UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

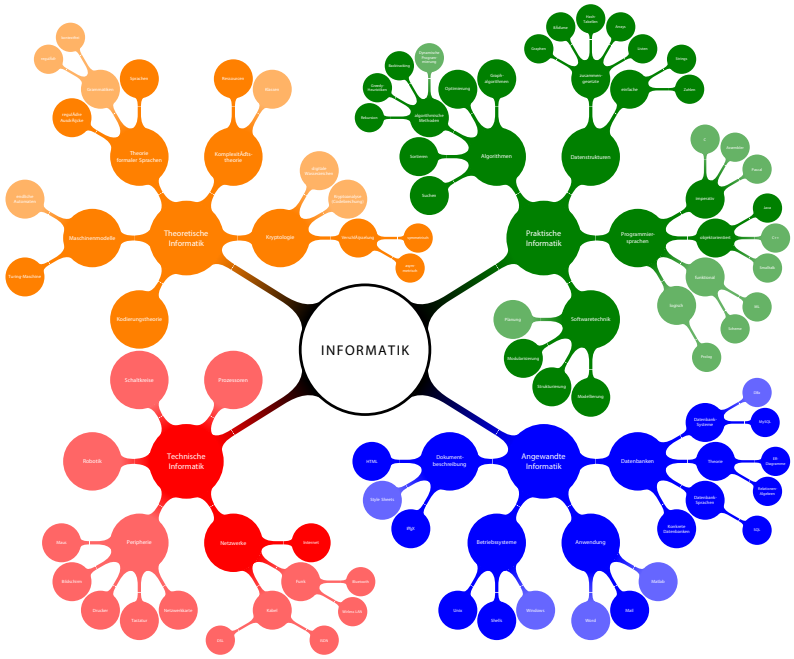
Vorlesungsskript

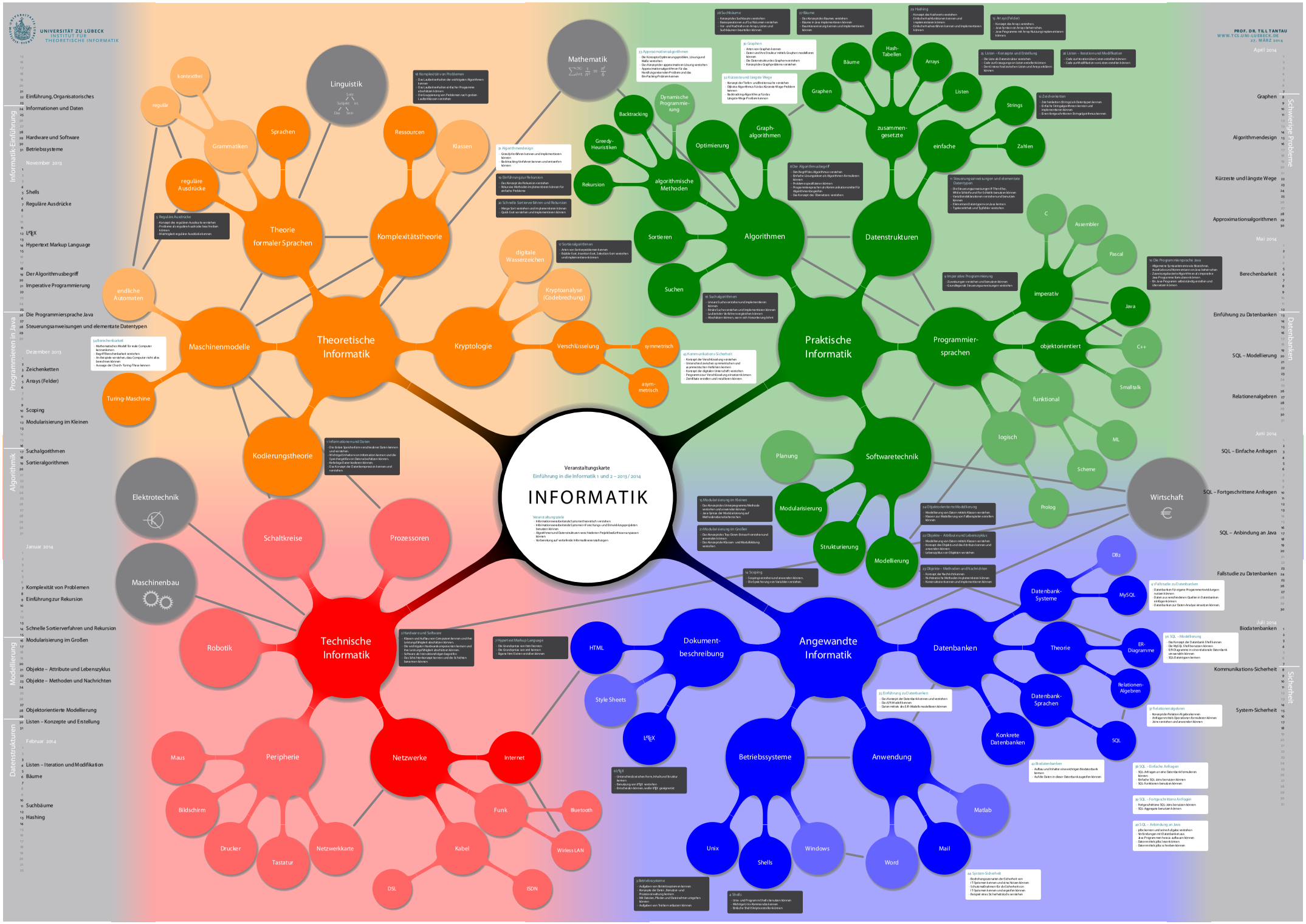
Einführung in die Informatik 1 und 2

CS1012, Wintersemester 2013, Sommersemester 2014

Fassung vom 23. Juni 2014

Till Tantau




UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR THEORETISCHE INFORMATIK
PROF. DR. TILL TANTAU
WWW.TCS.UNI-LUEBECK.DE
27. MÄRZ 2014
Veranstaltungskarte
Einführung in die Informatik 1 und 2 – 2013 / 2014
INFORMATIK
Veranstaltungsziele
Informatikkonzepte anhand Systemen theoretisch verstehen
Informatikkonzepte anhand Systemen in Forschungs- und Entwicklungsprojekten benutzen können
Algorithmen und Datenstrukturen wie fachlichen Projektbedürfnissen anpassen können
Vorbereitung auf weiterführende Informatikveranstaltungen
Informatik-Einführung
Einführung, Organisatorisches
Informationen und Daten
Hardware und Software
Betriebssysteme
November 2013
Shells
Reguläre Ausdrücke
LaTeX
Hypertext Markup Language
Der Algorithmusbegriff
Imperative Programmierung
Programmieren in Java
Die Programmiersprache Java
Steuerungsanweisungen und elementare Datentypen
Dezember 2013
Zeichenketten
Arrays (Felder)
Scoping
Modularisierung im Kleinen
Algorithmen
Suchalgorithmen
Sortieralgorithmen
Januar 2014
Komplexität von Problemen
Einführung zur Rekursion
Schnelle Sortierverfahren und Rekursion
Modularisierung im Großen
Modellierung
Objekte – Attribute und Lebenszyklus
Objekte – Methoden und Nachrichten
Objektorientierte Modellierung
Listen – Konzepte und Erstellung
Datenstrukturen
Februar 2014
Listen – Iteration und Modifikation
Bäume
Suchbäume
Hashing
Schwierige Probleme
Graphen
Algorithmendesign
Kürzeste und längste Wege
Approximationsalgorithmen
Mai 2014
Berechenbarkeit
Datenbanken
Einführung zu Datenbanken
SQL – Modellierung
Relationenalgebren
Juni 2014
SQL – Einfache Anfragen
SQL – Fortgeschrittene Anfragen
SQL – Anbindung an Java
Fallstudie zu Datenbanken
Juli 2014
Biodatenbanken
Sicherheit
Kommunikations-Sicherheit
System-Sicherheit
Mathematik
Linguistik
Wirtschaft
Elektrotechnik
Maschinenbau
Theoretische Informatik
Praktische Informatik
Technische Informatik
Angewandte Informatik
Maschinenmodelle
Theorie formaler Sprachen
Komplexitätstheorie
Kodierungstheorie
Kryptologie
Algorithmen
Datenstrukturen
Softwaretechnik
Programmier-sprachen
Betriebssysteme
Datenbanken
Anwendung
Netzwerke
Peripherie
Dokument-beschreibung
Datenbank-Systeme
Datenbank-Sprachen
Theorie
Konkrete Datenbanken
Relationen-Algebren
ER-Diagramme
algorithmische Methoden
Graph-algorithmen
Optimierung
Sortieren
Suchen
zusammen-gesetzte
einfache
imperativ
funktional
objektorientiert
logisch
Sprachen
Grammatiken
Ressourcen
Klassen
Verschlüsselung
symmetrisch
asym-metrisch
digitale Wasserzeichen
Kryptoanalyse (Codebrechung)
Backtracking
Dynamische Programmie-rung
Greedy-Heuristiken
Rekursion
Graphen
Bäume
Hash-Tabellen
Arrays
Listen
Strings
Zahlen
C
Assembler
Pascal
Java
C++
Smalltalk
ML
Scheme
Prolog
endliche Automaten
Turing-Maschine
kontextfrei
regulär
reguläre Ausdrücke
Planung
Modularisierung
Strukturierung
Modellierung
Robotik
Schaltkreise
Prozessoren
Maus
Bildschirm
Drucker
Tastatur
Netzwerkkarte
Internet
Funk
Kabel
Bluetooth
Wirless LAN
DSL
ISDN
HTML
Style Sheets
LaTeX
Unix
Shells
Windows
Word
Mail
Matlab
DB2
MySQL
SQL
Kapitel 1 Einführung in die Informatik
Kapitel 2 Algorithmen
Kapitel 3 Datenstrukturen
Kapitel 4 Modellierung
Kapitel 5 Programmieren in Java
Kapitel 6 Algorithmik
Kapitel 7 Datenbanken
Kapitel 8 Sicherheit
Kapitel 9 Schwierige Probleme
Kapitel 10 Modellierung
Kapitel 11 Technische Informatik
Kapitel 12 Angewandte Informatik
Kapitel 13 Praktische Informatik
Kapitel 14 Theoretische Informatik

Inhaltsverzeichnis

Vorwort 1

Teil I
Einführung in die Informatik

1 Informationen und Daten

1.1 Bits und Bytes 4
1.1.1 Bits 4
1.1.2 Bytes 4

1.2 Kodieren von Information 5
1.2.1 Kodierung von Zahlen 5
1.2.2 Kodierung von Texten 5
1.2.3 Kodierung von Bildern 6
1.2.4 Kodierung von Filmen 7

1.3 Datenkompression 7

Übungen zu diesem Kapitel 8

2 Hardware und Software

2.1 Arten von Computern 11
2.1.1 Was ist ein Computer? 11
2.1.2 Klassifikation von Computern 11

2.2 Aufbau von Computern 12
2.2.1 Die CPU 12
2.2.2 Der Hauptspeicher 13
2.2.3 Die Festplatte 14
2.2.4 Der Graphikprozessor 14

2.3 Aufbau von Software 15
2.3.1 Schicht 1: BIOS 15
2.3.2 Schicht 2: Betriebssystem 16
2.3.3 Schicht 3: Graphische Oberfläche 16
2.3.4 Schicht 4: Anwendungen 16

Übungen zu diesem Kapitel 17

3 Betriebssysteme

3.1 Einführung 21
3.1.1 Was ist ein Betriebssystem? 21
3.1.2 Welche Betriebssysteme gibt es? 21

3.2 Ressourcen 22
3.2.1 Druckauftragsverwaltung 22
3.2.2 Dateiverwaltung 23
3.2.3 Nutzerverwaltung 24
3.2.4 Prozessverwaltung 25

3.3 Aufbau von Betriebssystemen 25
3.3.1 Schon wieder Schichten 25
3.3.2 Untere Schicht: Treiber 25
3.3.3 Obere Schicht: Shells, Systemaufrufe . . 26

Übungen zu diesem Kapitel 27

4 Shells

4.1 Einführung zu Shells 29

4.2 Unix-Shell-Befehle 30
4.2.1 Dateiverwaltung 31
4.2.2 Rechteverwaltung 32
4.2.3 Nützliche Befehle 33

4.3 Unix-Shell-Programmierung 34
4.3.1 Um- und Weiterleitung 34
4.3.2 Shell-Skripte 35

Übungen zu diesem Kapitel 36

5 Reguläre Ausdrücke

5.1 Einführung 42
5.1.1 Muster-Suche in Texten 42
5.1.2 Theoretischer Hintergrund 42
5.1.3 Praktischer Hintergrund 43

5.2 Reguläre Ausdrücke 43
5.2.1 Die einfachsten Ausdrücke 43
5.2.2 Die Alternative 44
5.2.3 Die Verkettung 44
5.2.4 Der Kleene-Stern 45
5.2.5 Reguläre und arithmetische Ausdrücke . 45

5.3 Anwendungen 46
5.3.1 Einsatzgebiete 46
5.3.2 Referenz: Reguläre Ausdrücke in Grep . 47

Übungen zu diesem Kapitel 48

Teil II
Seitenbeschreibungssprachen

6 LaTeX

6.1 Inhalt – Struktur – Form 52
6.1.1 Drei Sichten auf einen Text 52
6.1.2 Beschreibungssprachen 53
6.1.3 Das Beispiel LaTeX 53

6.2 Gliederung von Dokumenten in LaTeX 53
6.2.1 Dokumentklassen 53
6.2.2 Die Präambel 54
6.2.3 Abschnitte und Paragraphen 54
6.2.4 Umgebungen 55

6.3 Typographisches und Graphiken in LaTeX 56
6.3.1 Schriftarten 56
6.3.2 Tabellen 56
6.3.3 Mathematik 57
6.3.4 Graphiken 57

6.4 Vorträge erstellen in LaTeX 58

7 Hypertext Markup Language

7.1 HTML 61
7.1.1 Einführung 61
7.1.2 Aufbau von HTML-Seiten 61
7.1.3 Aufbau von Tags 63

7.2 XML 63
7.2.1 Einführung 63
7.2.2 Vergleich mit HTML 64
7.2.3 Wohlgeformte Texte 64

Übungen zu diesem Kapitel 65

Teil III
Programmieren in Java

8 Der Algorithmusbegriff

8.1 Algorithmen 69
8.1.1 Geschichte 69
8.1.2 Definition 69
8.1.3 Steuerungsanweisungen 70

8.2 Spezifikationen 71

8.3 Programmiersprachen 71
8.3.1 Wozu dienen Programmiersprachen? . . 71
8.3.2 Übersetzer 73

9 Imperative Programmierung

9.1 Arten von Programmiersprachen 75

9.2 Berechnungen 77
9.2.1 Variablen 77
9.2.2 Ausdrücke 77
9.2.3 Zuweisungen 78

9.3 Steuerungsanweisungen 78
9.3.1 Komposition 78
9.3.2 Alternativen 78
9.3.3 While-Schleife 79

Übungen zu diesem Kapitel 80

10 Die Programmiersprache Java

10.1 Überblick über Java 83

10.2 Die Java-Syntax 84
10.2.1 Vom Algorithmus zum Programm 84
10.2.2 Bezeichner 84
10.2.3 Zahlen, Ausdrücke, Zuweisungen 84
10.2.4 Variablendeklaration 86
10.2.5 Formatierung und Kommentare 87

10.3 Java-Übersetzer 88
10.3.1 Übersetzerkonzept bei Java 88
10.3.2 Übersetzung in BlueJ und Eclipse 88

Übungen zu diesem Kapitel 89

11 Steuerungsanweisungen und elementate Datentypen

11.1 Steuerungsanweisungen 92
11.1.1 If-Then-Else 92
11.1.2 While-Schleife 93
11.1.3 For-Schleife 94

11.2 Datentypen 95
11.2.1 Der Begriff des Typs 95
11.2.2 Die Nutzen von Typen 95
11.2.3 Arten von Typen 95
11.2.4 Javas Datentypen 95

11.3 Typisierung 96
11.3.1 Typisierung von Variablen 96
11.3.2 Typisierung von Ausdrücken 96
11.3.3 Typfehler 97

Übungen zu diesem Kapitel 98

12 Zeichenketten

12.1 Zeichenketten 101
12.1.1 Der Begriff der Zeichenkette 101
12.1.2 Der Datentyp String 101
12.1.3 Die Datenstruktur String 102

12.2 Algorithmen auf Zeichenketten 103
12.2.1 Umdrehen 103
12.2.2 Trimmen 104
12.2.3 Naives Suchen 104
12.2.4 Intelligentes Suchen 105

Übungen zu diesem Kapitel 106

13 Arrays (Felder)

13.1 Benutzung von Arrays 109
13.1.1 Einführung 109
13.1.2 Array-Typen 110
13.1.3 Erzeugung von Arrays 110
13.1.4 Zugriff auf Arrays 111
13.1.5 Algorithmen auf Arrays 111

13.2 Speicherung von Arrays 112
13.2.1 Verweistypen 112
13.2.2 Zuweisung und Vergleich von Verweistypen 113

Übungen zu diesem Kapitel 114

14 Scoping

14.1 Problem: Variablennamen 117
14.1.1 Name-Clashing 117
14.1.2 Verwendungsdauer 118

14.2 Lösung: Scoping 119
14.2.1 Der Begriff des Scopes 119
14.2.2 Gültigkeit von Variablen 119
14.2.3 Sichtbarkeit von Variablen 120
14.2.4 Speicherung von Variablen 121

Übungen zu diesem Kapitel 122

15 Modularisierung im Kleinen

15.1 Sich wiederholender Code 124
15.1.1 Das Problem 124
15.1.2 Die Lösung 125

15.2 Methoden in Java 125
15.2.1 Die Syntax im Überblick 125
15.2.2 Syntax: Methodenname 126
15.2.3 Syntax: Parameter 126
15.2.4 Syntax: Rückgabetyp 127
15.2.5 Syntax: Aufruf 128

Übungen zu diesem Kapitel 129

Teil IV
Algorithmik

16 Suchalgorithmen

16.1 Die Problemstellung 135

16.2 Lineare Suche 135
16.2.1 Idee . . . 135
16.2.2 Implementation . . . 135
16.2.3 Laufzeit . . . 136
16.2.4 Anwendung . . . 136

16.3 Binäre Suche 137
16.3.1 Idee . . . 137
16.3.2 Implementation . . . 137
16.3.3 Laufzeit . . . 139
16.3.4 Vorsortierung . . . 139

Übungen zu diesem Kapitel 140

17 Sortieralgorithmen

17.1 Die Problemstellung 142
17.1.1 Motivation . . . 142
17.1.2 Problemvarianten . . . 142

17.2 Sortieralgorithmen 143
17.2.1 Bubble-Sort . . . 143
17.2.2 Selection-Sort . . . 144
17.2.3 Insertion-Sort . . . 146

17.3 *Untere Schranke für die Vergleichszahl 147

Übungen zu diesem Kapitel 148

18 Komplexität von Problemen

18.1 Die Laufzeit von Programmen 150
18.1.1 Welches Verfahren ist schneller? . . . 150
18.1.2 Laufzeitbestimmung I . . . 150

18.2 *O*-Klassen 151
18.2.1 Idee . . . 151
18.2.2 Definition . . . 151
18.2.3 Laufzeitbestimmung II . . . 153

18.3 Einfache und schwierige Probleme 153
18.3.1 Einfache Probleme . . . 153
18.3.2 Schwierige Probleme . . . 154

Übungen zu diesem Kapitel 155

19 Einführung zur Rekursion

19.1 Rekursion 158
19.1.1 Der Begriff . . . 158
19.1.2 Java-Syntax der Rekursion . . . 159

19.2 Beispiele 159
19.2.1 Fakultät . . . 159
19.2.2 Fibbonacci-Zahlen . . . 160
19.2.3 Die Türme von Hanoi . . . 160
19.2.4 Binäre Suche . . . 161

19.3 *Wechselseitige Rekursion 161
19.3.1 Tic-Tac-Toe . . . 161
19.3.2 Schach . . . 162

Übungen zu diesem Kapitel 162

20 Schnelle Sortierverfahren und Rekursion

20.1 Wozu schnelleres Sortieren? 167

20.2 Merge-Sort 167
20.2.1 Idee . . . 167
20.2.2 Implementation . . . 168
20.2.3 Analyse . . . 169

20.3 Quick-Sort 170
20.3.1 Idee . . . 170
20.3.2 Implementation . . . 171
20.3.3 Analyse . . . 171

20.4 *Linearzeit-Sortierung 172
20.4.1 Idee . . . 172
20.4.2 Implementation . . . 172
20.4.3 Analyse . . . 173

Übungen zu diesem Kapitel 173

Teil V
Modellierung

21 Modularisierung im Großen

21.1 Modularisierung von Software 177
21.1.1 Probleme im Großen 177
21.1.2 Motivierende Problemstellung 177
21.1.3 Modularisierungsebenen 178
21.1.4 Softwareentwurf 178

21.2 Klassen 179
21.2.1 Der Begriff der Klasse 179
21.2.2 Java-Syntax von Klassen 179
21.2.3 Benutzung von Klassen 180
21.2.4 Zugriffsrechte 180

21.3 Pakete 181
21.3.1 Der Begriff des Pakets 181
21.3.2 Java-Syntax für Pakete 181
21.3.3 Benutzung von Paketen 182

22 Objekte – Attribute und Lebenszyklus

22.1 Klassen von Objekten 185
22.1.1 Was sind Objekte? 185
22.1.2 Begriff der Klasse 185
22.1.3 Syntax von Klassen 186

22.2 Lebenszyklus von Objekten 186
22.2.1 Erzeugung (Geburt) 186
22.2.2 Zugriff 187
22.2.3 Veränderung 188
22.2.4 Vernichtung (Tod) 188

22.3 Objekthierarchien 189
22.3.1 Objekte als Attribute 189
22.3.2 Beispiel: Der Zellkern 189
22.3.3 Beispiel: Knöpfe 190

Übungen zu diesem Kapitel 191

23 Objekte – Methoden und Nachrichten

23.1 Einführung zu Nachrichten 194

23.2 Syntax von Nachrichten 195
23.2.1 Verschicken 195
23.2.2 Verarbeiten 196
23.2.3 Antworten senden 197

23.3 Konstruktoren 197
23.3.1 Begriff des Konstruktors 197
23.3.2 Syntax von Konstruktoren 198

Übungen zu diesem Kapitel 200

24 Objektorientierte Modellierung

24.1 Einführung 203
24.1.1 Was ist Modellierung? 203
24.1.2 Erst Analyse, dann Entwurf 204

24.2 Analyse 205
24.2.1 Ziel 205
24.2.2 Vorgehen 205
24.2.3 Beispiel: Bibliothek 205

24.3 Entwurf 206
24.3.1 Ziel 206
24.3.2 Vorgehen 206
24.3.3 Beispiel: Bibliothek 206

Übungen zu diesem Kapitel 207

Teil VI
Datenstrukturen

25 Listen – Konzepte und Erstellung

25.1 Einleitung 214
25.1.1 Motivation zu Listen 214
25.1.2 Idee hinter Listen 214

25.2 Die Datenstruktur 215
25.2.1 Idee 215
25.2.2 Umsetzung in Java-Code 215

25.3 Operationen auf Listen 216
25.3.1 Einfügen am Anfang 216
25.3.2 Löschen am Anfang 217

Übungen zu diesem Kapitel 218

26 Listen – Iteration und Modifikation

26.1 Iteration 220
26.1.1 Idee . . . 220
26.1.2 Anwendung: Längenbestimmung . . . 221
26.1.3 Anwendung: Ausgabe aller Elemente . . 222
26.1.4 Anwendung: Map . . . 222
26.1.5 Anwendung: Suche . . . 222

26.2 Modifikation 223
26.2.1 Einfügen von Elementen . . . 223
26.2.2 Löschen von Elementen . . . 224
26.2.3 Verketten von Listen . . . 225

Übungen zu diesem Kapitel 225

27 Bäume

27.1 Motivation 229
27.1.1 Modellierung . . . 229
27.1.2 Bessere Datenstrukturen . . . 230

27.2 Begriffe 230
27.2.1 Der Begriff des Baumes . . . 231
27.2.2 Arten von Bäumen . . . 231

27.3 Bäume in Java 232
27.3.1 Beteiligte Klassen . . . 232
27.3.2 Konstruktion . . . 233
27.3.3 Einfügen und Löschen . . . 234
27.3.4 Traversierung . . . 234

Übungen zu diesem Kapitel 236

28 Suchbäume

28.1 Einführung 239
28.1.1 Problemstellung . . . 239
28.1.2 Idee des Suchbaumes . . . 240
28.1.3 Suchbäume in Java . . . 240

28.2 Operationen 240
28.2.1 Suchen . . . 240
28.2.2 Einfügen . . . 242
28.2.3 Löschen aus Suchbäumen . . . 243

28.3 Vergleich der Implementationen 245

Übungen zu diesem Kapitel 246

29 Hashing

29.1 Einführung 249
29.1.1 Motivation . . . 249
29.1.2 Die Idee . . . 249
29.1.3 Die Problemstellung . . . 250

29.2 Hashwerte 250
29.2.1 Was ist eine gute Hashfunktion? . . . 250
29.2.2 Standard-Hashfunktionen . . . 251

29.3 Hashverfahren 252
29.3.1 Idee . . . 252
29.3.2 Implementation . . . 253
29.3.3 Suche . . . 253
29.3.4 Einfügen und Löschen . . . 253

Teil VII
Algorithmen für schwierige Probleme

30 Graphen

30.1 Modellierung mit Graphen 257
30.1.1 Modellierung mittels Graphen . . . 257
30.1.2 Arten und Formalisierungen . . . 257
30.1.3 Graphprobleme . . . 258

30.2 Graphen in Java 259
30.2.1 Adjazenzmatrizen . . . 259
30.2.2 Adjazenzlisten . . . 259

30.3 Graphproblem: Minimale Gerüste 260
30.3.1 Problemstellung . . . 260
30.3.2 Anwendungen . . . 260
30.3.3 Optimierungsalgorithmus . . . 261

30.4 Graphproblem: Der Handlungsreisende 261
30.4.1 Problemstellung . . . 261
30.4.2 Anwendungen . . . 262

Übungen zu diesem Kapitel 264

31 Algorithmendesign

31.1 Was ist Algorithmendesign? 267

31.2 Greedy-Verfahren 267
31.2.1 Idee . . . 267
31.2.2 Beispiel: Münzproblem . . . 268
31.2.3 Beispiel: Bin-Packing . . . 268
31.2.4 Implementation . . . 269
31.2.5 Vor- und Nachteile . . . 269

31.3 Backtracking 270
31.3.1 Idee . . . 270
31.3.2 Beispiel: Hamilton'sches Kreisproblem . 270
31.3.3 Beispiel: Färbeproblem . . . 271
31.3.4 Implementation . . . 272
31.3.5 Vor- und Nachteile . . . 273

Übungen zu diesem Kapitel 274

32 Kürzeste und längste Wege

32.1 Einführung 276

32.2 Erreichbarkeitsproblem 276
32.2.1 Problemstellung . . . 276
32.2.2 Traversierung I: Breitensuche . . . 277
32.2.3 Traversierung II: Tiefensuche . . . 278
32.2.4 Vergleich . . . 280

32.3 Kürzeste Wege 280
32.3.1 Problemstellung . . . 280
32.3.2 Lösung I: Breitensuche . . . 280
32.3.3 Lösung II: Dijkstra-Algorithmus . . . 281

32.4 Längste Wege 282
32.4.1 Problemstellung . . . 282
32.4.2 Lösung: Backtracking . . . 283

Übungen zu diesem Kapitel 284

33 Approximationsalgorithmen

33.1 Optimierungsprobleme 287
33.1.1 Einführung . . . 287
33.1.2 Formalisierung . . . 287
33.1.3 Beispiele . . . 287
33.1.4 Maß und Güte . . . 288

33.2 Approximationsalgorithmen 288
33.2.1 Heuristiken und Approximation . . . 288
33.2.2 Handlungsreisenden-Problem . . . 289
33.2.3 Bin-Packing . . . 290

Übungen zu diesem Kapitel 291

34 Berechenbarkeit

34.1 Die Turing-Maschine 295
34.1.1 Was bedeutet »berechenbar«? . . . 295
34.1.2 Turings Ideen . . . 295

34.2 Die Church-Turing-These 297
34.2.1 Historischer Rückblick . . . 297
34.2.2 Die These . . . 299

34.3 Nichtberechenbares 300
34.3.1 Das Postsche Korrespondenzproblem . . 300
34.3.2 Das Busy-Beaver-Problem . . . 301
34.3.3 Kolmogorov-Komplexität . . . 302

Teil VIII
Datenbanken

35 Einführung zu Datenbanken

35.1 Datenbanksysteme 306
35.1.1 Was sind Datenbanken? . . . 306
35.1.2 Aufbau von Datenbanken . . . 307
35.1.3 Arten von Datenbanken . . . 307

35.2 Datenmodelle 308

35.3 Das E/R-Modell 308
35.3.1 Einführung . . . 308
35.3.2 Entitäten . . . 309
35.3.3 Attribute . . . 309
35.3.4 Relationships . . . 310

Übungen zu diesem Kapitel 311

36 SQL – Modellierung

36.1 Einführung zu SQL 314
36.1.1 SQL zur Kommunikation . . . 314
36.1.2 Die MySQL-Shell . . . 314
36.1.3 Erste Schritte . . . 315

36.2 Erstellen einer Datenbank 316
36.2.1 Vom E/R-Modell zur Datenbank . . . 316
36.2.2 Anlegen einer Datenbank . . . 316
36.2.3 Anlegen von Tabellen . . . 316
36.2.4 Löschen . . . 317

Übungen zu diesem Kapitel 318

37 Relationenalgebren

37.1 Einführung 320
37.1.1 Alles ist eine Tabelle . . . 320
37.1.2 Relationen-Algebren . . . 320

37.2 Operationen auf Relationen 321
37.2.1 Vereinigung und Schnitt 321
37.2.2 Selektion 321
37.2.3 Projektion 322
37.2.4 Kreuzprodukt 322
37.2.5 Join 323

Übungen zu diesem Kapitel 325

38 SQL – Einfache Anfragen

38.1 Einfache Anfragen 326
38.1.1 Grundaufbau 326
38.1.2 Verbesserte Ausgaben 327
38.1.3 Joins 329

38.2 Anfragen für Fortgeschrittene 331
38.2.1 Prädikate 331
38.2.2 Funktionen 332

39 SQL – Fortgeschrittene Anfragen

39.1 Aggregate 334
39.1.1 Die Idee 334
39.1.2 Die Aggregatsfunktionen 334
39.1.3 Gruppierung 335

39.2 Unteranfragen (Subqueries) 337
39.2.1 Unteranfragen als Tabellen 337
39.2.2 Unteranfragen als Werte 338
39.2.3 Unteranfragen mit Quantoren 338

42 Biodatenbanken

42.1 Überblick 341
42.1.1 Welche Daten speichern Biodatenbanken? 342
42.1.2 Wie speichern sie die Daten? 342
42.1.3 Wie komme ich an die Daten ran? 342

42.2 Flat-Files: Fallbeispiel Protein-Data-Bank 344

42.3 SQL: Fallbeispiel Ensembl 345

42.4 XML: Fallbeispiel Kyoto Encyclopedia of Genes and Genomes 347

Teil IX
Sicherheit

43 Kommunikations-Sicherheit

43.1 Ziele von IT-Sicherheit 351

43.2 Verschlüsselung 351
43.2.1 Ziele 351
43.2.2 Symmetrische Verschlüsselung 351
43.2.3 Asymmetrische Verschlüsselung 353
43.2.4 Das El-Gamal-Verfahren 353

43.3 Sichere E-Mail 355
43.3.1 Vertraulichkeit: Digitale Briefumschläge 355
43.3.2 Authentizität: Digitale Unterschriften . . 358
43.3.3 Echtheits-Zertifikate: Digitale Notare . . 360

43.4 Sicheres Surfen 363

Übungen zu diesem Kapitel 364

44 System-Sicherheit

44.1 Systemsicherheit 367
44.1.1 Was ist zu schützen? 367
44.1.2 Wovor ist zu schützen? 368
44.1.3 Welche Maßnahmen helfen? 369

44.2 Fallbeispiel eines Sicherheitslochs 369
44.2.1 Die Methode: SQL-Injection 369
44.2.2 Fiktives Beispiel: Firmen-Intranet 370
44.2.3 Reales Beispiel: www.doc.state.ok.us . . 371

Übungen zu diesem Kapitel 372

Anhang

Lösungen zu den Übungen 373

Vorwort

Willkommen! Welcome! Bienvenue! Nicht im Cabaret, sondern bei der Veranstaltung *Einführung in die Informatik*. Auch wenn es nicht ganz so unterhaltsam werden wird wie das legendäre Musical und auch wenn weder gesungen noch getanzt werden wird, so kann ich Ihnen trotzdem schon eine spannende Veranstaltung versprechen, bei der Sie viel lernen werden. Einiges wird für Sie im späteren Leben sehr nützlich sein, anderes überhaupt nicht.

Diese Veranstaltung wendet sich primär an »blutige Anfänger«. Das heißt nicht, dass wir Ihnen beibringen, wie man eine Maus bedient oder ein Fenster öffnet, denn das wissen Sie schon. Wir nehmen aber an, dass Sie keinerlei oder nur sehr geringe Programmierkenntnisse haben, und Sie brauchen auch von Unix-Shells noch nie gehört zu haben. Wir versuchen auch an diejenigen unter Ihnen zu denken, denen das Programmieren schon jetzt leicht fällt: Wann immer möglich stellen wir Aufgaben mit verschiedenen Niveaus und oft können Sie sich aussuchen, welche Aufgabe Sie lösen möchten. Bei einigen der schweren würde wohl auch so manch ein professioneller Programmierer ins Schwitzen kommen.

Informatik ist kein »Lernfach« für das man viel auswendig lernen muss. Informatik ist vielmehr ein Fach, bei dem es darauf ankommt, bestimmte Fertigkeiten wie das Programmieren zu beherrschen. Wie alle Fertigkeiten kann man diese nicht bloßes Zuhören oder Zusehen erlernen, man muss vielmehr *ausprobieren*. Deshalb besteht diese Veranstaltung zu fast gleichen Teilen aus Vorlesungen, in denen Konzepte erklärt werden, und Übungen, in denen – wie der Name schon sagt – kräftig geübt wird. Sie werden sogar schon in den Vorlesungen im Rahmen von 5-Minuten-Aufgaben mit dem Üben beginnen können. Schließlich müssen Sie im Rahmen von Übungszetteln selbstständig Aufgaben bearbeiten; wobei wir Sie aber nicht alleine lassen, da die Aufgaben in den Tutorien vorbesprochen werden. Sie werden feststellen, dass die als »leicht« und in der Regel auch die als »mittel« eingestuften Aufgaben mit der Vorbereitung im Tutorium in der Tat mit vertretbarem Aufwand schaffbar sind.

Jedes Kapitel dieses Skripts entspricht einer Vorlesungsdoppelstunde und mit jedem Kapitel verfolge ich gewisse Ziele, welche Sie am Anfang des jeweiligen Kapitels genannt finden. Neben diesen etwas kleinteiligen Zielen gibt es auch folgende zentralen Veranstaltungsziele:

1. Sie sollen informationsverarbeitende Systeme theoretisch verstehen.
2. Sie sollen informationsverarbeitende Systeme in Forschungs- und Entwicklungsprojekten benutzen können.
3. Sie sollen Algorithmen und Datenstrukturen verschiedenen Projektbedürfnissen anpassen können.
4. Sie sollen vorbereitet werden auf vertiefende Informatikveranstaltungen.

Wahrscheinlich hört sich zumindest das Ziel betreffend »Algorithmen und Datenstrukturen« (was immer diese sein mögen) noch etwas kryptisch an, aber ich verspreche Ihnen, dass dies noch klar werden wird. Diese Veranstaltung soll sie *vorbereiten* auf alles, was Sie später mit Informatik im Rahmen Ihres Berufes und Ihrer Forschung zu tun haben werden. Sie werden am Ende dieser Veranstaltung vielleicht nicht »voll ausgebildete Informatiker« sein, Sie werden aber wissen »wie die Kisten funktionieren« und, was vielleicht noch wichtiger ist, »was sie alles nicht können«…

Ich wünsche Ihnen viel Spaß mit dieser Veranstaltung.

Till Tantau

Teil I

Einführung in die Informatik

Der Titel »Einführung in die Informatik« für diesen ersten Teil weckt Erwartungen, die hier nicht werden befriedigt werden können. Es geht in diesen ersten Vorlesungen zunächst darum, ein gewisses Grundverständnis dafür zu entwickeln wie Computer eigentlich (sehr grob gesprochen) funktionieren. Die praktischen Übungen sollen helfen, die »Angst vor der Maschine« zu verlieren. Wenn es sich um eine »Einführung in die Schwimmkunst« handeln würde, dann entspräche dieser Teil in etwa dem Seepferdchen: Sie werden noch keine Weltrekorde im Schmetterlingstil aufstellen können, aber Sie werden auch nicht mehr untergehen und werden gemütlich Ihre Bahnen ziehen können.

Ganz am Anfang geht es um die 10 Kinder der Informatik: Nullen und Einsen. Aus Sicht der Informatik ist nämlich alles ein Strom von Nullen und Einsen; das gilt für Zahlen (wo man sich das noch recht leicht vorstellen kann) ebenso wie für Texte (wo dies schon schwieriger ist) bis zu ganzen Filmen. Nachdem geklärt ist, worüber Computer den ganzen Tag reden (nämlich Nullen und Einsen), werden wir uns anschauen, wie sie dies machen, wie also Hard- und Software prinzipiell funktionieren. Zum Schluss soll es dann etwas praktischer werden: wir werden uns Betriebssysteme und Shells (welche zur Kommunikation mit Betriebssystemen dienen) genauer anschauen.

1-1

Kapitel 1
Informationen und Daten
Wie kommt das Video auf die Scheibe?

1-2

Lernziele dieses Kapitels

1. Die binäre Speicherform verschiedener Daten kennen und verstehen.
2. Wichtige Einheiten von Information kennen und die Speichergröße von Daten abschätzen können.
3. Beliebige Daten kodieren können.
4. Das Konzept der Datenkompression kennen und verstehen.

Inhalte dieses Kapitels

1.1	Bits und Bytes	4
1.1.1	Bits	4
1.1.2	Bytes	4
1.2	Kodieren von Information	5
1.2.1	Kodierung von Zahlen	5
1.2.2	Kodierung von Texten	5
1.2.3	Kodierung von Bildern	6
1.2.4	Kodierung von Filmen	7
1.3	Datenkompression	7
	Übungen zu diesem Kapitel	8

Worum es heute geht

Die Sicht eines Computers auf seine Umwelt ist recht schlicht: Alles, *wirklich alles,* ist eine Folge von Nullen und Einsen. Der schnöde Text einer Steuererklärung, das elektronische Geschnatter der Eingabe- und Ausgabesteuerungschips, die übelsten und die leuchtendsten Beispiele politischer Literatur, ja selbst ein Fünfzigerjahre-Herz-Schmerz-Film – für Computer sind sie alle nur Folgen von Nullen und Einsen, eine so aufregend wie die andere. In diesem ersten Kapitel geht es darum, wie Computer das Kunststück fertigbringen, solch unterschiedliche Arten von Dingen als Strom von Nullen und Einsen zu behandeln.

Folgen von Nullen und Einsen sind Information in Reinform, ungetrübt von Emotionen, (Fehl)Interpretationen oder Wünschen. Während uns später hauptsächlich die *Verarbeitung* von Information interessieren wird, ist in diesem Kapitel zunächst nur eine Eigenschaft von Information von Interesse: ihre Menge. Informationsmengen misst man in Bit, Byte, Kilo-, Mega- und Gigabyte; alles Größen von denen Sie sicherlich schon gehört haben. Es wird nicht wichtig sein, die genaue Anzahl an Bits in einem Gigabyte auswendig zu wissen; vielmehr wird es darum gehen, ein gewisses Gefühl dafür zu bekommen, wie viel beispielsweise ein Gigabyte eigentlich ist. So viel wie ein Buch? Wie ein Bakteriengenom? Wie das menschliche Genom? Wie ein Spielfilm? Ein Kurzfilm?

Zum Auftakt: Ein kleiner Film.

Regie

1.1 Bits und Bytes

1.1.1 Bits

1-4 **Das Bit ist die kleinste mögliche Informationseinheit.**
Computer arbeiten *digital*, das heißt, mit nur zwei Zuständen:

0	versus	1
Strom an	versus	Strom aus
Loch vorhanden	versus	Loch nicht vorhanden
Magnetisierung hoch	versus	Magnetisierung runter

Die Informationseinheit hierzu heißt *Bit* (*B*inary dig*it*).

1-5 **Mit Bitfolgen lassen sich mehr als nur zwei Möglichkeiten repräsentieren.**
Da man mit einem einzelnen Bit nicht viel speichern kann, benutzt man mehrere.

Beispiel: Die vier Nukleotide
Die vier Nukleotide können mit zwei Bits gespeichert werden:

00	$\hat{=}$	Adenin
01	$\hat{=}$	Cytosin
10	$\hat{=}$	Guanin
11	$\hat{=}$	Thymin

Mit 2 Bits kann man also 4 Dinge unterscheiden.
Mit 3 Bits kann man 8 Dinge unterscheiden.
Mit 8 Bits kann man 256 Dinge unterscheiden.

1-6 ✎ Zur Übung
Lösen Sie eine der folgenden Aufgaben. (Sie sind nach Schwierigkeit sortiert.)

1. Wie viele Bits braucht man zur Speicherung eines Codons?
2. Wie viele Dinge kann man mit Bitstrings der Länge *genau n* unterscheiden?
3. Wie viele Dinge kann man mit Bitstrings der Länge *höchstens n* unterscheiden?

1.1.2 Bytes

1-7 **Bitfolgen bestimmter Längen erhalten besondere Namen.**
Bitfolgen bestimmter Längen kommen oft vor und haben deshalb besondere Namen.

▶ Definition: Byte, Kilobyte
Ein Bitstring der Länge 8 heißt *Byte*.
Ein Bitstring der Länge 8192 heißt *Kilobyte* (kB).

Ein Kilobyte sind also $1024 = 2^{10}$ Byte.

Achtung
Korrekter ist es zu definieren, dass 1000 Byte eine Kilobyte bilden.
Deshalb definiert man manchmal genau dies und spricht von »Kibibyte«, wenn man 1024 Byte meint (Kibi = Kilo + Binary).

1-8 ✎ Zur Übung
Schreiben Sie möglichst viele Ihnen bekannte Bezeichnungen für Bitfolgen bestimmter Längen auf.

1.2 Kodieren von Information

1.2.1 Kodierung von Zahlen

Wie kann man natürliche Zahlen kodieren?

1-9

Problemstellung
Natürliche Zahlen sollen als Bitfolgen dargestellt werden.

✎ **Zur Diskussion**
Welche Möglichkeiten gibt es? (Es gibt viele!)

Wie kann man ganze Zahlen kodieren?

1-10

Problemstellung
Nun sollen auch ganze Zahlen (auch negative) als Bitfolgen dargestellt werden.

Lösungsmöglichkeit
Füge am Anfang der Zahlrepräsentation ein zusätzliches Bit an, das das Vorzeichen angibt.
Beispiel: Aus $6 \mathrel{\hat{=}} 110$ werden $+6 \mathrel{\hat{=}} 0110$ und $-6 \mathrel{\hat{=}} 1110$.

Nachteil: Es gibt zwei Arten, die Null darzustellen ($+0$ und -0). Dies führt unweigerlich zu Programmfehlern.
Für weitere Details, insbesondere zur Kodierung reeller Zahlen, siehe die Literatur.

1.2.2 Kodierung von Texten

Wie kann man Texte kodieren?

1-11

Beispiel
Wie kodiert man AGGCCCTAAAGT?

Ein zweistufiges Verfahren:

1. Man kodiert die vier möglichen Basen mit zwei Bits:
   00 $\mathrel{\hat{=}}$ A
   01 $\mathrel{\hat{=}}$ C
   10 $\mathrel{\hat{=}}$ G
   11 $\mathrel{\hat{=}}$ T
2. Man kodiert Folgen von Basen als Folgen von Bitpaaren:
   Aus AAC wird 000001.
   Aus 010111 wird wieder CCT.

Wie kann man Texte kodieren?

1-12

Beispiel
Wie kodiert man “Sehr geehrte Damen und Herren”?

1. Man kodiert die möglichen Buchstaben mit mehreren Bits:
   000000 $\mathrel{\hat{=}}$ A
   000001 $\mathrel{\hat{=}}$ B
   000010 $\mathrel{\hat{=}}$ C
   ...
   011000 $\mathrel{\hat{=}}$ a
   011001 $\mathrel{\hat{=}}$ b
2. Wieder kodiert man Texte als Folgen von Bitstrings:
   Aus AAC wird 0000000000000010.

1-13

Wie viele Bits sollte man pro Zeichen benutzen?

Lösungsmöglichkeiten:

ASCII Man benutzt 7 Bits, was 128 möglichen Buchstaben entspricht. Damit kann man die Klein- und Großbuchstaben kodieren, Satzzeichen, und einige Sonderzeichen.

ASCII erw. Man benutzt 8 Bits, womit man 128 zusätzliche Zeichen bekommt. Diese werden je nach Land anders interpretiert.

UNICODE Man benutzt 16 Bits, womit man 65536 Zeichen kodieren kann. Dies reicht für alle von Menschen derzeit benutzten Zeichen (inklusive Chinesisch).

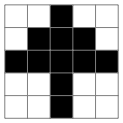
ISO 10646 Man benutzt etwa 21 Bits, was für alle jemals benutzte und jemals in der Zukunft benutzte Zeichen ausreicht.

1.2.3 Kodierung von Bildern

1-14

Wie kodiert man Bilder als Bitfolge?

Wie könnte man folgendes Bild kodieren?



Mögliche Lösung

0	0	1	0	0
0	1	1	1	0
1	1	1	1	1
0	0	1	0	0
0	0	1	0	0

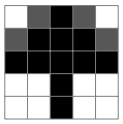
Kodierung: 0010001110111110010000100 (Zeilen hintereinander weg).

Besser: Stelle Breite und Höhe in den ersten vier Bytes voran.

1-15

Wie kodiert man Bilder als Bitfolgen?

Wie könnte man folgendes Bild kodieren?



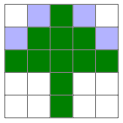
Mögliche Lösung

00	10	11	10	00
10	11	11	11	10
11	11	11	11	11
00	00	11	00	00
00	00	11	00	00

Kodierung: 0010111000101111111101111…
(Zwei Bit pro Pixel, Zeilen hintereinander weg).

1-16

Wie kodiert man Bilder als Bitfolgen?



Idee: Investiere ein Byte für den Rot-Anteil eines Bildpunktes, ein Byte für den Blau-Anteil und ein Byte für den Grün-Anteil.

Beispiel

Rot = $(255, 0, 0)$.
Blau = $(0, 0, 255)$.
Gelb = $(0, 255, 255)$.

1.2.4 Kodierung von Filmen

Wie kodiert man einen Film?

1-17

Problemstellung

Ein Film soll als Bitfolge kodiert werden.

Lösungsmöglichkeit

Ein Film besteht aus 25 Bildern pro Sekunde. Die Kodierung des Films ist dann die Kodierung der einzelnen Bilder hintereinander weg.

Ignorierte Probleme:

- Ton fehlt.
- Untertitel fehlen.
- Nicht fehlertolerant (Kratzer auf der DVD).
- …

1.3 Datenkompression

Eine Beispielrechnung, die ein Problem aufzeigt.

1-18

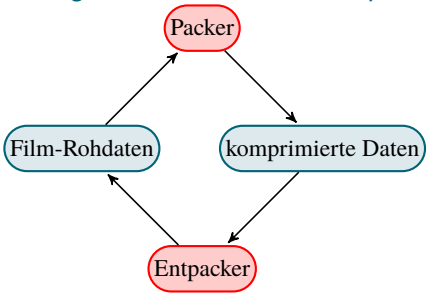
Nehmen wir an, wir wollen einen Spielfilm von 120 Minuten kodieren.

- Das sind 7200 Sekunden.
- Es gibt 25 Bilder pro Sekunde.
- Ein DVD-Bild hat 720 mal 480 Punkte.
- Ein Bildpunkt braucht 3 Byte oder 24 Bit.

Dies macht insgesamt $7200 \cdot 25 \cdot 720 \cdot 480 \cdot 3$ Byte $= 186624000000$ Byte ≈ 186 GB.
Eine DVD fasst 5 GB bis 16 GB, eine Bluray bis 50 GB.

Lösung des Problems: Datenkompression!

1-19



Es gibt zwei Arten von Packverfahren.

1-20

Verlustfreie Verfahren

Verlustfreie Verfahren packen Daten so, dass man aus den komprimierten Daten die Originaldaten exakt rekonstruieren kann.
Beispiel: `gzip`, `bzip2`, `zip`

Verlustbehaftete Verfahren

Verlustbehaftete Verfahren packen Daten so, dass man beim Auspacken nur ungefähr die Originaldaten bekommt.
Beispiel: `DivX`, `jpeg`, `mpeg`

Zusammenfassung dieses Kapitels

1-21

▶ Alle Daten sind Bitfolgen

Alle Daten, egal ob Zahlen, Texte, Bilder, Filme oder Musik werden als Bitfolgen gespeichert.

▶ Wichtige Einheiten von Information

Einheit	*Definition*	*Was man damit speichern kann*
Bit	Binary Digit / 0 oder 1	nicht viel
Byte	8 Bits	Buchstaben in unserem Alphabet
Kilobyte (kB)	1024 Byte oder 1000 Byte	kleines Icon, halbe Seite Text
Megabyte (MB)	1024 kB oder 1000 kB	ein Buch, eine Minute Musik, 1 Sekunde HD-Film
Gigabyte (GB)	1024 MB oder 1000 MB	ein Spielfilm, ein Genom
Terabyte (TB)	1024 GB oder 1000 GB	Wikipedia
Kibibyte	immer 1024 Byte	

▶ Datenkompression

- *Verlustfreie* Kompression kodiert (packt) Daten geschickt, so dass die Originaldaten *exakt* rekonstruiert (entpackt) werden können.
- *Verlustbehaftete* Kompression kodiert (packt) Daten geschickt, so dass sie nach dem Entpacken *genauso aussehen oder sich genauso anhören* wie das Original.

Übungen zu diesem Kapitel

Übung 1.1 Bildformate verstehen, mittel

Was passiert eigentlich, wenn ein Fax verschickt wird? Offenbar wird über die Telefonleitung eine digitalisierte Form des Faxtextes verschickt. In dieser Übung soll geklärt werden, wie viele Daten verschickt werden.

1. Bilden Sie zunächst Gruppen von zwei bis drei Personen. Jede Gruppe erhält vom Tutor einen gefaxten Text.
2. Versuchen Sie gemeinsam herauszufinden, wie viele Bits zur Übertragung einer Seite benötigt werden.
3. Auf einer altmodischen Telefonleitung kann man etwa 8 kBit/s übertragen. Wie lange würde die Übertragung der Ihnen vorliegenden Seite dauern?
   Vergleichen Sie Ihre Ergebnisse mit den Ergebnissen der anderen Gruppen. Ihr Tutor sammelt die Ergebnisse an der Tafel.
4. Wie könnte ein einfaches Kompressionsverfahren aussehen, um Faxe schneller zu übertragen? Diskutieren Sie diese Frage zunächst mit ihren Partnern, dann im Plenum.

Übung 1.2 Speichergrößen einschätzen, einfach

Bei der Speicherung von Bildern in einer Digitalkamera wird das Format JPEG verwendet. Eine moderne 8-Megapixel-Kamera benötigt damit etwa 2,5 MByte pro Bild bei mittlerer Bildqualität. Wie viele Bilder können Sie auf einer solchen Kamera mit einer 1 GByte-Speicherkarte unterbringen? Vergleichen Sie Ihr Ergebnis mit den Angaben eines Elektronikfachhändlers Ihres Vertrauens.

Übung 1.3 Speichergrößen einschätzen, einfach

Im menschlichen Genom kommen etwa $3 \cdot 10^9$ Basenpaare vor, die jeweils mit 2 Bit kodiert werden können. Wie lange dauert es, eine Datei mit dem kompletten Genom zu übertragen

1. über ISDN (64 kBit/s),
2. über die Internetanbindung der Universität (100 MBit/s),
3. über das Campusnetz (4 GBit/s)?

Übung 1.4 Bildformate verstehen, mittel

Viele gebräuchliche Röntgengeräte können noch nicht direkt digitale Bilder erzeugen. Stellen Sie sich vor, Sie hätten ein Röntgenbild zur digitalen Weiterverarbeitung mit einem DIN A2-Flachbettscanner eingescannt und nun ein Bild mit einer Auflösung von 1000 mal 1414 Bildpunkten vorliegen.

1. Wie groß (in mm) sind die kleinsten Details des Ursprungsbildes, die man auf dem gescannten Bild idealerweise noch erkennen kann?
2. Zur Weiterverarbeitung in unserem Bildverarbeitungsprogramm wollen wir das Bild nun im Format PGM *(Portable GrayMap)* speichern. PGM-Bilder sind Textdateien im ASCII-Code mit dem folgenden Aufbau:
   - In der 1. Zeile steht stets "P2"
   - In der 2. Zeile stehen Breite und Höhe des Bildes
   - In der 3. Zeile steht der höchste im Bild mögliche Grauwert
   - Daraufhin folgen die Grauwerte der einzelnen Bildpunkte, wobei jede Zeile einer Bildzeile entspricht

   Eine PGM-Datei kann also beispielsweise so aussehen:

```
P2
8 8
255
000 000 013 083 094 032 000 000
000 057 232 255 255 251 115 000
007 227 255 255 255 255 254 056
062 255 255 255 255 255 255 139
066 255 255 255 255 255 255 142
010 236 255 255 255 255 255 067
000 074 244 255 255 255 139 000
000 000 027 109 120 054 000 000
```

   Berechnen Sie den Speicherbedarf des Röntgenbildes im PGM-Format! Gehen Sie dabei davon aus, dass alle Grauwerte mit drei ASCII-Zeichen dargestellt werden wie im obigen Beispiel. Ein ASCII-Zeichen benötigt 8 Bit. Auch das Leerzeichen und der Zeilenvorschub sind ASCII-Zeichen.
3. Diskutieren Sie Vor- und Nachteile des PGM-Formats!

Übung 1.5 Fehlerkorrektur entwickeln, schwer

Die erste Version des ASCII-Codes verwendete zur Kodierung eines Zeichens 7 statt 8 Bits. Die kleinste Dateneinheit, die die meisten Computer speichern können, ist jedoch das Byte (8 Bit), so dass das übrige Bit für andere Zwecke eingesetzt werden kann.

Eine sinnvolle Nutzungsmöglichkeit ist, das achte Bit zur Erkennung und Behebung von Übertragungsfehlern zu nutzen. Können Sie eine Möglichkeit aufzeigen, wie dies geschehen könnte?

Prüfungsaufgaben zu diesem Kapitel

Übung 1.6 Speichergrößen einschätzen, leicht, original Klausuraufgabe, mit Lösung

Das menschliche Genom hat eine Länge von etwa $3 \cdot 1024 \cdot 1024 \cdot 1024$ Basenpaaren. Zur unkomprimierten Speicherung des Genoms in einem Computer werden für jedes Basenpaar 2 Bit Speicherplatz benötigt. Das Genom soll auf einem Rechner gespeichert werden, dessen Speicher nur in Einheiten von 2 Gigabyte aufgerüstet werden kann. Wie viele dieser Speichereinheiten benötigen Sie?

Übung 1.7 Speichergrößen einschätzen, leicht, original Klausuraufgabe, mit Lösung

Mit einer Digitalkamera wird ein 20 Sekunden langer Film bei einer Auflösung von 1024 mal 1024 Pixeln und einer Farbtiefe von 24 Bit pro Pixel aufgenommen.

1. Die Kamera liefert 25 Bilder pro Sekunde. Wie viel Speicherplatz belegt der Film, wenn keinerlei Datenkompression angewandt wird? Geben Sie das Ergebnis in Megabyte an.
2. Zur Datenkompression des Films stellt die Kamera zwei Verfahren zur Verfügung: Verfahren A mindert die Qualität leicht und komprimiert um Faktor 5, Verfahren B mindert die Qualität stark und komprimiert um Faktor 50. Sie wollen den Film in höchstmöglicher Qualität auf einer CD und auf einer DVD speichern. Müssen Sie den Film dazu komprimieren? Wenn ja, mit welchem Verfahren?

Übung 1.8 Speichergrößen abschätzen, leicht, typische Klausuraufgabe

Mittels Computer-Tomographie soll ein dreidimensionales Modell Ihres Körpers erstellt werden. Dabei wird pro Zentimeter Ihrer Körpergröße ein Schnitt aufgenommen. Die Schnitte haben eine Auflösung von 1024 mal 1024 Pixeln bei 1024 Graustufen. Das Modell soll unkomprimiert gespeichert werden.

Wie kodieren Sie die Pixel in Bytes? Wie viele Megabyte Speicherplatz benötigen Sie für Ihr Modell?

2-1

Kapitel 2
Hardware und Software

Was steckt in der Kiste? Wie arbeitet sie?

2-2

Lernziele dieses Kapitels

1. Klassen und Aufbau von Computern kennen und ihre Leistungsfähigkeit abschätzen können.
2. Die wichtigsten Hardwarekomponenten kennen und ihre Leistungsfähigkeit abschätzen können.
3. Software als Instruktionsfolgen begreifen
4. Das Schichtenkonzept kennen und die Schichten benennen können

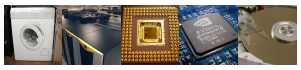
Inhalte dieses Kapitels

2.1	Arten von Computern	11
2.1.1	Was ist ein Computer?	11
2.1.2	Klassifikation von Computern	11
2.2	Aufbau von Computern	12
2.2.1	Die CPU	12
2.2.2	Der Hauptspeicher	13
2.2.3	Die Festplatte	14
2.2.4	Der Graphikprozessor	14
2.3	Aufbau von Software	15
2.3.1	Schicht 1: BIOS	15
2.3.2	Schicht 2: Betriebssystem	16
2.3.3	Schicht 3: Graphische Oberfläche	16
2.3.4	Schicht 4: Anwendungen	16
	Übungen zu diesem Kapitel	17

Worum es heute geht

»Computer«. Ein dreisilbiges Wörtchen, zwar aus dem Englischen, es könnte aber genauso gut aus einer altehrwürdigen deutschen Seemannsmundart stammen, würde man es seiner Aussprache entsprechend »Kompjuta« schreiben. Dieses Wort schafft es, bei Menschen so unterschiedliche Gefühle auszulösen wie kaum ein anderes: Computer werden geliebt, gehasst, gefürchtet, ignoriert, erlitten, bedauert oder auch einfach nur benutzt. Dies liegt daran, dass sie uns in unserem Alltag ständig begleiten, in aller Regel ohne dass wir es merken – die Probleme fangen meistens an, wenn wir sie bemerken. Sie gibt es auch in so vielen Ausführungen, dass schon eine Klassifikation schwer fällt; grob kann man sie nach »Rechenkraft« unterteilen. Am unteren Ende der Skala finden wir Rechner, die wenig mehr sind als elektronische Rechenschieber (ohne an dieser Stelle Rechenschieber in irgendeiner Weise herabwürdigen zu wollen), über die typischen »Personalcomputer« wie man sie in jedem Büro findet bis zu echten Superrechnern.

Wir fürchten oft Dinge, die wir nicht verstehen, die aber Einfluss auf uns haben (scheinen). Damit Sie in Zukunft keine Angst (mehr) vor Computern haben (oder sich auch nur ein eventuell vorhandenes gewisses Unbehagen verringert), soll es in diesem Kapitel darum gehen, wie Computer grob funktionieren. Klar dürfte sein, dass sie irgendwie digital mit Hilfe von Strom Bits und Bytes manipulieren. Was dort aber genau passiert, aus welchen Teilen sich ein Rechner zusammensetzt und wie diese Teile zusammenspielen, dies wollen wir uns in diesem Kapitel genauer anschauen.



2.1 Arten von Computern

2.1.1 Was ist ein Computer?

Was meinen wir, wenn wir von »Computern« sprechen?

2-4

- Übersetzt bedeutet Computer einfach »Rechner«.
- Mit Computern bezeichnete man früher Frauen, die in Großraumbüros rechneten.
- Die heute als Computer bezeichneten Maschinen werden kaum zum Rechnen benutzt.

Deshalb: Wir wollen alle »programmierbaren« Geräte als Computer bezeichnen.

✎ Zur Diskussion

Wo befinden sich überall im Hörsaal aktuell Computer?

2.1.2 Klassifikation von Computern

Wie klassifiziert man Computer?

2-5

Man klassifiziert Computer grob nach ihrer Leistungsfähigkeit:

Leistung ↑

- Supercomputer
- Mainframe
- Workstation
- Personal Computer
- Smartphone
- Computer in Waschmaschine

Was genau »Leistungsfähigkeit« bedeutet, kommt gleich.

Computer in einer Waschmaschine

2-6

Aus der Beschreibung einer Waschmaschine

- »Menügeführte Lavalogic Programmsteuerung mit großem LCD Display«
- »Beleuchtestes Display, Klartext in Ländersprache wählbar«
- »Memory-Funktion zum Abspeichern von 4 individuell zusammengestellten Programmen«
- »Fuzzy-geregelte Mengenautomatik«
- »Extrem große Einfüllöffnung (30 cm) mit metallisiertem Bullaugenring und AEG-Logo«
- »Software-Update für zukünftige Fortschritte in der Waschtechnik«

Unkown author, public domain

Eine Workstation

2-7

Typische Daten

- Prozessor: $8 \times 2{,}5$ GHz
- Speicher: 8GB

Copyright Justin Morgen, GNU Free Documentation License

Ein Supercomputer

2-8

Typische Daten

Googlen wir mal…

Copyright Silicon Graphics, GNU Free Documentation License

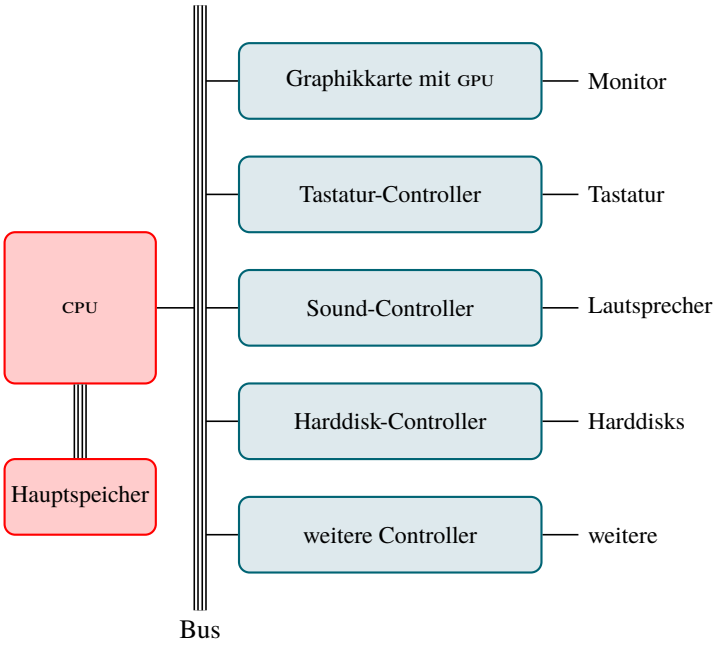
2.2 Aufbau von Computern

Regie

Ein Computer wird aufgeschraubt und wir schauen sein Innenleben an.

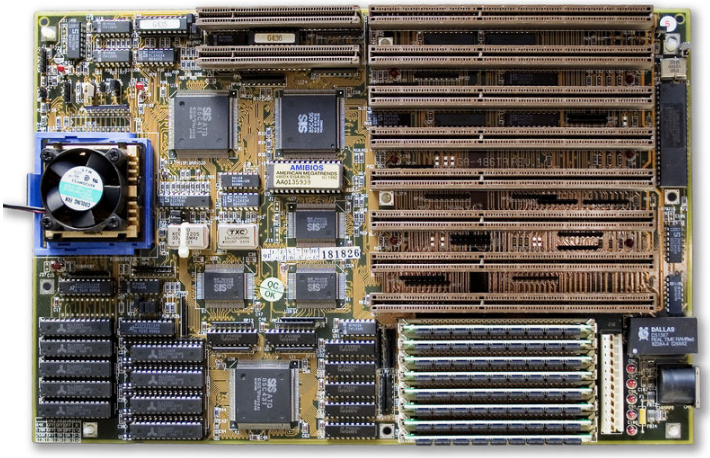
2-9

Wie sind Computer prinzipiell aufgebaut?



2-10

Das Innere eines Computers



Copyright Andrew Dunn, Creative Commons Attribution Sharealike License

2.2.1 Die CPU

2-11

Was ist die CPU?

Die Abkürzung steht für Central Processing Unit. Sie arbeitet *getaktet* und führt in jedem Takt eine *Instruktion* aus. Typische Instruktionen können sein

- Addiere die Zahlen in Speicherzellen 5 und 17 und schreibe das Ergebnis in Speicherzelle 2345872.
- Nimm ein Zeichen von der Tastatur entgegen und schreibe es in Speicherzelle 5.

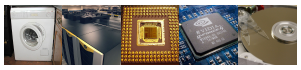
Ist eine Instruktion abgearbeitet, so führt die CPU die nächste Instruktion im Speicher aus (außer bei Sprungbefehlen).

2-12

Was ist Software?

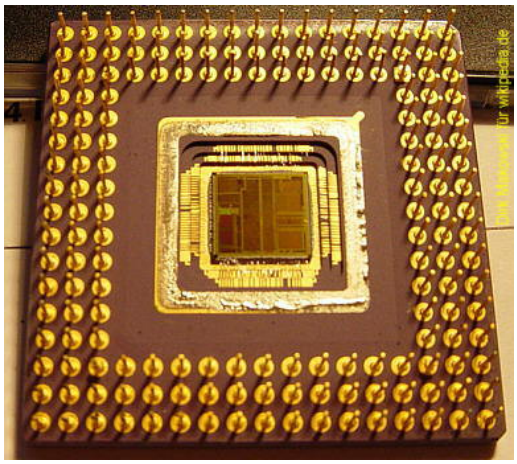
Merke

Die Instruktionsfolgen (Programme), die von CPUs ausgeführt werden, nennt man *Software*.



Beispiel einer CPU

2-13



Public Domain, unknown author

Beispiel einer Instruktionsfolge (Programm).

2-14

1. Hole eine Zahl von der Tastatur und schreibe sie in Speicherzelle 1.
2. Hole eine Zahl von der Tastatur und schreibe sie in Speicherzelle 2.
3. Falls der Inhalt von Speicherzelle 1 kleiner ist als der Inhalt von Speicherzelle 2, weiter bei 5.
4. Vertausche den Inhalt von Speicherzellen 1 und 2.
5. Schicke den Inhalt von Speicherzelle 1 an den Monitor.
6. Schicke den Inhalt von Speicherzelle 2 an den Monitor.

✎ Zur Übung

Finden Sie heraus, was das Programm »macht«.

Wie schnell sind CPUs?

2-15

Gemessen wird die Anzahl an Instruktionen, die pro Sekunde ausgeführt werden (mips). Frühre brauchte die CPU oft mehrere Takte für eine Instruktion, heute hingegen schafft sie mehrere Instruktionen pro Takt. Daher ist Taktfrequenz nicht gleich Leistung.

Jahr	Prozessor	MHz	mips
1975	6502 (C64)	1	0,365
1985	80386	16	4
2005	Pentium 4	2800	5.340
2011	Core i7	3400	128.300

2.2.2 Der Hauptspeicher

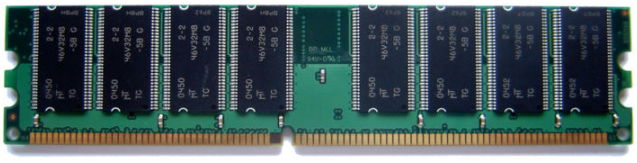
Was ist der Hauptspeicher?

2-16

Der Hauptspeicher enthält die Daten, auf die die CPU direkt zugreifen kann. Er enthält auch die Instruktionsfolge, die die CPU abarbeitet. Genau wie die CPU ist er ein Chip. Der Hauptspeicher muss ständig mit Strom versorgt werden, selbst wenn sich gar nichts ändert.

Beispiel eines Hauptspeicher-Riegels

2-17



Unknown author, Creative Commons Attribution Sharealike License

2-18

Wie groß ist der Hauptspeicher?

Jahr	Typischer Hauptspeicher
1975	16 kB
1985	128 kB
1995	32 MB
2005	1 GB
2015	8 GB

Zum Vergleich: Ein Desktop-Icon hat bis zu 50kB.

2.2.3 Die Festplatte

2-19

Was ist die Festplatte?

Die *klassische Festplatte:* Festplatten bestehen aus magnetischen Scheiben, auf denen Bits durch Magnetisierung gespeichert werden. Im Gegensatz zum Hauptspeicher behält sie ihren Inhalt auch ohne Strom, kann dafür aber viel mehr speichern als der Hauptspeicher (etwa Faktor 100 bis 1000). Die CPU kann nicht direkt auf die Daten der Festplatte zugreifen.
Die neueren *Solid-State-Disks:* Sie verhalten sich wie Festplatten, sind aber physikalisch »Speicherriegel, die auch ohne Strom ihren Inhalt behalten«. Sie sind wesentlich teurer und schnell als Festplatten.

2-20

Beispiel einer klassischen Festplatte

Unknown author, Creative Commons Attribution Sharealike License

2-21

Wie groß sind Festplatten?

Jahr	Typische Festplattengröße
1975	–
1985	1 MB
1995	200 MB
2005	250 GB
2015	2 TB

Zum Vergleich: Ein DVD hat ca. 5 bis 9 GB.

2.2.4 Der Graphikprozessor

2-22

Was ist der Graphikprozessor (GPU)?

Die Abkürzung steht für Graphics Processing Unit. Sie arbeitet genau wie eine CPU, hat aber zusätzlich spezielle Instruktionen für Graphik wie

- Male eine Linie von (x_1, y_1) nach (x_2, y_2).
- Rotiere diesen Pfeil um 30 Grad.
- Fülle dieses Dreieck mit einer Textur.

Wirklich benötigt werden die GPUs lediglich für 3D-Graphiken, insbesondere in Computerspielen.

Beispiel eines Graphikprozessors

2-23

Unknown author, Creative Commons Attribution Sharealike License

Wie leistungsfähig sind GPUs?

2-24

Gemessen wird die Geschwindigkeit von GPUs in »mflops« (statt »mips«), was für »million floating point operations per second« steht.

Jahr	mflops
1975	–
1985	–
1995	3
2005	1.000
2013	4.640.000

Die Rechenleistung von GPUs übertrifft die Leistung der CPUs.
Man missbraucht deshalb manchmal GPUs für wissenschaftliche Berechnungen.

2.3 Aufbau von Software

Probleme bei der Programmierung der CPU.

2-25

Programmieren auf der Ebene »Gib das Zeichen in Zeile 5, Spalte 7 aus.« ist mühselig, oft wissen wir bei der Programmierung gar nicht, wo das Zeichen nachher ausgegeben wird. Wichtiger noch: Es ist uns auch egal, wo genau es ausgegeben werden wird. Weiterhin ist »Speicherzelle 0« keine gute Idee, da andere Programme diese Speicherzelle vielleicht auch nutzen; ganz ähnlich liegen die Probleme, wenn zwei Programme gleichzeitig drucken wollen. Jeder Computer ist ein bisschen anders: ändert sich die Hardware, müssten die Programme umgeschrieben werden.
Die Lösung: Schichtung der Software.

2.3.1 Schicht 1: BIOS

Erste Schicht: Das BIOS

2-26

BIOS steht für *B*asic *I*nput *O*utput *S*ystem. Das BIOS stellt einen kleinen Satz von Programmen zur Verfügung, die sich um Dinge kümmern wie

- Ein Zeichen auf dem Bildschirm an der aktuellen Cursorposition ausgeben.
- Ein Zeichen von der Tastatur lesen.
- Einen Datenblock von der Festplatte in den Speicher lesen.

Das BIOS ist im Speicher vorhanden, sobald der Computer Strom hat.

2.3.2 Schicht 2: Betriebssystem

2-27

Zweite Schicht: Das Betriebssystem

Das *Betriebssystem* ist ein großes Programm, das folgende Dinge leistet:

- Speicherverwaltung (damit »Speicherzelle 0« von Word nicht dasselbe ist wie »Speicherzelle 0« von Mail).
- Dateiverwaltung (damit Programme auf »`/home/tantau/Vorlesung.pdf`« zugreifen können und nicht auf Platte 2, Zylinder 4, Spur 5, Sektor 36, Bits 341325 bis 341991).
- Prozessverwaltung (damit alle Programme gerecht Rechenzeit bekommen).

Beispiel

Windows, MacOS, Linux, Solaris, AIX.

2.3.3 Schicht 3: Graphische Oberfläche

2-28

Dritte Schicht: Graphische Oberfläche

Die graphische Oberfläche wird auch GUI genannt (Graphical User Interface). Auch sie ist eine Software, die sich darum kümmert, die Fenster zu verwalten und Graphiken anzuzeigen. Das GUI wird heutzutage oft mit dem Betriebssystem verschmolzen, besonders bei Windows und MacOS.

Beispiel

Windows, MacOS, X Windows

2.3.4 Schicht 4: Anwendungen

2-29

Vierte Schicht: Anwendungen

Anwendungsprogramme »thronen über allem«. Wie der Name schon sagt, werden sie direkt von Anwendern benutzt. Idealerweise sollten sie fehlertolerant (Motto: Benutzer machen keine Fehler!), schnell, anpassbar, intuitiv, leicht erlernbar, konsistent, …und noch viele weitere schöne Eigenschaften haben.

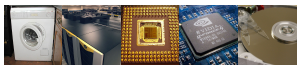
Beispiel

Word, Mail, Spiele, Firefox

2-30

Was passiert bei einem Mausklick?

Systemtakt	Was macht der Computer?
0	Mausknopf wird gedrückt, Anwendung malt gerade Pixel 35 eines Fensters
1.000	Mauscontroller bittet um Unterbrechung, Sicherung der aktuellen Instruktionsnummer
1.050	BIOS meldet dem Betriebssystem »Mausknopf-Runter«
1.150	Betriebssystem reiht »Mausknopf-Runter« in die Systemwarteschlange ein
1.250	Rückkehr zur Anwendung und Malen von Pixeln
1.000.000	Wechsel zum Betriebssystem-Programm
1.000.500	Schaue nächstes Ereignis in der Warteschlange an
1.001.000	Graphische Oberfläche berechnet, auf welches Fenster geklickt wurde
1.003.000	Graphische Oberfläche reiht »Mausknopf-Runter« in die Warteschlange der Anwendung des Fensters ein
1.004.000	Mail-Anwendung wird kurz ausgeführt
2.000.000	Wechsel zur Anwendung, auf die geklickt wurde
2.050.000	Anwendung schaut sich ihre Warteschlange an
2.051.000	Anwendung beginnt, Instruktionen auszuführen, die sich um den Mausklick kümmern.



Zusammenfassung dieses Kapitels

▶ **Hardware**
Computer sind programmierbare Geräte, die aus CPU, GPU, Hauptspeicher, Festplatte und Peripherie bestehen.

▶ **Software**
Programme sind Instruktionsfolgen. Durch Schichtung in BIOS, Betriebssystem, GUI und Anwendungen wird das Programmieren überhaupt erst möglich.

▶ **Moore's Law**
Die Leistungsfähigkeit aller Komponenten verdoppelt sich etwa alle zwei Jahre.

Übungen zu diesem Kapitel

Übung 2.1 Vertrautheit mit Rechnern erlangen, leicht

Ihr Tutor teilt Sie in Gruppen ein; jede Gruppe bekommt einen Rechner zugeteilt. Protokollieren Sie Ihre Beobachtungen zu den folgenden »Experimenten« kurz schriftlich.

Keine Angst, die Geräte sind nicht unsere neuesten. Schrauben Sie also ruhig drauf los!

1. Bauen Sie den Rechner auf und schließen Sie alle notwendigen Kabel an. Starten Sie den Rechner. Welches Betriebssystem ist installiert?
2. Schalten Sie nun den Rechner wieder aus und entfernen Sie alle Kabel, insbesondere die Stromversorgung! Öffnen Sie das Gehäuse. Finden Sie heraus, wo folgende Teile zu finden sind:
   - Festplatte
   - Hauptspeicher (RAM)
   - Prozessor (CPU)
   - Grafikkarte (GPU)

   Sind Erweiterungskarten eingebaut? Wenn ja, wozu dienen sie?
3. Man kann den Rechner auch ohne Gehäuse anschließen und starten. Nennen Sie zwei sinnvolle Gründe, weshalb man dies nicht tun sollte.
4. Entfernen Sie nun die Festplatte. Schließen Sie den Rechner wieder an und starten Sie ihn. Was passiert?
5. Bauen Sie Ihre Festplatte in den Rechner einer anderen Gruppe ein. Lässt sich dieser Rechner nun starten? Wenn ja, was ändert sich?

Übung 2.2 Begriffsnetz, leicht

Bei dieser Übung geht es darum, sich die Begriffe aus der Vorlesung besser zu merken.

1. Ihr Tutor teilt Ihnen einen Zettel zu, auf dem einer der unten angegebenen Begriff aus der Vorlesung aufgeschrieben ist.
2. Überlegen Sie sich ein bis zwei Sätze, mit denen Sie den Begriff verständlich erklären können. Schreiben Sie die Sätze auf die Rückseite des Zettels.
3. Der Tutor beginnt nun mit dem zentralen Begriff Computer. Er erklärt diesen und heftet seinen Begriff an die Tafel.
4. Falls Ihr Begriff gut zum letzten genannten Begriff passt, so melden Sie sich, gehen Sie nach vorne, erklären Sie ihren Begriff und heften Sie ihn an eine geeignete Stelle an die Tafel.

Die Begriffe: Hardware • CPU • GPU • Festplatte • Speicher • Speicherzelle • PC • Workstation • Mainframe • Software • Programm • BIOS • Betriebssystem • GUI • Anwendungsprogramm • mips • Takt • Instruktion • Programm

Übung 2.3 Vertrautheit mit Systemanforderungen erlangen, schwer

Bei dieser Übung sollen Sie für ein bestimmtes Szenario versuchen abzuschätzen, welche Hard- und Software benötigt wird. Der Ablauf:

1. Ihr Tutor teilt Sie in Gruppen ein. Jede Gruppe bekommt eines der unten angegebenen Szenarien zugeordnet.
2. Benennen Sie eine Person in der Gruppe, die auf die Zeit achtet.
3. Versuchen Sie als Gruppen innerhalb von etwa 40 Minuten folgende Fragen zu klären:
   - Welche aus der Vorlesung bekannten Hardwarekomponenten werden benötigt?
   - Wie leistungsfähig sollten diese sein?
   - Welche Softwareschichten werden gebraucht?

- Was wird das alles grob kosten?

4. Bereiten Sie innerhalb von etwa 20 Minuten eine Präsentation Ihres Szenarios und Ihrer Ergebnisse vor.
5. Stellen Sie Ihre Ergebnisse in einer 5-minütigen Präsentation vor.

Szenario 1: Fingerabdruckdatenbank

Unknown author, Public Domain

Es soll eine zentrale Datenbank mit Fingerabdrücken aufgebaut werden. Von einzelnen Rechnern aus soll man darauf zugreifen können, um Fingerabdrücke zu vergleichen. Was sollten die an diesem System beteiligten Computer leisten?

Szenario 2: EC-Automat

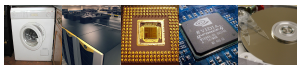
Unknown author, GNU Free Documentation License

Betrachten Sie einen gewöhnlichen EC-Kartenautomaten. Über welche Fähigkeiten verfügt ein solcher Automat?

Szenario 3: CampusCard

Copyright: Studentenwerk Schleswig Holstein

Das neue CampusCard-System einschließlich Terminals zum Laden und Abbuchen. Was müssen die beteiligten Karten/Computer können?



Szenario 4: TollCollect

Die circa 300 auf deutschen Autobahnen aufgestellten Mautbrücken erfassen mautpflichtige Fahrzeuge im Vorbeifahren und kontrollieren, ob die Maut bezahlt wurde. Welche Computer werden zum Betrieb einer Mautbrücke benötigt und was müssen sie leisten?

Copyright Stefan Kühn, GNU Free Documentation License

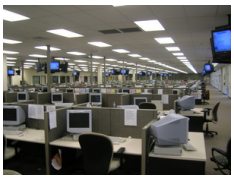
Szenario 5: Computer-Tomograph

Eine CT-Station in einem Krankenhaus – für welche Aufgaben werden Computer benötigt? Was müssen sie leisten?

Unknown Author, Public Domain

Szenario 6: Call-Center

Ein Call-Center für den Kunden-Support Ihres neuen Telefonanbieters. Wofür werden im Call-Center Computer benötigt und was müssen sie können?



Unknown author, Public Domain

3-1

Kapitel 3
Betriebssysteme
Von Äpfeln und Pinguinen

3-2

Lernziele dieses Kapitels

1. Aufgaben von Betriebssystemen kennen
2. Konzepte der Datei-, Benutzer- und Prozessverwaltung kennen
3. Mit Dateien, Pfaden und Dateirechten umgehen können
4. Aufgaben von Treibern erläutern können

Inhalte dieses Kapitels

3.1	Einführung	21
3.1.1	Was ist ein Betriebssystem?	21
3.1.2	Welche Betriebssysteme gibt es?	21
3.2	Ressourcen	22
3.2.1	Druckauftragsverwaltung	22
3.2.2	Dateiverwaltung	23
3.2.3	Nutzerverwaltung	24
3.2.4	Prozessverwaltung	25
3.3	Aufbau von Betriebssystemen	25
3.3.1	Schon wieder Schichten	25
3.3.2	Untere Schicht: Treiber	25
3.3.3	Obere Schicht: Shells, Systemaufrufe	26
	Übungen zu diesem Kapitel	27

Worum es heute geht

Betriebssysteme sind wie Behörden: Eigentlich machen sie selbst nichts wirklich Produktives, aber ohne sie läuft nichts. Wir alle haben schon – zu Recht! – über dieses oder jenes Amt geschimpft, sind aber im Großen und Ganzen mit einer recht effizienten Verwaltung gesegnet (wie es sich in einer komplett verwaltungsfreien Welt lebt, kann man in dem Buch *Snow Crash* schön nachlesen).

Es gibt für »normale« Computer derzeit drei wichtige Betriebssysteme: Windows, von der Firma Microsoft entwickelt; MacOS, von der Firma Apple entwickelt; und Linux, welches ursprünglich von Linus Torvald programmiert wurde und heute von einer so genannten *Community* weiterentwickelt wird. Im Gegensatz zu Windows und MacOS ist Linux komplett frei verfügbar – wer Spaß daran hat, kann Linux völlig legal beliebig abändern und erweitern. Erfahrungsgemäß haben Informatiker daran Spaß, anderen Menschen nicht.

In diesem Kapitel soll es nicht um die Details einzelner Betriebssysteme gehen, diese ändern sich sowieso alle paar Jahre. Wichtiger ist es, die grundlegenden Idee zu verstehen, welche auch in allen Betriebssystemen letztendlich sehr ähnlich umgesetzt sind:

1. Betriebssysteme *verwalten* »alles, worum sich Programme streiten könnten«. Beispielsweise streiten sich Programme sehr gerne um den Zugriff auf Drucker, Soundkarten, Graphikkarten oder Festplatten, weshalb hier das Betriebssystem schlichten muss.
2. Betriebssysteme sind in zwei Schichten getrennt, einen hardwareunabhängigen Teil und hardwareabhängige *Treiber*.
3. Daten werden von allen gängigen Betriebssystemen in Form von *Verzeichnisbäumen* organisiert.

3.1 Einführung

3.1.1 Was ist ein Betriebssystem?

Wiederholung: Das Betriebssystem

Das Betriebssystem ist ein großes Programm, das unter anderem folgende Dinge leistet:

- Druckauftragsverwaltung
- Dateiverwaltung
- Nutzerverwaltung
- Prozessverwaltung

✎ Zur Diskussion

Um welche Ressourcen könnte sich ein Betriebssystem noch kümmern?

3.1.2 Welche Betriebssysteme gibt es?

Unterscheidungsmerkmale von Betriebssystemen

Man kann Betriebssysteme unterscheiden nach:

- Funktionsumfang
  - Vom Betriebssystem eines Fahrradcomputers …
  - … über Windows und Linux …
  - … bis zum Betriebssystem einer Mainframe.
- Abstammung
  - Unix-Familie: Linux, MacOS X, Solaris, AIX, …
  - MS-DOS-Familie: MS-DOS, DR-DOS, Windows
  - Eigenentwicklungen: PalmOS, BeOS, OS/2, …

Zur Historie von Windows

1981 IBMs erste »Personal Computer« werden mit dem »Microsoft Disk Operating System« (*MS-DOS*) ausgeliefert, da ein eigenes nicht zur Verfügung steht.

1986 Microsofts *Windows* wird vorgestellt. Es soll den Umstieg von MS-DOS auf OS/2 erleichtern.

1987 IBMs eigenes Betriebssystem *OS/2* wird vorgestellt.

1991 Microsoft treibt Windows voran, welches sich gegen OS/2 durchsetzt.

Zur Historie von Linux

ab 1960 Verschiedene Firmen stellen Varianten des Betriebssystems *Unix* her.

1983 Das *GNU-Projekt* (»Gnu is Not Unix«) macht es sich zur Aufgabe, ein *freies* Unix zu schaffen.

1991 *Linus Torvald* startet ein Projekt, das GNU-Unix auf PCs mit Intel-Prozessoren zu portieren.

heute *Linux* ist des größte und verbreitetste freie Betriebssystem.

3.2 Ressourcen

3.2.1 Druckauftragsverwaltung

3-8

Ressource I: Die Verwaltung von Druckaufträgen ist scheinbar einfach.

Problemstellung

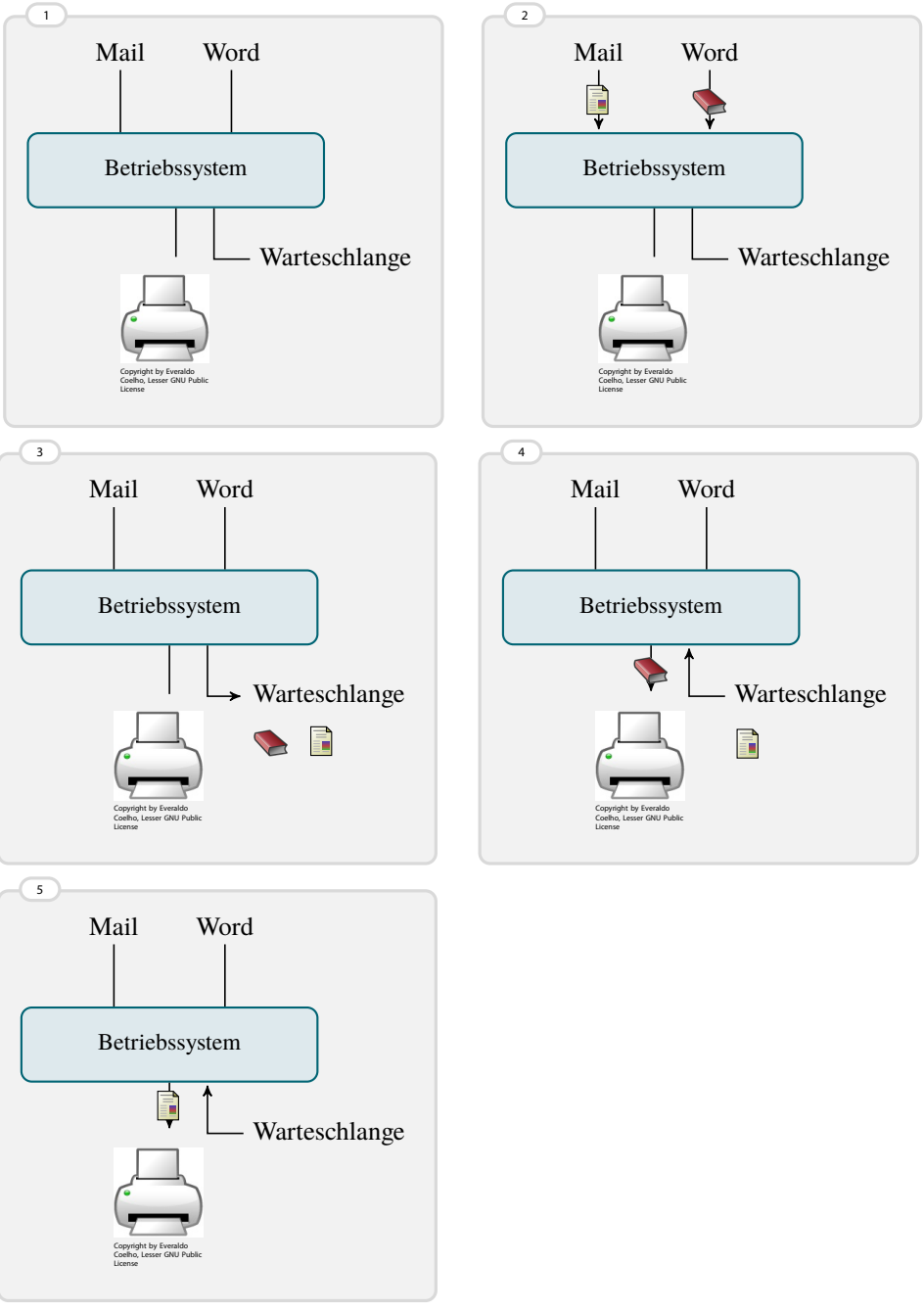
Anwendungsprogramme sollen Texte und Graphiken drucken können. Jedoch können mehrere Anwendungsprogramme gleichzeitig versuchen, etwas zu drucken.

Lösung

Anstatt Daten direkt an die Drucker zu schicken, erstellen Anwendungsprogramme *Druckaufträge*, die sie an das Betriebssystem schicken. Dieses reiht sie in eine *Warteschlange* ein und schickt sie dann in der Reihenfolge ihres Eintreffens an den Drucker.

3-9

Beispiel, wie die Druckauftragsverwaltung arbeitet.



3.2.2 Dateiverwaltung

Ressource II: Daten. 3-10

Problemstellung

Die *Daten* der Benutzer müssen verwaltet werden. Daten schließt *Texte*, *Graphiken*, *Filme*, *Programme* und viele Dinge mehr ein, die *sehr unterschiedliche Größe* haben. Weiterhin soll der Zugriff *schnell* sein.

Lösung

Alle Daten werden als *Folgen von Bytes* gespeichert. Jede Einheit von Daten wie ein Dokument oder ein Programm oder ein Film bildet eine *Datei* (File) und bekommt einen *Namen*. Zur größeren Übersichtlichkeit werden diese Dateien in einem *Verzeichnisbaum* angeordnet.

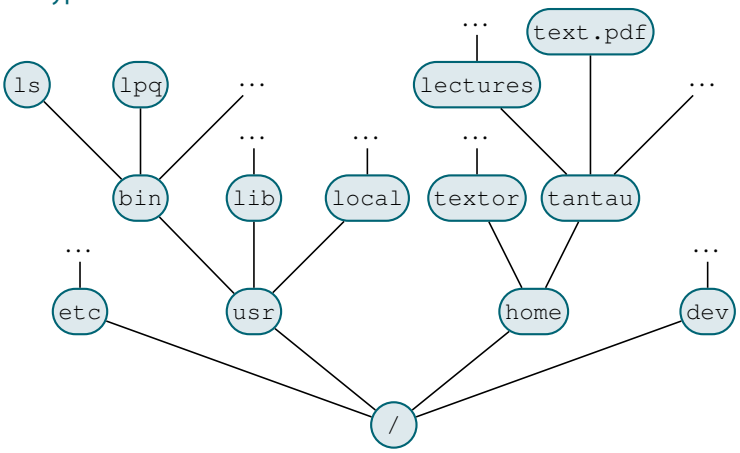
Was man über Dateien wissen sollte. 3-11

Dateien sind Folgen von Bytes. Ihre *Länge* oder *Größe* wird in Byte gemessen, wobei die kürzesten Dateien Null Bytes haben, die längsten mehrere Gigabyte. Ihr *Name* ist eine vom Inhalt unabhängige Zeichenfolge, die am besten keine Sonderzeichen enthält. Der Name besteht oft aus zwei durch einen Punkt getrennte Teile, wobei der zweite Teil (*Suffix* oder *Dateierweiterung* genannt) angibt, um welche Art Datei es sich handelt.

Was man über Verzeichnisse wissen sollte. 3-12

Verzeichnisse enthalten Dateien und wieder Verzeichnisse (Unterverzeichnisse). Genau wie Dateien haben Verzeichnisse einen Namen. Das oberste Verzeichnis nennt man das *Wurzelverzeichnis* (root directory).

Ein typischer Unixverzeichnisbaum. 3-13



Das Konzept des absoluten Pfads. 3-14

Die Folge von (Unter-)Verzeichnisnamen, die zu einer Datei führt, bezeichnet man als *absoluten Pfad*. Innerhalb des Pfades werden die verschiedenen Verzeichnisnamen durch Schrägstriche getrennt (Unix) oder durch umgedrehte Schrägstriche (Windows).
Beispiel: `/home/tantau/text.pdf`
Beispiel: `\windows\system32\windows.exe`
Kommt innerhalb eines Pfades »`..`« vor, so ist damit das Verzeichnis eins höher gemeint.

✎ Zur Übung
Welche Datei bezeichnet der Pfad `/usr/../home/tantau/../../dev/null`?

Das Konzept des relativen Pfads. 3-15

Ein *relativer Pfad* setzt einen Pfad relativ zu einem *aktuellen Verzeichnis* fort. Auf das aktuelle Verzeichnis kann man sich mittels »`.`« beziehen.
Beispiel: Wir befinden uns im Verzeichnis `/home`. Dann ist `./tantau/text.pdf` die Datei `/home/tantau/text.pdf`.
Beispiel: Wir befinden uns im Verzeichnis `/home/textor`. Dann ist ebenfalls die Datei `../tantau/text.pdf` dieselbe Datei wie oben.

✎ Zur Übung
Sie befinden sich im Verzeichnis `/usr/local/bin`.
Welche Datei bezeichnet dann `../../local/../bin/lpq`?

3.2.3 Nutzerverwaltung

3-16

Ressource III: Nutzer und Rechte.

Problemstellung

Die Daten von vielen Personen (Benutzern) sollen auf einem Computer gespeichert werden. Dabei soll es Benutzern gestattet sein, manche Daten anderer Benutzer zu lesen, andere aber wieder nicht.

Lösung

Jeder Benutzer bekommt ein Verzeichnis, in dem seine Daten liegen. Für jede Datei gibt es *Rechte*, die festlegen, wer was darf. Für jeden Benutzer speichert das Betriebssystem ein *Passwort* und Nutzer müssen sich *einloggen* und damit *authentifizieren*.

3-17

Dateibesitz in Unix.

Das Unix-Betriebssystem regelt zunächst den *Besitz* von Dateien:

1. Jede Datei gehört genau einem bestimmten *Nutzer*. Besitzer ist, wer die Datei angelegt hat. Nur der Besitzer kann die Rechte ändern.
2. Jede Datei gehört genau einer bestimmten *Gruppe*. Eine Gruppe ist eine Menge von Benutzern, wobei Benutzer aber Mitglieder mehrerer Gruppen sein können.
   Genau wie Dateien werden Gruppen auch vom Betriebssystem verwaltet.

3-18

Zugriffsrechte in Unix.

Das Unix-Betriebssystem kennt drei Arten von Rechten:

1. *Leserecht (r-Recht)*
   Das Recht, den Inhalt einer Datei / eines Verzeichnis zu lesen.
2. *Schreibrecht (w-Recht)*
   Das Recht, den Inhalt einer Datei / eines Verzeichnis zu ändern.
3. *Ausführungsrecht (x-Recht)*
   Das Recht, ein Programm auszuführen, dessen Code in der Datei liegt. Fehlt dieses Recht für ein Verzeichnis, so kann man es nicht in einem Dateipfad benutzen (»man kann nicht hineinwechseln«).

3-19

Die Rechtetabelle einer Datei.

Für jede Datei wird gespeichert, welche der drei Rechte bestimmte Personengruppen haben:

1. Es wird gespeichert, welche Rechte der Besitzer hat.
2. Es wird gespeichert, welche Rechte Angehörige der Gruppe haben.
3. Es wird gespeichert, welche Rechte alle anderen Benutzer haben.

Dies führt zu einer Rechtetabelle wie im folgenden Beispiel:

	read	write	execute
user	yes	yes	yes
group	yes	no	yes
other	no	no	no

Dies schreibt man auch kurz so: `rwxr-x---`.

3-20

✎ Zur Übung

Die Nutzer `adam` und `eva` sind beide in der Gruppe `eden`, aber nur `adam` ist in der Gruppe `men`. In einem Verzeichnis finden sich folgende Dateien:

Dateiname	Besitzer	Gruppe	Rechte
`apple.txt`	`eva`	`eden`	`rwxr-----`
`snake.txt`	`adam`	`eden`	`rw-rw-r--`
`eat`	`adam`	`men`	`rwxrwxr-x`

Welche Dateien können jeweils Adam und Eva lesen und welche schreiben?

3.2.4 Prozessverwaltung

Ressource IV: Prozesse. 3-21

Problemstellung
Benutzer wollen mehrere Programme gleichzeitig ausführen. Falls Prozesse dabei böse Dinge versuchen, muss man sie gewaltsam stoppen können.

Lösung
Das Betriebssystem verwaltet eine *Prozessliste*. Jeder Prozess hat eine *Nummer* und *gehört* einem Benutzer. (Nur) der Benutzer kann seine Prozesse *töten*.

✎ **Zur Diskussion**
Was könnten »böse Dinge« alles sein?

3.3 Aufbau von Betriebssystemen

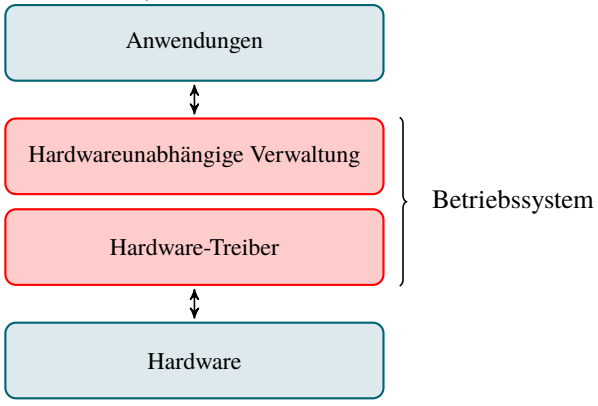
3.3.1 Schon wieder Schichten

Schon wieder die Druckauftragsverwaltung. 3-22
Beim Drucken gibt es *zwei unterschiedliche Probleme*:

1. Die Druckaufträge müssen verwaltet werden. Dies beinhaltet Tätigkeiten wie: Nummerieren der Druckaufträge, Verschicken von Rückmeldungen, Auflistung der angeschlossenen Drucker.
2. Das Übermitteln der Daten an den Drucker. Dies beinhaltet Tätigkeiten wie: Umwandlung der Daten in ein Format, das der Drucker versteht (jeder Drucker spricht eine eigene »Sprache«), oder Senden der Daten an das richtige Kabel und den richtigen Anschluss.

Wichtige Beobachtung
Die erste Aufgabe ist *hardwareunabhängig*. Die zweite ist *druckerabhängig* und muss *für jeden Drucker anders programmiert werden*.

Ein Betriebssystem besteht aus zwei Schichten. 3-23



3.3.2 Untere Schicht: Treiber

Was ist ein Hardware-Treiber? 3-24
Ein *Treiber* ist ein austauschbarer Teil des Betriebssystems. Er ist ein Programm, das für *eine bestimmte Hardware* eine *bestimmte Aufgabe* löst. Baut man eine neue Hardware in einen Computer ein, so muss man in der Regel einen passenden Treiber installieren.

Beispiel
Ein *Druckertreiber* kümmert sich darum, zu druckende Daten in ein Format zu konvertieren, das ein bestimmter Drucker versteht, oder den richtigen Anschluss (beispielsweise USB) anzusprechen.
Er kümmert sich *nicht* darum, Druckaufträge zu verwalten.

3.3.3 Obere Schicht: Shells, Systemaufrufe

3-25

Wie sagt man dem Betriebssystem, was man von ihm will?

Benutzer können auf zwei Arten mit dem Betriebssystem kommunizieren:

1. Mittels einer *Shell*.
   Dazu mehr im nächsten Kapitel.
2. Mittels eines *graphischen Tools*.
   Diese findet man in graphischen Oberflächen in der Regel in einer Art »Systemmenü«.

Programme kommunizieren mit dem Betriebssystem auf zwei Arten:

1. Ebenfalls mittels einer *Shell*.
2. Mittels so genannter *Systemaufrufe*.
   Hier gibt es für jeden *Verwaltungsakt* des Betriebssystems ein kleines Programm, das man aufrufen kann.

Zusammenfassung dieses Kapitels

3-26

Betriebssysteme

Ein *Betriebssystem* verwaltet verschiedene Ressourcen, insbesondere den Hauptspeicher, die Festplatten, die Benutzerliste, die Prozesse (laufende Programme), die Bildschirme (Graphik) und die Peripherie (Drucker, Maus, Tastatur, Soundkarten, etc.).
Für jede Ressource enthält das Betriebssystem einen allgemeinen Teil, der unabhängig von der Hardware ist, und einen austauschbaren, hardwareabhängigen Teil, genannt *Treiber*.

Dateien

Dateien haben einen *Name*, eine bestimmte *Länge,* die in Bytes gemessen wird, und einen *Namenssuffix,* der die »Art« der Datei angibt.

Dateibaum

Dateien werden in einem *Baum* angeordnet. Ein *absoluter Pfad* in diesem Baum startet bei der »Wurzel /«. Ein *relativer Pfad* startet im »aktuellen Verzeichnis«. In einem Pfad steht `..` für »einen Schritt zurück«.

Rechte

Es gibt drei Rechte:

- Das Leserecht »`r`«.
- Das Schreibrecht »`w`«.
- Das Ausführungsrecht »`x`«.

Es gibt drei Arten von Besitz:

- Man ist Besitzer der Datei (»`u`« wie *user*).
- Man gehört zur Gruppe der Datei (»`g`« wie *group*).
- Man ist jemand anderes (»`o`« wie *other*).

Man nutzt Folgen von neun Zeichen wie `rwxrw-rw-`, Rechte an einer Datei oder einem Verzeichnis zu notieren.

Übungen zu diesem Kapitel

Übung 3.1 Den Verzeichnisbaum erkunden, leicht

Setzen Sie sich an einen Rechner im Rechnerpool oder an einen Linux-Rechner zu Hause. Probieren Sie dann folgende Dinge aus:

1. Versuchen Sie, eine CD-ROM in Ihr Laufwerk einzulegen und diese dann im Verzeichnisbaum wiederzufinden. Klappt das auch mit einem USB-Stick?
2. Klicken Sie im Startmenü auf »Befehl ausführen« (Sie können auch die Tasten ALT und F2 drücken) und geben Sie `acroread` ein. Klicken Sie auf »Ausführen«. Was passiert? Können Sie das Programm `acroread` im Verzeichnisbaum finden? Versuchen Sie dasselbe mit dem Programm `kate`!
3. Schauen Sie sich das Verzeichnis `/usr/local` an. Können Sie dort ein ausführbares Programm finden?
4. Führen Sie den Befehl `konsole` aus, um eine Shell zu öffnen. Geben Sie dort (nacheinander) die Befehle `cat /proc/cpuinfo` und `cat /proc/meminfo` ein, um sich den Inhalt der Dateien `/proc/cpuinfo` und `/proc/meminfo` anzeigen zu lassen. Was steht in diesen Dateien?

Übung 3.2 Zugriffsrechte erkunden, mittel

Setzen Sie sich an einen Rechner im Rechnerpool oder an einen Linux-Rechner zu Hause. Probieren Sie dann folgende Dinge aus:

1. Erstellen Sie in Ihrem Home-Verzeichnis eine Textdatei `test.txt` mit beliebigem Inhalt. Geben Sie dieser Datei folgende Zugriffsrechte:
   `------rw-`
2. Können Sie diese Datei nun noch öffnen? Hat Ihr Partner Zugriff darauf? Wie verhält es sich mit folgenden Rechten:
   `---rw-rw-`
3. Schauen Sie sich die Zugriffsrechte der anderen Dateien und Ordner in Ihrem Home-Verzeichnis an. Ist alles so konfiguriert, wie Sie es gerne hätten? Gibt es im Home-Verzeichnis Ihres Nachbarn Dateien, die Sie verändern oder löschen *könnten*?

Übung 3.3 Prozesse verstehen, mittel

Setzen Sie sich an einen Rechner im Rechnerpool oder an einen Linux-Rechner zu Hause. Probieren Sie dann folgende Dinge aus:

1. Finden Sie das Programm, das die auf dem System momentan laufenden Prozesse anzeigt! Versuchen Sie, den Prozess `init` zu beenden. Klappt das? Was passiert, wenn Sie den Prozess `kicker` beenden?
2. Beenden Sie nun möglichst viele Prozesse. Können Sie das System so beschädigen, dass es neu gestartet werden muss?

Übung 3.4 Warteschlagen verstehen, mittel

Finden Sie das Programm zur Anzeige der Druckerwarteschlange! Können Sie auch Druckaufträge anzeigen lassen, die bereits abgearbeitet sind? Entspricht die Anzeige der Ihres Nachbarn?

Prüfungsaufgaben zu diesem Kapitel

Übung 3.5 Betriebssysteme, leicht, original Klausuraufgabe

Geben Sie jeweils die zutreffende Antwort an!

1. Wobei handelt es sich *nicht* um ein Betriebssystem?
   - Linux
   - LaTeX
   - Windows
2. Die Verwaltung einer der folgenden drei Baumarten ist eine der Hauptaufgaben eines Betriebssystems. Welche ist es?
   - Binärbaum
   - Dateibaum
   - Suchbaum

4-1

Kapitel 4
Shells
Dialog mit der Waschmaschine

4-2

Lernziele dieses Kapitels

1. Unix- und Programm-Shells benutzen können
2. Wichtige Unix-Kommandos kennen
3. Einfache Shell-Skripte erstellen können

Inhalte dieses Kapitels

4.1 Einführung zu Shells 29

4.2 Unix-Shell-Befehle 30
4.2.1 Dateiverwaltung 31
4.2.2 Rechteverwaltung 32
4.2.3 Nützliche Befehle 33

4.3 Unix-Shell-Programmierung 34
4.3.1 Um- und Weiterleitung 34
4.3.2 Shell-Skripte 35

Übungen zu diesem Kapitel 36

Worum es heute geht

Zur Erinnerung: Betriebssysteme sind die Behörden eines Rechners. Bekanntermaßen ist es recht schwierig, mit einer Behörde zu »reden«, zum einen wegen der abstrusen Öffnungszeiten, zum anderen wegen des »Behördendeutsch«, das dort gerne gesprochen wird. Hier ein typisches Beispiel eines amtlichen Bescheides, bei welchem Sie raten dürfen, was eine österreichische Behörde nun eigentlich entschieden hat:

> Im Nachprüfungsverfahren gemäß § 162 Abs. 2 Bundesvergabegesetz 2002, BGBl I Nr. 99/2002, betreffend das Vergabeverfahren »Ausschreibung: Transport und Lagerung von werthaltigen österreichischen Mautvignetten, Abholung vom Produzenten, Einlagerung im eigenen Sicherheitslager, Kommissionierung und Transport zu den Vertriebspartnern« der Auftraggeberin ÖSAG Österreichische Autobahnen- und Schnellstraßen GmbH, Alpenstraße 94, 5033 Salzburg, wird dem Antrag der A., vertreten durch X., auf Nichtigerklärung der Entscheidung der Antragsgegnerin, ein Verhandlungsverfahren durchzuführen, stattgegeben.

Bei Betriebssystemen liegen die Dinge (leider) ähnlich, auch hier bedient man sich einer eigenen »Sprache« zur Kommunikation mit dem Betriebssystem. Die Programme, die eine solche Kommunikation herstellen, heißen *Shells*. Auch die »Amtssprache« solcher Shells ist etwas kryptisch, wie Sie sehen werden.

Shells arbeiten *dialogbasiert*. Es wird immer abwechselnd eine *(An)frage* durch Sie, den Benutzer, gestellt, woraufhin das Betriebssystem die *Antwort* anzeigt. Die Shell zeigt ein Protokoll dieses fortlaufenden Dialogs an. Solche Dialoge können lustige Komödien sein (geben Sie mal `banner MLS` ein), sie können aber auch Zeugnis einer Tragödien von Shakespeare'scher Tragweite sein (geben Sie mal `cd /; rm -rf *` ein – oder vielleicht doch lieber nicht). Wenigstens wirkt die Tragödie im letzten Fall sehr kartharsisch.

4.1 Einführung zu Shells

Wozu dienen Shells?

4-4

Shells stellen eine Dialog zwischen einem *Benutzer oder einem Programm* und einem *Programm* her. Der Ablauf:

1. Die Shell zeigt einen *Prompt* an.
   Der Prompt ist eine Eingabeaufforderung, eine Frage der Form »Meister, wie kann ich Euch dienen?«.
2. Der Benutzer gibt einen *Befehl* ein.
3. Die Shell führt diesen aus.
4. Danach zeigt sie wieder einen Prompt und die Sache beginnt von neuem.

Die Befehle, die *eine Shell versteht,* hängen von der Shell ab.

Viele Programme haben Shells

4-5

- Shells zum Unix-Betriebssystem-Kern: `sh`, `bash`, `csh`, `ksh`.
- Shells zum Windows-Betriebssystem-Kern: `command.com`.
- Shells zu anderen Rechnern: `ftp` (file transport protocol), `ssh` (secure shell).
- Shells zu Datenbanken: `mysql`.
- Shells zu Quake, einem Computerspiel.

Der Prompt einer Shell

4-6

Bash-Prompt

Shells melden sich mit einem *Prompt*, auch *Eingabeaufforderung* genannt.

Beispiel

Die Unix-Shell Bash meldet sich mit

```
tantau@pcclt02:~/uebeung06>
```

Dies heißt so viel wie »Hallo Benutzer `tantau`! Hier ist die Bash-Shell auf dem Rechner `pcclt02`. Du bist gerade im Verzeichnis `~/uebungs06` und ich wüsste jetzt gerne, wie es weitergehen soll.«

Vorführung einer Shell in einem Fenster.

Regie

Der Prompt einer Shell

4-7

FTP- und MySQL-Prompts

Beispiel

Das Programm `ftp` meldet sich mit

```
ftp>
```

Dies heißt so viel wie »Hallo Benutzer, hier ist das `ftp` Programm. Wo wir gerade sind und was jetzt passieren soll musst du schon selbst wissen. Wenn du es weißt, dann sag mal Bescheid.«

Beispiel

Das Datenbank `mysql` meldet sich mit

```
mysql>
```

Dies heißt so viel wie »Hallo Benutzer, hier ist das `mysql` Programm. Ich sage auch nicht, was gerade los ist.«

4.2 Unix-Shell-Befehle

4-8

Start und Benutzung einer Unix-Shell

Zum Starten einer Unix-Shell ruft man ein *Terminal-* oder *Konsolenprogramm* auf. Dieses stellt ein Fenster dar, in dem dann eine Standard-Shell startet. Als Antwort auf den Prompt gibt man den Namen eines Befehls ein. (Ein Befehl ist ein kleines Programm.) Dieser wird dann ausgeführt und seine Ausgabe angezeigt. Ist der Befehl (das Programm) fertig, so erscheint wieder ein Prompt.

Beispiel

```
tantau@pcclt02:~> ls
public_html  tmp  uebeung06
tantau@pcclt02:~> pwd
/home/tantau
tantau@pcclt02:~> exit
```

4-9

Optionen bei Befehlen

Dem Befehlsnamen folgen bei Unix-Shells durch Leerzeichen getrennte *Optionen* und *Parameter*. Meistens werden Optionen mit einem oder zwei einleitenden Minus-Zeichen angedeutet, Parameter werden hingegen einfach so angegeben. Welche Optionen und Parameter ein Befehl akzeptiert, muss man wissen oder nachschauen.

Beispiel

```
tantau@pcclt02:~/uebeung06> ls -l test*
-rw-r--r--  1 tantau info 547 2005-11-28 19:19 test.class
-rw-r--r--  1 tantau info 314 2005-11-28 19:19 test.ctxt
-rw-r--r--  1 tantau info 464 2005-11-28 19:19 test.java
tantau@pcclt02:~/uebeung06>
```

4-10

Wie bekommt man Hilfe?

Die Optionen `-h` oder `-help` sind oft sehr nützlich. Ebenso das Programm `man`, dem man als Parameter einen Befehlsname übergibt.

```
tantau@pcclt02:~/uebeung06> ls --help
Aufruf: /bin/ls [OPTION]... [DATEI]...
Auflistung von Informationen der DATEIen (Standardvorgabe ist das momentane
Verzeichnis). Alphabetisches Sortieren der Eintraege, falls weder -cftuSUX
noch --sort angegeben.

Erforderliche Argumente fuer lange Optionen sind auch fuer kurze erforderlich.
  -a, --all                  Eintraege, die mit . beginnen, nicht verstecken
  -A, --almost-all           implizierte . und .. nicht anzeigen
      --author               den Urheber jeder Datei ausgeben
  -b, --escape               nicht-druckbarer Zeichen oktale ausgeben
...

tantau@pcclt02:~/uebeung06> man ls
Formatiere ls(1) neu, bitte warten...
LS(1)                           User Commands                           LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
...
```

Regie

Vorführung der nachfolgenden Befehle in einem Fenster.

4.2.1 Dateiverwaltung

`cd`, `pwd`, `ls`

Der aktuelle Pfad

4-11

Während man mit einer Unix-Shell arbeitet, gibt es immer ein *aktuelles Verzeichnis*, das in der Regel im Prompt angezeigt wird. Alle *relativen Pfade* beziehen sich auf dieses Verzeichnis.

Der `pwd` (print working directory) Befehl

Der Befehl gibt das aktuelle Verzeichnis aus.

Der `cd` (change directory) Befehl

Man gibt einen neuen Verzeichnisnamen an, dieser wird dann zum neuen aktuellen Verzeichnis. Insbesondere wechselt `cd ..` eine Verzeichnisebene nach oben.

Anzeigen des Inhalts eines Verzeichnisses

4-12

Der `ls` (list) Befehl

Er zeigt eine Liste aller Dateien im aktuellen Verzeichnis an. Der Befehl kennt viele Optionen. Die wichtigste ist *-l*, die dafür sorgt, dass die Anzeige sehr detailliert ist. Als *Parameter* kann man bestimmte Dateien oder Verzeichnisse angeben. Dann werden Informationen über diese Dateien oder der Inhalt dieser Verzeichnisse angezeigt. Parameter können auch *Sternchen* enthalten kann. *Ein Sternchen steht immer für einen beliebigen Text.* So bezeichnet `*.pdf` alle Dateien, die auf »`.pdf`« enden.

`cat`, `more`, `less`

Anzeigen des Inhalts einer Datei

4-13

Ganze Dateien betrachten

Der `cat` (concatenate) Befehl

Der Befehl nimmt die Namen mehrerer Dateien als Parameter und gibt deren Inhalt aneinandergereiht aus. Nützlich ist dieser Befehl hauptsächlich in Verbindung mit Um- und Weiterleitung der Ausgabe (später).

Der `more` Befehl

Der Befehl zeigt eine Datei seitenweise an. Die verbesserte Variante dieses Befehls haben verspielte Informatiker `less` getauft.

`head`, `tail`

Anzeigen des Inhalts einer Datei

4-14

Teile von Dateien betrachten

Die `head` und `tail` Befehle

Die Befehle zeigen die ersten oder die letzten Zeilen einer Datei an. Mit der Option *-n*, wobei *n* eine Zahl ist, kann man angeben, wie viele Zeilen man sehen möchte.
Beispiel: `head -5 beispiel.txt`

`rm`, `cp`, `mv`

4-15

Bearbeiten von Dateien

Der `rm` (remove) Befehl

Der Befehl löscht die Dateien, die als Parameter übergeben werden, *unwiederbringlich.* Er löscht jedoch *keine* Verzeichnisse und er löscht *nicht* rekursiv Dateien in Unterverzeichnissen. Mit der Option `-R` kann man allerdings erzwingen, dass Dateien doch rekursiv gelöscht werden.

✎ Zur Diskussion

Welchen Effekt haben folgende Befehle?

```
cd /
rm -R *
```

4-16

Kopieren und Umbenennen

Der `cp` (copy) Befehl

Der Befehl kopiert eine Datei. Parameter sind der alte und der neue Name. Es werden keine Verzeichnisse kopiert.

Der `mv` (move) Befehl

Dieser Befehl verschiebt Dateien (erster bis vorletzter Parameter) an einen neuen Ort (letzter Parameter). Man kann den Befehl »missbrauchen«, um eine Datei umzubenennen. Dazu ist der erste Parameter der alte Dateiname und der zweite Parameter der neue Name (der neue »Ort«).

`mkdir`, `rmdir`

4-17

Erstellen und Löschen von Verzeichnissen

Der `mkdir` (make directory) Befehl

Der Befehl nimmt als Parameter den Namen eines neu anzulegenden Verzeichnisses.

Der `rmdir` (remove directory) Befehl

Der Befehl löscht ein Verzeichnis, das leer sein muss. Typischerweise löscht man dazu mittels `rm *` vorher den Inhalt des Vereichnis.

4.2.2 Rechteverwaltung

`chmod`

4-18

Der Befehl zur Änderung von Rechten

Der `chmod` (change access mode) Befehl

Dieser Befehl bekommt eine *Rechteänderung* und einen *Dateinamen* als Parameter. Die Rechteänderung besteht aus

1. der Personengruppe, um die es geht (also `u`, `g` oder `o`),
2. einem Minus- oder Pluszeichen (für »Recht wird entzogen« oder »Recht wird gegeben«)
3. einem Recht (also `r`, `w` oder `x`).

Beispiel: `chmod o-r geheim.txt`

✎ Zur Übung

```
murmel:~/temp tantau$ ls -l
total 0
-rwxr-----   1 tantau  itcs  0 Nov  1 12:32 beispiel.txt
```

Die Datei `beispiel.txt` soll »welt-lesbar« werden und das Ausführbarkeitsrecht soll gelöscht werden. Wie lauten die nötigen Befehle?

4.2.3 Nützliche Befehle

`echo`, `wget`

Einfach mal etwas sagen

4-19

Der `echo` Befehl

Dieser Befehl gibt einfach alle seine Parameter aus. Dieser Befehl ist hauptsächlich nützlich in der Shell-Programmierung.

Beispiel

```
murmel:~/temp tantau$ echo Hallo Welt
Hallo Welt
murmel:~/temp tantau$
```

Daten aus dem Internet besorgen

4-20

Der `wget` (network downloader) Befehl

Dieser Befehl bekommt als Parameter eine Internetadresse (also die Adresse einer Webseite auf einem anderen Rechner). Die Datei wird geholt und eine lokale Kopie angelegt.

Beispiel

```
murmel:~/temp tantau$ wget http://www.tcs.uni-luebeck.de/index.html
--16:41:04--  http://www.tcs.uni-luebeck.de/index.html
          => 'index.html'
Resolving www.tcs.uni-luebeck.de... 141.83.63.70
Connecting to www.tcs.uni-luebeck.de[141.83.63.70]:80... connected.

...

16:41:04 (8.31 MB/s) - 'index.html' saved [9367/9367]

murmel:~/temp tantau$ head -3 index.html
<!DOCTYPE html PUBLIC "-//W3C//DTD␣HTML␣4.01␣Transitional//EN">
<html>
<head>
murmel:~/temp tantau$
```

`wc`, `diff`, `grep`

Wörter zählen

4-21

Der `wc` (word count) Befehl

Der Befehl bekommt einen Dateinamen als Parameter und zählt die Buchstaben, die Wörter und die Zeilen in der Datei. Mit den Optionen `-c`, `-w` und `-l` kann man entscheiden, was ausgegeben werden soll.

Beispiel

```
murmel:~/temp tantau$ cat beispiel.txt
Hallo Welt.
murmel:~/temp tantau$ wc -w beispiel.txt
       2 beispiel.txt
murmel:~/temp tantau$ wc -l beispiel.txt
       1 beispiel.txt
murmel:~/temp tantau$ wc -c beispiel.txt
      12 beispiel.txt
```

4-22

Vergleichen und Suchen

Der `diff` (difference) Befehl

Parameter des Befehls sind zwei Dateien. Die Ausgabe sind alle Differenzen zwischen den Dateien in einem menschenlesbaren Format.

Der `grep` Befehl

Die Abkürzung steht für »global search for a regular expression and print out matched lines«. Parameter sind ein regulärer Ausdruck und Dateinamen, Ausgabe sind alle Zeilen in den Dateien, in denen der reguläre Ausdruck vorkommt. Beispiel: `grep tantau beispiel.txt` finde alle Zeilen in `beispiel.txt`, in denen `tantau` vorkommt.

4.3 Unix-Shell-Programmierung

4.3.1 Um- und Weiterleitung

4-23

Umleitung der Ein- und Ausgabe

Die *Ausgabe* eines Befehls wird normalerweise einfach in der Shell ausgegeben. Gibt man hinter einem Befehl `> dateiname` an, so wird die Ausgabe stattdessen in die angegebene Datei geleitet. Analog kann man mittels `< dateiname` die *Eingabe* aus einer Datei lesen lassen, statt vom Benutzer.

Beispiel

```
tantau@pcclt02:~/public_html> ls
index.html  index.html~
tantau@pcclt02:~/public_html> ls > listing
tantau@pcclt02:~/public_html> cat listing
index.html
index.html~
listing
```

4-24

Umleitung der Ein- und Ausgabe

Man kann die *Ausgabe* eines Programmes an die *Eingabe* eines anderen Programmes leiten. Dazu wird eine *Pipe* benutzt, ein senkrechter Strich.

Beispiel

```
tantau@pcclt02:~/public_html> ls
index.html  index.html~
tantau@pcclt02:~/public_html> ls | wc -w
2
```

Beispiel

```
murmel:~/Documents/www/webpages/publications tantau$ grep Tantau *.bib | head -10
bibliography.bib:                 Schintke and Till Tantau and Baltasar
bibliography.bib:                 Schintke and Till Tantau and Baltasar
bibliography.bib:                 Till Tantau},
bibliography.bib:                 Till Tantau},
bibliography.bib:  author =       {Jens Gramm and Till Nierhoff and Till Tantau},
bibliography.bib:                 Tantau},
bibliography.bib:                 Tantau},
bibliography.bib:                 Tantau},
bibliography.bib:                 Tantau},
bibliography.bib:  author =       {Richard Karp and Till Nierhoff and Till Tantau},
murmel:~/Documents/www/webpages/publications tantau$
```

4.3.2 Shell-Skripte

Was sind Shell-Skripte?

Ein *Shell-Skript* ist eine Folge von Befehlen, die man immer wieder benutzen möchte. Man schreibt einfach die Folge der Befehle in eine Textdatei, jeden Befehl auf eine neue Zeile. Heißt die Datei beispielsweise `myscript.bash`, so kann man das Skript mittels `bash myscript.bash` aufrufen.

Beispiel

Inhalt von `myscript.bash`:

```
echo Das aktuelle Verzeichnis ist
pwd
echo Es enthaelt die folgende Anzahl an Dateien:
ls | wc -w
```

Aufruf des Skriptes:

```
murmel:~/Documents tantau$ bash myscript.bash
Das aktuelle Verzeichnis ist
/Users/tantau/Documents
Es enthaelt die folgende Anzahl an Dateien:
      16
murmel:~/Documents tantau$
```

Parameter für Shell-Skripte

Shell-Skripte können *Parameter* bekommen. Innerhalb des Skriptes wird jedes Vorkommen von `$1` durch den ersten Parameter ersetzt. Analog wird jedes Vorkommen von `$2` durch den zweiten Parameter ersetzt und so weiter.

Beispiel

Inhalt von `backup.bash`:

```
echo Erstellung eines Backups von $1
cp $1 $1.bak
```

Aufruf des Skriptes:

```
murmel:~/temp tantau$ ls
backup.bash     wichtig.txt
murmel:~/temp tantau$ bash backup.bash wichtig.txt
Erstellung eines Backups von wichtig.txt
murmel:~/temp tantau$ ls
backup.bash     wichtig.txt     wichtig.txt.bak
murmel:~/temp tantau$ diff wichtig.txt wichtig.txt.bak
murmel:~/temp tantau$
```

Zusammenfassung dieses Kapitels

4-27

Shells

Eine *Shell* ist ein Dialog-Programm, das die Kommunikation zwischen einem Nutzer und einer Anwendung ermöglicht. Eine *Unix-Shell* dient dazu, mit einem Unix-Betriebssystem zu kommunizieren. In dem Dialog gibt die Shell einen *Prompt* vor, der Benutzer gibt einen *Befehl* ein, die Shell zeigt *die Ausgaben des Befehlen* an und die Sache beginnt von vorne.

Wichtige Shell-Befehle

Die folgenden Shell-Befehle sollte man kennen: `cd`, `pwd`, `ls`, `cat`, `less`, `head`, `tail`, `rm`, `cp`, `mv`, `chmod`, `echo`, `wget`, `grep`, `wc` und `diff`.

Umleitung

Schreibt man `> dateiname` hinter einen Shell-Befehl, so werden alle Ausgaben des Befehls in die Datei geschrieben. Schreibt man `< dateiname` hinter einen Shell-Befehl, so werden alle Eingaben des Befehls aus der Datei gelesen. Schreibt man `befehl1 | befehl2`, so dienen die Ausgaben von `befehl1` als Eingaben von `befehl2`.

Shell-Skripte

Ein Shell-Skript ist eine Datei, in der jede Zeile einen Shell-Befehl enthält. Man ruft ein Shell-Skript `foo.bash` auf mittels `bash foo.bash parameter1 parameter2 ...`. Dann werden die Zeilen des Skript nacheinander genau so ausgeführt, als hätte man sie »per Hand« nacheinander eingegeben. Wenn im Shell-Skript irgendwo `$1` auftaucht, so wird dieses durch den ersten Parameter ersetzt, analog mit `$2` und so weiter.

Übungen zu diesem Kapitel

Übung 4.1 `wget` benutzen, leicht

Erstellen Sie für diese Aufgabe ein neues Verzeichnis. Wechseln Sie in dieses Verzeichnis und führen Sie dort folgenden Shell-Befehl aus:

```
wget http://www.expasy.org/uniprot/Q640A7.txt
```

Sehen Sie sich nun die Datei `Q640A7.txt` an. Beschreiben Sie kurz den Inhalt der Datei!

Übung 4.2 Shell-Befehle sinnvoll einsetzen, mittel

Die in der vorigen Aufgabe heruntergeladene Datei ist zwar hochinteressant, jedoch etwas kompliziert zu lesen. Unser Ziel ist es nun, mit einigen Shell-Befehlen nur die wichtigsten Informationen zu extrahieren.

Setzen Sie zunächst die Befehle `grep` und `tail` ein, um Folgendes herauszufinden:

- `grep`: Zu welchen Organismus gehört die Datei `Q640A7.txt`? [1]
- `tail`: Wie sieht die Nukleotidsequenz in der Datei `Q640A7.txt` aus?

Mit welchen Parametern für `grep` und `tail` erhalten Sie diese Informationen?

Geben Sie weiterhin eine Kombination der Befehle `grep` und `wc` mit einer Pipe (`|`) an, die ausgibt, wie viele Literaturreferenzen die Datei `Q640A7.txt` enthält![2]

Übung 4.3 Shell-Skripte erstellen, mittel

Schreiben Sie ein Shell-Skript `showprotein.bash`, das die in der letzten Aufgabe erarbeiteten Shell-Befehle hintereinander ausführt! Der Dateiname soll dabei als Parameter übergeben werden:

```
bash showprotein.bash Q640A7.txt
```

Erweitern Sie dann das Shell-Skript so, dass zusätzlich die Definition des Proteins und einige erklärende Hinweise angezeigt werden. Die Ausgabe sollte in etwa so aussehen:

[1] Tipp: Manchmal ist es nützlich, die Zeichensequenz, nach der Sie suchen wollen, in Anführungszeichen setzen. So gibt beispielsweise der Befehl `grep " " test.txt` alle Zeilen der Datei `test.txt` aus, die ein Leerzeichen enthalten.

[2] Noch ein Tipp: Schauen Sie sich die Zeilen an, die mit `RN` beginnen.

```
Analysiere Datei:
Q640A7.txt

Definition:
DE   Glucose-6-phosphate isomerase (EC 5.3.1.9).

Organismus:
OS   Xenopus tropicalis (Western clawed frog) (Silurana tropicalis).

Literaturreferenzen:
3

Sequenz:
SQ   SEQUENCE   553 AA;  62169 MW;  0E9252E164C2654D CRC64;
     MALSCDPVYQ KLSQWYEAHH AGLNMRQMFE ADKGRFSKFS KTLVTDHGDI
     [...]
     TNGLIEFIKK HRG
//
```

Übung 4.4 Parameter bei Shell-Skripten einsetzen, mittel

Der Bezeichner »Q640A7« aus den bisherigen Aufgaben ist eine *UniProt-ID*. Ändern Sie Ihr Shell-Skript aus der letzten Aufgabe so, dass es als Parameter nicht den Namen einer Textdatei, sondern eine beliebige UniProt-ID erhält und die zugehörige Textdatei dann selbstständig herunterlädt. Der Aufruf

```
bash showprotein.bash Q640A7
```

soll also den gleichen Effekt erzielen wie Ihr Skript aus der vorherigen Aufgabe, auch wenn die Datei `Q640A7.txt` nicht bereits heruntergeladen wurde.

Übung 4.5 Vertrautheit mit `grep` erlangen, schwer

Die Verwendung des `tail`-Befehls zur Ausgabe der Sequenz funktioniert nicht immer. Wenn Sie Ihr Skript aus der vorherigen Aufgabe mit der UniProt-ID P25963 durchführen, werden am Ende der Ausgabe einige Zeilen angezeigt, die nicht zur Sequenz gehören.

Können Sie eine bessere Lösung finden, die den Befehl `grep` mit der Option `-A` verwendet?

Übung 4.6 Shell-Befehle memorieren, mittel

Ergänzen Sie die fehlenden Einträge:

Befehl	Erklärung
	gibt den Namen des aktuellen Verzeichnisses aus
	wechselt das aktuelle Verzeichnis
	zeigt Verzeichnisinhalt an
	hängt Dateien aneinander und gibt sie aus
	zeigt den Anfang einer Datei an
	zeigt das Ende einer Datei an
	zeigt eine Datei seitenweise an
	löscht Dateien
	kopiert eine Datei
	verschiebt Dateien
	legt ein neues Verzeichnis an
	löscht ein leeres Verzeichnis
	ändert Zugriffsrechte von Dateien
	gibt Text aus
	lädt Dateien aus dem Internet herunter
	zählt Wörter in einer Datei
	zeigt Unterschiede zwischen Dateien an
	durchsucht eine Datei nach einem Ausdruck
	zeigt den Benutzerhandbucheintrag zu einem Befehl an

Übung 4.7 Shell-Skript entwerfen, leicht

Schreiben Sie ein Shell-Skript `inspectdir.bash`, das folgende Aufgaben erfüllt:

- In das als Parameter übergebene Verzeichnis wechseln
- Namen und Inhalt dieses Verzeichnisses ausgeben

– Namen und Inhalt des übergeordneten Verzeichnisses ausgeben

Beim Aufruf soll also Folgendes passieren:

```
textor@elefant:~> bash inspectdir.bash /usr/local/2007-ws-info-a/exercises/
/usr/local/2007-ws-info-a/exercises
class_materials  homework_assignments
/usr/local/2007-ws-info-a
evaluations  exam  exercises  lecture_slides  planning  projects  temp  www
```

Sie dürfen die Ausgabe gerne mit `echo`-Befehlen etwas ausschmücken.

Übung 4.8 Shell-Skript entwerfen, schwer

Schreiben Sie ein Shell-Skript `archive.bash`, das folgende Aufgaben erfüllt:

– Ein Verzeichnis `archive` erstellen
– Die als Parameter übergebene Datei dort hineinkopieren
– Die als Parameter übergebene Datei löschen
– Dabei jeweils ausgeben, was gerade gemacht wird

Beim Aufruf soll also Folgendes passieren:

```
textor@elefant:~> ls
archive.bash test.txt
textor@elefant:~> bash archive.bash test.txt
Erstelle Verzeichnis archive ...
Kopiere test.txt nach archive ...
Loesche test.txt ...
Fertig!
textor@elefant:~> ls
archive.bash archive
textor@elefant:~> ls archive
test.txt
```

Sie dürfen wieder die Ausgabe mit `echo`-Befehlen etwas ausschmücken.

Übung 4.9 Shell-Skript entwerfen, mittel

Schreiben Sie ein Shell-Skript `countsheep.bash`, das in der als Parameter übergebenen Datei nach dem Wort »Schaf« sucht und ausgibt, in wie vielen *Zeilen* dieses Wort vorkommt.

Hinweis: Hierzu müssen Sie zwei Befehle mit einer *Pipe* (`|`) kombinieren.

```
textor@elefant:~> bash countsheep.bash hundertschafe.txt
100
```

Übung 4.10 Shell-Skript entwerfen, mittel

Schreiben Sie ein Shell-Skript `summary.bash`, das folgende Aufgaben erfüllt:

– Die ersten zwei Zeilen der als Parameter übergebenen Datei anzeigen
– Die letzten zwei Zeilen der als Parameter übergebenen Datei anzeigen
– Anzeigen, wie viele Wörter die als Parameter übergebene Datei hat

```
textor@elefant:~> bash summary.bash maerchen.txt
Es war einmal im Lummerland,
dass ein Birnbaum im Garten stand.
Und wenn sie nicht tot sind,
dann leben sie auch heute noch gluecklich!
14444
```

Prüfungsaufgaben zu diesem Kapitel

Übung 4.11 Zugriffsrechte setzen, leicht, original Klausuraufgabe, mit Lösung

In einem Linux-Rechnerpool sind die Rechte Ihres Home-Verzeichnisses wie folgt gesetzt:

```
/home/ihrname          rwx------
```

Ihr Home-Verzeichnis enthält das Verzeichnis `projects` und die Datei `protokoll.pdf` mit folgenden Zugriffsrechten:

```
projects               rwxrwxrwx
protokoll.pdf          rwxrwxrwx
```

Sie wollen nun die Rechte wie folgt ändern: Mitglieder Ihrer Arbeitsgruppe sollen zwar die Datei `protokoll.pdf` öffnen, aber nicht löschen oder ändern können. Der Rest Ihres Home-Verzeichnisses soll für jeglichen Zugriff aller Benutzer außer Ihnen gesperrt sein. Selbst das Auflisten Ihres Home-Verzeichnisses soll anderen Benutzern nicht möglich sein. Sie selber sollen aber weiterhin Vollzugriff auf Ihr Homeverzeichnis und den Inhalt haben.

1. Wie müssen Sie die Zugriffsrechte für Ihr Home-Verzeichnis, das Verzeichnis `projects` und die Datei `protokoll.pdf` setzen, um dies zu erreichen?
2. Welche Shell-Befehle müssen Sie dafür ausführen?

Übung 4.12 Shellskript erstellen, mittel, original Klausuraufgabe, mit Lösung

Schreiben Sie ein Shell-Skript `prepostfix.bash`, das die erste Zeile einer als Parameter übergebenen Textdatei an das Ende dieser Datei anfügt. Sei zum Beispiel im aktuellen Verzeichnis eine Datei `test.txt` mit folgendem Inhalt vorhanden:

```
Blutegel, Moskitos, Bienen.
Eidechsen, auch Ameisen.
Spinnen und Flöhe auch?
```

Dann soll diese nach dem Aufruf

```
> bash prepostfix.bash test.txt
```

folgenden Inhalt haben:

```
Blutegel, Moskitos, Bienen.
Eidechsen, auch Ameisen.
Spinnen und Flöhe auch?
Blutegel, Moskitos, Bienen.
```

Ihr Skript darf temporäre Dateien anlegen, muss diese aber wieder löschen. Sie dürfen davon ausgehen, dass das aktuelle Verzeichnis bis auf die Datei `prepostfix.bash` und die als Parameter übergebene Datei keine weiteren Dateien enthält. Sie können diese Aufgabe beispielsweise mit den Befehlen `head`, `cat`, `cp` und `rm` lösen.

Übung 4.13 Shellverarbeitung nachvollziehen, leicht, typische Klausuraufgabe

Die Datei `hello.txt` hat folgenden Inhalt:

```
Hello, again!
```

Die Datei `magic.bash` hat folgenden Inhalt:

```
cp $1.txt $1.again.txt
cat $1.txt $1.again.txt > $1.double.txt
tail -2 $1.double.txt > $1.txt
```

Bis auf diese beiden Dateien sei das aktuelle Verzeichnis leer.

Geben Sie an, welche Dateien das aktuelle Verzeichnis nach Ausführen der Kommandozeile

```
> bash script.bash hello
```

enthält, und geben Sie den Inhalt dieser Dateien an (mit Ausnahme der Datei `script.bash`).

Übung 4.14 Shellskiprte erstellen, mittel, typische Klausuraufgabe

Schreiben Sie ein Shell-Skript `double.bash`, das den Inhalt einer als Parameter übergebenen Textdatei verdoppelt. Wenn also zum Beispiel im aktuellen Verzeichnis die Datei `hello.txt` mit dem Inhalt

```
Hello, again!
```

vorhanden ist, soll diese nach dem Aufruf

```
> bash double.bash hello.txt
```

folgenden Inhalt haben:

```
Hello, again!
Hello, again!
```

Ihr Skript darf temporäre Dateien anlegen, muss diese aber wieder löschen. Sie dürfen davon ausgehen, dass das aktuelle Verzeichnis bis auf die Datei `double.bash` und die als Parameter übergebene Datei keine weiteren Dateien enthält.

Übung 4.15 Shell-Programmierung, leicht bis mittel, original Klausuraufgabe, mit Lösung

Die Datei `nummern.txt` enthalte die folgenden drei Zeilen:

```
Ilka      286221
Minka     678516
Julia     752894
```

Weiter sei eine Datei `gehaltsliste.txt` mit folgendem Inhalt gegeben:

```
C.S.     T.T.     R.R.
1000     5000     10000
```

1. Mit welchen Shell-Befehl können Sie sich die Zeile von Minka ausgeben lassen?
2. Schreiben Sie ein Shell-Skript, das die Zeile einer als Parameter übergebenen Person aus der Datei `nummern.txt` heraussucht! (Geben Sie nur den Inhalt des Skripts an).
3. Sie möchten nur die Gehaltsdaten aus der Datei `gehaltsliste.txt`, aber nicht die Namen, in eine andere Datei `gehaelter.txt` kopieren. Geben Sie einen Shell-Befehl dafür an!
4. Angenommen, es gibt einen Shell-Befehl `mean`, der den Mittelwert einer Reihe von Zahlen ausgibt, die von der Standardeingabe gelesen werden. Mit welcher Kombination von Befehlen können Sie dann das mittlere Gehalt der Personen aus der Datei `gehaltsliste.txt` anzeigen?

Übung 4.16 Messreihen auswerten, leicht bis mittel, original Klausuraufgabe

Für Ihre Bachelorarbeit haben Sie zwei Messreihen ermittelt, die in den Dateien `a.txt` ubd `b.txt` vorliegen. Dabei wird jeweils ein Messwert pro Zeile angegeben. Beispielsweise hat die Datei `a.txt` folgenden Inhalt:

```
3
5
1
```

Weiter sei auf Ihrem Linux-Rechner ein Shell-Befehl `variance` installiert, der die Varianz einer Zahlenreihe errechnen kann, die in einer Textdatei gegeben ist. Der Befehl kann z.B. wie folgt benutzt werden:

```
ich@meinpc:~> variance a.txt
4
ich@meinpc:~>
```

Hinweis: Es spielt für diese Aufgabe keine Rolle, wenn Sie sich nicht erinnern, was eine Varianz ist und wie man sie berechnet.

1. Mit welchem Shell-Befehl können Sie sich direkt die Anzahl der Messwerte in `b.txt` anzeigen lassen?
2. Benutzen Sie die Befehle `grep` und `wc`, um herauszufinden, wie oft in der Datei `b.txt` der Messwert `5` vorkommt!
3. Mit welchen beiden Shell-Befehlen können Sie die Messwerte aus den Dateien `a.txt` und `b.txt` zu einer neuen Datei `c.txt` zusammenfügen und die Varianz für den gesamten Datensatz ermitteln?
4. Den Befehl `variance` kann man auch ohne Argumente aufrufen. Er liest dann von der Standardeingabe. Mit welchem Befehl (bzw. welcher Kombination von Befehlen) können Sie die Gesamtvarianz der Messreihen in den Dateien `a.txt` und `b.txt` ermitteln, ohne eine neue Datei zu erzeugen?

5-1

Kapitel 5
Reguläre Ausdrücke
Pattern Matching

Lernziele dieses Kapitels

1. Konzept des regulären Ausdrucks verstehen
2. Probleme als reguläre Ausdrücke beschreiben können
3. Mächtigkeit regulärer Ausdrücke kennen

5-2

Inhalte dieses Kapitels

5.1	Einführung	42
5.1.1	Muster-Suche in Texten	42
5.1.2	Theoretischer Hintergrund	42
5.1.3	Praktischer Hintergrund	43
5.2	Reguläre Ausdrücke	43
5.2.1	Die einfachsten Ausdrücke	43
5.2.2	Die Alternative	44
5.2.3	Die Verkettung	44
5.2.4	Der Kleene-Stern	45
5.2.5	Reguläre und arithmetische Ausdrücke	45
5.3	Anwendungen	46
5.3.1	Einsatzgebiete	46
5.3.2	Referenz: Reguläre Ausdrücke in Grep	47
	Übungen zu diesem Kapitel	48

Worum es heute geht

Die Suche in und die Überprüfung von Texten ist eine der Hauptbeschäftigungen von Computern. Ständig muss etwas gefunden werden wie Dateinamen, Gensequenzen oder Zahlen in Tabellen; ständig muss etwas überprüft werden wie die korrekte Form von Postleitzahlen, E-Mail-Adressen, Internet-Adressen oder Pizzabestellscheinformularadressangabefeldern.

Wie wir in späteren Kapiteln sehen werden, ist es gar nicht so leicht, Programme zu schreiben, die solches Suchen und Überprüfungen effizient durchführen. Um so besser, dass die Informatik seit langer Zeit ein Konzept zu bieten hat, das diese Aufgaben ungemein erleichter: Mit *regulären Ausdrücken* kann man herausfinden, ob Zeichenketten mit bestimmten – teils recht komplizierten – Eigenschaften in einem Text vorkommen.

Leider sind reguläre Ausdrücke oft, sagen wir, kryptisch. Selbst ein Profi wird den regulären Ausdruck `"[.?!][]\"')]*\\($\\| $\\|\t\\| \\)[ \t\n]*"` nicht sofort verstehen. Dies ist laut Emacs-Handbuch »a simplified version of the regexp that Emacs uses, by default, to recognize the end of a sentence together with any whitespace that follows.«

Auch wenn reguläre Ausdrücke fürchterlich komplex und unlesbar werden können, so ist die Grundidee doch recht einfach: Jeder Ausdruck beschreibt eine Menge von Wörtern, auf die er *passt*. Die einfachsten Ausdrücke passen einfach nur auf ein einzelnes Wort und sonst nichts. Man kann dann aber Ausdrücke zu komplexeren Ausdrücke zusammenbauen und diese beschreiben dann auch größere und komplexere Mengen von Wörtern. Die Hauptschwierigkeit bei regulären Ausdrücken ist auch nicht, dieses Konzept zu verstehen. Problematisch ist, dass sich keine einheitliche Syntax herausgebildet hat, wie man die Ausdrücke nun aufschreiben sollte. Hier kocht jeder sein eigenes Süppchen, welche wir nun alle auslöffeln müssen.

5.1 Einführung

5.1.1 Muster-Suche in Texten

5-4

Worum geht es bei der Mustersuche?

Das Problem, ein *Muster* innerhalb eines Textes zu finden, taucht in vielen Kontexten auf:

- Man sucht in einem elektronischen Dokument nach dem Vorkommen eines bestimmten Wortes; beispielsweise das Wort »Energiewende« in einem Koalitionsvertrag.
- Eine Suchmaschine möchte anzeigen, wo überall auf einer Webseite das Wort »flauschig« vorkommt.
- Man sucht in einer DNA-Sequenz nach allen Vorkommen einer bestimmten Basensequenz.

Das gesuchte Muster ist aber in der Regel *kein einzelnes Wort*:

- Statt »flauschig« würde man auch »Flauschig« finden wollen. Eventuell auch »FLAUSCHIG«.
- Sucht man die Basensequenz eines Gens, so würde man an Polymorphismus-Stellen auch zulassen, dass dort im Text vom Muster abgewichen wird.

5-5

Was ist ein Muster?

Muster und Muster-Suche

- Ein *Muster* (englisch *pattern*) *beschreibt* eine *Menge von möglichen Worten*.
- Bei der *Muster-Suche* (englisch *pattern search*) sind ein Muster und ein (längerer) Text gegeben.
  Ziel ist es, *Stellen im Text zu finden*, an denen ein Wort steht, das vom Muster beschrieben wird.

Beispiel

Wir werden bald definieren, dass das Muster `ab[cd]` die beiden Strings `abc` und `abd` beschreibt.
Würde man nach `ab[cd]` in `Ich werd' das abchecken.` suchen, so gäbe es genau einen Treffer, nämlich am Anfang von `abchecken`.

5-6

Was im Folgenden passiert.

- Es werden *reguläre Ausdrücke* eingeführt; dies ist eine Art, Muster aufzuschreiben.
- Es wird erklärt, welche Mengen an Wörtern ein regulärer Ausdruck beschreibt.
- Es wird *nicht* erklärt, wie die Algorithmen funktionieren, die nach Ausdrücken in Texten suchen.
  Es sei lediglich erwähnt, dass es *sehr schnelle* Algorithmen gibt und dass man *mehrere Kapitel über Theoretische Informatik* braucht, um diese wirklich zu verstehen.

5.1.2 Theoretischer Hintergrund

5-7

Wörter im Sinne der Theorie

▶ Definition

1. Ein *Alphabet* ist eine nichtleere, endliche Menge.
2. Die Elemente eines Alphabets nennt man *Buchstaben* oder auch *Symbole*.
3. Eine *endliche Folge* von Buchstaben nennt man ein *Wort*.

Beachte:

- Alphabete werden häufig mit griechischen Großbuchstaben bezeichnet, also Γ oder Σ. Praktische Beispiele sind die Menge $\{0,1\}$ (bei Informatikern beliebt), die Menge $\{A, C, G, T\}$ (bei Biologen beliebt) und die Zeichenmenge des UNICODE.
- In »Wörter« können Leerzeichen (!) als Symbole auftauchen.
- Es gibt auch ein *leeres Wort*, abgekürzt ε oder λ, das Länge Null hat.

Die vornehme Bezeichnung für Mengen von Wörtern: Sprachen. 5-8

▶ Definition

Eine *Sprache* ist eine Menge von Wörtern über einem Alphabet.

Sprachen müssen weder sinnvoll noch interessant sein, sie können endlich sein oder auch unendlich. Ein Beispiel einer endlichen Sprache ist $\{AAA, AAC, AAT\}$; ein Beispiel einer unendlichen Sprache die Menge aller Basensequenzen, die `TATA` genau zweimal enthalten.

Muster-Suche, noch einmal etwas formaler. 5-9

Ein *regulärer Ausdruck* wird ein *einzelnes Wort* sein, wie `ab[cd]` oder `x|y|z*`. Wir werden für jeden regulären Ausdruck R eine bestimmte Sprache $L(R)$ angeben, von der wir sagen werden, dass R *sie beschreibt*. Beispielsweise wird $L(\texttt{ab[cd]}) = \{abc, abd\}$ gelten. Das *Muster-Such-Problem* ist dann, zu einem gegeben regulären Ausdruck R und einem längeren Wort w ein Teilwort u in w zu finden mit $u \in L(R)$.

5.1.3 Praktischer Hintergrund

Das Programm Grep. 5-10

Das Programm Grep löst das Muster-Such-Problem, wobei GREP für »Global search for REgular Patterns« steht. Parameter sind

1. ein regulärer Ausdruck und
2. der Name einer Datei, in der nach Vorkommen des Musters gesucht werden soll.

Findet Grep das Muster in einer Datei, so gibt es die Zeile aus, in der das Muster vorkommt; mit der Option `-n` kann man sich die Zeilennummer auch anzeigen lassen. Aus historischen Gründen sollte man auch immer die Option `-E` angeben, damit die »moderne« Syntax für reguläre Ausdrücke benutzt wird.

Beispiel

```
maus:media tantau$ grep -n -E "Philosophie" faust.txt
536:Habe nun, ach! Philosophie,
maus:media tantau$
```

5.2 Reguläre Ausdrücke

5.2.1 Die einfachsten Ausdrücke

Die einfachsten regulären Ausdrücke sind einfache Wörter. 5-11

▶ Definition: Reguläre Ausdrücke 1 – Atomare Ausdrücke

Sei Σ ein Alphabet, in dem bestimmte Sonderzeichen nicht vorkommen. Dann ist jedes Wort w über diesem Alphabet ein regulärer Ausdruck, im Folgenden mit dem Buchstaben R abgekürzt.
Die von R beschriebene Sprache $L(R)$ ist einfach $\{w\}$, sie enthält also einfach nur w.

Für diese einfachen Ausdrücke ist der formale Apparat mit Wörtern und Sprachen und $L(R)$ natürlich Overkill. Trotzdem ist schon hier das *Muster-Such-Problem* interessant: Es ist einfach das Problem, in einem Text ein ganz bestimmtes Wort zu finden.

Beispiele 5-12

```
maus:media tantau$ grep -n -E "heilig" faust.txt
1799:Knurre nicht Pudel! Zu den heiligen Tönen,
1823:Das heilige Original
1951:Ich versenge dich mit heiliger Lohe!
4430:Und hier mit heilig reinem Weben
4527:          Und warf den heiligen Becher
4595:Ob das Ding heilig ist oder profan;
4808:Bey'm heiligen Antonius,
5979:Ihr selig machend ist, sich heilig quäle,
7902:Was will der an dem heiligen Ort?
7917:Ihr Engel! Ihr heiligen Schaaren,
maus:media tantau$
```

5.2.2 Die Alternative

5-13

Dies oder jenes. . .

▶ Definition: Reguläre Ausdrücke 2 – Alternativen
Sind R_1 und R_2 beides reguläre Ausdrücke, so auch $R_1 \mid R_2$.
Die von $R_1 \mid R_2$ beschriebene Sprache $L(R_1 \mid R_2)$ ist die Vereinigung der Sprachen $L(R_1)$ und $L(R_2)$:

$$L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$$

Ein senkrechter Strich kann also genutzt werden, um *alternativ* nach den einen Wörtern oder nach den anderen Wörtern zu suchen. Der senkrechte Strich gehört zu den *Sonderzeichen*. Will man tatsächlich nach einem senkrechten Strich suchen, so muss man ihm einen Backslash voranstellen.

5-14

Beispiele

```
maus:media tantau$ grep -n -E "Philosophie|Theologie|Medicin"
    faust.txt
536:Habe nun, ach! Philosophie,
538:Juristerey und Medicin,
539:Und leider auch Theologie!
2996:Fast möcht' ich nun Theologie studiren.
3029:Wollt ihr mir von der Medicin
3042:Der Geist der Medicin ist leicht zu fassen;
maus:media tantau$
```

5.2.3 Die Verkettung

5-15

Erst dies, dann das. . .

▶ Definition: Reguläre Ausdrücke 3 – Konkatenation
Sind R_1 und R_2 beides reguläre Ausdrücke, so auch R_1R_2.
Die von R_1R_2 beschriebene Sprache $L(R_1R_2)$ ist die so genannte *Verkettung* der Sprachen $L(R_1)$ und $L(R_2)$:

$$L(R_1R_2) = \{uv \mid u \in L(R_1), v \in L(R_2)\}.$$

Die Sprache $L(R_1R_2)$ enthält alle Wörter w, die man irgendwie in zwei Teile $w = w_1w_2$ trennen kann, so dass der *Präfix* in $L(R_1)$ und der *Suffix* in $L(R_2)$ liegt. Man kann *runde Klammern* benutzen, wenn Verkettungen und Alternativen gemischt werden, um die Reihenfolge der Auswertung vorzugeben. Wieder kann man auch nach Klammern selbst suchen, wenn man einen Backslash voranstellt.

5-16

Beispiele

```
maus:media tantau$ grep -n -E "(Phi|Theo)lo(soph|g)ie" faust.txt
536:Habe nun, ach! Philosophie,
539:Und leider auch Theologie!
2996:Fast möcht' ich nun Theologie studiren.
maus:media tantau$
```

✎ Zur Übung
Wie lautet $L(R)$ jeweils für folgende Ausdrücke R?

1. `(0|1|2)(3|4|5)`
2. `0|((1|2|3)3(4|5))(6|7)`
3. `(a||b)(c|d)`
4. `\||(\||\()(\)|\\|\|)`
   (Keine Angst, so etwas kommt zwar in Computerprogrammen, aber nicht in Klausuren vor.)

5.2.4 Der Kleene-Stern

Dies oder das oder dies oder das. . .

5-17

▶ Definition: Reguläre Ausdrücke 4 – Wiederholungen

Ist R ein regulärer Ausdruck, so auch R^*.
Die von R^* beschriebene Sprache $L(R)$ ist der so genannte *Kleene-Stern* von $L(R)$:

$$L(R^*) = \{u_1 \dots u_n \mid u_i \in L(R)\}$$

Die Sprache $L(R^*)$ enthält alle Wörter, die beliebige Aneinanderreihungen von *Wörtern in* $L(R)$ sind. In der Aneinanderreihungen kann ein Wort beliebig oft vorkommen, muss es aber nicht. Auch das leere Wort ε liegt in $L(R^*)$. Die Sprache $L(R^*)$ ist *unendlich*.

Beispiele

5-18

```
maus:media tantau$ grep -n -E "as*e" faust.txt
...
6465:Solvet saeclum in favilla.
6602:Und die langen Felsennasen,
6604:Wie sie schnarchen, wie sie blasen!
6606:     Durch die Steine, durch den Rasen
6627:Aus belebten, derben Masern
6628:Strecken sie Polypenfasern
6648:Fasse wacker meinen Zipfel!
...
maus:media tantau$
```

✎ Zur Übung

1. Welche Wörter der Länge ≤ 6 gehören zu $L(\texttt{(ab|cab)*})$?
2. Beschreiben Sie $L(R)$ für folgenden regulären Ausdruck R:
   `aug((a|u|g|c)(a|u|g|c)(a|u|g|c))*(uag|uaa|uga)`
3. Geben Sie drei Wörter an, die vom Ausdruck `0(1*0|2*0)*3` erzeugt werden.
4. Finden Sie einen regulären Ausdruck, der alle Worte über $\{a,b,c\}$ beschreibt, die mit a oder b beginnen und deren drittletzter Buchstabe ein c ist.

5.2.5 Reguläre und arithmetische Ausdrücke

Analogie von regulären und arithmetischen Ausdrücken.

5-19

Syntaktische Analogien

Arithmetische Ausdrücke

Bestandteile:

1. Zahlen
2. Unäre Operationen wie Wurzel.
3. Binäre Operationen wie
   - Addition
   - Multiplikation
   - Potenz
4. Klammern

Reguläre Ausdrücke

Bestandteile:

1. Ein-Wort-Sprachen
2. Unäre Operationen wie Kleene-Stern.
3. Binäre Operationen
   - Vereinigung
   - Verkettung
4. Klammern

5-20

Analogie von regulären und arithmetischen Ausdrücken.

Semantische Analogien

Arithmetische Ausdrücke

1. Ein arithmetischer *Ausdruck* und sein *Wert* sind verschiedene Dinge:
   Der *Ausdruck* $(5+6)\cdot 2$ hat den *Wert* 22.
2. Eine Zeichenfolge wie 22 kann man sowohl als Ausdruck als auch als Wert lesen.

Reguläre Ausdrücke

1. Ein regulärer *Ausdruck* und sein *Wert* sind verschiedene Dinge:
   Der *Ausdruck* `(a|b)*a` hat den *Wert* $\{w \mid w$ besteht aus a's und b's und endet auf a$\}$.
2. Der Ausdruck `hallo` hat als Wert gerade die Ein-Wort-Sprache $\{$hallo$\}$.

5.3 Anwendungen

5.3.1 Einsatzgebiete

5-21

Einsatzgebiete von regulären Ausdrücken.

- Innerhalb von Programmen werden reguläre Ausdrücke zur *Eingabekontrolle* benutzt.
- Innerhalb von Programmen werden reguläre Ausdrücke zum *Parsen von Eingaben* benutzt.
  Beispiel: Zerlegung einer URL in ihre Bestandteile.
- Menschen können reguläre Ausdrücke für *Suchanfragen* benutzen.
  - In der Datenbanksprache »SQL« kann man statt `like` auch `regexp` benutzen, um nach Attributen zu filtern.
  - In Microsoft Word kann man im Suchen-Dialog einen *Mustervergleich* einschalten.
  - Das Programm `grep` sucht nach einem Vorkommen eines regulären Ausdrucks in einer Datei.

5-22

Probleme beim Einsatz von regulären Ausdrücken.

Notationsprobleme: Will man reguläre Ausdrücke aufschreiben, so muss man irgendwie alles mit ASCII-Zeichen aufschreiben. Leider hat sich keinerlei Standard zum Aufschreiben von regulären Ausdrücken durchgesetzt.
Effizienzprobleme: Manche Arten regulärer Ausdrücke lassen sich *effizienter behandeln* als andere. Deshalb ist es manchmal sinnvoll, nur *bestimmte Arten von regulären Ausdrücken zuzulassen*. Einige Programme vereinfachen deshalb die Notation, sind dann aber inkompatibel mit anderen Notationen.

5-23

Typische Notationen für reguläre Ausdrücke.

Auch wenn die Notation für reguläre Ausdrücke nicht einheitlich ist, so gilt wenigstens *oft*:

- Statt `a|b|c|d|e|x|y|z` kann man auch schreiben `[abcdexyz]` oder auch kürzer `[a-exyz]`.
- Statt einer Aufzählung aller möglichen Zeichen kann man einfach einen Punkt schreiben, er steht für ein beliebiges Zeichen.
- Statt `(|`A`)`, was »gar kein Vorkommen von A oder ein Vorkommen von A« bedeutet, kann man schreiben A`?`.
- Statt AA^*, was »mindestens ein Vorkommen von A« bedeutet, kann man schreiben A`+`.
- Schreibt man $A\{n\}$, wobei n eine Zahl ist, so bedeutet dies »genau n Vorkommen von A«.
- Schreibt man $A\{n,m\}$, so bedeutet dies »mindestens n und höchstens m Vorkommen von A«.
- Sucht man tatsächlich nach einem Sonderzeichen, so kann man ihm einen Backslash voranstellen.

Skript-Referenz

5.3.2 Referenz: Reguläre Ausdrücke in Grep

Die folgenden Regeln geben einen kurzen Überblick über die genaue `grep`-Regel-Syntax. In der darauf folgenden Tabelle sind einige Beispiele angegeben, die diese Regeln verdeutlichen.

1. Die Sprache aller Zeichenketten, die das Zeichen `a` enthalten, wird einfach durch den regulären Ausdruck `a` beschrieben. Dasselbe gilt für alle anderen Zeichen, sofern sie keine Sonderzeichen sind.
2. Die folgenden Zeichen sind Sonderzeichen:

   `\ . ^ $ [ ] ( ) * ? + | /`

   Sonderzeichen haben besondere Funktionen in regulären Ausdrücken. Um trotzdem Sprachen zu definieren, die diese Zeichen enthalten, stellen Sie dem entsprechenden Zeichen den Backslash `\` voran. *(Beispiel 1)*
3. Das Sonderzeichen `.` steht für genau ein beliebiges Zeichen (auch ein Leerzeichen). *(Beispiele 3, 14, 15)*
4. Das Sonderzeichen `^` steht für den Anfang einer Zeichenkette. *(Beispiele 4, 12–15)*
5. Das Sonderzeichen `$` steht für das Ende eines Wortes. *(Beispiele 5, 12–15)*
6. Mit eckigen Klammern können mehrere Möglichkeiten für genau ein Zeichen definiert werden. *(Beispiele 6, 7, 13)*
7. Folgt auf eine öffnende eckige Klammer das Zeichen `^`, so wird die Zeichenmenge in der Klammer negiert (es bedeutet *hier* also *nicht* den Anfang der Zeichenkette). *(Beispiel 7)*
8. Zeichenmengen in eckigen Klammern können mit dem Zeichen `-` abgekürzt definiert werden. Statt `[0123456789]` kann man also `[0-9]` schreiben. *(Beispiel 13)*
9. Reguläre Ausdrücke werden verkettet, indem sie einfach hintereinander geschrieben werden. Der Ausdruck `ab` steht also für die Sprache aller Zeichenketten, die irgendwo den Teilstring `ab` enthalten. *(Beispiele 8–15)*
10. Die Zeichen `?`, `*` und `+` können regulären Ausdruck nachgestellt werden *(Beispiele 9–15)*

    `?` : Der vorangehende Ausdruck soll einmal oder keinmal vorkommen.
    `*` : Der vorangehende Ausdruck soll keinmal oder beliebig oft direkt hintereinander vorkommen.
    `+` : Der vorangehende Ausdruck soll einmal oder öfter direkt hintereinander vorkommen.
11. Durch runde Klammern können reguläre Ausdrücke gruppiert werden. Dies ist im Zusammenhang mit den Operationen `?`, `*`, `+` nützlich *(Beispiel 12)*: Ohne Gruppierung beziehen sich diese Operatoren immer nur auf das Zeichen direkt davor *(Beispiele 9–11)* beziehungsweise die eckige Klammer direkt davor *(Beispiel 13)*.
12. Mit dem Sonderzeichen `|` können zwei reguläre Ausdrücke vereinigt (»verodert«) werden. *(Beispiel 15)*

Beispiele für die Regeln:

	Ausdruck	*Die erzeugte Sprache enthält alle Zeichenketten, die …*
1	`\?`	… ein Fragezeichen enthalten.
2	`\\`	… einen Backslash enthalten.
3	`.`	… mindestens ein Zeichen lang sind.
4	`^a`	… mit `a` beginnen.
5	`a$`	… mit `a` enden.
6	`[abc]`	… irgendwo ein `a`, `b` oder `c` enthalten.
7	`[^abc]`	… irgendwo ein Zeichen enthalten, das kein ein `a`, `b` oder `c` ist.
8	`abc`	… den Teilstring `abc` enthalten.
9	`ab?a`	… den Teilstring `aa` oder den Teilstring `aba` enthalten.
10	`ab*a`	… einen der Teilstrings `aa`, `aba`, `abba`, `abbba` und so weiter enthalten.
11	`ab+a`	… einen der Teilstrings `aba`, `abba`, `abbba` und so weiter enthalten.
12	`^a(ab)*b$`	… mit einem `a` beginnen, gefolgt von beliebig vielen Wiederholungen von `ab`, und mit einem `b` enden.
13	`^[_a-zA-Z][_a-zA-Z0-9]*$`	… gültige Java-Bezeichner sind.
14	`^A.*s$`	… mit `A` beginnen und mit `s` enden.
15	`^A.*s$\|^.s$`	… mit `A` beginnen und mit `s` enden oder mit `s` enden und nur zwei Zeichen lang sind (nicht »entweder oder«!).

Zusammenfassung dieses Kapitels

5-24

Worte und Sprachen
Ein *Wort* ist eine endliche Folge von Symbolen über einem *Alphabet*. Eine *Sprache* ist eine beliebige (!) Menge von Worten über einem festen Alphabet.

Regulärer Ausdruck
Ein *regulärer Ausdruck R* ist selbst ein Wort über einem speziellen Alphabet, das einige Sonderzeichen enthält. Er *beschreibt* eine Sprache $L(R)$ nach den in diesem Kapitel beschriebenen Regeln.

Das Muster-Such-Problem
Beim Muster-Such-Problem hat man einen regulären Ausdruck R gegeben und einen Text und man *sucht* nach Vorkommen von Worten aus $L(R)$ in dem Text. Das Problem ist auch bei *Eingabekontrollen* wichtig.

Übungen zu diesem Kapitel

Übung 5.1 Reguläre Ausdrücke auswerten, leicht

Hier finden Sie fünf reguläre Ausdrücke in `grep`-Syntax. Streichen Sie für jeden regulären Ausdruck die Wörter durch, die nicht zu dem Ausdruck passen.

1. `a`
   a, aa, bab, labern, faseln, klönen
2. `a(a|b)*b`
   aaab, abab, caabb, lab, bbba, abba
3. `[ab]..[ab]`
   alpha, beta, gamma, gaga, dudu, aha
4. `^a|a$`
   Hannah, Anna, Mama, anders, Anders
5. `[Ll]e(hr|rn)e..`
   Lehrer, leer, leerer, lernen, lernt, lehren, erlernen, ehre

Übung 5.2 Ausdruck für Binärstrings gerader Länge, einfach

Geben Sie einen regulären Ausdruck in `grep`-Syntax an, der die Sprache aller Zeichenketten beschreibt, die gerade Länge haben und nur aus Nullen und Einsen bestehen!

Übung 5.3 Regulären Ausdruck testen, mittel

Die Sprache L sei durch folgenden regulären Ausdruck gegeben:

```
^c*(gg*|aa*)(cg*|ca*)*$
```

Welche der folgenden Wörter liegen in L: *ccc*, *cggcaacaaa*, *cgacga*, *ccggccgg*?

Übung 5.4 Regulärer Ausdruck für Exponentialschreibweise, schwer

Sehr große und sehr kleine Zahlen kann man in Java und in anderen Programmiersprachen in der sogenannten *Exponentialnotation* aufschreiben, die der in der Wissenschaft gebräuchlichen Darstellung mit Zehnerpotenzen entspricht:

Zahl	Darstellung mit Zehnerpotenzen	Exponentialnotation
90 000 000 000	$9 \cdot 10^{10}$	`9E12`
3 500 000 000 000	$3{,}5 \cdot 10^{12}$	`3.5E12`
0,000 000 000 001 234	$1{,}234 \cdot 10^{-12}$	`1.234E-12`

Allgemein besteht eine Zahl in Exponentialnotation aus einer gewöhnlichen **`double`**-Zahl (Mantisse) gefolgt von dem Buchstaben `E` gefolgt von einer positiven oder negativen Ganzzahl (Exponent). Führende Nullen sind weder bei der Mantisse noch beim Exponenten erlaubt. (Die Zahl 0 selber oder der Exponent 0 sollen aber trotzdem darstellbar bleiben!)

Geben Sie einen regulären Ausdruck für Zahlen in Exponentialnotation an!

Prüfungsaufgaben zu diesem Kapitel

Übung 5.5 Regulärer Ausdruck für grade Zahlen in Dezimaldarstellung, schwer, original Klausuraufgabe, mit Lösung

Geben Sie einen regulären Ausdruck an, der die Sprache aller geraden Zahlen in Dezimaldarstellung definiert. Führende Nullen sollen dabei nicht erlaubt sein, auch der leere String ist keine gerade Zahl.

Insbesondere soll Ihr regulärer Ausdruck zu allen Zeichenketten aus der linken Spalte der folgenden Tabelle passen, aber zu keiner Zeichenkette aus der rechten Spalte:

Soll passen	Soll nicht passen
0	ε
4	5
12	11
124	012

Sie können den regulären Ausdruck wahlweise in der in der Vorlesung eingeführten mathematischen Notation oder in `grep`-Syntax angeben.

Übung 5.6 Regulären Ausdruck analysieren, leicht, original Klausuraufgabe, mit Lösung

Wie lautet $L(R)$ jeweils für den Ausdruck $R =$ `\||(\||\()(\)|\\|\|)`?

Teil II

Seitenbeschreibungssprachen

Nehmen Sie ein beliebige bedruckte Seite (wie zum Beispiel diese hier, die Sie gerade betrachten). Die Frage ist nun, wie *beschreibt* man diese Seite so, dass ein Computer etwas damit anfangen kann? Diese Frage muss für jedes Programm, dass in irgendeiner Form Texte darstellen muss – also praktisch fast alle, von der Textverarbeitung über den Browser bis zum Statistikprogramm – beantwortet sein.

Sicherlich wird man den *Inhalt* der Seite beschreiben müssen, also beispielsweise wird man die Information aufschreiben müssen, dass der erste Absatz mit »Bevor wir uns . . . « beginnt. Ebenso wird man aufschreiben müssen, dass am oberen Rand der Seite Balken zu sehen sind und dass bestimmte Text größer sind als andere.

Es haben sich verschiedene Jargons gebildet, wie man solche Informationen aufschreibt. Treffend werden diese Jargons *Seitenbeschreibungssprachen* genannt, also »Sprechweisen«, wie man über den Inhalt und den Aufbau von Seiten redet.

Aus der ungeheuren Fülle der möglichen und auch real eingesetzten Sprachen (jede Textverarbeitung bringt mit jeder neuen Version eine neue Seitenbeschreibungssprache mit) wollen wir zum einen LaTeX betrachten, da es ein wichtiger und nützlicher Standard im naturwissenschaftlich-mathematischen Bereich ist, sowie HTML und XML, die Sprachen des World-Wide-Web.

6-1

Kapitel 6

LaTeX

Inhalt – Struktur – Form

Lernziele dieses Kapitels

1. Unterschied zwischen Form, Inhalt und Struktur kennen
2. Benutzung von LaTeX verstehen
3. Entscheiden können, wofür LaTeX geeignet ist

6-2

Inhalte dieses Kapitels

6.1 Inhalt – Struktur – Form 52
6.1.1 Drei Sichten auf einen Text 52
6.1.2 Beschreibungssprachen 53
6.1.3 Das Beispiel LaTeX 53

6.2 Gliederung von Dokumenten in LaTeX 53
6.2.1 Dokumentklassen 53
6.2.2 Die Präambel 54
6.2.3 Abschnitte und Paragraphen 54
6.2.4 Umgebungen 55

6.3 Typographisches und Graphiken in LaTeX 56
6.3.1 Schriftarten 56
6.3.2 Tabellen 56
6.3.3 Mathematik 57
6.3.4 Graphiken 57

6.4 Vorträge erstellen in LaTeX 58

Worum es heute geht

Wer TeX zum ersten Mal sieht, fragt sich oft besorgt, ob das wirklich ernst gemeint sei. Es fängt schon bei dem Namen an, dessen Aussprache unklar ist (er reimt sich auf »Pech«). Viel schlimmer ist aber, dass man reichlich kryptische Texte schreiben muss, um auch nur einen einfachen Brief zu produzieren. Bei LaTeX (was eine »Erweiterung« von TeX darstellt) liegen die Dinge kaum besser: Der Namen erscheint eher humoristisch und die kryptischen Texte sind immernoch kryptische Texte. Nach der ersten Begegnung kehren viele Nutzer reumütig zu Word und Co. zurück.

Wer TeX häufiger nutzt, gehört aber nicht zu den bekennenden Masochisten, sondern hat seine Stärken zu schätzen gelernt. Das Programm stürzt nie (!) ab, es produziert immer exakt die gleichen Ausgaben, unabhängig von Versionen oder Druckertreibern, die Qualität ist über jeden Zweifel erhaben, es ist frei verfügbar und auch Dokumente von vielen hundert Seiten sind kein Problem (ein Beispiel ist das Skript, das Sie gerade lesen). Der Formelsatz von TeX ist um Klassen besser als der praktisch aller anderen Programme: Eine von TeX gesetzte Formel ist ein optisches Juwel, Word verwurstet dieselbe Formel bestenfalls zu einem optischen Backstein.

Wohingegen der Kern von TeX seit mittlerweile Jahrzehnten fest und unveränderlich ist (was maßgeblich zu seiner Stabilität beigetragen hat), gibt es eine rege Szene von Nutzern, die immer neue Erweiterungen schreiben. Eine solche ist das Beamer-Paket, mit dem man TeX benutzen kann, um Präsentationen zu bauen – ein Einsatzzweck, für den TeX zwar nie vorgesehen war, bei dem es aber trotzdem recht erfreuliche Resultate vorweisen kann. Sollten Sie sich in die zu diesem Kapitel gehörende Vorlesung bequemen, so können Sie eine solche Präsentation selbst begutachten.

6.1 Inhalt – Struktur – Form

6.1.1 Drei Sichten auf einen Text

6-4

Drei grundsätzliche Dimensionen eines Dokuments.

Beispiel: Ein Beispieldokument

Für einen Kaiserschmarrn benötigt man:

1. 150g Mehl
2. 1/8l Milch
3. 3 Eier (getrennt)
4. Puderzucker und eine Prise Salz

Dieser (und auch jeder andere) Text hat

1. einen *Inhalt* – er gibt an, was der Text *bedeutet*,
2. eine *Struktur* – sie gibt an, wie der Text *aufgebaut* ist,
3. eine *Form* – sie gibt an, wie der Text *aussieht*.

6-5

Variation von Inhalt, Struktur und Form.

Beispiel: Variation des Inhalts

Für einen großen Kaiserschmarrn benötigt man:

1. 250g Mehl
2. 1/4l Milch
3. 6 Eier (getrennt)
4. Puderzucker und zwei Prisen Salz

Beispiel: Variation der Struktur

Für einen Kaiserschmarrn benötigt man 150g Mehl, 1/8l Milch, 3 Eier (getrennt), Puderzucker und eine Prise Salz.

Beispiel: Variation der Form

Für einen Kaiserschmarrn benötigt man:

1. *150g Mehl*
2. *1/8l Milch*
3. *3 Eier (getrennt)*
4. *Puderzucker und eine Prise Salz*

6-6

Ein gutes Dokumentenformat trennt Inhalt, Struktur und Form.

Ein »gutes« Dateiformat erlaubt es, *Inhalt, Struktur und Form* eines Textes *unabhängig voneinander zu notieren*.

Vorteile

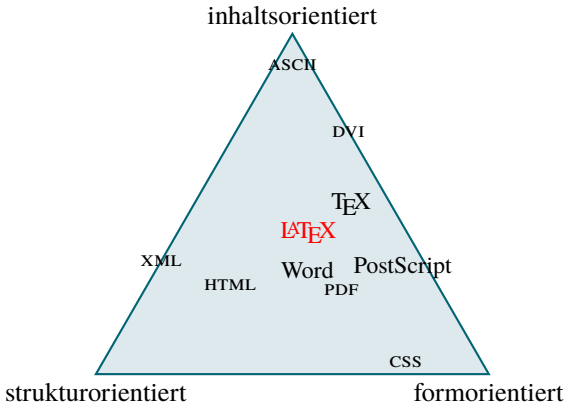
- Ist die *Struktur* separat notiert, kann ein Anzeigeprogramm leicht navigieren oder ein Inhaltsverzeichnis erzeugen.
- Ist die *Form* separat notiert, lässt sie sich leicht allgemein ändern: Wird *nur einmal* notiert, wie Bibliographieeinträge aussehen, so sind sie *automatisch einheitlich*.

Nachteile

- Das zusätzliche Notieren von Struktur und Form macht Arbeit.

6.1.2 Beschreibungssprachen

Wohin verschiedene Seitenbeschreibungssprachen tendieren. 6-7



6.1.3 Das Beispiel LaTeX

Was ist LaTeX? 6-8

LaTeX, geschrieben von Leslie Lamport, ist eine Erweiterung des Programms *TeX*, geschrieben von Donald Knuth. Ein *LaTeX-Dokument* ist ein *reines »Text«dokument (Folge von Unicode-Zeichen)*, das den *Inhalt* und die *Struktur* eines Textes enthält. Das LaTeX-Programm kümmert sich um *eine gute Form*.

Prinzipielles Vorgehen

Man erstellt ein *Manuskript*. Dies ist eine Unicode-Datei mit der Endung `.tex`. Dann benutzt man das Programm `lualatex` (eventuell sind mehrere Durchgänge nötig), um daraus eine `.pdf`-Datei zu erzeugen. Diese kann man dann drucken oder weiterreichen.

Vorführung des Übersetzungsprozesses an einem Beispiel Regie

6.2 Gliederung von Dokumenten in LaTeX

6.2.1 Dokumentklassen

Der Beginn eines LaTeX-Manuskripts. 6-9

Am Anfang steht eine Zeile, die LaTeX sagt, welche *Dokumentklasse* benutzt wird. Beispiele sind:

- `article` oder `scrartcl`,
- `report` oder `scrreprt`,
- `book` oder `scrbook`,
- `beamer`.

Die Zeile lautet dann beispielsweise wie folgt:

```
\documentclass[german,11pt]{article}
```

Die *Optionen in eckigen Klammern* sagen LaTeX, dass der Text auf Deutsch geschrieben ist und dass eine 11pt-Schrift die Standardgröße ist.
Die generelle Syntax für Befehle in LaTeXlautet:

```
\befehlsname[optionen]{argument1}{argument2}...
```

6.2.2 Die Präambel

6-10 **Die Präambel und der Körper.**

Der Kopfzeile folgt die *Präambel*. In ihr legt man *globale Einstellungen fest* und *lädt Erweiterungen*. Nach der Präambel folgt der *Körper* des Manuskript. Erst hier darf der eigentliche Text stehen.

```
\documentclass[german,11pt]{article}

% So schreibt man Kommentare in TeX

% Die Präambel:
\usepackage{babel}              % Sprachunterstützung
\usepackage[utf8]{luainputenc} % Manuskript ist in Unicode
\usepackage{graphicx}           % Zum Einbinden von Graphiken

\begin{document}
% Körper, der eigentliche Text
\end{document}
```

6-11 **Die wichtigsten Erweiterungen, die man so braucht.**

`luainputenc` Teilt TeX mit, wie der Text kodiert ist (Unicode oder ASCII). Sie sollten Ihre Texte immer in Unicode kodiren und müssen daher hier die Option `utf8` (Unicode) angeben.

`babel` Lädt umfangreiche Sprachunterstützung für alle möglichen Sprachen.

`graphicx` Lädt Befehle, mit denen sich extern erzeugte Bilder (JPEGS oder PDFS) gut einbinden lassen. Nachfolger von `graphics`.

`tikz` Lädt zusätzliche Befehle, mit denen sich Graphiken direkt in TeX-Notation beschreiben lassen.

6.2.3 Abschnitte und Paragraphen

6-12 **Wie man seinen Text strukturiert.**

Im Körper kommt nun der Text. In ihm finden sich besondere Befehle, die die *logische Struktur* des Textes angeben. Es ist die Aufgabe von TeX (und nicht die des Autors), diese logische Struktur optisch ansprechend umzusetzen.

```
\begin{document}

\title{Meine Bachelorarbeit}
\author{Ich \and mein Ego}
\maketitle
\tableofcontents

\section{Einleitung}
...
\subsection{Methoden}
...
\subsection{Ergebnisse}
...
\section{Zusammenfassung}
...
\end{document}
```

6-13 **Die wichtigsten Kommandos zur Strukturierung**

`title` Titel der Arbeit.

`author` Autor der Arbeit. Mehre Autoren trennt man mit dem speziellen Befehl `\and`.

`date` Spezielle Datumsangabe, wenn nicht das aktuelle gewünscht wird.

`maketitle` Erzwingt, dass der Titel dort gesetzt wird.

`section` Beginn eines Abschnitts

`subsection` Beginn eines Unterabschnitts. Man sollte Unterunterabschnitte nicht verwenden.

`chapter` Beginn eines Kapitels (nur bei der Dokumentklasse `book`).

Wie gibt man Textabsätze ein?

6-14

Zwischen die Strukturierungsbefehle schreibt man nun den eigentlichen Text. Er besteht aus *Absätzen*, die durch *Leerzeilen* voneinander getrennt werden. Innerhalb eines Absatzes haben Leerzeichen und Zeilenumbrüche alle denselben Effekt: Sie trennen Wörter.

```
\section{Einleitung}

Dieser Text ist Teil des ersten Absatzes. Der Umbruch
hier hat keinen Effekt, das Wort "`hier"' steht im fertigen
Dokument wahrscheinlich auf derselben Zeile wie "`Umbruch"'.

Hier ist erst der zweite Absatz.              Die vielen
Leerzeichen           in        diesen Zeilen haben
denselben Effekt, als wenn man immer nur eines eintippt.
```

Besonderheiten und Probleme.

6-15

- Als Anfänger versucht man häufig, Zeilenumbrüche und künstliche Abstände zu erzwingen. Dies ist schwierig und man sollte es bleiben lassen.
- In alten Systemen musste man Umlaute merkwürdig eingeben. Dies ist heute nicht mehr nötig und sollte vermieden werden.
- Anführungszeichen im Deutschen sehen „so“ aus. In TeX schreibt man dies aber so: `"`so"'`.
- Einen Bindestrich ist kurz wie in dem Wort Binde-Strich.
- Ein Gedankenstrich ist – lang. Man schreibt ihn in TeX mit zwei einfachen Strichen, also `ist -- lang`.

6.2.4 Umgebungen

Umgebungen dienen ebenfalls der Strukturierung.

6-16

Häufig möchte man in den Text nun *Aufzählungen* und *nummerierte Listen* einfügen. Dazu fasst man die Liste in eine *Umgebung*. Innerhalb der Umgebung benutzt man den `\item`-Befehl, um neue Punkte zu beginnen.

Im Manuskript

```
Hier kommt eine nummerierte
Liste:
\begin{enumerate}
\item erster Punkt
\item zweiter Punkt
\end{enumerate}

Hier noch eine
unnummerierte Liste:
\begin{itemize}
\item ein Punkt
\item ein anderer Punkt
\end{itemize}
```

Im Ergebnis

Hier kommt eine nummerierte Liste:

1. erster Punkt
2. zweiter Punkt

Hier noch eine unnummerierte Liste:

- ein Punkt
- ein anderer Punkt

6-17

Umgebungen müssen nicht Listen enthalten.

Es gibt viele Umgebungen, die keine Listen darstellen. Beispielsweise schreibt man den Abstract in einer `{abstract}`-Umgebung.

```
...
\maketitle
\begin{abstract}
  In dieser Arbeit zeige ich auf, wie man die Welt rettet und
  dabei reich wird.
\end{abstract}
```

6.3 Typographisches und Graphiken in LaTeX

6.3.1 Schriftarten

6-18

Änderung der Schriftart in Text.

Es gibt verschiedene Befehle, die die Schriftart ändern. Der `\emph`-Befehl hebt sein Argument hervor, indem er es schräg stellt (es sei denn, der Text ist schon schräg, dann macht er den Text gerade). Der `\textbf`-Befehl macht sein Argument fett. Der `\textsf`-Befehl benutzt eine Sans-Serif-Schrift für sein Argument. Mit den Befehlen `\small`, `\normalsize` und `\large` kann man ab einem Punkt die Schriftgröße ändern.

Im Manuskript

```
Es ist \emph{ausgesprochen
wichtig}, dass dies
verstanden wird. Es ist aber
\textbf{typographisch
schlecht}, Wörter im normalen
Text fett zu setzen.
```

Im Ergebnis

Es ist *ausgesprochen wichtig*, dass dies verstanden wird. Es ist aber **typographisch schlecht**, Wörter im normalen Text fett zu setzen.

6-19

Man kann auch die Schrift generell ändern.

Es gibt Erweiterungspakete wie `times`, mit denen man die Schriftart des gesamten Textes ändern kann. Es ist allerdings schwierig, im Text mehrere unterschiedliche Schriften zu mischen und das ist auch gut so.

6.3.2 Tabellen

6-20

Tabellen haben eine eigene Syntax.

Für Tabellen benutzt man die `{tabular}`-Umgebung, die als Argument eine *Formatierungsvorschrift* bekommt. Jede *Zeile* wird durch `\\` beendet. Jede *Zelle* wird durch `&` beendet.

Im Manuskript

```
\begin{tabular}{rcc}
  \emph{Spezies} & \emph{Anzahl} & \emph{cm} \\\hline
  Hund           & 100           & 7 \\
  Katze          & 50            & 37 \\
  Maus           & 6             & 47
\end{tabular}
```

Im Ergebnis

Spezies	*Anzahl*	*cm*
Hund	100	7
Katze	50	37
Maus	6	47

6.3.3 Mathematik

Die Hauptstärke von TeX: Mathematik. 6-21

TeX setzt Formeln viel besser, als es die meisten Menschen können.
Mathematischer Text wird in `$`-Zeichen eingeschlossen und es gibt eine eigene Syntax, wie man ihn aufschreibt:

- Tiefgestelltes wird durch _ eingeleitet.
- Hochgestelltes wird durch ^ eingeleitet.
- Sonderzeichen werden durch spezielle Befehle erzeugt.

Beispiele von mathematischem Text. 6-22

Im Manuskript

```
1. $a^2 + b^2 = c^2$
2. $\alpha + \beta = \gamma^2$
3. $\sum_{i=1}^n i = n(n+1)/2$
4. $\sum_{i=1}^\infty \frac{1}{i^2} = \frac{\pi^2}{6}$
5. $\lim_{n\to\infty} 1/n^2 = 0$
6. $$\int_{-\infty}^\infty e^{-x^2} dx < \infty$$
```

Im Ergebnis

1. $a^2 + b^2 = c^2$
2. $\alpha + \beta = \gamma^2$
3. $\sum_{i=1}^{n} i = n(n+1)/2$
4. $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$
5. $\lim_{n\to\infty} 1/n^2 = 0$
6. $$\int_{-\infty}^{\infty} e^{-x^2} dx < \infty$$

6.3.4 Graphiken

Externe Graphiken lassen sich leicht einbinden. 6-23

Der Befehl `\includegraphics` erlaubt es, `.pdf`-Graphiken und `.jpg`-Graphiken einzubinden. Als Optionen gibt man die gewünschte Größe an. Als Parameter gibt man den Dateinamen an.

```
Folgende Graphik zeigt den Verlauf der Kurve:

\includegraphics[width=6cm]{graphik1.pdf}

Im Folgenden...
```

Beispiel einer Graphik in einer `figure`-Umgebung 6-24

Es bietet sich an, Graphiken in `{figure}`-Umgebungen einzuschließen. Dann kann man ihnen eine Überschrift geben und es sieht hübscher aus.

```
...
und somit wird alles gut.

\begin{figure}
  \includegraphics[width=6cm]{graphik1.pdf}
  \caption{Verlauf der Kurve.}
  \label{graphik1}
\end{figure}

Wie wir in Abbildung \ref{graphik1} sehen, ist es nicht so,
dass...
```

6.4 Vorträge erstellen in LaTeX

6-25

Auch Vorträge lassen sich mit LaTeX erstellen.

Vorträge lassen sich mit Hilfe von Erweiterungen von LaTeX erstellen, bei Benutzung von pdfLaTeX empfiehlt sich `beamer`. Es gibt zwei wesentliche Änderungen gegenüber normalen Dokumenten:

1. Als Dokumentklasse muss man `beamer` angeben.
2. Jede »Folie« kommt in eine `{frame}`-Umgebung. Diese nimmt als Parameter eine Folienüberschrift.

Das Aussehen des Vortrags kann man durch Änderung des »themes« leicht ändern.

6-26

Beispiel eines Vortragsmanuskripts.

```
\documentclass[german]{beamer}

\usepackage{times}
\usepackage[utf8]{inputenc}
\usepackage{babel}

\usetheme{Luebeck}

\title{Meine Bachelorarbeit in 10 Minuten}
\author{Ich}

\begin{document}
\begin{frame}
  \maketitle
\end{frame}

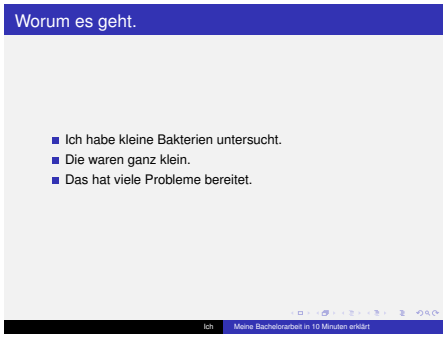
\begin{frame}{Gliederung}
  \tableofcontents
\end{frame}

\section{Einleitung}

\begin{frame}{Worum es geht.}
  \begin{itemize}
  \item Ich habe Bakterien untersucht.
  \item Die waren ganz klein.
  \item Das hat viele Probleme bereitet.
  \end{itemize}
\end{frame}
\end{document}
```

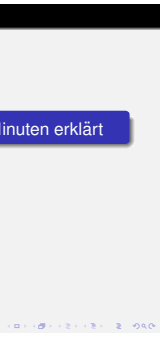
6-27

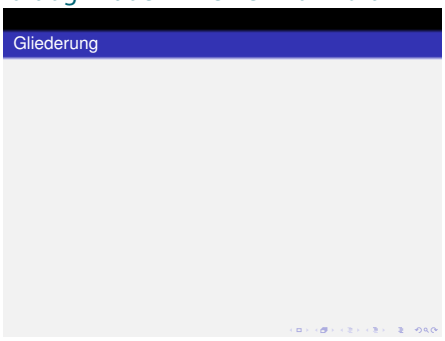
Die drei Seiten des Vortrags

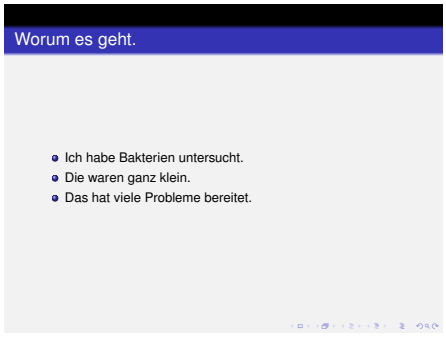


Der Vortrag mit dem Theme »Frankfurt«

6-28







Zusammenfassung dieses Kapitels

6-29

Inhalt – Struktur – Form

Dokumente haben einen *Inhalt*, eine *Struktur* und eine *Form*.

LaTeX

LaTeX ist ein vollkommen stabiles System, das Manuskripte in druckfertige Dokumente und Präsentationen verwandelt.

Vorteile

+ Unerreichte Qualität beim Setzen mathematischer Formeln.
+ Vollkommen stabiles System, das auch riesige Dokumente verarbeiten kann.
+ Die Trennung von Inhalt, Struktur und Form erlaubt es, aus einer Quelle unterschiedliche Dokumente zu erzeugen (wie das Skript und die Präsentationen dieser Vorlesung).
+ Flache Lernkurve.

Nachteile

– Man muss eine gewöhnungsbedürftige Syntax lernen.
– Man »sieht nicht sofort«, wie sich eine Änderung auswirkt.
– Die Erstellung eigener Layouts ist schwierig.
– Sehr lange Lernkurve.

Zum Weiterlesen

[1] H. Kopka. *LaTeX – Einführung,* Addison-Wesley, 1996.

7-1

Kapitel 7
Hypertext Markup Language
Die Sprache des WWW

7-2

Lernziele dieses Kapitels

1. Die Grundsyntax von HTML kennen
2. Die Grundsyntax von XML kennen
3. Eigene HTML-Seiten erstellen können

Inhalte dieses Kapitels

7.1	HTML	61
7.1.1	Einführung	61
7.1.2	Aufbau von HTML-Seiten	61
7.1.3	Aufbau von Tags	63
7.2	XML	63
7.2.1	Einführung	63
7.2.2	Vergleich mit HTML	64
7.2.3	Wohlgeformte Texte	64
	Übungen zu diesem Kapitel	65

Worum es heute geht

Copyright by Uldis Bojārs, Creative Commons Attribution Sharealike

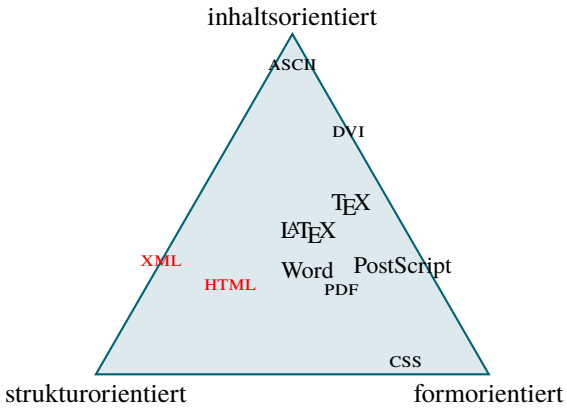
Gemeinfrei

Die »Erfindung« des World-Wide-Web durch Tim Berners Lee (Foto) im Jahre 1989 ist von ihrer Wichtigkeit meiner Meinung nach mit der Erfindung des Buchdrucks durch Johannes Gensfleisch (besser bekannt als *Gutenberg*) gut 500 Jahre vorher vergleichbar. Das Netz hat das alltägliche Leben nachhaltig verändert (allerdings nur in den Informationsgesellschaften) und ein Ende dieser Entwicklung ist derzeit nicht in Sicht.

Die technischen Details des www sollen uns weniger interessieren; im aktuellen Kapitel soll es um *die Sprachen des Netzes* gehen, also die Seitenbeschreibungssprachen, mit denen der Inhalt von Webseiten beschrieben wird. Diese Sprachen heißen HTML und XML.

Der enorme Erfolg des www ist nicht zuletzt darauf zurückzuführen, dass HTML sowohl für Menschen wie für Computer gleichermaßen »gut lesbar« ist. Natürlich hat ein HTML-Quelltext nicht die gleiche lyrische Qualität wie, sagen wir, der »Faust«, jedoch ist es mit recht wenig Übung möglich, eigene Texte in dieser Sprache zu verfassen. Dies bedeutet nicht, dass man jeden HTML-Quelltext immer einfach so lesen kann wie, sagen wir, die flott geschriebene Kolumne der Tageszeitung Ihres Vertrauens. Genau wie es unverständliche Texte gibt (Vorlesungsskripte sind leider häufige Beispiele), gibt es auch völlig unverständlichen HTML-Code (hier ist von Computern generierter Code ein Beispiel – Computern fehlt einfach das literarische Gespür für guten HTML-Code).

Wiederholung: Wohin Seitenbeschreibungssprachen tendieren. 7-4



7.1 HTML

7.1.1 Einführung

Die Hypertext-Markup-Language. 7-5

Die Hypertext-Markup-Language (HTML) ist eine *Seitenbeschreibungssprache,* in der die meisten Webseiten verfasst sind. Sie ist *keine Programmiersprache*, HTML beschreibt primär *Inhalt und Struktur* beliebiger Textdokumente, die nicht unbedingt im Netz stehen müssen. Die *Form* kann kaum festgelegt werden. Wie der Name schon sagt, erlaubt es HTML, *Verweise* auf Bilder oder *Hyperlinks* einfach zu notieren.

Vom HTML-Sourcecode zur Anzeige. 7-6

Originalseite in HTML

```
<p>
Dies ist
ein <i>Text</i>.
</p>
```

Anzeige des Browser

Dies ist ein *Text*.

Vorführung der Originalseite und ihrer Anzeige in einem Browser Regie

7.1.2 Aufbau von HTML-Seiten

Die Grundelemente von HTML-Seiten. 7-7

HTML-Seiten bestehen aus *Tags* und *normalem Text*. Tags sind Ausdrücke der Form `<tag>` und `</tag>`. Die erste Form von Tags heißt *öffnendes Tag*, die zweite Form heißt *schließendes Tag*; sie funktionieren wie öffnende und schließende Klammern. Alles, was von einem öffnenden und dem zugehörigen schließenden Tag umschlossen wird, nennt man ein *Element*. Je nach Art des Elements (also je nach Art des Tags) dürfen innerhalb des Elements wieder Tags und normaler Text stehen.

✎ Zur Übung 7-8

Wie viele Tags und wie viele Elemente enthält die folgende Seite?

```
<p>
  Dies ist
  ein <i>Text</i>.<br></br>
  <ul>
    <li> Foo </li>
    <li> Bar </li>
  </ul>
</p>
```

7-9

Der Grundaufbau einer HTML-Seite.

Das äußerste Tag einer HTML-Seite ist immer das Paar <html> und </html>. Innerhalb dieses äußersten Tags kommt erst das Paar <head> und </head>, dann das Paar <body> und </body>. Innerhalb von <head> kann man Titelinformation angeben mittels weiterer Tags wie <title>. Innerhalb von <body> kommt der eigentliche Text der Seite, der später angezeigt werden soll.

7-10

Gliederung einer HTML-Seite.

- Zur Gliederung des Haupttextes stehen weitere Tags zur Verfügung.
- Überschriften kann man mit den Tags <h1> bis <h6> einfügen.
- Absätze umschließt man mit den Tags und .
- Listen umschließt man mit <ul> für unsortierte Listen oder <ol> für sortierte Listen.
- Einzelne Listenpunkte umschließt man mit <li> (list item).
- Tabellen umschließt man mit <table>, Tabellenzeilen mit <tr> (table row) und Zellen mit <td> (table data).
- erzeugt »generische« Unterteilungen, sie werden nur in Verbindung mit so genannten Style-Sheets gebraucht.

7-11

Blockelemente versus Inline-Elemente.

Die bisherigen Gliederungselemente konnten nicht mitten in einem Absatz vorkommen. Man nennt sie *Blockelemente*. Es gibt aber auch Tags und Elemente, die mitten in einem Absatz vorkommen dürfen wie

- <i> (italics) setzt Text kursiv,
- <b> (bold) setzt Text fett,
- hebt Text besonders hervor,
- macht selber nichts, wird aber für Style-Sheets gebraucht.

Solche Elemente nennt man *Inlineelemente*.

Merkregel

Inlineelemente können mitten in einer Zeile beginnen und auf einer anderen Zeile enden. Blockelemente hingegen nicht.

7-12

Fehlersuche.

```
<html>
  <head>
    <title>
      Dies ist eine Beispielseite.
    </title>
  </head>
  <body>
    <h1> Haupt&uuml;berschrit </h1>
    <p>
      Hier ist <strong>normaler</strong> Text. Man
      beachte, dass f&uuml;r den Umlaut &uuml; eine
      merkw&uuml;rdige Zeichenfolge benutzt wurde.
    </p>
    <h2> Unterabschnitt </h1>
    <p>
       Blah, blah.
  </body>
</html>
```

✎ Zur Übung

Finden Sie zwei Fehler.

7.1.3 Aufbau von Tags

Tags mit Parametern. 7-13

Tags können *Parameter* bekommen. Diese werden *innerhalb des öffnenden Tags* in der Form `parameter="wert"` angegeben, mehrere Parameter werden durch Leerzeichen getrennt.

Beispiel: Ein Verweis auf eine Webseite

```
Auf <a href="http://wikipedia.org">dieser Seite</a> ist
ein Wiki zu finden.
```

✎ **Zur Übung** 7-14

Geben Sie den Originalquelltext einer HTML-Seite an, die in einer Liste Links auf die deutsche, die englische und die französische Wikipedia enthält.

Tags ohne Inhalt. 7-15

Oft gibt es Situationen, in denen ein Tag keinen »Inhalt« hat. So erzeugt das Tag-Paar `<br>` und `</br>` einen Zeilenumbruch (`br` steht für break). Innerhalb dieses Paares kann man aber nicht sinnvoll etwas schreiben. Statt `<br></br>` darf man kürzer schreiben `<br/>`. Allgemein bedeutet `<tag/>` dasselbe wie `<tag></tag>`.

Beispiel: Eine Stelle auf die man verweisen kann

```
<a name="anfang_des_textes"/> Im Anfang war das Wort...
```

Welche Tags gibt es? 7-16

Die Liste der Tags hängt von der HTML-Version ab. In der aktuellen Version HTML5 gibt es eine feste Liste, mit der man Dinge beschreiben kann wie: Überschriften, Angabe von Textarten wie Zitate, Programmtexte, Listen, Tabellen, Verweise und eingebettete Objekte wie Graphiken. Eine ausführliche Liste aller Tags mit Erklärungen und Tutorien findet sich auf der Webseite `http://de.selfhtml.org/`. Es ist *nicht möglich,* neue Tags hinzuzufügen für Dinge wie Atome, Gene, Moleküle, Spezies.

7.2 XML

7.2.1 Einführung

Die eXtensible Markup Language. 7-17

Die Beschreibung der Struktur und des Inhalts von Dokumenten mit HTML ist, wenn sie richtig angewendet wird, *sowohl für Menschen wie für Maschinen gut lesbar*. Ein großes Problem ist jedoch, dass *sich* HTML *nicht erweitern lässt*. Die *eXtensible Markup Language (*XML*)* löst dieses Problem, indem sie es erlaubt, *neue Tags zu definieren*. Damit ist aber noch nicht gesagt, wie diese Tags dann *angezeigt werden sollen*, was ein Style-Sheet regelt. Man kann XML aber auch einfach nur benutzen, um die Struktur von Dokumenten deutlich zu machen. Ein XML-Dokument muss immer mit folgender Zeile beginnen:

```
<?xml version="1.0"?>
```

7.2.2 Vergleich mit HTML

7-18

Ein Rezept in HTML und in XML.

In HTML

```
Ein Kaiserschmarrn besteht aus:
<ol>
  <li> 150g Mehl </li>
  <li> 1/8l Milch </li>
  <li> 3 Eier (getrennt) </li>
  <li> Puderzucker und eine Priese Salz </li>
</ol>
```

In XML

```
Ein Kaiserschmarrn besteht aus:
<rezept>
  <zutat> 150g Mehl </zutat>
  <zutat> 1/8l Milch </zutat>
  <zutat> 3 Eier (getrennt) </zutat>
  <zutat> Puderzucker und eine Priese Salz </zutat>
</rezept>
```

7-19

✎ Zur Übung

Geben Sie die Beschreibung eines Atoms in XML-Syntax an. (Es gibt nicht die »richtige« Lösung, es kommt darauf an, was Ihnen wichtig erscheint und auf den Kontext.)

7.2.3 Wohlgeformte Texte

7-20

Wohlgeformte und gültige XML-Texte

▶ Definition

Ein XML-Text heißt *wohlgeformt*, wenn alle Tags korrekt geschachtelt sind.

▶ Definition

Eine *Document Type Description (DTD)* gibt eine Menge von erlaubten Tags und erlaubten Parametern an. (Beispiel: Es gibt eine DTD für HTML.)

▶ Definition

Ein XML-Text heißt *gültig in Bezug auf eine DTD*, wenn die Tags wie von der DTD vorgeschrieben benutzt werden.

Zusammenfassung dieses Kapitels

7-21

1. HTML-Texte beschreiben Inhalt und Struktur, XML-Texte ebenso.
2. HTML- und XML-Texte bestehen aus *korrekt geschachtelten Tags*.

Zum Weiterlesen

[1] Götz Bürkle et al. http://de.selfhtml.org/, Zugriff Mai 2007.

Übungen zu diesem Kapitel

Übung 7.1 XML-Syntax benutzen, mittel

Die *Protein Database (PDB)* stellt die Raumkoordinaten der Atome von einer sehr großen Zahl von Atomen zur Verfügung. Hier ist ein typisches Beispiel:

```
HEADER    HYDROLASE                               25-JUL-03   1UJ1
TITLE     CRYSTAL STRUCTURE OF SARS CORONAVIRUS MAIN PROTEINASE
TITLE    2 (3CLPRO)
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: 3C-LIKE PROTEINASE;
COMPND   3 CHAIN: A, B;
COMPND   4 SYNONYM: MAIN PROTEINASE, 3CLPRO;
COMPND   5 EC: 3.4.24.-;
COMPND   6 ENGINEERED: YES
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: SARS CORONAVIRUS;
SOURCE   3 ORGANISM_COMMON: VIRUSES;
SOURCE   4 STRAIN: SARS;
...
REVDAT   1   18-NOV-03 1UJ1    0
JRNL        AUTH   H.YANG,M.YANG,Y.DING,Y.LIU,Z.LOU,Z.ZHOU,L.SUN,L.MO,
JRNL        AUTH 2 S.YE,H.PANG,G.F.GAO,K.ANAND,M.BARTLAM,R.HILGENFELD,
JRNL        AUTH 3 Z.RAO
JRNL        TITL   THE CRYSTAL STRUCTURES OF SEVERE ACUTE RESPIRATORY
JRNL        TITL 2 SYNDROME VIRUS MAIN PROTEASE AND ITS COMPLEX WITH
JRNL        TITL 3 AN INHIBITOR
JRNL        REF    PROC.NAT.ACAD.SCI.USA         V. 100 13190 2003
JRNL        REFN   ASTM PNASA6  US ISSN 0027-8424
....
ATOM      1  N   PHE A   3      63.478 -27.806  23.971  1.00 44.82           N
ATOM      2  CA  PHE A   3      64.607 -26.997  24.516  1.00 42.13           C
ATOM      3  C   PHE A   3      64.674 -25.701  23.723  1.00 41.61           C
ATOM      4  O   PHE A   3      65.331 -25.633  22.673  1.00 40.73           O
ATOM      5  CB  PHE A   3      65.912 -27.763  24.358  1.00 44.33           C
ATOM      6  CG  PHE A   3      67.065 -27.162  25.108  1.00 44.20           C
ATOM      7  CD1 PHE A   3      67.083 -27.172  26.496  1.00 43.35           C
ATOM      8  CD2 PHE A   3      68.135 -26.595  24.422  1.00 43.49           C
...
```

Das Format der PDB ist sowohl für Computer wie für Menschen etwas »schwer lesbar«. Wie würde die Beschreibung in XML aussehen? Teilen Sie diese Aufgabe gegebenenfalls auf mehrere Teilgruppen auf: Eine Gruppe kümmert sich um die XML-Beschreibung der `JRNL`-Einträge, eine andere um die `ATOM`-Einträge und so weiter.

Übung 7.2 HTML-Syntax üben, leicht

Erstellen Sie eine HTML-Seite, die eine Liste Ihrer persönlichen Bookmarks enthält.

Prüfungsaufgaben zu diesem Kapitel

Übung 7.3 Modellierung von Daten als XML, leicht, original Klausuraufgabe, mit Lösung

Ändert sich das Klima oder nicht? Zur endgültigen Beantwortung dieser Frage sind Sie mit der Erstellung einer Datenbank beauftragt worden, die die Entwicklung der monatlichen Temperaturen im Laufe der Jahre festalten soll. Dabei sollen für jeden Monat jeweils Tiefst-, Höchst-, und Durchschnittstemperaturen gespeichert werden. Hier sehen Sie einige Beispieldaten der Wetterstation in Lübeck-Blankensee:

Jahr	Monat	Minimum	Mittel	Maximum
2004	Juli	7.2	16.8	28.3
2009	August	8.6	18.5	33.6

Stellen Sie diese Daten im XML-Format dar! Dabei soll die *logische Struktur* der Daten wiedergegeben werden und nicht etwa die visuelle Formatierung der Tabelle. Führen Sie deshalb geeignet benannte Tags und/oder Attribute ein, z.B. kann ein Jahr durch ein Tag `<jahr>` modelliert werden.

Übung 7.4 Modellierung von Daten als XML, leicht, original Klausuraufgabe, mit Lösung

Es soll eine Gendatenbank aufgebaut werden, in der Gene als XML-Dateien gespeichert werden. Zu jedem Gen sollen gespeichert werden:

- Der Name des Gens,
- die Nummer des Chromosoms,
- die Startposition auf dem Chromosom und
- die Endposition auf dem Chromosom.

Erstellen Sie eine XML-Datei, die die Daten der folgenden zwei Gene enthält:

1. malT, Start 8986, Ende 11751, Chromosom 2,
2. pspG, Start 395473, Ende 395685, Chromosom 1.

Modellieren Sie die genannten Eigenschaften der Gene durch geeignete XML-Tags und/oder Attribute.

Übung 7.5 XML und Shell-Programmierung, leicht bis mittel, original Klausuraufgabe, mit Lösung

In einer Datei liegt eine XML-Beschreibung eines Moleküls vor. Leider kannte sich der Verfasser dieser Datei offensichtlich nicht besonders gut mit XML aus. Ihm sind daher zwei Syntaxfehler unterlaufen:

```
1 <?xml version="1.0"?>
2 <molecule name="hydrogen fluoride">
3         <atom protons1>hydrogen</atom>
4         <atom protons="6">fluorine</atom>
5 </mol>
```

Geben Sie an, welches diese Syntaxehler sind, und wie man sie sinnvoll (also z.B. nicht durch Löschen von Zeilen) korrigieren könnte.

Zum Finden solcher und ähnlicher Fehler in XML-Dateien kann man das Shell-Werkzeug `xmllint` benutzen. Dieses analysiert eine als Argument gegebene XML-Datei und gibt gefundene Fehler zeilenweise aus. Wird kein Fehler gefunden, ist die Ausgabe leer. Sei zum Beispiel `test.xml` eine (fehlerhafte) XML-Datei mit folgendem Inhalt:

```
<?xml version="1.0" >
<molecule>
```

Dann sieht die Ausgabe von `xmllint test.xml` wie folgt aus:

```
test.xml:1: error: parsing XML declaration: '?>' expected
test.xml:2: error: Premature end of data in tag molecule line 2
```

Verwenden Sie die Befehle `xmllint` und `wc`, um sich direkt die Anzahl der Fehler in der Datei `molekuel.xml` anzeigen zu lassen. Erzeugen Sie dabei keine zusätzlichen Dateien!

Schreiben Sie ein Shell-Skript `fehleranalyse.bash`, das die Anzahl der Fehler in einer als Parameter übergebenen XML-Datei ausgibt. Geben Sie dabei nicht nur die Zahl selbst, sondern auch eine sinnvolle Meldung aus. Geben Sie nur den Inhalt des Skripts an.

Sie erfahren in einer weiterführenden Vorlesung, dass es wie in Java auch in Shell-Skripten das Konzept der **`for`**-Schleife gibt, allerdings mit einer etwas anderen Syntax. Da Sie den Ausführungen des Dozenten nicht gut folgen konnten, informieren Sie sich stattdessen bei Wikipedia. Sie stoßen dabei auf folgendes Beispielskript, das für jede im aktuellen Ordner vorhandene Datei mit der Endung `.xml` die Anzahl der Zeilen ausgibt:

```
for i in *.xml ; do
  wc -l $i
done
```

Passen Sie dieses Beispielskript so an, dass für jede Datei mit der Endung `.xml` im aktuellen Ordner das Shell-Skript aus der vorigen Aufgabe aufgerufen wird, statt die Anzahl der Zeilen auszugeben.

Teil III

Programmieren in Java

Wenn man die Vorlesungskarte am Anfang dieses Skripts betrachtet, so erscheint die Informatik als ein großes, komplexes Gebiet, das durch Querbezüge mit vielen anderen Disziplinen verwoben ist. Manche Teile der Informatik erscheinen dabei (je nach Betrachterstandpunkt) wichtiger als andere. Gibt es aber etwas, ohne das die Informatik aufhört, Informatik zu sein? Zwei spontane Ideen könnten die Begriffe »Computer« und »Information« sein. Könnte man Informatik betreiben, wenn es keine Rechner gäbe? Die Antwort ist erstaunlicherweise »Yes, we can!«: Die zentrale Frage »Gibt es etwas, das Computer prinzipiell nicht können?« wurde von Alan Turing bereits beantwortet, als es noch gar keine Computer gab. Der Begriff der »Information« ist ebenso ungeeignet, um die Informatik (trotz des Namens) an ihm aufzuhängen: Die Informationstheorie ist lange vor der Informatik entwickelt worden, nämlich zu der Zeit, als Draht- und Funkübertragung von Daten möglich wurde.

Wo liegt also der »Kern« der Informatik? Es ist der *Algorithmusbegriff*. Er macht das spezifische, das Wesen der Informatik aus. Ein Algorithmus ist eine manchmal abstrakte, manchmal ganz konkrete Vorschrift, wie man ein Problem löst. Er besteht aus einzelnen Schritten, die nacheinander abgearbeitet werden, er kann aber auch so genannte *Schleifen* und *bedingte Anweisungen* enthalten, wodurch ein beliebig komplexes Gebilde entstehen kann. Ein bekanntes Beispiel eines Algorithmus ist der Gauß-Algorithmus zur Lösung eines Linearen Gleichungssystems: Für jede Zeilen (= eine Schleife) überprüfe, ob (= eine bedingte Anweisung) die Zahl in Spalte…

Ähnlich wie bei der Beschreibung von Seiten stellt sich bei Algorithmen das Problem, wie man sie »aufschreibt«. Die verschiedenen Jargons, wie man Algorithmen aufschreibt, nennt man nicht, was passend wäre, »Algorithmenbeschreibungssprachen«, sondern *Programmiersprachen*. Wir werden uns im Rahmen dieser Veranstaltung die Programmiersprache Java genauer anschauen.

Die spezielle Wahl der Sprache Java für diese Veranstaltung war ein wenig willkürlich und hat ihre Vor- und Nachteile. Ein Vorteil ist, dass sie weit verbreitet ist und »auch wirklich real genutzt wird«; nachteilig ist, dass sie nicht gerade nach didaktischen Kriterien entwickelt wurde (im Gegensatz zu beispielsweise Pascal). Auf jeden Fall wird es wichtiger sein, die Grundideen zu verstehen, als jede Besonderheit von Java perfekt zu beherrschen.

8-1

Kapitel 8
Der Algorithmusbegriff
Von der vagen Idee zur Instruktionsfolge

8-2

Lernziele dieses Kapitels

1. Den Begriff des Algorithmus verstehen
2. Einfache Lösungsideen als Algorithmen formulieren können
3. Probleme spezifizieren können
4. Programmiersprachen als Kommunikationsmittel für Algorithmen begreifen
5. Das Konzept des Übersetzers verstehen

Inhalte dieses Kapitels

8.1 Algorithmen 69
8.1.1 Geschichte 69
8.1.2 Definition 69
8.1.3 Steuerungsanweisungen 70

8.2 Spezifikationen 71

8.3 Programmiersprachen 71
8.3.1 Wozu dienen Programmiersprachen? . . 71
8.3.2 Übersetzer 73

Worum es heute geht

Das Wort »Algorithmus« ist schon ein wenig unheimlich, finden Sie nicht? Für Menschen mit einer Mathematikphobie klingt es irgendwie nach Logarithmus – und mit diesem standen sie schon immer auf Kriegsfuß. Für Technikverliebte klingt es irgendwie griechisch und damit altmodisch und uninteressant. Hypochonder hoffen inständig, nie einen Algorithmus zu bekommen, da die Apothekenrundschau noch nie über ein Heilmittel berichtet hat. Selbst einige Informatikstudenten weigern sich, Algorithmen als eigenständigen Wesen die nötige Aufmerksamkeit zu schenken – für sie gibt es Programmcode und alles andere ist von Übel.

Dabei benutzen wir alle Algorithmen Tag ein, Tag aus. Der Grund ist, dass jedem einigermaßen strukturierten Arbeitsablauf ein Algorithmus zu Grunde liegt. Wenn Sie zwei Zahlen schriftlich multiplizieren (was man zugegebenermaßen nicht jeden Tag macht), so gehen Sie ja nicht völlig planlos zu Werke, sondern Sie haben in der Schule mehr oder minder mühselig gelernt, was alles wie in welcher Reihenfolge aufgeschrieben werden muss. Das, was Sie gelernt haben, ist ein Algorithmus, nämlich der »Algorithmus für die schriftliche Multiplikation nach der Schulmethode«. Ein anderes Beispiel ist das Suchen eines Namens in einem Telefonbuch, wo man ja nicht Zeile für Zeile nach dem gesuchten Namen durchgeht, sondern zwischen den Seiten »hin- und herspringt« (wieder zugegebenermaßen kein alltägliches Geschäft heutzutage; was aber hauptsächlich daran liegt, dass wir das Ausführen des Im-Telefonbuch-suchen-Algorithmus heutzutage unserem Telefon überlassen).

Was könnte man tun, um dem Wort »Algorithmus« ein besseres Image zu verpassen? Am einfachsten wäre es wohl, einfach ein anderes, freundlicheres Wort zu benutzen. Dies hat sich in der Geschichte immer wieder bewährt: bei der Bundeswehr ist aus einem schrecklichen »Krieg« der an eine gemütliche, holzkohleofengewärmte Amtsstube erinnernde »Verteidigungsfall« geworden; die mittelalterliche, voraufklärerische »Folter« ist den zackigen »verschärften Verhörmethoden« gewichen; und »Raider« heißt jetzt »Twix« (warum auch immer). Wie wäre es mit »Berechnungsvorschrift«? Zu dirigistisch. Vielleicht »Rechenanleitung«? Schon besser, klingt aber etwas nach einer Mathematiknachhilfestunde. Letzter Vorschlag: »Kochrezepte für Computer«. Da stellt man sich unwillkürlich tomatensoßebespritzte Roboter in einer Küche vor und ist damit schonmal in einer positiven mentalen Grundhaltung.

Am Ende bleiben wir dann doch bei »Algorithmus«.

8.1 Algorithmen

8.1.1 Die Geschichte des Begriffs

Über das Wort »Algorithmus«.

8-4

Unkown author, public domain

- Das Wort stammt vermutlich vom Namen des Gelehrten *Muhammad ibn Musa, Abu Dscha'far al-Chwarizmi* (∗ ca. 783, † ca. 850)
- Entscheidend war sein Buch *Al-kitab al-muchtasar fi hisab al-dschabr wa-l-muqabala* (»Rechnen durch Ergänzung und Ausgleich«).

Das erste Programm stammt von einer Frau.

8-5

Unkown author, public domain

Aus *de.wikipedia.org/wiki/Algorithmus*

> Der erste für einen Computer gedachte Algorithmus wurde 1842 von Ada Lovelace, in ihren Notizen zu Charles Babbages Analytical Engine, festgehalten. Sie gilt deshalb als die erste Programmiererin. Weil Charles Babbage seine Analytical Engine nicht vollenden konnte, wurde Ada Lovelaces Algorithmus nie darauf implementiert.

Duplieren nach Adam Riese (1574).

8-6

Unknown author, public domain

Lehret wie du ein zahl zweyfaltigen ſolt.
Thu ihm alſo: Schreib die zahl vor dich
mach ein Linien darunter
heb an zu forderſt
Duplir die erſte Figur. Kompt ein zahl die du mit einer Figur ſchreiben magſt
ſo ſetz die unden. Wo mit zweyen
ſchreib die erſte
Die andere behalt im ſinn. Darnach duplir die ander
und gib darzu
das du behalten haſt
und ſchreib abermals die erſte Figur
wo zwo vorhanden
und duplir fort bis zur letzten
die ſchreibe ganz aus
als folgende Exempel auſweiſen.

8.1.2 Die Definition des Begriffs

Was sind Algorithmen?

8-7

- Ein *Algorithmus* ist eine abstrakte Folge von Handlungsanweisungen zur Lösung eines Problems.
- Eher schwaches Beispiel: *Backrezept*.
- Eher gutes Beispiel: *Anleitung zum schriftlichen Multiplizieren*.

Was ist der Unterschied zwischen Algorithmen und Programmen?

8-8

Eigenschaften von Algorithmen:

- Ein Algorithmus beschreibt *abstrakt*, wie ein Problem zu lösen ist.
- Ein Algorithmus ist *nicht* an einen bestimmten Computer oder eine *bestimmte Programmiersprache* gebunden.
- Derselbe Algorithmus kann oft *in vielen Situationen* benutzt werden.

Eigenschaften von Programmen:

1. Programme sind immer ganz *konkrete* Instruktionsfolgen.
2. Programme werden in einer ganz *bestimmten Programmiersprache* geschrieben.
3. Ein Programm löst *genau ein Problem* und nichts anderes.

8-9

Algorithmusbeschreibungen können unterschiedlich »genau« sein.

Die einzelnen Handlungsanweisungen eines Algorithmus können ganz unterschiedlich umfangreich sein:

- Eher elementar:
  »1. Tausche die ersten beiden Zahlen in der Liste.«
  »2. Wenn die erste Zahl größer ist als die zweite, mache bei Schritt 19. weiter.«
- Eher komplex:
  »1. Sortiere die Liste.«
  »2. Entferne das Maximum.«

Regie

Vorführung des Sortieralgorithmus auf drei echte Spielkarten.

8-10

✎ Zur Übung

Ein möglicher Algorithmus zum Sortieren von drei Spielkarten funktioniert wie folgt:

1. Falls die linke Karte größer ist als die mittlere, tausche sie.
2. Falls die mittlere Karte größer ist als die rechte, tausche sie.
3. Falls die linke Karte größer ist als die mittlere, tausche sie.

Schreiben Sie einen Algorithmus zum Sortieren von vier Karten auf.
Zusatzfrage: Kommt Ihr Algorithmus mit der minimalen Anzahl an Schritten aus (oder geht es auch schneller)?

8.1.3 Der Aufbau von Algorithmen – Steuerungsanweisungen

8-11

Es gibt immer wiederkehrende Strukturen in Algorithmen.

Die allermeisten Algorithmen benutzen Varianten der folgenden so genannten *Steuerungsanweisungen*:

1. Hintereinanderausführung von Anweisungen.
   Beispiel: Erst tue dies, dann jenes, dann dieses.
2. Bedingte Anweisungen.
   Beispiel: Falls dies gilt, tue jenes, sonst dieses.
3. Schleifen.
   Beispiel: Solange jenes gilt, tue folgendes:
   Beispiel: Tue folgendes 100 mal:
4. Sprünge (considered harmful!).
   Beispiel: Mache nun dort weiter.

8-12

Steuerungsanweisungen bei Adam Riese

1. Hintereinanderausführung.
2. Bedingte Anweisungen.
3. Schleifen.
4. Sprünge.

Lehret wie du ein zahl zweyfaltigen ſolt.
Thu ihm alſo: Schreib die zahl vor dich
mach ein Linien darunter
heb an zu forderſt
Duplir die erſte Figur. Kompt ein zahl die du mit einer Figur ſchreiben magſt
ſo ſetz die unden. Wo mit zweyen
ſchreib die erſte
Die andere behalt im ſinn. Darnach duplir die ander
und gib darzu
das du behalten haſt
und ſchreib abermals die erſte Figur
wo zwo vorhanden
und duplir fort bis zur letzten
die ſchreibe ganz aus
als folgende Exempel auſweiſen.

Historischer Exkurs

Zum Thema »Sprüngen« wurde oben angemerkt, dass sie »considered harmful« seien. Dies geht auf eine berühmte, sehr alte Arbeit von Dijkstra zurück mit dem Titel »Goto Considered Harmful«. In den 1970er Jahren wurden die ersten Computer wirklich groß und schnell und es entstand zum ersten Mal die Notwendigkeit, »große, umfangreiche« Software zu erstellen (im Vergleich zu dem Betriebssystem auf Ihrem Smartphone war diese Software zwar noch putzig klein, aber sie war eben groß genug, dass keine Einzelperson sie mehr komplett »im Kopf« haben konnte).

Bei der Programmierung dieser ersten größeren Softwaresysteme schälte sich nun nach und nach ein Problem heraus: Der Programmcode war schlichtweg zu unübersichtlich. Dijkstra wollte mit seiner Arbeit darauf hinweisen, dass die besagten *Sprünge* im Code das zentrale Problem darstellten: in solch alter Software »sprang« die CPU ständig wild im Code herum. Wenn man die verschiedenen Ausführungen einmal visualisiert (was man machen kann), dann gleich das Durcheinander einem wilden Knäuel; der Fachausdruck ist daher »Spaghetti-Code«.

Die »Errettung« vom Spaghetti-Code kam in Form der so genannten Strukturierten Programmierung, was im Wesentlichen nur bedeutet, dass wir heute eben keine Sprünge benutzen und stattdessen alles mit Hilfe von Schleifen und Alternativen erledigen. Tatsächlich war Dijkstras Aufsatz dermaßen einflussreich, dass es in vielen Programmiersprachen heute überhaupt keine Sprünge mehr *gibt.*

Es sollte aber nicht verschwiegen werden, dass Sprünge nur in so genannten »Höheren Programmiersprachen« wie Java oder Lua nicht mehr genutzt werden. Die »Maschinensprache«, die die CPU direkt versteht, nutzt wilden Sprüngen hingegen weiterhin exzessiv.

8.2 Spezifikationen

Spezifikationen sind Problembeschreibungen.

8-13

Probleme lassen sich nur dann sinnvoll lösen, wenn das Problem klar gestellt ist. Eine »klare Problemstellung« nennt man eine *Spezifikation*. Sie sollte die folgende Fragen umfassend beantworten:

1. Was sind die relevanten Rahmenbedingungen?
2. Welche Hilfsmittel und Basisoperationen sind zugelassen?
3. Wann ist eine Lösung korrekt oder zumindest akzeptabel?

Gute Spezifikationen zu erstellen ist schwierig.

8-14

Spezifikationen dienen oft dazu zu klären, was eine zu erstellende Software später leisten soll. Leider wissen Kunden aber nicht, was sie eigentlich wollen. Selbst Spezifikationen zu einfachsten Problemen lassen oft Fragen offen.
Beispiel: Berechne die Summe der ersten n Zahlen. Als elementare Operationen stehen Additionen und Vergleiche zur Verfügung.
Größe Spezifikationen sind in aller Regel in sich widersprüchlich.
Beispiel: Auf Seite 50 der Spezifikation steht, dass beim Drücken des ersten Knopfes der Text rot wird. Auf Seite 150 derselben Spezifikation steht dann, dass beim Drücken dieses Knopfes der Text grün werden soll.

8.3 Programmiersprachen

8.3.1 Wozu dienen Programmiersprachen?

Von der Idee zur Instruktionsfolge.

8-15

Spezifikation

Gegeben ist eine natürliche Zahl. Sie soll verdoppelt werden.

Idee

Verdopple die Zahl stellenweise von rechts nach links und beachte den Übertrag.

Algorithmus (Notation von Riese)

Thu ihm alſo: Schreib die zahl vor dich
mach ein Linien darunter
heb an zu forderſt
Duplir die erſte Figur. Kompt ein zahl die du mit einer Figur ſchreiben magſt
ſo ſetz die unden. Wo mit zweyen
ſchreib die erſte
Die andere behalt im ſinn. Darnach duplir die ander
und gib darzu
das du behalten haſt
und ſchreib abermals die erſte Figur
wo zwo vorhanden
und duplir fort bis zur letzten
die ſchreibe ganz aus

Algorithmus (modernere Notation)

Für jede Ziffer, beginnend mit der letzten, tue folgendes:

1. Verdopple die Ziffer.
2. Addiere einen eventuell vorhandenen Übertrag hinzu.
3. Schreibe die letzte Ziffer der Summe unter die gerade behandelte Ziffer.
4. Merke die erste Ziffer des Übertrags.

Falls nun noch ein Übertrag vorhanden ist, schreibe diesen links von allem.

Algorithmus (Pseudocode)

```
1  input Ziffern z_1, ..., z_n
2  c ← 0
3  for i ← n, ..., 1 do
4    d ← 2z_i + c
5    z'_i ← d mod 10
6    c ← ⌊d/10⌋
7  z'_0 ← c
8  for i ← 0, ..., n do
9    output z'_i
```

Algorithmus (Java)

```
void verdopple(int[] ziffern, int n)
{
  int[] ziffern_verdoppelt = new int[n+1];
  int carry = 0;
  for (int i = n; i >= 1; i = i-1)
  {
    int d = 2*ziffern[i] + carry;
    ziffern_verdoppelt[i] = d % 10;
    carry = d / 10;
  }
  ziffern_verdoppelt[0] = carry;
  for (int i = 0; i <= n; i = i+1)
    System.out.print(i);
}
```


Programmiersprachen dienen zum Aufschreiben von Algorithmen.

In einer Programmiersprache lässt sich ein Algorithmus so formulieren, dass ein Computer ihn versteht. Programmiersprachen sind unterschiedlich »maschinennah«: das Spektrum reicht von »Maschinensprache« bis zu Hochsprachen und es gibt hunderte von ihnen.
Genau wie natürliche Sprachen müssen Programmiersprachen von Menschen mehr oder weniger mühselig erlernt werden.

8.3.2 Übersetzer

Übersetzer sind Programme, die Sprachen ineinander übersetzen. 8-17

Die CPU »versteht« lediglich Maschinensprache. Wir wollen aber in einer Hochsprache wie Java programmieren, weshalb wir einen *Übersetzer* brauchen. Programme werden heutzutage von einer ganzen Kette von Übersetzern in immer maschinennähere Sprachen übersetzt.

Zusammenfassung dieses Kapitels

8-18

▶ Algorithmus

Ein *Algorithmus* ist eine abstrakte Folge von Handlungsanweisungen zur Lösung eines Problems.

▶ Programmiersprache

Ein *Programmiersprache* definiert eine Art, Algorithmen aufzuschreiben.

▶ Programm

Ein *Programm* ist ein in einer Programmiersprache verfasster Text, der einen Algorithmus *implementiert.*

▶ Spezifikation

Eine *Spezifikation* legt fest, was ein Programm leisten soll.

▶ Übersetzer

Ein *Übersetzer* ist ein Programm, das Programmtexte aus einer Programmiersprache in eine andere übersetzt.

▶ Der Arbeitsfluss beim Lösen eines Problems.

1. Ausgangspunkt ist ein *Problem*.
2. Um sich klar zu machen, was man will, erstellt man ein *Spezifikation*.
3. Dann entwickelt man eine *Lösungsidee*.
4. Diese verfeinert man zu einem *Lösungsalgorithmus*.
5. Diesen implementiert man als ein *Programm* in einer Hochsprache.
6. Auf dieses wendet man einen *Übersetzer* an, um eine Instruktionsfolge zu erhalten.

9-1

Kapitel 9
Imperative Programmierung

Tu was ich sage! (Leider nicht: Tu was ich meine.)

9-2

Lernziele dieses Kapitels

1. Zuweisungen verstehen und benutzen können
2. Grundlegende Steuerungsanweisungen verstehen

Inhalte dieses Kapitels

9.1 Arten von Programmiersprachen 75

9.2 Berechnungen 77
9.2.1 Variablen 77
9.2.2 Ausdrücke 77
9.2.3 Zuweisungen 78

9.3 Steuerungsanweisungen 78
9.3.1 Komposition 78
9.3.2 Alternativen 78
9.3.3 While-Schleife 79

Übungen zu diesem Kapitel 80

Worum es heute geht

Den meisten von uns wird es nie vergönnt sein, Diktator eines einigermaßen großen Landes zu werden. Lernen Sie doch stattdessen eine imperative Programmiersprache! Bei diesen Sprachen tanzt alles nach Ihrer Nase: Programme bestehen aus langen Abfolgen von *Befehlen*, gerne auch *Kommandos* genannt. Mit »Sprungbefehlen« können Sie den Rechner schinden, bis er so richtig ins Schwitzen kommt. Und wenn er gerade schön warm geworden ist, die Lüfter hochschalten und sein Puls bei 4GHz liegt, dann geben Sie ihm doch mit einer Endlosschleife den Rest. Hier gehorcht Ihnen endlich mal jemand aufs Wort.

Das Herumkommandieren ist allerdings nicht ganz einfach, was hauptsächlich daran liegt, dass Ihre Befehle buchstabengetreu befolgt werden. Das ist in aller Regel nicht, was Sie wollen, denn wenn Sie Ihrem Computer befehlen »kaufe mir einen Sack Kartoffeln«, so dürfen Sie sich nicht wundern, wenn er mit einem Zentner Saatgut wiederkommt. Genauso wird die Anweisung »rufe doch mal Frau Merkel an« nur dazu führen, dass sich Ihr Rechner vor das Kanzleramt stellt und laut zu rufen beginnt.

Hinzu kommt die Schwierigkeit, dass Sie ihre Befehle auch noch in der richtigen Sprache formulieren müssen. Es gibt einen ganzen Wust von Regeln, die eingehalten werden wollen, schon die kleinste Abweichung wird unerbittlich durch Fehlermeldungen aller Art bestraft. Es grenzt manchmal schon an Arbeitsverweigerung seitens des Rechners, wenn Ihnen dieser mitteilt, dass »dieser lange und komplexe Befehl zwar perfekt verstanden wurde und auch ohne weiteres ausgeführt werden könnte, dass aber laut Grammatikregeln aus der Vorschrift ISO/IEC 14882:2003 das letzte Zeichen das Befehls ein Semikolon hätte sein müssen, weshalb dies nun nicht geschieht«. Verglichen mit der Akribie, mit der Sie Programmierbefehle formulieren müssen, ist jedes Steuererklärungsformular ein anarchistisches Pamphlet.

9.1 Arten von Programmiersprachen

Wiederholung: Der Arbeitsfluss beim Lösen eines Problems.

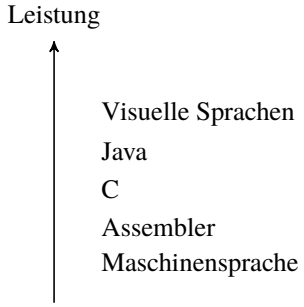
9-4

1. Ausgangspunkt ist ein *Problem*.
2. Um sich klar zu machen, was man will, erstellt man eine *Spezifikation*.
3. Dann entwickelt man eine *Lösungsidee*.
4. Diese verfeinert man zu einem *Lösungsalgorithmus*.
5. Diesen implementiert man als ein *Programm* in einer Hochsprache.
6. Auf dieses wendet man einen *Übersetzer* an, um eine Instruktionsfolge zu erhalten.

Programmiersprachen unterscheiden sich in verschiedenen Dimensionen.

9-5

Erste Dimension: Maschinennähe



Zweite Dimension: Paradigma

- Funktionale Sprachen (wenig verbreitet): Lisp, Scheme, ML
- Logische Sprachen (sehr wenig verbreitet): Prolog, Datalog
- Objektorientierte Sprachen (sehr verbreitet): Java, C++, C#, Modula-3, Smalltalk
- Imperative Sprachen (sehr verbreitet): C, Pascal, Maschinensprachen.

Die Evolution von Programmierern.

9-6

Grundschule

```
10 PRINT "Hallo␣Welt."
20 END
```

Gymnasium

```
program Hello(input, output)
begin
  writeln('Hello␣World')
end.
```

Uni, erstes Semester

```
#include <stdio.h>
void main(void)
{
    char *message[] = {"Hello␣", "World"};
    int i;

    for(i = 0; i < 2; ++i)
      printf("%s", message[i]);
    printf("\n");
}
```

Professioneller Programmierer

```
#include <iostream>
#include <iomanip>
class string
{
  private:
    int size;
    char *ptr;
    string() : size(0), ptr(new char[1]) { ptr[0] = 0; }
    string(const string &s) : size(s.size)
    {
      ptr = new char[size + 1];
      strcpy(ptr, s.ptr);
    }
    ~string() { delete [] ptr; }
    friend ostream &operator <<(ostream &, const string &);
};

ostream &operator<<(ostream &stream, const string &s)
{ return(stream << s.ptr); }

int main()
{
    string str;
    str = "Hello␣World";
    cout << str << endl;
    return(0);
}
```

Hacker (Anfänger)

```
#!/usr/local/bin/perl
$msg="Hello,␣world.\n";
if ($#ARGV >= 0) {
  while(defined($arg=shift(@ARGV))) {
    $outfilename = $arg;
    open(FILE, ">" . $outfilename) ||
      die "Can't␣write␣$arg:␣$!\n";
    print (FILE $msg);
    close(FILE) || die "Can't␣close␣$arg:␣$";
  }
} else {
  print ($msg);
}
1;
```

Hacker (Profi)

```
> cc -o a.out ~/src/misc/hw/hw.c
> a.out
```

Hacker (Guru)

```
> echo "Hello,␣world."
```

Junior-Professor

```
> mail -s "Hallo␣Welt." stockhus@uni-luebeck.de
Koenntest Du mir bis naechste Woche ein
Programm schreiben, das "Hallo␣Welt"
ausgibt? Danke.
^D
```

W2-Professor

```
> mail mitarbeiter@theory.uni-luebeck.de
Ich brauche bis zur 14-Uhr-Vorlesung
ein Hallo-Welt-Programm.
^D
```

W3-Professor

Sehr geehrte Damen und Herren,

im Anhang finden Sie den Verlängerungsantrag für das Projekt »Initiale Maschine-Mensch-Kommunikation«.

Wir bitten um Verlängerung um weitere zwei Jahre und Aufstockung um eine weitere E13 Stelle.

Mit freundlichen Grüßen
Ich

9.2 Berechnungen

9.2.1 Variablen

Variablen dienen zur Speicherung von Werten. 9-7

Ähnlich wie in der Mathematik stehen *Variablen* für veränderliche Werte. *Anders* als in der Mathematik können Variablen im Computer ihren Wert *tatsächlich ändern*. Variablen haben einen *Namen* (englisch *identifier*), der in Java aus Buchstaben und Zahlen bestehen darf.
Beispiel: Java-Variablennamen sind `i`, `index1`, `mein_toller_Variablenname`.

9.2.2 Ausdrücke

Ausdrücke bestehen aus der Anwendung von Funktionen auf Variablen und Konstanten. 9-8

Ein *Ausdruck* (englisch *expression*) kombiniert den Wert von verschiedenen Variablen zu einem neuen Wert.
Beispiel: Der Ausdruck $x+y$ gibt den Wert der Summe der Variablen x und y an.
Beispiel: Der Ausdruck $2\cdot(i+1)$ gibt den doppelten Wert des Nachfolgers von i an.
Beispiel: Der Ausdruck $\text{pow}(x,2)$ gibt denselben Wert wie $x\cdot x$ an.

Boolesche Ausdrücke können wahr oder falsch sein. 9-9

Eine besondere Art Ausdrücke sind *boolesche* Ausdrücke. Wie anderen Ausdrücke auch kombinieren sie Variablen und Konstanten zu einem neuen Wert – nun allerdings zu einem der beiden Werte **`true`** oder **`false`**.
Beispiel: Der Ausdruck $x>5$ ist genau dann gleich **`true`**, wenn x größer als fünf ist.
Beispiel: Der Ausdruck $x<y\cdot 2$ ist genau dann gleich **`true`**, wenn x kleiner dem Doppelten von y ist.
Beispiel: Der Ausdruck $2\leq x\wedge x\leq 5$ ist genau dann gleich **`true`**, wenn x zwischen 2 und 5 liegt.

Schreibweisen für Operatoren zur Bildung von booleschen Ausdrücken. 9-10

Mathematisch/Pseudocode	Pascal, Modula, Oberon	C, C++, C#
$x=y$	`x = y`	`x == y`
$x\neq y$	`x <> y`	`x != y`
$x\leq y$	`x <= y`	`x <= y`
$x\geq y$	`x >= y`	`x >= y`
$x<y$	`x < y`	`x < y`
$x>y$	`x > y`	`x > y`

9.2.3 Zuweisungen

9-11 Zuweisung dienen dazu, Variablen einen neuen Wert zu geben.

Eine *Zuweisung* ist der elementarste Befehl, den es in imperativen Programmen gibt. Eine Zuweisung weist den Computer an, einer Variablen den Wert zu geben, den ein Ausdruck gerade hat.

Beispiel 1. Die Zuweisung $x \leftarrow x + y$ besagt, dass x *nach* Ausführung der Zuweisung den Wert haben soll, den $x + y$ *vorher* hatte.

9-12 ✎ Zur Übung

Finden Sie heraus, welche Werte die Variablen x und y nach Ausführung folgender Zuweisungen haben:

```
1 x ← 5
2 y ← 10
3 z ← x + y
4 x ← z
5 z ← y
6 y ← z · z − y
7 x ← z + x + y
```

9-13 ✎ Zur Übung

Finden Sie heraus, welche Werte die Variablen x und y nach Ausführung folgender Zuweisungen haben:

```
1 y ← x + y
2 x ← y − x
3 y ← y − x
```

9.3 Steuerungsanweisungen

9.3.1 Komposition

9-14 Die Komposition bezeichnet einfach die Hintereinanderausführung von Befehlen.

Imperative Programme bestehen aus Folgen von Befehlen. Solche Folgen entstehen durch *Komposition* von Befehlen. Da diese Steuerungsanweisung so wichtig und einfach ist, ist sie syntaktisch sehr einfach:

- In Pascal deutet ein Semikolon die Komposition von Befehlen an.
- In Java kann man die Zuweisungen einfach hintereinander weg schreiben.

9.3.2 Alternativen

9-15 Die Alternativen-Steuerungsanweisung erlaubt es, in Abhängigkeit eines Ausdrucks unterschiedliche Dinge zu tun.

Die Alternative besteht aus zwei bis drei Teilen:

1. Einer *Bedingung*.
2. Einem *Then-Zweig*.
3. Einem *Else-Zweig* (optional).

Die Bedingung ist ein boolescher Ausdruck. Dieser wird als erstes ausgewertet:

- Wenn er wahr ist, werden die Anweisungen im Then-Zweig befolgt.
- Wenn er falsch ist, werden die Anweisungen im Else-Zweig befolgt. (Falls dieser fehlt, geht es einfach mit dem nächsten Befehl weiter.)

Schreibweise der Alternativen in Pseudo-Code. 9-16

```
if Bedingung then
  Anweisung des Then-Zweigs
else
  Anweisung des Else-Zweigs
```

✎ Zur Übung 9-17
Welche Werte habe die Variablen am Ende von folgendem Programm?

```
x ← 8
y ← 5
if x < y then
  x ← y
else
  z ← x
  x ← y
  y ← z
```

9.3.3 While-Schleife

Die While-Steuerungsanweisung erlaubt es, eine Befehlsfolge wiederholt auszuführen. 9-18
Die While-Schleife besteht aus zwei Teilen:

1. Einer *Bedingung*.
2. Einem *Körper*.

Die Bedingung ist ein boolescher Ausdruck. Dieser wird bei jedem Schleifendurchlauf am Anfang ausgewertet. Wenn die Bedingung wahr ist, wird der Körper ausgewertet, danach wieder die Bedingung, dann wieder der Körper und so fort. Wenn die Bedingung falsch ist, wird die Schleife beendet.

Schreibweise der Schleife in Pseudo-Code. 9-19

```
while Bedingung do
  Anweisung des Körpers
```

✎ Zur Übung 9-20
Welchen Wert hat *sum* am Ende von folgendem Programm?

```
n ← 0
sum ← 0
while n ≤1000 do
  sum ← sum + n
  n ← n + 1
```

Zusammenfassung dieses Kapitels

9-21

▶ Variablen

Ein »Computer-Variable« liegt irgendwo im Speicher und *ändert ihren Wert* nach jeder Zuweisung.

▶ Zuweisung

Ein *Zuweisung* ist ein Befehl der Form $v \leftarrow a$. Hierzu wird zuerst der *Ausdruck* a ausgewertet und das Resultat danach in den Speicherbereich geschrieben, den die Variable v belegt.

▶ Alternative

Eine *Alternative* hat die Form

```
1 if Bedingung then
2   Anweisung des Then-Zweigs
3 else
4   Anweisung des Else-Zweigs
```

▶ Schleife

Eine *Schleife* hat die Form

```
1 while Bedingung do
2   Körper
```

Übungen zu diesem Kapitel

Bei den folgenden Aufgaben geht es darum, einfache Algorithmen selbst zu formulieren. In allen Programmen ist am Anfang in einer (oder mehreren) Variablen ein Eingabewert gespeichert. Am Ende des Programms soll in der Variable *loesung* der Ausgabewert gespeichert sein.

Übung 9.1 Algorithmus zur Verzinsung I, einfach, mit Lösung

Ein Konto wird mit 4% im Jahr verzinst. In der Variable *anfangskontostand* ist ein Kontostand gespeichert. Geben Sie einen Algorithmus in Pseudo-Code an, der berechnet, wie hoch der Kontostand nach drei Jahren ist.

Übung 9.2 Algorithmus zur Verzinsung II, mittel

Ein Konto wird mit 4% im Jahr verzinst. In der Variable *anfangskontostand* ist ein Kontostand gespeichert, in der Variable *jahre* eine Anzahl von Jahren. Geben Sie einen Algorithmus in Pseudo-Code an, der berechnet, wie hoch der Kontostand nach der in der Variable *jahre* gespeicherten Anzahl an Jahren ist. (Falls also *jahre* am Anfang 3 ist, sollte dasselbe wie in Übung 9.1 herauskommen.)

Übung 9.3 Algorithmus zur Verzinsung III, mittel

Eine Variable *zinssatz* speichert eine Zahl in Prozent. Ein Konto wird mit diesem Zinssatz pro Jahr verzinst. In der Variable emphanfangskontostand ist ein Kontostand gespeichert, in der Variable *jahre* eine Anzahl von Jahren. Geben Sie einen Algorithmus in Pseudo-Code an, der berechnet, wie hoch der Kontostand nach der in der Variable *jahre* gespeicherten Anzahl an Jahren ist.

Übung 9.4 Trivialer Algorithmus, einfach

In der Variable *alter* ist das Alter einer Person gespeichert, die in diesem Jahr schon Geburtstag hatte. In der Variable *jahr* ist das aktuelle Jahr gespeichert. Geben Sie einen Algorithmus an, der berechnet, in welchem Jahr die Person geboren wurde.

Übung 9.5 Schaltjahr-Algorithmus, schwer

In der Variable *jahr* ist eine Jahreszahl gespeichert. Geben Sie einen Algorithmus an, der die Anzahl Tage in diesem Jahr berechnet.

Übung 9.6 Monatszahl-Algorithmus, mittel

In der Variable *monat* ist ein Monat als Zahl von 1 bis 12 gespeichert. Geben Sie einen Algorithmus an, der die Anzahl Tage in diesem Monat berechnet (für den Februar soll 28 zurückgegeben werden).

Übung 9.7 Minimumsbildung, mittel

In den Variablen a, b, c, d und e sind beliebige Zahlen gespeichert. Geben Sie einen Algorithmus an, der die kleinste dieser Zahlen berechnet.

Übung 9.8 Medianbildung, schwer

In den Variablen a, b und c sind beliebige Zahlen gespeichert. Geben Sie einen Algorithmus an, der den Median dieser Zahlen berechnet. Können Sie dies auch für fünf Variablen?

Übung 9.9 Sekunden seit Sylvester, mittel

In den Variablen *monat*, *tag*, *stunde*, *minute* ist das aktuelle Datum dieses Jahres gespeichert. Geben Sie einen Algorithmus an, der die Sekunden seit Sylvester berechnet.

Übung 9.10 Freitag der 13., schwer

In den Variablen *jahr* ist eine Jahreszahl gegeben. Geben Sie einen Algorithmus an, der berechnet, wie oft es in diesem Jahr einen Freitag den 13. gibt.

Übung 9.11 Iteriertes Multiplizieren I, mittel

In der Variable *ecoli* ist die Anzahl von Bakterien in einer Kultur gespeichert, die sich ungehemmt vermehren können. In der Variable *minuten* ist eine Zeit angegeben. Geben Sie einen Algorithmus an, der die Anzahl von Bakterien berechnet, die nach der gegebenen Anzahl Minuten vorhanden sind. (Wählen Sie die Vermehrungsrate biologisch sinnvoll.)

Übung 9.12 Iteriertes Multiplizieren II, mittel

In der Variable n ist eine Zahl gespeichert. Geben Sie einen Algorithmus zur Berechnung von $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$ an.

Übung 9.13 Quersumme, schwer

Geben Sie einen Algorithmus an, der die *Quersumme* einer Variable n berechnet. Beispiel: Die Quersumme von 5245 ist $5+2+4+5=16$.

Übung 9.14 Anzahl der Teiler bestimmen, schwer

Ein Teiler einer Zahl n ist eine Zahl t, so dass sich n durch t ohne Rest teilen lässt. Anders gesprochen: Berechnet man n modulo t, so muss sich 0 ergeben (mathematisch geschrieben $n \,mod\, t = 0$).

Gesucht ist ein Algorithmus, der die *Anzahl der Teiler einer Zahl* n bestimmt. Hier ist eine grobe Beschreibung eines solchen Algorithmus: Ein Zähler wird auf 0 gesetzt. In einer Schleife iteriert man über alle Zahlen zwischen 1 und n. Dies sind gerade die potentiellen Teiler von n. Für jede dieser Zahlen überprüft man, ob sie n ohne Rest teilt. Wenn dies der Fall ist, so wird der Zähler um eins hochgezählt. Nach Beendigung der Schleife gibt man den Stand des Zählers aus.

Formulieren Sie den Algorithmus im Pseudo-Code.

10-1

Kapitel 10
Die Programmiersprache Java
Eine Programmiersprache für alles und jedes

10-2

Lernziele dieses Kapitels

1. Allgemeine Syntaxelemente wie Bezeichner, Ausdrücke und Kommentare von Java beherrschen
2. Zuweisungsbasierte Algorithmen als imperative Java-Programme formulieren können
3. Ein Java-Programm selbstständig erstellen und übersetzen können

Inhalte dieses Kapitels

10.1	Überblick über Java	83
10.2	Die Java-Syntax	84
10.2.1	Vom Algorithmus zum Programm	84
10.2.2	Bezeichner	84
10.2.3	Zahlen, Ausdrücke, Zuweisungen	84
10.2.4	Variablendeklaration	86
10.2.5	Formatierung und Kommentare	87
10.3	Java-Übersetzer	88
10.3.1	Übersetzerkonzept bei Java	88
10.3.2	Übersetzung in BlueJ und Eclipse	88
	Übungen zu diesem Kapitel	89

Worum es heute geht

Das Programmieren war in den bisherigen Kapiteln eher eine Trockenübung. Da wurden Variablen zugewiesen, Schleifen durchlaufen und Alternativen abgewogen was das Zeug hält; aber alles nur mit Papier und Bleistift. In diesem Kapitel wird es nun endlich ernst und Ihr erstes echtes Computerprogramm entsteht, denn in diesem Kapitel lernen Sie (die ersten Elemente) einer »Sprache«, in der Sie Algorithmen aufschreiben können.

Diese Sprache nennt sich *Java*. Mit der Wahl dieses Namens durch die Firma Sun, in deren Laboren sie entwickelt wurde, hat man wirklich ein glückliches Händchen bewiesen. Englischsprachige Menschen denken bei diesem Wort zuerst an Kaffee und dann an Inselurlaub in den Tropen. Das ist wirklich schöner als bei Programmiersprachen mit Namen wie *Basic* oder *Logo* (hört sich eher nach Tigerentenclub an) oder auch C, C++, C– und C# (eher zufällige Symbolfolgen als vernünftige Namen). Aus Wikipedia erfährt man, dass das Wort »Java« noch weitere interessante Bedeutungen hat: Vier Schiffe hießen Java, darunter eine 44-Kanonen-Fregatte der US-Navy; mehrere Tierarten heißen Java, darunter das seltene Javahuhn, der schweinsäugige Javahai und das vom Aussterben bedrohte Javanashorn; eine Zigarettenmarke, eine Kaffeebohne und eine Cachaçasorte heißen Java; Java ist ein französischer Bal-Musette-Tanz; und, was eher weniger bekannt ist, Java ist auch ein grobes, locker eingestelltes Grundgewebe aus Leinen oder Baumwolle für Stickereiarbeiten.

Für diese Vorlesung wurde Java aber nicht wegen des originellen Namens ausgewählt, sondern weil sei eine sehr verbreitete Sprache ist und in realen System heute viel eingesetzt wird. Wenn man eine erste Fremdsprache aussuchen soll, dann sollte man auch eher Englisch als, sagen wir, Suahelie oder Esperanto wählen – selbst wenn Esperanto vielleicht einfacher ist. Wichtig erschien auch, dass viele andere wichtige Sprachen wie C++ oder JavaScript sehr ähnlich zu Java sind – können Sie Java, so werden Ihnen diese Sprachen auch nicht schwerfallen.

Es soll aber nicht verschwiegen werden, dass Java nicht gerade speziell für Anfänger konzipiert wurde. Deshalb wird vieles am Anfang (und manches auch noch später) mysteriös

oder umständlich erscheinen. Hier hilft nur Mut zur Lücke: Probieren Sie am besten selbst möglichst viel herum.

10.1 Überblick über Java

Eine sehr kurze Geschichte von Java. 10-4

1991 Ursprünglich sollte Java zur Programmierung kleiner Geräte (wie Toaster oder Waschmaschinen) dienen.

1993 Der Anspruch wurde geändert: Java sollte nun auf möglichst vielen unterschiedlichen Computern sofort laufen.

1995 Die Firma SUN stellt Java kostenlos zur Verfügung inklusive einer kompletten Entwicklungsumgebung.

heute Java wird von vielen Firmen und Universitäten eingesetzt.

Aus der Werbung. 10-5

- Java ist *portabel*: Java-Programme können auf beliebigen Rechnern ausgeführt werden.
- Java ist *sicher*: Bei der Ausführung von Java-Programmen kann man garantieren, dass sie keinen Unfug anstellen.
- Java ist *modern*: Java ist ein Hochsprache, in die viel Erfahrung sowohl aus Theorie und Praxis eingeflossen ist.
- Java wird *gut unterstützt*: Java wird von vielen Firmen und Universitäten eingesetzt und gefördert.
- Java ist *gut dokumentiert*: Umfangreiche Dokumentation ist frei verfügbar.
- Java hat eine *Standardsyntax*: Java hat dieselbe Syntax wie C und C++. Es lässt sich von erfahrenen Programmierern sehr schnell erlernen.

Was die Werbung verschweigt. 10-6

- Java ist langsamer als C und C++.
- Java ist für Anfänger schwer zu lernen.
- Java ist imperativ und deshalb nicht so elegant wie andere Sprachen.
- Java hat Designfehler.

Was gehört alles zu Java? 10-7

Zunächst ist Java eine *Programmiersprache*, also eine Art, Algorithmen aufzuschreiben. Damit man damit etwas anfangen kann, braucht man aber auch einen *Übersetzer*. Um nicht nur völlig triviale Dinge programmieren zu können, werden noch *Bibliotheken* benötigt. Sie enthalten vorprogrammierte Algorithmen und Datenstrukturen für viele Probleme. Schließlich gehört zu Java eine umfangreiche *Dokumentation*, die man über Google leicht findet.

Das Hallo-Welt-Programm in Java. 10-8

```
/**
 * Das Hello-Welt-Programm in Java.
 */
class Hello
{
    /**
     *  main method
     */
    public static void main(String[] args)
    {
       System.out.println("Hello, world");
    }
}
```

10.2 Die Java-Syntax

10.2.1 Vom Algorithmus zum Programm

10-9 **Ein Algorithmus, der als Java-Programm implementiert werden soll.**

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

Setze *average* auf $\frac{a+b+c}{3}$
Falls ($a \leq b$ und $a \geq c$) oder ($a \geq b$ und $a \leq c$), dann
  Setze *median* auf a
Sonst
  Falls ($b \leq a$ und $b \geq c$) oder ($b \geq a$ und $b \leq c$), dann
    Setze *median* auf b
  Sonst
    Setze *median* auf c

10.2.2 Bezeichner

10-10 **Ein Bezeichner dient zur Benennung von Variablen und anderer Dinge.**

Ein Bezeichner ist in Java eine Kette von Buchstaben und Ziffern ohne Leerzeichen, wobei das erste Zeichen *keine Ziffer* sein darf. Sie werden neben Variablen auch für die Bezeichnung von Klassen, Methoden, Objekten, Bibliotheken und vieler anderer Dinge benutzt.

10-11 **Der Algorithmus mit Bezeichnern in Java-Syntax.**

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

Setze `average` auf $\frac{\texttt{a+b+c}}{3}$
Falls ($\texttt{a} \leq \texttt{b}$ und $\texttt{a} \geq \texttt{c}$) oder ($\texttt{a} \geq \texttt{b}$ und $\texttt{a} \leq \texttt{c}$), dann
  Setze `median` auf `a`
Sonst
  Falls ($\texttt{b} \leq \texttt{a}$ und $\texttt{b} \geq \texttt{c}$) oder ($\texttt{b} \geq \texttt{a}$ und $\texttt{b} \leq \texttt{c}$), dann
    Setze `median` auf `b`
  Sonst
    Setze `median` auf `c`

10.2.3 Zahlen, Ausdrücke, Zuweisungen

10-12 **Wie werden Zahlen in Java geschrieben?**

- Ganze Zahlen können ganz normal geschrieben werden: `-56`, `5`, `100000`.
- Dezimalbrüche werden mit einem Punkt geschrieben: `-5.6`, `10.34`.
- Große Dezimalbrüche können mit Exponenten geschrieben werden: `54000000 == 54e+6`.

10-13 **Der Algorithmus mit Zahlen in Java-Syntax.**

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

Setze `average` auf $\frac{\texttt{a+b+c}}{3}$
Falls ($\texttt{a} \leq \texttt{b}$ und $\texttt{a} \geq \texttt{c}$) oder ($\texttt{a} \leq \texttt{b}$ und $\texttt{a} \geq \texttt{c}$), dann
  Setze `median` auf `a`
Sonst
  Falls ($\texttt{b} \leq \texttt{a}$ und $\texttt{b} \geq \texttt{c}$) oder ($\texttt{b} \geq \texttt{a}$ und $\texttt{b} \leq \texttt{c}$), dann
    Setze `median` auf `b`
  Sonst
    Setze `median` auf `c`

Welche Ausdrücke sind in Java möglich?

10-14

Zur Erinnerung: *Ausdrücke* verknüpfen Werte zu neuen Werten. Folgende Arten von Verknüpfung (auch Operatoren genannt) stehen zur Verfügung:

- Arithmetische (wie Addition).
- Vergleiche (wie Kleiner-Gleich).
- Boolesche (wie Und und Oder).
- Viele, viele weitere obskure.

Arithmetische Ausdrücke dienen zum Verknüpfen von Zahlen.

10-15

In Java sind folgende arithmetische Verknüpfungen erlaubt:

Verknüpfung	Schreibweise
Addition	`+`
Subtraktion	`-`
Multiplikation	`*`
Division	`/`
Modulo	`%`

Der Algorithmus mit arithmetischen Verknüpfungen in Java-Syntax.

10-16

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

Setze `average` auf `(a + b + c) / 3`
Falls (`a` $\leq$ `b` und `a` $\geq$ `c`) oder (`a` $\leq$ `b` und `a` $\geq$ `c`), dann
  Setze `median` auf `a`
Sonst
  Falls (`b` $\leq$ `a` und `b` $\geq$ `c`) oder (`b` $\geq$ `a` und `b` $\leq$ `c`), dann
    Setze `median` auf `b`
  Sonst
    Setze `median` auf `c`

Vergleichsverknüpfungen liefern boolesche Werte.

10-17

In Java sind folgende Vergleiche erlaubt:

Vergleichsart	Schreibweise
gleich	`==`
ungleich	`!=`
kleiner	`<`
größer	`>`
kleiner gleich	`<=`
größer gleich	`>=`

Der Algorithmus mit Vergleichen in Java-Syntax.

10-18

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

Setze `average` auf `(a + b + c) / 3`
Falls (`a <= b` und `a >= c`) oder (`a >= b` und `a <= c`) oder, dann
  Setze `median` auf `a`
Sonst
  Falls (`b <= a` und `b >= c`) oder (`b <= a` und `b >= c`), dann
    Setze `median` auf `b`
  Sonst
    Setze `median` auf `c`

10-19

Boolesche Verknüpfungen verknüpfen Boolesche Werte.

In Java sind folgende boolesche Verknüpfungen erlaubt:

Vergleichsart	Schreibweise
und	`&&`
oder	`||`
nicht	!

10-20

Der Algorithmus mit booleschen Verknüpfungen in Java-Syntax.

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

Setze `average` auf `(a + b + c) / 3`
Falls `(a <= b && a >= c) || (a >= b && a <= c)`, dann
  Setze `median` auf `a`
Sonst
  Falls `(b <= a && b >= c) || (b >= a && b <= c)`, dann
    Setze `median` auf `b`
  Sonst
    Setze `median` auf `c`

10-21

Zur Erinnerung: Zuweisungen weisen Variablen neue Werte zu.

Eine *Zuweisung* wie $a \leftarrow b$ wird benutzt, um einer Variable einen neuen Wert zu geben, nämlich den Wert eines Ausdrucks b. Die Variable a hat *nach* der Zuweisung den Wert, den der Ausdruck b *vor* der Zuweisung hatte. In Java schreibt man diabolischerweise `a = b;` für $a \leftarrow b$.

10-22

Der Algorithmus mit Zuweisungen in Java-Syntax.

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

```
average = (a + b + c) / 3;
Falls (a <= b && a >= c) || (a >= b && a <= c), dann
  median = a;
Sonst
  Falls (b <= a && b >= c) || (b >= a && b <= c), dann
    median = b;
  Sonst
    median = c;
```

10.2.4 Variablendeklaration

10-23

Variablen werden im Speicher abgelegt.

Für jede Variable sind im Speicher des Computers einige Bytes reserviert. Die Anzahl Bytes hängt ab vom *Typ* der Variable. Beispiele von Typen sind »ganze Zahl« oder »Zeichenkette« oder sogar »Bild« (dazu später mehr). Um die Reservierung und Freigabe des Speichers kümmert sich der Übersetzer. Variablen belegen nur so lange Speicher, wie sie gebraucht werden. Damit der Übersetzer weiß, wie viel Speicher zu reservieren ist, müssen Variablen vor ihrer Benutzung *deklariert* werden.

Syntax einer Deklaration einer Variable

```
int anzahl = 42;
```

`int`: Typ »ganze Zahl«; `anzahl`: Name der Variable; `42`: Startwert

10.2.5 Formatierung und Kommentare

Java-Programme können unterschiedlich formatiert werden. 10-24

- Wo in einem Programm Leerzeilen und Leerzeichen (auch *whitespace* genannt) erscheinen, ist in Java im Prinzip egal:

```
class Hello{public static void main
(String[]args){System.out.println(
"Hello, world");}}
```

- Wie man sein Programm *formatiert* ist also eher Geschmackssache.
- Schlecht formatierte Programme sind aber sehr schwer zu lesen.

✎ Zur Übung 10-25

Welches Programm tut nicht dasselbe wie die anderen?

```
void test ( )
{
  System.out.
  println("X");
}
```

```
void test () {
  System. out.
println("X ")
;}
```

```
          void
test () {
System. out.
println   (
"X" );}
```

Kommentare sind oft die wichtigsten Bestandteile eines Programms. 10-26

Kommentare dienen als Information für Menschen. Es gibt zwei Arten, Kommentare einzufügen:

1. Einen einzeiligen Kommentar beginnt man mit `//`. Der Kommentar ist dann der komplette Text bis zum Ende der Zeile.
2. Ein mehrzeiliger Kommentar steht zwischen `/*` und `*/`.

Anfänger kommentieren oft viel zu wenig. Fortgeschrittene kommentieren oft viel zu viel. Profis kommentieren richtig:
Kommentieren sollte man genau das, was nicht sowieso schon klar ist.

✎ Zur Übung 10-27

Vergeben Sie Schulnoten für die Qualität folgender Kommentare:

1.
```
// Jetzt werden x und y vertauscht:
y = x+y;
x = y-x;
y = y-x;
```

2.
```
/* Jetzt werden x und y addiert und das
  Ergebnis in y gespeichert. Dann wird
  x von y subtrahiert und in x gespeichert.
  Zum Schluss dann nochmal und das Ergebnis
  nach y. */
y = x+y; x = y-x; y = y-x;
```

3.
```
int i = 0; // i ist ein ganze Zahlen.
```

4.
```
int n = 1; // Anzahl der Nachrichten
```

10.3 Java-Übersetzer

10.3.1 Übersetzerkonzept bei Java

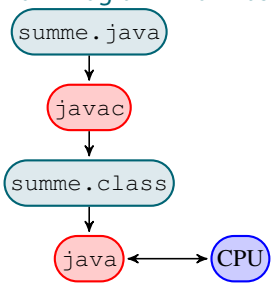
10-28

Vom Programm zur Maschinensprache bei Java.

Der Shell-Befehl `javac` ist der Java-Übersetzer. Er produziert allerdings nicht ein Programm in Maschinensprache, sondern ein Programm in einer speziellen Zwischensprache, die sich hochtrabend *Java-Byte-Code* nennt. Ein weiterer Shell-Befehl, `java`, führt den Byte-Code dann aus. Dazu übersetzt `java` den Byte-Code »On-the-fly« in Maschinencode.

10-29

Vom Programm zur Maschinensprache bei Java.



10.3.2 Übersetzung in BlueJ und Eclipse

10-30

Vereinfachung durch die Benutzung von BlueJ oder Eclipse.

Bei BlueJ und bei Eclipse ist die Benutzung etwas einfacher: Man erstellt ein neues Projekt. Darin erstellt man eine »Hauptklasse« mit einer »Methode« namens `main`. Dann drückt man auf den Compile-Knopf. Funktioniert dieses, kann das Programm durch einen Rechts-Klick auf die Klasse (BlueJ) oder durch den Menupunkt »Run« (Eclipse) starten.

10-31

Ein komplettes Programm in Java.

```
// Was die folgenden drei Zeilen bedeuten kommt später
class Durchschnittsberechnung {
    public static void main(String[] args)
    {

      // Hier kommt das eigentliche Programm
      int a = 5;
      int b = 15;
      int c = 9;
      int average = 0;
      average = (a+b+c)/3;

      // So kann man etwas auf den Bildschirm ausgeben:
      System.out.println(average);
    }
}
```

Zusammenfassung dieses Kapitels

▶ Die Programmiersprache Java

Java ist eine weitverbreitete *objektorientierte*, *imperative* Programmiersprache.

▶ Übersetzung

- Java Programme müssen *übersetzt* werden mittels `javac` und `java`.
- Die *BlueJ*- und die *Eclipse*-Entwicklungsumgebungen *vereinfachen* das Programmieren von Java-Programmen und den Übersetzungsvorgang.

▶ Java-Syntax: Deklarationen

Syntax `typ variablen_name = startwert;`

Effekt Reserviert speicher für die Variable, ihr kann nun etwas zugewiesen werden.

Beispiel `int average = 10;`

▶ Java-Syntax: Zuweisungen

Syntax `variable = ausdruck;`

Effekt Zunächst wird der Ausdruck ausgewertet. Das Ergebnis wird dann in der Variable gespeichert.

Beispiel `a = b+x;`

▶ Java-Syntax: Mögliche Verknüpfungen in Ausdrücken

Java kennt (unter anderen) die folgenden *Operatoren*: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `!`, `&&`, `||`.

Übungen zu diesem Kapitel

Übung 10.1 Algorithmus in Java-Syntax formulieren, mittel, mit Lösung

Formulieren Sie den Algorithmus von Übung 9.1 in Java-Syntax. Deklarieren Sie alle Variablen und geben Sie das Ergebnis mittels `System.out.println(loesung);` aus.

Übung 10.2 Algorithmen in Java-Syntax formulieren, mittel

Formulieren Sie nun analog zur vorherigen Aufgabe die Algorithmen von Übung 9.4, 9.5, 9.6, 9.7, 9.8, 9.9 und 9.10 in Java-Syntax.

Übung 10.3 Java-Programm mit der Shell übersetzen, leicht

1. Für diese Aufgabe sollten Sie zunächst ein neues Verzeichnis erstellen.
2. In einem Editor erstellen Sie eine neue Datei mit dem Namen `Durchschnittsberechnung.java`
3. Kopieren Sie den Text des Programms von Projektion 10-31 (beispielsweise durch Kopieren und Einfügen) in diese Datei und speichern Sie diese.
4. Öffnen Sie nun eine Shell und wechseln in das Verzeichnis.
5. Übersetzen Sie das Programm mit folgendem Befehl:

```
javac Durchschnittsberechnung.java
```

6. Führen Sie das Programm mit folgendem Befehl aus:

```
java Durchschnittsberechnung
```

7. Ändern Sie nun die Zahlen im Programm, kompilieren Sie das Programm neu und überprüfen Sie, was passiert, wenn Sie das Programm ausführen!

Übung 10.4 Java-Programm mit BlueJ übersetzen, leicht

1. Für diese Aufgabe können Sie dasselbe Verzeichnis benutzen wie in der vorherigen.
2. Rufen Sie BlueJ auf.
3. In BlueJ erstellen Sie ein neues Projekt in diesem Verzeichnis.
4. Mit einem Rechts-Klick erstellen Sie eine neue Klasse in diesem Projekt. Geben Sie der Klasse den Namen `Durchschnittsberechnung`.
5. Klicken Sie auf diese Klasse, so dass sich ein Editor öffnet.
6. Ersetzen Sie den Programmtext der Klasse wieder durch den Text von Projektion 10-31.
7. Drücken Sie nun auf den Compile-Knopf.
8. Kehren Sie nun in die Projektansicht zurück (da wo man die Klasse als kleines Rechteck sieht) und klicken Sie mit Rechts auf die Klasse. Wählen Sie den Auswahlpunkt `main`. Im folgenden Dialog einfach auf Ok klicken.
9. Ändern Sie nun das Programm wieder ein wenig ab und überprüfen Sie, was passiert.

Übung 10.5 Java-Programm erstellen, mittel

Sie können für folgende Aufgabe BlueJ oder die Shell verwenden, was immer Ihnen besser gefällt.

1. Wie in den vorherigen beiden Aufgaben erstellen Sie ein neues Projekt oder eine neue Datei.
2. In dieser Datei soll nun aber eines der Programme aus Übung 9.1 bis 9.10 eingefügt werden.
3. Erstellen Sie dazu das nötige Drum-Herum (**`class`** `Example`...und **`public static void`** `main`...).
4. Versuchen Sie das Programm zu übersetzen und korrigieren Sie gegebenenfalls die Fehler. Fragen Sie nach, wenn Sie einen Fehler nicht selbst beheben können.
5. Probieren Sie Ihr Programm aus.

Übung 10.6 Syntaxfehler finden, leicht, original Klausuraufgabe, mit Lösung

Ein angehender Informatikstudent hat für seine Java-Vorlesung folgende Methode programmiert:

```
1  int agz( int[] x ){
2     if( x length == 0 ){
3         return 0;
4     }
5     int r = x[0];
6     for( int i = 0 ; i < x.length ; i = i + 1 )
7         if( x[i] > r {
8             r = x[i];
9         }
10    }
11    return r;
12 }
```

1. Leider meckert der Compiler immer: »Syntax Error«! Tatsächlich hat der Student insgesamt 3 kleine, aber wichtige Zeichen vergessen. Welche sind dies, und in welcher Zeile wurden sie vergessen? Geben Sie jeweils nur das Zeichen und die Zeilennummer an!
2. Was tut die Methode `agz`, wenn alle Syntaxfehler behoben sind und man sie laufen lässt?

11-1

Kapitel 11

Steuerungsanweisungen und elementate Datentypen

Das Kleine-Ein-mal-Eins der Programmierung

11-2

Lernziele dieses Kapitels

1. Die Steuerungsanweisungen If-Then-Else, While-Schleife und For-Schleife benutzen können
2. Variablendeklarationen verstehen und benutzen können
3. Elementare Datentypen von Java kennen
4. Typkorrektheit und Typfehler verstehen

Inhalte dieses Kapitels

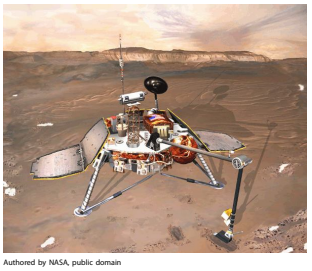
11.1	Steuerungsanweisungen	92
11.1.1	If-Then-Else	92
11.1.2	While-Schleife	93
11.1.3	For-Schleife	94
11.2	Datentypen	95
11.2.1	Der Begriff des Typs	95
11.2.2	Die Nutzen von Typen	95
11.2.3	Arten von Typen	95
11.2.4	Javas Datentypen	95
11.3	Typisierung	96
11.3.1	Typisierung von Variablen	96
11.3.2	Typisierung von Ausdrücken	96
11.3.3	Typfehler	97
	Übungen zu diesem Kapitel	98

Worum es heute geht

Hätte man zum Erstellen von Javaprogrammen nur die im letzten Kapitel eingeführten Zuweisungen zur Verfügung, so wären Javaprogramme recht langweilig: Ewige Folgen mehr oder minder sinnvoller Zuweisungen, die immer in stupider Weise abgearbeitet werden müssen. Wirklich spannend wird das Programmieren erst durch Schleifen und Alternativen. Dadurch kann »mal das und mal das« passieren – wie im richtigen Leben.

Wie schon bei der Zuweisung, die diabolischerweise mit einem einfachen Gleichheitszeichen aufgeschrieben wird, gibt es auch bei den Steuerungsanweisungen Fallstricke in der Syntax. So muss beispielsweise nach einem `if`, das immer kleingeschrieben werden muss, immer eine runde öffnende Klammer folgen, der so genannte Then-Zweig hingegen kann, muss aber nicht, in geschweiften Klammern stehen. Hier helfen nur üben, mehr üben sowie noch mehr üben; diese merkwürdige Schreibweisen müssen Ihnen in Fleisch und Blut übergehen. Vielleicht ist es dabei tröstlich zu wissen, dass es auch den besten Programmiererinnen und Programmierer dabei genauso erging wie Ihnen.

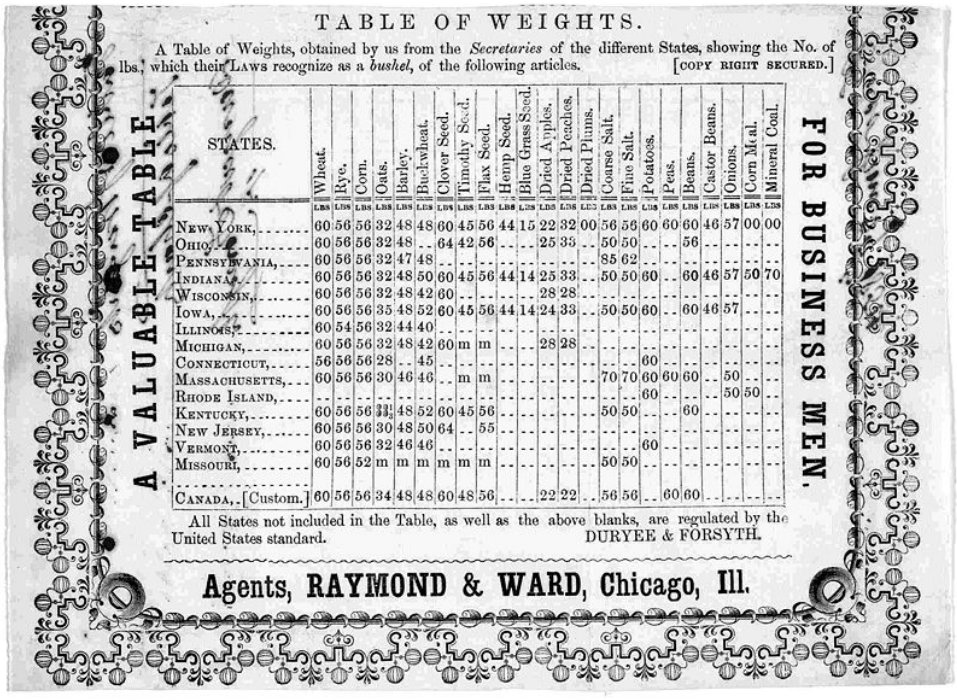
Wenn Sie die Syntax der Steuerungsanweisungen beherrschen, können Sie sich an die Datentypen heranwagen. Die Grundidee dabei: Man kann nicht Äpfel mit Birnen vergleichen.



Authored by NASA, public domain

Am 3. Dezember 1999 stürzte eine NASA-Marssonde auf den Roten Planeten. Hauptgrund hierfür war ein Einheitenfehler, es wurden metrische Angaben munter mit amerikanischen Maßen vermischt. Die Amerikaner halten an ihrem viktorianischen Maßsystem mit großem Stolz fest – und es auch durchaus lustiger, in der Küche $1\frac{3}{4}$ *Tassen* Zucker mit 5 *flüssigen Unzen* Milch zu vermengen statt 300ml Zucker mit 200ml Milch. Jedoch ist die Umrechnung von, sagen wir, amerikanische Gallonen pro amerikanischer Meile in amerikanische Tassen

pro amerikanischem Fuß wesentlich schwieriger als die von Liter pro Kilometer in Milliliter pro Meter. Deshalb benutzen auch amerikanische Ingenieure das metrische System – nur eben nicht beim Kochen und weniger wichtigen Parametern von Raumsonden. Die folgende Abbildung zeigt, dass die Lage eigentlich noch viel ernster ist: Jedes US-Staat hat seine ganz eigenen Vorstellungen darüber, wie viel genau ein *Scheffel* ist:



STATES.	Wheat.	Rye.	Corn.	Oats.	Barley.	Buckwheat.	Clover Seed.	Timothy Seed.	Flax Seed.	Hemp Seed.	Blue Grass Seed.	Dried Apples.	Dried Peaches.	Dried Plums.	Course Salt.	Fine Salt.	Potatoes.	Peas.	Beans.	Castor Beans.	Onions.	Corn Meal.	Mineral Coal.
	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS	LBS
New York,	60	56	56	32	48	48	60	45	56	44	15	22	32	00	56	56	60	60	60	46	57	00	00
Ohio,	60	56	56	32	48	..	64	42	56	..	..	25	33	..	50	50	..	..	56	..	..	..	..
Pennsylvania,	60	56	56	32	47	48	..	..	..	..	..	..	..	..	85	62	..	..	..	..	..	..	..
Indiana,	60	56	56	32	48	50	60	45	56	44	14	25	33	..	50	50	60	..	60	46	57	50	70
Wisconsin,	60	56	56	32	48	42	60	..	..	..	..	28	28	..	..	..	..	..	..	..	..	..	..
Iowa,	60	56	56	35	48	52	60	45	56	44	14	24	33	..	50	50	60	..	60	46	57	..	..
Illinois,	60	54	56	32	44	40	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
Michigan,	60	56	56	32	48	42	60	m	m	..	..	28	28	..	..	..	..	..	..	..	..	..	..
Connecticut,	56	56	56	28	..	45	..	..	..	..	..	..	..	..	..	..	60	..	..	..	..	..	..
Massachusetts,	60	56	56	30	46	46	..	m	m	..	..	..	..	..	70	70	60	60	60	..	50	..	..
Rhode Island,	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	60	..	..	..	50	50	..
Kentucky,	60	56	56	33	48	52	60	45	56	..	..	..	..	..	50	50	..	..	60	..	..	..	..
New Jersey,	60	56	56	30	48	50	64	..	55	..	..	..	..	..	..	..	..	..	..	..	..	..	..
Vermont,	60	56	56	32	46	46	..	..	..	..	..	..	..	..	..	..	60	..	..	..	..	..	..
Missouri,	60	56	52	m	m	m	m	m	m	..	..	..	..	..	50	50	..	..	..	..	..	..	..
Canada, [Custom.]	60	56	56	34	48	48	60	48	56	..	..	22	22	..	56	56	..	60	60	..	..	..	..

Public domain

Typisierung dient dazu, Probleme zu vermeiden. Jede Variable bekommt einen festen *Typ* und der Versuch, Scheffel Äpfel mit Scheffel Birnen zu vergleichen oder Meter mit Fuß wird bereits durch den Übersetzer verhindert – also lange *bevor* die Sonde abhebt. Moral von der Geschicht': Datentypen können Leben retten! (Zumindest das Leben von Marssonden.)

11.1 Steuerungsanweisungen

11.1.1 If-Then-Else

11-4

Die Alternativen-Steuerungsanweisung.

Die Alternative besteht aus zwei bis drei Teilen:

1. Einer *Bedingung*.
2. Einem *Then-Zweig*.
3. Einem *Else-Zweig* (optional).

Die Bedingung ist ein boolescher Ausdruck. Dieser wird als erstes ausgewertet:

- Wenn er wahr ist, werden die Anweisungen im Then-Zweig befolgt.
- Wenn er falsch ist, werden die Anweisungen im Else-Zweig befolgt. (Falls dieser fehlt, geht es einfach mit dem nächsten Befehl weiter.)

11-5

Schreibweise der Alternative in Java.

```
if (Bedingung) {
  Then-Zweig
}
else {
  Else-Zweig
}
```

Sind Then-Zweig oder Else-Zweig *einzelne Anweisungen*, so kann man die geschweiften Klammern weglassen. Im Else-Zweig darf eine »einzelne Anweisung« auch wieder eine Steuerungsanweisung sein. Dann wird es aber unübersichtlich.

Der Algorithmus mit If-Then-Else in Java-Syntax. 11-6

Spezifikation

Bestimme Durchschnitt und Median der Zahlen a, b und c.

Algorithmus

```
average = (a + b + c) / 3;
if ((a <= b && a >= c) || (a >= b && a <= c)) {
  median = a;
}
else {
  if ((b <= a && b >= c) || (b >= a && b <= c)) {
    median = b;
  }
  else {
    median = c;
  }
}
```

11.1.2 While-Schleife

Die While-Steuerungsanweisung erlaubt es, eine Befehlsfolge wiederholt auszuführen. 11-7

Die While-Schleife besteht aus zwei Teilen:

1. Einer *Bedingung*.
2. Einem *Körper*.

Die Bedingung ist ein boolescher Ausdruck. Dieser wird bei jedem Schleifendurchlauf am Anfang ausgewertet. Wenn die Bedingung wahr ist, wird der Körper ausgewertet, danach wieder die Bedingung, dann wieder der Körper und so fort. Wenn die Bedingung falsch ist, wird hinter dem Körper mit dem nächsten Befehl weitergemacht.

Die Syntax der While-Schleife ist einfach. 11-8

```
while (Bedingung) {
  Körper
}
```

Wieder kann man die geschweiften Klammern weglassen, wenn der Körper eine einzelne Anweisung ist.

11-9 ✎ Zur Übung

Welchen Wert haben `sum` und `n` am Ende von folgendem Programm?

```
int n = 0;
int sum = 0;
while (n <= 100) {
  sum = sum + n;
  n = n + 1;
}
```

Wem das zu leicht ist: Welchen Wert hat `sum` am Ende, wenn die Bedingung `sum <= 100` lautet?

11.1.3 For-Schleife

11-10 **Die For-Steuerungsanweisung ist eine bequeme While-Schleife.**

Die For-Schleife besteht aus vier Teilen:

1. Einer *Initialisierung*,
2. einer *Bedingung*,
3. einem *Inkrement* und
4. einem *Körper*.

Am Anfang wird die Initialisierung ausgeführt. Dann wird die Bedingung getestet. Solange sie wahr ist, wird der Körper ausgeführt. Jedesmal, nachdem der Körper ausgeführt wurde, wird das Inkrement ausgeführt.

11-11 **Die Syntax der For-Schleife.**

```
for (Initialisierung; Bedingung; Inkrement) {
  Körper
}
```

Wieder kann man die geschweiften Klammern weglassen, wenn der Körper eine einzelne Anweisung ist.

11-12 **Die folgenden Programme haben den gleichen Effekt.**

```
int n = 0;
int sum = 0;
while (n <= 1000) {
  sum = sum + n;
  n = n + 1;
}
...
```

```
int n;
int sum = 0;
for (n = 0; n <= 1000; n = n + 1) {
  sum = sum + n;
}

...
```

11-13 ✎ Zur Übung

Folgender Algorithmus druckt alle Teiler einer Zahl n aus:

Algorithmus
Für alle Zahlen i zwischen 1 und n tue
  Falls n modulo i gleich 0 ist
    Gib i aus

Formulieren Sie den Algorithmus als Java-Programm. Zur Ausgabe eine Zahl können Sie `System.out.println(i)` verwenden.

11.2 Datentypen

11.2.1 Der Begriff des Typs

Daten haben natürlicherweise einen Typ. 11-14

Daten lassen sich leicht in Klassen von *ähnlichen Daten* einteilen. Beispiele sind: Zahlen (in verschiedenen Varianten), Texte, Bilder, Dateien, Daten, Koordinatenpunkte.
Solche Klassen nennen wir einen *Typ*: Ein *Datentyp* ist eine Menge von möglichen Werten. Hat beispielsweise ein Wert den Datentyp `int`, so weiß man, dass dieser Wert eine ganze Zahl ist.

11.2.2 Die Nutzen von Typen

Wozu dienen Datentypen? 11-15

Datentypen helfen *Menschen*, Ordnung in ein Programm zu bringen. Datentypen helfen *Übersetzern*, die richtige Menge Speicher für einen Wert zu reservieren. Datentypen helfen *Menschen und Übersetzern*, Fehler zu finden: Man kann ein Bild nicht quadrieren.

11.2.3 Arten von Typen

Die Typen von Typen. 11-16

Es gibt viele unterschiedliche Typen. Man kann sie grob in zwei Klassen einteilen:

- *Elementare Typen*
  Diese einfachen Datentypen sind fest eingebaut in den Übersetzer. Sie haben keine innere Struktur.
  Beispiel: `int`
- *Zusammengesetzte Typen*
  Solche Datentypen haben eine innere Struktur. Diese Datentypen können (und müssen) im Programm selbst erzeugt werden.
  Beispiel: Ein Datumstyp. Er besteht aus drei Zahlen.

11.2.4 Javas Datentypen

Liste der elementaren Datentypen von Java. 11-17

Datentyp	Bits	Wertebereich
`boolean`	1	`true` und `false`
`byte`	8	−128 bis 127
`short`	16	−32768 bis 32767
`int`	32	ca −2.000.000.000 bis 2.000.000.000
`long`	64	−ganz viel bis ganz viel
`float`	32	ca. 8 Dezimalstellen Genauigkeit
`double`	64	ca. 16 Dezimalstellen Genauigkeit
`char`	16	ein Unicode Zeichen wie `'x'`

11.3 Typisierung

11.3.1 Typisierung von Variablen

11-18

Das Konzept der Typisierung.

In *stark typisierten Programmiersprachen* hat jeder Wert, jeder Ausdruck und jede Variable einen bestimmten Typ. Durch die Typisierung weiß der Übersetzer immer, wie viel Speicher für die Variablen und Ausdrücke benötigt wird und ob die Ausdrücke überhaupt »typkorrekt« sind.

Wir wissen schon: In Java muss für jede Variable *vor ihrer ersten Benutzung* der Typ dem Übersetzer mitgeteilt werden. Dazu schreibt man `type var;` um anzudeuten, dass eine Variable `var` den Typ `type` haben soll.

11-19

Eine Bequemlichkeit in der Syntax.

Java erlaubt es, mehrere Variablen des gleichen Typs in einer Deklaration zu deklarieren und ihnen auch gleich einen Wert geben.

Beispiel 2. Die folgenden Programmtexte haben den gleichen Effekt:

```
1. int a = 5, b = 6, c;
2. int a;
   a = 5;
   int b;
   b = 6;
   int c;
```

11-20

✎ Zur Übung

Welche der folgenden Deklarationen sind korrekt?

```
1. char
   c = 'a';
2. char a,b;
3. int int;
4. int i, char j;
5. double x = 5;
6. boolean flag = 'a';
7. boolean flag = true;
8. boolean flag == y;
9. boolean flag = y == 5;
```

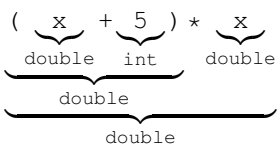
11.3.2 Typisierung von Ausdrücken

11-21

Jeder Ausdruck hat einen Typ.

Zur Erinnerung: *Ausdruck* ist eine Variable oder ein Wert (wie eine Zahl) oder die Verknüpfung von Ausdrücken mittels eines Operators (wie `x + 5`). In Java haben alle Ausdrücke und auch alle Teilausdrücke einen Typ.

Beispiel 3. Sei `x` als `double` deklariert (also `double x;`).

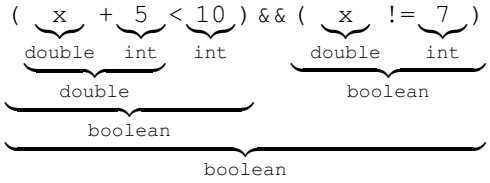


11-22

Wie bekommt der Ausdruck seinen Typ?

Die Typen der Teilausdrücke werden automatisch bestimmt.

Beispiel 4. Sei `x` als `double` deklariert.



11.3.3 Typfehler

Die zwei Arten von Fehlern in Programmen. 11-23

Es gibt zwei Hauptarten von Programmfehlern: Fehler zur Laufzeit (runtime errors) und Fehler zur Übersetzungszeit (compile time errors).

Beispiel: Runtime error

Ein Programm liest einen Wert in die Variable `x` von der Tastatur. Dann gibt es die Zuweisung `y = 5 / x;`. Falls nun eine Null gelesen wurde, so stürzt das Programm *jetzt* ab.

Beispiel: Compile time error

```
x = 5 +;
```

Typfehler sind Compile-time-errors. 11-24

- Ein *Typfehler* liegt vor, wenn man versucht, einer Variable einen Wert zuzuweisen, der den falschen Typ hat.
  Beispiel: `double x = true;`
- Ein *Typfehler* liegt auch vor, wenn man einen Operator falsch anwendet.
  Beispiel: `5 && x < 0`
- Ein *Typfehler* liegt auch vor, wenn ein anderer Typ als `boolean` in einer Bedingung benutzt wird.
  Beispiel: `if (42.4)`
- Die Erkennung von Typfehlern durch den Übersetzer ist eine große Hilfe bei der Erstellung großer Programme. Deshalb werden in großen Projekten in der Regel stark typisierte Sprachen verwendet.

Zusammenfassung dieses Kapitels

Java-Syntax: Die elementaren Datentypen 11-25

Die elementaren Typen in Java sind `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`. Jeder Ausdruck, jeder Wert, jede Variable hat in Java einen bestimmten Typ. Werden Typen falsch verwendet, so liegt ein *Typfehler* vor, was *während der Übersetzung* festgestellt wird.

Java-Syntax: Die Alternative

```
if (Bedingung) {
  Then-Zweig
}
else {
  Else-Zweig
}
```

Java-Syntax: Die While-Schleife

```
while (Bedingung) {
  Körper
}
```

Java-Syntax: Die For-Schleife

```
for (Initialisierung; Bedingung; Inkrement) {
  Körper
}
```

Übungen zu diesem Kapitel

Übung 11.1 Algorithmen in Java-Syntax formulieren, mittel

Formulieren Sie die Algorithmen der Übungen 9.11 und 9.12 in Java-Syntax. Deklarieren Sie die Variablen und geben Sie das Ergebnis mit `System.out.println(loesung);` aus.

Übung 11.2 Quersumme berechnen, mittel

Formulieren Sie den Quersummen-Algorithmus von Übung 9.13 in Java-Syntax.

Übung 11.3 Verschachtelte Schleifen üben, schwer

Die *Querwurzel* einer Zahl kann man berechnen, indem man so lange die Quersumme berechnet, bis nur noch eine Ziffer übrigbleibt. Beispiel: Die Querwurzel von 666 ist 9, da $6+6+6=18$ und $1+8=9$. Schreiben Sie ein Programm in Java-Syntax, dass die Querwurzel der Variable `n` berechnet. Lösen Sie dazu zuerst das Problem, die Quersumme einer Zahl zu berechnen (Übung 11.2). Diesen Algorithmus bauen Sie dann in eine While-Schleife ein. Hier ist der Pseudo-Code:

```
1 // Die Querwurzel von n soll berechnet werden
2 while n ≥ 10 do // Noch mehr als eine Ziffer
3   // Jetzt der Algorithmus, der die Quersumme von n berechnet
4   ...
5   // Jetzt steht in quersumme die Quersumme von n
6   // Tue nun so, als ob diese Quersumme die Eingabe gewesen waere
7   n ← quersumme
```

Übung 11.4 Quadratwurzel iterativ berechnen, mittel

Schreiben Sie ein Java-Programm, das die *ganzzahlige Wurzel* der Variablen n berechnet. Die ganzzahlige Wurzel $k = \lfloor\sqrt{n}\rfloor$ ist die größte natürliche Zahl k, so dass $k^2 \leq n$. Sie sollten hierbei *nicht* `Math.sqrt` benutzen.
Beispiele: $\lfloor\sqrt{23}\rfloor = 4$ und $\lfloor\sqrt{25}\rfloor = 5$
Ihr Programm sollte mit den folgenden Zeilen beginnen:

```
int n = 23; // Eingabewert
int wurzel; // Enthaelt am Ende des Programms
            // die ganzzahlige Wurzel von n
```

Übung 11.5 Der gute Stern, mittel

Finden Sie heraus, ob Ihr Leben unter einem guten Stern steht, indem Sie ein Java-Programm schreiben, das überprüft, ob die Quersumme Ihres Geburtsdatums gleich 42 ist.
Ihr Programm sollte mit den folgenden Zeilen beginnen:

```
int n = 29091984;      // Eingabewert
boolean bin_gesegnet; // Enthaelt am Ende des Programms "true",
                      // wenn die Quersumme von n gleich 42 ist,
                      // sonst "false"
```

Übung 11.6 Quadratwurzel überprüfen, mittel

Schreiben Sie ein Java-Programm, das überprüft, ob eine Zahl eine Quadratzahl ist (eine Zahl n ist genau dann eine Quadratzahl, wenn gilt $(\lfloor\sqrt{n}\rfloor)^2 = n$).
Ihr Programm sollte mit den folgenden Zeilen beginnen:

```
int n = 24;                // Eingabewert
boolean ist_quadratzahl;   // Enthaelt am Ende des Programms "true",
                           // wenn n Quadratzahl ist, sonst "false"
```

Übung 11.7 Anzahl der Teiler bestimmen, mittel

Ein Teiler einer Zahl n ist eine Zahl t, so dass sich n durch t ohne Rest teilen lässt. Anders gesprochen: Berechnet man n modulo t, so muss sich 0 ergeben (in Java-Syntax schreibt sich dies als `n % t == 0`).
Ein Programm soll die *Anzahl der Teiler einer Zahl n* bestimmen. Einen Algorithmus hierfür finden Sie in Übung 9.14 beschrieben. Setzen Sie den Algorithmus in ein Java-Programm um und probieren Sie es aus. Ermitteln Sie mit Hilfe Ihres Programms die Anzahl der Teiler der Zahlen 500, 1024 und 123456789.
Dokumentieren und formatieren Sie Ihr Programm sinnvoll und schön.

Übung 11.8 Einfacher Primzahltest, mittel

Eine Primzahl ist eine Zahl mit genau zwei Teilern. Verändern beziehungsweise erweitern Sie das Programm aus Übung 11.7, so dass es überprüft, ob eine Zahl eine Primzahl ist.

Übung 11.9 Primzahlen ausgeben, mittel

Gesucht ist ein Programm, das alle Primzahlen zwischen 1 und einer natürlichen Zahl n ausgibt. Der Algorithmus hierzu funktioniert wie folgt: In einer Schleife iteriert man über alle Zahlen i zwischen 1 und n. Für jede dieser Zahlen benutzt man den Algorithmus aus den vorherigen Aufgaben, um zu überprüfen, ob i eine Primzahl ist (hier hat man also eine Schleife in einer Schleife – achten Sie darauf, dass keine Namenskonflikte entstehen). Immer, wenn man feststellt, dass i eine Primzahl ist, gibt man i aus.

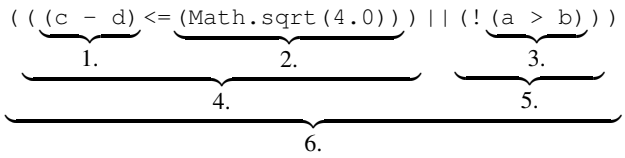
Übung 11.10 Goldbachvermütung überprüfen, schwer

Die Golbachvermutung besagt, dass alle geraden Zahlen ab 4 die Summe zweier Primzahlen sind. Schreiben Sie ein Programm, das die Goldbach-Vermutung für alle Zahlen bis 1000000 überprüft.

Prüfungsaufgaben zu diesem Kapitel

Übung 11.11 Typen bestimmen, leicht, original Klausuraufgabe, mit Lösung

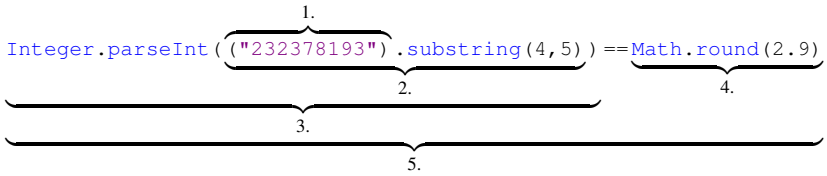
Geben Sie die Typen aller sechs geklammerten Teilausdrücke des folgenden Ausdrucks an. Es gelten dabei die Variablendeklarationen `double c,d;` und `int a,b;`.



Tipp: `Math.sqrt` berechnet die Quadratwurzel ihres Arguments und gibt immer einen `double` zurück.

Übung 11.12 Datentypen bestimmen, leicht, original Klausuraufgabe, mit Lösung

Bestimmen Sie die Datentypen der nummerierten Teilausdrücke des folgenden Java-Ausdrucks.



Die Programmierschnittstellen der verwendeten Methoden lauten wie folgt:

```
class String{
  String substring( int beginIndex, int endIndex );
}
class Integer{
  static int parseInt( String s );
}
class Math{
  static double round( double z );
}
```

Bestimmen Sie die Datentypen der nummerierten Teilausdrücke des obigen Java-Ausdrucks.

12-1

Kapitel 12
Zeichenketten
Vom Papyrus zur DNS-Sequenz

12-2

Lernziele dieses Kapitels

1. Zeichenketten (Strings) als Datentypen kennen
2. Einfache Stringalgorithmen kennen und implementieren können
3. Einen fortgeschrittenen Stringalgorithmus kennen

Inhalte dieses Kapitels

12.1 Zeichenketten 101
12.1.1 Der Begriff der Zeichenkette 101
12.1.2 Der Datentyp String 101
12.1.3 Die Datenstruktur String 102

12.2 Algorithmen auf Zeichenketten 103
12.2.1 Umdrehen 103
12.2.2 Trimmen 104
12.2.3 Naives Suchen 104
12.2.4 Intelligentes Suchen 105

Übungen zu diesem Kapitel 106

Worum es heute geht

Die Erfindung der Schrift ist auch an der Informatik nicht spurlos vorbeigegangen. Eigentlich wollen Computer, wie der Name schon sagt, rechnen und darin sind sie auch ziemlich gut. Jedoch bequemen sie sich auch durchaus dazu, mit Texten umzugehen.

Computer verlangen Texte jedoch in etwas »normierterer« Form, als wir Menschen sie gewohnt sind. Aus Sicht des Computers (genauer in unserem Fall aus Sicht von Java) sind Texte nicht anderes als *lange Folgen von Zeichen* oder kurz *Zeichenketten* (englisch *strings*). Diese Zeichenketten sind »eindimensional«, Zeilenumbrüche werden einfach durch spezielle Zeichen kodiert. Da neben Zeilenumbrüchen und Leerzeichen auch aller möglicher anderer Kram Teil einer Zeichenkette sein kann, braucht man ein Verfahren um klarzumachen, wo eine Zeichenkette beginnt und wo sie endet. Dies ist in Java ganz einfach: Zeichenkette beginnen und enden mit einem doppelten senkrechten Anführungszeichen (`"`). So ist `"Hallo␣Welt"` eine Zeichenkette mit zehn Zeichen.

Welche Zeichen können in einer Zeichenkette vorkommen? Nun, dies sind wahrlich viele, nämlich alle im Unicode aufgeschriebenen. Darin finden sich ganz langweilige Zeichen wie LATIN CAPITAL LETTER A (A); nur wenig spannendere wie AT SIGN (@); vom Namen her originelle wie LATIN SMALL LETTER TURNED H WITH FISHHOOK AND TAIL oder HEAVY TEARDROP SPOKED PINWHEEL ASTERISK; bis zu wirklich spannenden wie DOUBLE MALE SIGN (⚣) oder ARABIC LIGATURE SALLALLAHOU ALAYHE WASALLAM (ﷺ).

In Java ist die Verarbeitung von Zeichenketten, wie eigentlich in fast alle anderen Programmiersprachen auch, priviligiert. Es gibt besondere Befehle für ihre Verarbeitung und viele Sachen, die man für andere Typen nicht machen darf, sind für Zeichenketten erlaubt.

12.1 Zeichenketten

12.1.1 Der Begriff der Zeichenkette

Zeichen umgeben uns ständig.

12-4

Wir leben in einer Welt, in der Schrift allgegenwärtig ist:

✎ Zur Diskussion

- Wie viele Zeichen sind in einem Umkreis von 10 Metern auf Oberflächen gedruckt?
- Was ist die maximale Entfernung, die sie in Ihrem Leben jemals von Schriftzeichen entfernt waren?

Zeichenketten sind Folgen von Zeichen.

12-5

▶ Definition: Zeichenkette, Alphabet

Eine *Zeichenkette* ist eine Folge von Zeichen. Die Zeichen sind Elemente eines festen *Alphabets*.

Beispiel

- Eine Zeichenkette von vier Zeichen über dem Alphabet ASCII: `ABCD`
- Eine Zeichenkette von fünf Zeichen über dem Alphabet UNICODE: `ABϒ++`
- Eine Zeichenkette über dem genetischen Alphabet: `ACGTTACC`

Auch Leerzeichen sind Zeichen.

12-6

Auch Leerzeichen sind Zeichen. Deshalb ist es nützlich, Zeichenketten in Hochkommas einzuschließen: `"Hallo␣␣"` hat 7 Zeichen (zwei Leerzeichen). `"␣"` hat 1 Zeichen (ein Leerzeichen). `""` hat 0 Zeichen.
Auch Zeilenumbrüche sind Zeichen. Dadurch lassen sich auch »zweidimensionale« Texte als Zeichenketten auffassen:

Zweidimensionaler Text . . .

```
1 Sehr geehrte Damen und Herren,
2
3 hiermit ...
```

. . . als Zeichenkette

`Sehr geehrte Damen und Herren,¶¶hiermit ...`

Besonderheiten in Java.

12-7

Strings werden in doppelte Anführungszeichen eingeschlossen. Um einen Zeilenumbruch einzugeben, wird die Zeichenfolge `\n` (»n« wei »newline«) benutzt:

```
Sehr geehrte Damen und Herren,\n\nhiermit
```

Um Anführungszeichen einzugeben, kann man die Zeichenfolge `\"` verwenden. Um einen Backslash einzugeben, kann man die Zeichenfolge `\\` verwenden.

12.1.2 Der Datentyp String

Strings sind ein Datentyp in Java.

12-8

Zeichenkette bilden in Java einen eigenene Datentyp mit dem Namen *`String`*. Dieser Datentyp ist zwar kein elementarer Datentyp, er kann aber so verwendet werden:

```
String a = "hallo";
String b = "hal";
String c = "lo";
String d = b + c; // + ist die Verkettung

System.out.println(a + "␣und␣" + d + "␣sind␣gleich");
```

Man kann *nicht* einzelne Zeichen eines `String` verändern.

12-9 **Bequemlichkeiten in der Syntax.**
Man kann nicht nur Strings mittels + verketten, sondern auch Strings und andere Dinge. Die »anderen Dinge« werden dann automatisch in einen String verwandelt:

```
int i = 2+3;

System.out.println ("Fünf␣ist␣gleich␣" + i + ".");

// Gibt "Fünf ist gleich 5." aus
```

12.1.3 Die Datenstruktur String

12-10 **Datenstrukturen sind Datentypen zusammen mit ihren Operationen.**
Zur Erinnerung:

▶ Definition: Datentyp
Ein *Datentyp* ist eine Menge von Werten.

Neu:

▶ Definition: Datenstruktur
Eine *Datenstruktur* ist ein Datentyp zusammen mit Operationen, die man auf Werten dieses Typs ausführen kann.

12-11 **Beispiele von Datenstrukturen.**

Beispiel: Der Datentyp `byte`
Der *Datentyp* besteht aus

1. den Werten von −128 bis 127.

Beispiel: Die Datenstruktur `byte`
Die *Datenstruktur* `byte` besteht aus

1. den Werten von −128 bis 127 und
2. den Operationen wie +, ⋆, usw.

12-12 **Zeichenketten als Datenstrukturen.**

Die Datenstruktur `String`
Die *Datenstruktur* `String` besteht aus

1. allen Unicode-Zeichenketten als Werten,
2. der Operation + zum Verketten,
3. der Operation `equals` zum Vergleichen von Strings (*nicht* ==),
4. der Operation `length` zum Bestimmen der Länge,
5. der Operation `charAt` zur Bestimmung der Zeichens an der *i*-ten Stelle.

Es gibt noch vielen weiteren Operationen, siehe dazu http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html.

12-13 **Allgemeine Syntax für Operationen auf Strings.**
Die Syntax aller neuen Operationen lautet:

$$\underbrace{\texttt{result}}_{\text{Name der String-Variable}} \;.\; \underbrace{\texttt{charAt}}_{\text{Operationsname}} \;(\; \underbrace{\texttt{5}}_{\text{Parameter}} \;)$$

Selbst wenn es keine Parameter gibt, müssen die Klammern geschrieben werden.

Syntax für die neuen Operationen auf Strings.

12-14

Syntax

Das i-te Zeichen eines Strings lässt sich wie folgt bestimmen:

```
String s = "hallo";
System.out.println(s.charAt(0)); // Gibt 'h' aus
System.out.println(s.charAt(1)); // Gibt 'a' aus

if (s.charAt(2) != s.charAt(3)) {
  System.out.println("Kann␣nicht␣sein.");
}
```

Achtung: Die Zählung beginnt bei Null!

Syntax

Die Länge eines Strings lässt sich wie folgt bestimmen:

```
String s = "hallo";

if (s.length() == 5) {
  System.out.println("Alles␣wird␣gut");
}
```

Syntax für die Vergleichsoperation auf Strings.

12-15

Syntax

Zwei Strings lassen sich wie folgt vergleichen:

```
String s1 = "Hallo";
String s2 = "Welt";
String s3 = "Hal" + "lo";

if (s1.equals (s2)) {
  System.out.println("Quatsch");
}

if (s1.equals (s3)) {
  System.out.println("Schon␣eher.");
}
```

12.2 Algorithmen auf Zeichenketten

12.2.1 Umdrehen

Ein Algorithmus zum Umdrehen eines Strings.

12-16

```
String s      = "Dreh␣mich␣um!";

// Der folgende Algorithmus soll s umdrehen.

String result = "";

for (int i = s.length() - 1; i >= 0; i = i-1) {
  result = result + s.charAt(i);
}

s = result;

// jetzt ist s gerade "!mu hcim herD"
```

12-17

✎ Zur Übung

Geben Sie ein Java-Programm an, das das erste Zeichen eines Strings löscht. Ein Algorithmus hierzu: Beginnend mit einem leeren String `result` werden nacheinander alle Zeichen in `s` ab dem zweiten Zeichen an `result` angefügt.

```
String s      = "WWeg␣damit!";
String result = "";

... // Ihr Programm

s = result;

// jetzt ist s gerade "Weg damit!"
```

12.2.2 Trimmen

12-18

Ein Algorithmus zum Trimmen eines Strings.

```
String s      = "␣␣trimm␣mich!";

// Der folgende Algorithmus soll alle Leerzeichen
// am Anfang von s entfernen.

String result = "";
int    start = 0;

while (start < s.length () && s.charAt(start) == '␣') {
  start = start + 1;
}

for (int i = start; i < s.length(); i = i+1) {
  result = result + s.charAt(i);
}

s = result;

// jetzt ist s gerade "trimm mich!"
```

12.2.3 Naives Suchen

12-19

Naives Suchen.

```
String s         = "Sehr␣geehrte␣Damen␣und␣Herren";
String such_mich = "und";

// Folgender Algorithmus soll ermitteln, ob such_mich in s
// vorkommt.

boolean gefunden = false;

for (int i = 0;
     i <= s.length () - such_mich.length (); i = i+1)
{
  boolean ist_gleich = true;
  for (int j = 0; j < such_mich.length (); j = j+1) {
    if (s.charAt(i+j) != such_mich.charAt(j)) {
      ist_gleich = false;
    }
  }

  if (ist_gleich) {
    gefunden=true;
  }
}
```

12.2.4 Intelligentes Suchen

Die Idee hinter dem (vereinfachten) Boyer-Moore-Algorithmus.

12-20

Problem des naiven Suchens

Suchen eines Strings der Länge m in einem Text der Länge n dauert $n \cdot m$ Schritte.

Idee

Vergleich am Ende beginnen!

```
s == Sehr geehrte Damen und Herren
     und
        und
           und
              und
                 und
                    und
                     und
                        und
```

Der vereinfachte Boyer-Moore-Algorithmus

12-21

Input: Ein String `s` und ein Suchstring `such_mich`.
Frage: Ist `such_mich` als Teilstring in `s` enthalten?

Solange noch nicht gefunden tue:

1. Vergleich `such_mich` im aktuellen Fenster von hinten mit `s`.
2. Falls nicht gleich:
   - Betrachte letztes Zeichen von `s` im Fenster, an dem sich `such_mich` und `s` unterscheiden.
   - Schiebe das Fenster so weit vor, dass das letzte Vorkommen dieses Zeichens in dem String `such_mich` auf diesem Zeichen liegt.

Zusammenfassung dieses Kapitels

Alphabete, Wörter und Strings

12-22

Ein *Alphabet* ist eine nichtleere endliche Menge von *Symbolen* (auch *Buchstaben* genannt). Ein *Wort* ist eine (endliche) Folge von Symbolen. In Programmiersprachen heißen Wörter *Strings*.

Die Datenstruktur `String`

Eine *Datenstruktur* besteht aus eine *Datentyp* zusammen mit *Operationen*.
Bei Strings: Der Datentyp `String` enthält alle Worte über dem Unicode. Die wichtigsten Operationen sind die *Verkettung (+)* sowie

- `s.equals(t)` zum Vergleichen von Strings `s` und `t`,
- `s.length()` zum Bestimmen der Länge von `s` und
- `s.charAt(i)` zur Bestimmung des Zeichens an der i-ten Stelle von `s`.

Schleifen über Strings

Will man in einem String für alle Zeichen »etwas tun«, so benutzt man dazu eine Schleife:

```
String s = ...;
for (int i = 0; i < s.length(); i = i+1) {
  ... // tue etwas mit s.charAt(i)
}
```

Naive Suche

Bei der *naiven Suche* benutzt man (grob) obige Schleife, um für jede Stelle eines Strings zu testen, ob dort ein anderer String beginnt.

Übungen zu diesem Kapitel

Übung 12.1 Stringverarbeitung üben, mittel

Schreiben Sie ein Programm, das zu einer Nukleotidsequenz die komplementäre Sequenz ausgibt:

```
String sequenz    = "tcctat";
String komplement = "";  // enthaelt am Ende die komplementaere
                         // Sequenz, hier z.B. aggata
```

Prüfungsaufgaben zu diesem Kapitel

Übung 12.2 Zeichen zählen, leicht, typische Klausuraufgabe, mit Lösung

Schreiben Sie eine Methode `countchar`, die die Anzahl der Vorkommen eines Zeichens `c` in einem String `s` zählt und zurückgibt! Zum Beispiel:

Eingabe `s`	Eingabe `c`	Ergebnis
`"INFORMATIK"`	`'I'`	2
`"dampfschifffahrt"`	`'f'`	4
`"Achtung"`	`'k'`	0

```
int countChar( String s, char c) {
  /* Ihr Code hier */
}
```

Übung 12.3 Verschlüsselung, mittel, original Klausuraufgabe, mit Lösung

Erkennen Sie das Wort `"NIOFMRTAKI"`? Es handelt sich um eine verschlüsselte Version des Wortes `"INFORMATIK"`. Die Verschlüsselung funktioniert wie folgt: Die ersten beiden Buchstaben werden vertauscht, ebenso die nächsten beiden, dann wieder die darauf folgenden uns so weiter. Hier einige Beispiele:

Eingabe `s`	Ergebnis
`"JULI"`	`"UJIL"`
`"Computer"`	`"oCpmture"`
`"Johannes"`	`"oJahnnse"`

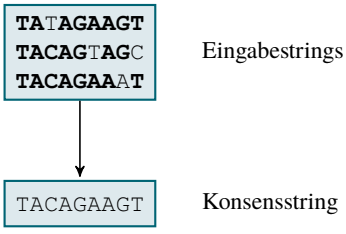
Schreiben Sie eine Methode, die diese Verschlüsselung durchführt! Gehen Sie der Einfachheit halber davon aus, dass die eingegebene Zeichenkette `s` eine gerade Länge hat.

```
String encrypt( String s ) {
  /* Ihr Code hier */
}
```

Übung 12.4 Primerdesign, schwer, original Klausuraufgabe

Für den Entwurf von Primern ist es notwendig, kurze DNA-Sequenzen zu bestimmen, die einer Menge von Eingabesequenzen möglichst ähnlich sind. Eine Möglichkeit, solche Sequenzen zu ermitteln, besteht in der Berechnung sogenannter »Konsensstrings«: Gegeben eine Menge gleichlanger Sequenzen (Eingabestrings), wird Stelle für Stelle das Nukleotid bestimmt, welches an dieser Stelle am häufigsten vorkommt. Die Aneinanderreihung dieser »Mehrheitsnukleotide« ergibt den Konsensstring. Die komplementäre DNA zum Konsensstring kann dann unter Umständen als Primer verwendet verden.

Die folgende Abbildung erläutert dieses Prinzip, wobei die Mehrheitsnukleotide in den Eingabestrings fett gesetzt sind:



Ergänzen Sie den Code der folgenden Methode, die den Konsensstring für ein gegebenes Array gleichlanger Sequenzen berechnet! Falls es an einer Stelle nicht eindeutig ist, welches Nukleotid am häufigsten vorkommt (weil zum Beispiel sowohl `T` als auch `A` jeweils 5 mal vorkommen), dann entscheiden Sie sich für einen beliebigen »Gewinner«.

```
String consensusSequence( String[] sequences ){
  if( sequences.length == 0 ){
      return "";
  }
  String consensus = "";

  // Ihr Code hier.

  return consensus;
}
```

Übung 12.5 Wörter zählen, mittel, original Klausuraufgabe

Gegeben sei ein String, der aus mehreren durch Leerzeichen getrennten Wörtern besteht, wie zum Beispiel `"Der␣␣␣Fuchs␣␣␣sprang␣über␣den␣See"`. Ergänzen Sie den Code der folgenden Methode, die die Anzahl der Wörter im String `s` berechnen und zurückgeben soll.

Sie dürfen davon ausgehen, dass der String außer Wörtern und Leerzeichen keine weiteren Zeichen, wie Punkte oder Kommata, enthält. Beachten Sie aber, dass zwischen zwei Wörtern mehrere Leerzeichen stehen können, wie zum Bespiel oben zwischen »Fuchs« und »sprang«. Am Anfang und am Ende des Strings können ebenfalls Leerzeichen vorkommen.

```
int countWords( String s ){
  int counter = 0;

  // Ihr Code hier.

  return counter;
}
```

Übung 12.6 Iteration über einen String, leicht, original Klausuraufgabe, mit Lösung

Ergänzen Sie den Code der untenstehenden Methode `isRNAString`, so dass überprüft wird, ob es sich bei dem String `s` um einen String handelt, der nur aus den Buchstaben a, c, g, u, A, C, G und U besteht. In diesem Fall soll **`true`** zurückgegeben werden, sonst **`false`**.

Hinweis: Sie können Schreibarbeit sparen mit der Methode **`public`** `String toLowerCase()` der `String`-Klasse. Diese Methode wandelt einen String in Kleinbuchstaben um und gibt den umgewandelten String zurück.

```
static boolean isRNAString( String s ){
  /* Fügen Sie hier Ihren Code ein */
}
```

13-1

Kapitel 13
Arrays (Felder)
Nummerierte Bankschließfächer

13-2

Lernziele dieses Kapitels

1. Konzept des Arrays verstehen.
2. Java-Syntax von Arrays beherrschen.
3. Java-Programme mit Array-Nutzung implementieren können.

Inhalte dieses Kapitels

13.1 Benutzung von Arrays 109
13.1.1 Einführung . . . 109
13.1.2 Array-Typen . . . 110
13.1.3 Erzeugung von Arrays . . . 110
13.1.4 Zugriff auf Arrays . . . 111
13.1.5 Algorithmen auf Arrays . . . 111

13.2 Speicherung von Arrays 112
13.2.1 Verweistypen . . . 112
13.2.2 Zuweisung und Vergleich von Verweistypen . . . 113

Übungen zu diesem Kapitel 114

Worum es heute geht

Verglichen mit einem einzelnen Zeichen (einem `char`) ist ein String wesentlich aufregender. Mit einzelnen Zeichen kann man nicht sonderlich viel anfangen: Man kann sie ausgeben, sie einlesen und wenn man möchte auch vergleichen, dann hört es aber auch schon wieder auf. Wie viel mehr ist mit Strings möglich: Man kann in ihnen suchen, sie umdrehen, zerhacken, zusammenfügen, durcheinanderwirbeln, verschlüsseln, trimmen, vergleichen, einlesen, ausgeben, verrücken und noch vieles mehr. Zeichen werden eigentlich überhaupt erst interessant, wenn man sie zu ganzen Ketten zusammenstellt.

Wie steht es nun mit Zahlen (einem `int` oder `float`)? Mit einer einzelnen Zahl kann man schon wesentlich mehr anstellen als mit einem einzelnen Zeichen. Bekanntermaßen kümmert sich ein ganzer Teilzweig der Mathematik, die Zahlentheorie, liebevoll um die vielfältigen Eigenschaften von einzelnen Zahlen. Wie viel aufregender müssen die Dinge dann erst werden, wenn wir Zahlen zu Ketten zusammenfügen? Eine solche »Kette von Zahlen« werden wir im Folgenden einen *Array* nennen, genaugenommen einen *Integer-Array*.

Arrays kennen Sie schon, auch wenn Ihnen das noch niemand verraten hat: Das, was Sie in der Mathematik als *Vektor* kennengelernt haben, ist aus Sicht der Informatik ein Array von Zahlen. Das, was Sie in der Mathematik als *Matrix* kennengelernt haben, ist aus Sicht der Informatik ein zweidimensionaler Array von Zahlen, genauer ein Array von Arrays von Zahlen.

Arrays kann man nicht nur von Zahlen bilden. Allgemein kann man für *jeden* Typ einen Array von Elementen dieses Typs bilden. Dieser enthält dann ein erstes Element, ein zweites Element, ein drittes Element und so weiter bis zu einem letzten, n-ten Element, wobei n die *Länge* des Arrays bezeichnet.

In der Mathematik würde man die, sagen wir, dritte Komponente eines Vektors x typischerweise mit x_3 bezeichnen. Da tiefgestellte Indizes auf herkömmlichen Tastaturen etwas schwierig einzugeben sind, benutzt man in der Informatik typischerweise eine andere Notation: Man stellt den Index in eckige Klammern hinter den Array, also `x[2]`. Dies ist kein Tippfehler: Das *dritte* Element des Arrays erhält man über den Index 2, denn die Zählung

beginnt bei 0. Es ist also `x[0]` das *erste* Element des Arrays und `x[n-1]` das *letzte* Element, wenn `x` die Länge `n` hat.

Auf den ersten Blick scheint es etwas merkwürdig zu sein, dass die Arrayindizierung in Java bei 0 statt bei 1 beginnt. Dieser Eindruck täuscht: In Wirklichkeit ist dies nicht nur eine kleine Merkwürdigkeit sondern eine noch viel diabolischere Gemeinheit als der Umstand, dass Zuweisungen durch ein einfaches Gleichheitszeichen geschrieben werden. Sollten Sie aus Versehen schreiben `if(a=b)`, obwohl Sie `if(a==b)` meinen, so wird der Übersetzer Ihnen dies mitteilen. Sollten Sie aus Versehen versuchen, mittels `x[10]` statt mit `x[9]` auf das zehnte Element eines Arrays zugreifen, so merken Sie das erst, wenn Ihr Programm schon längst läuft und schlimmstenfalls schon beim Kunden im Einsatz ist – also viel zu spät, um noch etwas zu ändern.

13.1 Benutzung von Arrays

13.1.1 Einführung

Die Probleme einer Schweizer Bank. 13-4

Eine Schweizer Bank möchte mit einem Programm den Kontostand von Nummernkonten verwalten:

Kontonummer	Betrag
1	500 SFr
2	−20 SFr
3	230000000 SFr
4	−500 SFr
…	…
500	23423523 SFr

Wie sollte dies in Java abgebildet werden?

Erste mögliche Implementierung. 13-5

```
double konto1   = 500;
double konto2   = -50;
double konto3   = 230000000;
double konto4   = -500;
// ...
double konto500 = 23423523;
```

Diese Implementierung ist aber *schlecht*, da wir beispielsweise *jedesmal das Programm ändern müssten, wenn es einen neuen Kunden gibt.*

Zweite mögliche Implementierung. 13-6

```
double[] kontos; ...

kontos [1]   = 500;
kontos [2]   = -50;
// ...
kontos [500] = 23423523;
```

Nun ist folgendes möglich:

```
double bilanz_summe = 0;
for (int i = 1; i <= 500; i = i+1) {
  bilanz_summe = bilanz_summe + kontos[i];
}
```

13-7

Was sind Arrays?

Arrays (Felder) entsprechen indizierten Variablen in der Mathematik (x_i). Sie halten ein *feste* Anzahl von Werten, die alle *gleichen Typ* haben müssen. Arrays liegen als »Block« irgendwo im Speicher, alle Werte hintereinander weg.

Beispiel

Die Schweizer Bank würde einen Array `kontos` von `double`-Werten benutzen. Man beachte: `kontos` ist *eine* Variable, die gewissenmaßen eine »interne Struktur« hat.

Beispiel

Eine Moleküldatenbank könnte einen Array benutzen, in der jeder Eintrag ein Molekül ist.

13.1.2 Array-Typen

13-8

Was sind Array-Typen?

Hat man einen beliebigen Typ `type` gegeben, so ist `type[]` der Typ eines *Arrays von `type`-Werten*. Nun kann man eine Variable dieses Typs deklarieren: Beispiel: `double[] konto;`

Der Array-*Typ* legt die Größe des Arrays *nicht* fest. Eine Array-Variable kann Arrays beliebiger Größe aufnehmen. Jeder *konkrete* Array hat aber eine *feste* Größe.

Analogie: Der Typ `String` legt auch die Länge der Zeichenkette nicht fest, jede konkrete Zeichenkette hat aber eine feste, unveränderliche Länge.

13.1.3 Erzeugung von Arrays

13-9

Lebenszyklus eines Arrays

- Man kann einen Array auf zwei Arten *erzeugen*. Der erzeugte Array hat dann eine *feste, unveränderliche Größe*.
- Dann können Arrays *benutzt* werden.
- Werden sie nicht mehr gebraucht, werden sie *automatisch gelöscht*.

Will man die Größe eines Arrays *ändern*, so muss man einen neuen Array der gewünschten Größe erzeugen und dann die Elemente aus dem alten Array in den neuen Array kopieren.

13-10

Wie erzeugt man neue Arrays?

Erste Methode

Bei der Deklaration einer Array-Variable darf man mittels einer speziellen Notation direkt einen Array angeben:

```
double[] konto = {50, 5000, -200, 10};
// Jetzt enthält die Variable konto
// einen Array der Länge 4
```

Zweite Methode

Man erzeugt einen leeren Array einer bestimmten Größe mittels einer anderen speziellen Notation:

```
double[] konto = new double[4];
// Jetzt enthält die Variable konto
// einen Array der Länge 4
```

13.1.4 Zugriff auf Arrays

Wie greift man auf Array zu? 13-11

Hat eine Variable `var` den Typ `type[]`, so kann man mittels `var[5]` auf das sechste (!) Element zugreifen: Die Zählung fängt nämlich wie bei Strings bei 0 an. Zugriff außerhalb der Größe des Arrays führt zum Absturz:

```
int[] werte = new int[1000];

// Beliebter Anfängerfehler:
for (int i = 1; i <= 1000; i = i+1) {
  werte[i] = 0;

  // Tausendmal berührt,
  // Tausendmal ist nichts passiert,
  // Tausend und eine Nacht,
  // Da hat es ArrayOutOfBounds gemacht.
}
```

Noch ein paar Stolpersteine. 13-12

- Die Operationen + und `charAt` für Strings gibt es *nicht* für Arrays.
- Für eine Array-Variable `var` liefert aber *`var.length`* die Größe des Arrays. Anders als für Strings dürfen aber *keine Klammern* dahinter gesetzt werden (ja, das *ist* vollkommen unlogisch – life is hard).

13.1.5 Algorithmen auf Arrays

Berechnung des Skalarprodukts zweier Vektoren. 13-13

```
double[] vector1 = {3, 5, 6, 10};
double[] vector2 = {-0.5, -1, 5, 10};

// Berechne das Skalarprodukt

double scalar_product = 0;

for (int i=0;
     i < vector1.length && i < vector2.length;
     i = i+1) {
  scalar_product = scalar_product +
                   vector1[i] * vector2[i];
}
```

Umdrehen eines Arrays. 13-14

```
double[] vec = new double[1000];

// ... vec wird gefüllt

// Drehe vec um
for (int i=0; i<vec.length/2; i = i+1)
{
  // Vertausche vec[i] und vec[vec.length-i-1]
  double temp          = vec[i];
  vec[i]               = vec[vec.length-i-1];
  vec[vec.length-i-1] = temp;
}
```

13-15

✎ Zur Übung

Geben Sie ein Programm mit einer For-Schleife an, das zwei Arrays verkettet.
Es sollen in `z` zuerst gerade die Werte aus `a1` kommen, gefolgt von den Werten aus `a2`.

```
// Zwei Arrays
char[] a1 = {'h', 'a', 'l'}; // oder etwas anders
char[] a2 = {'l', 'o'};      // oder etwas anders

// Der Array, in den die Verkettung hinein soll:
char[] z  = new char [a1.length+a2.length];

// Ihr Programm:
...
```

13.2 Speicherung von Arrays

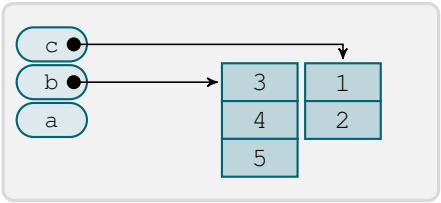
13.2.1 Verweistypen

13-16

Wohin mit dem Array?

Bei einem Array kann der Übersetzer offenbar nicht den Speicherbedarf vorher bestimmen.
Deshalb reserviert der Übersetzer lediglich den Platz für einen *Verweis*.

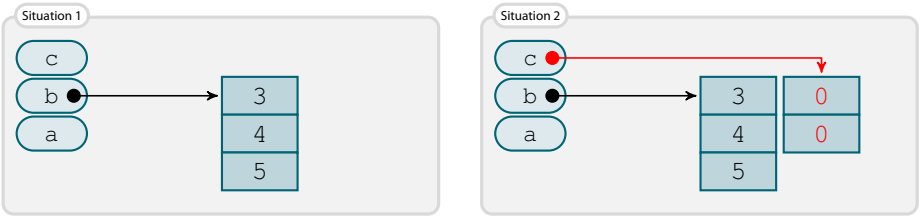
```
int   a;
int[] b = {3,4,5};
int[] c = {1,2};
```



13-17

Erzeugung eines neuen Arrays mittels `new`.

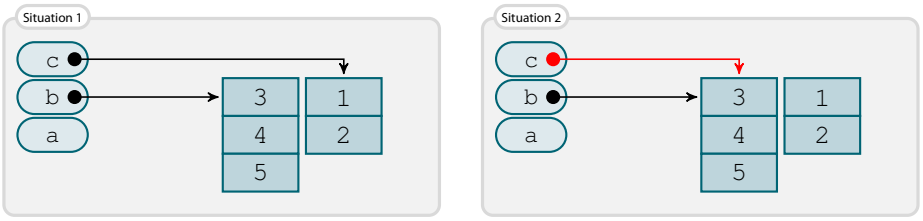
```
int   a;
int[] b = {3,4,5};
int[] c;               // Situation 1
c = new int[2];        // Situation 2
```



13.2.2 Zuweisung und Vergleich von Verweistypen

Zuweisung von Arrays. 13-18

```
int   a;
int[] b = {3,4,5};
int[] c = {1,2};       // Situation 1
c = b;                 // Situation 2
```



Es gibt unerwartete Effekte bei Zuweisungen von Arrays. 13-19

Wenn man mittels `b = c` einer Arrayvariable `b` den in einer anderen Arrayvariable `c` gespeicherten zuweist, so *verweisen `b` und `c` auf denselben Array*. Ändert man dann `b`, so ändert man auch gleichzeitig `c`, was man meistens nicht will.
Vergleicht man Arrayvariablen mittels ==, so wird lediglich überprüft, ob die Arrayvariablen auf denselben Array verweisen, und *nicht, ob die Arrays dieselben Elemente haben*; und auch dies will man meistens nicht.

Moral

1. Zuweisung von Arrays sind mit Vorsicht zu genießen.
2. Vergleiche von Arrays sind mit Vorsicht zu genießen.

Zusammenfassung dieses Kapitels

Der Array-Typ 13-20

Ist `type` ein Typ, so ist `type[]` der Typ eines *Arrays von Werten vom Typ `type`*. In einer Variable vom Typ `type[]` können Arrays *beliebiger Größe* gespeichert werden, jedoch hat jeder *einzelne Array* eine feste, unabänderliche Größe.

Vergleich von Arrays und Strings

Gemeinsamkeiten:

- Beide speichern Folgen von Elementen gleichen Typs (bei Strings **`char`**s, bei Arrays beliebiger Typ).
- Jeder einzelne String/Array hat eine feste Größe.
- Die Zählung der Elemente beginnt bei 0.

Unterschiede:

- Bei Arrays kann man einzelne Elemente *ändern*, bei Strings nicht.
- Man kann Strings mittelt + verketten, Arrays nicht.
- Bei Arrays schreibt man `a.length` für die Länge, bei Strings `a.length()`.

Arrays kann man auf zwei Arten erzeugen

- `int[] a = { 0, 0, 0, 0 };`
- `int[] a = new int [4];`

Über Arrays *iteriert* man mit Schleifen

```
int[] a = ...;
for (int i = 0; i < a.length; i = i+1) {
  ... // tue etwas mit a[i]
}
```

Vergleich von und Zuweisung von Arrays

Die Befehle `a = b;` und `a == b` machen für Arrays *nicht* das, was man erwartet.

Übungen zu diesem Kapitel

Übung 13.1 Arrays statt Strings, einfach

Schreiben Sie ein Programm analog zu Übung 12.1, welches die komplementäre Sequenz zu einer Basenfolge berechnet, nur dass die Sequenzen nun als `char`-Array statt einem String gespeichert wird:

```
char[] sequenz    = {'t','c','c','t','a','t'};
char[] komplement = new char[sequenz.length];
```

Um das Array auszugeben, müssen Sie in einer `for`-Schleife jedes einzelne Zeichen ausgeben.

Übung 13.2 Arrays verarbeiten, mittel

Schreiben Sie ein Programm, das zu einem Array von Zahlen das Minimum, das Maximum und den Mittelwert ermittelt. Testen Sie Ihr Programm mit folgendem Array:

```
int[] numbers = {3, 0, 610, 4181, 1, 89, 377, 13, 34, 2584, 1, 1597, 144,
    233, 21, 55, 987, 5, 8, 17711, 6765, 28657, 2, 10946};
```

Übung 13.3 Codons dekodieren, mittel

Schreiben Sie ein Programm, das zu einer Nukleotidsequenz die Sequenz der dadurch kodierten Aminosäuren ausgibt, zum Beispiel als 1-Letter-Codes.

Eine einfache, aber längliche und daher mit viel Tipparbeit verbundene Möglichkeit, diese Aufgabe zu lösen, hat etwas mit 64 `if`-Anweisungen in einer `for`-Schleife zu tun. Etwas eleganter geht es so:

- Codons kann man auf Zahlen von 0 bis 63 abbilden. Dazu weist man jeder Base eine Zahl von 0 bis 3 zu (z.B. $\mathrm{T}=0, \mathrm{C}=1, \mathrm{A}=2, \mathrm{G}=3$). Den Wert der zweiten Base multipliziert man mit 4, den Wert der dritten Base mit 16 und addiert die drei Werte. ATG $(\mathbf{2}, \mathbf{0}, \mathbf{3})$ ergibt sich beispielsweise zu $\mathbf{2}+4\cdot\mathbf{0}+16\cdot\mathbf{3}=50$.
- Die Zuordnung zwischen der Zahlendarstellung des Codons und der kodierten Aminosäure kann man gut als Array speichern.

Übung 13.4 Arrays von Strings verstehen, leicht

Eine Reihe verwandter DNA-Sequenzen sei als *Array von Strings* gegeben:

```
String[] j_region = { "ttatgtcttcggaactgggaccaaggtcaccgtcctag",
                      "tgtggtattcggcggagggaccaagctgaccgtcctag",
                      "tgtggtattcggcggagggaccaagctgaccgtcctag",
                      "ttgggtgttcggcggagggaccaagctgaccgtcctag",
                      "ttttgtatttggtggaggaacccagctgatcattttag",
                      "ctgggtgtttggtgaggggaccgagctgaccgtcctag",
                      "ctgggtgtttggtgaggggacggagctgaccgtcctag",
                      "taatgtgttcggcagtggcaccaaggtgaccgtcctcg",
                      "tgctgtgttcggaggaggcacccagctgaccgtcctcg",
                      "tgctgtgttcggaggaggcacccagctgaccgccctcg" };
```

Geben Sie Java-Ausdrücke an, mit denen man folgendes erhält:

1. Die fünfte DNA-Sequenz.
2. Die erste Base aus der vierten DNA-Sequenz.
3. Die zehnte Base aus der sechsten DNA-Sequenz.
4. Die letzte Base aus der sechsten DNA-Sequenz.
5. Die letzte Base aus der letzten DNA-Sequenz.

Übung 13.5 Arrays von Strings verarbeiten, mittel

Schreiben Sie ein Java-Programm, das für alle Stellen der Sequenzen aus der vorherigen Übung ausgibt, an wieviel Prozent dieser Stellen ein Thymin vorkommt. Beispielsweise so:

```
Stelle 1: 80% Thymin
Stelle 2: 50% Thymin
...
```

Übung 13.6 Arrays von Strings verarbeiten, schwer

In Erweiterung der vorherigen Aufgabe ordnen wir nun den Basen Zahlen zu: $\mathrm{T}=0, \mathrm{C}=1, \mathrm{A}=2, \mathrm{G}=3$. Damit können wir das Programm so erweitern, dass für jede Stelle die relativen Häufigkeiten *aller Basen* gezählt, und in einem Array gespeichert werden.

Erweitern Sie das Programm auf diese Weise, so dass folgende Ausgabe produziert wird:

```
Stelle 1: 80% Thymin, 20% Cytosin, 0% Adenin, 0% Guanin
Stelle 2: 50% Thymin, 0% Cytosin, 10% Adenin, 40% Guanin
...
```

Prüfungsaufgaben zu diesem Kapitel

Übung 13.7 Syntaxfehler finden, leicht, original Klausuraufgabe, mit Lösung

Betrachten Sie folgenden Java-Code:

```
int[] a = {1,2,3};
int[] b = new int{2*a.length};
for( int i = 0 ; i < a.length() ; i ++ )
{
  b[2*i] = a[i];
  b[2*i+1] = a[i];
}
```

1. In zwei Zeilen des Codes befinden sich Syntaxfehler. Wie lauten diese Zeilen richtig?
2. Wie lautet der Inhalt des Arrays `b` nach Ausführung des (berichtigten) Codes?

Übung 13.8 Abstand zweier Vektoren berechnen, mittel, typische Klausuraufgabe

Gegen sind zwei Arrays:

```
double[] a = {...};
double[] b = {...};
```

Geben Sie ein Programm an, dass den Abstand zwischen den beiden n-dimensionalen Vektoren `a` und `b` bestimmt. Zur Erinnerung: Der Abstand zwischen zwei n-dimensionalen Vektoren ist definiert als

$$\text{distance}(a,b) = \sqrt{\sum_{i=1}^{n} |a_i - b_i|^2}.$$

Die Quadratwurzel einer Zahl `x` erhalten Sie mittels `Math.sqrt(x)`. Sie dürfen annehmen, dass die Vektoren die gleiche Dimension haben.

Übung 13.9 Nukleotidfrequenzen berechnen, mittel, original Klausuraufgabe

Die Nukleotidfrequenz eines Gens gibt darüber Auskunft, welches Nukleotid in dem Gen wie oft vorkommt. Zum Bespiel kommt in der Sequenz `GATTACA` das `'A'` dreimal, das `'C'` einmal, das `'G'` einmal und das `'T'` zweimal vor. Diese Information kann man in Java in einem Array speichern, hier also `{3,1,1,2}` (in der Reihenfolge `'A'`, `'C'`, `'G'`, `'T'`).

Ergänzen Sie den Code der folgenden Methode, so dass die Nukleotidfrequenz in der beschriebenen Array-Form berechnet und zurückgegeben wird! Dabei können Sie davon ausgehen, dass der übergebene String wirklich nur aus den Zeichen `'A'`, `'C'`, `'G'` und `'T'` besteht.

```
int[] nucleotideFrequency( String s ){
  int[] r = new int[4];
  // Hier Ihr Code.

  return r;
}
```

14-1

Kapitel 14
Scoping
Globalisierung und Subsidiaritätsprinzip

14-2

Lernziele dieses Kapitels

1. Scoping verstehen und anwenden können.
2. Die Speicherung von Variablen verstehen.

Inhalte dieses Kapitels

14.1	Problem: Variablennamen	117
14.1.1	Name-Clashing	117
14.1.2	Verwendungsdauer	118
14.2	Lösung: Scoping	119
14.2.1	Der Begriff des Scopes	119
14.2.2	Gültigkeit von Variablen	119
14.2.3	Sichtbarkeit von Variablen	120
14.2.4	Speicherung von Variablen	121
	Übungen zu diesem Kapitel	122

Worum es heute geht

Die Bedeutung und Wichtigkeit des Scopings hat die katholische Kirche schon sehr viel früher erkannt als die Informatik:

Aus *de.wikipedia.org/wiki/Subsidiarität*

> Die Formulierung des Subsidiaritätsprinzips reicht in die Zeit unmittelbar nach der Reformation zurück und hat ihren Ursprung in der calvinistischen Konzeption des Gemeinwesens. Die Synode in Emden (Ostfriesland, 1571), welche über das entstehende neue Kirchenrecht zu befinden hatte, entschied in Abgrenzung zur bisher geltenden zentralistischen katholischen Kirchenlehre, dass Entscheidungen jeweils auf der niedrigst möglichen Ebene getroffen werden sollen:
>
> > Provinzial- und Generalsynoden soll man nicht Fragen vorlegen, die schon früher behandelt und gemeinsam entschieden worden sind [...] und zwar soll nur das aufgeschrieben werden, was in den Sitzungen der Konsistorien und der Classicalversammlungen nicht entschieden werden konnte oder was alle Gemeinden der Provinz angeht.
>
> Diese Vorstellung von Subsidiarität wurde 1604 von Johannes Althusius in einer philosophisch-politischen Reflexion über das Wesen des Staates formalisiert. In Aufnahme des biblischen »Bundes-Gedankens« verstand er die Gesellschaft als verschiedene, miteinander verbundene Gruppen, die jede ihre eigenen Aufgaben und Ziele zu erfüllen haben, die aber in gewissen Bereichen auf die Unterstützung (*subsidium*) der übergeordneten Gruppe angewiesen sind. Die Unterstützung soll aber nur dort einsetzen, wo sich Unzulänglichkeiten offenbaren, keinesfalls aber die Aufgabe der anderen Gruppe völlig übernehmen. Seine Theorien einer möglichst weit gehenden Autonomie aller Zwischeninstanzen zwischen Individuum und Staat konnte Althusius als Bürgermeister in Emden auch unmittelbar in der Praxis erproben.
>
> Ausgehend von Aristoteles und weiterentwickelt von Thomas von Aquin floss das Subsidiaritätsprinzip 1891 durch die Enzyklika *Rerum novarum* dann auch in die katholische Soziallehre ein und markierte eine entscheidende Wende in der katholischen Staatstheorie. Diese gab damit die päpstlich zentralistische Sicht des Staatswesens definitiv auf, das von einem Monarchen mit göttlichen Rechten gelenkt wurde.
>
> Eine klassische Formel des Subsidiaritätsprinzips findet sich in der Sozialenzyklika *Quadragesimo anno* von Papst Pius XI. »über die Gesellschaftliche Ordnung« vom 15. Mai 1931. Hiermit

schloss Papst Pius XI. an das genannte Rundschreiben Leos XIII. *Rerum novarum* (1891) an und entwarf unter dem Eindruck zunehmender zentralistischer und totalitärer staatlicher Tendenzen einen Gesellschaftsansatz, der das Individuum im Rahmen seiner individuellen Leistungsfähigkeit zum Maßstab und zur Begrenzung überindividuellen Handelns machte.

Sie sehen nicht sofort, was dies mit Informatik zu tun hat? Nun, das Subsidiaritätsprinzip bedeutet übersetzt für die Informatik: Löse Probleme dort, wo sie entstehen, und nicht auf einer übergeordneten Ebene. In der Informatik heißt dieses Konzept *Scoping*.

14.1 Problem: Variablennamen

14.1.1 Name-Clashing

Manche Namen sind besser als andere. 14-4

(Alle Tiere sind gleich. Aber manche sind gleicher als andere.)

Warum gibt es so viele Peter Müller und Jan Fischer? Warum gibt es so wenige Cornelia Schmalz-Jacobsen und Sabine Leutheusser-Schnarrenberger?

Moral

Gute Bezeichner sind selten!

Gute Bezeichner müssen viel leisten. 14-5

Gute Bezeichner sind kurz, aussagekräftig und leicht zu merken.

Bezeichner aus einem Programm. 14-6

Gute Bezeichner sind: `sum`, `length`, `List`, `velocity`.
Nicht ganz so gute Bezeichner sind `j`, `tikznumberofcurrentchild`.
Schlechte Bezeicher sind `tikz@collect@children@cchar`, `tikz@p@c@n@c@at`.

Das Problem des Name-Clashes. 14-7

Ein *Name-Clash* liegt vor, wenn derselbe Bezeichner für unterschiedliche Dinge verwendet werden soll:

```
...
int sum_n = 0, sun_m = 0;

int i;
for (i=0; i<=n; i = i+1) {
  sum_n = sum_n+i;
}

int i;  // Verboten!
for (i=0; i<=m; i = i+1) {
  sum_m = sum_m+i;
}
```

Einfache Lösungen der Name-Clashes. 14-8

- Man kann eine der Variablen umbenenen:

```
int i;
for (i=0; i<=n; i = i+1) {
  sum_n = sum_n+i;
}

int j;
for (j=0; j<=m; j = j+1) {
  sum_m = sum_m+j;
}
```

  Nachteil: Umbenennungen sind fehlerträchtig und nicht immer möglich.
- Man kann die Variable einfach zweimal benutzen:

```
int i;
for (i=0; i<=n; i = i+1) {
  sum_n = sum_n+i;
}

for (i=0; i<=m; i = i+1) {
  sum_m = sum_m+i;
}
```

Nachteil: Variablen werden nicht dort deklariert, wo sie benutzt werden.

14-9 Name-Clashes kommen auch auf anderen Ebenen vor.

```
// SAP:
class List {
...
}

...

// Software AG:
class List {
...
}
```

14.1.2 Verwendungsdauer

14-10 Variablen sollten nicht zu lange Speicher belegen.

Variablen belegen unter Umständen sehr viel Speicher (Beispiel: Variable für ein Bild). Wird eine Variable nicht mehr benutzt, so sollte der Speicher wiederverwendet werden.
Woher weiß nun aber der Übersetzer, wann eine Variable nicht mehr gebraucht wird?

14-11 Die Lebensdauer zweier Variablen.

```
int i;
for (i=0; i<=n; i = i+1) {
  sum_n = sum_n+i;
}

// Ab hier wird i gar nicht mehr gebraucht.
// Aber woher soll der Übersetzer das wissen?

int j;
for (j=0; j<=m; j = j+1) {
  sum_m = sum_m+j;
}
```

14.2 Lösung: Scoping

14.2.1 Der Begriff des Scopes

Die Lösung aller Probleme: Scopes. 14-12

Ein *Scope* ist ein bestimmter Bereich eines Programms, der in Java in der Regel durch geschweifte Klammern eingeschlossen wird. Solche Scopes lassen sich schachteln.

Beispiel eines Programms mit mehreren Scopes. 14-13

```
class Hello
{ // Start von Scope 1

  public static void main (String[] args)
  { // Start von Scope 2
    int i = 0;

    while (i < 5)
    { // Start von Scope 3
      i = i + 1;
    } // Ende von Scope 3


    while (i > 0)
    { // Start von Scope 4
      i = i - 1;
    } // Ende von Scope 4

  } // Ende von Scope 2

} // Ende von Scope 1
```

14.2.2 Gültigkeit von Variablen

Variablen sind nur innerhalb ihres Scopes gültig. 14-14

```
...
int sum_n = 0, sun_m = 0;

{
  int i;
  for (i=0; i<=n; i = i+1) {
    sum_n = sum_n+i;
  }
}

// Hier gibt es kein i mehr!

{
  int i;  // Ok!
  for (i=0; i<=m; i = i+1) {
    sum_m = sum_m+i;
  }
}
```

✎ Zur Übung 14-15

Welche Variablen sind an den Orten A, B und C gültig?

```
int n, m;
{
  for (int i = 0; i < n; i = i+1) {
    n = n+m;
  }

  // Ort A
```

```
  int j;
  {
    for (j = 0; j < n; j = j+1) {
      m = m+n;
    }
  }
  // Ort B
}
// Ort C
```

14.2.3 Sichtbarkeit von Variablen

14-16

Wer ist gemeint?

»Die Bank steht in einem Park.«

Copyright by Håstad Svensson, GNU Free Documentation License

Copyright by Jean-Jacques Milan, GNU Free Documentation License

14-17

Die Verdeckung von Bezeichnern.

Bezeichner innerhalb eines Scopes *verdecken* gleiche Bezeichner weiter außen. Es gilt also immer der Bezeichner, der am weitesten innen deklariert wurde.

14-18

✎ Zur Übung

Welchen Werte haben `i` und `j` am Ende?

```
int i = 0;
int j = 0;
{
  int i = 5;
  j = 0;
  {
    int j = i;
    i = j+1;
    j = i-1;
  }
  j = i;
}
```

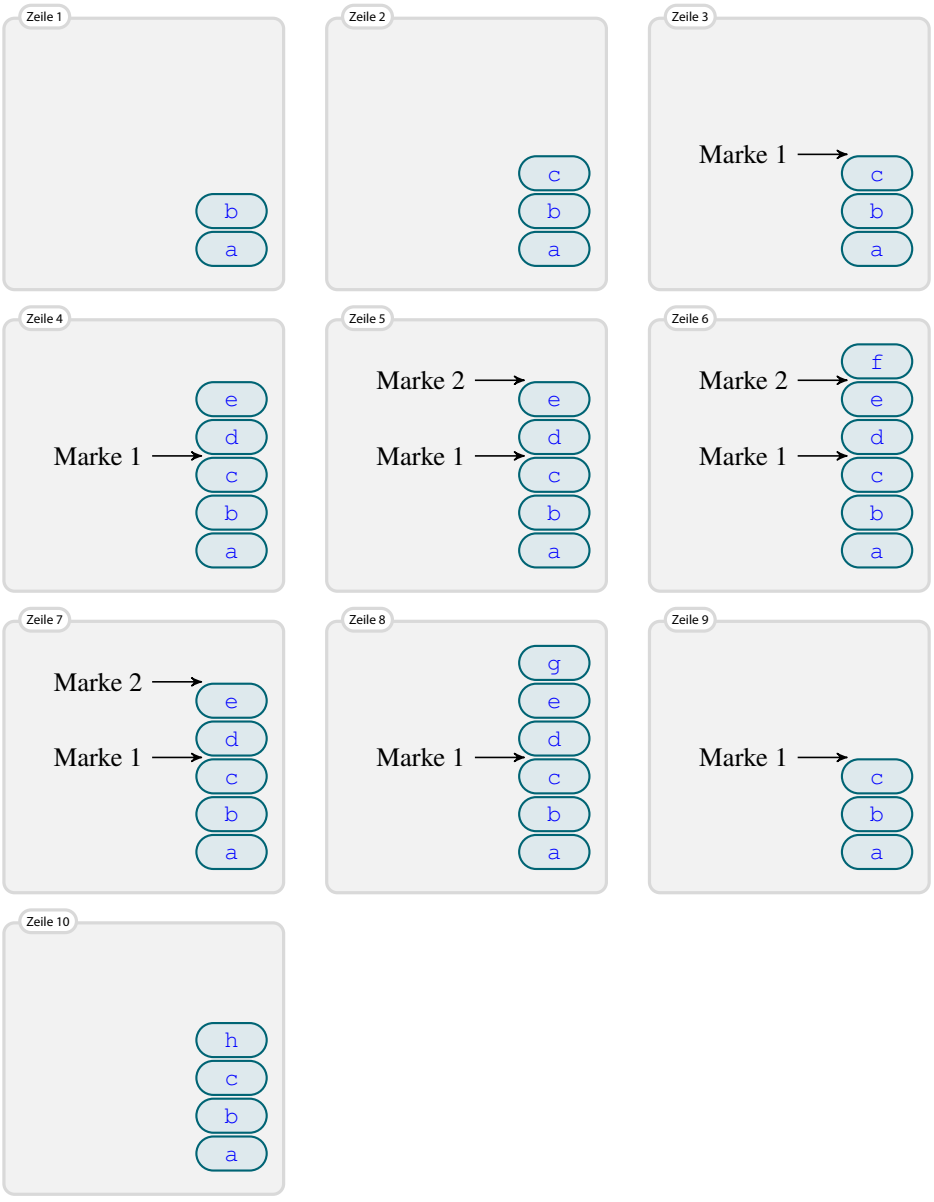
14.2.4 Speicherung von Variablen

Scopes helfen dem Übersetzer bei der Speicherreservierung. 14-19

Am Anfang eines Scopes merkt sich der Übersetzer, wie viel Speicherplatz bereits verbraucht ist. Wird dann eine neue Variable deklariert, so steigt der Speicherverbrauch. Am Ende eines Scopes kann aller Speicher freigegeben werden ab der gemerkten Stelle.

Speicherbelegung während des Ablaufs eines Programms. 14-20

```
int a, b;   // Zeile 1
int c;      // Zeile 2
{           // Zeile 3
  int d,e; // Zeile 4
  {         // Zeile 5
    int f; // Zeile 6
  }         // Zeile 7
  int g;    // Zeile 8
}           // Zeile 9
int h;      // Zeile 10
```



Zusammenfassung dieses Kapitels

14-21

▶ Syntax von Scopes

Scopes beginnen mit einer geschweiften Klammer und enden bei der zugehörigen schließenden geschweiften Klammer.

```
{
  int i;
  ...
}
// Wir sind jetzt außerhalb des Scopes und
// i ist damit verschwunden
```

Ausnahme: `for` eröffnet direkt einen Scope:

```
for (int i = 0; ...; ...) {
  ...
}
// Wir sind jetzt außerhalb des Scopes der For-Schleife
// und i ist damit verschwunden
```

▶ Die Regeln hinter dem Scoping

1. Variablen *existieren* nur innerhalb des Scopes, in dem sie deklariert wurden.
2. Eine Variable kann in einem Scope nur einmal deklariert werden; »Sub-Scopes« dürfen aber Variablen »lokal« neu deklarieren.
3. Gibt es eine Variable gleichen Names mehrmals, so ist immer nur die *innerste sichtbar*.

▶ Merke

Am Ende eines Scopes »verschwinden« möglicherweise einige Variable, jedoch ändern die verbliebenen *nicht ihren Wert*.

Übungen zu diesem Kapitel

Übung 14.1 Programmtexte nachvollziehen, mittel, original Klausuraufgabe, mit Lösung

Geben Sie die Werte der Variablen `w`, `x`, `y`, `z` und `tmp` an den Stellen 1 und 2 im folgenden Java-Programm an.

```
{
    String  w = "TOLL", x = "INFO␣A", y = "IST", z = "CHT", tmp = "";

    tmp = z;
    z = "NI" + z;
    x = x + "␣" + y;
    y = z;
    z = w;
    y = "E" + tmp;
    // Stelle 1
}

String  w = "ACCGCGTAG", x = "", y = "", z = "", tmp = "";

for (int i = 0; i < w.length(); i++) {
    if (w.charAt(i) == 'A') x = x + "T";
    if (w.charAt(i) == 'C') x = x + "G";
    if (w.charAt(i) == 'G') x = x + "C";
    if (w.charAt(i) == 'T') x = x + "A";
    tmp = x.charAt(x.length() - 1) + tmp;
    z = z + w.charAt(w.length() - i - 1);
}
// Stelle 2
```

15-1

Kapitel 15
Modularisierung im Kleinen

Wie man große Probleme in genießbare Häppchen aufteilt

15-2

Lernziele dieses Kapitels

1. Das Konzept des Unterprogramms/Methode verstehen und anwenden können
2. Java-Syntax der Modularisierung auf Methodenebene beherrschen

Inhalte dieses Kapitels

15.1 Sich wiederholender Code 124
15.1.1 Das Problem 124
15.1.2 Die Lösung 125

15.2 Methoden in Java 125
15.2.1 Die Syntax im Überblick 125
15.2.2 Syntax: Methodenname 126
15.2.3 Syntax: Parameter 126
15.2.4 Syntax: Rückgabetyp 127
15.2.5 Syntax: Aufruf 128

Übungen zu diesem Kapitel 129

Worum es heute geht

Vor einigen Jahren haben mehrere Gruppen von Studierenden im Rahmen ihres Mini-Programmierprojekts einen Horoskopgenerator programmiert. Diese Programme können aufgrund hochkomplexer, wissenschaftlich solide fundierter astrologisch-esoterischer Analysen die Zukunft vorhersagen. Dies ist um so erstaunlicher, da Prognosen ja im Allgemeinen schwierig sind, ganz besonders, wenn sie die Zukunft betreffen.

Bekanntermaßen ist aus astrologisch-esoterischer Sicht der Wochentag des Geburtstags eines Menschen ganz besonders wichtig für sein weiteres Schicksal. Aus diesem Grund ist eines der vielen komplexen Problem, die von diesen Programmen immer wieder gelöst werden müssen, zu einem gegebenen Datum den zugehörigen Wochentag zu bestimmen. Tatsächlich ist es ja nicht ganz einfach, den Wochentag eines gegebenen Datums zu bestimmen (welcher Wochentag war zum Beispiel der 22. April 1724?), weshalb man einigen Programmtext hierfür benötigt.

Da Wochentage für die Schicksalsbestimmung so wichtig sind, wird man in einem Horoskopgenerator immer wieder an verschiedenen Stellen im Programm Wochentage zu gegebenen Daten berechnen wollen. Sehr unschön wäre es, wenn man dazu den Programmtext immer und immer wieder an den verschiedenen Stellen im Programm einbauen müsste: Dies würde viel Schreibarbeit erzeugen und das Programm sehr lang werden lassen. Schließlich wäre es auch schwierig, noch etwas an dem Programmtext zu ändern, denn man müsste dann ja immer alles an allen Stellen ändern, wo der Programmtext auftaucht.

Die Lösung dieses Problems ist, eine *Methode* für die Wochentagsberechnung einzuführen. Eine solche Methode bekommt mehrere Eingaben, in unserem Fall drei Integer-Werte, welche den Tag, den Monat und das Jahr darstellen, und liefern als Ausgabe einen neuen Wert, in unserem Fall beispielsweise einen String, der den Tag enthält. Ein solche Methode wird nur einmal aufgeschrieben und kann dann später beliebig oft *aufgerufen* werden. Bei einem solchen Aufruf kann man sich vorstellen, dass die Methode angelaufen kommt, ihre Eingabe erhält, fleißig arbeitet, das Ergebnis überreicht und dann brav wieder zurückkehrt, von wo auch immer sie hergekommen ist.

Die Syntax zum Aufschreiben von Methoden ist leider »nicht optimal«, insbesondere ist die

Reihenfolge, in der man die Dinge aufschreibt, wenig intuitiv. Aber nachdem Sie sich schon an Gleichheitszeichen für Zuweisungen und mit Null beginnende Indizierungen gewöhnt haben, sind Sie sicherlich genügend abgehärtet, dass Ihnen die Methoden-Syntax von Java nur ein kurzes Unbehagen bereiten wird.

Hier übrigens mein Wochenhoroskop von dem Programm iHoroskop der Studierenden:

Geburtsdatum	26.12.1975
Sternzeichen	Steinbock
Gesundheit	Nein, Sie sind nicht krank, Sie lieben es nur, sich krank zu stellen. Ziehen Sie die Bettdecke über den Kopf und schlafen Sie die nächsten paar Tage.
Beruf	Sie argumentieren fundiert und überzeugend, das bringt Sie nach vorn. Gedulden Sie sich und lassen Sie sich nicht drängen.
Geld	Auf der letzten Feier haben Sie wohl nicht daran gedacht, dass das Geld auch noch für den Rest des Monats reichen muss, jetzt heißt es erstmal, auf schmalem Fuß leben. Halten Sie Ihre Finanzen zusammen, wenn Sie so weitermachen, stehen Sie bald vor einer Pleite.
Soziales	Es ist keine sehr günstige Woche für den Umgang mit anderen Menschen, am allerbesten wäre es für Sie, wenn Sie sich zurückziehen können.
Psyche	Auch wenn Sie denken, eine Schönheits-OP könnte Ihnen Ihr Selbstvertrauen zurückgeben, sollten Sie erst einmal über Alternativen nachdenken. Nehmen Sie an einer Lichttherapie teil, das wird Ihre Stimmung aufheitern.

15.1 Sich wiederholender Code

15.1.1 Das Problem

15-4

Wie trimmt man einen String von hinten?

Problemstellung

In einem String sollen alle Leerzeichen am Ende entfernt werden.

Lösungsidee

Es sind bereits Algorithmen zum Umdrehen und zum Trimmen vorne vorhanden. Hinten-Trimmen geht dann so:

1. Drehe den String um.
2. Trimme vorne.
3. Drehe den String um.

15-5

Zur Diskussion: Welche Probleme hat folgende Lösung?

```
String s = "trim␣me␣␣␣";
{ // first reverse:
  String result = "";
  for (int i = s.length() - 1; i >= 0; i=i-1) {
    result = result + s.charAt(i);
  }
  s = result;
}
{ // trim at front:
  String result = "";
  int start = 0;
  while (start < s.length () && s.charAt(start) == '␣') {
    start = start + 1;
  }
  for (int i = start; i < s.length(); i = i+1) {
    result = result + s.charAt(i);
  }
  s = result;
}
{ // second reverse:
```

```
  String result = "";
  for (int i = s.length() - 1; i >= 0; i=i-1) {
    result = result + s.charAt(i);
  }
  s = result;
}
```

15.1.2 Die Lösung

Lösung des Problems der Wiederholung. 15-6

Wie brauchen eine Möglichkeit, sich wiederholende Programmteile in ein eigenes *Unterprogramm* auszulagern. Solche Unterprogramme haben viele Namen:

- *Methode* (bei objektorientierten Sprachen wie Java)
- Funktion (bei funktionalen und imperativen Sprachen)
- Prozedur (bei imperativen Sprachen)
- Unterprogramm (bei imperativen Sprachen)

Es ist immer das gleiche Konzept.

Methoden sind kleine EVAs. 15-7

Methoden arbeiten nach dem EVA-Prinzip:

1. Eine Methode bekommt *Eingaben* (aber typischerweise *nicht* von der Tastatur!).
   Beispiel: Ein String
2. Eine Methode *verarbeitet* ihre Eingabe.
   Beispiel: Sie dreht den String um
3. Eine Methode produziert *Ausgaben* (aber typischerweise *nicht* auf dem Bildschirm!).
   Beispiel: Der umgedrehte String

Das Programm mit Methoden

```
String s = "trim␣me␣␣␣";
s = reverse (s);
s = trim (s);
s = reverse (s);
```

Wie die Methoden hingeschrieben werden, kommt gleich.

15.2 Methoden in Java

15.2.1 Die Syntax im Überblick

Die Bestandteile einer Methode in Java. 15-8

Eine Methode besteht aus einem *Namen* (ein Java-Bezeichner), einer Anzahl von (Eingabe)-*Parametern*, einem *Körper*, einem *Ausgabetyp* und *Attributen* wie **`public`** oder **`static`**. Die Attribute interessieren uns erstmal nicht, wir geben aber immer **`static`** an.

Die prinzipielle Syntax von Methoden in Java. 15-9

```
static return_type method_name(parameters)
{
  // Body
  return some_value;
}
```

Es ist

- *`return_type`* der Typ des Rückgabewertes.
- *`method_name`* der Name der Methode.
- *`parameters`* eine Liste der Parameter.
- *`some_value`* ein Ausdruck vom Typ `return_type`.

15-10 Die Reverse-Methode.

```
static String reverse (String s)
{
  String result = "";

  for (int i = s.length() - 1; i >= 0; i=i-1) {
    result = result + s.charAt(i);
  }

  return result;
}
```

15.2.2 Syntax: Methodenname

15-11 Syntax des Methodennames.

Der Name einer Methode muss ein *Java-Bezeichner* sein. Er darf also keine Leerzeichen oder Sonderzeichen enthalten und muss mit einem Buchstaben anfangen. Es ist *üblich*, dass er mit einem Kleinbuchstaben anfängt, und es *sollte* ein guter Bezeichner gewählt werden.

15-12 Die Reverse-Methode.

```
static String reverse (String s)
{
  String result = "";

  for (int i = s.length() - 1; i >= 0; i=i-1) {
    result = result + s.charAt(i);
  }

  return result;
}
```

15.2.3 Syntax: Parameter

15-13 Syntax der Parameter.

Eine Methode kann *mehrere* (Eingabe-)Parameter haben, die durch *Kommata* getrennt werden. Die Parameter *folgen* dem Methodennamen und stehen in *runden Klammern*. Für *jeden* Parameter muss sein Typ angegeben werden, wie bei einer Deklaration. In einer Methode kann man Parameter wie andere Variablen auch behandeln. Man *sollte* ihnen aber nichts zuweisen.

15-14 Die Reverse-Methode.

```
static String reverse (String s)
{
  String result = "";

  for (int i = s.length() - 1; i >= 0; i=i-1) {
    result = result + s.charAt(i);
  }

  return result;
}
```

15-15 Syntax der Parameter.

Methoden ohne Parameter.

Die Klammern nach dem Methodennamen sind selbst dann nötig, wenn es gar keine Parameter gibt.

```
static double pi ()
{
  return 3.141592653589793;
}
```

✎ Zur Übung 15-16
Welche der folgenden Parameterlisten sind korrekt?

1. `static int foo (int i)`
2. `static int foo (int i, int j)`
3. `static int foo (int i, j)`
4. `static int foo (int i; int j)`
5. `static int foo (int i, String a, char b)`
6. `static int foo (())`

15.2.4 Syntax: Rückgabetyp

Syntax des Rückgabetyps. 15-17
Eine Methode kann nur einen *einzigen Wert* zurückliefern. Man sagt auch, die Method *gibt* einen Wert *aus*, was aber nichts mit dem Schreiben auf den Bildschirm zu tun hat. Der Typ des Wertes ist der *Rückgabetyp*. Dieser wird der ganzen Methode aus historischen Gründen *vorangestellt*. Innerhalb einer Methode kann man jederzeit mittels **`return`** die Methode verlassen und einen Wert zurückliefern.

Die Reverse-Methode. 15-18

```
static String reverse (String s)
{
  String result = "";

  for (int i = s.length() - 1; i >= 0; i=i-1) {
    result = result + s.charAt(i);
  }

  return result;
}
```

Syntax des Rückgabetyps. 15-19
Produziert eine Methode überhaupt keinen Rückgabewert, so gibt man statt des Typs **`void`** an.

```
static void printInParantheses (String s)
{
  System.out.println("(" + s + ")");
}
```

✎ Zur Übung 15-20
Geben Sie den Code einer Methode an, die a^n berechnet. Die Methode arbeitet also wie folgt:

- Die *Eingabe* ist sind zwei Zahlen a und n.
- In der *Verarbeitung* wird $a \cdot a \cdot \cdots \cdot a$ berechnet.
- Das Produkt wird als *Ausgabe* zurückgegeben.

15.2.5 Syntax: Aufruf

15-21

Wie benutzt man nun Methoden?

Hat man eine Methode deklariert, so kann man sie *aufrufen*. Man gibt den Namen der Methode an, gefolgt von Parametern in Klammern (wie in der Mathematik bei $\sin(\pi)$). *Innerhalb der Methode* haben die *Parameter* nun die *Werte, die beim Aufruf angegeben* wurden.

Beispiel

– Der *Kopf* von `reverse` lautet:

```
static String reverse (String s)
```

– Zwei mögliche Aufrufe der Methode:

```
... reverse ("Hallo") ...
... reverse (str) ...
```

– Bei ersten Aufruf ist innerhalb von `reverse` die Variable *s* gleich `"Hallo"`. Bei zweiten Aufruf ist *s* gleich `str`.

15-22

Das ganze Programm.

```
class Trimmer {
  static String reverse (String s) {
    String result = "";
    for (int i = s.length() - 1; i >= 0; i=i-1) {
      result = result + s.charAt(i);
    }
    return result;
  }
  static String trim (String s) {
    String result = "";
    int start = 0;
    while (   start < s.length ()
            && s.charAt(start) == '␣') {
      start = start + 1;
    }
    for (int i = start; i < s.length(); i = i+1) {
      result = result + s.charAt(i);
    }
    return result;
  }
  public static void main (String[] args) {
    System.out.println
      (reverse(trim(reverse("trimm␣mich␣␣␣"))));
  }
}
```

Zusammenfassung dieses Kapitels

▶ Syntax einer Methoden-Deklaration 15-23

```
static return_type method_name(typ1 parameter1,
                               typ2 parameter2, ...)
{
  // Body
  return some_value;
}
```

- *return_type* ist der Typ des Rückgabewertes.
- *method_name* ist der Name der Methode.
- *typ1* ist der Typ von *parameter1*, entsprechend mit weiteren Parametern.
- *some_value* ist ein Ausdruck vom Typ return_type.

Besonderheiten:

- Gibt es keine Parameter, so müssen trotzdem (leere) runde Klammern stehen.
- Gibt es keine Rückgabewerte, so gibt man als Typ **void** an.

▶ Syntax eines Methoden-Aufrufs

Man ruft eine Methode auf mittels method(wert1,wert2,...). Ein solcher Aufruf ist ein *Ausdruck*. Er wertet zum *Rückgabewert der Methode aus*. Innerhalb der Method haben die Parameter-Variablen die Werte, die beim Aufruf angegeben wurden. Selbst wenn es keine Parameter gibt, müssen trotzdem die runden Klammern geschrieben werden.

Übungen zu diesem Kapitel

Übung 15.1 Programmierschnittstelle entwerfen, leicht

Um Methoden zu verwenden, müssen wir nicht wissen, wie diese programmiert sind. Es reicht aus, die *Programmierschnittstelle* (englisch *application programming interface*, API) zu kennen. Zum Beispiel finden wir unter

`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html`

die Programmierschnittstelle für die Methode cos der Klasse Math:

```
static double cos( double a )
// Returns the trigonometric cosine of an angle.
```

Nun wissen wir, dass wir mit Math.cos(0.5) in Java den Cosinus von 0,5 erhalten, brauchen uns aber nicht darum zu kümmern, wie genau Java das macht. Folgende Informationen sind in der Programmierschnittstelle enthalten:

1. Name der Methode und Datentyp des Rückgabewerts
2. Namen und Datentypen der Parameter
3. Eine für Menschen verständliche Beschreibung der Funktion der Methode.

Entwerfen Sie Programmierschnittstellen für Methoden, die folgendes leisten sollen:

1. Herausfinden, wie viele Tage ein Jahr hat.
2. Den Teil einer Zeichenkette extrahieren, der nach dem ersten Vorkommen einer anderen Zeichenkette folgt (zum Beispiel aus »Deutschland« und »Deut« wird »schland«).
3. Herausfinden, ob eine Zeichenkette eine DNA-Sequenz darstellen könnte, also nur aus den Buchstaben a, A, c, C, g, G, t und T besteht.

Wichtig ist neben den richtigen Rückgabetypen auch die Güte der verwendeten Bezeichner (Methoden und Parameter) sowie die Schlüssigkeit und Vollständigkeit der Beschreibung. Denken Sie dabei an Sonderfälle und/oder Ausnahmen!

Das Programmieren der Methoden ist nicht Teil dieser Aufgabe. Wer möchte, kann es natürlich trotzdem tun.

Übung 15.2 Probleme in Methoden aufteilen, mittel

In vier Arrays haben wir Daten aus verschiedenen Messreihen gegeben:

```
static double[] data1 =
  {0.406250,0.691406, ... , 0.726562, 0.160156};
static double[] data2 =
  {0.386719,0.664062, ... , 0.703125, 0.164062};
static double[] data3 =
  {0.320312,0.542969, ... , 0.582031, 0.132812};
static double[] data4 =
  {0.230469,0.488281, ... , 0.566406, 0.117188};
```

Diese Daten wollen wir nun ein wenig aufbereiten. Dazu soll ein Programm geschrieben werden, das eine Visualisierung in Textform ausgibt:

```
Wert     1               2               3               4
MW       0.6203124       0.5941406       0.4917968       0.4871095
SIGMA    0.1820145       0.1717725       0.1447577       0.1656548

         +-------------------------------------------------------+
Wert 1   ************************************
Wert 2   ***********************************
Wert 3   ****************************
Wert 4   ****************************
```

MW steht hier für den Mittelwert, SIGMA für die Standardabweichung.

1. Überlegen Sie, wie dieses Programm modular strukturiert werden könnte. Erstellen Sie dazu eine Liste von Unteraufgaben, die in Methoden ausgelagert werden könnten.
2. Formulieren Sie die Unteraufgaben als *Programmierschnittstellen* für Java-Methoden (siehe dazu Übung 15.1). Geben Sie lediglich den Methodenkopf und eine kurze Beschreibung an.
3. Gehen Sie nun davon aus, dass fleißige Helferlein diese Methoden schon für Sie implementiert haben. Schreiben Sie ein Hauptprogramm, das die gewünschte Ausgabe produziert.

Übung 15.3 Wurzeln berechnen, mittel

Sie kennen bereits Algorithmen zur Berechnung der ganzzahligen Quadratwurzel. Wir wollen dies nun auf beliebige Wurzeln verallgemeinern. Schreiben Sie eine Java-Methode

```
static int floor_root( int base, int radicand )
```

zur Berechnung der ganzzahligen r-ten Wurzel $\lfloor \sqrt[r]{} \rfloor$. Greifen Sie dabei auf die Methode zur Berechnung der Exponentiation zurück von Seite 15-20.

Testen Sie die Methode `floor_root` mit geeigneten Daten. Denken Sie an Sonderfälle. Böse Benutzer könnten die Methode mit einem negativen Radikanden aufrufen.

Übung 15.4 Hammingabstand berechnen, mittel

Der *Hamming-Abstand* zwischen zwei (gleichlangen) Zeichenketten ist die Anzahl der Stellen, an denen sich diese Zeichenketten unterscheiden. Der Hamming-Abstand zwischen »Till« und »Tall« ist beispielsweise 1.

Schreiben Sie eine Methode

```
static int hamming_distance( String a, String b )
```

die den Hamming-Abstand zwischen zwei Strings ermittelt. Gehen Sie bei der Programmierung davon aus, dass die Strings `a` und `b` gleichlang sind.

Übung 15.5 Minimaler Hammingabstand, schwer

Falls die beiden Strings bei der Berechnung des Hammingabstandes aus der vorherigen Aufgabe verschieden lang sind, kann man folgendes tun: Man »schiebt« den kürzeren String über den längeren und berechnet jeweils den Hamming-Abstand zu dem Fragment des längeren Strings, das an der aktuellen Stelle beginnt. Das Resultat ist dann der kleinste der berechneten Hamming-Abstände:

```
ANATOLIEN
TOLL        4
 TOLL       4
  TOLL      3
   TOLL     1
    TOLL    4
     TOLL   4
```

Hier wäre das Ergebnis also 1. Programmieren Sie eine neue Methode

```
static int min_hamming_distance( String a, String b )
```

die dieses Verfahren für zwei Strings jeweils beliebiger Länge implementiert. Dabei sollten Sie auf die Method aus Übung 15.4 zurückgreifen.
Tipp: Wenn `a` ein String ist, erhalten Sie mit `a.substring( 5, 8 )` einen Teil dieses Strings.

Übung 15.6 Alignments berechnen, mittel

Die in der vorherigen Übung erarbeitete Methode kann man verwenden, um DNA-Fragmente, die mutationsbehaftet sein können (etwa PCR-Produkte), mit längeren Sequenzen zu vergleichen. Wenden Sie Ihre Methode auf die Fragmente und die Sequenz aus dem Veranstaltungswiki an. Welches Fragment »passt« am ehesten?

Prüfungsaufgaben zu diesem Kapitel

Übung 15.7 Typbestimmung mit Methoden, leicht, typische Klausuraufgabe

Geben Sie die Typen aller geklammerten Teilausdrücke des folgenden Ausdrucks an. Es gelten dabei die Variablendeklarationen `int a` und `String tach="Moin"`.

```
((a * a) > (2.0 - Math.sqrt(4.0))) && !(tach.charAt(3)=='i'))
```

1. `(a * a)`
2. `(2.0 - Math.sqrt(4.0))`
3. `tach.charAt(3)`
4. `'i'`
5. `((a * a) > (2.0 - Math.sqrt(4.0)))`
6. `!(tach.charAt(3)=='i')`
7. `((a * a) > (2.0 - Math.sqrt(4.0))) && !(tach.charAt(3)=='i'))`

Übung 15.8 Methode auf Strings programmieren, leicht, typische Klausuraufgabe

Schreiben Sie eine Methode

```
static String removeIntrons(String sequence)
```

die als Eingabe eine DNS-Sequenz, d. h. einen String aus Buchstaben `a`, `c`, `g`, `t`, `A`, `C`, `G`, `T` erhält, und die DNS-Sequenz ohne Introns, d. h. den String ohne Kleinbuchstaben, zurückgibt.

Übung 15.9 Methode auf Strings programmieren, mittel, typische Klausuraufgabe

Schreiben Sie eine Methode

```
static boolean doesNotContainIntrons( String sequence )
```

die genau dann **`true`** zurückgibt, wenn der String `dnasequence` nur aus den Buchstaben A, C, G und T zusammengesetzt ist.

Übung 15.10 Methode auf Strings programmieren, mittel, typische Klausuraufgabe

Schreiben Sie eine Methode

```
static int whereIsTheTryptophane( String dnasequence )
```

die eine gegebene DNS-Sequenz nach dem Teilstring `TGG` durchsucht und die erste *durch drei teilbare* Position zurückgibt, an der dieser Teilstring vorkommt (wir beginnen dabei bei 0 zu zählen). Wenn der Teilstring `TGG` nicht an einer durch drei teilbaren Position gefunden werden kann, soll -1 zurückgegeben werden.

Übung 15.11 Methode für Integer programmieren, mittel, typische Klausuraufgabe

Schreiben Sie eine Methode

```
static int biggestDivisor( int n )
```

die den größten echten Teiler der Zahl `n` ermittelt und zurückgibt. (Eine Zahl ist ein echter Zeiler von `n`, wenn sie `n` teilt, aber ungleich `n` ist. Der größte echte Teiler von 15 ist zum Beispiel 5.)

Übung 15.12 CG oder AT?, leicht, original Klausuraufgabe, mit Lösung

Schreiben Sie eine Methode `boolean isMoreCG( String s )`, die in einer in einem String `s` gespeicherten DNA-Sequenz die Vorkommen von `'C'` und `'G'` zählt und mit den Vorkommen von `'A'` und `'T'` vergleicht. Wenn Summe der Anzahlen von `'C'` und `'G'` größer ist als die Summe der Anzahlen von `'A'` und `'T'`, soll `true` zurückgegeben werden, sonst `false`. Einige Beispiele:

Eingabe `s`	Anzahl `'C'`,`'G'`	Anzahl `'A'`,`'T'`	Ergebnis von `isMoreCG( String s )`
`"GACGA"`	3	2	`true`
`"GACGTAT"`	3	4	`false`
`"GGCTAT"`	3	3	`false`
`"GCCGC"`	5	0	`true`

Bei der Programmierung dürfen Sie davon ausgehen, dass Sie Übung 12.2 richtig gelöst haben, und eine Methode `countChar(String s, char c)` aufrufen, die für eine Zeichenkette `s` angibt, wie oft das Zeichen `c` dort vorkommt.

Teil IV
Algorithmik

In der Einleitung zum vorherigen Teil wurde behauptet, dass Algorithmen den wahren Kern der Informatik ausmachen und dass Programmiersprachen lediglich »Jargons sind, Algorithmen aufzuschreiben«. Der vorherige Teil hat sich dann lang und breit mit einer speziellen Art beschäftigt, Algorithmen aufzuschreiben, nämlich mit der »Java-Art«. Die ursprünglich beworbenen Algorithmen waren hingegen eher einfach, um nicht zu sagen banal, und man fragt sich, ob den Wochentag des eigenen Geburtstags nicht auch ohne Computer hätte bestimmen können (Motto »Alles dauert länger als man denkt, selbst wenn man den Computer weglässt«).

In diesem Teil wollen wir uns nun »richtige« Algorithmen anschauen, auf die man auch nicht unbedingt »so eben kommt«. Besonders interessant ist dabei der Sortierproblem: Man hat eine lange Liste von Zahlen gegeben (»lang« bedeutet hier irgendeine Länge zwischen einem Duzend und einigen Billionen Zahlen) und möchte diese in eine sortierte Reihenfolge bringen. Das Erstaunliche hierbei ist, dass es gleich mehrere Algorithmen zur Lösung dieses Problems gibt und dass diese auch noch ganz unterschiedliche Verhaltensweisen aufweisen. Beispielsweise sind manche Algorithmen besonders schnell, wenn die Zahlen schon »fast« sortiert sind, andere Algorithmen fühlen sich bei zufälligen Eingaben am wohlsten.

16-1

Kapitel 16
Suchalgorithmen
Welche Codons kodieren Arginin?

16-2

Lernziele dieses Kapitels

1. Lineare Suche verstehen und implementieren können
2. Binäre Suche verstehen und implementieren können
3. Laufzeit der Verfahren vergleichen können
4. Abschätzen können, wann sich Vorsortierung lohnt

Inhalte dieses Kapitels

16.1 Die Problemstellung 135

16.2 Lineare Suche 135
16.2.1 Idee . . . 135
16.2.2 Implementation . . . 135
16.2.3 Laufzeit . . . 136
16.2.4 Anwendung . . . 136

16.3 Binäre Suche 137
16.3.1 Idee . . . 137
16.3.2 Implementation . . . 137
16.3.3 Laufzeit . . . 139
16.3.4 Vorsortierung . . . 139

Übungen zu diesem Kapitel 140

Worum es heute geht

Nach dem zweiten Hauptsatz der Thermodynamik nimmt die Entropie im Universum ständig zu. Mit anderen Worten: Alles wird immer unordentlicher – und das aus grundsätzlichen Gründen. Ständig müssen wir viel Energie aufwenden, um Ordnung in das Chaos zu bekommen.

Es ist also schon aus physikalischen Gründen nicht weiter verwunderlich, dass Computer *sehr* viel Zeit auf das Suchen von Dingen verwenden. Sie sind vergesslicher als jeder Alzheimerpatient: Ständig sind sie damit beschäftigt, Dinge zu suchen, die sie eben noch hatten. Da wird in einer Datei nach einem Wort gesucht, in einem Menü nach einer bestimmten Zeile, in Tabellen nach bestimmten Einträgen, tief unten im System innerhalb eines Interrupthandlers nach dem aktuellen Systemkontext und hoch oben in einem Dialog nach einer Eingabe eines Benutzers in einem Text. Gesucht wird nicht nur einmal, oft werden Tabellen in jeder Sekunde viele Tausend oder gar Millionen Male durchsucht.

Glücklicherweise sind *Suchalgorithmen* eher einfach und deshalb auch sehr schnell. Im einfachsten Fall, der linearen Suche, werden einfach alle möglichen Stellen nacheinander (»linear«) durchgegangen. Dies ist nicht besonders intelligent, aber einfach und schnell. In umfangreichen, sortierten Listen wie einem Telefonbuch ist diese Art der Suche aber einfach nur dumm. Hier fängt man natürlich grob in der Mitte an und arbeitet sich dann »sprunghaft« nach vorne oder hinten vor. Mit jedem Spring halbiert sich grob die Menge der noch zu durchsuchenden Einträge, daher heißt diese Art der Suche »binär«.

Sollten Ihre Sortieralgorithmen wider Erwarten nicht funktionieren und den gesuchten Eintrag nicht finden, so empfiehlt sich ein fester Glaube an die Korrektheit Ihres Algorithmus, denn es steht bei Lukas 11:9–10 geschrieben (nach Luther): Vnd ich ſage euch auch, Bittet, ſo wirt euch gebenn, Sucht, ſo werdet yhr finden, Klopfft an, ſo wirt euch auff than. Denn wer do bittet, der nympt, vnd wer do ſucht, der findet, vnd wer do an klopfft, dem wirt auff than.

Ein Suchexperiment mit einer Reihe von aufklappbaren Schachteln. In jeder Schachtel befindet sich ein Schauspieler zusammen mit seinem Verdienst. Die Frage ist, nun, wie man möglichst wenig Schachteln öffnet, um Brad Pitt zu finden.

Regie

16.1 Die Problemstellung

Es gibt viele Varianten des Suchproblems. 16-4

Suchen eines Wertes
Eingaben: Ein Array von Werten und *ein Wert*, der gesucht wird.

Ausgabe: *Eine* Position, an der der Wert im Array vorkommt.

Suchen aller Werte
Eingaben: Ein Array von Werten und *ein Wert*, der gesucht wird.

Ausgabe: *Alle* Positionen, an der der Wert im Array vorkommt.

Suchen eines Wertes mit einer Eigenschaft
Eingaben: Ein Array von Werten und *eine Eigenschaft*, die Werte haben können oder auch nicht.

Ausgabe: *Eine* Position eines Wertes, der die Eigenschaft hat.

Suchen aller Werte mit einer Eigenschaft
Eingaben: Ein Array von Werten und *eine Eigenschaft*, die Werte haben können oder auch nicht.

Ausgabe: *Alle* Positionen von Werten, die die Eigenschaft haben.

16.2 Lineare Suche

16.2.1 Idee

Die lineare Suche ist das einfachste Suchverfahren. 16-5
Bei der *linearen Suche* werden einfach alle Werte des Arrays überprüft. Eine oder alle Positionen, an denen Werte mit der gewünschten Eigenschaft stehen, werden zurückgegeben.

16.2.2 Implementation

Beispiel einer linearen Suche. 16-6
Finden einer Telefonnummer, die auf 6 endet.

```
String[] telephoneNumbers = {"7974311",
                             "2147856",
                             "2161555",
                             "5553466"};

// Suche linear nach einer Telefonnummer, die auf 6 endet.

String nummer = "";

for (int i=0; i < telephoneNumbers.length; i++) {
  if (telephoneNumbers[i].charAt
       (telephoneNumbers[i].length()-1) == '6') {
    nummer = telephoneNumbers[i];
  }
}
// nummer ist jetzt "5553466"
```

16-7

Beispiel einer linearen Suche.

Finden der Position einer Telefonnummer im einem Array.

```
String[] telephoneNumbers = {"7974311",
                             "2147856",
                             "2161555",
                             "5553466"};

// Suche linear nach "2161555":

int position_of_value = -1; // noch nicht gefunden

for (int i=0; i < telephoneNumbers.length; i++) {
  if (telephoneNumbers[i].equals("2161555")) {
    positions_of_value = i;
  }
}
// position_of_value ist jetzt 2.
```

16-8

Eine allgemeine lineare Suche.

```
class SearchAlgorithms
{
  static int linearSearch(String[] strings,
                          String   value)
  {
    // Findet erstes Vorkommen von value in strings.
    // Kommt es nicht vor wird -1 zurückgegeben

    for (int i = 0; i < strings.length; i++) {
      if (strings[i].equals(value)) {
        return i;
      }
    }

    return -1;
  }
}
```

16.2.3 Laufzeit

16-9

✎ Zur Übung

Wie viele Vergleiche führt die Methode `linearSearch` bei einem Array der Länge n

1. mindestens,
2. höchstens und
3. im Durchschnitt

aus?

16.2.4 Anwendung

16-10

Beispielanwendung der linearen Suche.

Zu einem Codon soll die zugehörige Aminosäure ermitteln werden und umgekehrt. Dazu werden zwei Arrays erstellt: Das erste speichert die Codons, das zweite an der entsprechenden Position die Säure.

Codon zu Aminosäure Umrechnung. 16-11

```
static String codonToAcid(String codon)
{
  String[] codons = {
    "UUU","UUC","UUA","UUG","UCU","UCC","UCA","UCG",
    "UAU","UAC","UAA","UAG","UGU","UGC","UGA","UGG",
    "CUU","CUC","CUA","CUG","CCU","CCC","CCA","CCG",
    "CAU","CAC","CAA","CAG","CGU","CGC","CGA","CGG",
    "AUU","AUC","AUA","AUG","ACU","ACC","ACA","ACG",
    "AAU","AAC","AAA","AAG","AGU","AGC","AGA","AGG",
    "GUU","GUC","GUA","GUG","GCU","GCC","GCA","GCG",
    "GAU","GAC","GAA","GAG","GGU","GGC","GGA","GGG"};
  String[] acids = {
    "Phe","Phe","Leu","Leu","Ser","Ser","Ser","Ser",
    "Tyr","Tyr","Stp","Stp","Cys","Cys","Stp","Trp",
    "Leu","Leu","Leu","Leu","Pro","Pro","Pro","Pro",
    "His","His","Gln","Gln","Arg","Arg","Arg","Arg",
    "Ile","Ile","Ile","Met","Thr","Thr","Thr","Thr",
    "Asn","Asn","Lys","Lys","Ser","Ser","Arg","Arg",
    "Val","Val","Val","Val","Ala","Ala","Ala","Ala",
    "Asp","Asp","Glu","Glu","Gly","Gly","Gly","Gly"};
  return
    acids[linearSearch(codons, codon)];
}
```

✎ Zur Übung 16-12

Schreiben Sie eine Methode, die eine Aminosäure als Eingabe erhält und *alle* Codons zu dieser Aminosäure auf dem Bildschirm ausgibt.
(Für Fortgeschrittene: Wie können Sie alle Codons in einem Array zurückgeben?)

16.3 Binäre Suche

16.3.1 Idee

Die Grundidee der binären Suche. 16-13

Beobachtung

Suchen wir einen Namen im Telefonbuch, so suchen wir diesen natürlich nicht linear. Vielmehr fangen wir grob in der Mitte an und gehen dann sprungweise nach vorne oder nach hinten.

Binäre Suche

Binäre Suche arbeitet auf *sortierten* Arrays. Man *halbiert* zu Anfang den Suchraum in der Mitte. Dann behandelt man nur noch eine der beiden Seiten.

16.3.2 Implementation

Beispiel einer binären Suche. 16-14

```
String[] names =
 {"Alice", "Bob", "Charly", "Doris", "Till", "Viktor"};

// Binäre Suche nach "Till":

int lower_bound = 0;
int upper_bound = names.length - 1;

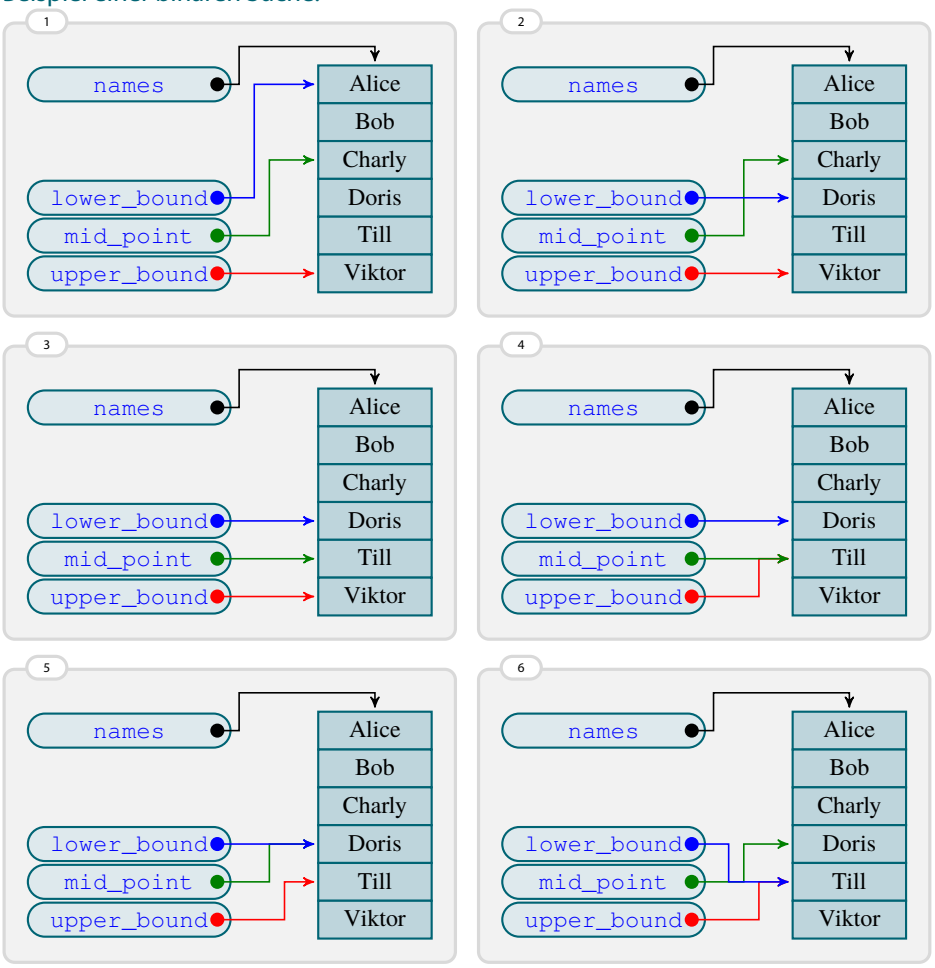
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") {// geschummelt
    lower_bound = mid_point + 1;
```

```
  }
  else {
    upper_bound = mid_point;
  }
}

return lower_bound;
```

16-15

Beispiel einer binären Suche.



16-16

Eine allgemeine binäre Suche.

```
class SearchAlgorithms
{
  static int binarySearch(String[] strings,
                          String   value)
  {
    // Finde Vorkommen von value in strings (sortiert).
    int lower_bound = 0;
    int upper_bound = strings.length - 1;

    while (lower_bound != upper_bound)
    {
      // Berechne Mitte des Intervals
      int mid_point = (lower_bound + upper_bound) / 2;

      if (strings[mid_point].compare(value) < 0) {
        // Im oberen Interval, erhöhe untere Schranke
        lower_bound = mid_point + 1;
      }
      else { // Unteres Interval, senke obere Schranke
        upper_bound = mid_point;
```

```
      }
    }
    return lower_bound;
  }
}
```

16.3.3 Laufzeit

✎ Zur Übung 16-17

Wie viele Vergleiche führt die Methode `binarySearch` bei einem Array der Länge n

1. mindestens,
2. höchstens und
3. im Durchschnitt

aus?

16.3.4 Vorsortierung

Vorsortierung ist nützlich, wenn öfters gesucht werden soll. 16-18

Binäre Suche *funktioniert nur* auf sortierten Daten. Will man unbedingt binäre Suche verwenden, so kann man die Daten *vorsortieren*. Das Sortieren von Daten dauert aber (wesentlich) länger als *eine* lineare Suche. Sind die Daten aber einmal sortiert, gehen *nachfolgende* binäre Suchen schnell.

Folgerung

Vorsortieren lohnt sich nur, wenn in den Daten *mehrmals* gesucht werden soll.

(Genauer: Vorsortieren lohnt sich ab etwa $\log_2 n$ Suchen.)

Zusammenfassung dieses Kapitels

▶ Lineare Suche 16-19

Die *lineare Suche* durchläuft einfach alle Elemente. Das Wesentliche ist folgende die Schleife:

```
for (int i = 0; i < values.length; i++) {
  if (values[i] == what_we_search_for) {
    return i;
  }
}
```

Die lineare Suche benötigt im Schnitt $n/2$ Vergleiche bei Arrays der Länge n.

▶ Binäre Suche

Die *binäre Suche* springt in einem *sortierten* Array herum. Das Wesentliche ist folgende die Schleife:

```
while (lower_bound != upper_bound) {
  int mid_point = (lower_bound + upper_bound) / 2;
  if (values[mid_point] < what_we_search_for) {
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Die binäre Suche benötigt $\log_2 n$ Vergleiche, was *sehr schnell ist*.

Übungen zu diesem Kapitel

Zum Test der folgenden Aufgabe ist es sinnvoll, einen großen Arrays von Zufallszahlen zu besitzen. Einen solchen kann man wie folgt erzeugen:

```
static int[] zufallszahlen = new int [10000000];
for (int i = 0; i < zufallszahlen.length; i++) {
  zufallszahlen[i] = (int) (Math.random() * 1000);
}
// In zufallszahlen sind nun 10.000.000 Zahlen zwischen 0 und 999
```

Übung 16.1 Suchalgorithmen implementieren, mittel

Implementieren Sie mit zwei Java Methoden

```
static int lineare_suche( int nadel, int[] heuhaufen )
static int binaere_suche( int nadel, int[] heuhaufen )
```

die beiden bekannten Suchalgorithmen aus der Vorlesung. Beide Methoden sollen den Index des Werts `nadel` im Array `heuhaufen` zurückgeben, wenn dieser dort gefunden werden kann. Ansonsten soll `-1` zurückgegeben werden.

Wenden Sie die Methoden auf das opben genannte Zufallsarray an und such Sie dort nach der Zahl 555. Wie lange dauert das ungefähr?

Übung 16.2 Suchalgorithmen implementieren, mittel

Ihr persönliches Telefonbuch ist in zwei Arrays gespeichert:

```
static String[] personen =
  {Till, Johannes, Markus, Claudia, ...};
static String[] nummern =
  {5311, 5313, 5314, 5300, ...};
```

Schreiben Sie ein Methode, das den Namen einer Person als Parameter bekommt und das dann die Telefonnummer dieser Person auf dem Bildschrim ausgibt.

Prüfungsaufgaben zu diesem Kapitel

Übung 16.3 Binäre Suche verstehen, leicht, typische Klausuraufgabe

Gegeben sei ein Array mit folgenden Einträgen:

12, 14, 27, 33, 56, 58, 63, 70, 75, 82, 89, 89, 90, 91, 92

Geben Sie die Vergleiche an, die bei einer binären Suche in diesem Array nach dem Element 58 durchgeführt werden!

17-1

Kapitel 17
Sortieralgorithmen
Skat, das Telefonbuch und Atome

Lernziele dieses Kapitels

1. Arten von Sortierproblemen kennen
2. Bubble-Sort, Insertion-Sort, Selection-Sort verstehen und implementieren können

17-2

Inhalte dieses Kapitels

17.1	Die Problemstellung	142
17.1.1	Motivation	142
17.1.2	Problemvarianten	142
17.2	Sortieralgorithmen	143
17.2.1	Bubble-Sort	143
17.2.2	Selection-Sort	144
17.2.3	Insertion-Sort	146
17.3	*Untere Schranke für die Vergleichszahl	147
	Übungen zu diesem Kapitel	148

Worum es heute geht

Wie wir schon im vorherigen Kapitel gesehen haben, ist der zweite Hauptsatz der Thermodynamik für Computer ein echtes Problem: Ständig wird alles unordentlich. Man kann sich, wie im letzten Kapitel geschehen, damit behelfen, ständig in den Daten im Speicher herumzusuchen. Wir haben aber auch gesehen, dass eine solche Suche besonders schnell geht, wenn die Daten sortiert – also wohlgeordnet – sind. Damit beißt sich die Katze in den Schwanz: Weil die Daten nicht sortiert sind, müssen wir suchen; aber um schnell zu suchen, müssen die Daten sortiert sein.

Um diesen mehr oder weniger gordischen Knoten zu durchschlagen, brauchen wir *Sortieralgorithmen*. Diese bekommen einen Array von Zahlen oder Dingen als Eingabe und ändern die Reihenfolge der Elemente des Arrays derart, dass hinterher alles schön sortiert ist. Danach fällt uns auch das Suchen viel leichter.

Das Sortieren von Zahlen ist schwieriger als das Suchen nach Zahlen. Dies ist zum einen ein Unglück, denn man muss komplexere Algorithmen lernen, diese sind schwieriger zu programmieren und sie sind langsamer als Suchalgorithmen. Aus Sicht Theoretischer Informatiker ist es hingegen ein Glück, denn so kann man viel mehr forschen, veröffentlichen und Drittmittel einwerben. Tatsächlich können Sie auch heute noch Artikel über Sortierverfahren auf renommierten Konferenzen vorstellen und in renommierten Zeitschriften veröffentlichen.

Sortieren lohnt sich nicht immer. Stellen Sie sich Ihren unaufgeräumten Schreibtisch vor. Sie suchen ein bestimmtes Blatt, von dem Sie wissen, dass es »da irgendwo sein muss«. Sie könnten nun hingehen und zunächst den ganzen Schreibtisch komplett aufräumen, alles gegebenenfalls abstauben und dann abheften, so dass sie am Ende das gewünschte Blatt mit einem Griff finden werden. In der Regel werden Sie dies aber nicht tun, sondern einfach kurz den Schreibtisch nach dem gesuchten Blatt durchwühlen. Das Aufräumen lohnt sich nur, wenn Sie in den nächsten Tagen ständig unterschiedliche Dinge suchen werden. Ganz ähnlich die Sachlage beim Sortieren von Daten im Rechner: Muss man nur einmalig etwas in den Daten suchen, so lohnt es sich nicht, diese erst zu sortieren. Werden Sie aber immer und immer wieder in den Daten suchen, so empfiehlt sich eine Vorsortierung.

17.1 Die Problemstellung

17.1.1 Motivation

17-4

Wo taucht Sortieren überall auf?

Das *abstrakte* Sortierproblem taucht allein in der Molekularbiologie in vielen Kontexten auf:

- Eine Liste von Genen soll alphabetisch sortiert angezeigt werden.
- Ein Molekül soll gezeichnet werden. Dazu sollen die Atome »von hinten nach vorne« gezeichnet werden.
- Gene sollen nach Entfernung von einer Site sortiert werden.
- Vorhersagen sollen nach Wahrscheinlichkeit sortieren werden.
- In Gen- oder Proteindatenbanken sollen die Daten vorsortiert werden.

17.1.2 Problemvarianten

17-5

Was meint man mit »Sortieren« eigentlich?

Das einfachste Sortierproblem

Eingabe Array von Zahlen

Ausgabe Array mit denselben Zahlen, aber in der Reihenfolge so verändert, dass jede Zahl höchstens so groß wie ihr Nachfolger ist.

Eine »Veränderung in der Reihenfolge« nennt man auch *Permutation*.

17-6

Definition des allgemeinen Sortierproblems.

Das allgemeine Sortierproblem

Eingabe Array von Objekten, die sich vergleichen lassen

Ausgabe Eine Permutation der Objekte, so dass jedes Objekt in der neuen Reihenfolge kleiner oder gleich dem nachfolgenden ist.

17-7

Wünschenswerte Eigenschaften von Sortieralgorithmen.

Folgende Eigenschaften sind bei Sortieralgorithmen besonders wünschenswert:

1. Ein Verfahren ist *stabil*, falls sich die Reihenfolge von gleichen Elementen nicht ändert. Beispiel: Eine Adressliste wird nach Namen sortiert. Kommt vor der Sortierung Peter Müller aus Berlin vor Peter Müller aus Aachen, so soll dies nach der Sortierung immernoch der Fall sein.
2. Ein Verfahren ist *in-place*, falls es lediglich eine kleine Menge extra Speicher benötigt, falls es also keine Kopie des Arrays benötigt.
3. Das Verfahren sollte mit *möglichst wenigen* Vergleichen, Vertauschungen und Verschiebungen auskommen.

17.2 Sortieralgorithmen

17.2.1 Bubble-Sort

Der Bubble-Sort-Algorithmus

17-8

Idee

Wir sind fertig, wenn für je zwei aufeinanderfolgende Objekte gilt, dass das erste kleiner oder gleich dem zweiten ist. Also suchen wir nach Paaren, bei denen dies nicht der Fall ist, und tauschen sie aus.

Sortieren von Spielkarten nach dem Bubble-Sort-Algorithmus. Regie

Erste Version von Bubble-Sort

17-9

```
static void stupidSort (int[] array)
{
  int i = 0;
  while (i < array.length-1) {
    if (array[i] > array[i+1]) {
      // Korrigiere die Reihenfolge:
      swap(array, i, i+1);

      // Neustart
      i = 0;
    }
    else {
      i = i+1;
    }
  }
}

static void swap (int[] array, int i, int j)
{
  // Vertausche array[i] und array[j]
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

Verbesserungen der ersten Version

17-10

Es macht keinen Sinn, nach jeder Vertauschung wieder am Anfang zu beginnen. Stattdessen macht man einfach mit dem nächsten Element weiter. Dann ist am Ende eines Durchgangs das größte Element am Ende. Jede folgende Runde kann dann eins früher enden.

Zweite Version von Bubble-Sort

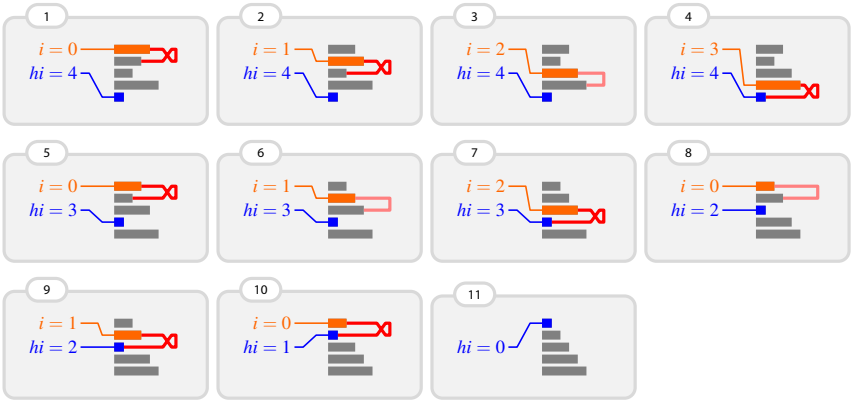
17-11

```
static void bubbleSort (int[] array)
{
  for (int hi = array.length-1; hi >= 0; hi = hi-1) {
    for (int i = 0; i < hi; i = i+1) {
      if (array[i] > array[i+1]) {
        swap(array, i, i+1);
      }
    }
  }
}
```

17-12

Ablauf von Bubble-Sort an einem Beispiel

Der zu sortierende Array sei `array == {4,3,2,5,1}`.



17-13

✎ Zur Übung

Bei einem Array der Länge n, wie viele

1. Vergleiche macht Bubble-Sort mindestens (grob)?
2. Vergleiche macht Bubble-Sort höchstens (grob)?
3. Vertauschungen macht Bubble-Sort mindestens (grob)?
4. Vertauschungen macht Bubble-Sort höchstens (grob)?

17-14

Vor- und Nachteile von Bubble-Sort

Vorteile

+ Einfach zu programmieren.
+ Einfach zu verstehen.
+ Kann leicht modifiziert werden, so dass er bei sortierten Daten sehr schnell ist.
+ In-place und stabil.

Nachteile

– Bei zufälligen Daten langsam, da viele Vergleiche.
– Bei fast sortierten Daten trotzdem langsam.

17.2.2 Selection-Sort

17-15

Der Selection-Sort-Algorithmus

Idee

Wir wollen möglichst wenig vertauschen. Deshalb suchen wir zunächst das kleinste Element im Array und tauschen es an die erste Stelle. Im Rest suchen wir dann wieder das kleinste und tauschen es an die zweite Stelle, und so weiter.

Regie

Sortieren von Spielkarten nach dem Selection-Sort-Algorithmus.

Selection-Sort

17-16

```
static void selectionSort (int[] array)
{
  for (int pos = 0; pos < array.length-1; pos++) {
    // Finde erstes Minimum ab Position pos
    int min = pos;

    for (int i = pos+1; i<array.length; i++) {
      if (array[min] > array[i]) {
        min = i;
      }
    }

    swap(array,pos,min);
  }
}
```

Ablauf von Selection-Sort an einem Beispiel

17-17

Der zu sortierende Array sei `array == {4,3,2,5,1}`.

Vor- und Nachteile von Selection-Sort

17-18

Vorteile

+ Einfach zu verstehen.
+ Minimale Anzahl an Vertauschungen.
+ In-place.

Nachteile

– Etwas aufwändiger zu programmieren.
– Immer viele Vergleiche, selbst bei sortierten Daten.
– Nicht stabil.

17.2.3 Insertion-Sort

17-19

Der Insertion-Sort-Algorithmus

Idee

Wir halten den ersten Teil des Arrays immer sortiert. Um den sortierten Teil des Arrays um ein Element zu erweitern, tauschen wir dies so lange nach links, bis es am Ziel angekommen ist.

Regie

Sortieren von Spielkarten nach dem Insertion-Sort-Algorithmus.

17-20

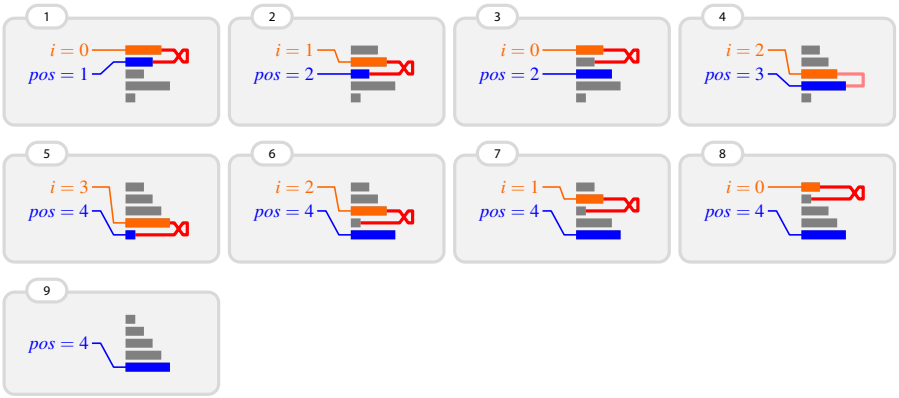
Insertion-Sort

```
static void insertionSort (int[] array)
{
  for (int pos = 1; pos < array.length; pos++) {
    int i = pos-1;
    while (i >= 0 && array[i] > array[i+1])
    {
      swap(array,i,i+1);
      i = i - 1;
    }
  }
}
```

17-21

Ablauf von Insertion-Sort an einem Beispiel

Der zu sortierende Array sei `array == {4,3,2,5,1}`.



17-22

Vor- und Nachteile von Insertion-Sort

Vorteile

+ Einfach zu verstehen.
+ Sehr schnell bei sortierten und fast sortierten Daten.
+ In-place und stabil.

Nachteile

– Bei zufälligen Daten viele Vergleiche und Vertauschungen.

17-23

✎ Zur Übung

Bei einem Array der Länge n, wie viele

1. Vergleiche macht Insertion-Sort mindestens (grob)?
2. Vergleiche macht Insertion-Sort höchstens (grob)?
3. Vertauschungen macht Insertion-Sort mindestens (grob)?
4. Vertauschungen macht Insertion-Sort höchstens (grob)?

17.3 *Untere Schranke für die Vergleichszahl

Wie viele Vergleiche werden mindestens benötigt?

17-24

Bubble-, Insertion- und Selection-Sort benötigten *grob* $n^2/2$ Vergleiche im schlimmsten Fall. Dies ist nicht optimal, Merge-Sort benötigt lediglich *grob* $n \log_2 n$ Vergleiche. Man kann sicherlich nicht mit weniger als mit $n-1$ Vergleichen auskommen. Wie viele Vergleiche benötigt also ein *optimaler* Algorithmus?

Die untere Schranke für die Anzahl der Vergleiche.

17-25

▶ Satz

Jeder Sortieralgorithmus, der lediglich Vergleiche zum Sortieren nutzt, benötigt mindestens $n \log_2 \frac{n}{e}$ *Vergleiche, um n Werte zu sortieren.*

Sortieralgorithmen wie Merge-Sort sind also optimal.

Beweis der unteren Schranke

17-26

Sei A ein beliebiges Sortieralgorithmus. Für jede Permutation π der Zahlen von 1 bis n betrachten wir die Folge der Vergleiche, die der Algorithmus durchführt. Jeder Vergleich liefert entweder »kleiner« oder »größer«; jede Antwortfolge entspricht also einem Bitstring. Sind zwei Permutationen unterschiedlich, so muss die Folge der gelieferten Antworten auf die Vergleiche unterschiedlich sein.
Es gibt $n!$ Permutationen, aber nur $2^{l+1}-1$ Bitstrings der Länge höchstens l. Es muss also, damit alle Permutationen durch einen anderen Bitstring repräsentiert werden, gelten:

$$2^{l+1} - 1 \geq n!.$$

Dies liefert

$$l \geq \log_2(n!+1) - 1.$$

Die Stirling-Approximation der Fakultät liefert nun

$$l \geq \log_2 \left(\sqrt{2\pi n} \left(\frac{n}{e} \right)^n + 1 \right) - 1 \geq n \log_2 \frac{n}{e}.$$

Zusammenfassung dieses Kapitels

▶ Sortieren

17-27

Sortieren ist das Problem, für eine Liste von Objekten eine *Permutation* zu finden, so dass sie in monoton wachsender Reihenfolge sind. Man muss mindestens $n \log_2(n/e)$ Vergleiche durchführen, um n Objekte zu sortieren. Die Algorithmen Bubble-Sort, Selection-Sort und Insertion-Sort benötigen alle schlimmstenfalls mindestens $n^2/2$ Vergleiche.

▶ Bubble-Sort

So lange noch zwei Objekte nebeneinander in der falschen Reihenfolge sind, vertausche sie.

▶ Selection-Sort

Für alle Positionen i tue nacheinander: Finde das kleinste Element ab Position i und tausche es an Stelle i.

▶ Insertion-Sort

Für alle Positionen i tue nacheinander: Tausche das Element an Stelle i so lange mit seinen Vorgängern, bis die Element von 1 bis i in sortierter Reihenfolge sind.

Übungen zu diesem Kapitel

Übung 17.1 Sortieralgorithmen implementieren, schwer

Implementieren Sie eine Methode

```
static void sortieren( int[] heuhaufen, int start, int end )
```

die das Array `heuhaufen` mit einem der Algorithmen aus der Vorlesung sortiert. Dabei soll allerdings der Algorithmus so erweitert werden, dass das Array nur im Bereich `start` bis `end` sortiert wird. Wie lange rechnet Ihr Sortierverfahren auf dem Zufallsarray von Übung 16.1?

Übung 17.2 Zeitmessung, schwer

Mit `long jetzt = System.currentTimeMillis();` können Sie herausfinden, wie viele Millisekunden seit dem 1. Januar 1970 vergangen sind. Messen Sie damit die Laufzeit Ihres Sortierverfahrens. Erstellen Sie eine Tabelle, die die Laufzeit für Array-Bereiche der Länge 10, 100, 1000 usw. angibt (so lange Ihr Speicher oder Ihre Geduld reicht).

Prüfungsaufgaben zu diesem Kapitel

Übung 17.3 Sortieralgorithmen verstehen, leicht, typische Klausuraufgabe

Gegeben sei ein Array mit folgenden Einträgen:

$$61,\ 30,\ 36,\ 77$$

Sortieren Sie das Array mit Bubble Sort, Insertion Sort und Selection Sort. Geben Sie dabei jeweils den Zustand des Arrays nach einem Durchlauf der äußeren Schleife an.

18-1

Kapitel 18
Komplexität von Problemen

Werden wir die Lösung in einer Sekunde oder in einem Jahr haben?

Lernziele dieses Kapitels

1. Das Laufzeitverhalten der wichtigsten Algorithmen kennen
2. Das Laufzeitverhalten einfacher Programme abschätzen können
3. Die Gruppierung von Problemen nach groben Laufzeitklassen verstehen

18-2

Inhalte dieses Kapitels

18.1	Die Laufzeit von Programmen	150
18.1.1	Welches Verfahren ist schneller?	150
18.1.2	Laufzeitbestimmung I	150
18.2	*O*-Klassen	151
18.2.1	Idee	151
18.2.2	Definition	151
18.2.3	Laufzeitbestimmung II	153
18.3	Einfache und schwierige Probleme	153
18.3.1	Einfache Probleme	153
18.3.2	Schwierige Probleme	154
	Übungen zu diesem Kapitel	155

Worum es heute geht

Bis jetzt ist unser Repertoire an Algorithmen zwar noch etwas bescheiden, aber eines zeichnet sich schon ab: Unterschiedliche Algorithmen für ein und dasselbe Problem können unterschiedlich schnell sein. Dabei sind absolute Aussage wie »Binäre Suche ist immer schneller als Linear Suche« in der Regel falsch, denn bei manchen Eingaben ist der eine Algorithmus schneller, bei anderen der andere.

In diesem Kapitel finden Sie eine kleine Einführung in die Theorie der Geschwindigkeit von Algorithmen. Ziele dieser Theorie sind:

1. Sie soll möglichst unabhängig von konkreter Hardware sein. Dass ein Supercomputer Zahlen schneller sortiert als Ihr Telefon, ist wenig überraschend und sagt auch wenig darüber aus, wie schnell ein bestimmter Algorithmus ist.
2. Sie soll ein allgemeines Bild von der Geschwindigkeit eines Algorithmus liefern. Dass ein Algorithmus bei bestimmten Spezialfällen sehr gut funktioniert, ist zwar schön für den Algorithmus und man kann ihn dafür auch belobigen, jedoch sagt dies wiederum wenig über sein allgemeines Verhalten aus.

Es hat sich herausgestellt, dass die so genannte *O-Notation* besonders geeignet ist, die Geschwindigkeit von Algorithmus zu beschreiben. Grob gesprochen bedeutet »der Algorithmus hat Laufzeit $O(n^2)$«, dass er bei Eingaben der Größe n grob n^2 Rechenschritte benötigt *unabhängig von der Hardware*.

Gibt es eigentlich noch andere interessante »Ressourcen« bei Algorithmen außer der *Zeit*? Nicht ganz unwichtig ist ebenfalls der Speicherbedarf: Ein `a = new int[100000000]` ist schnell hingeschrieben, der Rechner muss jetzt aber erstmal sehen, wo er die angeforderten 400 Megabyte herzaubert. Trotzdem ist die Geschwindigkeit eines Algorithmus seine mit Abstand wichtigste Eigenschaft, denn *time is money*. Wir werden in dieser Veranstaltung nur die Zeit als »kritische Eigenschaft« untersuchen.

18.1 Die Laufzeit von Programmen

18.1.1 Welches Verfahren ist schneller?

18-4

Wie kann man die Rechenzeit von Algorithmen vergleichen?

Fragestellung

Sollte man lieber lineare Suche oder binäre Suche einsetzen?

Probleme bei der Beantwortung

- Je nach *Art der Eingabedaten* ist die Rechenzeit erheblich unterschiedlich: Ist das zu suchende Element sehr nahe am Anfang, so ist lineare Suche schneller. Ist das zu suchende Element in der Mitte, so ist binäre Suche schneller.
- Je nach *verwendeter Hardware* ist die Rechenzeit erheblich unterschiedlich.
- Je nach *Können des Programmierers* ist die Rechenzeit erheblich unterschiedlich.

18-5

Schnellere Computer oder schnellere Algorithmen?

Benutzt man *lineare Suche*, so dauert das Suchen nach einem Element schlimmstenfalls $t_1 \cdot n$ Sekunden, wobei t_1 von der Hardware abhängig ist. Benutzt man *binäre Suche*, so dauert das Suchen nach einem Element schlimmstenfalls $t_2 \cdot \lceil \log_2 n \rceil$ Sekunden, wobei t_2 natürlich auch von der Hardware abhängig ist.

✎ Zur Diskussion

Typischerweise ist t_2 größer als t_1. Was ist nun besser – lineare oder binäre Suche?

18-6

Vergleich der Laufzeiten von linearer und binärer Suche.

- Programmierer Anton implementiert lineare Suche in Maschinensprache auf einem Pentium 4 GHz. Dann ist grob $t_1 = 0{,}25\,\text{ns}$.
- Programmiererin Beatrice implementiert binäre Suche in Java auf ihrem Handy mit 100 MHz. Dann ist grob $t_2 = 2\,\mu\text{s} = 2000\,\text{ns}$.

Eingabegröße	*lineare Suche*	*binäre Suche*
10	2,5 ns	8 µs
100	25 ns	14 µs
1.000	250 ns	20 µs
10.000	2,5 µs	28 µs
100.000	25 µs	34 µs
1.000.000	250 µs	40 µs

18-7

Moral

Moral

Ein guter Algorithmus schlägt einen schlechten Algorithmus, selbst wenn der gute Algorithmus schlecht implementiert wird und der schlechte gut implementiert wird. Die Terme n und $\log n$ in den Laufzeiten sind viel wichtiger als die Konstanten t_1 und t_2.

18.1.2 Laufzeitbestimmung I

18-8

Der Begriff der Laufzeit.

▶ Definition: Maschinenmodell

Ein *Maschinenmodell* legt fest, wie lange eine Addition dauert, wie lange eine Multiplikation dauert, wie lange eine Zuweisung dauert und so weiter.

▶ Definition: Laufzeiten

Für ein Programm und eine Eingabe x bezeichnet

- $T(x)$ die Anzahl der Rechenschritte, die das Programm bei Eingabe x verbraucht;
- $T(l)$ die Anzahl Rechenschritte, die das Programm bei Eingaben der Länge l höchstens braucht.

Die *Eingabelänge* ist bei Strings die Länge des Strings und bei Arrays die Anzahl der Elemente.

Beispiel einer Laufzeitbestimmung. 18-9

```
static boolean containsZero (String s) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '0') {
      return true;
    }
  }
  return false;
}
```

Die verschiedenen Befehle benötigen unterschiedlich lange.
Seien t_1 bis t_6 die Zeiten, die benötigt werden 1) für eine Zuweisung, 2) für einen Vergleich, 3) von der Methode `length`, 4) für die Berechnung von `i++`, 5) von der Methode `charAt` und 6) vom `return`-Befehl.

- $T(\texttt{ABCD05}) = t_1 + 10t_2 + 5t_3 + 4t_4 + 5t_5 + t_6$.
- $T(\texttt{ABCDEF}) = t_1 + 13t_2 + 7t_3 + 6t_4 + 6t_5 + t_6$.
- $T(6) = t_1 + 13t_2 + 7t_3 + 6t_4 + 6t_5 + t_6$.
- $T(l) = t_1 + (2l+1)t_2 + (l+1)t_3 + lt_4 + lt_5 + t_6$.

18.2 O-Klassen

18.2.1 Idee

Wie genau sollten wir Laufzeiten messen? 18-10

Die Laufzeit das Programm lautete $T(l) = t_1 + (2l+1)t_2 + (l+1)t_3 + lt_4 + lt_5 + t_6$. Wir hatten schon gesehen, dass die *genauen Werte* der Konstanten *eher unerheblich* sind. Wir führen deshalb eine spezielle Notation ein, mit der wir uns *auf das Wesentliche konzentrieren* können: Die *O-Klassen*. Die zentralen Ideen sind:

1. Wir ignorieren *Konstanten*, mit denen *multipliziert wird*.
2. Uns interessiert nur Laufzeiten bei *großen Eingaben*.

18.2.2 Definition

Definition der O-Klassen. 18-11

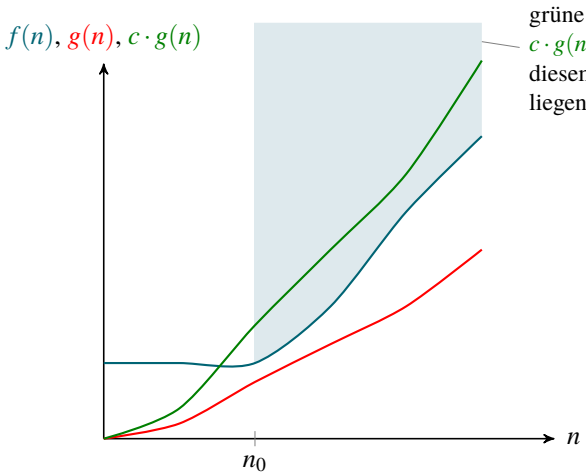
▶ Definition: O-Klasse
Sei $g\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann ist die O-Klasse $O(g)$ die Menge aller Funktionen $f\colon \mathbb{N} \to \mathbb{N}$, für die

- es eine Konstante c und
- eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \leq c \cdot g(n)$.

Merke
$f \in O(g)$ bedeutet, dass für *genügend große n* und einen *genügend großen Faktor* gilt, dass $g(n)$ mal diesen Faktor $f(n)$ dominiert.

18-12

Veranschaulichung von $f \in O(g)$



18-13

Beispiele zu O-Klassen.

Beispiel
Sei $g(n) = n^2$ und $f(n) = 3 + 17n^2$. Dann ist $f \in O(g)$.
(Wähle $c = 1000$ und $n_0 = 1000$. Dann ist sicherlich $3 + 17n^2 \leq cn^2$.)

Beispiel
Sei $g(n) = n^2$ und $f(n) = 1000\sqrt{n}$. Dann ist $f \in O(g)$.
(Wähle $c = 1000$ und $n_0 = 0$.)

Beispiel
Sei $g(n) = n$ und $f(n) = n^2$. Dann ist $f \notin O(g)$.

Beispiel
Sei $g(n) = \log_2 n$ und $f(n) = \log_2(n^2)$. Dann ist $f \in O(g)$.
(Es gilt $\log_2(n^2) = 2\log_2 n$.)

18-14

✎ **Zur Übung**
Geben Sie für folgende f und g an, ob $f \in O(g)$ gilt.

1. $f(n) = n^4$ und $g(n) = n^5$.
2. $f(n) = n^5$ und $g(n) = n^4$.
3. $f(n) = n^5$ und $g(n) = n^4 \log n$.
4. $f(n) = 3n^5$ und $g(n) = 2n^5$.
5. $f(n) = \log_2 n$ und $g(n) = 1$.
6. $f(n) = \log_2 n$ und $g(n) = \log_3 n$.
7. $f(n) = \log_3 n$ und $g(n) = \log_2 n$.
8. $f(n) = 2^n$ und $g(n) = 3^n$.
9. $f(n) = 3^n$ und $g(n) = 2^n$.

18-15

Zur Schreibweise von O-Klassen.

- Man schreibt einfach $O(n^2)$ für »$O(g)$, wobei g die Funktion $g(n) = n^2$ ist«.
- Man schreibt auch $O(g)$, wenn g von mehreren Parametern abhängt.
  So bedeutet $O(n^2 m)$ »die Laufzeit ist für hinreichend große n und m höchstens $cn^2 m$ für eine Konstante c«.
- Man schreibt auch $f(n) = O(n^2)$ statt $f \in O(n^2)$.

18-16

Die wichtigsten O-Klassen.

Die wichtigsten O-Klassen und ihre Inklusionsbeziehungen:

$$\begin{aligned} O(1) &\subsetneq O(\log n) \subsetneq O(\sqrt{n}) \\ &\subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n \log^2 n) \\ &\subsetneq O(n^2) \subsetneq O(n^3) \\ &\subsetneq O(2^n) \subsetneq O(3^n). \end{aligned}$$

18.2.3 Laufzeitbestimmung II

Wiederholung der Laufzeitbestimmung. 18-17

```
boolean containsZero (String s) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '0') {
      return true;
    }
  }
  return false;
}
```

$T(l) = t_1 + (2l+1)t_2 + (l+1)t_3 + lt_4 + lt_5 + t_6$. Mit Hilfe der O-Notation können wir kurz sagen $T \in O(l)$ oder auch $T(n) = O(n)$. Der Aufwand ist also *linear*.

✎ Zur Übung 18-18

1. Bestimmen Sie die O-Klasse des Zeitaufwands von folgendem Programm:

```
void prefix_sums (double[] vector) {
  int sum = 0;
  for (int i = 0; i < vector.length; i++) {
    sum = sum + vector[i];
    vector[i] = sum;
  }
}
```

2. Bestimmen Sie die O-Klasse des Zeitaufwands von folgendem Programm (schwierig):

```
double puzzle (double[] vector) {
  int i = 1;
  while (3*i < vector.length) {
    i = i*3;
  }
  return vector[i];
}
```

18.3 Einfache und schwierige Probleme

18.3.1 Einfache Probleme

Was sind einfache Probleme? 18-19

In der *Praxis* gelten alle Probleme als gut lösbar, die sich in Zeit bis $O(n^3)$ lösen lassen. Die *»versteckten Konstanten«* sollten allerdings nicht zu groß sein. In der *Theorie* gelten alle Probleme als gut lösbar, die sich in Zeit $O(n^k)$ lösen lassen für irgendein k.

Beispiele: Effizient lösbare Probleme

- Binäre und lineare Suche
- Sortieren
- Addieren, Multiplizieren, Dividieren
- Matrix-Multiplikation und Matrix-Invertierung
- Kürzeste Wege finden in Graphen
- Berechnen eines Bildes in einer Animation

18.3.2 Schwierige Probleme

18-20

Was sind schwierige Probleme?

Manche Probleme gelten als *praktisch nicht gut lösbar*. Für solche Probleme ist *kein Algorithmus mit einer Laufzeit von* $O(n^k)$ bekannt.

Beispiel: Das Partitionsproblem

Gegeben sind n Zahlen. Kann man sie so in zwei Teilmengen aufteilen, dass die Summen gleich sind?

Es gibt 2^n Möglichkeiten, die Zahlen aufzuteilen, was einen Algorithmus mit Laufzeit $O(2^n)$ liefert. Was ist die Laufzeit des *besten Algorithmus* für das Partition-Problem?

18-21

Wir mögen keine exponentielle Laufzeit.

Programm 2010

Programmiererin Ada hat den Algorithmus für das Partition-Problems in Maschinensprache implementiert. Auf einem Pentium 4GHz kann sie damit in einer Stunde Eingaben bis $n \approx \log_2(360 \cdot 10^9) \approx 38$ bearbeiten.

Programm 2020

10 Jahre später gibt es einen Quad-Core Rechner mit 40GHz. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(4 \cdot 3600 \cdot 10^9) \approx 44$ bearbeiten.

Programm 2030

Wieder 10 Jahre später bekommt sie einen Supercomputer mit 10.000 Prozessoren mit je 1000 GHz. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(3600 \cdot 10^{15}) \approx 62$ bearbeiten.

18-22

Wir mögen wirklich keine exponentielle Laufzeit.

Original C. Flammarion, public domain. Copyright coloring by Hugo Heikenwaelder, Creative Commons Attribution Sharealike License

Programm 2500

Ada baut einen großen Rechner. Jedes Atom im Universum ($\approx 10^{80}$) rechnet in jeder Planck-Zeit ($\approx 5{,}39 \cdot 10^{-43}$s) einen Rechenschritt. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(5{,}39 \cdot 3600 \cdot 10^{80+43}) \approx 423$ bearbeiten.

Moral

Eingaben mit 500 Zahlen wird man *niemals (!)* mit dem Algorithmus für das Partition-Problem lösen können.

18-23

Die Komplexitätsklassen P und NP.

In der *Theoretischen Informatik*, speziell in der *Komplexitätstheorie*, untersucht man, welche Probleme einfach und welche schwierig sind. Die *einfachen Probleme* (solche, die sich in Zeit $O(n^k)$ lösen lassen) bilden die *Klasse P*. Nimmt man noch eine Reihe *anscheinend schwieriger Probleme* hinzu, so erhält man die *Klasse NP*. Die Frage, ob P = NP gilt, ist die wichtigste ungelöste Frage der Theoretischen Informatik.

Zusammenfassung dieses Kapitels

18-24

▶ O-Klasse

Die *O-Klasse* $O(g)$ ist die Menge aller Funktionen $f\colon \mathbb{N} \to \mathbb{N}$, für die

- es eine Konstante c und
- eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \leq c \cdot g(n)$.

Man ignoriert also *konstante Faktoren* und *(zu) kurze Eingaben.*

▶ Die wichtigsten O-Klassen und ihre Inklusionsbeziehungen

$$\begin{aligned} O(1) \subsetneq O(\log_3 n) = O(\log_2 n) = O(\log n) \subsetneq O(\sqrt{n}) \\ \subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n \log^2 n) \\ \subsetneq O(n^2) \subsetneq O(n^3) \\ \subsetneq O(2^n) \subsetneq O(3^n). \end{aligned}$$

Hierbei gilt alles bis $O(n^3)$ als »noch vertretbar«.

Übungen zu diesem Kapitel

Übung 18.1 *O*-Klassen bestimmen, mittel

Für zwei Bitstrings (Strings aus Nullen und Einsen) gleicher Länge ist ihr Hamming-Abstand definiert als die Anzahl von Positionen, in denen sich die Strings unterscheiden (siehe Übung 15.4).

Die folgende Java-Methode berechnet für eine Liste von m Bitstrings der Länge n den maximalen Hamming-Abstand für Paare von Bitstrings aus der Liste.

```
public static int compute_max_distance
                    (String[] list_of_bitstrings) {
  int maxdist = 0;

  // Anzahl der Bitstrings
  int m = list_of_bitstrings.length;

  // Länge der Bitstrings; Annahme: alle gleich lang
  int n = list_of_bitstrings[0].length();

  for (int i = 0; i < m; i++) {
    for (int j = i+1; j < m; j++) {
      // alle Paare (i,j) mit i < j checken

      int counter = 0;
      // zählt Positionen mit unterschiedlichen Einträgen
      for (int k = 0; k < n; k++) {
        // Schleife berechnet Abstand der Strings i und j
        if (list_of_bitstrings[i].charAt(k) !=
            list_of_bitstrings[j].charAt(k)) {
          counter++;
        }
      }
      if (counter > maxdist) {
        maxdist = counter;     // Erhöht maxdist,
      }                        // wenn neuer Abstand größer
    }
  }
  return maxdist;
}
```

Analysieren Sie die Laufzeit des Programms in Abhängigkeit von den Parametern n und m. Geben Sie eine passende O-Klasse für die Laufzeit an.

Übung 18.2 Laufzeiten experimentell bestimmen, schwer

Prüfen Sie experimentell die Laufzeit der Methode aus der vorherigen Übung in Abhängigkeit der Parameter n und m. Das folgende Programm dient diesem Zweck:

```
class HammingTest{

  public static String[] generate_bitstrings(int n, int m) {
    String[] list_of_bitstrings = new String[m];
    for (int i = 0; i < m; i++){
      // Liste ab 0 mit Bitstrings füllen
      String bitstring = "";
      for (int k = 0; k < n; k++) {
        if (Math.random() > .5) {   // zufällig 0 oder 1
          bitstring = bitstring + "0";
        }
        else {
          bitstring = bitstring + "1";
        }
      }
      list_of_bitstrings[i] = bitstring;
    }
    return list_of_bitstrings;
  }

  /* Hier muss compute_max_distance eingefügt werden. */

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]); // Länge der Bitstrings
    int m = Integer.parseInt(args[1]); // Anzahl der Bitstrings
    String[] list_of_bitstrings = generate_bitstrings(n,m);
```

```
    long startTime = System.currentTimeMillis();
    int max = compute_max_distance(list_of_bitstrings);
    long runningTime = System.currentTimeMillis() - startTime;
    System.out.println("Die␣Laufzeit␣für␣n=␣" + n
      + "␣und␣m=␣" + m + "␣beträgt␣etwa␣"
      + runningTime + "␣Millisekunde(n).");
  }
}
```

Die Methode `generate_bitstrings` erzeugt zufällig Bitstring-Listen. Die Eingabeparameter sind n, die Länge der Strings, und m, die Anzahl der Strings.

Die Methode `main` erzeugt für die eingelesene Stringlänge und -anzahl eine zufällig erzeugte Stringliste und berechnet dann maximalen Hamming-Abstands für die Liste. Es wird (eine Annäherung an) die Rechenzeit für `compute_max_distance` gestoppt.

Die so ermittelte Rechenzeit für festes n und m kann durchaus schwanken. Erweitern Sie das Programm so, dass es die durchschnittliche Laufzeit für (zum Beispiel) 20 Durchläufe ermittelt. Lassen Sie das Programm für ausreichend viele passende Werte für n und m laufen. Wird die Laufzeitabschätzung aus der vorherigen Aufgabe bestätigt?

19-1

Kapitel 19
Einführung zur Rekursion
Kasparov versus Deep Blue

Lernziele dieses Kapitels

1. Das Konzept der Rekursion verstehen
2. Rekursive Methoden implementieren können für einfache Probleme

19-2

Inhalte dieses Kapitels

19.1 Rekursion 158
19.1.1 Der Begriff . . . 158
19.1.2 Java-Syntax der Rekursion . . . 159

19.2 Beispiele 159
19.2.1 Fakultät . . . 159
19.2.2 Fibbonacci-Zahlen . . . 160
19.2.3 Die Türme von Hanoi . . . 160
19.2.4 Binäre Suche . . . 161

19.3 *Wechselseitige Rekursion 161
19.3.1 Tic-Tac-Toe . . . 161
19.3.2 Schach . . . 162

Übungen zu diesem Kapitel 162

Worum es heute geht

In einem Betrieb macht die Chefin nicht alles selbst. Die Kunst, einen Konzern zu führen, ist zu *delegieren*. Wenn ein Konzern eine neue Strategie sucht – sagen wir zur Vermarktung von Klingeltönen –, wird die Chefin sicherlich erstmal ihren Assistenten einspannen, der eine solche vorbereiten soll. Dieser wiederum macht natürlich auch nicht die gesamte Arbeit im Konzern, sondern findet schnell eine Abteilung und damit einen Abteilungsleiter, der ihm helfen soll. Dieser wiederum hat seinen Stab, der Zugriff auf die Fachkräfte hat, die ganz am Ende die Arbeit einem Praktikanten geben, der am Wochenende seine Freundin bezirzt, ihm doch ein paar Ideen zu nennen, wie man Klingeltöne besser verkaufen könnte. Die geniale Idee der Freundin, das Geschäft mit Klingeltönen aus ethischen Gründen und aus Gründen der akustischen Hygiene einfach einzustellen, wandert dann die ganzen Hierarchiestufen wieder nach oben, jedesmal vielleicht leicht verändert, bis die Konzernchefin auf den nächsten Aktionärsversammlung die neue Firmenstrategie vorstellen kann, sich in Zukunft mehr auf die Vermarktung von Telefonen mit Vibrationsalarm zu konzentrieren.

Ganz ähnlich funktioniert Rekursion. Der Computer soll ein Problem lösen, aber anstatt dieses direkt zu lösen, wird erstmal jemand anderes (eine Methode) mit der Lösung des Problems beauftragt. Diese Method delegiert das Problem wieder, dann wird es wieder delegiert und so fort. Hierbei gibt es Folgendes zu beachten:

- Bei einer Rekursion wird in der Regel gar nicht an »andere Leute« delegiert, sondern eine Methode *ruft sich selbst auf*.
- Das wäre natürlich reichlich nutzlos und würde zu einer unendlichen Kette von Selbstdelegationen führen, wenn nicht bei jeder Delegation das Problem *etwas einfacher würde*.
- Wenn das Problem durch mehrfaches (manchmal millionenfaches) Delegieren trivial geworden ist, muss man die Rekursion *abbrechen und das Problem direkt lösen*.

Wenn man ein rekursives Programm zum ersten Mal sieht, wird man sich des Gefühls nicht erwehren können, dass hier »geschummelt« wird. Anscheinend löst man ein Problem, indem

man stattdessen genau dieses Problem löst; man bekommt leicht einen Knoten im Gehirn. Mit etwas Übung löst sich aber dieser Knoten und Sie werden bald rekursiv schummeln können.

19-4 **Kasparov versus Deep Blue.**

Copyright by IBM, free for non-commercial use

Unknown author, public domain

Im Jahr 1997 hat Deep Blue gegen Kasparov im Schach gewonnen. Wie ging das?

19.1 Rekursion

19.1.1 Der Begriff

19-5 **Was ist Rekursion?**

Rekursion in der Programmierung stammt ursprünglich aus der Mathematik. Die Idee ist, ein Problem *»auf sich selbst«* zurückzuführen. Wichtig ist aber, dass man bei der Rückführung

1. *einfacher* wird und
2. es eine *Abbruchbedingung* gibt.

19-6 **Ein praktisches Beispiel: Fahrtplanung.**

Problemstellung
Gesucht: Zugverbindung von Simbach nach Lübeck.

Da es einen Regionalexpress von Hamburg nach Lübeck gibt, *reduziert* sich das Problem auf ein *einfacheres*:

Einfachere Problemstellung
Gesucht: Zugverbindung von Simbach nach Hamburg.

Da ein ICE von München nach Hamburg fährt, *reduziert* sich das Problem noch weiter:

Noch einfachere Problemstellung
Gesucht: Zugverbindung von Simbach nach München.

Hier fährt direkt ein Zug.

19-7 **Ein mathematisches Beispiel: Rekursive Berechnung einer Summe.**

Problemstellung
Gesucht: Summe der ersten 42 Zahlen.

Diese Summe erhalten wir, wenn wir auf die Summe der ersten 41 Zahlen 42 addieren. Also *reduziert* sich das Problem auf:

Einfachere Problemstellung
Gesucht: Summe der ersten 41 Zahlen.

Diese Summe erhalten wir, wenn wir auf die Summe der ersten 40 Zahlen 41 addieren. Also *reduziert* sich das Problem auf:

Noch einfachere Problemstellung
Gesucht: Summe der ersten 40 Zahlen.

Und so weiter, bis die Summe der ersten wenigen Zahlen (zum Beispiel der ersten drei Zahlen) bekannt ist.

19.1.2 Java-Syntax der Rekursion

Rekursion in Java. 19-8

- Eine Methode darf auch sich selbst aufrufen.
- Es gibt keine spezielle Syntax hierfür.
- Jeder rekursive Aufruf erzeugt einen neuen Scope.

Rekursive Berechnung der Summe in Java. 19-9

```
static int sum(int n)
{
  if (n == 1) {           // Abbruchbedingung
    return 1;
  }
  else {
    return sum(n-1) + n;
      // Rückführung auf ein einfacheres Problem
  }
}
```

Jede Rekursion hat zwei Elemente:

1. Die Abbruchbedingung.
2. Die Rückführung auf ein einfacheres Problem.

19.2 Beispiele

19.2.1 Fakultät

Rekursive Berechnung der Fakultät. 19-10

Die *Fakultät* einer Zahl n ist das Produkt der ersten n Zahlen.

```
static int fac(int n)
{
  if (n <= 1) {
    return 1;
  }
  else {
    return n*fac(n-1);
  }
}
```

Vergleich von Rekursion und Iteration. 19-11

Rekursive Lösung:

```
static int fac(int n)
{
  if (n <= 1) {
    return 1;
  }
  else {
    return n*fac(n-1);
  }
}
```

Iterative Lösung:

```
static int fac(int n)
{
  int product = 1;
  for (int i = 1; i <= n; i++) {
    product = product * i;
  }
  return product;
}
```

Es gab (gibt?) einen *Glaubenskrieg*, ob Rekursion oder Iteration besser ist. *Hier* ist Iteration besser, da schneller und eventuell klarer. In *komplexeren* Situationen ist aber Rekursion einfacher und natürlicher.

19-12

✎ Zur Übung

Geben Sie eine rekursive und eine iterative Methode an, die die Summe der ersten n Quadratzahlen berechnet.

19.2.2 Fibbonacci-Zahlen

19-13

✎ Zur Übung

Die Fibbonacci-Folge beginnt mit zweimal 1. Die nächste Zahl in der Folge ist immer die Summe der beiden vorherigen:
1, 1, 2, 3, 5, 8, 13, 21, ...
Geben Sie den Code einer rekursiven Methode an, die die n-te Fibbonacci-Zahl berechnet.

19.2.3 Die Türme von Hanoi

19-14

Steht der Weltuntergang unmittelbar bevor?

Author Marubatsu, public domain

Die Legende über die Türme von Hanoi besagt, dass sich die Mönche eines Klosters in Hanoi folgender Aufgabe widmen:
Sie bewegen 100 goldene Scheiben vom ersten von drei Stäben auf den dritten. Die Scheiben haben alle unterschiedliche Größen und es liegen immer kleinere auf größeren.
Wenn die Mönche die hundert Scheiben auf den dritten Stab geschichtet haben, so wird die Welt enden.

Regie

Lösen Sie das Türme-von-Hanoi-Problem für drei Scheiben.

19-15

Ein Lösungsalgorithmus für das Türme-von-Hanoi-Problem

- Eingabe: Scheibenanzahl, Start-Turm, Ziel-Turm.
- Ausgabe: Folge der Verschiebungen, um die gewünschte Anzahl Scheiben vom Start-Turm zum Ziel-Turm zu bewegen.

Algorithmus `calcMoves(int n, int from, int to)`

1. Wenn $n = 1$, so schiebe einfach die eine Scheibe von `from` nach `to`.
2. Ansonsten sei `extra` der verbleibende Turm (also weder `from` noch `to`).
3. Schiebe $n - 1$ Scheiben von `from` nach `extra`.
4. Schiebe eine Scheibe von `from` nach `to`.
5. Schiebe $n - 1$ Scheiben von `extra` nach `to`.

19-16

Der Hanoi-Algorithmus in Java

```
static void calculateMoves(int n, int from, int to)
{
  if (n == 1) {
    System.out.println ("Move␣top␣disk␣from␣" +
                        from + "␣to␣" + to + ".");
  }
  else {
    // Calculate other stack using evil trickery:
    int other_stack = 6 - from - to;
```

```
    calculateMoves(n-1, from, other_stack);

    System.out.println ("Move␣top␣disk␣from␣" +
                        from + "␣to␣" + to + ".");

    calculateMoves(n-1, other_stack, to);
  }
}
```

19.2.4 Binäre Suche

Binäre Suche mit Rekursion. 19-17

```
static int search (String[] strings,
                   String   value,
                   int      lower_bound,
                   int      upper_bound)
{
  if (lower_bound == upper_bound) {
    return lower_bound;
  }
  else {
    int mid_point = (lower_bound + upper_bound)/2;

    if (strings[mid_point].compare(value) < 0) {
      return search(strings,value,mid_point+1,upper_bound);
    }
    else {
      return search(strings,value,lower_bound,mid_point);
    }
  }
}
```

✎ Zur Übung
Identifizieren Sie alle rekursiven Aufrufe und den Rekursionsabbruch.

19.3 *Wechselseitige Rekursion

19.3.1 Tic-Tac-Toe

Das Tic-Tac-Toe-Spiel. 19-18

Das Spiel

Gespielt wird auf einem 3-mal-3-Feld. Gezogen wird abwechselnd, ein Spieler setzt Kreuze, der andere Kreise in die Felder. Falls einer der Spieler drei gleiche Symbole in einer Reihe erzeugt, gewinnt er.

Strategie für einen guten Zug

Kann ich mit einem Zug gewinnen, so ist dies ein *guter Zug*. Sonst schaue ich für alle möglichen Züge, bei welchem *dem andere Spieler* nur schlechte Züge bleiben, und nehme diesen Zug.

Tic-Tac-Toe-Algorithmus 19-19

Algorithmus `findOptimalComputerMove`

1. Falls ein Zug zum sofortigen Gewinn führt, gib diesen zurück.
2. Sonst tue folgendes:
   2.1 Berechne für jeden möglichen Computerzug den optimalen Zug des Menschen.
   2.2 Gib den Zug zurück, bei dem der optimale Zug des Menschen am schlechtesten ist.

Algorithmus `findOptimalHumanMove`

1. Falls ein Zug zum sofortigen Gewinn führt, gib diesen zurück.
2. Sonst tue folgendes:
   2.1 Berechne für jeden möglichen Menschenzug den optimalen Zug des Computers.
   2.2 Gib den Zug zurück, bei dem der optimale Zug des Computers am schlechtesten ist.

19.3.2 Schach

19-20

Wie funktionieren Schachprogramme?

Man kann Schachprogramme im Prinzip genauso wie Tic-Tac-Toe programmieren. Dann würde die Berechnung aber viel zu lange dauern (viele, viele Mal länger, als das Universum alt ist; auf einem Computer, der so groß wie das Universum ist). Deshalb wird die Rekursion nach einer gewissen Anzahl Schritte abgebrochen.
Und dann braucht man noch viele, viele Tricks.

Zusammenfassung dieses Kapitels

19-21

▶ **Rekursion**

Rekursion ist eine Programmiermethode, bei der Probleme gelöst werden, indem jede Eingabe auf die Lösung zu einer *einfacheren Eingabe* zurückgeführt wird.

▶ **Typischer Aufbau einer Rekursion in Java**

```
static ... rekursiveMethode (int n, ...)
{
  if (n <= 1) {
    return ...;
  } else {
    ...;
    ... rekursiveMethode(n-1, ...) ...;
    ...;
    return ...;
  }
}
```

Übungen zu diesem Kapitel

Übung 19.1 Rekursives Skalarprodukt, mittel

Schreiben Sie eine rekursive Methode

```
static int dot_product( int[] a, int[] b )
```

die das Skalarprodukt zweier Vektoren, gegeben als Arrays von Integern, berechnet. Zur Erinnerung: Das Skalarprodukt $\langle \vec{a}, \vec{b} \rangle$ zweier Vektoren $\vec{a} = (a_1, \ldots, a_n)^T$ und $\vec{b} = (b_1, \ldots, b_n)^T$ ist definiert als

$$\langle \vec{a}, \vec{b} \rangle = \sum_{i=1}^{n} a_i b_i$$

Beim Programmieren dürfen Sie davon ausgehen, dass die beiden übergebenen Arrays gleich lang sind.

Übung 19.2 Rekursives Spiel, schwer

Ein aus der Steinzeit übermittelter Vorläufer von Solitaire funktioniert wie folgt: Von einem Haufen Kieselsteine dürfen Sie in einem Zug jeweils 3 oder 7 Steine wegnehmen. Sie haben gewonnen, wenn am Ende kein Stein mehr übrig ist. Bleiben ein oder zwei Steine übrig, dann haben Sie verloren. Die Steinzeitmenschen konnten nicht sehr gut rechnen, aber auch ihnen war klar, dass diese Aufgabe für fast alle Kieselsteinhaufen, mit nur einigen Ausnahmen, lösbar ist.

Schreiben Sie eine rekursive Java-Methode, die für eine Zahl n von Kieselsteinen ermittelt, ob die Aufgabe lösbar ist oder nicht.

Übung 19.3 Rekursives Spiel lösen, schwer

Helfen Sie den Steinzeitmenschen aus der vorherigen Aufgabe und schreiben Sie eine Methode, die für eine Zahl n eine Folge von Zügen ausgibt, die die Kieselsteinaufgabe löst, falls das möglich ist. Hierzu empfiehlt es sich, ein oder zwei Hilfsmethoden zu verwenden. Die Ausgabe kann über den Befehl `System.out.println` oder mit einem `String`-Rückgabewert erfolgen.

Übung 19.4 Rekursive Exponentiation, leicht

Schreiben Sie folgende Methode, die `b` hoch `e` berechnet, rekursiv neu.

```
static int pow( int base, int exponent )
{
  int i;
  int ret = 1;
  for( i = 0 ; i < exponent ; i ++) {
    ret = ret * base;
  }
  return ret;
}
```

Übung 19.5 Rekursive Sternchen, leicht

Schreiben Sie folgende Methode, die `n` viele Sternchen ausgibt, rekursiv neu.

```
static String stars( int n )
{
  String ret = "";
  for( int i = 0 ; i < n ; i ++ ) {
    ret = ret + "*";
  }
  return ret;
}
```

Übung 19.6 Rekursive »String-Multiplikation«, leicht

Schreiben Sie folgende Methode, die `n` Mal den String `s` hintereinanderkettet, rekursiv neu.

```
static String repeat( String s, int n )
{
  String ret = "";
  for( int i = 0 ; i < n ; i ++ ) {
    ret = ret + s;
  }
  return ret;
}
```

Übung 19.7 Rekursives Umdrehen, leicht

Schreiben Sie folgende Methode, die einen String umdreht, rekursiv neu.

```
static String revert( String s )
{
  String ret = "";
  for( int i = s.length()-1 ; i >= 0 ; i -- ) {
    ret = ret + s.charAt(i);
  }
  return ret;
}
```

Übung 19.8 Rekursives Trimmen, leicht

Schreiben Sie folgende Methode, die Leerzeichen und Tabulatorzeichen am Anfang entfernt, rekursiv neu.

```
static String trim( String s ){
  int i = 0;
  while( s.charAt(i) == '␣' || s.charAt(i) == '\n' || s.charAt(i) == '\t'
      ) {
    i++;
  }
  return s.substring(i);
}
```

Übung 19.9 Rekursives Zählen, leicht

Schreiben Sie folgende Methode, die die Anzahl von kleinen und großen A's in `s` berechnet, rekursiv neu.

```
static int count_a( String s )
{
  int ret = 0;
  for( int i = 0 ; i < s.length() ; i ++ ) {
    if( s.charAt(i) == 'a' || s.charAt(i) == 'A' ) {
      ret ++;
    }
  }
  return ret;
}
```

Übung 19.10 Rekursiver Hammingabstand, schwer

Schreiben Sie folgende Methode, die den Hammingabstand zweier Strings (Anzahl der Stellen, wo sie sich unterscheiden) berechnet, rekursiv neu. Wenn die String unterschiedlich lang sind, wird die Differenze dem Abstand hinzugezählt.

```
static int hamming_distance( String a, String b )
{
  int ret = 0;
  int min_length;
  if( a.length() < b.length() ) {
    min_length = a.length();
    ret = b.length() - a.length();
  }
  else {
    min_length = b.length();
    ret = a.length() - b.length();
  }
  for( int i = 0 ; i < min_length ; i ++ ) {
    if( a.charAt(i) != b.charAt(i) ) {
      ret ++;
    }
  }
  return ret;
}
```

Übung 19.11 Rekursive Arraysumme, mittel

Schreiben Sie folgende Methode, die die Summe der Elemente eines Arrays berechnet, rekursiv neu.

```
static int sum( int[] a )
{
  int ret = 0;
  for( int i = 0 ; i < a.length ; i ++ ) {
    ret = ret + a[i];
  }
  return ret;
}
```

Übung 19.12 Rekursiver Arraytest, mittel

Schreiben Sie folgende Methode, die testet, ob alle Zahlen in einem Array positiv sind, rekursiv neu.

```
static boolean is_all_positive( int[] a )
{
  for( int i = 0 ; i < a.length ; i ++ ) {
    if( a[i] <= 0 ) {
      return false;
    }
  }
  return true;
}
```

Übung 19.13 Rekursiver Mittelwert, mittel

Schreiben Sie folgende Methode, die den Durchschnitt eines Arrays berechnet, rekursiv neu.

```
static double mean( int[] a )
{
  double ret = 0.0;
  for( int i = 0 ; i < a.length ; i ++ ) {
    ret = ret + a[i] / a.length;
  }
  return ret;
}
```

Prüfungsaufgaben zu diesem Kapitel

Übung 19.14 Zahlensolitaire, schwer, original Klausuraufgabe

Zahlensolitaire ist ein Spiel mit folgenden Regeln: Sie schreiben eine Zahl auf ein Blatt Papier. In jedem Spielzug dürfen Sie jetzt eine beliebige Ziffer aus der Zahl von dieser Zahl subtrahieren. Beispiel: Sie haben 3792 aufgeschrieben, nun subtrahieren Sie im nächsten Zug entweder $3, 7, 9$, oder 2 von der Zahl. Ihr Ziel ist es, in so wenigen Zügen wie möglich die 0 zu erreichen. Hier einige Beispiele:

Eingabe `n`	Ergebnis	Spielverlauf
5	1	$5-5=0$
10	2	$10-1=9 \Rightarrow 9-9=0$
21	4	$21-2=19 \Rightarrow 19-9=10 \Rightarrow 10-1=9 \Rightarrow 9-9=0$

Schreiben Sie eine Methode, die ausrechnet, wie viele Züge Sie dafür mindestens benötigen!

Übung 19.15 Absolutbeträge, mittel, original Klausuraufgabe, mit Lösung

Gegeben sei folgende iterative Methode, die die Summe der *Absolutbeträge* ersten `n` Zahlen des Arrays `a` berechnet:

```
static int absolute_sum( int[] a, int n ){
  int ergebnis = 0;
  int i;
  for( i = 0 ; i < n ; i = i + 1 ){
    if( a[i] < 0 ){
      ergebnis = ergebnis + a[i]*(-1);
    } else {
      ergebnis = ergebnis + a[i];
    }
  }
}
```

Beispiel: Wenn `a` ein Array mit den Einträgen $-1, 0, 1, 2$ ist, dann ist `absolute_sum( a, 3 )` gleich $(-1)\cdot(-1)+0+1=2$, und `absolute_sum( a, 4 )` ergibt 4.

Ergänzen Sie den Code der Methode `absolute_sum_r`, so dass die gleiche Berechnung rekursiv, also ohne Verwendung von **`for`**- oder **`while`**-Schleifen, durchgeführt wird.

```
static int absolute_sum_r( int[] a, int n ){
  /* Fügen Sie hier Ihren Code ein */
}
```

20-1

Kapitel 20
Schnelle Sortierverfahren und Rekursion

Ein Terabyte sortieren

20-2

Lernziele dieses Kapitels

1. Merge-Sort verstehen und implementieren können
2. Quick-Sort verstehen und implementieren können

Inhalte dieses Kapitels

20.1 Wozu schnelleres Sortieren? 167

20.2 Merge-Sort 167
20.2.1 Idee . . . 167
20.2.2 Implementation . . . 168
20.2.3 Analyse . . . 169

20.3 Quick-Sort 170
20.3.1 Idee . . . 170
20.3.2 Implementation . . . 171
20.3.3 Analyse . . . 171

20.4 *Linearzeit-Sortierung 172
20.4.1 Idee . . . 172
20.4.2 Implementation . . . 172
20.4.3 Analyse . . . 173

Übungen zu diesem Kapitel 173

Worum es heute geht

Drei Sortiertalgorithmen – Bubble-Sort, Insertion-Sort und Selection-Sort – für ein und dasselbe Problem: das Sortieren von Zahlen und Dingen. Wozu braucht die Menschheit eigentlich drei unterschiedliche Sortieralgorithmen? Sollen damit Studierende geärgert werden? Hat dies etwas mit Diversity zu tun? Vielleicht mit Risikostreuung? Konnte sich ein Normierungskomitee nicht entscheiden? Handelt es sich um die jahrzehntealte Fehde konkurrierender Lehrmeinungen? War es einfach »schon immer so«, dass man diese drei Algorithmen untersucht hat? Ist es einfach Schicksal? – Und wieso kommen heute noch zwei weitere hinzu?

Wir betrachten verschiedene Sortiertalgorithmen, weil diese *unterschiedlich schnell* sind. Wie schnell sie genau sind, hängt dabei von den zu sortierenden Zahlen ab. Manche Sortieralgorithmen sind besonders schnell, wenn die Eingabe bereits (fast) sortiert ist; andere finden zufällig Eingaben besonders schön. Deshalb gibt es nicht *den* perfekten Sortieralgorithmus, es gibt immer nur für eine bestimmte Art von Eingaben einen besonders geeigneten Algorithmus.

Die beiden Algorithmen dieses Kapitels, Merge-Sort und Quick-Sort genannt, sind *rekursive* Algorithmen. Sie lösen also das Sortierproblem, indem sie das Sortierproblem lösen – nur auf kleineren Eingaben. Die Idee, so zu sortieren, ist nicht gerade nahe liegend, kein Mensch sortiert so; tatsächlich fällt es Menschen sehr schwer, diese Algorithmen per Hand nachzuvollziehen – probieren Sie es einmal aus. Es hat einige schlaue Leute eine Menge Nachdenken gekostet, auf diese Algorithmen zu kommen. Der Aufwand hat sich aber gelohnt und Quick-Sort heißt nicht umsonst so: Schon bei recht kleinen Eingabegrößen ist dieses Algorithmus um *Größenordnungen* schneller als die Algorithmen aus den vorherigen Kapiteln.

20.1 Wozu schnelleres Sortieren?

Wiederholung: Definition des allgemeinen Sortierproblems. 20-4

Das allgemeine Sortierproblem

Eingabe Array von Objekten, die sich vergleichen lassen.

Ausgabe Eine Permutation der Objekte, so dass jedes Objekt in der neuen Reihenfolge kleiner oder gleich dem nachfolgenden ist.

Wiederholung: Wünschenswerte Eigenschaften von Sortieralgorithmen. 20-5

Folgende Eigenschaften sind bei Sortieralgorithmen besonders wünschenswert:

1. Ein Verfahren ist *stabil*, falls sich die Reihenfolge von gleichen Elementen nicht ändert.
2. Ein Verfahren ist *in-place*, falls es lediglich eine kleine Menge extra Speicher benötigt, falls es also keine Kopie des Arrays benötigt.
3. Das Verfahren sollte mit möglichst wenig Vergleichen, Vertauschungen und Verschiebungen auskommen.

Was haben wir bis jetzt erreicht? 20-6

Wir kennen drei Sortierverfahren:

1. Bubble-Sort,
2. Selection-Sort und
3. Insertion-Sort.

Sie haben alle bei zufälligen Daten eine Laufzeit von $O(n^2)$. Das theoretische Minimum liegt aber bei $O(n \log n)$.

Brauchen wir schnellere Sortieralgorithmen? 20-7

Beispiel: Sortieren mit Bubble-Sort

Es soll ein Molekül mit 10.000 Atomen visualiert werden. Dazu muss (unter anderem) 30 mal pro Sekunde die Liste der Atome sortiert werden. Bei Bubble-Sort benötigt dies grob $10.000^2/2 = 50.000.000$ Vergleiche pro Sortierung. Bräuchte ein Vergleich nur einen Taktzyklus, so wäre eine 1.5GHz CPU *nur mit Sortieren beschäftigt*.

Beispiel: Schnelles Sortieren

Betrachten wir dasselbe Problem, nur wird nun ein Algorithmus verwendet, der $O(n \log n)$ Zeit benötigt. Das macht dann grob $10.000 \log_2 10.000 \approx 133.000$ Vergleiche pro Sortierung. Bei 30 Bildern pro Sekunde muss die CPU also grob 4.000.000 Vergleiche schaffen, wofür sie nur *wenige Millisekunden braucht*.

20.2 Merge-Sort

20.2.1 Idee

Der Merge-Sort-Algorithmus 20-8

Idee

Wir gehen rekursiv vor: Erst sortieren wir die erste Hälfte, dann die zweite Hälfte. Nun müssen wir die beiden sortierten Hälften zu einem sortierten Array zusammenfügen.

Sortieren von Spielkarten nach dem Merge-Sort-Algorithmus. Regie

20.2.2 Implementation

20-9

Hauptteil von Merge-Sort

```
static void mergeSort (int[] array)
{
  if (array.length > 1)
  {
    // Kopiere erste Hälfte von array nach first
    int[] first  = new int [array.length/2];
    for (int i = 0; i<first.length; i++) {
      first[i] = array[i];
    }

    // Kopiere Rest (zweite Hälfte) von array nach second
    int[] second = new int [array.length-first.length];
    for (int i = 0; i<second.length; i++) {
      second[i] = array[i+first.length];
    }

    // Sortiere first
    mergeSort(first);

    // Sortiere second
    mergeSort(second);

    // Verschmelze (merge) der sortierten Listen
    merge(first,second,array);
  }
}
```

20-10

✎ Zur Übung

Es wird `mergeSort` aufgerufen mit dem Array `{3,9,5,0,11,2}`.
Geben Sie die genauen Inhalte der Arrays `array`, `first` und `second` zum Zeitpunkt jeder der Kommentarzeilen an.

20-11

Die `merge`-Methode.

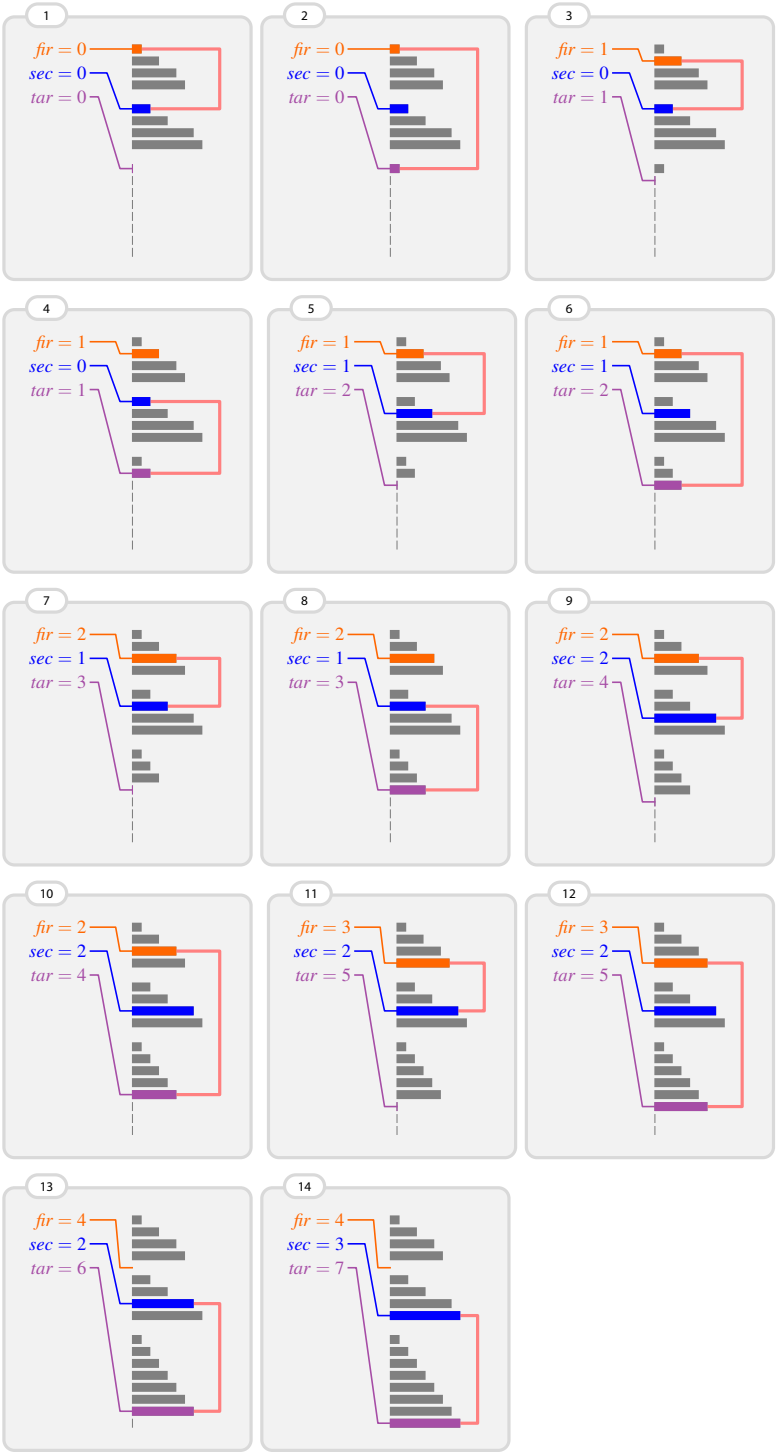
```
static void merge(int[] first, int[] second, int[] array)
{
  int first_pos  = 0;
  int second_pos = 0;
  int target_pos = 0;

  while (   first_pos  < first.length
         || second_pos < second.length)
  {
    if (first_pos < first.length &&
        (second_pos >= second.length ||
         first[first_pos] <= second[second_pos]))
    { // Wähle Element aus first
      array[target_pos] = first[first_pos];
      target_pos++;
      first_pos++;
    }
    else
    { // Wähle Element aus second
      array[target_pos] = second[second_pos];
      target_pos++;
      second_pos++;
    }
  }
}
```

Beispielablauf einer Verschmelzung

20-12

Verschmelzung von `first == {1,3,5,6}` und `second == {2,4,7,8}`.



20.2.3 Analyse

Wie schnell ist Merge-Sort?

20-13

Beitrag der ersten Rekursionsstufe

- Zunächst werden n Zahlen in $n/2$ und nochmal $n/2$ Zahlen aufgeteilt.
- Das dauert $n/2+n/2=n$ Schritte.
- Am Ende werden die Arrays verschmolzen.
- Das dauert nochmal $n/2+n/2=n$ Schritte.

Beitrag der zweiten Rekursionsstufe

- Beide Arrays der Größe $n/2$ werden in zwei Arrays der Größe $n/4$ aufgeteilt.
- Das dauert $2(n/4+n/4) = n$ Schritte.
- Am Ende werden je zwei Arrays wieder verscholzen.
- Das dauert $2(n/4+n/4) = n$ Schritte.

Beitrag der dritten Rekursionsstufe

- Die vier Arrays der Größe $n/4$ werden in je zwei Arrays der Größe $n/8$ aufgeteilt.
- Das dauert $4(n/8+n/8) = n$ Schritte.
- Am Ende werden je zwei Arrays wieder verscholzen.
- Das dauert $4(n/8+n/8) = n$ Schritte.

20-14

Die Geschwindigkeit von Merge-Sort.

- Auf jeder Rekursionsstufe werden $2n$ Schritte gemacht.
- Es gibt $\log_2 n$ Rekursionsstufen.
- Das macht $2n \log_2 n = O(n \log n)$ Schritte insgesamt.

20-15

Vor- und Nachteile von Merge-Sort

Vorteile

+ Garantierte Laufzeit von $O(n \log n)$.
+ Stabil.
+ Gut auch für Listen statt Arrays einzusetzen.

Nachteile

– Nicht in-place (braucht doppelten Speicher).

20.3 Quick-Sort

20.3.1 Idee

20-16

Der Quick-Sort-Algorithmus

Idee

Wir gehen wieder rekursiv vor. Diesmal wählen wir zunächst ein Element aus, das wir *Pivot-Element* nennen, es ist idealerweise der Median. Dann sorgen wir durch Vertauschungen dafür, dass folgendes gilt:

- Links stehen alle Elemente, die kleiner als das Pivot-Element sind.
- Rechts stehen alle Elemente, die größer als das Pivot-Element sind.

Dann sortieren wir rekursiv die linken und die rechten Elemente.

20-17

Offene Fragen zum Quick-Sort-Algorithmus

1. Was ist das Pivot-Element?
2. Wie »sorgt man dafür«, dass die Eigenschaften erfüllt sind?
3. Was sind die linke und rechte Seite?

20-18

Was ist das Pivot-Element?

Das Pivot-Element ist *idealerweise* der *Median* der Liste. Leider kennt man den Median am Anfang noch nicht. Deshalb gibt es verschiedene Strategien, das Pivot-Element zu wählen:

- Man nimmt das erste Element (sehr dumm).
- Man nimmt das mittlere Element (sehr klug).
- Man nimmt ein zufälliges Elment (nicht schlecht).
- Man führt eine so genannte Pivot-Suche durch (wer's mag).

20-19

Wie »sorgt man dafür«, dass die kleinen Element links sind?

- Man tauscht zunächst das Pivot-Element ans Ende, damit es nicht stört.
- Man hält in einer Variable `end_of_left_side` das rechte Ende der linken Seite.
- Man läuft nun über alle Elemente des Arrays.
- Immer, wenn man ein Element findet, das kleiner als das Pivotelement ist, tauscht man es an das Ende der linken Seite, die dann um dieses Element größer.

Regie

Sortieren von Spielkarten nach dem Quicksort-Algorithmus.

20.3.2 Implementation

Das Quick-Sort-Programm. 20-20

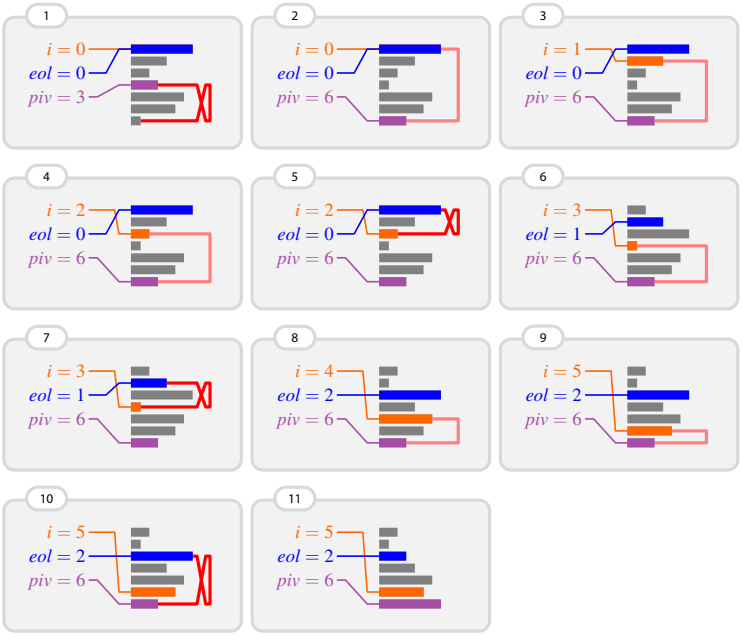
```
static void quickSort (int[] array, int start, int end)
{
  if (start < end) {
    int pivot = (start + end)/2;
    int end_of_left_side = start;

    swap (array, pivot, end);
    pivot = end;
    for (int i = start; i < end; i++) {
      if (array[i] < array[pivot])
      {
        swap(array, end_of_left_side, i);
        end_of_left_side++;
      }
    }
    swap (array, end_of_left_side, pivot);

    quickSort(array,start,end_of_left_side-1);
    quickSort(array,end_of_left_side+1,end);
  }
}
```

Beispielablauf des Quicksorts vor der Rekursion 20-21

Umsortierung von `array == {7,4,2,3,6,5,1}` mit `start == 0` und `end == 6`.



20.3.3 Analyse

Die Geschwindigkeit von Quick-Sort. 20-22

Auf jeder Rekursionsstufe werden n Schritte gemacht. Leider ist unklar, wie viele Rekursionsstufen es gibt: Dies hängt davon ab, wie nahe das Pivot-Element am Median liegt. Bei *sortierten*, *fast sortierten* und bei *umgekehrt sortierten* Daten ist das Pivot-Element der Median und bei *zufälligen* Daten ist das Pivot-Element nahe am Median. Quick-Sort hat in aller Regel eine Laufzeit von $O(n \log n)$.

20-23

Vor- und Nachteile von Quick-Sort

Vorteile

+ In der Regel um den Faktor 2 schneller als Merge-Sort.
+ In-place.
+ Liefert viele schöne Fragestellungen für Theoretiker.
+ In der Praxis oft der schnellste Algorithmus.

Nachteile

– Nicht stabil.
– Worst-case $O(n^2)$.

20.4 *Linearzeit-Sortierung

20.4.1 Idee

20-24

Kann man in linearer Zeit sortieren?

▶ **Satz**
Jeder Sortieralgorithmus, der lediglich Vergleiche zum Sortieren nutzt, benötigt mindestens $n \log_2 \frac{n}{e}$ *Vergleiche, um n Werte zu sortieren.*

Man kann also *nicht in linearer Zeit* sortieren. Sortieralgorithmen wie *Merge-Sort* sind also, bis auf einen kleinen Faktor, *optimal*. Wer *trotzdem schneller* werden will, muss einen Trick anwenden: Man vermeidet Vergleiche.

20-25

Der Distribution-Sort-Algorithmus

Idee

– Nehmen wir an, wir wissen, die Werte im Array liegen zwischen 0 und $k-1$.
– Dann zählen wir für jeden dieser möglichen Werte, wie oft er vorkommt.
– Dann durchlaufen wir alle möglichen Werte und schreiben so viele Elemente wie nötig in das Array.

20.4.2 Implementation

20-26

Das einfache Distribution-Sort-Programm

```
static void distributionSort (int[] array, int k)
{
  // Annahme: Alle Werte in array liegen zwischen 0 und k - 1.
  int[] values = new int[k];
  for (int i = 0; i<values.length; i++) {
    values[i] = 0;
  }

  for (int i = 0; i<array.length; i++) {
    values[array[i]] ++;
  }

  int pos = 0;
  for (int i = 0; i<values.length; i++) {
    for (int j = 0; j<values[i]; j++) {
      array[pos] = i;
      pos++;
    }
  }
}
```

20.4.3 Analyse

Vor- und Nachteile von Distribution-Sort

20-27

Vorteile

+ Braucht nur $O(k+n)$ Schritte.
+ Sehr einfach aufgebaut.
+ Erweiterungen können stabil gemacht werden.
+ Wenn k bekannt und klein ist, ist ein Geschwindigkeitsvorteil vom Faktor 100 gegenüber sehr guten Standardalgorithmen möglich.

Nachteile

– k muss bekannt sein (kann umgangen werden).
– Es müssen Zahlen sortiert werden (kann umgangen werden).

Zusammenfassung dieses Kapitels

▶ Merge-Sort

20-28

Algorithmus Teile den Array in zwei Hälften. Sortiere diese rekursiv. Verschmelze die sortierten Ergebnislisten.

Laufzeit $O(n \log n)$.

Vorteile Garantierte Laufzeit, stabil.

Nachteile Nicht in-place.

▶ Quick-Sort

Algorithmus Wähle ein Pivot-Element. Tausche Elemente so, dass im linken Teil alle Elemente kleiner als das Pivot-Element sind, im rechten alle größer. Sortire rekursiv beide Teile.

Laufzeit $O(n \log n)$ im Schnitt, $O(n^2)$ im schlimmsten Fall.

Vorteile In der Praxis schneller als Merge-Sort, in-place.

Nachteile Sehr schlechte Laufzeit im schlimmsten Fall.

Übungen zu diesem Kapitel

Übung 20.1 Idee des rekursiven Medians, mittel

Mit jedem Sortieralgorithmus können Sie den Median eines Arrays bestimmen, indem Sie das Array sortieren und dann den mittleren Eintrag des sortierten Arrays auslesen (wir gehen hier von einem Array ungerader Länge aus). Allerdings kann man diese Aufgabe mit einer Variante des Quicksort-Algorithmus auch ein wenig eleganter lösen. Dies funktioniert so: Statt zwei rekursiven Aufrufen wie bei Quicksort, mit denen der linken und rechten Teil des Arrays sortiert wird, beschränkt man bei der Rekursion auf den Teil des Arrays, in der der Median liegen muss. Auch in dem so »halbsortierten« Array steht dann am Ende der Median in der Mitte.

Machen Sie sich diese Vorgehensweise anhand des folgenden Arrays klar:

$$44, 75, 23, 43, 55, 12, 64, 77, 33$$

Geben Sie als Lösung mindestens die Grenzen der Array-Teile, das aktuelle Pivot-Element, und den aktuellen Stand des Arrays vor jedem rekursiven Aufruf an!

Übung 20.2 Rekursiven Median implementieren, schwer

Implementieren Sie den Algorithmus aus der vorherigen Aufgabe, indem Sie den Algorithmus aus der Vorlesung entsprechend abändern. Testen Sie anhand von drei Beispielen der Längen 9, 19 und 29, ob so der Median gefunden werden kann.

Übung 20.3 Laufzeit des rekursiven Medians, schwer

Wie ist im durchschnittlichen Fall ungefähr die Laufzeit dieses Verfahrens zur Mediansuche?

Prüfungsaufgaben zu diesem Kapitel

Übung 20.4 Quicksort verstehen, mittel, original Klausuraufgabe, mit Lösung

Gegeben sei ein Array aus Integer-Werten:

```
int[] a = {36, 83, 96, 42, 1, 17, 28 }
```

Das Array soll mit Quicksort sortiert werden. Dazu wird die Funktion `quicksort` aus der Vorlesung wie folgt aufgerufen:

```
quicksort( a, 0, 6 )
```

1. Welches Pivot-Element wird gewählt?
2. Wie sieht das Array `a` vor den ersten rekursiven Aufrufen von `quicksort` aus?
3. Mit welchen Parametern (Grenzen der zu sortierenden Teilarrays) wird `quicksort` daraufhin rekursiv aufgerufen?

Übung 20.5 Quicksort verstehen, mittel, typische Klausuraufgabe

Gegeben sei ein Array mit folgenden Einträgen:

47, 14, 46, 84, 52, 88, 60, 30, 87

Zeichnen Sie einen Aufrufbaum der rekursiven Aufrufe, die beim Sortieren dieses Arrays mit der Funktion `quicksort` entstehen. Geben Sie für jeden Aufruf das Pivot-Element und die linken und rechten Grenzen des zu sortierenden Teil-Arrays an.

Übung 20.6 Quicksort, leicht, mit Lösung

Folgende Zahlen sollen mit dem Quicksort-Algorithmus aus der Vorlesung sortiert werden:

−3	2	9	8	1	7	0

Zunächst wählt der Algorithmus ein Pivot-Element. Nun wird das Pivot-Element ans Ende des Arrays getauscht. Eine **`for`**-Schleife unterteilt das Array in eine »linke Seite« und eine »rechte Seite«. Am Ende wird das Pivot-Element zwischen den beiden Seiten eingefügt.

1. Wenn als Pivot-Element das mittlere Element gewählt wird, welchen Inhalt hat dann das Array bevor die rekursiven Aufrufe ausgeführt werden?
2. Auf welchen Teilarrays werden die rekursiven Aufrufe durchgefürt?
3. Warum sollte man das mittlere Element als Pivot-Element wählen und nicht das erste?

Teil V
Modellierung

Goethe sagte einmal: »Mathematiker sind wie Franzosen: Wenn man ihnen etwas erzählt, übersetzen sie es in ihre eigene Sprache, und es bedeutet etwas ganz anderes.« Lebte Goethe heute, so würde er wahrscheinlich auch noch Compiler neben Mathematiker und Franzosen stellen: Beim Programmierung tut man »der Wirklichkeit« oft ein wenig Gewalt antun, um sie in Arrays und Strings zu pressen. (Eventuell hätte sich Goethe aber aus Rücksichtnahme auf den Zeitgeist Vergleiche mit bestimmten Geschlechtern und Nationalitäten verkniffen.)

Die Sicht auf die Welt »durch die Brille des Übersetzers« kann zu einem echten Problem werden, wenn man an größeren Projekten arbeitet, vielleicht auch noch im Team. Am Ende versteht der Übersetzer, worum es geht, aber keiner der beteiligten Menschen. Wer schon einmal in den Programmtext eines Kommilitonen geschaut hat, weiß, was hier gemeint ist.

Mit den Jahren hat sich die Erkenntnis in der Informatik durchgesetzt, dass sich nicht die Wirklichkeit den Programmen anzupassen hat, sondern umgekehrt: Programmtexte sollten die Wirklichkeit in möglichst natürlicher Weise widerspiegeln. Statt über »zwei Arrays von Strings, wobei der erste Array die Vornamen und der zweite die Nachnamen der Studierenden speichert« zu reden, will man *schon im Programmtext* über »eine Liste von Studierenden« reden.

Sogenannte *objektorientierte Programmiersprachen* (Java ist eine solche) beherrschen solche Abbilder der Wirklichkeit im Rechner – man spricht von *Modellen* – besonders gut. In diesem Teil soll es nun darum gehen, wie man solche Modelle erstellt und aufschreibt.

21-1

Kapitel 21
Modularisierung im Großen

Wie man große Probleme in genießbare Häppchen aufteilt

21-2

Lernziele dieses Kapitels

1. Das Konzept des Top-Down-Entwurf verstehen und anwenden können
2. Das Konzept der Klassen- und Modulbildung verstehen

Inhalte dieses Kapitels

21.1	Modularisierung von Software	177
21.1.1	Probleme im Großen	177
21.1.2	Motivierende Problemstellung	177
21.1.3	Modularisierungsebenen	178
21.1.4	Softwareentwurf	178
21.2	Klassen	179
21.2.1	Der Begriff der Klasse	179
21.2.2	Java-Syntax von Klassen	179
21.2.3	Benutzung von Klassen	180
21.2.4	Zugriffsrechte	180
21.3	Pakete	181
21.3.1	Der Begriff des Pakets	181
21.3.2	Java-Syntax für Pakete	181
21.3.3	Benutzung von Paketen	182

Worum es heute geht

Die mittelalterliche Gesellschaft war eine stark *ständisch* geprägt. Es gab die Bürger, den Klerus und den Adel. Zur Zeit eines Karl Marx wurden die Stände theoretisch wie praktisch durch *Klassen* abgelöst, Menschen gehörten nun der proletarischen oder aber der kapitalistischen Klasse an. Heute ist in der Soziologie der Begriff der *Schicht* state-of-the-art, es gibt Unterschichten, Mittelschichten, Oberschichten, akademische Schichten und viele mehr. All diesen Einteilungen von Menschen liegt eine Idee zugrunde: Übersichtlichkeit. Wie viel einfacher wird die Welt, wenn man sich nicht mit dem Individuum auseinandersetzen muss, sondern mit einem Aggregat? Wie viel leichter ist es zu sagen, »Der Kapitalist ist böse« als »Max ist böse« oder »Der Adel ist faul« als »Frau Fleißighuber ist faul«. Aus soziologischer Sicht ist jede Zusammenfassung von Menschen zu einer »Klasse« immer mit viel Vorsicht zu genießen, man prüfe genau, wer hier warum einen Klassenbegriff einführt.

In der Informatik dienen »Klassen« letztendlich demselben Zweck wie in der Soziologie: Übersichtlichkeit. Statt Menschen werden wir später *Objekte* zu Klassen zusammenfassen; in diesem Kapitel soll es aber darum noch nicht gehen, sondern lediglich um einen anderen ordnenden Aspekt von Klassen: Mit ihrer Hilfe verschafft man sich Ordnung, ordnet Methoden in zusammengehörige Gruppen, schützt verschiedene Programmteile voreinander.

Glücklicherweise sind derzeit keinerlei ethisch-moralischen Einwände gegen die Verwendung von Klassen innerhalb der Informatik bekannt. Menschen sind eben keine Objekte, das können Sie im Grundgesetz nachlesen. Sie dürfen also in Ihren Programmen ruhigen Gewissens mit Klassen Ordnung stiften.

21.1 Modularisierung von Software

21.1.1 Probleme im Großen

Wie baut man ein Haus?

21-4

Soll ein Haus gebaut werden, so hängt es von der Größe ab, wie viele Personen beteiligt sind:

Hundehütte Zimmerer

Schuppen Mutter und Tochter

Einfamilienhaus Architekt, Maurerin, Elektriker, Zimmerer, Klempnerin, Fahrerin, Gehilfen, usw.

Hochhaus Architekturbüro, Bauaufsicht, Bauingenieure, Statikerin, Maurer, Elektrikerin, Zimmerer, Klempnerin, Fahrer, viele Gehilfen, usw.

Wie baut man Software?

21-5

Bei Software hängt es ebenfalls von der Größe ab, wie viele Personen beteiligt sind:

Skalarprodukt MLS-Student

Vier-Gewinnt Programmiererin

Mail-Programm Software-Architekt, Oberflächen-Designerin, Programmierer, Gehilfen, Pizza-Bote, usw.

Betriebssystem (Software)-Architekten, Projektmanagerin, Oberflächen-Designer, Kernel-Programmiererin, Treiber-Programmierer, Oberflächen-Programmiererin, Gehilfen, viele Pizza-Boten, usw.

Probleme, die bei großer Software entstehen.

21-6

Arbeiten an einem Softwareprojekt viele Beteiligte, so müssen sich diese *koordinieren*. Damit Beteiligte unabhängig arbeiten können, muss das Projekt in verschiedene *Module* aufgespalten werden. Programmiersprachen sollten diese Modularisierung *unterstützen*.

21.1.2 Motivierende Problemstellung

Ziel: Simulation des Immunsystems.

21-7

Eine komplexe Problemstellung

Mittels eines Computerprogramms soll *das Immunsystem simuliert werden*.

Literatur

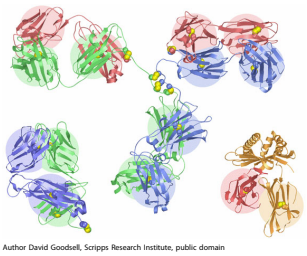
[1] P. E. Seiden, F. Celada. A model for simulating cognate recognition and response in the immune system. *Journal of Theoretical Biology*, 158(3):329-57, 1992.

> We have constructed a model of the immune system that focuses on the clonotypic cell types and their interactions with other cells, and with antigens and antibodies. [...] We propose using computer simulation as a tool for doing experiments in machine, in the computer, as an adjunct to the usual in vivo and in vitro techniques. [...] a model simulating areas of interest could be used for extensively testing ideas to help in the design of the critical biological experiments. [...]

21-8

Einige interessante Teile des Immunsystems

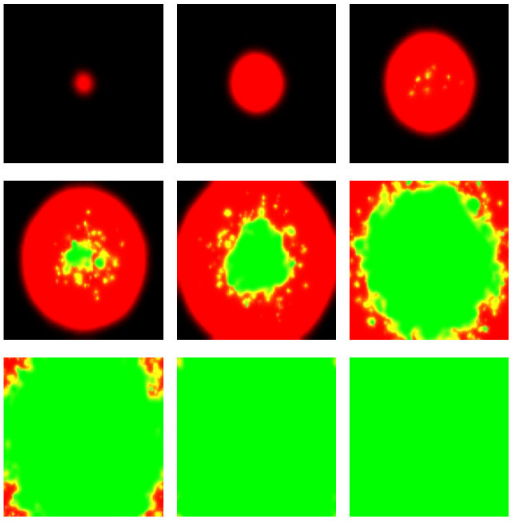
T-Zellen Rezeptor, Antikörper, Major Histocompatibility Complex



Author David Goodsell, Scripps Research Institute, public domain

21-9

Ergebnis einer möglichen Simulation.



Copyright by Johannes Textor

rot = Antigene, grün = Antikörper

21.1.3 Modularisierungsebenen

21-10

Wiederholung: Methoden erlauben die Strukturierung von Programmen.

- Programme werden in kleiner Unterprogramme, in Java *Methoden* genannt, aufgeteilt.
- Methoden sind in der Regel zwischen einer und etwa 30 Zeilen lang.
- Längere Methoden sollten in kleine Methoden aufgespalten werden.

21-11

Die Modularisierung mittels Methoden löst nicht alle Probleme.

Wenn ein Programm aus 10.000.000 Zeilen Code besteht, dann braucht man grob *eine Million Methoden*. Offenbar muss hier *weiter modularisiert* werden. Dazu werden Methoden wiederum *hierarchisch* zusammengefasst.

21-12

Die Modularisierungshierarchie.

Instruktionsfolgen werden zu immer größeren Einheiten zusammengefasst:

1. *Blöcke* (Scopes) sammeln Anweisungen.
2. *Methoden* sammeln Blöcke.
3. *Klassen* sammeln Methoden.
4. *Pakete* sammeln Klassen.
5. *Pakete* sammeln Pakete.

21.1.4 Softwareentwurf

21-13

Was ist Softwaretechnik?

Bei einem umfangreichen Simulationsprogramm sollte man *nicht* »drauflosprogrammieren«. Vielmehr muss das Programm *geplant* werden. Einer der ersten Schritte ist die *Modularisierung*. Das Teilgebiet der Informatik, das sich mit der Planung und Programmierung großer Programme beschäftigt, heißt *Softwaretechnik*.

Die zwei Richtungen beim Softwareentwurf. 21-14

- Das *Bottom-Up-Prinzip*:
  - Bei der Planung stellt man fest, dass eine kleine Teilaufgabe oder -aspekt wiederholt vorkommt.
    Beispiel: Es müssen immer wieder Zellen bewegt werden. Beispiel: Es müssen immer wieder Reaktionen berechnet werden.
  - Teilaufgaben werden dann in *Methoden ausgelagert* oder in *Klassen gekapselt*.

  Mehr dazu im nächsten Kapitel.
- Das *Top-Down-Prinzip*:
  - Das Problem wird in große Teilbereiche aufgeteilt.
    Beispiel: Teilbereiche sind immunologische Modelle, Simulationsalgorithmen, graphische Darstellung, Datenkonvertierung usw.
  - Jeder Teilbereich wird seinerseits in kleinere Teilprobleme aufgeteilt; und so fort.
    Beispiel: Teilprobleme sind die Simulation einer T-Zelle, Simulation einer B-Zelle usw.

  Mehr dazu im Folgenden.

21.2 Klassen

21.2.1 Der Begriff der Klasse

Was sind Klassen? 21-15

Klassen sind ein Konzept in der objektorientierten Programmierung. Sie dienen zur *Modellierung*, da man mit ihnen neue *Datenstrukturen* erzeugt. Gleichzeitig dienen sie auch zur *Modularisierung*:

1. Man identifiziert Teilbereiche des Problems und führt hierzu Klassen ein.
2. Man implementiert Methoden, die für die Klassen nützlich sind.

Beispiel: Top-Down Modularisierung mit Klassen

Wir benötigen Methoden zum Lesen und Schreiben von Dateien. Weiter benötigen wir Methoden, die Reaktionen modellieren. Deshalb führen wir zwei Klassen `FileManagement` und `Reactions` ein und überlegen, welche konkreten Methoden nützlich wären.

21.2.2 Java-Syntax von Klassen

Wie schreibt man die Klassen nun auf? 21-16

```
// In der Datei FileManagement.java
class FileManagement
{
  static void writeToFile(String s, String filename)
  {
    /* Code einer Methode, die den String s in die
    Datei filename schreibt. */
  }
  static String readFromFile(String filename)
  {
    /* Code einer Methode, die den Inhalt der Datei
    filename liest zurückgibt. */
  }
}
```

```
// In der Datei Reactions.java
class Reactions
{
  static double reactionLikelihood(String a, String b)
  {
    /* Diese Methode liefert die Wahrscheinlichkeit
    zurück, dass die durch die Strings a und b
    repräsentierten Oberflächen interagieren */
  }
}
```

21-17

Die Syntax von Klassen allgemein.

Eine Klasse wird mittels `class` eingeleitet, gefolgt von ihrem Namen. *Der Name* ist wie immer ein Java-Bezeichner. Im Inneren der Klasse finden sich die *Methoden* der Klasse. Eine Klasse darf auch Sub-Klassen haben; das brauchen wir aber nicht.

21.2.3 Benutzung von Klassen

21-18

Wie benutzt man die Klassen nun?

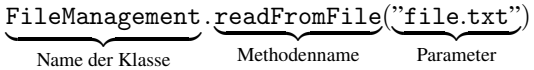
Jede Klasse ist ein eigener Scope, weshalb Methoden gleichen Namens in unterschiedlichen Klassen sich nicht stören.

Problem

Aber wie kann man dann Methoden benutzen, wenn sie doch außerhalb des Scopes nicht sichtbar sind?

Lösung

Es gibt eine spezielle Notation, mit der man *Scopes wieder öffnen kann* und dadurch Methoden *in anderen Scopes aufrufen* kann:



21-19

Eine Beispielbenutzung der Methoden aus `FileManagement`

```
// In der Datei FileManagement.java
class FileManagement {
  static void writeToFile(String s, String filename) {...}
  static String readFromFile(String filename)        {...}
}
```

```
// In der Datei Copy.java
class Copy
{
  public static void main (String[] args)
  {
    String file_contents =
      FileManagement.readFromFile("original.txt");
    FileManagement.writeToFile
      (file_contents, "copy␣of␣original.txt");
  }
}
```

21-20

✎ Zur Übung

Die Klasse `Math` aus der System-Bibliothek stellt eine Funktion `exp` zur Verfügung zur Berechnung von e^x. Geben Sie an, wie die Deklaration von `exp` innerhalb von `Math` aussieht.

21.2.4 Zugriffsrechte

21-21

Das Prinzip der Kapselung.

- Methoden können *privat* oder *öffentlich* sein.
- Öffentliche Methoden kann man von außen mittels `SomeClass.method()` aufrufen.
  Um anzuzeigen, dass eine Methode öffentlich ist, stellt man ihr das Attribut `public` vor.
- Private Methoden kann man von außen nicht aufrufen.
  Um anzuzeigen, dass eine Methode privat ist, stellt man ihr das Attribut `private` vor.

✎ Zur Diskussion

Welche Vorteile hat man durch die Unterscheidung in öffentliche und private Methoden?

✎ Zur Übung 21-22

Nun soll eine Methode `vectorLength` in einer neuen Klasse `MathInThePlane` implementiert werden.
Es soll die Wurzelmethode `sqrt` aus der Klasse `Math` benutzt werden.
Wie lautet der Code (Parameter: Koordinaten eines Vektors)?

21.3 Pakete

21.3.1 Der Begriff des Pakets

Die breiteste Hierarchieebene in Java: Pakete 21-23

Java erlaubt es, mehrere Klassen zu einem *Paket* (`package`) zusammenzufassen. Mehrere Pakete können wiederum zu einem Paket zusammengefasst werden und so fort. Pakete können nur Klassen und Pakete enthalten, keine Methoden.

21.3.2 Java-Syntax für Pakete

Java-Syntax für Pakete 21-24

Am Anfang einer Java-Programm-Datei kann man folgendes schreiben (man beachten: keine geschweiften Klammern):

```
package name_of_package;
```

Dies hat folgende Effekte:

- Alles, was nun folgt, wird dem Paket `name_of_package` hinzugefügt.
- Eine zweite Datei könnte ebenfalls so anfangen. Ihr Inhalt würde ebenfalls dem Paket hinzugeschlagen.
- Im *Gegensatz dazu* kann man einer *Klasse nicht* in einer zweiten Datei noch Methoden hinzufügen.

Das Paket `immuneSystemSimulation`. 21-25

```
// In der Datei immuneSystemSimulation/FileManagement.java
package immuneSystemSimulation;

class FileManagement {
  static void writeToFile(String s, String filename) {...}
  static String readFromFile(String filename)        {...}
}
```

```
// In der Datei immuneSystemSimulation/Reactions.java
package immuneSystemSimulation;

class Reactions {
  public static double reactionLikelihood
    (String a, String b)
  {...}
}
```

21.3.3 Benutzung von Paketen

21-26

Wie benutzt man die Pakete nun?

Genau wie eine Klasse ist ein Paket ein eigener Scope. Deshalb hat man das gleiche Problem wie bei Klassen mit dem Zugriff, was aber auch genauso gelöst wird:

- Möchte man auf `xyz` im Paket `somePackage` zugreifen, so schreibt man einfach *`somePackage.xyz`*.
- Jede Klasse kann auch noch `public` oder `private` sein.

```
class Copy
{
  public static void main (String[] args)
  {
    String file_contents =
      immuneSystemSimulation.FileManagement.readFromFile
        ("original.txt");
    immuneSystemSimulation.FileManagement.writeToFile
      (file_contents, "copy␣of␣original.txt");
  }
}
```

21-27

Vereinfachung des Lebens mittels der Import-Anweisung.

Oft benutzte Klassen können »importiert« werden.

```
import immuneSystemSimulation.*;

class Copy
{
  public static void main (String[] args)
  {
    String file_contents =
      FileManagement.readFromFile("original.txt");
    FileManagement.writeToFile
      (file_contents, "copy␣of␣original.txt");
  }
}
```

Zusammenfassung dieses Kapitels

21-28

▶ **Hierarchieebenen in Java**

1. *Blöcke* (Scopes) sammeln Anweisungen.
2. *Methoden* sammeln Blöcke.
3. *Klassen* sammeln Methoden.
4. *Pakete* sammeln Klassen.
5. *Pakete* sammeln Pakete.

▶ **Das »Öffnen benannter Scopes«**

Manche Scopes (Klassen, Pakete) haben einen Namen und lassen sich *von außen öffnen.* Die Syntax ist dabei immer `scope_name.ding_im_scope`. Dies ist allerdings nur möglich, wenn das »Ding im Scope« als **`public`** deklariert wurde.

▶ **Syntax von Klassen**

```
// Datei muss heißen wie die Klasse, also Example.java
// Klassennamen fangen mit Großbuchstaben an

class Example
{
  ... // Die Methoden
}
```

- Syntax von Paketen

```
// Verzeichnis muss heißen wie das Paket, also myPackage
// und darin dann myPackage/Example.java
// Paketnamen fangen mit Kleinbuchstaben an

package myPackage;

class Example
{
  ... // Die Methoden
}
```

22-1

Kapitel 22
Objekte – Attribute und Lebenszyklus

Ihre Klassen, ihre Attribute, ihr Lebenszyklus

22-2

Lernziele dieses Kapitels

1. Modellierung von Daten mittels Klassen verstehen
2. Konzept des Objekts und des Attributs kennen und anwenden können
3. Lebenszyklus von Objekten verstehen

Inhalte dieses Kapitels

22.1 Klassen von Objekten 185
22.1.1 Was sind Objekte? 185
22.1.2 Begriff der Klasse 185
22.1.3 Syntax von Klassen 186

22.2 Lebenszyklus von Objekten 186
22.2.1 Erzeugung (Geburt) 186
22.2.2 Zugriff 187
22.2.3 Veränderung 188
22.2.4 Vernichtung (Tod) 188

22.3 Objekthierarchien 189
22.3.1 Objekte als Attribute 189
22.3.2 Beispiel: Der Zellkern 189
22.3.3 Beispiel: Knöpfe 190

Übungen zu diesem Kapitel 191

Worum es heute geht

Wir beschäftigen uns nun schon viele Kapitel mit der Programmiersprache Java, die als *objektorientierte* Sprache angepriesen wurde. Von den *Objekten* war bis jetzt aber noch nie die Rede, was Sie sicherlich schon etwas misstrauisch gestimmt hat. Der zentrale Begriff der ganzen objektorientierten Programmierung kommt in dieser Vorlesung so spät vor aufgrund der Geschichte von Java: Java ist zwar eine objektorientierte Sprache, sie ist aber aus einer imperativen Sprache (nämlich C) entstanden. Bevor man überhaupt sinnvoll über Objekte reden kann, muss man zunächst die imperativen Grundkonstrukte verstanden haben (wie Variablen, Arrays, Zuweisungen, Schleifen oder auch Scopes). In so genannten *reinen* objektorientierten Sprachen wie Eiffel ist dies anders, hier ist der Objektbegriff einer der ersten, die man kennen lernt.

Im allgemeinen Sprachgebrauch ist das Wort »Objekt« ja recht allgemein. Genaugenommen ist »Objekt« die vornehme Form von »Ding«, also von etwas völlig unbestimmtem. Und tatsächlich kann schlichtweg *alles* ein Objekt im Sinne der objektorientierten Programmierung sein. Dateien können Objekte sein, ebenso wie Filme, Strings, Musikstücke, Arrays, Programme, Stühle, Tische, sowie das Leben, das Universum und der ganze Rest. Was Objekte eint, ist der Umstand, dass sie (a) Attribute haben und (b) einen Lebenszyklus durchlaufen: sie werden geboren, leben und sterben.

Die *Attribute* eines Objektes sind die veränderlichen oder auch unveränderlichen Eigenschaften eines Objekts. Lassen Sie uns als Beispiel ein Ikea-Regal aus objektorientierter Sicht betrachten. Zunächst fallen einem sicherlich die Standardattribute Höhe, Länge, Breite, Gewicht und Farbe ein. (Allerdings ist schon die Farbe etwas kniffelig zu modellieren, nehmen wir vereinfacht einen String.) Ein originelles Attribute ist sicherlich der Name, hieran arbeiten sehr kreative Leute oft wochenlang. Weiterhin ist auch das Attribut `String nachbildung_von` wichtig, welches speichert, welchem Naturprodukt die Plastikoberfläche

angeblich ähneln soll. Profis wissen natürlich, dass auch noch die Attribute Regalnummer und Fach sehr wichtig sind.

Ziel dieses und des folgenden Kapitels ist es, dass Sie am Ende die Welt durch eine objektorientierte Brille sehen und überall Objekte entdecken.

22.1 Klassen von Objekten

22.1.1 Was sind Objekte?

Alles ist ein Objekt. 22-4

In der Wirklichkeit begegnen uns ständig *Dinge* wie Tische, Stühle, T-Zellen, Antigene und so weiter. Die Idee der *objektorientierten Programmierung* ist, solche Dinge in der Programmiersprache *möglichst direkt zu repräsentieren*. Dazu benutzt man dann *Objekte*.

Beispiel

Wollen wir zwanzig Antigene und drei Antikörper simulieren, so würde das in einem OO-Programm mittels zwanzig Antigen-Objekten und drei Antikörper-Objekten geschehen.

Die zentralen Eigenschaften von Objekten. 22-5

1. Objekte haben eine *Identität*. Das bedeutet, dass es mehrere Objekte geben kann mit denselben Eigenschaften.
   Beispiel: Es viele identisch aussehende Norrebo-Regale bei Ikea. Trotzdem ist Ihr Norrebo-Regal-Objekt ein anderes als mein Norrebo-Regal-Objekt. Ihr und mein Norrebo-Regal-Objekt haben eine *unterschiedliche Identität*.
2. Objekte haben *Attribute*. Dies sind Eigenschaften wie *Höhe*, *Breite*, *Oberflächenstruktur* oder was auch immer. Man kann sie *erfragen* und auch *verändern*.
3. Objekte haben *Fähigkeiten*:
   - Man kann Objekten *Nachrichten schicken*.
   - Dazu werden *Objekt-Methoden* benutzt.

   Dies besprechen wir im nächsten Kapitel.

22.1.2 Begriff der Klasse

Klassen von Objekten. 22-6

Wir benutzen Klassen bereits zur *Modularisierung*. Ihre *eigentliche Aufgabe* ist aber, *Objekte zu Klassen zusammenzufassen. Dies ist eine ganz andere Aufgabe!*

Idee der Klassenbildung

Betrachten wir ein Modell des Warenlagers von Ikea. Jedes kaufbare Ikea-Ding ist ein Objekt im Modell. Man bildet nun *Klassen von gleichartigen Objekten*. Beispiel: Alle Tische bilden eine `Tisch`-Klasse, alle Stühle bilden eine `Stuhl`-Klasse und so weiter. Jedes Objekt gehört genau einer Klasse an und *die Klasse legt die Attribute des Objektes fest*.

✎ **Zur Diskussion**

Wir wollen eine *Antigen-Klasse* bauen. Die Objekte dieser Klasse sollen also Antigene modellieren.
Welche Attribute erscheinen Ihnen sinnvoll/nötig?

22.1.3 Syntax von Klassen

22-7

Eine einfache Studenten-Klasse.

Nehmen wir an, Ihre Tutorin möchte eine Liste der Studenten in ihrem Tutorium anlegen. Dies ist ein Array von »Studentenobjekten«. Sinnvolle Attribute sind Vor- und Nachname und Matrikelnummer. In Java sieht das so aus:

```
class Student
{
  String nachname;       // Erstes Attribut
  String vorname;        // Zweites Attribut
  int    matrikelnummer; // Drittes Attribut

  // Attribute werden wie Variablen deklariert,
  // sind aber keine echten Variablen.

  // Life is hard.
}
```

22-8

Klassen sind Datentypen.

Wiederholung: Was ist ein Datentyp?

Ein *Datentyp* war eine Menge von möglichen Werten. Hat ein Wert den Datentyp `int`, so weiß man, dass dieser Wert eine ganze Zahl ist. Es gibt nur sehr wenige *eingebaute* Datentypen.
Beispiel: `int`

Erstellung neuer Datentypen

- Jede Klasse definiert einen *neuen* Datentyp.
  Beispiel: `Student`
- Hat ein Wert den Datentyp `Student`, so weiß man, dass dieser Wert ein Studenten-Objekt ist.
- Man nennt die Objekte auch *Instanzen* ihrer Klasse.

22-9

✎ Zur Übung

Schreiben Sie den Code einer Klasse `Date` auf.

22.2 Lebenszyklus von Objekten

22-10

Der Lebenszyklus eines Objektes

1. Objekte werden irgendwann *erzeugt*.
2. Objekte werden danach *benutzt*.
3. Objekte werden *automatisch vernichtet*, wenn sie nicht mehr benutzt werden.

22.2.1 Erzeugung (Geburt)

22-11

Die Erzeugung von Objekten mittels `new`.

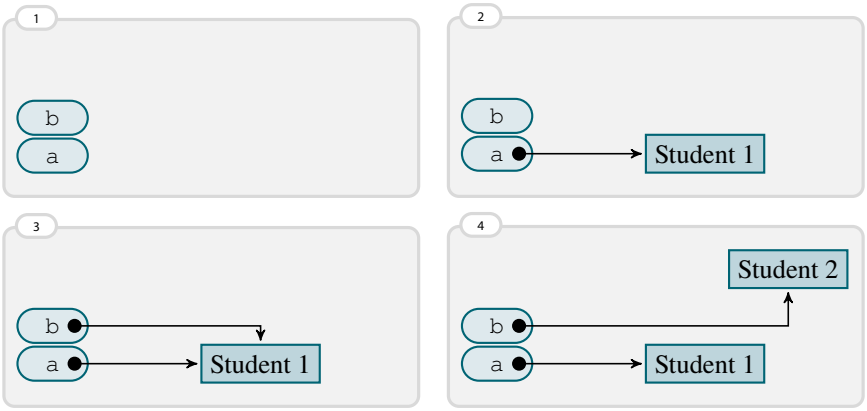
Irgendwo im Programm:

```
Student a, b;
// 1. Deklaration von zwei Variablen (noch keine Objekte!)

a = new Student ();
// 2. Jetzt erzeugen wir ein Objekt

b = a;
// 3. Jetzt verweisen sowohl a als auch b auf das Objekt

b = new Student ();
// 4. Jetzt erzeugen wir ein zweites Objekt
```



Die großen Unterschiede zwischen Variablen und Objekte

22-12

Objekte

Ein *Objekt* wird einmal mittels **`new`** erzeugt und bleibt dann an einer festen Stelle im Speicher. Lediglich seine Attribute ändern sich. Objekte können *sehr lange leben*. Insbesondere *leben sie auch weiter*, wenn die Methode, in der sie erzeugt wurden, beendet ist.

Variablen

Variablen enthalten lediglich *Verweise* auf Objekte, dabei können *viele Variablen* auf *dasselbe Objekt* verweisen. Variablen können auch *zuerst auf das eine, später auf ein anderes Objekt verweisen*. Schließlich können Variablen auch *auf gar kein Objekt* verweisen. (Dann ist diese Variable gleich **`null`**.) Im Gegensatz zu Objekten sind Variablen lokal zu ihrem Scope.

22.2.2 Zugriff

Zugriff auf die Attribute eines Objektes.

22-13

Jedes Objekt speichert für jedes seiner Attribute einen Wert. Um diese Attribute zu lesen oder zu schreiben, muss man das Objekt »aufmachen«, genau wie bei Scopes:

`fachschaftsvertreter`.`nachname`

`fachschaftsvertreter`: Variable, die auf ein Objekt verweist
`nachname`: Attribut

Beispiel

```
// Deklaration einer Variable und Erzeugung eines Objekts
Student a = new Student ();

// Setzen des Namens
a.nachname = "Musterfrau";
a.vorname  = "Petra";

// Setzen der Matrikelnummer
a.matrikelnummer = 1234567;

// Ausgeben der Matrikelnummer
System.out.println ("Die Matrikelnummer ist " + a.matrikelnummer);
```

22.2.3 Veränderung

22-14

Beispiel einer Objektveränderung.

```
// Datei Student.java
class Student
{
  String nachname, vorname;
  int    matrikelnummer;
}

// Datei Beispiel.java
class Beispiel {
  static void exmatrikuliere (Student s)
  {
    s.matrikelnummer = -1;
  }
}
```

22-15

✎ Zur Übung

Geben Sie den Code einer Methode `konflikt` an, die überprüft, ob zwei Studenten(objekte) die gleiche Matrikelnummer haben, aber unterschiedliche Namen.

```
static boolean konflikt (Student a, Student b)
```

22.2.4 Vernichtung (Tod)

22-16

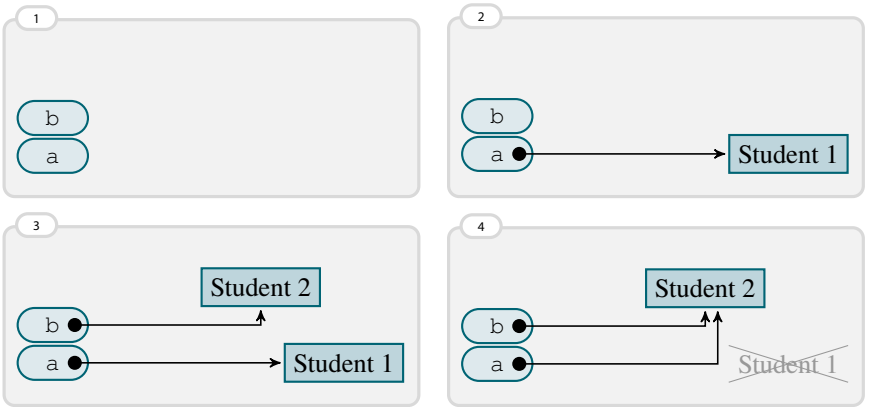
Objekte werden automatisch gelöscht.

```
Student a, b;
// 1. Deklaration von zwei Variablen (noch keine Objekte!)

a = new Student ();
// 2. Jetzt erzeugen wir ein Objekt

b = new Student ();
// 3. Jetzt erzeugen wir ein zweites Objekt

a = b;
// 4. Jetzt kann das erste gelöscht werden
```



22.3 Objekthierarchien

22.3.1 Objekte als Attribute

Objekte können Objekte als Attribute haben. 22-17

Manche Objekte »bestehen« aus anderen Objekten. (Man sagt auch, sie sind *aggregiert* aus anderen Objekten.) Zum Beispiel besteht eine Zelle aus verschiedenen Objekten: Dem Zellkern, der Membran, den Mitochondrien, etc. Dieser Umstand lässt sich auch im Programm abbilden:

- Es gibt `Cell`-Objekte, `Nucleus`-Objekte, `Membrane`-Objekte und `Mito`-Objekte.
- Ein `Cell`-Objekt muss sich nun merken, aus *welchen* der anderen Objekte es »besteht«.
- Dazu erhält das `Cell`-Objekte *Attribute*, die auf die anderen Objekte verweisen.

22.3.2 Beispiel: Der Zellkern

Code der Kern-, Membran- und Mitochondrienklassen. 22-18

```
class Nucleus {
  String dna;
  double weight;
  // ...
}

class Membrane {
  String kind;
  // ...
}

class Mito {
  String dna;
  double weight;
  double energy_level;
  // ...
}
```

Erweiterter Code der Zellklasse. 22-19

```
// Datei Cell.java
class Cell {
  int x, y;
  String kind;

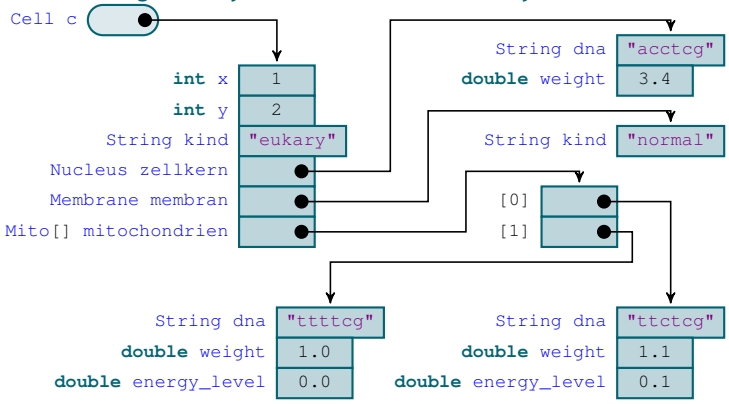
  // Verweis auf das Nucleus Objekt:
  Nucleus zellkern;

  // Verweis auf die Membran:
  Membrane membran;

  // Array von Verweisen auf die Mitochondrien
  Mito[] mitochondrien;
}
```

22-20

Visualisierung der Objekthierarchie eines Zellobjekts.



22-21

Code zur Erzeugung der Objekthierarchie.

```
Cell c     = new Cell ();     // Zellobjekt erzeugt, aber noch leer
c.x        = 1;
c.y        = 2;
c.kind     = "eukary";

Nucleus n = new Nucleus (); // Jetzt das Zellkernobjekt
n.dna     = "acctcg";
n.weight  = 3.4;

c.zellkern = n; // Jetzt verweist c.zellkern auf das n Objekt

c.membran = new Membran (); // Erzeugung und Zuweisung in einem
c.membran.kind = "normal";  // Die ''kind'' der Membran von c

c.mitochondrien = new Mito[2]; // Erzeuge das Array-Objekt

c.mitochondrien[0] = new Mito (); // Erzeuge erstes Mito-Objekt
c.mitochondrien[0].dna = "ttttcg";
c.mitochondrien[0].weight = ...;

c.mitochondrien[1] = new Mito (); // Erzeuge zweites Mito-Objekt
c.mitochondrien[1].dna = "ttctcg";
c.mitochondrien[1].weight = ...;
```

22.3.3 Beispiel: Knöpfe

22-22

Ein Knopf hält Verweise auf Farbobjekte.

```
class Color
{
  float red_part;   // Rot-Anteil; Zahl zwischen 0 und 1
  float green_part; // Grün-Anteil
  float blue_part;  // Blau-Anteil
}

class Button
{
  int height;
  int width;

  Color background; // Verweis auf ein Objekt.
  Color text_color; // Noch ein Verweis auf ein Objekt
                    // (vielleicht sogar auf dasselbe,
                    // aber besser nicht...)

  String text;      // Auch Strings sind in Wirklichkeit
```

```
                    // Objekte
}
```

✎ Zur Übung 22-23

Geben Sie Code an, um `Button`- und `Color`-Objekte zu erzeugen, so dass das `Button`-Objekt korrekte Verweise die Vorder- und Hintergrundfarbe hat.

Zusammenfassung dieses Kapitels

▶ Klassen sind Datentypen 22-24

Klasse sind *Datentypen* (so wie **`int`**), deren »Werte« (so wie `5` bei **`int`**) man *Objekte* nennt.

▶ Klassen – und damit auch Objekte – haben *Attribute*

In der Klasse deklariert man Attribute genau so, wie man Variablen deklariert.

```
class Example {
  int    attribute_1;
  String attribute_2;
}
```

▶ Der Lebenszyklus eines Objekts

```
// Geburt
Example my_object = new Example ();

// Leben: Schreib-Zugriff
my_object.attribute_1 = 5;
my_object.attribute_2 = "Hallo";

// Leben: Lese-Zugriff
if (my_object.attribute_1 > 0) {
  ...
}

// Tod: automatisch
```

Übungen zu diesem Kapitel

Übung 22.1 Einfach Klasse erstellen, einfach

Erstellen Sie eine Klasse `Triangle` zur Repräsentation von Dreiecken. Die Dreiecke seien durch drei Punkte (x_1, y_1), (x_2, y_2) und (x_3, y_3) gegeben, zu deren Darstellung in Java Sie bitte sechs Attribute vom Datentyp **`float`** verwenden.

Geben Sie dann den Code an, um ein gleichseitiges Dreieck der Größe Ihrer Wahl zu erzeugen.

Übung 22.2 Einfache Objekthierarchie erstellen, mittel

Erstellen Sie nun eine Klasse `Point`, die einen Punkt speichert (also zwei **`float`**-Attribute hat). Ändern Sie nun die Definition der Klasse `Triangle` aus der vorherigen Aufgabe, so dass sie nur noch drei `Point`-Attribute hat.

Geben Sie wieder Code an, um ein gleichseitiges Dreieck der Größe Ihrer Wahl zu erzeugen.

Übung 22.3 Objekthierarchie erstellen, schwer

Betrachten Sie das Fenster eines Terminalprogramms. Dieses Fenster soll durch ein Objekt einer Klasse `Window` repräsentiert werden. Diese Klasse wird viele Attribute haben, die auf weitere Objekte verweisen (wie die Menüleiste, der Haupttext, etc.).

Geben Sie hinreichend umfassende mögliche Deklarationen der nötigen Klassen an, um ein solches Terminalfenster zu modellieren.

Prüfungsaufgaben zu diesem Kapitel

Übung 22.4 Klasse erstellen, einfach, typische Klausuraufgabe

Mit einer Java-Klasse soll ein Atom modelliert werden. Dabei soll die Lage des Atoms im Raum (3D-Koordinaten) und die Ordnungszahl gespeichert werden.

- Geben Sie eine Java-Klassendeklaration für das Atom an.
- Wieviel Speicherplatz belegt ein Objekt dieser Klasse?

Übung 22.5 Speicherbedarf abschätzen, einfach, typische Klausuraufgabe

Betrachten Sie folgende Java-Klasse:

```
public class Spider{
  int eyes;
  double x,y;
  boolean is_hairy;
}
```

Wie viel Speicherplatz wird für ein Objekt der Klasse `Spider` benötigt? Wie viel Speicherplatz wird für eine Horde von Spinnen benötigt, die mittels `Spider[] horde = new Spider[1000]` deklariert wurde?

23-1

Kapitel 23
Objekte – Methoden und Nachrichten

Objekte beim Tratschen

Lernziele dieses Kapitels

1. Konzept der Nachricht kennen
2. Nichtstatische Methoden implementieren können
3. Konstruktoren kennen und implementieren können

23-2

Inhalte dieses Kapitels

23.1	Einführung zu Nachrichten	194
23.2	Syntax von Nachrichten	195
23.2.1	Verschicken	195
23.2.2	Verarbeiten	196
23.2.3	Antworten senden	197
23.3	Konstruktoren	197
23.3.1	Begriff des Konstruktors	197
23.3.2	Syntax von Konstruktoren	198
	Übungen zu diesem Kapitel	200

Worum es heute geht

Kehren wir kurz zum Ikea-Regal-aus-objektorientierter-Sicht aus dem vorherigen Kapitel zurück. Ein solches Regal hatte viele Attribute, von denen wir nur einige genannt haben (wenn Sie alle Attribute erfahren möchten, dann jobben Sie doch einfach mal bei Ikea in der IT). Mit diesem Objekt kann man nun einiges anstellen: Man kann seine Attribute fleißig setzten, lesen oder verändern, man kann es in Variablen speichern und zwischen Methoden hin- und herreichen. Jedoch sind dies alles *passive* Dinge, die *mit* dem Objekt gemacht werden. In diesem Kapitel werden wir Objekten nun »Leben« einhauchen, die Objekte sollen *selbst-tätig* werden.

Die Idee ist folgende: Man kann ein Objekt bitten, »etwas zu tun«, indem man ihm eine Nachricht schickt. Das Objekt wird aufgrund einer solchen Nachricht eine vorbereitete Reihe von Befehlen abarbeiten und dann gegebenenfalls eine Antwort an den Absender der Nachricht zurückschicken. Bei »vorbereitete Reihe von Befehlen« denken Sie hoffentlich unwillkürlich an »Methoden«, denn diese haben wir ja genau dafür eingeführt, um eben eine vorbereitete Folge von Befehlen aufzuschreiben. Und tatsächlich benutzten wir *Methoden*, um die Reaktion eines Objektes auf eine Nachricht zu spezifizieren.

Bei manchen Objekten fällt es uns leichter uns vorzustellen, dass dieses Objekt selber etwas macht, als bei anderen Objekten. Schicken wir einem Hunde-Objekt die Nachricht »fass!«, dann kann man sich lebhaft vorstellen, was passiert. Schickt man hingegen einem Regal-Objekt die Nachricht »Wie heißt du?«, so ist man eher erschrocken, wenn dieses zurücksäuselt, es heiße »Norrebo. Und was machst du so?«

Am besten stellt man sich deshalb die objektorientierte Modellierung des Ikea-Warenlagers in der Harry-Potter-Variante vor: Alles ist lebendig, hüpft herum und schwatzt mit seinen Nachbar-Objekten. Kunden-Objekte wandeln zwischen den Regal-Objekten hindurch und zeigen mit digitalen Zauberstäben auf Regal-Objekte, die unter mehr oder wenig viel Protest zu den Warenkorb-Objekten schweben, wo sie mit den bereits dort vorhandenen Billy-, Lack-, Observatör- und Värde-Objekten ins Gespräch kommen.

23.1 Einführung zu Nachrichten

23-4

Wiederholung: Die zentralen Eigenschaften von Objekten.

1. Objekte haben eine Identität.
2. Objekte haben Attribute. Attribute sind Eigenschaften wie Höhe, Breite, Oberflächenstruktur oder was auch immer, die man benutzen und verändern kann.
3. *Objekte haben Fähigkeiten.* Man kann Objekten *Nachrichten schicken*. Dazu werden *Objekt-Methoden* benutzt.

23-5

Wiederholung: Beispiele

```
// In der Datei Student.java
class Student
{
  String nachname, vorname;
  int matrikelnummer;
}

// In der Datei Color.java
class Color
{
  float red_part;   // Rot-Anteil; Zahl zwischen 0 und 1
  float green_part; // Grün-Anteil
  float blue_part;  // Blau-Anteil
}
```

23-6

Objekte tauschen Nachrichten aus.

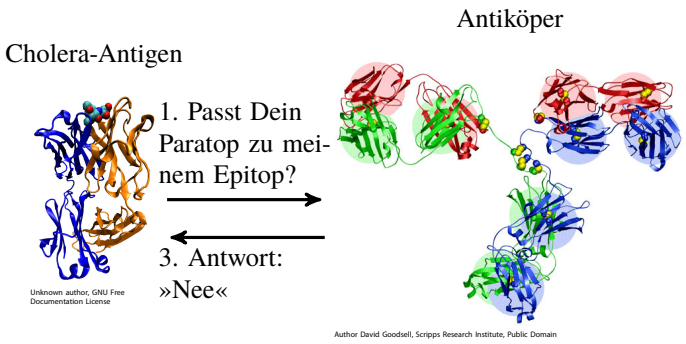
Objekte können anderen Objekten *Nachrichten* schicken. Objekte können die *Fähigkeit haben*, auf bestimmte Nachrichten zu *reagieren*. Der Ablauf einer Nachricht ist folgender:

1. Objekt A schickt *eine Nachricht an Objekt B*. Objekt A *wartet* nun auf Antwort.
2. Objekt B hat die *Fähigkeit*, auf die Nachricht zu reagieren. Dazu führt es *Berechnungen* durch.
3. Objekt B schickt *eine Antwort zurück*.
4. Objekt A kann nun *weiter rechnen*.

23-7

Beispiel eines Nachrichtenaustauschs.

Wir haben ein Antigen-Objekt und ein Antikörper-Objekt. Das Antigen-Objekt möchte vom Antikörper-Objekt wissen, wie hoch die Affinität von Epitop und Paratop ist. Dazu schickt es eine Nachricht.



23.2 Syntax von Nachrichten

23.2.1 Verschicken

Schritt 1: Abschicken einer Nachricht.

23-8

Wir wollen dem Objekt `main_antibody` (eine Instanz der Klasse `Antibody`) Nachrichten schicken. Die erste Nachricht heißt `moveUp` und soll dem Antikörper bitten, sich nach oben zu bewegen. Die zweite Nachricht heißt `doYouMatch` und fragt den Antikörper, ob sein Paratop zu einem gegebenen Epitop passt. Diese Nachricht bekommt noch einen *Parameter*, nämlich das Epitop.

Syntax der Nachrichtenverschickungen

```
main_antibody . moveUp ()
```

Variable, die auf ein Objekt verweist — Nachrichtenname

```
main_antibody . doYouMatch ("0010101101")
```

Variable, die auf ein Objekt verweist — Nachrichtenname — Parameter

Beispielcode, in dem Nachrichten verschickt werden.

23-9

```
...
// Irgendwo in einem Programm

// Erzeuge neuen Antikörper
Antibody main_antibody = new Antibody ();

// Setze einige Attribute
main_antibody.x = 5;
main_antibody.y = 6;
main_antibody.paratope = "0010101101";

// Schicke main_antibody die Nachricht moveUp
main_antibody.moveUp ();

// Schicke main_antibody nochmal die Nachricht moveUp
main_antibody.moveUp ();

// Schicke main_antibody die Nachricht doYouMatch
if (main_antibody.doYouMatch ("1101010010")) {
  System.out.println ("you␣match");
}
```

✎ Zur Übung

23-10

Finden Sie alle Stellen, wo einem Objekt eine Nachricht geschickt wird:

```
Antibody a = new Antibody ();
String s = "00101000101";

a.x = Math.sqrt (9);
a.x = a.y;

a.moveUp ();

if (s.length () > 5 && a.doYouMatch (s)) {
  System.out.println ("Schwierig");
}
```

23.2.2 Verarbeiten

23-11

Schritt 2: Verarbeitung der Nachricht.

Idee

Das Nachrichtenprotokoll folgt EVA: Das Objekt erhält eine *Eingabe* (die Nachricht), es führt eine *Verarbeitung* (Berechnung) durch und es liefert eine *Ausgabe*. Nun galt: *»Methoden sind kleine EVAs.«* Deshalb benutzen wir *besondere Methoden, um Nachrichten zu verarbeiten*.

Syntax der Methoden zur Nachrichtenverarbeitung

- **`static`** *fehlt*.
- Der Methoden-Code wird »vom Objekt ausgeführt«.
- Im Code verweist die Variable **`this`** auf das Objekt.
- Der Rückgabewert der Methode ist gerade der Wert, der an den Absender der Nachricht zurückgesendet wird.

23-12

Beispielcode, der die Nachrichten verarbeitet.

```
class Antibody {
  // Attribute
  int x, y;
  String paratope;

  // Erste Objekt-Methode
  void moveUp ()
  {
    this.y = this.y + 1;
  }

  // Zweite Objekt-Methode
  boolean doYouMatch (String epitope)
  {
    if (this.paratope.equals(epitope)) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

23-13

✎ Zur Übung

Ein *Zeit-Objekt* speichert eine Uhrzeit, bestehend aus Stunden, Minuten und Sekunden:

```
class Time
{
  int hours;
  int minutes;
  int seconds;
}
```

Geben Sie den Code zweier Methoden an, die auf folgende Nachrichten reagieren:

1. Im Objekt gespeicherte Zeit soll auf Mitternacht zurückgesetzt werden.
2. Im Objekt gespeicherte Zeit soll um eine Minute vorgestellt werden.

Regie

Der Ablauf der nächsten Kommunikation wird von Freiwilligen vorgemacht.

Beispielablauf einer Kommunikation. 23-14

```
Studi a = new Studi ();
Studi b = new Studi ();
Studi c = new Studi ();

a.name = "...";
b.name = "...";
c.name = "...";

a.sprich("Hallo.");
a.vorstellung ();

b.sprich("Hallo␣"+a.name);
b.vorstellung ();

c.vorstellung_von (b);
```

```
class Studi {
  String name;

  void sprich (String text)
  {
    System.out.println (text);
  }

  void vorstellung ()
  {
    this.sprich
      ("Ich␣bin␣" + this.name);
  }

  void vorstellung_von(Studi s)
  {
    s.sprich
      ("Das␣ist␣" + this.name);
  }
}
```

23.2.3 Antworten senden

Schritt 3: Antworten auf eine Nachricht. 23-15

Ein Objekt kann auf eine Nachricht *antworten*, indem es am Ende einer Methode einen Wert mit **`return`** zurückgibt. An dem Ort, an dem die Nachricht verschickt wurde, geht die Berechnung erst weiter, wenn die Antwort angekommen ist. Soll auf eine Nachricht nicht geantwortet werden, so gibt man als Typ **`void`** an. Auch in diesem Fall geht die Berechnung erst weiter, wenn die Methode fertig abgearbeitet wurde.

23.3 Konstruktoren

23.3.1 Begriff des Konstruktors

Konstruktoren dienen der Bequemlichkeit. 23-16

Die Erzeugung eines Objektes geschieht oft in zwei Phasen:

1. Das Objekt wird mittels `a = new Antibody ()` erzeugt.
2. Die Attribute des Objektes werden mittels `a.x = 0; a.y = 0;` auf Startwerte gesetzt.

Dieses Verfahren lässt sich mit *Konstruktoren* vereinfachen: Konstruktoren sind spezielle Methoden, die unmittelbar nach der Erzeugung eines Objektes automatisch einmalig aufgerufen werden. Ihre Aufgabe ist es, die Attribute des Objektes auf sinnvolle Anfangswerte zu setzen.

23.3.2 Syntax von Konstruktoren

23-17

Syntax von Konstruktoren.
Der Name des Konstruktors einer Klasse wie `Antibody` ist der *Name der Klasse* (also wiederum `Antibody`). Er hat *keinen Rückgabewert*, noch nicht einmal **`void`**. Man ruft den Konstruktor nicht explizit auf, sondern er wird automatisch aufgerufen, wenn ein Objekt mittels **`new`** erzeugt wurde.

Beispiel

```
class Antibody {
  int x,y;
  String paratope;

  // Konstruktor:
  Antibody ()
  {
    this.x = 0;
    this.y = 0;
  }
}
```

```
...
Antibody a, b;

// Noch nichts passiert

a = new Antibody ();
// Jetzt wurde der
// Konstruktor
// aufgerufen

b = new Antibody ();
// Jetzt nochmal,
// aber für das
// zweite Objekt
```

23-18

Syntax von Konstruktoren mit Parametern.
Konstruktoren können auch Parameter haben, deren konkrete Werte bei **`new`** mit angegeben werden. Eine Klasse kann mehrere Konstruktoren haben, die sich dann aber in den Parametern unterscheiden müssen. Der richtige Konstruktor wird automatisch ausgewählt.

Beispiel

```
class Antibody {
  int x,y;
  String paratope;

  // Konstruktor:
  Antibody
    (int a, int b)
  {
    this.x = a;
    this.y = b;
  }
}
```

```
...
Antibody a;

// Noch nichts passiert
```

```
a = new Antibody (2,3);
// Jetzt wurde der
// Konstruktor
// aufgerufen

// Jetzt gilt
// a.x == 2 und a.y == 3
```

✎ Zur Übung

Entwerfen Sie zwei sinnvolle Konstruktoren für die Zeit-Klasse.


```
class Time
{
  int hours;
  int minutes;
  int seconds;
}
```

Zusammenfassung dieses Kapitels

▶ Syntax für den Aufruf von Objektmethoden (= Nachricht versenden)

```
some_object.methodName(parameter1,parameter2,...);
```

▶ Syntax für Objektmethoden

```
return_type methodName (typ1 parameter1, typ2 parameter2, ...)
{
  // Körper, in dem auf this zugegriffen werden kann
  return some_value;
}
```

▶ Syntax für Konstruktoren

```
ClassName (typ1 parameter1, typ2 parameter2, ...)
{
  // Körper, in dem auf this zugegriffen werden kann
}
```

▶ Welche statische Methode einer nichtstatischen entspricht

Statt

```
class Antigen {
  int x,y;
  void moveUp (int how_far) {
    this.y = this.y + how_far;
  }
}
...
Antigen a = new Antigen ();
a.moveUp();
```

könnte man auch schreiben:

```
class Antigen {
  int x,y;
  static void moveUp (Antigen that, int how_far) {
    that.y = that.y + how_far;
  }
}
...
Antigen a = new Antigen ();
Antigen.moveUp(a);
```

Übungen zu diesem Kapitel

Übung 23.1 Konstruktoren erstellen, leicht

Erweitern Sie die Klasse `Triangle` aus Übung 22.1 um

1. einen Konstruktor ohne Parameter:

```
public Triangle()
```

2. einen Konstruktor, bei dem die Koordinaten der drei Punkte als Parameter übergeben werden:

```
public Triangle(float x1, float y1,
                float x2, float y2,
                float x3, float y3)
```

Übung 23.2 Methoden erstellen, mittel

Erweitern Sie die Klasse aus der vorherigen Aufgabe weiterhin um eine Methode zur Berechnung des Flächeninhalts des Dreiecks:

```
public float area()
```

Tipp: Wenn man zwei Vektoren $\binom{a}{b}$ und $\binom{c}{d}$ gegeben hat, so gibt die Determinante $a \cdot d - b \cdot c$ den Flächeninhalt eines Parallelogramms an, dessen Seiten diese Vektoren sind.

Noch ein Tipp: Ja, das hat tatsächlich viel mit dem Flächeninhalt des gesuchten Dreiecks zu tun.

Übung 23.3 Methoden erstellen, leicht

Erweitern Sie die Klasse aus der vorherigen Aufgabe als nächsten um eine Methode zur Verschiebung des Dreiecks um einen gegebenen Vektor (Δ_x, Δ_y):

```
public void move( delta_x, delta_y )
```

Übung 23.4 Dreiecke zeichnen, mittel

Implementieren Sie nun eine Methode zum Zeichnen des Dreiecks:

```
public void draw( Window w )
```

Die `Window`-Klasse finden Sie im Veranstaltungswiki. Dort befindet sich auch ein kleines Beispielprogramm, das die Methoden der `Window`-Klasse erläutert.

Hinweis: Bei der Implementierung von `draw` wird Ihnen die Methode `Math.round` von Nutzen sein, die eine gegebene Zahl von `float` durch Rundung nach `int` umwandelt.

Übung 23.5 Vollständiges Programm schreiben, mittel

Testen Sie Ihre `Triangle`-Klasse, indem Sie eine neue Klasse `TriangleTest` erstellen, die folgende statische Methode enthalten soll:

```
static void sierpinsky( Triangle t, Window w )
{
  if( t.area() < 10 ) {
    t.draw(w);
    return;
  }
  t.x2 = (t.x1+t.x2)/2; t.y2 = (t.y1+t.y2)/2;
  t.x3 = (t.x1+t.x3)/2; t.y3 = (t.y1+t.y3)/2;
  t.move( t.x2-t.x1,t.y2-t.y1 );
  sierpinsky( t, w );
  t.move( t.x1-t.x3,t.y3-t.y1 );
  sierpinsky( t, w );
  t.move( t.x3-t.x2,t.y2-t.y3 );
  sierpinsky( t, w );
  t.x2 = 2*t.x2-t.x1; t.y2 = 2*t.y2-t.y1;
  t.x3 = 2*t.x3-t.x1; t.y3 = 2*t.y3-t.y1;
}
```

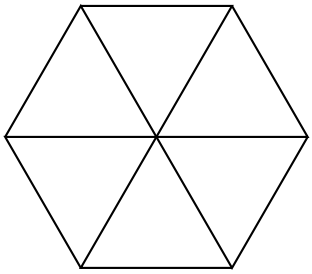
Fügen Sie dazu in Ihre Klasse `TriangleTest` eine `main`-Methode ein, in der ein neues Fenster und ein neues Dreieck geeigneter Größe erzeugt werden und dann mit diesen Objekten die Methode `sierpinsky` aufgerufen wird.

Übung 23.6 Rekursion anpassen, schwer

Versuchen Sie, die generierte Figur aus der vorherigen Aufgabe »hübsch« zu färben.

Übung 23.7 Vollständiges Programm schreiben, mittel

Schreiben Sie ein Programm (Klasse mit `main`-Methode), in dem unter Verwendung von genau zwei `Triangle`-Objekten folgende Figur gezeichnet wird:



Prüfungsaufgaben zu diesem Kapitel

Übung 23.8 Objekte und Zeiger, leicht, original Klausuraufgabe

Betrachten Sie die folgenden Klassen und das darunterstehende Java-Programm:

```
class Mann{
   String name;
   Frau ehefrau;
   public Vater( String name, Frau ehefrau ){
      this.name = name; this.ehefrau = ehefrau;
   }
}
class Frau{
   String name;
   Mann ehemann;
   public Mutter( String name, Mann ehemann ){
      this.name = name; this.ehemann = ehemann;
   }
}
class Kind{
   String name;
   Frau mutter;
   Mann vater;
   public Frau( String name, Frau mutter, Mann vater ){
      this.name = name; this.mutter = mutter; this.vater = vater;
   }
}
```

```
Mann theo = new Mann( "Theo", null );
Mann klaus = new Mann( "Klaus", null );
Frau julia = new Frau( "Julia", theo );
theo.ehefrau = julia;
Kind susi = new Kind( "Susi", julia, theo );
Kind gesi = new Kind( "Gesi", new Frau( "Maria", klaus ), klaus );
Kind ralfi = new Kind( "Ralfi", theo.ehefrau, gesi.mama.ehemann );
klaus.ehefrau = gesi.mutter;
```

Offensichtlich soll hier eine Art Stammbaum modelliert werden. Beantworten Sie folgende Fragen, indem Sie die Verweise zwischen den erzeugten Objekten nachvollziehen:

- Wer ist der Ehemann von Julia?
- Wer ist die Ehefrau von Klaus?
- Wer sind die Eltern von Susi?
- Wer sind die Eltern von Gesi?
- Wer sind die Eltern von Ralfi?

24-1

Kapitel 24
Objektorientierte Modellierung
Das Abbild der Wirklichkeit im Rechner

24-2

Lernziele dieses Kapitels

1. Modellierung von Daten mittels Klassen verstehen
2. Klassen zur Modellierung von Fallbeispielen erstellen können

Inhalte dieses Kapitels

24.1	Einführung	203
24.1.1	Was ist Modellierung?	203
24.1.2	Erst Analyse, dann Entwurf	204
24.2	Analyse	205
24.2.1	Ziel	205
24.2.2	Vorgehen	205
24.2.3	Beispiel: Bibliothek	205
24.3	Entwurf	206
24.3.1	Ziel	206
24.3.2	Vorgehen	206
24.3.3	Beispiel: Bibliothek	206
	Übungen zu diesem Kapitel	207

Worum es heute geht

Das Erlernen einer Programmierensprache ist in gewisser Weise durchaus vergleichbar mit dem Erlernen einer natürlichen Sprache: Der Anfang ist schwierig und braucht viel Übung. Die Syntax und die Grammatik wollen beherrscht werden, man muss sich an merkwürdige, teils völlig widersinnige Sprachkonstrukte gewöhnen. Will man ausdrücken, dass zwei Dinge gleich sind, so muss man in Java komischerweise »`a == b`« schreiben. Möchte man im Französischen einfach nur wissen »Was ist das?«, so muss man »Was ist das, das das ist?« formulieren, »kesskeze?« sagen und »Qu'est-ce que c'est?« schreiben. Diese Merkwürdigkeiten erlernt man am besten, indem man an kleinen, hoffentlich interessanten Aufgaben so viel wie irgend möglich übt. Wenn man dann mit seinem Schulfranzösisch in einem französischsprachigen Land erstmals unterwegs ist, versteht man aber oft niemanden und keiner versteht einen. Dies liegt daran, dass der Sprachalltag erheblich von der behüteten Lernsituation in einer Schulklasse abweicht. Ganz ähnlich liegen die Dinge beim Programmieren.

Ihre zweite Woche bei einer kleinen Bio-Tech-Firma. Arbeitspaket in Ihrem Maileingang: Programmieren eines Web-Service für eine Moleküldatenbank. Verwirrung: Web-Service? Was ist das? Und was ist zu tun? Projektleiter hat leider keine Zeit, ist Häppchen essen. Vertretung ist im Urlaub, es ist schließlich August. Kauf eines Buches zum Thema Web-Services. Kleine Erleuchtungen. Gewünschte Funktionalität des Programms leider nicht in Erfahrung zu bringen. Code programmiert. Neue Mail mit Bitte um Beachtung der geänderten Datenbankschnittstelle. Code neu programmiert. Treffen mit Stellvertreter des stellvertretenden Projektleiters. Code neu programmiert. Treffen mit Administrator der Datenbank. Code neu programmiert. Krisen-Team-Sitzung. Code neu programmiert. Web-Service wird doch nicht gebraucht. Code gelöscht. Große Erleichterung.

In diesem Kapitel verlassen wir den wohlbehüteten einführenden Rahmen dieser Vorlesung und schauen uns (allerdings eher von Weitem) die raue Wirklichkeit der Softwareentwicklung an. Bei »echten« Softwareprojekten ist das Problem nicht, eine For-Schleife korrekt zu implementieren. Es ist noch nicht einmal das Problem, den richtigen Algorithmus auszu-

wählen. Das eigentliche Problem ist, die *Anforderungen* an das Programm im *Vorhinein* zu klären und das Programmieren zeitlich und inhaltlich *sinnvoll zu planen*.

Merken Sie sich bitte unbedingt den Hauptsatz der Softwaretechnik: *Erst planen, dann programmieren.*

24.1 Einführung

24.1.1 Was ist Modellierung?

Arten der Modellierung

24-4

Bei der *Modellierung* (im IT-Kontext) will man *einen Ausschnitt der Wirklichkeit* abbilden. Dazu *analysiert* man, was wichtig ist, und erstellt daraus ein *Modell*. Es gibt viele Arten von Modellen, wie zum Beispiel:

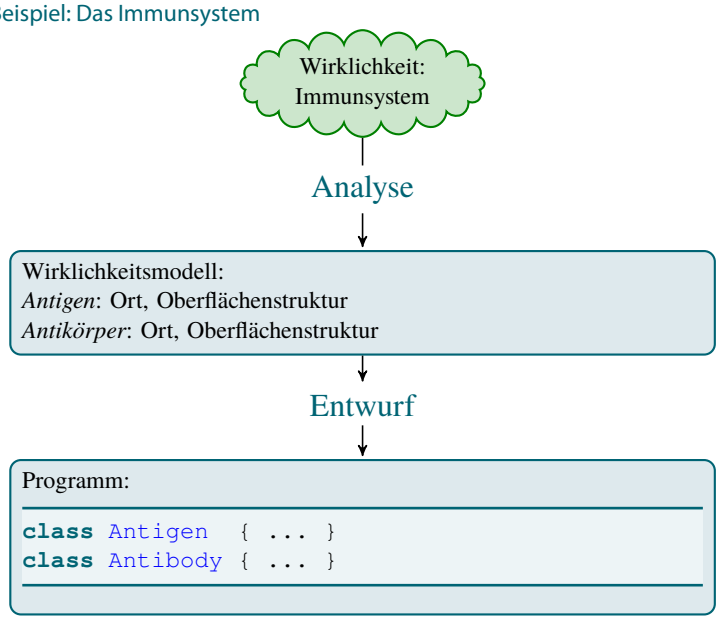
- Datenmodell (Wer? Was?)
- Aktivitätsmodell (Wer? Wann?)
- Sequenzmodell (Zuerst? Danach?)
- Anwendungsfallmodell (Was wäre wenn?)

Die Modelle sind Startpunkte für die Entwicklung der Software. Während der Entwicklung schaut man immer wieder, ob die Software den Modellen (noch) entspricht.

Ein Modell bildet ein Problem ab.

24-5

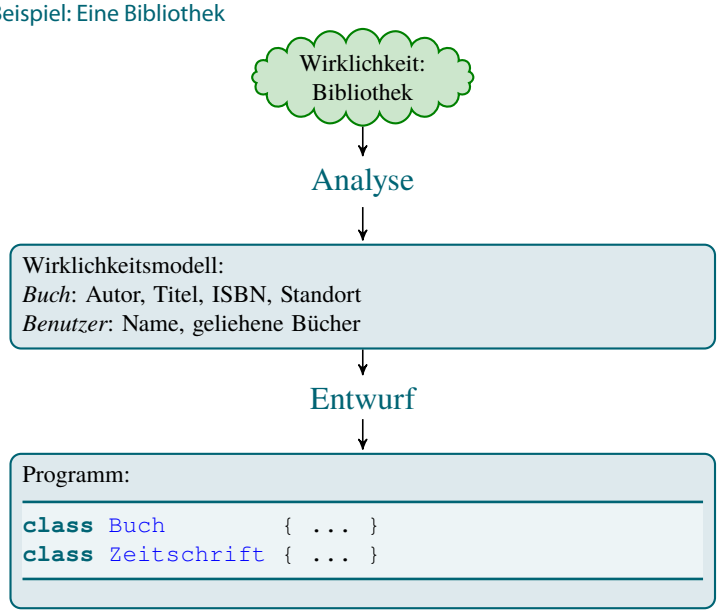
Beispiel: Das Immunsystem



24-6

Ein Modell bildet ein Problem ab.

Beispiel: Eine Bibliothek



24.1.2 Erst Analyse, dann Entwurf

24-7

Es gibt viele Verfahren, Modelle zu erstellen.

Das Teilgebiet der *Softwaretechnik* beschäftigt sich (unter Anderem) damit, welche Verfahren besonders geeignet sind, Analysen und Modellierung durchzuführen. Genau wie es viele Programmiersprachen gibt, gibt es viele Verfahren zur Analyse und Modellbildung; und genau wie Programmiersprachen Vor- und Nachteile haben und einige mehr en vogue sind als andere, haben Analyse- und Modellierungsmethoden Vor- und Nachteile und sind en vogue oder auch nicht. Wir werden im Folgenden eine einfache Form der weit verbreiteten *objektorientierten Analyse und Modellierung* anschauen. (Das trifft sich, da Java eine objektorientierte Sprache ist.)

24-8

Hauptsatz der Softwareentwicklung.

Merke

Erst nachdenken, was eigentlich gebraucht wird; programmiert wird später.

Dies widerspricht der Art, wie man im Allgemeinen Programmieren erlernt: Man bekommt klar beschriebene Aufgaben, die dann in Code umgesetzt werden müssen, wodurch man einen Aspekt der Programmierung erlernt. »In der Wirklichkeit« gibt es aber keine klaren Aufgaben sondern diffuse Wünsche und Unbehaglichkeiten. Softwaretechniker beklagen zu Recht, dass viel zu viel »wild rumprogrammiert« wird.

24-9

Modellierung bei größeren Projekten.

1. Man teilt das Problem *top-down* in Teilbereich ein (siehe das Kapitel »Modularisierung im Großen«).
2. Für jedes Paket führt man nun *bottom-up* eine Analyse durch: Man findet alle relevanten Begriffe und Beziehungen und ordnet sie.
3. Dann führt man einen *Entwurf* durch: Geeignete Klassen, Objekte, Attribute und Methoden werden identifiziert, die tatsächlich in der Software vorkommen sollen.

24.2 Analyse

24.2.1 Ziel

Wozu macht man eine Analyse?

24-10

Ziel der Analyse ist es herauszufinden, was alles zum zu modellierenden Ausschnitt der Realität gehört. Dies schließt alles ein, was für die spätere Anwendung relevant sein könnte. *Insbesondere kann die Analyse Dinge einschließen, die später gar nicht im Programm vorkommen werden.*
Beispiel: Man möchte Software für einen Geldautomaten entwickeln. Dann wird in der Analyse sicherlich der Mensch vorkommen, der Geld haben will – in der späteren Software gibt es aber wahrscheinlich keine `class Human`.

24.2.2 Vorgehen

Ein einfaches Vorgehen bei der statischen objektorientierten Analyse.

24-11

Man kann (und sollte) viele *Aspekte* eines Systems analysieren wie Verhalten, Datenaufbau, Anwendungsfälle und vieles mehr. Wir beschränken uns auf das *statische Datenmodell*, welches man so findet:

1. Man erstellt eine Liste aller *Begriffe*, die für das Gesamtsystem wichtig erscheinen.
2. Dann wird »aufgeräumt« und »vereinheitlicht«: gleichartige Begriffe werden zusammengefasst (Clustering) und Standardbegriffe festgelegt.
3. Man erstellt eine Liste aller *Beziehungen* zwischen den Begriffen wie: »ist ein Teil von« (»part-of«), »ist ein Spezialfall von« (»is-a«), »hängt ab von«, …

24.2.3 Beispiel: Bibliothek

Die Wirklichkeit: Eine Bibliothek

24-12

Es soll eine Software zur Verwaltung des Bestandes einer Institutsbibliothek erstellt werden. Sie sind bei *Molecular Sheep* angestellt und sollen diese Software entwerfen. Zunächst soll nur die Buchverwaltung durch Software unterstützt werden (und nicht die Ausleihe, Bestellung, etc.).

Copyright by Stephan Brunker, GNU Free Documentation License

Schritt 1: Die Begriffsliste.

24-13

Der erste Schritt der Analyse war die Erstellung einer Liste interessanter Begriffe wie: Buch, Zeitschrift, Raum, Regal, Signatur, Buchtitel, Autor, Artikeltitel, …

✎ Zur Übung

Ergänzen Sie die Liste. Die Ergebnisse werden dann gesammelt.

Schritt 2: Vereinheitlichung.

24-14

Der zweite Schritt ist, Begriffe zu vereinheitlichen. Heraus kommt eine Art Glossar:

Titel Dieser Begriff schließt alle Arten von Titeln wie Buchtitel, Artikeltitel oder Zeitschriftentitel ein.

Autor Dieser Begriff bezeichnet speziell nur die Personen, die selber einen Text geschrieben haben. Er bezeichnet nicht Herausgeber, dies ist ein Extrabegriff.

Herausgeber Siehe Autor. Eine Person kann sowohl Autor wie Herausgeber eines Buches sein.

Buch …

✎ Zur Übung

Führen Sie das Clustering für die Liste aus der vorherigen Aufgabe durch. Die Ergebnisse werden dann gesammelt.

24-15

Schritt 3: Beziehungen finden.

Der nächste Schritt ist, Beziehungen zwischen den jetzt ermittelten Begriffen zu finden.

- Ein Buch *hat* Autoren und/oder Herausgeber.
- Eine Zeitschrift *hat* einen Herausgeber.
- Ein Buch *steht* in einem Regal.
- Ein Regal *steht* in einem Raum.
- Ein Konferenzband *besteht aus* Artikeln.

✎ Zur Übung

Finden Sie möglichst viele Beziehungen, in denen ein *Autor* involviert ist.

24.3 Entwurf

24.3.1 Ziel

24-16

Der Entwurf bildet einen Teil der Wirklichkeit ab.

Ziel des *Entwurfs* ist es, die Wirklichkeit in Software abzubilden. Bei einer objektorientierten Sprache ist dies *besonders einfach*, da man viele Begriffe (wie »Buch«) direkt auf Klassen der Programmiersprache abbilden kann. Es wird aber *nicht* zu jedem Begriff eine Klasse eingeführt; einige Begriffe werden sogar *gar nicht* in der Software widergespiegelt.

24.3.2 Vorgehen

24-17

Ein einfaches Vorgehen bei dem statischen objektorientierten Entwurf.

Wieder behandeln wir nur die statischen Datenaspekte. Wir erstellen also nur ein *statisches Datenmodell*.

1. Man teilt die in der Analyse gefundenen Begriffe in *Klassen, Objekte und Attribute* ein. Dies gilt nur für die Begriffe, die tatsächlich in Software abgebildet werden sollen.
2. Man legt so genannte *Vererbungsbeziehungen* zwischen den Klassen fest. (Lassen wir in dieser Vorlesung weg.)
3. Man legt geeignete Methoden für die Klassen fest. (Dazu benötigt man eigentlich eine *Verhaltensanalyse*, die wir aber nicht gemacht haben.)

24.3.3 Beispiel: Bibliothek

24-18

Schritt 1: Identifikation der Klassen, Objekte, Attribute.

Man geht die Begriffe durch und entscheidet, ob sie Klassen sein sollen oder Attribute. Aus den Begriffen »Autor«, »Name«, »Buch« und »ISBN« könnte werden:

```
class Autor {
  String vorname;
  String nachname;
}

class Buch {
  Autor[] autoren; // Liste der Autoren
  String  titel;
  String  isbn;
}
```

✎ Zur Übung

Wie sollten die Begriffe »Herausgeber«, »Regal« und »Gebäude« modelliert werden?

Schritt 3: Bestimmung der Methoden.

Die Objekte der entworfenen Klassen haben ein *erwünschtes Verhalten*, das durch *Methoden* modelliert wird. (Die gewünschte Verhalten findet man durch eine *Verhaltensanalyse*, die wir aber nicht gemacht haben.) Die Methoden werden *noch nicht implementiert*.
Was können Bücher?

```
class Buch {
  // ...

  void   regalwechsel (Regal wohin) {...}
  void   ausmusterung ()            {...}
  int    seitenzahl   ()            {...}
}
```

✎ Zur Übung
Nennen Sie sinnvolle Methoden für eine Klasse `Regal`.

Zusammenfassung dieses Kapitels

▶ Ablauf einer (statischen) OO-Analyse

1. Teile das Thema top-down in sinnvolle Teile auf.
2. Erstelle eine Liste aller Begriffe, die für das Thema irgendwie wichtig sein könnten.
3. Vereinheitliche die Begriffe.
4. Bestimme die Beziehungen der Begriffe untereinander.

▶ Ablauf eines OO-Entwurfs

1. Entscheide, welche Begriffe aus der Analyse überhaupt in Software abgebildet werden sollen.
2. Lege Klassen und Attribute fest, die die Begriffe repräsentieren.
3. Lege fest, welche Methoden die Klassen haben.

Übungen zu diesem Kapitel

Übung 24.1 Objekthierarchien erstellen, schwer

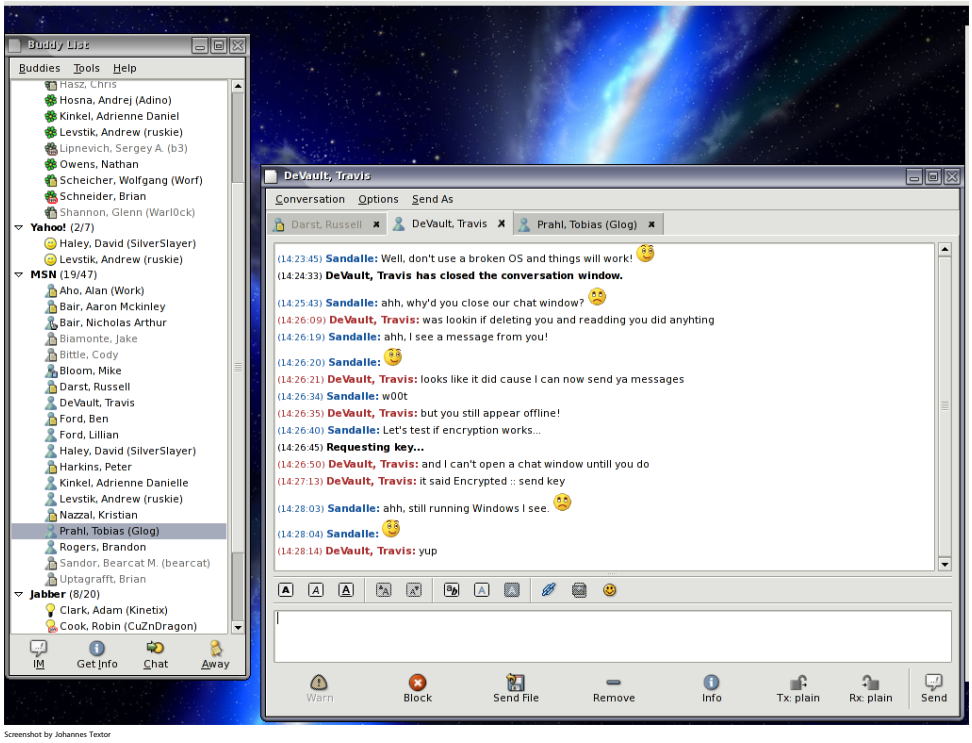
Bei dieser Übung sollen Sie für ein Szenario eine Modellierung durchführen. Dazu sollen Sie die folgenden Schritte durchführen:

1. Ermitteln Sie anhand der Screenshots mindestens fünf Klassen, die vermutlich in dem entsprechenden Programm verwendet werden. Schreiben Sie zu jeder dieser Klassen kurz den Namen und einige Attribute auf.
2. Ermitteln Sie mindestens drei Klassen, die zwar nicht direkt auf dem Screenshot erkennbar sind, aber doch im Programm existieren müssen, um eine sinnvolle Funktion zu gewährleisten. Ein Beispiel für solche »unsichtbaren Helfer« ist die Klasse `FileManagement`, um Dateien zu lesen und zu schreiben.
3. Geben Sie nun für mindestens drei der ermittelten Klassen explizite Deklarationen in Java-Syntax an.

Szenario 1: Spiel »Breakout«

Screenshot by Johannes Textor

Szenario 2: Chat-Programm

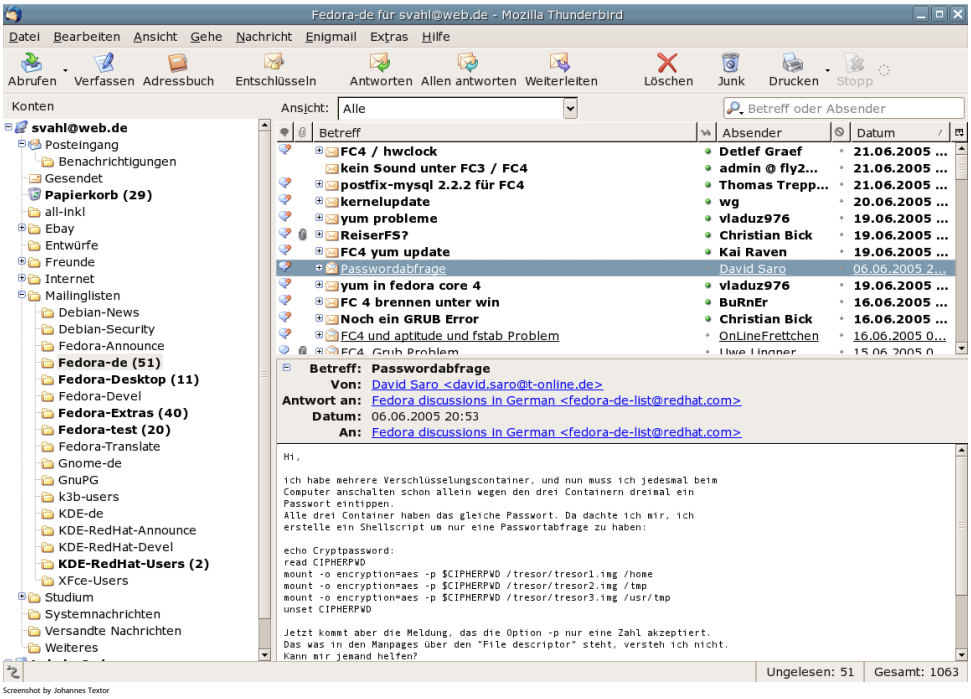


Screenshot by Johannes Textor

Szenario 3: Online-Shop für Bücher CDs und DVDs

Screenshot by Johannes Textor

Szenario 4: E-Mail-Programm



Screenshot by Johannes Textor

Prüfungsaufgaben zu diesem Kapitel

Übung 24.2 Klassen erstellen, mittel, original Klausuraufgabe

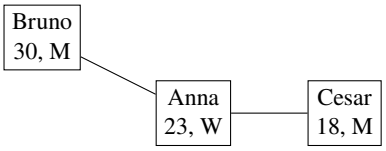
Mit Java-Klassen sollen ein Bücherregal und die darin stehenden Bücher modelliert werden. Dazu wird eine Klasse `Regal` und eine Klasse `Buch` benötigt. Bei den Büchern soll der Titel und die Anzahl der Seiten gespeichert werden. Im Regal sollen bis zu 100 Bücher stehen können.

Geben Sie Java-Klassendeklarationen (nur Attribute, keine Methoden) für die Klassen `Regal` und `Buch` an. Entwerfen Sie auch einen sinnvollen Konstruktor für die Klasse `Buch`.

Geben Sie Java-Code an, mit dem anhand Ihrer Klassen ein Regal erzeugt wird. In dem Regal sollen die 3 Bücher »Moby Dick« (423 Seiten), »Molekulare Zellbiologie« (1251 Seiten), und »Laubsturm« (82 Seiten) stehen. Schreiben Sie dazu keine weiteren Methoden, sondern verwenden Sie nur den **`new`**-Operator, Ihren Konstruktor für die Klasse `Buch` und direkten Zugriff auf die Attribute der erzeugten Klassen.

Übung 24.3 Soziale Netzwerke, leicht bis mittel, original Klausuraufgabe, mit Lösung

Sie sollen ein soziales Netzwerk der nächsten Generation programmieren, das Sie hoffentlich reicher als die Besitzer von Facebook und StudiVZ zusammen machen wird. Zunächst müssen Sie sich aber überlegen, wie Sie Ihr Netzwerk objektorientiert modellieren können. Betrachten Sie folgendes Netzwerk als Beispiel, das die drei Mitglieder Anna, Bruno und Cesar enthält:



Offensichtlich ist also Anna (eine Frau) mit Bruno und Cesar (beides Männer) befreundet, die aber ihrerseits nicht miteinander befreundet sind.

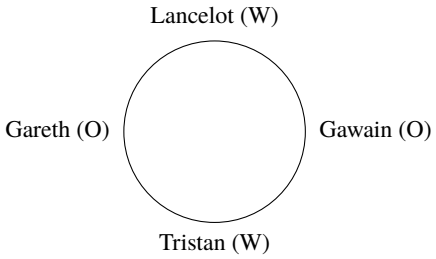
1. Geben Sie die Java-Deklarationen zweier Klassen `Netzwerk` und `Mitglied` an, mit denen die im Beispiel enthaltenen Informationen über Mitglieder des sozialen Netzwerks sowie die Beziehungen zwischen ihnen sinnvoll abgebildet werden. Dabei soll die Klasse `Netzwerk` Verweise auf alle Mitglieder enthalten. Erstellen Sie für die Klasse `Mitglied` außerdem einen sinnvollen Konstruktor.
2. Geben Sie Java-Code an, mit dem das oben dargestellte kleine Netzwerk anhand Ihrer Klassen erzeugt wird.

Übung 24.4 Die Ritter der Tafelrunde, leicht bis mittel, original Klausuraufgabe, mit Lösung

Im England des frühen 12. Jahrhunderts wurden zwischen den Rittern des Königs erbitterte Kriege um die besten Plätze an der königlichen Tafel geführt. Um diese Streitigkeiten zu beenden, erging durch König Artus folgender Erlass:

> »Von nun an tagt die königliche Tafel an einem runden Tisch, der *Tafelrunde*. Die Tischordnung wird folgenden Regeln genügen: Neben jedem Ritter aus dem Osten des Reiches werden zwei Ritter aus dem Westen des Reiches sitzen. Gleicherweise werden neben jedem Ritter aus dem Westen zwei Ritter aus dem Osten sitzen.«

Die erste Tafelrunde im Hause des Königs war wie folgt zusammengesetzt:



(Gareth und Gawain kamen aus dem Osten des Reiches, Lancelot und Tristan aus dem Westen.)

Geben Sie die Java-Deklarationen zweier Klassen `Tafelrunde` und `Ritter` an, mit denen Informationen über solche Tafelrunden sinnvoll abgebildet werden können. Jeder `Ritter` soll insbesondere Verweise auf seinen linken und seinen rechten Nachbarn enthalten. Die Klasse `Tafelrunde` soll Verweise auf alle Ritter enthalten (also nicht nur auf den ersten). Erstellen Sie für die Klasse `Ritter` außerdem einen sinnvollen Konstruktor.

Geben Sie Java-Code an, mit die dargestellte Tafelrunde anhand Ihrer Klassen erzeugt wird.

Fügen Sie Java-Code in die untenstehende Deklaration der Methode `pruefen` der Klasse `Tafelrunde` ein, so dass getestet wird, ob tatsächlich neben jedem Ritter aus dem Osten zwei Ritter aus dem Westen sitzen und umgekehrt. Gehen Sie dabei davon aus, dass alle Verweise zwischen den Rittern und von der Tafelrunde auf die Ritter korrekt gesetzt und nicht `null` sind.

```
class Tafelrunde{
  ...

  boolean pruefen()
    // Fügen Sie hier Ihren Code ein.
  }

}
```

Übung 24.5 Gruppeneinteilung mit Objekten, leicht bis mittel, original Klausuraufgabe, mit Lösung

Für das Informatik-1-Miniprojekt wird Ihr Semester in Gruppen eingeteilt. Hier ein (sehr kleines) Beispielsemester, das in 3 Gruppen eingeteilt ist:

Gruppe 1

Vorname	Nachname
Fritz	Müller
Hans	Meier

Gruppe 2

Vorname	Nachname
Ute	Einsam

Gruppe 3

Vorname	Nachname
Kerstin	Klingel
Susi	Stock

1. Die Modellierung der Gruppeneinteilung in Java soll durch drei Klassen `Semester`, `Gruppe` und `Person` erfolgen. Die Klasse `Person` ist vorgegeben. Vervollständigen Sie die beiden anderen Klassen durch Attribute, so dass sie sich zur Modellierung der Gruppeneinteilung eignen! Implementieren Sie außerdem für beide Klassen je *einen* sinnvollen Konstruktor.

```
class Semester{
  // Hier Attribut/e und genau einen Konstruktor ergänzen
}

class Gruppe{
  // Hier Attribut/e und genau einen Konstruktor ergänzen
}

class Person{
  String vorname, nachname;
  public Person( String vorname, String nachname ){
    this.vorname = vorname; this.nachname = nachname;
  }
}
```

2. Fügen Sie in die `main`-Methode der Klasse `Gruppenverwaltung` Java-Code an, mit dem anhand Ihres Objektmodells die als Beispiel angegebenen Gruppen erzeugt werden!

```
class Gruppenverwaltung{
  public static void main( String[] args ){
    // Fügen Sie hier Ihren Code ein
  }
}
```

Teil VI

Datenstrukturen

Datenstrukturen gehören zu Algorithmen wie der Aktenordner in die Behörde: Algorithmen geben an, mit welchen Schritten man ein Problem löst; Datenstrukturen geben an, wie man die Daten für die Problemlösung organisiert. Zum Beispiel kann man die Zahlen einer Matrix in einem großen Array speichern (erste mögliche Datenstruktur) oder aber auch in einem Array von Arrays (zweite mögliche Datenstruktur). Ähnlich wie ein Problem von verschiedenen Algorithmen gelöst werden kann, kann man auch oft unterschiedliche Datenstrukturen benutzen. Sie haben alle ihre Vor- und Nachteile; die Kunst ist hauptsächlich, diese Vor- und Nachteile zu kennen.

Zur Implementation von Datenstrukturen benutzt man in Java die im vorherigen Teil eingeführten Klassen und Objekte (andere Programmiersprachen machen dies anders). Dabei handelt sich aber, ganz streng genommen, um einen Missbrauch: Die Klassen und Objekte von Datenstrukturen modellieren in aller Regel nichts »reales«, sie werden lediglich intern gebraucht. Bestenfalls könnte man sagen, sie würden eben Akten in einer Behörde modellieren: Auch diese haben keinen »realen Nutzen«, trotzdem gibt es kaum etwas schlimmeres als eine abhandengekommene Akte.

25-1

Kapitel 25

Listen – Konzepte und Erstellung

Eine Schnitzeljagd im Rechner

Lernziele dieses Kapitels

1. Die Liste als Datenstruktur verstehen
2. Code zur Erzeugung von Listen erstellen können
3. Den Unterschied zwischen Listen und Arrays erklären können

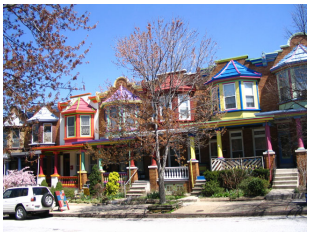
25-2

Inhalte dieses Kapitels

25.1	Einleitung	214
25.1.1	Motivation zu Listen	214
25.1.2	Idee hinter Listen	214
25.2	Die Datenstruktur	215
25.2.1	Idee	215
25.2.2	Umsetzung in Java-Code	215
25.3	Operationen auf Listen	216
25.3.1	Einfügen am Anfang	216
25.3.2	Löschen am Anfang	217
	Übungen zu diesem Kapitel	218

Worum es heute geht

Listen sind eine so genannte *fortgeschrittene Datenstruktur*. »Fortgeschritten« heißen sie, weil sie im Gegensatz zu Arrays eine, wie Sie noch sehen werden, recht komplexe innere Struktur aufweisen, mit Verwaltungsklassen, Zellklassen und wilden Verzeigerungen im Speicher. Der Name »Liste« ist eigentlich eher schlecht gewählt (aber, wie so vieles historisch Gewachsenes, nicht mehr zu ändern). Unter einer Liste stellt man sich landläufig eine zeilenweise untereinandergeschriebene Aufstellung von Wörtern oder Sätzen vor. Jedoch meint man in der Informatik mit »Liste« eine eher chaotische »Verkettung« der Listenpunkte, wo bei jedem Punkt am Ende steht, wo der nächste Punkt zu finden ist.



Public domain

Einen normalen Array kann man sich ganz gut als eine sehr lange Straße vorstellen, an deren Rand nummerierte Häuser stehen. In jedem Haus »wohnt« ein Array-Element, in Haus 0 beispielsweise ein »A«, in Haus 1 ein »C« und in Haus 2 noch ein »C«. (Natürlich fängt die Nummerierung von Häusern in Wirklichkeit bei 1 an, aber die Informatik ist eben nicht die Wirklichkeit.) Was passiert nun, wenn man nach dem, sagen wir, zweiundvierzigsten Haus ein weiteres einfügen will? Dies ist bei Arrays nicht möglich. Vielmehr muss man sich eine neue Straße suchen mit einem Haus mehr und alle Elemente bis zum zweiundvierzigsten Haus in das Haus mit derselben Nummer in der neuen Straße umziehen lassen, dann alle Elemente in Häusern aus der alten Straße mit höheren Nummern in die Häuser mit der eins höheren Nummer in der neuen Straße. Ein reichlich aufwendiger Vorgang, bei dem Kolonnen von Umzugswagen benötigt werden.

Public domain

Eine Liste im Informatiksinne kann man sich eher als Zeltplatz vorstellen. Überall stehen wild verstreut Zelte herum, in denen Elemente hausen. Die genaue Position eines Zeltes auf dem Platz ist völlig unerheblich. Wenn man aber eine Reihenfolge auf den Zelten braucht, so speichert man bei jedem Zelt neben dem Element auch *den Ort, wo sich das nächste Zelt in der Reihenfolge befindet*. Diese Information könnte man zum Beispiel außen auf das Zelt malen. Will man nun die Zelte gemäß diese Reihenfolge besuchen, so geht man zum ersten Zelt, dessen Ort am Eingang des Zeltplatzes auf einer besonderen Tafel steht. Von dort aus besucht man das Zelt, dessen Ort auf dem ersten Zelt steht. Von dort aus das Zelt, dessen Ort auf dem zweiten Zelt steht; und so weiter. Auf dem letzten Zelt steht dann »Ende«. Kommt

ein neuer Zelter, so kann er einfach irgendwo sein Zelt aufschlagen. Auf sein Zelt schreibt man dann den Ort des alten ersten Zeltes (der ja auf der Tafel steht) und schreibt dafür auf die Tafel den Ort des neuen Zeltes. Ähnlich einfach kann man auch Zelte in der Mitte oder am Ende einer Liste einfügen und Zelte können auch recht leicht den Zeltplatz verlassen.

Die Organisation von Daten als Zeltplatz ist sehr vorteilhaft, wenn ständig Leute (Elemente) kommen und gehen. Jedoch dauert bei dieser Organisation das Finden von Zelt Nummer 123 in der Reihenfolge recht lange, man muss 123 Mal kreuz und quer über den Zeltplatz laufen. Bei einer Straße ist es hingegen sehr leicht und schnell möglich, Haus 123 zu finden.

Moral von der Geschicht': Ob man sein Daten als Zeltplatz (=Liste) oder als Straße (=Array) organisiert, hängt hauptsächlich davon ab, wie oft »Daten kommen und gehen«.

25.1 Einleitung

25.1.1 Motivation zu Listen

25-4

Ein Problem aus der Bioinformatik

Beim Shotgun-Sequencing entsteht folgendes Problem:

Problemstellung

Eingaben Sequenzen einer sehr großen Menge kurzer DNS-Stränge

Ausgabe Einzelner DNS-Strang, der alle kurzen Stränge möglichst sinnvoll erklärt.

✎ Zur Diskussion

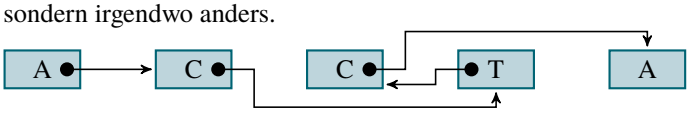
1. Welche Nachteile haben Strings als Darstellung von langen DNS-Sequenzen, die man »zusammenpuzzeln« möchte?
2. Was wären die idealen Eigenschaften einer Datenstruktur zur Darstellung von DNS-Sequenzen?

25.1.2 Idee hinter Listen

25-5

Die Datenstruktur der Liste

Die *Liste* ist eine Datenstruktur, die nach dem Vorbild eines kafkaesquen Amtes aufgebaut ist. Jeder Sachbearbeiter weiß über irgendwas Bescheid, für alles andere wird man zum nächsten Sachbearbeiter geschickt. Der nächste Sachbearbeiter sitzt in der Regel nicht nebenan, sondern irgendwo anders.



25-6

Vor- und Nachteile von Listen gegenüber Arrays und Strings.

Vorteile

Folgende Operationen gehen *sehr leicht* und *sehr schnell*:

+ Einfügen neuer Elemente.
+ Verketten von Listen zu neuen Listen.
+ Löschen vorhandener Elemente.
+ Ausschneiden von Teilen aus einer Liste.

Nachteile

Um das i-te Element zu finden, muss man »i Sachbearbeiter nacheinander aufsuchen«. Deshalb ist binäre Suche *nicht möglich* und man muss immer *lineare Suche* durchführen.

25.2 Die Datenstruktur »einfach verkettete Liste«

25.2.1 Idee

Listen bestehen aus Zellen. 25-7

Eine Liste besteht aus vielen *Zell-Objekten* (»Informatikzellen«, nicht biologische Zellen). Eine Zelle speichert:

1. Einen Wert, wie zum Beispiel eine Base.
2. Einen Verweis auf die nächste Zelle in der Liste.

In der letzten Zelle ist der Verweis auf die nächste Zelle `null`. Das *Listen-Objekt* speichert lediglich einen Verweis auf die erste Zelle.

25.2.2 Umsetzung in Java-Code

Die Listen- und Zell-Klasse 25-8

```
// Datei DNASequence.java
class DNASequence
{
  // Attribute
  Cell start;

  // Konstruktoren
  DNASequence () {
    this.start = null;
  }
}
```

```
// Datei Cell.java
class Cell
{
  // Die Basen ('A', 'C', 'G', 'T')
  char base;

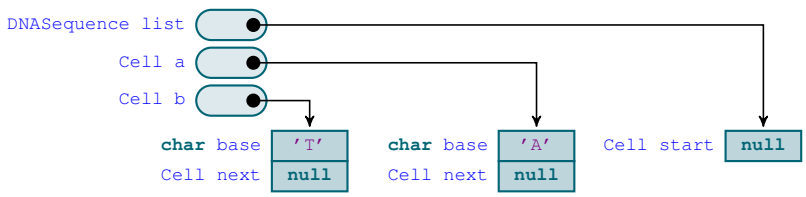
  // Nächstes Listenelement:
  Cell next;
}
```

Erzeugung einer Liste 25-9

Schritt 1

Der folgende Code erzeugt erst ein Listenobjekt und zwei Zellobjekte. Die Zellobjekte haben schon Daten, sind aber noch nicht verkettet.

```
DNASequence list = new DNASequence ();
Cell a = new Cell ();
Cell b = new Cell ();
a.base = 'A';
b.base = 'T';
```



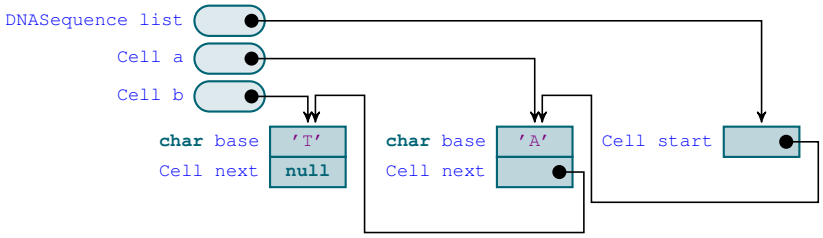
25-10

Erzeugung einer Liste

Schritt 2

Nun werden die Objekte verkettet.

```
...
list.start = a;
a.next = b;
```



25-11

✎ Zur Übung

1. Geben Sie den Code der Klassen `Studentenliste` und `Studentenzelle` zur Verwaltung einer Liste von Studenten.
2. Geben Sie weiter den Code zur Erzeugung einer Studentliste mit drei Studenten an.

25.3 Operationen auf Listen

25.3.1 Einfügen am Anfang

25-12

Einfügen eines neuen Elements am Anfang

Problemstellung

Am Anfang der DNA-Sequenz soll eine neue Base angefügt werden.

Algorithmus

1. Erzeuge eine *neue Zelle* für die neue Base.
2. Der *Nachfolger* dieser neuen Zelle ist der *Start der Liste*.
3. Setze den *Start der Liste* auf die *neue Zelle*.

Beachte: Dieser Algorithmus ist eine *Fähigkeit der Listen-Klasse*: Man kann eine Listen-Klasse durch ein Nachricht »bitten«, ein Element hinzuzufügen. Deshalb fügen wir ihn als Methode zur Klasse `DNASequence` hinzu.

25-13

Code der Methode.

```
class DNASequence {
  // Attribute
  Cell start;

  // Konstruktor
  DNASequence () {
    this.start = null;
  }

  // Methoden
  void addBaseAtFront (char b) {
    Cell new_front = new Cell ();

    new_front.base = b;
    new_front.next = this.start;

    this.start = new_front;
  }
}
```

✎ Zur Übung 25-14
Visualisieren Sie wie zuvor graphisch alle Objekte und Variablen, die folgender Code erzeugt:

```
DNASequence list = new DNASequence ();

list.addBaseAtFront ('A');
list.addBaseAtFront ('C');
list.addBaseAtFront ('T');
```

25.3.2 Löschen am Anfang

Löschen des Elements am Anfang 25-15

Problemstellung
Das Element am Anfang der Liste soll gelöscht werden.

Algorithmus
Ersetze `start` durch den Nachfolger von `start`.

```
class DNASequence {
  ...
  void deleteFirst () {
    this.start = this.start.next;
  }
}
```

✎ Zur Übung 25-16
Visualisieren Sie wie zuvor graphisch alle Objekte und Variablen, die folgender Code erzeugt:

```
DNASequence list = new DNASequence ();

list.addBaseAtFront ('A');
list.addBaseAtFront ('C');
list.addBaseAtFront ('T');
list.deleteFirst ();
list.deleteFirst ();
```

Zusammenfassung dieses Kapitels

▶ Listen versus Arrays 25-17
In einem Array werden Elemente schön »hinteinander weg« gespeichert (Modell Reihenhaus). Der Zugriff auf das *i*te Element ist schnell, Löschen und Einfügen sind langsam. In einer Liste wissen Elemente immer nur, wo das nächste Element ist (Modell Zeltplatz). Der Zugriff auf das *i*te Element ist langsam, Löschen und Einfügen sind schnell.

▶ Zwei Klassen pro Listenart
Man benötigt *zwei Klassen* um Listen einer Art zu modellieren: Zunächst braucht man »Zellen«, die die eigentlichen Daten enthalten und immer einen Verweis auf die nächste Zelle. Als Zweites braucht man die »eigentliche Listenklasse«, die einen Verweis auf den Anfang speichert. *Dieser* Klasse schickt man alle Nachrichten.

Übungen zu diesem Kapitel

Übung 25.1 Datenstruktur Ring entwerfen, mittel, mit Lösung

Ein *einfach verketteter Ring* ist eine Datenstruktur, die sich von der einfach verketteten Liste nur dadurch unterscheidet, dass der `next`-Zeiger des letzten Elements nicht ins Leere (`null`) zeigt, sondern wieder auf das erste Element. Geben Sie Klassendefinition einer Klasse `DNASequenceRing` und der dazugehörigen `Cell` Klasse an.

Übung 25.2 Einelementigen Ring erzeugen, leicht, mit Lösung

Geben Sie eine Folge von Java-Befehlen an, mit denen anhand von `new` und direkten Zugriffs auf die Attribute der erzeugten Objekte ein einfach verketteter Ring `der_eine_ring` mit nur einer Base erstellt wird. Die eine Base in dem Ring soll Adenin sein.

Übung 25.3 Mehrelementigen Ring erzeugen, leicht

Erzeugen Sie nun ähnlich wie eben einen Ring, aber mit den Elementen »T«, »A« und »T«.

Übung 25.4 Ring rotieren, leicht

Erweitern Sie die Klasse `DNASequenceRing` um eine Methode

```
void rotateForward()
```

die den Ring um eine Base »dreht«. (Hinweis: Dazu genügt es, den Anfang des Rings geeignet zu verschieben.)

Übung 25.5 Einfügen in einen Ring, mittel

Erweitern Sie die Klasse `DNASequenceRing` um eine Methode

```
void addBase(char base)
```

die eine Base *direkt nach dem ersten Element* des Rings einfügt. Behandeln Sie die Sonderfälle eines leeren Rings und eines Rings mit nur einem Element gesondert.

Übung 25.6 Löschen aus einem Ring, schwer

Erweitern Sie die Klasse `DNASequenceRing` um eine Methode

```
void deleteSecond()
```

die die zweite Zelle des Rings löscht. Behandeln Sie die Fälle von Ringen mit weniger als drei Elementen besonders und sinnvoll.

Prüfungsaufgaben zu diesem Kapitel

Übung 25.7 Modellierung mit Listen, leicht, original Klausuraufgabe, mit Lösung

In Java soll eine Warteschlange aus Personen, wie zum Beispiel bei einer bekannten Bäckerei im Zentralklinikum, modelliert werden. Dies soll anhand der beiden Klassen `Queue` und `Person` geschehen. Die Klasse `Queue` soll Verweise auf die erste (als nächste drankommende) und die letzte (als letzte drankommende) Person enthalten. Zu jeder Person sollen der Vorname, der Nachname und ein Verweis auf die vor ihr stehende Person gespeichert werden.

1. Geben Sie Java-Klassendeklarationen (nur Attribute, keine Methoden) für die Klassen `Queue` und `Person` an. Entwerfen Sie auch einen sinnvollen Konstruktor für die Klasse `Person`.
2. Geben Sie Java-Code an, mit dem anhand Ihrer Klassen eine Warteschlange mit den Personen *Till Tantau*, *Jan Arpe* und *Johannes Textor* erzeugt wird, wobei *Till Tantau* zuerst und *Johannes Textor* zuletzt drankommen soll.

26-1

Kapitel 26

Listen – Iteration und Modifikation

Von verbogenen Zeigern

Lernziele dieses Kapitels

1. Code zur Iteration über Listen erstellen können
2. Code zur Modifikation von Listen erstellen können

Inhalte dieses Kapitels

26-2

26.1	Iteration	220
26.1.1	Idee	220
26.1.2	Anwendung: Längenbestimmung	221
26.1.3	Anwendung: Ausgabe aller Elemente	222
26.1.4	Anwendung: Map	222
26.1.5	Anwendung: Suche	222
26.2	Modifikation	223
26.2.1	Einfügen von Elementen	223
26.2.2	Löschen von Elementen	224
26.2.3	Verketten von Listen	225
	Übungen zu diesem Kapitel	225

Worum es heute geht

So. Nun ist die Liste da. Und was jetzt?

Im letzten Kapitel ging es darum, Listen aufzubauen – ein speichertechnisch chaotischer, aber doch erfolgreicher Vorgang. Um nun etwas mit der Liste anzufangen, muss man auch »die Liste entlanglaufen« können. Beispielsweise könnte man dann bei jedem besuchten Listenelement (= Zelt) den Inhalt ausgeben (= den Insassen des Zeltes zu einer Unterschrift bewegen auf einer Unterschriftenliste beispielsweise gegen den kommerziellen Walfang, Softwarepatente, globale Erwärmung oder die Revision der Abschaffung der Abschaffung lokalitätsspezifischer Rauchverbote). Dieses Entlanglaufen ist recht einfach durch eine While-Schleife zu implementieren.

Etwas vertrackter wird die Sache, wenn man irgendwo inmitten der Liste Elemente einfügen oder löschen möchte. Letztendlich muss man nur die »richtigen Zeiger umsetzen«, jedoch liefert dies etwas mystischen Code wie

```
delete_next_cell.next = delete_next_cell.next.next
```

Alles klar? Am Ende dieses Kapitels hoffentlich schon.

26.1 Iteration

26.1.1 Idee

26-4

Wie besucht man alle Elemente einer Liste?

Problemstellung

Wir wollen alle Elemente (= Zellen) einer Liste »besuchen« und dort »irgend etwas tun«.

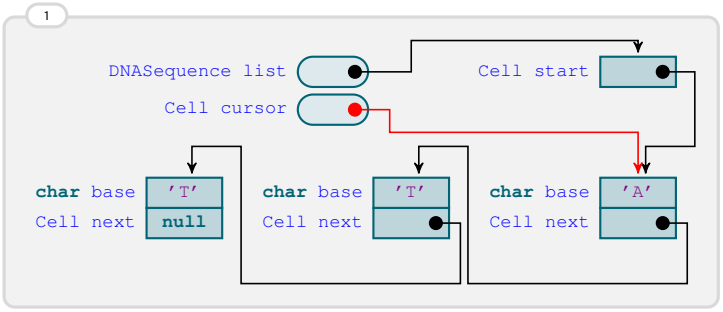
Iterativer Algorithmus

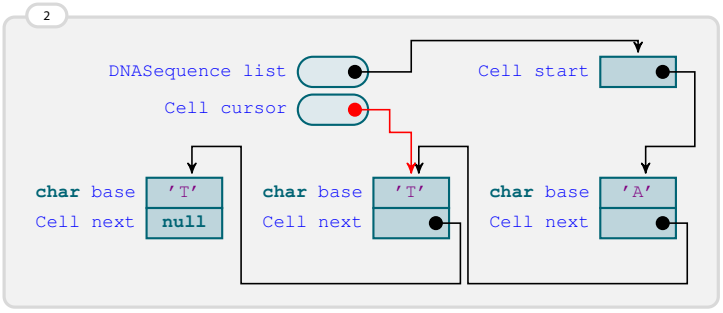
1. Setze `cursor` auf die Startzelle.
2. Tue das Gewünschte für diese Startzelle.
3. Solange `cursor` einen Nachfolger hat, tue:
   - 3.1 Setze `cursor` auf den Nachfolger von `cursor`.
   - 3.2 Tue das Gewünschte für die aktuelle Stelle.

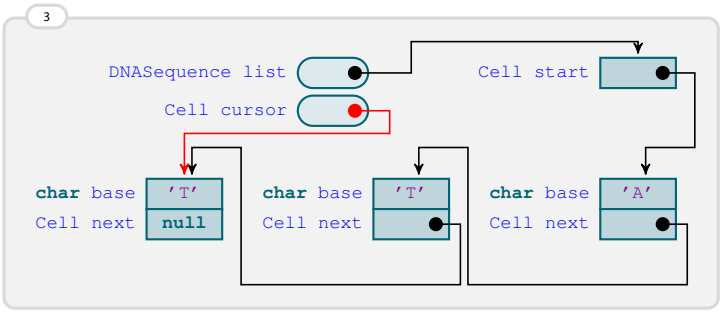
26-5

Wer zeigt wann wohin?

```
Cell cursor = list.start; // Startsituation
tue_was (cursor);
while (cursor.next != null) {
  cursor = cursor.next; // Vorwärts!
  tue_was (cursor);
}
```







Eine effizientere Version des Codes.

26-6

Der Code von eben hat noch zwei Nachteile:

1. Der Aufruf von `tue_etwas` steht zweimal im Code. Dies ist unschön, wenn man stattdessen etwas komplexeres machen möchte.
2. Der Code funktioniert nicht, wenn die Liste leer ist, also sofort `cursor == null` gilt.

Diese Probleme lassen sich wie folgt umgehen:

```
Cell cursor = list.start;
while (cursor != null) {
  tue_etwas (cursor);
  cursor = cursor.next;
}
```

26.1.2 Anwendung: Längenbestimmung

Problemstellung: Die Länge einer Liste bestimmen.

26-7

Die Länge einer Liste »sieht man ihr nicht an«. Man *muss* alle Elemente einmal besuchen und dabei einen Zähler für jedes besuchte Element hochzählen. Die Aktion »tue etwas« ist hier gerade das Hochzählen dieses Zählers.

```
// Algorithmus zum Zählen der Elemente einer Liste
int counter = 0;
Cell cursor = list.start;
while (cursor != null) {
  counter++;
  cursor = cursor.next;
}
```

Die Längenmethode.

26-8

Die Längenberechnung sollte durch eine Methode der Listenklasse implementiert werden. Dann kann man ein Listenobjekt erstellen und es später mittels einer Nachricht fragen, was seine Länge ist.

```
class DNASequence {
  // ...

  public int length () {
    int counter = 0;
    Cell cursor = this.start;
    while (cursor != null) {
      counter++;
      cursor = cursor.next;
    }
    return counter;
  }
}
```

✎ Zur Übung

26-9

Geben Sie den Code einer Methode `count_As`, die die Anzahl an »A«'s in der Liste zurückgibt.

26.1.3 Anwendung: Ausgabe aller Elemente

26-10

Problemstellung: Ausgabe aller Elemente.
Um alle Elemente einer Liste auszugeben, muss man sie einfach alle besuchen. Die Aktion »tue etwas« ist dann gerade die Ausgabe.

```
class DNASequence {
  // ...
  public void print () {
    Cell cursor = this.start;
    while (cursor != null) {
      System.out.print(cursor.base);
      cursor = cursor.next;
    }
  }
}
```

26-11

✎ **Zur Übung**
Geben Sie den Code einer Methode an, die die Basen in der Liste als einen String zurückgibt. Die Idee ist, jede Base nacheinander an das Ende eines Strings anzuhängen:

```
return_me = return_me + cursor.base;
```

26.1.4 Anwendung: Map

26-12

Problemstellung: Verändern aller Elemente.
Wir wollen nun alle Elemente einer Liste verändern, beispielsweise durch ihre Komplemente ersetzen. Die Aktion »tue etwas« ist dann gerade diese Modifikation. Ein solches Verändern aller Element wird in der funktionalen Programmierung *Map* genannt.

```
class DNASequence {
  // ...
  public void complement () {
    Cell cursor = this.start;
    while (cursor != null) {
      if      (cursor.base == 'A') { cursor.base = 'T'; }
      else if (cursor.base == 'T') { cursor.base = 'A'; }
      else if (cursor.base == 'C') { cursor.base = 'G'; }
      else if (cursor.base == 'G') { cursor.base = 'C'; }
      cursor = cursor.next;
    }
  }
}
```

26.1.5 Anwendung: Suche

26-13

Problemstellung: Suche nach einem Element.
Wir wollen nun ein Element finden mit einer bestimmten Eigenschaft; beispielsweise das erste »A«. Die Aktion »tue etwas« ist dann der Test, ob der `cursor` eine Zelle mit der gesuchten Eigenschaft erreicht hat.

```
class DNASequence {
  // ...
  public Cell searchForFirstA () {
    Cell cursor = this.start;
    while (cursor != null) {
      if (cursor.base == 'A') {
        return cursor; // Gefunden! Danke und tschüss.
      }
      cursor = cursor.next;
    }
    return null; // Nicht gefunden. Grrr.
```

```
  }
}
```

26.2 Modifikation

26.2.1 Einfügen von Elementen

Wie fügt man ein Element in die Mitte einer Liste ein?

26-14

Wir wollen ein neues Element nicht am Anfang einer Liste, sondern irgendwo zwischendrin einfügen. Dazu muss man *eigentlich nur lokal die Verkettung ändern*. Genauer braucht man zunächst einen Verweis auf ein Element `a`, *nach dem man* das neue Element `b` einfügen möchte. Dann ändert man zwei Verweise:

- Der Nachfolger von `a` ist nun `b`.
- Der Nachfolger von `b` ist nun der alte Nachfolger von `a`.

Der Code einer Einfügemethode.

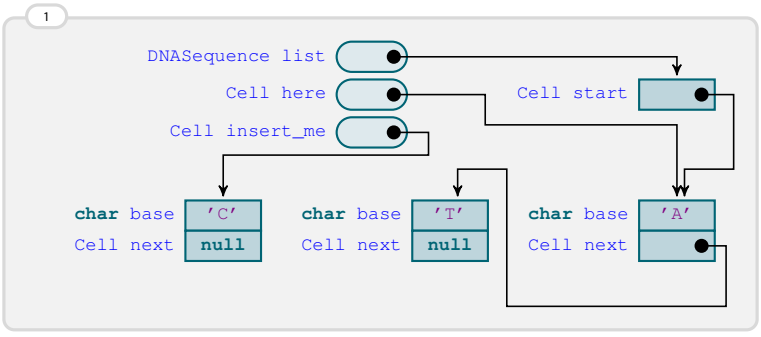
26-15

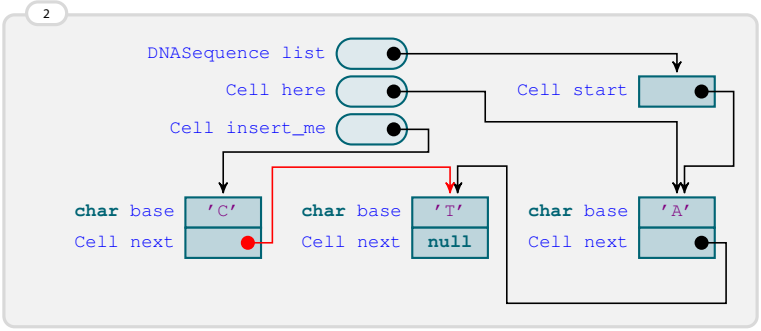
```
class DNASequence {
  // ...
  void insertAfter (Cell here, char b) {
    Cell insert_me = new Cell ();
    insert_me.base = b;
    insert_me.next = here.next;
    here.next      = insert_me;
  }
}
```

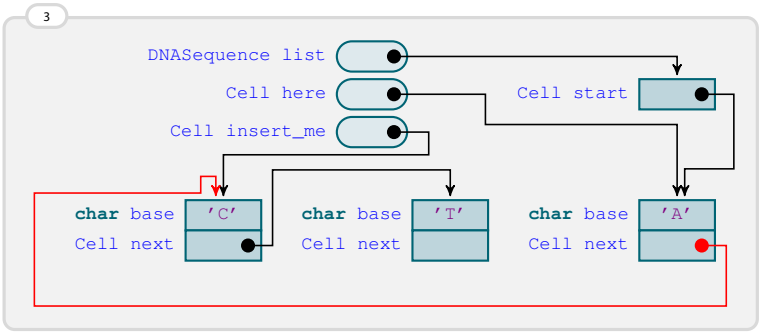
Wer zeigt wann wohin?

26-16

```
                              // 1. Ausgangssituation
insert_me.next = here.next; // 2. Erste Veränderung
here.next      = insert_me; // 3. Fertig
```







26.2.2 Löschen von Elementen

26-17

Wie löscht man ein Element aus der Mitte einer Liste?

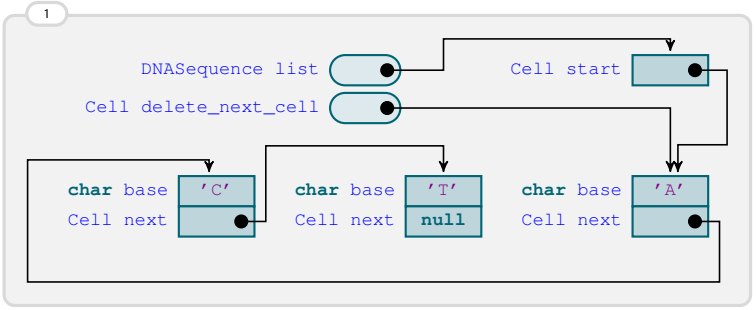
Wir wollen ein Element irgendwo in der Mitte einer Liste löschen. Dazu muss lediglich den *Nachfolger* des Vorgängers ändern.

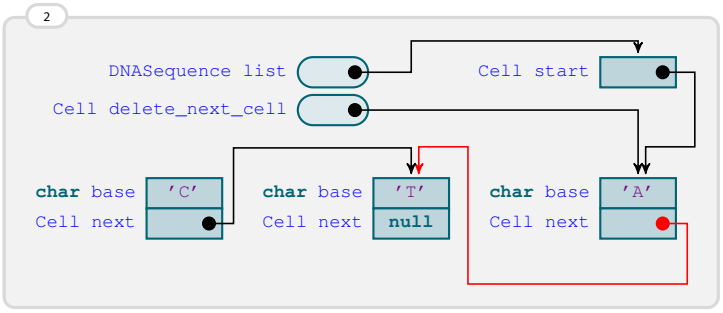
```
class DNASequence {
  // ...
  void deleteAfter (Cell delete_next_cell) {
    // Löscht die Zelle, die auf delete_next_cell folgt:
    delete_next_cell.next = delete_next_cell.next.next;
  }
}
```

26-18

Wer zeigt wann wohin?

```
// 1. Vorher
delete_next_cell.next = delete_next_cell.next.next;
// 2. Nachher
```





26.2.3 Verketten von Listen

Wie verkettet man zwei Listen? 26-19

Wir wollen aus zwei Listen eine Liste machen. Dazu muss der *Nachfolger* des letzten Elements der einen Liste der Anfang der zweiten Liste werden. Dazu muss man allerdings erst »das Ende finden« – dies macht man mit einer Iteration.

✎ Zur Übung

Geben Sie den Code einer Methode zur Verkettung zweier Listen an:

```
class DNASequence {
  // ...
  void concatWith (DNASequence me) {
    // ?
  }
}
```

Zusammenfassung dieses Kapitels

▶ Iteration über alle Elemente einer Liste 26-20

```
Cell cursor = list.start;
while (cursor != null) {
  tue_etwas (cursor);
  cursor = cursor.next;
}
```

▶ Einfügen eines Elements in eine Liste

```
new_cell.next                = insert_after_this_cell.next;
insert_after_this_cell.next = new_cell;
```

▶ Löschen eines Elements aus einer Liste

```
delete_cell_after_this_cell.next =
    delete_cell_after_this_cell.next.next;
```

Übungen zu diesem Kapitel

Übung 26.1 Länge eines Ring bestimmen, mittel, mit Lösung

Erweitern Sie die Klasse `DNASequenceRing` aus Übung 25.1 um eine Methode

```
int length()
```

die die Länge des Rings ausgibt.

Hinweise: Bei einer einfach verketteten Liste würden Sie hier eine **`while`**-Schleife verwenden, die abbricht, wenn der Nachfolger der zuletzt betrachteten Zelle **`null`** ist. Das funktioniert bei einem Ring nicht, da ja jede Zelle immer einen Nachfolger hat. Sie müssen daher in der Abbruchbedingung stattdessen testen, ob der Nachfolger der zuletzt betrachteten Zelle der Ringanfang ist.

Übung 26.2 Länge eines Ring bestimmen, mittel

Erweitern Sie die Klasse `DNASequenceRing` um eine Methode

```
void toString()
```

die die im Ring gespeicherte Sequenz als String zurückgibt.

Übung 26.3 Einfügen in einen Ring, schwer

Erweitern Sie die Klasse `DNASequenceRing` aus Übung 25.1 um eine Methode

```
void addBaseAtFront(char base)
```

die eine Base *vor dem ersten Element* des Rings einfügt, so dass die erste Zelle des Rings hinterher diese neue Base ist.

Hinweise: Das Problem ist, dass die letzte Zelle nun auch auf diese neue Zelle verweisen muss. Dazu muss man die letzte Zelle erstmal finden.

Übung 26.4 Löschen aus einem Ring, schwer

Erweitern Sie die Klasse `DNASequenceRing` um eine Methode

```
void deleteFirst()
```

die die erste Zelle des Rings löscht. Wieder muss der Verweis in der letzten Zelle angepasst werden.

Prüfungsaufgaben zu diesem Kapitel

Folgende Klasse implementiert eine einfach verkettete Liste ganzer Zahlen:

```
class List
{
   Cell start;
}

class Cell
{
   Cell next;
   int number;
}
```

Übung 26.5 Kürzen einer Liste, mittel, original Klausuraufgabe, mit Lösung

Erweitern Sie die Klasse `List` um eine Methode

```
void truncate( int n )
```

die die Liste auf eine Länge von `n` Elementen verkürzt, indem nur die ersten `n` Zellen der Liste beibehalten und der Rest gelöscht werden. Dabei dürfen Sie davon ausgehen, dass die Liste mindestens `n` Elemente enthält.

Übung 26.6 Löschen aus einer Liste, schwer, original Klausuraufgabe, mit Lösung

Erweitern Sie die Klasse `List` um eine Methode

```
void deleteZeros( int n )
```

die alle Zellen mit dem Eintrag 0 aus der Liste löscht. Dabei dürfen Sie davon ausgehen, dass die Liste mindestens eine Zelle enthält, und dass der Eintrag der ersten Zelle ungleich 0 ist.

Übung 26.7 Erzeugen einer Liste, leicht, typische Klausuraufgabe

Geben Sie Java-Code an, der unter Benutzung der Klassen `List` und `Cell` eine einfach verkettete Liste mit den Elementen 77, 88 und 99 erzeugt!

Übung 26.8 Maximum einer Liste, mittel, typische Klausuraufgabe

Erweitern Sie die Klasse `List` um eine Methode

```
int listMaximum()
```

die die größte Zahl in der Liste ermittelt und ausgibt. Falls `start` leer ist (also gleich `null`), soll der Wert -1 zurückgegeben werden.

Übung 26.9 Summe einer Liste bestimmen, mittel, typische Klausuraufgabe

Erweitern Sie die Klasse `List` um eine Methode

```
int sum()
```

die alle Elemente der Liste addiert und die Summe zurückgibt. Ist die Liste leer, so soll 0 zurückgegeben werden.

Übung 26.10 Elemente hinten löschen, mittel, typische Klausuraufgabe

Erweitern Sie die Klasse `List` um eine Methode

```
void deletePenultimate()
```

die das vorletzte Element aus der Liste löscht. Sie dürfen dabei davon ausgehen, dass die Liste mindestens drei Elemente enthält.

Übung 26.11 Elemente finden, mittel, typische Klausuraufgabe

Erweitern Sie die Klasse `List` um eine Methode

```
void getElement (int i)
```

die das `i`-te Element der Liste ausgibt, falls die Liste mindestens `i` Einträge hat. Ansonsten soll `-1` zurückgegeben werden.

Übung 26.12 Elemente hinten finden, schwer, typische Klausuraufgabe

Erweitern Sie die Klasse `List` um eine Methode

```
void getElementReverse (int i)
```

die das `i`-te Element der Liste *von hinten* ausgibt, falls die Liste mindestens `i` Einträge hat. Ansonsten soll `-1` zurückgegeben werden.

Übung 26.13 Iteration über eine Liste, leicht, original Klausuraufgabe, mit Lösung

Eine Liste ganzer Zahlen sei durch die folgenden beiden Klassen implementiert:

```
class List
{
  Cell start;
  boolean hasEvenLength(){
    // siehe unten
  };
}
class Cell
{
  Cell next;
  int number;
}
```

Ergänzen Sie den Code der Methode `hasEvenLength` der Klasse `List`, die ermitteln soll, ob die Liste eine gerade Länge hat, also z.B. 0, 2 oder 4 Elemente enthält. Falls dem so ist, soll **`true`** zurückgegeben werden, sonst **`false`**.

Übung 26.14 Zugriff auf Listen, mittel, original Klausuraufgabe, mit Lösung

Eine Liste ganzer Zahlen sei durch die folgenden beiden Klassen implementiert:

```
class List
{
  Cell start;
  Cell get( int i ){
    // siehe unten
  };
}
class Cell
{
  Cell next;
  int number;
}
```

Ergänzen Sie den Code der Methode `get`, die das i-te Element der Liste zurückgeben soll. Dabei wird bei 0 angefangen zu zählen; für $i \leq 0$ wird also das Startelement zurückgegeben. Wenn die Liste weniger als $i+1$ Elemente enthält, soll **`null`** zurückgegeben werden.

```
Cell get( int i ){
  /* Fügen Sie hier Ihren Code ein */
}
```

27-1

Kapitel 27
Bäume
Der Stammbaum

27-2

Lernziele dieses Kapitels

1. Das Konzept des Baumes verstehen
2. Bäume in Java implementieren können
3. Baumtraversierung kennen und implementieren können

Inhalte dieses Kapitels

27.1 Motivation 229
27.1.1 Modellierung 229
27.1.2 Bessere Datenstrukturen 230

27.2 Begriffe 230
27.2.1 Der Begriff des Baumes 231
27.2.2 Arten von Bäumen 231

27.3 Bäume in Java 232
27.3.1 Beteiligte Klassen 232
27.3.2 Konstruktion 233
27.3.3 Einfügen und Löschen 234
27.3.4 Traversierung 234

Übungen zu diesem Kapitel 236

Worum es heute geht

Bevor Sie weiterlesen, nehmen Sie sich kurz Zeit, einen Blick aus dem Fester zu wagen. (Sollten Sie sich im Freien befinden, schauen Sie sich einfach etwas um; sollten Sie in einem Raum ohne Fenster sein, so nehmen Sie sich ein Beispiel an Ihren Vorfahren, die ihr Höhlendasein auch vor einigen tausend Jahren erfolgreich hinter sich gelassen haben und mit dieser Entscheidung im Großen und Ganzen immernoch zufrieden sind.) Dort erblicken Sie mit hoher Wahrscheinlichkeit einen Baum. (Wenn nicht, so sollten Sie vielleicht über einen Umzug nachdenken.) Mit Ihrem hoffentlich bereits geschulten Biologenblick werden Sie unschwer die wichtigsten Teile eines Baumes entdecken: Da gibt es zum einen den Stamm, die Verzweigungen und Äste, die Blätter und sicherlich auch eine Wurzel, auch wenn Sie sie nicht sehen können, da sie *unter* der Erde ist.

Die Bestandteile eines Baumes finden sich auch in der Datenstruktur des Baumes wieder. Zunächst gibt es eine *Wurzel*, von der alles ausgeht. Ohne Wurzel kein Baum. Von der Wurzel gehen dann Äste aus, von denen wiederum Äste ausgehen und so weiter, bis die letzten Äste in Blättern enden. Die Informatik-Bäume haben keinen Stamm, man hat ihn wegrationalisiert, weshalb Informatik-Bäume eigentlich etwas bescheidener »Büsche« heißen sollten, aber diese kleine biologische Ungenauigkeit sei verziehen.

Was Biologen den Informatikern hingegen nicht verzeihen, ist der Umstand, dass Informatiker die Wurzel eines Baumes grundsätzlich *oben*, die Blätter hingegen *unten* anordnen. Aus diesem Grund sprechen Informatiker auch gerne statt von der Höhe eines Baumes von dessen *Tiefe*. Bei diesem biologischen Bildungsstand der Informatikerzunft stimmt es durchaus bedenklich, dass die großen Bio-Datenbanken von Informatikern programmiert und am Leben gehalten werden. Leuten, die nach einem Ahornblatt als erstes in einer Tropfsteinhöhle suchen würden, vertrauen wir Genomdaten an! Dies sind übrigens dieselben Leute, die die Software von Flugzeug-Autopiloten programmieren und vorher in ihrem Studium gelernt haben, dass ein *Clean Crash* tolerierbar ist. Guten Flug.

27.1 Motivation

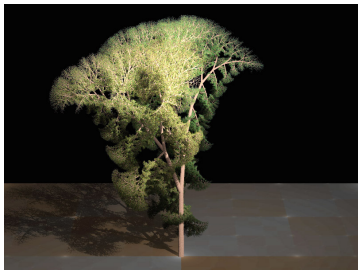
27.1.1 Modellierung

Was ist ein Baum?

27-4

Vertraute Antworten einer Suchmaschine.

Caspar David Friedrich, public domain

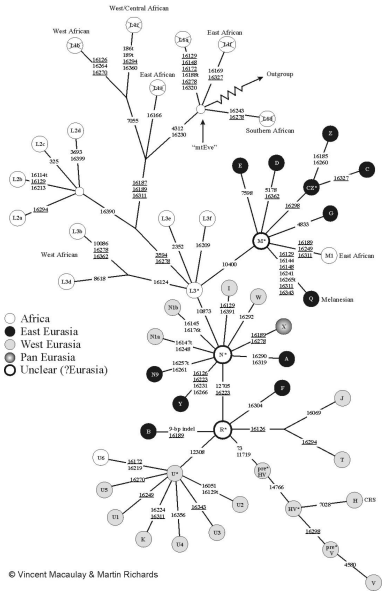


Unknown author, public domain

Was ist ein Baum?

27-5

Einem Biologen vertraute Antwort einer Suchmaschine.



© Vincent Macaulay & Martin Richards

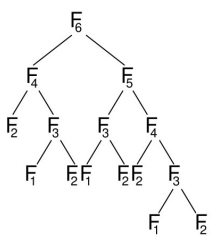
Copyright by Vincent Macaulay and Martin Richards, Creative Commons Attribution Sharealike

Was ist ein Baum?

27-6

Einer Informatikerin vertraute Antworten einer Suchmaschine.

Unknown author, public domain



Unknown author, public domain

27-7

»Bäume« sind hierarchische Strukturen.

Bäume in der Informatik

Der Begriff *Baum* steht in der Informatik allgemein für eine *hierarchische Struktur*.

Beispiele: Dinge, die als Bäume modelliert werden können

- Genetische Stammbäume
- Dateibäume
- Menüs
- Verwaltungsstrukturen in Behörden und Firmen

27-8

Erste Motivation von Bäumen.

Baumartige Strukturen kommen so häufig vor, dass es sich lohnt,

1. ihre Eigenschaften allgemein zu studieren und
2. ihre Implementation zu beherrschen.

27.1.2 Bessere Datenstrukturen

27-9

Zweite Motivation: Ungelöste Probleme

Betrachten Sie eine der folgenden Situationen: Eine Telekom-Firma möchte die Telefonnummern der deutschen Bevölkerung verwalten. Eine Biotech-Firma möchte ihre Produktdatenbank verwalten. Eine Fachschaft möchte ihre Mitgliederliste verwalten.

Probleme

- Sortierte Arrays sind als Datenstrukturen wenig geeignet, da ständig neue Einträge hinzukommen und alte gelöscht werden.
  Einfügen und Löschen dauern bei Arrays zu lange.
- Sortierte Listen sind als Datenstrukturen wenig geeignet, da ständig Einträge gesucht werden müssen.
  Suchen dauert bei Listen zu lange.

27-10

Zweite Motivation von Bäumen.

In speziellen Bäume, genannt *Suchbäume*, kann man

- in Zeit $O(\log n)$ Elemente einfügen,
- in Zeit $O(\log n)$ Daten löschen,
- in Zeit $O(\log n)$ Daten suchen.

Suchbäume sind also ein guter Ausgleich der Eigenschaften von Listen und Arrays.

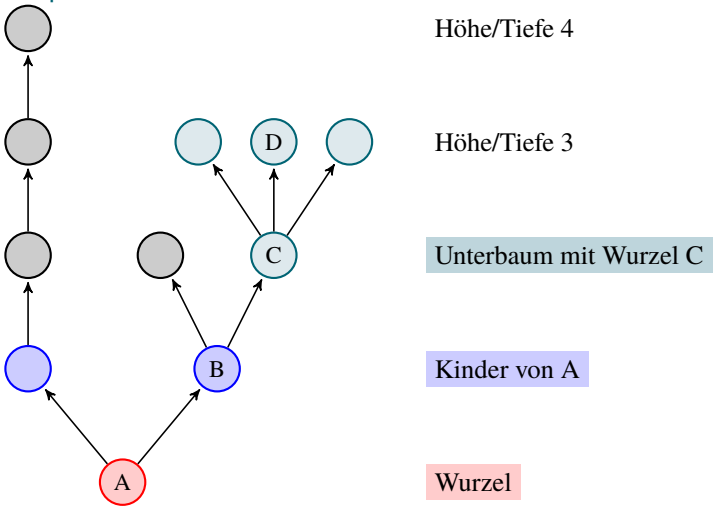
27.2 Begriffe

27.2.1 Der Begriff des Baumes

Woraus bestehen Bäume in der Informatik? 27-11

Informatik-Bäume beginnen an einer *Wurzel*. Die Wurzel hat eine Reihe von *Kindern*. Jedes Kind kann *wieder Kinder* haben. Die Wurzel und alle ihre Kinder und Kindeskinder heißen *Knoten*. Knoten ohne Kinder heißen *Blätter*.

Beispiel eines Baumes. 27-12



Wichtige Begriffe zu Bäumen 27-13

- **Knoten** Die Elemente des Baumes
- **Elternknoten** Knoten, von denen Pfeile zu Kindern ausgehen.
- **Kinder** Knoten, die durch Pfeile mit einem Elternknoten verbunden sind
- **Enkelkinder** Kinder von Kindern
- **Nachfahren** Kinder, Enkelkinder, Urenkelkinder, usw.
- **Vorfahren** Elternknoten, Großelternknoten, usw.
- **Wurzel** Ursprung des Baumes; einziger Knoten ohne Elternknoten
- **Blatt** Knoten ohne Kinder
- **innerer Knoten** Knoten mit Kindern
- **äußerer Knoten** Andere Bezeichnung für Blätter
- **Tiefe** Bei Knoten, die Entfernung zur Wurzel; bei Bäumen, die maximale Tiefe
- **Höhe** Andere Bezeichnung für Tiefe (!)
- **Unterbaum** Baum, der entsteht, wenn man nur einen Knoten und seine Nachfahren betrachtet
- **Teilbaum** Baum, der entsteht, wenn man einige Knoten weglässt

✎ **Zur Übung** 27-14

Schlagen Sie Definitionen der folgenden Begriffe vor: Geschwisterknoten, Schicht, Level, Einzelkind, Onkel, Tante.

27.2.2 Arten von Bäumen

Varianten von Bäumen 27-15

Wo stehen die Daten?

1. In allen Knoten (Normalfall).
2. Nur in den Blättern.
3. Zusätzlich oder ausschließlich an den Kanten.

Wie viele Kinder gibt es?

1. Beliebig viele Kinder an allen Knoten.
2. Genau zwei oder null Kinder (Binärbaum).
3. Höchstens zwei Kindern (auch Binärbaum genannt, grrr).
4. Genau zwei, drei oder null Kinder (2-3-Baum).

27.3 Bäume in Java

27.3.1 Beteiligte Klassen

27-16

Wie implementiert man Bäume in Java?

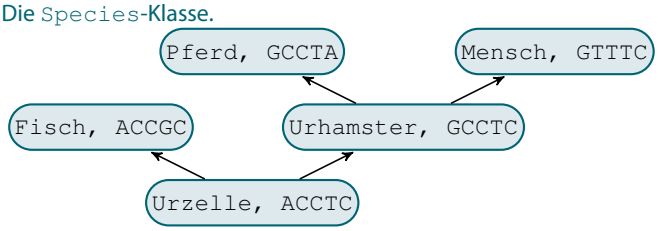
Wie bei Listen benutzt man mehrere Klassen, um Bäume zu implementieren:

1. Eine *Knotenklasse*. Diese entspricht den »Zellen« bei Listen.
2. Eine *eigentliche Baumklasse*, die einen Verweis auf die Wurzel enthält.
3. Optional eine Klasse für die *Werte*, die an den Knoten stehen.
   Beispiel: Bei einem phylogenetischen Baum wäre diese Klasse `Species`

27-17

Implementation phylogenetischer Bäume

Die `Species`-Klasse.




```
class Species {
  String speciesName;
  String genome;

  // Konstruktor
  Species (String name, String genome)
  {
    this.speciesName = name;
    this.genome      = genome;
  }
}
```

27-18

Implementation phylogenetischer Bäume

Die Knoten-Klasse

Implementation mit beliebig vielen Kindern:

```
class Node {
  // Die Spezies, die der Knoten speichert
  Species species;
  // Die ganze Brut von Kindern...
  Node[] children;
}
```

Implementation mit höchstens zwei Kindern:

```
class Node {
  Species species;
  Node leftChild, rightChild;
}
```

Bemerkungen: Zur Vereinfachung betrachten wie im Folgenden nur binäre Bäume. Man kann zusätzliche ein `parent` Attribut einführen. Dies entspricht dem `prev`-Attribut bei doppelt verketteten Listen.

27-19

Implementation phylogenetischer Bäume

Die eigentliche Baumklasse

```
class PhylogeneticTree {

  // Attribut, das auf die Wurzel verweist
  Node root;

  // Default-Konstruktor
```

```
  PhylogeneticTree () {
    this.root = null;
  }

  // Methoden
  boolean isEmpty () {
    return this.root == null;
  }
  ...
}
```

27.3.2 Konstruktion

Konstruktion neuer Bäume

27-20

Ein neu erzeugter Baum ist erstmal leer. Dann kann man nach und nach Elemente einfügen. Alternativ kann man aber auch neue Bäume erzeugen, indem man zwei bestehende zu einem neuen zusammenfasst.

```
class PhylogeneticTree {
  // Konstruktor, der einen Baum aufbaut
  // mit bereits konstruierten linken und rechten
  // Teilbäumen. Die Wurzel wird neu erzeugt.
  PhylogeneticTree (Species s,
                    PhylogeneticTree left,
                    PhylogeneticTree right) {
    this.root = new Node ();
    this.root.species = s;
    if (left != null) {
      this.root.leftChild = left.root;
    }
    if (right != null) {
      this.root.rightChild = right.root;
    }
  }
}
```

27-21

✎ Zur Übung

Visualisieren Sie die Objekte und Verweise, die durch folgenden Code erzeugt werden:

```
Species start  = new Species("first cell", "CCCT");
Species human  = new Species("human",      "ACGT");
Species monkey = new Species("monkey",     "ACCT");
Species fish   =
  new Species("Hommingberger Gepardenforelle", "CTCT");

PhylogeneticTree myTree =
  new PhylogeneticTree (start,
    new PhylogeneticTree (monkey,
      null,
      new PhylogeneticTree (human, null, null)
    ),
    new PhylogeneticTree (fish, null, null));
```

27.3.3 Einfügen und Löschen

27-22

Einfügen und Löschen in Bäumen

Das Einfügen und Löschen einzelner Knoten in der Mitte eines Baumes ist *fummelig*. Einfacher ist es, ganze Teilbäume zu löschen oder einzufügen.

```
class PhylogeneticTree {
  ...

  void addTreeAsLeftChild (Node             where,
                           PhylogeneticTree what) {
    where.leftChild = what.root;
  }

  void removeLeftSubtree (Node where) {
    where.leftChild = null;
  }
}
```

27.3.4 Traversierung

27-23

Traversierung von Bäumen

Traversieren bedeutet, dass man alle Knoten eines Baumes abläuft. An jedem Knoten kann man nun etwas »tun«, beispielsweise, die Spezies ausdrucken oder verändern. Die *Reihenfolge*, in der die Knoten traversiert werden, ist bei Bäumen nicht ganz klar. Es gibt drei wichtige Reihenfolgen

1. In-Order:
   Erst den linken Teilbaum, dann der Knoten, dann der rechte Teilbaum.
2. Pre-Order:
   Erst der Knoten, dann der linke Teilbaum, dann der rechte Teilbaum.
3. Post-Order:
   Erst der linke Teilbaum, dann der rechte Teilbaum, dann der Knoten.

27-24

Beispiel einer Traversierung

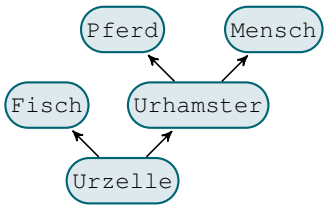
```
class PhylogeneticTree {
  ...
  int countNodesInSubtree (Node node) {
    // Rekursive Post-Order-Traversierung
    if (node == null) {
      return 0;
    }
    else {
      return countNodesInSubtree (node.leftChild) +
```

```
            countNodesInSubtree (node.rightChild) +
            1;
    }
  }

  int countNodeInTree () {
    return countNodeInSubtree (this.root);
  }
}
```

✎ Zur Übung 27-25

Was geben die Methoden bei Eingabe `Urzelle` aus?




```
void printAllSpeciesPreOrder (Node node) {
  if (node != null) {
    System.out.println (node.species.speciesName);
    printAllSpeciesPreOrder (node.leftChild);
    printAllSpeciesPreOrder (node.rightChild);
} }
void printAllSpeciesInOrder (Node node) {
  if (node != null) {
    printAllSpeciesInOrder (node.leftChild);
    System.out.println (node.species.speciesName);
    printAllSpeciesInOrder (node.rightChild);
} }
```

Zusammenfassung dieses Kapitels

▶ Bäume in der Informatik 27-26

Unter *Bäumen* versteht man in der Informatik alle Arten hierarchischer Strukturen. Bäume bilden eine *Datenstruktur*, die ähnlich wie Listen aufgebaut ist:

```
class Tree {
  Node root;
}

class Node {
  SomeType data;

  Node leftChild;
  Node rightChild;
}
```

Einfügen und Löschen in Bäumen ist »fummelig«.

▶ Die drei Arten der Traversierung eines Baumes

```
void doSomethingPreOrder (Node node) {
  if (node != null) {
    do_something_for (node);
    doSomethingPreOrder (node.leftChild);
    doSomethingPreOrder (node.rightChild);
} }

void doSomethingPostOrder (Node node) {
```

```
  if (node != null) {
    doSomethingPostOrder (node.leftChild);
    doSomethingPostOrder (node.rightChild);
    do_something_for (node);
} }

void doSomethingInOrder (Node node) {
  if (node != null) {
    doSomethingInOrder (node.leftChild);
    do_something_for(node);
    doSomethingInOrder (node.rightChild);
} }
```

Übungen zu diesem Kapitel

Übung 27.1 Baum aufbauen, einfach

Ein Binärbaum von Integer-Werten sei durch folgende Java-Klassen realisiert:

```
class Node
{
   int number;
   Node left;
   Node right;
}

class Tree
{
   Node root;
}
```

Mit welchen Java-Befehlen können Sie einen Baum aufbauen, der als Wurzel die Zahl 2 hat, das linke Kind 1 und das rechte Kind 3 hat?

Übung 27.2 Knotensumme bestimmen, schwer, mit Lösung

Schreiben Sie eine Methode `sum`, die die Summe aller Zahlen in einem Baum aus Übung 27.1 bestimmt. Dies lässt sich am besten rekursiv lösen. Dazu erweitert man die Klasse `Tree` um die Methoden

```
class Tree {

  public int sum() {
    return sumOfChildrenFrom (this.root);
  }

  private int sumOfChildrenFrom(Node n) {
    // ?
  }
}
```

Zur Implemention von `numberOfChildernFrom` implementieren Sie folgende Idee: Die Anzahl der Knoten eines nichtleeren Teilbaum ist gleich 1 plus der Anzahl der Knoten im linken plus im echten Unterbaums.

Übung 27.3 Höhe bestimmen, schwer

Schreiben Sie eine Methode, die die Höhe eines Baumes bestimmt. Gehen Sie dazu analog zu Übung **??** vor.

Übung 27.4 Blattzahl bestimmen, schwer

Schreiben Sie eine Methode, die die Anzahl der Blätter eines Baumes bestimmt. Gehen Sie dazu analog zu Übung **??** vor.

Prüfungsaufgaben zu diesem Kapitel

Übung 27.5 Algorithmus auf Bäumen, schwer, original Klausuraufgabe, mit Lösung

Die Knoten eines Binärbaums werden durch folgende Klasse realisiert.

```
class Node {
  int value;
  Node left;
  Node right;
}
```

Schreiben Sie eine Methode `static int maxSumOnPath(Node root)`, die die größte Summe von Werten auf einem Pfad von der Wurzel `root` zu einem Blatt zurückgibt.

28-1

Kapitel 28
Suchbäume

Suchet und ihr werdet finden

28-2

Lernziele dieses Kapitels

1. Konzept des Suchbaums verstehen
2. Basisoperationen auf Suchbäumen verstehen
3. Vor- und Nachteile von Arrays, Listen und Suchbäumen beurteilen können

Inhalte dieses Kapitels

28.1 Einführung 239
28.1.1 Problemstellung 239
28.1.2 Idee des Suchbaumes 240
28.1.3 Suchbäume in Java 240

28.2 Operationen 240
28.2.1 Suchen 240
28.2.2 Einfügen 242
28.2.3 Löschen aus Suchbäumen 243

28.3 Vergleich der Implementationen 245

Übungen zu diesem Kapitel 246

Worum es heute geht

Es war eine Hochzeit *made in heaven*. Der Bräutigam: der sortierte Array. Die Braut: die Liste. Die Presse war voll davon, in den verschiedensten Fachzeitschriften wurden dem interessierten Fachpublikum noch einmal die vielen Vorzüge des Paares präsentiert. Da wurde die rasend schnelle Suche beim Bräutigam lobend erwähnt, in Zeit $O(\log n)$ konnte er Gesuchtes wiederfinden. Bei der Braut wurde ihre Gabe hervorgehoben, neue Elemente an beliebiger Stelle sogar in Zeit $O(1)$ einzufügen. Man sprach davon, dass sich die beiden doch perfekt ergänzen würden, die Nachteile des einen würden ja durch die Vorteile des anderen aufgehoben. Vieldeutig wünschte man dem Paar viel Spaß miteinander. Der Spaß war erfolgreich und der sortierter Array und die Liste zeugten einen prächtigen Sohn: den Suchbaum, der von seinen Eltern die Vorteile erbt.

Suchbäume sind, wie der Name schon sagt, Bäume, also weder Listen noch sortierte Arrays. Aber sie haben in der Tat die wichtigen Eigenschaften ihrer Eltern: Wie in einem sortierten Array sucht man in einem Suchbaum mit einer Art binären Suche; gleichzeitig kann man aber in einen Suchbaum neue Elemente sehr schnell einfügen, fast so schnell wie in eine Liste.

Die einfachen Suchbäume, die in diesem Kapitel eingeführt werden, sind allerdings ein wenig Muttersöhnchen. Sie tendieren nämlich mit Vorliebe dazu, sich sehr eng an ihre Mutter, die Liste, zu halten. In der Tat kann es unter bestimmten (in der Praxis leider recht typischen) Bedingungen passieren, dass ein Suchbaum sich genau wie eine Liste verhält. Ausgefeiltere Suchbäume, wie die in dieser Vorlesung nicht behandelten 2-3-Bäume, kommen da mehr nach dem Vater und können ähnlich einem sortierten Array eine gewisse Ausgeglichenheit bei der Suche immer garantieren.

28.1 Einführung

28.1.1 Problemstellung

Die Mitgliederliste der Fachschaft.

28-4

Problemstellung

Eine Fachschaft möchte ihre Mitglieder verwalten. Zu jedem Mitglied werden Daten gespeichert; beispielsweise soll für jedes Mitglied die E-Mail-Adresse gespeichert werden. Man möchte nun (schnell) nach der E-Mail-Adresse einer Person suchen können, sowie neue Personen anlegen und Personen löschen können.

Die Klasse für die Mitglieder lautet:

```
class Studie {
  String name;
  String email;
}
```

Was unsere Datenstruktur können soll.

28-5

Wir suchen eine möglichst gute Implementationen einer Klasse mit folgenden Methoden:

```
class Map {

  Studie search (String name);
  // Sucht nach einem Studenten-Objekt
  // anhand eines Namens. Wenn es ein solches nicht
  // gibt, soll null zurückgegeben werden.

  void add (Studie s);
  // Fügt einen Studenten in die Datenstruktur ein.

  void remove (String name);
  // Löscht einen Studenten anhand seines Namens.
}
```

Arrays und Listen als Implementationen haben Nachteile

28-6

- Wir können einen *sortierten Array* benutzen:
  - Suchen dauert $O(\log n)$.
  - *Löschen und Einfügen dauern* $O(n)$.
- Wir können eine *unsortierte Liste* benutzen:
  - Löschen und Einfügen dauern $O(1)$.
  - *Suchen dauert* $O(n)$.

Keine der Implementationen scheint besonders geeignet. Da aber das Suchen viel öfter vorkommt als Einfügen und Löschen, würden wir wohl einen Array benutzen.

Auf dem Weg zum Suchbaum.

28-7

Was macht Suchen in sortierten Arrays schnell?

Suchen ist schnell, da wir bei der binären Suche in jedem Schritt den Suchraum halbieren können.

Was macht Einfügen und Löschen langsam?

Arrays sind zu starr. Erst durch Verweise und Zellen wie bei Listen wird man flexibler.

Bei einem *Suchbaum* versucht man, die Vorteile von sortierten Listen und von Arrays zu vereinen.

28.1.2 Idee des Suchbaumes

28-8

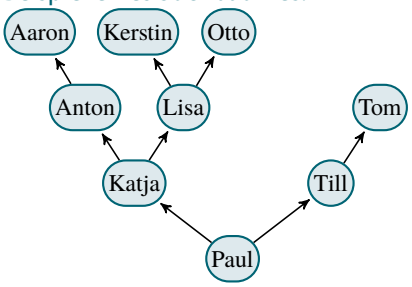
Die Idee des Suchbaumes

Bei einem Suchbaum nimmt man (wie bei einem sortierten Array) an, dass sich je zwei Studenten *vergleichen* lassen. Die Studenten werden aber so eingefügt, dass für jeden Knoten Folgendes gilt:

1. Alle Studenten im *linken Unterbaum* sind *kleiner* als der Student des Knotens.
2. Alle Studenten im *rechten Unterbaum* sind *größer oder gleich* dem Studenten des Knotens.

28-9

Beispiel eines Suchbaumes.



28.1.3 Suchbäume in Java

28-10

Suchbäume implementiert man genau wie normale Bäume.

```
class Node {
  // Verweis auf den zum Knoten gehörende Studenten
  Studie studie;

  // Verweise auf die Wurzeln der Kinder
  Node smallerChildren;
  Node largerChildren;

  // Einfacher Konstruktor (Kinder sind automatisch null):
  Node (Studie s) { this.studie = s; }
}

class StudieTree {
  // Die Wurzel ist das einzige Attribut
  Node root;

  // Die drei Operationen
  Studie search (String name) {...}
  void   add    (Studie s)    {...}
  void   remove (String name) {...}
}
```

28.2 Operationen

28.2.1 Suchen

28-11

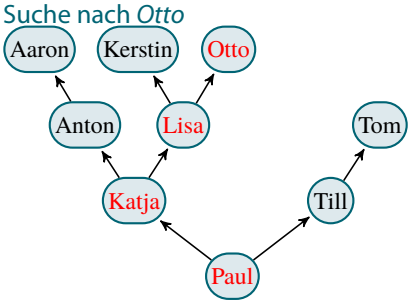
Wie sucht man in Suchbäumen?

Wir gehen ähnlich einer binären Suche vor:

1. Wir vergleichen den zu suchenden Namen mit dem Namen am aktuellen Knoten.
2. Sind sie *gleich*, so sind wir fertig.
3. Ist der zu suchende Namen *kleiner*, so suchen wir im *linken* Teilbaum weiter.
4. Ist der zu suchende Namen *größer*, so suchen wir im *rechten* Teilbaum weiter.

Suche nach *Otto*

28-12



Java-Code der Such-Methode

28-13

```
class StudieTree {
  Studie search (String find_me) {
    // Aufruf der eigentlichen rekrusiven Suche:
    return recursiveSearch (find_me, this.root);
  }

  Studie recursiveSearch (String find_me, Node node) {
    if (node == null) { // Baum ist leer, nichts gefunden
      return null;
    }
    else if (node.studie.name.equals(find_me)) {
      // Bingo!
      return node.studie;
    }
    else if (node.studie.name.compare(find_me) < 0) {
      // Mache mit den kleineren Kindern weiter
      return
        recursiveSearch (find_me, node.smallerChildren);
    }
    else { // Mache mit den größeren Kindern weiter
      return
        recursiveSearch (find_me, node.largerChildren);
    }
  }
}
```

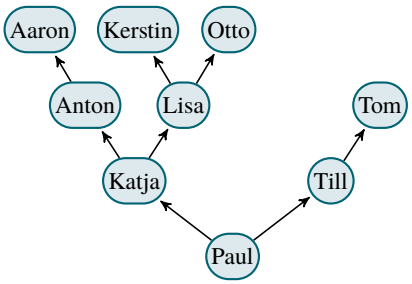
28.2.2 Einfügen

28-14

Einfügen von Knoten

✎ Zur Diskussion

Wohin sollten wir *Peter* und wohin sollten wir dann *Petra* einfügen?



28-15

Wie fügt man in einen Suchbaum ein?

Man fügt neue Knoten immer als Blätter ein (und nicht irgendwie in der Mitte). Dabei gibt es immer genau einen Ort, wo man den Knoten als Blatt einfügen kann, den man wie beim Such-Algorithmus findet:

- Ist der einzufügende Name *kleiner* als der des aktuellen Knoten, mache *links* weiter.
- Ist der einzufügende Name *größer* als der des aktuellen Knoten, mache *rechts* weiter.

28-16

Wir bauen einen Suchbaum mit den Namen der Studenten in den ersten zwei Reihen auf.

28-17

Java-Code zum Einfügen in einen Suchbaum.

`Add`-Methode der Verwaltungsklasse.

```
class StudieTree {
  ...
  void add (Studie s)
  {
    if (this.root == null) {
      this.root = new Node (s);
    }
    else {
      recursiveAdd (s, this.root);
    }
  }
```

```
  void recursiveAdd (Studie s, Node node)
  {
    if (node.studie.name.compareTo(s.name) < 0)
    {
      // s muss nach links
      if (node.smallerChildren == null) {
        node.smallerChildren = new Node(s);
      }
      else {
        recursiveAdd(s, node.smallerChildren);
      }
    }
    else
    {
      // s muss nach rechts
      if (node.largerChildren == null) {
        node.largerChildren = new Node(s);
      }
      else {
        recursiveAdd(s, node.largerChildren);
```

```
            }
        }
    }
}
```

28.2.3 Löschen aus Suchbäumen

Löschen von Knoten

28-18

✎ Zur Diskussion

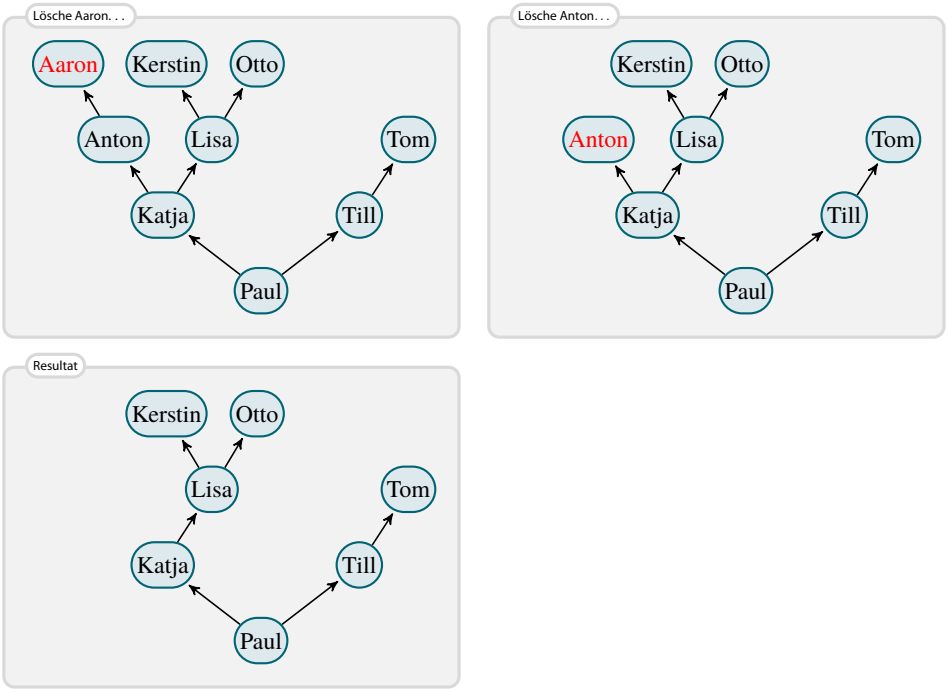
Es ist leicht, ein Element zu löschen, das ein Blatt ist. Viel schwieriger ist es aber, einen inneren Knoten zu löschen. Was kann man tun?

Löschen eines Knotens

28-19

Fall 1: Blätter

Ein *Knoten ohne Kinder* (ein Blatt) kann einfach gelöscht werden.

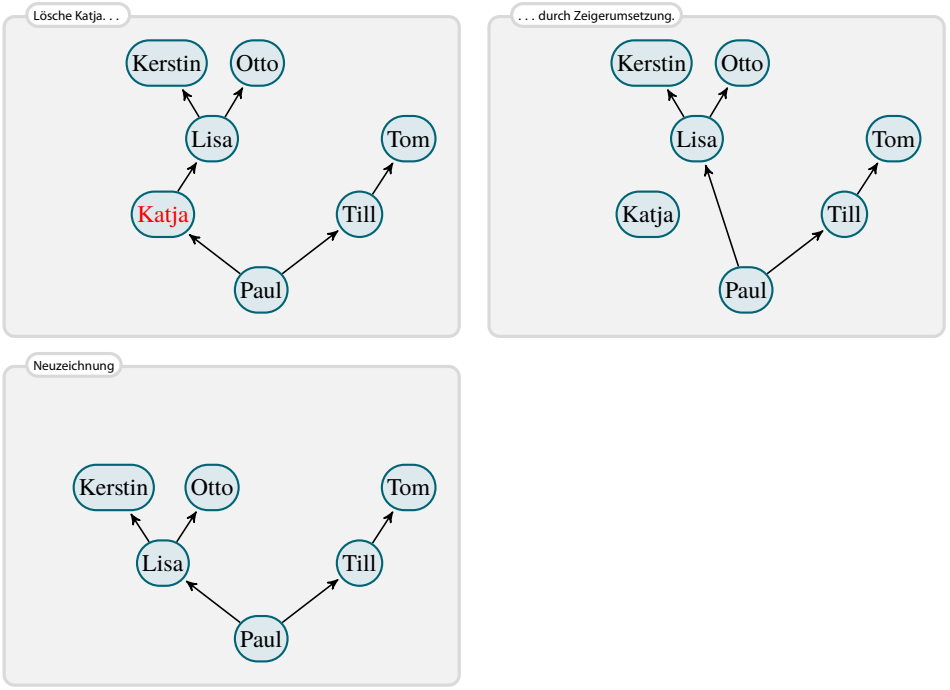


28-20

Löschen eines Knotens

Fall 2: Knoten mit nur einem Kind

Bei einem *Knoten mit einem Kind* nimmt das Kind den Platz des Knotens ein.



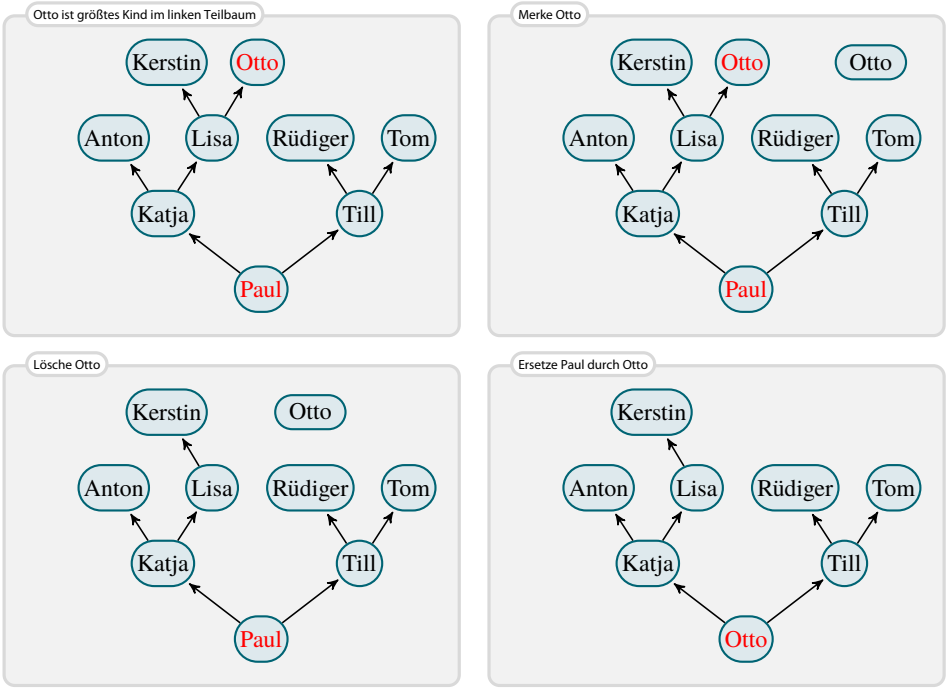
28-21

Löschen von *Paul*

Fall 3: Knoten mit zwei Kindern

Löschen eines *Knoten mit zwei Kindern*:

1. Finde den größten Knoten im linken Teilbaum.
2. Merke den Inhalt.
3. Lösche den gefunden Knoten.
4. Ersetze Inhalt des zu löschenden Knotens durch gemerkten.



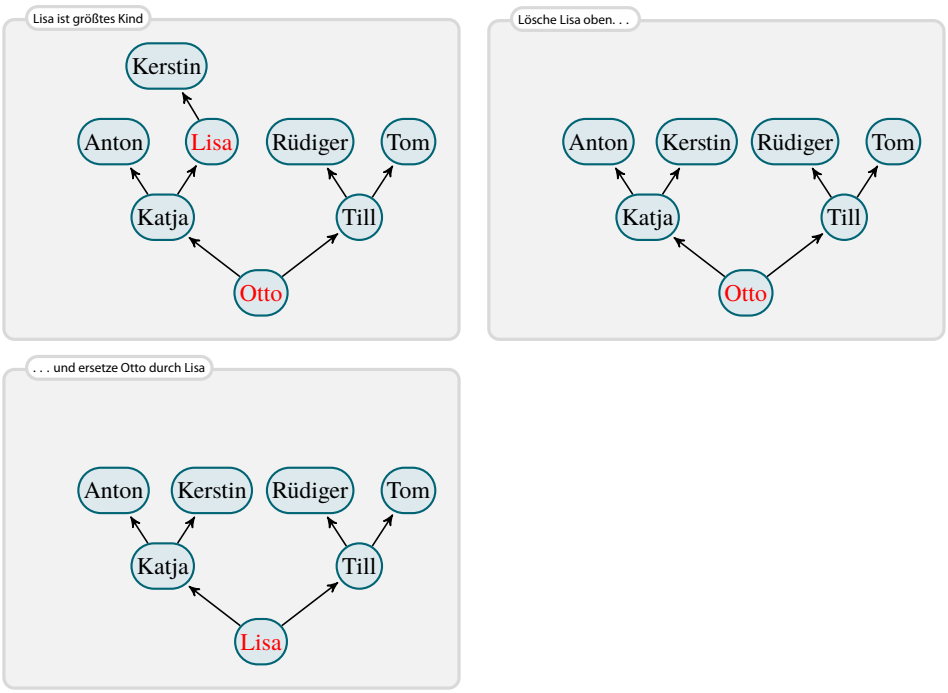
Löschen von *Otto*

28-22

Fall 3: Knoten mit zwei Kindern

Löschen eines *Knoten mit zwei Kindern*:

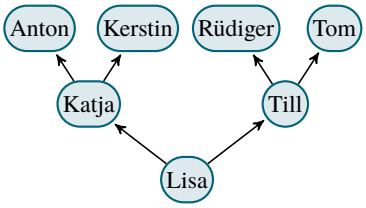
1. Finde den größten Knoten im linken Teilbaum.
2. Merke den Inhalt.
3. Lösche den gefunden Knoten.
4. Ersetze Inhalt des zu löschenden Knotens durch gemerkten.



✎ Zur Übung

28-23

Wie sieht der Suchbaum aus, wenn nacheinander erst *Lisa*, dann *Till* und dann *Kerstin* gelöscht werden?



28.3 Vergleich der Implementationen

Vergleich von Listen, Arrays und Suchbäumen

28-24

Zeitverbrauch bei n Datenelementen und Baumhöhe h.

Implementation	Suchen	Einfügen	Löschen
sortierter Array	$O(\log n)$	$O(n)$	$O(n)$
unsortierte Liste	$O(n)$	$O(1)$	$O(n)$
Suchbaum	$O(h)$	$O(h)$	$O(h)$

- Offensichtlich ist *die Höhe h* wichtig – je kleiner desto besser.
- Ist der Baum *ausgeglichen*, so ist $h = \log_2 n$.
- Ist der Baum *zufällig*, so ist ebenfalls $h = O(\log n)$.
- Werden die Einträge aber *in sortierter Reihenfolge eingefügt* ist sie aber $O(n)$; hier helfen *fortgeschrittene Suchbäume*.

Zusammenfassung dieses Kapitels

28-25

▶ Zentrale Eigenschaften von Suchbäumen
Ein Suchbaum ist ein binärer Baum, in dem für jeden Knoten gilt:

1. Alle Knoten im *linken Unterbaum* sind *kleiner* als der Knoten.
2. Alle Knoten im *rechten Unterbaum* sind *größer oder gleich* dem Knoten.

Die drei zentralen Operationen Suchen, Einfügen und Löschen benötigen Zeit $O(h)$, wobei h die Baumhöhe ist. Sie ist bei zufälligen Bäumen $O(\log n)$, im schlimmsten Fall aber auch $O(n)$.

▶ Suchen in Suchbäumen
Beginnend bei der Wurzel, gehe zum linken Kind, wenn das gesuchte Element kleiner als die Wurzel ist, sonst zum rechten Kind.

▶ Einfügen in Suchbäumen
Suche die Stelle, wo das Element sein müsste, und füge es dort als neues Blatt ein.

▶ Löschen in Suchbäumen

1. Blätter können einfach gelöscht werden.
2. Knoten mit nur einem Kind können durch dieses Kind ersetzt werden.
3. Bei Knoten mit zwei Kindern, finde den größten Knoten im linken Teilbaum, lösche ihn dort und ersetze mit seinem Wert den eigentlich zu löschenden Knoten.

Übungen zu diesem Kapitel

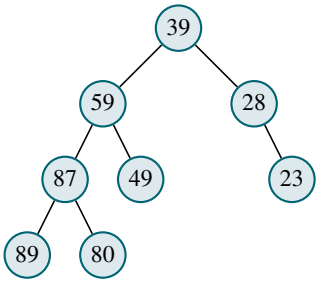
Übung 28.1 Operationen auf Suchbäumen, mittel

Schreiben Sie Methoden `int minElement()` und `int maxElement()` zum Auffinden der kleinsten und größten Elemente eines Suchbaums.

Prüfungsaufgaben zu diesem Kapitel

Übung 28.2 Löschen aus Suchbäumen, einfach, original Klausuraufgabe, mit Lösung

Löschen Sie nacheinander die Werte 59, 28 und 39 aus folgendem binären Suchbaum und zeichnen Sie den Baum nach jedem Löschvorgang.



Übung 28.3 Operationen auf Suchbäumen, einfach, typische Klausuraufgabe

Zeichnen Sie den binären Suchbaum, der entsteht, wenn in einen leeren binären Suchbaum nacheinander folgende Werte eingefügt werden:

$$47,\ 14,\ 46,\ 84,\ 52,\ 88,\ 60,\ 30,\ 87,\ 15$$

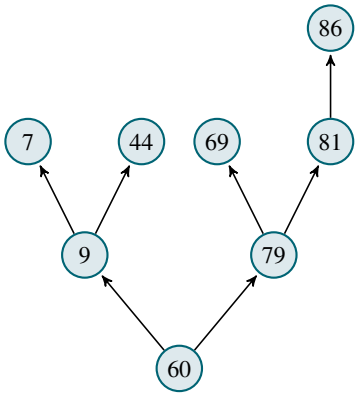
Löschen Sie aus dem resultierenden Baum nacheinander die Werte 60, 88 und 47 und zeichnen Sie die daraus entstehenden Bäume.

Übung 28.4 Einfügen und Löschen aus Suchbaum, original Klausuraufgabe, mit Lösung

1. Fügen Sie nacheinander die Zahlen $-3, 2, 9, 8, 1, 7$ und 0 in einen Suchbaum ein und zeichnen Sie das Ergebnis.
2. Löschen Sie aus diesem Baum die Zahl 9 und zeichnen Sie den entstandenen Baum.
3. Löschen Sie nun auch die Zahl -3 und zeichnen Sie das Ergebnis.

Übung 28.5 Eigenschaften von Bäumen und Löschen von Knoten, original Klausuraufgabe

Betrachten Sie folgenden Baum.



1. Welcher Knoten ist die *Wurzel* des Baums? Geben die Nummer des Knotens an.
2. Welche Knoten sind die *Blätter* des Baums? Geben Sie die Nummern der Knotens an.
3. Wie groß ist die *Tiefe* des Baums?
4. Löschen Sie nacheinander die Zahlen 86, 79, und 60 aus dem Baum und zeichnen Sie den Baum nach jedem Löschschritt.

29-1

Kapitel 29
Hashing
Wirklich schnelles Suchen

29-2

Lernziele dieses Kapitels

1. Konzept des Hashwerts verstehen
2. Einfache Hashfunktionen kennen und implementieren können
3. Einfache Hashverfahren kennen und implementieren können

Inhalte dieses Kapitels

29.1	Einführung	249
29.1.1	Motivation	249
29.1.2	Die Idee	249
29.1.3	Die Problemstellung	250
29.2	Hashwerte	250
29.2.1	Was ist eine gute Hashfunktion?	250
29.2.2	Standard-Hashfunktionen	251
29.3	Hashverfahren	252
29.3.1	Idee	252
29.3.2	Implementation	253
29.3.3	Suche	253
29.3.4	Einfügen und Löschen	253

Worum es heute geht

Was ist eigentlich ein »Hash«? Wie es sich für ein kurzes englisches Wort gehört, hat es ziemlich viele Bedeutungen, denn generell kann man bei einem guten englischen Wörterbuch eine fast beliebige Kombination von vier Buchstaben suchen und mit ziemlicher Sicherheit kommen eine große Anzahl von Bedeutungen heraus – von denen außer den Wörterbuchschreibern noch niemand gewusst hat. Beispielsweise bezeichnet das doch eher selten gebrauchte Substantiv »balk« solche unterschiedliche Dinge wie »an illegal motion made by a baseball pitcher that may decieve a base runner«, »a roughly squared timber beam«, »any area on a pool or billiard table in which play is restricted in some way« und bekanntermaßen natürlich auch »a ridge left unplowed between furrows«.

Ein »Hash« ist nun laut Webster-Wörterbuch »a dish of cooked meat cut into small pieces and recooked, usually with potatoes«. Es gibt Hashes in verschiedenen Varianten, die auch liebevoll aufgezählt werden; die offenbar kulinarisch belesenen Autoren des Wörterbuches haben so auch an Vegetarier im Rahmen eines »hash of raw tomatoes, chilies, and coriander« gedacht. Entsprechend bedeutet dann »to hash«, »make meat or other food into a hash«, man kann »hashing« also grob als »verhackstückseln« übersetzen. (Anderseits bedeutet »to make a hash of something« so viel wie wie »es verpatzen«.)

Wie verhackstückselt man nun aber Daten? Dies erscheint doch als eher gefährlicher, vielleicht sogar verbotener Vorgang. Tatsächlich wird beim Hashing ein Objekt genommen und daraus durch wildes »Herumrechnen« eine einzelne Zahl errechnet – der so genannte *Hashwert* des Objekts. Ähnlich wie man beim gekochten Hash auch nicht mehr erkennen kann, von welchem Rindvieh es stammt, so kann man einem Hashwert auch nicht mehr ansehen, von welchem Objekt es stammt. Wichtig ist beim Hashing eigentlich nur, dass unterschiedliche Objekte (möglichst) unterschiedliche Hashwerte bekommen sollten.

Nehmen wir nun an, wir wollen – wie wir es ja schon in den vorherigen Kapiteln öfters getan haben – Objekte verwalten mittels der drei Grundoperationen Einfügen, Löschen und Suchen. Zu Objekten können wir durch Verhackstückselung Zahlen errechnen, sagen wir

zwischen 0 und 999. Was nützt uns dies? Die geniale Idee ist, einen Array der Größe 1000 zu benutzen, *Hash-Tabelle* genannt, in dem an Stelle i das Objekt zu finden ist, welches den Hashwert i hat. Will man also ein Objekt einfügen, so errechnet man seinen Hashwert und schreibt es an die entsprechende Stelle in der Hash-Tabelle. Will man ein Objekt suchen, so errechnet man seinen Hashwert und schaut an der entsprechenden Stelle nach, ob es denn vorhanden ist. So weit die Grundidee, Sie sehen vielleicht schon Probleme, die sich dabei ergeben. Wie man diese löst, soll in diesem Kapitel erklärt werden.

Das Wort »Hash« ist übrigens im Vergleich zu den Alternative, die der Thesaurus bereithält, eher langweilig. Es würde die Informatikliteratur sicherlich etwas auflockern, redete man statt von Hashwerten von *mishmash values*, die man statt in eine Hash-Tabelle in eine *gallimaufry table* einfügt.

29.1 Einführung

29.1.1 Motivation

Schon nicht schlecht.

29-4

Es gibt viele Problemstellungen, bei denen wir Objekte verwalten müssen.

- Verwaltung von Fachschaftslisten
- Verwaltung von Dateisystemen
- Verwaltung von Molekül- und Gendaten

Dazu kann man *Arrays*, *Listen* oder *Suchbäume* verwenden.
Der Zeitverbrauch war (mit h irgendwo zwischen $\log n$ und n):

Implementation	Suchen	Einfügen	Löschen
sortierter Array	$O(\log n)$	$O(n)$	$O(n)$
unsortierte Liste	$O(n)$	$O(1)$	$O(n)$
Suchbaum	$O(h)$	$O(h)$	$O(h)$

Das Versprechen der Hashtabellen

29-5

In *Hashtabellen* kann man

- in Zeit $O(1)$ etwas einfügen,
- in Zeit $O(1)$ etwas löschen,
- in Zeit $O(1)$ etwas suchen.

Man sollte allerdings das Kleingedruckte lesen.

29.1.2 Die Idee

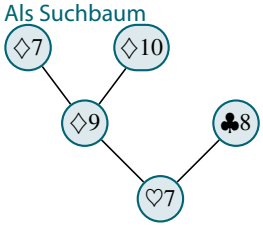
Die Verwaltung von Spielkarten.

29-6

Problemstellung

Wir wollen ein Skatprogramm schreiben. Dazu wird eine Datenstruktur benötigt, die die Hand eines Spielers speichert.

Als Suchbaum



Als sortierter Array

♢7
♢9
♢10
♡7
♣8

29-7

Die erste Idee beim Hashing.

- Es gibt nur 32 Spielkarten.
- Wir könnten also einfach einen Array der Größe 32 anlegen.
- An Stelle 0 kommt dann (falls vorhanden) das Objekt für die ♢7.
- An Stelle 1 kommt dann (falls vorhanden) das Objekt für die ♢8.
- Und so weiter.

29-8

Die zweite Idee beim Hashing.

- In einer Hand kommen nur 10 Karten vor.
- Wir reservieren deshalb nur 10 Positionen im Array.
- Dann benutzen wir dieselbe Position für verschiedene Karten.
  Beispiel: Alle Siebenerkarten an Stelle 0, alle Achterkarten an Stelle 1, usw.
- Wenn dann zwei Karten an dieselbe Stelle sollen, so tut man etwas Schlaues.

29.1.3 Die Problemstellung

29-9

Die allgemeine Problemstellung beim Hashing.

- Es ist eine Datenstruktur zur Verwaltung von Objekten gesucht.
- Die Objekte entstammen einem bekannten *Universum*.
- Die Objekte werden in einem *Array fester Größe* gespeichert.
- Jedem Objekt wird durch eine *Hashfunktion* eine Position in diesem Array zugeordnet.
- Hashverfahren unterscheiden sich danach
  - wie Hashfunktion berechnet wird und
  - was passiert, wenn zwei Objekten dieselbe Stelle im Array zugeordnet wird (Kollisionsauflösung).

29.2 Hashwerte

29.2.1 Was ist eine gute Hashfunktion?

29-10

Wie können wir einen geeignete Hashwert bestimmen?

- Bei Karten ist es einfach, Kartenwerte auf Arraypositionen abzubilden.
- Bei Strings ist dies schon wesentlich schwieriger.
- Ebenfalls schwierig erscheint dies bei einem Objekt vom Typ `Atom`, bestehend aus einer Ordnungszahl und 3D-Koordinaten.

Wünschenswerte Eigenschaften von Hashfunktionen

1. Die Hashfunktion sollte *einfach und schnell* zu berechnen sein.
2. Sie sollte die Werte möglichst *gleichmäßig* auf die möglichen Hashwerte verteilen.
3. Sie sollte die Werte möglichst *zufällig* auf die möglichen Hashwerte verteilen.

29-11

✎ Zur Übung

Eine Hashtabelle habe die Größe 100.
Entwerfen Sie Hashfunktionen für folgende Arten von Objekten:

1. `double` Zahlen
2. Strings

29.2.2 Standard-Hashfunktionen

Standardmethodik bei Hashfunktionen

29-12

Hashing geschieht oft in zwei Schritten:

1. Eine Funktion wandelt Objekte in (eventuell sehr große) natürliche Zahlen um.
2. Eine zweite Funktion bildet beliebige Zahlen auf Werte im Intervall $[0, T-1]$ ab, wobei T die Größe der Hashtabelle ist.

Beispiel

Ein String bestehe aus den Zeichen $a_0 a_1 a_2$. Die ASCII-Werte der Zeichen seien 43, 68 und 100. Dann kann der String auf die Zahl

$$43 + 256 \cdot 68 + 256 \cdot 256 \cdot 100 = 6571051$$

abgebildet werden.

Von der Zahl zum Tabelleneintrag

29-13

Wie sollte man nun eine Zahl n auf das Intervall $[0, T-1]$ abbilden?

Die Modulo-Methode

Man bildet n auf $n \bmod T$ ab.

Die Multiplikations-Methode

Man wählt eine »krumme« Zahl φ und bildet n auf $\lfloor T \cdot (\varphi n - \lfloor \varphi n \rfloor) \rfloor$ ab.

Probleme bei Hashfunktionen

29-14

✎ Zur Diskussion

Welche Probleme können bei der Modulo-Methode und welche bei der Multiplikations-Methode entstehen?
Zur Erinnerung:

Wünschenswerte Eigenschaften von Hashfunktionen

1. Die Hashfunktion sollte *einfach und schnell* zu berechnen sein.
2. Sie sollte die Werte möglichst *gleichmäßig* auf die möglichen Hashwerte verteilen.
3. Sie sollte die Werte möglichst *zufällig* auf die möglichen Hashwerte verteilen.

Erfahrungen aus der Praxis.

29-15

- Bei der Modulo-Methode sollte T eine Primzahl sein und keinesfalls eine Zweier- oder Zehnerpotenz.
- Bei der Multiplikations-Methode sollte φ gleich dem goldenen Schnitt sein.
- Die Berechnung eines guten Hashwertes ist eine schwarze Kunst und wird in der Regel falsch gemacht.

29.3 Hashverfahren

29.3.1 Idee

29-16

Zurück zum Ausgangsproblem

- Wir haben nun eine Hashtabelle (also einen Array) mit T Stellen angelegt.
- Unsere Objekte werden mittels einer Hashfunktion h auf Zahlen zwischen 0 und $T-1$ gleichmäßig abgebildet.
- Wir wollen also Objekt x an Stelle $h(x)$ speichern.
- Was aber sollen wir tun, wenn zwei unterschiedliche Objekte x und y an derselben Stelle $h(x) = h(y)$ gespeichert werden sollen?

29-17

Erstes Verfahren zur Kollisionsauflösung: Verkettete Listen

- In den Arrayfeldern speichern wir *nicht direkt die Werte*.
- Vielmehr ist jedes Arrayfeld der Anfang einer *verketteten Liste*.
- Alle Objekte, die von der Hashfunktion auf dasselbe Feld abgebildet werden, werden in der Liste des Feldes gespeichert.
- *Suchen in dieser Liste* ist zwar *langsam*, aber da Kollisionen selten sind, ist die Liste sehr *kurz*.

29-18

Beispiel einer Hashtabelle mit verketteten Listen

Eckdaten der verketteten Hashtabelle

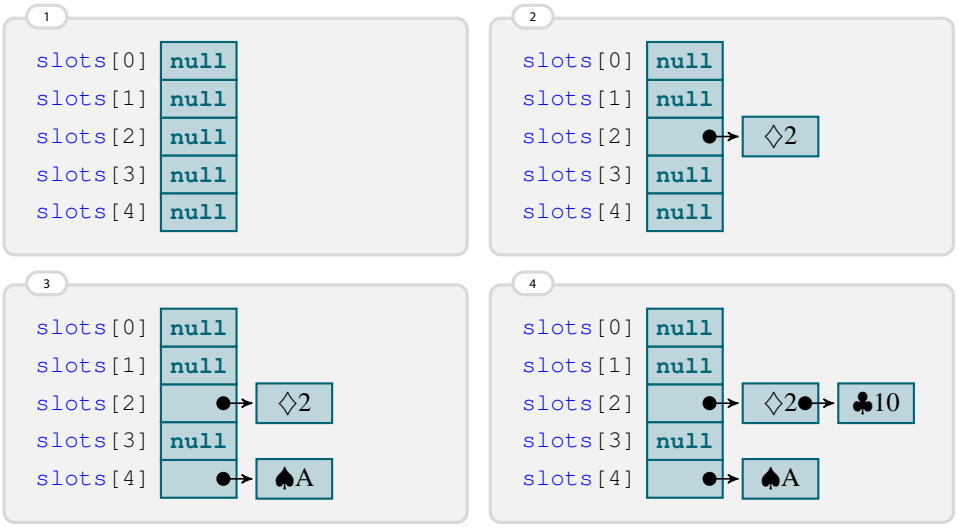
Tabellengröße $T = 5$

Objekte Spielkarten

Karte als Zahl Kartenwert (Joker $= 0$) plus Farbe mal 14.

Hashfunktion Karte als Zahl modulo T.

Einfügen von ♢2 *($h = 2$)*, ♠A *($h = 4$)*, ♣10 *($h = 2$)*.



29.3.2 Implementation

Java-Code einer Hashtabelle für die Fachschaft 29-19

```
class Cell {
  Studie studie;
  Cell   next;

  Cell (Studie s) { this.studie = s; }
}

class HashTable {
  private int    T;
  private Cell[] slots;

  HashTable (int size) {
    this.T = size;
    this.slots = new Cell[this.T];
  }

  Studie search (String name) {...}
  void   add    (Studie s)    {...}
  void   remove (String name) {...}
}
```

29.3.3 Suche

Java-Code der `search`-Methode 29-20

```
class HashTable {
  ...

  Studie search (String name)
  {
    Studie return_me = null;
    Cell cursor      = this.slots[name.hashValue ()
                                  % this.T];
    while (cursor != null)
    {
      if (cursor.studie.name.equals(name)) {
        return_me = cursor;
      }
      cursor = cursor.next;
    }
    return return_me;
  }
}
```

29.3.4 Einfügen und Löschen

✎ Zur Übung 29-21

1. Entwerfen Sie den Code der Methode `void add (Studie s)`.
2. Wie würde das Löschen funktionieren?

Zusammenfassung dieses Kapitels

29-22

▶ Hash-Funktion

Eine *Hash-Funktion* h bildet beliebige Objekte auf Zahlen in einem Intervall $[0, T-1]$ ab, wobei T die Größe der Hashtabelle ist.
Eine gute Hash-Funktionen ist, eine »Binärdarstellung« des Objektes zu betrachten, diese als Zahl aufzufassen und dann modulo einer Primzahl T zu rechnen.

▶ Hash-Tabelle

Eine Hash-Tabelle ist ein Array der Größe T. In ihm wird ein Objekt x an der Stelle $h(x)$ gespeichert. Kommt es zu einer *Kollision*, so kann man die kollidierenden Elemente in einer verketteten Liste speichern.
»Wenn alles gut geht«, so kann man in einer Hash-Tabelle in Zeit $O(1)$ sowohl Suchen, Einfügen wie Löschen.

Zum Weiterlesen

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms,* The MIT Press, 1992. Kapitel 12 (Hash Tables)

Die in diesem Kapitel vorgestellten verketteten Listen für die Kollisionsauflösung sind nicht immer die beste Wahl (vielleicht fällt Ihnen spontan schon eine bessere ein?). Verschiedene andere Ideen, wie man Kollisionen auflösen kann, sind in diesem Buchkapitel schön beschrieben.

[2] R. Pagh und F. F. Rodler. Cuckoo Hashing, *Proceedings of ESA 2001*, Lecture Notes in Computer Science, vol. 2161, pages 121–, 2001.

Das Kuckuckshashing gehört zu den schönsten Ideen, die die Informatik zu bieten hat. Hash-Tabellen gibt es schon sehr lange und sie sind wirklich gut erforscht. Über die Jahre sind immer raffiniertere Varianten vorgeschlagen worden. Beispielsweise hat man es geschafft, Hash-Tabellen zu entwerfen, die *garantiert* nur $O(1)$ Zeit brauchen für einen Such-Zugriff (statt, wie bei den verketteten Listen aus diesem Kapitel, bis zu $O(n)$ im Worst-Worst-Worst-Case). Das ist aber nicht so einfach und es brauchte eine ganze Folge immer längerer Theorie-Papers, bis am Ende eine sehr komplexe Datenstruktur stand (so genannte »dynamische perfekte Geburtstagstabellen«, der Name sagt schon alles).
Um so beeindruckender war dann der Artikel über das Kuckuckshashing, da hier ein extrem einfaches Verfahren vorgestellt wurde, das genauso gut ist wie die viel komplizierteren Verfahren: Man benutzt einfach zwei Hash-Tabellen mit zwei unabhängigen Hash-Funktionen. Jedes Element steht an seiner Hash-Position in einer der beiden Tabellen. Will man ein neues Element einfügen und beide der möglichen Positionen sind belegt, so platziert man das Element an eine der beiden Position. Das Element, das dort war, fliegt raus (der »Kuckuckseffekt«) und landet an seiner alternativen Position in der anderen Tabelle. Ist dort schon ein Element, so fliegt auch dieses dort raus und landet wieder in der ersten Tabelle und so fort.
Die Implementation eines Kuckuckshashing ist kaum schwieriger als das in diesem Kapitel vorgestellte Hashing. Mit einem sehr aufwändigen Beweis kann man aber zeigen, dass Kuckuckshashing »amortisiert im Erwartungswert Einfügen in Zeit $O(1)$ durchführt«. Übersetzt bedeutet das einfach, dass dieses Hashing-Verfahren optimal ist.

Teil VII

Algorithmen für schwierige Probleme

In diesem Teil wollen wir uns *schwierigen Problemen* widmen. Ein Problem heißt nicht deshalb schwierig, weil es schwierig zu verstehen wäre und weil es irgendwie komplex zu erklären ist. Es heißt *schwierig*, wenn es nur mit erheblichem Zeitaufwand gelöst werden kann.

Im Rahmen der Komplexitätstheorie hat die Theoretische Informatik sich sehr viele Gedanken darüber gemacht, welche Probleme schwierig sind und welche nicht. Leider kann man nach fast einem halben Jahrhundert Forschung das Hauptergebnis ehrlich nur so zusammenfassen: »Bei den meisten praktischen Problemen wir wissen leider auch nicht, welche wirklich schwierig sind. Aber wir haben einige ganz gut begründbare Vermutungen.« Für die Praxis hieß das: Wenn die Theorie mitteilt, dass ein Problem NP-vollständig ist (was immer das genau bedeuten mag), dann sollte man besser die Finger davon lassen.

Dem Praktiker hilft die Information »die Theoretiker glauben, dieses Problem ist schwierig, sind sich aber nicht vollkommen einig« nicht wirklich weiter, wenn das Problem nun trotzdem gelöst werden soll oder muss. Deshalb sind, teilweise recht einfache, Verfahren entwickelt worden, mit denen man schwierige Problem trotzdem »irgendwie« löst. Ein Ansatz ist, »roher Gewalt anzuwenden« und alle möglichen Lösungen durchzuprobieren, dies aber dafür recht flott. *Backtracking* ist ein Beispiel hierfür; es funktioniert aber nur bei kleinen Eingaben. Ein anderer Ansatz ist, lediglich *approximative Lösungen* zu suchen, die zwar theoretisch nicht optimal sein mögen, für praktische Zwecke aber ausreichen. Hier kann man oft *Greedyverfahren* verwenden, die einfach gierig Teillösungen immer so schnell und so stark wie möglich verbessern.

Zur Ehrenrettung der Theoretiker sei erwähnt, dass ein Verfahren zur Lösung schwieriger Probleme, der so genannte Fixed-Parameter-Ansatz, von Theoretikern entwickelt wurde. Mit diesem Ansatz lassen sich bestimmte schwierige Problem praktisch lösen für reale Eingaben von einer Größenordnung, die früher schlicht unmöglich zu lösen gewesen wären.

30-1

Kapitel 30
Graphen
Busnetze, Molekülverbindungen, Genecluster

30-2

Lernziele dieses Kapitels

1. Arten von Graphen kennen
2. Daten und ihre Struktur mittels Graphen modellieren können
3. Die Datenstruktur des Graphen verstehen
4. Konzept des Graphproblems verstehen

Inhalte dieses Kapitels

30.1 Modellierung mit Graphen 257
30.1.1 Modellierung mittels Graphen 257
30.1.2 Arten und Formalisierungen 257
30.1.3 Graphprobleme 258

30.2 Graphen in Java 259
30.2.1 Adjazenzmatrizen 259
30.2.2 Adjazenzlisten 259

30.3 Graphproblem: Minimale Gerüste 260
30.3.1 Problemstellung 260
30.3.2 Anwendungen 260
30.3.3 Optimierungsalgorithmus 261

30.4 Graphproblem: Der Handlungsreisende 261
30.4.1 Problemstellung 261
30.4.2 Anwendungen 262

Übungen zu diesem Kapitel 264

Worum es heute geht

Bei Graphen geht es darum, Sachverhalte *darzustellen*, weshalb sich auch das Wort Graphik davon ableitet. Dabei sind, mathematisch gesehen, Graphen ein *sehr* allgemeines Konzept, fast so allgemeine wie Mengen oder Relationen. Deshalb kann man auch alles und jedes als Graph darstellen, was auch gerne getan wird. Graphen modellieren »Dinge und ihre Beziehungen«.

Anders als beispielsweise bei Mengen oder Relationen gibt es nicht »die« Definition von Graphen. Zwar ist die grundlegende Idee, Dinge und ihre Beziehungen zu modellieren, immer gleich, die konkreten mathematischen Ausprägungen können aber sehr variieren. Graphen können ge-allesmögliche sein, so zum Beispiel gerichtet, gewichtet, gelabelt, gefärbt und das auch alles in Kombination. Jede dieser Grapharten hat aber ihre Daseinsberechtigung, wie wir sehen werden. Um die mathematische Definition von Graphen im Rechner abzubilden, werden in diesem Kapitel zwei Verfahren kurz skizziert: Adjazenzlisten und Adjazenzmatrizen.

Die Wirklichkeit lediglich als mehr oder weniger schöne Graphen zu modellieren, ist für sich genommen noch nicht sonderlich aufregend. Auf ein aufgeregtes »Das www kann man als ein Graph modellieren!« werden wohl die meisten Menschen zunächst etwas kühl »wie nett« erwidern. Spannend wird es erst, wenn man nun versucht, *Graphprobleme* zu lösen. Dies sind Probleme, bei denen die Eingaben Graphen sind. Ein typisches Graphproblem ist das *Erreichbarkeitsproblem*, bei dem man wissen möchte, ob man von einem gegebenen Knoten einen anderen gegebenen Knoten erreichen kann. In diesem Kapitel lernen wir erste solche Probleme kennen, weitere folgen später.

30.1 Modellierung mit Graphen

30.1.1 Modellierung mittels Graphen

Graphen: Knoten und Kanten

30-4

▶ Definition: Graph
Ein *Graph* besteht aus *Knoten*, die durch *Kanten* verbunden sind.

Die *Idee* ist, dass die Knoten *Dinge* modellieren und die Kanten *Beziehungen* modellieren.

Beispiel: Verkehrsnetze

30-5

Wir können *Verkehrsnetze* mittels Graphen wie folgt modellieren: Die *Knoten* sind Orte oder Straßenkreuzungen. Die *Kanten* sind Straßen.

✎ Zur Diskussion
Wenn wir ein Verkehrsnetz als Graph modellieren, welche Informationen sollten wir dann speichern betreffend die Knoten (die Orte) und die Kanten (die Straßen)?

Beispiele aus der Molekularbiologie

30-6

Szenario: Moleküle
Wir können *Moleküle* mittels Graphen wie folgt modellieren: Die *Knoten* sind die Atome. Die *Kanten* sind die Bindungen.

Szenario: Genregulation
Wir können *Genregulation* mittels Graphen wie folgt modellieren: Die *Knoten* sind Gene. Die *Kanten* sind die Abhängigkeiten zwischen Genen; Kanten geben also an, wie Gene einander beeinflussen.

30.1.2 Arten und Formalisierungen

Es gibt vieles, was man bei Graphen festlegen kann.

30-7

Welche Art von Knoten gibt es? Sind Beziehungen gerichtet oder ungerichtet? Sind die Knoten beschriftet? Sind die Kanten beschriftet? ...

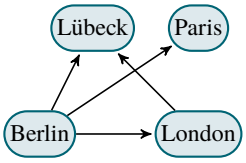
Der mathematische Begriff des Graphen.

30-8

▶ Definition: Graph
Ein *Graph* besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Solche Graphen nennen wir auch *gerichtete Graphen*.

Beispiel: Gerichteter Graph
$V = \{\text{Berlin}, \text{London}, \text{Paris}, \text{Lübeck}\}$.
$E = \{(\text{Berlin}, \text{London}), (\text{Berlin}, \text{Paris}), (\text{Berlin}, \text{Lübeck}), (\text{London}, \text{Lübeck})\}$

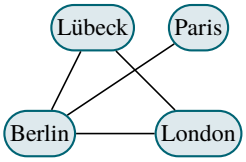


Spezielle Graphen: Ungerichtete Graphen.

30-9

Bei *ungerichteten* Graphen haben die Kanten keine Richtung. Dies kann man mathematisch unterschiedlich modellieren. Typisch ist, dass man annimmt, dass E symmetrisch ist. (Das heißt, falls $(u, v) \in E$, so auch $(v, u) \in E$.)

Beispiel: Ungerichteter Graph

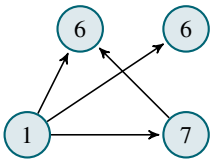


30-10

Spezielle Graphen: Knotengewichtete Graphen.

Bei *knotengewichteten* Graphen wird jedem Knoten noch eine Zahl zugeordnet, genannt *Gewicht* des Knotens.

Beispiel: Knotengewichteter Graph

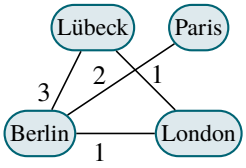


30-11

Spezielle Graphen: Kantengewichtete Graphen.

Bei *kantengewichteten* Graphen wird jeder Kante noch eine Zahl zugeordnet, genannt *Gewicht* der Kante.

Beispiel: Kantengewichteter Graph

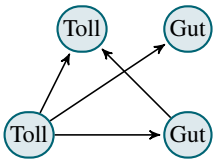


30-12

Spezielle Graphen: Gelabelte Graphen.

Manchmal möchte man den Knoten und/oder den Kanten noch ein *Label* (einen kleinen Notizzettel) ankleben.

Beispiel: Knotengelabelter Graph



30.1.3 Graphprobleme

30-13

Was sind Graphprobleme?

▶ Definition

Graphprobleme sind alle Problemstellungen, bei denen die Eingaben (kodierte) Graphen sind.

Insbesondere gibt es Graphprobleme als

1. Entscheidungsprobleme, wobei dann die Frage ist, ob ein gegebener Graph eine bestimmte Eigenschaft hat wie »Gibt es einen Weg vom ersten zum letzten Knoten?«; sowie als
2. Optimierungsprobleme, wobei dann die Frage ist, wie eine optimale Lösung aussieht (»Wie sieht der kürzeste Weg vom ersten zum letzten Knoten aus?«).

30.2 Graphen in Java

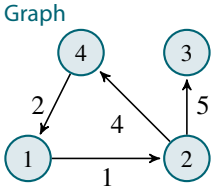
30.2.1 Adjazenzmatrizen

Was ist eine Adjazenzmatrix?

30-14

Ein Graph habe n Knoten. Bilde nun eine Tabelle, die n Zeilen und n Spalten hat. Trage in die i-te Zeile und die j-te Spalte das Gewicht der Kante vom i-ten Knoten zum j-ten Knoten ein. Trage dort eine 0 ein, wenn es keine Kante gibt.

Graph



Adjazenzmatrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Graphen als Adjazenzmatrizen in Java

30-15

```
class GraphAsAdjacencyMatrix {
  int [][] weights;

  GraphAsAdjacencyMatrix (int numberOfVertices) {
    this.weights =
      new int[numberOfVertices][numberOfVertices];
  }

  void addEdge(int u, int v, int weight) {
    this.weights[u][v] = weight;
  }

  void removeEdge(int u, int v) {
    this.weights[u][v] = 0;
  }

  boolean connected(int u, int v) {
    return (this.weights[u][v] != 0); // Boole'scher Ausdruck!
  }

  int weight(int u, int v) {
    return this.weights[u][v];
  }
}
```

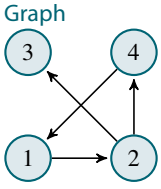
30.2.2 Adjazenzlisten

Was ist eine Adjazenzliste?

30-16

Ein Graph habe n Knoten. Für jeden Knoten wird eine Liste gespeichert, in der alle Knoten gespeichert sind, zu denen er eine Kante hat.

Graph



Adjazenzlisten
Liste von Knoten 1: 2
Liste von Knoten 2: 3, 4
Liste von Knoten 3: leer
Liste von Knoten 4: 1

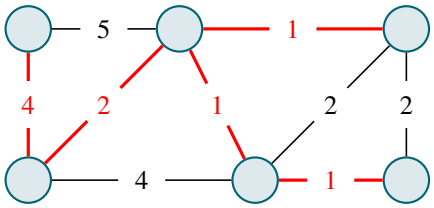
30.3 Graphproblem: Minimale Gerüste

30.3.1 Problemstellung

30-17

Einführung zu Gerüsten

▶ Definition: Gerüst
Ein *Gerüst* (englisch *spanning tree*) ist ein Teilbaum des Graphen, der jeden Knoten berührt.
Ein *minimales Gerüst* ist ein Gerüst, so dass die Summe der Kantengewichte minimal ist.



30-18

Das Gerüstproblem als Optimierungsproblem

Das Minimale-Gerüst-Problem

Eingaben Ein kantengewichteter Graph.

Ausgabe Ein Gerüst, so dass die Summe der Gewichte im Baum minimal ist.

30.3.2 Anwendungen

30-19

Eine Anwendung des Gerüstproblems.

Literatur

[1] Y. Inbar, H. Benyamini, R. Nussinov und H. Wolfson Protein structure prediction via combinatorial assembly of sub-structural units, *Bioinformatics,* 19(1) i158–i168, 2003.

Vorgehen in obiger Arbeit:

1. Teile die Aminosäuresequenz in kleine Stücke auf.
2. Berechne für die Stücke, welcher bekannten Aminosäuresequenz sie am ähnlichsten sind. Weise den Stücken die Raumstruktur dieser ähnlichsten Sequenzen zu.
3. Berechne für je zwei Stücke, wie gut sie zusammenpassen (wie Legosteine).
4. Bilde daraus einen Graphen, dessen Knoten die Stücke sind und dessen Kanten für je zwei Stücke angeben, wie gut sie zusammenpassen.
5. Berechne ein minimales Gerüst in dem Graphen.
6. Puzzle die Stücke entsprechend dem Gerüst zusammen.

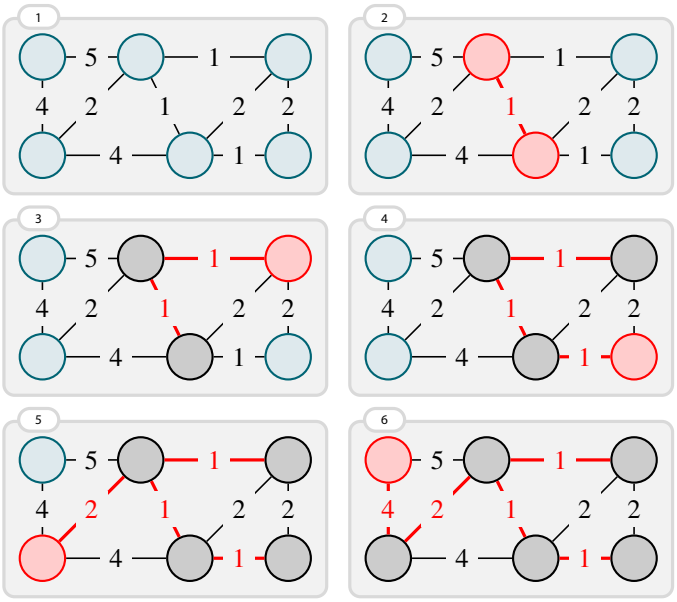
30.3.3 Optimierungsalgorithmus

Ein effizienter Algorithmus für das Gerüstproblem. 30-20

Algorithmus

Eingabe ist ein kantengewichteter Graph.

1. Finde eine Kante minimalen Gewichts. Diese Kante bildet unser initiales Teilgerüst.
2. Wiederhole folgendes, solange noch kein Gerüst vorliegt:
   - 2.1 Finde die Kante minimalen Gewichts, die das Teilgerüst mit einem noch nicht erreichten Knoten verbindet.
   - 2.2 Füge die Kante zum Teilgerüst hinzu.



30.4 Graphproblem: Der Handlungsreisende

30.4.1 Problemstellung

Problem des Handlungsreisenden. 30-21

Ein Handlungsreisender möchte in einer Rundreise eine Reihe von Städten besuchen. Dabei soll er jede Stadt genau einmal besuchen. Die Benzinkosten für eine Fahrt zwischen zwei Städten hängt linear von deren Entfernung ab. Ziel ist es, eine möglichst billige Rundreise zu finden.

Vorführung des Problems anhand einer Landkarte. Regie

Formalisierungen des Handlungsreisenden-Problems 30-22

Euklidisches Handlungsreisenden-Problem

Eingaben Ein Menge von Punkten in der Ebene.

Ausgabe Eine Rundreise (Folge der Punkte, die jeden Punkt genau einmal enthält), so dass die Summe der Länge der Strecken entlang der Rundreise minimal ist.

Allgemeines Handlungsreisenden-Problem

Eingaben Ein kantengewichteter Graph.

Lösungen Rundreise (Folge von miteinander verbundenen Knoten, die jeden Knoten genau einmal enthält), so dass die Summe der Gewichte der Kanten entlang der Rundreise minimal ist.

30.4.2 Anwendungen

30-23

Ein Ziel der Bioinformatik: Gene-Clustering

Im Rahmen einer Untersuchung wird für eine große Anzahl Gene untersucht, wie stark sie unter verschiedenen Bedingungen exprimiert werden, was für jedes Gen einen Eigenschaftsvektor liefert. Ziel ist es nun, Cluster von Genen zu bilden, deren Eigenschaftsvektoren »zusammengehören«. Solches »Clustering« ist in hochdimensionalen Räumen eine Kunst für sich.

30-24

Beispiel aus der Literatur zum Gene-Clustering-Problem.

Zitat aus

Literatur

[1] S. Climer und W. Zhang. A traveling salesman's approach to clustering gene expression data, Technischer Bericht WUCSE-2005-5, *Washington University in St. Louis*, 2005.

> »The dataset consists of 2,467 genes in the budding yeast *Saccharomyces cerevisiae* that were studied during the diauxic shift, mitotic cell division cycle, sporulation, and temperature and reducing shocks, yielding 79 measurements that are used as the features for the genes.«

30-25

Vorgehensweise zum Gene-Clustering.

Literatur

[1] S. Climer und W. Zhang. A traveling salesman's approach to clustering gene expression data, Technischer Bericht WUCSE-2005-5, *Washington University in St. Louis*, 2005.

Vorgehen in obiger Arbeit:

1. Messe für jedes Gen sein Verhalten unter verschiedenen Einflüssen.
2. Dies liefert für jedes Gen einen *Eigenschaftsvektor*.
3. Berechne für jedes Genpaar eine *»Distanz«*, basierend auf der »Ähnlichkeit« der Eigenschaftsvektoren.
4. Dies induziert einen so genannte *Metrik* auf den Genen. Vereinfachend stellen wir uns die Gene in der Ebene vor und die Metrik ist durch die Entfernungen gegeben.
5. Löse das Handlungsreisenden-Problem für die Gene.
6. Alle aufeinanderfolgenden Städte mit kleiner Distanz bilden einen Cluster.

30-26

Was haben Handlungsreisende mit Gene-Clustering zu tun?

Idee

Eine kürzeste Rundreise wird nicht oft zwischen den Clustern »herumspringen«.

Zusammenfassung dieses Kapitels

30-27

▶ Graph

Graphen bestehen aus *Knoten* und *Kanten* zwischen Paaren von Knoten. Wichtige Arten von Graphen sind:

»gerichtet« Die Kanten haben eine »Richtung«, gehen also »von« einem Knoten »zu« einem Knoten.

»ungerichtet« Die Kanten haben keine »Richtung«.

»gewichtet« Eine »Wichtung« ist eine Zahl, die die »Bedeutung« einer Kante oder einem Knoten angibt. Man kann sowohl »knotengewichtete« wie auch »kantengewichtete« Graphen betrachten.

»gelabelt« Ein »Label« ist ein kleiner Text, der an Kanten oder Knoten »geklebt« ist.

▶ Implementation in Java

Man kann Graphen als *Adjazenmatrizen* implementieren oder mit Hilfe von *Adjazenzlisten*.

▶ Graphprobleme

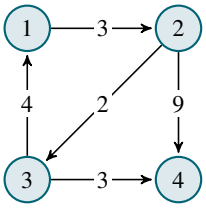
Ein *Graphproblem* ist ein Berechnungsproblem, bei dem Graphen die Eingaben sind. Drei typische Beispiele sind:

- Das Finden kürzester Wege.
- Das Finden eines minimalen Gerüsts.
- Das Finden einer minimalen Rundreise.

Übungen zu diesem Kapitel

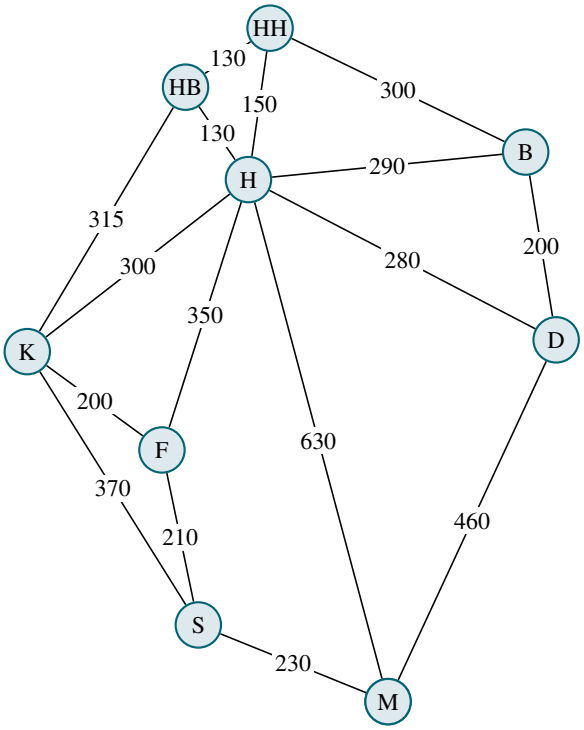
Übung 30.1 Adjazenzmatrix und Adjazenzliste angeben, leicht

Geben Sie zu folgendem gerichteten Graphen mit Kantengewichten die Adjazenzmatrix, die Adjazenzliste und den Abstand des Knotens 4 vom Knoten 1 an.



Übung 30.2 Gerüst konstruieren, leicht

Konstruieren Sie zu folgendem ungerichteten Graphen mit dem Algorithmus aus der Vorlesung ein minimales Gerüst und berechnen Sie die Summe der Kantengewichte des Gerüsts.



Übung 30.3 Gerüst-Algorithmus in Java implementieren, mittel

Benutzen Sie für diese Übung folgende Variante der Klasse `GraphAsAdjacencyMatrix`, bei der stets ein ungerichteter Graph erzeugt wird, indem die Adjazenzmatrix symmetrisch gehalten wird:

```
class GraphAsAdjacencyMatrix {
  int [][] weights;

  GraphAsAdjacencyMatrix (int numberOfVertices) {
    weights =
      new int[numberOfVertices][numberOfVertices];
  }

  void addEdge(int u, int v, int weight) {
    weights[u][v] = weight;
    weights[v][u] = weight;
  }

  void removeEdge(int u, int v) {
    weights[u][v] = 0;
    weights[v][u] = 0;
  }

  boolean connected(int u, int v) {
    return weights[u][v] != 0;
  }
```

```
  int weight(int u, int v) {
    return weights[u][v];
  }
}
```

Erweitern Sie diese Klasse um eine Methode `int minimumSpanningTree()`, die die Summe der Kantengewichte des minimalen Gerüsts des ungerichteten Graphen berechnet und zurückgibt. Das Gerüst selbst soll *nicht* zurückgegeben werden. Testen Sie Ihre Methode anhand des Graphen aus Übung 30.2. Diesen können Sie mit den folgenden Java-Befehlen erzeugen:

```
// Graph erzeugen
GraphAsAdjacencyMatrix g = new GraphAsAdjacencyMatrix(10);
g.addEdge(0,1,130); // HH - HB
g.addEdge(3,0,150); // H - HH
g.addEdge(1,3,130); // HB - H
g.addEdge(3,4,290); // H - B
g.addEdge(3,6,630); // H - M
g.addEdge(7,6,230); // S - M
g.addEdge(5,3,280); // D - H
g.addEdge(5,4,200); // D - B
g.addEdge(5,6,460); // D - M
g.addEdge(7,8,210); // S - F
g.addEdge(9,3,300); // K - H
g.addEdge(1,9,315); // HB - K
g.addEdge(0,4,300); // HH - B
g.addEdge(9,8,200); // K - F
g.addEdge(3,8,350); // H - F
g.addEdge(7,9,370); // S - K

// Gerüst berechnen und ausgeben
System.out.println(g.minimumSpanningTree());
```

31-1

Kapitel 31
Algorithmendesign
Von gierigen Algorithmen und der Suche nach dem Minotaurus

31-2

Lernziele dieses Kapitels

1. Greedy-Verfahren kennen und implementieren können
2. Backtracking-Verfahren kennen und entwerfen können

Inhalte dieses Kapitels

31.1 Was ist Algorithmendesign? 267

31.2 Greedy-Verfahren 267
31.2.1 Idee . . . 267
31.2.2 Beispiel: Münzproblem . . . 268
31.2.3 Beispiel: Bin-Packing . . . 268
31.2.4 Implementation . . . 269
31.2.5 Vor- und Nachteile . . . 269

31.3 Backtracking 270
31.3.1 Idee . . . 270
31.3.2 Beispiel: Hamilton'sches Kreisproblem . 270
31.3.3 Beispiel: Färbeproblem . . . 271
31.3.4 Implementation . . . 272
31.3.5 Vor- und Nachteile . . . 273

Übungen zu diesem Kapitel 274

Worum es heute geht

Aus *de.wikipedia.org/wiki/Minotauros*

Minos, ein Sohn des Zeus, der auf dem vom Meer umgebenen Kreta wohnte, bat seinen Onkel, den Meeresgott Poseidon, ihm zur Festigung seiner Königswürde und zur Abschreckung eventueller Thronanwärter ein Wunder zu gewähren. Er solle einen weißen Stier aus dem Meer emporsteigen lassen, später wolle er ihn dem Gotte auch opfern. Poseidon ließ daraufhin einen Stier aus dem Wasser steigen. Kretas König fand den Stier jedoch derart schön, dass er ihn in seine Herde aufnahm und statt dessen ein minderwertigeres Tier opferte.

Poseidon erzürnte und verfluchte Minos Frau Pasiphaë, die sich daraufhin in den Stier verliebte. Sie ließ sich von Daidalos ein hölzernes Kuhgestell bauen und eine Kuhhaut darüberspannen, um dann in dieses Gestell zu kriechen und sich in diesem mit dem weißen Stier zu vereinigen. Aus dieser Vereinigung ging der Minotauros hervor, eine Gestalt mit menschlichem Körper und dem Kopf eines Stieres.

Minos ließ für das Tierwesen, das er eigentlich töten wollte (zeugte dieses doch auch vom Fehltritt seiner Gemahlin), auf Bitten seiner Tochter Ariadne, die ihn am Leben lassen wollte, durch Daidalos ein Gefängnis, das Labyrinth in Knossós, erbauen.

Herakles befreite als siebte seiner zehn Aufgaben den kretischen Stier und brachte ihn auf die Peloponnes. Dort richtete dieser Verwüstungen und Unheil an. Androgeos, ein Sohn des Minos, hielt sich gerade in der Gegend auf, wurde aber bei der Jagd auf den Stier hinterrücks ermordet. Deswegen führte Minos nun Krieg gegen Athen. Da Athen nicht zu bezwingen war, erbat Minos Hilfe von seinem Vater (Zeus), die dieser gewährte. Er schickte die Pest, und Athen ergab sich. Doch um Minos zu besänftigen, so berichtet die attische Volkssage, musste Athen nun jeweils alle neun Jahre sieben Jünglinge und sieben Jungfrauen als Tributzahlung nach Kreta schicken, die von Minos zu Minotaurus ins Labyrinth geschickt und so diesem geopfert wurden.

Schließlich löste Theseus, der spätere König von Athen, das Problem, indem er sich selbst mit der dritten Tributfahrt auf den Weg nach Kreta machte. Minos gestattete Theseus den Zugang zum Labyrinth, in der Hoffnung, dass Theseus vom Minotauros gefressen würde. Theseus konnte jedoch den Minotauros besiegen und das Labyrinth wieder verlassen. Die kretische Prinzessin Ariadne, die zwar bereits mit Dionysos verlobt war, sich jedoch in den kühnen Recken verliebt hatte, hatte ihm geholfen, indem sie ihm den bekannten Ariadnefaden und seltsame Pillen aus Pech und Haaren, die in den Rachen des Minotaurus zu werfen waren, gegeben hatte. Der Rat, auf der Suche nach dem Ungeheuer den Faden abzuspulen, kam von Daidalos. Theseus erschien mit dem Haupt in der Hand am Eingang des Labyrinths und entfloh mit Ariadne in der allgemeinen Aufregung, nicht ohne zuvor alle minoischen Schiffe unbrauchbar gemacht zu haben nach Naxos. Daidalos wurde daraufhin zur Strafe in das leerstehende Labyrinth eingesperrt und entkam später mit seinem Sohn Ikarus.

Public domain

31.1 Was ist Algorithmendesign?

Wie entwirft man neue Algorithmen? 31-4

Will man einen neuen Algorithmus entwerfen, so gibt es verschiedene Herangehensweisen:

- Wenn das zu lösende Problem einfach ist, so »sieht« man den Algorithmus vielleicht sofort.
- Man kennt schon einen Algorithmus, zumindest für ein sehr verwandtes Problem. Diesen wandelt man leicht ab.
- Man hat eine Eingebung/Erleuchtung.
- Man benutzt eines der *Standardverfahren des Algorithmendesign*.

Wichtige Verfahren des Algorithmendesign. 31-5

Eine Auswahl der Verfahren, die sich beim Design von Algorithmen bewährt haben:

1. Rekursion (Zurückführung des Problems auf ein vereinfachtes)
2. Teile-und-Herrsche (Aufteilen in Teilprobleme, Beispiele sind Mergesort und Quicksort)
3. Greedy-Verfahren (heute)
4. Backtracking (heute)
5. Rohe Gewalt (stures Durchprobieren aller Möglichkeiten, liebevoll Brute-Force genannt)
6. Dynamische Tabellen (intelligente Art der Rekursion)

All diese Verfahren *sind keine Algorithmen* sondern *Grundideen, nach denen man Algorithmen entwerfen kann.*

31.2 Greedy-Verfahren

31.2.1 Idee

Die Grundidee von Greedy. 31-6

Nimm, was du kriegen kannst!

Die Grundidee von Greedy etwas wissenschaftlicher formuliert. 31-7

Greedy-Verfahren dienen (in der Regel) zur Lösung von *Optimierungsproblemen*. Ausgehend von einer *leeren Teillösung* wird diese Teillösung *schrittweise vervollständigt*. Die durchgeführte Vervollständigung ist immer die, die *aktuell den meisten Vorteil bringt* (das Verfahren »giert« nach einem Vorteil). Wenn die Lösung fertig ist, terminiert das Verfahren.

31.2.2 Beispiel: Münzproblem

31-8 Das Optimierungsproblem »Münzrückgabe«.

Das Optimierungsproblem

Eingabe Ein (Multi-)Menge an Münzen und ein Wert w.

Ausgabe Ein möglichst kleine Teilmenge der Münzen, deren Summe w ist.

Beispiel

Beim Münzrückgabeproblem sollen 1,86 Euro zurückgegeben werden, folgende Münzen stehen zur Verfügung :

Dann lautet die richtige Ausgabe: dreimal 50 Cent, einmal 20 Cent, einmal 10 Cent, dreimal 2 Cent.

31-9 Der Greedy-Algorithmus zur »Lösung« des Münzproblems.

Algorithmus

Wir transferieren nach und nach Münzen vom *Eingabestapel* auf den *Rückgabestapel*. Anfangs ist der Rückgabestapel leer. Solange möglich, wiederhole Folgendes:

1. Von den verbleibenden Münzen im Eingabestapel, identifiziere jede Münze, *die man noch auf den Rückgabestapel legen könnte, so dass dessen Wert höchstens w ist.*
2. Unter all diesen Münzen, transferiere die Münze mit dem höchsten Wert vom Eingabestapel auf den Rückgabestapel.

Regie *Vorführung des Greedy-Verfahrens anhand des Beispiels.*

31-10 Beobachtungen zum Greedy-Verfahren.

- Das Verfahren ist *sehr schnell und einfach*.
- Das Verfahren findet *aber nicht immer eine Lösung*.
- Die Euro-Münzen sind von ihren Werten her *so gemacht*, dass das Verfahren *immer funktioniert, wenn von jeder Münzart genügend Münzen vorhanden sind*.
- In diesem Fall liefert der Algorithmus sogar *immer eine optimale Lösung*.

31.2.3 Beispiel: Bin-Packing

31-11 Das Bin-Packing-Problem als Optimierungsproblem

Das Bin-Packing-Problem

Eingabe Eine Liste von Objektgrößen $(g_1, \ldots, g_n)$ und eine Eimergröße b.

Ausgabe Zuordnung von Objekten zu möglichst wenigen Eimern, so dass die Summe der Größen aller Objekte, die demselben Eimer zugeordnet sind, maximal b ist.

31-12 Ein Greedy-Algorithmus für das Bin-Packing-Problem.

Der First-Fit-Algorithmus

Für jeden Gegenstand tue folgendes:

1. Finde, von links beginnend, den ersten Eimer, in den der Gegenstand noch passt.
2. Platziere den Gegenstand in diesen Eimer.

Regie *Durchführung des Algorithmus durch einen Studenten anhand eines Modells, bei dem Gegenstände durch Rohre und Container durch Stangen repräsentiert sind.*

31.2.4 Implementation

Wie implementiert man Greedy-Verfahren? 31-13

Allgemeines Schema eines Greedy-Algorithmus

Solange die gefundene Teillösung noch nicht fertig ist, wiederhole:

- Unter allen möglichen nächsten Schritten, bestimme denjenigen, der *den meisten Gewinn bringt*.
- Erweitere die Teillösung entsprechend.

Beispielcode für das Münzproblem 31-14

```
int[] wechselgeld (int[] geldbeutel, double betrag) {
  // Münzwerte in Euros
  double[] muenzwerte = {2, 1, .5, .2, .1, .05, .02, .01};
  int[] wechselgeld = {0, 0, 0, 0, 0, 0, 0, 0};
  while (betrag > 0) {
    int beste_wahl = -1;
    for (int i = 0; i < muenzwerte.length; i++) {
      if (geldbeutel[i] > 0 && muenzwerte[i] <= betrag) {
        beste_wahl = i;
        break;
      }
    }
    if (beste_wahl == -1) // Kein Wechselgeld mehr!
      return null;
    wechselgeld[beste_wahl]++;
    geldbeutel[beste_wahl]--;
    betrag = betrag - muenzwerte[best_wahl];
  }
  return wechselgeld;
}
```

31.2.5 Vor- und Nachteile

Zusammenfassung der Vor- und Nachteile von Greedy-Verfahren. 31-15

Vorteile

+ Fast immer einsetzbar
+ Arbeiten immer sehr schnell
+ Produzieren in der Regel gute Lösungen
+ Leicht zu merken und zu programmieren

Nachteile

– Liefern manchmal gar keine Lösungen
– Liefern manchmal nur schlechte Lösungen

31.3 Backtracking

31.3.1 Idee

31-16

Das Labyrinth des Minotaurus.

31-17

Public domain

Warum wir in einem Labyrinth mit Greedy-Strategien nicht weiterkommen.

Greedy funktioniert *nicht*: In einem Labyrinth ist es *schwierig, lokal eine optimale Entscheidung* zu fällen. Kommt man an einer Kreuzung an, so erscheint jede Richtung »gleichermaßen gut oder schlecht«.

Wie löst man nun ein Labyrinthproblem?

- Man probiert die verschiedenen Richtungen nacheinander durch.
- Stößt man auf eine *Sackgasse*, so muss man die letzten Schritte rückwärts gehen (*backtracking*).
- Dazu führt man einen Ariadnefaden mit, den man aus- und einrollt.

31-18

Die Grundideen des Backtracking.

Ein Algorithmus muss eine *Folge von Auswahlen* treffen.

Beispiel: Welchen Knoten sollte ich als nächstes besuchen?

Beispiel: In welchen Container sollte ich die Waren packen?

Die *Strategie* ist, eine Folge von Auswahlen zu treffen, und diese, falls sie noch nicht optimal ist, *immer wieder in den letzten Schritten zu revidieren*.

31.3.2 Beispiel: Hamilton'sches Kreisproblem

31-19

Erstes Beispiel: Hamilton'sches Kreisproblem

▶ **Definition:** Hamilton'scher Kreis

Sei G ein Graph. Ein *Hamilton'scher Kreis* ist eine Folge von Knoten des Graphen, so dass

1. jeder Knoten genau einmal in der Folge vorkommt und
2. jeder Knoten mit seinem Nachfolger in der Folge in dem Graphen durch eine Kante verbunden ist und
3. der letzte mit dem ersten durch eine Kante verbunden ist.

▶ **Definition:** Hamilton'sches Kreisproblem

Das *Hamilton'sche Kreisproblem* ist ein Entscheidungsproblem. Eingabe ist ein Graph. Die Frage ist, ob dieser einen Hamilton'schen Kreis enthält.

Offenbar ist das Hamilton'sche Kreisproblem etwas leichter als das allgemeine Handlungsreisendenproblem, da es eine »Vorstufe« dazu darstellt.

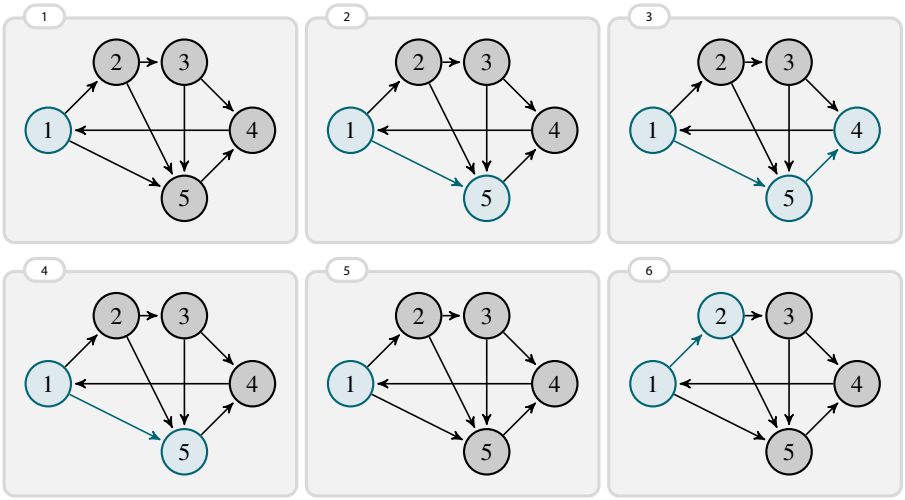
31-20

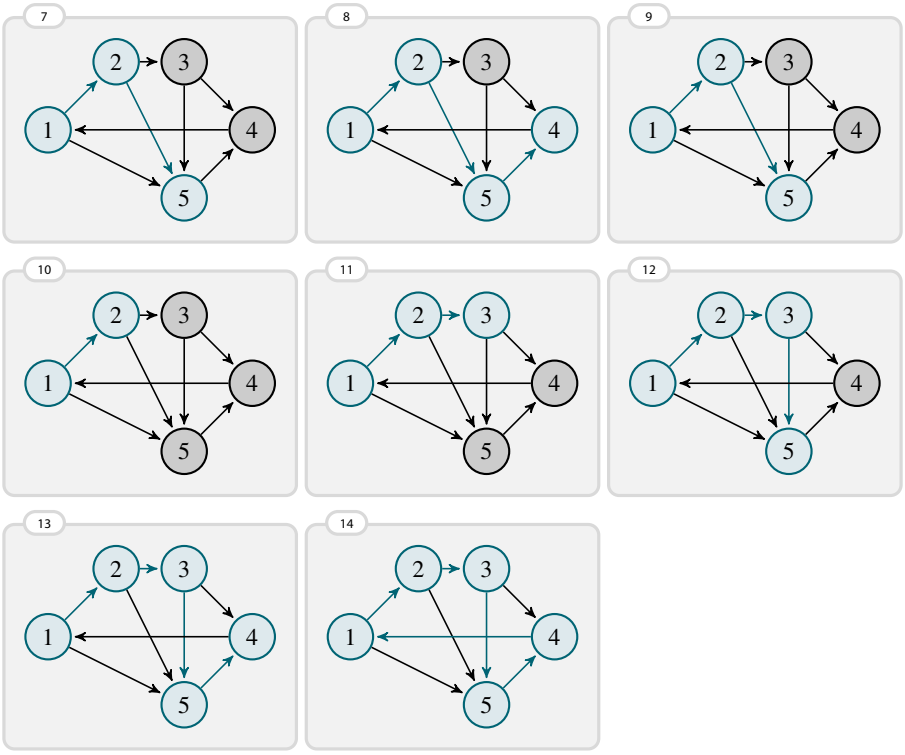
Backtracking-Algorithmus zum Finden Hamilton'scher Kreise

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Wenn alle Knoten besucht sind und es eine Kante vom Ende zum Anfang gibt, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.





31.3.3 Beispiel: Färbeproblem

Das Färbeproblem und ein Backtracking-Algorithmus

31-21

Die Problemstellung

Eingabe Ein Graph.

Lösungen Färbung des Graphen (je zwei durch eine Kante verbundenen Knoten müssen eine unterschiedliche Farbe haben)

Maß Anzahl der benutzten Farben

Ziel Minimierung

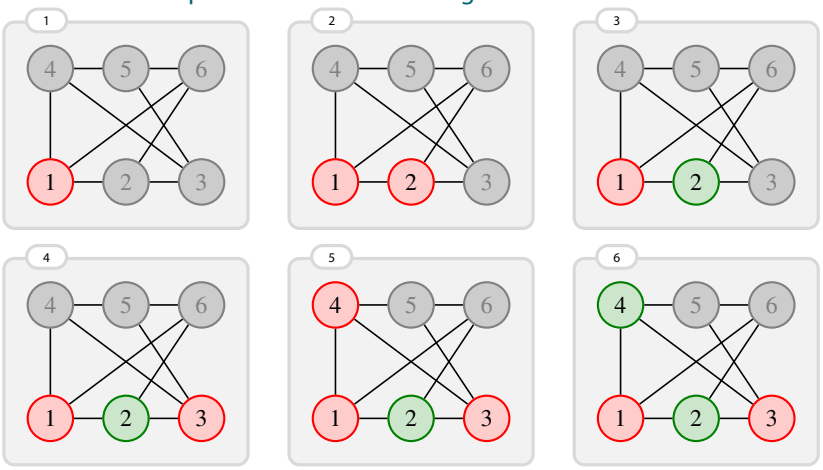
Algorithmus zur 3-Färbung

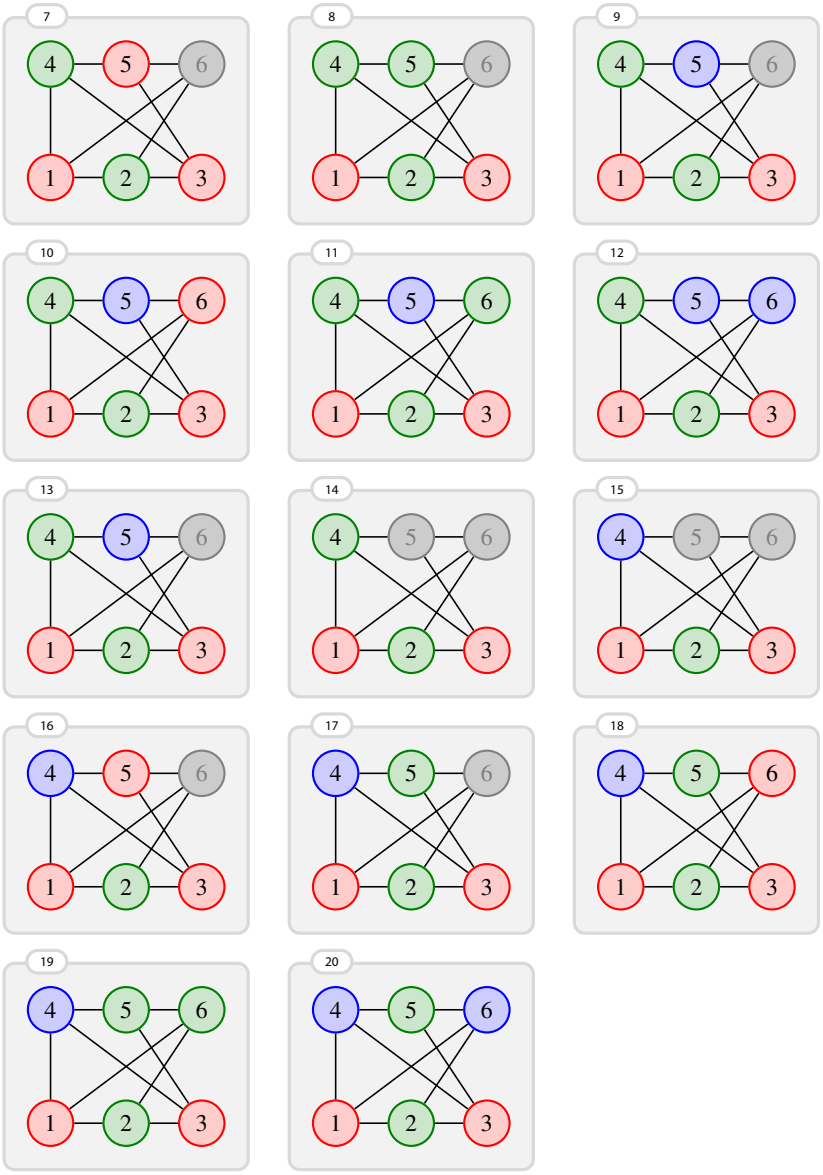
Wiederhole Folgendes, bis es nicht mehr geht:

1. Färbe die Knoten mit drei Farben, bis es nicht mehr geht.
2. Wenn alle Knoten gefärbt sind, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis man mit einer andere Färbung weitermachen kann.

Färben eines Graphen mittels Backtracking

31-22





31.3.4 Implementation

31-23

Wie implementiert man Backtracking?

Es gibt zwei Ansätze zur Implementation von Backtracking-Verfahren:

1. Beim *iterativen Ansatz* wird der »Ariadnefaden« in einem Array gespeichert. Jede Array-Position gibt eine der bisherigen Entscheidungen an.
   Immer, wenn ein Backtracking nötig ist, wird die letzten Entscheidungen geändert. Lässt sich diese nicht mehr ändern, so wird zur vorletzten zurückgegangen und diese geändert. Lässt sich auch diese nicht mehr ändern, so geht es zurück zur vorvorletzten und so weiter.
2. Beim *rekursiven Ansatz* wird der »Ariadnefaden« indirekt durch den Compiler verwaltet.
   Jedesmal, wenn eine Entscheidung nötig ist, ruft sich eine Methode rekursiv für alle möglichen Entscheidungen auf.

Insgesamt ist ein Backtracking in der Regel schwierig zu implementieren.

31.3.5 Vor- und Nachteile

Zusammenfassung der Vor- und Nachteile von Backtracking-Verfahren. 31-24

Vorteile

+ Liefern immer optimale Ergebnisse.
+ Funktionieren im Notfall immer.

Nachteile

– Schwierig zu programmieren.
– Funktionieren nur bei sehr kleinen Eingaben (max. 30 Knoten).
– Werden niemals (!) bei wirklich großen Graphen (über 500 Knoten) funktionieren.

Zusammenfassung dieses Kapitels

▶ Grundlegende Verfahren des Algorithmendesigns 31-25

– Rekursion
– Teile-und-Herrsche
– Greedy-Verfahren
– Backtracking
– Brute-Force
– Dynamische Tabellen

▶ Greedy-Verfahren

– Idee: Erweitere eine Teillösung immer in lokal optimaler Weise.
– Vorteile: Schnell, einfach, oft gute Lösungen
– Nachteile: Führt nicht immer zum Ziel.
– Anwendungsbeispiele: Münzrückgabe, Bin-Packing

▶ Backtracking-Verfahren

– Idee: Erweitere eine Teillösung in allen lokal möglichen Weisen und revidiere getroffene Entscheidungen, wenn keine Lösung gefunden wurde.
– Vorteile: Findet immer die optimale Lösung.
– Nachteile: Dauert schon bei kleinen Eingabegrößen sehr oder zu lange.
– Anwendungsbeispiele: Münzrückgabe, Bin-Packing, Färbung, Hamilton'sches Kreisproblem

Übungen zu diesem Kapitel

Copyright by Türelio, Creative Commons Attribution Sharealike License

Übung 31.1 Greedy-Algorithmen entwerfen, leicht

Bei Scheduling geht es allgemein darum, Aktivitäten auf Orte geschickt zu verteilen. Aktivitäten können beispielsweise das Ein- und Aussteigen von Personen an einem Bahnsteig sein; Orte können die Bahnsteige sein, an denen die Züge ankommen.

Formal kann man das Scheduling-Problem so fassen:

Eingabe Eine Liste von Paaren $(ankunft_i, abfahrt_i)$ von Ankunfts- und Abfahrtzeiten verschiedener Züge.

Ausgaben Zuordnung der Paare auf möglichst wenige Bahnsteige, so dass sich an einem Bahnsteig keine Aktivitäten überlappen.

Geben Sie drei sinnvolle Greedy-Strategien für das Scheduling-Problem an.

Übung 31.2 First-Fit ist nicht optimal, leicht

Das Greedy-Verfahren First-Fit für Bin-Packing liefert nicht immer eine optimale Lösung. Geben Sie eine Eingabe an, für die es besonders schlecht arbeitet.

Übung 31.3 Wechselgeld, mittel

Wie gut würde der Greedy-Algorithmus zur Lösung des Münzproblems funktionieren, wenn es neben den bekannten Münzen auch 9- und 90-Cent-Münzen geben würde?

Übung 31.4 Bruchrechnen wie ein Ägypter

Die alten Ägypter konnten bereits mit Brüchen rechnen, allerdings kannten sie das arabische Zahlensystem noch nicht und mussten mit ihren Hieroglyphen sparsam umgehen. Ägyptische Brüche wurden daher stets als Summen sogenannter *Elementarbrüche*, also Brüche mit Zähler 1, dargestellt. So hätte Kleopatra vermutlich den Bruch $\frac{3}{7}$ als Summe $\frac{1}{3} + \frac{1}{11} + \frac{1}{231}$ geschrieben (wobei sie natürlich die entsprechenden Hieroglyphen benutzt hätte). Man kann sich überlegen, dass es tatsächlich immer solch eine Darstellung gibt.

1. Geben Sie eine Greedy-Strategie zur Lösung dieses Problems an.
2. Implementieren Sie diese Greedy-Strategie in einer Java-Methode

```
int[] aegyptischer_bruch( int zaehler, int nenner )
```

wobei Zähler und Nenner des darzustellenden Bruchs als Ganzzahlen gegeben sind und die Methode ein Array von ganzen Zahlen zurückgeben soll, so dass die Summe der Kehrwerte dieser Zahlen gleich `zaehler/nenner` ist. Zum Beispiel so:

```
int[] aegyptisch_3_7 = aegyptischer_bruch( 3, 7 );
// Jetzt hat aegyptisch_3_7 den Wert { 3, 11, 231 }
```

Gehen Sie dabei davon aus, dass der gegebene Bruch kleiner als 1 ist.

32-1

Kapitel 32

Kürzeste und längste Wege

Wer sucht, der findet

Lernziele dieses Kapitels

1. Konzept der Tiefen- und Breitensuche verstehen
2. Dijkstra-Algorithmus für das Kürzeste-Wege-Problem kennen
3. Backtracking-Algorithmus für das Längste-Wege-Problem kennen

32-2

Inhalte dieses Kapitels

32.1	Einführung	276
32.2	Erreichbarkeitsproblem	276
32.2.1	Problemstellung	276
32.2.2	Traversierung I: Breitensuche	277
32.2.3	Traversierung II: Tiefensuche	278
32.2.4	Vergleich	280
32.3	Kürzeste Wege	280
32.3.1	Problemstellung	280
32.3.2	Lösung I: Breitensuche	280
32.3.3	Lösung II: Dijkstra-Algorithmus	281
32.4	Längste Wege	282
32.4.1	Problemstellung	282
32.4.2	Lösung: Backtracking	283
	Übungen zu diesem Kapitel	284

Worum es heute geht

Find the Longest Path

by Dan Barrett

(Original *The Longest Time* by Billy Joel)

Oh, oh, oh,
find the longest path.
Oh, oh, oh,
find the longest path.

If you said P is NP tonight,
there would still be papers left to write.
I have a weakness:
I'm addicted to completeness
and I keep searching for the longest path.

The algorithm I would like to see
is of polynomial degree.
But it's elusive,
nobody has found conclusive
evidence that we can find the longest path.

I have been hard
working for so long.
I swear it's right,
and he marks it wrong.

Somehow I'll feel sorry when it's done:
GPA 2.1
is more than I hope for.

Garey, Johnson, Karp and other men (and women)
try to make it order $n \log n$.
Am I a math fool
if I spend my life in grad school
forever following the longest path?

Oh, oh, oh,
find the longest path.
Oh, oh, oh,
find the longest path.
Oh, oh, oh,
find the longest path…

32.1 Einführung

32-4

Worum es in diesem Kapitel geht.

- Ganz allgemein geht es bei *Wege-Finde-Problemen* darum, in einem Graphen einen Weg zwischen zwei gegebenen Knoten zu finden.
- Die einfachste Variante ist das *Erreichbarkeitsproblem*, bei dem es nur darum geht, ob überhaupt ein Weg existiert.
- Etwas schwieriger ist das *Kürzeste-Wege-Problem*, bei dem man einen kürzesten Weg finden möchte.
- Noch schwieriger ist das *Längste-Wege-Problem*, bei dem man einen möglichst langen Weg finden möchte (wobei man aber keinen Knoten zweimal besuchen darf).

In diesem Kapitel sollen *Algorithmen zur Lösung dieser Probleme vorgestellt werden*.

32-5

Warum sind die Problem interessant?

Das *Finden von Wegen in Graphen* kommt in sehr vielen Anwendungen vor:

Beispiele

- Navigationsgerät
- Routenberechner der Bahn
- Analyse von Gen-Pathways

✎ Zur Diskussion

Nennen Sie weitere Anwendungen, bei denen man sich für kürzeste Wege interessiert. Fallen Ihn auch Anwendungen ein, bei welchen *längste* Wege wichtiger sind?

32.2 Erreichbarkeitsproblem

32.2.1 Problemstellung

32-6

Das Erreichbarkeitsproblem formal.

▶ Definition: Weg in einem Graph

Sei G ein Graph. Ein *Weg (auch Pfad genannt) in G* ist eine Folge von *unterschiedlichen* Knoten $k_1, \ldots, k_n$, so dass es jeweils eine Kante von k_i zu k_{i+1} in G gibt.

▶ Definition: Erreichbarkeitsproblem

Das *Erreichbarkeitsproblem* ist folgendes Entscheidungsproblem:

Eingabe (Gerichteter) Graph sowie zwei Knoten s und t in dem Graphen (»source« und »target«).

Frage *Gibt* es einen Weg von s nach t in G?

Erste Lösungsidee: Alle erreichbaren Knoten besuchen.

32-7

Um herauszufinden, ob t erreichbar ist, *besucht* man alle von s aus erreichbare Knoten – ist t dabei, so ist t erreichbar, sonst eben nicht. Den Vorgang, alle von einem Knoten aus erreichbaren Knoten einmal zu besuchen, nennt man *traversieren*. Wie bei Bäumen gibt es mehrere mögliche *Reihenfolgen,* nach denen man die Knoten besuchen kann. Wir betrachten die zwei wichtigsten:

1. Breitensuche
2. Tiefensuche

32.2.2 Traversierung I: Breitensuche

Die Idee der Breitensuche.

32-8

Ziel ist es, alle von s aus erreichbaren Knoten zu besuchen.

- Wir beginnen beim Startknoten s. (Tiefe 0)
- Dann besuchen wir alle seine Nachbarn. (Tiefe 1)
- Dann besuchen wir die Nachbarn der Knoten auf Tiefe 1, aber keinen Knoten, an dem wir schon waren. (Tiefe 2)
- Dann besuchen wir die Nachbarn der Knoten auf Tiefe 2, aber keinen Knoten, an dem wir schon waren. (Tiefe 3)
- Und so weiter.

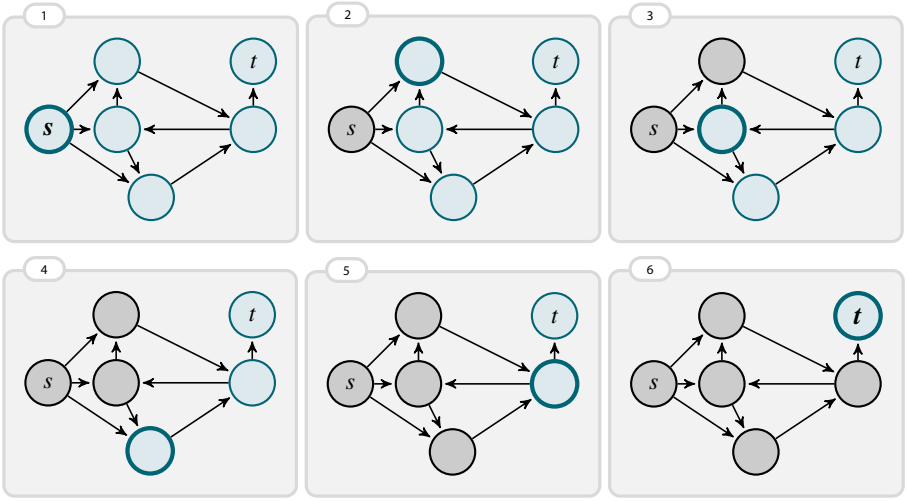
Motto der Breitensuche

Fraternité! (Geschwister zuerst!)

Beispiel einer Breitensuche.

32-9

Der aktuell besuchte Knoten ist **fett**. Schon besuchte Knoten sind dunkel.



Zur Implementation einer Breitensuche.

32-10

Bei der Implementation einer Breitensuche stellen sich folgende Probleme:

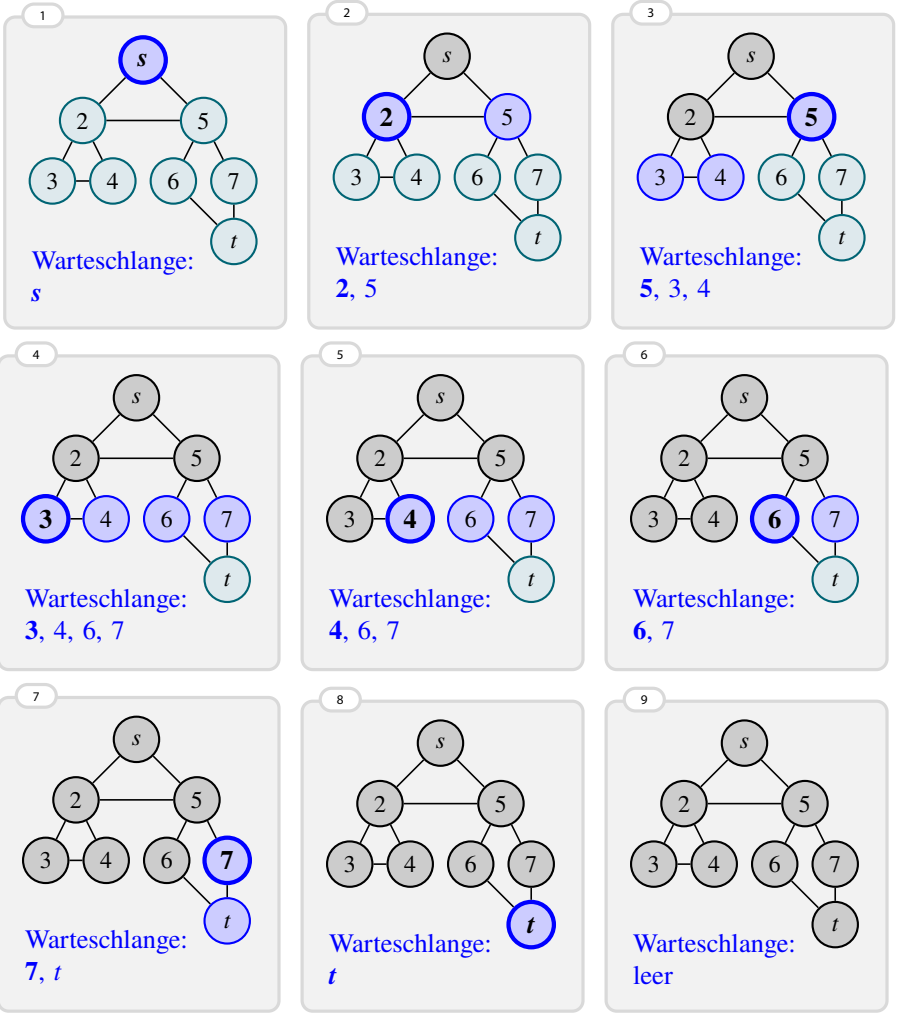
1. Wie merkt man sich, welche Knoten schon besucht wurden?
2. Woher weiß man, welcher Knoten als nächstes kommt?

Lösungen:

1. Man führt einen Array boolescher Werte mit, der für jeden Knoten angibt, ob man dort schon war.
2. Man überlegt sich, dass Breitensuche folgender Regel entspricht:
   »Besuche den ersten Knoten in einer Liste noch zu besuchender Knoten. Füge dann die noch nicht besuchten Nachbarn dieses ersten Knotens ans Ende der Liste an.«

32-11

Beispiel der Warteschlange in einer Breitensuche.



Grünblau = noch nicht besucht, dunkel = schon besucht, blau = in der Warteschlange, **fett** = gerade besucht

32.2.3 Traversierung II: Tiefensuche

32-12

Die Idee hinter der Tiefensuche.

Ziel ist es wieder, alle von s erreichbare Knoten zu besuchen. Diesmal wird der Graph aber nicht »schichtenweise« durchsucht. Man versucht vielmehr, möglichst schnell »in die Tiefe zu kommen«.

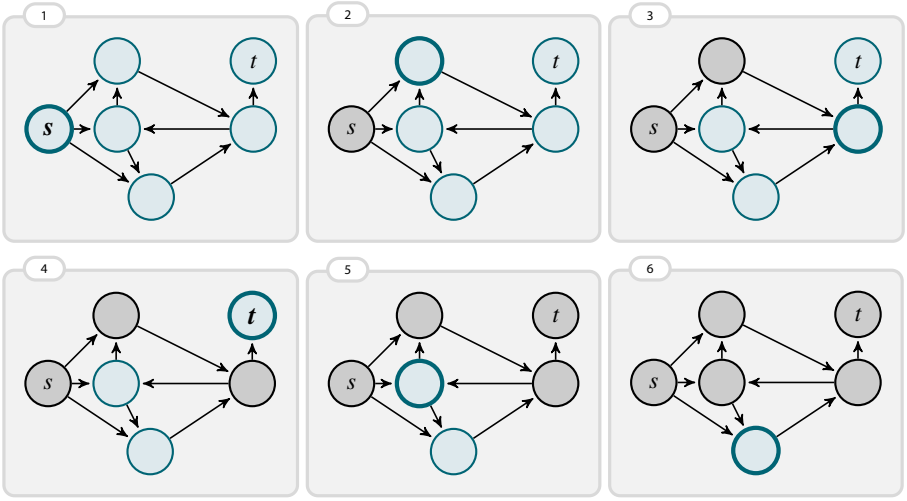
Motto der Tiefensuche

Vive les enfants! (Kinder zuerst.)

Beispiel einer Tiefensuche.

32-13

Der aktuell besuchte Knoten ist **fett**. Schon besuchte Knoten sind dunkel.



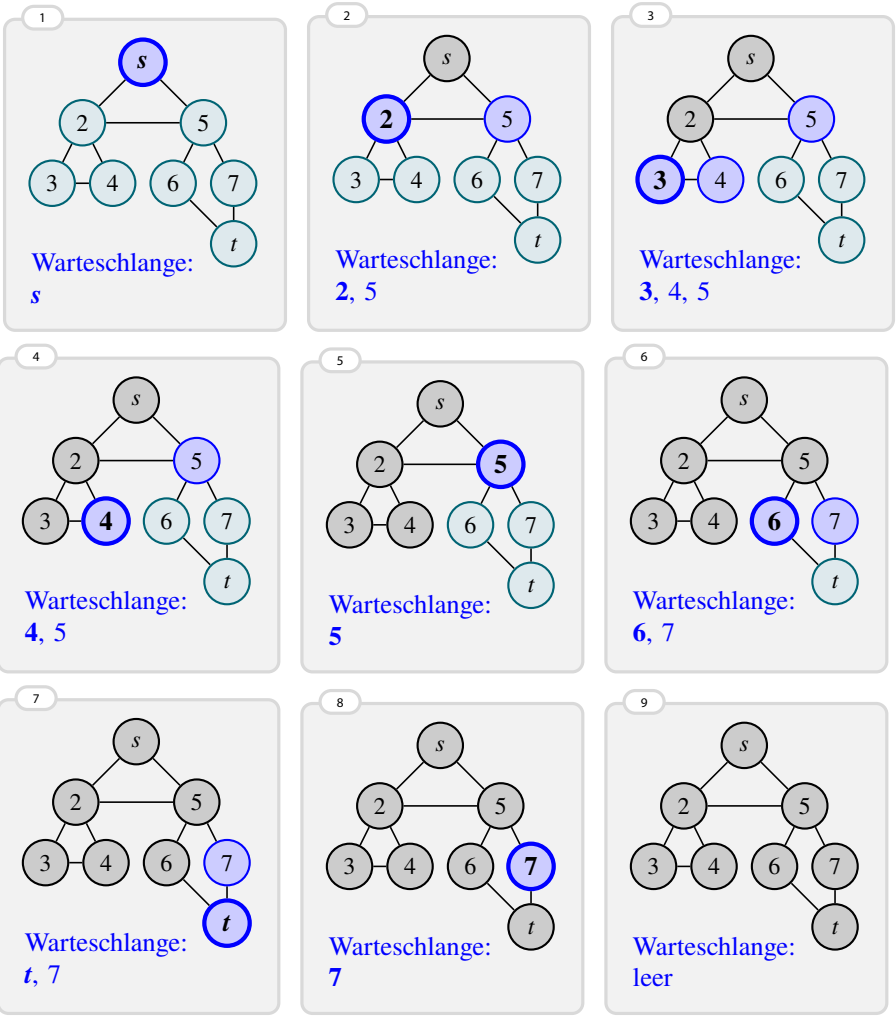
Zur Implementation einer Tiefensuche.

32-14

Die Regel ist fast gleich:
»Besuche den ersten Knoten in einer Liste noch zu besuchender Knoten. Füge dann die noch nicht besuchten Nachbarn dieses ersten Knotens *an den Anfang* der Liste an.«

Beispiel der Warteschlange in einer Tiefensuche.

32-15

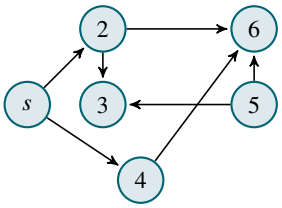


Grünblau = noch nicht besucht, dunkel = schon besucht, blau = in der Warteschlange, **fett** = gerade besucht

32-16

✎ Zur Übung

In welcher Reihenfolge werden die Knoten des folgenden Graphen bei einer Tiefensuche und in welcher bei einer Breitensuche, jeweils beginnend bei Knoten s, durchlaufen?



32.2.4 Vergleich

32-17

Vergleich Breiten- und Tiefensuche

Breiten- und Tiefensuche werden ähnlich implementiert. Der einzige Unterschied liegt darin, wo die noch nicht besuchten Nachbarn in die Warteschlage eingefügt werden. Bei einer *Breitensuche* werden sie *hinten* angefügt. (Man spricht von »fifo« – first in, first out.) Bei einer *Tiefensuche* werden sie *vorne* angefügt. (Man spricht von »lifo« – last in, first out.)

32.3 Kürzeste Wege

32.3.1 Problemstellung

32-18

Das Kürzeste-Wege-Problem

Beispiel: Kürzeste-Wege-Problem

Eingabe Ein gerichteter oder ungerichteter Graph eventuell mit *positiven* Kantengewichten.
Zusätzlich zwei Knoten s (source) und t (target).

Ausgabe Ein kürzester Weg von s nach t (die Summe der Gewichte soll minimal sein).

Dieses Problem muss gelöst werden von einem Navigationssystem, von einer Telefongesellschaft oder auch von einem Internet-Anbieter.

32.3.2 Lösung I: Breitensuche

32-19

Breitensuche löst das Kürzeste-Wege-Problem in Graphen ohne Kantengewichte.

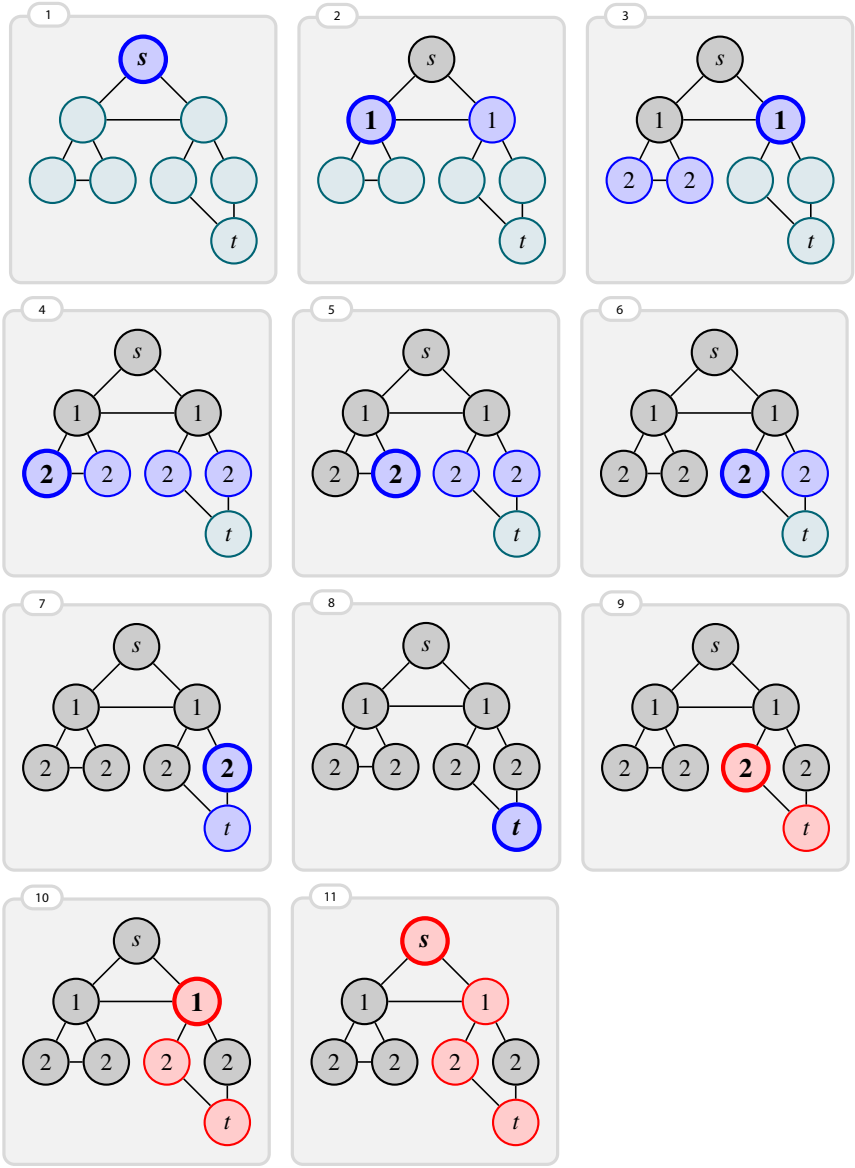
Beobachtung

Die Breitensuche besucht zuerst die Knoten mit Entfernung 1 von s, dann die Knoten mit Entfernung 2 von s, dann die Knoten mit Entfernung 3 von s und so weiter.

Algorithmus für kürzeste Wege in ungewichteten Graphen

1. Führe eine Breitensuche ausgehend von s durch.
2. Speichere dabei in jedem Knoten nicht nur, ob er besucht wurde, sondern auch seine Entfernung von s.
3. Der gesuchte Weg von s nach t ergibt sich am Ende, indem man von t immer zum Nachbarknoten mit der um 1 kleineren Entfernung geht.

Beispielablauf des Algorithmus. 32-20

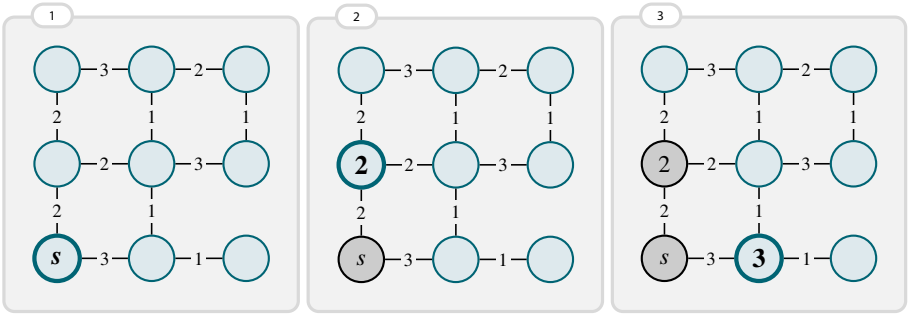


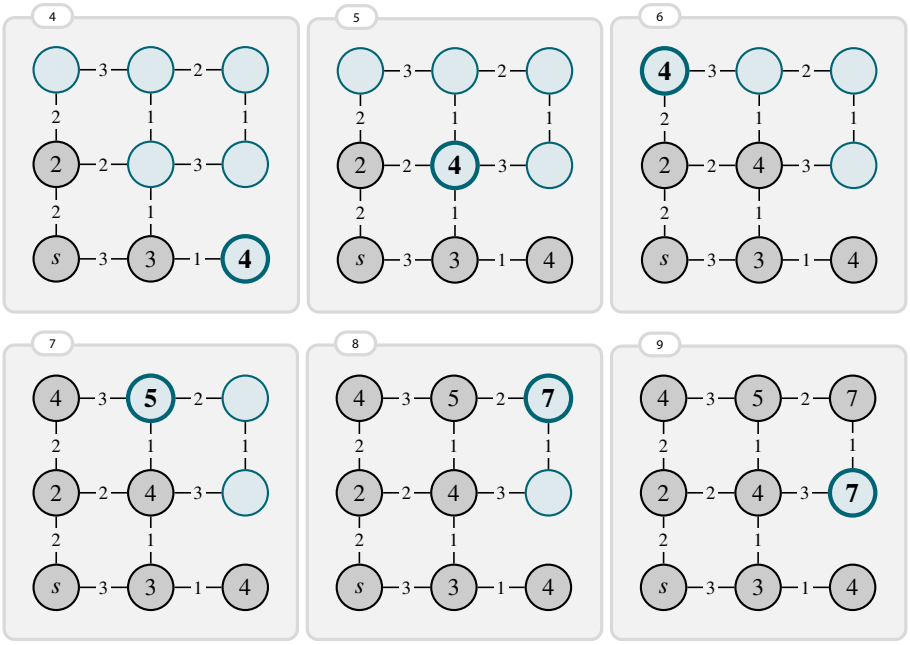
32.3.3 Lösung II: Dijkstra-Algorithmus

Die Grundidee von Dijkstras Algorithmus. 32-21

Dijkstras Algorithmus ist eine Art *verallgemeinerte Breitensuche, die auch Kantengewichte zulässt*. Der Algorithmus besucht ebenfalls alle von s erreichbare Knoten. Er speichert ebenfalls für jeden schon besuchten Knoten seine *Entfernung vom Startknoten*. Neu ist: In jedem Schritt besuchen wir als nächstes den Knoten, der durch eine Kante so mit einem schon besuchten Knoten verbunden werden kann, dass die *Entfernung zu s minimal* ist.

Der Dijkstra-Algorithmus anhand eines Beispiels. 32-22





32-23

Zur Laufzeit des Dijkstra-Algorithmus.

Sei n die Anzahl der Knoten und m die Anzahl der Kanten.

- Naive Implementation:
  Man muss n mal den nächsten Knoten finden, was jeweils n^2 lang dauert, insgesamt also $O(n^3)$.
- Nicht ganz so naive Implementation:
  Man merkt sich für jeden noch nicht besuchten Knoten den Abstand von s, wenn man ihn durch eine Kante zu den bereits besuchten verbinden würde. Dann kann man immer das Minimum dieser Liste nehmen.
  Man bekommt eine Laufzeit von $O(n^2)$ hin.
- Clevere Implementation:
  Man benutzt eine neue Datenstruktur, genannt Heap. Dies liefert eine Laufzeit von $O\big((n+m)\log n\big)$.
- Ganz, ganz clevere Implementation:
  Man benutzt Fibbonacci-Heaps (eine ziemlich komplexe Datenstruktur). Dies liefert eine Laufzeit von $O(n\log n+m)$.

32.4 Längste Wege

32.4.1 Problemstellung

32-24

Das Längste-Wege-Problem

Beispiel: Längste-Wege-Problem

Eingabe Ein gerichteter oder ungerichteter Graph.
Zusätzlich zwei Knoten s (source) und t (target).

Ausgabe Ein längster Weg von s nach t (die Summe der Gewichte soll maximal sein).

Dieses Problem muss gelöst werden

- bei der Berechnung *kritischer Pfade*,
- als Vorstufe des allgemeinen Handlungsreisendenproblems.

32.4.2 Lösung: Backtracking

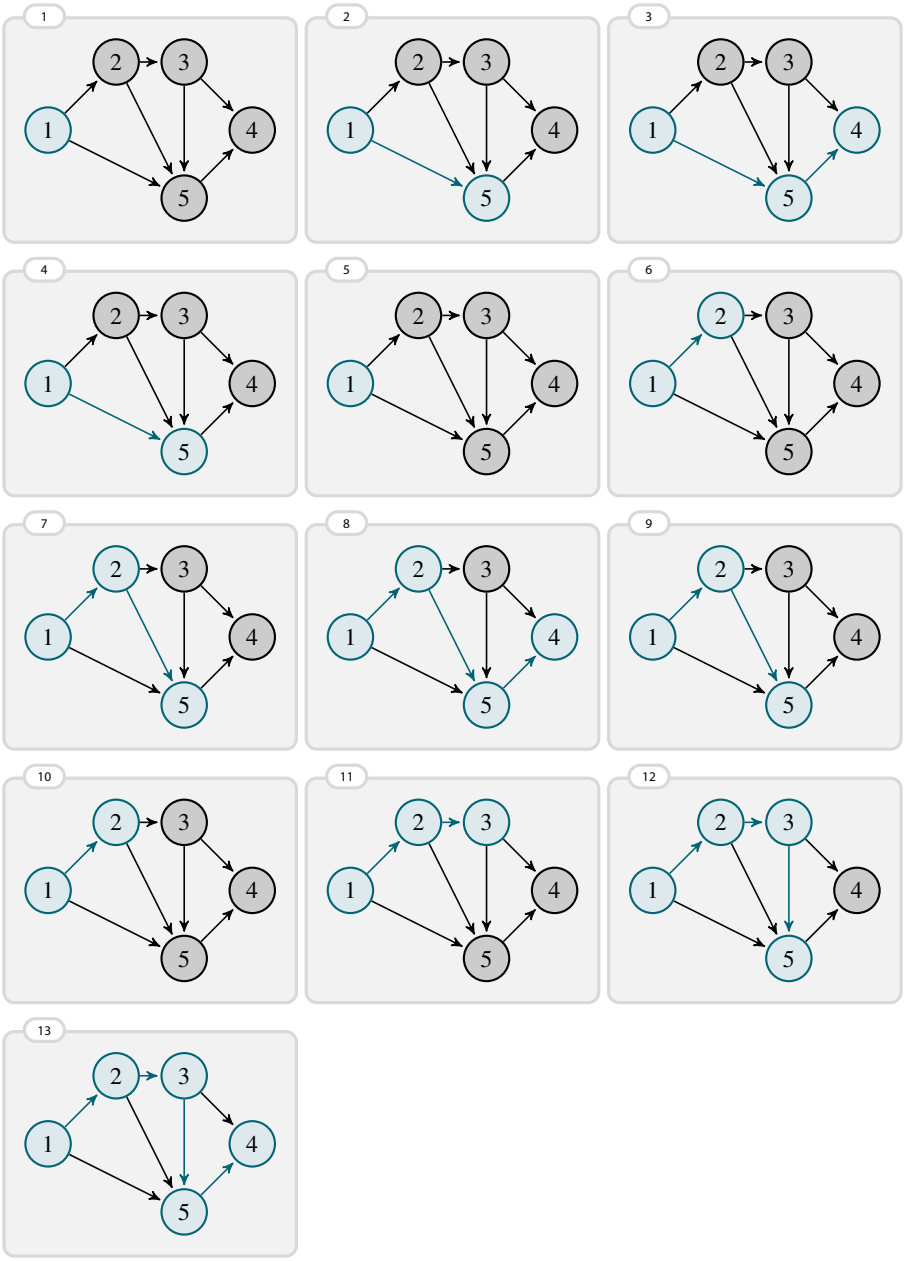
Lösung durch Backtracking

32-25

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.



Zusammenfassung dieses Kapitels

32-26

Wege-Finden-Probleme
Bei *Wege-Finden-Probleme* hat man einen Graphen und zwei Knoten gegeben. Das Ziel ist dann, einen möglichst kurzen oder möglichst langen Weg zwischen den Knoten zu finden. Bei gewichteten Graphen ist die »Länge« des Weges die Summe der Kantengewichte.

Erreichbarkeit
Beim *Erreichbarkeitsproblem* überprüft man, ob *überhaupt ein Weg zwischen zwei Knoten existiert*. Man löst es mittels

- Breitensuchen: besuche alle Knoten in aufsteigender Reihenfolge ihrer Entfernung; oder
- Tiefensuche: besuche immer möglichst schnell Knoten »in der Tiefe« des Graphen.

Beide Verfahren werden sehr ähnlich implementiert.

Dijkstra-Algorithmus zum Finden kürzester Wege
Speichere für jeden schon besuchten Knoten k die »Entfernung« $d(s,k)$ von s bis k. Solange noch nicht alle Knoten besucht sind, wiederhole:

1. Finde die Kante e, die einen schon besuchten Knoten k mit einem noch nicht besuchten v verbindet, so dass $d(s,k)+w(e)$ minimal ist, wobei $w(e)$ das Gewicht der Kante ist.
2. Markiere u als besucht und speichere dort $d(s,k)+w(e)$.

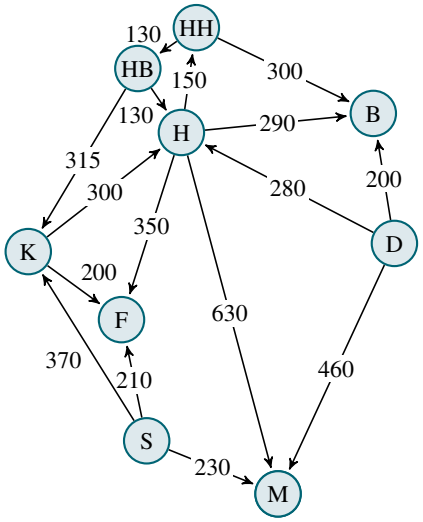
Längste-Wege-Finden
Längste Wege finden ist *schwierig* und muss mittels *Backtracking* gelöst werden.

Übungen zu diesem Kapitel

Übung 32.1 Breiten- und Tiefensuche in Java implementieren, mittel

Betrachten Sie wieder die Klasse `GraphAsAdjacencyMatrix` aus Kapitel 30 (in der gerichteten Variante aus des Kapitels, nicht in der ungerichteten aus Übung 30.2). Erweitern Sie diese Klasse um eine Methode `boolean depthFirstSearch( int start, int target )` und eine Methode `boolean breadthFirstSearch( int start, int target )`, die ausgehend vom Knoten mit der Nummer `start` per Tiefen- bzw. Breitensuche den Knoten `target` suchen. Die Methoden sollen genau dann `true` zurückliefern, wenn die beiden gegebenen Knoten durch einen Weg im Graphen verbunden sind.

Hinweis: Verwenden Sie bei der Implementierung für die Warteschlange mit den noch zu besuchenden Knoten einfach einen Array – Sie wissen ja, wie lang die Warteschlange maximal werden kann. Verwenden Sie ein weiteres Array, um sich schon besuchte Knoten zu merken. Der Unterschied zwischen Breiten- und Tiefensuche liegt dann darin, auf welche Weise das Warteschlangenarray auf- und abgebaut wird. Zum leichteren Verständnis empfiehlt es sich, die Tiefensuche zuerst zu implementieren. Testen Sie Ihre Methode anhand verschiedener erreichbarer und nicht erreichbarer Knotenpaare aus folgendem Graphen:



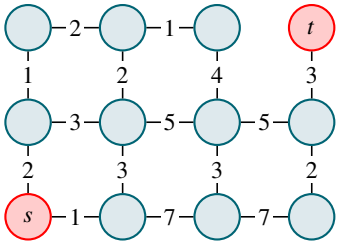
Diesen können Sie, genau wie in Übung 30.2, mit folgendem Java-Code erzeugen:

```
GraphAsAdjacencyMatrix g = new GraphAsAdjacencyMatrix(10);
g.addEdge(0,1,130); // HH -> HB
g.addEdge(3,0,150); // H -> HH
g.addEdge(1,3,130); // HB -> H
g.addEdge(3,4,290); // H -> B
g.addEdge(3,6,630); // H -> M
g.addEdge(7,6,230); // S -> M
g.addEdge(5,3,280); // D -> H
g.addEdge(5,4,200); // D -> B
g.addEdge(5,6,460); // D -> M
g.addEdge(7,8,210); // S -> F
g.addEdge(9,3,300); // K -> H
g.addEdge(1,9,315); // HB -> K
g.addEdge(0,4,300); // HH -> B
g.addEdge(9,8,200); // K -> F
g.addEdge(3,8,350); // H -> F
g.addEdge(7,9,370); // S -> K
```

Prüfungsaufgaben zu diesem Kapitel

Übung 32.2 Dijkstra-Algorithmus anwenden, leicht, original Klausuraufgabe, mit Lösung

Wenden Sie den Algorithmus von Dijkstra auf den folgenden ungerichteten Graphen an, um den kürzesten Weg von s nach t zu finden. Tragen Sie dafür die Werte, die der Algorithmus für die einzelnen Knoten speichert, in die Abbildung ein und markieren Sie dann den kürzesten Weg.



33-1

Kapitel 33
Approximationsalgorithmen
Wie faltet sich ein Protein?

33-2

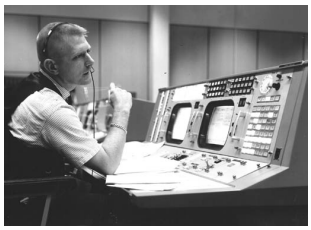
Lernziele dieses Kapitels

1. Die Konzepte Optimierungsproblem, Lösung und Maße verstehen
2. Das Konzept der approximativen Lösung verstehen
3. Approximationsalgorithmen für das Handlungsreisenden-Problem und das Bin-Packing-Problem kennen

Inhalte dieses Kapitels

33.1	Optimierungsprobleme	287
33.1.1	Einführung	287
33.1.2	Formalisierung	287
33.1.3	Beispiele	287
33.1.4	Maß und Güte	288
33.2	Approximationsalgorithmen	288
33.2.1	Heuristiken und Approximation	288
33.2.2	Handlungsreisenden-Problem	289
33.2.3	Bin-Packing	290
	Übungen zu diesem Kapitel	291

Worum es heute geht



Copyright by NASA.

Während der Apollo-13-Mission kam es zu einem folgendschweren Unfall. Infolge dessen hatte die Mannschaft der Raumfähre noch für 48 Stunden Luft und für etwa 12 Stunden Strom – bei einer Rückreisezeit von 72 Stunden. Der Mission-Controller Gene Granz kommentierte dies mit den Worten: »Failure is not an option.«

In den vorigen Kapiteln haben wir einige Probleme kennengelernt, bei denen auch niemand weiß, wie man sie lösen soll. So kennt man beispielsweise kein effizientes Verfahren für das Handlungsreisenden-Problem und ebensowenig für das Bin-Packing-Problem. Wir haben zwar Backtracking kennen gelernt, aber dieses Verfahren ist für realistische Eingabegrößen oft unbrauchbar. Man könnte deshalb auf die Idee kommen, dass »es eben einfach nicht geht« und sich um andere Dinge kümmern. In vielen Anwendungen gilt jedoch »Failure is not an option« und wir werden uns irgend etwas ausdenken müssen, um diese Probleme doch zu lösen. Genau darum soll es in diesem Kapitel gehen.

Der Trick ist, sich mit nicht ganz (aber fast) optimalen Lösungen zu begnügen. Wenn Sie in einer Klausur nur 98% der geforderten Leistung erbringen, dann genügt das vollkommen; weshalb sollten wir von Computern mehr erwarten als von Menschen? Wenn man »fast-optimale« Lösungen sucht, dann wird manch schwieriges Problem plötzlich ganz einfach; bei solchen Problemen steckt also die Schwierigkeit einzig darin, das letzte Quäntchen Optimalität herauszupressen. Die Dinge liegen hier ähnlich wie bei einer gute gestellten Klausur: Mit recht wenig Arbeitsaufwand wird man doch wohl bestehen, will man ihn dann aber immer näher an die »volle Punktzahl« heran, so steigen die »Vorbereitungskosten« irgendwann in astronomische Höhen.

In diesem Kapitel werden zwei einfache Approximationsalgorithmen für einmal das Handlungsreisenden-Problem und einmal für das Bin-Packing-Problem vorgestellt. Für das Bin-Packing-Problem gibt es sogar noch viel bessere Algorithmen, hier kann man tatsächlich mit etwas Getrickse beliebig dicht an das Optimum herankommen; der Aufwand steigt nur entsprechend an.

Es soll aber nicht verschwiegen werden, dass nicht alle Probleme approximierbar sind. Dies ist beispielsweise für das Färbeproblem der Fall: Man kennt keinen effizienten Algorithmus,

um beliebige Graphen mit einer minimalen Anzahl an Farben zu färben (so dass benachbarte Knoten unterschiedliche Knoten haben); man kennt noch nicht einmal einen Algorithmus, der einen beliebigen Graphen mit, sagen wir, einer Million mal so vielen Farben wie unbedingt nötigt färbt.

33.1 Optimierungsprobleme

33.1.1 Einführung

Optimierungsprobleme

33-4

▶ Definition: Optimierungsprobleme
Ein *Optimierungsproblem* ist ein Problem, bei dem es zu einer *Eingabe*

- viele mögliche *Lösungen* geben kann (eventuell auch keine),
- jede Lösung ein gewisses *Maß* hat (eine Zahl) und
- man eine Lösung sucht, deren Maß möglichst groß oder klein ist.

Achtung
Die Begriffe in dieser Definition sind alle wichtig und man muss sie sauber unterscheiden.

Ziele bei Optimierungsproblemen.

33-5

Sei ein Optimierungsproblem gegeben. Dann kann man mehrere Ziele haben (in absteigender Schwierigkeitsreihenfolge):

1. Für eine gegebene Eingabe möchte man eine *optimale Lösung* finden (also eine Lösung, deren Maß möglichst klein oder groß ist).
2. Für eine gegebene Eingabe möchte man eine *möglichst gute, aber vielleicht nicht optimale Lösung* finden.
3. Für eine gegebene Eingabe möchte man *überhaupt irgendeine Lösung finden*.
4. Für eine gegebene Eingabe möchte man *entscheiden, ob es eine Lösung gibt*.

33.1.2 Formalisierung

Die Formalisierung von Optimierungsproblemen

33-6

▶ Definition: Formales Optimierungsproblem
Ein *Optimierungsproblem* besteht dann aus:

1. Einer Lösungsrelation S. Ist ein Paar (i,s) in dieser Menge, so ist s eine Lösung zu der Eingabe i.
2. Einer Maßfunktion. Dies ist eine Funktion $m\colon S \to \mathbb{N}$, die jeder Lösung ein Maß zuordnet.
3. Einem Typ. Dieser ist entweder »Maximierung« oder »Minimierung«.

33.1.3 Beispiele

Das Optimierungsproblem »Münzrückgabe«.

33-7

Das Optimierungsproblem

Eingabe	Ein (Multi-)Menge an Münzen und ein Wert w.
Lösungen	Teilmenge der Münzen, deren Summe w ist.
Maß	Anzahl der Münzen.
Ziel	Minimierung (man will möglichst wenig Kleingeld).

✎ Zur Übung

33-8

Formulieren Sie das Bin-Packing-Problem als Optimierungsproblem, geben Sie also die Eingaben, die Lösungen, das Maß und das Ziel explizit an.

33.1.4 Maß und Güte

33-9

Gute versus schlechte Lösungen.

▶ **Definition**

Sei P ein Optimierungsproblem. Dann ist

1. das *optimale Maß* zur Eingabe das minimale (oder maximale) Maß aller Lösungen zu x,
2. die *Güte* einer konkreten Lösung s zu x das Verhältnis

$$\frac{\text{Maß von } s}{\text{optimales Maß einer Lösung zu } x}.$$

Bei Maximierungsproblemen ist die Güte gerade der Kehrwert, so dass sie immer mindestens 1 ist.

Merke

Das *Maß* einer Lösung gibt ihre *»absoluten Kosten«* an. Die *Güte* einer Lösung gibt an, *um welchen Faktor* sie schlechter ist als eine optimale Lösung.

33-10

✎ **Zur Übung**

Beim Münzrückgabeproblem sollen 1,86 Euro zurückgegeben werden, folgende Münzen stehen zur Verfügung :

Ein Algorithmus liefert als Lösung zweimal 50 Cent, dreimal 20 Cent, zweimal 10 Cent und dreimal 6 Cent. Berechnen sie das Maß und die Güte dieser Lösung.

33.2 Approximationsalgorithmen

33.2.1 Heuristiken und Approximation

33-11

Was tun, wenn man kein effizientes Lösungsverfahren kennt?

Für viele Optimierungsprobleme *kennt man kein effizientes Lösungsverfahren*. Man kann dann probieren, eine *Heuristik* für das Problem zu finden. Eine solche liefert *eine Lösung* für das Problem, die im Allgemeinen recht gut ist (ihr Maß ist nahe am Optimum und ihre Güte damit nahe bei 1). Eine Heuristik *kann* aber auch (hoffentlich selten) sehr schlechte oder gar keine Lösungen liefern.

Beispiel

Optimierungsprobleme, für die man nur Heuristiken kennt:

- Handlungsreisenden-Problem,
- Protein-Faltung,
- DNA-Fragment-Assembly.

33-12

Approximationen – die besseren Heuristiken.

Bei allgemeinen Heuristiken ist es unschön, dass sie sehr schlechte Lösungen liefern können. *Approximationsverfahren* sind Heuristiken, die eine *garantierte Güte* haben. Hat ein Verfahren beispielsweise Güte 3, so bedeutet dies, dass das Verfahren immer *Lösungen liefert, deren Maß höchstens dreimal so groß ist wie das Maß des Optimums.*

Beispiel

Optimierungsprobleme, die sich approximieren lassen:

- Handlungsreisenden-Problem,
- Bin-Packing.

33-13

Definition von Approximationsalgorithmen.

▶ **Definition**

Sei P ein Optimierungsproblem und $r > 1$ eine rationale Zahl. Ein *r-Approximationsalgorithmus für P* ist ein Algorithmus, der

- zu jeder Eingabe x, zu der es eine Lösung gibt,
- eine Lösung ausrechnet, deren Güte kleiner oder gleich r ist.

33.2.2 Handlungsreisenden-Problem

Zur Erinnerung: Das Handlungsreisenden-Problem

33-14

Euklidisches Handlungsreisenden-Problem

Eingaben Ein Menge von Punkten in der Ebene.

Lösungen Rundreise (Folge der Punkte, die jeden Punkt genau einmal enthält).

Maß Summe der Länge der Strecken entlang der Rundreise.

Ziel Minimierung.

Allgemeines Handlungsreisenden-Problem

Eingaben Ein kantengewichteter Graph.

Lösungen Rundreise (Folge von miteinander verbundenen Knoten, die jeden Knoten genau einmal enthält).

Maß Summe der Gewichte der Kanten entlang der Rundreise.

Ziel Minimierung.

Ein Approximationsalgorithmus für das euklidische Handlungsreisenden-Problem.

33-15

Man kennt *keinen effizienten Algorithmus* für das Handlungsreisenden-Problem. Man kennt aber viele gute *Approximations-Algorithmen*.

Algorithmus für das euklidische Handlungsreisenden-Problem

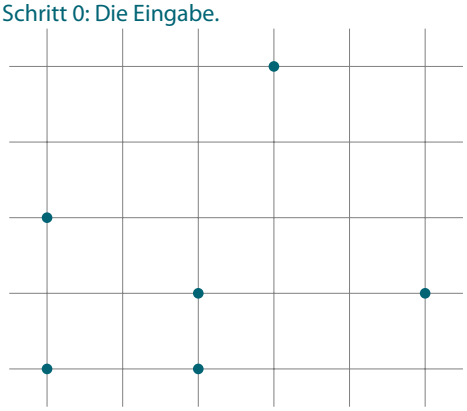
Eingabe ist eine Menge von Städten.

1. Erzeuge einen vollständigen Graphen, dessen Knoten Städte sind und dessen Kanten mit den Entfernungen der Städte gewichtet sind.
2. Berechne ein minimales Gerüst des Graphen.
3. Erzeuge eine Eulertour aus dem Gerüst.
4. Verkürze die Eulertour, bis jeder Knoten nur einmal besucht wird.

Der Algorithmus an einem Beispiel.

33-16

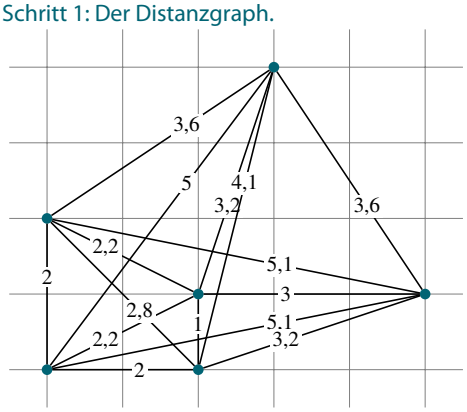
Schritt 0: Die Eingabe.



Der Algorithmus an einem Beispiel.

33-17

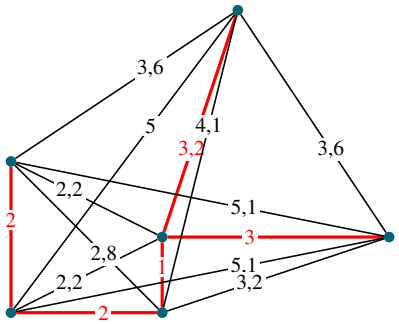
Schritt 1: Der Distanzgraph.



33-18

Der Algorithmus an einem Beispiel.

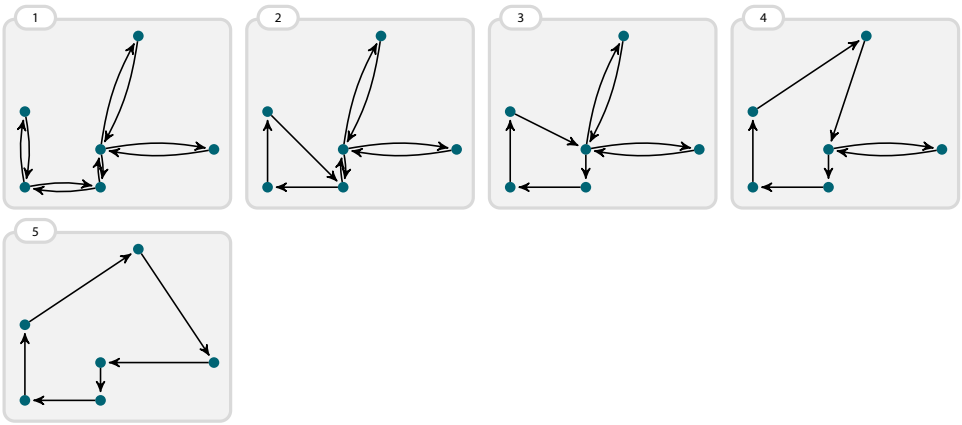
Schritt 2: Das Gerüst.



33-19

Der Algorithmus an einem Beispiel.

Schritte 3 und 4: Geradeziehen der Eulertour.



33-20

Der Algorithmus liefert eine 2-Approximation.

▶ Satz

Der Algorithmus liefert immer eine Rundreise, die höchstens doppelt so lang ist wie die kürzeste.

Beweis.

1. Die kürzeste Rundreise habe die Länge x.
2. Löschen wir daraus eine Kante, so erhalten wir einen Pfad, der noch kürzer ist.
3. Dieser Pfad ist ein Gerüst. Also ist das Gewicht y des minimalen Gerüsts noch kleiner: $y < x$.
4. Die Eulertour hat Länge $2y < 2x$.
5. Die ausgegebene Tour ist kürzer als die Eulertour, also kürzer als $2x$. □

33.2.3 Bin-Packing

33-21

Das Bin-Packing-Problem

Das formale Bin-Packing-Problem.

Eingabe Eine Liste von Objektgrößen $(g_1, \ldots, g_n)$ und eine Eimergröße b.

Ausgaben Zuordnung von Objekten zu Eimern, so dass die Summe der Größen aller Objekte, die demselben Eimer zugeordnet sind, maximal b ist.

Maß Anzahl der benutzten Eimer.

Ziel Minimierung.

Wiederholung: Ein Greedy-Algorithmus für Bin-Packing.

33-22

Der First-Fit-Algorithmus

Für jeden Gegenstand tue folgendes:

1. Finde, von links beginnend, den ersten Eimer, in den der Gegenstand noch passt.
2. Platziere den Gegenstand in diesen Eimer.

Wie gut ist First-Fit?

33-23

▶ **Satz**

Der First-Fit-Algorithmus ist ein 2-Approximationsalgorithmus für Bin-Packing.

Beweis. Betrachten wir eine Lösung, die First-Fit produziert hat. Dann *passt der Inhalt von je zwei* nebeneinander stehenden Eimer *nicht* in einen einzigen Eimer (sonst hätte First-Fit das nämlich getan). Also muss auch in einer optimalen Lösung für je zwei von First-Fit benutzte Eimer mindestens ein Eimer genutzt werden. □

Zusammenfassung dieses Kapitels

▶ **Optimierungsprobleme bestehen aus:**

33-24

1. Einer Lösungsrelation S. Ist ein Paar (i,s) in dieser Menge, so ist s eine Lösung zu der Eingabe i.
2. Einer Maßfunktion. Dies ist eine Funktion $m\colon S \to \mathbb{N}$, die jeder Lösung ein Maß zuordnet.
3. Einem Typ. Dieser ist entweder »Maximierung« oder »Minimierung«.

▶ **r-Approximationsalgorithmus**

Der Algorithmus gibt

- zu jeder Eingabe x, zu der es eine Lösung gibt,
- eine Lösung aus, deren Güte kleiner oder gleich r ist.

▶ **Beispiele approximierbarer Probleme**

Das euklidische Handlungsreisenden-Problem und das Bin-Packing-Problem haben beide 2-Approximationsalgorithmen.

Übungen zu diesem Kapitel

Übung 33.1 Optimierungsprobleme, mittel

Formulieren Sie jedes der folgenden Probleme als ein Optimierungsproblem. Geben Sie dazu die erlaubten Eingaben, die Lösungen, das Maß und das Ziel an:

1. Bei der *Linearen Programmierung* ist eine mathematische Funktion und ein Polygon (beziehungsweise bei mehr als zwei Dimensionen ein *Polytop*) gegeben. Man möchte nun den Punkt des Polygons ermitteln, auf dem die Funktion den geringsten Wert liefert.
2. Bei der *Sequenzdatenbanksuche* geht es (vereinfacht ausgedrückt) darum, aus einer Menge von Sequenzen (Datenbank) diejenige zu finden, die zu einer vorgegebenen Sequenz (Abfragesequenz) den geringsten Editierabstand hat.
3. Beim *Färbbarkeitsproblem* möchte man eine politische Weltkarte mit möglichst wenigen Farben so färben, dass keine zwei benachbarten Staaten die gleiche Farbe bekommen.
4. Bei dem Geduldsspiel *Tangram* ist ein Quadrat in sieben Plättchen in einfachen Formen (genauer gesagt zwei große Dreiecke, ein mittelgroßes Dreieck, zwei kleine Dreiecke und ein Parallelogramm) zerschnitten. Man bekommt nun eine Form vorgegeben und soll versuchen, diese mit den Plättchen nachzulegen.

Übung 33.2 Optimum und Güte nicht optimaler Lösungen berechnen, leicht

Ein Dieb bricht in die Villa eines reichen Informatikprofessors ein und möchte dort Gegenstände mit möglichst hohem Wert erbeuten. Da er bei seinen zahlreichen Raubzügen schon einen kleinen Rückenschaden abbekommen hat, kann er aber leider nicht mehr als 8 Kilo in seinem Rucksack mitnehmen. Er kann unter folgenden Gegenständen auswählen:

Gegenstand	Gewicht	Wert
Laptop	3 Kilo	800 Euro
PC	6 Kilo	1400 Euro
Informatikbuch	2 Kilo	15 Euro
Designerhocker	3 Kilo	300 Euro

- Formulieren Sie dieses Problem als Optimierungsproblem.
- Geben Sie *alle* Lösungen dieses Problems an.
- Welches ist die optimale Lösung? Wie groß ist das Maß der optimalen Lösung?
- Wie groß sind Maß und Güte der nicht optimalen Lösungen?

Übung 33.3 Greedy-Algorithmus für das Problem des Handlungsreisenden, mittel

Eine weniger gute Strategie zur Approximation des Handlungsreisendenproblems ist, auch hier eine Greedy-Heuristik zu verwenden. Was sind bei dieser Heuristik die Teillösungen, und nach welchem Kriterium werden diese Teillösungen schrittweise vervollständigt? Geben Sie eine Probleminstanz an, bei der die Greedy-Strategie für das Handlungsreisendenproblem besonders schlecht funktioniert, und vergleichen Sie die Approximationsgüte mit dem in der Vorlesung besprochenen Verfahren!

Übung 33.4 Approximation des Matching-Problems, mittel

Beim *Matching-Problem* sind zwei Mengen gegeben, aus denen man Paare bilden kann. Dabei sind verschiedene Möglichkeiten der Paarung mit unterschiedlichen »Kompatibilitäten« gegeben (tatsächlich wird dieses Problem auch oft *Heiratsproblem* genannt). Man möchte nun die Paare so bilden, dass die Gesamtkompatibilität möglichst hoch ist.

Das Matching-Problem lässt sich wie folgt als Optimierungsproblem formalisieren:

Bipartites Matching

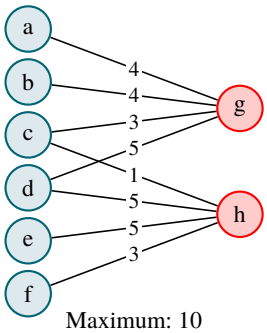
Eingaben: Ein kantengewichteter Graph, dessen Knoten in zwei Mengen unterteilt sind, so dass jeder Knoten aus einer Menge nur mit Knoten aus der anderen Menge verbunden ist. Solche Graphen nennt man *bipartit*.

Lösungen: Teilmengen der Kanten, so dass jeder Knoten von höchstens einer der Kanten berührt wird.

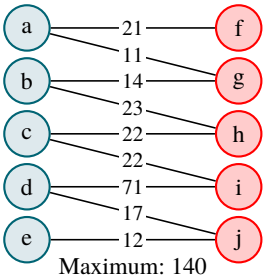
Maß: Summe der Kantengewichte.

Ziel: Maximierung.

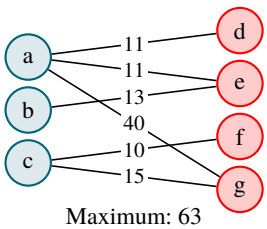
1. Geben Sie einen möglichst einfachen Algorithmus an, der Lösungen für das Matching-Problem ermittelt. Ihr Algorithmus muss nicht unbedingt immer die beste Lösung finden!
2. Wenden Sie Ihren Algorithmus auf die folgenden vier Beispiele an und ermitteln Sie die Approximationsgüte! Das Gewicht des maximalen Matchings ist jeweils mit angegeben.



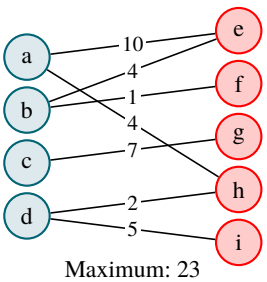
Maximum: 10



Maximum: 140



Maximum: 63



Maximum: 23

Übung 33.5 Approximationsverfahren für Partitionsproblem bewerten, mittel

Beim *Partitionsproblem* möchte man eine vorgegebene Liste natürlicher Zahlen so in zwei Unterlisten aufteilen, dass die Summen der Zahlen in den beiden Unterlisten genau gleich sind.

1. Formulieren Sie das Partitionsproblem als Optimierungsproblem!
2. Eine Greedy-Heuristik zur Approximation des Partitionsproblems funktioniert wie folgt: Sei S die Summe aller gegebenen Zahlen. Man entnimmt in jedem Schritt die größtmögliche Zahl aus der gegebenen Liste, so dass die Summe aller entnommenen Zahlen nicht größer als $S/2$ wird. Ist es nicht mehr möglich, eine weitere Zahl zu entnehmen, ohne den Wert $S/2$ zu überschreiten, bricht man ab.
   Geben Sie eine Probleminstanz an, für die diese Heuristik die optimale Lösung liefert, sowie eine Instanz, für die die optimale Lösung nicht gefunden wird! Wie ist beim zweiten Fall die Güte der approximierten Lösung?

Übung 33.6 Greedy-Heuristik in Java implementieren, schwer

Implementieren Sie die in Übung 33.5 beschriebene Greedy-Heuristik für das Partitionsproblem als Java-Methode

```
int partitioniere( int[] zahlen )
```

die als Eingabe die zu partitionierenden Zahlen enthält und die Summe der durch die Greedy-Heuristik ermittelten Approximation zurückgeben soll. Dabei dürfen Sie davon ausgehen, dass die gegebenen Zahlen aufsteigend sortiert sind.

Prüfungsaufgaben zu diesem Kapitel

Übung 33.7 Approximationsalgorithmus nachvollziehen, mittel, original Klausuraufgabe, mit Lösung

Die Studentin Sarah arbeitet nebenbei als Freiberuflerin, und zwar so erfolgreich, dass sie sich vor Projektangeboten kaum retten kann. Jedes Projekt benötigt eine bestimmte Anzahl an Tagen zur Abarbeitung und bringt einen bestimmten Gewinn ein (beides seien natürliche Zahlen). Da Sarah nur während der Sommersemesterferien arbeiten kann, möchte sie eine Untermenge der Projekte so auswählen, dass sie während dieser Zeit so viel Geld wie möglich verdient. Sie kann dabei selbst bestimmen, wann sie mit der Bearbeitung eines Projekts beginnen möchte.

Eine Greedy-Heuristik zur Lösung dieses Problems besteht darin, für jedes Projekt den *Tageslohn* auszurechnen, also den Gewinn durch die Anzahl der benötigten Tage zu teilen. Dann werden die Projekte in der Reihenfolge »höchster Tageslohn zuerst« direkt hintereinander abgearbeitet.

Angenommen, Sarah hat insgesamt 60 Tage Zeit, Projekte abzuarbeiten. Geben Sie eine Probleminstanz aus drei Projektangeboten an, bei der die beschriebene Greedy-Heuristik nicht die optimale Lösung liefert! Geben Sie weiterhin die Güte der erhaltenen Approximation bezogen auf das Maß »Gesamtgewinn« an.

Übung 33.8 Approximationsalgorithmus implementieren, mittel, original Klausuraufgabe, mit Lösung

Diese Aufgabe bezieht sich auf das in Übung 33.7 erläuterte Problem der »Gewinnmaximierung«. Vervollständigen Sie die unten angegebene Java-Methode `greedyGewinn`, so dass der durch die Greedy-Heuristik ermittelte Gewinn zurückgegeben wird.

Die Methode `greedyGewinn` erhält als Parameter ein Array `tageslohn`, wobei `tageslohn[i]` der Tageslohn des i-ten Projekts ist, sowie ein Array `tage`, wobei `tage[i]` die Anzahl der Arbeitstage des `i`-ten Projekts ist. Der Parameter `arbeitstage_gesamt` gibt an, wie viele Arbeitstage insgesamt zur Bearbeitung von Projekten zur Verfügung stehen. Sie dürfen bei der Implementierung davon ausgehen, dass die Werte in `tageslohn` absteigend geordnet vorliegen.

```
int greedyGewinn( int[] tageslohn, int[] tage, int arbeitstage_gesamt ){
   int gewinn = 0;
   int verbrauchte_tage = 0;

   /* Fügen Sie hier Ihren Code ein */

   return gewinn;
}
```

34-1

Kapitel 34
Berechenbarkeit
Möglichkeiten und Grenzen von Computern

34-2

Lernziele dieses Kapitels

1. Ein mathematisches Modell für Computer kennen
2. Begriff der Berechenbarkeit verstehen
3. An Beispiele verstehen, dass Computer nicht alles berechnen können
4. Aussage der Church-Turing-These kennen

Inhalte dieses Kapitels

34.1 Die Turing-Maschine 295
34.1.1 Was bedeutet »berechenbar«? 295
34.1.2 Turings Ideen 295

34.2 Die Church-Turing-These 297
34.2.1 Historischer Rückblick 297
34.2.2 Die These 299

34.3 Nichtberechenbares 300
34.3.1 Das Postsche Korrespondenzproblem . . 300
34.3.2 Das Busy-Beaver-Problem 301
34.3.3 Kolmogorov-Komplexität 302

Worum es heute geht

In diesem letzten Kapitel zum Thema »schwierige Probleme« soll es um »wirklich schwierige Probleme« gehen, konkret um Probleme, die so schwierig sind, dass man sie *überhaupt nicht* mit Computern lösen kann. Dazu werden wir drei Fragen angehen:

1. Was heißt eigentlich »mit einem Computer ein Problem lösen«?
   Dass Ihr Smartphone nicht alle Probleme lösen kann, ist klar (es hat ja zum Beispiel nur begrenzten Speicher), aber wenn es um die *prinzipielle* Frage geht, was man berechnen kann, so ist dies erstmal nicht klar.
2. Gibt es viele verschiedene Arten von »Berechenbarkeit« oder nur eine? Wenn es nur eine gibt, warum? Wenn es viele gibt, wie verhalten die sich zueinander?
3. Gibt es tatsächlich praktisch relevante Probleme, die man nicht berechnen kann?

Betreffend die letzte Frage werden Studierende der Informatik gerne mit dem Halteproblem gequält, von dem in einem mehr als verwirrlichen Argument gezeigt wird, dass es nicht berechenbar ist. (Fragen Sie einen beliebigen Informatik-Studenten. Er wird ihnen entweder bestätigen, dass er den Beweis nicht ganz verstanden hat, oder er will angeben.) Auf komplexe Diagonalisierungsbeweise zum Halteproblem verzichte ich in diesem Kapitel. Stattdessen schauen wir uns lieber einige besonders putzige Exemplare von nichtberechenbaren Sprachen an.

Den Anfang macht das Postsche Korrespondenzproblem (das PKP), das, wie der Namen schon sagt, auf Emil Post zurückgeht. Dieses Problem ist deshalb so interessant, weil es so einfach aussieht – wenn man es zum ersten Mal sieht, würde man nie auf die Idee kommen, dass es sich um ein unentscheidbares Problem handelt. (Es soll auch schon Assistenten an Universitäten gegeben haben, die Studierenden zur Übung die Aufgabe gestellt haben, ein Programm zur Lösung des PKP zu schreiben.) Es geht einfach nur um eine Reihe von Paaren von Wörtern und wie man diese aneinanderreihen könnte.

Am Ende geht es auch noch um Super-Komprimierer, vornehm *Kolmogorov-Komprimierer* genannt. Eine mit einem Kolmogorov-Komprimierer komprimierte Zeichenkette verhält sich zu einer mit `bzip2` komprimierten in Bezug auf ihre Packungsdichte etwa wie ein

Neutronen-Stern zu einem interstellaren Nebel. Mit anderen Worten: Kolmogorov-Komprimierer sind die besten Komprimierer, die es überhaupt geben kann. Da ist es ausgesprochen schade, dass sie leider nicht berechenbar sind.

34.1 Die Turing-Maschine

34.1.1 Was bedeutet »berechenbar«?

Auf dem zum Begriff des »Berechenbaren«.

34-4

Public domain

34-5

Um 1900 erschien dem großen Mathematiker *David Hilbert* die Zeit reif, eine Vision zu verwirklichen. Sein Ziel war es, nicht mehr für jeden mathematischen Satz *mühselig einen Beweis zu ergrübeln,* er wollte *den Beweis einfach »berechnen«.* Gesucht war also eine Art »Verfahren«, bei dem man mit dem Satz begann und dann den Beweis als Resultat erhielt (oder einen Beweis für das Gegenteil).
Man begann also, den Begriff der »Berechenbarkeit« zu formalisieren. Wie bei Mathematikern üblich, waren diese Modell *nur für Mathematiker verständlich*. Er war völlig unklar, ob eines der vorgeschlagenen Modelle *tatsächlich* den Begriff der Berechenbarkeit adäquat fasste. Die Arbeit 1936 von Alan Turing war anders: Hier wurde erstmalig ein *maschinenbasiertes Modell* vorgeschlagen.

34.1.2 Turings Ideen

Turing machte alles anders in seiner Definition

State of the Art vor Turing

- Komplexe, undurchsichtige mathematische Kalküle
- Begründet wurden die Kalküle dadurch, dass sie »bei vielen Beispielen funktionierten«
- Rechnen mit Zahlen
- Grenzen der Mächtigkeit der Kalküle unklar

Turings Ideen

- Minimalistisches *mathematisches Modell* einer *Maschine*
- Eine Argumentation, dass *alle überhaupt denkbaren Berechnungen* durch das Modell abgedeckt werden
- Rechnen mit Zeichenketten
- Grenzen der Berechenbarkeit (recht) leicht beweisbar

Public domain

34-6

Turings Ziel

Turing wollte formal fassen, »wie ein Mathematiker eine sehr lange Berechnung durchführen würde«:

Ausgangspunkt

Eine »Berechnung« bedeutet für Turing, einen formalen Beweis zu überprüfen. Der Mathematiker hat dazu

- einen *beliebig großen Stapel an Karopapier*,
- einen *unerschöpflichen Bleistift*,
- einen *unerschöpflichen Radiergummi* und
- eine *eiserne Disziplin*.

Ein Turings Worten:

> »A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.«

Turings erste Idee: Endlich vielen Symbole pro Zelle

34-7

Turings Ideen

Der Mathematiker mag gute Augen haben, jedoch passen in ein Kästchen nur endlich viele unterschiedliche Symbole. Es nützt dem Mathematiker auch nichts, Symbole »über mehrere Kästchen zu strecken«, denn das entspricht einfach mehreren Symbolen in mehreren Kästchen.

Formalisierung

Es gibt ein *Bandalphabet*. Wie jedes Alphabet ist dieses *endlich*. Die Symbole stehen in Kästchen, formal (Speicher-)*Zellen* genannt.

34-8

Turings zweite Idee: Zellen auf einem Band

Turings Idee

Die Zellen mögen auf einem zweidimensionalen »Blatt« verteilt sein, bei einer sehr langen Rechnung muss man aber so oder so »blättern«. Da kann man die zweidimensionalen Blätter gleich weglassen und nur ein *Band* benutzen.

Formalisierung

Die Zellen werden in *eindimensionalen Bändern* angeordnet. Prinzipiell sind Bänder »unendlich lang«, sie sind aber bis auf einen endlichen Anteil leer. Dieser endliche Anteil ist damit einfach ein *Wort über dem Bandalphabet*.

34-9

Turings dritte Idee: Diskrete Zeitschritte

Turings Idee

Der Mathematiker kann nicht »unendlich schnell« arbeiten.

Formalisierung

Die Berechnung verläuft in *diskreten Schritten*.

34-10

Turings vierte Idee: Der Schreib-Lese-Kopf

Turings Idee

Während der Mathematiker arbeitet, hat er immer nur ein Blatt »aktuell vor sich«. Will er ein anderes Blatt anschauen, so muss er sich zunächst dorthin »durchblättern«.

Formalisierung

Für jedes Band gibt es einen »Schreib-Lese-Kopf«. Dies ist einfach ein Index einer Bandposition. In jedem Berechnungsschritt kann sich der Kopf bewegen, aber nur eine Zelle vor oder zurück.

34-11

Turings fünfte Idee: Die endlich vielen Geisteszustände

Turings Idee

Während der Mathematiker arbeitet, ist sein Gehirn immer in einem »Geisteszustand«.
Beispiel: Ich muss jetzt den aktuellen Beweis überprüfen.
Beispiel: Ich suche nach der schließenden Klammer. Turing argumentiert, es gäbe zwar viele, aber eben nur endlich viele Geisteszustände.

Formalisierung

Es gibt eine endliche Menge Q von Zuständen.

34-12

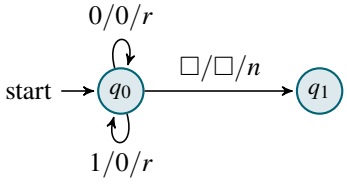
Turing-Berechenbarkeit

▶ **Definition:** Turing-Berechenbar

Eine Funktion f, die Wörter auf Wörter abbildet, heißt *Turing-berechenbar,* wenn eine Turing-Maschine bei Eingabe eines beliebigen Wortes w nach endlich vielen Schritten das Wort $f(w)$ auf ihrem Band stehen hat und anhält.

Beispiel: Eine Turing-Maschine für $f\colon w \mapsto 0^{|w|}$

- Die Zustandsmenge ist $Q = \{q_0, q_1\}$.
- Der Anfangszustand ist q_0.
- Das Bandalphabet ist $\Gamma = \{0, 1, \square\}$.
- Das Eingabealphabet ist $\Sigma = \{0, 1\}$.
- Die Bandanzahl ist 1.
- Das Programm ist

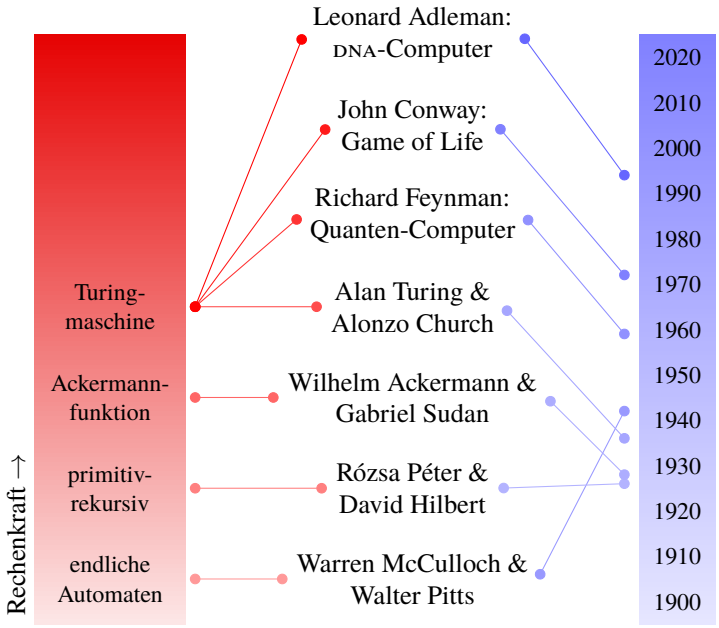


34.2 Die Church-Turing-These

34.2.1 Historischer Rückblick

Geschichte der Berechnungsmodelle

34-13

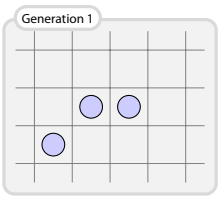


Conways Game of Life.

34-14

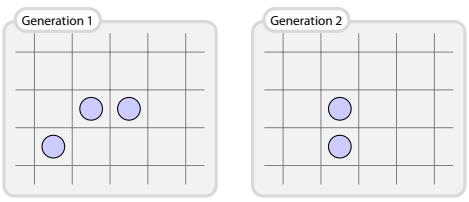
Das Spielbrett

- Das *Spielbrett* ist ein unendliches Gitter.
- In jedem Gitterquadrat kann sich eine *Zelle* befinden oder auch nicht.
- Am Anfang sind einige Gitterquadrate mit Zellen belegt. Sie bilden die *Anfangspopulation*.



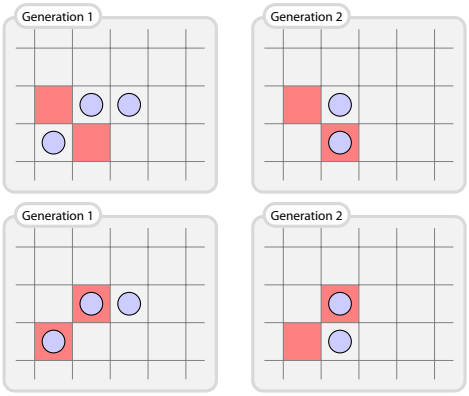
Die Spielrunden

- Das Spiel verläuft in *Generationen* (Runden).
- In jeder Generation werden eventuell neue Zellen *geboren*, alte können *sterben* und Zellen können auch einfach *überleben*.
- Dafür sind die acht umliegenden Zellen wichtig, genannt ihre *Umgebung*.



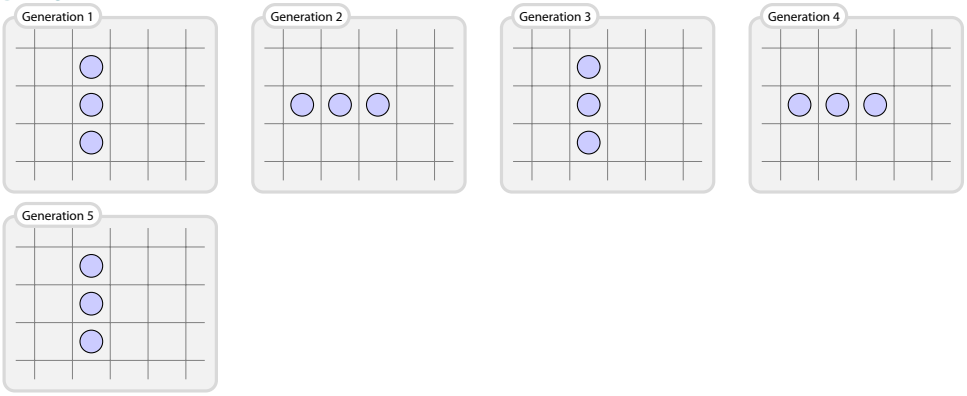
Die Regeln

- Eine Zelle wird *geboren*, wenn es in ihrer Umgebung genau *3* Zellen gibt.
- Eine Zelle *überlebt*, wenn es in ihrer Umgebung genau *2* oder *3* Zellen gibt.
  Sonst *stirbt* sie an Vereinsamung oder Überbevölkerung.

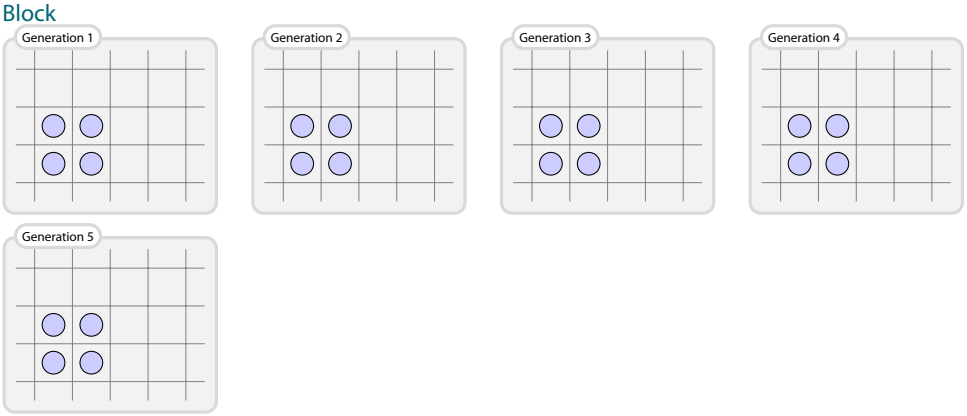


Ein paar interessante Anfangspopulationen

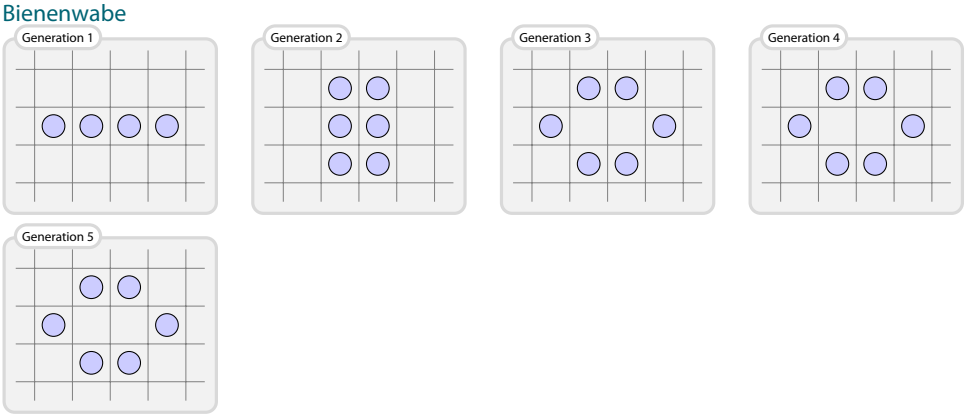
Blinker



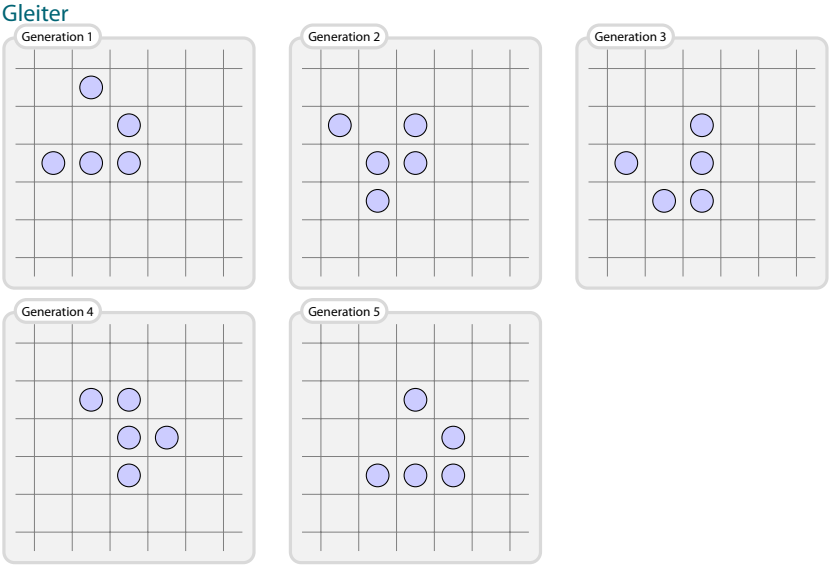
Block



Bienenwabe



Gleiter



Einfache Regeln können komplexe Effekte haben. 34-16

Behauptung

Das Game of Live kann genauso viel wie Turing-Maschinen. (Man sagt, es sei *Turing-mächtig*.)

Etwas genauer:

▶ Satz

Es gibt eine (einfache) Kodierung von Wörtern $w \in \{0,1\}^*$ als Life-Spielbretter, so dass für jede Turing-berechenbare Funktion f gilt: Wenn man das Spiel mit w startet, so wird nach endlich vielen Schritten eine Population erreicht, die gerade $f(w)$ kodiert.

Dies beweist man, indem man mit Gleitern, Gleiterkanonen, Reflektoren, Verknüpfern und allerlei weiteren Konstruktionen die Berechnung einer Turing-Maschine auf dem Spielbrett nachvollzieht.

34.2.2 Die These

Die Church-Turing-These 34-17

These

Die »intuitiv berechenbaren« Funktionen sind gerade Turing-berechenbaren.

- Diese These kann man *prinzipiell nicht beweisen*.
- Man kann aber versuchen, sie zu *widerlegen*.
- Dazu müsse man eine Funktion finden, »die man berechnen kann« (wie auch immer), die aber nicht von Turing-Maschinen berechnet werden kann.

34.3 Nichtberechenbares

34.3.1 Das Postsche Korrespondenzproblem

34-18

Über das Postsche Korrespondenzproblem.

Public domain

- Das *Postsche Korrespondenzproblem* (PKP) ist eine *Sprache*.
- Sie geht auf Emil Post zurück.
- Die Sprache ist recht einfach aufgebaut und *erscheint zunächst einfach zu berechnen*.
- Erstaunlicherweise ist sie aber *nicht berechenbar*.

34-19

Die Idee

Die *Eingabe* für das PKP ist ein *Wörterbuch*:

Deutsch	Vogonisch
apfel	apf
birne	rnx
mus	elmus
nixe	qwertz

(Vogonisch, insbesondere in Gedichtform, ist nicht unbedingt für seine melodische Struktur bekannt.) Die Frage ist nun, ob man einen Satz in der einen Sprache finden kann, so dass die *wortwörtliche Übersetzung* in die andere Sprache *genau denselben Satz* wie in der ersten Sprache liefert. Beispielsweise wird aus »apfel mus« die Übersetzung »apf elmus«, also – wenn man die Leerzeichen ignoriert – in beiden Fällen »Apfelmus«. Man nennt dies eine *Korrespondenz*.

34-20

Ein erstaunliches Resultat.

▶ **Definition:** Die Sprache PKP

Das *Postsche Korrespondenzproblem (*PKP*)* enthält (die Codes von) allen Wörterbüchern, für die es eine Korrespondenz gibt.

Beispiel

$\{(1,100000),(00,1),(0110,0)\} \in$ PKP

▶ **Satz**

PKP *ist nicht berechenbar.*

34.3.2 Das Busy-Beaver-Problem

Über fleißige Biber.

34-21

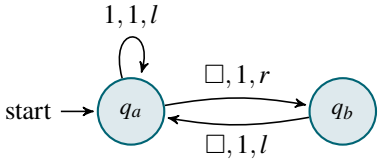
Unknown author, Creative Commons Attribution Sharealike

- Ein *fleißiger Biber* ist ein Turing-Maschine.
- Er hat *möglichst wenige Zustände*,...
- ...kennt nur die Symbole 1 und □,...
- ...startet auf einem leeren Band,...
- ...schreibt möglichst viele 1en...
- ...und hält dann aber irgendwann an.

Definition von fleißigen Bibern.

34-22

Beispiel: Fleißiger Biber mit zwei Zuständen

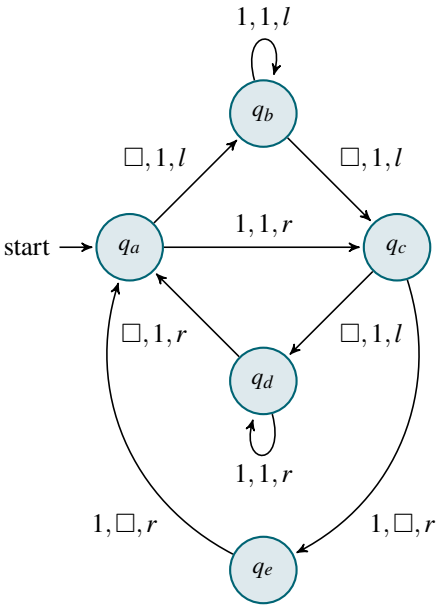


▶ Definition: Fleißige-Biber-Funktion
Die *Fleißige-Biber-Funktion busybeaver*: $\mathbb{N} \to \mathbb{N}$ ordnet jeder Zahl $n \geq 0$ die Anzahl an 1en zu, die ein fleißiger Biber mit n Zuständen schreibt.

Anzahl n an Zustände	$busybeaver(n)$
1	0
2	3
3	5
4	12
5	vermutlich 4097
6	$\geq 4{,}64 \cdot 10^{1439}$

Kandidat für einen fleißigen Biber mit fünf Zuständen.

34-23



- Dieser Biber macht 47.176.870 Schritte, bevor er anhält.
- Er hinterlässt 4097 viele 1en.
- Es ist nicht bewiesen, dass dies tatsächlich ein *fleißiger* Biber ist.

Die Fleißige-Biber-Funktion ist nicht berechenbar.

34-24

▶ Satz
Die Fleißige-Biber-Funktion ist nicht berechenbar.

34.3.3 Kolmogorov-Komplexität

34-25

Über Komprimierer.

Ein *Komprimierer* ist ein Programm. Es bekommt als *Eingabe einen String*. Es liefert als *Ausgabe einen String,* der möglichst kurz ist und aus dem sich die Eingabe rekonstruieren lässt.

Beispiel: Klassische Komprimierer

- Lempel-Ziv-Komprimierer (`gzip`)
- Huffmann-Komprimierer (`gzip`, `mpeg`)

Beispiel: Moderne Komprimierer

Burrows-Wheeler-Transformation (`bzip2`)

34-26

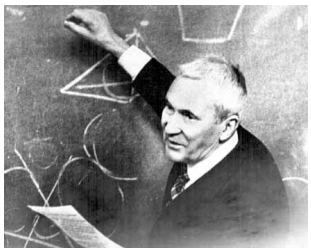
Die Idee hinter dem Kolmogorov-Komprimierer

Yan Shuangchun, Lesser GNU public license

Bei guten Komprimierern gibt es *generell viele Möglichkeiten, einen String zu kodieren.* Dadurch ist dann der Komprimierer besonders flexibel: Er kann die *»passendste« Art* suchen, einen String zu kodieren. Man kann sich nun fragen: *Was ist die ultimativ flexibelste Art, einen String zu kodieren?* Die Antwort lautet: *Als Programmtext, der diesen String als Ausgabe liefert.*

34-27

Der Kolmogorov-Komprimierer



Public domain

▶ **Definition:** Kolmogorov-Komprimierer

Der *Kolmogorov-Komprimierer* ist eine Funktion $K\colon \{0,1\}^* \to \text{ASCII}^*$, so dass für alle Worte $w \in \{0,1\}^*$ gilt:

1. $K(w)$ ist ein Java-Programm,
2. das, wenn man es startet, w ausgibt, und
3. minimale Länge hat.

34-28

Der Kolmogorov-Komprimierer kann sehr stark komprimieren. . .

Manche Worte lassen sich sehr gut Kolmogorov-komprimieren:

Beispiel

Das Wort $1^{1000000}$ (also eine Million 1en) lässt sich mittels eines sehr kurzen Programms beschreiben:

```
for (int i=0; i<1000000; i++) System.out.print("1");
```

Beispiel

Das Wort $1^{10^{100}}$ (also ein Googol viele 1en) lässt sich auch kurz beschreiben:

```
int p = 1;
for (int i=0; i<100; i++) p = p * 10;
for (int i=0; i<p;   i++) System.out.print("1");
```

34-29

. . . aber nicht alles lässt sich komprimieren.

Andererseits lassen sich *viele Worte gar nicht komprimieren*. Dies liegt daran, dass man ja sonst *rekursiv immer wieder komprimieren könnte* – irgendwann muss Schluss sein. Grob gesprochen lassen sich »zufällige Worte« nicht komprimieren:

Gut komprimierbar	schlecht komprimierbar
0000000000	0101101101
0000011111	1011111010
0101010101	0010110111

Ein trauriges Resultat.

34-30

▶ Satz
Der Kolmogorov-Komprimierer ist nicht berechenbar.

Beweis. Nehmen wir an, wir könnten K berechnen. Sei w das kürzeste und lexikographisch erste Wort, so dass die Länge des Programms $K(w)$ gerade eine Million ist. Dann kann man ein (kurzes) Programm schreiben, das in einer Schleife alle Wörter in der Reihenfolge λ, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, ... durchläuft. Für jedes Wort x berechnet es $K(x)$ und gibt das erste aus, für das $|K(x)| = 1000000$. Dieses Programm ist dann ein *sehr kurzes Programm, das w beschreibt* – aber das kürzeste solche Programm sollte ja Länge 1000000 haben. □

Zusammenfassung dieses Kapitels

34-31

▶ Turing-Maschine
Die Turing-Maschine ist ein *extrem einfaches Modell von Rechnern,* das Alan Turing 1936 vorgeschlagen hat.

▶ Turing-Berechenbarkeit
Eine Funktion heißt *Turing-berechenbar,* wenn eine Turing-Maschine zu jedem Eingabewort den Funktionswert nach endlich vielen Schritten auf ihrem Band zu stehen hat.

▶ Church-Turing-These
Alles, was sich berechnen lässt, lässt sich von Turing-Maschinen berechnen.

▶ Nichtberechenbare Probleme
Es gibt Funktionen, die nicht berechenbar sind. Beispiele sind das Postsche Korrespondenzproblem, die Fleißige-Biber-Funktion und der Kolmogorov-Kompressor.

Zum Weiterlesen

[1] Paul Chapman. Life Universal Computer, 2002. http://www.igblan.free-online.co.uk/igblan/ca/, Zugriff April 2014

Hier wird gezeigt, wie man mit dem Game of Life Register-Maschinen simulieren kann.

[2] The LifeWiki. http://www.conwaylife.com/wiki, Zugriff April 2014

Eine Quelle vielfältiger Informationen zum Thema. Hier findet man auch ein recht gutes Applet zur Simulation des Spiels.

Teil VIII
Datenbanken

Was tun Computer eigentlich den ganzen Tag? Die Bezeichnung »Computer« ebenso wie das deutsche »Rechner« legt nahe, dass sie ständig spannende mathematische Probleme lösen, die die Menschheit der Weltformel etwas näher bringt. Die Wirklichkeit sieht leider etwas prosaischer aus: In Wirklichkeit ähneln Computer eher gewissenhaften Verwaltungsbeamten. Hauptsächlich sind sie damit beschäftigt, in irgendwelche Tabellen Daten einzufügen, sie zu löschen oder in diesen Tabellen zu suchen. (Daher erscheint es mir wenig wahrscheinlich, dass Computer, sollten sie jemals intelligent werden, sofort wie Skynet in *Terminator* die Weltherrschaft übernehmen werden. Realistischer erscheint, dass sie wie Marvin in *Per Anhalter durch die Galaxis* gelangweilt und depressiv ihre Einfüge-, Löschen- und Suchoperationen durchführen werden.)

Die Informatik-Teildisziplin der *Datenbanken* beschäftigt sich mit der Frage, wie man Programme so gestalten kann, so dass sie diese Einfügen-, Löschen- und Suchoperationen möglichst effizient hinbekommen. Man kann sagen, dass diese Teildisziplin mehr als erfolgreich war: In Form der *relationalen Datenbanken*, die durch die *Structured Query Language* angesprochen werden, gibt es Systeme, die nur noch wenige Wünsche offen lassen. Solche Datenbanken können riesige Mengen an Daten hocheffizient verwalten (»riesig« heißt »viele, viele Terabyte«). Sie sind sowohl frei wie kommerziell verfügbar. Sie werden durch eine einheitliche, standardisierte und recht einfache Sprache angesprochen. Schließlich (das freut den Theoretiker besonders) steckt hinter ihnen eine ebenso elegante wie einfache Theorie, die auch noch wirklich nützlich ist.

In diesem Teil über Datenbanken soll es zunächst darum gehen, wie Datenbanken aufgebaut sind und wie die angesprochene Theorie dahinter aussieht. Wir werden nur so genannte relationale Datenbanken behandeln (aus Gründen, die noch genauer erläutert werden). Dann schauen wir uns die Sprache SQL genauer an, mit der man mit Datenbanksystemen kommuniziert. Als sehr praktische Anwendungen werden Sie lernen, wie Sie mit dieser Sprache auf Bioinformatik-Datenbankserver zugreifen und in deren Datenbeständen stöbern.

35-1

Kapitel 35

Einführung zu Datenbanken

Wie speichert man eine Milliarde Molekülstrukturen – und wie findet man sie wieder?

Lernziele dieses Kapitels

1. Das Konzept der Datenbank kennen und verstehen
2. Das E/R-Modell kennen
3. Daten mittels des E/R-Modells modellieren können

35-2

Inhalte dieses Kapitels

35.1 Datenbanksysteme 306
35.1.1 Was sind Datenbanken? 306
35.1.2 Aufbau von Datenbanken 307
35.1.3 Arten von Datenbanken 307

35.2 Datenmodelle 308

35.3 Das E/R-Modell 308
35.3.1 Einführung 308
35.3.2 Entitäten 309
35.3.3 Attribute 309
35.3.4 Relationships 310

Übungen zu diesem Kapitel 311

Worum es heute geht

Das Problem, eine große Menge an Daten zu speichern und in ihnen zu suchen, ist uns schon häufiger über den Weg gelaufen. Wir haben auch schon mehr oder weniger raffinierte Datenstrukturen kennengelernt, um Daten so zu speichern, dass man effizient darauf zugreifen kann: Man kann die Daten in einem sortierten Array speichern, in einer verketteten Liste, einem Suchbaum oder einer Hash-Tabelle. Alle Arten haben ihre Vor- und Nachteile.

Trotzdem beschleicht Sie wahrscheinlich auch das Gefühl, dass die Programme, die beispielsweise die Daten der Biodatenbank Ensembl verwalten, keine einfachen Java-Programme mit ein paar Suchbäumen sein werden. Auf Ensembl greifen jede Sekunde viele unterschiedliche Benutzer gleichzeitig zu, der Service von Ensembl muss auch verfügbar bleiben, wenn eine der Platten der Server kaputtgeht, und noch viele weitere Anforderungen werden an Ensembl gestellt, die wir mit unseren bisherigen Programmierkenntnissen nur schwer werden lösen können.

Glücklicherweise muss man das Rad nicht immer wieder neu erfinden. Vielmehr gibt es fix und fertig implementierte *Datenbanksysteme*, welche die oben angesprochenen Probleme (und noch einige Probleme, an die Sie oder ich noch gar nicht gedacht haben) in sehr effizienter Weise lösen. Der interne Aufbau von solchen Datenbanksystemen ist eine Wissenschaft für sich – uns wird nur interessieren, wie man sie benutzt. (Es sei aber verraten, dass sehr fortgeschrittene Suchbäume intern genutzt werden.)

Eine Datenbanksystem ist also ein Programm, dass es in der Regel mehreren Personen gleichzeitig erlaubt, auf Daten zuzugreifen. Dabei werden die klassischen Grundoperationen unterstützt: Einfügen, Löschen und Suchen.

Wenn nun aber Datenbanksysteme keine Java-Programm sind, so stellt sich das *Modellierungsproblem* neu. Sie erinnern sich: In Java haben wir mittels Klassen und Objekten »die Wirklichkeit« modelliert und waren auch recht zufrieden damit. Leider hat sich bei Datenbanksystemen die Objektorientierung noch nicht so stark durchgesetzt wie bei Program-

miersprachen – die meisten Datenbanksysteme sind leider nicht objektorientiert. Deshalb werden zur Modellierung der Wirklichkeit für nichtobjektorientierte Datenbanksysteme ein Vorläufer der Klassendiagramme verwendet: Die *Entity-Relationship-Diagramme*.

In der heutigen Vorlesung soll es zunächst darum gehen, was man von einem Datenbanksystem prinzipiell erwarten darf. Danach werden Entity-Relationship-Modelle vorgestellt. Die genaue Syntax zur realen Kommunikation mit einem Datenbanksystem werden wir in der Vorlesung über SQL kennen lernen.

35.1 Datenbanksysteme

35-4

Die Probleme von Molecular Sheep.

Die Firma *Molecular Sheep* hat an verschiedenen Standorten Labore, in denen das Genom von Schafen sequenziert wird. Sequenzierte Fragmente müssen *gesammelt* und *weiterverarbeitet* werden. Dabei stellt sich zunächst das Problem, die Daten an eine zentrale Stelle *zu übertragen*. Ein zweites Problem ist der *gleichzeitige Zugriff* verschiedener Personen auf die Daten.

35.1.1 Was sind Datenbanken?

35-5

Datenbanken und Datenbanksysteme.

Was sind Datenbanken?

Eine *Datenbank* ist ein strukturierte Sammlung von Daten.

- Man kann Daten in eine Datenbank *einfügen*.
- Man kann Daten aus einer Datenbank *löschen*.
- Man kann nach Daten in einer Datenbank *suchen*.

Was sind Datenbanksysteme?

Ein *Datenbanksystem* ist ein Programm, das eine oder mehrere Datenbanken verwaltet.

35-6

Was bieten Datenbanksysteme?

Ein Datenbanksystem kann:

1. Daten aus Datenbanken physikalisch effizient speichern.
2. Zugang zu Daten in Datenbanken herstellen.
   - Man kann Daten in Datenbanken *einfügen*.
   - Man kann Daten in Datenbanken *löschen*.
   - Man kann Daten in Datenbanken *suchen*.
3. Benutzer verwalten.
   - Mehrere Benutzer können *gleichzeitig* auf die Datenbanken zugreifen.
   - Benutzer können verschiedene *Rechte* haben (wie »darf nur Suchen«).

35-7

Sollte man eine Datenbank verwenden?

Was für Datenbanken spricht

+ Grundoperationen sind *viel schneller und besser* implementiert als man sie »selbst programmieren könnte«.
+ Daten sind immer *automatisch* auf der Festplatte gesichert.
+ *Mehrere Benutzer* können *gleichzeitig* zugreifen.
+ *Verschiedene Programme* können *gleichzeitig* zugreifen.
+ Datenbanksysteme können *sehr billig* sein.

Was gegen Datenbanken spricht

– Die Grundoperationen *Einfügen*, *Löschen* und *Suchen* kann man auch »selbst programmieren«.
– Man muss *neue Sprachen* lernen (zum Beispiel SQL).
– Datenbanksysteme können *sehr teuer* sein.

35.1.2 Aufbau von Datenbanken

Schichten, durch die eine Anfrage an eine Datenbank durchläuft. 35-8

1. *Anwendungsprogramme*
   Sie stellen Anfragen an die Datenbank mit Hilfe einer speziellen *Anfragesprache*.
2. *Externe Schemata*
   Die Anfragen einer Anwendung beziehen sich auf eines von mehreren externen Schemata. Sie »gaukeln einen bestimmten Aufbau der Daten vor«. So kann ein Schema Teile der Datenbank ausblenden, auf die ein Programm keinen Zugriff haben soll.
3. *Konzeptionelles Schema*
   Die Anfragen in Bezug auf ein externes Schema werden in Anfragen in Bezug auf das konzeptionelle Schema umgewandelt. Das konzeptionelle Schema beschreibt, wie die Daten logisch strukturiert sind. Dieses Schema ist das zentrale Schema, das man beim Aufbau einer Datenbank zu Anfang festlegen muss.
4. *Internes Schema*
   Anfragen werden weiter verwandelt in Anfragen in Bezug auf ein internes Schema. Es wird vom Datenbanksystem automatisch erstellt und ist eine optimierte, hocheffiziente Verwaltungsstruktur.
5. *Externe Speicher*
   Die Anfragen in Bezug auf des interne Schema werden in Zugriffe auf die *Festplatten* umgewandelt.

Welche Schichten gehören zu einer Datenbank? 35-9

Nur die drei mittleren Schichten gehören zu einer Datenbank und werden vom Datenbanksystem verwaltet. Das *konzeptionelle Schema* wird bei der Erstellung der Datenbank einmal angegeben und dann in der Regel nicht mehr geändert. Die *externen Schemata* werden ebenfalls bei der Erstellung der Datenbank angegeben, können aber oft auch noch später geändert werden. Das *interne Schema* wird automatisch erzeugt und man hat darauf keinen Zugriff.

Merke

Die Anfragesprachen für Datenbanken erlauben zwei unterschiedliche Dinge:

1. Erstellung und Veränderung der Schemata.
2. Modifikation der Daten, wenn das Schema festgelegt ist.

Datenbanken und Mehrbenutzerbetrieb. 35-10

Wenn mehrere Anwender gleichzeitig Daten in der Datenbank *suchen*, ist das im Allgemeinen kein Problem. Wenn sie aber gleichzeitig Daten *ändern* wollen (schlimmstenfalls sogar die gleichen Daten), so können *vielfältige Konflikte* entstehen. *Ein Datenbanksystem kümmert sich um all diese Probleme und löst sie automatisch auf.*

35.1.3 Arten von Datenbanken

Arten von Datenbanken. 35-11

Datenbanksysteme unterscheiden sich unter anderem in folgenden Punkten.

- Menge der verwaltbaren Daten (von einigen Megabytes bis zu tausenden Terabytes).
- Anzahl der verwaltbaren Benutzer (von einem einzigen bis zu Millionen).
- Art der verwaltbaren Daten (Tabellen, Graphiken, Objekte, Filme).
- Art der verwaltbaren Schemata (relational, hierarchisch, objekt-orientiert).
- Geschwindigkeit und Sicherheit.
- Hersteller und Preis.

Arten von Schemata. 35-12

Ein fundamentaler Unterschied zwischen Datenbanken ist, wie ihre Schemata aufgebaut sein können:

relational In der Datenbank lassen sich (nur) Relationen (hocheffizient) speichern. Eine Relation setzt verschiedene Dinge in Beziehung, wie zum Beispiel Schafe mit den für sie sequenzierten Fragmenten.

hierarchisch In der Datenbank lassen sich (nur) hierarchische Strukturen speichern.

OO In der Datenbank lassen sich (nur) Objekte im Sinne der objektorientierten Programmierung speichern. Das Schema ist durch die Klassenstruktur gegeben.

35-13

Wir betrachten nur relationale Datenbanken.

Wir werden im Folgenden *nur relationale Datenbanken* betrachten. Dies hat verschiedene Gründe:

- Diese Datenbanksysteme sind *ausgereift*.
- Es gibt *frei verfügbare, gute* Implementationen von relationalen Datenbanken.
- Sie lassen sich *extrem gut optimieren* und sind daher oft *sehr effizient*.
- Es gibt eine *einheitliche, einfache Anfragesprache* für sie, nämlich *SQL*.

Hier ein paar Nachteile von relationalen Datenbanken:

- Das relationale Modell passt schlecht zum objektorientierten Modell, das Anwendungsprogramme benutzen.
- In manchen Situationen sind hierarchische Datenbanken wesentlich schneller.

35.2 Datenmodelle

35-14

Was sind Datenmodelle?

Die verschiedenen Arten von Schemata spiegeln verschiedene Arten von *Datenmodellen* wider. Ein Datenmodell beschreibt *die konzeptionelle Struktur* der Daten.

Beispiel

In der Datenbank von Molecular Sheep sollen identifizierte Schafsgene gespeichert werden. Das Datenmodell legt nun fest:

- Welche Informationen über ein isoliertes Gen sollen gespeichert werden (Ort, Wissen über die Funktion, Länge)?
- Welche Informationen über Beziehungen zu anderen Genen sollen gespeichert werden (welche Gene bilden einen Pathway)?
- Welche Informationen über Beziehungen zu anderen Objekten in der Datenbank sollen gespeichert werden (welche Schafe haben dieses Gen, welche identifizierten Promoter hat das Gen)?

35-15

Wir betrachten nur E/R-Modelle.

Es gibt zwei wichtige Arten von Datenmodellen:

- Entity-Relationship-Modelle.
  Sie passen gut zu relationalen Datenbanken, weshalb wir diese betrachten werden. Wie man vom E/R-Modell zu den Relationen in einer relationalen Datenbank kommt, wird im nächsten Kapitel erklärt.
- UML-Modelle (unified modelling language)
  UML-Modelle passen gut zu objektorientierten Programmen, weshalb sie in der Softwaretechnik viel eingesetzt werden.

35.3 Das E/R-Modell

35.3.1 Einführung

35-16

Was ist ein E/R-Modell?

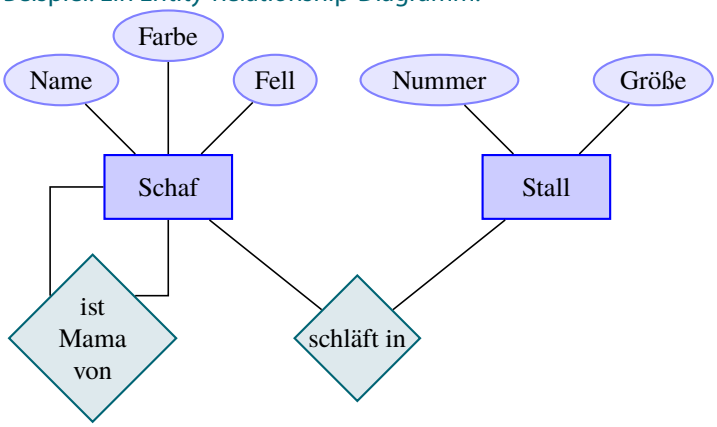
Das *Entity-Relationship-Modell* ist ein *Datenmodell*. Es dient dazu, *Entitäten* und deren *Beziehungen* zu beschreiben. Ein E/R-Modell legt fest, welche *Arten* von Entitäten es geben kann und welche *Arten* von Beziehungen. Es legt noch nicht fest, welche Entitäten und Beziehungen zwischen konkreten Entitäten es gibt.

Die Bestandteile eines E/R-Modells. 35-17

Ein E/R-Modell besteht aus drei Arten von Dingen:

1. *Entitätstypen*
   Dies sind Arten von Objekten oder Dingen, über die Informationen in der Datenbank gespeichert werden sollen.
   Beispiel: Schafe, Gene
2. *Attribute*
   Dies sind Eigenschaften von Entitäten, die in der Datenbank gespeichert werden sollen.
   Beispiel: Farbe des Schafs, Basenposition des Gens
3. *Relationshiptypen*
   Dies sind Arten von Beziehungen, die zwischen Entitäten bestehen.
   Beispiel: Schafe haben Gene, Gene wirken zusammen mit anderen Genen

Beispiel: Ein Entity-Relationship-Diagramm. 35-18



35.3.2 Entitäten

Näheres zu Entitäten. 35-19

- Eine *Entität* ist ein Objekt, über das wir Informationen speichern.
  Beispiel: Das Schaf Dolly
- Eine *Entitätsmenge* ist eine Menge von Entitäten. Man spricht aber (salopp und fälschlicherweise) auch oft von *entities*, obwohl man Entitätsmengen meint.
  Beispiel: Schafe, schwarze Schafe

✎ Zur Übung
Geben Sie Entitätsmengen an, die Molecular Sheep in seiner Datenbank halten sollte.

35.3.3 Attribute

Näheres zu Attributen. 35-20

- Eine *Attribut* ist eine Eigenschaft von Entitäten.
  Beispiel: Die Farbe von Schafen
- Die Menge aller Entitäten, die bestimmte Attribute haben, bilden den *Entitätstyp* dieser Attribute.
  Beispiel: Schafe haben eine Farbe, eine Fellart, einen Namen.
- In einem E/R-Modell malt man Entitätstypen als *Rechtecke* auf.
- In einem E/R-Modell malt man Attribute als *Ovale* auf mit einer Kante zum Entitätstyp.

✎ Zur Übung
Geben Sie Attribute für die Entitäten aus der vorherigen Aufgabe an.

35-21

Entitätstabellen enthalten Entitäten mit ihren Attributen.

Für jeden Entitätstyp gibt es in der Datenbank später eine *Tabelle*. Diese enthält für jede Entität eine *Zeile*. Die *Spalten* sind die Attribute der Entität. Eine Spalte, anhand derer man die Entität eindeutig identifizieren kann, heißt *Schlüsselattribut* oder einfach nur *Schlüssel*.

Beispiel

Name	Farbe	Fell
Dolly	weiß	lockig
Max	schwarz	wuschelig
Peter	schwarz	wuschelig
Flauschi	beige	glatt

35.3.4 Relationships

35-22

Näheres zu Relationships.

- Eine *Relationship* ist eine Beziehung zwischen zwei oder mehr Entitäten.
  Beispiel: Dolly *hat* das Asthma-Gen
- Der *Relationshiptyp* beschreibt, dass Beziehungen zwischen den Entitäten bestimmter Typen bestehen können.
  Beispiel: Schafe *haben* Gene
- In einem E/R-Modell malt man Relationshiptypen als *Rauten* auf mit Kanten zu den Entitätstypen.

✎ **Zur Übung**

Geben Sie Relationships für die Entitäten aus der vorherigen Aufgabe an.

35-23

Relationshiptabellen enthalten Tupel von Entitätsschlüsseln.

Für jeden Relationshiptyp gibt es in der Datenbank später wieder eine *Tabelle*. Diese enthält für jede Relationship eine *Zeile*. Die *Spalten* sind die Schlüssel der beteiligten Entitätstypen.

Beispiel

Die Tabelle des Relationshiptyps »haben«.

Schafs-Name	Gen-Name
Dolly	Asthma-Gen
Dolly	Intelligenz-Gen
Max	Asthma-Gen
Peter	Intelligenz-Gen

35-24

Relationships können auch Attribute haben.

Auch Relationships können *Attribute* haben. Diese werden wie bei Entitäten im Diagramm als Ovale dargestellt.

Beispiel

Die Tabelle des Relationshiptyps »leiden an«.

Schafs-Name	Krankheit	seit
Dolly	Scrapie	1. Juli 2006
Dolly	MKS	2. Juli 2006
Max	MKS	2. Juli 2006
Peter	Scrapie	5. Juli 2006

Zusammenfassung dieses Kapitels

35-25

▶ Datenbanken versus Datenbanksysteme

Eine *Datenbank* ist eine Sammlung von Daten. Ein *Datenbanksystem* ist ein Programm zur Verwaltung von Datenbanken.

▶ Schemata

Ein *Datenbankschema* beschreibt in Form von »Tabellenköpfen«, welche Daten prinzipiell in einer Datenbank gespeichert werden können.

▶ E/R-Modelle

Ein E/R-Modell ist ein Schema, das Entitätstypen, deren Attribute und Relationshiptypen graphisch beschreibt. Zu jedem Entitätstyp und jedem Relationshiptyp gibt es in der Datenbank später genau eine Tabelle.

Übungen zu diesem Kapitel

Übung 35.1 E/R-Diagramme entwerfen, mittel

Bei dieser Gruppenübung sollen Sie E/R-Diagramme zur Modellierung verschiedener Szenarien erstellen. Der Ablauf:

1. Ihr Tutor teilt Sie in Gruppen ein. Jede Gruppe bekommt eines der unten angegebenen Szenarien zugeordnet.
2. Überlegen Sie, welche Entitätsmengen und Relationen Sie benötigen. Schreiben Sie die Entitätsmengen und Relationen mit allen Attributen in Tabellenform auf. (30 Minuten)
3. Erstellen Sie auf einer Flip-Chart-Folie ein E/R-Diagramm Ihres Szenarios. (15 Minuten)
4. Präsentieren Sie Ihr E/R-Diagramm der gesamten Gruppe. (5 Minuten je Szenario)

Bei jedem Szenario soll Ihr E/R-Modell mindestens drei Entitätsmengen und drei Relationen enthalten. Jede Entitätsmenge soll mindestens drei Attribute aufweisen. Mindestens eine Relation soll mindestens ein Attribut aufweisen.

Szenario 1: CD-Sammlung

Ihre CD-Sammlung platzt in letzter Zeit aus allen Nähten, so dass Sie kaum etwas wiederfinden. Daher möchten Sie eine Datenbank Ihrer Sammlung erstellen, um eine Übersicht zu erhalten. Die Datenbank soll in der Lage sein, auf verschiedene Weise Querbezüge zwischen den Alben herzustellen, z.B. nach Genre oder anhand der Interpreten. Auch Sampler mit mehreren Interpreten sollten für Ihre Datenbank kein Problem sein.

Szenario 2: Fußballtippspiel

Nach dem Spiel ist vor dem Spiel und die nächste Europa- oder Weltmeisterschaft ist nie weit. Diesmal wollen Sie richtig auftrumpfen und mit Ihren neu erworbenen Informatikkenntnissen ein internetbasiertes Tippspiel aufziehen. Natürlich reicht Ihnen dafür eine simple Tabelle nicht aus. Sie möchten es »richtig« machen und kaufen schon mal eine Oracle-Lizenz. Damit alles glatt läuft, muss Ihre Datenbank natürlich schon einmal den Spielplan selbst enthalten. Aber auch die Tipper und ihre Punktzahlen müssen selbstverständlich irgendwie gespeichert werden.

Szenario 3: Verwaltung eines Kinos

Auch in einem Kino werden Datenbanken eingesetzt – schließlich möchten Sie normalerweise neben Ihrem Nachbarn und nicht auf oder unter ihm sitzen. Wenn Sie eine Karte für »Rambo IV« kaufen, möchten Sie vermutlcih auch genau diesen Film sehen… Wie würden Sie eine Datenbank entwerfen, die dafür sorgt, dass hier alles glatt läuft?

Szenario 4: Baum des Lebens

Das Web-Projekt “Tree Of Life” bietet umfangreiche Informationen über die Vielfalt der Organismen der Erde, ihre evolutionäre Geschichte und ihre Charakteristika. Auf den einzelnen Seiten werden Informationen über Gruppen von Organismen (Cephalopden, Tyrannosaurier, der Salamanderfisch Westaustraliens) angezeigt. Wie könnte die Datenbank hinter diesem Projekt aussehen?

Prüfungsaufgaben zu diesem Kapitel

Übung 35.2 Erstellen eines E/R-Diagramms, einfach, original Klausuraufgabe, mit Lösung

Es soll eine Datenbank zu Infektionskrankheiten eingerichtet werden. Dazu soll zunächst ein E/R-Modell erstellt werden. Als Entitätsmengen sollen *Krankheit*, *Erreger* und *Medikament* vorkommen. Fügen Sie drei passende Relationen und insgesamt zwei sinnvolle Attribute hinzu und zeichnen Sie ein E/R-Diagramm.

36-1

Kapitel 36
SQL – Modellierung
Wir bauen eine Datenbank

Lernziele dieses Kapitels

1. Das Konzept der Datenbank-Shell kennen
2. Die MySQL-Shell benutzen können
3. E/R-Diagramme in eine relationale Datenbank umwandeln können
4. SQL-Datentypen kennen

36-2

Inhalte dieses Kapitels

36.1	Einführung zu SQL	314
36.1.1	SQL zur Kommunikation	314
36.1.2	Die MySQL-Shell	314
36.1.3	Erste Schritte	315
36.2	Erstellen einer Datenbank	316
36.2.1	Vom E/R-Modell zur Datenbank	316
36.2.2	Anlegen einer Datenbank	316
36.2.3	Anlegen von Tabellen	316
36.2.4	Löschen	317
	Übungen zu diesem Kapitel	318

Worum es heute geht

Im vorigen Kapitel ging es darum, was eine Datenbank *prinzipiell* ist und was *prinzipiell* in sie hinein soll. Mit Prinzipienreiterei kommt man aber auf Dauer nicht weit, irgendwann muss auch mal Butter bei die Fische – konkret: wir brauchen ist eine *Sprache* zur Kommunikation mit Datenbanksystemen. Dabei handelt es sich weder um eine Seitenbeschreibungssprache wie HTML oder LaTeX (schließlich gibt es hier keine Seiten, die beschrieben werden wollen) noch um eine Programmiersprache wie Java, denn wir wollen ja keine Schleifen durchlaufen, sondern Tabellen verwalten. Die Sprache SQL, welche zur Kommunikation mit Datenbanksystemen dient, ist deshalb eine ganz eigene Sprache, die eigentlich keine Gemeinsamkeiten mit anderen Sprachen hat. Sie werden also eine neue Sprache lernen müssen.

So weit die schlechte Nachricht. Die gute Nachricht ist, dass man auch »nur« SQL lernen muss; im Gegensatz zu den normalen Programmiersprachen, die es wie Sand am Meer gibt, gibt es in der Welt der relationalen Datenbanken eigentlich nur SQL. Kann man diese Sprache, so kann man mit jedem Datenbankserver reden, egal ob es sich um einen Miniserver für ein einzelnes Programm handelt oder um das Data-Warehouse eines Großunternehmens. (Der Teufel steckt aber natürlich auch bei SQL im Detail.)

Heute geht es zunächst einmal darum, Daten in eine Datenbank *hinein* zu bekommen und zu verändern. Dafür sind, wenig überraschend, Befehle mit Namen wie »create«, »insert into« oder »delete« zuständig. Um Daten *hinaus* zu bekommen, benutzt man den »select«-Befehl, der aber etwas komplexer ist und dem gleich zwei Kapitel gewidmet sind.

Mit den oben genannten Befehlen hat man alles beisammen, um Datenbanksysteme zu nutzen: Möchte man eine Datenbank anlegen, so überlegt man sich zunächst mit einem E/R-Diagramm, wie die Daten zu modellieren sind. Dann legt man mit diversen Create-Befehlen die notwendigen Tabellen an. Später werden diese Tabellen mit Insert- und Delete-Befehlen gefüllt und auf dem neuesten Stand gehalten. Select-Befehle verwendet man schließlich, um im laufenden Betrieb Information aus der Datenbank herauszubekommen.

36.1 Einführung zu SQL

36.1.1 SQL zur Kommunikation

36-4

Was ist SQL?

SQL steht für *structured query language*, deren Syntax »angeblich menschenlesbar« ist. Sie erfüllt drei Hauptfunktionen:

1. Sie stellt Befehle zur Verfügung, um Relationen und Datenbanken (Ansammlungen von Relationen) zu erstellen und zu verwalten.
2. Sie stellt Befehle zur Verfügung, um Einträge in Tabellen *einzufügen* und *zu löschen*.
3. Sie erlaubt es, *Anfragen* zu formulieren wie »Wer ist der Vater von Dolly?«

Die Sprache ist *deklarativ*. Das bedeutet, dass man bei Anfragen angibt, was man gerne hätte, aber nicht, wie man das berechnen sollte.

36-5

Prinzipielle Kommunikation mit einer Datenbank.

Datenbanken werden von einem *Datenbanksystem* verwaltet. Das Programm, das die Verwaltung durchführt, heißt auch manchmal *Datenbankserver*. Um mit dem Datenbankserver zu reden, wird eine *Datenbank-Shell* benutzt. (Zur Erinnerung: Eine Shell ist ein Programm, mit dem man dialogbasiert mit einem anderen Programm spricht.) In der Datenbank-Shell gibt man SQL-Befehle ein, die an den Datenbankserver übermittelt werden. Die Antworten des Servers zeigt die Shell dann an.

36-6

Fallbeispiel: Ensembl-Datenbank.

Die *Ensembl-Datenbank* ist eine große Datenbank, die vielfältige molekularbiologische Informationen speichert. Der Ensembl-Server ist ein Computer irgendwo auf diesem Globus, auf dem das *MySQL-Server-Programm* läuft und auf dessen Festplatten die Daten lagern. Auf einem beliebigen Computer (zum Beispiel einem im Rechnerpool) kann man nun die Datenbankshell `mysql` starten und die Adresse des Ensembl-Servers angeben. Daraufhin hat man (nur lesenden) Zugriff auf alle Tabellen der Ensembl-Datenbanken. Genauer verwaltet der Ensembl-Server viele Datenbanken und jede enthält wiederum viele Tabellen. Als Nutzer muss man am Anfang eine dieser Datenbanken auswählen, man arbeitet dann nur noch mit den Tabellen dieser einen Datenbank.

36.1.2 Die MySQL-Shell

36-7

Start der MySQL-Shell.

Das Programm `mysql` dient dazu, eine Verbindung zu einem Datenbankserver aufzubauen. Es nimmt viele optionale Parameter, die man sich mittels `mysql --help` alle anzeigen lassen kann. Auf den *Prompt* hin kann man nun SQL-Befehle eingeben. Nützlich sind dort die Hoch- und Runterpfeiltasten, um die letzten eingegebenen Befehle nochmal zu betrachten. Befehle sollte man mit einem Semikolon (wie in Java) abschließen.
Die wichtigsten Optionen sind:

`--host=` Dieser Option sollte der Name des Servers folgen (bei Ensembl beispielsweise `ensembldb.ensembl.org`).

`--user=` Dieser Option sollte der Name des Benutzers folgen, auf dessen Account man sich einloggen möchte (bei Ensembl beispielsweise `anonymous`).

36-8

Beispiele für den Verbindungsaufbau mit zwei Datenbanken.

Verbindung mit Ensembl aufbauen

```
murmel:~ tantau$ mysql \
    --host=ensembldb.ensembl.org --user=anonymous
Welcome to the MySQL monitor.  Commands end with ; or \g.
mysql>
```

Verbindung mit einem Datenbankserver der Uni

```
murmel:~ tantau$ mysql \
    --host=mls-db-server.tcs.uni-luebeck.de \
    --user=student --password
Enter password: *********
Welcome to the MySQL monitor.  Commands end with ; or \g.
mysql>
```

Die Backslashes an den Zeilenenden bedeuten, dass es dort eigentlich *keinen* Zeilenumbruch geben darf (der Text hat aber einfach nicht auf eine Zeile gepasst).

36.1.3 Erste Schritte

Erste nützliche Befehle. 36-9

Auswahl der Datenbank.

Wenn man zum ersten Mal mit einem Datenbankserver verbunden ist, dann sind folgende Befehle nützlich:

- Der `status;` Befehl gibt jede Menge (meist unverständliche) Informationen über den aktuellen Zustand der Verbindung mit dem Server.
- Der **`show databases;`** Befehl liefert eine Liste aller Datenbanken, die der Server verwaltet. Ein Server kann also mehrere Datenbanken verwalten, die ihrerseits viele Tabellen enthalten.
- Der **`use`** *`database`*`;` macht eine Datenbank zur aktiven Datenbank. Alle weiteren Befehle beziehen sich auf diese Datenbank.
  Dabei ist *`database`* der Name einer der Datenbanken, die bei **`show databases`** aufgelistet werden.

Erste nützliche Befehle. 36-10

Beschreibung der Tabellen.

Hat man mittels **`use`** *`database`*`;` eine Datenbank auf dem Server ausgewählt, dann sind folgende Befehle weiter nützlich:

- Der **`show`** `tables;` Befehl liefert eine Liste aller Tabelle, die in der Datenbank vorhanden sind.
- Der **`describe`** *`table`*`;` liefert eine Beschreibung der Spalten (Attribute) der Tabelle *`table`*.

Erste nützliche Befehle. 36-11

Anzeigen einer Tabellen.

Ist man an einer bestimmten Tabelle interessiert, dann kann man sich ihren Inhalt mit folgendem Befehl anschauen:

- **`select count`**`(*)` **`from`** *`table`*`;` zeigt an, wie viele Zeilen die Tabelle *`table`* hat.
- **`select`** `*` **`from`** *`table`*`;` zeigt den gesamten Inhalt der Tabelle an.
- **`select`** `*` **`from`** *`table`* **`limit`** `10;` zeigt nur die ersten 10 Zeilen der Tabelle an.

Der **`select`**-Befehl kann noch *viel mehr*, ihm sind noch zwei Kapitel gewidmet.

36.2 Erstellen einer Datenbank

36.2.1 Vom E/R-Modell zur Datenbank

36-12

Wiederholung: Entitätstabellen enthalten Entitäten mit ihren Attributen.

Für jeden Entitätstyp gibt es in der Datenbank eine *Tabelle*. Diese enthält für jede Entität eine *Zeile*. Die *Spalten* sind die Attribute der Entität. Ein Spalte, anhand derer man die Entität eindeutig identifizieren kann, heißt *Schlüsselattribut* oder einfach nur *Schlüssel*.

Beispiel

Name	Farbe	Fell
Dolly	weiß	lockig
Max	schwarz	wuschelig
Peter	schwarz	wuschelig
Flauschi	beige	glatt

36-13

Wiederholung: Relationshiptabellen enthalten Tupel von Entitätsschlüsseln.

Für jeden Relationshiptyp gibt es in der Datenbank wieder eine *Tabelle*. Diese enthält für jede Relationship eine *Zeile*. Die *Spalten* sind die Schlüssel der beteiligten Entitätstypen.

Beispiel

Die Tabelle des Relationshiptyps »haben«.

Schafs-Name	Gen-Name
Dolly	Asthma-Gen
Dolly	Intelligenz-Gen
Max	Asthma-Gen
Peter	Intelligenz-Gen

36.2.2 Anlegen einer Datenbank

36-14

Wie legt man eine Datenbank in SQL an?

Zunächst baut man einer Verbindung zum Datenbankserver auf. Nun benutzt man aber nicht den `use`-Befehl, um eine Datenbank auszuwählen, sondern den ***`create database`***-Befehl. Dann kann man beginnen, für jeden Entitätstyp und für jeden Relationshiptyp eine Tabelle anzulegen.

```
murmel:~ tantau$ mysql \
    --host=mls-db-server.tcs.uni-luebeck.de --user=student
Welcome to the MySQL monitor.  Commands end with ; or \g.
mysql> create database molecular_sheep;
mysql> use molecular_sheep;
Database changed
mysql>
```

36.2.3 Anlegen von Tabellen

36-15

Wie legt man Tabellen an?

In der aktuellen Datenbank kann man eine Tabelle wie folgt anlegen:

```
mysql> create table schaf (
  name  char(20) primary key,
  farbe char(20),
  fell  char(20)
);
mysql> create table stall (
  nummer  integer primary key,
  groesse float
);
mysql> create table liegt_in (
  name   char(20),
  nummer integer
);
```

Syntax des Create-Befehls.

36-16

Dem **`create table`**-Befehl folgt der Name der Tabelle. Danach kommen in Klammern die Attribute der Tabelle. Jedem Attribut *folgt* der Typ des Attributs (anders herum als in Java). Die wichtigsten erlaubten Typen sind:

- **`tinyint`** ist ein 8-Bit Integer
- **`int`** ist ein 32-Bit Integer
- **`bigint`** ist ein 64-Bit Integer
- **`double`** ist eine 64-Bit Gleitkommzahl
- **`time`**, **`date`** Zeit und Datum
- **`char`**`(n)` ist ein String der Länge $n \leq 255$
- **`varchar`**`(n)` ist ein String der Länge maximal $n \leq 255$
- `text` ist ein String der Länge maximal 65535
- **`longtext`** ist ein String der Länge maximal 4GB.
- **`blob`** (binary large object) ist ein (rotes blubberndes) Ding der Größe maximal 65536 Byte.
- **`longblob`** ist ein Ding der Größe maximal $4GB$.

Wie fügt man etwas in eine Tabelle ein?

36-17

Man kann Werte in eine Tabelle wie folgt einfügen:

```
mysql> insert into schaf
  values ("Dolly","weiss",  "lockig"),
         ("Flauschi",  "schwarz","wuschelig"),
         ("Peter",  "schwarz","wuschelig");
mysql> insert into stall
  values (1,50.0),
         (5,25.0);
mysql> insert into liegt_in
  values ("Dolly", 1),
         ("Flauschi", 5);
```

36.2.4 Löschen

Wie löscht man Dinge aus einer Datenbank?

36-18

- Um die ganze Datenbank zu löschen, schreibt man

  ```
  mysql> drop database molecular_sheep;
  ```

- Um eine Tabelle zu löschen, schreibt man

  ```
  mysql> drop table liegt_in;
  ```

- Um einen Eintrag aus einer Tabelle zu löschen, schreibt man:

  ```
  mysql> delete from schaf
                where name = "Dolly" and farbe = "weiss";
  ```

  Wir werden später noch sehen, dass dies hier nur ein Spezialfall ist. Man kann allgemein Anfragen benutzen zum Löschen.

Zusammenfassung dieses Kapitels

36-19

1. Eine *Datenbank-Shell* wie `mysql` dient dazu, eine Kommunikation mit einem Datenbank-Server herzustellen.
2. Die Sprache *SQL* dient dazu, Datenbank zu verwalten und Anfragen zu formulieren.
3. Datenbanken und Tabellen kann man mit den Befehlen **`create`**, **`insert`**, **`drop`** und **`delete`** verwalten.

Übungen zu diesem Kapitel

Übung 36.1 SQL-Abfragen in der ensembl-Datenbank, einfach

Diese Übung sollte allein am Rechner bearbeitet werden.

1. Öffnen Sie eine Shell und benutzen Sie das Programm mysql, um sich mit der Ensembl Genom-Datenbank auf dem Server `ensembldb.ensembl.org` mit dem Benutzernamen `anonymous` zu verbinden.
2. Lassen Sie sich eine Liste aller Datenbanken anzeigen, die auf dem Server verfügbar sind.
3. Lassen Sie sich alle Tabellen der Datenbank »Core 28.1« von *xenopus tropicalis* anzeigen. Benutzen Sie im Folgenden diese Datenbank.
4. Lassen Sie sich das Schema der Tabelle `gene` ausgeben. Wie viele Einträge enthält die Tabelle?

Übung 36.2 Tabellen angeben, leicht

Geben Sie für wenigstens drei der Entitätsmengen und für wenigstens eine Relationship aus dem von Ihnen bearbeiteten Szenario aus Übung 35.1 je eine (kurze) Tabelle mit Beispieldaten an.

Übung 36.3 Tabellen erstellen, mittel

Diese Aufgabe bezieht sich auf Ihre Lösung zu den E/R-Diagrammen aus Übung 35.1. Geben Sie SQL-Statements an, mit denen zu jeder Entitätsmenge und zu jeder Relation aus den Diagrammen die entsprechende Tabelle in einer SQL-Datenbank erzeugt wird. Geben Sie außerdem SQL-Statements an, mit denen Sie in jede Tabelle Ihre Beispieldaten aus Übung 36.2 einfügen.

Übung 36.4 Tabellen optimieren, mittel

Die Speicherung jeder Relation in einer extra Tabelle ist nicht immmer optimal. Untersuchen Sie die Relationen Ihrer E/R-Diagramme aus Übung 35.1. Muss wirklich jede Relation in einer eigenen Tabelle gespeichert werden, oder finden Sie für einige Relationen eine bessere Lösung? (Hinweis: Abhängig von Ihren Diagrammen muss es nicht unbedingt sein, dass sich eine bessere Lösung finden lässt. Es gibt dann aber immer einen guten Grund dafür, eine extra Tabelle zu verwenden. Welchen?)

Prüfungsaufgaben zu diesem Kapitel

Übung 36.5 Länderdatenbank erstellen, mittel, original Klausuraufgabe, mit Lösung

Sie möchten eine Datenbank der Länder erstellen, die aktuell zur Europäischen Union gehören. Zu jedem Land sollen dabei der Name, die Hauptstadt und die Einwohnerzahl angegeben werden. Außerdem soll anhand der Datenbank ersichtlich sein, welche Länder benachbart sind (zum Beispiel sind Deutschland und Frankreich benachbart, Polen und Spanien sind nicht benachbart).

In SQL können Sie die Länderdatenbank anhand von zwei Tabellen umsetzen, wobei in der ersten Tabelle die Daten der Länder selbst und in der zweiten Tabelle die Nachbarschaften gespeichert werden.

Geben Sie den beiden Tabellen Namen und geben Sie zu jeder Tabelle geeignete Spaltennamen mit den dazugehörigen Datentypen an!

37-1

Kapitel 37
Relationenalgebren
Wer ist Dollys Vater?

37-2

Lernziele dieses Kapitels

1. Konzept der Relation-Algebra kennen
2. Anfragen mittels Operationen formulieren können
3. Joins verstehen und anwenden können

Inhalte dieses Kapitels

37.1 Einführung 320
37.1.1 Alles ist eine Tabelle 320
37.1.2 Relationen-Algebren 320

37.2 Operationen auf Relationen 321
37.2.1 Vereinigung und Schnitt 321
37.2.2 Selektion 321
37.2.3 Projektion 322
37.2.4 Kreuzprodukt 322
37.2.5 Join 323

Übungen zu diesem Kapitel 325

Worum es heute geht

Wie sollte man Daten in einer Datenbank organisieren? Als eine riesige unsortierte Ansammlung von Daten (also eine Art Ursuppe)? Als komplexes, hierarchisches Gebilde? Als verzweigter Graph mit Schleifen? Als Menge von Objekten mit Verweisen, so wie in Java? Und welche Organisationsform ist eigentlich für das Suchen besonders geeignet?

Der Vater der *relationalen Datenbanken*, Edgar Codd, hat folgende scheinbar schlichte Antwort auf diese Fragen gegeben: Eine Datenbank besteht für ihn aus einer Menge von Tabellen mit einer flexiblen Anzahl an Zeilen und einer für jede Tabelle festen Anzahl an Spalten. Punkt. Nicht mehr und nicht weniger. Da »Tabellen« mathematisch nichts anderes sind als »Relationen«, spricht man von relationalen Datenbanken.

Die einzige Aufgabe eines relationalen Datenbanksystems ist es nun, solche Tabellen/Relationen so effizient wie irgend möglich zu speichern. Da Tabellen eher schlichte Geschöpfe sind (verglichen beispielsweise mit Bäumen), ist dies auch ausgesprochen gut möglich. Tatsächlich sind beispielsweise objektorientierte Datenbanken, bei denen statt Tabellen wie in Java Objekte mit Verweisen direkt gespeichert werden, noch nicht so effizient wie relationale Datenbanken.

Da es in relationalen Datenbanken wirklich *nur* Tabellen gibt, muss man alles auf Tabellen zurückführen, was nicht immer ganz logisch ist. Sucht man etwas in einer solchen Datenbank, wie zum Beispiel Dollys Vater, so ist die Idee, aus bestehenden Tabellen neue Tabellen auf geschickte Weise zu erzeugen (zur Erinnerung: Tabellen manipulieren können relationale Datenbanksysteme sehr gut). Am Ende hat man dann eine Tabelle mit nur einer Zeile, die die gewünschte Information enthält. Um relationale Datenbanken zu benutzen, muss man »in Tabellen denken lernen«.

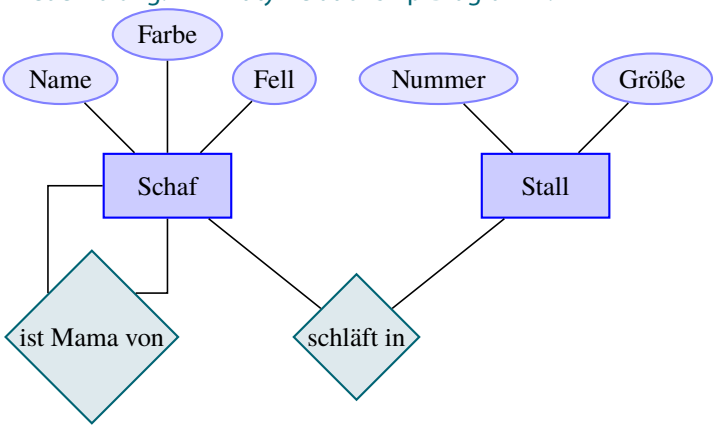
In der heutigen Vorlesung wollen wir uns die Theorie hinter diesen Tabellen etwas genauer anschauen. Insbesondere soll es darum gehen, was Tabellen mathematisch sind und wie man aus bestehenden Tabellen neue Tabellen machen kann. Im nächsten Kapitel werden wir diese Theorie dann direkt auf eine reale Datenbank anwenden.

37.1 Einführung

37.1.1 Alles ist eine Tabelle

37-4

Wiederholung: Ein Entity-Relationship-Diagramm.



37-5

Alles ist eine Tabelle

Ein E/R-Diagramm beschreibt die *Struktur* von Daten. Hinter jedem Entitätstyp (»Schaf« und »Stall«) steht eine Tabelle: Die Spalten der Tabelle sind die Attribute dieses Typs. Die Zeilen der Tabelle sind die Entitäten dieses Typs. Eine Spalte bildet den primären Schlüssel.
Hinter jedem Relationshiptyp (»ist Mama von«, »schläft in«) steht ebenfalls eine Tabelle: Die Spalten der Tabelle sind die Primärattribute der Entitäten, die in Beziehung gesetzt werden. Die Zeilen der Tabelle repräsentieren Beziehungen zwischen den in der Zeile angegebenen Entitäten.

37.1.2 Relationen-Algebren

37-6

Die Relationen-Algebra.

Idee

Die Tabellen für Entitäten und für Relationships werden zwar unterschiedlich interpretiert, *mathematisch* und *speichertechnisch* gibt es aber *keinen Unterschied*. Man konzentriert sich deshalb darauf, *Tabellen besonders effizient* zu verwalten. Statt *Tabellen* spricht man mathematisch von *Relationen*. In einer *relationalen Datenbank* werden (wenig überraschend) genau solche *Relationen* verwaltet.
Merke: Die Datenbank »weiß nichts« von dem E/R-Modell; aus ihrer Sicht wird eine Menge gleichberechtigter Tabellen verwaltet.

37-7

Die Relationen-Algebra.

Was waren nochmal Relationen?

▶ **Definition**

Seien A_1 bis A_n Mengen. Eine *Relation* auf diesen Mengen ist eine Teilmenge von $A_1 \times \cdots \times A_n$.

Bemerkungen:

- Das *Kreuzprodukt* $A_1 \times \cdots \times A_n$ enthält gerade alle Tupel, deren erste Komponente aus A_1 stammt, deren zweite Komponente aus A_2 und so weiter.
- Eine *Teilmenge* dieses Kreuzprodukts ist also eine Menge von Tupeln, wobei jedes jeweils ein Element aus A_1 in Beziehung setzt mit einem Element aus A_2 und gleichzeitig einem aus A_3 und so weiter.
- Ist beispielsweise A_1 die Menge aller Namen, A_2 die Menge aller Farben und A_3 die Menge aller Fellarten, so ist eine Relation auf A_1 bis A_3 gerade eine Tabelle für den Entitätstyp Schaf.
- Es gibt zwei Unterschiede zwischen Tabellen und Relationen: Eine Relation kann jedes Tupel nur einmal enthalten und die Reihenfolge der Tupel ist egal.

Die Relationen-Algebra. 37-8

Operationen auf Relationen.

In einer Datenbank werden viele Relationen gespeichert, eine pro Entitätstyp und Relationshiptyp. Solche Relationen lassen sich nun *verknüpfen*, um neue Relationen zu bilden. Beispielsweise kann man aus der Relation aller Schafe durch Filterung die Relation aller schwarzen Schafe gewinnen. Die Klasse aller Relationen zusammen mit Operationen wie »Filterung« heißt mathematisch etwas hochtrabend *Relationen-Algebra*.

37.2 Operationen auf Relationen

37.2.1 Vereinigung und Schnitt

Die Operationen »Vereinigung« und »Schnitt«. 37-9

▶ Definition

Seien $A, B \subseteq A_1 \times \cdots \times A_n$ Relationen auf denselben Mengen. Dann heißt $A \cup B$ die *Vereinigung* von A und B und $A \cap B$ der *Schnitt* von A und B.

Beispiel

- Sei A die Relation, die nur schwarze Schafe enthält (also alle Tupel in der Schafstabelle, bei denen die Farbkomponente »schwarz« lautet).
- Sei B die Relation, die nur lockige Schafe enthält.
- Dann ist $A \cap B$ die Relation, die nur lockige, schwarze Schafe enthält.
- Dann ist $A \cup B$ die Relation, die alle Schafe enthält, die lockig oder schwarz sind.

37.2.2 Selektion

Die Operation »Selektion«. 37-10

▶ Definition

Seien $A \subseteq A_1 \times \cdots \times A_n$ eine Relation und P ein Prädikat auf Tupeln (also ein Test, der für jedes Tupel entweder wahr oder falsch ist). Dann ist die Selektion $\sigma_P(A) \subseteq A_1 \times \cdots \times A_n$ die Relation, die nur diejenigen Tupel aus A enthält, für die P gilt.

Beispiel

- $\sigma_{\text{Farbe}=\text{schwarz}}(\text{Schaf})$ ist die Relation aller schwarzen Schafe.
- $\sigma_{\text{Größe}>50}(\text{Stall})$ ist die Relation aller Ställe, die größer als 50 sind.
- Die folgenden Relationen sind gleich:
  1. $\sigma_{\text{Farbe}=\text{schwarz}}(\text{Schaf}) \cap \sigma_{\text{Fell}=\text{lockig}}(\text{Schaf})$
  2. $\sigma_{\text{Farbe}=\text{schwarz} \wedge \text{Fell}=\text{lockig}}(\text{Schaf})$

✎ Zur Übung 37-11

Sei A die Relation aller schwarzen Schafe, B die Relation aller lockigen Schafe und C die Relation aller kranken Schafe.
Versuchen Sie, mit Hilfe von Vereinigung, Schnitt und/oder Selektion folgende Mengen zu beschreiben:

1. Die Menge alle schwarzen Schafe, die lockig und krank sind.
2. Die Menge aller gesunden lockigen Schafe.

37.2.3 Projektion

37-12 Die Operationen »Projektion«.

▶ Definition
Seien $A \subseteq A_1 \times \cdots \times A_n$ eine Relation und seien A_{i_1} bis A_{i_j} einige dieser Mengen. Dann entsteht die *Projektion* $\pi_{A_{i_1},\ldots,A_{i_j}}(A) \subseteq A_{i_1} \times \cdots \times A_{i_j}$, indem man »nur die Spalten« A_{i_1} bis A_{i_j} betrachtet.

Beispiel

– Schaf =

Name	Farbe	Fell
Dolly	weiß	lockig
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig
Flauschi	beige	glatt

– $\pi_{\text{Farbe,Fell}}(\text{Schaf}) =$

Farbe	Fell
weiß	lockig
schwarz	wuschelig
beige	glatt

37.2.4 Kreuzprodukt

37-13 Die Operation »Kreuzprodukt«.

▶ Definition
Seien $A \subseteq A_1 \times \cdots \times A_n$ und $B \subseteq B_1 \times \cdots \times B_m$ Relationen. Dann ist das *Kreuzprodukt* $A \times B \subseteq A_1 \times \cdots \times A_n \times B_1 \times \cdots \times B_m$ die Relation, die alle Kombinationen von Elementen aus A und Elementen aus B enthält.

Beim Kreuzprodukt paart man alle Elemente der ersten Menge mit allen Elementen der zweiten Menge. Die Größe des Kreuzproduktes ist gerade das Produkt der Größen der Relationen A und B.

37-14 Beispiel für das Kreuzprodukt.
Gegeben seien folgende Relationen:

Name	Farbe	Fell
Dolly	weiß	lockig
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig

Schaf-Name	Stall-Nummer
Dolly	1
Peter	5

Ihr Kreuzprodukt lautet:

Name	Farbe	Fell	Schaf-Name	Stall-Nr.
Dolly	weiß	lockig	Dolly	1
Dolly	weiß	lockig	Peter	5
Flauschi	schwarz	wuschelig	Dolly	1
Flauschi	schwarz	wuschelig	Peter	5
Peter	schwarz	wuschelig	Dolly	1
Peter	schwarz	wuschelig	Peter	5

37.2.5 Join

Eine typische Anfrage. 37-15

- Nehmen wir an, wir wollen wissen, in *welchen Ställen schwarze Schafe schlafen*.
  Problem: Diese Information liefert uns *weder* die Schafrelation *noch* die *Schläft-in*-Relation.
- Wir müssen diese beiden Tabellen deshalb *vereinigen*.
  Problem: Das Kreuzprodukt setzt immer alles mit allem in Relation.
- Wir filtern deshalb nach solchen Paaren im Kreuzprodukt, bei denen der Schafname im ersten Teil des Tupels zum Schafname im zweiten Teil des Tupels passt.
  Problem: Dies liefert große Tupel, wir wollen aber nur die Stall-Nummer wissen.
- Dann projizieren wir auf die Stall-Nummer.

$$\pi_{\text{Stall-Nummer}}\big(\sigma_{\text{Name=Schaf-Name}}(\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) \times \text{Schläft-in})\big).$$

Die Anfrage Schritt für Schritt. 37-16

Die Ausgangstabellen.

Schaf =

Name	Farbe	Fell
Dolly	weiß	lockig
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig

Schläft-In =

Schaf-Name	Stall-Nr.
Dolly	1
Peter	5

Die Anfrage Schritt für Schritt. 37-17

Die gefilterte Schaftabelle.

$\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) =$

Name	Farbe	Fell
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig

Schläft-In =

Schaf-Name	Stall-Nr.
Dolly	1
Peter	5

Die Anfrage Schritt für Schritt. 37-18

Das Kreuzprodukt.

$\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) \times \text{Schläft-In} =$

Name	Farbe	Fell	Schaf-Name	Stall-Nr.
Flauschi	schwarz	wuschelig	Dolly	1
Flauschi	schwarz	wuschelig	Peter	5
Peter	schwarz	wuschelig	Dolly	1
Peter	schwarz	wuschelig	Peter	5

Die Anfrage Schritt für Schritt. 37-19

Die Selektion.

$\sigma_{\text{Name=Schaf-Name}}(\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) \times \text{Schläft-In}) =$

Name	Farbe	Fell	Schaf-Name	Stall-Nr.
Peter	schwarz	wuschelig	Peter	5

37-20

Die Anfrage Schritt für Schritt.

Die Projektion.

$$\pi_{\text{Stall}}\left(\sigma_{\text{Name=Schaf-Name}}\left(\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) \times \text{Schläft-In}\right)\right) =$$

Stall-Nr.
5

37-21

✎ Zur Übung

Beschreiben Sie mit Hilfe der bisherigen Operationen folgende Relation: Sie soll alle Paare enthalten von einem Schafsnamen zusammen mit der Größe des Stalls, in dem das Schaf schläft.
(Wem das zu einfach ist: Alle Paare von Schafsnamen zusammen mit der Größe des Stalls, in dem die Mutter des Schafs schläft.)

37-22

Die Operation »Join«.

Es kommt sehr oft vor, dass man zwei Relationen derart »zusammenfügen« möchte, dass nur diejenigen Tupel des Kreuzproduktes betrachtet werden sollen, bei denen zwei bestimmte Attribute gleich sind. Man nennt dies einen *Join* der Relationen *anhand* zweier Attribute.

▶ Definition

Sei $A \subseteq A_1 \times \cdots \times A_n$ und $B \subseteq B_1 \times \cdots \times B_m$. Sei $X = A_i = B_j$ für geeignete i und j. Dann ist der *Join* $\theta_X(A,B)$ die Menge $\sigma_{\text{alte } i\text{-Komponente=alte } j\text{-Komponente}}(A \times B)$.

Beispiel

Die Anfrage von vorhin lässt sich nun kürzer schreiben:

$$\pi_{\text{Stall}}(\theta_{\text{Name}}(\sigma_{\text{Farbe=schwarz}}(\text{Schaf}), \text{Schläft-In})).$$

Zusammenfassung dieses Kapitels

37-23

▶ Relationen

Eine *Relation* auf Mengen A_1 bis A_n ist eine Teilmenge des Kreuzprodukts dieser Mengen. Sie »entspricht« einer Tabelle, mit n Spalten und so vielen Zeilen wie es Tupel in der Relation gibt.

▶ Operationen auf Relationen

Seien $A, B \subseteq A_1 \times \cdots \times A_n$ Relationen.

- $A \times B$ ist das *Kreuzprodukt* der Relationen.
- $A \cup B$ ist die *Vereinigung* der Relationen.
- $A \cap B$ ist der *Schnitt* der Relationen.
- $\sigma_P(A)$ ist die *Selektion* anhand eines Prädikats P.
- $\pi_{A_i}(A)$ ist die *Projektion* auf ein Attribut.
- $\theta_{A_i}(A,B)$ ist der *Join* der Relationen anhand eines Attributs.

Übungen zu diesem Kapitel

Übung 37.1 Abfragen in der Relationenalgebra in vorgegebener Datenbank, mittel

In einem *Weblog* verfassen verschiedene Autoren Beiträge, die jeweils von anonymen Besuchern kommentiert werden können. Das Weblog »Just My 2 Cents« speist sich aus folgender Datenbank:

Autor =

Nickname	Vorname	Nachname
murmel	Till	Tantau
kracher	Johannes	Textor

Beitrag =

Nr.	Datum	Titel	Text	Nickname
1	2008-01-01	Hallo, Welt	Unser neuer Blog ist da!	kracher
2	2008-01-03	Grüsse	Frohes neues Jahr!	murmel
3	2008-01-05	Info B	Geht im SS 2008 los!	murmel
4	2008-04-04	Hmm ...	Lang nichts mehr geschrieben.	kracher

Kommentar =

BeitragNr.	Datum	Uhrzeit	Text
1	2008-01-01	20:01	Hammer!
1	2008-01-01	20:12	Gratuliere!
1	2008-01-02	11:11	Auch ich freu mich!
2	2008-01-04	13:10	Auch Dir frohes Neues ..
2	2008-01-04	14:10	Ich lehne Neujahr aus ethischen Gründen ab.
3	2008-01-06	10:11	Ob es genau so gut wird wie Info A?

Geben Sie Ausdrücke in der Relationenalgebra an, mit denen Sie folgende Informationen erhalten:

1. Alle Daten aller Beiträge
2. Datum und Titel aller Beiträge
3. Titel und Text aller Beiträge, die nach dem 31. Dezember 2007 verfasst wurden
4. Texte aller Kommentare, die am 4. Januar 2008 verfasst wurden
5. Texte aller Kommentare zu Beitrag Nummer 2, die nach 14 Uhr oder vor dem 1. Februar 2008 verfasst wurden
6. Texte, Daten und Uhrzeiten aller Kommentare zum Beitrag mit dem Titel »Hallo, Welt«
7. Daten aller Beiträge und der dazugehörigen Kommentare
8. Vor- und Nachnamen der Verfasser aller Beiträge, zu denen nach 20 Uhr Kommentare verfasst wurden

Übung 37.2 Abfragen in der Relationenalgebra in eigener Datenbank, mittel

Entwerfen Sie für Ihr Szenario aus Übung 35.1 zwei *sinnvolle* Anfragen, zu deren Beantwortung Sie Daten aus mindestens zwei Tabellen benötigen. Formulieren Sie diese Anfragen in der Syntax der Relationenalgebra. Welche Ergebnisse liefern diese Anfragen angewandt auf die Beispieldaten aus Übung 36.2?

38-1

Kapitel 38
SQL – Einfache Anfragen
Die strukturierte Anfragesprache

38-2

Lernziele dieses Kapitels

1. SQL-Anfragen an eine Datenbank formulieren können
2. Einfache SQL-Joins benutzen können
3. SQL-Funktionen benutzen können

Inhalte dieses Kapitels

38.1 Einfache Anfragen 326
38.1.1 Grundaufbau 326
38.1.2 Verbesserte Ausgaben 327
38.1.3 Joins 329

38.2 Anfragen für Fortgeschrittene 331
38.2.1 Prädikate 331
38.2.2 Funktionen 332

Worum es heute geht

In den vorherigen Kapiteln haben wir schon ein gesehen, wie man mit relationalen Datenbanken Daten modellieren kann und mit – mathematisch ganz vornehm – algebraischen Operatoren Dinge wiederfindet. Aber wie bringe ich nun der Maschine bei, dass ich gerne den Join $\theta_{A,B}(A)$ hätte? Ein Theta sucht man auf den meisten Tastaturen vergeblich. Kurz: Wie sage ich es meinem Computer?

In diesem ersten Kapitel zu SQL soll es darum gehen, wie man Anfragen an eine Datenbank syntaktisch korrekt formuliert. Dazu wird der Select-Befehl verwendet, der eine eierlegende Wollmilchsau ist. Wir haben ihn schon in der eher schlichten Version `select * from foo;` kennengelernt, – wahre Meister können aber mit einem mehrseitigen Select-Befehl Informationen aus einer Datenbank herauskitzeln, von denen Normalsterbliche gar nicht vermuten würden, dass sie überhaupt in der Datenbank drin sind. Beginnen werden wir mit einfachen Anfragen. Am Ende stehen dann schon recht komplexe Select-Befehle mit denen Sie dann auch wenigstens ein bisschen mitkitzeln können.

38.1 Einfache Anfragen

38.1.1 Grundaufbau

38-4

Formulierung einer Anfrage in SQL.
Anfragen haben generell folgende Form:

```
select A_1,...,A_n
from R_1,...,R_m
where P
```

Hierbei sind A_1 bis A_n Attribute, R_1 bis R_m Relationen und P ein Prädikat.
Der Effekt dieser Anfrage ist es, die folgende Relation auszugeben:

$$\pi_{A_1,\ldots,A_n}(\sigma_P(R_1 \times \cdots \times R_m))$$

Beispiele einfacher Anfragen. 38-5

– Welche schwarzen Schafe gibt es?

```
mysql> select name
         from schaf
         where farbe = "schwarz";
+----------+
| name     |
+----------+
| Flauschi |
| Peter    |
+----------+
```

– Welche schwarzen Schafe gibt es und welches Fell haben sie?

```
mysql> select name, fell
         from schaf
         where farbe = "schwarz";
+----------+-----------+
| name     | fell      |
+----------+-----------+
| Flauschi | wuschelig |
| Peter    | wuschelig |
+----------+-----------+
```

Beispiele einfacher Anfragen. 38-6

– Welche Ställe haben eine Größe von mindestens 50?

```
mysql> select nummer
         from stall
         where groesse >= 50;
+--------+
| nummer |
+--------+
|      1 |
+--------+
```

– Welche Stall-Entitäten haben eine Größe von mindestens 50?

```
mysql> select *
         from stall
         where groesse >= 50;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      1 |      50 |
+--------+---------+
```

38.1.2 Verbesserte Ausgaben

Beschränkung der Zeilenanzahl. 38-7

Möchte man wissen, was beispielsweise alles in der Tabelle `stall` steht, so geht dies sehr leicht mit folgender Anfrage:

```
mysql> select * from stall;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      1 |      50 |
|      5 |      25 |
+--------+---------+
```

Bei großen Tabellen (wie denen von Ensembl mit vielen Megabyte Inhalt), dauert dies aber viel zu lange. Der Trick ist nun, hinter einen **`select`**-Befehl eine *Limitierung* zu schreiben:

```
mysql> select * from stall limit 5;
```

Dies sorgt dafür, dass nur die ersten 5 Zeilen ausgegeben werden.

38-8

Beschränkung der Strings.

Man kann statt einem gesamten String wie einem Namen oder einer DNA-Sequenz auch nur einen Teil ausgeben lassen. Dazu schreibt man statt dem Attribut einfach `substring(n,s,l)`, wobei *n* der Name des Attributs ist, *s* das Anfangszeichen im String (mit 1 beginnend) und *l* die Länge des gewünschten Teilstrings ist.

```
mysql> select substring(name,1,3), farbe, fell from stall;
+--------------------+---------+-----------+
| substring(name,1,3) | farbe   | fell      |
+--------------------+---------+-----------+
| Dol                | weiss   | lockig    |
| Max                | schwarz | wuschelig |
| Pet                | schwarz | wuschelig |
+--------------------+---------+-----------+
mysql> select substring(name,2,3), farbe, fell from stall;
+--------------------+---------+-----------+
| substring(name,2,3) | farbe   | fell      |
+--------------------+---------+-----------+
| oll                | weiss   | lockig    |
| ax                 | schwarz | wuschelig |
| ete                | schwarz | wuschelig |
+--------------------+---------+-----------+
```

38-9

Sortierung der Ausgabe.

Oft möchte man die Ausgabe *sortieren*. (Normalerweise werden die Daten in einer nicht vorhersagbaren Reihenfolge ausgegeben.) Dazu kann man nach einem `select`-Befehl ein *Sortierungsattribut* angeben:

```
mysql> select * from stall order by groesse;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      5 |      25 |
|      1 |      50 |
+--------+---------+
```

Nach dem Attribut kann man noch `desc` angeben, um die Reihenfolge umzudrehen.

```
mysql> select * from stall order by nummer desc;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      5 |      25 |
|      1 |      50 |
+--------+---------+
```

38-10

✎ Zur Übung

Geben Sie den SQL-Befehl an, mit dessen Hilfe Sie die fünf größten Ställe angezeigt bekommen.

38.1.3 Joins

Zur Erinnerung 38-11

Will man Informationen aus zwei Relationen *zusammenfügen*, so benötigt man ein *Kreuzprodukt*, gefolgt von einer *Filterung*. Bei der Filterung streicht man alle Zeilen, in der zwei bestimmte gleichbenannte Attribute nicht gleich sind. Eine solche Verbindung nennt man auch einen *Join* anhand dieses Attributs.

Beispiele einer Kreuzproduktes. 38-12

Das Kreuzprodukt von Schafen und der Liegt-In-Relation.

```
mysql> select * from schaf, liegt_in;
+----------+---------+-----------+----------+--------+
| name     | farbe   | fell      | name     | nummer |
+----------+---------+-----------+----------+--------+
| Dolly    | weiss   | lockig    | Dolly    |      1 |
| Dolly    | weiss   | lockig    | Flauschi |      5 |
| Flauschi | schwarz | wuschelig | Dolly    |      1 |
| Flauschi | schwarz | wuschelig | Flauschi |      5 |
| Peter    | schwarz | wuschelig | Dolly    |      1 |
| Peter    | schwarz | wuschelig | Flauschi |      5 |
+----------+---------+-----------+----------+--------+
```

Die Ställe der schwarzen Schafe. 38-13

Die Anfrage $\pi_{\text{Stall-Nummer}}\big(\sigma_{\text{Name=Schaf-Name}}\big(\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) \times \text{Schläft-in}\big)\big)$ in SQL:

```
mysql> select nummer
         from schaf, liegt_in
         where
           farbe       = "schwarz" and
           schaf.name  = liegt_in.name;
+--------+
| nummer |
+--------+
|      5 |
+--------+
```

Eine komplexe Anfrage. 38-14

Welche Größe haben die Ställe der Schafe?

```
mysql> select schaf.name, groesse
         from schaf, liegt_in, stall
         where
           schaf.name   = liegt_in.name and
           stall.nummer = liegt_in.nummer;
+----------+---------+
| name     | groesse |
+----------+---------+
| Dolly    |      50 |
| Flauschi |      25 |
+----------+---------+
```

✎ **Zur Übung** 38-15

Nehmen wir an, die folgenden Befehle wurden ausgeführt und die Tabellen gefüllt.

```
create table schaf    (name char(20),
                       farbe char(20),
                       fell char(20));
create table liegt_in (name char(20), nummer integer);
create table stall    (nummer integer, groesse float);
create table mama     (name char(20), mutter char(20));

insert into ...;
```

1. Geben Sie einen SQL-Befehl an, um die Größe der Ställe zu bestimmen, in denen braune Schafe liegen.
2. Geben Sie einen SQL-Befehl an, um den Stall zu bestimmen, in dem die Mutter von Dolly schläft.

38-16

Erste Vereinfachung der Syntax bei Joins.

Da Joins so oft vorkommen, gibt es eine spezielle Syntax hierfür: Der Befehl

```
select ... from tabelle1 join tabelle2
                         using (A1, ..., An)
```

hat denselben Effekt wie

```
select ... from tabelle1, tabelle2
           where tabelle1.A1 = tabelle2.A1 and
                 ...
                 tabelle1.An = tabelle2.An and
```

Das Beispiel von oben schreibt sich dann so:

```
mysql> select schaf.name, groesse
         from schaf join liegt_in using (name)
                    join stall    using (nummer);
+----------+---------+
| name     | groesse |
+----------+---------+
| Dolly    |      50 |
| Flauschi |      25 |
+----------+---------+
```

38-17

Zweite Vereinfachung der Syntax bei Joins.

Beim Join von zwei Tabellen ist es »natürlich«, die Tabellen gerade anhand derjenigen Spalten zu »joinen«, die in beiden Tabellen vorkommen. Diesen **`natural join`** kann man direkt hinschreiben:

```
select ... from tabelle1 natural join tabelle2
```

hat denselben Effekt wie

```
select ... from tabelle1, tabelle2
           where tabelle1.A1 = tabelle2.A1 and
                 ...
                 tabelle1.An = tabelle2.An and
```

wobei `A1` bis `An` die Spalten sind, die in beiden Tabellen vorkommen.
Das Beispiel von oben schreibt sich dann so:

```
mysql> select name, groesse
         from schaf natural join liegt_in
                    natural join stall;
+----------+---------+
| name     | groesse |
+----------+---------+
| Dolly    |      50 |
| Flauschi |      25 |
+----------+---------+
```

38.2 Anfragen für Fortgeschrittene

38.2.1 Prädikate

Wie lautet die Prädikatsyntax?

38-18

Prädikate folgen in SQL dem Schlüsselwort **`where`**. Sie sind (ähnlich wie bei Java) boolesche Ausdrücke. Die Variablen in einem Prädikat sind gerade die Attribute einer Zeile. (Haben zwei Relationen `R_1` und `R_2` das gleiche Attribut `A`, so muss man `R_1.A` oder `R_2.A` schreiben.) Die Hauptunterschiede zu Java-Ausdrücken:

1. Gleichheit wird mit = statt mit == geprüft.
2. Strings lassen sich mittels <, <= usw. lexikographisch vergleichen.
3. Statt `!`, `||` und `&&` kann man (und tut man) auch **`not`**, **`or`** und **`and`** schreiben.
4. Mit **`like`** und **`regexp`** kann man komplexe Textvergleiche anstellen (dazu gleich mehr).

Beispiele für Prädikate.

38-19

```
mysql> select ... where
   farbe = "schwarz" and
   not(nummer < 5);
...
mysql> select ... where
  liegt_in.nummer = stall.nummer and
  not(farbe like "wei%");
...
```

Die drei Arten von Stringvergleichen in SQL.

38-20

In SQL gibt es drei Methoden, Strings zu vergleichen:

1. Ein *direkter lexikographischer Vergleich* mittels = oder >=.
2. Eine Art *Ähnlichkeitsvergleich* mittels `a like b`. Dieser Vergleich ist wahr, falls:
   - 2.1 Die Strings `a` und `b` gleich sind, wobei aber
   - 2.2 für jedes Vorkommen des Unterstrichs in `b` an der entsprechenden Stelle in `a` ein beliebiges Zeichen stehen darf und
   - 2.3 für jedes Vorkommen eines Prozentzeichens in `b` an der entsprechenden Stelle in `a` eine beliebige (auch leere) Zeichenfolge stehen darf.

   Ist beispielsweise `b` der String `H_llo`, so könnte `a` sowohl `Hallo` als auch `Hello` sein, nicht aber `Haallo`. Ist `b` der String `H%llo`, so könnte `a` nun `Haallo` sein, nicht aber `Halo`.
3. Ein Vergleich mit einem *regulären Ausdruck*.

Anwendung von regulären Ausdrücke in SQL.

38-21

Beispiel

Wir wollen in einer DNA-Sequenz nach einem Startkodon, gefolgt von beliebigen Basentriplets, gefolgt von einem Stoppkodon suchen. Zwei mögliche Schreibweisen für den Ausdruck:

1. `atg((a|t|g|c)(a|t|g|c)(a|t|g|c))*(tag|taa|tga)`
2. `atg([atgc]{3})*(tag|taa|tga)`

Anwendung in SQL

```
mysql> select * from dna
              where sequence regexp "atg([atgc]{3})*(tag|taa|tga)"
```

✎ Zur Übung

38-22

Geben Sie einen SQL-Befehl an, der aus der Tabelle `dna` alle Zeilen auswählt, bei denen das Attribut `sequence` alle der folgenden Eigenschaften hat:

1. Es kommt irgendwo `agu` vor.
2. Es kommt irgendwo `ccc` vor und später `aaa`.
3. Der String endet nicht mit `aaa`.

38.2.2 Funktionen

38-23

Neu: Zeilenweise Anwendung statt Projektionen.

▶ **Definition**

Seien $A \subseteq A_1 \times \cdots \times A_n$ eine Relation und seien $f_i \colon A_1 \times \cdots \times A_n \to B_i$ Funktionen. Dann ist die *zeilenweise Anwendung* dieser Funktionen wie folgt definiert:

$$\begin{aligned}\zeta_{f_1,\ldots,f_m}(A) &= \{(f_1(a),\ldots,f_m(a)) \mid a \in A\} \\ &\subseteq B_1 \times \cdots \times B_m.\end{aligned}$$

Beispiel

$\zeta_{\text{erstes Zeichen von(Name)},\text{Länge(Farbe)+Länge(Fell)}}(\text{Schaf}) =$

B_1	B_2
D	10
F	16
P	16

38-24

Die zeilenweise Anwendung in SQL.

Bei einem Select-Befehl kann man statt Attributen auch beliebige Funktionen angeben. Tatsächlich sind die normalen Projektionen auch nur Spezialfälle der zeilenweise Anwendung einer (sehr einfachen) Funktion. Welche Funktionen möglich sind, muss man in der Dokumentation nachschlagen. Einige Beispiele: `abs` für den Absolutbetrag ein Zahl, `sin` für den Sinus einer Zahl, `length` für die Länge eines Strings, **`substring`** für einen Teilstring, `concat` für die Verkettung mehrerer Strings.

```
mysql> select
  substring(name,1,1), length(farbe)+length(fell)
  from schaf;
```

Zusammenfassung dieses Kapitels

38-25

▶ **Grundsyntax von Anfragen**

Anfragen haben generell folgende Form:

```
select A_1,...,A_n
from R_1,...,R_m
where P
```

Hierbei sind A_1 bis A_n Attribute, R_1 bis R_m Relationen und P ein Prädikat.

▶ **Join-Syntax**

Joins kann man besonders einfach schreiben:

```
select A_1,...,A_n
from R_1 natural join R2 natural join R3 ...
where P
```

Dies liefert den Join der Relationen anhand der Spalten, die jeweils die Relationen mit den vorherigen gemeinsam haben.

▶ **Funktionen statt Projektionen**

Man kann statt Projektionen auch beliebige Funktionen zeilenweise anwenden.

39-1

Kapitel 39
SQL – Fortgeschrittene Anfragen

Wie viele?

Lernziele dieses Kapitels

1. SQL-Aggregate verstehen und benutzen können
2. SQL-Unteranfragen verstehen und benutzen können

39-2

Inhalte dieses Kapitels

39.1	Aggregate	334
39.1.1	Die Idee	334
39.1.2	Die Aggregatsfunktionen	334
39.1.3	Gruppierung	335
39.2	Unteranfragen (Subqueries)	337
39.2.1	Unteranfragen als Tabellen	337
39.2.2	Unteranfragen als Werte	338
39.2.3	Unteranfragen mit Quantoren	338

Worum es heute geht

Was uns die eleganten Theorie der relationalen Algebren nicht alles erlaubt zu ermitteln. Wer ist Dollys Vater? Select father from fathers where child = “Dolly.” Wie groß ist der Stall, in dem Dolly schläft? Select groesse from stall natural join liegt_in where name = “Dolly.” Was ist der Sinn Ihres Lebens? Select reason from persons natural join reasons_for_life where person = “ich”.

Was ist aber mit der Frage, »haben alle Schafe genug Platz«? Mit den bisherigen Mitteln der Theorie können wir diese Frage nicht beantworten, denn sie betrifft *nicht* einzelnen Zeilen, sondern eine Spalte als ganzes. Was wir bräuchten, ist eine Methode, die Werte einer Spalte zu »aggregieren« oder (weniger vornehme ausgedrückt) zusammen zu zählen. Genau dies erlaubt uns SQL: Wir können mit Hilfe von *Aggregaten* die Werte einer ganzen Spalten zusammenfassen – und dabei nicht nur die Summe bilden, sondern auch den Durchschnitt, die Standardabweichung, das Minimum oder Maximum bestimmen oder einfach nur die Anzahl zählen. Richtig interessant werden Aggregate dann, wenn wir nicht die ganze Spalte aggregieren, sondern nur alle Zeilen, die in anderen Spalten den gleichen Wert haben. (Hier wird es dann kompliziert…)

Hat man Aggregate erstmal zur Verfügung, so kommt man schnell auf den Geschmack, das Ergebnis einer Aggregation wieder in einer Anfrage zu nutzen: Mit einer Aggregation kann man zum Beispiel leicht die durchschnittliche Größe der Ställe bestimmen. Dann möchte man vielleicht alle Ställe finden, die größer als dieser Durchschnitt sind. Wir wollen also das *Ergebnis* einer Anfrage *innerhalb* einer anderen Anfrage nutzen. Hierfür bietet SQL die so genannten *Subqueries* (Unteranfragen) an. Diese sind ein sehr mächtiges Werkzeug, da man sie recht beliebig verschachteln kann. (Hier wird es dann auch kompliziert…)

39.1 Aggregate

39.1.1 Die Idee

39-4

Sind die Ställe groß genug?

Author Bob Jagendorf, Creative Commons Licence

Das Ziel

Die Firma *Molecular Sheep* möchte, dass ihre Schafen glücklich sind. Dazu möchte sie garantieren, dass nicht zu viele Schafe in einem Stall schlafen, jedes Schaf soll mindestens vier Quadratmeter Platz haben.

Das Problem

Es gibt keine Zeile in der Datenbank, in der steht, wie viele Schafe in einem Stall liegen. Auch mit einem Join lässt sich das Problem nicht lösen, da auch so keine *Zeile* erzeugt werden kann, die diese Zahl enthält. Wir brauchen eine Methode, *»etwas mit Spalten zu machen«*.

39-5

Selektion versus Aggregate.

Die Liegt-In-Tabelle:

Schafname	Stallnummer	Seit
Dolly	1	2012
Peter	1	2013
Max	2	2012
Ferdinand	2	2012
Flauschi	1	2013

Bei einer *Selektion* (in SQL: `where ...`) haben wir Zugriff auf

- alle Spalten *einer Zeile*.
- Da die Spalten feste Namen haben, können wir uns direkt auf diese beziehen.

Bei einer *Aggregation* wollen wir Zugriff auf

- alle Zeilen *einer Spalte*.
- Die Anzahl an Zeilen variabel ist, könne wir und nicht auf einzelnen Zeilen beziehen.

39.1.2 Die Aggregatsfunktionen

39-6

Die Syntax der einfachen Aggregation.

```
select Aggregatsfunktion(Spaltenauswahl)
  from ...
 where ...
```

Hierbei ist die `Aggregatsfunktion` eine der folgenden möglichen Funktion, die »Dinge aggregieren«:

- `avg(a)` liefert den Durchschnitt des Attributs `a`.
- `count(a)` liefert die Anzahl an Zeilen.
- `count(distinct a)` liefert die Anzahl an unterschiedlichen Zeilen (in Bezug auf `a`).
- `max(a)` und `min(a)` liefern das Maximum und das Minimum.
- `std(a)` und `stddev(a)` liefern beide die Standardabweichung.
- `sum(a)` liefert die Summe.

(Es gibt noch mehr Funktionen, man findet sie in der SQL-Referenz.)

39-7

Semantik (=Bedeutung) der einfachen Aggregation.

Ein Befehl der Form

```
select Aggregatsfunktion(Spaltenauswahl)
  from ...
 where ...
```

hat folgenden Effekt:

- Zunächst wird der `from ... where ...` Teil ganz normal ausgewertet. Hier kann es beliebig komplizierte Joins und Selektionen geben.

- Ebenso wird die `Spaltenauswahl` »ganz normal zeilenweise« ausgewertet. Normalerweise steht hier nur ein Attribut (ein Spaltenname), es können aber auch Ausdrücke wie `length(name)` hier stehen.
- Das Resultat ist dann eine neue Tabelle.
- In dieser Tabelle wird dann die Spalte `Spaltenauswahl` betrachtet und auf ihre Zeilen die angegebene Aggregatsfunktion angewandt.

Beispiele der einfachen Aggregation.

39-8

Beispiel: Der größte Stall

```
select max(groesse) from stall;
```

Beispiel: Die durchschnittliche Stallgröße

```
select sum(groesse) / count(groesse) from stall;
```

oder

```
select avg(groesse) from stall;
```

Beispiel: Die durchschnittliche Länge der Name von Schafen

```
select avg(length(name)) from schaf;
```

✎ Zur Übung

39-9

Nehmen wir an, die folgenden Befehle wurden ausgeführt und die Tabellen gefüllt.

```
create table schaf     (name char(20),
                        farbe char(20),
                        fell char(20));
create table liegt_in  (name char(20), nummer integer);
create table stall     (nummer integer, groesse float);
insert into ...;
```

Geben Sie einen SQL-Befehl an, der die maximale Größe von Ställen berechnet, in denen schwarze Schafe liegen.

39.1.3 Gruppierung

Aggregation von Teilen von Spalten.

39-10

Zurück zu unserem Ausgangsproblem: Jedes Schaf soll mindestens vier Quadratmeter Platz heben.
Für *einen konkrete Stallnummer* (wie »Stall 1«) können wir dies nun wie folgt überprüfen:

```
mysql> select count(name) * 4, max(groesse)
         from liegt_in natural join stall
        where nummer = 1;

+-----------------+--------------+
| count(name) * 4 | max(groesse) |
+-----------------+--------------+
|              12 |           50 |
+-----------------+--------------+
```

Wir wollen diese Daten aber für *alle* Ställe! Wir wollen also viele mit »`where nummer = ...`« erstellte Tabellen in eine Tabelle zusammenfassen. Dies nennt man *gruppieren*.

39-11

Die gruppierte Aggregation.

Syntax

```
select     Spalte1, Spalte2, ..., Aggregatsf.(Spalte)
     from ...
    where ...
 group by Spalte1, Spalte2, ...
```

Semantik

Zunächst wird wieder **from** ... **where** ... normal ausgeführt, was eine Tabelle T ergibt. In T kann es nun mehrere Zeilen geben, die in Bezug auf die Werte in den Spalten Spalte1, Spalte2, ... gleich sind. (Nur) diese Zeilen werden nun gemäß der Aggregatsfunktion zusammengefasst. Das Ergebnis bildet dann zusammen mit der Werten aus den ersten Spalte *eine* Zeile in der Ergebnistabelle.

39-12

Beispiele gruppierter Anfragen

Wie viele Schafe schlafen in einem Stall?

```
mysql> select nummer, count(name)
         from liegt_in
     group by nummer;

+--------+-------------+
| nummer | count(name) |
+--------+-------------+
|      1 |           3 |
|      2 |           2 |
+--------+-------------+
```

Größe versus Anzahl an Schafen in einem Stall?

```
mysql> select nummer, count(name)*4, max(groesse)
         from liegt_in natural join stall
     group by nummer;

+--------+---------------+--------------+
| nummer | count(name)*4 | max(groesse) |
+--------+---------------+--------------+
|      1 |            12 |           50 |
|      2 |             8 |            6 |
+--------+---------------+--------------+
```

39-13

Einschränkung der Ergebnisse einer Gruppierung.

Wir können mittels Gruppierung nun eine Tabelle erzeugen, die alle Informationen enthält, die wir benötigen. Nun wollen wir hieraus wiederum die *Zeilen* auswählen, bei denen die Größe der Ställe zu klein ist. Hierfür können wir aber *nicht* ein **where** ... verwenden: Ein **where** ... wird immer ausgewertet, *bevor* gruppiert wird –wir wollen aber *nach* der Gruppierung Zeilen streichen.

Syntax

```
select     Spalte1, Spalte2, ..., Aggregatsf.(Spalte)
     from ...
    where ...
 group by Spalte1, Spalte2, ...
   having ...;
```

Der Test nach **having** wird erst *nach* der Gruppierung ausgeführt und darf deshalb *aggregierte Werte* nutzen.

Das komplette Beispiel. 39-14

```
mysql> select nummer
         from liegt_in natural join stall
     group by nummer
       having count(name) * 4 > max(groesse);

+--------+
| nummer |
+--------+
|      2 |
+--------+
```

✎ Zur Übung 39-15
Geben Sie einen SQL-Befehl an, der alle Ställe ausgibt, in denen mehr als drei schwarze Schafe schlafen.

39.2 Unteranfragen (Subqueries)

39.2.1 Unteranfragen als Tabellen

Wie viele problematische Ställe gibt es? 39-16
Wir können mittlerweile eine Tabelle erstellen, die alle Ställe enthält, in denen zu viele Schafe liegen. Wir können aber noch *nicht* zählen, wie viele Zeilen diese Tabelle hat. Der Grund ist, dass wir dazu nach der Gruppierung und nach der Filterung mittels »having« *erneut* aggregieren wollen. Statt einem zweiten Group-by sieht SQL hierfür ein allgemeineres Konzept vor: Die *Unteranfrage* (»subquery«).

Unteranfragen als Tabellen 39-17
Wählt man mit `from ...` aus Tabellen etwas aus, so darf statt eines Tabellenamen auch stehen

```
(select ... ) as ein_name
```

Hierbei darf der `select`-Befehl beliebig kompliziert sein. Der *Effekt* ist, als ob man das Ergebnis des `select`-Befehls zuerst in eine temporäre Tabelle namens `ein_name` eingefügt hätte und diese dann normal genutzt hätte.

Beispiele von Unteranfragen als Tabellen. 39-18

Beispiel: Zwei äquivalente Anfragen

```
select name from (select *
                    from schafe
                   where farbe = "schwarz") as schwarze_schafe
            where fell = "lockig";
```

liefert dasselbe wie

```
select name from schafe
            where farbe = "schwarz" and
                  fell = "lockig";
```

Beispiel: Wie viele problematische Ställe gibt es?

```
select count(*)
  from (select nummer
          from liegt_in natural join stall
      group by nummer
        having count(name) * 4 > max(groesse)
        ) as problematisch;
```

✎ Zur Übung 39-19
Geben Sie eine SQL-Befehl an, um die *durchschnittliche Anzahl an Schafen pro Stall* zu bestimmen.

39.2.2 Unteranfragen als Werte

39-20

Unteranfragen als Werte

Eine besondere Art an Unteranfragen sind solche, die als Ergebnis ein Tabelle mit *genau einer Zeile und genau einer Spalte* liefern. In diesem Fall erlaubt es SQL, diesen einen Eintrag »wie einen normalen Wert« praktisch überall zu nutzen, wo man eben Werte nutzt.

Syntax von Werten aus Unteranfragen

An (fast) allen Stellen, wo in SQL ein Wert benötigt wird, kann statt dessen stehen

```
(select ...)
```

Voraussetzung ist, dass die Unteranfrage eine Tabelle mit genau einer Zeile und genau einer Spalte liefert.

Beispiel: Liste alle überdurchschnittlich großen Ställe

```
select * from stall
        where groesse > (select avg(groesse) from stall);
```

39.2.3 Unteranfragen mit Quantoren

39-21

Unteranfragen mit Quantoren

Unteranfragen werden häufig wie eben benutzt, um ein Attribut mit »dem Wert« zu vergleichen, den eine Unteranfrage liefert:

```
select name from liegt_in
           where seit = (select max(jahr)
                           from krank
                          where name = "Dolly");
```

Dies findet alle Schafe, die in einem Stall liegen genau seit dem letzten Jahr, in dem Dolly krank war. (Die Krankheitstabelle habe die Spalten »krank« und »name«.)
Wir wollen nun die Schafe finden, die in einem Stall liegen seit *irgendeinem* Jahr (und nicht dem letzten), in dem Dolly krank war. Hierzu bietet SQL eine besondere Syntax:

```
select name from liegt_in
           where seit = any (select jahr
                               from krank
                              where name = "Dolly");
```

Allgemein gilt: Schreibt man `... = any (select ...)`, so darf die Unteranfrage eine Spalte mit *mehreren* Zeilen zurückliefern. Der Ausdruck ist dann wahr, wenn *wenigstens eine Zeile* in der Tabelle gleich der linken Seite vom Gleichheitszeichen ist.

39-22

Allgemeine Syntax von Unteranfragen mit Quantoren

Any-Unteranfragen

- `... = any (select ...)` ist wahr, wenn die linke Seite gleich *wenigstens einer* Zeile der Unteranfrage ist.
- Statt = kann man auch <, <=, >, >= und <> (ungleich) schreiben mit der entsprechenden Bedeutung.

All-Unteranfragen

- `... = all (select ...)` ist wahr, wenn die linke Seite gleich *allen* Zeilen der Unteranfrage ist.
- Statt = kann man wieder auch <, <=, >, >= und <> schreiben.

Exists-Unteranfragen

`exists (select ...)` ist wahr, wenn die Unterfrage wenigstens eine Zeile zurückliefert.

Beispiele von Unteranfragen mit Quantoren

39-23

Beispiel: Welche Schafe haben keinen Stall?

```
select name
  from schaf
 where not exists (select * from liegt_in
                           where liegt_in.name = schaf.name);
```

Beispiel: Welche Schafe schlafen in einem Stall, in dem kein schwarzes Schaf schläft?

```
select name
  from liegt_in
 where nummer <> all (select nummer
                        from liegt_in natural join schaf
                       where farbe = "schwarz");
```

Zusammenfassung dieses Kapitels

▶ Aggregate und Gruppierungen

39-24

```
select     Spalte1, Spalte2, ..., Aggregatsfunktion(Spalte)
      from ...
     where ...
  group by Spalte1, Spalte2, ...
    having ...;
```

- Alle Zeilen, die die *gleichen Werte* haben in den bei **group by** angegebenen Spalten werden zusammengefasst.
- *Aggregatsfunktionen* beziehen sich dann gerade immer auf die Mengen von Zeilen, die zusammengefasst werden.
- **having** wählt Zeilen *nach* der Aggregation aus, **where** hingegen Zeilen *vor* der Aggregation.

▶ Unteranfragen als Tabellen

Bei **from** ... kann statt einem Spaltennamen auch stehen

```
(select ...) as name
```

Dies produziert eine temporäre Tabelle namens `name`.

▶ Unteranfragen als Wert

Statt einem Wert kann überall

```
(select ...)
```

stehen, wenn dies genau eine Zeile und Spalte produziert.

▶ Unteranfragen mit Quantoren

Vergleiche der Form

```
... = any (select ...)
```

sind wahr, wenn der Wert auf der linken Seite gleich irgendeiner Zeile in der rechten Tabelle ist.

- Statt **any** ist auch **all** möglich.
- Statt der Gleichheit sind auch andere Vergleiche erlaubt.
- Will man nur testen, ob die Unteranfrage nicht leer ist, so schreibt man einfach nur **exists** vor die Anfrage.

42-1

Kapitel 42
Biodatenbanken
Wohin mit den Genomen?

42-2

Lernziele dieses Kapitels

1. Überblick über das Feld der Biodatenbanken bekommen
2. Arten von Biodatenbanken kennen
3. Grenzen und Möglichkeiten von Biodatenbanken einschätzen können
4. Auf Daten in Beispieldatenbanken zugreifen können

Inhalte dieses Kapitels

42.1 Überblick 341
42.1.1 Welche Daten speichern Biodatenbanken? 342
42.1.2 Wie speichern sie die Daten? 342
42.1.3 Wie komme ich an die Daten ran? 342
42.2 Flat-Files: Fallbeispiel Protein-Data-Bank 344
42.3 SQL: Fallbeispiel Ensembl 345
42.4 XML: Fallbeispiel Kyoto Encyclopedia of Genes and Genomes 347

Worum es heute geht

Zugegeben: Die Physiker sind die uneinholbaren Weltmeister im Produzieren von Daten. Maschinen wie der Large Hedron Collider des CERN produzieren, während sie laufen, Daten im Terabyte-Bereich *jede Sekunde*. Jedoch kann sich die Biologie berechtigte Hoffnungen auf den Vizeweltmeistertitel machen: Die Fülle an Informationen über Genome, Spezies, Proteine, Pathways und was die Molekularbiologie noch so zu bieten hat, wächst exponentiell. Schon seit vielen Jahren ist es völlig undenkbar, Genomdaten einfach nur »in Büchern« aufzuschreiben; vielmehr werden die gesammelten Biodaten in *Biodatenbanken* gespeichert. Deren Anzahl und Umfang wächst seit etwa Anfang der 1990er Jahre exponentiell.

Wie in Pionierzeiten häufig, ist die »Szene« der Biodatenbank noch reichlich chaotisch und wenig konsolidiert. Viele der Datenbanken wurden im Rahmen einzelner Projekte aufgesetzt und dienten oft einzig dem Zweck, bestimmtes Datenmaterial »irgendwie« der Öffentlichkeit zugänglich zu machen. In vielen Fällen kann man einfach nur einen so genannten Flat-File herunterladen, der in einem Ad-Hoc-Format die Daten beschreibt. Einen mächtigen Zugriff auf die Daten etwa über SQL-Anfragen stellen nur wenige Biodatenbanken zur Verfügung.

42.1 Überblick

42-4

Was sind Biodatenbanken?

Eine *Biodatenbank* ist eine digitalisierte *Sammlung von biologischen Daten*. Es muss sich *nicht* um eine relationale Datenbank handeln; selbst eine Ansammlung von Textdateien kann eine Biodatenbank darstellen.

Liste der Biodatenabanken

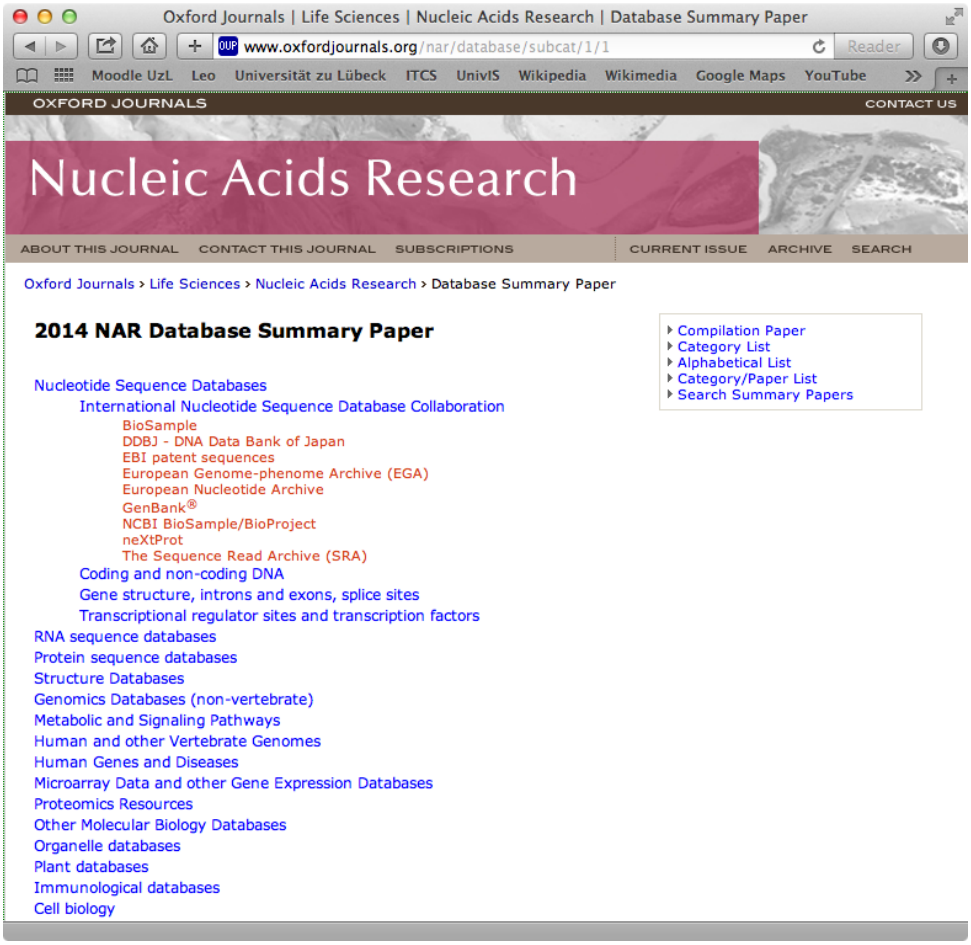
- 1000 Genomes Selection Browser
- 16S and 23S Ribosomal RNA Mutation Database
- 2D-PAGE
- 2P2Idb
- 3D rRNA modification maps

- 3D-Footprint
- ...
- ZiFDB
- ZInC

Es gibt viele Biodatenbanken.

42-5

Die Anzahl der Biodatenbanken ist so groß (über 1500 im Juni 2014), dass es eine Datenbank dieser Datenbanken gibt auf http://www.oxfordjournals.org/nar/database/cap:



42.1.1 Welche Daten speichern Biodatenbanken?

Klassifikation von Biodatenbanken.

42-6

Man kann Biodatenbanken klassifizieren nach der *Art der gespeicherten Daten*:

- DNA-Sequenz-Datenbanken
- RNA-Sequenz-Datenbanken
- Protein-Sequenz-Datenbanken
- Molekülstruktur-Datenbanken
- Gen-Datenbanken
- Genexpressions-Datenbanken
- Pathway-Datenbanken
- Publikations-Datenbanken

Von jedem Typ gibt es sehr viele Datenbanken ganz unterschiedlicher Art, Umfang und Zielsetzung.

42-7

Was speichert eine Biodatenbank?
Rohdaten versus aufbereitete Daten

Eine Biodatenbank kann zwei Arten von Dingen speichern:

- Die *Rohdaten*, die in Untersuchungen gewonnen wurden. Dies schließt Informationen über das *wer, wie, was, wo, warum* betreffend die Experimente ein.
  Beispiel: Bei einem Microarray-Experiment die Fotos des Arrays.
  Beispiel: Bei einer Sequenzierung die ermittelten Fragmente.
- *Aufbereitete Daten,* die aus den Rohdaten gewonnen wurden.
  Beispiel: Bei einem Microarray-Experiment verschiedene ermittelte Gencluster.
  Beispiel: Bei einer Sequenzierung der prognostizierte »Golden Path«.

42-8

Aus welchen Quellen speist sich eine Biodatenbank?
Rohquellen versus aufbreitete Quellen

Die Daten in einer Biodatenbank können auf zwei Arten gesammelt werden:

- Forschungsgruppen *»submitten«* Daten zu einem Thema bei einer Biodatenbank, die diese dann allen Interessenten zugänglich macht.
- Die Betreiber der Biodatenbank *sammeln* die Daten bei verschiedenen Forschungsgruppen regelmäßig nach eigenen Kriterien ein, bereiten diese auf und stellen dann das Ergebnis allen Interessenten zur Verfügung.

✎ Zur Diskussion
Welche Vor- und Nachteile haben die Verfahren?

42.1.2 Wie speichern sie die Daten?

42-9

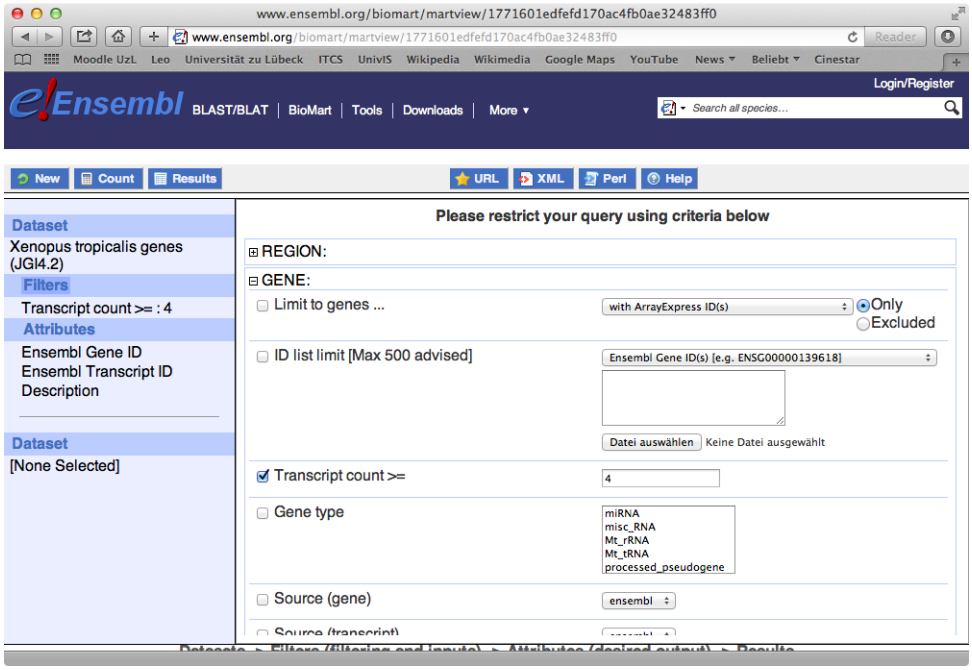
Arten, wie Daten in Biodatenbanken gespeichert werden.
Die in einer Biodatenbank gespeicherten Daten können unterschiedlich stark strukturiert sein. Mögliche *Datenmodelle* sind:

1. *Reiner Text:* Beispiele sind Sammlungen von Artikeln zu einem Thema.
2. *Entry-basiert:* Lange Listen von Zeilen (Entries), die eine spezielle Syntax haben (beispielsweise mit einem Schlüsselwort wie `ATOM` oder `REMARK` anfangen).
3. *Relationale:* Die Datenbank ist eine SQL-Datenbank mit einem Entity-Relationship-Schema.
4. *Objektorientiert:* Die Datenbank ist eine OO-Datenbank.
5. *XML:* Die Daten werden als XML-Dateien entsprechend einem bestimmten Schema gespeichert.

42.1.3 Wie komme ich an die Daten ran?

42-10

Wer kann wie auf die Daten zugreifen?
Bei *öffentlichen* Biodatenbanken kann erstmal jeder auf die Daten zugreifen. Es gibt in der Regel ein *Web-Interface,* über das man Daten finden kann und oft auch komplexere Anfragen stellen kann. Unter Umständen gibt es auch einen direkten Zugang zu einem SQL-Server.



Interne Speicherung versus externer Zugriff

42-11

Jede Biodatenbank speichert Daten intern auf eine bestimmte Art. Bei einem *externen Zugriff* kann man auf die Daten aber meist auf mehrere Weisen herunterladen; die Daten werden »zur Not umgerechnet«. Zur Verfügung stehen:

- Flat-Files,
- XML-Dateien und machmal
- SQL-Zugriff auf die Datenbank.

Biodaten als Flat-Files

42-12

Ein *Flat-File* ist eine einfache Textdatei, in der jede Zeile einen kleinen Datensatz darstellt. Es gibt keinerlei Standard für die Formate, jede Biodatenbank nutzt typischerweise selber mehrere »selbst ausgedachte« Formate.

```
HEADER    HYDROLASE/HYDROLASE INHIBITOR          01-SEP-11   3TNS
TITLE     CRYSTAL STRUCTURE OF SARS CORONAVIRUS MAIN PROTEASE COMPLEXED WITH AN
TITLE    2 ALPHA, BETA-UNSATURATED ETHYL ESTER INHIBITOR SG83
...
ATOM      1  N   SER A   1      23.652   4.764 -24.058  1.00 24.36           N
ANISOU    1  N   SER A   1     2645   3238   3373   -593    126   -557       N
ATOM      2  CA  SER A   1      22.698   5.744 -23.500  1.00 24.91           C
ANISOU    2  CA  SER A   1     2800   3292   3372   -616    100   -508       C
```

Biodaten als XML-Dateien

42-13

Bei *XML* handelt es sich um einen Standard, *wie man Daten strukturiert und aufschreibt*. (Siehe auch das Kapitel hierzu.) Man kann mit einer *Document Type Definition* genau und recht »sauber« festlegen, welche Tags in einer Datei vorkommen dürfen und was sie bedeuten. Generell lassen sich XML-Dateien von Computern leichter als Flat-Files bearbeiten, sind aber eher groß.

```
<?xml version="1.0" encoding="UTF-8" ?>
<PDBx:datablock datablockName="3TNS"...>
   <PDBx:atom_siteCategory>
      <PDBx:atom_site id="1">
         <PDBx:B_iso_or_equiv>24.36</PDBx:B_iso_or_equiv>
         <PDBx:Cartn_x>23.652</PDBx:Cartn_x>
         <PDBx:Cartn_y>4.764</PDBx:Cartn_y>
         <PDBx:Cartn_z>-24.058</PDBx:Cartn_z>
         <PDBx:auth_atom_id>N</PDBx:auth_atom_id>
         <PDBx:auth_comp_id>SER</PDBx:auth_comp_id>
         ...
      </PDBx:atom_site>
      <PDBx:atom_site id="2">
         <PDBx:B_iso_or_equiv>24.91</PDBx:B_iso_or_equiv>
         <PDBx:Cartn_x>22.698</PDBx:Cartn_x>
         <PDBx:Cartn_y>5.744</PDBx:Cartn_y>
         <PDBx:Cartn_z>-23.500</PDBx:Cartn_z>
         <PDBx:auth_atom_id>CA</PDBx:auth_atom_id>
         ...
```

42-14

Zugriff auf SQL-Datenbanken

- Bei den meisten Biodatenbank steht heutzutage »zumindest im Hintergrund« eine SQL-Datenbank.
- Manche Biodatenbanken geben jedem sofort (Ensembl) oder zumindest auf Anfrage einen direkten Lese-Zugriff auf die Datenbank.

```
> mysql --host=ensembldb.ensembl.org --user=anonymous
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3872065
...
mysql> use xenopus_tropicalis_core_75_42;
```

42.2 Flat-Files: Fallbeispiel Protein-Data-Bank

42-15

Die Protein-Data-Bank: www.rcsb.org

Die Protein-Data-Bank wird von den Universitäten Rutgers und UCSD betrieben. Sie speichert *Molekülstrukturdaten,* die beispielsweise aus NMR-Spektroskopien entstanden sind. Der Zugriff erfolgt in zwei Schritten:

1. Über das Webinterface und dessen Suchfunktionen findet man die Identifikationsnummer (ID) des interessierenden Proteins.
2. Mit dieser Nummer kann man dann die Struktur des Proteins als Flat-File oder als XML-Datei herunterladen.

Alternativ kann man auch gleich »im Browser« das Molekül betrachten.

42-16

Das Flat-File-Format der Protein-Data-Bank

```
HEADER    HYDROLASE/HYDROLASE INHIBITOR           01-SEP-11   3TNS
TITLE     CRYSTAL STRUCTURE OF SARS CORONAVIRUS MAIN PROTEASE COMPLEXED WITH AN
TITLE    2 ALPHA, BETA-UNSATURATED ETHYL ESTER INHIBITOR SG83
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: SARS CORONAVIRUS MAIN PROTEASE;
...
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: SARS CORONAVIRUS;
SOURCE   3 ORGANISM_COMMON: SARS-COV;
...
KEYWDS    3C-LIKE PROTEASE, PROTEASE, HYDROLASE-HYDROLASE INHIBITOR COMPLEX
EXPDTA    X-RAY DIFFRACTION
AUTHOR    L.ZHU,R.HILGENFELD
REVDAT   1   05-SEP-12 3TNS    0
JRNL        AUTH   L.ZHU,R.HILGENFELD
JRNL        TITL   CRYSTAL STRUCTURES OF SARS-COV MAIN PROTEASE COMPLEXED WITH
JRNL        TITL 2 A SERIES OF PEPTIDIC UNSATURATED ESTERS
...
ATOM      1  N   SER A   1      23.652   4.764 -24.058  1.00 24.36           N
ANISOU    1  N   SER A   1     2645   3238   3373   -593    126   -557       N
ATOM      2  CA  SER A   1      22.698   5.744 -23.500  1.00 24.91           C
ANISOU    2  CA  SER A   1     2800   3292   3372   -616    100   -508       C
```

42-17

Wie das Format funktioniert

```
HEADER    HYDROLASE/HYDROLASE INHIBITOR           01-SEP-11   3TNS
TITLE     CRYSTAL STRUCTURE OF SARS CORONAVIRUS MAIN PROTEASE COMPLEXED WITH AN
TITLE    2 ALPHA, BETA-UNSATURATED ETHYL ESTER INHIBITOR SG83
...
ATOM      1  N   SER A   1      23.652   4.764 -24.058  1.00 24.36           N
```

Das Flat-File-Format der PDB folgt einer festen Struktur

- Jede Zeile hat maximal 80 Zeichen und enthält keine Sonderzeichen und keine Umlaute.
- Die ersten sechs Zeichen jeder Zeile spezifizieren den »Zeilentyp«. Beispielsweise ist der Typ der ersten Zeile oben `HEADER`, der der zweiten `TITLE`, später kommt dann ein `ATOM`.

Beispiele, wie Zeilentypen spezifiziert werden.

42-18

Für jeden Zeilentyp schreibt das Format dann genau vor, was danach kommen darf.

Beispiel: Der Zeilentyp `HEADER`

Spalten	Datentyp	Beschreibung
1–6	Zeilentyp	Hier muss `RECORD` stehen
11–50	String	Klassifikation des Moleküls
51–59	Datum	Upload-Datum im Format DD-MMM-YY
63–67	Code	PDB-ID

Beispiel: Der Zeilentyp `TITLE`

Spalten	Datentyp	Beschreibung
1–6	Zeilentyp	Hier muss `TITLE` stehen
9–10	Zahl	Laufende Nummer (mehrzeiligen Titel)
11–80	String	Der Titel des Experiments

Beispiel: Der Zeilentyp `ATOM`

Spalten	Datentyp	Beschreibung
1–6	Zeilentyp	Hier muss `ATOM` stehen
7–11	Zahl	Laufende Atomnummer (5′ beginnt).
13–16	Name	Atomname.
17	Zeichen	Indikator für alternativen Ort
18–20	String	Aminosäurenname
22	Zeichen	Ketten-ID
23–26	Zahl	Aminosäuren-Serienummer
27	Zeichen	Einfügecode für Säuren
31–38	Zahl	x-Koordinate in Ångström
39–46	Zahl	y-Koordinate in Ångström
47–54	Zahl	z-Koordinate in Ångström
55–60	Zahl	Occupancy
61–66	Zahl	Temperaturfaktor
77–78	String	Elementsymbol, rechtsbündig
79–80	String	Ladung

42.3 SQL: Fallbeispiel Ensembl

Die Biodatenbank Ensembl

42-19

> *Ensembl is a joint project between EMBL-EBI (European Molecular Biology Laboratory, The European Bioinformatics Institute) and the Wellcome Trust Sanger Institute to develop a software system which produces and maintains automatic annotation on selected eukaryotic genomes.*

Die Biodatenbank Ensembl stellt Genomdaten ausgewählter Spezies zur Verfügung. Neben den Genomsequenzen sind Gendaten, deren Transkriptionen, Exons und vieles mehr gespeichert.
Zugriff ist auf viele Arten möglich:

1. Es gibt eine Webseite, auf der die Daten direkt durchsucht werden können.
2. Es werden auf der Webseite auch ein Reihe von speziellen Analyse-Tools angeboten.
3. Schließlich ist ein anonymer SQL-Zugang möglich.

42-20

Die Datenbank-Schemata von Ensembl.

Zentrale Schemata

In Ensembl gibt es vier Hauptschemata:

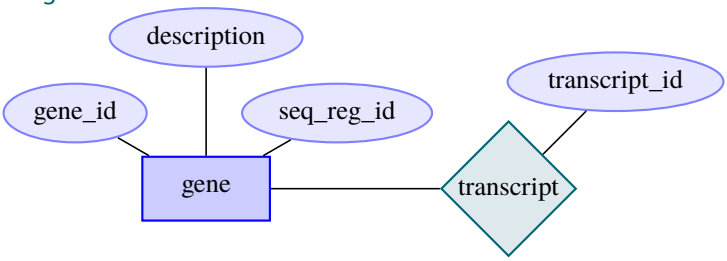
- Der »Kern«, der Genomdaten und Gene speichert.
- »Vergleichende Genomik«, in der Homologe, Paraloge und Proteinfamilien gespeichert sind.
- »Variationen«, in der SNP-Varianten, somatische Mutationen und Strukturvarianten gespeichert sind.
- »Regulation«, in der regulatorische Motive gespeichert sind.

Viele Datenbanken

In Ensembl gibt es pro Spezies mehrere Datenbanken. So wird immer, wenn genügend neue Daten gesammelt wurden, ein neuer »Build« erstellt und dieser in einer neuen Datenbank zur Verfügung gestellt.

42-21

Ein ganz kleiner Ausschnitt des Kernschemas



42-22

Ein typische Frage, die Ensembl beantworten kann.

Die zu beantwortende Frage

Welche für die *Phosporylasekinase* zuständigen Gene haben bei *Xenopus tropicalis* mehr als eine Transkription?

Schritt 1: Auswahl der richtigen Datenbank

```
> mysql --host=ensembldb.ensembl.org --user=anonymous
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3872065
...
mysql> use xenopus_tropicalis_core_75_42;
```

42-23

Schritt 2: Erstellung der Anfrage

✎ Zur Übung

Geben Sie SQL-Befehle an, die jeweils folgendes leisten:

1. Alle mit *Phosporylasekinase* befassten Gene auswählen.
2. Einen Join der Transkriptionstabelle mit der obigen Tabelle erstellen.
3. Die Anzahl der Transkriptionen pro Gen zählen, wobei die Ausgabespalten sein sollen: Anzahl der Transkriptionen, Gene-ID und Gen-Beschreibung.
4. Die Einschränkung der obigen Ausgabe auf Gene, die mindestens zwei Transkriptionen haben.

42-24

Die Anfrage

```
select    count(transcript_id) as c, gene_id, gene.description
     from transcript join gene using (gene_id)
    where gene.description like "%phosphorylase_kinase%"
 group by gene_id, gene.description
   having count(transcript_id) > 1;

+---+---------+-----------------------------------------------+
| c | gene_id | description                                   |
+---+---------+-----------------------------------------------+
| 2 |    7298 | phosphorylase kinase, alpha 2 (liver) ...     |
| 2 |    8294 | calmodulin 1 (phosphorylase kinase, delta) ...|
+---+---------+-----------------------------------------------+
```

Wie man Ensembl »wirklich« nutzt.

42-25

In der Praxis wird man nur sehr selten SQL-Anfragen »per Hand« erstellen. Vielmehr werden diese Anfragen *von Programmen* erstellt, die dann mit der Datenbank »sprechen«. Solche Programme kann man in Java schreiben und dann den *Java Database Connector* nutzen oder in Perl (eine Programmiersprache) und dort entsprechende Klassen und Methoden.

42.4 XML: Fallbeispiel Kyoto Encyclopedia of Genes and Genomes

Die Biodatenbank Kyoto Encyclopedia of Genes and Genomes.

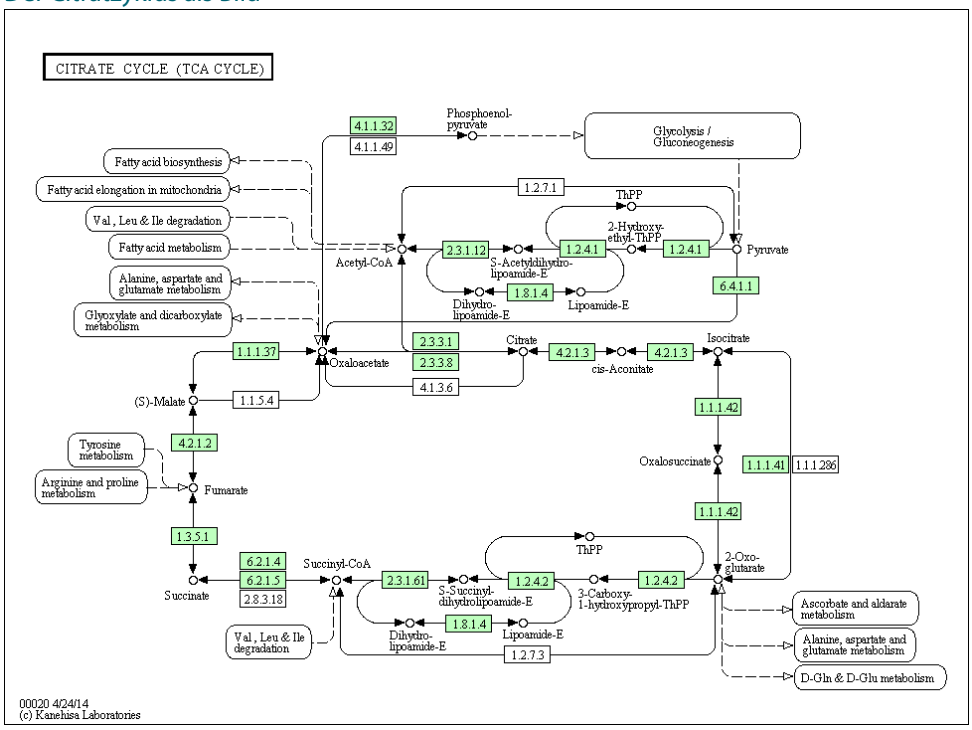
42-26

Die KEGG-Biodatenbank speichert Gene und besonders *Pathways*. Die Pathways werden auch graphisch dargestellt und können durchsucht werden.
Zugriff ist wieder auf verschiedene Arten möglich:

1. Auf der Webseite kann man die Pathways anschauen und durch sie »mit der Maus« navigieren.
2. Die Pathways lassen sich auch als XML-Dateien herunterladen.

Der Citratzyklus als Bild

42-27



Der Citratzyklus als XML-Datei

42-28

Das KEGG stellt Pathways auch in der KEGG *Markup Language* KGML zur Verfügung:

```
<?xml version="1.0"?>
<!DOCTYPE pathway SYSTEM "http://www.kegg.jp/kegg/xml/KGML_v0.7.1_.dtd">
<pathway name="path:hsa00020" org="hsa" number="00020"
         title="Citrate␣cycle␣(TCA␣cycle)" ...>
    <entry id="28" name="ko:K00174␣ko:K00175␣ko:K00177␣ko:K00176"
         type="ortholog" reaction="rn:R01197" ...>
       <graphics name="K00174..." fgcolor="#000000"
             bgcolor="#FFFFFF" type="rectangle" x="526" y="649"
             width="46" height="17"/>
    </entry>
...
    <reaction id="92" name="rn:R00431␣rn:R00726" type="irreversible">
        <substrate id="60" name="cpd:C00036"/>
        <product id="94" name="cpd:C00074"/>
    </reaction>
</pathway>
```

Zusammenfassung dieses Kapitels

42-29

▶ Was sind Biodatenbanken?

Eine *Biodatenbank* ist eine digitalisierte *Sammlung von biologischen Daten*.

▶ In welchen Formaten stellen Biodatenbanken ihre Inhalte zur Verfügung?

- Flat-Files
- XML-Dateien
- SQL-Zugriff oder SQL-Dumps

▶ Was speichern Biodatenbanken?

- DNA-Sequenz-Datenbanken
- RNA-Sequenz-Datenbanken
- Protein-Sequenz-Datenbanken
- Molekülstruktur-Datenbanken
- Gen-Datenbanken
- Genexpressions-Datenbanken
- Pathway-Datenbanken
- Publikations-Datenbanken

Zum Weiterlesen

[1] Nucleotide Acids Research, *Database Summary Paper,* http://www.oxfordjournals.org/nar/database/cap, Zugriff Juni 2014

Diese Webseite stellt eine jeweils aktuelle Liste aller Biodatenbanken zur Verfügung.

Teil IX
Sicherheit

Über manche Dinge sollte man besser schweigen. Stellen Sie sich vor, ihr Computer teilt freudig anderen Computern alles mit, was auf Ihrer Festplatte so steht. Auf dieser finden sich höchstwahrscheinlich Informationen über Ihre E-Mail, Ihre Passwörter und vermutlich auch über Ihre Liebhaber (definitiv aber darüber, wen Sie gerne als Liebhaber hätten). Aus diesem Grund ist *Computersicherheit* ein wichtiges Thema. Dabei geht es darum, Zugriff durch Unbefugte zu verhindern, aber auch darum, wie man Daten ganz banal vor, sagen wir, Feuer schützt. Eines der spannendsten Teilgebiete der IT-Sicherheit ist die Kryptographie, also die Frage, wie man Daten verschlüsselt. Dieses Gebiet ist nicht nur deshalb spannend, weil es nach Geheimdiensten und Spionagethrillern klingt, sondern weil es auch wirklich schöne Anwendungen von reiner Mathematik in der Praxis darstellt.

43-1

Kapitel 43
Kommunikations-Sicherheit
Verschlüsselung: Der digitale Briefumschlag

43-2

Lernziele dieses Kapitels

1. Konzept der Verschlüsselung verstehen
2. Unterschied zwischen symmetrischen und asymmetrischen Verfahren kennen
3. Konzept der digitalen Unterschrift verstehen
4. Programme zur Verschlüsselung einsetzen können
5. Zertifikate erstellen und installieren können

Inhalte dieses Kapitels

43.1	Ziele von IT-Sicherheit	351
43.2	Verschlüsselung	351
43.2.1	Ziele	351
43.2.2	Symmetrische Verschlüsselung	351
43.2.3	Asymmetrische Verschlüsselung	353
43.2.4	Das El-Gamal-Verfahren	353
43.3	Sichere E-Mail	355
43.3.1	Vertraulichkeit: Digitale Briefumschläge	355
43.3.2	Authentizität: Digitale Unterschriften	358
43.3.3	Echtheits-Zertifikate: Digitale Notare	360
43.4	Sicheres Surfen	363
	Übungen zu diesem Kapitel	364

Worum es heute geht

Verschlüsselung von Daten gehört seit der Antike zu den Grundmethoden der Kriegsführung. Eine sehr alte (und sehr einfache) Verschlüsselungsmethode ist beispielsweise die Cäsar-Chiffre, bei der einfach jeder Buchstabe durch einen Buchstaben etwas weiter hinten im Alphabet ersetzt wird. Ob sich Julius Cäsar dieses Verfahren allerdings selbst ausgedacht hat? Berthold Brechts lesender Arbeiter würde sich sicherlich fragen: »Cäsar eroberte ganz Gallien. Hatte er nicht wenigstens einen Kryptographen dabei?«

Copyright Austin Mill. GNU Free Documentation License

Im zweiten Weltkrieg wurde erstmals im großen Stil Kryptographie *technisiert*, insbesondere in Form der *Enigma*. Es entwickelte sich ein Katz-und-Maus-Spiel, bei dem die deutschen Truppen ihre Enigma kryptographisch immer sicherer machten, während die Alliierten ihre *Kryptoanalyse* gleichzeitig immer weiter verbesserten. Die meisten mit Hilfe der Enigma verschlüsselten Funksprüche der deutschen Truppen konnten von den Alliierten mitgelesen werden. Zur Bedeutung des Brechens des Enigma-Codes schreibt Wikipedia: »Die Kompromittierung der Enigma wird als ein strategischer Vorteil angesehen, der den Alliierten den Gewinn des Krieges erheblich erleichtert hat. Es gibt sogar Historiker, die diese Tatsache für kriegsentscheidend halten, denn die Entzifferungen waren nicht nur auf militärisch-taktischer Ebene (Heer, Luftwaffe und Marine) eine große Hilfe, sondern sie erlaubten aufgrund der nahezu vollständigen Durchdringung des deutschen Nachrichtenverkehrs auf allen Ebenen (Polizei, Geheimdienste, diplomatische Dienste, SD, SS, Reichspost und Reichsbahn) auch einen genauen Einblick in die strategischen und wirtschaftlichen Planungen der deutschen Führung. Speziell schätzten die Alliierten die Authentizität der aus Enigma-Funksprüchen gewonnenen Informationen, die aus anderen Quellen, wie Aufklärung, Spionage oder Verrat, nicht immer gegeben war. So konnten die Briten ihre zu Beginn des Krieges noch begrenzten Ressourcen optimal koordinieren und gezielt gegen die deutschen Schwächen einsetzen, und später, zusammen mit ihren amerikanischen Verbündeten, die Überlegenheit noch besser ausspielen.«

Einer der Hauptgründe, weshalb der Enigma-Code gebrochen werden konnte, war, dass das folgende schon 1883 von Kerckhoffs formulierte Prinzip nicht eingehalten wurde:

> Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Algorithmus abhängen, sie gründet sich nur auf die Geheimhaltung des Schlüssels.

Die heute verwendeten Verschlüsselungsalgorithmen haben diese Schwachstelle nicht, man kann die Algorithmen in jedem Lehrbuch nachlesen. Trotzdem sei darauf hingewiesen, dass ihre Sicherheit nur *vermutet* wird – es gibt keinen Beweis, dass irgend eines der heute eingesetzten Standardverfahren wirklich sicher ist.

43.1 Ziele von IT-Sicherheit

Worum geht es bei IT-Sicherheit? 43-4

Bei *IT-Sicherheit* geht es um folgende Anliegen:

1. Schutz vor und die Aufrechterhaltung des Betriebs bei
   - Ausfall von Teilen des Systems (Stromausfall, Absturz)
   - Angriffen auf das System (durch Hacker, korrumpierte Mitarbeiter)

   Diese *Systemsicherheit* wird uns im nächsten Kapitel interessieren.
2. Schutz von Daten und Kommunikation vor
   - Spionage
   - Fälschung

   Diese *Daten- und Kommunikationssicherheit* wird uns in diesem Kapitel interessieren.

43.2 Verschlüsselung

43.2.1 Ziele

Ziele der Verschlüsselung von Daten und Kommunikation 43-5

Vertraulichkeit Es muss sichergestellt werden, dass Daten und Kommunikation nur von »den Guten« gelesen werden können.
(Meine Daten gehen niemand etwas an.)

Integrität Es muss sichergestellt werden, dass Daten und Kommunikation nicht verfälscht werden können.
(Aus 1000 Euro dürfen nicht 10000 Euro werden.)

Authentizität Es muss sichergestellt werden, dass Daten und Kommunikation wirklich von den behaupteten Personen stammen.
(Die Email mit Alices Absenderadresse wurde tatsächlich von Alice versandt; der Online-Banking-Server muss wirklich der Server meiner Bank sein.)

✎ Zur Übung

Beurteilen Sie Postkarten, versiegelte Briefe und die Eröffnung eines Bankkontos bei einer Online-Bank in Bezug auf die drei Kriterien.

43.2.2 Symmetrische Verschlüsselung

43-6

Symmetrisch Verschlüsselung mittels eines Schlüssels

- Zum Schutz vor unbefugtem Lesen kann man Daten und Kommunikation *verschlüsseln*. Dazu benutzt man ein *Verschlüsselungsverfahren* sowie einen *geheimen Schlüssel*.
- Beim *Verschlüsseln* (encryption) wird ein *Klartext* m (message) zusammen mit dem *Schlüssel* k (key) in ein *Chiffretext* $c = e(m,k)$ verwandelt.
  Dies entspricht dem »Abschließen« mit dem Schlüssel.
- Beim *Entschlüsseln* (decryption) wird der Chiffretext zusammen mit dem Schlüssel in den Klartext zurückverwandelt, d.h. $m = d(c,k)$.
  Dies entspricht dem »Aufschließen« mit dem Schlüssel.
- Da man zum »Auf- und Zuschließen« denselben Schlüssel verwendet, spricht man von *symmetrischen Verfahren*.

43-7

Verfahren I: Die Cäsar-Chiffre

Bereits in der Antike nutzte Julius Cäsar Verschlüsselungen, wenn er Befehle an seine Feldherren übermittelte. Das als *Cäsar-Chiffre* bekannte Verfahren funktioniert wie folgt: Jeder Buchstabe der Nachricht wird durch den Buchstaben ersetzt, der *drei Buchstaben später im Alphabet* kommt. Allgemeiner kann man statt »drei Buchstaben später« auch »k Buchstaben später« benutzen. Mathematisch ist also e die Funktion, die eine Nachricht und eine Zahl k nimmt und jeden Buchstaben der Nachricht um k viele Stellen im Alphabet vorwärts schiebt. Entsprechend schiebt d jeden Buchstaben um k viele Stellen zurück.

✎ Zur Diskussion

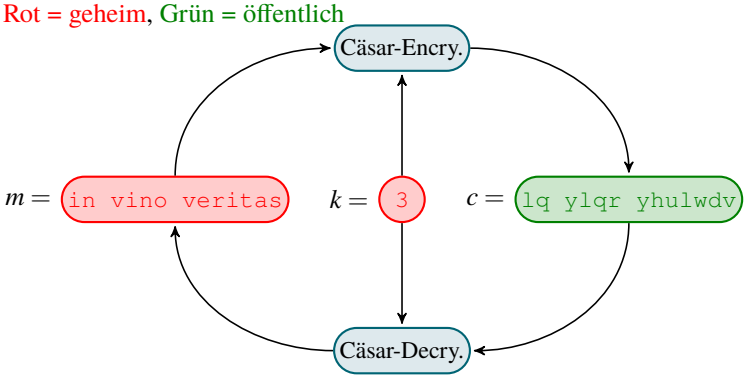
Betätigen Sie sich als *Kryptoanalytiker*! Wie kann man – ohne Kenntnis des Schlüssels k – eine mit einer Cäsar-Chiffre kodierte Nachricht entschlüsseln?

43-8

Beispiel einer Cäsar-Verschlüsselung

Rot = geheim, Grün = öffentlich

Public domain



43-9

Verfahren II: Der One-Time-Pad

Da die Cäsar-Chiffre offenbar nicht sonderlich sicher ist, benötigen wir ein besseres Verfahren:

One-Time-Pad-Algorithmus (Ver- und Entschlüsselung)

Eingaben seien die Nachricht m und der Schlüssel k *gleicher Länge*

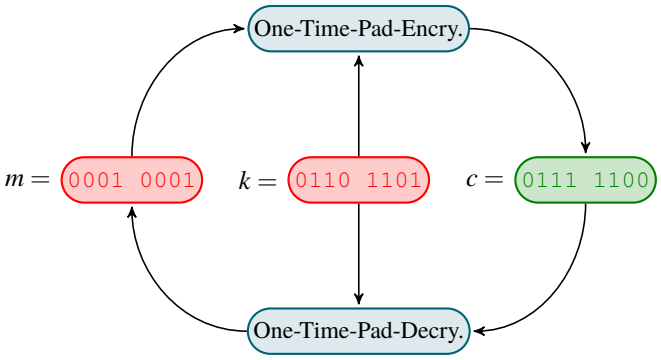
1. Schreibe Nachricht und Schlüssel als Bitstrings auf (wie im ersten Kapitel).
2. Bilde nun das bitweise XOR von Nachricht und Schlüssel. Dies bedeutet: Flippe das i-te Bit der Nachricht, wenn das i-te Bit des Schlüssels eine 1 ist.

Wie der Name schon sagt, kann man das Verfahren leider nur einmal pro Schlüssel (sicher) verwenden. Außerdem sind die Schlüssel schrecklich lang.

Beispiel einer One-Time-Pad-Verschlüsselung

43-10

Rot = geheim, Grün = öffentlich



State-of-the-Art in Sachen symmetrische Verschlüsselung.

43-11

Moderne symmetrische Verschlüsselungverfahren kann man sich als eine *sehr clevere Mischung* aus Cäsar-Verfahren und One-Time-Pad vorstellen. Lange Zeit war das amerikanische DES (data encryption standard) das wichtigste Verfahren. Wegen Problemen wie Ausfuhrverbot, zu kurze Schlüssellänge und Patenten wurde es vor gut 10 Jahren durch ein moderneres und sicheres Verfahren ersetzt. Der neue internationale Standard nennt sich AES (advanced encryption standard).

Sicherheit von Verschlüsselungsverfahren.

43-12

Wann ist ein Verfahren »sicher«?

Informationstheoretisch sicher heißt ein Verfahren, wenn man Chiffretexte ohne Kenntnis des Schlüssels nicht entschlüsseln kann. Leider kann man zeigen, dass nur One-Time-Pad-Verfahren in diesem Sinn sicher sind. *Komplexitätstheoretisch sicher* heißt ein Verfahren, wenn es für die Entschlüsselung kein wesentlich schnelleres Verfahren gibt, als alle Schlüssel durchzuprobieren. Bei Schlüssellängen ab 500 Bits ist solch eine Sicherheitsstufe dann *in diesem Universum nicht zu brechen*.

43.2.3 Asymmetrische Verschlüsselung

Eine ungewöhnliche Idee.

43-13

Symmetrische Verschlüsselungsverfahren haben den *Nachteil*, dass Kommunikationspartner erstmal einen *Schlüssel sicher austauschen müssen*. Rivest, Shamir und Adleman haben 1977 ein Verfahren vorgeschlagen, *bei dem man keinen Schlüssel auszutauschen braucht* – das RSA-Verfahren. (Es gab vorher auch schon ein schlechteres, geheimgehaltenes Verfahren.) Heute ist dies ein Spezialfall der *Klasse der asymmetrischen Verschlüsselungsverfahren*.

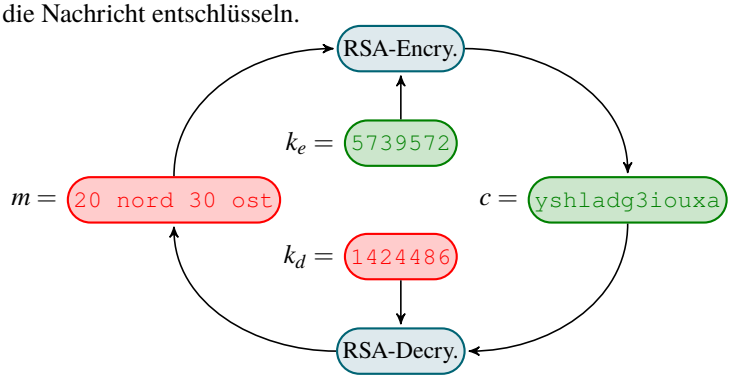
Idee der asymmetrischen Verfahren

Man benutzt *zwei Schlüssel*. Wenn man eine Nachricht mit einem ersten Schlüssel »abschließt«, kann man sie *nur mit dem zweiten Schlüssel »aufschließen«*. Kennt man einen der Schlüssel, so kann man daraus nicht mit vertretbarem Aufwand den anderen Schlüssel generieren.

43-14

Ablauf einer asymmetrischen Verschlüsselung.

Die Royal Airforce möchte den Aufenthaltsort von Prinz Harry an den Buckingham Palace schicken. Dazu braucht sie nur den *öffentlichen Schlüssel 5739572 des Buckingham Palace* zu kennen. Nur im Palast kennt man den *privaten Schlüssel 1424486 des Palastes* und kann die Nachricht entschlüsseln.



43.2.4 Das El-Gamal-Verfahren

Skript-Referenz

Im Folgenden soll das *El-Gamal-Verfahren* vorgestellt werden. Allerdings habe ich, der Darstellung von Bongartz und Unger folgend, zum besseren Verständnis die mathematischen Operationen zunächst durch (zu) radikal vereinfachte ersetzt. Am Ende wird dann noch angedeutet werden, wie es in Wirklichkeit geht.

Grundideen

- Zum *Verschlüsseln* benutzen wir eine *einfache mathematische Operation*. Konkret wird dies *im Wesentlichen eine Multiplikation sein*.
- Zum *Entschlüsseln* muss man diese Operation also rückgängig machen, man muss also im Wesentlichen dividieren, was aber prinzipiell *schwerer* ist als Multiplizieren.
- Genauer nehmen wir an, dass Addition, Subtraktion und Multiplikation *leicht* sind, Division hingegen *sehr schwer*.
- Der *private Schlüssel* hilft uns aber, *statt der Division* die Nachricht doch schnell zu entschlüsseln.

Da das El-Gamal-Verfahren mathematische Operationen benutzt, müssen sinnigerweise sowohl Nachrichten wie Schlüssel auch mathematische Objekte sein, konkret Zahlen:

- Die Nachricht ist eine Zahl; beispielsweise $m = 5$.
- Der private Schlüssel ist eine Zahl; beispielsweise $k_d = 13$.
- Der öffentliche Schlüssel ist ein Paar, bestehend aus einer Zufallszahl z und dem Produkt dieser Zahl mit dem privaten Schlüssel; beispielsweise $k_e = (z, k_d \cdot z) = (11, 143)$.
- *Könnte man dividieren*, so kann man offenbar den privaten Schlüssel k_d leicht aus dem öffentlichen Paar k_e errechnen. Wir nehmen aber in der radikalen Vereinfachung an, dass man nicht effizient dividieren kann.

Eine *Verschlüsselung* funktioniert nun wie folgt:

- Zur Verschlüsselung wählt man *eine zufällige Zahl aus*, beispielsweise $s = 9$.
- Die Verschlüsselung der Nachricht $m = 5$ mit dem Schlüssel $k_e = (z, k_d \cdot z) = (11, 143)$ besteht aus zwei Teilen:

$$t_1 = m + s \cdot k_d \cdot z = 5 + 9 \cdot 143 = 1292$$
$$t_2 = s \cdot z = 9 \cdot 11 = 99$$

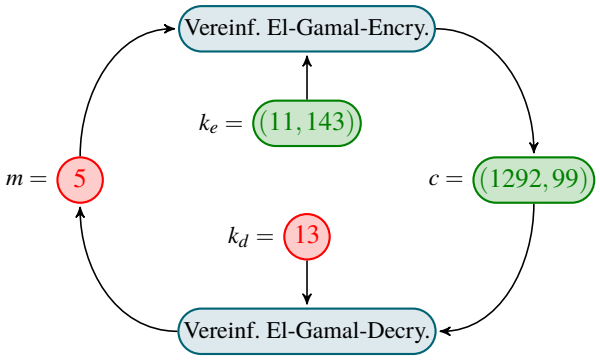
  Also ist die Verschlüsselung von 5 das Chiffrat $(1292, 99)$.

Umgekehrt läuft nun das Entschlüsseln wie folgt ab:

- Es gilt $t_1 = m + t_2 \cdot k_d$ und somit $m = t_1 - t_2 \cdot k_d$.
- Um also aus dem Chiffrat $(1292, 99)$ die Nachricht 5 zu rekonstruieren, kann man beim Entschlüsseln *unter Kenntnis von* k_d wie folgt rechnen:

$$m = t_1 - t_2 \cdot k_d = 1292 - 99 \cdot 13 = 5.$$

- Es wird nur addiert, subtrahiert und multipliziert.



Soweit zu dem vereinfachten El-Gamal-Verfahren. In Wirklichkeit müssen offenbar andere mathematische Operationen angewandt werden, da ja die Annahme, man könne nicht effizient dividieren, recht unrealistisch ist.:

- Im *echten El-Gamal-Verfahren* werden die Grundoperationen wie folgt ersetzt (p sei eine große Primzahl – es gibt Verfahren, solche Primzahlen schnell zu bestimmen):
  1. Addition wird ersetzt durch »Modulo-Multiplikation«: $a + b$ wird zu $(a \cdot b) \bmod p$.
  2. Subtraktion wird ersetzt durch »Modulo-Division«: $a - b$ wird zu $(ab^{p-2}) \bmod p$.
  3. Multiplikation wird ersetzt durch »Modulo-Exponentenbildung«: $a \cdot b$ wird zu $a^b \bmod p$.
  4. Division wird ersetzt durch den »Modulo-Logarithmus«.
- Man überlegt sich nun recht leicht, dass zunächst die Modulo-Multiplikation recht flott berechnet werden kann, selbst wenn die Zahlen a, b und p alle einige Hundert Stellen haben. Dazu benutzt man einfach das schriftliche Multiplizieren und Dividieren, wie man es aus der Schule kennt.
- Für die Modulo-Division und -Exponentenbildung ist zunächst nicht klar, wie diese schnell funktionieren sollen: Es ist in der Tat nicht möglich, in vertretbarer Zeit a^b zu berechnen, wenn a und b jeweils hunderte Ziffern haben. Der Grund ist einfach, dass in diesem Fall das Ergebnis länger ist, als es Atome im Universum gibt.
  Nun soll aber auch nicht a^b berechnet werden, sondern $a^b \bmod p$. Der Trick ist grob folgender: Will man beispielsweise $a^{256} \bmod p$ berechnen, so kann man dies schnell machen, indem man nacheinander a bin p, a^2 bin p, a^4 bin p, a^8 bin p, a^{16} bin p, a^{32} bin p, a^{64} bin p, a^{128} bin p und a^{256} bin p berechnet. Jede Zahl in der Reihe ergibt sich, indem man die vorherige Zahl quadriert und modulo p rechnet. Will man also a^b bin p berechnen für ein b mit tausend Stellen, so muss man diese Operation etwa eintausend Mal ausführen – was noch eine vertretbare Laufzeit ergibt.

Entscheidend für die Sicherheit des El-Gamal-Verfahrens ist nun, dass man für die Berechnung des Modulo-Logarithmus schlicht kein schnelles Verfahren kennt.

43.3 Sichere E-Mail

43.3.1 Vertraulichkeit: Digitale Briefumschläge

Eine Nachricht soll vertraulich sein.

43-15

Ziel: Vertraulichkeit

- Eine Nachricht soll einer Person zugestellt werden.
- Nur diese Person soll die Nachricht lesen können.

Dies nennt man auch einen *digitalen Briefumschlag*.

Methode

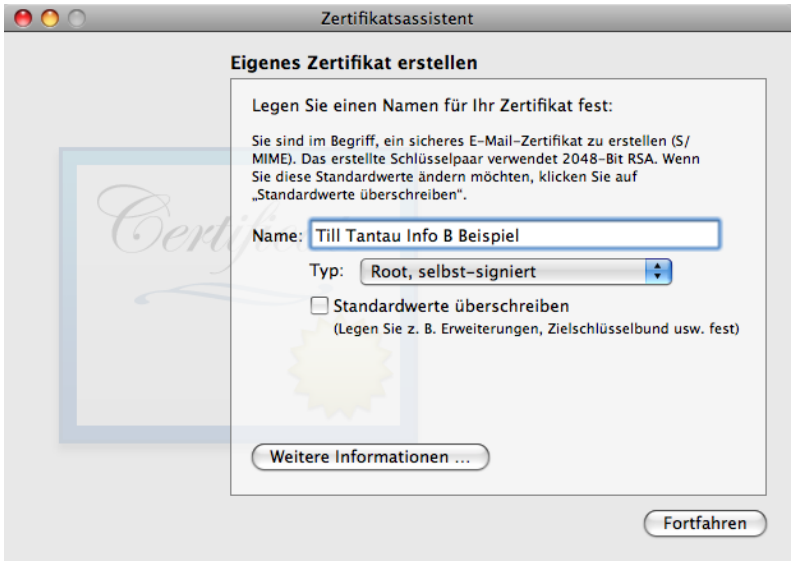
Die Person erzeugt ein RSA-Paar (k_e, k_d). Der Schlüssel k_e wird »allgemein bekanntgegeben« und heißt nun *öffentlicher Schlüssel*. Nachrichten werden mit dem öffentlichen Schlüssel der Person verschlüsselt. Effekt: Nur diese Person kann die Nachricht (mit dem nur ihr bekannten privaten Schlüssel k_d) entschlüsseln.

43-16

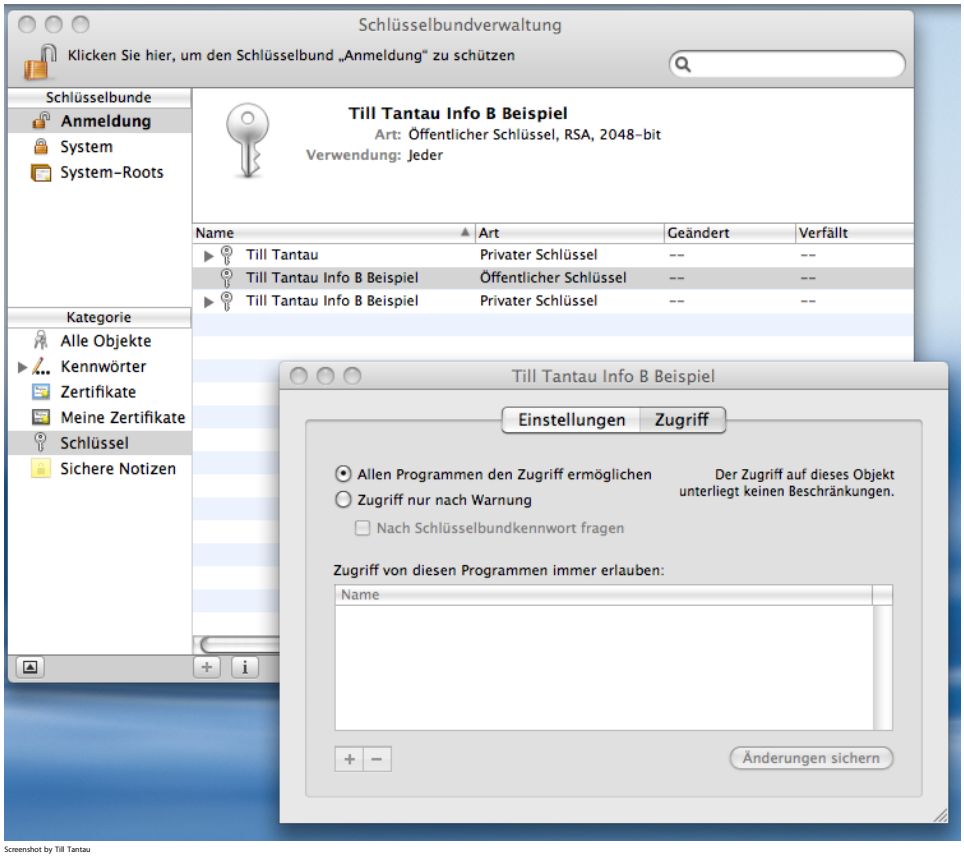
Praktische Umsetzung: Erzeugen des Schlüsselpaares.

Zur Erzeugung eines Schlüsselpaares benutzt man ein *geeignetes Programm* wie beispielsweise `openssl`. Bei MacOS kann man auch komfortabel den *Schlüsselbundverwaltung* nutzen.

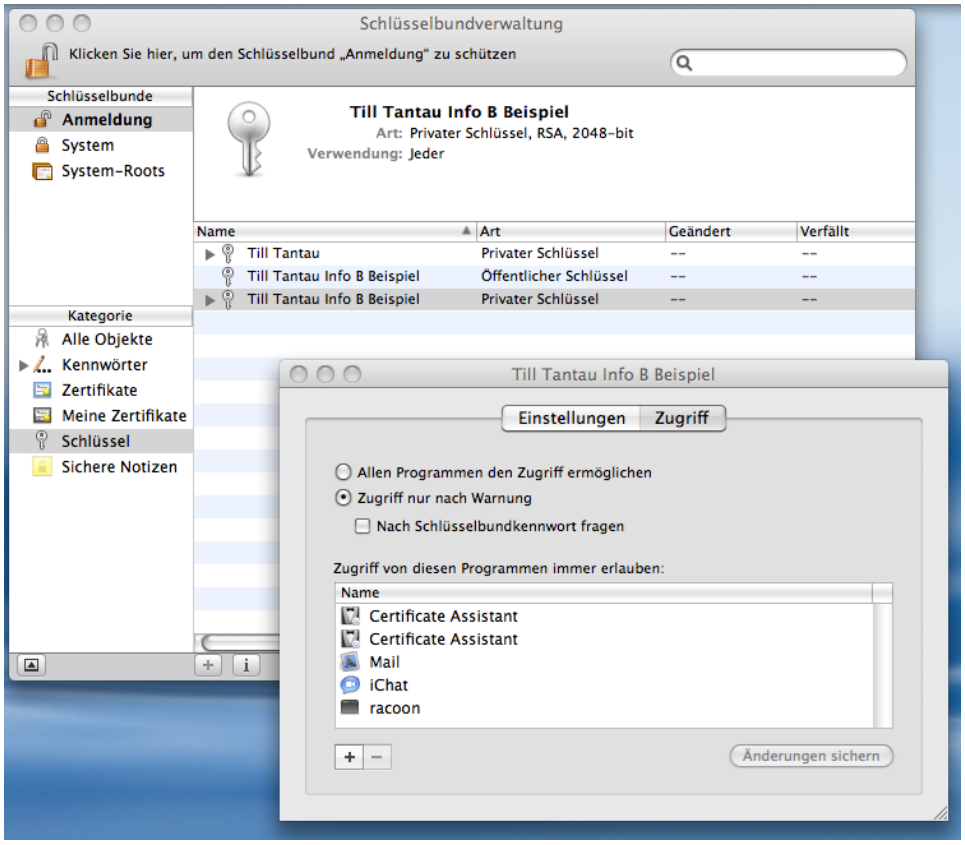
(Das Verfahren wird gleich noch etwas komplizierter werden, aber kümmern wir uns erstmal um den einfachen Fall.)



Screenshot by Till Tantau



Screenshot by Till Tantau



Screenshot by Till Tantau

Praktische Umsetzung: E-Mail an Johannes.

43-17

Schritt 1: Wir brauchen Johannes öffentlichen Schlüssel.

Um eine Mail zu verschlüsseln, muss das E-Mail-Programm den *öffentlichen Schlüssel der Person kennen, der man eine E-Mail schreiben möchte*. Diesen öffentlichen Schlüssel kann einem die Person beispielsweise vorher geschickt haben. E-Mail-Programme oder die Schlüsselverwaltung erlaubt es, solche öffentliche Schlüssel zu *importieren* (ein geeigneter Menüpunkt). So sieht der öffentliche Schlüssel von Johannes beispielsweise wie folgt aus:

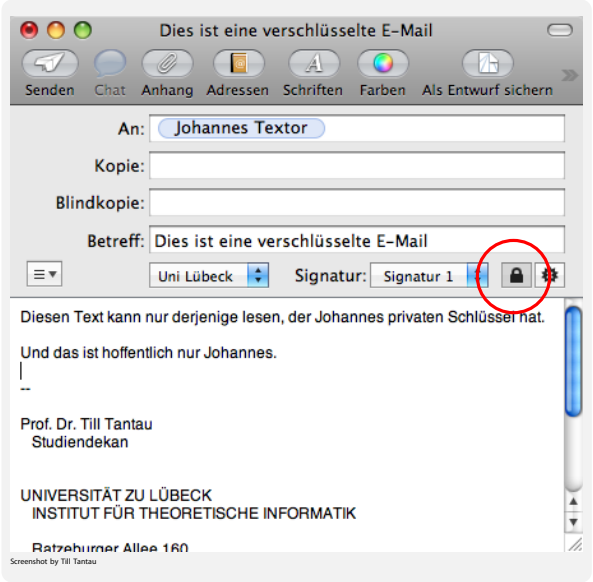
```
-----BEGIN CERTIFICATE-----
MIIFSTCCBDGgAwIBAgIECz1arTANBgkqhkiG9w0BAQUFADCBqzELMAkGA1UEBhMC
REUxIDAeBgNVBAoTF1VuaXZlcnNpdGFldCB6dSBMdWViZWNrMS4wLAYDVQQLEyVJ
bnN0aXR1dCBmdWVyIE1lZGl6aW5pc2NoZSBJbmZvcm1hdGlrMScwJQYDVQQDEx5D
QSBkZXIgVW5pdmVyc2l0YWV0IHp1IEx1ZWJlY2sxITAfBgkqhkiG9w0BCQEWEnBr
aUB1bmktbHVlYmVjay5kZTAeFw0wNzEwMjMxMTM4MDdaFw0xMDEwMjIxMTM4MDda
... 20 ausgelassene Zeilen ...
Q8RGGPHGKcAocX3kGB3VTWZptDCACiJ9E5Q5pD4mWMRPYAgYnjJwWv4KzF5Moboe
29IKgSvifR5ttcdqCFn5gfwVYrHWOLxxSZkbUpqGIsAhoGeWd65Z9u4FArI87UIw
3KCso+ohahGUeVqZQk6ZXU6zJX/hw6K4y2l1QfiBwVQuNjvudADM3Q6RSedECK4m
MsUhRqoUqvXegCZ7mA==
-----END CERTIFICATE-----
```

Er kann auf `https://pki.pca.dfn.de/uzl-ca/cgi-bin/pub/pki` heruntergeladen werden. Alternativ kann Johannes den Schlüssel zunächst von seinem Mail-Programm oder von der Schlüsselverwaltung *exportieren* und dann verschicken (zur Not per Post).

43-18

Praktische Umsetzung: E-Mail an Johannes.
Schritt 2: Schreiben und abschicken der E-Mail.

Hat man dem eigenen System erstmal Johannes öffentlichen Schlüssel beigebracht, so kann man ihm eine verschlüsselte E-Mail schreiben. Bei modernen E-Mail-Programmen muss man dazu einfach auf einen Verschlüsselungsknopf drücken.



An Johannes wird dann ein mit dem *öffentlichen Schlüssel von Johannes verschlüsselter* Text geschickt. Um den Text zu entschlüsseln, braucht man den privaten Schlüssel von Johannes. Folglich kann niemand außer Johannes – *nichtmal der Autor der Mail* – den Text wieder entschlüsseln.

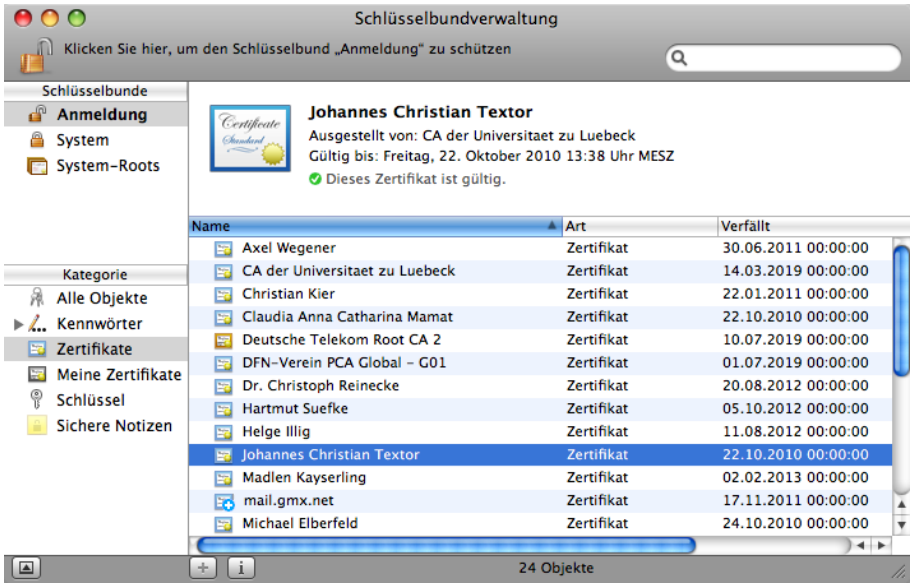
```
Message-Id: <AF944753-2276-487B-9BB1-995DA5D2058F@tcs.uni-luebeck.de>
From: Till Tantau <tantau@tcs.uni-luebeck.de>
To: Johannes Textor <textor@tcs.uni-luebeck.de>
Content-Type: application/pkcs7-mime;
        name=smime.p7m;
        smime-type=enveloped-data
Content-Transfer-Encoding: base64
X-Smtp-Server: theogate.tcs.uni-luebeck.de:tantau
Content-Disposition: attachment;
        filename=smime.p7m
Mime-Version: 1.0 (Apple Message framework v936)
Subject: =?ISO-8859-1?Q?Dies_ist_eine_verschl=FCsselte_E-Mail?=
Date: Fri, 11 Jun 2010 16:05:31 +0200

MIAGCSqGSIb3DQEHA6CAMIACAQAxggOiMIIBzQIBADCBtDCBqzELMAkGA1UEBhMCREUxIDAeBgNV
BAoTF1VuaXZlcnNpdGFldCB6dSBMdWViZWNrMS4wLAYDVQQLEyVJbnN0aXR1dCBmdWVyIE1lZGl6
aW5pc2NoZSBJbmZvcm1hdGlrMScwJQYDVQQDEx5DQSBkZXIgVW5pdmVyc2l0YWV0IHplIEx1ZWJl
Y2sxITAfBgkqhkiG9w0BCQEWEnBraUB1bmktbHVlYmVjay5kZQIECz1arTANBgkqhkiG9w0BAQEF
AASCAQDbXJGvqKCu+Qd1xfTLSJKW6I8T2Eds7ks1qtvBVoD6RFzcHF7XOTllzcOmxgC2GS8frzNr
pVIC5VBssy/BEnm3BCpD8sw8HCxE0pcIm96/p5oHp9Qyk0FYs97JF3GLHVXPAs8AoHEMoVBXAuku
... 66 ausgelassene Zeilen ...
FbiGph3to8Q/xaLEGHplQNN245YGcAY3xPAKibQfs3725ctGA9bBUXliZnejMXR15DMWcd8VozAA
3MAQYwRIfJprLx7pYpUMEbSUN1Zdqi4WhTYS+6MvVEbj5ItZ68PWDx2OFsqm5enql58XViNLuS58
aTzsJ6uw86lgEgqbbiD8wpMExZxpBCBxX07+jmfobYe16/gFjInEB7XOV+VC95GEryi87pu3GAQI
qJB1HZWcdHQAAAAAAAAAAAA
```

43-19

Zusammenfassung: Wer bekommt welchen Schlüssel?

- Generell gilt: Ihr *privater* Schlüssel ist nur Ihnen bekannt (auch nicht Ihrem Lover!) und möglichst gut auf Ihrem Computer geschützt.
  - Am sichersten ist es, wenn der Schlüssel auf einer separaten Karte liegt, die ihrerseits durch eine PIN geschützt ist – dies ist mit dem neuen Personalausweis möglich.
  - In der Praxis liegt Ihr privater Schlüssel aber auf Ihrer Festplatte.
  - Dort ist er selbst wieder durch ein Master-Passwort symmetrisch verschlüsselt und damit selbst vor Diebstahl des Computers oder vor bösartigen Hackern geschützt.
- Generell gilt: *Öffentliche* Schlüssel sind möglichst breit gestreut und überall verfügbar. Sie sollten auf Ihrem System die öffentlichen Schlüssel aller Ihnen bekannter Personen haben.



Screenshot by Till Tantau

43.3.2 Authentizität: Digitale Unterschriften

Eine Nachricht soll digital unterschrieben werden. 43-20

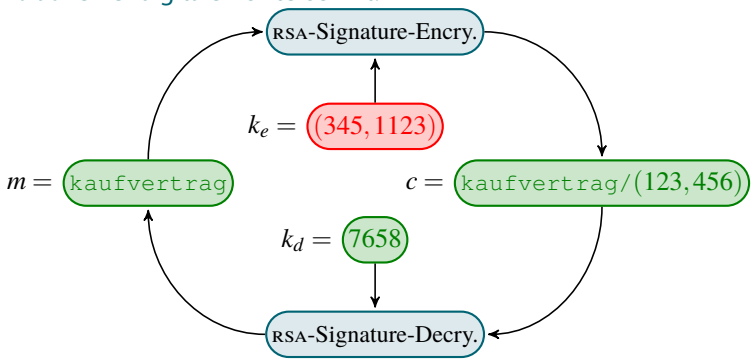
Ziel: Authentizität

Es soll garantiert werden, dass eine Nachricht von einer bestimmten Person stammt. Dies nennt man auch eine *digitale Unterschrift*.

Methode

Man benutzt dieselben Schlüssel wie beim Verschlüsseln von Nachrichten, *nur umgekehrt*: Die zu unterschreibende Nachricht wird mit dem *privaten Schlüssel* k_e verschlüsselt und dieser Text an die Originalnachricht angehängt. Überprüfung: Man entschlüsselt den verschlüsselten Teil mit dem öffentlichen Schlüssel k_d und vergleicht ihn mit dem behaupteten Text. Effekt: Nur die Person, die k_e kennt, kann die unterschriebene Nachricht erzeugen.

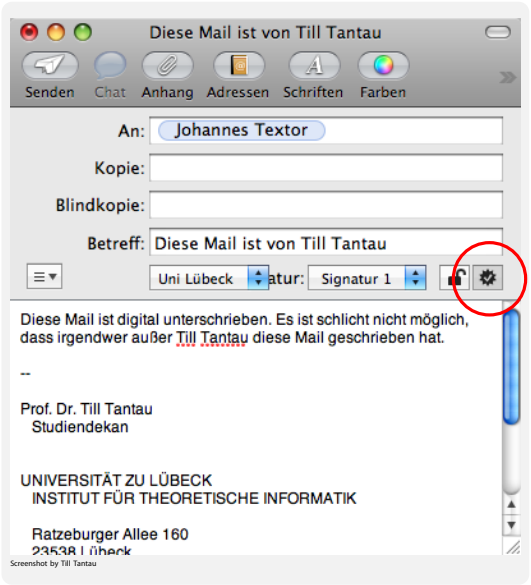
Ablauf einer digitalen Unterschrift. 43-21



43-22

Praktische Umsetzung: Unterschriebene E-Mail an Johannes.

Um eine E-Mail zu unterschreiben, braucht man lediglich den eigenen privaten Schlüssel – und den haben wir ja schon erzeugt. Damit jemand die E-Mail überprüfen kann, braucht er den öffentlichen Schlüssel – diese sind ja aber frei zugänglich.



Achtung: Mit »Signatur« wird auch manchmal der Standard-Text am Ende einer Mail bezeichnet. Dieser ist *keine* digitale Unterschrift.

```
Message-Id: <A15605B5-9389-442D-BD9E-FC65D76A18DD@tcs.uni-luebeck.de>
From: Till Tantau <tantau@tcs.uni-luebeck.de>
To: Johannes Textor <textor@tcs.uni-luebeck.de>
Content-Type: multipart/signed;
        boundary=Apple-Mail-11--309560619;
        micalg=sha1;
        protocol="application/pkcs7-signature"
X-Smtp-Server: theogate.tcs.uni-luebeck.de:tantau
Mime-Version: 1.0 (Apple Message framework v936)
Subject: Diese Mail ist von Till Tantau
Date: Mon, 14 Jun 2010 08:44:26 +0200

... 13 ausgelassene Zeilen ...
Diese Mail ist digital unterschrieben. Es ist schlicht nicht m=F6glich, =20=

dass irgendwer au=DFer Till Tantau diese Mail geschrieben hat.
... 62 ausgelassene Zeilen ...

--Apple-Mail-11--309560619
Content-Disposition: attachment;
        filename=smime.p7s
Content-Type: application/pkcs7-signature;
        name=smime.p7s
Content-Transfer-Encoding: base64

MIAGCSqGSIb3DQEHAqCAMIACAQExCzAJBgUrDgMCGgUAMIAGCSqGSIb3DQEHAQAAoIIOnzCCBCEw
ggMJoAMCAQICAgDHMA0GCSqGSIb3DQEBBQUAMHExCzAJBgNVBAYTAkRFMRwwGgYDVQQKExNEZXV0
c2NoZSBUZWxla29tIEFHMR8wHQYDVQQLExZULVRlbGVTZWMgVHJ1c3QgQ2VudGVyMSMwIQYDVQQD
... 79 ausgelassene Zeilen ...
DfpFZundteQqc6R/FdbTj67j23Y5h/8+qCIewT//LLWh4hvW/kEs7bi1Ibcm3yniFd4PzjkKI9gi
WuPJkqz3VrcuGYfjCHT3RGyAANNQOF6fiJQ5tipmN4dHkfoxWQ7nPJBobS13MLLd+fIvBrI78pTp
8mFDaOsObxdlyOXhm5RTBor2U/4uDGtZJddGOCNLpNjnCqEti3PWAHD/IAAAAAAAAA==

--Apple-Mail-11--309560619--
```

43.3.3 Echtheits-Zertifikate: Digitale Notare

43-23

Das Henne-Ei-Problem bei öffentlichen Schlüsseln

Damit ich Johannes eine sichere Mail schreiben kann, muss er mir seinen öffentlichen Schlüssel zukommen lassen. Nun könnte auch irgendjemand anderes mir einen Schlüssel schicken und behaupten, er sei Johannes und dies sei sein Schlüssel. Um das auszuschließen, würde ich gerne verlangen, dass die Mail von Johannes unterschrieben ist – aber dazu brauche ich ja gerade den öffentlichen Schlüssel, um den es gerade geht.

Certificate-Authorities zertifizieren die Echtheit von Schlüsseln.

43-24

Ziel: Echtheit von Schlüsseln bezeugen

A will sichergehen, dass ein vermeintlicher öffentliche Schlüssel von B tatsächlich von B stammt.

Methode

Eine *vertrauenswürdige Instanz*, »Certificate Authority (CA)« oder »Trust-Center« genannt, legt *Root-Schlüssel* (k_d, k_e) für digitale Unterschriften an. Der öffentliche Schlüssel k_d ist allgemein bekannt (er ist zum Beispiel in Ihren Browser schon fest eingebaut). Benutzer B lässt sich von der vertrauenswürdigen Instanz den folgenden Text unterschreiben: »Person B hat den öffentlichen Schlüssel 1234567.« Wenn A mit B zum ersten Mal redet, schickt B diesen unterschriebenen Text. A kann die Unterschrift der CA überprüfen (A kennt ja das Root-Zertifikat) und kann dann den öffentlichen Schlüssel von B benutzen.

Die privaten Schlüssel der Root-CAs sind *extrem gut gesichert*. Das darf man sich in etwa wie bei Mission Impossible vorstellen – inklusive gepanzerter Räume mit Servern, die vom Netz und auch von sonst allem getrennt sind. Selbst wenn Sie mit Ihrer Privatarmee das Gebäude stürmen, die wackeren Systemadministratoren überwältigen bevor diese die Schlüssel löschen können und den Schlüssel stehlen, würde das nicht viel nützen. Schlüssel können in zentralen Verzeichnissen als ungültig erklärt werden, was einige Minuten nach Ihrem Angriff der Fall wäre. Merke: Wenn man den privaten Schlüssel einer Root-CA klaut, so darf dies niemand merken. Ich empfehle, sich beispielsweise an Herrn Hunt zu wenden.

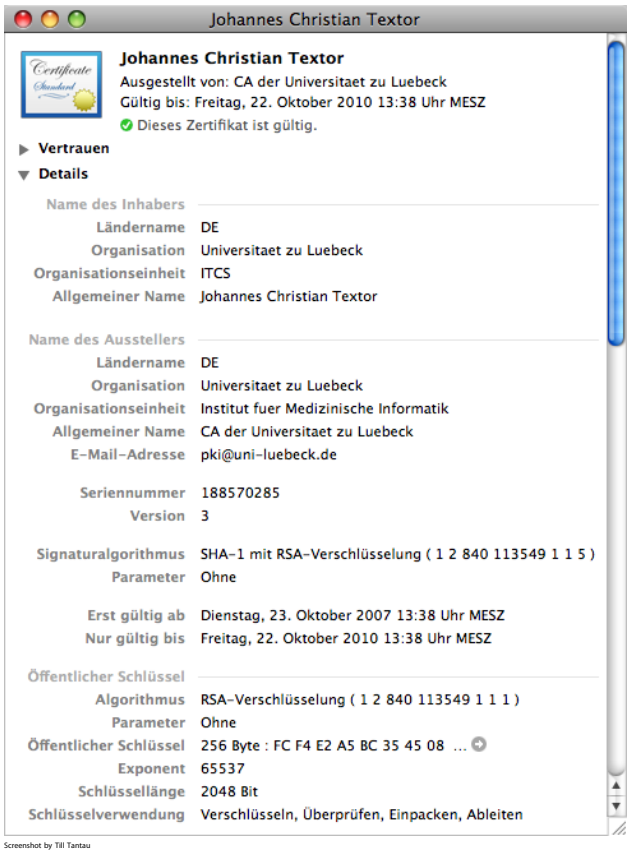
Wenn die Zeit es erlaubt, so würde hier die Schlüsselszene aus Mission Impossible gut passen, bei der Herr Cruise an Seilen schwebend Daten auf eine recht altertümliche Diskette kopiert und dabei Problem mit Schweißbildung hat.

Regie

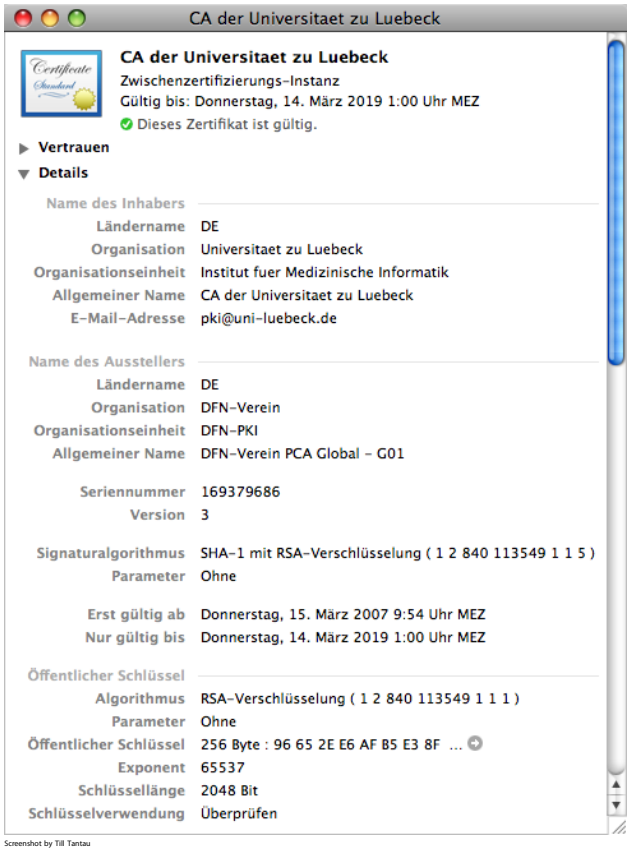
Kaskaden von Unterschriften

43-25

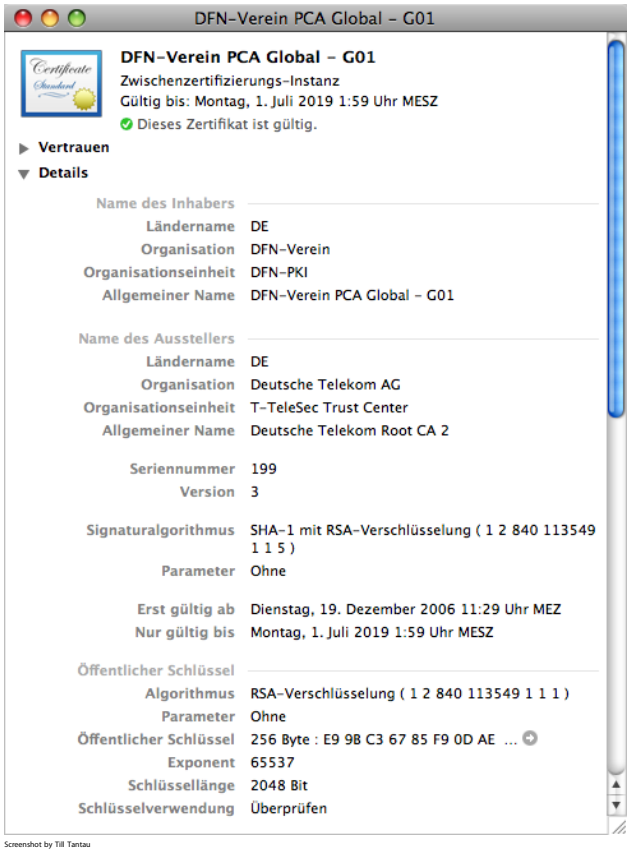
In der Praxis kann die Root-CA nicht die Schlüssel beispielsweise aller Deutschen unterschreiben. Stattdessen gibt es »Zwischen-CAs«.



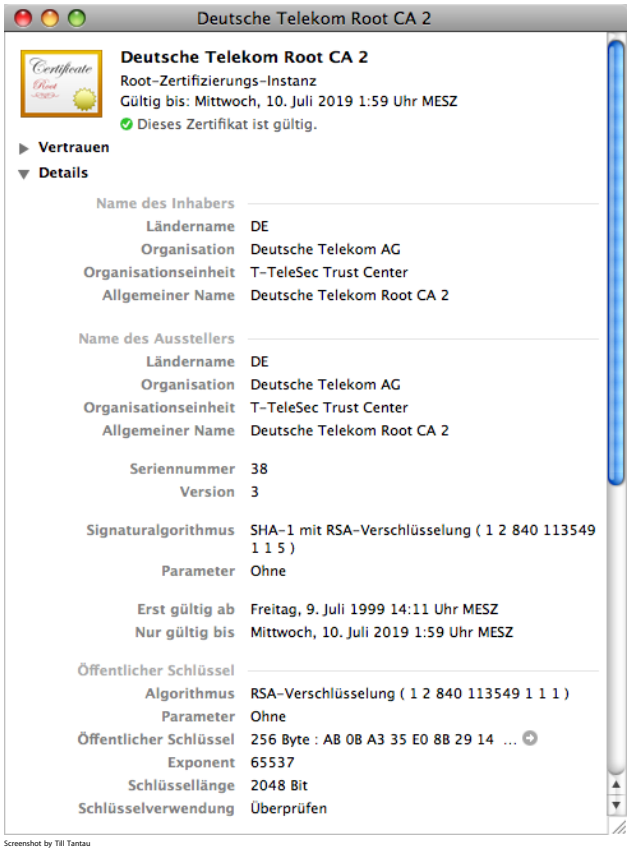
Screenshot by Till Tantau



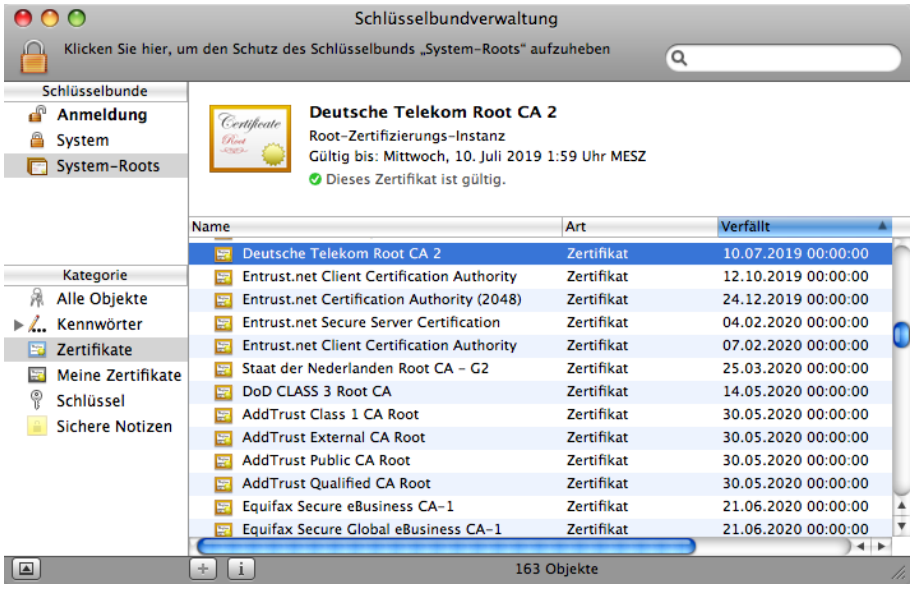
Screenshot by Till Tantau



Screenshot by Till Tantau



Screenshot by Till Tantau



Screenshot by Till Tantau

43.4 Sicheres Surfen

Sicheres Surfen funktioniert wie sichere E-Mail.

43-26

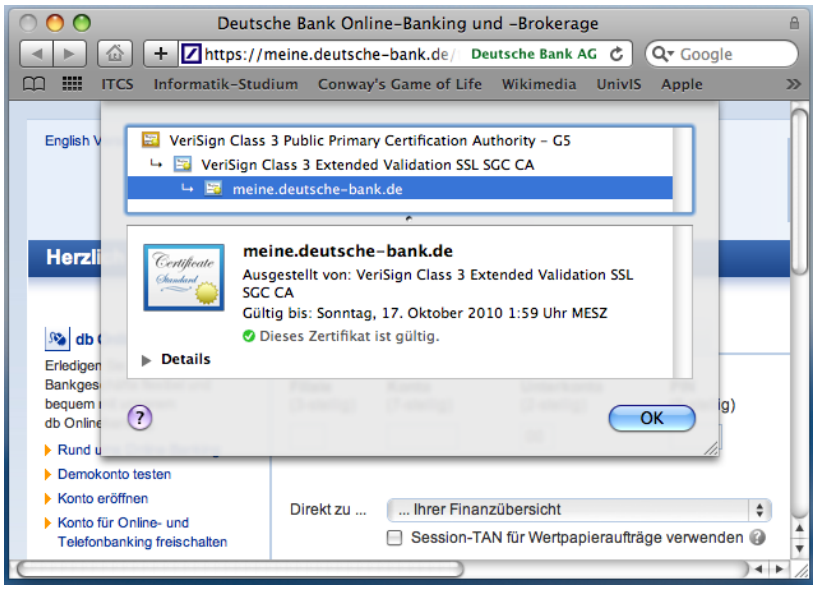
Wenn Sie mit einem Online-Shop oder Ihrer Bank kommunizieren, haben Sie *dieselben Problem wie bei E-Mail*:

- Niemand soll die Kommunikation abhören können.
- Sie müssen sicher sein können, dass Ihre Bank auch wirklich Ihre Bank ist.

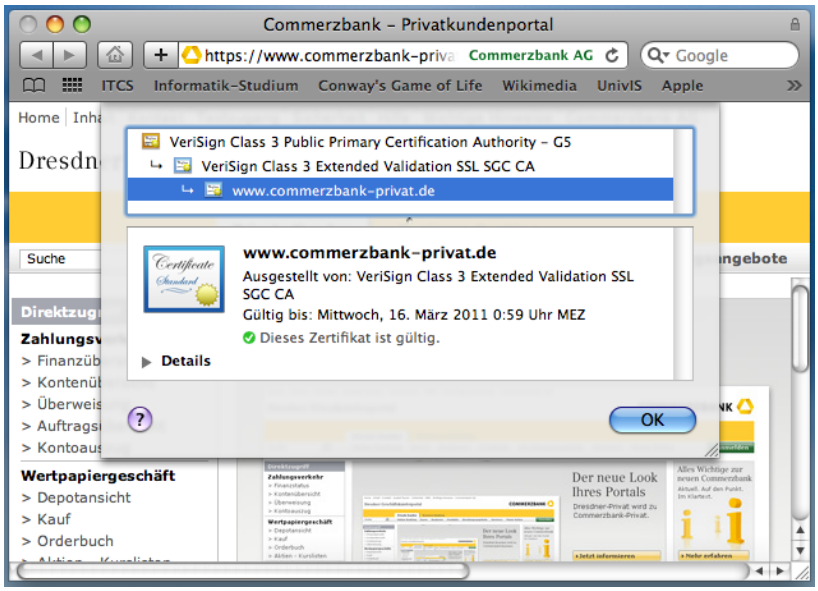
Diese Probleme werden auch genauso gelöst:

- Die Bank oder der Online-Shop hat ein Schlüsselpaar, dass nun aber *keine Person* identifiziert sondern *eine Webseite*.
- Das Schlüsselpaar ist über eine Kette von CAs unterschrieben.

Die Protokolle *https* und *ssh* bauen auf diese Art sichere Kanäle auf.



Screenshot by Till Tantau



Screenshot by Till Tantau

Zusammenfassung dieses Kapitels

43-27

Symmetrische Verschlüsselung
Eine Nachricht m wird mit einem Schlüssel k zu einem Chiffrat $c = e(m,k)$ verarbeitet. Mit *demselben* Schlüssel k lässt sich dann $m = d(c,k)$ berechnen. Das Standardverfahren heißt AES.

Asymmetrische Verschlüsselung
Eine Nachricht m wird mit einem *öffentlichen Schlüssel* k_e zu einem Chiffrat $c = e(m,k_e)$ verarbeitet. Mit einem *ganz anderen* Schlüssel k_d lässt sich dann $m = d(c,k_d)$ zurückgewinnen. Das Standardverfahren heißt RSA.

Vertraulichkeit
Die *Vertraulichkeit* von E-Mails und der Kommunikation mit Webservern (https) wird sichergestellt, indem mit dem *öffentlichen Schlüssel des Empfängers* verschlüsselt wird.

Authentizität
Die *Authentizität* einer Nachricht wird sichergestellt, indem mit dem *privaten Schlüssel des Absenders* verschlüsselt wird. Dies nennt man *digitale Unterschrift*.

▶ Zertifikate
Ein *digitales Zertifikat* ist ein von einer *Certificate Authority* unterschriebener Text, der bezeugt, dass ein öffentlicher Schlüssel zu einer bestimmten Person gehört.

Übungen zu diesem Kapitel

Übung 43.1 Sicherheit einer Briefzustellung bewerten, einfach

Wenn Ihre EC-Karte abläuft oder Sie den PIN vergessen haben, bekommen Sie eine neue PIN per normaler Briefpost zugestellt. Der Zettel mit der PIN wird in einem durchleuchtungssicheren Umschlag verschickt und ist darin nochmals in einem durchleuchtungssicheren Briefumschlag verschlossen. Um die Sicherheit zusätzlich zu erhöhen, wird auf dem Umschlag kein Absender angegeben, so dass man nicht sehen kann, dass der Brief von Ihrer Bank kommt.

Werden durch diese Vorsichtsmaßnahmen die Sicherheitsziele »Geheimhaltung«, »Integrität« und »Authentizität« erreicht?

Übung 43.2 Asymmetrische Verschlüsselung anwenden, einfach

Alice und Bob sind Geheimagenten, die regelmäßig über das Internet streng geheime Informationen austauschen. Dabei baut Alice eine Verbindung zu einem Rechner auf, der Bob gehört. Um sich zu identifizieren, übermittelt Alice am Anfang des Dialogs ein geheimes, zuvor vereinbartes Codewort.

Eva hat nun vom Treiben der Beiden Wind bekommen und einen finsteren Plan geschmiedet: Sie möchte Bob hinters Licht führen und sich selbst als Alice ausgeben. Dazu bemächtigt sie sich eines Routers im Internet, über den die Kommunikation zwischen Alice und Bob abläuft. Damit ist sie in der Lage, das geheime Codewort von Alice auszuschnüffeln und sich somit gegenüber Bob als Alice ausgeben zu können.

Da Alice und Bob bereits Verdacht geschöpft haben, ändern sie ihre Vorgehensweise wie folgt ab: Bob erzeugt ein Zertifikat und leitet den öffentlichen Schlüssel über einen sicheren Kanal an Alice weiter. Alice verschlüsselt dann ihr Passwort mit diesem öffentlichen Schlüssel, damit es sicher über das Netz übertragen wird. Beurteilen Sie, wie stark die Sicherheit der Kommunikation zwischen Alice und Bob durch diese Verschlüsselung erhöht wird!

Übung 43.3 Authentifizierungsprotokoll entwerfen, mittel

Angenommen, Alice und Bob aus Übung 43.2 sind beide im Besitz des öffentlichen Schlüssels des jeweils Anderen, und Eva kann nicht an die dazugehörigen privaten Schlüssel gelangen. Was könnten Alice und Bob dann tun, um Eva daran zu hindern, sich für Alice oder Bob auszugeben?

Übung 43.4 Replay-Attacken verhindern, mittel

Bei einer *Replay-Attacke* wird eine verschlüsselte Botschaft abgefangen und aufgezeichnet, um durch das erneute Abspielen dieser Botschaft zum Beispiel eine fremde Identität vorzutäuschen. Ein einfaches Beispiel für ein System, das potenziell für Replay-Attacken anfällig ist, sind moderne Autoschlüssel, bei denen man über Knopfdruck bereits aus einigen Metern Entfernung das Auto öffnen kann. Beim Entwurf eines solchen Systems muss darauf geachtet werden, dass ein Dieb nicht einfach das vom Schlüssel zum Auto übertragene Funksignal aufzeichnen und sich damit später Zugriff zum Auto verschaffen kann, ohne den Schlüssel zu besitzen.

Beschreiben Sie ein Kommunikationsprotokoll zwischen Schlüssel und Auto, das immun gegen Replay-Attacken ist! Gehen Sie dabei davon aus, dass sowohl der Schlüssel als auch das Auto kleine Computer enthalten, die über Funk in beide Richtungen miteinander kommunizieren können.

Übung 43.5 Sniffing, mittel

Für diese Übung sind im Arbeitsraum »Abhörstationen« aufgebaut, auf denen das Programm *wireshark* läuft. Mit diesem Programm kann man die Pakete, die im lokalen Netzwerk unterwegs sind, abfangen und damit sensible Informationen erschnüffeln. Sie können auch eigene Laptops mitbringen und abhören lassen, achten Sie aber darauf, keine wirklich sensiblen Daten preiszugeben.

Diskutieren Sie zu Beginn, wie realistisch der Versuchsaufbau Ihnen erscheint.

Teilen Sie sich dann in Gruppen auf. Während eine Gruppe am abgehörten Rechner im Internet surft (im Folgenden auch »Opfer« genannt), ist es die Aufgabe der Gruppe am Abhörrechner, so viel wie möglich über die Aktivitäten der anderen Gruppe herauszufinden. Sie können zum Beispiel versuchen, folgendes herauszufinden:

- Welche Programme auf dem Computer des Opfers kommunizieren mit dem Internet?
- Welches Betriebssystem und welchen Webbrowser verwendet das Opfer?
- Welche Webseiten wurden vom Opfer besucht?

- Wie lautet das GMX-Passwort des Opfers?
- Wie lautet die ICQ-Nummer des Opfers?
- Mit wem chattet das Opfer worüber?

Insbesondere beim GMX-Szenario sollten Sie nicht Ihre echten Passwörter verwenden.

Welche Protokolle spielen hier eine Rolle und auf welchen Schichten des Schichtenmodells spielen sich diese ab? Wie kann man verhindern, dass das eigene GMX-Passwort ausgeschnüffelt wird?

44-1

Kapitel 44
System-Sicherheit
Von Viren, Würmern und SQL-Spritzen

Lernziele dieses Kapitels

1. Bedrohungsszenarien der Sicherheit von IT-Systemen kennen und einschätzen können
2. Schutzmaßnahmen für die Sicherheit von IT-Systemen kennen und ergreifen können
3. Beispiel eines Sicherheitslochs verstehen

44-2

Inhalte dieses Kapitels

44.1	Systemsicherheit	367
44.1.1	Was ist zu schützen?	367
44.1.2	Wovor ist zu schützen?	368
44.1.3	Welche Maßnahmen helfen?	369
44.2	Fallbeispiel eines Sicherheitslochs	369
44.2.1	Die Methode: SQL-Injection	369
44.2.2	Fiktives Beispiel: Firmen-Intranet	370
44.2.3	Reales Beispiel: www.doc.state.ok.us	371
	Übungen zu diesem Kapitel	372

Worum es heute geht

In dem Film *War Games* aus dem Jahr 1983 löst der sympathische junge Held beinahe den Dritten Weltkrieg aus, indem er per Modem zufällig Telefonnummern anruft und dabei auf die des Zentralcomputers des Pentagons trifft. Der Computer gibt sich als Spieleserver aus, die wirklich spannenden Spiele wie *Global Thermonuclear War* sind aber durch ein Passwort geschützt. Mit ein paar Nachforschungen kommt der Held an das Passwort (der Name des Sohnes des Programmierers) und kann dann mit dem Computer eine Partie wagen. Leider setzt der Computer das Spiel gleich in die Realität um und nur der Held kann die Maschine in eine Endlosschleife schicken, woraufhin sie im wahrsten Sinne des Wortes durchbrennt. (Wenn die Uni-Computer jedesmal anfangen würden zu qualmen, wenn Sie mal wieder eine Endlosschleife programmiert haben, wäre die Universität schon längst abgefackelt.)

Wie sieht die Bedrohungslage 30 Jahre später aus? Sie werden sich nicht mehr mit einem Modem in das Pentagon einwählen können; darf man zumindest hoffen. Jedoch gibt es das moderne Äquivalent zu solchen »Hintertürchen« zu Systemen immer noch. Ein besonders krasses Beispiel aus dem Jahr 2008 werden Sie in diesem Kapitel kennen lernen: Die Webseite der Justizvollzugsanstalten des US-Staates Oklahoma. Über deren Webseiten hätte man zwar nicht den Dritten Weltkrieg auslösen, aber zumindest die Justizverwaltung gehörig durcheinander bringen können.

Bei Sicherheit in der Informationstechnologie (IT-Sicherheit) geht es aber nicht nur darum, Systeme möglichst gut abzuschotten. Seit Ende der neunziger Jahre hat ein Prozess begonnen, in dessen Rahmen Menschen immer mehr Persönliches digitalisieren und häufig anderen Menschen zugänglich machen. Dies gilt für digitale soziale Räume wie Facebook ebenso wie für die personalisierte Google-Seite, durch die die Firma Google die komplette Historie Ihrer Suchanfragen protokollieren und auswerten kann. Bei IT-Sicherheit geht es auch um die Frage, wie hier der Schutz der Daten vor Verlust und unbefugtem Zugriff zu gewährleisten ist. Dieses Problem hat viele Aspekte, unter anderem rechtliche, technische, algorithmische und auch soziale.

44-4

Wiederholung: Worum geht es bei IT-Sicherheit?

Bei *IT-Sicherheit* geht es um folgende Anliegen:

1. Schutz vor und die Aufrechterhaltung des Betriebs bei
   - Ausfall von Teilen des Systems (Stromausfall, Absturz)
   - Angriffen auf das System (durch Hacker, korrumpierte Mitarbeiter)

   Diese *Systemsicherheit* wird uns in diesem Kapitel interessieren.
2. Schutz von Daten und Kommunikation vor
   - Spionage
   - Fälschung

   Diese *Daten- und Kommunikationssicherheit* war Thema des letzten Kapitels.

44-5

Wie erreicht man IT-Sicherheit?

Es gibt verschiedene Maßnahmen, um die IT-Sicherheit zu erhöhen. *Perfekte Sicherheit kann es nicht geben.*
Mögliche Maßnahmen sind:

Redundanz Daten liegen mehrfach vor.
Stichwörter: Backups.

Abschottung Es wird schwierig gemacht, in das System hineinzukommen.
Stichwörter: Passwörter und Firewalls.

Aktive Kontrolle Es wird aktiv im laufenden Betrieb überprüft, ob das Systemverhalten normal ist.
Stichwort: Virenchecker, Vier-Augen-Prinzip, Intrusion-Detection

Verschlüsselung Alle Daten werden verschlüsselt. Ohne die Schlüssel sind die Daten nichts wert.
Stichwörter: ssh (secure shell), pgp (pretty good privacy), gpg (gnu privacy guard), https (http secure)

44.1 Systemsicherheit

44.1.1 Was ist zu schützen?

44-6

Rechner müssen geschützt werden.

1. Hardware kann ausfallen, was *Datenverlust* und/oder *Produktivitätsverlust* zur Folge hat.
2. Hardware kann sich *bösartig verhalten*: Moderne Rechner sind Mehr-Prozessor- und Mehr-Benutzer-Systeme. Sie können *selbstständig* mit anderen Rechnern kommunizieren. Dadurch kann ein Rechner *von außen übernommen werden*. Dies bedeutet, dass sich jemand als ein Benutzer ausgibt und dann dem Computer (bösartige) Befehle erteilt. Ist der Angreifer geschickt, so merkt man davon nichts. Solche Rechner heißen dann *Zombies*.

44-7

Daten sind wichtiger als Hardware.

Neulich auf einem Schild in einem kleinen Laden in Berlin:

> Gestern wurde in diesen Laden eingebrochen und mehrere Computer gestohlen. Die Computer sind uns egal, aber wir benötigen die Daten auf den Rechnern! Bitte, ihr Diebe, gebt uns die Daten zurück, Diskretion und eine Belohnung garantiert!

Praktisch alle Daten, die in den Systemen einer Firma lagern, sind schützenswert. Ein paar wichtige sind: Kundenkontaktdaten, Forschungsergebnisse, Lagerbestandsdatenbanken oder Buchhaltung. Diese Daten müssen nicht nur gegen Diebstahl, sondern auch gegen Verlust durch Feuer, etc. geschützt werden.

44-8

Private Daten müssen geschützt werden.

Zu Ihrer Privatsphäre gehört nicht nur Ihre Wohnung, sondern auch Ihre Festplatte. Das Bundesverfassungsgericht hat dies kürzlich sogar in einem Grundsatzurteil zu einem Grundrecht erhoben. Sie müssen aber Ihre Grundrechte auch selbst aktiv verteidigen. Viele Leute legen ihre Daten *völlig ungeschützt* und *für alle lesbar* ab.

Motto

Zeige mir deinen Web-Browser-Cache und ich sage dir, was für ein Mensch du bist.

Von jedem lesbare Daten von Informatikstudierenden an der TU Berlin. 44-9

```
murmel:~ tantau$ ssh conde.cs.tu-berlin.de
tantau@conde.cs.tu-berlin.de's password:
Last login: Tue Apr 29 13:31:48 2008 from murmel.tcs.uni-
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
conde tantau 1 (~): cslocate sex
...
/home/all/b/believer/.kde/share/apps/krn/alt.sex.homosexual
/home/all/b/belo/man/man1/sex.1
/home/all/m/mawia/bin/BORUSSIA/XP/profile/Cookies/mawia@counter6.sextracker[1].txt
/home/all/m/mawia/bin/BORUSSIA/XP/profile/Cookies/mawia@rb4.worldsex[2].txt
/home/all/m/mawia/bin/BORUSSIA/XP/profile/Cookies/mawia@sextracker[1].txt
/home/all/m/mayer/.kde/share/cache/favicons/sextensiv.com.png
/home/all/s/salou/.kde/share/cache/favicons/www.sexmosaic.com.png
/home/all/s/schaefni/ml_projekt/doku/emerkmalsextraktion.tex
/home/all/s/seemanns/.kde/share/cache/favicons/www.perfectsexteens.com.png
/home/all/s/seemanns/.kde/share/cache/favicons/www.sexycrazybabes.com.png
/home/all/t/tabet/BORUSSIA/DOTNET/profile/Cookies/tabet@counter.sexsuche[1].txt
/home/all/t/thommy/.sigfiles/sig.Linuxsex
/home/all/t/thommy/.sigfiles/sig.alt.sex
/home/all/t/thommy/.sigfiles/sig.computer.sex
/home/cis/cissoft/.netscape/xover-cache/host-/alt.homosexual.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.politics.homosexuality.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.politics.sex.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.sex.bestiality.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.sex.bondage.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.sex.graphics.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.sex.homosexual.snm
/home/cis/cissoft/.netscape/xover-cache/host-/alt.sex.masturbation.snm
...
```

44.1.2 Wovor ist zu schützen?

Die größte Sicherheitslücke: Der Mensch. 44-10

Die größte Gefahr für Systeme geht häufig von den *Benutzern* aus. In Unternehmen können die *eigenen Mitarbeiter* ihren Zugriff nutzen, um Daten oder gleich die ganze Hardware zu stehlen. Menschen können *auf Zettel aufgeschriebene Passwörter* ausspionieren. Menschen benutzen oft ganz *leicht zu erratende* Passwörter wie `gott` oder auch `26121975`.

Moral

Der »Faktor Mensch« muss in jedes Sicherheitskonzept einbezogen werden.

Die zweite Sicherheitslücke: Bösartige Programme. 44-11

Damit bösartige Software überhaupt zum Zuge kommt, müssen sie erstmal *ausgeführt* werden. Normalerweise geschieht dies *nicht freiwillig* durch den Benutzer oder das Betriebssystem. Vielmehr nutzt bösartige Software so genannten *Sicherheitslöcher* aus (dazu gleich mehr).

Glossar der Schädlinge. 44-12

Wurm Programm, das sich selbstständig über ein Netzwerk ausbreitet, indem es Kopien von sich selbst an andere Rechner schickt.

Trojaner Programm, das etwas sinnvolles oder hübsches macht, aber eine Schadensroutine enthält (normalerweise den Rechner zum Zombie macht).

Virus Programmteil, der eine Kopie von sich selbst an andere Programme anhängt und immer dann gestartet wird, wenn ein infiziertes Programm gestartet wird.

✎ Zur Diskussion

Wie bekommt man diese Schädlinge?

44-13

Public domain

Leider kein Science-Fiction: Der Wurm der Apokalypse.
Eine Studie der Wurmforscher des ICSI (International Computer Science Institute, Berkeley) ergab 2005 folgendes: Würmer sind in der Regel *extrem schlecht* und schlampig programmiert. Ein *sehr gut programmierter Wurm*, der alle bekannten Tricks nutzt und in ein einziges IP-Paket passt, könnte sich in ca. *30 Sekunden weltweit* ausbreiten. *In wenigen Minuten* könnte er das Internet komplett lahmlegen. Lädt er noch einen Firmware-Flasher nach (sehr schwierig), dann könnte er die weltweite IT-Infrastruktur *für Wochen lahmlegen*. Die Folgen für die Weltwirtschaft könnte man wohl als apokalyptisch bezeichnen.

44.1.3 Welche Maßnahmen helfen?

44-14

Gegen Datenverlust helfen nur Sicherungskopien.
Daten müssen regelmäßig gesichert werden. Punkt.

44-15

Glossar der Sicherungsmaßnahmen.

Kennwörter Systeme werden zum Schutz in verschiedene *Bereiche* aufgeteilt, zu denen man nur mittels des richtigen Kennworts Zugriff bekommt.

Firewall Programm oder Rechner, der die Verbindung eines Rechners oder eines Teilnetzes zum Internet überwacht. Er lässt nur als *sicher eingestufte* und *vertrauenswürdige* Kommunikation zu.

Virenscanner Programm, das Speicher und Festplatte nach den ihm bekannten Viren, Würmern oder Trojanern durchsucht.

IDS Intrusion-Detection-Systeme beobachten das Verhalten von Rechner(netzen). Im Falle von auffälligem Verhalten (beispielsweise massenhafte E-Mails) wird der Rechner gestoppt oder verlangsamt.

44.2 Fallbeispiel eines Sicherheitslochs

44.2.1 Die Methode: SQL-Injection

44-16

Das Sicherheitsloch »SQL-Injection«.
SQL-Injection ist ein Methode, Schwachstellen von *Web-Servern* auszunutzen, die auf eine SQL-*Datenbank* zugreifen. Die *Angreifer* sind Menschen oder Computer. Die *Angegriffenen* sind Web-Server. Ziel des Angreifers ist es, den Web-Server dazu zu bringen, dass er SQL-*Code des Angreifers ausführt*. Man sagt, der Angreifer *»injiziert* SQL-*Code in den Web-Server«*, daher der Name.

44-17

Ablauf einer SQL-Injection.

Normale Kommunikation

1. Nutzer trägt Daten in ein Web-Formular ein
2. Web-Client schickt Formular-Daten an den Web-Server
3. Web-Server führt daraufhin einen SQL-Befehl aus, um Daten aus der Datenbank zu holen
4. Web-Server schickt Antwort an den Web-Client

Kommunikation mit SQL-Injection

1. Nutzer trägt *ungewöhnliche Daten* in ein Web-Formular ein
2. Web-Client schickt Formular-Daten an den Web-Server
3. Web-Server führt SQL-Befehl aus, der aber aufgrund der ungewöhnlichen Daten *ungewöhnliche Effekte* hat.
4. Web-Server schickt *nicht gewollte* Antwort an den Web-Client

44.2.2 Fiktives Beispiel: Firmen-Intranet

Das Intranet von Molecular Sheep.

44-18

Die Firma Molecular Sheep verfügt über ein *Intranet*. Dieses bietet Zugang auf firmeninterne Daten (wie die Fellfarben von Dolly und Flauschi). Man muss sich *authentifizieren*, um Einlass zu erhalten. Die Liste der berechtigten Personen und deren Passwörter ist in einer Datenbanktabelle gespeichert. Leider wurde an der Sicherheit gespart, weshalb die Eingangskontrolle schlampig programmiert wurde.

Der Login-Vorgang von Molecular Sheep.

44-19

```
<!-- HTML-Login-Seite --!>
<form action="http://molecular-sheep.com/login.java" method="post">
  <p>User:     <input name="user" type="text"/> </p>
  <p>Password: <input name="pass" type="text"/> </p>
  <p><input name="submitButton" value="Login" type="submit"/></p>
</form>
```

Wenn sich User `ich` mit dem Passwort `gott` einloggt, wird folgende Anfrage an den Web-Server von Molecular-Sheep geschickt:

```
http://molecular-sheep.com/login.java?user=ich&pass=gott
```

Daraufhin ruft der Web-Server das Programm `login.java` auf mit den Parametern `ich` und `gott` auf. Darin:

```
boolean checkPassword (String user, String password) {
  ...
  String sqlQuery =
    "select␣*␣from␣password_table␣where␣user_name=\"" +
    user + "\"␣and␣password=\"" + password + "\"";

  Statement statement = connection.createStatement();
  statement.executeQuery (sqlQuery);
  if (statement.getMoreResults () == false)
    return false;
  else
    return true;
}
```

Das Sicherheitsloch von Molecular-Sheep.

44-20

Login für User `ich` mit Passwort `gott`

Der `sqlString` lautet

```
select * from password_table where
  user_name="ich" and password="gott";
```

Dies liefert genau fann mehr als null Treffen, wenn es den passenden Eintrag in der Tabelle `password_table` gibt.

Login mit SQL-Injection als User mit leerem Passwort

Ein Angreifer tippt nun als »Benutzernamen« folgendes ein:

```
egal" or true; --
```

Dann wird folgender SQL-Befehl ausgeführt:

```
select * from password_table where
  user_name="egal" or true; --" and password="";
```

Da `--` einen Kommentar in SQL beginnt, liefert dies immer Treffer.

Moral

44-21

1. Vertraue *niemals* Benutzereingaben!
2. Benutzereingaben dürfen *niemals* ohne besondere Vorkehrungen mit Befehlen vermischt werden.

44.2.3 Reales Beispiel: www.doc.state.ok.us

44-22

Little Shop of Horror für Datenschützer: Webseite des Department of Corrections, Oklahoma.

Im US-Staat Oklahoma sind (im Jahr 2008) die Insassen *aller Gefängnisse* über das Internet zugreifbar. Für *jeden Gefangenen* sind über ein *komfortable Suchfunktion* folgende Informationen bequem abrufbar:

- Name, Geburtsdatum, Rasse (!),
- Foto(s) des Gefangenen (!!),
- Komplettes Vorstrafenregister.

Für ehemalige Sexualstraftäter sind auch *nach der Entlassung (für mindestens 15 Jahre bis lebenslang)* verfügbar:

- aktuelle Anschrift,
- Telefonnummer.

Sexualstraftaten sind dort neben Vergewaltigung auch »distribution of obscene videos« (= Verkauf von Pornos). Sehr liebevoll gemacht ist auch die Seite mit dem *Hinrichtungs-Countdown* für die Menschen im Todestrakt. (Mit dem Betreiben einer solchen Seite würden Sie sich in Deutschland strafbar machen.)

44-23

Das Sicherheitsloch des DOC Oklahoma.

Die Informationen über die (ehemaligen) Gefangenen stehen in einer relationalen Datenbank. Die Suche in diesen Daten wird durch eine SQL-Anfrage bewerkstelligt. Das (absolut vollkommen unvorstellbar gigantische) Sicherheitsproblem besteht darin, dass die SQL-Anfrage in einen Link eingebettet ist. Dieses Sicherheitsproblem bestand zwischen den Jahren 2005 und 2008. Das DOC wurde auf das Problem aufmerksam gemacht, reagierte durch (völlig nutzlose) Änderungen der SQL-Anfrage. Das DOC löste das Problem erst, als man ihm die (ebenfalls in der Datenbank gespeicherte) Liste der medizinischen Behandlungen des Personals des DOC schickte.

44-24

Der Link im Original, umformatiert.

```
<a href="http://docapp8.doc.state.ok.us/pls/portal30/url/page/sor_roster?sqlString=
     select distinct
       o.offender_id,doc_number,o.social_security_number, o.date_of_birth,
       o.first_name,o.middle_name,o.last_name,o.sir_name,sor_data.getCD(race) race,
       sor_data.getCD(sex) sex,l.address1 address,l.city,l.state stateid,l.zip,
       l.county,sor_data.getCD(l.state) state,l.country countryid,
       sor_data.getCD(l.country) country,decode(habitual,'Y','habitual','')
       habitual,decode(aggravated,'Y','aggravated','') aggravated,
       l.status,x.status,x.registration_date,x.end_registration_date,l.jurisdiction
     from registration_offender_xref x, sor_last_locn_v lastLocn, sor_offender o,
       sor_location l, (select distinct offender_id
                        from sor_location
                        where status = 'Verified' and upper(zip) = '73064' ) h
     where lastLocn.offender_id(%2B) = o.offender_id and
           l.location_id(%2B) = lastLocn.location_id and
           x.offender_id = o.offender_id and
           x.status not in ('Merged') and x.REG_TYPE_ID = 1 and
           nvl(x.admin_validated,to_date(1,'J')) >= nvl(x.entry_date,to_date(1,'J'))
           and x.status = 'Active' and x.status <> 'Deleted' and
           h.offender_id = o.offender_id
     order by o.last_name,o.first_name,o.middle_name&sr=yes">
Print Friendly </a>
```

44-25

✎ Zur Übung

Das Sicherheitsloch (eher Sicherheitsabgrund) dieser Webseite lässt sich ausnutzen, indem Sie einfach eine eigene Web-Seite erstellen, auf der Sie den Link kopieren, dann aber den Link-Text an entscheidenden Stellen ändern.
Was müssen Sie ändern, um

1. die Liste der Gefangenen zu bekommen, die eigentlich von der Liste gestrichen sind?
2. herauszubekommen, welche anderen interessanten Tabellen lesbar sind?
3. Gefangene aus der Datenbank zu löschen?
4. fiktive Gefangen in die Datenbank einzutragen?
5. dem Spuk ein Ende zu bereiten und die ganze Datenbank zu löschen?

Zusammenfassung dieses Kapitels

44-26

1. *Daten, Kommunikation und Rechner* sind vielfältigen Gefahren ausgesetzt, die hauptsächlich von (unvorsichtigen, bösartigen oder dummen) *Menschen* ausgehen sowie von *Würmern*.
2. Gute Passwörter, regelmäßiges Schließen von Sicherheitslöchern und die Benutzung von Verschlüsselung bieten *guten, aber längst nicht perfekten Schutz*.
3. Seine Daten zu schützen sollte genauso selbstverständlich sein, wie das Abschließen der Wohnungstür oder des Fahrrades.
4. Der Staat Oklahoma ist allerdings der Meinung, dass all dies für ihn nicht gilt.

Übungen zu diesem Kapitel

Übung 44.1 SQL-Injection-Schwachstelle erkennen und beheben, mittel

Auf dem Server eines Online-Buchversands liegen Java-Programme, die Benutzereingaben eines Formulars im HTML-Format auswerten und entsprechende Datenbankabfragen ausführen. Dabei wird unter anderem folgende Methode aufgerufen:

```
String buecherliste( String suchbegriff, String maximale_anzahl ){
        ResultList r = Database.executeQuery( "SELECT␣*␣FROM␣books␣"
                +"WHERE␣title␣LIKE␣'%"+suchbegriff+"%'␣"
                +"LIMIT␣"+maximale_anzahl );
        String html = formatiere_resultate( r );
        return html;
}
```

1. Geben Sie drei verschiedene Möglichkeiten an, wie durch Benutzereingaben im HTML-Formular die Tabelle `books` aus der Datenbank gelöscht werden kann.
2. Erweitern Sie die Methode `buecherliste` um zusätzliche Befehle zur Vorverarbeitung der Eingabe, durch die die SQL-Injection-Schwachstelle behoben wird. Legitime Benutzereingaben sollen allerdings durch diese zusätzlichen Sicherheitsmaßnahmen nicht beeinträchtigt werden.

Prüfungsaufgaben zu diesem Kapitel

Übung 44.2 Absichten von E-Mail-Scams einschätzen, leicht, original Klausuraufgabe

Sie erhalten drei E-Mails, die in böswilliger Absicht an Sie geschrieben wurden.

1. Absender ist angeblich Ihre Bank. Sie werden aufgefordert, eine Website anzusurfen und Ihre Konto-Daten zu überprüfen beziehungsweise zu aktualisieren.
2. Absender ist angeblich Toni. Sie werden aufgefordert, die Datei Geburtstagsfeier.exe im Anhang der Mail zu öffnen.
3. Absender ist angeblich ein Königssohn aus Nigeria. Sie werden herzlich gebeten zu helfen, einen Geldtranfer nach Europa, den die nigerianische Regierung behindert, möglich zu machen. Nehmen Sie bitte E-Mail-Kontakt zum Absender auf.

Geben Sie für jede der drei E-Mails an, ob eine oder einige der folgenden drei Absichten jeweils dahinter stecken könnte.

1. Ausspionieren von Passwörtern.
2. Verbreiten eines Wurms.
3. Missbrauch des PC für eine DoS-Attacke.

Anhang

Lösungen

Beispiellösungen zu ausgesuchten Übungen

Beispiellösung zu Übung 1.6

Das Genom benötigt 0,75 Gigabyte und somit einen Speicherriegel.

Beispiellösung zu Übung 1.7

1. 24 Bit / Pixel = 3 Byte / Pixel
   $1024 \cdot 1024 \cdot 3$ Byte / Bild = 3 Megabyte / Bild
   $20 \cdot 25 \cdot 3$ Megabyte / Film = 1500 Megabyte / Film
2.

Medium	Verfahren
CD	Verfahren A
DVD	keine Kompression

Beispiellösung zu Übung 3.5

LaTeX ist kein Betriebssystem. Die Verwaltung des Dateibaums ist eine der Hauptaufgaben eines Betriebssystems.

Beispiellösung zu Übung 4.11

1.

Datei/Verzeichnis	Rechte
`/home/ihrname`	`rwx--x---`
`projects`	`rwx------`
`protokoll.pdf`	`rwxr-----`

2.
```
cd /home/ihrname
chmod go-rwx projects
chmod go-rwx protokoll.pdf
chmod g+r protokoll.pdf
chmod g+x .
```

Beispiellösung zu Übung 4.12

```
head -1 $1 > $1.zeile1
cat $1 $1.zeile1 > $1.erg
cp $1.erg $1
rm $1.zeile1 $1.erg
```

Beispiellösung zu Übung 4.15

1. `grep Minka nummern.txt`
2. `grep $1 nummern.txt`
3. `tail -1 gehaltsliste.txt > gehaelter.txt`
4. `tail -1 gehaltsliste.txt | mean`

Beispiellösung zu Übung 4.16

1. `wc -l b.txt`
2. `grep "^5$" b.txt | wc -l`
3. `cat a.txt b.txt > c.txt` und `variance c.txt`
4. `cat a.txt b.txt | variance`

Beispiellösung zu Übung 5.5

```
([02468]|[1-9][0-9]*[02468])
```

Beispiellösung zu Übung 5.6

Kleiner Scherz… Im Text stand doch, dass solche Aufgaben nicht in der Klausur vorkommen.

Beispiellösung zu Übung 9.1

1 *kontostand_nach_einem_jahr* ← *anfangskontostand* ·1,04
2 *kontostand_nach_zwei_jahren* ← *kontostand_nach_einem_jahr* ·1,04
3 *loesung* ← *kontostand_nach_zwei_jahren* ·1,04

Beispiellösung zu Übung 9.1

1 *loesung* ← *anfangskontostand* ·1,04 ·1,04 ·1,04

Beispiellösung zu Übung 10.1

```
// In anfangskontostand steht ein Wert
int anfangskontostand = 1000;

// Erstmal geben wir an, welche Variablen wir brauchen.
int kontostand_nach_einem_jahr;
int kontostand_nach_zwei_jahren;
int kontostand_nach_drei_jahren;

// Jetzt wird gerechnet:
kontostand_nach_einem_jahr = anfangskontostand * 1.04;
kontostand_nach_zwei_jahren = kontostand_nach_einem_jahr * 1.04;
kontostand_nach_drei_jahren = kontostand_nach_zwei_jahren * 1.04;

loesung = kontostand_nach_drei_jahren;
System.out.println(loesung);
```

Beispiellösung zu Übung 10.6

1. Die folgenden Zeichen wurden vergessen:
   1.1 `.` in Zeile 2,
   1.2 `{` in Zeile 6,
   1.3 `)` in Zeile 7.
2. Es wird die größte Zahl im Array zurückgegeben, und 0 falls das Array die Länge 0 hat.

Beispiellösung zu Übung 11.11

Die Typen der Teilausdrücke lauten:

1. **`double`**
2. **`double`**
3. **`boolean`**
4. **`boolean`**
5. **`boolean`**
6. **`boolean`**

Beispiellösung zu Übung 11.12

Die Typen der Teilausdrücke laufen:

1. `String`
2. `String`
3. **`int`**
4. **`double`**
5. **`boolean`**

Beispiellösung zu Übung 13.7

1.
```
int[] b = new int[2*a.length];
```
und
```
for( int i = 0 ; i < a.length ; i ++ )
```

2.
```
{1,1,2,2,3,3}
```

Beispiellösung zu Übung 13.9

```
int[] nucleotideFrequency( String s ){
  int[] r = new int[4];

  for( int i = 0 ; i < s.length() ; i ++ ){
    if( s.charAt( i ) == 'A' ){
      r[0] = r[0] + 1;
    }
    if( s.charAt( i ) == 'C' ){
      r[1] = r[1] + 1;
    }
    if( s.charAt( i ) == 'G' ){
      r[2] = r[2] + 1;
    }
    if( s.charAt( i ) == 'T' ){
      r[3] = r[3] + 1;
    }
  }

  return r;
}
```

Beispiellösung zu Übung 14.1

Bei Stelle 1:

```
w == "TOLL"
x == "INFO A IST"
y == "ECHT"
z == "TOLL"
tmp == "CHT"
```

Bei Stelle 2:

```
w == "ACCGCGTAG"
x == "TGGCGCATC"
y == ""
z == "GATGCGCCA"
tmp == "CTACGCGGT"
```

Beispiellösung zu Übung 15.12

```
boolean isMoreCG(String s) {
  int countCG = countChar(s, 'C') + countChar(s, 'G');
  int countAT = countChar(s, 'A') + countChar(s, 'T');
  return countCG > countAT;
}
```

Beispiellösung zu Übung 19.15

```
static int absolute_sum_r(int[] a, int n) {
  if (n == 0) {
    return 0;
  } else {
    if (a[n-1] > 0) {
      return a[n-1] + absolute_sum_r(a, n-1);
    } else {
      return -1*a[n-1] + absolute_sum_r(a, n-1);
    }
  }
}
```

Beispiellösung zu Übung 20.4

1. 42
2. `{36, 1, 17, 28, 42, 96, 83}`
   Abhängig von der Quicksort-Implementierung können hier auch andere Ergebnisse herauskommen. Richtig ist letztendlich alles, wo die 42 an drittletzter Stelle steht und von der 83 und der 96 gefolgt wird.
3. ```
   quicksort (a, 0, 3);
   quicksort (a, 5, 6);
   ```

Beispiellösung zu Übung 20.6

1. 1.1 Teilarray 1 von Stelle 0 bis Stelle 4,
   1.2 Teilarray 2 von Stelle 6 bis Stelle 6.
2. Wenn die Daten völlig zufällig sind, ist es egal, welches Element das Pivot-Element wird. Oft sind Daten aber schon grob vorsortiert, z.B. eine Liste die schon einmal sortiert wurde und in die dann einige neue Elemente eingefügt wurden. Dann ist es günstiger, das mittlere Element zu wählen, da dies dann tatsächlich etwa der Median des Arrays ist.

Beispiellösung zu Übung 24.3

Die Klassen `Mitglied` und `Netzwerk`:

```
class Mitglied {
  int alter;
  char geschlecht;
  Mitglied[] freunde;

  Mitglied(int alter, char geschlecht, int anzahlFreunde) {
    this.alter = alter;
    this.geschlecht = geschlecht;
    freunde = new Mitglied[anzahlFreunde];
  }
}

class Netwerk {
  Mitglied[] mitglieder;
}
```

Der Code mit dem das dargestellte Netzwerk erzeugt wird:

```
Netzwerk schludriVZ = new Netzwerk();
schludriVZ.mitglieder = new Mitglied[3];

Mitglied bruno = new Mitglied(30, 'm', 1);
Mitglied anna = new Mitglied(23, 'w', 2);
Mitglied cesar = new Mitglied(18, 'm', 1);

bruno.freunde[0] = anna;
anna.freunde[0] = bruno;
anna.freunde[1] = cesar;
cesar.freunde[0] = anna;

schludriVZ.mitglied[0] = bruno;
schludriVZ.mitglied[1] = anna;
schludriVZ.mitglied[2] = cesar;
```

Beispiellösung zu Übung 24.4

Die Deklaration der Klassen `Tafelrunde` und `Ritter`:

```
class Tafelrunde {
  Ritter[] ritter;
}

class Ritter {
  char himmelsrichtung;
  Ritter linkerRitter;
  Ritter rechterRitter;
```
```

```
  Ritter(char himmelsrichtung, Ritter linkerRitter, Ritter rechterRitter) {
    this.himmelsrichtung = himmelsrichtung;
    this.linkerRitter = linkerRitter;
    this.rechterRitter = rechterRitter;
  }
}
```

Der Javacode, mit dem die dargestellte Tafelrunde erzeugt wird:

```
Tafelrunde runde = new Tafelrunde();
runde.ritter = new Ritter[4];

Ritter gareth = new Ritter('O', null, null);
Ritter gawain = new Ritter('O', null, null);
Ritter lancelot = new Ritter('W', gawain, gareth);
Ritter Tristan = new Ritter('W', gareth, gawain);
gawain.linkerRitter = tristan;
gawain.rechterRitter = lancelot;
gareth.linkerRitter = lancelot;
gareth.rechterRitter = tristan;

runde.ritter[0] = gareth;
runde.ritter[1] = lancelot;
runde.ritter[2] = gawain;
runde.ritter[4] = tristan;
```

Der Javacode der `pruefen`-Methode:

```
boolean pruefen() {
  Ritter r;
  for (int i = 0; i < ritter.length; i++) {
    r = ritter[i];
    if (r.himmelsrichtung == 'O') {
      if (r.linkerRitter.himmelsrichtung == 'O' ||
          r.rechterRitter.himmelsrichtung == 'O') {
        return false;
      }
    } else {
      if (r.linkerRitter.himmelsrichtung == 'W' ||
          r.rechterRitter.himmelsrichtung == 'W') {
        return false;
      }
    }
  }
  return true;
}
```

Beispiellösung zu Übung 24.5

Die Attribute und Konstruktoren:

```
class Semester {
  Gruppe[] gruppen;

  Semester(Gruppe[] gruppen) {
    this.gruppen = gruppen;
  }
}

class Gruppe {
  Person[] personen;

  Gruppe(Person[] personen) {
    this.personen = personen;
  }
}
```

Die `main`-Methode in der `Gruppenverwaltung`:

```
class Gruppenverwaltung{
  public static void main( String[] args ){
    Person mueller = new Person("Fritz", "Müller");
    Person meier = new Person("Hans", "Meier");
    Person einsam = new Person("Ute", "Einsam");
```

```
    Person klingel = new Person("Kerstin", "Klingel");
    Person stock = new Person("Susi", "Stock");

    Person[] pg1 = { mueller, meier };
    Person[] pg2 = { einsam };
    Person[] pg3 = { klingel, stock };

    Gruppe g1 = new Gruppe(pg1);
    Gruppe g2 = new Gruppe(pg2);
    Gruppe g3 = new Gruppe(pg3);

    Gruppe[] sg = { g1, g2, g3 };

    Semester s1 = new Semester(sg);
  }
}
```

Beispiellösung zu Übung 25.1

```
public class DNASequenceRing
{
   Cell start;
}

public class Cell {
   public char base;
   public Cell next;
}
```

Beispiellösung zu Übung 25.2

```
DNASequenceRing der_eine_ring = new DNASequenceRing();
der_eine_ring.start = new Cell();
der_eine_ring.start.base = 'A';
der_eine_ring.start.next = der_eine_ring.start;
```

Beispiellösung zu Übung 25.7

1.
```
class Queue {
  Person erster;
  Person letzter;
}

class Person {
  String vorname;
  String nachname;
  Person vordermann;

  public Person (String vorname, String nachname) {
    this.vorname = vorname;
    this.nachname = nachname;
  }
}
```

2.
```
Person till = new Person ("Till", "Tantau");
Person jan = new Person("Jan", "Arpe");
Person johannes = new Person("Johannes", "Textor");

Queue baecker = new Queue();

johannes.vordermann = jan;
jan.vordermann = till;
baecker.erster = till;
baecker.letzter = johannes;
```

Beispiellösung zu Übung 26.1

```
int length ()
{
  // Ein leerer Ring wird gesondert behandelt:
  if (this.start == null)
    return 0;

  Cell current = this.start;
  int gesehene_elemente = 1;
  while (current.next != this.start) {
    // Ok, current ist nicht das letzte Element.

    // Schiebe current eine Zelle weiter:
    current = current.next;

    // Vermerke, das wir etwas Neues gesehen haben
    gesehene_elemente++;
  }

  return gesehene_elemente;
}
```

Beispiellösung zu Übung 26.5

```
void truncate( int n )
{
  if (n == 0) {
    this.start = null;
    return;
  }

  Cell cursor = this.start;
  for( int i = 0 ; i < n-1 ; i ++ )
    cursor = cursor.next;
  cursor.next = null;
}
```

Beispiellösung zu Übung 26.6

```
void deleteZeroes()
{
  Cell cursor = this.start;
  while( cursor.next != null ) {
    if( cursor.next.number == 0 ) {
      cursor.next = cursor.next.next;
    }
    else {
      cursor = cursor.next;
    }
  }
}
```

Beispiellösung zu Übung 26.13

```
boolean hasEvenLength(){
  Cell cursor = start;
  int count = 0;

  while( cursor != null ){
    count ++;
    cursor = cursor.next;
  }

  return count % 2 == 0;
}
```

Beispiellösung zu Übung 26.14

```
Cell get( int i ){
  Cell cursor = this.start;
  while( cursor != null && i > 0 ){
    cursor = cursor.next;
    i = i - 1;
  }
  return cursor;
}
```

Beispiellösung zu Übung 27.2

```
class Tree {

  public int sum() {
    return sumOfChildrenFrom (this.root);
  }

  private int sumOfChildrenFrom(Node n) {
    if (n == null) {
      return 0;
    }
    else {
      return
        n.number +
        sumOfChildrenFrom(n.left) +
        sumOfChildrenFrom(n.right);
    }
  }
}
```

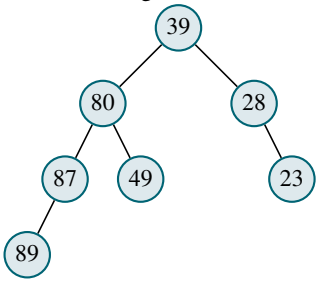
Beispiellösung zu Übung 27.5

```
static int maxSumOnPath(Node root) {
  if (root == null)
    return 0;

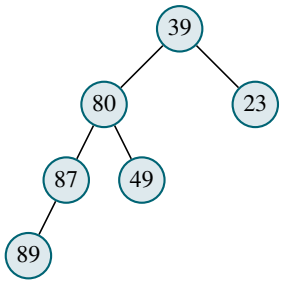
  int leftSum = maxSumOnPath(root.left);
  int rightSum = maxSumOnPath(root.right);
  if (leftSum > rightSum) {
    return root.value + leftSum;
  }
  else {
    return root.value + rightSum;
  }
}
```

Beispiellösung zu Übung 28.2

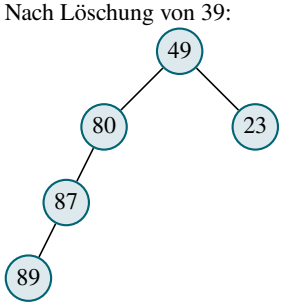
1. Nach Löschung von 59:



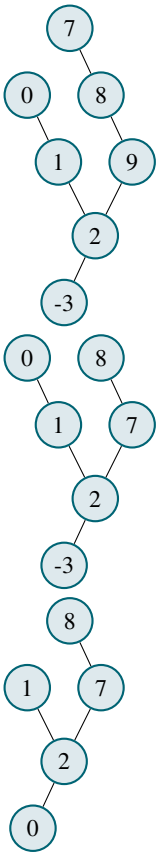
2. Nach Löschung von 28:



3. Nach Löschung von 39:

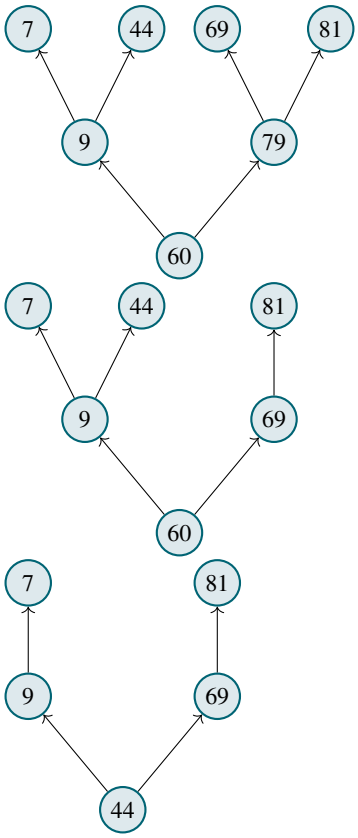


Beispiellösung zu Übung 28.4

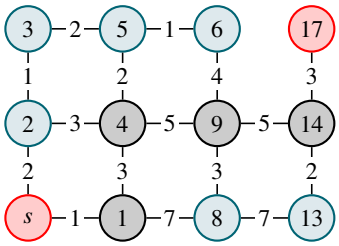


Beispiellösung zu Übung 28.5

1. 60
2. 7, 44, 69, 86
3. 3
4. Die Bäume sehen nach den einzelnen Löschschritten so aus:



Beispiellösung zu Übung 32.2



Beispiellösung zu Übung 33.7

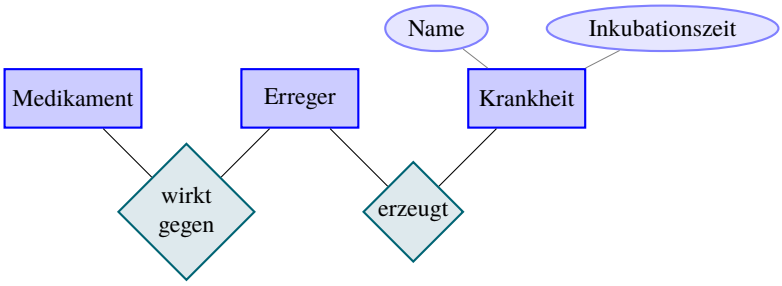
Projekt	Gewinn	Tage	Tageslohn
1	3100	31	100
2	2970	30	99
3	2970	30	99

Die Greedy-Heuristik wählt Projekt 1 aus, womit keines der anderen Projekte mehr im Zeitlimit bearbeitet werden kann. Die optimale Lösung ist die Wahl von Projekt 2 und Projekt 3. Die Güte der durch die Greedy-Heuristik gelieferten Approximation beträgt $5940/3100$, also ungefähr $1,92$.

Beispiellösung zu Übung 33.8

```
int greedyGewinn( int[] tageslohn, int[] tage, int arbeitstage gesamt ){
   int gewinn = 0;
   int verbrauchte tage = 0;
   for( int i = 0 ; i < tageslohn.length ; i ++ ){
      if( verbrauchte tage + tage[i] <= arbeitstage gesamt ){
         gewinn = gewinn + tageslohn[i] * tage[i];
         verbrauchte tage = verbrauchte tage + tage[i];
      }
   }
   return gewinn;
}
```

Beispiellösung zu Übung 35.2



Beispiellösung zu Übung 36.5

Tabelle 1: `laender`

Spaltenname	Datentyp
`id`	`varchar(255)`
`name`	`varchar(255)`
`hauptstadt`	`varchar(255)`
`einwohnerzahl`	`int`

Tabelle 2: `nachbarschaft`

Spaltenname	Datentyp
`land_1_id`	`int`
`land_2_id`	`int`

Beispiellösung zu Übung ??

1. 1.1 Anzahl der Bücher in der Datenbank (hier: 3)
   1.2 ISBN der Bücher, deren Titel »Algorithms« enthält (hier: 0-262-03141-8)
2. 2.1

```
SELECT name FROM authors
           WHERE name LIKE "R%"
           ORDER BY name ASC
```

   2.2

```
SELECT authors.name FROM authors, books
                    WHERE authors.book_id = books.id
                          AND books.title = "Immunobiology"
```

   2.3

```
SELECT books.title FROM authors, books
                   WHERE authors.book_id = books.id AND
                       authors.name = "Rivest"
```