

Mitschrift

zur Vorlesung:

Algorithmen und Programme

gehalten von:

Prof. Dr. Ing. F. Wahl
Institut für Robotik und Prozeßinformatik
Technische Universität Braunschweig

bearbeitet von:

Thorsten Kamphenkel
(t.kamphenkel@tu-bs.de)

Algorithmen und Programme

Vorwort

Diese Vorlesungsmitschrift beinhaltet den von Professor Dr. Wahl im WS 2000/2001 angeschriebenen Vorlesungsstoff.

Aufgrund der hohen Schreiarbeit, welcher der/die Student/in während der Vorlesung ausgesetzt war bzw. ist, erschien es mir, auch aus eigenem Interesse, sinnvoll, diese Mitschrift zu erstellen. Sie soll sowohl der Aufwandsersparnis des Mitschreibens während der Vorlesung, als auch als übersichtliche Unterlage zu eigenen Übungs- und Rekapitulationszwecken dienen.

Allerdings ersetzt sie nicht die Vorlesung, in der die Sachverhalte konkret erläutert werden. Deshalb sollte, so stelle ich es mir zumindest vor, diese Mitschrift genau mit dem Vortrag verglichen werden, wobei sich sicherlich gleichzeitig ein Lernerfolg einstellt.

Da der Stoff so umfangreich ist, scheint es mir dennoch unerlässlich, nebenbei **Notizen** zu den einzelnen Passagen zu machen, da das alleinige Lesen dieser Mitschrift keine Garantie für eine erfolgreiche Klausur geben kann!

Ich erhoffe mir, daß ich mit dieser Mitschrift die Aufmerksamkeit mehr auf das Mitarbeiten lenken kann, denn dauerndes Abschreiben von der Tafel verhindert, diese Erfahrung habe ich zumindest dabei gemacht, das gezielte Lernen und vermindert die Motivation.

Ich wünsche allen viel Erfolg beim Lernen und weise darauf hin, daß ich für die Richtigkeit der in dieser Mitschrift gemachten Angaben keine Gewähr geben kann. Außerdem kann sich der ein oder andere Tippfehler eingeschlichen haben. Über Hinweise darüber wäre ich sehr dankbar,

Thorsten Kamphenkel

(Benutzte Schriftarten: MathSoft Text , StarMath, Symbol, Arial, Times New Roman – bei daraus resultierenden Problemen: bitte mailen!)

Algorithmen und Programme

Inhaltsverzeichnis

Kapitel	Titel	Seite
1	Einführung	4
2	Algorithmische Grundkonzepte	6
2.1	Eigenschaften von Algorithmen	6
2.2	Daten, Operanden und Operationen	7
2.3	Imperative Algorithmen	10
2.4	Strukturierung von Algorithmen / Programmen	12
2.5	Prozeduren	13
2.6	Statische (implizite) Datenstrukturen	16
3	Komplexität von Algorithmen	19
3.1	Einführung	19
3.2	Asymptotische Analyse / Groß-O-Notation	20
3.3	Beispiele	22
3.4	Weitere Komplexitätsmaße	25
4	Sortierverfahren	25
4.1	Einführung	25
4.2	Ein einfacher Sortieralgorithmus	27
4.3	Quicksort	30
4.4	Sortieren	33
5	Dynamische Datenstrukturen	34
5.1	Speicherverwaltung mit 'new' bzw. 'delete'	34
5.2	Lineare Listen	35
5.3	Bäume	39
5.4	Binäre Suchbäume	42
5.5	Graphen	45
5.6	Adreßberechnung (Hash – Verfahren)	50
6	Abstrakte Datentypen und objektorientierte Programmierung	53
6.1	Abstrakte Datentypen (ADT)	53
6.2	ADT – Beispiel: „Warteschlange“	54
6.3	Objektorientierte Programmierung – Einführung	57
7	Einige theoretische Aspekte	67
7.1	Schaltnetze, Automaten, Maschinenmodelle	67
7.2	Was ist berechenbar?	70
7.3	Halteproblem	71

Algorithmen und Programme

1. Einführung

Intuitiver Algorithmusbegriff: **Algorithmus = Verarbeitungsvorschrift**

↳ im Alltag z. B.: Kochrezepte, Bastelanleitungen, Gebrauchsanweisungen,...

Geschichtliches: „Algorithmus“, abgeleitet aus dem Namen eines Arabers (lat.: liber Algorithmi), der sich um 800 n. Chr. mit Erbschaftsteilung beschäftigt hat.

Entwicklung von Algorithmen geht bis ins Altertum zurück:

- Alt-Ägypten: Verfahren zur Multiplikation
- Alt-Babylon: erste Verfahren zur Lösung spezieller Gleichungen
- um 300 v. Chr.: „Euklids Algorithmus“ für den größten gemeinsamen Teiler

Beispiel für „Euklids Algorithmus“:

Der ggT. von 30 und 8: $\rightarrow 30-8=22 \rightarrow 22-8=14 \rightarrow 14-8=6 \rightarrow 8-6=2 \rightarrow 6-2=4 \rightarrow 4-2=2$
Bei Übereinstimmung ($2=2$) ist der ggT. gefunden (**2** - wie man auch sofort sieht ...)!

- 1545/55: Auflösung höherer algebraischer Gleichungen
- seitdem: Entwicklung vieler Algorithmen für spezielle Aufgaben
- 1931: „Gödel“: erste mathematische Präzisierung des Algorithmusbegriffes
- seitdem: Ausbau der Algorithmustheorie (Berechenbarkeit, Entscheidbarkeit,...)

Man spricht von einem Algorithmus (im engeren Sinne), wenn die Vorschriften / Regeln präzise, eindeutig, vollständig und ausführbar sind.

Definition:

„Ein Algorithmus ist eine in einer festgelegten Sprache präzise formulierte Verarbeitungsvorschrift, die unter Verwendung ausführbarer elementarer Operationen einen Eingangszustand bzw. Eingabewerte in einen Ausgangszustand bzw. in Ausgabewerte überführt.“

$f: \text{Eingabe} \rightarrow \text{Ausgabe}$

(Abbildung wird auch als „Semantik“ bezeichnet)

Algorithmen und Programme

Beispiele:

- Addition zweier Zahlen:

$$f: \mathbb{Q} \times \mathbb{Q} \longrightarrow \mathbb{Q}$$

$$f(p,q) = p + q$$

- Primzahltest:

$$f: \mathbb{N} \longrightarrow \{\text{ja, nein}\}$$

$$f(n) = \begin{cases} \text{ja, falls } n: \text{ Primzahl} \\ \text{nein sonst} \end{cases}$$

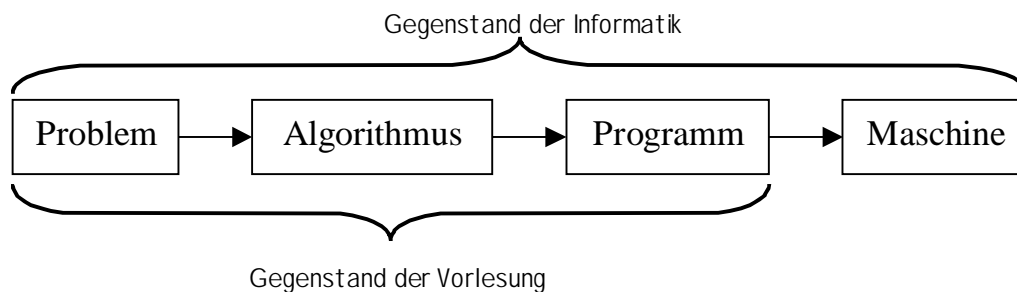
- Einfügen einer Karte in sortierte Kartei:

$$k \in K: \text{ Karteikarte}$$

$$sk: \text{ sortierte Folge von Karteikarten}$$

$$f(k, sk) = sk'$$

Algorithmen dienen der Lösung von Problemen; sie werden als Programme so abgefaßt, daß sie von Rechnern ausgeführt werden können:



➔ **Informatik** ist die Wissenschaft, die sich mit der **Beschreibung**, **Analyse** und **Gestaltung** informationsverarbeitender Prozesse beschäftigt (engl.: computer science).

Algorithmen und Programme

Teilgebiete der Informatik:

- Theoretische Informatik (Automaten, formale Sprachen, Komplexitätstheorie,...)
- Programmiersprachen (Semantik, Verifikation, neue Sprachkonzepte,...)
- Betriebssysteme und Rechnernetze (Verwaltung und Organisation / Verteilung von Ressourcen)
- Informationssysteme (Datenbanken,...)
- Computergraphik (Visualisierung, Modellierung komplexer Zusammenhänge → *Virtual Reality*)
- Robotik und Prozeßinformatik (Systeme zur gezielten Veränderung von „Weltzuständen“, auch: Sensoren → *Computer Vision*,...)
- Künstliche Intelligenz (KI, AI, Nachbildung kognitiver Prozesse)
- Rechnerarchitekturen (Rechner, Entwurf von Hardware)
- Wissenschaftliches Rechnen (Lösungsverfahren hochkomplexer Probleme)

2. Algorithmische Grundkonzepte

2.1 *Eigenschaften von Algorithmen*

„Muß - Eigenschaft“:

- **Finitheit** der Beschreibung: ein Algorithmus muß sich durch endlichen Text über einem endlichen, festen Alphabet darstellen lassen.

„Kann - Eigenschaft“:

- **Terminiertheit**: ein Algorithmus hält für jede Wahl von gültigen Eingabewerten nach endlich vielen Schritten an.

Beispiele:

- **Multiplikation** zweier Zahlen → terminiert
- **Division** → nicht immer terminiert ($1/3 = 0,\overline{3} \rightarrow \infty$ - Wert)

- Berechnung von $e := \sum_{n=0}^{\infty} \frac{1}{n!}$ → nicht terminiert

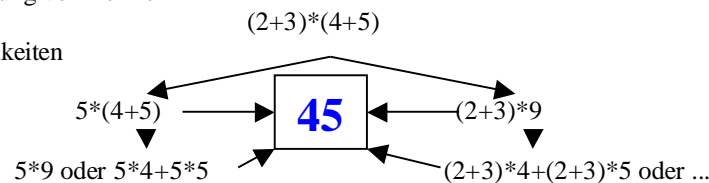
Algorithmen und Programme

⇒ Einführung von Abbruchkriterien!

- **„Determiniertheit“**: ein Algorithmus ist determiniert, wenn er bei gleicher Eingabe stets auf das gleiche Ergebnis führt
- **„Determinismus“**: ein Algorithmus ist deterministisch, wenn er bei gleicher Eingabe stets über die gleichen Zwischenschritte bzw. Zwischenergebnisse zum gleichen Ergebnis führt

Beispiel: „Auswertung von Termen“

→ mehrere Möglichkeiten



Aussagen:

- Berechnung hält immer an (**terminiert**)
- bei gleicher Eingabe wird immer gleiches Ergebnis geliefert (**determiniert**)
- die einzelnen Zwischenschritte auf dem Weg zum Ergebnis können verschieden sein (**nicht deterministisch**)

2.2 Daten, Operanden und Operationen

Daten

- Darstellung von Informationen im Rechner zur Eingabe, Verarbeitung und Ausgabe
- beschrieben durch Zahlen, Zeichen, Texte, Tabellen, Graphen, Bilder, ...
- rechnerinterne Darstellung

Datentyp

- Zusammenfassung von Wertebereichen und Operationen zu einer Einheit
- Standarddatentypen: **int, float, char**
- ein Algorithmus läßt sich als schrittweises Anwenden von Operationen auf Objekte bestimmter Datentypen (= Operanden) auffassen

Algorithmen und Programme

- Operanden können Konstante, Variable oder Ausdrücke sein
- Ausdrücke (Terme) entstehen, indem Operanden mit Operatoren verknüpft werden („rekursive Definition“!)
- Datentypen legen Wertemenge fest, aus der die Operanden Werte aufnehmen können

Standarddatentypen (in den meisten Programmiersprachen vorgegeben)

- Integer (**in C**: short int, int, long int) → ganzzahlige Werte
Wertemenge: $= \{-\infty, \dots, -1, 0, 1, \dots, +\infty\}$
(im realen Rechner endliche Teilmenge)

Operatoren: +, -, *, /, <, >, ...


↳ Problem bei Division: Ergebnis wird ganzzahlig abgerundet
(Bsp.: $23/9 \Rightarrow 2$, Rest wird durch sog. „Modulo“ – Operator „%“ bestimmt: $23\%9 \Rightarrow 5$)

in C: Gleichheit „==“, Ungleichheit „!=“

- Real (**in C**: float, double, long double) → Dezimalzahlen (1.7, 42E-1, -4.7E2)
Wertemenge: (realer Rechner: endliche Teilmengen)

Operatoren: +, -, *, /, ...

Gleichheitsoperatoren (==, !=, <, >, ...)

↳ problematisch: z.B.: $0.0 == 200/3.0 - 200(1/3.0)$

 muß als Rechnerergebnis nicht immer korrekt sein

- Character (char)
Wertemenge z.B.: {„a“, „b“, ..., „A“, „B“, ..., „-“, ..., „2“, ..., „#“, ...}

Operatoren: ==, <, > (lexikalische Ordnung)

↳ ASCII

Algorithmen und Programme

- String (in C kein Standarddatentyp, statt dessen z.B.: char[10])
Wertemenge: Zeichenketten, z.B.: „Hallo“, „Name_100“
- Boolean (bool, in C kein Standarddatentyp, statt dessen: int)
Wertemenge: {true, false} oder {1, 0}

Operatoren: not, and, or, ...

↙ (einstelliger Operator)

↘ (zweistelliger Operator)

Beispiel für
not:

bool_e	bool_a
true	false
false	true

in C z.B.: `bool_x != (3<2); /* true */`

Beispiel für
and:

bool_e1	bool_e2	bool_a
true	true	true
true	false	false
false	true	false
false	false	false

in C z.B.:
`bool_a = bool_e1 && bool_e2`

Beispiel für
or:

bool_e1	bool_e2	bool_a
true	true	true
true	false	true
false	true	true
false	false	false

in C z.B.:
`bool_a = bool_e1 || bool_e2`

Algorithmen und Programme

- ⇒ Die Reihenfolge der Auswertung von Ausdrücken ergibt sich durch Klammerung und Vorrangregeln:
Punkt vor Strich, z.B.: $(a*b + c*d)/e$

2.3 Imperative Algorithmen

- Basis für imperative Programmiersprachen wie **C**, **JAVA**, **PASCAL**, **Modula**, **Basic**, ...
- „bekannteste, häufigste“ Art, Algorithmen zu formulieren
- Alternativen: Applikative / deduktive Algorithmen / Programmiersprachen wie *Lisp* (KI), *Prolog*, ...
- **Anweisungen** (Befehle) verändern **Variablen** und damit den **Zustand** von Programmen
- Variablen bestehen aus einem **Namen** (Referenz auf Speicherplatz) und einem **veränderlichen Wert** (Inhalt des referenzierten Speicherplatzes)
- Der Wert einer Variablen wird durch eine Wertzuweisung verändert
z.B.: $x = t$ oder $y = f(x_1, \dots, x_m)$

Grundform imperativer Algorithmen

C – Syntax:

```

1      int [Funktion](int [eingabe1], int [eingabe2]); /* Funktionsname,
2      Eingabewerte */
3      {
4      int x, y, [ergebnis]; /* Variablendeklaration*/
5      [Anweisungen] α;
6      return([ergebnis]); /* Rückgabe des Ergebnisses*/
7      }
```

Mit folgenden Formen lassen sich mit den **[Anweisungen] α** beliebige Algorithmen formulieren:

- ➡ Folge (Sequenz: $\alpha_1; \alpha_2; \alpha_3; \dots$)

Algorithmen und Programme

➡ **Auswahl** bzw. bedingte Anweisung (Selektion):

```
if (B)  $\alpha_1$ ; else  $\alpha_2$ 
```

kann (je nach Bedarf) auch wegfallen – **B**: boolescher Ausdruck

➡ **Wiederholungsanweisung** (Schleife)

```
while (B)  $\alpha$ ;
```

Mehrere Anweisungen lassen sich dabei zu Blöcken zusammenfassen:

(in **C**: { . })

z.B.:

```
1 if (B) {
2      $\alpha_1$ ;  $\alpha_2$ ;  $\alpha_3$ ;
3 }
```

Programmbeispiel:

Fibonacci – Zahlenfolge:

1, 1, 2, 3, 5, 8, 13, 21, ...
0-te 1-te 2-te n-te Fibonacci-Zahl

($a_n := a_{n-1} + a_{n-2}$: mathematische Notation)

```
1 int fib(int) /* Berechnung der n-ten Fibonacci – Zahl */
2 {
3     int a, b, c; /* Variablendeklaration */
4     a = 1;
5     b = 1;
6
7     while (n>0) {
8         c = a + b;
9         a = b;
10        b = c;
11        n = n - 1;
12    }
13    return(a); /* Rückgabe des Ergebnisses */
14 }
```

Algorithmen und Programme

Beispiel zum Programm: Eingabe = 4

n	a	b	c	
4	1	1	⊥	Anfangszustand; (⊥ = undefiniert)
3	1	2	(1+1)= 2	Zustand nach 1. Schleifendurchlauf
2	2	3	3	Zustand nach 2. Schleifendurchlauf
1	3	5	5	Zustand nach 3. Schleifendurchlauf
0	5	8	8	Endzustand

↓
Ausgabe

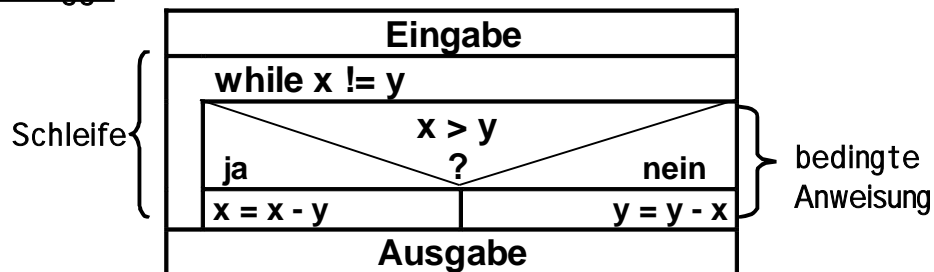
2.4 Strukturierung von Algorithmen / Programmen

Systematischer, übersichtlicher Aufbau, insbesondere bei großen Programmen sehr wichtig.

Zum Entwurf und zur Darstellung von Algorithmen:

Struktogramme nach Nassi - Schneidermann

Beispiel: ggT



Alternativen: Ablauf- / Flußdiagramme

Nachteil: unübersichtliche Sprünge

Algorithmen und Programme

Konzepte zur strukturierten Formulierung imperativer Algorithmen:

- **Blockkonzept:**
Zusammenfassen von Anweisungen in: { . }
- **Prozedurkonzept:**
Unterprogramme für funktionale oder sich häufig wiederholende Einheiten
- **Modulkonzept:**
Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren zu Untereinheiten eines Programmes, z.B.:

Definitionsmodul
Implementierungsmodul

nach außen sichtbar für andere Programmteile verborgen



in objektorientierter Programmierung (OOP, s.:6.3): „Klassen“, „Datenkapselung“ mit Schnittstelle für öffentlichen Zugriff

2.5 Prozeduren (in C: Funktionen)

Zusammenfassung von Anweisungsfolgen zu einer Anweisung

Beispiel:

Unterprogramm (Prozedur)

← greift zurück auf

Hauptprogramm

```

1   int berechne(int x, int y)
2   {
3       int z1,z2; /* Hilfsvariablen*/
4   x z1=x*x;
5   y z2=y*y;
6       return(z1+z2);
7   }           x y
    
```

```

1   main()
2   {
3       int a,b;
4   int x=5, y=3;
5       a=berechne(5,3);
6                       x,y
7       b=berechne(7,10);
8       /* a = 34, b = 149 */
9       /* x = 5, y = 3 */
10  }
    
```

— , abc... = ersetze / füge hinzu

Algorithmen und Programme

Ablauf (des Beispiels):

- 1) Aufruf der Prozedur: Aktuelle Parameter werden an lokale Variablen im Prozedurkopf übergeben
- 2) Ausführung des Prozedurrumpfes
- 3) ggf. Rückgabe des Ergebnisses mit *return* – Anweisung
 - lokale Variablen : werden **in** der Prozedur deklariert
 - globale Variablen: werden **außerhalb** der Prozedur deklariert

Parameterübergabe durch

- Wertaufruf (nur Eingabe, s.o.; Übergabeparameter sind lokale Größen; sie werden im Prozedurrumpf kopiert)
- Referenzaufruf (Prozedur hat Zugriff auf Größen außerhalb)

in C: mit Zeigern

➤ Zeiger

sind Adressen von Variablen, symbolischen Konstanten oder Funktionen. Variablen, deren Werte Zeiger sind, heißen Zeigervariablen.

Beispiel:

```
int *p, x; x=10;
```

Adressen (z.B.)	Speicherinhalt	Bedeutung des Inhalts
...	...	
50 000		}
50 001		
50 002		
...		
58 000	0 0 0 0 1 0 1 0	}
58 001		
...		

58 000

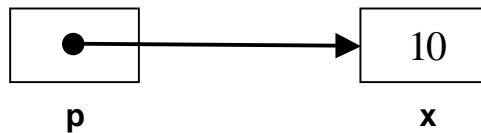
0 0 0 0 1 0 1 0

Algorithmen und Programme

Durch

p=&x; /* Adreßzuweisung*/

wird dem Zeiger p die Adresse von x zugewiesen, d.h. p „zeigt“ auf x, bzw. graphische Notation:



↳ „&“ bezeichnet den **Adreßoperator**; auf Variable beliebigen Typs angewendet, liefert er die Adresse dieser Variablen.

Mit **Dereferenzier-** bzw. **Sternoperator** „*“ ist der Zugriff auf den Wert der mit Zeigern „referenzierten“ Variablen möglich.

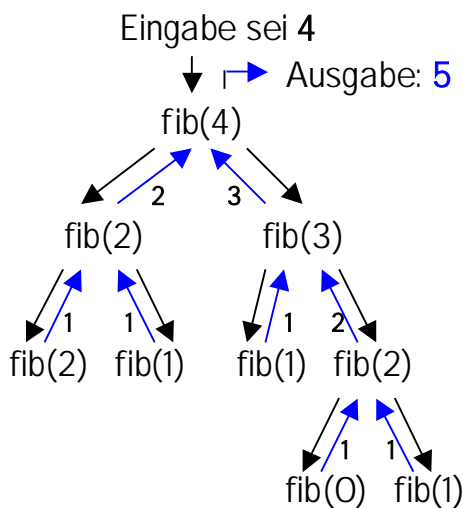
Im Beispiel führt
auf das gleiche Ergebnis wie

int j=*p + 10	
int j= x + 10	(=20)

Prozeduren können **sich selbst** direkt oder indirekt aufrufen → „**Rekursion**“

Beispiel: Fibonacci – Zahlenberechnung

Ablauf:



```

1  int fib(int x)
2  {
3    if (x<2) return(1)
4    else return (fib(x - 1) + fib(x - 2));
5    /* Rekursion */
6  }
```

Algorithmen und Programme

↳ Beispiel **ineffizienter** als iterative Lösung (mit *while* – Anweisung), da identische Berechnungen mehrfach wiederholt werden.

➡ Oft lassen sich Algorithmen durch rekursive Techniken einfacher und anschaulicher formulieren!

2.6 Statische (implizite) Datenstrukturen

- Aufbau von Wertebereichen aus elementaren Datentypen mit Hilfe von Konstruktoren
- Strukturelle Beziehungen zwischen Datenelementen *implizit* durch Deklaration im Programm gegeben
- **statisch** $\hat{=}$ konstante Größe
- **Array**: Vektor bzw. eindimensionales Feld aus Elementen desselben Datentyps

in C z.B.:

```
int a[100]; /* 100 Integerwerte – Index von 0 bis 99 */
```

Integerwert	10	-7	37	1	73
Index	0	1	2	98	99

↳ Zugriff (z.B.):

```

1-n   ...
n+1   i = 0;
n+2   while (i<100)
n+3   {
n+4     a[i]=...;
n+5     i = i + 1;
n+6   }
n+k   ...
(k=7-...)
```

Algorithmen und Programme

String (Zeichenkette) als Array:

```
char name[7] = "Frieda";
```

- Zweidimensionales Array: Matrix

Deklaration z.B.: float a[M][N];

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

in C: $a_{11} \hat{=} [0][0]$
 , ... ,
 $a_{mn} \hat{=} [M-1][N-1]$

Im Rechner werden Elemente zeilenweise in aufeinanderfolgenden Speichereinheiten abgelegt:

Sei **b** Basisadresse des Arrays (des ersten Elements), so ist **a[i][j]** unter der Adresse $\alpha(i, j) = b + n(i - 1) + j - 1$ gespeichert. (n: Spaltenanzahl)

- ➡ Arrays mit **mehr als zwei** Dimensionen entsprechend, z.B.:

```
int [m1][M2]...[Mk];
```

- Struktur (structure, record):
 Zusammenfassung von Elementen unterschiedlicher Typen

C – Beispiel:

```

1 struct person           k+2-m ...
2 {                       m+1  person a;
3   char vorname[20];     m+2-n ...
4   char nachname[30];    n+1   a.vorname[0]='E' /* 1. Buchstabe des Vornamens */
5   int alter;           n+2   a.alter=29;
6   int groesse;
7-k   ...
k+1 };
    
```

Algorithmen und Programme

Adressen	Speicherinhalt	Bedeutung	
30 001 Adresse zur variablen Struktur 'person'	'E'	Vorname 20 Bytes	Für Struktur 'person' reservierter Speicherbedarf
	'R'		
	'N'		
	...		
30 021	'M'	Nachname 30 Bytes	
	'U'		
	'E'		
	...		
30 051	00011101	Alter 4 Bytes	
	00000000		
	00000000		
	00000000		
30 055	-----	Größe 4 Bytes	

bei mehreren Personen: **Array**

↳ Personentabelle

```

1-m ...
m+1 person.a[100];
m+2-n ...
n+1 a[37].vorname[0]='E'
n+2 a[37].alter=29;
    
```

Geeignete Organisation / Struktur von Daten \Rightarrow **effiziente Algorithmen**

Algorithmen und Programme

3. Komplexität von Algorithmen

3.1 Einführung

<pre> 1 prog_1(int n) 2 { 3 n = n*4 4 while(0<n) 5 { 6 tue_etwas(...); 7 n = n - 1; 8 } 9 } 10 /* Aufwand: n*4 */ </pre>	<pre> 1 prog_2(int n) 2 { 3 int i, j; 4 i = n; 5 while(0<i) 6 { 7 j = n; 8 while(0<j) 9 { 10 tue_etwas(...); 11 j = j - 1; 12 } 13 i = i - 1; 14 } 15 } 16 /* Aufwand: n² */ </pre>
--	--

Aufwandsvergleich:

n	1	2	3	4	5	6	7
prog_1	4	8	12	16	20	24	...
prog_2	1	4	9	16	25	36	...

Aufwand, um Algorithmen auszuführen, ist abhängig von

- Größe „n“ der Eingabe, Anzahl der Daten;
- der **Komplexität** des Algorithmus‘.



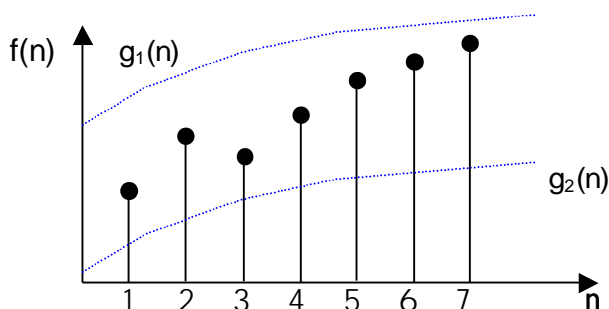
Unterscheidung zwischen:

- a) Zeitkomplexität (Laufzeit des Algorithmus‘)
- b) Speicherkomplexität (benötigte Größe des Speichers [z.B.: Arrays])

Algorithmen und Programme

3.2 Asymptotische Analyse / „Groß – O – Notation“

Idee: „Wachstumsverhalten“ möglichst allgemein für große „n“ beschreiben
(Konstante Faktoren; Summanden werden nicht berücksichtigt)



Asymptotische Analyse weist einer Funktion $g: \mathbb{N} \rightarrow \mathbb{N}$ eine Menge von Funktionen zu, die in einer bestimmten Wachstumsbeziehung zu g stehen. Hier z.B.:

g_1 ist obere Schranke von f ;



dies läßt sich mit **Groß – O – Notation** ausdrücken: $f \in O(g_1)$

➡ Vgl. Beispiel aus 3.1:

$f(n)$ = Anzahl der *tue_etwas(...)* – Aufrufe

d.h.: prog_1: $f(n) \in O(n)$

prog_2: $f(n) \in O(n^2)$

formale Definition:

„ $O(g)$ ist die Menge aller Funktionen f , die durch g asymptotisch beschränkt sind, multipliziert mit einem konstanten Faktor.“

D.h.:

$$O(g) = \{ f: \mathbb{N}_0 \rightarrow \mathbb{N}_0 \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}_0, \forall n > n_0: f(n) \leq c \cdot g(n) \}$$

Algorithmen und Programme

Beispiele:

- | | |
|----------------------------------|---------------------------------------|
| $O(1)$ = konstanter Aufwand | $O(\log n)$ = logarithmischer Aufwand |
| $O(n^k)$ = polynomialer Aufwand | $O(n)$ = linearer Aufwand |
| $O(n^2)$ = quadratischer Aufwand | |

- $O(2^k)$ = superpolynomialer (exponentieller) Aufwand
- $O(n!)$ = superpolynomialer (exponentieller) Aufwand
- $O(n^n)$ = superpolynomialer (exponentieller) Aufwand

- Exponentielle Probleme sind i.a. nicht lösbar!
- Reduktion der Komplexität durch Entwicklung effizienterer Algorithmen (z.B. Sortierverfahren, Fouriertransformationen: $O(n^2)$)
⇨ schnelle Fouriertransformation (FFT): $O(\log n)$)

Rechenregeln:

- Linearität:** $O(c_1 \cdot g(n) + c_2) = O(g(n))$
- Distributivität:** $O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$
- Multiplikation:** $O(g_1(n)) \cdot O(g_2(n)) = O(g_1(n) \cdot g_2(n))$
- außerdem:** $O(O(g(n))) = O(g(n))$
- $O(g_1(n) \cdot g_2(n)) = g_1(n) \cdot O(g_2(n))$

Merke:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \Rightarrow f(n) \in O(n^k)$$

höchstes Polynom (niederwertigere: vernachlässigbar)

Algorithmen und Programme

3.3 Beispiele

1) Berechnung von x^n (Version 1)

```

1  int potenz_1(int x, int n)
2  {
3      int y;
4      y = 1;
5      while(n>0)
6      {
7          y = y*x;
8          n = n-1
9      }
10     return(y);
11 }

```

Schleife wird **n-mal** durchlaufen
 $\Rightarrow f(n) = n \Rightarrow f(n) \in O(n)$

2) Berechnung von x^n (Version 2 – schneller)

Lösung des Problems durch fortlaufende Halbierung des Exponenten:

z.B.: $x^8 = x^4 \cdot x^4 \rightarrow x^4 = x^2 \cdot x^2$

```

1  int potenz_2(int x, int n)
2  {
3      int y;
4      if(n==0) return(1);      /* x0 = 1 */
5      if(n==1) return(x);     /* x1 = x */
6      y = potenz_2(x, n/2);    /* für n>=2 */
7      y = y*y;
8      if(n%2==1) y=y*y;       /* n ist ungerade */
9      return(y);
10 }

```

Wie oft läßt sich **n** halbieren?

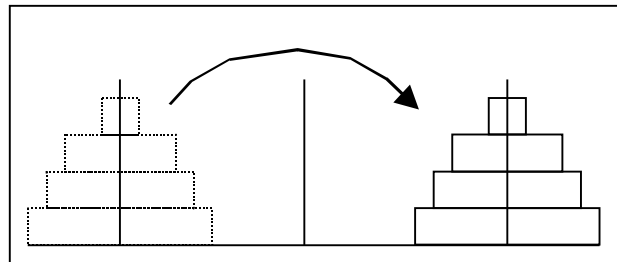
\rightarrow Logarithmus **n** (zur Basis 2)

$\rightarrow f(n) \sim \log_2 n \rightarrow f(n) \in O(\log n)$

Algorithmen und Programme

3) „Türme von Hanoi“

n	2	3	4	...
Züge	3	7	15	...



Umsetzen des Scheibenhaufens durch Versetzen einzelner Scheiben so, daß nie eine größere Scheibe auf einer kleineren zum Liegen kommt.

—→ Aufwand —→ s.: Tabelle

D.h.: bei **n** Scheiben ergeben sich $2^n - 1$ Bewegungen → $O(2^n)$ -Problem

Vgl. **64-Scheiben-Fall**:

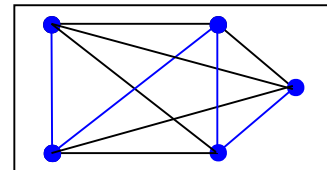
Bei 10^6 Bewegungen pro Sekunde sind ca. **580.000 Jahre** notwendig, um das Problem zu lösen!

4) „Travelling – Salesman – Problem“

gegeben: ein Graph mit

- **Knoten:** n Städte
- **Kanten:** Verbindungen zwischen den Städten und Entfernungsangaben

z.B.:



gesucht:

kürzeste Route, bei der alle Städte **genau einmal** besucht werden

Zahl der möglichen Routen:

- n mögliche Startwerte
 - n-1 mögliche Anfangswerte
 - n-2 mögliche zweite Wege
 - ... u.S.W.
- ↓
- $n(n-1)(n-2) \cdot \dots \cdot 2 \cdot 1 = n!$ - Möglichkeiten (entspricht extremem Aufwand)

Algorithmen und Programme

Zum Vergleich: 25 Städte $\rightarrow f(25) = 25!$

Bei Überprüfung von 10^6 Routen pro Sekunde würde die Gesamtrechenzeit $490 \cdot 10^9$ Jahre betragen!

Viele Anwendungen für dieses Problem:

- Transport- und Kommunikationsprobleme
- Entwurf von Schaltungen (Leiterbahnen)
- Programmierung von Robotern, Plotter, etc.

Meist sind keine optimalen Lösungen möglich!

Statt dessen:

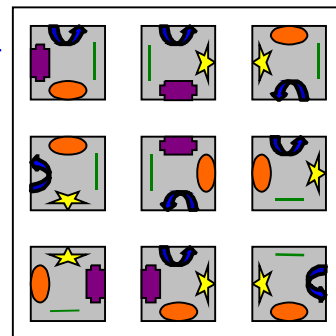
Heuristiken (Vereinfachungen, einschränkende Annahmen, u.s.w.)

suboptimale Lösungen

5) „Schildkröten-/ Lioriotpuzzle“

$m \times m = n$ Karten sind so zu legen, daß aneinandergrenzende Karten zueinander passen.

- $m = 3:$
- Legen der 1. Karte: **$4n$** Möglichkeiten
 (n Positionen und 4 Drehungen)
- Legen der 2. Karte: **$4(n-1)$** Möglichkeiten
- ...
- Legen der n -ten Karte: **4** Möglichkeiten



insgesamt: $(4^n \cdot n!)$ -Möglichkeiten

z.B.: 9 Karten $\rightarrow 95 \cdot 10^9$ Möglichkeiten

☞ „Blindes“ Ausprobieren ist also nicht praktikabel!

Komplexität: $f(n) \in O(4^n \cdot n!)$

Algorithmen und Programme

3.4 Weitere Komplexitätsmaße

$f \in \Omega(g)$: untere Grenze \longrightarrow „best case“
 \longrightarrow „f wächst asymptotisch mindestens wie g“

$$\Omega(g) = \{ f: \mathbb{N}_0 \rightarrow \mathbb{N}_0 \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}_0, \forall n \geq n_0: f(n) \geq c \cdot g(n) \}$$

$f \in \Theta(g)$: \longrightarrow „f wächst gleich schnell wie g“

$$\Theta(g) = O(g) \cap \Omega(g)$$

„Worst case“ und „best case“ unterscheiden sich nur durch einen **konstanten Faktor**:

$f \in \sim(g)$: \longrightarrow „f ist gleich g für $n \rightarrow \infty$ “

$$\sim(g) = \{ f: \mathbb{N}_0 \rightarrow \mathbb{N}_0 \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1 \}$$

\longrightarrow Vorfaktoren werden berücksichtigt!

4. Sortierverfahren

4.1 Einführung

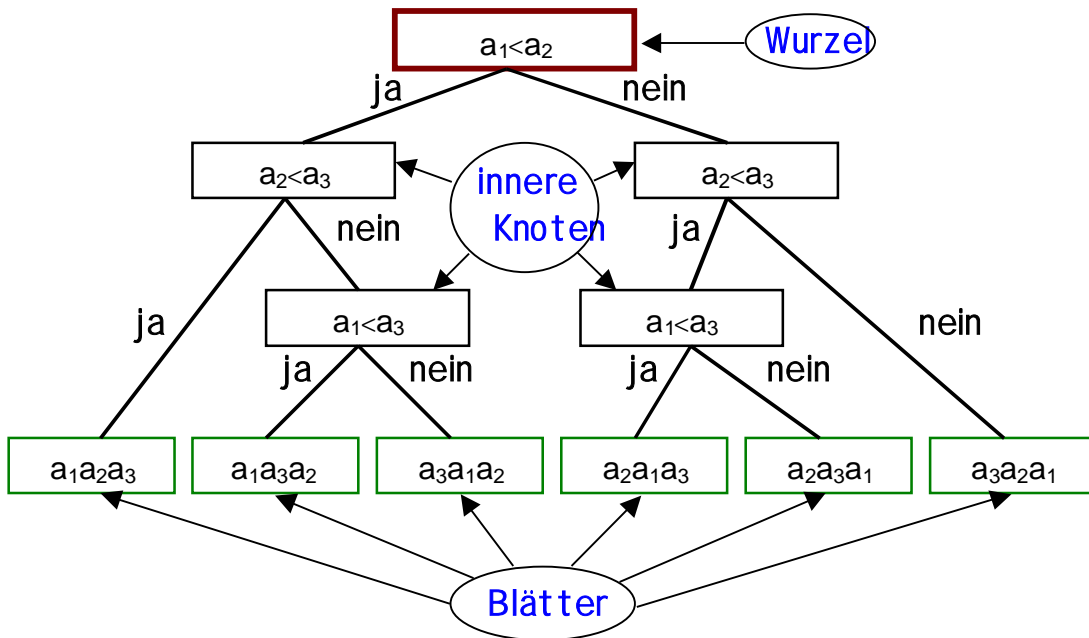
Auf Datenelementen ist oft eine **Ordnung** definiert (z.B. *lexikalisch* oder „ \leq “)
 Daher: Sortierverfahren von großer Bedeutung

Problem: Umordnen einer Folge von Objekten a_1, \dots, a_n in die Folge $a_{i_1}, a_{i_2}, \dots, a_{i_n}$
 so, daß eine Ordnungsrelation, z.B. $a_{ij} \leq a_{ij+1}$ mit $j = 1, \dots, n-1$ gilt.

Algorithmen und Programme

Beispiel: Sortieren von Folgen a_1, a_2, a_3

Lösung: schrittweiser Vergleich zweier Elemente; darstellbar als *binärer Entscheidungsbaum* ($h=4$)
 (binärer Baum: jeder Knoten hat maximal zwei Nachfolgeknoten)

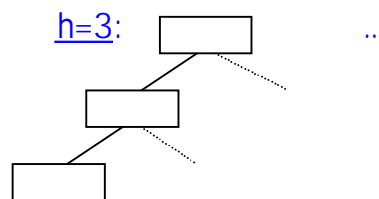
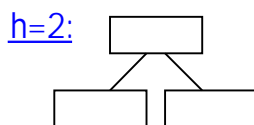


Bei n Elementen gibt es $(n!)$ -verschiedene Reihenfolgen (Permutationen), d.h. der Entscheidungsbaum hat $(n!)$ -Blätter.

Komplexität des Sortierens → Wie viele Vergleiche sind bei n Elementen maximal notwendig?



Höhe „ h “ eines Baumes: maximale Anzahl von Knoten (inklusive Wurzel und Blättern) für einen Pfad von der Wurzel zu einem Blatt.



...

Algorithmen und Programme

Entscheidungsbaum der Höhe h : maximal $V(n) = h-1$ Vergleiche

Maximale Anzahl der Blätter für Binärbaum: 2^{h-1}

Hieraus folgt: $2^{h-1} = 2^{V(n)} \geq n!$

$$V(n) \geq \log_2(n!) = \sum_{i=1}^n \log(i) \geq \sum_{i=\frac{n}{2}}^n \log(i) \geq \frac{n}{2} \log_2\left(\frac{n}{2}\right)$$

$$V(n) \in \Omega(n \log(n))$$

Untere Schranke für Sortierverfahren, d.h., Sortieren von n Objekten durch paarweisen Vergleich ist mindestens ein $O(n \log(n))$ -Problem!

4.2 Ein einfacher Sortieralgorithmus

gegeben sei: Integer Array $a[n+1]$ → als Reserve

Die zu sortierenden n Zahlen belegen die Feldelemente von $a[1]$ bis $a[n]$.
 Feldelement $a[0]$ ist mit **kleinstmöglicher** Integer – Zahl `MIN_INT` vorbelegt.

Idee: Sortiere Array abschnittsweise von links nach rechts.

index	0	1	2	...	$i-1$	i	$i+1$...	n
$a[\text{index}]$	$-\infty$	5	7	...	23	6	3	...	20

sortiertunsortiert

- im i -ten Schritt: die ersten $(i-1)$ -Elemente sind sortiert
- als nächstes wird i -tes Element an der richtigen Stelle der bereits sortierten Folge eingefügt

Algorithmen und Programme

Algorithmus:

```

1  sort (int n, int a[ ]) /* Array mit n zu sortierenden Zahlen wird per */
2  /* Referenz uebergeben*/
3  {
4  int i, j;
5  a[0] = MIN_INT; /* kleinste Integer – Zahl */
6  i = 2;
7  while(i<=n)
8  {
9  j = i;
10 while(a[j-1]>a[j])
11 {
12 a[j-1](:=) a[j]; /* Tausche */
13 j = j - 1;
14 }
15 i = i + 1;
16 }
17 }
    
```

„ := “ bezeichnet den Tauschoperator. In C so nicht vorhanden, kann aber leicht implementiert werden.



Ablauf – Beispiel:

0	1	2	3	4	5	6	
-∞	8	3	2	7	4	5	i=2, j=2
-∞	3	8	2	7	4	5	i=3, j=3
-∞	3	2	8	7	4	5	i=3, j=2
-∞	2	3	8	7	4	5	i=4, j=4
-∞	2	3	7	8	4	5	i=4, j=3
...							
usw.							

} „Bubble - Sort-Algorithmus“

Algorithmen und Programme

(Aufwands-) Analyse:

V: Anzahl der Vergleichsoperationen
 t: Anzahl der Vertauschoperationen

- **besten Fall:** (Folge liegt bereits sortiert vor)

$$V_{\text{best}} = n - 1$$

$$t_{\text{best}} = 0$$

- **mittlerer Fall:**

$$V_{\text{mittel}} = \underbrace{n - 1}_{1x \text{ while - Bedingung}} + \underbrace{inv_{\text{mittel}}}_{\text{mittlere Anzahl der Inversionen einer Permutation}} = (n^2 + 3n - 4)/4$$

1x while - Bedingung mittlere Anzahl der Inversionen einer Permutation

Exkurs:

Sei $(a_i; 1 \leq i \leq n)$:

eine **Permutation** (a_i, a_j) mit $1 \leq i < j \leq n$ heißt **Inversion**, falls $a_i > a_j$.

Zu jeder Permutation $A: a_1 = x_1, a_2 = x_2, \dots, a_n = x_n$ gibt es auch eine umgekehrte Permutation $A': a_1 = x_n, a_2 = x_{n-1}, \dots, a_n = x_1$.

Ein Paar x_i, x_j ist in A oder A' in gewünschter Reihenfolge; in der jeweils anderen bilden sie eine Inversion. Alle Permutationen sind gleich wahrscheinlich.

Wahrscheinlichkeit, daß x_i, x_j Inversion in A ist, ist $1/2$.

Überprüfung:

	x_1	x_2	x_3	x_4	...	x_n
x_1	X					
x_2		X				
x_3			X			
x_4				X		
...					X	
x_n						X

→ Diagonale streichen

Da es insgesamt $(n^2 - n)/2$ Paare (x_i, x_j) mit $1 \leq i < j \leq n$ gibt, ist

$$inv_{\text{mittel}} = 1/2 [n(n - 1)/2].$$

$$t_{\text{mittel}} = inv_{\text{mittel}} = (n^2 - n)/4$$

Algorithmen und Programme

- schlimmster Fall: (monoton abnehmende Folge)
- $i = 2 \longrightarrow 2$ Vergleiche und 1 Tauschoperation
- $i = 3 \longrightarrow 3$ Vergleiche und 2 Tauschoperationen
-
- $i = n \longrightarrow n$ Vergleiche und $(n-1)$ Tauschoperationen



$$V_{\text{schlimm}} = 1 + 2 + 3 + \dots + n - 1 = [n(n+1)/2] - 1$$

$$= (n^2 - n - 2)/2$$

$$t_{\text{schlimm}} = 1 + 2 + 3 + \dots + (n - 1) = [n(n - 1)/2]$$



Komplexität im Mittel: $O(n^2)$ — (noch recht aufwendig; vgl.: $n \log(n)$)

4.3 Quicksort

- schnelles Sortierverfahren nach Hoare (1962)
- algorithmisches Prinzip: „**Divide and Conquer**“ (Teile und Herrsche)

Sei $A(D)$ die Ausführung eines Algorithmus' und D die Datenmenge.

Falls

- D klein und einfach: lösen durch einfache Operation A_0
- D groß und kompliziert: teile D in mehrere Datenmengen D_1, \dots, D_k derart, daß Lösung $A(D)$ aus Teillösungen $A(D_1), \dots, A(D_k)$ leicht zusammengesetzt werden kann.

Divide and conquer führt meist zu rekursiven Algorithmen:

Formel:

1	$A(D)$	6	‘teile D in D_1, \dots, D_k ‘
2	{	7	‘berechne $A(D_1), \dots, A(D_k)$ ‘ /* Rekursion */
3	if(D einfach) $A_0(D)$;	8	,setze $A(D)$ aus $A(D_1), \dots, A(D_k)$ zusammen‘
4	else	9	}
5	{	10	}

Algorithmen und Programme

Idee von Quicksort:

gegeben: Array $\text{int } a[\dots]$, das im Bereich von $a[l]$ bis $a[r]$ sortiert werden soll;
z.B.: $l = 0, r = n-1$

Ablauf:

- wähle beliebiges Element x aus dem Sortierbereich $a[l \dots r]$
- teile den Bereich in zwei Teilbereiche $a[l \dots k]$ und $a[k+1 \dots r]$ so auf, daß alle Elemente aus $a[l \dots k]$ kleiner als x sind und alle aus $a[k+1 \dots r]$ größer gleich x sind
- sortiere die beiden Teilbereiche rekursiv mit dem gleichen Verfahren

Algorithmus:

```

1  quicksort(int l, int r, int a[ ]) /* l und r: Grenzen des Sortierbereichs */
2  {
3      int i, j, x;
4      x = a[(l + r)/2]; /* allgemein: beliebiges Element aus a[l, ..., r] */
5      i = l; j = r; /* Versuche, von links und rechts kommend, Elemente zu */
6      /* vertauschen, falls sie nicht in der richtigen Reihenfolge sind */
7      while(i<=j)
8      {
9          while(a[i]<x) i = i + 1;
10         while(a[j]>x) j = j - 1;
11         if(i<=j)
12         {
13             a[i]:=a[j] /* tausche */
14             i = i + 1;
15             j = j - 1;
16         }
17     }
18     /* Rekursiver Aufruf für die beiden Teilbereiche */
19     if(l<j) quicksort(l, j, a);
20     if(i<r) quicksort(i, r, a);
21 }

```

Algorithmen und Programme

Beispielablauf:

0	1	2	3	4	5	6	7	$\rightarrow x = (0 + 7)/2 = 3._ \Rightarrow 23$
<u>32</u>	30	10	23	11	14	26	<u>13</u>	
13	<u>30</u>	10	23	11	<u>14</u>	<u>26</u>	32	
13	14	<u>10</u>	<u>23</u>	<u>11</u>	30	26	32	
<u>13</u>	14	10	<u>11</u>	<u>23</u>	30	26	32	„divide and conquer“
								$l_1=0, r_1=3, l_2=4, r_2=7$ $(0 + 3)/2 = 1._ \Rightarrow 14$
<u>13</u>	14	10	11	analog				
<u>13</u>	11	<u>10</u>	<u>14</u>	„divide and conquer“				
								$(0 + 2)/2 = 1 \Rightarrow 11$
<u>13</u>	11	10	14	analog				
10	<u>11</u>	13	14					
10	11	13	14	23	26	30	32	

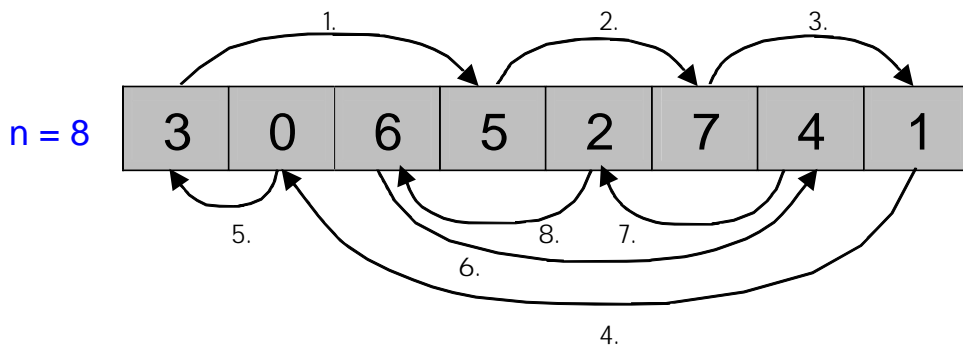
Komplexität: $O(n \log(n))$

Algorithmen und Programme

4.4 Sortieren (schneller als Quicksort?)

Beispiel 1:

Sortieren von Permutationen der ganzen Zahlen von $0 \dots n - 1$



D.h.: maximal n Tauschoperationen

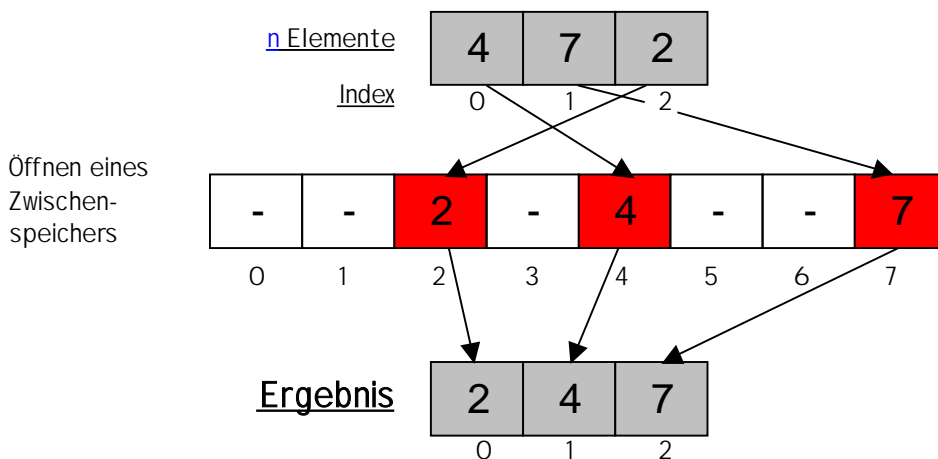
Komplexität: $O(n)$

vgl.: „Bucketsort“ (hier nicht beschrieben)

Beispiel 2:

Sortieren von n Elementen aus endlichem Wertebereich $W = [1, w]$ mit $n \ll w$; danach: Schließen von „Lücken“

z.B.: $w = 7$:



Komplexität: $O(w)$

Algorithmen und Programme

Kein Widerspruch zu den Überlegungen in [Abschnitt 4.1](#), da hier implizit mehrwertige Vergleiche durchgeführt werden!

5 Dynamische Datenstrukturen

Bisher:

statische Datenstrukturen haben konstante, zu Programmbeginn festgelegte Größe

dynamische Datenstrukturen:

Größe wird während des Programmlaufs festgelegt / verändert

5.1 Speicherverwaltung mit 'new' bzw. 'delete'

Heap (Halde): von Betriebssystem verwalteter großer Adreßbereich

- mit Operator '**new**' kann während der Laufzeit zusätzlicher Speicherbereich angefordert werden
- mit Operator '**delete**' wird Speicherplatz wieder freigegeben
- Zugriff auf Speicherplatz mit Zeigervariablen (s.: Abschnitt 2.5)

Beispiele:

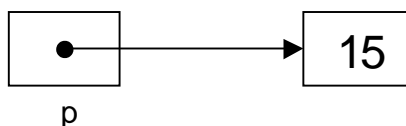
1)

```

1   int *p; // Zeiger auf int
2   p = new int; // int – Objekt erzeugen
3   *p = 15; // Wert zuweisen
4   cout <<*p <<endl; // '15' ausgeben
5   // '*': Inhalts- bzw. Dereferenzieroperator

```

C++ - Beispiel

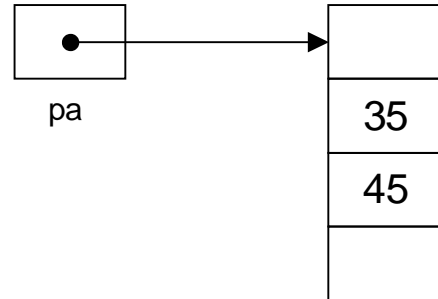


Algorithmen und Programme

2) Dynamisches Erzeugen eines Feldes

```

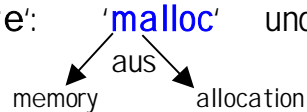
1   *pa = new int[4]; // Array von int – Zahlen
2   pa[1] = 35;
3   pa[2] = 45;
4   cout <<pa[1] <<endlich; //35
```



Wenn ein Zeiger inkrementiert / dekrementiert wird, zeigt er (in C++) nicht auf die nächste Speicheradresse, sondern auf die Adresse des nächsten Wertes; z.B. würde `pa + 2` auf den Wert 45 zeigen.

Im Gegensatz zu C wird in C++ der Typ eines Zeigers geprüft.

In C analog zu 'new' und 'delete': 'malloc' und 'free' mit a = Zeigervariable



5.2 Lineare Listen

- lineare Listen sind verkettete Folgen von Datenelementen
- jedes Element enthält neben den eigentlichen Daten einen Zeiger auf das nächste Datenelement

(Fortsetzung: Seite 37)

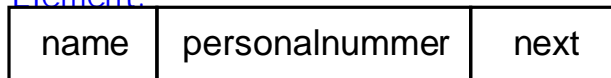
Algorithmen und Programme

Beispiel:

```

1 struct element
2 {
3     char name[20];
4     int personalnummer;
5     element *next;
6 };
    
```

Element:



Liste:



Erzeugen und Aneinanderhängen zweier Elemente:

```

1 element *anfang;
2 element *zweites;
3 anfang = new element;
4 anfang --> name[0] = 'A';
5 - k .....
k+1 anfang --> personalnummer = 17; // in C:
k+2-m ..... // (*anfang).personalnummer
m+1 zweites = new element;
m+2 anfang --> next = zweites; // Zeiger vom ersten Element zum zweiten
    
```

Algorithmen und Programme

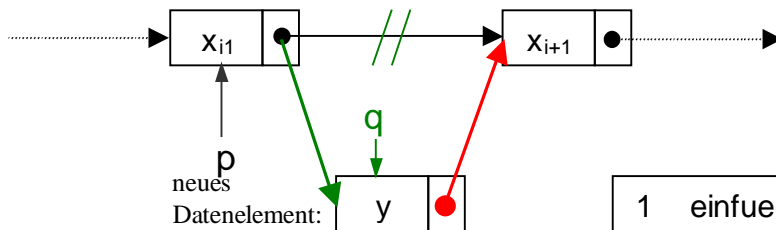
- dynamisch angeforderter Speicher -				
Name	Adresse	Speicher	Datentyp	Bedeutung
	0 1 . . .			
anfang		124456	struct element*	Zeiger auf 1. Element der Liste
(*anfang)	124456	Peter Meier 230009 126500	.name .nummer .next } struct element	1. Element
	126500	Gabi Schmidt 231127 126700	.name .nummer .next } struct element	2. Element
	126700	Ute Müller 230120 0	.name .nummer .next } struct element	3. Element
	Nullzeiger			
 4 000 000			

- Größe und Struktur einer Liste können **während** des Programmlaufs verändert werden (Einfügen / Löschen von Elementen)
- Speicherplatz wird **dynamisch** zugeteilt
- die einzelnen Listenelemente können sich an **beliebigen** Stellen im Speicher befinden
- Zugriff auf Speicher erfolgt über Zeiger (→ - **Operator**, bzw. bei Strukturen: -->)
- **NULL - Zeiger** (z.B.: p = 0 bzw. p = NULL) markiert, daß Zeiger auf keine gültige Speicherstelle zeigt (z.B., wenn Element keinen „Listennachfolger“ hat)

Algorithmen und Programme

Operationen auf Listen:

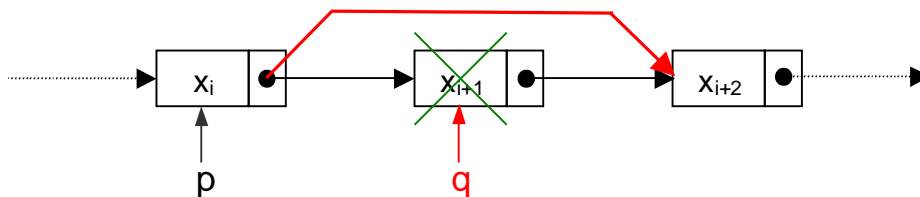
- füge neues Element hinter dem Element ein, auf das Zeiger „p“ zeigt



```

1  einfuegen (element *p)
2  {
3    element *p;
4    q = new element; // Speicher bereit-
5    q --> next = p --> next; // stellen
6    p --> next = q;
7    // ggf.: Daten eintragen
8  }
    
```

- Löschen eines Elementes hinter Element p



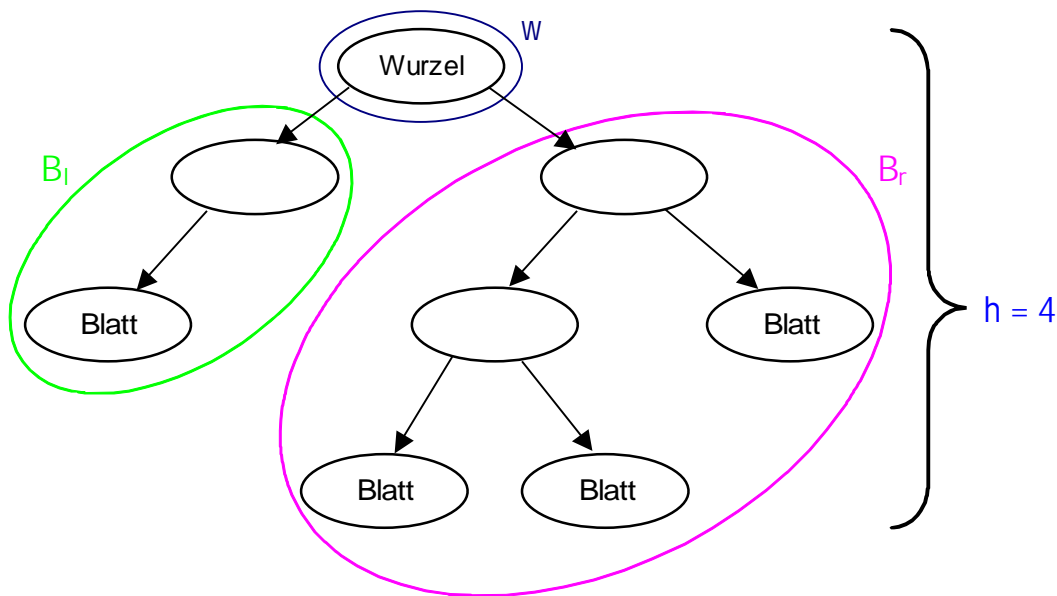
```

1  loeschen (element *q)
2  {
3    element *q; // zu loeschendes Objekt
4    q = p --> next;
5    if(q!=0)
6    {
7      p --> next = q --> next; // Zeiger aendern
8      delete q; // Speicher des gelöschten Elementes freigeben
9    }
10 }
    
```

Algorithmen und Programme

5.3 Bäume

vgl.: 4.1 (S. 26)



- Elemente des Baumes (Knoten) sind durch gerichtete Kanten verbunden (keine Zyklen!)
- alle Elemente außer *Wurzel* haben einen Vorgänger
- Elemente **ohne** Nachfolger heißen *Blätter*
- **Höhe (h)** des Baumes ist die maximale Knotenanzahl eines Weges **von der Wurzel zu einem Blatt**
- **Binärer Baum:** jedes Element hat maximal zwei direkte Nachfolger

Definition:
 „Ein binärer Baum ist eine Menge von Elementen.
 Es gilt:
 → B besteht aus Wurzel w, einem linken Teilbaum $B_l(w)$ und einem rechten Teilbaum $B_r(w)$
 → o d e r: B ist leer ()“

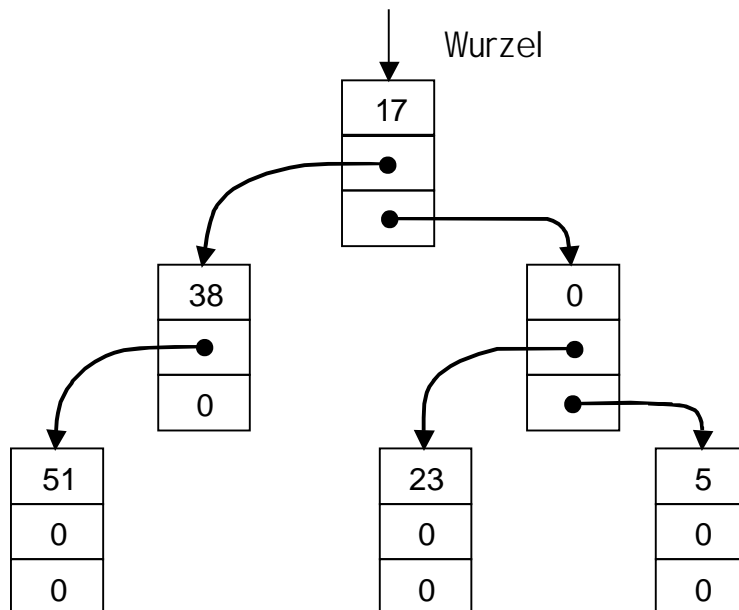
Algorithmen und Programme

Datenstruktur: ähnlich wie Liste, jedoch kann jedes Element zwei Nachfolger haben

```

1  struct knoten
2  {
3      int x;          // zu speicherndes Element
4      knoten *links; // Zeiger auf linkes bzw.
5      knoten *rechts; // rechtes Folgeelement
6  }
7  knoten *wurzel;    // Zeigervariable auf Wurzel
8  wurzel = new knoten; // Speicher für Wurzelelement anfordern und Zeiger-
9                      // variable mit Adresse und Wurzel belegen

```



Baumtraversierung:

Vollständiges Durchlaufen eines Baumes, um, z.B., auf allen Elementen eine Operation anzuwenden.

Man unterscheidet **drei Arten** des Durchlaufs:

preorder – inorder – postorder

Algorithmen und Programme

Beispiel:

<pre> graph TD 1 --- 2 1 --- 3 2 --- 4 2 --- 5 3 --- 6 3 --- 7 4 --- 8 </pre>	<p>→ preorder: w, B_l, B_r 1, 2, 4, 8, 5, 3, 6, 7 (präfix)</p> <hr/> <p>→ inorder: B_l, w, B_r 8, 4, 2, 5, 1, 6, 3, 7 (infix)</p> <hr/> <p>→ postorder: B_l, B_r, w 8, 4, 5, 2, 6, 7, 3, 1 (suffix)</p>
---	---



Algorithmus zur Traversierung in **preorder** oder **inorder** oder **postorder**:

```

traverse (knoten *p)
{
  if(p != 0)
  {
    Operation auf p --> x;
    traverse (p --> links);
    Operation auf p --> x;
    traverse (p --> rechts);
    Operation auf p --> x;
  }
}
    
```

in **preorder** ←

in **inorder** ←

in **postorder** ←

Algorithmen und Programme

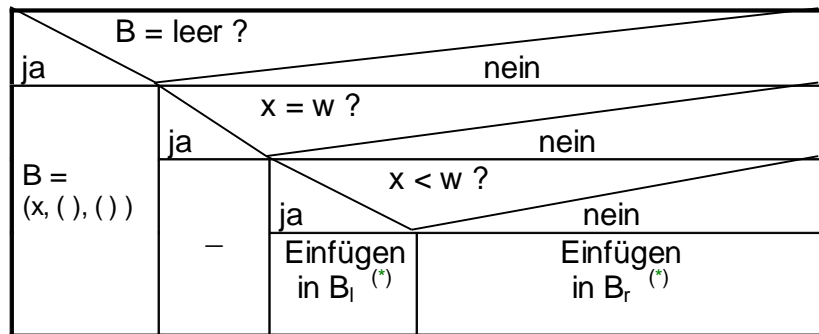
5.4 Binäre Suchbäume

Ein binärer Baum heißt *binärer Suchbaum*, wenn gilt:

- B ist leer
- o d e r:
- $\forall x \in B_l(w)$ mit $x < w$ und $\forall x \in B_r(w)$ mit $x > w$ und $B_l(w), B_r(w)$ sind binäre Suchbäume.

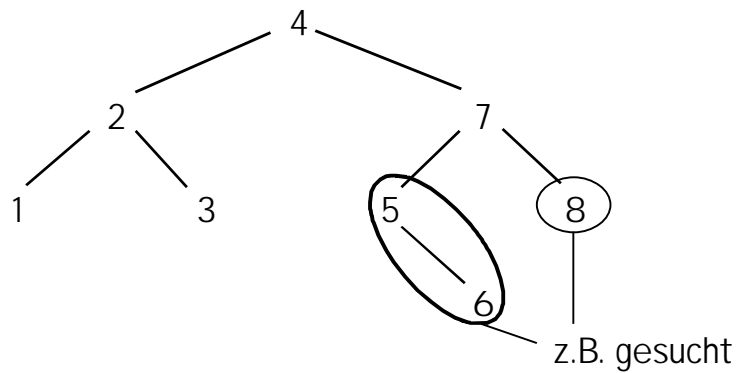
• Einfügen neuer Elemente in B:

(Nassi – Schneidemann Diagramm:)



(*) = Rekursion

Beispielfolge: 4, 7, 5, 6, 2, 8, 3, 1



Algorithmen und Programme

Suche Knoten mit $x = k$ in B

```

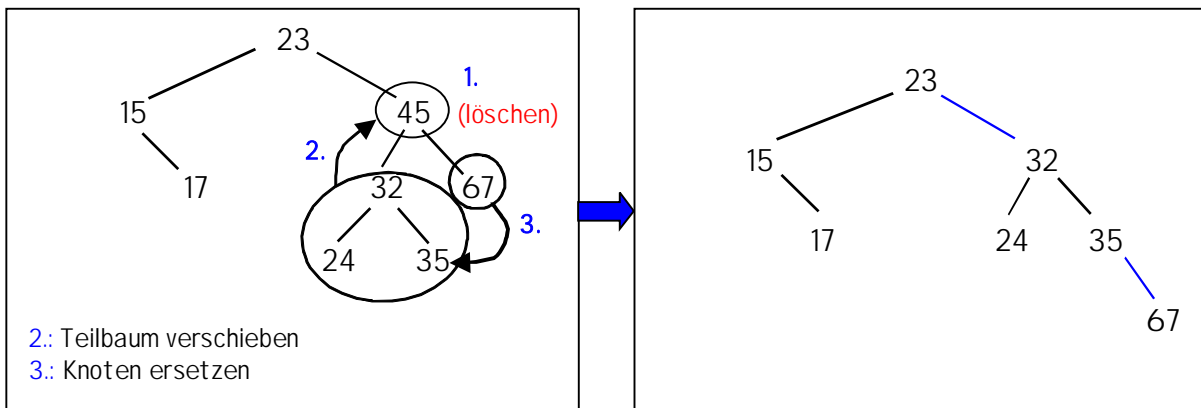
knoten *suche (int k, knoten *wurzel)
{
  if (wurzel == 0) return (0);           // leerer Baum – Knoten k nicht gefunden
  else if (k < wurzel --> x) return (suche (k, wurzel --> links)); // Rekursion
  else if (k > wurzel --> x) return (suche (k, wurzel --> rechts)); // Rekursion
  else return (wurzel);                 // Knoten mit Wert k gefunden
}
    
```

➡ Zeiger auf Knotenelement mit $x = k$ wird von 'suche' zurückgegeben (falls vorhanden)

• Löschen von Knoten mit $x = k$:

- 1) Element mit $x = k$ suchen
- 2) Element entfernen
- 3) Teilbäume neu verzweigen

Beispiel:



Algorithmen und Programme

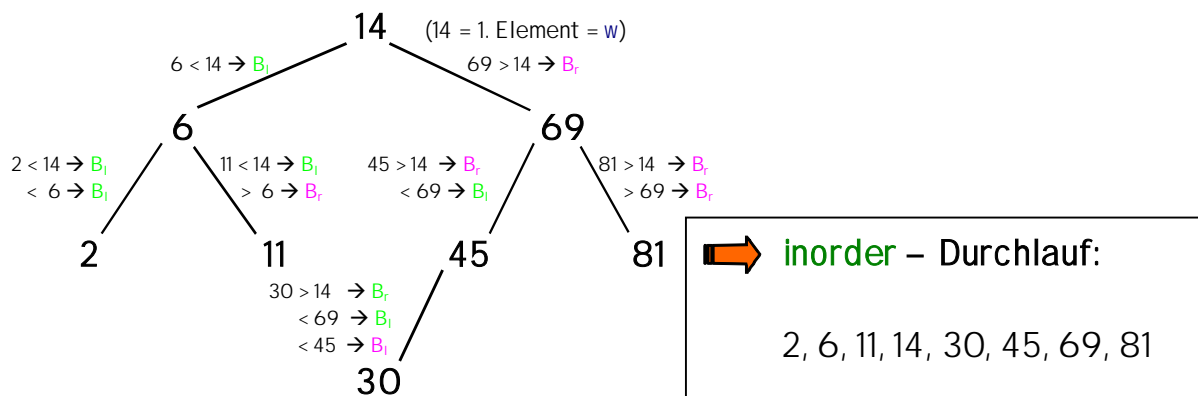
Aufwand für Suchen, Einfügen, Löschen:

- **günstiger Fall:**
 binärer Suchbaum mit n Elementen hat Höhe $\lceil \lg(n+1) \rceil \longrightarrow O(\log(n))$
 (nach oben gerundet)
- **schlechtester Fall:**
 Baum entartet zu linearer Liste (z.B. Aufbau eines Suchbaumes mit monotoner Folge) $\longrightarrow O(n)$
- **im Mittel:**
 $O(\log(n)) + 40\%$

Mit binären Suchbäumen lassen sich Zahlenfolgen effizient sortieren:

- Schritt 1:** Zahlen schrittweise in den (zu Beginn leeren) Baum einfügen
- Schritt 2:** Baum in 'inorder' – Reihenfolge durchlaufen und Werte ausgeben

Beispiel: (zu sortierende Zahlenfolge:) 14, 69, 45, 6, 2, 81, 30, 11



Algorithmen und Programme

Aufwand für Schritt 1:

- Einfügen eines Elementes $O(\log(n))$
- Einfügen von n Elementen $O(n \log(n))$

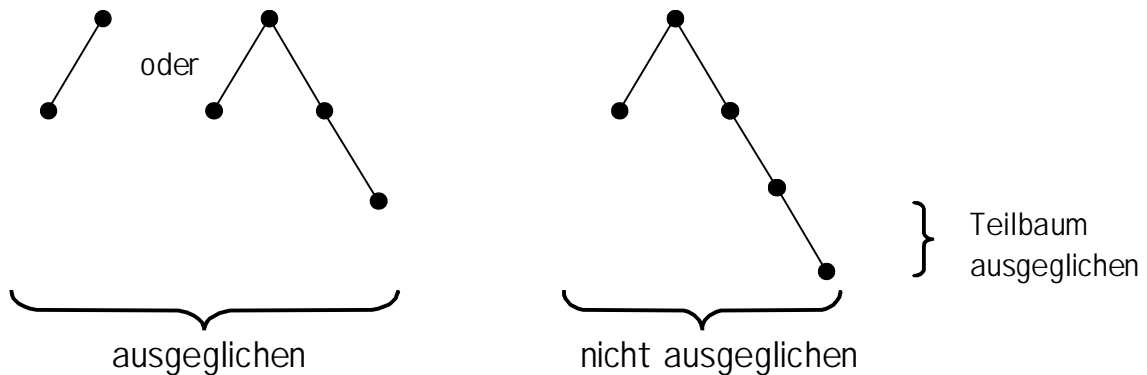
Aufwand für Schritt 2:

- jedes Element wird einmal ausgegeben $O(n)$

Gesamtaufwand:

$O(n \log(n))$

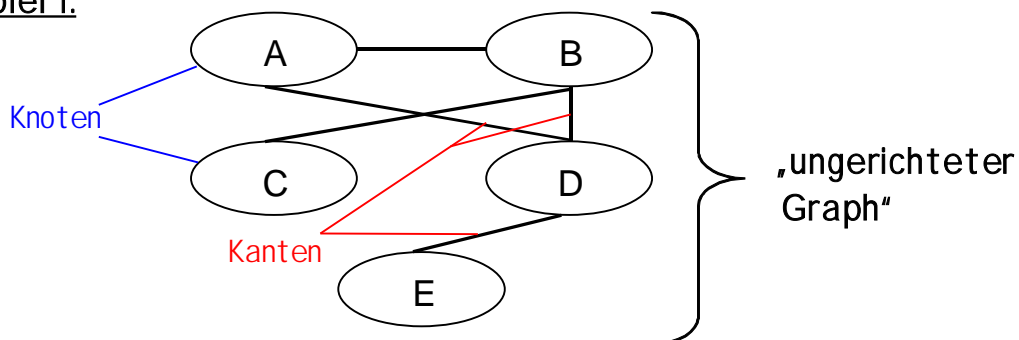
Ein Baum heißt **ausgeglichen** (bzw. ALV – Baum), wenn sich für jeden Knoten die Höhe der zugehörigen Teilbäume um **höchstens 1** unterscheidet!



5.5 Graphen

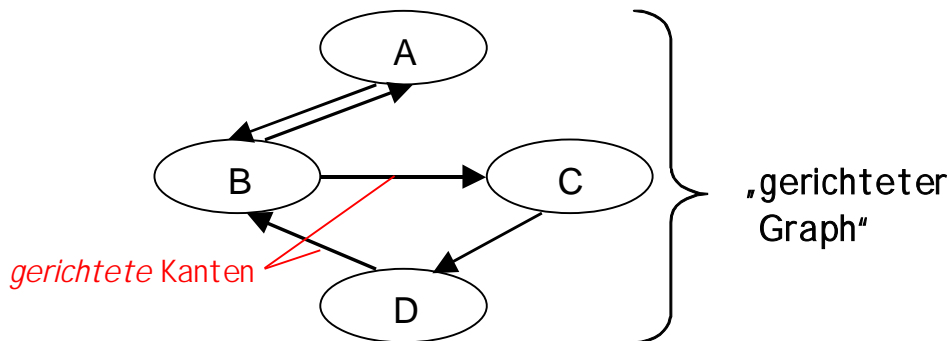
Allgemeines Modell zur Darstellung virtueller Strukturen.

Beispiel 1:



Algorithmen und Programme

Beispiel 2:



Anwendungsbeispiele:

- Straßenverbindungen
- Kommunikationsnetzwerke (Telefon, Internet)
- Zustandsübergänge technischer Systeme
- Molekülstrukturen
- elektrische Schaltungen, ...

- ➡ Ein gerichteter / ungerichteter Graph $G = (V, E)$ besteht aus
- einer Menge V von Knoten (*vertices*)
 - einer Menge E von Kanten (*edges*)

Eine Menge $e \in E$ ist ein geordnetes / ungeordnetes Paar $e = (v, w)$ von Knoten mit $v, w \in V$. **Man sagt:** v und w sind „adjazent“.

Bei ungerichteten Graphen: $(v, w) = (w, v)$

zu Beispiel 1 (ungerichteter Graph):

$$V = \{ A, B, C, D, E \}$$

$$E = \{ (A, B), (A, D), (B, C), (B, D), (D, E) \}$$

zu Beispiel 2 (gerichteter Graph):

$$V = \{ A, B, C, D \}$$

$$E = \{ (A, B), (B, A), (B, C), (C, D), (D, B) \}$$

Algorithmen und Programme

Datenstruktur zur Darstellung von Graphen:

1) „Adjazenz – Matrix“ (zu Beispiel 2; s.o.)

Kante	zu Knoten j				
	A	B	C	D	
von Knoten i	A	0	1	0	0
	B	1	0	1	0
	C	0	0	0	1
	D	0	1	0	0

Sei n Anzahl der Knoten eines Graphen:

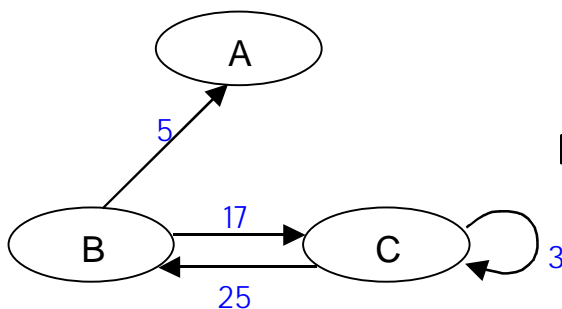
Adjazenzmatrix zu G ist eine $n \times n$ - Matrix $A = (a_{ij})$ mit

$$a_{ij} \begin{cases} 1: \text{ falls es Kante von } i \text{ nach } j \text{ gibt} \\ 0: \text{ sonst} \end{cases}$$

Bei ungerichteten Graphen:

Adjazenzmatrix symmetrisch [(A, B), (B, A), (A, D), (D, A), ...]

Attribuierte (markierte) Graphen:



	A	B	C
A	-1	-1	-1
B	5	-1	25
C	-1	17	3

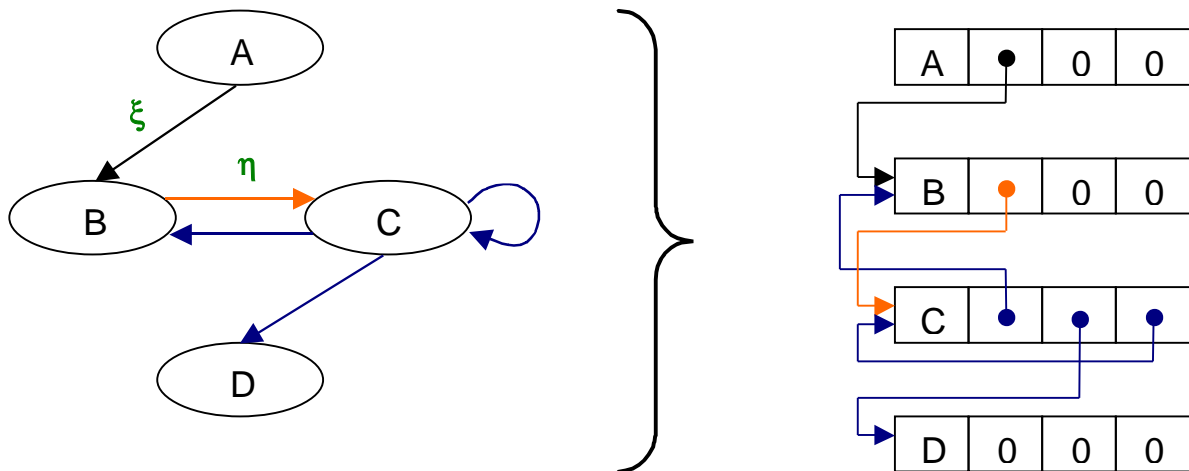
-1 markiert hier, daß **keine** Kante existiert

- Speicherbedarf der Adjazenzmatrix bei n Knoten: $O(n^2)$
- Adjazenzmatrix ist eine *statische* Datenstruktur
- ineffizient bei vielen Knoten und wenig Kanten

2) „Verzeigerte Knoten“

Ähnlich wie bei Listen / Bäumen: jeder Knoten hat mehrere Zeiger auf benachbarte Knoten.

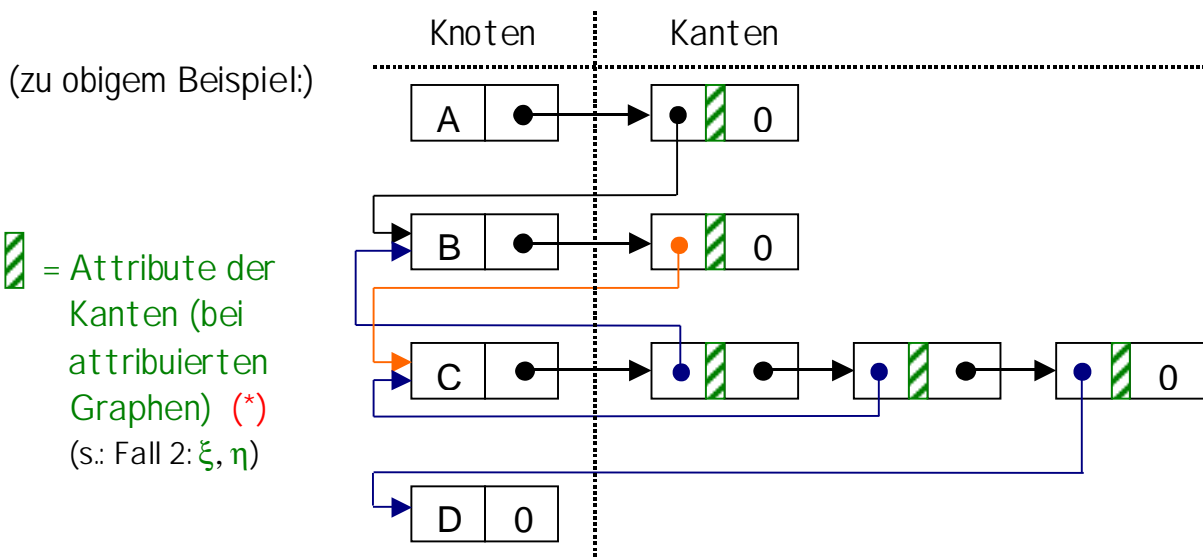
Algorithmen und Programme



Problem: Die maximale Anzahl der Zeiger (ausgehende Kanten pro Knoten) muß vorab festgelegt werden. (Im Extremfall: n Zeiger)

3) „Adjazenz - Listen“

Für jeden Knoten wird eine Liste der ausgehenden Knoten verwaltet.



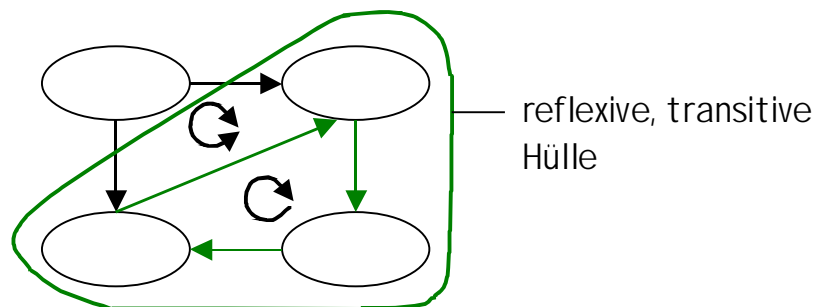
- Knoten werden als Array oder Liste verwaltet
- Attribuierung der Kanten: **Erweiterung** der 'Kanten' – Datenstruktur (*)

Algorithmen und Programme

Zwei wichtige Eigenschaften von Graphen:

- 1) Erreichbarkeit von Knoten in einem **gerichteten** Graphen:
 die **reflexive, transitive** Hülle eines Graphen $G = (V, E)$ ist der Teilgraph $G^* = (V, E^*)$, für den gilt:
 für alle Knoten $v, w \in E^*$ gibt es einen gerichteten Weg **zurück** von w nach v

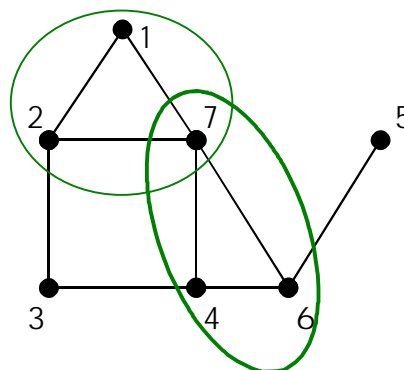
Beispiel:



- 2) „**Cliquen**“ in **ungerichteten** Graphen:
 Sei $G = (V, E)$; es gibt $V' \subseteq V$ mit k Elementen, so daß $\forall v, w \in V' (v, w) \in E$
 (jeder Knoten mit jedem anderen der $k-1$ Knoten verbunden)

Beispiel:

zwei 3 - Cliquen:
 $\{1, 2, 7\}, \{4, 6, 7\}$



Algorithmen und Programme

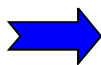
5.6 Adreßberechnung (Hash – Verfahren)

Verfahren

- zur dynamischen Speicherung von Datenelementen (z.B.: Matrikelnummern + Namen)
- zum *schnellen / gezielten* Zugriff auf Datenelemente (z.B.: „gib Namen zu Matrikelnummer 2345678 aus“)

Realisierung mit Array:

0	---
1	---
.	
.	
.	
2345678	Kurt Mayer
.	
.	
.	
2346221	Erwin Loch
.	
.	
.	



schneller Zugriff, aber:

↪ **sehr speicheraufwendig!**

bisherige Alternativen:

- Listen, (binäre Such-) Bäume
- ➔ mehrere, evtl. alle Elemente müssen durchsucht werden, bis das gewünschte Element gefunden ist

↪ **langsam!**

Hashing:

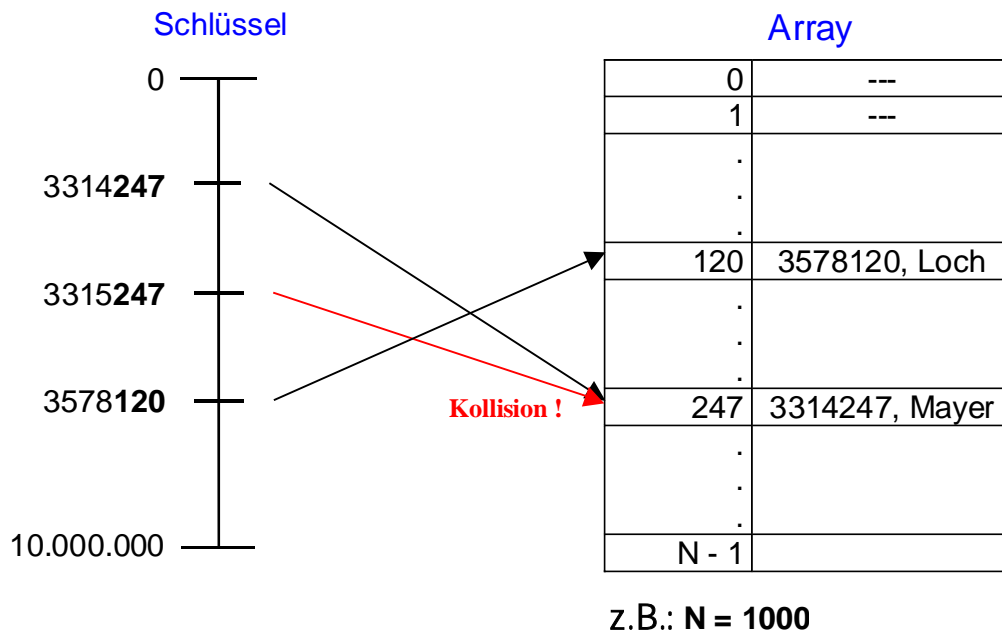
- Speicher wird als Array $a[N]$ bereitgestellt
- Zugriff erfolgt über **Schlüssel x** (z.B.: Matrikelnummer) mit einem umfangreichen Schlüsselwertebereich **X** mit üblicherweise $N \ll |X|$ (z.B.: Matrikelnummer 0 bis 10.000.000)
- Die Adresse bzw. der **Index i** , an dem ein Datenelement zu speichern ist, wird aus dem Schlüssel mit Hilfe einer **Hashfunktion $h(x)$** berechnet:

$$i = h(x)$$

(z.B.: $i = x \% N$)

Algorithmen und Programme

- Die Hashfunktion ist **nicht bijektiv**, d.h. verschiedene Schlüssel können zum selben Arrayelement führen → **Kollision**



Beispiele für Hashfunktionen:

1) $h(x) = x \% N$

- 2) x_1, x_2 seien Ordinalzahlen der ersten beiden Buchstaben des zu speichernden Namens:

$h(x_1, x_2) = x_1 + x_2$ → **Nachteil:** $h(x_1, x_2) = h(x_2, x_1)$

3) $h(x_1, x_2) = [347 * x_1 + x_2] \% 1147$

Beachte:

- Ergebnis $h(x)$ muß innerhalb des Array – Indexbereiches 0 ... N - 1 liegen
- bei zufälligem Schlüssel sollten **alle** Adressen möglichst *gleichwahrscheinlich* sein

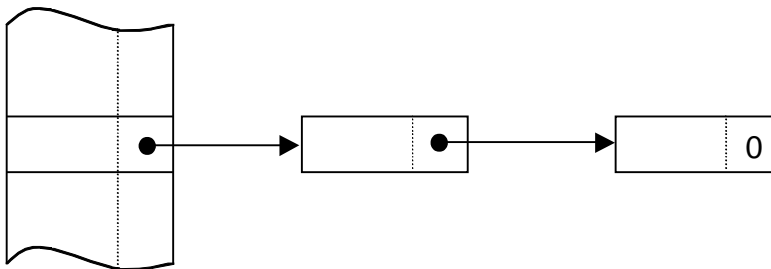
Algorithmen und Programme

Kollisionsbehandlung:

Frage: „Wohin soll ein Element geschrieben werden, wenn sein Arrayplatz schon von einem anderen Element belegt ist?“

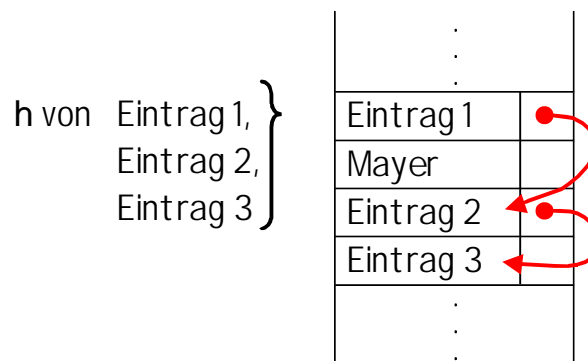
1) lineare Verkettung:

Erweiterung des Arrays durch lineare Listen



2) verschmolzene Ketten:

Extreme Listen aus 1) werden in Array „verschmolzen“, d.h. Speicherung auf dem nächsten freien Speicherplatz



3) offene Adressierung:

- keine expliziten Zeiger
- jedes Element wird über Sondierungspfad $\alpha_0 = h(x); \alpha_1; \alpha_2; \dots$ erreicht

a) lineares Sondieren:

$$\alpha_0 = h(x)$$

$$\alpha_{i+1} = (\alpha_i + 1) \% N$$

Algorithmen und Programme

b) doppeltes Hashing:

$$\alpha_0 = h(x), \quad \delta = g(x), \quad \alpha_{i+1} = (\alpha_i + \delta) \% N$$

mit $g(x)$: weitere Hashfunktion

➡ Bei 80prozentiger Belegung von $a [N]$ ist im Schnitt mit höchstens drei Kollisionen zu rechnen.

Bei Belegung $> 80\%$: starker Anstieg der Kollisionen.

Suchen / Einfügen / Löschen mit konstantem Zeitaufwand $O(1)$; unabhängig von $N!$

Vergleich zu anderen Datenstrukturen:

	lineare Listen	Bäume	Hashing
gegebenes Element finden	$O(n)$	$O(\log n)$	$O(1)$
Nachfolger finden	$O(1)$	$O(\log n)$	$O(n)$
Element einfügen oder löschen	$O(n)$	$O(\log n)$	$O(1)$

6. Abstrakte Datentypen und objektorientierte Programmierung

6.1 Abstrakte Datentypen (ADT)

ADT faßt Datenstruktur und zugehörige Operationen zu einer Einheit zusammen.

vgl. Beispiel (Übung / Aufgabe 12: „Stapelrechner“):

Stapel kann als ADT aufgefaßt werden, d.h. als Datenstruktur zum Speichern von Elementen p l u s Operationen

↘ initialize, push, pop

Algorithmen und Programme

Daten und Operationen werden als zusammengehörig betrachtet!

➔ **Wichtig** hierbei:

- Operationen und ihre Semantik (= *Wirkung* auf die Daten)
- Definition der „Schnittstelle“ (Eingabe-/ Ausgabeparameter)
- ➔ Mehr muß nicht bekannt sein, um mit ADT zu arbeiten (z.B.: Stapel als Stapelrechner)

➔ **Unwichtig** (vor dem Anwender bzw. vor dem ausführenden Programm *versteckt*):

- interne Verwaltung der Daten (z.B.: Array oder Liste)
- konkrete Implementierung der Operationen (meist mehrere Möglichkeiten)

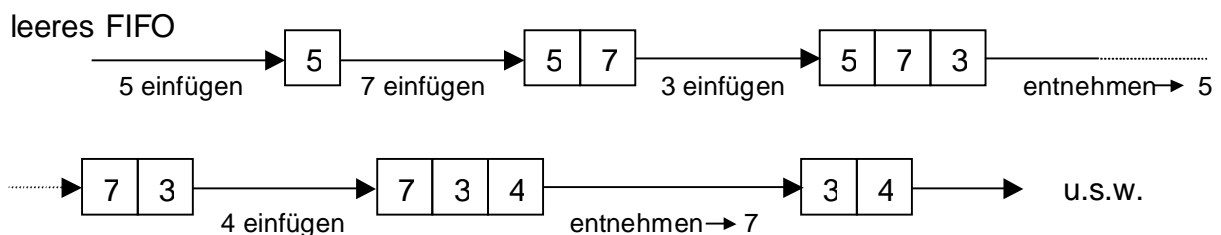
Vorteile dieser „Kapselung“:

- weitgehend unabhängige Programmkomponenten („Modularisierung“; vgl. Abschnitt: 2.4)
- Übersichtlichkeit (klare Aufgabenteilung, klare Schnittstellen)
- bessere Wartbarkeit (Fehler lassen sich z.B. auf einzelne Komponenten eingrenzen, Komponenten können unabhängig von anderen getestet und ausgetauscht werden)

6.2 ADT – Beispiel: „Warteschlange“

(Queue, FIFO = first in first out)

Prinzip:



Algorithmen und Programme

Realisierung von Warteschlangen mit FIFOs, z.B. zur Datenübertragung zwischen Programmen oder zwischen verschiedenen Geräten (z.B.: Rechner → Drucker); typisch: „Produzent“ fügt Daten in FIFO ein, „Konsument“ liest Daten aus.

Realisierung des FIFO als ADT:

1) Festlegung der benötigten Operationen

- Initialisierung des FIFO → 'initialize' → lösche Eintrag des FIFO
- Hinzufügen eines Elementes am Ende des FIFO
- Entnahme eines Elementes am Anfang des FIFO
- Abfrage, ob FIFO leer

2) Festlegung der Schnittstellen:

Namen der Operationen, zu übergebende und zurückgelieferte Parameter



Funktionsköpfe in C:

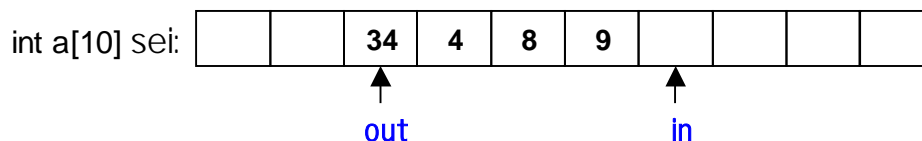
```
initialize (struct FIFO *fifo) // initialisieren
enter (struct FIFO *fifo, int element) // einfuegen
int remove (struct FIFO *fifo) // Element auslesen
int empty (struct FIFO *fifo) // gibt 1 zurück, wenn FIFO leer; 0 sonst
```

struct FIFO sei eine Datenstruktur, welche alle Daten des FIFO enthält.

3) Implementierung

- Definition der Datenstruktur struct FIFO
- Implementierung der zugehörigen C-Funktion

Implementierung mit Array:



Algorithmen und Programme

```

int remove (FIFO *fifo)
{
  int x;
  if (empty(fifo))
    cout<<"Fehler: FIFO ist leer!\n";
  else
  {
    x = fifo --> a[fifo --> out];
    fifo --> out = (fifo --> out + 1)%100;
    return (x);
  }
}

```

6.3 Objektorientierte Programmierung - Einführung

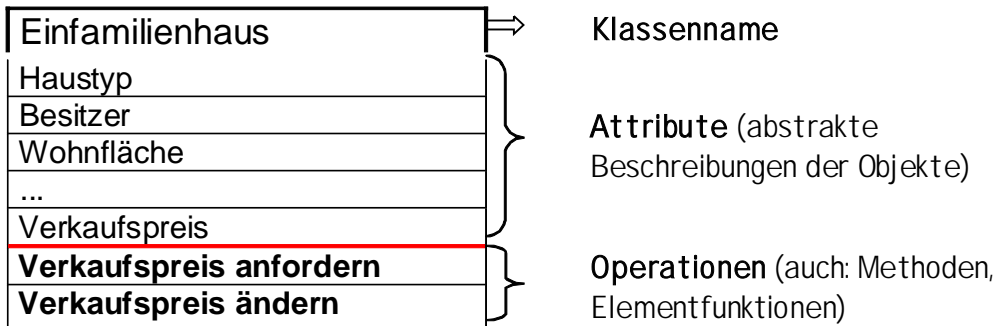
- **Objekte** sind Einheiten der realen oder gedachten Welt mit bestimmten Eigenschaften und Verhalten; Objekte fassen Daten und auf ihnen ausführbare Funktionen zusammen
- **Klassen** sind Beschreibungen von Objekten bzw. Abstraktionen von ähnlichen Eigenschaften und Verhaltensweisen ähnlicher Objekte
- Ein Objekt ist die konkrete Ausprägung („Instanz“) einer Klasse. Der **Zustand** eines Objektes kann durch Operationen, die auf Objektdaten ausgeführt werden, geändert werden
- **Objektorientierte Programmierung (OOP)** greift damit auf *ADT – Konzept* zurück:

Klasse	Datentyp + anwendbare Operationen
Objekt	(konkrete) Variable (→ belegt Speicher)

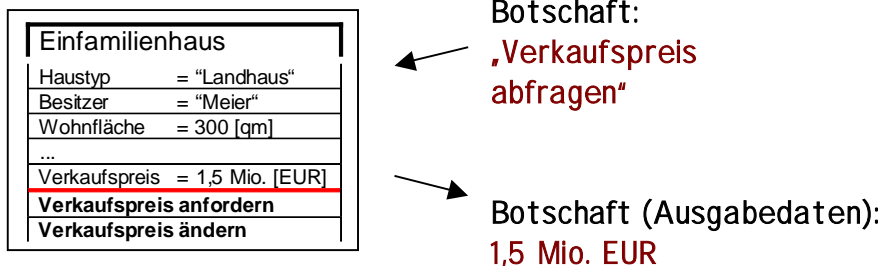
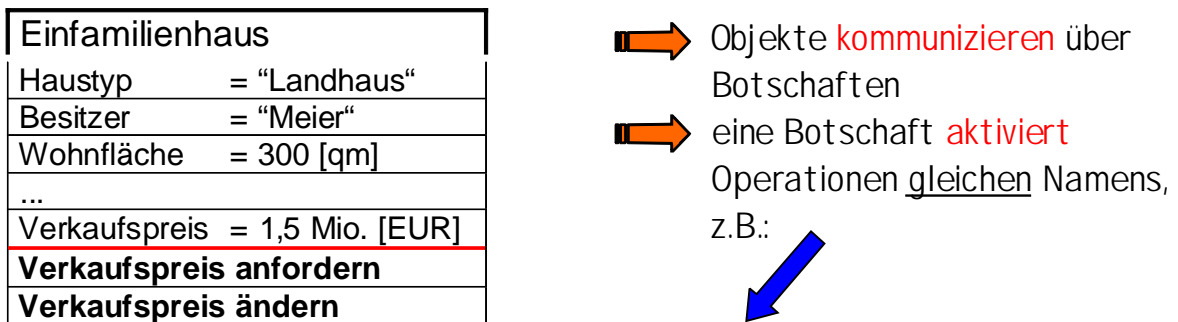
Algorithmen und Programme

Allgemeines Beispiel: „Immobilienmakler“

Klasse Einfamilienhaus:



Ein konkretes Objekt (Instanz) der Klasse Einfamilienhaus:



Algorithmen und Programme

Klassendeklaration in C++:

Beispiel: „Arbeiten mit Brüchen“

```

class Bruch // Klassenkopf mit Klassenname
{
  public:    // Schnittstelle nach aussen
    Bruch (int z=0, int n=1);    // Konstruktor
    ~ Bruch ();                // Destruktor
    Bruch Multiplikation (Bruch &b);
    bool Kehrwert (Bruch &b);
    void Drucke_Bruch ( )const;
  private: // Kapsel; nur intern verfügbbar
    int zaehler, nenner;        // interne Variable(n)
    bool Pruefe_Zaehler ( );    // Hilfsfunktion
};

```

← public – Teil definiert Schnittstelle nach außen

← private – Teil nur für Elementfunktion der Klasse selbst erreichbar = **Kapsel**

→ **Definition von Objekten einer Klasse:** Bruch a;

→ **Definition der Elementfunktion**

Prinzip:

- Elementfunktion in Klasse **deklarieren**
- und außerhalb der Klasse **definieren**

Bei Aufruf einer Elementfunktion für konkretes Objekt der Klasse werden für die Elementfunktionen die Adressen der Datenelemente dieses Objektes mit internem „**this – Zeiger**“ bekannt gemacht; mit „**::**“ **als Bereichsoperator**.

```

Bruch Bruch :: Multiplikation (Bruch &b)
{
  Bruch ergebnis;    // neues Objekt der Klasse Bruch
  ergebnis.zaehler = zaehler * b.zaehler;
  ergebnis.nenner = nenner * b.nenner;
  return ergebnis;
}

```

Algorithmen und Programme

Ausgabe von Zähler und Nenner eines impliziten Bruchs (s.o.):

```
void Bruch : : Drucke_Bruch ( ) const
{
    cout<<zaehler <<" / " <<nenner;
}
```

Zählerprüfung:

```
bool Bruch : : Pruefe_Zaehler ( )
{
    if (zaehler == 0) return true;
    else return false;
}
```

Kehrwert:

```
bool Bruch : : Kehrwert (Bruch &b)
{
    if (! Pruefe_Zaehler ( ) )
    {
        b.zaehler = nenner;
        b.nenner = zaehler;
        return true;
    }
    return false;
}
```

Definieren und Verändern von Objekten der Klasse Bruch im Anwendungsprogramm:

Bruch a, b, c; 🏹

Erzeugen dreier Objekte („Objektvariablen“) mit Initialwert **NULL**

a.Kehrwert (c); 🏹

„Kehrwert“ operiert auf Objekt a und ersetzt c durch Kehrwert von a

c = a.Multiplikation (b); 🏹

Bruch in c wird durch Produkte der Zähler und Nenner von a und b ersetzt

NICHT ERLAUBT:

```
int x;
x = a.zaehler;
a.nenner = 10;
```

} Kein direkter Zugriff der Anwendung auf Elementdaten!

Algorithmen und Programme

→ Deklaration des Konstruktors 'Bruch':

```
Bruch :: Bruch (int z, int n)
{
  zaehler = z;
  nenner = n;
}
```

Vorbelegung in Klassendeklaration mit (z.B.) z = 0 und n = 1

- mögliche Definition von Objekten:

```
Bruch obj1;           // → 0/1
Bruch obj2 (7, 13);  // → 7/13
Bruch obj3 (5);      // → 5/1
```

Dynamische Speicherzuordnung für Objekte mit *new* – Operator:

```
Bruch *obj = new Bruch;
```

↳ Zugriff auf Elementdaten des Objektes obj. mit Pfeiloperator --> ,
z.B.: obj --> Kehrwert (c);

↳ Objektfelder:

```
int n = 10;
Bruch *objects = new Bruch [n];

Kehrwert des dritten Bruchs:
(objects + 2) --> Kehrwert (c);
```

↳ Freigeben dynamisch angelegten Speichers:

```
delete obj;
delete [ ] objects;
```

↳ konstante Objekte:

```
const Bruch eins (1); // Bruch hat den Wert 1
```

Algorithmen und Programme

<pre>a.Kehrwert(c); // c = 1 / a c = a.Multiplikation(b); // c = a * b</pre>	} Unschöne Notation!
--	----------------------

➔ Lösung in C++: [Überladen von Operatoren](#)

- **Operator**, den man auf verschiedene Datentypen (Objekte) anwenden kann, wird als **überladen** bezeichnet

↪ In C++ kann der Anwender Operatoren für eigene Datentypen (Objekte) überladen, z.B. für Klasse mit Objekten a, b, c:

<pre>~a; // Kehrwert des Bruchs a c = a*b; // Multiplikation zweier Brueche cout<<a <<"\n"; // Ausgabe des Bruchs</pre>
--

Implementierung von ~ und * für Klasse Bruch:

```
void Bruch : : operator ~ ( )
{
    if( ! Pruefe_Zaehler ( ) )
    {
        int h = zaehler;           // Hilfsvariable
        zaehler = nenner;
        nenner = h;
    }
}
Bruch Bruch : : operator *(Bruch &b)
{
    return Bruch( zaehler * b.zaehler, nenner * b.nenner );
}
```

Im **public** – Teil der Klassendefinition:

<pre>void operator ~ (); Bruch operator *(Bruch &b);</pre>

Algorithmen und Programme

Ableitung von Klassen > Vererbung

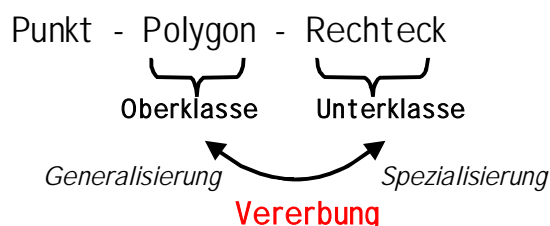
(wichtiges Konzept für *Wiederverwendbarkeit* des Programmcodes)

Vererbungsmechanismus:

- gemeinsame Eigenschaften einer Menge von Objekten können als verallgemeinertes Konzept betrachtet werden → **Oberklasse**
- weitere Eigenschaften dieser Objekte sind unterschiedlich; sie bilden **Unterklassen** und „erben“ von Oberklassen
- Oberklassen sind also Abstraktionen oder *Generalisierungen* von ähnlichen Eigenschaften und Verhaltensweisen ihrer Unterklassen. Unterklassen sind *Spezialisierung(en)* ihrer Oberklasse(n)
- Wenn die Oberklasse bekannt ist, brauchen in der Unterklasse nur **Abweichungen** beschrieben werden. Alles andere kann wiederverwendet werden, weil es in der Oberklasse bereits vorliegt.
- **Methoden** eines Objektes setzen sich aus eigenen Methoden **plus** Menge der Methoden der Oberklasse(n) zusammen
- Vererbung ist **hierarchisch** organisiert
- Syntax der Vererbung:

```
class <Klassenname>: public <Oberklassenname>
```

Beispiel:



```
class Punkt
{
  public:
    Punkt( int x_wert = 0, int y_wert = 0 );           // Konstruktor
    double Abstand( Punkt &p );
  private:
    int x, y;
  friend void Polygon : : Translation( int dx, int dy ); // friend: siehe nächste Seite
};
```

Algorithmen und Programme

„friend“ – Konstrukt bricht Kapsel für „Freunde“ auf.

Hier: Elementfunktion Translation der Klasse Polygon hat Zugriff auf private-Elemente der Klasse Punkt.

```
Punkt : : Punkt( int x_wert, int y_wert )
{
  x = x_wert; y = y_wert;
}
double Punkt : : Abstand( Punkt &p )
{
  return sqrt( (x - p.x) * (x - p.x) + (y - p.y) * (y - p.y) );
}
```



sqrt (= Quadratwurzel → squareroot) erfordert Datei math.h:

```
# include < math.h >
```

(Präprozessoroperation # include fügt Datei math.h ins Programm ein)

```
class Polygon
{
  public:
  Polygon( int n, Punkt *p );
  double Umfang ( );
  void Translation( int dx, int dy );
  protected:
  int Ecken_Anzahl;
  Punkt *Punkte;
};
```

→ **protected**: Elemente und Methoden sind nur in der eigenen und in allen public – abgeleiteten Klassen zugreifbar.

Algorithmen und Programme

```

Polygon :: Polygon( int n, Punkt *p)
{
  Ecken_Anzahl = n;
  Punkte = new Punkte[ n ];
  int i = 0;
  while( i < n )
  {
    Punkte[ i ] = p[ i ];      // Eckpunkte uebernehmen
    i = i++;
  }
};

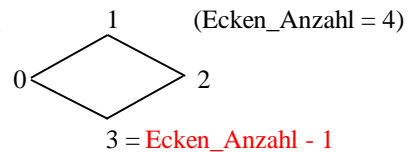
```

```

double Polygon :: Umfang ( )
{
  double sum = 0; int i =0;
  while( i < Ecken_Anzahl - 1 )
  {
    sum = sum + Punkte[ i ].Abstand( Punkte[ i + 1 ] );
    i = i++;
  }
  sum = sum + Punkte[ Ecken_Anzahl - 1 ].Abstand( Punkte[ 0 ] );
  return sum;
}

```

Skizze:



```

void Polygon :: Translation( int dx, int dy )
{
  int i = 0;
  while( i < Ecken_Anzahl )
  {
    Punkte[ i ].x = Punkte[ i ].x + dx;
    Punkte[ i ].y = Punkte[ i ].y + dy;
    i = i++;
  }
}

```



Translation hat als „Freund“ der Klasse Punkt direkten Zugriff auf die Koordinaten der Klasse Punkt.

Algorithmen und Programme

```
class Rechteck : public Polygon // erben
{
  public:
  Rechteck( Punkt *p );
  double Umfang ( );           // „ueberschreibt“ Umfang der Klasse Polygon
  double Flaeche ( );
  private:
  double a, b;                 // Seitenlaengen
};

Rechteck :: Rechteck( Punkt *p )
{
  this --> : : Polygon( 4, P );
  a = Punkte[ 0 ].Abstand( Punkte[ 1 ] );
  b = Punkte[ 1 ].Abstand( Punkte[ 2 ] );
}

double Rechteck :: Umfang ( )
{
  return 2*( a + b );
}

double Rechteck :: Flaeche ( )
{
  return a * b;
}
```

➡ Weitere wichtige Konzepte der OOP:

- Polymorphismus
- Templates
- Mehrfachvererbung, u.s.w.



Zur Vertiefung bzw. Übung:
Literatur + eigenständiges Programmieren!!!

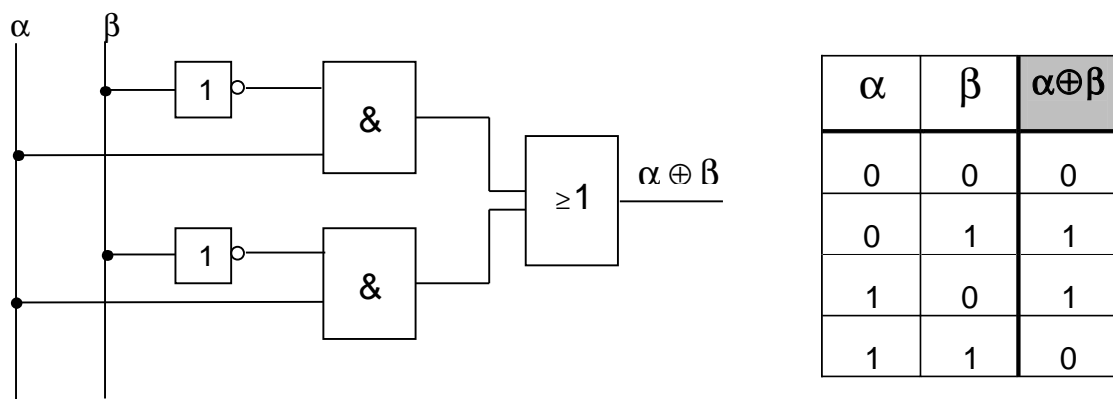
Algorithmen und Programme

7 Einige theoretische Aspekte

7.1 Schaltnetze, Automaten, Maschinenmodelle

Schaltnetze bilden (binäre) Eingangsgrößen in (binäre) Ausgangsgrößen ab.

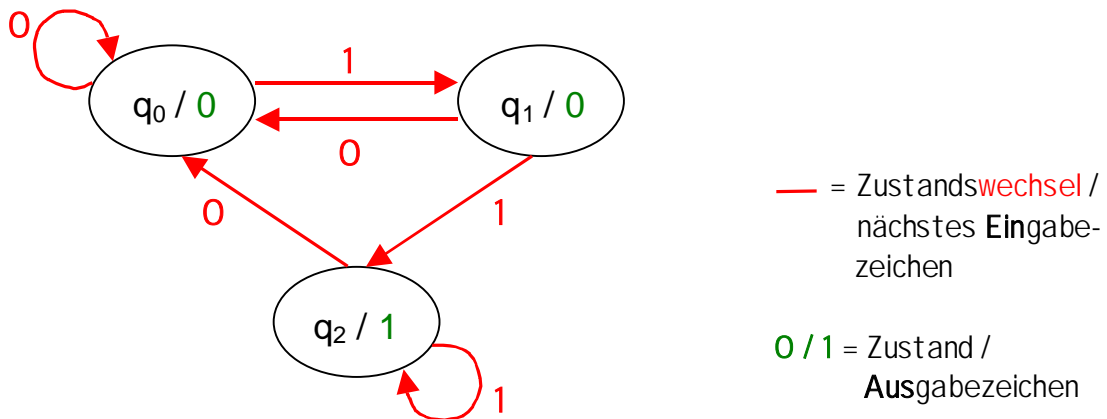
Beispiel: modulo 2 $\longrightarrow \oplus$



Ergebnis unabhängig von früheren Zuständen!

Automaten bilden Eingangsgrößen und frühere Zustände in Ausgangsgrößen ab.

Beispiel: Finden zweier aufeinanderfolgender 1-en in beliebigen Wörtern über Alphabet $A = \{0, 1\}$, d.h. in $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$:



Algorithmen und Programme

Binärzeichen werden sequentiell gelesen und verändern den Zustand des Automaten

$$\delta: \{q_0, q_1, q_2\} \times \{0, 1\} \longrightarrow \{q_0, q_1, q_2\}$$

➤ Ausgabe hängt in diesem Beispiel **direkt** vom Zustand ab (*Moore – Automat*)

$$\beta: \{q_0, q_1, q_2\} \longrightarrow \{0, 1\}$$

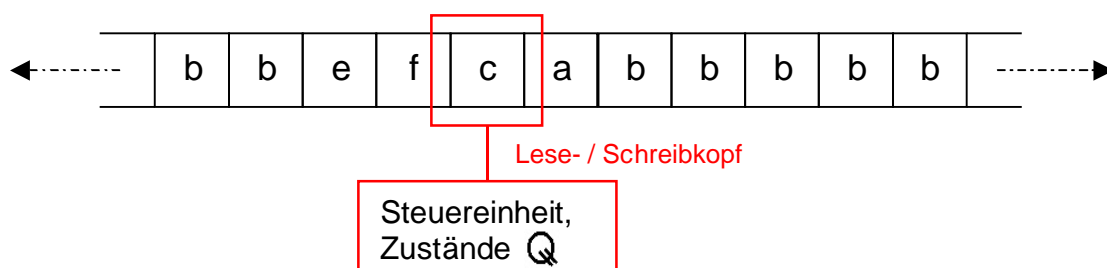
realisierte Funktion:

$$f: \{0, 1\}^* \longrightarrow \{0, 1\} \text{ mit } f(w) = \begin{cases} 1, & \text{falls } w = w'11, w' \in \{0, 1\}^* \\ 0, & \text{sonst} \end{cases}$$

➡ Automaten haben beschränkte Fähigkeiten, können z.B. nicht das Produkt zweier Dualzahlen bilden

Allgemeine Maschinenmodelle zur Ausführung algorithmischer Verfahren / Berechnung von Funktionen

Beispiel: Turingmaschine



- Algorithmus + Eingabe steht auf Band
- Lese- / Schreibkopf steht am Anfang auf erster nichtleerer Bandstelle; Steuereinheit befinde sich im Zustand q_0
- Maschine liest Eingabe, wechselt in Abhängigkeit von Eingabe und Zustand den Zustand und schreibt neues Zeichen auf Band oder bewegt den Kopf

Algorithmen und Programme

Komplexe Maschinenmodelle:

z.B.: verallgemeinerte Registermaschine



Bemerkungen:

- „Abstrakte Rechner“ (Maschinenmodelle) zur mathematischen Untersuchung von Algorithmen und ihren Eigenschaften
- jeder intuitive Algorithmus, der Abbildung $f: I^* \rightarrow O^*$ für Alphabete I und O beschreibt, lässt sich mit Turingmaschine berechnen
- Maschinenmodelle sind ineinander überführbar ↻

„Church’sche These“

7.2 Was ist berechenbar?

→ Antwort: „Fast nichts!“

Nachweis mit Hilfe der *Diagonalisierung* nach „Cantor“.

Sei $A = \{ a_1, \dots, a_n \}$ Alphabet; Menge aller Texte (Algorithmen) über A :

$$A^* = \{ \epsilon, a_1, \dots, a_n, a_{11}, a_{12}, \dots, a_{731983}, \dots \}$$



A ist abzählbar!

$\text{Texte} \subseteq \text{Algorithmen} \subseteq \text{Funktionen} \subseteq \text{einstellige Funktionen}$

Beispiel:

$$f: \mathbb{N} \rightarrow \mathbb{N}, \mathbb{N} = \{1, 2, \dots\}$$

Algorithmen und Programme

	x_0	x_1	x_2	...
f_0	$f_0(x_0) + 1$	$f_0(x_1)$	$f_0(x_2)$...
f_1	$f_1(x_0)$	$f_1(x_1) + 1$	$f_1(x_2)$...
f_2	$f_2(x_0)$	$f_2(x_1)$	$f_2(x_2) + 1$...
...	$\dots + 1$

} alle in A^* enthaltenen einstelligen Funktionen
(+1 = Trick nach Cantor)

Modifikation: $f_\xi(\xi) \longrightarrow f_\xi(\xi) + 1$

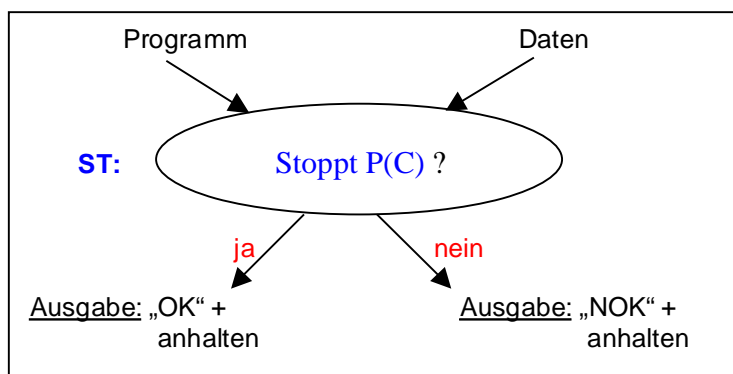
Diagonale bildet eine neue einstellige Funktion, die nicht in A^* enthalten ist. D.h., es gibt mehr (überabzählbar viele) Funktionen / Probleme als sie beschreibende Algorithmen / Programme.

quod erat demonstrandum

7.3 Halteproblem

Gibt es einen Algorithmus, der beschreibt, ob Programme (Automaten, Computer, ...) für gewisse oder alle Eingaben anhalten oder nicht ?

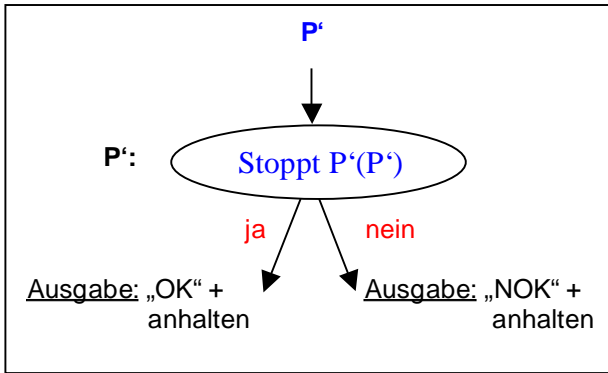
→ Annahme:
es gibt *Stop_Tester ST*:



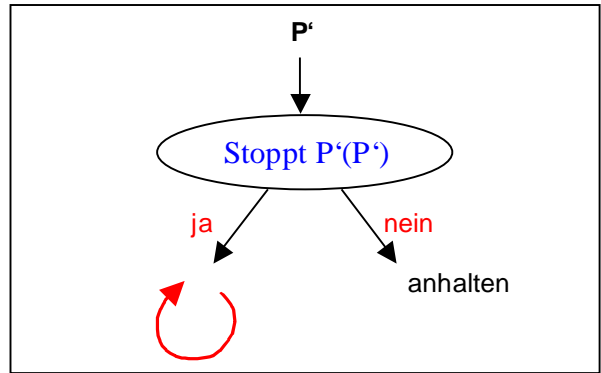
Algorithmen und Programme



Umbau von ST in `Stop_Tester_Neu P'`:

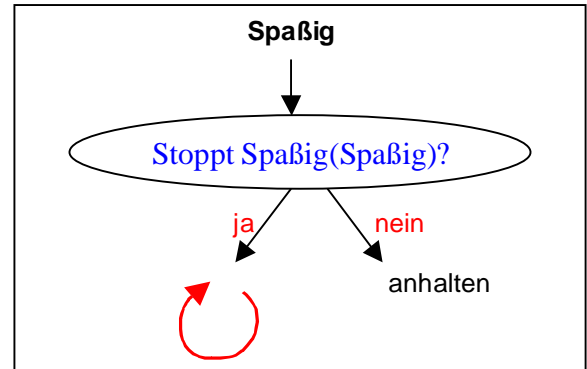


Hieraus entwickeln wir Algorithmus „Spaßig“:



```

Spaßig (char Programm [ ])
{
    ...
    if (Stop_Test_Neu (Programm) = "NOK"
        Stoppe;
    else Schleife endlos;
}
    
```



Zwei Möglichkeiten	<code>Stop_Tester_Neu</code>	<code>Spaßig</code>
„Spaßig“ <u>stoppt</u>	OK, anhalten	<u>Schleife endlos</u>
„Spaßig“ <u>stoppt nicht</u>	NOK, anhalten	<u>stoppt</u>

WIDERSPRUCH ➡ Annahme falsch!

D.h.: **Annahme**, daß `Stop_Tester_Neu` existiert, **ist falsch!**

➡ „Halteproblem“ allgemein **nicht** entscheidbar!

Algorithmen und Programme

Terminierungsuntersuchungen sind immer nur für Einzelfälle / bestimmte Klassen von Programmen möglich!

„Einfaches“, bisher ungelöstes Problem:

Gibt es Zahlen $n > 0$, für die folgender Algorithmus nicht terminiert?

```
f (int n)
{
  while (n != 1)
  {
    if ( n%2 == 0) n = n/2;    // gerade Zahlen halbieren
    else          n = 3*n + 1;
  }
}
```

Weitere interessante und wichtige Gebiete der Informatik:

- Formale Sprachen (mathematische Formulierung von Algorithmen)
 - Programmverifikation (Korrektheit von Programmen bzgl. formaler Spezifikation)
 - Komplexität
 - ...
-

Ende der Mitschrift