Einführung in die Informatik
Inf, SAT, Wing, Mi

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- mwilhelm@hs-harz.de
- http://www.miwilhelm.de
- Raum 2.202
- Tel. 03943 / 659 338

Inhalt

1. Grundlagen der Informatik

   Zahlensysteme, Nachkommazahlen, Single-Format

2. Rechnerarchitekturen (Hard- und Softwaresysteme)

   Rechnerarchitekturen, Betriebssysteme

3. **Programmierung**

   Sprachen, Datentypen, Datenstrukturen, Programmiermodelle, Compiler, Programmierparadigmen, Domänenspezifische Sprachen, Algorithmen

4. Verteilte Systeme (Netzwerke)

5. Themenfelder der Informatik

   Datenbanken, Big Data, Multimediaverarbeitung, Software Engineering, KI, Human Computing Interface, Robotics, VR/AR

Sprachen

- 1. Generation: Maschinensprachen (hexadezimal)
- 2. Generation: Assembler (Mnemonic-Gedächtnisstütze)
- 3. Generation: Hochsprachen (Imperative Sprachen)
- 4. Generation: Module
- 5. Generation: Objekte
- Weitere Generationen:
  - KI-Sprachen
  - Web-Sprachen
  - Skriptsprachen
  - Funktionale Sprachen
  - Esoterische Programmiersprachen
  - Echtzeit Programmiersprachen

Semantische Lücke

- Die „Semantische Lücke“ beschreibt den semantischen, also bedeutungsbezogenen Unterschied zwischen zwei Beschreibungen eines Objekts, der dadurch entsteht, dass verschiedene Repräsentationsformen (Sprachen) gewählt werden. Dieser in der Informatik verwendete Begriff wird im Allgemeinen dort deutlich, wo ein Abbild des realen Lebens in eine formale, maschinell verarbeitbare Repräsentation übertragen werden muss.

- Präziser bezeichnet der Begriff den Unterschied zwischen Formulierung von Kontextwissen in einer **mächtigen Sprache** (z. B. Natürliche Sprache) und dessen formaler und automatisiert reproduzierbaren Repräsentation in einer **weniger mächtigen formalen** Sprache (z. B. Programmiersprache). In der natürlichen Sprache lassen sich Zusammenhänge ausdrücken, die in einer formalen Sprache nicht auswertbar sind. Aus diesem Grund ist der Unterschied der Ausdrucksmächtigkeit auch nicht formal beschreibbar.

1. Generation:Maschinensprachen

- Jeder Befehl der CPU ist binär/hexadezimal codiert.
- Eingabe über Schalter (sehr schnell)!
- Nur Hauptspeicher
- Es werden die Register der CPU benutzt.
- ca. 60 KB RAM
- Beispiele
  - 8ED8 MOV DS,AX
  - B0FE MOV AL, 0FEH
  - 2AC3 SUB AL, BL
  - 3AC3 CMP AL, BL
  - B3FD MOV, BL, -3

1. Generation:Maschinensprachen

```
C:\WINDOWS\system32\cmd.exe
 -  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help
[■]=CPU Pentium Pro
#a1#start: start: mov ax,data_seg
  cs:0000►B8520C          mov     ax,0C52
#a1#18:  mov ds,ax
  cs:0003 8ED8            mov     ds,ax
#a1#20:  mov bl,-3
  cs:0005 B3FD            mov     bl,FD
#a1#21:  mov ax,7812h
  cs:0007 B81278          mov     ax,7812
#a1#22:  mov bx,6712h
  cs:000A BB1267          mov     bx,6712
#a1#23:  add ax,bx
  cs:000D 03C3            add     ax,bx
#a1#25:  mov ax,0F876h
  cs:000F B876F8          mov     ax,F876
#a1#26:  mov bl,08Fh
  cs:0012 B38F            mov     bl,8F
#a1#27:  idiv bl
  cs:0014 F6FB            idiv    bl
#a1#29:  mov ax,9876h
  cs:0016 B87698          mov     ax,9876
#a1#30:  mov bx,7654h
  cs:0019 BB5476          mov     bx,7654
  ds:0000 CD 20 FB 9F 00 9A F0 FE = ╦f Ü-■
  ds:0008 1D F0 32 0B 33 09 0F 07 ↔-2♂3○☼•
  ds:0010 B7 06 56 01 E5 03 9A 06 À♠V☺σ♥Ü♠
ax 0000
bx 0000
cx 0000
dx 0000
si 0000
di 0000
bp 0000
sp 0064
ds 0C3B
es 0C3B
ss 0C4B
cs 0C55
ip 0000
ss:007E
ss:007C
ss:007A
ss:0078
ss:0076
ss:0074
ss:0072
ss:0070
ss:006E
ss:006C
```

2. Generation: Assembler mit Mnemonic

- Jeder Befehl des Programm wird mit Buchstaben codiert.
  - ADD Add
  - CMP Compare
  - DIV Divide (unsigned)
  - IDIV Integer divide (signed)
  - IMUL Integer multiply (signed)
  - MUL Muliply (unsigned)
- Ein Compiler übersetzt die Assembler-Befehle in echten Maschinencode.
- Es werden die Register der CPU benutzt.
- Sprungmarken werden in Adressen umgesetzt.

3. Generation: Hochsprachen

- Die Assemblerbefehle werden ersetzt durch eine „mathematische“ Notation. Man bezeichnet sie auch als (problemorientierten) imperativen Sprachen.
- Wichtigstes Merkmal ist die algorithmische Herangehensweise an Probleme.
  - **summe := summe + a;**
- Der Compiler ersetzt einen Befehl in mehrere Assembler-befehle.
- Bibliotheken werden beim „Linken“ hinzugefügt.
- Beispiele:
  - C
  - Turbo Pascal
  - Cobol
  - Fortran
  - Modula

3. Generation: Hochsprachen: Bedingungen

- **C:**

```
if (a==12 || a==13)  {
   c = a+b;
}
else {
  //
};
```

- **Cobol:**

```
IF a=12  OR a=13  THEN
   //
ELSE
   //
END-IF
```

- **Turbo Pascal:**

```
if (a=12) or (a=13) then
begin
   c := a+b;
end
else
begin
  //
end;
```

- **Python:**

```
if a==12 or a==13:
      # …
else:
      # …
```

3. Generation: Hochsprachen: Schleife

- **C:**

```
for (int i=0; i<10; i++)  {
  //
}
```

- **Turbo Pascal:**

```
for i:=0 to 10 do
begin
  //
end;
```

- **Python:**

```
for i in range(0,nLen-1):
    # …
```

- **Cobol:**

```
PERFORM WITH TEST AFTER UNTIL LAEUFER = 10
  DISPLAY LAEUFER
  ADD 1 TO LAEUFER
END-PERFORM
```

4. Generation: Objekte

- Die Module werden durch das Konzept „Klasse“ ersetzt.
  - Vererbung,
  - Abtrakte Klasse
  - Polymorphismus
  - Interface
-

Künstliche Intelligenz-Programmierung: KI

- Stichworte:
  - Neurale Netze
  - Wissensbasierte Systeme
  - Musteranalyse und Mustererkennung
  - Mustervorhersage
  - Robotik
  - Modellierung anhand der Entropiekraft
  - Künstliches Leben
- Beispiele:
  - **Lisp**
  - **Prolog** (Prädikatenkalküls erster Stufe)
  - Python
  - C++
  - Java
  - ELSA
  - Scry

Künstliche Intelligenz-Programmierung: KI

- Amazon:
  - Entwicklung eines System zur Bewertung einer Bewerbung (ab 2014).
  - Gefüttert mit Eingangsdaten.
  - „Erfolgreich“ eingesetzt.
  - Der Algorithmus hatte aber unerwünschte Nebenwirkungen.
    - Es wurden nur wenige Frauen eingestellt.
  - System entschied:
    - Mitglied in einer „Frauengemeinschaft“ ist nicht gut.
    - Selbst das Geschlecht wurde bewertet!

https://www.heise.de/newsticker/meldung/Amazon-KI-zur-Bewerbungspruefung-benachteiligte-Frauen-4189356.html

Künstliche Intelligenz-Programmierung: KI

Problem nicht unbekannt

- Amazons Erfahrungen mit maschinellem Lernen sind nicht neu und decken sich mit ähnlichen Ergebnissen in ganz anderen Bereichen.
- Zwar gibt es immer wieder Hoffnungen auf völlig neutrale und damit letztliche faire Entscheidungen durch Algorithmen, aber solange die Technik von Menschen programmiert wird und auf menschengemachten Datensätzen beruht, scheint das ein größeres Problem zu sein, als lange gedacht.
- So benachteiligen Algorithmen, die in den USA routinemäßig eingesetzt werden, um die Rückfallgefahr von Straftätern zu berechnen, systematisch Afroamerikaner.

4. Generation: Web-Sprachen

- Webserver entstanden mit dem Internet.
- Auf dem Server ist eine Software installiert, die Anfragen entgegennimmt und **dynamisch** die HTML-Seiten erstellt und an den Browser sendet.
- Beispiele:
  - PHP Nummer eins, einfach zu installieren
    - Framework: Symfony
    - schwache Typisierung
  - Pearl
  - Java Typsicher, komplizierte Installation
  - Javascript
  - Python
  - Ruby
  - ASP.net C#

https://www.computerwoche.de/a/web-programmiersprachen-im-vergleich,2486075

4. Generation: Web-Sprachen

	Java	PHP	Phyton	Ruby
Entwicklungsaufwand	0	++	++	++
Erweiterbarkeit	++	+	+	+
Wartung	+	0	++	++
Robustheit	+	0	+	0
Ausführungsgeschwindigkeit	+	+	0	-
Skalierbarkeit	++	+	0	0
Monitoring	++	0	0	0
Laufzeitumgebung	++	+	+	+
Hosting	0	++	0	0
Tool-Unterstützung	++	+	0	0
Benötigtes Know-How	0	++	++	++
Personalmarkt	+	++	0	0

4. Generation: Web-Sprachen

- Mit dem Umstieg von mehren HTL-Seiten zur „Single-Page-Application“ ist die Server-Sprache nicht mehr so relevant.
- Aufwand
  - Man muss die Parameter abfangen
  - Datenbank abfragen
  - Ergebnisse als JSON-String konvertieren
  - Nur die **Daten** werden transportiert

- Mit SPA ist es ausreichend mit PHP zu arbeiten.

https://t3n.de/news/single-page-webanwendung-1023623/

https://t3n.de/news/single-page-webdesign-beispiele-323225/

https://airnauts.com/

https://delhi-timeline.in/

4. Generation: Scriptsprachen

- Häufig vorhandene Merkmale sind:
  - implizit deklarierte Variablen (auch dynamische Funktionsnamen),
  - dynamische Typisierung,
  - automatische Speicherverwaltung, automatische Speicherbereinigung,
  - dynamische Klassenzugehörigkeit oder prototypenbasierte Vererbung,
  - unmittelbare Ausführung durch Interpretation des Quelltextes ohne getrennte Übersetzungsphase.
  - Bei einigen Skriptsprachen kann der Programmcode durch das Programm selbst manipuliert werden; das macht jene Sprachen besonders flexibel.
- Beispiele:
  - Bash, cmd, Lisp, PHP,
  - Pearl, Python (Julia), Ruby
  - Java-Script
    - CoffeeScript, wird nach JavaScript transcompiliert
    - TypeScript, wird nach JavaScript transcompiliert

4. Generation: Funktionale Sprachen

Eigenschaften:

- Basis ist die Funktionen.
- Funktionen werden nicht als Abfolge von Anweisungen dargestellt, sondern als ineinander verschachtelte Funktionsaufrufe.
- Funktionen sind gegenüber allen anderen Datenobjekten gleichberechtigt. Das bedeutet, dass sie als Parameter in Funktionen eingehen dürfen und ebenso als Berechnungsergebnisse aus Funktionen hervorgehen können.
- **Insbesondere können Funktionen wie andere Datenobjekte zur Laufzeit erstellt werden oder entfallen.**
- Funktionsdefinitionen können ohne explizite Namensgebung literal in der Stellung eines Funktionssymbols verwendet werden. Solche Funktionen heißen anonym und werden oft salopp „Lambdas“ genannt.

- Beispiele:
  - Lisp F# Haskell
  - Python Kotlin Swift

4. Generation: Funktionale Sprachen

Eingabe:
  Zahl n
Ausgabe:
  Zahl b (= 1 · 2 · 3 · … · n)

Algorithmus:
  b := 1 (Zuweisung)
  solange n > 0 führe aus
    b := n · b
    n := n − 1

  Ausgabe: b

Funktion f(n):
  wenn n > 0 dann:
    Ausgabe: n · f(n - 1)
  sonst wenn n = 0 dann:
    Ausgabe: 1

- map f [] = []
- map f (x:xs) = f x : map f xs

4. Generation: Funktionale Sprachen

$$\pi \cdot r_2^2 \quad - \quad \pi \cdot r_1^2$$

Common Lisp

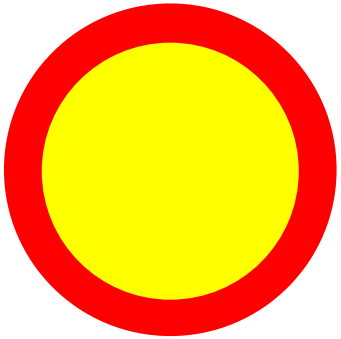
```
(defun ringarea (r1 r2)
   (flet ((sq (x)  (* x x)))
   (* pi (- (sq r1)
      (sq r2)))))
```

F# und OCaml

```
let ringarea (r1, r2) =
   let pi = 3.14 in
   let sq x = x*x in
   pi * (sq r1 - sq r2)
```

Haskell

```
sq :: (Floating a) => a -> a
        sq x = x * x

ringArea :: (Floating a) => a -> a -> a
ringArea r1 r2 = pi * (sq r1 - sq r2)
```



4. Generation: Esoterische Programmiersprachen

Eigenschaften:

- Esoterische Programmiersprachen sind Programmiersprachen, die nicht für den praktischen Einsatz entwickelt wurden, sondern um ungewöhnliche Sprachkonzepte umzusetzen.
- Eine einfache Bedienung ist selten, teilweise werden Sprachen konzipiert, um möglichst komplizierte Algorithmen oder eine unverständliche Syntax zu haben, oft aber auch um neue Ideen auszuprobieren, oder um ungewöhnliche Möglichkeiten wie extreme Vereinfachung aufzuzeigen.

- Beispiele:
  - Brainfuck – minimale imperative Programmiersprache, winziger Compiler von Urban Müller
  - Shakespeare Programming Language – Code sieht aus wie ein Stück von William Shakespeare, von Kalle Hasselström und Jon Åslund
  - Piet – Programmcode besteht aus abstrakten Bildern, von David Morgan-Mar

4. Generation: Brainfuck

a[0] = 0; a[1] = 70; a[2] = 100; a[3] = 30; a[4] = 10;

```
++++++++++
 [
  >+++++++>++++++++++>+++>+<<<<-
 ]                        Schleife zur Vorbereitung der Textausgabe
 >++.                     Ausgabe von 'H'
 >+.                      Ausgabe von 'e'
 +++++++.                 'l'
 .                        'l'
 +++.                     'o'
 >++.                     Leerzeichen
 <<+++++++++++++++.       'W'
 >.                       'o'
 +++.                     'r'
 ------.                  'l'
 --------.                'd'
 >+.                      '!'
 >.                       Zeilenvorschub
 +++.                     Wagenrücklauf
```

https://de.wikipedia.org/wiki/Brainfuck

4. Generation: Brainfuck

Brainfuck besitzt acht Befehle, jeweils bestehend aus einem einzigen Zeichen:

Zeichen	C-Aquivalent	Semantik
>	++ptr	inkrementiert den Zeiger
<	--ptr	dekrementiert den Zeiger
+	++*ptr	inkrementiert den aktuellen Zellenwert
-	--*ptr	dekrementiert den aktuellen Zellenwert
.	putchar(*ptr)	Gibt den aktuellen Zellenwert als ASCII-Zeichen auf der Standardausgabe aus
,	*ptr=getchar()	Liest ein Zeichen von der Standardeingabe und speichert dessen ASCII-Wert in der aktuellen Zelle
[	while (*ptr) {	Springt nach vorne, hinter den passenden]-Befehl, wenn der aktuelle Zellenwert 0 ist
}	}	Springt zurück, hinter den passenden [-Befehl, wenn der aktuelle Zellenwert nicht 0 ist

- Zur besseren Lesbarkeit ist dieser Brainfuckcode auf mehrere Zeilen verteilt und kommentiert worden. Brainfuck ignoriert alle Zeichen, die keine Brainfuckbefehle sind. Alle Zeichen mit Ausnahme von +-<>[],. können deswegen zur Kommentierung des Quellcodes genutzt werden.
- Zunächst wird die erste (die „nullte“) Zelle des Datenfelds auf den Wert 10 gesetzt (a[0] = 10). Die Schleife am Anfang des Programms errechnet dann mit Hilfe dieser Zelle weitere Werte für die zweite, dritte, vierte und fünfte Zelle. Für die zweite Zelle wird der Wert 70 errechnet, welcher nahe dem ASCII-Wert des Buchstaben 'H' (ASCII-Wert 72) liegt. Die dritte Zelle erhält den Wert 100, nahe dem Buchstaben 'e' (ASCII-Wert 101), die vierte den Wert 30 nahe dem Wert für Leerzeichen (ASCII-Wert 32), die fünfte den Wert 10, welches dem ASCII-Zeichen „Line Feed“ entspricht und als Zeilenumbruch interpretiert wird (oder werden sollte, siehe dazu den Abschnitt „Implementierungen“).
- Die Schleife errechnet die Werte, indem einfach auf die anfangs mit 0 initialisierten Zellen 10-mal 7, 10, 3 und 1 addiert wird. Nach jedem Schleifendurchlauf wird a[0] dabei um eins verringert, bis es den Wert 0 hat und die Schleife dadurch beendet wird.
- Am Ende der Schleife hat das Datenfeld dann folgende Werte: a[0] = 0; a[1] = 70; a[2] = 100; a[3] = 30; a[4] = 10;
- Als Nächstes wird der Zeiger auf die zweite Zelle des Datenfelds (a[1]) positioniert und der Wert der Zelle um zwei erhöht. Damit hat es den Wert 72, welches dem ASCII-Wert des Zeichens 'H' entspricht. Dieses wird daraufhin ausgegeben. Nach demselben Schema werden die weiteren auszugebenden Buchstaben mit Hilfe der durch die Schleife initialisierten Werte, sowie der bereits verwendeten Werte, errechnet.

4. Generation: Echtzeit Programmiersprachen

Eigenschaften:

- Nicht-Echtzeit-Datenverarbeitung–
  - Richtigkeit des Ergebnisses
- Echtzeit-Datenverarbeitung
  - Richtigkeit des Ergebnisses
  - **Rechtzeitigkeit des Ergebnisses**
  - **Zeitlimit**

- Anwendungen:
  - Regelungen in Flugzeugen
  - Motorsteuerung (Formel 1: 18000 Umdrehungen/min)
- Beispiele:
  - Ada 95
  - C
  - C++
  - Pearl

Programmierparadigma

- Imperative Programmierung
  - Strukturierte Programmierung
  - Prozedurale Programmierung
  - Modulare Programmierung
  - Programmierung mit abstrakten Datentypen (struct, class, interface)
- Deklarative Programmierparadigmen
  - Funktionale Programmierung
  - Logische Programmierung (OPS-5, Prolog)
  - Constraintprogrammierung
- Objektorientierte Programmierparadigmen
- Komponentenorientierte Programmierparadigmen
  - Schnittstellen
- Parallele Programmierparadigmen
  - Threads, OpenMP, MPI (Netzwerk), OpenCL, CUDA (Grafikkarte)

Open GL und Open CL bei Grafikkarten für Photoshop und Lightroom

- Photoshop und Lightroom nutzen immer mehr die GPU und machen dabei Gebrauch von OpenGL und OpenCL.
- Ab Version 3.3 von OpenGL ist OpenCL in Open GL integriert.

Quelle: (http://de.wikipedia.org/wiki/OpenGL).

https://www.pcwelt.de/produkte/Die_beste_GPU-Computing-Grafikkarte_im_Test-Beschleunigen_von_parallelen_Berechnungen-8096201.html

Platz 1: Gigabyte Radeon R9 290X OC

Algorithmen

- Ein Algorithmus gibt in Einzelschritten vor, wie man ein Problem löst.
  - **Eindeutigkeit:**
    - Ein Algorithmus darf keine widersprüchliche Beschreibung haben. Diese muss eindeutig sein.
  - **Ausführbarkeit:**
    - Jeder Einzelschritt muss ausführbar sein.
  - **Finitheit (= Endlichkeit):**
    - Die Beschreibung des Algorithmus muss endlich sein.
  - **Terminierung:**
    - Nach endlich vielen Schritten muss der Algorithmus enden und ein Ergebnis liefern.
  - **Determiniertheit:**
    - Der Algorithmus muss bei gleichen Voraussetzungen stets das gleiche Ergebnis liefern.
  - **Determinismus:**
    - Zu jedem Zeitpunkt der Ausführung besteht höchstens eine Möglichkeit der Fortsetzung. Der Folgeschritt ist also eindeutig bestimmt.

Algorithmen

- Weitere Begriffe:
  - **Rekursiv /Iteration:**
    - Iterative Algorithmen können in rekursive Algorithmen umgewandelt werden und umgekehrt.
    - rekursive Algorithmen sind natürlicher, mathematischer definiert.
  - **Komplexität :**
    - Algorithmen können unterschiedlich „schnell“ sein.
    - Je nach Algorithmus kann man 0,5 Sekunden oder **Jahre** benötigen!
      - Sortieren
      - Suchen
      - Fibonacci
  - **Hardware:**
    - Algorithmen können auch in Hardware eingesetzt.
    - Field Programmable Gate Arrays (FPGA)
      - Digital/Analog-Wandler

Algorithmenbeispiele

- Zip-Datei
- Sortierung mit Bubblesort, MergeSort
- MonteCarlo Integration
- Fibonacci: for-loop vs. Rekursion
- Games of live
- HashTable
- Türme von Hanoi: GUI
- Dijkstra
- Newton: Bestimmen der Nullstellen von $x^4=1$
- PDF Erstellung
- Datenbank Indexdatei, Aufbau, Benutzung, Vorteile, Nachteile
- Thread
- Backgroundworker

Zip-Datei: Kenngrößen

Statistische Informationsgehalt

$$\mathrm{I(x)} = \log_b \frac{1}{\mathrm{w(x)}}$$

Die Entropie einer Nachricht

$$\mathrm{H} = \sum_i^n w(x_i) \cdot ld\left(\frac{1}{w(x_i)}\right) = \sum_i^n w(x_i) \cdot I(x_i)$$

Die mittlere Wortlänge

$$\mathrm{L} = \sum_i^n w(i) \cdot l(i)$$

Zip-Datei: Codierung

- Alphabet besteht aus A, B, C
- Codierung:
  - A: 10
  - B: 01
  - C: 0
- Die Folge „ABC“ ergibt dann „10010“
- Die Folge „ACA“ ergibt dann „10010“
- $\Rightarrow$ Nicht eindeutig

Um diese Mehrdeutigkeit zu verhindern, müssen die den Zeichen zugewiesenen Bitfolgen die Eigenschaft besitzen, *präfixfrei* zu sein, d.h. keine Bitfolge, die für ein einzelnes Zeichen steht, darf am Anfang eines anderen Zeichens stehen.

Zip-Datei: Code-Erzeugung

Ein Code mit variabler Länge erzeugt man am einfachsten nach der Berechnung des Informationsgehaltes:

x_i	$w(x_i)$	$I(x_i)$	$Len(x_i)$	Beispielcode
c	0,1643	2,606	3	001
v	0,0455	4,458	5	10111
w	0,0874	3,516	4	0001
u	0,1963	2,349	3	011
r	0,4191	1,255	2	11
z	0,0874	3,516	4	0000

$$H = \sum_{i}^{n} w(x_i) \cdot I(x_i)$$

$$L = \sum_{i}^{n} w(i) \cdot l(i)$$

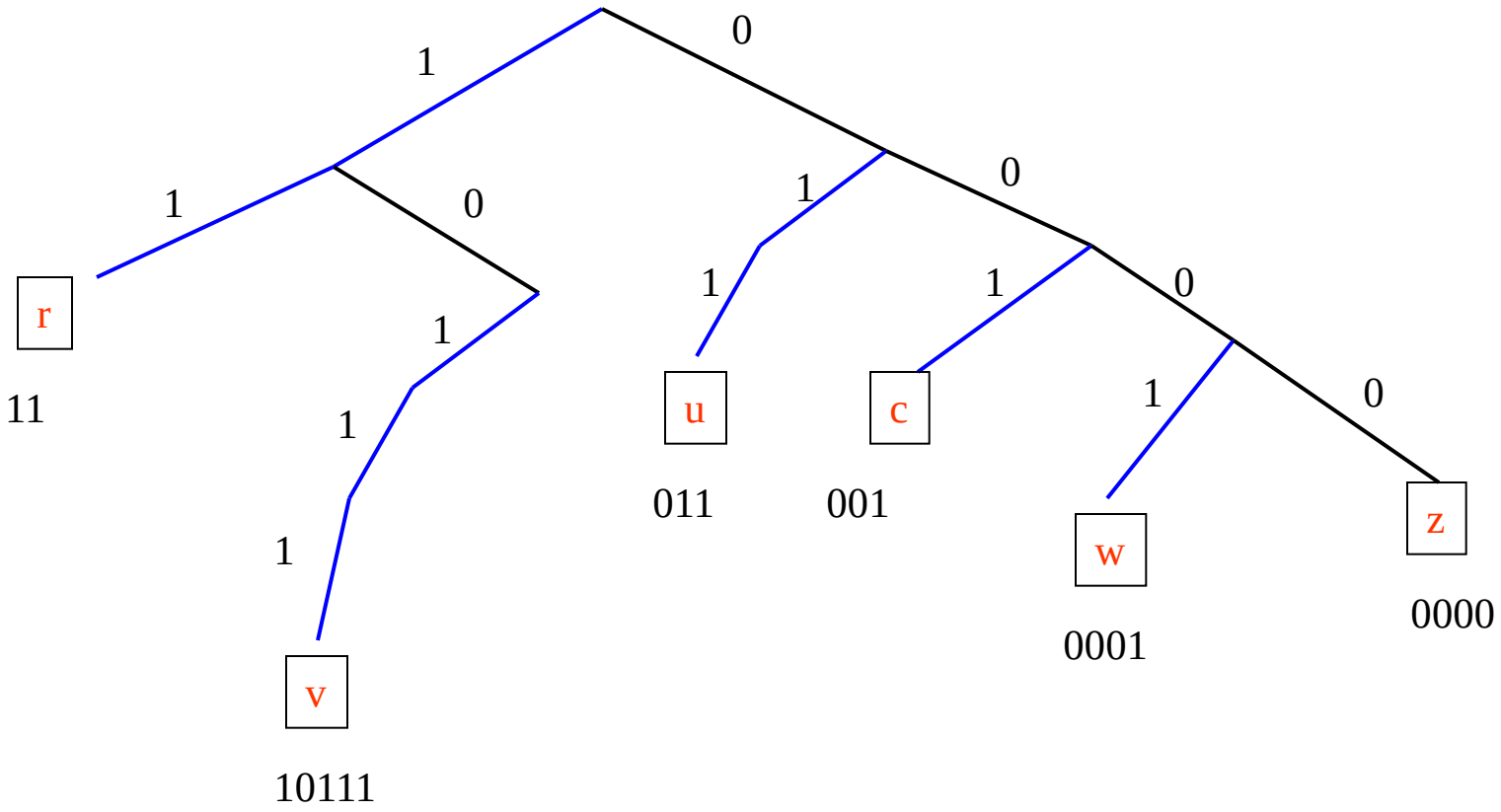
Entropie ? **H = 2,237 Bit/Zeichen**

Redundanz ? **L = 2,8467 Bit/Zeichen**

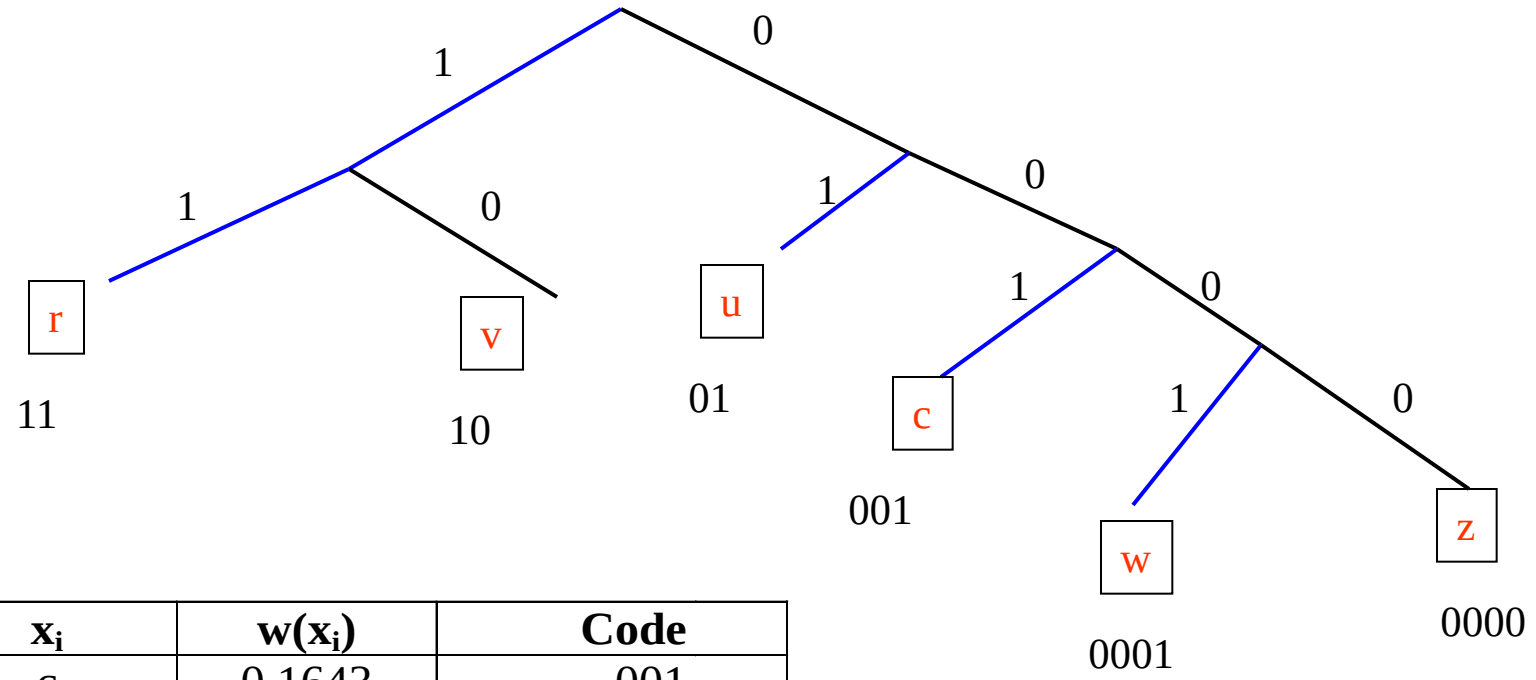
R = 0,6140 Bit/Zeichen

R = L – H [Bit / Zeichen]

Zip-Datei: Code-Erzeugung: Baumdarstellung

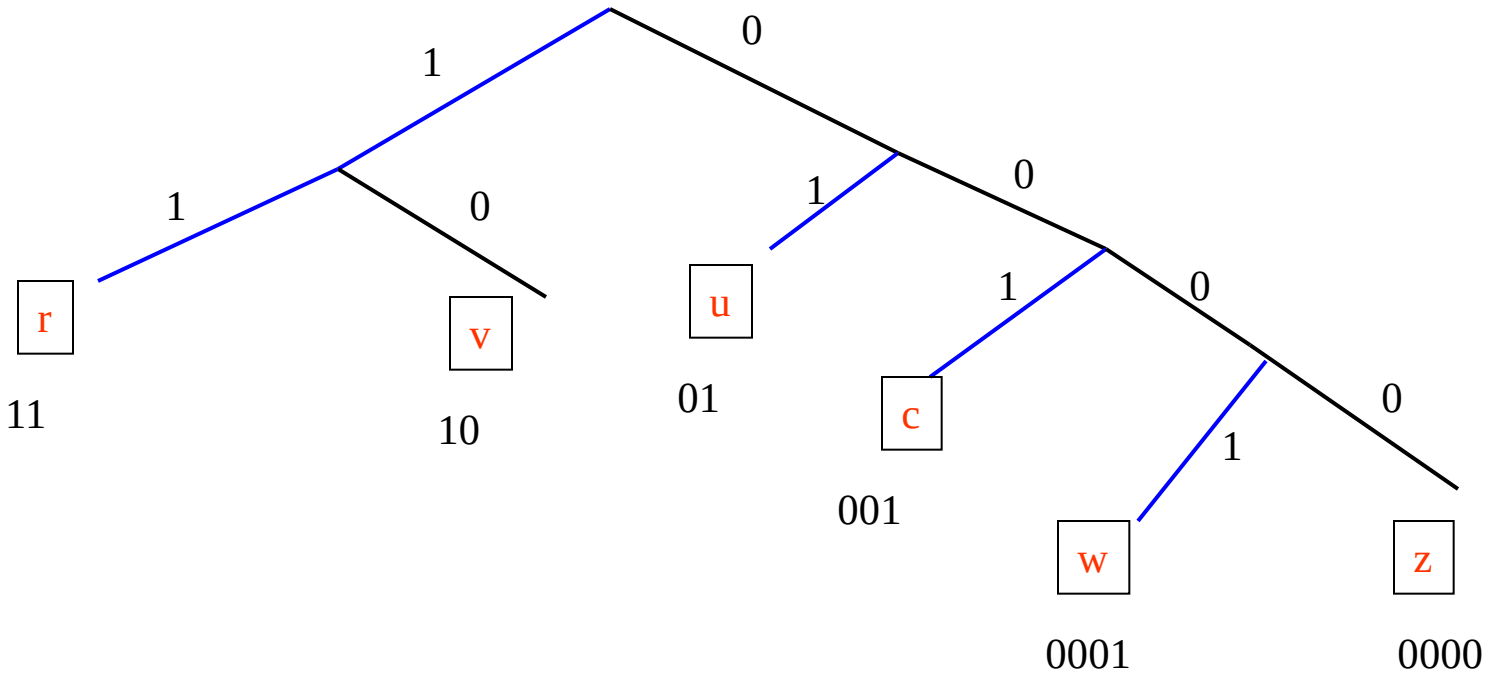


Zip-Datei: Code-Erzeugung: Beispiel1



x_i	$w(x_i)$	Code
c	0,1643	001
v	0,0455	10111
w	0,0874	0001
u	0,1963	011
r	0,4191	11
z	0,0874	0000

Zip-Datei: Code-Erzeugung: Beispiel2



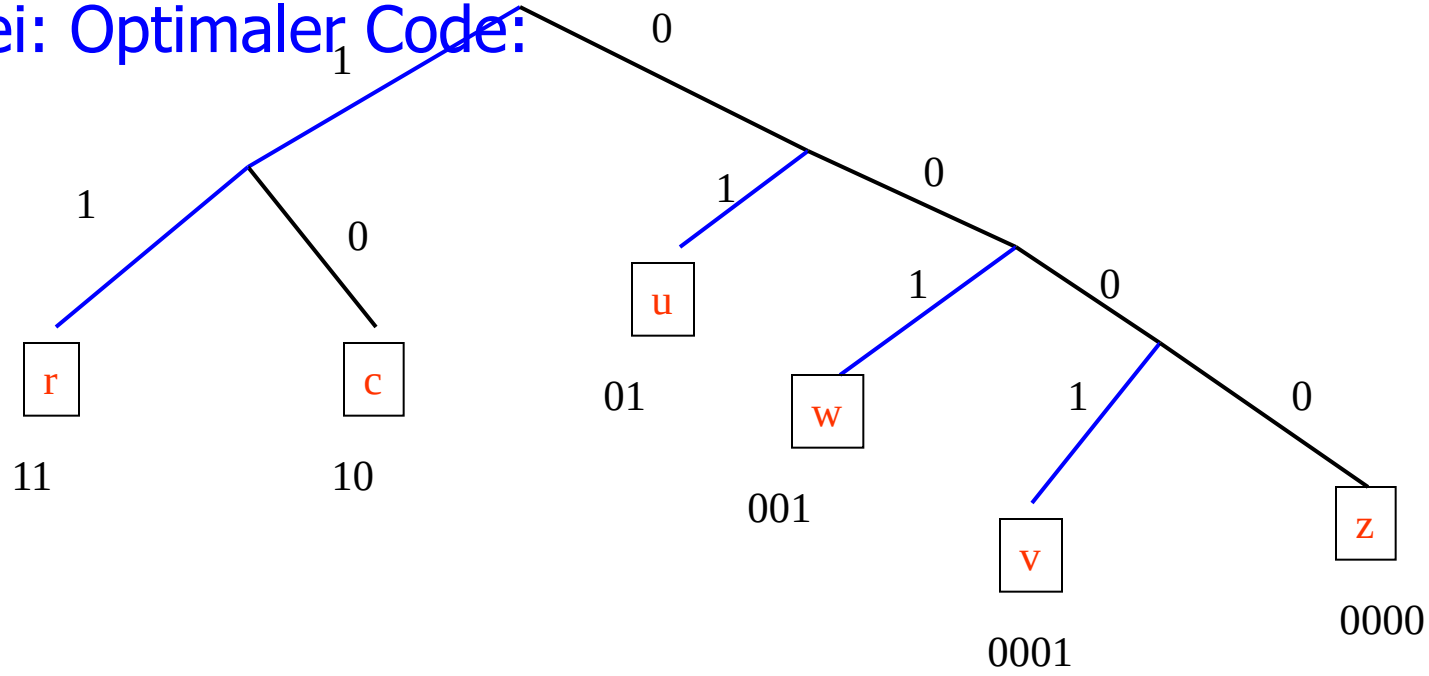
Zip-Datei: Code-Optimierung:

Der vorliegende Code ist nicht optimal, da es noch unbesetzte Endknoten - Blätter - gibt, die unbesetzt sind.

Optimierung:

- Freie Blätter werden mit Zeichen besetzt, deren Wortlänge größer ist als die zu dem freien Blatt gehörende Wortlänge.
- Der Code eines Zeichens mit geringerer Auftrittswahrscheinlichkeit darf nicht kürzer sein als der Code eines Zeichens mit höherer Auftrittswahrscheinlichkeit.

Zip-Datei: Optimaler Code:



x_i	$w(x_i)$	$I(x_i)$	$Len(x_i)$	Code
c	0,1643	2,606	2	10
v	0,0455	4,458	4	0001
w	0,0874	3,516	3	001
u	0,1963	2,349	2	01
r	0,4191	1,255	2	11
z	0,0874	3,516	4	0000

Zip-Datei: Optimaler Code:

Mittlere Wortlänge:

$L = (0{,}4191+0{,}1963+0{,}1643)\cdot 2 + 0{,}0874\cdot 3 + (0{,}0874+0{,}0455)\cdot 4$

$L = 2{,}3532$ [Bits / Zeichen]

Neue Redundanz:

$R = 2{,}3532 - 2{,}2327 = 0{,}1205$ [Bits / Zeichen]

Alte Redundanz:

$R = 0{,}6140$ [Bits / Zeichen]

Zip-Datei: Code-Erzeugung

Bestimmung des optimalen Codes:

Kriterium: Minimale Redundanz

Beispiel:

x_i	$w(x_i)$	$I(x_i)$	Len	Code	H = w*I	L = w*l
c	0,16	2,643	3	001	0,423	0,480
v	0,04	4,643	5	10111	0,186	0,20
w	0,09	3,474	4	0001	0,313	0,360
u	0,19	2,396	3	011	0,455	0,570
r	0,42	1,252	2	11	0,526	0,840
z	0,10	3,322	4	0000	0,332	0,400

H = 2,235

L = 2,850

R = L - H = 0,615

Zip-Datei: Code-Erzeugung

2. Variante;

x_i	$w(x_i)$	$I(x_i)$	Len	Code	H = w*I	L = w*l
c	0,16	2,643	1	1	0,423	0,16
v	0,04	4,643	2	10	0,186	0,08
w	0,09	3,474	4	1010	0,313	0,36
u	0,19	2,396	3	111	0,455	0,570
r	0,42	1,252	3	110	0,526	1,26
z	0,10	3,322	1	0	0,332	0,1

H = 2,232

L = 2,530

R = L - H = 0,398

Zip-Datei: Huffman-Code-Verfahren:

- Sortiere alle Zeichen nach der Größe ihrer Auftrittswahrscheinlichkeit (oben das seltenste)
- Den beiden seltensten wird als letztes Bit 0 bzw. 1 zugeordnet
- Danach werden sie unter Addition ihrer Einzelhäufigkeit zu einer Klasse zusammengefasst. Diese wird nach aufsteigender Häufigkeit eingeordnet.
- Das Verfahren wird wiederholt.
- Die Codierung ergibt sich in umgekehrter Reihenfolge

Zip-Datei: Huffman-Code-Verfahren:

x_i	$w(x_i)$ [%]	Code
a	16	
b	20	
c	40	
d	8	
e	4	
f	10	
g	2	

a	**110**
b	**111**
c	**0**
d	**1011**
e	**10101**
f	**100**
g	**10100**

Zip-Datei: Huffman-Code-Verfahren:

x_i	$w(x_i)$	Code
a	16	
b	20	
c	40	
d	8	
e	4	
f	10	
g	2	

x_i	$w(x_i)$	Code
g	2	0
e	4	1
d	8	
f	10	
a	16	
b	20	
c	40	

→ **A**

x_i	$w(x_i)$	Code
A	6	0
d	8	1
f	10	
a	16	
b	20	
c	40	

→ **B**

x_i	$w(x_i)$	Code
f	10	0
B	14	1
a	16	
b	20	
c	40	

→ **C**

Zip-Datei: Huffman-Code-Verfahren:

x_i	$w(x_i)$	Code
a	16	0
b	20	1
C	24	
c	40	

→ **D**

x_i	$w(x_i)$	Code
C	24	0
D	36	1
c	40	

→ **E**

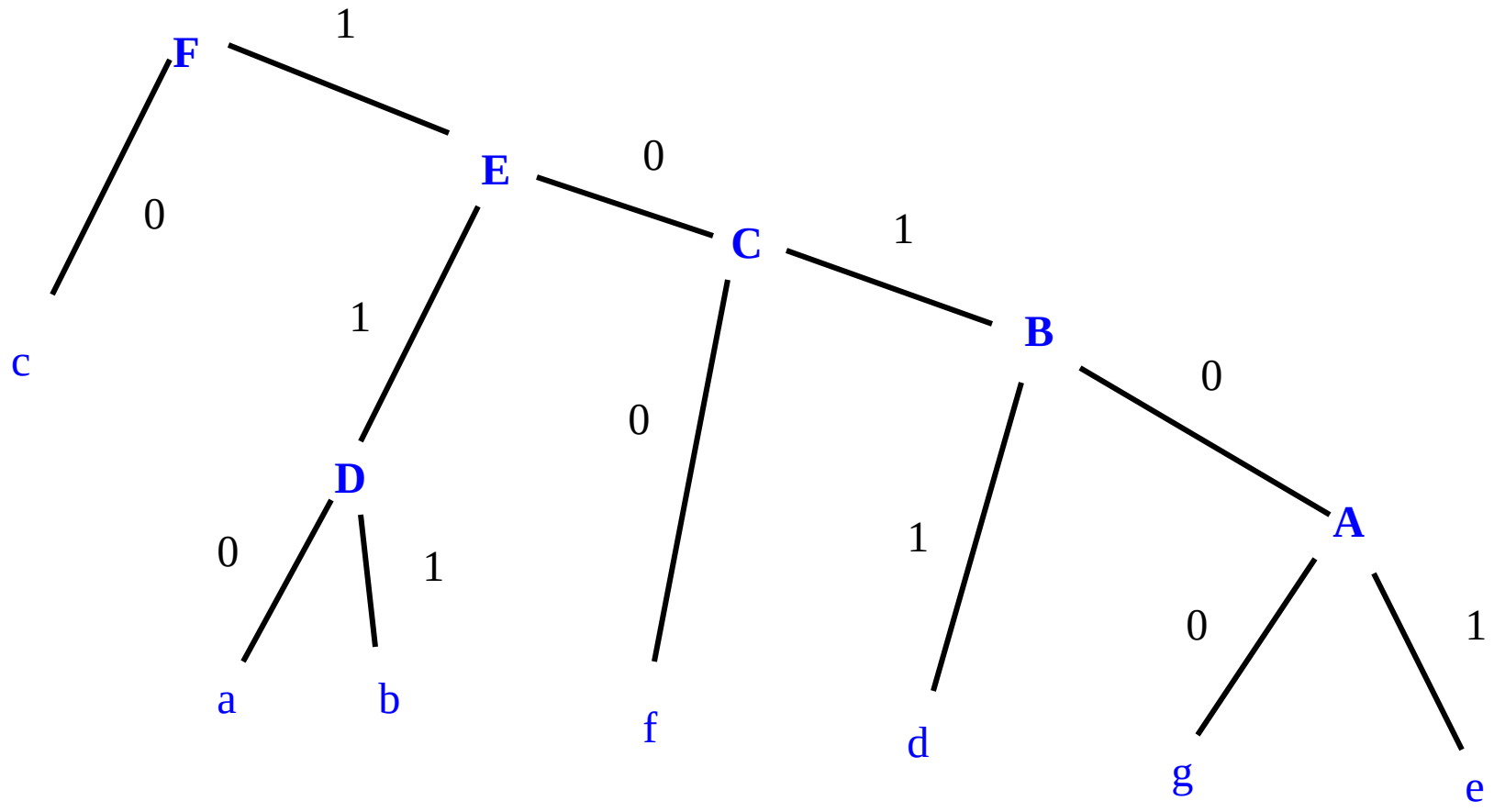
x_i	$w(x_i)$	Code
c	40	0
E	60	1

→ **F**

a	**110**
b	**111**
c	**0**
d	**1011**
e	**10101**
f	**100**
g	**10100**

Von a bis F gehen, Aufsammeln, dann drehen!

Zip-Datei: Huffman-Code-Verfahren:



Zip-Datei: Huffman-Code-Verfahren:

- Minimale Redundanz
- asymmetrische Gestalt des Code-Baumes
- Nicht immer in eindeutiger Weise decodierbar
- Bei Code mit konstanter Wortlänge ist die Decodierbarkeit durch das Abzählen gegeben.
  - ASCII-Code, Ansi-Code
- Bei Code mit variabler Wortlänge ist die Decodierbarkeit nicht immer gegeben.
  - Morsealphabet

Zip-Datei: 1. Beispiel:

x_i	$w(x_i)$	$I(x_i)$	**Len**	**Code**	H = w*I	L = w*l
v	0,012	6,381				
w	0,410	1,286				
x	0,218	2,198				
y	0,181	2,467				
z	0,179	2,482				

Zip-Datei: 2. Beispiel Huffmann-Code

x_i	$w(x_i)$	$I(x_i)$	Len	Code	H = w*I	L = w*l
v	0,51					
w	0,20					
x	0,20					
y	0,08					
z	0,01					

1001000100011

Zip-Datei: 3. Beispiel Huffmann-Code

x_i	$w(x_i)$	$I(x_i)$	Len	Code	H = w*I	L = w*l
v	0,20					
w	0,20					
x	0,20					
y	0,20					
z	0,20					

1001000100011

Zip-Datei: Lösung des 1. Beispiel Huffmann-Code

x_i	$w(x_i)$	$I(x_i)$	Len	Code	H = w*I	L = w*l
v	0,012	6,381	4	1110	0,0766	0,048
w	0,410	1,286	1	0	0,5273	0,410
x	0,218	2,198	2	10	0,4792	0,436
y	0,181	2,467	3	110	0,4465	0,543
z	0,179	2,482	4	1111	0,4443	0,716
					1,9739	2,153

H = 1,9739
L = 2,153
R = L - H = 0,1791

Sortierung

- Sortieren von Feldern und Listen ist Informatiker-Alltag.
- Es gibt viele Verfahren:
  - BubbleSort
  - Mergesort
  - Insertionsort
  - Shellsort
  - Selectionsort
  - Quicksort
  - Heapsort
  - Countingsort
  - Slowsort
  - Binary Tree Sort
- Kriterien
  - Sind die Daten komplett im Hauptspeicher
  - Stabil: Bleibt die vorherige Sortierung erhalten
  - Laufzeit, Landau-Symbol
  - sort (int[] a)
  - sort (Object[] a) int compare(…)

Sortierverfahren	Best-Case	Average-Case	Worst-Case	Stabil	Zusätzlicher Speicherbedarf (sofern nicht in-place)
Binary Tree Sort (höhen-balanciert)	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	ja[1]	$\mathcal{O}(n)$
Binary Tree Sort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$	ja[1]	$\mathcal{O}(n)$
Bubblesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	ja	–
Combsort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$	nein	–
Gnomesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	ja	–
Heapsort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	nein	–
Insertionsort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	ja	–
Introsort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	nein	$\mathcal{O}(\log(n))$
Merge Insertion	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	ja	implementierungsabhängig keine, $\mathcal{O}(n)$ oder $\mathcal{O}(n \cdot \log(n))$
Mergesort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	ja	bei Arrays: $\mathcal{O}(n)$, je nach Implementierung auch $\mathcal{O}(n \cdot \log(n))$; bei verketteten Listen: $\mathcal{O}(\log(n))$
Natural Mergesort	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	ja	bei Arrays: $\mathcal{O}(n)$, je nach Implementierung auch $\mathcal{O}(n \cdot \log(n))$; bei verketteten Listen: $\mathcal{O}(\log(n))$
Quicksort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$	nein	$\mathcal{O}(\log(n))$ (je nach Implementierung immer oder nur im average case)
Selectionsort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	nein	–
Shakersort (Cocktailsort)	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	ja	–
Shellsort		$\mathcal{O}(n \cdot \log(n)^2)$	$\mathcal{O}(n^{1,25})$	nein	–
Smoothsort	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	nein	$\mathcal{O}(n \cdot \log(n))$ für den Stack
Stoogesort	$\mathcal{O}(n^{2,71})$	$\mathcal{O}(n^{2,71})$	$\mathcal{O}(n^{2,71})$	nein	$\mathcal{O}(\log(n))$
Swap-Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	–	–
Timsort	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	ja	implementierungsabhängig $\mathcal{O}(n)$ oder $\mathcal{O}(\log(n))$
Bogosort	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot n!)$	∞	nein	–
Slowsort	$\mathcal{O}\left(n^{\frac{\log(n)}{(2+\epsilon)}}\right)$	$\mathcal{O}\left(n^{\frac{\log(n)}{2}}\right)$	$\mathcal{O}\left(n^{\frac{\log(n)}{2}}\right)$	nein	–

https://github.com/MartinThoma/algorithms/blob/master/sorting.py

Sortierung: Bubblesort in Java (imperativ)

```
public static void bubblesort (int sortieren []) {
  for (int i = 0; i < sortieren.length - 1; i++) {
    for (int j = 0;  j < sortieren.length-i; j++) {
      if (sortieren[j] > sortieren[j+1]) {
        int temp = sortieren[j];
        sortieren[j] = sortieren[j+1];
        sortieren[j+1] = temp;
      }  // if
    }  // for
  } // for
}
```

Sortierung: Selectsort in Java (imperativ)

- Die Funktionsweise von Selectionsort stellt sich so dar:
- Man hat ein Array in dem sich die zu sortierenden Daten befinden. Nun teilt man das Array im Geiste in zwei Teile. Der erste Teil soll den sortierten Teil darstellen, der zweite Teil den unsortierten. Da am Anfang noch nichts sortiert ist, stellt dementsprechend das gesamte Array den unsortierten Teil dar.

- Nun sucht man das kleinste Element im Array und vertauscht es mit dem ersten Element. Nun befindet sich also das erste Element im sortierten Bereich und die restlichen Elemente im unsortierten. Nun wird wieder im unsortierten Teil nach dem kleinsten Element gesucht und dieses nun hinter dem ersten Element im sortierten Bereich eingefügt. Dies geht so lange, bis alle Elemente im sortierten Bereich und keins mehr im unsortierten Bereich sich befindet. Wer nach der Größe sortieren will, angefangen mit dem größten Element, sucht statt dem kleinsten, immer nach dem größten Element.

Sortierung: Selectsort in Java (imperativ)

```
public static void selectsort (int sortieren []) {
  for (int i = 0; i < sortieren.length - 1; i++) {
    for (int j = i + 1; j < sortieren.length; j++) {
      if (sortieren[i] > sortieren[j]) {
        int temp = sortieren[i];
        sortieren[i] = sortieren[j];
        sortieren[j] = temp;
      }  // if
    }  // for
  } // for
}
```

Sortierung: Selectsort in Java (imperativ)

```
public static void selectsort (double sortieren []) {
  for (int i = 0; i < sortieren.length - 1; i++) {
    for (int j = i + 1; j < sortieren.length; j++) {
      if (sortieren[i] > sortieren[j]) {
        int temp = sortieren[i];
        sortieren[i] = sortieren[j];
        sortieren[j] = temp;
      }  // if
    }  // for
  } // for
}
```

Suchen

- Sortieren von Feldern und Listen ist Informatiker-Alltag.
- Es gibt viele Verfahren:
  - BubbleSort
  - Mergesort
  - Insertionsort
  - Shellsort
  - Selectionsort
  - Quicksort
  - Heapsort
  - Countingsort
  - Slowsort
  - Binary Tree Sort
- Kriterien
  - Sind die Daten komplett im Hauptspeicher
  - Stabil: Bleibt die vorherige Sortierung erhalten
  - Laufzeit, Landau-Symbol
  - sort (int[] a)
  - sort (Object[] a) int compare(…)

Lineare Suche

- schnellstmöglicher Algorithmus,
  - falls an die zu durchsuchende Menge keine weiteren Bedingungen geknüpft sind.
  - falls die zu durchsuchende Menge jedoch geordnet ist, kann ein besserer Algorithmus gefunden werden.

- Um die Elemente einer Menge M anordnen zu können, müssen die Elemente vergleichbar sein, d. h. es ist möglich für a, b ∈ M zu entscheiden, ob gilt:

- a < b oder a == b oder a > b

bessere Suchmethode: Suche 450

Voraussetzung: die zu durchsuchende Menge ist sortiert.

2
4
7
8
22
45
47
99
120
450
550

bessere Suchmethode: Suche 450

Voraussetzung: die zu durchsuchende Menge ist sortiert.

2
4
7
8
22
45
47
99
120
450
550

mittlere Element, 45<450

mittlere Element, 120<450

gefunden

bessere Suchmethode

Feld der Größe n mit Ganzzahlwerten:

Wie viele Vergleiche sind nötig bis ein Element x gefunden wird?

<table>
<tr><td>1</td><td colspan="8">x < 13?</td></tr>
<tr><td>2</td><td colspan="4">x < 4?</td><td colspan="4">x < 31?</td></tr>
<tr><td>3</td><td colspan="2">x < -1?</td><td colspan="2">x < 11?</td><td colspan="2">x < 27?</td><td colspan="2">x < 92?</td></tr>
<tr><td>Wert</td><td>-4</td><td>-1</td><td>4</td><td>11</td><td>13</td><td>27</td><td>31</td><td>92</td></tr>
<tr><td>Index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
</table>

Binäre Suchmethode

Verbale Beschreibung der Binärsuche:

- Suche in einem sortierten Datenbestand den Wert der in der Mitte liegt und vergleiche ihn mit dem gesuchten Wert.
- Ist der gesuchte Wert kleiner, so wiederhole den Vorgang nur mit der linken Hälfte,
- ist der gesuchte Wert größer, so wiederhole den Vorgang nur mit der rechten Hälfte.
- Wiederhole den Vorgang, bis das gesuchte Element gefunden wurde oder klar ist, dass es nicht im Datenbestand vorhanden ist.

Leonardo von Pisa

- **Zahlenfolge:**

  1 1 2 3 5 8

- Welches ist die nächste Zahl?

Lösung mittels Schleife:

```
a=1
b=1
syso(a);
syso(b);
n=10-2;
for (int i=0; i<n; i++) {
  c=a+b;
  b=c;
  a=b;
  syso(c);
}
```

Leonardo von Pisa: Fibonacci-Folge

- **Zahlenfolge:**
  - 1 1 2 3 5 8
- Welches ist die nächste Zahl?

Lösung mittels Schleife:

```
a=1
b=1
syso(a);
syso(b);
n=10-2;
for (int i=0; i<n; i++) {
  c=a+b;
  a=b;
  b=c;
  syso(c);
}
```

Leonardo von Pisa: Fibonacci-Folge

Fibonacci-Folge: $f(n) = f(n-1) + f(n-2)$ falls $n > 1$,

$f(1) = 1$,

$f(0) = 0$

```
long calcFibonacci(int n) {
    long result = 0;
    if(n > 1) {   // Verhindert eine unendliche Rekursion
      result = calcFibonacci(n-1)+calcFibonacci(n-2);
    } else if(n == 1) {
      result = 1;
    }
    return result;
}
```

Weshalb ist diese rekursive Implementierung keine gute Idee?

Rekursive Implementierung der Fibonacci-Folge

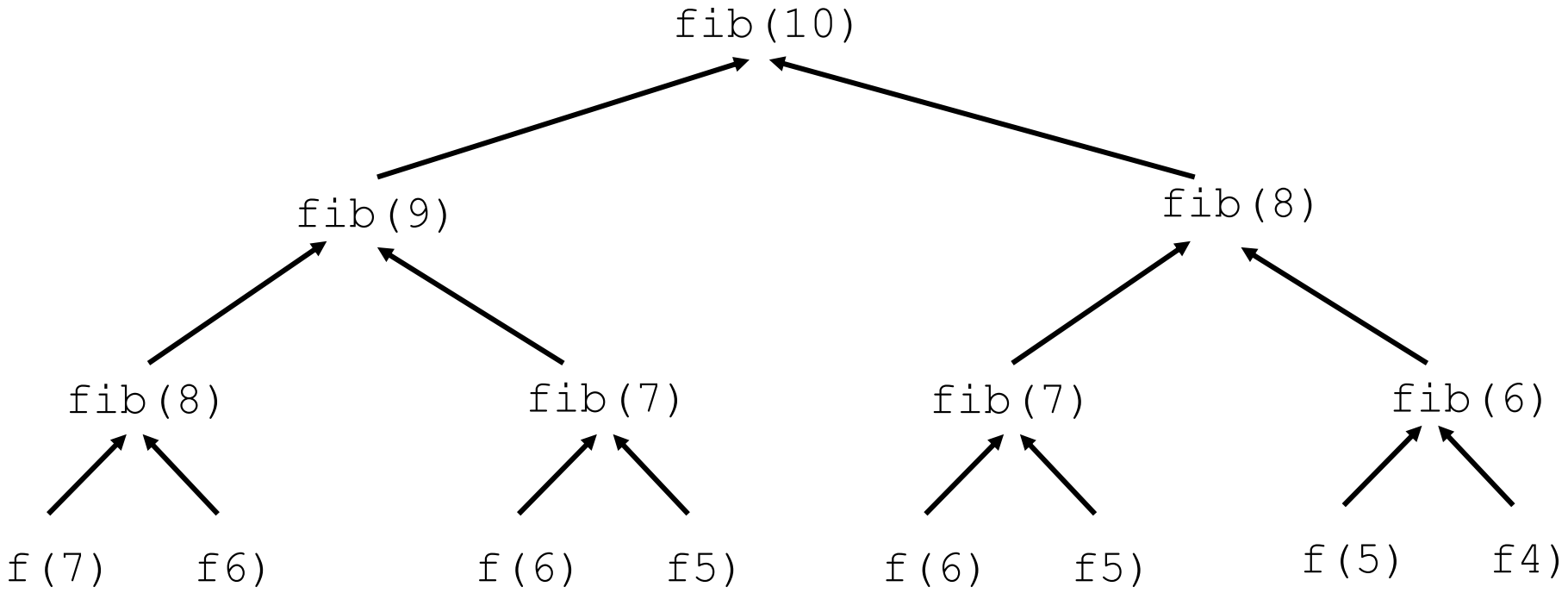
- Fibonacci-Funktion wird 2 mal rekursiv aufgerufen.
- Insgesamt für das Argument n also 2n+1 mal.
- Dabei werden dieselben Werte mehrfach berechnet.

Beispielrechner:

- Taktfrequenz 2,8 GHz
- CPU: Intel i7 2640M, 6 MB Cache

n	Java Zeit (iterativ)	Java Zeit	C#	Delphi	C / C++
20	1000 ns	0 ms	0 ms	0 ms	0 ms
30	1000 ns	16 ms	0 ms	16 ms	7 ms
35	1000 ns	46 ms	0,187 s	0,109 s	0,080 s
40	1000 ns	0,6 sec	1,8 sec	0,952 sec	1,02 s
45	1000 ns	6,7 sec	29,1 sec	12,2 sec	14,2 s
50	1000 ns	1:47 min	5:40 min	2:54 min	2:40 min
55	1000 ns	21 min	01:05:52	34:13 min	
60	1000 ns	4 Std			
70	1200 ns	Jahre			
90	1500 ns	Jahre			

Leonardo von Pisa: Fibonacci-Folge



Game of live von Cornway

Übersicht:

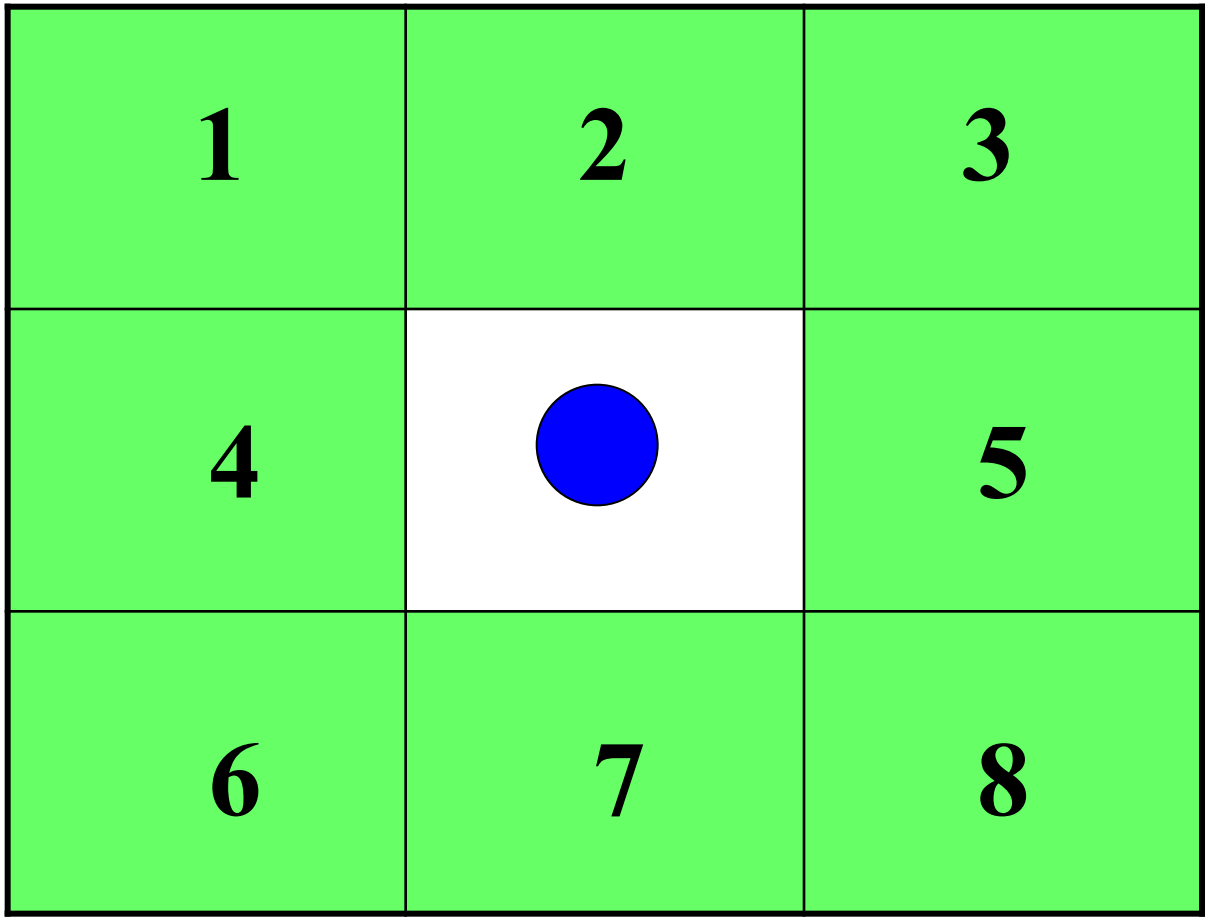
- „Games of Life“ ist eine Simulation.
- Entwickelt vom Mathematiker John Horton Conway 1970.
- Das Spielfeld besteht aus beliebig vielen Feldern (Zeilen/Spalten).
- Jede Zelle kann genau ein Lebewesen enthalten (leer/besetzt).
- Beim Start gibt eine Anfangsbelegung.
- Bei jedem Schritt werden für alle Zellen gemeinsam die neue Belegung ausgerechnet.
- Es ist also ein zellulärer Automat.

Game of live von Cornway

Die Regeln sind:

- Eine tote, leere, Zelle mit genau drei lebenden Nachbarn wird in der Folgegeneration neu geboren.
- Eine lebende Zellen kann nur weiterleben, wenn sie zwei oder drei lebende Nachbarn hat. Bei einem Nachbarn stirb sie an Einsamkeit. Bei mehr als drei Nachbarn stirbt sie an Überbevölkerung.
- Gezählt werden vertikale, horizontale und diagonale Richtungen.
- Eine Zelle am linken Rand hat nicht den Nachbarn ganz rechts.

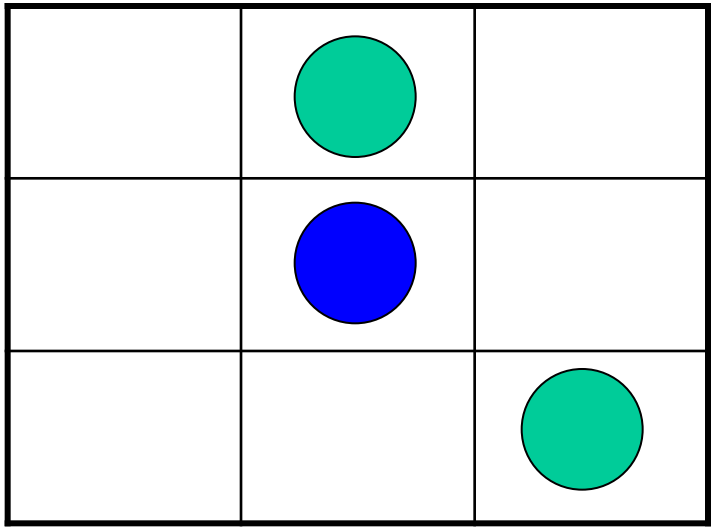
Game of live von Cornway: Anzeige der Nachbarn:



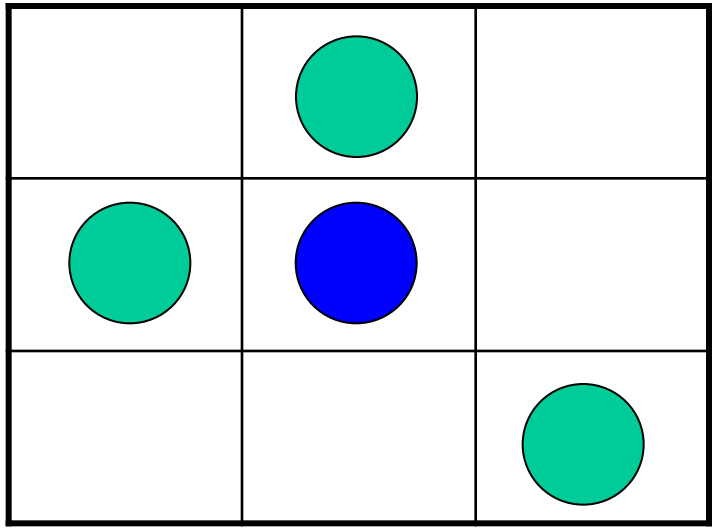
Game of live von Cornway

Beispiel für überlebende Zellen: (2→3)

Anzahl der Nachbarn = 2



Anzahl der Nachbarn = 3

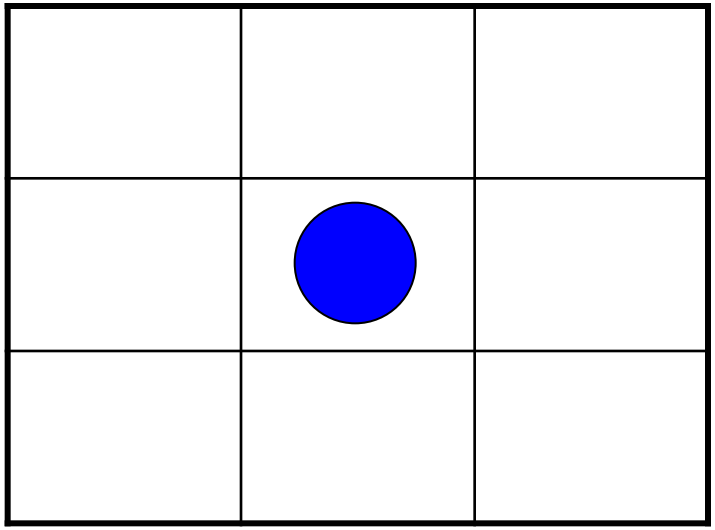


Blaue Zelle bleibt im nächsten Schritt erhalten

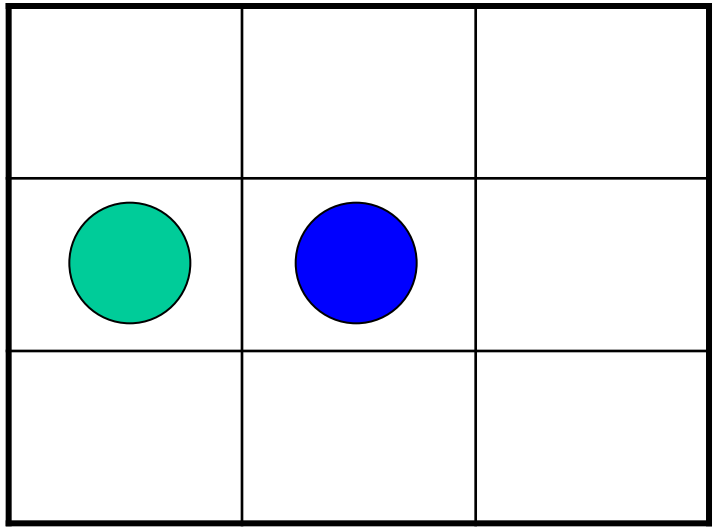
Game of live von Cornway: Anzeige der Nachbarn:

Beispiel für tote Zellen: aus Vereinsamung (0→1)

Anzahl der Nachbarn = 0



Anzahl der Nachbarn = 1

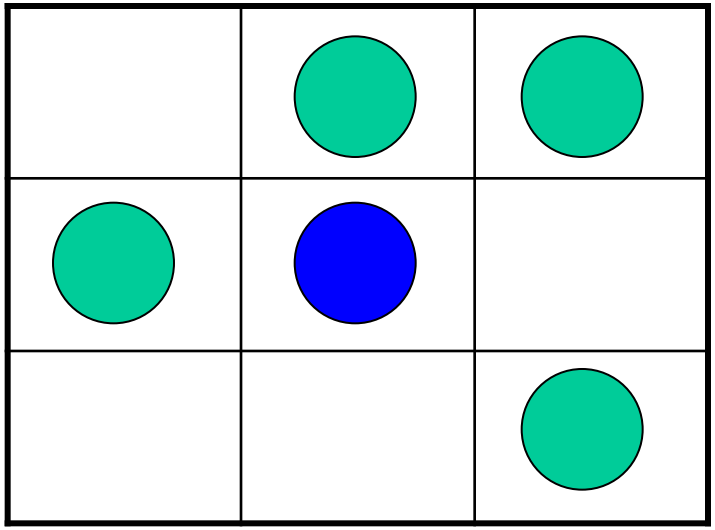


Blaue Zelle verschwindet im nächsten Schritt

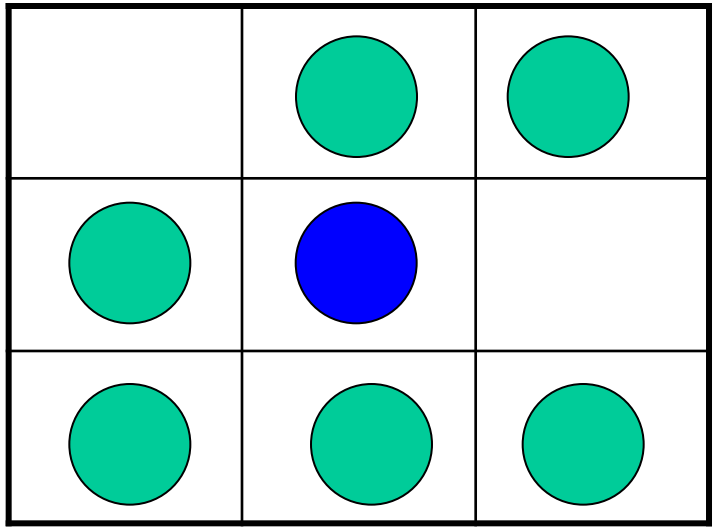
Game of live von Cornway: Anzeige der Nachbarn:

Beispiel für tote Zellen: aus Geschwätzigkeit (4→8)

Anzahl der Nachbarn = 4



Anzahl der Nachbarn = 6

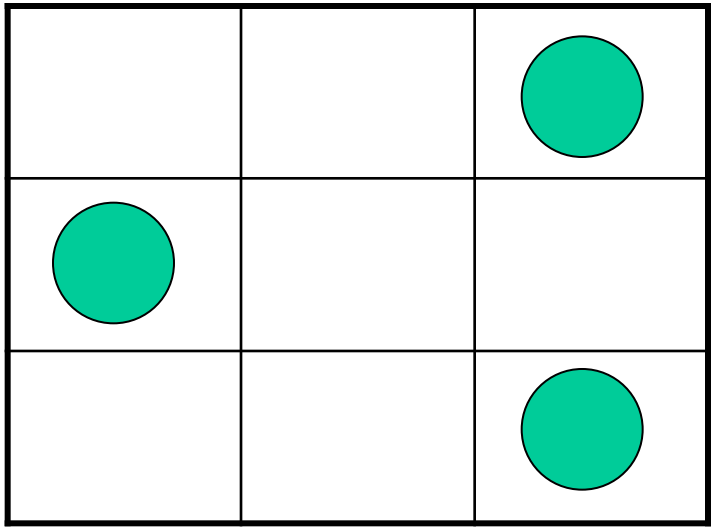


Blaue Zelle verschwindet im nächsten Schritt

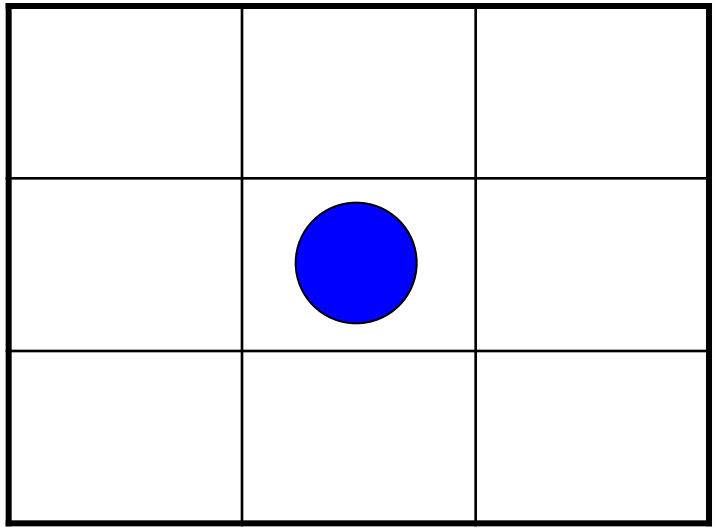
Game of live von Cornway: Anzeige der Nachbarn:

Beispiel Geburt einer neuen Zelle: (3)

Anzahl der Nachbarn = 3

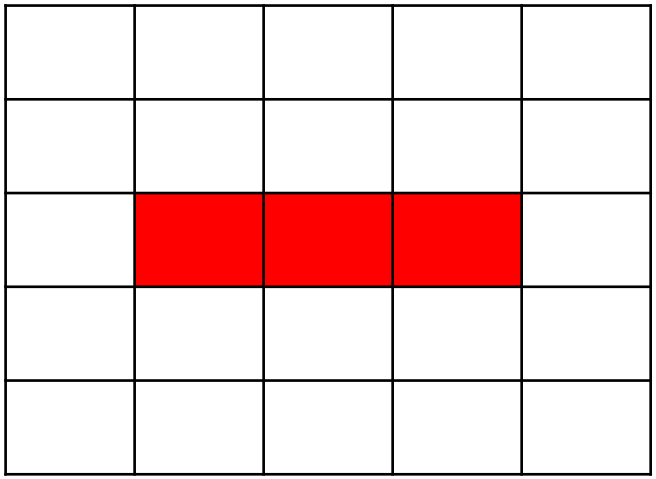


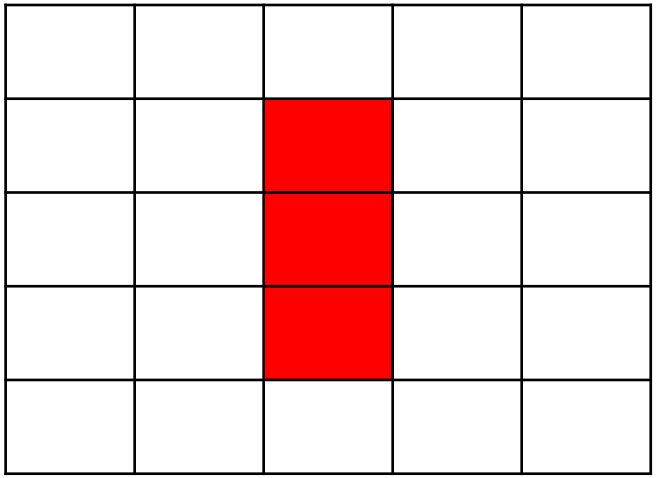
Neue Zelle im nächsten Schritt



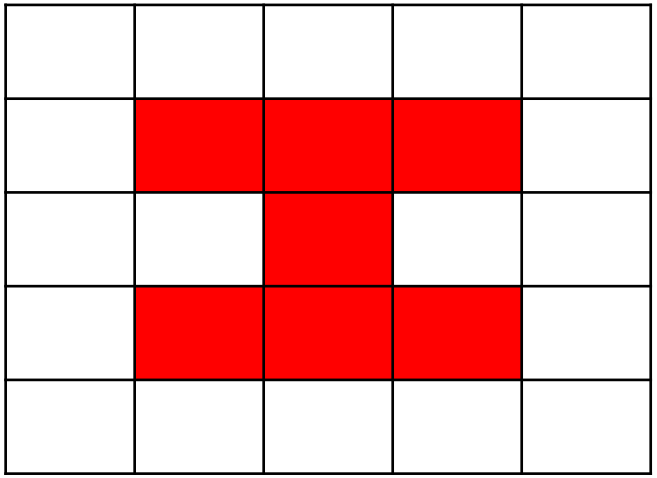
Die blaue Zelle wird neu geboren

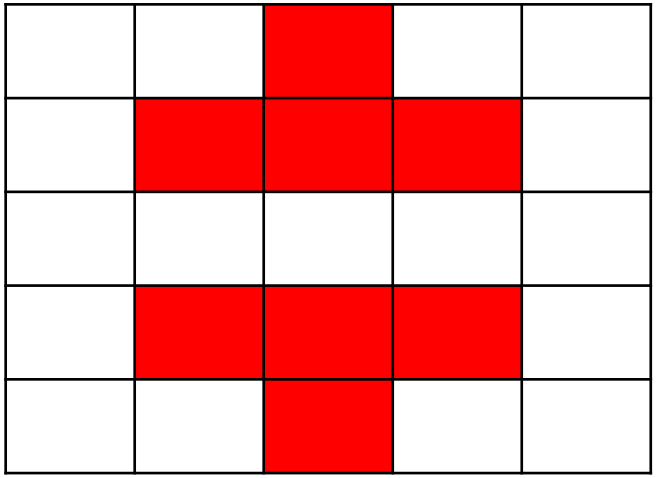
Game of live von Cornway: Beispiele



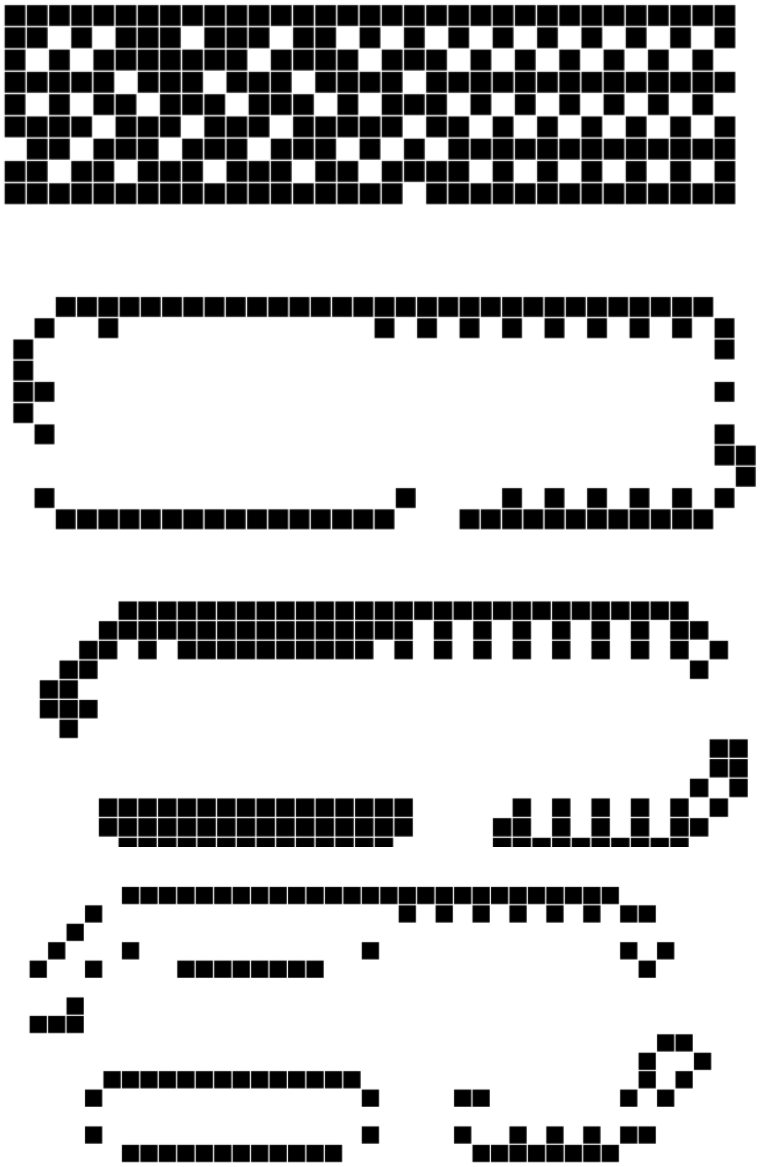


Game of live von Cornway: Beispiele





Game of live von Cornway: Beispiele



HashTable

Vorhanden

- Array mit 1 Million Datensätze
- Unsortiert
- **Aufgabe:**
  - Suchen eines Elementes
  - Lösung: Maximal **ALLE** Elemente durchsuchen
- Bessere Lösung:
  - Hashtable

HashTable

Definition Hashing:

- Die Objekte werden in einem Feld mit Indizies 0 bis N-1 gespeichert.
- Die einzelnen Speicherpositionen werden Buckets genannt.
- Eine Hashfunktion h : e --> N bestimmt für ein Schlüsselelement e einen Bucket h(e) im Feld, in dem ein mit dem Schlüssel e assoziertes Element gespeichert werden soll.
- Die Wahl der Funktion h hat entscheidenden Einfluss auf die Qualität des Verfahrens.

Kollision:

- Eine Kollision tritt auf, wenn für $e1 \neq e2$ gilt $h(e1) = h(e2)$

Beispiel:

Wir wählen die ganzen Zahlen als Element e. Ein Feld der Größe N. Die Hashfunktion sei h(e) = e mod N. Gespeichert werden sollen 42 und 119. Hier ist N = 10.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eintrag			42							119

Problem:

- Es soll noch der Wert 69 gespeichert werden. Jedoch ist 69 mod 10 = 9. Dieser Bucket ist bereits belegt. Wie soll bei einer Kollision verfahren werden?

Beachte:

- Man kann zeigen: Idealerweise ist die Größe der Hashtabelle eine Primzahl, die aber nicht nahe bei einer Zweierpotenz liegen sollte.

Hashfunktionen

- Hashfunktionen sollen “gute” Eigenschaften besitzen.
- Insbesondere sollen sie die zur Verfügung stehenden Buckets möglichst gleichmäßig füllen.
- Sie sollen aber auch einfach berechenbar sein, denn die Kosten ein Element zu speichern sind direkt proportional zu den Kosten der Berechnung der Hashfunktion.
- Der Zugriff auf Elemente einer Hashtabelle ist O(1).

Hashfunktionen

Umgang mit Kollisionen:

- **Verkettung:**
  - Alle kollidierenden Elemente werden im selben Bucket gespeichert, etwa in einer ArrayList.

- **Sondieren**: suche einen freien Bucket
  - lineares Sondieren
  - quadratisches Sondieren

Doppelte Hashwerte in einer Liste speichern

- Ein Überläufer ist ein Element, welches eine bereits gefüllte Position in der Hashtabelle zum Überlaufen bringt. Bei der Verkettung der Überläufer werden diese für jeden Bucket in einem Array [oder einer verketteten Liste (siehe 2. Semester)] untergebracht.
- Bei vielen Überläufern kann die Tabelle entarten, d.h. viele Elemente befinden sich in einem gemeinsamen Bucket, mit Zugriffszeiten von O(N).

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eintrag										

42

109

69

Lineares Sondieren

- Bei einer Kollision für T[h(e]] werden der Reihe nach die Buckets überprüft
  - T[h(e) + 1]; T[h(e) + 2], ...
- Das Element wird im ersten so gefundenen freien Bucket abgespeichert.
- Hierbei bezeichnet T[x] den zu x mod N gehörigen Bucket.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eintrag	69	29	42							119

Quadratisches Sondieren

- lineares Sondieren führt oft zu einer Konzentration der Elemente in der Nähe weniger Buckets.
- Quadratisches Sondieren verhindert dies.
  - T[h(e) + 1]; T[h(e) + 4]; T[h(e) + 9]; ....

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eintrag	69		42	29						119

Re-Hashing

- Wird keine Verkettung der Überläufer durchgeführt, so muss die Hashtabelle groß genug sein um alle zu speichernden Elemente aufnehmen zu können.

- Sollte dies nicht der Fall sein, muss sie entsprechend vergrößert werden.

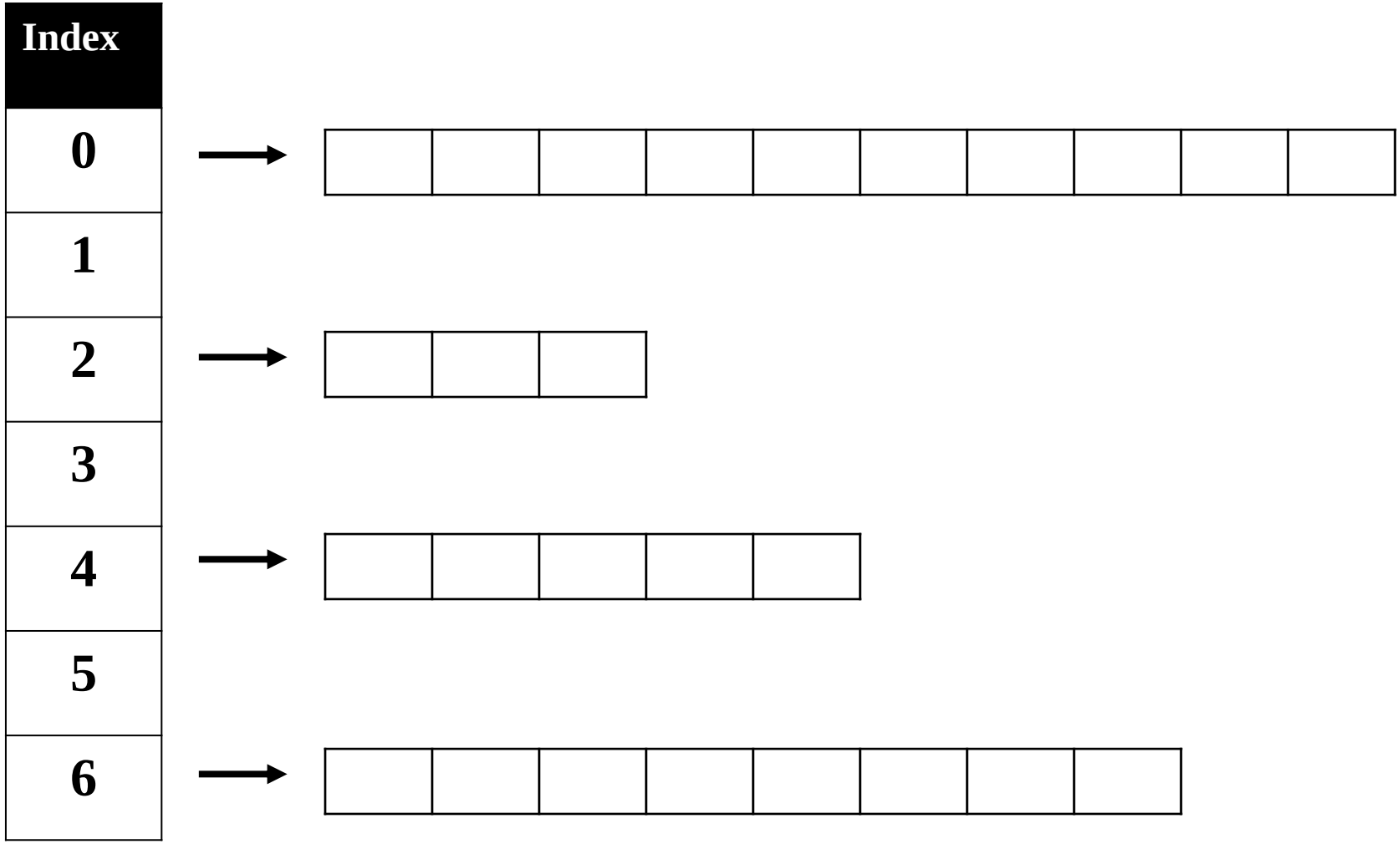
- Dann muss für jeden Eintrag ein neuer Bucket, entsprechend der neu berechneten Hashfunktion, bestimmt werden.

Hashtable mit Listen

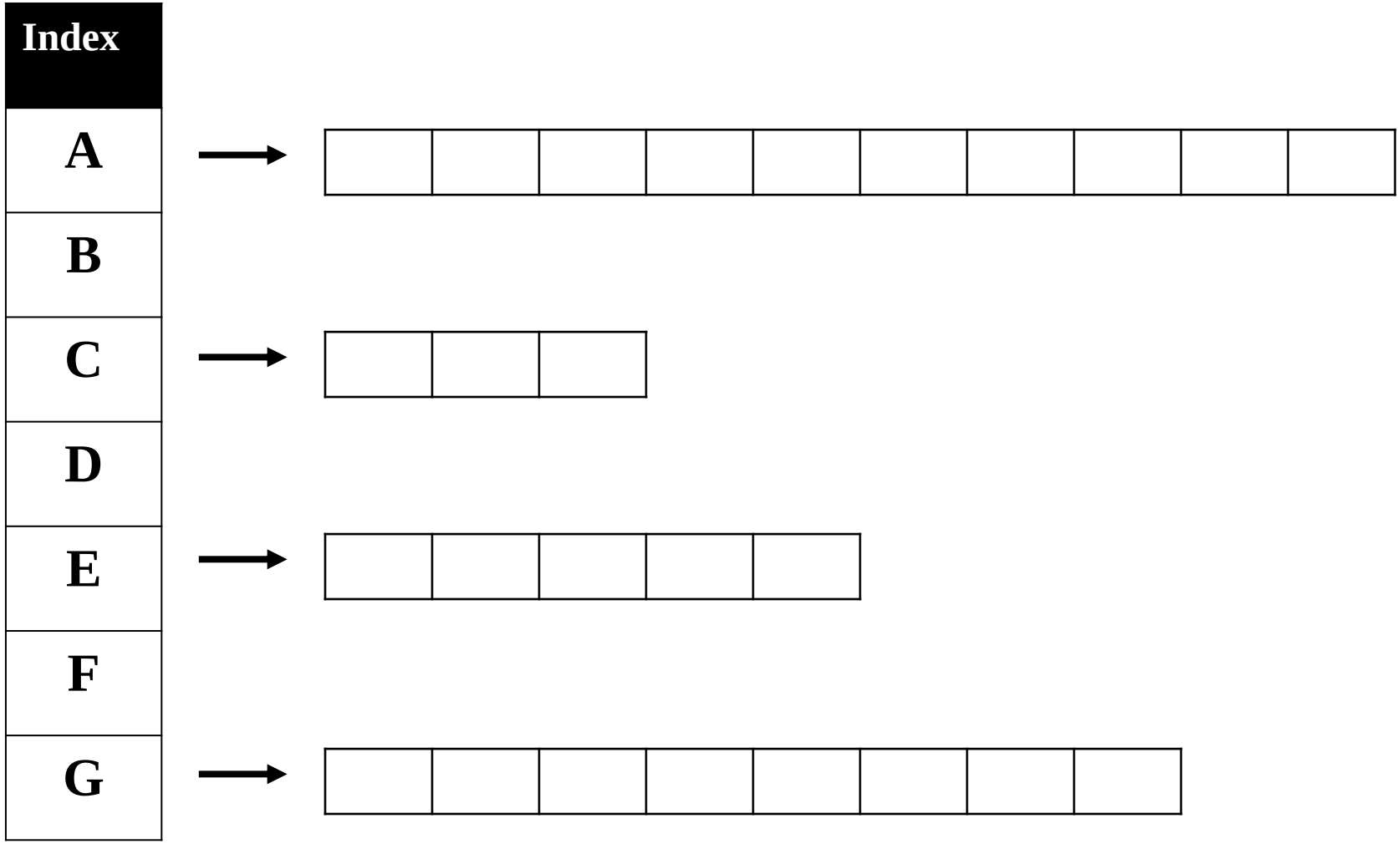
- Bei der Hashtable mit einem Feld muss man **vorher** die Maximalanzahl der zu speichernden Elemente wissen.
- **Alternative:**
  - Man speichert Listen von Elementen
  - (+) Einfaches Einfügen von „doppelten“ Hashwerten
  - (–) Suchen in der Hastable **und** in der Liste

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eintrag	69		42	29						119

Hashtable mit Listen: Beispiel PLZ mit Orten



Hashtable mit Listen: Beispiel Ortsnamen mit Orten



```
import java.util.ArrayList;
//import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;
import java.util.HashMap;
// d:\progs\eclipse\bs\HashTable
class HashTablePLZBeispiel {

  private final int min=1;
  private final int max=99999;
  private final int maxFeld=100000;

  int getPLZ() {
    int randomNum = ThreadLocalRandom.current().nextInt(min, max + 1);
    return randomNum;
  }
```

```
public void test1() {
  int[] feld = new int[maxFeld];
  for (int i=0; i<maxFeld; i++) {
    feld[i] =getPLZ();
  }
  long t1 =  System.currentTimeMillis();
  for (int i=0; i<maxFeld; i++) {
    int plz =getPLZ();
    for (int j=0; j<maxFeld; j++) {
      if (plz == feld[i]) {
        break;
      }  // if
    } // for
  }  // for
  long t2 =  System.currentTimeMillis();
  System.out.println("Zeit test1: "+(t2-t1)+" ms" );  // 8400 ms
}
```

```
public void test2() {
    Object[][] feld = new Object[maxFeld][2];
    for (int i=0; i<maxFeld; i++) {
     feld[i][0] =getPLZ();
     feld[i][1] ="Stadt"+getPLZ();
    }
    long t1 =  System.currentTimeMillis();
    for (int i=0; i<maxFeld; i++) {
     int plz =getPLZ();
     for (int j=0; j<maxFeld; j++) {
       if (plz == (int)feld[i][0]) {
         break;
       }  // if
     } // for
    }  // for
    long t2 =  System.currentTimeMillis();
    System.out.println("Zeit test2: "+(t2-t1)+" ms" );  // 8400 ms
}
```

```
public void test3() {
    Object[] karteikarte=null;
    HashMap<Integer,Object[]> hashtable = new
    HashMap<Integer,Object[]>(150000, 0.70f);
    for (int i=0; i<maxFeld; i++) {
        karteikarte = new Object[2];
        int plz = getPLZ();
        karteikarte[0] =plz;
        karteikarte[1] ="Stadt"+getPLZ();
        hashtable.put(plz , karteikarte);
    }
    long t1 =  System.nanoTime() ; // System.currentTimeMillis();
    for (int i=0; i<maxFeld; i++) {
        int plz =getPLZ();
        karteikarte = (Object[]) hashtable.get( plz );
    }  // for
    long t2 =  System.nanoTime() ; // System.currentTimeMillis();
     System.out.println("Zeit test3: "+(t2-t1)*1.0E-6+" ms" );  // 8400 ms
}
```

```
public void test4() {
    int n=10;
    Object[] karteikarte=null;
    ArrayList<Object[]>[] array = new ArrayList[n];
    for (int i=0; i<n; i++) {
        array[i] = new ArrayList<Object[]>(100000);
    }

    for (int i=0; i<maxFeld; i++) {
        karteikarte = new Object[2];
        int plz = getPLZ();
        int key = plz/10000;
        karteikarte[0] =plz;
        karteikarte[1] ="Stadt"+getPLZ();
        array[key].add(karteikarte);
    }
```

```
public void test4() {
    …
    long t1 =  System.nanoTime() ; // System.currentTimeMillis();
    for (int i=0; i<maxFeld; i++) {
      int plz =getPLZ();
      int key = plz/10000;

      ArrayList<Object[]> liste = array[key];
      for (int j=0; j<liste.size(); j++) {
        Object[] obj =liste.get(j);
        if (plz == (int)obj[0]) {
          break;
        }
      }
    }  // for
    long t2 =  System.nanoTime() ; // System.currentTimeMillis();
    System.out.println("Zeit test4: "+(t2-t1)*1.0E-6+" ms" );  // 8400 ms
}  // test4
```

Hashtable mit Listen: Beispiel PLZ mit Orten

- test1: Lineare SucheZeit: 11355 ms
- test2: Lineare Suche PLZ-Array Zeit: 12049 ms
- test3: Hashtable PLZ-Array Zeit: 14 ms
- test4: n=10 HashArray lin. Suche Zeit: 4567,48 ms
- test5: n=1000 HashArray lin. Suche Zeit: 107,29 ms
- test6: HashTable mit Object Zeit: 7,73 ms
- Test7 n=10 Object-Arrays Zeit: 2840,36 ms
- Test8 n=1000 Object-Arrays Zeit: 64,22 ms

Hashtable mit Listen: Beispiel PLZ mit Orten

- test1: Lineare SucheZeit: 277465 ms
- test2: Lineare Suche PLZ-Array Zeit: 304115 ms
- test3: Hashtable PLZ-Array Zeit: 28 ms
- test4: n=10 HashArray lin. Suche Zeit: 32054,17 ms
- test5: n=1000 HashArray lin. Suche Zeit: 407,76 ms
- test6: HashTable mit Object Zeit: 38,79 ms
- Test7 n=10 Object-Arrays Zeit: 48908,42 ms
- Test8 n=1000 Object-Arrays Zeit: 596,96 ms

Datenaustausch

- Der Datenaustausch zwischen Programmen einer oder unterschiedlichen Plattformen verläuft nicht immer reibungslos:
- Das Datenformat ist binär und proprietär, einzigartig
- Man muss für jedes Programm die Ein- und Ausgabemethoden schreiben
- Problem Little- und Big-Endian (Vertauschung der Byte-Reihenfolge)
  - Big-Endian (Linux)
  - Little Endian (Windows)
  - Big-Endian (Java auch unter Windows)

- Plausibilitätsüberprüfung
- Die weitere Verarbeitung erfordert wiederum individuelle Programme
  - Export nach HTML
  - Export nach PDF
  - Schnittstellen wie WML
  - SMIL

XML-Beispiel

```
<buch>
   <titel>Der Herr der Ringe</titel>
   <verlag>Klett Cotta</verlag>
   <autor>
            <vorname>John Renald Reuel</vorname>
            <nachname>Tolkien</nachname>
   </autor>
</buch>
```

Fast wie HTML !

```
<buch>
   <titel>Der Herr der Ringe</titel>
   <verlag>Klett Cotta</verlag>
   <autor>
            <vorname>John Renald Reuel</vorname>
            <nachname>Tolkien</nachname>
   </autor>
</buch>
<buch>
   <titel>Einführung in die Informatik</titel>
   <verlag>Oldenbourg</verlag>
   <autor>
            <vorname>Heinz-Peter</vorname>
            <nachname>Gumm</nachname>
   </autor>
</buch>
```

```
<regal>
    <buch>
      <titel>Der Herr der Ringe</titel>
      <verlag>Klett Cotta</verlag>
      <autor>
                <vorname>John Renald Reuel</vorname>
                <nachname>Tolkien</nachname>
      </autor>
    </buch>
    <buch>
      <titel>Einführung in die Informatik</titel>
      <verlag>Oldenbourg</verlag>
      <autor>
                <vorname>Heinz-Peter</vorname>
                <nachname>Gumm</nachname>
      </autor>
    </buch>
</regal>
```

Prüfungen bezüglich XML

Wellformed:

- Alle Tag‘s müssen mit einem Anfangstag beginnen und mit einem Endtag enden.
- Es darf nur eine Wurzel, Root, geben

Validierung:

- Die Struktur muss der DTD bzw. der XSD entsprechen.
- Die Anzahl der Elemente wird in der DTD bzw. der XSD definiert.
  - Nichts — genau ein Element
  - ? — kein oder ein Element
  - + — ein oder mehrere Elemente
  - * — kein, ein oder mehrere Elemente
  - {m,n}, {n,}, {n} — nur XSD
- Die weiteren Prüfungen (Anzahl, Wertebereiche, Aufzählung etc.) später

```
<regal>
    <buch>
      <titel>Der Herr der Ringe</titel>
      <verlag>Klett Cotta</verlag>
       <preis> 12,78</preis>
      <autor>
                <vorname>John Renald Reuel</vorname>
                <nachname>Tolkien</nachname>
      </autor>
    </buch>
    <buch>
      <titel>Einführung in die Informatik</titel>
      <verlag>Oldenbourg</verlag>
      <preis> -7,77 </preis>
      <autor>
               <vorname>Heinz-Peter</vorname>
                <nachname>Gumm</nachname>
      </autor>
    </buch>
</regal>
```

Preisproblem

- In DTD nicht erkennbar
- In XSD prüfbar

Eigenschaften von XML

- eXtensible Markup Language
- Ähnelt auf den ersten Blick HTML, ist aber wesentlich flexibler:
  - Hat alle Fähigkeiten von HTML, darüber hinaus aber auch
    - Allgemeiner Standard
    - Definition **eigener *Tags***
    - Man kann mit eigenen und fremden Programmen auf XML-Dokumente zugreifen
    - Vollständige Beschreibungssprache für eigene Datensätze
    - Zielsetzung: *Inhalte (Semantik)* von Dokumenten beschreiben
    - Direkte Speicherung in Datenbanken

Dateierweiterungen (extensions)

- XML: XML-Datei (Inhalt und Struktur)
- DTD: Document Type Definition
  - Definition der Tags und der Struktur (veraltet)
- XSD: XML Schemata
  - Definition der Tags und der Struktur
  - Bessere Datentypen und Wertebereiche

- XSL: Extensible Stylesheet Language
  - XSLT- Datei (Transformation z. B. nach HTML (SAXON)

Schemata, Nachfolger von DTD

- XML Schema beschreibt in einer XML-Schemasprache mittels Datentypen und Hierarchien die Struktur einer XML-Datei.
- Ein XML Schema wird auch als eine XSD (XML Schema Definition) bezeichnet und hat als Datei üblicherweise die Endung „xsd“.

Eigenschaften:

- Schemata ist der Ersatz einer DTD-Datei.
- Bessere Datentypen (int, double, date, time, boolean).
- Bessere Definitionen:
  - bzgl. der Anzahl minOccurs und maxOccurs
  - bzgl. der Einschränkung des Wertebereichs eines Attributs (minExclusive, maxInclusive, minLength, Pattern).

Schemata

Aufbau einer XSD-Datei:

- Header (xsd:schema) als Root
  - **Eine XSD-Datei ist auch eine XML-Datei!**
- Elemente und Unterelemente à la DTD (ComplexType)
- Type-Definitionen für einen Mehrfach-Gebrauch (SimpleType)
- „Vererbung“ mittels base
- ComplexType als Alternative zu base
- Auswahl mit „Choice“

Einfache Datentypen (atomare Datentypen à la Java)

- xsd:string
- xsd:byte, signed 8-Bit
- xsd:unsignedByte
- xsd:short, 16 Bit
- xsd:unsignedShort
- xsd:int, 32 Bit
- xsd:unsignedInt
- xsd:integer, unbegrenzt
- xsd:long, 64 Bit
- xsd:unsignedLong
- xsd:hexBinary MIME
- xsd:float -INF, +INF, NAN
- xsd:double -INF, +INF, NAN
- xsd:decimal, feste Stellen
- xsd:boolean
- xsd:date
- xsd:time
- xsd:dateTime
- xsd:ID ID à la m#
  - muss mit einem Buchstaben beginnen
  - <xsd:attribute name="ID" type="xsd:ID" />
  - Prüft, ob ein Eintrag doppelt ist.

ComplexType: Aufzählung der Attribute eines Elements

<complexType

- abstract = Boolean : false
- block = (#all | List of (extension | restriction))
- final = (#all | List of (extension | restriction))
- id = ID
- mixed = Boolean : false
- name = NCName
- {any attributes with non-schema Namespace...}>
- Content: (annotation?, (simpleContent | complexContent | ((group | all | choice | sequence)?, ((attribute | attributeGroup)*, anyAttribute?))))

</complexType>

ComplexType: Aufzählung der Attribute eines Elements

- mixed=true (Attribute)
- **Content:**
  - **sequence** besteht aus 1,n Elementen
  - simpleContent complexType mit Attribute
  - complexContent Vererbung, extension base
  - choice Man kann man aus **Elementen** wählen.
  - substitution Wahl der Elemente (RadioButton)
  - group Komposition, Element **hat** einen Ball
  - all Reihenfolge nicht zwingend, min=max=1
  - any eigene und externe XSD !!!
  - anyAttribute erlaubt beliebige Attribute

1. Beispiel: Speichern eines Studenten (xsd_bsp1)

XML-Datei: xsd_bsp1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<vorlesung  xsi:noNamespaceSchemaLocation="xsd_bsp1.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <student>
      Meier
    </student>

    <student>Schulze</student>

</vorlesung>
```

1. Beispiel: Speichern eines Studenten (xsd_bsp1)

XSD-Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="vorlesung">      // root
  <xsd:complexType>                 // à la Karteikarte, Object
      <xsd:sequence>
          <xsd:element name="student"
                  type="xsd:string" maxOccurs="unbounded" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Erläuterung:

- Die XSD-Datei ist wie eine XML-Datei aufgebaut und kann auch nach deren Regeln getestet werden.
- Die beiden ersten Zeilen zeigen die xml-Startsequenz:
  - Version und Schema
- Die dritte Zeile startet mit dem „root-Element“, hier „vorlesung“
- Mit den Anweisungen „complexType“ und „sequence“ werden Unterelement und Strukturen definiert.
- Der Student hat als Datentyp eine Zeichenkette.
- Die Anzahl wird mit „minOccurs“ und „maxOccurs“ bestimmt.

2. Beispiel: Speichern eines Studenten (xsd_bsp2)

```
<?xml version="1.0" encoding="UTF-8"?>
<vorlesungen   xsi:…. Location="xsd_bsp2.xsd" …>
  <vorlesung>
    <student>
    <name>Meier</name>
      <note>4.566 </note>
    </student>
    <student>
      <name>Schulze</name>
      <note> 3.0 </note>
    </student>
    <dozent> Müller </dozent>
  </vorlesung>
  </vorlesungen>
```

2. Beispiel: XSD-Datei (xsd_bsp2.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="vorlesungen">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="vorlesung" type="vorlesungsType" maxOccurs="unbn"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
  <xsd:complexType name="vorlesungsType">
   <xsd:sequence>
    <xsd:element name="student" type="studentType" maxOccurs="unbounded" />
    <xsd:element name="dozent" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
```

2. Beispiel: XSD-Datei (xsd_bsp2.xsd)

```
<xsd:complexType name="studentType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="note" type="notenType" />
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="notenType">
  <xsd:restriction base="xsd:double">
    <xsd:minInclusive value="1.0"/>
    <xsd:maxInclusive value="5.0"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Elemente einer XSD-Datei:

- Bemerkungen (annotation)
- Bemerkungen können an jeder Stelle eingetragen werden:

```
<xsd:annotation>
    <xsd:documentation>XML macht Spaß</xsd:documentation>
</xsd:annotation>
```

Begrenzungen in SimpleType, also „Attribut“

- Um die einzelnen Knoten einschränken zu können, stehen folgende Techniken zur Verfügung:

- **Anzahl** (minimal und maximal, auch in ComplexType)
  - minOccurs bestimmt die untere Grenze (einschließlich).
  - maxOccurs bestimmt die obere Grenze (einschließlich). maxOccurs kann auch "unbounded" als Wert haben.

- **Wertebereich eines Elementes:**
  - minExclusive >
  - minInclusive >=
  - maxExclusive <
  - maxInclusive <=

Weitere Begrenzungen in SimpleType

- **Längenbeschränkungen bei Strings:**
  - length
  - minLength
  - maxLength

```
<xsd:simpleType name="myType">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="3"/>
  </xsd:restriction>
</xsd:simpleType>
```

Weitere Begrenzungen in SimpleType

- **enumeration**
  - Beschränkung durch Angabe alternativer Werte.
  - enumeration value="ATI" , "IB" , "MB" oder "W"

```
<xsd:simpleType name="computerType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Tablet"/>
    <xsd:enumeration value="Smartphone"/>
    <xsd:enumeration value="Subnotebook"/>
    <xsd:enumeration value="Desktop"/>
</xsd:restriction>
</xsd:simpleType>
```

Weitere Begrenzungen in SimpleType

- **pattern**
  - Beschränkung durch Angabe eines regulären Ausdrucks.
  - Deutsche Postleitzahlen: pattern value="(D)?[0-9]{5}"
  - „D“ ist optional gefolgt von fünf Ziffern

```
<xs:element name="Name">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][a-zßüäö]+"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
<xs:pattern value="male|female|unisex"/>
```

Weitere Begrenzungen in SimpleType:

Begrenzungen bei Dezimalstellen

- Einschränkung der Dezimalstellen (Gesamtzahl und Nachkommastellen)
  - totalDigits
  - fractionDigits

```
<xsd:simpleType name="decimalTyp">
  <xsd:restriction base="xsd:decimal">
    <xsd:totalDigits value="20"/>
    <xsd:fractionDigits value="11"/>
  </xsd:restriction>
</xsd:simpleType>
```

Weitere Begrenzungen in SimpleType:

whiteSpace – (Leerzeichen, Tabs, CR, LF)

- Mögliche Varianten:
  - "preserve" ist der Standardwert und alle "white space" bleiben erhalten.
  - "replace" ersetzt alle "white space characters" mit Leerzeichen.
  - "collapse löscht die "white spaces" am Anfang und am Ende.

```
<xs:element name="addresse">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Beispiele

- minOccurs bzw. maxOccurs dürfen nur im Elementtag eingetragen werden. Im Simpletype sind sie nicht erlaubt.

```
<xsd:element name="anzStudenten"  type="xsd:integer" minOccurs="0"  />

 <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="student"
                    type="xsd:string" minOccurs="1"  maxOccurs="unbounded" />
     </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
```

Selbstdefinierter Typ für Mehrfachverwendung:

```
<xsd:element name="monat1"   type="monatIntType">
<xsd:element name="monat2"   type="monatIntType">
<xsd:element name="monat3"   type="monatIntType">
<xsd:element name="monat4"   type="monatIntType">

<xsd:simpleType name="monatIntType">
   <xsd:restriction base="xsd:integer">
     <xsd:minInclusive value="1"/>                ≥
      <xsd:maxExclusive value="13"/>              <
   </xsd:restriction>
</xsd:simpleType>
```

Selbstdefinierter Typ mit interner Definition:

1. Möglichkeit (implizit, ohne den Typ „autoType“)

```
<xsd:element name="auto">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Audi"/>
      <xsd:enumeration value="Volkswagen"/>
      <xsd:enumeration value="BMW"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Selbstdefinierter Typ mit externer Definition:

2. Möglichkeit

```
<xsd:element name="auto" type="autoType"/>

<xsd:simpleType name="autoType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Audi"/>
    <xsd:enumeration value="Volkswagen"/>
    <xsd:enumeration value="BMW"/>
  </xsd:restriction>
</xsd:simpleType>
```

simpleContent

```
<xsd:element name="studentType">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="name" type="xs:string" />
        <xsd:restriction base="xsd:string">
          <xsd:minLength value="3"/>
        </xsd:restriction>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

3. Beispiel: (xsd_bsp3.xml): Note als Enumeration

```
<vorlesungen  …Location="xsd_bsp3.xsd"  … >
    <vorlesung>
        <student>
            <name>Meier</name>
            <note> 4.0 </note>
        </student>
        <student>
            <name>Schulze</name>
            <note> 3.0 </note>
        </student>
        <dozent> Müller </dozent>
    </vorlesung>
</vorlesungen>
```

3. Beispiel: (xsd_bsp3.xsd): Note als Enumeration

```
<xsd:complexType name="studentType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="note" type="notenType" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:simpleType name="notenType">
  <xsd:restriction base="xsd:double">
  <xsd:enumeration value="1.0"/>
          …
  <xsd:enumeration value="4.0"/>
  <xsd:enumeration value="5.0"/>
 </xsd:restriction>
 </xsd:simpleType>
```

4. Beispiel: (xsd_bsp4.xml): Note als Enumeration

```
<vorlesungen  …Location="xsd_bsp4.xsd"  … >
    <vorlesung>
        <student>
            <name>Meier</name>
            <note> 4.0 </note>
        </student>
        <student>
            <name>Schulze</name>
            <note> 3.0 </note>
        </student>
        <dozent>Prof. Müller </dozent>
    </vorlesung>
</vorlesungen>
```

4. Beispiel: (xsd_bsp4.xsd): Name mit Pattern

```
<xsd:simpleType name="namensType">
  <xsd:restriction base='xsd:string'>
    <xsd:minLength value='2'/>
    <xsd:pattern value="[A-Z][a-züöäß]{1,}"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="dozentType">
  <xsd:restriction base='xsd:string'>
    <xsd:minLength value='2'/>
    <xsd:pattern value="((Prof. )|(Dr. ))?[A-Z][a-züöäß]{1,}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Typen

• Literal escape Zeichen	\t, \n, \\, \., \[, \], \{, \}
• Gruppe für **ein** Zeichen	[...]
• Bereich	a-z
• Oder	([a-k]|[123])
• Oder	(abc|def)
• Hintereinander	Unr
• Negation	[^0-9] Keine Ziffern

Wildcards:

- . beliebiges Zeichen (Punkt)
- ? kein oder ein Zeichen, besser {0,1}, veraltet
- + ein oder mehrere Zeichen, besser {1,}, veraltet
- {n} das vorangegangene Zeichen tritt genau n-mal auf
- {n,} das vorangegangene Zeichen tritt n-mal oder öfter auf
- {n,m} das vorangegangene Zeichen tritt mindestens n-mal und maximal m-mal auf

Welche Texte passen zu dem Muster:

- [0-9]{1,}_[A-Za-z] _ = Leerzeichen
- [A-Za-züäöÜÄÖß]{1,7}
- ([abc]|[012]){3}

Welche Muster passen zu den Aufgaben:

- Uhrzeit mit dem Format HH:MM 00:00 bis 23:59
- Uhrzeit mit dem Format HH:MM 00:00 bis 11:59_am bzw. 00:00 bis 11:59_pm
- Matrikelnummer: M12345, M1, m1, m1111
- Postleitzahl 00000 bis 99999
- KFZ Kennzeichen (Beispiele)
  - HBS_H_1
  - HBS_AI_1
  - HBS_AI_19
  - HBS_AI_199 fehlerhaft
  - BS_AI_199 ok
  - HBS_AI_1999 fehlerhaft
  - BS_AI_1999 ok
- Ein Portzahl eines Netzwerks von 0 bis 455

Erläuterung:

- Der Vorname und Nachname beziehen sich auf den Namenstype. Als Regel wurde ein Pattern verwendet
  - <xsd:pattern value="[A-Z][a-züöäß]{1,}"/>

- Der Dozentenname wird auch mit einem Pattern geprüft.
- Am Anfang optional „Prof“ oder „Dr. “. Danach ein normaler Nachname, der mit einem Großbuchstaben anfängt. Danach beliebige Kleinbuchstaben. Mehrfachnamen sind **noch nicht** erlaubt.
  - <xsd:pattern value="((Prof.)|(Dr.))?[A-Z][a-züöäß]{1,}"/>

Beispiel: Rechnung

Aufbau:

- rechnungen
  - rechnung
    - datum
  - kunde
    - vorname
    - nachname
    - knr
- artikel
  - anr
  - einzelpreis
  - anzahl
  - gesamtpreis
- summe
  - netto
  - mwst
  - brutto

Beispiel: Rechnung.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rechnungen  …Location="Rechnung.xsd"   …>

    <rechnung>

        <datum>2015-12-01</datum>
        <kunde>
            <vorname>Petra</vorname>
            <nachname>Schulze</nachname>
            <knr>1234</knr>
        </kunde>
```

Beispiel: Rechnung.xml

```
        <artikel>
            <anr>12345</anr>
            <einzelpreis>1.34</einzelpreis>
            <anzahl>11</anzahl>
            <gesamtpreis>14.74</gesamtpreis>
        </artikel>
        <summe>
            <netto>14.74</netto>
            <mwst>2.80</mwst>
            <brutto>17.54</brutto>
        </summe>
    </rechnung>
</rechnungen>
```

Beispiel: Rechnung.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema …  ww.w3.org/2001/XMLSchema">

<xsd:element name="rechnungen">
<xsd:complexType >
  <xsd:sequence>
   <xsd:element name="rechnung"
      type="rechnungsType" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Beispiel: Rechnung.xsd

```
<xsd:complexType name="rechnungsType" >
  <xsd:sequence>
   <xsd:element name="datum" type="xsd:date" />
   <xsd:element name="kunde" type="kundenType" />
   <xsd:element name="artikel" type="artikelType" />
   <xsd:element name="summe" type="summenType" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="kundenType">
  <xsd:sequence>
   <xsd:element name="vorname" type="namensType" />
   <xsd:element name="nachname" type="namensType" />
   <xsd:element name="knr" type="knrType" />
  </xsd:sequence>
</xsd:complexType>
```

Beispiel: Rechnung.xsd

```
<xsd:simpleType name="knrType">
  <xsd:restriction base="xsd:unsignedInt">
    <xsd:minExclusive value="0"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="artikelType">
  <xsd:sequence>
   <xsd:element name="anr" type="xsd:unsignedInt" />
   <xsd:element name="einzelpreis" type="preisType" />
   <xsd:element name="anzahl" type="anzahlType" />
   <xsd:element name="gesamtpreis" type="preisType" />
  </xsd:sequence>
</xsd:complexType>
```

Beispiel: Rechnung.xsd

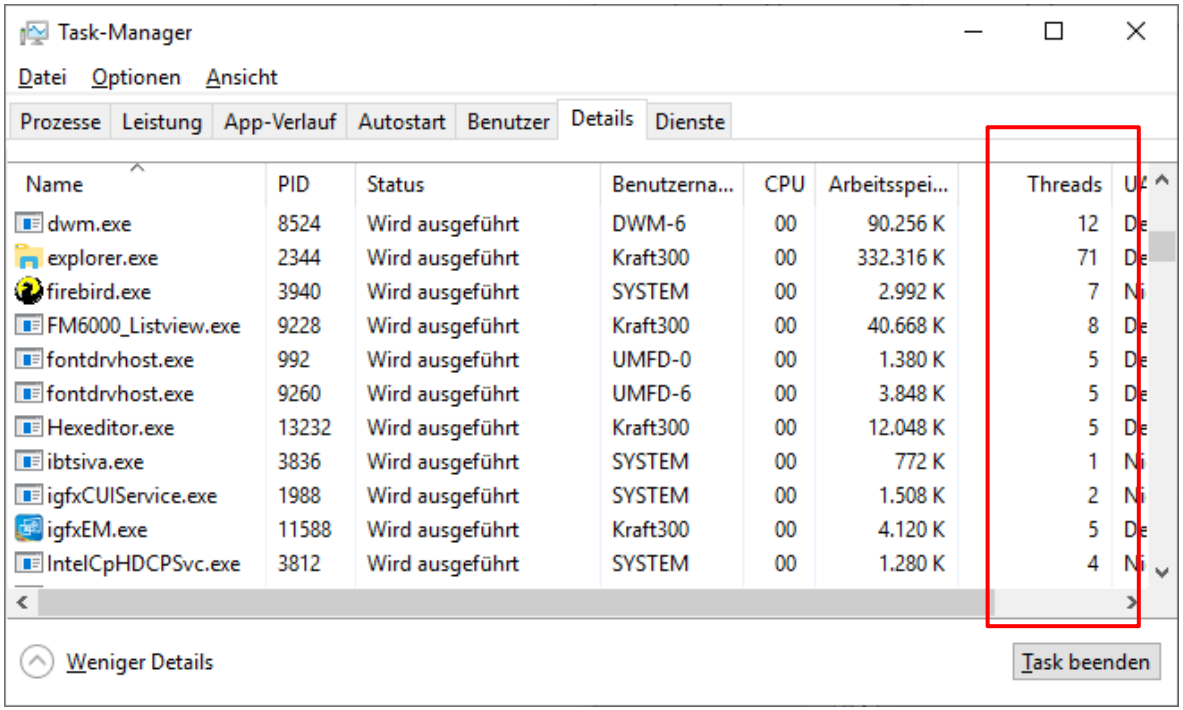
```
<xsd:complexType name="summenType">
  <xsd:sequence>
   <xsd:element name="netto" type="preisType" />
   <xsd:element name="mwst" type="preisType" />
   <xsd:element name="brutto" type="preisType" />
  </xsd:sequence>
</xsd:complexType>

          // eventuell wäre double besser. Zum Beispiel 1,45 Tonnen
<xsd:simpleType name="anzahlType">
  <xsd:restriction base="xsd:int">          // oder unsignedInt
   <xsd:minExclusive value="0"/>
   </xsd:restriction>
</xsd:simpleType>
```

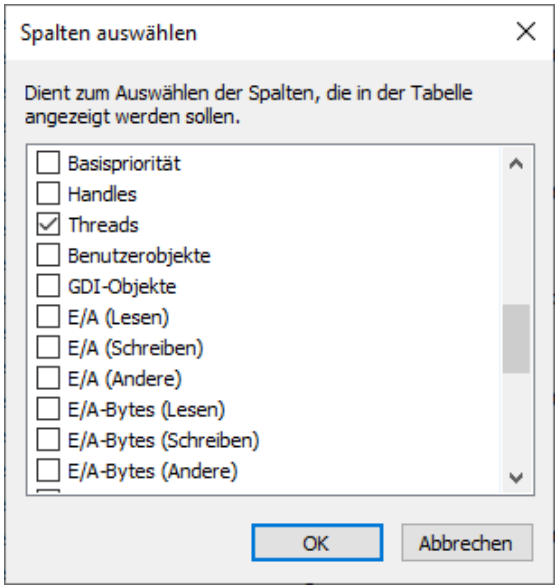
Beispiel: Rechnung.xsd

```
<xsd:simpleType name="preisType">
  <xsd:restriction base="xsd:double">
    <xsd:minExclusive value="0.0"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="namensType">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="2"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Thread / Backgroundworker

- Moderne Rechern haben mehrere CPU‘s.
- Konventielle Programmierung benutzt nur **eine** CPU.
- Im Taskmanager wird die Spalte „Tasks“ für die Threads des Programms verwendet.





Thread: Beispiel Primzahlberechnung

Berechnen der Primzahlen bis 600.000 in Java

Verfahren	Zeit	
1 Thread	50258 ms	
2 Threads	30963 ms	24410 ms
4 Threads	17584 ms	12248 ms
6 Threads	13460 ms	11256 ms
8 Threads	11315 ms	10625 ms
10 Threads	10222 ms	10521 ms
12 Threads	9773 ms	
14 Threads	9310 ms	
16 Threads	8904 ms	
18 Threads	8870 ms	
20 Threads	8869 ms	10269 ms
50 Threads	8368 ms	10193 ms
100 Threads	8223 ms	10074 ms

Compiler

Quellcode (source):

- in einer Hochsprache.
- Wird nicht von der CPU verstanden

Zielprogramm:

- in Maschinensprache (Hexadezimale-Befehle)
- Inhalt besteht aus 0 und 1 (Binärdaten) $\Rightarrow$ Binärdatei, Binaries

Compiler:

- wandelt das Hochsprachenprogramm in Maschinencode um:
  - Lexikalische Analyse:
    - Erzeugt die Wörter und Zeichen des Quellprogramms.
    - printf ("Hallo") ;
  - Syntaktische Analyse:
    - die Grammatik des Quellprogramms geprüft: **printf "Hello"**
  - Semantische Analyse:
    - Ergibt das Quellprogramm Sinn: **printf (4711)**

Compiler

Quellcode (source):

- (1+3)*(4+5)

Compiler:

- Programm liest eine Zeile und baut eine Baumstruktur aus
- Frage:
  - Was ist eine Zahl?
    - 123
    - 1A
    - hallo
  - Was ist ein gültiger Ausdruck:
    - (2)
    - 12+33
    - (1+)*4
- **Antwort:**
  - Kontextfreie Grammatik

```
class SimpleCompiler {
    private int nextchar;
    static boolean isdigit(int c) {
        return c >= '0' && c <= '9';
    }
    void error() {
        System.out.println("Syntax Error");
    }

    int number(int startWert) {
        int value = 10 * startWert + nextchar - '0';
        nextchar = Console.Read();
        if (isdigit(nextchar)) {
            value = number(value);
        }
        return value;
    }

    int factor() {
        int value = 0;
        if (nextchar == '(') {
            nextchar = Console.Read();
            if (nextchar == '(' || isdigit(nextchar)) {
                value = expr();
                if (nextchar != ')')
                    error();
                    nextchar = Console.Read();
            } else
                error();
        } else  {   // if (isdigit(nextchar))
            value = number(0);
        }
        return value;
    }
    int term() {
        int value = factor();
        if (nextchar == '*') {
            nextchar = Console.Read();
            if (nextchar == '(' || isdigit(nextchar)) {
                value *= term();
            } else
                error();
        }
        return value;
    }
    int expr() {
        int value = term();
        if (nextchar == '+') {
            nextchar = Console.Read();
            if (nextchar == '(' || isdigit(nextchar)) {
                value += expr();
            } else
                error();
        }
        return value;
    }
    void stmt() {
        int value = expr();
        if (nextchar == '=') {
            nextchar = Console.Read();
            Console.WriteLine(value);
        } else
            error();
    }
    public void test1() {
        nextchar = Console.Read();
        if (nextchar == '(' || isdigit(nextchar)) {
            stmt();
        } else
            error();
        }
```

Compiler

Nun eine einfachere Lösung in C mit Hilfe der GNU-Compilerbau-Werkzeuge ex und bison:

```
%{
#include "adamry.tab.h"
extern int yylval;
%}
%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }
[\t\n ] ;
. return yytext[0];
%%
```

\t\n_: tab, return, space

Compiler

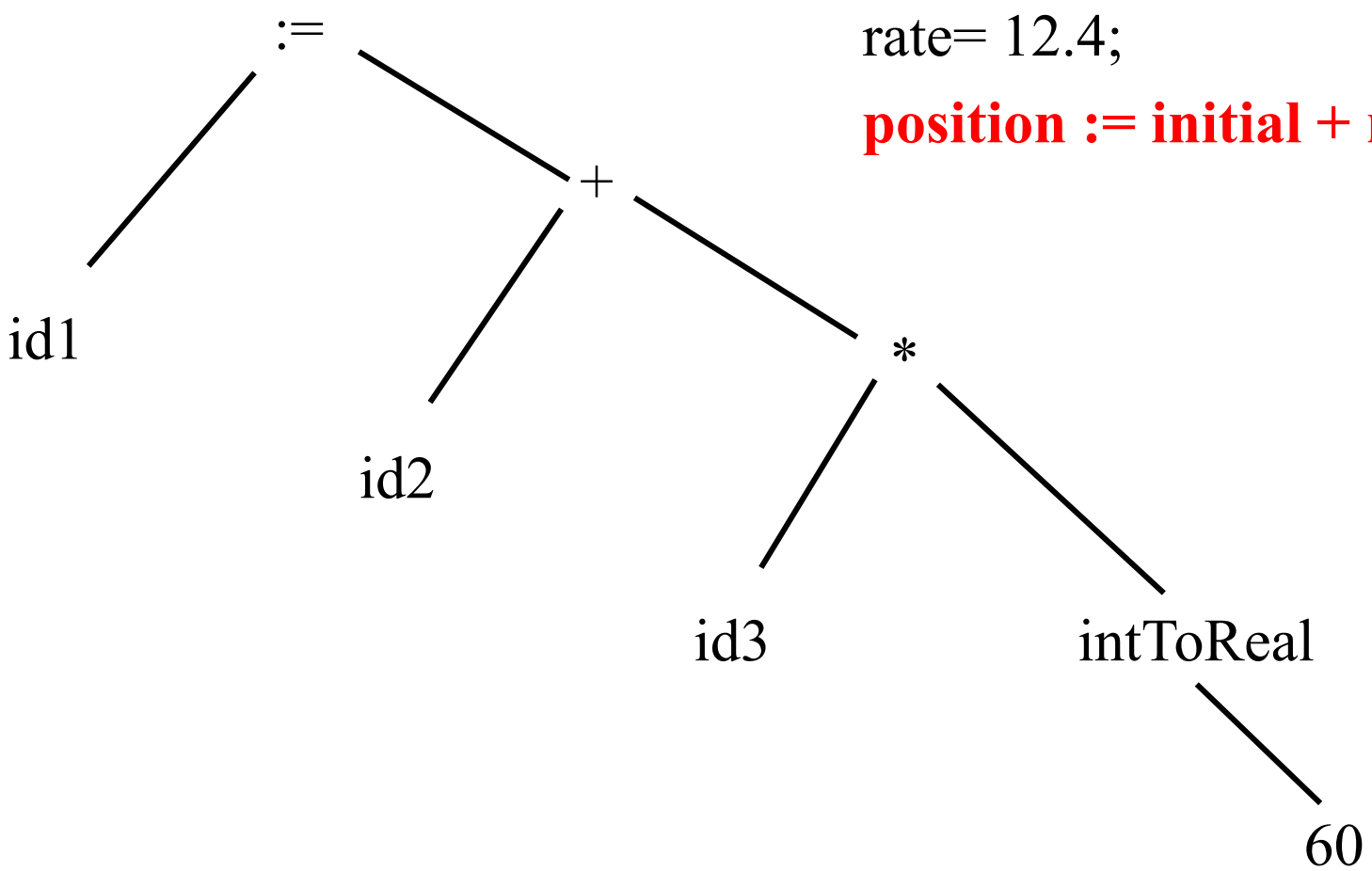
Quellcode: **position := initial + rate * 60;**

- Lexikalische Analyse (scanning):
  - 1. Bezeichner (position) id1
  - 2. Zuweisungssymbol :=
  - 3. Bezeichner (initial) id2
  - 4. Additionssymbol +
  - 5. Bezeichner (rate) id3
  - 6. Multiplikationssymbol *
  - 7. Konstante (60)

Compiler: Contextfreie Grammatik

Quellcode:

```
double position, initial, rate;
initial=33.3;
rate= 12.4;
position := initial + rate * 60;
```



Compiler: Contextfreie Grammatik

double position, initial, rate;

initial=33.3;

rate= 12.4;

position := initial + rate * 60;

Zwischencode

- temp1 := inttoreal(60)
- temp2 := id3 * temp1
- temp3 := id2 + temp2

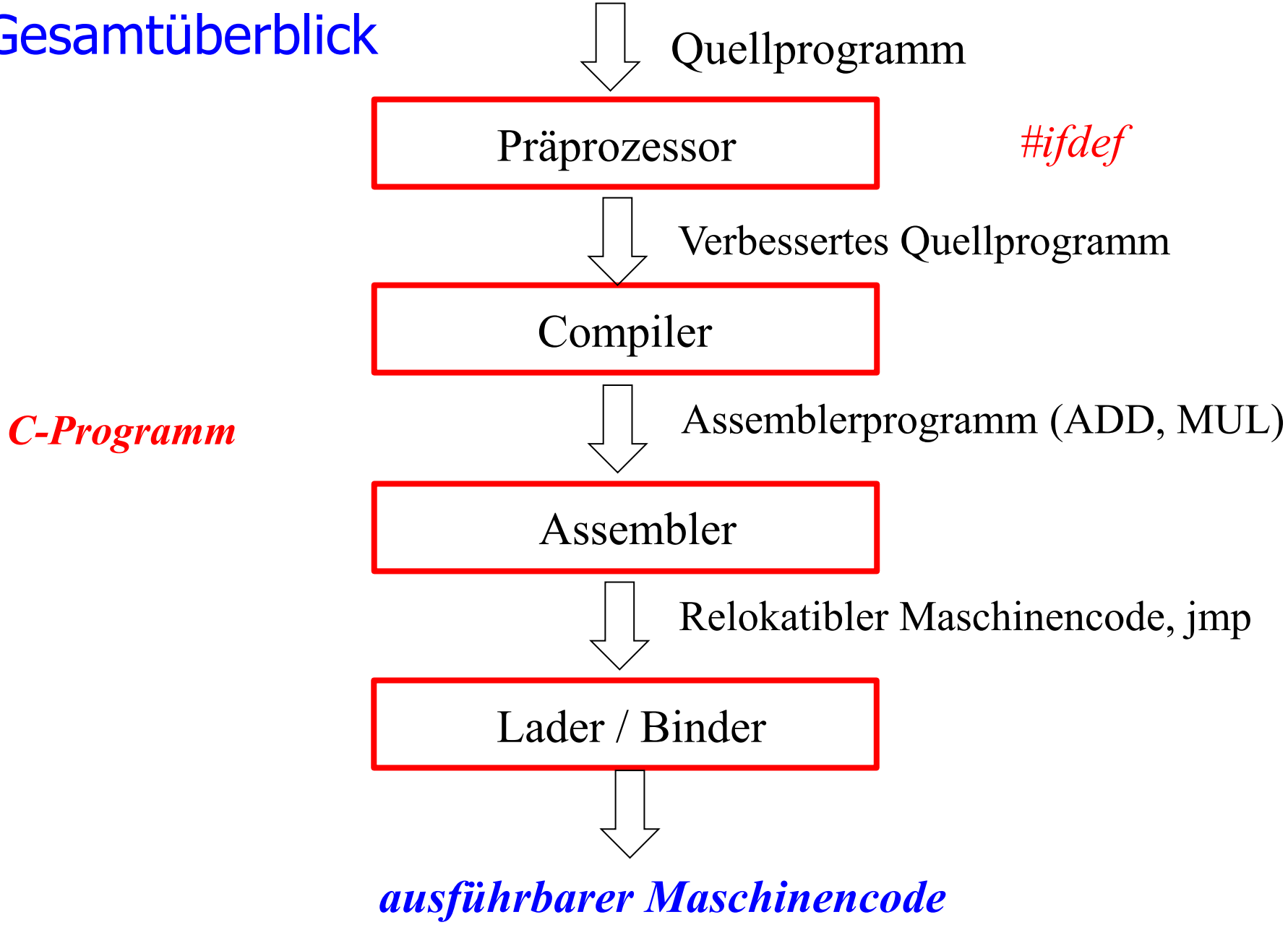
Optimierung

- temp1 := id3 * 60.0
- temp2 := id2 + temp1

Maschinencode

- MOVF Reg2, id3
- MULF Reg2,#60.0
- MOVF Reg1, id2
- ADDF R2, R1
- **MOVF id1, R2**

Gesamtüberblick



Tipps

Abfragen:

int fb:

enthält die Werte 1 bis 3

Abfragen:

```
if (fb==1) {
   …
}
else {
    if (fb==2) {
        …
    }
    else {
        // hier fb3
    }
}
```

Tipps

Abfragen:

int fb:

enthält die Werte 1 bis 3

Abfragen:

```
if (fb==1) {
   …
}
else {
    if (fb==2) {
        …
    }
    else {
        if (fb==3) {
            …
        }
        else {
            // hier ERROR
        }
    }
}
```

Tipps

Abfragen:

int fb:

enthält die Werte 1 bis 3

Abfragen:

```
switch (fb) {
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    default:
        ERROR
        break;
}
```

Tipps

const vs. enum:

```
const int FB =1;
const int VW =2;
const int W =3;

int fachbereich = VW;
druckeFB(fb);            // druckeFB(int fb) {…}

int fachbereich = 4;
druckeFB(fb);
```

Tipps

const vs. enum:

```
enum Fachbereich {
 FB, VW, W
}

Fachbereich fachbereich = Fachbereich.VW;
druckeFB(fb);            // druckeFB(Fachbereich fb) {…}

int fachbereich = 4;
druckeFB(fb);                    //ERROR beim compilieren
```

Tipps

```
class Fachbereich extends Enum {
    public static final AI = new Fachbereich("AI", 1);
    public static final VW = new Fachbereich("VW", 2);
    public static final W = new Fachbereich("W", 3);

    public Fachbereich(String s, int i) {
        super( s, i );
    }
}

Fachbereich fachbereich = Fachbereich.VW;
druckeFB(fb);            // druckeFB(Fachbereich fb) {…}
```

Tipps

Tipp:

- „**const**“ bitte nur verwenden, wenn man bestimmte Zahlen, Grenzen braucht.

- „**enum**“ bitte verwenden, wenn man eine bestimmte Menge von Zahlen resp. Zustände braucht.

Tipps

Zusammenstellen von String :

```
int n=1000;
String s="";
for (int i=1; i<=n; i++) {
    s = s + Integer.toString(i) + "\n";
}
```

Bemerkungen:

- Bei jedem Durchlauf werden **DREI** String erzeugt
- Am Ende ist ein Zeilenumbruch!

Tipps

Zusammenstellen von String :

```
int n=1000;
String s="";
for (int i=1; i<=n; i++) {
    if (i==1)
        s = s + Integer.toString(i);
    else
        s = s + "\n"  +  Integer.toString(i);   // es werden 3 String erzeugt
}
```

Tipps

Zusammenstellen von String mit einem StringBuilder :

```
int n=1000;
StringBuilder sb= new StringBuilder();
for (int i=1; i<=n; i++) {
    if (i==1) {
        sb.append( Integer.toString(i) );
    }
    else {
        sb.append( "\n"  +  Integer.toString(i) );
    }
}
```

Tipps

Zusammenstellen von String mit einem StringBuilder :

```
int n=1000;
StringBuilder sb= new StringBuilder();
for (int i=1; i<=n; i++) {
    if (i==1) {
        sb.append( Integer.toString(i) );
    }
    else {
        sb.append( "\n");
        sb.append( Integer.toString(i)  );
    }
}
```