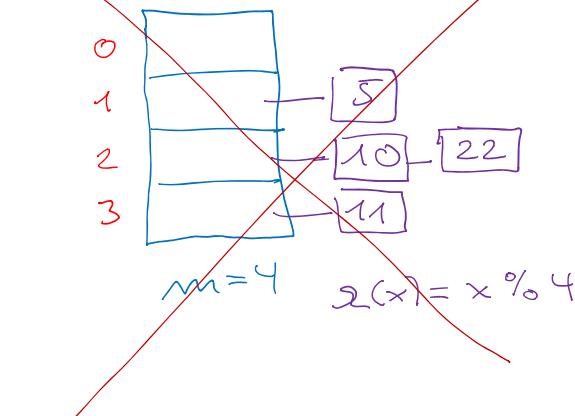
*insbesondere die Anzahl ist*

# Rehash 2/4

in der neuen Tabelle,  
neue H-Funktion verwenden

## ■ Lösung für beide Probleme: Rehash

- Bei einem Rehash macht man einfach Folgendes:
  1. Eine neue Hashfunktion auswählen
  2. Die Elemente von der alten in die neue Tabelle kopieren
  3. Die alte Tabelle löschen
- Beispiel:  $S = \{5, 10, 11, 22\}$ , alte Fkt  $h(x) = x \bmod 4$ ,  
neue Hashfunktion  $h(x) = 3 \cdot x - 1 \bmod 8$



# Rehash 3/4

---

## ■ Kosten für einen Rehash

- Ein Rehash ist teuer: er kostet Zeit  $\Theta(n)$ , wobei  $n$  die Anzahl Elemente zum Zeitpunkt des Rehash ist
- Wenn man es richtig macht, ist er allerdings selten nötig:  
Mit clever gewählten Hashfunktionen (siehe nächste Woche) ist das unwahrscheinlich

Wenn die Hashtabelle zu klein geworden ist, und man die neue Hashtabelle doppelt so groß wählt ( $m \rightarrow 2m$ ), dauert es lange, bis man wieder vergrößern muss

Diese "Verdoppelungsstrategie" analysieren wir in Vorlesung 6a und 6b genauer (amortisierte Analyse)

# Rehash 4/4

---

## ■ Verkleinerung der Schlüsselmenge

- Die Schlüsselmenge kann auch wieder kleiner werden, indem Schlüssel gelöscht werden

Python: **del**    Java: **remove**    C++: **erase**

- Wenn  $|S| \ll m$  wird, kann man die Hashtabelle auch wieder verkleinern ... **siehe ebenfalls Vorlesung 6a+b**
- Macht man aber in der Praxis oft nicht, weil:
  1. In sehr vielen Anwendungen braucht man nur **insert** und **lookup**, kein **del / remove / erase**
  2. Zu irgendeinem Zeitpunkt braucht man sowie den Platz für die maximale Anzahl Schlüssel der Anwendung

## ■ Beschreibung des Algorithmus

- Es gibt eine Hashtabelle der Größe  $m$  wie gehabt
- Es gibt **zwei** Hashfunktionen  $h_1, h_2 : U \rightarrow \{0, \dots, m - 1\}$
- Jede Position der Hashtabelle hat nur Platz für **ein** Element
- Versuche ein neues Element  $x$  bei  $h_1(x)$  zu speichern
- Falls schon belegt von einem Element  $y$ , dann speichere  $y$  bei  $h_i(y)$  falls vorher bei  $h_j(y)$  gespeichert,  $\{i, j\} = \{1, 2\}$ 

Für jeden Schlüssel gibt es also **genau** zwei mögliche Plätze
- Falls neuer Platz für  $y$  belegt von einem Element  $z$ :  
dann verfare genau so mit  $z$  ... und so weiter

## ■ Zyklus

- Es kann so zu einem **Zyklus** kommen, und zwar wenn für eine Teilmenge von Schlüsseln  $S'$  gilt:

$$|\{h_1(x) : x \in S'\} \cup \{h_2(x) : x \in S'\}| < |S'|$$

**Intuitiv: es gibt weniger Plätze als Schlüssel**

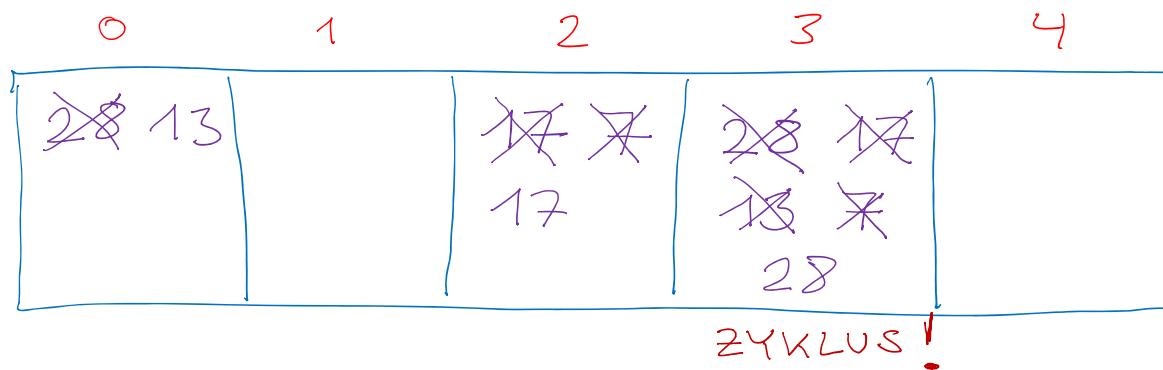
- Wenn das passiert, wählt man neue Hashfunktionen  $h_1$  und  $h_2$  und macht einen **Rehash** wie erklärt

# Cuckoo Hashing 3/5

| $x$ | $z_1(x)$ | $z_2(x)$ |
|-----|----------|----------|
| 17  | 2        | 3        |
| 28  | 3        | 0        |
| 7   | 2        | 3        |
| 13  | 3        | 0        |

## ■ Beispiel ... ohne Rehash, nur mit Schlüsseln, ohne Elemente

- Mit:  $m = 5$ ,  $h_1(x) = x \bmod 5$ ,  $h_2(x) = 2x - 1 \bmod 5$
- Füge nacheinander ein: 17, 28, 7, 13



weil für 17, 28, 7, 13  
4 Schlüssel

nur Plätze 0, 2, 3  
3 Plätze

## ■ Wahl der Hashfunktionen

- Sollte man **unabhängig** voneinander wählen, so dass jede für sich eine gute Hashfunktion wäre
  - D.h. die Schlüssel werden möglichst gleichmäßig verteilt
  - Dazu mehr nächste Woche, wie man das hinkriegt
- Dann kann man zeigen, dass es hinreichend selten zu einem Zyklus (und dem dann nötigen teuren Rehash) kommt, solange  $|S| \leq m/2$
- Bei wachsender Schlüsselmenge wie gehabt ein Rehash mit  $m \rightarrow 2m$ , zum Beispiel sobald  $|S| > m/2$

## ■ Laufzeit

- Die Laufzeit von `lookup(x)` ist **immer  $\Theta(1)$**

Man muss ja immer nur an zwei Positionen nachschauen,  
nämlich  $h_1(x)$  und  $h_2(x)$  ... **das ist gerade der Clou**

- Dasselbe gilt dann auch für `remove(x)`

An beiden Positionen nachschauen, und wo gefunden  
einfach löschen, die Position ist dann wieder frei

- Man kann zeigen, dass ein `insert(x)` **im Durchschnitt** in  
Zeit  $\Theta(1)$  geht

Beweis siehe Referenzen ... aber nicht Klausur-Elefant

# Literatur / Links

---

## ■ Universelles Hashing

- In Mehlhorn / Sanders:  
    4 Hash Tables and Associative Arrays
- In Wikipedia  
[http://en.wikipedia.org/wiki/Hash\\_table](http://en.wikipedia.org/wiki/Hash_table)

## ■ Cuckoo Hashing

- [http://en.wikipedia.org/wiki/Cuckoo\\_hashing](http://en.wikipedia.org/wiki/Cuckoo_hashing)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 5a, Dienstag, 23. Mai 2017  
(Universelles Hashing, Teil 1)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- Feedback Tutoren zum ÜB3 O-Notation
- Erfahrungen mit dem ÜB4 eigene Hash Map
- Mathe Kurzanleitung

## ■ Inhalt

- Universelles Hashing Erklärung + Theorem
- Wahrscheinlichkeitsrechnung Crash-Kurs
- ÜB5: Berechnen sie die Kollisionswahrscheinlichkeiten für 4 Klassen von Hashfunktionen (aus der Vorlesung 5b)

Achtung: es gibt in der Klausur **mit Sicherheit** eine Aufgabe zu universellem Hashing (wie bisher auch immer)

# Feedback Tutoren zum ÜB3

- Rückmeldung aus den Korrekturen
  - Öfter sowas wie  $O(n^2 + n + 10)$  anstatt  $O(n^2)$
  - Das  $n_0$  wurde oft vergessen bzw. nicht explizit genannt
  - Probleme mit Implikationspfeilen und ihrer Richtung
  - Redundante Semikolons im Python-Code auf dem ÜB3
  - Bei den Aufgaben 2 oder 3 oft wenig oder keine Begründung  
**Sowas gibt in der Klausur definitiv (erheblichen) Punktabzug**
  - Man schreibt  $n = O(n) = O(n^2)$  statt  $f \in O(n) \subseteq O(n^2)$
  - Die Ableitung von  $2^n$  nach  $n$  ist definitiv nicht  $n \cdot 2^{n-1}$
  - Immer hinschreiben: 1. Was ist gegeben, 2. was ist zu zeigen

# Erfahrungen mit dem ÜB4

---

## ■ Zusammenfassung / Auszüge

- Die Aufgabe (und die Anwendung) hat vielen Spaß gemacht
- Endlich wieder ein ÜB mit Programmieren
- Dankbar für den Code zum Auslesen der Wörter
- Probleme mit Gnuplot ... **habe ich aber in VL 1a vorgemacht**
- "Aufgabe 1: ging sau schnell, war nur falsch"
- "Fand es schwierig zu verstehen, was man von mir wollte"
- "Live-Coding relativ langweilig im Vergleich zur Theorie, aber beim Rest scheint es ja gut anzukommen"
- Rückmeldung von Tutor/in sehr hilfreich

## ■ Kurzanleitung

einige Umformungen  $\underline{A = B}$   
eine Gleichung  $\underline{\underline{A = B}}$

eine Implikation  $\underline{A \Rightarrow B}$   
*linke Seite*  $\uparrow$

- Hinschreiben: 1. Was ist gegeben? + 2. Was ist zu zeigen?
- Variante 1: "normaler" Beweis

Die linke Seite von "was ist zu zeigen" hinschreiben + darauf anwenden, was sonst noch gegeben ist + Ausdrücke vereinfachen ... bis man zu dem kommt, was gezeigt werden soll

- Variante 2: Widerspruchsbeweis *wie selbmer*
- Annehmen, dass das Gegenteil von dem gilt, was zu zeigen ist + umformen wie oben ... bis man zu etwas kommt, was nicht sein kann (aber nicht weil man sich verrechnet hat!)
- Die meisten Beweise in dieser Vorlesung brauchen eine oder keine Idee, der Rest geht im Wesentlichen "automatisch"

# Universelles Hashing 1/10

## ■ Zur Erinnerung

- Wenn die Schlüsselmenge zufällig ist, tut es auch die einfache Hashfunktion  $h(x) = x \bmod m$
- Für bestimmte Schlüsselmengen kann diese Funktion dagegen beliebig schlecht sein, zum Beispiel
$$h(x) = x \bmod 10 \text{ und } x = \begin{matrix} 2(x) & 2 & 2 & 2 & 2 & 2 \\ 12, 42, 32, 72, 102, \dots \end{matrix}$$
- Allgemeiner: keine einzelne Hashfunktion  $h$  kann für alle Schlüsselmengen gut sein, weil:  
*z.B. alle Wörter*  
 $h$  ist Funktion von  $U$  nach  $\{0, \dots, m - 1\}$  und  $|U| \gg m$   
Selbst im besten Fall werden so  $|U| / m$  Schlüssel auf denselben Wert abgebildet

# Universelles Hashing 2/10

## Idee

z.B.  $m = 101$ ,  $101 = 100$ ,  $p = 101$

dann wäre z.B.

$$g_2(x) = (57x + 83) \bmod 101 \bmod 10$$

einige Fkt. aus der Klasse wären

$p$  ist eine Primzahl

mit  $p \geq |U|$

- Eine Menge (Klasse) von Hashfunktionen zur Auswahl
- Und zwar so, dass man leicht ein zufälliges Element aus dieser Menge wählen kann
- Beispiel:  $h(x) = a \cdot x \bmod m$  mit  $a \in \{0, \dots, m-1\}$

Das sind (nur)  $m$  verschiedene Hash-Funktionen

Wir sehen morgen, dass das keine gute Klasse ist

gut für die Klasse

$$U = \{0, \dots, |U|-1\}$$

- Beispiel:  $h(x) = (a \cdot x + b) \bmod p \bmod m$  mit  $a, b \in U$

Das sind  $|U|^2$  verschiedene Hash-Funktionen

Wir sehen morgen, dass das eine sehr gute Klasse ist

Wie findet man  $p \geq |U|$ ,  $p$  min?

ANTWORT: gute Frage! Siehe ÜBS vom letzten Jahr.

# Universelles Hashing 3/10

## Was ist eine gute Klasse (informal)

- Eine Klasse  $H$  von Hashfunktionen ist dann gut, wenn:

Für jede Schlüsselmenge  $S$  gibt es viele Funktionen in  $H$ , die  $S$  "möglichst gut über die Hashtabelle verteilen"

Das machen wir auf den nächsten Folien formaler

- Dann können wir einfach eine zufällige Funktion aus  $H$  wählen und hoffen, dass alles gut klappt

Wenn nicht, merken wir das und machen nach einer Weile einen Rehash, mit einer anderen Funktion aus  $H$

Wenn  $H$  die obige Eigenschaft hat, wird das nicht oft passieren und "im Mittel" gut funktionieren

## ■ Zufälliges Werfen

z.B.  $m = 10, |S| = 50$   
 $\Rightarrow |S|/m = 5$

- Schlüsselmenge  $S$ , Hashtabelle  $T$  mit  $m$  "Plätzen"
- Die beste Art  $S$  möglichst gut über  $T$  zu verteilen" ist, wenn jeder Platz genau  $|S| / m$  Schlüssel abbekommt
- Das erreicht man mit **zufälligem Werfen**:

Für jedes  $x \in S$ , wähle einen zufälligen Platz in  $T$

Dann ist die erwartete Anzahl Elemente an einem bestimmten Platz genau  $|S| / m$

- Das beweisen wir jetzt erstmal
- Vorher ein Crash-Kurs Wahrsch.keitsrechnung (Folien 15–18)
- Zur Auffrischung oder um es zum ersten Mal zu verstehen

# Universelles Hashing 5/10

## ■ Zufälliges Werfen, Verteilung

- Schlüsselmenge  $S$ , Hashtabelle mit  $m$  Plätzen
  - Sei  $h(x)$  der Platz von Schlüssel  $x$
  - Sei  $S_i = \{ x \in S : h(x) = i \}$
  - Wir zeigen jetzt, dass  $E(|S_i|) = |S| / m$

mit einem uns Si für ein freies an.

Definie Indikatorvariable  $I_x = 1$  genau dann wenn  $\vartheta(x) = i$   
Folie 19  $I_x = 0$  sonst.

$$E(I_x) = \Pr(I_x=1) = \frac{1}{m}$$

$$|S_G| = \sum_{x \in S} I_x \quad \dots \text{wie auf Folie 19}$$

$$\Rightarrow E(|S|) = E\left(\sum_{x \in S} I_x\right) = \sum_{x \in S} E(I_x) = |S| / m$$

## ■ Zufälliges Werfen, Problem

- "Zufälliges Werfen" ist keine gute Klasse von Hashfunktionen
- Die Hashfunktionen, die wir uns bisher angeschaut haben, waren sehr leicht zu speichern und auszuwerten
- Zum Beispiel:  $h(x) = x \bmod m$

Kann man in  $O(1)$  Platz speichern und in  $O(1)$  Zeit auswerten für einen beliebigen Schlüssel  $x$  aus dem Universum

- Für einen zufälligen Wurf nicht so einfach:  
Man müsste sich für jeden Schlüssel im Universum (oder zumindest in  $S$ ) merken, wo er hingeworfen wurde  
**Denken Sie darüber nach: man bräuchte eine Hash Map, um das effizient abzuspeichern / darauf zuzugreifen**

## ■ Ziel

- Unser Ziel ist eine Klasse von Hashfunktionen für die gilt:
  1. Eine zufällige Funktion aus dieser Klasse verhält sich (fast) genauso gut wie zufälliges Werfen
  2. Man kann leicht eine zufällige Funktion auswählen, abspeichern und für einen Schlüssel  $x$  auswerten
- Die Definition von universellem Hashing versucht genau Eigenschaft 1 formal zu fassen ... siehe nächste Folie

## ■ Definition

- Sei  $H$  eine Menge von Hashfunktionen  $U \rightarrow \{0, \dots, m-1\}$
- $H$  ist  $c$ -universell wenn für alle  $x, y \in U$  mit  $x \neq y$  gilt:

$$|\{h \in H : h(x) = h(y)\}| \leq c \cdot |H| / m$$

Heißt:  $x$  und  $y$  fallen unter einer Brücke von  $H$

- Mit anderen Worten, wenn  $h \in H$  zufällig gewählt, dann

$$\text{Prob}(h(x) = h(y)) \leq c \cdot 1 / m$$

- Bei zufälligen Werten gilt das mit  $c = 1 \dots$  und man kann zeigen, dass besser als  $c = 1$  auch nicht geht

Morgen sehen wir Klassen von Hash-Funktionen mit  $c = 1$  oder  $c = 2$ , was für praktische Zwecke oft genauso gut ist

# Universelles Hashing 9/10

## ■ Zentraler Satz

- Sei  $H$  eine  $c$ -universelle Klasse von Hashfunktionen
- Sei  $S$  eine Menge von Schlüsseln und  $h \in H$  zufällig gewählt
- Für ein  $x \in S$  sei  $S_x$  die Menge der Schlüssel  $y$  mit  $h(y) = h(x)$
- Dann ist  $E(|S_x|) \leq 1 + c \cdot |S| / m$  ...  $|S|/m$  wäre optimal  
 $m = 100, |S| = 200, c = 2 \Rightarrow E(|S_x|) \leq 1 + 2 \cdot \frac{200}{100} = 5$
- Insbesondere: Falls  $m = \Omega(|S|)$  gilt  $E(|S_x|) = O(1)$
- Das beweisen wir auf der nächsten Folie
- Man beachte: die vermeintlich einfachere Aussage, dass  $E(|S_i|) \leq 1 + c \cdot |S| / m$  gilt im Allgemeinen **nicht**

Das macht aber nichts, für z.B. die Laufzeit von insert bei Hashing mit Verkettung reicht auch die Aussage von oben

# Universelles Hashing 10/10

gegeben:  $\exists \text{ zu jedem } x \neq y \Rightarrow \Pr(\varphi(x) = \varphi(y)) \leq \frac{c}{m}$

## ■ Beweis von $E(|S_x|) \leq 1 + c \cdot |S| / m$

Zu zeigen  $\underline{E(|S_x|)} \leq \dots$

↳ damit fangen wir an (im Kurs)

$$I_y = 1 \text{ gdw } \varphi(y) = \varphi(x)$$

$$I_y = 0 \text{ sonst}$$

für  $y \in S$

→ nach Def universell.

$$E(I_y) = \Pr(I_y = 1) \leq \frac{c}{m} \dots \text{ für } y \in S \setminus \{x\}$$

$$\text{für } y = x : \Pr(I_y = 1) = 1$$

→  $x$  verschwindet

$$E(|S_x|) = E\left(1 + \sum_{y \in S \setminus \{x\}} I_y\right) = 1 + \sum_{y \in S \setminus \{x\}} E(I_y) \leq \frac{c}{m}$$

$$\leq 1 + c \cdot \frac{|S|-1}{m}$$

$$\leq 1 + c \cdot |S| / m \quad \text{QED}$$

# Wahrscheinlichkeitsrechnung 1/4

## ■ Wahrscheinlichkeitsraum / Ereignisse

- Wir beschränken uns hier auf den diskreten Fall
- Wahrscheinlichkeitsraum  $\Omega$  von sog. Elementarereignissen
- Die haben Wahrscheinlichkeiten ... Bedingung  $\sum_{e \in \Omega} \Pr(e) = 1$
- Ereignis  $E$  = Teilmenge von  $\Omega$ , Wahrsch.  $\Pr(E) = \sum_{e \in E} \Pr(e)$
- Zum Beispiel: zweimal würfeln, dann  $\Omega = \{1, \dots, 6\}^2$

Jedes  $e$  aus  $\Omega$  hat dann Wahrscheinlichkeit  $\Pr(e) = 1/36$

$E$  = beide Augenzahlen sind gerade, dann  $\Pr(E) = \frac{3 \cdot 3}{36} = \frac{1}{4}$

3 · 3 der Elementarereignisse oben  
haben diese Eigenschaft

$$\Omega = \{(1,1), (1,2), \dots, (1,6), (2,1), (2,2), \dots, (2,6), (3,1), (3,2), \dots, (3,6), (4,1), (4,2), \dots, (4,6), (5,1), (5,2), \dots, (5,6), (6,1), (6,2), \dots, (6,6)\}$$

## ■ Zufallsvariable

- ... weist einem Ausgang des Zufallsexperiments eine Zahl zu
- Zum Beispiel:  $X$  = Summe Augenzahlen bei zweimal Würfeln
- Sowas wie  $X = 12$  oder  $X \geq 7$  sind dann einfach Ereignisse
- Beispiel 1:  $\text{Prob}(X = 2) = \frac{1}{36}$  *Erl. Ereignisse: (1,1)*
- Beispiel 2:  $\text{Prob}(X = 4) = \frac{3}{36} = \frac{1}{12}$  *Erl. Ereignisse: (1,3), (2,2), (3,1)*
- **Erwartungswert** ist definiert als  $E(X) = \sum k \cdot \text{Pr}(X = k)$

Intuitiv: gewichtetes Mittel der möglichen Werte von  $X$ , wobei die Gewichte die Wahrscheinlichkeiten der entspr. Werte sind

- Beispiel von oben:  $X$  = Summe Augenzahl zweimal Würfeln

$$E(X) = 2 \cdot \underbrace{\text{Pr}(X=2)}_{=\frac{1}{36}} + 3 \cdot \underbrace{\text{Pr}(X=3)}_{=\frac{1}{18}} + \dots + 12 \cdot \underbrace{\text{Pr}(X=12)}_{=\frac{1}{36}}$$

## ■ Summe von Erwartungswerten

- Für beliebige (diskrete) Zufallsvariablen  $X_1, \dots, X_n$  gilt

$$E(X_1 + \dots + X_n) = E(X_1) + \dots + E(X_n)$$

Gilt auch, wenn die  $X_1, \dots, X_n$  **nicht** unabhängig sind!

- Beispiel: Summe Augenzahl bei zweimal Würfeln
- Sei  $X_1$  = Augenzahl Würfel 1 und  $X_2$  = Augenzahl Würfel 2
- Sei  $X = X_1 + X_2$  die Summe der Augenzahlen

$$E(X_1) = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + \dots + 6 \cdot \frac{1}{6} = \frac{1+2+\dots+6}{6} = \frac{6 \cdot 7 / 2}{6} = 3.5$$

$$E(X_2) = 3.5 \quad \text{(der unumstößliche Satz von oben)}$$

$$E(X) = E(X_1 + X_2) = E(X_1) + E(X_2) = 3.5 + 3.5 = 7 \quad \blacksquare$$

## ■ Summe von Erwartungswerten, Korollar

- Bei einem Zufallsexperiment tritt das Ereignis  $E$  mit Wahrscheinlichkeit  $p$  auf. Sei  $X$  die Anzahl der Auftreten von  $E$  bei  $n$  Ausführungen dieses Experimentes, dann ist  $E(X) = n \cdot p$
- Beispiel:  $E(\text{Anzahl Sechser bei } 60 \text{ mal Würfeln}) = 10$
- Beweis: Definiere eine sogenannte **Indikatorvariable**

$I_j = 1$  wenn  $E$  eintritt bei Ausführung  $j$ , sonst  $I_j = 0$

$$\Pr(I_j = 1) = p \quad \dots \text{im Beispiel oben: } \frac{1}{6} = \text{eine 6 gerungen}$$

$$\Pr(I_j = 0) = 1-p \quad \dots \text{im Beispiel oben: } \frac{5}{6} = \text{keine 6 gew.}$$

$$E(I_j) = 1 \cdot \underbrace{\Pr(I_j = 1)}_{=p} + 0 \cdot \underbrace{\Pr(I_j = 0)}_{=1-p} = p \quad (= \Pr(I_j = 1))$$

$$X = \sum_{j=1}^n I_j \quad \dots \text{genau deswegen hat man } I_j \text{ so definiert}$$

$$E(X) = E\left(\sum_{j=1}^n I_j\right) = \sum_{j=1}^n E(I_j) = \sum_{j=1}^n p = n \cdot p \quad \blacksquare$$

# Literatur / Links

---

## ■ Universelles Hashing

- In Mehlhorn / Sanders:  
    4 Hash Tables and Associative Arrays
- In Wikipedia

[http://en.wikipedia.org/wiki/Universal\\_hashing](http://en.wikipedia.org/wiki/Universal_hashing)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 5b, Mittwoch, 24. Mai 2017  
(Universelles Hashing Teil 2, Perfektes Hashing)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Drumherum

- Mathe-Phobie

Ursachen und Heilung

## ■ Inhalt

- Universelle Klassen von Hash-Funktionen
- Worst-Case Komplexität
- Perfektes Hashing
- Histogramme

Zwei Negativ- und drei Positivbeispiele

Gestern: average-case

Wenn S vorher bekannt

Für das ÜB5

## ■ Kommentare aus den erfahrungen.txt

- Wie bei Spinnen: zu viele Beine aka Unbekannte
- 2 Punkte im Mathe-Abi ist doch besser als nichts
- Ich habe schon als kleines Kind nicht gerne geteilt
- Gründe 1: Mathe, Mathe 1, Mathe 2, schlechte Lehrer, ...
- Gründe 2: Vorurteile, mangelnde Motivation, ungenügende geistige Ausdauer (in dieser Reihenfolge)
- Nicht post-faktisch genug: man kann sich nicht rauslabern ... wenn es falsch ist, ist es falsch, kaum einer hat gerne Unrecht
- Angst vor dem Scheitern; es gibt nur richtig oder falsch, "ein bisschen richtig" wie bei anderen Fächern gibt es nicht

## ■ Aus der Neuropsychologie

- Es gibt zwei Arten von Zahlensystemen im Gehirn:
- **Object tracking system (OTS):** funktioniert über die "parallele" Wahrnehmung von verschiedenen Objekten
  - Grundlage unserer internen Darstellung der Zahlen
  - 0, 1, 2, 3 (bei Kindern) bzw. 0, 1, 2, 3, 4 (bei Erwachsenen)
- **Approximate number system (ANS):** Abschätzung und Vergleich der Größenordnungen von Gruppen
  - Erwachsene können damit "auf einen Blick" Unterschiede in Gruppengrößen von ca. 10 – 15% erkennen
  - Kleine Kinder erst Unterschiede von Faktor 2 oder größer ...
  - wird dann bis zum Erwachsenenalter immer trennschärfer

# Mathe-Phobie 0.75

---

## ■ Aus der Neuropsychologie, Fortsetzung

- OTS und ANS sind evolutionär beides sehr alte Systeme  
Gibt es auch schon bei Tieren, sogar schon bei **Insekten**
- Beim Menschen viel Forschung zum Zusammenhang zwischen Präzision des ANS und späteren mathematischen Fähigkeiten  
Libertus et al: "Preschool acuity of the approximate number system correlates with school math ability", Development 2011

Ergebnis der Studie: die Genauigkeit des ANS im Vorschulalter ist stark mit Matheleistungen im Grundschulalter korreliert

Das sollte aber keine Entschuldigung sein, siehe nächste Folie

## ■ Von der Mathe-Phobie zur Mathe-Philie

- Selbst bei ungünstigen ANS-Voraussetzungen, kann jeder etwas für eine gesunde Mathe-Philie tun

Es gibt auch Studien, die zeigen, dass im weiteren Verlauf der Entwicklung nicht die Grundintelligenz sondern die **Übung** der entscheidende Faktor ist

- Die "Phobie" kann tatsächlich real sein und aktiviert in der Amygdala dieselben Reflexe wie bei Schmerz und Angst

Das Gehirn ist dann mit der Verarbeitung der Stress-Situation beschäftigt anstatt mit dem mathematischen Problem

Wichtig für den Umgang damit: Umfeld zum Üben schaffen ohne kritischen Gegenüber und vor allem **ohne Zeitstress**

## ■ Vorberichtigung

- Im folgenden sind vorgegeben:
    - die Größe  $m$  der Hashtabelle
    - das Universum  $U$  und seine Größe  $|U|$
- Das weiß man ja in der Praxis beides, bevor man eine Hash-Funktion auswählt

# Klassen von Hashfunktionen 2/8

## ■ Negativbeispiel 1

- Alle  $h$  mit  $h(x) = a \cdot x \bmod m$ , mit  $a \in \{0, \dots, m-1\}$

Das sind  $m$  verschiedene Hashfunktionen

- Diese Klasse von Funktionen ist **nicht** universell
- Zum Beweis erst noch mal die Definition von universell:  
Für alle  $x \neq y$  ist  $|\{h \in H : h(x) = h(y)\}| \leq c \cdot |H| / m$
- Für die Nicht-Universalität reicht also ein Schlüsselpaar  $x, y$  mit  $x \neq y$  für dass das nicht gilt:

zum Beispiel:  $x = m$  und  $y = 2m$

$$\begin{aligned} \text{dann } \forall a \in \{0, \dots, m-1\} : \quad g(x) &= a \cdot x \bmod m \\ &= a \cdot m \bmod m = 0 \\ g(y) &= a \cdot 2m \bmod m = 0 \end{aligned}$$

$$\Rightarrow |\{g \in H : g(x) = g(y)\}| = |H| \quad \Rightarrow \text{nicht universell!}$$

alle  $g \in H$  sind ~~sicher~~  
für diese  $x$  und  $y$

*x und y kollidieren nur  
für einen Bruchteil  
der Hash-Funktionen*

# Klassen von Hashfunktionen 3/8

## ■ Negativbeispiel 2

- Die Menge  $H$  **aller** Funktionen von  $U \rightarrow \{0, \dots, m - 1\}$

Das sind ganz schön viele, nämlich:  $m \cdot m \cdot \dots \cdot m = m^{|\mathcal{U}|}$   
 $m$   $\underbrace{\quad \quad \quad}_{|\mathcal{U}| \text{ mal}}$

- Für eine zufällige Funktion  $h \in H$  ist dann für jedes  $x \in U$   $h(x)$  zufällig aus  $\{0, \dots, m - 1\}$

Eine zufällige Funktion  $h$  aus  $H$  wählen und eine Menge  $S$  damit abbilden ist also wie zufälliges Werfen

- Für zufälliges Werfen gilt:  $\Pr(h(x) = h(y)) = 1/m$
- Die Klasse ist also 1-universell

Besser geht es nicht, warum dann Negativbeispiel ?

## ■ Negativbeispiel 2, Fortsetzung

- Als Klasse von Hashfunktionen trotzdem ungeeignet
- Die Funktionen haben keine "kompakte" Form

So wie zum Beispiel  $a \cdot x \bmod m$

- Um eine Funktion  $h$  aus  $H$  zu repräsentieren, müsste man für jedes  $x \in U$  speichern, was  $h(x)$  ist

Das braucht  $\Theta(|U|)$  Platz

- Es würde auch reichen, den Wert von  $h(x)$  nur für  $x \in S$  zu speichern, aber dafür bräuchte man ... eine Map

Das kommt öfter vor: man trifft bei einem Lösungsversuch wieder auf das ursprüngliche Problem

# Klassen von Hashfunktionen 5/8

falls  $p > m$ , dann  $c$  sehr viele an 1.  
 falls  $m$  prim sogar  $c = 1$

genauer:  $c = \frac{\lceil p/m \rceil}{p/m}$

man braucht nicht  $p \geq |U|$  wie gestern behauptet

## ■ Positivbeispiel 1

- Sei  $p$  eine Primzahl mit  $p > m$  und  $U = \{0, \dots, p - 1\}$   
 $p$  ist Fix für die Klasse entscheidend  
 $\text{Kerbe} = \text{ggT}(m, p) = 1$
- Sei  $H$  die Menge aller Hash-Funktionen  $h$  mit

$$h(x) = (a \cdot x + b) \bmod p \bmod m \dots \text{wobei } a, b \in \{0, \dots, p - 1\}$$

- Diese Menge von Hashfunktionen ist  $\approx 1$ -universell

Beweis siehe Exercise 59 in Mehlhorn/Sanders ... nicht K 

- Intuition: man kann sich klarmachen, wie das  $\bmod p$  vor dem  $\bmod m$  die Probleme vom Negativbeispiel 1 verhindert

Beispiel:  $m = 10$ ,  $p = 101$ ,  $x = 80$ ,  $y = 30$ ,  $a = 2$ ,  $b = 17$

mit Negativ-Klasse 1:  $g_2(x) = a \cdot x \% m = 0$ ;  $g_2(y) = a \cdot y \% m = 0$   
 $= 2 \cdot 80 \% 10 = 0$        $= 2 \cdot 30 \% 10 = 0$

mit der Klasse oben:  $g_2(x) = (a \cdot x + b) \bmod p \bmod m = 76 \% 10 = 6$   
 $= 2 \cdot 80 + 17 \% 101 = 6$

$$g_2(y) = (a \cdot y + b) \bmod p \bmod m = 77 \% 10 = 7$$
 $= 2 \cdot 30 + 17 \% 101 = 7$

## ■ Positivbeispiel 2

- Die Menge aller  $h$  mit  $h(x) = a \bullet x \bmod m$

wobei  $a \in U$  und  $m$  eine **Primzahl** sein muss

Zusatzaufgabe: finde die nächste Primzahl  $\geq m$

- Die Operation  $\bullet$  ist dabei wie folgt definiert:

Schreibe  $a = \sum_{i=0..k-1} a_i \cdot m^i$ , wobei  $k = \lceil \log_m |U| \rceil$

Entsprechend  $x = \sum_{i=0..k-1} x_i \cdot m^i$

Dann  $a \bullet x := \sum_{i=0..k-1} a_i \cdot x_i$

- Intuitiv:  $a \bullet x$  ist quasi das "Skalarprodukt" der Darstellung von  $a$  und  $x$  zur Basis  $m$  ... Rechenbeispiel siehe nächste Folie

# Klassen von Hashfunktionen 7/8

## ■ Positivbeispiel 2, Fortsetzung

- Diese Klasse ist sogar **1-universell**

Beweis siehe Theorem 12 in Mehlhorn/Sanders ...

- Beispiel für eine Hashfunktion aus dieser Klasse mit:

$$m = 11, U = \{0, \dots, 11^3 - 1\}, a = 274, x = 47$$

$$a = 274 = 2 \cdot 11^2 + 2 \cdot 11 + 10 \cdot 1 = (2 \ 2 \ 10) \text{ zur Basis } 11$$

$$= 242 \qquad = 22 \qquad = 10$$

$$x = 47 = 0 \cdot 11^2 + 4 \cdot 11 + 3 \cdot 1 = (0 \ 4 \ 3) \text{ zur Basis } 11$$

$$= 0 \qquad = 44$$

$$a \odot x = (2 \ 2 \ 10) \odot (0 \ 4 \ 3) = 2 \cdot 0 + 2 \cdot 4 + 10 \cdot 3 = 38$$

$$a \odot x \bmod m = 38 \bmod 11 = 5 \quad \square$$

zur Basis 10:

$$a = 274 = 2 \cdot 10^2 + 7 \cdot 10 + 4 \cdot 1 = 10^2 + 10^1 + 10^0$$

# Klassen von Hashfunktionen 8/8

## ■ Positivbeispiel 3

- Die Menge  $H$  aller  $h$  mit  $h(x) = a \cdot x \bmod 2^k \text{ div } 2^{k-l}$

wobei  $U = \{0, \dots, 2^k - 1\}$ ,  $a \in U$  ungerade und  $m = 2^l$

$\subseteq 0 \dots 2^{k-1}$        $\subseteq 0 \dots 2^{k-1}$

sogar:  $(2^{k-1})^2 < 2^{2k} - 1$

- Das (normale) Produkt  $a \cdot x$  gibt eine Zahl aus  $0 \dots 2^{2k} - 1$

- In Binärdarstellung:  $a \cdot x$  lässt sich mit  $2k$  Bits darstellen, eine Position in der Hashtabelle mit  $l$  Bits

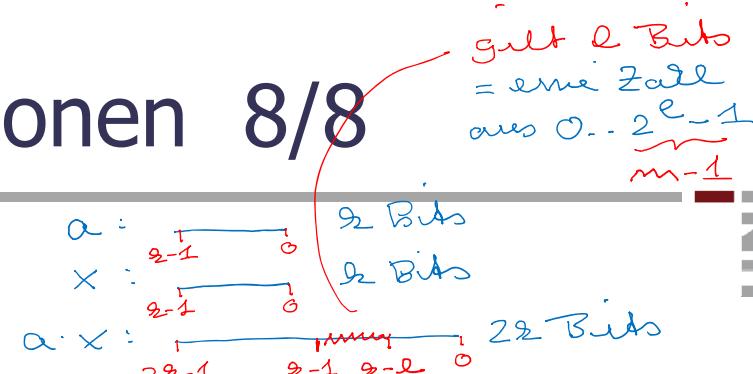
zum Beispiel:  
 $15 \Rightarrow 2 = 3$        $= \lfloor \frac{15}{2^2} \rfloor$

- $h(x)$  ist dann einfach der Wert der Bits  $k-l \dots k-1$  von  $a \cdot x$

Für das ÜB5: mit Bitschiebe-Operationen ausschneiden

- Diese Menge von Hashfunktionen ist **2-universell**

Siehe Exercise 62 in Mehlhorn / Sanders



## ■ Bisher: Average-Case Komplexität

- Wenn  $h$  aus einer universellen Klasse gewählt wird, dann ist für jeden Schlüssel  $x$  aus  $S$ :  $E(|S_x|) \leq 1 + c \cdot |S| / m$   
Insbesondere, wenn  $m = \Theta(|S|)$ :  $E(|S_x|) = O(1)$
- Sei  $S_{\max}$  der Platz in der Hashtabelle, auf den die meisten Schlüssel abgebildet werden, gilt dann auch  
 $E(|S_{\max}|) = O(1)$  ?
- Im Allgemeinen:  $E(\max\{X_1, \dots, X_n\}) \neq \max\{E(X_1), \dots, E(X_n)\}$   
E und Summe kann man vertauschen, E und max nicht

## ■ Zufälliges Werfen

- Nehmen wir an, wir werfen zufällig  $n$  Bälle in  $m$  Kisten  
Besser kann eine zufällige Hashfunktion nicht verteilen
- Sei  $S_i$  die Menge der Bälle in der  $i$ -ten Kiste und sei  $S_{\max}$  die Kiste mit den meisten Bällen
- Dann ist  $E(|S_i|) = n / m$
- Aber auch in dem Fall **nicht**  $E(|S_{\max}|) = n / m$

Dazu schreiben wir gerade ein kleines Programm ...

# Worst-Case Komplexität 3/3

## ■ Satz

- Nehmen wir an, wir werfen zufällig  $n$  Bälle in  $n$  Kisten und seien  $S_i$  und  $S_{\max}$  wie auf der Folie vorher

Dann ist  $|S_{\max}| = O(\log n / \log \log n)$  mit hoher W-keit

- Wenn man  $n$  Bälle in  $m$  Kisten wirft, mit  $n \geq m \cdot \log m$

Dann ist  $|S_{\max}| = \Theta(n / m)$  mit hoher W-keit

- Der Beweis würde hier zu weit führen ... nur kurz, wie der Term  $\log n / \log \log n$  überhaupt zustande kommt:

$$k^k = n \Rightarrow k \approx \log n / \log \log n \quad \text{Beweis: zu Hause}$$

zum Vergleic:  $2^{\varrho} = m \Rightarrow \varrho = \log_2 m$

## ■ Vorberichtigung

- Gegeben eine Schlüsselmenge  $S$
- Die Größe der Hashtabelle  $m$  sei  $\geq |S|$
- Dann gibt es (sogar viele) Hashfunktionen  $h$ , die  $S$  ohne Kollisionen (also perfekt) auf  $\{0, \dots, m - 1\}$  abbilden

Mathematisch gleichbedeutend mit:  $h$  ist **injektiv**

- Allerdings kann man diese Funktionen nicht unbedingt in wenig Platz speichern ... siehe Negativbeispiel 2

Man kann aber Funktionen finden, für die das geht

# Perfektes Hashing 2/4

---

## ■ Definition

- Gegeben eine Schlüsselmenge  $S$
- Die Größe der Hashtabelle  $m$  sei  $\geq |S|$
- Eine Hashfunktion  $h$  heißt **perfekt** für  $S$ , wenn gilt:
  - $h$  bildet  $S$  injektiv auf  $\{0, \dots, m - 1\}$  ab
  - $h$  kann in  $O(m)$  Platz gespeichert werden
  - $h(x)$  kann für alle  $x$  in  $O(1)$  ausgewertet werden

# Perfektes Hashing 3/4

## ■ Beispiel

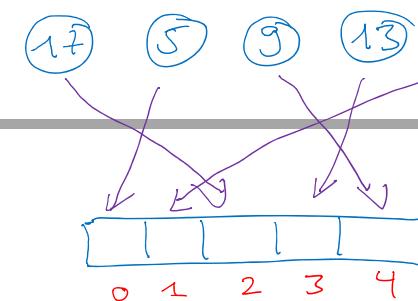
- Sei  $S = \{17, 5, 9, 13, 1\}$  und  $m = 5$
- Dann wäre folgende Hashfunktion perfekt:

$$h(x) = x \bmod 5$$

- Kann man immer so eine einfache Funktion finden?
- Antwort: so einfach nicht immer, aber einfach genug!

Achtung: das klappt wohlgemerkt nur, wenn die gesamte Menge  $S$  bekannt ist, bevor wir eine Map bauen

Das ist manchmal, aber oft auch nicht der Fall



# Perfektes Hashing 4/4

---

## ■ Satz

- Sei  $S$  eine beliebige Schlüsselmenge und  $m \geq 2 \cdot |S|$

Dann gibt es eine perfekte Hashfunktion  $S \rightarrow \{0, \dots, m - 1\}$

Und man kann sie in  $O(|S|)$  Zeit finden

- Der Beweis würde hier zu weit führen, siehe:

Storing a Sparse Table with  $O(1)$  Worst Case Access Time

Fredman, Komlós, Szemerédi

Journal of the ACM, Vol 31, No 3, 1984

# Histogramme 1/3

$$\begin{aligned} & 128 \cdot 127 \\ & = 16.384 - 128 \\ & = 16.256 \end{aligned}$$

## ■ Brauchen Sie für das Übungsblatt

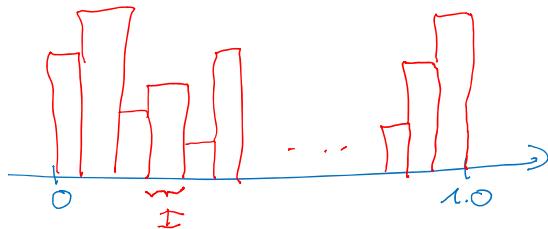
- Für vier von den fünf Klassen (Folie 6 – 12), sollen Sie die Kollisions-Wahrscheinlichkeiten von allen  $x \neq y$  berechnen
- Die Anzahl Paare  $x, y$  mit  $x \neq y$  ist  $u \cdot (u - 1)$  ... also groß!
- Die visualisiert man am besten mit einem Histogramm:

Werte  $\underbrace{x_1, x_2, x_3, x_4, \dots}_{\text{es sind } w \text{-größen}}$  Wertebereich hier  $[0,1]$

Unterteile Wertebereich in  $n$  disjunkte Teil-Intervalle

In unserem Fall hier kann man die gleich groß wählen

Zähle für jedes Teil-Intervall  $I$  die Anzahl aller  $x_i \in I$



# Histogramme 2/3

## ■ Erstellen der Daten

- Zeilenbasiert in eine Datei ausgeben

0.0 120

0.1 47

0.2 88

...

oder "Bin"

Erste Spalte: "Bucket" (x-Achse)

Zweite Spalte: Anzahl (y-Achse)

# Histogramme 3/3

## ■ Wie malt man so ein Histogramm?

- Zum Beispiel mit **gnuplot**

|                                       |                 |
|---------------------------------------|-----------------|
| set term pdf                          | Ausgabe als PDF |
| set output "data.pdf"                 | Ausgabedatei    |
| set style fill solid 0.4              | Gefüllte Boxen  |
| plot [] [0:200] "data.txt" with boxes | Malen           |
| plot ... lt rgb "orange"              | Andere Farbe    |

- Das geht aber auch mit vielen anderen Programmen, z.B.

S ... oder die open-source Variante R  
Matlab ... oder die open-source Variante Octave  
Mathematica  
Excel

# Literatur / Links

---

- Universelle Klassen von Hashfunktionen

- In Mehlhorn / Sanders:  
4 Hash Tables and Associative Arrays

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 6a, Dienstag, 30. Mai 2017  
(Dynamische Felder, Teil 1)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- Erfahrungen mit ÜB5

Universelles Hashing

## ■ Inhalt

- Felder fester Größe
- Dynamische Felder

in Java, C++, Python

Implementierung,  
Laufzeitmessung,  
Laufzeitanalyse

ÜB6: Implementierung verallgemeinern (auf dem Papier) +  
einen "Kilometerzähler" implementieren und analysieren  
(mit amortisierter Analyse → siehe Vorlesung 6b morgen)

## ■ Zusammenfassung / Auszüge

- Hat eine Weile gebraucht, alles zu verstehen
- Relativ zeitaufwändiges Blatt für viele
- Unit Tests waren nicht so einfach
- Histogramm interpretieren: gefühlt um drei Ecken denken
- Aufgabe 3: ohne Histogramme schwer dazu etwas zu sagen
- Durch Aufgabe 3 klarer, wie Universalität funktioniert
- Note to Self: Die Übungsblätter gehen bedeutend leichter, wenn man auf die Professorin hört
- Viel (sehr gutes) Feedback zum Feedback der Tutoren

## ■ Musterlösung

- Wie immer gibt es einen Musterlösung
  - Für eine Klasse wollen wir das gerade mal live codeN
- Um zu demonstrieren, dass es wirklich nicht viel Code ist und der Code an sich auch nicht kompliziert ist

## ■ Plagiate

- Es gibt immer noch einige Plagiate pro Übungsblatt
- Einige (wenige) treiben erheblichen Aufwand, um das Plagiat zu vertuschen
- Wir merken es aber trotzdem
- **Verwenden Sie Ihre Energie doch lieber darauf, das Übungsblatt zu machen !**

## ■ Eigenschaften

- Die Größe des Feldes wird bei der Deklaration festgelegt

**Vorteil:** effizient umzusetzen, weil nur einmal eine feste Menge Speicher alloziert werden muss

**Nachteil:** bei manchen Anwendungen weiß man vorher nicht, ein wie großes Feld man benötigt

# Felder fester Größe 2/5

## ■ Felder fester Größe in Java

|                                                                                               |                                                                |
|-----------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| int[] numbers = new int[100];<br>System.out.println(numbers[12]);                             | Feld von 100 ints<br>Druckt "0"                                |
| String[] strings = new String[10];<br>System.out.println(strings[7]);<br>strings[8] = "doof"; | Feld von 10 strings<br>Druckt "null"<br>Setzt das 9-te Element |

In Java werden die Elemente des Feldes bei der Deklaration grundsätzlich **immer** initialisiert

Bei nativen Typen wie int mit 0, bei Objekten mit null

# Felder fester Größe 3/5

## ■ Felder fester Größe in C++

```
int[] numbers = new int[100];
cout << numbers[12] << endl;
```

Feld von 100 ints  
Druckt irgendwas

```
string[] strings = new string[10];
cout << strings[7] << endl;
strings[8] = "doof";
```

Feld von 10 strings  
Druckt leeren string  
Setzt das 9-te Element

In C++ werden die Elemente bei nativen Datentypen wie `int` grundsätzlich **nicht** initialisiert (aus Effizienzgründen)

Bei Objekten wird dagegen grundsätzlich der Default-Konstruktor der Klasse aufgerufen

# Felder fester Größe 4/5

---

## ■ Felder fester Größe in Python

- In **Python** gibt es keine Felder fester Größe

Der Schwerpunkt bei Python liegt auf möglichst einfacher und schneller Entwicklung, nicht auf effizientem Code

- Die "Listen" von Python sind Felder variabler Größe, wobei die Elemente Referenzen auf (beliebige) Objekte sind

In **numpy** gibt es auch effizientere Felder (`numpy.array`), wo die Elemente Zahlen und keine Referenzen sind

Aber auch ein `numpy.array` hat schon variable Größe

# Felder fester Größe 5/5

---

## ■ Begriffsverwirrung

- Felder fester Größe werden auch manchmal **statische Felder** genannt, weil sich ihre Größe nicht ändert
- Das hat aber nichts mit statischer vs. dynamischer Allokation oder dem keyword **static** zu tun:

Bei dem keyword **static** geht es darum, ob Speicherplatz schon vom Compiler (statisch) oder erst zur Laufzeit (dynamisch) zugewiesen wird

- Statisch zugewiesener Platz hat eine feste Größe
- Aber ein Feld fester Größe kann auch dynamisch zugewiesen werden:

`int[] array = new int[100]; // New array of 100 ints.`

## ■ Eigenschaften

- Dynamische Felder können im Laufe ihres Lebens beliebig vergrößert und verkleinert werden

Alternative Bezeichnung daher: Felder variabler Größe

- Das (nicht-triviale) Speichermanagement ist dabei vor der Benutz-Person versteckt
- Zu Beginn kann das Feld entweder leer sein oder schon eine gewisse Größe haben

Wenn man weiß, dass man eine gewisse Größe sowieso braucht, spart man sich dann am Anfang das Management

## ■ Dynamische Felder in Java

- In Java nimmt man dafür `java.util.ArrayList`

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("doof");
strings.add("doofer");
strings.add("am doofsten");
System.out.println(strings.get(0));
strings.remove(string.length() - 1);
```

Druckt "doof"  
Lösche letztes Element

- **Wichtig:** `ArrayList` funktioniert **nicht** mit nativen Typen, z.B. `int`, man muss dann `Integer` nehmen

Das ist für große Datenmengen sehr ineffizient ... wenn die Laufzeit zählt, sollte man dann lieber ein eigenes `ArrayInt` implementieren, das intern mit nativen Feldern realisiert ist

oder Bibliotheken  
benutzen wie GUAVA  
oder TROVE

## ■ Dynamische Felder in C++

- In C++ nimmt man dafür std::vector aus der STL

```
std::vector<std::string> strings;
strings.push_back("doof");
strings.push_back("doofer");
strings.push_back("am doofsten");
cout << strings[0] << endl; Druckt "doof"
strings.pop_back(); Lösche letztes Element
```

- **Gut:** vector funktioniert mit allen Typen (nativ und Klassen) und ist genauso effizient wie ein Feld fester Größe

Grund: **templates** ... siehe Programmieren in C++, VL 8

# Dynamische Felder 4/9

## ■ Dynamische Felder in Python

- In Python hat man "Listen" und die sind immer dynamisch

```
strings = [];
strings.append("doof");
strings.append("doofer");
strings.append("am doofsten");
print(strings[0]); Druckt "doof"
strings.pop(); Lösche letztes Element
```

- Für native Typen gibt es in Python auch die Klasse array

Die ist effizienter als eine Liste, aber auch dynamisch in dem Sinne, dass sie Ihre Größe beliebig ändern kann ... wie gesagt, ein Feld fester Größe gibt es in Python nicht

## ■ Realisierung dynamischer Felder (intuitiv)

- Intern ein **fixed-size array (FSA)** von ausreichender Größe
- Wenn Elemente dazu kommen oder entfernt werden:

**Schauen:** passt die Größe des internen FSA noch?

**Falls nein:** erzeuge ein neues FSA passender Größe und kopiere die Elemente vom alten in das neue FSA

Diesen Vorgang nennt man **Reallokation**

- Das implementieren wir jetzt zusammen

Erst mal `pushBack`

neues Element anhängen

Dann auch `popBack`

letztes Element entfernen

## ■ Vergrößerungsstrategie 1

- Wir vergrößern das Feld nach jedem `push_back` ... und machen es dabei aber immer nur genau **um eins** größer

**Beobachtung:** akkumulierte Laufzeit sieht quadratisch aus

## ■ Analyse

- Sei  $T_i$  die Laufzeit für das  $i$ -te `push_back`
- Dann ist  $T_i \geq A \cdot i$  für irgendeine Konstante  $A$

Bei einer Reallokation müssen alle Elemente kopiert werden

$$\sum_{i=1}^m T_i \geq \sum_{i=1}^m A \cdot i = A \cdot \sum_{i=1}^m i = A \cdot \frac{1}{2} m(m+1) = \Theta(m^2)$$

*1/C besser, aber immer noch*

*QUADRATISCH*

## ■ Vergrößerungsstrategie 2

- Wie vorher, aber jetzt bei jedem Vergrößern **um C größer**, für ein festes  $C$ , zum Beispiel  $C = 100$  oder  $C = 1000$

**Beobachtung:** akkumulierte Laufzeit immer noch quadratisch

## ■ Analyse

- Sei wieder  $T_i$  die Laufzeit für das  $i$ -te `push_back`
- Dann sind die meisten  $T_i$  jetzt  $O(1)$
- Aber für  $i = C, 2C, 3C, \dots$  ist nach wie vor  $T_i \geq A \cdot i$

*$C = 1000 : 1000, 2000, 3000, \dots$*

*Anmerkung:  
n Vielfache von C*

$$\sum_{i=1}^m T_i \geq \sum_{j=1}^{m/c} T_{cj} \geq \sum_{j=1}^{m/c} A \cdot C \cdot j = A \cdot C \cdot \sum_{j=1}^{m/c} j$$

$$= A \cdot C \cdot \frac{1}{2} m/c \cdot (m/c + 1) \geq A \cdot C \cdot \frac{1}{2} \cdot m^2/c^2 = \frac{1}{2} A \cdot \frac{1}{C} \cdot m^2$$

*$\geq m/c$*

## ■ Vergrößerungsstrategie 3

- Wie vorher, aber jetzt machen wir bei jedem Vergrößern das Feld genau **doppelt so groß** wie vorher

Beobachtung: jetzt sieht die Laufzeitkurve linear aus

## ■ Analyse

- Jetzt Reallokationen nur noch bei  $i = 1, 2, 4, 8, 16, \dots$

Bei den Reallokationen  $T_i \leq A \cdot i$ , sonst  $T_i \leq A$

$$\sum_{i=1}^m T_i \leq \sum_{i=1}^m A + A(1 + 2 + 4 + \dots + 2^{2^{\frac{m}{2}}})$$

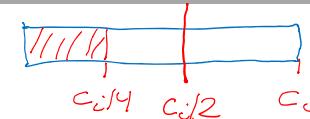
$\underbrace{2^{2^{\frac{m}{2}}+1}-1}_{2m}$

$$\begin{aligned} & \leq A \cdot m + A \cdot (2m - 1) \\ & \leq 3 \cdot A \cdot m = O(m) \end{aligned}$$

$1 + 2 + 4 + 8 = 15$   
 $= 16 - 1$   
 $= 2^4 - 1$

## ■ Entfernen von Elementen

- Analog zum Vergrößern, könnten wir das Feld auf die Hälfte verkleinern wenn es nur noch halb voll ist



**Aber:** wenn man danach zwei pushBack macht,  
muss man das Feld gleich wieder vergrößern

**Und:** wenn man danach zwei popBack macht,  
muss man das Feld gleich wieder verkleinern

- Deswegen machen wir es (erst mal) so:

Wenn Feld **ganz voll** → Größe **verdoppeln**

Wenn Feld **viertel voll** → Größe **halbieren**

# Literatur / Links

---

- Mehlhorn / Sanders
  - Kapitel 3: Representing Sequences by Arrays ...
- Plotly
  - <https://plot.ly>
- Doof
  - <http://de.wiktionary.org/wiki/doof>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 6b, Mittwoch, 31. Mai 2017  
(Dynamische Felder, Teil 2: amortisierte Analyse)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Drumherum

- Das offensichtlich Richtige      Warum so schwer?
- Nächste Woche ist Pfingsten      Keine Vorlesungen

## ■ Dynamische Felder, Teil 2

- Vergrößerungsstrategien      Fortsetzung von VL6a
- Laufzeitanalyse      amortisierte Analyse
- Potenzialfunktion      Konzept dahinter
- ÜB6: kurze Besprechung dazu und Gelegenheit für Fragen

## ■ Warum so schwer? ... Ihre Kommentare dazu

- Frage sehr allgemein ... dafür viele individuelle Antworten!
- Kenne das nur, wenn das offensichtliche ein Übungsblatt ist
- Das ist nichts für Bad Boys... und Ladys lieben Bad Boys
- Mensch: 99% Körperintelligenz, 0.9% Kulturintelligenz, 0.1% Individualität ... Gefühl fehlender Kontrolle kein Wunder
- Die Wahrheit ist, wir haben nicht die Kontrolle
- Satan ist überall! Ihr habt den Teufel zum Vater! (vgl. Johannes 8, 44) Gelobt seist du Christi!
- No man chooses evil because it is evil; he only mistakes it for happiness, the good he seeks.

## ■ Präfrontaler Kortex

- Linke Seite: leitet **Aktionen** ein
- Rechte Seite: **widersteht** Versuchungen
- Mitte: **wägt ab** zwischen Alternativen
- Folgen bei Verletzung oder Zerstörung dieser Areale:
  - Keine Langzeitplanung mehr möglich
  - Entscheidungsunfähigkeit
  - Krankhaftes Beharren und Inflexibilität
  - Emotionale Verflachung, Enthemmung, Euphorie ohne Grund

## ■ Präfrontaler Kortex, Evolution

- Die Kortikalisierung des Gehirns ist wohl der wesentlichste Faktor für die besondere Intelligenz des Homo Sapiens

**Viel wichtiger als die Zunahme des Gehirnvolumens**

- Kam in der Evolution als letztes dazu
- Entsprechend wird in existenziellen **Stress-Situationen** und bei **Energiemangel** der präfrontale Kortex auch als erstes "abgeschaltet" bzw. auf "Sparflamme" gestellt

Dann übernehmen entwicklungsgeschichtlich ältere Teile des Gehirns die Kontrolle ... das will man (meistens) nicht

## ■ Präfrontaler Kortex, Unterstützung

- Genügend **Schlaf** und genügend Nahrung

Das ist quasi die Grundversorgung, wenn die fehlt ist das so wie wenn der Strom abgeschaltet wurde

Gehirn verbraucht sehr viel Energie (in der Form von Glukose)

- Vermeidung von negativem (existenziellen) Stress  
Sonst geht die ganze Energie in die Stress-Reaktion
- Willenskraft verhält sich in vielerlei Hinsicht wie Muskelkraft

Es ist eine begrenzte Ressource, die irgendwann aufgebraucht ist und dann Zeit zur Regeneration braucht

**Ganz wichtig: man kann (und sollte) sie trainieren**

## ■ Präfrontaler Kortex, Training

- Bewegung ... das erfordert Planung und Willenskraft
  - Trainiert Körper **und** Geist → zwei Fliegen mit einer Klappe
- Sich etwas vornehmen und es dann auch umsetzen
  - Kann etwas ganz einfaches sein, wie z.B. für X Wochen immer fünf Minuten vorher zu einem Termin kommen
  - Sollte nicht-trivial sein, aber auch keine Überforderung
  - Wenn das Ziel erreicht ist, oder es langweilig wird, eine andere Aufgabe angehen (aber eine nach der anderen)
- Und wie gesagt: genügend **schlafen** und vernünftig essen
  - Sonst geht schon rein energietechnisch gar nichts

## ■ Vorberichtigungen

- Unsere bisherigen Laufzeitanalysen haben angenommen, dass wir nur Elemente hinzufügen (mit `pushBack`)
- Dann können wir leicht ausrechnen, wann es zu einer Reallokation kommt

Mit der Verdoppelungstrategie bei: 1, 2, 4, 8, ...

- Im Folgenden wollen wir unsere Analyse verallgemeinern auf beliebige Abfolgen von `pushBack` und `popBack`

Dann können wir nicht mehr so leicht vorhersagen, wann realloziert werden muss

## ■ Notation

- Gegeben  $n$  Operationen  $O_1, \dots, O_n$   
Eine beliebige Abfolge von **pushBack** und **popBack**
- Sei  $s_i$  die Anzahl Elemente **nach** Operation  $O_i$  ( $s_0 := 0$ )
- Sei  $c_i$  die Größe des Feldes **nach** Operation  $O_i$  ( $c_0 := 0$ )  
Es gilt immer  $c_i \geq s_i$  (Feld muss immer "groß genug" sein)
- Sei wie in VL6a gestern  $T_i$  die Zeit für Operation  $O_i$ 
  - Falls Reallokation nicht nötig:  $T_i \leq A$
  - Falls Reallokation nötig:  $T_i \leq A + B \cdot s_i$

für irgendwelche Konstanten  $A$  und  $B$  unabhängig von  $n$

# Laufzeitanalyse 3/7

## ■ Wir analysieren folgende Variante

- Falls Operation  $O_i$  ein pushBack ist:
  - Reallokation genau dann, wenn  $s_{i-1} = c_{i-1}$
  - Vergrößerung so, dass danach  $c_i = 2 \cdot s_i$
- Falls Operation  $O_i$  ein popBack ist:
  - Reallokation genau dann, wenn  $4 \cdot s_{i-1} \leq c_{i-1}$
  - Verkleinerung so, dass danach  $c_i = 2 \cdot s_i$
- In beiden Fällen ist also direkt nach der Reallokation
$$c_i = 2 \cdot s_i \dots \text{also das interne Feld exakt } \mathbf{doppelt} \text{ so groß}$$
- Das Feld ist immer zu mindestens **einem Viertel** voll

zum Beispiel :

$$s_{i-1} = 17 = c_{i-1} \\ (\text{vor der Operation})$$

$$s_i = 18 ; c_i = 36 \\ (\text{nach der Operation})$$

$$s_{i-1} = 12 ; c_{i-1} = 50 \\ (\text{vor der Operation})$$

$$s_i = 11 ; c_i = 22 \\ (\text{nach der Operation})$$

nicht genauer:  
 $x+1$  Operationen à 2000 €  
 $\Rightarrow \frac{2000 \text{ €}}{x+1} < 2 \text{ € / OP.}$

## ■ Beweisidee

- Nach einer teuren Op. kommen viele billige Operationen  
**Teuer sind nur Operationen, wo realloziert werden muss**
  - **Genauer:** wenn nach einer Operation die  $X$  gekostet hat  
 $\frac{1000}{X}$  Operationen kommen die alle nur  $1 \text{ €}$  kosten, sind die Gesamtkosten bei  $n$  Operationen höchstens  $2 \cdot n$   
 $\frac{2000 \text{ €}}{2 \text{ € / OP.}}$
  - **Allgemeiner:** wenn nach einer Operation mit Kosten  $c_1 \cdot X$   
 $\frac{1000}{X}$  Operationen kommen mit Kosten  $c_2$ , dann sind die Gesamtkosten bei  $n$  Operationen höchstens  $(c_1 + c_2) \cdot n$   
 $\frac{7000 \text{ €}}{7 \text{ € / OP.}}$
- Man kann die Kosten der teuren Operationen quasi auf die billigen Operationen "umlegen" (amortisieren)
- Analogie: neue Heizung (teuer) spart dann monatlich Öl

## ■ Formaler Beweis

- **Lemma:** Wenn bei  $O_i$  eine Reallokation stattfindet, und die nächste Reallokation danach bei  $O_j$ , dann  $j - i \geq s_i / 2$

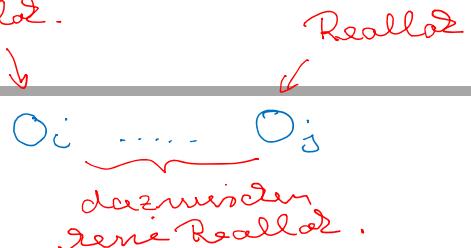
Nächste Reallokation frühestens nach  $s_i/2$  Operationen

- **Korollar:** Seien die Kosten einer Operation  $O_i$  ohne Reallokation  $T_i \leq A$  und mit Reallokation  $T_i \leq A + B \cdot s_i$

Dann ist  $T_1 + T_2 + \dots + T_n \leq (A + 3B) \cdot n$

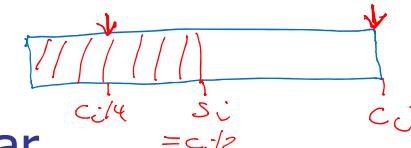
Eine Operation kostet also im Durchschnitt  $\leq A + 3B = O(1)$

- Wir beweisen auf den nächsten beiden Folien erst das Lemma und dann, weil es so schön ist, auch das Korollar



## ■ Beweis des Lemmas

- Zu zeigen: Wenn bei  $O_i$  eine Reallokation stattfindet, und die nächste Reallokation danach bei  $O_j$ , dann  $j - i \geq s_i / 2$
- Nach  $O_i$  ist auf jeden Fall  $c_i = 2 \cdot s_i$   
Egal ob es ein pushBack oder ein popBack war
- Nächste Vergrößerung wenn  $s_j = c_i = 2 \cdot s_i$   
Also nach frühestens  $s_i \geq s_i / 2$  Operationen
- Nächste Verkleinerung wenn  $s_j = \lfloor c_i / 4 \rfloor = \lfloor s_i / 2 \rfloor$   
Also nach frühestens  $s_i - \lfloor s_i / 2 \rfloor \geq s_i / 2$  Operationen
- Das Lemma gilt also in jedem Fall



# Laufzeitanalyse 7/7

## ■ Beweis des Korollars

- Seien  $O_{i_1}, O_{i_2}, \dots, O_{i_k}$  die Op.n bei denen realloziert in chronologischer Reihenfolge, also:  $i_1 < i_2 < \dots < i_k$
- Damit  $T_1 + T_2 + \dots + T_n \leq A \cdot n + B \cdot (s_{i_1} + s_{i_2} + \dots + s_{i_k})$
- Zu zeigen: dann  $T_1 + T_2 + \dots + T_n \leq (A + 3B) \cdot n$

$$\text{LEMMA} \Rightarrow \underbrace{i_2 - i_1 \geq s_{i_2}/2}_{\Leftrightarrow s_{i_1} \leq 2 \cdot (i_2 - i_1)}, \underbrace{i_3 - i_2 \geq s_{i_3}/2}_{\Leftrightarrow s_{i_2} \leq 2 \cdot (i_3 - i_2)}, \dots$$

$$\begin{aligned} \text{Dann: } s_{i_1} + \dots + s_{i_k} &\leq 2 \cdot (i_2 - i_1) + 2 \cdot (i_3 - i_2) + \dots + 2 \cdot (i_k - i_{k-1}) \\ &\quad + s_{i_k} \\ &= 2 \cdot \underbrace{(i_2 - i_1 + i_3 - i_2 + \dots + i_k - i_{k-1})}_{\text{sog. Teleskopsumme}} + s_{i_k} \\ &= i_k - i_1 \end{aligned}$$

$$\text{Also: } \sum_{j=1}^m T_j \leq A \cdot m + B \cdot (2(i_k - i_1) + s_{i_k}) \leq (A + 3B) \cdot m$$

$\leq m$   
 weil = alle  
 $i_1, \dots, i_k \leq m$   
 weil =  $m$  OP.

$\leq m$   
 weil = worst  
 case,  $m$  mal  
 passiert



## ■ Variante des Beweises

- Der Beweis auf den vorherigen Folien hat die Kosten für eine Folge von Operationen quasi "zu Fuß" analysiert
- Man kann solche Beweise auch mit Hilfe einer sogenannten **Potenzialfunktion** führen
- Intuitiv misst die Potenzialfunktion, wie lange es bis zur nächsten teuren Operation dauert

Teure Operationen (wie unser `realloc`) sollen die Potenzialfunktion erhöhen, und zwar um mindestens  $X$ , wenn die Kosten der Operation  $\Theta(X)$  waren

Billige Operationen sollen die Potenzialfunktion nur geringfügig (ideal: um eine Konstante) erniedrigen

# Beweis mit Potenzialfunktion 2/5

## ■ Potenzialfunktionen, Mastertheorem

- Gegeben eine Folge von  $n$  Operationen  $O_1, \dots, O_n$  auf einer beliebigen Datenstruktur
- Sei  $\Phi$  eine Potenzialfunktion, wobei  $\Phi_i$  = der Wert der Potenzialfunktion nach  $O_i$  und  $\Phi_0$  = Wert am Anfang  $\geq 0$
- Sei  $T_i$  die Laufzeit für  $O_i$  mit  $T_i \leq A + B \cdot (\Phi_i - \Phi_{i-1})$
- Dann ist die Gesamtlaufzeit  $\sum T_i = O(n + \Phi_n)$
- **Beweis:**

$$\begin{aligned} \sum_{i=1}^m T_i &\leq A \cdot m + B \cdot (\underbrace{\Phi_1 - \Phi_0 + \Phi_2 - \Phi_1 + \Phi_3 - \Phi_2 + \dots + \Phi_m - \Phi_{m-1}}_{\text{wie vorher eine Teleskopsumme}}) \\ &= A \cdot m + B \cdot \underbrace{(\Phi_m - \Phi_0)}_{\geq 0} \leq A \cdot m + B \cdot \Phi_m \\ &= O(m + \Phi_m) \end{aligned}$$

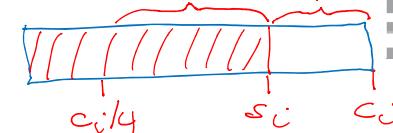
## ■ Anwendung des Satzes für dynamische Felder

- Wie vorher  $s_i$  = Anz. Elemente und  $c_i$  = Kapazität **nach**  $O_i$
- Definiere  $\Phi_i := \min \{ c_i - s_i, s_i - c_i / 4 \}$  ...  $\Phi_0 := 0$   
Minimum Anzahl freier Plätze nach links und nach rechts
- Dann gilt  $T_i \leq A' + B' \cdot (\Phi_i - \Phi_{i-1})$  ... **Beweis nächste Folie**
- Und damit gemäß Satz  $\sum T_i = O(n + \underbrace{\Phi_n}_{\leq n}) = O(n)$

# Beweis mit Potenzialfunktion

4/5

$$\Phi_i = \min \left\{ \underbrace{c_i - s_i}_{\text{Platz "nach rechts"}, \text{ Platz 2}}, \underbrace{s_i - c_i/4}_{\text{Platz "nach links", Platz 1}} \right\}$$



UNI  
FREIBURG

- Beweis, dass  $T_i \leq A' + B' \cdot (\Phi_i - \Phi_{i-1})$

- Wir unterscheiden zwei Fälle:

**Fall 1:**  $O_i$  ist "billig" (ohne Reallokation) ...  $T_i \leq A$

Potenzial ändert sich: um höchstens 1

$$\text{Damit } \Phi_i - \Phi_{i-1} \geq -1 \text{ und also } T_i \leq A \leq A + 1 - \frac{1}{\Phi_i - \Phi_{i-1}} \leq A + 1 + \Phi_i - \Phi_{i-1}$$

**Fall 2:**  $O_i$  ist "teuer" (mit Reallokation) ...  $T_i \leq A + B \cdot s_i$

Potenzial vor Reallokation: ○

siehe Folie X < 18

Potenzial nach Reallokation:  $\min \{ s_i, s_i/2 \} \geq s_i/2$

$$\text{Damit } \Phi_i - \Phi_{i-1} \geq s_i / 2 \Leftrightarrow s_i \leq 2(\Phi_i - \Phi_{i-1})$$

$$\text{Also } T_i \leq A + B \cdot s_i \leq A + 2B \cdot (\Phi_i - \Phi_{i-1}) \leq A' \quad := B'$$

## ■ Vergleich der beiden Beweise

- Für die dynamischen Felder war der "zu Fuß" Beweis nicht wirklich einfacher, aber direkter
- Der Beweis über die Potenzialmethode war intuitiver, weil das Potenzial eine intuitive Bedeutung hat:  
**Und zwar: Mindestdauer bis zur nächsten Reallokation**
- Spätere Vorlesung: amortisierte Analyse, wo der Beweis über eine Potenzialfunktion einfacher **und** intuitiver ist  
**ÜB6, Aufgabe 3: dort ebenfalls !**

# Literatur / Links

---

## ■ Dynamische Felder: Laufzeitanalyse

- In Mehlhorn/Sanders:  
[3.2 Unbounded Arrays](#)
- In Wikipedia  
[http://en.wikipedia.org/wiki/Dynamic\\_array](http://en.wikipedia.org/wiki/Dynamic_array)
- Potenzial vs. Potential  
<http://www.duden.de/rechtschreibung/potenzial>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 7a, Dienstag, 13. Juni 2017  
(Verkettete Listen)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- ## – Erfahrungen ÜB6      Dynamische Felder

## ■ Inhalt

- ## – Verkettete Listen Prinzip + Implementierung

- Listen vs. Felder Was ist wann besser?

- Cache-Effizienz morgen in Vorlesung 7b

- ÜB7: die Klasse LinkedList aus der Vorlesung um ein paar Operationen erweitern + deren Cache-Effizienz bestimmen

## ■ Zusammenfassung / Auszüge

- Aufgabe 2 (Implementierung "Kilometerzähler") war für die meisten gut machbar

Fehler in TIP: Uhrzeit ist kein Zähler im Sinne des ÜB6, wurde im Forum besprochen aber in TIP nicht korrigiert

- Theorieaufgaben hilfreich für das Verständnis, allerdings haben es einige nicht gut verstanden bzw. aufgegeben

Siehe Lösungsskizzen auf der nächsten Folie

Hier bin ich ehrlich gesagt etwas ratlos: das Thema wurde in der VL6b sehr ausführlich und mit viel Intuition erklärt

- Machine Learning ist viel spannender als so Basis-Zeug

Aber keine Chance das zu verstehen ohne Basis-Zeug

# Erfahrungen mit dem ÜB6 2/3

Runden braucht man, weil nicht unbedingt  $A, C \in \mathbb{N}$

UNI  
FREIBURG

## ■ Lösungsidee Aufgabe 1 (Feld immer $\geq f \cdot s$ voll)

- Bei pushBack, wenn Feld ganz voll, dann  $c_i = \lfloor A \cdot s_i \rfloor$
- Bei popBack, wenn  $s_i \leq c_{i-1} / B$ , dann  $c_i = \lfloor C \cdot s_i \rfloor$
- Bedingungen:  $A, B, C > 1$  (klar) und  $B > C$  (damit keine Vergrößerung kurz nach Verkleinerung oder andersrum)
- In der Analyse von Vorlesung 6b war  $A = C = 2$  und  $B = 4$

Damit ist das Feld immer mindestens  $1 / B = 1 / 4$  gefüllt

- Für beliebiges  $f$  mit  $0 < f < 1$ :

$B = 1 / f$  ... dann ist immer  $s_i \geq f \cdot c_i$

oder irgendein anderes  $C$   
mit  $f < C < B$

$C = (B + 1) / 2$  ... dann  $1 < C < B$

$A = C$  ... gibt keinen guten Grund für  $A \neq C$

$$\text{z.B. } g = 80\% = \frac{4}{5}$$

$$B = \frac{5}{4} = 1.25$$

$$C = \frac{\frac{5}{4} + 1}{2} = 1.125$$

$$A = 1.125$$

$$\text{Bedingung für Master-Theorem: } T_i \leq A + B \cdot (\Phi_i - \Phi_{i-1})$$

# Erfahrungen mit dem ÜB6 3/3

$\varrho_k = \text{Anzahl Ziffern insgesamt}$

und für alle  
Operationen gleich

zumindest für  
jede OP  
eindess sein

$\varrho'_k = 4$

## ■ Lösungsidee Aufgabe 3 (Potenzialfunktion Zähler)

$\varrho'_k \leq \varrho_k$

- Sei  $k'$  die Anzahl Ziffern, die nach einem "increment" auf null zurückspringen ... zum Beispiel  $009999 \rightarrow 01\mathbf{0000}$
- Die Laufzeit einer increment Operation ist dann  $O(k' + 1)$   
Plus 1, weil es für  $k' = 0$  auch dauert
- Das legt folgende Definition der Potenzialfunktion nahe
  - Potenzial am Anfang  $\Phi_0 = m$ , wobei  $m = \# \text{Stellen}$
  - Potenzial am Ende  $\Phi_m \leq \varrho_k \Rightarrow \text{Laufzeit } O(m + \varrho_k)$
- Mit  $k$  wie oben definiert, erhöht sich dann nach einer Operation das Potenzial um mindestens  $k' - 1$   
Minus 1, weil man vorne evtl. eine Null verliert
- Der Rest folgt dann aus dem Mastertheorem

↑  
unabhängig  
von  $n$ , also  
der Konstante.

# Verkettete Listen 1/8

---

## ■ Grundprinzip

- Wir betrachten hier **doppelt** verkettete Listen
- Jedes Element kennt seinen Vorgänger (`previous`) und seinen Nachfolger (`next`)
- Außerdem kennt man den Anfang (`first`) und das Ende (`last`) der Liste
- Das ermöglicht uns Einfügen und Löschen **an beliebiger Stelle** in  $O(1)$  Zeit

Das können Felder nicht, siehe spätere Folie 12

Beispiele von solchen Listen auf den nächsten Folien ...

# Verkettete Listen 2/8

PREV  $\stackrel{!}{=}$  previous  
NEXT  $\stackrel{!}{=}$  next

## ■ Einfügen (insert) eines neuen Elementes

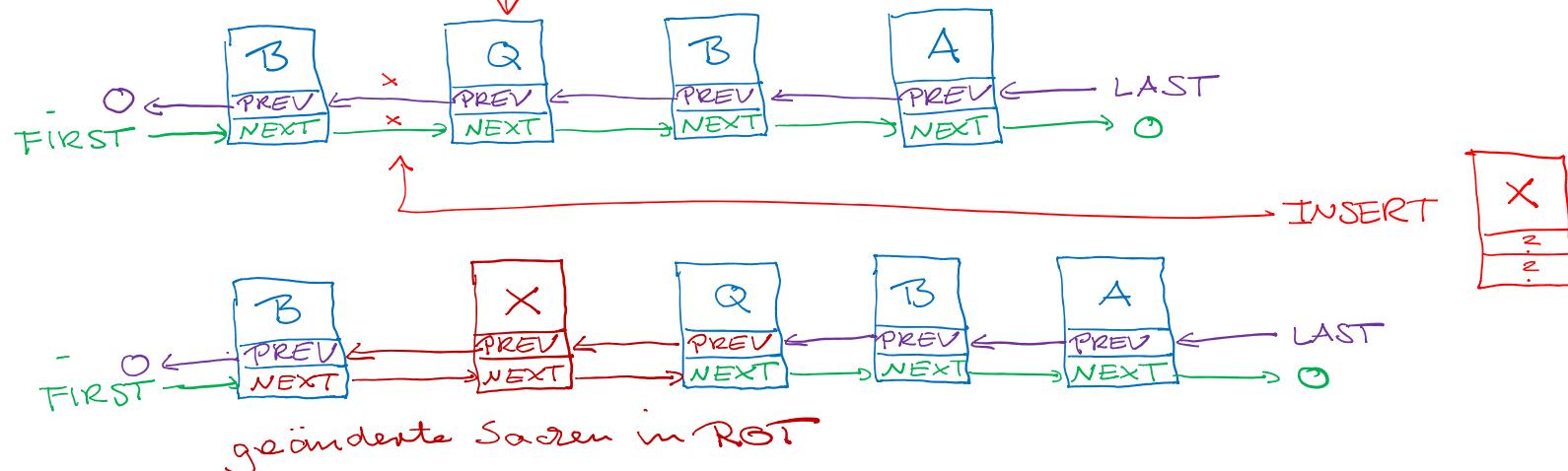
- Im Prinzip nicht schwer: man muss nur die betroffenen Zeiger / Referenzen (siehe Folie 12) richtig "umbiegen"

Den Nachfolgerzeiger vom Vorgänger

Den Vorgängerzeiger vom Nachfolger

Die beiden Zeiger des eingefügten Elementes

Evtl. die Zeiger auf das erste und das letzte Element



# Verkettete Listen 3/8

## ■ Angabe der Einfügestelle

- **Variante 1:** Referenz auf den neuen Nachfolger `next_item`

Dann nennt man die Operation oft **insert\_before**

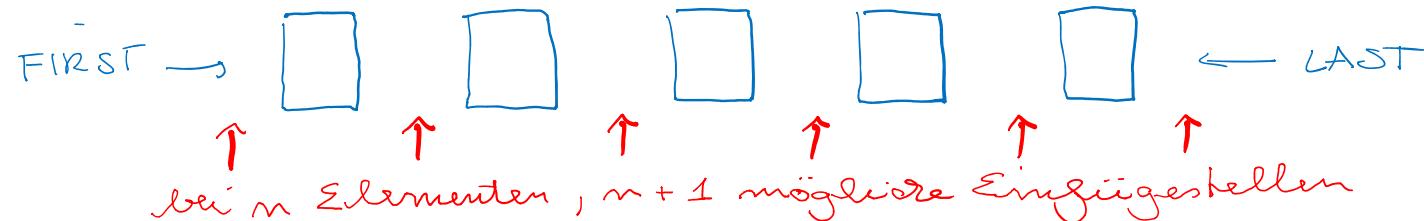
Einfügen am Ende, indem man `next_item = 0` übergibt

- **Variante 2:** Referenz auf den neuen Vorgänger `prev_item`

Dann nennt man die Operation oft **insert\_after**

Einfügen am Anfang, indem man `prev_item = 0` übergibt

- **Variante 3:** Über einen zusätzlichen Typ **ListIterator**,  
der auf die  $n + 1$  möglichen Stellen in einer Liste mit  $n$   
Elementen zeigt ... so wird es in Java und C++ gemacht



# Verkettete Listen 4/8

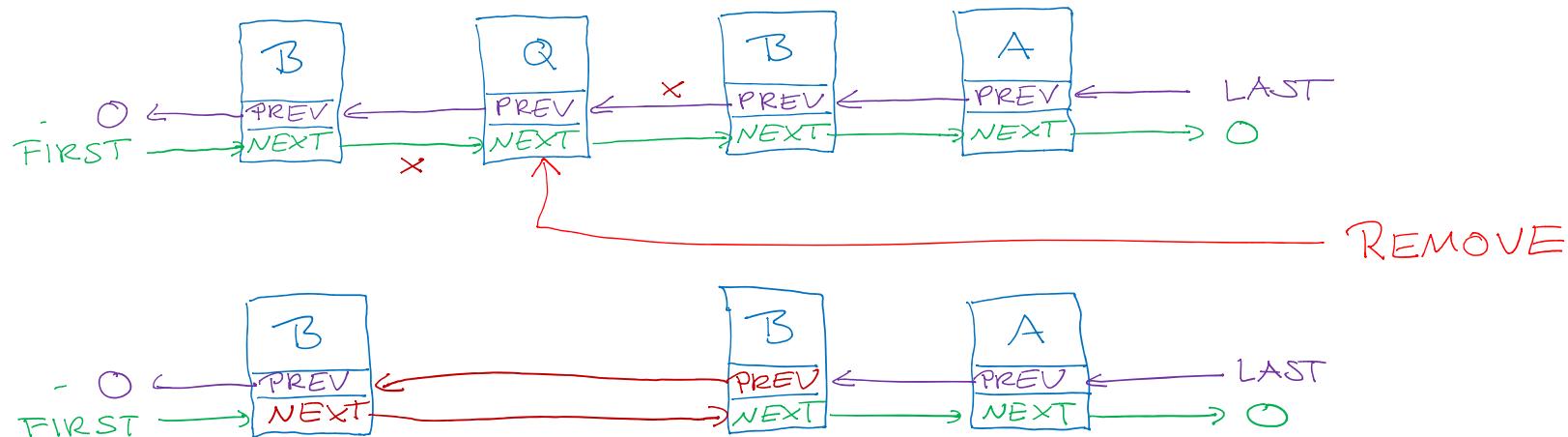
## ■ Entfernen (remove) eines geg. Elementes item

- Auch hier müssen einfach nur die richtigen "Zeiger" "umgebogen" werden

Nachfolgezeiger des Vorgängers von item

Vorgängerzeiger des Nachfolgers von item

Evtl. die Zeiger auf das erste und das letzte Element



# Verkettete Listen 5/8

---

## ■ Implementierung

- Das machen wir jetzt mal zusammen (in Python)  
Der Einfachheit halber nur mit Einfügen (`insert_before`)
- Sobald man mit Zeigern hantiert, kann man sehr viele Fehler machen, die sehr schwer zu debuggen sind
- Deswegen ist eine gute Funktion, die den aktuellen Zustand einer Liste anzeigt, hier entscheidend wichtig

Das wird uns die Hälfte der Arbeit kosten, aber das ist in dem Fall hier absolut angemessen

Das ist eine der take-home-messages für heute: Zeit für gute Funktionen zum Debuggen ist oft bestens investiert

## ■ Anzahl der Elemente

- Ohne Weiteres muss man einmal von vorne nach hinten durch die Liste gehen, um die Anzahl Elemente zu ermitteln

Laufzeit dafür  $\Theta(n)$ , wenn  $n$  = Anzahl Elemente

- Es geht aber auch leicht in  $O(1)$  Zeit

Man kann ja einfach separat einen Zähler haben, und macht bei jedem insert +1 und bei jedem remove -1

# Verkettete Listen 7/8

## ■ Zugriff auf das i-te Element

- In einer verketteten Liste können die Elemente **beliebig** im Speicher verteilt stehen
- Will man das  $i$ -te Element haben, bleibt einem nichts anderes übrig, als sich "durchzuhangeln"

Von vorne oder von hinten, was immer schneller ist

- Die Laufzeit dafür ist also  $\Theta(\min\{i, n - i\})$
- Zum Vergleich: in einem Feld stehen die Elemente immer garantiert hintereinander im Speicher
- Von daher Zugriff auf das i-te Element in **O(1)** Zeit  
Steht an Stelle Anfangsadr. +  $i * \text{Größe eines Elementes}$

# Verkettete Listen 8/8

---

## ■ In Java, C++ und Python

- In Java: `java.util.LinkedList<T>` ... in C++: `std::list<T>`
- In Python gibt es keine native verlinkte Liste

Einfügen an beliebiger Stelle kein häufiger Anwendungsfall

Hier in der VL vor allem als Grundlage für Vorlesung 8a+b

- Für die Vorlesung heute (und das Übungsblatt) wollen wir verkettete Listen **von Grund auf selbst** implementieren
- Das Konzept des "Zeigers" gibt es in allen drei Sprachen:

In C++ benutzt man wörtlich Zeiger ... `LinkedListItem*`\*

In Java und Python sind Namen von Objekten grundsätzlich Referenzen, also auch Zeiger ... `LinkedListItem`

# Laufzeit Listen vs. Felder 1/5

## ■ Laufzeit doppelt verkettete Liste

- Einfügen an beliebiger Stelle:  $\mathcal{O}(1)$  ✓
- Entfernen an beliebiger Stelle:  $\mathcal{O}(1)$  ✓
- Zugriff auf  $i$ -tes Element der Liste:  $\mathcal{O}(\min\{i, m-i\})$  ✗

## ■ Laufzeit dynamisches Feld

- Einfügen am Ende:  $\mathcal{O}(1)$  amortisiert (=im Durchschnitt) manchmal teuer ✓
- Entfernen am Ende: ~~ditto~~ ✓
- Zugriff auf  $i$ -tes Element der Liste:  $\mathcal{O}(1)$  ✓
- Einfügen an  $i$ -ter Stelle:  $\mathcal{O}(m-i)$  ✗
- Entfernen an  $i$ -ter Stelle:  $\mathcal{O}(m-i)$  ✗

## ■ Zeitmessung in der Praxis

- Beim Einfügen / Entfernen am Ende scheinen Liste und dynamisches Feld gleich gut zu sein

Die Liste sieht sogar besser aus, weil **immer**  $O(1)$  und das dynamische Feld nur **amortisiert**  $O(1)$

- Die Laufzeit wollen wir jetzt mal konkret nachmessen

Wir fügen dabei der Einfachheit halber nur am Ende ein

- Beobachtung:

In C++, `std::list` ca. 4 mal LANGSAMER als der `std::vector`

## ■ Laufzeitunterschied, Grund 1

- Bei der Liste müssen wir für jede Operation **vier** Zeiger umbiegen

Beim dynamischen Feld müssen wir ohne Reallokation einfach nur **einen** Eintrag schreiben

Und die Reallokationen fallen bei geeigneten Parametern (z.B.  $f = 0.5$  im Sinne des ÜB6) nicht ins Gewicht

## ■ Laufzeitunterschied, Grund 2

- Bei der Liste müssen wir für jedes Element **einzelN** Speicher allozieren

Beim dynamischen Feld tun wir das für viele Elemente **auf einmal**

Jede Speicherallokation hat fixe Kosten, die unabhängig von der Größe des allozierten Speichers sind

Außerdem benötigen wir durch die Zeiger pro Element auch insgesamt etwas mehr Platz

## ■ Laufzeitunterschied, Grund 3

- Das dynamische Feld hat eine viel bessere sogenannte **Lokalität** der Speicherzugriffe

Bei einem Feld stehen ja, wie gesagt, die n Elemente im Speicher hintereinander

Bei einer verketteten Liste können die Elemente beliebig im Speicher verteilt stehen

Warum das einen Unterschied macht, ist genau das Thema von VL7b

# Literatur / Links

---

- Doppelt verkettete Liste
  - Wikipedia  
[https://en.wikipedia.org/wiki/Linked\\_list](https://en.wikipedia.org/wiki/Linked_list)
- Allozieren
  - <http://www.duden.de/rechtschreibung/allozieren>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 7b, Mittwoch, 14. Juni 2017  
(Fortsetzung Verkettete Listen, Cache-Effizienz)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Inhalt

- Verkettete Listen Fortsetzung von gestern
  - Listen vs. Felder Laufzeitvergleich + Analyse
  - Lokalitt Speicherzugriffe Definition + Hintergrnde
  - Blockoperationen Alternatives Effizienzma

## ■ Einfach(st)es Beispiel

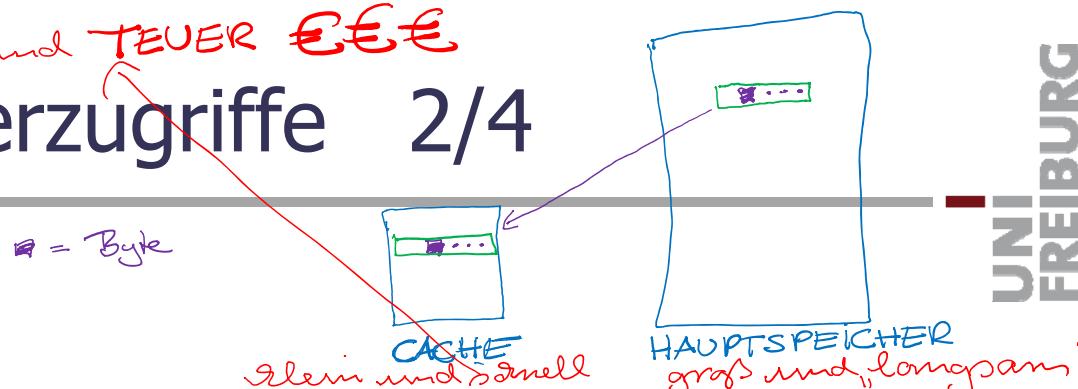
- Wir addieren die  $n$  Elemente eines Feldes auf
  - ... in der natürlichen Reihenfolge:  $1 + 2 + 3 + 4 + 5$
  - ... in einer zufälligen Reihenfolge:  $2 + 5 + 3 + 1 + 4$
- Das Ergebnis ist in beiden Fällen **identisch**
- Die Anzahl der Operationen ist ebenfalls **identisch**
- **Beobachtung:**

$m = 10.000.000$  : „random“ ca. 10 mal LANGSAMER

$m = 100.000.000$  : „random“ ca. 20 mal LANGSAMER

# Lokalität Speicherzugriffe 2/4

## ■ CPU Cache, Prinzip



- Zugriff auf ein Byte im Hauptspeicher kostet ca. 50 - 100ns
- Zugriff auf ein Byte im Level-1 (L1) Cache kostet ca. 1ns
- Bei Zugriff auf ein oder mehrere Bytes im Hauptspeicher holt man gleich einen ganzen Block (cache line) in den Cache

Größe einer cache line für x86 Prozessoren: 64 Bytes

- Solange dieser Block im Cache ist, braucht man für Bytes aus diesem Block nicht mehr auf den Hauptspeicher zuzugreifen
- Der Cache hat Platz für viele solcher Blöcke

Typische Größe eines L1-Cache: 32 KB (= 500 cache lines)

Unter LINUX: `getconf -a | grep CACHE`

$$1 \text{ MB/s} \hat{=} 1 \mu\text{s/Byte}$$

## ■ Disk Cache, Prinzip

- "Seek Time" ist  $\sim 5\text{ms}$  (HDD) bzw.  $\sim 0.1\text{ms}$  (SSD)  
Bei HDD: Lesekopf muss an die Stelle gehen, wo die gewünschten Daten stehen
  - Transferrate ist  $\sim 50 \text{ MB/s}$  (HDD) bzw.  $\sim 200 \text{ MB/s}$  (SSD)  
Bei HDD: Kopf bleibt stehen, Platte dreht sich schnell weiter
  - Deshalb geht das Betriebssystem wie folgt vor  
Wird ein Byte von der Platte gelesen, wird gleich ein ganzer Block eingelesen ... z.B. 128 KB auf einmal  
Solange dieser Block im Speicher ist, braucht man für Bytes aus diesem Block nicht mehr auf die Platte zugreifen
- Also dasselbe Prinzip wie beim CPU Cache

# Lokalität Speicherzugriffe 4/4

---

## ■ Wenn der Cache voll ist

- ... muss einer der Blöcke entfernt werden, dafür gibt es zahlreiche Strategien, zum Beispiel:
  - **LRU** (Least Recently Used) = der Block, für den es am längsten her ist, das darauf zugegriffen wurde
  - **LFU** (Least Frequently Used) = der Block, auf den am wenigsten zugegriffen wurde, seit er im Cache ist
- Das ist aber nicht das Thema der Vorlesung heute
  - Für unsere einfachen Analysen heute und für das ÜB7 spielt es keine Rolle, welche Strategie verwendet wird

# Blockoperationen 1/8

---

## ■ Abstraktion der bisherigen Beobachtungen

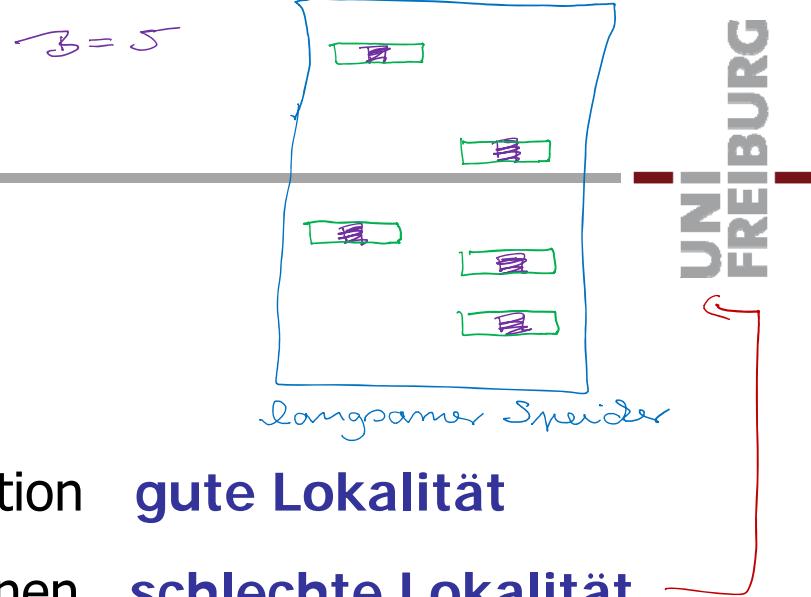
- Es gibt einen **langsamen** und einen **schnellen** Speicher
- Beide Speicher sind in Blöcke der Größe **B** unterteilt
- Der schnelle Speicher ist **M** groß = Platz für **M/B** Blöcke
- Stehen die Daten nicht im schnellen Speicher, wird der entsprechende Block in den schnellen Speicher geladen
- Das Programm kann sich aussuchen, welche Blöcke im schnellen Speicher gehalten werden
- **Wir zählen nur die Anzahl der Blockoperationen**

Also +1 für jedes Mal, wenn ein Block in den schnellen Speicher geladen wird, der da gerade nicht drin steht

## ■ Was wir alles vernachlässigen

- Sämtliche Berechnung auf einem Block im schnellen Speicher
- Kosten für das Verwalten der Blöcke im schnellen Speicher
- Wie genau der langsame Speicher in Blöcke unterteilt ist
- Ob eine Operation 1, 2, 4 oder 8 Bytes liest
- Grund 1: die Zeit, die man braucht, um einen Block in den schnellen Speicher zu holen, dominiert oft alles andere
- Grund 2: die anderen Sachen machen nur einen konstanten Faktor Unterschied → egal für  $O(\dots)$  oder  $\Theta(\dots)$  Schranken

# Blockoperationen 3/8



## ■ Gute vs. Schlechte Lokalität

- Für  $B$  Operationen hat man also:
  - Im "best case" nur 1 Blockoperation **gute Lokalität**
  - Im "worst case"  $B$  Blockoperationen **schlechte Lokalität**
- Sonderfall: für  $n \leq M$  (Eingabe passt ganz in den schnellen Speicher) hat man trivial  $\lceil n/B \rceil$  Blockoperationen

Eine Analyse der Anzahl Blockoperationen ist also erst für große bzw. sehr großen Eingaben ( $n \gg M$ ) interessant

# Blockoperationen 4/8

## ■ Typische Werte (für einen Server)

- CPU L1-Cache:  $B = 64 \text{ Bytes}$ ,  $M = 32 \text{ KB}$  512 blocks
- CPU L2-Cache:  $B = 64 \text{ Bytes}$ ,  $M = 256 \text{ KB}$  4096 blocks
- CPU L3-Cache:  $B = 64 \text{ Bytes}$ ,  $M = 8 \text{ MB}$  131072 blocks
- Disk Cache:  $B = 64 \text{ KB}$ ,  $M = 64 \text{ GB}$   $\approx 1 \text{ million blocks}$

Die meisten Betriebssysteme benutzen alles, was vom Hauptspeicher gerade nicht genutzt wird, als Disk Cache

- Sinnvollerweise wählt man dabei  $B$  so, dass die transfer time für einen Block ungefähr gleich der seek time ist

Wenn man schon die viele Zeit für ein "seek" aufwendet, kann man auch gerade noch mal so viel Zeit aufwenden, um möglichst viele Elemente auf einmal zu lesen

# Blockoperationen 5/8

---

## ■ Terminologie

- Blockoperationen nennt man beim CPU Cache: in der Regel **cache misses**  
Weil sie dann nötig werden, wenn ein Stück vom (langsamem) Hauptspeicher nicht im (schnellen) Cache ist
- Disk Cache: oft einfach **I/Os**  
IO oder I/O = Input/Output ... eher historische Bezeichnung für Datentransfer von der oder auf die Platte
- Wenn man die Anzahl Blockoperationen eines Algorithmus analysiert, spricht man deswegen oft von seiner **Cache-Effizienz** oder **IO-Effizienz**

## ■ IO-Effizienz von **ArraySumMain** 1/2

- Wenn wir über die  $n$  Elemente in der Reihenfolge  $1, 2, 3, \dots$  iterieren, dann ist die Anzahl Blockoperationen

im best case:  $\lceil n/B \rceil$

im worst case:  $\lceil n/B \rceil$

- Wenn wir über die  $n$  Elemente in einer zufälligen Reihenfolge iterieren, dann ist die Anzahl Blockoperationen

im best case:  $\lceil n/B \rceil$  ... aber sehr unwahrscheinlich  
bei  $n \gg M$

im worst case:  $n$

# Blockoperationen 7/8

---

## ■ IO-Effizienz von **ArraySumMain** 2/2

- In der Praxis ist der Unterschied zwischen den beiden Varianten kleiner als die Blockgröße ( $B = 64$ )
- Grund: der durchschnittliche Fall (average case) liegt irgendwo zwischen best case und worst case

Wenn  $n$  nicht viel größer als  $M$  ist, steht das nächste Element manchmal schon zufällig im schnellen Speicher

Außerdem wird auch bei zufälliger Reihenfolge pro Element auf 4 benachbarte Bytes (ein **int**) auf einmal zugegriffen

## ■ IO-Effizienz von MergeSort

- Kurze Wiederholung der Funktionsweise:
  - Teile das Feld in zwei gleich große Teile
  - Sortiere die beiden Teile rekursiv
  - Mische die beiden sortierten Folgen zu einer sortierten Folge
- Mischen von zwei Folgen der Gesamtlänge  $n$  geht mit  $\text{IO}(n) \leq [n/B]$  Blockoperationen
- Außerdem ist  $\text{IO}(n) = 1$  für  $1 \leq n \leq B$
- Durch Auflösen der Rekursion kann man dann zeigen, dass  $\text{IO}(n) = \Theta(n/B \cdot \log_2(n/B))$
- Man kann sogar zeigen:  $\text{IO}(n) = \Theta(n/B \cdot \log_{M/B}(n/B))$

# Literatur / Links

---

## ■ Cache-Effizienz / IO-Effizienz

- In Mehlhorn/Sanders:  
[2 Introduction 2.2.1 External Memory](#)
- In Wikipedia
  - <http://en.wikipedia.org/wiki/Cache>
  - <http://de.wikipedia.org/wiki/Cache>

Da wird das Prinzip eines Caches beschrieben, es gibt aber keinen separaten Artikel zur Cache-Effizienz bei Algorithmen

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 8a, Dienstag, 20. Juni 2017  
(Sortierte Folgen, Binäre Suchbäume)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Organisatorisches

- Erfahrungen mit dem ÜB7                      Listen, Cache-Effizienz
- Was ist ein Plagiat                              Erinnerung + Beispiel

## ■ Inhalt

- Sortierte Folgen                                  Definition + Beispiel
- Binäre Suchbäume
- ÜB8: ein praktisches Problem, und Sie müssen selber schauen, welche Datenstrukturen geeignet sind  
(Bewusst weniger Vorgaben als sonst)

Achtung: stellen wir heute gegen 17 Uhr erst online

## ■ Zusammenfassung / Auszüge

- Aufgabe 1 gut machbar + hat vielen viel Spaß gemacht
- Verständnisschwierigkeiten bei Aufgabe 2 (Block-Ops)

Häufige Rückmeldung: "Waren die Aufgaben diesmal einfacher, oder habe ich etwas falsch gemacht?"

**Wichtig: das zu erkennen ist ein wichtiger Verständnistest**

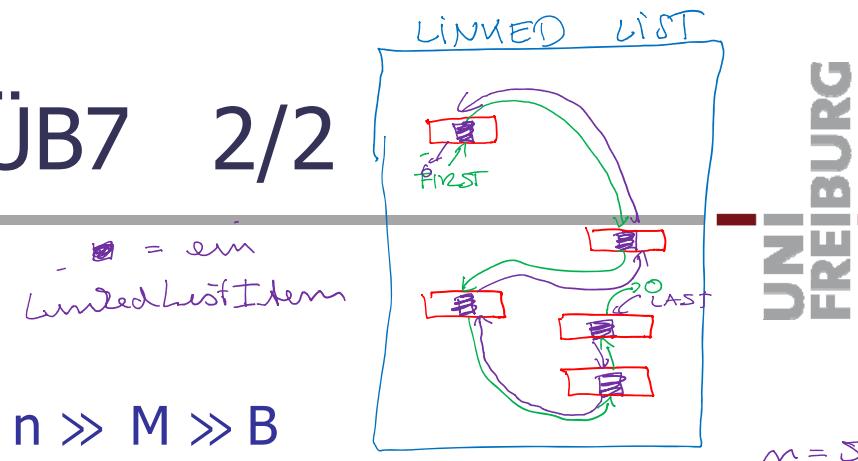
- "Sobald die Elemente auf dem Heap liegen, ist es mit der Cache-Effizienz eh dahin"
- Können sie bei Gelegenheit den Livestream an die Tafel projizieren, um Endlosrekursion zu erzeugen?

**Gute Idee! Geht auch zu Hause mit Handy+TV+SmartView**

# Erfahrungen mit dem ÜB7 2/2

## ■ Lösungsskizze Aufgabe 2

- Sie konnten annehmen, dass  $n \gg M \gg B$   
Stand nicht explizit auf dem ÜB, wurde im Forum geklärt



- Anzahl Blockoperationen von **reverse()**  
Ohne Weiteres muss man für reverse() alle  $n$  Elemente der Liste anschauen und verändern

Die Elemente stehen an beliebigen Stellen im Speicher

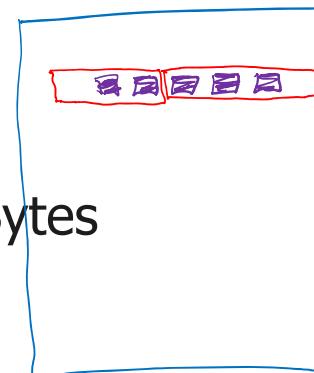
ARRAY

Also im worst case  $\Theta(n)$  Blockoperationen

- Anzahl Blockoperationen von **splice()**

Man muss nur an zwei Stellen konstant viele Bytes anschauen bzw. verändern

Also immer  $\Theta(1)$  Blockoperationen



# Was ist ein Plagiat

---

## ■ Erinnerung + Beispiel

- Im Rahmen von Lehrveranstaltungen geht es vor allem um das Übernehmen fremder Texte (inklusive Code)
- Ab wann gilt ein Text als übernommen vs. selbst erdacht?

Einzelne Worte darf man offenbar wieder verwenden

Ganze Sätze sind in der Regel schon einzigartig

Schon die Aneinanderreihung von  $k$  Worten ist oft so charakteristisch, dass man damit ein Dokument eindeutig identifizieren kann, für relativ kleine  $k$

Beispiele von unseren Webseiten:

"Das bedeutet, dass wir uns komplexe"

"research interest is aptly described"

## ■ Problem

- Wir wollen wieder (key, value) Paare / Elemente verwalten
- Wir haben wieder eine Ordnung  $<$  auf den Keys
- Diesmal wollen wir folgende Operationen unterstützen

`insert(key, value)`: füge das gegebene Paar ein

`remove(key)`: entferne das Paar mit dem gegebenen Key

`lookup(key)`: finde ein Element mit kleinsten Key  $\geq$  key

`next / previous(element)`: finde das Element mit dem nächstgrößeren / nächstkleineren Schlüssel, falls es existiert

(so, dass Iteration über alle Elemente möglich ist)

muss nicht  $=$  key sein

# Sortierte Folgen 2/6

---

## ■ Typisches Anwendungsbeispiel: Bereichssuche

- Ein große Menge von Objekten

Zum Beispiel Bücher, Wohnungen, sonstige Produkte

- Typische Suchanfrage: alle Wohnungen zwischen 400 und 600 Euro Monatsmiete

Das bekommt man mit **lookup** und **next**

Man beachte: es ist dafür nicht wichtig, dass es eine Wohnung gibt, die **genau** 400 Euro kostet

- Wenn man ein paar Objekte hinzufügt oder alte löscht, will man nicht jedes Mal erst alles wieder neu sortieren

# Sortierte Folgen 3/6

wo die Elemente in  
sortierter Reihenfolge  
stehen

## ■ Lösung 1: Einfaches (dynamisches) Feld

- lookup in Zeit:  $\Theta(\log n)$

wenn man bei mehreren gleichen KEYS  
das "linkste" haben will  
lookup (30)

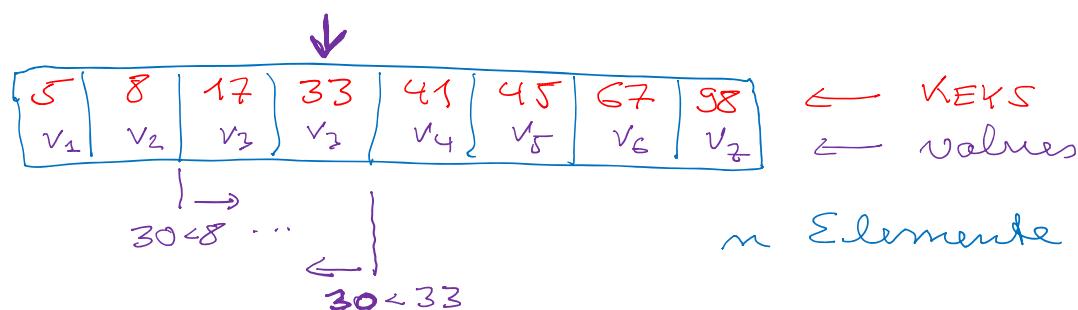
Das geht mit binärer Suche, siehe unten

- next und previous in Zeit:  $\Theta(1)$

Benachbarte Elemente stehen direkt nebeneinander

- insert und remove in Zeit: *lief zu  $\Theta(n)$*

Bis zu  $\Theta(n)$  Elemente müssen umkopiert werden



# Sortierte Folgen 4/6

Looken(key):  
rechter Schlüssel  $\geq$  key

## ■ Lösung 2: Hashtabellen

- insert und remove in Zeit:  $O(1)$  im Erwartungsfall  
wenn  $m = \Theta(n)$

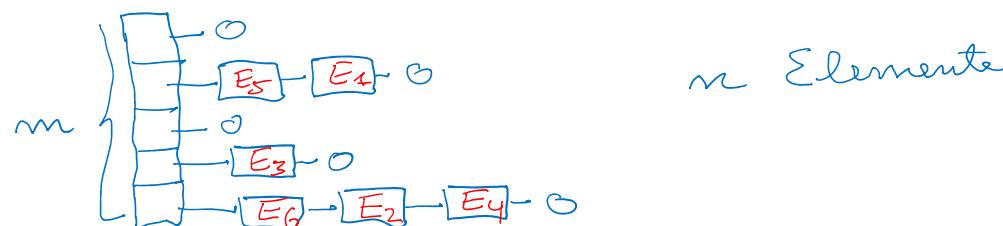
Bei genügend großer Hashtabelle und guter Hashfunktion

- lookup in ~~erwarteter~~ Zeit:  $O(1)$  oder bis zu  $\Theta(m)$

Aber nur wenn es ein Element mit **exakt** dem gegebenen Key gibt, sonst bekommt man **gar nichts**

- next und previous in Zeit:  $\text{bis zu } \Theta(n)$

Die Reihenfolge, in der die Elemente in einer Hashtabelle stehen hat nichts mit der Reihenfolge der Keys zu tun!



# Sortierte Folgen 5/6

## ■ Lösung 3: (Doppelt) Verkettete Listen

- next und previous in Zeit:  $\mathcal{O}(1)$

Jedes Element hat einen Zeiger zum Vorgänger / Nachfolger

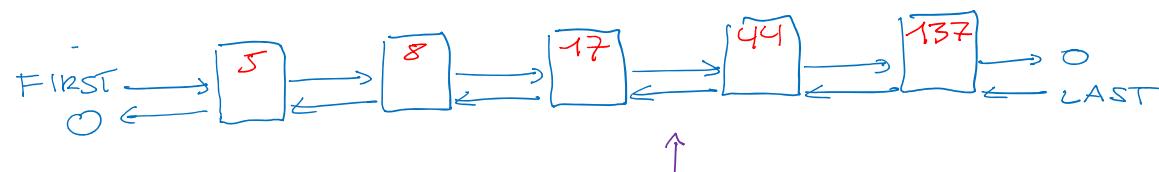
- insert und remove in Zeit:  $\mathcal{O}(1)$

Es müssen nur konstant viele Zeiger umgesetzt werden

- lookup in Zeit: *Zeit zu  $\mathcal{O}(n)$*       *Lookup ( $\mathcal{O}(n)$ )*

Man könnte die Liste sortiert halten, aber um die richtige Einfügestelle zu finden, muss man sich "durchhangeln"

Binäre Suche geht nicht auf einer verketteten Liste, weil man nicht einfach (wie im Feld) an Position  $i$  springen kann



# Sortierte Folgen 6/6

---

## ■ Lösung 4: Suchbäume

- next und previous in Zeit:  $\mathcal{O}(1)$

Entsprechende Zeiger wie bei der verketteten Liste

- insert und remove in Zeit:  $\mathcal{O}(1)$

Ebenfalls wie bei der verketteten Liste

- lookup in Zeit:  $\mathcal{O}(\log n)$

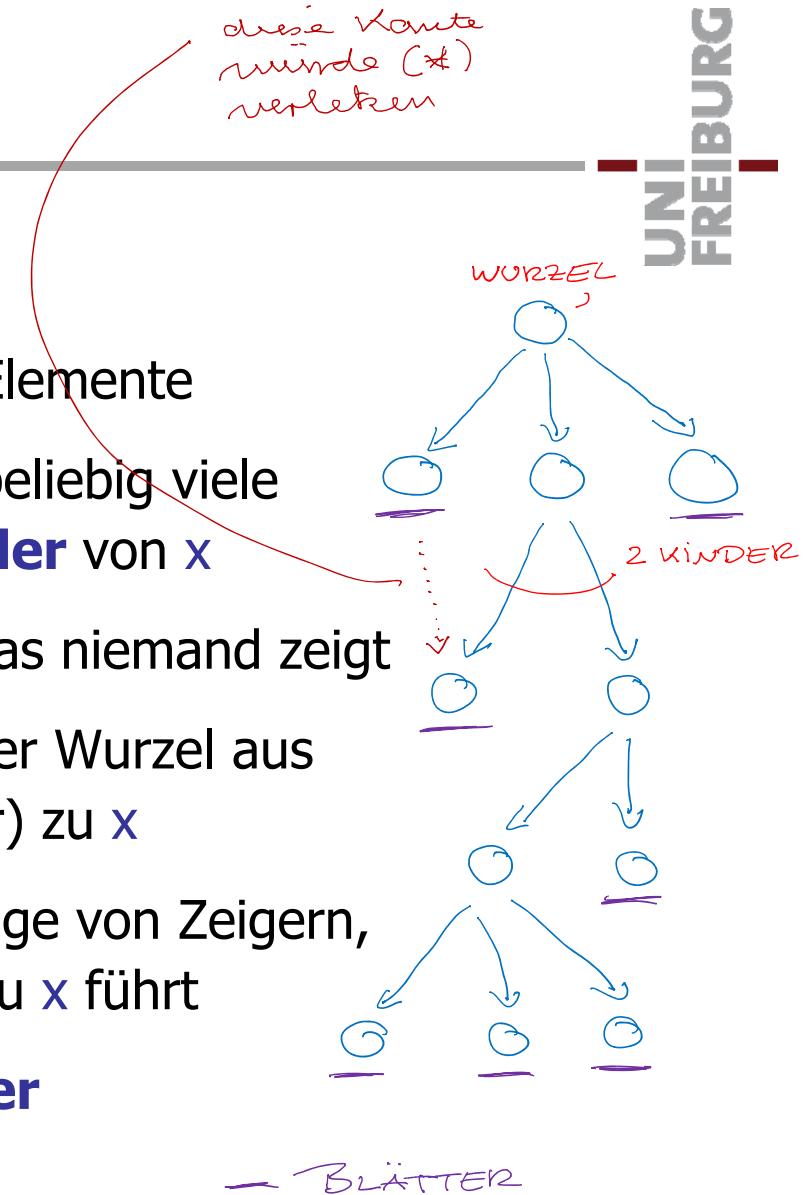
Eine Baumstruktur hilft jetzt beim effizienten Suchen

Wie genau, schauen wir uns im Rest der Vorlesung heute und weiter morgen an

## ■ Allgemeiner Baum, Definition

- Elemente, mit Zeiger auf andere Elemente

Von jedem Element  $x$  Zeiger auf beliebig viele andere Elemente, die heißen **Kinder** von  $x$



Es gibt ein **Wurzel**element, auf das niemand zeigt

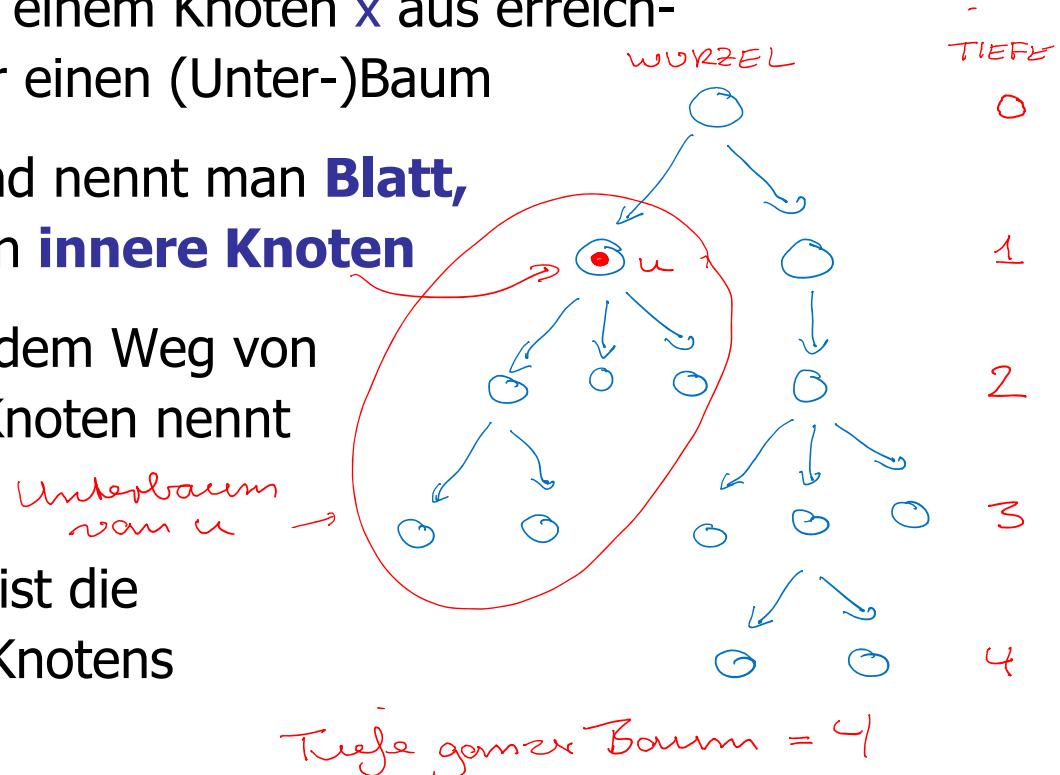
(\*) Für jedes Element  $x$  gibt es von der Wurzel aus genau einen Weg (über die Zeiger) zu  $x$

Es gibt keinen **Zyklus** = keine Folge von Zeigern, die von einem Element  $x$  wieder zu  $x$  führt

Knoten ohne Kinder heißen **Blätter**

## ■ Allgemeiner Baum, Terminologie

- Die Elemente nennt man auch **Knoten** (English: **node**)
- Alle Elemente, die von einem Knoten  $x$  aus erreichbar sind, bilden wieder einen (Unter-)Baum
- Einen Knoten ohne Kind nennt man **Blatt**, die anderen nennt man **innere Knoten**
- Die Anzahl Zeiger auf dem Weg von der Wurzel zu einem Knoten nennt man dessen **Tiefe**
- Die Tiefe des Baumes ist die maximale Tiefe eines Knotens

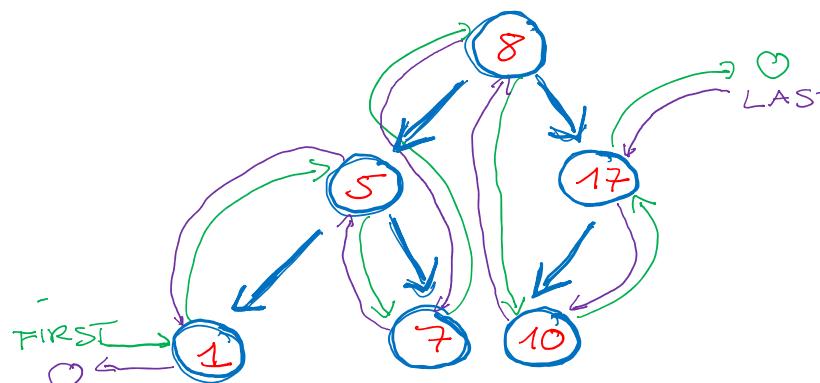


die lassen wir  
bei den folgenden  
Bildern wieder  
weg

## ■ Binärer Suchbaum, Definition

- Jeder Knoten hat **höchstens** zwei Kinder
- Für jeden Knoten gilt: alle Elemente im linken Unterbaum haben einen kleineren Key + alle Elemente im rechten Unterbaum haben einen größeren Key
- Und **gleichzeitig** eine doppelt verkettete Liste der Elemente

Braucht man (nur) für next und previous in  $O(1)$  Zeit

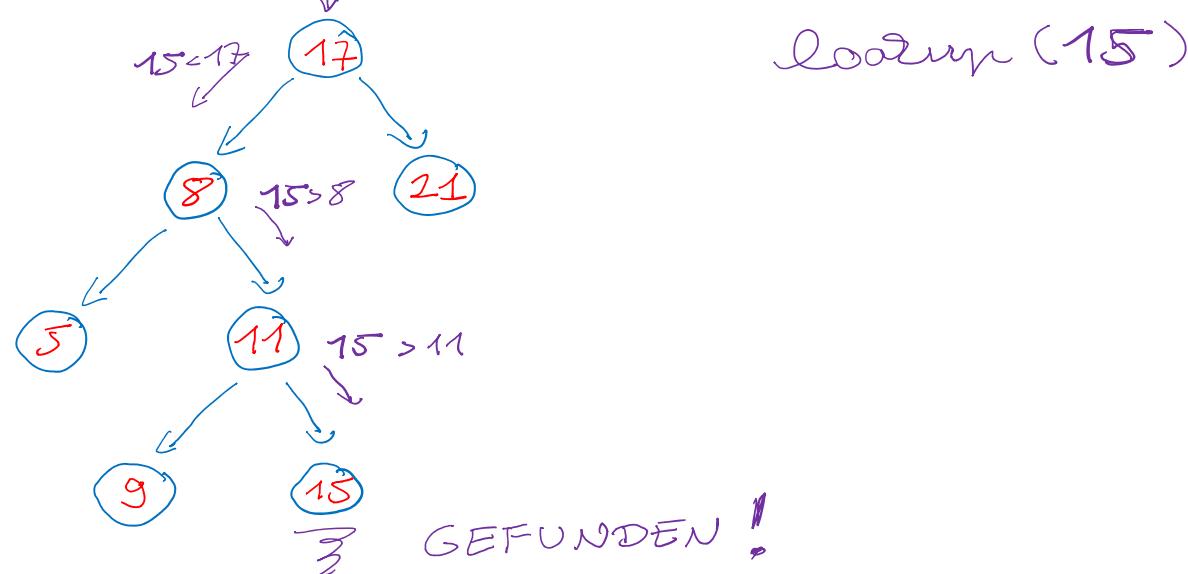


Die values lasse  
ihr die und um  
Folgenden weg

## ■ Die Operation **lookup(x)**

- Von der Wurzel abwärts suchen, und an jeden Knoten `node`
  - falls  $x == node.key$  ... gefunden!
  - falls  $x < node.key$  ... nach links weiter suchen
  - falls  $x > node.key$  ... nach rechts weiter suchen

Wenn es den Key im Baum gibt, findet man ihn so sicher



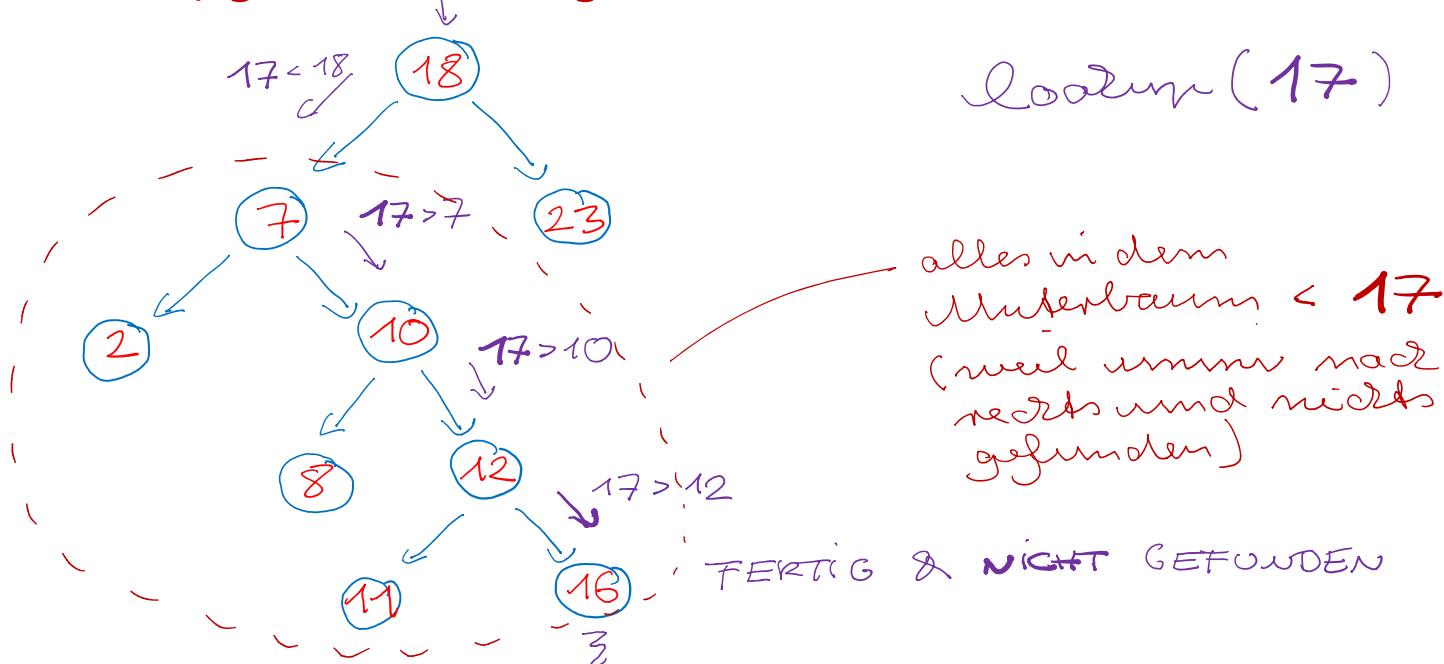
Binärer Suchbaum 5/11

## ■ Die Operation **lookup(x)**

- Wenn es den Key im Baum **nicht** gibt:

Dann ist der nächstgrößere Key an dem Knoten, bei dem man zum letzten Mal nach **links** gegangen ist

Wenn man immer nur nach rechts geht und den Key nie findet, gibt es keinen größeren Schlüssel im Baum



## ■ Die Operation **insert(x, value)**

- Erst mal ein `lookup(x)`
- Wenn es `x` im Baum schon gibt, überschreiben wir einfach das Element an dem Knoten und sind fertig
- Wenn es `x` im Baum **nicht** gibt, können wir so lange nach unten gehen, wie gilt

Entweder: `x < node.key`, und es gibt ein linkes Kind

Oder: `x > node.key`, und es gibt ein rechtes Kind

Laufzeit auf Folie  
vor der Zeit die Laufzeit  
vom LookUp nicht  
dazu gezählt

## ■ Die Operation **insert(x, value)**

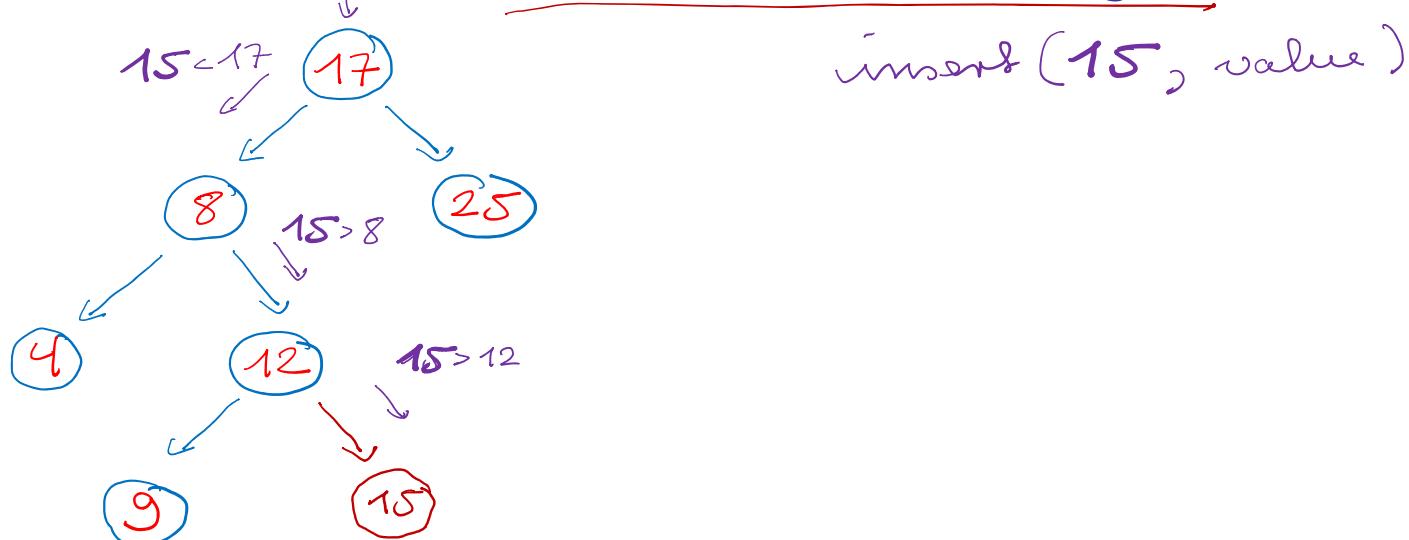
– Wenn es an einem Knoten nicht mehr weitergeht, ist also

entweder:  $x < \text{node.key}$  aber es gibt **kein** linkes Kind

Dann können wir das neue Element links einfügen !

oder:  $x > \text{node.key}$  aber es gibt **kein** rechtes Kind

Dann können wir das neue Element rechts einfügen !



## ■ Laufzeit von `insert` und `lookup`

- In Zeit  **$O(d)$** , wobei  **$d$**  die Tiefe des Baumes ist

Es geht in jedem Schritt eins nach unten, nie nach oben

Und wenn es nicht mehr nach unten geht, ist man fertig

Wenn man den Schlüssel schon weiter oben im Baum findet, kann es auch schneller gehen

- Wir hätten gerne eine Abhängigkeit von der Anzahl  **$n$**  der Elemente ... wie hängt die mit  **$d$**  zusammen ?

## ■ Tiefe des Baumes, best case

- Die Tiefe des Baumes (siehe Folie 12) ist am niedrigsten, wenn jeder innere Knoten zwei Kinder hat

Außer vielleicht einige Knoten der "vorletzten" Tiefe

- Dann ist  $d = \lfloor \log_2 n \rfloor$  ... BEWEIS

$$n > 1 + 2 + 4 + \dots + 2^{d-1} = 2^d - 1$$

$$\Rightarrow n \geq 2^d \Rightarrow d \leq \log_2 n$$

$$n \leq 1 + 2 + 4 + \dots + 2^d$$

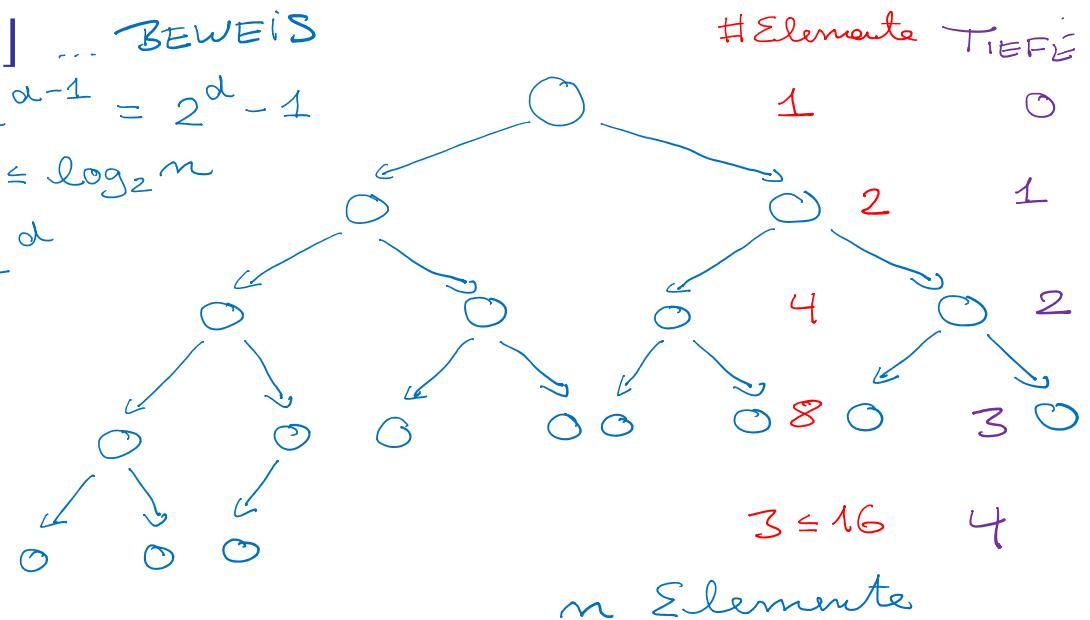
$$= 2^{d+1} - 1$$

$$n < 2^{d+1}$$

$$\Rightarrow d+1 > \log_2 n$$

$$d > \log_2 n - 1$$

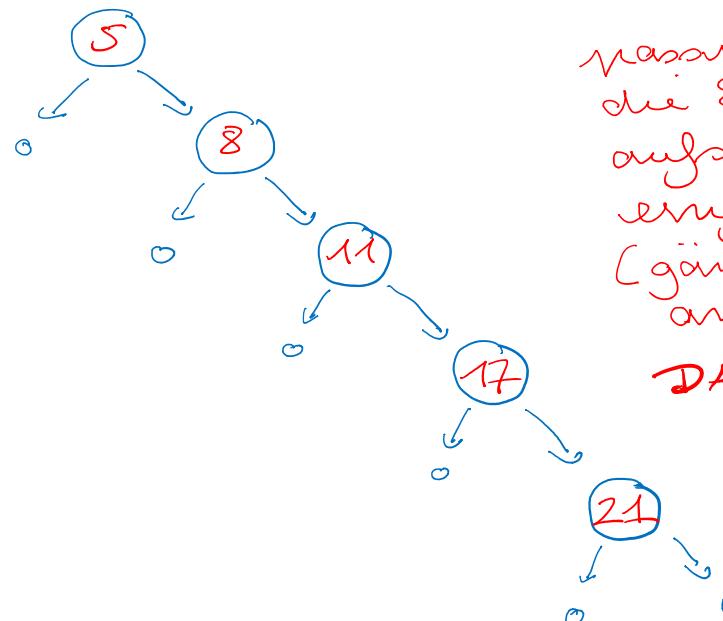
$$\Rightarrow d = \lfloor \log_2 n \rfloor$$



## ■ Tiefe des Baumes, worst case

- Die Tiefe des Baumes (siehe Folie 12) ist am höchsten, wenn jeder innere Knoten nur ein Kind hat
- Dann ist  $d = n - 1$

Wenn man immer  $\Theta(\log n)$  will, muss man den Baum gelegentlich rebalancieren ... das machen wir morgen



passiert, wenn man die Elemente in gänzlich aufsteigender Reihenfolge einfügt  
(gänzlich absteigend Anna - log?)

DAS PASSIERT SCHON MAL

## ■ Verwendung in Java, C++ und Python

- **Java:** `java.util.TreeMap<KeyType, ValueType>`
- **C++:** `std::map<KeyType, ValueType>`
- **Python:** `bintrees.BinaryTree`
- In Python ist `bintrees` nicht Teil der Standardsprache und muss von Hand nachinstalliert werden, z.B. so:

```
wget https://pypi.python.org/.../bintrees-2.0.2.zip
```

```
unzip bintrees-2.0.2.zip
```

```
cd bintrees.2.0.2
```

```
python3 setup.py install --user
```

AxEL sagt, es geht auch:

`pip install bintrees`

Siehe <https://pypi.python.org/pypi/bintrees/2.0.2>

# Literatur / Links

---

## ■ Suchäume

- In Mehlhorn/Sanders:

7 Sorted Sequences

- In Wikipedia

[http://de.wikipedia.org/wiki/Binärer\\_Suchbaum](http://de.wikipedia.org/wiki/Binärer_Suchbaum)

[http://en.wikipedia.org/wiki/Binary\\_search\\_tree](http://en.wikipedia.org/wiki/Binary_search_tree)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 8b, Mittwoch, **längster Tag** 2017  
(Balancierte Suchbäume)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

# ■ Drumherum

- Unser Gehirn Warum so groß / nicht größer?

## ■ Inhalt

- Eindeutige Schlüssel      Was tun, wenn nicht der Fall?

- (a, b)-Bäume Prinzip + viele Beispiele

- (2, 4)-Bäume amortisierte Analyse

- ÜB8, Bonusaufgabe: Potenzialfunktion für (3, 7)-Bäume

Dafür gibt es 5 Bonuspunkte (= man braucht sie nicht für die volle Punktzahl, sie können einem aber dabei helfen, die volle Punktzahl zu erreichen)

reine Verständnisaufgabe, wenig Schreiben erforderlich

# Nicht eindeutige Schlüssel

---

## ■ Umgang damit

- In der VL8a hatten wir angenommen, dass alle Schlüssel (Keys) verschieden sind
- Es ist nicht trivial, den Suchbaum so abzuändern, dass insert und remove nach wie vor in Zeit **O(d)** laufen
  - Z.B. wenn alle Schlüssel gleich: unklar, ob man bei einem inneren Knoten nach links oder nach rechts gehen soll
- In der Praxis zwei typische Lösungsansätze:
  1. Schlüssel **eindeutig machen** (zum Beispiel einen Zähler anhängen: Berlin.1, Berlin.2, Berlin.3, ...)
  2. Pro Schlüssel **eine Menge von Values** speichern (z.B. in einer Liste oder in einem Feld) und nicht nur ein Value

- Warum so groß / nicht größer, Zitate von Ihnen
  - "Gehirn verbraucht sehr viel Energie (vor allem bei Mathe)"
  - "Zu schwer ... würde beim Nachdenken zur Seite kippen"
  - "Würde nicht mehr dem Schönheitsideal entsprechen"
  - "Es wächst ja noch (meinen zumindest die Evolutionsforscher)"
  - "Noch mehr Dickschädel verträgt die Welt nicht"
  - "Allein die Tatsache, dass man diese Frage stellt, zeigt doch, dass wir selbst diese Größe nicht mal verdient haben"
  - "Es kommt nicht auf die Mächtigkeit der Teile eines Systems an, sondern auf die Mächtigkeit ihrer Interaktionen"

## ■ Entwicklung der Größe (Durchschnittswerte)

|                         |                       |
|-------------------------|-----------------------|
| – Dinosaurier           | 100 cm <sup>3</sup>   |
| – Homo habilis          | 600 cm <sup>3</sup>   |
| – Homo erectus          | 1.000 cm <sup>3</sup> |
| – Homo neanderthalensis | 1.500 cm <sup>3</sup> |
| – Homo sapiens          | 1.300 cm <sup>3</sup> |
| – Blauwal               | 7.000 cm <sup>3</sup> |

## ■ Sinn und Zweck des Gehirns

- Aus evolutionärer Sicht **alles andere als klar**
- Unser Gehirn scheint **viel** mehr zu können als zum (guten) Überleben und Fortpflanzen notwendig ist
- Es gibt dazu verschiedene Hypothesen, zum Beispiel

### **Social Brain Hypothesis**

Management von sozialen Beziehungen (Menschen können bis zu 150), deutlich mehr als z.B. bei Schimpansen (ca. 50)

### **Sexuelle Attraktivität**

Beeindrucken des anderen Geschlechts bzw. die Fähigkeit zu beurteilen, ob das auch beindruckend ist

## ■ Der Pfau

- Darwin: "The sight of a feather in a peacock's tail, whenever I gaze at it, makes me sick"

Die Pfauenfedern sind hinderlich beim Fliegen und machen den Träger zu einer leichteren Beute für natürliche Feinde

Aus evolutionärer Sicht also scheinbar eine schlechte Idee

- Allerdings: herausragende Rolle bei der **Partnerwahl** ...  
Studien zeigen starke Korrelation zwischen:
  1. Prächtigkeit des Gefieders
  2. Wahrnehmung sexueller Attraktivität beim Gegenüber
  3. Überlebensfähigkeit des zugehörigen Nachwuchses

## ■ Fisherian Runaway ... Ronald Fisher, 1930

- Laut Fisher reichen schon folgende Voraussetzungen
  1. Das andere Geschlecht findet Merkmal X attraktiv
  2. Bei Merkmal X sind die Nachkommen überlebensfähiger
- Das führt dann zu folgender positiver Feedback-Schleife
  1. Präferenz für Partner mit mehr X
  2. Bei den Nachkommen ist X übermäßig ausgeprägt
  3. Man braucht jetzt besonders viel X um attraktiv zu sein
  4. Es werden Partner mit noch mehr X ausgewählt, usw.
- Ähnliches Prinzip wie bei sozialen Systemen mit bestimmten "Erfolgsmaßen" und der dann eintretenden "Inflation"

## ■ Motivation

- Bei dem einfachen binären Suchbaum von gestern: lookup und insert in Zeit  $\Theta(d)$ , wobei  $d$  = Tiefe des Baumes
- Wenn es gut läuft, ist  $d = O(\log n)$

Zum Beispiel wenn die Schlüssel zufällig gewählt sind

- Wenn es schlecht läuft, ist  $d = \Theta(n)$
- Zum Beispiel wenn der Reihe nach 1, 2, 3, ... eingefügt wird

- Wir wollen uns aber nicht auf eine bestimmte Eigenschaft der Schlüsselmenge verlassen müssen

Das Problem hatten wir auch schon beim Hashing ... die Lösung da waren universelle Klassen von Hashfunktionen

## ■ Wie erreicht man immer Tiefe **O(log n)** ?

- Es gibt Dutzende verschiedener Verfahren dafür:

AVL-Bäume

AA-Bäume

Rot-Schwarz-Bäume

Splay trees

Treaps

...

- Wir machen heute **(a,b)-Bäume**

Die sind intuitiv, einfach, praktisch und biologisch abbaubar

braucht man z.B.  
wenn der Baum sehr  
wenige Elemente hat

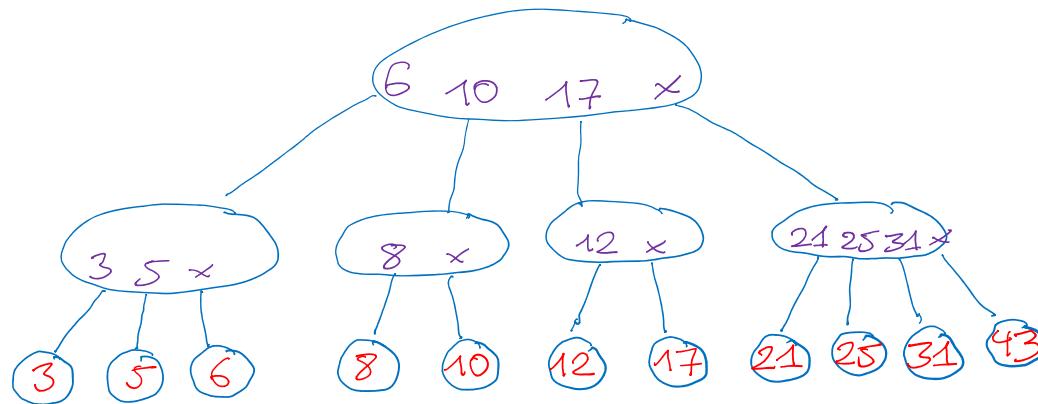
z.B. nur eins:



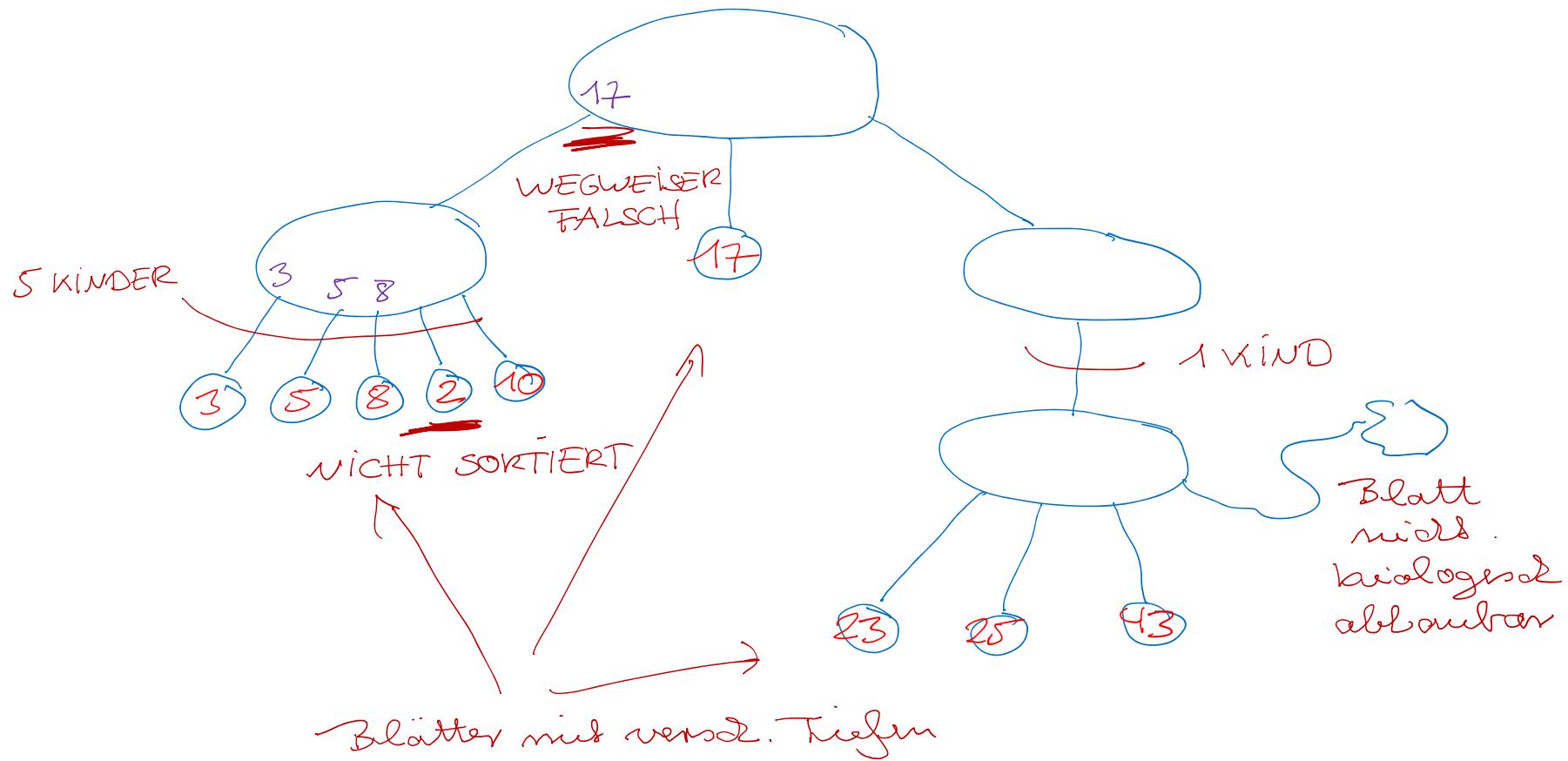
## ■ Definition (a,b)-Baum

- Die Elemente / Schlüssel stehen nur in den Blättern
- Alle Blätter haben die gleiche Tiefe
- Jeder innere Knoten hat  $\geq a$  und  $\leq b$  Kinder
  - Wurzel darf weniger Kinder haben, warum sehen wir gleich
- Wir verlangen  $a \geq 2$  und  $b \geq 2a - 1$ 
  - Warum sehen wir auch gleich
- An jedem inneren Knoten steht **für jedes Kind außer dem rechtesten** der größte Schlüssel in dessen Unterbaum
  - Jeder "Wegweiser" gehört zu genau einem Blatt
- Jedes Blatt weiß, wo sein zugehöriger Wegweiser steht

- Positivbeispiel für einen (2,4)-Baum



## ■ Negativbeispiel für einen (2,4)-Baum

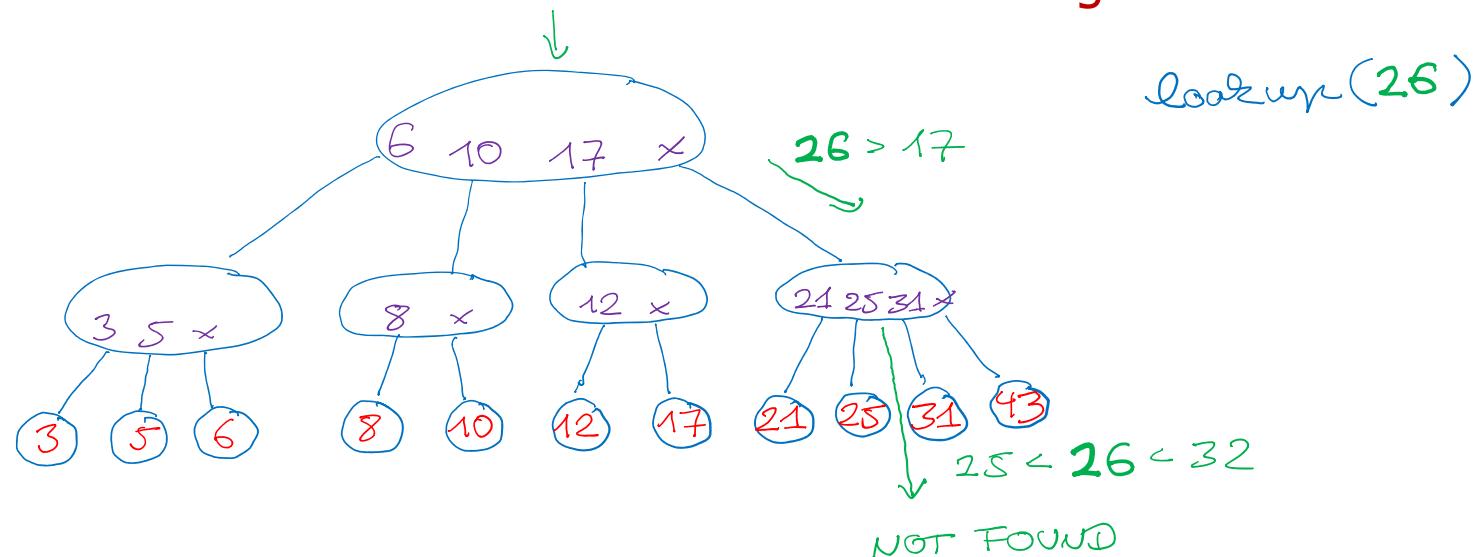


## ■ Die Operation **lookup**

- Im Prinzip genau so wie beim binären Suchbaum

Suche von der Wurzel abwärts, und die Schlüssel an den inneren Knoten weisen den Weg

Bei Knoten mit  $k$  Kindern reichen  $k - 1$  "Wegweiser"



## ■ Die Operation **insert**

- Finde die Stelle, wo der neue Schlüssel einzufügen ist, und füge dort ein neues Blatt ein

Der Elternknoten kann jetzt  $b + 1$  Knoten haben

Falls das der Fall ist, **Aufspalten** des Elternknotens in zwei Knoten, einer mit  $\lfloor b/2 \rfloor$  und einer mit  $\lfloor b/2 \rfloor + 1$  Kindern

Für  $b \geq 2a - 1$  ist  $\lfloor b/2 \rfloor \geq a$  und  $\lfloor b/2 \rfloor + 1 \geq a$

Der Großelternknoten kann jetzt  $b + 1$  Kinder haben

Dann spalten wir den auf dieselbe Weise auf ... usw.

- Wenn das bis zur Wurzel geht, spalten wir auch diese auf und erzeugen einen neuen Wurzelknoten

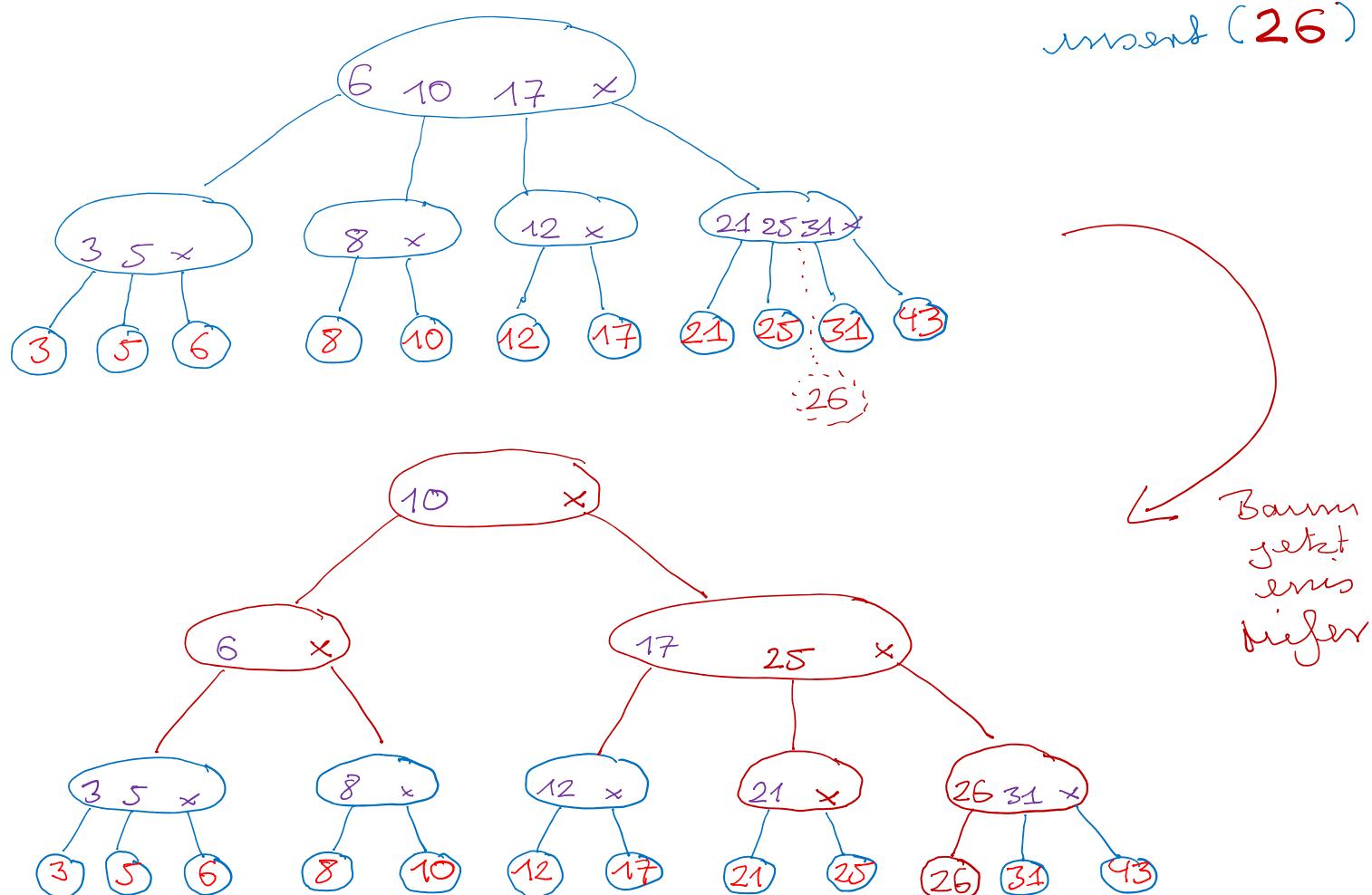
Dann (und nur dann) wird der Baum um **1** tiefer

# (a,b)-Bäume 8/11

alle Änderungen  
in ROT

UNI  
FREIBURG

## Die Operation **insert** ... Beispiel



$$OK, \text{ weil } a+a-1 = 2a-1 \leq b$$

Bedingung  
für (a,b)-Bäume

## ■ Die Operation **remove**

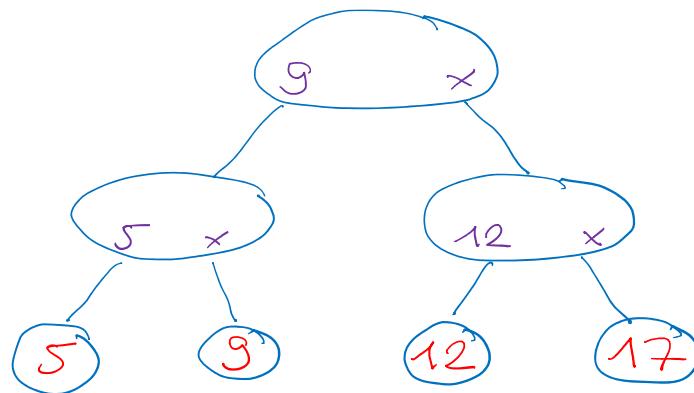
- Finde das zu entfernende Blatt und lösche es  
**Der Elternknoten kann jetzt  $a - 1$  Kinder haben**
- **Fall 1 (Klauen):** Eines der benachbarten Geschwister vom Elternknoten hat  $> a$  Kinder  $\rightarrow$  ein Kind von da klauen
- **Fall 2 (Verschmelzen):** wir verschmelzen den Elternknoten mit einem der benachbarten Geschwister mit nur **a** Kindern  
**Der Großelternknoten kann jetzt  $a - 1$  Kinder haben**  
Damit verfahren wir dann genauso ... usw.
- Wenn das bis zur Wurzel geht und die am Ende nur noch ein Kind hat, mache dieses Kind zur neuen Wurzel  
**Dann (und nur dann) wird der Baum um **1** weniger tief**

# (a,b)-Bäume 10/11

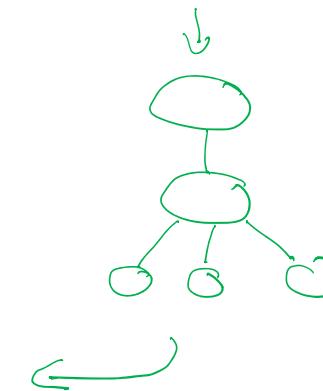
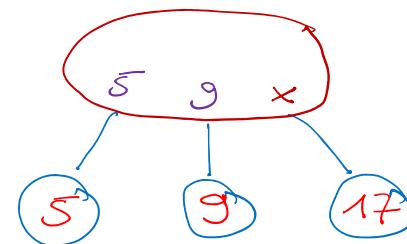
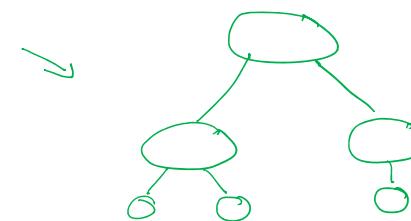
zuris den-  
sche  
im grün

UNI  
FREIBURG

## ■ Die Operation **remove** ... Beispiel



remove (12)



## ■ Update der "Wegweiser"

- Sowohl bei einem `insert` wie bei einem `remove` ändern sich auch einige Wegweiser
- Das ist aber kein Problem, weil (siehe Folie 11):
  - ... jedem Wegweiser genau ein Blatt entspricht
  - ... jedes Blatt weiß, wo sein zugehöriger Wegweiser steht
- Wir können also bei jeder Änderung an den Blättern einfach die zugehörigen Wegweiser mit ändern
- An den inneren Knoten können wir die Wegweiser leicht beim Aufspalten / Verschmelzen anpassen

Asymptotisch dadurch also **keine** zusätzlichen Kosten

## ■ Laufzeit für **lookup**, **insert**, **remove**

- Gehen alle in Zeit  $O(d)$ , wobei  $d$  = Tiefe des Baumes
- Jeder Knoten, außer evtl. der Wurzel, hat  $\geq a$  Kinder  
deshalb  $n \geq a^d$  und deshalb  $d \leq \log_a n = O(\log n)$

- Bei genauerem Hinsehen fällt auf

Die Operation **lookup** braucht **immer** Zeit  **$\Theta(d)$**

Aber **insert** und **remove** gehen "oft" in Zeit  **$O(1)$**

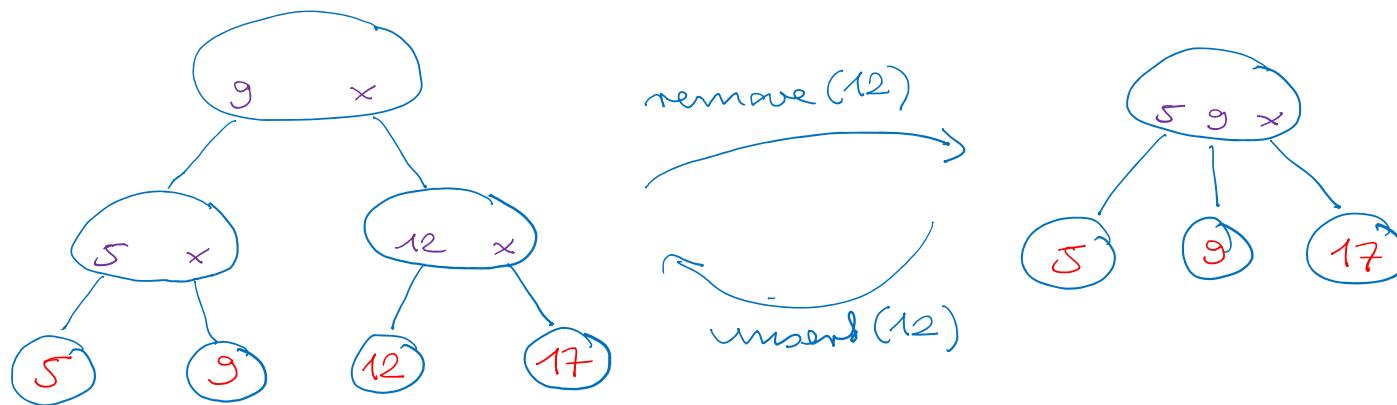
Nur im worst case müssen alle Knoten auf dem Weg zur Wurzel geteilt / verschmolzen werden

- Das wollen wir jetzt genauer analysieren ...

# Analyse (2,4)-Bäume 2/8

## ■ Wir brauchen jetzt $b \geq 2a$

- Für  $b = 2a - 1$  kann man eine Folge von Operationen konstruieren, mit Kosten  $\Theta(d)$  pro Operation
- Beispiel für einen (2,3)-Baum:



## ■ Satz

- Für  $b \geq 2a$  ist die Laufzeit für eine beliebige Abfolge von  $n$  insert oder remove Operationen  **$O(n)$**

Also amortisiert / im Durchschnitt  $O(1)$  pro Operation

- Im Folgenden wollen wir das für  $a = 2, b = 4$  beweisen

ÜB8, Bonusaufgabe: Potenzialfunktion für  $a = 3, b = 7$

Wenn man das Prinzip einmal verstanden hat, ist es auch leicht, das für allgemeine  $a$  und  $b$  mit  $b \geq 2a$  zu beweisen

## ■ Beweis, Intuition

- **Beobachtung:** wann ist ein insert oder remove teuer:

Wenn alle Knoten im Baum **2** Kinder haben, müssen wir nach einem remove alle Knoten bis zur Wurzel verschmelzen

Wenn alle Knoten im Baum **4** Kinder haben, müssen wir nach einem insert alle Knoten bis zur Wurzel aufspalten

Wenn alle Knoten im Baum **3** Kinder haben, dauert es lange bis wir in eine dieser beiden Situationen kommen

- **Idee für Analyse:** nach einer teuren Operation ist der Baum in einem Zustand, dass es dauert, bis es wieder teuer wird

Ähnlich wie bei dynamischen Feldern: Reallokation ist teuer, aber danach dauert es, bis wieder realloziert werden muss

## ■ Terminologie

- Wir betrachten eine Folge von  $n$  Operationen
- Seien  $T_i$  die Kosten = Laufzeit der  $i$ -ten Operation
- Sei  $\Phi_i$  das Potenzial des Baumes nach der  $i$ -ten Operation
  - $\Phi_i$  := die Anzahl der Knoten mit Grad genau 3
  - $\Phi_0 := 0$  (Potenzial am Anfang, für den leeren Baum)

## ■ Mastertheorem aus Vorlesung 6b, Folie 15

- Falls gilt  $T_i \leq A \cdot (\Phi_i - \Phi_{i-1}) + B$  für irgendwelche  $A, B > 0$

Dann  $\sum_{i=1..n} T_i = O(n)$  ... wenn  $\Phi_n = O(n)$  *gibt der Fall, weil #Knoten ≤ n*

# Analyse (2,4)-Bäume 6/8

and vorher Grad 2  
und macht  
Grad 3 möglich,  
dann

$$\Phi_i - \Phi_{i-1} = m + 1$$

## ■ Fall 1: i-te Operation ist ein insert

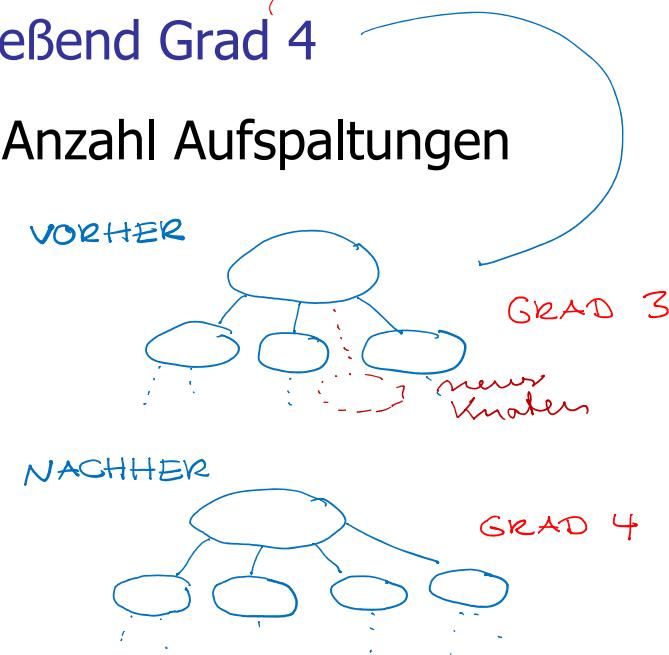
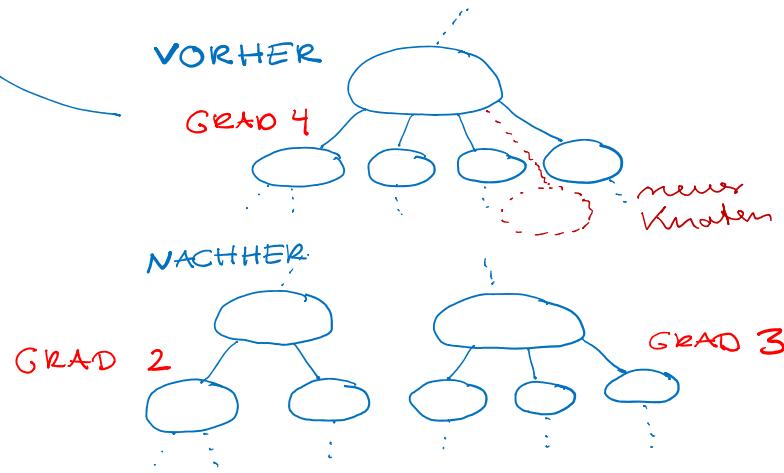
- Pro Aufspaltung erhöht sich das Potenzial um 1

Vorher Grad 4, nachher Grad 2 und Grad 3

- An dem Knoten, an dem die Kette von Aufspaltungen endet, kann sich das Potenzial um 1 verringern

Wenn vorher Grad 3 und anschließend Grad 4

- Also  $\Phi_i - \Phi_{i-1} \geq m - 1$ , mit  $m$  = Anzahl Aufspaltungen



# Analyse (2,4)-Bäume 7/8

## ■ Fall 2: i-te Operation ist ein remove

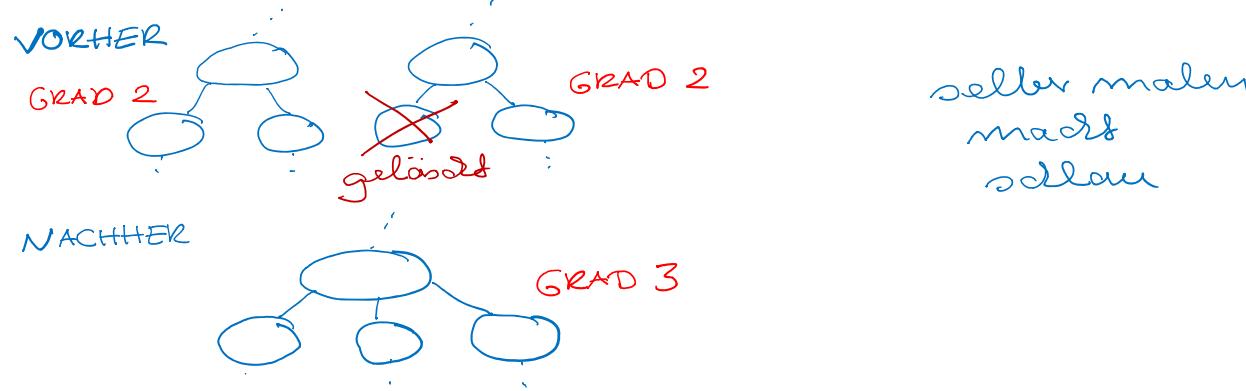
- Pro Verschmelzung erhöht sich das Potenzial um 1

Vorher Grad 2 und 2, nachher Grad 3

- An dem Knoten, an dem die Kette von Verschmelzungen endet, kann sich das Potenzial um 1 verringern

Wenn vorher Grad 3 und anschließend Grad 2 (entweder der Knoten selber, oder der Nachbar von dem man klaut)

- Also  $\Phi_i - \Phi_{i-1} \geq m - 1$ , mit  $m$  = Anzahl Verschmelzungen



## ■ Zusammenfassung Analyse

- In beiden Fällen gilt also  $\Phi_i - \Phi_{i-1} \geq m - 1$

Wobei  $m = \text{Anzahl Aufspaltungen bzw. Verschmelzungen}$

- Daraus folgt  $T_i \leq A \cdot (\Phi_i - \Phi_{i-1}) + B$

Nochmal zur Intuition: das heißt, wenn es teuer wird, dann erhöht sich das Potenzial entsprechend

- Aus dem Master-Theorem folgt dann  $\sum_{i=1..n} T_i = O(n)$

Also amortisiert / im Durchschnitt konstante Laufzeit

$$\begin{aligned}
 T_i &\leq A' \cdot m + B' \quad \text{weil: Kosten Aufspaltung oder} \\
 &\stackrel{(*)}{\leq} A' \cdot (\Phi_i - \Phi_{i-1} + 1) + B' \quad \text{Verschmelzung + } O(1) \\
 &= \underbrace{A' \cdot (\Phi_i - \Phi_{i-1})}_{=: A} + \underbrace{A' + B'}_{=: B}
 \end{aligned}$$

# Literatur / Links

---

- (a,b)-Bäume
  - In Mehlhorn/Sanders:  
    7 Sorted Sequences (Kapitel 7.2 und 7.4)
  - In Wikipedia
    - [http://en.wikipedia.org/wiki/\(a,b\)-tree](http://en.wikipedia.org/wiki/(a,b)-tree)
    - [https://de.wikipedia.org/wiki/\(a,b\)-Baum](https://de.wikipedia.org/wiki/(a,b)-Baum)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 9a, Dienstag, 27. Juni 2017  
(Prioritätswarteschlangen, Binäre Heaps)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- Diese Woche alles **online** Und eine Umfrage dazu
- Erfahrungen ÜB8 Suchbäume

## ■ Inhalt

- Prioritätswürgeschlangen Motivation + Definition
- Implementierung Prinzip + Code + Laufzeit
- ÜB9, Aufgabe 1: Effiziente Berechnung der k größten Städte in unserer cities.txt mit einer PW

Sie müssen die PW nicht selber implementieren, sondern können die von Python/Java/C++ verwenden → Folien 13-15

# Blick über die Vorlesung heute

---

- Diese Woche ist **alles** online (auch die Vorlesungen)
  - Grund: Terminkonflikte diese Woche (siehe Post im Forum)
  - Es gibt aber aktuelle Videoaufzeichnungen
  - Den Beginn haben wir im Voraus neu aufgenommen
  - Der Rest ist von den Aufzeichnungen vom SS 2015
  - Uns interessieren Ihre Erfahrungen damit, insbesondere:
    - Gibt es für Sie einen Nachteil gegenüber der Live-Vorlesung?
    - Warum gab es in den letzten Vorlesungen so wenige Nachfragen? (trotz zahlreicher Anregungsversuche meinerseits)
- Siehe Frage dazu am Ende vom ÜB9

# Erfahrungen mit dem ÜB8 1/2

---

## ■ Zusammenfassung / Auszüge

Stand 26. Juni 19:00

- Die freiere Aufgabenstellung (bei Aufgabe 1) hat vielen gefallen und Spaß gemacht
- Zahlreiche Fragen zum Umgang mit Sonderzeichen
  - Haben wir extra nicht entfernt, weil man die in jedem größeren Datensatz hat und also damit umgehen mussAusführliche Hilfestellung dazu auf dem Forum
- Einige haben trotz Warnung die cities.txt mit hochgeladen!
- Gar nicht so einfach, keinen Roman zu schreiben

# Erfahrungen mit dem ÜB8 2/2

---

## ■ Datenstruktur Aufgabe 1, zwei gute Optionen

- **Option 1:** balancierter Suchbaum (wer hätte das gedacht)

Pro Anfrage braucht man einen lookup (Zeit  $O(\log n)$ ) und eine Folge von next Operationen (Zeit je  $O(1)$ )

Der Aufbau des Baumes geht in Zeit  $O(n \cdot \log n)$  und wenn man ausnutzt, dass Eingabe schon sortiert, sogar in Zeit  $O(n)$

- **Option 2:** sortiert abspeichern + binäre Suche / Anfrage

Man kann die Eingabe auch erstmal ganz einlesen und dann einmal sortieren (oder ausnutzen, dass bereits sortiert)

Pro Anfrage reichen dann zwei binäre Suchen → Zeit  $O(\log n)$

Option 2 sogar besser, weil effizienter beim Einlesen und einfacher (man braucht keine komplexe Datenstruktur)

## ■ Definition

- Eine Prioritätswarteschlange (PW) verwaltet eine Menge von Key-Value Paaren bzw. Elementen und es gibt wieder eine Ordnung  $\leq$  auf den Keys
- Es werden folgende Operationen unterstützt:
  - `insert(item)`: füge das gegebene Element ein
  - `getMin()`: gebe das Element mit dem kleinsten Key zurück
  - `deleteMin()`: entferne das Element mit dem kleinsten Key
  - `changeKey(item)`: ändere Key des gegebenen Elementes
  - `remove(item)`: entferne das gegebene Element

## ■ Vergleich mit `HashMap` und `BinarySearchTree`

- Bei der `HashMap` sind die Keys in keiner besonderen Ordnung abgespeichert

Von daher würden uns `getMin` und `deleteMin` dort  $\Theta(n)$  Zeit kosten, wobei  $n$  = Anzahl Schlüssel

- Der `BinarySearchTree` kann alles was eine `PriorityQueue` kann und **mehr** (nämlich `lookup` von beliebigen `Elem.`)

Wir werden sehen, dass dafür die `PriorityQueue`, für das was sie kann und macht, effizienter ist

Und tatsächlich gibt es viele Anwendungen, wo eine `PriorityQueue` ausreicht ... siehe Folien 8 – 11

## ■ Mehrere Elemente mit dem gleichen Key

- Das ist für viele PW-Anwendungen nötig und darf man deswegen nicht einfach ausschließen
- Man muss dann nur klären, welches Element `getMin` und `deleteMin` auswählen, wenn es mehrere kleinste Keys gibt
- Das übliche Vorgehen ist so

`getMin` : gibt irgendein Element mit kleinstem Key zurück

`deleteMin` : löscht eben dieses Element

Bei unserer Implementierung gleich wird das quasi "von selber" der Fall sein

- Argument der Operationen `changeKey` und `remove`
  - Eine PW erlaubt **keinen** Zugriff auf ein beliebiges Element
  - Deshalb geben `insert` und `getMin` eine Referenz auf das entsprechende Element zurück
  - Mit so einer Referenz kann man dann später über `changeKey` bzw. `remove` den Schlüssel ändern bzw. das Element entfernen

## ■ Anwendungsbeispiel 1

- Man kann mit einer PW einfach **sortieren**, und zwar so:

Alle Elemente einfügen:  $\text{insert}(x_1), \text{insert}(x_2), \dots, \text{insert}(x_n)$

Dann wieder rausholen, immer das kleinste was noch da ist:  $\text{getMin}(), \text{deleteMin}(), \text{getMin}(), \text{deleteMin}(), \dots$

Der entsprechende Algorithmus heißt **HeapSort**

- Wir sehen später: alle Operationen gehen in  $O(\log n)$  Zeit

Damit läuft **HeapSort** in  $O(n \cdot \log n)$  Zeit

Also asymptotisch optimal für vergleichsbasiertes Sortieren

Insbesondere genauso gut wie **MergeSort** (im allgemeinen Fall) und **QuickSort** (im besten Fall)

and noch nicht  
sortiert

## ■ Anwendungsbeispiel 2

- Mischen von  $k$  sortierten Listen ... englisch: k-way merge
- Dazu  $k$  "Zeiger", auf jede Liste einen ... in jeder Iteration das kleinste von den betreffenden Elementen berechnen
- Das geht mit einer PW in Zeit  $O(\log k)$  pro Iteration, also insgesamt Zeit  $O(n \cdot \log k)$  ...  $n$  = Gesamtzahl Elemente

$L_1 : 5, 7, 15, 23 \times$

$L_2 : 8, 9, 10 \times$

$L_3 : 11, 17, 19, 25 \times$

$R : 5, 7, 8, 9, 10, 11, 15, \dots$

RESULT

## ■ Anwendungsbeispiel 3

- Die PW ist die grundlegende Datenstruktur bei **Dijkstra's Algorithmus** zur Berechnung kürzester Wege

Das machen wir nächste Woche !

## ■ Anwendungsbeispiel 4

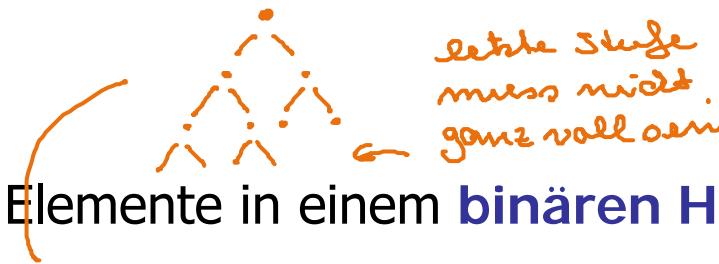
- Man kann mit einer PW einfach und effizient die k größten Elemente von einer Menge berechnen

Das ist Aufgabe 1 vom Ü9

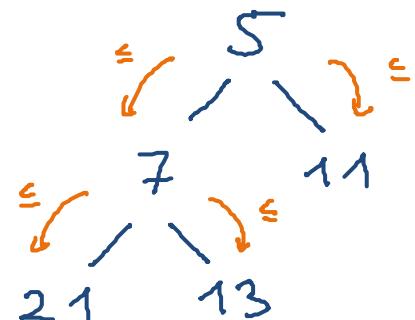
Allgemeiner geht das auch dann noch effizient, wenn sich die Menge laufend ändert, und zwar in Zeit  $O(\log k)$  pro Element, das dazukommt bzw. weggenommen wird

## ■ Grundidee

- Wir speichern die Elemente in einem **binären Heap**
- Das ist ein **vollständiger binärer Baum**, für dessen Schlüssel die **Heap-Eigenschaft (HE)** gilt
- HE = Key jedes Knotens  $\leq$  die Keys von beiden Kindern



Das ist eine **schwächere** Eigenschaft als beim binären Suchbaum, insbesondere sind die Blätter **nicht** sortiert



Jetzt male dir und  
in Folgenden nur  
die Keys dir,  
nicht die Values  
(die sterben einfach  
immer nur mit dabei.)

# PW – Implementierung 2/15

~~Dann ist die Formel einfacher  
um zu wiederholen / Eltern zu drucken~~

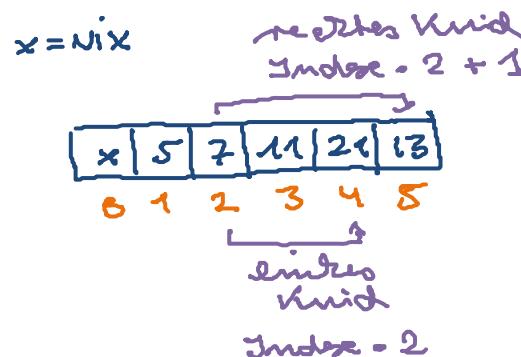
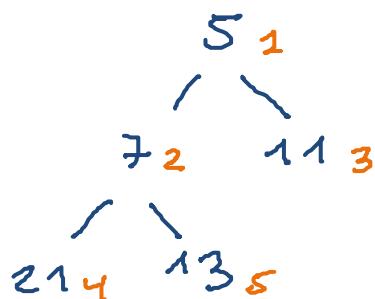
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
...

## ■ Wie speichert man einen binären Heap

- Anders als beim `BinarySearchTree` geht das **ohne Zeiger**
- Wir nummerieren die Knoten von oben nach unten und von links nach rechts durch, beginnend mit **1**
- Wir können die Elemente dann in einem Feld speichern, und leicht zu Kinder- bzw. Elternknoten springen:

Die Kinder von Knoten  $i$  sind Knoten  $2i$  und  $2i + 1$

Der Elternknoten von einem Knoten  $i$  ist Knoten  $\lfloor i/2 \rfloor$

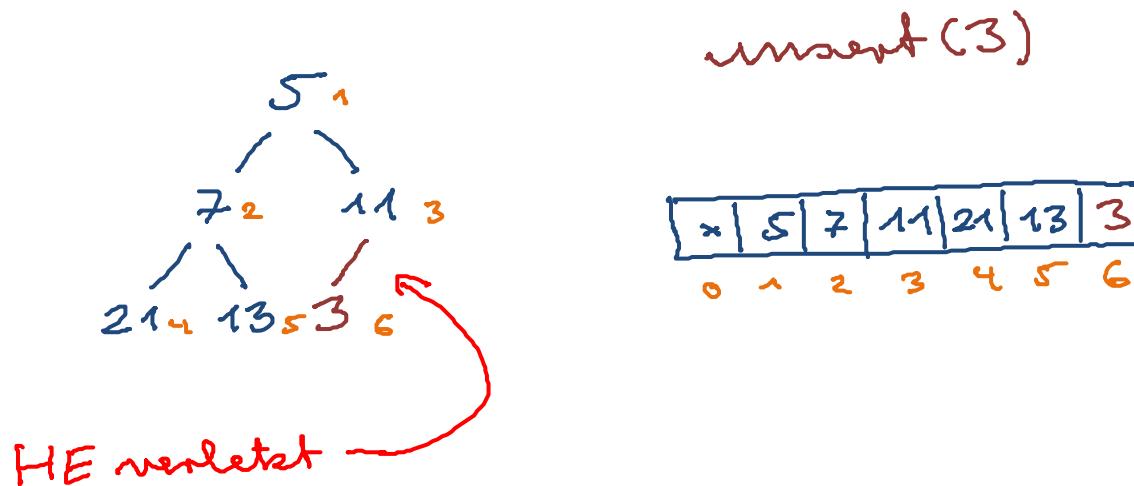


## ■ Die Operation insert

- Erstmal hinzufügen am Ende des Feldes
- Danach kann die Heapeigenschaft verletzt sein

Aber nur genau an dieser (letzten) Position

Wiederherstellung der HE siehe Folien 20 und 21



## ■ Die Operation `getMin`

- Einfach das oberste Element zurückgeben

Im Feld ist das einfach das Element an Position 1

Falls Heap leer, einfach null zurückgeben

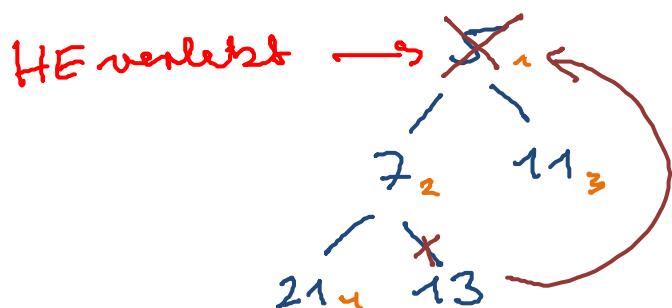
## ■ Die Operation `deleteMin`

- Einfach das Element von der letzten Position an die erste Stelle setzen (falls Heap nicht leer)

Element an der ersten Stelle wird dabei überschrieben

- Danach kann die Heapeigenschaft (HE) verletzt sein  
Aber wieder nur genau an dieser (ersten) Position

Wiederherstellung der HE siehe Folie 21



`deleteMin()`

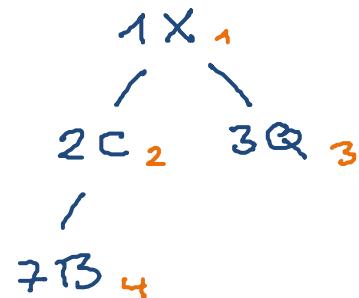
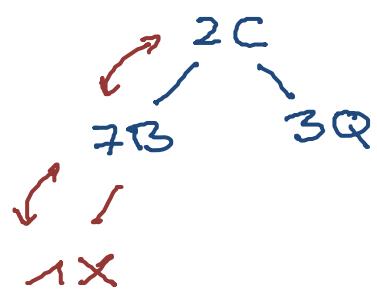
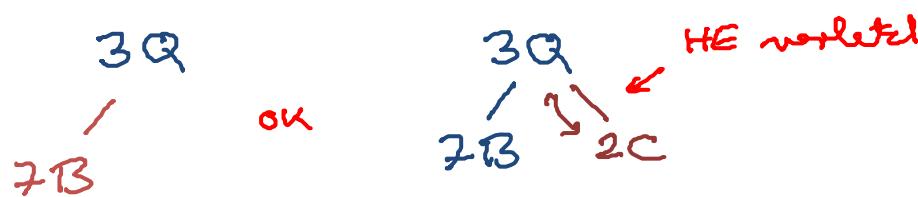
|   |    |   |    |    |
|---|----|---|----|----|
| x | 13 | 7 | 11 | 21 |
| 0 | 1  | 2 | 3  | 4  |

unsent (3Q)

unsend (7B)

unsend (2C)

unsend (1X)



|   |    |    |    |    |
|---|----|----|----|----|
| x | 1X | 2C | 3Q | 7B |
| 0 | 1  | 2  | 3  | 4  |

## ■ Die Operation `changeKey`

- Das Element wird als Argument übergeben ... wir können also einfach seinen Schlüssel ändern
- Danach kann die Heapeigenschaft (HE) verletzt sein

Aber wieder nur genau an dieser Position

Wiederherstellung der HE siehe Folien 20 und 21

Element muss dazu seine Position im Feld wissen

Siehe dazu Folie 22

## ■ Die Operation `remove`

- Das Element wird als Argument übergeben
- Element von der letzten Position an diese Stelle setzen
- Danach kann die Heapeigenschaft (HE) verletzt sein

Aber wieder nur genau an dieser Position

Element muss dazu wieder seine Position im Feld wissen

Siehe Folien 20 und 21

*zuerst wie selektiv*

## ■ Reparieren der Heapeigenschaft

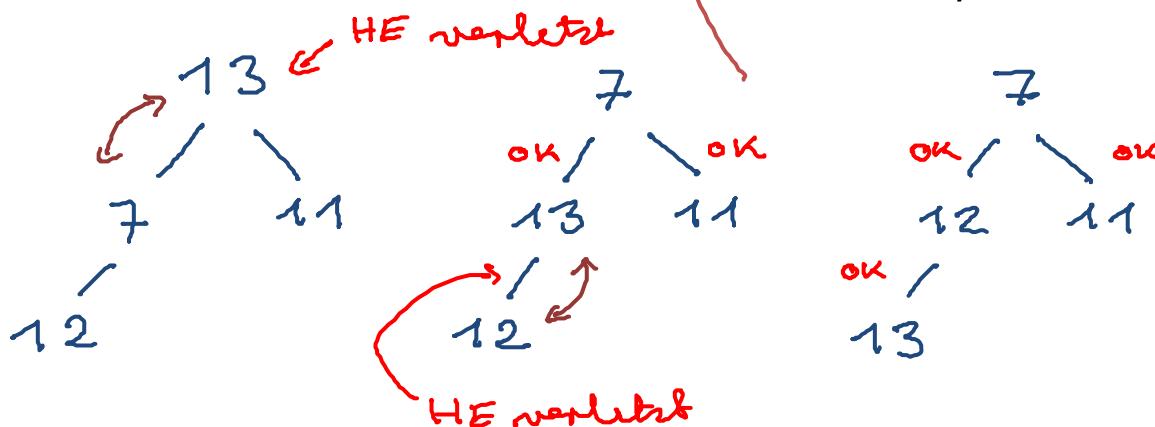
- Nach `insert`, `deleteMin`, `changeKey`, `remove` kann die Heapeigenschaft (HE) verletzt sein
  - Aber nur an genau einer (bekannten) Position i
- Die HE kann auf eine von zwei Arten verletzt sein
  - Schlüssel an Position i ist nicht  $\leq$  der seiner Kinder
  - Schlüssel an Position i ist nicht  $\geq$  der vom Elternkn.
- Entsprechend brauchen wir zwei Reperaturmethoden
  - `repairHeapDownwards` und `repairHeapUpwards`

## ■ Methode repairHeapDownwards

- Knoten **x** mit dem Kind **y** tauschen, das den kleineren Key von den beiden Kindern hat

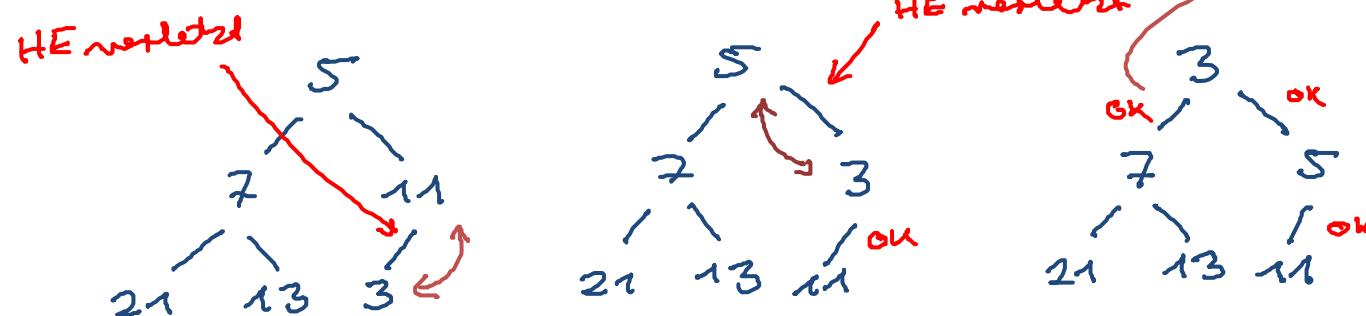
Zum Elternknoten hin stimmt dann mit Sicherheit alles

- Jetzt ist bei diesem Kind eventuell die **HE** verletzt, indem sein Key größer ist als der von einem der beiden Kinder
- In dem Fall einfach da dasselbe nochmal, usw.



## ■ Methode repairHeapUpwards

- Knoten  $x$  mit dem Elternknoten  $y$  tauschen
- Zum Kindknoten hin stimmt dann mit Sicherheit alles
- Jetzt ist bei dem Elternknoten eventuell die HE verletzt
- In dem Fall einfach da dasselbe nochmal, usw.



## ■ Index eines Elementes in der PW

sc@3 ↔ ix@7  
ix@3 sc@7

- **Achtung:** für `changeKey` und `remove` muss ein Element wissen, wo es im Heap steht
- Lösung: jedes Element hat eine zusätzliche Variable `heapIndex`, die angibt, wo es im internen Feld steht
- Wann immer das Element im Feld verschoben wird, darauf achten, `heapIndex` entsprechend anzupassen

Element wird nur innerhalb von `repairHeapDownwards` und `repairHeapUpwards` verschoben

## ■ Laufzeit

- Die Laufzeit von `repairHeapDownwards` bzw. `Upwards` ist  $O(d)$ , wobei  $d$  die Tiefe des (vollst. binären) Baumes ist
- Aus Vorlesung 8a wissen wir:  $d = O(\log n)$
- Die Operationen `insert`, `deleteMin`, `changeKey` und `remove` laufen also in  $O(\log n)$  Zeit

Nach jeder davon muss die HE wiederhergestellt werden

Es geht auch noch schneller, siehe Vorlesung 9b morgen

- Die Operation `getMin` läuft sogar in  $O(1)$  Zeit
- Das Minimum ist einfach das oberste Element im Heap

- Benutzung in Java    import java.util.PriorityQueue;
  - Element-Typ unterscheidet nicht zwischen Key und Value
  - PriorityQueue<T> pq;
  - Defaultmäßig wird die Ordnung  $\leq$  auf T genommen
  - Eigene Ordnung über einen Comparator, wie bei sort
  - Operationen: insert = add, getMin = peek, deleteMin = poll
  - Die Operation changeKey gibt es nicht, dafür remove
  - Mit remove und insert kann man changeKey leicht simulieren

- Benutzung in C++
    - #include <queue>;
    - Element-Typ unterscheidet nicht zwischen Key und Value
    - std::priority\_queue<T> pq;
    - Es wird die Ordnung  $\geq$  auf T genommen, und nicht  $\leq$
    - Beliebige Vergleichsfunktion wie bei std::sort
    - Operationen: insert = push, getMin = top, deleteMin = pop
    - Es gibt kein changeKey und auch kein beliebiges remove
- Eine Implementierung ohne diese beiden ist effizienter, weil man den heapIndex pro Element nicht braucht
- In Vorlesung 10a sehen wir, wie man ohne changeKey zurechtkommt, wenn man es eigentlich doch braucht

## ■ Benutzung in Python from queue import PriorityQueue

- Elemente sind beliebige Tupel

```
 pq = PriorityQueue()
 pq.put((priority, value))
```

- Es wird die Ordnung auf den Tupeln genommen
- Operationen: `insert` = `put`, `deleteMin` = `get`
- Statt `getMin` kann man `pq.queue[0]` aufrufen
  - queue ist das interne Feld, aber Beginn bei 0
- Die Operationen `changeKey` und `remove` gibt es nicht
  - Workaround siehe Bemerkung auf der Folie vorher

# Literatur / Links

---

## ■ Prioritätswarteschlangen

- In Mehlhorn/Sanders:
  - 6 Priority Queues [einfache und fortgeschrittenere Varianten]
- In Wikipedia
  - <http://de.wikipedia.org/wiki/Vorrangwarteschlange>
  - [http://en.wikipedia.org/wiki/Priority\\_queue](http://en.wikipedia.org/wiki/Priority_queue)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 9b, Mittwoch, 28. Juni 2017  
(Bucket Queues)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Inhalt

- Prioritätswürgeschlangen Fortsetzung von gestern
  - Fibonacci Heaps nur kurz zur Komplexität
  - Bucket Queues effizienter als Heaps für einen wichtigen Spezialfall
  - ÜB9, Aufgabe 2: Implementierung einer BucketQueue

Sie brauchen dazu u.a. eine `LinkedList`

Für Python haben wir eine Implementierung auf dem Wiki bereitgestellt, für Java können Sie die "java.util.LinkedList" benutzen, für C++ die "std::list"

# Evolution des Homo Sapiens

---

- Was war die entscheidende Mutation und wann?
  - Ein spannendes Thema!
  - Das möchte ich jetzt nicht hier in der Voraufzeichnung (ohne Publikum) abhandeln
  - Deswegen auf nächste Woche (Mittwoch) verschoben

# Fibonacci Heaps



## ■ Grundidee

- Ein "Wald" von (nicht mehr unbedingt vollständigen) binären Bäumen, die im Verlauf ineinander gehängt werden

## ■ Laufzeit

getMin in Zeit  $O(1)$  ... wie beim binary heap

insert in Zeit  $O(1)$  ... binary heap  $O(\log n)$

decreaseKey in amortisierter Zeit  $O(1)$  ... bin. heap  $O(\log n)$

deleteMin in amortisierter Zeit  $O(\log n)$  ... bin. heap  $O(\log n)$

In der Praxis ist der binäre Heap aufgrund seiner Einfachheit und guten Lokalität (Feld) aber schwer zu schlagen

Selbst für  $n = 2^{20} \approx 1.000.000$  ist  $\log_2 n$  ja nur 20

# Bucket Queues 1/14

## ■ Monotone ganzzahlige Prioritätswarteschlangen

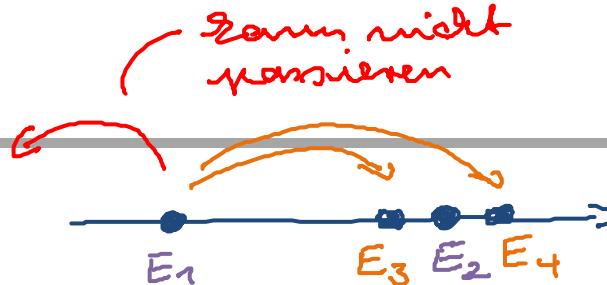
- Eine Folge von Operationen auf einer PW heißt **monoton ganzzahlig**, wenn gilt:

Die Keys sind ganze Zahlen aus dem Bereich  $0 \dots M - 1$

Das Minimum der gespeicherten Elemente wird durch neue Operationen **nie kleiner**, höchstens größer

Man beachte: die Argumente der Operationen müssen dazu nicht unbedingt monoton steigen

- Positivbeispiel:  $\text{ins}(7), \text{ins}(18), \text{ins}(12), \text{delMin}(), \text{ins}(14), \dots$
- Negativbeispiel:  $\text{ins}(7), \text{ins}(18), \text{ins}(12), \text{delMin}(), \text{ins}(10), \dots$



## ■ Anwendungen dafür

- Simulationen in der Zeit

Es gibt Events zu diskreten (ganzzahligen) Zeitpunkten

Ein Event kann neue Events **in der Zukunft** generieren

Die Events will man chronologisch abarbeiten

- Dijkstra's Algorithmus (zur Berechnung kürzester Wege)

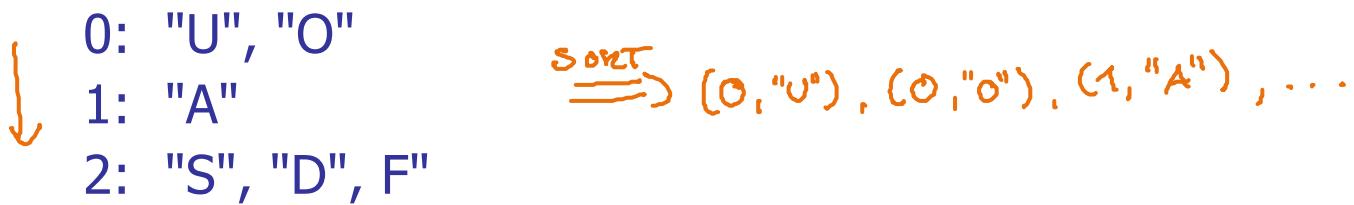
Man beginnt an einem Startort A

Man besucht von dort alle anderen Orte in der Reihenfolge **aufsteigender Kosten** (Zeit oder Entfernung)

Mehr dazu nächste Woche ...

## ■ Grundidee

- Ähnlich wie beim ganzzahligen Sortieren mit vielen gleichen Schlüsseln aus einem kleinen Bereich  $0..M-1$ , zum Beispiel  
 $(2, "S") (1, "A") (0, "U") (2, "D") (0, "O") (2, "F")$
- Da können wir mit einem Feld der Größe  $M$  alle Elemente mit demselben Schlüssel zusammen gruppieren



- Genau so fasst auch die Bucket Queue alle Elemente mit demselben Key zusammen, in sogenannten Buckets

# Bucket Queues 4/14

## ■ Datenstruktur

- Ein Feld von verketteten Listen bietet sich an, dann:
  - Zugriff auf Bucket für Schlüssel  $x$  in Zeit  $O(1)$
  - Einfügen in / Löschen aus Bucket in  $O(1)$  Zeit
- Außerdem merken wir uns immer ein Element **minItem** mit dem aktuell minimalen Key (es kann mehrere geben)



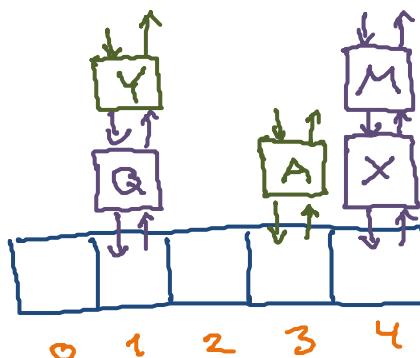
## ■ Die Operation `insert`

- Einfach neues Element an entsprechende Liste anhängen

Geht mit einer verketteten Liste pro Eintrag in Zeit  $O(1)$

- `minItem` muss nicht neu gesetzt werden, weil monoton

Außer natürlich beim Einfügen des allerersten Elementes

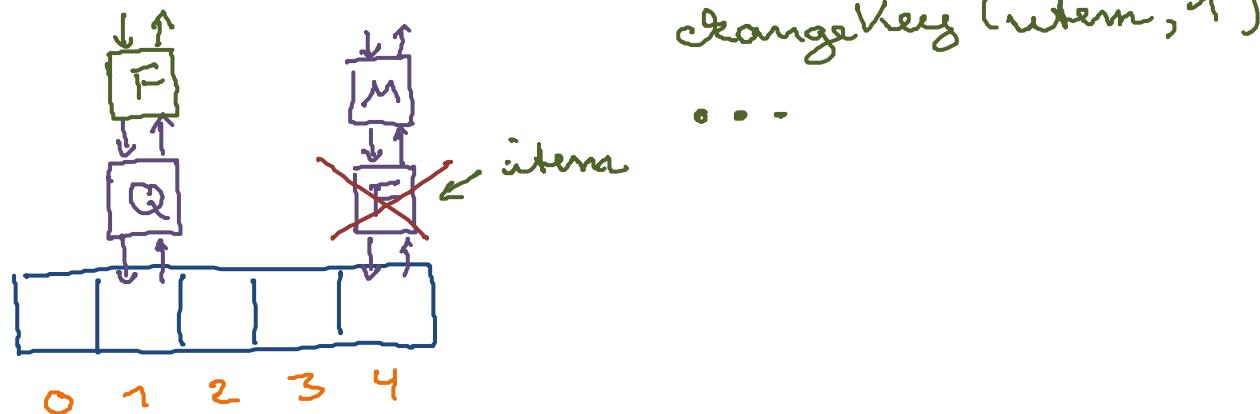


insert (3, A)  
insert (1, Y)  
...

# Bucket Queues 6/14

## ■ Die Operation `changeKey`

- Aus der alten Liste löschen und in die neue einfügen  
Geht mit einer verketteten Liste ebenfalls in  $O(1)$  Zeit
- `minItem` muss nicht neu gesetzt werden, weil monoton



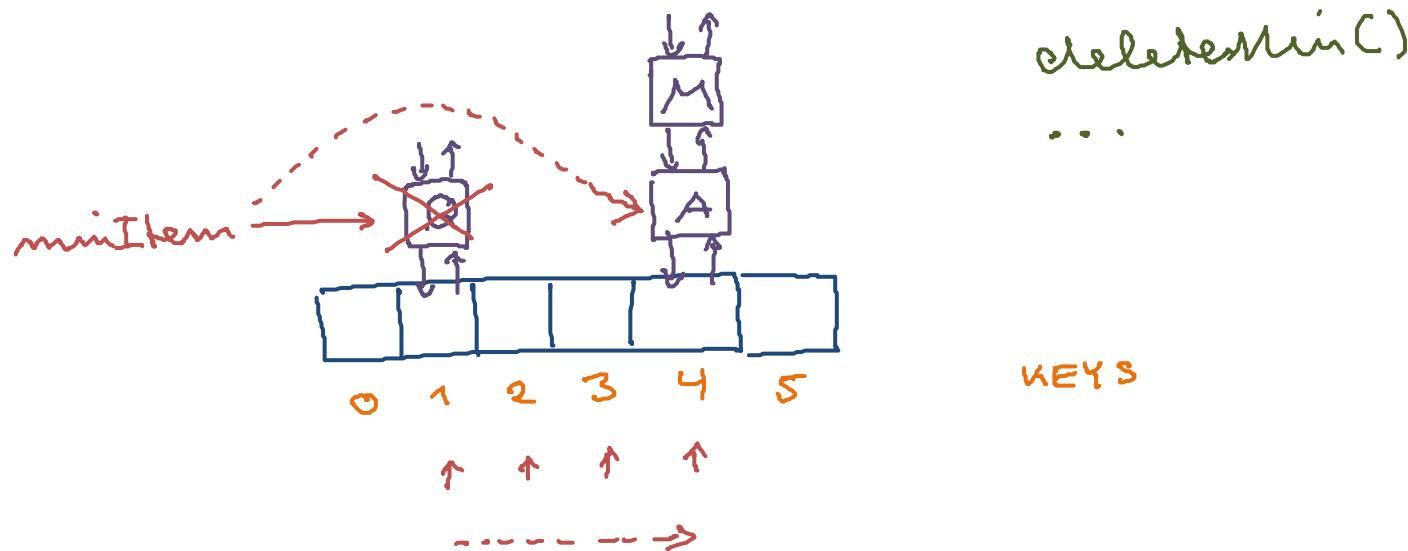
## ■ Die Operation `getMin`

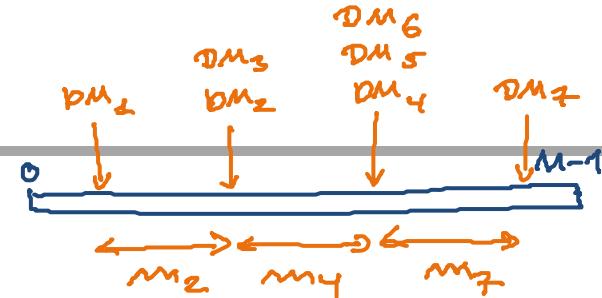
- Wir geben einfach das `minItem` zurück, das merken wir uns ja zu jedem Zeitpunkt explizit

Geht offensichtlich in  $O(1)$  Zeit

## ■ Die Operation `deleteMin`

- Das `minItem` aus seinem Bucket löschen
- Falls dieser Bucket jetzt leer, so weit im Feld nach rechts gehen, bis man die nächste nicht-leere Liste findet, und dort ein neues `minItem` wählen





## ■ Laufzeitanalyse deleteMin

- Ein einzelnes deleteMin kann bis zu  $\Theta(M)$  dauern
- Aber: sei  $m_i$  die Anzahl Schleifendurchläufe für das  $i$ -te deleteMin, dann ist  $\sum m_i = O(M)$

Man geht immer nur weiter nach rechts in dem Feld, nie nach links, und das Feld ist nur  $M$  groß

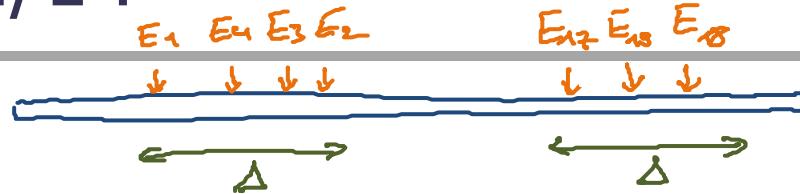
Ohne die Monotonie müsste man bei jedem deleteMin im worst case das ganze Feld durchgehen, um das neue Minimum zu finden



## ■ Laufzeit insgesamt

- Mit einer Bucket Queue lässt sich also eine beliebige Folge von  $n$  Operationen in Zeit  $O(n + M)$  bearbeiten

Für  $M = O(n)$  ist das durchschnittlich  $O(1)$  pro Operation



## ■ Platzverbrauch

- Die beschriebene Variante braucht  $\Theta(n + M)$  Platz
- Oft ist es in Anwendungen so, dass zu jedem Zeitpunkt die in der PW gespeicherten Schlüssel um maximal  $\Delta < M$  auseinander liegen, also in einem Intervall  
 $[minItem.key .. minItem.key + \Delta - 1]$
- Dann geht es auch mit  $\Theta(n + \Delta)$  Platz ... und  $O(n \cdot \Delta)$  Zeit

Das ist die **Zusatzaufgabe** zu Ü9.2

Das geht sehr elegant mit wenig zusätzlichem Code, aber etwas tricky ... Hilfestellung dazu auf der nächsten Folie

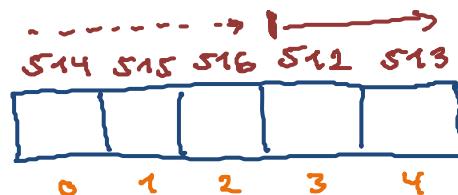
## ■ Hilfestellung zur Zusatzaufgabe

- Zu jedem Zeitpunkt muss man aber nur Keys aus dem Bereich  $[\text{minItem.key} .. \text{minItem.key} + \Delta - 1]$  speichern

Dafür reicht eigentlich ein Feld der Größe  $\Delta$  ... allerdings verändert sich  $\text{minItem.key}$  im Laufe der Zeit

- **Idee:** man muss das erste Element ja nicht unbedingt an Position 0 abspeichern, sondern kann an einer beliebigen Position i anfangen

Am Ende des Feldes dann wieder vorne anfangen



Möchte abspeichern:

$$s12 - s16$$

$$\Delta = 5$$

- Verbesserung für  $M \gg n$  zum Beispiel  $M = \Theta(n^2)$ 
  - Dann ist die Laufzeit der Bucket Queue  $\Theta(n^2)$   
*Also schlechter als der gewöhnliche binäre Heap*
  - Problem dabei: man hat dann ein großes Feld **buckets** der Größe  $M \gg n$  und die meisten Einträge sind leer  
*Es kann ja maximal  $n$  nicht-leere Einträge geben*
  - **Idee:** viele Einträge zu einem zusammenfassen, so dass man lange Folgen von nicht-leeren Einträgen einfach überspringen kann  
*Die resultierende Datenstruktur nenn man **Radix Heaps***

## ■ Laufzeit von Radix Heaps

- Falls die Keys aus dem Bereich  $[0 .. M - 1]$  sind, dann:

Bucket Queues:  $O(n + M)$

Radix Heaps:  $O(n \cdot \log M)$

- Falls die gespeicherten Keys zu jedem Zeitpunkt in  $[minItem.key .. minItem.key + \Delta - 1]$  liegen, dann:

Bucket Queues:  $O(n \cdot \Delta)$

Radix Heaps:  $O(n \cdot \log \Delta)$

# Literatur / Links

---

- Monotone ganzzahlige Prioritätswarteschlangen
  - In Mehlhorn/Sanders:
    - 10.5 Monotone Integer Priority Queues
    - 10.5.1 Bucket Queues

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 10a, Dienstag, 4. Juli 2017  
(Graphen, Exploration, Zusammenhang)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches



## ■ Inhalt

- Graphen Terminologie
  - Breitensuche Algorithmus + Beispiel + Code
  - Tiefensuche Algorithmus + Beispiel
  - Zusammenhangskomponenten Algorithmus + Beispiel + Code
  - ÜB10: Routenplanung in Baden-Württemberg

# Offizielle Evaluation der Veranstaltung

---

## ■ Läuft über das zentrale EvaSys der Uni

- Sie sollten gestern (Montag, 3. Juli) eine Mail vom System bekommen haben

**Falls nicht, bitte umgehend bei Axel Lehmann melden !**

- Nehmen Sie sich bitte Zeit und füllen Sie den Bogen **sorgfältig und gewissenhaft** aus

Sie haben soviel Zeit in die Vorlesung investiert, dann können Sie auch 30 Minuten für die Evaluation aufwenden

Sie bekommen außerdem 20 Punkte dafür, die die Punkte vom schlechtesten ÜB ersetzen ... siehe ÜB10, Aufgabe 1

- Uns interessieren besonders die **Freitextkommentare**

## ■ Zusammenfassung / Auszüge

- Aufgabe 1 hat vielen gefallen (Nachdenken + wenig Code)
- Ungläublich, dass Aufgabe 1 in  $O(n \cdot \log k)$  gehen soll  
Siehe Lösungsskizze nächste Folie
- Aufgabe 2 war auch gut machbar
- Coden in C++ relativ aufwändig
- Es sollte "im Wiki/Forum" heißen, nicht "auf dem Wiki/Forum"

## ■ Lösungsskizze + Programm Aufgabe 1

- Idee: zu jeden Zeitpunkt höchstens  $k$  Elemente in der PW
- Dadurch Laufzeit  $O(\log k)$  / Operation, insgesamt  $O(n \log k)$
- Falls schon  $k$  Elemente in der PW sind, das neue Element  $x$  mit dem aktuellen **Min** in der PW vergleichen
  - Falls  $x > \text{Min}$ : `deleteMin` und `insert(x)`; sonst: nix tun
- Am Ende die  $k$  Elemente eins nach dem anderen mit `deleteMin` aus der PW holen, in Zeit  $O(k \log k)$

Damit bekommt man sie in absteigender Reihenfolge, aber die kann man ja leicht umdrehen

## ■ Live-Vorlesung vs. Online-Vorlesung

- Wer eh nur Aufzeichnungen schaut: **kein Unterschied**  
*Das war mit Abstand der häufigste Kommentar*
- Einige hören die Vorlesung trotzdem lieber live, allerdings mit relativ "weichen" Argumenten  
*Termin zu dem man hin muss, weniger Ablenkung*
- Aufzeichnung hat objektive Vorteile: man kann anhalten, zurückspulen, vorrspulen, Geschwindigkeit erhöhen, etc.  
*Ideales Tempo variiert stark zwischen Teilnehmern*
- Wichtig: Inhalt aktuell + die VL lebendig + das **Forum**
- Gleiches T-Shirt wie SS 2015, aber schlechtere Beleuchtung :-)

- Kaum Fragen in den letzten Wochen?
  - Die meisten Fragen kommen bei der Bearbeitung des ÜB
  - Bei Verständnisschwierigkeiten oft nicht leicht, sich auf die schnelle eine gute Frage zu überlegen + sich nicht trauen
  - Außerdem häufiges Feedback: die Erklärungen seien so gut
- Fazit (aus Sicht der Dozentin)
  - Qualität von Aufzeichnungen / VL / Forum stark gelobt
  - Vorteile der Live-Vorlesung eher "nice to have"
  - Vorteile der Video-Aufzeichnungen sind objektiver
  - Live-Vorlesung würde mehr Sinn machen, **nachdem** man sich mit dem Stoff / einer Aufgabe auseinandergesetzt hat

# Graphen 1/5

## ■ Definition:

- Ein Graph  $G$  besteht aus zwei Mengen  $V$  und  $E$   
 $V$  = Menge der **Knoten** ... engl. "nodes" oder "vertices"  
 $E$  = Menge der **Kanten** ... engl. "edges" oder "arcs"
- Eine Kante  $e$  verbindet jeweils zwei Knoten  $u$  und  $v$   
ungerichtete Kante:  $e = \{u, v\}$   $\stackrel{= \{v, u\}}{\text{Menge}}$   
gerichtete Kante:  $e = (u, v)$   $\stackrel{\neq (v, u)}{\text{Tupel}}$
- Bei einem **gewichteten** Graph hat man für jede Kante ein Gewicht, auch Länge oder Kosten der Kante genannt  
**Für ÜB10: die Reisezeit zwischen zwei Punkten**

# Graphen 2/5

$$\begin{matrix} m=6, m=0 \\ \textcircled{0} \quad \textcircled{0} \\ \textcircled{0} \quad \textcircled{0} \\ \textcircled{0} \quad \textcircled{0} \end{matrix}$$

zusammenhängendes  
Grp. :  
 $m \geq m-1$   
 $\textcircled{0}-\textcircled{0}-\textcircled{0}-\textcircled{0}$

generell  
 $m \leq m^2$   
falls keine  
"Multi"-Kanten

## ■ Repräsentation gerichteter Graph

- Adjazenzmatrix ... Platzverbrauch  $\Theta(|V|^2)$

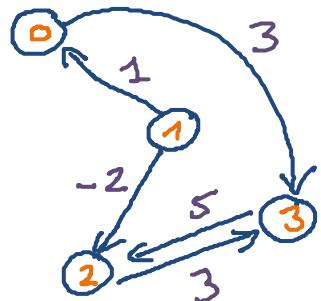
- Adjazenzlisten ... Platzverbrauch  $\Theta(|V| + |E|)$

4 Knoten

5 Kanten

mit Knotennummern  
und Kanten gewichten

ADJ. MATRIX



|   | 0 | 1 | 2  | 3 |
|---|---|---|----|---|
| 0 | x | x | x  | 3 |
| 1 | 1 | x | -2 | x |
| 2 | x | x | x  | 3 |
| 3 | x | x | 5  | x |

$\times$  = keine Kante  
Anzahl nicht- $\times$  Einträge  
= Anzahl Kanten

Für jeden Knoten  
die Liste aller von  
ihm ausgehenden  
Kanten  
ADJ. LISTEN

|     |                                                                                 |   |    |   |    |
|-----|---------------------------------------------------------------------------------|---|----|---|----|
| 0 : | <table border="1"> <tr> <td>3</td><td>3</td></tr> </table>                      | 3 | 3  |   |    |
| 3   | 3                                                                               |   |    |   |    |
| 1 : | <table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>-2</td></tr> </table> | 0 | 1  | 2 | -2 |
| 0   | 1                                                                               | 2 | -2 |   |    |
| 2 : | <table border="1"> <tr> <td>3</td><td>3</td></tr> </table>                      | 3 | 3  |   |    |
| 3   | 3                                                                               |   |    |   |    |
| 3 : | <table border="1"> <tr> <td>2</td><td>5</td></tr> </table>                      | 2 | 5  |   |    |
| 2   | 5                                                                               |   |    |   |    |

sonde Einträge  
wie es Knoten  
gibt

## ■ Repräsentation ungerichteter Graph

- Einen ungerichteten Graphen kann man einfach als gerichteten Graphen darstellen, bei dem es jede Kante in beide Richtungen gibt
- Falls es Kantenkosten gibt, sind die Kosten dann in beiden Richtungen gleich

So machen wir es auch für unseren Code nachher

# Graphen 4/5

## ■ Grad, Eingangsgrad, Ausgangsgrad

- Grad eines Knotens  $u$  in einem ungerichteten Graph

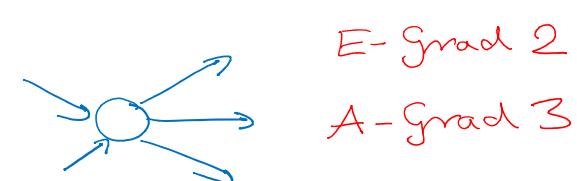
$$\text{degree}(u) = |\{u, v\} : \{u, v\} \in E|$$



- Eingangs- und Ausgangsgrad eines Knotens  $u$  in einem gerichteten Graph

$$\text{in-degree}(u) = |(v, u) : (v, u) \in E|$$

$$\text{out-degree}(u) = |(u, v) : (u, v) \in E|$$



# Graphen 5/5

## ■ Pfade

- Ein Pfad in  $G$  ist eine Folge  $u_1, u_2, u_3, \dots, u_l \in V$  mit
  - ( $u_1, u_2$ ), ( $u_2, u_3$ ), ..., ( $u_{l-1}, u_l$ )  $\in E$  gerichteter Graph
  - { $u_1, u_2$ }, { $u_2, u_3$ }, ..., { $u_{l-1}, u_l$ }  $\in E$  ungerichteter Graph
- Die **Länge** bzw. **Kosten** eines Pfades
  - ohne Kantengewichte: Anzahl der Kanten
  - mit Kantengewichten: Summe der Gewichte auf dem Pfad
- Der **kürzeste Pfad** (engl. *shortest path*) zwischen zwei Knoten  $u$  und  $v$  ist der Pfad  $u, \dots, v$  mit minimalen Kosten  
**Dazu Beispiele und mehr in der VL10b (Dijkstra Algorithmus)**

## ■ Informale Definition

- Gegeben ein Startknoten  $s$ , besuche "systematisch" alle Knoten von  $V$ , die von  $s$  aus erreichbar sind
- Breitensuche = in der Reihenfolge der "Entfernung" von  $s$   
englisch: **breadth first search = BFS**
- Tiefensuche = erstmal "möglichst weit weg" von  $s$   
englisch: **depth first search = DFS**
- Das ist kein relevantes "Problem" an sich, taucht aber oft als Teil / Subroutine von anderen Algorithmen auf

Zum Beispiel zur Berechnung der Zusammenhangskomponenten eines Graphen, siehe Folien 20 – 22

# Graphexploration 2/7

---

## ■ Breitensuche, Idee

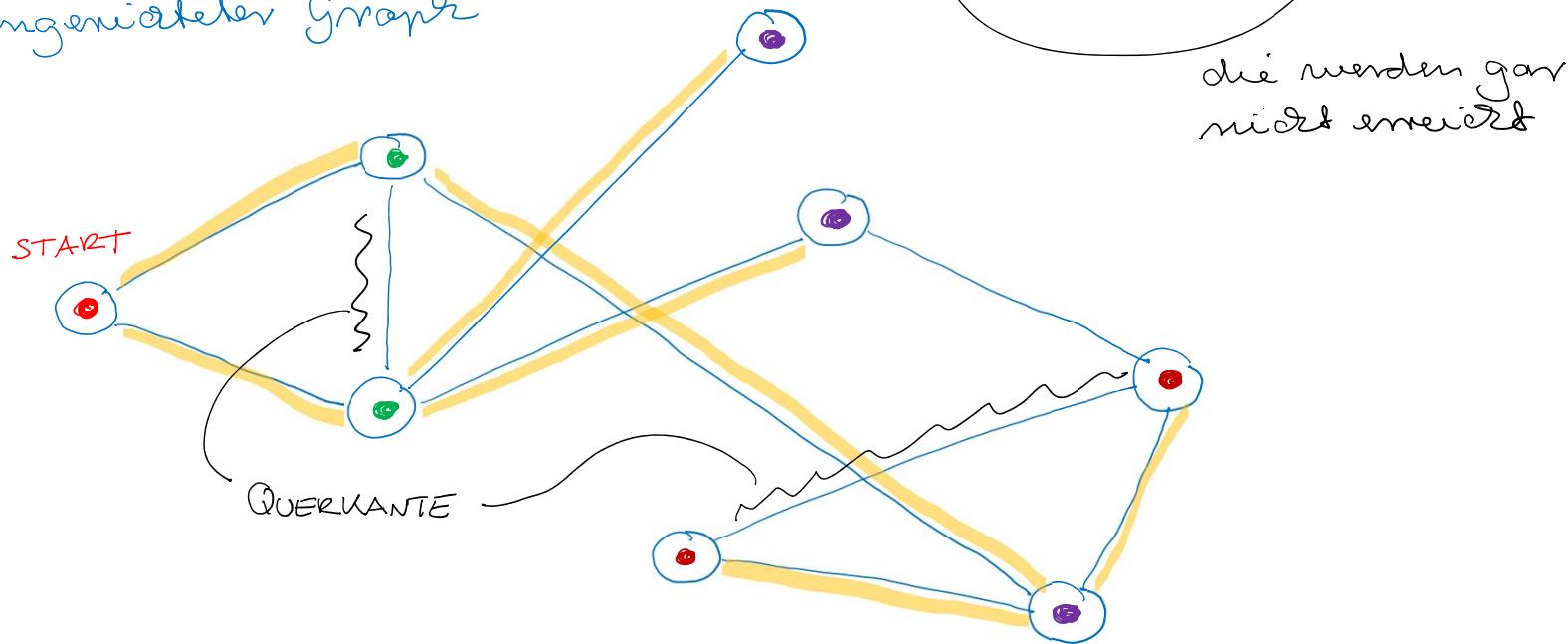
- Markierung für jeden Knoten, zu Beginn alle unmarkiert
- Beginne mit einem **Startknoten** und markiere ihn (**Level 0**)
- Finde alle Knoten die zum **Startknoten** benachbart und noch nicht markiert sind und markiere sie (**Level 1**)
- Finde alle Knoten, die zu einem **Level 1** Knoten benachbart und noch nicht markiert sind und markiere sie (**Level 2**)
- Usw. bis ein Level keine benachbarten Knoten mehr hat, die noch nicht markiert sind

Das markiert insbesondere alle Knoten, die in derselben Zusammenhangskomponente sind wie der Startknoten

# Graphexploration 3/7

## ■ Breitensuche, Beispiel

ungekennzeichneter Graph



LEVEL 0

LEVEL 1

LEVEL 2

LEVEL 3

Die gelben Kanten

- (vom einem Level zum nächsten, nur eine Kante für jeden Knoten vom nächsten Level)

bilden ein BAUM (einen sog. Spannbaum der ZK des Graphen)

## ■ Tiefensuche, Idee

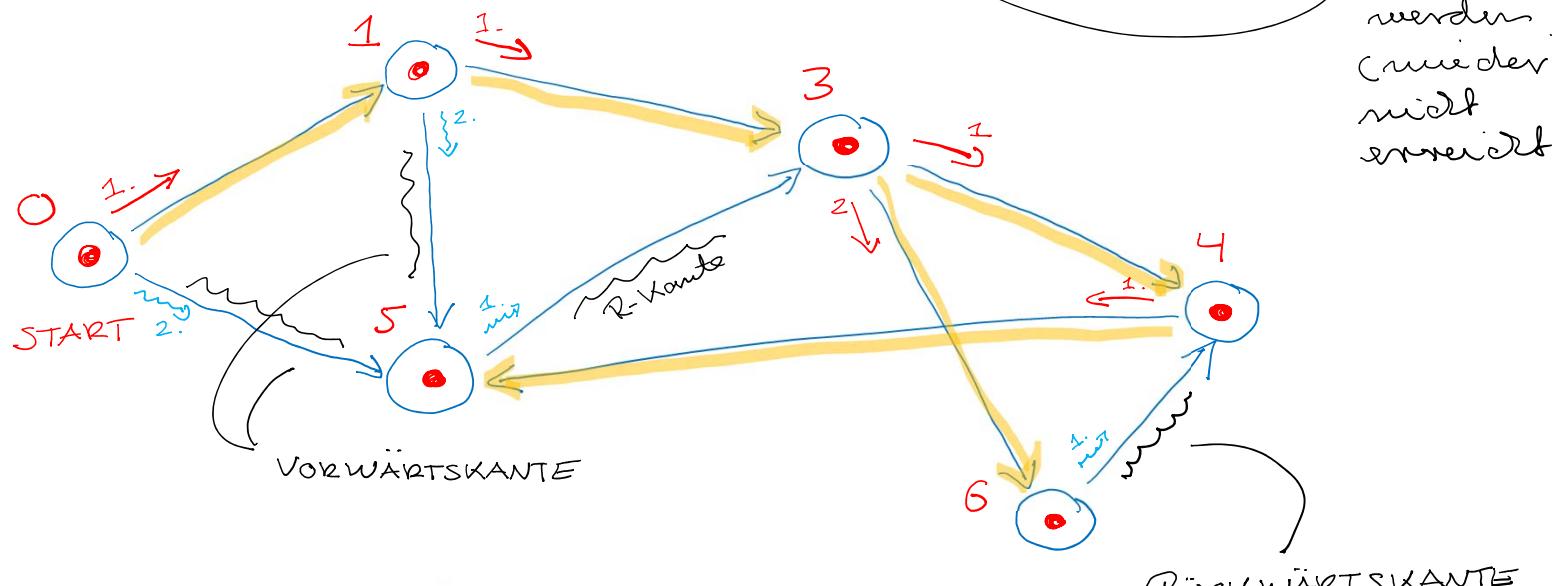
- Markierung für jeden Knoten, zu Beginn alle unmarkiert
- Beginne mit einem **Startknoten** und markiere ihn
- Gehe in irgendeiner Reihenfolge die zum Startknoten benachbarten Knoten durch und tue Folgendes:  
**Falls der Knoten noch nicht markiert ist, markiere ihn und starte **rekursiv** eine Tiefensuche von dort aus**
- Das sucht zuerst "in die Tiefe" (vom Startknoten aus)
- Auch **DFS** markiert schließlich alle Knoten, die in derselben Zusammenhangskomponenten liegen wie der Startknoten

# Graphexploration 5/7

UNI  
FREIBURG

## ■ Tiefensuche, Beispiel

gekennzeichneter Graph



Die gelben Pfeile bilden wieder einen Baum (den Spannbaum)

# Graphexploration 6/7

## ■ Tiefensuche, weitere Eigenschaft

- Auf **azyklischen** Graphen liefert Tiefensuche eine sogenannte **topologische Sortierung**

Das ist eine Nummerierung der Knoten, so dass jede Kante von einem Knoten mit kleinerer Nummer zu einem mit größerer Nummer geht

$\Leftrightarrow$  es gibt keine R-Kanten

Wohlgemerkt: mit einem **Zyklus** kann das nicht gehen

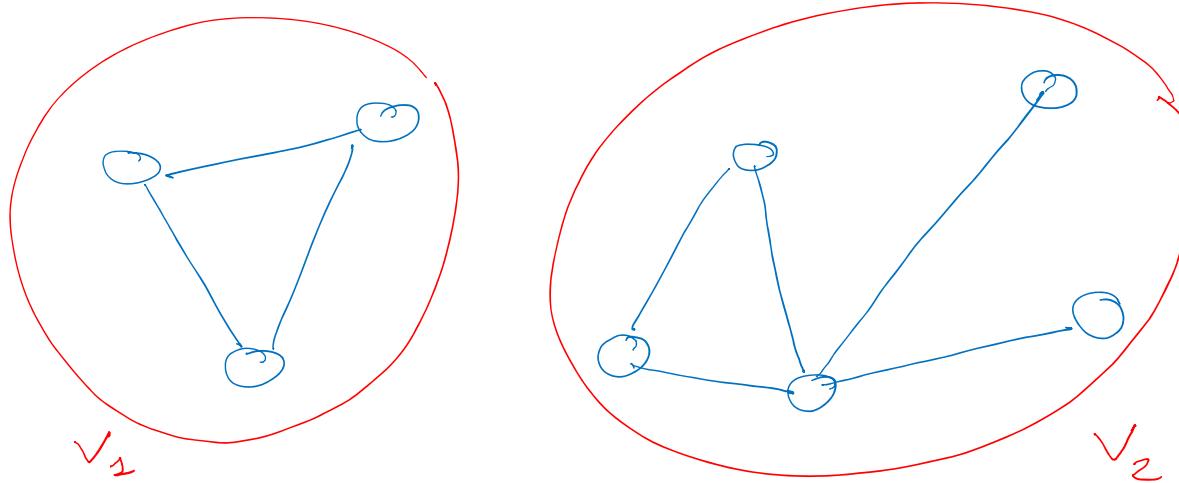
## ■ Komplexität von BFS und DFS

- Für beide Verfahren gilt:
  1. Man folgt jeder Kante genau einmal
  2. Man "verarbeitet" jeden Knoten genau einmal (in dem Sinne, das man seinen adjazenten Kanten folgt)
- Die Laufzeit ist also  $O(|V'| + |E'|)$ , wobei  $V'$  und  $E'$  die Anzahl Knoten und Kanten in der Zusammenhangskomponente sind, in der der Startknoten liegt
- Besser geht es offenbar nicht

Weil man jeden Knoten und jede Kante mindestens einmal "anschauen" muss

## ■ Für einen **ungerichteten** Graphen

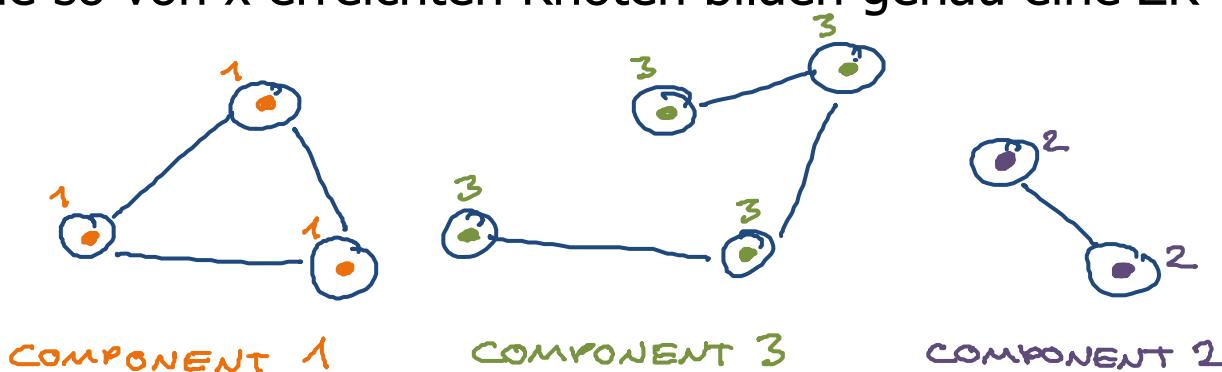
- Die Zusammenhangskomponenten (ZK) bilden eine Partition von  $V$ , also  $V = V_1 \cup \dots \cup V_k$
- Zwei Knoten  $u$  und  $v$  sind in derselben ZK genau dann, wenn es einen Pfad zwischen  $u$  und  $v$  gibt



# Zusammenhangskomponenten 2/3

## ■ Berechnung durch DFS oder BFS

- Markierung für jeden Knoten, zu Beginn alle unmarkiert
- Solange es noch einen unmarkierten Knoten  $x$  gibt:
  - Starte **DFS** oder **BFS** von  $x$  und finde alle von  $x$  aus erreichbaren Knoten und markiere sie
- Die Reihenfolge, in der die Knoten besucht werden ist hier egal, deswegen auch hier egal ob DFS oder BFS
- Die so von  $x$  erreichten Knoten bilden genau eine ZK



# Zusammenhangskomponenten 3/3

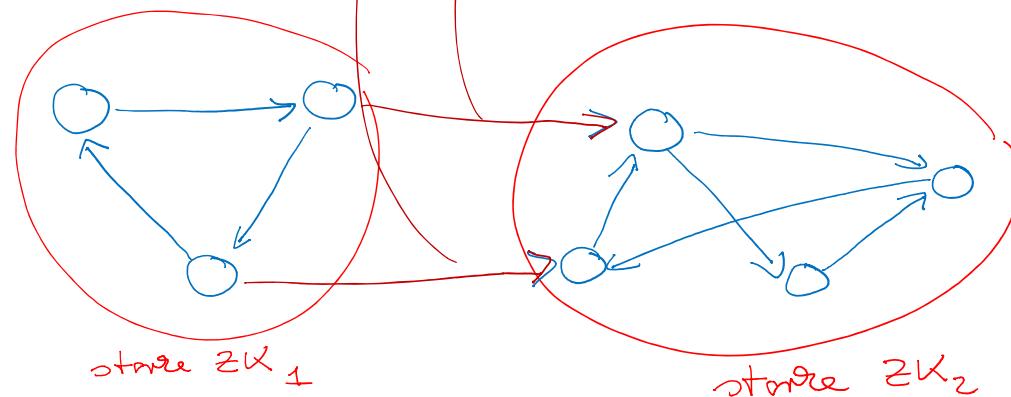
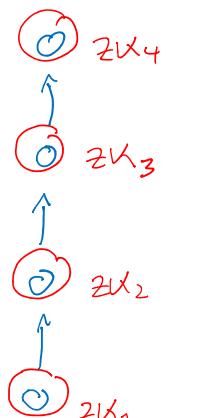
## ■ Definition für einen gerichteten Graphen

- Man spricht dann von starken Zus.komponenten (ZK)
- Eine starke ZK ist eine maximale Teilmenge von Knoten  $V'$ , so dass es für alle  $u, v \in V'$  eine Pfad von  $u$  nach  $v$  gibt

hats dieser  
beiden Knoten  
zwei starke ZK

Nicht mehr so intuitiv, wie bei ungerichteten Graphen

Der Algorithmus ist auch komplizierter und machen wir hier nicht ... geht aber sehr elegant mit Tiefensuche



# Literatur / Links

---

## ■ Graphen, Breitensuche, Tiefensuche, ZK

- In Mehlhorn/Sanders:

- 8 Graph Representation

- 9 Graph Traversal

- In Wikipedia

- [http://en.wikipedia.org/wiki/Graph \(mathematics\)](http://en.wikipedia.org/wiki/Graph_(mathematics))

- [http://en.wikipedia.org/wiki/Breadth-first search](http://en.wikipedia.org/wiki/Breadth-first_search)

- [http://en.wikipedia.org/wiki/Depth-first search](http://en.wikipedia.org/wiki/Depth-first_search)

- [http://en.wikipedia.org/wiki/Connected component](http://en.wikipedia.org/wiki/Connected_component)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 10b, Mittwoch, 5. Juli 2017  
(Dijkstras Algorithmus)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Drumherum

- ## – Evolution Homo Sapiens die entscheidende Mutation?

## ■ Inhalt

- Dijkstras Algorithmus Algorithmus + Beispiel
  - Korrektheitsbeweis endlich wieder Mathe :-)
  - Laufzeit + Implementierung Analyse + Tipps
  - ÜB10: Implementieren Sie Dijkstras Algorithmus zur (einfachen) Routenplanung auf Baden-Württemberg

## ■ Entscheidende Mutation? Ihre Kommentare

- Die Entstehung von Mehrzellern vor 2-3 Milliarden Jahren
- Aufrechter Gang + Hände frei, vor 5-7 Millionen Jahren
- Zunahme des Gehirnvolumen durch erhöhten Fleischkonsum
- Entdeckungsdrang, im Gegensatz zu den Neanderthalern
- Entwicklung zu sprechendem sozialen Wesen / von Kultur
- "Die Erfindung von TrapRap bei Lil Waynes Geburt"
- "Da ich Informatiker bin: die Erfindung der Pornografie"
- "Der Moment als die Frauen kompliziert wurden und die Männer sie nicht mehr verstanden"

# Evolution zum Homo Sapiens 2/5

---

## ■ Ein paar wichtige Stationen

|                            |                        |
|----------------------------|------------------------|
| – Sauerstoffkatastrophe    | ~ 2.4 Milliarden Jahre |
| – Erste Wirbeltiere        | ~ 525 Millionen Jahre  |
| – Übergang Wasser → Land   | ~ 400 Millionen Jahre  |
| – Perm-Trias-Massensterben | ~ 252 Millionen Jahre  |
| – Erste Säugetiere         | ~ 225 Millionen Jahre  |
| – Dinosaurier futsch       | ~ 65 Millionen Jahre   |
| – Erste Primaten           | ~ 50 Millionen Jahre   |
| – Aufrechter Gang          | ~ 4 Millionen Jahre    |
| – Homo Sapiens             | ~ 200 Tausend Jahre    |

## ■ Analogie: ein Menschenleben

- Schauen Sie sich an, wie Sie heute sind und aussehen
- Wann war der entscheidende Moment in Ihrem Leben der diesen Zustand hervorgebracht hat?

|                        |               |
|------------------------|---------------|
| Geburt                 | 0 Jahre       |
| Laufen                 | ~ 1.5 Jahre   |
| Sprechen               | ~ 2 Jahre     |
| Ich-Bewusstsein        | ~ 3 Jahre     |
| Erstes Smartphone      | ?             |
| Einsetzen der Pubertät | 12 – 50 Jahre |

## ■ Menschenleben Zeitraffer

- Es gibt zwar Zeiten, in denen in relativ kurzer Zeit relativ viel passiert
- Aber insgesamt ist es ein **fließender Übergang**

Es ändert sich in jedem Augenblick und diese kontinuierliche Änderung von Augenblick zu Augenblick kann über einen längeren Zeitraum beliebig viel verändern

[https://www.youtube.com/watch?v=iVEiAU\\_F2qw](https://www.youtube.com/watch?v=iVEiAU_F2qw)

## ■ Übergangsformen ("Transitional Forms")

- In der Evolution ist es tatsächlich ganz genauso
- Wenn man sich das ganze in Zeitraffer anschauen würde, sähe man eine kontinuierliche Veränderung, wie beim Altern
- Schlagender Beweis dafür sind Fossilien von Übergangsformen zwischen Lebewesen und ihren ganz andersartigen Ahnen, z.B.

Pakicetus (wolfsähnlich) → Wal

Reptilkiefer + Ohr → Säugetierkiefer + Ohr

Dinosaurier → Vögel

Gorilla → aufrechter Gang

# Dijkstras Algorithmus 1/4

---



## ■ Ursprung

- Benannt nach **Edsger Dijkstra** (1930 – 2002)  
Niederländischer Informatiker, einer der wenigen  
Europäer, die den Turing-Award gewonnen haben  
(für seine Arbeiten zur strukturierten Programmierung)
- Der Algorithmus ist aus dem Jahr **1959**

## ■ Grundidee und Terminologie

- Sei  $s$  der Startknoten und sei  $\text{dist}(s, u)$  die Länge des kürzesten Pfades von  $s$  nach  $u$ , für alle Knoten  $u$
- Besuche die Knoten in der Reihenfolge der  $\text{dist}(s, u)$
- Für jeden Knoten wird während der Ausführung eine vorläufige Distanz  $\text{dist}[u]$  gespeichert, zu Beginn  $\infty$
- Es gibt dann drei Arten von Knoten

**unerreicht:**  $\text{dist}[u] = \infty$

**aktiv:**  $\text{dist}[u] \geq \text{dist}(s, u)$  aber nicht  $\infty$

**gelöst:** siehe nächste Folie

Auf Englisch: unreached, active, settled

## ■ Algorithmus

- Zu Beginn nur  $s$  aktiv, mit  $\text{dist}[s] = 0$  und  $\text{dist}[v] = \infty$
- In jeder Runde holen wir uns den aktiven Knoten  $u$  mit dem kleinsten Wert für  $\text{dist}[u]$
- Den Knoten  $u$  betrachten wir dann als **gelöst**
- Für alle  $(u, v) \in E$ : prüfe ob  $\text{dist}[u] + \text{cost}(u, v) < \text{dist}[v]$  und falls ja, setze  $\text{dist}[v] = \text{dist}[u] + \text{cost}(u, v)$

Das nennt man **Relaxieren** von  $(u, v)$

- Wiederhole, bis es keine aktiven Knoten mehr gibt
- Alle gelösten Knoten kennen dann ihre Entfernung von  $s$
- Falls alle Kantenkosten 1 sind, ist das **genau** BFS

# Dijkstras Algorithmus 4/4

Kantenkosten in LILA

● START

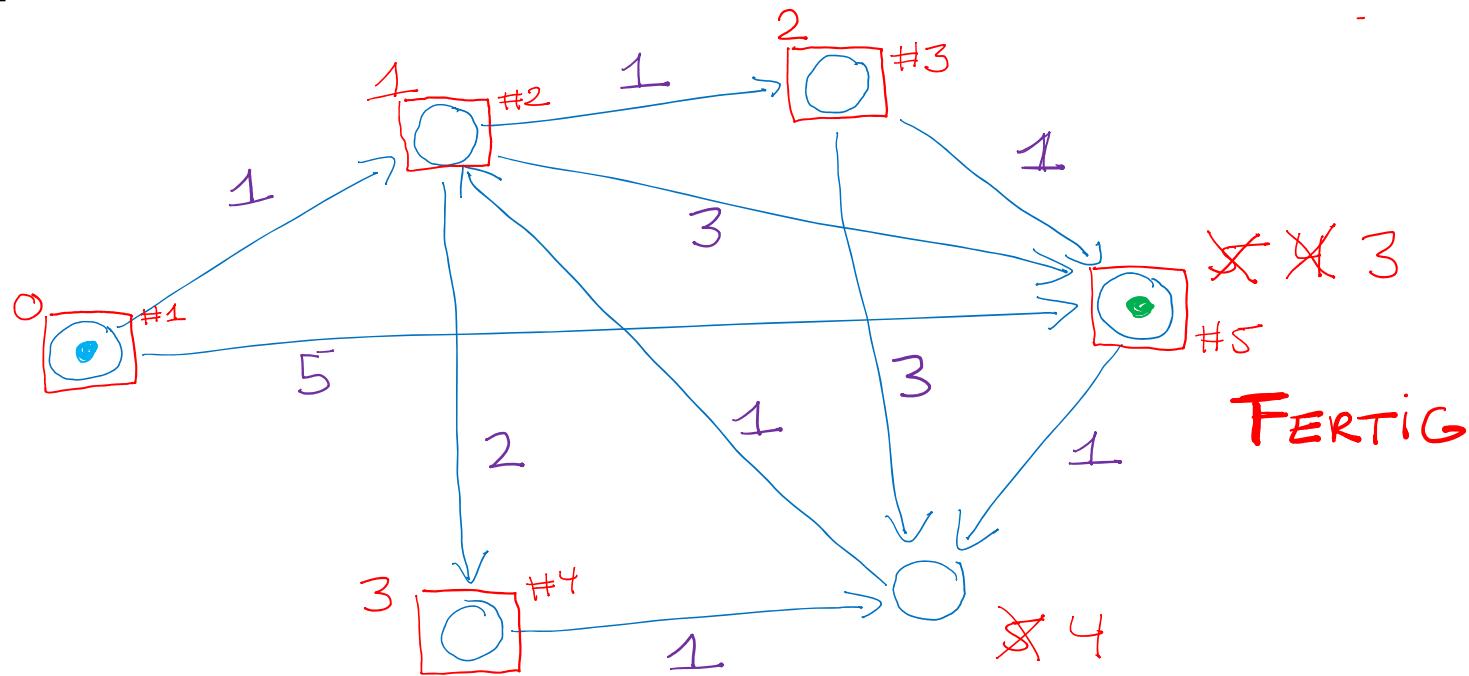
● ZIEL

UNI  
FREIBURG

$\#i$  gelöst in Schritt i

$\infty$  schreiben nur nicht dran

## ■ Beispiel



#4 wäre auch für den Ziernahen gegangen (und Kosten 3 → freie Wahl)

## ■ Annahmen

- **Annahme 1:** Alle Kantenlängen sind  $> 0$
- **Annahme 2:** Die  $\text{dist}(s, u)$  sind alle **verschieden**

Es gibt dann eine Anordnung  $u_1, u_2, u_3, \dots$  der Knoten

so dass gilt  $\text{dist}(s, u_1) < \text{dist}(s, u_2) < \text{dist}(s, u_3) < \dots$

- Es geht auch mit Kantenlängen  $\geq 0$  und ohne Annahme 2  
Beweis dazu siehe Referenzen (Mehlhorn/Sanders)

Mit den Annahmen ist der Beweis einfacher und intuitiver und enthält trotzdem alles Wesentliche

# Korrektheitsbeweis 2/6

## ■ Argumentationslinie

- Wir wollen zeigen, dass am Ende von Dijkstras Algorithmus  $\text{dist}[u_i] = \text{dist}(s, u_i)$  für jeden Knoten  $u_i$
- Im Folgenden zeigen wir, durch Induktion über  $i$ 
  - In der  $i$ -ten Runde gilt  $\text{dist}[u_i] = \text{dist}(s, u_i)$
  - In der  $i$ -ten Runde wird Knoten  $u_i$  gelöst

die orange Zahl am Knoten  $u_i$   
auf Folie 11

die Kosten des „kürzesten“ Weges

## ■ Induktionsanfang: $i = 1$

- In Runde 1 ist nur  $u_1 = s$  aktiv
- $\text{dist}[u_1] = 0 = \text{dist}(s, u_1)$
- $u_1$  wird als einziger aktiver Knoten gelöst

# Korrektheitsbeweis 4/6

## ■ Induktionsschritt: $i \rightarrow i + 1$ für $i \geq 1$

- Wir betrachten einen kürzesten Weg von  $s$  nach  $u_{i+1}$

Wir nehmen nicht an, dass unser Algorithmus diesen Weg kennt, aber wir können ihn im Beweis trotzdem betrachten

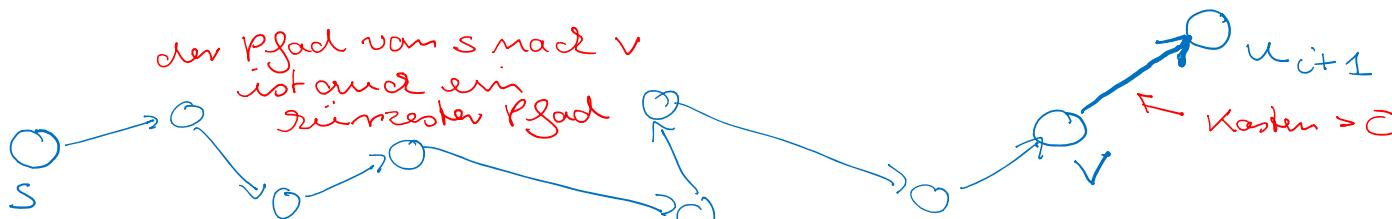
- Sei  $v$  der Knoten direkt vor  $u_{i+1}$  auf diesem Weg ... dann:

$$\text{dist}(s, u_{i+1}) = \text{dist}(s, v) + \underbrace{\text{cost}(v, u_{i+1})}_{>0} > \text{dist}(s, v)$$

Das benutzt Annahme 1: alle Kantenkosten sind positiv

- $v$  muss also einer von  $u_1, \dots, u_i$  sein (aber nicht unbedingt  $u_i$ )

Das benutzt Annahme 2:  $\text{dist}(s, u_1) < \text{dist}(s, u_2) < \dots$

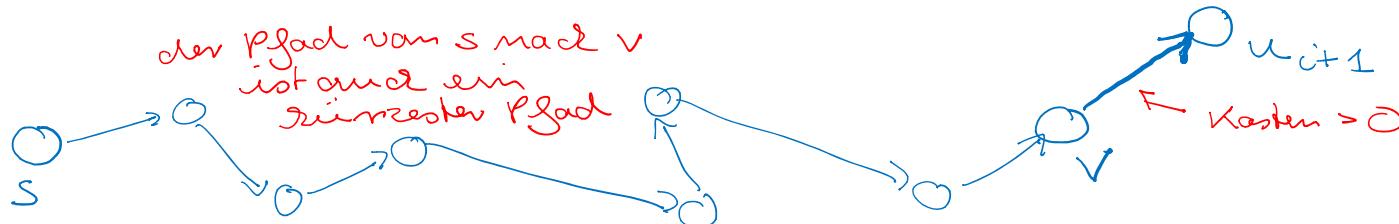


# Korrektheitsbeweis 5/6

## ■ Induktionsschritt: $i \rightarrow i + 1$ für $i \geq 1$ ... Fortsetzung

- Es ist also  $v = u_j$  wobei  $j \in 1 .. i$
- Nach Induktionsvoraussetzung gilt seit spätestens Runde  $j$   
 $\text{dist}[u_j] = \text{dist}(s, u_j)$
- In der Runde hat man dann, nach Relaxieren von  $(u_j, u_{i+1})$   
 $\text{dist}[u_{i+1}] = \text{dist}(s, u_j) + \text{cost}(u_j, u_{i+1}) = \text{dist}(s, u_{i+1})$

Das gilt schon seit Runde  $j$ , aber erst in Runde  $i + 1$  kann sich der Algorithmus sicher sein, dass es nicht besser geht



# Korrektheitsbeweis 6/6

weil die Zwischenzosten  
imms einem tatsächlichen  
Pfad entsprechen

UNI  
FREIBURG

## ■ Induktionsschritt: $i \rightarrow i + 1$ für $i \geq 1$ ... Fortsetzung 2

- Wir müssen noch zeigen, dass in Runde  $i + 1$  auch  $u_{i+1}$  gelöst wird, und nicht  $u_k$  mit  $k > i + 1$
- Aber für  $k > i + 1$  gilt nach Annahme 2 (Monotonie):  
 $\text{dist}[u_k] \geq \text{dist}(s, u_k) > \text{dist}(s, u_{i+1})$
- Also ist  $u_{i+1}$  in Runde  $i+1$  der aktive Knoten mit dem kleinsten  $\text{dist}$  Wert und wird also in der Runde gelöst

## ■ Grundprinzip

- Wir müssen die Menge der aktiven Knoten verwalten
  - Ganz am Anfang ist das nur der Startknoten
  - Am Anfang jeder Runde brauchen wir den aktiven Knoten  $u$  mit dem kleinsten Wert für  $\text{dist}[u]$
  - Es bietet sich also an, die aktiven Knoten in einer **Prioritätswürgeschlange** zu verwalten
- Mit Schlüssel  $\text{dist}[u]$  und Wert  $u$

## ■ Update von $\text{dist}[u]$

- Beobachtung: der **dist** Wert eines aktiven Knotens kann sich mehrmals ändern, bevor er schließlich gelöst wird

Wir müssen dann seinen Wert in der PW verkleinern, ohne dass wir den Knoten rausnehmen

- Genau dafür gibt es die Operation **changeKey**

Allerdings steht diese Operation nicht bei allen PWs zur Verfügung, z.B. bei der `std::priority_queue` von C++

# Implementierung 3/9

---

## ■ Implementierung ohne changeKey

- Statt `changeKey` macht man einfach ein `insert` mit dem neuen (niedrigeren) `dist` Wert

Den Eintrag mit dem alten Wert lässt man einfach drin

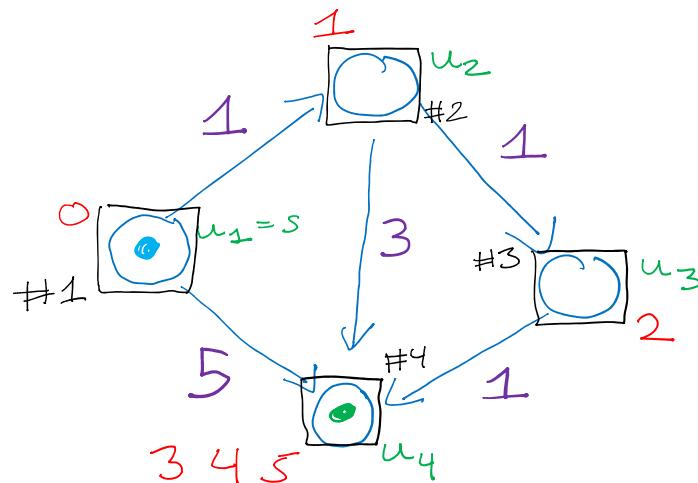
Bei gleichen oder höheren `dist` Wert macht man nichts
- Wenn der Knoten gelöst wird, dann mit dem niedrigsten Wert mit dem er in die PW eingefügt wurde
- Wenn man dann später nochmal auf den Knoten trifft, mit höherem `dist` Wert, nimmt man ihn einfach heraus und macht **nichts**

# Implementierung 4/9

$$\begin{aligned} \text{dist}(s, u_1) &= 0 \\ \text{dist}(s, u_2) &= 1 \\ \text{dist}(s, u_3) &= 2 \\ \text{dist}(s, u_4) &= 3 \end{aligned}$$

siehe  
ANNAHME 2.  
vom Beweis

## ■ Beispiel für Dijkstra mit PW ohne changeKey



ZUSTAND DER PW

|              |                             |    |
|--------------|-----------------------------|----|
| <del>0</del> | <del><math>u_1</math></del> | #1 |
| <del>1</del> | <del><math>u_2</math></del> | #2 |
| <del>5</del> | <del><math>u_4</math></del> | -  |
| <del>4</del> | <del><math>u_4</math></del> | -  |
| <del>2</del> | <del><math>u_3</math></del> | #3 |
| <del>3</del> | <del><math>u_4</math></del> | #4 |

IGNORIEREN  
IGNORIEREN

das ist leicht  
weil man ein  
Feld für die  
**dist** Werte hat

## ■ Berechnung der kürzesten Pfade

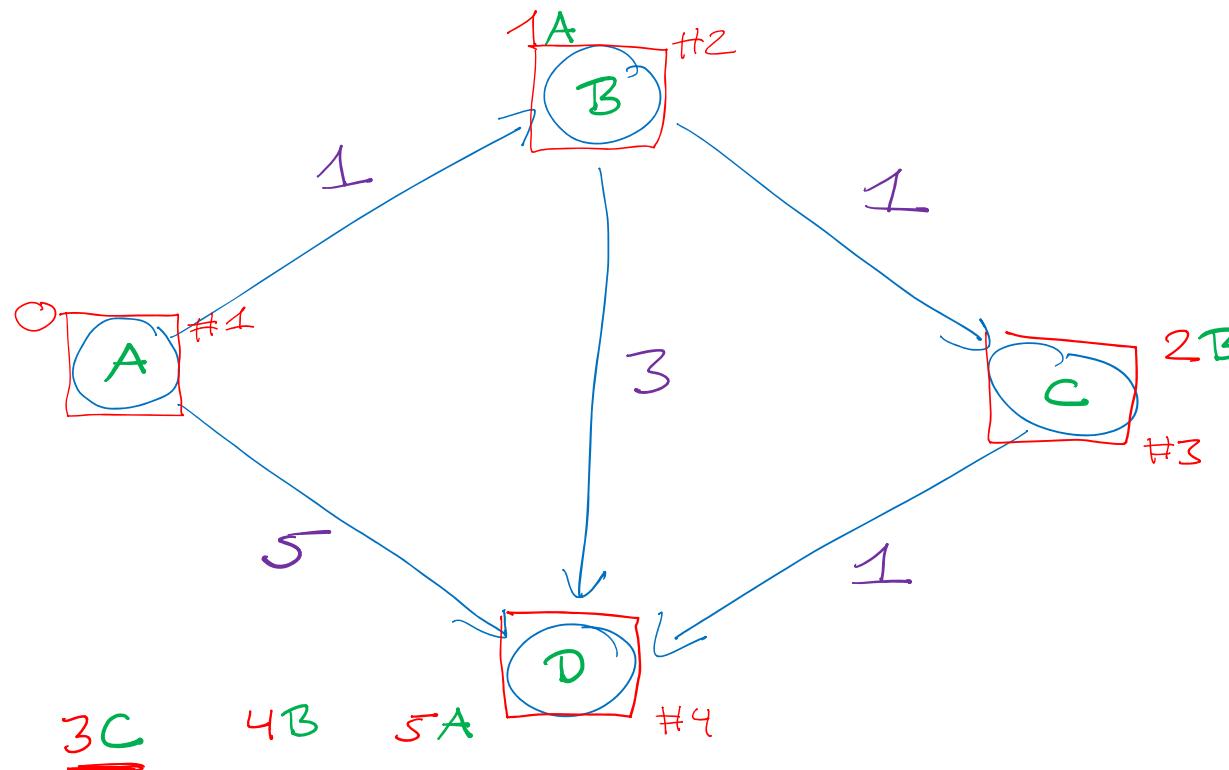
- So wie wir Dijkstras Algorithmus bisher beschrieben haben, berechnet er nur die **Länge** des kürzesten Weges
- Wenn man sich bei jeder **Relaxierung** den Vorgängerknoten auf dem aktuell kürzesten Pfad merkt, kriegt man aber auch leicht die tatsächlichen **Pfade**
- Es reicht für jeden Knoten ein Zeiger, weil jeder Präfix eines kürzesten Weges selber ein kürzester Weg ist
- Um den kürzesten Weg zu bekommen, kann man dann einfach die Zeiger bis zum Startknoten zurückverfolgen

# Implementierung 6/9

START = A  
ZIEL = D

UNI  
FREIBURG

## Berechnung der kürzesten Pfade, Beispiel

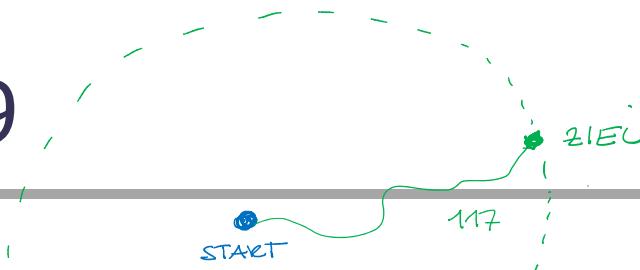


## ■ Visualisierung eines Pfades mit MapBBCode

- Für das **ÜB10** bekommen Sie einen Datensatz mit Geo-Koordinaten für jeden Knoten
- Man kann einen Pfad dann also auf einer Karte malen
- Das geht sehr einfach mit MapBBCode

<http://share.mapbbcode.org>

Ich mache das jetzt mal an einem einfachen Beispiel vor



## ■ Abbruchkriterium

- Sobald der Zielknoten  $t$  gelöst wird kann man aufhören  
**Aber nicht vorher, dann kann noch  $\text{dist}[t] > \text{dist}(s, t)$  sein**
- Bevor Dijkstras Algorithmus  $t$  erreicht, hat er die kürzesten Wege zu **allen** Knoten  $u$  mit  $\text{dist}(s, u) < \text{dist}(s, t)$  berechnet
- Das hört sich verschwenderisch an, es gibt aber für allgemeine Graphen keine (viel) bessere Methode

Grund: erst wenn man alles im Umkreis von  $\text{dist}(s, t)$  um den Startknoten  $s$  abgesucht hat, kann man sicher sein, dass es keinen kürzeren Weg zum Ziel  $t$  gibt

für das ÜB 10 Zähnen  
Sie auch Ihre dicke  
Bücher Quelle nehmen  
(missen Sie aber  
nicht)

## ■ Laufzeit dieser Implementierung

- Jeder der  $n$  Knoten wird genau **einmal** gelöst
- Genau dann werden seine ausgehenden Kanten betrachtet
- Jede der  $m$  Kanten führt also zu höchstens einem **insert**
- Die Anzahl der Operationen auf der PW ist also  $O(m)$
- Die Laufzeit von Dijkstras Algorithmus ist also  **$O(m \cdot \log n)$**
- Mit einer komplizierteren PW geht auch  **$O(m + n \cdot \log n)$**
- In der Praxis ist aber oft  $m = O(n)$

Dann ist die asymptotische Laufzeit für die kompliziertere PW nicht besser und man nimmt besser die einfachere PW

# Literatur / Links

---

## ■ Kürzeste Wege und Dijkstras Algorithmus

- In Mehlhorn/Sanders:

10 Shortest Paths

- In Wikipedia

[http://en.wikipedia.org/wiki/Shortest\\_path\\_problem](http://en.wikipedia.org/wiki/Shortest_path_problem)

[http://en.wikipedia.org/wiki/Dijkstra's\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra's_algorithm)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 11a, Dienstag, 11. Juli 2017  
(Editierdistanz, Teil 1)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches



## ■ Inhalt

- Editierdistanz Motivation + Notation
  - Rekursive Berechnung monotone Folgen
  - Rekursive Implementierung Code + Laufzeit
  - ÜB11: diesmal wieder ein reines Theorieblatt 😊

## ■ Zusammenfassung

- Schöne Aufgabe mit Praxisbezug, hat vielen Spaß gemacht
- Einige Kämpfe mit dem Design / Code / Debuggen

Extra weniger Vorgaben diesmal, ist ja schon das ÜB10

Es hat sich aber gezeigt, dass viele noch Probleme mit der Umsetzung eines abstrakten Algorithmus in Code haben

- Einigen ist offenbar nicht klar, wie Sie richtig testen sollen  
**Jede Funktion einzeln**, siehe 10 Gebote auf dem ÜB10  
Grundsätzlich auf kleinen Beispielen, das geht **immer**
- Einige haben etwas Zeit gebraucht, um zu verstehen, was der "längste kürzeste Weg" ist

## ■ Ergebnis

- Erklärung längster kürzester Pfad ab einem Startknoten **s**

Der Zielknoten **v**, für den  $\text{dist}(s, v)$  maximal ist

Wobei  $\text{dist}(s, v)$  der kürzeste Weg von **s** nach **v** ist

- Längster kürzester Weg ab der Freiburger TF in BaWü:

<http://share.mapbbcode.org/cusya> "Auf zur Kunigundenkapelle"

- Längster kürzester Weg ab der Informatik im Saarland:

<http://share.mapbbcode.org/psdbh> eine Kreuzung, laangweilig

# Editierdistanz 1/9

## ■ Motivation

- Es gibt viele Anwendungen, wo man ein Maß für die Ähnlichkeit zwischen zwei Zeichenketten braucht

Drei Beispiele dazu auf der nächsten Folie

- Die **Editierdistanz** ist ein solches Ähnlichkeitsmaß

Das in der Praxis am häufigsten verwendete Maß

Definition auf Folie 7

Benannt nach dem **Edi-Tier**



## ■ Anwendungsbeispiele

- Beispiel 1: Dubletten in Adressdatenbanken

Hein Blöd, 27568 Bremerhaven

Hein Bloed, 27568 Bremerhaven

Hein Doof, 27478 Cuxhaven

- Beispiel 2: Produktsuche

Memori Stik

- Beispiel 3: Websuche

eyjaföllajaküll

semesta verien 2017

# Editierdistanz 3/9

---

## ■ Definition Editierdistanz ... alternativ: Levenshtein-Distanz

- Gegeben zwei Zeichenketten  $x$  und  $y$
- $ED(x, y) = \text{die } \underline{\text{minimale}}$  Anzahl der folgenden Operationen, die man braucht, um  $x$  in  $y$  zu transformieren:

Einfügen (**insert**) eines Buchstabens

Ersetzen (**replace**) eines Buchstabens durch einen anderen

Löschen (**delete**) eines Buchstabens

- Die Position einer Operation ist die Stelle im String, an der etwas geändert wird ... siehe Beispiel nächste Folie

Positionen fangen hier und in der Folge mit **1** an, nicht 0  
(weil das intuitiver ist bei der mathematischen Analyse)

# Editierdistanz 4/9

DOOF  
SAUDOOF

UNI  
FREIBURG

## ■ Beispiel

- $x = \text{DOOF}$ ,  $y = \text{BLOED}$  ...  $\text{ED}(x, y) = ?$

$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \text{D} & \text{O} & \text{O} & \text{F} \end{array}$  } REPLACE(1, B)  
 $\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \text{B} & \text{O} & \text{O} & \text{F} \end{array}$  } REPLACE(2, L)  
 $\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \text{B} & \text{L} & \text{O} & \text{F} \end{array}$  } REPLACE(3, O)  
 $\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \text{B} & \text{L} & \text{O} & \text{E} \end{array}$  } INSERT(4, E)  
 $\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \text{B} & \text{L} & \text{O} & \text{E} \end{array}$  } REPLACE(5, D)  
 $\text{BLOED}$

Das ist eine  
monotone  
Folge (s. Folie 14)  
 $1 < 2 < 4 < 5$

Damit haben wir erstmal nur  
geeignet:  $\text{ED}(x, y) \leq 4$

## ■ Notation

- Mit  $\varepsilon$  bezeichnen wir das leere Wort
- Mit  $|x|$  bezeichnen wir die Länge von  $x$  (= Anzahl Zeichen)
- Mit  $x[i..j]$  bezeichnen wir die Teilfolge der Zeichen  $i$  bis  $j$  der Zeichenkette  $x$ , wobei  $1 \leq i \leq j \leq |x|$

Wie gesagt: Positionen / Indizes fangen mit **1** an, nicht 0

$$x = \begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ B & L & O & E & D \end{smallmatrix}, |x| = 5, x[2..4] = LOE$$

## ■ Ein paar einfache Eigenschaften

- $ED(x, y) = ED(y, x)$
- $ED(x, \varepsilon) = |x|$
- $ED(x, y) \geq \text{abs}(|x| - |y|)$   $\text{abs}(x) = x > 0 ? x : -x$
- $ED(x, y) \leq \max(|x|, |y|)$
- $ED(x, y) \leq ED(x[1..n-1], y[1..m-1]) + 1$   $n = |x|, m = |y|$
- Die Beweise sind alle einfache Zwei- oder Dreizeiler

Sehr gute Übung zur Klausurvorbereitung und zur Überwindung der Mathe-Phobie

# Editierdistanz 7/9

---

## ■ Lösungsidee 1: möglichst viel "erhalten"

–  $\text{ED}(\text{"VERIEN"}, \text{"FERIEN"}) = \underline{1}$

Einfach, weil die Zeichenketten größtenteils gleich

–  $\text{ED}(\text{"SEMESTERFERIEN"}, \text{"SEMESTERVERIEN"}) = \underline{1}$

Dito ... dann auch für längere Zeichenketten noch einfach

–  $\text{ED}(\text{"MEXIKO"}, \text{"AMERIKA"}) = \underline{3}$

Auch hier gibt es noch relativ große Übereinstimmung

–  $\text{ED}(\text{"AAEBEAABEAREEEAE"}, \text{"RBEAAEEBAAAEBBAE"}) = \underline{?}$

Spätestens hier wird es sehr schwierig mit dieser Idee

## ■ Lösungsidee 2: in zwei Hälften teilen

- In zwei gleich große Hälften teilen und die dann jeweils rekursiv lösen

$$\text{ED}(\text{GRAU}, \text{RAUM}) = 2$$

$$\text{ED}(\text{GR}, \text{RA}) = 2$$

$$\text{ED}(\text{AU}, \text{UM}) = 2$$

- **Keine gute Idee:** die ED zwischen den Hälften hat nicht viel zu tun mit der ED zwischen den ursprünglichen Zeichenketten

Formal: wenn  $x = x_1x_2$  und  $y = y_1y_2$ , dann ist im Allgemeinen **nicht**  $\text{ED}(x, y) = \text{ED}(x_1, y_1) + \text{ED}(x_2, y_2)$

## ■ Lösungsidee 3: alternative rekursive Formel

– Das Problem auf ein Problem "eins kleiner" zurückführen

–  $\text{ED}(\text{"FERIEN"}, \text{"VERIEN"}) = \text{ED}(\text{"FERIE"}, \text{"VERIE"}) = 1$

Weil die Worte rechts mit dem selben Buchstaben aufhören

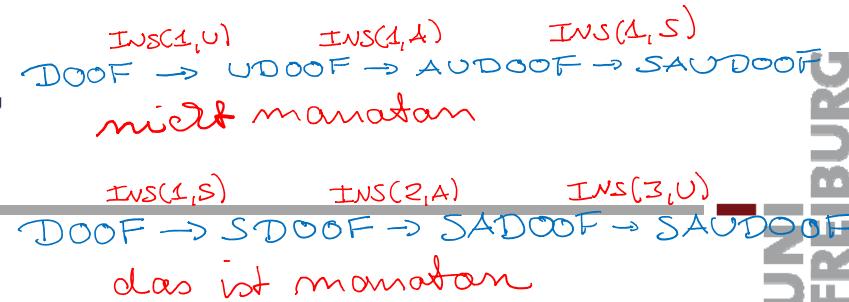
–  $\text{ED}(\text{"SAUM"}, \text{"RAUS"}) = \text{ED}(\text{"SAU"}, \text{"RAU"}) + 1 = 2$

Weil eine optimale Folge ein replace am Ende macht

Das gilt aber nicht immer, wenn sich die letzten beiden Buchstaben unterscheiden, zum Beispiel:

$2 = \text{ED}(\text{"RAUM"}, \text{"GRAU"}) \neq \text{ED}(\text{"RAU"}, \text{"GRA"}) + 1 = 3$

– Mit einer etwas komplizierteren Formel kriegt man es aber hin, das sehen wir jetzt auf den nächsten Folien



## ■ Monotonie

- Seien **x** und **y** unsere beiden Zeichenketten
- Seien  $\sigma_1, \dots, \sigma_k$  eine Folge von  $k = ED(x, y)$  Operationen, für  $x \rightarrow y$ , das heißt, um  $x$  in  $y$  zu überführen

Wir nehmen im Folgenden nicht an, dass wir die Folge schon kennen, sondern nur, dass es so eine gibt

- Eine Folge von Operationen heißt **monoton**, wenn die Position von  $\sigma_{i+1}$  ist  $\geq$  die Position von  $\sigma_i$ , wobei = nur dann erlaubt ist, wenn beides "delete" Operationen sind

Eine Folge von delete Operationen mit **gleichen** Positionen braucht man zum Beispiel bei  $ED("saudoof", "doof")$

$$SAU\bar{D}\bar{O}OF \xrightarrow{\text{DELETE}(1)} AU\bar{D}\bar{O}OF \xrightarrow{\text{DELETE}(1)} U\bar{D}\bar{O}OF \xrightarrow{\text{DELETE}(1)} DOOF$$

# Rekursive Formel 2/7

---

## ■ Hilfssatz zur Monotonie

- **Lemma:** Für beliebige  $x$  und  $y$  mit  $k = ED(x, y)$  gibt es eine monotone Folge von  $k$  Operationen für  $x \rightarrow y$
- Der Beweis ist Aufgabe 1 vom ÜB11
- Beweisidee 1: für den Fall  $k = 2$  (zwei Operationen), muss man sich im Prinzip nur alle neun Kombinationen der drei Arten von Operationen anschauen
- Beweisidee 2: eine allgemeine nicht-monotone Folge lässt sich durch "Nachbartranspositionen" immer in eine monotone Folge derselben Länge überführen

Nachbartransposition = zwei Operationen hintereinander in nicht-monotoner Reihenfolge werden "umgedreht"

# Rekursive Formel 3/7

## ■ Fallunterscheidung

*das ist also die, die am weitesten rechts was macht*

*einer monotonen Folge*

- Wir betrachten die letzte Operation  $\sigma_k$

$\sigma_1, \dots, \sigma_{k-1} : x \rightarrow z$  und  $\sigma_k : z \rightarrow y$

Seien  $n = |x|$  und  $m = |y|$  und  $m' = |z|$

Man beachte, dass  $m' \in \{m - 1, m, m + 1\}$

- **Fall 1:**  $\sigma_k$  macht etwas "ganz am Ende" von  $z$ :

Fall 1a:  $\sigma_k = \text{insert}(m' + 1, y[m])$  [dann ist  $m' = m - 1$ ]

Fall 1b:  $\sigma_k = \text{delete}(m')$  [dann ist  $m' = m + 1$ ]

Fall 1c:  $\sigma_k = \text{replace}(m', y[m])$  [dann ist  $m' = m$ ]

Wenn keines von den dreien der Fall ist, dann sind die letzten Zeichen von  $x$  und  $y$  (und  $z$ ) gleich ... siehe nächste Folie

BEISPIELE FÜR DIE FÄLLE

FALL 1A:  $\overset{x}{\text{GRAU}} \xrightarrow{\sigma_1} \overset{z}{\text{RAU}} \xrightarrow{\sigma_2} \overset{y}{\text{RAUM}}$

FALL 1B:  $\overset{x}{\text{BÄUME}} \xrightarrow{\sigma_1} \overset{z}{\text{BAUME}} \xrightarrow{\sigma_2} \overset{y}{\text{BAUM}}$

FALL 1C:  $\overset{x}{\text{DOOF}} \xrightarrow{\sigma_1, \sigma_2, \sigma_3} \overset{z}{\text{BLOEF}} \xrightarrow{\sigma_4} \overset{y}{\text{BLOED}}$

# Rekursive Formel 4/7

## ■ Fallunterscheidung

- Wir betrachten die letzte Operation  $\sigma_k$

$\sigma_1, \dots, \sigma_{k-1} : x \rightarrow z$  und  $\sigma_k : z \rightarrow y$

Seien  $n = |x|$  und  $m = |y|$  und  $m' = |z|$

Man beachte, dass  $m' \in \{m - 1, m, m + 1\}$

- **Fall 2:**  $\sigma_k$  macht nichts "ganz am Ende" von  $z$

Dann  $z[m'] = y[m]$  und  $x[n] = z[m']$

Damit  $\sigma_1, \dots, \sigma_k : x[1..n-1] \rightarrow y[1..m-1]$  und  $x[n] = y[m]$

DOOFI  $\longrightarrow$  BLOEDI  
 $\sigma_1, \sigma_2, \sigma_3, \sigma_4$

y ist das, was man  
am Ende holen will

## ■ Wir haben also einen dieser vier Fälle

- Fall 1a (insert am Ende):  $\sigma_1, \dots, \sigma_{k-1} : x[1..n] \rightarrow y[1..m-1]$
- Fall 1b (delete am Ende):  $\sigma_1, \dots, \sigma_{k-1} : x[1..n-1] \rightarrow y[1..m]$
- Fall 1c (replace am Ende):  $\sigma_1, \dots, \sigma_{k-1} : x[1..n-1] \rightarrow y[1..m-1]$
- Fall 2 (nichts am Ende):  $\sigma_1, \dots, \sigma_k : x[1..n-1] \rightarrow y[1..m-1]$

Wichtig für die Formel auf der nächsten Seite: diese vier Fälle decken **alle** Möglichkeiten ab, andere gibt es nicht

# Rekursive Formel 6/7

## ■ Daraus folgt die folgende rekursive Formel

- Für  $|x| > 0$  und  $|y| > 0$  ist  $\text{ED}(x, y)$  das **Minimum** von
  - $\text{ED}(x[1..n], y[1..m-1]) + 1$  <sup>INS</sup> *das + 1 steht für die jeweils letzte Operation*
  - $\text{ED}(x[1..n-1], y[1..m]) + 1$  <sup>DEL</sup>
  - $\text{ED}(x[1..n-1], y[1..m-1] + 1$  ... falls  $x[n] \neq y[m]$  <sup>REP</sup>
  - $\text{ED}(x[1..n-1], y[1..m-1]$  ... falls  $x[n] = y[m]$

Jeder der vier führt zu einer möglichen Folge, und wir wissen, dass die minimale Folge dabei ist, deswegen das **Minimum**

- Für  $|x| = 0$  ist  $\text{ED}(x, y) = |y|$
- Für  $|y| = 0$  ist  $\text{ED}(x, y) = |x|$

# Rekursive Formel 7/7

ALIGNMENT

## ■ Alternative Sichtweise

- Visualisieren einer Folge von Operationen, indem man **x** und **y** mit geeigneten "Lücken" untereinander schreibt

|   |   |     |     |     |     |     |     |     |     |  |               |
|---|---|-----|-----|-----|-----|-----|-----|-----|-----|--|---------------|
|   |   | REP | REP | INS | REP | REP | INS | DEL | REP |  | 8 OPERATIONEN |
| x | D | O   | O   | V   | M   | A   | N   | N   |     |  |               |
| y | B | L   | O   | E   | D   | F   | R   | A   | U   |  |               |

insert = oben leer, darunter ein Buchstabe

delete = unten leer, darüber ein Buchstabe

replace = zwei ungleiche Buchstaben übereinander

- Wenn man da jetzt von links nach rechts durchgeht, hat man wieder genau eine monotone Folge
- Die rekursive Formel unterscheidet die vier Möglichkeiten in der letzten Spalte (insert, delete, replace, nix)

## ■ Code

- Mit der Formel von der Folie vorher können wir jetzt sehr leicht ein rekursives Programm schreiben
- Es funktioniert auch! (immerhin)

Aber es dauert unverhältnismäßig lange, selbst schon für relative kurze Zeichenketten

## ■ Laufzeit

- Für die Laufzeit gilt folgende rekursive Formel

$$T(n, m) = T(n-1, m) + T(n, m-1) + T(n-1, m-1) + \Theta(1)$$

$\geq T(n-1, m-1)$     $\geq T(n-1, m-1)$

- Insbesondere:

$$T(n, n) \geq 3 \cdot T(n-1, n-1) \geq 3 \cdot 3 \cdot T(n-2, n-2) \geq \dots$$

- Also:

$$T(n, n) \geq 3^{n-1} \cdot T(1, 1) \geq 3^{n-1}$$

Also **exponentielle** Laufzeit, wie auch schon bei der rekursiven Fibonacci-Berechnung

Insbesondere : Zeichenketten um 1 länger  
→ drei mal so lange

# Rekursive Implementierung 4/4

## ■ Problem

- Das rekursive Programm berechnet die gleichen ED Werte **immer und immer wieder**

Das hatten wir auch schon bei der Berechnung der Fibonacci Zahlen gesehen (in Vorlesung 1b)

z.B.  $ED(ab, cd)$

$|$  = ein rekursiver Aufruf

19 AUFRUFE

Basisfall

|               | $\varepsilon$ | c | d |
|---------------|---------------|---|---|
| $\varepsilon$ |               | - |   |
| a             |               |   |   |
| b             |               | ) |   |

→ Basisfall

→ diese Zelle entspricht  $ED(\varepsilon, cd)$

→ die hier  $ED(ab, c)$

# Literatur / Links

---

## ■ Editierdistanz

- In Wikipedia (Definition + Algorithmen)

[http://en.wikipedia.org/wiki/Levenshtein\\_distance](http://en.wikipedia.org/wiki/Levenshtein_distance)

<http://de.wikipedia.org/wiki/Levenshtein-Distanz>

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 11b, Mittwoch, 12. Juli 2017  
(Editierdistanz, Teil 2)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Drumherum

- Evaluation
- Kohlenstoff

### **Letzte Erinnerung**

Woher kommt er?

## ■ Inhalt

- Rekursives Programm
  - Iteratives Programm
  - Verfeinerung
  - Dynamische Programmierung
  - ÜB11, Aufgabe 2: beschreiben Sie einen Algorithmus, der Laufzeit  $O(\min\{|x|, |y|\} \cdot ED(x, y))$  erzielt + Begründung
- Fortsetzung von gestern
- Algorithmus + Beispiel
- bessere Laufzeit + Platz
- allgemeines Prinzip

## ■ Woher kommt er? Ihre Antworten

- "Wenn sich zwei Sterne ganz arg lieb haben"
- "Mama hat immer gesagt, dass ich aus Sternenstaub bin"
- "Man sollte sich allerdings nichts darauf einbilden, so ziemlich alles ist aus Sternenstaub"
- "Vom Großhandel"
- "Wahrscheinlich aus China"
- "Durch den Kapitalismus der USA bekommen wir täglich viel frischen Kohlen(Dollar)stoff"
- "Einem geschenkten Gaul schaut man nichts ins Maul"
- Quark-Gluonen Plasma, Tri-Alpha-Prozess, ...

# Kohlenstoff 2/5

## ■ Beginn des Universums ... heiß, sehr heiß



- Ganz am Anfang (falls es einen gab), war da erstmal nur eine Suppe aus kleinsten Teilchen (Quark-Gluonen Plasma)

Die Anzahl der Quarks ist seitdem vermutlich fix

- Nach ca. 1 Mikrosekunde haben sich Protonen (**P**) und Neutronen (**N**) gebildet ... Elektronen (**E**) waren schon da

Proton: 2 Up + 1 Down Quark, Neutron: 1 Up + 2 Down

- Nach ca. 3 Minuten haben sich die ersten **Atomkerne** gebildet, aber erstmal fast nur ganz kleine:

Wasserstoff-Kern: 1 P, Helium-4-Kern: 2P, 2N

Auch ein paar wenige Isotope (Deuterium-Kern: 1P, 1N oder Helium-3-Kern: 2P, 1 N) und noch größere Kerne

# Kohlenstoff 3/5

## ■ Recombination

- Solange es noch sehr heiß war, waren die Elektronen einfach zu schnell, um von Atomkernen eingefangen zu werden
  - Ca. 380.000 Jahre nach dem Big Bang dann kühlig genug
  - Jetzt konnten sich halbwegs stabil die ersten Atome bilden, aber auch erstmal nur die ganz leichten

Wasserstoff: 1 P + 1 E (knapp 75%)

Helium-4:       $2 \text{ P} + 2 \text{ N} + 2 \text{ E}$       (knapp 25%)

- Auch ein paar Isotope (Deuterium, Helium-3) und sehr sehr wenige schwerere Atome wie Lithium (3P) o. Beryllium (4P)

Auch schon Kohlenstoff (6P), aber nur klitzeklitzekleine Mengen

# Kohlenstoff 4/5

---

## ■ Kernfusion in Sternen

- Falls genügend H auf einem Haufen, kommt es zum Gravitationskollaps ... und zwar genau dann wenn:

Gravitationsenergie  $> 2 \cdot$  kinetische Energie

- Durch den Gravitationsdruck setzt irgendwann Kernfusion ein: je zwei H-Atome fusionieren zu Helium

Diese "Hauptsequenz" hat bei unserer Sonne vor ca. 4.6 Milliarden Jahren angefangen und dauert noch mal so lange

Tipp: Sie sollten also in den nächsten 5 Milliarden Jahren mit dem Studium fertig werden

# Kohlenstoff 5/5

---

## ■ Das Ende (von einer / unserer Sonne)

- Irgendwann (in besagten 5 Milliarden Jahren) ist der ganze Wasserstoff in der Sonne zu Helium fusioniert
- Das wabert jetzt noch ca. 120 Millionen Jahre vor sich hin und wird dabei immer heißer (i.W. durch den Quantendruck)
- Erst ab ca. 100 Millionen Grad Celsius ist es dann heiß genug, dass endlich auch die Helium-Kerne miteinander fusionieren
- Die fusionieren dann im Tri-Alpha-Prozess zu **Kohlenstoff**  
**He + He + He → C** ... auch  $\text{He} + \text{He} \rightarrow \text{Be}$ , aber instabiler  
Passiert blitzartig ("Helium-Flash") in gerade mal **3 Minuten**
- Das ist, wo der meiste Kohlenstoff herkommt 😊😊😊

## ■ Algorithmus, Idee

- Wir berechnen jedes  $\text{ED}(x', y')$ , wobei  $x'$  Präfix von  $x$  und  $y'$  Präfix von  $y$ , nur **einmal** und merken es uns
- Das sind insgesamt  $(|x| + 1) \cdot (|y| + 1)$  Werte  
*Es gibt  $|x| + 1$  Präfixe von  $x$ , nicht nur  $|x|$  ... das + 1 ist für das leere Wort ... dasselbe für  $y$*
- Wenn wir die Präfix-EDs **in der richtigen Reihenfolge** berechnen, haben wir immer schon alle, die wir gemäß der rekursiven Formel brauchen

# Iterative Implementierung 2/10

## Algorithmus, Beispiel

|   |   | $\Sigma$ | B | L | O | E | D | y →                                  |
|---|---|----------|---|---|---|---|---|--------------------------------------|
| Σ | ↓ | 0        | 1 | 2 | 3 | 4 | 5 | dieser Eintrag ist gerade ED(D, BLO) |
|   |   | 1        | 1 | 2 | 3 | 4 | 4 |                                      |
| D | 2 | 2        | 2 | 2 | 3 | 4 |   | ← BASISFALL $ x =0$                  |
| O | 3 | 3        | 3 | 2 | 3 | 4 |   | das ist das gesuchte ED(DOOF, BLOED) |
| O | 4 | 4        | 4 | 3 | 3 | 4 |   |                                      |
| F |   |          |   |   |   |   |   |                                      |

↑  
BASISFALL  $|y|=0$

## ■ Laufzeitanalyse

- Jeder Eintrag kann in Zeit  $O(1)$  berechnet werden

Für Zeile oder Spalte 0 ist das trivial

Sonst Minimum aus den drei benachbarten Einträgen

- Es gibt genau  $(|x| + 1) \cdot (|y| + 1)$  Einträge
- Also insgesamt  $\Theta(|x| \cdot |y|)$  Zeit
- Und ebenso  $\Theta(|x| \cdot |y|)$  Platz

Es geht aber auch noch schneller und mit weniger Platz,  
siehe Folien 13 – 17 und ÜB11, Aufgabe 2

## ■ Wie kommt man zu der Folge der Operationen

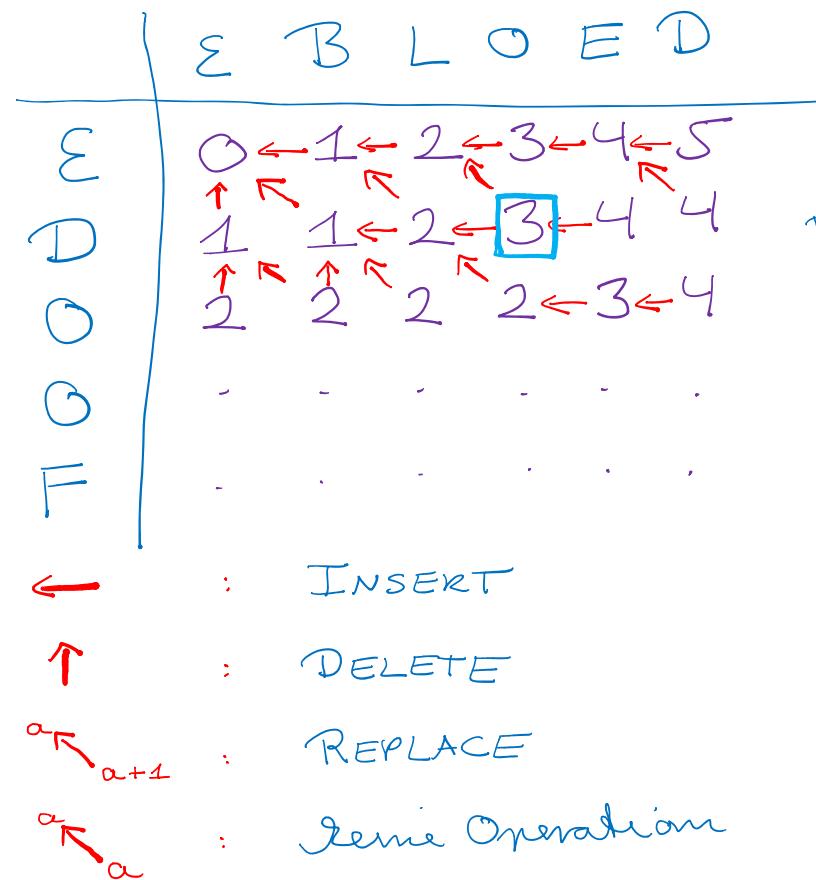
- Man merkt sich bei der Berechnung des Minimums der drei benachbarten Einträge, über welche das Minimum erzielt wurde

Das können mehrere sein (eins, zwei oder drei)

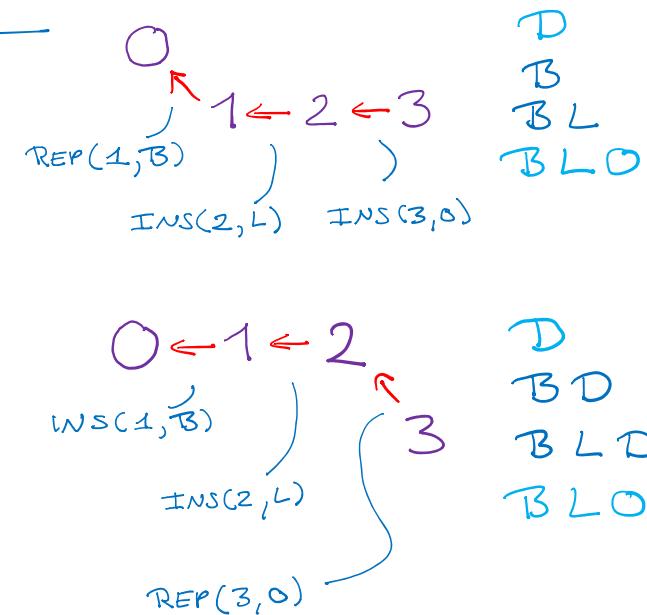
- Jeder Pfad von "unten rechts" (Eintrag bei  $|x|, |y|$ ) nach "oben links" (Eintrag bei  $0, 0$ ) gibt einem dann eine mögliche Folge von Operationen
- Man erhält so **alle** optimalen **monotonen** Folgen

Das schauen wir uns jetzt anhand unseres Beispiels an

## ■ Folge der Operationen, Beispiel



$$ED(D, BLO) = 3$$



das sind beides  
manatane Folgen  
(es gilt insgesamt  
drei für  $D \rightarrow BLO$ )

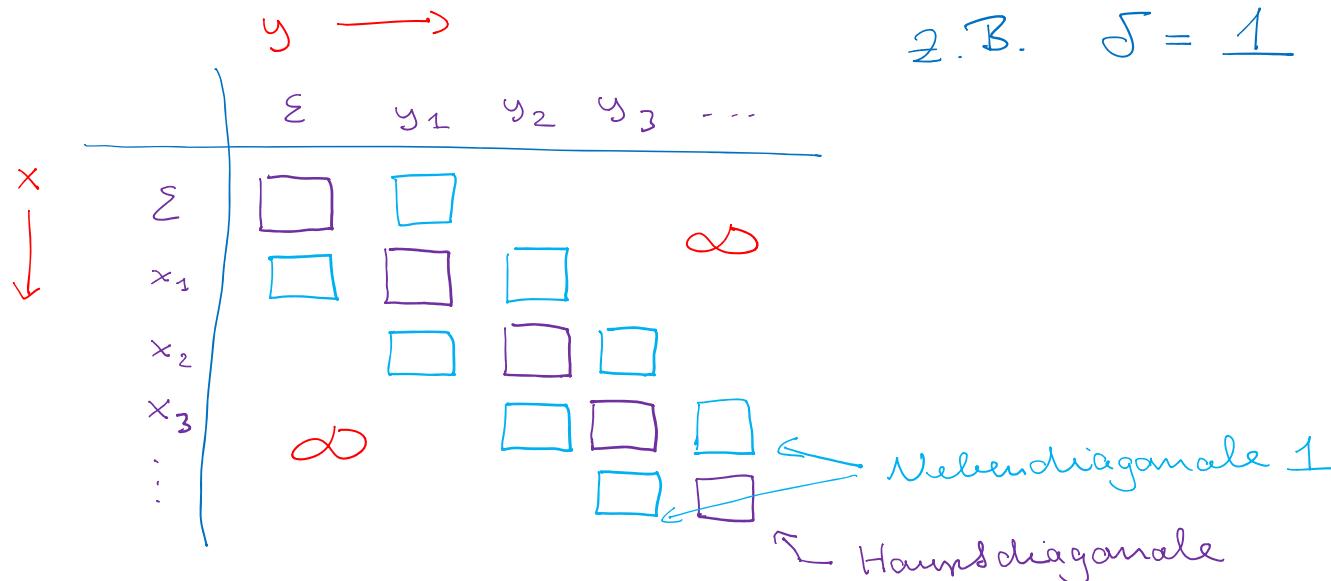
## ■ Weniger Platz

- Beobachtung: am Ende interessiert einen nur der Eintrag "unten rechts" = an Stelle  $|x|, |y|$
- Um den zu berechnen kann man auch Zeile für Zeile vorgehen, und sich nur die jeweils letzte Zeile merken  
Oder analog Spalte für Spalte berechnen, und sich dabei immer nur die jeweils letzte Spalte merken
- Das benötigt dann nur Platz  **$O(\min(|x|, |y|))$**   
Falls  $|x| > |y|$  zeilenweise, sonst spaltenweise  
**Dann hat man allerdings die Pfadinformation nicht mehr**

## ■ Bessere Laufzeit, Idee

- Nehmen wir an, wir wüssten, dass  $\text{ED}(\mathbf{x}, \mathbf{y}) \leq \delta$
- Dann reicht es, die Einträge auf der Hauptdiagonalen und den  $\delta$  Nebendiagonalen darüber und darunter zu berechnen

Alle anderen Werte können auf  $\infty$  gesetzt werden



## ■ Bessere Laufzeit, Korrektheit

- Alle Pfade aus dem Ursprungstableau, die ganz innerhalb dieses "Streifens" bleiben, finden wir auch weiterhin
- Nehmen wir an, es gibt einen optimalen Pfad, der diesen Streifen verlässt
- Dann geht er  $> \delta$  mal nach oben (je ein delete) oder  $> \delta$  mal nach links (je ein insert)
- In beiden Fällen wäre  $ED > \delta$ , aber wir hatten ja gerade angenommen, dass  $ED \leq \delta$

## ■ Bessere Laufzeit, Analyse

- Die Laufzeit ist dann  $\Theta(\min\{|x|, |y|\} \cdot \delta)$

Die Länge der Hauptdiagonale ist gerade  $\min\{|x|, |y|\}$

Die  $\infty$  außerhalb des Streifens brauchen wir ja nicht explizit hinzuschreiben

- Man bekommt auch leicht Platz  $\Theta(\min\{|x|, |y|\} \cdot \delta)$

Einfach ein Feld der Größe  $\min\{|x|, |y|\}$  pro Diagonale

Die Indizes der Nachbarn lassen sich dann immer noch leicht berechnen

## ■ Bessere Laufzeit, noch besser

- Die beschriebene Algorithmus nimmt an, dass wir ein  $\delta$  kennen, so dass  $ED(x, y) \leq \delta$
- Schöner wäre Laufzeit und Platz:

$$\Theta(\min\{|x|, |y|\} \cdot ED(x, y))$$

Also proportional zur Editerdistanz, ohne vorher irgendetwas darüber zu wissen, wie groß sie ist

Das ist Aufgabe 2 vom ÜB11

Hinweis: probieren Sie eine geeignete Auswahl von Werten für  $\delta$  aus, aber nicht einfach 1, 2, 3, 4, ...

für  $x \neq y$   
(für  $x = y$  ist das Problem trivial und  $ED(x, y) = 0$ )

## ■ Allgemeines Prinzip

- Man hat eine **rekursive** Formel, bei der dieselben Teilprobleme mehrfach vorkommen (in verschiedenen rekursiven Aufrufen)
- Man berechnet dann **iterativ**, in einer systematischen Reihenfolge, die Lösung aller relevanten Teilprobleme
- Zusammen mit dem "Wert" der optimalen Lösung erhält man so immer auch den "Weg" dorthin
- Der Name "dynamische Programmierung" ist ziemlich irreführend und hat wenig mit dem Grundprinzip zu tun
- Dijkstras Algorithmus war auch ein (verkapptes) Beispiel für dynamische Programmierung ... siehe Folie 21

## ■ Warum "dynamische Programmierung"

- Das Prinzip wurde im informatischen Kontext erstmals beschrieben von Richard Bellman in den **1950er Jahren**
- Die Benennung ist **sehr** merkwürdig
- Aber es gibt einen historischen Grund dafür:

Er hatte beim Militär gearbeitet und sein Chef hatte eine extreme Abneigung gegen Mathematik und Forschung

Sie haben aber viel Planung gemacht, und die Pläne hießen damals "programs"

Also nannte Bellman es "dynamic programming", um den mathematischen Charakter zu **verschleiern**

## ■ Beispiele

- Edi-Tier Distanz haben wir gerade gesehen
- Die Fibonacci Zahlen kann man auch mit "dynamischer Programmierung" berechnen

Einfach iterativ der Reihe nach ... und man muss sich dann immer nur die letzten beiden merken
- Dijkstras Algo. ist auch dynamische Programmierung

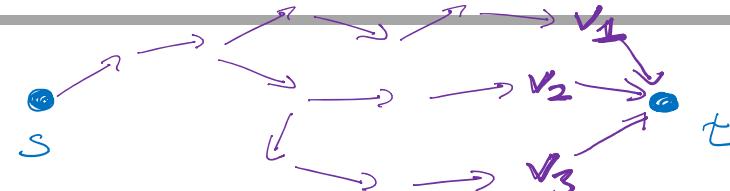
Warum sehen wir auf der nächsten Folie
- In der Freiburger Bioinformatik wird sehr viel mit dynamischer Programmierung gemacht

"Alignment" von DNA/RNA-Sequenzen = Zeichenketten

# Dynamische Programmierung 4/4

## ■ Dijkstras Algorithmus

- **Ziel:** für gegebene  $s$  und  $t$ , berechne  $\text{dist}(s, t)$   
 $s$  START       $t$  TARGET
- Rekursive Formel:  $\text{dist}(s, t) = \underbrace{\min \text{dist}(s, v) + \text{cost}(v, t)}$   
Minimum über alle nach  $t$  eingehenden Kanten ( $v, t$ )  
  
Würde man das trivial rekursiv programmieren, hätte man  
das gleiche Problem wie beim rekursiven Programm für ED
- Stattdessen macht man es **iterativ**, in der Reihenfolge  
aufsteigender Entfernung (bezüglich  $\text{dist}$  Wert) von  $s$
- Besonderheit 1: Dijkstra berechnet das Minimum für einen  
Knoten nicht auf einmal, sondern nach und nach
- Besonderheit 2: die Reihenfolge der Operationen ist bei  
Dijkstra aufwändiger zu realisieren: man braucht eine PW



# Literatur / Links

---

## ■ Dynamische Programmierung

- In Mehlhorn / Sanders

12.3 Dynamic Programming

- In Wikipedia

[http://en.wikipedia.org/wiki/Dynamic\\_programming](http://en.wikipedia.org/wiki/Dynamic_programming)

[http://de.wikipedia.org/wiki/Dynamische\\_Programmierung](http://de.wikipedia.org/wiki/Dynamische_Programmierung)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 12a, Dienstag, 18. Juli 2017  
(String Matching, Teil 1)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Organisatorisches

- Erfahrungen ÜB11 Edi-Tier

## ■ Inhalt

- String Matching Definition + Beispiel
- Naiver Algorithmus Beispiel + Code
- Knuth-Morris-Pratt Algorithmus Beispiel + Code
- Karp-Rabin Algorithmus kommt morgen dran
- ÜB12: Implementieren Sie den Karp-Rabin Algorithmus und benutzen Sie ihn zur automatischen **Plagiatserkennung**

## ■ Zusammenfassung / Auszüge

- Am Montagnachmittag hatte noch kaum jemand etwas abgegeben und es gab genau eine erfahrungen.txt
- Am Montagabend dann ein paar mehr Abgaben
- Aufgabe 1 durch die ausführlichen Hinweise gut machbar

Einige waren sich unsicher, ob Beweis ausreichend, weil man das Sieht-man-Doch Theorem nicht verwenden durfte

Das ist ein Zeichen, dass man Beweise noch üben muss!

- Bei Aufgabe 2 haben einige keinen guten Ansatz gefunden
- Bei vielen laut eigener Aussage auch aus Zeitmangel

## ■ Lösungsskizze Aufgabe 1

- Erstmal betrachten wir alle Fälle von **zwei benachbarten** Operationen  $\sigma_1$  und  $\sigma_2$  in nicht monotoner Reihenfolge

- **Fall 1:**  $\text{insert}(i_1, c_1)$ ,  $\text{insert}(i_2, c_2)$  mit  $i_1 \geq i_2$   
Äquivalent:  $\text{insert}(i_2, c_2)$ ,  $\text{insert}(i_1 + 1, c_1) \dots i_2 < i_1 + 1$

Geht analog, wenn erste Operation replace oder delete ist

- **Fall 2:**  $\text{delete}(i_1)$ ,  $\text{delete}(i_2)$  mit  $i_1 > i_2$   
Äquivalent:  $\text{delete}(i_2)$ ,  $\text{delete}(i_1 - 1) \dots i_2 \leq i_1 - 1$

Hier braucht man, dass bei delete Positionsgleichheit erlaubt

- So bekommt man für alle **neun** Kombination von insert / replace / delete eine äquivalente monotone Folge

## ■ Lösungsskizze Aufgabe 1, Fortsetzung

- Betrachten wir jetzt eine optimale nicht monotone Folge **beliebiger** Länge:  $\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_k$
- Wenn Sie nicht monoton ist, gibt es mindestens eine Stelle von benachbarten Operationen in "falscher" Reihenfolge
- Die können wir umdrehen, wie auf der Folie vorher skizziert
- Damit ist der Beweis aber noch nicht fertig

Wenn wir so argumentieren wollen, müssen mir noch zeigen,  
dass dieser Prozess auch irgendwann aufhört

Das ist nicht so einfach, deswegen argumentieren wir lieber  
etwas anders, siehe nächste Folie

## ■ Lösungsskizze Aufgabe 1, Fortsetzung

- Betrachten wir jetzt eine optimale nicht monotone Folge **beliebiger** Länge:  $\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_k$
- Betrachten wir die Operation mit der **kleinsten** Position  $i_{\min}$   
Bei mehreren inserts an Position  $i_{\min}$  nehmen wir das rechteste, bei mehreren deletes mit Position  $i_{\min}$  das linkeste  
**Mehrere replaces mit  $i_{\min}$  oder Mischung kann es nicht geben**
- Diese Operation können wir jetzt mit einer Folge von  $< k$  Nachbar-Vertauschungen an die erste Stelle bringen
- Für die Folge  $\sigma_2, \sigma_3, \dots, \sigma_k$  verfahren wir jetzt genau so, usw.
- Damit erhalten wir nach einer endlichen Anzahl von Vertauschungen eine monotone Folge

## ■ Lösungsskizze Aufgabe 2, Variante 1

- Wir benutzen den " $\delta$ -Algorithmus" aus Vorlesung 11b

Der berechnet für ein gegebenes  $\delta$  genau  $\min\{\delta, \text{ED}(x, y)\}$
- Variante 1: der Reihe nach  $\delta = 1, 2, 3, 4, \dots$  durchprobieren, bis der berechnete Wert **kleiner als**  $\delta$  ist
- Das ist korrekt: Für  $i = 1, \dots, \text{ED}(x, y)$  wird in Runde  $i$  der Wert  $i$  berechnet, in Runde  $\text{ED}(x, y) + 1$  der Wert  $\text{ED}(x, y)$
- Die Laufzeit ist  **$O(\min\{|x|, |y|\} \cdot D)$** , wobei

$D = 1 + 2 + \dots + (\text{ED}(x, y) + 1) = \Theta(\text{ED}(x, y)^2)$

Also **quadratisch** in der tatsächlichen Editerdistanz

## ■ Lösungsskizze Aufgabe 2, Variante 2

- Variante 2: der Reihe nach  $\delta = 1, 2, 4, 8, \dots$  durchprobieren, bis der berechnete Wert **kleiner als**  $\delta$  ist

Also gerade alle **Zweierpotenzen**

- Das ist korrekt: sobald  $\delta > ED(x, y)$  wird der richtige Wert berechnet und in den Runden i davor der Wert i

Das ist dasselbe Argument wie bei Variante 1

- Das  $\delta_{\max}$  bei dem der Algorithmus abbricht ist höchstens doppelt so groß wie  $ED(x, y) \dots$  also  $\delta_{\max} \leq 2 \cdot ED(x, y)$
- Also ist die Laufzeit  **$O(\min\{|x|, |y|\} \cdot D)$** , wobei

$$D = 1 + 2 + 4 + \dots + \delta_{\max} \leq 2 \cdot \delta_{\max} = O(ED(x, y))$$

# String Matching 1/3

## ■ Definition

- Gegeben zwei Zeichenketten / strings:

Ein Text (engl. **text**) typischerweise **lang**

Ein Muster (engl. **pattern**) typischerweise **kurz**

- Finde alle Vorkommen des Musters im Text
  - Es werden die Anfangspositionen zurückgegeben
- Notation: wir benutzen durchgängig **n** für die Länge des Textes und **m** für die Länge des Musters

TEXT :      D U B i D U B i D U B A D U B i D U  
                ^      ^      ^  
PATTERN .    D U B i

AUSGABE: 0, 4, 12

## ■ Motivation

- Jeder Editor hat eine "Find" Funktion (Strg+F)
- Jede Programmiersprache hat Methoden dafür

Python: `str.find(pattern, start, end)`

Java: `String.indexOf(pattern, start)`

C++: `std::string.find(pattern, start)`

- Damit bekommt man das nächste Vorkommen ab einer bestimmten Position (das `start`)
- Durch wiederholtes Aufrufen dann alle Vorkommen

## ■ Mehr Motivation

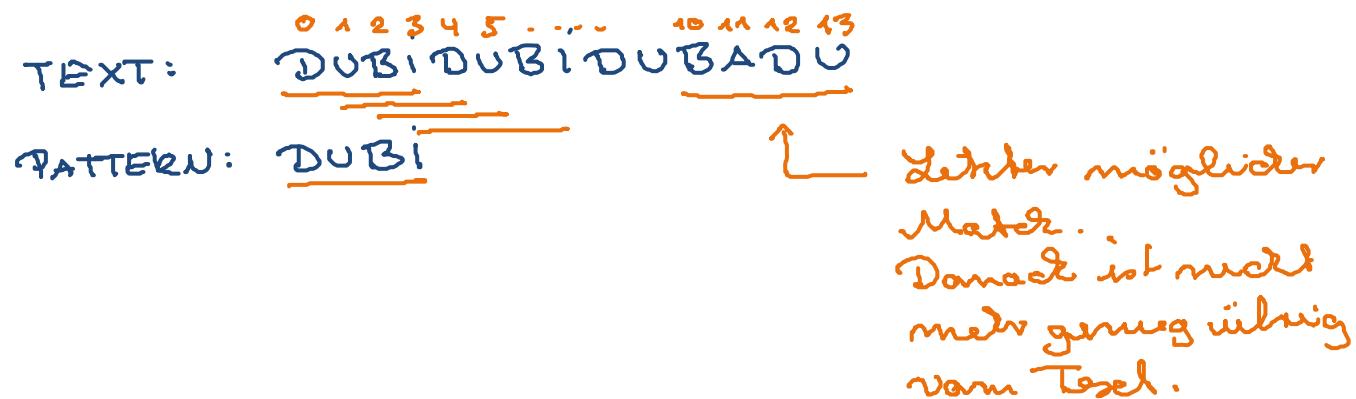
- Auch zentral für die Plagiatserkennung
- Da hat man allerdings typischerweise nicht nur ein Pattern, sondern eine Menge davon, die man wiedererkennen will

Dazu mehr in der Vorlesung morgen

# Naiver Algorithmus 1/2

## ■ Prinzip + Beispiel

- Gehe den Text von links nach rechts durch
- Prüfe an jeder Stelle ob das Muster passt, indem man es Buchstabe für Buchstabe mit dem Text dort vergleicht
- Den jeweiligen Ausschnitt (der Größe  $m$ ) aus dem Text nennen wir Fenster (engl. window)
- Das implementieren wir jetzt zusammen !



# Naiver Algorithmus 2/2

## ■ Laufzeit

$$n \gg m$$

- Sei wie gehabt  $n$  = Länge Text,  $m$  = Länge Pattern
- Laufzeit im worst case + Beispiele:  $\mathcal{O}(n \cdot m)$

text = AAAAAAAA...

pattern = AAA

text = DUDADUDIDUDI...

pattern = DUDA

Man muss (oft) durch das ganze Pattern "durchgehen"

- Laufzeit im best case + Beispiele:  $\mathcal{O}(m)$

text = AACTAACCTAAGC

pattern = XACAG

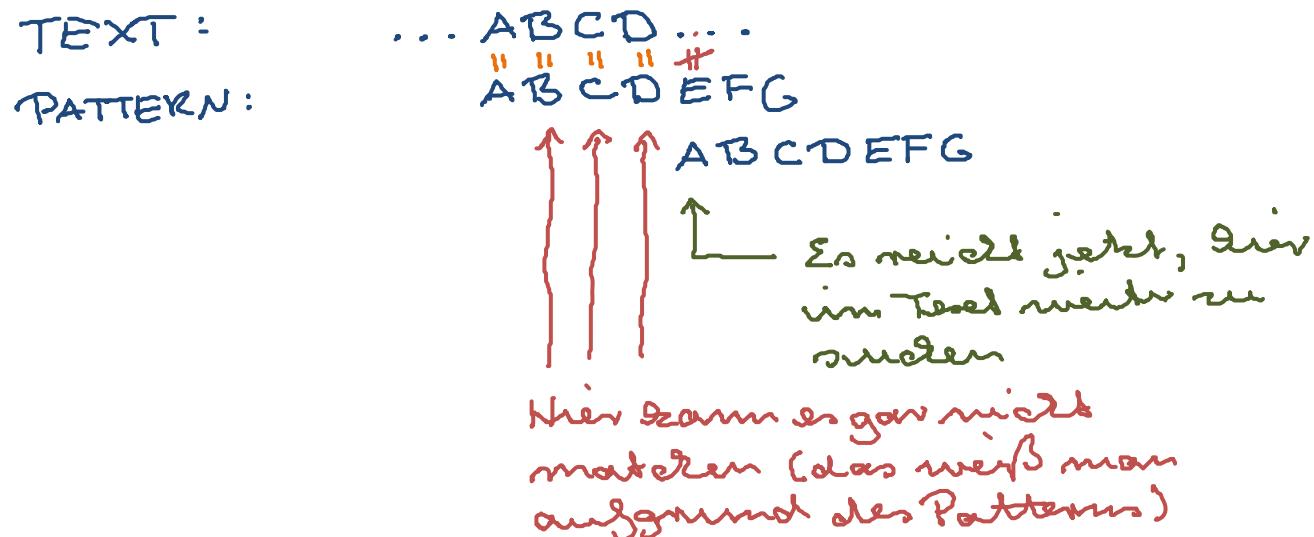
Man merkt bei den ersten Buchstaben schon, dass das Pattern nicht passt

## ■ Motivation für den Algorithmus

- Wenn man die ersten  $k$  Zeichen des Musters mit den ersten  $k$  Zeichen eines Fensters im Text verglichen hat
  - ... und jetzt das Fenster im Text um eins weiter schiebt
  - ... dann hat man  $k - 1$  Zeichen dieses Fensters schon mal mit dem Muster verglichen
- Man möchte gerne vermeiden, die nochmal anzuschauen
- Wie das gehen könnte, sieht man am besten an ein paar Beispielen ... siehe nächste Folien

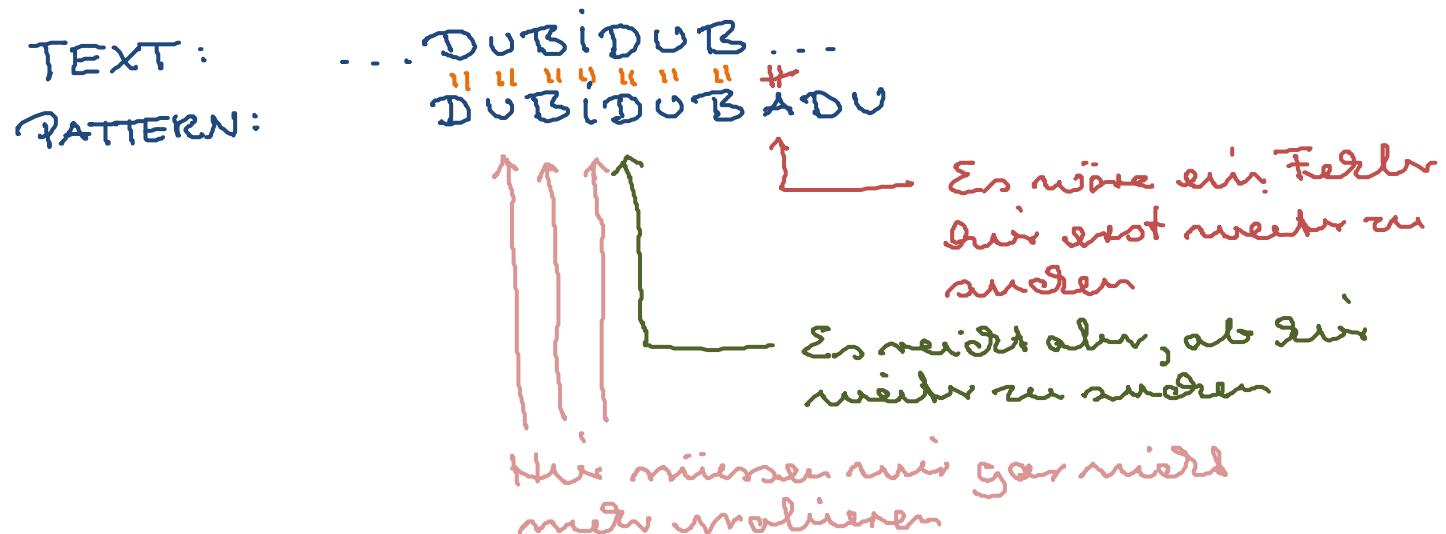
## ■ Beispiel nicht-repetitives Muster

- Im besten Fall kann man die Suche im Text da fortsetzen, wo der letzte "Mismatch" mit dem Muster war
- Nehmen wir an, dass Muster ist **ABCDEFG**
- Und nehmen wir an, an der aktuellen Textstelle passt es bis vor das **E** und dann nicht mehr **ABCDEFG**



## ■ Beispiel repetitives Muster

- Es kann aber auch sein, dass es davor auch noch einen Treffer gibt
- Nehmen wir an, dass Muster ist **DUBIDUBADU**
- Und nehmen wir an, an der aktuellen Textstelle passt es bis vor das **A** und dann nicht mehr **DUBIDUBADU**

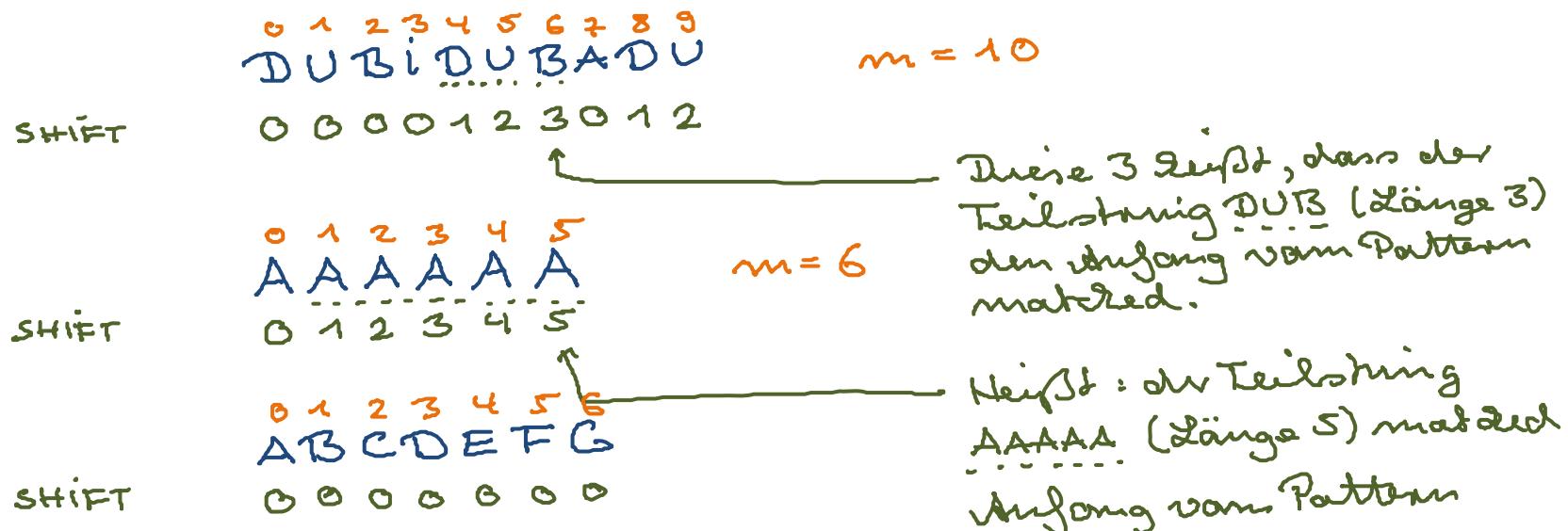


## ■ Vorverarbeitung des Musters 1/3

- Wir berechnen für jede Stelle des Musters vor, um wie viel links von der Stelle des letzten "Mismatches" man die Suche fortsetzen kann, ohne einen Treffer zu verpassen

Dabei wollen wir so weit nach rechts gehen wie möglich

- Erst mal ein paar Beispiele (für Muster)



## ■ Vorverarbeitung des Musters 2/3

- Genauer gesagt, berechnen wir für jedes  $j \in \{0, \dots, m - 1\}$

$\text{shift}[j] = \max \{ k \leq j : P[j - k + 1 .. j] = P[0 .. k - 1] \}$

In Worten: die Länge des längsten Teilstückes bis Stelle  $j$  ( $<$  alles bis  $j$ ), die gleich dem Anfang des Musters ist

- Man beachte, dass per Definition  $\text{shift}[j] \leq j$

## ■ Vorverarbeitung des Musters 3/3

- Das Feld **shift** lässt sich einfach iterativ in Zeit  $O(m)$  von links nach rechts berechnen:

- Entweder **shift[j]** ist gleich **shift[j - 1] + 1**

Wenn das Teilstück, dass zu  $\text{shift}[j - 1] > 0$  geführt hat auch noch bei  $\text{pattern}[j]$  passt

- Oder **shift[j]** ist gleich **0** bzw. gleich **1**

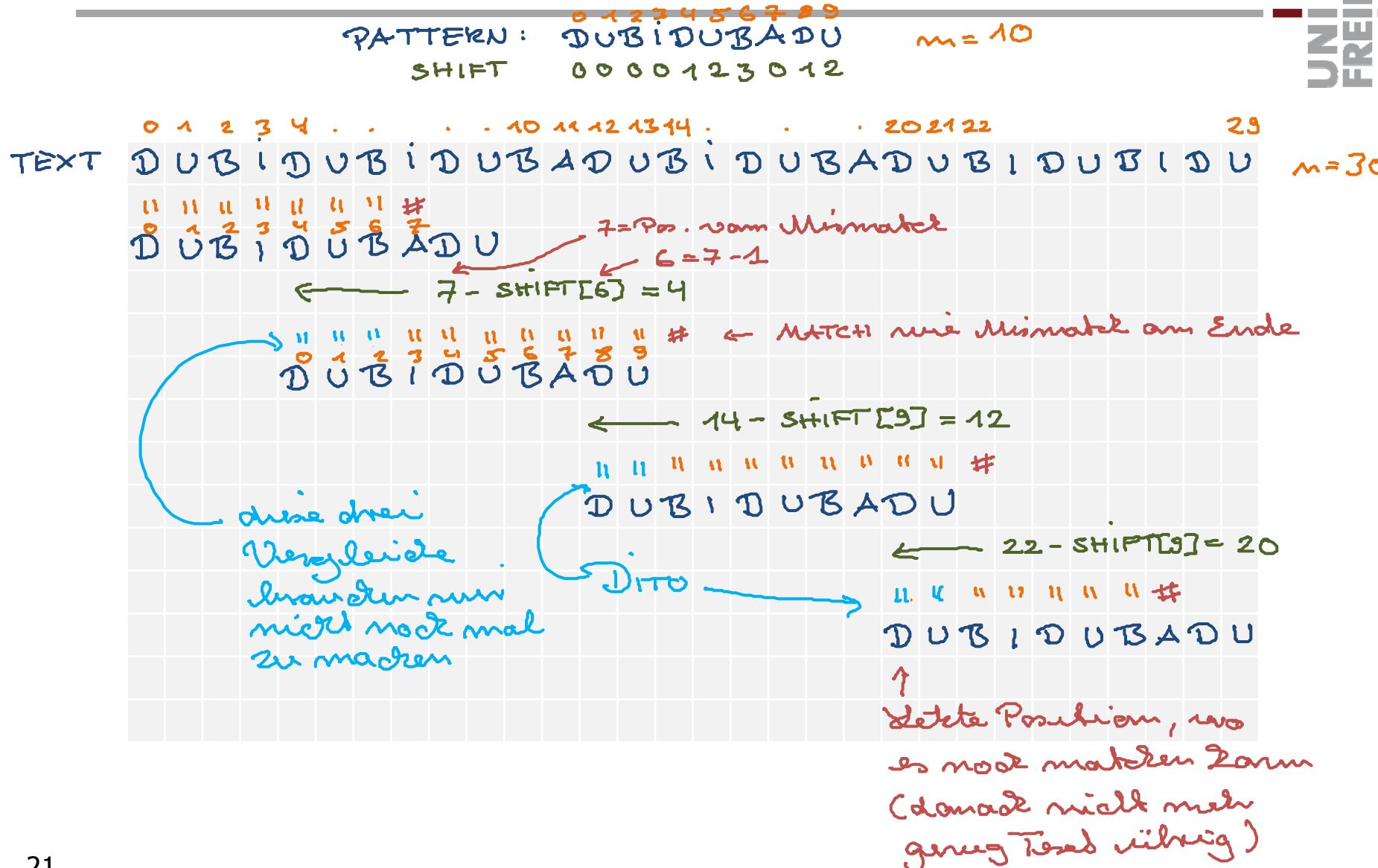
Je nachdem ob  $\text{pattern}[j] \neq \text{pattern}[0]$  oder nicht

|           |             |
|-----------|-------------|
| PATTERN : | 0 1 2 3 4   |
| SHIFT     | M i M M M 1 |
|           | 0 0 1 1 2   |

## ■ Beschreibung des Algorithmus

- Vorberechnung des `shift` Feldes wie gerade erklärt
  - Genau wie beim naiven Algorithmus:
    - Fenster der Größe `m` über den Text schieben
    - An Stelle `i` prüfen ob das Muster passt
  - Einziger Unterschied zum naiven Algorithmus:
    - Falls erster Mismatch an Stelle `j` in `P`, dann im Text weiter an Stelle `i + j - shift[j - 1]` bzw. bei `i + 1` falls `j = 0`
    - Treffer dabei wie Mismatch bei `j = |P|` behandeln
- Zum Vergleich: naiver Algo. macht immer bei `i + 1` weiter

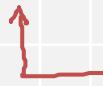
# Knuth-Morris-Pratt Algorithmus 8a/9



# Knuth-Morris-Pratt Algorithmus 8b/9

0 1 2 3 4 5  
AAAAAA  
SHIFT 0 1 2 3 4 5

AAAX  
" " "#  
AAAAAA



Jetzt würde es sogar weiter,  
als für weiter zu suchen

Die SHIFT Werte sind nicht  
optimal bei KMP  
verwenden aber für Laufzeit  $\Theta(n)$

## ■ Laufzeit

- Die Laufzeit ist proportional zur Anzahl der Vergleiche eines Zeichens des Textes mit einem Zeichen des Patterns
- Für jeden Vergleich gilt (siehe Bild auf Folie 21):
  1. Man schaut sich ein neues Zeichen im Text an  
*Eins rechts von dem, dass man zuletzt verglichen hat*
  2. Man schaut sich dasselbe Zeichen nochmal an, aber hat das Muster mindestens eins weiter "nach rechts geschoben"  
*Die Verschiebung ist gerade  $j - \text{shift}[j - 1] > 0$*
- Da man im Bild höchstens  $n$  mal "nach rechts" gehen kann, gibt es also höchstens  $2n$  Vergleiche → Laufzeit  $O(n)$

# Literatur / Links

---

## ■ String Matching

- Mehlhorn/Sanders: [gar nichts zu dem Thema!](#)

## ■ Wikipedia

- <http://de.wikipedia.org/wiki/Knuth-Morris-Pratt-Algorithmus>
- [http://en.wikipedia.org/wiki/Knuth-Morris-Pratt\\_algorithm](http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm)

## ■ Originalarbeit

- [Donald Knuth](#) und [James Morris](#) und [Vaughan Pratt](#)

Fast Pattern Matching in Strings

1977 SIAM Journal on Computing

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 12b, Mittwoch, 19. Juli 2017  
(String Matching, Teil 2)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Inhalt

- KMP Algorithmus Fortsetzung von gestern
  - Karp-Rabin (KR) Algorithmus Prinzip + Beispiele
  - Plagiatserkennung Anwendung von KR
  - ÜB12: Implementieren Sie den Karp-Rabin Algorithmus und benutzen Sie ihn zur automatischen **Plagiatserkennung**

# Edi-Tier 1/3

---

## ■ Woher stammt es und was hat es uns voraus?

- "Eins ist klar: es kommt aus dem großen Welt-Edi-Tor"
- "Es verträgt mehr Wodka"
- "Das Edi-Tier stammt sicher vom Edi-Fant ab"
- "Ein Kumpel von mir heißt Eddie und der hat einen Hund"
- "Hier ist ein Foto: <https://goo.gl/yvHDCJ>"
- "The Edi-Tier is called Leafcutter Ant and it's fascinating"

"In the Zoo Burkart, Lörracher Straße, there is a very big colony. Its amazing to watch the ants work. They transport leafs, deposit waste on their waste site, farming their mushrooms. Its worth visiting just to see the ants."

## ■ Blattschneiderameisen (Leafcutter ants)

- Eine faszinierende Tierart, die uns einiges voraus haben
- Sie sind bekannt für die grünen Blätter, die sie durch die Gegend tragen (bis zum 20-fachen ihres Körpergewichtes)
- Die Blätter sind aber nicht für sie selber, sondern werden in vielen Arbeitsschritten zu kleinen Kugelchen verarbeitet, mit denen die Ameisen dann schließlich einen **Pilz** füttern
- Von diesem Pilz ernähren sich dann die Larven der Ameisen

Die Ameisen betreiben also komplexe **Landwirtschaft**, und sie tun das seit mindestens 15 Millionen Jahren

Menschen seit gerade mal 10.000 Jahren

## ■ Schädlingsbekämpfung

- Forscher haben sich lange gewundert / gefragt, warum die Ameisen keine Probleme mit Schädlingen haben
  - Aber sie haben: ohne die Ameisen, wird der Pilz in relativ kurzer Zeit von (anderen) Schimmelpilzen dahingerafft
- Einigen Forschern ist aufgefallen, dass manche von den Ameisen weißes Zeug auf dem Rücken tragen
- Das sind Bakterien, die u.a. **Antibiotika** produzieren, gegen die Schädlinge, die für den kultivierten Pilz gefährlich sind
  - Insbesondere sind da viele Antibiotika dabei, die auch wir Menschen dann im 20ten Jahrhundert mal entdeckt haben
- Die Bakterien helfen außerdem beim **Düngen** des Pilzes

## ■ Grundidee

- Wir schieben wieder ein Fenster der Größe  $m$  über den Text und schauen an jeder Stelle, ob es zu dem Muster passt
- Nehmen wir an die Buchstaben sind die Ziffern 0..9
- Dann kann man das Muster als (große) ganze Zahl auffassen ... und das Stück Text im aktuellen Fenster ebenso
- Verschiebt man das Fenster um eins nach rechts, lässt sich die neue Zahl leicht aus der alten berechnen

$m = 12$

Text: 5 7 2 8 3 0 3 5 4 8 2 6

$m = 3$

Pattern: 2 8 3

$$\begin{array}{rcl} \times & 283 & \neq 572 \\ \times & 283 & \neq 728 \\ \checkmark & 283 & = 283 \end{array}$$

$\begin{array}{l} = 22 \\ - 5 \times 100 \\ \times 10 \\ + 8 \end{array}$     $\begin{array}{l} = 720 \\ \times 10 \\ + 8 \end{array}$     $\begin{array}{l} = 728 \\ \times 100 \\ \times 10 \\ + 3 \end{array}$

# Karp-Rabin Algorithmus 2/11

## ■ Rechnen mit diesen "Zahlen"

- Pro Fensterverschiebung braucht man nur konstant viele Rechen-Operationen auf diesen Zahlen
- Wenn wir mit den Zahlen in konstanter Zeit operieren könnten, wäre die Laufzeit also  $O(n)$

Aber diese Zahlen können sehr groß werden

Bei Basis  $b$  und einem Muster der Länge  $m$  bis zu  $b^m$

Dafür braucht man  $\log_2 b^m = m \cdot \log_2 b$  Bits

$$\log_2 b = 8$$

Für  $b = 256$  (ASCII) und  $m = 10$  sind das schon 80 Bits ...

zu viel für z.B. ein int auf einem 64-Bit Rechner

$$b = 10 : \\ 283 = 2 \cdot 10^2 + 8 \cdot 10^1 + 3 \\ b^{m-1} \quad b^{m-2} \quad b^0$$

## ■ Hashwerte

- Statt Zahl  $x$  betrachten wir Hashwert  $h(x) = x \bmod M$

Die Hashwerte sind dann aus dem Bereich  $\{0, \dots, M - 1\}$

- Bei einem Match sind sicher auch die Hashwerte gleich

Aber gleiche Hashwerte bedeuten nicht unbedingt Match

Wenn  $M$  groß ist, ist es allerdings unwahrscheinlich, dass ungleiche Zahlen auf denselben Wert abgebildet werden

Das kennen wir ja schon vom Hashing (Kollisionen)

- Wenn die Hashwerte gleich sind, überprüfen wir wie beim naiven Algorithmus Buchstabe für Buchstabe

# Karp-Rabin Algorithmus 4/11

## ■ Beispiel

Text: 5 7 2 8 3 0 3 5 4 8 2 6

Pattern: 2 8 3       $\varrho(283) = 3$

Modulus:  $M = 5$       ,     $\varrho(x) = x \bmod 5$

In der Praxis benutzt man viel größere Werte für M

$\varrho(572) = 2 \neq 3 = \varrho(283) \Rightarrow$  SICHER kein Match

$\varrho(728) = 3 = 3 = \varrho(283) \Rightarrow$  Kandidat für ein Match, aber Nachprüfen der übrigen Zeichen (O(n) Zeit)  $\rightarrow$  Kein Match

$\varrho(283) = 3 = 3 = \varrho(283)$



$\Rightarrow$  Kandidat für Match und es ist auch einer (mindest O(n) Zeit)

# Karp-Rabin Algorithmus 5/11

## ■ Laufzeit mit Modulus

$$n = \# \text{Textes} , m = \# \text{Pattern}$$

- Im schlechtesten Fall:  $\mathcal{O}(n \cdot m)$

Wenn Hashwert für Muster und Fenster immer gleich,  
obwohl das Muster gar nicht überall passt

Bei guter Wahl von M **unwahrscheinlich** ... Folie 11+16

- Im besten Fall:  $\mathcal{O}(n)$

Das passiert, wenn der Modulus für das Muster und für  
das Textfenster nur dann gleich ist, wenn das Muster  
auch passt *und "wenige" Matches*

Bei guter Wahl von M **wahrscheinlich** ... Folie 11+16

# Karp-Rabin Algorithmus 6/11

## ■ Wahl der Basis $b$ und des Modulus $M$

- Wir hätten gerne, dass es unwahrscheinlich ist, dass  $x \neq y$  aber  $h(x) = h(y)$ , für  $h(x) = x \bmod M$
- Dazu sollte  $M$  möglichst groß sein und keinen Teiler mit  $b$  gemeinsam haben ... Negativbeispiel dazu:

Wenn  $b$  durch  $M$  teilbar, würde nur letzte "Stelle" zählen

- Wir wählen deshalb im Programm  $b = 257$

Kleinste Primzahl, die größer ist als jeder ASCII Code

- Wir können dann  $M$  im Prinzip beliebig wählen

$$m=4, b=257$$

$$\text{doof} = 100 \cdot 257^3 + 111 \cdot 257^2 + 111 \cdot 257^1 + 102 \cdot 1$$

Ascii     $100 \quad 111 \quad 102$        $b^{m-1}$        $b^{m-2}$        $b^1$        $b^0$

## ■ Rechnen modulo $M$

- Es gelten folgende Rechenregeln (für jedes  $M \in \mathbb{N}$ )

$$(a \cdot b) \bmod M = ((a \bmod M) \cdot (b \bmod M)) \bmod M$$

$$(a + b) \bmod M = ((a \bmod M) + (b \bmod M)) \bmod M$$

Beweis: gute Aufgabe zur Mathe-Phobie-Überwindung !

- Das heißt, wir können bei einem größeren Ausdruck das **mod M** auch schon auf Teilterme anwenden

Das ist wichtig, damit die Zahlen nicht zu groß werden

$$\begin{aligned} & \text{10111011 mod } 2^4 \\ & = 1011 \end{aligned}$$

# Karp-Rabin Algorithmus 8/11

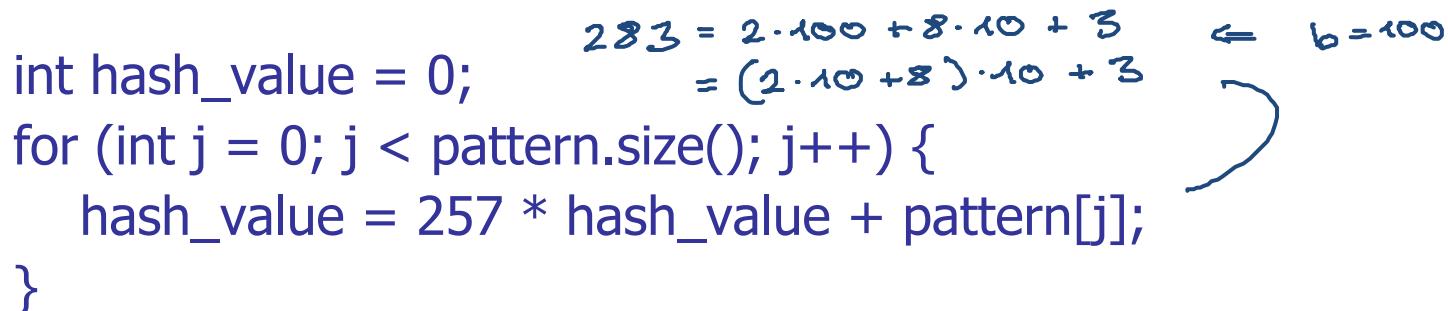
~~z.B. bei einem 64-Bit unsigned und in C++  
maß an Überlauf „automatisch“ mod  $2^{64}$~~

## Berechnung Hashwert in Java, C++, Python

- In Java und C++ kann man statt einem expliziten  $\text{mod } M$  einfach den Überlauf beim Rechnen mit `int` nutzen

$283 = 2 \cdot 100 + 8 \cdot 10 + 3 \quad \leftarrow b = 100$   
 $= (2 \cdot 10 + 8) \cdot 10 + 3$

```
int hash_value = 0;
for (int j = 0; j < pattern.size(); j++) {
 hash_value = 257 * hash_value + pattern[j];
}
```



- In Python können Zahlen beliebig groß werden, dort braucht man also ein explizites  $\text{mod } M$

```
hash_value = 0
for c in pattern:
 hash_value = (257 * hash_value + c) % M
```

$$\begin{aligned} -38 \bmod 10 \\ = -38 + 4 \cdot 10 = 2 \end{aligned}$$

## ■ Modulo bei negativen Zahlen

- Für eine beliebige ganze Zahl  $x$  ist mathematisch einfach

$$x \bmod m = \min \{ x + q \cdot m : q \in \mathbb{Z} \text{ und } x + q \cdot m \geq 0 \}$$

Damit ist immer  $x \bmod m \in \{0, \dots, m-1\}$

Zum Beispiel:  $24 \bmod 10 = 4$  und  $-4 \bmod 10 = 6$

- In Java und C++ ist aber  $-x \% m = -(x \% m)$

Zum Beispiel:  $24 \% 10 = 4$  und  $-4 \% 10 = -4$

Wenn man es für Java oder C++ so macht, wie auf der Folie vorher, braucht man sich darum nicht zu kümmern

- In Python ist  $x \% m = x \bmod m$  ... insbes.  $-4 \% 10 = 6$

## ■ Erweiterung auf mehrere Patterns

- Nehmen wir jetzt an, wir haben **k** Patterns und wollen alle Vorkommen **aller** dieser Patterns in einem Text finden
- Eine typische Anwendung dafür ist die Plagiatserkennung
  - Patterns = Sätze / Fragmente aus einer Originalarbeit
  - Text = vermeintliches Plagiat
- Wir könnte jetzt einfach einen unserer bisherigen Algorithmen **k** mal anwenden, für jedes Pattern einmal
  - Die Laufzeit würde sich dann um dem Faktor k erhöhen, das dauert für große k (viele Patterns) sehr lange
  - Für das ÜB12, Aufgabe 2 ist k immerhin gleich 288

## ■ Erweiterung von Karp-Rabin für **k** Patterns

- Speichere die Patterns in einer Map, mit dem Hashwert als Schlüssel und dem Pattern als Wert
- Für jedes Textstück (aus dem aktuellen Fenster), schauen wir dann, ob es Patterns mit demselben Hashwert gibt

Das geht mit einer geeignete Map in  $O(1)$  Zeit

- Falls ja, prüfen wir für jedes solche Pattern nach, ob es passt, wie beim naiven Algorithmus

Für  $M \gg k$  wird es selten vorkommen, dass der Hashwert übereinstimmt, aber das Pattern trotzdem nicht matched
- Die Laufzeit ist dann  **$O(n + \text{Gesamtlänge der Matches})$**

# Literatur / Links

---

## ■ Karp-Rabin Algorithmus

- Wikipedia

<http://en.wikipedia.org/wiki/Rabin-Karp-Algorithmus>

- Originalarbeit von Rabin & Karp:

[Richard Karp](#) und [Michael Rabin](#)

Efficient Randomized Pattern Matching

1987 IBM Journal of Research and Development

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 13a, Dienstag, 25. Juli 2017  
(Profiling, Compileroptimierung, Maschinencode)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

## ■ Organisatorisches

- Erfahrungen ÜB12                  String Matching

## ■ Inhalt

- Performance Tuning

Profiling                  welcher Teil der Laufzeit wofür

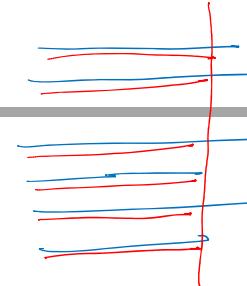
Compileroptimierung                  optimierter Maschinencode

Maschinencode                  kurzer Überblick / Crashkurs

- Letzte Vorlesung morgen: **Evaluation**sergebnisse + Infos zur **Klausur** + aktuelle **Forschung** an unserem Lehrstuhl

## ■ Zusammenfassung / Auszüge

- Nur noch relativ wenige Abgaben
  - Aufgabe war gut machbar und wie immer interessant
  - Von den **287** Textstücken aus fragments.txt kommen sage und schreibe **39** in phd-thesis.txt vor  
**150 Zeichen pro Textstück, wörtlich + mit Interpunktions !**
  - Einige meinten, das wäre kein Plagiat, weil ja nicht alles kopiert wurde bzw. es käme drauf an
- Wir werden in Zukunft zu Beginn jeder Veranstaltung genauer erklären, was genau ein Plagiat ist und was nicht



## ■ Pattern verschiedener Größe

- Die Pattern in "fragments.txt" waren alle gleich lang
- Wichtig, weil das Textfenster **eine** feste Größe hat

Man schiebt das ja Zeichen für Zeichen über den Text und berechnet dabei jeweils in  $O(1)$  Zeit den neuen Hashwert

- Trick bei Patterns verschiedener Größe:

Betrachte von allen Patterns die ersten **min** Zeichen und die dazu gehörigen Hashwerte, **min** = Länge kürzestes Pattern

Bei gleichem Hashwert, Vergleich mit dem ganzen Pattern

Funktioniert gut, wenn ein min-Präfixe meistens nur dann matched, wenn auch das ganze Pattern matched ... siehe Forum

## ■ Donald Knuth

- Autor von TeX
- Autor von "The Art of Computer Programming"
- Ko-Autor von "Concrete Mathematics"
- Hat sich von der Uni Stanford fr hpensionieren lassen

Weil er ausgerechnet hat, dass er 20 Jahre am St ck braucht, um sein Lebenswerk (sein Buch) fertig zu stellen

"Email is a wonderful thing for people whose role in life is to be on top of things. But not for me; my role is to be on the bottom of things. What I do takes long hours of studying and uninterrupted concentration."

- Retro-Webseite: <http://www-cs-faculty.stanford.edu/~uno/>

## ■ Motivation

- Wie viel Prozent der Laufzeit verbringt mein Programm mit welcher Funktion (auch Bibliotheksaufrufe)
- Programme, die das messen, nennt man **Profiler**  
Sie laufen üblicherweise mit dem Programm mit und verlangsamen es (durch die Messungen)
- Hier am Beispiel eines sehr einfachen Programms  
**ArrayFill**: fülle ein Feld mit 1 Millionen ints
- Wir schauen uns sehr einfache Profiler an  
Die sind für einfache Programme schon ganz nützlich ...  
für tiefere Analysen, gibt es teure kommerzielle Software

# Profiling 2/4

---

## ■ Java: hprof

- Einfach das kompilierte Java-Programm ausführen mit  
`java -agentlib:hprof=cpu=times ArrayFillMain`
- Erzeugt eine menschenlesbare Textdatei **java.hprof.txt**

Die Prozentzahlen (wie viel % der Laufzeit in welcher Funktion verbraucht werden) stehen ganz am Ende

- Beobachtung für unser ArrayFillMain Programm:

`ArrayList<Integer>`      braucht ca. 20ms

`Natives int array`      braucht ca. 2ms

**Weniger als 20%** der Laufzeit werden in der eigentlichen `java.util.ArrayList.add` Funktion verbracht

## ■ C++: gprof

- Übersetzen: `g++ -pg -o ArrayFillMain ArrayFillMain.cpp`
- Ausführen: `./ArrayFillMain` → erzeugt Binärdatei `gmon.out`
- Anschauen: `gprof ./ArrayFillMain ... nicht gprof gmon.out`

– Beobachtung für unser `ArrayFillMain` Programm, **mit -pg**

`std::vector<int>` braucht ca. 7ms

`Natives int array` braucht ca. 2ms

– Ohne -pg und mit Optimierungsoption -O3 (s. spätere Folie):

`std::vector<int>` braucht ca. 1ms

`Natives int array` braucht ca. 1ms

# Profiling 4/4

---

## ■ Python: cProfile

- Einfach ausführen mit

```
python3 -m cProfile -s time array_fill.py
```

Messergebnis wird dann gleich am Ende mit ausgegeben

- Beobachtung für unser [ArrayFillMain](#) Programm:

`array = [] ...` braucht ca. 100ms

- Python verwaltet als ungetypte Sprache intern komplexe Objekte, selbst wenn die Werte letztendlich nur ints sind

Das kostet viel Zeit; um das zu umgehen, gibt es Compiler so wie Cython ... siehe spätere Folie

## ■ Grundprinzip eines Compilers

- Der Code wird in eine entsprechende Folge von Anweisungen in Maschinencode übersetzt

Kurze Einführung dazu auf den **Folien 15 – 23**

- Im einfachsten Fall wird jede Zeile Code in eine Folge von Anweisungen in Maschinencode übersetzt

Zwar korrekt, ergibt aber selten den schnellsten Code

Das schauen wir uns jetzt mal anhand eines sehr einfachen Programms für alle drei Programmiersprachen an

## ■ C++

- In C/C++ lässt sich der Assemblercode leicht erzeugen mit

**g++ -S Simple.cpp**

Das gibt dann eine Datei Simple.s die man sich einfach in einem Texteditor anschauen kann

- Ohne Optimierung: der Code wird in der Tat Zeile für Zeile in Maschinencode übersetzt
- Mit Optimierung: der Compiler tut erstaunliche Dinge

Das meiste passiert schon bei  $-O 1$  (Optimierungsstufe 1)

Mit  $-O 3$  (Optimierungsstufe 3) werden dann alle Tricks, die es überhaupt gibt, aktiviert ... siehe man g++

## ■ Java

- Der Java-Compiler übersetzt erst in sog. Bytecode  
Ein abstrakter Maschinencode ... siehe Folie 23
- Den Bytecode kann man sich einfach anschauen mit
  - `javac Simple.java` kompiliert zu `Simple.class`
  - `javap -c Simple` Bytecode aus `Simple.class`
- Dieser Bytecode wird dann zur Laufzeit in richtigen (auf der CPU ausführbaren) Maschinencode übersetzt
- Bei häufig benutzten Funktionen wird M-Code wiederbenutzt:  
`java -XX:+PrintCompilation Simple`  
Benötigt `hsdis-amd64.so`, siehe <https://github.com/abak/openjdk-hsdis>

## ■ Python

- Python übersetzt ebenfalls in einen Bytecode

Aber ein etwas anderer als der von Java

- Den kann man sich in Python anschauen mit z.B.

`>>> import dis` Modul zum "disassembeln"

`>>> import array_fill` Unser Code

`>>> print(dis.dis(array_fill))` Eine Funktion daraus

## ■ Cython

- Mit Cython kann man auch äquivalenten C-Code erzeugen  
`cython -3 --embed -o array_fill.c array_fill.py`
- Kann man dann mit irgendeinem C Compiler übersetzen  
`gcc -o array_fill array_fill.c`  
    `-I /usr/include/python3.5m -lpython3.5m`
- Mit Cython kann man im Python Code auch getypte Variablen (C-Style) benutzen, und damit enorm viel schneller sein

`array = []`

Python-Style      ca. 100ms

`cdef int array[10e6]`

C-Style      ca. 2 ms

Mit Cython so schnell wie Java native arrays

## ■ Motivation

- Das ist der (einige) Code, den die CPU versteht
- Code in einer höheren Sprache muss erstmal in Maschinencode übersetzt werden, damit man ihn ausführen kann

Insbesondere Code in Python, Java oder C++

- Anweisungen in Maschinencode sind durch Zahlen codiert
- Die menschenlesbare Form nennt man **Assembler**

Dafür sehen wir gleich einige Beispiele

## ■ Kurz zur Geschichte

- 1972: Intel 8008 (der erste 8-Bit Mikroprozessor)
- 1974: Intel 8080 (die ersten 16-Bit Operationen)
- 1978: Intel 8086 (16 Bit, erstes Mitglied der x86 Familie)
- 1985: Intel 80386 aka i386 (32 Bit)
- 1993: Intel Pentium (32 Bit)
- 2003: AMD 64, Intel 64 (64 Bit, manchmal x64 genannt)
- Die sind alle rückwärts kompatibel bis zum Intel 8086 !
- Grundprinzip über die Jahre unverändert ... nächste Folien

# Maschinencode 3/9

---

## ■ Register

- Das sind Variablen, die es "in Hardware" in der CPU gibt
- Die ursprünglichen Intel 8086 Register (**16 Bit**) heißen:
  - AX, BX, CX, DX** : "accumulator", "base", "counter", "data"
  - SI, DI**: "source index", "destination index"
  - SP, BP**: "stack pointer", "base pointer"
- Die können im Prinzip alle für alles verwendet werden, haben aber für bestimmte Befehle / in bestimmten Kontexten eine besondere Bedeutung
  - Zum Beispiel arbeiten viele Rechenoperationen auf **AX**

# Maschinencode 4/9

---

## ■ Register

- Die Intel 80836 Register (32 Bit) heißen:  
**EAX, EBX, ECX, EDX**, etc. [E = extended]  
außerdem zusätzliche 64-Bit Register **MMX0, MMX1, ...**
- Die AMD Opteron Register (64 Bit) heißen:  
**RAX, RBX, RCX, RDX**, etc. [R = register]  
außerdem zusätzliche 64-Bit Register **R8, R9, ..., R15**  
und sechzehn 128-Bit Register **XMM0, XMM1, ...**

## ■ Heap und Stack

- Es gibt zwei Arten von Speicher

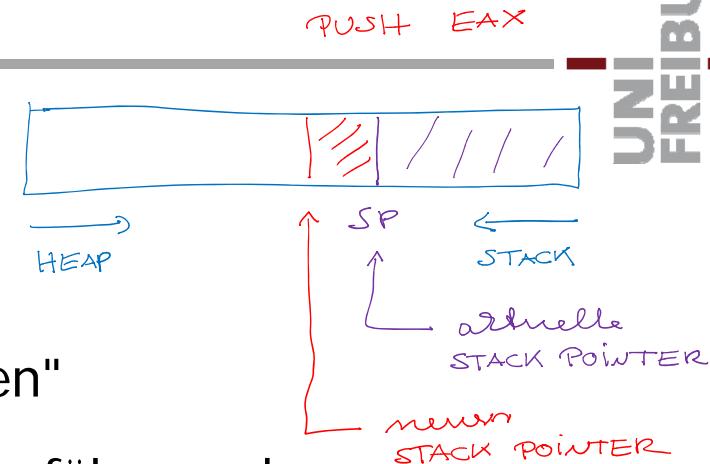
- **Heap:** wächst von "unten nach oben"

Hier liegt alles, was während der Ausführung des Programms dynamisch alloziert wird (mit `new`)

- **Stack:** wächst von "oben nach unten"

Jeder Funktionsaufruf hat ein zusammenhängendes Stück auf dem Stack, da liegen:

die Argumente, die lokalen Variablen, die Rücksprungadresse, die Adresse des Stücks Stack von der aufrufenden Funktion



## ■ Basisinstruktionen

- mov X, Y : weise den Wert von X an Y zu

Hier, wie auch bei vielen anderen Instruktionen, können X und Y Register sein oder auch Inhalt einer Stelle im Speicher, auf die ein Register zeigt

**Beispiel:** -4(%rbp) ist der Inhalt an der Adresse, die im Register RBP steht, minus 4

## ■ Arithmetische Operationen

- Zum Beispiel:

`add, sub, mul, div, inc` (increment), `dec` (decrement), ...

`and, or, xor, sal` (shift left), `sar` (shift right), ...

- Suffixe bei den Anweisungen

Kein Suffix = 16 bits, `l` = 32 bits ("long"), `q` = 64 bits ("quad")

Beispiele: `mov, movl, movq, add, addl, addq`, ...

## ■ Operationen auf dem Stack

- push X : X auf Stack legen ... vermindert SP = stack pointer
- pop X : X vom Stack holen ... erhöht SP = stack pointer

## ■ Vergleiche und Sprünge

- cmp X, Y : vergleiche X und Y ob < oder > oder =
- je X, jne X, jl X : springe nach X je nach < oder > oder =
- jmp X : springe nach X ohne Bedingung

## ■ Java Bytecode

- Ein abstrakter Maschinencode
- Sehr ähnlich zu [x86](#), aber bewusst einfach gehalten
- Register heißen einfach [1](#), [2](#), [3](#), ...
- Beispiel [x86](#) Assembler (links) vs. Java Bytecode (rechts)

|                   |                          |
|-------------------|--------------------------|
| mov eax, -4(%rbp) | <a href="#">iload_1</a>  |
| mov edx, -8(%rbp) | <a href="#">iload_2</a>  |
| add eax, edx      | <a href="#">iadd</a>     |
| mov ecx, eax      | <a href="#">istore_3</a> |

# Literatur / Links

---

## ■ Profiling with gprof / hprof / cProfile

- <http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html>
- <http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html>
- <https://docs.python.org/3/library/profile.html>

## ■ Heap und Stack

- [http://en.wikipedia.org/wiki/Memory\\_management](http://en.wikipedia.org/wiki/Memory_management)
- [http://en.wikipedia.org/wiki/Call\\_stack](http://en.wikipedia.org/wiki/Call_stack)

## ■ x86 Befehlssatz / Java Bytecode

- [http://en.wikipedia.org/wiki/X86\\_instruction\\_listings](http://en.wikipedia.org/wiki/X86_instruction_listings)
- [http://en.wikipedia.org/wiki/Java\\_bytecode](http://en.wikipedia.org/wiki/Java bytecode)

# Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 13b, Mittwoch, 26. Juli 2017  
(Evaluation, Klausur, Aktuelle Forschung)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Inhalt

- Evaluationsergebnisse      Zusammenfassung + Ausblick
- Klausur                          Termin + Modus + Aufgaben
- Vorstellung Lehrstuhl      Aktuelle Forschung + nächste VLen

## ■ Teilnahme

- Noch aktive Teilnehmer\*innen : **160**
- An der Evaluation teilgenommen : **121**  
**92 x Info, 8 x Info Nebenfach, 20 x ESE, 18 x Sonstige**  
**70 x 2. Semester, 36 x 4. Semester, 31 x höhere Semester**
- Nominierungen für Lehrpreis : **83**
- Im Folgenden, eine Zusammenfassung des Feedbacks  
Die **vollständigen Ergebnisse, inklusive aller Statistiken und aller Freitextkommentare, finden Sie auf dem Wiki**

## ■ Art und Weise

- **Viel gelernt:** 67% trifft voll zu, 26% trifft zu, 6% ok
- **Verständlich:** 83% trifft voll zu, 12% trifft zu, 5% ok
- **Niveau:** 57% angemessen, 37% hoch, 6% tief
- **Qualität:** 72% sehr gut, 22% gut, 6% geht so
- Dozentin: motivierend, verständlich, kompetent, lockere Stimmung, unterhaltsam, humorvoll, authentisch, engagiert
- Inhalt: ein Thema pro Woche, gute Mischung aus Theorie und Praxis, "keine Overkill-Formalismen", Bezüge zu anderen Themen / Fragen über den Tellerrand
- Tempo, Bilder, Live-Coding: nächste Folie

## ■ Tempo, Bilder, Live-Coding

- Viele fanden das Tempo gerade richtig, einigen wenigen war es zu langsam bzw. hätten sich mehr Stoff gewünscht

"Sehr angenehmes Tempo ohne Gehetze"

"Schneller durch den Stoff gehen, sind nicht in der Schule"

- Viele fanden das gemeinsame Entwickeln von Zeichnungen und Code sehr hilfreich, aber einigen dauert es zu lange

"Das Live-Coding hat Klasse und lockert die VL ungemein auf ... bitte beibehalten"

"Weniger Zeit für das Zeichnen von simplen Schaubildern, stattdessen tiefergehende theoretische Hintergründe bzw. ausführlichere mathematische Beweis"

# Evaluationsergebnisse 4/8

## ■ Übungsblätter

- Aufwand relativ zu ECTS ... 1 = sehr hoch, 5 = sehr gering  
 $5\% \times 1$   $40\% \times 2$   $50\% \times 3$   $4\% \times 4$   $0\% \times 5$  diese Veranstaltung  
 $14\% \times 1$   $30\% \times 2$   $52\% \times 3$   $3\% \times 4$   $1\% \times 5$  Durchschnitt Informatik
- Aufgabe haben viel Spaß gemacht, interessant und realitätsnah, gut machbar, sehr gut auf die Vorlesung abgestimmt, schnelle Korrektur, Musterlösungen, 3 Sprachen zur Auswahl
- Immer noch vereinzelt Wunsch nach persönlichem Tutorat
- Lob für die Tutoren: Maya Schöchlin, Sebastian Holler, Daniel Tischer, Daniel Bindemann, Danny Stoll, Simon Selg
- Lob für die Arbeit des Assistenten: Axel Lehmann (SVN)

## ■ Materialien / Online Support

- **Hilfreich:** 81% trifft voll zu, 10% trifft zu, 9% ok
- Konsumiert durch Anwesenheit / Aufzeichnungen / Folien:  
23% Anw, 36% Aufz, 30% beides, 11% Folien diese Veranstaltung  
36% Anw, 16% Aufz, 18% beides, 30% Folien Durchschnitt Informatik
- Für die Aufzeichnungen und den Schnitt großen Dank an:  
**Frank Dal-Ri (Technik) & Alexander Monneret (Schnitt)**
- Sehr aktives + hilfsbereites Forum, ausgezeichnetes Team,  
Deluxe-Videoaufzeichnungen, diesmal sogar Live-Stream
- Suche im Forum könnte besser sein → **stimmt**

# Evaluationsergebnisse 6/8

---

## ■ Kritik / Wünsche von letztem Mal (SS 2015)

- Ein Tutorat gelegentlich bzw. am Anfang

Maßnahme: Fragestunde in Vorlesung 3b, früh darauf hingewiesen, dass persönliches Treffen mit Tutor möglich

- Zu viel Zeit für Malen / Farbauswahl ... eine Mehrheit fand das aber im Gegenteil gerade gut

Maßnahme: Zeichnungen teilweise vorbereitet, auch Programm nicht mehr immer "from scratch" geschrieben

- In der Mitte des Semesters teilweise zu leicht

Maßnahme: Aufgaben angepasst und auf gleichmäßiges Niveau geachtet + diverse Zusatz- bzw. Bonusaufgaben

- Was wir besser / weiter gut gemacht haben
  - Maßnahmen von der vorherigen Folie
  - Drei Programmiersprachen zur Auswahl (Python, Java, C++), mit gleichwertigen Vorlagen für die Ü-Blätter
  - Mehr "freiere" Übungsblätter, wo man sich selber einen Algorithmus überlegen / selber weiterdenken muss
  - Zusatz- bzw. Bonusaufgaben für die Unterforderten
  - Zeitmanagement ... ziemlich gut dieses Semester
  - Zahlreiche (meistens kleine) Fehler korrigiert + Folien teilweise besser strukturiert oder weiter perfektioniert
  - Etwas härterer Umgang mit Plagiaten

## ■ Geplante Verbesserungen für nächstes Mal

- Ein **Präsenztutorat** in der zweiten oder dritten Woche
- Von Beginn sehr klar definieren + aufschreiben, was ein **Plagiat** ist und was nicht und was der Sinn der Regel ist
- **LaTeX-Vorlage** für die Abgabe der Übungsblätter
- Überlegen, wie man die (sehr wenigen) Unterforderten mehr fordern kann ohne die anderen abzuhängen

Aber nicht ganz sicher, ob diese wirklich unterfordert sind

- Folien überarbeiten + verbleibende Fehler ausmerzen
- Ich mache mir zu jeder Vorlesung ausführliche Notizen
- Interaktivere Formate ausprobieren, zumindest ab und zu

## ■ Termin + Punkte

- Am **29. August 2017** von 14 – 17 Uhr, Gebäude 101  
**Wir teilen Ihnen noch mit, wer wo sitzt (Forum + Mail)**
- 6 Aufgaben a 20 Punkte, wir zählen die besten 5
- Also maximal 100 Punkte

## ■ Endnote

- Ergibt sich linear aus der Punktzahl in der Klausur
  - 50 – 54: 4.0; 55 – 59: 3.7; 60 – 64: 3.3
  - 65 – 69: 3.0; 70 – 74: 2.7; 75 – 79: 2.3
  - 80 – 84: 2.0; 85 – 89: 1.7; 90 – 94: 1.3
  - 95 – 100: 1.0

# Klausur 2/6

## ■ Modus

- Die Klausur ist **open book** : sie dürfen Bücher, Papier, usw. im Gesamtgewicht bis zu **527 kg** mitbringen

Aber **bitte sparsam beim Ausdrucken der Folien sein !**

- Elektronische Geräte jeder Art sind nicht gestattet
- Außerdem bitte mitbringen:  
**Studiausweis, Buntstifte, Gehirn**



## ■ Drei Typen von Aufgaben

- **Typ 1:** Einen Algorithmus, oder eine Variante davon, an einem Beispiel nachvollziehen  
*Siehe Buntstifte, aber bitte kein Rot verwenden*
- **Typ 2:** Kleineres Programm schreiben oder gegebenes Programm verstehen und/oder die Laufzeit analysieren
- **Typ 3:** Kleinere Rechenaufgaben oder Beweise oder Denkaufgaben ... *siehe Gehirn*
- Auf den nächsten drei Folien ein Beispiel zu jedem Typ

*Auf dem Wiki finden Sie die Klausuren vom SS 2015 und SS 2013 + drei ältere Klausuren zu AlgoDat für ESE*

# Klausur 4/6

## ■ Beispielaufgabe vom Typ 1 (Algorithmus am Beispiel)

- Klausur SS 2017, Aufgabe 3.1: Zeichnen Sie den Zustand eines binären Heaps nach Einfügen von 5, 4, 3, 2, 1

OPERATION

insert(5)

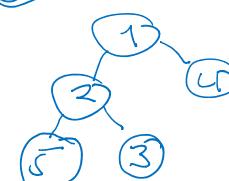
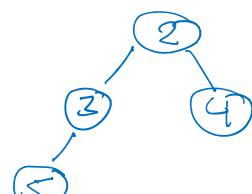
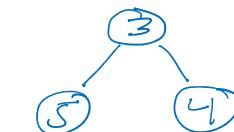
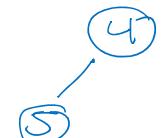
insert(4)

insert(3)

insert(2)

insert(1)

BAUM



FELD

|   |   |  |
|---|---|--|
| 0 | 1 |  |
| x | 5 |  |

|   |   |   |
|---|---|---|
| 0 | 1 | 2 |
| x | 4 | 5 |

|   |   |   |   |
|---|---|---|---|
| x | 3 | 5 | 4 |
|---|---|---|---|

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| x | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| x | 1 | 2 | 4 | 5 | 3 |

## ■ Beispielaufgabe vom Typ 2 (Programm + Laufzeit)

- Klausur SS 2017, Aufgabe 3.2: Funktion, die für einen (als Feld) gegebenen Heap überprüft, ob Heapeigenschaft gilt
- Klausur SS 2017, Aufgabe 3.3: Die Laufzeit der Funktion

```
def checkHeapProperty (deap) :
 for i in range (2, len(deap)) :
 child_value = deap [i]
 parent_value = deap [int(i/2)]
 if child_value < parent_value :
 return False

 return True
```

n = Größe von dem Feld  
Schleife läuft  $n-2$  mal, Zustand viele Operationen  
→  $\Theta(n)$  pro Durchlauf

## ■ Beispielaufgabe vom Typ 3 (Rechenaufgabe / Beweis)

- Klausur SS 2017, Aufgabe 6.3: Wahrscheinlichkeit, dass bei zufälligen Pivot bei Quicksort beide Teile  $\geq n/4$

Dabei:  $n$  = Feldgröße,  $n$  durch 4 teilbar, alle Zahlen sind verschieden, der Pivot gehört zu keinem der beiden Teile

Beispiel  $n = 8$  :  $\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \uparrow & \uparrow \\ \text{selekt} & & \text{gut} & & & & \text{schlecht} & \\ \end{array}$   $n/4 = 2$

(beide Teile  $\geq n/4$ )

Elemente in sortierter Reihenfolge:  $x_1 < x_2 < \dots < x_m$

Private aus  $x_1, \dots, x_{n+4}$ : Teil 1  $\leftarrow \mathcal{M}_4$

Rivat aus  $x_{m-n}, \dots, x_m$ : Teil 2 <  $\frac{M}{4}$

Samst

$$\Rightarrow w_{\text{real}} = \frac{1}{2} \pi$$

## ■ Wie wir arbeiten

- Wir lösen praktisch relevante Probleme

Routenplanung auf Google Maps, "Search As You Type",  
Semantische Suche, "Question Answering"

- Wir machen unsere Software + Ergebnisse verfügbar

Dazu braucht man gute Software, gute Dokumentation,  
gute Benutzerschnittstellen, usw.

- Theorie als Werkzeug, nicht um der Theorie willen

Aber wichtig: ohne theoretisches Verständnis beim Lösen  
komplexer Problem nur "Rumgehacke" und "Raterei"

## ■ Betreuung

- Ähnlich wie in der Vorlesung

Sehr gute Infrastruktur und Support

Abgesehen davon sollten Sie sehr unabhängig arbeiten

Insbesondere haben Sie bei uns viel Freiheit

Gut geeignet für enthusiastische Leute, die gerne praktische Probleme lösen und Sachen gemacht kriegen wollen

## ■ Maschinelles Lernen

- Wir benutzen zunehmend maschinelles Lernen zur Lösung vieler (nicht aller) unserer Probleme

Nicht weil es in Mode ist, sondern weil es praktisch ist

Es ist ziemlich offensichtlich, dass maschinelles Lernen der Ansatz der Wahl für komplexe Probleme ist, wie z.B. die Verarbeitung / das Verständnis von natürlicher Sprache

- Beim maschinellen Lernen geht es weniger um ausgefeilte Algorithmen sondern um ein tiefes Verständnis der grundlegenden Prinzipien und wie und warum sie funktionieren

Ein Verständnis der Algorithmen und vor allem die "Denke" aus der Info II Vorlesung ist dabei Grundvoraussetzung

# Vorstellung Lehrstuhl 4/6

---

## ■ Aktuelle Projekte und Demos

- Routenplanung (Teil von Google Maps) [demo](#)
- Visualisierung des weltweiten ÖPNV (Travic) [demo](#)
- Automatisches Malen von ÖPNV-Karten (Loom) [demo](#)
- Interaktive Semantische Suche (Broccoli) [demo](#)
- Large-Scale SPARQL+Text Suche (QLever) [demo](#)
- Question Answering (Aqqu) [demo](#)
- Question Completion [demo](#)
- Text extraction from PDF (Icecite) [paper](#)

## ■ Vorlesungen

- **WS 17/18: Information Retrieval** (Spezialvorlesung)

Alles was man braucht, um eine Suchmaschine (oder verwandte Services) gemäß dem Stand der Kunst zu bauen

Potpourri aus vielen Techniken und Gebieten: Algorithmen, Kodierungstheorie, Web Apps, Maschinelles Lernen, ...

- **SS 2018: Programmieren in C++** (Grundvorlesung)

C/C++ lernen von Grund auf

Inklusive Makefiles, Compiler, Linker, Debugger, Templates, und das ganze Drumherum

# Vorstellung Lehrstuhl 6/6

---

## ■ Projekte + Abschlussarbeiten

- Auf unserem Wiki:

[Liste von Themen + Informationen zum Ablauf](#)

Keine festen Zeiten, einfach nachfragen, siehe Folie 18

Die Wiki-Seite vorher gründlich durchlesen

[Leitfaden zum Schreiben einer Abschlussarbeit](#)